Mustererkennung/Machine Learning - Assignment 9

Load the spam dataset:

In [1]:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
```

In [2]:

```
path_to_train = 'zip.train'
path_to_test = 'zip.test'
training_data = np.array(pd.read_csv(path_to_train, sep=' ', header=None))
test_data = np.array(pd.read_csv(path_to_test, sep =' ',header=None))

X_train, y_train = training_data[:,1:-1], training_data[:,0]
X_test, y_test = test_data[:,1:], test_data[:,0]
```

1. Add Backpropagation to your MLP and train the model on the ZIP-Dataset

In [3]:

```
def standardize(x):
    return (x - np.mean(x, axis=0)) / np.std(x, axis=0)

def to_one_hot(y, num_classes):
    return np.eye(num_classes)[y.astype(np.uint)]

def softmax(x):
    exp = np.exp(x)
    summed = np.sum(exp, axis=1, keepdims=True)
    return exp / summed

def accuracy(y_true, y_pred):
    y_true = np.argmax(y_true, axis=1)
    y_pred = np.argmax(y_pred, axis=1)
    return np.mean(y_true == y_pred)

class Sigmoid():
    def forward(self, x):
        z = 1 / (1 + np.exp(-x))
        self.z = z
        return z

    def backward(self, right_dev):
        return self.z * (1.0 - self.z) * right_dev.T

class CategoricalCrossEntropy():
    """
    softmax + categorical crossentropy
    """
    def forward(self, x, y):
        y_pred = softmax(x)
        loss = (1.0 / x.shape[0]) * -np.sum(np.log(np.max(y_pred * y, axis=1)))
        self.y_pred = y_pred
        self.y_true = y
        return loss

    def backward(self, right_dev):
        return (self.y_pred - self.y_true) * right_dev

class DenseLayer():
    def __init__(self, input_dims, output_dims, activation=None):
        self.w = np.random.normal(size=(input_dims, output_dims))
        self.b = np.ones((1, output_dims))
        self.activation = activation

    def forward(self, x):
        self.x = x
        x = x @ self.w + self.b
        if self.activation:
            x = self.activation.forward(x)
        return x

    def backward(self, right_dev):
        if self.activation:
            act_grad = self.activation.backward(right_dev)
        else:
            act_grad = right_dev
        self.grad_w = self.x.T @ act_grad
        self.grad_b = np.sum(act_grad, axis=0, keepdims=True)
```

```
        return self.w @ act_grad.T

    def apply_gradients(self, lr):
        self.w = self.w - lr * self.grad_w
        self.b = self.b - lr * self.grad_b

class NeuralNetwork():
    def __init__(self, layers, loss):
        self.layers = layers
        self.loss = loss

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def predict(self, x):
        return softmax(self.forward(x))

    def fit(self, x, y, epochs, lr, x_val = None, y_val = None):
        losses = []
        train_accs = []
        val_accs = []

        for epoch in range(1, epochs + 1):
            y_pred = self.forward(x)
            loss = self.loss.forward(y_pred, y)
            right_dev = 1.0
            right_dev = self.loss.backward(right_dev)
            for layer in self.layers[::-1]:
                right_dev = layer.backward(right_dev)
            for layer in self.layers:
                layer.apply_gradients(lr)

            if isinstance(x_val, np.ndarray) and isinstance(y_val, np.ndarray):
                y_pred_val = self.predict(x_val)
                val_accs.append(accuracy(y_val, y_pred_val))

            train_accs.append(accuracy(y, softmax(y_pred)))
            losses.append(loss)

        return losses, train_accs, val_accs
```

In [4]:

```
num_classes = len(np.unique(y_train))
y_train, y_test = to_one_hot(y_train, num_classes), to_one_hot(y_test, num_classes)
X_train, X_test = standardize(X_train), standardize(X_test)
```

In [5]:

```
best_acc = 0.0

for width in range(10, 50, 10):
    for depth in range(0, 4):
        first_layer = DenseLayer(X_train.shape[1], width, activation=Sigmoid())
        layers = [first_layer]
        for i in range(1, depth - 1):
            layers.append(DenseLayer(width, width, activation=Sigmoid()))
        layers.append(DenseLayer(width, num_classes))
        clf = NeuralNetwork(layers, CategoricalCrossEntropy())
        losses, train_accs, val_accs = clf.fit(X_train, y_train, epochs=500, lr=0.0
        print(f"For a depth of {depth + 2} and a width of {width} the network has a
        if train_accs[-1] > best_acc:
            best_acc = train_accs[-1]
            best_settings = {
                "width": width,
                "depth": depth + 2,
                "losses": losses,
                "train_accs": train_accs,
                "val_accs": val_accs,
                "clf": clf
            }
```

```
For a depth of 2 and a width of 10 the network has an accuracy of 0.80
67
For a depth of 3 and a width of 10 the network has an accuracy of 0.79
19
For a depth of 4 and a width of 10 the network has an accuracy of 0.81
17
For a depth of 5 and a width of 10 the network has an accuracy of 0.74
49
For a depth of 2 and a width of 20 the network has an accuracy of 0.85
32
For a depth of 3 and a width of 20 the network has an accuracy of 0.85
82
For a depth of 4 and a width of 20 the network has an accuracy of 0.86
64
For a depth of 5 and a width of 20 the network has an accuracy of 0.83
23
For a depth of 2 and a width of 30 the network has an accuracy of 0.87
81
For a depth of 3 and a width of 30 the network has an accuracy of 0.87
66
For a depth of 4 and a width of 30 the network has an accuracy of 0.87
72
For a depth of 5 and a width of 30 the network has an accuracy of 0.86
97
For a depth of 2 and a width of 40 the network has an accuracy of 0.89
14
For a depth of 3 and a width of 40 the network has an accuracy of 0.89
75
For a depth of 4 and a width of 40 the network has an accuracy of 0.88
82
For a depth of 5 and a width of 40 the network has an accuracy of 0.88
59
```

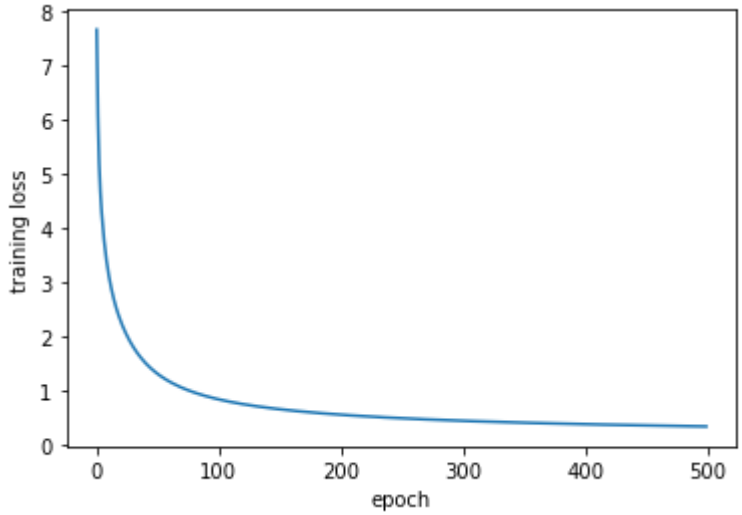
In [6]:

```
print(f"The best architecture we found is {best_settings['depth']} layers with {bes
```

```
The best architecture we found is 3 layers with 40 neurons, which has
an accuracy of 0.8975.
```

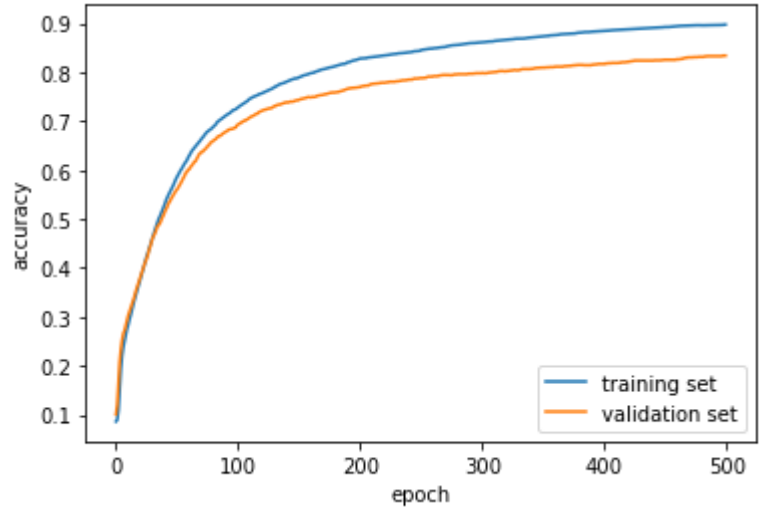
In [7]:

```
plt.plot(list(range(len(best_settings["losses"]))), best_settings["losses"])
plt.xlabel("epoch")
plt.ylabel("training loss")
plt.show()
```



In [8]:

```
plt.plot(list(range(len(best_settings["train_accs"]))), best_settings["train_accs"]
plt.plot(list(range(len(best_settings["val_accs"]))), best_settings["val_accs"], la
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.legend()
plt.show()
```



b) Show some digits that are classified incorrectly

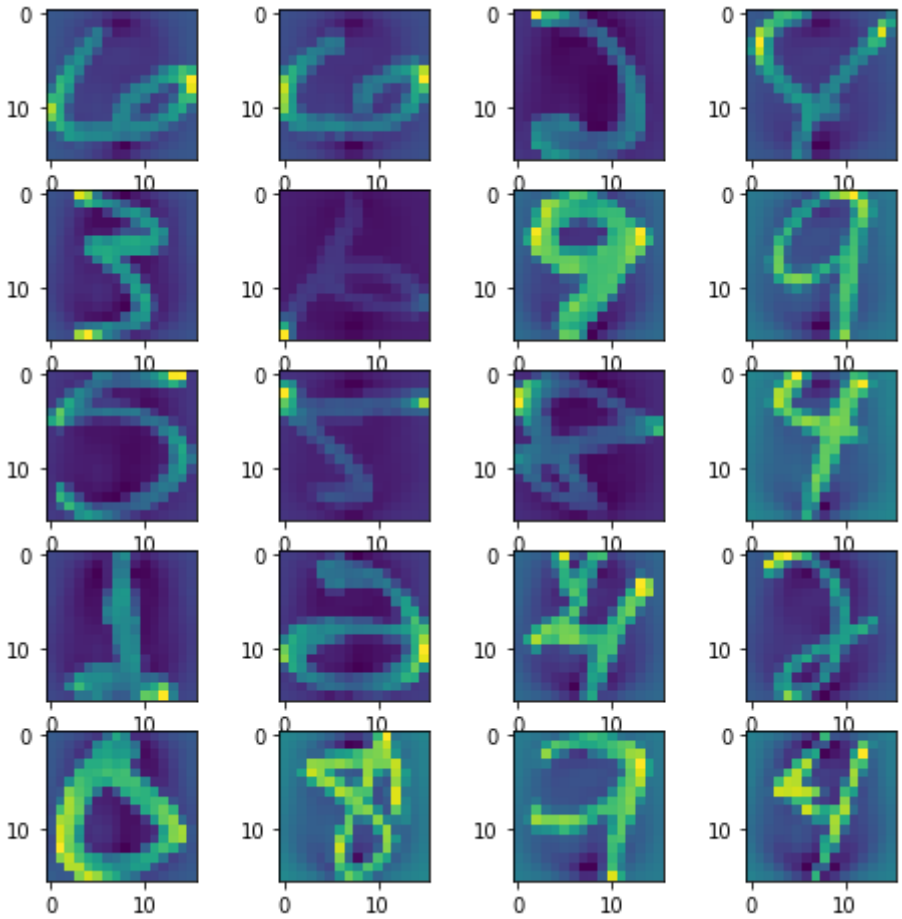
In [9]:

```
y_pred = best_settings["clf"].predict(X_test)

y_pred = np.argmax(y_pred, axis=1)
y_test = np.argmax(y_test, axis=1)
X_missclassified = X_test[y_pred != y_test]
```

In [10]:

```
w=10
h=10
fig=plt.figure(figsize=(8, 8))
columns = 4
rows = 5
for i in range(1, columns*rows +1):
    img = np.reshape(X_missclassified[i - 1], (16, 16))
    fig.add_subplot(rows, columns, i)
    plt.imshow(img)
plt.show()
```



c) Plot your first weight layer as a grayscale image.

In [11]:

```
plt.imshow(best_settings["clf"].layers[0].w, cmap='gray')
plt.show()
```