Assignment 1

Ilian Ariesen & Johannes Schillberg

In this Assignment we define and train an Autoencoder. This Autoencoder encodes a one-hot-encoded vector of shape $ \mathsf{R}^{1\times8} $ into a compressed space pof shape $ \mathsf{R}^{1\times3} $ then decodes it back into the original shape $ \mathsf{R}^{1\times8} $. Additionally, we implement the backpropagation algorithm and compare batch gradient descent with stochastic or mini-batch gradient descent with different hyperparameters.

```
import numpy as np
import matplotlib.pyplot as plt
```

We chose numpy arrays for the Weights of Network to utilize the fast Vector Operations build into numpy. We the then decided to store the different layer weights into a list instead of stacking them into a larger numpy ndarray because doing that makes things more complicated.

```
# Initialize Network
def initializeNetworkWeights(scale=0.01):
    W = [np.random.random((3, 8)), np.random.random((8, 3))]
    W = [(x - np.ones(x.shape) * 0.5) * scale for x in W]
    b = [np.random.random(3), np.random.random(8)]
    b = [(x - np.ones(x.shape) * 0.5) * scale for x in b]
    return W, b
```

We chose to use the sigmoid function as our activation function and the half sum squared as our error. Alternatives could have been using tanh as an activation function and the cross entropy loss as the error. For the cross entropy loss you would then also have to use the softmax activation function in the output layer.

```
# Useful functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoidPrime(x):
    return sigmoid(x) * (1 - sigmoid(x))


def half_sum_squared(X, Y):
    return np.linalg.norm(X - Y) ** 2 / 2


def forward(X, W, b):
    """
    :return: the final output and the z's of the layers after the
```

```
input
    """
    a = X
    z_list = []
    for w, b in zip(W, b):
        z = np.matmul(w, a.T).T + b
        z_list.append(z)
        a = sigmoid(z)
    return a, z_list
```

Next we get the contribution of the specific weights in each layer to the final error, also called delta's. And then update the original weights in the network using the average over the provided batch.

```

# Getting Deltas
def getDeltas(z_list, Y, W):
    delta_list = []
    for i in reversed(range(len(z_list))):
        if i == len(z_list) - 1:
            z = z_list[i]
            a = sigmoid(z)
            delta = -(Y - a) * sigmoidPrime(z)
            delta_list.append(delta)
        else:
            z = z_list[i]
            delta = np.matmul(delta_list[len(delta_list) - 1], W[i +
1]) * sigmoidPrime(z)
            # delta = np.matmul(W[i + 1].T, delta_list[len(delta_list)
- 1]).T*sigmoidPrime(z)
            delta_list.append(delta)
    return delta_list


# Backpropagate
def backprop(X, Y, W, b, learning_rate=0.03, weightDecay=0.01):
    a, zlist = forward(X, W, b)
    delta_list = getDeltas(zlist, Y, W)
    # Dont average here!!
    # delta_list = [np.average(x, axis=0) for x in delta_list]
    delta_list = list(reversed(delta_list))
    a_list = [sigmoid(x) for x in zlist]
    a_list.insert(0, np.array(X))
    a_list = [x.reshape(x.shape[0], 1, x.shape[1]) for x in a_list]
    delta_list = [x.reshape(x.shape[0], x.shape[1], 1) for x in
delta_list]
    for i in range(len(delta_list)):
        derivativeW = a_list[i] * delta_list[i]
        derivativeW = np.average(derivativeW, axis=0)
        # Average here!!
        W[i] = W[i] - learning_rate * (derivativeW + W[i] *
(weightDecay / len(X)))
        b[i] = b[i] - learning_rate * np.average(delta_list[i],
axis=0).reshape(-1)
    return W, b
```

Next we define two different Functions combining everything we defined above into the actual gradient descent algorithm. For that we differentiate between a whole batch gradient descent and the stochastic or mini batch gradient descent.

```
def batch_GD(X, Y, epoch=20000, learning_rate=0.003, weightDecay=0.1,
scale=0.1):
    W, b = initializeNetworkWeights(scale)
    a, zlist = forward(X, W, b)
    for i in range(epoch):
        W, b = backprop(np.copy(X), Y, W, b, learning_rate,
weightDecay)
        a, z_list = forward(X, W, b)
    return W, b


def stochastic_GD(X, Y, epoch=20000, learning_rate=0.003,
weightDecay=0.01, scale=0.1, sample_size=1):
    W, b = initializeNetworkWeights(scale)
    a, zlist = forward(X, W, b)
    for i in range(epoch):
        random_indices = np.random.choice(X.shape[0],
size=sample_size, replace=False)
        W, b = backprop(X[random_indices], Y[random_indices], W, b,
learning_rate, weightDecay)
        a, z_list = forward(X, W, b)
    return W, b
```

Experiments

First define the different Parameters we want to test.

```
random_scales = [0.01, 0.1, 5]
learning_rates = [0.03, 0.3, 3]
weight_decays = [0.00003, 0.0003, 0.03]
epochs = [20, 200, 2000, 20000, 200000]
sample_sizes = [1, 3, 5, 8]
```

Next to the half sum squared error we also want to know the accuracy of the network, for that we take the argmax of the networks output and compare it to the orignal data.

```
def accuracy(a, Y):
    y_pred = np.argmax(a, axis=1)
    Y_true = np.argmax(Y, axis=1)
    TP = np.sum(y_pred == Y_true)
    return TP / len(Y)
```

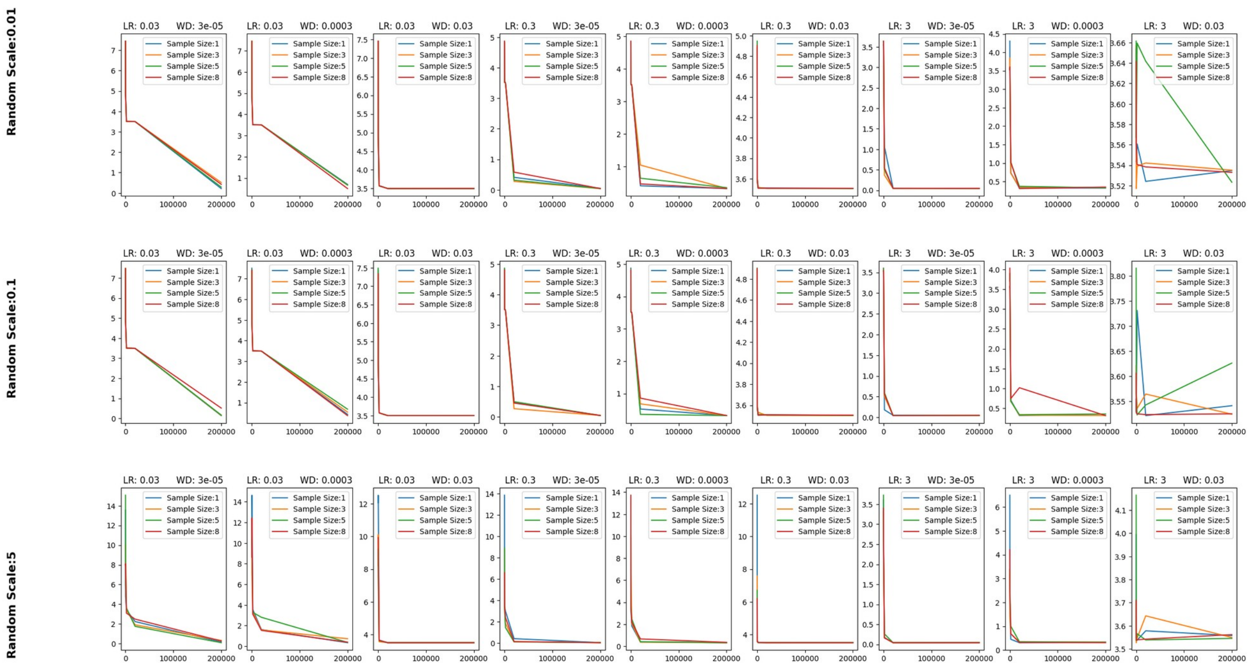
Then we gather our data for each combination of hyper parameters and safe them in a list.

```
results = []
X = np.zeros((8, 8))
for i in range(8):
    X[i, i] = 1
Y = np.array(X)
for scale in random_scales:
    result_scale = []
    for learning_rate in learning_rates:
        result_learning_rate = []
        for weight_decay in weight_decays:
            result_weight_decay_ = []
            for sample in sample_sizes:
                result_sample = []
                for epoch in epochs:
                    W, b = stochastic_GD(X, Y, epoch=epoch,
learning_rate=learning_rate, weightDecay=weight_decay,
                                         scale=scale)
                    a, zlist = forward(X, W, b)
                    half_sum_squared_error = half_sum_squared(a, Y)
                    accuracy_error = accuracy(a, Y)
                    result_sample.append([accuracy_error,
half_sum_squared_error])
                result_weight_decay_.append(result_sample)
            result_learning_rate.append(result_weight_decay_)
        result_scale.append(result_learning_rate)
    results.append(result_scale)
```

And now we will need to visualize the results. For that we create 27 Plots with the $ \verb| random_scale|$, $ \verb|learning_rate|$ and $ \verb|weight_decay|$ fixed.

```
import math

fig, axes = plt.subplots(3, 9, figsize=(27, 15))
fig.subplots_adjust(hspace=0.4)
for i in range(3):
    title = f"Random Scale:{random_scales[i]}"
    y_coord = 1 - (i + 0.5) / 3
    fig.text(0.05, y_coord, title, va='center', ha='center',
fontsize=16, weight='bold', rotation=90)
    for j in range(9):
        learning_rate_index = int(j / 3)
        weight_decay_index = int(j % 3)
        unique_title = f"LR: {learning_rates[learning_rate_index]}
WD: {weight_decays[weight_decay_index]}"
        axes[i][j].set_title(unique_title)
        result = results[i][learning_rate_index][weight_decay_index]
        for k in range(len(sample_sizes)):
            sample_size_result = result[k]
            axes[i][j].plot(epochs, np.array(sample_size_result)[:,
1], label=f"Sample Size:{sample_sizes[k]}")
        axes[i][j].legend()
plt.show()
```



We can see for very small Learning Rates it takes a lot of epochs to learn correct weights. Additionally, we can see with a really large Weight decay parameters the error converges somewhere sub optimal. With the Random Scale we can see a light trend in the number of epochs needed to converge.

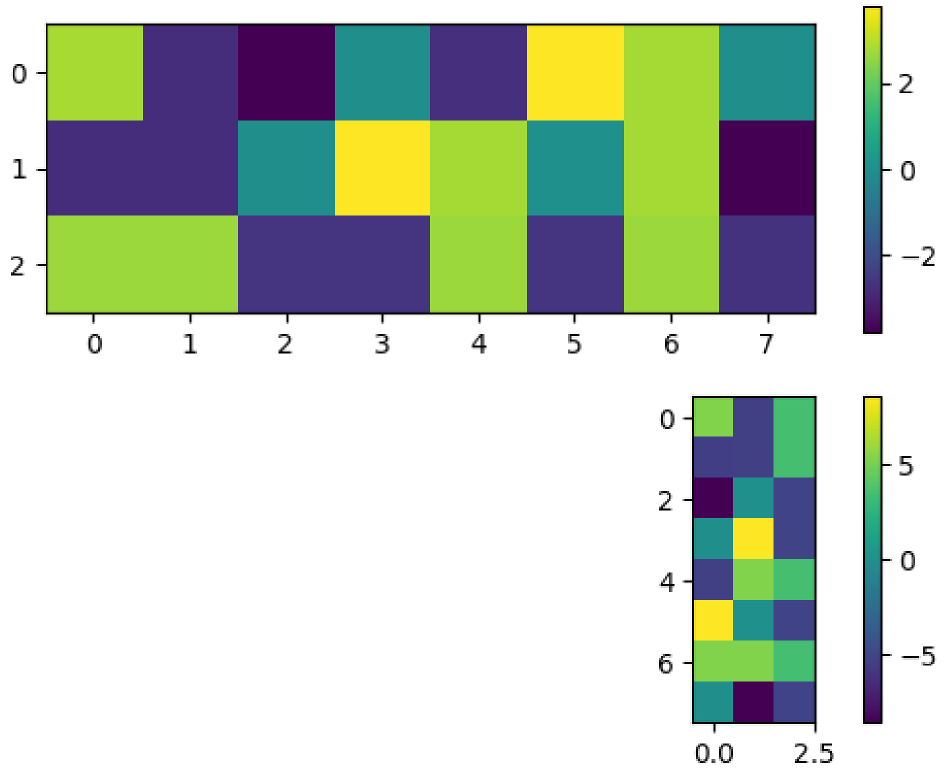
```
X = np.zeros((8, 8))
for i in range(8):
    X[i, i] = 1
Y = np.array(X)
W, b = stochastic_GD(X, Y, epoch=200000, learning_rate=0.003,
weightDecay=1, scale=0.1)
print(W[0])
```

```
[[-2.12947284e-07  1.17041894e-06 -4.99950905e-07  4.10030671e-07
   5.53570234e-07  6.71942984e-07 -1.04209146e-06 -1.14982775e-06]
 [-2.10804737e-07  1.15864297e-06 -4.94920738e-07  4.05905167e-07
   5.48000605e-07  6.65182344e-07 -1.03160660e-06 -1.13825886e-06]
 [-2.06082230e-07  1.13268690e-06 -4.83833447e-07  3.96811960e-07
   5.35724253e-07  6.50280838e-07 -1.00849633e-06 -1.11275930e-06]]
```

From the code above we can see the effects of a large Weight Decay parameter, the values in the first weight layer converge to zero. This is because of the vanishing gradient going deeper in the layers favouring the weight decay when updating the values.

```
W, b = stochastic_GD(X, Y, epoch=2000000, learning_rate=0.3,
weightDecay=0.0001, scale=0.1)

fig, ax = plt.subplots(2)
im1=ax[0].imshow(W[0])
im2=ax[1].imshow(W[1])
fig.colorbar(im1, ax=ax[0])
fig.colorbar(im2, ax=ax[1])
plt.show()
```



```
a, zlist = forward(X, W, b)
print("accuracy:")
print(accuracy(a, Y))
```

```
a_hidden = sigmoid(zlist[0])
print(a_hidden)
```

```
accuracy:
1.0
[[0.94220321 0.0579803  0.93483679]
 [0.05737594 0.05822982 0.93434782]
 [0.02183669 0.50250713 0.06513916]
 [0.49807731 0.97820591 0.06505701]
 [0.05787973 0.94245745 0.93467481]
 [0.97810976 0.50337965 0.06529461]
 [0.94148443 0.94262799 0.93434802]
 [0.49772272 0.02214271 0.0646093 ]]
```

```
threshold = 0.5  # adjest for higher confidence

binary_representation = (a_hidden > threshold).astype(int)

print(binary_representation)
```

```
[[1 0 1]
 [0 0 1]
 [0 1 0]
 [0 1 0]
 [0 1 1]
 [1 1 0]
 [1 1 1]
 [0 0 0]]
```

It looks like the Encoder is converging towards binary representations of the input data, in order to map the input date into a smaller representation space.