

## РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра математического моделирования и искусственного интеллекта

### ✓ ОТЧЕТ ПО КОНТРОЛЬНОЙ РАБОТЕ № 6

Дисциплина: Методы машинного обучения

Студент: Петров Артем Евгеньевич

Группа: НКНбд-01-21

Москва 2024

#### Задание:

1. Загрузите заданный в индивидуальном задании набор данных с изображениями из Tensorflow Datasets с разбиением на обучающую, валидационную и тестовую выборки. Если при дальнейшей работе с данными возникнет нехватка вычислительных ресурсов, то разрешение изображений можно уменьшить.
2. Оставьте в наборе изображения, указанных в индивидуальном задании, и визуализируйте несколько изображений.
3. Постройте нейронные сети MLP, CNN и RNN для задачи многоклассовой классификации изображений (требования к архитектуре сетей указаны в индивидуальном задании), используя функцию потерь, указанную в индивидуальном задании. Подберите такие параметры, как функции активации, оптимизатор, начальная скорость обучения, размер мини-пакета и др. самостоятельно, обеспечивая обучение нейронных сетей. Обучайте нейронные сети с использованием валидационной выборки, сформированной в п. 1. Останавливайте обучение нейронных сетей в случае роста потерь на валидационной выборке на нескольких эпохах обучения подряд. Для каждой нейронной сети выведите количество потребовавшихся эпох обучения.
4. Оцените качество многоклассовой классификации нейронными сетями MLP, CNN и RNN на тестовой выборке при помощи показателя качества, указанного в индивидуальном задании, и выведите архитектуру нейронной сети с лучшим качеством.
5. Визуализируйте кривые обучения трех построенных моделей для показателя потерь на валидационной выборке на одном рисунке в зависимости от эпохи обучения, подписывая оси и рисунок и создавая легенду. Используйте для визуализации относительные потери (потери, деленные на начальные потери на первой эпохе).
6. Визуализируйте кривые обучения трех построенных моделей для показателя доли верных ответов на валидационной выборке на одном рисунке в зависимости от эпохи обучения, подписывая оси и рисунок и создавая легенду.
7. Используя модель нейронной сети с лучшей долей верных ответов на тестовой выборке, определите для каждого из классов два изображения в тестовой выборке, имеющие минимальную и максимальную вероятности классификации в правильный класс, и визуализируйте эти изображения.

#### Вариант 27

1. Набор данных oxford\_iiit\_pet с изменением разрешения до 60x96
2. Классы с метками 31,43,55,67
3. Требования к архитектуре сети MLP:
  - Последовательный API с методом add() при создании
  - Функция потерь: категориальная кросс-энтропия
  - Кол-во скрытых слоев 4
  - Кол-во нейронов 40 в первом скрытом слое, увеличивающееся на 20 с каждым последующим скрытым слоем
  - Использование слоев с регуляризацией L1L2
4. Требования к архитектуре сети CNN:
  - Функциональный API при создании
  - Функция потерь: разреженная категориальная кросс-энтропия

Кол-во слоев пулинга 3

Количество фильтров в сверточных слоях 16

Размеры фильтра 5x5

Использование слоев пакетной нормализации

#### 5. Требования к архитектуре сети RNN:

Последовательный API со списком слоев при создании

Функция потерь: категориальная кросс-энтропия

Слой LSTM с 96 нейронами

Использование слоев dropout

#### 6. Показатель качества многоклассовой классификации:

минимальная точность классов, где точность (precision) класса равна доле правильных предсказаний для всех точек, относимых классификатором к этому классу.

1. Загрузите заданный в индивидуальном задании набор данных с изображениями из Tensorflow Datasets с разбиением на обучающую, валидационную и тестовую выборки. Если при дальнейшей работе с данными возникнет нехватка вычислительных ресурсов, то разрешение изображений можно уменьшить.

1. Набор данных oxford\_iiit\_pet с изменением разрешения до 60x96

2. Классы с метками 31,43,55,67

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
import tensorflow_datasets as tfds
```

```
ds_train = tfds.load("oxford_iiit_pet", split = 'train')
```

Downloading and preparing dataset 773.52 MiB (download: 773.52 MiB, generated: 774.69 MiB) ...  
 DI Completed...: 100% 2/2 [01:42<00:00, 31.07s/ url]  
 DI Size...: 100% 773/773 [01:42<00:00, 16.42 MiB/s]  
 Extraction completed...: 100% 18473/18473 [01:42<00:00, 627.89 file/s]

```
ds_test = tfds.load("oxford_iiit_pet", split = 'test')
```

```
df_train = tfds.as_dataframe(ds_train)
df_train.head(2)
```

|   | file_name         | image  | label | segmentation_mask                                | species |
|---|-------------------|--|-------|--|---------|
| 0 | b'Sphynx_158.jpg' | [[[3, 3, 3],<br>[5, 3, 4], [7,<br>5, 6], [4, 2,<br>3], ... | 33    | [[[2], [2], [2], [2], [2],<br>[2], [2], [2], ... | 0       |

Next steps: [View recommended plots](#)

```
df_test = tfds.as_dataframe(ds_test)
```

```
print(type(df_train['image'][1]), df_train['image'][0].shape)
```

```
<class 'numpy.ndarray'> (500, 500, 3)
```

```
from PIL import Image
```

```
def decrease_size(img: np.ndarray) -> np.ndarray:
    need = (60, 96)
    res = Image.fromarray(img)
    ans = res.resize(need)
    return np.array(ans)
```

```
img = decrease_size(df_train['image'][4])
img.shape
```

↗ (96, 60, 3)

```
df_train['image'] = df_train['image'].apply(decrease_size)
```

```
df_test['image'] = df_test['image'].apply(decrease_size)
```

```
df_train['image'][0].shape, df_train['image'][0].shape
```

↗ ((96, 60, 3), (96, 60, 3))

```
df_train['label'].value_counts()
```

↗

| label | count |
|-------|-------|
| 33    | 100   |
| 21    | 100   |
| 29    | 100   |
| 13    | 100   |
| 30    | 100   |
| 34    | 100   |
| 16    | 100   |
| 20    | 100   |
| 19    | 100   |
| 27    | 100   |
| 31    | 100   |
| 18    | 100   |
| 26    | 100   |
| 23    | 100   |
| 24    | 100   |
| 4     | 100   |
| 15    | 100   |
| 8     | 100   |
| 28    | 100   |
| 10    | 100   |
| 9     | 100   |
| 2     | 100   |
| 36    | 100   |
| 5     | 100   |
| 6     | 100   |
| 25    | 100   |
| 35    | 100   |
| 14    | 100   |
| 3     | 100   |
| 0     | 100   |
| 1     | 100   |
| 17    | 100   |
| 32    | 99    |
| 7     | 96    |
| 22    | 96    |
| 12    | 96    |
| 11    | 93    |

Name: count, dtype: int64

```
df_train['image'][4]
```

↗ ndarray (96, 60, 3) [show data](#)



```
df_test['image'][0]
```

↗ ndarray (96, 60, 3) [show data](#)




- ✓ 2. Оставьте в наборе изображения, указанных в индивидуальном задании, и визуализируйте несколько изображений.

2. Классы с метками 31,43,55,67

```
df_train = df_train[(df_train['label'] == 0) | (df_train['label'] == 10) | (df_train['label'] == 20) | (df_train['label'] == 31)]
```

```
df_train.head()
```

|    | file_name             | image   | label | segmentation_mask                            | species |  |
|----|-----------------------|---|-------|--|---------|--|
| 13 | b'chihuahua_187.jpg'  | [[[3, 7, 6], [14, 20, 18], [10, 21, 17], [4, 1... | 1     | [[[3], [3], [3], [3], [3], [3], [3], [3],... | 1       |  |
| 21 | b'Abyssinian_196.jpg' | [[[116, 122, 120], [117, 123, 121], [119, 125,... | 0     | [[[2], [2], [2], [2], [2], [2], [2], [2],... | 0       |  |
| 46 | b'Maine_Coon_10.jpg'  | [[[3, 52, 51], [2, 47, 45], [2, 44, 43], [1...    | 2     | [[[2], [2], [2], [2], [2], [2], [2], [2],... | 0       |  |

Next steps: [View recommended plots](#)

```
df_train['label'].value_counts()
```

```
label
10    100
0      100
20    100
31    100
Name: count, dtype: int64
```

```
df_test = df_test[(df_test['label'] == 0) | (df_test['label'] == 10) | (df_test['label'] == 20) | (df_test['label'] == 31)]
```

```
df_test['label'].value_counts()
```

```
label
10    100
0      100
20    100
31    100
Name: count, dtype: int64
```

3. Постройте нейронные сети MLP, CNN и RNN для задачи многоклассовой классификации изображений (требования к архитектуре сетей указаны в индивидуальном задании), используя функцию потерь, указанную в индивидуальном задании. Подберите такие параметры, как функции активации, оптимизатор, начальная скорость обучения, размер мини-пакета и др. самостоятельно, обеспечивая обучение нейронных сетей. Обучайте нейронные сети с использованием валидационной выборки, сформированной в п. 1. Останавливайте обучение нейронных сетей в случае роста потерь на валидационной выборке на нескольких эпохах обучения подряд. Для каждой нейронной сети выведите количество потребовавшихся эпох обучения.


```
normal_features = {0:0, 10:1, 20:2, 31:3}
```

```
df_train['label'] = df_train['label'].apply(lambda x: normal_features[x])
```

```
<ipython-input-21-c0304d02a927>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

See the caveats in the documentation: [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/indexing.html#returning-a-view-versus](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus)  
`df_train['label'] = df_train['label'].apply(lambda x: normal_features[x])`

```
df_test['label'] = df_test['label'].apply(lambda x: normal_features[x])
```

 <ipython-input-22-c3bb781bed57>:1: SettingWithCopyWarning:  
 A value is trying to be set on a copy of a slice from a DataFrame.  
 Try using `.loc[row_indexer,col_indexer] = value` instead


See the caveats in the documentation: [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/indexing.html#returning-a-view-versus](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus)  
`df_test['label'] = df_test['label'].apply(lambda x: normal_features[x])`

```
# tmp = df_train['image'].apply(lambda x: x / 255)
```


```
# tmp.head()
```

```
def getX(arr: np.ndarray) -> np.ndarray:
    x = np.empty((400, 96, 60, 3))
    for i in range(arr.shape[0]):
        x[i] = arr[i] / 255
    return x
```


```
train_x = getX(df_train['image'].values)
train_x.shape
```

 (400, 96, 60, 3)

```
test_x = getX(df_test['image'].values)
test_x.shape
```

 (400, 96, 60, 3)

```
train_x[0]
```

 array([[0.01176471, 0.02745098, 0.02352941],  
 [0.05490196, 0.07843137, 0.07058824],  
 [0.03921569, 0.08235294, 0.06666667],  
 ...,  
 [0.09411765, 0.10588235, 0.12156863],  
 [0.09411765, 0.09019608, 0.09411765],  
 [0.09803922, 0.08627451, 0.09019608]],  
  
 [[0.01568627, 0.03529412, 0.03137255],  
 [0.05098039, 0.07843137, 0.07058824],  
 [0.03921569, 0.0745098 , 0.0627451 ],  
 ...,  
 [0.09411765, 0.10588235, 0.12156863],  
 [0.09411765, 0.09019608, 0.09803922],  
 [0.09803922, 0.08235294, 0.09411765]],  
  
 [[0.01960784, 0.03921569, 0.03529412],  
 [0.05490196, 0.07843137, 0.07058824],  
 [0.05882353, 0.08235294, 0.0745098 ],  
 ...,  
 [0.10196078, 0.10980392, 0.13333333],  
 [0.10196078, 0.09411765, 0.11372549],  
 [0.10588235, 0.09019608, 0.10980392]],  
  
 ...,  
  
 [[0.4745098 , 0.44313725, 0.4745098 ],  
 [0.45490196, 0.42352941, 0.45490196],  
 [0.41568627, 0.39607843, 0.43921569],  
 ...,  
 [0.17254902, 0.11764706, 0.07843137],  
 [0.29019608, 0.22352941, 0.16078431],  
 [0.38431373, 0.29411765, 0.21568627]],  
  
 [[0.43921569, 0.43529412, 0.45882353],  
 [0.42745098, 0.41568627, 0.44313725],  
 [0.40392157, 0.41568627, 0.44705882],  
 ...,  
 [0.18039216, 0.1254902 , 0.09019608],  
 [0.2 , 0.14509804, 0.09411765],  
 [0.35686275, 0.2745098 , 0.2 ]],  
  
 [[0.43137255, 0.42745098, 0.45098039],  
 [0.42745098, 0.41960784, 0.44705882],  
 [0.41960784, 0.43137255, 0.46666667],

```

...,
[0.20392157, 0.14509804, 0.10980392],
[0.12941176, 0.08235294, 0.03921569],
[0.31764706, 0.23921569, 0.16470588]])

# train_x = np.empty((400, 96, 60, 3))
# for i in range(tmp.shape[0]):
#     train_x[i] = tmp.values[i]

# train_x[130].shape
# df_train['image'][21]
# train_x[0]

def to_one_hot(labels, dimension=3):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

train_y = to_one_hot(df_train['label'].values, 4)

test_y = to_one_hot(df_test['label'].values, 4)

type(train_y), type(train_x), type(test_y), type(test_x)

(numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray)

```

## ✓ MLP:

### 3. Требования к архитектуре сети MLP:

Последовательный API с методом add() при создании

Функция потерь: категориальная кросс-энтропия

Кол-во скрытых слоев 4

Кол-во нейронов 40 в первом скрытом слое, увеличивающееся на 20 с каждым последующим скрытым слоем

Использование слоев с регуляризацией L1L2

### 4. Показатель качества многоклассовой классификации:

минимальная точность классов, где точность (precision) класса равна доле правильных предсказаний для всех точек, относимых классификатором к этому классу.

```

mlp = tf.keras.Sequential()
mlp.add(tf.keras.layers.Input(shape = (96, 60, 3)))
mlp.add(tf.keras.layers.Flatten())
mlp.add(tf.keras.layers.Dense(40, activation = 'relu', kernel_regularizer = tf.keras.regularizers.L1L2(0.001)))
mlp.add(tf.keras.layers.Dense(60, activation = 'relu', kernel_regularizer = tf.keras.regularizers.L1L2(0.001)))
mlp.add(tf.keras.layers.Dense(80, activation = 'relu', kernel_regularizer = tf.keras.regularizers.L1L2(0.001)))
mlp.add(tf.keras.layers.Dense(100, activation = 'relu', kernel_regularizer = tf.keras.regularizers.L1L2(0.001)))
mlp.add(tf.keras.layers.Dense(4, activation = 'softmax'))

mlp.compile(
    loss = tf.keras.losses.CategoricalCrossentropy(label_smoothing = 0.01),
    optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001),
    metrics = [tf.keras.metrics.CategoricalAccuracy(name = "accuracy")]
)

mlp_hist = mlp.fit(
    x = train_x,
    y = train_y,
    verbose = 1,
    epochs = 300,
    batch_size = 64,
    validation_data = (test_x, test_y)
)

```

```

Epoch 136/300
7/7 [=====] - 0s 32ms/step - loss: 1.4067 - accuracy: 0.6150 - val_loss: 1.3606 - val_accuracy: 0.6625
Epoch 137/300
7/7 [=====] - 0s 25ms/step - loss: 1.2991 - accuracy: 0.7275 - val_loss: 1.2920 - val_accuracy: 0.6975
Epoch 138/300
7/7 [=====] - 0s 26ms/step - loss: 1.3556 - accuracy: 0.6575 - val_loss: 1.2588 - val_accuracy: 0.7125
Epoch 139/300
7/7 [=====] - 0s 26ms/step - loss: 1.3499 - accuracy: 0.6275 - val_loss: 1.3008 - val_accuracy: 0.6975
Epoch 140/300
7/7 [=====] - 0s 26ms/step - loss: 1.3216 - accuracy: 0.6775 - val_loss: 1.2773 - val_accuracy: 0.6975
Epoch 141/300
7/7 [=====] - 0s 28ms/step - loss: 1.4848 - accuracy: 0.5500 - val_loss: 2.5063 - val_accuracy: 0.2950
Epoch 142/300
7/7 [=====] - 0s 29ms/step - loss: 1.9607 - accuracy: 0.3550 - val_loss: 1.5939 - val_accuracy: 0.5825
Epoch 143/300
7/7 [=====] - 0s 27ms/step - loss: 1.6304 - accuracy: 0.5975 - val_loss: 1.5944 - val_accuracy: 0.6300
Epoch 144/300
7/7 [=====] - 0s 31ms/step - loss: 1.5066 - accuracy: 0.6750 - val_loss: 1.4354 - val_accuracy: 0.6825
Epoch 145/300
7/7 [=====] - 0s 32ms/step - loss: 1.6181 - accuracy: 0.5900 - val_loss: 1.6087 - val_accuracy: 0.5675
Epoch 146/300
7/7 [=====] - 0s 29ms/step - loss: 1.6293 - accuracy: 0.5750 - val_loss: 1.5056 - val_accuracy: 0.7550
Epoch 147/300
7/7 [=====] - 0s 29ms/step - loss: 1.6211 - accuracy: 0.6525 - val_loss: 1.7824 - val_accuracy: 0.5450
Epoch 148/300
7/7 [=====] - 0s 32ms/step - loss: 1.7520 - accuracy: 0.5850 - val_loss: 1.5490 - val_accuracy: 0.7350
Epoch 149/300
7/7 [=====] - 0s 31ms/step - loss: 1.5423 - accuracy: 0.6725 - val_loss: 1.4076 - val_accuracy: 0.6850
Epoch 150/300
7/7 [=====] - 0s 26ms/step - loss: 1.3726 - accuracy: 0.7000 - val_loss: 1.3203 - val_accuracy: 0.7450
Epoch 151/300
7/7 [=====] - 0s 31ms/step - loss: 1.3091 - accuracy: 0.7250 - val_loss: 1.2616 - val_accuracy: 0.7475
Epoch 152/300
7/7 [=====] - 0s 30ms/step - loss: 1.3756 - accuracy: 0.6500 - val_loss: 2.0670 - val_accuracy: 0.3375
Epoch 153/300
7/7 [=====] - 0s 26ms/step - loss: 1.7203 - accuracy: 0.5100 - val_loss: 1.6507 - val_accuracy: 0.4775
Epoch 154/300
7/7 [=====] - 0s 27ms/step - loss: 1.6671 - accuracy: 0.5325 - val_loss: 1.4764 - val_accuracy: 0.6575
Epoch 155/300
7/7 [=====] - 0s 37ms/step - loss: 1.4462 - accuracy: 0.6950 - val_loss: 1.4285 - val_accuracy: 0.6750
Epoch 156/300
7/7 [=====] - 0s 40ms/step - loss: 1.4054 - accuracy: 0.7025 - val_loss: 1.3229 - val_accuracy: 0.7650
Epoch 157/300
7/7 [=====] - 0s 39ms/step - loss: 1.3884 - accuracy: 0.6850 - val_loss: 1.4265 - val_accuracy: 0.5850
Epoch 158/300
7/7 [=====] - 0s 39ms/step - loss: 1.4983 - accuracy: 0.5775 - val_loss: 1.4364 - val_accuracy: 0.6025
Epoch 159/300
7/7 [=====] - 0s 37ms/step - loss: 1.4644 - accuracy: 0.5825 - val_loss: 1.3781 - val_accuracy: 0.6250
Epoch 160/300
7/7 [=====] - 0s 36ms/step - loss: 1.6121 - accuracy: 0.4675 - val_loss: 1.6460 - val_accuracy: 0.5200
Epoch 161/300
7/7 [=====] - 0s 39ms/step - loss: 1.6345 - accuracy: 0.5300 - val_loss: 1.5735 - val_accuracy: 0.5425

```

Как мы видим, нейросеть обучилась, и теперь она определяет принадлежность к фотографии к классу(всего их 4) с вероятностью 86 для валидационной выборки на последней эпохе(хотя на предыдущих было больше)

## ✓ CNN

```

train_y = df_train['label'].values
# test_y =

train_y[:10]
array([1, 0, 2, 3, 0, 3, 1, 2, 0, 0])

test_y = df_test['label'].values

test_y[:10]
array([1, 0, 2, 3, 0, 3, 1, 2, 0, 0])

```

Требования к архитектуре сети CNN:

- Функциональный API при создании
- Функция потерь: разреженная категориальная кросс-энтропия
- Кол-во слоев пулинга 3

- Количество фильтров в сверточных слоях 16
- Размеры фильтра 5x5
- Использование слоев пакетной нормализации

```
inputs = tf.keras.layers.Input(shape = (96, 60, 3))

x = tf.keras.layers.Conv2D(filters = 64, kernel_size = (5, 5), activation = 'relu')(inputs)

x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same')(x)

x = tf.keras.layers.Conv2D(filters = 16, kernel_size = (5, 5), activation = 'relu')(x)

# x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same')(x)

x = tf.keras.layers.Conv2D(filters = 16, kernel_size = (5, 5), activation = 'relu')(x)

x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same')(x)

x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dropout(rate = 0.25)(x)
x = tf.keras.layers.Dense(256, activation = 'relu')(x)
outputs = tf.keras.layers.Dense(4, activation = 'softmax')(x)

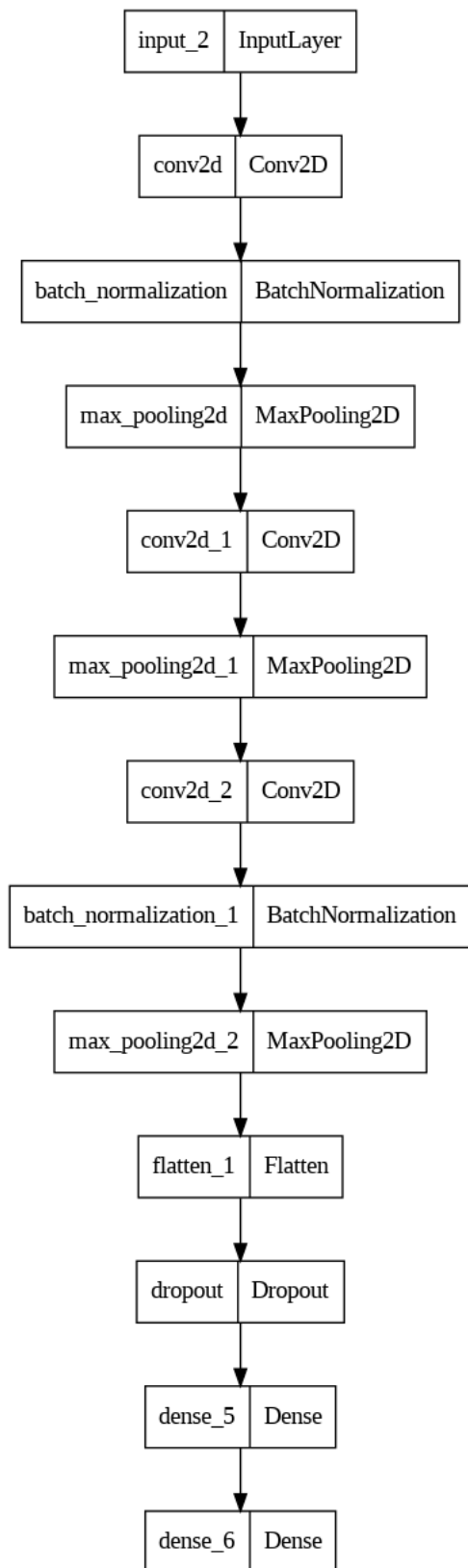
cnn = tf.keras.Model(inputs = inputs, outputs = outputs)
cnn.summary()
```

↗ Model: "model"

| Layer (type)                                | Output Shape        | Param # |
|---|---------------------|---------|
| =====                                       |                     |         |
| input_2 (InputLayer)                        | [(None, 96, 60, 3)] | 0       |
| conv2d (Conv2D)                             | (None, 92, 56, 64)  | 4864    |
| batch_normalization (Batch Normalization)   | (None, 92, 56, 64)  | 256     |
| max_pooling2d (MaxPooling2D)                | (None, 46, 28, 64)  | 0       |
| conv2d_1 (Conv2D)                           | (None, 42, 24, 32)  | 51232   |
| max_pooling2d_1 (MaxPooling2D)              | (None, 21, 12, 32)  | 0       |
| conv2d_2 (Conv2D)                           | (None, 17, 8, 16)   | 12816   |
| batch_normalization_1 (Batch Normalization) | (None, 17, 8, 16)   | 64      |
| max_pooling2d_2 (MaxPooling2D)              | (None, 9, 4, 16)    | 0       |
| flatten_1 (Flatten)                         | (None, 576)         | 0       |
| dropout (Dropout)                           | (None, 576)         | 0       |
| dense_5 (Dense)                             | (None, 256)         | 147712  |
| dense_6 (Dense)                             | (None, 4)           | 1028    |
| =====                                       |                     |         |
| Total params: 217972 (851.45 KB)            |                     |         |
| Trainable params: 217812 (850.83 KB)        |                     |         |
| Non-trainable params: 160 (640.00 Byte)     |                     |         |

```
tf.keras.utils.plot_model(cnn)
```





```
cnn.compile(  
    loss = tf.keras.losses.sparse_categorical_crossentropy,  
    optimizer = tf.keras.optimizers.Adam(),  
    metrics = [tf.keras.metrics.SparseCategoricalAccuracy()]  
)
```

```
cnn_hist = cnn.fit(  
    x = train_x,  
    y = train_y,  
    epochs = 20,  
    verbose = 1,  
    batch_size = 128,  
    validation_data = (test_x, test_y)  
)
```

```

Epoch 1/20
4/4 [=====] - 13s 3s/step - loss: 1.8118 - sparse_categorical_accuracy: 0.3175 - val_loss: 1.3917 - val_sparse_categorical_accuracy: 0.3175
Epoch 2/20
4/4 [=====] - 9s 2s/step - loss: 1.3815 - sparse_categorical_accuracy: 0.4550 - val_loss: 1.3403 - val_sparse_categorical_accuracy: 0.4550
Epoch 3/20
4/4 [=====] - 11s 2s/step - loss: 1.2798 - sparse_categorical_accuracy: 0.4650 - val_loss: 1.3968 - val_sparse_categorical_accuracy: 0.4650
Epoch 4/20
4/4 [=====] - 12s 3s/step - loss: 1.1278 - sparse_categorical_accuracy: 0.5450 - val_loss: 1.3484 - val_sparse_categorical_accuracy: 0.5450
Epoch 5/20
4/4 [=====] - 11s 3s/step - loss: 1.0189 - sparse_categorical_accuracy: 0.5825 - val_loss: 1.3068 - val_sparse_categorical_accuracy: 0.5825
Epoch 6/20
4/4 [=====] - 10s 3s/step - loss: 0.8536 - sparse_categorical_accuracy: 0.6575 - val_loss: 1.3172 - val_sparse_categorical_accuracy: 0.6575
Epoch 7/20
4/4 [=====] - 11s 3s/step - loss: 0.7883 - sparse_categorical_accuracy: 0.6750 - val_loss: 1.2981 - val_sparse_categorical_accuracy: 0.6750
Epoch 8/20
4/4 [=====] - 11s 3s/step - loss: 0.7274 - sparse_categorical_accuracy: 0.7150 - val_loss: 1.2999 - val_sparse_categorical_accuracy: 0.7150
Epoch 9/20
4/4 [=====] - 12s 3s/step - loss: 0.6389 - sparse_categorical_accuracy: 0.7425 - val_loss: 1.3535 - val_sparse_categorical_accuracy: 0.7425
Epoch 10/20
4/4 [=====] - 13s 3s/step - loss: 0.6144 - sparse_categorical_accuracy: 0.7450 - val_loss: 1.4429 - val_sparse_categorical_accuracy: 0.7450
Epoch 11/20
4/4 [=====] - 10s 2s/step - loss: 0.5334 - sparse_categorical_accuracy: 0.8075 - val_loss: 1.4644 - val_sparse_categorical_accuracy: 0.8075
Epoch 12/20
4/4 [=====] - 10s 2s/step - loss: 0.4604 - sparse_categorical_accuracy: 0.8350 - val_loss: 1.4701 - val_sparse_categorical_accuracy: 0.8350
Epoch 13/20
4/4 [=====] - 12s 3s/step - loss: 0.4463 - sparse_categorical_accuracy: 0.8300 - val_loss: 1.5201 - val_sparse_categorical_accuracy: 0.8300
Epoch 14/20
4/4 [=====] - 11s 3s/step - loss: 0.3655 - sparse_categorical_accuracy: 0.8950 - val_loss: 1.6344 - val_sparse_categorical_accuracy: 0.8950
Epoch 15/20
4/4 [=====] - 13s 3s/step - loss: 0.3239 - sparse_categorical_accuracy: 0.9025 - val_loss: 1.8099 - val_sparse_categorical_accuracy: 0.9025
Epoch 16/20
4/4 [=====] - 13s 3s/step - loss: 0.2630 - sparse_categorical_accuracy: 0.9325 - val_loss: 1.9591 - val_sparse_categorical_accuracy: 0.9325
Epoch 17/20
4/4 [=====] - 10s 2s/step - loss: 0.2347 - sparse_categorical_accuracy: 0.9450 - val_loss: 2.0449 - val_sparse_categorical_accuracy: 0.9450
Epoch 18/20
4/4 [=====] - 11s 3s/step - loss: 0.2566 - sparse_categorical_accuracy: 0.9250 - val_loss: 2.2366 - val_sparse_categorical_accuracy: 0.9250
Epoch 19/20
4/4 [=====] - 11s 3s/step - loss: 0.2060 - sparse_categorical_accuracy: 0.9525 - val_loss: 2.3112 - val_sparse_categorical_accuracy: 0.9525
Epoch 20/20
4/4 [=====] - 11s 3s/step - loss: 0.2078 - sparse_categorical_accuracy: 0.9275 - val_loss: 2.2512 - val_sparse_categorical_accuracy: 0.9275

```

К сожалению, CNN переобучилась, из-за чего точность составляет 25%, что не лучше случайного выбора принадлежности фотографии к одному из четырёх имеющихся классов

## ✓ RNN

Требования к архитектуре сети RNN:

- Последовательный API со списком слоев при создании
- Функция потерь: категориальная кросс-энтропия
- Слой LSTM с 96 нейронами
- Использование слоев dropout

```
train_y = to_one_hot(df_train['label'].values, 4)
```

```
test_y = to_one_hot(df_test['label'].values, 4)
```

```
from PIL import ImageOps
```

```
def grayscale_img(x: np.ndarray) -> np.ndarray:
    img = ImageOps.grayscale(Image.fromarray(x).resize((96, 96)))
    res = np.array(img)
    return res
```

```
def getX(arr: np.ndarray) -> np.ndarray:
    x = np.empty((400, 96, 96))
    for i in range(arr.shape[0]):
        x[i] = arr[i] / 255
    return x
```

```
train_x = getX(df_train['image'].apply(grayscale_img).values)
```

```
train_x.shape
```

```
(400, 96, 96)
```

```
test_x = getX(df_test['image']).apply( grayscale_img ).values)
```

```
rnn = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(96, input_shape = (None, 96)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(4, activation = 'softmax')
])
```

```
rnn.compile(
    loss = tf.keras.losses.CategoricalCrossentropy(),
    optimizer = tf.keras.optimizers.Adam(),
    metrics = [tf.keras.metrics.CategoricalAccuracy()]
)
```

```
rnn_hist = rnn.fit(
    train_x, train_y, epochs = 50, batch_size = 64, validation_data = (test_x, test_y)
)
```

```
7/7 [=====] - 1s 156ms/step - loss: 0.9881 - categorical_accuracy: 0.5750 - val_loss: 1.2840 - val_categorical_accuracy: 0.5750
Epoch 23/50
7/7 [=====] - 1s 160ms/step - loss: 0.9808 - categorical_accuracy: 0.5925 - val_loss: 1.2120 - val_categorical_accuracy: 0.5925
Epoch 24/50
7/7 [=====] - 1s 156ms/step - loss: 0.9981 - categorical_accuracy: 0.5675 - val_loss: 1.1958 - val_categorical_accuracy: 0.5675
Epoch 25/50
7/7 [=====] - 2s 248ms/step - loss: 0.9354 - categorical_accuracy: 0.6050 - val_loss: 1.1625 - val_categorical_accuracy: 0.6050
Epoch 26/50
7/7 [=====] - 2s 294ms/step - loss: 0.9083 - categorical_accuracy: 0.6125 - val_loss: 1.1595 - val_categorical_accuracy: 0.6125
Epoch 27/50
7/7 [=====] - 1s 165ms/step - loss: 0.8932 - categorical_accuracy: 0.6100 - val_loss: 1.1422 - val_categorical_accuracy: 0.6100
Epoch 28/50
7/7 [=====] - 1s 152ms/step - loss: 0.9174 - categorical_accuracy: 0.6200 - val_loss: 1.1653 - val_categorical_accuracy: 0.6200
Epoch 29/50
7/7 [=====] - 1s 157ms/step - loss: 0.8908 - categorical_accuracy: 0.6100 - val_loss: 1.1483 - val_categorical_accuracy: 0.6100
Epoch 30/50
7/7 [=====] - 1s 158ms/step - loss: 0.8973 - categorical_accuracy: 0.6250 - val_loss: 1.1113 - val_categorical_accuracy: 0.6250
Epoch 31/50
7/7 [=====] - 1s 156ms/step - loss: 0.8372 - categorical_accuracy: 0.6575 - val_loss: 1.1045 - val_categorical_accuracy: 0.6575
Epoch 32/50
7/7 [=====] - 1s 156ms/step - loss: 0.8341 - categorical_accuracy: 0.6425 - val_loss: 1.0863 - val_categorical_accuracy: 0.6425
Epoch 33/50
7/7 [=====] - 1s 158ms/step - loss: 0.7970 - categorical_accuracy: 0.7025 - val_loss: 1.0311 - val_categorical_accuracy: 0.7025
Epoch 34/50
7/7 [=====] - 1s 162ms/step - loss: 0.7310 - categorical_accuracy: 0.7125 - val_loss: 1.0386 - val_categorical_accuracy: 0.7125
Epoch 35/50
7/7 [=====] - 1s 158ms/step - loss: 0.7457 - categorical_accuracy: 0.7300 - val_loss: 0.9995 - val_categorical_accuracy: 0.7300
Epoch 36/50
7/7 [=====] - 1s 206ms/step - loss: 0.7378 - categorical_accuracy: 0.7000 - val_loss: 0.9909 - val_categorical_accuracy: 0.7000
Epoch 37/50
7/7 [=====] - 2s 299ms/step - loss: 0.7165 - categorical_accuracy: 0.7125 - val_loss: 0.9297 - val_categorical_accuracy: 0.7125
Epoch 38/50
7/7 [=====] - 1s 195ms/step - loss: 0.6667 - categorical_accuracy: 0.7475 - val_loss: 0.9954 - val_categorical_accuracy: 0.7475
Epoch 39/50
7/7 [=====] - 1s 155ms/step - loss: 0.6430 - categorical_accuracy: 0.7325 - val_loss: 0.8518 - val_categorical_accuracy: 0.7325
Epoch 40/50
7/7 [=====] - 1s 156ms/step - loss: 0.6140 - categorical_accuracy: 0.7500 - val_loss: 0.7731 - val_categorical_accuracy: 0.7500
Epoch 41/50
7/7 [=====] - 1s 155ms/step - loss: 0.6645 - categorical_accuracy: 0.7300 - val_loss: 0.8894 - val_categorical_accuracy: 0.7300
Epoch 42/50
7/7 [=====] - 1s 156ms/step - loss: 0.6880 - categorical_accuracy: 0.7400 - val_loss: 0.8123 - val_categorical_accuracy: 0.7400
Epoch 43/50
7/7 [=====] - 1s 157ms/step - loss: 0.6446 - categorical_accuracy: 0.7325 - val_loss: 0.8155 - val_categorical_accuracy: 0.7325
Epoch 44/50
7/7 [=====] - 1s 156ms/step - loss: 0.6273 - categorical_accuracy: 0.7575 - val_loss: 0.8334 - val_categorical_accuracy: 0.7575
Epoch 45/50
7/7 [=====] - 1s 156ms/step - loss: 0.6207 - categorical_accuracy: 0.7450 - val_loss: 0.7617 - val_categorical_accuracy: 0.7450
Epoch 46/50
7/7 [=====] - 1s 155ms/step - loss: 0.6172 - categorical_accuracy: 0.7725 - val_loss: 0.7785 - val_categorical_accuracy: 0.7725
Epoch 47/50
7/7 [=====] - 1s 181ms/step - loss: 0.5423 - categorical_accuracy: 0.8000 - val_loss: 0.6439 - val_categorical_accuracy: 0.8000
Epoch 48/50
7/7 [=====] - 2s 361ms/step - loss: 0.4581 - categorical_accuracy: 0.8400 - val_loss: 0.6627 - val_categorical_accuracy: 0.8400
Epoch 49/50
7/7 [=====] - 1s 177ms/step - loss: 0.4321 - categorical_accuracy: 0.8575 - val_loss: 0.5947 - val_categorical_accuracy: 0.8575
Epoch 50/50
7/7 [=====] - 1s 164ms/step - loss: 0.4230 - categorical_accuracy: 0.8425 - val_loss: 0.5345 - val_categorical_accuracy: 0.8425
```

RNN отлично справилась с задачей. Нейросеть однозначно обучилась, так как потери на каждой эпохе уменьшались, а точность для двух выборок(тестовой и валидационной) повысилась до 80-86%

- 
4. Оцените качество многоклассовой классификации нейронными сетями MLP, CNN и RNN на тестовой выборке при помощи показателя качества, указанного в индивидуальном задании, и выведите архитектуру нейронной сети с лучшим качеством.

Из трех нейросетей самый большой показатель качества на тестовой выборке – MLP, который обгоняет RNN всего на 2%. На третьем же месте стоит CNN, который не обучился вовсе. Стоит отметить, что RNN и MLP делят первое место, так как на каждом из новых запусков меняется победитель.

- 
5. Визуализируйте кривые обучения трех построенных моделей для показателя потерь на валидационной выборке на одном рисунке в зависимости от эпохи обучения, подписывая оси и рисунок и создавая легенду. Используйте для визуализации относительные потери (потери, деленные на начальные потери на первой эпохе).

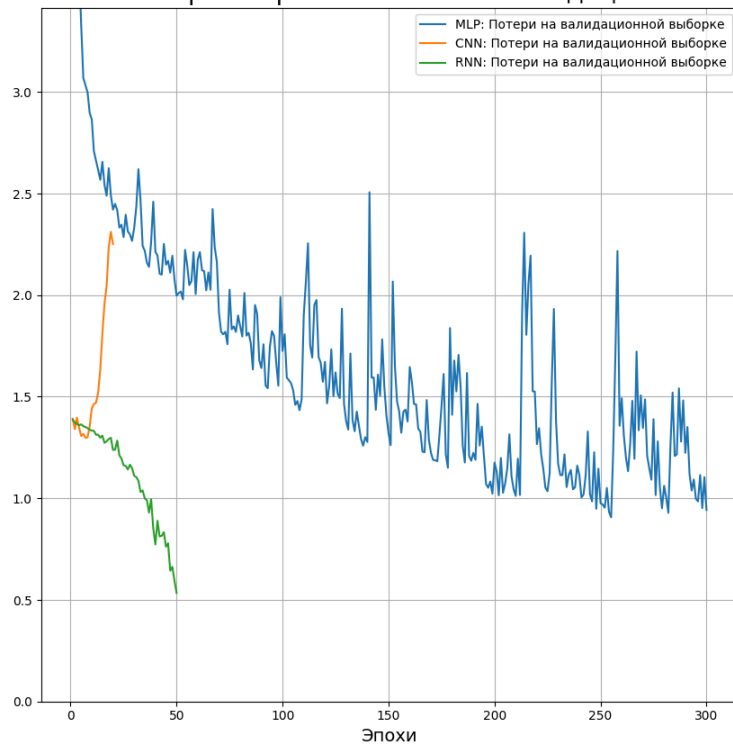
```
def plot_loss(hist: tf.keras.callbacks.History, epochs:int, lim = False):
    plt.plot(np.arange(1, epochs+1), hist.history['loss'], label='Потери на тренировочной выборке')
    plt.plot(np.arange(1, epochs+1), hist.history['val_loss'], label='Потери на валидационной выборке')
    plt.title('Показатели потери нейронной сети', size=20)
    plt.xlabel('Эпохи', size=14)
    if lim:
        plt.ylim([0, max(hist.history['loss'])*0.5])
    plt.grid(True)
    plt.legend();

def plot_metrics(hist: tf.keras.callbacks.History, metrics_name: str, epochs: int):
    plt.plot(np.arange(1, epochs+1), hist.history[metrics_name], label=metrics_name + ' на тренировочной')
    plt.plot(np.arange(1, epochs+1), hist.history['val_'+metrics_name], label=metrics_name + ' на валидационной выборке')
    plt.title(f'Показатели метрики {metrics_name} нейронной сети', size=20)
    plt.xlabel('Эпохи', size=14)
    plt.grid(True)
    plt.legend();

plt.figure(figsize = (10, 10))
plt.plot(np.arange(1, 301), mlp_hist.history['val_loss'], label='MLP: Потери на валидационной выборке')
plt.plot(np.arange(1, 21), cnn_hist.history['val_loss'], label='CNN: Потери на валидационной выборке')
plt.plot(np.arange(1, 51), rnn_hist.history['val_loss'], label='RNN: Потери на валидационной выборке')
plt.ylim([0, max(mlp_hist.history['val_loss']) * 0.5])
plt.title('Показатели потери нейронных сетей на валидационной выборке', size=20)
plt.xlabel('Эпохи', size=14)
plt.grid(True)
plt.legend();
```



## Показатели потери нейронных сетей на валидационной выборке

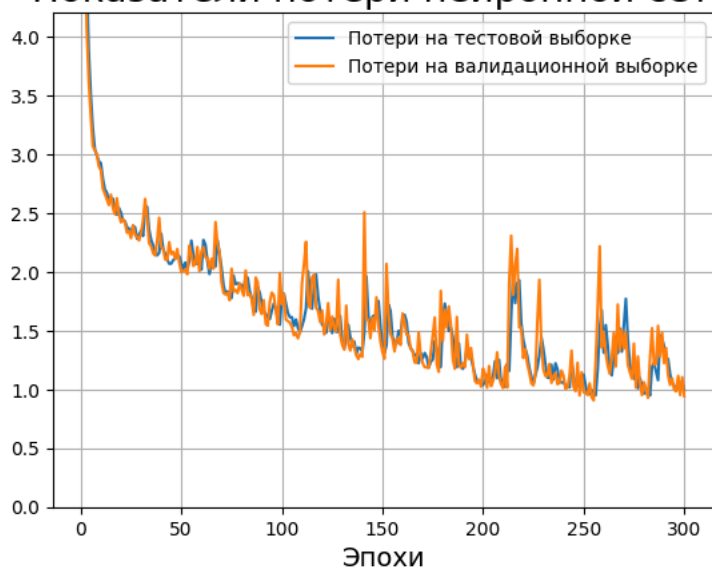


### MLP

```
plot_loss(mlp_hist, epochs = 300, lim = True)
```



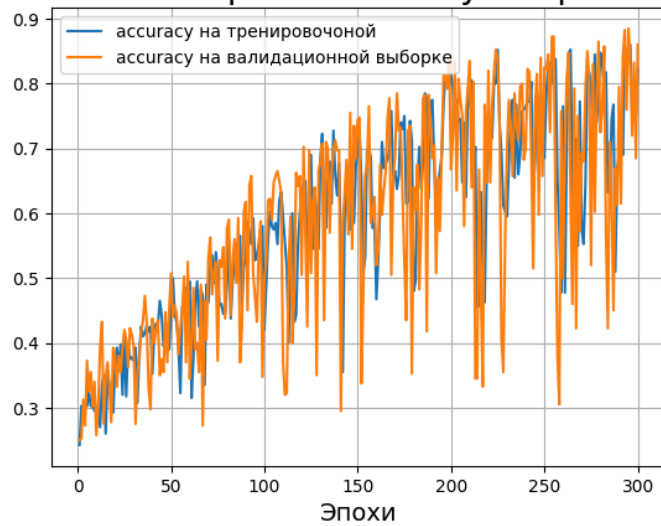
## Показатели потери нейронной сети



```
plot_metrics(mlp_hist, epochs = 300, metrics_name = 'accuracy')
```



## Показатели метрики ассигасу нейронной сети

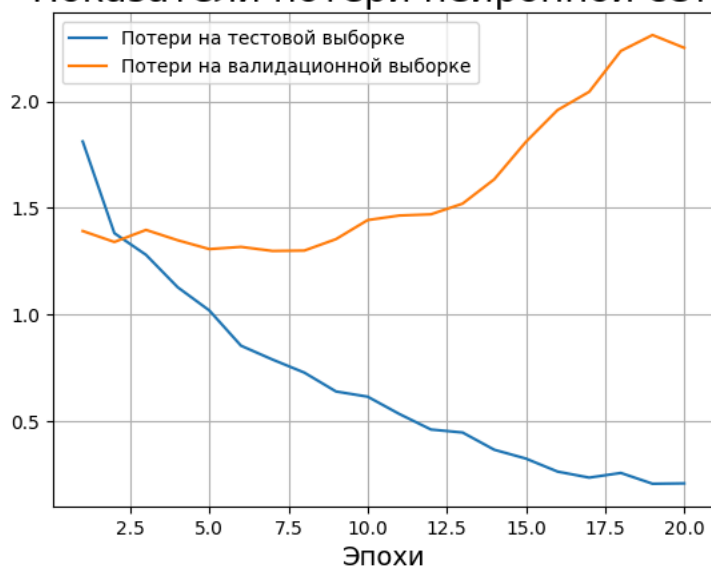


### ✓ CNN

```
plot_loss(cnn_hist, epochs = 20)
```



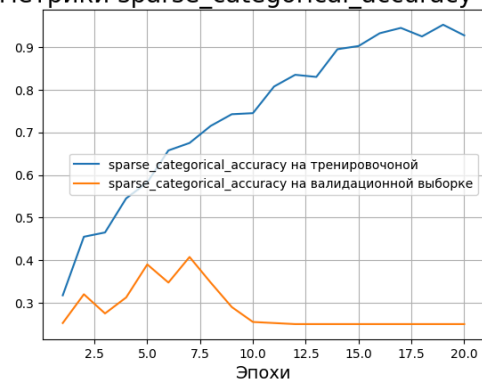
## Показатели потери нейронной сети



```
plot_metrics(cnn_hist, epochs = 20, metrics_name = 'sparse_categorical_accuracy')
```



## Показатели метрики sparse\_categorical\_accuracy нейронной сети

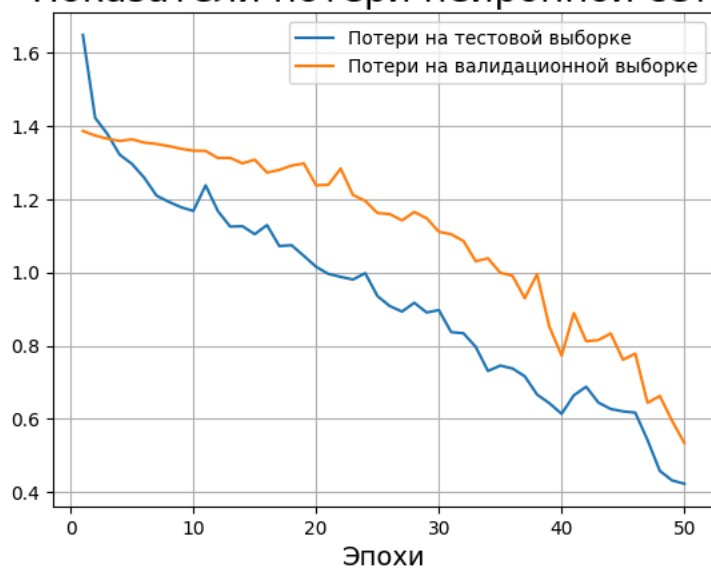


## ✓ RNN

```
plot_loss(rnn_hist, epochs = 50)
```



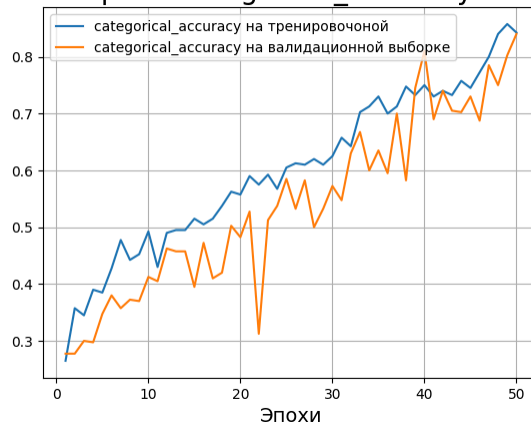
## Показатели потери нейронной сети



```
plot_metrics(rnn_hist, epochs=50, metrics_name = 'categorical_accuracy')
```



## Показатели метрики categorical\_accuracy нейронной сети



6. Визуализируйте кривые обучения трех построенных моделей для показателя

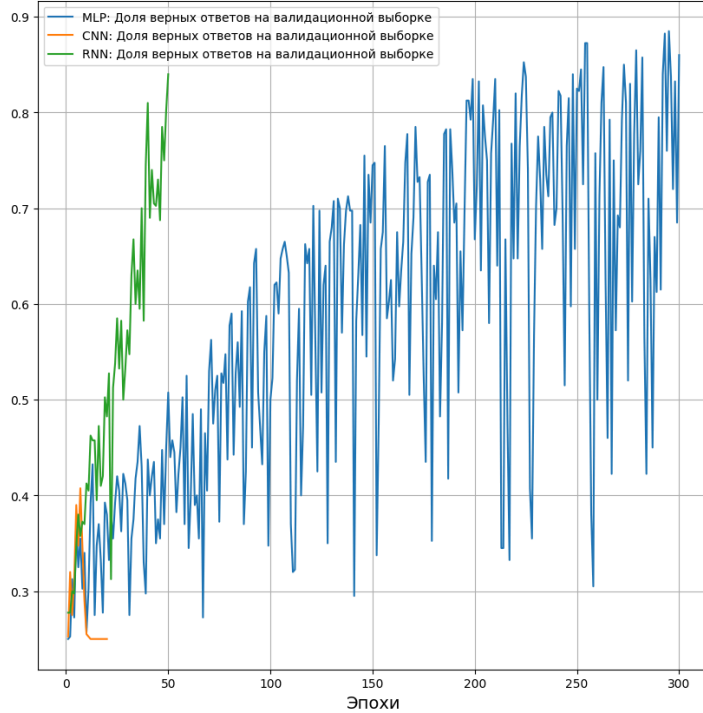
- ✓ доли верных ответов на валидационной выборке на одном рисунке в зависимости от эпохи обучения, подписывая оси и рисунок и создавая легенду.

```
plt.figure(figsize = (10, 10))
plt.plot(np.arange(1, 301), mlp_hist.history['val_accuracy'], label='MLP: Доля верных ответов на валидационной выборке')
plt.plot(np.arange(1, 21), cnn_hist.history['val_sparse_categorical_accuracy'], label='CNN: Доля верных ответов на валидационной выборке')
plt.plot(np.arange(1, 51), rnn_hist.history['val_categorical_accuracy'], label='RNN: Доля верных ответов на валидационной выборке')
# plt.ylim([0, max(mlp_hist.history['val_loss']) * 0.5])
plt.title('Показатель доли верных ответов нейронных сетей на валидационной выборке', size=17)
plt.xlabel('Эпохи', size=14)
plt.grid(True)
plt.legend()
```



 <matplotlib.legend.Legend at 0x7d7aa6ef5f00>

Показатель доли верных ответов нейронных сетей на валидационной выборке




7. Используя модель нейронной сети с лучшей долей верных ответов на тестовой выборке, определите для каждого из классов два изображения в тестовой выборке, имеющие минимальную и максимальную вероятности классификации в правильный класс, и визуализируйте эти изображения.


```
def getX(arr: np.ndarray) -> np.ndarray:
    x = np.empty((400, 96, 60, 3))
    for i in range(arr.shape[0]):
        x[i] = arr[i] / 255
    return x
```

```
test_x = getX(df_test['image'].values)
```

```
test_x[0].shape
```

 (96, 60, 3)

```
predict_x = mlp.predict(test_x[:2])
```

 1/1 [=====] - 0s 62ms/step

```
test_y = to_one_hot(df_test['label'].values, 4)
```

predict\_x

```
array([[6.6302741e-06, 9.7155231e-01, 2.6608919e-04, 2.8175112e-02],
       [7.2865081e-01, 4.3131722e-04, 2.6179436e-01, 9.1234269e-03]],
      dtype=float32)
```

predict\_x[0][np.argmax(predict\_x[0])]

```
0.9715523
```

```
def find_min_prob(model: tf.keras.Model, test_x, test_y):
    pred_y = mlp.predict(test_x)
```

```
    tp_indexes = []
```

```
    min_prob = 2
```

```
    for i in range(len(test_y)):
```

```
        if np.argmax(pred_y[i]) == np.argmax(test_y[i]):
```

```
            if pred_y[i][np.argmax(pred_y[i])] < min_prob:
```

```
                min_prob = pred_y[i][np.argmax(pred_y[i])]
```

```
                tp_indexes.append([i, min_prob])
```

```
    return tp_indexes[len(tp_indexes)-3:len(tp_indexes)-1]
```

find\_min\_prob(mlp, test\_x, test\_y)

```
13/13 [=====] - 0s 4ms/step
[[25, 0.528391], [68, 0.5116516]]
```

Данная функция определяет две минимальные вероятности определения фотографии к верному классу и возвращает индексы фотографий и саму вероятность

df\_test['image'].values[25]

```
ndarray (96, 60, 3) show data
```



df\_test['image'].values[68]

```
ndarray (96, 60, 3) show data
```



✓ Ф-ция определения минимальной вероятности правильного определения класса для фотографии

```
def find_min_prob_for_class(model: tf.keras.Model, test_x, test_y, feature: int):
    pred_y = mlp.predict(test_x)
```

```
    tp_indexes = []
```

```
    min_prob = 2
```

```
    for i in range(len(test_y)):
```

```
        if np.argmax(pred_y[i]) == np.argmax(test_y[i]) == feature:
```

```
            if pred_y[i][np.argmax(pred_y[i])] < min_prob:
```

```
                min_prob = pred_y[i][np.argmax(pred_y[i])]
```

```
                tp_indexes.append([i, min_prob])
```

```
    return tp_indexes[len(tp_indexes)-3:len(tp_indexes)-1]
```

✓ Класс 0 -- Abyssinian (cat)

```
find_min_prob_for_class(mlp, test_x, test_y, 0)
```

```
13/13 [=====] - 0s 14ms/step  
[[14, 0.67092186], [25, 0.528391]]
```

```
df_test['image'].values[14]
```

```
ndarray (96, 60, 3) show data
```



```
df_test['image'].values[25]
```

```
ndarray (96, 60, 3) show data
```



---

## ✓ Класс 1 -- chihuahua

```
find_min_prob_for_class(mlp, test_x, test_y, 1)
```

```
13/13 [=====] - 0s 4ms/step  
[[18, 0.8699674], [28, 0.6345469]]
```

```
df_test['image'].values[18]
```

```
ndarray (96, 60, 3) show data
```



```
df_test['image'].values[28]
```

```
ndarray (96, 60, 3) show data
```



---

## ✓ Класс 2 -- Maine\_Coon

```
find_min_prob_for_class(mlp, test_x, test_y, 2)
```

```
13/13 [=====] - 0s 6ms/step  
[[51, 0.5447994], [68, 0.5116516]]
```

```
df_test['image'].values[51]
```

```
ndarray (96, 60, 3) show data
```



```
df_test['image'].values[68]
```

↗ ndarray (96, 60, 3) [show data](#)



### ✓ Класс 3 - shiba\_inu

```
find_min_prob_for_class(mlp, test_x, test_y, 3)
```

↗ 13/13 [=====] - 0s 8ms/step  
[[3, 0.72566915], [13, 0.6561771]]

```
df_test['image'].values[3]
```

↗ ndarray (96, 60, 3) [show data](#)



```
df_test['image'].values[13]
```

↗ ndarray (96, 60, 3) [show data](#)



### ✓ Функция определения максимальной вероятности причисления к правильному классу

```
def find_max_prob_for_class(model: tf.keras.Model, test_x, test_y, feature: int):
    pred_y = mlp.predict(test_x)

    tp_indexes = []

    max_prob = 0
    for i in range(len(test_y)):
        if np.argmax(pred_y[i]) == np.argmax(test_y[i]) == feature:
            if pred_y[i][np.argmax(pred_y[i])] > max_prob:
                max_prob = pred_y[i][np.argmax(pred_y[i])]
                tp_indexes.append([i, max_prob])

    return tp_indexes[len(tp_indexes)-3:len(tp_indexes)-1]
```

### ✓ Класс 0 -- Abyssinian (cat)

```
find_max_prob_for_class(mlp, test_x, test_y, 0)
```


↗ 13/13 [=====] - 0s 4ms/step  
[[1, 0.7286509], [47, 0.78715193]]

```
df_test['image'].values[1]
```

↗ ndarray (96, 60, 3) [show data](#)




```
df_test['image'].values[47]
```

 ndarray (96, 60, 3) [show data](#)



## ✓ Класс 1 -- chihuahua

```
find_max_prob_for_class(mlp, test_x, test_y, 1)
```

 13/13 [=====] - 0s 8ms/step  
[[131, 0.9996285], [254, 0.99971163]]