РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра информационных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 6

Дисциплина: Методы машинного обучения

Студент: Шалыгин Георгий

Группа: НФИ-02

Москва 2023

Вариант № 18

1. Загрузите заданный в индивидуальном задании набор данных с изображениями из **Tensorflow Datasets** с разбиением на обучающую и тестовую выборки. Набор данных **stl10**

In [2]:

```
#!pip install -q tfds-nightly
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
import tensorflow_datasets as tfds
from PIL import Image, ImageOps
```

In [3]:

```
ds = tfds.load("stl10", split=['train','test'])
df_train = tfds.as_dataframe(ds[0])
df_test  = tfds.as_dataframe(ds[1])
df_train.shape, df_test.shape
```

Out[3]:

```
((5000, 2), (8000, 2))
```

In [4]:

```
df_train.head()
```

Out[4]:

	image	label
0	[[[136, 144, 153], [125, 127, 136], [125, 126,...	1
1	[[[70, 132, 186], [81, 139, 189], [143, 176, 2...	0
2	[[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], ...	8
3	[[[104, 90, 69], [101, 87, 66], [103, 88, 67],...	3

4	[[[189, 204, 235], [189, 204, 235], [189, 204 image	label 9

In [5]:

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

Out[5]:

```
(96, 96, 3)
```

In [6]:

```
train_labels = df_train['label'].to_numpy(dtype=np.float32)
test_labels = df_test['label'].to_numpy(dtype=np.float32)
train_labels.shape, test_labels.shape

train_images = np.zeros(shape=(df_train.shape[0],96,96,3), dtype=np.float32)
test_images  = np.zeros(shape=(df_test.shape[0],96,96,3), dtype=np.float32)
train_images.shape, test_images.shape

for idx in range(train_labels.shape[0]):
    train_images[idx,:,:,:] = np.array(Image.fromarray(df_train.iloc[idx]['image']))

for idx in range(test_labels.shape[0]):
    test_images[idx,:,:,:] = np.array(Image.fromarray(df_test.iloc[idx]['image']))

train_images /= 255
test_images /= 255

train_images.shape, test_images.shape
```

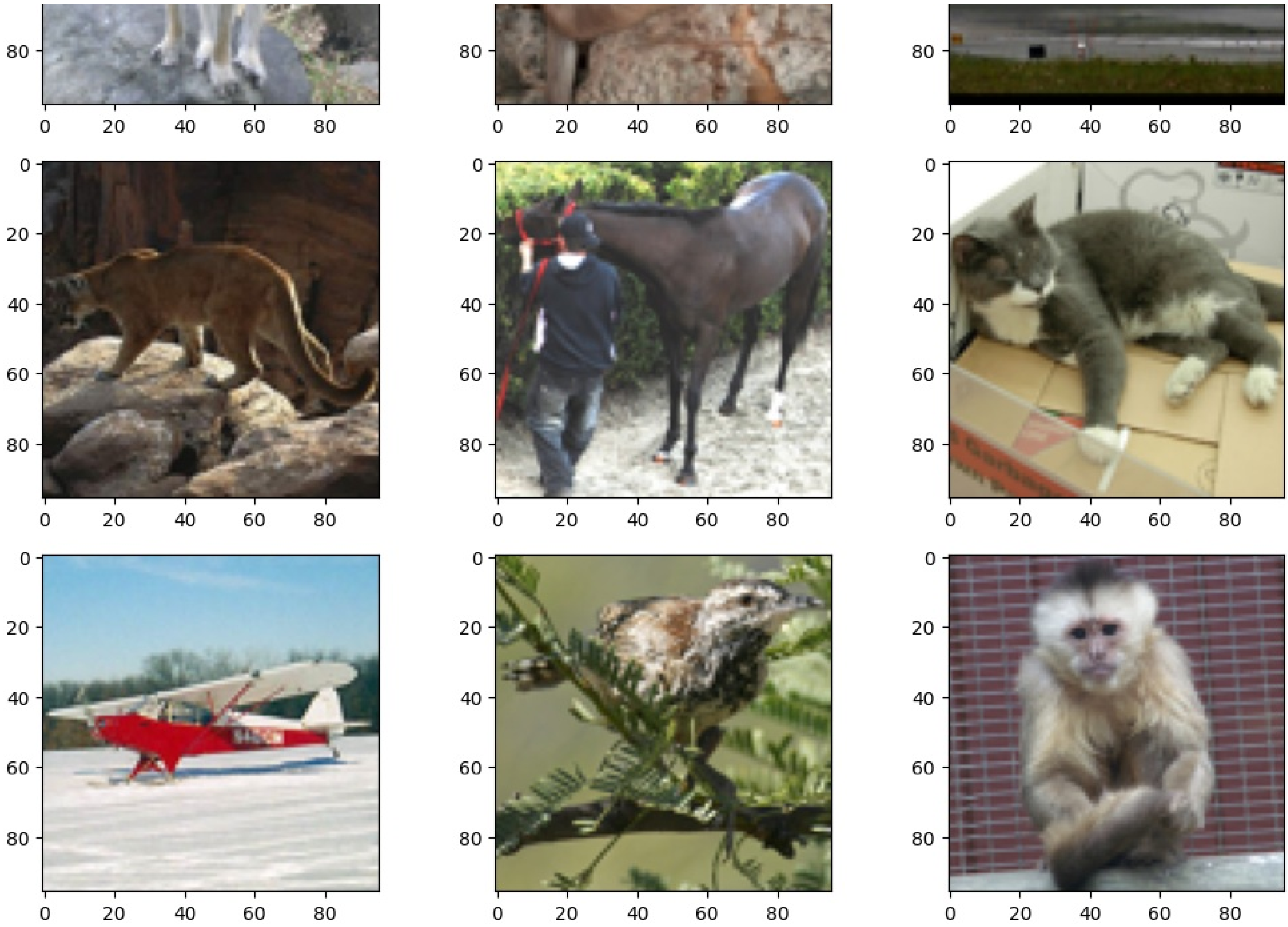
Out[6]:

```
((5000, 96, 96, 3), (8000, 96, 96, 3))
```

1. **Визуализируйте несколько изображений, отобранных случайным образом из обучающей выборки.**

In [7]:

```
import random

def plot_random_sample(images):
    n = 10
    imgs = random.sample(list(images), n)

    num_row = 3
    num_col = 3

    fig, axes = plt.subplots(num_row, num_col, figsize=(3.5 * num_col, 3 * num_row))
    # For every image
    for i in range(num_row * num_col):
        # Read the image
        img = imgs[i]
        # Display the image
        ax = axes[i // num_col, i % num_col]
        ax.imshow(img)

    plt.tight_layout()
    plt.show()

plot_random_sample(test_images)
```



1. Оставьте в наборе изображения двух классов, указанных в индивидуальном задании первыми. Обучите нейронные сети **MLP** и **CNN** задаче бинарной классификации изображений (архитектура сетей по вашему усмотрению).

   Классы с метками **1,3,5**

In [8]:

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

Out[8]:

```
array([1, 0, 8, 3, 9, 2, 4, 6, 7, 5])
```

In [9]:

```
df_train1 = df_train[df_train['label'].isin([1, 3])]
df_test1 = df_test[df_test['label'].isin([1, 3])]
df_train1.loc[df_train1['label'] == 3,'label'] = 0
df_test1.loc[df_test1['label'] == 3,'label'] = 0

df_train1.shape, df_test1.shape
```

Out[9]:

```
((1000, 2), (1600, 2))
```

In [14]:

```
train_labels = df_train1['label'].to_numpy(dtype=np.float32)
test_labels = df_test1['label'].to_numpy(dtype=np.float32)

train_images = np.zeros(shape=(df_train1.shape[0],96,96,3), dtype=np.float32)
test_images  = np.zeros(shape=(df_test1.shape[0],96,96,3), dtype=np.float32)

for idx in range(train_labels.shape[0]):
```

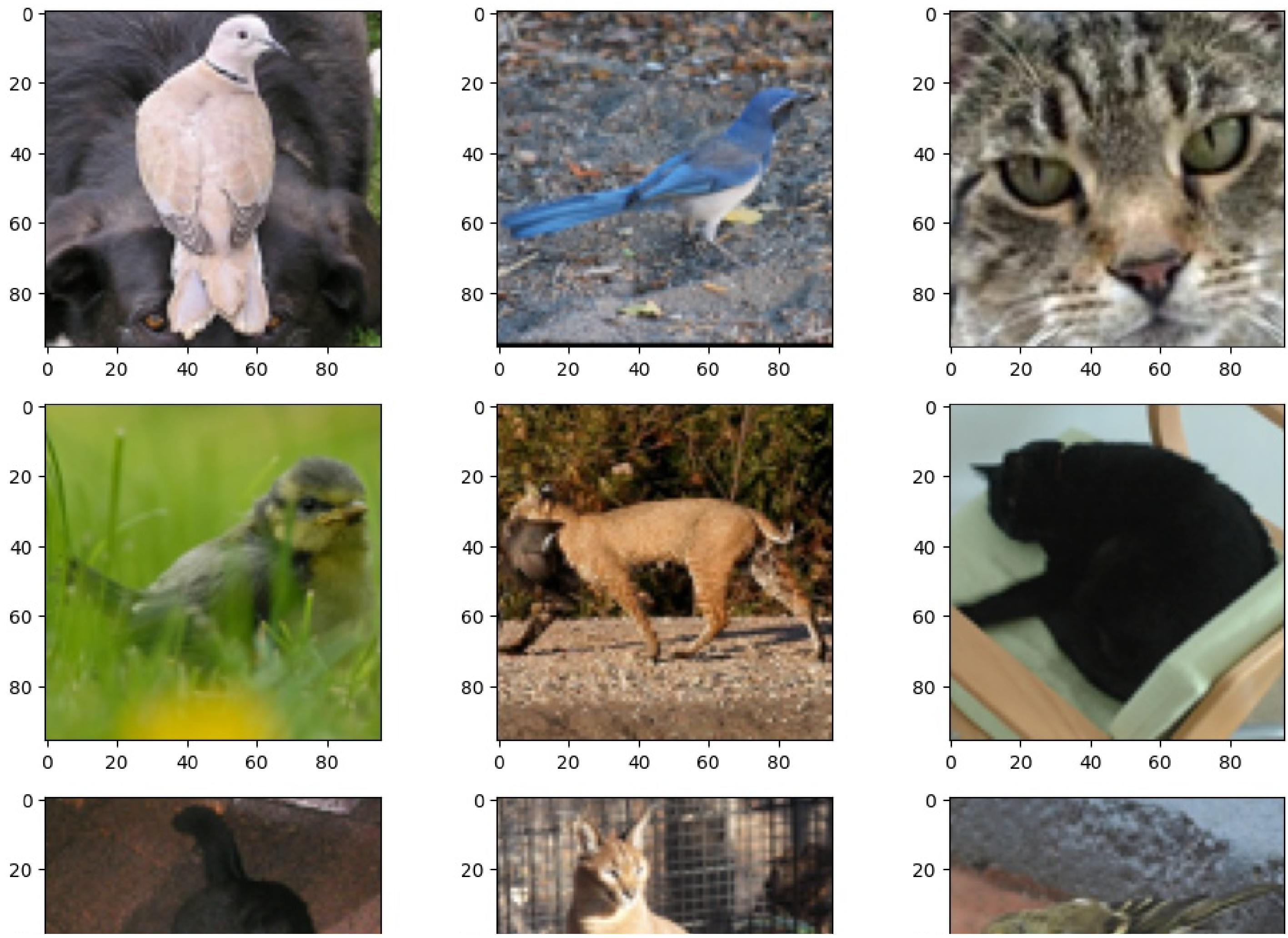
```python
    train_images[idx,:,:,:] = np.array(Image.fromarray(df_train1.iloc[idx]['image']))

for idx in range(test_labels.shape[0]):
    test_images[idx,:,:,:] = np.array(Image.fromarray(df_test1.iloc[idx]['image']))

train_images /= 255
test_images /= 255

train_images.shape, test_images.shape
```

Out[14]:

```
((1000, 96, 96, 3), (1600, 96, 96, 3))
```

In [15]:

```python
import random

def plot_random_sample(images):
    n = 10
    imgs = random.sample(list(images), n)

    num_row = 3
    num_col = 3

    fig, axes = plt.subplots(num_row, num_col, figsize=(3.5 * num_col, 3 * num_row))
    # For every image
    for i in range(num_row * num_col):
        # Read the image
        img = imgs[i]
        # Display the image
        ax = axes[i // num_col, i % num_col]
        ax.imshow(img)

    plt.tight_layout()
    plt.show()

plot_random_sample(test_images)
```



In [17]:

```
tf.random.set_seed(42)

model_1 = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(96, 96, 3)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model_1.compile(
    loss='bce',
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    metrics=[tf.keras.metrics.BinaryAccuracy(name='accuracy')]
)

history_1 = model_1.fit(
    train_images,
    train_labels,
    epochs=50,
    batch_size=256,
    validation_data=(test_images, test_labels)
)
```

```
Epoch 1/50
4/4 [==============================] - 2s 196ms/step - loss: 1.5018 - accuracy: 0.5020 -
val_loss: 0.6648 - val_accuracy: 0.5875
Epoch 2/50
4/4 [==============================] - 0s 75ms/step - loss: 0.9959 - accuracy: 0.5320 - v
al_loss: 0.7154 - val_accuracy: 0.5206
Epoch 3/50
4/4 [==============================] - 0s 75ms/step - loss: 0.7851 - accuracy: 0.5010 - v
al_loss: 0.8570 - val_accuracy: 0.5000
Epoch 4/50
4/4 [==============================] - 0s 78ms/step - loss: 0.7213 - accuracy: 0.5600 - v
al_loss: 0.7212 - val_accuracy: 0.5169
Epoch 5/50
4/4 [==============================] - 0s 70ms/step - loss: 0.7397 - accuracy: 0.5040 - v
al_loss: 0.6717 - val_accuracy: 0.5525
Epoch 6/50
4/4 [==============================] - 0s 60ms/step - loss: 0.6711 - accuracy: 0.5670 - v
al_loss: 0.7115 - val_accuracy: 0.5256
Epoch 7/50
4/4 [==============================] - 0s 63ms/step - loss: 0.6771 - accuracy: 0.5740 - v
al_loss: 0.6599 - val_accuracy: 0.5819
Epoch 8/50
4/4 [==============================] - 0s 79ms/step - loss: 0.6678 - accuracy: 0.5590 - v
al_loss: 0.6634 - val_accuracy: 0.5719
Epoch 9/50
4/4 [==============================] - 0s 66ms/step - loss: 0.6449 - accuracy: 0.6120 - v
al_loss: 0.6717 - val_accuracy: 0.6019
Epoch 10/50
4/4 [==============================] - 0s 65ms/step - loss: 0.6560 - accuracy: 0.6120 - v
al_loss: 0.6505 - val_accuracy: 0.6075
Epoch 11/50
4/4 [==============================] - 0s 69ms/step - loss: 0.6429 - accuracy: 0.6160 - v
al_loss: 0.6539 - val_accuracy: 0.5831
Epoch 12/50
4/4 [==============================] - 0s 68ms/step - loss: 0.6355 - accuracy: 0.6200 - v
al_loss: 0.6533 - val_accuracy: 0.6169
Epoch 13/50
```

```
4/4 [==============================] - 0s 69ms/step - loss: 0.6367 - accuracy: 0.6360 - v
al_loss: 0.6460 - val_accuracy: 0.6144
Epoch 14/50
4/4 [==============================] - 0s 65ms/step - loss: 0.6365 - accuracy: 0.6260 - v
al_loss: 0.6459 - val_accuracy: 0.6050
Epoch 15/50
4/4 [==============================] - 0s 79ms/step - loss: 0.6242 - accuracy: 0.6460 - v
al_loss: 0.6505 - val_accuracy: 0.6187
Epoch 16/50
4/4 [==============================] - 0s 75ms/step - loss: 0.6250 - accuracy: 0.6470 - v
al_loss: 0.6418 - val_accuracy: 0.6187
Epoch 17/50
4/4 [==============================] - 0s 70ms/step - loss: 0.6209 - accuracy: 0.6530 - v
al_loss: 0.6412 - val_accuracy: 0.6300
Epoch 18/50
4/4 [==============================] - 0s 77ms/step - loss: 0.6173 - accuracy: 0.6550 - v
al_loss: 0.6398 - val_accuracy: 0.6313
Epoch 19/50
4/4 [==============================] - 0s 65ms/step - loss: 0.6121 - accuracy: 0.6660 - v
al_loss: 0.6380 - val_accuracy: 0.6275
Epoch 20/50
4/4 [==============================] - 0s 62ms/step - loss: 0.6085 - accuracy: 0.6740 - v
al_loss: 0.6367 - val_accuracy: 0.6331
Epoch 21/50
4/4 [==============================] - 0s 64ms/step - loss: 0.6063 - accuracy: 0.6790 - v
al_loss: 0.6375 - val_accuracy: 0.6363
Epoch 22/50
4/4 [==============================] - 0s 76ms/step - loss: 0.6071 - accuracy: 0.6720 - v
al_loss: 0.6351 - val_accuracy: 0.6263
Epoch 23/50
4/4 [==============================] - 0s 76ms/step - loss: 0.6045 - accuracy: 0.6800 - v
al_loss: 0.6340 - val_accuracy: 0.6406
Epoch 24/50
4/4 [==============================] - 0s 63ms/step - loss: 0.5995 - accuracy: 0.6980 - v
al_loss: 0.6329 - val_accuracy: 0.6425
Epoch 25/50
4/4 [==============================] - 0s 66ms/step - loss: 0.5935 - accuracy: 0.6970 - v
al_loss: 0.6326 - val_accuracy: 0.6225
Epoch 26/50
4/4 [==============================] - 0s 78ms/step - loss: 0.5948 - accuracy: 0.6850 - v
al_loss: 0.6324 - val_accuracy: 0.6456
Epoch 27/50
4/4 [==============================] - 0s 76ms/step - loss: 0.5882 - accuracy: 0.6950 - v
al_loss: 0.6304 - val_accuracy: 0.6244
Epoch 28/50
4/4 [==============================] - 0s 63ms/step - loss: 0.5874 - accuracy: 0.7030 - v
al_loss: 0.6296 - val_accuracy: 0.6469
Epoch 29/50
4/4 [==============================] - 0s 72ms/step - loss: 0.5820 - accuracy: 0.7160 - v
al_loss: 0.6285 - val_accuracy: 0.6288
Epoch 30/50
4/4 [==============================] - 0s 77ms/step - loss: 0.5834 - accuracy: 0.6930 - v
al_loss: 0.6272 - val_accuracy: 0.6506
Epoch 31/50
4/4 [==============================] - 0s 61ms/step - loss: 0.5811 - accuracy: 0.7080 - v
al_loss: 0.6265 - val_accuracy: 0.6519
Epoch 32/50
4/4 [==============================] - 0s 75ms/step - loss: 0.5739 - accuracy: 0.7120 - v
al_loss: 0.6273 - val_accuracy: 0.6481
Epoch 33/50
4/4 [==============================] - 0s 77ms/step - loss: 0.5705 - accuracy: 0.7220 - v
al_loss: 0.6264 - val_accuracy: 0.6300
Epoch 34/50
4/4 [==============================] - 0s 69ms/step - loss: 0.5694 - accuracy: 0.7180 - v
al_loss: 0.6278 - val_accuracy: 0.6531
Epoch 35/50
4/4 [==============================] - 0s 75ms/step - loss: 0.5659 - accuracy: 0.7160 - v
al_loss: 0.6246 - val_accuracy: 0.6363
Epoch 36/50
4/4 [==============================] - 0s 78ms/step - loss: 0.5651 - accuracy: 0.7200 - v
al_loss: 0.6260 - val_accuracy: 0.6550
Epoch 37/50
4/4 [==============================] - 0s 64ms/step - loss: 0.5663 - accuracy: 0.7100
```

```
4/4 [==============================] - 0s 64ms/step - loss: 0.5663 - accuracy: 0.7190 - v
al_loss: 0.6266 - val_accuracy: 0.6256
Epoch 38/50
4/4 [==============================] - 0s 66ms/step - loss: 0.5677 - accuracy: 0.7100 - v
al_loss: 0.6296 - val_accuracy: 0.6513
Epoch 39/50
4/4 [==============================] - 0s 78ms/step - loss: 0.5740 - accuracy: 0.7000 - v
al_loss: 0.6273 - val_accuracy: 0.6281
Epoch 40/50
4/4 [==============================] - 0s 79ms/step - loss: 0.5714 - accuracy: 0.6940 - v
al_loss: 0.6233 - val_accuracy: 0.6531
Epoch 41/50
4/4 [==============================] - 0s 137ms/step - loss: 0.5683 - accuracy: 0.7100 -
val_loss: 0.6212 - val_accuracy: 0.6475
Epoch 42/50
4/4 [==============================] - 0s 140ms/step - loss: 0.5538 - accuracy: 0.7380 -
val_loss: 0.6207 - val_accuracy: 0.6475
Epoch 43/50
4/4 [==============================] - 0s 96ms/step - loss: 0.5463 - accuracy: 0.7430 - v
al_loss: 0.6218 - val_accuracy: 0.6519
Epoch 44/50
4/4 [==============================] - 0s 83ms/step - loss: 0.5452 - accuracy: 0.7430 - v
al_loss: 0.6217 - val_accuracy: 0.6388
Epoch 45/50
4/4 [==============================] - 0s 86ms/step - loss: 0.5434 - accuracy: 0.7410 - v
al_loss: 0.6224 - val_accuracy: 0.6500
Epoch 46/50
4/4 [==============================] - 0s 68ms/step - loss: 0.5486 - accuracy: 0.7300 - v
al_loss: 0.6209 - val_accuracy: 0.6506
Epoch 47/50
4/4 [==============================] - 0s 75ms/step - loss: 0.5448 - accuracy: 0.7320 - v
al_loss: 0.6210 - val_accuracy: 0.6350
Epoch 48/50
4/4 [==============================] - 0s 75ms/step - loss: 0.5435 - accuracy: 0.7450 - v
al_loss: 0.6227 - val_accuracy: 0.6575
Epoch 49/50
4/4 [==============================] - 0s 80ms/step - loss: 0.5425 - accuracy: 0.7330 - v
al_loss: 0.6235 - val_accuracy: 0.6294
Epoch 50/50
4/4 [==============================] - 0s 61ms/step - loss: 0.5344 - accuracy: 0.7510 - v
al_loss: 0.6240 - val_accuracy: 0.6619
```

In [18]:

```
model_2 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), input_shape=(96, 96, 3), acti
vation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model_2.compile(
    loss=tf.keras.losses.binary_crossentropy,
    optimizer=tf.keras.optimizers.Adam(),
    metrics=[tf.keras.metrics.BinaryAccuracy(name='accuracy')]
)

history_2 = model_2.fit(
    train_images,
    train_labels,
    epochs=15,
    batch_size=256,
    validation_data=(test_images, test_labels)
)
```

```
Epoch 1/15
```

```
4/4 [==============================] - 8s 407ms/step - loss: 0.7850 - accuracy: 0.5020 -
val_loss: 0.7433 - val_accuracy: 0.5000
Epoch 2/15
4/4 [==============================] - 0s 88ms/step - loss: 0.7087 - accuracy: 0.4990 - v
al_loss: 0.6903 - val_accuracy: 0.5319
Epoch 3/15
4/4 [==============================] - 0s 98ms/step - loss: 0.6897 - accuracy: 0.5210 - v
al_loss: 0.6912 - val_accuracy: 0.5000
Epoch 4/15
4/4 [==============================] - 0s 149ms/step - loss: 0.6881 - accuracy: 0.5000 -
val_loss: 0.6858 - val_accuracy: 0.5019
Epoch 5/15
4/4 [==============================] - 0s 109ms/step - loss: 0.6784 - accuracy: 0.5460 -
val_loss: 0.6733 - val_accuracy: 0.5369
Epoch 6/15
4/4 [==============================] - 0s 98ms/step - loss: 0.6548 - accuracy: 0.5880 - v
al_loss: 0.6430 - val_accuracy: 0.6506
Epoch 7/15
4/4 [==============================] - 0s 100ms/step - loss: 0.6168 - accuracy: 0.6870 -
val_loss: 0.6283 - val_accuracy: 0.6225
Epoch 8/15
4/4 [==============================] - 0s 107ms/step - loss: 0.5894 - accuracy: 0.6830 -
val_loss: 0.5858 - val_accuracy: 0.6938
Epoch 9/15
4/4 [==============================] - 0s 93ms/step - loss: 0.5572 - accuracy: 0.7100 - v
al_loss: 0.5859 - val_accuracy: 0.6794
Epoch 10/15
4/4 [==============================] - 0s 79ms/step - loss: 0.5329 - accuracy: 0.7270 - v
al_loss: 0.5525 - val_accuracy: 0.7244
Epoch 11/15
4/4 [==============================] - 0s 89ms/step - loss: 0.5028 - accuracy: 0.7660 - v
al_loss: 0.5462 - val_accuracy: 0.7306
Epoch 12/15
4/4 [==============================] - 0s 83ms/step - loss: 0.4710 - accuracy: 0.7840 - v
al_loss: 0.5364 - val_accuracy: 0.7337
Epoch 13/15
4/4 [==============================] - 0s 88ms/step - loss: 0.4531 - accuracy: 0.7890 - v
al_loss: 0.5105 - val_accuracy: 0.7456
Epoch 14/15
4/4 [==============================] - 0s 87ms/step - loss: 0.4175 - accuracy: 0.8120 - v
al_loss: 0.5093 - val_accuracy: 0.7487
Epoch 15/15
4/4 [==============================] - 0s 84ms/step - loss: 0.4002 - accuracy: 0.8140 - v
al_loss: 0.5624 - val_accuracy: 0.7262
```

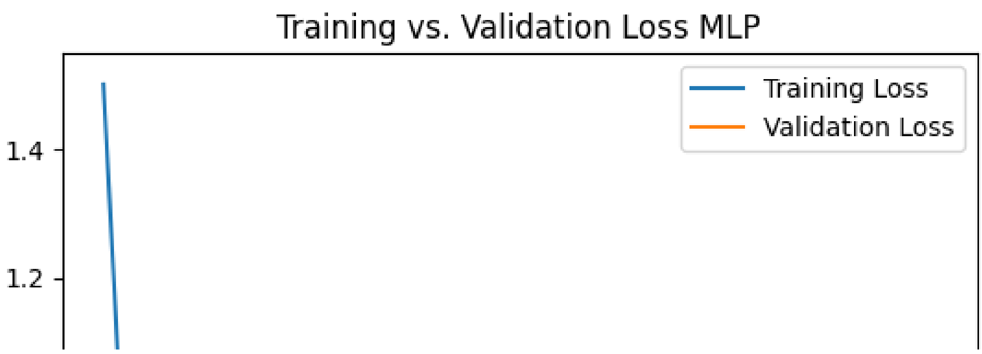
1. **Постройте кривые обучения нейронных сетей бинарной классификации для показателей ошибки и доли верных ответов в зависимости от эпохи обучения, подписывая оси и рисунок и создавая легенду.**

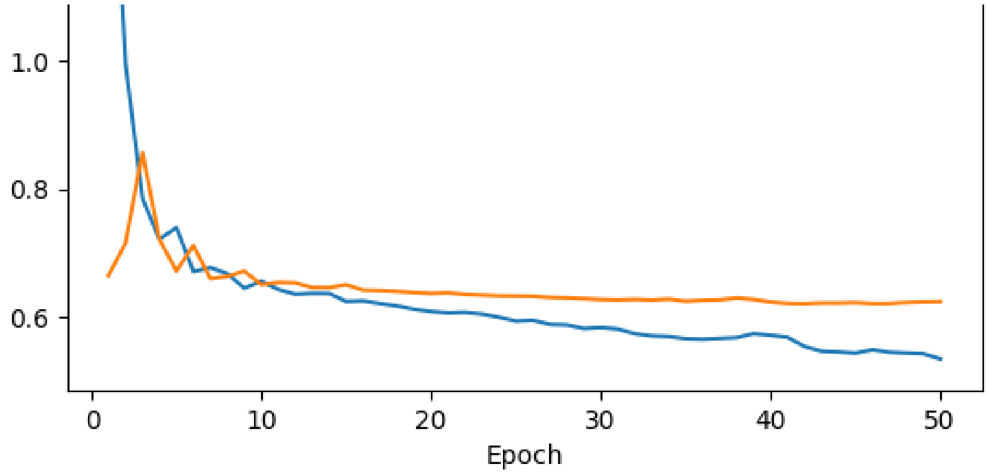
In [20]:

```
plt.plot(np.arange(1, 51), history_1.history['loss'], label='Training Loss')
plt.plot(np.arange(1, 51), history_1.history['val_loss'], label='Validation Loss')
plt.title('Training vs. Validation Loss MLP')
plt.xlabel('Epoch')
plt.legend()
```

Out[20]:

```
<matplotlib.legend.Legend at 0x7f39c8e48160>
```



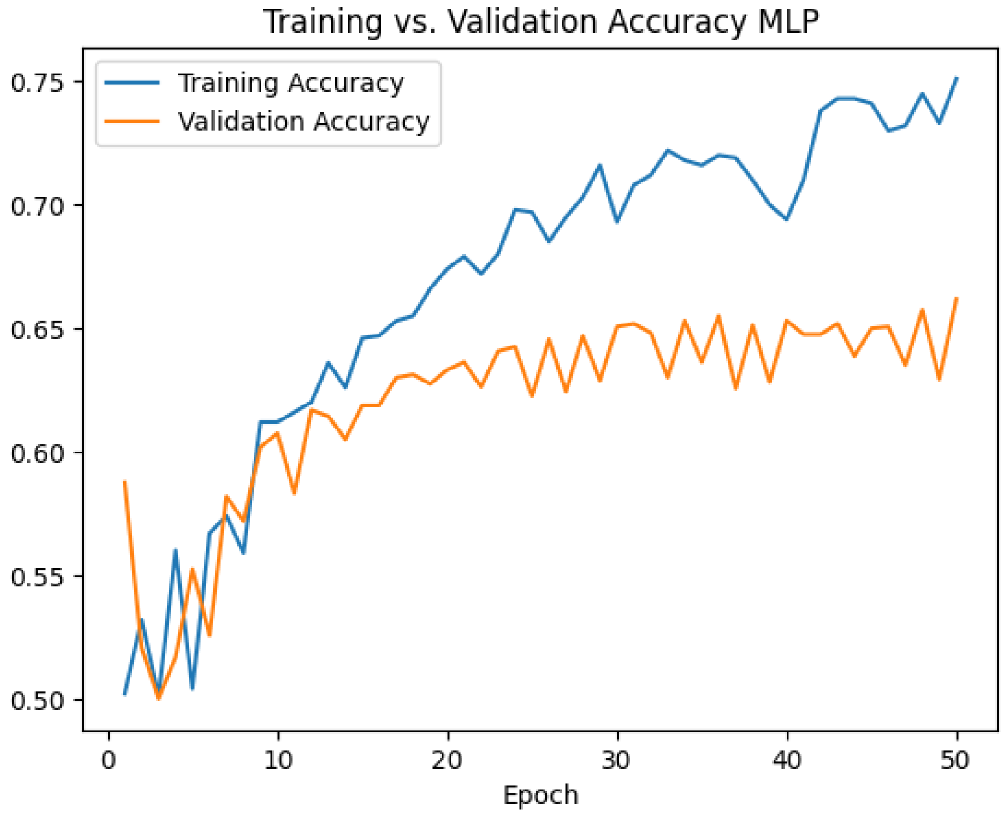


In [101]:

```
plt.plot(np.arange(1, 51), history_1.history['accuracy'], label='Training Accuracy')
plt.plot(np.arange(1, 51), history_1.history['val_accuracy'], label='Validation Accuracy
')
plt.title('Training vs. Validation Accuracy MLP')
plt.xlabel('Epoch')
plt.legend()
```

Out[101]:

```
<matplotlib.legend.Legend at 0x7f39ab279e40>
```

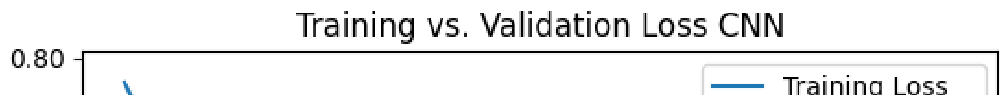


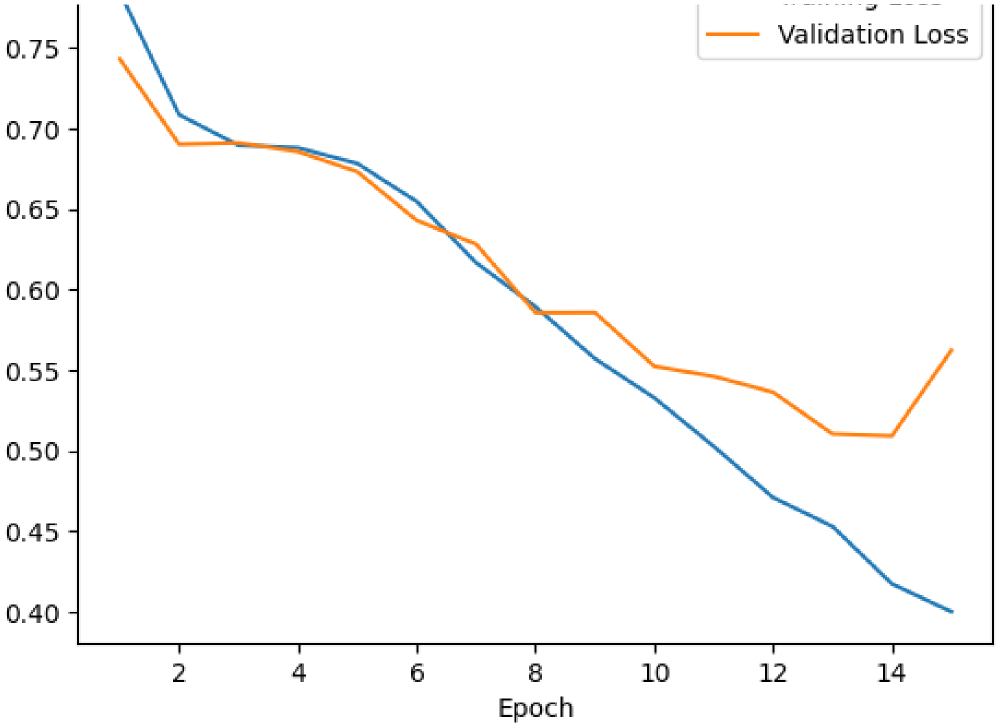
In [99]:

```
plt.plot(np.arange(1, 16), history_2.history['loss'], label='Training Loss')
plt.plot(np.arange(1, 16), history_2.history['val_loss'], label='Validation Loss')
plt.title('Training vs. Validation Loss CNN')
plt.xlabel('Epoch')
plt.legend()
```

Out[99]:

```
<matplotlib.legend.Legend at 0x7f39ab35ae30>
```



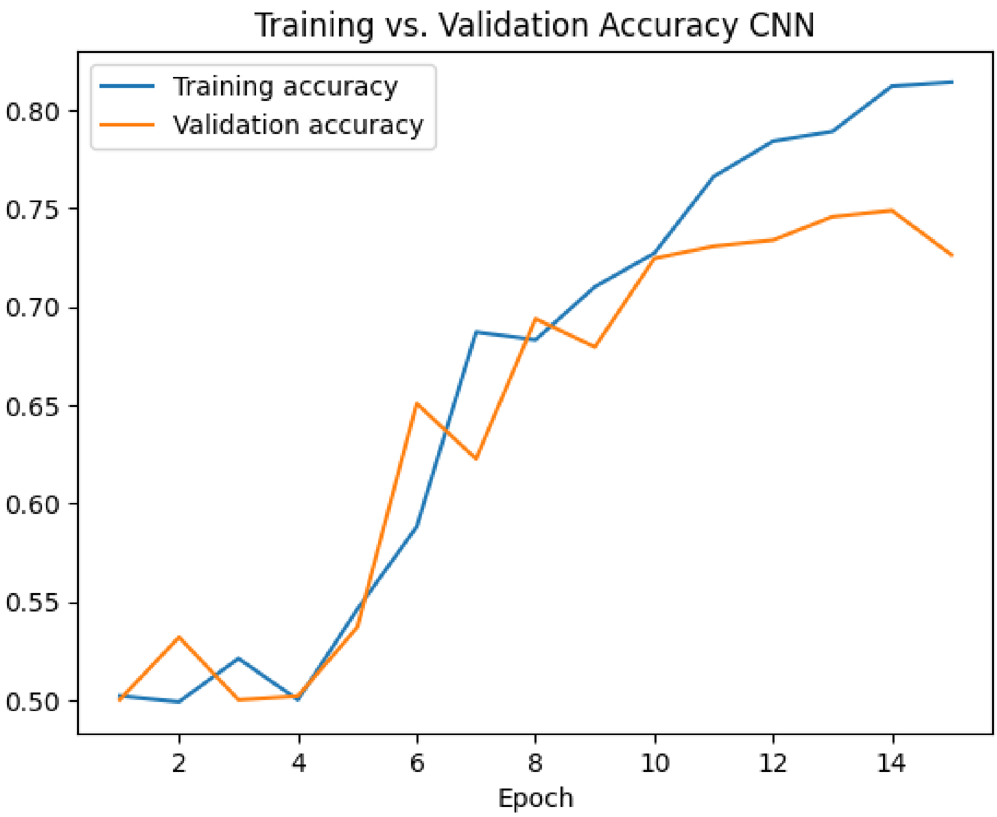


In [100]:

```
plt.plot(np.arange(1, 16), history_2.history['accuracy'], label='Training accuracy')
plt.plot(np.arange(1, 16), history_2.history['val_accuracy'], label='Validation accuracy
')
plt.title('Training vs. Validation Accuracy CNN')
plt.xlabel('Epoch')
plt.legend()
```

Out[100]:

<matplotlib.legend.Legend at 0x7f39ab333f10>



1. Сравните качество бинарной классификации нейронными сетями при помощи матрицы ошибок для тестовой выборки.

In [23]:

```
from sklearn.metrics import confusion_matrix as cm
```

In [29]:

```
y_pred = model_1.predict(test_images)
cm(test_labels, y_pred.round(0))
```

```
50/50 [==============================] - 0s 6ms/step
```

Out[29]:

```
array([[604, 196],
       [345, 455]])
```

In [30]:

```
y_pred = model_2.predict(test_images)
cm(test_labels, y_pred.round(0))
```

```
50/50 [==============================] - 1s 6ms/step
```

Out[30]:

```
array([[762,  38],
       [400, 400]])
```

1. Визуализируйте **ROC**-кривые для построенных классификаторов на одном рисунке **(**с легендой**)** и вычислите площади под **ROC**-кривыми**.**

In [ ]:

```

```

In [31]:

```
def true_false_positive(threshold_vector, y_test):
    true_positive = np.equal(threshold_vector, 1) & np.equal(y_test, 1)
    true_negative = np.equal(threshold_vector, 0) & np.equal(y_test, 0)
    false_positive = np.equal(threshold_vector, 1) & np.equal(y_test, 0)
    false_negative = np.equal(threshold_vector, 0) & np.equal(y_test, 1)

    tpr = true_positive.sum() / (true_positive.sum() + false_negative.sum())
    fpr = false_positive.sum() / (false_positive.sum() + true_negative.sum())

    return tpr, fpr
```

In [32]:

```
def roc_from_scratch(probabilities, y_test, partitions=100):
    roc = np.array([])
    for i in range(partitions + 1):

        threshold_vector = np.greater_equal(probabilities, i / partitions).astype(int)
        tpr, fpr = true_false_positive(threshold_vector, y_test)
        roc = np.append(roc, [fpr, tpr])

    return roc.reshape(-1, 2)
```

In [34]:

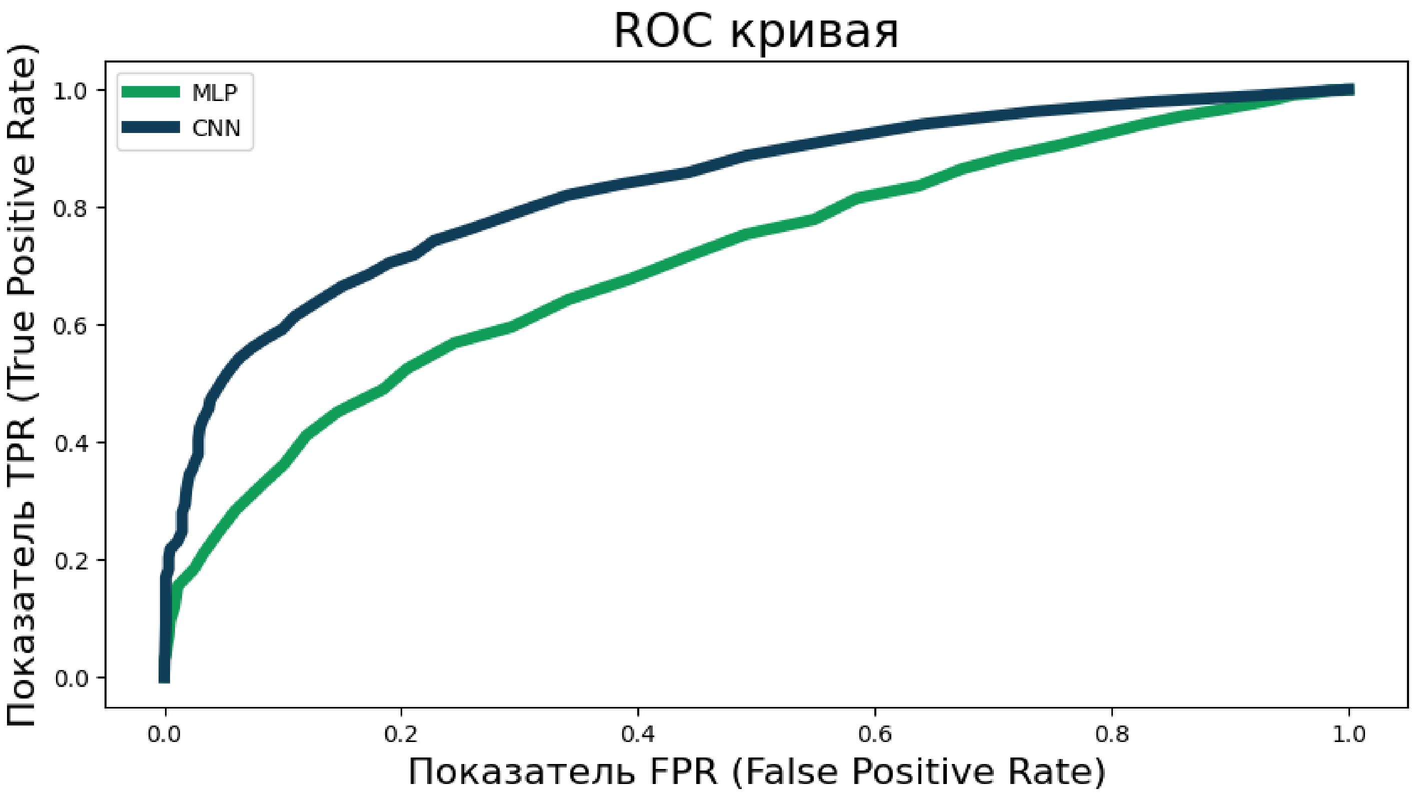
```
prediction1 = model_1.predict(test_images)
prediction2 = model_2.predict(test_images)

plt.figure(figsize=(10,5))

ROC1 = roc_from_scratch(prediction1.reshape(-1), test_labels,partitions=50)
ROC2 = roc_from_scratch(prediction2.reshape(-1), test_labels,partitions=50)
#plt.scatter(ROC[:,0],ROC[:,1],color='#0F9D58',s=100)
plt.plot(ROC1[:,0],ROC1[:,1],color='#0F9D58',lw=5, label='MLP')
plt.plot(ROC2[:,0],ROC2[:,1],color='#0F3D58',lw=5, label='CNN')
```

```
plt.title('ROC кривая',fontsize=20)
plt.xlabel('Показатель FPR (False Positive Rate)',fontsize=16)
plt.ylabel('Показатель TPR (True Positive Rate)',fontsize=16);
plt.legend()
```

```
50/50 [==============================] - 0s 3ms/step
50/50 [==============================] - 0s 4ms/step
```

Out[34]:

```
<matplotlib.legend.Legend at 0x7f39e83e2bc0>
```



In [35]:

```
def rocauc(x, y):
  yroc = y[np.argsort(x)]
  xroc = np.sort(x)
  sum = 0
  cur = (0, 0)
  for xi, yi in zip(xroc, yroc):
    sum += (xi -cur[0]) * (cur[1] + yi) / 2
    cur = (xi, yi)
  print('AUC ROC = ', sum)
```

In [36]:

```
print('ROC AUC MLP: ', end='')
rocauc(ROC1[:,0], ROC1[:,1])
print('ROC AUC CNN: ', end='')
rocauc(ROC2[:,0], ROC2[:,1])
```

```
ROC AUC MLP: AUC ROC =  0.7101148437500001
ROC AUC CNN: AUC ROC =  0.8348796875000001
```

1. Оставьте в наборе изображения трех классов, указанных в индивидуальном задании. Обучите нейронные сети **MLP** и **CNN** задаче многоклассовой классификации изображений (архитектура сетей по вашему усмотрению).

In [37]:

```
df_train1 = df_train[df_train['label'].isin([1, 3, 5])]
df_test1 = df_test[df_test['label'].isin([1, 3, 5])]
```

```
df_train1.shape, df_test1.shape
```

Out[37]:

```
((1500, 2), (2400, 2))
```

In [38]:

```
train_labels = df_train1['label'].to_numpy(dtype=np.float32)
test_labels = df_test1['label'].to_numpy(dtype=np.float32)

train_images = np.zeros(shape=(df_train1.shape[0],96,96,3), dtype=np.float32)
test_images  = np.zeros(shape=(df_test1.shape[0],96,96,3), dtype=np.float32)

for idx in range(train_labels.shape[0]):
    train_images[idx,:,:,:] = np.array(Image.fromarray(df_train1.iloc[idx]['image']))

for idx in range(test_labels.shape[0]):
    test_images[idx,:,:,:] = np.array(Image.fromarray(df_test1.iloc[idx]['image']))

train_images /= 255
test_images /= 255

train_images.shape, test_images.shape
```

Out[38]:

```
((1500, 96, 96, 3), (2400, 96, 96, 3))
```

In [41]:

```
from sklearn.preprocessing import OneHotEncoder as ohe
```

In [59]:

```
enc = ohe()
y_train = enc.fit_transform(train_labels.reshape(-1, 1)).toarray()
y_test = enc.transform(test_labels.reshape(-1, 1)).toarray()
y_train.dtype
```

Out[59]:

```
dtype('float64')
```

In [60]:

```
tf.random.set_seed(42)

model_1 = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(96, 96, 3)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])

model_1.compile(
    loss='categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    metrics=['accuracy']
)

history_3 = model_1.fit(
    train_images,
    y_train,
    epochs=50,
    batch_size=256,
    validation_data=(test_images, y_test)
)
```

```
Epoch 1/50
6/6 [==============================] - 2s 213ms/step - loss: 1.4619 - accuracy: 0.3293 -
val_loss: 1.2273 - val_accuracy: 0.3338
```

```
Epoch 2/50
6/6 [==============================] - 0s 86ms/step - loss: 1.1433 - accuracy: 0.3487 - v
al_loss: 1.1109 - val_accuracy: 0.3542
Epoch 3/50
6/6 [==============================] - 0s 58ms/step - loss: 1.1047 - accuracy: 0.3493 - v
al_loss: 1.0875 - val_accuracy: 0.3629
Epoch 4/50
6/6 [==============================] - 0s 57ms/step - loss: 1.0824 - accuracy: 0.3767 - v
al_loss: 1.0759 - val_accuracy: 0.4133
Epoch 5/50
6/6 [==============================] - 0s 85ms/step - loss: 1.0677 - accuracy: 0.4187 - v
al_loss: 1.0705 - val_accuracy: 0.4325
Epoch 6/50
6/6 [==============================] - 0s 88ms/step - loss: 1.0612 - accuracy: 0.4333 - v
al_loss: 1.0660 - val_accuracy: 0.4200
Epoch 7/50
6/6 [==============================] - 0s 58ms/step - loss: 1.0540 - accuracy: 0.4340 - v
al_loss: 1.0626 - val_accuracy: 0.4308
Epoch 8/50
6/6 [==============================] - 0s 63ms/step - loss: 1.0467 - accuracy: 0.4500 - v
al_loss: 1.0588 - val_accuracy: 0.4350
Epoch 9/50
6/6 [==============================] - 0s 57ms/step - loss: 1.0428 - accuracy: 0.4473 - v
al_loss: 1.0568 - val_accuracy: 0.4354
Epoch 10/50
6/6 [==============================] - 0s 85ms/step - loss: 1.0354 - accuracy: 0.4647 - v
al_loss: 1.0660 - val_accuracy: 0.4254
Epoch 11/50
6/6 [==============================] - 0s 59ms/step - loss: 1.0350 - accuracy: 0.4647 - v
al_loss: 1.0509 - val_accuracy: 0.4363
Epoch 12/50
6/6 [==============================] - 0s 59ms/step - loss: 1.0187 - accuracy: 0.4800 - v
al_loss: 1.0505 - val_accuracy: 0.4304
Epoch 13/50
6/6 [==============================] - 0s 85ms/step - loss: 1.0078 - accuracy: 0.5187 - v
al_loss: 1.0416 - val_accuracy: 0.4446
Epoch 14/50
6/6 [==============================] - 0s 60ms/step - loss: 0.9962 - accuracy: 0.5127 - v
al_loss: 1.0627 - val_accuracy: 0.4379
Epoch 15/50
6/6 [==============================] - 0s 84ms/step - loss: 1.0040 - accuracy: 0.4853 - v
al_loss: 1.0436 - val_accuracy: 0.4329
Epoch 16/50
6/6 [==============================] - 0s 57ms/step - loss: 0.9852 - accuracy: 0.5407 - v
al_loss: 1.0519 - val_accuracy: 0.4538
Epoch 17/50
6/6 [==============================] - 0s 91ms/step - loss: 0.9799 - accuracy: 0.5233 - v
al_loss: 1.0351 - val_accuracy: 0.4487
Epoch 18/50
6/6 [==============================] - 0s 54ms/step - loss: 0.9706 - accuracy: 0.5413 - v
al_loss: 1.0451 - val_accuracy: 0.4467
Epoch 19/50
6/6 [==============================] - 0s 57ms/step - loss: 0.9619 - accuracy: 0.5513 - v
al_loss: 1.0359 - val_accuracy: 0.4558
Epoch 20/50
6/6 [==============================] - 0s 58ms/step - loss: 0.9564 - accuracy: 0.5407 - v
al_loss: 1.0364 - val_accuracy: 0.4579
Epoch 21/50
6/6 [==============================] - 0s 73ms/step - loss: 0.9428 - accuracy: 0.5553 - v
al_loss: 1.0338 - val_accuracy: 0.4512
Epoch 22/50
6/6 [==============================] - 1s 96ms/step - loss: 0.9348 - accuracy: 0.5767 - v
al_loss: 1.0248 - val_accuracy: 0.4579
Epoch 23/50
6/6 [==============================] - 0s 79ms/step - loss: 0.9257 - accuracy: 0.5947 - v
al_loss: 1.0264 - val_accuracy: 0.4579
Epoch 24/50
6/6 [==============================] - 1s 95ms/step - loss: 0.9248 - accuracy: 0.5813 - v
al_loss: 1.0283 - val_accuracy: 0.4567
Epoch 25/50
6/6 [==============================] - 0s 89ms/step - loss: 0.9287 - accuracy: 0.5600 - v
al_loss: 1.0339 - val_accuracy: 0.4642
```

```
Epoch 26/50
6/6 [==============================] - 0s 66ms/step - loss: 0.9181 - accuracy: 0.5820 - v
al_loss: 1.0238 - val_accuracy: 0.4512
Epoch 27/50
6/6 [==============================] - 0s 86ms/step - loss: 0.8995 - accuracy: 0.5927 - v
al_loss: 1.0208 - val_accuracy: 0.4571
Epoch 28/50
6/6 [==============================] - 0s 84ms/step - loss: 0.8965 - accuracy: 0.6007 - v
al_loss: 1.0702 - val_accuracy: 0.4308
Epoch 29/50
6/6 [==============================] - 0s 84ms/step - loss: 0.9215 - accuracy: 0.5587 - v
al_loss: 1.0667 - val_accuracy: 0.4317
Epoch 30/50
6/6 [==============================] - 0s 90ms/step - loss: 0.9187 - accuracy: 0.5620 - v
al_loss: 1.0226 - val_accuracy: 0.4604
Epoch 31/50
6/6 [==============================] - 0s 59ms/step - loss: 0.8839 - accuracy: 0.6073 - v
al_loss: 1.0398 - val_accuracy: 0.4512
Epoch 32/50
6/6 [==============================] - 0s 87ms/step - loss: 0.8823 - accuracy: 0.6240 - v
al_loss: 1.0176 - val_accuracy: 0.4592
Epoch 33/50
6/6 [==============================] - 0s 57ms/step - loss: 0.8770 - accuracy: 0.6193 - v
al_loss: 1.0401 - val_accuracy: 0.4533
Epoch 34/50
6/6 [==============================] - 0s 86ms/step - loss: 0.8689 - accuracy: 0.6160 - v
al_loss: 1.0203 - val_accuracy: 0.4571
Epoch 35/50
6/6 [==============================] - 0s 87ms/step - loss: 0.8551 - accuracy: 0.6373 - v
al_loss: 1.0376 - val_accuracy: 0.4487
Epoch 36/50
6/6 [==============================] - 0s 54ms/step - loss: 0.8570 - accuracy: 0.6407 - v
al_loss: 1.0286 - val_accuracy: 0.4554
Epoch 37/50
6/6 [==============================] - 0s 56ms/step - loss: 0.8577 - accuracy: 0.6207 - v
al_loss: 1.0213 - val_accuracy: 0.4633
Epoch 38/50
6/6 [==============================] - 0s 55ms/step - loss: 0.8506 - accuracy: 0.6373 - v
al_loss: 1.0303 - val_accuracy: 0.4558
Epoch 39/50
6/6 [==============================] - 0s 55ms/step - loss: 0.8409 - accuracy: 0.6580 - v
al_loss: 1.0374 - val_accuracy: 0.4529
Epoch 40/50
6/6 [==============================] - 0s 57ms/step - loss: 0.8415 - accuracy: 0.6520 - v
al_loss: 1.0416 - val_accuracy: 0.4525
Epoch 41/50
6/6 [==============================] - 0s 59ms/step - loss: 0.8324 - accuracy: 0.6653 - v
al_loss: 1.0223 - val_accuracy: 0.4700
Epoch 42/50
6/6 [==============================] - 0s 59ms/step - loss: 0.8189 - accuracy: 0.6787 - v
al_loss: 1.0192 - val_accuracy: 0.4646
Epoch 43/50
6/6 [==============================] - 0s 85ms/step - loss: 0.8114 - accuracy: 0.6760 - v
al_loss: 1.0335 - val_accuracy: 0.4600
Epoch 44/50
6/6 [==============================] - 0s 86ms/step - loss: 0.8054 - accuracy: 0.6927 - v
al_loss: 1.0351 - val_accuracy: 0.4583
Epoch 45/50
6/6 [==============================] - 0s 56ms/step - loss: 0.8329 - accuracy: 0.6313 - v
al_loss: 1.0406 - val_accuracy: 0.4617
Epoch 46/50
6/6 [==============================] - 0s 58ms/step - loss: 0.8371 - accuracy: 0.6167 - v
al_loss: 1.0208 - val_accuracy: 0.4654
Epoch 47/50
6/6 [==============================] - 0s 86ms/step - loss: 0.8183 - accuracy: 0.6547 - v
al_loss: 1.0363 - val_accuracy: 0.4638
Epoch 48/50
6/6 [==============================] - 0s 84ms/step - loss: 0.7979 - accuracy: 0.6633 - v
al_loss: 1.0567 - val_accuracy: 0.4583
Epoch 49/50
6/6 [==============================] - 0s 56ms/step - loss: 0.7987 - accuracy: 0.6613 - v
al_loss: 1.0472 - val_accuracy: 0.4588
```

```
Epoch 50/50
6/6 [==============================] - 0s 82ms/step - loss: 0.7816 - accuracy: 0.6893 - v
al_loss: 1.0269 - val_accuracy: 0.4708
```

In [70]:

```python
model_2 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), input_shape=(96, 96, 3), acti
vation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])

model_2.compile(
    loss='categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy']
)

history_4 = model_2.fit(
    train_images,
    y_train,
    epochs=30,
    batch_size=256,
    validation_data=(test_images, y_test))
```

```
Epoch 1/30
6/6 [==============================] - 3s 229ms/step - loss: 1.2882 - accuracy: 0.3533 -
val_loss: 1.1259 - val_accuracy: 0.3333
Epoch 2/30
6/6 [==============================] - 0s 81ms/step - loss: 1.0987 - accuracy: 0.3740 - v
al_loss: 1.0935 - val_accuracy: 0.4008
Epoch 3/30
6/6 [==============================] - 0s 74ms/step - loss: 1.0849 - accuracy: 0.3833 - v
al_loss: 1.0845 - val_accuracy: 0.3421
Epoch 4/30
6/6 [==============================] - 0s 81ms/step - loss: 1.0692 - accuracy: 0.3993 - v
al_loss: 1.0659 - val_accuracy: 0.4546
Epoch 5/30
6/6 [==============================] - 0s 73ms/step - loss: 1.0292 - accuracy: 0.5247 - v
al_loss: 1.0354 - val_accuracy: 0.4558
Epoch 6/30
6/6 [==============================] - 1s 99ms/step - loss: 0.9778 - accuracy: 0.5413 - v
al_loss: 0.9881 - val_accuracy: 0.5267
Epoch 7/30
6/6 [==============================] - 0s 91ms/step - loss: 0.9207 - accuracy: 0.5787 - v
al_loss: 0.9685 - val_accuracy: 0.5179
Epoch 8/30
6/6 [==============================] - 1s 109ms/step - loss: 0.8662 - accuracy: 0.6060 -
val_loss: 0.9399 - val_accuracy: 0.5604
Epoch 9/30
6/6 [==============================] - 0s 88ms/step - loss: 0.8126 - accuracy: 0.6347 - v
al_loss: 0.9241 - val_accuracy: 0.5654
Epoch 10/30
6/6 [==============================] - 1s 99ms/step - loss: 0.7588 - accuracy: 0.6840 - v
al_loss: 0.9518 - val_accuracy: 0.5558
Epoch 11/30
6/6 [==============================] - 1s 105ms/step - loss: 0.7195 - accuracy: 0.7033 -
val_loss: 0.9101 - val_accuracy: 0.5821
Epoch 12/30
6/6 [==============================] - 0s 70ms/step - loss: 0.6682 - accuracy: 0.7260 - v
al_loss: 0.9147 - val_accuracy: 0.5729
Epoch 13/30
6/6 [==============================] - 1s 98ms/step - loss: 0.6330 - accuracy: 0.7387 - v
al_loss: 0.9642 - val_accuracy: 0.5454
Epoch 14/30
```

```
6/6 [==============================] - 1s 97ms/step - loss: 0.5991 - accuracy: 0.7713 - v
al_loss: 0.9486 - val_accuracy: 0.5700
Epoch 15/30
6/6 [==============================] - 1s 99ms/step - loss: 0.5357 - accuracy: 0.8080 - v
al_loss: 0.9549 - val_accuracy: 0.5671
Epoch 16/30
6/6 [==============================] - 0s 72ms/step - loss: 0.5092 - accuracy: 0.8120 - v
al_loss: 0.9591 - val_accuracy: 0.5663
Epoch 17/30
6/6 [==============================] - 1s 97ms/step - loss: 0.4614 - accuracy: 0.8480 - v
al_loss: 0.9738 - val_accuracy: 0.5663
Epoch 18/30
6/6 [==============================] - 1s 99ms/step - loss: 0.4570 - accuracy: 0.8447 - v
al_loss: 0.9773 - val_accuracy: 0.5775
Epoch 19/30
6/6 [==============================] - 0s 74ms/step - loss: 0.4410 - accuracy: 0.8453 - v
al_loss: 1.0086 - val_accuracy: 0.5608
Epoch 20/30
6/6 [==============================] - 1s 98ms/step - loss: 0.3960 - accuracy: 0.8760 - v
al_loss: 1.0028 - val_accuracy: 0.5788
Epoch 21/30
6/6 [==============================] - 1s 99ms/step - loss: 0.3498 - accuracy: 0.9033 - v
al_loss: 1.1006 - val_accuracy: 0.5429
Epoch 22/30
6/6 [==============================] - 0s 71ms/step - loss: 0.3433 - accuracy: 0.8833 - v
al_loss: 1.0382 - val_accuracy: 0.5704
Epoch 23/30
6/6 [==============================] - 0s 74ms/step - loss: 0.3136 - accuracy: 0.9087 - v
al_loss: 1.0586 - val_accuracy: 0.5617
Epoch 24/30
6/6 [==============================] - 1s 98ms/step - loss: 0.2845 - accuracy: 0.9173 - v
al_loss: 1.1048 - val_accuracy: 0.5579
Epoch 25/30
6/6 [==============================] - 0s 75ms/step - loss: 0.2791 - accuracy: 0.9193 - v
al_loss: 1.1549 - val_accuracy: 0.5583
Epoch 26/30
6/6 [==============================] - 1s 98ms/step - loss: 0.2775 - accuracy: 0.9160 - v
al_loss: 1.1471 - val_accuracy: 0.5537
Epoch 27/30
6/6 [==============================] - 0s 79ms/step - loss: 0.2539 - accuracy: 0.9273 - v
al_loss: 1.1159 - val_accuracy: 0.5700
Epoch 28/30
6/6 [==============================] - 1s 101ms/step - loss: 0.2058 - accuracy: 0.9547 -
val_loss: 1.1865 - val_accuracy: 0.5537
Epoch 29/30
6/6 [==============================] - 0s 73ms/step - loss: 0.2046 - accuracy: 0.9453 - v
al_loss: 1.2818 - val_accuracy: 0.5433
Epoch 30/30
6/6 [==============================] - 0s 72ms/step - loss: 0.2113 - accuracy: 0.9373 - v
al_loss: 1.1888 - val_accuracy: 0.5596
```

1. **Сравните качество многоклассовой классификации нейронными сетями при помощи матрицы ошибок (для трех классов) для тестовой выборки.**

In [89]:

```
y_pred = model_1.predict(test_images)
pred=(np.argmax(y_pred, axis=1) + 1) * 2 - 1
```

```
75/75 [==============================] - 0s 3ms/step
```

In [90]:

```
cm(pred, test_labels)
```

Out[90]:

```
array([[427, 162, 247],
       [237, 449, 299],
       [136, 189, 254]])
```

In [91]:

```
y_pred = model_2.predict(test_images)
pred=(np.argmax(y_pred, axis=1) + 1) * 2 - 1
```

```
75/75 [==============================] - 0s 4ms/step
```

In [92]:

```
cm(pred, test_labels)
```

Out[92]:

```
array([[546, 185, 242],
       [144, 451, 212],
       [110, 164, 346]])
```

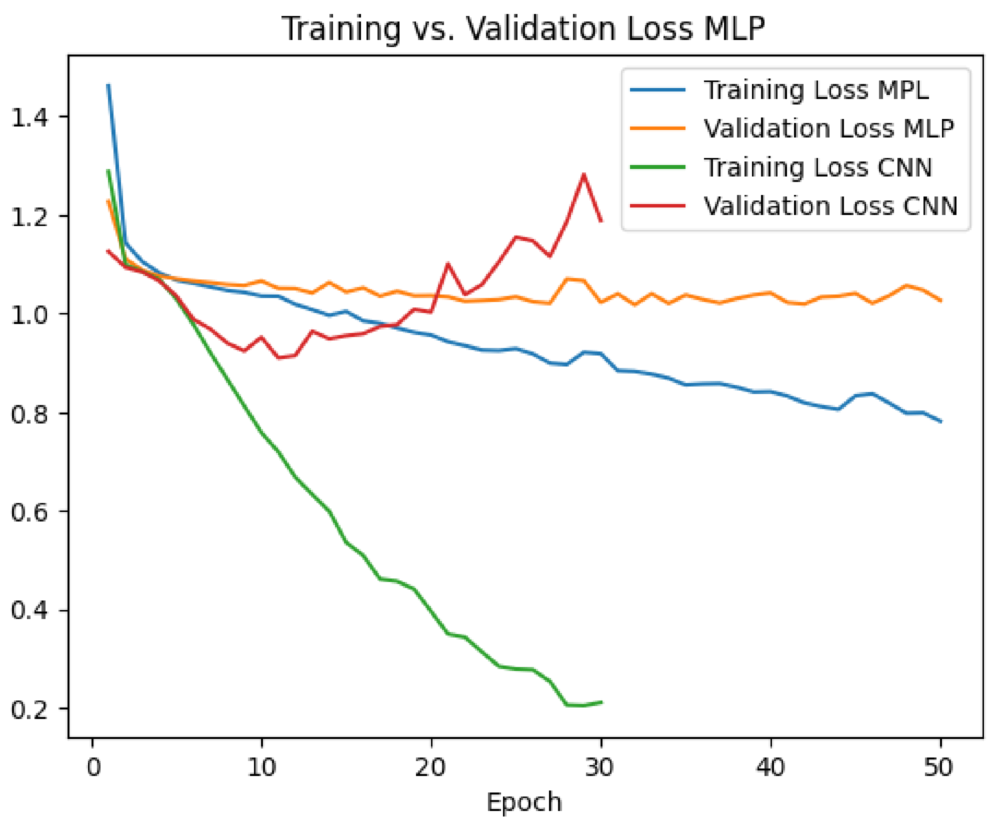
1. Постройте кривые обучения нейронных сетей многоклассовой классификации для показателей ошибки и доли верных ответов в зависимости от эпохи обучения, подписывая оси и рисунок и создавая легенду.

In [94]:

```
plt.plot(np.arange(1, 51), history_3.history['loss'], label='Training Loss MPL')
plt.plot(np.arange(1, 51), history_3.history['val_loss'], label='Validation Loss MLP')
plt.plot(np.arange(1, 31), history_4.history['loss'], label='Training Loss CNN')
plt.plot(np.arange(1, 31), history_4.history['val_loss'], label='Validation Loss CNN')
plt.title('Training vs. Validation Loss MLP')
plt.xlabel('Epoch')
plt.legend()
```

Out[94]:

```
<matplotlib.legend.Legend at 0x7f39b00b4760>
```



CNN очевидно переобучилась

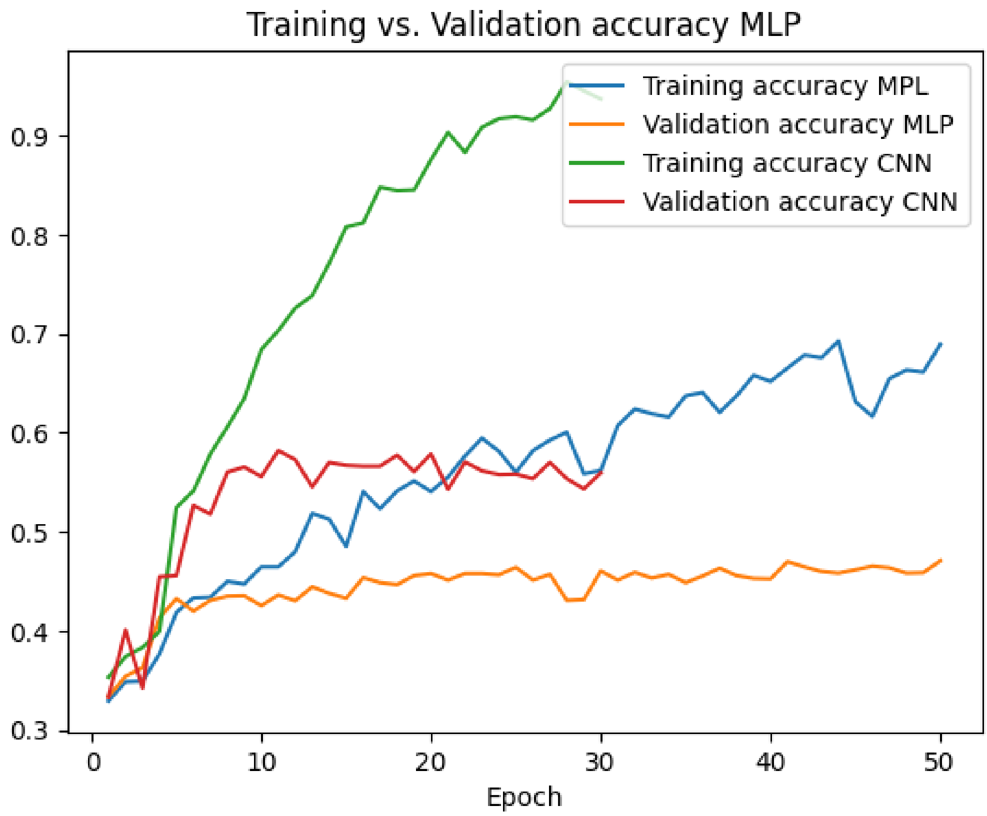
In [102]:

```
plt.plot(np.arange(1, 51), history_3.history['accuracy'], label='Training accuracy MPL')
plt.plot(np.arange(1, 51), history_3.history['val_accuracy'], label='Validation accuracy
MLP')
plt.plot(np.arange(1, 31), history_4.history['accuracy'], label='Training accuracy CNN')
```

```
plt.plot(np.arange(1, 31), history_4.history['val_accuracy'], label='Validation accuracy CNN')
plt.title('Training vs. Validation accuracy MLP')
plt.xlabel('Epoch')
plt.legend()
```

Out[102]:

```
<matplotlib.legend.Legend at 0x7f39ab101ba0>
```



In [ ]: