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

ОТЧЕТ

по индивидуальному заданию
по дисциплине «Искусственные нейронные сети»
Тема: White Blood Cell segmentation

Студентка гр. 7382 _____ Головина Е.С.
Студентка гр. 7383 _____ Маркова А.В.
Студентка гр. 7383 _____ Тян Е.
Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2020

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 представлены на рис. 1 (сверху оригинальные изображения, снизу их сегментационные маски):

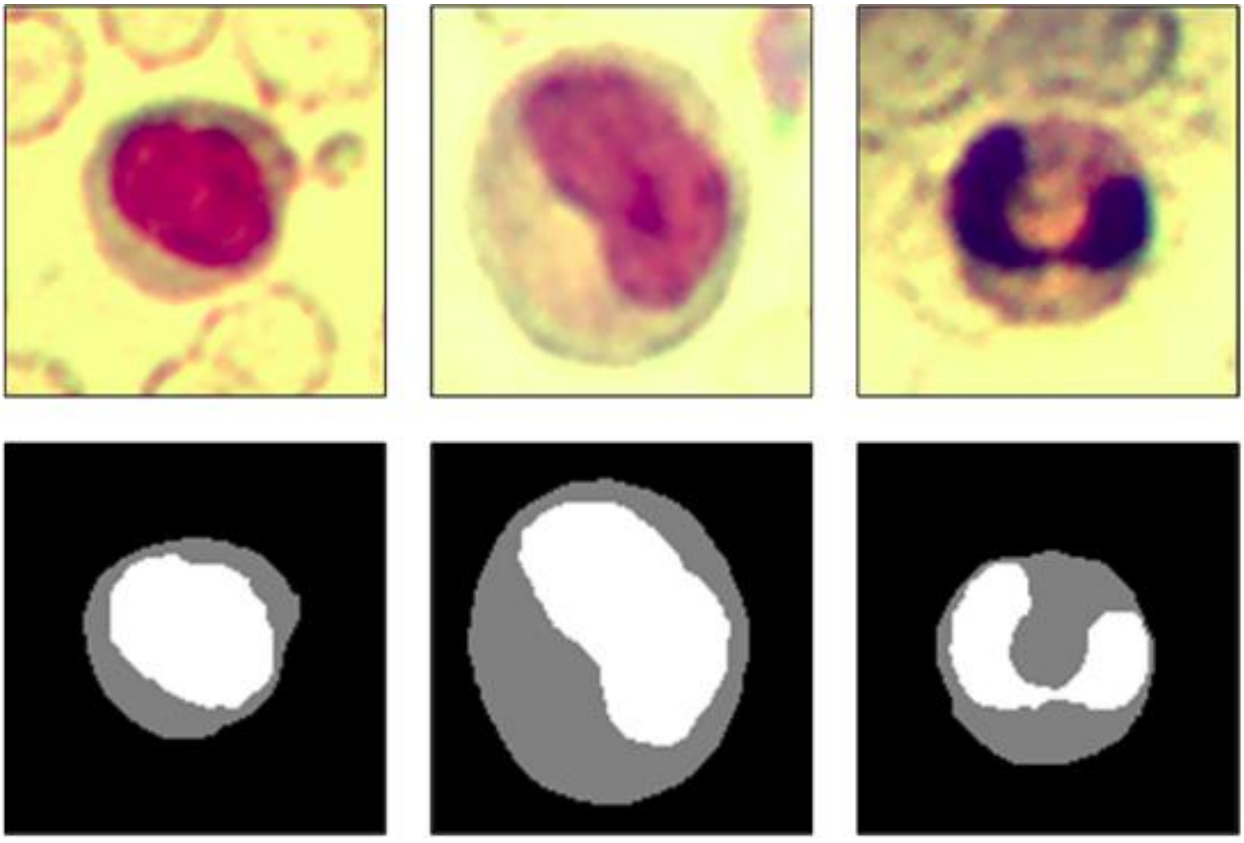

Рисунок 1 – Пример изображений из исходного датасета и их масок

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 вывести примеры масок, которые предсказывает сеть и соотнести 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 показан на рис. 2-3.

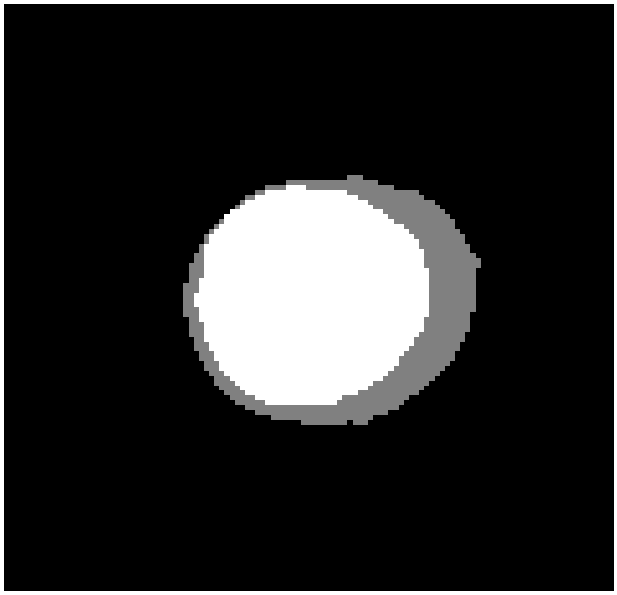

Рисунок 2 – Файл 001.png

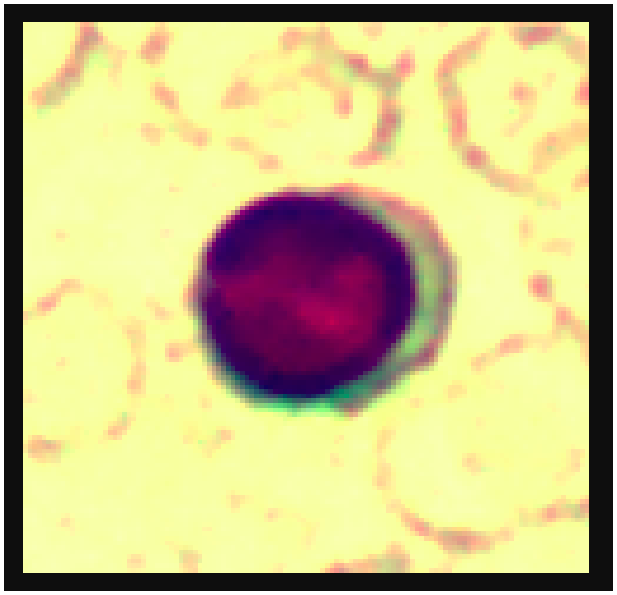

Рисунок 3 – Файл 001.bmp

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 директная ссылка на скачивание датасета. Воспользуемся ей и напишем функцию `download`, которая скачает датасет в папку со скриптом, а также распакует zip архив. Код продемонстрирован на рис. 4.

```
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 4 – 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 с помощью метода `resize`. Чтобы избежать antialias-а 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 показан на рис. 5.

```
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 5 – 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, так как 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 для каждого класса, вместо пикселя конкретного класса подставляется бинарный вектор. 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 класса 3, то можем вручную задать 1 и 2 значения вместо 128 и 255, фон

уже по умолчанию 0. Также, разделим наш исходный датасет на тренировочную и тестовую выборки. Код представлен на рис. 6.

```
# 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 6 – 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`, со скоростью обучения 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 количество 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, так как 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, код функции показан на рис. 7.

```
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 7 – 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, начальная модель продемонстрирована на рис. 8.

```
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 8 – 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.

Моделью будут выдаваться метрики, посчитанные в каждом конкретном батче, поэтому среднее значение, нам нужно будет считать самим.

Такой подход дает возможность удобно на нужной эпохе оценить точность модели подав ей на вход тестовую выборку, так как она небольшая, то 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 показан на рис. 9.

```
# 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 = decodeImgs(outputs)
        # show some screens/masks/preds
        batch_show(test_x, test_y, decoded)
```

Рисунок 9 – 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 первой эпохе результат предсказания сети представлен на рис. 10.

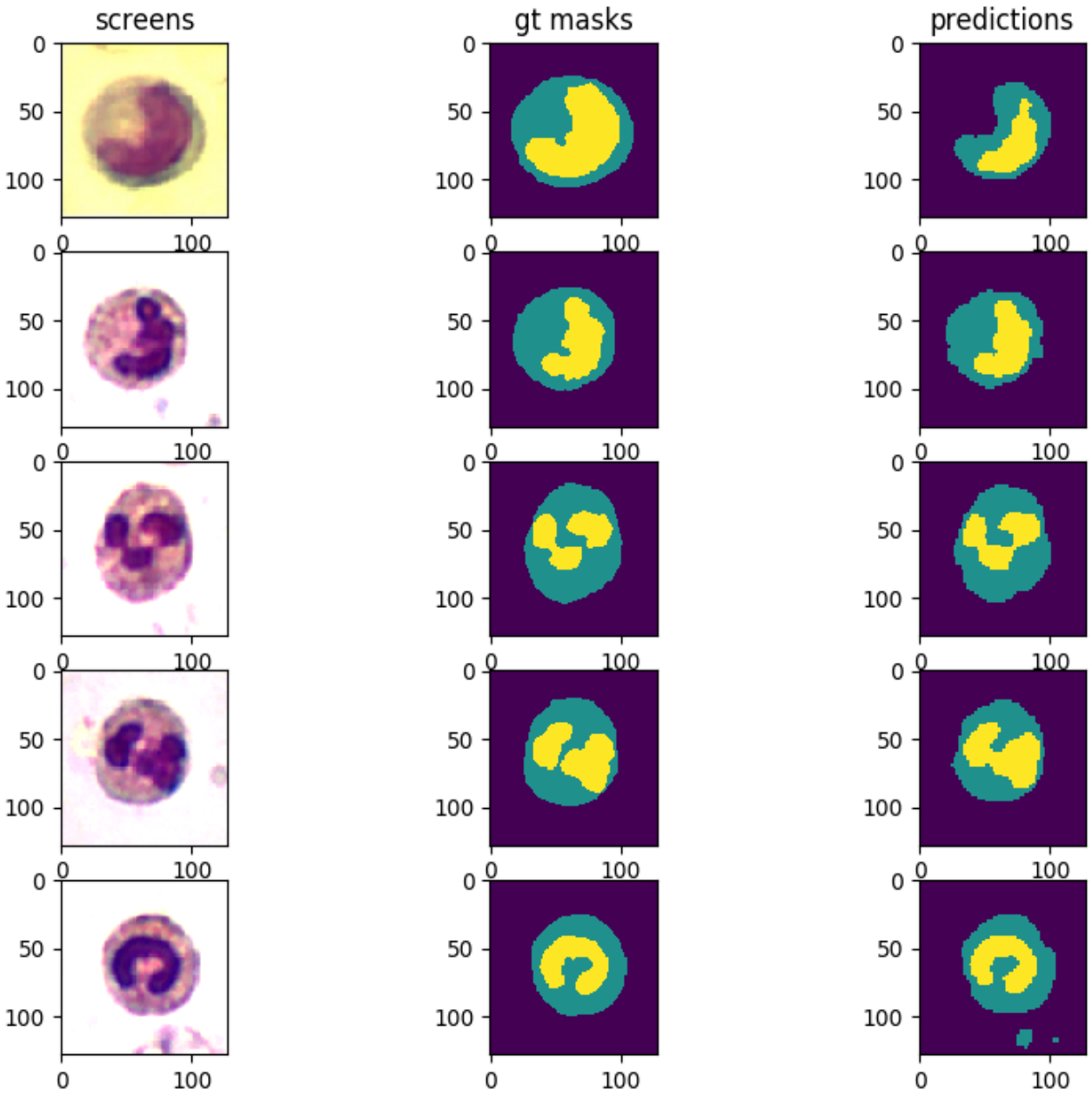

Рисунок 10 – Слева направо: снимки, исходные маски, предсказания модели

На пятой эпохи результат предсказания сети представлен на рис. 11.

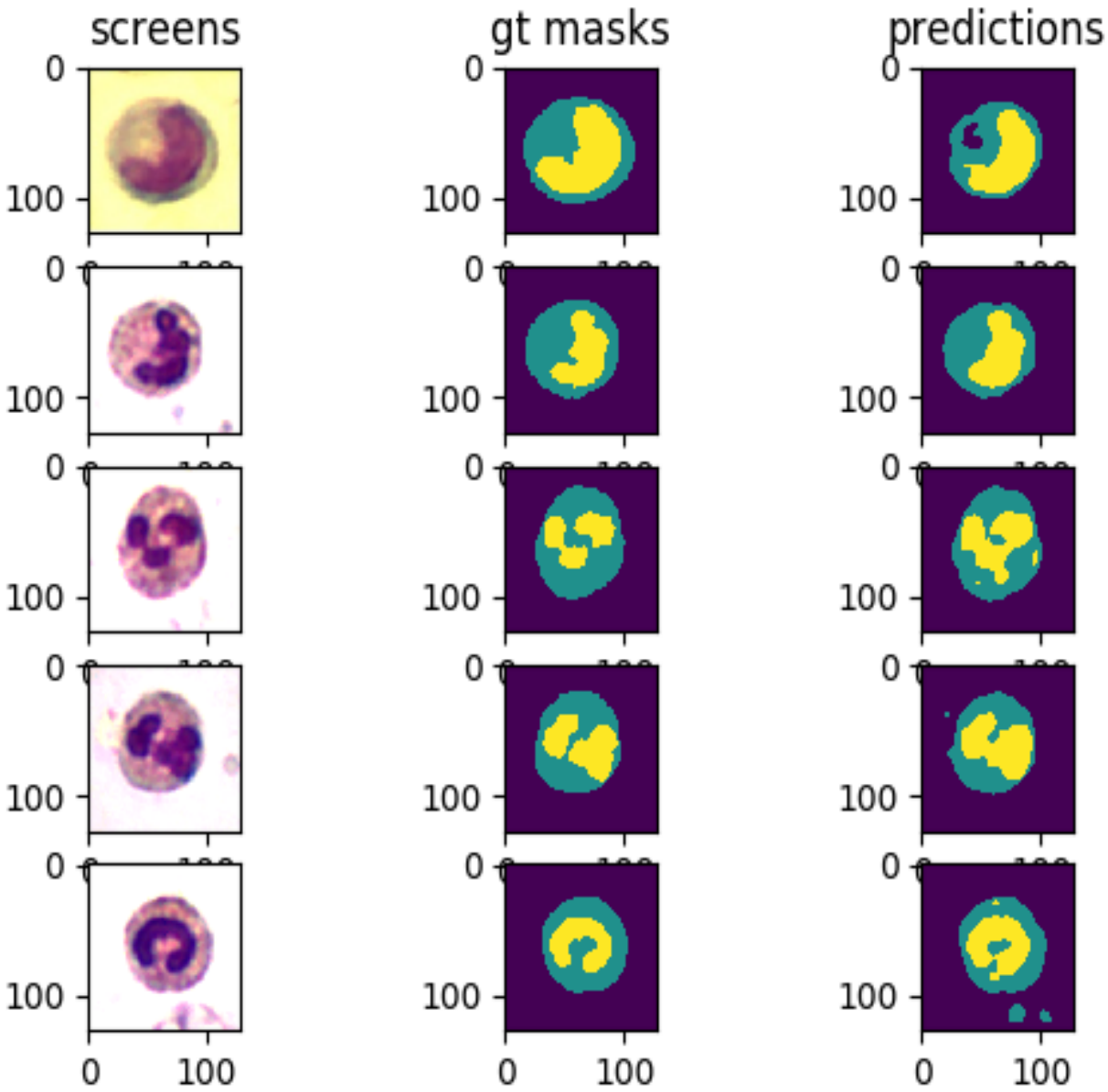

Рисунок 11 – Слева направо: снимки, исходные маски, предсказания модели

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, модель после изменений показана на рис. 12.

```
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 12 – Код модели 2

Будем выводить графики метрики `dice` от эпохи, чтобы следить за переобучением, график продемонстрирован на рис. 13.

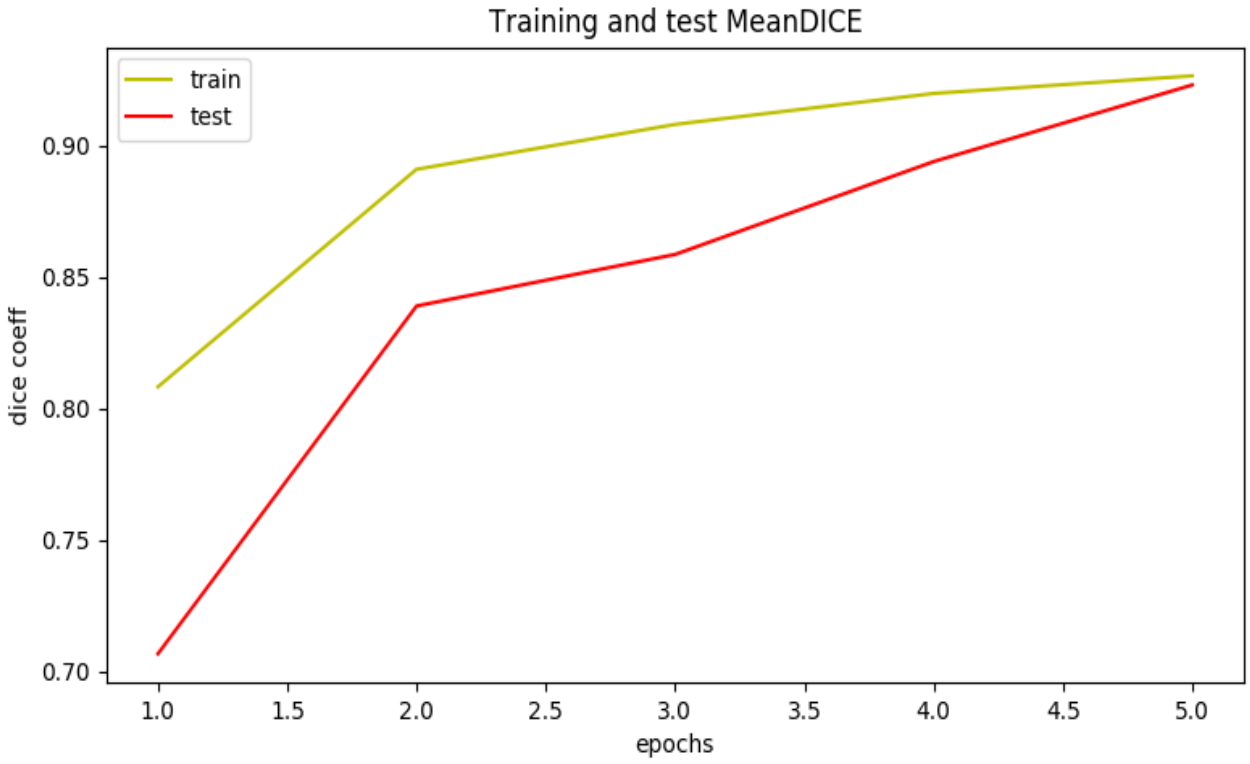

Рисунок 13 – График зависимости метрики `MeanDICE` от эпохи

На первой эпохи результат предсказания сети представлен на рис. 14.

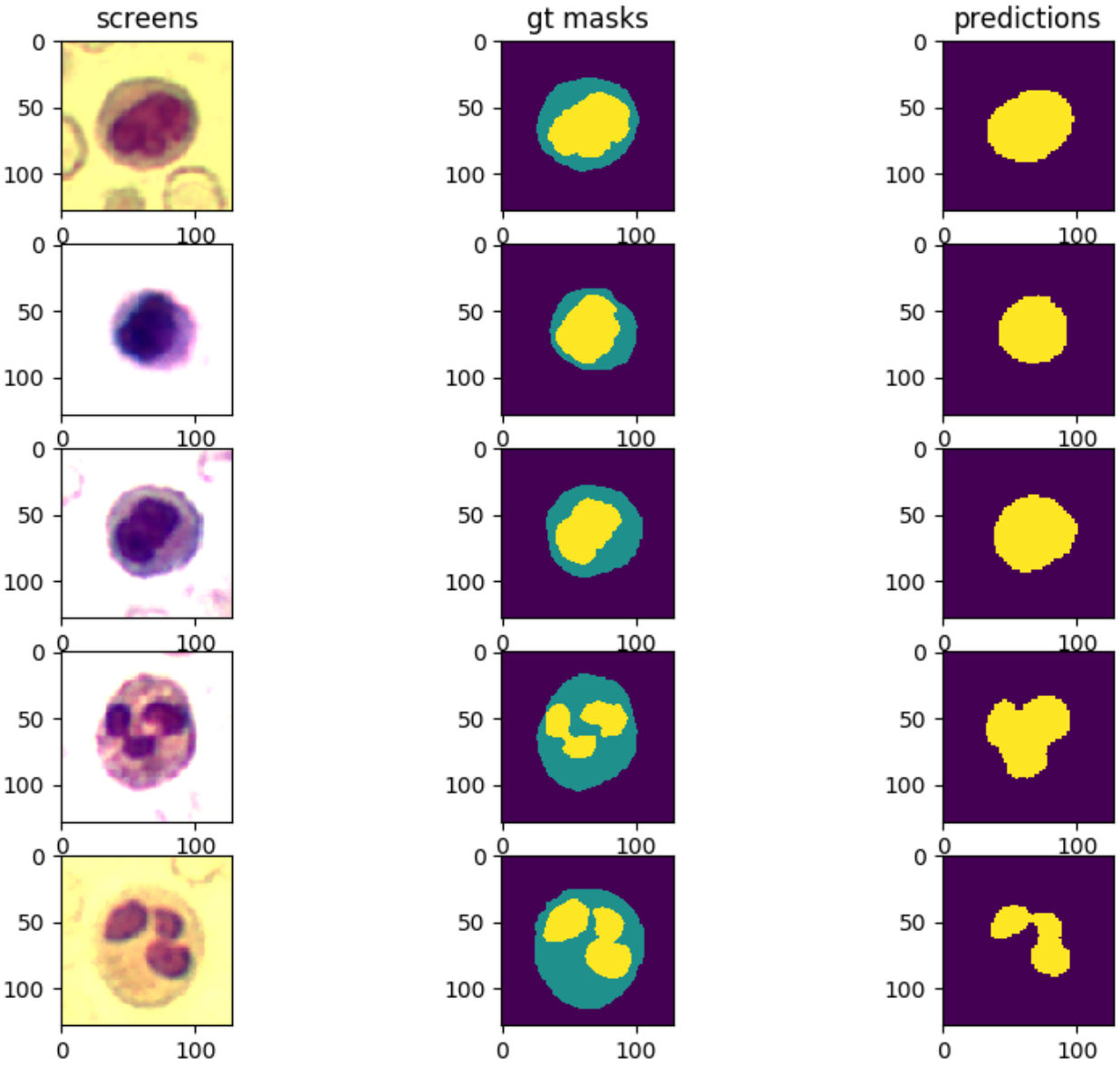

Рисунок 14 – Слева направо: снимки, исходные маски, предсказания модели

После пятой эпохи результат предсказания сети представлен на рис. 15.

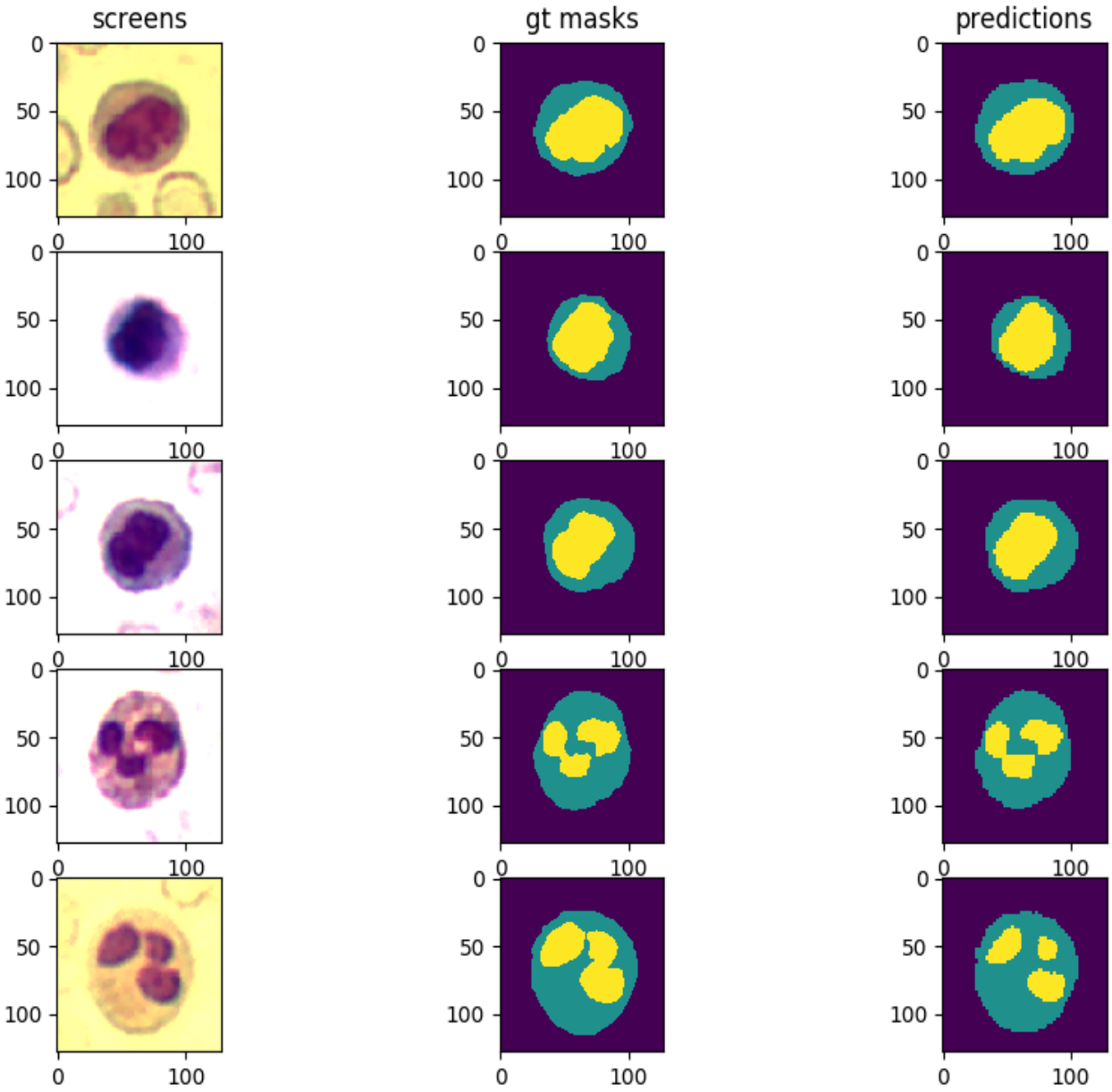

Рисунок 15 – Слева направо: снимки, исходные маски, предсказания модели

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 количество эпох. Переобучение попробуем уменьшить использованием в слоях свертки L2 регуляризации, количество эпох увеличим до 20. Модель представлена на рис. 16.

```
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 16 – Код модели 3

График метрики `MeanDICE`, показан на рис. 17.

Рисунок 17 – График зависимости метрики `MeanDICE` от эпохи

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

Предсказания сети после первой эпохи представлены на рис. 18.

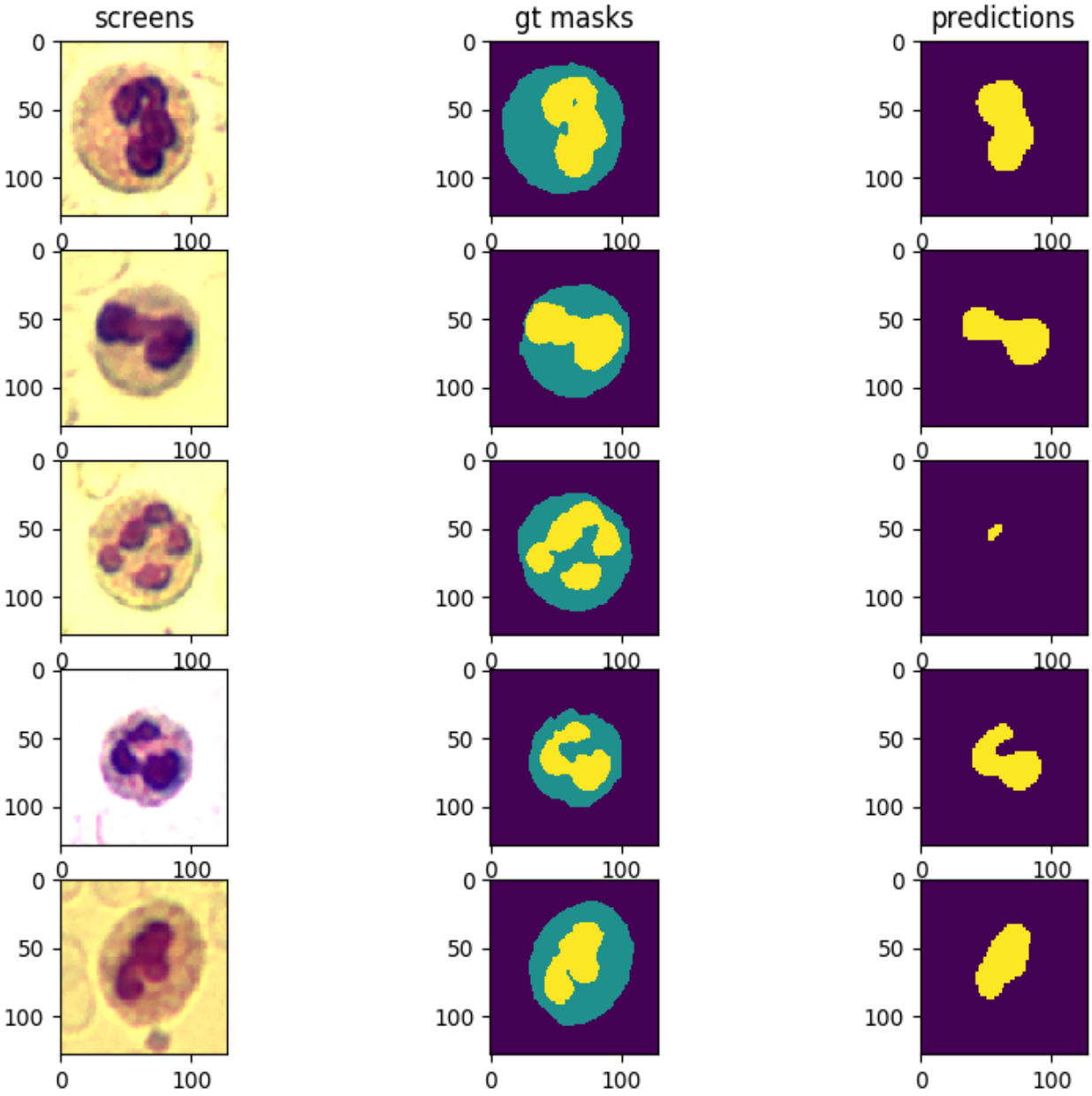

Рисунок 18 – Слева направо: снимки, исходные маски, предсказания модели

Предсказания сети после десятой эпохи (метрика – 0.91) показаны на рис. 19.

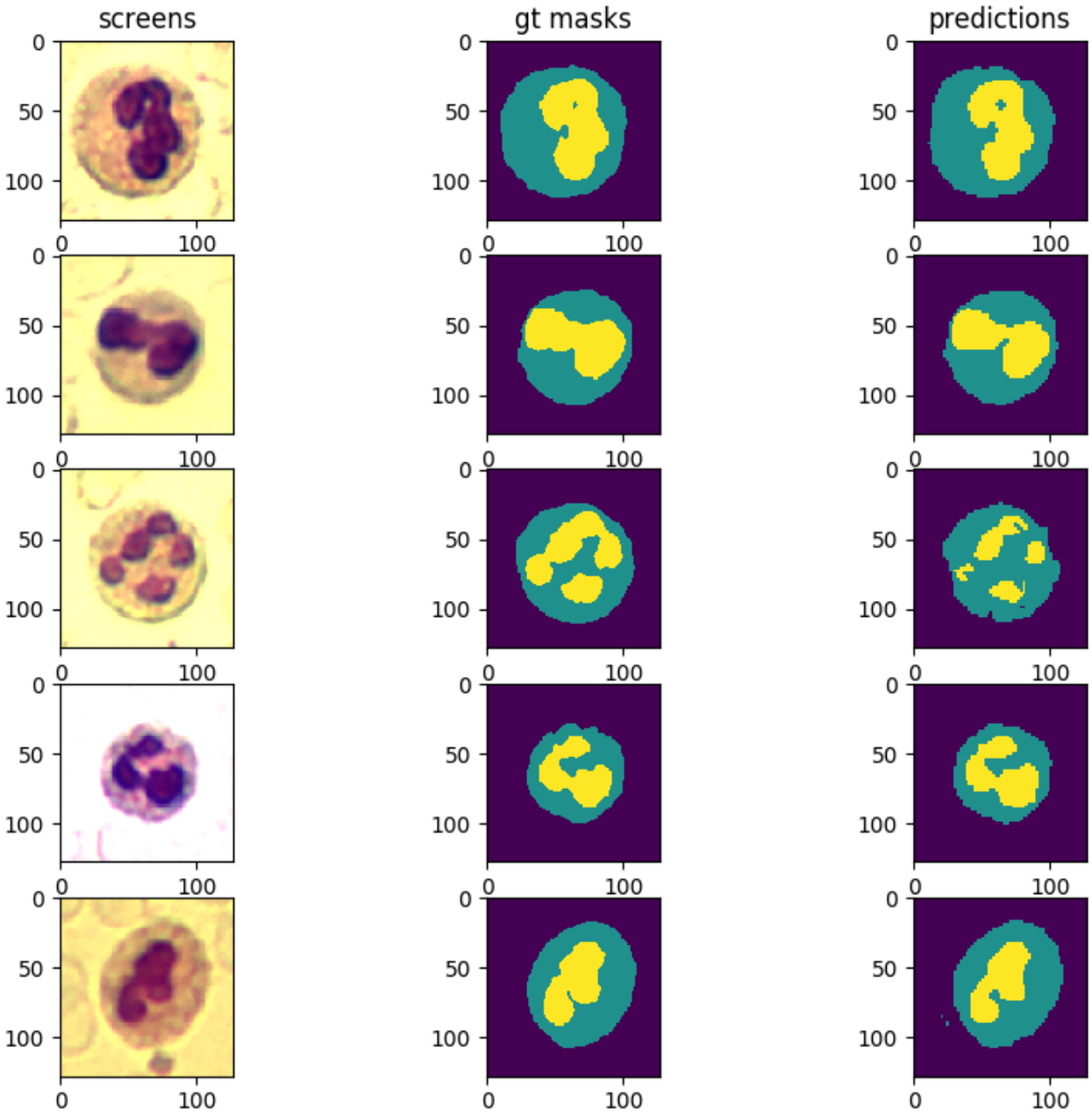

Рисунок 19 – Слева направо: снимки, исходные маски, предсказания модели

Предсказания сети после восемнадцатой эпохи (метрика – 0.942) представлены на рис. 20.

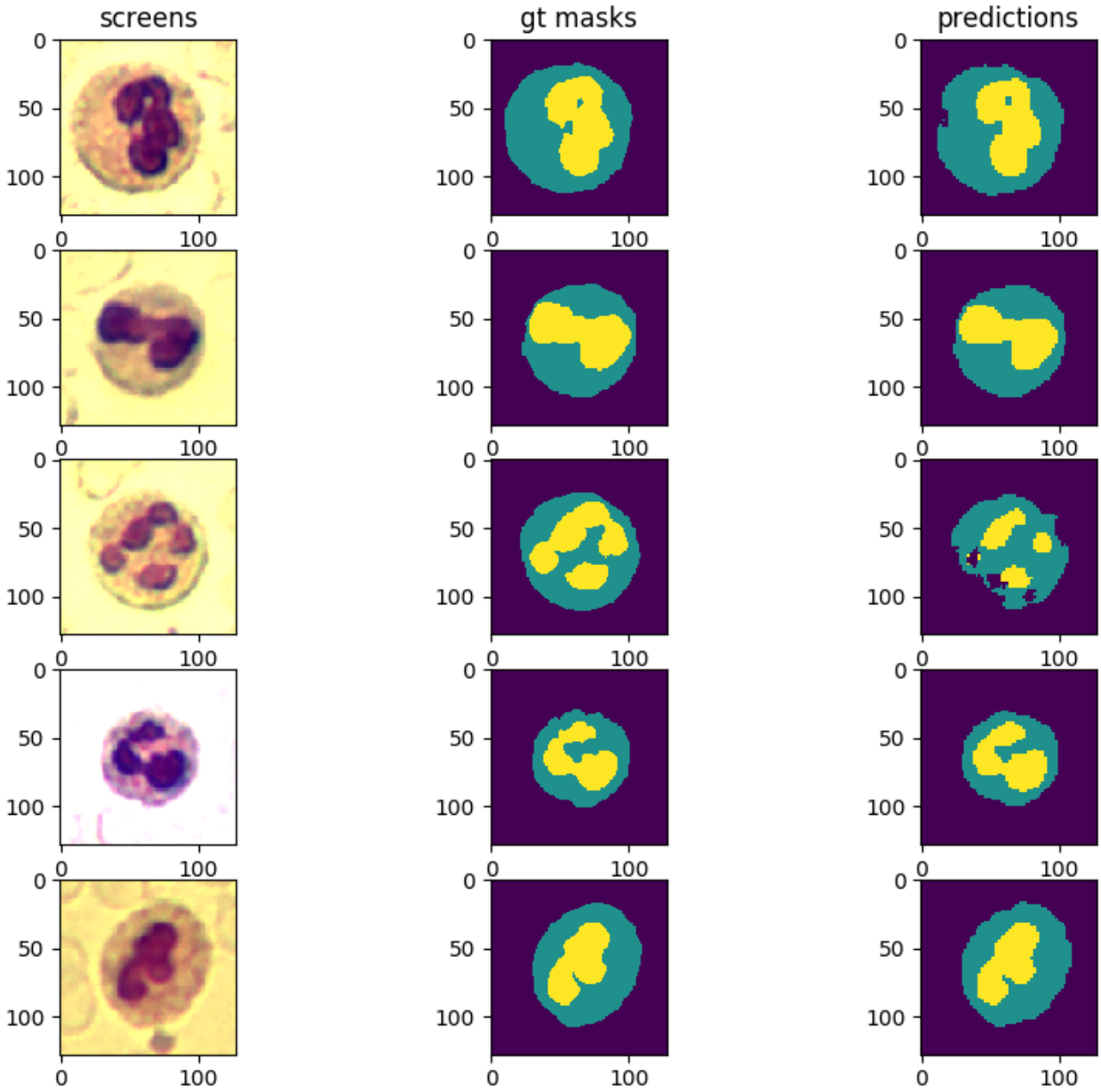

Рисунок 20 – Слева направо: снимки, исходные маски, предсказания модели

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 незначительный промах.

Решенные проблемы.

В данной работе сложнее всего было правильно подготовить исходный датасет так, чтобы сеть смогла его правильно обработать и выдать корректный результат. Проблем с достижением хорошей точности на тестовом наборе практически не было, самая первая модель уже 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 нужно для того, чтобы можно было посмотреть на результат в виде изображения. Для этого потребовалось применить к выходным маскам `argmax`, 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. Для этого потребовалось 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, описанной 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: совершенствовать модель, 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.д. 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.

Разделение ролей.

- Маркова Ангелина – подготовка датасета для обучения нейронной сети;
- Головина Екатерина – разработка модели;
- Тян Екатерина – усовершенствование разработанной модели.

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.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

```
import os
import urllib
from zipfile import ZipFile
import PIL
import numpy as np
import matplotlib.pyplot as plt

from tensorflow.keras.utils import to_categorical
from tensorflow.keras import Sequential
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.layers import Conv2D, BatchNormalization, UpSampling2D,
MaxPool2D
from tensorflow.keras.optimizers import Adam
import tensorflow.python.keras.backend as K
from tensorflow.keras.regularizers import l2

from sklearn.model_selection import train_test_split


def mean_dice(y_true, y_pred, smooth=1):
    intersection = K.sum(y_true * y_pred, axis=[1, 2, 3])
    union = K.sum(y_true, axis=[1, 2, 3]) + K.sum(y_pred, axis=[1, 2, 3])
    dice = K.mean((2. * intersection + smooth) / (union + smooth), axis=0)
    return dice


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)


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()


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)


def batch_show(A1, A2, A3):
    cols = 3
    rows = 5
    for j in range(1):
        fig = plt.figure(1, figsize=(10, 8))
        for i in range(j*rows, (j*rows)+rows):
            ax1 = fig.add_subplot(rows, cols, ((i - j * rows) * cols) + 1)
            plt.imshow(A1[i])
            ax2 = fig.add_subplot(rows, cols, ((i - j * rows) * cols) + 2)
            plt.imshow(A2[i])
            ax3 = fig.add_subplot(rows, cols, ((i - j * rows) * cols) + 3)
            plt.imshow(A3[i])
            if i == 0:
                ax1.set(title='screens')
                ax2.set(title='gt masks')
                ax3.set(title='predictions')
        plt.show()
        plt.clf()


def metric_plot(test, train, epochs):
    x = range(1, epochs+1)
    plt.figure()
    plt.title("Training and test MeanDICE")
    plt.plot(x, train, 'y', label='train')
```

```
    plt.plot(x, test, 'r', label='test')
    plt.ylabel('dice coeff')
    plt.xlabel('epochs')
    plt.legend()
    plt.show()


num_classes = 3
epochs = 20
batch_size = 2
show_interval = 2

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'),
])

# download and unzip dataset
filepath = 'wbc.zip'
download('https://github.com/zxaoyou/segmentation_WBC/archive/master.zip',
name=filepath)

# load and standartize dataset
screens, masks = load_images()
print(screens.shape)
print(masks.shape)

# 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)

# compile model
opt = Adam()
loss = CategoricalCrossentropy()
model.compile(optimizer=opt, loss=loss, metrics=[mean_dice])

his_dice_test = []
his_dice_train = []

# fit model
for i in range(1, epochs+1):
    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)
    his_dice_train.append(train_dice * batch_size/len(train_x))
    his_dice_test.append(his[1])
    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 = decodeImgs(outputs)
        # show some screens/masks/preds
        batch_show(test_x, test_y, decoded)

metric_plot(his_dice_test, his_dice_train, epochs)
```