HOMEWORK 4

MADHUSHREE NIJAGAL
9084490524

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.
As given in the problem statement, we know the value of θ and from the strategy we can imply that the value of θ is maximum. When θ is maximum, we have $P(\hat{x} = x)$ and the loss is 0 and when we have values of θ less than the maximum value we have $P(\hat{x} \neq x)$ and the loss is 1. That is when x is misclassified ($\hat{x} \neq x$), the loss is 1 and 0 when correctly classified ($\hat{x} = x$). Therefore we can write the expected loss function

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

as:

$$= \sum x * P(x) = 1 * P(\hat{x} \neq x) + 0 * P(\hat{x} = x)$$
$$= 1 * P(\hat{x} \neq x) = 1 - P(\hat{x} = x) = 1 - \theta_{max}$$

(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)
As mentioned above, the loss function can be written as a function of all the values that are misclassified, that is when loss is 1 and values that are correctly classified, that is when loss is 0. Here the values of $\hat{x}$ and x are independent. Therefore the expected loss function can be written as

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

$$= \sum x * P(x) = 1 * P(\hat{x} \neq x) + 0 * P(\hat{x} = x)$$
$$= 1 - [P(\hat{x} = 1, x = 1) + P(\hat{x} = 2, x = 2) + ... + P(\hat{x} = k, x = k)]$$
$$= 1 - [{\theta_1}^2 + {\theta_2}^2 + ... + {\theta_3}^2]$$
$$= 1 - \sum_{i=1}^k \theta_i^2$$

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 $\hat{x}$.
We are given that the x = i and $\hat{x}$ = j and associated cost is $c_{ij} >= 0$ and if x = $\hat{x} = i$, the associated cost is

$c_{ii} = 0$. We can represent the same information in the following matrix as:

$$\begin{bmatrix} 0 & c_{12} & c_{13} \\ c_{21} & 0 & c_{23} \\ c_{31} & c_{32} & 0 \end{bmatrix}$$

Therefore the loss function can be represented as $\mathbb{E}[c_{x\hat{x}}] = \sum_{j=1}^{k} \sum_{i=1}^{k} c_{ij}\theta_i\theta_j$.
The θ_i and θ_j are the probabilities of x and $\hat{x}$ respectively. To minimize the expected loss, we need to minimize the function by finding the smallest value of c_{ij} for every i value and vary j, the function is represented as:
$\hat{x} \in \text{argmin}_j(\sum_{i=1}^{k} \theta_i c_{ij})$

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.

So we are given that for every true value of x, if the prediction is incorrect we have $\boldsymbol{w}$ updated by $\boldsymbol{w} - (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$
The convergence upper bound proof:
Therefore for every mistake the update done can be represented as below: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - (\boldsymbol{w}_t \cdot \boldsymbol{x})\boldsymbol{x}$
Squaring both the sides we get:

$$\|\boldsymbol{w}_{t+1}\|^2 = \|\boldsymbol{w}_t - (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}\|^2$$
$$= \|\boldsymbol{w}_t - (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x})\|^T \|\boldsymbol{w}_t - (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}\|$$

Expanding the above we get:

$$\|\boldsymbol{w}_{t+1}\|^2 = \|\boldsymbol{w}_t\|^2 - 2\|\boldsymbol{w}_t(\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}\|^2 + \|(\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}\|^2$$
$$\|\boldsymbol{w}_{t+1}\|^2 = \|\boldsymbol{w}_t\|^2 - |\boldsymbol{w} \cdot \boldsymbol{x}| \text{ ; since we have } \|\boldsymbol{x}\|_2 = 1$$
$$\frac{|\boldsymbol{w}_{t+1}|^2}{\|\boldsymbol{w}_t\|^2} = 1 - \frac{|\boldsymbol{w}\cdot\boldsymbol{x}|^2}{\|\boldsymbol{w}\|^2} \leq 1 - \delta^2$$

After m_t mistakes the update to $\boldsymbol{w}$ changes to:

$$\|\boldsymbol{w}_{t+1}\|^2 \leq (1-\delta^2)^{m_t} \text{ -> Proof 1}$$

The convergence lower bound proof:
$\boldsymbol{w}^{init} = 1$
$\boldsymbol{w}_{t+1} = \boldsymbol{w}^{init} - (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$
$\boldsymbol{w}_{t+1} - \boldsymbol{w}^{init} = -(\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$
After m_t mistakes we have:

$$\boldsymbol{w}_{t+1} - \boldsymbol{w}^{init} = \sum_{i=0}^{m_t} -(\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$$
$$\boldsymbol{w}_{t+1} - \boldsymbol{w}^{init} = -\sum_{i=0}^{m_t} (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x}$$

Taking the dot product of $\boldsymbol{v}$ on both sides:

$$\boldsymbol{w}_{t+1} \cdot \boldsymbol{v} - \boldsymbol{w}^{init} \cdot \boldsymbol{v} = -\sum_{i=0}^{m_t} (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x} \cdot \boldsymbol{v}$$
$$\boldsymbol{w}_{t+1} \cdot \boldsymbol{v} = -\sum_{i=0}^{m_t} (\boldsymbol{w} \cdot \boldsymbol{x})\boldsymbol{x} \cdot \boldsymbol{v} + \boldsymbol{w}^{init} \cdot \boldsymbol{v}$$

We have $\boldsymbol{w}^{init} \cdot \boldsymbol{v} >= \gamma$ and $\|\boldsymbol{v}\|_2 = 1$

$$\|\boldsymbol{w}_{t+1}\| >= \gamma \text{—> Proof 2}$$

Combining both Proof 1 and Proof 2, we have

$$(1-\delta^2)^{m_t} >= \gamma^2$$

Taking ln on both sides:

$$m_t ln(1-\delta^2) >= 2ln\gamma$$
We know that $-ln(1-x) >= x$ and replacing it the above equation we get:
$$-m_t\delta^2 >= 2ln(\gamma)$$
$$-m_t >= 2ln(\gamma)/\delta^2$$
Simplifying, we get $m_t <= 2/\delta^2 * ln(1/\gamma)$

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)
   We are given cross entropy loss function is given by $L(\boldsymbol{x}, \boldsymbol{y}) = -\sum_{i=1}^{k} \boldsymbol{y} \log(\hat{\boldsymbol{y}})$.
   Some of the notations can be defined as:

$$z^{(1)} = W^{(1)}x$$
$$a^{(1)} = \sigma(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{(1)}$$
$$a^{(2)} = \hat{y} = softmax(z^{(2)})$$

To derive the back-propagation update, we need to calculate the partial derivative of the loss function with respect to the $W^{(2)}$
By chain rule we can express the partial derivative as:

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$
$$\frac{\partial z^{(2)}}{\partial W^{(2)}} = a^{(1)}$$

The softmax function is given by:

$$softmax(z^{(i)}) = \frac{\exp(\boldsymbol{z}_i)}{\sum_{i=1}^{k} \exp(z_i)}$$

The derivative of the softmax is expressed as below: (The notes in the lecture were referred to arrive at the derivative):

$$\frac{\partial softmax(z^{(i)})}{\partial z^{(i)}} = \frac{(\sum e^{(z_i)})(e^{(z_i)}) - (e^{(z_i)})^2}{(\sum e^{(z_i)})^2} = s(z_i)(1 - s(z_i))$$
$$\frac{\partial softmax(z^{(i)})}{\partial z^{(j)}} = \frac{0 - (e^{(z_i)})(e^{(z_j)})}{(\sum e^{(z_i)})^2} = -s(z_i)s(z_j)$$

The partial derivative of the L with respect to $z^{(2)}$

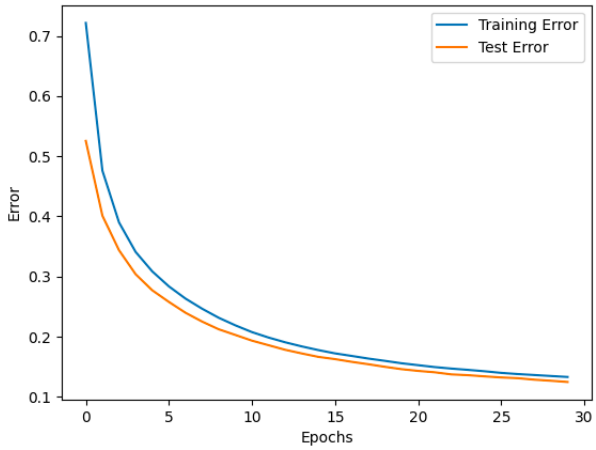
$$\frac{\partial L}{\partial z^{(2)}} = -\sum \frac{y^{(i)}}{\hat{y}^i} \frac{\partial \hat{y}^{(i)}}{\partial z^{(2)}}$$
$$-[\frac{y^{(i)}}{s(z_i)} s(z_i)(1 - s(z_i)) + \frac{\sum_{i \neq j} y^{(i)}}{s(z_i)} - s(z_i)s(z_j)]$$
$$= -(y^{(i)}(1 - s(z_i)) + \sum_{i \neq j} -y^{(i)} s(z_j))$$
$$= -y^{(i)} + y^{(i)} s(z_i) + \sum_{i \neq j} -y^{(i)} s(z_j)$$
$$= -y^{(i)} + \sum_{\forall i} -y^{(i)} s(z_i)$$
$$= -y^{(i)} + s(z_i) \sum_{\forall i} -y^{(i)}$$
$$= s(z_i) - y^{(i)}$$
$$=> \frac{\partial L}{\partial z^{(2)}} = a^{(2)} - y$$
$$=> \frac{\partial L}{\partial w^{(2)}} = (a^{(2)} - y)(a^{(1)})$$
$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z(2)}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$
$$\frac{\partial a^{(1)}}{\partial z^{(1)}} = \sigma(z^{(1)})(1 - \sigma(z^{(1)})) \; ; \; \frac{\partial z^{(2)}}{\partial a^{(1)}} = W^{(2)} \; ; \; \frac{\partial z^{(1)}}{\partial W^{(1)}} = x$$
$$=> \frac{\partial L}{\partial W^{(1)}} = (a^{(2)} - y)(a^{(1)})((\sigma(z^{(1)}))(1 - \sigma(z^{(1)})(x))$$

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)

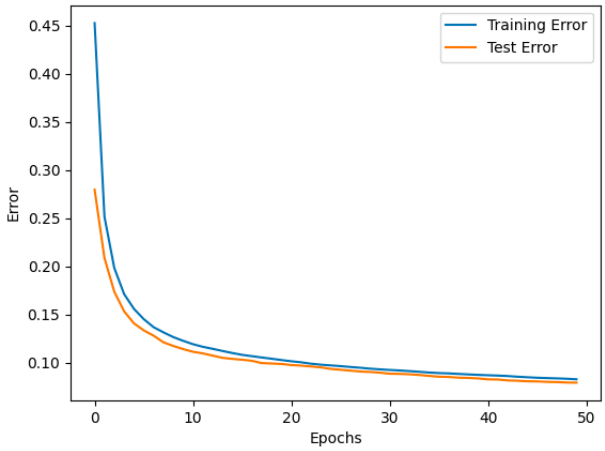
d	Learning Rate	Batch Size	Epoch	Test Error
300	0.01	128	30	0.19977666666666666
300	0.001	32	50	0.105526
300	0.005	64	100	0.107887

Table 1: Please find the respective graphs of each row below in order.

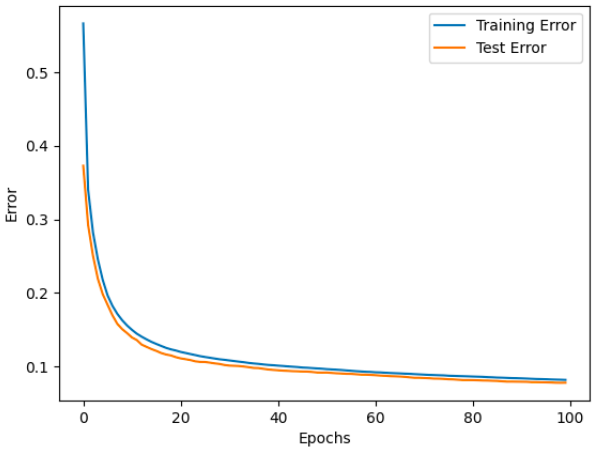
For d1 = 300:



lr = 0.01 batch-size = 128 epoch = 30



lr = 0.001 batch-size = 32 epoch = 50



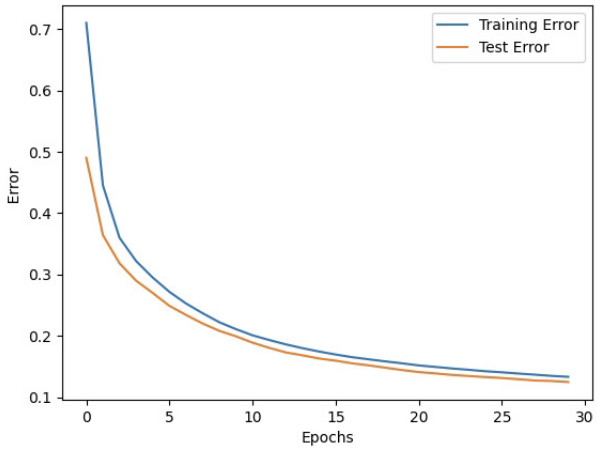
lr = 0.005 batch-size = 64 epoch = 100

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)

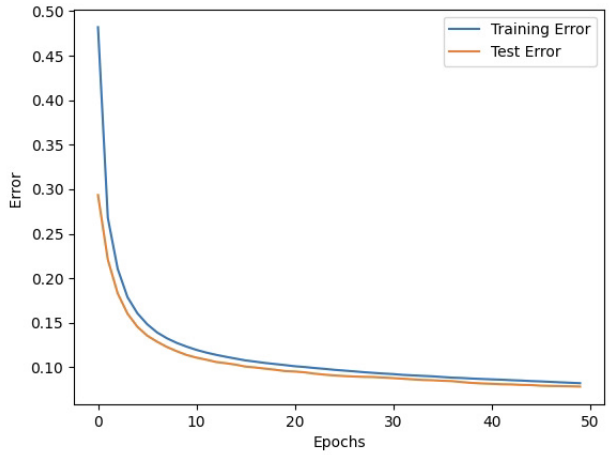
For d1 = 300:

d	Learning Rate	Batch Size	Epoch	Test Error
300	0.01	128	30	0.193653
300	0.001	32	50	0.10417
300	0.005	64	100	0.106197

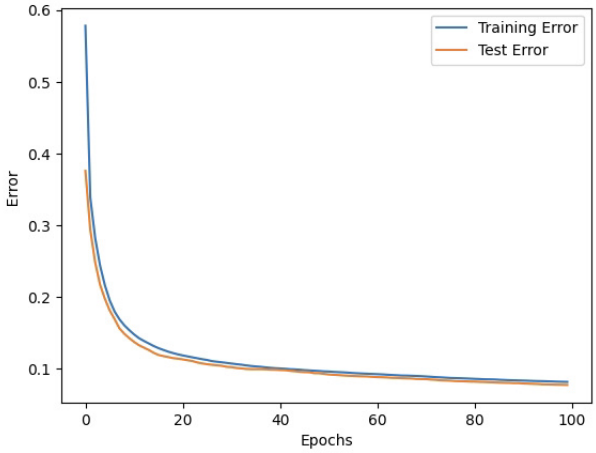
Table 2: Please find the respective graphs of each row below in order



lr = 0.01 batch-size = 128 epoch = 30

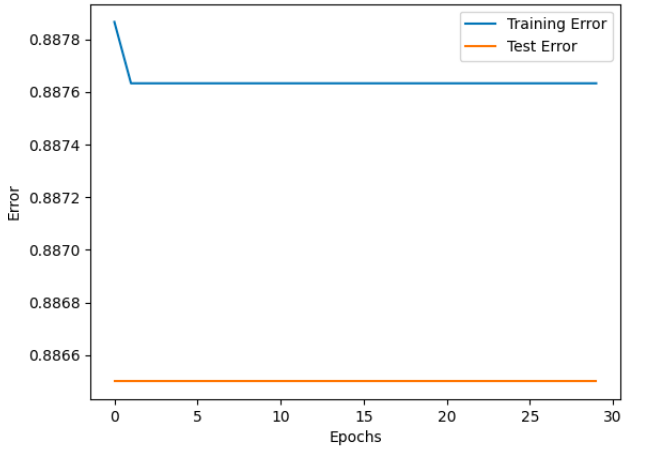


lr = 0.001 batch-size = 32 epoch = 50

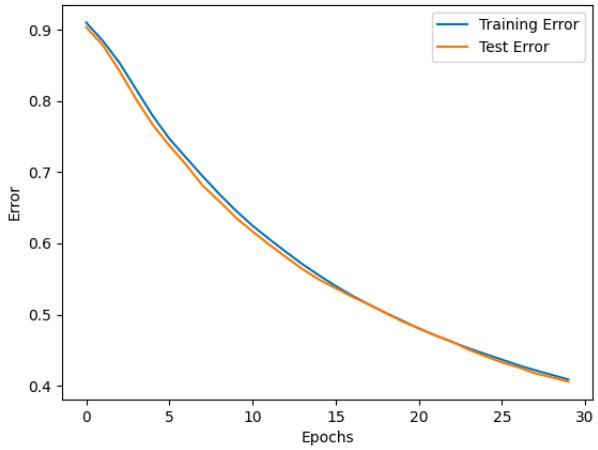


lr = 0.005 batch-size = 64 epoch = 100

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)
   a) Test error: 0.8865



   b) Test error: 0.5833



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)
```