МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Искусственные нейронные сети»

Тема: Распознавание рукописных символов

Студент гр. 8382 _____ Нечепуренко Н.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2021

Цели работы.

Реализовать классификацию черно-белых изображений рукописных цифр (28x28) по 10 категориям (от 0 до 9).

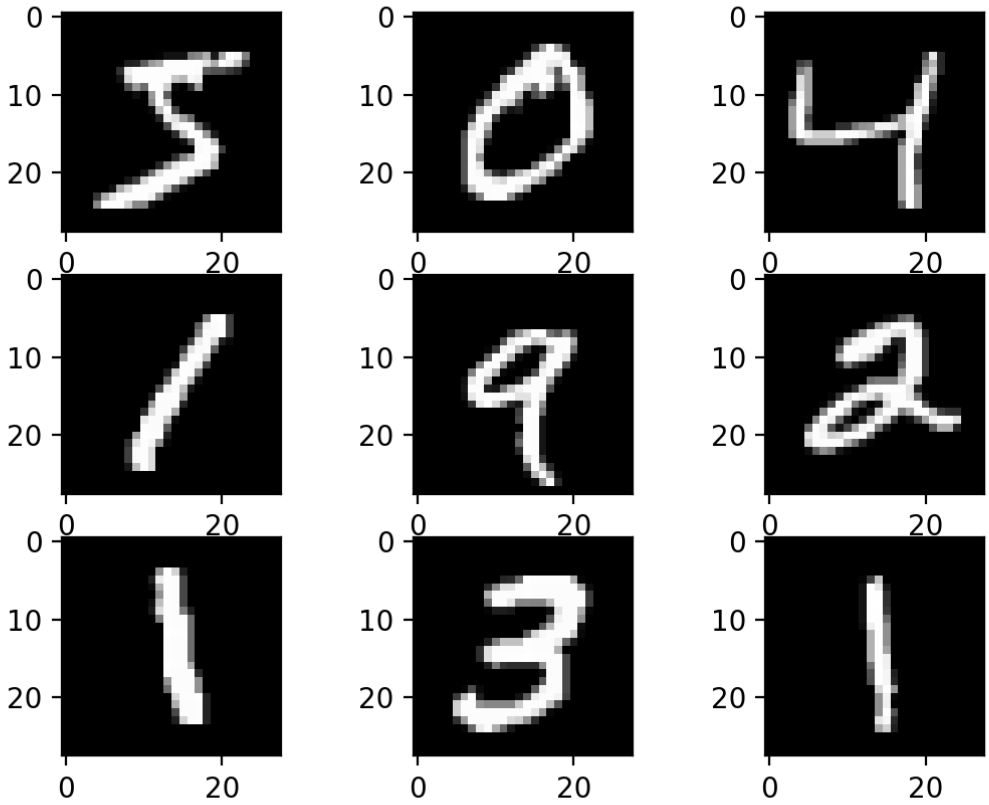

Рисунок 1 – Примеры изображений из датасета MNIST

Набор данных содержит 60 000 изображений для обучения и 10 000 изображений для тестирования.

Задачи.

- Ознакомиться с представлением графических данных
- Ознакомиться с простейшим способом передачи графических данных нейронной сети

- Создать модель
- Настроить параметры обучения
- Написать функцию, позволяющая загружать изображение пользователи и классифицировать его

Требования.

1. Найти архитектуру сети, при которой точность классификации будет не менее 95
2. Исследовать влияние различных оптимизаторов, а также их параметров, на процесс обучения
3. Написать функцию, которая позволит загружать пользовательское изображение не из датасета

Выполнение работы.

Загрузим встроенный в keras набор размеченных изображений рукописных символов.

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

Каждое наблюдение содержит в себе черно-белое изображение 28 на 28 пикселей с числом от 0 до 9 и соответствующую метку. Убедимся в корректности загруженных данных, выведем первое изображение (см. рис. 2).

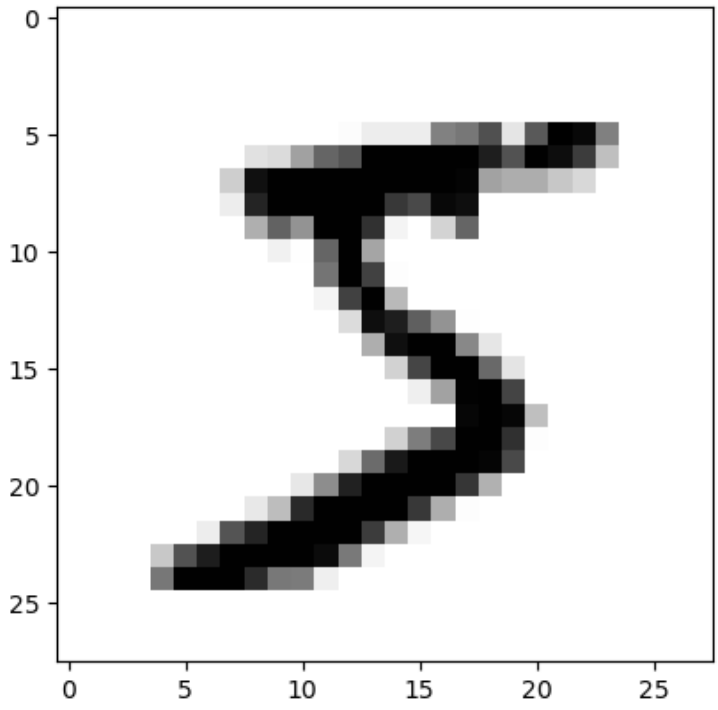

Рисунок 2 – Первое изображение датасета

Каждый пиксель изображения представлен числом от 0 до 255. Нормализуем значения пикселей с помощью следующего кода:

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

Для столбца меток проведем one-hot кодирование с помощью функции to_categorical.

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Найдем архитектуру сети, при которой точность классификации будет не менее 95%. Протестируем архитектуру, приведенную в листинге 5 методических указаний. Модель выглядит следующим образом

```
model = Sequential()
```

```
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer=RMSprop(), loss='
   categorical_crossentropy', metrics=['accuracy'])
```

В первом слое вытягиваем изображение в вектор (728, 1). Затем идет скрытый слой с функцией активации relu на 256 нейронов, в последнем слое имеем 10 нейронов и функцию активации softmax. Выходной слой позволяет определить наиболее вероятный класс числа на изображении.

Обучим модель со следующими параметрами

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

Получаем следующий результат

```
Epoch 1/5
469/469 [==============================] - 2s 2ms/step - loss:
   0.4670 - accuracy: 0.8699
Epoch 2/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.1380 - accuracy: 0.9596
Epoch 3/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0913 - accuracy: 0.9734
Epoch 4/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0656 - accuracy: 0.9804
Epoch 5/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0493 - accuracy: 0.9853
313/313 [==============================] - 0s 576us/step - loss:
   0.0783 - accuracy: 0.9770
```

```
test_acc: 0.9769999980926514
```

Сразу же получили необходимую точность на тестовом наборе.

Проведем исследование зависимости результатов работы модели от выбранного оптимизатора. Выше приведен результат модели с оптимизатором RMSprop. Он представлен в keras следующим образом

```
tf.keras.optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    name="RMSprop",
    **kwargs
)
```

Алгоритм на каждой итерации высчитывает скользящее среднее квадрата градиентов, вычисляемое по формуле

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2$$

где t – номер шага, g – вектор градиента, $\beta \in (0;1)$ – параметр значения (обычно равен 0.9). Веса изменяются следующим образом

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}$$

η – скорость обучения, ϵ – маленькое число, стабилизирующее знаменатель.

Изменение параметра ρ на 0.3 и на 0.5 не повлияло на результаты модели. Попробуем уменьшить η в десять раз.

```
    Epoch 1/5
469/469 [==============================] - 3s 3ms/step - loss:
   1.2480 - accuracy: 0.6831
```

```
Epoch 2/5
469/469 [==============================] - 2s 4ms/step - loss:
   0.3553 - accuracy: 0.9055
Epoch 3/5
469/469 [==============================] - 2s 4ms/step - loss:
   0.2824 - accuracy: 0.9215
Epoch 4/5
469/469 [==============================] - 2s 4ms/step - loss:
   0.2395 - accuracy: 0.9341
Epoch 5/5
469/469 [==============================] - 2s 4ms/step - loss:
   0.2120 - accuracy: 0.9412
313/313 [==============================] - 0s 691us/step - loss:
   0.1983 - accuracy: 0.9436
    test_acc: 0.9435999989509583
```

Результаты модели оказались хуже. В таком виде ей требуется больше времени для обучения, чтобы достигнуть лучшего результата.

Рассмотрим другие оптимизаторы. Например, SGD – стохастический градиентный спуск.

```
tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.0, nesterov=False, name="
       SGD", **kwargs
)
```

При нулевом значении импульса веса изменяются как

$$w_t = w_{t-1} - \eta g_t$$

Если значение импульса больше 0, формула имеет вид

$$w_t = w_{t-1} + pv_{t-1} - \eta g_t$$

где p – импульс, v – аналог скорости. При использовании импульса Нестерова

$$v_t = pv_{t-1} - \eta g_t$$

$$w_t = w_{t-1} + pv_t - \eta g_t$$

Наличие импульса позволяет ускорять спуск в необходимом направлении и тушит возникающие колебания.

Запустим модель с параметрами SGD по умолчанию.

```
    Epoch 1/5
469/469 [==============================] - 1s 1ms/step - loss:
   1.6138 - accuracy: 0.5826
Epoch 2/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.5868 - accuracy: 0.8629
Epoch 3/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.4542 - accuracy: 0.8820
Epoch 4/5
469/469 [==============================] - 1s 1ms/step - loss:
   0.3974 - accuracy: 0.8938
Epoch 5/5
469/469 [==============================] - 1s 1ms/step - loss:
   0.3627 - accuracy: 0.9001
313/313 [==============================] - 0s 545us/step - loss:
   0.3254 - accuracy: 0.9130
    test_acc: 0.9129999876022339
```

Модель сходится медленно.

Добавим импульс равный 0.5 и правило Нестерова.

```
    Epoch 1/5
469/469 [==============================] - 1s 2ms/step - loss:
   1.3049 - accuracy: 0.6581
```

```
Epoch 2/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.4367 - accuracy: 0.8861
Epoch 3/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.3576 - accuracy: 0.9021
Epoch 4/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.3193 - accuracy: 0.9119
Epoch 5/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.2959 - accuracy: 0.9173
313/313 [==============================] - 0s 577us/step - loss:
   0.2705 - accuracy: 0.9234
    test_acc: 0.9233999848365784
```

Модель с SGD все равно заметно уступает модели с RMSprop.

Рассмотрим оптимизатор Adam. Он выглядит как смесь SGD и RMSprop. В нем содержатся два параметра β_1 и β_2

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
    **kwargs
)
```

Формула обновления весов имеет вид

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\tilde{v}_{t-1} + \epsilon}} \tilde{m}_{t-1}$$

$$\tilde{m}_t = \frac{\beta_1 m_{t-1} + (1-\beta_1) g_t}{1-\beta_1}$$
$$\tilde{v}_t = \frac{\beta_2 v_{t-1} + (1-\beta_2) g_t^2}{1-\beta_2}$$

Запустим модель со значениями по умолчанию

```
    Epoch 1/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.5346 - accuracy: 0.8528
Epoch 2/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.1422 - accuracy: 0.9596
Epoch 3/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0970 - accuracy: 0.9717
Epoch 4/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0688 - accuracy: 0.9803
Epoch 5/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0499 - accuracy: 0.9855
313/313 [==============================] - 0s 585us/step - loss:
   0.0751 - accuracy: 0.9770
    test_acc: 0.9769999980926514
```

Точность как у RMSprop. Протестируем модель с параметрами $\beta_1 = 0.8$ и $\beta_2 = 0.8$.

```
    Epoch 1/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.5129 - accuracy: 0.8604
Epoch 2/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.1307 - accuracy: 0.9620
Epoch 3/5
```

```
469/469 [==============================] - 1s 2ms/step - loss:
   0.0853 - accuracy: 0.9753
Epoch 4/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0618 - accuracy: 0.9818
Epoch 5/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.0484 - accuracy: 0.9854
313/313 [==============================] - 0s 545us/step - loss:
   0.0780 - accuracy: 0.9773
    test_acc: 0.9772999882698059
```

Результаты немного улучшились, но это изменение статистически не значимо.

Оба параметра равны нулю

```
    Epoch 1/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.5478 - accuracy: 0.8495
Epoch 2/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.1924 - accuracy: 0.9488
Epoch 3/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.1641 - accuracy: 0.9607
Epoch 4/5
469/469 [==============================] - 1s 2ms/step - loss:
   0.1604 - accuracy: 0.9668
Epoch 5/5
469/469 [==============================] - 1s 1ms/step - loss:
   0.1597 - accuracy: 0.9694
313/313 [==============================] - 0s 569us/step - loss:
   0.2148 - accuracy: 0.9641
    test_acc: 0.9641000032424927
```

Сохраним модель с оптимизатором Adam со стандартными параметрами в файл mnist_model. Напишем скрипт, читающий данную модель с диска и проводящий классификацию числа на изображении, путь к которому передается через аргумент командной строки.

Исходный код скрипта приведен в приложении Б. Функция предсказания числа на изображении

```
def predict_image(model, image_path):
    img = load_img(image_path, color_mode="grayscale",
       target_size=(28, 28))
    pixels_array = np.array([img_to_array(img)]) / 255
    prediction_vector = model.predict(pixels_array)
    prediction = np.argmax(prediction_vector, 1)[0]
    print("Number on the image is:")
    print(prediction)
    return prediction
```

Нарисуем в графическом редакторе (MS Paint) цифры от 0 до 9 и прогоним их через скрипт. Например, для картинки 4.png вывод

```
Number on the image is:
4
```

Модель ошиблась с 6, выдав в качестве ответа 5, для 7 выдала 3, для 9 выдала 3. Остальные цифры были угаданы верно. Возможно, модель немного переобучилась. Так как числа были нарисованы в MS Paint, значениями пикселей были только 0 и 255 (либо очень близкие к 255), возможно, это тоже сыграло роль на точности распознавания мелких деталей.

Выводы.

В результате выполнения лабораторной работы был реализован классификатор рукописных цифр из датасета MNIST. Была найдена архитектура

сети, при которой достигалась точность на тестовых данных порядка 97Были проанализированы три оптимизатора: RMSprop, SGD и Adam. Был написан скрипт, позволяющий классифицировать введенное пользователем изображение.

ПРИЛОЖЕНИЕ А. КОД ОБУЧЕНИЯ МОДЕЛИ.

```
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import matplotlib.pyplot as plt
from keras.utils import to_categorical
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Sequential
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import *

(train_images, train_labels), (test_images, test_labels) = mnist.
   load_data()


def plt_show_bitmap_image(image):
    plt.imshow(image, cmap=plt.cm.binary)
    plt.show()


train_images = train_images / 255.0
test_images = test_images / 255.0
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)


def build_model(optimizer):
    model = Sequential()
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dense(10, activation='softmax'))
    model.compile(optimizer=optimizer, loss='
```

```
        categorical_crossentropy', metrics=['accuracy'])
    return model


model = build_model(Adam())
model.fit(train_images, train_labels, epochs=5, batch_size=128)
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('test_acc:', test_acc)
model.save('models/mnist_model')
```

ПРИЛОЖЕНИЕ Б. КОД ИСПОЛНЯЕМОГО СКРИПТА.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import sys

import numpy as np
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import load_img,
   img_to_array



def predict_image(model, image_path):
    img = load_img(image_path, color_mode="grayscale",
       target_size=(28, 28))
    pixels_array = np.array([img_to_array(img)]) / 255
    prediction_vector = model.predict(pixels_array)
    prediction = np.argmax(prediction_vector, 1)[0]
    print("Number on the image is:")
    print(prediction)
    return prediction


if __name__ == '__main__':
    if len(sys.argv) != 2:
        exit("You must specify path to image.")
    model = load_model('models/mnist_model')
    try:
        predict_image(model, sys.argv[1])
    except Exception as e:
        print(e, file=sys.stderr)
```