

**Лабораторна робота**

**Тема: «CIFAR-100 Розпізнавання зображень за допомогою  
згорткової нейронної мережі»**

Виконав студент групи 2КН-24м

Лавров Михайло Васильович

Перевірив викладач

Колесницький Олег Костянтинович

**Мета:** написати комп'ютерну програму мовою Python, що створює та навчає згорткову нейронну мережу для розпізнавання зображень з набору даних CIFAR-100.

### **Вихідні дані:**

- Мова програмування: Python 3.8+ (у вищих версія деякі функції бібліотек можуть бути змінені, їх можна виправити з допомогою інтернету та підказок стек трейсу помилки).
- Бібліотеки: Keras, NumPy, Pandas, Tensorflow.
- Середовище розробки Intelij IDEA (або будь яке інше).
- Відеокарта NVIDIA+CUDA (для пришвидшеного навчання моделі).
- Вхідні дані для нейронної мережі: CIFAR-100 набір даних.
- 9 шарова згорткова нейронна мережа.

### **Теоретичні відомості:**

CIFAR-100 — це набір даних, який зазвичай використовується в машинному навчанні та дослідженнях комп'ютерного зору, зокрема для завдань класифікації зображень. Це розширена версія набору даних CIFAR-10 і містить:

- 100 класів зображень (порівняно з 10 класами CIFAR-10).
- 600 зображень на клас із загальною кількістю 60 000 зображень.
- Роздільна здатність кожного зображення: 32x32 пікселів, кольорове (RGB).

### **Структура класу**

Класи в CIFAR-100 є більш дрібнозернистими та організовані в 20 суперкласів, причому кожен суперклас містить 5 дрібніших підкласів. Наприклад, суперклас «великі хижаки» може включати такі класи, як «ведмідь», «леопард», «лев», «тигр» і «вовк».

Кожне зображення в CIFAR-100 має:

- Прекрасна мітка: конкретна мітка класу (наприклад, «ведмідь» або «тигр»).
- Груба мітка: ширша мітка суперкласу (наприклад, «великі хижаки»).

### **Призначення**

Завдяки своїй детальній природі CIFAR-100 часто використовується для порівняння здатності алгоритмів виконувати складніші завдання класифікації порівняно з CIFAR-10, оскільки він вимагає розрізнення між більшою кількістю класів і більш тонкими функціями.

Згортокова нейронна мережа (convolutional neural network) — це клас глибокої нейронної мережі, який зазвичай використовується для аналізу зображень під час їх роботи добре вилучати важливі характеристики із зображень. Допомагають початкові шари згортокової нейронної мережі у вилученні простих ознак із зображень і складність вилучення ознак зростає, коли ми рухаємося до вихідного шару від вхідного шару.

Архітектура ConvNet цієї моделі містить три стеки рівнів CONV-RELU, за якими слідує рівень POOL, а потім два повністю підключених (Fully Connected - FC) шари RELU, за якими слідує повністю підключений вихідний рівень. Це одна з вдалих комбінацій шарів для більшої та глибшої нейронної мережі, оскільки кілька стеків шарів CONV-RELU допомагають витягувати складніші характеристики вхідного зображення перед виконанням операції об'єднання. Ця 9-шарова мережа допомогла отримати хорошу точність не лише на навчальному наборі, але й на тестовому наборі.

Для введення нелінійності в модель, Rectified Linear Unit (ReLU) використовувався як функція активації для прихованих шарів, оскільки він є розрідженим і зменшує ймовірність проблеми зникнення градієнта. ReLU показав хорошу продуктивність конвергенції та також був ефективним з точки зору обчислень. На вихідному рівні була використана функція активації Softmax, щоб гарантувати, що підсумкова сума активацій дорівнює 1 і, таким чином, відповідає обмеженням щільності ймовірності. Пізніше ці ймовірності допомогли в аналізі прогнозу моделі для деяких нових випадкових зображень.

У модель було додано агрегувальні шари (pooling), щоб зменшити просторовий розмір представлення (зменшення дискретизації), що, у свою чергу, зменшило кількість параметрів і вартість обчислень і, таким чином, допомогло контролювати проблему надмірного підгонки. Набір даних CIFAR-100 містить зображення низької якості, тому була необхідна техніка об'єднання, яка могла б отримати максимальну кількість характеристик із цих зображень. Оскільки операція максимального об'єднання обчислює максимальне або найбільше значення в кожній ділянці карти об'єктів, те саме було використано для зменшення дискретизації зображення та виділення найпоширенішої об'єкти в кожній ділянці карти об'єктів.

Техніка відсіву (dropout) також використовувалася для запобігання моделі від overfitting. Оскільки виходи з шарів, що беруть участь у випаданні, випадково відбираються таким чином, це запобігло переобладнанню, яке може статися через залучення мільйонів параметрів у навчання а глибока нейронна мережа. Модель, створена для цього проекту, має приблизно 13,8 мільйона параметрів, тож у нього було багато шансів надмірного оснащення, якого вдалося уникнути за допомогою відсіву.

Алгоритм оптимізації Адама використовувався в модель для оптимізації, оскільки вона допомогла моделі швидше сходиться, а отже, була більш ефективною з точки зору обчислень ніж інші алгоритми оптимізації. Також потрібно було менше пам'ять і добре працював для моделі, а також дав хороші результати завдяки дуже незначній настройці його гіперпараметра.

Теоретичні відомості взяти з статті **CIFAR-100: Object Recognition**, Chetna Khanna, Professor Jerome J. Braun. З повною версією якої можна ознайомитись за посиланням:

[https://github.com/chetnakhanna16/CIFAR100\\_ImageRecognition?tab=readme-ov-file](https://github.com/chetnakhanna16/CIFAR100_ImageRecognition?tab=readme-ov-file)

## Хід роботи (написання програми):

### 1. Створити папку проекту **network** та завантажити необхідні бібліотеки:

```
pip install keras # для створення моделі
pip install pandas # для обробки даних
pip install numpy # для обробки даних
pip install pickle # для завантаження даних
pip install matplotlib # для роботи з зображеннями
pip install pylab # для роботи з зображеннями
```

Для швидкого навчання моделі (30-60 хв) потрібно налаштувати навчання з допомогою GPU (відеокарти), тому що по замовчуванню буде використовуватись CPU та навчання буде набагато повільніше та може бути гіршим за результатами.

Потрібно:

- 1) мати NVIDIA відеокарту (AMD може не мати цієї можливості, але це можна перевірити в інтернеті та спробувати налаштувати).
- 2) Встановити CUDA, є різні способи це зробити, один з них (також можливо потрібно перезавантажити ПК після встановлення):

```
conda install -c conda-forge cudatoolkit=11.2 cudnn=8.1.0
```

- 3) Встановити GPU версію tensorflow бібліотеки:

```
python3 -m pip install tensorflow[and-cuda]
```

<https://www.tensorflow.org/install/pip#linux>

- 4) Перевірити використання GPU:

```
python3 -c "import tensorflow as
tf;print(tf.config.list_physical_devices('GPU'))"
```

Оригінал інструкції: <https://stackoverflow.com/questions/45662253/can-i-run-keras-model-on-gpu>

### 2. Підготовка вхідних та тестових даних:

- завантажити CIFAR-100 набір даних;
- створити папку data та перемістити файли з даними у неї (файли meta, test, train);
- створити папку models для збереження файлів моделей
- створити папку img та завантажити довільні зображення для перевірки (bottle, cat, clock, lion, orange). Зображення можуть бути будь якого формату (jpg, png, jpeg);

### 3. Опишемо функції для завантаження та підготовки даних для тренування.

Створимо файл **data\_utils.py**, лістинг коду з поясненнями наведено нижче:

```

import pickle
import keras
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pylab import rcParams

# функція для завантаження даних для тренування/тестування мережі
def get_raw_data_set(file_name):
    with open(file_name, 'rb') as f:
        return pickle.load(f, encoding='latin1')

# функція для попередньої обробки даних
# та приведення їх у необхідний формат для навчання мережі
def preprocess_data(raw_data, num_class):
    # дістаємо дані
    x_data = raw_data['data']
    # перетворюємо дані
    x_data = x_data.reshape(len(x_data), 3, 32, 32).transpose(0, 2, 3, 1)
    # Зміна масштабу шляхом ділення кожного пікселя зображення на 255
    x_data = x_data / 255.
    # дістаємо конкретні надписи (ярлики)
    y_data = raw_data['fine_labels']
    # перетворюємо написи
    y_data = keras.utils.to_categorical(y_data, num_class)
    return x_data, y_data

```

#### 4. Опис функції для побудови загорткової нейронної мережі. Створимо файл **model\_utils.py**. Лістинг коду з коментарями:

```

from keras.layers import Conv2D, MaxPool2D, Dropout, Flatten, Dense
from keras.models import Sequential

# Функція для побудови згорткової нейронної мережі
def build_model(input_shape, num_class):
    # Ініціалізація моделі послідовного типу (має лінійний стек шарів)
    model = Sequential()

    # Стек 1
    # Згорткові шари (convolution)
    model.add(Conv2D(filters=128, kernel_size=3, padding="same", activation="relu",
input_shape=input_shape))
    model.add(Conv2D(filters=128, kernel_size=3, padding="same", activation="relu"))
    # Агрегувальні шари (pooling)
    model.add(MaxPool2D(pool_size=2, strides=2))
    # Виключення з'єднань (dropout)
    model.add(Dropout(0.2))

    # Стек 2
    # Згорткові шари (convolution)
    model.add(Conv2D(filters=256, kernel_size=3, padding="same", activation="relu"))
    model.add(Conv2D(filters=256, kernel_size=3, padding="same", activation="relu"))
    # Агрегувальні шари (pooling)
    model.add(MaxPool2D(pool_size=2, strides=2))
    # Виключення з'єднань (dropout)
    model.add(Dropout(0.5))

    # Стек 3
    # Згорткові шари (convolution)
    model.add(Conv2D(filters=512, kernel_size=3, padding="same", activation="relu"))
    model.add(Conv2D(filters=512, kernel_size=3, padding="same", activation="relu"))
    # Агрегувальні шари (pooling)

```

```

model.add(MaxPool2D(pool_size=2, strides=2))
# Виключення з'єднань (dropout)
model.add(Dropout(0.5))

# Згладжування/розрівнювання (flattening)
model.add(Flatten())
# Повноз'єднані шари (fully connected layers)
model.add(Dense(units=1000, activation="relu"))
# Виключення з'єднань (dropout)
model.add(Dropout(0.5))
# Повноз'єднані шари (fully connected layers)
model.add(Dense(units=1000, activation="relu"))
# Виключення з'єднань (dropout)
model.add(Dropout(0.5))
# Вихідний шар (output layer)
model.add(Dense(units=num_class, activation="softmax"))
model.summary()
return model

```

5. Навчання мережі та візуалізація результатів. Для цього створимо файл **train\_model.py**, лістинг коду та описом:

```

import keras
import matplotlib.pyplot as plt
from keras.callbacks import EarlyStopping, ModelCheckpoint

from network.data_utils import get_raw_data_set, preprocess_data
from network.model_utils import build_model

##### Оголошення змінних та початкових даних

# Кількість класів у наборі даних CIFAR-100
num_class = 100
# Кількість епох для навчання мережі
epochs = 100
# Розмір партії під час навчання
batch_size = 64

# завантаження даних для тренування, тестування та метадані
trainData = get_raw_data_set('data/train')
testData = get_raw_data_set('data/test')
metaData = get_raw_data_set('data/meta')

# ініціалізація даних у потрібному для навчання моделі форматі
# x - дані про малюнок
# y - дані про відповідний клас (ярлик) малюнку
x_train, y_train = preprocess_data(trainData, num_class)
x_test, y_test = preprocess_data(testData, num_class)

# створення моделі нейронної мережі
model = build_model(x_train.shape[1:], num_class)

##### Навчання згорткової нейронної мережі

# створення оптимізатора
optimizer = keras.optimizers.Adam(lr=0.0001)

# компіляція моделі
model.compile(optimizer=optimizer, loss='categorical_crossentropy',
metrics=['accuracy'])

```

```

# рання зупинка для моніторингу втрат підтвердження та уникнення переобладнання
early_stop = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=10)

# збереження контрольної точки моделі для найкращої моделі
model_checkpoint = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min',
save_best_only=True, verbose=1)

# підключення даних до моделі та початок навчання з відображенням історії
model_history = model.fit(x=x_train,
                           y=y_train,
                           batch_size=batch_size,
                           epochs=epochs,
                           verbose=1,
                           callbacks=[early_stop, model_checkpoint],
                           validation_split=0.2,
                           steps_per_epoch=40000 // batch_size,
                           validation_steps=10000 // batch_size,
                           validation_batch_size=batch_size)

##### Візуалізація втрат і точності
# Графік для візуалізації втрат і точності залежно від кількості епох
plt.figure(figsize=(18, 8))
plt.suptitle('Loss and Accuracy Plots', fontsize=18)
plt.subplot(1, 2, 1)
plt.plot(model_history.history['loss'], label='Training Loss')
plt.plot(model_history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.xlabel('Number of epochs', fontsize=15)
plt.ylabel('Loss', fontsize=15)
plt.subplot(1, 2, 2)
plt.plot(model_history.history['accuracy'], label='Train Accuracy')
plt.plot(model_history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.xlabel('Number of epochs', fontsize=14)
plt.ylabel('Accuracy', fontsize=14)
plt.show()

```

## 6. Створимо файл **evaluate\_model.py** для оцінки якості розпізнання. Лістинг з поясненнями:

```

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from keras.models import load_model
from pylab import rcParams
from sklearn.metrics import confusion_matrix, classification_report

from network.data_utils import get_raw_data_set, preprocess_data

##### Оголошення змінних та початкових даних

# Кількість класів у наборі даних CIFAR-100
num_class = 100
# Розмір партії для розпізнання
batch_size = 64
# Розмір масиву даних для розпізнання
# повинен бути меншим за 10000 для test, або меншим за 50000 для train)
data_set_size = 10000

# завантаження даних з файлу
testData = get_raw_data_set('data/test')

```



```

metaData = get_raw_data_set('data/meta')

# ініціалізація даних у потрібному для моделі форматі
# x - дані про малюнок
# y - дані про відповідний клас (ярлик) малюнку
x_test_full, y_test_full = preprocess_data(testData, num_class)

# завантаження моделі з файлу
model = load_model('models/best_model.h5')

# завантаження категорій (класів) зображень
category = pd.DataFrame(metaData['coarse_label_names'], columns=['SuperClass'])
subCategory = pd.DataFrame(metaData['fine_label_names'], columns=['SubClass'])

# беремо частину даних з масиву
x_test = x_test_full[:data_set_size]
y_test = y_test_full[:data_set_size]
fine_labels = testData['fine_labels'][:data_set_size]

# визначення точності розпізнання даних
test_loss, test_accuracy = model.evaluate(x=x_test,
                                          y=y_test,
                                          batch_size=batch_size,
                                          steps=data_set_size // batch_size)

print('Accuracy: ', round((test_accuracy * 100), 2), "%")
print('Loss: ', round(test_loss, 2))

# запускаємо модель для розпізнання даних
y_pred = model.predict(x_test)
cm = confusion_matrix(np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1))
print(cm)

# звіт, щоб побачити, яку категорію було передбачено неправильно, а яку - правильно
target = ["Category {}".format(i) for i in range(num_class)]
print(classification_report(np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1),
                           target_names=target))

##### Візуалізація прогнозів
prediction = np.argmax(y_pred, axis=1)
prediction = pd.DataFrame(prediction)

# генерування випадкового числа для відображення випадкового зображення з набору
даних разом із істинною та прогнозованою міткою
# 16 випадкових зображень для одночасного відображення разом із їхніми справжніми та
випадковими мітками
rcParams['figure.figsize'] = 12, 15

num_row = 4
num_col = 4

imageId = np.random.randint(0, len(x_test), num_row * num_col)

fig, axes = plt.subplots(num_row, num_col)

for i in range(0, num_row):
    for j in range(0, num_col):
        k = (i * num_col) + j
        axes[i, j].imshow(x_test[imageId[k]])
        axes[i, j].set_title("True: " +
                             str(subCategory.iloc[fine_labels[imageId[k]]][0]).capitalize()
                             + "\nPredicted: " +
                             str(subCategory.iloc[prediction.iloc[imageId[k]]]).split()[
                                 2].capitalize(),
                             fontsize=14)

```

```

        axes[i, j].axis('off')
        fig.suptitle("Images with True and Predicted Labels", fontsize=18)

plt.show()
print()

```

## 7. Створимо файл **test\_model.py** для перевірки розпізнання довільних зображень. Лістинг з поясненнями:

```

import cv2
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from keras.models import load_model

from network.data_utils import get_raw_data_set

# модель та категорії (класи) зображень
model = None
subCategory = None

# функція для зміни розміру зображення
def resize_test_image(test_img):
    img = cv2.imread(test_img)
    # plt.imshow(img)
    # plt.show()
    img_RGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # plt.imshow(img_RGB)
    # plt.show()
    resized_img = cv2.resize(img_RGB, (32, 32))
    # plt.imshow(resized_img)
    # plt.show()
    resized_img = resized_img / 255.
    # plt.imshow(resized_img)
    # plt.show()
    return resized_img

# функція для отримання прогнозу для тестового зображення з моделі
def predict_test_image(test_img):
    resized_img = resize_test_image(test_img)
    prediction = model.predict(np.array([resized_img]))
    return prediction

# функція для отримання відсортованого прогнозу
def sort_prediction_test_image(test_img):
    prediction = predict_test_image(test_img)
    index = np.arange(0, 100)
    for i in range(100):
        for j in range(100):
            if prediction[0][index[i]] > prediction[0][index[j]]:
                temp = index[i]
                index[i] = index[j]
                index[j] = temp
    return index

```

```

# функція для отримання фрейму даних для 5 найкращих прогнозів
def df_top5_prediction_test_image(test_img):
    sorted_index = sort_prediction_test_image(test_img)
    prediction = predict_test_image(test_img)
    subCategory_name = []
    prediction_score = []
    k = sorted_index[:6]
    for i in range(len(k)):
        subCategory_name.append(subCategory.iloc[k[i]][0])
        prediction_score.append(round(prediction[0][k[i]], 2))

    df = pd.DataFrame(list(zip(subCategory_name, prediction_score)),
columns=['Label', 'Probability'])
    return df

# функція для отримання графіку для 5 найкращих прогнозів
def plot_top5_prediction_test_image(test_img):
    fig, axes = plt.subplots(1, 2, figsize=(15, 4))
    fig.suptitle("Prediction", fontsize=18)
    new_img = plt.imread(test_img)
    axes[0].imshow(new_img)
    axes[0].axis('off')
    data = df_top5_prediction_test_image(test_img)
    x = data['Label']
    y = data['Probability']
    axes[1] = sns.barplot(x=x, y=y, data=data, color="green")
    plt.xlabel('Label', fontsize=14)
    plt.ylabel('Probability', fontsize=14)
    plt.ylim(0, 1.0)
    axes[1].grid(False)
    axes[1].spines["top"].set_visible(False)
    axes[1].spines["right"].set_visible(False)
    axes[1].spines["bottom"].set_visible(False)
    axes[1].spines["left"].set_visible(False)
    plt.show()

# завантаження моделі з файлу
model = load_model('models/best_model.h5')

# завантаження мета даних з файлу та назв категорій (класів) зображень
metaData = get_raw_data_set('data/meta')
subCategory = pd.DataFrame(metaData['fine_label_names'], columns=['SubClass'])

# розпізнання зображення та показ результатів
plot_top5_prediction_test_image('img/orange.png')
plot_top5_prediction_test_image('img/orchid.png')
plot_top5_prediction_test_image('img/cat.png')
plot_top5_prediction_test_image('img/lion.png')
plot_top5_prediction_test_image('img/clock.jpg')
plot_top5_prediction_test_image('img/bottle.jpg')

```

## Хід роботи (запуск та перевірка програми):

1. Потрібно запустити файл **train\_model.py**, який завантажить тренувальні дані, створить модель, запустить процес навчання та збереже результати у файл. Навчання даної мережі може тривати 30-60 хв, в залежності від відеокарти. У моєму випадку це було 25 хв (NVIDIA GeForce GTX 1060 3GB).

Вигляд даних до та після попередньої обробки (функція **preprocess\_data**):

**x\_data = raw\_data['data']**

```
10 data_size = {int} 50000
01
10 num_class = {int} 100
01
▼ raw_data = {dict: 5} {'batch_label': 'training batch 1 of 1', 'coarse_labels': [11, 15, 4, 14, 1, 5, 18, 3, 10, 11, 5, 17, 2, 9, ... View
> i='filenames' = {list: 50000} ['bos_taurus_s_000507.png', 'stegosaurus_s_000125.png', 'mcintosh_s_000643.png', ... View
10 'batch_label' = {str} 'training batch 1 of 1'
01
> i='fine_labels' = {list: 50000} [19, 29, 0, 11, 1, 86, 90, 28, 23, 31, 39, 96, 82, 17, 71, 39, 8, 97, 80, 71, 74, 59, 70, 87... View
> i='coarse_labels' = {list: 50000} [11, 15, 4, 14, 1, 5, 18, 3, 10, 11, 5, 17, 2, 9, 10, 5, 18, 8, 16, 10, 16, 17, 2, 5, 17, 6, 12... View
> i='data' = {ndarray: (50000, 3072)} [[255 255 255 ... 10 59 79], [255 253 253 ... 253 253 255], [250 241...View as Array
10 __len__ = {int} 5
01
> Protected Attributes
> i=x_data = {ndarray: (50000, 3072)} [[255 255 255 ... 10 59 79], [255 253 253 ... 253 253 255], [250 248 ...View as Array
```

**x\_data = x\_data.reshape(data\_size, 3, 32, 32).transpose(0, 2, 3, 1)**

```
▼ i=x_data = {ndarray: (50000, 32, 32, 3)} [[[[255 255 255], [255 255 255], [255 255 255], ..., [195 205 1...View as Array
10 min = {str} 'ndarray too big, calculating min would slow down debugging'
01
10 max = {str} 'ndarray too big, calculating max would slow down debugging'
01
> i=shape = {tuple: 4} (50000, 32, 32, 3)
> i=dtype = {dtype[uint8]: ()} uint8
10 size = {int} 153600000
01
> i=array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContair
> Protected Attributes
```

	0	1	2
11	255	255	255
12	255	255	255
13	255	255	255
14	255	255	255
15	255	255	255
16	255	255	255
17	255	252	248
18	231	222	213
19	176	168	163
20	237	236	235
21	255	255	255
22	255	255	255
23	255	255	255
24	255	255	255
25	255	255	255
26	252	255	255
27	242	252	252
28	229	240	234
29	195	205	193
30	212	224	204
31	182	194	167

x\_data[0][0] Format: %d

☒ Colored cells

☒ Resize automatically

Close

**x\_data = x\_data / 255.**

	0	1	2
11	1.00000	1.00000	1.00000
12	1.00000	1.00000	1.00000
13	1.00000	1.00000	1.00000
14	1.00000	1.00000	1.00000
15	1.00000	1.00000	1.00000
16	1.00000	1.00000	1.00000
17	1.00000	0.98824	0.97255
18	0.90588	0.87059	0.83529
19	0.69020	0.65882	0.63922
20	0.92941	0.92549	0.92157
21	1.00000	1.00000	1.00000
22	1.00000	1.00000	1.00000
23	1.00000	1.00000	1.00000
24	1.00000	1.00000	1.00000
25	1.00000	1.00000	1.00000
26	0.98824	1.00000	1.00000
27	0.94902	0.98824	0.98824
28	0.89804	0.94118	0.91765
29	0.76471	0.80392	0.75686
30	0.83137	0.87843	0.80000
31	0.71373	0.76078	0.65490

x\_data[0][0] Format: %.5f

☒ Colored cells

☒ Resize automatically

Close

**y\_data = raw\_data['fine\_labels']**

```

y_data = {list: 50000} [19, 29, 0, 11, 1, 86, 90, 28, 23, 31, 39, 96, 82, 17, 71, 39, 8, 97, 80, 71, 74, 59, 70, 87, 59, 84, ... View
  10 00000 = {int} 19
  01
  10 00001 = {int} 29
  01
  10 00002 = {int} 0
  01
  10 00003 = {int} 11
  01
  10 00004 = {int} 1
  01
  10 00005 = {int} 86
  01
  10 00006 = {int} 90
  01
  10 00007 = {int} 28
  01
  10 00008 = {int} 23
  01
  10 00009 = {int} 31
  01
  10 00010 = {int} 39
  01

```

**y\_data = keras.utils.to\_categorical(y\_data, num\_class)**

```

y_data = {ndarray: (50000, 100)} [[0. 0. 0. ... 0. 0. 0.], [0. 0. 0. ... 0. 0. 0.], [1. 0. 0. ... 0. 0. 0.], ..., [0. 0. 0. ... View as Array
  10 min = {str} 'ndarray too big, calculating min would slow down debugging'
  01
  10 max = {str} 'ndarray too big, calculating max would slow down debugging'
  01
> shape = {tuple: 2} (50000, 100)
> dtype = {dtype[float32]: ()} float32
  10 size = {int} 5000000
  01
> array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContain
> Protected Attributes

```

y\_data



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
2	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000
4	0.00000	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
9	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
10	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
11	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
12	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
13	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
14	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
15	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000	0.00000
17	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
18	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
19	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
20	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

y\_data

☒ Colored cells

☒ Resize automatically

Інформація про назви класів та суперкласів є в metaData:

```
▼  metaData = {dict: 2} {'coarse_label_names': ['aquatic_mammals', 'fish', 'flowers', 'food_containers', 'fruit_anc... View
  >  'fine_label_names' = {list: 100} ['apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee', 'beetle', 'bicyc... View
  >  'coarse_label_names' = {list: 20} ['aquatic_mammals', 'fish', 'flowers', 'food_containers', 'fruit_and_vegeta... View
    10 __len__ = {int} 2

['apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee', 'beetle', 'bicycle', 'bottle', 'bowl', 'boy', '
  ↗ 'bridge', 'bus', 'butterfly', 'camel', 'can', 'castle', 'caterpillar', 'cattle', 'chair', 'chimpanzee', 'clock',
  ↗ 'cloud', 'cockroach', 'couch', 'crab', 'crocodile', 'cup', 'dinosaur', 'dolphin', 'elephant', 'flatfish',
  ↗ 'forest', 'fox', 'girl', 'hamster', 'house', 'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
  ↗ 'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain', 'mouse', 'mushroom', 'oak_tree', 'orange',
  ↗ 'orchid', 'otter', 'palm_tree', 'pear', 'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy', 'porcupine',
  ↗ 'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket', 'rose', 'sea', 'seal', 'shark', 'shrew', 'skunk',
  ↗ 'skyscraper', 'snail', 'snake', 'spider', 'squirrel', 'streetcar', 'sunflower', 'sweet_pepper', 'table', 'tank',
  ↗ 'telephone', 'television', 'tiger', 'tractor', 'train', 'trout', 'tulip', 'turtle', 'wardrobe', 'whale',
  ↗ 'willow_tree', 'wolf', 'woman', 'worm']

['aquatic_mammals', 'fish', 'flowers', 'food_containers', 'fruit_and_vegetables', 'household_electrical_devices',
  ↗ 'household_furniture', 'insects', 'large_carnivores', 'large_man-made_outdoor_things',
  ↗ 'large_natural_outdoor_scenes', 'large_omnivores_and_herbivores', 'medium_mammals', 'non-insect_invertebrates',
  ↗ 'people', 'reptiles', 'small_mammals', 'trees', 'vehicles_1', 'vehicles_2']
```

Результат після створення моделі (**build\_model**):  
**model = build\_model(x\_train, num\_class)**

Model: "sequential"		
-----		
Layer (type)	Output Shape	Param #
-----		
conv2d (Conv2D)	(None, 32, 32, 128)	3584
conv2d_1 (Conv2D)	(None, 32, 32, 128)	147584
max_pooling2d (MaxPooling2D)	(None, 16, 16, 128)	0
dropout (Dropout)	(None, 16, 16, 128)	0
conv2d_2 (Conv2D)	(None, 16, 16, 256)	295168
conv2d_3 (Conv2D)	(None, 16, 16, 256)	590080
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 256)	0
dropout_1 (Dropout)	(None, 8, 8, 256)	0
conv2d_4 (Conv2D)	(None, 8, 8, 512)	1180160
conv2d_5 (Conv2D)	(None, 8, 8, 512)	2359808
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 512)	0
-----		
dropout_2 (Dropout)	(None, 4, 4, 512)	0
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 1000)	8193000
dropout_3 (Dropout)	(None, 1000)	0
dense_1 (Dense)	(None, 1000)	1001000
dropout_4 (Dropout)	(None, 1000)	0
dense_2 (Dense)	(None, 100)	100100
-----		
Total params: 13870484 (52.91 MB)		
Trainable params: 13870484 (52.91 MB)		
Non-trainable params: 0 (0.00 Byte)		
-----		

Результат під час навчання моделі:

```
Epoch 1/100
625/625 [=====] - ETA: 0s - loss: 4.4953 - accuracy: 0.0204
Epoch 1: val_loss improved from inf to 4.26972, saving model to best_model.h5
625/625 [=====] - 39s 56ms/step - loss: 4.4953 - accuracy: 0.0204 - val_loss:
4.2697 - val_accuracy: 0.0458
Epoch 2/100
625/625 [=====] - ETA: 0s - loss: 4.1720 - accuracy: 0.0483
```

```
Epoch 2: val_loss improved from 4.26972 to 3.97379, saving model to best_model.h5
625/625 [=====] - 35s 55ms/step - loss: 4.1720 - accuracy: 0.0483 - val_loss:
3.9738 - val_accuracy: 0.0834
Epoch 3/100
625/625 [=====] - ETA: 0s - loss: 3.8778 - accuracy: 0.0913
Epoch 3: val_loss improved from 3.97379 to 3.60587, saving model to best_model.h5
625/625 [=====] - 36s 57ms/step - loss: 3.8778 - accuracy: 0.0913 - val_loss:
3.6059 - val_accuracy: 0.1488
Epoch 4/100
625/625 [=====] - ETA: 0s - loss: 3.6024 - accuracy: 0.1386
Epoch 4: val_loss improved from 3.60587 to 3.32255, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 3.6024 - accuracy: 0.1386 - val_loss:
3.3225 - val_accuracy: 0.1951
Epoch 5/100
625/625 [=====] - ETA: 0s - loss: 3.3583 - accuracy: 0.1779
Epoch 5: val_loss improved from 3.32255 to 3.18945, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 3.3583 - accuracy: 0.1779 - val_loss:
3.1894 - val_accuracy: 0.2261
Epoch 6/100
625/625 [=====] - ETA: 0s - loss: 3.1418 - accuracy: 0.2179
Epoch 6: val_loss improved from 3.18945 to 2.90718, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 3.1418 - accuracy: 0.2179 - val_loss:
2.9072 - val_accuracy: 0.2702
Epoch 7/100
625/625 [=====] - ETA: 0s - loss: 2.9451 - accuracy: 0.2577
Epoch 7: val_loss improved from 2.90718 to 2.73023, saving model to best_model.h5
625/625 [=====] - 35s 55ms/step - loss: 2.9451 - accuracy: 0.2577 - val_loss:
2.7302 - val_accuracy: 0.3127
Epoch 8/100
625/625 [=====] - ETA: 0s - loss: 2.7791 - accuracy: 0.2913
Epoch 8: val_loss improved from 2.73023 to 2.57866, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 2.7791 - accuracy: 0.2913 - val_loss:
2.5787 - val_accuracy: 0.3382
Epoch 9/100
625/625 [=====] - ETA: 0s - loss: 2.6357 - accuracy: 0.3173
Epoch 9: val_loss improved from 2.57866 to 2.45438, saving model to best_model.h5
625/625 [=====] - 36s 58ms/step - loss: 2.6357 - accuracy: 0.3173 - val_loss:
2.4544 - val_accuracy: 0.3659
Epoch 10/100
625/625 [=====] - ETA: 0s - loss: 2.5013 - accuracy: 0.3478
Epoch 10: val_loss improved from 2.45438 to 2.39349, saving model to best_model.h5
625/625 [=====] - 36s 57ms/step - loss: 2.5013 - accuracy: 0.3478 - val_loss:
2.3935 - val_accuracy: 0.3796
Epoch 11/100
625/625 [=====] - ETA: 0s - loss: 2.3915 - accuracy: 0.3675
Epoch 11: val_loss improved from 2.39349 to 2.23361, saving model to best_model.h5
625/625 [=====] - 36s 58ms/step - loss: 2.3915 - accuracy: 0.3675 - val_loss:
2.2336 - val_accuracy: 0.4098
Epoch 12/100
625/625 [=====] - ETA: 0s - loss: 2.2941 - accuracy: 0.3929
Epoch 12: val_loss improved from 2.23361 to 2.18320, saving model to best_model.h5
625/625 [=====] - 37s 58ms/step - loss: 2.2941 - accuracy: 0.3929 - val_loss:
2.1832 - val_accuracy: 0.4233
Epoch 13/100
625/625 [=====] - ETA: 0s - loss: 2.2006 - accuracy: 0.4121
Epoch 13: val_loss improved from 2.18320 to 2.14189, saving model to best_model.h5
625/625 [=====] - 36s 58ms/step - loss: 2.2006 - accuracy: 0.4121 - val_loss:
2.1419 - val_accuracy: 0.4338
Epoch 14/100
625/625 [=====] - ETA: 0s - loss: 2.1165 - accuracy: 0.4295
Epoch 14: val_loss improved from 2.14189 to 2.04757, saving model to best_model.h5
625/625 [=====] - 36s 57ms/step - loss: 2.1165 - accuracy: 0.4295 - val_loss:
2.0476 - val_accuracy: 0.4555
Epoch 15/100
625/625 [=====] - ETA: 0s - loss: 2.0385 - accuracy: 0.4516
Epoch 15: val_loss improved from 2.04757 to 2.04400, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 2.0385 - accuracy: 0.4516 - val_loss:
2.0440 - val_accuracy: 0.4559
Epoch 16/100
625/625 [=====] - ETA: 0s - loss: 1.9547 - accuracy: 0.4669
Epoch 16: val_loss improved from 2.04400 to 1.97365, saving model to best_model.h5
```



```
625/625 [=====] - 35s 56ms/step - loss: 1.9547 - accuracy: 0.4669 - val_loss: 1.9736 - val_accuracy: 0.4728
Epoch 17/100
625/625 [=====] - ETA: 0s - loss: 1.8817 - accuracy: 0.4822
Epoch 17: val_loss improved from 1.97365 to 1.93098, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.8817 - accuracy: 0.4822 - val_loss: 1.9310 - val_accuracy: 0.4842
Epoch 18/100
625/625 [=====] - ETA: 0s - loss: 1.8034 - accuracy: 0.4989
Epoch 18: val_loss improved from 1.93098 to 1.88203, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.8034 - accuracy: 0.4989 - val_loss: 1.8820 - val_accuracy: 0.4897
Epoch 19/100
625/625 [=====] - ETA: 0s - loss: 1.7434 - accuracy: 0.5185
Epoch 19: val_loss improved from 1.88203 to 1.85258, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.7434 - accuracy: 0.5185 - val_loss: 1.8526 - val_accuracy: 0.5038
Epoch 20/100
625/625 [=====] - ETA: 0s - loss: 1.6810 - accuracy: 0.5281
Epoch 20: val_loss improved from 1.85258 to 1.84182, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.6810 - accuracy: 0.5281 - val_loss: 1.8418 - val_accuracy: 0.5061
Epoch 21/100
625/625 [=====] - ETA: 0s - loss: 1.6284 - accuracy: 0.5397
Epoch 21: val_loss did not improve from 1.84182
625/625 [=====] - 36s 57ms/step - loss: 1.6284 - accuracy: 0.5397 - val_loss: 1.8464 - val_accuracy: 0.5072
Epoch 22/100
625/625 [=====] - ETA: 0s - loss: 1.5577 - accuracy: 0.5574
Epoch 22: val_loss improved from 1.84182 to 1.76817, saving model to best_model.h5
625/625 [=====] - 36s 58ms/step - loss: 1.5577 - accuracy: 0.5574 - val_loss: 1.7682 - val_accuracy: 0.5191
Epoch 23/100
625/625 [=====] - ETA: 0s - loss: 1.5022 - accuracy: 0.5727
Epoch 23: val_loss did not improve from 1.76817
625/625 [=====] - 36s 58ms/step - loss: 1.5022 - accuracy: 0.5727 - val_loss: 1.7850 - val_accuracy: 0.5216
Epoch 24/100
625/625 [=====] - ETA: 0s - loss: 1.4537 - accuracy: 0.5841
Epoch 24: val_loss improved from 1.76817 to 1.73475, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.4537 - accuracy: 0.5841 - val_loss: 1.7348 - val_accuracy: 0.5282
Epoch 25/100
625/625 [=====] - ETA: 0s - loss: 1.3982 - accuracy: 0.5994
Epoch 25: val_loss improved from 1.73475 to 1.71191, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.3982 - accuracy: 0.5994 - val_loss: 1.7119 - val_accuracy: 0.5395
Epoch 26/100
625/625 [=====] - ETA: 0s - loss: 1.3416 - accuracy: 0.6094
Epoch 26: val_loss did not improve from 1.71191
625/625 [=====] - 35s 56ms/step - loss: 1.3416 - accuracy: 0.6094 - val_loss: 1.7152 - val_accuracy: 0.5349
Epoch 27/100
624/625 [=====>.] - ETA: 0s - loss: 1.3025 - accuracy: 0.6236
Epoch 27: val_loss improved from 1.71191 to 1.70736, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.3028 - accuracy: 0.6236 - val_loss: 1.7074 - val_accuracy: 0.5407
Epoch 28/100
625/625 [=====] - ETA: 0s - loss: 1.2421 - accuracy: 0.6375
Epoch 28: val_loss improved from 1.70736 to 1.69613, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.2421 - accuracy: 0.6375 - val_loss: 1.6961 - val_accuracy: 0.5459
Epoch 29/100
625/625 [=====] - ETA: 0s - loss: 1.1941 - accuracy: 0.6496
Epoch 29: val_loss improved from 1.69613 to 1.68106, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.1941 - accuracy: 0.6496 - val_loss: 1.6811 - val_accuracy: 0.5487
Epoch 30/100
625/625 [=====] - ETA: 0s - loss: 1.1527 - accuracy: 0.6615
Epoch 30: val_loss did not improve from 1.68106
```

```
625/625 [=====] - 35s 56ms/step - loss: 1.1527 - accuracy: 0.6615 - val_loss: 1.7181 - val_accuracy: 0.5486
Epoch 31/100
625/625 [=====] - ETA: 0s - loss: 1.1177 - accuracy: 0.6689
Epoch 31: val_loss did not improve from 1.68106
625/625 [=====] - 35s 55ms/step - loss: 1.1177 - accuracy: 0.6689 - val_loss: 1.6928 - val_accuracy: 0.5554
Epoch 32/100
624/625 [=====>.] - ETA: 0s - loss: 1.0725 - accuracy: 0.6817
Epoch 32: val_loss did not improve from 1.68106
625/625 [=====] - 36s 57ms/step - loss: 1.0723 - accuracy: 0.6817 - val_loss: 1.6842 - val_accuracy: 0.5549
Epoch 33/100
625/625 [=====] - ETA: 0s - loss: 1.0363 - accuracy: 0.6905
Epoch 33: val_loss improved from 1.68106 to 1.67202, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 1.0363 - accuracy: 0.6905 - val_loss: 1.6720 - val_accuracy: 0.5583
Epoch 34/100
625/625 [=====] - ETA: 0s - loss: 1.0012 - accuracy: 0.6975
Epoch 34: val_loss did not improve from 1.67202
625/625 [=====] - 34s 55ms/step - loss: 1.0012 - accuracy: 0.6975 - val_loss: 1.6991 - val_accuracy: 0.5544
Epoch 35/100
625/625 [=====] - ETA: 0s - loss: 0.9577 - accuracy: 0.7111
Epoch 35: val_loss improved from 1.67202 to 1.65677, saving model to best_model.h5
625/625 [=====] - 35s 56ms/step - loss: 0.9577 - accuracy: 0.7111 - val_loss: 1.6568 - val_accuracy: 0.5642
Epoch 36/100
624/625 [=====>.] - ETA: 0s - loss: 0.9314 - accuracy: 0.7184
Epoch 36: val_loss did not improve from 1.65677
625/625 [=====] - 34s 55ms/step - loss: 0.9317 - accuracy: 0.7185 - val_loss: 1.6917 - val_accuracy: 0.5694
Epoch 37/100
625/625 [=====] - ETA: 0s - loss: 0.8978 - accuracy: 0.7281
Epoch 37: val_loss did not improve from 1.65677
625/625 [=====] - 35s 56ms/step - loss: 0.8978 - accuracy: 0.7281 - val_loss: 1.6899 - val_accuracy: 0.5609
Epoch 38/100
625/625 [=====] - ETA: 0s - loss: 0.8735 - accuracy: 0.7333
Epoch 38: val_loss did not improve from 1.65677
625/625 [=====] - 35s 56ms/step - loss: 0.8735 - accuracy: 0.7333 - val_loss: 1.6782 - val_accuracy: 0.5681
Epoch 39/100
625/625 [=====] - ETA: 0s - loss: 0.8367 - accuracy: 0.7449
Epoch 39: val_loss did not improve from 1.65677
625/625 [=====] - 34s 55ms/step - loss: 0.8367 - accuracy: 0.7449 - val_loss: 1.7090 - val_accuracy: 0.5653
Epoch 40/100
625/625 [=====] - ETA: 0s - loss: 0.8010 - accuracy: 0.7534
Epoch 40: val_loss did not improve from 1.65677
625/625 [=====] - 35s 56ms/step - loss: 0.8010 - accuracy: 0.7534 - val_loss: 1.6848 - val_accuracy: 0.5728
Epoch 41/100
624/625 [=====>.] - ETA: 0s - loss: 0.7855 - accuracy: 0.7572
Epoch 41: val_loss did not improve from 1.65677
625/625 [=====] - 35s 55ms/step - loss: 0.7855 - accuracy: 0.7572 - val_loss: 1.7044 - val_accuracy: 0.5734
Epoch 42/100
625/625 [=====] - ETA: 0s - loss: 0.7443 - accuracy: 0.7703
Epoch 42: val_loss did not improve from 1.65677
625/625 [=====] - 35s 55ms/step - loss: 0.7443 - accuracy: 0.7703 - val_loss: 1.7381 - val_accuracy: 0.5711
Epoch 43/100
625/625 [=====] - ETA: 0s - loss: 0.7201 - accuracy: 0.7742
Epoch 43: val_loss did not improve from 1.65677
625/625 [=====] - 35s 55ms/step - loss: 0.7201 - accuracy: 0.7742 - val_loss: 1.7136 - val_accuracy: 0.5644
Epoch 44/100
625/625 [=====] - ETA: 0s - loss: 0.6985 - accuracy: 0.7826
Epoch 44: val_loss did not improve from 1.65677
```

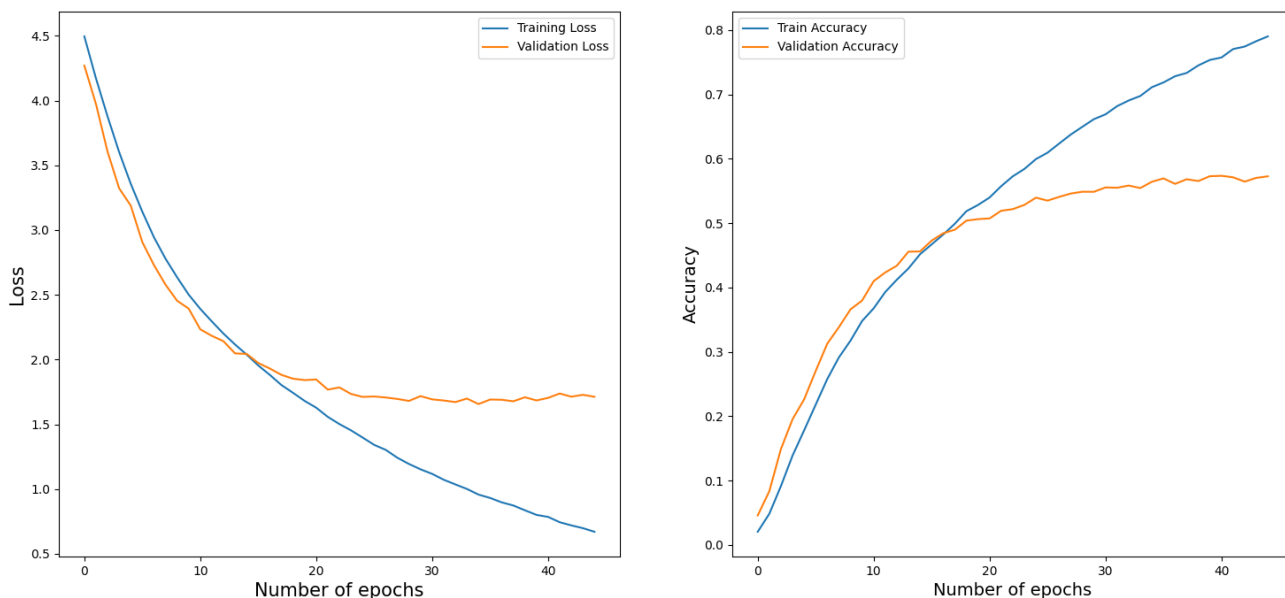
```

625/625 [=====] - 35s 55ms/step - loss: 0.6985 - accuracy: 0.7826 - val_loss:
1.7280 - val_accuracy: 0.5702
Epoch 45/100
625/625 [=====] - ETA: 0s - loss: 0.6707 - accuracy: 0.7900
Epoch 45: val_loss did not improve from 1.65677
625/625 [=====] - 35s 56ms/step - loss: 0.6707 - accuracy: 0.7900 - val_loss:
1.7129 - val_accuracy: 0.5726
Epoch 45: early stopping

```

Візуалізація втрат і точності після навчання мережі:

Loss and Accuracy Plots



2. Для оцінки результатів моделі потрібно запустити **evaluate\_model.py**.

Результат оцінки точності розпізнання:

```
test_loss, test_accuracy = model.evaluate_generator(generator=test_data_gen,
                                                    steps=data_set_size // batch_size)
```

```

Accuracy: 57.54 %
Loss: 1.63

```

Результат розпізнання набору даних (10000 зображення з тестової вибірки):

	precision	recall	f1-score	support
Category 0	0.80	0.82	0.81	100
Category 1	0.60	0.70	0.65	100
Category 2	0.37	0.40	0.38	100
Category 3	0.39	0.31	0.34	100
Category 4	0.39	0.43	0.41	100
Category 5	0.60	0.55	0.58	100
Category 6	0.61	0.72	0.66	100
Category 7	0.63	0.60	0.62	100
Category 8	0.78	0.65	0.71	100
Category 9	0.75	0.74	0.74	100

Category 10	0.59	0.32	0.42	100
Category 11	0.39	0.45	0.42	100
Category 12	0.61	0.54	0.57	100
Category 13	0.70	0.44	0.54	100
Category 14	0.48	0.52	0.50	100
Category 15	0.51	0.55	0.53	100
Category 16	0.76	0.61	0.68	100
Category 17	0.79	0.76	0.78	100
Category 18	0.60	0.50	0.55	100
Category 19	0.60	0.43	0.50	100
Category 20	0.81	0.79	0.80	100
Category 21	0.50	0.81	0.62	100
Category 22	0.61	0.51	0.55	100
Category 23	0.70	0.68	0.69	100
Category 24	0.80	0.73	0.76	100
Category 25	0.65	0.45	0.53	100
Category 26	0.49	0.67	0.57	100
Category 27	0.30	0.33	0.31	100
Category 28	0.80	0.73	0.76	100
Category 29	0.66	0.50	0.57	100
Category 30	0.52	0.45	0.48	100
Category 31	0.47	0.58	0.52	100
Category 32	0.47	0.59	0.52	100
Category 33	0.66	0.49	0.56	100
Category 34	0.57	0.58	0.57	100
Category 35	0.30	0.39	0.34	100
Category 36	0.53	0.56	0.55	100
Category 37	0.56	0.58	0.57	100
Category 38	0.37	0.41	0.39	100
Category 39	0.76	0.74	0.75	100
Category 40	0.61	0.49	0.54	100
Category 41	0.79	0.81	0.80	100
Category 42	0.59	0.57	0.58	100
Category 43	0.50	0.57	0.53	100
Category 44	0.31	0.28	0.30	100
Category 45	0.38	0.38	0.38	100
Category 46	0.37	0.29	0.32	100
Category 47	0.61	0.63	0.62	100
Category 48	0.65	0.93	0.76	100
Category 49	0.64	0.72	0.68	100
Category 50	0.43	0.40	0.41	100
Category 51	0.50	0.60	0.55	100
Category 52	0.64	0.76	0.69	100
Category 53	0.63	0.87	0.73	100
Category 54	0.66	0.77	0.71	100
Category 55	0.27	0.26	0.26	100
Category 56	0.76	0.80	0.78	100
Category 57	0.75	0.62	0.68	100
Category 58	0.62	0.83	0.71	100
Category 59	0.60	0.53	0.56	100
Category 60	0.87	0.85	0.86	100
Category 61	0.60	0.66	0.63	100
Category 62	0.68	0.64	0.66	100
Category 63	0.44	0.51	0.47	100
Category 64	0.32	0.38	0.35	100
Category 65	0.47	0.37	0.42	100
Category 66	0.55	0.50	0.52	100
Category 67	0.41	0.41	0.41	100
Category 68	0.84	0.81	0.83	100
Category 69	0.80	0.70	0.74	100
Category 70	0.62	0.66	0.64	100
Category 71	0.65	0.75	0.69	100
Category 72	0.27	0.26	0.27	100
Category 73	0.49	0.38	0.43	100
Category 74	0.31	0.34	0.33	100
Category 75	0.71	0.84	0.77	100
Category 76	0.81	0.78	0.80	100
Category 77	0.49	0.38	0.43	100
Category 78	0.44	0.44	0.44	100
Category 79	0.63	0.56	0.59	100
Category 80	0.34	0.33	0.34	100

Category 81	0.56	0.78	0.65	100
Category 82	0.93	0.76	0.84	100
Category 83	0.59	0.47	0.52	100
Category 84	0.55	0.54	0.55	100
Category 85	0.67	0.70	0.69	100
Category 86	0.73	0.61	0.66	100
Category 87	0.57	0.73	0.64	100
Category 88	0.57	0.63	0.60	100
Category 89	0.65	0.63	0.64	100
Category 90	0.62	0.57	0.59	100
Category 91	0.68	0.69	0.69	100
Category 92	0.60	0.36	0.45	100
Category 93	0.35	0.39	0.37	100
Category 94	0.86	0.83	0.85	100
Category 95	0.59	0.64	0.62	100
Category 96	0.57	0.47	0.52	100
Category 97	0.46	0.62	0.53	100
Category 98	0.37	0.34	0.36	100
Category 99	0.75	0.53	0.62	100
accuracy			0.58	10000
macro avg	0.58	0.58	0.57	10000
weighted avg	0.58	0.58	0.57	10000

Візуальне представлення випадкових 16 зображень з вибірки, їхні класи та класи, які розпізнала модель:

Images with True and Predicted Labels

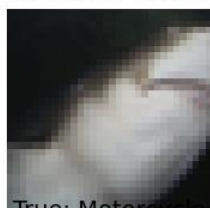
True: Motorcycle  
Predicted: Motorcycle



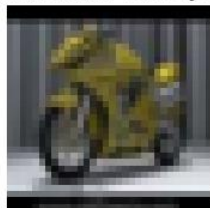
True: Sunflower  
Predicted: Sunflower



True: Shark  
Predicted: Rabbit



True: Motorcycle  
Predicted: Motorcycle



True: Streetcar  
Predicted: Streetcar



True: Lawn\_mower  
Predicted: Beetle



True: Cockroach  
Predicted: Cockroach



True: Ray  
Predicted: Ray



True: Bicycle  
Predicted: Bicycle



True: Clock  
Predicted: Clock



True: Couch  
Predicted: Television



True: Lion  
Predicted: Lion



True: Tiger  
Predicted: Tiger



True: Spider  
Predicted: Spider



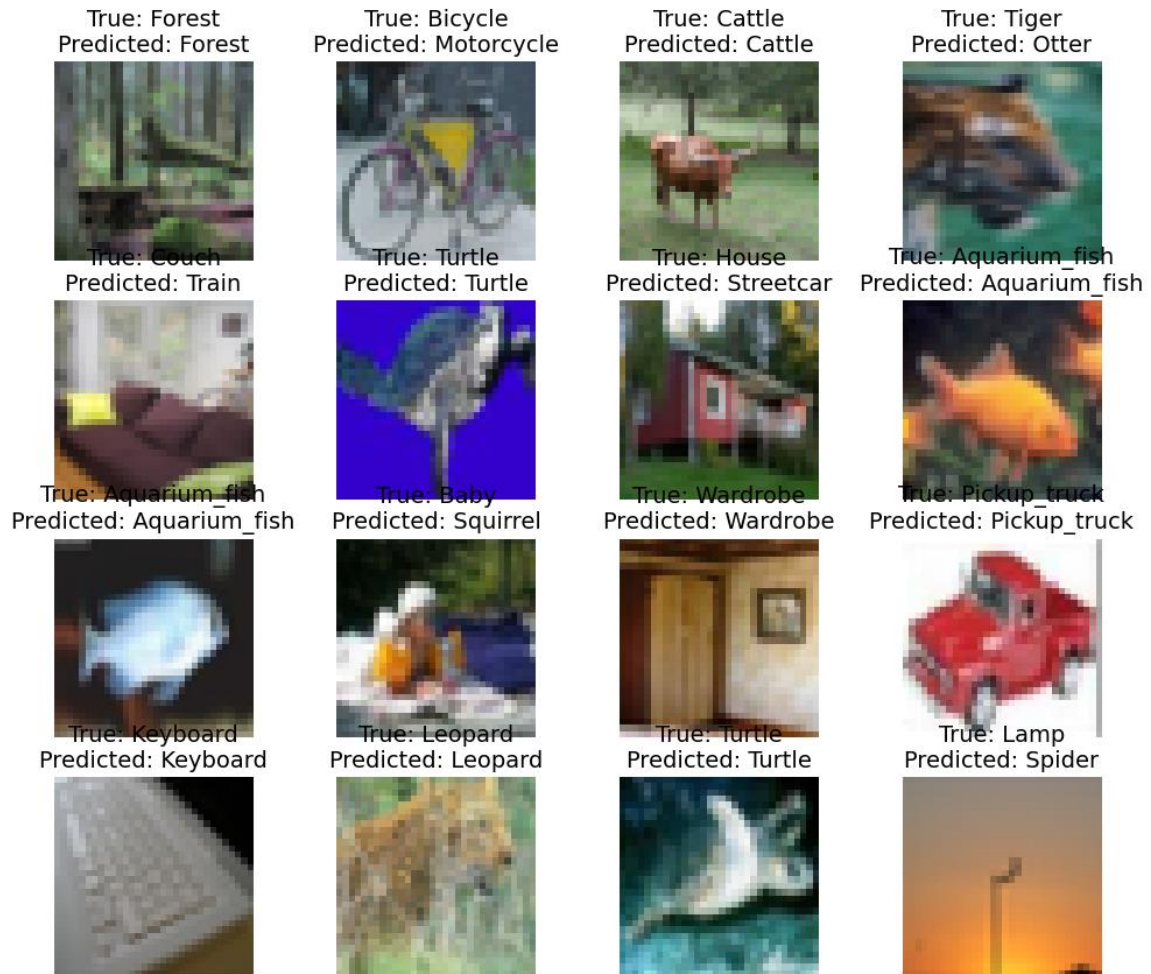
True: Man  
Predicted: Baby



True: Hamster  
Predicted: Fox



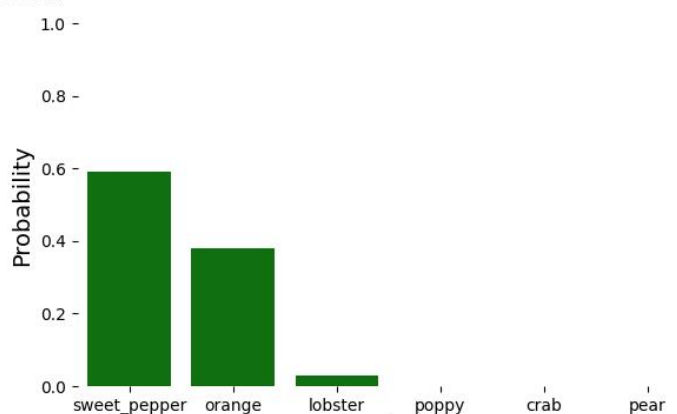
## Images with True and Predicted Labels



3. Для перевірки розпізнання довільних зображень (папка **img**) потрібно запустити **test\_model.py**.

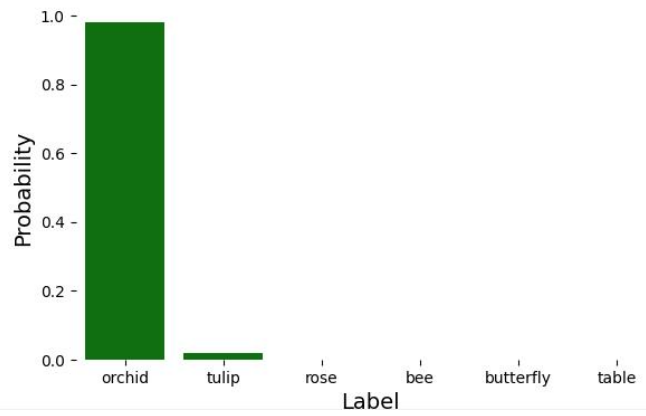


Prediction

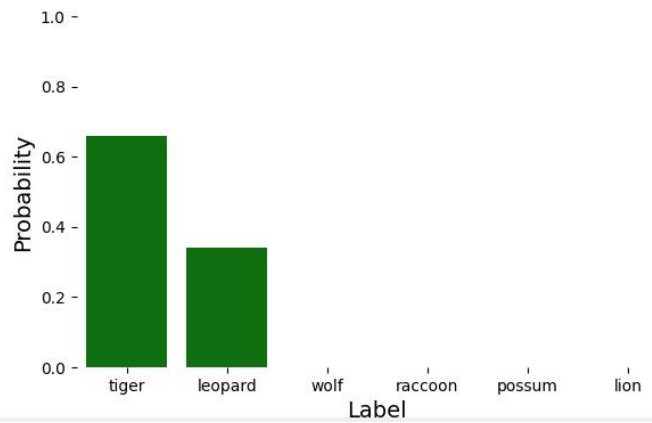




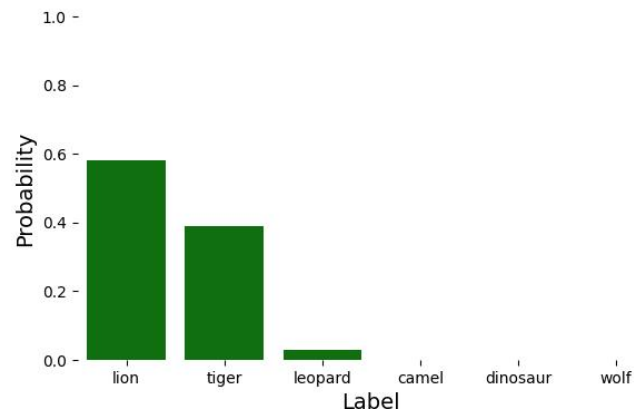
Prediction



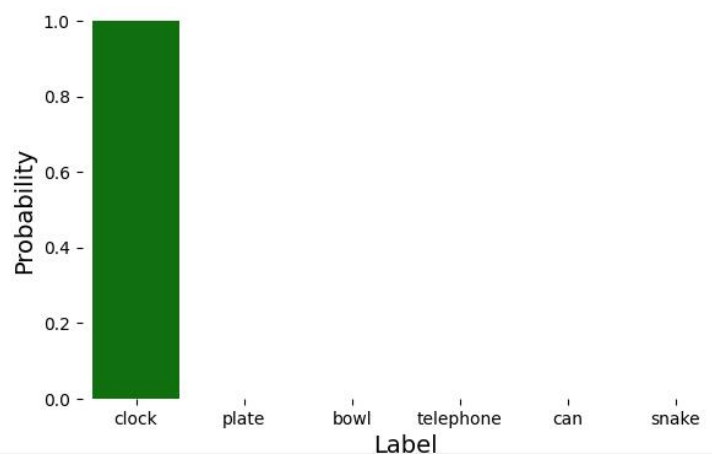
Prediction



Prediction

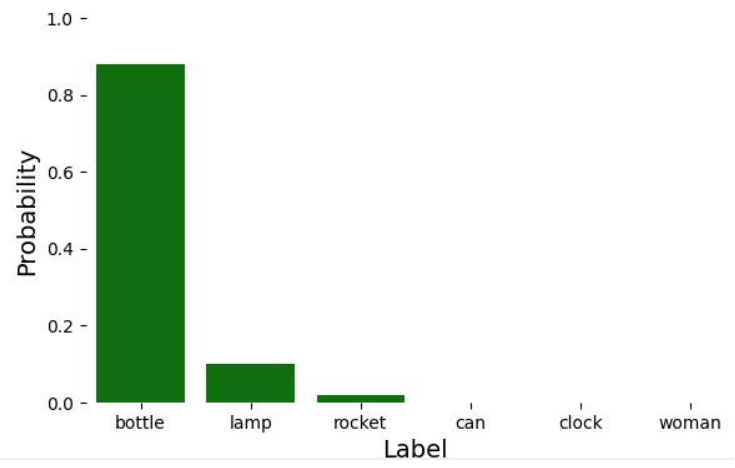


Prediction





Prediction





## Вимоги до звіту

1. Покроково виконати хід роботи.
2. Запустити навчання мережі (train\_model.py) та відобразити результати:
  - логи навчання мережі
  - результати навчання
  - візуальний графік навчання
3. Запустити оцінку мережі (test\_model.py) та відобразити результати:
  - загальний результат точності та втрат
  - результат точності по категоріях
  - скриншот 16 випадкових зображень з тестового датасету (**у кожного має бути унікальний**, так як 10 000 зображень в датасеті)
4. Запустити тестування мережі (train\_model.py) та відобразити результати:
  - підготувати 5-10 зображень різних категорій та покласти в папку img
  - скриншоти виконання програми з результатами розпізнання зображень (**у всіх мають бути унікальні зображення**)
5. Описати висновки.

### **Список літератури:**

1. Гітхаб коду даної лабораторної роботи –  
<https://github.com/misha-lavrov/CIFAR-100-CNN>
2. Оригінал статті та коду -  
[https://github.com/chetnakhanna16/CIFAR100\\_ImageRecognition?tab=readme-ov-file](https://github.com/chetnakhanna16/CIFAR100_ImageRecognition?tab=readme-ov-file)
3. Налаштування бібліотеки tensorflow –  
<https://www.tensorflow.org/install/pip#linux>
4. Налаштування бібліотеки tensorflow –  
<https://stackoverflow.com/questions/45662253/can-i-run-keras-model-on-gpu>
5. Набір даних CIFAR-100 –  
<https://www.cs.toronto.edu/~kriz/cifar.html>