

# Отчёт по задаче 5 (распознавание картинок)

Владимирова Элина, Евдокимов Данил, Назаров Максим, Стельмах Татьяна

Декабрь 2020

## Содержание

1 Введение	1
2 Описание метода	2
3 Описание данных	2
4 Описание постановки эксперимента	3
5 Результаты	8
6 Краткие выводы	10

## 1 Введение

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

**Наша задача:** построить и обучить свёрточную нейронную сеть, которая сможет распознавать картинки (сообщать нам: Пикачу или же нет изображён на рисунке), оценить полученные результаты, построить графики зависимости точности распознавания от эпохи и зависимости потерь от эпох, сделать краткие выводы.



Рисунок 1. Тот самый покемон

## 2 Описание метода

Более подробно алгоритм нашей работы будет описан ниже.

В своём проекте мы выбрали свёрточную нейронную сеть, поскольку имея небольшой набор данных, мы можем обучить модель распознавать нашего зверька с точностью 81 %. Работа свёрточной нейронной сети основана на переходе от конкретных особенностей изображения к более абстрактным деталям, и далее к ещё более абстрактным деталям вплоть до выделения понятий высокого уровня. При этом сеть самонастраивается и вырабатывает сама необходимую иерархию абстрактных признаков, фильтруя маловажные детали и выделяя существенное.

Мы будем использовать свёрточную сеть для выделения признаков изображений и будем следовать популярной, эффективной и простой архитектуре под названием *VGGnet* (смотреть рисунки 2 и 3).

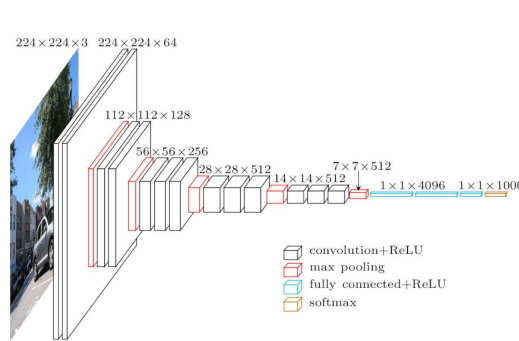


Рисунок 2. Архитектура VGG-16

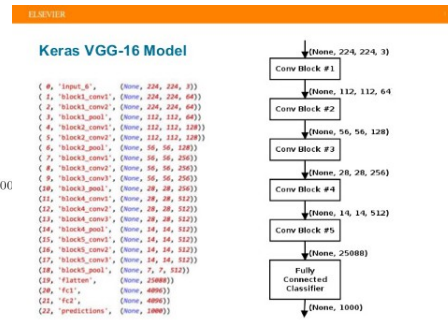


Рисунок 3. Более подробно

## 3 Описание данных

До начала работы с кодом мы нашли нужные *datasets* с Пикачу и другими покемонами. В результате у нас имелись папки Test и Train для тестов и обучения модели соответственно, причём изображений разыскиваемого героя у нас около 900, а картинок с другими покемонами примерно столько же. Мы используем одинаковое количество изображений каждого класса, в связи с выбранной метрикой *accuracy* (её удобно отображать на графиках), поскольку в условиях, когда изображений неравное количество, она выдаёт подозрительно высокую точность (ложную).

В папке *Train* были собраны картинки с названиями, начинающиеся либо на "pikachu.", либо на "not\_pika." для того, чтобы уже в ходе работы с программой мы могли легко создать массив меток для массива изображений (для переименования изображений воспользовались модулем *os*) То есть

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

## 4 Описание постановки эксперимента

Как работает код? После непосредственной обработки найденных данных мы переходим к написанию кода. Мы 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
```

Затем получаем массивы для тестов и обучения нашей модели. В следующем блоке кода мы собираемся изменить размер изображений с помощью модуля *cv2*.

Размер наших изображений после обработки - 150x150. Цветное изображение

состоит из 3 каналов, то есть каждый пиксель мы можем описать кодами красного, зеленого и синего цветов. Опишем функцию, которая помогает изменить и считать полученные изображения:

```
def read_and_process_image(list_of_images):
    global n_rows
    global n_columns
    mas_of_rgb = []
    signs = []
    i = 0
    for image in list_of_images:
        mas_of_rgb.append(cv2.resize(cv2.imread(image, cv2.IMREAD_COLOR), (n_rows, n_columns), 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` является массивом значений пикселей изображения, а `signs` - списком меток.

Давайте 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, отведённого для обучения модели. При создании нашей модели мы будем использовать *Keras* - открытую нейросетевую библиотеку, которая содержит многочисленные реализации широко применяемых строительных блоков нейронных сетей, таких как слои. Повторимся и поясним, что 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 того, как мы спускаемся вниз по слоям.

32 → 64 → 128 → 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, такие как:
1. filter size [32] - размер выходных данных
2. kernel size [3,3] - определяет высоту и ширину 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,
а на тестовых результаты будут далеки 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 и неработоспособной на одном слое -
помогает при переобучении """

model.add(layers.Dropout(0.5))

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

""" Последний же слой имеет выходной размер 1 и другую функцию
активации, называемую сигмидом. Её область значений - [0;1], что как раз и позволяет сделать вывод о том,
Пикачу или же нет изображён на картинке (возвращает вероятность нахождение 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, которые будут использованы в обучении, и общее расположение различных слоев (Рисунок 6).

Составим же нашу модель:

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

Мы передаем три параметра команде `model.compile()`:

- Loss ['binary\_crossentropy'] - указываем функцию потерь, которую `optimizer` будет минимизировать, а поскольку мы работаем с двумя классами ('пикачу' и 'не пикачу'), то будем использовать *binary\_crossentropy*
- optimizer [rmsprop] - выбираем алгоритм оптимизации, в данном случае мы выбрали *rmsprop* - обновляем меньше те веса, которые слишком часто обновляются, но вместо полной суммы обновлений, будем использовать усреднённый по истории квадрат градиента.
- metrics [acc] - выбираем метрику для оценки качества моделей и сравнения различных алгоритмов; мы выбрали *accuracy* - доля правильных ответов алгоритма.

Также прежде чем начать обучение мы должны обработать наши изображения, чтобы предложить их модели. Для этого воспользуемся `ImageDataGenerator` - класс, который определяет конфигурацию для подготовки и дополнения

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 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_2 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_3 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_3 (MaxPooling2D)	(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 функции

данных изображения, позволяет нам быстро настраивать генераторы python, которые автоматически превращают файлы изображений в предварительно обработанные тензоры. Например, с его помощью мы можем "переложить" значения пикселей [от 0 до 255] на [0,1]

```
train_datagen = ImageDataGenerator(rescale=1./255, # коэффициент масштабирования (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 """
```

Затем мы создали на основе объектов ImageDataGenerator генераторы python, передав 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 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 данные для проверки
loss = history.history['loss']         # потери при обучении
val_loss = history.history['val_loss'] # потери на данные для проверки

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

Строим графики:

```
plt.plot(epochs, acc, 'b', label='Training accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation accuracy')
plt.title('Training and Validation accuracy')
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 сможем увидеть в результатах ниже.

Теперь мы отобразим несколько изображений из папки *Tests* и определим есть ли наш небезызвестный герой 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) # обрабатываем изображения, причем y_test будет нулём
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()
```

## 5 Результаты

Итак, спустя 64 эпохи мы получили точность около 80%. Ниже приведём долгожданные графики, описанные выше:

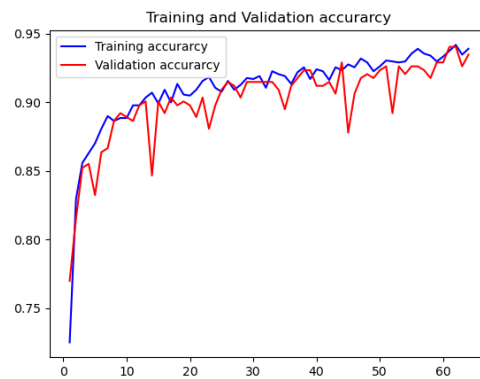


Рисунок 8. асс(epoch)

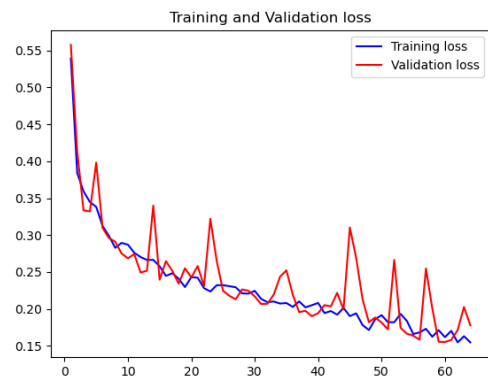


Рисунок 9. loss(epoch)

Выводы по этим двум графикам:



- Следует отметить, что мы не переобучаемся, поскольку точность модели на обучающих данных довольно близка с точностью модели на проверочных данных (точность на данных для обучения существенно не превосходит точность на данных проверки, более того, они чередуются по величине)
- Можем заметить, 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 снижаются. Можно предположить, что увеличение кол-ва эпох приведет к снижению потерь.

Посмотрим теперь, как наша модель справляется на невиданных ранее изображениях (из папки *Tests*):

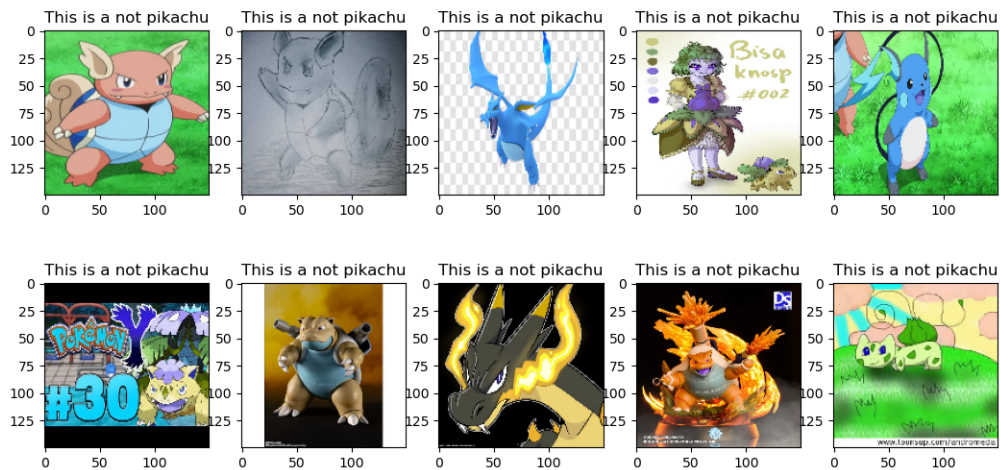


Рисунок 10. Тест 1

Как мы видим, первый тест (Рисунок 10) был успешным, однако Пикачу на них не встретилось (так как мы выбирали картинки случайно из папки *Test*)

Теперь посмотрим на второй тест (Рисунок 11) и увидим, что наша модель ошиблась и дала ложноположительный результат, это указывает на то, что наша модель несовершенна.

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.

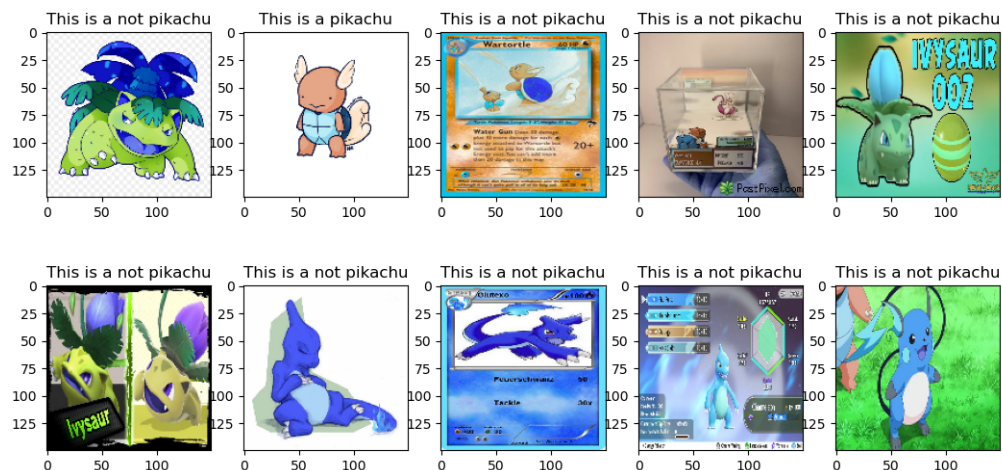


Рисунок 11. Тест 2

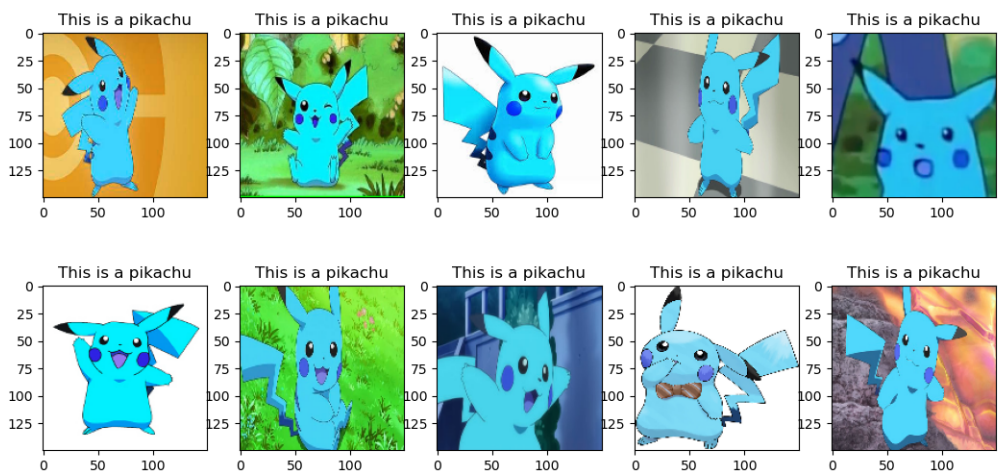


Рисунок 12. Тест 3

Восстановив модель, мы получим точность 74,2% на картинках, которые модель увидела впервые, такой результат вполне объясним. Наконец перейдем к выводам.

## 6 Краткие выводы

Мы построили и обучили свёрточную нейронную сеть, которая даёт точность около 80% на данных для проверки, при этом обучение состояло из 64 эпох. На графиках мы увидели, что с увеличением кол-ва эпох точность растёт,

а потери сокращаются, а так же важен тот факт, что мы не переобучили модель. Можем предположить, что если мы увеличим число эпох, то наша модель станет более точной. 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 дало бы нам также более высокую точность. Но даже при таких условиях мы достигли нужный нам результат.



**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
Проверка отчета	Владимирова Элина, Евдокимов Данил
Подбор данных	Евдокимов Данил, Владимирова Элина, Стельмах Татьяна Назаров Максим