Дисциплина «Системы искусственного интеллекта и большие данные»

Рабочая тетрадь № 7 (часть 2)

Теоретический материал – TensorFlow, Keras и среда выполнения GPU на Colaboratory

Нейронные сети являются мощным инструментом в области машинного обучения и искусственного интеллекта. Они имитируют работу человеческого мозга, используя искусственные нейроны для обработки и анализа данных. Нейронные сети способны решать широкий спектр задач, начиная от классификации изображений и заканчивая прогнозированием временных рядов. В этой части мы рассмотрим несколько примеров программ нейронных сетей, которые помогут вам лучше понять, как они работают и как их можно использовать в различных задачах. Мы начнем с простых примеров и постепенно перейдем к более сложным моделям.

TensorFlow — бесплатная платформа машинного обучения на Python с открытым исходным кодом, разработанная в основном в Google. Как и у NumPy, основная цель TensorFlow – дать инженерам и исследователям возможность манипулировать математическими выражениями с числовыми тензорами. Но TensorFlow может намного больше, чем NumPy, в том числе:

- автоматически вычислять градиент любого дифференцируемого выражения, что делает ее прекрасной основой для машинного обучения;
- работать не только на обычных, но также на графических и тензорных процессорах — высокопараллельных аппаратных ускорителях;
- распределять вычисления между множеством компьютеров;
- экспортировать вычисления другим окружениям выполнения, таким как C++, JavaScript (для веб-приложений, выполняющихся в браузере) или TensorFlow Lite (для приложений, действующих в мобильных или встраиваемых устройствах) и т. Д. Это упрощает развертывание приложений TensorFlow в практических условиях.

Keras — это высокоуровневая библиотека для создания и обучения нейронных сетей на языке Python. Она упрощает процесс разработки моделей, предоставляя удобный интерфейс для работы с TensorFlow и другими фреймворками. Keras позволяет быстро прототипировать модели и легко экспериментировать с различными архитектурами нейронных сетей.

Благодаря TensorFlow библиотека Keras может работать на различных типах оборудования – графическом, тензорном или обычном процессоре и поддерживает простую возможность распределения вычислений между тысячами компьютеров. (рис. 1).

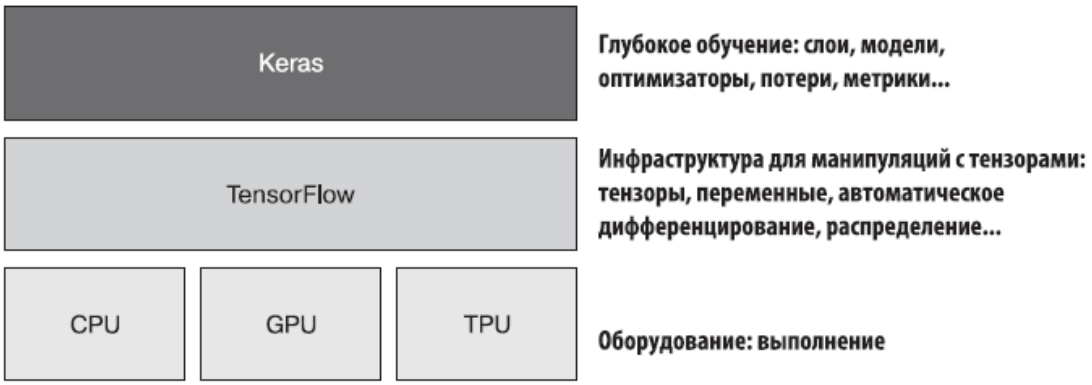


Рис. 1. TensorFlow — это низкоуровневая платформа тензорных вычислений, а Keras — высокоуровневая библиотека глубокого обучения

Colaboratory и работа со средой выполнения GPU. Colaboratory (или просто Colab) — это бесплатная облачная служба для блокнотов Jupyter, не требующая установки дополнительного программного обеспечения. По сути, это веб-страница, позволяющая сразу же писать и выполнять сценарии, использующие Keras. Она дает доступ к бесплатной (но ограниченной) среде выполнения на графическом процессоре и даже к среде выполнения на тензорном процессоре (TPU), благодаря чему вам не придется покупать свой GPU. Чтобы начать работу со средой выполнения GPU в Colab, выберите в меню пункт Runtime→Change Runtime Type (Среда выполнения→Сменить среду выполнения) и в раскрывающемся списке Hardware Accelerator (Аппаратный ускоритель) выберите T4 GPU (рис. 2).

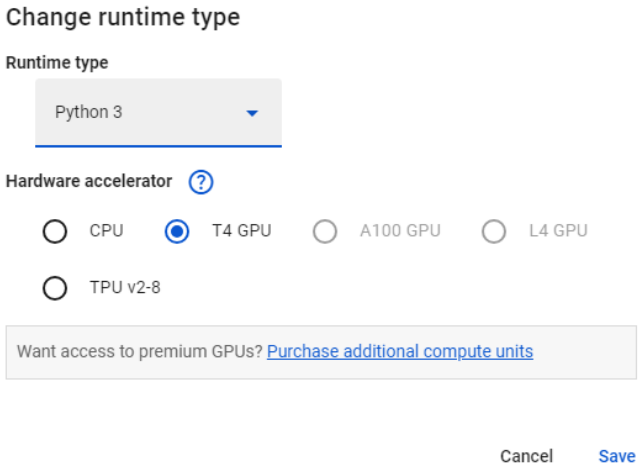


Рис. 2. Выбор среды выполнения GPU в Colab

Для выполнения примеров и заданий данной рабочей тетради рекомендуем использовать Jupyter Notebook, а также можно Colaboratory (https://colab.research.google.com/) со средой выполнения GPU для увеличения скорости обучения.

Пример 1. Простой пример нейронной сети на Python с использованием библиотеки Keras

Установка Keras и TensorFlow

Для начала установим необходимые библиотеки:

```
pip install keras tensorflow
```

Создание простой нейронной сети

Рассмотрим пример создания простой нейронной сети для задачи классификации. В этом примере мы создадим модель, которая будет классифицировать данные на два класса. Мы будем использовать 20 входных признаков и один выходной нейрон с сигмоидной активацией.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Генерация данных
X = np.random.rand(1000, 20)
y = np.random.randint(2, size=(1000, 1))

# Создание модели
model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Компиляция модели
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Обучение модели
model.fit(X, y, epochs=10, batch_size=32)

# Оценка модели
loss, accuracy = model.evaluate(X, y)
print(f'Loss: {loss}, Accuracy: {accuracy}')
```

Результат выполнения:

```
Epoch 1/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.4758 - loss: 0.7010
Epoch 2/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5225 - loss: 0.6969
Epoch 3/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5407 - loss: 0.6901
Epoch 4/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5237 - loss: 0.6917
Epoch 5/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5663 - loss: 0.6868
Epoch 6/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5594 - loss: 0.6863
Epoch 7/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5558 - loss: 0.6871
Epoch 8/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5906 - loss: 0.6827
Epoch 9/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5292 - loss: 0.6845
Epoch 10/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5833 - loss: 0.6776
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5849 - loss: 0.6768
Loss: 0.6794527173042297, Accuracy: 0.5799999833106995
```

Этот пример демонстрирует, как создать и обучить простую нейронную сеть для бинарной классификации. Мы используем 20 входных признаков и один выходной нейрон с сигмоидной активацией. Модель состоит из двух слоев: первый слой содержит 64 нейрона с активацией *ReLU*, а второй слой - один нейрон с активацией сигмоид. Мы используем функцию потерь *binary_crossentropy* и оптимизатор *adam*.

Задание 1.

1. Изучите и выполните пример 1 на Jupyter Notebook или Colab..
2. Поэкспериментируйте с новыми параметрами:
   - Перый слой содержит 128 нейронов с активацией *tanh*
   - Второй слой – один нейрон с активацией *softmax*
   - Изменить оптимизатор на *rmsprop*.

Решение

Пример 2. Пример нейронной сети для классификации изображений с использованием TensorFlow

Перед нами стоит задача: реализовать классификацию черно-белых изображений рукописных цифр (28 × 28 пикселей) по десяти категориям (от 0 до 9). Мы будем использовать набор данных MNIST, популярный в сообществе исследователей глубокого обучения, который существует практически столько же, сколько сама область машинного обучения, и широко используется для обучения. Этот набор содержит 60000 обучающих изображений и 10000 контрольных изображений, собранных Национальным институтом стандартов

и технологий США (National Institute of Standards and Technology — часть NIST в аббревиатуре MNIST) в 1980-х годах. «Решение» задачи MNIST можно рассматривать как своеобразный аналог Hello World в глубоком обучении — часто это первое действие, которое выполняется для уверенности, что алгоритмы действуют в точности как ожидалось. По мере углубления в практику машинного обучения вы увидите, что MNIST часто упоминается в научных статьях, блогах и т. д. Некоторые образцы изображений из набора MNIST можно видеть на рис. 3.

Рис. 3. Образцы изображений MNIST

Набор данных MNIST уже входит в состав Keras в форме набора из четырех массивов NumPy.

Загрузка набора данных MNIST в Keras

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Здесь train_images и train_labels — это *обучающий набор*, то есть данные, на которых модель обучается. После обучения модель будет проверяться тестовым (или контрольным) набором, test_images и test_labels. Здесь train_images и train_labels — это *обучающий набор*, то есть данные, на которых модель обучается. После обучения модель будет проверяться тестовым (или контрольным) набором, test_images и test_labels.

Изображения хранятся в массивах NumPy, а метки — в массиве цифр от 0 до 9. Изображения и метки находятся в прямом соответствии, один к одному.

Рассмотрим обучающие данные:

```
train_images.shape
```

```
(60000, 28, 28)
```

```
len(train_labels)
```

```
60000
```

```
train_labels
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

И контрольные данные:

```
test_images.shape
```

```
(10000, 28, 28)
```

```
len(test_labels)
```

```
10000
```

```
test_labels
```

```
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Вот как мы будем действовать дальше: сначала передадим нейронной сети обучающие данные, train_images и train_labels. Сеть обучится подбирать правильные метки для изображений. А затем мы предложим ей классифицировать изображения в test_images и проверим точность классификации по меткам из test_labels.

Теперь сконструируем сеть.

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Основным строительным блоком нейронных сетей является *слой*. Слой можно рассматривать как фильтр для данных: он принимает их и выводит в некоторой более полезной форме. В частности, слои извлекают *представления* из входных данных, которые, как мы надеемся, будут иметь больше смысла для решаемой задачи. Фактически методика глубокого обучения заключается в объединении простых слоев, реализующих некоторую форму поэтапной *очистки* данных.

Модель глубокого обучения можно сравнить с ситом, состоящим из последовательности фильтров — слоев — все более тонкой работы с данными.

В нашем случае сеть состоит из последовательности двух слоев Dense, которые являются тесно связанными (их еще называют *полносвязными*) нейронными слоями. Второй (и последний) слой — это десятипеременный слой классификации *softmax*, возвращающий массив с десятью оценками вероятностей (в сумме дающих 1). Каждая оценка определяет вероятность принадлежности текущего изображения к одному из десяти классов цифр.

Чтобы подготовить модель к обучению, нужно настроить еще три параметра для этапа *компиляции*:

- *оптимизатор* — механизм, с помощью которого сеть будет обновлять себя, опираясь на наблюдаемые данные и функцию потерь;
- *функцию потерь* — определяет, как сеть должна оценивать качество своей работы на обучающих данных и, соответственно, корректировать ее в правильном направлении;
- *метрики для мониторинга на этапах обучения и тестирования* — здесь нас будет интересовать только точность (доля правильно классифицированных изображений).

Этап компиляции:

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Перед обучением мы выполним предварительную обработку данных, преобразовав в форму, которую ожидает получить нейронная сеть, и масштабируем их так, чтобы все значения оказались в интервале [0, 1]. Исходные данные — обучающие изображения — хранятся в трехмерном массиве (60000, 28, 28) типа uint8, значениями в котором являются числа в интервале [0, 255]. Мы преобразуем его в массив (60000, 28 * 28) типа float32 со значениями в интервале [0, 1].

Подготовка исходных данных:

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

Теперь можно начинать обучение сети, для чего в случае библиотеки Keras достаточно вызвать метод fit модели — он попытается *адаптировать* (fit) модель под обучающие данные.

Обучение («адаптация») модели:

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 9s 16ms/step - accuracy: 0.8715 - loss: 0.4424
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 8s 11ms/step - accuracy: 0.9650 - loss: 0.1188
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.9782 - loss: 0.0740
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9850 - loss: 0.0509
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9894 - loss: 0.0377
```

В процессе обучения отображаются две величины: потери сети на обучающих данных и точность сети на обучающих данных. Мы быстро достигли точности 0,9894 (98,94 %).

Теперь у нас есть обученная модель, которую можно использовать для прогнозирования вероятностей принадлежности новых цифр к классам — изображений, которые не входили в обучающую выборку, как те из контрольного набора.

Использование модели для получения предсказаний:

```
test_digits = test_images[0:10]
predictions = model.predict(test_digits)
predictions[0]
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step
array([6.1924133e-08, 2.0428699e-09, 2.4057456e-06, 4.6557561e-05,
       6.3339944e-11, 6.8695813e-09, 9.9112320e-13, 9.9994940e-01,
       1.3561096e-07, 1.3052734e-06], dtype=float32)
```

Каждое число в элементе массива с индексом i соответствует вероятности принадлежности изображения цифры test_digits[0] к классу i.

Наивысшая оценка вероятности (0,9994940 — почти 1) для этого тестового изображения цифры находится в элементе с индексом 7, то есть согласно нашей модели — перед нами изображение цифры 7:

```
predictions[0].argmax()
```

```
7
```

```
predictions[0][7]
```

```
0.9999494
```

Прогноз можно проверить по массиву меток:

```
test_labels[0]
```

```
7
```

В целом, насколько хорошо справляется наша модель с классификацией прежде не встречавшихся ей цифр? Давайте проверим, вычислив среднюю точность по всему контрольному набору изображений.

Оценка качества модели на новых данных:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"test_acc: {test_acc}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9750 - loss: 0.0761
test_acc: 0.9787999987602234
```

Точность на контрольном наборе составила 97,88 % — немного меньше, чем на обучающем (98,94 %). Эта разница демонстрирует пример переобучения (overfitting), когда модели машинного обучения показывают точность на новом наборе данных худшую, чем на обучающем.

Задание 2.

Изучите и выполните пример 2 на Jupyter Notebook или Colab. Измените параметры:

- размер мини-выборки (batch_size) на 64
- число эпох (epochs) на 20
- Посмотрите новый результат точности на контрольном наборе и сравните с результатом примера 2.

Решение

Пример 3. Пример простой сверточной нейронной сети

Рассмотрим практический пример простой сверточной нейронной сети, классифицирующей изображения рукописных цифр из набора MNIST. Эту задачу мы решили в примере 2, использовав полносвязную сеть (ее точность на контрольных данных составила 97,88 %). Несмотря на простоту сверточной нейронной сети, ее точность будет значительно выше полносвязной модели из примера 2.

В следующем листинге показано, как выглядит простая сверточная нейронная сеть. Это стек слоев *Conv2D* и *MaxPooling2D*. Как она действует, рассказывается чуть ниже. Мы построим модель с помощью функционального API.

Создание небольшой сверточной нейронной сети:

```
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Важно отметить, что данная сеть принимает на входе тензоры с формой (высота_изображения, ширина_изображения, каналы), не включая измерение, определяющее пакеты. В данном случае мы настроили сеть на обработку входов с размерами (28, 28, 1), соответствующими формату изображений в наборе MNIST.

Рассмотрим поближе текущую архитектуру сети.

Сводная информация о сети:

```
model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73,856
flatten (Flatten)	(None, 1152)	0
dense (Dense)	(None, 10)	11,530

Total params: 104,202 (407.04 KB)
Trainable params: 104,202 (407.04 KB)
Non-trainable params: 0 (0.00 B)

Как видите, все слои *Conv2D* и *MaxPooling2D* выводят трехмерный тензор с формой (высота, ширина, каналы). Измерения ширины и высоты сжимаются с ростом глубины сети. Количество каналов управляется первым аргументом, передаваемым в слои *Conv2D* (32, 64 или 128).

Последний слой *Conv2D* выдает результат с формой (3, 3, 128) — карту признаков 3 × 3 со 128 каналами. Следующий шаг — передача этого результата на вход полносвязной классифицирующей сети, подобной той, с которой мы уже знакомы: стека слоев *Dense*. Эти классификаторы обрабатывают векторы — одномерные массивы, — тогда как текущий выход является трехмерным тензором. Чтобы преодолеть это несоответствие, мы преобразуем трехмерный вывод в одномерный с помощью слоя *Flatten*, а затем добавляем полносвязные слои *Dense*.

В заключение выполняется классификация по десяти категориям, поэтому последний слой имеет десять выходов и активацию softmax.

Теперь обучим сверточную сеть распознаванию цифр MNIST. Мы будем повторно брать большое количество программного кода из примера 2. Поскольку модель выполняет классификацию по десяти категориям с активацией *softmax*, мы используем функцию потерь категориальной перекрестной энтропии, а так как метки являются целыми числами, нам понадобится разреженная версия *sparse_categorical_crossentropy*.

Обучение сверточной нейронной сети на данных из набора MNIST:

```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype("float32") / 255
model.compile(optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Оценим модель на контрольных данных. Оценка сверточной сети:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc:.3f}")
```

```
Test accuracy: 0.991
```

Полносвязная сеть из примера 2 показала точность 97,88 % на контрольных данных, а простенькая сверточная нейронная сеть — 99,1 %: мы уменьшили процент ошибок на 68 % (относительно). Неплохо!

Задание 3.

1. Изучите и выполните пример 3 на Jupyter Notebook или Colab.
2. На основе блокнота (ноутбука) примера 3 постройте новую модель сверточных нейронных сетей с 5 слоями *Conv2D* и 4 слоями *MaxPooling2D* для классификации изображений на наборе данных MNIST.
3. Обучите модель и сравните новый результат точности с результатом в примере 3.

Решение

Пример 4. Пример сверточной нейронной сети для распознавания объектов на изображениях из набора данных CIFAR-10

Сверточные нейронные сети (Convolutional Neural Networks - CNN) особенно эффективны для обработки изображений. Они используют сверточные слои для автоматического извлечения признаков из изображений. Рассмотрим пример использования CNN для распознавания объектов на изображениях из набора данных CIFAR-10.

CIFAR-10 - это один из самых популярных наборов данных для обучения моделей компьютерного зрения. Он содержит 60,000 цветных изображений размером 32x32 пикселя, разделенных на 10 классов (самолёт, автомобиль, птица, кот, олень, собака, лягушка, лошадь, корабль и грузовик), по 6000 изображений на класс. Имеется 50000 обучающих изображений и 10000

тестовых изображений. Вот классы в наборе данных, а также 10 случайных изображений из каждого (рис. 4):

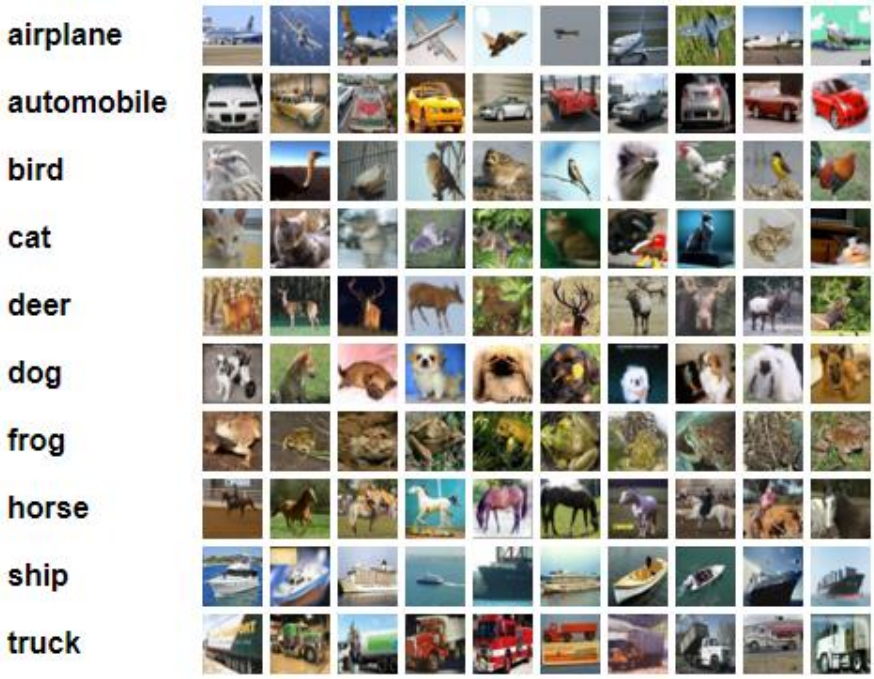
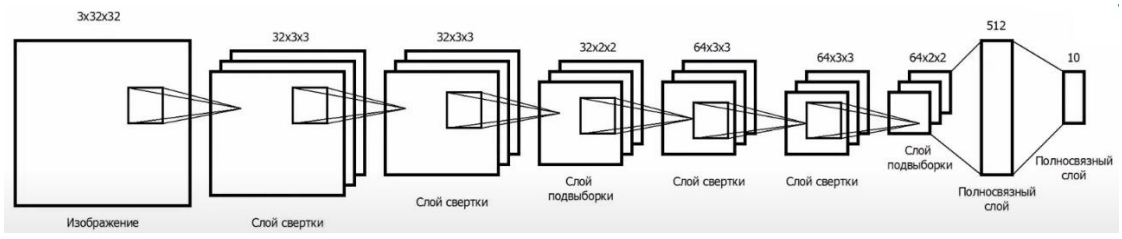


Рис. 4. Набор данных CIFAR-10

Набор данных разделен на пять обучающих пакетов и один тестовый пакет, каждый из которых содержит 10000 изображений. Тестовый пакет содержит 1000 случайно выбранных изображений из каждого класса. Обучающие пакеты содержат оставшиеся изображения в случайном порядке, но некоторые обучающие пакеты могут содержать больше изображений из одного класса, чем из другого. Между ними обучающие пакеты содержат 5000 изображений из каждого класса.

Для распознавания объектов на изображениях из набора данных CIFAR-10 мы будем использовать такую сверточную нейронную сеть:



Сеть включает два каскада из слоев свертки и подвыборки (всего 6 слоев), которые предназначены для выделения признаков изображений. Затем следует классификатор из двух полносвязных слоев (512 и 10 нейронов). Также как и при распознавании цифр выходной слой содержит вероятности принадлежности изображения к тому или иному классу.

На вход нейронной сети поступают изображения размером 32x32 в трех каналах (RGB - красный, зеленный и синий). На первом слое свёртки используются 32 карты признаков размера 3x3, т.е. каждый нейрон

сверточного слоя подключен к квадратному участку. Всего на этом слое используются 32 разных карт признаков. Следующий свёрточный слой имеет такую же архитектуру 32 карты признаков с ядром свёртки 3x3. После этого идёт слой подвыборки, на который выполняется уменьшение размерности для каждой карты признаков отдельно, поэтому здесь тоже используются 32 карты и размер поля подвыборки 2x2.

После слоя подвыборки начинается новый каскад сверточных слоев. На третьем и на четвертом слое свертки используются 64 карты признаков размером 3x3. А на втором слое подвыборки, которые следуют после этих сверточных слоев также происходит уменьшение размерности в квадрате 2x2. После этого данные преобразуются из двумерного формата в одномерный и передаются на полносвязный слой на котором уже и выполняется классификация.

Теперь рассмотрим как реализовать такую сеть с помощью библиотеки Keras и обучить её на наборе данных CIFAR-10. Как всегда сначала выполняем импорт необходимых элементов из библиотеки Keras и Numpy:

```
#Импорт необходимых библиотек
import numpy as np
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense, Flatten, Activation
from keras.layers import Dropout
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from keras.utils import to_categorical
from keras.optimizers import SGD
import matplotlib.pyplot as plt
```

Задаем seed для повторяемости результатов:

```
np.random.seed(42)
```

Загружаем данные из набора CIFAR-10:

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

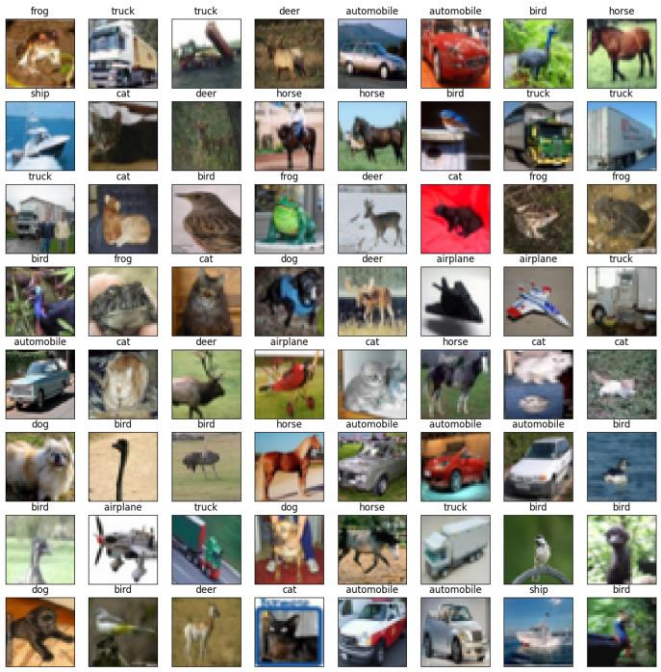
Список с названиями классов набора данных CIFAR-10:

```
# CIFAR-10 классы
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

Просматриваем примеры изображений:

```
# Создаем новую фигуру
plt.figure(figsize=(15,15))

# Прокрутить первые 25 изображений
for i in range(64):
    # Создаем подсюжет (subplot) для каждого изображения
    plt.subplot(8, 8, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

    # Показать изображение
    plt.imshow(X_train[i])

    # Установить метку в качестве заголовка
    plt.title(class_names[y_train[i][0]], fontsize=12)

# Показать изображения
plt.show()
```

Мы получим примеры изображений:



Установим параметры:

```
# Размер мини-выборки
batch_size = 32
# Количество классов изображений
nb_classes = 10
# Количество эпох для обучения
nb_epoch = 25
# Размер изображений
img_rows, img_cols = 32, 32
# Количество каналов в изображении: RGB
img_channels = 3
```

Нам необходимо выполнить предварительную обработку данных. Нормализуем данные о интенсивности пикселей изображений, чтобы все они находились в диапазоне [0,1]. Для этого преобразуем их в тип *float32* и делим на 255:

```
# Нормализуем данные
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

А метки классов необходимо преобразовать в категории:

```
# Преобразуем метки в категории
Y_train = to_categorical(y_train, nb_classes)
Y_test = to_categorical(y_test, nb_classes)
```

Наша сеть имеет 10 нейронов и выходной сигнал нейронов соответствует вероятности того, что изображение принадлежит к данному классу. Соответственно, номера классов в метках мы должны преобразовать в представление по категориям.

Теперь, когда наши данные подготовлены мы можем приступить к созданию сети. Создаем модель *Sequential* – последовательная сеть, где слои идут друг за другом, и перый каскад из слоев свертки и подвыборки.

```
# Создаем последовательную модель
model = Sequential()
```

Добавляем в модель свёрточный слой, который работает с двухмерными данными тип *Conv2D*:

```
# Первый сверточный слой
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=(32, 32, 3), activation='relu'))
```

Этот слой будет иметь 32 карты признаков, размер ядра свертки на каждой карте 3x3. Размер входных данных 32x32x3, что соответствует трём каналам изображений для кодов трёх цветов RGB размера изображения 32x32. В качестве функции активации используем *ReLU*.

Второй сверточный слой устроен точно так же 32 карты признаков размером 3x3:

```
# Второй сверточный слой
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
```

Затем идет слой подвыборки, размер уменьшения размерности 2x2 и в качестве слоя подвыборки мы используем *MaxPooling2D*, т.е. из квадратика размером 2x2 выбирается максимальное значение.

```
# Первый слой подвыборки
model.add(MaxPooling2D(pool_size=(2, 2)))
```

После каскада из двух сверточных слоев и слоя подвыборки мы добавляем слой регуляризации *Dropout*:

```
# Слой регуляризации Dropout
model.add(Dropout(0.25))
```

Dropout – одна из техник предотвращения переобучения (overfitting). В сверточных нейронных сетях, переобучение часто возникает когда находящиеся друг с другом нейроны настраиваются на совместную работу. За счёт этого они настраиваются на особенности конкретной выборки, а не на общие закономерности, характерные для различных изображений. Техника *Dropout* позволяет достаточно просто и эффективно снизить переобучение, которое возникает в сверточные нейронные сети. Для этого в процессе обучения когда на вход нейронной сети подается каждый новый объект случайным образом выключаются некоторое количество нейронов с заданной вероятностью. *Dropout* 0.25 означает, что нейрон будет отключаться с вероятностью 25%. Оставшиеся нейроны обучаются распознавать необходимые признаки без участия соседних нейронов.

После слоя регуляризации идёт ещё один каскад из двух сверточных слоев и слоя подвыборки. Но в этом каскаде больше карт признаков – 64.

```
# Третий сверточный слой
model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
# Четвертый сверточный слой
model.add(Conv2D(64, (3, 3), activation='relu'))
# Второй слой подвыборки
model.add(MaxPooling2D(pool_size=(2, 2)))
# Слой регуляризации Dropout
model.add(Dropout(0.25))
```

Слой подвыборки устроен точно также: выбор максимального значения из квадратика размером 2x2. После слоя подвыборки снова идет слой регуляризации *Dropout*, который выключает нейроны с вероятностью 25%.

После двух каскадов сверточных слоев и слоев подвыборки следует классификатор, который по признакам найденной сверточной сетью выполняет определение к какому конкретно классу принадлежит объект на картинке. Сначала нам необходимо преобразовать нашу сеть из двумерного представления в плоское. Для этого добавляем слой *Flatten* и затем добавляем два полносвязных слоев типа *Dense*. В слое содержится 512 нейронов, используется функция активации *ReLU*. В выходном слое содержится 10 нейронов по количеству классов. На этом слое используется функция активации *softmax*, которая соответствует вероятности появления того или иного класса.

```
# Слой преобразования данных из 2D представления в плоское
model.add(Flatten())
# Полносвязный слой для классификации
model.add(Dense(512, activation='relu'))
# Слой регуляризации Dropout
model.add(Dropout(0.5))
# Выходной полносвязный слой
model.add(Dense(nb_classes, activation='softmax'))
```

Суммарное выходное значение всех 10 нейронов равно единице. Между двумя полносвязными слоями у нас есть слой регуляризации *Dropout*, который выключает нейроны в этот раз с вероятностью 50%. Сеть, которую мы создали в этом примере уже является глубокой. В ней используются 8 слоев (4 сверточных слоев, 2 слоя подвыборки и 2 полносвязных слоев.

После того как мы задали сеть, можно её скомпилировать. Вызываем метод *model.compile*, в качестве функции ошибки используем *categorical_crossentropy*, которое хорошо подходит когда на выходе у нас значение вероятности появления классов. Оптимизируем с помощью стохастического метода градиентного спуска и в качестве метрики используем точность *accuracy*:

```
# Задаем параметры оптимизации
sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
```

Вызываем метод *fit* для обучения сети. Обучаем сеть на данных для обучения *X_train* содержит изображение для обучения *Y_train* правильные ответы уже в преобразованные в предоставление по категориям. Параметр *validation_split* говорит о том, что мы разбиваем набора *X_train* и *Y_train* на две части: обучающая выборка 90% и проверочная выборка 10%. Разбивку и проверку качества обучения на обеих выборках Keras выполняет автоматически. Размер мини-выборки *(batch_size = 32)*, т.е. мы изменяем веса нейронных сетей после того как обрабатываем каждые 32 объекта и обучение сети выполняется в течение 25 эпох *(nb_epoch)*. Параметр *shuffle* установленный в True, это значение по умолчанию говорит о том, что библиотека Keras в начале каждой эпохи будет перемешивать данные, чтобы они шли в разном порядке. Это повышает качество обучения, т.к. мы используем стохастический метод градиентного спуска.

```
# Обучаем модель
history = model.fit(X_train, Y_train,
                    batch_size=batch_size,
                    epochs=nb_epoch,
                    validation_split=0.1,
                    shuffle=True,
                    verbose=2)
```

```
Epoch 1/25
1407/1407 - 19s - 14ms/step - accuracy: 0.3689 - loss: 1.7155 - val_accuracy: 0.4760 - val_loss: 1.4084
Epoch 2/25
1407/1407 - 6s - 4ms/step - accuracy: 0.5338 - loss: 1.2939 - val_accuracy: 0.6000 - val_loss: 1.1083
Epoch 3/25
1407/1407 - 10s - 7ms/step - accuracy: 0.6035 - loss: 1.1141 - val_accuracy: 0.6716 - val_loss: 0.9630
```

```
Epoch 23/25
1407/1407 - 5s - 4ms/step - accuracy: 0.8005 - loss: 0.5758 - val_accuracy: 0.7782 - val_loss: 0.6921
Epoch 24/25
1407/1407 - 10s - 7ms/step - accuracy: 0.8007 - loss: 0.5778 - val_accuracy: 0.7694 - val_loss: 0.6985
Epoch 25/25
1407/1407 - 10s - 7ms/step - accuracy: 0.8034 - loss: 0.5663 - val_accuracy: 0.7538 - val_loss: 0.7433
```

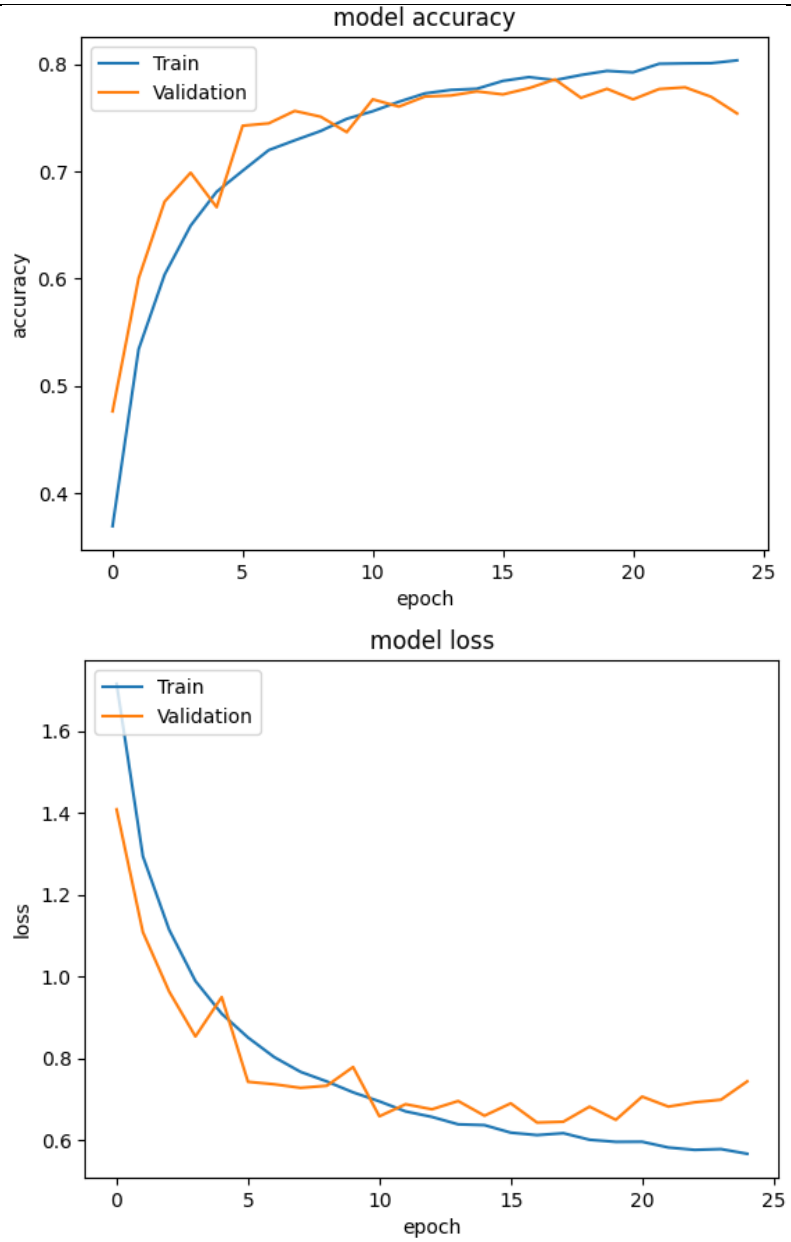
Оцениваем качество обучения модели на тестовых данных:

```
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Точность работы на тестовых данных: %.2f%%" % (scores[1]*100))
```

```
Точность работы на тестовых данных: 73.99%
```

Теперь мы сгенерируем графики с помощью библиотеки *matplotlib* для визуализации потерь при обучении и проверке, а также изменения точности по эпохам, используя модель history:

```
# Подвести итог history для точности (accuracy)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
# Подвести итог history для потери (loss)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Задание 3.

1. Изучите и выполните пример 4 на Jupyter Notebook или Colab.
2. Попробуйте изменить нейронную сеть, чтобы улучшить качество решения:
   - Изменяйте количество нейронов в слоях
   - Добавляйте новые скрытые слои
   - Изменяйте количество эпох обучения
   - Изменяйте размер мини-выборки (batch_size)
3. После подбора лучших гиперпараметров, обучите сеть еще раз на полном объеме данных без разделения на обучающий и проверочный наборы. Во время обучения следите, чтобы не возникло переобучение.

Решение