МІНІСТЕРСТВООСВІТИІ НАУКИУКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
Навчально-науковий інститут Інформаційних технологій
Кафедра штучного інтелекту

Чичкарьов Є.А.

Методичні рекомендації щодо проведення практичних занять з дисципліни «Сучасні технології програмування в системах зі штучним інтелектом» для студентів спеціальності 122 Комп’ютерні науки всіх форм навчання

Практична робота № 2. Бібліотека Keras, побудова і навчання моделей

Розглянуто на засіданні кафедри ШІ
Протокол № 1 від «29» серпня 2024 р

Київ
2024

УДК 004.658

Чичкарьов Є.А. Методичні рекомендації щодо проведення практичних занять з дисципліни «Сучасні технології програмування в системах зі штучним інтелектом» для студентів спеціальності 122 Комп’ютерні науки всіх форм навчання. Практична робота № 2. Бібліотека Keras, побудова і навчання моделей. - Київ: ДУТ, 2024. 18 с.

Методичні рекомендації призначені для ознайомлення студентів спеціальності 122 Комп’ютерні науки всіх форм навчання з різними аспектами створення додатків для вирішення задач штучного інтелекту та набуття ними практичного досвіду створення додатків, які використовують технології глибокого навчання і пакету TensorFlow, при виконанні практичних занять з дисципліни «Сучасні технології програмування в системах зі штучним інтелектом».

Рецензенти:

Рекомендовано
на засіданні кафедри штучного інтелекту,
протокол № 1 від 29 серпня 2024 р.

ЗМІСТ

Практична робота №2. Бібліотека Keras, побудова і навчання моделей

Мета: *ознайомитися з пакетом Keras, його можливостями для побудови моделей TensorFlow, з варіантами API для побудови моделей, типами і особливостями основних шарів, які використовують для створення моделей.*

1 Теоретичний розділ

Keras — це API з відкритим вихідним кодом, який використовується для вирішення різноманітних проблем сучасного машинного та глибокого навчання. Це дозволяє користувачеві більше зосередитися на логічному аспекті глибокого навчання, а не на аспектах грубого кодування. Keras — це надзвичайно потужний API, який забезпечує чудову масштабованість, гнучкість і когнітивну легкість завдяки зменшенню робочого навантаження користувача. Він написаний на Python і використовує TensorFlow як серверну частину.

1.1 Створення моделей з шарів

Layers API є ключовим компонентом Keras, що дозволяє складати попередні визначені шари або створювати спеціальні шари для вашої моделі.

Шари є основними елементами, необхідними під час створення нейронних мереж. Послідовні верстви відповідають за архітектуру моделі глибокого навчання. Кожен з них виконує обчислення на основі даних, отриманих із попереднього. Потім інформація передається далі. Зрештою, останній шар видає потрібний результат. У цьому матеріалі розберемо типи шарів Keras, їх властивості та параметри.

Для визначення або створення шару Keras потрібна наступна інформація:

Форма введення: для розуміння структури вхідної інформації

Кількість: для визначення кількості вузлів/нейронів у шарі

Ініціалізатор: для визначення ваг кожного входу, що важливо для виконання обчислень

Активатори: для перетворення вхідних даних на нелінійний формат, щоб кожен нейрон міг навчатися ефективніше

Обмежувачі: для накладання обмежень на ваги під час оптимізації

Регулятори: для застосування штрафів до параметрів під час оптимізації

1.2 Основні шари Keras

1.2.1. Згортковий шар

Цей шар в основному використовується у випадку обробки зображення або відеозадач для просторової згортки зображення або компонентів. Функціональність шару згортки відображається в зображенні вказаних фільтрів для вхідного зображення для створення функцій картки. Різні типи

шарів згортки: Conv1D, Conv2D, Conv3D, DepthwiseConv2D Layer, SeparableConv2D Layer тощо, серед яких Conv2D є тим, який використовується найчастіше.

Синтаксис: *layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1,1), activation='relu', padding='same')*

Параметри:

- filters: вказує кількість фільтрів для шару згортки для створення карти об'єктів.
- kernel_size: Розміри ядра Convolution включені через цей параметр.
- strides: вказує розмір кроку ковзного вікна під час згортання.
- activation: вказує функцію активації для введення нелінійності в шар.
- Інші додаткові параметри включають padding, kernel_initializer тощо,
- Приклад демонструє конволюційний шар із 32 фільтрами та ядром 3×3 з рухом по 1 кроку в обох напрямках і функцією активації relu.

1.2.2. Pooling Layer

Шар Pooling використовується для зменшення розмірів карти функцій із попереднього шару перед передачею на наступний, щоб зробити обчислення швидшим і запобігти переобладнанню. Два основних типи шару об’єднання — max pooling layer and average pooling layer.

- шар MaxPooling займає максимум вхідної області. Якщо ми розглядаємо матрицю 2×2, вона замінюється одним значенням, яке є максимальним серед чотирьох значень.
- шар Average Pooling подібним чином приймає середнє значення всіх вхідних значень. Якщо ми розглядаємо матрицю 2×2, вона замінюється одним значенням, яке є середнім з усіх чотирьох значень.

Syntrax *layers.MaxPooling2D(pool_size=(2,2), strides=(1,1), padding='same')*

Параметри:

- pool_size: представляє розміри вікна або сітки, над якими виконується об’єднання у вхідній матриці.
- strides, padding — два інші параметри, які можна використовувати з шаром Pooling.
- Приклад - шар максимального Pooling з розміром вікна 2 × 2 і однаковим відступом, що робить вихідну форму такою ж, як вхідна.

1.2.3. Щільний (Dense) шар

Шар DeНайбільш часто використовуваний шар keras, який з’єднує кожен нейрон попереднього шару з кожним нейроном поточного шару. Він також відомий як повністю підключений рівень.

Синтаксис: *layers.Dense(units = 4, use_bias = False)*

Параметр:

- units: це основний параметр шару Dense. Він являє собою кількість нейронів у шарі.
- use_bias: логічне значення, яке вказує, чи потрібно включити вектор зсуву в шар чи ні. Значенням за замовчуванням для цього параметра є True.

- деякі інші загальні параметри, такі як kernel_initializer, activation, kernel_regularizer, можна вказати за допомогою щільного шару
- units = 4 - Шар Dense із 4 нейронами без будь-яких зміщень, доданих до суми ваг нейронів.

1.2.4. Шар Flatten

Шар Flatten використовується для вирівнювання вхідних даних. Незалежно від розмірів тензора, шар Flatten перетворює його на одновимірний тензор. Наприклад, якщо ми розглянемо застосування шару Flatten до вхідних даних 2×2, на виході буде одновимірний тензор із 4 значеннями.

Синтаксис: *layers.Flatten(data_format = None),*

- Єдиним параметром, який приймає шар зведення, є data_format, який використовується для перемикання між форматами даних.

1.2.5. Шар Dropout

Переобладнання є однією з критичних проблем, пов'язаних із нейронними мережами. Щоб гарантувати відсутність переобладнання моделі до даних, до яких введено цей щар Dropout. Цей шар випадковим чином обнулює частину або фракцію вхідних даних під час навчання.

Синтаксис: *layers.Dropout(rate = 0,5,noise_shape=(16,32),seed=32)*

Параметри:

- rate: вказує частку загальних одиниць введення, які потрібно викинути. Значення швидкості лежить між 0 і 1.
- noise_shape : одновимірний цілочисельний тензор, який відображає форму бінарної маски вилучення та множиться на вхідні дані.
- seed: випадкове ціле число Python для відтворюваності.

1.2.6. Активаційний шар

- Функцію активації також можна ввести як окремий шар замість того, щоб включати її як параметр в інші шари keras.
- ***Синтаксис:*** *activation_layer = layers.Activation('relu')*

1.3 Моделі в Keras

У Keras ви збираєте шари (layers) для побудови моделей (models). Модель це (зазвичай) граф шарів. Найбільш поширеним видом моделі є стек шарів: модель tf.keras.Sequential.

Існує три основних API моделі keras (див. рис. 1).

1.3.1. Послідовний (Sequential)

Sequential API — це найпростіший і найзрозуміліший спосіб створення нейронних мереж у TensorFlow. Особливості:

- Послідовний API є найпростішим і широко використовуваним способом створення моделі Keras. Послідовна модель keras створюється шляхом послідовного додавання шарів з одним вхідним і вихідним тензором. Це не працює для тензорів з декількома входами та вихідними сигналами.
- Послідовна модель із лінійним набором згорткових і щільних шарів.

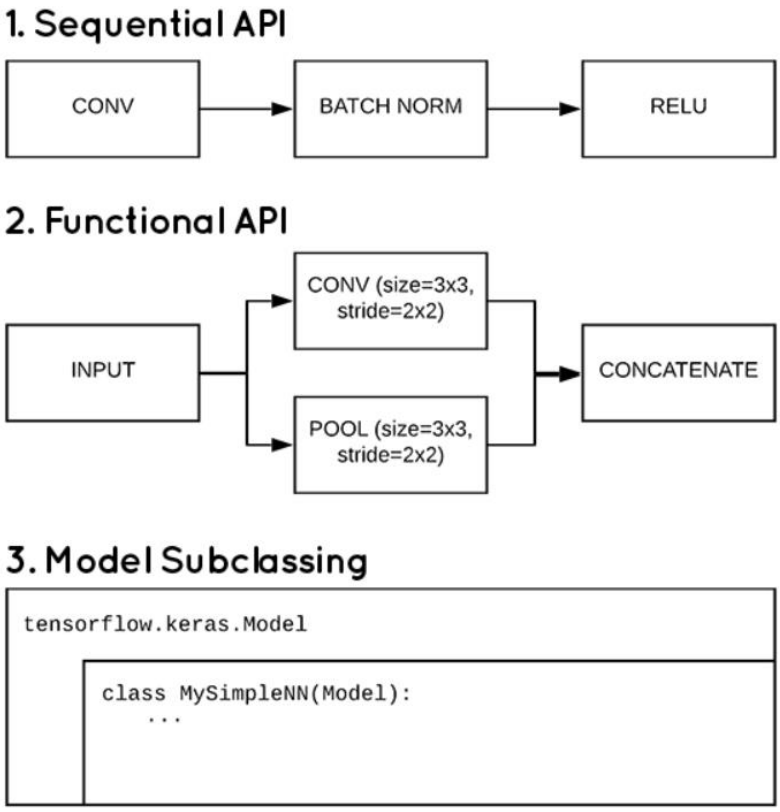

Рис. 1 – Варіанти API побудови моделей в Keras

Переваги Sequential API:

- Простота: ідеально підходить для побудови простих мереж прямого зв'язку, де шари складаються послідовно.
- Простота використання: інтуїтивно зрозумілий і простий для розуміння, що робить його чудовим для початківців.
- Швидке створення прототипів: швидке створення прототипів для базових моделей.

Недоліки Sequential API:

- Обмежена гнучкість: послідовний API розроблено для створення простих архітектур із прямою передачею даних, він може не підходити для більш складних моделей з розгалуженням або кількома входами та виходами.
- Відсутність явного потоку даних: у послідовній моделі передача даних між шарами не настільки чітка, це може бути недоліком під час створення моделей із складною архітектурою.
- Неможливість поділитися шарами: він нелегко підтримує спільне використання рівнів між різними частинами мережі, що може бути важливим для певних типів моделей.

Приклад:

```
from tensorflow.keras import models
from tensorflow.keras import layers
```

```
model=models.Sequential()
model.add(layers.Conv2D(32,(3,3),activation="relu"))
model.add(layers.Dense(32))
```

Ще один приклад:

```
імпорт tensorflow як tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
   Dense( 64 , activation= 'relu' , input_shape=(input_shape,)),
   Dense( 32 , activation= 'relu' ),
   Dense( 10 , activation= 'softmax' )
])
```

1.3.2. Функціональний (Functional)

- У Keras представлено функціональний API для гнучкого створення моделей зі складною архітектурою. Для роботи з кількома вхідними та вихідними тензорами використовується функціональна модель Кераса.
- У випадку функціонального API виходи одного рівня підключаються до входів наступного рівня зовні для більшої гнучкості та контролю над архітектурою моделі. За допомогою цього API можна створювати складні мережеві архітектури, такі як моделі графів.

Переваги функціонального API:

- Гнучкість: підходить для побудови моделей з непослідовними архітектурами, такими як мережі з кількома входами та кількома виходами.
- Можливість повторного використання: дає змогу ділитися шарами та повторно використовувати частини моделі в різних конфігураціях.
- Явний потік даних: надає чіткий і чіткий спосіб визначення потоку даних через модель.

Недоліки функціонального API:

- Крута крива навчання: порівняно з Sequential API, Functional API має крутішу криву навчання, це вимагає глибшого розуміння TensorFlow і основного обчислювального графіка.
- Більш складний для реалізації: створення моделей із декількома входами, декількома виходами або спільними шарами може бути більш складним і докладним у функціональному API.
- Менш інтуїтивно зрозумілий для простих моделей: для дуже простих моделей функціональний API може створити непотрібну складність, у таких випадках Sequential API може бути більш доречним.

Приклад:

```
from tensorflow.keras import models
from tensorflow.keras import layers
inputs = layers.Input(shape=(784,))
x = layers.Dense(64)(inputs)
x = layers.Activation('relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)
model = models.Model(inputs=inputs, outputs=outputs)
```

1.3.3 Підклас моделі

Підкласи моделі — це найбільш настроюваний підхід до створення моделей у TensorFlow. Замість того, щоб використовувати попередньо визначений API, ви створюєте спеціальний клас, який успадковує `tf.keras.Model` та визначає перехід у методі `call`.

Переваги підкласу моделі:

- Максимальна гнучкість: дозволяє повністю налаштувати архітектуру моделі та цикл навчання.
- Динамічна поведінка: дозволяє динамічно змінювати архітектуру на основі умов або вхідних даних.
- Складні архітектури: підходить для побудови вузькоспеціалізованих або науково-орієнтованих моделей.

Недоліки підкласу моделі:

- Потенціал коду, схильного до помилок: підкласи моделі включають написання спеціального коду Python для архітектури моделі, це може бути схильним до помилок, особливо для тих, хто новачок у TensorFlow.
- Менш прозоре подання графіка: динамічний характер підкласів моделі ускладнює перевірку архітектури моделі, оскільки її не так легко представити на статичному графіку.
- Складніше налагодження: налагодження моделей, створених за допомогою підкласів моделей, може бути складнішим порівняно з іншими API через використання спеціального коду.
- Обмежена серіалізація: збереження та завантаження моделей, створених за допомогою підкласів моделей, вимагає особливої обережності. Це не так просто, як із послідовними або функціональними API.

Зразок коду:

```
import tensorflow як tf
from tensorflow.keras.layers import Dense

class CustomModel (tf.keras.Model):
   def  __init__ ( self, num_classes ):
      super (CustomModel, self).__init__()
```

```
        self.dense1 = Dense( 64 , activation= 'relu' )
        self.dense2 = Dense( 32 , activation= 'relu' )
        self.dense3 = Dense(num_classes, activation= 'softmax' )

    def  call ( self, inputs ):
        x = self.dense1(inputs)
        x = self.dense2(x)
        return self.dense3(x)

model = CustomModel(num_classes= 10 )
```

Підсумовуючи, Keras Layers API пропонує багату та універсальну структуру для створення широкого діапазону архітектур нейронних мереж, від простих до складних. Він забезпечує різноманітність рівнів, необхідних для побудови різних типів нейронних мереж, придатних для широкого спектру застосувань від обробки зображень і відео до аналізу тексту

1.4 Приклад побудови бінарного класифікатора

1.4.1 Поняття бінарної та багатокласової класифікації

Бінарна класифікація є фундаментальним завданням у машинному навчанні, де метою є класифікувати дані в один із двох класів або категорій.

Бінарна класифікація використовується в широкому діапазоні програм, таких як виявлення спаму, медична діагностика, аналіз настроїв, виявлення шахрайства та багато іншого.

У цій роботі досліджено бінарну класифікацію за допомогою TensorFlow.

Проблема класифікації — це тип машинного навчання або статистичної задачі, мета якої — призначити категорію або мітку набору вхідних даних на основі їхніх характеристик або ознак. Мета полягає в тому, щоб вивчити відображення між вхідними даними та попередньо визначеними класами або категоріями, а потім використовувати це відображення для прогнозування міток класів нових, невидимих точок даних (рис. 2-3).

На рис. 2 ілюстрація до виконання бінарної класифікації, де дані класифікуються за двома типами класів.

На рис. 3 наведено діаграму, яка ілюструє задачу мультикласифікації. Дані будуть класифіковані за більш ніж двома (тут трьома) типами класів.

Цієї простої концепції достатньо, щоб зрозуміти проблеми класифікації.

Нижче досліджено приклад побудови бінарного класифікатора.

1.4.2 Приклад прогнозування серцевого нападу за допомогою двійкової класифікації

Розглянемо побудову прогностичної моделі для аналізу інфаркту з використанням простих бібліотек глибокого навчання.

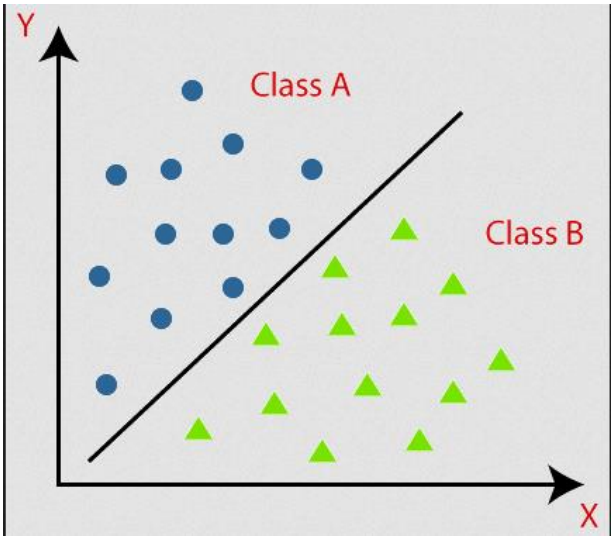

Рис. 2 Зразок бінарної класифікації

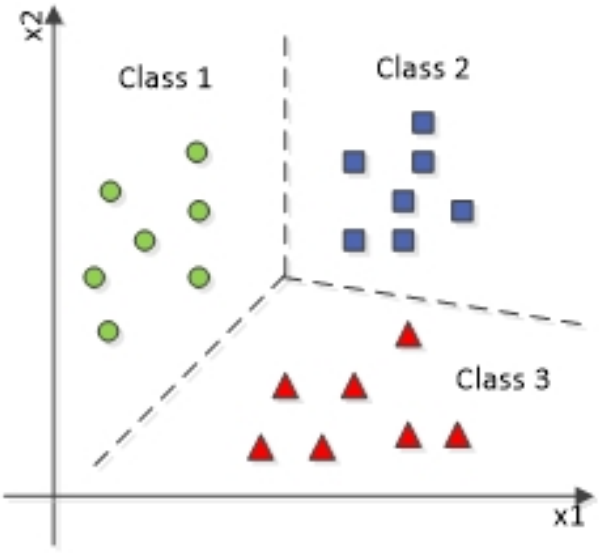

Рис. 3 Зразок мультикласифікації

Вирішення проблем реального світу через призму машинного навчання передбачає низку важливих кроків:

1. Збір даних і аналітика
2. Попередня обробка даних
3. Створення моделі ML
4. Навчання моделі
5. Прогнозування та оцінка

Збір даних і аналітика

Дані для аналізу треба взяти зі сторінки Kaggle (для завантаження треба зареєструватися на Kaggle з google-акаунтом):

https://www.kaggle.com/datasets/rashikrahmanpritom/heart-attack-analysis-prediction-dataset

Цей набір даних добре структурований, і немає негайної потреби в подальшому аналізі.
Для початку роботи на моделлю треба імпортувати необхідні бібліотеки:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import sklearn
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import tensorflow as tf
from sklearn.metrics import confusion_matrix,ConfusionMatrixDisplay
from sklearn.preprocessing import MinMaxScaler
```

Припускаємо, що набір даних для аналізу збережено на Google-disk (це зовсім не обов'язково, але досить зручно). Завантажуємо набір даних в Pandas Dataframe за допомогою методу read_csv:

```
df = pd.read_csv("/content/drive/MyDrive/Datasets/heart.csv")
df.head() # результат наведено на рис. 4
```

	age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	caa	thall	output
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

Рис. 4 – Вигляд зразку перших 5 рядків у наборі даних

Набір даних містить тринадцять вхідних стовпців (вік, стать та інши) і один вихідний стовпець (output), який міститиме дані 0 або 1.

Зважаючи на вхідні дані, 0 на виході (стовпчику output) означає, що у людини не буде серцевого нападу, а 1 означає, що у людини буде серцевий напад..

Давайте розділимо наші вхідні та вихідні дані з наведеного вище набору даних, щоб навчити нашу модель:

```
target_column = "output"
```

```
numerical_column = df.columns.drop(target_column)
output_rows = df[target_column]
df.drop(target_column,axis=1,inplace=True)
```

Оскільки наша мета — передбачити ймовірність серцевого нападу (0 або 1), представлену цільовим стовпцем, ми розділили це на окремий набір даних.

Попередня обробка даних

Попередня обробка даних є важливим кроком у конвеєрі машинного навчання, і двійкова класифікація не є винятком. Це передбачає очищення, перетворення та організацію необроблених даних у формат, який підходить для навчання моделей машинного навчання.

Набір даних для навчання моделі може міститиме кілька типів даних, як-от числові дані, категоричні дані, дані позначки часу тощо.

Але більшість алгоритмів машинного навчання розроблено для роботи з числовими даними. Вони вимагають, щоб вхідні дані були в числовому форматі для математичних операцій, оптимізації та навчання моделі.

У наборі даних в розглянутому прикладі (див. вище) усі стовпці містять числові дані, тому немає потреби кодувати дані.

Якщо у наборі даних, який обрано для навчання моделі, є нечислові стовпці, треба перетворити їх на числові, виконавши одноразове кодування або використовуючи інші алгоритми кодування.

Для нормалізації даних обрано перетворювач MinMaxScaler від Scikit-Learn, який працює за формулою:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Код використання scaler:

```
scaler = MinMaxScaler()
scaler.fit(df)
t_df = scaler.transform(df)
```

Рядок scaler = MinMaxScaler() створює відповідний об’єкт масштабувальника.

Рядок scaler.fit(df) обчислює середнє значення та стандартне відхилення (або інші параметри масштабування), необхідні для виконання операції масштабування. Метод fit, по суті, обчислює ці параметри з даних.

Рядок t_df = scaler.transform(df) безпосередньо трансформує набір даних.

Перетворення зазвичай масштабує ознаки, щоб мати середнє значення 0 і стандартне відхилення 1 (стандартизація) або масштабує їх до певного діапазону (наприклад, [0, 1] з мінімально-максимальним масштабуванням) залежно від використовуваного масштабувальника.

Наступним важливим кроком є розділення набору даних на набори для навчання та тестування. Зручним механізмом для цього є використання функції train_test_split з пакету Scikit-learn.

Нижче змінні X_train і X_test містять незалежні дані.

Змінні y_train і y_test містять залежні дані, тобто результат, який ми прагнемо передбачити.

```
X_train, X_test, y_train, y_test = train_test_split(t_df, output_rows,
test_size=0.25, random_state=0)
print('X_train:',np.shape(X_train))
print('y_train:',np.shape(y_train))
print('X_test:',np.shape(X_test))
print('y_test:',np.shape(y_test))
```

Код вище розділяє набір даних на 75% і 25%, де 75% йде на навчання нашої моделі, а 25% йде на тестування нашої моделі.

Створення моделі ML

Модель машинного навчання — це обчислювальне представлення проблеми або системи, розроблене для вивчення шаблонів, зв’язків і асоціацій із даних. Він служить математичною та алгоритмічною основою, здатною робити прогнози, класифікувати або приймати рішення на основі вхідних даних.

По суті, модель інкапсулює знання, отримані з даних, дозволяючи їй узагальнювати та робити обґрунтовані відповіді на нові, раніше невідомі дані.

Нижче побудовано проста послідовна модель з одним вхідним і одним вихідним шарами.

Ініціалізація послідовної моделі

```
basic_model = Sequential()
```

Sequential - це тип моделі в Keras, який дозволяє створювати нейронні мережі шар за шаром у послідовний спосіб. Кожен шар додається поверх попереднього.

В якості вхідного додаємо щільний (Dense) шар:

```
basic_model.add(Dense(units=16, activation='relu', input_shape=(13,)))
```

Dense - це тип шару в Keras, що представляє повністю зв’язаний шар. Він має 16 нейронів (units) з функцією активації Rectified Linear Unit (ReLU), яка зазвичай використовується у вхідних або прихованих рівнях нейронних мереж (activation='relu').

Форма вхідних даних для цього шару вказана як input_shape=(13,). У цьому випадку ми використовуємо 13 вхідних елементів (стовпців).

В якості вихідного додаємо щільний (Dense) шар:

```
basic_model.add(Dense(1, activation='sigmoid'))
```

Цей рядок додає вихідний шар до моделі.

Це один нейрон (1 unit), тому що це проблема бінарної класифікації, коли ви передбачаєте один із двох класів (0 або 1).

Тут використовується функція активації 'sigmoid', яка зазвичай використовується для завдань двійкової класифікації. Він стискає результат до діапазону від 0 до 1, що представляє ймовірність належності до одного з двох класів.

В якості оптимізатора обираємо алгоритм Adam:

```
adam = keras.optimizers.Adam(learning_rate=0.001)
```

Цей рядок ініціалізує оптимізатор Adam зі швидкістю навчання 0,001. Оптимізатор відповідає за оновлення вагових коефіцієнтів моделі під час навчання, щоб мінімізувати визначену функцію втрат.

Для подальшого використання модель треба скомпілювати

```
basic_model.compile(loss='binary_crossentropy', optimizer=adam,
metrics=["accuracy"])
```

параметри, які вказано при компіляції:

loss='binary_crossentropy' - це функція втрат, яка використовується для двійкової класифікації. Вона вимірює різницю між прогнозованими та фактичними значеннями та мінімізується під час навчання.

metrics=["accuracy"] – вказуємо показник точності під час навчання моделі.

Навчання моделі

Для навчання моделі нижче використано метод fit:

```
basic_model.fit(X_train, y_train, epochs=100)
```

Параметри навчання:

X_train - дані для навчання, які складаються з незалежних змінних (ознак).

y_train – дані з відповідними цільовими мітками або залежними змінними для навчальних даних.

epochs=100 - параметр визначає кількість разів, коли модель буде повторювати весь навчальний набір даних. Кожен прохід у наборі даних називається епохою. У цьому випадку ми маємо 100 епох, тобто модель переглядатиме весь навчальний набір даних 100 разів під час навчання.

Перевірка результатів навчання:

```
loss_and_metrics = basic_model.evaluate(X_test, y_test)
print(loss_and_metrics)
print('Loss = ',loss_and_metrics[0])
print('Accuracy = ',loss_and_metrics[1])
```

Метод evaluate використовується для оцінки того, наскільки добре навчена модель працює на тестовому наборі даних (X_test, y_test). Він обчислює втрати (часто ту саму функцію втрат, яка використовується під час навчання) і будь-які визначені показники (наприклад, точність) для прогнозів моделі на тестових даних.

Прогноз за навченою моделлю виконується з використанням методу predict:

```
predicted = basic_model.predict(X_test)
```

Метод predict використовується для створення прогнозів з моделі на основі вхідних даних (X_test у цьому випадку). Вихід (predicted) міститиме передбачення моделі для кожної точки даних у наборі навчальних даних.

При використанні мінімального набора даних, прогнозування теж виконано за тестовим набором даних. Однак рекомендовано розділити частину набору даних (скажімо, 10%) для використання як набору даних перевірки.

Оцінка ефективності прогнозування

Одним із поширених інструментів, що використовуються для оцінки моделей класифікації, є матриця помилок.

У задачі бінарної класифікації (два класи, наприклад, «позитивний» і «негативний») матриця помилок зазвичай виглядає так:

	Прогнозовано негативно (0)	Прогнозований позитивний (1)
Фактично негативний (0)	Справжній негатив	Помилковий позитивний результат
Фактично позитивний (1)	Помилково негативний	Справжній позитив

Нижче наведено приклад коду для побудови матриці помилок на основі прогнозованих даних навченої моделі:

```
predicted = tf.squeeze(predicted)
predicted = np.array([1 if x >= 0.5 else 0 for x in predicted])
actual = np.array(y_test)
conf_mat = confusion_matrix(actual, predicted)
displ = ConfusionMatrixDisplay(confusion_matrix=conf_mat)
displ.plot()
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7e2dc622e2c0>
```

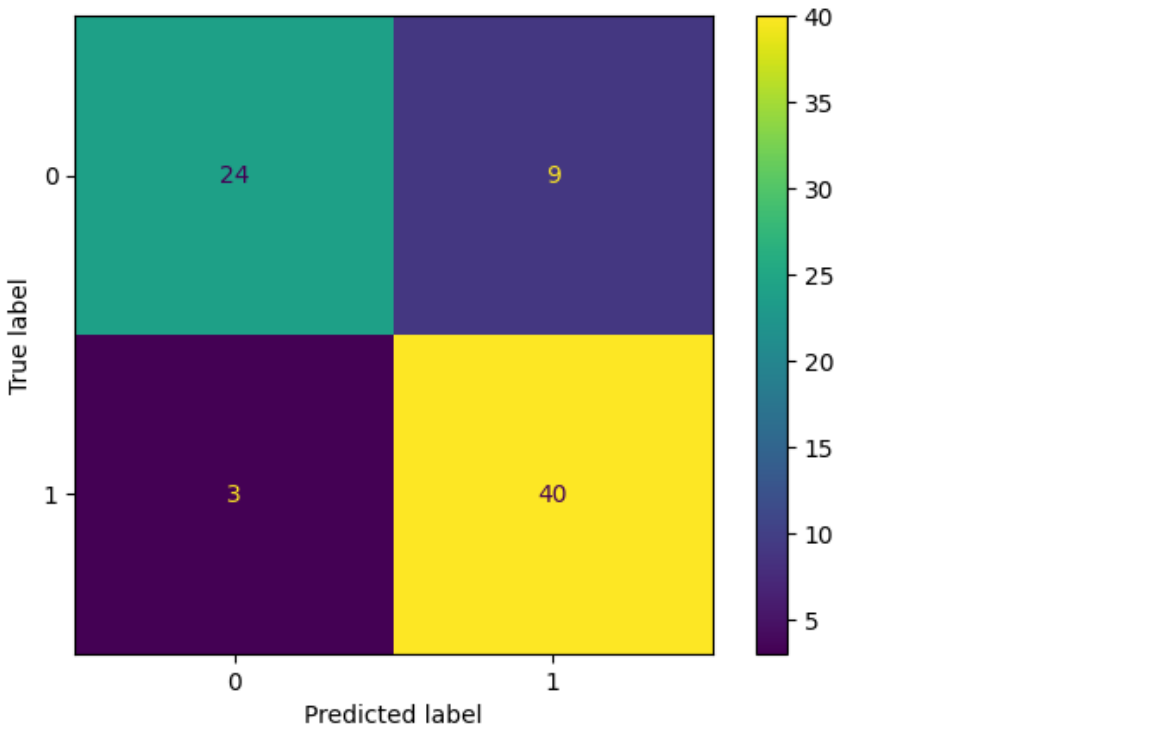

Рис. 5 - Матриця помилок для прогнозованого результату

Ось кілька ключових факторів, які слід враховувати під час роботи зі складними наборами даних.

1. Комплексна попередня обробка даних
2. Розширене кодування даних
3. Розуміння кореляції даних
4. Кілька рівнів нейронної мережі
5. Інженерія ознак
6. Регуляризація

2 Завдання

1. Модель в прикладі містить лише 2 шари – вхідний і вихідний. Додати один або декілька прихованих шарів. Як це вплинуло на результат навчання моделі?
2. Перетворити моделі з прихованими шарами у форму функціонального

API.

3. Перетворити моделі з прихованими шарами у форму субкласа Model.
4. Створити власний набір даних для бінарної класифікації, перевірити розроблені моделі на новому датасеті.

3 Контрольні питання

1. Що таке Keras і нащо він потрібен при роботі з Tensotflow?
2. Які шари зазвичай використовують для побудови моделей в Keras?
3. Які API побудови моделей використовуються в TensorFlow?
4. Для яких цілей використовується шар Dense?
5. Для яких цілей використовується шар Flatten?
6. Для яких цілей використовується шар Dropout?
7. Як підготувати дані для навчання моделі?
8. Як навчити модель в Tensorflow/Keras?
9. Як перевірити навчену модель?
10. Як виконати прогноз за навченою моделлю?