EE496 Homework 1 - Training Multi-Layer Perceptron

Berken Utku Demirel - 2166221

March 31, 2020

1 Basic Concepts

1.1 Which Function ?

What function does the MLP classifier of Scikit-Learn [2] approximates ?

Multi-layer Perceptron (MLP) is a supervised learning algorithm that approximates the function $f(.) : R^m \rightarrow R^o$ by training on a given dataset, where m and o are the dimensions of input and output respectively.

How is the loss defined to approximate that function ?

Multi-layer perceptron(MLP) uses Cross-Entropy as a loss function for classification. In binary classification, cross-entropy can be calculated as in equation 1 where the $\hat{y}$ is the predicted probability observation and y is the binary indicator if class label is the correct classification for that observation.

$$Loss(\hat{y}, y, W) = -yln\hat{y} - (1-y)ln(1-\hat{y}) \tag{1}$$

If the classification is multiclass, which is in our case, the loss for each class is calculated and sum the result as in equation 2 where the M is the number of classes.

$$Loss = -\sum_{k=1}^{M} y_k ln\hat{y_k} \tag{2}$$

The reason why there is a defined loss to approximate that function is that the program learn by means of a loss function. It's a method of evaluating how well specific algorithm models the given dataset. If predictions deviates too much from actual results, loss function would result in a very large number. Gradually, with the help of some optimization function, loss function learns to reduce the error in prediction. Furthermore, an important aspect of cross-entropy loss is that it penalizes the predictions that are confident but wrong since when actual label is 1 ($y(i) = 1$), second half of function disappears whereas in case actual label is 0 ($y(i) = 0$) first half is dropped off.

1.2 Gradient Computation

The equation that describes the update rule for weights is given in equation 3,

$$w_{k+1} = w_k - \gamma \nabla_w \mathcal{L}|_{w=w_k} \tag{3}$$

If we solve this equation for the gradient of the loss, we would end up with the equation 4.

$$\nabla_w \mathcal{L}|_{w=w_k} = \frac{w_k - w_{k+1}}{\gamma} \tag{4}$$

1.3 Some Training Parameters and Basic Parameter Calculations

1.3.1

Batch is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.
Epoch is also a hyperparameter that defines the how many times the learning algorithm work through entire training dataset.

1.3.2

If the dataset has N samples and the batch size is B , the number of batches per epoch is N/B.

1.3.3

Stochastic gradient decent (SGD) approximate the gradient using only one data point. So, SGD iterates only once for a batch. The number of SGD iterations in an epoch is floor(N/B). Therefore, the total number of SGD iteration in the training is $E \times floor$(N/B).

1.4 Computing Number of Parameters of an MLP Classifier

To calculate the number of parameters in an MLP classifier, we have to find the total number of the weights and biases of the neural network. The calculation of the number of weights, which corresponds to the total number of connections between the layers, is given in equation 5.

$$\#\text{ of weights} = D_{in} \times H_1 + D_{out} \times H_k + \sum_{k=2}^{k-1} H_k \times H_{k+1} \tag{5}$$

Furthermore, the biases in each layer should be computed as in equation 6.

$$\#\text{ of biases} = \sum_{k=1}^{k} H_k \tag{6}$$

The sum of these two equations yields the number of parameters of the MLP, which is given in equation 7.

$$\#\text{ of parameters of MLP} = D_{in} \times H_1 + D_{out} \times H_k + \sum_{k=2}^{k-1} H_k \times H_{k+1} + \sum_{k=1}^{k} H_k \tag{7}$$

2 Experimenting MLP Architectures

2.1 Experimental Work

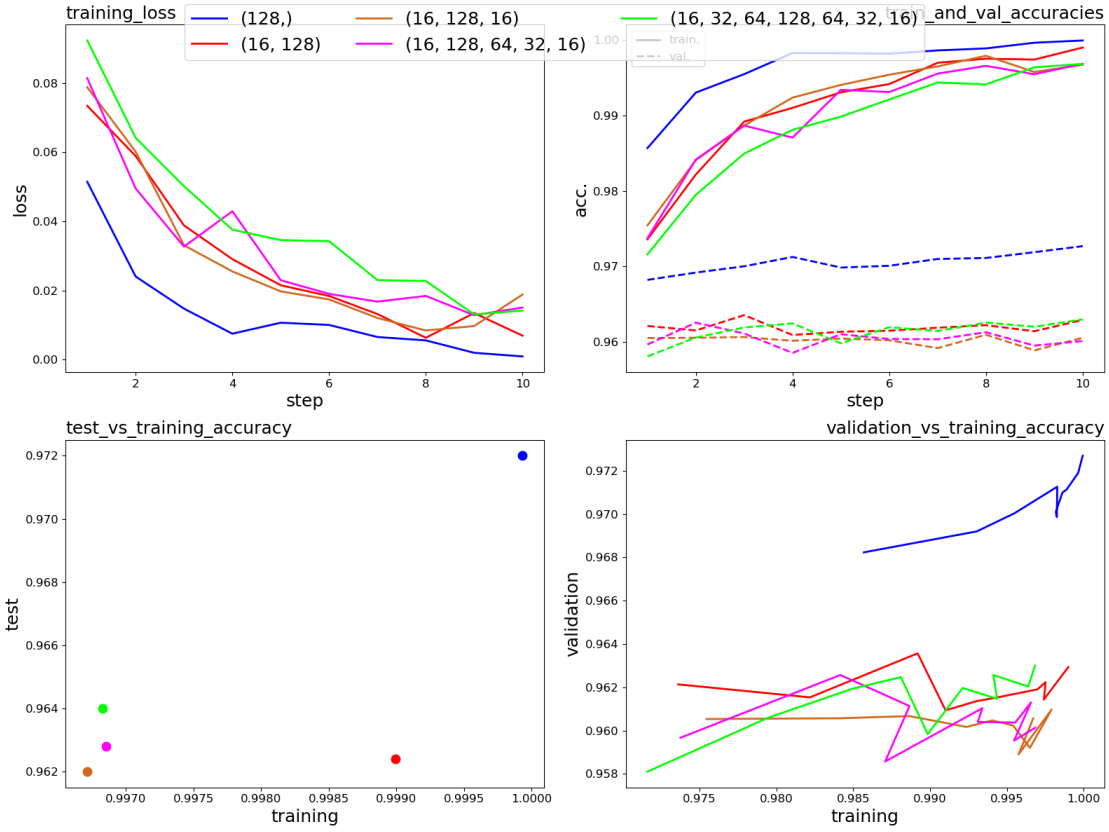

Figure 1: The result of *part2Plot* function with 100 epoch

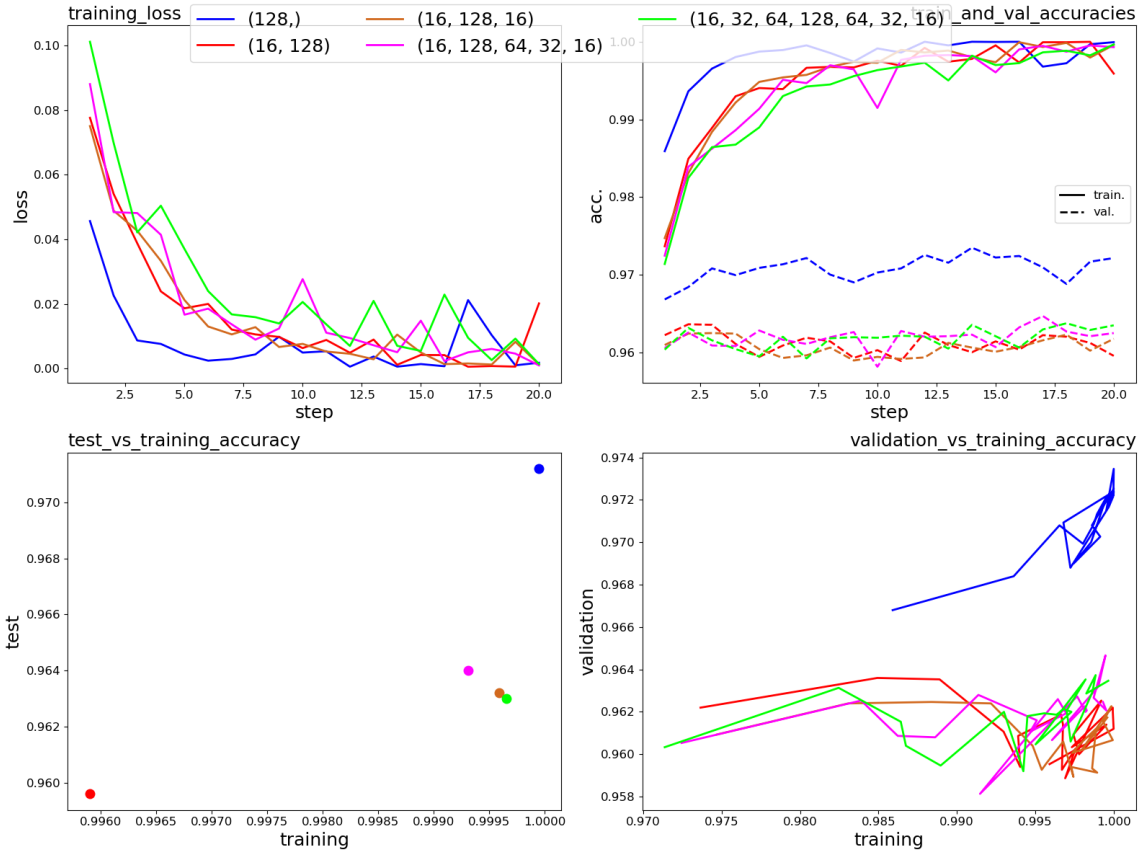

Figure 2: The result of the *part2Plot* function with 200 epoch

The figure 2 shows that the result of the part2 when the MLP has trained for 200 epochs. This plot is not wanted in the homework, however, in order to compare the generalization performance of the classifiers, it is obtained.

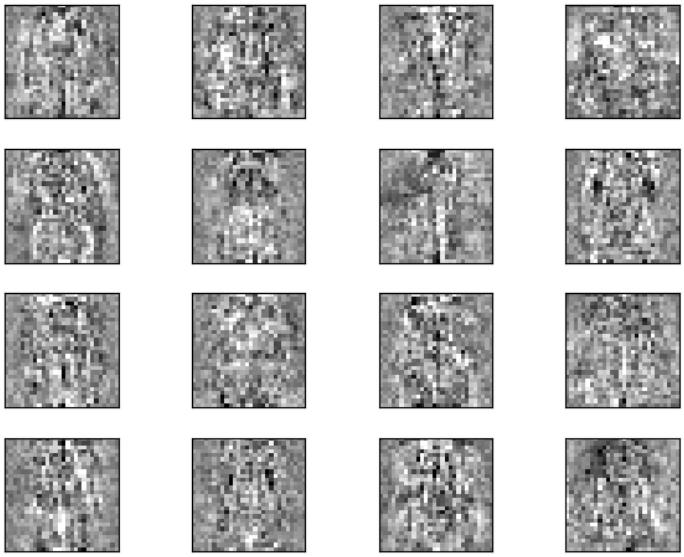

Figure 3: The result of the *visualizeWeights* function for the arch_2

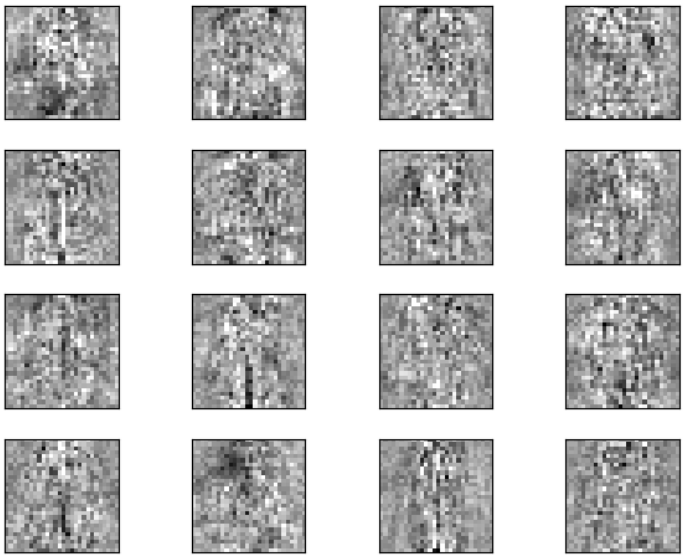

Figure 4: The result of the *visualizeWeights* function for the arch_3

Figure 5: The result of the *visualizeWeights* function for the arch_5

Figure 6: The result of the *visualizeWeights* function for the arch_7

2.2 Discussions

2.2.1 What is the generalization performance of a classifier?

Generalization performance of a classifier refers to your model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model.

2.2.2 Which plots are informative to inspect generalization performance?

We can make an inference about the generalization of the trained neural network by looking at the validation and test accuracies curve. Since the data, which is used for validation and test, are previously unseen data, drawn from the same distribution as the one used to create the model.

2.2.3 Compare the generalization performance of the architectures

When we look at the validation and test accuracies curve, we can observe that the 'arch_1' reaches the highest values in both. So the generalization performance of the 'arch_1' is superior to others. The other architectures are not so different in terms of the generalization. However, the 'arch_7', which is the deepest one, shows the lowest generalization performance amongst the others which can be seen from the figure 2. When the number of the epoch is increased to 200, the training accuracy of the 'arch_7' is better than 'arch_1' whereas the validation accuracy of the 'arch_7' is much worse than the 'arch_1'.

2.2.4 How does the number of parameters affect the classification and generalization performance?

When a neural network has more parameters, it has sufficient capacity to overfit the training data. Reducing the number of parameters in the network will increase the generalization capacity of the network. In general, with a fixed number of training patterns, overfitting can occur when the model has too many parameters (too many degrees of freedom).

2.2.5 How does the depth of the architecture affect the classification and generalization performance?

More layers offer more opportunity for hierarchical re-composition of abstract features learned from the data. However, the deep architectures need more data for training. Also, when we increase the size of the network, we'll introduce more parameters that network needs to learn, and hence increasing the chances of overfitting.

2.2.6 Considering the visualizations of the weights, are they interpretable?

The weights are similar to the inputs, which are fed to the network. However, when we look at the visualizations of the weights for the deeper networks especially 6, some of the figures are not detailed, which means that the network couldn't learn the features entirely for the classification.

2.2.7 Can you say whether the units are specialized to specific classes

2.2.8 Weights of which architecture are more interpretable?

When we look at the figures from 3 to 6, we can deduce that some of the images in the figure 6 are similar to dress and footwear input images.

2.2.9 Considering the architectures, comment on the structures (how they are designed). Can you say that some architecture are akin to each other? Compare the performance of similarly structured architectures and architectures with different structure.

The hidden layer size of all architectures is chosen as the power of 2. The most shallow one is composed of 1 hidden layer, whereas the deepest one is designed as consisting of 7 hidden layers. So we can separate these 5 different architectures as the shallow and deep neural network. When we compare the figures 1 and 2, we can easily observe that when the neural network is going to deeper, the new parameters are introduced to that network, and it is more prone to overfitting, thus the generalization performance of these neural networks is worse than the shallow one.

2.2.10 Which architecture would you pick for this classification task? Why?

I would choose the 'arch_1' because it can be observerd from figures 1 and 2, the generalization performance of the classifier is the best amongst the others.

3 Experimenting Activation Functions

3.1 Experimental Work

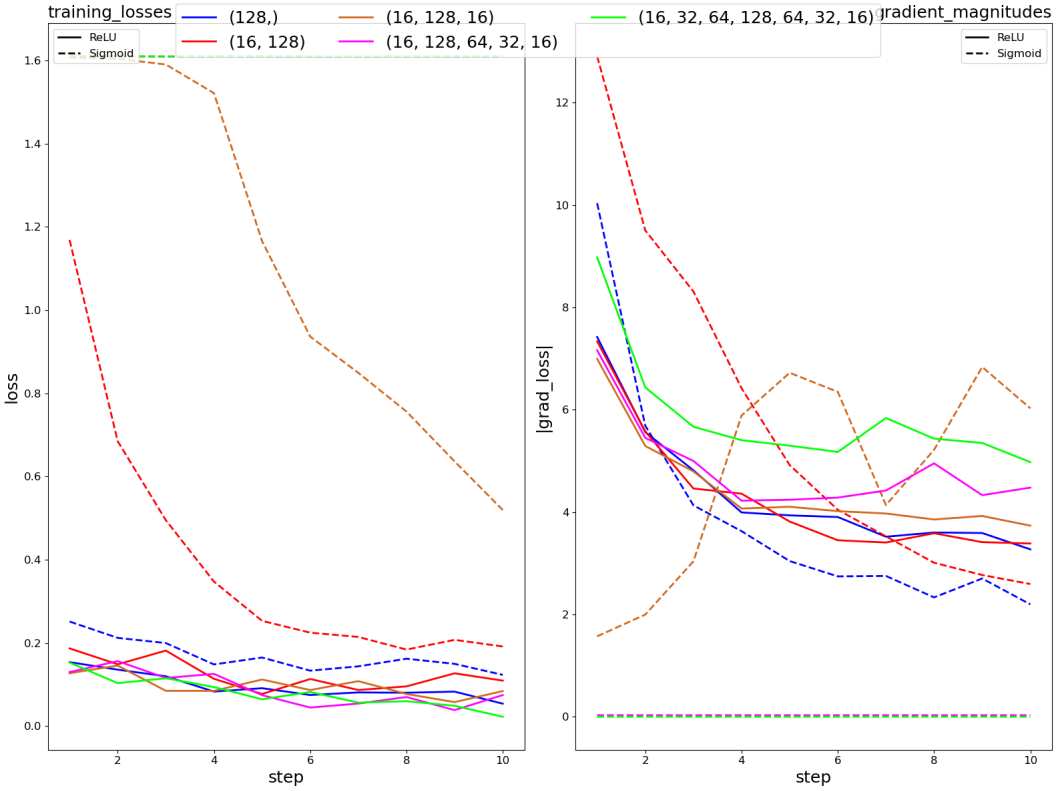

Figure 7: The result of *part3Plot* function

3.2 Discussions

3.2.1 How is the gradient behavior in different architectures? What happens when depth increases?

When we look at the figure 7, we can observe that when the network architecture is deeper, the magnitude of the loss gradient is decreasing especially for the architectures which use sigmoid function. As more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train.

3.2.2 Why do you think that happens?

Certain activation functions, like the sigmoid function, squishes a large input space into a small input space between 0 and 1. Therefore, a significant change in the input of the sigmoid function will cause a minimal change in the output. Hence, the derivative becomes small. This is not a big problem for shallow networks like 'arch_1'. However, when more layers are used, it can cause the gradient to be too small for training because n hidden layers use the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers.

3.2.3 What might happen if we do not scale the inputs to the range [-1.0,1.0]?

If we don't scale the data, then the convergence will be slower. The training time will be more compared to training using normalized data. Also, the gradient descent will take a longer time to converge compared to train by using scaled data. In addition, standardizing the inputs can reduce the chances of getting stuck in local optima.

4 Experimenting Learning Rate

4.1 Experimental Work

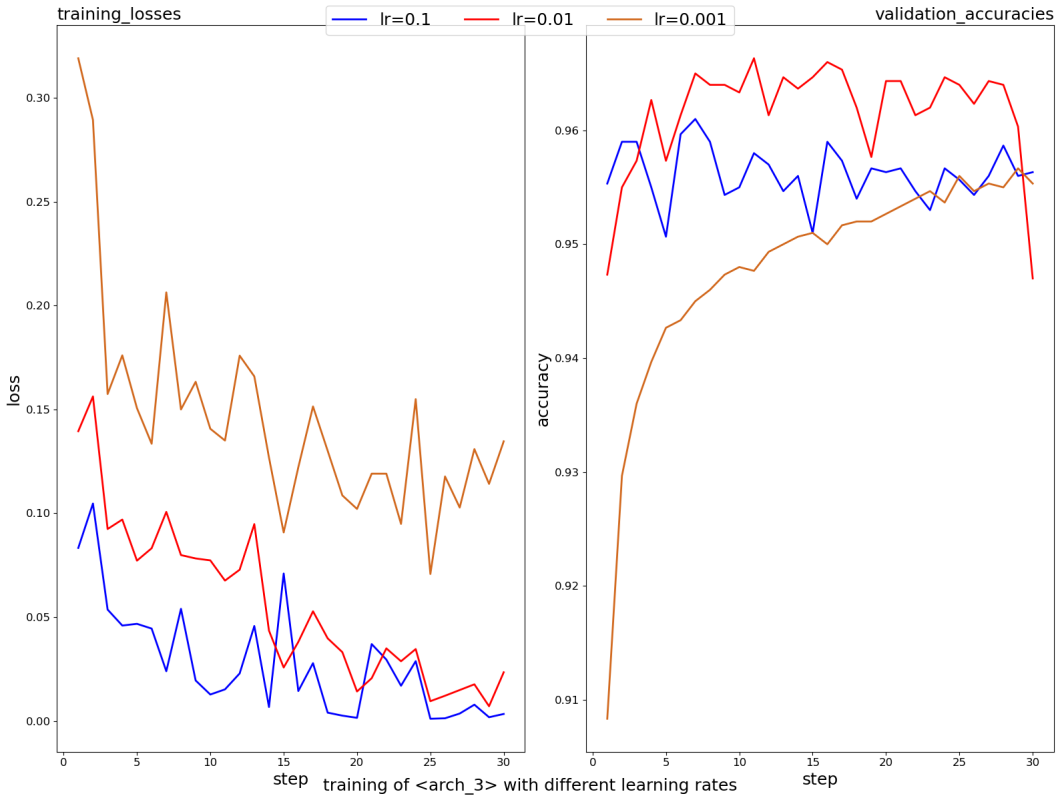

Figure 8: The result of *part4Plot* function for 'arch_3'

Figure 8 shows the result of the *part4Plot* function for the three different MLP classifier with learning rates 0.1, 0.01 and 0.001, respectively.

Figure 9 represents the validation accuracy when we set the learning rate to 0.01 and continue training until 200 epochs.

Figure 10 represents the validation accuracy when the learning rate is changed to twice first 0.01 then 0.001 and continue training until 200 epochs.

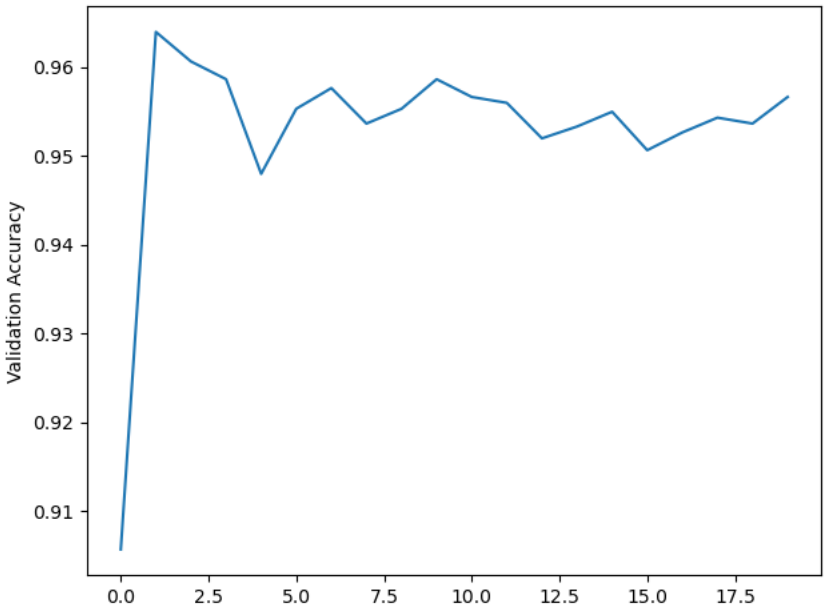

Figure 9: The validation curve for the part 4.3 with $\gamma = 0.01$

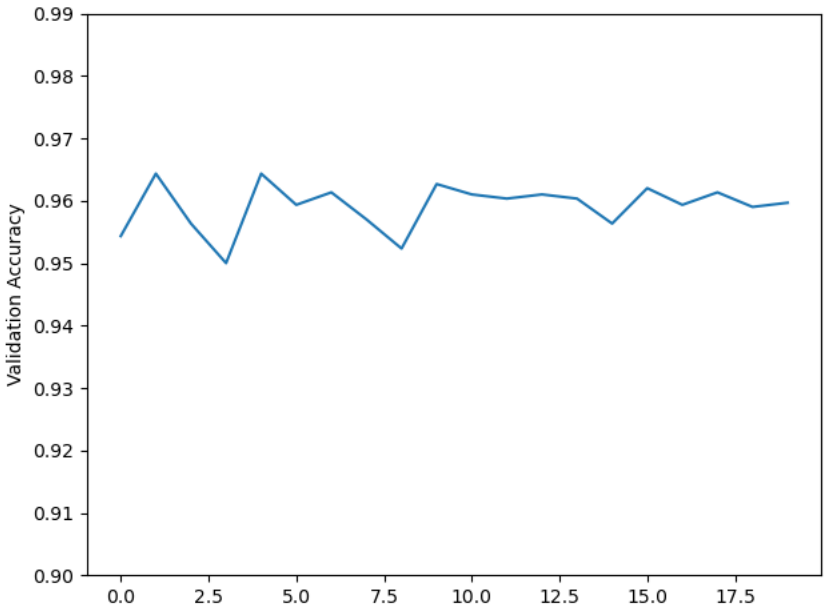

Figure 10: The validation curve for the part 4.6 with $\gamma = 0.001$

4.2 Discussions

4.2.1 How does the learning rate affect the convergence speed?

The learning rate controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is important since if it is too small, the learning results in a long training process that could get stuck. In contrast, a large learning rate might result in learning a sub-optimal set of weights too fast or an unstable training process. The figure 8 shows that the architecture, which has a bigger learning rate, converges to its maximum validation accuracy in a smaller step.

4.2.2 How does the learning rate affect the convergence to a better point?

When the learning rate is too large, the learning may end in a not-optimal point since as the learning rate increases, the network becomes more unstable. This fact can also be seen in figure 8. The validation accuracy of the architecture with $\gamma = 0.1$ is behaving unstable(lots of peaks), whereas the architecture, which has a learning rate of 0.001, is more stable.

4.2.3 Does your scheduled learning rate method work? In what sense?

Yes, It works. When we look at the figures 9 and 10, we can easily see that the instability of the model is getting smaller in the figure 10 that the scheduled learning rate is applied. Also, the validation accuracy(0.96) is better than the model in which there is no scheduled learning rate.

4.2.4 Compare the accuracy and convergence performance of your scheduled learning rate method with ADAM.

When the ADAM method was used, the best accuracy is 0.9723, which can be seen from figure 1. The accuracy of the scheduled learning rate method is 0.9933. So, we can deduce that the scheduled learning rate method is better than ADAM in terms of accuracy. For the comparison of the convergence performance, both architectures work well since, at the end of the steps, they are stable. However, when we have used the scheduled learning rate the number of epoch which is needed for the convergence is bigger than the ADAM. Therefore, we can make an inference that the convergence performance of the ADAM is better than the scheduled learning rate method.

5 Appendix

File - C:\Users\berkenutku\Desktop\496_hw\MLP_part2.py

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
import numpy as np

# Load the train and test images with labels
train_images = np.load('train_images.npy')
train_labels = np.load('train_labels.npy')
test_images = np.load('test_images.npy')
test_labels = np.load('test_labels.npy')

train_images_reshaped = np.reshape(train_images, (30000, 28, 28))
test_images_reshaped = np.reshape(test_images, (test_images.shape[0], 28, 28))

scaled_images = np.empty(shape=(30000, 28, 28))
scaled_images_test = np.empty(shape=(5000, 28, 28))

# In order to scale pixel values to [-1, 1], The MinMaxScaler was used with a range [-1,1]
counter = 0
scaler = MinMaxScaler(feature_range=(-1, 1))

for x in train_images_reshaped:
    scaled_images[counter, :, :] = scaler.fit_transform(x)
    counter += 1

counter = 0
for x in test_images_reshaped:
    scaled_images_test[counter, :, :] = scaler.fit_transform(x)
    counter += 1

# In order to split 10 percent of the training data set to validation
X_train, X_validate, y_train, y_validate = train_test_split(scaled_images,
                                                            train_labels, test_size=0.1,
                                                            random_state=40, stratify=train_labels)

# Definition of all architectures
arch_1 = (128,)
arch_2 = (16, 128,)
arch_3 = (16, 128, 16,)
arch_5 = (16, 128, 64, 32, 16,)
arch_7 = (16, 32, 64, 128, 64, 32, 16,)

arch = [arch_1, arch_2, arch_3, arch_5, arch_7]

list_of_dict = []

# For loop for all architectures
for m in arch:
    average_loss = np.zeros(10)
    average_valid_accuracy = np.zeros(10)
    average_training_accuracy = np.zeros(10)
    overall_score = np.zeros(10)
    weights_first_layer = []
    arc_dict = {}
    counter_for_list = 0
    # Repeat the all training for 10 times, then take the average of these
    for x in range(0, 10):
        mlp = MLPClassifier(hidden_layer_sizes=m, activation='relu',
                            solver='adam', max_iter=1, shuffle=True)

        # Flatten input data
        nsamples, nx, ny = X_train.shape
        d2_train_dataset = X_train.reshape((nsamples, nx*ny))

        nsamples, nx, ny = X_validate.shape
        d2_validate_dataset = X_validate.reshape((nsamples, nx*ny))

        nsamples, nx, ny = scaled_images_test.shape
        d2_test_dataset = scaled_images_test.reshape((nsamples, nx*ny))

        valid_accuracy = np.empty(shape=(10,))
        training_accuracy = np.empty(shape=(10,))
        Loss = np.empty(shape=(10,))

        # For loop to 100 epochs
        counter_10_epoch = 0
```

File - C:\Users\berkenutku\Desktop\496_hw\MLP_part2.py

```
78          for i in range(1, 101):
79              # Shuffle training set after each epoch
80              random_perm = np.random.permutation(X_train.shape[0])
81              # Set the mini_batch indexes to 0 after each epoch
82              mini_batch_index = 0
83
84              # Until all data is fed to network, continue to fed batches.
85              while True:
86                  indices = random_perm[mini_batch_index:mini_batch_index + 500]
87                  mlp.partial_fit(d2_train_dataset[indices], y_train[indices], np.unique(y_train))
88                  mini_batch_index += 500
89
90                  # When the batches are bigger than dataset, break the epoch
91                  if mini_batch_index >= X_train.shape[0]:
92                      break
93
94              # Record the training loss, training accuracy, validation accuracy for every 10 steps
95              if i % 10 == 0:
96                  valid_accuracy[counter_10_epoch] = mlp.score(d2_validate_dataset, y_validate)
97                  training_accuracy[counter_10_epoch] = mlp.score(d2_train_dataset, y_train)
98                  Loss[counter_10_epoch] = mlp.loss_
99                  counter_10_epoch += 1
100
101         average_loss = (average_loss + Loss)
102         average_valid_accuracy = (average_valid_accuracy + valid_accuracy)
103         average_training_accuracy = (average_training_accuracy + training_accuracy)
104         overall_score[x] = mlp.score(d2_test_dataset, test_labels)
105         weights_first_layer.append(mlp.coefs_[0])
106
107     # At the final divide the each parameter to 10, in order to get average values.
108     average_loss = average_loss / 10
109     average_valid_accuracy = average_valid_accuracy / 10
110     average_training_accuracy = average_training_accuracy / 10
111
112     # Get the index and value of the best test accuracy
113     best_test_accuracy_index = np.argmax(overall_score)
114     best_accuracy = overall_score[best_test_accuracy_index]
115     best_weights = weights_first_layer[best_test_accuracy_index]
116
117     arc_dict['name'] = m
118     arc_dict['loss_curve'] = average_loss
119     arc_dict['train_acc_curve'] = average_training_accuracy
120     arc_dict['val_acc_curve'] = average_valid_accuracy
121     arc_dict['test_acc'] = best_accuracy
122     arc_dict['weights'] = best_weights
123     list_of_dict.append(arc_dict)
124
125 # Visualization Part
126 from utils import part2Plots, visualizeWeights
127 part2Plots(list_of_dict, save_dir='', filename='', show_plot=True)
128
129 a = list_of_dict[1]['weights']
130 visualizeWeights(a, save_dir='', filename='weigths')
131
132 a = list_of_dict[2]['weights']
133 visualizeWeights(a, save_dir='', filename='weigths')
134
135 a = list_of_dict[3]['weights']
136 visualizeWeights(a, save_dir='', filename='weigths')
137
138 a = list_of_dict[4]['weights']
139 visualizeWeights(a, save_dir='', filename='weigths')
140
141
```

File - C:\Users\berkenutku\Desktop\496_hw\MLP_part3.py

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
import numpy as np
import copy

train_images = np.load('train_images.npy')
train_labels = np.load('train_labels.npy')
test_images = np.load('test_images.npy')
test_labels = np.load('test_labels.npy')

train_images_reshaped = np.reshape(train_images, (30000, 28, 28))
test_images_reshaped = np.reshape(test_images, (test_images.shape[0], 28, 28))

scaled_images = np.empty(shape = (30000, 28, 28))
scaled_images_test = np.empty(shape=(5000, 28, 28))
# In order to scale pixel values to [-1, 1], The MinMaxScaler was used.
counter = 0
scaler = MinMaxScaler(feature_range=(-1, 1))

for x in train_images_reshaped:
    scaled_images[counter, :, :] = scaler.fit_transform(x)
    counter += 1

counter = 0
for x in test_images_reshaped:
    scaled_images_test[counter, :, :] = scaler.fit_transform(x)
    counter += 1

# In order to split 10 percent of the training data set to validation
X_train, X_validate, y_train, y_validate = train_test_split(scaled_images,
                                                            train_labels, test_size=0.1,
                                                            random_state=40, stratify=train_labels)

arch_1 = (128,)
arch_2 = (16, 128,)
arch_3 = (16, 128, 16,)
arch_5 = (16, 128, 64, 32, 16,)
arch_7 = (16, 32, 64, 128, 64, 32, 16,)

arch = [arch_1, arch_2, arch_3, arch_5, arch_7]

list_of_dict = []
for m in arch:
    relu_loss = np.zeros(10)
    sigmoid_loss = np.zeros(10)
    relu_grad = np.zeros(10)
    sigmoid_grad = np.zeros(10)
    arc_dict = {}
    counter_for_list = 0

    mlp_relu = MLPClassifier(hidden_layer_sizes=m, activation='relu',
                             solver='sgd', max_iter=1, shuffle=True, learning_rate_init=0.01, momentum=
0.0)

    mlp_sigmoid = MLPClassifier(hidden_layer_sizes=m, activation='logistic',
                             solver='sgd', max_iter=1, shuffle=True, learning_rate_init=0.01, momentum=
0.0)

    # Flatten input data
    nsamples, nx, ny = X_train.shape
    d2_train_dataset = X_train.reshape((nsamples, nx*ny))

    nsamples, nx, ny = X_validate.shape
    d2_validate_dataset = X_validate.reshape((nsamples, nx*ny))

    nsamples, nx, ny = scaled_images_test.shape
    d2_test_dataset = scaled_images_test.reshape((nsamples, nx*ny))

    valid_accuracy = np.empty(shape=(10,))
    training_accuracy = np.empty(shape=(10,))
    Loss = np.empty(shape=(10,))

    # For loop to 100 epochs
    counter_10_epoch = 0
    for i in range(1, 101):
```

File - C:\Users\berkenutku\Desktop\496_hw\MLP_part3.py

```
        # Shuffle training set after each epoch
        random_perm = np.random.permutation(X_train.shape[0])
        mini_batch_index = 0

        # After the first epoch, save the previous weights in order to calculate magnitude of the gradient loss
        if i != 1:
            weights_relu_before = copy.deepcopy(mlp_relu.coefs_)
            weights_relu_before = weights_relu_before[0]
            weights_sigmoid_before = copy.deepcopy(mlp_sigmoid.coefs_)
            weights_sigmoid_before = weights_sigmoid_before[0]

        # Until all data is fed to network, continue to fed batches.
        while True:
            indices = random_perm[mini_batch_index:mini_batch_index + 500]
            mlp_relu.partial_fit(d2_train_dataset[indices], y_train[indices], np.unique(y_train))
            mlp_sigmoid.partial_fit(d2_train_dataset[indices], y_train[indices], np.unique(y_train))
            mini_batch_index += 500

            if mini_batch_index >= X_train.shape[0]:
                break

        # Record the parameters for every 10 steps
        if i % 10 == 0:
            weights_relu_after = mlp_relu.coefs_[0]
            weights_sigmoid_after = mlp_sigmoid.coefs_[0]
            relu_grad[counter_10_epoch] = np.linalg.norm((weights_relu_before - weights_relu_after) * 100)
            sigmoid_grad[counter_10_epoch] = np.linalg.norm((weights_sigmoid_before - weights_sigmoid_after) * 100)
            relu_loss[counter_10_epoch] = mlp_relu.loss_
            sigmoid_loss[counter_10_epoch] = mlp_sigmoid.loss_
            counter_10_epoch += 1


    arc_dict['name'] = m
    arc_dict['relu_loss_curve'] = relu_loss
    arc_dict['sigmoid_loss_curve'] = sigmoid_loss
    arc_dict['relu_grad_curve'] = relu_grad
    arc_dict['sigmoid_grad_curve'] = sigmoid_grad
    list_of_dict.append(arc_dict)

# Visualization Part
from utils import part3Plots
part3Plots(list_of_dict, save_dir='', filename='', show_plot=True)


```

File - C:\Users\berkenutku\Desktop\496_hw\MLP_part4.py

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
import numpy as np


train_images = np.load('train_images.npy')
train_labels = np.load('train_labels.npy')
test_images = np.load('test_images.npy')
test_labels = np.load('test_labels.npy')

train_images_reshaped = np.reshape(train_images, (30000, 28, 28))
test_images_reshaped = np.reshape(test_images, (test_images.shape[0], 28, 28))

scaled_images = np.empty(shape=(30000, 28, 28))
scaled_images_test = np.empty(shape=(5000, 28, 28))
# In order to scale pixel values to [-1, 1], The MaxAbsScaler was used.
counter = 0
scaler = MinMaxScaler(feature_range=(-1, 1))

for x in train_images_reshaped:
    scaled_images[counter, :, :] = scaler.fit_transform(x)
    counter += 1

counter = 0
for x in test_images_reshaped:
    scaled_images_test[counter, :, :] = scaler.fit_transform(x)
    counter += 1

# In order to split 10 percent of the training data set to validation
X_train, X_validate, y_train, y_validate = train_test_split(scaled_images,
                                                            train_labels, test_size=0.1,
                                                            random_state=40, stratify=train_labels)
# parameter for the number of epoch
number_of_epoch = 20

# Chosen favourite architecture
arch_fav = (16, 128, 16)

average_loss_1 = np.zeros(number_of_epoch)
average_loss_01 = np.zeros(number_of_epoch)
average_loss_001 = np.zeros(number_of_epoch)

average_valid_scheduled = np.zeros(number_of_epoch)

average_valid_accuracy_1 = np.zeros(number_of_epoch)
average_valid_accuracy_01 = np.zeros(number_of_epoch)
average_valid_accuracy_001 = np.zeros(number_of_epoch)

counter_for_list = 0
arc_dict = {}

# There is no averaging in this part, so just one loop will be enough
for x in range(0, 1):
    mlp_1 = MLPClassifier(hidden_layer_sizes=arch_fav, activation='relu',
                          solver='sgd', max_iter=1, shuffle=True, learning_rate_init=0.1, momentum=0.0)

    mlp_01 = MLPClassifier(hidden_layer_sizes=arch_fav, activation='relu',
                          solver='sgd', max_iter=1, shuffle=True, learning_rate_init=0.01, momentum=0.0)

    mlp_001 = MLPClassifier(hidden_layer_sizes=arch_fav, activation='relu',
                          solver='sgd', max_iter=1, shuffle=True, learning_rate_init=0.001, momentum=0.0
)

    mlp_scheduled = MLPClassifier(hidden_layer_sizes=arch_fav, activation='relu',
                      solver='sgd', max_iter=1, shuffle=True, learning_rate_init=0.1, momentum=0.0)

    # Flatten input data
    nsamples, nx, ny = X_train.shape
    d2_train_dataset = X_train.reshape((nsamples, nx*ny))

    nsamples, nx, ny = X_validate.shape
    d2_validate_dataset = X_validate.reshape((nsamples, nx*ny))

    nsamples, nx, ny = scaled_images_test.shape
    d2_test_dataset = scaled_images_test.reshape((nsamples, nx*ny))
```

File - C:\Users\berkenutku\Desktop\496_hw\MLP_part4.py

```
77
78      valid_accuracy_1 = np.empty(shape=(number_of_epoch,))
79      valid_accuracy_01 = np.empty(shape=(number_of_epoch,))
80      valid_accuracy_001 = np.empty(shape=(number_of_epoch,))
81
82      valid_accuracy_scheduled = np.empty(shape=(number_of_epoch))
83
84      Loss_1 = np.empty(shape=(number_of_epoch,))
85      Loss_01 = np.empty(shape=(number_of_epoch,))
86      Loss_001 = np.empty(shape=(number_of_epoch,))
87
88      # For loop to 200 epochs
89      counter_10_epoch = 0
90      for i in range(1, 201):
91          # Shuffle training set after each epoch
92          random_perm = np.random.permutation(X_train.shape[0])
93          mini_batch_index = 0
94
95          # First change of learning rate for scheduled learning method
96          if i == 70:
97              mlp_scheduled.set_params(learning_rate_init=0.01)
98
99          # Second change of learning rate for scheduled learning method
100         if i == 120:
101             mlp_scheduled.set_params(learning_rate_init=0.001)
102
103         # Until all data is fed to network, continue to fed batches.
104         while True:
105             indices = random_perm[mini_batch_index:mini_batch_index + 500]
106             mlp_1.partial_fit(d2_train_dataset[indices], y_train[indices], np.unique(y_train))
107             mlp_01.partial_fit(d2_train_dataset[indices], y_train[indices], np.unique(y_train))
108             mlp_001.partial_fit(d2_train_dataset[indices], y_train[indices], np.unique(y_train))
109             mlp_scheduled.partial_fit(d2_train_dataset[indices], y_train[indices], np.unique(y_train)
    )
110             mini_batch_index += 500
111
112             if mini_batch_index >= X_train.shape[0]:
113                 break
114
115         # Record the training loss, training accuracy, validation accuracy for every 10 steps
116         if i % 10 == 0:
117             valid_accuracy_1[counter_10_epoch] = mlp_1.score(d2_validate_dataset, y_validate)
118             valid_accuracy_01[counter_10_epoch] = mlp_01.score(d2_validate_dataset, y_validate)
119             valid_accuracy_001[counter_10_epoch] = mlp_001.score(d2_validate_dataset, y_validate)
120             valid_accuracy_scheduled[counter_10_epoch] = mlp_scheduled.score(d2_validate_dataset,
    y_validate)
121
122             Loss_1[counter_10_epoch] = mlp_1.loss_
123             Loss_01[counter_10_epoch] = mlp_01.loss_
124             Loss_001[counter_10_epoch] = mlp_001.loss_
125             counter_10_epoch += 1
126
127     # No division is required for this part, since averaging of the parameters is not desired.
128     average_loss_1 = (average_loss_1 + Loss_1)
129     average_loss_01 = (average_loss_01 + Loss_01)
130     average_loss_001 = (average_loss_001 + Loss_001)
131
132     average_valid_accuracy_1 = (average_valid_accuracy_1 + valid_accuracy_1)
133     average_valid_accuracy_01 = (average_valid_accuracy_01 + valid_accuracy_01)
134     average_valid_accuracy_001 = (average_valid_accuracy_001 + valid_accuracy_001)
135     average_valid_scheduled = (average_valid_scheduled + valid_accuracy_scheduled)
136
137
138
139 arc_dict['name'] = 'arch_3'
140 arc_dict['loss_curve_1'] = average_loss_1
141 arc_dict['loss_curve_01'] = average_loss_01
142 arc_dict['loss_curve_001'] = average_loss_001
143 arc_dict['val_acc_curve_1'] = average_valid_accuracy_1
144 arc_dict['val_acc_curve_01'] = average_valid_accuracy_01
145 arc_dict['val_acc_curve_001'] = average_valid_accuracy_001
146
147 # Visualization Part
148 from utils import part4Plots
149 part4Plots(arc_dict, save_dir='', filename='', show_plot=True)
150
151 plt.plot(average_valid_scheduled)
```

```
152 plt.ylabel('Validation Accuracy')
153 plt.show()
154
155 print(mlp_scheduled.score(d2_train_dataset, y_train))
156
```