Міністерство освіти і науки України
Національна академія наук України
Національний центр «Мала академія наук України»

С. Б. МОГИЛЬНИЙ

# МАШИННЕ НАВЧАННЯ З ВИКОРИСТАННЯМ МІКРОКОМП'ЮТЕРІВ

Навчально-методичний посібник

КИЇВ
2019

Міністерство освіти і науки України
Національна академія наук України
Національний центр «Мала академія наук України»

**С. Б. МОГИЛЬНИЙ**

# МАШИННЕ НАВЧАННЯ З ВИКОРИСТАННЯМ МІКРОКОМП'ЮТЕРІВ

Навчально-методичний посібник

Київ
2019

Рекомендовано науково-методичною радою
Національного центру «Мала академія наук України»

**Могильний С. Б.**
Машинне навчання з використанням мікрокомп’ютерів: навч.-метод. посіб. / за ред. О. В. Лісового та ін. – К., 2019. – 224 с.

Посібник містить практичні приклади, які дають змогу вивчити основи фреймворку TensorFlow і його застосування в реальних конструкціях на мікрокомп’ютерах.

Посібник призначено слухачам Всеукраїнських наукових профільних шкіл – дослідникам-початківцям, а також викладачам, студентам і всім, хто проводить серйозні наукові дослідження.

© С. Б. Могильний, 2019
© Національний центр «Мала академія наук України», 2019

## ЗМІСТ


**Вступ. Штучний інтелект і машинне навчання ....................................................................5**

**1. Машинне навчання з TensorFlow...........................................................................................11**

Що таке TensorFlow ..............................................................................................................11
Архітектура TensorFlow........................................................................................................12
Компоненти TensorFlow .......................................................................................................13
Простий приклад TensorFlow...............................................................................................13
Завантаження даних у TensorFlow.......................................................................................15

**2. Встановлення TensorFlow на Windows...............................................................................18**

Версії TensorFlow ..................................................................................................................18
Встановлення Anaconda........................................................................................................18
Створення .yml-файла для встановлення TensorFlow і залежностей ................................21
Використання блокнота Jupyter ...........................................................................................24

**3. Встановлення TensorFlow на Raspberry Pi .........................................................................34**

Встановлення TensorFlow на RPi у віртуальному середовищі ..........................................34
Встановлення TensorFlow безпосередньо на ОС Raspbian................................................37
Встановлення класифікатора зображень на RPi..................................................................39

**4. Основи TensorFlow.................................................................................................................40**

Що таке тензор, представлення тензора .............................................................................40
Типи тензора..........................................................................................................................41
Створення тензора n-розмірності ........................................................................................42
Форма тензора .......................................................................................................................44
Типи даних.............................................................................................................................45
Створення оператора.............................................................................................................45
Змінні......................................................................................................................................47
Заповнювач ............................................................................................................................48
Сесія........................................................................................................................................48
Граф .......................................................................................................................................51

**5. Візуалізація графа з TensorBoard .........................................................................................53**

Що таке TensorBoard.............................................................................................................53
Як користуватися TensorBoard..............................................................................................55

**6. Pandas Python .........................................................................................................................59**

Що таке Pandas і чому його використовують.....................................................................59
Встановлення Pandas.............................................................................................................60
Кадр даних і серія..................................................................................................................60
Створення кадра даних .........................................................................................................61
Діапазон дат ...........................................................................................................................61
Перевірка даних.....................................................................................................................62
Фрагментація даних ..............................................................................................................63
Імпорт даних CSV за допомогою Pandas.read_csv()...........................................................66

**7. Лінійна регресія з TensorFlow ..............................................................................................67**

Тренування моделі лінійної регресії ...................................................................................68
Тренування моделі лінійної регресії за допомогою TensorFlow........................................71
Тренування з використанням Pandas....................................................................................72
Рішення за допомогою оцінювача Numpy ..........................................................................77
Рішення з TensorFlow............................................................................................................80

**8. Лінійна регресія для машинного навчання .........................................................................85**

Лінійна регресія TensorFlow із взаємодією..........................................................................85
Зведена статистика................................................................................................................87

Facets Overview для огляду набору даних....89
Facets Deep Dive для окремих фрагментів даних....89
Встановлення вебдодатка Facets....89
Overview для обчислення статистики....91
Побудова матриці кореляції....93
Використання Facets Deep Dive....96
API оцінювачів TensorFlow....98
Удосконалення моделі: врахування взаємодії....101

**9. Лінійний класифікатор в TensorFlow: бінарна класифікація....104**

Як працює бінарний класифікатор....104
Вимірювання продуктивності лінійного класифікатора....106
Лінійний класифікатор з TensorFlow....107

**10. Методи ядра в машинному навчанні: ядро Гаусса....124**

Для чого потрібні методи ядра....125
Що таке ядро в машинному навчанні....128
Типи методів ядра....129
Тренування класифікатора ядра Гаусса з TensorFlow....129

**11. Штучна нейронна мережа з TensorFlow....138**

Що таке штучна нейронна мережа....138
Нейромережева архітектура....139
Обмеження нейронної мережі....141
Приклад нейронної мережі в TensorFlow....142
Тренування нейронної мережі з TensorFlow....143

**12. Конволюційна нейронна мережа: тензорна класифікація зображень....147**

Архітектура конволюційної нейронної мережі....147
Компоненти конвнетів....148
Тренування CNN з TensorFlow....153

**13. Автоенкодер в глибокому навчанні: приклад TensorFlow....160**

Що таке автоенкодер....160
Приклад автоенкодера....161
Побудова автоенкодера з TensorFlow....161

**14. Рекурентна нейронна мережа: приклад з TensorFlow....171**

Що таке RNN....171
Побудова RNN для прогнозування часових рядів....177

**15. Проєкти машинного навчання на мікрокомп'ютерах....183**

Мікрокомп'ютери для машинного навчання....183
Movidius Neural Compute Stick 2 від Intel на Raspberry Pi 4....190
Запуск моделі Keras на Movidius Neural Compute Stick 2....198
Автономний робот-танк....202
Класифікатор зображень на Raspberry Pi з Intel Movidius....208
API для розпізнавання об'єктів з Raspberry Pi і TensorFlow....216

# Вступ. Штучний інтелект і машинне навчання

Штучний інтелект (ШІ) зі сфери наукової фантастики поступово став частиною нашого повсякдення. Це може бути програма, здатна обіграти професіонала, або розумний помічник Siri від Apple. Однак не кожен до кінця розуміє, що таке машинне навчання, чому сьогодні воно таке важливе і яке відношення воно має до ШІ.

Машинне навчання (Machine Learning – ML) – великий підрозділ штучного інтелекту, що вивчає методи побудови алгоритмів, здатних навчатися (рис.1). Машинне навчання – це процес, під час перебігу якого система опрацьовує велику кількість прикладів, виявляє закономірності і використовує їх, щоб прогнозувати вихідні характеристики для нових вхідних даних.

Рис. 1. Машинне навчання і його складові

У традиційному програмуванні, щоб отримати рішення, необхідно розробити алгоритм і написати код[1] [1]. Потім слід задати вхідні параметри, а реалізований алгоритм вже видає результат (рис.2).

Рис. 2. Розв'язання задачі за допомогою традиційного програмування

Для розв'язання тієї ж задачі методами ML застосовується зовсім інший підхід. Замість того, щоб розробляти алгоритм, збирають масив даних, який буде використано для напівавтоматичної побудови моделі.

Зібраний достатній набір даних завантажується у відповідний алгоритм машинного навчання. Результатом є модель, яка може прогнозувати новий результат, отримуючи на вхід нові дані.

[1] https://dou.ua/lenta/articles/ml-in-real-life/

Використовується готова модель аналогічно традиційному програмуванню: вона отримує вхідні дані і видає результат (рис. 3).

Рис. 3. Розв'язання задачі за допомогою машинного навчання

Залежно від задачі, яку треба розв'язати, обирається відповідний алгоритм. При цьому для певного класу задач можна спробувати використати різні алгоритми для досягнення найбільшої точності прогнозування результату. У табл. 1 наведено методи, призначення використання і популярні алгоритми ML для розв'язання задач.

Більшу частину завдань ML можна поділити на навчання з учителем (supervised learning) і навчання без вчителя (unsupervised learning). У разі навчання з учителем маємо вхідні дані, на підставі яких потрібно щось передбачити, і деякі гіпотези, а отже, готові правильні відповіді. У разі навчання без вчителя маємо тільки дані, характерні властивості яких хочемо виділити, щоб розділити дані, наприклад, на групи.

Розглянемо такий класичний приклад навчання з учителем. Припустимо, що маємо дані про 10 000 квартир в Києві, причому відома площа кожної квартири, кількість кімнат, поверх, на якому вона розташована, район, наявність місця паркування, відстань до найближчої станції метро тощо. Крім того, відома вартість кожної квартири. Завданням є побудова моделі, яка на основі цих ознак передбачатиме вартість квартири, що не внесена в нашу базу даних. Таке завдання називається регресією.

Машинне навчання – це не тільки математична, а й практична інженерна дисципліна. Здебільшого сама лише теорія не призводить зразу до методів і алгоритмів, які можуть застосовуватися на практиці. Задля доброго результату слід винаходити додаткові евристики, що компенсують невідповідність зроблених в теорії припущень умовам реальних завдань.

ML перебуває на стику математичної статистики, методів оптимізації та класичних математичних дисциплін, але має також і власну специфіку, що пов'язана з проблемами обчислювальної ефективності та перенавчання (рис. 4). Багато методів тісно пов'язано з витягуванням інформації й інтелектуальним аналізом даних (Data Mining).

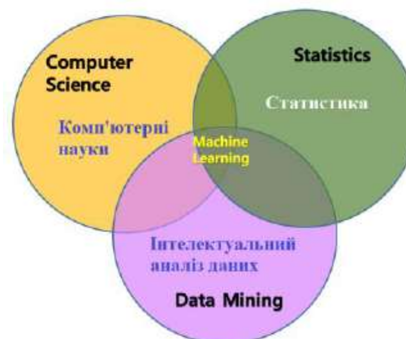



Рис. 4. Місце машинного навчання серед інших дисциплін

*Таблиця 1*

Завдання і алгоритми машинного навчання

| Завдання | Призначення використання | Популярні алгоритми |
|---|---|---|
| Класифікація | Спам-фільтри<br>Визначення мови<br>Пошук подібних документів<br>Аналіз тональності<br>Розпізнавання рукописних букв і цифр<br>Визначення підозрілих транзакцій | Наївний Байєс<br>Дерева розв’язку<br>Логістична регресія<br>K-найближчих сусідів<br>Машини опорних векторів |
| Регресія | Прогноз вартості цінних паперів<br>Аналіз попиту, обсягу продаж<br>Медичні діагнози<br>Будь-які залежності кількості від часу | Лінійна регресія<br>Поліноміальна регресія |
| Кластеризація | Сегментація ринку (типів покупців, лояльності)<br>Об’єднання близьких точок на карті<br>Стискання зображень<br>Аналіз і розмітки нових даних<br>Детектори аномальної поведінки | Метод K-середніх<br>Mean-Shift<br>DBSCAN |
| Зменшення розмірності | Рекомендаційні системи<br>Красиві візуалізації<br>Визначення тематики і пошуку подібних документів<br>Аналіз фейкових зображень<br>Ризик-менеджмент | Метод головних компонент (PCA)<br>Сингулярне розкладання (SVD)<br>Латентне розміщення Дирихле (LDA)<br>Латентно-семантичний аналіз (LSA, pLSA, GLSA)<br>t-SNE |
| Пошук правил | Прогнозування акцій і розпродаж<br>Аналіз товарів, які купуються разом<br>Розміщення товарів на полицях<br>Аналіз шаблонів поведінки на вебсайтах | Apriori<br>Euclat<br>FP-growth |
| Навчання з підкріпленням | Самокеровані автомобілі<br>Роботи-пилососи<br>Ігри<br>Автоматична торгівля<br>Управління ресурсами підприємства | Q-Learning<br>SARSA<br>DQN<br>A3C<br>Генетичний алгоритм |
| Ансамблі | Там, де підходять всі класичні алгоритми (але працюють точніше)<br>Пошукові системи<br>Комп’ютерний зір<br>Розпізнавання об’єктів | Random Forest<br>Gradient Boosting |
| Нейронні мережі | Замість всіх вищенаведених алгоритмів<br>Визначення об’єктів на фото та відео<br>Розпізнавання і синтез мови<br>Опрацювання зображень, перенесення стилю<br>Машинний переклад | Перцептрон<br>Згорткові мережі (CNN)<br>Рекурентні мережі (RNN)<br>Автоенкодери |

Щоб отримати уявлення про кількість алгоритмів, які використовуються на сьогодні, можна побудувати «карту» машинного навчання[2] [2] (рис. 5). На цій карті відображено обмежену кількість алгоритмів.

[2] https://vas3k.ru/blog/machine_learning/

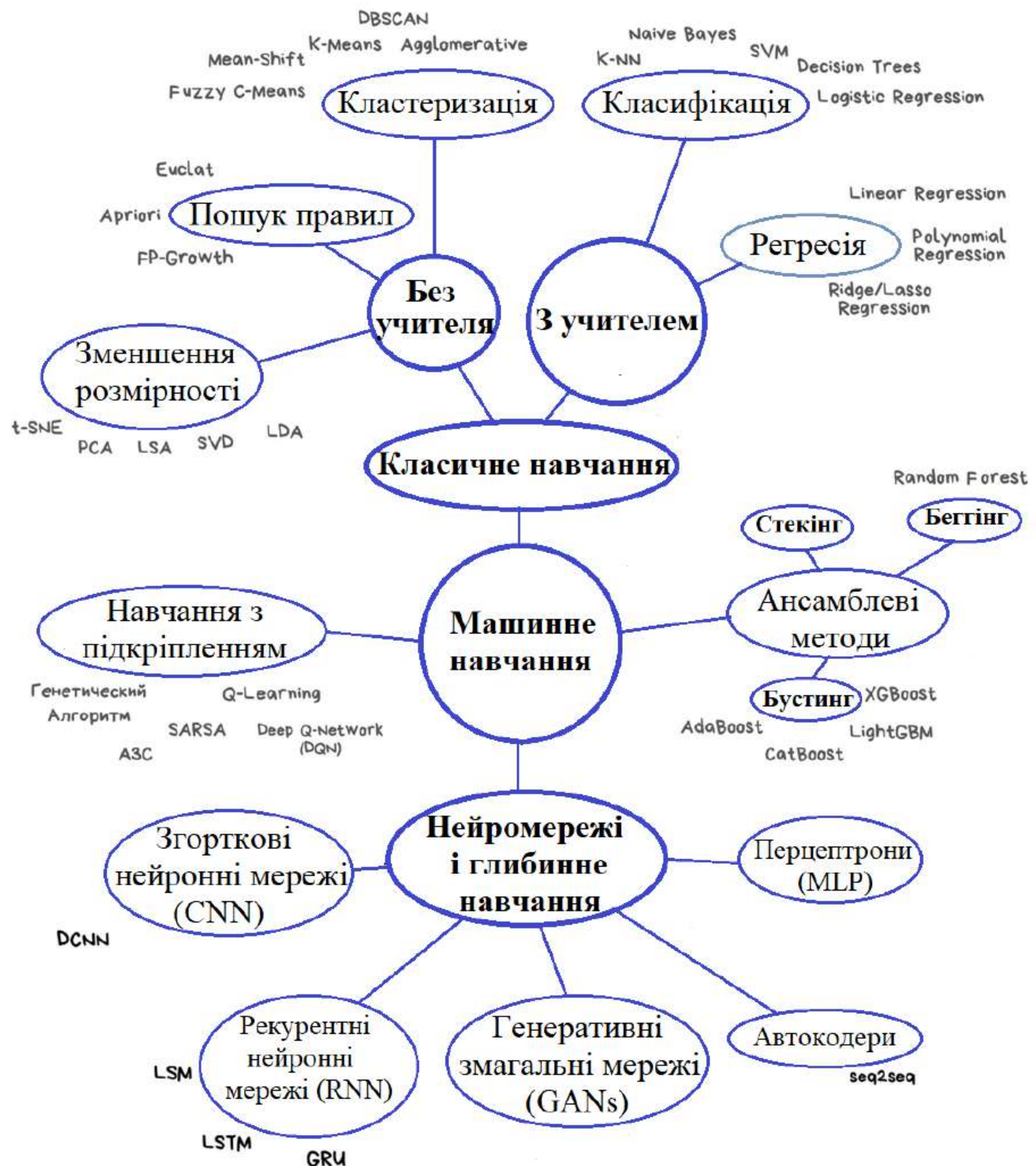



Рис. 5. Карта машинного навчання

У складі ML перебувають штучні нейронні мережі (ШНМ) (рис. 1). ШНМ – це обчислювальні системи, які мають здібності до самонавчання, поступового підвищення своєї продуктивності. Основними елементами структури ШНМ є:

- штучні нейрони, що являють собою елементарні, пов'язані між собою одиниці;
- синапс – з'єднання, що використовується для відправки-отримання інформації між нейронами;
- сигнал – власне інформація, яка підлягає передачі.

Як зазначено в табл.1, зараз ШНМ можна застосувати всюди, де працюють стандартні алгоритми ML, і отримати при цьому точніші результати. На рис. 6 наведено архітектури найбільш популярних ШНМ[3] [3].

[3] https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deep-learning-big-data-678c51b4b463

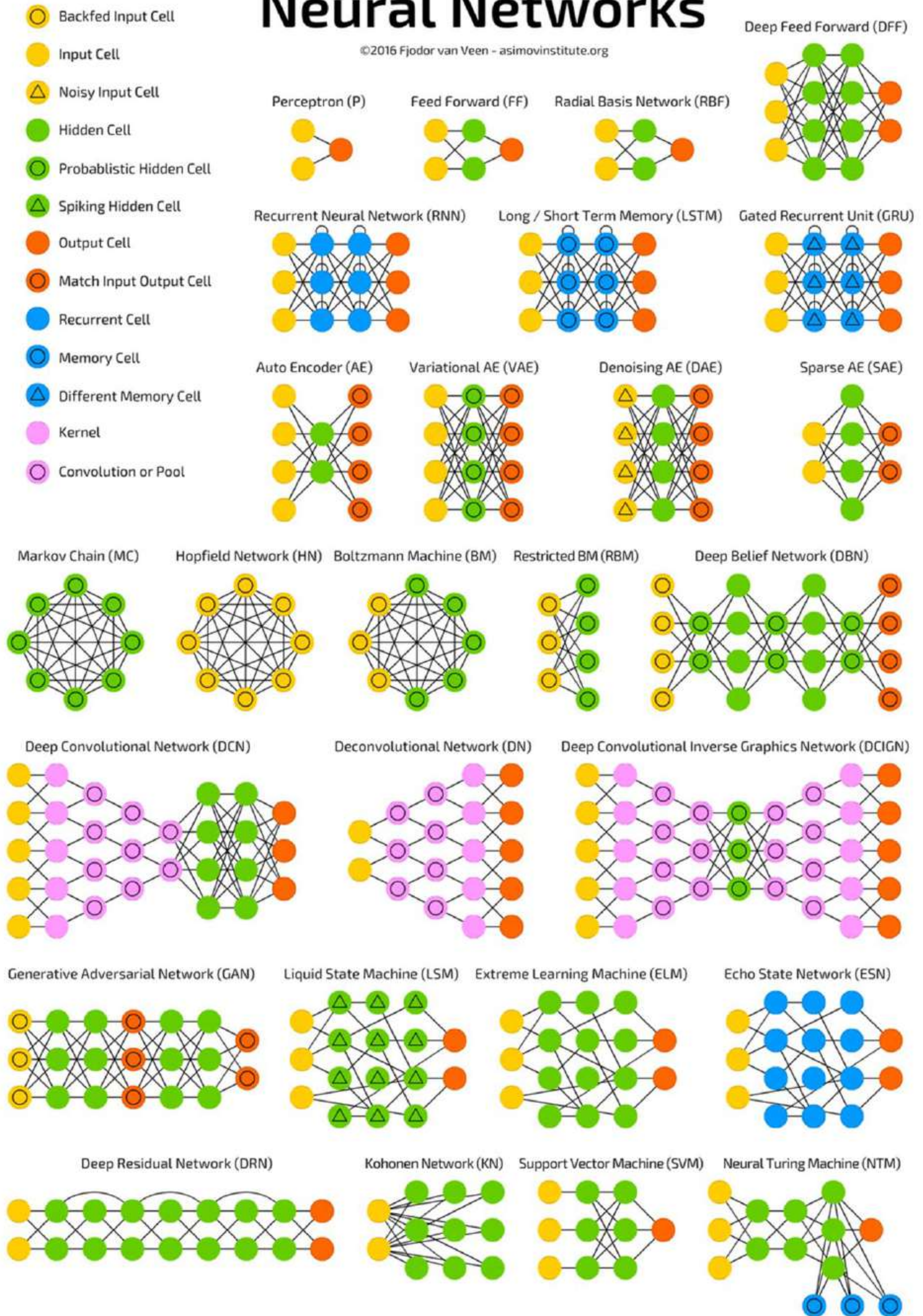



Рис. 6. Архітектури штучних нейронних мереж

Важливою відмінністю ML є кількість вхідних параметрів (ознак), які модель здатна опрацювати. Для коректного прогнозу погоди в тій чи іншій локації за теорією знадобиться ввести тисячі параметрів, які вплинуть на результат. Людина апріорі не може побудувати такий алгоритм, який буде використовуватися розумним способом. Для ML таких обмежень не існує. Поки вистачає потужності процесора і пам'яті, можна аналізувати стільки вхідних параметрів, скільки необхідно.

Будь-яку працюючу технологію ML можна умовно розподілити до одного з трьох рівнів доступності. Перший рівень – технологія доступна тільки різним технологічним гігантам рівня Google, Microsoft або IBM. Другий рівень – технологією може скористатися, наприклад, студент з деяким багажем знань. Третій рівень – технологію навіть бабуся здатна опанувати.

Зараз ML перебуває на стику другого і третього рівнів, внаслідок чого швидкість зміни світу за допомогою цієї технології зростає щодня. Велику роль в цьому відіграли такі великі фірми, як Google, AWS, Microsoft та інші, які надали для роботи свої фреймворки та обчислювальні ресурси (багато з них – безкоштовно).

Наведемо коротку характеристику основних фреймворків для машинного навчання.

**TensorFlow**

TensorFlow (TF) добре працює із зображеннями і даними на основі послідовностей. Для початківців поглиблене вивчення TF може бути непростим. Рекомендуємо більше практикувати, читати статті, щоб опанувати TF. Після того як ви добре зрозумієте фреймворк, впровадження моделей глибокого навчання стане для вас дуже простим.

**Keras**

Keras – це доволі міцна основа для старту. Якщо ви ознайомлені з Python і не займаєтесь високим рівнем досліджень, не розробляєте якусь особливу нейронну мережу, то Keras – це для вас. Основна увага зосереджена здебільшого на досягненні результатів, а не на нюансах моделі. Тож, якщо розробляєте проєкт, пов'язаний, наприклад, з класифікацією зображень чи моделями послідовностей, почніть з Keras. Ви зможете отримати робочу модель дуже швидко. Keras також інтегрований у TF.

**PyTorch**

Порівняно з TF, PyTorch є більш інтуїтивно зрозумілим. Навіть якщо у вас немає досвіду машинного навчання, ви зрозумієте моделі PyTorch. У PyTorch немає такого інструмента візуалізації, як TensorBoard, але завжди можна використати бібліотеку на зразок matplotlib.

**Caffe**

Caffe працює дуже добре, коли будують моделі глибокого навчання з даними про зображення. А коли йдеться про рекурентні нейронні мережі й мовні моделі, то Caffe відстає від інших фреймворків. Ключова перевага Caffe полягає в тому, що навіть якщо ви не володієте машинним навчанням, ви зможете будувати моделі глибокого навчання. Caffe використовується переважно для створення моделей глибокого навчання, для мобільних телефонів й інших обчислювальних платформ.

**Deeplearning4j**

Deeplearning4j – це рай для програмістів Java. Він пропонує широку підтримку таких різних нейронних мереж, як CNN, RNN та LSTM. Фреймворк може опрацювати величезну кількість даних і при цьому не позначитися на швидкості.

У пропонованому посібнику будемо працювати з фреймворком TF від Google, як з найбільш поширеним фреймворком.

# 1. Машинне навчання з TensorFlow

TensorFlow від Google – відкрита й найпопулярніша бібліотека з глибокого навчання для досліджень і розроблення. Цей курс охоплює основи таких тем, як лінійна регресія, класифікація, автоматичні енкодери, створення, навчання та оцінювання нейронних мереж CNN, RNN тощо.

Методичний посібник розроблено для початківців з невеликим досвідом роботи з TF (або без нього), але які мають базові знання Python[4] [4].

## Що таке TensorFlow

Нині найвідомішою бібліотекою глибокого навчання у світі є TF від Google. Google використовує машинне навчання у всіх своїх продуктах для вдосконалення пошукової системи, перекладу, підпису зображень або рекомендацій тощо.

Наведемо конкретний приклад: користувачі Google отримують швидкий і вдосконалений пошук за допомогою ШІ. Якщо користувач вводить ключове слово в рядку пошуку, Google надає рекомендації щодо того, яким може бути наступне слово (рис. 1.1).

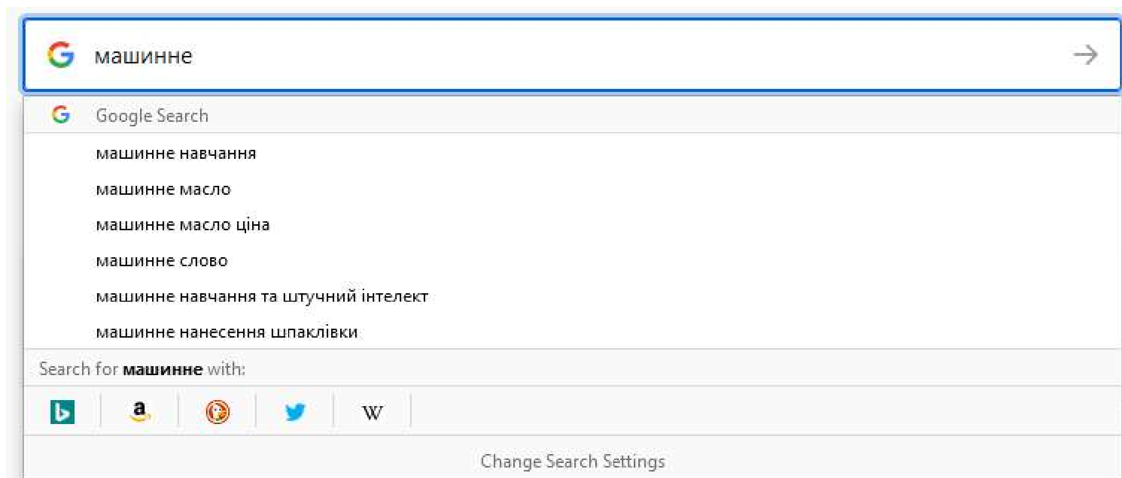



Рис. 1.1. Пошукові рекомендації Google

Google прагне використовувати машинне навчання, щоб скористатися своїми масивними наборами даних і забезпечити користувача найкращим досвідом. Три різні групи використовують машинне навчання: дослідники, спеціалісти з даних, програмісти.

Всі вони можуть використовувати один і той же набір інструментів для співпраці один з одним, а також підвищення своєї ефективності.

Google не просто має будь-які дані, а у них найбільш широко масштабований у світі комп'ютер, тому TF побудовано для масштабування. TF – це бібліотека, розроблена командою Google Brain для прискорення машинного навчання і досліджень глибокої нейронної мережі.

[4] https://www.guru99.com/what-is-tensorflow.html

Бібліотеку TF побудовано для роботи на декількох звичайних або графічних процесорах і навіть мобільних операційних системах, а також вона має кілька оболонок на різних мовах програмування: Python, C++, Java.

Нещодавно глибоке навчання почало перевершувати всі інші алгоритми машинного навчання при отриманні величезної кількості даних, а отже, Google може використовувати глибокі нейронні мережі для покращення своїх послуг:

- gmail;
- фото;
- пошукова система Google.

Розроблено основу, яка називається TF, щоб дати змогу дослідникам і розробникам спільно працювати над моделлю ШІ. Тож розширена TF допомагає багатьом людям користуватися нею.

Вперше TF було оприлюднено наприкінці 2015 року, а перша стабільна версія з'явилася у 2017 році (з відкритим кодом за ліцензією Apache Open Source). Отже, можна користуватися TF, змінювати і розповсюджувати модифіковану версію, отримуючи за це окрему плату, а нічого за це не платити Google.

Архітектура TF працює в трьох частинах:

- попереднє опрацювання даних;
- побудова моделі;
- тренування і оцінювання моделі.

Назву TF вибрано тому що вхідні дані приймаються як багатовимірний масив, відомий також як тензор. Можемо побудувати своєрідну блок-схему операцій (Graph), які потрібно виконати на конкретному вході. Вхідні дані надходять з одного кінця, а потім вони протікають через задану систему багатьох операцій і виходять з іншого кінця як вихід.

Фрейкворк назвали **TensorFlow** тому, що тензор протікає в ньому через список операцій, а потім виходить з іншого кінця.

## Архітектура TensorFlow

TF має певні вимоги до апаратного і програмного забезпечення:

- фаза розроблення – це коли режимом роботи є тренування (тренування зазвичай проводиться на десктопному комп'ютері або ноутбуці);
- фаза запуску або фаза виведення (після того як тренінг закінчено, TF можна запускати на багатьох різних платформах).

Можна запустити TF на:

- персональних комп'ютерах під керуванням Windows, macOS або Linux;
- хмарі як вебсервіс;
- таких мобільних пристроях, як iOS і Android;
- мікрокомп'ютерах (наприклад, Raspberry Pi).

Можемо тренувати TF на декількох машинах, а потім запустити його на іншій машині, як тільки отримаємо навчену модель, що дає змогу використати мікрокомп'ютери.

Модель може бути навчена і використана як на графічних процесорах (**GPU**), так і на звичайних процесорах (**CPU**). Спочатку графічні процесори були розроблені для відеоігор. Наприкінці 2010 року дослідники Стенфорда виявили, що GPU також дуже добре допомагає здійснювати матричні й алгебраїчні операції, оскільки виконує їх дуже швидко. Глибоке навчання покладається на безліч множення матриць. TF дуже швидко обчислює множення матриць, оскільки він написаний на C++. Незважаючи на те, що він реалізований на C++, до TF можна отримати доступ і керувати ним, використовуючи інші мови (переважно Python).

Також важливою особливістю TF є TensorBoard. TensorBoard дає змогу графічно і візуально відстежувати, що робить TF.

## Компоненти TensorFlow

### Тензор

У TF всі обчислення включають тензори. Тензор – це вектор або матриця n-розмірів, що представляє всі типи даних. Усі значення в тензорі містять однаковий тип даних з відомою (або частково відомою) формою. Форма даних – це розмірність матриці або масиву.

Тензор може бути створений з вхідних даних або в результаті обчислень.

### Графи

У TF всі операції виконуються всередині графа. Граф – це набір обчислень, які відбуваються послідовно. Кожна з операцій називається операційним вузлом (**op node**), які з'єднані між собою.

На графі надписується операція (**ops**) і будуються з'єднання між вузлами. Однак значення не відображаються. Ребро вузлів – це тензор, тобто спосіб заповнити операцію даними. Граф збирає і описує всі послідовні розрахунки, що зроблені під час тренінгу. Він має багато переваг:

1. Граф зроблено для роботи на декількох звичайних або графічних процесорах і навіть на мобільній операційній системі.
2. Можливість перенесення графа дає змогу зберегти обчислення для негайного чи пізнішого використання. Граф можна зберегти для виконання у майбутньому.
3. Усі обчислення на графі виконуються шляхом з'єднання тензорів разом.
4. У тензора є вузол і ребро. Вузол здійснює математичну операцію і генерує виходи кінцевих точок. На краях ребер, які з'єднують вузли, пояснюють співвідношення вводу / виводу між вузлами.

TF – найкраща бібліотека, вона створена з метою доступності для всіх. Бібліотека TF включає різні API, побудовані на такій масштабованій архітектурі глибокого навчання, як CNN або RNN. TF засновано на обчисленні графа, що допомагає розробнику візуалізувати побудову нейронної мережі за допомогою TensorBoard. Цей інструмент корисний для налагодження програми. TF працює на звичайному процесорі і GPU.

TF набуває найбільшої популярності на GitHub, порівняно з іншими фрейворками глибокого навчання.

Натепер TF 2 має вбудований API для:

- лінійної регресії: tf.estimator.LinearRegressor;
- класифікації: tf.estimator.LinearClassifier;
- класифікації глибокого навчання: tf.estimator.DNNClassifier;
- поглибленого і більш широкого глибокого навчання: tf.estimator.DNNLinearCombinedClassifier;
- регресії бустерного дерева: tf.estimator.BoostedTreesRegressor;
- класифікації бустерного дерева: tf.estimator.BoostedTreesClassifier тощо.

## Простий приклад TensorFlow

```
import numpy as np
import tensorflow as tf
```

У перших двох рядках коду імпортуємо TensorFlow як tf. Для Python звичайна практика використовувати для бібліотеки коротке ім'я. Перевага полягає в тому, щоб уникнути введення повного імені бібліотеки, коли нам треба її використовувати. Наприклад, можемо імпортувати TensorFlow як tf і викликати tf, коли потрібно використати функцію TF.

Розглянемо елементарний робочий процес TF на простому прикладі. Створимо обчислювальний граф, який перемножує два числа (рис. 1.2).

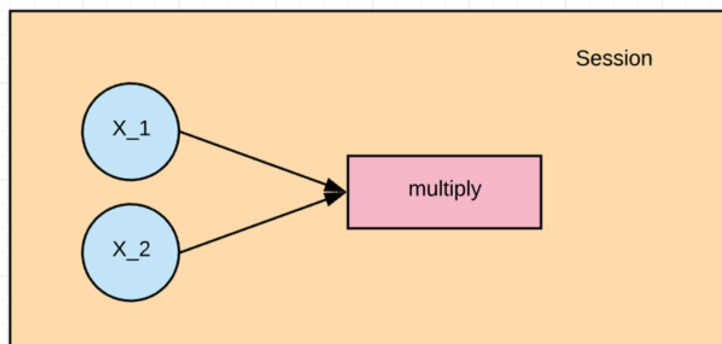



Рис.1.2. Операція множення в TensorFlow

Для прикладу помножимо X_1 на X_2. TF створить вузол для підключення операції. У нашому прикладі це буде множення. Коли буде визначений граф, обчислювальні двигуни TF будуть множити X_1 на X_2.

Отже, викликаємо сеанс TF, який запустить обчислювальний граф зі значеннями X_1 та X_2 і виведе результат множення.

Визначимо вхідні вузли X_1 і X_2. Коли ми створюємо вузол в TF, то вибираємо, який тип вузла треба створити. Вузли X_1 і X_2 будуть вузлами заповнення. Їм призначаються нові значення щоразу, коли проводиться розрахунок. Створимо їх як вузол заповнення точок tf.

**Крок 1. Визначення змінної**

```
X_1 = tf.placeholder(tf.float32, name = "X_1")
X_2 = tf.placeholder(tf.float32, name = "X_2")
```

Коли ми створюємо вузол заповнення, то треба передати тип даних: будемо подавати числа, а тип таких даних буде число з плаваючою комою, тому скористаємося `tf.float32`. Також треба дати назву цьому вузлу. Ця назва з'явиться, коли подивимось на візуалізацію графа нашої моделі. Назвемо цей вузол X_1, передавши в параметрі `name` значення X_1, і таким же чином визначимо X_2.

**Крок 2. Визначення обчислення**

```
multiply = tf.multiply(X_1, X_2, name = "multiply")
```

Тепер визначимо вузол, який виконує операцію множення. У TF можемо це зробити, створивши вузол `tf.multiply`.

Перейдемо від вузлів X_1 і X_2 до вузла множення. Він вказує TF зв'язати ці вузли в обчислювальному графі, тому ми вказуємо йому витягнути значення **x** та **y** і перемножити для отримання результату. Назвемо вузол множення `multiply`. Це всі визначення для розглядуваного простого обчислювального графа.

**Крок 3. Виконання операції**

Для виконання операцій на графі треба створити сеанс. У TF це робиться з `tf.Session()`. Отже, коли є сеанс, можемо попросити сеанс запустити операції на нашому обчислювальному графі, викликавши сеанс. Для виконання обчислень скористаємося `run`.

Коли операція буде запущена, то вона побачить, що необхідно взяти значення вузлів X_1 і X_2, тому слід подати значення для X_1 і X_2. Можна це зробити, поставивши параметр, який називається `feed_dict`. Передаємо значення 1, 2, 3 для X_1 і 4, 5, 6 для X_2.

Результат виводимо через `print(result)`.

На виході маємо побачити 4, 10 і 18 для $1 \times 4$, $2 \times 5$ і $3 \times 6$:

```
X_1 = tf.placeholder(tf.float32, name = "X_1")
X_2 = tf.placeholder(tf.float32, name = "X_2")

multiply = tf.multiply(X_1, X_2, name = "multiply")

with tf.Session() as session:
    result = session.run(multiply, feed_dict={X_1:[1,2,3], X_2:[4,5,6]})
    print(result)

[ 4. 10. 18.]
```

## Завантаження даних у TensorFlow

Перший крок перед тренуванням алгоритму машинного навчання – це завантаження даних. Відомо два способи завантаження даних:

1. *Завантаження даних у пам'ять* – це найпростіший метод. За цим методом завантажуємо всі свої дані в пам'ять як єдиний масив. Можна написати код Python і рядок коду не буде пов'язаний з TF.

2. *Конвеєр (pipeline) даних TF*. TF має вбудований API, який допомагає завантажувати дані, виконувати операції й легко використовувати алгоритм машинного навчання. Цей метод працює дуже добре, особливо, коли є великий набір даних. Наприклад, відомо, що записи зображень є величезними і не уміщуються в пам'ять. Конвеєр даних сам керує пам'яттю.

Яке рішення використовувати? Спробуємо відповісти на це питання.

### Завантаження даних у пам'ять

Якщо набір даних не надто великий (менше 10 гігабайт), можна використовувати перший метод. Дані можуть уміститися в пам'яті. Можна використати відому бібліотеку Pandas для імпорту файлів CSV.

### Завантаження даних через конвеєр TensorFlow

Другий метод найкраще працює, якщо є великий набір даних. Наприклад, якщо є набір даних 50 гігабайт, а комп'ютер має лише 16 гігабайт пам'яті, то комп'ютер вийде з ладу.

У цій ситуації потрібно побудувати конвеєр TF. Конвеєр завантажить дані партією або невеликими порціями. Кожна партія буде подана на конвеєр і буде готова до навчання. Побудова конвеєра – це відмінне рішення, оскільки дає змогу використовувати паралельні обчислення. Це означає, що TF буде тренувати модель на декількох процесорах, що сприятиме швидкості обчислення і допоможе тренувати потужну нейронну мережу.

Отже, якщо є невеликий набір даних, то можна завантажити дані у пам'ять за допомогою бібліотеки Pandas. Якщо є великий набір даних і можливість використовувати декілька процесорів, то буде зручніше працювати з конвеєром TF.

**Створення конвеєра TensorFlow**

У попередньому прикладі ми вручну додавали три значення для X_1 і X_2, а тепер наведемо приклад, як завантажити дані в TF через конвеєр.

**Крок 1. Створення даних**

Передусім скористаємося бібліотекою numpy для створення двох випадкових значень для даних:

```
import numpy as np
x_input = np.random.sample((1,2))
print(x_input)

[[0.8835775 0.23766977]]
```

**Крок 2. Створення заповнювача**

Як і у попередньому прикладі, створюємо заповнювач з іменем **X**. Нам треба чітко вказати форму тензора. У випадку, якщо завантажуємо масив лише з двома значеннями, можемо записати форму так shape = [1,2]:

```
# використання placeholder
x = tf.placeholder(tf.float32, shape=[1,2], name = 'X')
```

**Крок 3. Визначення методу набору даних**

Отже, необхідно визначити набір даних, в який зможемо записати значення заповнювача **x**. Для цього слід використати метод tf.data.Dataset.from_tensor_slices:

```
dataset = tf.data.Dataset.from_tensor_slices(x)
```

**Крок 4. Створення конвеєра**

На цьому кроці нам необхідно ініціалізувати конвеєр, куди будуть надходити дані. Для цього треба виконати make_initializable_iterator, щоб створити так званий ітератор. Тепер викликаємо цей ітератор для подачі наступної партії даних get_next. Цей крок так і називається: get_next. Звертаємо увагу, що в нашому прикладі є лише одна партія даних і лише з двома значеннями.

```
iterator = dataset.make_initializable_iterator()
get_next = iteraror.get_next()
```

**Крок 5. Виконання операції**

Останній крок схожий на попередній приклад. Ініціюємо сеанс і запускаємо ітератор операції. Подаємо feed_dict зі значенням, згенерованим numpy. Ці два значення заповнять заповнювач **x**. Потім запускаємо get_next для виведення результату.

```
with tf.Session() as sess:
    # заповнення placeholder даними
    sess.run(iterator.initializer, feed_dict={ x: x_input })
    print(sess.run(get_next)) # вихід [ 0.52374458  0.71968478]

[0.8835775  0.23766978]
```

**Висновки**

TensorFlow – найбільш відома нині бібліотека глибокого навчання. Практикуючий користувач TF може побудувати будь-яку структуру глибокого навчання (наприклад, CNN, RNN або просту штучну нейронну мережу).

Здебільшого TF використовують науковці, стартапи і великі компанії. Google використовує TF майже у всіх своїх щоденних продуктах, включаючи Gmail, фото, а також пошукову систему Google.

Команда Google Brain розробила TF, щоб заповнити розрив між дослідниками і розробниками продуктів. У 2015 році вони оприлюднили TF, який швидко набуває популярності. Нині TF – це бібліотека для глибокого навчання з найбільшою кількістю сховищ на GitHub.

TF легко розгорнути в масштабі. TF створено для роботи в хмарі або на таких мобільних пристроях, як iOs й Android, а також на мікрокомп’ютерах.

TF працює в сеансі. Кожен сеанс визначається графом з різними обчисленнями. Простим прикладом може бути множення чисел, в якому TF треба виконати три кроки:

1. Визначити змінну:

```
X_1 = tf.placeholder(tf.float32, name = "X_1")
X_2 = tf.placeholder(tf.float32, name = "X_2")
```

2. Визначити обчислення:

```
multiply = tf.multiply(X_1, X_2, name = "multiply")
```

3. Виконати операцію:

```
with tf.Session() as session:
    result = session.run(multiply, feed_dict={X_1:[1,2,3], X_2:[4,5,6]})
    print(result)
```

Поширеною практикою в TF є створення конвеєра для завантаження даних. Якщо дотримуватися наведених п’яти кроків, можна завантажити дані в TF:

1. Створення даних:

```
import numpy as np
x_input = np.random.sample((1,2))
print(x_input)
```

2. Створення заповнювача:

```
x = tf.placeholder(tf.float32, shape=[1,2], name = 'X')
```

3. Визначення методу набору даних:

```
dataset = tf.data.Dataset.from_tensor_slices(x)
```

4. Створення конвеєра:

```
iterator = dataset.make_initializable_iterator() get_next =
iteraror.get_next()
```

5. Виконання програми:

```
with tf.Session() as sess:
    sess.run(iterator.initializer, feed_dict={ x: x_input })
    print(sess.run(get_next))
```

# 2. Встановлення TensorFlow на Windows

Розглянемо, як встановити **TensorFlow** за допомогою **Anaconda,** як користуватися TF з **Jupyter** – переглядачем у вигляді блокнота[5] [5].

## Версії TensorFlow

TF підтримує одночасне обчислення на декількох звичайних і графічних процесорах. Це означає, що обчислення можуть бути розподілені між пристроями для збільшення швидкості навчання. Завдяки паралелізму нам не треба чекати тижнями, щоб отримати результати алгоритмів тренувань.

Для користувача Windows TF пропонує дві версії:

- **TF лише з підтримкою процесора (CPU)**: якщо ваш комп'ютер працює не на NVIDIA GPU, можна встановити лише таку версію.
- **TF з підтримкою GPU**: для більш швидкого обчислення можна використовувати цю версію TF. Вона має сенс лише тоді, коли потрібні потужні обчислювальні можливості.

У пропонованому посібнику достатньо базової версії TF з підтримкою CPU.

***Примітка.*** TF не забезпечує підтримку GPU на MacOS і мікрокомп'ютері Raspberry Pi.

Щоб запустити TF з блокнотом Jupyter, необхідно створити середовище в Anaconda. Це означає, що треба встановити Ipython, Jupyter і TF у відповідну папку на комп'ютері. Крім того, для наукових даних слід додати одну важливу бібліотеку – Pandas. Бібліотека Pandas допомагає маніпулювати кадром даних.

## Встановлення Anaconda

Пакет Anaconda включає в себе інтерпретатор мови Python (є версії 2 і 3), набір бібліотек, які найчастіше використовуються, і зручне середовище розроблення й виконання, яке запускається в браузері.

**Для встановлення пакета попередньо треба завантажити дистрибутив**.

Завантажуємо Anaconda[6] [6] версії 2019.07 (для Python 3.7) для відповідної системи.

Anaconda допоможе керувати всіма бібліотеками, необхідними для Python або R. Є варіанти для Windows, Linux і MacOS.

---

[5] https://www.guru99.com/download-install-tensorflow.html

[6] https://www.anaconda.com/download/

**Встановлення Anaconda на Windows**

1. Запустимо раніше завантажений інсталятор. У першому вікні треба натиснути «Next» (рис. 2.1).

Рис. 2.1. Запуск інсталятора

2. Приймаємо ліцензійну угоду (рис. 2.2).

Рис. 2.2. Ліцензійна угода

3. Обираємо одну з опцій встановлення (рис. 2.3).

Рис. 2.3. Вибір опцій встановлення

- Just Me – лише для користувача, який запустив встановлення.
- All Users – для всіх користувачів.

4. Вказуємо шлях, за яким буде встановлено Anaconda (рис. 2.4).

Рис. 2.4. Шлях до папки встановлення

5. Задаємо додаткові опції (рис. 2.5):

- Add Anaconda to the system PATH environment variable – додати Anaconda до системної змінної PATH;
- Register Anaconda as the system Python 3.7 – використовувати Anaconda як інтерпретатор Python 3.7 за замовчуванням.

Для початку встановлення натискаємо кнопку «Install».

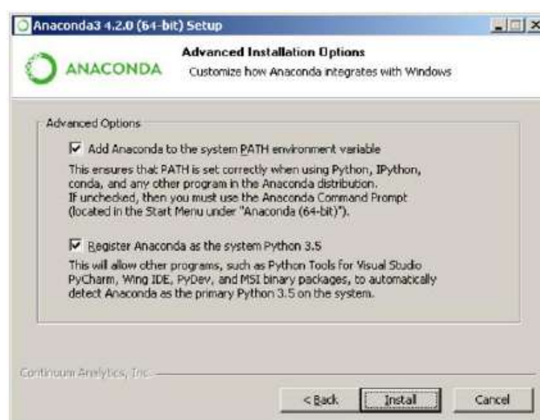



Рис. 2.5. Додаткові опції встановлення

5. Після цього буде виконано встановлення Anaconda (рис. 2.6).

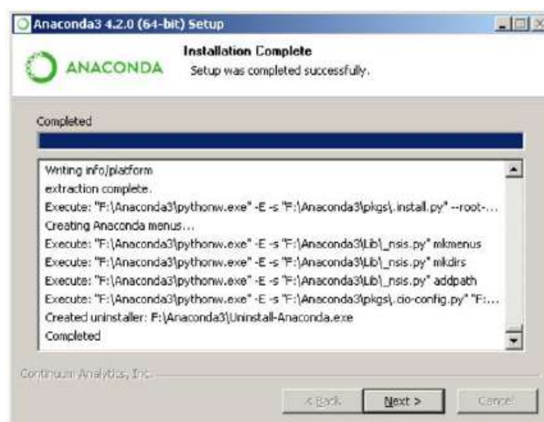



Рис. 2.6. Завершення встановлення Anaconda

## Створення .yml-файла для встановлення TensorFlow і залежностей

Необхідно виконати ще кілька кроків, щоб завершити встановлення TF.

**Крок 1. Знаходження Anaconda**

Перший крок, який треба зробити, – це знайти шлях до Anaconda. Потім створимо нове середовище conda, що включає необхідні бібліотеки, які будемо використовувати під час вивчення TF.

В ОС Windows можемо скористатись Anaconda-рядком і набрати (рис. 2.7):

```
C:\>where anaconda
```

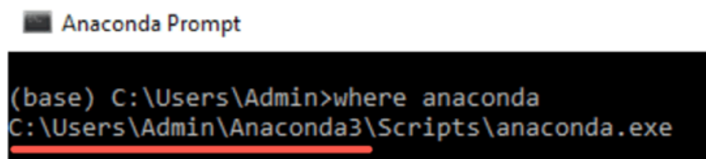



Рис. 2.7. Знаходження Anaconda

Слід знати назву папки, в якій встановлена Anaconda, оскільки ми створюємо нове середовище всередині цієї папки. Наприклад, на рис. 2.7 Anaconda встановлена в папці Admin. Можна зробити так само, тобто, Admin або ім’я користувача.

Далі встановимо робочий каталог в c:\Anaconda3.

Необхідно створити нову папку всередині Anaconda, в якій будуть розміщені Ipython, Jupyter і TF. Швидкий спосіб встановити бібліотеки і програмне забезпечення – написати файл .yml.

**Крок 2. Встановлення робочого каталогу**

Слід вказати робочий каталог, в якому створимо файл .yml. Як було зазначено, він буде розташований всередині Anaconda.

В ОС Windows вводимо (подивіться спочатку, який шлях до папки з Anaconda3):

```
cd C:\Users\Admin\Anaconda3
```

або шлях, який буде показано з командою `where anaconda` (рис. 2.8).

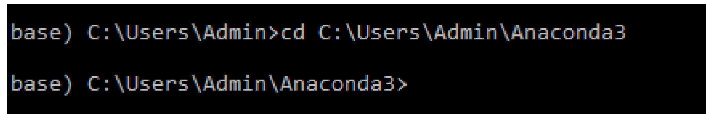



Рис. 2.8. Робочий каталог Anaconda

**Крок 3. Створення файла .yml**

Можемо створити файл .yml всередині нового робочого каталогу. У файлі встановлюються залежності, необхідні для запуску TF. Скопіюємо і вставимо у Terminal наведений код:

```
echo.>hello-tf.yml
```

З’явиться новий файл з назвою hello-tf.yml (рис. 2.9).

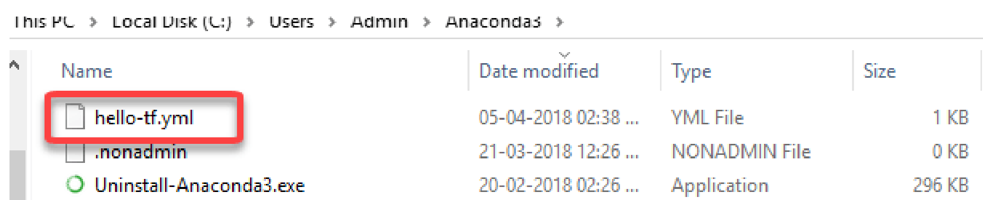



Рис. 2.9. Новий файл hello-tf.yml в робочому каталозі

**Крок 4. Редагування файла .yml**

Скористаємося блокнотом для виконання наступного кроку:

```
notepad hello-tf.yml
```

Додаємо у файл зміст:

```
name: hello-tf
dependencies:
- python=3.7
- jupyter
- ipython
- pandas
```

Пояснення коду:

`name: hello-tf` ім’я файлу .yml,

`dependencies`: залежності:

`python = 3,7`

`jupyter`

`ipython`

`pandas`: встановить бібліотеки Python версії 3.7, бібліотеки Jupyter, Ipython й pandas

Відкриваємо блокнот і можемо редагувати файл звідти (рис. 2.10).

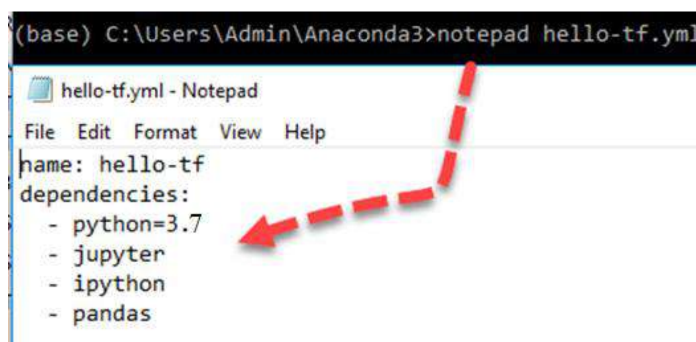



Рис. 2.10. Редагування файлу hello-tf.yml

***Примітка.*** Користувачі Windows встановлюють TF на наступному кроці. На цьому кроці ми тільки готували conda-середовище.

**Крок 5. Компілюємо файл .yml**

Можемо зкомпілювати .yml файл з таким кодом (рис. 2.11):

```
conda env create -f hello-tf.yml
```

```
(base) C:\Users\Admin\Anaconda3>conda env create -f hello-tf.yml
Solving environment: done

==> WARNING: A newer version of conda exists. <==
  current version: 4.4.10
  latest version: 4.5.0

Please update conda by running

    $ conda update -n base conda


Downloading and Extracting Packages
bleach 2.1.3: ######################################################################
numpy 1.14.2: ######################################################################
sip 4.19.8: ########################################################################
jedi 0.11.1: #######################################################################
notebook 5.4.1: ####################################################################
mkl 2018.0.2: ##################################1
```

Рис. 2.11. Компілювання файлу .yml

***Примітка.*** Нове середовище створюється всередині поточного каталогу користувачів. Це потребує часу і близько 1,1 ГБ місця на жорсткому диску.

**Крок 6. Активізація середовища conda**

Отже, все майже зроблено. Зараз маємо два середовища conda (рис. 2.12). Ми створили ізольоване середовище conda з бібліотеками, якими будемо користуватися під час навчання. Це рекомендована практика, оскільки кожен проєкт машинного навчання потребує різних бібліотек. Коли проєкт закінчено, можна це середовище видалити або залишити.

```
conda env list
```

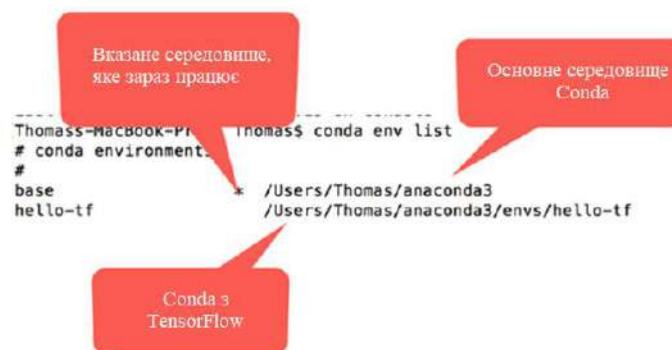



Рис. 2.12. Два середовища conda

Зірочка вказує на замовчування. Потрібно переключитися на hello-tf, щоб активувати середовище (рис. 2.13):

```
activate hello-tf
```

```
#
# To activate this environment, use:
# > source activate hello-tf
#
# To deactivate an active environment, use:
# > source deactivate
#
```

Рис. 2.13. Активація/деактивація середовища

Можна перевірити, чи всі залежності перебувають в одному середовищі. Це важливо, оскільки дає змогу Python використовувати Jupyter і TF з одного середовища. Якщо не видно трьох середовищ в одній папці, потрібно почати все заново.

### Крок 7. Встановлення TensorFlow

```
where python
where jupyter
where ipython
```

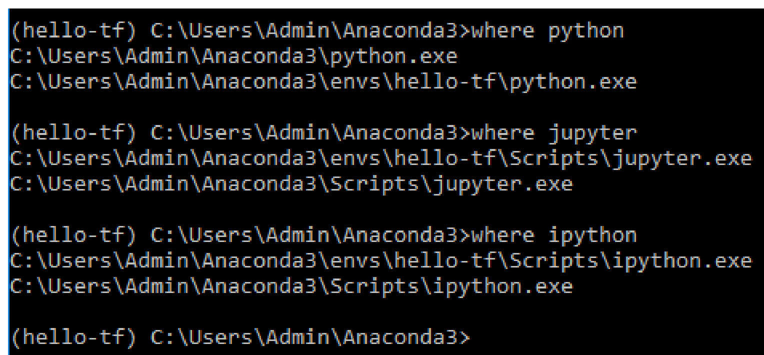



Рис. 2.14. Встановлені Python, Jupyter і Ipython

Отже, тепер є два середовища Python. Основне і новостворене в \envs\hello-tf. Основне середовище conda не має TF, встановлено лише hello-tf. З рис. 2.14 видно, що Python, Jupyter і Ipython встановлюються в одному середовищі. Це означає, що можна використовувати TF з блокнотом Jupyter.

Тепер можна встановити TF за допомогою такої команди (рис. 2.15):

```
pip install TensorFlow
```

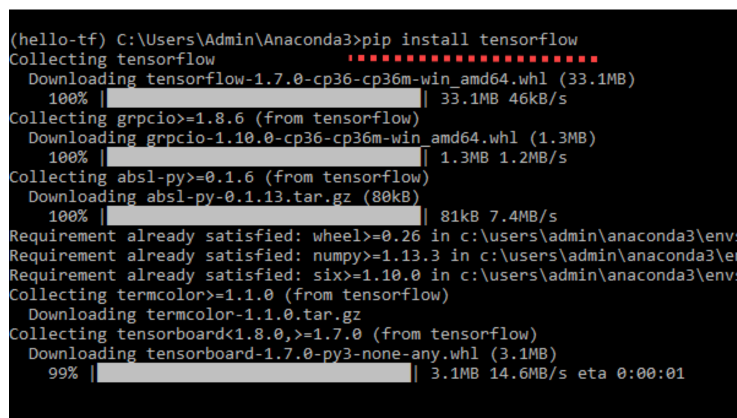



Рис. 2.15. Встановлення TensorFlow

## Використання блокнота Jupyter

TF можна відкрити за допомогою блокнота Jupyter.

Блокнот Jupyter – це вебдодаток, який дає змогу користувачу писати коди і текстові елементи різноманітної форми. Всередині нього можна писати абзаци, рівняння, заголовки, додавати посилання, фігури тощо. Блокнот корисний для того, щоб ділитися інтерактивними алгоритмами зі своєю аудиторією, зосереджуючись на навчанні або демонстрації техніки. Jupyter Notebook – це також зручний спосіб запустити аналіз даних.

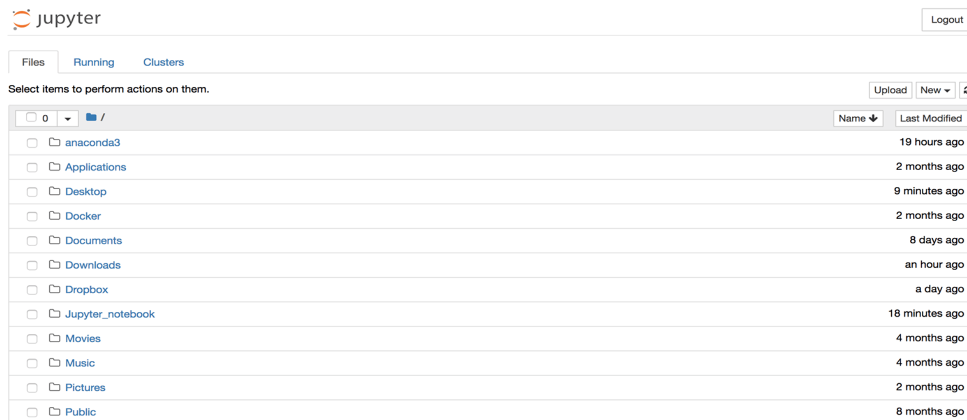



Рис. 2.16. Додаток Jupyter Notebook

Додаток Jupyter Notebook – інтерфейс (рис. 2.16), в якому можна писати сценарії та коди у веббраузері. Додаток використовується локально (тобто без доступу до мережі Інтернет) або на віддаленому сервері.

Кожне обчислення проводиться через ядро. Нове ядро створюється щоразу, коли запускаємо блокнот Jupyter.

***Примітка.*** Щоразу, коли бажаєте відкрити TF, треба ініціалізувати середовище і діяти у такій послідовності:

- активуємо hello-tf conda середовище;
- відкриваємо Jupyter;
- імпортуємо TF;.
- видаляємо блокнот;
- закриваємо Jupyter.

### Крок 1. Активуємо conda

```
conda activate hello-tf
```

### Крок 2. Відкриваємо Jupyter

Можемо відкрити Jupyter із Terminal (рис. 2.17).

```
jupyter notebook
```

```
Thomass-MacBook-Pro:anaconda3 Thomas$ source activate hello-tf
(hello-tf) Thomass-MacBook-Pro:anaconda3 Thomas$ jupyter notebook
[W 16:20:59.106 NotebookApp] Error loading server extension jupyter_tensorboard
    Traceback (most recent call last):
      File "/Users/Thomas/anaconda3/envs/hello-tf/lib/python3.6/site-packages/notebook/notebookapp.py", li
ne 1451, in init_server_extensions
        mod = importlib.import_module(modulename)
      File "/Users/Thomas/anaconda3/envs/hello-tf/lib/python3.6/importlib/__init__.py", line 126, in impor
t_module
        return _bootstrap._gcd_import(name[level:], package, level)
      File "<frozen importlib._bootstrap>", line 994, in _gcd_import
      File "<frozen importlib._bootstrap>", line 971, in _find_and_load
      File "<frozen importlib._bootstrap>", line 953, in _find_and_load_unlocked
    ModuleNotFoundError: No module named 'jupyter_tensorboard'
[I 16:20:59.113 NotebookApp] Serving notebooks from local directory: /Users/Thomas/anaconda3
[I 16:20:59.113 NotebookApp] 0 active kernels
[I 16:20:59.113 NotebookApp] The Jupyter Notebook is running at:
[I 16:20:59.113 NotebookApp] http://localhost:8888/?token=9ee3a11fea254e6982421366001678d5a6c01cebc8d5658e
[I 16:20:59.113 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip co
nfirmation).
[C 16:20:59.114 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=9ee3a11fea254e6982421366001678d5a6c01cebc8d5658e
[I 16:20:59.378 NotebookApp] Accepting one-time-token-authenticated connection from ::1
[W 16:21:00.612 NotebookApp] 404 GET /api/tensorboard?_=1522570860085 (::1) 24.15ms referer=http://localho
st:8888/tree
```

Якщо браузер не відкриває середовище, копіюйте цей URL

Рис. 2.17. Jupyter, відкритий в Terminal

Вебпереглядач має відкриватися автоматично, а якщо не відкривається, то копіюємо і вставляємо URL-адресу, вказану в терміналі (рис. 2.17). Починається з http://localhost:8888.

У Jupyter Notebook можна побачити всі файли всередині робочого каталогу. Щоб створити новий блокнот, треба просто натиснути на **new** і **Python 3** (рис. 2.18).

***Примітка.*** Новий блокнот автоматично зберігається всередині робочого каталогу.

Рис. 2.18. Збереження нового блокнота

**Крок 3. Імпортуємо TensorFlow**

Всередині блокнота можна імпортувати TensorFlow із псевдонімом tf. Клацніть для запуску (рис. 2.19, позначка 2). Знизу створюється нова комірка:

```
import tensorflow as tf
```

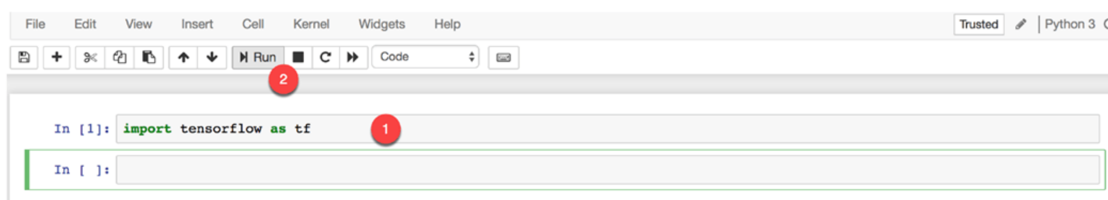



Рис. 2.19. Імпорт TensorFlow із псевдонімом tf

Отже, напишемо перший код за допомогою TF (рис. 2.20):

```
hello = tf.constant('Hello, Guru99!')
```

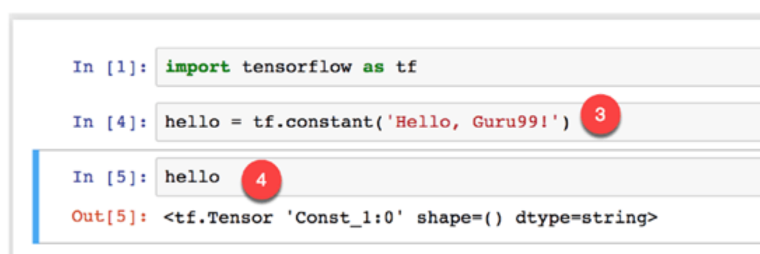



Рис. 2.20. Код за допомогою TensorFlow

```
hello
```

Створено новий тензор. Вітаємо, ви успішно встановили TF з Jupyter на свій комп’ютер.

**Крок 4. Видаляємо файл**

Можемо видалити файл Untitled.ipynb всередині Jupyter (рис. 2.21).

Рис. 2.21. Видалення створеного файлу

**Крок 5. Закриваємо Jupyter**

Є два способи, щоб закрити Jupyter. Перший спосіб – безпосередньо з блокнота. Другий спосіб – за допомогою терміналу (або Anaconda Prompt).

***З Jupyter***

На головній панелі блокнота Jupyter просто натисніть кнопку **Logout** (рис. 2.22).

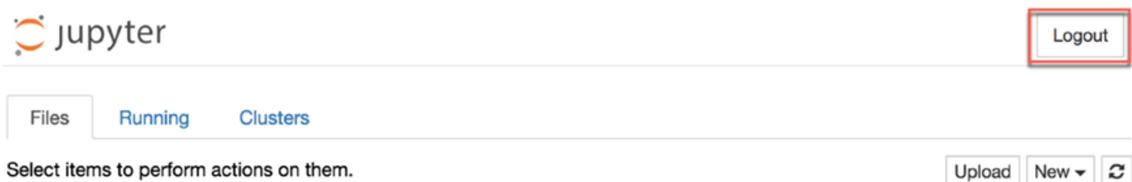
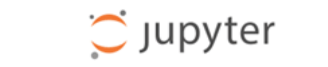



Рис. 2.22. Закривання Jupyter в блокноті

Ви будете переспрямовані на сторінку виходу (рис. 2.23).




Successfully logged out.
Proceed to the login page.

Рис. 2.23. Переспрямування на сторінку виходу

***З терміналу***

Вибрати термінал або запит Anaconda і двічі натиснути ctr+c.

Перший раз, коли натискаєте ctr+c, з’явиться прохання підтвердити, що ви хочете вимкнути блокнот (рис. 2.24). Повторіть ctr+c для підтвердження.

```
^C[I 12:38:40.238 NotebookApp] interrupted
Serving notebooks from local directory: /Users/Thomas
1 active kernel
The Jupyter Notebook is running at:
http://localhost:8888/?token=79edffd9c156eac054cf668c6576cb074716c2182a391
Shutdown this notebook server (y/[n])? No answer for 5s: resuming operatio
```

Двічі використайте Ctr+c, щоб закрити Jupyter

Рис. 2.24. Закривання Jupyter в терміналі

### Jupyter з основним середовищем conda

Якщо необхідно запустити TF з Jupyter для подальшого використання, треба відкрити новий сеанс.

```
source activate hello-tf
```

```
In [1]: import tensorflow as tf
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-41389fad42b5> in <module>()
----> 1 import tensorflow as tf

ModuleNotFoundError: No module named 'tensorflow'
```

Рис. 2.25. Помилка знаходження TensorFlow

Якщо цього не зробити, Jupyter не знайде TF (рис. 2.25).

Розглянемо детальніше деякі особливості використання Jupyter Notebook[7] [7]. Для цього напишемо рядок простого коду, щоб ознайомитися з оточенням Jupyter.

**Крок 1.** Додаємо папку всередині робочого каталогу, яка буде містити всі блокноти, які створимо під час вивчення TF.

Відкриємо термінал і введемо:

```
mkdir jupyter_tf
jupyter notebook
```

Пояснення коду:

- `mkdir jupyter_tf`: створюємо папку з назвою jupyter_tf;
- `jupyter notebook`: відкриваємо вебдодаток Jupyter.

**Крок 2.** Можна побачити нову папку всередині середовища (рис. 2. 26). Клацнемо папку jupyter_tf.

Рис. 2.26. Нова папка всередині середовища Jupyter

**Крок 3.** Всередині цієї папки створимо свій перший блокнот. Клацнемо на кнопці **New**, а потім на **Python 3** (рис. 2.27).

---

[7] https://www.guru99.com/jupyter-notebook-tutorial.html

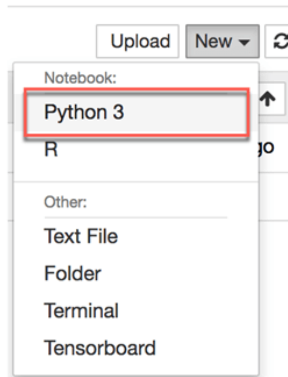



Рис. 2.27. Створення блокнота

**Крок 4.** Ми перебуваємо у середовищі Jupyter. Зараз наш блокнот називається Untitled.ipynb. Це ім'я за замовчуванням, яке дав Jupyter. Можна перейменувати його, натиснувши на **File** і вибравши **Rename** (рис. 2.28).

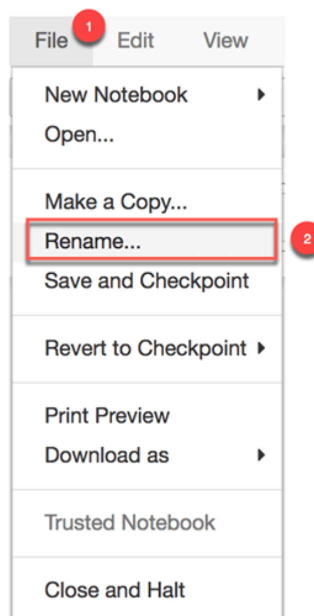



Рис. 2.28. Перейменування блокнота

Можемо перейменувати його в **Introduction_jupyter** (рис. 2.29).

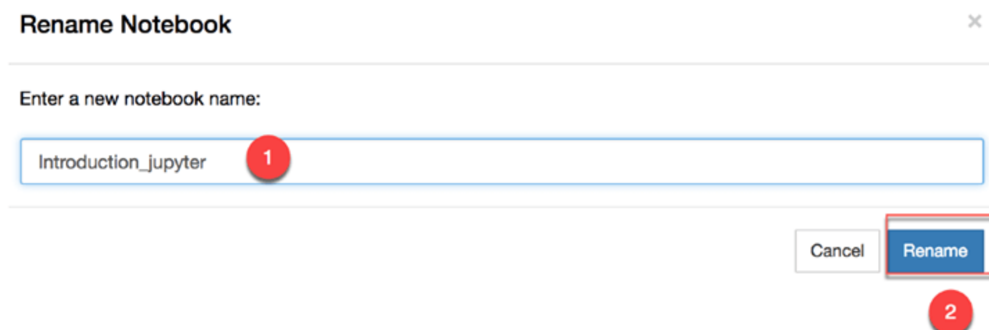



Рис. 2.29. Перейменування блокнота в **Introduction_jupyter**

У блокноті Jupyter пишемо коди, примітки або текст всередині комірок (рис. 2.30).

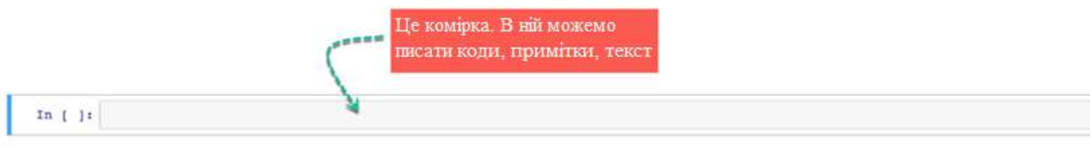



Рис. 2.30. Комірка блокнота

Всередині комірки можна написати один рядок коду (рис. 2.31).

```
In [1]: # Single line
        import matplotlib.pyplot as plt
```

Рис. 2.31. Рядок коду в комірці

Також можна написати кілька рядків (рис. 2.32). Jupyter читає один рядок коду за іншим.

```
In [2]: # Multiple line
        import numpy as np
        import pandas as pd
        from scipy import stats, integrate
```

Рис. 2.32. Кілька рядків коду в комірці

Наприклад, напишемо такий код всередині комірки (рис. 2.33).

```
In [6]: # Run the code
        %matplotlib inline
        import seaborn as sns
        sns.set(color_codes=True)
        np.random.seed(sum(map(ord, "distributions")))
        x = np.random.normal(size=100)
        sns.distplot(x)
        plt
```

Рис. 2.33. Код всередині комірки

На рис. 2.34 виведено виконання цього коду.

```
Out[6]: <module 'matplotlib.pyplot' from '/Users/Thomas/anaconda3/lib/python3.6/site-packages/matplotlib/pyplot.py'>
```

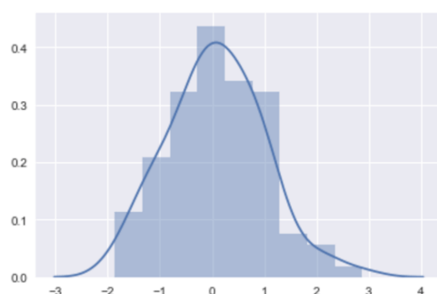



Рис. 2.34. Результат виконання коду

**Крок 5.** Тепер готові написати свій перший рядок коду. Можете помітити, що клітина має два кольори. Зелений колір означає, що ми перебуваємо в **editing mode** (режимі редагування) (рис. 2.35).

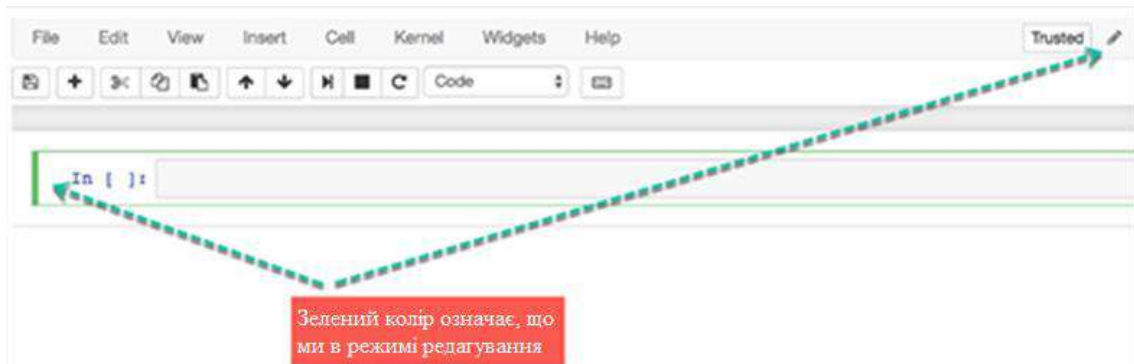



Рис. 2.35. Режим редагування

А синій колір вказує, що ми перебуваємо в режимі виконання (**executing mode**) (рис. 2.36).

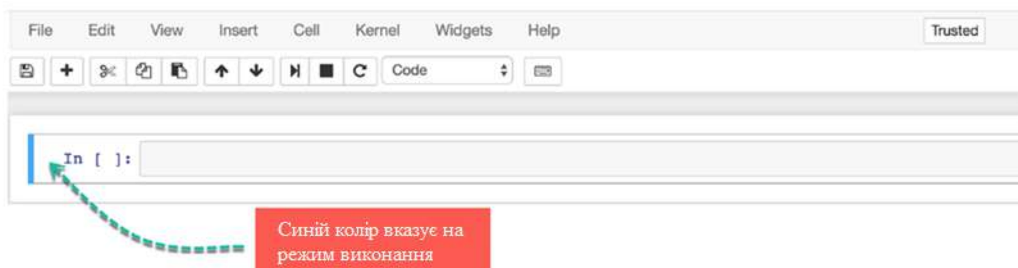



Рис. 2.36. Режим виконання

Перший рядок коду буде виводити Guru99!. Всередині клітинки можна написати:

```
print("Guru99!")
```

Є два способи запуску коду в Jupyter:

- клацнути і запустити (**Run**);
- скористатися гарячими клавішами.

Щоб запустити код, можна натиснути на **Cell**, а потім **Run Cells and Select Below** (рис. 2.37).

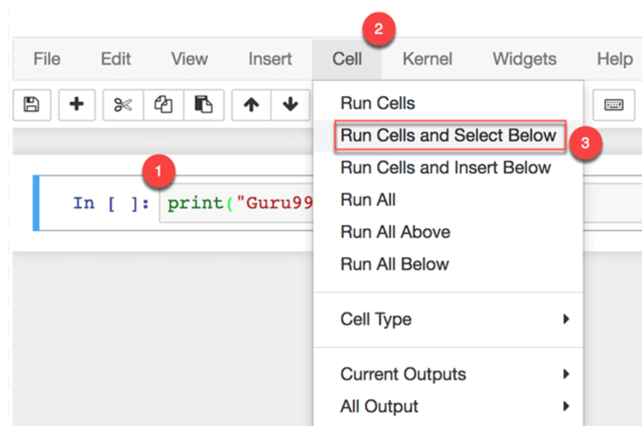



Рис. 2.37. Запуск коду з меню

Можна побачити, що код виводиться під коміркою, а нова комірка з’явилася відразу після виводу (рис. 2.38).

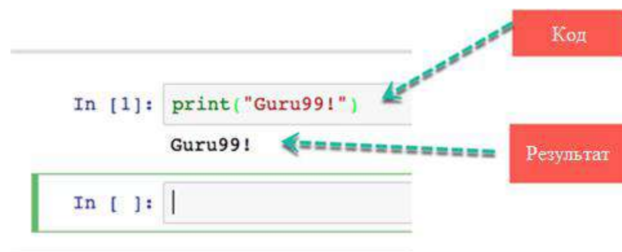



Рис. 2.38. Виконання коду

Більш швидкий спосіб запуску коду – це використання комбінацій клавіш. Щоб отримати доступ до комбінацій клавіш, перейдіть до **Help** і **Keyboard Shortcuts** (рис. 2.39).

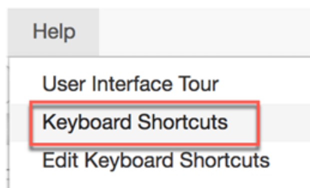



Рис. 2.39. Доступ до комбінації клавіш

Комбінації клавіш для Windows наведено на рис. 2.40.

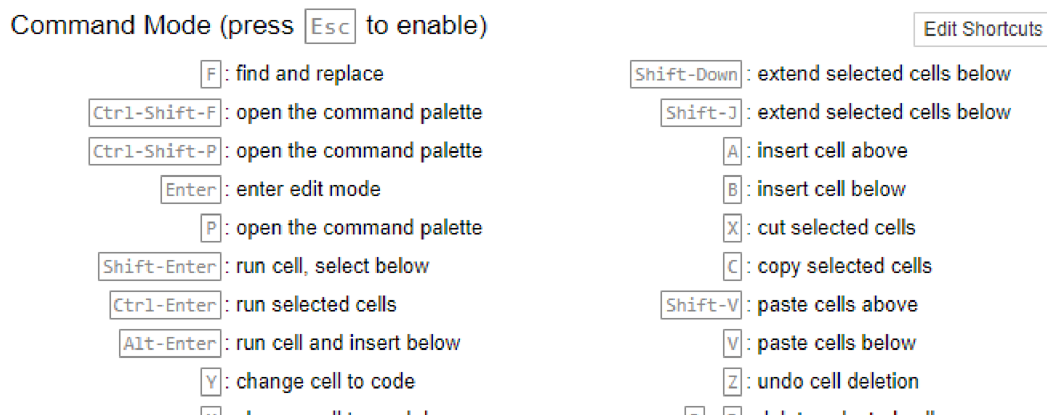



Рис. 2.40. Комбінації клавіш для Windows

Напишемо рядок:

```
print("Hello world!")
```

і спробуємо використати клавіші швидкого доступу для запуску коду. Натиснемо alt+enter: буде виконано код в комірці, а також вставлена нова порожня комірка внизу, як і раніше (рис. 2.41).

Рис. 2.41. Використання клавіш швидкого доступу

**Крок 6.** Настав час закрити блокнот. Перейдемо у **File** і натиснемо кнопку **Close and Halt** (рис. 2.42).

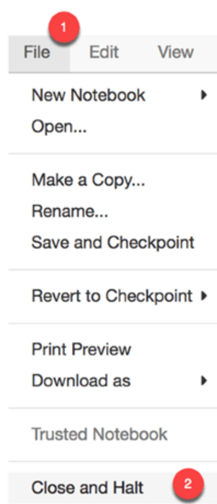



Рис. 2.42. Закривання блокнота

***Примітка.*** Jupyter автоматично зберігає блокнот з контрольною точкою. Може з’явитися таке повідомлення (рис. 2.43).

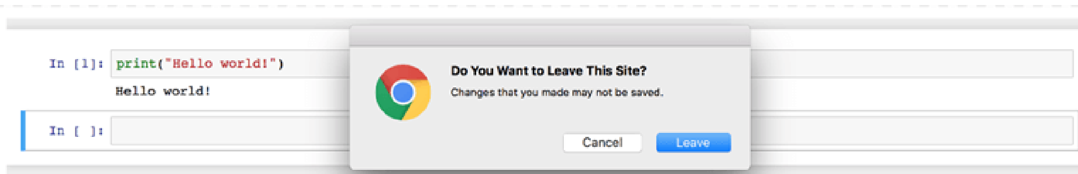



Рис. 2.43. Повідомлення при незбереженні файлу

Це означає, що Jupyter не зберіг файл з останньої контрольної точки. Можна вручну зберегти блокнот (рис. 2.44):

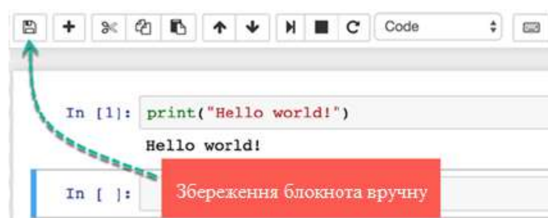



Рис. 2.44. Збереження блокнота

Будемо переспрямовані на головну панель. Можна побачити, що блокнот був збережений хвилину тому (рис. 2.45). Тепер можна безпечно вийти.

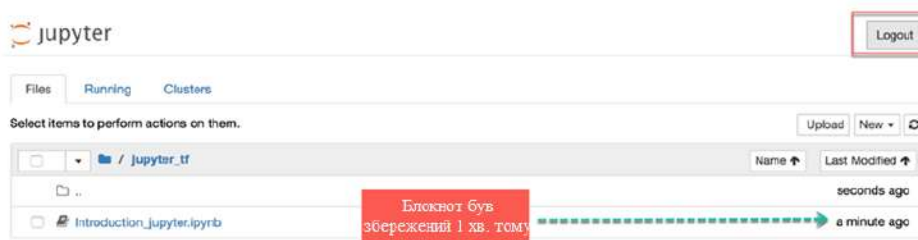



Рис. 2.45. Блокнот, збережений хвилину тому

**Висновки**

- Блокнот Jupyter – це вебдодаток, в якому можна запускати коди на Python і R. За допомогою Jupyter легко обмінюватися даними і виконувати детальний їх аналіз.
- Для запуску Jupyter вводимо у терміналі **jupyter notebook**.
- Зберегти блокнот можна де завгодно.
- Комірка містить наш код Python. Код зчитується послідовно рядок за рядком.
- Можна використовувати комбінацію гарячих клавіш для запуску комірки. За замовчуванням: Ctrl+Enter.

# 3. Встановлення TensorFlow на Raspberry Pi

Зараз на ринку пропонується багато пристроїв, які використовують потужність машинного навчання і штучного інтелекту. Наприклад, камера смартфона використовує функції ШІ для активізації розпізнавання обличчя і визначення очевидного віку виявленого обличчя.

Не дивно, що Google є одним із піонерів у цій технології. Google вже створив багато фреймворків ML і ШІ, які ми можемо легко використати у своїх програмах. TF – одна з добре відомих бібліотек Google з відкритим кодом нейронної мережі, яка використовується в таких машинних програмах, як класифікація зображень, виявлення об'єктів тощо.

Щоб навчитися використовувати деякі алгоритми машинного навчання на портативних пристроях скористаємося мікрокомп'ютером Raspberry Pi (RPi).

У цьому розділі дізнаємось, як встановити TF на RPi, а також покажемо кілька прикладів із простою класифікацією зображень у заздалегідь підготовленій нейронній мережі[8] [8].

Вимоги:

1. RPi з встановленою на нього Raspbian OS (мікроSD-карта не менше 16 ГБ 10 класу).
2. Підключення до Інтернету.

Використаємо SSH для доступу до RPi з ноутбука. Можемо використати підключення VNC або віддаленого робочого столу на ноутбуці, або підключити RPi до монітора.

RPi, будучи портативним і менш енергоємним пристроєм, використовується у багатьох програмах для опрацювання зображень у режимі реального часу, таких як розпізнавання облич, відстеження об'єктів, домашня система безпеки, камера спостереження тощо. Використовуючи таке програмне забезпечення комп'ютерного бачення, як OpenCV з RPi, можна створити багато ефективних програм опрацювання зображень.

Розробники ML і ШІ зробили операцію встановлення TF дуже простою, тож тепер TF можна встановити за допомогою лише кількох команд. Навіть якщо ви новачок у галузі машинного навчання, не виникне жодних проблем, а отже, ви зможете продовжити навчання і використовувати деякі приклади програм для вивчення.

## Встановлення TensorFlow на RPi у віртуальному середовищі

Встановимо версію TF 1.14 на робочому столі Raspbian Stretch (якщо обираємо Python 3.5) – це для RPi3 B+ або на робочому столі Raspbian Buster (якщо обираємо Python 3.7 або 2.7) – це для RPi4.

[8] https://circuitdigest.com/microcontroller-projects/intalling-machine-learning-software-tensorflow-on-raspberry-pi

Відкриємо термінал і виконаємо нижченаведені кроки.

**Крок 1. Встановлюємо залежності**

Оскільки ми будемо встановлювати `tensorflow` з Python 3.7.x, то спочатку треба встановити деякі залежності.

Передусім встановимо **pip** за допомогою команди:

```
sudo apt-get install python3-pip
```

В результаті буде встановлена остання версія pip для Python 3. Крім того, налаштуємо інструменти розробки Python 3 за допомогою:

```
sudo apt-get install python3-dev
```

**Крок 2. Встановлюємо віртуальне середовище**

Ми плануємо встановити TF у віртуальному середовищі, яке буде ізольоване від інших установлень python. Це захистить від проблем сумісності пакетів. Отже, у нас буде власний світ встановлених бібліотек (окремо від решти установлень на Raspbian).

Тому введемо у терміналі таку команду:

```
sudo apt-get install python-virtualenv
```

У процесі встановлення зі сховища будуть завантажуватися відповідні пакети і нас запитають:

*Do you want to continue? [Y/n]* (Ви хочете продовжити? [Y / n])

Натискаємо «y», а потім Enter, щоб продовжити. Установлення буде завершене через 2 хв.

**Крок 3. Створюємо нове віртуальне середовище**

Вводимо:

```
vitutalenv --system-site-packages -p python3 tensorflow
```

Створюється нове середовище під назвою «tensorflow» зі встановленим Python 3.

**Крок 4. Активуємо віртуальне середовище**

Для того щоб увійти в новостворене середовище, введемо команду:

```
source ~/tensorflow/bin/activate
```

Натискаємо Enter, переходимо в середовище `tensorflow`. Тепер командний рядок буде мати вигляд, як показано на рис. 3.1.

```
pi@raspberrypi:~ $ virtualenv  --system-site-packages -p python3 tensorflow
Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/pi/tensorflow/bin/python3
Also creating executable in /home/pi/tensorflow/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
pi@raspberrypi:~ $ source ~/tensorflow/bin/activate
(tensorflow) pi@raspberrypi:~ $
```

Рис. 3.1. Перехід в середовище `tensorflow`

**Крок 5. Отримуємо доступ до wheel-архіва TensorFlow**

Розглянемо вирішальну частину встановлення. Для цього треба знайти відповідні бінарні файли для апаратної архітектури RPi. У GitHub є сховище[9] [9], яке може надати нам заздалегідь складені бінарні файли для останніх версій TF (рис. 3.2).

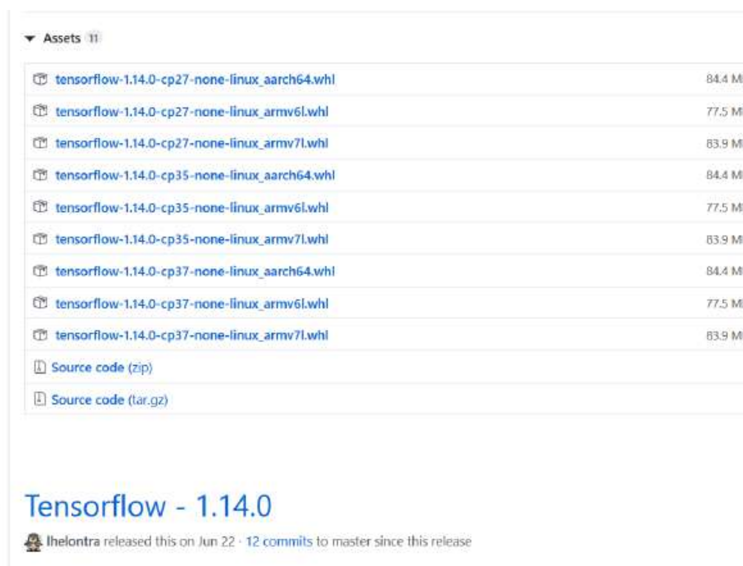



Рис. 3.2. Бінарні файли у сховищі GitHub

Перейшовши на сторінку GitHub, побачимо найновіші wheel-файли TF, доступні для завантаження.

Завантажимо відповідний wheel-архів для RPi 3 (Python 3.5) або для RPi 4 (Python 3.7). Для цього копіюємо URL-адресу відповідного wheel-файлу на сайті.

**Крок 6. Встановлення Tensorflow за допомогою pip**

Після копіювання URL-адреси wheel-файлу переходимо в термінал, який все ще перебуває в середовищі TF, і вводимо:

```
pip3 install 'URL-адреса wheel-файла'
```

Wheel-файл буде завантажено зі сховища і встановлено. Процес завантаження потребує деякого часу, оскільки залежить від швидкості підключення до мережі Інтернет. Завантажаться самостійно всі необхідні пакети TF. Цей процес здійснюється приблизно 5–10 хв.

Після завершення інсталяції pip повернемося до командного рядка.

**Крок 7. Перевірка налаштування**

Щоб перевірити встановлення, відкриємо інтерпретатор Python всередині терміналу, ввівши:

```
python
```

Тепер наберемо невеликий код:

```
>>import tensorflow as tf
>>tf.__version__
```

[9] https://github.com/lhelontra/tensorflow-on-arm/releases

Вищевказаний код приведе до відображення версії TF. Оскільки на цей час остання версія, яку встановили з Git, r1.14, то буде виведено:

```
'1.14.0'
```

Введемо рядок за рядком такий код (без коментарів):

```
>>hello=tf.constant('hello world') # буде створена константа hello
>>with tf.Session() as sess:
    print(sess.run(hello))
    #натискаємо Enter
```

Код приведе до виводу:

```
b'hello world'
```

Це очікуваний вихід, який підтверджує успіх налаштування.

## Встановлення TensorFlow безпосередньо на ОС Raspbian

Починаючи з версії TF 1.9, на офіційному сайті є підтримка встановлення TF на RPi.

**Крок 1.** Перш ніж встановлювати TF на RPi, треба поновити ОС Raspbian, використовуючи команди:

```
sudo apt-get update
sudo apt-get upgrade
```

**Крок 2.** Встановимо бібліотеку **Atlas**, щоб отримати підтримку **Numpy** й інших залежностей:

```
sudo apt install libatlas-base-dev
```

**Крок 3.** Встановимо TF з pip3 за допомогою команди:

```
pip3 install tensorflow
```

Для встановлення TF знадобиться деякий час. Якщо виникне повідомлення про помилку, ігноруємо його, або просто повторюємо встановлення за допомогою вищевказаної команди.

**Крок 4.** Щоб перевірити встановлення TF, спробуємо ввести

```
python3
```

А потім

```
import tensorflow
```

Якщо використовуємо версію Python більше 3.4, то може з'явитися повідомлення про помилку, ігноруємо його – все буде працювати.

Для перевірки версії встановленого TF вводимо (рис. 3.3):

```
tensorflow.__version__
```

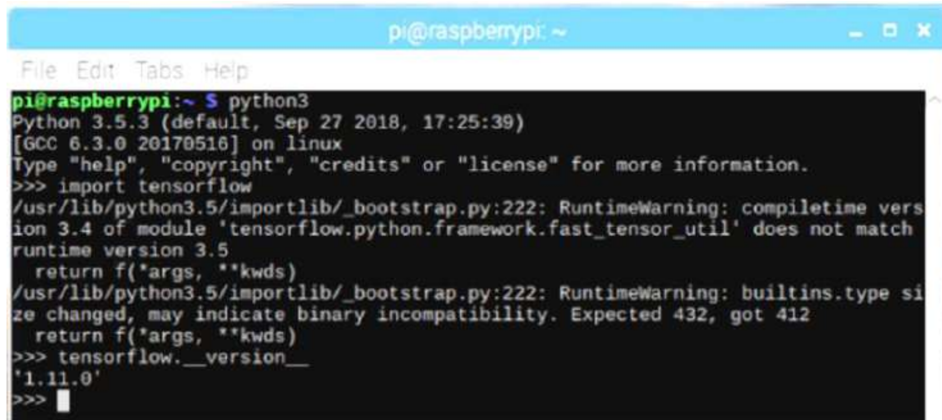



Рис. 3.3. Перевірка версії

Тепер можемо перевірити роботу TF, запустивши невелику програму Hello world. Для цього відкриваємо текстовий редактор `nano` за допомогою команди:

```
sudo nano tfcheck.py
```

Копіюємо і вставляємо наведені нижче рядки у терміналі `nano`, зберігаємо файл за допомогою ctrl+x. Натискаємо клавішу Enter.

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

**Крок 5.** Запускаємо збережений скрипт у терміналі, використовуючи команду:

```
python3 tfcheck.py
```

Якщо всі пакети встановлені належним чином, то побачимо повідомлення ***Hello Tensorflow!*** в останньому рядку, як показано на рис. 3.4.

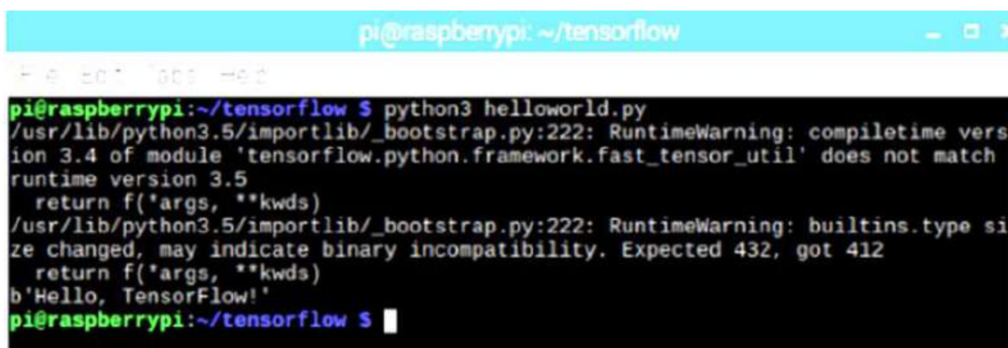



Рис. 3.4. Запуск тестової програми

Якщо ми працюємо з версією Python вище 3.4, то можемо отримати кілька попереджень при виконанні програми. Офіційні підручники TF визнають, що це відбувається, але рекомендують ці попередження ігнорувати.

Якщо все чудово працює, тоді зробимо щось більш цікаве за допомогою TF. Не треба мати ніяких знань з машинного і глибокого навчання, щоб зробити такий проєкт.

## Встановлення класифікатора зображень на RPi

Зображення буде подаватися за попередньо побудованою моделлю і TF визначить зображення, а також виведе ймовірність того, що є на зображенні.

**Крок 1.** Створимо каталог і перейдемо до каталогу за допомогою команд:

```
mkdir tf
cd tf
```

**Крок 2.** Завантажимо моделі, які доступні у сховищі TF GIT. Клонуємо сховище у каталог tf за допомогою команди:

```
git clone https://github.com/tensorflow/models.git
```

Установлення потребує певного часу, оскільки має великі розміри, тому переконайтеся, що є достатній обсяг пам'яті для розміщення даних.

**Крок 3.** Використаємо приклад класифікації зображень, який можна знайти в *models/tutorials/image/imagenet*. Переходимо у вказану папку:

```
cd models/tutorials/image/imagenet
```

**Крок 4.** Передаємо зображення у заздалегідь вбудовану нейронну мережу за допомогою команди:

```
python3 classify_image.py --image_file=/home/pi/image_file_name
```

Не забуваємо замінити `image_file_name` на файл зображення, яке треба передати, а потім натискаємо клавішу Enter.

Нижче наведено кілька прикладів виявлення і розпізнавання зображень за допомогою TF.

Визначення стільникового телефону з майже 99% точністю (рис. 3.5).

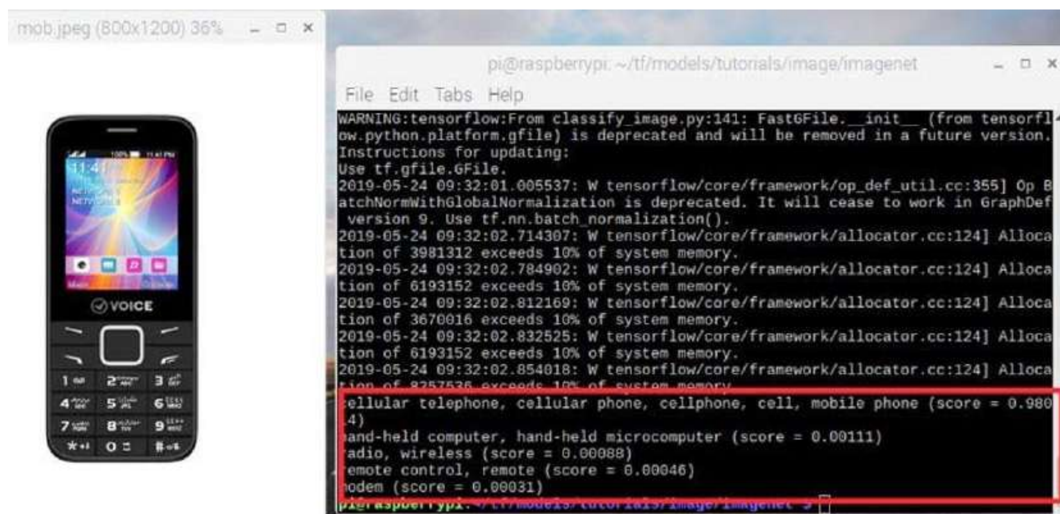



Рис. 3.5. Визначення стільникового телефону

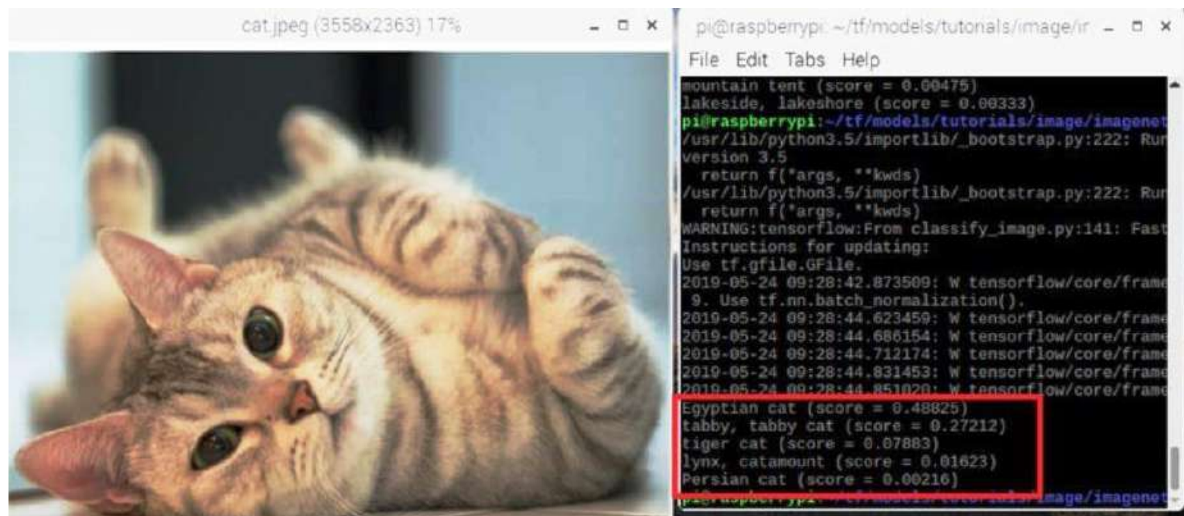



Рис.3.6. Визначення зображення кішки

Непогано! Нейронна мережа класифікувала зображення єгипетської кішки з високою ймовірністю порівняно з іншими варіантами (рис. 3.6).

У вищенаведених прикладах результати є доволі добрими, а отже, TF може легко класифікувати зображення з достатньою точністю. Можна спробувати це за допомогою налаштованих зображень.

# 4. Основи TensorFlow

## Що таке тензор, представлення тензора

Назва TensorFlow безпосередньо походить від основи фреймворка – **Tensor**[10] [10]. У TF всі обчислення включають тензори. Тензор – це **вектор** або **матриця** n-розмірів, що представляє всі типи даних. Усі значення в тензорі містять однаковий тип даних з відомою (або частково відомою) **формою**. Форма даних – це розмірність матриці або масиву.

Тензор може виникнути з вхідних даних або з результату обчислення. У TF всі операції проводяться всередині графа. Граф – це набір обчислень, які виконуються послідовно. Кожна з операцій називається **op node** і вони з'єднані між собою.

На графі окреслюються **ops** і з'єднання між вузлами, однак значення не відображаються. Ребра вузлів – це тензор, тобто спосіб заповнити операцію даними.

У машинному навчанні моделі подаються зі списком об'єктів, які називаються векторами параметрів. Вектор параметрів може бути будь-якого типу даних. Вектор параметрів зазвичай є первинним входом для заповнення тензора. Ці значення будуть надходити в **op node** через тензор, а результат цієї операції (обчислення) створить новий тензор, який також буде використовуватися в новій операції. Всі ці операції можна переглянути на графі.

У TF тензор – це сукупність векторів параметрів (тобто масиву) n-розмірів. Наприклад, якщо у нас є матриця 2 × 3 зі значеннями від 1 до 6, то пишемо:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

[10] https://www.guru99.com/tensor-tensorflow.html

TF представляє цю матрицю так:

```
[[1, 2, 3],
   [4, 5, 6]]
```

Якщо створимо тривимірну матрицю зі значеннями від 1 до 8, то отримаємо рис. 4.1.

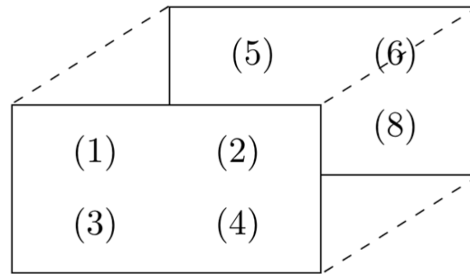



Рис. 4.1. Тривимірна матриця

TF представляє цю матрицю так:

```
[ [[1, 2],
       [3, 4]],
       [[5, 6],
       [7,8]] ]
```

***Примітка.*** Тензор може бути представлений скаляром або мати форму розмірності більше трьох, оскільки вищий рівень розмірності складніше візуалізувати.

## Типи тензора

У TF всі обчислення проходять через один або кілька тензорів. Тензор – це об'єкт з трьома властивостями:

- унікальна мітка (`name`);
- розмірність (`shape`);
- тип даних (`dtype`).

Кожна операція, яку будемо виконувати з TF, включає маніпулювання тензором. Можна створити чотири основні тензори:

- tf.Variable;
- tf.constant;
- tf.placeholder;
- tf.SparseTensor.

У цьому розділі дізнаємося, як створити tf.constant і tf.Variable.

Спочатку переконаємося, що активували середовище conda за допомогою TF. Ми назвали це середовище hello-tf:

```
activate hello-tf
```

Після цього можемо імпортувати TF:

```
# Імпорт tf
import tensorflow as tf
```

## Створення тензора n-розмірності

Почнемо зі створення тензора розмірності 1, так званого скаляра.
Для створення тензора можемо використати `tf.constant()`:

```
tf.constant(value, dtype, name = "")
arguments

- `value`: Value of n dimension to define the tensor. Optional
- `dtype`: Define the type of data:
    - `tf.string`: String variable
    - `tf.float32`: Flot variable
    - `tf.int16`: Integer variable
- "name": Name of the tensor. Optional. By default, `Const_1:0`
```

Для створення тензора розмірності 0 запустимо такий код:

```
## Розмірність 0
# Ім'я за замовчування
r1 = tf.constant(1, tf.int16)
print(r1)
```

**Результат на виході** (рис. 4.2).

```
Tensor("Const:0", shape=(), dtype=int16)
```

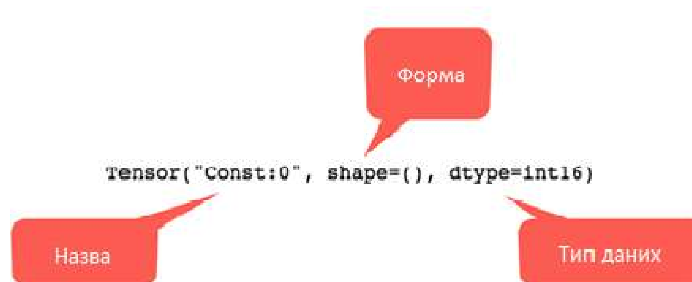



Рис. 4.2. Результат створення тензора порядку 0

```
# Названо my_scalar
r2 = tf.constant(1, tf.int16, name = "my_scalar")
print(r2)
```

**Результат на виході:**

```
Tensor("my_scalar:0", shape=(), dtype=int16)
```

Кожен тензор відображається своєю назвою. Кожен об'єкт тензора визначається унікальною міткою (назвою, англ. `name`), розмірністю (формою, англ. `shape`) і типом даних (`dtype`).

Можна визначити тензор з десятковими значеннями або з рядком, змінивши тип даних:

```
# Десятковий
r1_decimal = tf.constant(1.12345, tf.float32)
print(r1_decimal)
```

```
# Рядковий
r1_string = tf.constant("MAN_2019", tf.string)
print(r1_string)
```

**Результат на виході:**

```
Tensor("Const_1:0", shape=(), dtype=float32)
Tensor("Const_2:0", shape=(), dtype=string)
```

Тензор розмірності 1 можна створити так:

```
## Розмірність 1
r1_vector = tf.constant([1,3,5], tf.int16)
print(r1_vector)
r2_boolean = tf.constant([True, True, False], tf.bool)
print(r2_boolean)
```

**Результат на виході:**

```
Tensor("Const_3:0", shape=(3,), dtype=int16)
Tensor("Const_4:0", shape=(3,), dtype=bool)
```

Можна побачити, що форма складається лише з одного стовпця.

Щоби створити двовимірний масив, треба закривати дужки після кожного рядка, як в наведених нижче прикладах:

```
## Розмірність 2
r2_matrix = tf.constant([ [1, 2],
                          [3, 4] ],tf.int16)
print(r2_matrix)
```

**Результат на виході:**

```
Tensor("Const_5:0", shape=(2, 2), dtype=int16)
```

Матриця має 2 рядки і 2 стовпця, заповнені значеннями 1, 2, 3, 4.

Матриця розмірності 3 будується шляхом додавання іншого рівня, використовуючи дужки.

```
## Розмірність 3
r3_matrix = tf.constant([ [[1, 2],
                           [3, 4],
                           [5, 6]] ], tf.int16)
print(r3_matrix)
```

**Результат на виході:**

```
Tensor("Const_6:0", shape=(1, 3, 2), dtype=int16)
```

Матриця виглядає так, як наведено на рис. 4.2.

## Форма тензора

Коли ми виводимо тензор, то TF визначає форму. Однак можна отримати форму тензора з властивості форми.

Наприклад, побудуємо матрицю, заповнену числами від 10 до 15, і перевіримо форму `m_shape`:

```
# Форма тензора
m_shape = tf.constant([ [10, 11],
                        [12, 13],
                        [14, 15] ]
                     )
m_shape.shape
```

**Результат на виході:**

```
TensorShape([Dimension(3), Dimension(2)])
```

Матриця має 3 рядки і 2 стовпця.

TF має корисні команди для створення вектора або матриці, заповнених 0 або 1. Наприклад, якщо хочемо створити 1-D тензор із заданою формою 10, заповнений 0, можемо запустити код нижче:

```
# Побудова вектора з 0
print(tf.zeros(10))
```

**Результат на виході:**

```
Tensor("zeros:0", shape=(10,), dtype=float32)
```

Властивість також працює для матриці. Код нижче створює матрицю 10 × 10, заповнену 1:

```
# Побудова вектора з 1
print(tf.ones([10, 10]))
```

**Результат на виході:**

```
Tensor("ones:0", shape=(10, 10), dtype=float32)
```

Можемо використати форму заданої матриці, щоб зробити вектор. Матриця `m_shape` має розміри 3 × 2. Можемо створити тензор з трьох рядків, заповнених символами, за допомогою такого коду:

```
# Побудова вектора з таких самих рядків, як m_shape
print(tf.ones(m_shape.shape[0]))
```

**Результат на виході:**

```
Tensor("ones_1:0", shape=(3,), dtype=float32)
```

Якщо ми передаємо значення 1 в дужках, то можемо побудувати вектор, який дорівнює кількості стовпців у матриці `m_shape`:

```
# Побудова вектора з одиниць з такою ж кількістю стовпців, що і m_shape
print(tf.ones(m_shape.shape[1]))
```

**Результат на виході:**
```
Tensor("ones_2:0", shape=(2,), dtype=float32)
```

Отже, можемо створити матрицю 3 × 2 лише однією командою:

```
print(tf.ones(m_shape.shape))
```

**Результат на виході:**

```
Tensor("ones_3:0", shape=(3, 2), dtype=float32)
```

## Типи даних

Друга властивість тензора – тип даних. Тензор одночасно може мати лише один тип даних. Можемо визначити тип за допомогою `dtype`:

```
print(m_shape.dtype)
```

**Результат на виході:**

```
<dtype: 'int32'>
```

У деяких випадках потрібно змінити тип даних. У TF це можна зробити з використанням методу `tf.cast`.

**Приклад**

Використовуючи метод `cast`, тензор з плаваючою комою перетворюється на ціле число:

```
# Зміна типу даних
type_float = tf.constant(3.123456789, tf.float32)
type_int = tf.cast(type_float, dtype=tf.int32)
print(type_float.dtype)
print(type_int.dtype)
```

**Результат на виході:**

```
<dtype: 'float32'>
<dtype: 'int32'>
```

TF автоматично вибирає тип даних, коли при створенні тензора не вказується аргумент. TF «здогадується», які найбільш ймовірні типи даних. Наприклад, якщо передаємо текст, він «здогадається», що це рядок, а отже, перетворить його в рядок.

## Створення оператора

Ми вже знаємо, як створити тензор за допомогою TF. Настав час навчитися виконувати математичні операції.

TF містить всі основні операції. Почати можна з простого. Використаємо метод TF для обчислення квадрата числа. Ця операція проста, оскільки для побудови тензора потрібен лише один аргумент.

Квадрат числа будуємо за допомогою `tf.sqrt (x)` з **x**, тип числа з плаваючою комою:

```
x = tf.constant([2.0], dtype = tf.float32)
print(tf.sqrt(x))
```

**Результат на виході:**

```
Tensor("Sqrt:0", shape=(1,), dtype=float32)
```

***Примітка.*** Вихід повертає об'єкт тензора, а не результат квадрата 2. У прикладі виводиться визначення тензора, а не фактичний результат операції. У наступному розділі дізнаємось, як працює TF для виконання операцій.

Далі наведено список часто використовуваних операцій. Кожна операція потребує одного або декількох аргументів:

- tf.add (a, b) – додавання;
- tf.multiply (a, b) – множення;
- tf.div (a, b) – ділення;
- tf.pow (a, b) – піднесення a до степеня b;
- tf.exp (a) – експонента;
- tf.sqrt (a) – квадрат числа.

**Приклад**

```
# Додавання
tensor_a = tf.constant([[1,2]], dtype = tf.int32)
tensor_b = tf.constant([[3, 4]], dtype = tf.int32)

tensor_add = tf.add(tensor_a, tensor_b)print(tensor_add)
```

**Результат на виході:**

```
Tensor("Add:0", shape=(1, 2), dtype=int32)
```

Пояснення коду:

Створюємо і додаємо два тензори:

- один тензор з 1 і 2;
- один тензор з 3 і 4.

***Примітка.*** Обидва тензори повинні мати однакову форму.

Можемо перемножити два тензори:

```
# Множення
tensor_multiply = tf.multiply(tensor_a, tensor_b)
print(tensor_multiply)
```

**Результат на виході:**

```
Tensor("Mul:0", shape=(1, 2), dtype=int32)
```

## Змінні

Дотепер ми створювали лише тензори з константами, а це не дуже корисно. Оскільки дані завжди надходять з різними значеннями, то для їх збереження можна використати клас `Variable`. Він буде являти собою вузол, в якому значення завжди змінюються.

Для створення змінної можна використати метод `tf.get_variable()`:

```
tf.get_variable(name = "", values, dtype, initializer)
argument
- `name = ""`: Name of the variable
- `values`: Dimension of the tensor
- `dtype`: Type of data. Optional
- `initializer`: How to initialize the tensor. Optional
If initializer is specified, there is no need to include the `values` as
the shape of `initializer` is used.
```

Наприклад, наведений нижче код створює двовимірну змінну з двома випадковими значеннями. За замовчуванням TF повертає випадкове значення. Ми назвали змінну `var`:

```
# Створюємо Variable
## Створюємо 2 випадкових значення
var = tf.get_variable("var", [1, 2])
print(var.shape)
```

**Результат на виході:**

```
(1, 2)
```

У наступному прикладі створюємо змінну з одним рядком і двома стовпцями. Нам треба використати [1, 2] для створення змінної такої розмірності.

Початкові значення цього тензора дорівнюють нулю. Наприклад, коли ми тренуємо модель, то треба мати початкові значення для обчислення вагових коефіцієнтів параметрів. Нижче встановлюємо ці початкові значення в нуль:

```
var_init_1 = tf.get_variable("var_init_1", [1, 2], dtype=tf.int32,
initializer=tf.zeros_initializer)
print(var_init_1.shape)
```

**Результат на виході:**

```
(1, 2)
```

Можемо передати значення константи тензора в змінну. Створюємо тензор константи методом `tf.constant()`. Далі використовуємо цей тензор для ініціалізації змінної.

Перші значення змінної – 10, 20, 30 і 40. Новий тензор матиме форму $2 \times 2$:

```
# Створюємо матрицю 2x2
tensor_const = tf.constant([[10, 20],
                            [30, 40]])
# Ініціалізуємо перше значення тензора рівне tensor_const
var_init_2 = tf.get_variable("var_init_2", dtype=tf.int32,
initializer=tensor_const)
print(var_init_2.shape)
```

**Результат на виході:**

```
(2, 2)
```

## Заповнювач

Заповнювач має за мету живити тензор. Заповнювач використовується для ініціалізації даних, які надходять у тензори. Щоб наповнити заповнювач, потрібно скористатися методом `feed_dict`. Заповнювач буде наповнюватися лише протягом сеансу.

У наступному прикладі побачимо, як створити заповнення за допомогою методу `tf.placeholder`. На наступному сеансі навчимося наповнювати заповнювач фактичними значеннями.

Синтаксис:

```
tf.placeholder(dtype,shape=None,name=None )
arguments:
- `dtype`: Type of data
- `shape`: dimension of the placeholder. Optional. By default, shape of
the data
- `name`: Name of the placeholder. Optional
data_placeholder_a = tf.placeholder(tf.float32, name =
"data_placeholder_a")
print(data_placeholder_a)
```

**Результат на виході:**

```
Tensor("data_placeholder_a:0", dtype=float32)
```

## Сесія

TF працює з трьома основними компонентами:

- Graph;
- Tensor;
- Session.

| Компоненти | Опис |
|---|---|
| Graph | ***Граф*** є основоположним у TF. Усі математичні операції (ops) виконуються всередині графа. Можемо уявити граф як проєкт, де виконуються всі операції. Вузли представляють ці операції. Вони можуть поглинати або створювати нові тензори. |
| Tensor | ***Тензор*** представляє дані, які переміщуються між операціями. Як ініціалізувати тензор ми вже знаємо. Різниця між постійною і змінною полягає в тому, що початкові значення змінної з часом змінюватимуться. |
| Session | ***Сеанс*** виконує операцію з графом. Щоби подати граф зі значеннями тензора, потрібно відкрити сеанс. Всередині сеансу треба запустити оператор, щоб створити вихід. |

Граф і сеанси незалежні. Можемо запустити сеанс і отримати значення, які потім використати для подальших обчислень.

У наведеному далі прикладі:

- створимо два тензори;
- створимо операцію;
- відкриємо сеанс;
- виведемо результат.

**Крок 1.** Створюємо два тензори **x** і **y**

```
## Створимо, запустимо і оцінимо сеанс
x = tf.constant([2])
y = tf.constant([4])
```

**Крок 2.** Створюємо операцію множення **x** на **y**

```
## Створюємо оператор
multiply = tf.multiply(x, y)
```

**Крок 3.** Відкриваємо сеанс. Усі обчислення будуть відбуватися протягом сеансу. Коли закінчимо, то сеанс треба закрити.

```
## Створюємо сеанс для запуску коду
sess = tf.Session()result_1 = sess.run(multiply)
print(result_1)
sess.close()
```

**Результат на виході:**

```
[8]
```

Пояснення коду:

- tf.Session(): відкриття сесії: всі операції виконуються всередині сесії;
- run(multiply): виконання операції, створеної на кроці 2;
- print(result_1): завершення, виведення результату;
- close(): закриття сесії.

Результат показує 8, що є множенням **x** на **y**.

Ще один спосіб створити сеанс – це всередині блоку. Перевага способу полягає в тому, що він автоматично закриває сеанс.

```
with tf.Session() as sess:
result_2 = multiply.eval()
print(result_2)
```

**Результат на виході:**

```
[8]
```

У контексті сеансу можна використовувати метод `eval()` для виконання операції. Це еквівалентно `run()`, але робить код більш читабельним.

Можемо створити сеанс і побачити значення всередині створених раніше тензорів:

```
## Перевірка раніше створеного тензора
sess = tf.Session()
print(sess.run(r1))
print(sess.run(r2_matrix))
print(sess.run(r3_matrix))
```

**Результат на виході:**

```
1
[[1 2]
 [3 4]]
[[[1 2]
  [3 4]
  [5 6]]]
```

Змінні за замовчуванням порожні, навіть після створення тензора. Перед використанням змінної її треба ініціалізувати. Для ініціалізації значення змінної треба викликати об’єкт `tf.global_variables_initializer()`. Цей об’єкт буде явно ініціалізувати всі змінні. Це корисно, перш ніж тренувати модель.

Можемо перевірити значення створених раніше змінних. Зауважимо, що для оцінювання тензора треба використовувати `run`:

```
sess.run(tf.global_variables_initializer())
print(sess.run(var))
print(sess.run(var_init_1))
print(sess.run(var_init_2))
```

**Результат на виході:**

```
[[-0.05356491  0.75867283]]
[[0 0]]
[[10 20]
 [30 40]]
```

Можемо використати заповнювач, який створили раніше, і подати його з фактичним значенням. Дані треба передати в метод `feed_dict`.

Наприклад, беремо значення двох заповнювачів `data_placeholder_a`:

```
import numpy as np
power_a = tf.pow(data_placeholder_a, 2)
with tf.Session() as sess:
    data = np.random.rand(1, 10)
    print(sess.run(power_a, feed_dict={data_placeholder_a: data}))
# Виконано успішно.
```

Пояснення коду:

- import numpy as np: Імпорт бібліотеки numpy для створення даних;
- tf.pow(data_placeholder_a, 2): Створення ops;
- np.random.rand(1, 10): Створення випадкового масиву даних;
- feed_dict={data_placeholder_a: data}: Подаємо заповнювач з даними.

**Результат на виході:**

```
[[0.05478134 0.27213147 0.8803037  0.0398424  0.21172127 0.01444725
0.02584014 0.3763949  0.66022706 0.7565559 ]]
```

## Граф

У TF всі обчислення представлені схемою потоку даних. Граф потоку даних розроблено для відображення залежності даних між окремими операціями. Математична формула або алгоритм складаються з низки послідовних операцій. Граф – це зручний спосіб візуалізації координації обчислень.

На графі наведено вузол (**node**) і ребро (**edge**). Вузол – це представлення операції, тобто одиниці обчислення. Ребро є тензором і може виробляти новий тензор або використовувати вхідні дані. Це залежить від залежності між окремими операціями.

Структура графа з'єднує між собою операції (тобто вузли) і те, як ці операції подаються. Зауважимо, що на графі не відображається результат операцій, він лише допомагає візуалізувати зв'язок між окремими операціями.

Розглянемо приклад.

Уявіть, що нам необхідно оцінити функцію:

$$f(x,z) = xz + x^2 + z + 5$$

TF створить граф для виконання функції. Граф виглядає приблизно так, як наведено на рис. 4.3.

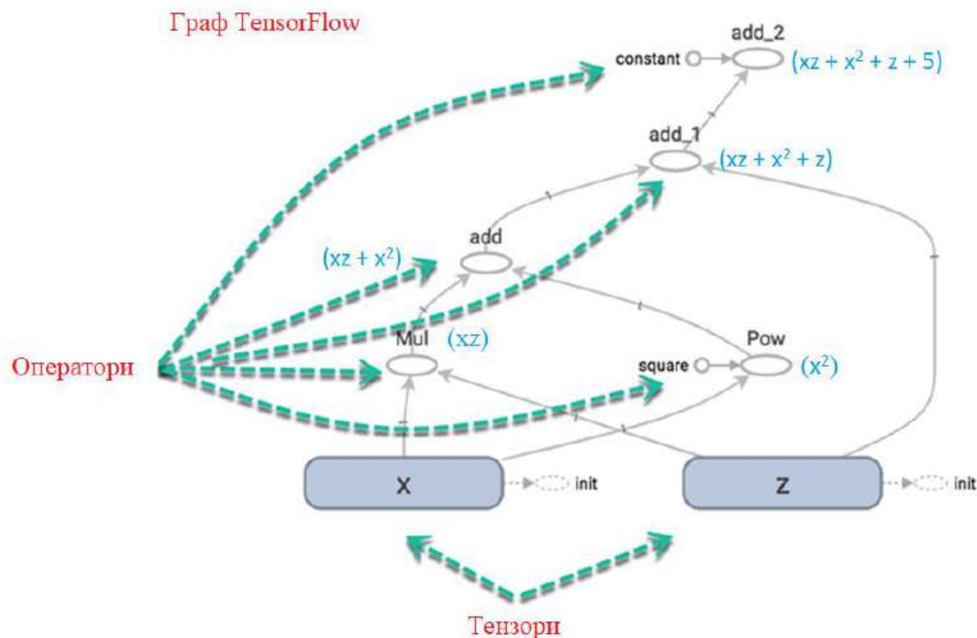



Рис. 4.3. Граф для виконання функції

З рис. 4.3 наочно видно шлях, який пройдуть тензори, щоб досягти остаточного пункту призначення.

Наприклад, очевидно, що операцію **add** не можна робити до **and**. Граф пояснює це:

- обчислити **and**; **(xz)**
- додати **add;** **(xz + x$^2$)**
- додати **add_1;** **(xz + x$^2$ + z)**
- додати **add_2.** **(xz + x$^2$ + z + 5)**

```
x = tf.get_variable("x", dtype=tf.int32, initializer=tf.constant([5]))
z = tf.get_variable("z", dtype=tf.int32, initializer=tf.constant([6]))
c = tf.constant([5], name = "constant")square = tf.constant([2], name =
"square")
f = tf.multiply(x, z) + tf.pow(x, square) + z + c
```

Пояснення коду:

- x: ініціалізація змінної з назвою x зі значенням константи 5;
- z: ініціалізація змінної z зі значенням константи 6;
- c: ініціалізація тензора константи з назвою c зі значенням константи 5;
- square: ініціалізація тензора константи з назвою square зі значенням константи 2;
- f: побудова оператора.

У цьому прикладі вибираємо збереження значень змінних константами. Ми також створили постійний тензор, що називається **c**, який є постійним параметром у функції **f**. Він має фіксоване значення **5**. На графі можна побачити цей параметр в тензорі, який називається константою.

Також побудовано постійний тензор для піднесення до степеня в операторі `tf.pow()`. Це не обов'язково, але ми зробили це для того, щоб можна було побачити назву тензора на графі (коло з назвою `square`).

З графа можна зрозуміти, що буде з тензорами і як можна отримати на виході 66.

Нижченаведений код визначає функцію в сеансі.

```
init = tf.global_variables_initializer() # prepare to initialize all
variables
with tf.Session() as sess:
    init.run() # Ініціалізація x і y
    function_result = f.eval()
    print(function_result)
```

**Результат на виході:**

```
[66]
```

**Висновки**

TF працює з:

- графом – обчислювальне середовище, що містить операції і тензори;
- тензорами – представляє дані (або значення), які будуть надходити в граф, а це ребра графа;
- сеансами – дає змогу виконання операцій.

***Створення тензора константи***

| Константа | Об'єкт |
|---|---|
| D0 | tf.constant(1, tf.int16) |
| D1 | tf.constant([1,3,5], tf.int16) |
| D2 | tf.constant([ [1, 2], [3, 4] ], tf.int16) |
| D3 | tf.constant([ [[1, 2],[3, 4], [5, 6]] ], tf.int16) |

***Створення оператора***

| Створений оператор | Об'єкт |
|---|---|
| a+b | tf.add(a, b) |
| a*b | tf.multiply(a, b) |

***Створення тензора змінної***

| Створена змінна | Об'єкт |
|---|---|
| випадкове значення | tf.get_variable("var", [1, 2]) |
| ініціалізоване перше значення | tf.get_variable("var_init_2", dtype=tf.int32, initializer=[ [1, 2], [3, 4] ]) |

***Відкриття сеансу***

| **Сеанс** | **Об’єкт** |
|---|---|
| Створити сеанс | tf.Session() |
| Запустити сеанс | tf.Session.run() |
| Виконати тензор | variable_name.eval() |
| Закрити сеанс | sess.close() |
| Сеанс за блоком | with tf.Session() as sess: |

# 5. Візуалізація графа з TensorBoard

## Що таке TensorBoard

TensorBoard – це інтерфейс, який використовується для візуалізації графа й інших інструментів для розуміння, налагодження і оптимізації моделі[11] [11].

### Приклад

Нижченаведене зображення (рис. 5.1) використовує граф, який згенеруємо в цьому розділі. Це основна панель.

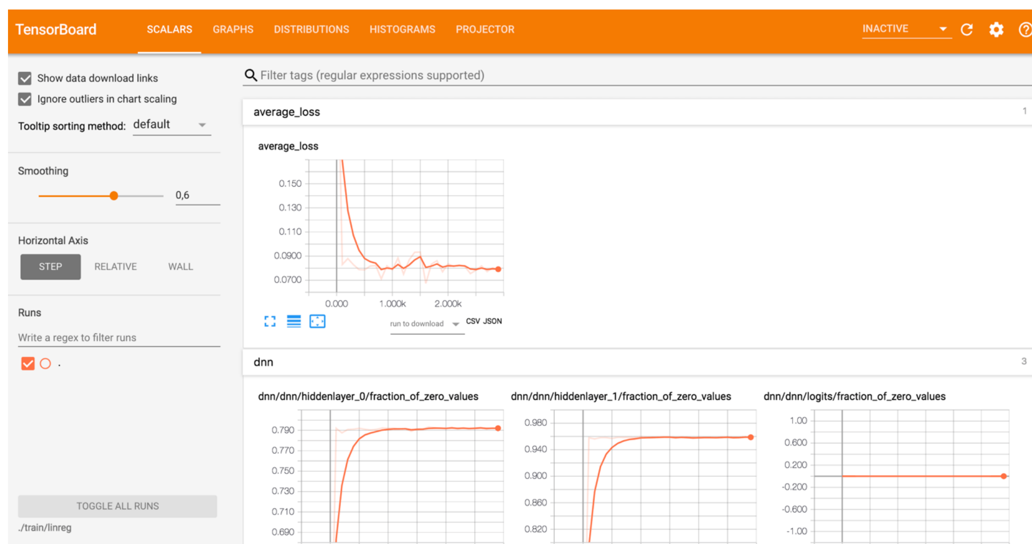



Рис. 5.1. Основна панель TensorBoard

На рис. 5.2 наведено елементи панелі TensorBoard. Панель містить різні вкладки, які пов’язані з рівнем інформації, яку ми додаємо під час запуску моделі.

Рис. 5.2. Вкладки панелі TensorBoard

- Скаляри: відображається різна корисна інформація під час тренування моделі.
- Графи: відображається модель.

[11] https://www.guru99.com/tensorboard-tutorial.html

- Розподіли: відображається розподіл вагових коефіцієнтів.
- Гістограми: відображаються вагові коефіцієнти за допомогою гістограми.
- Проєктор: показує аналіз основних компонентів і алгоритм T-SNE. Методика використовується для зменшення розмірності.

В цьому розділі навчимо просту модель глибокого навчання.

Якщо подивитися на граф (рис. 5.3), то можна зрозуміти, як працює модель:

1. Задаємо дані для моделі (покажемо на моделі кількість даних, що дорівнює розміру партії, тобто кількості поданих даних після кожної ітерації).
2. Передаємо дані тензорам.
3. Виконаємо тренування моделі.
4. Покажемо кількість партій під час тренінгу.
5. Збережемо модель на диску.

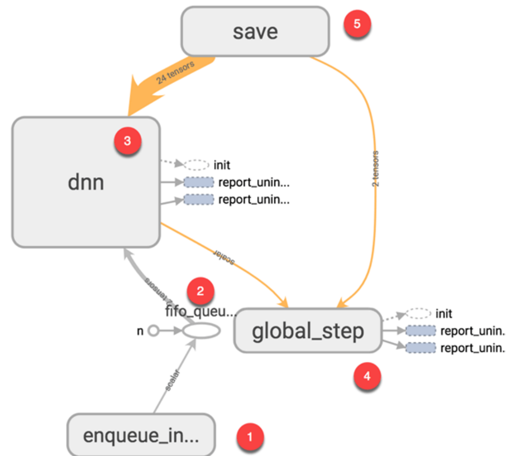



Рис. 5.3. Граф моделі глибокого навчання

Основна ідея TensorBoard полягає в тому, що нейронна мережа може сприйматися як чорна скринька, а нам треба мати інструмент для перевірки того, що є всередині цієї скриньки. Можна уявити TensorBoard як ліхтарик, за допомогою якого можна почати занурення в нейронну мережу.

Це допомагає зрозуміти залежності між операціями, як обчислюються вагові коефіцієнти, відображається функція втрат і багато іншої корисної інформації. Коли збирається все це разом, то маємо чудовий інструмент для налагодження і пошуку способів вдосконалення моделі.

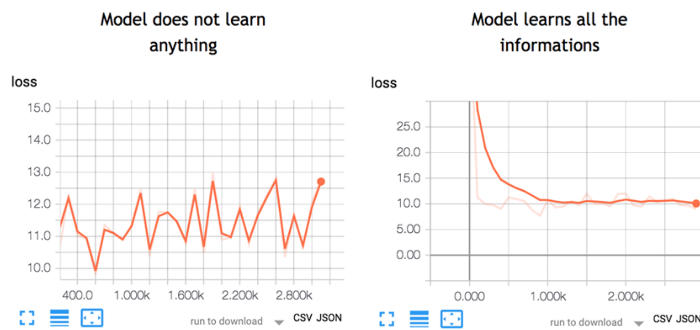



Рис. 5.4. Модель не вчиться і модель вчиться

Наскільки корисним може бути граф наочно можна побачити на рис. 5.4.

Нейронна мережа вирішує, як з’єднати різні «нейрони» і скільки треба шарів, перш ніж модель зможе передбачити результат. Після того як визначили архітектуру, треба не

тільки підготувати модель, але й метрику для обчислення точності прогнозування. Цей показник називається **функцією втрат**. Мінімізація функції втрат і є метою. Отже, це означає, що модель робить менше помилок. Усі алгоритми машинного навчання багато разів повторюють обчислення, поки втрати не досягнуть більш плоскої кривої. Щоб мінімізувати функцію втрат, треба визначити **рівень навчання**. Це швидкість, яку ми хочемо мати для навчання моделі. Якщо визначити рівень навчання занадто високий, то модель не встигне нічого навчитися. На рис. 5.4 цей випадок наведено на лівому зображенні. Крива рухається вгору і вниз, тобто модель прогнозує результат з абсолютною здогадкою, а праворуч видно, що втрати за ітерацію зменшуються, допоки крива не стає більш плоскою, тобто модель знайшла рішення.

TensorBoard – чудовий інструмент для візуалізації таких показників і висвітлення потенційних проблем. Навчання нейронної мережі може здійснюватися від години до тижнів, перш ніж знайдеться рішення. TensorBoard оновлює показники дуже часто. У цьому випадку не потрібно чекати до кінця, щоб побачити, чи правильно навчається модель. Можна відкрити TensorBoard, щоб перевірити перебіг процесу навчання, а у разі необхідності зробити відповідні зміни.

## Як користуватися TensorBoard

У цьому розділі дізнаємось, як відкрити TensorBoard з командного рядка Windows.

Насамперед треба імпортувати бібліотеки, які будемо використовувати під час навчання:

```
## Імпорт бібліотеки
import tensorflow as tf
import numpy as np
```

Створимо дані. Це масив із 10 000 рядків і 5 стовпців:

```
X_train = (np.random.sample((10000,5)))
y_train =  (np.random.sample((10000,1)))
X_train.shape
```

**Результат на виході:**

```
(10000, 5)
```

Нижченаведені коди перетворюють дані і створюють модель.

Звертаємо увагу, що рівень навчання дорівнює 0,1. Якщо змінити цей показник на більше значення, модель не знайде рішення, а це те, що сталося у випадку, який наведено на рис. 5.4 ліворуч.

У більшості навчальних розділів з TF будемо використовувати оцінювач TensorFlow. Це API TensorFlow, який містить усі математичні обчислення.

Щоби створити файли журналів, треба вказати шлях. Це робиться за допомогою аргументу `model_dir`.

У наведеному прикладі модель зберігаємо всередині робочого каталогу, тобто там, де зберігаємо блокнот і файл python. Всередині цього шляху TF створить папку під назвою `train` з іменем дочірньої папки `linreg`.

```
feature_columns = [
      tf.feature_column.numeric_column('x', shape=X_train.shape[1:])]
DNN_reg = tf.estimator.DNNRegressor(feature_columns=feature_columns,
```

```
# Показує, коли зберігається лог файл
     model_dir='train/linreg',
     hidden_units=[500, 300],
     optimizer=tf.train.ProximalAdagradOptimizer(
          learning_rate=0.1,
          l1_regularization_strength=0.001
      )
)
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'train/linreg',
'_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1818e63828>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Останній крок полягає у підготовці моделі. Під час навчання TF записує інформацію в каталог моделей.

```
# Тренування оцінювача
train_input = tf.estimator.inputs.numpy_input_fn(
     x={"x": X_train},
     y=y_train, shuffle=False,num_epochs=None)
DNN_reg.train(train_input,steps=3000)
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into train/linreg/model.ckpt.
INFO:TensorFlow:loss = 40.060104, step = 1
INFO:TensorFlow:global_step/sec: 197.061
INFO:TensorFlow:loss = 10.62989, step = 101 (0.508 sec)
INFO:TensorFlow:global_step/sec: 172.487
INFO:TensorFlow:loss = 11.255318, step = 201 (0.584 sec)
INFO:TensorFlow:global_step/sec: 193.295
INFO:TensorFlow:loss = 10.604872, step = 301 (0.513 sec)
INFO:TensorFlow:global_step/sec: 175.378
INFO:TensorFlow:loss = 10.090343, step = 401 (0.572 sec)
INFO:TensorFlow:global_step/sec: 209.737
INFO:TensorFlow:loss = 10.057928, step = 501 (0.476 sec)
INFO:TensorFlow:global_step/sec: 171.646
INFO:TensorFlow:loss = 10.460144, step = 601 (0.583 sec)
INFO:TensorFlow:global_step/sec: 192.269
INFO:TensorFlow:loss = 10.529617, step = 701 (0.519 sec)
INFO:TensorFlow:global_step/sec: 198.264
```

```
INFO:TensorFlow:loss = 9.100082, step = 801 (0.504 sec)
INFO:TensorFlow:global_step/sec: 226.842
INFO:TensorFlow:loss = 10.485607, step = 901 (0.441 sec)
INFO:TensorFlow:global_step/sec: 152.929
INFO:TensorFlow:loss = 10.052481, step = 1001 (0.655 sec)
INFO:TensorFlow:global_step/sec: 166.745
INFO:TensorFlow:loss = 11.320213, step = 1101 (0.600 sec)
INFO:TensorFlow:global_step/sec: 161.854
INFO:TensorFlow:loss = 9.603306, step = 1201 (0.619 sec)
INFO:TensorFlow:global_step/sec: 179.074
INFO:TensorFlow:loss = 11.110269, step = 1301 (0.556 sec)
INFO:TensorFlow:global_step/sec: 202.776
INFO:TensorFlow:loss = 11.929443, step = 1401 (0.494 sec)
INFO:TensorFlow:global_step/sec: 144.161
INFO:TensorFlow:loss = 11.951693, step = 1501 (0.694 sec)
INFO:TensorFlow:global_step/sec: 154.144
INFO:TensorFlow:loss = 8.620987, step = 1601 (0.649 sec)
INFO:TensorFlow:global_step/sec: 151.094
INFO:TensorFlow:loss = 10.666125, step = 1701 (0.663 sec)
INFO:TensorFlow:global_step/sec: 193.644
INFO:TensorFlow:loss = 11.0349865, step = 1801 (0.516 sec)
INFO:TensorFlow:global_step/sec: 189.707
INFO:TensorFlow:loss = 9.860596, step = 1901 (0.526 sec)
INFO:TensorFlow:global_step/sec: 176.423
INFO:TensorFlow:loss = 10.695, step = 2001 (0.567 sec)
INFO:TensorFlow:global_step/sec: 213.066
INFO:TensorFlow:loss = 10.426752, step = 2101 (0.471 sec)
INFO:TensorFlow:global_step/sec: 220.975
INFO:TensorFlow:loss = 10.594796, step = 2201 (0.452 sec)
INFO:TensorFlow:global_step/sec: 219.289
INFO:TensorFlow:loss = 10.4212265, step = 2301 (0.456 sec)
INFO:TensorFlow:global_step/sec: 215.123
INFO:TensorFlow:loss = 9.668612, step = 2401 (0.465 sec)
INFO:TensorFlow:global_step/sec: 175.65
INFO:TensorFlow:loss = 10.009649, step = 2501 (0.569 sec)
INFO:TensorFlow:global_step/sec: 206.962
INFO:TensorFlow:loss = 10.477722, step = 2601 (0.483 sec)
INFO:TensorFlow:global_step/sec: 229.627
INFO:TensorFlow:loss = 9.877638, step = 2701 (0.435 sec)
INFO:TensorFlow:global_step/sec: 195.792
INFO:TensorFlow:loss = 10.274586, step = 2801 (0.512 sec)
INFO:TensorFlow:global_step/sec: 176.803
INFO:TensorFlow:loss = 10.061047, step = 2901 (0.566 sec)
INFO:TensorFlow:Saving checkpoints for 3000 into
train/linreg/model.ckpt.
INFO:TensorFlow:Loss for final step: 10.73032.

<TensorFlow.python.estimator.canned.dnn.DNNRegressor at 0x1818e63630>
```

Можна побачити цю інформацію в TensorBoard (рис. 5.5).

Отже, коли записані події журналу, можна відкрити TensorBoard. TensorBoard працює на порту 6006 (Jupyter працює на порту 8888). У Windows можна скористатися підказкою Anaconda.

```
cd C:\Users\Admin\Anaconda3
activate hello-tf
```

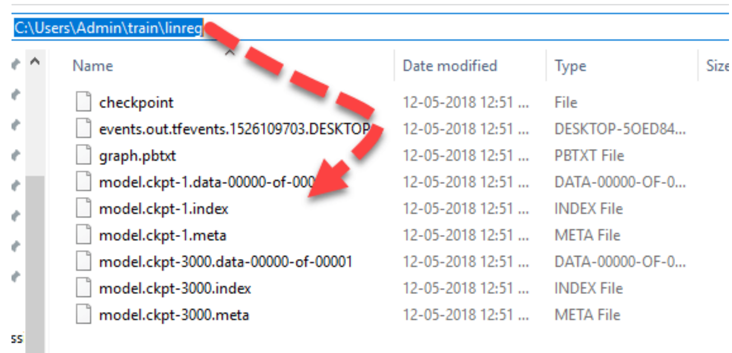



Рис. 5.5. Папка з файлом журналу

Блокнот зберігається за шляхом C:\Users\Admin\Anaconda3.
Щоб запустити TensorBoard, можна використати код:

```
tensorboard --logdir=.\train\linreg
```

TensorBoard розташований за URL-адресою: http://localhost:6006
Він також може бути розташований у такому місці (рис. 5.6).

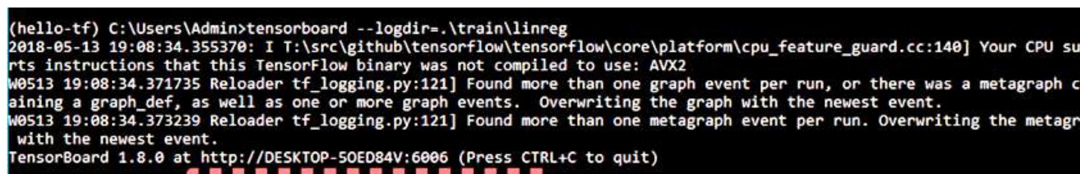



Рис. 5.6. Розташування TensorBoard

Скопіюємо і вставимо URL у браузер. Це можна побачити на рис. 5.7.

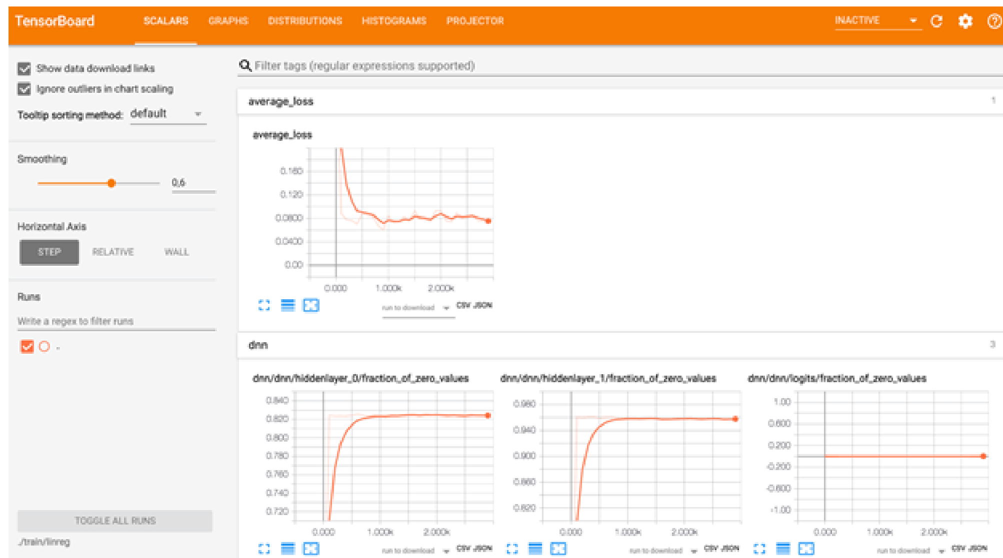



Рис. 5.7. TensorBoard

Іноді можна побачити щось подібне (рис. 5.8).

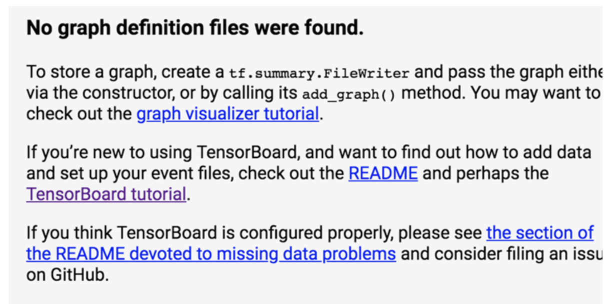



Рис. 5.8. Повідомлення про помилку

Це означає, що TensorBoard не може знайти файл журналу. Переконайтеся, що в команді `cd` вказано правильний шлях або двічі перевірте, чи створювалась подія журналу. Якщо ні, то запустіть код повторно.

Якщо необхідно закрити TensorBoard, натисніть Ctrl+c.

***Порада.*** Перевірте свій запит Anaconda для поточного робочого каталогу (рис. 5.9).

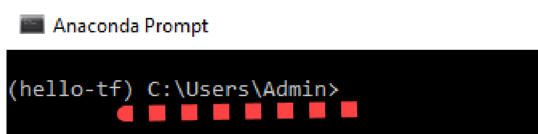



Рис. 5.9. Запит Anaconda

Файл журналу слід створити в C:\Users\Admin

**Висновки**

TensorBoard – чудовий інструмент для візуалізації вашої моделі. Крім того, під час тренінгу відображається багато показників, таких як втрата, точність або вагові коефіцієнти.

Щоб активувати TensorBoard, треба встановити шлях до файлу:

```
cd /Users/Guru99/tuto_TF
```

Активація оточення TensorFlow:

```
activate hello-tf
```

Запуск TensorBoard:

```
tensorboard --logdir=.+ PATH
```

# 6. Pandas Python

## Що таке Pandas і чому його використовують

Pandas – це бібліотека з відкритим джерелом, яка дає змогу здійснювати маніпуляції з даними в Python[12] [12]. Бібліотека Pandas побудована зверху Numpy, тобто для роботи Pandas

[12] https://www.guru99.com/python-pandas-tutorial.html

треба Numpy. Pandas надає простий спосіб, щоб створювати, маніпулювати і змінювати дані. Pandas – це також елегантне рішення для часових рядів.

Вчені, які працюють з даними, отримують з Pandas такі переваги:

- легко опрацьовувати відсутність даних;
- використовуються серії для одновимірної структури даних і DataFrame для багатовимірної структури даних;
- забезпечується ефективний спосіб фрагментації даних;
- забезпечується гнучкий спосіб об'єднання, конкатенції або переформатування даних;
- маємо потужний інструмент для роботи з часовим рядом.

Якщо коротко, то Pandas – це корисна бібліотека для аналізу даних. За допомогою цієї бібліотеки можна виконувати маніпуляції і аналіз даних. Pandas надає потужні і прості у використанні структури даних, а також засоби для швидкого виконання операцій над цими структурами.

## Встановлення Pandas

Щоб встановити бібліотеку Pandas, перегляньте розділ ***Як встановити TensorFlow***. Pandas встановлюється за замовчуванням. Якщо ж чомусь Pandas не встановлено, то можемо встановити Pandas за допомогою:

- anaconda: conda install -c anaconda pandas;
- у блокноті Jupyter:

```
import sys
!conda install --yes --prefix {sys.prefix} pandas
```

## Кадр даних і серія

Кадр даних – це двовимірний масив з міченими осями (рядки і стовпці), а також це стандартний спосіб зберігання даних.

Кадр даних добре відомий статистикам й іншим практикам з даних. Кадр даних – це табличні дані з рядками для зберігання інформації, а також стовпцями для іменування інформації. Наприклад, назвою стовпця може бути ціна, 2, 3, 4 – значення ціни.

Нижче наведено зображення кадру даних Pandas:

| | Item | Price |
|---|---|---|
| **0** | A | 2 |
| **1** | B | 3 |

Серія – це одновимірна структура даних. Вона може мати будь-який тип даних: цілі, з плаваючою комою, а також рядкові. Це корисно, коли потрібно виконати обчислення або повернути одновимірний масив. Серія, за визначенням, не може мати більше одного стовпця. Для багатьох стовпців рекомендовано використовувати структуру кадра даних.

У серії є один параметр:

- дані – може бути список, словник або скалярне значення.

```
pd.Series([1., 2., 3.])
0    1.0
1    2.0
2    3.0
dtype: float64
```

Можна додати індекс з індексом. Це допомагає назвати рядки. Довжина має дорівнювати розміру стовпчика

```
pd.Series([1., 2., 3.], index=['a', 'b', 'c'])
```

Створимо серію Pandas з відсутнім значенням для третього рядка. Звертаємо увагу, що відсутні значення в Python позначаються «`NaN`». Можемо використати numpy, щоб штучно створити відсутнє значення: `np.nan`

```
pd.Series([1,2,np.nan])
```

**Результат на виході:**

```
0    1.0
1    2.0
2    NaN
dtype: float64
```

## Створення кадра даних

Можна перетворити масив numpy у кадр даних pandas за допомогою `pd.Data frame()`. Можна зробити і протилежне. Для перетворення кадра даних pandas (Data Frame) в масив можна використати `np.array()`

```
## з Numpy в Pandas
import numpy as np
h = [[1,2],[3,4]]
df_h = pd.DataFrame(h)
print('Data Frame:', df_h)

## з Pandas в Numpy
df_h_n = np.array(df_h)
print('Numpy array:', df_h_n)
Data Frame:    0  1
0  1  2
1  3  4
Numpy array: [[1 2]
 [3 4]]
```

Також можна скористатися словником для створення кадра даних Pandas.

```
dic = {'Name': ["John", "Smith"], 'Age': [30, 40]}
pd.DataFrame(data=dic)
```

| | **Age** | **Name** |
|---|---|---|
| **0** | 30 | John |
| **1** | 40 | Smith |

## Діапазон дат

Pandas має зручний API для створення діапазону дат

```
pd.data_range(date,period,frequency):
```

- перший параметр – дата початку;
- другий параметр – кількість періодів (необов’язково, якщо вказана кінцева дата);
- останній параметр – частота: день: 'D', місяць: 'M' та рік: 'Y'.

```
## Створення дати
# Дні
dates_d = pd.date_range('20300101', periods=6, freq='D')
print('Day:', dates_d)
```

**Результат на виході:**

```
Day: DatetimeIndex(['2030-01-01', '2030-01-02', '2030-01-03', '2030-01-
04', '2030-01-05', '2030-01-06'], dtype='datetime64[ns]', freq='D')
```

```
# Місяці
dates_m = pd.date_range('20300101', periods=6, freq='M')
print('Month:', dates_m)
```

**Результат на виході:**

```
Month: DatetimeIndex(['2030-01-31', '2030-02-28', '2030-03-31', '2030-
04-30','2030-05-31', '2030-06-30'], dtype='datetime64[ns]', freq='M')
```

## Перевірка даних

Можна перевірити заголовок й хвіст набору даних за допомогою `head()` або `tail()`, які йдуть попереду назви кадра даних pandas.

**Крок 1.** Створимо випадкову послідовність з numpy. Послідовність має 4 стовпчики і 6 рядків:

```
random = np.random.randn(6,4)
```

**Крок 2.** Створимо кадр даних, використовуючи pandas.

Використаємо `dates_m` як індекс для кадра даних. Це означає, що кожен рядок буде мати «name» або індекс згідно з датою.

Отже, дамо назви чотирьом стовпцям зі стовпцями аргументів:

```
# Створення даних з датою
df = pd.DataFrame(random,
                  index=dates_m,
                  columns=list('ABCD'))
```

**Крок 3.** Використаємо функції заголовка

```
df.head(3)
```

| | **A** | **B** | **C** | **D** |
|---|---|---|---|---|
| **2030-01-31** | 1.139433 | 1.318510 | -0.181334 | 1.615822 |
| **2030-02-28** | -0.081995 | -0.063582 | 0.857751 | -0.527374 |
| **2030-03-31** | -0.519179 | 0.080984 | -1.454334 | 1.314947 |

**Крок 4.** Використаємо функцію хвоста

```
df.tail(3)
```

| | A | B | C | D |
|---|---|---|---|---|
| **2030-04-30** | -0.685448 | -0.011736 | 0.622172 | 0.104993 |
| **2030-05-31** | -0.935888 | -0.731787 | -0.558729 | 0.768774 |
| **2030-06-30** | 1.096981 | 0.949180 | -0.196901 | -0.471556 |

**Крок 5.** Чудовою практикою для отримання поняття про дані є використання `describe()`. Команда забезпечує підрахунок, середнє значення, std, min, max і процентиль набору даних.

```
df.describe()
```

| | A | B | C | D |
|---|---|---|---|---|
| **count** | 6.000000 | 6.000000 | 6.000000 | 6.000000 |
| **mean** | 0.002317 | 0.256928 | -0.151896 | 0.467601 |
| **std** | 0.908145 | 0.746939 | 0.834664 | 0.908910 |
| **min** | -0.935888 | -0.731787 | -1.454334 | -0.527374 |
| **25%** | -0.643880 | -0.050621 | -0.468272 | -0.327419 |
| **50%** | -0.300587 | 0.034624 | -0.189118 | 0.436883 |
| **75%** | 0.802237 | 0.732131 | 0.421296 | 1.178404 |
| **max** | 1.139433 | 1.318510 | 0.857751 | 1.615822 |

## Фрагментація даних

Розглянемо, як фрагментувати кадр даних pandas.

Можемо використати назву стовпця для збереження даних у певному стовпці:

```
## Фрагмент. Використання імені
df['A']

2030-01-31   -0.168655
2030-02-28    0.689585
2030-03-31    0.767534
2030-04-30    0.557299
2030-05-31   -1.547836
2030-06-30    0.511551
Freq: M, Name: A, dtype: float64
```

Для вибору декількох стовпців треба скористатись дужкою двічі, [[.., ..]]. Перша пара дужок означає, що нам треба вибрати стовпці, а друга – вказує, які стовпці треба повернути.

```
df[['A', 'B']].
```

| | A | B |
|---|---|---|
| **2030-01-31** | -0.168655 | 0.587590 |
| **2030-02-28** | 0.689585 | 0.998266 |
| **2030-03-31** | 0.767534 | -0.940617 |
| **2030-04-30** | 0.557299 | 0.507350 |
| **2030-05-31** | -1.547836 | 1.276558 |
| **2030-06-30** | 0.511551 | 1.572085 |

Можемо фрагментувати рядки за допомогою коду, який повертає перші три рядки:

```
### Використання фрагмента для рядка
df[0:3]
```

| | A | B | C | D |
|---|---|---|---|---|
| **2030-01-31** | -0.168655 | 0.587590 | 0.572301 | -0.031827 |
| **2030-02-28** | 0.689585 | 0.998266 | 1.164690 | 0.475975 |
| **2030-03-31** | 0.767534 | -0.940617 | 0.227255 | -0.341532 |

Функція `loc` використовується для вибору стовпців за назвами. Як завжди, значення перед комою належать до рядків, а після неї – до стовпця. Треба користуватися дужками, щоб вибрати більше одного стовпця.

```
## Багато стовпців
df.loc[:,['A','B']]
```

| | A | B |
|---|---|---|
| **2030-01-31** | -0.168655 | 0.587590 |
| **2030-02-28** | 0.689585 | 0.998266 |
| **2030-03-31** | 0.767534 | -0.940617 |
| **2030-04-30** | 0.557299 | 0.507350 |
| **2030-05-31** | -1.547836 | 1.276558 |
| **2030-06-30** | 0.511551 | 1.572085 |

Відомий ще один спосіб вибору декількох рядків і стовпців у Pandas. Можна використати `iloc[]`. Цей метод використовує індекс замість назви стовпців. Код нижче повертає той самий кадр даних, що і вище:

```
df.iloc[:, :2]
```

| | A | B |
|---|---|---|
| **2030-01-31** | -0.168655 | 0.587590 |
| **2030-02-28** | 0.689585 | 0.998266 |
| **2030-03-31** | 0.767534 | -0.940617 |
| **2030-04-30** | 0.557299 | 0.507350 |
| **2030-05-31** | -1.547836 | 1.276558 |
| **2030-06-30** | 0.511551 | 1.572085 |

**Видалення стовпця**

Можемо видаляти стовпці, використовуючи `pd.drop()`:

```
df.drop(columns=['A', 'C'])
```

| | B | D |
|---|---|---|
| **2030-01-31** | 0.587590 | -0.031827 |
| **2030-02-28** | 0.998266 | 0.475975 |

| | | |
|---|---|---|
| **2030-03-31** | -0.940617 | -0.341532 |
| **2030-04-30** | 0.507350 | -0.296035 |
| **2030-05-31** | 1.276558 | 0.523017 |
| **2030-06-30** | 1.572085 | -0.594772 |

**Конкатенція**

Можна об'єднати два DataFrame у Pandas. Для цього можна використати `pd.concat()`

Передусім треба створити два DataFrames, а отже, добре, що ми вже ознайомлені зі створенням кадрів даних

```
import numpy as np
df1 = pd.DataFrame({'name': ['John', 'Smith','Paul'],
                     'Age': ['25', '30', '50']},
                    index=[0, 1, 2])
df2 = pd.DataFrame({'name': ['Adam', 'Smith' ],
                     'Age': ['26', '11']},
                    index=[3, 4])
```

Тепер можемо об'єднати два DataFrame

```
df_concat = pd.concat([df1,df2])
df_concat
```

| | **Age** | **name** |
|---|---|---|
| **0** | 25 | John |
| **1** | 30 | Smith |
| **2** | 50 | Paul |
| **3** | 26 | Adam |
| **4** | 11 | Smith |

**Видалення дублікатів**

Якщо набір даних містить дублікати інформації, то можна скористатися `drop_duplicates`, щоб легко виключити рядки, які повторюються. У наведеному прикладі можна побачити, що у `df_concat` є повторюване спостереження: `Smith` відображається двічі у колонці `name`.

```
df_concat.drop_duplicates('name')
```

| | **Age** | **name** |
|---|---|---|
| **0** | 25 | John |
| **1** | 30 | Smith |
| **2** | 50 | Paul |
| **3** | 26 | Adam |

**Сортування значень**

Значення можна сортувати за допомогою `sort_values`:

```
df_concat.sort_values('Age')
```

| | Age | name |
|---|---|---|
| 4 | 11 | Smith |
| 0 | 25 | John |
| 3 | 26 | Adam |
| 1 | 30 | Smith |
| 2 | 50 | Paul |

**Перейменування: зміна індексу**

Для перейменування стовпця в Pandas можна скористатися `rename`. Перше значення – це поточна назва стовпця, а друге – нова його назва.

```
df_concat.rename(columns={"name": "Surname", "Age": "Age_ppl"})
```

| | Age_ppl | Surname |
|---|---|---|
| 0 | 25 | John |
| 1 | 30 | Smith |
| 2 | 50 | Paul |
| 3 | 26 | Adam |
| 4 | 11 | Smith |

## Імпорт даних CSV за допомогою Pandas.read_csv()

Вивчаючи TF, будемо використовувати набір даних про доросле населення[13] [13]. Його часто використовують із завданням класифікації.

Дані зберігаються у форматі CSV. Цей набір даних включає вісім категорій змінних:

- workclass;
- education;
- marital;
- occupation;
- relationship;
- race;
- sex;
- native_country.

Крім того, є шість безперервних змінних:

- age;
- fnlwgt;
- education_num;
- capital_gain;
- capital_loss;
- hours_week.

Щоб імпортувати набір даних CSV, можемо використати об’єкт `pd.read_csv()`. Основний аргумент всередині. Синтаксис:

[13] https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data

```
pandas.read_csv(filepath_or_buffer,sep=',
',`names=None`,`index_col=None`,`skipinitialspace=False`)
```

- filepath_or_buffer: шлях або URL-адреса даних;
- sep=', ': визначення роздільника, який слід використовувати;
- `names = None': назви стовпців (якщо в наборі даних є десять стовпців, нам треба передати десять назв);
- `index_col = None': якщо таке визначення, то перший стовпець використовується як індекс рядків;
- `skipinitialspace=False': пропуск пробілів після роздільника.

Для отримання додаткової інформації про `readcsv()` слід звертатися до офіційної документації[14] [14].

**Висновки**

Наведено зібрані найбільш корисні методи вивчення даних з Pandas.

| | |
|---|---|
| Імпорт даних | `read_csv` |
| Створення серій | `Series` |
| Створення Dataframe | `DataFrame` |
| Створення діпазону дат | `date_range` |
| Повернення заголовка | `head` |
| Повернення хвоста | `tail` |
| Опис | `describe` |
| Фрагментація з використанням назви | `dataname['columnname']` |
| Фрагментація з використанням рядків | `data_name[0:5]` |

# 7. Лінійна регресія з TensorFlow

TF надає інструменти для повного контролю над обчисленнями[15] [15]. Це робиться за допомогою API низького рівня. Крім того, TF має широкий набір API для виконання багатьох алгоритмів машинного навчання. Це API високого рівня. TF називає їх оцінювачами.

- API низького рівня: побудова архітектури, оптимізація моделі з нуля. Це складно для початківця.
- API високого рівня: визначення алгоритму. Це зручно. TF надає інструментарій виклику оцінювача для побудови, навчання, оцінювання і прогнозування.

У цьому розділі використаємо лише оцінювач. Розрахунки можна здійснювати швидше і легше. Перша частина розділу пояснює, як використовувати оптимізатор градієнтного спуску для тренування лінійної регресії. У другій частині використаємо набір даних Бостона[16] [16], щоб передбачити ціну будинку за допомогою оцінювача TF.

---

[14] https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html
[15] https://www.guru99.com/linear-regression-tensorflow.html
[16] https://drive.google.com/uc?export=download&id=10I5aboiafcqf0iVWnd1SiAEVbgWzgkW4

## Тренування моделі лінійної регресії

Перш ніж почнемо тренувати модель, розглянемо що таке лінійна регресія.

Уявіть, що є дві змінні, **x** і **y**, і нашим завданням буде передбачити нове значення, вже знаючи деякі значення. Якщо побудувати діаграму для даних (рис. 7.1), то можна побачити позитивну залежність між нашою незалежною змінною **x** і залежною змінною **y**.

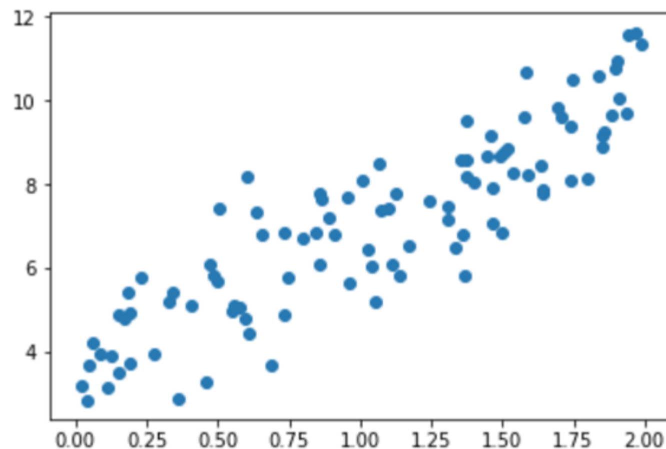



Рис.7.1. Діаграма для даних

Це не дуже точний метод, особливо з набором даних із сотнями тисяч точок.

Лінійна регресія оцінюється рівнянням. Змінна **y** пояснюється одним або багатьма коваріатами (незалежними змінними). У нашому прикладі є лише одна залежна змінна. Якщо треба написати рівняння, то це буде:

$$y = \beta + \alpha X + \epsilon \ ,$$

де

- $\beta$– упередженість, тобто, якщо x = 0, y = $\beta$;
- $\alpha$ – вага, пов'язана з **x;**
- $\epsilon$ – залишок або помилка моделі, яка включає те, що модель не може дізнатися з даних.

Припустимо, що вам підходить модель, а отже, знаходимо таке рішення для:

- $\beta$ = 3,8;
- $\alpha$ = 2,78.

Можемо підставити ці числа в рівняння і воно виглядатиме так:

$$y = 3{,}8 + 2{,}78x.$$

Тепер маємо кращий спосіб знайти значення для **y,** а отже, можемо замінити **x** будь-яким значенням, з яким хочемо передбачити **y**. На зображенні (рис. 7.2) підставили **x** у рівнянні для всіх значень набору даних і побудували результат.

Червона лінія являє собою прилаштоване значення, тобто значення **y** для кожного значення **x**. Нам не треба бачити значення **x**, щоб передбачити **y**, для кожного **x** воно вже є і належить до червоної лінії. Також можемо передбачити значення для **x** більше 2!

Якщо хочемо розширити лінійну регресію на більше коваріатів, то можна додати в модель більше змінних. Різниця між традиційним аналізом і лінійною регресією полягає в тому, що лінійна регресія розглядає, як **y** буде реагувати на кожну змінну **x**, взяту незалежно.

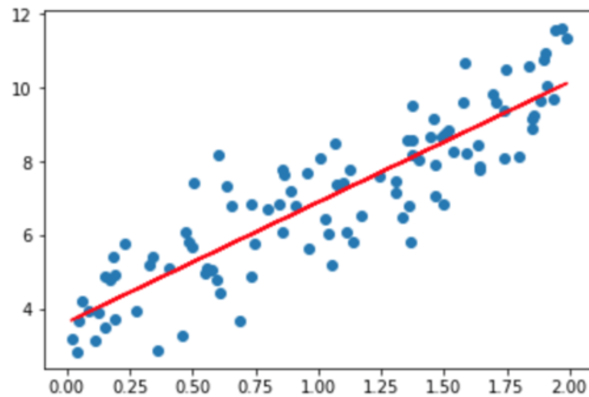



Рис.7.2. Значення y із рівняння для кожного x

Розглянемо приклад. Уявіть, що ми хочемо передбачити продажі магазину морозива. Набір даних містить різну інформацію, таку як погода (тобто дощова, сонячна, хмарна), інформацію про клієнтів (тобто зарплата, стать, сімейний стан).

Традиційний аналіз спробує передбачити продаж, наприклад, обчисливши середнє значення для кожної змінної, й спробує оцінити продаж за різними сценаріями. Це призведе до поганих прогнозів і обмежить аналіз обраним сценарієм.

Якщо ж скористаємося лінійною регресією, то ми зможемо написати таке рівняння:

$$Sales = \beta + \alpha_1 weather + \alpha_2 salary + \alpha_3 gender + \alpha_4 marital + \epsilon.$$

Алгоритм знайде найкраще рішення для вагових коефіцієнтів. Це означає, що за допомогою них ми намагатимемося мінімізувати втрати (різницю між встановленою лінією і точками даних).

**Як працює алгоритм**

Алгоритм вибере випадкове число для кожного $\beta$ і $\alpha$, а також замінить значення **x**, щоб отримати передбачуване значення **y**. Якщо в наборі даних є 100 спостережень, тоді алгоритм обчислить 100 передбачених значень.

Можна обчислити помилку $\epsilon$, визначену моделлю, яка є різницею між прогнозованим значенням і реальним значенням. Позитивна помилка означає, що модель недооцінює передбачення **y**, а негативна помилка означає, що модель завищує прогноз **y**:

$$\epsilon = y - ypred .$$

Наша мета – мінімізувати квадрат помилки. Алгоритм обчислює середнє значення квадратичної помилки. Цей крок називається мінімізацією помилки. Для лінійної регресії – середня квадратична помилка, яка також називається MSE (**Mean Square Error**).

Математично це:

$$MSE(X) = \frac{1}{m}\sum_{i=1}^{m}(\theta^T x^i - y^i)^2 ,$$

де

- $\theta^T$– ваговий коефіцієнт, тому $\theta^T x^i$ належить до передбачуваного значення;
- y – реальне значення;
- m – кількість спостережень.

Рис.7.3. Алгоритм моделі лінійної регресії

Зауважимо, що $\theta^T$ означає використання транспортування матриці; $\frac{1}{m}\Sigma$ є математичним визначенням середнього значення.

Метою є визначення кращого $\theta$, щоб мінімізувати MSE (рис. 7.3).

Якщо середня помилка велика, це означає, що модель працює погано, а вагові коефіцієнти не вибрані належним чином. Для корекції вагових коефіцієнтів потрібно скористатися оптимізатором. Традиційний оптимізатор називається градієнтним спуском (Gradient Descent).

Градієнтний спуск обчислює похідну і зменшує або збільшує ваговий коефіцієнт. Якщо похідна додатна – він зменшується. Якщо похідна від'ємна – ваговий коефіцієнт збільшується. Модель буде оновлювати вагові коефіцієнти і перераховувати помилку. Цей процес буде повторюватися допоки помилка не перестане змінюватися. Кожен цикл обчислення помилки називається ітерацією. Крім того, градієнти множать на коефіцієнт навчання. Він вказує на швидкість навчання.

Якщо швидкість навчання занадто мала, алгоритм витрачає дуже багато часу (тобто треба багато ітерацій). Якщо швидкість навчання занадто висока, алгоритм може ніколи не зійтися.

Як бачимо на рис. 7.4, модель повторює процес приблизно 20 разів, перш ніж знайти стабільне значення для вагових коефіцієнтів, коли досягається найменша помилка.

Звертаємо увагу, що помилка не дорівнює нулю, але стабілізується на рівні біля 5. Це означає, що модель робить типову помилку 5. Якщо необхідно зменшити помилку, то треба додати більше інформації в модель, наприклад, більше змінних або використовувати різні оцінки.

Згадаймо перше рівняння

$$y = \beta + \alpha X + \epsilon$$

Кінцеві вагові коефіцієнти – 3,8 і 2,78.

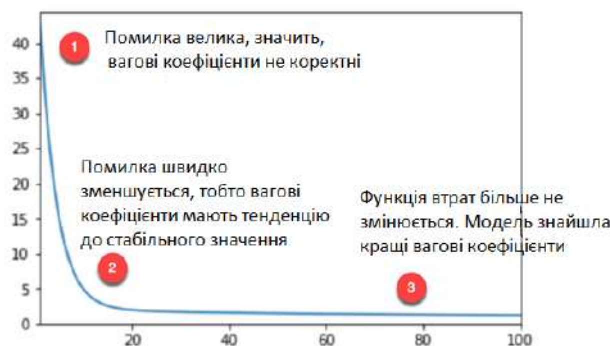



Рис. 7.4. Процес навчання моделі

## Тренування моделі лінійної регресії за допомогою TensorFlow

А тепер, коли ми краще розуміємо, що відбувається «під капотом», спробуємо використати API оцінювача, що наданий TF, для підготовки першої лінійної регресії.

Знову будемо використовувати набір даних Boston, який включає нижченаведені змінні:

| **crim** | **Відсоток злочинності на душу населення по містах** |
|---|---|
| zn | Питома вага землі для проживання, районованої для ділянок понад 25 000 кв. футів |
| indus | Питома вага нероздрібних акцій на місто |
| nox | Концентрація оксидів азоту |
| rm | Середня кількість кімнат у помешканні |
| age | Частина власників облаштованих помешкань, побудованих до 1940 року |
| dis | Зважені відстані до п'яти бостонських центрів зайнятості |
| tax | Повна вартість ставки податку на нерухомість за 10 000 доларів |
| ptratio | Співвідношення учні – вчитель в місті |
| medv | Середня вартість будинків, зайнятих власниками, у тисячах доларів |

Створимо три різних набори даних:

| **Набір даних** | **Призначення** | **Форма** |
|---|---|---|
| Training | Тренування моделі і визначення вагових коефіцієнтів | 400, 10 |
| Evaluation | Оцінювання працездатності моделі за невидимими даними | 100, 10 |
| Predict | Використання моделі для прогнозування вартості будинку за новими даними | 6, 10 |

Мета – використати параметри з набору даних для прогнозування вартості будинку.

Далі розглянемо, як використовувати TF трьома різними способами імпорту даних:

- з Pandasr;
- з Numpy;
- тільки з TF.

Зазначимо, що всі варіанти **дають однакові результати**.

Дізнаємося, як використовувати API високого рівня для побудови і навчимося оцінювати лінійну регресивну модель. Якщо будемо використовувати API низького рівня, то нам треба вручну визначити:

- функцію втрат;
- виконати оптимізацію (градієнтний спуск);
- помножити матриці;
- побудувати граф і тензор.

## Тренування з використанням Pandas

Нам треба імпортувати необхідні бібліотеки для тренування моделі:

```
import pandas as pd
from sklearn import datasets
import tensorflow as tf
import itertools
```

**Крок 1. Імпорт даних за допомогою pandas.**

Визначаємо назви стовпців і зберігаємо їх у COLUMNS. Можна використати `pd.read_csv()` для імпорту даних.

```
COLUMNS = ["crim", "zn", "indus", "nox", "rm", "age",
           "dis", "tax", "ptratio", "medv"]

training_set = pd.read_csv("E:/boston_train.csv",
skipinitialspace=True,skiprows=1, names=COLUMNS)

test_set = pd.read_csv("E:/boston_test.csv",
skipinitialspace=True,skiprows=1, names=COLUMNS)

prediction_set = pd.read_csv("E:/boston_predict.csv",
skipinitialspace=True,skiprows=1, names=COLUMNS)
```

Тепер можемо вивести дані:

```
print(training_set.shape, test_set.shape, prediction_set.shape)
```

**Результат на виході:**

```
(400, 10) (100, 10) (6, 10)
```

Зверніть увагу, що мітка, тобто **y**, включена в набір даних. Тож потрібно визначити ще два списки. Один, що містить лише параметри, а другий лише з назвою мітки. Ці два списки підкажуть оцінювачу, які параметри в наборі даних, а також яка назва стовпця є міткою.

Це робиться за допомогою коду:

```
FEATURES = ["crim", "zn", "indus", "nox", "rm",
                 "age", "dis", "tax", "ptratio"]
LABEL = "medv"
```

**Крок 2. Перетворення даних**

Нам треба перетворити числові змінні у відповідний формат. TF забезпечує метод перетворення безперервної змінної: `tf.feature_column.numeric_column()`.

На попередньому кроці ми визначили список параметрів, які хочемо включити в модель. Тепер можна використати цей список для перетворення їх у числові дані. Якщо

необхідно виключити ознаки зі своєї моделі, то треба видалити одну або кілька змінних у списку FEATURES, перш ніж створити element_cols.

Зауважимо, що ми будемо використовувати список включення Python зі списком FEATURES для створення нового списку з назвою function_cols. Це допомагає уникнути написання дев'яти разів tf.feature_column.numeric_column(). Включення списків – це швидший і чіткіший спосіб створення нових списків:

```
feature_cols = [tf.feature_column.numeric_column(k) for k in FEATURES]
```

**Крок 3. Визначення оцінювача**

На цьому кроці треба визначити оцінювач. TF наразі пропонує шість попередньо створених оцінювачів, у т. ч. три для завдання класифікації і три для завдання регресії.

- Регресія
    - DNNRegressor;
    - LinearRegressor;
    - DNNLineaCombinedRegressor.
- Класифікатор
    - DNNClassifier;
    - LinearClassifier;
    - DNNLineaCombinedClassifier.

У цьому розділі використаємо лінійний регресор. Для доступу до цієї функції треба скористатися tf.estimator.

Функції треба два аргументи:

- feature_columns: містить змінні, які включають в модель;
- model_dir: шлях до зберігання графа, збереження параметрів моделі тощо.

TF автоматично створить файл з назвою train у робочому каталозі. Треба скористатися вказаним шляхом для доступу до TensorBoard:

```
estimator = tf.estimator.LinearRegressor(
        feature_columns=feature_cols,
        model_dir="train")
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'train', '_tf_random_seed':
None, '_save_summary_steps': 100, '_save_checkpoints_steps': None,
'_save_checkpoints_secs': 600, '_session_config': None,
'_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000,
'_log_step_count_steps': 100, '_train_distribute': None, '_service':
None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1a215dc550>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Цікава частина з TF – це спосіб подати модель. TF призначений для роботи з паралельними обчисленнями і дуже великими наборами даних. Через обмеження ресурсів машини неможливо подати модель відразу з усіма даними. Для цього нам потрібно щоразу подавати пакет даних. Зверніть увагу, що йдеться про величезний набір даних з мільйонами і більше записів. Якщо не додати пакет, то ми отримаємо помилку пам'яті.

Наприклад, якщо наші дані містять 100 спостережень і визначаємо розмір партії 10, то це означає, що модель буде бачити 10 спостережень за кожну ітерацію (10*10).

Коли модель побачила всі дані, вона закінчує одну епоху (**epoch**). Епоха визначає, скільки разів ми хочемо, щоб модель бачила дані. Краще встановити цей крок не в одиницю і нехай модель виконує ітерацію кілька разів.

Друга інформація, яку потрібно додати, полягає в тому, чи хочемо ми перемішувати дані перед кожною ітерацією. Під час тренінгу важливо перетасовувати дані, щоб модель не вивчила конкретний зразок набору даних. Якщо модель дізнається деталі основної структури даних, у неї виникнуть труднощі узагальнити передбачення для небачених даних. Це називається перенавчання (**overfitting**). Модель добре працює на даних для навчання, але не може правильно передбачати для небачених (тобто нових) даних.

TF легко виконує ці два кроки. Коли дані надходять на конвеєр, вже задано: скільки спостережень (пакетів) потрібно; чи треба перетасовувати дані.

Для інструктажу TF, щоб подати модель, можна використати `pandas_input_fn`. Для цього об'єкта потрібно п'ять параметрів:

- x: дані про параметри;
- y: дані мітки;
- batch_size: розмір пакету (за замовчуванням 128);
- num_epoch: кількість епох (за замовчуванням 1);
- shuffle: перемішувати дані чи ні (за замовчуванням – None).

Нам треба подавати модель багато разів, тому визначимо функцію для повторення цього процесу. Все це у функції `get_input_fn`:

```
def get_input_fn(data_set, num_epochs=None, n_batch = 128,
shuffle=True):
        return tf.estimator.inputs.pandas_input_fn(
        x=pd.DataFrame({k: data_set[k].values for k in FEATURES}),
        y = pd.Series(data_set[LABEL].values),
        batch_size=n_batch,
        num_epochs=num_epochs,
        shuffle=shuffle)
```

Типовим методом оцінювання продуктивності моделі є послідовність:

- тренування моделі;
- оцінювання моделі з іншим набором даних;
- виконання прогнозу.

Оцінювач TF пропонує три різні функції для легкого виконання цих трьох кроків.

**Крок 4. Тренування моделі**

Можемо використати тренування оцінювача для оцінювання моделі. Тренування оцінювача вимагає `input_fn` і декілька кроків. Можна використати функцію, створену вище, для подачі моделі. Потім даємо змогу моделі повторити 1 000 разів. Звертаємо увагу, що не вказуємо кількість епох, а дозволяємо моделі повторюватись 1 000 разів. Якщо встановити кількість епох, що дорівнює 1, то модель повториться 4 рази: у навчальному наборі 400 записів, а розмір партії – 128.

1. 128 рядків.
2. 128 рядків.
3. 128 рядків.
4. 16 рядків.

Тому простіше встановити число епох в `None` і визначити кількість ітерацій:

```
estimator.train(input_fn=get_input_fn(training_set,
                                      num_epochs=None,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
```

**Результат на виході**:

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into train/model.ckpt.
INFO:TensorFlow:loss = 83729.64, step = 1
INFO:TensorFlow:global_step/sec: 238.616
INFO:TensorFlow:loss = 13909.657, step = 101 (0.420 sec)
INFO:TensorFlow:global_step/sec: 314.293
INFO:TensorFlow:loss = 12881.449, step = 201 (0.320 sec)
INFO:TensorFlow:global_step/sec: 303.863
INFO:TensorFlow:loss = 12391.541, step = 301 (0.327 sec)
INFO:TensorFlow:global_step/sec: 308.782
INFO:TensorFlow:loss = 12050.5625, step = 401 (0.326 sec)
INFO:TensorFlow:global_step/sec: 244.969
INFO:TensorFlow:loss = 11766.134, step = 501 (0.407 sec)
INFO:TensorFlow:global_step/sec: 155.966
INFO:TensorFlow:loss = 11509.922, step = 601 (0.641 sec)
INFO:TensorFlow:global_step/sec: 263.256
INFO:TensorFlow:loss = 11272.889, step = 701 (0.379 sec)
INFO:TensorFlow:global_step/sec: 254.112
INFO:TensorFlow:loss = 11051.9795, step = 801 (0.396 sec)
INFO:TensorFlow:global_step/sec: 292.405
INFO:TensorFlow:loss = 10845.855, step = 901 (0.341 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into train/model.ckpt.
INFO:TensorFlow:Loss for final step: 5925.9873.
```

Можемо перевірити TensorBoard з такою командою:

```
activate hello-tf
tensorboard --logdir=train
```

**Крок 5. Оцінювання моделі**

Можемо оцінити відповідність нашої моделі на тестовому наборі за допомогою нижченаведеного коду:

```
ev = estimator.evaluate(
          input_fn=get_input_fn(test_set,
          num_epochs=1,
          n_batch = 128,
          shuffle=False))
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-05-13-01:43:13
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Finished evaluation at 2018-05-13-01:43:13
INFO:TensorFlow:Saving dict for global step 1000: average_loss =
32.15896, global_step = 1000, loss = 3215.896
```

Величину втрат можемо вивести за допомогою коду:

```
loss_score = ev["loss"]
print("Loss: {0:f}".format(loss_score))
```

**Результат на виході:**

```
Loss: 3215.895996
```

Модель має величину втрат 3 215. Можемо перевірити підсумкову статистику, щоб зрозуміти, наскільки це велика помилка:

```
training_set['medv'].describe()
```

**Результат на виході:**

```
count    400.000000
mean      22.625500
std        9.572593
min        5.000000
25%       16.600000
50%       21.400000
75%       25.025000
max       50.000000
Name: medv, dtype: float64
```

З вищенаведеної підсумкової статистики знаємо, що середня ціна на будинок становить 22 тисячі, мінімальна ціна – 5 тисяч, а максимальна – 50 тисяч. Модель робить типову помилку в 3 тис. доларів.

**Крок 6. Виконання передбачення**

Отже, можемо виконати передбачення для оцінювання значення шести будинків у Бостоні:

```
y = estimator.predict(
         input_fn=get_input_fn(prediction_set,
         num_epochs=1,
         n_batch = 128,
         shuffle=False))
```

Для виводу передбачуваних значень використаємо код:

```
predictions = list(p["predictions"] for p in itertools.islice(y, 6))
print("Predictions: {}".format(str(predictions)))
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
Predictions: [array([32.297546], dtype=float32), array([18.96125],
dtype=float32), array([27.270979], dtype=float32), array([29.299236],
dtype=float32), array([16.436684], dtype=float32), array([21.460876],
dtype=float32)]
```

Модель прогнозує такі значення:

| Будинок | Прогноз |
|---|---|
| 1 | 32.29 |
| 2 | 18.96 |
| 3 | 27.27 |
| 4 | 29.29 |
| 5 | 16.43 |
| 6 | 21.46 |

Звертаємо вашу увагу, що ми не знаємо справжнього значення, а маємо лише прогноз.

## Рішення за допомогою оцінювача Numpy

Розглянемо, як тренувати модель, використовуючи оцінювач numpy для подачі даних. Метод той самий, але буде використовувати оцінювач `numpy_input_fn`:

```
training_set_n = pd.read_csv("E:/boston_train.csv").values
test_set_n = pd.read_csv("E:/boston_test.csv").values
prediction_set_n = pd.read_csv("E:/boston_predict.csv").values
```

**Крок 1. Імпорт даних**

Передусім потрібно відділити змінні ознак від мітки. Це потрібно зробити для даних задля навчання і оцінювання. Це прискорить визначення функції для розділення даних.

```
def prepare_data(df):
    X_train = df[:, :-3]
    y_train = df[:,-3]
    return X_train, y_train
```

Можна використати функцію для відділення мітки від ознак в наборі даних тренування / оцінювання.

```
X_train, y_train = prepare_data(training_set_n)
X_test, y_test = prepare_data(test_set_n)
```

Треба виключити останній стовпець набору даних прогнозування, оскільки він містить лише NaN:

```
x_predict = prediction_set_n[:, :-2]
```

Підтвердимо форму масиву. Зауважимо, що мітка не повинна мати розмірності, а це означає (400,).

```
print(X_train.shape, y_train.shape, x_predict.shape)
```

**Результат на виході:**

```
 (400, 9) (400,) (6, 9)
```

Можемо сконструювати стовпці параметрів таким чином:

```
feature_columns = [      tf.feature_column.numeric_column('x',
shape=X_train.shape[1:])]
```

Оцінювач визначається, як і раніше: вказуємо стовпці ознак і де зберегти граф:

```
estimator = tf.estimator.LinearRegressor(
         feature_columns=feature_columns,
         model_dir="train1")
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'train1',
'_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1a218d8f28>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Можна використати оцінювач numpy, щоб подавати дані в модель, а потім тренувати модель. Звертаємо увагу, що ми визначили функцію `input_fn` раніше, щоб полегшити читабельність.

```
# Тренування оцінювача
train_input = tf.estimator.inputs.numpy_input_fn(
           x={"x": X_train},
           y=y_train,
           batch_size=128,
           shuffle=False,
           num_epochs=None)
           estimator.train(input_fn = train_input,steps=5000)
```

**Результат на виході**:

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into train1/model.ckpt.
INFO:TensorFlow:loss = 83729.64, step = 1
INFO:TensorFlow:global_step/sec: 490.057
INFO:TensorFlow:loss = 13909.656, step = 101 (0.206 sec)
INFO:TensorFlow:global_step/sec: 788.986
INFO:TensorFlow:loss = 12881.45, step = 201 (0.126 sec)
INFO:TensorFlow:global_step/sec: 736.339
INFO:TensorFlow:loss = 12391.541, step = 301 (0.136 sec)
INFO:TensorFlow:global_step/sec: 383.305
INFO:TensorFlow:loss = 12050.561, step = 401 (0.260 sec)
```

```
INFO:TensorFlow:global_step/sec: 859.832
INFO:TensorFlow:loss = 11766.133, step = 501 (0.117 sec)
INFO:TensorFlow:global_step/sec: 804.394
INFO:TensorFlow:loss = 11509.918, step = 601 (0.125 sec)
INFO:TensorFlow:global_step/sec: 753.059
INFO:TensorFlow:loss = 11272.891, step = 701 (0.134 sec)
INFO:TensorFlow:global_step/sec: 402.165
INFO:TensorFlow:loss = 11051.979, step = 801 (0.248 sec)
INFO:TensorFlow:global_step/sec: 344.022
INFO:TensorFlow:loss = 10845.854, step = 901 (0.288 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into train1/model.ckpt.
INFO:TensorFlow:Loss for final step: 5925.985.
Out[23]:
<TensorFlow.python.estimator.canned.linear.LinearRegressor at
0x1a1b6ea860>
```

Повторюємо цей же крок за допомогою іншого оцінювача для оцінювання нашої моделі:

```
eval_input = tf.estimator.inputs.numpy_input_fn(
       x={"x": X_test},
       y=y_test,
       shuffle=False,
       batch_size=128,
       num_epochs=1)
estimator.evaluate(eval_input,steps=None)
```

**Результат на виході**:

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-05-13-01:44:00
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train1/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Finished evaluation at 2018-05-13-01:44:00
INFO:TensorFlow:Saving dict for global step 1000: average_loss =
32.158947, global_step = 1000, loss = 3215.8945
Out[24]:
{'average_loss': 32.158947, 'global_step': 1000, 'loss': 3215.8945}
```

Отже, можемо обчислити передбачення. Це схоже на pandas:

```
test_input = tf.estimator.inputs.numpy_input_fn(
        x={"x": x_predict},
        batch_size=128,
        num_epochs=1,
        shuffle=False)
y = estimator.predict(test_input)
predictions = list(p["predictions"] for p in itertools.islice(y, 6))
print("Predictions: {}".format(str(predictions)))
```

**Результат на виході**:

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
```

```
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train1/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
Predictions: [array([32.297546], dtype=float32), array([18.961248],
dtype=float32), array([27.270979], dtype=float32), array([29.299242],
dtype=float32), array([16.43668], dtype=float32), array([21.460878],
dtype=float32)]
```

## Рішення з TensorFlow

Розглянемо рішення з TF. Цей метод дещо складніший за попередній.

Зауважимо, що якщо використовувати блокнот Jupyter, то для запуску сеансу треба перезапустити і очистити ядро.

TF – чудовий інструмент для передачі даних у конвеєрі. Побудуємо функцію `input_fn`.

**Крок 1. Визначаємо шлях і формат даних**

Передусім оголошуємо дві змінні зі шляхом до файлу csv. Звертаємо увагу, що маємо два файли: один для навчального набору, а другий для тестового набору.

```
import tensorflow as tf
df_train = "E:/boston_train.csv"
df_eval = "E:/boston_test.csv"
```

Тепер треба визначити стовпці, які ми хочемо використати, з файлу csv. Використаємо всі стовпці. Після цього оголосимо тип змінної.

Змінна з плаваючою комою визначається з [0.]

```
COLUMNS = ["crim", "zn", "indus", "nox", "rm", "age",
               "dis", "tax", "ptratio", "medv"]RECORDS_ALL = [[0.0],
[0.0], [0.0], [0.0],[0.0],[0.0],[0.0],[0.0],[0.0],[0.0]]
```

**Крок 2. Визначаємо input_fn function**

Функцію можна розділити на три частини:

1. Імпорт даних.
2. Створення ітератора.
3. Поглинання даних.

Наведено загальний код для визначення функції. Код пояснюється нижче:

```
def input_fn(data_file, batch_size, num_epoch = None):
    # Крок 1
    def parse_csv(value):
        columns = tf.decode_csv(value, record_defaults= RECORDS_ALL)
        features = dict(zip(COLUMNS, columns))
        #labels = features.pop('median_house_value')
        labels =  features.pop('medv')
        return features, labels

        # Крок 2. Витягуємо рядки з вхідних файлів, використовуючи
Dataset API.
        dataset = (tf.data.TextLineDataset(data_file) # Читаємо
текстовий файл
        .skip(1) # Пропускаємо рядок з заголовками
        .map(parse_csv))
```

```
    dataset = dataset.repeat(num_epoch)
    dataset = dataset.batch(batch_size)
    # Крок 3
    iterator = dataset.make_one_shot_iterator()
    features, labels = iterator.get_next()
    return features, labels
```

Для csv-файлу метод набору даних читає по одному рядку. Щоб побудувати набір даних, треба використати об'єкт `TextLineDataset`. У нашому наборі даних є заголовок, тому можна скористатися `skip(1)`, щоб пропустити перший рядок. У цей момент лише читаємо дані і виключаємо заголовок з конвеєра. Щоб подати модель, треба відокремити параметри від мітки. Методом, який застосовується для будь-якого перетворення даних, є `map`.

Цей метод викликає функцію, яку створимо для того, щоб вчити, як трансформувати дані. Коротко: треба передати дані в об'єкт `TextLineDataset`, виключити заголовок і застосувати перетворення, яке визначається функцією.

Пояснення коду:

- tf.data.TextLineDataset(data_file): читаємо файл CSV;
- skip(1): пропуємо заголовок;
- map(parse_csv)): розбираємо записи в тензори.

Нам треба визначити функцію для вказівки об'єкта карти. Можемо викликати цю функцію з `parse_csv`.

Функція `map` аналізує файл csv методом `tf.decode_csv` й оголошує функції і мітку. Особливості можна оголосити словником або кортежем. Використовуємо метод словника, оскільки це зручніше.

Пояснення коду:

- tf.decode_csv(value, record_defaults= RECORDS_ALL): метод decode_csv використовує вихід TextLineDataset для зчитування файлу csv. A record_defaults вказує TF тип стовпців;
- dict(zip(_CSV_COLUMNS, columns)): заповнення словника усіма стовпцями, вилученими під час цього опрацювання даних;
- features.pop('median_house_value'): виключення цільової змінної зі змінних ознак і створення змінної мітки.

Набір даних потребує додаткових елементів, щоб ітераційно подавати тензори. Дійсно, нам треба додати метод повторення, щоб набір даних безперервно продовжувався для подачі моделі. Якщо не додаємо метод, то модель повториться лише один раз, а потім видасть помилку, оскільки в конвеєр більше не будуть надходити дані.

Після цього можемо контролювати розмір партії за допомогою методу `batch`. Це означає, що ми вказуємо набір даних, скільки даних хочемо передати в конвеєрі за кожну ітерацію. Якщо встановити великий розмір партії, то модель буде повільною.

**Крок 3. Створення ітератора**

Отже, тепер ми готові до наступного кроку: створення ітератора для повернення елементів у набір даних.

Найпростіший спосіб створення оператора – за допомогою методу `make_one_shot_iterator`.

Після цього можна створити з ітератора ознаки і мітки.

**Крок 4. Споживання даних**

Можемо перевірити, що відбувається з функцією `input_fn`. Нам треба викликати функцію під час сеансу, щоб споживати дані. Спробуємо з розміром партії, що дорівнює 1.

Зауважимо, що параметри виводяться у словнику і мітки як масив.

Буде показано перший рядок файлу csv. Можемо спробувати запустити цей код багато разів з різним розміром партії.

```
next_batch = input_fn(df_train, batch_size = 1, num_epoch = None)
with tf.Session() as sess:
     first_batch  = sess.run(next_batch)
     print(first_batch)
```

**Результат на виході**:

```
({'crim': array([2.3004], dtype=float32), 'zn': array([0.],
dtype=float32), 'indus': array([19.58], dtype=float32), 'nox':
array([0.605], dtype=float32), 'rm': array([6.319], dtype=float32),
'age': array([96.1], dtype=float32), 'dis': array([2.1], dtype=float32),
'tax': array([403.], dtype=float32), 'ptratio': array([14.7],
dtype=float32)}, array([23.8], dtype=float32))
```

**Крок 5. Визначення стовпця ознак**

Нам необхідно визначити числові стовпці так:

```
X1= tf.feature_column.numeric_column('crim')
X2= tf.feature_column.numeric_column('zn')
X3= tf.feature_column.numeric_column('indus')
X4= tf.feature_column.numeric_column('nox')
X5= tf.feature_column.numeric_column('rm')
X6= tf.feature_column.numeric_column('age')
X7= tf.feature_column.numeric_column('dis')
X8= tf.feature_column.numeric_column('tax')
X9= tf.feature_column.numeric_column('ptratio')
```

Зауважимо, що треба об’єднати всі змінні:

```
base_columns = [X1, X2, X3,X4, X5, X6,X7, X8, X9]
```

**Крок 6. Побудова моделі**

Можемо тренувати модель з оптимізатором LinearRegressor.

```
model = tf.estimator.LinearRegressor(feature_columns=base_columns,
model_dir='train3')
```

**Результат на виході**:

```
INFO:TensorFlow:Using default config. INFO:TensorFlow:Using config:
{'_model_dir': 'train3', '_tf_random_seed': None, '_save_summary_steps':
100, '_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1820a010f0>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Треба використати функцію лямбда, щоб записати аргумент у функцію `input_fn`. Якщо не використаємо лямбда-функцію, то не зможемо тренувати модель.

```
# Тренування оцінювача
model.train(steps =1000,
          input_fn= lambda : input_fn(df_train,batch_size=128, num_epoch
= None))
```

**Результат на виході**:

```
INFO:TensorFlow:ng model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into train3/model.ckpt.
INFO:TensorFlow:loss = 83729.64, step = 1
INFO:TensorFlow:global_step/sec: 72.5646
INFO:TensorFlow:loss = 13909.657, step = 101 (1.380 sec)
INFO:TensorFlow:global_step/sec: 101.355
INFO:TensorFlow:loss = 12881.449, step = 201 (0.986 sec)
INFO:TensorFlow:global_step/sec: 109.293
INFO:TensorFlow:loss = 12391.541, step = 301 (0.915 sec)
INFO:TensorFlow:global_step/sec: 102.235
INFO:TensorFlow:loss = 12050.5625, step = 401 (0.978 sec)
INFO:TensorFlow:global_step/sec: 104.656
INFO:TensorFlow:loss = 11766.134, step = 501 (0.956 sec)
INFO:TensorFlow:global_step/sec: 106.697
INFO:TensorFlow:loss = 11509.922, step = 601 (0.938 sec)
INFO:TensorFlow:global_step/sec: 118.454
INFO:TensorFlow:loss = 11272.889, step = 701 (0.844 sec)
INFO:TensorFlow:global_step/sec: 114.947
INFO:TensorFlow:loss = 11051.9795, step = 801 (0.870 sec)
INFO:TensorFlow:global_step/sec: 111.484
INFO:TensorFlow:loss = 10845.855, step = 901 (0.897 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into train3/model.ckpt.
INFO:TensorFlow:Loss for final step: 5925.9873.
Out[8]:
<TensorFlow.python.estimator.canned.linear.LinearRegressor at
0x18225eb8d0>
```

Можемо оцінити відповідність нашої моделі на тестовому наборі за допомогою нижченаведеного коду:

```
results = model.evaluate(steps =None,input_fn=lambda: input_fn(df_eval,
batch_size =128, num_epoch = 1))
for key in results:
    print("   {}, was: {}".format(key, results[key]))
```

**Результат на виході**:

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-05-13-02:06:02
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train3/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Finished evaluation at 2018-05-13-02:06:02
```

```
INFO:TensorFlow:Saving dict for global step 1000: average_loss = 
32.15896, global_step = 1000, loss = 3215.896
   average_loss, was: 32.158958435058594
   loss, was: 3215.89599609375
   global_step, was: 1000
```

Останнім кроком є прогнозування значення на основі значень матриці ознак. Можемо написати словник зі значеннями, які хочемо передбачити. Наша модель має 9 ознак і нам необхідно вказати значення для кожної. Модель дасть передбачення для кожної з них.

У коді записуємо значення кожної ознаки, які збережені в csv-файлі `df_predict`.

Нам треба записати нову функцію `input_fn function`, тому що немає мітки в наборі даних. Можемо використати API `Dataset.from_tensors`:

```
prediction_input = {
          'crim': [0.03359,5.09017,0.12650,0.05515,8.15174,0.24522],
          'zn': [75.0,0.0,25.0,33.0,0.0,0.0],
          'indus': [2.95,18.10,5.13,2.18,18.10,9.90],
          'nox': [0.428,0.713,0.453,0.472,0.700,0.544],
          'rm': [7.024,6.297,6.762,7.236,5.390,5.782],
          'age': [15.8,91.8,43.4,41.1,98.9,71.7],
          'dis': [5.4011,2.3682,7.9809,4.0220,1.7281,4.0317],
          'tax': [252,666,284,222,666,304],
          'ptratio': [18.3,20.2,19.7,18.4,20.2,18.4]
     }
    def test_input_fn():
        dataset = tf.data.Dataset.from_tensors(prediction_input)
        return dataset

    # Прогнозуємо весь наш prediction_input
    pred_results = model.predict(input_fn=test_input_fn)
```

Отже, виводимо передбачення:

```
for pred in enumerate(pred_results):
    print(pred)
```

**Результат на виході**:

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train3/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
(0, {'predictions': array([32.297546], dtype=float32)})
(1, {'predictions': array([18.96125], dtype=float32)})
(2, {'predictions': array([27.270979], dtype=float32)})
(3, {'predictions': array([29.299236], dtype=float32)})
(4, {'predictions': array([16.436684], dtype=float32)})
(5, {'predictions': array([21.460876], dtype=float32)})

INFO:TensorFlow:Calling model_fn. INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Graph was finalized. INFO:TensorFlow:Restoring
parameters from train3/model.ckpt-5000 INFO:TensorFlow:Running
local_init_op. INFO:TensorFlow:Done running local_init_op. (0,
{'predictions': array([35.60663], dtype=float32)}) (1, {'predictions':
```

```
array([22.298521], dtype=float32)}) (2, {'predictions':
array([25.74533], dtype=float32)}) (3, {'predictions':
array([35.126694], dtype=float32)}) (4, {'predictions':
array([17.94416], dtype=float32)}) (5, {'predictions':
array([22.606628], dtype=float32)})
```

**Висновки**

Для тренування моделі необхідно:

- визначити ознаки: незалежні змінні: x;
- визначити мітку: залежна змінна: y;
- створити набір тренування / тестування;
- визначити початкові вагові коефіцієнти;
- визначити функцію втрат: MSE;
- оптимізувати модель: градієнтний спуск;
- визначити:
  - швидкість навчання;
  - кількість епох;
  - розмір пакета.

Отже, ми вивчили, як використовувати багаторівневий API для оцінювача лінійної регресії. Необхідно визначити:

1. Стовпці ознак. Якщо вводити безперервно: `tf.feature_column.numeric_column()`. Можемо заповнити список за допомогою списку охоплення python.
2. Оцінювач: `tf.estimator.LinearRegressor(feature_columns, model_dir)`.
3. Функцію для імпорту даних, розміру пакета і епох: `input_fn()`.

Після цього вже готові тренувати, оцінювати і робити передбачення з `train()`, `evaluate()` і `predict()`

# 8. Лінійна регресія для машинного навчання

## Лінійна регресія TensorFlow із взаємодією

Розглянемо, як перевірити дані і підготувати їх для створення простого завдання лінійної регресії[17] [17].

Розділ поділено на дві частини:

- пошук взаємодії;
- тестування моделі.

Раніше ми використовували набір даних Бостона, щоб оцінити середню ціну будинку. Набір даних Бостона має невеликий розмір, лише 506 спостережень. Цей набір даних вважається еталоном для перевірки нових алгоритмів лінійної регресії.

Набір даних складається з таких 12 параметрів:

| **Змінна** | **Опис** |
|---|---|
| zn | частина житлових земель, зонованих для ділянок понад 25 000 кв. футів |
| indus | частина нероздрібного бізнесу в місті |

[17] https://www.guru99.com/linear-regression-for-machine-learning.html

| | |
|---|---|
| nox | концентрація оксидів азоту |
| rm | середня кількість кімнат на житло |
| age | вікова частина заселених власником будівель, побудованих до 1940 року |
| dis | зважені відстані до п’яти бостонських центрів зайнятості |
| tax | податкова повноцінна ставка податку на нерухомість за 10 000 доларів |
| ptratio | співвідношення учня – вчителя по місту |
| medv | середня вартість заселених власником будинків у тисячах доларів |
| crim | рівень злочинності на душу населення за містом |
| chas | підставна змінна річки Чарльз-Рівер (1, якщо берег річки; 0 в іншому випадку) |
| B | частина чорношкірого населення по місту |

Оцінимо медіанну ціну за допомогою лінійної регресії, але акцент зробимо на одному конкретному процесі машинного навчання – «підготовці даних».

Модель виділяє шаблон в даних. Щоб зафіксувати такий шаблон, треба його спочатку знайти. Доброю практикою є проведення аналізу даних перед запуском будь-якого алгоритму машинного навчання.

**Вибір правильних параметрів (ознак) має основне значення для успіху моделі.**

**Уявіть, що ми намагаємося оцінити заробітну плату людей, а якщо не включимо гендер як коваріату, то все закінчиться поганою оцінкою.**

Інший спосіб вдосконалити модель – це переглянути зв’язок між незалежними змінними. Повернувшись до попереднього прикладу, можемо вважати освіту відмінним кандидатом для прогнозування заробітної плати, а також професії. Справедливо сказати, що професія залежить від рівня освіти, а саме вища освіта часто приводить до кращої професії. Якщо узагальнити цю ідею, то можна сказати, що кореляція між залежною змінною і пояснювальною змінною може бути збільшена додаванням ще однієї пояснювальної змінної (рис. 8.1 – ілюстрація Chris Albon[18] [18]).

Щоб зафіксувати обмежений вплив освіти на професію, можемо використати термін взаємодії.

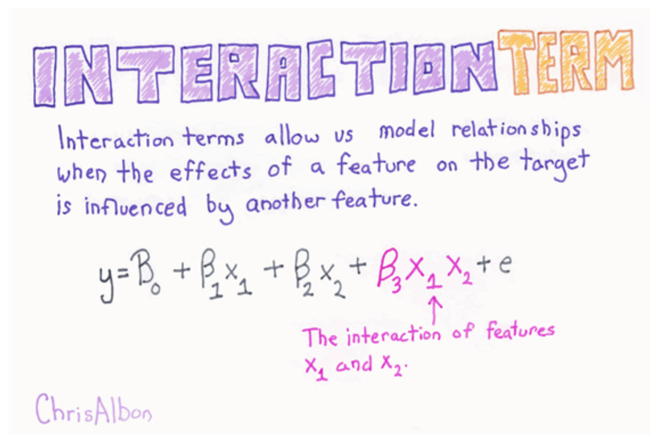



Рис. 8.1. Термін взаємодії

Якщо подивитися на рівняння заробітної плати, то воно виглядає так:

$$wage = \alpha + \beta_1 occupation + \beta_2 education + \beta_3 occupation * education + \epsilon\ .$$

---

[18] https://chrisalbon.com/machine_learning/linear_regression/adding_interaction_terms/

Якщо $\beta_3$ додатне, то це означає, що додатковий рівень освіти приносить більший приріст середньої вартості будинку за високого рівня професії, а отже, існує взаємозв'язок між освітою і професією.

Спробуємо розібратися, які змінні можуть бути добрим кандидатом на умови взаємодії. Також перевіримо, чи додавання подібної інформації приведе до кращого прогнозування цін.

## Зведена статистика

Перш ніж перейти до моделі, слід виконати кілька кроків. Як було зазначено, модель – це узагальнення даних. Найкраща практика – це зрозуміти дані і зробити прогнозування. Якщо ми не знаємо своїх даних, то у нас малі шанси покращити свою модель.

На першому кроці завантажимо дані як кадр даних pandas й створимо навчальний і тестовий набори.

***Поради.*** Для цього розділу треба мати встановлені на Python matplotlit і seaborn. Можемо встановити пакети Python за допомогою Jupyter. Ми **не повинні обов'язково це робити.**

```
!conda install -- yes matplotlib
```

але

```
import sys
!{sys.executable} -m pip install matplotlib # Вже інстальовано
!{sys.executable} -m pip install seaborn
```

Зауважимо, що цей крок не є необхідним, якщо вже встановлені `matplotlib` і `seaborn`.

Matplotlib є бібліотекою для створення графа в Python. Seaborn – бібліотека візуалізації статистики, що побудована зверху matplotlib. Це забезпечує привабливі і красиві діаграми.

Нижченаведений код імпортує необхідні бібліотеки:

```
import pandas as pd
from sklearn import datasets
import tensorflow as tf
from sklearn.datasets import load_boston
import numpy as np
```

Бібліотека `sklearn` включає набір даних Boston. Можемо викликати його API для імпорту даних:

```
boston = load_boston()
df = pd.DataFrame(boston.data)
```

Назви параметрів зберігаються в масиві об'єкта `feature_names`:

```
boston.feature_names
```

**Результат на виході:**

```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD','TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

Можемо перейменувати стовпці:

```
df.columns = boston.feature_names
df['PRICE'] = boston.target
df.head(2)
```

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | PRICE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.00632 | 18.0 | 2.31 | 0.0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1.0 | 296.0 | 15.3 | 396.9 | 4.98 | 24.0 |
| 1 | 0.02731 | 0.0 | 7.07 | 0.0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2.0 | 242.0 | 17.8 | 396.9 | 9.14 | 21.6 |

Перетворимо змінну CHAS в рядкову змінну і позначимо її як **yes,** якщо CHAS = 1, і **no,** якщо CHAS = 0

```
df['CHAS'] = df['CHAS'].map({1:'yes', 0:'no'})
df['CHAS'].head(5)
0    no
1    no
2    no
3    no
4    no
Name: CHAS, dtype: object
```

Для використання pandas просто розділимо набір даних. Випадковим чином розділяємо набір даних на 80-відсотковий набір для тренувань і 20-відсотковий тестовий набір. Pandas мають вбудовану функцію витрат для розділення вибірки кадра даних.

Перший параметр `frac` – це значення від 0 до 1. Встановимо його таким, що дорівнює 0,8, щоб випадково вибрати 80 відсотків кадра даних.

`random_state` дає змогу повернути однаковий кадр даних для всіх.

```
### Створення набору тренування/тестування
df_train=df.sample(frac=0.8,random_state=200)
df_test=df.drop(df_train.index)
```

Можемо отримати форму даних. Вона має бути така:

- `Train set: 506*0.8 = 405`
- `Test set: 506*0.2 = 101`

```
print(df_train.shape, df_test.shape)
```

**Результат на виході:**

```
(405, 14) (101, 14)
```

```
df_test.head(5)
```

**Результат на виході**:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | PRICE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | no | 0.538 | 6.575 | 65.2 | 4.0900 | 1.0 | 296.0 | 15.3 | 396.90 | 4.98 | 24.0 |
| **1** | 0.02731 | 0.0 | 7.07 | no | 0.469 | 6.421 | 78.9 | 4.9671 | 2.0 | 242.0 | 17.8 | 396.90 | 9.14 | 21.6 |
| **3** | 0.03237 | 0.0 | 2.18 | no | 0.458 | 6.998 | 45.8 | 6.0622 | 3.0 | 222.0 | 18.7 | 394.63 | 2.94 | 33.4 |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **6** | 0.08829 | 12.5 | 7.87 | no | 0.524 | 6.012 | 66.6 | 5.5605 | 5.0 | 311.0 | 15.2 | 395.60 | 12.43 | 22.9 |
| **7** | 0.14455 | 12.5 | 7.87 | no | 0.524 | 6.172 | 96.1 | 5.9505 | 5.0 | 311.0 | 15.2 | 396.90 | 19.15 | 27.1 |

Дані «брудні»: вони часто не зрівноважені і мають чужі значення, які скидають аналіз машинного навчання.

**Першим кроком до очищення набору даних є розуміння, що треба очищати.** Очищення набору даних може бути складним завданням, особливо для використання якогось узагальненого способу.

Команда Google Research розробила інструмент для цієї роботи з назвою Facets, який допомагає візуалізувати дані і нарізати їх різними способами. Це добра відправна точка для розуміння того, як складається база даних.

Facets дає змогу знайти дані не зовсім такі, як нам здається.

За винятком своїх вебдодатків, Google допомагає легко вставляти інструментарій у блокнот Jupyter.

Є дві частини Facets:

- Facets Overview;
- Facets Deep Dive.

## Facets Overview для огляду набору даних

- Facets Overview надає огляд набору даних. Facets Overview розбиває стовпці даних на рядки з яскравим відображенням інформації:
- відсоток відсутнього спостереження;
- значення min і max;
- статистика, типу середнє, середнє і стандартне відхилення;
- додається стовпчик, який показує відсоток значень, які дорівнюють нулю, що корисно, коли більшість значень дорівнюють нулю;
- можна побачити розподіли для кожного параметра на тестовому наборі даних, а також на тренувальному наборі, а це означає, що можна двічі перевірити, чи тест має схожий розподіл з навчальними даними.

Це мінімум, який потрібно зробити перед будь-яким завданням машинного навчання. За допомогою такого інструменту зробимо цей важливий крок, а він висвітить деякі відхилення.

## Facets Deep Dive для окремих фрагментів даних

Facets Deep Dive – інструмент, який дає змогу отримати деяку чіткість на нашому наборі даних і збільшити масштаб настільки, щоб побачити окремий фрагмент даних. Це означає, що можна переглядати дані за рядками і стовпцями будь-якого параметра з набору даних.

Будемо використовувати обидва ці інструменти з набором даних Бостона.

***Примітка***. Не можна одночасно використовувати Facets Overview і Facets Deep Dive. Спочатку потрібно очистити блокнот, щоб змінити інструмент.

## Встановлення вебдодатка Facets

Вебдодаток Facets можна використовувати для здійснення більшості аналізів. Розглянемо, як ним користуватися в блокноті Jupyter.

Передусім необхідно встановити nbextensions. Це робиться за допомогою наведеного коду. Копіюємо і вставляємо код у терміналі машини:

```
pip install jupyter_contrib_nbextensions
```

Відразу після цього треба клонувати сховище на наш комп’ютер. Є два варіанти:

**Варіант 1.** Скопіюємо і вставимо наданий код у терміналі (рекомендується).

Якщо на комп’ютері не встановлено Git, переходимо за URL-адресою[19] [19] і дотримуємося інструкцій для встановлення. Після завершення роботи можна скористатися командою `git` в Anaconda для Windows:

```
git clone https://github.com/PAIR-code/facets
```

**Варіант 2.** Переходимо на репозиторій[20] [20] і завантажуємо код (рис. 8.2).

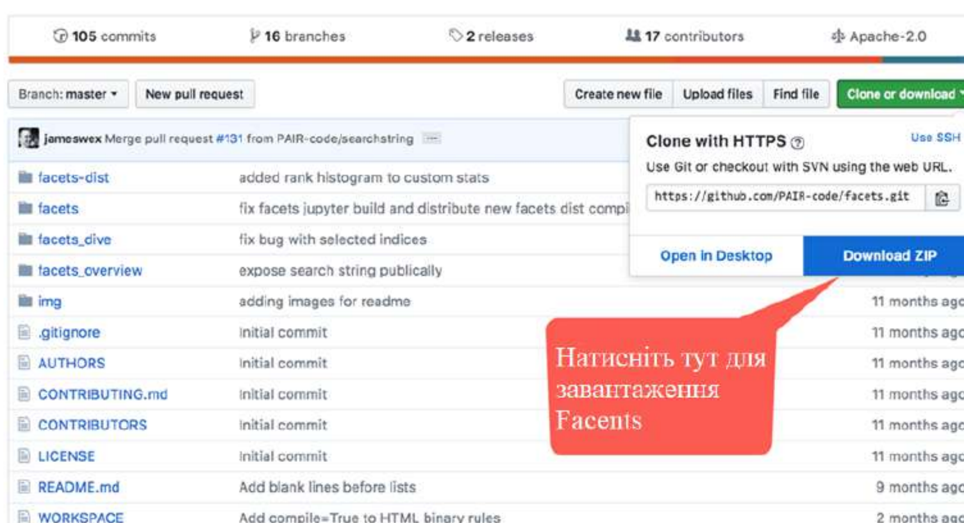



Рис. 8.2. Завантаження Facents з Git

Якщо виберемо перший варіант, файл завантажиться у папку завантаження. Можна завантажити файл туди, а можна вказати інший шлях для завантаження.

Можемо перевірити, де саме зберігається Facets за допомогою командного рядка:

```
echo `pwd`/`ls facets`
```

Після цього треба його встановити в блокнот Jupyter. Для цього слід встановити робочий каталог у папку, в якій зберігається Facets.

Наш поточний робочий каталог і розташування архіву Facets мають бути в одному місці (рис.8.3).

Вказуємо робочий каталог на Facet:

```
cd facets
```

[19] https://git-scm.com/download/win
[20] https://github.com/PAIR-code/facets

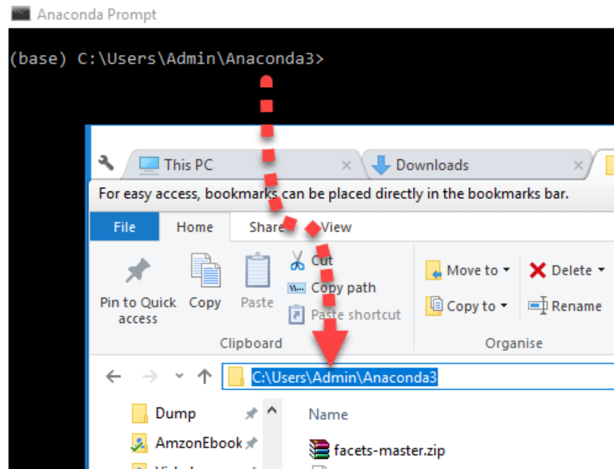



Рис. 8.3. Вибір папки для встановлення Facents

Маємо два варіанти, щоб інсталювати Facets в Jupyter. Якщо інсталюємо Jupyter з Conda для всіх користувачів, то копіюємо код:

```
jupyter nbextension install facets-dist/
```

в іншому випадку використовуємо:

```
jupyter nbextension install facets-dist/ --user
```

Тепер все налаштовано. Можемо відкрити Facet Overview.

## Overview для обчислення статистики

Overview використовує сценарій Python для обчислення статистики. Для цього треба імпортувати скрипт під назвою `generic_feature_statistics_generator` в Jupyter. Це не проблема, оскільки сценарій перебуває у файлах facets.

Знайдемо шлях до нього. Спочатку відкриємо facets, потім файл facets_overview і python (рис. 8.4). Копіюємо шлях.

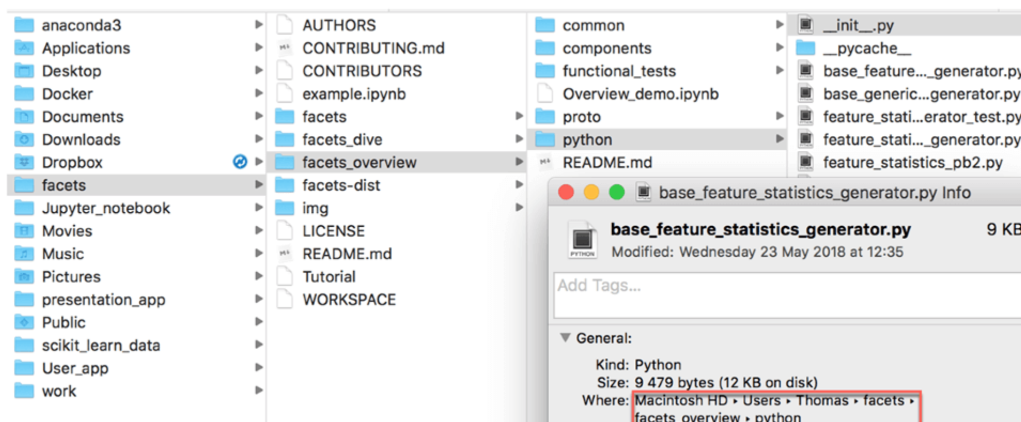



Рис. 8.4. Шлях до сценарію

Після цього повертаємося в Jupyter і записуємо наступний код. Змінюємо шлях '/Users/Thomas/facets/facets_overview/python' на наш шлях. У Windows:

```
import sys
sys.path.append(r"C:\Users\Admin\Anaconda3\facets-
master\facets_overview\python")
```

Можемо імпортувати сценарій за допомогою коду:

```
from generic_feature_statistics_generator import
GenericFeatureStatisticsGenerator
```

Щоб обчислити статистику параметрів, необхідно використати функцію `GenericFeatureStatisticsGenerator()` і об’єкт ProtoFromDataFrames. Можемо передати кадр даних у словник. Наприклад, якщо потрібно створити підсумкову статистику для тренувального набору, то можна зберегти інформацію у словнику і використати її в об’єкті `ProtoFromDataFrames`

```
'name': 'train', 'table': df_train
```

`Name` – це назва відображуваних таблиць. Ми використовуємо назву таблиці, для якої потрібно обчислити підсумок. У нашому прикладі таблиця, що містить дані, – це `df_train`.

```
# Обчислюємо прототип статистичних даних функцій з наборів даних для
використання їх в facets overview
import base64

gfsg = GenericFeatureStatisticsGenerator()

proto = gfsg.ProtoFromDataFrames([{'name': 'train', 'table': df_train},
                                  {'name': 'test', 'table': df_test}])

#proto = gfsg.ProtoFromDataFrames([{'name': 'train', 'table':
df_train}])
protostr = base64.b64encode(proto.SerializeToString()).decode("utf-8")
```

Отже, просто копіюємо і вставляємо код, наведений нижче. Код беремо безпосередньо з GitHub, як наведено на рис. 8.5.

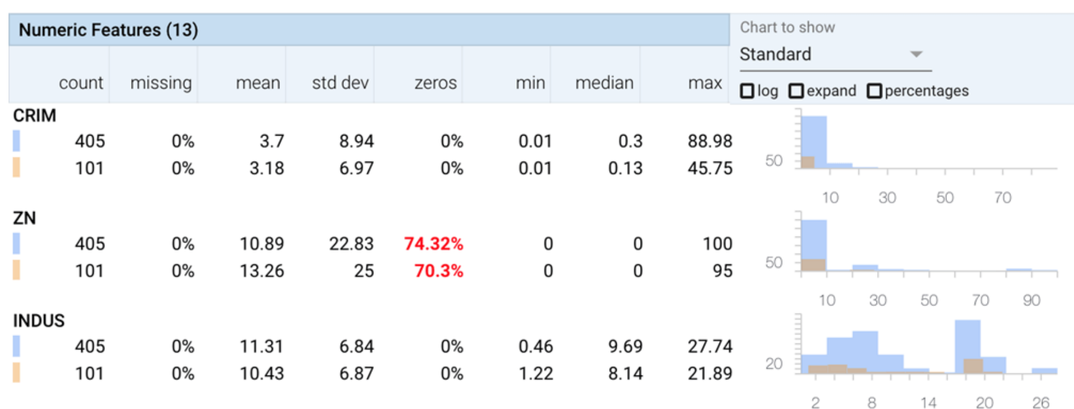



Рис. 8.5. Статистичні дані

```
# Відображення даних візуалізацією facets overview # Displ
```

```
from IPython.core.display import display, HTML

HTML_TEMPLATE = """<link rel="import" href="/nbextensions/facets-
dist/facets-jupyter.html" >
        <facets-overview id="elem"></facets-overview>
        <script>
          document.querySelector("#elem").protoInput = "{protostr}";
        </script>"""
html = HTML_TEMPLATE.format(protostr=protostr)
display(HTML(html))
```

## Побудова матриці кореляції

Після перевірки даних і їх розподілу можна побудувати матрицю кореляції. Матриця кореляції обчислює коефіцієнт Пірсона. Цей коефіцієнт має значення між -1 і 1, при цьому додатне значення вказує на позитивну кореляцію, а від'ємне – на негативну.

Цікаво побачити, які змінні можуть бути добрими кандидатами на умови взаємодії.

```
## Вибір важливого параметра і перевірка з Dive
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="ticks")
# Розрахунок матриці кореляції
corr = df.corr('pearson')
# Створення маски для верхнього трикутника
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
# Встановлення фігури matplotlib
f, ax = plt.subplots(figsize=(11, 9))

# Створення кольорової карти налаштовуваного відхиляння
cmap = sns.diverging_palette(220, 10, as_cmap=True)

# Малювання теплової карти за допомогою маски та виправлення
співвідношення
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,annot=True,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

**Результат на виході:**

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a184d6518>
```
png

Із матриці можна побачити (рис.8.6), що LSTAT і RM сильно корелюються з PRICE. Ще одна значуща особливість – сильна позитивна кореляція між NOX і INDUS, а це означає, що ці дві змінні рухаються в одному напрямку. Крім того, вони також співвідносяться з PRICE. Також сильно корелює DIS з IND і NOX.

У нас є перший натяк на те, що IND і NOX можуть бути добрими кандидатами на термін взаємодії, а DIS також може бути цікавим для розгляду.

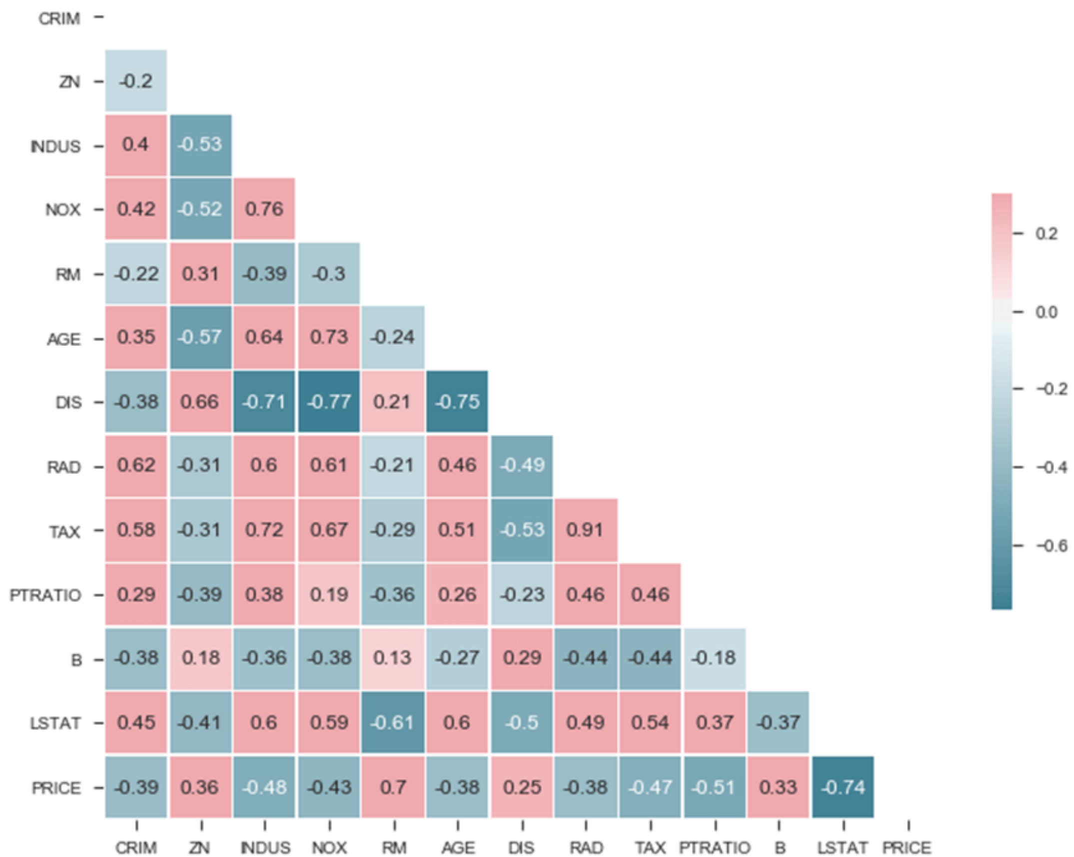



Рис. 8.6. Матриця кореляції

Розглянемо це детальніше, побудувавши сітку пар, яка розкриє карту кореляції, що склали раніше.

Сітку пар складаємо так:

- верхня частина: ділянка розкидання з приталеною лінією;
- діагональ: діаграма щільності ядра;
- нижня частина: діаграма щільності багатовимірного ядра.

Обираємо фокус на чотирьох незалежних змінних. Вибір відповідає змінним, які мають сильну кореляцією з PRICE:

- INDUS;
- NOX;
- RM;
- LSTAT,

крім того, беремо PRICE.

***Примітка.*** Стандартна помилка додається до діаграми розкиду за замовчуванням.

```
attributes = ["PRICE", "INDUS", "NOX", "RM", "LSTAT"]

g = sns.PairGrid(df[attributes])
g = g.map_upper(sns.regplot, color="g")
g = g.map_lower(sns.kdeplot,cmap="Reds", shade=True, shade_lowest=False)
g = g.map_diag(sns.kdeplot)
```

**Результат на виході:**

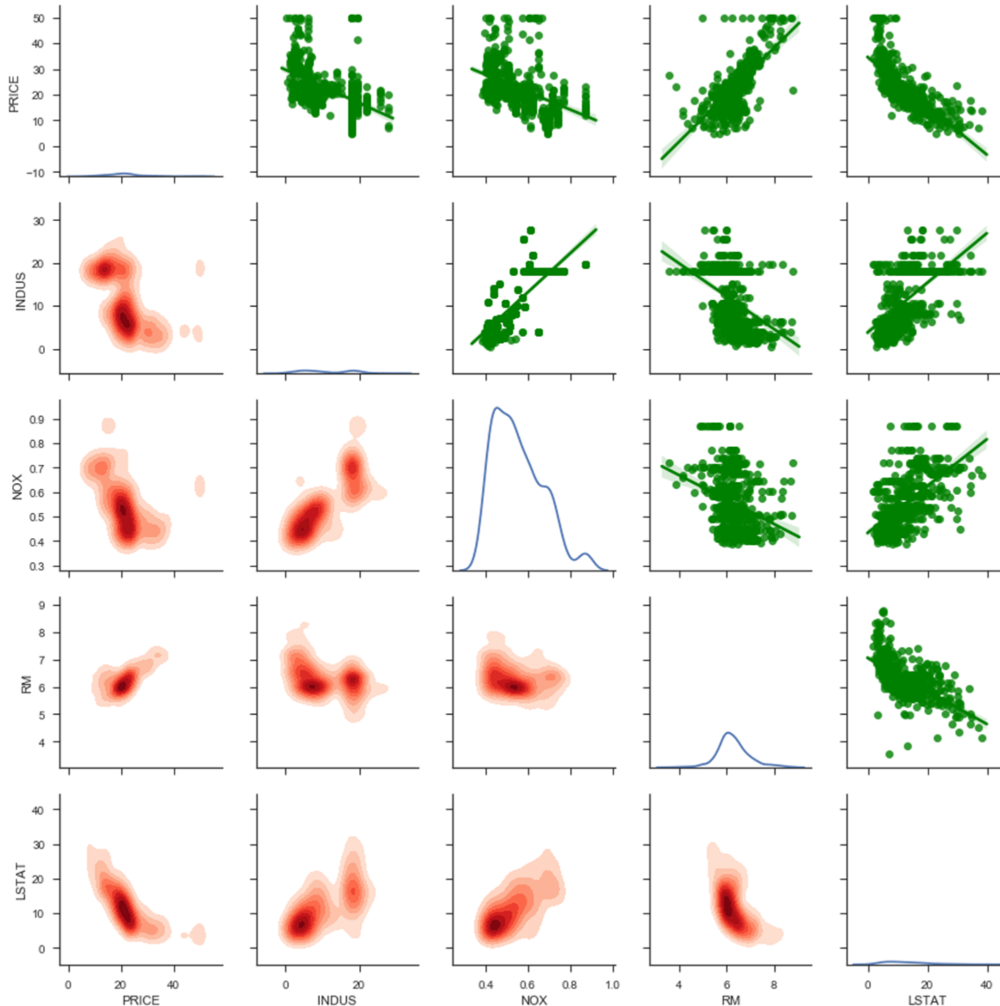



Рис. 8.7. Сітка пар

Почнемо з верхньої частини (рис.8.7):

- PRICE негативно корелює з INDUS, NOX і LSTAT; позитивно корелює з RM;
- існує деяка нелінійність з LSTAT і PRICE;
- існує майже пряма лінія, коли PRICE дорівнює 50.

З опису набору даних PRICE була обрізана при значенні 50.

Діагональ:

- здається, що у NOX два кластери: один близько 0,5, а другий близько 0,85.

Щоб дізнатися більше про це, слід подивитися на нижню частину. Багатоваріантна щільність ядра цікава тим, що вона забарвлена там, де більшість точок. Різниця з графіком розкиду приводить до щільності ймовірності, навіть якщо для такої координати немає сенсу в наборі даних. Більш насичений колір вказує на високу концентрацію в околі точки.

Якщо перевірити багатоваріантну щільність для INDUS і NOX, то можна побачити позитивну кореляцію і два кластери. Коли частина промисловості перевищує 18, концентрація оксидів азоту понад 0,6.

Можна подумати над тим, щоб додати взаємодію між INDUS і NOX у лінійному співвідношенні, а також можна використати інший інструмент, створений Google, Facets Deep Dive. Інтерфейс розділений на чотири основні секції. Центральна зона в центрі – це масштабоване відображення даних. У верхній частині панелі розташоване випадаюче меню, де можна змінити розташування даних, щоб керувати faceting, розташуванням і кольором.

Праворуч – детальний вигляд конкретного рядка даних. Це означає, що можна натиснути будь-яку точку даних у центрі візуалізації, щоб побачити деталі про цю конкретну точку даних.

Під час кроку візуалізації даних цікаво знайти попарну кореляцію між незалежною змінною на ціну будинку. Однак вона містить щонайменше три змінні, а тривимірні графіки складні для роботи.

Один зі способів розв’язати цю проблему – створити змінну категорій. Отже, можемо створити двовимірну діаграму кольором точки, а також розділити змінну PRICE на чотири категорії, причому кожна категорія – це чверть (тобто 0,25, 0,5, 0,75). Назвемо цю нову змінну `Q_PRICE` (рис. 8.8).

```
## Перевірка нелінійності з важливими параметрами
df['Q_PRICE'] =  pd.qcut(df['PRICE'], 4, labels=["Lowest", "Low",
"Upper", "upper_plus"])
## Відображення нелінійності між RM і LSTAT
ax = sns.lmplot(x="DIS", y="INDUS", hue="Q_PRICE", data=df, fit_reg =
False,palette="Set3")
```

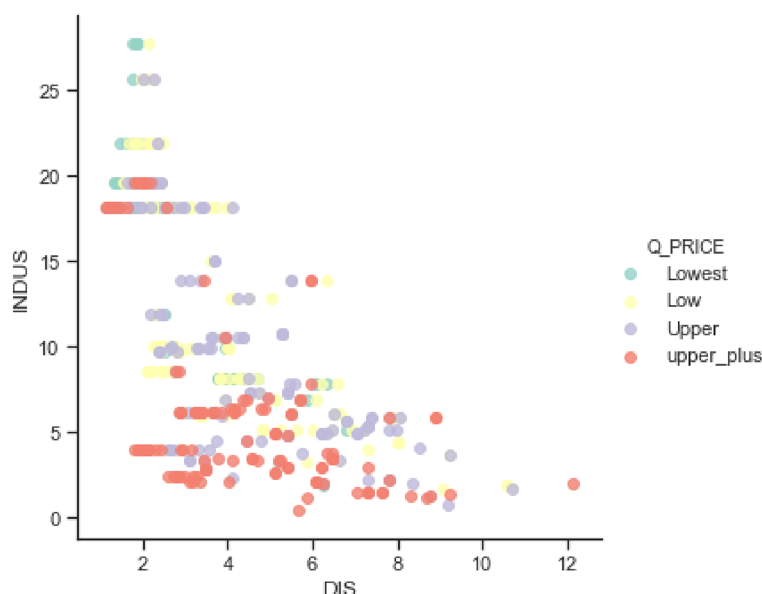



Рис. 8.8. Змінна Q_PRICE

## Використання Facets Deep Dive

Необхідно перетворити дані в json-формат, щоб відкрити Deep Dive, а також використати Pandas як об’єкт для цього. Можемо використати `to_json` після набору даних Pandas.

Перший рядок коду опрацьовує розмір набору даних.

```
df['Q_PRICE'] =  pd.qcut(df['PRICE'], 4, labels=["Lowest", "Low",
"Upper", "upper_plus"])
sprite_size = 32 if len(df.index)>50000 else 64
jsonstr = df.to_json(orient='records')
```

Код нижче взято з Google GitHub. Результат виконання коду показано на рис. 8.9.

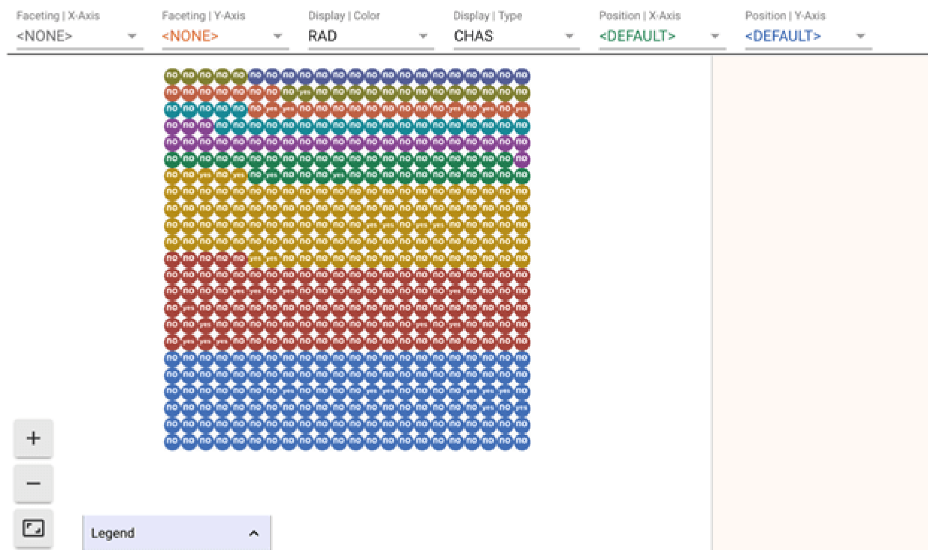



Рис. 8.9. Виведення аналізу з Facets Deep Dive

```
# Відображення Dive-візуалізації для цих даних
from IPython.core.display import display, HTML

# Створення шаблону Facets
HTML_TEMPLATE = """<link rel="import" href="/nbextensions/facets-dist/facets-jupyter.html">
        <facets-dive sprite-image-width="{sprite_size}" sprite-image-height="{sprite_size}" id="elem" height="600"></facets-dive>
        <script>
          document.querySelector("#elem").data = {jsonstr};
        </script>"""

# Завантаження набору даних і sprite_size в шаблон
html = HTML_TEMPLATE.format(jsonstr=jsonstr, sprite_size=sprite_size)

# Відображення шаблону
display(HTML(html))
```

Цікаво дізнатись, чи існує зв’язок між розвитком промисловості, концентрацією оксиду, відстанню до центру роботи й ціною будинку.

Для цього спочатку розділимо дані за розвитком промисловості і кольором за ціновим квартилем:

- виберемо faceting X і обираємо INDUS;
- виберемо Display і обираємо DIS.

Це забарвить точки в квартилі від ціни будинку (більш темні кольори означають, що відстань до першого центру роботи велика).

Отже, це знову показує те, що ми знаємо: нижчий рівень промисловості – вища ціна. Тепер можна подивитися на розподіл від INDUX, від NOX:

- виберемо faceting Y і обираємо NOX.

Можна побачити, що будинок, далекий від першого центру роботи, має найнижчу частину промисловості, а отже, найнижчу концентрацію оксиду. Якщо ми вирішимо відобразити тип за допомогою Q_PRICE і збільшити масштаб у нижньому лівому куті, то побачимо цю ціну.

Є ще одна підказка, що взаємодія між IND, NOX і DIS може стати добрим кандидатом для вдосконалення моделі.

## API оцінювачів TensorFlow

Оцінимо лінійний класифікатор за допомогою API оцінювачів TF. Послідовність наших дій така:

- підготовка даних;
- оцінювання benchmark-моделі (без взаємодії);
- оцінювання моделі зі взаємодією.

Пам'ятаємо, що мета машинного навчання – мінімізувати помилки. У цьому випадку виграє модель з найнижчою середньо квадратичною помилкою. Оцінювач TF автоматично обчислює цей показник.

### Підготовка даних

У більшості випадків нам необхідно перетворити свої дані, а Facets Overview є корисним для цього. З узагальненої статистики ми побачили, що є чужинці. Їх значення впливають на оцінки, оскільки вони не схожі на решту населення, яке ми аналізуємо. Зазвичай чужинці змінюють результати. Наприклад, позитивні чужі тенденції здебільшого завищують коефіцієнт.

Добрим рішенням для розв'язання цієї проблеми є стандартизація змінної. Стандартизація означає стандартне відхилення одиниці і нуля. Процес стандартизації включає два етапи. По-перше, він віднімає середнє значення змінної. По-друге, він ділиться на дисперсію, а отже, розподіл має одиничну дисперсію.

Бібліотека `sklearn` корисна для стандартизації змінних. Для цього можна використати модуль попереднього опрацювання з масштабуванням об'єкта.

Можна використати нижченаведену функцію для масштабування набору даних. Зауважимо, що ми не масштабуємо стовпчик міток і категорійні змінні.

```
from sklearn import preprocessing
def standardize_data(df):
    X_scaled = preprocessing.scale(df[['CRIM', 'ZN', 'INDUS', 'NOX',
'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT']])
    X_scaled_df = pd.DataFrame(X_scaled, columns = ['CRIM', 'ZN',
'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'])
    df_scale = pd.concat([X_scaled_df,
                       df['CHAS'],
                       df['PRICE']],axis=1, join='inner')
    return df_scale
```

Можна використати функцію для створення масштабованого набору тренування / тестування:

```
df_train_scale = standardize_data(df_train)
df_test_scale = standardize_data(df_test)
```

### Основна регресія:Benchmark

Передусім тренуємо і тестуємо модель без взаємодії. Мета – побачити показники продуктивності моделі.

Спосіб навчання моделі такий самий, як підручник API високого рівня. Використаємо оцінювач TF LinearRegressor.

Для нагадування нам треба:

- вибрати параметри, які слід застосувати в моделі;
- перетворити параметри;
- побудувати лінійний регресор;
- побудувати функцію input_fn;
- навчити модель;
- протестувати модель.

Використовуємо всі змінні в наборі даних для навчання моделі. Всього є кілька одиночних безперервних змінних і одна змінна для категорії:

```
## Додавання параметрів:
### Визначення безперервного списку
CONTI_FEATURES  = ['CRIM', 'ZN', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD','TAX', 'PTRATIO', 'B', 'LSTAT']
CATE_FEATURES = ['CHAS']
```

Перетворюємо параметри в числовий або стовпчик категорії

```
continuous_features = [tf.feature_column.numeric_column(k) for k in CONTI_FEATURES]
#categorical_features = tf.feature_column.categorical_column_with_hash_bucket(CATE_FEATURES, hash_bucket_size=1000)
categorical_features = [tf.feature_column.categorical_column_with_vocabulary_list('CHAS', ['yes','no'])]
```

Створюємо модель за допомогою `linearRegressor`. Зберігаємо модель у папці `train_Boston`

```
model = tf.estimator.LinearRegressor(
        model_dir="train_Boston",
    feature_columns=categorical_features + continuous_features)
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'train_Boston', '_tf_random_seed': None, '_save_summary_steps': 100, '_save_checkpoints_steps': None, '_save_checkpoints_secs': 600, '_session_config': None, '_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100, '_train_distribute': None, '_service': None, '_cluster_spec': <TensorFlow.python.training.server_lib.ClusterSpec object at 0x1a19e76ac8>, '_task_type': 'worker', '_task_id': 0, '_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '', '_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Кожен рядок даних для тренування або тестування перетворюється в тензор за допомогою функції `get_input_fn`

```
FEATURES = ['CRIM', 'ZN', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD','TAX', 'PTRATIO', 'B', 'LSTAT', 'CHAS']
LABEL= 'PRICE'
def get_input_fn(data_set, num_epochs=None, n_batch = 128, shuffle=True):
    return tf.estimator.inputs.pandas_input_fn(
```

```
        x=pd.DataFrame({k: data_set[k].values for k in FEATURES}),
        y = pd.Series(data_set[LABEL].values),
        batch_size=n_batch,
        num_epochs=num_epochs,
        shuffle=shuffle)
```

Оцінюємо модель на даних тренування:

```
model.train(input_fn=get_input_fn(df_train_scale,
                                      num_epochs=None,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into train_Boston/model.ckpt.
INFO:TensorFlow:loss = 56417.703, step = 1
INFO:TensorFlow:global_step/sec: 144.457
INFO:TensorFlow:loss = 76982.734, step = 101 (0.697 sec)
INFO:TensorFlow:global_step/sec: 258.392
INFO:TensorFlow:loss = 21246.334, step = 201 (0.383 sec)
INFO:TensorFlow:global_step/sec: 227.998
INFO:TensorFlow:loss = 30534.78, step = 301 (0.439 sec)
INFO:TensorFlow:global_step/sec: 210.739
INFO:TensorFlow:loss = 36794.5, step = 401 (0.477 sec)
INFO:TensorFlow:global_step/sec: 234.237
INFO:TensorFlow:loss = 8562.981, step = 501 (0.425 sec)
INFO:TensorFlow:global_step/sec: 238.1
INFO:TensorFlow:loss = 34465.08, step = 601 (0.420 sec)
INFO:TensorFlow:global_step/sec: 237.934
INFO:TensorFlow:loss = 12241.709, step = 701 (0.420 sec)
INFO:TensorFlow:global_step/sec: 220.687
INFO:TensorFlow:loss = 11019.228, step = 801 (0.453 sec)
INFO:TensorFlow:global_step/sec: 232.702
INFO:TensorFlow:loss = 24049.678, step = 901 (0.432 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into
train_Boston/model.ckpt.
INFO:TensorFlow:Loss for final step: 23228.568.


<TensorFlow.python.estimator.canned.linear.LinearRegressor at
0x1a19e76320>
```

Отже, оцінюємо характеристики моделі з набором для тестування:

```
model.evaluate(input_fn=get_input_fn(df_test_scale,
                                      num_epochs=1,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-05-29-02:40:43
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train_Boston/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Finished evaluation at 2018-05-29-02:40:43
INFO:TensorFlow:Saving dict for global step 1000: average_loss =
86.89361, global_step = 1000, loss = 1650.9785


{'average_loss': 86.89361, 'global_step': 1000, 'loss': 1650.9785}
```

Втрати моделі дорівнюють 1 650. Це показник, який слід обіграти в наступному розділі.

## Удосконалення моделі: врахування взаємодії

З вищенаведеного матеріалу простежується цікавий взаємозв'язок між змінними. Різні методи візуалізації свідчать, що INDUS і NOS пов'язані між собою й збільшують вплив на ціну. Не тільки взаємодія між INDUS і NOS впливає на ціну, але цей ефект сильніший, коли він взаємодіє з DIS.

Отже, настав час узагальнити цю ідею, а також спробувати вдосконалити модель для передбачення.

Треба додати два нових стовпчики до кожного набору даних: тренування + тест. Для цього створимо одну функцію для обчислення терміну взаємодії, а іншу для обчислення потрійного терміну взаємодії. Кожна функція створює один стовпець. Після створення нових змінних можемо об'єднати їх у навчальний і тестовий набори даних.

По-перше, створюємо нову змінну для взаємодії між INDUS і NOX.

Функція нижче повертає два фрейми даних: тренування і тест з взаємодією між var_1 й var_2 (у нашому випадку INDUS і NOX).

```
def interaction_term(var_1, var_2, name):
    t_train = df_train_scale[var_1]*df_train_scale[var_2]
    train = t_train.rename(name)
    t_test = df_test_scale[var_1]*df_test_scale[var_2]
    test = t_test.rename(name)
    return train, test
```

Зберігаємо два нові стовпці:

```
interation_ind_ns_train, interation_ind_ns_test=
interaction_term('INDUS', 'NOX', 'INDUS_NOS')
interation_ind_ns_train.shape
(325,)
```

По-друге, створюємо другу функцію для обчислення потрійного терміну взаємодії:

```
def triple_interaction_term(var_1, var_2,var_3, name):
    t_train =
df_train_scale[var_1]*df_train_scale[var_2]*df_train_scale[var_3]
    train = t_train.rename(name)
```

```
    t_test =
df_test_scale[var_1]*df_test_scale[var_2]*df_test_scale[var_3]
    test = t_test.rename(name)
    return train, test
interation_ind_ns_dis_train, interation_ind_ns_dis_test=
triple_interaction_term('INDUS', 'NOX', 'DIS','INDUS_NOS_DIS')
```

Тепер, коли у нас є всі необхідні стовпці, можемо додати їх до наборів даних для тренування і тестування. Називаємо ці два нові фрейми даних:

- df_train_new;
- df_test_new

```
df_train_new = pd.concat([df_train_scale,
                         interation_ind_ns_train,
                         interation_ind_ns_dis_train],
                         axis=1, join='inner')
df_test_new = pd.concat([df_test_scale,
                         interation_ind_ns_test,
                         interation_ind_ns_dis_test],
                         axis=1, join='inner')
df_train_new.head(5)
```

**Результат на виході:**

| | CRIM | ZN | INDUS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | CHAS | PRICE | INDUS_NOS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2** | -0.275582 | -0.47701 | -0.464046 | -0.162933 | -0.188265 | 0.812916 | 0.105941 | -0.661477 | -0.616881 | 1.147718 | 0.444455 | 0.803221 | no | 34.7 | 0.075608 |
| **4** | 1.017983 | -0.47701 | 0.992729 | 1.594192 | -0.595967 | 0.987593 | -0.907724 | 1.636106 | 1.502932 | 0.776192 | -1.278797 | 1.504488 | no | 36.2 | 1.582601 |
| **5** | -0.407050 | -0.47701 | -1.155868 | -0.589166 | 1.036257 | 0.620414 | -0.164461 | -0.891235 | -0.835358 | -0.338387 | 0.444455 | -1.025327 | no | 28.7 | 0.680999 |
| **8** | -0.367794 | -0.47701 | -0.747795 | -0.432591 | -0.157121 | 0.798656 | -0.346465 | -0.201960 | -0.616881 | -0.524150 | 0.426162 | 1.182209 | no | 16.5 | 0.323489 |
| **9** | 1.832214 | -0.47701 | 0.992729 | 1.246247 | -2.699598 | 0.795092 | -1.129861 | 1.636106 | 1.502932 | 0.776192 | -0.779497 | 2.450575 | no | 18.9 | 1.237185 |

У результаті можемо оцінити нову модель з умовами взаємодії і побачити, як працює показник ефективності.

```
CONTI_FEATURES_NEW  = ['CRIM', 'ZN', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD','TAX', 'PTRATIO', 'B', 'LSTAT',
                       'INDUS_NOS', 'INDUS_NOS_DIS']
### Визначення списку категорій
continuous_features_new = [tf.feature_column.numeric_column(k) for k in
CONTI_FEATURES_NEW]
model = tf.estimator.LinearRegressor(
    model_dir="train_Boston_1",
    feature_columns= categorical_features + continuous_features_new)
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'train_Boston_1',
'_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1a1a5d5860>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Код

```
FEATURES = ['CRIM', 'ZN', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD','TAX', 'PTRATIO', 'B', 'LSTAT','INDUS_NOS',
'INDUS_NOS_DIS','CHAS']
LABEL= 'PRICE'
def get_input_fn(data_set, num_epochs=None, n_batch = 128,
shuffle=True):
    return tf.estimator.inputs.pandas_input_fn(
       x=pd.DataFrame({k: data_set[k].values for k in FEATURES}),
       y = pd.Series(data_set[LABEL].values),
       batch_size=n_batch,
       num_epochs=num_epochs,
       shuffle=shuffle)
model.train(input_fn=get_input_fn(df_train_new,
                                      num_epochs=None,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into train_Boston_1/model.ckpt.
INFO:TensorFlow:loss = 56417.703, step = 1
INFO:TensorFlow:global_step/sec: 124.844
INFO:TensorFlow:loss = 65522.3, step = 101 (0.803 sec)
INFO:TensorFlow:global_step/sec: 182.704
INFO:TensorFlow:loss = 15384.148, step = 201 (0.549 sec)
INFO:TensorFlow:global_step/sec: 208.189
INFO:TensorFlow:loss = 22020.305, step = 301 (0.482 sec)
INFO:TensorFlow:global_step/sec: 213.855
INFO:TensorFlow:loss = 28208.812, step = 401 (0.468 sec)
INFO:TensorFlow:global_step/sec: 209.758
INFO:TensorFlow:loss = 7606.877, step = 501 (0.473 sec)
INFO:TensorFlow:global_step/sec: 196.618
INFO:TensorFlow:loss = 26679.76, step = 601 (0.514 sec)
INFO:TensorFlow:global_step/sec: 196.472
INFO:TensorFlow:loss = 11377.163, step = 701 (0.504 sec)
INFO:TensorFlow:global_step/sec: 172.82
INFO:TensorFlow:loss = 8592.07, step = 801 (0.578 sec)
INFO:TensorFlow:global_step/sec: 168.916
INFO:TensorFlow:loss = 19878.56, step = 901 (0.592 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into
train_Boston_1/model.ckpt.
INFO:TensorFlow:Loss for final step: 19598.387.

<TensorFlow.python.estimator.canned.linear.LinearRegressor at
0x1a1a5d5e10>
model.evaluate(input_fn=get_input_fn(df_test_new,
                                      num_epochs=1,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-05-29-02:41:14
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from train_Boston_1/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Finished evaluation at 2018-05-29-02:41:14
INFO:TensorFlow:Saving dict for global step 1000: average_loss =
79.78876, global_step = 1000, loss = 1515.9863

{'average_loss': 79.78876, 'global_step': 1000, 'loss': 1515.9863}
```

Нові втрати дорівнюють 1515. Просто додавши дві нові змінні, ми змогли зменшити втрати. А це означає, що можна зробити краще прогнозування, ніж з benchmark-моделлю.

# 9. Лінійний класифікатор в TensorFlow: бінарна класифікація

Дві найпоширеніші задачі, що розв'язуються під час навчання з учителем: лінійна регресія і лінійний класифікатор. Лінійна регресія прогнозує значення, а лінійний класифікатор прогнозує клас. В цьому розділі зосередимось на лінійному класифікаторі[21] [21].

Проблеми з класифікацією становлять приблизно 80 відсотків завдань машинного навчання. Класифікація має на меті передбачити ймовірність кожного класу для заданого набору даних. Мітка (тобто залежна змінна) – це дискретне значення, яке називається класом.

1. Якщо мітка має лише два класи, алгоритм навчання є двійковим класифікатором.
2. Багатокласовий класифікатор знаходить мітки для більш, ніж двох класів.

Наприклад, типова проблема бінарної класифікації – передбачити ймовірність того, що клієнт здійснить другу покупку. Прогнозувати тип тварини, що відображається на малюнку, є проблемою класифікації в багатокласовій формі, оскільки існує більше двох різновидів тварин.

В теоретичній частині цього розділу основну увагу звернемо на бінарний клас.

## Як працює бінарний класифікатор

У попередньому розділі ми дізналися, що функція складається з двох видів змінних: залежної змінної і набору ознак (незалежних змінних). У лінійній регресії залежна змінна – це дійсне число без діапазону. Основна мета – передбачити її значення шляхом мінімізації середньої квадратичної помилки.

Для бінарного завдання мітка може мати лише два цілих значення. У більшості випадків це або [0,1], або [1,2]. Наприклад, мета полягає в тому, щоб передбачити, купить клієнт товар чи ні. Мітка визначається так:

- Y = 1 (клієнт придбав товар);
- Y = 0 (клієнт товар не купив).

[21] https://www.guru99.com/linear-classifier-tensorflow.html

Модель використовує ознаки **X** для класифікації кожного клієнта у найбільш ймовірному класі, до якого він належить (потенційний покупець чи ні).

Ймовірність успіху обчислюється **логістичною регресією**. Алгоритм обчислює ймовірність на основі ознаки **X** і прогнозує успіх, коли ця ймовірність понад 50 відсотків. Більш формально ймовірність обчислюється так:

$$P(Y = 1|x) = \frac{1}{1 + exp(-(\theta^T x + b))},$$

де $\theta^T$ – набір вагових коефіцієнтів; $x$ – ознаки; $b$ – зміщення.

Функцію можна розкласти на дві частини:

- лінійна модель;
- логістична функція.

**Лінійна модель**

Ми вже опанували спосіб обчислення вагових коефіцієнтів. Вони обчислюються за допомогою суми добутків: $\theta^T x + b \text{ that is } \sum_{i=0}^{n} x_i w_i + b.$ **Y** – лінійна функція всіх ознак $\mathbf{x_i}$. Якщо модель не має ознак, то передбачення дорівнює зміщенню **b**.

Вагові коефіцієнти вказують напрямок кореляції між ознаками $\mathbf{x_i}$ і міткою **y**. Позитивна кореляція збільшує ймовірність позитивного класу, а негативна кореляція наближує ймовірність до 0 (тобто негативного класу).

Лінійна модель повертає лише дійсне число, що не відповідає мірі ймовірності діапазону [0,1]. Логістична функція необхідна для перетворення результатів лінійної моделі в ймовірність.

**Логістична функція**

У штучній нейронній мережі (ШНМ) функція активації нейрона формує вихідний сигнал, який визначається вхідним сигналом або набором вхідних сигналів. Стандартна комп’ютерна мікросхема може розглядатися як цифрова мережа функцій активації, які можуть мати значення «ON» (1) або «OFF» (0) залежно від входу (функція Хевісайта). Це схоже на поведінку лінійного перцептрона в нейронних мережах. Однак тільки нелінійні функції активації дають змогу таким мережам вирішувати нетривіальні завдання з використанням малого числа вузлів. У ШНМ ця функція також називається функцією передачі.

На рис. 9.1 наведено деякі функції активації, які широко використовуються[22] [22] в нейронних мережах. Найчастіше використовується логістична функція або функція S-подібного виду (сигмоїд).

Одна з причин, згідно якої сигмоїд широко використовується в нейронних мережах, це просте обчислення похідної функції через саму функцію, що допомагає істотно скоротити обчислювальну складність методу зворотного поширення помилки, зробивши його придатним для практики.

Обчислення похідної необхідне, тому що для коригування вагових коефіцієнтів під час навчання ШНМ за допомогою алгоритму зворотного поширення використовується метод градієнтного спуску.

Вихід сигмоїдної функції завжди перебуває між 0 і 1, тому вихід лінійної регресії легко замінити на сигмоподібну функцію. Це призводить до отримання нового числа – ймовірності від 0 до 1.

Класифікатор може перетворити ймовірність у клас:

- значення від 0 до 0,49 стають класом 0;
- значення від 0,5 до 1 стають класом 1.

[22] https://en.wikipedia.org/wiki/Activation_function

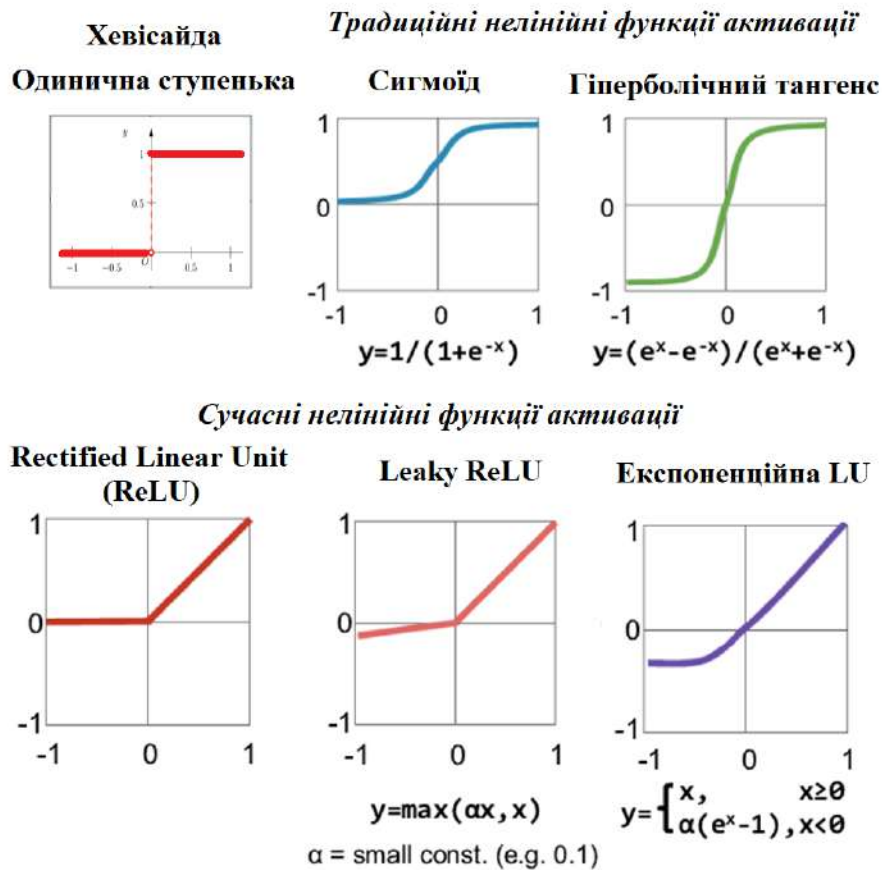



Рис. 9.1. Функції активації

## Вимірювання продуктивності лінійного класифікатора

Загальна ефективність класифікатора вимірюється метрикою точності. Для визначення точності збираємо всі правильні значення і ділимо на загальну кількість спостережень. Наприклад, значення точності 80 відсотків означає, що модель правильна у 80 відсотках випадків.

Можемо відзначити недолік цієї метрики, особливо для класу дисбалансу. Набір даних для дисбалансу виникає, якщо кількість спостережень на групу різна. Наприклад, ми намагаємося класифікувати рідкісну подію за допомогою логістичної функції. Уявімо, що класифікатор намагається оцінити смерть пацієнта після захворювання. За даними 5 відсотків пацієнтів помирають. Можемо навчити класифікатор прогнозувати кількість померлих і використовувати метрику точності для оцінювання виконання. Якщо класифікатор прогнозує 0 смертей для всього набору даних, то це буде правильним у 95 відсотках випадків.

### Матриця плутанини

Кращий спосіб оцінити ефективність класифікатора – переглянути матрицю плутанини (рис. 9.2).

| | | Передбачене | |
|---|---|---|---|
| | | TRUE | FALSE |
| Реальне | TRUE | TP | FN |
| | FALSE | FP | TN |

Рис. 9.2. Матриця плутанини

Матриця плутанини візуалізує точність класифікатора, порівнюючи фактичні і передбачувані класи. Двійкова матриця плутанини складається з квадратів:

- TP: істинне позитивне – прогнозовані значення правильно прогнозуються як фактичні позитивні;
- FP: хибне позитивне – прогнозовані значення неправильно передбачили фактичний позитив, а отже, негативні значення прогнозуються як позитивні;
- FN: хибне негативне – позитивні значення прогнозуються як негативні;
- TN: істинне негативне значення – прогнозовані правильні значення прогнозуються як фактичні негативні.

З матриці плутанини легко порівняти фактичний клас і передбачуваний клас.

Матриця плутанини дає змогу добре зрозуміти істинне позитивне й хибне позитивне. У деяких випадках бажано мати більш стислу метрику.

**Точність**

Метрика точності показує точність позитивного класу. Вона вимірює ймовірність правильного прогнозу позитивного класу:

$$Precision = \frac{TP}{TP + FP},$$

Максимальний бал – 1, якщо класифікатор ідеально класифікує всі позитивні значення. Точність сама по собі не дуже корисна, оскільки вона ігнорує негативний клас. Метрику, як правило, поєднують з метрикою нагадування. Нагадування також називають чутливістю або справжньою позитивною швидкістю.

**Чутливість**

Чутливість обчислює відношення правильно визначених позитивних класів. Ця метрика засвідчує наскільки добре модель розпізнає позитивний клас:

$$Recall = \frac{TP}{TP + FN}.$$

## Лінійний класифікатор з TensorFlow

Використаємо знову набір даних перепису. Метою є використання змінних з набору для прогнозування рівня доходів. Зауважимо, що в нашому випадку величина доходу є двійковою змінною:

- зі значенням 1, якщо дохід >50 тис.;
- зі значенням 0, якщо дохід <50 тис.

Ця змінна – наша мітка.

Набір даних включає вісім категорійних змінних:

- місце роботи;
- освіта;

- сімейний стан;
- професія;
- відносини;
- раса;
- стать;
- країна народження,
- а крім того, шість безперервних змінних:
- age;
- fnlwgt;
- education_num;
- capital_gain;
- capital_loss;
- hours_week.

На цьому прикладі розглянемо тренування лінійного класифікатора за допомогою оцінювача TF, а також спосіб поліпшення показника точності.

Будемо діяти у такій послідовності:

- Крок 1: імпортуємо дані;
- Крок 2: перетворюємо дані;
- Крок 3: тренуємо класифікатор;
- Крок 4: удосконалюємо модель;
- Крок 5: додаємо гіперпараметр: Lasso&Ridge.

**Крок 1. Імпортуємо дані**

Спочатку імпортуємо бібліотеки, які використовуються в розділі:

```
import tensorflow as tf
import pandas as pd
```

Далі імпортуємо дані з архіву UCI і довизначаємо назви стовпців. Будемо використовувати COLUMNS для назви стовпців у фреймах даних pandas.

Зазначимо, що тренувати класифікатор будемо за допомогою кадра даних Pandas.

```
## Визначення даних про шлях
COLUMNS = ['age','workclass', 'fnlwgt', 'education', 'education_num',
'marital',
           'occupation', 'relationship', 'race', 'sex', 'capital_gain',
           'capital_loss','hours_week', 'native_country', 'label']
PATH = "https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data"
PATH_test = "https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.test"
```

Дані зберігаються онлайн і вже розділені між тренувальним і тестовим наборами.

```
df_train = pd.read_csv(PATH, skipinitialspace=True, names = COLUMNS,
index_col=False)
df_test = pd.read_csv(PATH_test,skiprows = 1, skipinitialspace=True,
names = COLUMNS, index_col=False)
```

Набір для тренування має 32,561 спостереження, а тестовий набір – 16,281.

```
print(df_train.shape, df_test.shape)
```

```
print(df_train.dtypes)
(32561, 15) (16281, 15)
age                int64
workclass         object
fnlwgt             int64
education         object
education_num      int64
marital           object
occupation        object
relationship      object
race              object
sex               object
capital_gain       int64
capital_loss       int64
hours_week         int64
native_country    object
label             object
dtype: object
```

Для тренування класифікатора TF вимагає логічного значення. Нам треба передати значення від рядка до цілого числа. Мітка зберігається як об'єкт, однак треба перетворити її в числове значення. Нижченаведений код створює словник зі значеннями для перетворення і переходу циклу на елемент стовпця. Зауважимо, що виконуємо цю операцію двічі: раз для тренувального набору, а другий – для тестового.

```
label = {'<=50K': 0,'>50K': 1}
df_train.label = [label[item] for item in df_train.label]
label_t = {'<=50K.': 0,'>50K.': 1}
df_test.label = [label_t[item] for item in df_test.label]
```

У даних для тренування є 24 720 спостережень, в яких дохід менший за 50 тис., а також 7 841 спостережень, в яких він вищий. Майже таке ж співвідношення і для тестового набору.

```
print(df_train["label"].value_counts())
### Модель буде коректною мінімум в 70% випадків
print(df_test["label"].value_counts())
## Незбалансована мітка
print(df_train.dtypes)
0    24720
1     7841
Name: label, dtype: int64
0    12435
1     3846
Name: label, dtype: int64
age                int64
workclass         object
fnlwgt             int64
education         object
education_num      int64
marital           object
occupation        object
relationship      object
race              object
sex               object
capital_gain       int64
capital_loss       int64
hours_week         int64
```

```
native_country    object
label              int64
dtype: object
```

**Крок 2. Перетворення даних**

Необхідно пройти кілька кроків, перш ніж тренувати лінійний класифікатор за допомогою TF. Треба підготувати ознаки, які будуть включені до моделі. У бенчмарку регресії будемо використовувати вихідні дані, не застосовуючи жодних перетворень.

Оцінювач повинен мати перелік ознак для тренування моделі. Отже, дані стовпців треба перетворити в тензор.

Доброю практикою є визначення двох списків ознак, залежно від їх типу, а потім передача їх у `feature_columns` оцінювача.

Почнемо з перетворення безперервних ознак, а потім визначимо пакет з категорійними даними.

Ознаки з набору даних мають два параметри:

- ціле число;
- об'єкт.

Кожна ознака перерахована у наступних двох змінних відповідно до їх типів:

```
## Додавання ознак:
### Визначення безперервного списку
CONTI_FEATURES  = ['age', 'fnlwgt','capital_gain', 'education_num', 
'capital_loss', 'hours_week']
### Визначення списку категорій
CATE_FEATURES = ['workclass', 'education', 'marital', 'occupation', 
'relationship', 'race', 'sex', 'native_country']
```

`Feature_column` має об'єкт `numeric_column`, який допомагає у перетворенні безперервних змінних у тензор. У нижченаведеному коді перетворюємо всі змінні з `CONTI_FEATURES` в тензор із числовим значенням. Це обов'язково для побудови моделі. Усі незалежні змінні необхідно перетворити у відповідний тип тензора.

Для того щоб побачити, що відбувається з функцією наводимо код `function_column.numeric_column`. Будемо виводити перетворене значення для віку. Код python з коментарями, а отже, не потрібно детально розбиратися в ньому.

```
def print_transformation(feature = "age", continuous = True, size = 2):
    #X = fc.numeric_column(feature)
    ## Створення імені ознаки
    feature_names = [
    feature]

    ## Створення dict з даними
    d = dict(zip(feature_names, [df_train[feature]]))

    ## Перетворення віку
    if continuous == True:
        c = tf.feature_column.numeric_column(feature)
        feature_columns = [c]
    else:
        c = 
tf.feature_column.categorical_column_with_hash_bucket(feature, 
hash_bucket_size=size)
        c_indicator = tf.feature_column.indicator_column(c)
        feature_columns = [c_indicator]
```

```
## Використання вхідного шару для виведення значення
    input_layer = tf.feature_column.input_layer(
        features=d,
        feature_columns=feature_columns
        )
    ## Створення таблиці пошуку
    zero = tf.constant(0, dtype=tf.float32)
    where = tf.not_equal(input_layer, zero)
    ## Повернення таблиці пошуку
    indices = tf.where(where)
    values = tf.gather_nd(input_layer, indices)
    ## Ініціалізація графа
    sess = tf.Session()
    ## Виведення значення
    print(sess.run(input_layer))
print_transformation(feature = "age", continuous = True)
```

```
[[39.]
 [50.]
 [38.]
 ...
 [58.]
 [22.]
 [52.]]
```

Значення ті самі, що і в `df_train`

```
continuous_features = [tf.feature_column.numeric_column(k) for k in
CONTI_FEATURES]
```

Згідно з документацією TF є інший шлях для перетворення категорійних даних. Якщо список словника для ознак відомий і має невелику кількість значень, то можна створити категорійний стовпчик з `categorical_column_with_vocabulary_list`. Він призначить ідентифікатор для всіх унікальних списків словника.

Наприклад, якщо статус змінної має три різні значення:

- чоловік;
- дружина;
- одинокий,

то буде присвоєно три ідентифікатори особи. Наприклад, у чоловіка буде ідентифікатор 1, у дружини – ідентифікатор 2 тощо.

Для ілюстрації можемо використовувати цей код для перетворення змінної об’єкта у стовпець категорій у TF.

Опис статі може мати лише два значення: чоловік або жінка. Коли перетворимо опис статі, тоді TF створить 2 нові стовпці (один для чоловічої, а другий для жіночої статі). Якщо стать дорівнює чоловічій, то значення нового стовпця чоловіка буде дорівнювати 1, а жіночого – 0 і навпаки. Цей приклад відображено в таблиці:

| Рядки | Стать | Після перетворення | Чоловік | Жінка |
|---|---|---|---|---|
| 1 | чоловік | => | 1 | 0 |
| 2 | чоловік | => | 1 | 0 |
| 3 | жінка | => | 0 | 1 |

У TF:

```
print_transformation(feature = "sex", continuous = False, size = 2)
[[1. 0.]
 [1. 0.]
 [1. 0.]
 ...
 [0. 1.]
 [1. 0.]
 [0. 1.]]
```

```
relationship =
tf.feature_column.categorical_column_with_vocabulary_list(
    'relationship', [
        'Husband', 'Not-in-family', 'Wife', 'Own-child', 'Unmarried',
        'Other-relative'])
```

Нижче додано код Python, який нам не обов'язково розуміти, а наша мета – побачити перетворення.

Однак швидшим способом перетворення даних є використання методу `categorical_column_with_hash_bucket`. Корисна зміна рядкових змінних у рідкій матриці. Рідка матриця – це матриця з більшістю нулів. Метод реалізує все сам. Треба лише вказати кількість пакетів і стовпець ключів. Кількість пакетів – це максимальна кількість груп, яку може створити TF. Ключовий стовпець – це просто назва стовпця для перетворення.

У наведеному коді створюємо цикл над усіма категорійними ознаками:

```
categorical_features =
[tf.feature_column.categorical_column_with_hash_bucket(k,
hash_bucket_size=1000) for k in CATE_FEATURES]
```

**Крок 3. Тренування класифікатора**

TF нині забезпечує оцінювання лінійної регресії і лінійної класифікації:

- лінійна регресія: LinearRegressor;
- лінійна класифікація: LinearClassifier.

Синтаксис лінійного класифікатора такий же, як у розділі з лінійної регресії, за винятком одного аргументу, `n_class`. Отже, нам необхідно визначити стовпчик ознаки, каталог моделей і порівняти з лінійним регресором, а також визначити номер класу. Для логічної регресії кількість класів дорівнює 2.

Модель буде обчислювати вагові коефіцієнти стовпців, що містяться в безперервних даних і категорійних ознаках:

```
model = tf.estimator.LinearClassifier(
    n_classes = 2,
    model_dir="ongoing/train",
    feature_columns=categorical_features+ continuous_features)
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'ongoing/train',
'_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x181f24c898>, '_task_type': 'worker', '_task_id': 0,
```

```
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Отже, коли класифікатор визначено, можна створити вхідну функцію. Метод такий самий, як і в розділі про лінійний регресор. Використовуємо пакет розміром 128 і переміщуємо дані.

```
FEATURES = ['age','workclass', 'fnlwgt', 'education', 'education_num',
'marital', 'occupation', 'relationship', 'race', 'sex', 'capital_gain',
'capital_loss', 'hours_week', 'native_country']
LABEL= 'label'
def get_input_fn(data_set, num_epochs=None, n_batch = 128,
shuffle=True):
    return tf.estimator.inputs.pandas_input_fn(
       x=pd.DataFrame({k: data_set[k].values for k in FEATURES}),
       y = pd.Series(data_set[LABEL].values),
       batch_size=n_batch,
       num_epochs=num_epochs,
       shuffle=shuffle)
```

Створюємо функцію з аргументами, необхідними лінійному оцінювачу, тобто кількістю епох, кількістю партій і переміщуємо набір даних або мітки. Оскільки використовуємо метод Pandas для передачі даних у модель, то треба визначити змінні **X** як кадр даних pandas. Зауважимо, що ми перебуваємо в циклі для всіх даних, які зберігаються в `FEATURES`.

Навчимо модель за допомогою об’єкта `model.train`. Використаємо попередньо визначену функцію для подачі моделі з відповідними значеннями. Зауважимо, що ми встановили розмір партії 128, а кількість епох – `None`. Модель буде навчена з виконанням понад тисячі кроків.

```
model.train(input_fn=get_input_fn(df_train,
                                      num_epochs=None,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)

INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow: Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into ongoing/train/model.ckpt.
INFO:TensorFlow:loss = 88.722855, step = 1
INFO:TensorFlow:global_step/sec: 65.8282
INFO:TensorFlow:loss = 52583.64, step = 101 (1.528 sec)
INFO:TensorFlow:global_step/sec: 118.386
INFO:TensorFlow:loss = 25203.816, step = 201 (0.837 sec)
INFO:TensorFlow:global_step/sec: 110.542
INFO:TensorFlow:loss = 54924.312, step = 301 (0.905 sec)
INFO:TensorFlow:global_step/sec: 199.03
INFO:TensorFlow:loss = 68509.31, step = 401 (0.502 sec)
INFO:TensorFlow:global_step/sec: 167.488
INFO:TensorFlow:loss = 9151.754, step = 501 (0.599 sec)
INFO:TensorFlow:global_step/sec: 220.155
INFO:TensorFlow:loss = 34576.06, step = 601 (0.453 sec)
INFO:TensorFlow:global_step/sec: 199.016
```

```
INFO:TensorFlow:loss = 36047.117, step = 701 (0.503 sec)
INFO:TensorFlow:global_step/sec: 197.531
INFO:TensorFlow:loss = 22608.148, step = 801 (0.505 sec)
INFO:TensorFlow:global_step/sec: 208.479
INFO:TensorFlow:loss = 22201.918, step = 901 (0.479 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into
ongoing/train/model.ckpt.
INFO:TensorFlow:Loss for final step: 5444.363.

<TensorFlow.python.estimator.canned.linear.LinearClassifier at
0x181f223630>
```

Звертаємо увагу, що втрати зменшилися на останніх 100 кроках, тобто з 901 до 1 000.

Остаточні втрати після тисячі ітерацій – 5 444. Можемо оцінити свою модель на тестовому наборі і побачити продуктивність. Для оцінювання продуктивності нашої моделі треба використовувати об'єкт оцінки. Подаємо в модель тестовий набір і встановлюємо кількість епох 1, тобто дані перейдуть в модель лише один раз.

```
model.evaluate(input_fn=get_input_fn(df_test,
                                      num_epochs=1,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)

INFO:TensorFlow:Calling model_fn.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-06-02-08:28:22
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from ongoing/train/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Evaluation [100/1000]
INFO:TensorFlow:Finished evaluation at 2018-06-02-08:28:23
INFO:TensorFlow:Saving dict for global step 1000: accuracy = 0.7615626,
accuracy_baseline = 0.76377374, auc = 0.63300294, auc_precision_recall =
0.50891197, average_loss = 47.12155, global_step = 1000, label/mean =
0.23622628, loss = 5993.6406, precision = 0.49401596, prediction/mean =
0.18454961, recall = 0.38637546

{'accuracy': 0.7615626,
 'accuracy_baseline': 0.76377374,
 'auc': 0.63300294,
 'auc_precision_recall': 0.50891197,
 'average_loss': 47.12155,
 'global_step': 1000,
 'label/mean': 0.23622628,
 'loss': 5993.6406,
 'precision': 0.49401596,
 'prediction/mean': 0.18454961,
 'recall': 0.38637546}
```

TF повертає всі показники, про які ми дізналися в теоретичній частині. Очікувано, що точність велика завдяки незбалансованій мітці. А насправді модель працює трохи краще, ніж

випадкова здогадка. Уявіть, що модель передбачає, що всі домогосподарства з доходом нижче 50 тис., тоді модель має точність 70 відсотків. При більш детальному аналізі можемо побачити прогноз і досить низькі відкликання.

**Крок 4. Удосконалення моделі**

Тепер, коли ми маємо еталонну модель, можна спробувати її вдосконалити, тобто підвищити точність. У попередньому розділі ми дізналися, як покращити точність передбачення за допомогою терміну взаємодії. У цьому розділі переглянемо цю ідею, додавши до регресії поліноміальний член.

Поліноміальна регресія має важливе значення при нелінійності даних. Відомо два способи зафіксувати нелінійність даних:

- додати многочлен;
- додати безперервну змінну в категорійну змінну.

**Поліноміальні многочлени**

З рис. 9.3 можна зрозуміти що таке поліноміальна регресія. Це рівняння зі змінними **x** різних степенів. Поліноміальна регресія другого степеня має дві змінні: **x** та $\mathbf{x^2}$. Третій

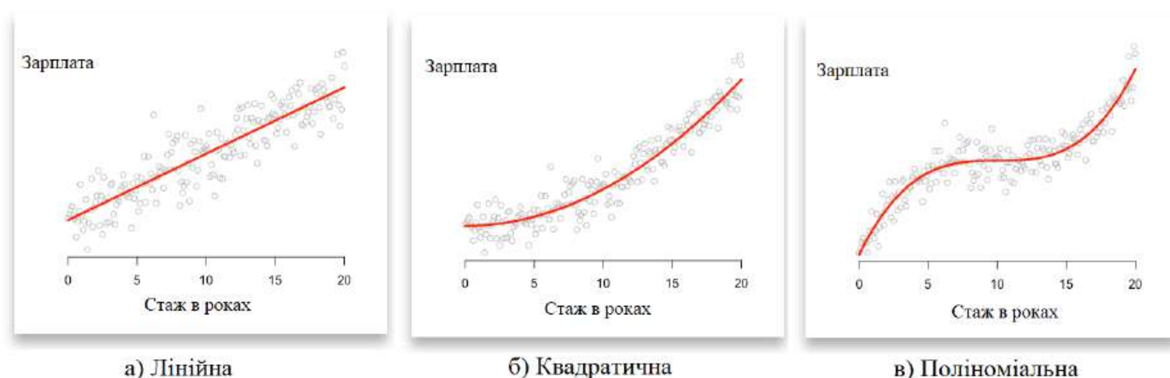



Рис. 9.3. Поліноміальна регресія

степінь має три змінні: **x**, $\mathbf{x^2}$ та $\mathbf{x^3}$.

Формула в цьому випадку моделюється так:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n + \varepsilon.$$

На рис. 9.4 побудовано графіки з двома змінними: **X** та **Y**. Очевидно, що зв'язок не є лінійним. Якщо додати лінійну регресію, то можна побачити, що модель не в змозі захопити шаблон (ліве зображення).

Тепер подивимося на праве зображення на рис. 9.3: до регресії додали поліном 5 степеня (тобто $y = x + x^2 + x^3 + x^4 + x^5$). Модель тепер фіксує шаблон набагато краще. Це ефективність поліноміальної регресії.

Повернемося до нашого прикладу. Вік не пов'язаний лінійно з доходом. Юний вік може мати дохід, близький до нуля, оскільки діти і молодь не працюють. Потім дохід збільшується у працездатному віці і зменшується під час виходу на пенсію. Це, як правило, перевернута форма U. Один зі способів захоплення цього шаблону – додавання `pow(2)` до регресії.

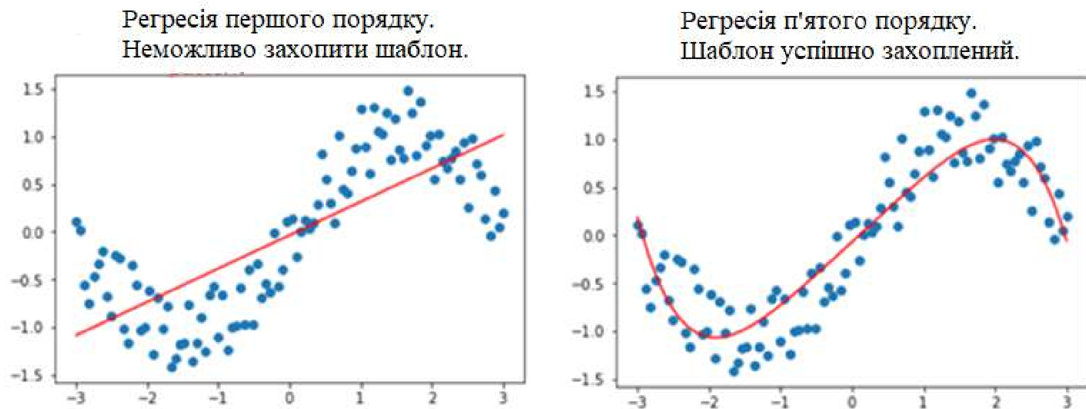



Рис. 9.4. Ефективність поліноміальної регресії

Подивимось, чи підвищується точність.

Нам треба додати цю нову функцію до набору даних і до списку безперервної функції.

Додаємо нову змінну в набір даних для тренування і тестів, тому зручніше написати функцію:

```
def square_var(df_t, df_te, var_name = 'age'):
    df_t['new'] = df_t[var_name].pow(2)
    df_te['new'] = df_te[var_name].pow(2)
    return df_t, df_te
```

Функція має три аргументи:

- df_t: визначає набір для тренування;
- df_te: визначає набір для тестування;
- var_name = 'age': визначає змінну для перетворення.

Можемо використати об'єкт `pow(2)` для квадрату змінної віку. Зазначимо, що нова змінна названа `'new'`.

Тепер, коли функція `square_var` записана, можна створити новий набір даних:

```
df_train_new, df_test_new = square_var(df_train, df_test, var_name =
'age')
```

Отже, новий набір даних має на одну ознаку більше.

```
print(df_train_new.shape, df_test_new.shape)
```

```
(32561, 16) (16281, 16)
```

Квадратична змінна названа `new` в наборі даних. Нам треба додати її до списку безперервних ознак.

```
CONTI_FEATURES_NEW  = ['age', 'fnlwgt','capital_gain', 'education_num',
'capital_loss', 'hours_week', 'new']
continuous_features_new = [tf.feature_column.numeric_column(k) for k in
CONTI_FEATURES_NEW]
```

**Зауважимо**, що ми змінили каталог Graph. Ми не можемо тренувати різні моделі в одному каталозі. Це означає, що треба змінити шлях аргументу `model_dir`. Якщо цього не зробити, TF буде видавати помилку.

```
model_1 = tf.estimator.LinearClassifier(
    model_dir="ongoing/train1",
    feature_columns=categorical_features+ continuous_features_new)
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'ongoing/train1',
'_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1820f04b70>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
FEATURES_NEW = ['age','workclass', 'fnlwgt', 'education',
'education_num', 'marital', 'occupation', 'relationship', 'race', 'sex',
'capital_gain', 'capital_loss', 'hours_week', 'native_country', 'new']
def get_input_fn(data_set, num_epochs=None, n_batch = 128,
shuffle=True):
    return tf.estimator.inputs.pandas_input_fn(
       x=pd.DataFrame({k: data_set[k].values for k in FEATURES_NEW}),
       y = pd.Series(data_set[LABEL].values),
       batch_size=n_batch,
       num_epochs=num_epochs,
       shuffle=shuffle)
```

Тепер, коли класифікатор розроблено за допомогою нового набору даних, можемо тренувати і оцінювати модель.

```
model_1.train(input_fn=get_input_fn(df_train,
                                      num_epochs=None,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into ongoing/train1/model.ckpt.
INFO:TensorFlow:loss = 88.722855, step = 1
INFO:TensorFlow:global_step/sec: 81.487
INFO:TensorFlow:loss = 70077.66, step = 101 (1.228 sec)
INFO:TensorFlow:global_step/sec: 111.169
INFO:TensorFlow:loss = 49522.082, step = 201 (0.899 sec)
INFO:TensorFlow:global_step/sec: 128.91
INFO:TensorFlow:loss = 107120.57, step = 301 (0.776 sec)
INFO:TensorFlow:global_step/sec: 132.546
INFO:TensorFlow:loss = 12814.152, step = 401 (0.755 sec)
INFO:TensorFlow:global_step/sec: 162.194
INFO:TensorFlow:loss = 19573.898, step = 501 (0.617 sec)
INFO:TensorFlow:global_step/sec: 204.852
INFO:TensorFlow:loss = 26381.986, step = 601 (0.488 sec)
```

```
INFO:TensorFlow:global_step/sec: 188.923
INFO:TensorFlow:loss = 23417.719, step = 701 (0.529 sec)
INFO:TensorFlow:global_step/sec: 192.041
INFO:TensorFlow:loss = 23946.049, step = 801 (0.521 sec)
INFO:TensorFlow:global_step/sec: 197.025
INFO:TensorFlow:loss = 3309.5786, step = 901 (0.507 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into
ongoing/train1/model.ckpt.
INFO:TensorFlow:Loss for final step: 28861.898.

<TensorFlow.python.estimator.canned.linear.LinearClassifier at
0x1820f04c88>
model_1.evaluate(input_fn=get_input_fn(df_test_new,
                                      num_epochs=1,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
INFO:TensorFlow:Calling model_fn.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-06-02-08:28:37
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from ongoing/train1/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Evaluation [100/1000]
INFO:TensorFlow:Finished evaluation at 2018-06-02-08:28:39
INFO:TensorFlow:Saving dict for global step 1000: accuracy = 0.7944229,
accuracy_baseline = 0.76377374, auc = 0.6093755, auc_precision_recall =
0.54885805, average_loss = 111.0046, global_step = 1000, label/mean =
0.23622628, loss = 14119.265, precision = 0.6682401, prediction/mean =
0.09116262, recall = 0.2576703

{'accuracy': 0.7944229,
 'accuracy_baseline': 0.76377374,
 'auc': 0.6093755,
 'auc_precision_recall': 0.54885805,
 'average_loss': 111.0046,
 'global_step': 1000,
 'label/mean': 0.23622628,
 'loss': 14119.265,
 'precision': 0.6682401,
 'prediction/mean': 0.09116262,
 'recall': 0.2576703}
```

Змінна в квадраті підвищила точність з 0,76 до 0,79. З’ясуємо, чи можна зробити ще краще, якщо поєднати разом терміни групування і взаємодії.

**Групування і взаємодія**

Як засвідчено, лінійний класифікатор не в змозі правильно зафіксувати шаблон віку і доходу, оскільки він використовує один ваговий коефіцієнт для кожної функції. Насамперед треба згрупувати цю функцію, щоб полегшити роботу класифікатора. Групування

перетворює числову ознаку на кілька інших, залежно від діапазону, до якого вона потрапляє, і кожна з цих нових ознак вказує, коли вік людини входить в цей діапазон.

Завдяки цим новим можливостям лінійна модель може охоплювати взаємозв’язок, вивчаючи різні вагові коефіцієнти для кожної групи.

У TF це можна зробити з `bucketized_column`. Треба додати діапазон значень для границь:

```
age = tf.feature_column.numeric_column('age')
age_buckets = tf.feature_column.bucketized_column(
    age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
```

Отже, ми вже знаємо, що залежність доходу від віку не є лінійною. Ще один спосіб вдосконалити модель – через взаємодію функцій. Мовою TF – це особливість перехрещування. Перехрещування функцій – це спосіб створити нові функції, що є комбінаціями існуючих, а це може бути корисним для лінійного класифікатора, який не може моделювати взаємодії між функціями.

Можемо розбити вік за допомогою іншої функції (наприклад, освіти). Отже, деякі групи, ймовірно, мають високий дохід, а інші низький (згадаймо про студента Ph.D.).

```
education_x_occupation = [tf.feature_column.crossed_column(
    ['education', 'occupation'], hash_bucket_size=1000)]
age_buckets_x_education_x_occupation =
[tf.feature_column.crossed_column(
    [age_buckets, 'education', 'occupation'], hash_bucket_size=1000)]
```

Щоб створити стовпчик перехресних функцій, використовуємо `cross_column` зі змінними для перетину в дужці. Величина `hash_bucket_size` вказує на максимальні можливості перетину. Для створення взаємодії між змінними (принаймні одна змінна має бути категорійною) можна використати `tf.feature_column.crossed_column`. Щоб використати цей об’єкт треба додати в квадратній дужці змінну для взаємодії, а також другий аргумент – розмір групи. Розмір групи – це максимально можлива чисельність групи в межах змінної. Тут встановимо її в 1000, оскільки не знаємо точного числа груп.

До того як додати їх до стовпців з ознаками, потрібно піднести в квадрат `age_buckets`. Також додаємо нові функції до стовпців ознак і готуємо оцінювач.

```
base_columns = [
    age_buckets,
]

model_imp = tf.estimator.LinearClassifier(
    model_dir="ongoing/train3",

feature_columns=categorical_features+base_columns+education_x_occupation
+age_buckets_x_education_x_occupation)
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'ongoing/train3',
'_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
```

```
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1823021be0>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
FEATURES_imp = ['age','workclass', 'education', 'education_num',
'marital',
                'occupation', 'relationship', 'race', 'sex',
'native_country', 'new']

def get_input_fn(data_set, num_epochs=None, n_batch = 128,
shuffle=True):
    return tf.estimator.inputs.pandas_input_fn(
       x=pd.DataFrame({k: data_set[k].values for k in FEATURES_imp}),
       y = pd.Series(data_set[LABEL].values),
       batch_size=n_batch,
       num_epochs=num_epochs,
       shuffle=shuffle)
```

Тепер можна оцінити нову модель і подивитися, чи збільшилась точність:

```
model_imp.train(input_fn=get_input_fn(df_train_new,
                                      num_epochs=None,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)

INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into ongoing/train3/model.ckpt.
INFO:TensorFlow:loss = 88.722855, step = 1
INFO:TensorFlow:global_step/sec: 94.969
INFO:TensorFlow:loss = 50.334488, step = 101 (1.054 sec)
INFO:TensorFlow:global_step/sec: 242.342
INFO:TensorFlow:loss = 56.153225, step = 201 (0.414 sec)
INFO:TensorFlow:global_step/sec: 213.686
INFO:TensorFlow:loss = 45.792007, step = 301 (0.470 sec)
INFO:TensorFlow:global_step/sec: 174.084
INFO:TensorFlow:loss = 37.485672, step = 401 (0.572 sec)
INFO:TensorFlow:global_step/sec: 191.78
INFO:TensorFlow:loss = 56.48449, step = 501 (0.524 sec)
INFO:TensorFlow:global_step/sec: 163.436
INFO:TensorFlow:loss = 32.528934, step = 601 (0.612 sec)
INFO:TensorFlow:global_step/sec: 164.347
INFO:TensorFlow:loss = 37.438057, step = 701 (0.607 sec)
INFO:TensorFlow:global_step/sec: 154.274
INFO:TensorFlow:loss = 61.1075, step = 801 (0.647 sec)
INFO:TensorFlow:global_step/sec: 189.14
INFO:TensorFlow:loss = 44.69645, step = 901 (0.531 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into
ongoing/train3/model.ckpt.
INFO:TensorFlow:Loss for final step: 44.18133.
```

```
<TensorFlow.python.estimator.canned.linear.LinearClassifier at
0x1823021cf8>
model_imp.evaluate(input_fn=get_input_fn(df_test_new,
                                      num_epochs=1,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)

INFO:TensorFlow:Calling model_fn.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-06-02-08:28:52
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from ongoing/train3/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Evaluation [100/1000]
INFO:TensorFlow:Finished evaluation at 2018-06-02-08:28:54
INFO:TensorFlow:Saving dict for global step 1000: accuracy = 0.8358209,
accuracy_baseline = 0.76377374, auc = 0.88401634, auc_precision_recall =
0.69599575, average_loss = 0.35122654, global_step = 1000, label/mean =
0.23622628, loss = 44.67437, precision = 0.68986726, prediction/mean =
0.23320661, recall = 0.55408216


{'accuracy': 0.8358209,
 'accuracy_baseline': 0.76377374,
 'auc': 0.88401634,
 'auc_precision_recall': 0.69599575,
 'average_loss': 0.35122654,
 'global_step': 1000,
 'label/mean': 0.23622628,
 'loss': 44.67437,
 'precision': 0.68986726,
 'prediction/mean': 0.23320661,
 'recall': 0.55408216}
```

Значення нової точності 83,58%. Це на чотири відсотки краще, ніж для попередньої моделі.

Отже, можемо додати термін регуляризації, щоб запобігти надмірному навчанню.

### Крок 5. Гіперпараметр: Lasso & Ridge

Наша модель може потерпати від **надмірного** або **недостатнього тренування:**

- Overfitting (надмірне): модель не може зробити передбачення для нових даних.
- Underfitting (недостатнє): модель не може захопити шаблон даних, наприклад, використана лінійна регресія, якщо дані нелінійні.

Якщо модель має багато параметрів і відносно низький обсяг даних, це призводить до поганих прогнозів. Уявіть, що одна група має лише три спостереження; модель буде обчислювати вагові коефіцієнти для цієї групи. Вони використовуються для прогнозування. Якщо спостереження за тестовим набором для цієї конкретної групи повністю відрізняються від навчального набору, то модель зробить неправильне передбачення. Під час оцінювання з навчальним набором точність буде доброю, але не буде доброю для тестового набору,

оскільки обчислені вагові коефіцієнти не є істинними для узагальнення шаблону. У такому випадку це не дає розумного прогнозу щодо нових (небачених) даних.

Регуляризація дає можливість контролювати складність перенавчання. Є два методи регуляризації:

- L1: Lasso (лассо);
- L2: Ridge (край).

У TF можемо додати ці два гіперпараметри в оптимізатор. Наприклад, чим більший гіперпараметр L2, тим більш низький і близький до нуля буде ваговий коефіцієнт. Лінія тренування буде дуже плоскою, а L2, близький до нуля, означає, що вагові коефіцієнти близькі до регулярної лінійної регресії.

Можемо самостійно спробувати різні значення гіперпараметрів і побачити, чи зможемо підвищити точність.

**Звертаємо увагу**, що якщо змінимо гіперпараметр, то треба видалити папку ongoing/train4, інакше модель стартує з попередньо навченої моделі.

Розглянемо точність із шумом:

```
model_regu = tf.estimator.LinearClassifier(
    model_dir="ongoing/train4",
feature_columns=categorical_features+base_columns+education_x_occupation
+age_buckets_x_education_x_occupation,
    optimizer=tf.train.FtrlOptimizer(
        learning_rate=0.1,
        l1_regularization_strength=0.9,
        l2_regularization_strength=5))
```

**Результат на виході:**

```
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_model_dir': 'ongoing/train4',
'_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_steps': None, '_save_checkpoints_secs': 600,
'_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100,
'_train_distribute': None, '_service': None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1820d9c128>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
model_regu.train(input_fn=get_input_fn(df_train_new,
                                       num_epochs=None,
                                       n_batch = 128,
                                       shuffle=False),
                                       steps=1000)
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into ongoing/train4/model.ckpt.
INFO:TensorFlow:loss = 88.722855, step = 1
INFO:TensorFlow:global_step/sec: 77.4165
INFO:TensorFlow:loss = 50.38778, step = 101 (1.294 sec)
```

```
INFO:TensorFlow:global_step/sec: 187.889
INFO:TensorFlow:loss = 55.38014, step = 201 (0.535 sec)
INFO:TensorFlow:global_step/sec: 201.895
INFO:TensorFlow:loss = 46.806694, step = 301 (0.491 sec)
INFO:TensorFlow:global_step/sec: 217.992
INFO:TensorFlow:loss = 38.68271, step = 401 (0.460 sec)
INFO:TensorFlow:global_step/sec: 193.676
INFO:TensorFlow:loss = 56.99398, step = 501 (0.516 sec)
INFO:TensorFlow:global_step/sec: 202.195
INFO:TensorFlow:loss = 33.263622, step = 601 (0.497 sec)
INFO:TensorFlow:global_step/sec: 216.756
INFO:TensorFlow:loss = 37.7902, step = 701 (0.459 sec)
INFO:TensorFlow:global_step/sec: 240.215
INFO:TensorFlow:loss = 61.732605, step = 801 (0.416 sec)
INFO:TensorFlow:global_step/sec: 220.336
INFO:TensorFlow:loss = 46.938225, step = 901 (0.456 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into
ongoing/train4/model.ckpt.
INFO:TensorFlow:Loss for final step: 43.4942.

<TensorFlow.python.estimator.canned.linear.LinearClassifier at
0x181ff39e48>
model_regu.evaluate(input_fn=get_input_fn(df_test_new,
                                      num_epochs=1,
                                      n_batch = 128,
                                      shuffle=False),
                                      steps=1000)
```

**Результат на виході:**

```
INFO:TensorFlow:Calling model_fn.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-06-02-08:29:07
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from ongoing/train4/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Evaluation [100/1000]
INFO:TensorFlow:Finished evaluation at 2018-06-02-08:29:09
INFO:TensorFlow:Saving dict for global step 1000: accuracy = 0.83833915,
accuracy_baseline = 0.76377374, auc = 0.8869794, auc_precision_recall =
0.7014905, average_loss = 0.34691378, global_step = 1000, label/mean =
0.23622628, loss = 44.12581, precision = 0.69720596, prediction/mean =
0.23662092, recall = 0.5579823

{'accuracy': 0.83833915,
 'accuracy_baseline': 0.76377374,
 'auc': 0.8869794,
 'auc_precision_recall': 0.7014905,
 'average_loss': 0.34691378,
 'global_step': 1000,
 'label/mean': 0.23622628,
 'loss': 44.12581,
```

```
 'precision': 0.69720596,
 'prediction/mean': 0.23662092,
 'recall': 0.5579823}
```

За допомогою введеного гіперпараметра ми трохи збільшили показники точності. У наступному розділі дізнаємось, як покращити лінійний класифікатор за допомогою методу ядра.

**Висновки**

Для тренування моделі треба:

- визначити ознаки: незалежні змінні: X;
- визначити мітку: залежна змінна: y;
- побудувати набори тренування/тест;
- визначити початкові вагові коефіцієнти;
- визначити функцію втрат: MSE;
- оптимізувати модель: градієнтний спуск;
- визначити:
    - швидкість навчання;
    - кількість епох;
    - розмір пакета;
    - число класів.

Отже, ми вивчили як використовувати високорівневий API для класифікатора лінійної регресії. Треба визначити:

1. Стовпці ознак. Якщо безперервна: tf.feature_column.numeric_column(). Можемо заповнити список за допомогою використання списку python.
2. Оцінювач: tf.estimator.LinearClassifier(feature_columns, model_dir, n_classes = 2).
3. Функцію для імпорту даних, розміри пакета і epoch: input_fn().

Після цього можна тренувати, оцінювати і робити передбачення з `train()`, `evaluate()` і `predict()`.

Для поліпшення параметрів моделі:

- використати поліноміальну регресію;
- використати терміни взаємодії: `tf.feature_column.crossed_column`
- додати параметр регуляризації.

# 10. Методи ядра в машинному навчанні: ядро Гаусса

Мета цього розділу – зробити набір даних лінійно відокремленим. Розділ має дві частини:

1. Особливість перетворення.
2. Тренування класифікатор ядра за допомогою TF.

У першій частині розглянемо ідею класифікатора ядра, а у другій частині – тренування класифікатора ядра за допомогою TF[23] [23]. Використаємо, як і раніше, набір даних про доросле населення. Завдання цього набору даних класифікувати дохід нижче і вище 50 тис., знаючи поведінку кожного домогосподарства.

[23] https://www.guru99.com/kernel-methods-machine-learning.html

## Для чого потрібні методи ядра

Мета кожного класифікатора – правильно передбачити класи. Для цього набір даних має бути відокремленим. На діаграмі (рис. 10.1) легко побачити, що всі точки над чорною лінією належать до першого класу, а інші – до другого класу. Однак набір даних тут дуже простий, а у більшості випадків дані не так легко розділити. Це важкий випадок для наївних класифікаторів типу логістичної регресії.

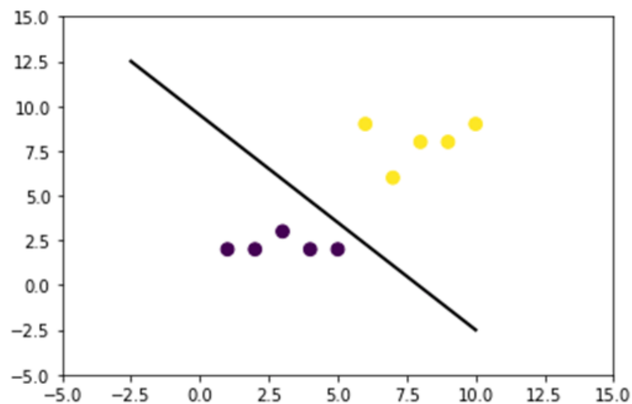



Рис. 10.1. Легкий для розділення набір даних

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x_lin = np.array([1,2,3,4,5,6,7,8,9,10])
y_lin = np.array([2,2,3,2,2,9,6,8,8,9])
label_lin = np.array([0,0,0,0,0,1,1,1,1,1])

fig = plt.figure()
ax=fig.add_subplot(111)
plt.scatter(x_lin, y_lin, c=label_lin, s=60)
plt.plot([-2.5, 10], [12.5, -2.5], 'k-', lw=2)
ax.set_xlim([-5,15])
ax.set_ylim([-5,15])plt.show()
```

На рис. 10.2 маємо набір даних, який не є лінійно відокремленим, а отже, якщо проведемо пряму, то більшість точок не будуть класифіковані у правильному класі.

Один зі способів розв’язати цю проблему – взяти набір даних і перетворити дані в іншу карту ознак. Це означає, що ми використаємо функцію для перетворення даних в інший план, який має бути вирівняним.

```
x = np.array([1,1,2,3,3,6,6,6,9,9,10,11,12,13,16,18])
y = np.array([18,13,9,6,15,11,6,3,5,2,10,5,6,1,3,1])
label = np.array([1,1,1,1,0,0,0,1,0,1,0,0,0,1,0,1])
fig = plt.figure()
plt.scatter(x, y, c=label, s=60)
plt.show()
```

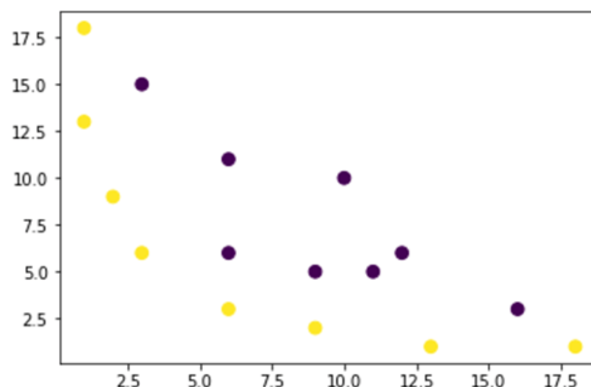



Рис. 10.2. Набір даних, лінійно не відокремлений

Дані, наведені на рис. 10.2, є двовимірним планом, який не можна розділити. Можемо спробувати перетворити ці дані у тривимірні, а це означає, що ми створимо фігуру з трьома осями.

У нашому прикладі застосуємо поліноміальне відображення, щоб звести наші дані до тривимірності. Формула перетворення даних така:

$$\phi(x, y) = (x^2, \sqrt{(2)}xy, y^2) .$$

Визначаємо функцію в Python для побудови нової карти ознак.

Можемо використати numpy для програмування наведеної формули:

| **Формула** | **Еквівалентний код Numpy** |
|---|---|
| x | x[:,0]** |
| y | x[:,1] |
| $x^2$ | x[:,0]**2 |
| $\sqrt{(2)}$ | np.sqrt(2)* |
| xy | x[:,0]*x[:,1] |
| $y^2$ | x[:,1]**2 |

```
### Ілюстрація
def mapping(x, y):
    x = np.c_[(x, y)]
    if len(x) > 2:
        x_1 = x[:,0]**2
        x_2 = np.sqrt(2)*x[:,0]*x[:,1]
        x_3 = x[:,1]**2
    else:
        x_1 = x[0]**2
        x_2 = np.sqrt(2)*x[0]*x[1]
        x_3 = x[1]**2
    trans_x = np.array([x_1, x_2, x_3])
    return trans_x
```

Нова карта буде мати 3 розміри з 16 точками:

```
x_1  = mapping(x, y)
x_1.shape
(3, 16)
```

Отже, побудуємо нову діаграму з трьома осями x, y і z відповідно.

```
# Графік
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x_1[0], x_1[1], x_1[2], c=label, s=60)
ax.view_init(30, 185)ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
plt.show()
```

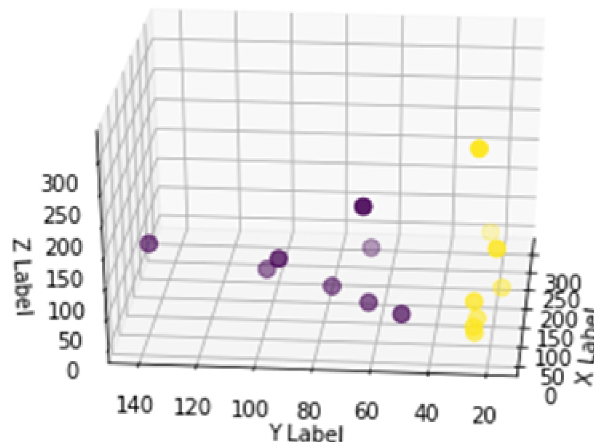



Рис. 10.3.

Бачимо покращення, але якщо змінити орієнтацію ділянки, то стане зрозуміло, що тепер набір даних можна розділити.

```
# Графік
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x_1[0], x_1[1], x_1[1], c=label, s=60)
ax.view_init(0, -180)ax.set_ylim([150,-50])
ax.set_zlim([-10000,10000])
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')plt.show()
```

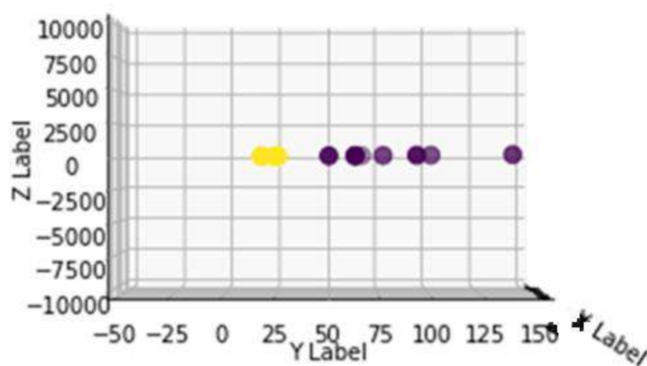



Рис. 10.4.

Можливо для того, щоб маніпулювати великим набором даних, треба буде створити більше двох вимірів, однак тоді виникне велика проблема використання вищевказаного методу, оскільки це потребуватиме дуже багато часу, а у комп'ютера може не вистачити пам'яті.

Найпоширеніший спосіб подолати цю проблему – використати ядро.

## Що таке ядро в машинному навчанні

Ідея полягає у використанні функціонального простору більших розмірів, щоб зробити дані майже лінійно відокремленими, як показано на рис. 10.4.

Є безліч просторів більших розмірів, щоб зробити точки даних відокремленими. Наприклад, було засвідчено, що поліноміальне відображення – це чудовий початок. Також продемонстровано, що за наявності великої кількості даних ця трансформація не є ефективною. Натомість можна використати функцію ядра для зміни даних, не будуючи новий план ознак.

Магія ядра полягає в тому, щоб знайти функцію, яка допомагає уникнути всіх неприємностей, пов’язаних з обчисленням великих розмірів. Результатом ядра є скаляр, а отже, якщо сказати іншими словами, то ми повертаємося до одновимірного простору.

Після того як знайдемо цю функцію, можемо підключити її до стандартного лінійного класифікатора.

Розглянемо приклад для розуміння концепції ядра. У нас є два вектори: **x1** і **x2**. Мета – створити вищий вимір за допомогою поліноміального відображення. На виході отримаємо скалярний добуток нової карти ознак. З описаного вище способу нам треба:

1. Перетворити **x1** і **x2** у новий вимір.
2. Обчислити скалярний добуток: спільний для всіх ядер.
3. Перетворити **x1** і **x2** у новий вимір.

Можемо використати функцію, створену вище, для обчислення вищого виміру.

```
## Ядро
x1 = np.array([3,6])
x2 = np.array([10,10]

x_1 = mapping(x1, x2)
print(x_1)
```

**Результат на виході**:

```
[[  9.          100.        ]
      [ 25.45584412 141.42135624]
      [ 36.         100.        ]]
```

Обчислюємо скалярний добуток.

Можемо використати об’єкт `dot` з numpy для обчислення скалярного добутку між першим і другим вектором, що зберігаються в x_1.

```
print(np.dot(x_1[:,0], x_1[:,1]))

8100.0
```

Результат дорівнює 8100. Отже, є проблема: нам треба зберегти у пам’яті нову карту ознак для обчислення скалярного добутку. Якщо ми маємо набір даних з мільйонами записів, то це розрахунково неефективно.

Натомість можемо використати **ядро полінома** для обчислення скалярного добутку без перетворення вектора. Ця функція обчислює скалярний добуток x1 і x2 так, ніби ці два вектори перетворені у вищий вимір, тобто функція ядра обчислює результати скалярного добутку з іншого простору ознак.

Можемо записати функцію ядра полінома у Python, як описано нижче:

```
def polynomial_kernel(x, y, p=2):
```

```
    return (np.dot(x, y)) ** p
```

Це квадрат скалярного добутку двох векторів. Далі повертаємо другий степінь ядра полінома. Результат дорівнює іншому методу. Це магія ядра.

```
polynomial_kernel(x1, x2, p=2)
```

```
8100
```

## Типи методів ядра

Є багато різних ядер. Найпростішим є лінійне ядро. Ця функція працює доволі добре для класифікації тексту. Відомі інші ядра:

- поліномне ядро;
- ядро Гаусса.

У прикладі з TF використаємо Random Fourier. TF має вбудований оцінювач для обчислення нового простору ознак. Ця функція є наближенням функції ядра Гаусса:

$$e^{\frac{-\| x - y \|^2}{2\sigma^2}}.$$

Функція обчислює схожість між точками даних у значно більшому розмірному просторі.

## Тренування класифікатора ядра Гаусса з TensorFlow

Завдання алгоритму – класифікувати заробіток домогосподарств, які заробляють більше або менше 50 тис.

Оцінимо логістичну регресію, щоб мати модель для порівняння. Після цього навчимо класифікатор ядра, щоб побачити, чи зможемо отримати кращі результати.

Використовуємо такі змінні з даних про доросле населення:

- вік;
- клас роботи;
- fnlwgt;
- освіта;
- education_num;
- сімейний стан;
- професія;
- відносини;
- раса;
- стать;
- capital_gain;
- capital_loss;
- hours_week;
- країна народження;
- мітка.

Пройдемо кроки, які робили раніше, перш ніж тренувати і оцінювати модель:

- Крок 1: Імпортуємо бібліотеки.
- Крок 2: Імпортуємо дані.

- Крок 3: Готуємо дані.
- Крок 4: Будуємо input_fn.
- Крок 5: Будуємо логістичну модель: базова модель.
- Крок 6: Оцінюємо модель.
- Крок 7: Побудуємо класифікатор ядра.
- Крок 8: Оцінимо класифікатор ядра.

**Крок 1. Імпортуємо бібліотеки**

Щоб імпортувати і навчати модель, треба імпортувати TF, pandas та numpy:

```
# Імпорт numpy як np
from sklearn.model_selection
import train_test_split
import tensorflow as tf
import pandas as pd
import numpy as np
```

**Крок 2. Імпортуємо дані**

Завантажимо дані з вебсайта[24] [24] і імпортуємо їх як кадр даних pandas:

```
## Визначаємо дані про шлях
COLUMNS = ['age','workclass', 'fnlwgt', 'education', 'education_num',
'marital', 'occupation', 'relationship', 'race', 'sex', 'capital_gain',
'capital_loss', 'hours_week', 'native_country', 'label']
PATH = "https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data"
PATH_test ="https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.test
### Імпорт
df_train = pd.read_csv(PATH, skipinitialspace=True, names = COLUMNS,
index_col=False)
df_test = pd.read_csv(PATH_test,skiprows = 1, skipinitialspace=True,
names = COLUMNS, index_col=False)
```

Отже, якщо визначені набори для тренування і тестів, можемо змінити тип мітки стовпця з рядкової на ціле число. Нагадуємо, що TF не приймає рядкове значення для мітки.

```
label = {'<=50K': 0,'>50K': 1}
df_train.label = [label[item] for item in df_train.label]
label_t = {'<=50K.': 0,'>50K.': 1}
df_test.label = [label_t[item] for item in df_test.label]
df_train.shape

(32561, 15)
```

**Крок 3. Готуємо дані**

Набір даних містить як безперервні, так і категорійні функції. Доброю практикою є стандартизація значень безперервних змінних. Можемо використати функцію `StandardScaler` від sci-kit learn. Також створюємо визначену користувачем функцію, щоб полегшити перетворення тренувальних і тестових наборів. Зауважимо, що ми об’єднуємо безперервні і категорійні змінні в загальний набір даних, а масив повинен мати тип float32.

[24] https://archive.ics.uci.edu/ml/machine-learning-databases/adult/

```
COLUMNS_INT = ['age','fnlwgt','education_num','capital_gain',
'capital_loss', 'hours_week']
CATE_FEATURES = ['workclass', 'education', 'marital', 'occupation',
'relationship', 'race', 'sex', 'native_country']
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing

def prep_data_str(df):
    scaler = StandardScaler()
    le = preprocessing.LabelEncoder()
    df_toscale = df[COLUMNS_INT]
    df_scaled = scaler.fit_transform(df_toscale.astype(np.float64))
    X_1 = df[CATE_FEATURES].apply(le.fit_transform)
    y = df['label'].astype(np.int32)
    X_conc = np.c_[df_scaled, X_1].astype(np.float32)
    return X_conc, y
```

**Крок 4. Будуємо input_fn**

Функція перетворення готова, тож можемо перетворити набір даних і створити функцію `input_fn`.

```
X_train, y_train = prep_data_str(df_train)
X_test, y_test = prep_data_str(df_test)
print(X_train.shape)
(32561, 14)
```

На наступному кроці будемо тренувати логістичну регресію. Це дасть базову точність. Мета полягає в тому, щоб обійти базову лінію за допомогою іншого алгоритму, а саме – класифікатора Kernel.

**Крок 5. Будуємо логістичну модель: базова модель**

Будуємо стовпець ознак за допомогою об’єкта `real_valued_column`. Переконаємося, що всі змінні є реальними числовими даними.

```
feat_column = tf.contrib.layers.real_valued_column('features',
dimension=14)
```

Оцінювач визначаємо з використанням оцінювача TF, вказуємо стовпці ознак і місце, де зберігається граф:

```
estimator = tf.estimator.LinearClassifier(feature_columns=[feat_column],
                                          n_classes=2,
                                          model_dir = "kernel_log"
                                         )
INFO:TensorFlow:Using default config.INFO:TensorFlow:Using config:
{'_model_dir': 'kernel_log', '_tf_random_seed': None,
'_save_summary_steps': 100, '_save_checkpoints_steps': None,
'_save_checkpoints_secs': 600, '_session_config': None,
'_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000,
'_log_step_count_steps': 100, '_train_distribute': None, '_service':
None, '_cluster_spec':
<TensorFlow.python.training.server_lib.ClusterSpec object at
0x1a2003f780>, '_task_type': 'worker', '_task_id': 0,
'_global_id_in_cluster': 0, '_master': '', '_evaluation_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

Тренувати логістичну регресію будемо з використанням мініпакетів розміром 200.

```
# Тренування моделі
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"features": X_train},
    y=y_train,
    batch_size=200,
    num_epochs=None,
    shuffle=True)
```

Тренуємо модель за 1 000 ітерацій:

```
estimator.train(input_fn=train_input_fn, steps=1000)
INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into kernel_log/model.ckpt.
INFO:TensorFlow:loss = 138.62949, step = 1
INFO:TensorFlow:global_step/sec: 324.16
INFO:TensorFlow:loss = 87.16762, step = 101 (0.310 sec)
INFO:TensorFlow:global_step/sec: 267.092
INFO:TensorFlow:loss = 71.53657, step = 201 (0.376 sec)
INFO:TensorFlow:global_step/sec: 292.679
INFO:TensorFlow:loss = 69.56703, step = 301 (0.340 sec)
INFO:TensorFlow:global_step/sec: 225.582
INFO:TensorFlow:loss = 74.615875, step = 401 (0.445 sec)
INFO:TensorFlow:global_step/sec: 209.975
INFO:TensorFlow:loss = 76.49044, step = 501 (0.475 sec)
INFO:TensorFlow:global_step/sec: 241.648
INFO:TensorFlow:loss = 66.38373, step = 601 (0.419 sec)
INFO:TensorFlow:global_step/sec: 305.193
INFO:TensorFlow:loss = 87.93341, step = 701 (0.327 sec)
INFO:TensorFlow:global_step/sec: 396.295
INFO:TensorFlow:loss = 76.61518, step = 801 (0.249 sec)
INFO:TensorFlow:global_step/sec: 359.857
INFO:TensorFlow:loss = 78.54885, step = 901 (0.277 sec)
INFO:TensorFlow:Saving checkpoints for 1000 into kernel_log/model.ckpt.
INFO:TensorFlow:Loss for final step: 67.79706.

<TensorFlow.python.estimator.canned.linear.LinearClassifier at
0x1a1fa3cbe0>
```

**Крок 6. Оцінюємо модель**

Обираємо оцінювач numpy для оцінювання моделі. Використаємо вхідний набір даних для оцінювання:

```
# Оцінювання
test_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"features": X_test},
    y=y_test,
    batch_size=16281,
    num_epochs=1,
    shuffle=False)
estimator.evaluate(input_fn=test_input_fn, steps=1)
```

```
INFO:TensorFlow:Calling model_fn.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-AUCs; please switch to "careful_interpolation" instead.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-07-12-15:58:22
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from kernel_log/model.ckpt-1000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Evaluation [1/1]
INFO:TensorFlow:Finished evaluation at 2018-07-12-15:58:23
INFO:TensorFlow:Saving dict for global step 1000: accuracy = 0.82353663, accuracy_baseline = 0.76377374, auc = 0.84898686, auc_precision_recall = 0.67214864, average_loss = 0.3877216, global_step = 1000, label/mean = 0.23622628, loss = 6312.495, precision = 0.7362797, prediction/mean = 0.21208474, recall = 0.39417577
{'accuracy': 0.82353663,
 'accuracy_baseline': 0.76377374,
 'auc': 0.84898686,
 'auc_precision_recall': 0.67214864,
 'average_loss': 0.3877216,
 'global_step': 1000,
 'label/mean': 0.23622628,
 'loss': 6312.495,
 'precision': 0.7362797,
 'prediction/mean': 0.21208474,
 'recall': 0.39417577}
```

Маємо точність 82%. У наступному розділі виконаємо тренування логістичного класифікатора з використанням класифікатора ядра.

**Крок 7. Будуємо класифікатор ядра**

Оцінювач ядра не дуже сильно відрізняється від традиційного лінійного класифікатора, принаймні за побудовою. Ідея полягає у використанні потужності явного ядра з лінійним класифікатором.

Для підготовки класифікатора ядра нам потрібні два попередньо визначені оцінювачі, доступні в TF:

- RandomFourierFeatureMapper;
- KernelLinearClassifier.

У попередній частині ми дізналися, що треба перетворити малу розмірність у велику за допомогою функції ядра. Отже, будемо використовувати RandomFourier, який є наближенням функції Гаусса. Дуже добре, що TF має функцію `RandomFourierFeatureMapper` у своїй бібліотеці. Модель може бути навчена за допомогою оцінювача KernelLinearClassifier.

Для побудови моделі виконаємо такі кроки:

1. Встановимо функцію ядра великої розмірності.
2. Встановимо гіперпараметр L2.
3. Побудуємо модель.
4. Виконаємо тренування моделі.
5. Оцінимо модель.

**Крок A. Встановлюємо функції ядра великої розмірності**

Поточний набір даних містить 14 ознак, які перетворимо на новий 5000-вимірний вектор великої розмірності. Для досягнення трансформації використаємо випадкові функції Фур'є. Якщо згадати формулу ядра Гаусса, то побачимо, що є параметр стандартного відхилення для визначення. Цей параметр управління параметром схожості використовується під час класифікації.

Можемо налаштувати всі параметри в `RandomFourierFeatureMapper` за допомогою:

- input_dim = 14;
- output_dim= 5 000;
- stddev=4.

```
### Підготовка ядра
kernel_mapper =
tf.contrib.kernel_methods.RandomFourierFeatureMapper(input_dim=14,
output_dim=5000, stddev=4, name='rffm')
```

Треба сконструювати картограф ядра, використовуючи створені раніше стовпці ознак `feat_column`.

```
### Карта ядра
kernel_mappers = {feat_column: [kernel_mapper]}
```

**Крок B. Встановлюємо гіперпараметр L2**

Щоб запобігти перенавчанню, штрафуємо функцію втрат за допомогою регулятора L2. Встановимо регулятор L2 в 0.1 і швидкість навчання в 5:

```
optimizer = tf.train.FtrlOptimizer(learning_rate=5,
l2_regularization_strength=0.1)
```

**Крок C. Будуємо модель**

Наступний крок схожий на лінійну класифікацію. Використовуємо вбудований оцінювач KernelLinearClassifier. Звертаємо увагу, що ми додаємо попередньо визначену карту ядра і змінюємо каталог для моделі.

```
### Підготовка оцінювача
estimator_kernel = tf.contrib.kernel_methods.KernelLinearClassifier(
    n_classes=2,
    optimizer=optimizer,
    kernel_mappers=kernel_mappers,
    model_dir="kernel_train")

WARNING:TensorFlow:From /Users/Thomas/anaconda3/envs/hello-
tf/lib/python3.6/site-
packages/TensorFlow/contrib/kernel_methods/python/kernel_estimators.py:3
05: multi_class_head (from
TensorFlow.contrib.learn.python.learn.estimators.head) is deprecated and
will be removed in a future version.
Instructions for updating:
Please switch to tf.contrib.estimator.*_head.
WARNING:TensorFlow:From /Users/Thomas/anaconda3/envs/hello-
tf/lib/python3.6/site-
packages/TensorFlow/contrib/learn/python/learn/estimators/estimator.py:1
179: BaseEstimator.__init__ (from
```

```
TensorFlow.contrib.learn.python.learn.estimators.estimator) is
deprecated and will be removed in a future version.
Instructions for updating:
Please replace uses of any Estimator from tf.contrib.learn with an
Estimator from tf.estimator.*
WARNING:TensorFlow:From /Users/Thomas/anaconda3/envs/hello-
tf/lib/python3.6/site-
packages/TensorFlow/contrib/learn/python/learn/estimators/estimator.py:4
27: RunConfig.__init__ (from
TensorFlow.contrib.learn.python.learn.estimators.run_config) is
deprecated and will be removed in a future version.
Instructions for updating:
When switching to tf.estimator.Estimator, use tf.estimator.RunConfig
instead.
INFO:TensorFlow:Using default config.
INFO:TensorFlow:Using config: {'_task_type': None, '_task_id': 0,
'_cluster_spec': <TensorFlow.python.training.server_lib.ClusterSpec
object at 0x1a200ae550>, '_master': '', '_num_ps_replicas': 0,
'_num_worker_replicas': 0, '_environment': 'local', '_is_chief': True,
'_evaluation_master': '', '_train_distribute': None, '_tf_config':
gpu_options {
  per_process_gpu_memory_fraction: 1.0
}
, '_tf_random_seed': None, '_save_summary_steps': 100,
'_save_checkpoints_secs': 600, '_log_step_count_steps': 100,
'_session_config': None, '_save_checkpoints_steps': None,
'_keep_checkpoint_max': 5, '_keep_checkpoint_every_n_hours': 10000,
'_model_dir': 'kernel_train'}
```

**Крок D. Тренуємо модель**

Отже, якщо класифікатор ядра побудовано, ми можемо починати тренування. Повторимо модель 2 000 разів.

```
### Тренуємо
estimator_kernel.fit(input_fn=train_input_fn, steps=2000)

WARNING:TensorFlow:Casting <dtype: 'int32'> labels to bool.
WARNING:TensorFlow:Casting <dtype: 'int32'> labels to bool.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:From /Users/Thomas/anaconda3/envs/hello-
tf/lib/python3.6/site-
packages/TensorFlow/contrib/learn/python/learn/estimators/head.py:678:
ModelFnOps.__new__ (from
TensorFlow.contrib.learn.python.learn.estimators.model_fn) is deprecated
and will be removed in a future version.
Instructions for updating:
When switching to tf.estimator.Estimator, use
tf.estimator.EstimatorSpec. You can use the `estimator_spec` method to
create an equivalent one.
INFO:TensorFlow:Create CheckpointSaverHook.
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Saving checkpoints for 1 into kernel_train/model.ckpt.
```

```
INFO:TensorFlow:loss = 0.6931474, step = 1
INFO:TensorFlow:global_step/sec: 86.6365
INFO:TensorFlow:loss = 0.39374447, step = 101 (1.155 sec)
INFO:TensorFlow:global_step/sec: 80.1986
INFO:TensorFlow:loss = 0.3797774, step = 201 (1.247 sec)
INFO:TensorFlow:global_step/sec: 79.6376
INFO:TensorFlow:loss = 0.3908726, step = 301 (1.256 sec)
INFO:TensorFlow:global_step/sec: 95.8442
INFO:TensorFlow:loss = 0.41890752, step = 401 (1.043 sec)
INFO:TensorFlow:global_step/sec: 93.7799
INFO:TensorFlow:loss = 0.35700393, step = 501 (1.066 sec)
INFO:TensorFlow:global_step/sec: 94.7071
INFO:TensorFlow:loss = 0.35535482, step = 601 (1.056 sec)
INFO:TensorFlow:global_step/sec: 90.7402
INFO:TensorFlow:loss = 0.3692882, step = 701 (1.102 sec)
INFO:TensorFlow:global_step/sec: 94.4924
INFO:TensorFlow:loss = 0.34746957, step = 801 (1.058 sec)
INFO:TensorFlow:global_step/sec: 95.3472
INFO:TensorFlow:loss = 0.33655524, step = 901 (1.049 sec)
INFO:TensorFlow:global_step/sec: 97.2928
INFO:TensorFlow:loss = 0.35966292, step = 1001 (1.028 sec)
INFO:TensorFlow:global_step/sec: 85.6761
INFO:TensorFlow:loss = 0.31254214, step = 1101 (1.167 sec)
INFO:TensorFlow:global_step/sec: 91.4194
INFO:TensorFlow:loss = 0.33247527, step = 1201 (1.094 sec)
INFO:TensorFlow:global_step/sec: 82.5954
INFO:TensorFlow:loss = 0.29305756, step = 1301 (1.211 sec)
INFO:TensorFlow:global_step/sec: 89.8748
INFO:TensorFlow:loss = 0.37943482, step = 1401 (1.113 sec)
INFO:TensorFlow:global_step/sec: 76.9761
INFO:TensorFlow:loss = 0.34204718, step = 1501 (1.300 sec)
INFO:TensorFlow:global_step/sec: 73.7192
INFO:TensorFlow:loss = 0.34614792, step = 1601 (1.356 sec)
INFO:TensorFlow:global_step/sec: 83.0573
INFO:TensorFlow:loss = 0.38911164, step = 1701 (1.204 sec)
INFO:TensorFlow:global_step/sec: 71.7029
INFO:TensorFlow:loss = 0.35255936, step = 1801 (1.394 sec)
INFO:TensorFlow:global_step/sec: 73.2663
INFO:TensorFlow:loss = 0.31130585, step = 1901 (1.365 sec)
INFO:TensorFlow:Saving checkpoints for 2000 into
kernel_train/model.ckpt.
INFO:TensorFlow:Loss for final step: 0.37795097.

KernelLinearClassifier(params={'head':
<TensorFlow.contrib.learn.python.learn.estimators.head._BinaryLogisticHe
ad object at 0x1a2054cd30>, 'feature_columns':
{_RealValuedColumn(column_name='features_MAPPED', dimension=5000,
default_value=None, dtype=tf.float32, normalizer=None)}, 'optimizer':
<TensorFlow.python.training.ftrl.FtrlOptimizer object at 0x1a200aec18>,
'kernel_mappers': {_RealValuedColumn(column_name='features',
dimension=14, default_value=None, dtype=tf.float32, normalizer=None):
[<TensorFlow.contrib.kernel_methods.python.mappers.random_fourier_featur
es.RandomFourierFeatureMapper object at 0x1a200ae400>]}})
```

**Крок Е. Оцінюємо модель**

Оцінюємо параметри своєї моделі, адже ми маємо отримати кращий результат, ніж з логістичною регресією.

```
# Оцінюємо та звітуємо про показники.
eval_metrics = estimator_kernel.evaluate(input_fn=test_input_fn,
steps=1)
WARNING:TensorFlow:Casting <dtype: 'int32'> labels to bool.
WARNING:TensorFlow:Casting <dtype: 'int32'> labels to bool.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
WARNING:TensorFlow:Trapezoidal rule is known to produce incorrect PR-
AUCs; please switch to "careful_interpolation" instead.
INFO:TensorFlow:Starting evaluation at 2018-07-12-15:58:50
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from kernel_train/model.ckpt-2000
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Evaluation [1/1]
INFO:TensorFlow:Finished evaluation at 2018-07-12-15:58:51
INFO:TensorFlow:Saving dict for global step 2000: accuracy = 0.83975184,
accuracy/baseline_label_mean = 0.23622628,
accuracy/threshold_0.500000_mean = 0.83975184, auc = 0.8904007,
auc_precision_recall = 0.72722375, global_step = 2000,
labels/actual_label_mean = 0.23622628, labels/prediction_mean =
0.23786618, loss = 0.34277728,
precision/positive_threshold_0.500000_mean = 0.73001117,
recall/positive_threshold_0.500000_mean = 0.5104004
```

Кінцева точність 84%, що на 2% краще порівняно з логістичною регресією. Існує компроміс між підвищенням точності і вартістю розрахунків. Треба подумати про те, а чи варте покращення на 2% витраченого часу (який витрачає інший класифікатор) і чи має воно переконливий вплив на наш бізнес.

**Висновки**

Ядро – чудовий інструмент для перетворення нелінійних даних у (майже) лінійні. Недолік цього методу – це трудомісткі і дорогі обчислення.

Наведено найважливіший код для тренування класифікатора ядра.

Встановлюємо функцію ядра високої розмірності:

- input_dim = 14;
- output_dim= 5 000;
- stddev=4.

```
### Підготовка Kernelkernel_mapper =
tf.contrib.kernel_methods.RandomFourierFeatureMapper(input_dim=14,
output_dim=5000, stddev=4, name='rffm')
```

Встановлюємо гіперпараметр L2

```
optimizer = tf.train.FtrlOptimizer(learning_rate=5,
l2_regularization_strength=0.1)
```

Будуємо модель

```
estimator_kernel = tf.contrib.kernel_methods.KernelLinearClassifier(
n_classes=2,
```

```
    optimizer=optimizer,
    kernel_mappers=kernel_mappers,
    model_dir="kernel_train")
```

Тренуємо модель

```
estimator_kernel.fit(input_fn=train_input_fn, steps=2000)
```

Оцінюємо модель

```
eval_metrics = estimator_kernel.evaluate(input_fn=test_input_fn,
steps=1)
```

# 11. Штучна нейронна мережа з TensorFlow

## Що таке штучна нейронна мережа

Штучна нейронна мережа (Artificial Neural Network – **ANN**) складається з чотирьох основних об’єктів[25] [25]:

- Шари: все навчання відбувається в шарах. Є три шари (вхідний, прихований і вихідний).
- Ознаки і мітка: введення даних у мережу (ознаки) і вихід із мережі (мітки).
- Функція втрат: метрика, яка використовується для оцінювання ефективності фази навчання.
- Оптимізатор: покращує навчання шляхом оновлення знань у мережі.

З рис. 11.1 можна зрозуміти основний механізм.

Нейронна мережа бере вхідні дані і підштовхує їх до ансамблю шарів. Мережі необхідно оцінити свою ефективність за допомогою функції втрат. Функція втрат дає мережі уявлення про шлях, який вона має пройти, перш ніж знання будуть засвоєні. Мережі треба вдосконалити свої знання за допомогою оптимізатора.

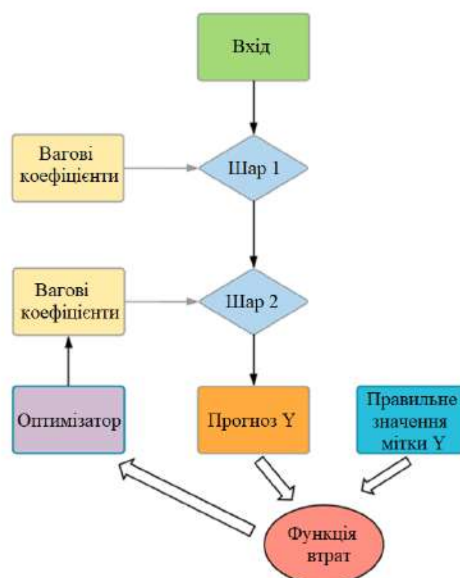



Рис. 11.1. Основні механізми штучної нейронної мережі

[25] https://www.guru99.com/artificial-neural-network-tutorial.html

Програма приймає деякі вхідні значення і просуває їх на два повністю пов'язаних шари. Уявіть, що є математична проблема, а отже, перше, що нам потрібно зробити, – прочитати відповідний розділ, щоб розв'язати проблему. Застосуємо свої нові знання для розв'язання проблеми, але є велика ймовірність того, що ми не дуже добре все зрозуміли. Те ж саме стосується і мережі. Коли вона вперше побачить дані і зробить прогноз, то він не буде повністю відповідати фактичним даним.

Для вдосконалення своїх знань мережа використовує оптимізатор. За нашою аналогією оптимізатором можна вважати перечитування розділу. Коли ми неодноразово перечитуємо матеріал, то отримуємо нові уявлення (урок). Так само мережа використовує оптимізатор, оновлює свої знання і тестує свої нові знання, щоб перевірити, чи потрібно ще вчитися. Програма повторюватиме цей крок допоки не зробить найменшу можливу помилку.

За аналогією з математикою це означає, що ми багато разів читаємо розділ підручника, аж поки не вивчимо досконало зміст курсу. Якщо навіть після декількох разів читання ми продовжуємо робити помилку, то це означає, що ми досягли потенціалу знань з поточним матеріалом. Нам треба використати інший підручник або спробувати інший метод, щоб покращити свою оцінку. Для нейронної мережі – це той самий процес. Якщо точність зовсім не 100%, але крива плоска, то це означає, що з поточною архітектурою вона вже нічого не може навчитися. Мережа має бути оптимізована для покращення знань.

## Нейромережева архітектура

### Шари

Шар – це місце, де відбувається все навчання. Усередині шару перебуває нескінченна кількість вагових коефіцієнтів (нейронів). Типова нейронна мережа часто опрацьовується щільно з'єднаними шарами (їх також називають повністю з'єднаними шарами). Це означає, що всі входи підключені до виходу.

Типова нейронна мережа приймає вектор введення і скаляр, який містить мітки. Найпростіше налаштування – це двійкова класифікація, що має лише два класи: 0 і 1.

Мережа приймає вхід, надсилає його до всіх підключених вузлів і обчислює сигнал за допомогою функції активації ***f***.

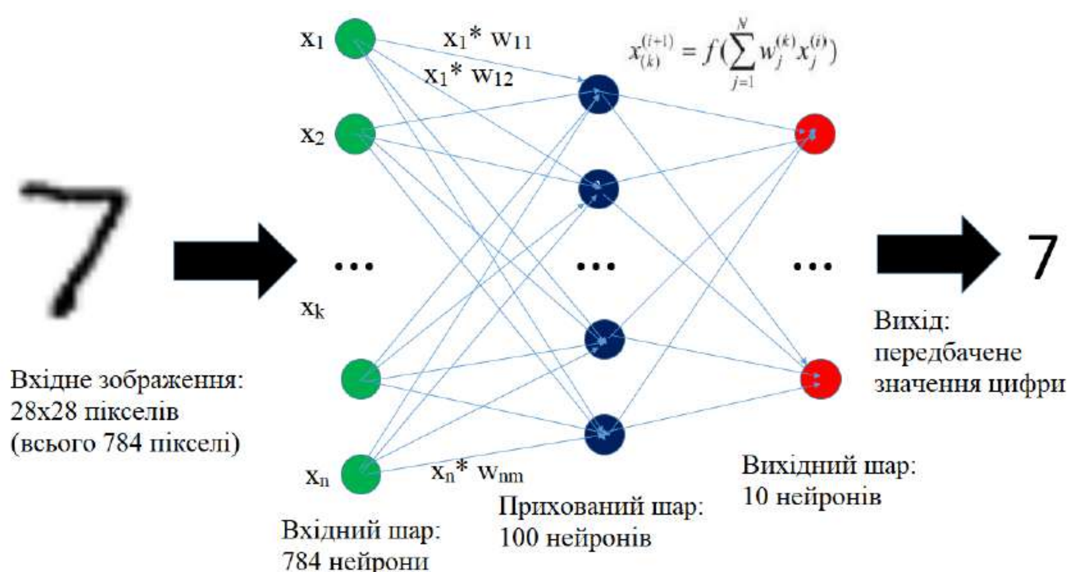



Рис. 11.2. Структура типової нейронної мережі з 10 класами

На рис. 11.2 відображено цю ідею. Перший шар – це вхідні значення для другого, так званого, прихованого шару, що приймає зважені вхідні значення від попереднього шару.

1. Перші вузли – це вхідні значення.
2. Нейрон розкладається на вхідну частину і функцію активації. Ліва частина отримує весь вхід з попереднього шару. Права частина – це сума входів, що переходить у функцію активації.
3. Вихідне значення, обчислене з прихованих шарів, використовується для прогнозування. Для класифікації воно дорівнює кількості класів. Для регресії прогнозується лише одне значення.

**Функція активації**

Функція активації вузла визначає вихід, заданий набором входів. Нам потрібна функція активації для того, щоби дати змогу мережі навчати нелінійний шаблон. Одна із загальних функцій активації – **ReLU (Rectified Linear Unit)**, випрямлена лінійна одиниця (рис. 11.3). Функція дає нуль для всіх від'ємних значень.

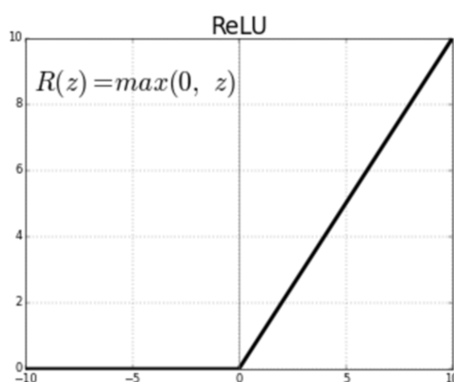



Рис. 11.3. Функція активації ReLU

Іншими функціями активації є (рис. 9.1):

- Piecewise Linear;
- Sigmoid;
- Tanh;
- Leaky ReLU.

Найважливішими рішеннями, які слід прийняти при побудові нейронної мережі, є такі:

- скільки шарів у нейромережі;
- скільки прихованих одиниць для кожного шару.

Нейронна мережа з великою кількістю шарів і прихованих одиниць може навчитися складному представленню даних, але це робить обчислення в мережі дуже дорогим.

**Функція втрат**

Після визначення прихованих шарів і функції активації, нам треба вказати функцію втрат і оптимізатор.

Для двійкової класифікації прийнято використовувати функцію втрати бінарної перехресної ентропії. У лінійній регресії використовуємо середню квадратичну помилку.

Функція втрат є важливим показником для оцінювання ефективності оптимізатора. Під час навчання цей показник мінімізується. Треба ретельно вибирати показник якості залежно від типу проблеми, яку необхідно розв'язувати.

### Оптимізатор

Функція втрат – це показник ефективності моделі. Оптимізатор допомагає покращити вагові коефіцієнти мережі, щоб зменшити втрати. Є різні оптимізатори, але найпоширеніший – це стохастичний градієнтний спуск.

Поширеними оптимізаторами є:

- оптимізація імпульсу;
- прискорений градієнт Нестерова;
- Ada Grad;
- оптимізація Адама.

## Обмеження нейронної мережі

### Перенавчання

Поширена проблема складної нейронної мережі – труднощі узагальнення невидимих даних. Нейронна мережа з великою кількістю вагових коефіцієнтів може дуже добре визначити конкретні деталі для тренувального набору, але це часто призводить до перенавчання. Якщо дані будуть незрівноваженими в межах груп (тобто недостатньо доступних даних у деяких групах), мережа буде дуже добре вчитися під час тренінгу, але не матиме можливості узагальнити такий шаблон до раніше небачених даних.

Існує компроміс у машинному навчанні між оптимізацією і узагальненням.

Оптимізація моделі вимагає пошуку найкращих параметрів, що мінімізують втрати навчального набору.

Узагальнення свідчить про те, як модель поводиться з небаченими даними.

Щоб модель не захоплювала конкретні деталі або небажані зразки даних тренувань, можемо використати різні методи. Найкращий метод – це наявність збалансованого набору даних із достатньою кількістю даних. Мистецтво зменшення перенавчання називається **регуляризація**. Розглянемо деякі традиційні методи.

### Розмір мережі

Нейронна мережа із занадто великою кількістю шарів і прихованих одиниць, як відомо, є надзвичайно складною. Безпосередній спосіб зменшити складність моделі – зменшити її розмір. Немає найкращої практики для визначення кількість шарів. Почати треба з невеликої кількості шарів і збільшувати розмір мережі, поки не визначимо прийнятну модель.

### Регуляризація вагових коефіцієнтів

Стандартна методика запобігання перенавчанню – додавання обмежень на вагові коефіцієнти мережі. Обмеження змушує розмір мережі приймати лише невеликі значення. Обмеження додається до функції втрат помилки. Відомі два види регуляризації:

- L1: Lasso: відбирає найбільш важливі фактори, які найсильніше впливають на результат;
- L2: Ridge: запобігає перенавчанню моделі шляхом заборони на непропорційно великі вагові коефіцієнти.

### Випадання

Випадання – це дивна, але корисна техніка. Мережа з відсівом означає, що деякі вагові коефіцієнти будуть випадковим чином визначені такими, що дорівнюють нулю. Уявіть, що є масив вагових коефіцієнтів [0.1, 1.7, 0.,7, -0.9]. Якщо нейронна мережа має

випадання, вони стануть такими [0.1, 0, 0, -0.9], з випадковим чином розподіленим нулем. Параметром, який керує випаданням, є швидкість випадання. Коефіцієнт визначає, скільки вагових коефіцієнтів треба встановити в нулі. Частота між 0,2 і 0,5 є звичайною.

## Приклад нейронної мережі в TensorFlow

Розглянемо в дії, як працює нейронна мережа для типової проблеми класифікації. Є два входи (**x1** і **x2)** з випадковим значенням. Вихід – це двійковий клас. Мета – класифікація мітки на основі двох ознак. Для виконання цього завдання архітектура нейронної мережі визначається таким чином.

- Два прихованих шари:
    - перший шар має чотири повністю пов'язаних нейрони;
    - другий шар має два повністю пов'язаних нейрони.
- Функцією активації є ReLU.
- Додаємо регуляризацію L2 зі швидкістю навчання 0,003.

Мережа оптимізує вагові коефіцієнти протягом 180 епох з розміром партії 10.

Передусім мережа призначає випадкові значення всім ваговим коефіцієнтам.

- При випадкових вагових коефіцієнтах, тобто без оптимізації, втрати на виході становлять 0,453. На рис. 11.4 наведено мережу в різних кольорах.
- Помаранчевий колір засвідчує негативні значення, а сині кольори вказують на позитивні значення.
- Точки даних мають таке ж представлення: сині – позитивні мітки, а помаранчеві – негативні.

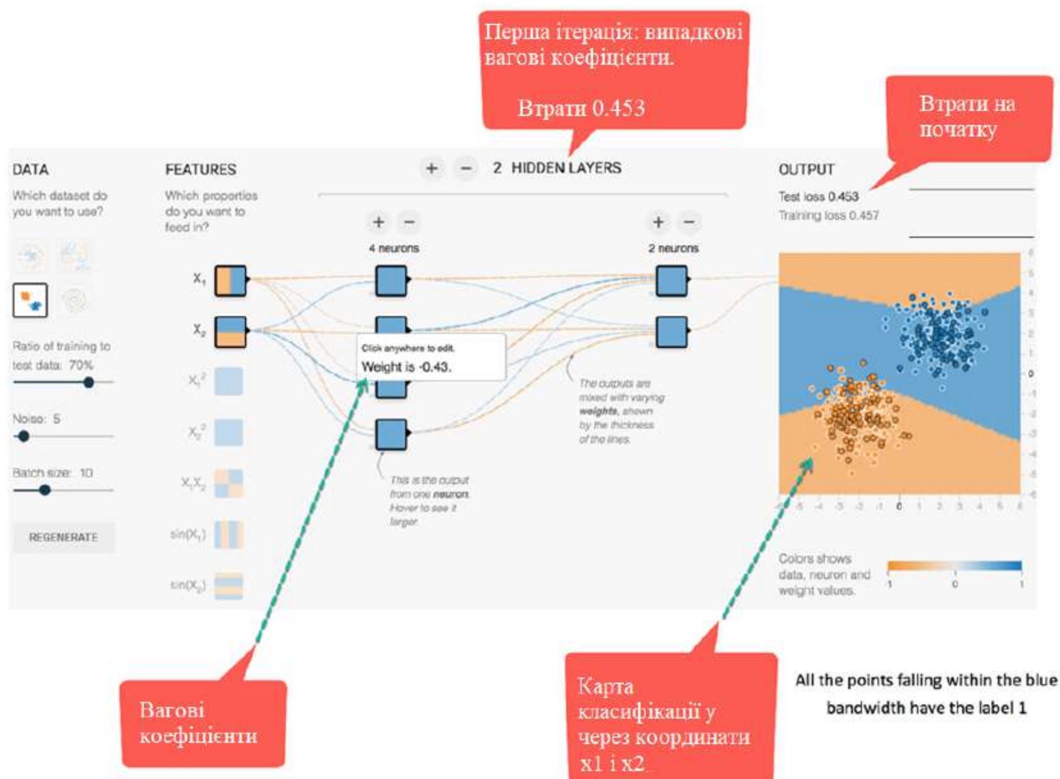



Рис. 11.4. Нейронна мережа на початку тренування

Всередині другого прихованого шару кольорові лінії відповідають знакам вагових коефіцієнтів. Помаранчеві лінії призначають негативним ваговим коефіцієнтам, а сині – позитивним.

Отже, у відображенні на виході мережа робить досить багато помилок. Подивимося, як поводиться мережа після оптимізації.

На рис. 11.5 наведено результати оптимізованої мережі. Передусім помітно, що мережа успішно навчилася класифікувати точку даних, оскільки раніше (рис. 11.4), початкові вагові коефіцієнти були -0,43, а після оптимізації вагові коефіцієнти стали -0,95.

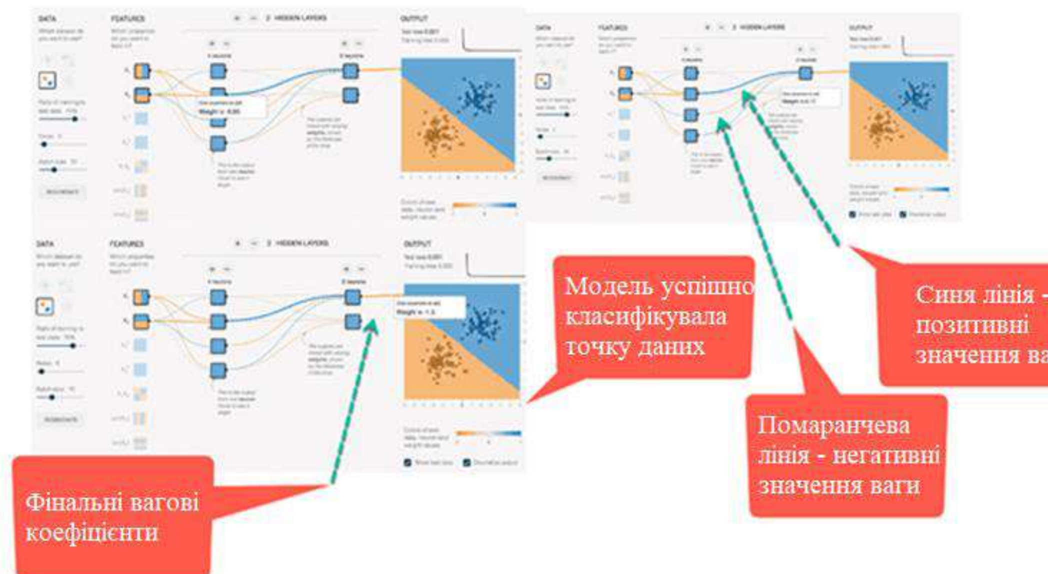



Рис. 11.5. Результати тренування мережі

Ідея може бути реалізована для мережі з більшою кількістю прихованих шарів і нейронів, що можна перевірити за посиланням[26] [26].

## Тренування нейронної мережі з TensorFlow

Розглянемо можливість тренування нейронної мережі за допомогою TF, використовуючи оцінювач API DNNClassifier.

Будемо використовувати набір даних MNIST для навчання першої нейронної мережі. Навчити нейронну мережу за допомогою TF не дуже складно. Крок попереднього опрацювання виглядає так само, як і в попередніх навчальних розділах. Послідовність наших дій така:

- Крок 1: імпорт даних;
- Крок 2: перетворення даних;
- Крок 3: конструювання тензора;
- Крок 4: побудова моделі;
- Крок 5: тренування і оцінювання моделі;
- Крок 6: вдосконалення моделі.

**Крок 1. Імпорт даних**

Передусім нам треба імпортувати необхідні бібліотеки. Можна імпортувати набір даних MNIST за допомогою scikit learn.

Набір даних MNIST – це часто використовуваний набір даних для тестування нових методів або алгоритмів. Вказаний набір даних – це набір зображень розміром 28 × 28 пікселів із рукописною цифрою від 0 до 9. Зараз найнижча помилка в тесті становить 0,27% для комітету із семи конволюційних нейронних мереж.

```
import numpy as np
import tensorflow as tf
np.random.seed(1337)
```

[26] http://playground.tensorflow.org/

Можна тимчасово завантажити дані scikit learn за вказаною адресою. Скопіюємо і вставимо набір даних у зручну папку. Щоб імпортувати дані в python, можна використати `fetch_mldata` з scikit learn. Вставимо шлях до файлу всередині `fetch_mldata`, щоб отримати дані.

```
from sklearn.datasets
import fetch_mldata
mnist = fetch_mldata('
/Users/Thomas/Dropbox/Learning/Upwork/tuto_TF/data/mldata/MNIST
original')
print(mnist.data.shape)
print(mnist.target.shape)
```

Після цього імпортуємо дані і отримуємо форми для обох наборів даних:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(mnist.data,
mnist.target, test_size=0.2, random_state=42)
y_train  = y_train.astype(int)
y_test  = y_test.astype(int)
batch_size =len(X_train)

print(X_train.shape, y_train.shape,y_test.shape )
```

**Крок 2. Перетворення даних**

У попередньому розділі ми дізналися, що треба перетворити дані з метою обмеження впливу чужинців. У цьому розділі перетворимо дані, використовуючи масштабувач min – max. Формула така:

```
(X-min_x)/(max_x - min_x)
```

Scikit learns вже має функцію для цього: `MinMaxScaler()`

```
## Масштабування
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
# Тренування
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
# Тестування
X_test_scaled = scaler.fit_transform(X_test.astype(np.float64))
```

**Крок 3. Конструювання тензора**

Отже, нам вже відомий спосіб створення тензора у TF. Можна перетворити набір для тренування в числовий стовпчик.

```
feature_columns = [tf.feature_column.numeric_column('x',
shape=X_train_scaled.shape[1:])]
```

**Крок 4. Побудова моделі**

Архітектура нейронної мережі містить два приховані шари з 300 одиницями для першого шару і 100 одиницями для другого. Використовуємо ці значення на основі власного досвіду. Можна налаштувати вказані значення і побачити, як це впливає на точність мережі.

Щоб побудувати модель, використовуємо оцінювач DNNClassifier. Можемо додати кількість шарів до аргументів `element_column`. Треба встановити кількість класів на 10, оскільки в навчальному наборі є десять класів. Ми вже ознайомлені із синтаксисом об'єкта

оцінювача. Аргументи, що містять стовпці (кількість класів і `model_dir`), точно такі ж, як у попередньому навчальному розділі. Новий аргумент `hidden_unit` контролює кількість шарів і кількість вузлів для підключення до нейронної мережі. У нижченаведеному коді є два прихованих шари, перший з яких з'єднує 300 вузлів, а другий з'єднаний зі 100 вузлами.

Щоб створити оцінювач, використаємо `tf.estimator.DNNClassifier` з такими параметрами:

- feature_column: визначаємо стовпці для використання в мережі;
- hidden_units: визначаємо кількість прихованих нейронів;
- n_classes: визначаємо кількість класів для прогнозування;
- model_dir: визначаємо шлях TensorBoard.

```
estimator = tf.estimator.DNNClassifier(
    feature_columns=feature_columns,
    hidden_units=[300, 100],
    n_classes=10,
    model_dir = '/train/DNN')
```

**Крок 5. Тренування і оцінювання моделі**

Для тренування і оцінювання моделі можемо використати метод numpy

```
# Тренування оцінювача
train_input = tf.estimator.inputs.numpy_input_fn(
    x={"x": X_train_scaled},
    y=y_train,
    batch_size=50,
    shuffle=False,
    num_epochs=None)
estimator.train(input_fn = train_input,steps=1000)
eval_input = tf.estimator.inputs.numpy_input_fn(
    x={"x": X_test_scaled},
    y=y_test,
    shuffle=False,
    batch_size=X_test_scaled.shape[0],
    num_epochs=1)
estimator.evaluate(eval_input,steps=None)
```

Результат на виході:

```
{'accuracy': 0.9637143,
 'average_loss': 0.12014342,
 'loss': 1682.0079,
 'global_step': 1000}
```

Використана архітектура приводить до точності набору для оцінювання 96%.

**Крок 6. Вдосконалення моделі**

Можна спробувати вдосконалити модель, додавши параметри регуляризації.

Використаємо оптимізатор Адама зі швидкістю випадання 0,3, L1 X і L2 y. У TF можна керувати оптимізатором за допомогою об'єкта тренування, слідуючи за іменем оптимізатора. TF – це вбудований API для оптимізатора Proximal AdaGrad.

Щоб додати регуляризацію до глибокої нейронної мережі, можемо використати `tf.train.ProximalAdagradOptimizer` із такими параметрами:

- швидкість навчання: `learning_rate`;
- L1 регуляризація: `l1_regularization_strength`;
- L2 регуляризація: `l2_regularization_strength`.

```
estimator_imp = tf.estimator.DNNClassifier(
    feature_columns=feature_columns,
    hidden_units=[300, 100],
    dropout=0.3,
    n_classes = 10,
    optimizer=tf.train.ProximalAdagradOptimizer(
      learning_rate=0.01,
      l1_regularization_strength=0.01,
      l2_regularization_strength=0.01
    ),
    model_dir = '/train/DNN1')
estimator_imp.train(input_fn = train_input,steps=1000)
estimator_imp.evaluate(eval_input,steps=None)
```

**Результат на виході:**

```
{'accuracy': 0.95057142,
 'average_loss': 0.17318928,
 'loss': 2424.6499,
 'global_step': 2000}
```

Значення, вибрані для зменшення перенавчання, не підвищили точність моделі. Наша перша модель мала точність 96%, а модель з регулятором L2 має точність 95%. Можемо спробувати різні значення і з’ясувати вплив на точність.

**Висновки**

У цьому розділі ми дізнались, що для того щоби побудувати нейронну мережу необхідно вказати:

- кількість прихованих шарів;
- кількість повністю підключених вузлів;
- функцію активації;
- оптимізатор;
- кількість класів.

У TF можемо навчити нейронну мережу класифікації за допомогою:

- tf.estimator.DNNClassifier.

Оцінювач потребує вказати:

- feature_columns=feature_columns;
- hidden_units=[300, 100];
- n_classes=10;
- model_dir.

Можна вдосконалити модель, використовуючи різні оптимізатори. Ми дізналися, як використовувати оптимізатор Adam Grad із заданою швидкістю навчання, а також додали елемент керування для запобігання перенавчанню.

# 12. Конволюційна нейронна мережа: тензорна класифікація зображень

Конволюційна нейронна мережа, також відома як конвнет або CNN (Convolutional Neural Network), є добре відомим методом у програмах комп’ютерного зору. Цей тип архітектури є домінуючим для розпізнавання об’єктів на зображеннях або відео.

Розглянемо побудову конвнету й використання TF для рішення рукописного набору даних[27] [27].

## Архітектура конволюційної нейронної мережі

Згадайте, який був Facebook донедавна. Після того як ви завантажували зображення у свій профіль, вас просили додати ім’я до обличчя на малюнку вручну. Сьогодні ж Facebook використовує конвнет, щоб автоматично позначити вашого друга на зображенні.

Конволюційну нейронну мережу не дуже важко зрозуміти. Вхідне зображення опрацьовується під час фази згортання, а пізніше присвоюється мітці.

Типова архітектура конвнету викладена на рис. 12.1. Передусім зображення передається в мережу – це вхідне зображення. Потім вхідне зображення проходить нескінченну кількість кроків – це згорткова частина мережі. Нарешті, нейронна мережа може передбачити цифру на зображенні.

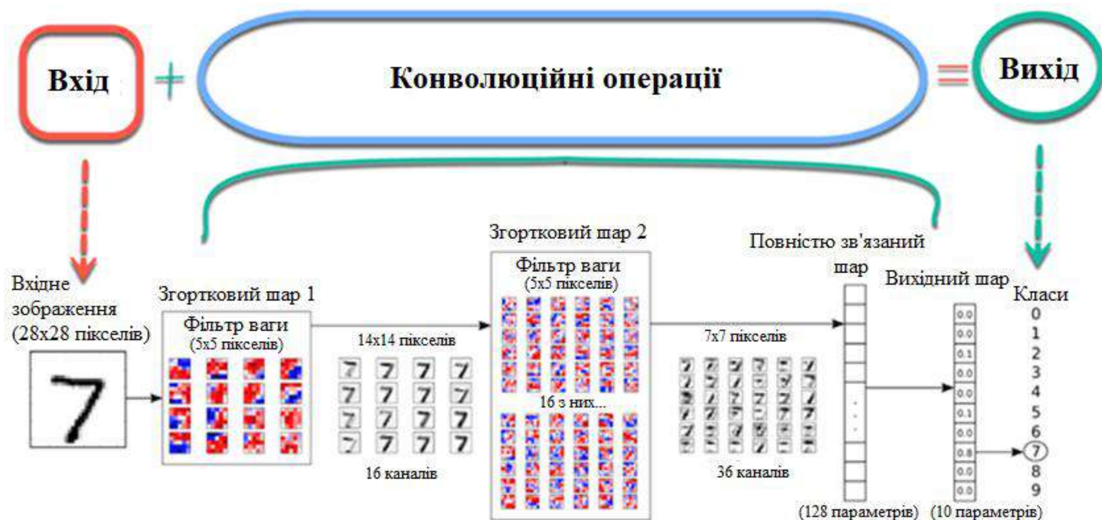



Рис. 12.1. Типова архітектура конвнету

Зображення складається із масиву пікселів, які мають висоту й ширину. Зображення у градаціях сірого має лише один канал, а кольорове зображення – три канали (окремий для червоного, зеленого і синього). Канал укладається один над одним. У цьому розділі будемо використовувати зображення в градаціях сірого лише з одним каналом. Кожен піксель має значення від 0 до 255 для відображення інтенсивності кольору. Наприклад, піксель, який дорівнює 0, покаже білий колір, а піксель зі значенням, близьким до 255, буде чорним.

Отже, подивимось на зображення, які зберігаються в наборі даних MNIST[28] [28].

---

[27] https://www.guru99.com/convnet-tensorflow-image-classification.html

[28] http://yann.lecun.com/exdb/mnist/

Рис. 12.2. Набір рукописних цифр MNIST

На рис. 12.3 показано, як відобразити зображення ліворуч в матричному форматі. Зауважимо, що оригінальна матриця стандартизована на рівні від 0 до 1. Для більш темного кольору значення в матриці становить приблизно 0,9, а білі пікселі мають значення 0.

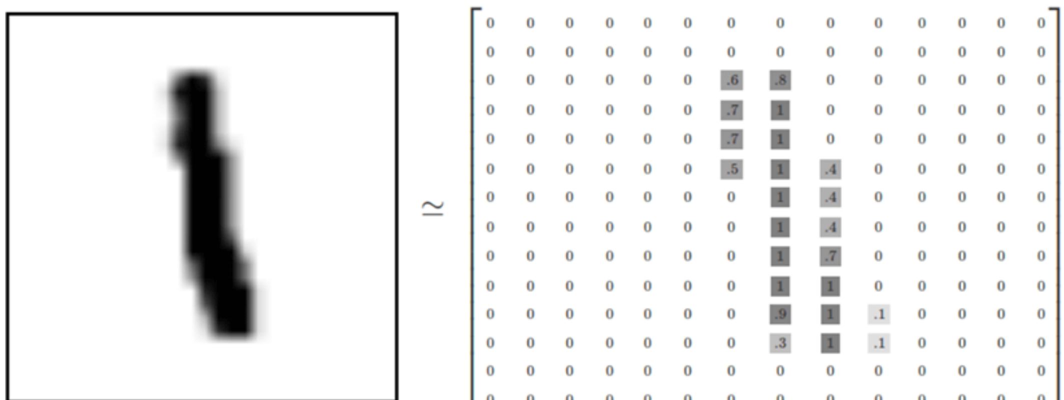


Рис. 12.3. Зображення в матричному форматі

**Згорткова операція**

Найбільш критичною складовою в моделі є згортковий шар. Ця частина спрямована на зменшення розміру зображення для більш швидкого обчислення вагових коефіцієнтів і вдосконалення його узагальнення.

Під час згорткової частини мережа зберігає основні риси зображення і виключає невідповідні шуми. Наприклад, модель вивчає, як розпізнати слона на картині з горою на задньому плані. Якщо використаємо традиційну нейронну мережу, модель присвоїть вагові коефіцієнти всім пікселям, у т. ч. і горі, що не є суттєвим й може ввести в оману мережу.

Натомість конволюційна нейронна мережа використовуватиме математичну техніку для вилучення лише найвідповідніших пікселів. Ця математична операція називається згорткою. Така методика дає змогу мережі вивчати все більш складні функції на кожному шарі. Згортання ділить матрицю на невеликі шматочки, щоб навчитися найважливішим елементам у кожному шматочку.

## Компоненти конвнетів

Є чотири компоненти конвнетів:

1. Згортання.
2. Нелінійність (ReLU).
3. Об’єднання або макс-пул операція.
4. Класифікація (повністю пов’язаний шар).

**Згортання**

Мета згортання – витягнути локально риси об’єкта на зображенні. Це означає, що мережа вивчить конкретні шаблони всередині малюнка і зможе розпізнати їх скрізь на малюнку.

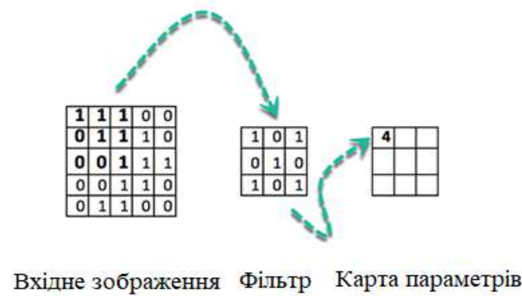



Рис. 12.4. Згортка (конволюція)

Конволюція – це множення елемента зображення на деякий фільтр. Поняття легко зрозуміти. Комп’ютер сканує частину зображення (як правило, розміром 3 × 3) і кожен елемент такої частини множиться на відповідне значення комірки фільтра. Потім отримані добутки додаються і отримане значення записується в один елемент (рис. 12.4). Результат називають картою функцій. Цей крок повторюється, допоки не буде скановане все зображення. Зауважимо, що після згортання розмір зображення зменшується.

На рис. 12.5 наведено деякі із доступних численних каналів[29] [29]. Бачимо, що кожен фільтр має певне призначення. Зверніть увагу, що ядро (Kernel) є синонімом фільтра.

| Operation | Kernel | Image result |
|---|---|---|
| Identity | $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ | |
| Edge detection | $\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$ | |
| | $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ | |
| | $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ | |
| Sharpen | $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ | |

Рис. 12.5. Канали для згортки

[29] https://en.wikipedia.org/wiki/Kernel_(image_processing)

**Арифметика за згорткою**

Фаза згортки застосовує фільтр до невеликого масиву пікселів всередині зображення. Фільтр буде рухатися по вхідному зображенню загальною формою 3 × 3 або 5 × 5. Це означає, що мережа пересуне ці вікна по вхідному зображенню і обчислить згортання. На зображенні[30] [30] (рис.12.6) ще раз показано, як працює згортка. Розмір вікна становить 3 × 3, а вихідна матриця є результатом операції перемножування матриці зображення й фільтра.

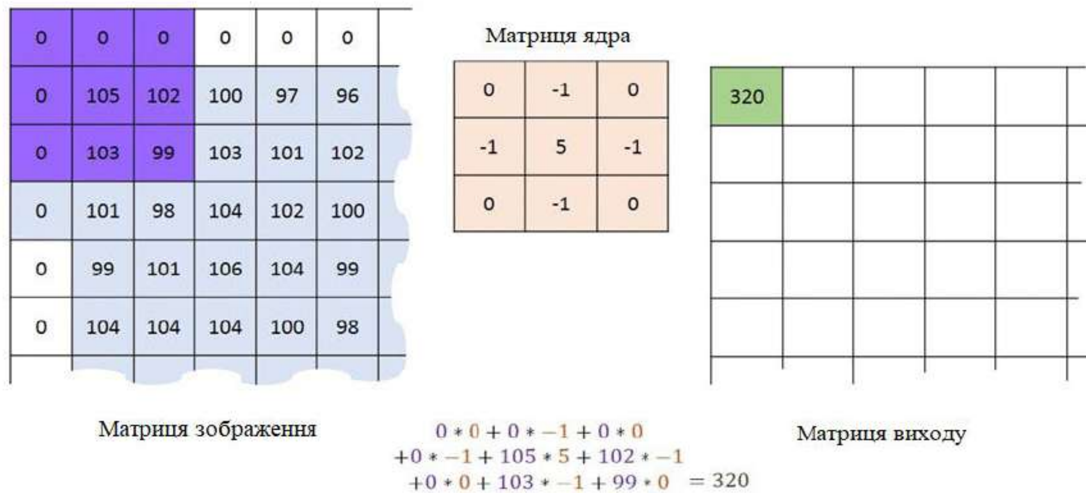



Рис. 12.6. Фаза згортки

Зауважимо, що ширина і висота на виході можуть відрізнятися від ширини і висоти входу. Це відбувається через ефект межі.

**Ефект межі**

Зображення має карту параметрів розміром 5 × 5 і фільтр розміром 3 × 3. У центрі є лише одне вікно, де фільтр може відображати сітку 3 × 3 (рис. 12.7). Карта функції на виході зменшиться на дві клітини разом з розміром 3 × 3.

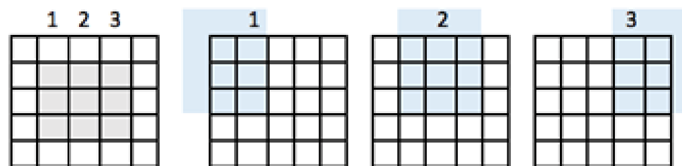



Рис. 12.7. Ефект межі

Щоб отримати такий же вихідний розмір, що і на вході, треба додати підкладку. Підкладка складається з додавання потрібної кількості рядків і стовпців на кожній стороні матриці. Це допоможе згортці по центру помістити кожну вхідну клітину. На рис. 12.8 матриця вводу / виводу має однаковий розмір 5 × 5.

Коли визначаємо мережу, то згорткові функції керуються трьома параметрами:

1. Глибина (Depth): визначає кількість фільтрів, які слід застосувати під час згортання. У попередньому прикладі ми бачили глибину 1, тобто використовувався лише один фільтр. У більшості випадків використовується більше одного фільтра. На рис. 12.9 наведено операції, виконані в ситуації з трьома фільтрами.

[30] http://machinelearninguru.com/computer_vision/basics/convolution/convolution_layer.html

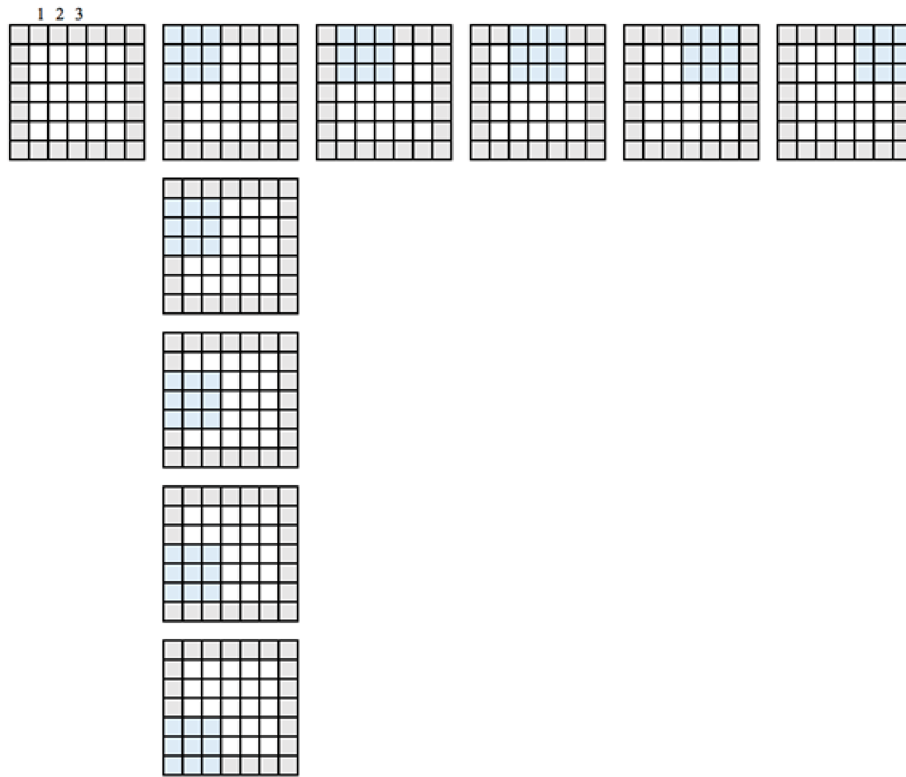



Рис. 12.8. Однаковий розмір матриці вводу/виводу

2. Крок (Stride): він визначає кількість «стрибків пікселя» між двома фрагментами. Якщо крок дорівнює 1, вікна будуть рухатися з розширенням пікселя на одиницю. Якщо крок дорівнює двом, вікна будуть стрибати на 2 пікселі. Якщо збільшимо крок, то отримаємо менші карти функцій.

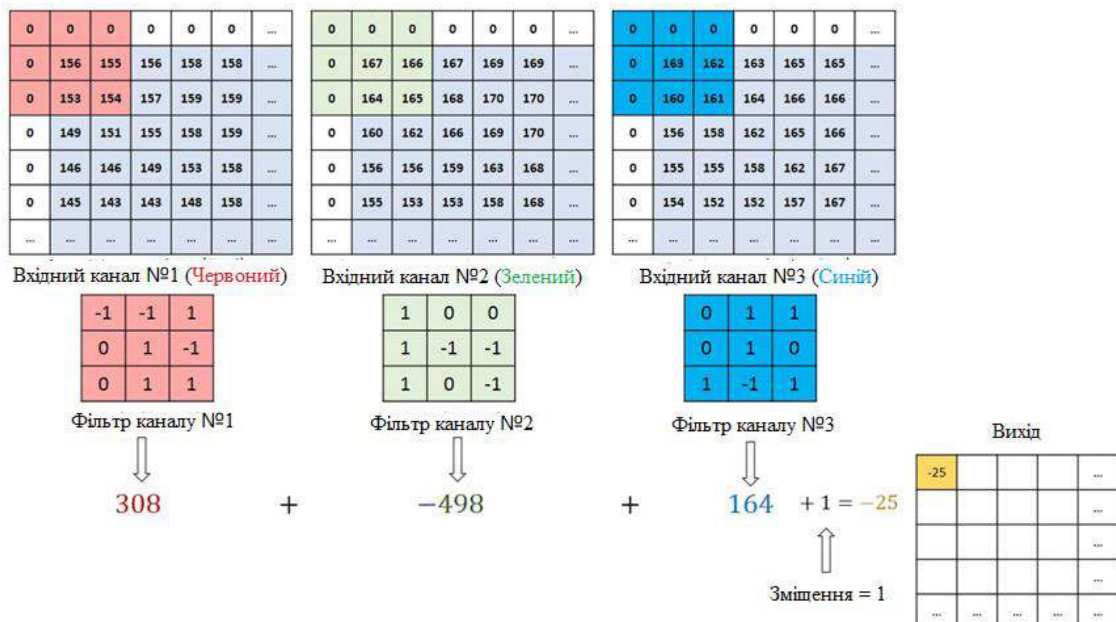



Рис. 12.9. Ситуація з трьома фільтрами

Приклад кроку 1 (рис.12.10).

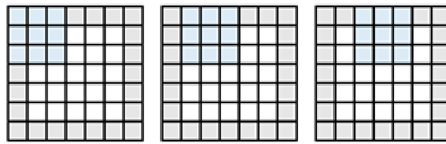


Рис. 12.10. Стрибок вікна через 1 піксель

Приклад з кроком 2 (рис. 12.11).

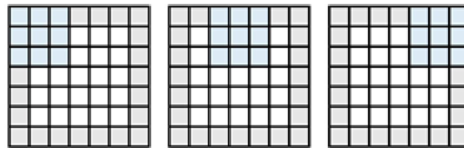


Рис. 12.11. Стрибок вікна через 2 пікселі

3. Нульова підкладка (Zero-padding): Підкладка – це операція додавання відповідної кількості рядків і стовпців з кожної сторони вхідних карт. У цьому випадку вихідний фрагмент має той же розмір, що і вхідний.

**Нелінійність (ReLU)**

В кінці операції згортання вихідний сигнал перетворюється функцією активації, щоб дозволити нелінійність. Звичайною функцією активації для конвнет є ReLU. Всі пікселі з негативним значенням будуть замінені на нуль.

**Макс-пул-операція**

Цей крок легко зрозуміти. Мета об'єднання – зменшити розмірність вхідного зображення. Кроки здійснюються для зменшення обчислювальної складності операції. Зменшуючи розмірність, мережа має менші вагові коефіцієнти для обчислення, тож вона запобігає перенавчанню.

На цьому етапі потрібно визначити розмір і крок. Стандартний спосіб об'єднати вхідне зображення – використати максимальне значення карти функцій. Розглянемо рис. 12.12. «Об'єднання» відображає чотири підматриці карти функцій 4 × 4 і повертає максимальне значення. Пул набуває максимального значення масиву 2 × 2, а потім переміщує це вікно на два пікселі. Наприклад, перша підматриця дорівнює [3,1,3,2], об'єднання поверне максимум, який дорівнює 3.

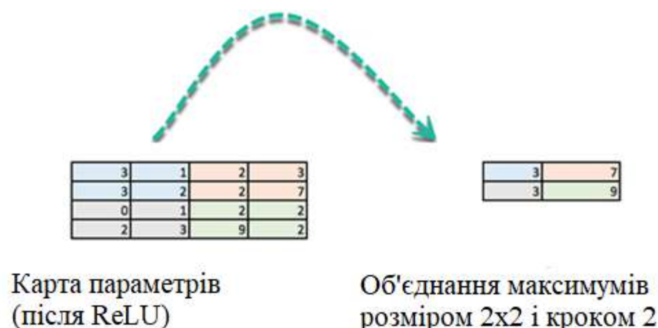



Рис. 12.12. Об'єднання через визначення максимума

Відома ще одна така операція об'єднання, як середнє.

Ця операція агресивно зменшує розмір карти об'єктів.

**Повністю пов’язані шари**

Останній крок полягає у створенні традиційної штучної нейронної мережі, як це робили у попередньому розділі. Доєднаємо всі нейрони з попереднього шару до наступного шару. Використовуємо функцію активації softmax для класифікації номера на вхідному зображенні.

**Висновки**

Конволюційна нейронна мережа збирає різні шари, перш ніж робити прогноз.
Нейронна мережа має:

- згортковий шар;
- функцію активації ReLU;
- об’єднаний шар;
- повністю пов’язаний шар.

Згорткові шари застосовують різні фільтри для субрегіонів малюнка. Функція активації ReLU додає нелінійність, а шари об’єднання зменшують розмірність карт функцій.

Усі ці шари витягують із зображення важливу інформацію, а отже, карта функцій подається на первинний повністю пов’язаний шар з функцією softmax для прогнозування.

## Тренування CNN з TensorFlow

Після ознайомлення з будівельним блоком конвентів можна побудувати його за допомогою TF. Використаємо набір даних MNIST для класифікації зображень.

Підготовка даних така ж, як і в попередньому розділі. Можемо запустити коди і перейти безпосередньо до архітектури CNN.

Виконаємо такі дії:
Крок 1: завантаження набору даних.
Крок 2: вхідний шар.
Крок 3: згортання шару.
Крок 4: об’єднання шару.
Крок 5: другий згортковий шар і шар об’єднання.
Крок 6: повністю пов’язаний шар.
Крок 7: шар Logit.

**Крок 1. Завантаження набору даних**

Набір даних MNIST можна завантажити з fetch_mldata('MNIST original').

***Створення наборів тренування / тестування***

Необхідно розділити набір даних за допомогою train_test_split.

***Масштабування функцій***

Отже, можемо масштабувати функції з MinMaxScaler.

```
import numpy as np
import tensorflow as tf
from sklearn.datasets import fetch_mldata

#Замініть USERNAME іменем своєї машини
## Windows USER
mnist = fetch_mldata('C:\\Users\\USERNAME\\Downloads\\MNIST original')
print(mnist.data.shape)
print(mnist.target.shape)
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(mnist.data,
mnist.target, test_size=0.2, random_state=42)
y_train  = y_train.astype(int)
```

```
y_test  = y_test.astype(int)
batch_size =len(X_train)

print(X_train.shape, y_train.shape,y_test.shape )
## Масштабування
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
# Тренування
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
# Тестування
X_test_scaled = scaler.fit_transform(X_test.astype(np.float64))
feature_columns = [tf.feature_column.numeric_column('x',
shape=X_train_scaled.shape[1:])]
X_train_scaled.shape[1:]
```

**Визначення CNN**

CNN використовує фільтри на неопрацьованому пікселі зображення, щоб вивчити порівняння деталей шаблону з глобальним шаблоном традиційної нейронної мережі. Для побудови CNN нам треба визначити:

1. Згортковий шар. Застосовуємо n-кількість фільтрів до карти функцій. Після згортання нам треба використати функцію активації ReLU, щоб додати нелінійність до мережі.
2. Шар об'єднання. Наступний крок після згортання – зменшення параметра max. Метою є зменшення розмірності карти функцій для запобігання надмірного розміщення і покращення швидкості обчислення. Max-об'єднання – це звичайна методика, яка ділить карти зображень на субрегіони (зазвичай розміром 2 × 2) і зберігає лише максимальні значення.
3. Повністю пов'язані шари. Всі нейрони з попередніх шарів з'єднані з наступними шарами. CNN класифікує мітку відповідно до ознак згорткових шарів, скорочених за допомогою шару об'єднання.

**Архітектура CNN**

На рис. 12.1 наведено архітектуру CNN, яку будуємо:

- згортковий шар 1: застосовує 14 фільтрів 5 × 5 (витягуючи субрегіони 5 × 5 пікселів) з функцією активації ReLU;
- шар об'єднання: виконує max-об'єднання за допомогою фільтрів 2 × 2 й з кроком 2 (це вказує, що об'єднані регіони не перетинаються);
- згортковий шар 2: застосовує 36 фільтрів 5 × 5 з функцією активації ReLU;
- шар об'єднання: знову виконує max-об'єднання з фільтром 2 × 2 й кроком 2;
- 1764 нейрони: зі швидкістю регуляризації випадання 0,4 (ймовірність 0,4, що будь-який елемент може бути скинуто під час тренування);
- повністю пов'язаний шар (шар Logits): 10 нейронів – по одному для кожного розрядного цільового класу (0–9).

Для створення CNN використовуються три важливі модулі:

- conv2d(): будує двовимірний згортковий шар з кількістю фільтрів, розміром ядра фільтра, підкладкою і функцією активації як аргументів;
- max_pooling2d(): конструює двовимірний шар об'єднання за допомогою алгоритму max-об'єднання;
- dense(): будує повністю пов'язаний шар із прихованими шарами і блоками.

Визначимо функцію побудови CNN. Отже, детально розглянемо, як побудувати кожен будівельний блок, перш ніж згорнути все разом у функції.

**Крок 2. Вхідний шар**

```
def cnn_model_fn(features, labels, mode):
    input_layer = tf.reshape(tensor = features["x"],shape =[-1, 28, 28,
1])
```

Треба визначити тензор із формою даних. Для цього можна використати модуль `tf.reshape`. У цьому модулі треба оголосити тензор для перегляду форми, а також форму тензора. Перший аргумент – це параметри даних, які визначені в аргументі функції.

Зображення має висоту, ширину і канал. Набір даних MNIST – це монохронічне зображення розміром 28 × 28. В аргументі фігури встановлюємо розмір партії до -1 таким чином, щоб він набував форми ознак ["x"]. Перевага полягає в тому, щоб гіперпараметри розміру партії були налаштовані. Якщо розмір партії встановлено на 7, то тензор подає 5 488 значень (28*28*7).

**Крок 3. Згортковий шар**

```
# Перший конволюційний шар
conv1 = tf.layers.conv2d(
     inputs=input_layer,
     filters=14,
     kernel_size=[5, 5],
     padding="same",
     activation=tf.nn.relu)
```

Перший згортковий шар має 14 фільтрів з розміром ядра 5 × 5 і такою ж підкладкою. Одна і та ж підкладка означає, що і вихідний, і вхідний тензори повинні мати однакову висоту і ширину. TF додає нулі до рядків і стовпців, щоб забезпечити однаковий розмір.

Використовуємо функцію активації ReLU. Розмір виходу буде [28, 28, 14].

**Крок 4. Шар об'єднання**

Наступним кроком після згортання є обчислення об'єднання. Обчислення об'єднання дасть змогу зменшити розмірність даних. Можна використати модуль `max_pooling2d` розміром 2 × 2 й кроком 2. Використовуємо попередній шар як вхідний. Розмір виходу буде [batch_size, 14, 14, 14]

```
# Перший шар об'єднання
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2],
strides=2)
```

**Крок 5. Другий згортковий шар і шар об'єднання**

Другий згортковий шар має 32 фільтри, з вихідним розміром [batch_size, 14, 14, 32]. Шар об'єднання має той же розмір, що і раніше, а вихідна форма [batch_size, 14, 14, 18].

```
conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=36,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)
pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2],
strides=2)
```

**Крок 6. Повністю пов'язаний шар**

Треба визначити повністю пов'язаний шар. Карта параметрів має бути вирівняна до з'єднання з повністю пов'язаним шаром. Можна використати перестановку модуля розміром $7 \times 7 \times 36$.

Повністю пов'язаний шар з'єднає 1 764 нейрони. Додаємо функцію активації ReLU. Крім того, додаємо термін регуляції випадання зі швидкістю 0,3, тобто 30 відсотків вагових коефіцієнтів буде встановлено в 0. Зауважимо, що випадання відбувається лише під час тренувального етапу. Функція cnn_model_fn має режим аргументу, щоб оголосити, чи потрібно моделі навчатися, чи потрібно пройти оцінювання.

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 36])

dense = tf.layers.dense(inputs=pool2_flat, units=7 * 7 * 36,
activation=tf.nn.relu)
dropout = tf.layers.dropout(
      inputs=dense, rate=0.3, training=mode ==
tf.estimator.ModeKeys.TRAIN)
```

**Крок 7. Шар Logit**

Отже, можна визначити останній шар для моделі прогнозування. Форма виводу дорівнює розміру партії й 10, загальній кількості зображень цифр.

```
# Шар Logits
logits = tf.layers.dense(inputs=dropout, units=10)
```

Можна створити словник, що містить класи, а також ймовірність кожного класу. Модуль `tf.argmax()` повертає найбільше значення, якщо шар logit. Функція softmax повертає ймовірність кожного класу.

```
predictions = {
    # Генерування передбачення
    "classes": tf.argmax(input=logits, axis=1),
    "probabilities": tf.nn.softmax(logits, name="softmax_tensor")  }
```

Треба повернути передбачення словника лише тоді, коли режим встановлений на передбачення. Додаємо код для відображення передбачення:

```
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode,
predictions=predictions)
```

Наступний крок полягає в обчисленні втрат моделі. З останнього розділу ми дізналися, що функцією втрат для багатокласової моделі є перехресна ентропія. Втрати легко обчислити за допомогою такого коду:

```
# Розрахунок втрат (для обох моделей TRAIN і EVAL)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
logits=logits)
```

Останнім кроком є оптимізація моделі, тобто пошук найкращих значень вагових коефіцієнтів. Для цього використовуємо оптимізатор градієнтного спуску зі швидкістю навчання 0,001. Мета – мінімізувати втрати.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
```

Отже, після побудови CNN, у разі необхідності можна відобразити показники ефективності в режимі оцінювання. Показники ефективності для багатокласової моделі – це показники точності. TF оснащений модулем точності з двома аргументами, мітками і прогнозованими значеннями:

```
eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(labels=labels,
predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
eval_metric_ops=eval_metric_ops)
```

Ми створили свою першу CNN, а також готові вкласти все у функцію, щоб використати її для навчання і оцінювання моделі:

```
def cnn_model_fn(features, labels, mode):
  """Модель функції для CNN."""
  # Вхідний шар
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  # Шар згортки
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Шар об'єднання
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2],
strides=2)

  # Шар згортки #2 і шар об'єднання
  conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=36,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2],
strides=2)

  # Щільно пов'язаний шар
  pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 36])
  dense = tf.layers.dense(inputs=pool2_flat, units=7 * 7 * 36,
activation=tf.nn.relu)
  dropout = tf.layers.dropout(
      inputs=dense, rate=0.4, training=mode ==
tf.estimator.ModeKeys.TRAIN)

  # Шар Logits
  logits = tf.layers.dense(inputs=dropout, units=10)

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
```

```
  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode,
predictions=predictions)

  # Розрахунок втрат
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
logits=logits)

  # Налаштування конфігурації тренування (для режиму TRAIN)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
train_op=train_op)

  # Додавання показників оцінювання в режим Evaluation
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

Нижченаведені кроки такі ж, як і в попередньому розділі.

Передусім визначаємо оцінювач з моделлю CNN:

```
# Створення оцінювача
mnist_classifier = tf.estimator.Estimator(
    model_fn=cnn_model_fn, model_dir="train/mnist_convnet_model")
```

CNN виконується багато разів для тренування, тож створюємо журнал реєстрації, щоб зберігати значення шарів softmax кожні 50 ітерацій:

```
# Налаштування журналу для прогнозів
tensors_to_log = {"probabilities": "softmax_tensor"}
logging_hook = tf.train.LoggingTensorHook(tensors=tensors_to_log,
every_n_iter=50)
```

Отже, тепер ми готові оцінити модель. Встановлюємо розмір партії 100 і переміщуємо дані. Зауважимо, що кількість кроків тренування вибрана 16 000, тож тренування потребує багато часу. Будьте терплячими!

```
# Тренування моделі
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": X_train_scaled},
    y=y_train,
    batch_size=100,
    num_epochs=None,
    shuffle=True)
mnist_classifier.train(
    input_fn=train_input_fn,
    steps=16000,
    hooks=[logging_hook])
```

Тепер, коли модель натренована, можемо оцінити її і вивести результат:

```
# Оцінювання моделі та виведення результату
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": X_test_scaled},
    y=y_test,
    num_epochs=1,
    shuffle=False)
eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)

INFO:TensorFlow:Calling model_fn.
INFO:TensorFlow:Done calling model_fn.
INFO:TensorFlow:Starting evaluation at 2018-08-05-12:52:41
INFO:TensorFlow:Graph was finalized.
INFO:TensorFlow:Restoring parameters from
train/mnist_convnet_model/model.ckpt-15652
INFO:TensorFlow:Running local_init_op.
INFO:TensorFlow:Done running local_init_op.
INFO:TensorFlow:Finished evaluation at 2018-08-05-12:52:56
INFO:TensorFlow:Saving dict for global step 15652: accuracy = 0.9589286,
global_step = 15652, loss = 0.13894269
{'accuracy': 0.9689286, 'loss': 0.13894269, 'global_step': 15652}
```

З використаною архітектурою отримано точність 97%. Можемо змінити архітектуру, розмір партії і кількість ітерацій, щоб підвищити точність. Нейронна мережа CNN працює набагато краще, ніж ANN або логістична регресія. У розділі про штучну нейронну мережу отримано точність 96%, що нижче за точність CNN. Характеристики CNN вражають більшим набором зображень як з точки зору швидкості, так і точності обчислення.

**Висновки**

Конволюційна нейронна мережа дуже добре працює для оцінювання зображення. Цей тип архітектури є домінуючим для розпізнавання об’єктів на зображенні або відео.

Для побудови CNN нам треба виконати шість кроків.

**Крок 1**. Вхідний шар

Цей крок опрацьовує дані. Форма дорівнює кореню квадратному кількості пікселів. Наприклад, якщо зображення має 156 пікселів, то форма – 26 × 26. Потрібно вказати, чи має малюнок колір (якщо так, то у нас було б 3 для форми RGB, якщо ні – 1).

```
input_layer = tf.reshape(tensor = features["x"],shape =[-1, 28, 28, 1])
```

**Крок 2.** Згортковий шар

Далі потрібно створити шари згортання. Застосовуємо різні фільтри, щоб дати змогу мережі вивчити важливу функцію. Вказуємо розмір ядра і кількість фільтрів:

```
conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=14,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)
```

**Крок 3.** Шар об’єднання

На третьому кроці додаємо шар об’єднання. Цей шар зменшує розмір введення. В ньому зберігаємо лише максимальне значення підматриці. Наприклад, якщо підматриця [3,1,3,2], то об’єднання поверне максимум, який дорівнює 3.

```
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2],
strides=2)
```

**Крок 4.** Додавання згорткового шару і шару об'єднання

На цьому кроці можемо додати скільки завгодно згорткових шарів і шарів об'єднання. Google використовує архітектуру з понад 20 згортковими шарами.

**Крок 5.** Повністю пов'язаний шар

Цей крок вирівнює попередній, щоб створити повністю пов'язані шари. На цьому кроці можемо використовувати різні функції активації й додати ефект випадання:

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 36])

dense = tf.layers.dense(inputs=pool2_flat, units=7 * 7 * 36,
activation=tf.nn.relu)
dropout = tf.layers.dropout(
      inputs=dense, rate=0.3, training=mode ==
tf.estimator.ModeKeys.TRAIN)
```

**Крок 6.** Шар Logit

Завершальним кроком є передбачення:

```
logits = tf.layers.dense(inputs=dropout, units=10)
```

# 13. Автоенкодер в глибокому навчанні: приклад TensorFlow

## Що таке автоенкодер

Автоенкодер – це інструмент для відтворення введення. Наприклад, машина робить зображення і може створити тісно пов'язану картину. Вхід у цю нейронну мережу не є маркованим, тобто мережа здатна навчатися без учителя. Отже, вхід кодується мережею, щоб зосередити увагу лише на найважливішій функції. Це одна з причин, чому автоенкодер популярний для зменшення розмірності. Крім того, автоенкодери можуть бути використані для створення **генеративних моделей навчання**. Наприклад, нейронну мережу можна навчити з набором облич, а потім можна створити нові обличчя.

Мета автоенкодера – виробляти приближення введення, зосереджуючись лише на суттєвих ознаках. Можемо подумати над тим, щоб навчитися копіювати і вставляти дані для отримання результату. Фактично, автоенкодер – це набір обмежень, які змушують мережу вивчати нові способи подання даних, відмінні від простого копіювання результатів.

Типовий автоенкодер визначається введенням, внутрішнім поданням і виведенням (приближенням входу). Навчання відбувається в шарах, прикріплених до внутрішнього представлення. Є два основні блоки шарів, схожих на традиційну нейронну мережу. Незначна різниця полягає в тому, що шар, який містить вихід, має дорівнювати вхідному. На рис. 13.1 оригінальний вхід переходить у перший блок, який називається **кодером**. Це внутрішнє подання стискує (зменшує) розмір введення. У другому блоці відбувається реконструкція входу. Це фаза розшифровки (**декодер**).

Модель буде оновлювати вагові коефіцієнти, мінімізуючи функцію втрат. Модель штрафується, якщо вихід реконструкції відрізняється від вхідного.

Отже, уявіть собі зображення розміром 50 × 50 (тобто 250 пікселів) і нейронну мережу з лише одним прихованим шаром, що складається із 100 нейронів. Навчання проводиться на карті функцій, яка в два рази менша за вхідну. Це означає, що мережі потрібно знайти спосіб реконструювати 250 пікселів лише з вектором нейронів, що дорівнює 100.

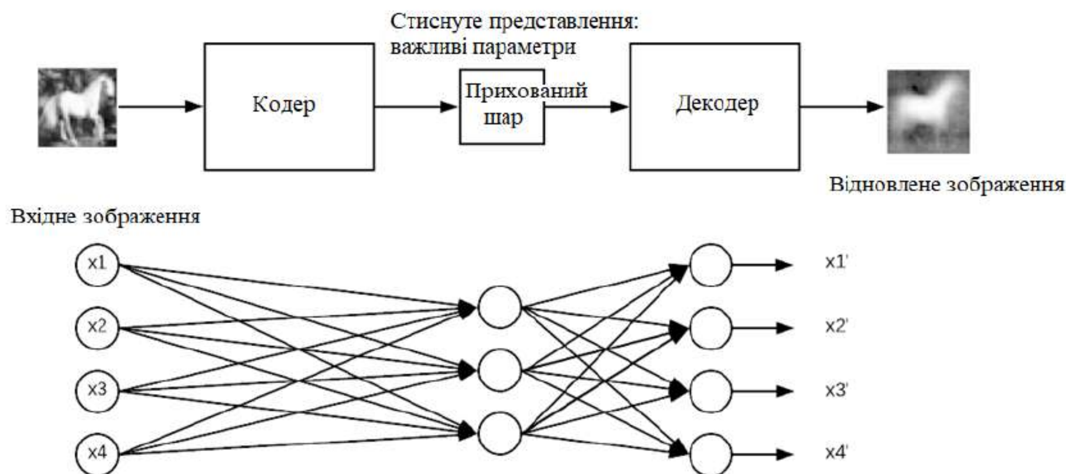



Рис. 13.1. Типовий автоенкодер

## Приклад автоенкодера

Розглянемо процес використання автоенкодера[31] [31]. Архітектура схожа на традиційну нейронну мережу. Вхід переходить до прихованого шару для стиснення або зменшення свого розміру, а потім доходить до шарів реконструкції. Мета полягає у створенні вихідного зображення максимально близького до оригіналу. Модель має навчитися способу досягнення своєї задачі з набором обмежень, тобто з меншим виміром.

Зараз для видалення шуму на зображенні здебільшого використовуються автоенкодери. Уявіть зображення з подряпинами, на якому людина все ще здатна розпізнати зміст. Ідея автоенкодера, який видаляє шум, – це додати шум на зображення, щоб змусити мережу вивчити шаблон даних.

Іншим корисним сімейством автоенкодерів є варіаційний автоенкодер. Цей тип мережі може генерувати нові зображення. Уявіть, що ми натренували мережу із зображенням людини, а після цього така мережа може створювати нові обличчя.

## Побудова автоенкодера з TensorFlow

Розглянемо процес побудови автокодера для реконструкції зображення.

Використаємо набір даних CIFAR-10[32] [32], в якому 60 000 кольорових зображень розміром 32 × 32. Набір даних вже розділено на 50 000 зображень для тренування і 10 000 для тестування.

У наборі 10 класів:

- літаки;
- автомобілі;
- птахи;

[31] https://www.guru99.com/autoencoder-deep-learning.html
[32] https://www.cs.toronto.edu/~kriz/cifar.html

- коти;
- олені;
- собаки;
- жаби;
- коні;
- кораблі;
- вантажівки.

Нам треба завантажити зображення за наведеним на попередній сторінці посиланням і розархівувати їх. Папка `for-10-batches-py` матиме 5 партій даних з 10 000 зображень в кожному у випадковому порядку.

Перш ніж будувати і тренувати свою модель, треба виконати деяке опрацювання даних. Зробимо опрацювання у такій послідовності:

1. Імпорт даних.
2. Перетворення даних в чорно-білий формат.
3. Додавання всіх партій.
4. Побудова набору для тренування.
5. Побудова візуалізатора зображень.

**Опрацювання зображення**

**Крок 1. Імпорт даних**

За даними офіційного вебсайту можна завантажити дані з нижченаведеним кодом. Код завантажить дані у словник із **даними** і **міткою**. Зауважимо, що код є функцією.

```
import numpy as np
import tensorflow as tf
import pickle
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='latin1')
    return dict
```

**Крок 2. Перетворення даних у чорно-білий формат**

Для простоти перетворимо дані у масштаб сірого. Тобто, лише один вимір проти трьох для кольорових зображень. Більшість нейромереж працює лише з входом однієї розмірності.

```
def grayscale(im):
    return im.reshape(im.shape[0], 3, 32,
32).mean(1).reshape(im.shape[0], -1)
```

**Крок 3. Додавання всіх партій**

Отже, коли створено обидві функції і завантажено набір даних, можемо написати цикл для додавання даних у пам'ять. Якщо перевірити, то розпакований файл з даними називається `data_batch_` з номерами від 1 до 5. Можна переглянути файли і додати їх до даних.

Після виконання цього кроку перетворимо дані про кольори у формат сірої шкали. Як бачимо, форма даних становить 50 000 і 1 024. Тепер 32*32 пікселі вирівнюються до 2014.

```
# Завантаження даних в пам'ять
data, labels = [], []
```

```
## Цикл над b
for i in range(1, 6):
    filename = './cifar-10-batches-py/data_batch_' + str(i)
    open_data = unpickle(filename)
    if len(data) > 0:
        data = np.vstack((data, open_data['data']))
        labels = np.hstack((labels, open_data['labels']))
    else:
        data = open_data['data']
        labels = open_data['labels']

data = grayscale(data)
x = np.matrix(data)
y = np.array(labels)
print(x.shape)
(50000, 1024)
```

***Примітка.*** Замініть `'./cifar-10-batches-py/data_batch_'` на актуальне розміщення файлів. Наприклад, для Windows шлях буде `filename = 'E:\cifar-10-batches-py\data_batch_'+ str(i)`

**Крок 4. Побудова набору для тренування**

Щоби зробити навчання більш швидким і легким, будемо тренувати модель лише на зображеннях коня. Коні – сьомий клас за даними мітки. Як зазначено в документації набору даних CIFAR-10, кожен клас містить 5 000 зображень. Можемо вивести форму даних, щоб підтвердити наявність 5 000 зображень з 1 024 стовпцями:

```
horse_i = np.where(y == 7)[0]
horse_x = x[horse_i]
print(np.shape(horse_x))
(5000, 1024)
```

**Крок 5. Побудова візуалізатора зображень**

Побудуємо функцію для виведення зображень. Нам ця функція знадобиться для того, щоб вивести відновлене зображення з автоенкодера.

Найпростішим способом виведення зображень є використання об'єкта `imshow` з бібліотеки matplotlib. Зауважимо, що нам треба перетворити форму даних з 1 024 у 32 × 32 (тобто формат зображення).

```
# Накреслити чудові фігури
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
def plot_image(image, shape=[32, 32], cmap = "Greys_r"):
    plt.imshow(image.reshape(shape), cmap=cmap,interpolation="nearest")
    plt.axis("off")
```

Функція має три аргументи:

- `image`: вхід;
- `shape`: список, розмір зображення;
- `cmap`: вибір карти кольору, за замовчуванням, сірий (grey).

Можемо спробувати побудувати перше зображення з набору даних. Маємо побачити чоловіка на коні.

```
plot_image(horse_x[1], shape=[32, 32], cmap = "Greys_r")
```

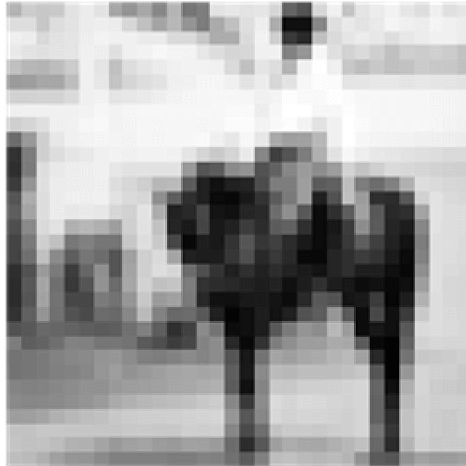


Рис. 13.2. Відновлене зображення

**Встановлення оцінювача набору даних**

Якщо набір даних вже є, можемо почати використовувати TF. Перш ніж побудувати модель, скористаємося оцінюванням набору даних TF для забезпечення мережі.

Створимо набір даних за допомогою оцінювача TF. Нагадаємо, що нам треба використати:

- `from_tensor_slices`
- `repeat`
- `batch`

Повний код для складання набору даних:

```
dataset =
tf.data.Dataset.from_tensor_slices(x).repeat().batch(batch_size)
```

Зазначимо, що **x** є placeholder з такою формою:

- [None,n_inputs]: встановимо значення None, оскільки кількість каналів зображення в мережі дорівнює розміру партії.

Для детального розгляду слід звернутися до розділу 7 «Лінійна регресія».

Після цього треба створити ітератор. Без цього рядка коду жодна інформація не пройде через конвеєр:

```
iter = dataset.make_initializable_iterator() # create the
iteratorfeatures = iter.get_next()
```

Зазначимо, що конвеєр готовий, тож можемо перевірити, чи перше зображення таке, як раніше (тобто людина на коні).

Встановимо розмір партії в 1, оскільки будемо подавати набір даних лише з одним зображенням. Можна бачити розмірність даних з `print(sess.run(features).shape)`. Вона дорівнює (1, 1024), де 1 означає, що лише одне зображення з 1 024 подається щоразу. Якщо розмір партії встановити 2, то два зображення пройдуть через конвеєр (не змінюйте

розмір партії, оскільки це призведе до помилки; одночасно перейти до функції `plot_image()` може лише одне зображення.)

```
## Параметри
n_inputs = 32 * 32
BATCH_SIZE = 1
batch_size = tf.placeholder(tf.int64)

# Використовуємо placeholder
x = tf.placeholder(tf.float32, shape=[None,n_inputs])
## Набір даних
dataset =
tf.data.Dataset.from_tensor_slices(x).repeat().batch(batch_size)
iter = dataset.make_initializable_iterator() # create the iterator
features = iter.get_next()

## Виведення зображення
with tf.Session() as sess:
    # Потік з placeholder з даними
    sess.run(iter.initializer, feed_dict={x: horse_x,
                              batch_size: BATCH_SIZE})
    print(sess.run(features).shape)
    plot_image(sess.run(features), shape=[32, 32], cmap = "Greys_r")
(1, 1024)
```

Рис. 13.3. Виведення нового зображення

**Побудова мережі**

Настав час будувати мережу, а отже, будемо тренувати складений автоенкодер, тобто мережу з декількома прихованими шарами.

Наша мережа матиме один вхідний шар із 1 024 точками, тобто формою зображення $32 \times 32$.

Блок кодера матиме один верхній прихований шар із 300 нейронами, центральний шар із 150 нейронами. Блок декодера симетричний кодеру. Отже, можемо візуалізувати мережу на рис. 13.4. Зауважимо, що можна змінювати значення прихованого і центрального шарів.

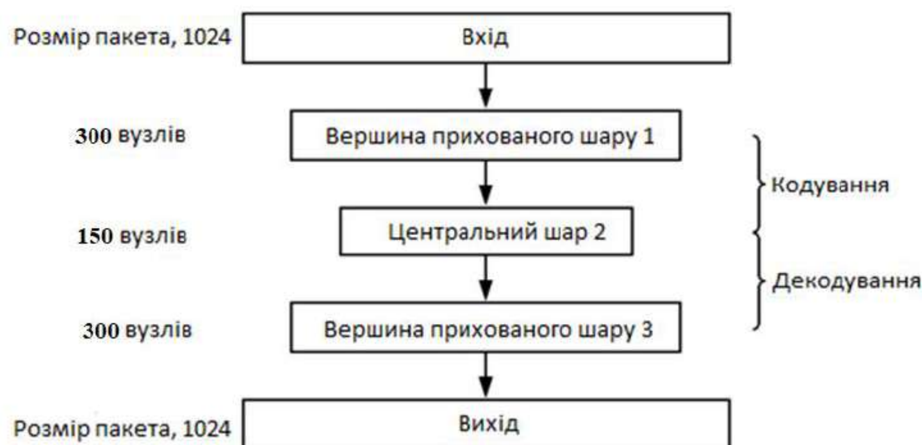



Рис. 13.4. Архітектура мережі для автоенкодера

Побудова автоенкодера дуже схожа на будь-яку іншу модель глибокого навчання. Будемо будувати модель такими кроками:

1. Визначаємо параметри.
2. Визначаємо шари.
3. Визначаємо архітектуру.
4. Визначаємо оптимізацію.
5. Запускаємо модель.
6. Оцінюємо модель.

У попередньому розділі ми дізналися, як створити конвеєр для подачі моделі, тому не треба створювати ще один набір даних. Побудуємо автокодер з чотирма шарами. Використаємо ініціалізацію Xavier. Це методика визначення початкових вагових коефіцієнтів, що дорівнюють дисперсії входу і виходу. Отже, використаємо функцію активації ELU (рис. 13.5). Функцію втрат регулюємо за допомогою регулятора L2.

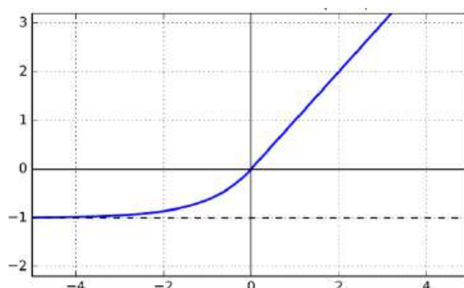



Рис. 13.5. Функція активації ELU

**Крок 1. Визначаємо параметри**

Перший крок передбачає визначення кількості нейронів у кожному шарі, швидкість навчання і гіперпараметр регуляризатора.

Перед цим частково імпортуємо функцію. Це кращий метод визначення параметрів повністю пов’язаних шарів. Нижченаведений код визначає значення архітектури автоенкодера. Як було зазначено, автоенкодер має два шари, в яких 300 нейронів у першому шарі і 150 у другому. Їх значення зберігаються в `n_hidden_1` й `n_hidden_2`.

Потрібно визначити швидкість навчання і гіперпараметр L2. Значення зберігаються в `learning_rate` та `l2_reg`

```
from functools import partial
```

```
## Кодер
n_hidden_1 = 300
n_hidden_2 = 150  # codings

## Декодер
n_hidden_3 = n_hidden_1
n_outputs = n_inputs

learning_rate = 0.01
l2_reg = 0.0001
```

Техніка ініціалізації Xavier викликається об'єктом `xavier_initializer` з оцінювача `contrib`. У цьому ж оцінювачі можемо додати регуляризатор за допомогою `l2_regularizer`:

```
## Визначаємо ініціалізацію Xavier
xav_init =  tf.contrib.layers.xavier_initializer()
## Визначаємо регуляризацію L2
l2_regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
```

**Крок 2. Визначаємо шари**

Визначено всі параметри повністю пов'язаних шарів, а отже, можемо все запакувати у змінну `dense_layer`, використовуючи об'єкт `partial. dense_layer` , активацію ELU, ініціалізацію Xavier і регуляризацію L2:

```
## Створення прихованого шару
dense_layer = partial(tf.layers.dense,
                         activation=tf.nn.elu,
                         kernel_initializer=xav_init,
                         kernel_regularizer=l2_regularizer)
```

**Крок 3. Визначаємо архітектуру**

Якщо поглянути на архітектуру (рис. 13.4), то можна побачити, що мережа має три шари з вихідним шаром. У нижченаведеному коді підключаємо відповідні шари. Наприклад, перший шар обчислює скалярний добуток між вхідними ознаками матриці і матрицями, що містять 300 вагових коефіцієнтів. Після обчислення скалярного добутку вихід переходить у функцію активації ELU. Вихід стає входом наступного шару, тому використовуємо його для обчислення `hidden_2` тощо. Множення матриць однакове для кожного шару, оскільки використовуємо ту ж саму функцію активації. Зауважимо, що останній шар (вихідний) не застосовує функції активації, оскільки це реконструйований вхід.

```
## Make the mat mul
hidden_1 = dense_layer(features, n_hidden_1)
hidden_2 = dense_layer(hidden_1, n_hidden_2)
hidden_3 = dense_layer(hidden_2, n_hidden_3)
outputs = dense_layer(hidden_3, n_outputs, activation=None)
```

**Крок 4. Визначення оптимізації**

Останній крок – побудова оптимізатора. Використовуємо середню квадратичну помилку як функцію втрат. Згадаємо з розділу про лінійну регресію, що MSE обчислюється як різниця між передбачуваним результатом і реальною міткою. Тут мітка є параметром, оскільки модель намагається відновити вхід. Отже, необхідно мати середнє значення квадрата різниці між передбачуваним результатом і входом. За допомогою TF можемо кодувати функцію втрат так:

```
loss = tf.reduce_mean(tf.square(outputs - features))
```

Тепер нам треба оптимізувати функцію втрат. Використовуємо оптимізатор Адама для обчислення градієнтів. Цільова функція – мінімізувати втрати.

```
## Оптимізація
loss = tf.reduce_mean(tf.square(outputs - features))
optimizer = tf.train.AdamOptimizer(learning_rate)
train  = optimizer.minimize(loss)
```

Пропонуємо ще одне налаштування перед навчанням моделі. Якщо будемо використовувати розмір партії 150, тобто подавати конвеєр із 150 зображеннями за кожну ітерацію, то потрібно обчислити кількість ітерацій вручну. Це робиться тривіально.

*Якщо необхідно щоразу передавати 150 зображень і відомо, що в наборі даних є 5 000 зображень, то кількість ітерацій дорівнюватиме 33. У Python можемо запустити код і переконатися, що вихід дорівнює 33:*

```
BATCH_SIZE = 150
### Число ітерацій:  розмір набору даних / розмір ітерації
n_batches = horse_x.shape[0] // BATCH_SIZE
print(n_batches)
```

```
33
```

**Крок 5. Запускаємо модель**

Отже, розглянемо останнє, але не менш важливе, тренування моделі. Тренуємо модель зі 100 епохами. Тобто модель буде бачити в 100 разів більше зображень для оптимізації вагових коефіцієнтів.

Ми вже ознайомлені з кодами для навчання моделі у TF. Незначна різниця полягає в передачі даних перед початком тренувань. Так модель тренується швидше.

Нам потрібно виводити втрати через десять епох, щоб побачити, чи навчається модель (тобто чи зменшуються втрати). Навчання триває від 2 до 5 хв, залежно від потужності комп’ютера.

```
## Встановлення параметрів
n_epochs = 100

## Викликаємо Saver для збереження моделі і повторного її використання
пізніше під час оцінки
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Ініціалізація ітератора з даними тренування
    sess.run(iter.initializer, feed_dict={x: horse_x,
                                batch_size: BATCH_SIZE})
    print('Training...')
    print(sess.run(features).shape)
    for epoch in range(n_epochs):
        for iteration in range(n_batches):
            sess.run(train)
        if epoch % 10 == 0:
            loss_train = loss.eval()    # not shown
            print("\r{}".format(epoch), "Train MSE:", loss_train)
        #saver.save(sess, "./my_model_all_layers.ckpt")
```

```
    save_path = saver.save(sess, "./model.ckpt")
    print("Model saved in path: %s" % save_path)
Training...
(150, 1024)
0 Train MSE: 2934.455
10 Train MSE: 1672.676
20 Train MSE: 1514.709
30 Train MSE: 1404.3118
40 Train MSE: 1425.058
50 Train MSE: 1479.0631
60 Train MSE: 1609.5259
70 Train MSE: 1482.3223
80 Train MSE: 1445.7035
90 Train MSE: 1453.8597
Model saved in path: ./model.ckpt
```

**Крок 6. Оцінювання моделі**

Оскільки наша модель вже натренована, то настав час її оцінити. Нам треба імпортувати тестовий набір з файла `cifar-10-batches-py/test_batch`.

```
test_data = unpickle('./cifar-10-batches-py/test_batch')
test_x = grayscale(test_data['data'])
#test_labels = np.array(test_data['labels'])
```

***Примітка.*** В ОС Windows код стає `test_data = unpickler("E:\cifar-10-batches-py\test_batch")`

Можемо спробувати вивести зображення (рис. 13.6), на якому кінь

```
plot_image(test_x[13], shape=[32, 32], cmap = "Greys_r")
```

Рис. 13.6. Зображення коня, отримане в моделі

Для оцінювання моделі використаємо значення пікселів цього зображення і подивимось, чи може кодер реконструювати те саме зображення після скорочення до 1 024 пікселів. Зауважимо, що ми визначаємо функцію для оцінювання моделі на різних зображеннях. Модель має краще працювати лише із зображенням коней.

Функція бере два аргументи:

- df: імпорт даних тесту;
- image_number: вказати, яке зображення імпортувати.

Функцію поділимо на три частини:

1. Переформатуємо зображення до правильного розміру, тобто 1, 1024.
2. Подаємо модель з невидимим зображенням, кодуємо / декодуємо зображення.
3. Виведемо реальне і реконструйоване зображення.

```
def reconstruct_image(df, image_number = 1):
    ## Частина 1: Переформатування зображення до потрібного розміру
тобто 1, 1024
    x_test = df[image_number]
    x_test_1 = x_test.reshape((1, 32*32))

    ## Частина 2: Подача моделі з невидимим зображенням,
кодування/декодування зображення
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        sess.run(iter.initializer, feed_dict={x: x_test_1,
                                        batch_size: 1})
    ## Частина 3: Виведення реального і відновленого зображень
      # Відновлення змінних з диска.
        saver.restore(sess, "./model.ckpt")
        print("Model restored.")
      # Відновлення зображення
        outputs_val = outputs.eval()
        print(outputs_val.shape)
        fig = plt.figure()
      # Plot real
        ax1 = fig.add_subplot(121)
        plot_image(x_test_1, shape=[32, 32], cmap = "Greys_r")
      # Plot estimated
        ax2 = fig.add_subplot(122)
        plot_image(outputs_val, shape=[32, 32], cmap = "Greys_r")
        plt.tight_layout()
        fig = plt.gcf()
```

Якщо функція оцінювання визначена, можемо переглянути реконструйоване зображення (рис. 13.7).

```
reconstruct_image(df =test_x, image_number = 13)
INFO:TensorFlow:Restoring parameters from ./model.ckpt
Model restored.
(1, 1024)
```

**Висновки**

Основна мета автоенкодера – стиснути вхідні дані, а потім розпакувати їх на виході, схожими на оригінальні дані.

Архітектура автоенкодера симетрична відносно центрального шару.

Можемо створити автоенкодер, використовуючи:

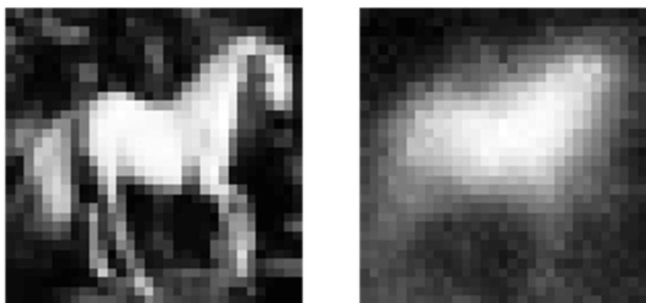


Рис. 13.7. Реконструйоване зображення

`partial`: для створення повністю пов'язаних шарів з типовим налаштуванням:

```
tf.layers.dense,
activation=tf.nn.elu,
kernel_initializer=xav_init,
kernel_regularizer=l2_regularizer
```

- `dense_layer()`: для створення матриці перемноження.

Можемо визначити функцію втрат і оптимізацію з:

```
loss = tf.reduce_mean(tf.square(outputs - features))
optimizer = tf.train.AdamOptimizer(learning_rate)
train  = optimizer.minimize(loss)
```

Насамкінець запускаємо сеанс для тренування моделі.

# 14. Рекурентна нейронна мережа: приклад з TensorFlow

## Що таке RNN

Структура рекурентної нейронної мережі (Recurrent Neural Network – RNN) відносно проста і переважно стосується множення матриці. На першому кроці вхідні дані множать на початкові випадкові вагові коефіцієнти, додають зміщення. Далі вони перетворюються за допомогою функції активації, а вихідні значення використовуються для прогнозування. Цей крок дає уявлення про те, як далеко мережа перебуває від реальності.

Використовується показник – втрати. Чим більша функція втрат, тим «тупішою» є модель. Для вдосконалення знань про мережу потрібна деяка оптимізація шляхом регулювання вагових коефіцієнтів мережі. Стохастичний градієнтний спуск – це метод, що застосовується для зміни значень вагових коефіцієнтів у правильному напрямку. Після того, як буде здійснено коригування, мережа може використати ще одну групу даних для перевірки своїх нових знань.

Помилка у попередньому прикладі менша, ніж раніше, але недостатньо мала. Крок оптимізації виконується ітераційно, поки помилка не буде зведена до мінімуму, тобто неможливо отримати більше інформації.

Проблема з цим типом моделі полягає в тому, що вона не має пам'яті. Це означає, що вхід і вихід незалежні, а отже, модель не переймається тим, що було раніше. Виникає певна

проблема, коли потрібно передбачити часові ряди чи пропозиції, оскільки мережа повинна мати інформацію про історію даних або попередні слова.

Для подолання цієї проблеми розроблено новий тип архітектури: рекурентна нейронна мережа (RNN – Recurrent neural network).

Рекурентна нейронна мережа виглядає доволі схоже на традиційну нейронну мережу, за винятком того, що до нейронів додається стан пам’яті. Обчислення для включення пам’яті прості[33] [33].

Уявіть просту модель із лише одним нейроном, що живиться пакетом даних. У традиційній нейронній мережі модель генерує вихід шляхом множення вхідного значення на вагові коефіцієнти і додавання функції активації. За допомогою RNN цей вихід повертається назад кількість разів. Вказуємо **часовий крок** – через скільки часу вихід стає входом наступного множення матриці.

Наприклад, на рис. 14.1 можна побачити мережу, яка складається з одного нейрона. Мережа обчислює множення матриць вхідних даних і вагових коефіцієнтів, а також додає нелінійність завдяки функції активації. Результат стає виходом на **t-1**. Цей вихід буде входом множення другої матриці.

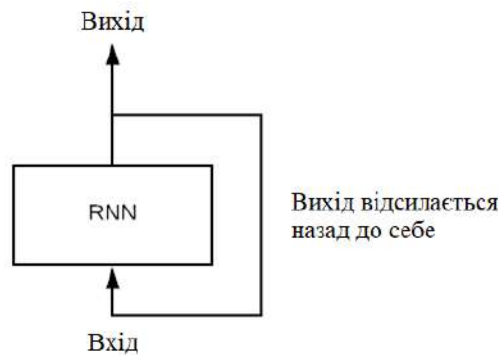



Рис. 14.1. Мережа RNN з одним нейроном

Далі кодуємо просту RNN у TF, щоб зрозуміти кроки, а також форму виводу.

Мережа має:

- чотири входи;
- шість нейронів;
- двочасові кроки.

Мережа діятиме так, як зображено на рис. 14.2.

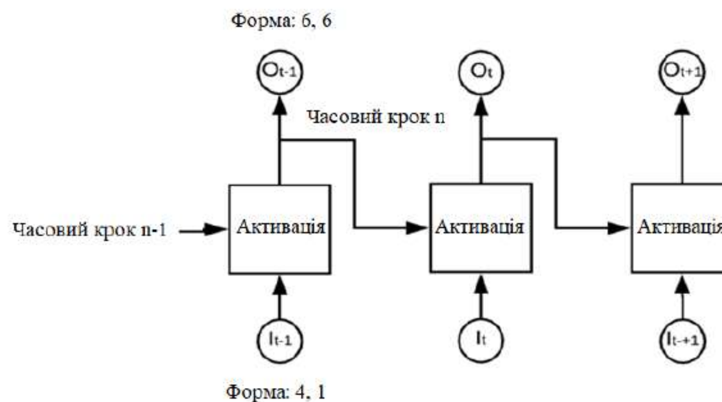



Рис. 14.2. Структура простої RNN мережі

[33] https://www.guru99.com/rnn-tutorial.html

Мережа називається «рекурентною», тому що вона виконує ту ж саму операцію в кожному активованому квадраті. Мережа обчислює вагові коефіцієнти входів й попереднього виходу, перш ніж використовувати функцію активації.

```
import numpy as np
import tensorflow as tf
n_inputs = 4
n_neurons = 6
n_timesteps = 2
## Дані - це послідовність чисел від 0 до 9 і поділені на три групи даних.
## Дані
X_batch = np.array([
        [[0, 1, 2, 5], [9, 8, 7, 4]], # Партія 1
        [[3, 4, 5, 2], [0, 0, 0, 0]], # Партія 2
        [[6, 7, 8, 5], [6, 5, 4, 2]], # Партія 3
    ])
```

Можна побудувати мережу із заповнювачем для даних, рекурентного періоду і виводу.

1. Визначаємо заповнювач для даних

```
X = tf.placeholder(tf.float32, [None, n_timesteps, n_inputs])
```

Тут:

- `None`: невідоме і буде дорівнювати розміру партії;
- `n_timesteps`: кількість разів, коли мережа буде надсилати вихід до нейрона;
- `n_inputs`: число входів на партію.

2. Визначаємо рекурентну мережу

Як наведено на рис. 14.2, мережа складається з 6 нейронів. Мережа обчислить два значення:

- вхідні дані з першим набором вагових коефіцієнтів (тобто 6: дорівнює кількості нейронів);
- попередній вихід з другим набором вагових коефіцієнтів (тобто 6: відповідає кількості вихідних даних).

Зауважимо, що під час першого подання значення попереднього виходу дорівнюють нулям, оскільки у нас немає жодного значення.

Об'єктом побудови RNN є `tf.contrib.rnn.BasicRNNCell` з аргументом `num_units` для визначення кількості входів.

```
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
```

Отже, коли мережа визначена, можна обчислити виходи і стани

```
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```

Пропонований об'єкт використовує внутрішній цикл для множення матриць відповідну кількість разів. Зауважимо, що рекурентний нейрон є функцією всіх входів попередніх кроків часу. Ось так мережа будує власну пам'ять. Інформація з попереднього часу може поширюватися у майбутньому часі. Це магія рекурентної нейронної мережі

```
## Визначення форми тензора
X = tf.placeholder(tf.float32, [None, n_timesteps, n_inputs])
## Визначення мережі
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
init = tf.global_variables_initializer()
init = tf.global_variables_initializer()
with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
print(states.eval(feed_dict={X: X_batch}))

[[ 0.38941205 -0.9980438   0.99750966  0.7892596   0.9978241   0.9999997
]
 [ 0.61096436  0.7255889   0.82977575 -0.88226104  0.29261455 -
0.15597084]
 [ 0.62091285 -0.87023467  0.99729395 -0.58261937  0.9811445
0.99969864]]
```

Для пояснення виведемо значення попереднього стану. Вихід, наведений вище, показує вихід з останнього стану. Тепер виведемо всі виходи і зможемо помітити, що стани – це попередній вихід кожної партії. Тобто попередній вихід містить інформацію про всю послідовність.

```
print(outputs_val)
print(outputs_val.shape)
[[[-0.75934666 -0.99537754  0.9735819  -0.9722234  -0.14234993
   -0.9984044 ]
  [ 0.99975264 -0.9983206   0.9999993  -1.         -0.9997506
   -1.        ]]

 [[ 0.97486496 -0.98773265  0.9969686  -0.99950117 -0.7092863
   -0.99998885]
  [ 0.9326837   0.2673438   0.2808514  -0.7535883  -0.43337247
    0.5700631 ]]

 [[ 0.99628735 -0.9998728   0.99999213 -0.99999976 -0.9884324
   -1.        ]
  [ 0.99962527 -0.9467421   0.9997403  -0.99999714 -0.99929446
   -0.9999795 ]]]
(3, 2, 6)
```

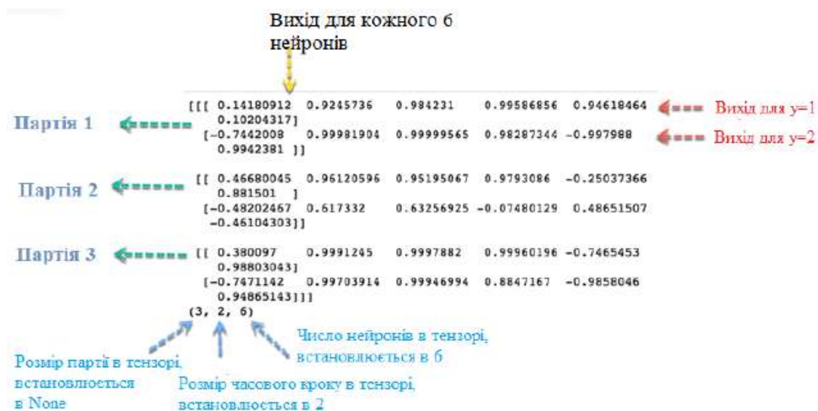



Рис. 14.3.

Вихід має форму (3, 2, 6) (рис. 14.3):

- 3: число партій;
- 2: число часових кроків;
- 6: число нейронів.

Оптимізація періодичної нейронної мережі ідентична традиційній нейронній мережі.

**Застосування RNN**

RNN має широке застосування, особливо якщо йдеться про прогнозування майбутнього. У фінансовій галузі RNN може бути корисною для прогнозування цін на акції або ознак напряму біржового ринку (тобто, позитивний або негативний).

RNN корисний для автономного автомобіля, оскільки він допомагає уникнути ДТП, передбачаючи траєкторію руху транспортного засобу.

RNN широко застосовується для аналізу тексту, у субтитрах зображення, для аналізу настроїв, а також машинному перекладі. Наприклад, можна використати огляд фільму, щоб зрозуміти почуття, яке отримав глядач після перегляду фільму. Автоматизація цього завдання дуже корисна, коли у кінокомпанії не вистачає часу на перегляд, маркування, консолідацію й аналіз оглядів. Машина може виконувати роботу з більш високим рівнем точності.

**Обмеження RNN**

Теоретично RNN має нести інформацію про час. Однак поширювати всю цю інформацію досить складно, коли крок часу занадто довгий. Якщо в мережі є занадто багато глибоких шарів, вона стає недосяжною. Ця проблема називається проблемою зникнення градієнта. Пам'ятаємо, що нейронна мережа оновлює вагові коефіцієнти за допомогою алгоритму градієнтного спуску. Градієнти зменшуються, коли мережа просувається до нижчих шарів.

На закінчення градієнти залишаються постійними, тому немає простору для вдосконалення. Модель вчиться зі зміною градієнта: ця зміна впливає на вихід мережі. Однак у разі, якщо різниця у градієнті занадто мала (тобто вагові коефіцієнти змінюються мало), мережа нічого не може навчитися і йде на вихід. Тому мережа, що стикається з проблемою градієнта, який зникає, не може збігатися до прийнятного рішення.

**Поліпшення LSTM**

З метою подолання потенційної проблеми зникаючого градієнта, з якою стикаються RNN, троє дослідників Хохрейтер, Шмідхубер і Бенджо вдосконалили RNN архітектуру, яка називається Long Short-Term Memory (LSTM) – довга короткострокова пам'ять. Отже, LSTM надає мережі відповідну минулу інформацію для більш пізнього часу. Машина використовує кращу архітектуру для вибору і перенесення інформації до пізнішого часу.

LSTM-архітектура доступна у TF, `tf.contrib.rnn.LSTMCell`. LSTM виходить за межі цього розділу. Для отримання додаткової інформації можна звернутися до офіційної документації[34] [34].

**RNN у часових рядах**

Розглянемо, як використовувати RNN з даними часових рядів. Часові ряди залежать від попереднього часу, а отже, минулі значення включають відповідну інформацію, яку може

[34] https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/LSTMCell

дізнатися мережа. Ідея прогнозування часових рядів полягає в оцінюванні майбутньої вартості серії (наприклад, ціни акцій, температури, ВВП тощо).

Підготовка даних для RNN і часових рядів має особливості. По-перше, мета полягає в тому, щоб передбачити наступне значення серії, тобто використати попередню інформацію для оцінювання значення при **t+1**. Мітка дорівнює послідовності введення і зміщена на один період вперед. По-друге, кількість вхідних даних встановлюється в 1, тобто одне спостереження за раз. Отже, часовий крок дорівнює послідовності числового значення. Наприклад, якщо встановити крок часу в 10, послідовність введення повернеться десять разів поспіль.

Подивимося на графіки (рис. 14.4), де наведено дані часового ряду зліва і вигадана послідовність введення праворуч. Створюємо функцію для повернення набору даних із випадковим значенням за кожен день із січня 2001 до грудня 2016 року.

```
# Для графіка чудових фігур
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
def create_ts(start = '2001', n = 201, freq = 'M'):
    rng = pd.date_range(start=start, periods=n, freq=freq)
    ts = pd.Series(np.random.uniform(-18, 18, size=len(rng)),
rng).cumsum()
    return ts
ts= create_ts(start = '2001', n = 192, freq = 'M')
ts.tail(5)
```

**Результат на виході:**

```
2016-08-31    -93.459631
2016-09-30    -95.264791
2016-10-31    -95.551935
2016-11-30   -105.879611
2016-12-31   -123.729319
Freq: M, dtype: float64
ts = create_ts(start = '2001', n = 222)

# Ліворуч
plt.figure(figsize=(11,4))
plt.subplot(121)
plt.plot(ts.index, ts)
plt.plot(ts.index[90:100], ts[90:100], "b-", linewidth=3, label="A
training instance")
plt.title("A time series (generated)", fontsize=14)

# Праворуч
plt.subplot(122)
plt.title("A training instance", fontsize=14)
plt.plot(ts.index[90:100], ts[90:100], "b-", markersize=8,
label="instance")
plt.plot(ts.index[91:101], ts[91:101], "bo", markersize=10,
label="target", markerfacecolor='red')
plt.legend(loc="upper left")
plt.xlabel("Time")

plt.show()
```

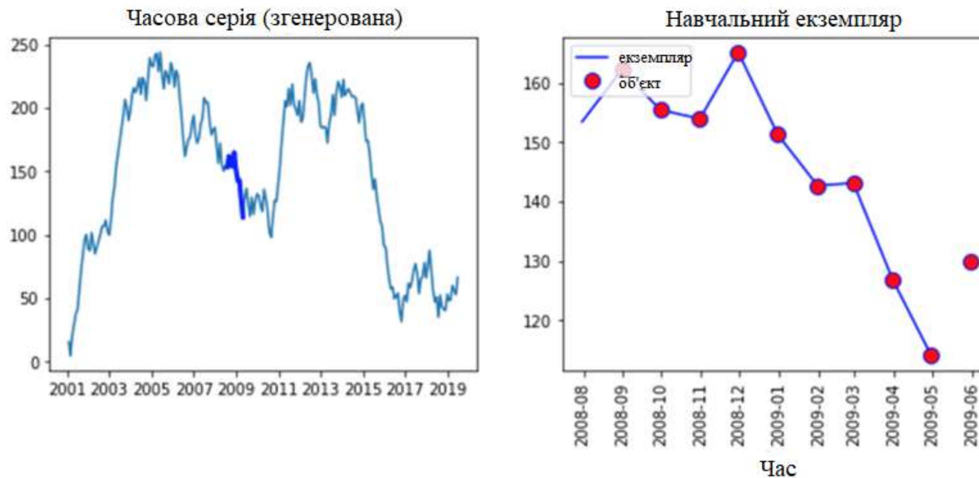



Рис. 14.4. Функція для повернення набору даних із випадковим значенням

У правій частині графіка наведено всі серії. Вони починаються з 2001 року і закінчується у 2019 році. Немає сенсу подавати всі дані в мережу, натомість потрібно створити пакет даних завдовжки, що дорівнює кроку часу. Ця партія буде змінною **X**. Змінна **Y** така сама, як **X**, але зміщена на один період (тобто потрібно прогнозувати **t+1**).

Обидва вектори мають однакову довжину. Можемо бачити це у правій частині вищевказаного графіка. Рядок представляє десять значень вводу **X**, а червоні точки – десять значень мітки **Y**. Зауважимо, що мітка починається на один період попереду від **X** і закінчується на один період після.

## Побудова RNN для прогнозування часових рядів

Отже, настав час створити свою першу RNN, щоб передбачити серію вище. Для моделі потрібно вказати деякі гіперпараметри (параметри моделі, тобто кількість нейронів тощо):

- кількість вводів: 1;
- крок часу (вікна у часових рядах): 10;
- кількість нейронів: 120;
- кількість виходів: 1.

Наша мережа вивчатиме через 10 днів і міститиме 120 повторюваних нейронів. Ми живимо модель одним входом, тобто одним днем. Намагатимемося змінювати значення, щоб побачити, чи покращилася модель.

Перш ніж побудувати модель, треба розділити набір даних на набір для тренування і тестовий набір. Повний набір даних має 222 точки даних. Використаємо перші 201 точку для тренування моделі, а останні 21 – для тестування.

Визначивши тренувальний і тестовий набори, створимо об’єкт, що містить партії. У цих партіях є значення **X** і **Y**. Пам’ятаємо, що значення **X** мають один період відставання. Тому використовуємо перші 200 спостережень, а крок часу дорівнює 10. Об’єкт `X_batches` має містити 20 партій розміром 10*1. `Y_batches` має таку ж форму, що і об’єкт `X_batches`, але на один період попереду.

**Крок 1. Створюємо тренування і тестуємо**

Насамперед перетворимо серію в рядковий масив, а потім визначимо вікна (тобто кількість часу, коли мережа вчиться), кількість вводу, виводу і розмір тренувального набору.

```
series = np.array(ts)
n_windows = 20
n_input =  1
n_output = 1
size_train = 201
```

Після цього просто розділимо масив на два набори даних.

```
## Розділення даних
train = series[:size_train]
test = series[size_train:]
print(train.shape, test.shape)
(201,) (21,)
```

**Крок 2. Будуємо функцію для повернення `X_batches` і `y_batches`**

Для спрощення можна створити функцію, яка повертає два різних масиви: один для `X_batches` , а другий для `y_batches`.

Запишемо функцію побудови партій.

Зауважимо, що партії **X** відстають на один період (беремо значення **t-1**). Вихід функції буде мати три виміри: перший дорівнює кількості партій, другий – розмір вікон, а останній – кількість входів.

Складна частина полягає у правильному виборі точок даних. Для точок даних **X** вибираємо спостереження від t = 1 до t = 200, а для точки даних **Y** повертаємо спостереження від t = 2 до 201. Після отримання правильних точок даних, просто змінимо форму серії.

Щоб побудувати об'єкт з партіями, потрібно розділити набір даних на десять партій однакової довжини (тобто 20). Можна використовувати метод переформатування і пропустити -1, щоб серія була схожа на розмір партії. Значення 20 – це кількість спостережень за партію, а 1 – кількість введення.

Необхідно зробити той самий крок, але для мітки.

Зауважимо, що слід перенести дані на кількість часу, який ми будемо прогнозувати. Наприклад, якщо треба передбачити один день, то змістимо серію на 1, а якщо треба прогнозувати два дні, то змістимо дані на 2.

```
x_data = train[:size_train-1]: Select all the training instance minus
one day
X_batches = x_data.reshape(-1, windows, input): create the right shape
for the batch e.g (10, 20, 1)
def create_batches(df, windows, input, output):
    ## Створюємо X
        x_data = train[:size_train-1] # Select the data
        X_batches = x_data.reshape(-1, windows, input)  # Reshape the
data
    ## Створюємо y
        y_data = train[n_output:size_train]
        y_batches = y_data.reshape(-1, windows, output)
        return X_batches, y_batches
```

Отже, якщо функція визначена, то можна запустити її для створення партій.

```
X_batches, y_batches = create_batches(df = train,
                                      windows = n_windows,
                                      input = n_input,
                                      output = n_output)
```

Можемо виводити форму, щоби переконатися, що розмірність коректна.

```
print(X_batches.shape, y_batches.shape)
(10, 20, 1) (10, 20, 1)
```

Необхідно створити тестовий набір лише з однієї партії даних і 20 спостережень.

Зауважимо, що ми прогнозуємо дні за днями, а це означає, що друге передбачуване значення буде базуватися на справжньому значенні першого дня (**t+1**) тестового набору даних. Тож буде відоме справжнє значення.

Якщо треба прогнозувати **t+2** (тобто на два дні вперед), то слід використати передбачуване значення **t+1**; якщо збираємося передбачити **t+3** (на три дні вперед), то треба використати передбачувані значення **t+1** і **t+2**. Звичайно, важко точно передбачити на **t+n** дні вперед.

```
X_test, y_test = create_batches(df = test, windows = 20,input = 1,
output = 1)
print(X_test.shape, y_test.shape)
(10, 20, 1) (10, 20, 1)
```

Отже, розмір партії готовий, тож можна побудувати архітектуру RNN. Пам'ятаємо, що у нас 120 рецидивуючих нейронів.

**Крок 3. Будуємо модель**

Для створення моделі треба визначити три частини:

1. Змінні з тензорами.
2. RNN.
3. Втрати і оптимізація.

**Крок 3.1. Змінні**

Треба вказати змінні **X** і **y** відповідній формі. Цей крок банальний. Тензор має той самий розмір, що і для об'єктів `X_batches` і `y_batches`.

Наприклад, тензор **X** є заповнювачем і має три виміри:

- Note: розмір партії;
- n_windows: довжина вікна, тобто тривалість часу, коли модель дивиться назад;
- n_input: число входів.

Маємо:

```
tf.placeholder(tf.float32, [None, n_windows, n_input])
## 1. Будуємо тензори
X = tf.placeholder(tf.float32, [None, n_windows, n_input])
y = tf.placeholder(tf.float32, [None, n_windows, n_output])
```

**Крок 3.2. Створюємо RNN**

У другій частині треба визначити архітектуру мережі. Як і раніше, використовуємо об'єкт `BasicRNNCell` і `dynamic_rnn` з оцінювача TF.

```
## 2. Створення моделі
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=r_neuron,
activation=tf.nn.relu)
rnn_output, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```

Наступна частина трохи складніша, але дає змогу швидшого обчислення. Треба перетворити запущений вихід у повністю пов'язаний шар, а потім перетворити його знову, щоб мати той самий розмір, що і вхідний.

```
stacked_rnn_output = tf.reshape(rnn_output, [-1, r_neuron])
stacked_outputs = tf.layers.dense(stacked_rnn_output, n_output)
outputs = tf.reshape(stacked_outputs, [-1, n_windows, n_output])
```

**Крок 3.3. Створюємо втрати і оптимізацію**

Оптимізація моделі залежить від завдання, яке ми виконуємо. У попередньому розділі про CNN наша мета була – класифікувати зображення, а у цьому розділі мета дещо інша. Нам пропонується зробити прогнозування на безперервну змінну, яка порівнюється з класом.

Ця різниця важлива, оскільки вона змінює проблему оптимізації. Проблема оптимізації для безперервної змінної полягає у мінімізації середньої квадратичної помилки. Для побудови цих показників у TF можемо використати:

```
tf.reduce_sum(tf.square(outputs - y))
```

Решта коду така сама, як і раніше: використовуємо оптимізатор Адама для зменшення втрат (тобто, MSE):

```
tf.train.AdamOptimizer(learning_rate=learning_rate)
optimizer.minimize(loss)
```

Тепер можемо спакувати все разом, а отже, модель готова до навчання.

```
tf.reset_default_graph()
r_neuron = 120

## 1. Побудова тензорів
X = tf.placeholder(tf.float32, [None, n_windows, n_input])
y = tf.placeholder(tf.float32, [None, n_windows, n_output])

## 2. Створення моделі
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=r_neuron,
activation=tf.nn.relu)
rnn_output, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

stacked_rnn_output = tf.reshape(rnn_output, [-1, r_neuron])
stacked_outputs = tf.layers.dense(stacked_rnn_output, n_output)
outputs = tf.reshape(stacked_outputs, [-1, n_windows, n_output])

## 3. Втрати + оптимізація
learning_rate = 0.001

loss = tf.reduce_sum(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

Будемо тренувати модель за допомогою 1 500 епох і виводити втрати кожні 150 ітерацій. Після навчання моделі оцінимо її на тестовому наборі і створимо об'єкт, який містить передбачення.

```
iteration = 1500

with tf.Session() as sess:
    init.run()
    for iters in range(iteration):
        sess.run(training_op, feed_dict={X: X_batches, y: y_batches})
        if iters % 150 == 0:
            mse = loss.eval(feed_dict={X: X_batches, y: y_batches})
            print(iters, "\tMSE:", mse)

    y_pred = sess.run(outputs, feed_dict={X: X_test})
0	MSE: 502893.34
150	MSE: 13839.129
300	MSE: 3964.835
450	MSE: 2619.885
600	MSE: 2418.772
750	MSE: 2110.5923
900	MSE: 1887.9644
1050	MSE: 1747.1377
1200	MSE: 1556.3398
1350	MSE: 1384.6113
```

Отже, можемо побудувати фактичне значення серії з передбачуваним значенням. Якщо наша модель виправлена, тоді передбачувані значення слід поставити поверх фактичних значень (рис. 14.5).

Як бачимо, модель має можливість вдосконалення. Від нас залежить зміна гіперпараметрів, таких як вікна, розмір партії кількості повторюваних нейронів.

```
plt.title("Forecast vs Actual", fontsize=14)
plt.plot(pd.Series(np.ravel(y_test)), "bo", markersize=8,
label="Actual", color='green')
plt.plot(pd.Series(np.ravel(y_pred)), "r.", markersize=8,
label="Forecast", color='red')
plt.legend(loc="lower left")
plt.xlabel("Time")
plt.show()
```

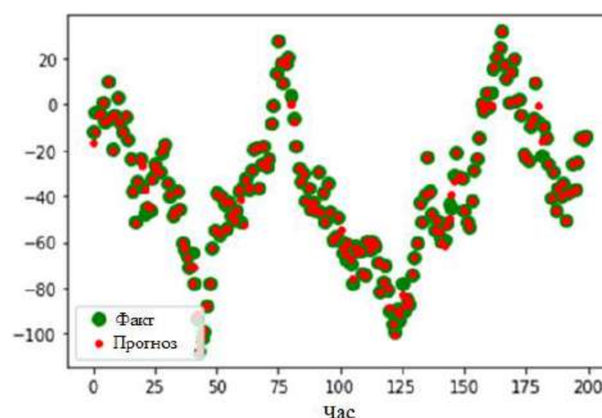



Рис. 14.5. Прогноз проти фактичних значень

**Висновки**

Рекурентна нейронна мережа – це надійна архітектура для вирішення часових рядів або аналізу тексту. Вихід попереднього стану – це зворотний зв'язок для збереження пам'яті мережі за часом або послідовності слів.

У TF можемо використати наведені коди для тренування рекурентної нейронної мережі для часових рядів:

**Параметри моделі**

```
n_windows = 20
n_input =  1
n_output = 1
size_train = 201
```

**Визначення моделі**

```
X = tf.placeholder(tf.float32, [None, n_windows, n_input])
y = tf.placeholder(tf.float32, [None, n_windows, n_output])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=r_neuron,
activation=tf.nn.relu)
rnn_output, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

stacked_rnn_output = tf.reshape(rnn_output, [-1, r_neuron])
stacked_outputs = tf.layers.dense(stacked_rnn_output, n_output)
outputs = tf.reshape(stacked_outputs, [-1, n_windows, n_output])
```

**Побудова оптимізації**

```
learning_rate = 0.001

loss = tf.reduce_sum(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
```

**Тренування моделі**

```
init = tf.global_variables_initializer()
iteration = 1500

with tf.Session() as sess:
    init.run()
    for iters in range(iteration):
        sess.run(training_op, feed_dict={X: X_batches, y: y_batches})
        if iters % 150 == 0:
            mse = loss.eval(feed_dict={X: X_batches, y: y_batches})
            print(iters, "\tMSE:", mse)

    y_pred = sess.run(outputs, feed_dict={X: X_test})
```

# 15. Проєкти машинного навчання на мікрокомп’ютерах

## Мікрокомп’ютери для машинного навчання

Якщо йдеться про машинне навчання з використанням мікрокомп’ютерів, то треба зауважити, що ми не можемо навчити модель глибокого навчання на RPi або іншому мікрокомп’ютері. На їх платах не вистачає можливостей, щоб виконати величезну кількість множень з плаваючою комою, необхідних під час тренувань. Отже, ми можемо лише імпортувати і запускати вже навчену модель на мікрокомп’ютерах.

Розглянемо основні плати мікрокомп’ютерів, їх параметри і можливості додаткових модулів для машинного навчання.

### Raspberry Pi

| Параметри | Raspberry Pi 4 B | Raspberry Pi 3 B+ |
|---|---|---|
| Дата випуску | 24 червня 2019 | 14 березня 2018 |
| Тип SoC (Процесор) | Broadcom BCM2711 | Broadcom BCM2837B0 |
| Тип ядра | Cortex-A72 64-bit (ARMv8) | Cortex-A53 64-bit (ARMv8) |
| Кількість ядер | Quad-Core | |
| GPU | VideoCore VI | VideoCore IV |
| Мультимедіа | H.265 decode (4Kp60)<br>H.264 decode (1080p60)<br>H.264 encode (1080p30)<br>OpenGL ES 1.1, 2.0, 3.0 Graphics | H.264, MPEG-4 decode (1080p30)<br>H.264 encode (1080p30)<br>OpenGL ES 1.1, 2.0 Graphics |
| Тактова частота CPU | 1.5 GHz | 1.4 GHz |
| Постійна пам'ять | microSD | |
| Оперативна пам'ять | LPDDR4 1GB, 2GB або 4GB | LPDDR2 1GB |
| Ethernet | True Gigabit Ethernet | Gigabit over USB 2.0 (Max 300Mbps) |
| Порт USB | 2 x USB 3.0 + 2 x USB 2.0 | 4 x USB 2.0 |
| HDMI | 2 x micro HDMI support Dual Display | 1 x full size HDMI |
| WiFi | 802.11 b/g/n/ac (2.4 ГГц + 5 ГГц) | |
| Bluetooth | 5.0 + BLE | 4.2 + BLE |
| Антена | PCB Antenna | |
| GPIO | 40 виводів | |
| Операційна система | Raspbian (> 24 June 2019) | Raspbian (> March 2018) |
| Розміри | 85 мм x 56 мм | |
| Вхід живлення | 5V via USB Type C (upto 3A)<br>5V via GPIO header (upto 3A)<br>Power over Ethernet, вимагає PoE HAT | 5V via USB Micro B (upto 2.5A)<br>5V via GPIO header (upto 3A)<br>Power over Ethernet, вимагає PoE HAT |

Рис. 15.1. Порівняння параметрів Raspberry Pi 4 B і Raspberry Pi 3 B+

Розгляд почнемо з найбільш поширеного Raspberry Pi (в усьому світі продано понад 25 млн плат). Нині для машинного навчання використовують дві модифікації: RPi 3 B+ і RPi 4 B. Для порівняння на рис. 15.1 наведено параметри обох моделей.

Червоним кольором позначено основні відмінності між платами. Збільшення оперативної пам’яті і новий її тип у RPi 4 B є основним фактором використання цієї плати для машинного навчання. Згідно експериментальних даних RPi 4 B дає змогу прискорити опрацювання зображень в три рази порівняно з RPi 3 B+ (до 25 кадрів у секунду).

**JeVois**

JeVois = відеокамера + 4 ядра CPU + USB відео + послідовний порт – все в крихітній і автономній конструкції вагою 17 г (рис. 15.2). Вставляємо microSD-карту, попередньо завантажену алгоритмами комп’ютерного зору з відкритим кодом (включаючи OpenCV 4.1.0, TensorFlow, Caffe, Darknet та ін.), підключаємо до робочого столу ноутбука та / або Arduino і негайно отримуємо в своїх проєктах комп’ютерний зір. JeVois запускає нейронну мережу глибокого навчання, яка здатна розпізнати 1 000 різних типів об’єктів в реальному часі.

JeVois має такі параметри апаратного забезпечення: чотириядерний процесор Cortex-A7, відео з двоядерним Mali-400, частота процесора 1.35 ГГц, 256 МБ оперативної пам’яті DDR3, USB 2.0, UART, камера 1,3 мегапікселів, живлення 5 В через USB (0,7 А).

Програмне забезпечення: чотириядерний ARM працює на ОС Linux, як і RPi. Він може підтримувати будь-яке програмне забезпечення, у т. ч. популярні програмні пакети: Python, OpenCV, TensorFlow, DarkNet (YOLO), OpenMP.

Серед недоліків – недостатній обсяг оперативної пам’яті для задекларованих задач машинного навчання.

Рис. 15.2. Мікрокомп’ютер JeVois

**Intel Neural Compute Stick 2**

«Флешка» USB 3 від Intel (рис.15.3) реалізує нейронну мережу для ПК, а також таких окремих плат, як RPi, що надзвичайно прискорює тензорну арифметику.

В основі модуля візуальний процесор Intel® Movidius™ Myriad™ X, який має 16 векторних 128 бітних LIW-ядра SHAVE (Streaming Hybrid Architecture Vector Engine). Myriad X має апаратну підтримку кодування 4k-відео зі швидкістю до 60 кадрів за секунду. Максимальна теоретична продуктивність Movidius Myriad X становить 4 Терафлопси, при цьому енергоспоживання SoC не більше 1 Вт.

Рис.15.3. Intel Neural Compute Stick 2

Підтримуються інфраструктури: TensorFlow і Caffe. Сумісні операційні системи: Ubuntu 16.04.3 LTS (64-розрядна), CentOS 7.4 (64-розрядна) і Windows 10 (64-розрядна). Також Intel Neural Compute Stick 2 підтримує Open Visual Inference & Neural Network Optimization (OpenVINO). Важливо, що можна об’єднати кілька «флешок» Intel® NCS 2 в одну платформу для масштабування продуктивності.

Рис. 15.4. Google Coral USB

**Google Coral USB**

На рис. 15.4 наведено «голий» Google Coral Edge TPU з інтерфейсом USB 3.0. Параметри точно такі, як і у плати Coral, яка розглядається нижче.

**Rockchip RK1808 NPU**

Нейронний процесорний блок від Rockchip у форматі USB 3 (рис.15.5). У нього є 1 Гб оперативної пам’яті і 8 Гб пам’яті eMMC. Має двоядерний процесор Arm Cortex-A35, що підтримує гібридну роботу INT8/INT16/FP16.

Рис.15.5. NPU Rockchip AI RK1808

**Rockchip RK3399Pro**

На рис.15.6 8-ми і 16-бітовий нейронний процесорний блок із SoC – шестиядерним процесором Rochchip RK3399Pro архітектури big.LITTLE з 2-ма ядрами Cortex A72 частотою до 1.8/2.0 ГГц, 4-ма ядрами Cortex A53@1.4 ГГц і графічним процесором Arm Mali-T860 MP46x Cortex-A53. Оперативна пам'ять – 3 або 6 ГБ. Мережа – гігабітний Ethernet, модуль WiFi, живлення через порт USB type-C.

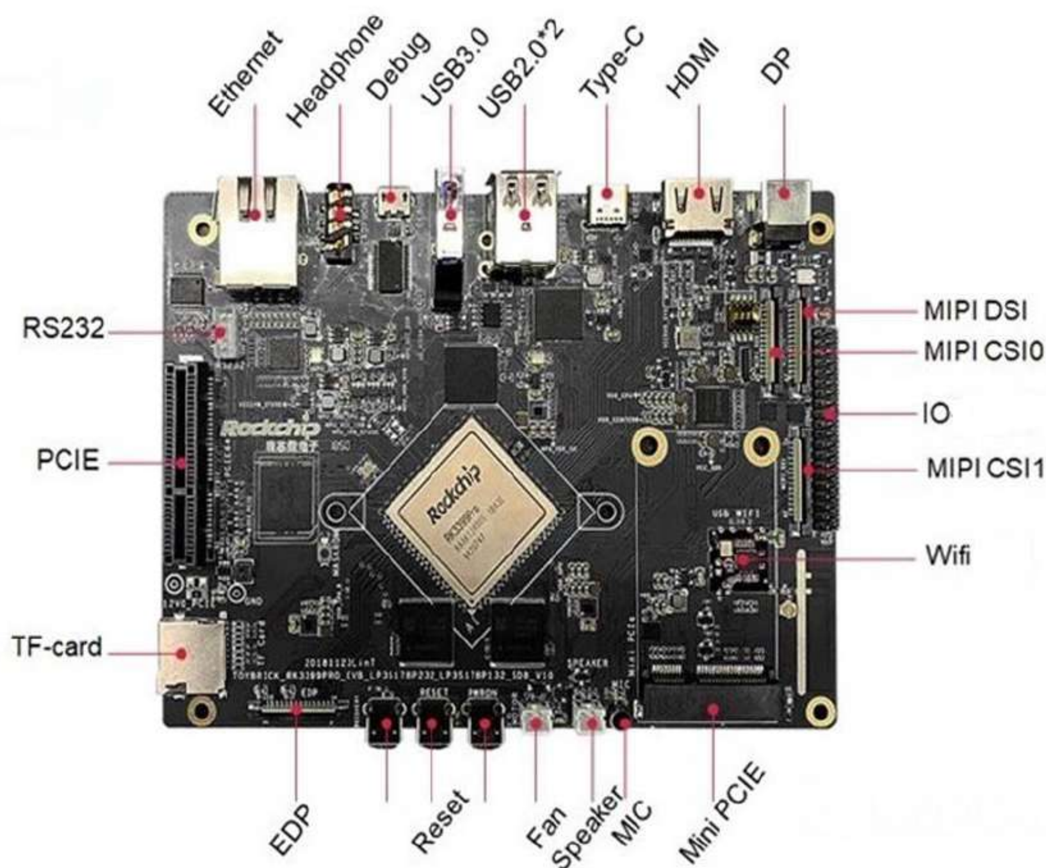



Рис.15.6. NPU Rockchip RK3399Pro

Плата RK3399Pro поставляється з попередньо встановленими операційними системами Android і Linux в конфігурації з подвійним завантаженням. Таке є програмне забезпечення для використання NPU з підтримкою моделей на TensorFlow/TensorFlow Lite/Caffe.

**HiKey 970**

На рис.15.7 8-бітовий нейронний процесорний блок на Kirin 970 SoC з 4-ма Cortex A73 @ 2.36 ГГц, 4-ма Cortex A53 @ 1.8 ГГц і графічним процесором Mali G72-MP12. Оперативна пам'ять 6 ГБ LPDDR4. Постійна пам'ять на платі – до 64 ГБ eMMC Flash. Мережа – гігабітний Ethernet, модуль WiFi (2.4 ГГц і 5 ГГц з двома антенами), рекомендоване живлення 8–12 В, 2 А.

З програмного забезпечення пропонуються готові образи операційних систем Android, Debian, Lebuntu тощо.

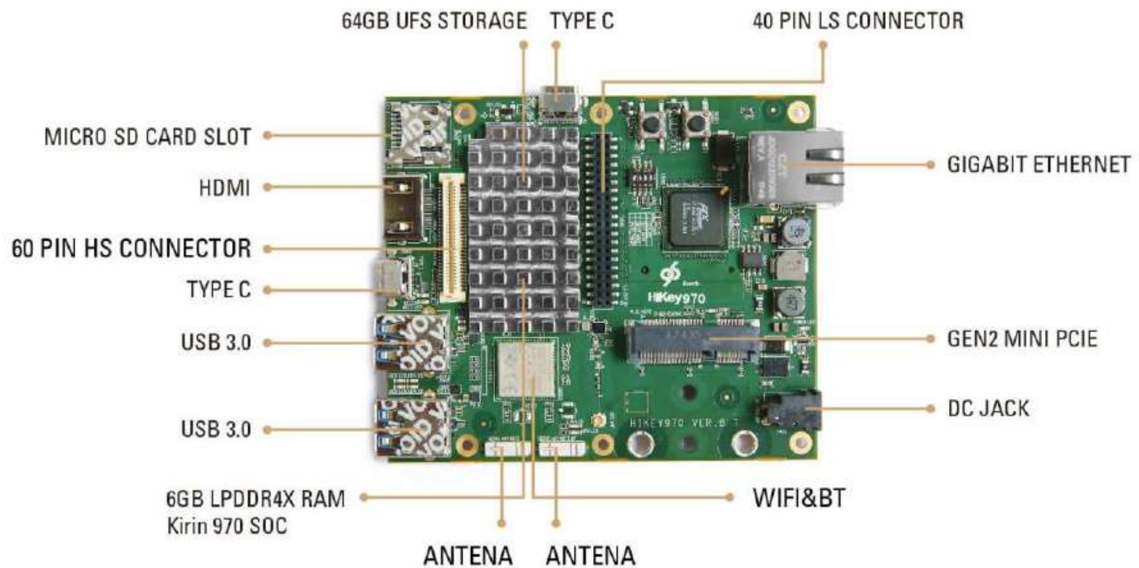



Рис.15.7. NPU HiKey970

**Jetson Nano**

Найважливіше в цій платі — це графічний процесор архітектури NVIDIA Maxwell™ з 128 ядрами NVIDIA CUDA®, тому на платі можна запускати GPU-орієнтовані задачі, типу CUDA або TensorFlow. Основний процесор чотириядерний ARM® Cortex®-A57 MPCore частотою 1,43 ГГц. Оперативна пам'ять 4ГБ LPDDR4, 64-біт, спільна між CPU і GPU.

Плата сумісна з RPi, має 40-виводний роз'єм з різноманітними інтерфейсами (I2C, SPI тощо), а також є роз'єм камери, який теж сумісний з RPi (рис.15.8). Можна припустити, що велика кількість наявних аксесуарів (екрани, плати керування двигунами тощо, тестовані на RPi) будуть працювати.

На платі також є два відеовиходи, гігабітний Ethernet і USB 3.0. Живлення 5 В можна брати через microUSB або через окремий роз'єм. Як і в RPi, програмне забезпечення завантажується з образу мікроSD-карти. Загалом плата дуже схожа на RPi. Серед недоліків – відсутність WiFi.

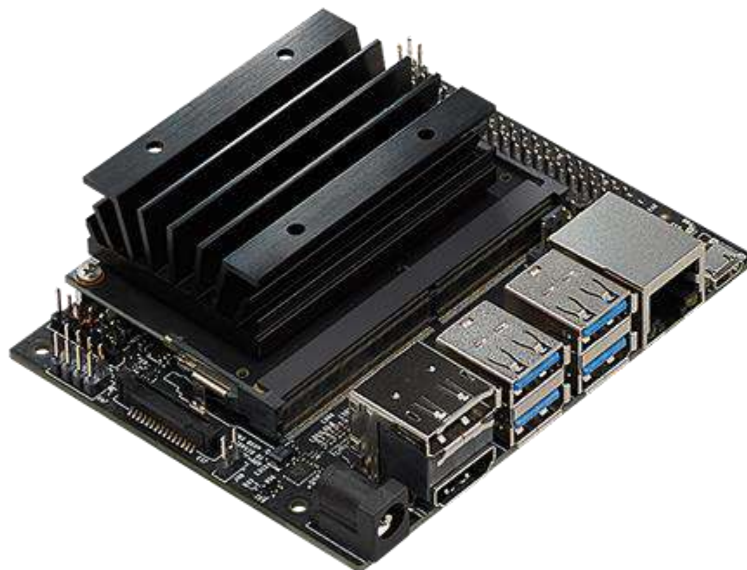


Рис.15.8. GPU Jetson Nano

## Sophon BM1880

На рис. 15.9 наведено 8-бітовий модуль опрацювання для нейронної мережі. В його складі SoC ASIC – процесор Sophon BM1880 з двома ядрами Cortex-A53@1.5 ГГц, одноядерний процесор RISC-V@1 ГГц, а також TPU (Tensor Processing Unit).

Оперативна пам’ять – 1 ГБ LPDDR4@3200 МГц. Постійна флеш-пам’ять – 8 ГБ eMMC + слот для мікроSD-карти. Опрацювання відео – декодер H.264, кодер / декодер MJPEG. Мережа – гігабітний Ethernet, Wifi, Bluetooth. Живлення – 12 В / 2 А.

Плата для розробників Bitmain Sophon Edge працює під управлінням Linux і підтримує операції і моделі DNN, CNN, RNN й LSTM через свій TPU і за допомогою різноманітних інструментів. Плата також підтримує модулі для Arduino і RPi, що дає змогу використовувати її в недорогих і малопотужних DL/ML рішеннях, таких як виявлення і розпізнавання облич, аналіз виразу обличчя, виявлення і розпізнавання об’єктів, розпізнавання номерних знаків автомобіля тощо. Підтримувані фреймворки – Caffe, ONNX, Pytorch і Tensorflow.

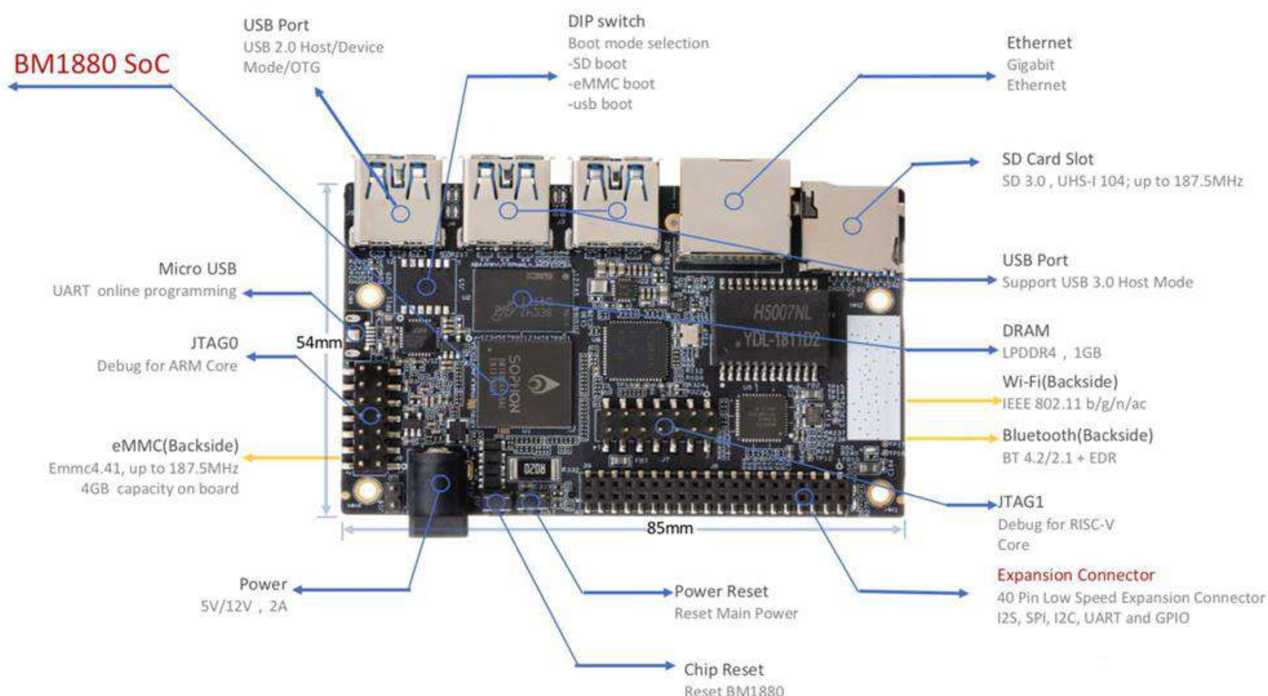



Рис.15.9. Плата для розробника 96Boards AI Sophon Edge

## Google Coral

Coral Dev Board – одноплатний комп’ютер зі знімною системою на модулі (system-on-module - SoM), до складу якої входять SoC, eMMC, бездротові системи передачі даних і Edge TPU. Dev Board може використовуватися як одноплатний комп’ютер, якщо необхідне прискорене опрацювання ML в невеликому форм-факторі, але плата також використовується як оціночний комплект для SoM. Можна використати плату dev для створення прототипів пристроїв Інтернету речей (IoT), а також інших вбудованих систем, які потребують швидкого виведення ML на пристрій, а потім масштабування для виробництва з використанням лише SoM-плати 40 × 48 мм разом зі своєю власною друкованою платою.

SoM базується на системі-на-кристалі (system-on-a-chip - SoC) iMX8M від NXP, але забезпечує його унікальну потужність співпроцесор Edge TPU. Edge TPU – невелика ASIC, розроблена Google, яка забезпечує високопродуктивне виведення ML з низьким енергоспоживанням. Наприклад, він може реалізувати сучасні моделі комп’ютерного зору,

такі як MobileNet v2, зі швидкістю понад 100 кадрів у секунду, а також ефективним енергоспоживанням.

До базової плати входять всі периферійні інтерфейси, необхідні для створення прототипу проєкту, у т. ч. порти USB 2.0/3.0, інтерфейс дисплея DSI, інтерфейс камери CSI-2, порт Ethernet, термінали гучномовців і 40-контактний роз’єм GPIO (рис. 15.10).

Рис.15.10. Плата розробника Coral Dev Board

**Google SoM**

Крихітний модуль 40 × 48 мм з повними входами / виходами і прискорювачем Edge TPU (рис. 15.11) входить до складу попередньої плати розробника Coral Dev Board.

Рис.15.11. Google SoM

На рис. 15.12 наведено деякі моделі нейронних мереж і порівняльні швидкості опрацювання зображення в кадрах за секунду (FPS). Для навчання найефективнішим рішенням за доступною ціною буде комбінація мікрокомп’ютера Raspberry Pi 4 з Intel Neural Compute Stick 2 і фреймворком TF.

Розглянемо в наступних підрозділах використання такої комбінації у проєктах машинного навчання.

| Model | Framework | Raspberry Pi (use TF-Lite) | Raspberry Pi (our NCNN) | Raspberry Pi Intel Neural Stick 2 | Raspberry Pi Google Coral USB | JeVois | Jetson Nano | Google Coral |
|---|---|---|---|---|---|---|---|---|
| EfficientNet-B0 (224x224) | TensorFlow | 14.6 FPS (Pi 3) 25.8 FPS (Pi 4) | - | 95 FPS (Pi 3) 180 FPS (Pi 4) | 105 FPS (Pi 3) 200 FPS (Pi 4) | - | 216 FPS | 200 FPS |
| ResNet-50 (244x244) | TensorFlow | 2.4 FPS (Pi 3) 4.3 FPS (Pi 4) | 1.7 FPS (Pi 3) 3 FPS (Pi 4) | 16 FPS (Pi 3) 60 FPS (Pi 4) | 10 FPS (Pi 3) 18.8 FPS (Pi 4) | - | 36 FPS | 18.8 FPS |
| MobileNet-v2 (300x300) | TensorFlow | 4.4 FPS (Pi 3) 8 FPS (Pi 4) | 8 FPS (Pi 3) 8.9 FPS (Pi 4) | 30 FPS (Pi 3) | 46 FPS (Pi 3) | 30 FPS | 64 FPS | 130 FPS |
| SSD Mobilenet-V2 (300-300) | TensorFlow | 2.6 FPS (Pi 3) 4.7 FPS (Pi 4) | 3.7 FPS (Pi 3) 5.8 FPS (Pi 4) | 11 FPS (Pi 3) 41 FPS (Pi 4) | 17 FPS (Pi 3) 55 FPS (Pi 4) | - | 39 FPS | 48 FPS |
| Binary model (300x300) | XNOR | 6.8 FPS (Pi 3) 12.5 FPS (Pi 4) | - | - | - | - | - | - |
| Inception V4 (299x299) | PyTorch | - | - | - | 3 FPS (Pi 3) | - | 11 FPS | 9 FPS |
| Tiny YOLO V3 (416x416) | Darknet | 0.5 FPS (Pi 3) 1 FPS (Pi 4) | 1.1 FPS (Pi 3) 1.9 FPS (Pi 4) | - | - | 2.2 FPS | 25 FPS | - |
| OpenPose (256x256) | Caffe | - | - | 5 FPS (Pi 3) | - | - | 14 FPS | - |
| Super Resolution (481x321) | PyTorch | - | - | 0.6 FPS (Pi 3) | - | - | 15 FPS | - |
| VGG-19 (224x224) | MXNet | 0.5 FPS (Pi 3) 1 FPS (Pi 4) | - | 5 FPS | - | - | 10 FPS | - |
| Unet (1x512x512) | Caffe | - | - | 5 FPS | - | - | 18 FPS | - |

Рис. 15.12. Порівняння ефективності мікрокомп’ютерних платформ машинного навчання

## Movidius Neural Compute Stick 2 від Intel на Raspberry Pi 4

Три роки тому стартап, що називається Movidius, запустив перший у світі процесор глибокого навчання на USB-накопичувачі. Базуючись на своєму Myriad 2 Vision Processor[35] [35] (VPU), перший нейтронний обчислювач Fathom став першим у своєму роді. Але потім компанію придбав Intel і через рік Intel випустила ребрендовану версію «флешки».

Минув ще один рік, і Intel представила «флешку» другого покоління, використовуючи VPI Myriad X. На відміну від нового USB-прискорювача Google Coral, який постачається з підтримкою RPi з коробки, спочатку не було жодної підтримки для використання з архітектурами, які не є x86_64.

Ситуація змінилась в грудні 2018 р. завдяки підтримці програмного забезпечення і документації, коли було видано посібник про те, як користуватися програмою Raspbian (але початкові відгуки підказували, що процес був не дуже зручним для користувачів).

***Примітка.*** Пропонований покроковий посібник також буде корисним при налаштуванні і використанні оригінального Movidius Neural Compute Stick з RPi без змін.

Neural Compute Stick другого покоління має розміри 73 × 27 × 14 мм і вагу 35 г. Це не здається важливим допоки не прийде розуміння, що «флешка» настільки велика, що може блокувати сусідні USB-порти на RPi (рис. 15.13), а з деякими комп’ютерами її взагалі важко використовувати. Можливо знадобиться USB-концентратор, щоб працювати з NCS.

[35] https://uploads.movidius.com/1463156689-2016-04-29_VPU_ProductBrief.pdf

Рис. 15.13. Raspberry Pi 4 B з підключеним NCS

**Налаштування Raspberry Pi**

На відміну від нової плати Coral Dev Google, яка потребує чималої роботи з налаштування, перш ніж почати роботу з нею, тут все простіше. Візьмемо RPi, блок живлення, USB-кабель й мікро SD-карту – майже все готово.

Якщо використовувати RPi, то можливо доброю ідеєю буде встановити нову версію операційної системи і почати працювати з чистого аркуша. Завантажимо останню версію програми Raspbian (з робочим столом) і налаштуємо RPi.

Якщо ви не використовуєте дротову мережу або не маєте дисплея і клавіатури (приєднаних до RPi), то вам треба буде підключити RPi до своєї бездротової мережі і ввімкнути SSH. Також можна ввімкнути VNC, оскільки це може виявитися корисним.

Після встановлення RPi подаємо живлення, а потім відкриємо вікно терміналу на своєму ноутбуці і SSH в RPi:

```
ssh pi@raspberrypi.local
```

Після входу в систему можна змінити ім’я хоста на щось менш загальне за допомогою інструменту `raspi-config`, щоб відрізнити цю плату від усіх інших плат RPi у своїй мережі (наприклад, назвати `neural2`).

**Живлення Raspberry Pi**

Більш сучасні плати RPi, а особливо остання модель RPi 4 B, потребують джерела живлення USB-C, яке буде забезпечувати +5 В при струмі до 3 А. Залежно від того, які є периферійні пристрої, плата має підтримувати їх енергоспоживання, а це може бути проблемою.

Наприклад, для підключення монітора до порту HDMI необхідно додатково 50 мА, камера для модуля потребує додатково 250 мА, а клавіатура і миша можуть забирати до 100 мА або понад 1 000 мА залежно від моделі. Для самої нейронної обчислювальної «флешки» потрібно не менше 500 мА.

У разі недостатньої потужності джерела живлення RPi почне знижатися швидкість роботи процесора. Якщо все погіршиться далі, плата почне періодично і багаторазово перезавантажуватися.

Якщо до плати RPi підключено монітор, то у верхньому правому куті екрана ви побачите жовту блискавку при проблемах з живленням. Якщо ж використовуєте RPi без монітора, то перевірити живлення можна з командного рядка за допомогою vcgencmd[36] [36]:

```
vcgencmd get_throttled
```

Однак вихід у вигляді двійкових кодів, а тому дещо непрозорий. Але не складно написати сценарій, щоб проаналізувати вихід виконання команди і отримати на виході щось таке, що можна читати:

```
sh ./throttled.sh
Status: 0x50005
Undervolted:
   Now: YES
   Run: YES
Throttled:
   Now: YES
   Run: YES
Frequency Capped:
   Now: NO
   Run: NO
```

Якщо сценарій повідомляє, що плата має недостатню напругу живлення, то, як правило, слід замінити джерело живлення, перш ніж продовжити. Якщо ви не впевнені в своєму власному джерелі живлення, краще вибрати офіційний блок живлення RPi 4 USB-C (або USB для RPi 3 B+).

Офіційне джерело живлення було розроблене так, щоб забезпечувати +5,1 В, незважаючи на швидкі коливання величини струму. Коливання величини струму трапляються досить часто, якщо ви використовуєте з RPi периферійні пристрої.

**Встановлення програмного забезпечення**

Все готове до встановлення програмного забезпечення, необхідного для підтримки Neural Compute Stick. Слід ігнорувати інструкції[37] [37], на які вказувало повідомлення у вікні, вони для комп'ютерів x86_64.

Натомість Intel надала альтернативні вказівки[38] [38], тож будемо базуватися на них. Спочатку завантажимо інструментарій OpenVINO:

```
wget
https://download.01.org/opencv/2019/openvinotoolkit/l_openvino_toolkit_r
aspbi_p_2019.1.094.tgz
tar -zxvf l_openvino_toolkit_raspbi_p_2019.1.094.tgz
```

а потім змінимо сценарій налаштування, щоб відобразити шлях встановлення:

```
sed-i "s||$(pwd)/inference_engine_vpu_arm|"
inference_engine_vpu_arm/bin/setupvars.sh
```

перед тим як додавати сценарій встановлення до кінця файлу `.bashrc`

[36] https://raspberrypi.stackexchange.com/questions/85345/what-is-the-vcgencmd-command
[37] https://software.intel.com/en-us/neural-compute-stick/get-started
[38] https://software.intel.com/en-us/articles/OpenVINO-Install-RaspberryPI

```
echo "source inference_engine_vpu_arm/bin/setupvars.sh" >> .bashrc
source .bashrc
[setupvars.sh] OpenVINO environment initialized
```

В останньому випуску, яким був OpenVINO 2019R1, є проблема з конфігурацією `PYTHONPATH`. Тож треба внизу у файл `.bashrc` додати невелике доповнення до шляху, щоб подолати проблему:

```
export
PYTHONPATH="${PYTHONPATH}:/home/pi/inference_engine_vpu_arm/python/pytho
n3.5/armv7l"
```

Далі запустимо сценарій для встановлення нових `udev`-правил, щоб RPi міг розпізнавати нейронну обчислювальну «флешку», коли ми її підключаємо.

```
sudo usermod-a -G users"$(whoami)"
sh
inference_engine_vpu_arm/install_dependencies/install_NCS_udev_rules.sh
Update udev rules so that the toolkit can communicate with your neural
compute stick
[install_NCS_udev_rules.sh] udev rules installed
```

Тепер перезавантажимо RPi, щоб могли запрацювати всі зроблені зміни.

Підключимо Neural Compute Stick.

Перевіряючи з командою `dmesg`, маємо побачити щось подібне:

```
[ 1491.382860] usb 1-1.2: new high-speed USB device number 5 using
dwc_otg
[ 1491.513491] usb 1-1.2: New USB device found, idVendor=03e7,
idProduct=2485
[ 1491.513504] usb 1-1.2: New USB device strings: Mfr=1, Product=2,
SerialNumber=3
[ 1491.513513] usb 1-1.2: Product: Movidius MyriadX
[ 1491.513522] usb 1-1.2: Manufacturer: Movidius Ltd.
[ 1491.513530] usb 1-1.2: SerialNumber: 03e72485
```

Якщо не видно подібних повідомлень, «флешка» не розпізнана. Спробуємо перезавантажити RPi і перевірити ще раз:

```
dmesg | grep Movidius
[    2.062235] usb 1-1.2: Product: Movidius MyriadX
[    2.062244] usb 1-1.2: Manufacturer: Movidius Ltd.
```

Отже, треба, щоб «флешка» була виявлена.

**Запуск першої моделі машинного навчання**

На відміну від новітнього обладнання Edge TPU на базі Coral від Google, або нового Jetson Nano від NVIDIA, робота з Neural Compute Stick побудована на C++, а не на Python. Однак нам бракує деяких інструментів, щоб почати роботу, тому насамперед треба встановити `cmake`.

```
sudo apt-get install cmake
```

Тепер можемо створити одну з попередньо підготовлених демонстрацій розпізнавання облич.

```
cd inference_engine_vpu_arm/deployment_tools/inference_engine/samples
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS="-march=armv7-a"
make-j2 object_detection_sample_ssd
   .
[100%] Linking CXX executable
../armv7l/Release/object_detection_sample_ssd
[100%] Built target object_detection_sample_ssd
```

На відміну від дистрибутива Intel, у версії інструментарію RPi немає моделі і асоційованого XML-файлу з топологією моделі. Отже, перед тим як запустити модель, треба завантажити обидва ці файли.

```
wget --no-check-certificate
https://download.01.org/openvinotoolkit/2018_R5/open_model_zoo/face-
detection-adas-0001/FP16/face-detection-adas-0001.bin
wget --no-check-certificate
https://download.01.org/openvinotoolkit/2018_R5/open_model_zoo/face-
detection-adas-0001/FP16/face-detection-adas-0001.xml
```

Нам знадобиться зображення для запуску демонстрації обличчя. Автор скопіював зображення в домашній каталог на RPi з ноутбука, використовуючи `scp`.

Тепер можемо запустити демонстрацію:

```
./armv7l/Release/object_detection_sample_ssd -m face-detection-adas-
0001.xml -d MYRIAD -i ~/me.jpg
[ INFO ] InferenceEngine:
API version ............ 1.4
Build .................. 19154
Parsing input parameters
[ INFO ] Files were added: 1
[ INFO ]     /home/pi/me.jpg
[ INFO ] Loading plugin
API version ............ 1.5
Build .................. 19154
Description ....... myriadPlugin
[ INFO ] Loading network files:
face-detection-adas-0001.xml
face-detection-adas-0001.bin
[ INFO ] Preparing input blobs
[ INFO ] Batch size is 1
[ INFO ] Preparing output blobs
[ INFO ] Loading model to the plugin
[ WARNING ] Image is resized from (960, 960) to (672, 384)
[ INFO ] Batch size is 1
[ INFO ] Start inference (1 iterations)
[ INFO ] Processing output blobs
[0,1] element, prob = 1    (410.391,63.5742)-(525.469,225.703) batch id
: 0 WILL BE PRINTED!   .
   .
   .
[ INFO ] Image out_0.bmp created!
```

```
total inference time: 155.233
Average running time of one iteration: 155.233 ms
Throughput: 6.44194 FPS
[ INFO ] Execution successful
```

Під час опрацювання зображення можна побачити низку виявлень: серед введених зображень буде одне виявлене обличчя з ймовірністю '1', а потім понад сто інших виявлень (всі з ймовірністю менше '0,03'), які кодом були визнані незначними.

Автоматично згенерується вихідний BMP-файл, коли всі значні виявлення матимуть обмежувальне поле навколо об’єкта. На цей момент принаймні будемо знати, що все працює.

Але, переглядаючи в каталозі `inference_engine/samples/object_detection_sample_ssd` код `object_detection_sample_ssd` (на C++), побачимо, що для роботи в обгортці Python знадобляться ще й інші можливості, окрім тих, що використовуються для роботи з машинним навчанням на мовах високого рівня.

**Додавання камери**

Використаємо модуль камери RPi для наступної демонстрації, а отже, перед тим як розпочати роботу з обгортками Python, переконаємося, що камера встановлена і працює.

Якщо RPi увімкнений, то треба його вимкнути, перш ніж приєднувати модуль камери. В сеансі SSH слід вимкнути RPi, використовуючи команду вимкнення, щоб зупинка була чистою (без залишених на картці тимчасових файлів):

```
sudo shutdown -h now
```

Відключаємо кабель живлення і під’єднуємо камеру до плати. Після цього вмикаємо плату і знову увійдемо на неї через SSH.

Тепер у нас фізично підключена камера, її потрібно ввімкнути. Для цього можна використовувати утиліту `raspi-config`.

```
sudo raspi-config
```

Прокрутимо вниз і виберемо «*Interfacing Options»*, а потім – «*Camera»* у наступному меню. Коли з’явиться запит, натискаємо «*Yes*», а потім «*Finish*», щоб повністю вийти з інструменту конфігурації. Вибераємо «*Yes»*, коли запитають, чи треба перезавантажуватись.

Можемо перевірити, чи працює камера, скориставшись командою `raspistill`:

```
raspistill -o testshot.jpg
```

Це дасть змогу зберегти файл з назвою `testshot.jpg` у домашньому каталозі, який можна скопіювати, використовуючи `scp`, з RPi на свій ноутбук.

Ми можемо використовувати фотознімки, зняті камерою, і подавати їх вручну до нашої моделі, якщо підключений монітор. В комплект програмного забезпечення для розроблення входить деякий код, який можна запустити, щоб демонструвати вгорі екрана реалізацію в реальному часі відеоканалу з камери.

Сценарію потрібно буде отримати доступ до камери з Python, але модуль `picamera` Python може бути не встановлено на RPi. Отже, перед тим як запустити демокод, нам треба спочатку це зробити з командою:

```
sudo apt-get install python3-picamera
```

Якщо необхідно підтримувати сумісний з Video4Linux (V4L) пристрій, то слід завантажити відповідний модуль ядра BCM2835:

```
sudo modprobe bcm2835-v4l2
```

для створення сумісного з V4L `/dev/video0` пристрою. Також можна додати модуль BCM2835 в `/etc/modules`,

```
sudo -i
# echo 'bcm2835-v4l2' >> /etc/modules
# exit
```

щоб модуль завантажувався при перезавантаженні.

**Запуск машинного навчання в Python**

Обгортки Python для інструментарію Intel OpenVINO потребують підтримки NumPy і OpenCV, а отже, перш ніж щось робити інше, треба встановити обидва ці пакети для Python 3.7.

```
sudo apt-get install python3-numpy
pip3 install opencv-python
```

Також встановимо деякі інші додаткові бібліотеки, необхідні для коду:

```
sudo apt-get install libgtk-3-dev
sudo apt-get install libavcodec-dev
sudo apt-get install libavformat-dev
sudo apt-get install libswscale-dev
```

Використаємо модифіковану версію демо-сценарію `object_detection_demo_ssd_async.py`, оскільки в основній версії виникають проблеми під час відкриття потоку з камери.

Відкриваємо сценарій у своєму улюбленому редакторі:

```
cd
~/inference_engine_vpu_arm/deployment_tools/inference_engine/samples/pyt
hon_sample/object_detection_demo_ssd_async
sudo nano object_detection_demo_ssd_async.py
```

Тепер змінимо рядки з 83 до 85,

```
if args.input == 'cam':
        input_stream = 0
    else:
```

щоб мати повний шлях до нашого V4L пристрою.

```
if args.input == 'cam':
        input_stream = '/dev/video0'
    else:
```

По-перше, це зупинить зависання сценарію з помилкою на конвеєрі, по-друге, уникнемо сумнозвісної помилки "`Trying to dispose element pipeline0, but`

`it is in PAUSED instead of the NULL state"`, яка виникає при спробі виходу зі сценарію, після того як він зависне.

Можна повторно використовувати ту ж модель, яку ми завантажили для попереднього прикладу. Отже, продовжимо і збережемо копію нового сценарію в домашньому каталозі. Можемо налаштувати його:

```
cd
~/inference_engine_vpu_arm/deployment_tools/inference_engine/samples/pyt
hon_sample/object_detection_demo_ssd_async
python3 ./object_detection_demo_ssd_async.py -m ../../build/face-
detection-adas-0001.xml  -i cam -d MYRIAD -pt 0.5
```

Якщо все правильно, то можна побачити зверху на робочому столі вікно, яке відкривається з відеоканалу модуля `picamera`, з накладенням у реальному часі.

Це добре видно на основному дисплеї RPi – вікно просто відкривається.

Якщо ж ми працюємо без монітора, найпростіше – увімкнути VNC і таким способом підключитися до свого RPi. Однак слід зауважити, що якщо немає підключеного монітора, треба встановити нову роздільну здатність за допомогою `raspi-config`, оскільки дисплей за замовчуванням має роздільну здатність всього 720 × 480 пікселів. Перейдемо у розділ `Advanced Options`, а потім `Resolution` і виберемо роздільну здатність дисплея, яка підходить для дисплея локальної машини.

При підключенні до RPi з локальної машини ми зможемо відкрити вікно, яке відображатиметься на локальному робочому столі, якщо запущений X-сервер і увімкнена переадресація X11 на нашу локальну машину.

```
ssh -XY pi@neural.local
password:
```

Інакше сценарій закриється з помилкою `cannot open display`.

**Висновки**

На сайті розробника є декілька версій документації, і вони часто суперечать одна одній. Також наявна низка (хоча і незначних) помилок у демонстраційному коді Intel. Сценарії налаштування, які встановлюють попередньо складені моделі, мабуть відсутні у дистрибутиві.

Інструкції щодо запуску Compute Stick не зовсім зручні для розробників. Пакувальники Python також мають низький рівень абстрагування.

Отже, для перетворення власних моделей у необхідний формат для використання з Compute Stick на RPi треба спочатку встановити OpenVINO в іншому вікні Linux на базі x86.

Загалом розробник на RPi відчуває незручності скороченої версії повного набору інструментів OpenVINO. Хоча останні матеріали, викладені на сайті [38], досить детально описують процес встановлення OpenVINO на ОС Raspbian, але вони перевірені лише з 32-розрядною ОС і лише на Raspberry Pi 3B.

Сподіваюсь, що вказані недоліки і проблеми вас не розчарують.

## Запуск моделі Keras на Movidius Neural Compute Stick 2

Нейронна обчислювальна флешка Movidius (Neural Compute Stick – NCS) разом з іншими апаратними засобами поступово привносять глибоке навчання в пристрої Інтернету, які мають обмежені ресурси.

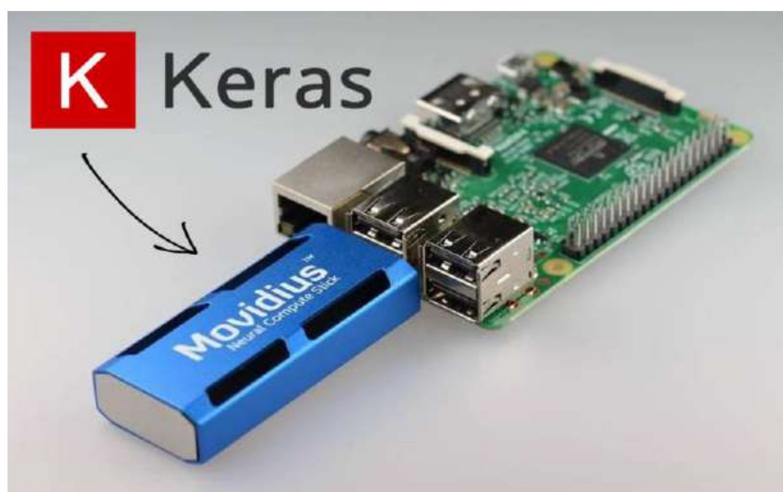



Рис.15.14. Movidius з Raspberry Pi

Розглянемо, як навчити просту модель MNIST Keras і розгорнути її на NCS, яку також можна підключити до ПК або до RPi[39] [39].

Треба виконати кілька кроків:

1. Тренування моделі в Keras (з бекендом TF).
2. Збереження файлу моделі і вагових коефіцієнтів у Keras.
3. Перетворення моделі Keras для TF.
4. Компіляція моделі TF для графа NCS.
5. Розгортання і запуск графа на NCS.

Розглянемо кожен з кроків.

Вихідний код для цієї публікації доступний у репозиторії GitHub[40] [40].

### Тренування і збереження моделі Keras

Так само, як «Hello world!» є першим виведенням на консолі початківця, так для програмістів з глибокого навчання еквівалентне навчання розпізнаванню рукописних цифр у моделі MNIST.

Модель Keras добре виконує цю роботу з декількома згортковими шарами з подальшим кінцевим етапом виходу. Повний код train-mnist.py перебуває на GitHub[41] [41], а короткий фрагмент коду наведено нижче, щоб показати суть методу.

```
from keras import layers, models

model = models.Sequential()
model.add(layers.Conv2D(16, 3, activation='relu', input_shape=(28, 28,
1)))
model.add(layers.MaxPool2D())
model.add(layers.Conv2D(32, 3, activation='relu'))
```

[39] https://www.dlology.com/blog/how-to-run-keras-model-on-movidius-neural-compute-stick/
[40] https://github.com/Tony607/keras_mnist
[41] https://github.com/Tony607/keras_mnist/blob/master/train-minst.py

```
model.add(layers.MaxPool2D())
model.add(layers.Conv2D(64, 3, activation='relu'))
model.add(layers.MaxPool2D())
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))
model.summary()

model.compile(optimizer='adam', metrics=['accuracy'],
loss='categorical_crossentropy')

history = model.fit(x_train, y_train, epochs=2, batch_size=128)

output = model.predict(test_image.reshape(1, 28, 28, 1))[0]
print("Keras \r\n", output, '\r\nPredicted:',output.argmax())
```

Тренування може тривати 3 хв на GPU або довше на CPU. Якщо немає тренувальної машини для GPU, то можна ознайомитись з підручником[42] [42], де викладено як безкоштовно тренувати свою модель в GPU Google. Все, що для цього треба мати – це обліковий запис Gmail.

У будь-якому випадку, після тренувань зберігаємо модель і вагові коефіцієнти в два окремі файли:

```
with open("model.json", "w") as file:
    file.write(model.to_json())
model.save_weights("weights.h5")
```

Крім того, можемо викликати `model.save('model.h5', include_optimizer=False)`, щоб зберегти модель в одному файлі, зауваживши, що виключаємо оптимізатор, встановивши `include_optimizer` значення `False`, оскільки оптимізатор використовується лише для тренування.

**Перетворення моделі Keras для TensorFlow**

Оскільки Movidius NCSDK2 компілює лише модель TF або Caffe, знімаємо прив’язку Keras до графа TF. Код нижче виконує цю роботу. Розглянемо, як він працює, щоб налаштувати його у разі потреби.

```
from keras.models import model_from_json
from keras import backend as K
import tensorflow as tf

model_file = "model.json"
weights_file = "weights.h5"

with open(model_file, "r") as file:
    config = file.read()

K.set_learning_phase(0)
model = model_from_json(config)
model.load_weights(weights_file)
```

[42] https://www.dlology.com/blog/how-to-run-object-detection-and-segmentation-on-video-fast-for-free/

```
saver = tf.train.Saver()
sess = K.get_session()
saver.save(sess, "./TF_Model/tf_model")

fw = tf.summary.FileWriter('logs', sess.graph)
fw.close()
```

Спочатку вимикаємо фазу навчання, а потім завантажуємо модель стандартним способом Keras з двох окремих файлів, які зберегли раніше.

Викликавши `K.get_session()` з Keras із бекендом TF, робимо доступним за замовчуванням сеанс TF. Можемо побачити більше (що є всередині графа TF), викликавши `sess.graph.get_operations()`, який повертає список операцій TF у нашій моделі. Це може бути корисним для пошуку операцій, які не підтримуються NCS, а отже, зможемо відстежити їх зі списку. Нарешті, клас TF `Saver` зберігає модель у чотири файли за вказаним шляхом.

Кожен файл виконує різну задачу:

1. **checkpoint** визначає шлях точки перевірки моделі, яка в нашому випадку є "tf_model".
2. **.meta** зберігає структуру графа.
3. **.data** зберігає значення кожної змінної у графі.
4. **.index** ідентифікує точку перевірки.

**Компіляція моделі TensorFlow з mvNCCompile**

Інструмент командного рядка mvNCCompile постачається з інструментарієм NCSDK2, що перетворює мережі Caffe або TF у файли графа, які можуть використовуватися API Movidius Neural Compute Platform. Визначаємо вузли вводу і виводу як імена операцій TF для mvNCCompile під час генерації графів. Знайти список операцій TF можна, викликавши `sess.graph.get_operations()`, як показано вище. У нашому випадку знаходимо '**conv2d_1_input**' як вхідний вузол, а '**dense_2/Softmax**' як вихідний вузол.

Нарешті, команда компіляції буде виглядати приблизно так:

```
mvNCCompile TF_Model/tf_model.meta -in=conv2d_1_input -
on=dense_2/Softmax
```

Файл графа з назвою «graph» за замовчуванням буде створений у поточному каталозі.

**Розгортання графа і виконання передбачення**

API Python NCSDK2 знаходить пристрій NCS, підключається, виділяє граф у пам’яті і робить передбачення.

Наступний код показує суттєву частину, а `input_img` є попередньо опрацьованим зображенням як масив numpy форми (28, 28). Вихід такий самий, як у Keras: десять чисел, що представляють ймовірності класифікації для кожної з десяти цифр. Застосовуємо функцію `argmax`, щоб знайти індекс найбільш ймовірного передбачення.

```
from mvnc import mvncapi as mvnc
# Отримуємо перший пристрій NCS за його ім’ям. Для цієї програми будемо
завжди відкривати першим пристрій NCS.
devices = mvnc.enumerate_devices()
# Отримуємо перший пристрій NCS за його ім’ям. Для цієї програми будемо
завжди відкривати першим пристрій NCS.
dev = mvnc.Device(devices[0])
```

```
# Читаємо компільований мережевий граф з файла (вкажіть правильно graph_filepath для вашого файла графа)
with open("graph", mode='rb') as f:
    graphFileBuff = f.read()

graph = mvnc.Graph('graph1')

# Виділяємо граф на пристрої і створюємо fifo входу і виходу.
in_fifo, out_fifo = graph.allocate_with_fifos(dev, graphFileBuff)

# Записуємо вхід в буфер input_fifo і черга висновку в одному виклику
graph.queue_inference_with_fifo_elem(in_fifo, out_fifo, input_img.astype('float32'), 'user object')

# Читаємо результат на виході fifo
output, userobj = out_fifo.read_elem()
print('Predicted:',output.argmax())
```

Тепер модель Keras працює на NCS! Можемо запустити її на одноплатному комп'ютері RPi замість ПК з Ubuntu.

**Передбачення за допомогою відеопотоку вебкамери на Raspberry Pi**

Встановлення NCSDK2 на RPi може тривати десятки хвилин, що є не дуже добра новина для нетерплячих. Однак можна вибрати встановлення на RPi лише суттєвої частини NCSDK2, щоб виконати роботу з графом, зкомпільованим на ПК з Ubuntu.

По-перше, замість клонування сховища NCSDK2 на RPi, що може тривати багато часу, завантажимо zip-файл версії NCSDK2[43] [43]. Це може зберегти значний обсяг диска, оскільки всі файли контролю версій git пропускаються.

По-друге, пропустимо встановлення TF і Caffe під час встановлення NCSDK2, змінивши файл `ncsdk.conf`:

```
INSTALL_CAFFE=no
INSTALL_TENSORFLOW=no
```

Для запуску відеопотоку вебкамери потрібно встановити OpenCV 3, запустивши чотири рядки в терміналі RPi:

```
sudo pip3 install opencv-python==3.3.0.10
sudo apt-get update
sudo apt-get install libqtgui4
sudo apt-get install python-opencv
```

Як тільки встановлення NCSDK2 і OpenCV 3 завершиться, скопіюємо створений раніше **graph** файл на свій RPi. Зауважимо, що ми пропустили досить багато матеріалів, тож команди **mvNC**** не будуть працювати на нашому RPi, оскільки вони залежать від встановлення Caffe і TF.

Модель MNIST була навчена розпізнавати рукописні цифри білого кольору на чорному тлі зображення в градаціях сірого з роздільною здатністю 28 × 28, тому для перетворення захопленого зображення потрібен певний етап попереднього опрацювання.

[43] https://ncs-forum-uploads.s3.amazonaws.com/ncsdk/ncsdk-02_05_00_02-full/ncsdk-2.05.00.02.tar.gz

1. Обріжемо центральну ділянку зображення.
2. Знайдемо краї зображення, цей крок також перетворить зображення в градації сірого.
4. Розширимо краї, щоб зробити краї товщими для заповнення простору між двома близькими паралельними краями.
5. Змінимо розмір зображення на 28 × 28.

Подібну реалізацію з Python OpenCV 3 можна знайти у файлі ImageProcessor.py[44] [44]. Після завершення демонстрації вебкамери кожен знятий кадр передається у функцію попереднього опрацювання зображень, потім подається на граф NCS, який повертає остаточні ймовірності передбачення, як і раніше. Звідти представляємо кінцевий передбачуваний результат, як накладення зображення, що відображається на дисплеї (рис.15.15).

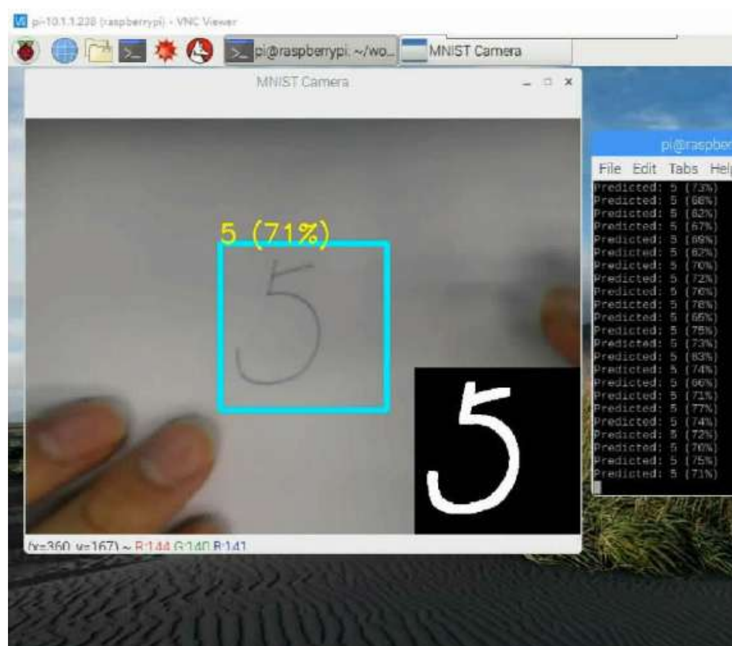



Рис. 15.15. Результат передбачення на дисплеї

**Висновки**

Ми розгорнули модель Keras на NCS. Оскільки NCS побудований як «блок опрацювання зображень», він підтримує згорткові шари разом з деякими іншими, а періодичні шари нейронної мережі, такі як LSTM і GRU, на NCS можуть не працювати.

У нашому демонстраційному прикладі ми вказали mvNCCompile взяти вихідний вузол остаточної класифікації, а можна використати проміжний рівень як вихідний вузол. В цьому сенсі використано модель як екстрактор функцій, тобто аналогічно тому, як працює демонстрація NCS faceNet для перевірки обличчя.

## Автономний робот-танк

Дистанційне керування різними роботизованими платформами – це ще не повноцінні роботи, оскільки керує таким пристроєм людина. Для того щоб навчити робота самостійно рухатися, визначати перешкоди, їх оминати, необхідно додати йому деякий штучний інтелект. Одним з таких варіантів є проєкт робота-танка на RPi[45] [45]. Для реалізації проєкту автору довелося:

[44] https://github.com/Tony607/keras_mnist/blob/master/ImageProcessor.py
[45] https://habr.com/ru/post/459126/

- навчити власну мережу E-net під необхідний розмір зображень;
- передати запуск нейромережі із самої RPi на спеціальний пристрій розширення – вищерозглянутий найчастіше використовуваний Intel Movidius (NCS).

## Intel NCS

Ви вже знаєте, що в другій версії NCS пропонується фреймворк OpenVino, до складу якого входить OpenCV 4.1 і багато іншого (у т. ч. різні інструменти для роботи з нейромережами). Intel з самого початку зробив підтримку операційної системи Raspbian.

Автор використовував Keras, підтримуваного TF, для тренування мережі. Хоча RPi може використовувати цю мережу, переконайтеся, що наявні версії TF для RPi і TF для Keras сумісні. Встановлювати Keras на RPi не обов'язково, але сам TF містить адаптер для читання мережевого графа.

NCS підтримує лише свій власний формат нейромереж, а Intel надає інструмент Model Optimizer у складі OpenVino для конвертації графів найбільш популярних фреймворків: TF, Caffe, Torch.

Крім цього, Intel також надає model zoo[46] [46] — набір готових моделей для багатьох реальних випадків. Серед цих моделей є і моделі для дорожньої сегментації.

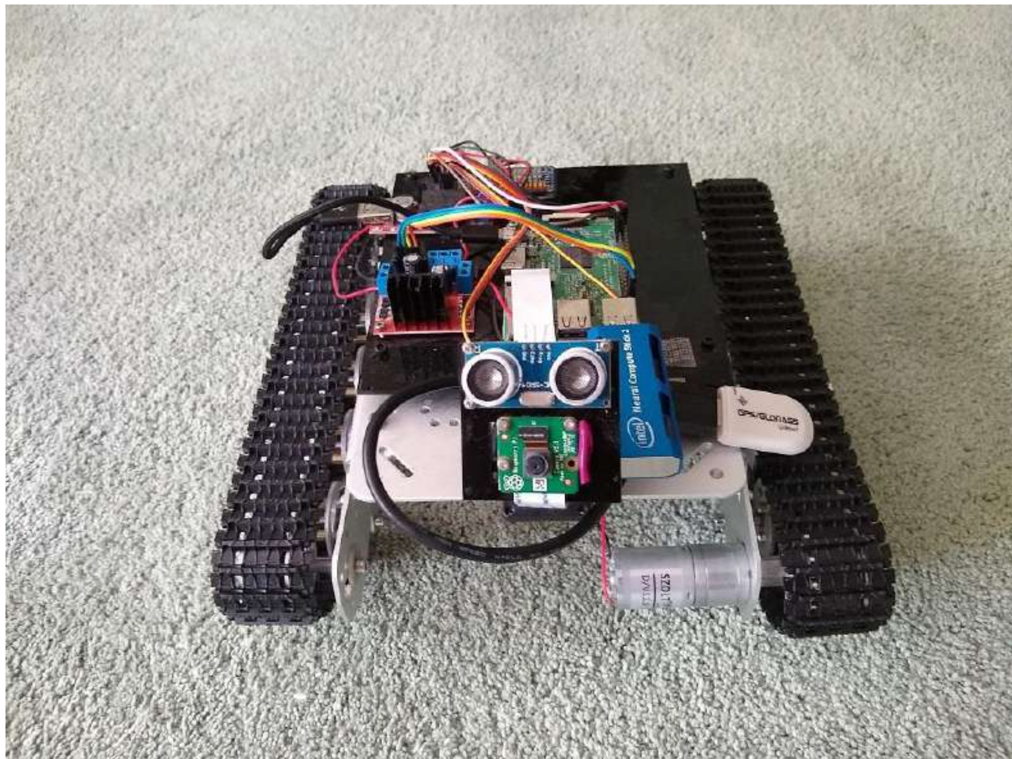


Рис. 15.16. Автономний робот з дистанційним керуванням

## Нейромережі на NCS

Для того щоб запустити нейромережу на «флешці», треба зробити кілька кроків.

### Крок 1. Ініціалізація пристрою

Назва пристрою MYRIAD повторює модель процесора «флешки», а при завантаженні бібліотеки треба вказати шлях до нього:

[46] https://software.intel.com/en-us/openvino-toolkit/documentation/pretrained-models

```
from openvino.inference_engine import IENetwork, IEPlugin
ncs_plugin = IEPlugin(device="MYRIAD",
plugin_dirs="/opt/intel/openvino/inference_engine/lib/armv7l")
```

**Крок 2. Завантаження моделі**

Модель треба завантажити на пристрій. Завантажується модель лише один раз і можна завантажити кілька моделей.

```
model = IENetwork(model=xml_path, weights=bin_path)
net = ncs_plugin.load(network=model)
```

**Крок 3. Запуск розрахунку**

Тепер модель можна використати:

```
input_blob = next(iter(model.inputs))
out_blob = next(iter(model.outputs))
n, c, h, w = model.inputs[input_blob].shape
images = np.ndarray(shape=(n, c, h, w))
images[0] = image
res = net.infer(inputs={input_blob: images})
res = res[out_blob]
```

**Класифікація напрямків**

Для прийняття рішення про напрямки руху танк використовує просту нейромережу multi-category classification, яка дає цілком прийнятні результати.

Нейромережа навчена на Keras і працює на RPi через TF, який має вбудований адаптер для цього формату. Модель[47] [47] проста і навіть на RPi 3 показує прийнятні за швидкістю результати (0.35 с на зображення).

У репозиторії викладено код і дані для тренування нейронної мережі, щоб вирішити в якому напрямку (вперед, ліворуч, праворуч) робот має рухатися. Все було розроблено для pitanq – робота, керованого RPi, що працює і на інших подібних транспортних засобах.

**Вміст репозиторія**

- Data – дані, які використовувалися для тренування моделі.
- Models – тренована модель у форматах Keras, Tensorlow і OpenVino.
- Train.py – тренування мережі Keras з використанням даних з набору у папці даних.
- Check_nn.py – класифікація простого зображення або папки за допомогою тренованої моделі.
- Keras2tf.py – перетворення моделі з оригінального формату Keras у формат TF.
- Check_tf.py – перевірка перетворення TF через класифікацію всіх даних проти перетвореної моделі.

**Дані**

Нейронна мережа – це багатозначний класифікатор, який тренувався з трьома класами зображень: вперед, вліво і вправо. Дані отримували шляхом розпізнавання дороги на фотографіях, потім фотографії зменшувались, а їхні центри 64 × 64 були взяті як вхід для цієї мережі. Zip-архів із початковими зображеннями є у вказаному репозиторії.

[47] https://github.com/tprlab/pitanq-selfwalk

Зображення вперед мають подібний вигляд.

**Вперед:**

Особливістю руху вперед є кут правого краю близько 45 градусів. Ось як нормально виглядає зображення, якщо робот рухається по правому краю дороги.

**Ліворуч:**

Коли робот заходить занадто багато вправо, то правий край починає нахилятися вліво. Це ключова особливість фотографій для руху ліворуч. Іноді вони можуть виглядати схожими на фотографії для руху вперед.

**Праворуч:**

Коли робот заходить занадто далеко ліворуч, то правий край зникає із зображення. Отже, якщо явного краю немає – пора повертати праворуч.

**Код**

Використана найпростіша архітектура для класифікації кількох міток. Виявилося, що вона працює доволі добре.

```
def createModel(input_shape, cls_n):
    model = Sequential()

    activation = "relu"

    model.add(Conv2D(20, 5, padding="same", input_shape=input_shape))
    model.add(Activation(activation))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Визначаємо друге налаштування шарів CONV => ACTIVATION => POOL
layers
    model.add(Conv2D(50, 5, padding="same"))
    model.add(Activation(activation))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Визначаємо перші шари FC => ACTIVATION
    model.add(Flatten())
    model.add(Dense(500))
    model.add(Activation(activation))

    # Визначаємо другий шар FC
    model.add(Dense(cls_n))

    opt = SGD(lr=0.01)
    # Нарешті, визначаємо класифікатор soft-max
```

```
    model.add(Activation("softmax"))

    model.compile(loss="categorical_crossentropy", optimizer=opt,
metrics=["accuracy"])
    return model
```

**Перетворення у формат Tensorflow**

Модель необхідно конвертувати у формат TF. Код перебуває у файлі keras2tf.py:

```
import tensorflow as tf
from tensorflow.python.framework.graph_util import
convert_variables_to_constants

from keras import backend as K
from keras.models import load_model
from keras.models import model_from_json


def load_keras_model(json_file, model_file):
    jf = open(json_file, 'r')
    loaded_model_json = jf.read()
    jf.close()
    loaded_model = model_from_json(loaded_model_json)
    loaded_model.load_weights(model_file)
    return loaded_model


def freeze_session(session, keep_var_names=None, output_names=None,
clear_devices=True):
    graph = session.graph
    with graph.as_default():
        freeze_var_names = list(set(v.op.name for v in
tf.global_variables()).difference(keep_var_names or []))
        output_names = output_names or []
        output_names += [v.op.name for v in tf.global_variables()]
        # Graph -> GraphDef ProtoBuf
        input_graph_def = graph.as_graph_def()
        if clear_devices:
            for node in input_graph_def.node:
                node.device = ""
        frozen_graph = convert_variables_to_constants(session,
input_graph_def,
                  output_names, freeze_var_names)
        return frozen_graph


model = load_keras_model('./model.json', './model.h5')
frozen_graph = freeze_session(K.get_session(),
                              output_names=[out.op.name for out in
model.outputs])

tf.train.write_graph(frozen_graph, ".", "ktf_model.pb", as_text=False)
```

Конвертована модель також перебуває в репозиторії: model/ktf_model.pb

**OpenVino**

Далі треба конвертувати модель TF у формат OpenVino, щоб запустити з Intel NCS:

```
python mo_tf.py --input_model "model/ktf_model.pb" --log_level=DEBUG -b1
--data_type FP16
```

У моделі OpenVino два файли: `model/ktf_model.xml` і `model/ktf_model.bin`.

Тести засвідчили, що класифікація зображення відбувається за 0.007 с. Це дуже добрий результат.

**Розпізнавання об’єктів**

Задача сегментації – не єдина, яку доводиться розв’язувати роботу.

У model_zoo від Intel достатньо детекторів більш вузької специфіки на базі MobileSSD, але точний аналог відсутній. Проте ця мережа вказана як сумісна в списку підтримуваних моделей TF[48] [48].

На момент експерименту було вказано версію MobileSSD 2018_01_28, але читати цю модель OpenCV відмовляється:

```
cv2.error: OpenCV(4.1.0-openvino)
/home/jenkins/workspace/OpenCV/OpenVINO/build/opencv/modules/dnn/src/ten
sorflow/tf_importer.cpp:530:
error: (-2:Unspecified error) Const input blob for weights not found in
function 'getConstBlob'
```

При цьому перетворення в OpenVino здійснується успішно.

Якщо ж спробувати конвертувати версію Mobile SSD, сумісну з OpenCV-DNN (11_06_2017), то отримаємо:

```
[E0919 main.py:317] Unexpected exception happened during extracting
attributes for node
FeatureExtractor/MobilenetV1/Conv2d_13_pointwise_1_Conv2d_2_1x1_256/Relu
6

Original exception message: operands could not be broadcast together
with remapped shapes [original->remapped]: (0,) and
 requested shape (1,0,10,256)
```

Отже, технічно OpenVino і OpenCV-DNN разом, але несумісні за версіями використаних нейромереж.

Порівняння швидкостей переконливо на користь NCS: 0.1 с проти 1.7 с.

**Класифікація зображень**

Танк вміє класифікувати зображення за допомогою TF, використовуючи Inception на Imagenet.

Вже є чотири версії Inception і всі вони підтримуються в OpenVino. Завантажуємо останню четверту версію.

---

[48] https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html

Виконуємо класифікацію зображення кота і ноута на столі.
Запам'ятовуємо результати роботи поточної версії:

- laptop, laptop computer 62%;
- notebook, notebook computer 11%;
- 13 секунд.

Тепер читаємо інструкцію[49] [49], як конвертувати Inception в OpenVino.
Конвертація завершується успішно, запускаємо класифікатор на NCS:

- laptop, laptop computer 85%;
- notebook, notebook computer 8%;
- 0.2 секунди.

**Висновки**

Отже, всі сценарії, які вимагали TF, були відтворені за допомогою NCS, а це означає, що від використання TF на самому RPi можна відмовитися. Цей фреймворк дещо важкуватий для RPi 3. Швидкість, з якою NCS опрацьовує нейромережі, дає змогу розширити горизонти його застосування.

Завдання семантичної сегментації і класифікації робот вже вирішує, але є й інші задачі типу об'єктної сегментації або передачі відео з визначеними об'єктами в реальному часі. Про це не можна було й подумати на «голій» RPi.

## Класифікатор зображень на Raspberry Pi з Intel Movidius

### Що таке класифікація зображень

Класифікація зображень – це проблема комп'ютерного зору, яка має на меті класифікувати предмет або об'єкт, наявний у зображенні, для попередньо визначених класів. Типовим у реальному прикладі класифікації зображень є показ флеш-карти дитині з проханням розпізнати об'єкт, надрукований на картці. Традиційні підходи до надання такого візуального сприйняття машинами покладаються на складні комп'ютерні алгоритми, які використовують такі дескриптори функцій, як краї, кути, кольори тощо для ідентифікації або розпізнавання об'єктів на зображенні.

Глибоке навчання потребує досить цікавого і найбільш ефективного підходу до розв'язання проблем із зображенням у реальному світі. Воно використовує кілька шарів взаємопов'язаних нейронів, де кожен шар застосовує певний комп'ютерний алгоритм для ідентифікації і класифікації конкретного дескриптора.

Наприклад, якщо необхідно класифікувати знак зупинки руху, то використовуємо глибоку нейронну мережу (DNN), яка має один шар для виявлення країв і меж знаку, інший шар для виявлення кількості кутів, наступний шар для виявлення червоного кольору, наступний – для виявлення білої рамки навколо червоного тощо. Здатність DNN розбивати завдання на багато шарів простих алгоритмів допомагає йому працювати з більшим набором дескрипторів, що робить опрацювання зображень на основі DNN набагато ефективнішими в реальних програмах.

Класифікація зображень відрізняється від виявлення об'єктів. Класифікація передбачає, що в усьому зображенні є лише один об'єкт, подібний до прикладу «флеш-карти для малюків», про який згадувалося вище. Також виявлення об'єктів може опрацьовувати декілька об'єктів у межах одного зображення. Воно також може повідомити про розташування об'єкта в межах зображення.

---

[49] https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_tf_specific_Convert_Slim_Library_Models.html

Апаратне забезпечення:
- Raspberry Pi 3B або 3B+ і microSD-карта 16 ГБ або більше;
- Intel Movidius (NCStick).

Програмне забезпечення:
- Python 3.x;
- OpenCV 3.3.0;
- NCSDK 2.xx);
- Ncappzoo.

**Класифікатор зображень**

Виконаємо класифікацію зображень за допомогою глибоких нейронних мереж (DNN) на Intel®Movidius™ Neural Compute Stick (NCS). У блозі розробників NCS є покроковий посібник[50] [50] щодо побудови цього проєкту, а також детальне пояснення вихідного коду.

Змінимо директорію:

```
cd ncappzoo/apps/image-classifier/

pi@raspberrypi:~/ncappzoo/apps/image-classifier $
```

Перевіряємо файли в директорії:

```
ls
AUTHORS image-classifier.py Makefile README.md screen_shot.jpg
```

Отримуємо довідку:

```
make help
```

Можливі команди make:

make help – показує це повідомлення;
make – будує всі залежності, але не запускає цю програму;
make run – запускає програму;
make clean – видаляє всі файли і директорії, створені в цій директорії.

Будуємо всі залежності:

```
make
```

```
\making ilsvrc12
(cd ../../data/ilsvrc12; make;)
make[1]: Entering directory '/home/pi/ncappzoo/data/ilsvrc12'
make[1]: Leaving directory '/home/pi/ncappzoo/data/ilsvrc12'

making googlenet
(cd ../../caffe/GoogLeNet; make compile;)
make[1]: Entering directory '/home/pi/ncappzoo/caffe/GoogLeNet'
making prereqs
(cd ../../data/ilsvrc12; make)
```

[50] https://movidius.github.io/blog/ncs-image-classifier/

```
make[2]: Entering directory '/home/pi/ncappzoo/data/ilsvrc12'
make[2]: Leaving directory '/home/pi/ncappzoo/data/ilsvrc12'

making prototxt
Prototxt file already exists

making caffemodel
caffemodel file already exists

making compile
mvNCCompile -w bvlc_googlenet.caffemodel -s 12 deploy.prototxt
```

**mvNCCompile v02.00, Copyright @ Intel Corporation 2017**

```
Layer  inception_3b/1x1  forced to im2col_v2, because its output is used in concat
/usr/local/bin/ncsdk/Controllers/FileIO.py:65: UserWarning: You are using a large type. Consider reducing your data sizes for best performance

Blob generated
make[1]: Leaving directory '/home/pi/ncappzoo/caffe/GoogLeNet'
```

Запускаємо код Python:

```
python3 image-classifier.py
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: UserWarning: The default mode, 'constant', will be changed to 'reflect' in skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: UserWarning: Anti-aliasing will be enabled by default in skimage 0.15 to avoid aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
/usr/local/lib/python3.5/dist-packages/mvnc/mvncapi.py:418: DeprecationWarning: The binary mode of fromstring is deprecated, as it behaves surprisingly on unicode inputs. Use frombuffer instead
  tensor = numpy.fromstring(tensor.raw, dtype=numpy.float32)
```

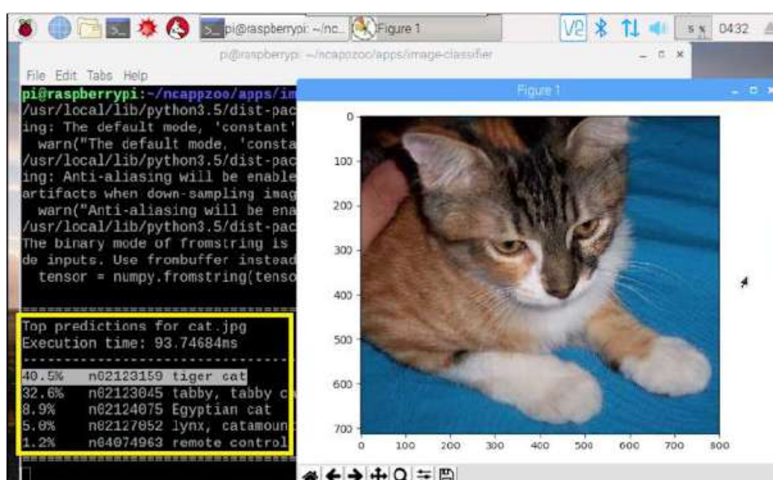



Рис. 15.17. Результат на виході

==============================================================

Top predictions for cat.jpg
Execution time: 93.694756ms
---------------------------------------------------------------
40.5% n02123159 tiger cat
32.6% n02123045 tabby, tabby cat
8.9% n02124075 Egyptian cat
5.0% n02127052 lynx, catamount
1.2% n04074963 remote control, remote
=====================================================================

На рис. 15.17 наведено зображення, запущене з VNC, або на робочому столі RPi.

Параметри коду (рис. 15.18 ):

```
139 if __name__ == '__main__':
140
141     parser = argparse.ArgumentParser(
142                          description="Image classifier using \
143                          Intel® Movidius™ Neural Compute Stick." )
144
145     parser.add_argument( '-g', '--graph', type=str,
146                          default='../../caffe/GoogLeNet/graph',
147                          help="Absolute path to the neural network graph file." )
148
149     parser.add_argument( '-i', '--image', type=str,
150                          default='../../data/images/cat.jpg',
151                          help="Absolute path to the image that needs to be inferred." )
152
153     parser.add_argument( '-l', '--labels', type=str,
154                          default='../../data/ilsvrc12/synset_words.txt',
155                          help="Absolute path to labels file." )
156
157     parser.add_argument( '-M', '--mean', type=float,
158                          nargs='+',
159                          default=[104.00698793, 116.66876762, 122.67891434],
160                          help="',' delimited floating point values for image mean." )
161
162     parser.add_argument( '-S', '--scale', type=float,
163                          default=1,
164                          help="Absolute path to labels file." )
165
166     parser.add_argument( '-D', '--dim', type=int,
167                          nargs='+',
168                          default=[224, 224],
169                          help="Image dimensions. ex. -D 224 224" )
170
171     parser.add_argument( '-c', '--colormode', type=str,
172                          default="BGR",
173                          help="RGB vs BGR color sequence. TensorFlow = RGB, Caffe = BGR" )
```

Рис. 15.18. Параметри коду

`GRAPH_PATH`: Розміщення файлу графа, за яким треба зробити вихід результату. За замовчуванням воно встановлене в `~/workspace/ncappzoo/caffe/GoogLeNet/graph`

`IMAGE_PATH`: Розміщення зображення, яке необхідно класифікувати.
За замовчуванням воно встановлене в `~/workspace/ncappzoo/data/images/cat.jpg`

`IMAGE_DIM`: Розміри зображення, які визначені вибраною нейронною мережею, наприклад, GoogLeNet використовує 224 x 224 пікселів, AlexNet використовує 227 x 227 пікселів.

`IMAGE_STDDEV`: Стандартне відхилення (значення масштабування) визначене вибраною нейронною мережею, наприклад, GoogLeNet не використовує коефіцієнт масштабування, InceptionV3 використовує 128 (stddev = 1/128).

`IMAGE_MEAN`: Середнє віднімання – це звичайна методика, яка використовується в глибокому навчанні для центрування даних. Для даних ILSVRC, середнім є B = 102 Green = = 117 Red = 123.

Якщо хочемо змінити зображення:

```
python3 image-classifier.py -i image file
```

Наприклад,

```
python3 image-classifier.py -i ../../data/images/pic_005.jpg
```

**Код Python**

До використання фреймворка API NCSDK, нам треба імпортувати модуль `mvncapi` з бібліотеки mvnc:

```
import mvnc.mvncapi as mvnc
```

**Крок 1. Відкриваємо під'єднаний пристрій**

Як і для будь-якого іншого USB-пристрою, коли підключаємо NCS до порту USB використовуваної плати сопроцесора (Ubuntu на ноутбуці / настільному ПК), він визначається як USB-пристрій. Викликаємо API для пошуку під'єднаного пристрою NCS.

```
# Пошук підключеного пристрою(їв) Intel Movidius NCS; зупинка програми,
якщо пристрій не знайдений.
devices = mvnc.EnumerateDevices()
if len( devices ) == 0:
    print( 'No devices found' )
    quit()
```

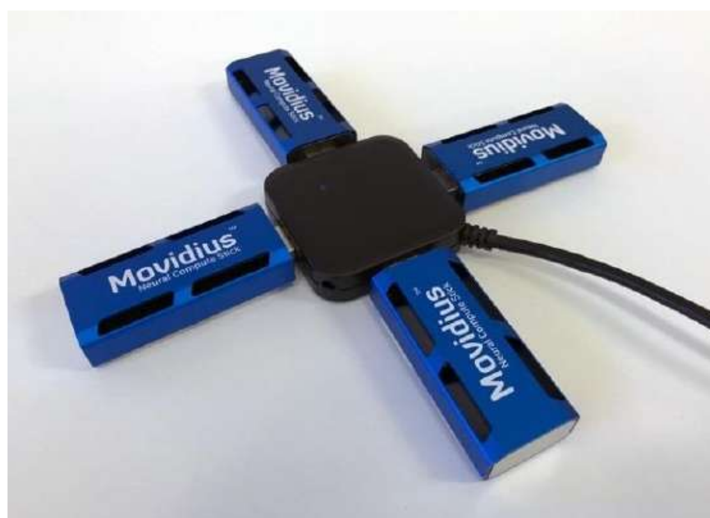



Рис. 15.19. Підключення кількох модулів NCS

Чи знаєте ви, що можна підключити декілька нейронних обчислювальних «флешок» до однієї і тієї ж плати процесора (рис. 15.19), щоб масштабувати параметри продуктивності? Отже, викликаємо API, щоб вибрати лише один NCS і відкрити його (підготуємо його до роботи).

```
# Дістаємося до першого відкритого пристрою та відкриваємо його
device = mvnc.Device( devices[0] )
device.OpenDevice()
```

**Крок 2. Завантажуємо файл графа на NCS**

Щоб зробити проєкт простішим, використаємо складений граф заздалегідь підготовленої моделі AlexNet, який вже завантажено і скомпільовано, коли запускали `make` всередині папки ncappzoo. Розглянемо, як завантажити граф в NCS.

```
# Читання файла графа в буфер
with open( GRAPH_PATH, mode='rb' ) as f:
    blob = f.read()
```

```
# Завантаження буфера з графом на NCS
graph = device.AllocateGraph( blob )
```

**Крок 3. Завантаження одного зображення на Intel Movidius NCS для запуску прогнозування**

Система Intel Movidius NCS працює на візуальному процесорі Intel Movidius (VPU), який забезпечує візуальний інтелект мільйонам смарт-камер безпеки, дронів, керованих жестами, обладнання промислового машинного зору тощо. Як і VPU, NCS являє собою візуальний спільний процесор для всієї системи. У нашому випадку використовуємо систему Ubuntu для простого зчитування зображень із папки і завантаження їх в NCS для прогнозування. Все опрацювання нейронної мережі здійснюється виключно NCS, тим самим звільняючи процесор і ресурси пам'яті для виконання інших завдань на рівні додатків.

Перед завантаженням зображення на NCS необхідно попередньо його опрацювати.

1. Змінюємо розмір / обріжемо зображення відповідно до розмірів, визначених заздалегідь підготовленою мережею.
   - GoogLeNet використовує 224 × 224 пікселі, AlexNet використовує 227 × 227 пікселів.
   - Віднімаємо середнє значення на канал (синій, зелений і червоний) від усього набору даних.
   - Це звичайна методика, яка використовується в глибокому навчанні для центрування даних.
2. Перетворюємо зображення в масив з плаваючою точкою на півточності (fp16) і використаємо `LoadTensorfunction-call` для завантаження зображення на NCS.
   - Бібліотека skimage може це зробити лише в одному рядку коду.

```
# Читаємо і зменшуємо зображення [Розмір зображення визначений під час
тренування]
img = print_img = skimage.io.imread( IMAGES_PATH )
img = skimage.transform.resize( img, IMAGE_DIM, preserve_range=True )

# Перетворюємо RGB в BGR [skimage читає зображення в RGB, але Caffe
використовує BGR]
img = img[:, :, ::-1]

# Віднімаємо середнє значення і масштабуємо & scaling [Загальноприйнята
техноогія для центрування даних]
img = img.astype( numpy.float32 )
img = ( img - IMAGE_MEAN ) * IMAGE_STDDEV

# Завантажуємо зображення як масив значень з плавачою комою половинної
точності
graph.LoadTensor( img.astype( numpy.float16 ), 'user object' )
```

**Крок 4. Читання і виведення результатів прогнозування з NCS**

Залежно від того, як треба інтегрувати результати прогнозування у потік додатків, можна використати або блокуючий, або неблокуючий виклик функції для завантаження тензора (попередній крок) і зчитати результати прогнозування. Просто використаємо функціональність за замовчуванням, яка є блокуючим викликом (не потрібно викликати конкретний API).

```
# Отримуємо результат з NCS
output, userobj = graph.GetResult()

# Виводимо результати
```

```
print('\n------- predictions --------')

labels = numpy.loadtxt( LABELS_FILE_PATH, str, delimiter = '\t' )

order = output.argsort()[::-1][:6]
for i in range( 0, 5 ):
    print ('prediction ' + str(i) + ' is ' + labels[order[i]])

# Відображаємо зображення яке обране за прогнозом
skimage.io.imshow( IMAGES_PATH )
skimage.io.show( )
```

**Крок 5. Вивантаження графа і закривання пристрою**

Щоб уникнути витоків пам’яті та / або помилок сегментації, треба закрити будь-які відкриті файли чи ресурси, а також звільнити всю використовувану пам’ять.

```
graph.DeallocateGraph()
device.CloseDevice()
```

Розглянутий код за замовчуванням використовує мережу GoogLeNet, але можемо налаштувати його за допомогою інших попередньо навчених глибоких нейронних мереж. Далі наведено кілька прикладів коду для використання інших мереж:

**AlexNet (Caffe)**

```
python3 image-classifier.py --graph ../../caffe/AlexNet/graph --dim 227
227 --image ../../data/images/pic_053.jpg
```

**SqueezeNet (Caffe)**

```
python3 image-classifier.py --graph ../../caffe/SqueezeNet/graph --dim
227 227 --image ../../data/images/pic_053.jpg
```

**Mobilenet (Tensorflow)**

```
python3 image-classifier.py --graph
../../tensorflow/mobilenets/model/graph --labels
../../tensorflow/mobilenets/model/labels.txt --mean 127.5 --scale
0.00789 --dim 224 224 --colormode="RGB" --image
../../data/images/pic_053.jpg
```

**Inception (Tensorflow)**

```
python3 image-classifier.py --graph
../../tensorflow/inception/model/v3/graph --labels
../../tensorflow/inception/model/v3/labels.txt --mean 127.5 --scale
0.00789 --dim 299 299 --colormode="RGB" --image
../../data/images/pic_053.jpg
```

**Тестування моделі Inception за допомогою TensorFlow**

Переходимо в директорію `~/ncappzoo/tensorflow/inception` (рис. 15.20 ):

```
cd ncappzoo/tensorflow/inception/
```

Запускаємо допомогу (рис. 15.20):

```
make help
```

Можливі команди make:

```
make help - Показує це повідомлення.
make all - Будує всі залежності, але не запускає програму.
make checkpoint - Завантажує попередньо підготовлені файли контрольної
точки.
make clean - Видаляє всі файли, створені в цьому проекті.
make export - Експортує модель нейронної мережі для передбачення.
make freeze - Заморожує модель нейронної мережі для передбачення.
make compile - Компілює заморожену модель у файл графа Movidius.
make check - Порівнює результати передбачення (прогнозування) з
результатами роботи TensorFlow, запущеному на CPU/GPU.
make profile - Запускає модель на NCS та витягує складність, пропускну
здатність та час виконання для кожного шару.
```

Будуємо всі залежності без запуску програми:

```
make all
```

TF_SRC_PATH not set, making tf_src
(cd ../tf_src; make all; cd /home/pi/ncappzoo/tensorflow/inception)
make[1]: Entering directory '/home/pi/ncappzoo/tensorflow/tf_src'
TF_SRC_PATH not set, will use project directory
TF_SRC_PATH is now: /home/pi/ncappzoo/tensorflow/tf_src/tensorflow
skipping clone, directory already exists: /home/pi/ncappzoo/tensorflow/tf_src/tensorflow
make[1]: Leaving directory '/home/pi/ncappzoo/tensorflow/tf_src'
TF_SRC_PATH is /home/pi/ncappzoo/tensorflow/inception/../tf_src/tensorflow

Завантаження файлів контрольної точки...

Exporting GraphDef file...
Freezing model for inference...
Profiling the model...

Запуск коду Python

```
python3 image-classifier.py -g ../../tensorflow/inception/model/v3/graph
-D 299 299 -M 127.5 -S 0.00789  -l
../../tensorflow/inception/model/v3/labels.txt -i
~/ncappzoo/data/images/cat.jpg
```

**Результат на виході:**

```
==============================================================
Top predictions for cat.jpg
Execution time: 317.015ms
--------------------------------------------------------------
54.7% 286:Egyptian cat
21.8% 282:tabby, tabby cat
8.6% 283:tiger cat
3.7% 288:lynx, catamount
2.3% 285:Siamese cat, Siamese
==============================================================
```

```
pi@raspberrypi:~ $ cd ncappzoo/
pi@raspberrypi:~/ncappzoo $ ls
apps   CONTRIBUTING.md  LICENSE   MAKEFILE_GUIDANCE.md  OWNERS     tensorflow
caffe  data             Makefile  ncapi2_shim           README.md
pi@raspberrypi:~/ncappzoo $ cd tensorflow/
pi@raspberrypi:~/ncappzoo/tensorflow $ cd inception
pi@raspberrypi:~/ncappzoo/tensorflow/inception $ ls
AUTHORS  Makefile  model  README.md
pi@raspberrypi:~/ncappzoo/tensorflow/inception $ make help

Possible make targets:
  make help - Shows this message.
  make all - Builds all dependencies, but does not run this program.
  make checkpoint - Downloads pre-trained checkpoint files.;
  make clean - Removes all files created in this project.;
  make export - Export the neural network model for inference.
  make freeze - Freeze the neural network model for inference.
  make compile - Convert the frozen model into Movidius graph file.
  make check - Compare inference results with that of TensorFlow running on CPU/GPU.
  make profile - Run the model on NCS and extract complexity, bandwidth and execution time for each layer.
pi@raspberrypi:~/ncappzoo/tensorflow/inception $ make run
```

Рис. 15.20. Вікно допомоги для доступних параметрів команди

Отримане передбачення визначило на зображенні «єгипетську кішку», але тепер вже з ймовірністю 54,7%.

## API для розпізнавання об'єктів з Raspberry Pi і TensorFlow

Розглянемо, як налаштувати API виявлення об'єктів TF на RPi[51] [51]. Виконуючи кроки з цього розділу, зможна використовувати свій RPi для виявлення об'єктів на відеоканалах із вебкамери Picamera або USB. Завдяки цьому ми зможемо навчити власну нейронну мережу ідентифікувати конкретні об'єкти і використовувати свій RPi для програм виявлення.

Необхідно пройти такі кроки:

1. Оновити Raspberry Pi.
2. Встановити TF.
3. Встановити OpenCV.
4. Компілювати і встановити Protobuf.
5. Налаштувати структуру каталогів TF і змінну PYTHONPATH.
6. Ідентифікувати об'єкти.

---

[51] https://github.com/EdjeElectronics/TensorFlow-Object-Detection-on-the-Raspberry-Pi/blob/master/README.md

У репозиторії також перебуває сценарій `Object_detection_picamera.py`, який є сценарієм Python для завантаження моделі виявлення об'єктів у TF і використання його для виявлення об'єктів у відеострічці Picamera. Розділ написано для TF v1.8.0 на RPi Model 3B під керуванням Raspbian Stretch v9. Ймовірно, що все буде працювати для новіших версій TF.

### 1. Оновлення Raspberry Pi

По-перше, RPi потрібно повністю оновити. Відкриваємо термінал і вводимо:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

Залежно від часу, який минув з моменту оновлення RPi, оновлення може тривати від хвилини до години (рис. 15.21).

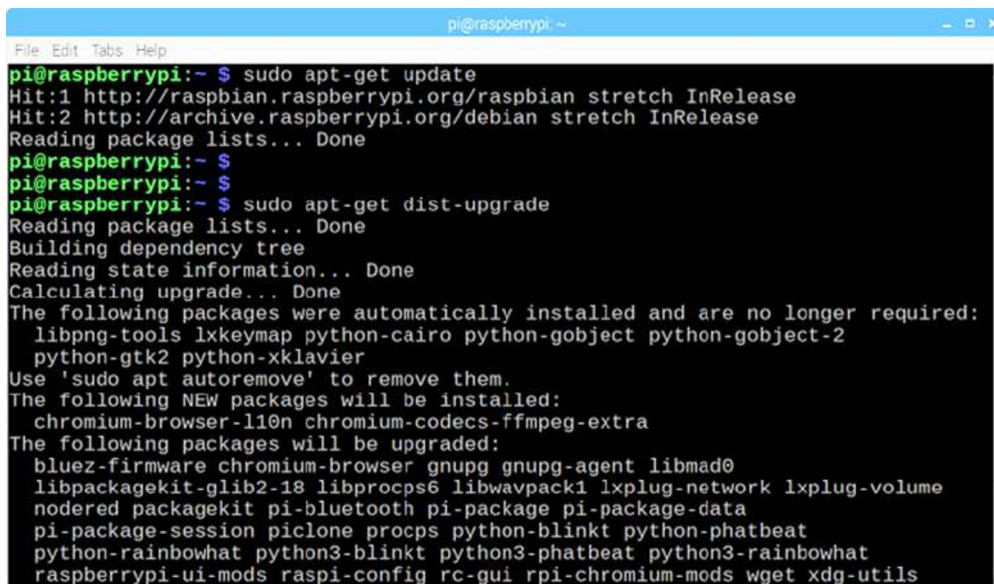



Рис. 15.21 Оновлення дистрибутива Raspberry Pi

### 2. Встановлення TensorFlow

Далі ми встановимо TF. Завантаження досить велике (понад 100 Мб), тому може тривати деякий час. Запускаємо команду:

```
pip3 install tensorflow
```

TF також потребує пакет LibAtlas. Встановимо його, запустивши наступну команду (якщо ця команда не працює, виконайте команду "`sudo apt-get update`" і повторіть спробу).

```
sudo apt-get install libatlas-base-dev
```

Поки пакет встановлюється, можна встановити інші залежності, які будуть використовуватися API TF виявлення об'єктів. Вони наведені в інструкціях з установки у сховищі GitHub для виявлення об'єктів. Виконаємо:

```
sudo pip3 install pillow lxml jupyter matplotlib cython
sudo apt-get install python-tk
```

Отже, це все, що нам потрібно для TF.

**3. Встановлення OpenCV**

Приклади виявлення об'єктів TF зазвичай використовують matplotlib для відображення зображень, але краще використовувати OpenCV, оскільки з ним простіше працювати і він менше схильний до помилок. Сценарії виявлення об'єктів у сховищі GitHub цього розділу використовують OpenCV. Отже, нам потрібно встановити OpenCV.

Щоб OpenCV працював на RPi, відомо доволі багато залежностей, які потрібно встановити через apt-get. Якщо якась з наведених нижче команд не працює, виконайте «`sudo apt-get update`» і повторіть спробу. Виконаємо:

```
sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev
libv4l-dev
sudo apt-get install libxvidcore-dev libx264-dev
sudo apt-get install qt4-dev-tools libatlas-base-dev
```

Отже, коли встановлено все, можемо встановити OpenCV. Виконуємо:

```
sudo pip3 install opencv-python
```

Зараз OpenCV встановлено.

**4. Компіляція і встановлення Protobuf**

API виявлення об'єктів TF використовує Protobuf – пакет, який реалізує формат даних протоколу Google Buffer. Раніше треба було компілювати його із першоджерела, а тепер це просте встановлення:

```
sudo apt-get install protobuf-compiler
```

Запустимо `protoc --version` після перевірки встановлення. Ми маємо отримати відповідь на `libprotoc 3.6.1` або подібний.

**5. Налаштування структури каталогів TensorFlow і змінної PYTHONPATH**

Отже, коли встановлені всі пакети, треба створити каталог TF. Повернемося в домашній каталог, а потім створимо каталог, який називається «tensorflow1», і перейдемо в нього:

```
mkdir tensorflow1
cd tensorflow1
```

Завантажимо репозиторій tensorflow з GitHub, виконавши:

```
git clone --depth 1 https://github.com/tensorflow/models.git
```

Далі треба змінити змінну середовища PYTHONPATH, щоб вказати на деякі каталоги всередині завантаженого репозиторія TF. Нам потрібно, щоб PYTHONPATH встановлювався щоразу, коли ми відкриваємо термінал, тому треба змінити файл `.bashrc`. Відкриваємо його, ввівши:

```
sudo nano ~/.bashrc
```

Переходимо в кінець файлу, а в останньому рядку додаємо:

```
export PYTHONPATH=$PYTHONPATH:/home/pi/tensorflow1/models/research:/home/pi/tensorflow1/models/research/slim
```

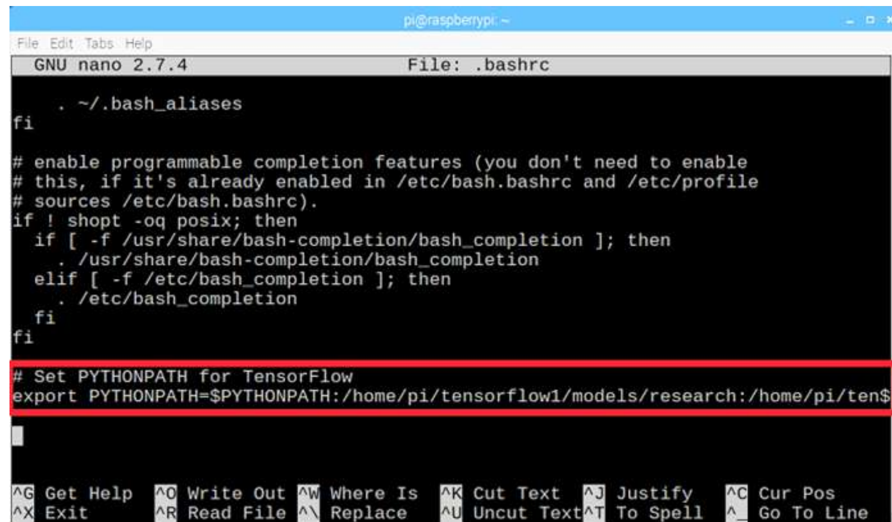



Рис. 15.22 Встановлення змінної середовища

Зберігаємо і закриваємо файл. Тепер буде так, що команда «export PYTHONPATH» викликається щоразу, коли відкриваємо новий термінал, тому змінна PYTHONPATH завжди буде встановлена відповідним чином. Закриваємо і знову відкриваємо термінал.

Тепер нам треба використовувати Protoc для компіляції файлів буфера протоколу (.proto), використовуваних API виявлення об'єктів. Файли .proto розташовані в `/research/object_detection/protos`, але нам треба виконати команду з каталогу `/research`. Вводимо:

```
cd /home/pi/tensorflow1/models/research
protoc object_detection/protos/*.proto --python_out=.
```

Ця команда перетворює всі файли "ім'я".proto у "name_pb2".py-файли. Далі переходимо у каталог object_detection:

```
cd /home/pi/tensorflow1/models/research/object_detection
```

Завантажимо модель SSD_Lite із TensorFlow detection model zoo[52] [52]. «Зоопарк» моделей – це колекція Google заздалегідь підготовлених моделей виявлення об'єктів, які мають різні рівні швидкості і точності. RPi має слабкий процесор, тому нам треба використовувати модель, яка потребує меншої потужності опрацювання. Хоча модель працюватиме швидше, вона матиме меншу точністю. Для цього розділу будемо використовувати SSDLite-MobileNet, яка є найшвидшою доступною моделлю.

Google постійно випускає моделі з підвищеною швидкістю і продуктивністю, тож часто переглядайте model zoo, щоб побачити, чи є кращі моделі.

[52]https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

Завантажимо модель SSDLite-MobileNet і розпакуємо її, ввівши:

```
wget
http://download.tensorflow.org/models/object_detection/ssdlite_mobilenet
_v2_coco_2018_05_09.tar.gz
tar -xzvf ssdlite_mobilenet_v2_coco_2018_05_09.tar.gz
```

Тепер модель перебуває в каталозі `object_detection` і готова до використання.

## 6. Виявлення об'єктів

Отже, тепер все налаштовано для виявлення об'єктів з RPi. Сценарій Python у нашому репозиторії Object_detection_picamera.py виявляє об'єкти у прямих каналах із вебкамери Picamera або USB. Здебільшого сценарій встановлює шляхи до карти моделі і мітки, завантажує модель у пам'ять, ініціалізує Picamera, а потім починає виконувати виявлення об'єктів на кожному кадрі відео з Picamera.

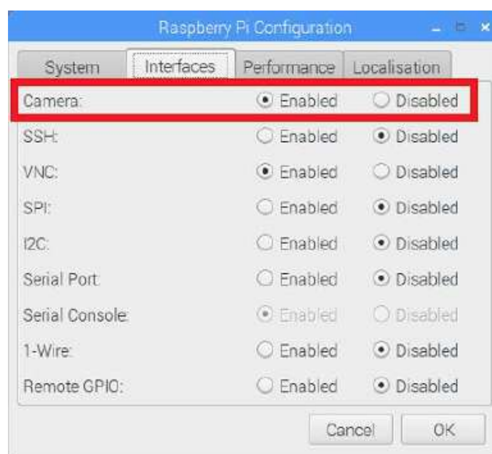



Рис. 15.23 Дозвіл камери

Якщо ми використовуємо Picamera, то треба переконатися, що камера дозволена в меню конфігурації RPi (рис. 15.23).

Завантажуємо файл `Object_detection_picamera.py` в каталог object_detection, ввівши:

```
wget https://raw.githubusercontent.com/EdjeElectronics/TensorFlow-Object-
Detection-on-the-Raspberry-Pi/master/Object_detection_picamera.py
```

Запускаємо сценарій, ввівши:

```
python3 Object_detection_picamera.py
```

Сценарій за замовчуванням використовує приєднану Picamera. Якщо натомість у нас є вебкамера USB, додаємо --usbcam до кінця команди:

```
python3 Object_detection_picamera.py --usbcam
```

Як тільки сценарій ініціалізується (що може тривати до 30 с), побачимо вікно, в якому відображається огляд зображення з камери. Загальні об'єкти всередині подання будуть ідентифіковані, а навколо них намальовано прямокутник (рис. 15.24).

З моделлю SSDLite, RPi 3 працює досить добре, досягаючи частоти кадрів вище 1FPS (1 кадр у секунду). Це доволі швидко для більшості програм виявлення об'єктів у реальному часі.

Можемо також використати модель, яку навчимо самостійно (див. посібник[53] [53]), додавши в каталог `object_detection` збережений граф і змінивши в сценарії шлях до моделі.

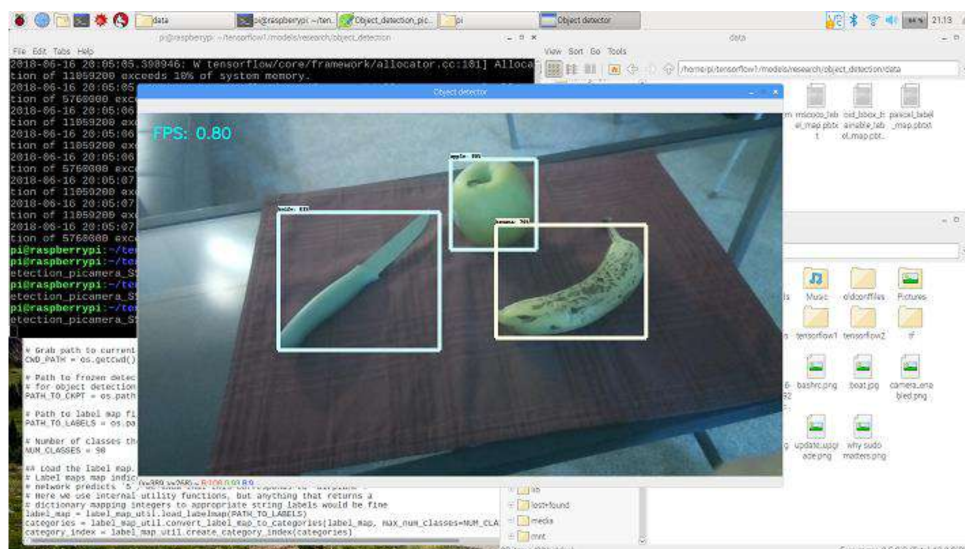



Рис. 15.24. Ідентифікація виявлених об’єктів

***Примітка.*** Якщо плануєте запускати все це на RPi протягом тривалого періоду часу (більше 5 хвилин), переконайтеся, що на основному процесорі RPi встановлено радіатор. Опрацювання викликає нагрівання процесора, а без радіатора він відключиться через високу температуру.

## Висновки

У навчальному посібнику викладено основи машинного навчання. Нові терміни пояснюються за допомогою практичних вправ з реалізації методів ML. Більшість прикладів реалізована з використанням найбільш поширеного фрейворку TensorFlow. Оскільки посібник призначено насамперед початківцям, приклади коду широко коментуються. В розглянутих завданнях наведено детальні результати виконання програмних кодів, щоб легше повторювати навчальні приклади.

У задачах розпізнавання зображень використано популярний набір рукописних цифр MNIST, для якого застосовуються різні алгоритми і порівнюються точності розпізнавання.

Рекомендована комбінація мікрокомп’ютера Raspbery Pi 4 B і Intel Neural Compute Stick 2 дає змогу не лише зручно і ефективно вивчати методи машинного навчання, але й може бути основою для створення сучасних пристроїв і систем комп’ютерного зору, кібербезпеки, робототехніки, голосового керування тощо.

Напрям машинного навчання розвивається швидкими темпами і матеріали пропонованого посібника можуть стати фундаментом для реалізації тих ідей, які щодня з’являються в Інтернеті, тож не зупиняйтесь на досягнутому, відстежуйте зміни й нові версії програмного забезпечення.

[53] https://www.youtube.com/watch?v=Rgpfk6eYxJA

## Використані джерела

1. Машинное обучение в повседневной жизни: типы ML и способы их применения [Електронний ресурс]: — Режим доступу: https://dou.ua/lenta/articles/ml-in-real-life/ — Назва з екрана.

2. Машинное обучение для людей [Електронний ресурс]: — Режим доступу: https://vas3k.ru/blog/machine_learning/ — Назва з екрана.

3. Cheat Sheets for AI, Neural Networks, Machine Learning, Deep Learning & Big Data [Електронний ресурс]: Stefan Kojouharov — Режим доступу: https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deep-learning-big-data-678c51b4b463 — Назва з екрана.

4. What is TensorFlow? Introduction, Architecture & Example [Електронний ресурс]: — Режим доступу: https://www.guru99.com/what-is-tensorflow.html — Назва з екрана.

5. How to Download and Install TensorFLow Windows and Mac [Електронний ресурс]: — Режим доступу: https://www.guru99.com/download-install-tensorflow.html — Назва з екрана.

6. Anaconda Distribution [Електронний ресурс]: — Режим доступу: https://www.anaconda.com/download/ — Назва з екрана.

7. Jupyter Notebook Tutorial: How to use with AWS [Електронний ресурс]: — Режим доступу: https://www.guru99.com/jupyter-notebook-tutorial.html — Назва з екрана.

8. Installing Machine Learning Software TensorFlow on Raspberry Pi [Електронний ресурс]: Rishabh Jain — Режим доступу: https://circuitdigest.com/microcontroller-projects/intalling-machine-learning-software-tensorflow-on-raspberry-pi — Назва з екрана.

9. Tensorflow - 2.0 [Електронний ресурс]: — Режим доступу: https://github.com/lhelontra/tensorflow-on-arm/releases — Назва з екрана.

10. TensorFlow Basics: Tensor, Shape, Type, Graph, Sessions & Operators [Електронний ресурс]: — Режим доступу: https://www.guru99.com/tensor-tensorflow.html — Назва з екрана.

11. Tensorboard Tutorial: Graph Visualization with Example [Електронний ресурс]: — Режим доступу: https://www.guru99.com/tensorboard-tutorial.html — Назва з екрана.

12. Python Pandas Tutorial: Dataframe, Date Range, Slice [Електронний ресурс]: — Режим доступу: https://www.guru99.com/python-pandas-tutorial.html — Назва з екрана.

13. [Електронний ресурс]: — Режим доступу: https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data — Назва з екрана.

14. pandas.read_csv [Електронний ресурс]: — Режим доступу: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html — Назва з екрана.

15. Linear Regression with TensorFlow [Examples] [Електронний ресурс]: — Режим доступу: https://www.guru99.com/linear-regression-tensorflow.html — Назва з екрана.

16 [Електронний ресурс]: — Режим доступу: https://drive.google.com/uc?export=download&id=10I5aboiafcqf0iVWnd1SiAEVbgWzgkW4 — Назва з екрана.

17. TensorFlow Linear Regression with Facet & Interaction Term. [Examples] [Електронний ресурс]: — Режим доступу: https://www.guru99.com/linear-regression-for-machine-learning.html — Назва з екрана.

18. Adding Interaction Terms. [Електронний ресурс]: — Режим доступу: https://chrisalbon.com/machine_learning/linear_regression/adding_interaction_terms/ — Назва з екрана.

19. Downloading Git. [Електронний ресурс]: — Режим доступу: https://git-scm.com/download/win — Назва з екрана.

20. Visualizations for machine learning datasets. [Електронний ресурс]: — Режим доступу: https://github.com/PAIR-code/facets — Назва з екрана.

21. Linear Classifier in TensorFlow: Binary Classification Example. [Електронний ресурс]: — Режим доступу: https://www.guru99.com/linear-classifier-tensorflow.html — Назва з екрана.

22. Activation function. [Електронний ресурс]: — Режим доступу: https://en.wikipedia.org/wiki/Activation_function — Назва з екрана.

23 Kernel Methods in Machine Learning: Gaussian Kernel (Example). [Електронний ресурс]: — Режим доступу: https://www.guru99.com/kernel-methods-machine-learning.html — Назва з екрана.

24. Index of /ml/machine-learning-databases/adult. (Example). [Електронний ресурс]: — Режим доступу: https://archive.ics.uci.edu/ml/machine-learning-databases/adult/ — Назва з екрана.

25. Neural Network Tutorial: TensorFlow ANN Example. [Електронний ресурс]: — Режим доступу: https://www.guru99.com/artificial-neural-network-tutorial.html — Назва з екрана.

26. Tinker With a Neural Network Right Here in Your Browser. [Електронний ресурс]: — Режим доступу: http://playground.tensorflow.org/ — Назва з екрана.

27. TensorFlow Image Classification: CNN (Convolutional Neural Network). [Електронний ресурс]: — Режим доступу: https://www.guru99.com/convnet-tensorflow-image-classification.html — Назва з екрана.

28. The MNIST Database. [Електронний ресурс]: — Режим доступу: http://yann.lecun.com/exdb/mnist/ — Назва з екрана.

29. Kernel (image processing). [Електронний ресурс]: — Режим доступу: https://en.wikipedia.org/wiki/Kernel_(image_processing) — Назва з екрана.

30. Undrestanding Convolutional Layers in Convolutional Neural Networks (CNNs). [Електронний ресурс]: — Режим доступу: http://machinelearninguru.com/computer_vision/basics/convolution/convolution_layer.html — Назва з екрана.

31. TensorFlow Autoencoder: Deep Learning Example. [Електронний ресурс]: — Режим доступу: https://www.guru99.com/autoencoder-deep-learning.html — Назва з екрана.

32. The CIFAR-10 dataset. [Електронний ресурс]: — Режим доступу: https://www.cs.toronto.edu/~kriz/cifar.html — Назва з екрана.

33. RNN (Recurrent Neural Network) Tutorial: TensorFlow Example. [Електронний ресурс]: — Режим доступу: https://www.guru99.com/rnn-tutorial.html — Назва з екрана.

34. tf.nn.rnn_cell.LSTMCell. [Електронний ресурс]: — Режим доступу: https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/LSTMCell — Назва з екрана.

35. Myriad 2 MA2x5x Vision Processor. [Електронний ресурс]: — Режим доступу: https://uploads.movidius.com/1463156689-2016-04-29_VPU_ProductBrief.pdf — Назва з екрана.

36. What is the vcgencmd command?. [Електронний ресурс]: — Режим доступу: https://raspberrypi.stackexchange.com/questions/85345/what-is-the-vcgencmd-command — Назва з екрана.

37. Get Started with Intel® Neural Compute Stick 2. [Електронний ресурс]: — Режим доступу: https://software.intel.com/en-us/neural-compute-stick/get-started — Назва з екрана.

38. Install OpenVINO™ toolkit for Raspbian* OS. [Електронний ресурс]: — Режим доступу: https://software.intel.com/en-us/articles/OpenVINO-Install-RaspberryPI — Назва з екрана.

39. How to run Keras model on Movidius neural compute stick. [Електронний ресурс]: — Режим доступу: https://www.dlology.com/blog/how-to-run-keras-model-on-movidius-neural-compute-stick/ — Назва з екрана.

40. Train a Keras MNIST model for Movidius NCSDK2. [Електронний ресурс]: — Режим доступу: https://github.com/Tony607/keras_mnist — Назва з екрана.

41. Tony607/keras_mnist. [Електронний ресурс]: — Режим доступу: https://github.com/Tony607/keras_mnist/blob/master/train-mnist.py — Назва з екрана.

42. How to run Object Detection and Segmentation on a Video Fast for Free. [Електронний ресурс]: — Режим доступу: https://www.dlology.com/blog/how-to-run-object-detection-and-segmentation-on-video-fast-for-free/ — Назва з екрана.

43. ncsdk-2.05.00.02.tar.g. [Електронний ресурс]: — Режим доступу: https://ncs-forum-uploads.s3.amazonaws.com/ncsdk/ncsdk-02_05_00_02-full/ncsdk-2.05.00.02.tar.gz — Назва з екрана.

44. ImageProcessor.py. [Електронний ресурс]: — Режим доступу: https://github.com/Tony607/keras_mnist/blob/master/ImageProcessor.py — Назва з екрана.

45. Робот-танк на Raspberry Pi с Intel Neural Computer Stick 2. [Електронний ресурс]: — Режим доступу: https://habr.com/ru/post/459126/ — Назва з екрана.

46. Pretrained Models. [Електронний ресурс]: — Режим доступу: https://software.intel.com/en-us/openvino-toolkit/documentation/pretrained-models — Назва з екрана.

47. pitanq-selfwalk. [Електронний ресурс]: — Режим доступу: https://github.com/tprlab/pitanq-selfwalk — Назва з екрана.

48. Converting a TensorFlow* Model. [Електронний ресурс]: — Режим доступу: https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html — Назва з екрана.

49. Converting TensorFlow*-Slim Image Classification Model Library Models. [Електронний ресурс]: — Режим доступу: https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_tf_specific_Convert_Slim_Library_Models.html — Назва з екрана.

50. Build an Image Classifier in 5 steps. [Електронний ресурс]: — Режим доступу: https://movidius.github.io/blog/ncs-image-classifier/ — Назва з екрана.

51. Tutorial to set up TensorFlow Object Detection API on the Raspberry Pi. [Електронний ресурс]: — Режим доступу: https://github.com/EdjeElectronics/TensorFlow-Object-Detection-on-the-Raspberry-Pi/blob/master/README.md— Назва з екрана.

52. Tensorflow detection model zoo. [Електронний ресурс]: — Режим доступу: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md — Назва з екрана.

53. How To Train an Object Detection Classifier Using TensorFlow (GPU) on Windows 10. [Електронний ресурс]: — Режим доступу: https://www.youtube.com/watch?v=Rgpfk6eYxJA — Назва з екрана.