prepare images

In [1]:

```
#import packages
from PIL import Image #Bildbearbeitungspacket (PIL = Pillow)
import numpy as np
import os # provides functions for creating and removing a directory (folder), fetching its
from sklearn.preprocessing import OneHotEncoder
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Dropout
from tensorflow.keras import Model

import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:

```
def read_traindata(path):
    images = []
    labels = []
    for image in sorted(os.listdir(str(path))):
        im = Image.open(str(path)+str(image))
        im = im.resize((200,200)) #resize to get all images to the same dimension
        images.append(np.array(im))
        if image[4] == 'M':
            labels.append(True)
        else:
            labels.append(False)

    return images, labels
```

In [3]:

```
#read in trainings and test images

train_images, train_labels = read_traindata('fold1/train/400X/')

test_images, test_labels = read_traindata('fold1/test/400X/')
```

In [4]:

```
#convert train_images, train_labels, test_images and test_labels to a numpy array
train_images = np.array(train_images)
train_labels = np.array(train_labels)
test_images = np.array(test_images)

#reshape train_labels
train_labels = np.reshape(train_labels,(-1,1)) #-1 for unknown dimension
```

In [5]:

```
#conver train_labels to one-hot encoding
#hierbei wird aus [False] [1. 0.] und aus [True] [0. 1.]

enc = OneHotEncoder() #a new binary variable is added for each unique integer value
train_lab = enc.fit_transform(train_labels).toarray()
```

In [6]:

```
# normalize images by dividing them with the maximum number of pixel
# jetzt liegen die Werte zwischen 0 und 1

train_im = train_images / 255.0
test_im = test_images / 255.0
```

In [7]:

```
#build model
buffer_size = len(train_im)
print(buffer_size)
train_ds = tf.data.Dataset.from_tensor_slices((train_im, train_lab)).shuffle(buffer_size).b
```

```
1165
```

CNN Model

In [8]:

```
# define class CNN_Model by using Model and modifying it
# CNN_Model can identify patterns and is therefore usefull for Image analysis
# we use a convolutional/faltendes Neural Network (CNN) with dense layers for class predict
# A convolutional layer has a number of filters that do convolutional operations.


class CNN_Model(Model):
    def __init__(self):
        super(CNN_Model, self).__init__()
        self.conv1 = Conv2D(32, 3, padding='same', activation='relu') #2D convolution layer
        self.pool1 = MaxPool2D((2,2)) #Max pooling operation for 2D spatial data. --> nimmt
        self.conv2 = Conv2D(64, 3, padding='same', activation='relu')
        self.pool2 = MaxPool2D((2,2))
        self.flatten = Flatten() #Flattens the input. Does not affect the batch size. --> "
        self.d1 = Dense(512, activation='relu') #normal NN layer, with relu as an activatio
        self.dropout1 = Dropout(0.4) # Applies Dropout to the input.
        self.d2 = Dense(128, activation='relu')
        self.dropout2 = Dropout(0.4)
        self.d3 = Dense(43, activation='relu')
        self.d4 = Dense(2, activation='softmax')

    def call(self, x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.flatten(x)
        x = self.d1(x)
        x = self.dropout1(x)
        x = self.d2(x)
        x = self.dropout2(x)
        x = self.d3(x)
        x = self.d4(x)
        return x

model_CNN = CNN_Model()
```

In [9]:

```
#create an object for our model
#we use  Adam as our optimizer
loss_object = tf.keras.losses.CategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()

#define our loss functions
#We use categorical cross-entropy as our loss function
train_loss = tf.keras.metrics.Mean(name='train_loss') # mean value of all losses for each e
train_accuracy = tf.keras.metrics.CategoricalAccuracy(name='train_accuracy') #the accuracy
```

In [10]:

```
@tf.function #this decorator converts the function into a graph.
def train_step(images, labels):
    '''function to train our model and computes the loss and gradients'''
    with tf.GradientTape() as tape:
        #tf.GradientTape() is a high-level API that is used to compute differentiations
        predictions = model_CNN(images)
        loss = loss_object(labels, predictions)

        gradients = tape.gradient(loss, model_CNN.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model_CNN.trainable_variables))

        train_loss(loss)
        train_accuracy(labels, predictions)
```

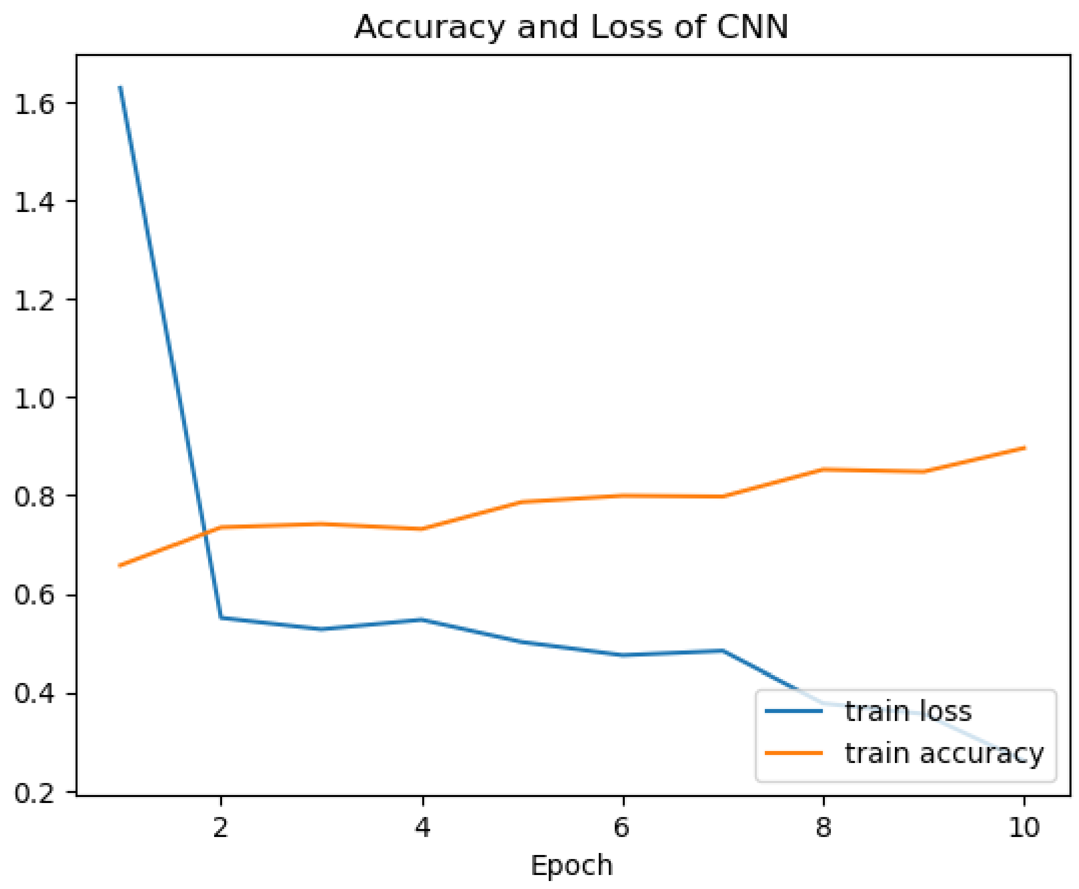
In [11]:

```
#build and train our model
EPOCHS = 10
train_loss_CNN = []
train_accuracy_CNN = []

for epoch in range(EPOCHS):
    for images, labels in train_ds:
        train_step(images, labels)

    model_CNN.save_weights('./content', save_format='tf')
    train_loss_CNN.append(float(train_loss.result()))
    train_accuracy_CNN.append(float(train_accuracy.result()))
    print('Epoch:',str(epoch+1),' Loss:',str(train_loss.result()),' Accuracy:', str(train_a
    train_loss.reset_states()
    train_accuracy.reset_states()
```

```
Epoch: 1  Loss: tf.Tensor(1.627959, shape=(), dtype=float32)  Accuracy: tf.T
ensor(65.751076, shape=(), dtype=float32)
Epoch: 2  Loss: tf.Tensor(0.550692, shape=(), dtype=float32)  Accuracy: tf.T
ensor(73.476395, shape=(), dtype=float32)
Epoch: 3  Loss: tf.Tensor(0.5273283, shape=(), dtype=float32)  Accuracy: tf.
Tensor(74.16309, shape=(), dtype=float32)
Epoch: 4  Loss: tf.Tensor(0.5467841, shape=(), dtype=float32)  Accuracy: tf.
Tensor(73.13305, shape=(), dtype=float32)
Epoch: 5  Loss: tf.Tensor(0.5014125, shape=(), dtype=float32)  Accuracy: tf.
Tensor(78.62661, shape=(), dtype=float32)
Epoch: 6  Loss: tf.Tensor(0.47455716, shape=(), dtype=float32)  Accuracy: t
f.Tensor(79.91416, shape=(), dtype=float32)
Epoch: 7  Loss: tf.Tensor(0.48361468, shape=(), dtype=float32)  Accuracy: t
f.Tensor(79.74249, shape=(), dtype=float32)
Epoch: 8  Loss: tf.Tensor(0.37763134, shape=(), dtype=float32)  Accuracy: t
f.Tensor(85.236046, shape=(), dtype=float32)
Epoch: 9  Loss: tf.Tensor(0.35612142, shape=(), dtype=float32)  Accuracy: t
f.Tensor(84.80686, shape=(), dtype=float32)
Epoch: 10  Loss: tf.Tensor(0.26067603, shape=(), dtype=float32)  Accuracy: t
f.Tensor(89.61374, shape=(), dtype=float32)
```

In [12]:

```
plt.plot(range(1,EPOCHS+1), train_loss_CNN, label='train loss')
plt.plot(range(1,EPOCHS+1), train_accuracy_CNN, label='train accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.title('Accuracy and Loss of CNN')
plt.show()
```



In [13]:

```
predictions_CNN = np.argmax(model_CNN(test_im),axis=1)
```

In [14]:

```
correct = predictions_CNN == test_labels
print(sum(correct)/len(correct))
```

```
0.8213740458015267
```

fully connected NN Model

In [15]:

```
#fully connected --> erst flatten, dann dense
# we use a fully connected neural network
# connect every neuron in one layer to every neuron in the other layer.

class fully_c_Model(Model):
    def __init__(self):
        super(fully_c_Model, self).__init__()
        self.flatten = Flatten() #input layer
        self.d0 = Dense(2330, activation='relu') #2330 = number of weights
        self.d1 = Dense(1165, activation='relu')
        self.d2 = Dense(582, activation='relu')
        self.d3 = Dense(291, activation='relu')
        self.d4 = Dense(146, activation='relu')
        self.d5 = Dense(2, activation='softmax') #output layer

    def call(self, x):
        x = self.flatten(x)
        x = self.d0(x)
        x = self.d1(x)
        x = self.d2(x)
        x = self.d3(x)
        x = self.d4(x)
        x = self.d5(x)
        return x

model_fully_c = fully_c_Model()
```

In [16]:

```
optimizer = tf.keras.optimizers.Adam()
```

In [17]:

```
@tf.function #this decorator converts the function into a graph.
def train_step(images, labels):
    '''function to train our model and computes the loss and gradients'''
    with tf.GradientTape() as tape:
        #tf.GradientTape() is a high-level API that is used to compute differentiations
        predictions = model_fully_c(images)
        loss = loss_object(labels, predictions)

        gradients = tape.gradient(loss, model_fully_c.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model_fully_c.trainable_variables))

        train_loss(loss)
        train_accuracy(labels, predictions)
```

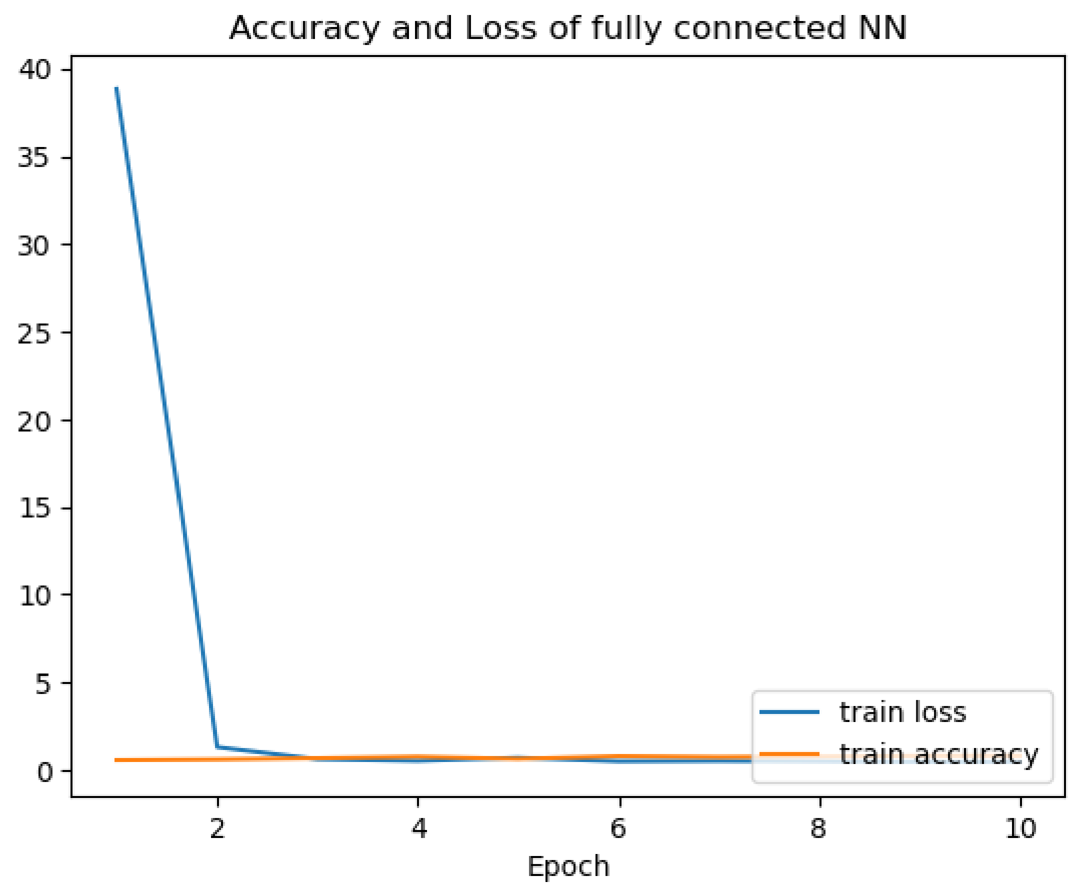
In [18]:

```
#build and train our model
EPOCHS = 10
train_loss_fully_c = []
train_accuracy_fully_c = []

for epoch in range(EPOCHS):
    for images, labels in train_ds:

        train_step(images, labels)

    model_fully_c.save_weights('./content', save_format='tf')
    train_loss_fully_c.append(float(train_loss.result()))
    train_accuracy_fully_c.append(float(train_accuracy.result()))
    print('Epoch:',str(epoch+1),' Loss:',str(train_loss.result()),' Accuracy:', str(train_a
    train_loss.reset_states()
    train_accuracy.reset_states()
```

```
Epoch: 1  Loss: tf.Tensor(38.834263, shape=(), dtype=float32)  Accuracy: tf.
Tensor(57.25322, shape=(), dtype=float32)
Epoch: 2  Loss: tf.Tensor(1.3143337, shape=(), dtype=float32)  Accuracy: tf.
Tensor(61.03004, shape=(), dtype=float32)
Epoch: 3  Loss: tf.Tensor(0.6259637, shape=(), dtype=float32)  Accuracy: tf.
Tensor(68.32618, shape=(), dtype=float32)
Epoch: 4  Loss: tf.Tensor(0.5317937, shape=(), dtype=float32)  Accuracy: tf.
Tensor(76.48069, shape=(), dtype=float32)
Epoch: 5  Loss: tf.Tensor(0.70797074, shape=(), dtype=float32)  Accuracy: t
f.Tensor(64.377686, shape=(), dtype=float32)
Epoch: 6  Loss: tf.Tensor(0.50524217, shape=(), dtype=float32)  Accuracy: t
f.Tensor(79.570816, shape=(), dtype=float32)
Epoch: 7  Loss: tf.Tensor(0.5253344, shape=(), dtype=float32)  Accuracy: tf.
Tensor(74.67811, shape=(), dtype=float32)
Epoch: 8  Loss: tf.Tensor(0.50063676, shape=(), dtype=float32)  Accuracy: t
f.Tensor(77.16738, shape=(), dtype=float32)
Epoch: 9  Loss: tf.Tensor(0.46883667, shape=(), dtype=float32)  Accuracy: t
f.Tensor(81.88841, shape=(), dtype=float32)
Epoch: 10  Loss: tf.Tensor(0.47124588, shape=(), dtype=float32)  Accuracy: t
f.Tensor(82.06009, shape=(), dtype=float32)
```

In [43]:

```
plt.plot(range(1,EPOCHS+1), train_loss_fully_c, label='train loss')
plt.plot(range(1,EPOCHS+1), train_accuracy_fully_c, label='train accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.title('Accuracy and Loss of fully connected NN')
plt.show()
```



In [21]:

```
predictions_fully_c = np.argmax(model_fully_c(test_im),axis=1)
```

In [22]:

```
correct = predictions_fully_c == test_labels
print(sum(correct)/len(correct))
```

```
0.6580152671755726
```

shallow NN model 3

In [23]:

```
#from tensorflow.keras.layers import Embedding, LSTM
#lstm = tf.keras.layers.LSTM(4)
```

In [24]:

```
class shallow_Model(Model):
    def __init__(self):
        super(shallow_Model, self).__init__()
        self.flatten = Flatten()
        self.d1 = Dense(2000, activation='relu')
        self.d2 = Dense(2, activation='softmax')


    def call(self, x):
        x = self.flatten(x)
        x = self.d1(x)
        x = self.d2(x)
        return x

model_shallow = shallow_Model()
```

In [25]:

```
optimizer = tf.keras.optimizers.Adam()
```

In [26]:

```
@tf.function #this decorator converts the function into a graph.
def train_step(images, labels):
    '''function to train our model and computes the loss and gradients'''
    with tf.GradientTape() as tape:
        #tf.GradientTape() is a high-level API that is used to compute differentiations
        predictions = model_shallow(images)
        loss = loss_object(labels, predictions)

        gradients = tape.gradient(loss, model_shallow.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model_shallow.trainable_variables))

        train_loss(loss)
        train_accuracy(labels, predictions)
```

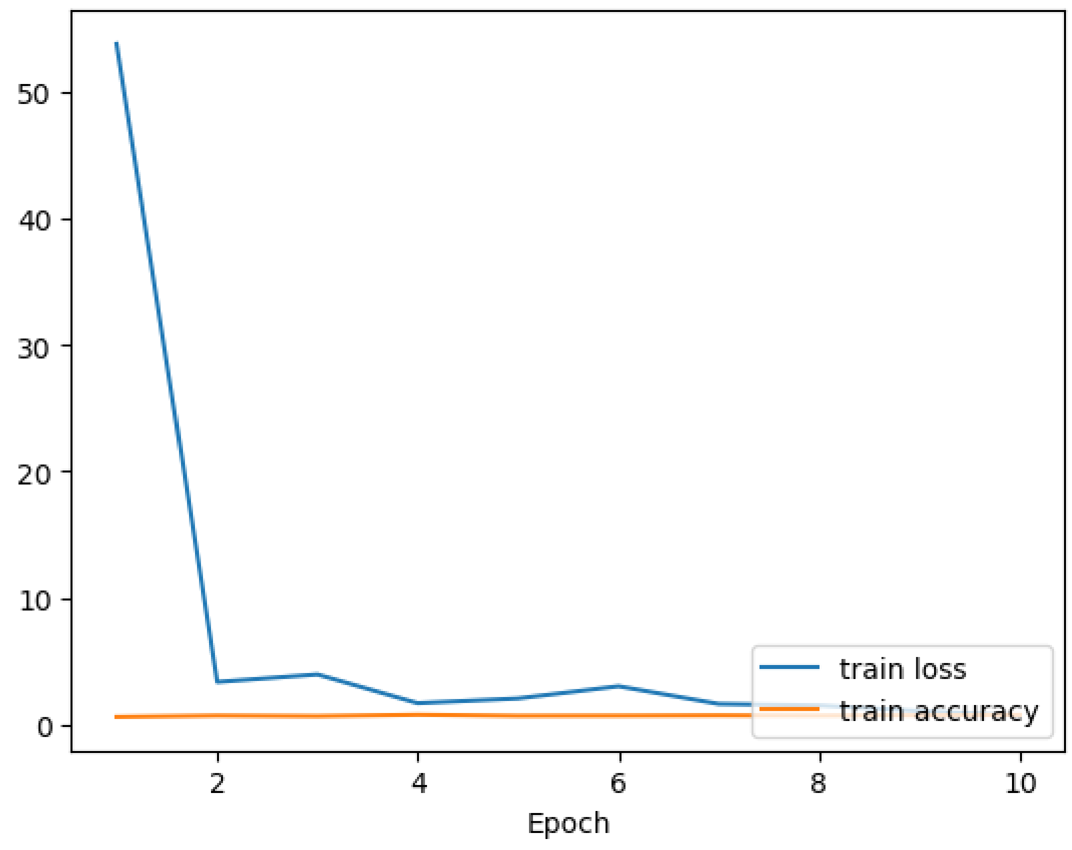
In [27]:

```
#build and train our model
EPOCHS = 10
train_loss_shallow = []
train_accuracy_shallow = []

for epoch in range(EPOCHS):
    for images, labels in train_ds:
        train_step(images, labels)

    model_shallow.save_weights('./content', save_format='tf')
    train_loss_shallow.append(float(train_loss.result()))
    train_accuracy_shallow.append(float(train_accuracy.result()))
    print('Epoch:',str(epoch+1),' Loss:',str(train_loss.result()),' Accuracy:', str(train_a
    train_loss.reset_states()
    train_accuracy.reset_states()
```

```
Epoch: 1  Loss: tf.Tensor(53.79301, shape=(), dtype=float32)  Accuracy: tf.T
ensor(59.39914, shape=(), dtype=float32)
Epoch: 2  Loss: tf.Tensor(3.3569958, shape=(), dtype=float32)  Accuracy: tf.
Tensor(69.18455, shape=(), dtype=float32)
Epoch: 3  Loss: tf.Tensor(3.9524095, shape=(), dtype=float32)  Accuracy: tf.
Tensor(65.23605, shape=(), dtype=float32)
Epoch: 4  Loss: tf.Tensor(1.6777343, shape=(), dtype=float32)  Accuracy: tf.
Tensor(76.051506, shape=(), dtype=float32)
Epoch: 5  Loss: tf.Tensor(2.0512567, shape=(), dtype=float32)  Accuracy: tf.
Tensor(67.55364, shape=(), dtype=float32)
Epoch: 6  Loss: tf.Tensor(3.018675, shape=(), dtype=float32)  Accuracy: tf.T
ensor(68.75536, shape=(), dtype=float32)
Epoch: 7  Loss: tf.Tensor(1.6172634, shape=(), dtype=float32)  Accuracy: tf.
Tensor(71.1588, shape=(), dtype=float32)
Epoch: 8  Loss: tf.Tensor(1.4986204, shape=(), dtype=float32)  Accuracy: tf.
Tensor(68.669525, shape=(), dtype=float32)
Epoch: 9  Loss: tf.Tensor(1.0597134, shape=(), dtype=float32)  Accuracy: tf.
Tensor(71.75966, shape=(), dtype=float32)
Epoch: 10  Loss: tf.Tensor(0.7079052, shape=(), dtype=float32)  Accuracy: t
f.Tensor(75.36481, shape=(), dtype=float32)
```

In [28]:

```
plt.plot(range(1,EPOCHS+1), train_loss_shallow, label='train loss')
plt.plot(range(1,EPOCHS+1), train_accuracy_shallow, label='train accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.show()
```



In [29]:

```
predictions_shallow = np.argmax(model_shallow(test_im),axis=1)
```

In [30]:

```
correct = predictions_shallow == test_labels
print(sum(correct)/len(correct))
```

```
0.6442748091603053
```

Evaluation

In [31]:

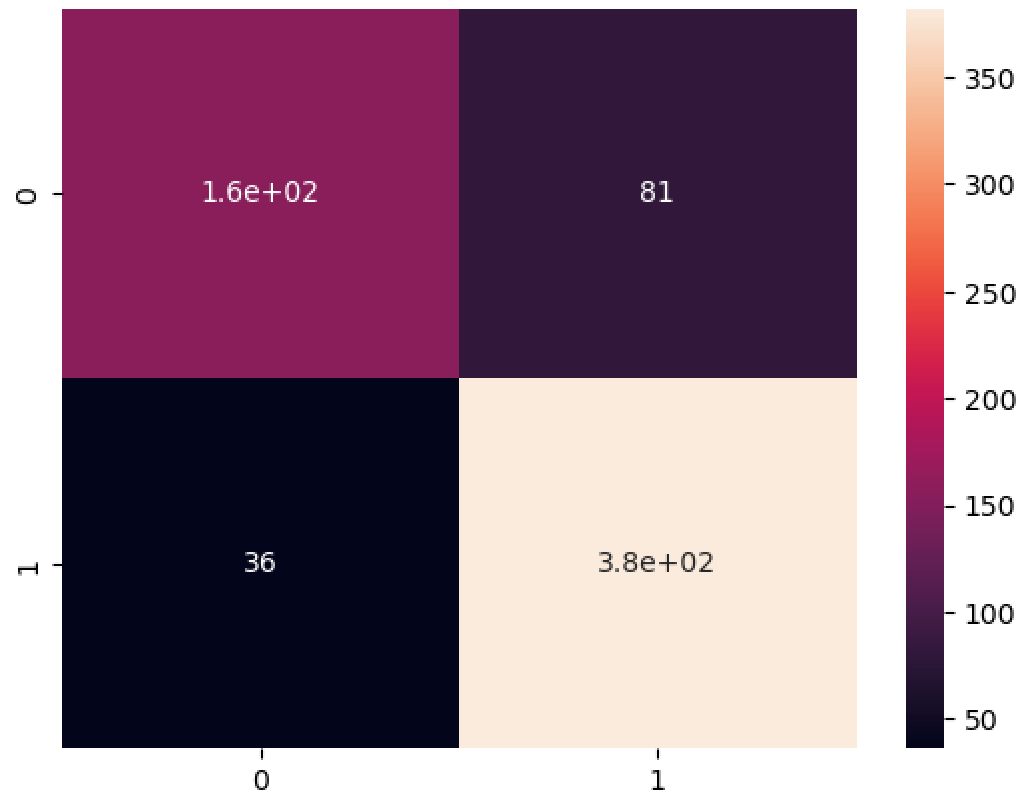
```
from sklearn.metrics import confusion_matrix
# Define function to calculate multi-class sensitivity, specificity, and confusion matrix
# Based on the link: https://towardsdatascience.com/multi-class-classification-extracting-p
def matrix(y_test, y_pred):
    cnf_matrix = confusion_matrix(y_test, y_pred)
    FP = cnf_matrix.sum(axis=0) - np.diag(cnf_matrix)
    FN = cnf_matrix.sum(axis=1) - np.diag(cnf_matrix)
    TP = np.diag(cnf_matrix)
    TN = cnf_matrix.sum() - (FP + FN + TP)
    FP = FP.astype(float)
    FN = FN.astype(float)
    TP = TP.astype(float)
    TN = TN.astype(float)

    # Sensitivity, hit rate, recall, or true positive rate
    TPR = round((TP/(TP+FN)).mean(),2)
    # Specificity or true negative rate
    TNR = round((TN/(TN+FP)).mean(),2)

    return TPR, TNR, cnf_matrix
```

```
In [32]:
```

```
# Confusion matrix for CNN
sns.heatmap(matrix(test_labels, predictions_CNN)[2], annot=True)
```

```
Out[32]:
```

```
<AxesSubplot:>
```

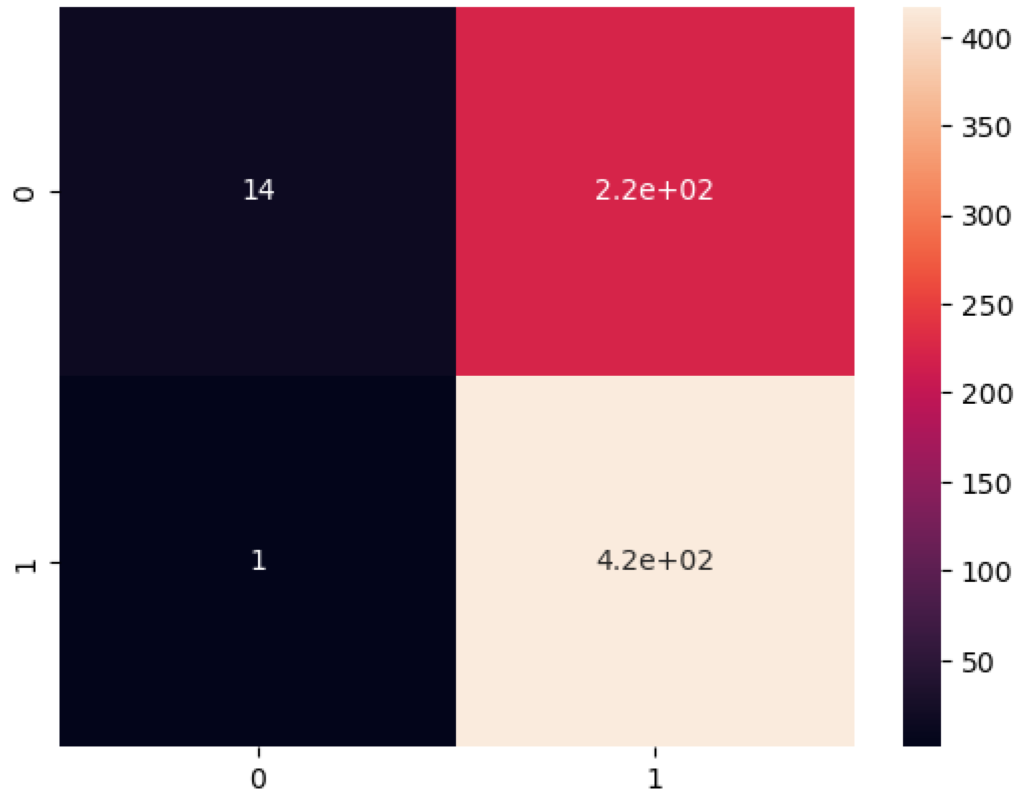


```
In [33]:
```

```python
# Confusion matrix for fully connected NN
sns.heatmap(matrix(test_labels, predictions_fully_c)[2], annot=True)
```

```
Out[33]:
```

```
<AxesSubplot:>
```

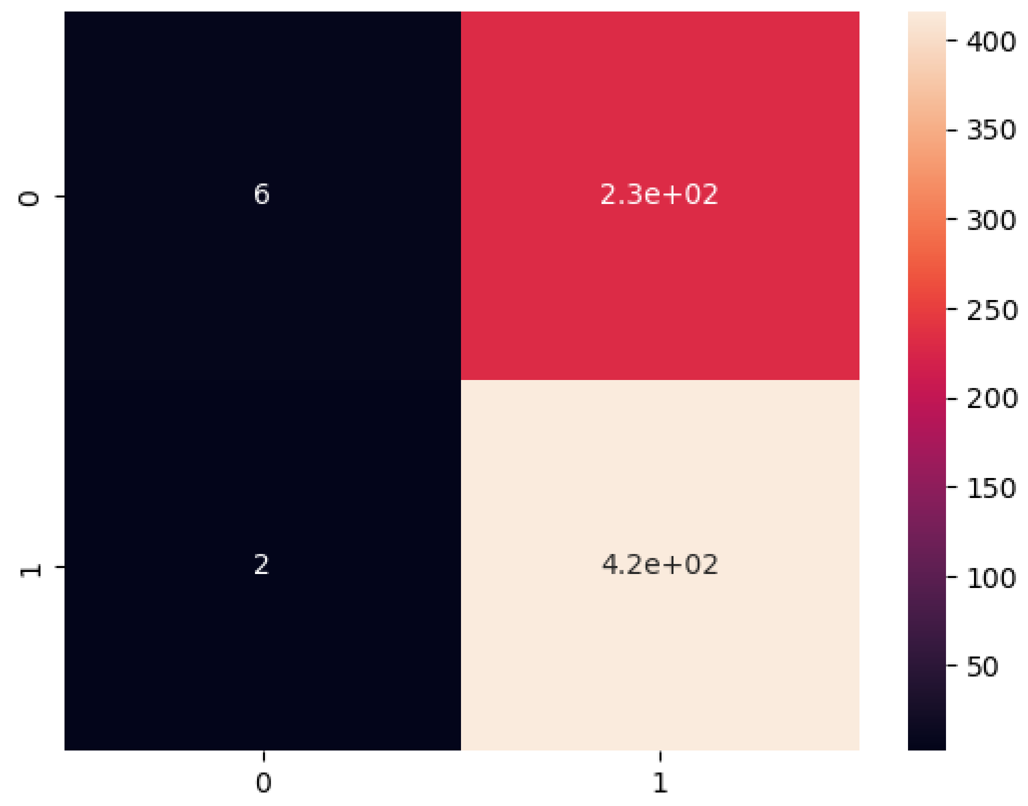


```
In [34]:
```

```python
# Confusion matrix for shallow NN
sns.heatmap(matrix(test_labels, predictions_shallow)[2], annot=True)
```

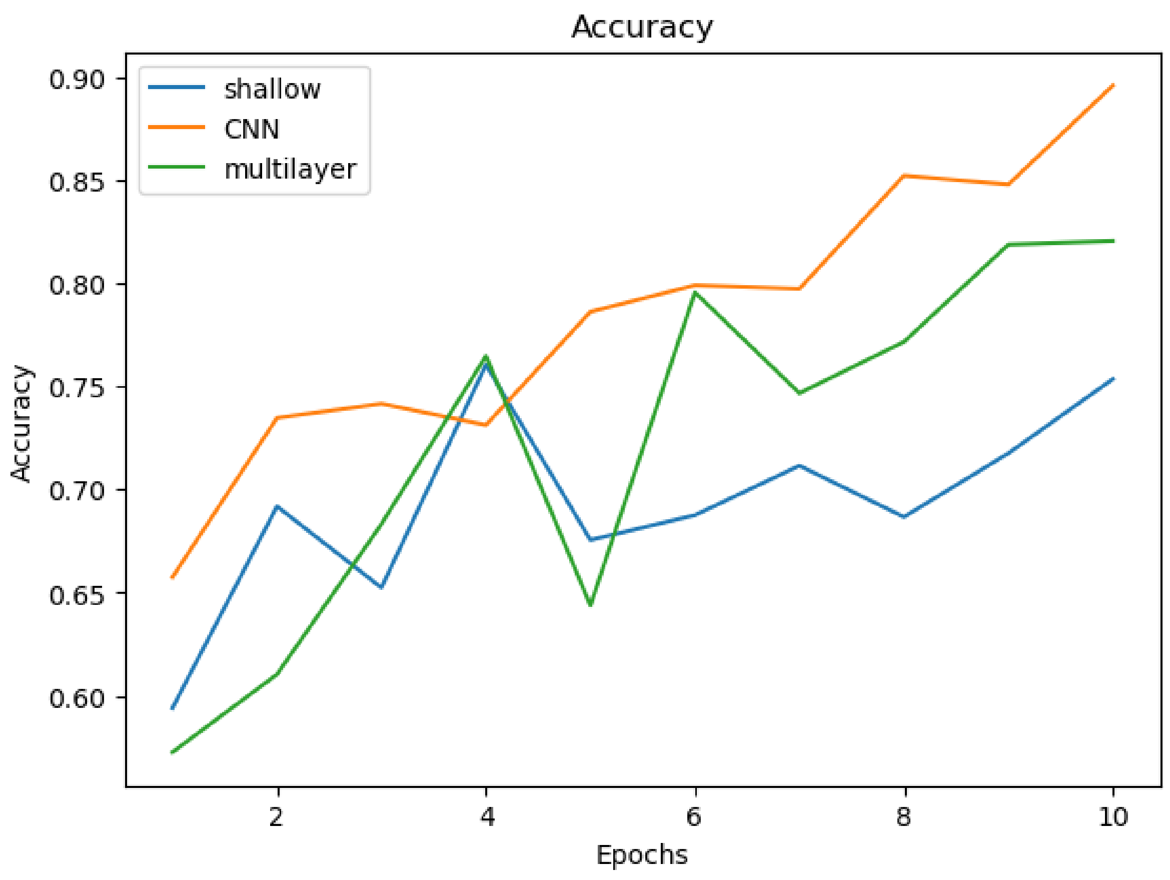
```
Out[34]:
```

```
<AxesSubplot:>
```



In [39]:

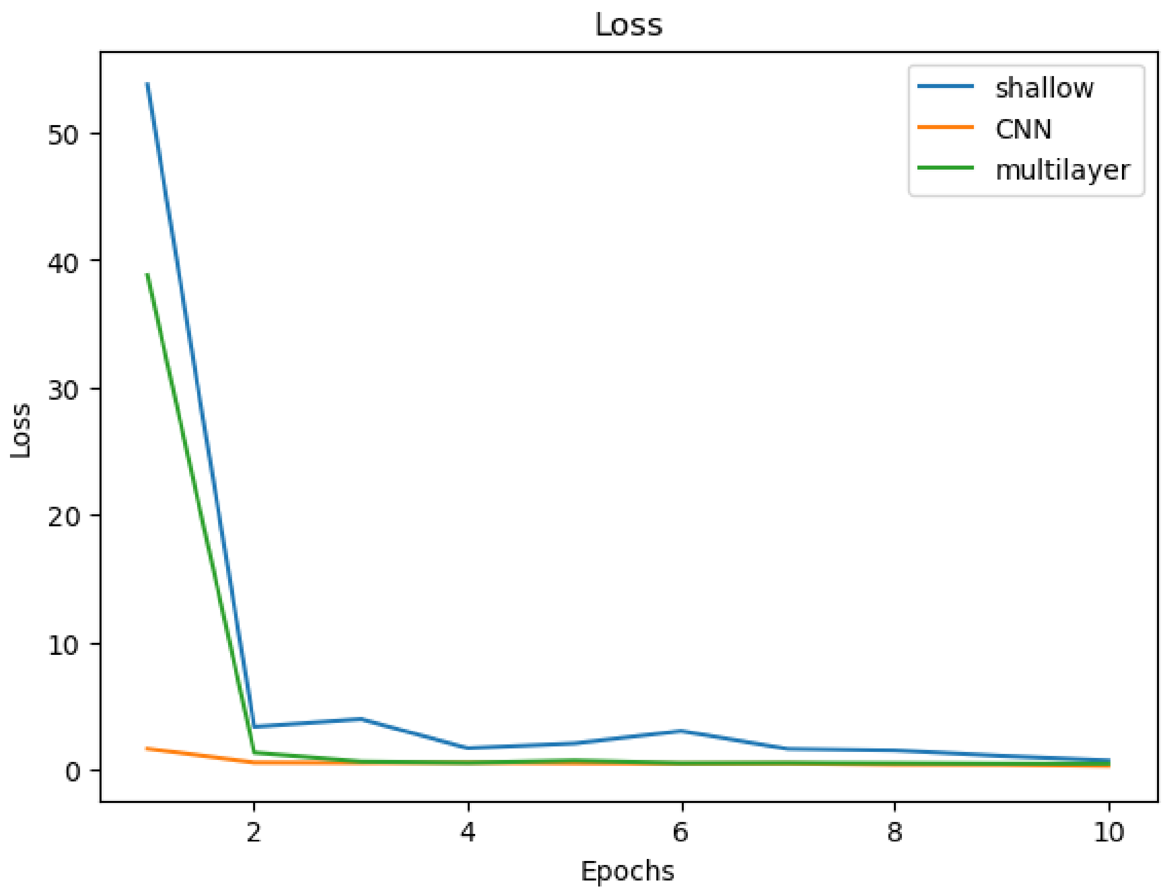
```
plt.figure(figsize=(7,5))
plt.plot(range(1,EPOCHS+1), np.array(train_accuracy_shallow),label='shallow')
plt.plot(range(1,EPOCHS+1), np.array(train_accuracy_CNN),label='CNN')
plt.plot(range(1,EPOCHS+1), np.array(train_accuracy_fully_c),label='multilayer')

#plt.ylim(50, 90)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy')
plt.legend()
plt.show()
plt.savefig('Accuracy.png', bbox_inches='tight')
```



```
<Figure size 640x480 with 0 Axes>
```

In [37]:

```
plt.figure(figsize=(7,5))
plt.plot(range(1,EPOCHS+1), np.array(train_loss_shallow),label='shallow')
plt.plot(range(1,EPOCHS+1), np.array(train_loss_CNN),label='CNN')
plt.plot(range(1,EPOCHS+1), np.array(train_loss_fully_c),label='multilayer')

#plt.ylim(50, 90)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss')
plt.legend()
plt.show()
plt.savefig('Loss.png', bbox_inches='tight')
```



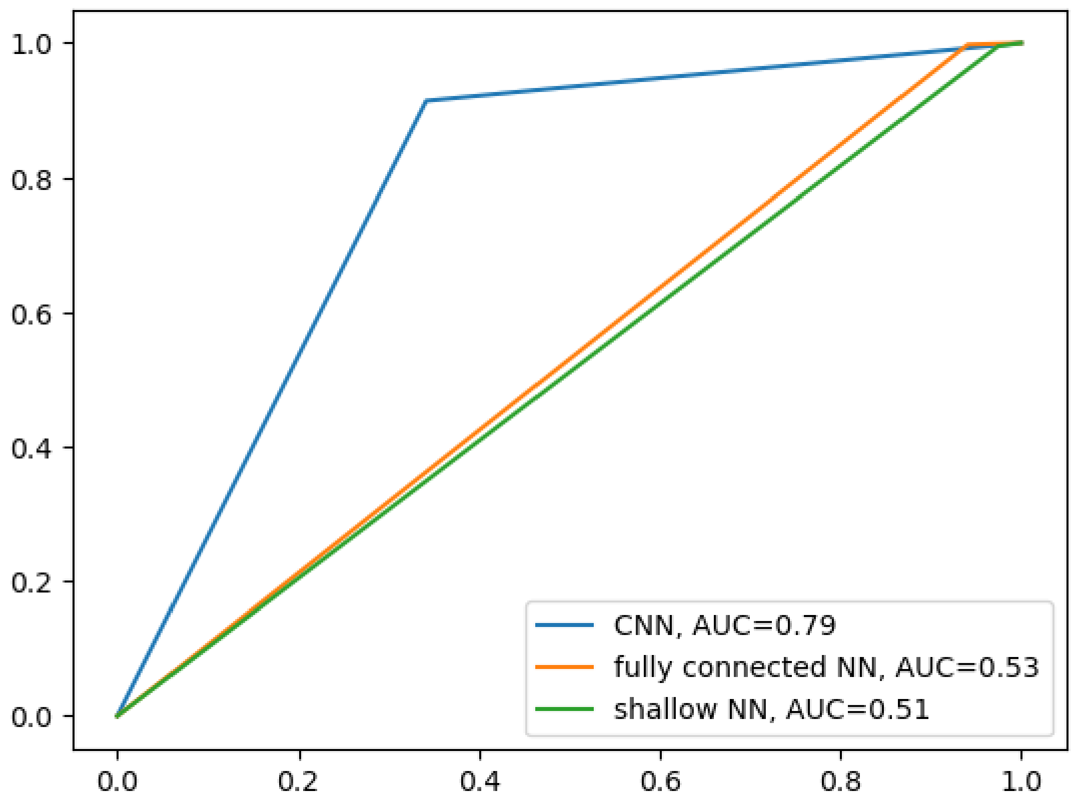
```
<Figure size 640x480 with 0 Axes>
```

In [40]:

```
# ROC curves
from sklearn.metrics import roc_curve, RocCurveDisplay, roc_auc_score
classifiers = [predictions_CNN, predictions_fully_c, predictions_shallow]
labels = ["CNN", "fully connected NN", "shallow NN"]

#set up plotting area
plt.figure(0).clf()

# Plot ROC curve for each classifier
for i,j in zip(classifiers, labels):
    fpr, tpr, _ = roc_curve(test_labels, i)
    auc = round(roc_auc_score(test_labels, i), 2)
    plt.plot(fpr,tpr,label=j+", AUC="+str(auc))

#add legend
plt.legend()
```

Out[40]:

```
<matplotlib.legend.Legend at 0x2bc9fbb7be0>
```



In [ ]:

```

```