Middle East Technical University | Department of Computer Engineering

CENG 499

Introduction to Machine Learning

Fall '2023-2024

Homework 1

Due date: 06 November 2023, Monday, 23:55

Objectives

This assignment aims to fulfill the following objectives:

- To familiarize you with the backpropagation algorithm which is crucial for neural network learning, and enable you to gain hands-on experience in solving problems with this method by employing multilayer perceptrons (feed-forward networks).
- To familiarize you with the basic operations and functionalities of Pytorch along with its high-level (abstract) features.
- To familiarize you with the hyperparameter search (grid-search) to perform model selection.
- To provide some idea of how to report statistically significant results in the case of stochasticity (randomness) while applying an ML method to a particular task.

Part 1

Multilayer-Perceptrons (MLP)(feed-forward neural networks) are universal function approximators [1], that is to say, they can approximate any given function ($y = f(x)$, functions to be approximated have to comply with certain requirements, e.g continuous, bounded functions) arbitrarily closely provided that MLP has sufficiently many nodes and layers. In addition, they are able to do feature engineering (they find the most important features automatically for the problem) at their first layer and provide generalization capabilities (e.g they can interpolate their results for the new unseen samples).

In this part, we employ MLPs for regression and classification tasks (for 3 classes) of machine learning. Figures 1 and 2 depict almost identical two MLPs on which the backpropagation update rules should be extracted. Both networks have 3 layers (the input layer ($O_i^{(0)}$), a hidden layer ($O_k^{(1)}$), and the output layer ($O_n^{(1)}$)). The subscripts represent the index of the nodes (the order is from top to bottom) in each layer and the layer index is provided in the exponent. $O_i^{(0)}$ represents the input to the network whereas $O_n^{(2)}$ is the output of it. For instance, suppose the following input vector is fed to the regression network: $[0.5, 0.2]$ ($O_0^{(0)} = 1$, $O_1^{(0)} = 0.5$, $O_2^{(0)} = 0.2$) (the first item of the input is always +1, it is the input value for bias parameters of the hidden layer). After the network performs its calculation at the output we observe a numeric value (e.g $O_0^{(2)} = 0.5$). With the same procedure we observe a vector at the output of

the classification architecture (e.g $O^{(2)} = [0.7, 0.1, 0.2]$, $O_0^{(2)} = 0.7$, $O_1^{(2)} = 0.1$, $O_2^{(2)} = 0.2$). Moreover, for both architectures $O_k^{(1)}$ represents the output of the hidden layer. The first node of the hidden layer holds a constant value of 1 again, it is needed for the bias parameters of the output layer.

Weights between the input and hidden layer are represented with $a_{ik}^{(0)}$. The index i is the index of a node in the input layer whereas k is the index of a node in the hidden layer so $a_{ik}^{(0)}$ is the weight of the connection from the i^{th} node in the input layer to k^{th} node in the hidden layer. Similarly $a_{kn}^{(1)}$ represents the weights between the hidden layer and output layer.

In both architectures, we utilize the sigmoid function as the activation function for the hidden layer of both architectures. For the output of the classification architecture, we employ the softmax function whereas no function is applied to the output of the regression architecture. With these basics, the following are the operations taking place in the regression architecture.

$$O_k^{(1)} = \sigma(\sum_{i=0} O_i^{(0)} * a_{ik}^{(0)})$$

$$O_0^{(2)} = \sum_{k=0} O_k^{(1)} * a_{k0}^{(1)}$$

Similarly, the following calculations are for the classification architecture:

$$O_k^{(1)} = \sigma(\sum_{i=0} O_i^{(0)} * a_{ik}^{(0)})$$

$$X_n^{(2)} = \sum_{k=0} O_k^{(1)} * a_{kn}^{(1)}$$

$$O_n^{(2)} = softmax(X_n^{(2)}, X^{(2)})$$

where

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$softmax(x, X^{(2)}) = \frac{e^x}{\sum_{s=0} e^{X_s^{(2)}}}$$

The softmax function requires all output node values to be known for calculations. This is the reason why a new temporary variable (X_n^2) has been introduced. Depending on this temporary variable's values, the softmax function normalizes all output values and converts them as probability scores.

Basically, we want a neural network to learn any information that we provide to it and make guesses for the never-provided information. Hopefully, it yields correct results for this unknown information. To assess whether the network learns or not, we can easily check the output of the network and the ground truth for the information. So basically we want it to output values/guesses as closely as possible to the ground truth. We can define a function that measures the closeness between the truth and the predicted values by the neural network. If they are close, the function can generate lower values otherwise higher values. So we can cast this neural network learning problem as a function optimization problem. With this respective, we alter the network weights in such a way that a particular function is minimized. Here minimization corresponds to bringing the network outputs as close as to the actual truth. From the calculus, we simply know that to minimize a function we should take steps in the reverse direction of the gradient vector. The backpropagation algorithm basically applies this logic, it is the direct application of the gradient-descent method for minimizing particular functions. So for the regression network, we can utilize the squared error (SE) function and it has the following definition:

$$SE(y, y') = (y - y')^2$$

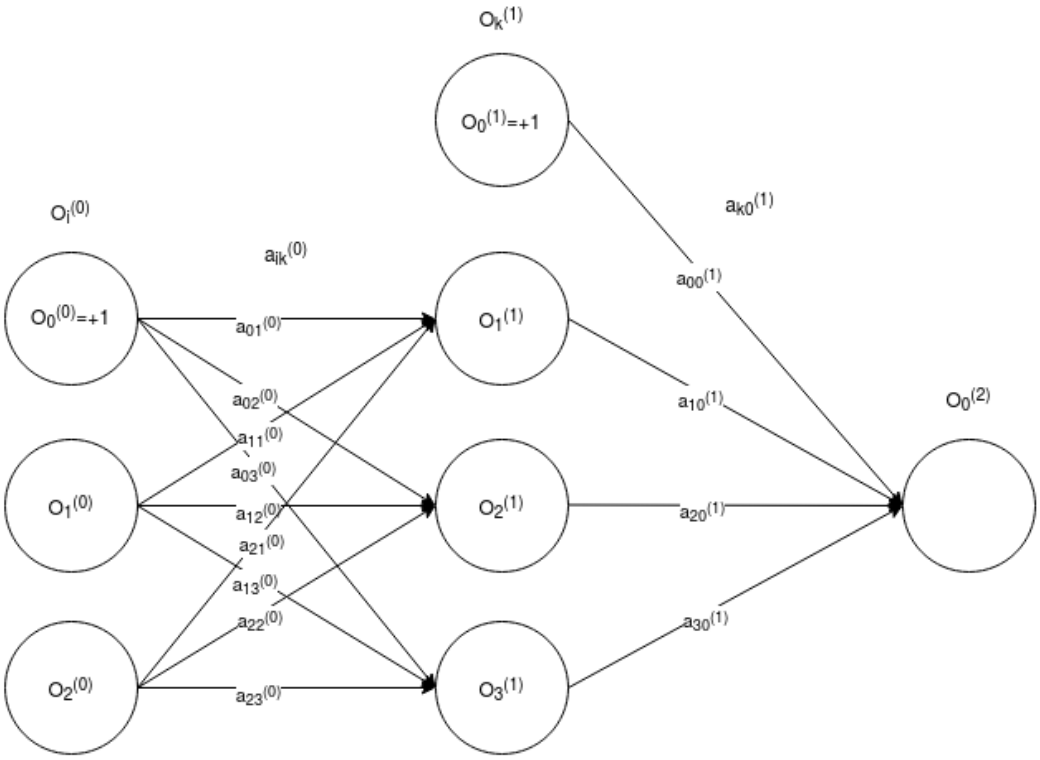

Figure 1: The MLP architecture of Part 1 for the regression problem.

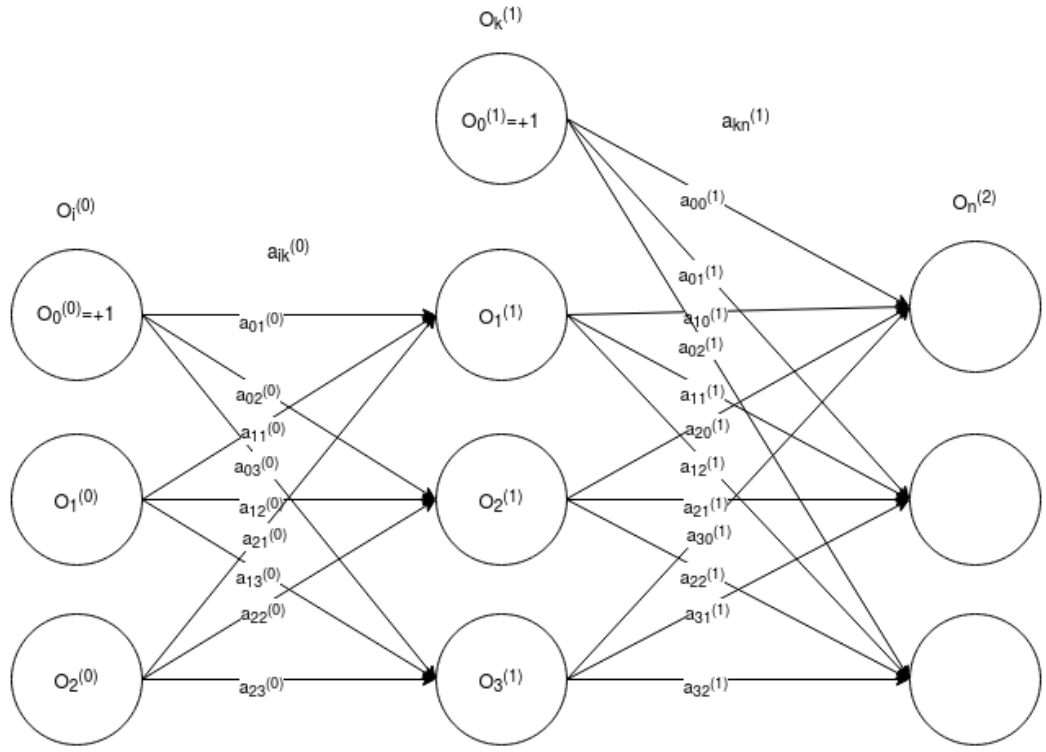

Figure 2: The MLP architecture of Part 1 for the classification problem.

where y is the ground truth and y' is the output of the network. For our setting this function can be

written as:

$$SE(y, O_0^{(2)}) = (y - O_0^{(2)})^2$$

Similarly, we can utilize the cross-entropy function for the classification architecture and it has the following form:

$$CE(l, l') = -\sum_i l_i * log(l'_i)$$

where l stores the actual ground truth for the classification task, and the l' is the output of the network. The classification task requires more output since each output provides a probability score for each different class in the classification problem. The $softmax$ function converts the raw output values into probability scores. As we can easily see the cross-entropy function works on vectors (not single values) and it aims to raise the probability score generated by the network for the correct class. In a classification problem of three classes, each class can be represented with the following vector: [1.0, 0.0, 0] (for the first class), [0.0, 1.0, 0.0] (for the second class), [0.0, 0.0, 1.0] (for the third class), which is called one-hot encoding. And the network outputs three values ([$O_0^{(2)}$, $O_1^{(2)}$, $O_2^{(2)}$] hence the cross-entropy function for our setting is as follows:

$$CE(l = [l_0, l_1, l_2], l' = O^2 = [O_0^2, O_1^2, O_2^2]) = -\sum_{i=0}^{2} l_i * log(O_i^{(2)})$$

In this assignment, you are expected to obtain update rules for $a_{ik}^{(0)}$ and $a_{kn}^{(1)}$ for both regression and classification (for regression $a_{ik}^{(0)}$ and $a_{k0}^{(1)}$) by applying the backpropagation algorithm.

The generic weight update rule is as follows:

$$w = w - \alpha * \frac{\partial E(x)}{\partial w}$$

where α is the learning rate and E is the function to be minimized with respect to w.

Part 1 Specifications

- You are expected to show/derive each weight update rule in detail and in a step-by-step fashion by utilizing the backpropagation algorithm for both regression and classification problems represented here.
- For regression, you are given a single ground truth y and its input is $[x_1, x_2]$ ($O_0^{(0)} = 1$, $O_1^{(0)} = x1$, $O_2^{(0)} = x2$) and your calculations should involve these values. So the following are the update rules for the regression problem:

$$a_{ik}^{(0)} = a_{ik}^{(0)} - \alpha \frac{\partial SE(y, O_0^{(2)})}{\partial a_{ik}^{(0)}}$$

$$a_{k0}^{(1)} = a_{k0}^{(1)} - \alpha \frac{\partial SE(y, O_0^{(2)})}{\partial a_{k0}^{(1)}}$$

- Similarly, for classification, you are given a single ground truth $[l_0, l_1, l_2]$ as its input vector is $[x_1, x_2]$. So the following updates are for the classification problem (only for the weights between the hidden and output layers):

$$a_{kn}^{(1)} = a_{kn}^{(1)} - \alpha \frac{\partial CE([l_0, l_1, l_2], O^{(2)} = [O_0^{(2)}, O_1^{(2)}, O_2^{(2)}])}{\partial a_{kn}^{(1)}}$$

- You are expected to show/derive the update rules for each problem separately (regression, classification). You can prepare your solutions in Word or Latex or you can scan or take a photograph of your handwritten solutions.
- Your solutions are going to be inspected manually, so please make sure that your solution steps are easy to follow and your writing is legible, if need be, please add explanations for the calculations.

Part 2

In this part, you are expected to implement two neural network architectures for one classification task and one regression task, respectively, by using the basic operations of Pytorch (no high-level features such as custom modules, or Pytorch modules are allowed). The classification task of this part involves learning to predict class labels (out of 3 classes) for a given unseen data sample whereas the regression task requires learning to predict a single numeric value.

To this end, for the classification task, you are expected to implement a multilayer perceptron which consists of 1 input layer (with 3 units), 1 hidden layer (with 16 units), and 1 output layer (with 3 units). The hidden layer is expected to apply the sigmoid function and the output should apply the softmax function. The softmax function (with the 3 output units) is utilized to calculate posterior class distribution for a given data sample x (e.g $P(C|x)$ = [class1 probability, class2 probability, class3 probability]). In order to predict the label of x, we choose the class whose probability score is the highest among the others. For instance, if the network outputs the following scores for x, [class1=0.6, class2=0.3, class3=0.1], we pick class1 as the label of x. Since it is a multiclass classification task, the network is trained with the cross-entropy loss function which is defined as follows:

$$CrossEntropy(groundtruth = l, predicted = p) = -\sum_i l_i * log(p_i)$$

where l_i and p_i represent the ith component of the true label and the predicted distribution, respectively. For instance, suppose that the label vector is $[1, 0, 0]$ (the instance is of type class 1) and the network has yielded $[0.4, 0.1, 0.5]$ (the network predicts class3 to be the most probable class for x). So the cross entropy loss ($CrossEntropy(predicted = [0.4, 0.1, 0.5], groundtruth = [1, 0, 0])$) is $-(1 * \log(0.4) + 0 * log(0.1) + 0 * \log(0.5)) = 0.916$. The aim of the network is to learn the class distribution of the training data by minimizing the cross entropy loss and to classify unseen data (by hoping that the training data distribution represents the unseen data distribution).

For the regression task, you are expected to implement a multilayer perceptron which consists of 1 input layer (2 units), 1 hidden layer (32 units), and 1 output layer (with 1 unit). The hidden layer is expected to apply the sigmoid function. The output layer does not apply any activation function. The output yields a single numeric value for a given data sample. The network is trained with the squared error which is defined as follows:

$$SE(groundtruth = y, predicted = y') = (y - y')^2$$

For instance, if the network outputs 0.5 for a data sample x and the ground truth of x is 1.5, the squared error score becomes $(1.5 - 0.5)^2 = 1.0$. The aim of the network is to learn the underlying generative function of the training data by minimizing the squared error loss and to predict numeric values of unseen data (by hoping that the training data distribution is capable of representing the unseen data distribution).

So far, we have considered only a single dataset while training a network. In practice, we come across three main approaches: stochastic gradient descent learning, batch gradient descent learning, and mini-batch gradient descent learning. In stochastic gradient descent learning, all the network weights are updated via a single instance. A single instance is fed to the network and backpropagation is executed for the incurred cost due to this single instance (the same procedure is repeated for others). In batch learning,

all data instances are used to update the weights in one go. All instances are fed to the network and the backpropagation algorithm is executed with the error incurred due to all instances. The minibatch gradient ascend learning stands between the other two. A specific number of instances are utilized for the weight updates (not a single instance nor all instances). The data set is divided into groups of fixed size. A group of instances is fed to the network and the backpropagation algorithm is executed over the loss incurred due to this group (the next step another group is picked and this procedure is continued). For this part, you are expected to consider only batch gradient descent learning. To this end, **we are going to utilize average/mean cross entropy loss and average/mean squared error for training the networks mentioned above**. Averaging error enables us to compare a method's performance on datasets with a different number of instances (e.g it is useful for forming the SRM plot.). In addition, it can provide insights into a method's average performance on a dataset. To clarify average cross entropy loss and squared error loss calculations, consider the following datasets and ground truths for classification and regression tasks. Each row vector represents a single data sample (in X_i, each x_i^j is a feature value of j^th instance) and its corresponding label (in L_i).

$$X_1 = \begin{bmatrix} \text{instance 1} \\ \text{instance 2} \\ \text{instance 3} \\ \text{instance 4} \end{bmatrix} = \begin{bmatrix} x_1^1 & x_2^1 & x_3^1 \\ x_1^2 & x_2^2 & x_3^2 \\ x_1^3 & x_2^3 & x_3^3 \\ x_1^4 & x_2^4 & x_3^4 \end{bmatrix}, L_1 = \begin{bmatrix} \text{label of instance 1} \\ \text{label of instance 2} \\ \text{label of instance 3} \\ \text{label of instance 4} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$X_2 = \begin{bmatrix} \text{instance 1} \\ \text{instance 2} \\ \text{instance 3} \end{bmatrix} = \begin{bmatrix} a_1^1 & a_2^1 \\ a_1^2 & a_2^2 \\ a_1^3 & a_2^3 \end{bmatrix}, L_2 = \begin{bmatrix} \text{label of instance 1} \\ \text{label of instance 2} \\ \text{label of instance 3} \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.5 \\ -0.3 \end{bmatrix}$$

$x_i^j, a_k^l \in \mathbb{R}$. Suppose that we have constructed an MLP (MLP_1) for the classification task and another MLP (MLP_2) for the regression task, and when we feed the datasets to them, we obtain the following outputs, respectively.

$$MLP_1(X_1) = \begin{bmatrix} MLP_1(\text{instance 1}) \\ MLP_1(\text{instance 2}) \\ MLP_1(\text{instance 3}) \\ MLP_1(\text{instance 4}) \end{bmatrix} = \begin{bmatrix} 0.1 & 0.8 & 0.1 \\ 0.7 & 0.2 & 0.1 \\ 0.2 & 0.25 & 0.55 \\ 0.05 & 0.93 & 0.02 \end{bmatrix}$$

$$MLP_2(X_2) = \begin{bmatrix} MLP_2(\text{instance 1}) \\ MLP_2(\text{instance 2}) \\ MLP_2(\text{instance 3}) \end{bmatrix} = \begin{bmatrix} 0.1 \\ -2.0 \\ 1.1 \end{bmatrix}$$

With these neural network outputs and ground truths, we can calculate the average cross entropy (for the classification task) and the average mean squared error loss (for the regression task as follows).

$$CrossEntropy(l = L_1, p = MLP_1(X_1)) = \begin{bmatrix} -(1 * \log(0.1) + 0 * \log(0.8) + 0 * \log(0.1)) \\ -(0 * \log(0.7) + 1 * \log(0.2) + 0 * \log(0.1)) \\ -(0 * \log(0.2) + 0 * \log(0.25) + 1 * \log(0.55)) \\ -(1 * \log(0.05) + 0 * \log(0.93) + 0 * \log(0.02)) \end{bmatrix} = \begin{bmatrix} 2.30 \\ 1.61 \\ 0.60 \\ 3.00 \end{bmatrix}$$

$$SE(L_2, MLP_2(X_2)) = \begin{bmatrix} (2.0 - 0.1)^2 \\ (1.5 - (-2.0))^2 \\ (-0.3 - 1.1)^2 \end{bmatrix} = \begin{bmatrix} 3.61 \\ 12.25 \\ 1.96 \end{bmatrix}$$

$$MeanCrossEntropy(l = L_1, p = MLP_1(X_1)) = Mean(\begin{bmatrix} 2.30 \\ 1.61 \\ 0.60 \\ 3.00 \end{bmatrix}) = \begin{bmatrix} 1.88 \end{bmatrix}$$

$$MeanSE(L_2, MLP_2(X_2)) = Mean(\begin{bmatrix} 3.61 \\ 12.25 \\ 1.96 \end{bmatrix}) = \begin{bmatrix} 5.95 \end{bmatrix}$$

Part 2 Specificiations

- The classification task involves the classification of data samples into three categories: class1, class2, and class3. Each data sample consists of three real numbered features. The labels (ground truth) of the data samples are represented with the one-hot vector encoding mechanism. The class1, class2 and class3 samples are represented with $[1,0,0], [0,1,0], [0,0,1]$, respectively (the labels are three dimensional vectors since the tasks involves three categories).

- The regression task of this part requires predicting a numerical value for data samples. Each sample consists of two real numbered features and the labels are single real numbers.

- You have been provided with two .py files (`part2_mlpclassification.py`, `part2_mlpregression.py`) and are expected to complete the missing parts (the `forward()` function, and other parts marked via '...') by using solely the basic operations of Pytorch (e.g torch.matmul, torch.argmax, torch.sigmoid, torch.from_numpy .etc). The files contain helper comments for details.

- For each task, three datasets are provided: training, validation, and test. The neural networks are trained on the training datasets. At each training step, neural networks' loss scores (mean cross-entropy loss for the classification task, mean squared error for the regression task) are measured both on training datasets and validation sets. For the classification task, as another metric, the accuracy scores are calculated. When the training operation completes, the performance of the trained network is measured on the test dataset for each task type.

- The accuracy performance metric is defined as follows:

$$Accuracy(predicted, groundtruth) = \frac{\#\text{ of data instances whose labels are predicted correctly}}{\#\text{ of instances}} * 100$$

- The implementations train the networks with the gradient descent algorithm with a fixed number of iterations by using all dataset instances (batch gradient descent learning). At the end of the training, the history of training dataset loss versus validation dataset loss is plotted (the SRM plot).

- This part aims to familiarize you with the basic working principle of Pytorch along with its basic operations. You don't need to perform any hyperparameter tuning or search in this part, which is the main consideration of the third part.

- As a reference, the following outputs are obtained at the end of the executions of the completed implementations.

```
Iteration : 14997 - Train Loss 0.2419 - Train Accuracy : 99.17 - Validation Loss : 0.8198 Validation Accuracy : 60.95
Iteration : 14998 - Train Loss 0.2418 - Train Accuracy : 99.17 - Validation Loss : 0.8198 Validation Accuracy : 60.95
Iteration : 14999 - Train Loss 0.2418 - Train Accuracy : 99.17 - Validation Loss : 0.8198 Validation Accuracy : 60.95
Iteration : 15000 - Train Loss 0.2418 - Train Accuracy : 99.17 - Validation Loss : 0.8198 Validation Accuracy : 60.95
Test accuracy : 67.62
```

Figure 3: A sample output for the classification task implementation.

```
Iteration : 1497 - Train Loss 0.5034 - Validation Loss : 2.27
Iteration : 1498 - Train Loss 0.5026 - Validation Loss : 2.27
Iteration : 1499 - Train Loss 0.5017 - Validation Loss : 2.27
Iteration : 1500 - Train Loss 0.5008 - Validation Loss : 2.27
Test loss : 2.2363
```

Figure 4: A sample output for the regression task implementation.

Note: You are not expected to obtain exactly the same outputs shown here since the results printed are likely to differ from run to run for MLPs (if we do not take any measures against it).

Part 3

Almost every machine learning method/algorithm possesses one or more hyperparameters that need to be tuned and are very crucial for the method to work properly. Multiplayer perceptrons are no exception. First of all, a structure (network topology) must be determined. The input and output layer sizes are determined via a problem to be solved but other factors such as how many hidden layers there should be, how many neuron units each of these hidden layers should contain, what type of activation functions (sigmoid, tanh, relu, leakyrelu, etc.) should be employed on these units, if a neuron should be connected to the all neuron units in the previous layer, etc. need to be determined before attempting to solve the given problem. Since we need to employ the gradient descent algorithm (via backpropagation), the learning rate by which the amount of change occurs on every weight of the network and the number of times that the weights are updated (iterations, number of epochs) should be decided. All these (and more) constitute the hyperparameters of an MLP. Within the scope of this part, the following are the hyperparameters of an MLP: number of hidden layers, number of units in a hidden layer or hidden layers, learning rate, the number of epochs (number of iterations), the activation function utilized in a neuron unit.

In order to determine which hyperparameter values are the most suitable for a given problem (the optimal hyperparameter values are task-dependent), we need to perform a hyperparameter search. In a typical machine learning problem setting, we have a training dataset and a test dataset. The training dataset is used to fine-tune the parameters (e.g neural network weights) of the algorithm whereas the test is used to assess the generalization performance of the model/algorithm (after training). Each hyperparameter value defines a different instance of a model/algorithm family (e.g MLP with 2 hidden layers, MLP with 1 hidden layer, since they can restrain or expand the function space that an algorithm searches through during learning, please recall MLPs are function approximators). So different hyperparameters result in different instances of the same model/algorithm. Each of these model instances may have different generalization capabilities. To compare their generalization capabilities, we can not use the training dataset, since a group of MLPs with different hyperparameter values can easily memorize the training dataset (overfitting). Hence, they can be easily thought to perform equally well, which is a very critical mistake in regard to generalization. The test dataset can not be used either because they can easily overfit to the test set too (this time the hyperparameters are tuned so as to perform well on the test set, which is another methodological mistake). Instead, along with the training dataset and test dataset, we have a validation dataset. On this dataset, we test the generalization capabilities of different hyperparameters. Even if different hyperparameters overfit the training data, their performance over the validation dataset provides a useful idea for generalization. By tuning the hyperparameter with respect to the validation dataset, this time, we overfit the validation dataset but we have another dataset (test data) to measure the unbiased generalization performance. In short, we are utilizing a training set to tune the parameters of a model/algorithm, a validation dataset to fine-tune the hyperparameters of a model/algorithm, and a test set to assess the generalization performance of these fine-tune parameters and hyperparameters.

During a hyperparameter search, we can fine-tune hyperparameters at a time. For instance, suppose that we have three hyperparameters: A, B, and C. First, we can set B and C some fixed values and alter A's values to determine the best value for A depending on the performance on a validation dataset. Later, we set A to its best value that has been found and C to a fixed value then alter B's value. While searching for C, we can use the best values of A and B and then alter C values. This type of strategy may be useful for some algorithms but it lacks the ability to capture the relationship between hyperparameter values, hence it is not a generic strategy to consider. A better approach is to test all possible combinations (cartesian product) of hyperparameters values, which is coined as grid search. For instance, we decide the following hyperparameters and their values: the number of neuron units in a hidden layer (referred

as number of neurons) = $\{10, 30\}$, learning rate = $\{0.001, 0.0001\}$, number of epochs = $\{150, 250\}$. The grid search test all the following hyperparameter configurations (combinations):

Number of Neurons	Learning Rate	Number of Epochs
10	0.001	150
10	0.001	250
10	0.0001	150
10	0.0001	250
30	0.001	150
30	0.001	250
30	0.0001	150
30	0.0001	250

Table 1: A sample grid-search setting with three hyperparameters.

In this part, you are expected to perform an extensive hyperparameter search for the MLP algorithm/method by utilizing a validation dataset.

Reproducibility and statistical significance of reported results are of great importance for a machine learning application/study. Especially, if there is stochasticity (randomness) at any step of a machine learning study/application/project (e.g randomness due to the method considered, randomness due to data loading, randomness while selecting instances during training, etc.), the final results reported after a single run may be a fluke and bears no, unfortunately, statistical significance, which may lead to very inaccurate and deceptive interpretations, especially while comparing different methods/algorithms over a particular ML task. As a result, the results are highly likely not to be reproducible, which especially hinders and harms advancements in machine learning research. MLPs are inherently stochastic since their weights are initialized randomly at each execution. In addition, other factors can further amplify randomness such as the usage of stochastic gradient descent (where a single instance is used to update the weights), the order of instances considered, and shuffling of the dataset instances, etc. The reader is encouraged to refer to the reproducibility page [3] of Pytorch for further details regarding the randomness sources. To alleviate the effects of these randomness sources over the results measured (e.g performance results change from run to run), we can run methods multiple times and calculate a statistical measure that can provide an idea about the distribution of the attained performance results. To this end, in this part we are going to employ the confidence interval (CI) measure (95%) which is defined as follows [4]:

$$\text{Confidence Interval}(s_1, s_2, s_3, \ldots, s_N) = \mu \pm 1.96 * \frac{\sigma}{\sqrt{N}}$$

where s_is are data samples for which the confidence interval is calculated, N is the number of s_is, μ and σ are the average and standard deviation of s_is, respectively. With this measure, we can be 95% sure that the sample mean lies in this interval ($[\mu - 1.96 * \frac{\sigma}{\sqrt{N}}, \mu + 1.96 * \frac{\sigma}{\sqrt{N}}]$).

Part 3 Specifications

- This part asks you to both implement an MLP training code for a classification task and prepare a report (in Word or Latex) about the training setting, procedures you have considered, and the decisions made throughout the process as well as the results attained.

- You have been provided with three datasets that have been obtained from the very well-known MNIST dataset and the initial source code file (`part3.py`) that loads these datasets (validation and test datasets are modified). In this part, only a single classification task is considered: identification of handwritten characters.

- In order to solve this problem, you are expected to work and implement solely multilayer perceptrons by utilizing higher-level Pytorch features (e.g loss functions, custom modules, etc.).
- You are expected to perform an extensive parameter search (grid search) over the MLP hyperparameters (number of hidden layers, number of neurons in these layers, learning rate, number of iterations (epochs), activation functions) (You are free to expand this list but these are the necessary hyperparameters for this part). During parameter search, each hyperparameter configuration should be executed 10 times and their confidence interval scores for the accuracy metric should be reported. After the hyperparameter search (determining the best hyperparameter values with respect to these interval scores (e.g picking the configuration that attains the highest mean accuracy score)), the validation dataset and the training sets should be combined into a larger dataset. The best hyperparameter setting should be employed on this new training dataset and the accuracy score on the test dataset should be measured. Again this operation should be repeated 10 times and a confidence interval for the accuracy performance score on the test dataset should be reported.
- You are free to choose which hyperparameter values to test (please state them in your report clearly) and the report format. You can utilize the Adam optimizer rather than the SGD optimizer of Pytorch. If you know or have devised a way to eliminate the search over the number of iterations (epochs) parameter, you don't need to perform a hyperparameter search over it. If so, please explain how you have achieved eliminating it in your report.
- You may consider different types of activation functions rather than sigmoid and tanh (or you can test only these two). You may refer to the Pytorch documentation for other activation functions.
- During training you can prefer to use batch gradient descent learning or minibatch gradient descent learning. If you consider choosing the second way, the minibatch size becomes another hyperparameter whose optimal value should be determined via the grid search.
- If you would like, you can save/load your trained models via `torch.save` and `torch.load`. In addition, if you would like to store temporarily or make a copy of the current parameter values of the network you can use the following code snippet:

```
import copy
...
...
stored_weights = copy.deepcopy(model.state_dict()) // saving current
    weight values
...
...
...
// restore the previously saved weight values
// the current weight values of the model are overwritten with
    stored_weights
model.load_state_dict(stored_weights)
```

- You are expected to test at least 10 hyperparameter configurations for this part. You may provide SRM plots (by taking the average of both train and validation losses over 10 runs) for the hyperparameter search procedure. The hyperparameters and their values that you have tried should be visible in your implementations.
- You can provide a single source code that performs all the work of this part in a single run or provide separate python files for each different hyperparameter configuration. If you consider the second way, please compress all these source code files into a single file during the submission.
- The input layer of your MLP design should consist of 784 units (the original handwritten digits are provided as 28x28 gray-scale images, and each image has been flattened to a vector of size

28*28=784 in order to be fed to the network as an input). As for the output layer size, 10 must be considered since there are 10 different handwritten digits (from 0 to 9, 10 classes).

- In Pytorch, there are more than one ways to implement and train via the cross entropy loss function. If you consider using **torch.nn.CrossEntropyLoss**, you should not apply the softmax function at the output layer since this function's implementation contains the softmax function already. In this case, your model implementation is going to provide raw output values (without the softmax function) when you try to predict the label of an instance. To be able to obtain probability distribution scores, you should explicitly apply the softmax function, especially when checking the accuracy performance on the training, validation, and test datasets. To this end, the following code snippet can be utilized:

```
...
softmax_function = nn.Softmax(dim=1)
...
nn_output = model(training_dataset) // model does not apply the softmax
    function due to nn.CrossEntropyLoss
loss = loss_function(nn_output, groundtruth)
...
...
probability_scores = softmax_function(nn_output) // we explicitly apply
    the softmax function to get probability scores
...
```

- You don't need to set any seed values for Pytorch or Numpy at the very beginning of source files (The training dataset of this part comes already shuffled. So the data loading procedure is deterministic). In this part, we attempt to alleviate the effect of the randomness emerging by calculating a statistical metric over the results attained.

- Please add explanatory comments on your implementation since all the work is going to be graded manually. Please, in your report, explain your training setup (hyperparameters, values, neural network architecture, etc.), procedure (how you have trained your model, the accuracy performance results of hyperparameter configurations on the validation dataset, .etc). and the final test set accuracy results (all result's confidence intervals should be provided explicitly).

- Additionally, please answer the following questions in your report:
  - What type of measure or measures have you considered to prevent overfitting?
  - How could one understand that a model being trained starts to overfit?
  - Could we get rid of the search over the number of iterations (epochs) hyperparameter by setting it to a relatively high value and doing some additional work? What may this additional work be? (Hint: You can think of this question together with the first one.)
  - Is there a "best" learning rate value that outperforms the other tested learning values in all hyperparameter configurations? (e.g it may always produce the smallest loss value and highest accuracy score among all of the tested hyperparameter configurations.).
  - Is there a "best" activation function that outperforms the other tested activation functions in all hyperparameter configurations? (e.g it may always produce the smallest loss value and highest accuracy score among all of the tested hyperparameter configurations.).
  - What are the advantages and disadvantages of using a small learning rate?
  - What are the advantages and disadvantages of using a big learning rate?
  - Is it a good idea to use stochastic gradient descent learning with a very large dataset? What kind of problem or problems do you think could emerge?

– In the given source code, the instance features are divided by 255 (Please recall that in a gray scale-image pixel values range between 0 and 255). Why may such an operation be necessary? What would happen if we did not perform this operation? (Hint: These values are indirectly fed into the activation functions (e.g sigmoid, tanh) of the neuron units. What happens to the gradient values when these functions are fed with large values?)

Regulations

1. You are expected to write your code in Python by using Pytorch, Numpy, copy, and Matplotlib libraries.
2. For Part 3, please include a README file explaining the source code file/s (along with their purpose) and how to run them if you have divided your work into multiple files.
3. The test set is only used to assess a model's unbiased generalization performance. You must not use it during training or hyperparameter optimization.
4. Falsifying results or changing the composition of training, validation, and test data is strictly forbidden, and you will receive 0 if this is the case. Your programs will be examined to see if you have actually reached the results and if it is working correctly.
5. **Late Submission:** You have a total of 5 late days for all homework without receiving a penalty. As soon as you have depleted your quota, penalization will be in effect. The late submission penalty will be calculated using $5d^2$, that is, 1 day late submission will cost you 5 points, 2 days will cost you 20 points, and 3 days will cost you 45 points. No late submission is accepted after reaching a total of 3 late days.
6. **Cheating:** Using any piece of code that is not your own is strictly forbidden and constitutes cheating. This includes friends, previous homework, or the internet. However, example code snippets shared on PyTorch's website can be used. **We have a zero-tolerance policy for cheating**. People involved in cheating will be punished according to university regulations.
7. **Discussion:** You must follow ODTUClass for discussions and possible updates/corrections on a daily basis. If you think that your question concerns everyone, please ask them on ODTUClass.
8. **Evaluation:** Your assignment is going to be graded manually.

When can I train my network?

You can always use your local computer's CPU/GPU to train the models. You can use lab computers in the department (i.e., inek machines) as they have PyTorch - CUDA. However, please check if there are any processes already running on the CPU/GPU of the same computer before running your program so that you will not slow down each other. To check tasks on the GPU, you can use the nvidia-smi command. Another option could be Google Colab even though it imposes some restrictions on free users.

Submission

Submission will be done via the ODTUClass system. For Part 1, you are expected to upload a single pdf file named **part1.pdf**. For Part 2 you are expected to upload the completed **part2_mlpclassification.py** and **part2_mlpregression.py** files. For Part 3, you are expected to upload your report (**report.pdf**) along with its implementation. For the implementations, if you have aggregated all your work into a single file, you can upload it under the name **part3.py** or if you have divided your implementation into multiple

files, all of them should be compressed (along with the README file) into a single file (**part3.zip**) and this file should be submitted.

References

1. Kurt Hornik, Maxwell Stinchcombe, Halbert White, Multilayer feedforward networks are universal approximators, Neural Networks, Volume 2, Issue 5, 1989.
2. For the `backpropagation` algorithm, you can refer to:
   - Lecture notes
   - Machine Learning, Tom M. Mitchell, Chapter 4.
   - Pattern Recognition and Machine Learning, Christopher M. Bishop, Chapter 5.
   - Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 11.
   - Pattern Classification, 2nd Edition by Richard O. Duda, Peter E. Hart, David G. Stork, Chapter 6.
   - Neural Networks and Learning Machines, 3rd edition, Simon Haykin, Chapter 4.
3. https://pytorch.org/docs/stable/notes/randomness.html
4. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 19.
5. Announcements Page
6. Discussion Forum