

- This is an individual assignment. You are not allowed to discuss the problems with other students.
  - Make sure to write your code only in the .py files provided. Avoid creating new files. Do not rename the files or functions as it will interfere with the autograder's ability to evaluate your work correctly. Also, do not change the input or output structure of the functions.
  - When Submitting to GradeScope, be sure to submit
    1. A .zip file containing all your Python codes (in .py format) to the **Assignment 4 - Code** section on Gradescope. Do not submit data, environment files, or any other files.
    2. A .pdf file that contains your answers to the questions to the **Assignment 4 - Report** entry.
  - Part of this assignment will be auto-graded by Gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want. We provide some tests that you can use to check that your code will be executed properly on Gradescope. These are **not** meant to check the correctness of your answers. We encourage you to write your own tests for this.
  - Please provide the title and the labels of the axes for all the figures you are plotting.
  - If you have questions regarding the assignment, you can ask for clarifications in Piazza. You should use the corresponding tag for this assignment.
  - Before starting the assignment, make sure that you have downloaded all the data and tests related to the assignment and put them in the appropriate locations.
  - You are allowed to use **Scikit-learn** for training, testing, and evaluating algorithms such as Support Vector Machines and Decision Trees. Please do not use any other machine learning library for applying the classification algorithms. Appropriate instructions are provided for each question.
  - You cannot use ChatGPT or any other code assistance tool for the programming part; however, if you use ChatGPT to edit grammar in your report, you have to explicitly state it in the report.
-

# 1 The Building Blocks (55 points)

In this assignment, you will implement a simple neural network with the components from scratch. The goal is to build your version of a minimalistic deep learning library inspired by PyTorch. This means that **every class and method you write should be somehow interchangeable with its officially related PyTorch function.**

For starters, let us introduce the class `Layer`. This class will serve as the most general abstraction level for every component you will be implementing. It contains the following abstract methods: `init_weights`, `forward`, and `backward`. The names correspond to the algorithm parts you will implement for each required sub-class of `Layer`. Not all these methods need to be implemented for every layer; for instance, activation layers do not need to be initialized. You may find the complete definition of `Layer` in the `q1.py` file.

1. **Fully-Connected Layer.** (15 Points) Implement the `Dense` class that works with any generic input and output sizes. This class' output will be the linear combination of parameters  $W$  and inputs  $x$ , plus the bias  $b$ , as follows:

$$\text{Dense}(x) = xW^T + b$$

The input  $x \in \mathbb{R}^{n \times d}$  represents a batch of  $n$  examples each with  $d$  dimensions. Hence,  $W \in \mathbb{R}^{m \times d}$ , and  $b \in \mathbb{R}^m$ , where  $m$  is the output size. Inside the class, define the weight matrix and the bias vector. All of the `init_weights`, `forward`, and `backward` methods should be overwritten and dimensions should be checked.

To initialize this layer, you will implement the Xavier (or Glorot) distribution . For every weight tensor  $W$  , we sample from the uniform distribution on the following bounds:

$$W \sim U\left(-\sqrt{\frac{6}{d+m}}, \sqrt{\frac{6}{d+m}}\right)$$

where  $d$  and  $m$  are the input and output dimensions, as defined above. Furthermore, every bias tensor is to be initialized with all zeros.

Your code should be written in the `Dense` class located in `q1.py`.

2. **Activation Functions.** (30 Points) Implement the activation functions `softmax`, `tanh`, and `relu`. Each activation layer should implement both forward and backward passes:

- (a) Softmax (10 Points). Note that you cannot use `scipy.special.softmax` here.

$$\text{Softmax}(x_j) = \frac{\exp(x_j)}{\sum_{k=1}^n \exp(x_k)}$$

- (b) (10 Points). Note that you cannot use `np.tanh` here

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

(c) ReLU (10 points).

$$\text{ReLU}(x) = \max(x, 0)$$

Each implementation of the `forward` and `backward` methods should be in the corresponding class located in `q1.py` (`ReLULayer`, `TanhLayer`, `SoftmaxLayer`).

3. A Loss Function. (10 Points) Implement the cross-entropy loss layer. When working with this class, you should bear in mind that any generic neural network outputs should be well processed by the forward pass. Hence the forward method should compute the cross-entropy loss given the predicted probabilities and target labels, while the backward pass should compute the gradient. The average cross-entropy loss is given by:

$$\text{Cross-Entropy}(p, y) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p_i)$$

where  $N$  is the number of given examples (batch size). Note that while some literature might present this definition without batch size normalization, in this case, we require you to implement this version. Your code should be written in the `CrossEntropyLossLayer` class located in `q1.py`.

## 2 The Multi-layer Perceptron (10 Points)

1. Multi-layer Perceptron. (10 Points) You will now construct a multi-layer perceptron (MLP) using the layers you have implemented. For this class, you will need to write forward and backward passes, as well as, an `init_weights` method. Please refer to the provided class documentation for more technical details.

Your code should be written in the `MLP` class located in `q2.py`.

## 3 Putting It All Together (35 points)

You now will study what happens when everything is put together. To do this, you will use the **Fashion MNIST** dataset, consisting of gray-scale images of clothes resembling its well-known digit-based cousin (See Fig. 1).

We provide everything that you need in `fashion_mnist.py`; in particular:

- **FashionMNIST**: a class to load and pre-process this dataset; hence, you will not need to worry about anything but making sure your code can work with arbitrary batches of images. This function will give you `train`, `test`, and `validation` sets: we use 20% of the train data to validate the training performance after some epochs.
- **PytorchMLPFashionMNIST**: a class that implements a PyTorch model that is tailored to give; optimal performance when training with Fashion MNIST.

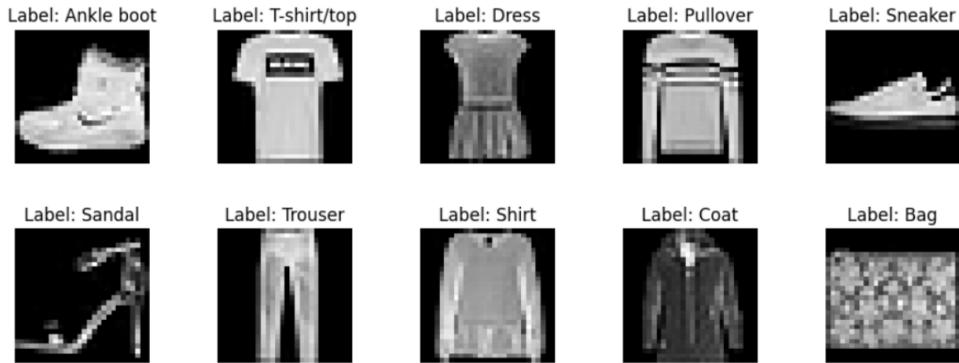


Figure 1: Examples of Fashion-MNIST Samples

- `CustomMLPFashionMNIST`: a class that inherits the `Layer` class and implements the same MLP architecture, as above, but will use all the components you worked on previously, and only those.

The provided optimal architecture to handle Fashion MNIST is the following: firstly our inputs will be  $28 \times 28$ -dimensional batches of images; then we will have:

- Input layer with 784 neurons ( $28 \times 28$  flattened Fashion MNIST images).
- First hidden layer with 256 neurons, followed by a `ReLU` activation function.
- Second hidden layer with 128 neurons, followed by a `ReLU` activation function.
- Output layer with 10 neurons (one for each clothing category in Fashion MNIST), with a `Softmax` activation to recover the final predictions.

Both `PytorchMLPFashionMNIST` and `CustomMLPFashionMNIST` come with a `train` method that takes an instance of a `FashionMNIST` object and performs *one full epoch* on the training data, as well as, validation accuracy computation. Similarly, the `test` method is provided to report the final test accuracy. Finally, after training and testing, we provide a function to plot loss and accuracy curves. All this functionality can be observed on the `__main__` part of `fashion_mnist.py` script.

1. **Validation Accuracy.** (20 pts) Train, both a PyTorch model and your customized version for around 10 epochs and a learning rate equal to 0.002. This should be enough to produce optimal models.

Plot the training loss and validation accuracy curves over each epoch. Repeat the procedure with a very small and a very large value for the learning rate. Plot the results in the report. You should have both PyTorch and customized curves in the same plot where they correspond: either loss or accuracy curves. Therefore, you should report 6 figures in total. Justify how the choice of the learning rate affects the training.

2. **Test Performance.** (15 Points) Modify the CUSTOMIZED and PyTorch models with the following configurations, and report the results in a table:

1. Change the first hidden size to  $\{8, 16, 64, 1024\}$  and keep the rest as it is.
2. Change the second hidden size to  $\{8, 16, 64, 1024\}$  and keep the rest (including the first hidden layer) as it is.
3. Remove the second input layer. Keep the layer sizes equal to their original size.

For each one, you may need to tweak your batch size and/or learning rate to be able to converge. Track the training time for each configuration; report the cases where training speeds or slows down. Record test accuracies and contrast them to any speed gains you may obtain (from either reducing or increasing hidden sizes). Justify well your observations in the report.

## 4 [BONUS] Theoretical question (extra 50 points)

This is a bonus question for extra points. This means, if you leave this field empty, you can still gain the full score for this homework by answering all previous questions correctly. However, we give extra points for this question that can sum above the maximal homework score, therefore, you can improve your score in the course.

1. (50 points) Consider a deep linear net:

$$f(x; W_{0:L}) = W_L W_{L-1} \dots W_0 x. \quad (1)$$

Let  $n_l$  be the width of  $l$ -th layer (i.e.  $W_l \in \mathbb{R}^{n_{l+1} \times n_l}$ ). The square loss of this net on a finite dataset  $X \in \mathbb{R}^{n_0 \times m}$ ,  $Y \in \mathbb{R}^{n_{L+1} \times m}$  is given as follows:

$$\mathcal{L}(W_{0:L}) = \frac{1}{2} \|Y - W_L W_{L-1} \dots W_0 X\|_F^2. \quad (2)$$

Assume  $L \geq 1$ . Assume  $YX^T \neq 0$ . Assume also that there are no bottlenecks in architecture, i.e.  $\min_l n_l = \min\{n_0, n_{L+1}\}$ . Prove that  $W_{0:L} = 0$  is a saddle point.