

Assignment 2 - 2IIG0

Teodor-Cristian Lungu (1416332)
t.lungu@student.tue.nl

Radu Lucian Radulescu (1438808)
@student.tue.nl

Vlad Pintilie (1447637)
v.pintilie@student.tue.nl

December 23, 2021

Exercise 6

All exercises have been done using Python, inside the provided notebook, which has been attached in its entirety throughout the document. Firstly, the data has been imported and visualized through a few common methods.

(a) We have imported the data in Python using:

```
1 #For Windows use 'r' in front of the string
2 path = r"D:\2IIG0\2IIG0---Data-Mining-and-Machine-Learning\HW2\data"
3 train_data = pd.read_csv(path + r"\train_data.csv")
4 validate_data = pd.read_csv(path + r"\validate_data.csv")
```

After importing the data we analyzed the resulting table and used summary statistics to get a grasp for how the data is structured.

```
1 train_data.describe()
2 validate_data.describe()
```

(b) For the data normalization procedure, we chose the Min-Max normalization method. Since it guarantees all features will have the exact same scale, we chose this method. The function that we used for the normalization is:

```
1 def normalize_data_minmax(data):
2     min_and_max = [[min(feature), max(feature)] for feature in zip(*data)]
3     for obs in data:
4         for i in range(len(obs)-1):
5             obs[i] = (obs[i] - min_and_max[i][0]) / (min_and_max[i][1] -
                min_and_max[i][0])
```

Furthermore, we use two auxiliary functions to convert strings to float and strings to integers respectively. An additional auxiliary function is used for loading the csv.

```
1 #For the feature columns: convert string to float
2 def convert_float(data, feature):
3     for line in data:
4         line[feature] = float(line[feature].strip())
5
6 #For the classes column: convert string to int
7 def convert_int(data, feature):
8     for line in data:
9         line[feature] = int(line[feature].strip())
```

```
1 def load_csv_file(path):
2     data = list()
3     with open(path, 'r') as file:
4         file_reader = reader(file)
5
6         for i in file_reader:
7             if not i:
8                 continue
9
10            data.append(i)
11
```

```
12     return data
```

The full data preparation process is done by loading the dataset and calling the above defined functions:

```
1 def prepare_data(path):
2     datafile = path
3     data = load_csv_file(datafile)
4     data = data[1:]
5     for i in range(len(data[0])-1):
6         convert_float(data, i)
7     convert_int(data, len(data[0])-1)
8     normalize_data_minmax(data)
9     return data
```

The first 4 samples of the training data after min-max data normalization are:

```
1 [[0.29148411546616415, 0.3282996259301394, 0],
2  [0.18776046734102808, 0.07576615122002546, 0],
3  [0.41972327129796977, 0.15359536711078225, 0],
4  [0.8188758159059557, 0.9842582787339653, 0]]
```

(c) Since there are 2 features, we will have 2 inputs, hence 2x10 connections with the first hidden layer. Because there are 2 classes, then we'll have two neurons in the last layer.

(d) The activation function of choice is ReLU as it is the most common for such problems. We have also tested with sigmoid and we managed to get 97% accuracy without batches.

(e) The initialization of the neuralnet is done as follows:

```
1 def init_neuralnet(nrof_inputs, nrof_neurons_hidden, nrof_outputs):
2     neuralnet = list()
3     hidden_layer_first = [{'weights':[random() for i in range(nrof_inputs + 1)]} for i
4                             in range(nrof_neurons_hidden)]
5     neuralnet.append(hidden_layer_first)
6
7     hidden_layer_second = [{'weights':[random() for i in range(nrof_neurons_hidden +
8                             1)]} for i in range(nrof_neurons_hidden)]
9     neuralnet.append(hidden_layer_second)
10
11     output_layer = [{'weights':[random() for i in range(nrof_neurons_hidden + 1)]} for
12                      i in range(nrof_outputs)]
13     neuralnet.append(output_layer)
14
15     return neuralnet
```

As can be seen, the weights are assigned randomly, because this is an expectation of the SGD algorithm used to train the model.

(f) The MLP network has been implemented from scratch with the specification stated above and the backpropagation algorithm to train the network. Apart from the functions shown above, we have defined and used the following functions.

The `evaluate_accuracy()` function defines a neural network algorithm, retrieves the outputs and compares them to the groundtruth. The accuracy is computed by an auxiliary function that counts the number of correct outputs and divides them by the total number of observations.

```

1 # Evaluate and compute accuracy of neural network
2 def evaluate_accuracy(trainset, validationset, algorithm, *args):
3     outputs = algorithm(trainset, validationset, *args)
4     groundtruth = [line[-1] for line in validationset]
5     accuracy = compute_accuracy(groundtruth, outputs[0])
6     return accuracy, outputs
7
8 def compute_accuracy(groundtruth, outputs):
9     counter = 0
10    for line in range(len(groundtruth)):
11        if groundtruth[line] == outputs[line]:
12            counter += 1
13    return counter / float(len(groundtruth)) * 100.0

```

The `algorithm()` function defines the number of inputs and outputs for a neural network, uses those to initialize a neural network with those parameters and then trains the created neural network. The training errors/losses are stored. Then, the network is run on the validation set and the 'guesses' or the 'forecasts' of the network on these observations are stored. The auxiliary function `classify_observation` passes an observation through the network and retrieves the best guess. Both of the aforementioned information fields are then returned.

```

1 #Declare, define, run and collect outputs of a neural network
2 def algorithm(trainset, validationset, lr, epochs, nrof_neurons_hidden, batches):
3     #Number of I/O
4     inputs = len(trainset[0]) - 1
5     outputs = len(set([line[-1] for line in trainset]))
6
7     #Retrieve neural network
8     network = init_neuralnet(inputs, nrof_neurons_hidden, outputs)
9     errors = train_network(network, trainset, lr, epochs, outputs, batches)
10
11    #Run on validation set
12    validation_results = list()
13    for line in validationset:
14        guess = classify_observation(network, line)
15        validation_results.append(guess)
16
17    return validation_results, errors
18
19 # Predicts the class of an observation
20 # Pass forward and pick the best prediction
21 def classify_observation(network, row):
22     outputs = forward_pass(network, row)
23     return outputs.index(max(outputs))

```

Our first attempt was a training without any batch split of the data. We train for a fixed number of epochs, for each observation we forward it through the network, calculate the loss/error, and then we do a backward pass. Finally, we update the weights.

```

1 # Train without batches, only SGD
2 def train_network_SGD(network, training_dataset, learning_rate, nrof_epochs,
    nrof_outputs, dummy):

```

```

3     errors = list()
4
5     for epoch in range(nrof_epochs):
6         error = 0
7
8         for observation in training_dataset:
9             outputs = forward_pass(network, observation)
10            groundtruths = [0 for i in range(nrof_outputs)]
11            groundtruths[observation[-1]] = 1
12            error += sum([(groundtruths[i] - outputs[i]) ** 2 for i in
13                           range(len(groundtruths))])
14
15            backward_pass(network, groundtruths)
16            update_weights(network, observation, learning_rate)
17
18            print('>epoch=%d, error=%.3f' % (epoch, error))
19            errors.append(error)
20
21     return errors

```

Our next attempt was to train by splitting the data into batches. Now, instead of updating the weights directly after a backward pass, we do backward passes for all the observations in a batch and then we update the entire batch.

```

1 # Train with batches
2 def train_network(neuralnet, train, learning_rate, nrof_epochs, nrof_outputs,
3                  batch_size):
4     errors = list()
5     for epoch in range(nrof_epochs):
6         batch_error = 0
7         for batch in make_batches(train, batch_size=batch_size):
8             error = 0
9             for line in batch:
10                outputs = forward_pass(neuralnet, line)
11                groundtruths = [0 for i in range(nrof_outputs)]
12                groundtruths[line[-1]] = 1
13                error += sum([(groundtruths[i] - outputs[i]) ** 2 for i in
14                               range(len(groundtruths))])
15                backward_pass(neuralnet, groundtruths)
16
17                for row in batch:
18                    update_weights(neuralnet, row, learning_rate)
19                batch_error += error
20            errors.append(batch_error/batch_size)
21            print(epoch)
22
23     return errors
24
25 def make_batches(iterable, batch_size=1):
26     n = len(iterable)
27     for i in range(0, n, batch_size):
28         yield iterable[i:min(i + batch_size, n)]

```

For a forward pass we just apply the activation like in the lectures. We do the sum of the products between weights and inputs and then we put it through the activation function, in this case ReLU.

For the backward pass, we start from the output layer and we calculate the difference be-

tween the groundtruth and the guess. We then calculate with what amount we have to change the weights, i.e. the B times the derivative of the activation. Then, for the hidden layers we adjust the loss/errors by the calculated weight change.

We update the weights as in the slides by the product of the learning rate, the weight change and the inputs.

```

1 # Forward pass, backward pass, update weights
2 def forward_pass(neuralnet, row):
3     inputs = row
4     for layer in neuralnet:
5         new_inputs = []
6         for neuron in layer:
7             neuron['output'] = activation(sum_weight_input(neuron['weights'], inputs))
8             new_inputs.append(neuron['output'])
9         inputs = new_inputs
10    return inputs
11
12 def backward_pass(neuralnet, expected):
13     for i in reversed(range(len(neuralnet))):
14         layer = neuralnet[i]
15         errors = list()
16         if i != len(neuralnet)-1:
17             for j in range(len(layer)):
18                 error = 0.0
19                 for neuron in neuralnet[i + 1]:
20                     error += (neuron['weights'][j] * neuron['weight_change'])
21                 errors.append(error)
22         else:
23             for j in range(len(layer)):
24                 neuron = layer[j]
25                 errors.append(neuron['output'] - expected[j])
26         for j in range(len(layer)):
27             neuron = layer[j]
28             neuron['weight_change'] = errors[j] * activation_derivative(neuron['output'])
29
30 def update_weights(neuralnet, line, l_rate):
31     for i in range(len(neuralnet)):
32         inputs = line[:-1]
33
34         if i != 0:
35             inputs = [neuron['output'] for neuron in neuralnet[i - 1]]
36
37         for neuron in neuralnet[i]:
38             for j in range(len(inputs)):
39                 neuron['weights'][j] -= l_rate * neuron['weight_change'] * inputs[j]
40                 neuron['weights'][-1] -= l_rate * neuron['weight_change']

```

```

1 # Activation sum of weights
2 def sum_weight_input(weights, inputs):
3     y = weights[-1]
4     for i in range(len(weights)-1):
5         y += weights[i] * inputs[i]
6     return y
7
8 # Activation ReLU
9 def activation(swi):
10    return max(0, swi)

```

```

11
12 def activation_derivative(output):
13     if output <= 0:
14         return 0
15     else:
16         return 1

```

Of course, the code can also be found in the provided notebook, where it can be run. As can be seen in the report, every function has been defined by us and we have not used any library that implements MLP models. One aspect to mention is that we do not have a so-called loss function.

(g) We have tuned the hyperparameters and obtained the following results. As we had many difficulties implementing the training to work with batches we present two results. One for a non-batch implementation and another with batches.

1. Training with batches with learning_rate = 0.0002, batch_size = 10, nrof_epochs = 800 resulted in a 95% accuracy.
2. Training without batches with learning_rate = 0.001, nrof_epochs = 800 resulted in a 97% accuracy.

For the batch implementation the accuracy is not consistent as we got various networks with accuracies ranging from 70% to 95%.

(h) The stopping criteria is the number of epochs where the loss starts to converge. We found that the optimal number of epochs for some better results is 800.

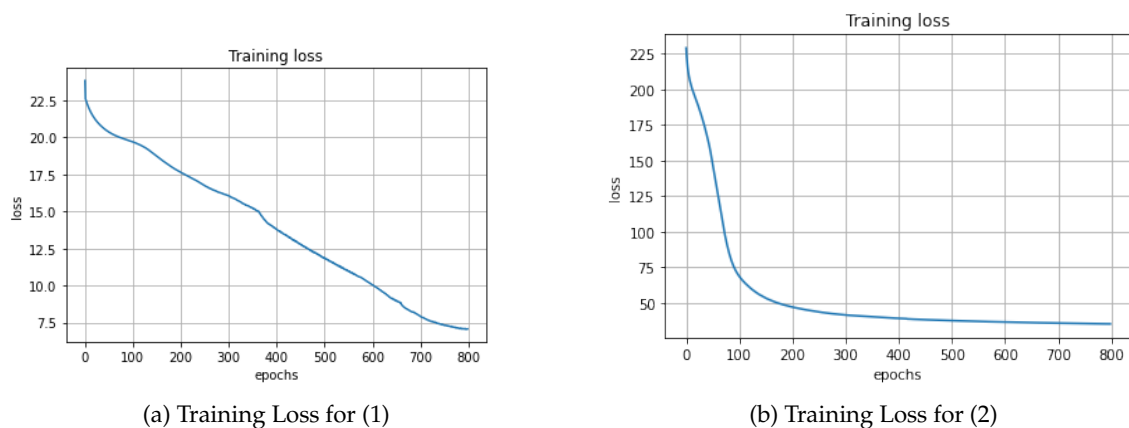


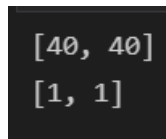
Figure 1: Training Loss Plots

(i) For the confusion matrix we have used the following code:

```

1 true_pos = 0
2 true_neg = 0
3 false_pos = 0
4 false_neg = 0
5
6 for i in range(len(testset)):
7     if testset[i][2] == results[1][0][i]:
8         if testset[i][2] == 1:
9             true_pos += 1

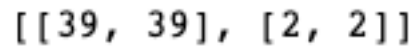
```



A 2x2 confusion matrix displayed in a dark box with white text. The first row contains the values [40, 40] and the second row contains the values [1, 1].

[40, 40]
[1, 1]

Figure 2: Confusion matrix on validation for non-batch training.



A 2x2 confusion matrix displayed in a dark box with white text. The first row contains the values [39, 39] and the second row contains the values [2, 2].

[39, 39]
[2, 2]

Figure 3: Confusion matrix on validation for batch training.

```
10     else:
11         true_neg += 1
12     else:
13         if testset[i][2] == 1:
14             false_neg += 1
15         else:
16             false_pos += 1
17
18 matrix = [[true_pos, true_neg], [false_pos, false_neg]]
19
20 print(matrix)
```

Reiterating the final accuracy we obtained for the non-batch implementation was 97% and for the batch implementation 95%.