HOMEWORK 4

Varun Kaundinya
9082410649

Instructions: Use this latex file as a template to develop your homework. Submit your homework on time as a single zip (containing the pdf and the code) file to Canvas. Please, check Piazza for updates about the homework. It is recommended that you use Python for your solutions. You are allowed to use the libraries `numpy`, `matplotlib`, `pandas` and `pytorch`.

1 Questions (50 pts)

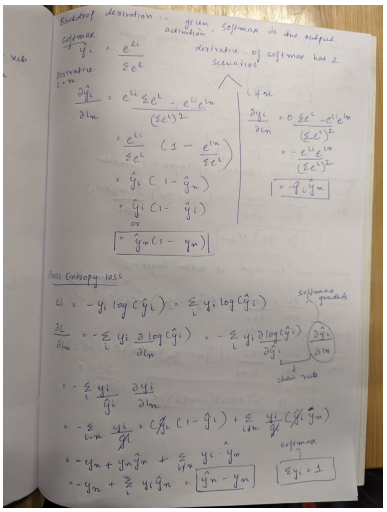
1. (10 Points) Suppose the world generates a single observation $x \sim$ multinomial($\boldsymbol{\theta}$), where the parameter vector $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_k)$ with $\theta_i \geq 0$ and $\sum_i^k \theta_i = 1$. Note $x \in \{1, \ldots, k\}$. You know θ and want to predict x. Call your prediction $\hat{x}$. What is your expected 0-1 loss:

$$\mathbb{E}[1\{\hat{x} \neq x\}]$$

using the following two prediction strategies respectively? Prove your answer.

   (a) (5 pts) Strategy 1: $\hat{x} \in \text{argmax}_x \theta_x$, the outcome with the highest probability.

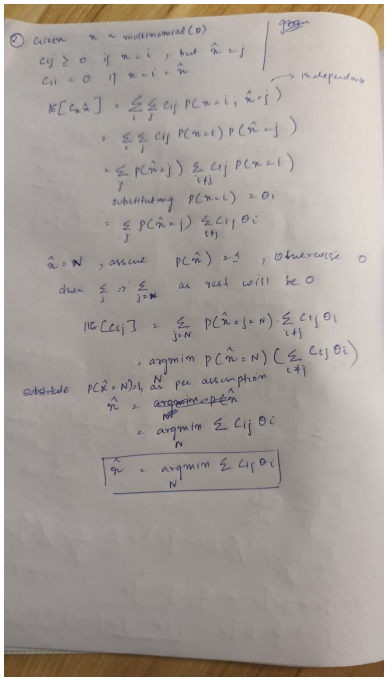
   (b) (5 pts) Strategy 2: You mimic the world by generating a prediction $\hat{x} \sim$ multinomial($\boldsymbol{\theta}$). (Hint: your randomness and the world's randomness are independent)



2. (10 points) Like in the previous question, the world generates a single observation $x \sim$ multinomial($\boldsymbol{\theta}$). Let $c_{ij} \geq 0$ denote the loss you incur, if $x = i$ but you predict $\hat{x} = j$, for $i, j \in \{1, \ldots, k\}$. $c_{ii} = 0$ for all i. This is a way to generalize different costs on false positives vs false negatives from binary classification to multi-class classification. You want to minimize your expected loss:

$$\mathbb{E}[c_{x\hat{x}}]$$

Derive your optimal prediction $\hat{x}$.



3. (30 Points)The Perceptron Convergence Theorem shows that the Perceptron algorithm will not make too many mistakes as long as every example is "far" from the separating hyperplane of the target halfspace. In this problem, you will explore a *variant* of the Perceptron algorithm and show that it performs well (given a little help in the form of a good initial hypothesis) as long as every example is "far" (in terms of angle) from the separating hyperplane of the *current hypothesis*.

   Consider the following variant of Perceptron:

   - Start with an initial hypothesis vector $\boldsymbol{w} = \boldsymbol{w}^{\text{init}}$.
   - Given example $\boldsymbol{x} \in \mathbb{R}^n$, predict according to the linear threshold function $\boldsymbol{w} \cdot \boldsymbol{x} \geq 0$.
   - Given the true label of $\boldsymbol{x}$, update the hypothesis vector $\boldsymbol{w}$ as follows:
     - If the prediction is correct, leave $\boldsymbol{w}$ unchanged.
     - If the prediction is incorrect, set $\boldsymbol{w} \leftarrow \boldsymbol{w} - (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$.

   So the update step differs from that of Perceptron shown in class in that $(\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$ (rather than $\boldsymbol{x}$) is added or subtracted to $\boldsymbol{w}$. (Note that if $\|\boldsymbol{x}\|_2 = 1$, then this update causes vector $\boldsymbol{w}$ to become orthogonal to $\boldsymbol{x}$, i.e., we add or subtract the multiple of x that shrinks $\boldsymbol{w}$ as much as possible.)

   Suppose that we run this algorithm on a sequence of examples that are labeled according to some linear threshold function $\boldsymbol{v} \cdot \boldsymbol{x} \geq 0$ for which $\|\boldsymbol{v}\|_2 = 1$. Suppose moreover that

   - Each example vector $\boldsymbol{x}$ has $\|\boldsymbol{x}\|_2 = 1$;
   - The initial hypothesis vector $\boldsymbol{w}^{\text{init}}$ satisfies $\|\boldsymbol{w}^{\text{init}}\|_2 = 1$ and $\boldsymbol{w}^{\text{init}} \cdot \boldsymbol{v} \geq \gamma$ for some fixed $\gamma > 0$;
   - Each example vector $\boldsymbol{x}$ satisfies $\frac{|\boldsymbol{w} \cdot \boldsymbol{x}|}{\|\boldsymbol{w}\|_2} \geq \delta$, where $\boldsymbol{w}$ is the current hypothesis vector when $\boldsymbol{x}$ is received. (Note that for a unit vector $\boldsymbol{x}$, this quantity $\frac{|\boldsymbol{w} \cdot \boldsymbol{x}|}{\|\boldsymbol{w}\|_2}$ is the cosine of the angle between vectors $\boldsymbol{w}$ and $\boldsymbol{x}$.)

   Show that under these assumptions, the algorithm described above will make at most $\frac{2}{\delta^2} \ln(1/\gamma)$ many mistakes.

Upper Bound :
Given the update step, $\boldsymbol{w} \leftarrow \boldsymbol{w} - (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$.
Taking a square on either side,
$\|\boldsymbol{w}'\|^2 = \|\boldsymbol{w}\|^2 - 2\|\boldsymbol{w}(wx)x\| + \|(wx)x\|^2$
On Simplifying, with $\|\boldsymbol{x}\|^2 = 1$

$\|\boldsymbol{w}'\|^2 = \|\boldsymbol{w}\|^2 - \|\boldsymbol{w}x\|^2$, Dividing both sides by $\|\boldsymbol{w}\|^2$ and substituting δ
$\frac{\|\boldsymbol{w}'\|^2}{\|\boldsymbol{w}\|^2} = 1 - \frac{\|\boldsymbol{w}x\|^2}{\|\boldsymbol{w}\|^2}$
$\frac{\|\boldsymbol{w}'\|^2}{\|\boldsymbol{w}\|^2} <= 1 - \delta^2$
After One mistake, $\|\boldsymbol{w}'\|^2 <= 1 - \delta^2$
After m mistakes, $\|\boldsymbol{w}\|_m <= \sqrt{(1-\delta^2)^m}$
Lower Bound :
$w^{init} = -(wx)x$
$w1 = w^{init} - (wx)x$
$w1 - w^{init} = -(wx)x$
After m mistakes,
$w_m - w_{init} = -\Sigma_1^m(wx)x$
Multiplying by v on either side,
$w_m.v - w_{init}.v = -\Sigma_1^m(wx)x.v$
$w_m.v = -\Sigma_1^m(wx)x.v + w_{init}.v$
Ignoring the negative term and substituting γ,
$w_m.v >= \gamma$, Given $|\boldsymbol{v}\| = 1$
$|\boldsymbol{w}_m\| >= \gamma$
Combining both the bounds
$\gamma <= \|\boldsymbol{w}\|_m <= \sqrt{(1-\delta^2)^m}$
$\gamma <= \sqrt{(1-\delta^2)^m}$
Taking Log (ln) on either sides
$\frac{m}{2} ln(1-\delta^2) >= ln(\gamma)$
By applying Taylor's expansion on $ln(1-\delta^2)$ we get $ln(1-\delta^2)$ approx equal to -δ^2
$-\frac{m}{2}\delta^2 >= ln(\gamma)$
$m <= \frac{2}{\delta^2} \ln(1/\gamma)$
Proved.

2 Programming (60 pts)

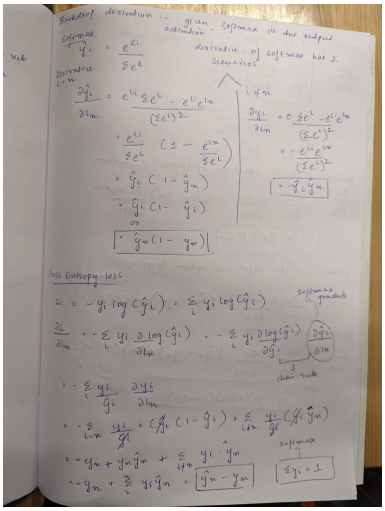
In this exercise, you will derive, implement back-propagation for a simple neural network, and compare your output with some standard library's output. Consider the following 3-layer neural network.

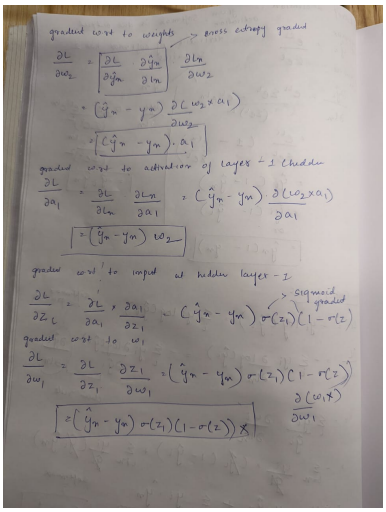
$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{W}_2\boldsymbol{\sigma}(\boldsymbol{W}_1\boldsymbol{x}))$$

Suppose $\boldsymbol{x} \in \mathbb{R}^d, \boldsymbol{W}_1 \in \mathbb{R}^{d_1 \times d}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{k \times d_1}$ i.e., $f : \mathbb{R}^d \mapsto \mathbb{R}^k$, Let $\boldsymbol{\sigma}(\boldsymbol{z}) = [\sigma(\boldsymbol{z}_1), \ldots, \sigma(\boldsymbol{z}_n)]$ for any $\boldsymbol{z} \in \mathbb{R}^n$ where $\sigma(t) = \frac{1}{1+\exp(-t)}$ is the sigmoid (logistic) activation function and $\boldsymbol{g}(\boldsymbol{z})_i = \frac{\exp(\boldsymbol{z}_i)}{\sum_{i=1}^k \exp(z_i)}$ is the softmax function. Suppose that the true pair is $(\boldsymbol{x}, \boldsymbol{y})$ where $\boldsymbol{y} \in \{0,1\}^k$ with exactly one of the entries equal to 1 and you are working with the cross-entropy loss function given below,

$$L(\boldsymbol{x}, \boldsymbol{y}) = -\sum_{i=1}^{k} \boldsymbol{y} \log(\hat{\boldsymbol{y}}) \ .$$

1. Derive backpropagation updates for the above neural network. (10 pts)





2. Implement it in `numpy` or `pytorch` using basic linear algebra operations. (e.g. You are not allowed to use auto-grad, built-in optimizer, model, etc. in this step. You can use library functions for data loading, processing, etc.). Evaluate your implementation on `MNIST` dataset, report test error, and learning curve. (25 pts)

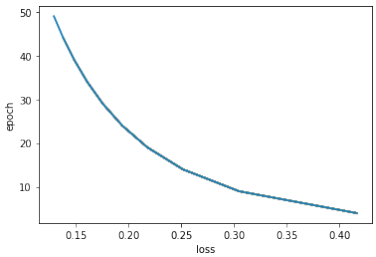


Figure 1: Batch Size = 32, Learning Rate = 0.05, Accuracy = 96.25%

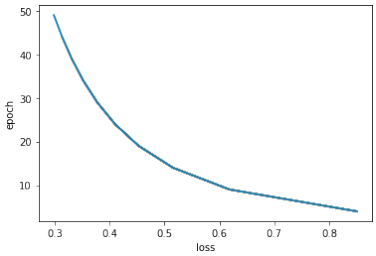


Figure 2: Batch Size = 32, Learning Rate = 0.01, Accuracy = 94.23%

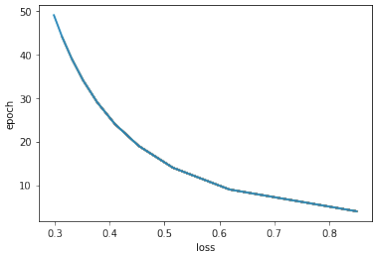


Figure 3: Batch Size = 64, Learning Rate = 0.01, Accuracy = 92.51%

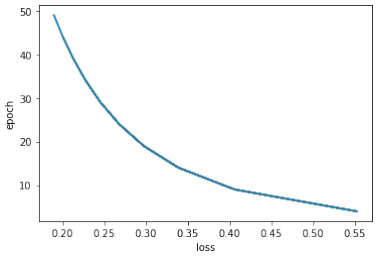


Figure 4: Batch Size = 64, Learning Rate = 0.05, Accuracy = 95.48%

3. Implement the same network in `pytorch` (or any other framework). You can use all the features of the framework e.g. auto-grad etc. Evaluate it on `MNIST` dataset, report test error, and learning curve. (20 pts)

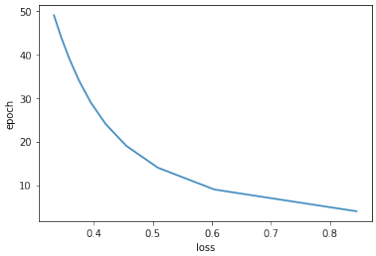


Figure 5: Batch Size = 32, Learning Rate = 0.01, Accuracy = 93.76%

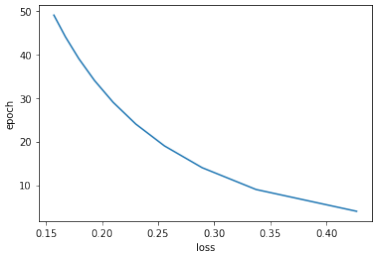


Figure 6: Batch Size = 32, Learning Rate = 0.01, Accuracy = 97.42%

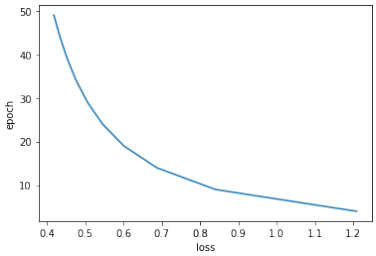


Figure 7: Batch Size = 64, Learning Rate = 0.01, Accuracy = 94.23%

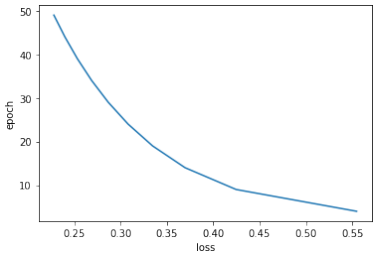


Figure 8: Batch Size = 64, Learning Rate = 0.05, Accuracy = 96.28%

4. Try different weight initializations a) all weights initialized to 0, and b) Initialize the weights randomly between -1 and 1. Report test error and learning curves for both. (You can use either of the implementations) (5 pts)

   We see that when all weights are initialized to zero the convergence is very slow, The loss barely decreases over epochs, to compare the models below are the graph run on numpy implementation.

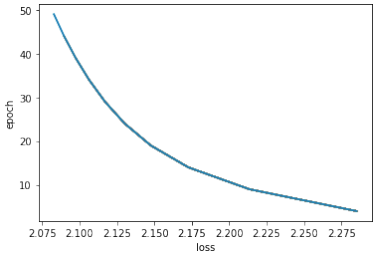


Figure 9: Zero Weights Batch Size = 32, Learning Rate = 0.05

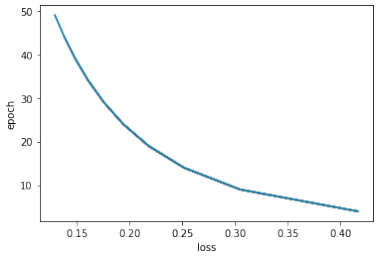


Figure 10: Batch Size = 32, Learning Rate = 0.05

You should play with different hyper-parameters like learning rate, batch size, etc. For Questions 2-3; you should report the answers for at least 3 different sets of hyperparameters. You should mention the values of those along with the results. Use $d_1 = 300, d_2 = 200$. For optimization use SGD (Stochastic gradient descent) without momentum, with some batch-size say 32, 64 etc. `MNIST` can be obtained from here (`https://pytorch.org/vision/stable/datasets.html`)
`Check Piazza for a sample PyTorch project`
To load the dataset:

```
import torch
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader
import torchvision

mnist_data_train = torchvision.datasets.MNIST('path/to/download',
    train=True,download=True)
data_loader = torch.utils.data.DataLoader(mnist_data,
                                          batch_size=,
                                          shuffle=True)
mnist_data_test = torchvision.datasets.MNIST('path/to/download',
    train=False,download=True)
```