Отчёт по задаче 5 (Распознавание изображений)

Владимирова Элина, Евдокимов Данил, Назаров Максим, Стельмах Татьяна

Декабрь 2020

Содержание

1 Введение

В этой работе мы построим и обучим свёрточную нейронную сеть, которая будет отличать известного героя серии игр, манги и аниме - Пикачу (Рисунок 1) от других существ из мира покемонов.

2 Постановка задачи

Необходимо построить и обучить свёрточную нейронную сеть, которая сможет распознавать картинки (сообщать нам, изображен ли на рисунке Пикачу или нет), оценить полученные результаты, построить графики зависимости точности распознавания от эпохи и зависимости потерь от эпох, сделать краткие выводы.

Рисунок 1. Тот самый покемон

3 Описание метода

Более подробно алгоритм нашей работы будет описан ниже.
Для работы мы выбрали свёрточную нейронную сеть, поскольку мы можем обучить модель распознавать заданного персонажа с точностью 81%, предоставив сравнительно небольшой набор данных. Работа свёрточной нейронной сети основана на переходе от конкретных деталей изображения ко все более абстрактным свойствам - вплоть до выделения понятий высокого уровня. При этом сеть самонастраивается, самостоятельно вырабатывая необходимую иерархию абстрактных признаков, а также осуществляя фильтрацию путем удаления незначительных и выделения существенных качеств.
Мы будем использовать сверточную сеть для выделения признаков изображений и собираемся следовать популярной, эффективной и простой архитектуре *VGGnet* (см. рисунки 2 и 3).

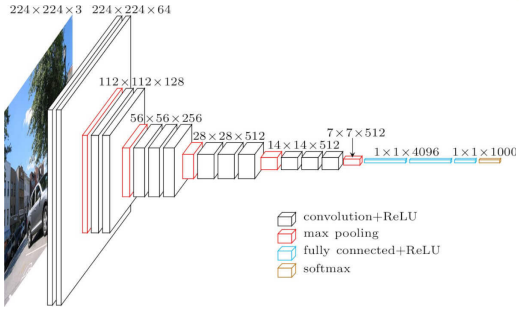

Рисунок 2. Архитектура VGG-16

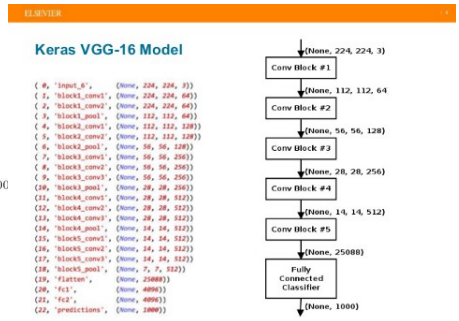

Рисунок 3. Более подробно

4 Описание данных

До начала работы с кодом мы нашли нужные *datasets* с Пикачу и другими покемонами. В результате были сформированы папки Train и Test для обучения и тестирования модели соответственно. Изображений разыскиваемого

героя у нас около 900, картинок с другими покемонами примерно столько же: мы используем одинаковое количество изображений каждого типа в связи с выбранной метрикой *accuracy* (её удобно отображать на графиках), поскольку в условиях, когда изображений неравное количество, она выдаёт подозрительно высокую точность, не соответствующую действительности.
В папке *Train* были собраны картинки с названиями, начинающимися либо на `"pikachu.."`, либо на `"not_pika.."` для облегчения процесса создания нами массива меток для массива изображений в ходе работы с программой (для переименования изображений воспользовались модулем *os*).
Т.е., входными данными являются заранее заготовленные наборы для обучения и тестов (папки с изображениями различного размера в формате *.jpg*). Примеры можно увидеть ниже.

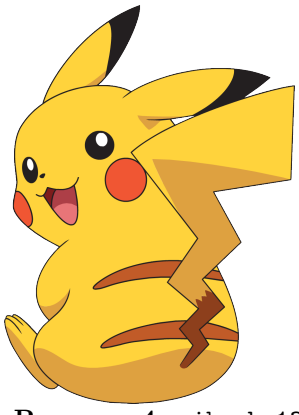

Рисунок 4. `pikachu139.jpg`

Рисунок 5. `not_pika645.jpg`

5 Описание постановки эксперимента

Как работает код?

После непосредственной обработки найденных данных мы переходим к написанию кода. Будем использовать следующие библиотеки для реализации нашего эксперимента:

```
import cv2          #чтение и изменение размера наших изображений
import gc           #ручная очистка и удаление ненужных переменных
import numpy as np
import pandas as pd
```

```
import os
import random
from sklearn.model_selection import train_test_split
from keras import layers
from keras import models
from keras import optimizers
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing.image import img_to_array, load_img
import matplotlib.pyplot as plt
```

Теперь получаем массивы для тестов и обучения нашей модели. В следующем блоке кода мы изменим размер изображений с помощью модуля *cv*2 до 150x150 вне зависимости от начальных параметров.
Цветное изображение состоит из 3 каналов, т.е. каждый пиксель может быть описан кодами красного, зеленого и синего цветов. Опишем функцию, которая поможет нам изменить и считать полученные изображения:

```
def read_and_process_image(list_of_images):
    global nrows
    global ncolumns
    mas_of_rgb = []
    signs = []
    i = 0
    for image in list_of_images:
        mas_of_rgb.append(cv2.resize(cv2.imread(image, cv2.IMREAD_COLOR), (nrows, ncolumns), interpolation=cv2.INTER_CUBIC))
        if len(mas_of_rgb) == i + 1:
            if 'pikachu' in image:
                signs.append(1)
            elif 'not_pika' in image:
                signs.append(0)
        i += 1

    return mas_of_rgb, signs

mas_of_rgb, signs = read_and_process_image(train_imgs)
```

Теперь переменная `mas_of_rgb` указывает на массив кодов пикселей изображения, а `signs` - на список меток.
Разделим наши данные на наборы для обучения и проверки, используя `train_test_split модуля sklearn.model_selection`:

```
x_train, x_val, y_train, y_val = train_test_split(mas_of_rgb, signs, test_size=0.20, random_state=2)
```

20% данных будут назначены набору проверки, а остальные 80% - набору, отвёденному для обучения модели.
При создании нашей модели мы будем использовать *Keras* - открытую нейросетевую библиотеку, содержащую многочисленные реализации широко применяемых строительных блоков нейронных сетей, таких как слои. Напомним, какие модули мы импортируем из *Keras*:

```
from keras import layers      # модуль слоёв из Keras (включает различные типы слоёв)
from keras import models      # содержит Sequential модель, которую мы и будем использовать
from keras import optimizers  # модуль, который содержит различные типы алгоритмов обратного распространения
from keras.preprocessing.image import ImageDataGenerator
```

Мы собираемся использовать небольшой *vggnet* (как было сказано выше), но ниже можно видеть, что размер выходных данных (`filter size`) увеличивается по мере нашего движения вниз по слоям:
$32 \to 64 \to 128 \to 512$ - последний слой равен 1.

```
model = models.Sequential()  #создаем нашу sequential модель

""" Тут создаём первый слой и указываем нужный его тип.
Так как этот слой первый, т.е. входной, он имеет некоторые важные параметры:
1. filter size [32] - размер выходных данных
2. kernel size [3,3] - высота и ширина окна двумерной свертки
3. activation ['relu'] - функция активации (в данном случае функция линейного выпрямителя)
4. input shape [150,150,3] - формат входных данных (изображения 150x150, разбитые на mas_of_rgb) """
model.add(layers.Conv2D(32, (3, 3), activation='relu',input_shape=(150, 150, 3)))

model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2))) # уменьшение размера входящих объектов
# -> уменьшение количества подбираемых параметров
# т.е уменьшаем время обучения
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

""" Overfitting происходит, когда наша модель хорошо работает на данных для обучения,
но на тестовых показывает результаты, далекие от истины.
Добавляем "сглаживающий слой", который удаляет все измерения,
кроме одного (преобразует матрицу в единый массив) """

model.add(layers.Flatten())

""" Dropout случайным образом отбрасывает несколько слоев в нейронных сетях (в данном случае половину),
а затем учится с сокращенной сетью: учится быть независимой, то есть при отсутствии некоторых слоёв она будет работать корректно
помогает решать проблему переобучения """


model.add(layers.Dropout(0.5))


model.add(layers.Dense(512, activation='relu'))

""" Последний слой имеет выходной размер 1 и другую функцию активации, называемую сигмоидом.
Её область значений - [0;1], что и позволяет сделать вывод о том, Пикачу или же нет изображён на
картинке (возвращает вероятность нахождения на рисунке нужного покемона) """

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

Можем ли мы увидеть расположение и размер параметров нашей свертки? Для этого вызовем функцию `Keras.summary()` и наглядно представим количество параметров, используемых в обучении, а также общее расположение различных слоев (Рисунок 6).

Составим же нашу модель:

```
model.compile(loss='binary_crossentropy', optimizer=optimizers.RMSprop(lr=1e-4), metrics=['acc'])
```

Команде `model.compile()` мы передаем три параметра :

- Loss [‘binary_crossentropy’] - указываем функцию потерь, которую `optimizer` будет минимизировать: поскольку мы работаем с двумя классами (’пикачу’ и ’не пикачу’), будем использовать *binarycrossentropy*
- optimizer [rmsprop] - выбираем алгоритм оптимизации: в данном случае будем использовать *rmsprop* - наиболее часто обновляющиеся веса обновляем меньше, однако вместо полной суммы обновлений используем усреднённый по истории квадрат градиента.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dropout (Dropout)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3211776
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Рисунок 6. Результат вызова фунцкии

- metrics [acc] - выбираем метрику для оценки качества моделей и сравнения различных алгоритмов: будем использовать *accuracy*, показывающую долю правильных ответов алгоритма.

Перед началом обучения мы должны обработать наши изображения, чтобы предложить их модели. Для этого воспользуемся `ImageDataGenerator` - класс, который определяет конфигурацию для подготовки и дополнения данных изображения, позволит нам быстро настраивать генераторы python, автоматически превращающие файлы изображений в предварительно обработанные тензоры. Например, с его помощью мы можем "переложить" значения пикселей 0-255 на [0,1]

```
train_datagen = ImageDataGenerator(rescale=1./255,      # коэффицент маштабирования (конвертирует 0-255 в 0-1)
                                   rotation_range=40,  # случайным образом применяем какое-либо преобразование к Image
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,)

val_datagen = ImageDataGenerator(rescale=1./255)

""" Создаем объект ImageDataGenerator для нашего набора проверки. Мы выполняем только
масштабирование, в случае выше мы хотим улучшить обучение путём преобразования изображения """
```

После этого мы создаем на основе объектов ImageDataGenerator генераторы python, передав им данные для обучения и проверки.

```
train_generator = train_datagen.flow(x_train, y_train, batch_size=batch_size)
# вызываем flow() для созданных генераторов данных, передавая в метод набор данных и меток.
```

```
val_generator = val_datagen.flow(x_val, y_val, batch_size=batch_size) # batch_size - кол-во изображений за раз
# аналогично для наборов проверки

history = model.fit_generator(train_generator,
                              steps_per_epoch=ntrain // batch_size,
                              epochs=64,
                              validation_data=val_generator,
                              validation_steps=nval // batch_size)
```

Затем с помощью метода `.fit()` мы тренируем нашу модель, указывая следующие параметры:

- объект обучающего набора `ImageDataGenerator [train_generator]`
- `steps_per_epoch` - кол-во изображений, которое мы хотим обработать перед завершением эпохи (столько раз мы обновим градиент)
- `epochs` - кол-во просмотров массива обучающих данных
- генератор данных проверки
- `validation_steps` - кол-во использованных проверочных изображений перед завершением эпохи

```
Epoch 2/64
44/44 [==============================] - 36s 822ms/step - loss: 0.3860 - acc: 0.8374 - val_loss: 0.4558 - val_acc: 0.8097
Epoch 3/64
44/44 [==============================] - 38s 869ms/step - loss: 0.3311 - acc: 0.8558 - val_loss: 0.3293 - val_acc: 0.8892
Epoch 4/64
44/44 [==============================] - 46s 1s/step - loss: 0.3360 - acc: 0.8679 - val_loss: 0.3362 - val_acc: 0.8750
Epoch 5/64
44/44 [==============================] - 50s 1s/step - loss: 0.3063 - acc: 0.8707 - val_loss: 0.3649 - val_acc: 0.8494
Epoch 6/64
44/44 [==============================] - 50s 1s/step - loss: 0.3092 - acc: 0.8714 - val_loss: 0.3002 - val_acc: 0.8750
Epoch 7/64
44/44 [==============================] - 50s 1s/step - loss: 0.2954 - acc: 0.8849 - val_loss: 0.3119 - val_acc: 0.8977
Epoch 8/64
44/44 [==============================] - 49s 1s/step - loss: 0.2710 - acc: 0.9027 - val_loss: 0.2777 - val_acc: 0.8949
Epoch 9/64
44/44 [==============================] - 50s 1s/step - loss: 0.2703 - acc: 0.8991 - val_loss: 0.2958 - val_acc: 0.8977
Epoch 10/64
23/44 [==============>.............] - ETA: 22s - loss: 0.2722 - acc: 0.8927
```

Рисунок 7. Процесс обучения

После обучения мы сохраняем нашу модель:

```
model.save_weights('weights_pika.ckpt')
model.save('model_pika.ckpt')
```

Теперь для построения графиков мы должны извлечь необходимые нам данные:

```
acc = history.history['acc']             # точность при обучении (доля правильно распознанных изображений)
val_acc = history.history['val_acc']     # точность на данных для проверки
loss = history.history['loss']           # потери при обучении
val_loss = history.history['val_loss'] # потери на данных для проверки

epochs = range(1, len(acc) + 1)          # массив наших эпох [1 ,..., 64]
```

Строим графики:

```
plt.plot(epochs, acc, 'b', label='Training accurarcy')
plt.plot(epochs, val_acc, 'r', label='Validation accurarcy')
plt.title('Training and Validation accurarcy')
plt.legend() # график зависимости точности от эпохи.

plt.figure()
#Train and validation loss
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and Validation loss')
plt.legend() # график зависимости потерь от эпохи.

plt.show()
```

Сами графики будут показаны в Результатах ниже.
Теперь мы отобразим несколько изображений из папки *Tests* и найдем нашего героя на картинке:

```
mas = []
for i in range(10):
    ind = random.randint(0,len(test_imgs)-1)
    mas.append(test_imgs[ind])
X_test, y_test = read_and_process_image(mas)      # обработаем изображения, причем y_test будет пуст
mas_of_rgb = np.array(X_test)
test_datagen = ImageDataGenerator(rescale=1./255) # перейдем от кодов цветов к [0;1], чтобы отправить их модели
i = 0
text_labels = []
plt.figure(figsize=(30,20))
for batch in test_datagen.flow(mas_of_rgb, batch_size=1):
    pred = model.predict(batch)
    if pred > 0.5:
        text_labels.append('pikachu')
    else:
        text_labels.append('not pikachu')
    plt.subplot(5 // columns + 1, columns, i + 1)
    plt.title('This is a ' + text_labels[i])
    imgplot = plt.imshow(batch[0])
    i += 1
    if i % 10 == 0:
        break
plt.show()
```

6 Результаты

Итак, спустя 64 эпохи мы получили точность около 80%. Ниже привёдем долгожданные графики, описанные выше:
Выводы по этим двум графикам:

- Следует отметить, что мы не переобучаемся, поскольку точность модели на обучающих данных довольно близка к точности на проверочных данных (точность на данных для обучения существенно не превосходит точность на данных проверки - более того, они чередуются по величине)
- Заметим, что точность растёт по мере увеличения эпохи. Вероятно, если мы увеличим кол-во эпох ещё, то сможем добиться большей точности.
- Обратим внимание на график `loss(epoch)`. Тут также видно, что мы не переобучаемся, так как потери при обучении и проверке постепенно

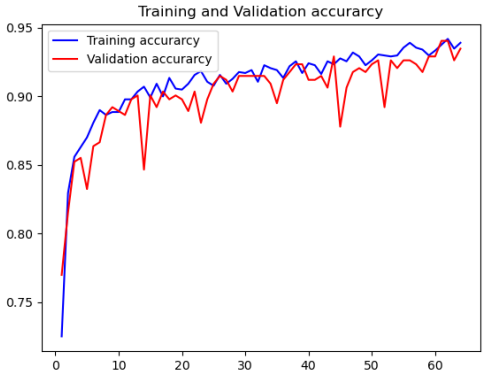

Рисунок 8. `acc(epoch)`

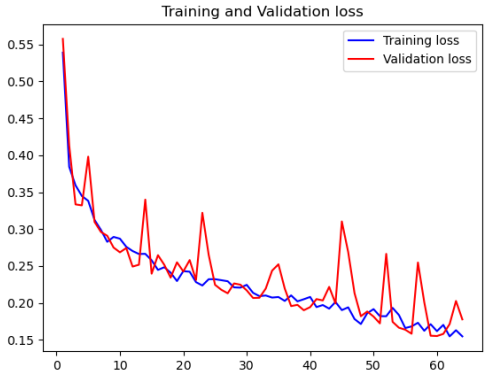

Рисунок 9. `loss(epoch)`

снижаются. Можно предположить, что увеличение кол-ва эпох приведет к снижению потерь.

Посмотрим теперь, как наша модель справляется с поставленной задачей на новый для нее изображениях (из папки *Tests*):

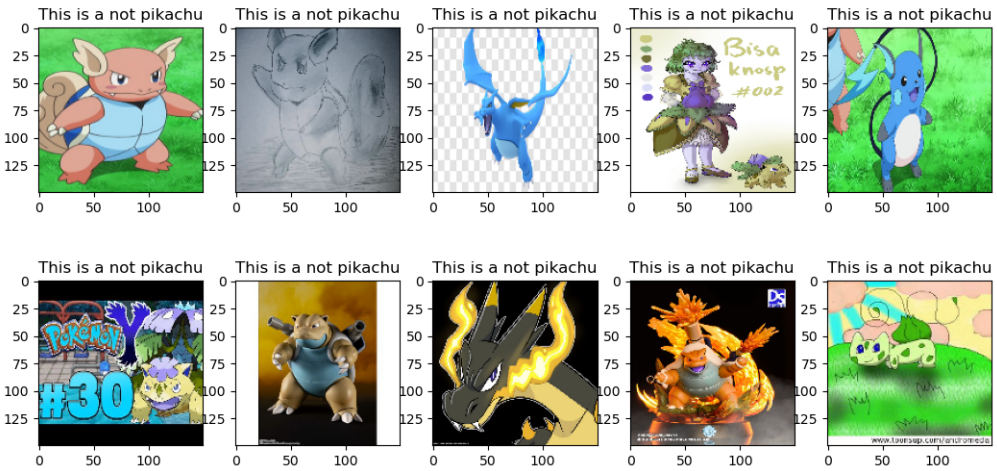

Рисунок 10. Тест 1

Как мы видим, первый тест (Рисунок 10) был успешным, однако Пикачу на изображениях не встретился, поскольку картинки из папки *Test* выбирались в случайном порядке.
Теперь посмотрим на второй тест (Рисунок 11): здесь наша модель ошиблась и дала ложноположительный результат, что указывает на ее несовершенство.

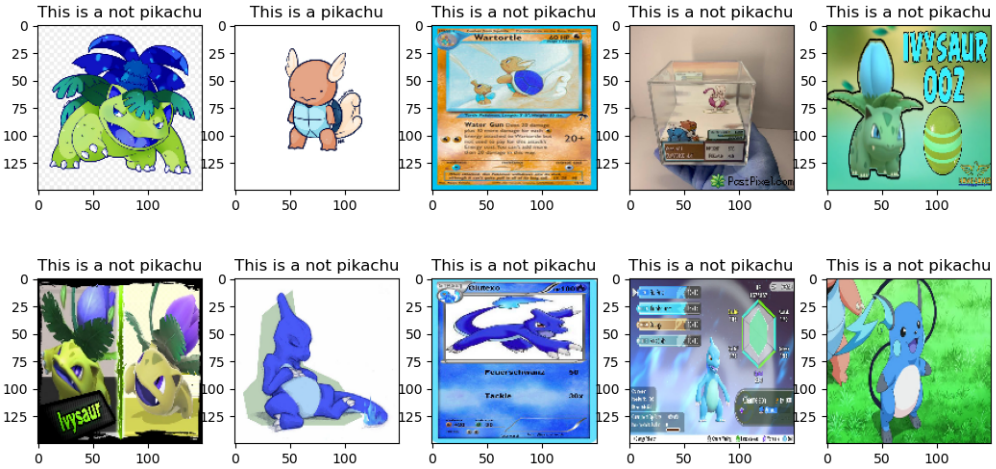

Рисунок 11. Тест 2

Теперь сгенерируем наш массив неслучайным образом и поместим туда только изображения разыскиваемого покемона (Рисунок 12). Как мы видим, на каждой картинке он смог узнать Пикачу.

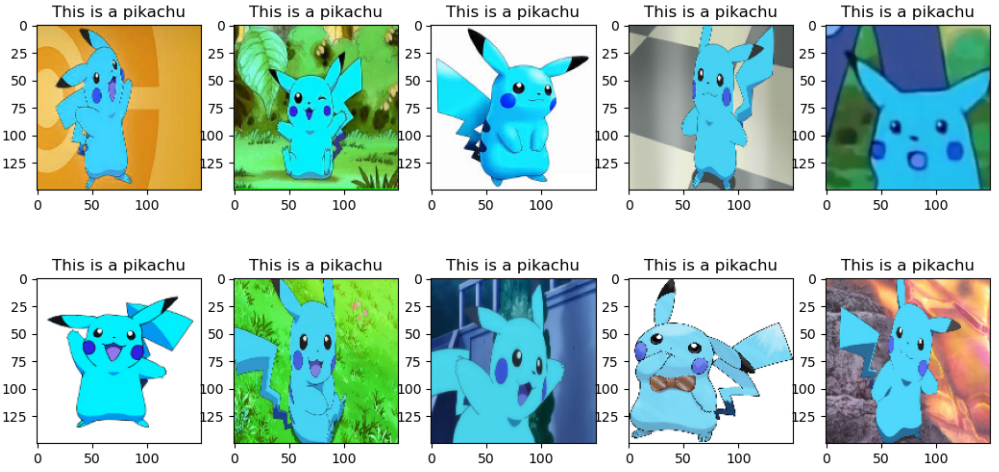

Рисунок 12. Тест 3

Восстановив модель, мы получим точность 74,2% на картинках, которые модель увидела впервые - такой результат вполне объясним.

7 Краткие выводы

Мы построили и обучили свёрточную нейронную сеть, дающую точность около 80% на данных для проверки при обучении из 64 эпох. На графиках мы увидели, что с увеличением кол-ва эпох точность растёт, а потери сокращаются (при этом важно учитывать тот факт, что мы не переобучали модель). Можем предположить, что при увеличении числа эпох показатели точности нашей модели улучшатся. К сожалению, на просторах интернета довольно мало датасетов с Пикачу, поэтому и данных для обучения и проверки у нас не так много, но, возможно, увеличение кол-ва входных изображений также помогло бы нам достичь более высокой точности. Однако даже при имеющихся условиях мы достаточно успешно достигли нужного нам результата.

Рисунок 13

Спасибо за внимание.

Идея	Евдокимов Данил, Владимирова Элина, Стельмах Татьяна Назаров Максим
Тактическая концепция	Евдокимов Данил, Назаров Максим, Владимирова Элина
Код и его проверка	Стельмах Татьяна, Евдокимов Данил
Отчет	Стельмах Татьяна
Редакция отчета	Владимирова Элина
Подбор данных	Евдокимов Данил, Владимирова Элина, Стельмах Татьяна Назаров Максим