

HW6\_Petrov.ipynb ☆

File Edit View Insert Runtime Tools Help All changes saved

Comment Share Settings

+ Code + Text RAM Disk

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. Используйте 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

валидационную и тестовую выборки. Если при дальнейшей работе с данными возникнет нехватка вычислительных ресурсов, то разрешение изображений можно уменьшить.

1. Набор данных oxford\_iit\_pet с изменением разрешения до 60x96
  2. Классы с метками 31,43,55,67

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



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



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

2. Классы с метками 31,43,55,67

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

```
[131] 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], [3]...	1
21	b'Abyssinian_196.jpg'	[[[16, 122, 120], [117, 123, 121], [119, 125, ..	0	[[[2], [2], [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, 3...	2	[[[2], [2], [2], [2], [2], [2], [2], [2], [2], [2]...	0
51	b'shiba_inu_143.jpg'	[[[146, 166, 84], [147, 168, 84], [141, 162, 8...	3	[[[2], [2], [2], [2], [2], [2], [2], [2], [2], [2]...	1
61	b'Abyssinian_134.jpg'	[[[135, 135, 137], [133, 133, 135], [136, 137, ..	0	[[[2], [2], [2], [2], [2], [2], [2], [2], [2], [2]...	0

Next steps: [View recommended plots](#)

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

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

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

```
[19] 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.

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

```
[21] 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-a-copy
df_train['label'] = df_train['label'].apply(lambda x: normal_features[x])
```

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

```
<ipython-input-22-c3bb781bd57>: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-a-copy
df_test['label'] = df_test['label'].apply(lambda x: normal_features[x])
```

```

✓ 0s [23] # tmp = df_train['image'].apply(lambda x: x / 255)

✓ 0s [24] # tmp.head()

✓ 0s [25] 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

✓ 0s [26] train_x = getX(df_train['image'].values)
train_x.shape

→ (400, 96, 60, 3)

✓ 0s [27] test_x = getX(df_test['image'].values)
test_x.shape

→ (400, 96, 60, 3)

✓ 0s [28] 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, 0.0862451, 0.09019608]],

          [[0.01568627, 0.03529412, 0.03137258],
           [0.05098839, 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, 0.39667843, 0.43921569],
           ...,
           [0.17254902, 0.11764706, 0.07843137],
           [0.29019608, 0.22352941, 0.16078431],
           [0.38431573, 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, 0.14598904, 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, 0.14598904, 0.19980392],
           [0.12941178, 0.08235294, 0.03921569],
           [0.31764706, 0.23921569, 0.16470588]]])

✓ 0s [29] # train_x = np.empty((400, 96, 60, 3))
# for i in range(tmp.shape[0]):
#     train_x[i] = tmp.values[i]

✓ 0s [30] # train_x[130].shape
# df_train['image'][21]
# train_x[0]

✓ 0s [31] 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

✓ 0s [32] train_y = to_one_hot(df_train['label'].values, 4)

✓ 0s [33] test_y = to_one_hot(df_test['label'].values, 4)

✓ 0s [34] 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

```

✓ [35] 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(50, 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'))

✓ [36] 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")]
)

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

```

Epoch 89/300  
7/7 [=====] - 0s 39ms/step - loss: 1.8217 - accuracy: 0.5175 - val\_loss: 1.6800 - val\_accuracy: 0.6825  
Epoch 90/300  
7/7 [=====] - 0s 38ms/step - loss: 1.7362 - accuracy: 0.5300 - val\_loss: 1.6408 - val\_accuracy: 0.6175  
Epoch 91/300  
7/7 [=====] - 0s 37ms/step - loss: 1.6604 - accuracy: 0.5875 - val\_loss: 1.7581 - val\_accuracy: 0.4500  
Epoch 92/300  
7/7 [=====] - 0s 38ms/step - loss: 1.6439 - accuracy: 0.5700 - val\_loss: 1.5553 - val\_accuracy: 0.6425  
Epoch 93/300  
7/7 [=====] - 0s 39ms/step - loss: 1.6138 - accuracy: 0.5525 - val\_loss: 1.5417 - val\_accuracy: 0.6575  
Epoch 94/300  
7/7 [=====] - 0s 38ms/step - loss: 1.5989 - accuracy: 0.5925 - val\_loss: 1.7446 - val\_accuracy: 0.5075  
Epoch 95/300  
7/7 [=====] - 0s 38ms/step - loss: 1.6696 - accuracy: 0.5275 - val\_loss: 1.8226 - val\_accuracy: 0.4700  
Epoch 96/300  
7/7 [=====] - 0s 42ms/step - loss: 1.7305 - accuracy: 0.5350 - val\_loss: 1.7995 - val\_accuracy: 0.4325  
Epoch 97/300  
7/7 [=====] - 0s 40ms/step - loss: 1.6856 - accuracy: 0.5525 - val\_loss: 1.6686 - val\_accuracy: 0.5500  
Epoch 98/300  
7/7 [=====] - 0s 42ms/step - loss: 1.6100 - accuracy: 0.5775 - val\_loss: 1.5540 - val\_accuracy: 0.5875  
Epoch 99/300  
7/7 [=====] - 0s 38ms/step - loss: 1.5505 - accuracy: 0.5800 - val\_loss: 1.9899 - val\_accuracy: 0.3475  
Epoch 100/300  
7/7 [=====] - 0s 40ms/step - loss: 1.7802 - accuracy: 0.4200 - val\_loss: 1.7252 - val\_accuracy: 0.5000  
Epoch 101/300  
7/7 [=====] - 0s 39ms/step - loss: 1.8132 - accuracy: 0.4900 - val\_loss: 1.8071 - val\_accuracy: 0.5225  
Epoch 102/300  
7/7 [=====] - 0s 39ms/step - loss: 1.7289 - accuracy: 0.5600 - val\_loss: 1.5933 - val\_accuracy: 0.6200  
Epoch 103/300  
7/7 [=====] - 0s 40ms/step - loss: 1.6481 - accuracy: 0.6025 - val\_loss: 1.5812 - val\_accuracy: 0.6225  
Epoch 104/300  
7/7 [=====] - 0s 28ms/step - loss: 1.6139 - accuracy: 0.5800 - val\_loss: 1.5666 - val\_accuracy: 0.5900  
Epoch 105/300  
7/7 [=====] - 0s 25ms/step - loss: 1.6147 - accuracy: 0.5750 - val\_loss: 1.5317 - val\_accuracy: 0.6475  
Epoch 106/300  
7/7 [=====] - 0s 25ms/step - loss: 1.5392 - accuracy: 0.5850 - val\_loss: 1.4598 - val\_accuracy: 0.6575  
Epoch 107/300  
7/7 [=====] - 0s 30ms/step - loss: 1.5953 - accuracy: 0.5525 - val\_loss: 1.4788 - val\_accuracy: 0.6650  
Epoch 108/300  
7/7 [=====] - 0s 29ms/step - loss: 1.5217 - accuracy: 0.6200 - val\_loss: 1.4339 - val\_accuracy: 0.6500  
Epoch 109/300  
7/7 [=====] - 0s 28ms/step - loss: 1.4888 - accuracy: 0.6325 - val\_loss: 1.4867 - val\_accuracy: 0.6325  
Epoch 110/300  
7/7 [=====] - 0s 26ms/step - loss: 1.5387 - accuracy: 0.5950 - val\_loss: 1.9021 - val\_accuracy: 0.3700  
Epoch 111/300  
7/7 [=====] - 0s 26ms/step - loss: 1.5975 - accuracy: 0.5550 - val\_loss: 2.0664 - val\_accuracy: 0.3200  
Epoch 112/300  
7/7 [=====] - 0s 27ms/step - loss: 1.6887 - accuracy: 0.5175 - val\_loss: 2.2554 - val\_accuracy: 0.3225  
Epoch 113/300  
7/7 [=====] - 0s 31ms/step - loss: 2.0031 - accuracy: 0.4000 - val\_loss: 1.7542 - val\_accuracy: 0.5200  
Epoch 114/300  
7/7 [=====] - 0s 32ms/step - loss: 1.8337 - accuracy: 0.5150 - val\_loss: 1.6928 - val\_accuracy: 0.5950  
Epoch 115/300  
7/7 [=====] - 0s 26ms/step - loss: 1.6862 - accuracy: 0.6000 - val\_loss: 1.9520 - val\_accuracy: 0.4000  
Epoch 116/300  
7/7 [=====] - 0s 26ms/step - loss: 1.8759 - accuracy: 0.4325 - val\_loss: 1.9759 - val\_accuracy: 0.4725  
Epoch 117/300  
7/7 [=====] - 0s 25ms/step - loss: 1.9771 - accuracy: 0.4425 - val\_loss: 1.6956 - val\_accuracy: 0.6625

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

```

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

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

✓ [40] test_y = df_test['label'].values

✓ [41] 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

→ Код во следующем З.

• Код для слоя Conv2D

- Количество фильтров в сверточных слоях 16

- Размеры фильтра 5x5

- Использование слоев пакетной нормализации

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

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

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

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

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

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

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

[49] 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)

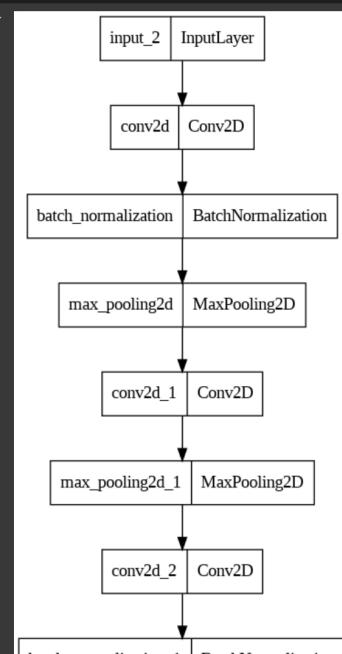
[50] cnn = tf.keras.Model(inputs = inputs, outputs = outputs)
cnn.summary()
```

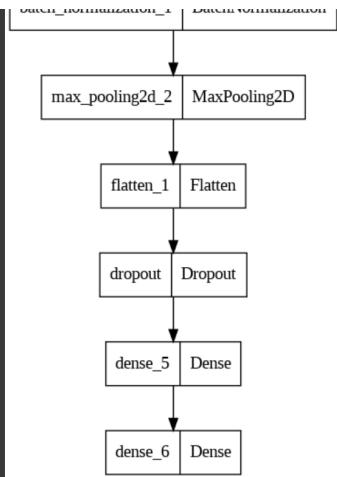
Model: "model"

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)

```
[51] tf.keras.utils.plot_model(cnn)
```





```

✓ [52] cnn.compile(
    loss = tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer = tf.keras.optimizers.Adam(),
    metrics = [tf.keras.metrics.SparseCategoricalAccuracy()])
)

✓ [53] 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: 0.2525
Epoch 2/20
4/4 [=====] - 9s 2s/step - loss: 1.3815 - sparse_categorical_accuracy: 0.4550 - val_loss: 1.3403 - val_sparse_categorical_accuracy: 0.3200
Epoch 3/20
4/4 [=====] - 11s 2s/step - loss: 1.2798 - sparse_categorical_accuracy: 0.4650 - val_loss: 1.3968 - val_sparse_categorical_accuracy: 0.2750
Epoch 4/20
4/4 [=====] - 12s 3s/step - loss: 1.1278 - sparse_categorical_accuracy: 0.5450 - val_loss: 1.3484 - val_sparse_categorical_accuracy: 0.3125
Epoch 5/20
4/4 [=====] - 11s 3s/step - loss: 1.0189 - sparse_categorical_accuracy: 0.5825 - val_loss: 1.3068 - val_sparse_categorical_accuracy: 0.3900
Epoch 6/20
4/4 [=====] - 10s 3s/step - loss: 0.8536 - sparse_categorical_accuracy: 0.6575 - val_loss: 1.3172 - val_sparse_categorical_accuracy: 0.3475
Epoch 7/20
4/4 [=====] - 11s 3s/step - loss: 0.7883 - sparse_categorical_accuracy: 0.6750 - val_loss: 1.2981 - val_sparse_categorical_accuracy: 0.4075
Epoch 8/20
4/4 [=====] - 11s 3s/step - loss: 0.7274 - sparse_categorical_accuracy: 0.7150 - val_loss: 1.2999 - val_sparse_categorical_accuracy: 0.3475
Epoch 9/20
4/4 [=====] - 12s 3s/step - loss: 0.6389 - sparse_categorical_accuracy: 0.7425 - val_loss: 1.3535 - val_sparse_categorical_accuracy: 0.2900
Epoch 10/20
4/4 [=====] - 13s 3s/step - loss: 0.6144 - sparse_categorical_accuracy: 0.7458 - val_loss: 1.4429 - val_sparse_categorical_accuracy: 0.2550
Epoch 11/20
4/4 [=====] - 10s 2s/step - loss: 0.5334 - sparse_categorical_accuracy: 0.8075 - val_loss: 1.4644 - val_sparse_categorical_accuracy: 0.2525
Epoch 12/20
4/4 [=====] - 10s 2s/step - loss: 0.4604 - sparse_categorical_accuracy: 0.8350 - val_loss: 1.4701 - val_sparse_categorical_accuracy: 0.2500
Epoch 13/20
4/4 [=====] - 12s 3s/step - loss: 0.4463 - sparse_categorical_accuracy: 0.8300 - val_loss: 1.5201 - val_sparse_categorical_accuracy: 0.2500
Epoch 14/20
4/4 [=====] - 11s 3s/step - loss: 0.3655 - sparse_categorical_accuracy: 0.8950 - val_loss: 1.6344 - val_sparse_categorical_accuracy: 0.2500
Epoch 15/20
4/4 [=====] - 13s 3s/step - loss: 0.3239 - sparse_categorical_accuracy: 0.9025 - val_loss: 1.8099 - val_sparse_categorical_accuracy: 0.2500
Epoch 16/20
4/4 [=====] - 13s 3s/step - loss: 0.2630 - sparse_categorical_accuracy: 0.9325 - val_loss: 1.9591 - val_sparse_categorical_accuracy: 0.2500
Epoch 17/20
4/4 [=====] - 10s 2s/step - loss: 0.2347 - sparse_categorical_accuracy: 0.9458 - val_loss: 2.0449 - val_sparse_categorical_accuracy: 0.2500
Epoch 18/20
4/4 [=====] - 11s 3s/step - loss: 0.2566 - sparse_categorical_accuracy: 0.9250 - val_loss: 2.2366 - val_sparse_categorical_accuracy: 0.2500
Epoch 19/20
4/4 [=====] - 11s 3s/step - loss: 0.2060 - sparse_categorical_accuracy: 0.9525 - val_loss: 2.3112 - val_sparse_categorical_accuracy: 0.2500
Epoch 20/20
4/4 [=====] - 11s 3s/step - loss: 0.2078 - sparse_categorical_accuracy: 0.9275 - val_loss: 2.2512 - val_sparse_categorical_accuracy: 0.2500

```

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

```

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

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

✓ [56] from PIL import ImageOps

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

```

```

  [58] 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

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

  [60] train_x.shape
  ⏺ (400, 96, 96)

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

  [65] 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')
  ])

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

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

```

Epoch 22/50  
 7/7 [=====] - 1s 156ms/step - loss: 0.9881 - categorical\_accuracy: 0.5750 - val\_loss: 1.2840 - val\_categorical\_accuracy: 0.3125  
 Epoch 23/50  
 7/7 [=====] - 1s 160ms/step - loss: 0.9808 - categorical\_accuracy: 0.5925 - val\_loss: 1.2120 - val\_categorical\_accuracy: 0.5125  
 Epoch 24/50  
 7/7 [=====] - 1s 156ms/step - loss: 0.9981 - categorical\_accuracy: 0.5675 - val\_loss: 1.1958 - val\_categorical\_accuracy: 0.5375  
 Epoch 25/50  
 7/7 [=====] - 2s 248ms/step - loss: 0.9354 - categorical\_accuracy: 0.6050 - val\_loss: 1.1625 - val\_categorical\_accuracy: 0.5850  
 Epoch 26/50  
 7/7 [=====] - 2s 294ms/step - loss: 0.9083 - categorical\_accuracy: 0.6125 - val\_loss: 1.1595 - val\_categorical\_accuracy: 0.5325  
 Epoch 27/50  
 7/7 [=====] - 1s 165ms/step - loss: 0.8932 - categorical\_accuracy: 0.6100 - val\_loss: 1.1422 - val\_categorical\_accuracy: 0.5825  
 Epoch 28/50  
 7/7 [=====] - 1s 152ms/step - loss: 0.9174 - categorical\_accuracy: 0.6200 - val\_loss: 1.1653 - val\_categorical\_accuracy: 0.5000  
 Epoch 29/50  
 7/7 [=====] - 1s 157ms/step - loss: 0.8908 - categorical\_accuracy: 0.6100 - val\_loss: 1.1483 - val\_categorical\_accuracy: 0.5325  
 Epoch 30/50  
 7/7 [=====] - 1s 158ms/step - loss: 0.8973 - categorical\_accuracy: 0.6250 - val\_loss: 1.1113 - val\_categorical\_accuracy: 0.5725  
 Epoch 31/50  
 7/7 [=====] - 1s 156ms/step - loss: 0.8372 - categorical\_accuracy: 0.6575 - val\_loss: 1.1045 - val\_categorical\_accuracy: 0.5475  
 Epoch 32/50  
 7/7 [=====] - 1s 156ms/step - loss: 0.8341 - categorical\_accuracy: 0.6425 - val\_loss: 1.0863 - val\_categorical\_accuracy: 0.6300  
 Epoch 33/50  
 7/7 [=====] - 1s 158ms/step - loss: 0.7970 - categorical\_accuracy: 0.7025 - val\_loss: 1.0311 - val\_categorical\_accuracy: 0.6675  
 Epoch 34/50  
 7/7 [=====] - 1s 162ms/step - loss: 0.7310 - categorical\_accuracy: 0.7125 - val\_loss: 1.0386 - val\_categorical\_accuracy: 0.6000  
 Epoch 35/50  
 7/7 [=====] - 1s 158ms/step - loss: 0.7457 - categorical\_accuracy: 0.7300 - val\_loss: 0.9995 - val\_categorical\_accuracy: 0.6350  
 Epoch 36/50  
 7/7 [=====] - 1s 206ms/step - loss: 0.7378 - categorical\_accuracy: 0.7000 - val\_loss: 0.9909 - val\_categorical\_accuracy: 0.5950  
 Epoch 37/50  
 7/7 [=====] - 2s 299ms/step - loss: 0.7165 - categorical\_accuracy: 0.7125 - val\_loss: 0.9297 - val\_categorical\_accuracy: 0.7000  
 Epoch 38/50  
 7/7 [=====] - 1s 195ms/step - loss: 0.6667 - categorical\_accuracy: 0.7475 - val\_loss: 0.9954 - val\_categorical\_accuracy: 0.5825  
 Epoch 39/50  
 7/7 [=====] - 1s 155ms/step - loss: 0.6430 - categorical\_accuracy: 0.7325 - val\_loss: 0.8518 - val\_categorical\_accuracy: 0.7450  
 Epoch 40/50  
 7/7 [=====] - 1s 156ms/step - loss: 0.6148 - categorical\_accuracy: 0.7500 - val\_loss: 0.7731 - val\_categorical\_accuracy: 0.8100  
 Epoch 41/50  
 7/7 [=====] - 1s 155ms/step - loss: 0.6645 - categorical\_accuracy: 0.7300 - val\_loss: 0.8894 - val\_categorical\_accuracy: 0.6900  
 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: 0.7050  
 Epoch 44/50  
 7/7 [=====] - 1s 156ms/step - loss: 0.6273 - categorical\_accuracy: 0.7575 - val\_loss: 0.8334 - val\_categorical\_accuracy: 0.7025  
 Epoch 45/50  
 7/7 [=====] - 1s 156ms/step - loss: 0.6207 - categorical\_accuracy: 0.7450 - val\_loss: 0.7617 - val\_categorical\_accuracy: 0.7300  
 Epoch 46/50  
 7/7 [=====] - 1s 155ms/step - loss: 0.6172 - categorical\_accuracy: 0.7725 - val\_loss: 0.7785 - val\_categorical\_accuracy: 0.6875  
 Epoch 47/50  
 7/7 [=====] - 1s 181ms/step - loss: 0.5423 - categorical\_accuracy: 0.8000 - val\_loss: 0.6439 - val\_categorical\_accuracy: 0.7850  
 Epoch 48/50  
 7/7 [=====] - 2s 361ms/step - loss: 0.4581 - categorical\_accuracy: 0.8400 - val\_loss: 0.6627 - val\_categorical\_accuracy: 0.7500  
 Epoch 49/50  
 7/7 [=====] - 1s 177ms/step - loss: 0.4321 - categorical\_accuracy: 0.8575 - val\_loss: 0.5947 - val\_categorical\_accuracy: 0.8025  
 Epoch 50/50  
 7/7 [=====] - 1s 164ms/step - loss: 0.4230 - categorical\_accuracy: 0.8425 - val\_loss: 0.5345 - val\_categorical\_accuracy: 0.8400

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.

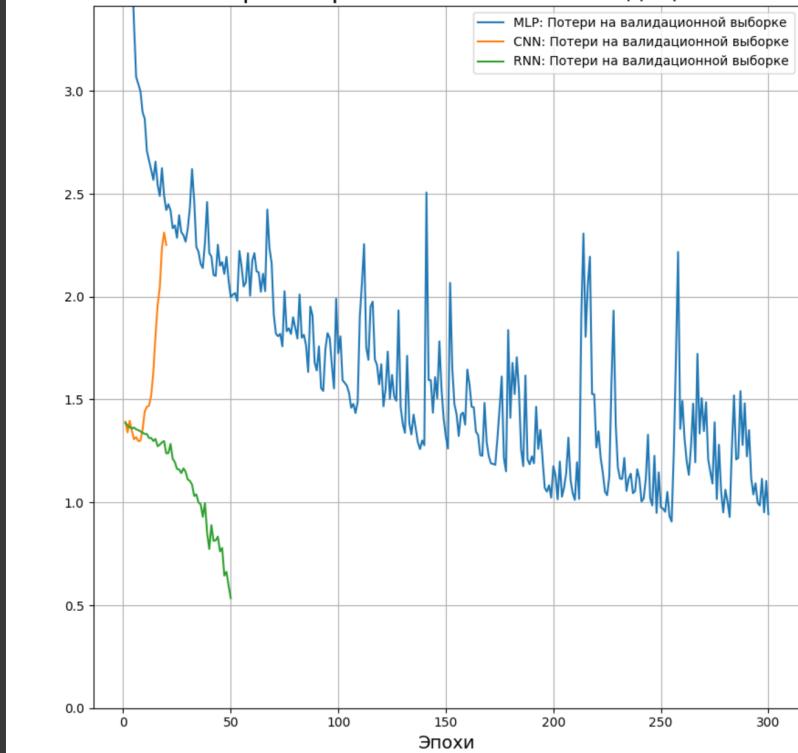
5. Визуализируйте кривые обучения трех построенных моделей для показателя потерь на валидационной выборке на одном рисунке в зависимости от эпохи
- ✓ обучения, подписывая оси и рисунок и создавая легенду. Используйте для визуализации относительные потери (потери, деленные на начальные потери на первой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

```
[101] 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();

✓ [120] 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')
    plt.title(f'Показатели метрики {metrics_name} нейронной сети', size=20)
    plt.xlabel('Эпохи', size=14)
    plt.grid(True)
    plt.legend();

✓ [128] 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), cmn_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.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

```
✓ [121] plot_loss(mlp_hist, epochs = 300, lim = True)
```

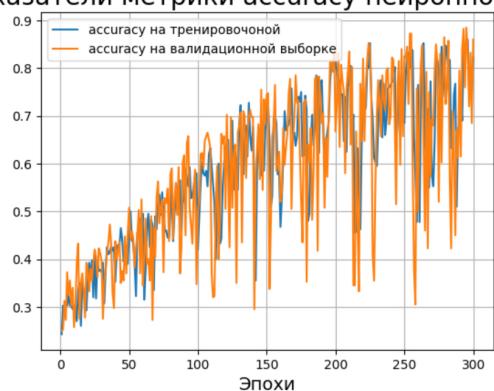
↗ Показатели потери нейронной сети





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

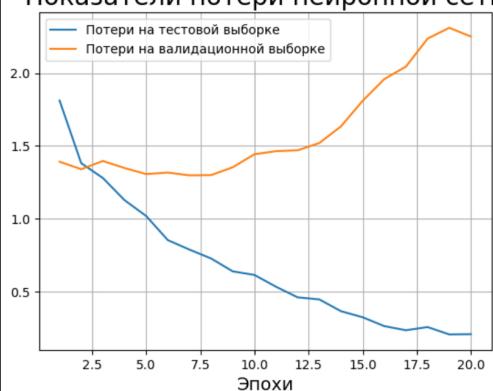
### Показатели метрики accuracy нейронной сети



### ▽ CNN

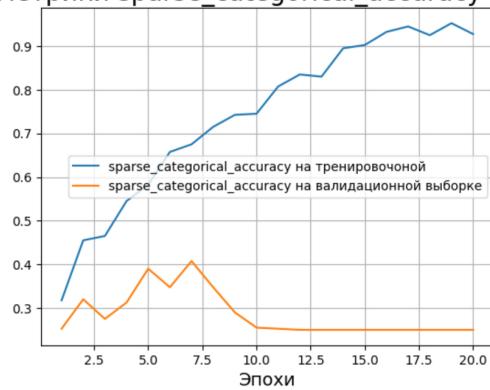
```
[123] plot_loss(cnn_hist, epochs = 20)
```

### Показатели потери нейронной сети



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

### Показатели метрики sparse\_categorical\_accuracy нейронной сети

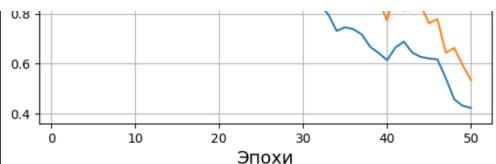


### ▽ RNN

```
[125] plot_loss(rnn_hist, epochs = 50)
```

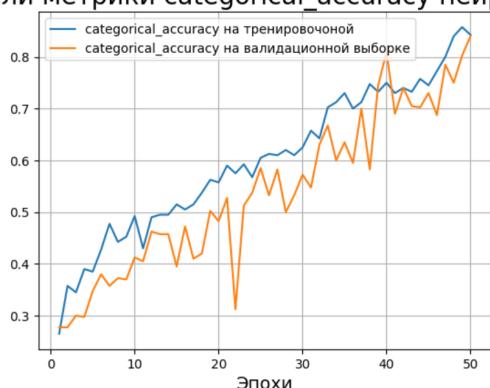
### Показатели потери нейронной сети





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

### ↗ Показатели метрики categorical\_accuracy нейронной сети



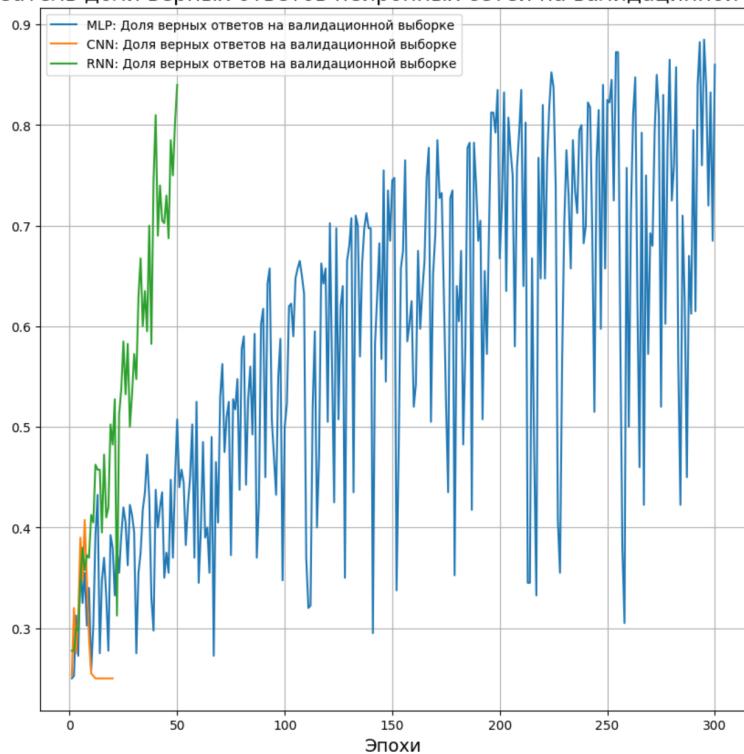
6. Визуализируйте кривые обучения трех построенных моделей для показателя

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

```
[130] 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 предсказанный класс и визуализируйте эти изображения

## 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

[135] test_x = getx(df_test['image'].values)

[140] test_x[0].shape
→ (96, 60, 3)

[142] predict_x = mlp.predict(test_x[:2])
→ 1/1 [=====] - 0s 62ms/step

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

[161] 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)

[165] predict_x[0][np.argmax(predict_x[0])]

→ 0.9715523

[179] 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]

[180] 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

```
[181] df_test['image'].values[25]
→ ndarray (96, 60, 3) show data
  
  
[174] df_test['image'].values[68]
→ ndarray (96, 60, 3) show data

```

Ф-ция определния минимальной вероятности правильного определения класса для фотографии

```
[182] 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)

```
[183] find_min_prob_for_class(mlp, test_x, test_y, 0)
→ 13/13 [=====] - 0s 14ms/step
[[14, 0.67092186], [25, 0.528391]]  
  
[185] df_test['image'].values[14]
```



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

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



#### ▼ Класс 1 -- chihuahua

```
✓ [187] find_min_prob_for_class(mlp, test_x, test_y, 1)
```

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

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

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



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

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



#### ▼ Класс 2 -- Maine\_Coon

```
✓ [190] find_min_prob_for_class(mlp, test_x, test_y, 2)
```

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

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

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



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

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



#### ▼ Класс 3 - shiba\_inu

```
✓ [193] find_min_prob_for_class(mlp, test_x, test_y, 3)
```

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

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

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



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

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



#### ▼ Функция определения максимальной вероятности принадлежности к правильному классу

```
✓ [204] 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)

```

[205] find_max_prob_for_class(mlp, test_x, test_y, 0)
13/13 [=====] - 0s 4ms/step
[[1, 0.7286509], [47, 0.78715193]]

```

```

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

```



```

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

```



#### ▼ Класс 1 -- chihuahua

```

[208] find_max_prob_for_class(mlp, test_x, test_y, 1)
13/13 [=====] - 0s 8ms/step
[[131, 0.9996285], [254, 0.99971163]]

```

```

[210] df_test['image'].values[131]

```



```

[209] df_test['image'].values[254]

```



#### ▼ Класс 2 -- Maine\_Coon

```

[211] find_max_prob_for_class(mlp, test_x, test_y, 2)
13/13 [=====] - 0s 4ms/step
[[23, 0.6633381], [74, 0.6653939]]

```

```

[212] df_test['image'].values[23]

```



```

[213] df_test['image'].values[74]

```



#### ▼ Класс 3 - shiba\_inu

```

[214] find_max_prob_for_class(mlp, test_x, test_y, 3)
13/13 [=====] - 0s 6ms/step
[[124, 0.8576575], [214, 0.8579284]]

```

```

[215] df_test['image'].values[124]

```





```
[216] df_test['image'].values[214]
```

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



*Вывод:* На этой контрольной мы построили три нейронные сети: MLP, CNN, RNN. MLP и RNN обучились очень хорошо(можно было даже дообучить). Однако, CNN у нас переобучилась, но точность не стала лучше случайной классификации. Хотелось бы отметить, что RNN показала себя лучше всего, так как хватило даже 50 эпох для хорошего обучения.

Colab paid products - Cancel contracts here

✓ 0s completed at 3:51PM

