lab05

May 19, 2024

0.0.1 People's Friendship University in Russia

Faculty of Science

Department of Mathematical Modeling and Artificial Intelligence

0.1 Labratory work №5 report

0.1.1 Meathods of machine learning

Student: Abu Suveilim Mukhammed M.

Group: NKNbd-01-21

0.2 Moscow 2024

0.2.1 Version №11

Option 11

1. stl10 data set
2. Classes labeled 1,2,3
3. Requirements for MLP network architecture:

Number of hidden layers 5

The number of neurons is 50 in the first hidden layer, increasing by 10 with each subsequent hidden layer

Adagrad optimizer

Activation function in hidden layers swish

L2 regularization in every odd hidden layer

4. CNN network architecture requirements:

Number of convolutional layers 5

Number of filters in convolutional layers 16

Filter dimensions 2x2

RMSprop optimizer

Activation function in convolutional layers leaky_relu

Activation function in hidden dense layers relu

Dropout layers after each pooling layer

5. Binary classification quality indicator:

Foulkes–Mallows index equal to the square root of TP/(TP + TN) * TP/(TP + FP)

6. Quality indicator of multi-class classification:

maximum completeness of classes, where completeness (recall) of a class is equal to the proportion of correct predictions for all points belonging to this class.

0.3 1. Load the data set with images specified in the individual task from Tensorflow Datasets, divided into training and test samples.

[58]:
```
#load needed libraries and packags
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
import tensorflow_datasets as tfds
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from matplotlib import rcParams
from mpl_toolkits.mplot3d import Axes3D
from PIL import Image, ImageOps
from math import sqrt
```

[2]:
```
# loading stl10 dataset
ds = tfds.load("stl10", split=['train','test'])
print(ds)
```

```
[<_PrefetchDataset element_spec={'image': TensorSpec(shape=(96, 96, 3),
dtype=tf.uint8, name=None), 'label': TensorSpec(shape=(), dtype=tf.int64,
name=None)}>, <_PrefetchDataset element_spec={'image': TensorSpec(shape=(96, 96,
3), dtype=tf.uint8, name=None), 'label': TensorSpec(shape=(), dtype=tf.int64,
name=None)}>]
```

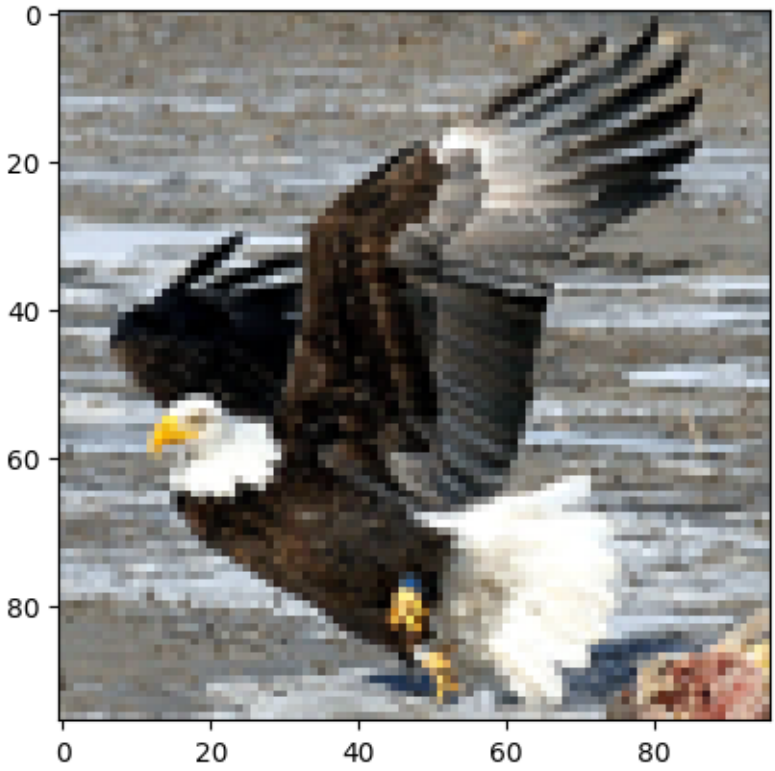
[3]:
```
df_train = tfds.as_dataframe(ds[0])
df_test  = tfds.as_dataframe(ds[1])
df_train.shape, df_test.shape
```

[3]:
```
((5000, 2), (8000, 2))
```

0.4 2. Render a few images randomly selected from the training set.

[4]:
```
img = Image.fromarray(df_train.iloc[0]['image'])
plt.imshow(img)
```

[4]:
```
<matplotlib.image.AxesImage at 0x1540fa5f2d0>
```



[5]:
```
np.array(img).shape
```

[5]:
```
(96, 96, 3)
```

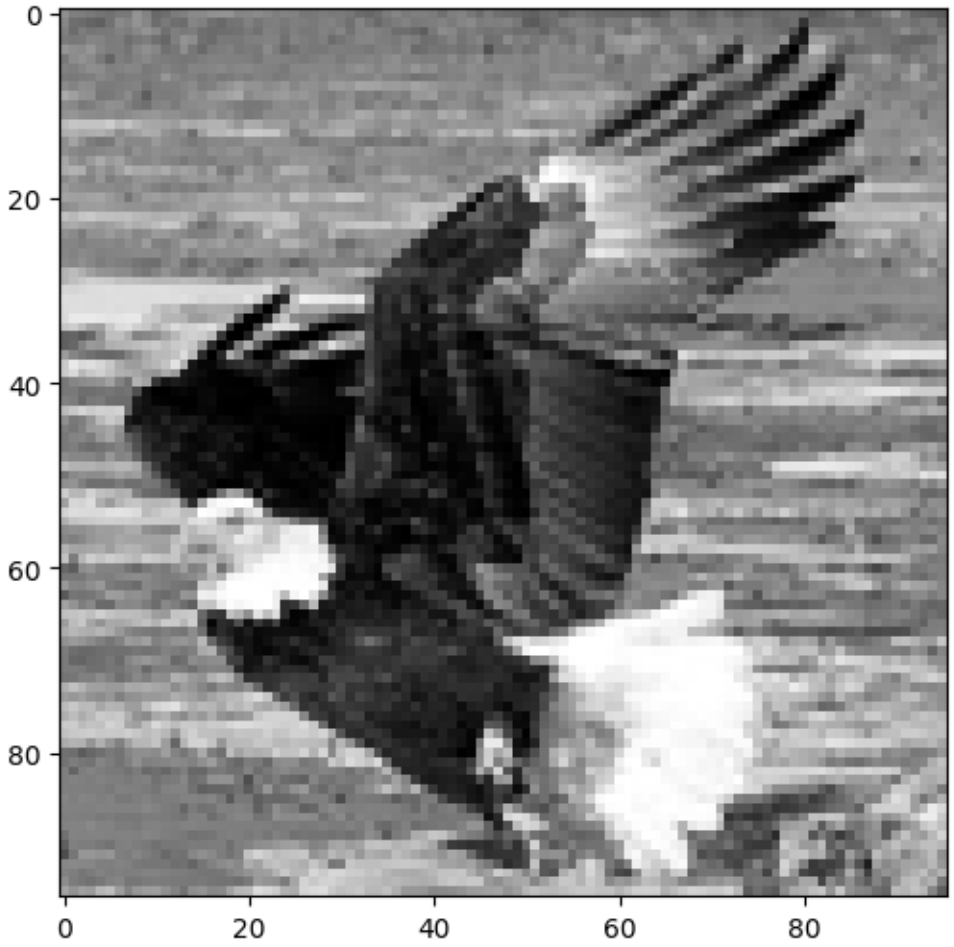
[6]:
```
img = ImageOps.grayscale(img)
np.array(img).shape
```

[6]:
```
(96, 96)
```

[7]:
```
def plot_image(img: np.array):
    plt.figure(figsize=(6, 6))
    plt.imshow(img, cmap='gray');

def plot_two_images(img1: np.array, img2: np.array):
    _, ax = plt.subplots(1, 2, figsize=(12, 6))
```

```
    ax[0].imshow(img1, cmap='gray')
    ax[1].imshow(img2, cmap='gray');
```

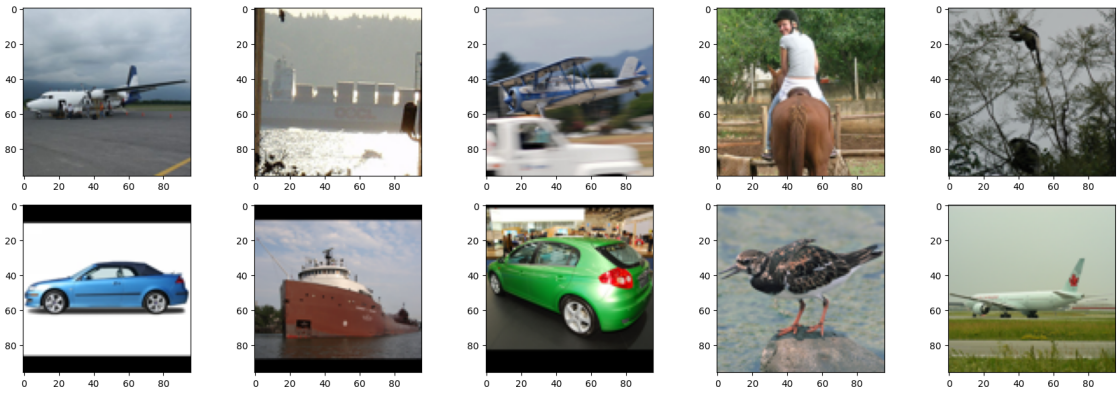
[8]:
```
plot_image(img=img)
```



[9]:
```
import random

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

    num_row = 2
    num_col = 5

    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()
```

[10]:
```
plot_random_sample(df_train['image'])
```



0.5 3. Leave in the set the images of the two classes listed first in the individual assignment. Train MLP and CNN neural networks for the task of binary image classification (network architecture requirements are specified in the individual assignment). Track the training of neural networks and indicate by what percentage the losses decreased as a result of training in relation to the losses in the first training epoch. Evaluate the results of training neural networks (options: neural network trained, undertrained, overtrained).

0.6 4. Plot learning curves for binary classification neural networks for loss rates and the proportion of correct answers depending on the training epoch, labeling the axes and figure and creating a legend.

[11]:
```
X = df_train[df_train['label'] == 1]
X['label'] = 0
y = df_train[df_train['label'] == 2]
y['label'] = 1
df_train_01 = pd.concat([X, y])
```

```
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\3770910647.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
```

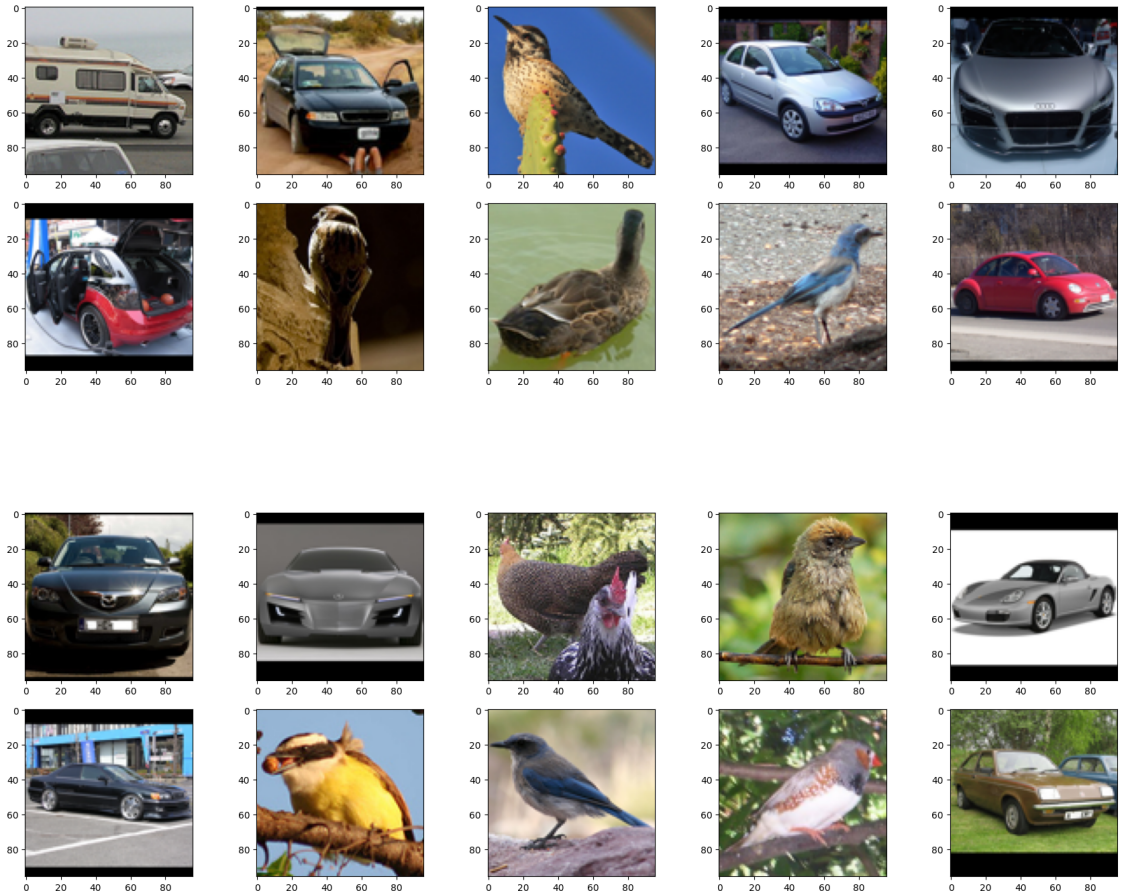
```
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  X['label'] = 0
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\3770910647.py:4:
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
  y['label'] = 1
```

[12]:
```
X = df_test[df_test['label'] == 1]
X['label'] = 0
y = df_test[df_test['label'] == 2]
y['label'] = 1
df_test_01 = pd.concat([X, y])
df_test_01['label'].value_counts()
```

```
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\3143479026.py:2:
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
  X['label'] = 0
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\3143479026.py:4:
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
  y['label'] = 1
```

[12]:
```
label
0    800
1    800
Name: count, dtype: int64
```

[16]:
```
#let's check if we have correctly chose the lable
plot_random_sample(df_train_01['image'])
plot_random_sample(df_test_01['image'])
```



Labels 1 & 2 are birds and cars

[17]:
```
train_labels = df_train_01['label'].to_numpy(dtype=np.float32)
test_labels = df_test_01['label'].to_numpy(dtype=np.float32)
train_labels.shape, test_labels.shape
```

[17]:
```
((1000,), (1600,))
```

[20]:
```
train_images = np.zeros(shape=(df_train_01.shape[0],96,96,3), dtype=np.float32)
test_images  = np.zeros(shape=(df_test_01.shape[0],96,96,3), dtype=np.float32)
train_images.shape, test_images.shape
```

[20]:
```
((1000, 96, 96, 3), (1600, 96, 96, 3))
```

[21]:
```
for idx in range(train_labels.shape[0]):
    train_images[idx,:,:,:] = \
        np.array(Image.fromarray(df_train_01.iloc[idx]['image']))

for idx in range(test_labels.shape[0]):
    test_images[idx,:,:,:] = \
```

```
        np.array(Image.fromarray(df_test_01.iloc[idx]['image']))

train_images.shape, test_images.shape
```

[21]: ((1000, 96, 96, 3), (1600, 96, 96, 3))

[22]:
```
#make them into one channel/one array
train_images /= 255
test_images  /= 255
```

0.7 Training the MLP neural network

Binary cross-entropy is used as the loss function.

[23]:
```
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(50, activation='swish', kernel_regularizer ='l2'), #1
    tf.keras.layers.Dense(60, activation='swish'),
    tf.keras.layers.Dense(70, activation='swish', kernel_regularizer ='l2'), #3
    tf.keras.layers.Dense(80, activation='swish'),
    tf.keras.layers.Dense(90, activation='swish', kernel_regularizer ='l2'), #5
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model_1.compile(
    loss=tf.keras.losses.binary_crossentropy,
    optimizer=tf.keras.optimizers.Adagrad(),
    metrics=[tf.keras.metrics.BinaryAccuracy(name='accuracy')]
)

history_1 = model_1.fit(
    train_images,
    train_labels,
    epochs=50,
    batch_size=128,
    validation_data=(test_images, test_labels)
)
```

```
Epoch 1/50
8/8             1s 53ms/step -
accuracy: 0.4540 - loss: 3.1864 - val_accuracy: 0.5831 - val_loss: 3.1829
Epoch 2/50
8/8             0s 11ms/step -
accuracy: 0.5952 - loss: 3.1817 - val_accuracy: 0.5888 - val_loss: 3.1780
Epoch 3/50
```

```
8/8            0s 10ms/step -
accuracy: 0.5955 - loss: 3.1762 - val_accuracy: 0.5337 - val_loss: 3.1721
Epoch 4/50
8/8            0s 10ms/step -
accuracy: 0.5465 - loss: 3.1696 - val_accuracy: 0.5050 - val_loss: 3.1650
Epoch 5/50
8/8            0s 10ms/step -
accuracy: 0.5227 - loss: 3.1616 - val_accuracy: 0.5013 - val_loss: 3.1568
Epoch 6/50
8/8            0s 10ms/step -
accuracy: 0.5165 - loss: 3.1526 - val_accuracy: 0.5013 - val_loss: 3.1483
Epoch 7/50
8/8            0s 10ms/step -
accuracy: 0.5165 - loss: 3.1434 - val_accuracy: 0.5013 - val_loss: 3.1401
Epoch 8/50
8/8            0s 10ms/step -
accuracy: 0.5165 - loss: 3.1344 - val_accuracy: 0.5019 - val_loss: 3.1323
Epoch 9/50
8/8            0s 10ms/step -
accuracy: 0.5169 - loss: 3.1257 - val_accuracy: 0.5050 - val_loss: 3.1243
Epoch 10/50
8/8            0s 10ms/step -
accuracy: 0.5247 - loss: 3.1169 - val_accuracy: 0.5150 - val_loss: 3.1161
Epoch 11/50
8/8            0s 10ms/step -
accuracy: 0.5457 - loss: 3.1076 - val_accuracy: 0.5412 - val_loss: 3.1072
Epoch 12/50
8/8            0s 10ms/step -
accuracy: 0.5776 - loss: 3.0975 - val_accuracy: 0.5987 - val_loss: 3.0975
Epoch 13/50
8/8            0s 10ms/step -
accuracy: 0.6333 - loss: 3.0862 - val_accuracy: 0.6744 - val_loss: 3.0865
Epoch 14/50
8/8            0s 10ms/step -
accuracy: 0.6915 - loss: 3.0734 - val_accuracy: 0.7194 - val_loss: 3.0738
Epoch 15/50
8/8            0s 10ms/step -
accuracy: 0.7515 - loss: 3.0585 - val_accuracy: 0.7713 - val_loss: 3.0589
Epoch 16/50
8/8            0s 10ms/step -
accuracy: 0.7845 - loss: 3.0409 - val_accuracy: 0.7962 - val_loss: 3.0415
Epoch 17/50
8/8            0s 10ms/step -
accuracy: 0.8042 - loss: 3.0205 - val_accuracy: 0.8112 - val_loss: 3.0217
Epoch 18/50
8/8            0s 10ms/step -
accuracy: 0.8263 - loss: 2.9973 - val_accuracy: 0.8250 - val_loss: 3.0002
Epoch 19/50
```

```
8/8            0s 10ms/step -
accuracy: 0.8422 - loss: 2.9722 - val_accuracy: 0.8263 - val_loss: 2.9780
Epoch 20/50
8/8            0s 10ms/step -
accuracy: 0.8451 - loss: 2.9465 - val_accuracy: 0.8294 - val_loss: 2.9564
Epoch 21/50
8/8            0s 11ms/step -
accuracy: 0.8459 - loss: 2.9216 - val_accuracy: 0.8306 - val_loss: 2.9366
Epoch 22/50
8/8            0s 11ms/step -
accuracy: 0.8466 - loss: 2.8987 - val_accuracy: 0.8269 - val_loss: 2.9190
Epoch 23/50
8/8            0s 10ms/step -
accuracy: 0.8518 - loss: 2.8784 - val_accuracy: 0.8213 - val_loss: 2.9036
Epoch 24/50
8/8            0s 10ms/step -
accuracy: 0.8558 - loss: 2.8607 - val_accuracy: 0.8213 - val_loss: 2.8903
Epoch 25/50
8/8            0s 10ms/step -
accuracy: 0.8593 - loss: 2.8453 - val_accuracy: 0.8250 - val_loss: 2.8786
Epoch 26/50
8/8            0s 13ms/step -
accuracy: 0.8653 - loss: 2.8319 - val_accuracy: 0.8275 - val_loss: 2.8683
Epoch 27/50
8/8            0s 10ms/step -
accuracy: 0.8658 - loss: 2.8201 - val_accuracy: 0.8281 - val_loss: 2.8590
Epoch 28/50
8/8            0s 10ms/step -
accuracy: 0.8661 - loss: 2.8095 - val_accuracy: 0.8306 - val_loss: 2.8505
Epoch 29/50
8/8            0s 10ms/step -
accuracy: 0.8663 - loss: 2.7998 - val_accuracy: 0.8325 - val_loss: 2.8426
Epoch 30/50
8/8            0s 10ms/step -
accuracy: 0.8670 - loss: 2.7910 - val_accuracy: 0.8344 - val_loss: 2.8352
Epoch 31/50
8/8            0s 11ms/step -
accuracy: 0.8688 - loss: 2.7828 - val_accuracy: 0.8350 - val_loss: 2.8283
Epoch 32/50
8/8            0s 10ms/step -
accuracy: 0.8716 - loss: 2.7751 - val_accuracy: 0.8363 - val_loss: 2.8216
Epoch 33/50
8/8            0s 10ms/step -
accuracy: 0.8726 - loss: 2.7678 - val_accuracy: 0.8369 - val_loss: 2.8153
Epoch 34/50
8/8            0s 10ms/step -
accuracy: 0.8762 - loss: 2.7609 - val_accuracy: 0.8381 - val_loss: 2.8092
Epoch 35/50
```

```
8/8            0s 10ms/step -
accuracy: 0.8766 - loss: 2.7543 - val_accuracy: 0.8406 - val_loss: 2.8033
Epoch 36/50
8/8            0s 10ms/step -
accuracy: 0.8760 - loss: 2.7479 - val_accuracy: 0.8400 - val_loss: 2.7976
Epoch 37/50
8/8            0s 11ms/step -
accuracy: 0.8793 - loss: 2.7417 - val_accuracy: 0.8413 - val_loss: 2.7921
Epoch 38/50
8/8            0s 10ms/step -
accuracy: 0.8804 - loss: 2.7358 - val_accuracy: 0.8450 - val_loss: 2.7868
Epoch 39/50
8/8            0s 10ms/step -
accuracy: 0.8856 - loss: 2.7300 - val_accuracy: 0.8469 - val_loss: 2.7816
Epoch 40/50
8/8            0s 10ms/step -
accuracy: 0.8895 - loss: 2.7243 - val_accuracy: 0.8481 - val_loss: 2.7766
Epoch 41/50
8/8            0s 10ms/step -
accuracy: 0.8905 - loss: 2.7189 - val_accuracy: 0.8519 - val_loss: 2.7717
Epoch 42/50
8/8            0s 10ms/step -
accuracy: 0.8913 - loss: 2.7135 - val_accuracy: 0.8519 - val_loss: 2.7671
Epoch 43/50
8/8            0s 10ms/step -
accuracy: 0.8968 - loss: 2.7082 - val_accuracy: 0.8525 - val_loss: 2.7625
Epoch 44/50
8/8            0s 10ms/step -
accuracy: 0.8951 - loss: 2.7030 - val_accuracy: 0.8544 - val_loss: 2.7581
Epoch 45/50
8/8            0s 10ms/step -
accuracy: 0.8956 - loss: 2.6980 - val_accuracy: 0.8562 - val_loss: 2.7539
Epoch 46/50
8/8            0s 10ms/step -
accuracy: 0.8963 - loss: 2.6930 - val_accuracy: 0.8569 - val_loss: 2.7498
Epoch 47/50
8/8            0s 10ms/step -
accuracy: 0.8977 - loss: 2.6880 - val_accuracy: 0.8575 - val_loss: 2.7459
Epoch 48/50
8/8            0s 10ms/step -
accuracy: 0.8994 - loss: 2.6832 - val_accuracy: 0.8587 - val_loss: 2.7421
Epoch 49/50
8/8            0s 10ms/step -
accuracy: 0.8970 - loss: 2.6784 - val_accuracy: 0.8594 - val_loss: 2.7384
Epoch 50/50
8/8            0s 10ms/step -
accuracy: 0.8979 - loss: 2.6737 - val_accuracy: 0.8587 - val_loss: 2.7348
```

The model trained (accuracy was 0.4540 at epoch 1 and at epoch 50 it is 0.8979

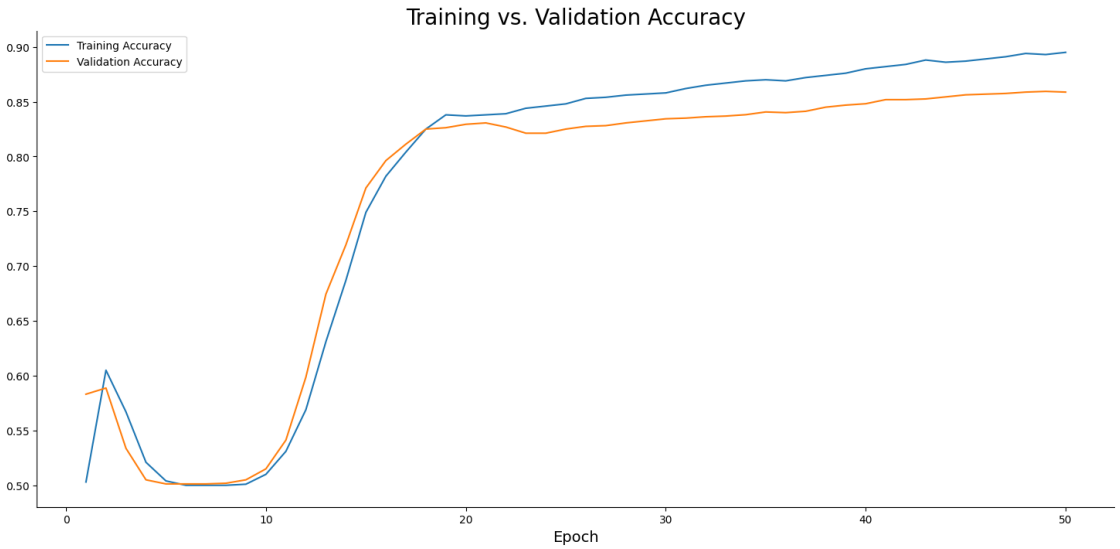
0.8 Model quality indicators

We visualize the losses on the training and test samples and the share of correct answers (accuracy) on the training and test samples.

[25]:
```
from matplotlib import rcParams

rcParams['figure.figsize'] = (18, 8)
rcParams['axes.spines.top'] = False
rcParams['axes.spines.right'] = False
```

[26]:
```
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', size=20)
plt.xlabel('Epoch', size=14)
plt.legend();
```

[27]:
```
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', size=20)
plt.xlabel('Epoch', size=14)
plt.legend();
```



, 5% .

0.9 Training a convolutional model

As with conventional neural networks (with dense layers), convolutional neural networks require experimentation. It is not known in advance how many convolutional layers will be needed, what is the ideal number of filters for each layer, and what is the optimal kernel size.

Convolutional layers are usually followed by a pooling layer to reduce the image size. After the convolutional layers, you must add a `Flatten` layer. Dense layers follow.

The output layer and loss function must be chosen correctly.

[29]:
```
tf.random.set_seed(42)

model_2 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           input_shape=(98, 96, 3), activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           input_shape=(98, 96, 3), activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           activation='leaky_relu'),
```

```
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),


    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.RMSprop(),
    metrics=[tf.keras.metrics.BinaryAccuracy(name='accuracy')]
)

history_2 = model_2.fit(
    train_images,
    train_labels,
    epochs=50,
    batch_size=128,
    validation_data=(test_images, test_labels)
)
```

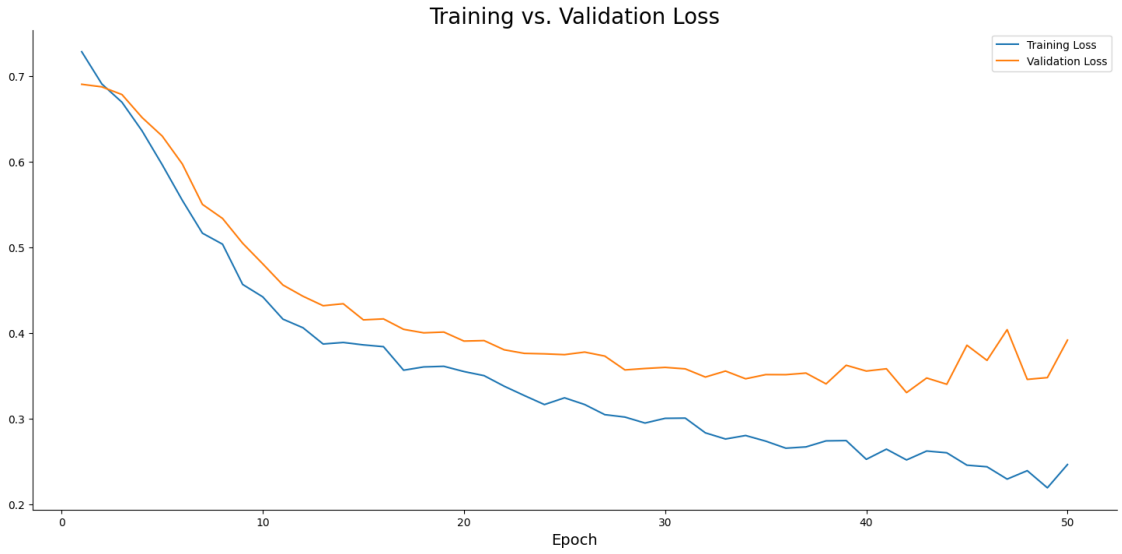
```
Epoch 1/50
8/8             1s 77ms/step -
accuracy: 0.4832 - loss: 0.7589 - val_accuracy: 0.5331 - val_loss: 0.6903
Epoch 2/50
8/8             0s 48ms/step -
accuracy: 0.5416 - loss: 0.6961 - val_accuracy: 0.5431 - val_loss: 0.6872
Epoch 3/50
8/8             0s 46ms/step -
accuracy: 0.5968 - loss: 0.6754 - val_accuracy: 0.5594 - val_loss: 0.6783
Epoch 4/50
8/8             0s 46ms/step -
accuracy: 0.6436 - loss: 0.6399 - val_accuracy: 0.7144 - val_loss: 0.6514
Epoch 5/50
8/8             0s 46ms/step -
accuracy: 0.6641 - loss: 0.6065 - val_accuracy: 0.6606 - val_loss: 0.6298
Epoch 6/50
8/8             0s 47ms/step -
accuracy: 0.7315 - loss: 0.5568 - val_accuracy: 0.7237 - val_loss: 0.5972
Epoch 7/50
8/8             0s 46ms/step -
```

```
accuracy: 0.7446 - loss: 0.5255 - val_accuracy: 0.7794 - val_loss: 0.5500
Epoch 8/50
8/8             0s 46ms/step -
accuracy: 0.7481 - loss: 0.5125 - val_accuracy: 0.7881 - val_loss: 0.5336
Epoch 9/50
8/8             0s 47ms/step -
accuracy: 0.7875 - loss: 0.4592 - val_accuracy: 0.7925 - val_loss: 0.5046
Epoch 10/50
8/8             0s 46ms/step -
accuracy: 0.8160 - loss: 0.4288 - val_accuracy: 0.8044 - val_loss: 0.4805
Epoch 11/50
8/8             0s 47ms/step -
accuracy: 0.8245 - loss: 0.4134 - val_accuracy: 0.8081 - val_loss: 0.4559
Epoch 12/50
8/8             0s 47ms/step -
accuracy: 0.8416 - loss: 0.3957 - val_accuracy: 0.8131 - val_loss: 0.4427
Epoch 13/50
8/8             0s 46ms/step -
accuracy: 0.8416 - loss: 0.3776 - val_accuracy: 0.8181 - val_loss: 0.4317
Epoch 14/50
8/8             0s 46ms/step -
accuracy: 0.8404 - loss: 0.3769 - val_accuracy: 0.8119 - val_loss: 0.4340
Epoch 15/50
8/8             0s 47ms/step -
accuracy: 0.8388 - loss: 0.3808 - val_accuracy: 0.8294 - val_loss: 0.4152
Epoch 16/50
8/8             0s 46ms/step -
accuracy: 0.8509 - loss: 0.3739 - val_accuracy: 0.8319 - val_loss: 0.4163
Epoch 17/50
8/8             0s 46ms/step -
accuracy: 0.8642 - loss: 0.3425 - val_accuracy: 0.8338 - val_loss: 0.4041
Epoch 18/50
8/8             0s 50ms/step -
accuracy: 0.8476 - loss: 0.3499 - val_accuracy: 0.8363 - val_loss: 0.4000
Epoch 19/50
8/8             0s 47ms/step -
accuracy: 0.8650 - loss: 0.3409 - val_accuracy: 0.8369 - val_loss: 0.4009
Epoch 20/50
8/8             0s 46ms/step -
accuracy: 0.8480 - loss: 0.3432 - val_accuracy: 0.8388 - val_loss: 0.3904
Epoch 21/50
8/8             0s 46ms/step -
accuracy: 0.8601 - loss: 0.3409 - val_accuracy: 0.8394 - val_loss: 0.3910
Epoch 22/50
8/8             0s 46ms/step -
accuracy: 0.8679 - loss: 0.3237 - val_accuracy: 0.8456 - val_loss: 0.3802
Epoch 23/50
8/8             0s 47ms/step -
```

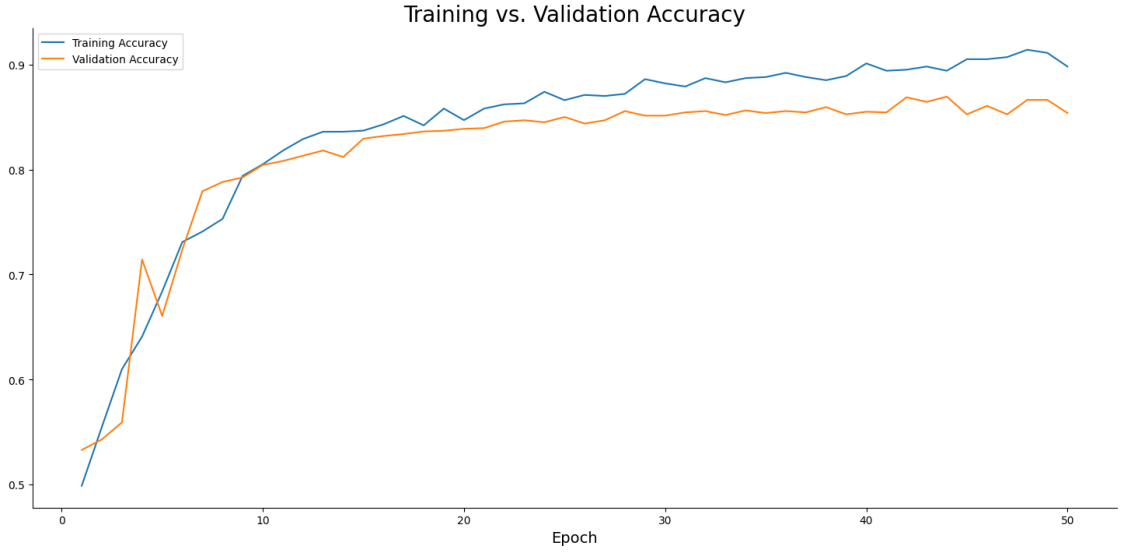
```
accuracy: 0.8741 - loss: 0.3047 - val_accuracy: 0.8469 - val_loss: 0.3760
Epoch 24/50
8/8            0s 46ms/step -
accuracy: 0.8823 - loss: 0.2991 - val_accuracy: 0.8450 - val_loss: 0.3755
Epoch 25/50
8/8            0s 46ms/step -
accuracy: 0.8723 - loss: 0.3076 - val_accuracy: 0.8500 - val_loss: 0.3746
Epoch 26/50
8/8            0s 46ms/step -
accuracy: 0.8761 - loss: 0.2993 - val_accuracy: 0.8438 - val_loss: 0.3775
Epoch 27/50
8/8            0s 47ms/step -
accuracy: 0.8766 - loss: 0.2836 - val_accuracy: 0.8469 - val_loss: 0.3729
Epoch 28/50
8/8            0s 46ms/step -
accuracy: 0.8770 - loss: 0.2834 - val_accuracy: 0.8556 - val_loss: 0.3567
Epoch 29/50
8/8            0s 49ms/step -
accuracy: 0.8888 - loss: 0.2847 - val_accuracy: 0.8512 - val_loss: 0.3584
Epoch 30/50
8/8            0s 47ms/step -
accuracy: 0.8901 - loss: 0.2859 - val_accuracy: 0.8512 - val_loss: 0.3597
Epoch 31/50
8/8            0s 47ms/step -
accuracy: 0.8846 - loss: 0.2794 - val_accuracy: 0.8544 - val_loss: 0.3580
Epoch 32/50
8/8            0s 46ms/step -
accuracy: 0.8991 - loss: 0.2658 - val_accuracy: 0.8556 - val_loss: 0.3484
Epoch 33/50
8/8            0s 47ms/step -
accuracy: 0.8947 - loss: 0.2554 - val_accuracy: 0.8519 - val_loss: 0.3554
Epoch 34/50
8/8            0s 47ms/step -
accuracy: 0.8915 - loss: 0.2594 - val_accuracy: 0.8562 - val_loss: 0.3464
Epoch 35/50
8/8            0s 46ms/step -
accuracy: 0.8924 - loss: 0.2565 - val_accuracy: 0.8537 - val_loss: 0.3512
Epoch 36/50
8/8            0s 46ms/step -
accuracy: 0.8963 - loss: 0.2567 - val_accuracy: 0.8556 - val_loss: 0.3512
Epoch 37/50
8/8            0s 47ms/step -
accuracy: 0.8954 - loss: 0.2490 - val_accuracy: 0.8544 - val_loss: 0.3530
Epoch 38/50
8/8            0s 47ms/step -
accuracy: 0.8978 - loss: 0.2526 - val_accuracy: 0.8594 - val_loss: 0.3404
Epoch 39/50
8/8            0s 48ms/step -
```

```
accuracy: 0.8952 - loss: 0.2484 - val_accuracy: 0.8525 - val_loss: 0.3622
Epoch 40/50
8/8             0s 47ms/step -
accuracy: 0.9108 - loss: 0.2377 - val_accuracy: 0.8550 - val_loss: 0.3554
Epoch 41/50
8/8             0s 47ms/step -
accuracy: 0.9030 - loss: 0.2406 - val_accuracy: 0.8544 - val_loss: 0.3581
Epoch 42/50
8/8             0s 47ms/step -
accuracy: 0.9019 - loss: 0.2342 - val_accuracy: 0.8687 - val_loss: 0.3302
Epoch 43/50
8/8             0s 46ms/step -
accuracy: 0.9039 - loss: 0.2416 - val_accuracy: 0.8644 - val_loss: 0.3473
Epoch 44/50
8/8             0s 47ms/step -
accuracy: 0.9066 - loss: 0.2308 - val_accuracy: 0.8694 - val_loss: 0.3400
Epoch 45/50
8/8             0s 47ms/step -
accuracy: 0.9123 - loss: 0.2264 - val_accuracy: 0.8525 - val_loss: 0.3855
Epoch 46/50
8/8             0s 51ms/step -
accuracy: 0.9115 - loss: 0.2257 - val_accuracy: 0.8606 - val_loss: 0.3678
Epoch 47/50
8/8             0s 47ms/step -
accuracy: 0.9073 - loss: 0.2205 - val_accuracy: 0.8525 - val_loss: 0.4037
Epoch 48/50
8/8             0s 47ms/step -
accuracy: 0.9176 - loss: 0.2242 - val_accuracy: 0.8662 - val_loss: 0.3457
Epoch 49/50
8/8             0s 46ms/step -
accuracy: 0.9198 - loss: 0.2081 - val_accuracy: 0.8662 - val_loss: 0.3478
Epoch 50/50
8/8             0s 47ms/step -
accuracy: 0.9034 - loss: 0.2313 - val_accuracy: 0.8537 - val_loss: 0.3916
```

[34]:
```
plt.plot(np.arange(1, 51), history_2.history['loss'], label='Training Loss')
plt.plot(np.arange(1, 51), history_2.history['val_loss'], label='Validation␣
 ↪Loss')
plt.title('Training vs. Validation Loss', size=20)
plt.xlabel('Epoch', size=14)
plt.legend();
```



[36]:
```
plt.plot(np.arange(1, 51), history_2.history['accuracy'], label='Training␣
↪Accuracy')
plt.plot(np.arange(1, 51), history_2.history['val_accuracy'], label='Validation␣
↪Accuracy')
plt.title('Training vs. Validation Accuracy', size=20)
plt.xlabel('Epoch', size=14)
plt.legend();
```



[45]:
```
loss01 = (history_1.history['loss'][0] - history_1.history['loss'][-1])
loss02 = (history_2.history['loss'][0] - history_2.history['loss'][-1])
print("Model 1 MLP loss: ", loss01)
print("Model 2 CNN loss: ", loss02)
```

```
Model 1 MLP loss:  0.5064220428466797
Model 2 CNN loss:  0.48201826214790344
```

Both models 1 and 2 were trained and improved. However, model 1 "learned" better as the gap between the first epoch and the 50th epoch is bigger than that of the 2nd model. In absoulte loss values model 2 is considerbly more accurate.

[44]:
```
print(history_1.history['loss'][0], history_1.history['loss'][-1])
print(history_2.history['loss'][0], history_2.history['loss'][-1])
```

```
3.1853199005126953 2.6788978576660156
0.728262186050415 0.2462439239025116
```

0.10 5. Compare the quality of binary classification by neural networks using the quality indicator specified in the individual task.

Binary classification quality indicator: Foulkes–Mallows index equal to the square root of TP/(TP + TN) * TP/(TP + FP)

0.11 6. Visualize the ROC curves for the constructed classifiers in one figure (with legend) and calculate the areas under the ROC curves.

[46]:
```
def TN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 0) & (y_predict == 0))
def FP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 0) & (y_predict == 1))
def FN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 1) & (y_predict == 0))
def TP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 1) & (y_predict == 1))
def confusion_matrix(y_true, y_predict):
    return np.array([
        [TP(y_true, y_predict), FN(y_true, y_predict)],
        [FP(y_true, y_predict), TN(y_true, y_predict)]
    ])
```

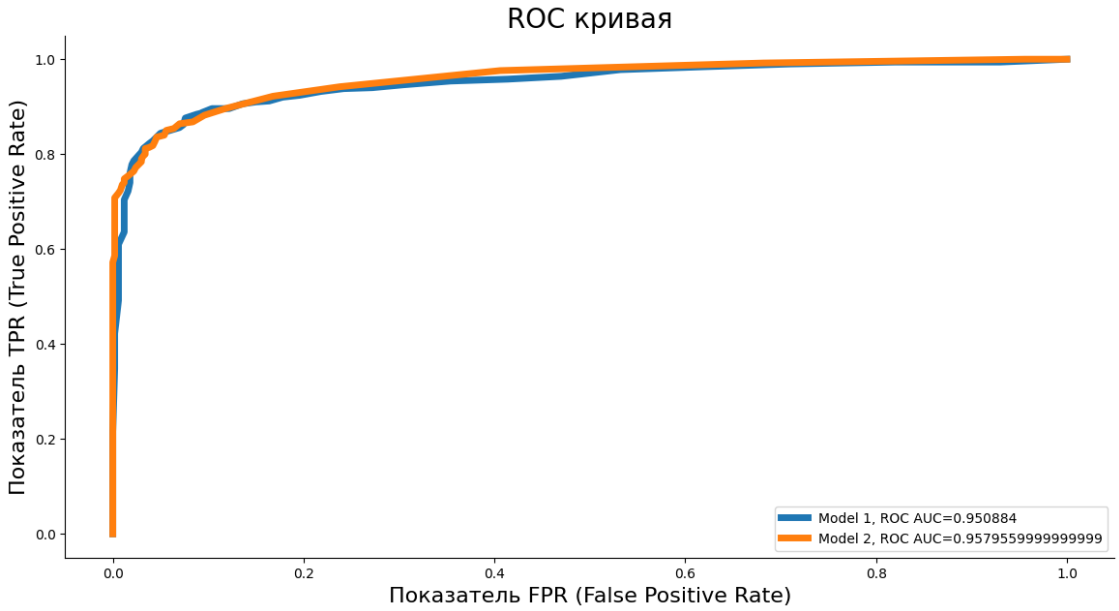
[56]:
```
def FMI(y_true, y_predict):
        return TP(y_true, y_predict) / sqrt((TP(y_true, y_predict) + FP(y_true,␣
 ↪y_predict)) * (TP(y_true, y_predict) + FN(y_true, y_predict)))
```

[48]:
```
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
```

[49]:
```
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)
```

[52]:
```
plt.figure(figsize=(14,7))
predictionFromModel_01 = model_1.predict(train_images)
predictionFromModel_02 = model_2.predict(train_images)
r1 = roc_auc_score(train_labels, predictionFromModel_01)
r2 = roc_auc_score(train_labels, predictionFromModel_02)
ROC1 = roc_from_scratch(predictionFromModel_01.
 ↪reshape(-1),train_labels,partitions=50)
plt.plot(ROC1[:,0],ROC1[:,1],lw=5, label='Model 1, ROC AUC=' + str
(r1))
ROC2 = roc_from_scratch(predictionFromModel_02.
 ↪reshape(-1),train_labels,partitions=50)
plt.plot(ROC2[:,0],ROC2[:,1],lw=5, label='Model 2, ROC AUC=' + str
(r2))
plt.title('ROC     ',fontsize=20)
plt.xlabel('       FPR (False Positive Rate)',fontsize=16)
plt.ylabel('       TPR (True Positive Rate)',fontsize=16)
plt.legend();
```

```
32/32             0s 1ms/step
32/32             0s 2ms/step
```



[60]:
```
y_predict_01 = np.array([1 if prob > 0.5 else 0 for prob in np.
 ↪ravel(predictionFromModel_01)])
y_predict_02 = np.array([1 if prob > 0.5 else 0 for prob in np.
 ↪ravel(predictionFromModel_02)])
FMI(np.array(train_labels), y_predict_01), FMI(np.array(train_labels),␣
 ↪y_predict_02)
```

[60]:
```
(0.8895457136504374, 0.8592627465056701)
```

[62]:
```
from sklearn.metrics.cluster import fowlkes_mallows_score
fowlkes_mallows_score(np.array(train_labels), y_predict_01),␣
 ↪fowlkes_mallows_score(np.array(train_labels), y_predict_02)
```

[62]:
```
(0.8090419573545398, 0.7780152274006994)
```

0.12 7. Leave in the set the images of the three classes specified in the individual assignment. Train MLP and CNN neural networks for the task of multi-class image classification (network architecture requirements are specified in the individual task).

0.13 8. Compare the quality of multi-class classification by neural networks using the quality indicator specified in the individual task.

0.14 9. Plot learning curves for multiclass classification neural networks for error rates and the proportion of correct answers depending on the training epoch, labeling the axes and figure and creating a legend. Provide the program code with the necessary comments.

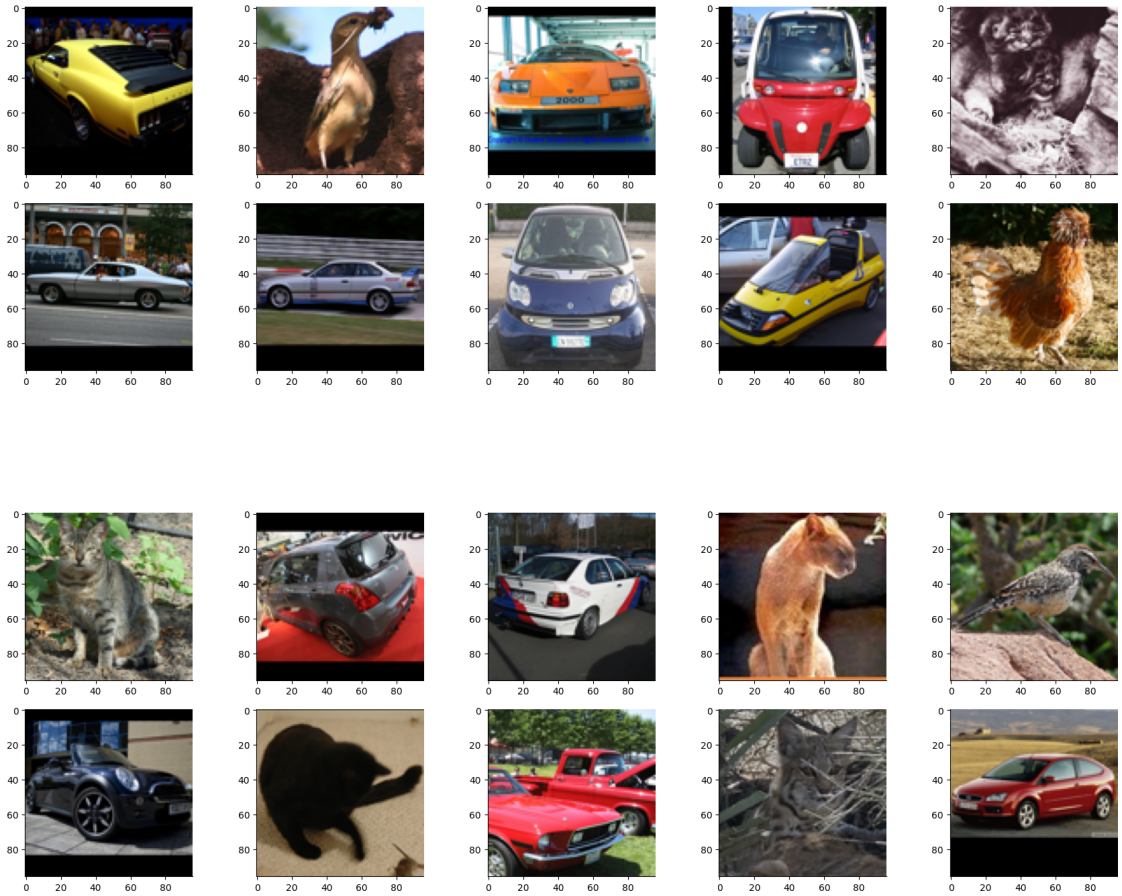
[70]:
```
X = df_train[df_train['label'] == 1]
X['label'] = 0
y = df_train[df_train['label'] == 2]
y['label'] = 1
z = df_train[df_train['label'] == 3]
z['label'] = 2
df_train_02 = pd.concat([X, y, z])

X = df_test[df_test['label'] == 1]
X['label'] = 0
y = df_test[df_test['label'] == 2]
y['label'] = 1
z = df_test[df_test['label'] == 3]
z['label'] = 2
df_test_02 = pd.concat([X, y, z])
df_train_02['label'].value_counts(), df_test_02['label'].value_counts()
```

```
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\169528981.py:2:
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
  X['label'] = 0
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\169528981.py:4:
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
  y['label'] = 1
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\169528981.py:6:
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
  z['label'] = 2
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\169528981.py:10:
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
  X['label'] = 0
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\169528981.py:12:
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
  y['label'] = 1
C:\Users\Mo\AppData\Local\Temp\ipykernel_15028\169528981.py:14:
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
  z['label'] = 2
```

[70]:
```
(label
 0    500
 1    500
 2    500
 Name: count, dtype: int64,
 label
 0    800
 1    800
 2    800
 Name: count, dtype: int64)
```

[71]:
```
#let's check if we have correctly chose the lable
plot_random_sample(df_train_02['image'])
plot_random_sample(df_test_02['image'])
```



[72]:
```
train_labels_02 = df_train_02['label'].to_numpy(dtype=np.float32)
test_labels_02 = df_test_02['label'].to_numpy(dtype=np.float32)
train_labels_02.shape, test_labels_02.shape
```

[72]:
```
((1500,), (2400,))
```

[88]:
```
label_train = list(df_train_02['label'])
label_test = list(df_test_02['label'])
```

[83]:
```
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
```

[90]:
```
train_labels_02 = to_one_hot(label_train)
test_labels_02 = to_one_hot(label_test)
train_labels_02.shape, test_labels_02.shape
```

[90]:
```
((1500, 3), (2400, 3))
```

[86]:
```
train_images_02 = np.zeros(shape=(df_train_02.shape[0],96,96,3), dtype=np.
 ↪float32)
test_images_02 = np.zeros(shape=(df_test_02.shape[0],96,96,3), dtype=np.float32)
```

[93]:
```
for idx in range(train_labels.shape[0]):
    train_images_02[idx,:,:,:] = \
    np.array(Image.fromarray(df_train_02.iloc[idx]['image']))
for idx in range(test_labels.shape[0]):
    test_images_02[idx,:,:,:] = \
    np.array(Image.fromarray(df_test_02.iloc[idx]['image']))
```

[94]:
```
train_images_02 /= 255
test_images_02 /= 255
train_images_02.shape, test_images_02.shape, train_labels_02.shape,␣
 ↪test_labels_02.shape
```

[94]:
```
((1500, 96, 96, 3), (2400, 96, 96, 3), (1500, 3), (2400, 3))
```

[97]:
```
tf.random.set_seed(42)

model_3 = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(96, 96, 3)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(50, activation='swish', kernel_regularizer ='l2'), #1
    tf.keras.layers.Dense(60, activation='swish'),
    tf.keras.layers.Dense(70, activation='swish', kernel_regularizer ='l2'), #3
    tf.keras.layers.Dense(80, activation='swish'),
    tf.keras.layers.Dense(90, activation='swish', kernel_regularizer ='l2'), #5
    tf.keras.layers.Dense(3, activation='softmax')
])

model_3.compile(
    loss=tf.keras.losses.categorical_crossentropy,
    optimizer=tf.keras.optimizers.Adagrad(),
    metrics=[tf.keras.metrics.CategoricalAccuracy(name='accuracy')]
)

history_3 = model_3.fit(
    train_images_02,
    train_labels_02,
    epochs=50,
    batch_size=128,
    validation_data=(test_images_02, test_labels_02)
)
```

```
Epoch 1/50
```

```
12/12            1s 26ms/step -
accuracy: 0.3988 - loss: 3.5746 - val_accuracy: 0.6808 - val_loss: 3.5460
Epoch 2/50
12/12            0s 9ms/step -
accuracy: 0.6784 - loss: 3.5330 - val_accuracy: 0.6667 - val_loss: 3.4786
Epoch 3/50
12/12            0s 9ms/step -
accuracy: 0.6761 - loss: 3.4622 - val_accuracy: 0.6667 - val_loss: 3.4132
Epoch 4/50
12/12            0s 9ms/step -
accuracy: 0.6761 - loss: 3.4032 - val_accuracy: 0.6667 - val_loss: 3.3724
Epoch 5/50
12/12            0s 9ms/step -
accuracy: 0.6761 - loss: 3.3652 - val_accuracy: 0.6667 - val_loss: 3.3420
Epoch 6/50
12/12            0s 9ms/step -
accuracy: 0.6761 - loss: 3.3361 - val_accuracy: 0.6667 - val_loss: 3.3165
Epoch 7/50
12/12            0s 9ms/step -
accuracy: 0.6766 - loss: 3.3111 - val_accuracy: 0.6667 - val_loss: 3.2932
Epoch 8/50
12/12            0s 10ms/step -
accuracy: 0.6785 - loss: 3.2872 - val_accuracy: 0.6892 - val_loss: 3.2695
Epoch 9/50
12/12            0s 10ms/step -
accuracy: 0.7155 - loss: 3.2623 - val_accuracy: 0.8087 - val_loss: 3.2440
Epoch 10/50
12/12            0s 9ms/step -
accuracy: 0.8096 - loss: 3.2347 - val_accuracy: 0.8637 - val_loss: 3.2155
Epoch 11/50
12/12            0s 9ms/step -
accuracy: 0.8684 - loss: 3.2035 - val_accuracy: 0.8813 - val_loss: 3.1849
Epoch 12/50
12/12            0s 9ms/step -
accuracy: 0.8915 - loss: 3.1701 - val_accuracy: 0.8838 - val_loss: 3.1548
Epoch 13/50
12/12            0s 10ms/step -
accuracy: 0.8987 - loss: 3.1372 - val_accuracy: 0.8829 - val_loss: 3.1282
Epoch 14/50
12/12            0s 9ms/step -
accuracy: 0.9005 - loss: 3.1080 - val_accuracy: 0.8842 - val_loss: 3.1062
Epoch 15/50
12/12            0s 9ms/step -
accuracy: 0.9056 - loss: 3.0835 - val_accuracy: 0.8850 - val_loss: 3.0883
Epoch 16/50
12/12            0s 9ms/step -
accuracy: 0.9054 - loss: 3.0633 - val_accuracy: 0.8842 - val_loss: 3.0735
Epoch 17/50
```

```
12/12            0s 13ms/step -
accuracy: 0.9040 - loss: 3.0466 - val_accuracy: 0.8838 - val_loss: 3.0608
Epoch 18/50
12/12            0s 9ms/step -
accuracy: 0.9035 - loss: 3.0324 - val_accuracy: 0.8863 - val_loss: 3.0496
Epoch 19/50
12/12            0s 9ms/step -
accuracy: 0.9058 - loss: 3.0198 - val_accuracy: 0.8867 - val_loss: 3.0393
Epoch 20/50
12/12            0s 9ms/step -
accuracy: 0.9049 - loss: 3.0086 - val_accuracy: 0.8879 - val_loss: 3.0297
Epoch 21/50
12/12            0s 9ms/step -
accuracy: 0.9074 - loss: 2.9982 - val_accuracy: 0.8883 - val_loss: 3.0207
Epoch 22/50
12/12            0s 9ms/step -
accuracy: 0.9083 - loss: 2.9884 - val_accuracy: 0.8883 - val_loss: 3.0120
Epoch 23/50
12/12            0s 9ms/step -
accuracy: 0.9071 - loss: 2.9792 - val_accuracy: 0.8883 - val_loss: 3.0036
Epoch 24/50
12/12            0s 9ms/step -
accuracy: 0.9095 - loss: 2.9703 - val_accuracy: 0.8892 - val_loss: 2.9955
Epoch 25/50
12/12            0s 9ms/step -
accuracy: 0.9118 - loss: 2.9618 - val_accuracy: 0.8917 - val_loss: 2.9876
Epoch 26/50
12/12            0s 9ms/step -
accuracy: 0.9124 - loss: 2.9535 - val_accuracy: 0.8921 - val_loss: 2.9800
Epoch 27/50
12/12            0s 9ms/step -
accuracy: 0.9132 - loss: 2.9455 - val_accuracy: 0.8917 - val_loss: 2.9725
Epoch 28/50
12/12            0s 9ms/step -
accuracy: 0.9158 - loss: 2.9376 - val_accuracy: 0.8929 - val_loss: 2.9651
Epoch 29/50
12/12            0s 9ms/step -
accuracy: 0.9139 - loss: 2.9298 - val_accuracy: 0.8963 - val_loss: 2.9579
Epoch 30/50
12/12            0s 10ms/step -
accuracy: 0.9162 - loss: 2.9222 - val_accuracy: 0.8975 - val_loss: 2.9509
Epoch 31/50
12/12            0s 9ms/step -
accuracy: 0.9167 - loss: 2.9146 - val_accuracy: 0.8983 - val_loss: 2.9440
Epoch 32/50
12/12            0s 9ms/step -
accuracy: 0.9184 - loss: 2.9072 - val_accuracy: 0.8996 - val_loss: 2.9372
Epoch 33/50
```

```
12/12            0s 9ms/step -
accuracy: 0.9175 - loss: 2.8998 - val_accuracy: 0.8996 - val_loss: 2.9305
Epoch 34/50
12/12            0s 9ms/step -
accuracy: 0.9193 - loss: 2.8925 - val_accuracy: 0.9008 - val_loss: 2.9240
Epoch 35/50
12/12            0s 9ms/step -
accuracy: 0.9201 - loss: 2.8852 - val_accuracy: 0.9021 - val_loss: 2.9175
Epoch 36/50
12/12            0s 9ms/step -
accuracy: 0.9201 - loss: 2.8780 - val_accuracy: 0.9029 - val_loss: 2.9112
Epoch 37/50
12/12            0s 9ms/step -
accuracy: 0.9211 - loss: 2.8708 - val_accuracy: 0.9033 - val_loss: 2.9050
Epoch 38/50
12/12            0s 9ms/step -
accuracy: 0.9215 - loss: 2.8636 - val_accuracy: 0.9025 - val_loss: 2.8989
Epoch 39/50
12/12            0s 9ms/step -
accuracy: 0.9240 - loss: 2.8565 - val_accuracy: 0.9033 - val_loss: 2.8929
Epoch 40/50
12/12            0s 9ms/step -
accuracy: 0.9242 - loss: 2.8495 - val_accuracy: 0.9042 - val_loss: 2.8870
Epoch 41/50
12/12            0s 9ms/step -
accuracy: 0.9246 - loss: 2.8424 - val_accuracy: 0.9058 - val_loss: 2.8812
Epoch 42/50
12/12            0s 9ms/step -
accuracy: 0.9251 - loss: 2.8354 - val_accuracy: 0.9071 - val_loss: 2.8755
Epoch 43/50
12/12            0s 9ms/step -
accuracy: 0.9256 - loss: 2.8284 - val_accuracy: 0.9087 - val_loss: 2.8699
Epoch 44/50
12/12            0s 9ms/step -
accuracy: 0.9268 - loss: 2.8214 - val_accuracy: 0.9112 - val_loss: 2.8643
Epoch 45/50
12/12            0s 9ms/step -
accuracy: 0.9304 - loss: 2.8144 - val_accuracy: 0.9112 - val_loss: 2.8589
Epoch 46/50
12/12            0s 9ms/step -
accuracy: 0.9324 - loss: 2.8075 - val_accuracy: 0.9125 - val_loss: 2.8535
Epoch 47/50
12/12            0s 9ms/step -
accuracy: 0.9338 - loss: 2.8006 - val_accuracy: 0.9129 - val_loss: 2.8482
Epoch 48/50
12/12            0s 9ms/step -
accuracy: 0.9351 - loss: 2.7937 - val_accuracy: 0.9137 - val_loss: 2.8430
Epoch 49/50
```

```
12/12              0s 9ms/step -
accuracy: 0.9360 - loss: 2.7869 - val_accuracy: 0.9146 - val_loss: 2.8379
Epoch 50/50
12/12              0s 9ms/step -
accuracy: 0.9365 - loss: 2.7801 - val_accuracy: 0.9150 - val_loss: 2.8329
```

[96]:
```
plt.plot(np.arange(1, 51), history_3.history['loss'], label='Training Loss')
plt.plot(np.arange(1, 51), history_3.history['val_loss'], label='Validation␣
 ↪Loss')
plt.title('Training vs. Validation Loss', size=20)
plt.xlabel('Epoch', size=14)
plt.legend();
```

[99]:
```
tf.random.set_seed(42)

model_4 = tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           input_shape=(98, 96, 3), activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                           input_shape=(98, 96, 3), activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
```

```
                            activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Conv2D(filters=16, kernel_size=(2, 2),
                            activation='leaky_relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2, 2), padding='same'),
    tf.keras.layers.Dropout(rate=0.3),


    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_4.compile(
    loss=tf.keras.losses.categorical_crossentropy,
    optimizer=tf.keras.optimizers.RMSprop(),
    metrics=[tf.keras.metrics.CategoricalAccuracy(name='accuracy')]
)

history_4 = model_4.fit(
    train_images_02,
    train_labels_02,
    epochs=50,
    batch_size=128,
    validation_data=(test_images_02, test_labels_02)
)
```

```
Epoch 1/50
C:\Users\Mo\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kf
ra8p0\LocalCache\local-packages\Python311\site-
packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(
12/12            2s 65ms/step -
accuracy: 0.4254 - loss: 1.0016 - val_accuracy: 0.8758 - val_loss: 0.9482
Epoch 2/50
12/12            1s 46ms/step -
accuracy: 0.7545 - loss: 0.7691 - val_accuracy: 0.8779 - val_loss: 0.8371
Epoch 3/50
12/12            1s 44ms/step -
accuracy: 0.7997 - loss: 0.6405 - val_accuracy: 0.8783 - val_loss: 0.6818
Epoch 4/50
12/12            1s 45ms/step -
```

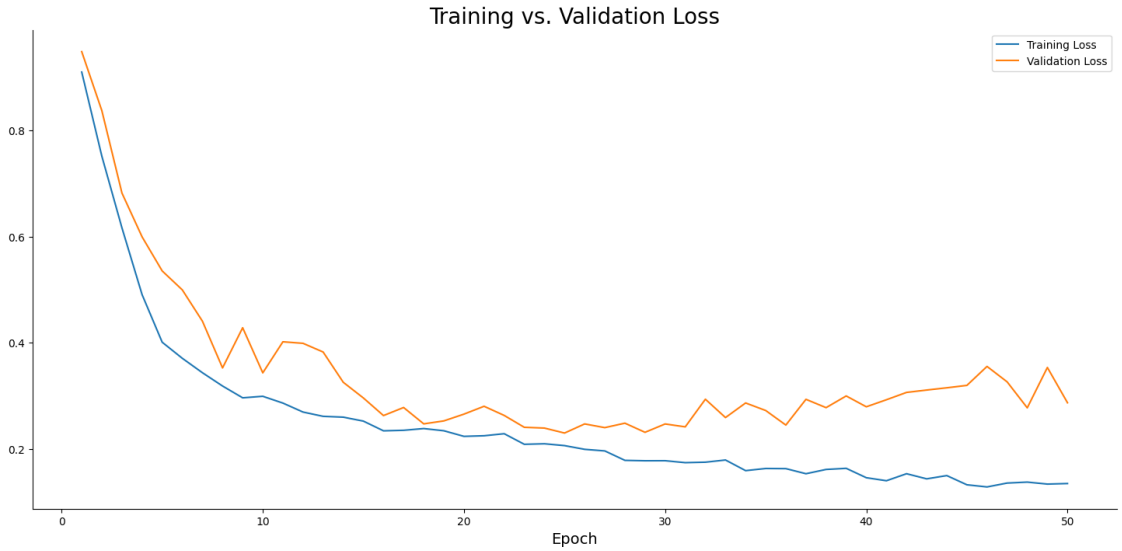
```
accuracy: 0.7981 - loss: 0.5272 - val_accuracy: 0.8537 - val_loss: 0.5994
Epoch 5/50
12/12            1s 44ms/step -
accuracy: 0.8274 - loss: 0.4092 - val_accuracy: 0.8408 - val_loss: 0.5352
Epoch 6/50
12/12            1s 44ms/step -
accuracy: 0.8305 - loss: 0.3701 - val_accuracy: 0.8413 - val_loss: 0.4995
Epoch 7/50
12/12            1s 44ms/step -
accuracy: 0.8495 - loss: 0.3365 - val_accuracy: 0.8558 - val_loss: 0.4405
Epoch 8/50
12/12            1s 49ms/step -
accuracy: 0.8548 - loss: 0.3153 - val_accuracy: 0.8800 - val_loss: 0.3526
Epoch 9/50
12/12            1s 44ms/step -
accuracy: 0.8776 - loss: 0.2933 - val_accuracy: 0.8517 - val_loss: 0.4282
Epoch 10/50
12/12            1s 44ms/step -
accuracy: 0.8762 - loss: 0.2973 - val_accuracy: 0.8846 - val_loss: 0.3433
Epoch 11/50
12/12            1s 44ms/step -
accuracy: 0.8827 - loss: 0.2767 - val_accuracy: 0.8612 - val_loss: 0.4018
Epoch 12/50
12/12            1s 44ms/step -
accuracy: 0.8899 - loss: 0.2579 - val_accuracy: 0.8658 - val_loss: 0.3989
Epoch 13/50
12/12            1s 44ms/step -
accuracy: 0.8954 - loss: 0.2572 - val_accuracy: 0.8658 - val_loss: 0.3825
Epoch 14/50
12/12            1s 44ms/step -
accuracy: 0.9027 - loss: 0.2528 - val_accuracy: 0.8904 - val_loss: 0.3255
Epoch 15/50
12/12            1s 44ms/step -
accuracy: 0.9025 - loss: 0.2424 - val_accuracy: 0.9000 - val_loss: 0.2960
Epoch 16/50
12/12            1s 44ms/step -
accuracy: 0.9099 - loss: 0.2299 - val_accuracy: 0.8992 - val_loss: 0.2628
Epoch 17/50
12/12            1s 44ms/step -
accuracy: 0.9049 - loss: 0.2251 - val_accuracy: 0.9013 - val_loss: 0.2780
Epoch 18/50
12/12            1s 44ms/step -
accuracy: 0.9017 - loss: 0.2373 - val_accuracy: 0.8983 - val_loss: 0.2472
Epoch 19/50
12/12            1s 44ms/step -
accuracy: 0.9113 - loss: 0.2290 - val_accuracy: 0.8988 - val_loss: 0.2528
Epoch 20/50
12/12            1s 44ms/step -
```

```
accuracy: 0.9179 - loss: 0.2183 - val_accuracy: 0.9042 - val_loss: 0.2655
Epoch 21/50
12/12            1s 44ms/step -
accuracy: 0.9147 - loss: 0.2199 - val_accuracy: 0.9083 - val_loss: 0.2803
Epoch 22/50
12/12            1s 45ms/step -
accuracy: 0.9195 - loss: 0.2228 - val_accuracy: 0.9029 - val_loss: 0.2631
Epoch 23/50
12/12            1s 44ms/step -
accuracy: 0.9284 - loss: 0.2033 - val_accuracy: 0.9050 - val_loss: 0.2407
Epoch 24/50
12/12            1s 48ms/step -
accuracy: 0.9271 - loss: 0.2058 - val_accuracy: 0.8938 - val_loss: 0.2393
Epoch 25/50
12/12            1s 44ms/step -
accuracy: 0.9182 - loss: 0.2137 - val_accuracy: 0.9021 - val_loss: 0.2299
Epoch 26/50
12/12            1s 44ms/step -
accuracy: 0.9283 - loss: 0.1971 - val_accuracy: 0.8946 - val_loss: 0.2471
Epoch 27/50
12/12            1s 44ms/step -
accuracy: 0.9225 - loss: 0.1959 - val_accuracy: 0.8938 - val_loss: 0.2401
Epoch 28/50
12/12            1s 44ms/step -
accuracy: 0.9312 - loss: 0.1719 - val_accuracy: 0.8913 - val_loss: 0.2485
Epoch 29/50
12/12            1s 44ms/step -
accuracy: 0.9356 - loss: 0.1878 - val_accuracy: 0.8967 - val_loss: 0.2313
Epoch 30/50
12/12            1s 44ms/step -
accuracy: 0.9302 - loss: 0.1728 - val_accuracy: 0.8929 - val_loss: 0.2470
Epoch 31/50
12/12            1s 44ms/step -
accuracy: 0.9332 - loss: 0.1792 - val_accuracy: 0.8950 - val_loss: 0.2416
Epoch 32/50
12/12            1s 44ms/step -
accuracy: 0.9340 - loss: 0.1782 - val_accuracy: 0.8858 - val_loss: 0.2936
Epoch 33/50
12/12            1s 44ms/step -
accuracy: 0.9359 - loss: 0.1870 - val_accuracy: 0.8908 - val_loss: 0.2591
Epoch 34/50
12/12            1s 44ms/step -
accuracy: 0.9467 - loss: 0.1554 - val_accuracy: 0.8875 - val_loss: 0.2866
Epoch 35/50
12/12            1s 44ms/step -
accuracy: 0.9405 - loss: 0.1719 - val_accuracy: 0.8908 - val_loss: 0.2723
Epoch 36/50
12/12            1s 44ms/step -
```

```
accuracy: 0.9375 - loss: 0.1720 - val_accuracy: 0.8954 - val_loss: 0.2449
Epoch 37/50
12/12             1s 45ms/step -
accuracy: 0.9418 - loss: 0.1579 - val_accuracy: 0.8896 - val_loss: 0.2934
Epoch 38/50
12/12             1s 45ms/step -
accuracy: 0.9391 - loss: 0.1656 - val_accuracy: 0.8900 - val_loss: 0.2777
Epoch 39/50
12/12             1s 45ms/step -
accuracy: 0.9360 - loss: 0.1701 - val_accuracy: 0.8875 - val_loss: 0.2998
Epoch 40/50
12/12             1s 45ms/step -
accuracy: 0.9440 - loss: 0.1545 - val_accuracy: 0.8929 - val_loss: 0.2794
Epoch 41/50
12/12             1s 47ms/step -
accuracy: 0.9454 - loss: 0.1471 - val_accuracy: 0.8908 - val_loss: 0.2926
Epoch 42/50
12/12             1s 50ms/step -
accuracy: 0.9382 - loss: 0.1537 - val_accuracy: 0.8879 - val_loss: 0.3064
Epoch 43/50
12/12             1s 45ms/step -
accuracy: 0.9404 - loss: 0.1407 - val_accuracy: 0.8896 - val_loss: 0.3109
Epoch 44/50
12/12             1s 44ms/step -
accuracy: 0.9406 - loss: 0.1569 - val_accuracy: 0.8883 - val_loss: 0.3151
Epoch 45/50
12/12             1s 45ms/step -
accuracy: 0.9440 - loss: 0.1398 - val_accuracy: 0.8879 - val_loss: 0.3197
Epoch 46/50
12/12             1s 44ms/step -
accuracy: 0.9553 - loss: 0.1331 - val_accuracy: 0.8867 - val_loss: 0.3554
Epoch 47/50
12/12             1s 44ms/step -
accuracy: 0.9487 - loss: 0.1453 - val_accuracy: 0.8896 - val_loss: 0.3263
Epoch 48/50
12/12             1s 45ms/step -
accuracy: 0.9522 - loss: 0.1390 - val_accuracy: 0.8992 - val_loss: 0.2774
Epoch 49/50
12/12             1s 44ms/step -
accuracy: 0.9452 - loss: 0.1404 - val_accuracy: 0.8867 - val_loss: 0.3534
Epoch 50/50
12/12             1s 45ms/step -
accuracy: 0.9434 - loss: 0.1502 - val_accuracy: 0.8988 - val_loss: 0.2869
```

[100]:
```
plt.plot(np.arange(1, 51), history_4.history['loss'], label='Training Loss')
plt.plot(np.arange(1, 51), history_4.history['val_loss'], label='Validation␣
↪Loss')
```

```
plt.title('Training vs. Validation Loss', size=20)
plt.xlabel('Epoch', size=14)
plt.legend();
```



[101]:
```
def Recall(y_true, y_predict):
    return TP(y_true, y_predict) / (TP(y_true, y_predict) + FN(y_true,␣
 ↪y_predict))
```

Here is where I got the formula from https://www.evidentlyai.com/classification-metrics/accuracy-precision-recall#:~:text=Recall%20is%20a%20metric%20that,the%20number%20of%20positive%20instances.

[103]:
```
y_pred_model_03 = model_3.predict(test_images_02)
y_pred_model_03 = np.argmax(y_pred_model_03, axis=1)
y_true_model_03 = np.argmax(test_labels_02, axis=1)

y_pred_model_04 = model_4.predict(test_images_02)
y_pred_model_04 = np.argmax(y_pred_model_04, axis=1)
y_true_model_04 = np.argmax(test_labels_02, axis=1)
```

```
75/75             0s 1ms/step
75/75             0s 2ms/step
```

[106]:
```
print("Recal model 3:", Recall(y_true_model_03, y_pred_model_03))
print("Recal model 4:", Recall(y_true_model_04, y_pred_model_04))
```

```
Recal model 3: 0.8325
Recal model 4: 0.70375
```