

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

ОТЧЕТ
по Индивидуальному заданию
по дисциплине «Нейронные сети - Магистратура»
Тема: «White Blood Cell segmentation»

Студент гр. 7304

Государкин Я.С.

Студент гр. 7304

Токарев А.П.

Преподаватель

Субботин А.Н.

Санкт-Петербург

2022

Постановка задачи.

Описание исходного датасета

Первый датасет, который описывается авторами (его и предлагается использовать) состоит из 300 изображений 120x120 (размеры некоторых могут быть меньше этих размеров, однако несущественно), глубиной 24 бита снятых с электронного микроскопа снимков клеток белой крови (белокровие или лейкемия, это особая болезнь крови). Также, есть 300 размеченных специалистами масок этих снимков для 3 классов: ядро дефектной кровяной клетки, плазма дефектной клетки, и фон в виде клеток нормальной красной крови. Примеры с [официального репозитория](#) (сверху оригинальные изображения, снизу их сегментационные маски):

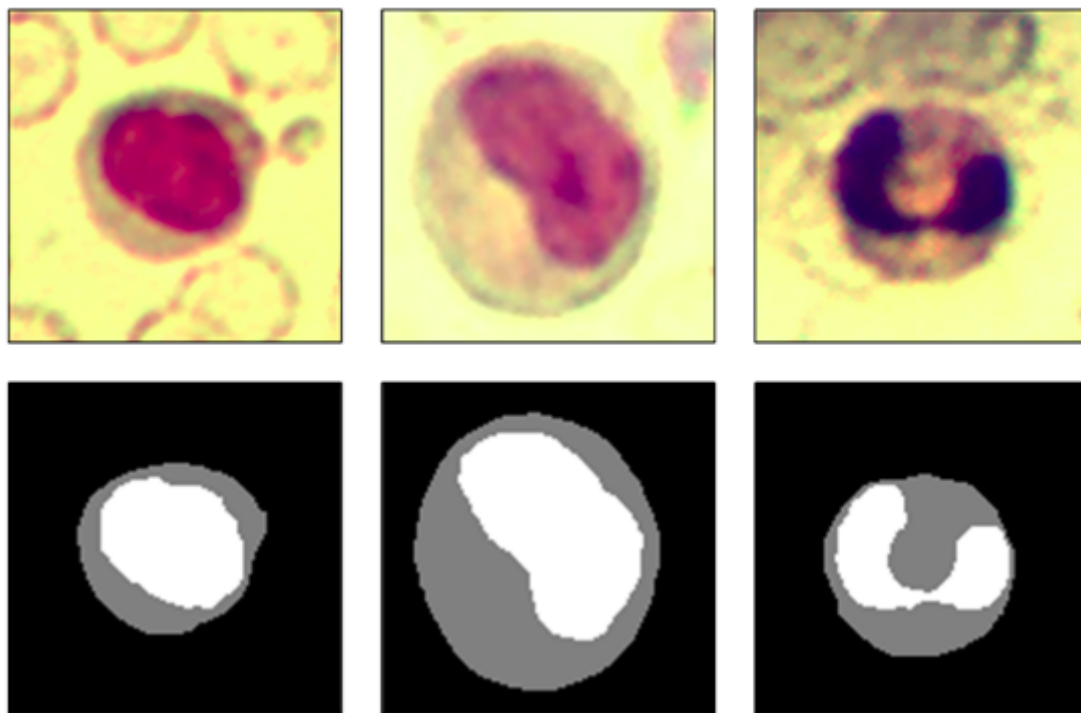


Рисунок 1 - пример изображений из исходного датасета и их масок

Задача

Предлагается решить задачу многоклассовой сегментации снимков клеток белой крови, научить сеть находить области на изображении относящиеся к разным классам, классы описаны выше. Результат представить в виде

какой-либо метрики, подходящей для задачи сегментации изображений, а также примерами масок, которые предсказывает сеть и соотнесение их с масками, которые были размечены в исходном датасете.

Ход работы.

Анализ датасета

Датасет, как было сказано выше состоит из 2 типов изображений:

1. Снимки клеток крови с микроскопа 120x120
2. Маски каждого снимка, также 120x120

Все изображения хранятся в одной папке. Первый тип - bmp картинки, второй - png картинки, соотносятся по номеру, пример:

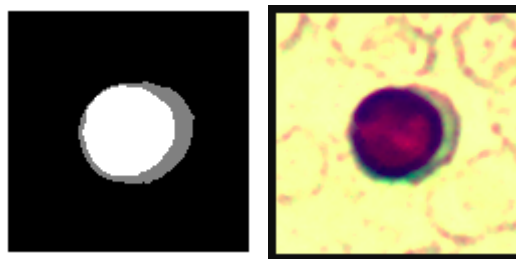


Рисунок 2 - файл 001.png (слева), файл 001.bmp (справа)

Датасет состоит из 300 пар маска\снимок, что довольно мало, и это отрицательно скажется на точности предсказаний сети.

Скачивание и предобработка датасета

В официальном репозитории есть прямая ссылка на скачивание датасета. Воспользуемся ей и напишем функцию *download*, которая скачает датасет в папку со скриптом, а также распакует zip архив. Код:

```
def download(url, name='wbc.zip'):
    if os.path.isfile('./' + name):
        return

    # download file
    dir = './'
    filename = os.path.join(dir, name)
    if not os.path.isfile(filename):
        urllib.request.urlretrieve(url, filename)

    # unzip downloaded file
    with ZipFile(name, 'r') as zipObj:
        zipObj.extractall()
```

Рисунок 3 - код функций загрузки датасета и распаковки

Следующий этап - загрузка датасета в память скрипта. Датасет состоит из картинок, для загрузки воспользуемся библиотекой Pillow, затем преобразуем каждое изображение в numpy массив.

Также, понадобится стандартизировать все изображения до одинаковых размеров, 128x128 (почему именно такой размер - написано далее в отчете) с помощью метода `resize`. Чтобы избежать [antialias-a](#) введем интерполяцию при расширении методом [ближайшего пикселя](#), при его использовании сегментационные маски не пострадают и останутся трехцветными, на стыках цветов не будет никаких промежуточных цветов. Код:

```
def load_images(dir='./segmentation_WBC-master/Dataset 1/'):
    screens = []
    masks = []
    # foreach img in folder
    list = os.listdir(dir)
    # sort files names in list
    list.sort()
    for f in list:
        img_format = os.path.splitext(f)[1]
        img = PIL.Image.open(dir + f)
        img.load()
        img = img.resize((128, 128), PIL.Image.NEAREST)
        img = np.asarray(img)
        if img_format == '.bmp':
            screens.append(img)
        else:
            masks.append(img)
    return np.asarray(screens), np.asarray(masks)
```

Рисунок 4 - код функции *load_images*

Изображения загружены, стандартизированы, теперь нужно их нормализовать по значению пикселей. Делается это делением каждого пикселя на всех снимках на величину байта (т.к. наши снимки 24-битные, соответственно, 8 бит или 1 байт на цвет). Маски же должны быть обработаны так, чтобы разные цвета составляли разные классы. Для этого необходимо преобразовать исходные маски с помощью [one-hot кодирования](#) в категориальные матрицы (когда вместо пикселя конкретного класса подставляется бинарный вектор) для каждого класса. Однако тут есть проблема, маски имеют значения пикселей 0, 128, 255, а особенность работы функции `to_categorical` такова, что при её использовании на таких масках будет инициализировано 255 классов, а не 3, что нам не нужно. Поскольку мы знаем, что классов 3, то можем вручную задать 1 и 2 значения вместо 128 и 255, фон уже по умолчанию 0. Также, разделим наш исходный датасет на train и test выборки. Код:

```
# normalize, categorize and split dataset
screens = screens / 255.
np.place(masks, masks == 128, [1])
np.place(masks, masks == 255, [2])
(train_x, test_x, train_y, test_y) = train_test_split(screens, masks, test_size=0.3)

enc_train_y = to_categorical(train_y)
enc_test_y = to_categorical(test_y)
```

Рисунок 5 - код нормализации, категоризации, разбиения датасета и кодирования маркированных масок

Подбор параметров обучения

Данные подготовлены, теперь необходимо подобрать некоторые [гиперпараметры](#) сети. подбираемые для начала параметры:

- Оптимизатор. В первом приближении стоит использовать Adam или SGD Nesterov с моментом, потому что они наиболее универсальные для многих задач. Выберем Adam, с lr по умолчанию.
- Лосс-функция. Поскольку решается задача многоклассовой сегментации, то стоит использовать Categorical Crossentropy в связке с выходной функцией активации softmax.
- Метрика точности сети. Это показатель того, насколько сеть хорошо делает предсказания. Для задач сегментации часто используют [меру Жаккара](#) (MeanIoU), или [меру Сёренсена](#) (MeanDice), выберем последнюю.
- Начальное кол-во эпох. Допустим будет 5.
- Batch-size. Поскольку датасет небольшой, то размер батча будет 2.

Разработка начальной модели

Параметры подготовлены, теперь можно создать начальную модель сети и попробовать протестировать её на датасете. Задача сегментации сводится к тому, что на вход сети подается картинка некоторого размера, а на выходе должна получиться картинка такого же размера, которая является

сегментационной маской исходной картинки. Такие задачи решаются сетями имеющими архитектуру Full-Convolutional-Network. Кратко, суть архитектуры в том, чтобы сначала сворачивать (pooling) картинку, а потом разворачивать (upsampling) до исходного размера.

Вернемся к вопросу о размере картинки. Для того, чтобы сохранить размерность картинки, на выходе нужно следить за тем, чтобы при свертке остаток от деления высоты и ширины картинки на шаг пулинга (pool_size) был всегда равен нулю, иначе выходная картинка будет меньше исходной. Чтобы такого не было, исходные картинки расширяются до 128x128, что позволяет делать свёртку и развертку с шагом 2 без потери исходных размеров. Это будет примерно выглядеть как 128x128 -> 64x64 -> 32x32 -> 64x64 -> 128x128.

На выходе должна стоять активация (softmax), которая на каждый пиксель выдаст вектор вероятностей принадлежности этого пикселя к конкретному классу, т.к. классов 3, то вектор будет размера 3x1, это закодированное изображение.

Здесь есть тонкость. Чтобы посмотреть на маску в виде картинки, необходимо ее декодировать. Это делается с помощью функции argmax, которая возвращает индекс (в нашем случае класс) наибольшего элемента массива. Если применить эту функцию к каждому вектору вероятностей, то получится маркированное изображение (сегментационная маска). Реализуем это в виде функции, получающей на вход закодированный массив картинок, возвращающий декодированный массив масок:

```
def decodeImgs(encoded_batch):  
    decodedBatch = []  
    for k in range(0, encoded_batch.shape[0]):  
        decodedBatch.append(np.argmax(encoded_batch[k], axis=-1))  
    return np.asarray(decodedBatch)
```

Рисунок 6 - код функции декодирования изображения

Начальная модель будет состоять из conv слоев, слоев pooling, и слоев upsampling. По выше описанной архитектуре сначала должна быть свертка с пуллингом, затем свертка с upsampling-ом:

```

model = Sequential([
    Conv2D(filters=16, kernel_size=(5, 5), padding='same', activation='relu'),
    MaxPool2D(pool_size=(2, 2)),

    Conv2D(filters=16, kernel_size=(5, 5), padding='same', activation='relu'),
    UpSampling2D(size=(2, 2)),

    Conv2D(filters=num_classes, kernel_size=(1, 1), activation='softmax'),
])

```

Рисунок 7 - код начальной модели

Модель построена, теперь можно обучить её методом `fit`, и оценить результат.

Модель 1

При попытке обучить модель на этом этапе возникла ситуация, что машина полностью зависла, и пришлось её перезагружать. Проблема оказалась в том, что наш скрипт хранит в памяти все изображения, где каждый пиксель - вектор 0\1, обозначающий принадлежность пикселя к классу. Это забило полностью ОЗУ машины. Чтобы этого избежать будем обучать сеть подавая ей на вход не весь датасет а батчи отдельно, и перед тем как подать батч преобразовывать его в one-hot код. Это вызывает необходимость рандомной перестановки (permutation) индексов изображений, чтобы на каждой эпохе батчи были разными.

Метрики будут выдаваться моделью посчитанные в каждом конкретном батче, поэтому `mean` (среднее), нам тоже нужно будет считать самим.

Такой подход дает возможность удобно на нужной эпохе оценить точность модели подав ей на вход тестовую выборку (она небольшая, и её можно целиком приводить в one-hot), а также вывести на экран предсказания сети в виде изображений.

Полный алгоритм обучения сети, вывода изображений на каждой N эпохе, а также, подсчета и вывода необходимых метрик:


```

# fit model
for i in range(1, epochs):
    print('Epoch:', i)
    train_loss = 0
    train_dice = 0

    # permute the indexes to get random batch
    batch_indexes = np.random.permutation(len(train_x))
    # one epoch fitting
    for k in range(0, len(train_x)-batch_size, batch_size):
        # train model by current batch
        batch_x = train_x[batch_indexes[k:(k+batch_size)]]
        batch_y = to_categorical(train_y[batch_indexes[k:(k+batch_size)]], num_classes=num_classes)
        his = model.train_on_batch(batch_x, batch_y)
        train_loss += his[0]

    # evaluate DICE
    batch_preds = model.predict(batch_x)
    train_dice += his[1]

    print('Train CCE loss:', train_loss * batch_size/len(train_x))
    print('Train DICE metric:', train_dice * batch_size/len(train_x))
    his = model.evaluate(test_x, to_categorical(test_y), verbose=0)
    print('Test DICE metric:', his[1])
    print('-----')

    # each N-th epoch show test predictions
    if i % show_interval == 0:
        outputs = model.predict(test_x)
        decoded = decode_imgs(outputs)
        # show some screens/masks/preds
        batch_show(test_x, test_y, decoded)

```

Рисунок 8 - алгоритм обучения и оценки точности сети

Каждые 2 эпохи будут выводиться изображения, спрогнозированные сетью, причем одни и те же. Это позволит от модели к модели следить за прогрессом в увеличении точности.

На 1 эпохе предсказания сети:

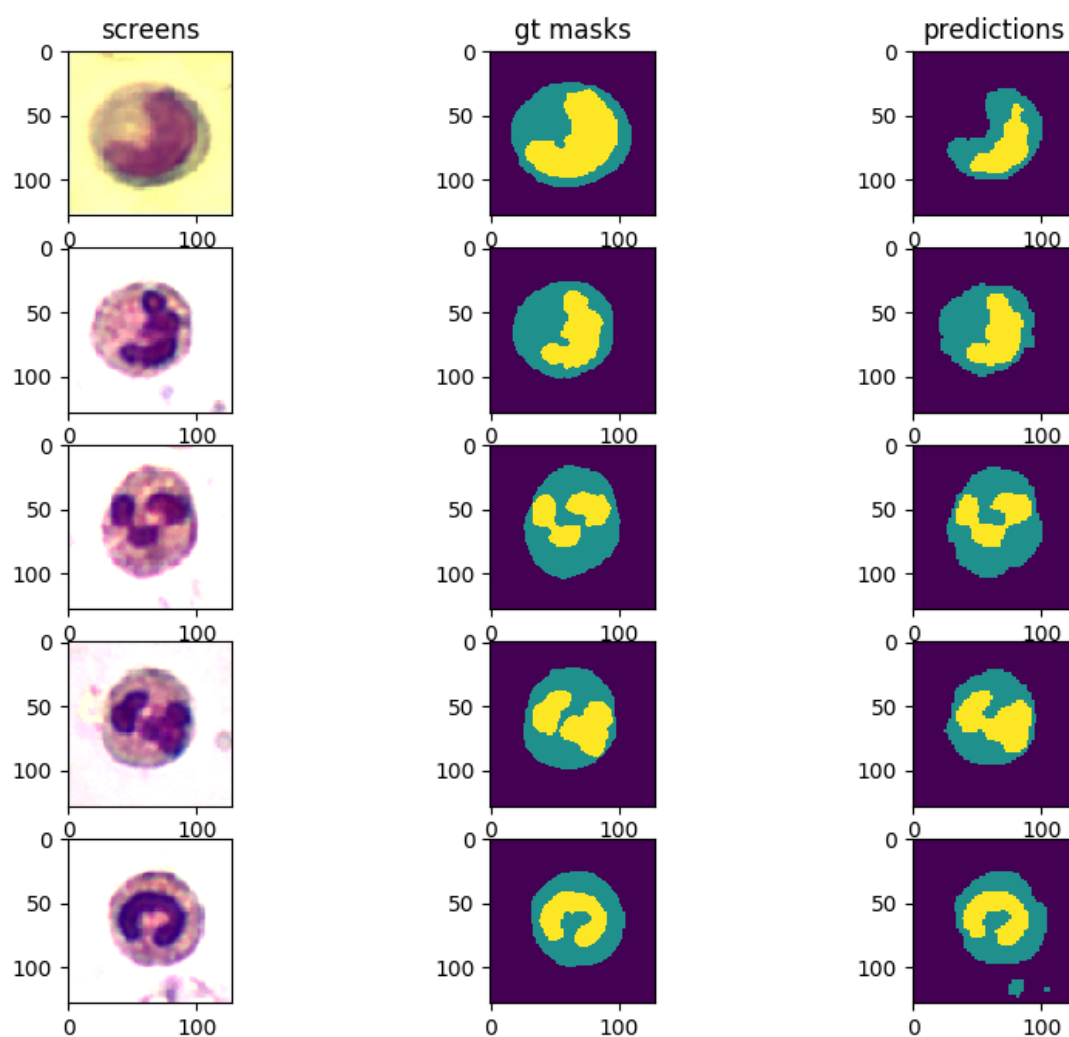


Рисунок 9 - слева направо: снимки, исходные маски, предсказания модели 1
после 1 эпохи

На 5 эпохе предсказания сети:

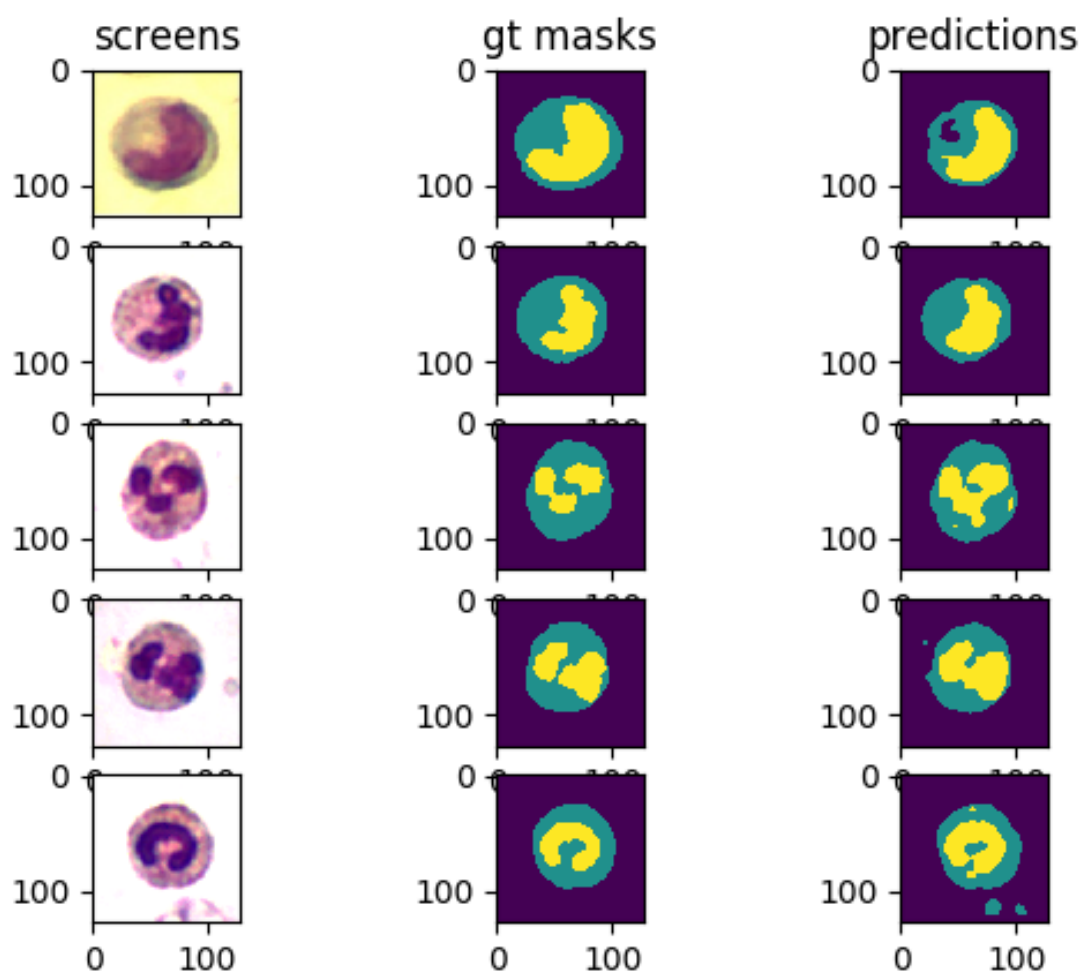


Рисунок 10 - слева направо: снимки, исходные маски, предсказания модели 1
после 5 эпохи

Можно заметить, что модель после первой же эпохи дает неплохой результат в предсказаниях. После 5 эпохи на предсказанных масках пропадают “заусенцы” на краях клетки, однако, немного хуже определяется ядро.

Метрика MeanDice на тестовых данных после обучения составила 0.86.

Модель 2

Чтобы улучшить результат усложним сеть. Также, чтобы ускорить обучение и увеличить его качество добавим слои BatchNormalization. Они нормализуют выходы в отрезок $[0, 1]$ и тем самым упростят тренировку сети:

```

model = Sequential([
    Conv2D(filters=16, kernel_size=(5, 5), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPool2D(pool_size=(2, 2)),
    Conv2D(filters=32, kernel_size=(5, 5), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPool2D(pool_size=(2, 2)),

    Conv2D(filters=32, kernel_size=(5, 5), padding='same', activation='relu'),
    BatchNormalization(),
    UpSampling2D(size=(2, 2)),
    Conv2D(filters=16, kernel_size=(5, 5), padding='same', activation='relu'),
    BatchNormalization(),
    UpSampling2D(size=(2, 2)),

    Conv2D(filters=num_classes, kernel_size=(1, 1), activation='softmax'),
])

```

Рисунок 11 - код модели 2

Будем выводить графики метрики dice от эпохи, чтобы следить за переобучением:

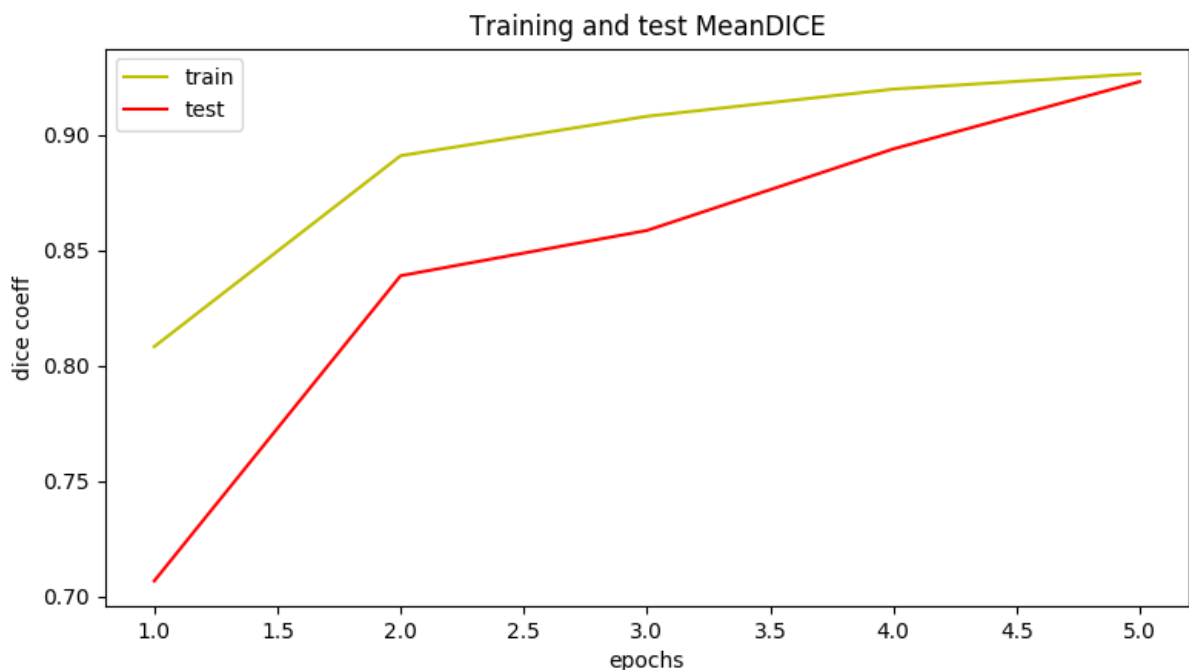


Рисунок 12 - график зависимости метрики MeanDICE от эпохи

На первой эпохе предсказания были:

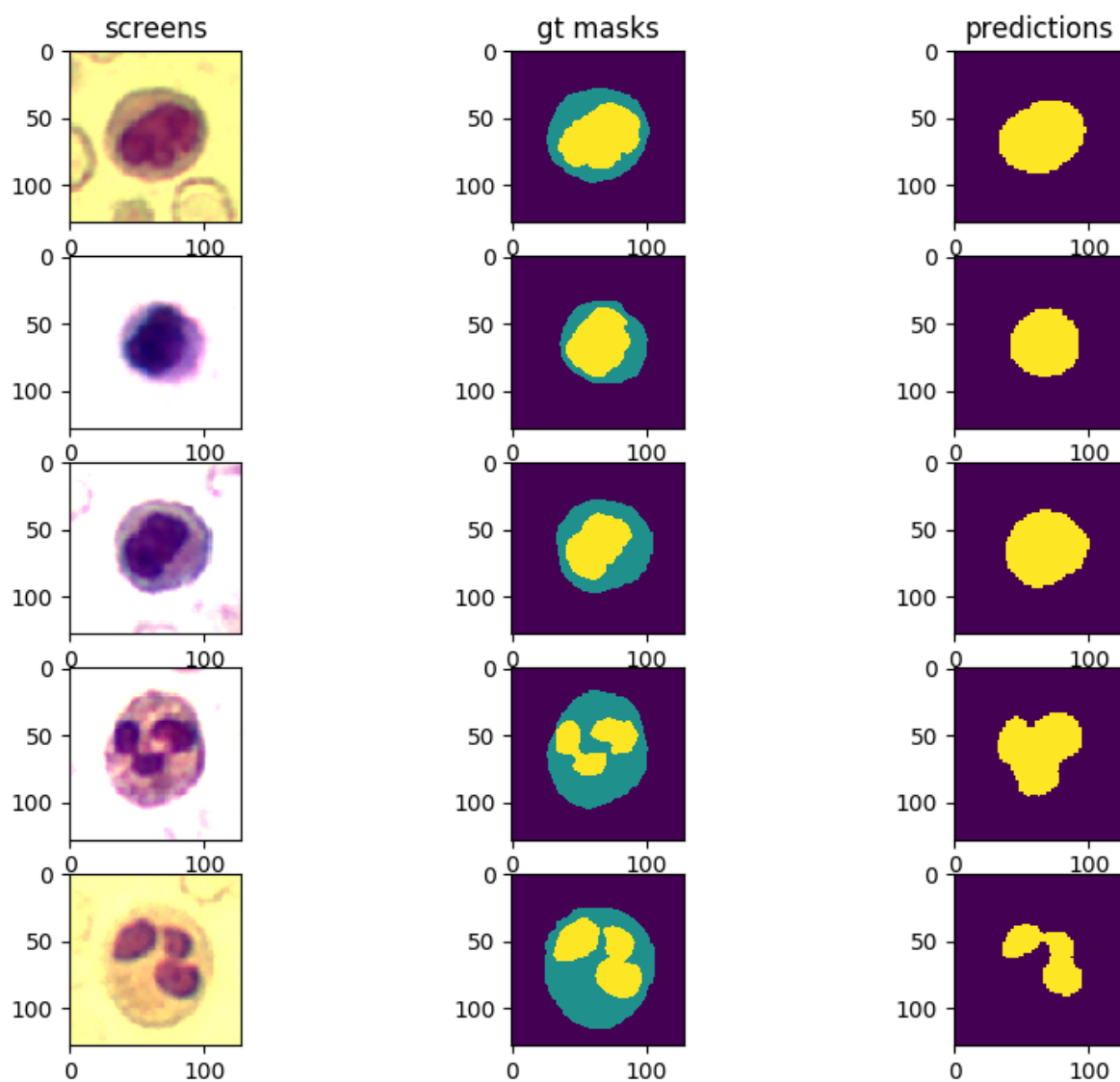


Рисунок 13 - слева направо: снимки, исходные маски, предсказания модели 2
после 1 эпохи

После 5 эпохи предсказания модели были:

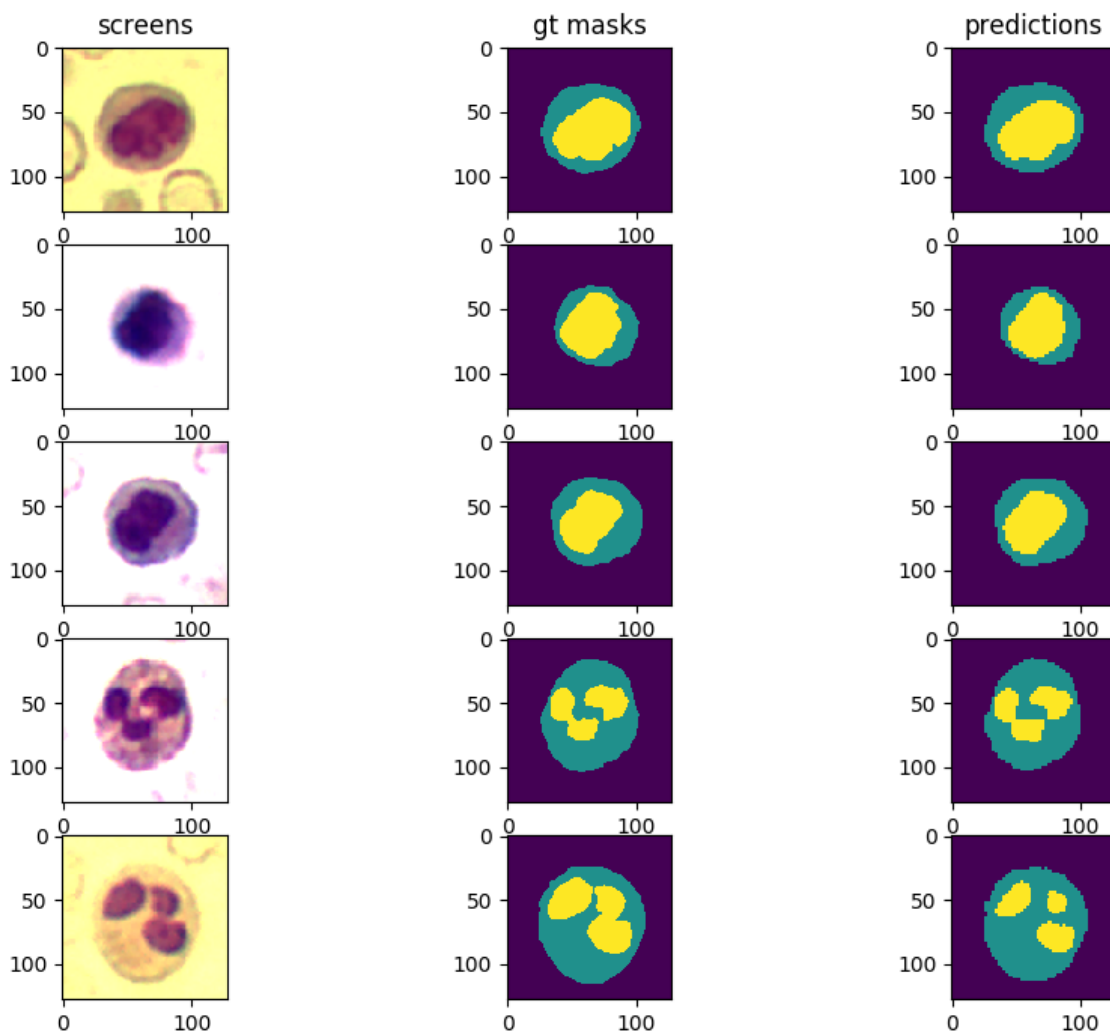


Рисунок 14 - слева направо: снимки, исходные маски, предсказания модели 2
после 5 эпохи

Несмотря на то, что на 1 эпохе прогноз плохой, после завершения обучения точность на тестовых данных стала выше, а именно MeanDice составила 0.92, что уже можно считать хорошим результатом. Построенные сетью маски очень похожи на исходные маски, а некоторые почти точь-в-точь их повторяют.

Модель 3

Результат предсказания сети все ещё можно улучшить, если немного снизить переобучение и увеличить кол-во эпох. Переобучение попробуем уменьшить использованием в слоях свертки L2 регуляризации, кол-во эпох увеличим до 20.

```
model = Sequential([
    Conv2D(filters=16, kernel_size=(5, 5), padding='same', activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    MaxPool2D(pool_size=(2, 2)),
    Conv2D(filters=32, kernel_size=(5, 5), padding='same', activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    MaxPool2D(pool_size=(2, 2)),

    Conv2D(filters=32, kernel_size=(5, 5), padding='same', activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    UpSampling2D(size=(2, 2)),
    Conv2D(filters=16, kernel_size=(5, 5), padding='same', activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    UpSampling2D(size=(2, 2)),

    Conv2D(filters=num_classes, kernel_size=(1, 1), activation='softmax'),
])
```

Рисунок 15 - код модели 3

График метрики MeanDICE:

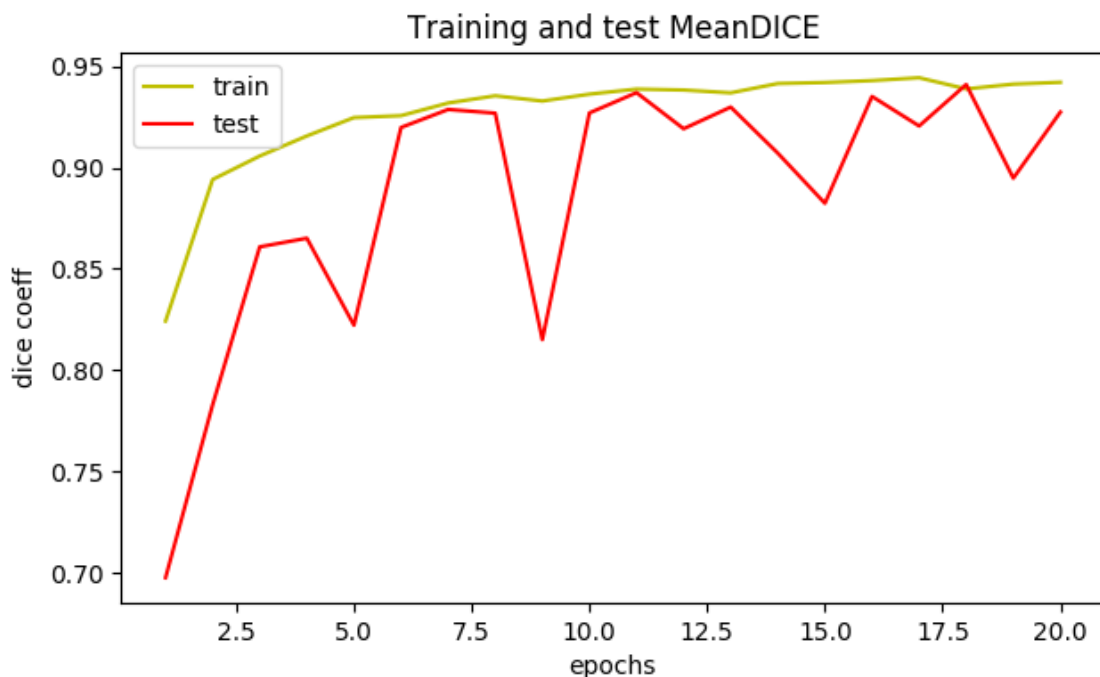


Рисунок 16 - график зависимости метрики MeanDICE от эпохи

Как можно видеть, переобучение свелось к минимуму и, в около половине случаев, значение dice коэффициента на тестовой выборке такое же или чем на тренировочной. Лучшая точность - 0.942, что можно считать удовлетворительным результатом.

Предсказания сети после 1 эпохи:

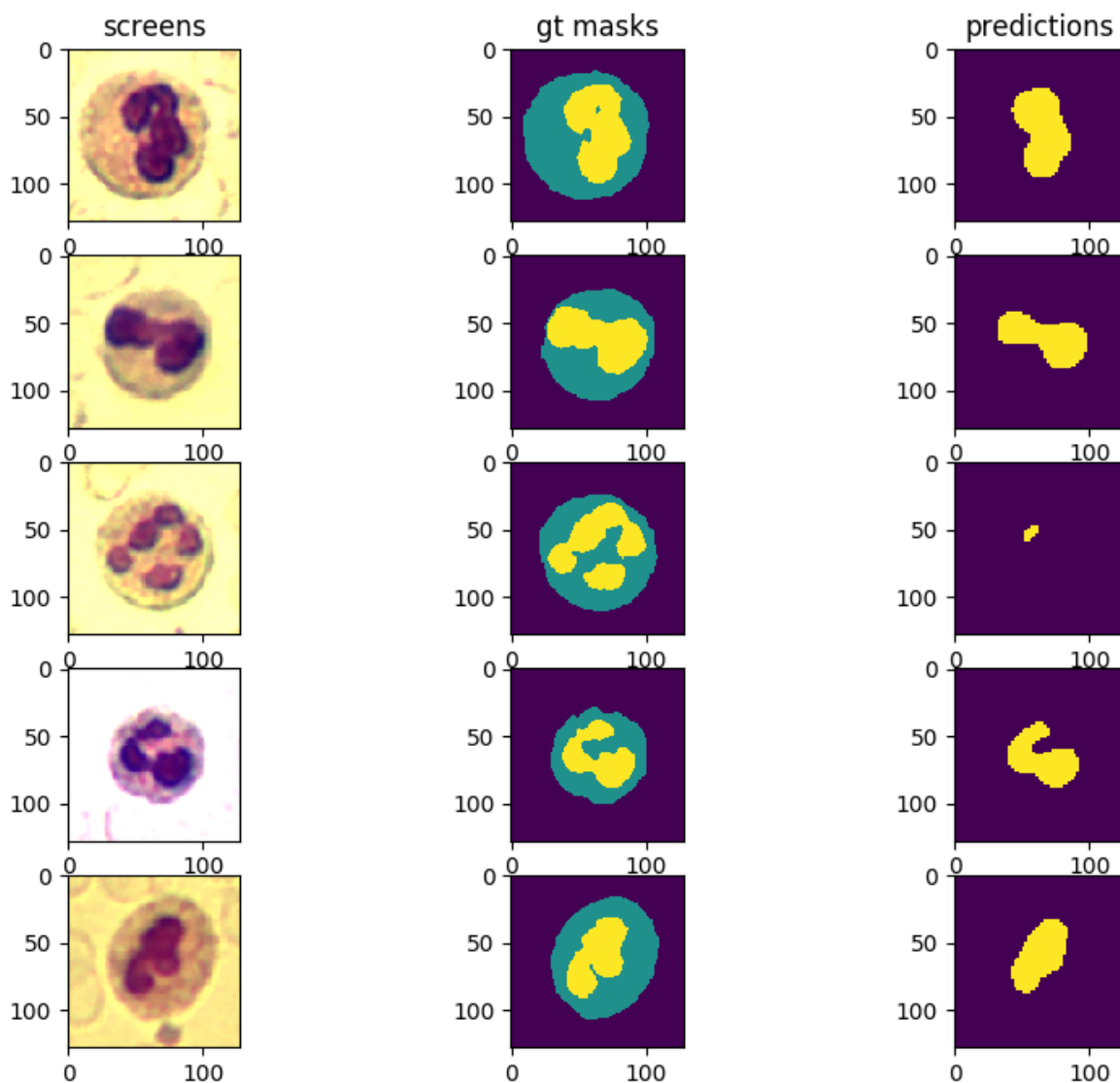


Рисунок 17 - слева направо: снимки, исходные маски, предсказания модели 3 после 1 эпохи

Предсказания сети после 10 эпохи (метрика - 0.91):

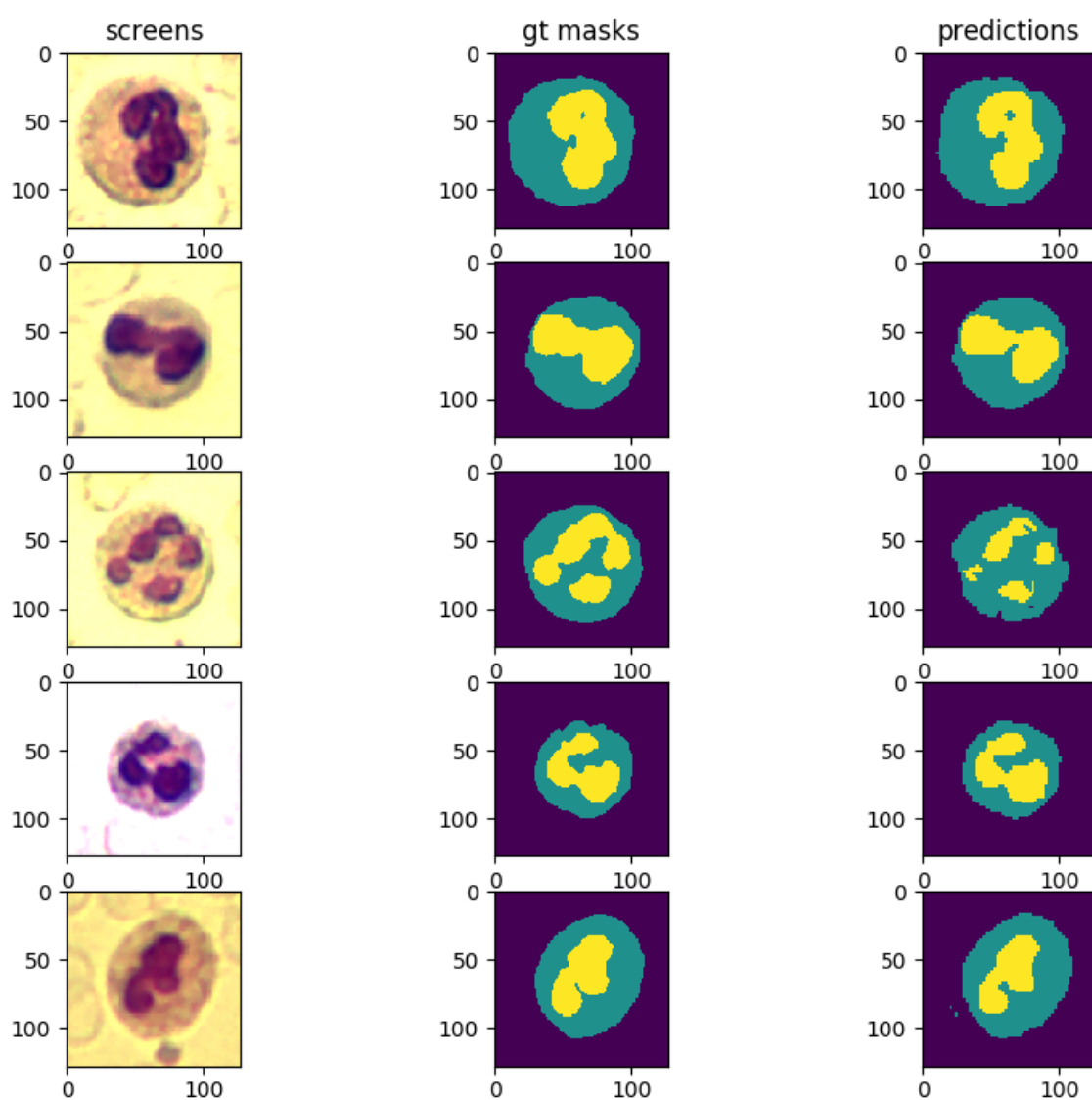


Рисунок 18 - слева направо: снимки, исходные маски, предсказания модели 3
после 10 эпохи

Предсказания сети после 18 эпохи (метрика - 0.942):

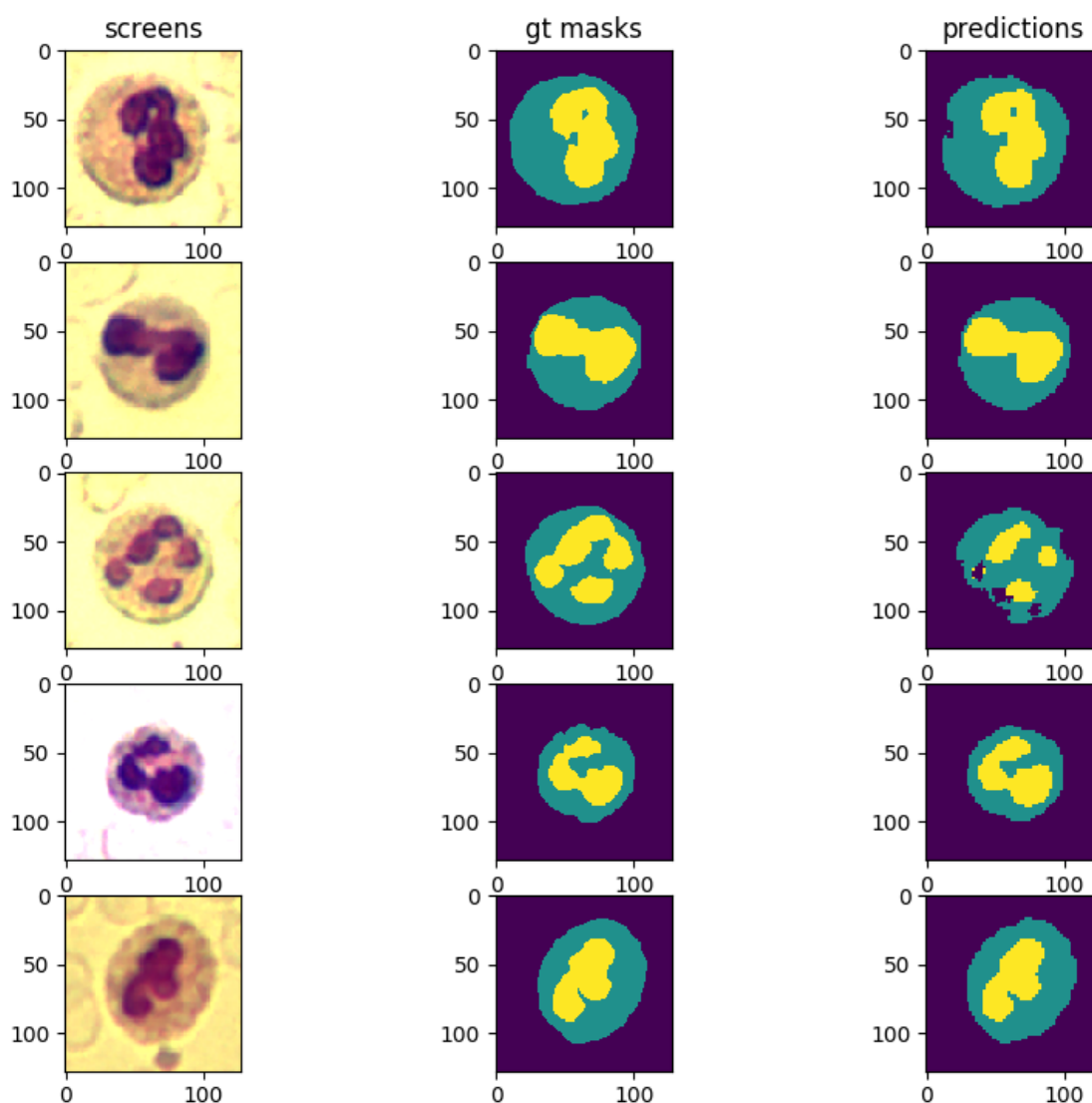


Рисунок 19 - слева направо: снимки, исходные маски, предсказания модели 3 после 18 эпохи

Результаты

Анализ конечной модели

Конечная модель (3) выдает точность (значение метрики MeanDICE) на тестовых данных ~ 0.94 , что является хорошим результатом. По результирующим изображениям также видно, что они предсказываются достаточно точно, несмотря на отклонения в некоторых ситуациях. Характер

отклонений такой, что предсказание либо почти идентично исходной маске, либо имеет значительный промах.

Решенные проблемы

В данной работе сложнее всего было правильно подготовить исходный датасет так, чтобы сеть смогла его правильно обработать и выдать адекватный результат. Проблем в том, чтобы достичь хорошей точности на тестовом наборе данных было меньше, самая первая же модель показывала довольно неплохой результат.

Первая проблема - разный размер исходных картинок в датасете. Каждая картинка имела свой размер, что не позволяло использовать их сразу в обучении сети. Чтобы решить проблему понадобилось стандартизировать все изображения до одинаковых размеров с помощью `resize` и отключить сглаживание при этой операции.

Вторая проблема - подготовка картинок к попиксельной классификации. Маски должны быть обработаны так, чтобы разные цвета составляли разные классы, однако возникла проблема с работой утилиты `keras to_categorical`. Решено было попиксельной категоризацией каждой исходной маски, затем кодированием в `one-hot`.

Третья проблема - декодирование закодированных масок, которые возвращает модель. Это потребовалось, чтобы можно было посмотреть на результат в виде изображения, чтобы это сделать потребовалось применить к выходным маскам `argmax`, т.е функцию, которая возвращает индекс (в нашем случае класс) наибольшего элемента массива.

Четвертая проблема - слишком большой вес всего массива закодированных масок. Из-за представления каждого пикселя в виде `one-hot` вектора компьютеру не хватило памяти для стабильной работы. Чтобы это исправить возникла необходимость динамически кодировать небольшие батчи масок, чтобы не перегружать память. Это потребовало написать вместо вызова метода `fit` свой алгоритм обучения и использовать метод `train_on_batch`. Таким

образом перед тем как подать батч в сеть он кодируется, не потребляя одновременно много памяти. Такой подход позволил также на нужной эпохе не только подсчитывать метрику на тестовой выборке, но и вывести изображения предсказанных сетью масок на этой выборке.

Не решенные проблемы

В целом, поднять точность работы сети выше 0.94 не удалось, и с используемой архитектурой сети (описана в пункте “Разработка начальной модели”) вряд ли удастся.

Также, на протяжении обучения, несмотря на влияние регуляризации в результирующей модели наблюдается небольшое переобучение.

Улучшение точности модели

Чтобы улучшить точность модели можно пойти разными путями, а именно, совершенствовать модель, или преобразовывать датасет или и то и другое одновременно.

Повысить точность за счет преобразования датасета можно с помощью аугментации (намеренное искажение исходного изображения различными трансформациями, например, поворот на N градусов, приближение в M раз, зеркалирование вдоль осей X , Y и т.п). Такая методика расширит “кругозор” сети, и она сможет определять нужные сущности даже если они искажены, что повышает точность в общем случае.

Второй вариант - улучшить модель. Для этого нужно экспериментировать с различными конфигурациями существующей архитектуры и пробовать новые. В задачах сегментации хорошо себя зарекомендовали модели архитектуры UNet. Это специальная разновидность FCN сетей, которая хорошо работает с задачей сегментации изображений.

Выводы

Датасет “White blood cell”, используется для решения задачи сегментации белых клеток крови на разные области. Для изучения этого датасета была построена модель архитектуры Full-Convolutional-Network для решения задач сегментации, для оценки точности предсказаний сети был использован dice коэффициент и для результирующей модели он составил ~ 0.94 . Дана оценка того, каким образом можно улучшить точность модели.

Также, были описаны проблемы и их решения, которые возникли при подготовке датасета, а именно, разный размер исходных картинок в датасете, подготовка картинок к попиксельной классификации, декодирование закодированных масок, которые возвращает модель и слишком большой вес всего массива закодированных масок.