

# CENG 499

## Introduction to Machine Learning

Fall 2024-2025

### Homework 1

---

Due date: 05 November 2024, Tuesday, 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 0

The learning (statistical learning) problem can be considered as "finding a desired dependence using a limited number of observations" (Vapnik Chapter 1). The solution to this problem can be formulated as a function identification/search problem of the following form (Vapnik Chapter 1):

$$\alpha^* = \arg \min_{\alpha} R(\alpha) = \int L(y, f(x, \alpha)) p(x, y)$$

Where  $f(x, \alpha)$  is the function implemented by a method (learning machine, statistical model),  $\alpha$  is the set of particular parameter/hyperparameter values,  $y$  is the ground truth information for a particular data instance  $x$ ,  $L$  is the loss function measuring the discrepancy between ground truth and the output of the method,  $p(x, y)$  is the joint probability function for a data instance  $x$  and its associated information  $y$  (e.g., label),  $R$  is the expected loss (risk) associated with a particular hyperparameter/parameter value  $\alpha$ . The mathematical formulation states that to solve the learning problem, the parameter/hyperparameter values of the method (learning machine) that attain the minimum expected risk value need to be found. Since, in practice, we have only a limited number of samples (e.g., data points), the expected risk formulation can be approximated via the empirical risk (Vapnik Chapter 1):

$$\alpha^* = \arg \min_{\alpha} R_{emp} = \frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i, \alpha))$$

. Typically, there are three main types of learning problems: pattern recognition (classification), regression, and density estimation (Vapnik Chapter 1). Within the scope of this assignment, our focus will be on classification and regression problems.

The learning problem is ill-posed (Alpaydm Chapter 2) due to the fact that there is not a single unique solution for it. This is the basic reason why there are many machine learning models/methods/algorithms that differ in the assumptions (hypothesis) they make regarding this problem. Hyperparameters of models/methods/algorithms determine the space of functions/hypotheses implemented by the models/methods/algorithms, whereas parameters determine the function/hypothesis implemented. Within the scope of machine learning, the main goal of learning is to obtain models that can generalize to unseen cases (e.g., determining labels of unseen data points) by using all currently available data. To obtain the best parameter/hyperparameter values for methods (estimating model parameters/hyperparameters ( $\alpha$ )) to solve a learning problem, there are mainly two estimation approaches: the frequentist estimation and the Bayesian estimation. A frequentist estimation obtains a single set of parameter/hyperparameter values whereas the Bayesian estimation aims to define a probability distribution over parameter/hyperparameter values to assess the uncertainty in parameter values for solving a given problem (Bishop Chapter 1). One of the most utilized frequentist estimation approaches is the maximum likelihood estimation (MLE) (Alpaydm Chapter 4, Bishop Chapter 1). MLE aims to find  $\theta^*$  such that  $p(D|\theta)$  (the joint probability of data instances for given parameter  $\theta$ ) is the maximum for a given set of data instances, whereas the Bayesian estimation approach models  $P(\theta|D)$  (the probability of parameter for a given dataset  $D$ ) and uses this probability distribution function for solving learning problems. The Bayesian approach includes MLE as a special case:

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)}$$

As can be seen, the Bayesian estimation approach treats model parameters/hyperparameters as random variables and aims to estimate probability distribution over them by utilizing a given dataset. Whereas MLE finds a point estimate (single value, no probability distribution over parameter values) for model parameters/hyperparameters. Both approaches have their own merits, advantages, and disadvantages. However, due to the Bayesian approach's computational demands, MLE is generally utilized in solving machine learning problems.

### 0.0.1 Maximum Likelihood Estimation

Suppose that we have a set of data points denoted  $D = \{\vec{x}_i | i = 1..N\}$  where  $i^{th}$  data instance is represent with  $\vec{x}_i$  (as a high dimensional vector). Let's assume that each data point  $\vec{x}_i$  is generated by an unknown data generation whose generation probability is  $q(\vec{x}_i)$  (we assume data instances are identically distributed via  $q$ ). To approximate this unknown data probability distribution, we can build a model that models  $p(\vec{x}_i|\theta)$ , where  $\theta$  is the model parameter. Once we obtain/learn  $\theta$ , we can use it to measure the likelihood probability of new data instances.

In order to approximate  $q(\vec{x}_i)$  with  $p(\vec{x}_i|\theta)$ , MLE finds a parameter value such that the likelihood of data points is maximized. Mathematically:

$$\theta_{MLE} = \arg \max_{\theta} p(D|\theta) \tag{1}$$

$$= \arg \max_{\theta} p(\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N | \theta) \tag{2}$$

Further, if we assume the data points are independently distributed:

$$\theta_{MLE} = \arg \max_{\theta} p(\vec{x}_1|\theta)p(\vec{x}_2|\theta)\dots p(\vec{x}_N|\theta) \tag{3}$$

$$= \arg \max_{\theta} \prod_{i=1}^N p(\vec{x}_i|\theta) \tag{4}$$

When data instances have associated labels, the data set can be represented as  $\bar{D} = \{(\vec{x}_i, label_i) | i = 1..N\}$  where  $\vec{x}_i$  is the  $i^{\text{th}}$  data instance and  $label_i$  is the label associated with it. So in this case, we can assume data instances and their labels are generated with an unknown data generation process whose generation probability distribution is  $q(\vec{x}_i, label_i)$ . Similarly, we can build a model that represents  $p(\vec{x}_i, label_i | \theta)$  and learn a parameter value that approximates  $q(\vec{x}_i, label_i)$  via MLE. Mathematically:

$$\theta_{MLE} = \arg \max_{\theta} p(\bar{D} | \theta) \quad (5)$$

$$= \arg \max_{\theta} p((\vec{x}_1, label_1), (\vec{x}_2, label_2), \dots, (\vec{x}_N, label_N) | \theta) \quad (6)$$

Further, if we assume each data instance-label pair is independently distributed:

$$\theta_{MLE} = \arg \max_{\theta} p(\vec{x}_1, label_1 | \theta), p(\vec{x}_2, label_2 | \theta), \dots, p(\vec{x}_N, label_N | \theta) \quad (7)$$

$$= \arg \max_{\theta} \prod_{i=1}^N p(\vec{x}_i, label_i | \theta) \quad (8)$$

After learning  $p(\vec{x}_i, label_i | \theta)$ , we can label a new data instance  $\vec{x}_y$ . Suppose that a label value can take two values,  $a$  and  $b$ , without loss of generality. To label  $\vec{x}_y$ , we can check the probability values:  $p(\vec{x}_y, a | \theta)$  and  $p(\vec{x}_y, b | \theta)$  since  $p(\vec{x}_i, label_i | \theta)$  provides a probability value as to how likely a data instance and a label value coexist (co-generated). Thus, if  $p(\vec{x}_y, a | \theta) > p(\vec{x}_y, b | \theta)$  ( $a$  is more likely to be associated with  $\vec{x}_y$ ), we can label  $\vec{x}_y$  as  $a$ , otherwise  $b$  (if  $p(\vec{x}_y, a | \theta) = p(\vec{x}_y, b | \theta)$ , we can label it with either label or defer labeling). Furthermore, we can observe that the labeling can be done via the conditional probabilities  $(p(label_i | \vec{x}_y), \theta)$  since:

$$p(\vec{x}_y, a | \theta) = \frac{p(a | \vec{x}_y, \theta)}{p(\vec{x}_y | \theta)}, p(\vec{x}_y, b | \theta) = \frac{p(b | \vec{x}_y, \theta)}{p(\vec{x}_y | \theta)}$$

While comparing, the common term  $p(\vec{x}_y | \theta)$  can be eliminated. As a consequence, for labeling, we don't need to approximate the joint distribution  $p(\vec{x}_i, label_i | \theta)$ . Instead, we can try modeling  $p(label_i | \vec{x}_i, \theta)$ , which can reduce computation because  $p(\vec{x}_i, label_i | \theta)$  includes more information and hence poses a harder learning problem compared to  $p(\vec{x}_i, label_i | \theta)$ . So, instead of learning about the coexistence of a data instance and a label via  $p(\vec{x}_y, label_i | \theta)$ , we can learn how an instance can be labeled via  $p(label_i | \vec{x}_y, \theta)$ . If we assume the unknown labeling procedure has a labeling distribution  $q(label_i | \vec{x}_i)$ , we can build a model that represents  $p(label_i | \vec{x}_i, \theta)$  and learn the parameter via MLE to approximate  $q(label_i | \vec{x}_i)$  (the unknown labeling probability distribution). We can separate the dataset  $\bar{D}$  into two sets:  $L = \{label_i | i = 1..N\}$  (labels) and  $X = \{\vec{x}_i | i = 1..N\}$  (data instances). With the conditional function  $p(label_i | \vec{x}_i, \theta)$ , we aim to model:

$$\theta_{MLE} = \arg \max_{\theta} p(L | X, \theta) \quad (9)$$

$$= \arg \max_{\theta} p(label_1, label_2, \dots, label_N | \vec{x}_1, \vec{x}_2, \dots, \vec{x}_N, \theta) \quad (10)$$

If we assume labels are independent of each other and the label is determined solely by the data instance it is associated with:

$$\theta_{MLE} = \arg \max_{\theta} p(label_1 | \vec{x}_1, \theta) p(label_2 | \vec{x}_2, \theta) \dots p(label_N | \vec{x}_N, \theta) \quad (11)$$

$$= \arg \max_{\theta} \prod_{i=1}^N p(label_i | \vec{x}_i, \theta) \quad (12)$$

Modeling/approximating/learning the likelihood of data instances is coined as the density estimation problem (Alpaydm Chapter 4), whereas modeling/approximating the unknown labeling procedure that

leads to the emergence of label-data instance pairs (dataset) is coined as either a classification or regression problem depending on the values that labels can take (Vapnik Chapter 1). MLE can be used to solve all these three learning problems.

Since our focus is on classification and regression problems for this assignment, the following subsections discuss the solution strategies for these problems.

## 0.1 Solution Strategies for a Classification Problem (Bishop Chapter 1)

1. Learn/estimate the joint probability function  $p(\vec{x}, label)$  via a given dataset and use this estimated probability function to get conditional class label probabilities:

$$p(label|\vec{x}) = \frac{p(\vec{x}, label)}{p(\vec{x})} = \frac{p(\vec{x}, label)}{\sum_{i=1}^K p(\vec{x}, label = l_i)}$$

Where  $l_i$  denotes class label values of the given classification problem. After calculating this conditional probability for every class label ( $p(label = l_1|\vec{x}), p(label = l_2|\vec{x}), \dots, p(label = l_K|\vec{x})$ ), we can assign the label  $l_i$  that attains the highest probability score to a given data instance  $\vec{x}$  in order to minimize the error probability as discussed previously. As a broader decision strategy, in a given problem, if we are given risk/loss scores for each labeling decision (e.g., labeling an instance as  $l_1$ ), we can utilize the probability score associated with each label to calculate the expected risk score for our labeling decisions. Let  $R(\alpha_i|\vec{x})$  be the expected risk/loss score when we label a data instance  $\vec{x}$  as  $l_i$  (Alpaydm Chapter 3):

$$R(\alpha_i|\vec{x}) = \sum_{j=1}^K \lambda_{ij} * p(label = l_j|\vec{x})$$

where  $\lambda_{ij}$  is the loss/risk incurred if we label  $\vec{x}$  as  $l_i$  when it actually belongs to the class  $l_j$ . For instance, in a medical application, labeling a patient who has cancer as healthy can lead to more disastrous results (high risk/loss, the life of the patient is at stake) compared to labeling a healthy patient as healthy (low loss risk/loss, several costly medical tests have been conducted). To incur the least amount of loss/risk we rationally need to pick the class label that has the smallest expected risk for a given data instance. As can be seen, decision via loss/risk minimization encompasses the minimizing labeling error probability as a special case where each  $\lambda_{ij} = 1$ . Since probability scores can give us an idea about how a method is certain of its labeling decisions, we can add a *reject* decision option. For instance, if the maximum class probability is below some threshold, a method can delegate making a labeling decision to a human decider, hence it can reject labeling an instance. Furthermore, the reject decision can have associated risk/loss values and can be included in the risk/loss minimization strategy (if the reject decision attains the smallest expected risk score, a method can reject the labeling decision). Machine learning methods that estimate  $p(x)$  (directly or indirectly) are called generative models because they model the likelihood of data instances and can be used to generate new data by sampling data points from  $p(x)$ .

2. Directly learn/estimate the class conditional probability function  $p(label|\vec{x})$  from a given dataset, instead of the joint probability  $p(label, \vec{x})$  because the joint probability function contains much more information ( $p(\vec{x})$  the likelihood of data instances, for labeling decision we do not need this information, the conditional probability function suffices.). We can utilize the error minimization strategy or the broader strategy of minimizing risk/loss with the estimated class conditional probabilities for labeling new data instances. Machine learning models that estimate the conditional probability scores are coined as discriminative models.

3. Instead of modeling a probability function for labeling, directly learn the labeling function (discriminant)  $f(\vec{x})$  from a given dataset and use  $f(\vec{x})$  to label new data instances. Machine learning models such as support vector machines and decision trees implement a discriminate  $f(\vec{x})$ . Since these models do not provide a probability score for labeling, we cannot utilize the labeling error minimization or risk/loss minimization strategies. Furthermore, a reject option is not applicable.

Utilization of one of these solution strategies is task/data/budget (time) dependent. All these strategies have their advantages and disadvantages over one another (Bishop chapter 1). Within the scope of this assignment, we are going to consider the second strategy for solving classification problems. We are going to employ multilayer perceptions (MLPs) to model  $p(label|\vec{x})$  and learn the MLP parameters/hyperparameters via MLE.

## 0.2 Solution Strategies for Regression Problems

The solution strategies are quite similar to those of the classification case.

1. Learn/estimate the joint probability function  $p(\vec{x}, t)$ ,  $t$  is a random variable for a numerical value, via a given dataset and use this estimated probability function to obtain the conditional probability  $p(t|\vec{x})$  function:

$$P(t|\vec{x}) = \frac{p(\vec{x}, t)}{p(\vec{x})} = \frac{p(\vec{x}, t)}{\int p(x, t) dt}$$

The conditional probability function does not provide a single value for a given data instance  $\vec{x}$  (it provides a distribution over values). To assign a numerical value (to label) we can calculate the expected numerical value for the data instance with the conditional function:

$$\int t p(t|\vec{x}) dt$$

2. Directly learn/estimate the class conditional probability function  $p(t|\vec{x})$  from a given dataset, instead of the joint probability  $p(t, \vec{x})$ . Similar to the first strategy, we can calculate the expected value for a given data instance.
3. Instead of modeling a probability function for labeling, directly learn the labeling function (regression function)  $f(\vec{x})$  from a given dataset and use  $f(\vec{x})$  to label new data instances.

Similar to the classification case, utilization of one of these solution strategies is task/data/budget (time) dependent. All these strategies have their advantages and disadvantages over one another (Bishop Chapter 1). Within the scope of this assignment, we are going to consider the second strategy for solving regression problems. We are going to employ multilayer perceptions (MLPs) to model  $p(t|\vec{x})$  and learn the MLP parameters/hyperparameters via MLE.

## 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. As a result, we can utilize them to model the probability distribution function  $p(label|\vec{x}, \theta)$  for labeling problems (classification and regression). In addition, they are able to do feature engineering (they find/create/extract the most important features automatically for a given problem) at their hidden layers and possess generalization capabilities (e.g., they can interpolate their results for new unseen data samples).

In this part, we employ MLPs for regression and classification tasks (for 3 classes) of machine learning. Figures 1 and 2 depict two MLPs on which the backpropagation update rules should be extracted and implemented. Both networks have 3 layers (the input layer ( $I$ ), a hidden layer ( $H$ ), and the output layer ( $O$ )). The subscripts represent the index of the nodes (the order is from top to bottom) in each layer.  $I_i$  represents the  $i^{\text{th}}$  input to the network whereas  $O_k$  is the  $k^{\text{th}}$  output of it. For instance, suppose the following input vector is fed to the regression network:  $[0.5, 0.2]$  ( $I_0 = 1$ ,  $I_1 = 0.5$ ,  $I_2 = 0.2$ ) (the first component 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.,  $O_0 = 0.5$ ). With the same procedure, we observe a vector at the output of the classification architecture (e.g.,  $O = [0.7, 0.1, 0.2]$ ,  $O_0 = 0.7$ ,  $O_1 = 0.1$ ,  $O_2 = 0.2$ ). Moreover, for both architectures,  $H_j$  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  $w_{ij}$ . The index  $i$  is the index of a node in the input layer whereas  $j$  is the index of a node in the hidden layer so  $w_{ij}$  is the weight of the connection from the  $i^{\text{th}}$  node in the input layer to  $j^{\text{th}}$  node in the hidden layer. Similarly,  $\gamma_{jk}$  represents the weights between the hidden layer and the output layer. Several weights are explicitly denoted in the figures for further clarification.

In both architectures, you are expected to utilize the sigmoid function as the activation function for the hidden layers. For the output of the classification architecture, we employ the softmax function whereas no function is applied to the output of the regression architecture (in other words, we utilize the identity function  $f(x) = x$  for the regression architecture output). With these basics, the following are the operations taking place in the regression architecture.

$$H_j = \sigma\left(\sum_{i=0}^2 I_i * w_{ij}\right)$$

$$O_0 = \sum_{j=0}^3 H_j * \gamma_{j0}$$

Similarly, the following calculations are for the classification architecture:

$$H_j = \sigma\left(\sum_{i=0}^4 I_i * w_{ij}\right)$$

$$X_k = \sum_{j=0}^3 H_j * \gamma_{jk}$$

$$O_k = \text{softmax}(X_k, X)$$

where

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

$$X = [X_0, X_1, X_2]$$

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

The softmax function requires all output node values to be known for calculations. This is the reason why a new temporary variable ( $X_k$ ) has been introduced. Depending on this temporary variable's values, the softmax function normalizes all output values and converts them as probability scores. The architecture models  $p(\text{label}|\vec{x}, \theta)$ , where  $\theta$  includes all the weights of it.

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

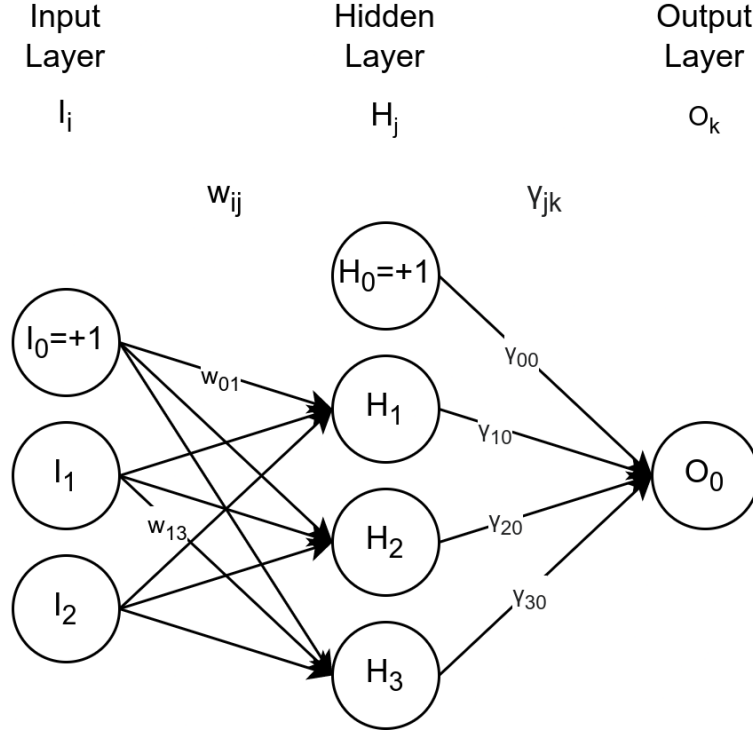


Figure 1: The MLP architecture of Part 1 for the regression problem.  $w_{ij}$  denotes the weights between the input and hidden layers. Similarly,  $\gamma_{jk}$  denotes the weights between the hidden layer and output layers.

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 respect, 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 possible 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 loss functions (e.g., loss functions should be differentiable to apply the backpropagation algorithm). So for the regression network, we can utilize the squared error (SE) function, which has the following definition:

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

where  $y$  is the ground truth and  $y'$  is the output of the network. For our setting, this function can be written as:

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

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

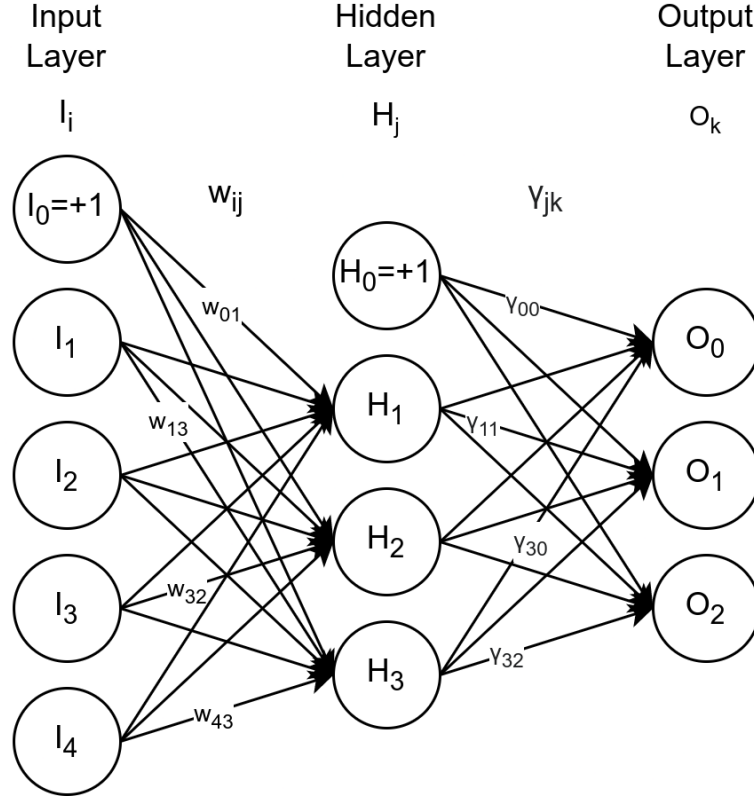


Figure 2: The MLP architecture of Part 1 for the classification problem.  $w_{ij}$  denotes the weights between the input and hidden layers. Similarly,  $\gamma_{jk}$  denotes the weights between the hidden layer and output layers.

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  $[O_0, O_1, O_2]$  hence the cross-entropy function for our setting is as follows:

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

In this part, you are expected to obtain update rules for  $w_{ij}$  and  $\gamma_{jk}$  for both regression and classification architectures separately by applying the backpropagation algorithm. Later, you are expected to implement these update rules for given regression and classification tasks.

The generic weight update rule is as follows:

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

where  $\alpha$  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 a step-by-step fashion by utilizing the backpropagation algorithm for both regression and classification problems represented here.



- For regression, assume that you are given a single ground truth  $y$  and its input  $[x_1, x_2]$  ( $I_0 = 1$ ,  $I_1 = x_1$ ,  $I_2 = x_2$ ), and your derivation calculations should involve these values. So the following are the update rules for the regression problem (you are expected to further expand these equations by employing the backpropagation algorithm):

$$w_{ij}^{new} = w_{ij} - \alpha \frac{\partial SE(y, O_0)}{\partial w_{ij}}$$

$$\gamma_{k0}^{new} = \gamma_{k0} - \alpha \frac{\partial SE(y, O_0)}{\partial \gamma_{k0}}$$

- Similarly, for classification, assume that you are given a single ground truth  $[l_0, l_1, l_2]$  (in the one-hot encoding format,  $\sum_i l_i = 1$ ) and its input vector  $[x_1, x_2, x_3, x_4]$ , and your derivation calculations should involve these values. So the following updates are for the classification problem (you are expected to further expand these equations by employing the backpropagation algorithm):

$$w_{ij}^{new} = w_{ij} - \alpha \frac{\partial CE([l_0, l_1, l_2], O = [O_0, O_1, O_2])}{\partial w_{ij}}$$

$$\gamma_{jk}^{new} = \gamma_{jk} - \alpha \frac{\partial CE([l_0, l_1, l_2], O = [O_0, O_1, O_2])}{\partial \gamma_{jk}}$$

- You are expected to show/derive the update rules for each problem separately (regression, classification). You are expected to prepare your solutions using LaTeX.
- 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.
- Instead of providing each single weight update one by one (e.g.,  $w_{0,0}^{new}$ ,  $w_{3,2}^{new}$ ), you can provide a single update rule for all the weights between layers (e.g.,  $w_{ij}^{new} = \dots$ ,  $\gamma_{jk}^{new} = \dots$ ) to save effort and time.
- In addition to the derivations, this part requires you to implement these backpropagation update rules for solving regression and classification tasks with the MLP architectures described above.
- For the regression architecture implementation, you are given an implementation file (**mlp\_regression\_backpropagation.py**) that loads a regression problem dataset and implements a class called **MLPRegressor**. Using the class and data it trains the architecture via backpropagation and prints performance results during training. Within this class, you are expected to implement the **forward** function that runs the forward propagation operation and the weight update part in the **train** function. The class implements the regression architecture depicted in Figure 1. To simplify numerical calculation, the class separates the weights and biases associated with each layer. In other words, in the class implementation, four variables are utilized to store weights: **W**, **W\_bias**, **GAMMA**, **GAMMA\_bias** whose definitions are as follows:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}, \mathbf{W\_bias} = [w_{01} \quad w_{02} \quad w_{03}]$$

$$\mathbf{GAMMA} = \begin{bmatrix} \gamma_{10} \\ \gamma_{20} \\ \gamma_{30} \end{bmatrix}, \mathbf{GAMMA\_bias} = [\gamma_{00}]$$

The class loads initial weight values from a file so that once you correctly implement the mentioned parts within the class (forward function and weight updates), your training output should be **exactly** the same as the following:

```

1 Epoch Number: 1 - Training Mean SE: 9.892
2 Epoch Number: 2 - Training Mean SE: 1.757
3 Epoch Number: 3 - Training Mean SE: 1.098
4 Epoch Number: 4 - Training Mean SE: 1.124
5 Epoch Number: 5 - Training Mean SE: 1.140
6 Epoch Number: 6 - Training Mean SE: 1.076
7 ...
8 Epoch Number: 247 - Training Mean SE: 0.002
9 Epoch Number: 248 - Training Mean SE: 0.002
10 Epoch Number: 249 - Training Mean SE: 0.002
11 Epoch Number: 250 - Training Mean SE: 0.002

```

- For the classification architecture implementation, you are given an implementation file (**mlp\_classification\_backpropagation.py**) that loads a classification problem dataset and implements a class called **MLPClassifier**. Using the class and data it trains the architecture via backpropagation and prints performance results during training. Within this class, you are expected to implement the **forward** function that runs the forward propagation operation and the weight update part in the **train** function. The class implements the classification architecture depicted in Figure 2. To simplify numerical calculation, the class separates the weights and biases associated with each layer. In other words, in the class implementation, four variables are utilized to store weights: **W**, **W\_bias**, **GAMMA**, **GAMMA\_bias** whose definitions are as follows:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}, W\_bias = [w_{01} \quad w_{02} \quad w_{03}]$$

$$GAMMA = \begin{bmatrix} \gamma_{10} & \gamma_{11} & \gamma_{12} \\ \gamma_{20} & \gamma_{21} & \gamma_{22} \\ \gamma_{30} & \gamma_{31} & \gamma_{32} \end{bmatrix}, GAMMA\_bias = [\gamma_{00} \quad \gamma_{01} \quad \gamma_{02}]$$

The class loads initial weight values from a file so that once you correctly implement the mentioned parts within the class (forward function and weight updates), your training output should be **exactly** the same as the following:

```

1 Epoch Number: 1 - Training Mean CE: 1.303 - Training Accuracy: 33.33
2 Epoch Number: 2 - Training Mean CE: 1.125 - Training Accuracy: 37.33
3 Epoch Number: 3 - Training Mean CE: 1.057 - Training Accuracy: 46.67
4 Epoch Number: 4 - Training Mean CE: 1.014 - Training Accuracy: 63.33
5 Epoch Number: 5 - Training Mean CE: 0.976 - Training Accuracy: 66.00
6 Epoch Number: 6 - Training Mean CE: 0.936 - Training Accuracy: 70.00
7 ...
8 Epoch Number: 117 - Training Mean CE: 0.114 - Training Accuracy: 97.33
9 Epoch Number: 118 - Training Mean CE: 0.113 - Training Accuracy: 97.33
10 Epoch Number: 119 - Training Mean CE: 0.113 - Training Accuracy: 97.33
11 Epoch Number: 120 - Training Mean CE: 0.112 - Training Accuracy: 97.33

```

## 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 617 nodes), 1 hidden layer (with 64 nodes), and 1 output layer (with 3 nodes). The hidden layer is expected to apply the tanh function and the output should apply the softmax function. The softmax function (with the 3 output units) is utilized to calculate the posterior class distribution for a given data sample  $x$  (e.g.,  $P(C|x) = [\text{class1 probability}, \text{class2 probability}, \text{class3 probability}]$ ). In order to predict the label of  $x$ , we can choose the class whose probability score is the highest among the others in order to minimize the error probability of making a wrong labeling decision. For instance, if the network outputs the following scores for  $x$ ,  $[\text{class1}=0.6, \text{class2}=0.3, \text{class3}=0.1]$ , we pick class1 as the label of  $x$  (when we pick class1 as the label, we have an error probability value of 0.4, on the other hand with class2 and class3 we have error probability values of 0.7 and 0.9, respectively). Since it is a multiclass classification task, the network can be 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  $i^{th}$  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 (is rich enough to generalize)).

For the regression task, you are expected to implement a multilayer perceptron which consists of 1 input layer (2 nodes), 1 hidden layer (512 nodes), and 1 output layer (with 1 node). The hidden layer is expected to apply the tanh 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 labeling 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 data instance to update network weights. 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).

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

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

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

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

## Part 2 Specifications

- 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 task 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 is completed, 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 0.0324 - Train Accuracy : 98.81 - Validation Loss : 0.1873 Validation Accuracy : 94.96
Iteration : 14998 - Train Loss 0.0324 - Train Accuracy : 98.81 - Validation Loss : 0.1873 Validation Accuracy : 94.96
Iteration : 14999 - Train Loss 0.0324 - Train Accuracy : 98.81 - Validation Loss : 0.1873 Validation Accuracy : 94.96
Iteration : 15000 - Train Loss 0.0324 - Train Accuracy : 98.81 - Validation Loss : 0.1873 Validation Accuracy : 94.96
Test accuracy : 89.81
```

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

```
Iteration : 1495 - Train MSE Loss 0.1001 - Validation MSE Loss : 12.58
Iteration : 1496 - Train MSE Loss 0.1001 - Validation MSE Loss : 12.58
Iteration : 1497 - Train MSE Loss 0.1000 - Validation MSE Loss : 12.58
Iteration : 1498 - Train MSE Loss 0.1000 - Validation MSE Loss : 12.57
Iteration : 1499 - Train MSE Loss 0.0999 - Validation MSE Loss : 12.57
Iteration : 1500 - Train MSE Loss 0.0999 - Validation MSE Loss : 12.57
Iteration : 1501 - Train MSE Loss 0.0998 - Validation MSE Loss : 12.57
Test MSE loss : 1.2107
```

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 since weights are initialized randomly at the beginning of each run (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. Multilayer 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 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 regards 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 hyperparameter 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 to as number of neurons) = {10, 30}, learning rate = {0.001, 0.0001}, number of epochs = {150, 250}. The grid search tests 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 [2] 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 (Alpaydm Chapter 20):

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

where  $s_i$ s are data samples for which the confidence interval is calculated,  $N$  is the number of  $s_i$ s,  $\mu$  and  $\sigma$  are the average and standard deviation of  $s_i$ s, respectively. This measure simply calculates the interval probability for the true average performance score:  $p(\mu - 1.96 * \frac{\sigma}{\sqrt{N}} \leq \text{true average method performance} \leq \mu + 1.96 * \frac{\sigma}{\sqrt{N}}) = 0.95$ . So when applied to method performance scores, it states that with a probability score of 0.95, a model's true average performance score is within the calculated range (with 0.05 probability it may be outside this range). Hence, it can provide statistical confidence as to how a method performs on average in a given problem. In addition, the width of the interval can give an idea about how robust a method is for a given problem since if a method yields almost identical results for each performance measurement, its average performance confidence interval will be very narrow due to the variance being very small.

### Part 3 Specifications

- This part asks you to both implement an MLP training code for a classification task and prepare a report (in 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 Fashion MNIST dataset and the initial source code file (`part3.py`) that loads these datasets. 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:

```

1 import copy
2 ...
3 ...
4 stored_weights = copy.deepcopy(model.state_dict()) // saving current
   weight values
5 ...
6 ...
7 ...
8 // restore the previously saved weight values
9 // the current weight values of the model are overwritten with
   stored_weights
10 model.load_state_dict(stored_weights)

```

- You are expected to test at least 10 hyperparameter configurations for this part. You can provide SRM plots (by taking the average of both train and validation losses over 10 runs) for the hyperparameter search procedure. The hyperparameters and the 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 (clothing items are depicted as 28x28 gray-scale images, and each image has been flattened to a vector of size  $28 \times 28 = 784$  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 clothing items (from 0 to 9, 10 classes) in the dataset.
- 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:

```

1 ...
2 softmax_function = nn.Softmax(dim=1)
3 ...
4 nn_output = model(training_dataset) // model does not apply the softmax
    function due to nn.CrossEntropyLoss
5 loss = loss_function(nn_output, groundtruth)
6 ...
7 ...
8 probability_scores = softmax_function(nn_output) // we explicitly apply
    the softmax function to get probability scores
9 ...

```

- 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 while training, performing hyperparameter tuning, or preprocessing the data instances of training and validation parts.
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 72 late hours 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 (0-24 hours) late submission will cost you 5 points, 2 days (24-48 hours) will cost you 20 points, and 3 days (48-72 hours) will cost you 45 points. No late submission is accepted after 3 days from the deadline of the assignment.
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. 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** which includes derivation rules, the completed `mlp_classification.backpropagation.py`

and `mlp_regression_backpropagation.py` files. 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. You are expected to prepare your reports using Latex.

## References

1. Kurt Hornik, Maxwell Stinchcombe, Halbert White, Multilayer feedforward networks are universal approximators, Neural Networks, Volume 2, Issue 5, 1989.
2. <https://pytorch.org/docs/stable/notes/randomness.html>
3. Vapnik: Vapnik, Vladimir (1999). The Nature of Statistical Learning Theory. Springer: New York.
4. Alpaydm: Introduction to Machine Learning, 4th edition, Ethem Alpaydm.
5. Bishop: Pattern Recognition and Machine Learning, Christopher M. Bishop.
6. For the **backpropagation** algorithm, you can refer to:
  - Lecture notes
  - Machine Learning, Tom M. Mitchell, Chapter 4.
  - Bishop Chapter 5.
  - Alpaydm 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.
7. Announcements Page
8. Discussion Forum