HOMEWORK 4

Hafeez Ali Anees Ali
908 460 3423

1. Questions (50 pts)

(10 Points) Suppose the world generates a single observation x ~ multinomial(θ), where the parameter vector θ = (θ1, . . . , θk) with θi ≥ 0 and Pki θi = 1. Note x ∈ {1, . . . , k}. You know θ and want to predict x. Call your prediction ˆx. What is your expected 0-1 loss:
E[1{ˆx ̸= x}]
using the following two prediction strategies respectively? Prove your answer.
(a) (5 pts) Strategy 1: ˆx ∈ argmaxxθx, the outcome with the highest probability.
(b) (5 pts) Strategy 2: You mimic the world by generating a prediction ˆx ~ multinomial(θ). (Hint: your randomness and the world's randomness are independent)

1) $x \sim \text{multinomial}(\theta)$
$\theta = (\theta_1, \ldots \theta_k)$ with $\theta_i \geq 0$ & $\sum_i^k \theta_i = 1$
$x \in \{1, \ldots k\}$
$E[1\{\hat{x} \neq x\}] = ?$

a) Strategy 1: $\hat{x} = \text{argmax}_x \theta_x$

$$E1[\hat{x} \neq x] = \sum_{x=1}^{k} P(\hat{x} \neq x) = 1 - P(\hat{x} = x)$$
$$= 1 - \theta_{\hat{x}}$$

$$\boxed{E1[\hat{x} \neq x] = 1 - \theta_{\hat{x}}}$$

b) Strategy 2: $\hat{x} \sim \text{multinomial}(\theta)$

$$E1[\hat{x} \neq x] = \sum_{y=1}^{k} P(\hat{x} \neq y, \hat{x} = y)$$
$$= \sum_{y=1}^{k} P(\hat{x} \neq y)\, P(\hat{x} = y)$$

(Because it is given that the world's randomness & my randomness are independent of each other)

$$E1[\hat{x} \neq x] = \sum_{y=1}^{k} (1 - P(\hat{x} = y))\, P(\hat{x} = y)$$
$$= \sum_{y=1}^{k} (1 - \theta_y)\, \theta_y$$
$$= \sum_{y=1}^{k} \theta_y - \sum_{y=1}^{k} \theta_y^2 = \boxed{1 - \sum_{y=1}^{k} \theta_y^2}$$

2. (10 points) Like in the previous question, the world generates a single observation x ~ multinomial(θ). Let cij ≥ 0 denote the loss you incur, if x = i but you predict ˆx = j, for i, j ∈ {1, . . . , k}. cii = 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:

E[cxˆx]

Derive your optimal prediction ˆx.

2)

$x \sim \text{multinomial}(\theta)$

$c_{ij} \geq 0$ is the loss if $x = i$ but prediction $\hat{x} = j$ , $i.j \in \{1, \ldots k\}$

$c_{ii} = 0 \;\; \forall \, i$

$$\mathbb{E}[c_{x\hat{x}}] = \sum_{j=1}^{k} \sum_{i=1, i \neq j}^{k} c_{ij} \, \theta_i \, \hat{\theta}_j$$

We need $\hat{x}$ such that $\sum_{j=1}^{k} \sum_{i=1, i \neq j}^{k} c_{ij} \theta_i \hat{\theta}_j$ is minimum.

$$\hat{x} = \operatorname{argmin}_j \left( \sum_{j=1}^{k} \sum_{i=1}^{k} c_{ij} \theta_i \right)$$

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 w = winit.

• Given example x ∈ Rn, predict according to the linear threshold function w °§ x ≥ 0.

• Given the true label of x, update the hypothesis vector w as follows:

– If the prediction is correct, leave w unchanged.

– If the prediction is incorrect, set w ← w − (w °§ x)x.

So the update step differs from that of Perceptron shown in class in that (w°§x)x (rather than x) is added or subtracted to w. (Note that if // x // 2 = 1, then this update causes vector w to become orthogonal to x, i.e., we add or subtract the multiple of x that shrinks 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 v °§ x ≥ 0 for which // v // 2 = 1. Suppose moreover that

• Each example vector x has // x // 2 = 1; The initial hypothesis vector winit satisfies // winit // 2 = 1 and winit °§ v ≥ γ for some fixed γ > 0;

• Each example vector x satisfies |w°§x| // w // 2 ≥ δ, where w is the current hypothesis vector when x is received. (Note that for a unit vector x, this quantity |w°§x| // w // 2 is the cosine of the angle between vectors w and x.) Show that under these assumptions, the algorithm described above will make at most 2 δ2 ln(1/γ) many
mistakes.

3) Perceptron Convergence

It is given that the initial hypothesis vector $w^{init}$ satisfies $\|w^{init}\|_2 = 1$ & $w^{init} \cdot v \geq \gamma$ for some fixed $\gamma > 0$

$$w^{init} \cdot v \geq \gamma \quad \text{—— (1)}$$

Let's say a mistake occurred for some data point $x$, then the new hypothesis vector is given by

$$w_1 = w^{init} - (w^{init} \cdot x)\, x$$

$$w_1 \cdot v = w^{init} \cdot v - (w^{init} \cdot x)\, x \cdot v$$

According to the linear threshold function, in the case of an error $w \cdot x < 0$

$$\Rightarrow w^{init} \cdot x < 0$$

So $-(w^{init} \cdot x) > 0$

$$-(w^{init} \cdot x)\, x \cdot v > 0 \quad \text{—— (2)}$$

Using (1) & (2), we get:

$$w^{init} \cdot v - (w^{init} \cdot x)\, x \cdot v \geq \gamma$$

$$\Rightarrow w_1 \cdot v \geq \gamma$$

$$\Rightarrow w_t \cdot v \geq \gamma \quad \forall t$$

Since $w_1 = w^{init} - (w^{init} \cdot x)x$

$$\|w_1\|^2 = \|w^{init}\|^2 - 2\|w^{init}\|^2\|x\|^2|\cos\theta| + \|w^{init}\|^2\cos^2\theta\,\|x\|^4$$

Sine $\|x\| = 1$ is given for all $x$

& taking $\theta$ to be angle between $w^{init}$ & $x$

we have

$$\|w_1\|^2 = \|w^{init}\|^2\left[1 - 2|\cos\theta| + \cos^2\theta\right] \quad -(3)$$

It is also given that $x$ satisfies $\dfrac{|w.x|}{\|w\|_2} \geq \delta$

But $\dfrac{|w.x|}{\|w\|_2} = |\cos\theta| \Rightarrow |\cos\theta| \geq \delta$

$$-|\cos\theta| \leq -\delta$$

$$1 - |\cos\theta| \leq 1 - \delta \quad -(4)$$

Continuing with (3) we have,

$$\|w_1\|^2 = \|w^{init}\|^2(1 - \cos\theta)^2$$

But $\|w^{init}\|_2 = 1$ is also given

So, $\|w_1\|^2 = (1 - \cos\theta)^2$

$$\|w_1\| = (1 - |\cos\theta|)$$

From (4), we have $\|w_1\| = (1 - |\cos\theta|) \leq 1 - \delta$ $\quad -(5)$

Extending this, we have $\|w_t\| \leq (1-\delta)^t$ for $t$ mistakes.

The geometric interpretation of a misclassification update is it shrinks the hypothesis vector after every update.

$$\Rightarrow \quad 1 \geq (1-\delta) \geq 0$$

$$(1-\delta) \leq 1$$
$$0 \geq -\delta$$
$$\delta \geq 0$$

Also, $\delta \leq |\cos\theta| \leq 1$

$$\Rightarrow \quad 0 \leq \delta \leq 1$$

Additionally, $w_t \cdot v \geq \gamma \Rightarrow w_t \cdot v \geq 0$ since $\gamma > 0$
$$\Rightarrow |w_t \cdot v| \geq \gamma$$

$$\Rightarrow \|w_t\| \, \|v\| \geq |w_t \cdot v| \geq \gamma$$

Now, $\|v\| = 1$

$$\Rightarrow \|w_t\| \geq \gamma \quad \text{— (6)}$$

From (5) & (6) $\quad \gamma \leq \|w_t\| \leq (1-\delta)^t \quad$ — (7)

Since $0 \leq \delta \leq 1$, we have $\delta \geq \delta^2$
$$\Rightarrow 1-\delta \leq 1-\delta^2$$

So (7) can be rewritten as $\gamma \leq \|w_t\| \leq (1-\delta^2)^t$

For $0 \le \delta \le 1$,

$-1 \le -\delta \le 0$

$0 \le 1-\delta \le 1$

$\Rightarrow (1-\delta)^2 \le (1-\delta)$

but $(1-\delta) \le 1-\delta^2$

$\Rightarrow (1-\delta)^2 \le 1-\delta^2$

From (7), $\gamma \le \|w_t\| \le (1-\delta)^{2(t/2)} \le (1-\delta^2)^{t/2}$

$\Rightarrow \gamma \le \|w_t\| \le (1-\delta^2)^{t/2}$

$\Rightarrow \gamma^2 \le (1-\delta^2)^t \quad (\gamma > 0 \text{ \& } (1-\delta^2)^{t/2} > 0)$

$\ln \gamma^2 \le \ln (1-\delta^2)^t$ (applying ln on both sides)

$2 \ln \gamma \le t \ln(1-\delta^2)$ — (8)

Now, $e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} \ldots$

So $e^{-\delta^2} \geq 1-\delta^2$

$\ln e^{-\delta^2} \geq \ln(1-\delta^2)$

From (8) $\Rightarrow 2 \ln \gamma \le t \ln(1-\delta^2) \le t \ln e^{-\delta^2}$

$2 \ln \gamma \le t \ln e^{-\delta^2}$

$2 \ln \gamma \le -\delta^2 t$

$-2 \ln \gamma \geq \delta^2 t$

$\boxed{t \le \frac{2}{\delta^2} \ln\left(\frac{1}{\gamma}\right)}$

$\Rightarrow$ number of mistakes $t$ is bounded by this inequality

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. y = f(x) = g(W2σ(W1x))
Suppose x ∈ Rd,W1 ∈ Rd1°ød and W2 ∈ Rk°ød1 i.e., f : Rd 7→ Rk, Let σ(z) = [σ(z1), . . . , σ(zn)] for any z ∈ Rn where σ(t) = 1 / 1+exp(−t) is the sigmoid (logistic) activation function and g(z)i = Pexp(zi) k i=1 exp(zi) is the softmax function. Suppose that the true pair is (x, y) where y ∈ {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(x, y) = −Xki=1y log(ˆy) .

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

2] Programming:

1) $$y = f(x) = g(W_2\, \sigma(W_1 x))$$

$x \in \mathbb{R}^d$, $W_1 \in \mathbb{R}^{d_1 \times d}$, $W_2 \in \mathbb{R}^{k \times d_1}$

$$f : \mathbb{R}^d \to \mathbb{R}^k$$

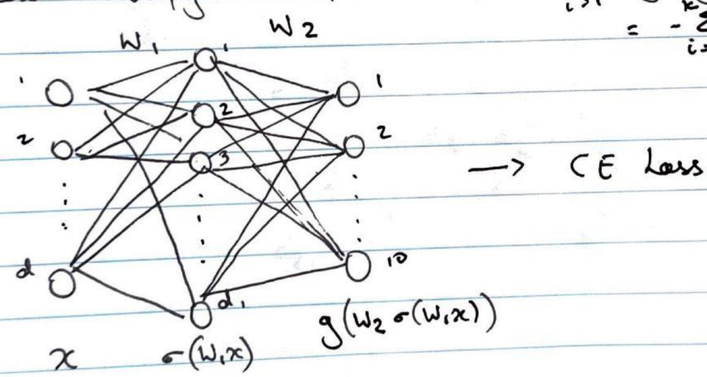
$\sigma(z) = [\sigma(z_1), \ldots, \sigma(z_n)]$ where $z \in \mathbb{R}^n$

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

$$g(z)_i = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_i}}$$

Cross entropy loss, $L(x, y) = -\sum_{i=1}^{k} y \log(\hat{y})$ $\quad y \in \{0, 1\}^k$

$$= -\sum_{i=1}^{k} y \log(g(z)_i)$$




* Derivative of softmax function:

Let $p(z)_i = e^{z_i}$

$$q(z)_i = \sum_{c=1}^{k} e^{z_c}$$

$$\frac{d\, g(z)_i}{dz_j} = \frac{d}{dz_j}\left(\frac{e^{z_i}}{\sum_{c=1}^{k} e^{z_c}}\right) = \frac{d}{dz_j}\left(\frac{p(z)_i}{q(z)_i}\right)$$

$$p'(z)_i = \frac{d\,(p(z)_i)}{dz_j} = \frac{d}{dz_j}(e^{z_i}) = \frac{d}{dz_i} e^{z_i} \frac{dz_i}{dz_j}$$

$$= e^{z_i} \frac{dz_i}{dz_j} = \begin{cases} e^{z_i} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

$$g'(x) = \frac{d}{dz_j}\left(\sum_{c=1}^{k} e^{z_c}\right) = \frac{d}{dz_j}\left(\sum_{c=1, c\neq j}^{k} e^{z_c} + e^{z_j}\right)$$

$$= \frac{d}{dz_j}\left(\sum_{c=1, c\neq j}^{k} e^{z_c}\right) + \frac{d}{dz_j}\left(e^{z_j}\right)$$

$$= 0 + e^{z_j}$$

$$= e^{z_j}$$

So, $$\frac{d\,g(z)_i}{dz_j} = \frac{e^{z_i} \sum_{c=1}^{k} e^{z_c} - e^{z_j} e^{z_i}}{\left(\sum_{c=1}^{k} e^{z_c}\right)^2} \quad \text{(when } i = j\text{)}$$

$$= \frac{e^{z_i}\left(\sum_{c=1}^{k} e^{z_c} - e^{z_j}\right)}{\left(\sum_{c=1}^{k} e^{z_c}\right)^2}$$

$$= \frac{e^{z_i}}{\sum_{c=1}^{k} e^{z_c}} \cdot \frac{\sum_{c=1}^{k} e^{z_c} - e^{z_j}}{\sum_{c=1}^{k} e^{z_c}}$$

$$= g(z)_i\left(1 - g(z)_j\right)$$

$$= \boxed{g(z)_i\left(1 - g(z)_i\right)} \quad (i = j)$$

$$\frac{d\,g(z)_i}{dz_j} = \frac{0 \times \sum_{c=1}^{k} e^{z_c} - e^{z_j} e^{z_i}}{\left(\sum_{c=1}^{k} e^{z_c}\right)^2} \quad \text{(when } i \neq j\text{)}$$

$$= \frac{-e^{z_j} e^{z_i}}{\left(\sum_{c=1}^{k} e^{z_c}\right)^2} = \boxed{-g(z)_i\, g(z)_j}$$

$(i \neq j)$

Derivation of cross-entropy loss with softmax function:

$$L = -\sum_{c=1}^{k} y_c \log\left(g(z)_c\right)$$

$$\frac{dL}{dz_i} = \frac{d}{dz_i}\left[-\sum_{c=1}^{k} y_c \log(g(z)_c)\right]$$

$$= -\sum_{c=1}^{k} y_c \frac{d(\log(g(z)_c))}{dz_i}$$

$$= -\sum_{c=1}^{k} y_c \frac{d\left(\log(g(z)_c)\right)}{d\, g(z)_c} \cdot \frac{d\, g(z)_c}{dz_i}$$

$$= -\sum_{c=1}^{k} \frac{y_c}{g(z)_c} \frac{d\, g(z)_c}{dz_i}$$

$$= -\left[\frac{y_i}{g(z)_i} \cdot \frac{d\, g(z)_i}{dz_i} + \sum_{c=1,\, c\neq i}^{k} \frac{y_c}{g(z)_c} \frac{d\, g(z)_c}{dz_i}\right]$$

$$= \frac{-y_i}{\cancel{g(z)_i}} \cdot \cancel{g(z)_i}(1-g(z)_i) - \sum_{c=1,\, c\neq i}^{k} \frac{y_c}{\cancel{g(z)_c}} \cdot \left(\cancel{g(z)_c}\, g(z)_i\right)$$

$$= -y_i + y_i\, g(z)_i + \sum_{c=1,\, c\neq i}^{k} y_c\, g(z)_i$$

$$= g(z)_i \left(y_i + \sum_{c=1,\, c\neq i}^{k} y_c\right) - y_i$$

$$= g(z)_i \left(\sum_{c=1}^{k} y_c\right) - y_i$$

$$= g(z)_i \cdot 1 - y_i \qquad \left(\text{because } \sum_{c=1}^{k} y_c = 1\right)$$

$$= \boxed{g(z)_i - y_i}$$

<u>Backpropagation</u> :

Using matrix notation, we can write

$$\frac{dL}{dZ} = \hat{Y} - Y$$

$$\frac{dL}{dW^2} = \frac{dL}{dZ^2}\frac{dZ^2}{dW^2}$$

$$= (\hat{Y} - Y)\frac{d}{dW^2}\left(W^2(A^1)\right)$$

$$= \boxed{(\hat{Y} - Y)\, A^1}$$

$A^1$ = output of first layer

$$\frac{dL}{dW^1} = \frac{dL}{dZ^2}\frac{dZ^2}{dA^1}\frac{dA^1}{dZ^1}\frac{dZ^1}{dW^1}$$

$$= (\hat{Y} - Y)\frac{d}{dA^1}(A^1 W^2) * \frac{d}{dZ^1}(\sigma(Z^1))$$

$$\frac{d}{dW^1}(W^1 X)$$

$$= (\hat{Y} - Y)\, W^2 \sigma'(Z^1)\, X$$

$$\frac{dL}{dW^1} = \boxed{(\hat{Y} - Y)\, W^2 \sigma'(Z^1)\, X}$$

$$\frac{dL}{dW^1} = \boxed{(\hat{Y} - Y)\, W^2 \sigma'(W^1 X)\, X}$$

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)

I.

```
# batch size == 60000 (full dataset)
# and learning rate = 0.1
# and epochs = 1000
```

```
predict(W1, W2, X_test, Y_test)
```

```
89.99000000000001
```

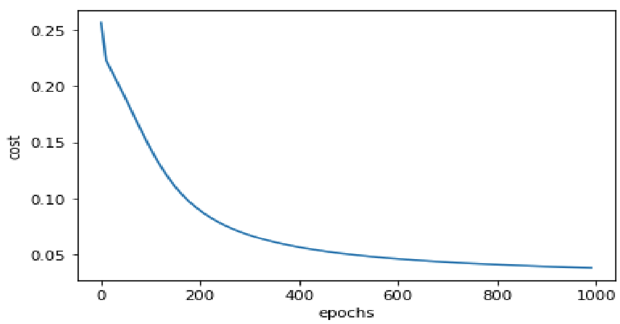
```
test_error = 100 - predict(W1, W2, X_test, Y_test)
```

```
test_error
```

```
10.009999999999991
```

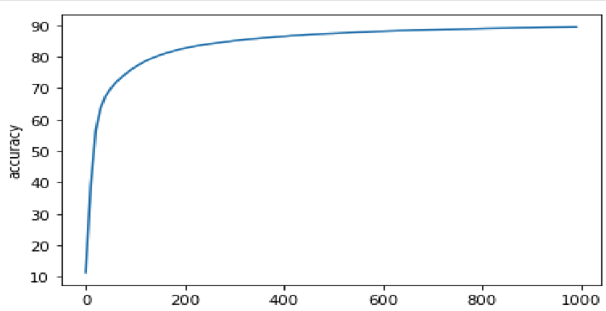
```
def plot_cost():
    plt.figure()
    plt.plot(epochs, costs)
    plt.xlabel("epochs")
    plt.ylabel("cost")
    plt.show()
```

```
plot_cost()
```




```
def plot_accuracy():
    plt.figure()
    plt.plot(epochs, accuracies)
    plt.xlabel("epochs")
    plt.ylabel("accuracy")
    plt.show()
```

```
plot_accuracy()
```



II.

```
predict(W1, W2, X_train, Y_train)
```

```
87.63
```

```
predict(W1, W2, X_test, Y_test)
```

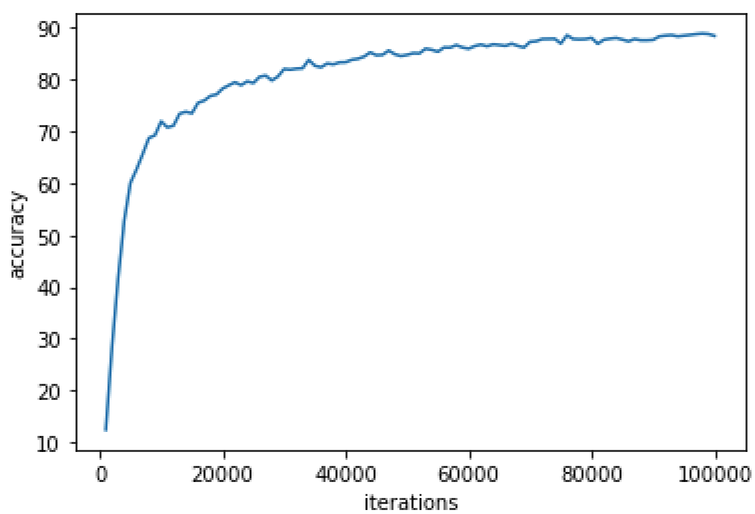
```
88.29
```

```
# batch_size = 64
# n_iterations = 100000
# learning rate = 0.5
```

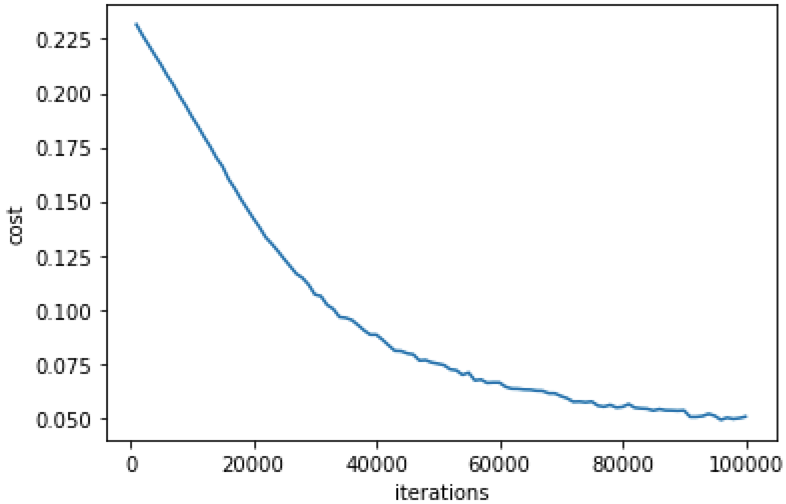
```
test_error = 100 - predict(W1, W2, X_test, Y_test)
test_error
```

```
11.709999999999994
```

```
plot_avg_accuracy()
```



```
plot_avg_cost()
```




III.

```
# batch_size = 32
# n_iterations = 500000
# learning rate = 0.3
```

```
predict(W1, W2, X_train, Y_train)
```

```
84.40666666666667
```

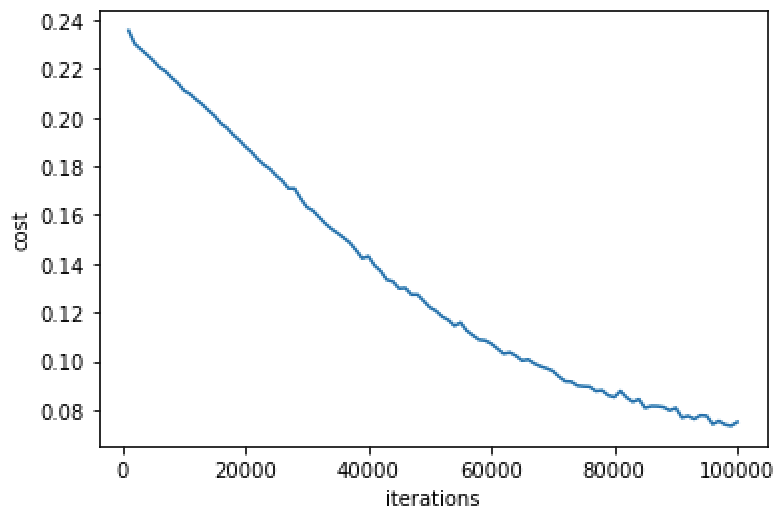
```
predict(W1, W2, X_test, Y_test)
```

```
85.35000000000001
```

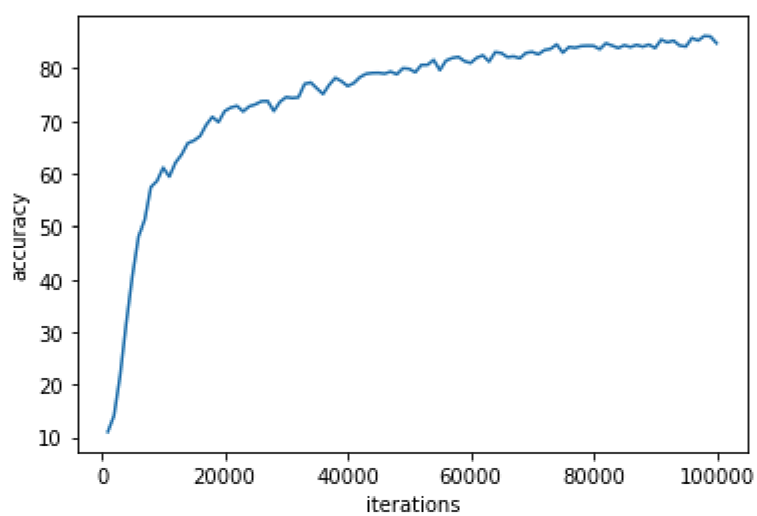
```
test_error = 100 - predict(W1, W2, X_test, Y_test)
test_error
```

```
14.649999999999991
```

```
plot_avg_cost()
```




```
plot_avg_accuracy()
```




IV.

```
# batch_size = 128
# n_iterations = 200000
# learning rate = 0.7
```

```
predict(W1, W2, X_train, Y_train)
```

```
91.58166666666666
```

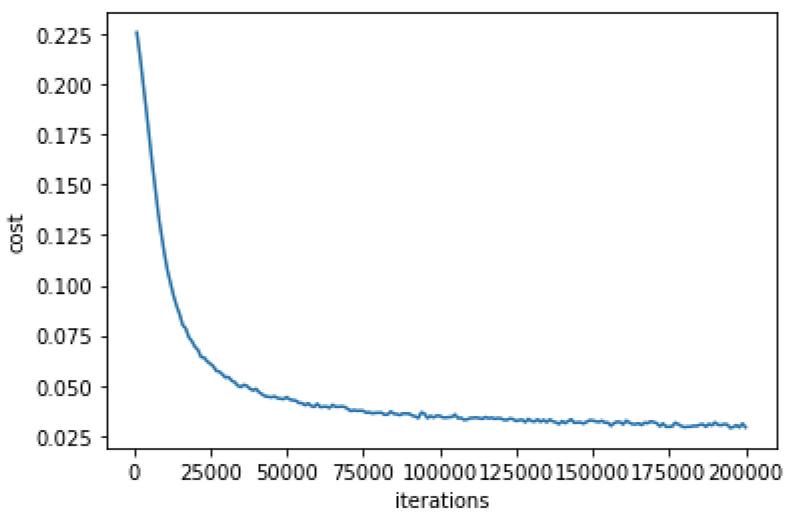
```
predict(W1, W2, X_test, Y_test)
```

```
91.79
```

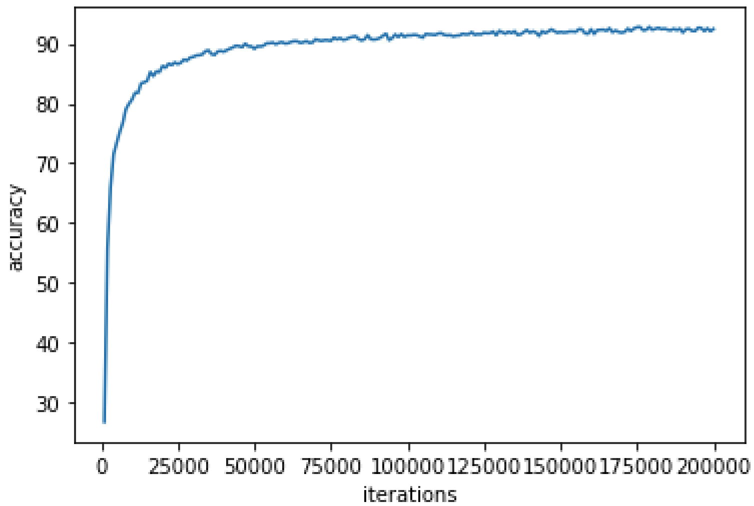
```
test_error = 100 - predict(W1, W2, X_test, Y_test)
test_error
```

```
8.209999999999994
```

```
plot_avg_cost()
```

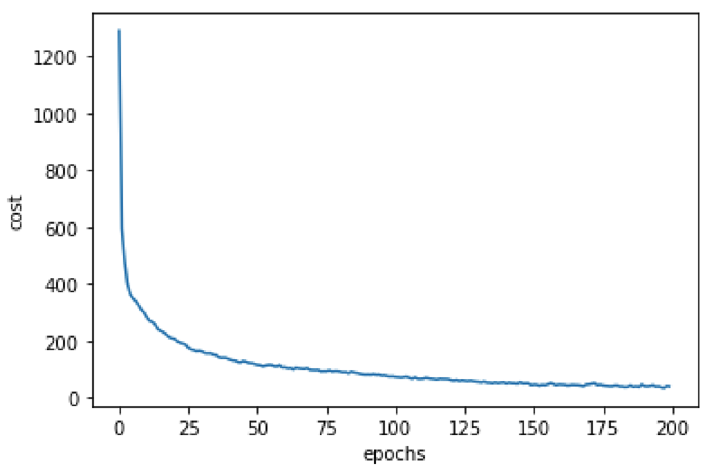


```
plot_avg_accuracy()
```




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)

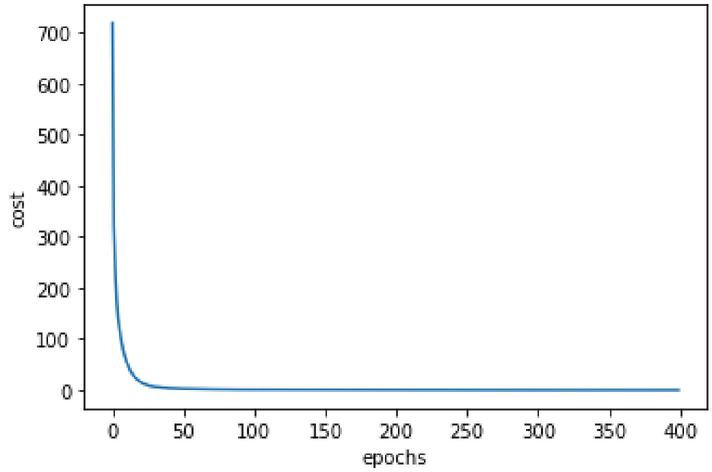
I.




```
# Batch size = 64
# epochs = 200
# lr = 10

Starting testing...
Test Accuracy: 95.09 %
Test Error: 4.91 %
```

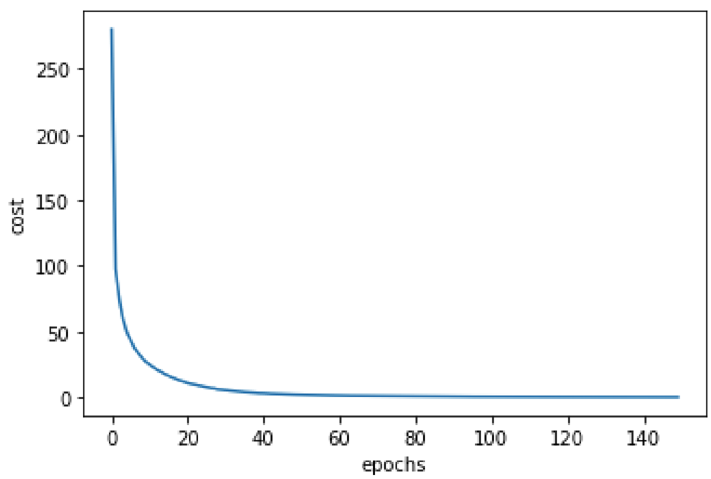
II.




```
]: # Batch size = 32
   # epochs = 400
   # lr = 0.5

   Starting testing...
   Train Accuracy: 98.17 %
   Test Error: 1.83 %
```

III.




```
: # Batch size = 128
  # epochs = 150
  # lr = 1

  Starting testing...
  Test Accuracy: 98.15 %
  Test Error: 1.85 %
```

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)

On Numpy:

a. All weights initialized to 0

```
# batch_size = 128
# n_iterations = 200000
# learning rate = 0.7
```

```
predict(W1, W2, X_train, Y_train)
```

```
11.236666666666666
```

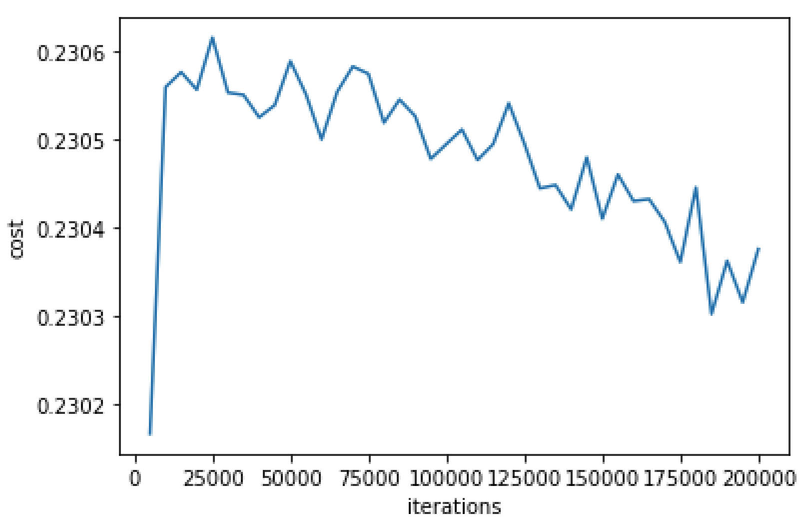
```
predict(W1, W2, X_test, Y_test)
```

```
11.35
```

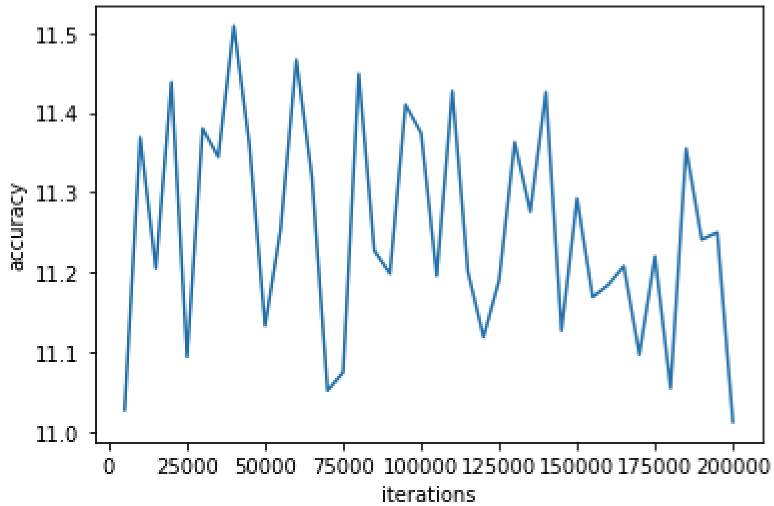
```
test_error = 100 - predict(W1, W2, X_test, Y_test)
test_error
```

```
88.65
```

```
plot_avg_cost()
```



```
plot_avg_accuracy()
```




b. Weights initialized randomly between -1 and 1

```
# batch_size = 128
# n_iterations = 200000
# learning rate = 0.7
```

```
predict(W1, W2, X_train, Y_train)
```

```
92.015
```

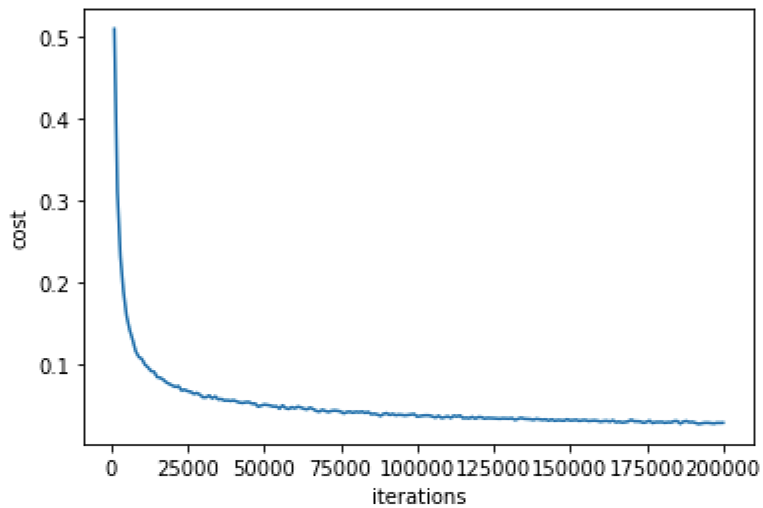
```
predict(W1, W2, X_test, Y_test)
```

```
91.61
```

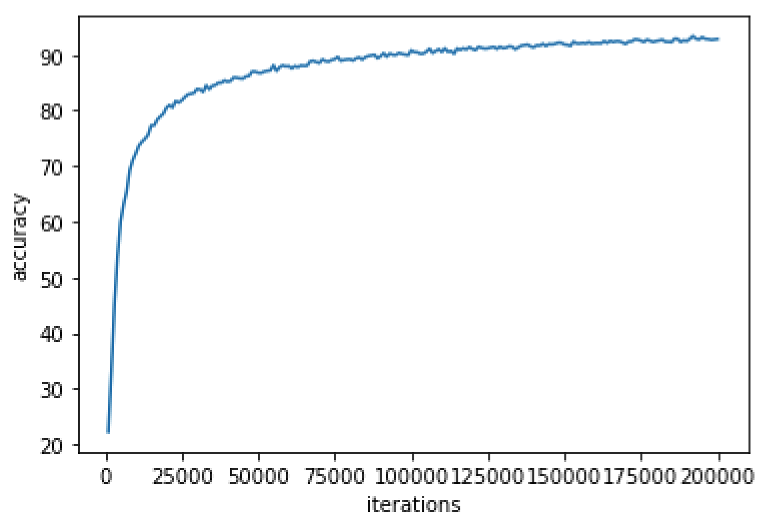
```
test_error = 100 - predict(W1, W2, X_test, Y_test)
test_error
```

```
8.39
```

```
plot_avg_cost()
```




```
plot_avg_accuracy()
```



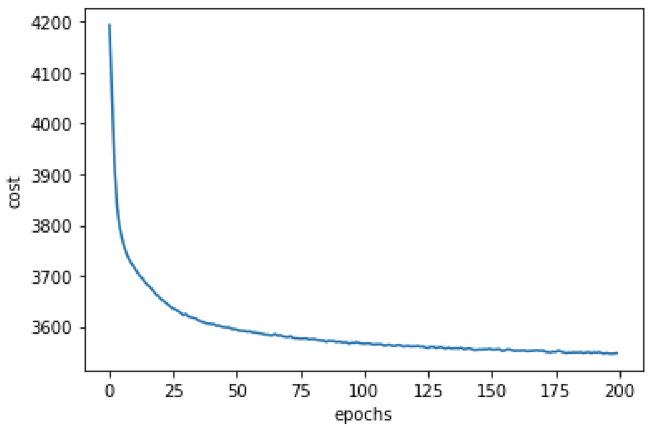

On Pytorch:

a. All weights initialized to 0

```
# Batch size = 32
# epochs = 200
# lr = 0.1

Starting testing...
Train Accuracy: 11.35 %
Test Error: 88.65 %
```

```
plot_cost()
```




b. Weights initialized randomly between -1 and 1

```
# Batch size = 64
# epochs = 200
# lr = 10

Starting testing...
Test Accuracy: 96.81 %
Test Error: 3.19 %
```

```
plot_cost()
```