Andrew
Curtis

Homework 4 CS 6685

1] Problem 1

incorrect form $C = -[a \ln y + (1-a)\ln(1-y)]$

when $y = 0$ $\quad C = -[a(\text{undefined}) + (1-a)(0)]$

cost function is undefined when $y=0$

when $y=1$ $\quad C = -[a(0) + (1-a)(\text{undefined})]$

cost function is also undefined when $y=1$

This does not affect the correct form because a will lie $0<a<1$ because of the sigmoid activation. Also $y=1$ will yield a low cost if the output, a, is near 1. this is what we want for learning

2]

For the bernoulli distribution of label y given inputs x,

$$P_{model}(y|\hat{y}) = P_{model}(y|P(y=1|x,\theta))$$

Using the negative log likelihood, $L(\theta) = -E_{x,y \sim P_{data}} \log P_{model}(y|x,\theta)$

→ plug in bernoulli dist $L(\theta) = -E \log(y(\theta) + (1-y)(1-\theta))$
whose cdf can be written as, $y(\theta) + (1-y)(1-\theta)$

→ distribute log $L(\theta) = -E[y\ln(\theta) + (1-y)\ln(1-\theta)]$

→ expected value is sum over inputs $C = L(\theta) = -\frac{1}{n}\sum_x [y\ln\theta + (1-y)\ln(1-\theta$

$$C = L(\theta) = -\frac{1}{n}\sum_x [y\ln\theta + (1-y)\ln(1-\theta)]$$

3] the support for a multinoulli distribution is $k \in \{1, 2, \ldots, K\}$, so it matches the probability of ending up in an output class.

So the multinoulli is like the sum over K classes of the bernoulli

→ plug in multinoulli into eqn 1

$$C = L(\theta) = -E_{x,y \sim \hat{p}data}\left[\log \sum_k y_k(\theta) + (1-y_k)(1-\theta)\right]$$

→ distribute log $\quad L(\theta) = -E\left[\sum_k y_k \ln\theta + (1-y_k)\ln(1-\theta)\right]$

→ take expected value $\quad L(\theta) = -\frac{1}{n}\sum_x \sum_k \left[y_k \ln\theta + (1-y_k)\ln(1-\theta)\right]$

$$C = L(\theta) = -\frac{1}{n}\sum_x \sum_k \left[y_k \ln\theta + (1-y_k)\ln(1-\theta)\right]$$

4] in order to show that C is still minimzed for $\sigma(z)=y$ we need to find C'. $\quad \sigma(z) = a$

Starting with $\quad C(a) = -(y \ln a + (1-y)\ln(1-a))$

→ take derivative $\quad C'(a) = -\frac{y}{a} + \frac{1-y}{1-a}$

→ solve for $C(a)=0 \quad -\frac{y}{a} + \frac{1-y}{1-a} = 0 \quad \frac{1-y}{1-a} = \frac{y}{a} \quad a - ay = y - ay$

$a = y$ extremum at this value

→ take second derivative $\quad C''(a) = \frac{y}{a^2} + \frac{1-y}{(1-a)^2}$

for $0 < y < 1$ and $0 < a < 1$ $C''(a)$ is greater than 0. meaning we had a minimzed function at $a = y = \sigma(z)$

5] Following the suggestion in the nielsen book, considering $a_j^L = \frac{e^{cz_j^L}}{\sum_k e^{cz_k^L}}$ what is the limit as $c \to \infty$?

rewrite function as $a_j^L = \frac{1}{1 + e^{-cz_j^L} \sum_{k \neq j} e^{cz_j^L}} = \frac{1}{1 + \sum_{k \neq j} e^{c(z_k^L - z_j^L)}}$

remember some input will be a maximum weight

if there exists an m, such that $z_m^L > z_j^L$ then z_j^L is not a maximal weight input

So $\lim_{c \to \infty} e^{c(z_m^L - z_j^L)} = \infty$ and $\lim_{c \to \infty} \sum_{k \neq j} e^{c(z_k^L - z_j^L)} = \infty$

it follows from our rewritten eqn, $\lim_{c \to \infty} a_j^L = 0$

So if z_j^L is a maximal input $\lim_{c \to \infty} e^{c(z_k^L - z_j^L)} = 1$

and if z_j^L is not maximal the limit is equal to 0

again for the denominator $\lim_{c \to \infty} \sum_{k \neq j} e^{c(z_k^L - z_j^L)} = n - 1$

$\lim_{c \to \infty} a_j^L = \frac{1}{n}$

this means that if c is greater than 1 it will put more weight on the bigger maximal values. but we want to account for all of the inputs, not just the maximal, hence using 1. this is the "soft" max idea for the name.

6| Show $\delta_j^L = a_j^L - y_j$ is suitable.

Starting with $\delta_j^L = \frac{\partial C}{\partial z_j^L}$

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \quad \text{Saying } y\tilde{y} = 1 \text{ for the output}$$

$$= \frac{\partial C}{\partial a_{\tilde{y}}^L} \frac{\partial a_{\tilde{y}}^L}{\partial z_j^L} \xrightarrow{\text{since } C = -\ln a_{\tilde{y}}^L \text{ for } \tilde{y}=1} = -\frac{1}{a_{\tilde{y}}^L} \frac{\partial a_{\tilde{y}}^L}{\partial z_j^L}$$

- if ouput is correctly identified, $j = \tilde{y}$

$$\delta_j^L = -\frac{1}{a_j^L}\left(e^{-z_j^L} \sum_{k \neq j} e^{z_k^L}\right)(a_j^L)^2 = -a_j^L\left(1 + e^{-z_j^L} \sum_{k \neq j} e^{z_j^L} - 1\right)$$

$$= -a_j^L\left(\frac{1}{a_j^L} - 1\right) = a_j^L - 1 = a_j^L - y_j \quad \text{← plugging in } \tilde{y} = 1$$

- if output is not correctly identified, $j \neq \tilde{y}$

$$\delta_j^L = -\frac{1}{a_{\tilde{y}}^L}\left(-e^{-z_{\tilde{y}}^L} e^{z_j^L}\right)(a_{\tilde{y}}^L)^2 = a_{\tilde{y}}^L e^{-z_{\tilde{y}}^L} e^{z_j^L}$$

plugging in a_j^L → $= \frac{e^{z_{\tilde{y}}^L}}{\sum_k e^{z_k^L}} e^{-z_{\tilde{y}}^L} e^{z_j^L} = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} = a_j^L = a_j^L - y_j$

Since $y_j = 0$ for misclassification.

this shows that $\delta_j^L = a_j^L - y_j$

Performs as expected for the given output

Problem 2

1] for the regularization parameter λ, making it large helps to have small weights but may slow down learning in gradient descent too much. if λ is small, the original cost function will be minimized more but accuracy might suffer and more overfitting could occur.

for the learning rate η, a larger learning rate will speed up gradient descent and perhaps reach better local minimums but you risk losing accuracy and minimizing the cost function as it bounces around. Too small of a learning rate will slow down learning and potentially miss better extrema.

2] if an activation function with a much larger derivative is used, then you will run into the exploding gradient problem. This is because the weights of the next layer are being multiplied by the derivative of the output and then multiplied together for each layer. $|w_j \sigma'(z_j)|$ Similarly, multiplying together by the sigmoid layers, $|\sigma'(z)| < \frac{1}{4}$ you run into the vanishing gradient. Simply put, the many hidden layers are intrensically unstable having so many products and being tied together. Although having an activation function with a derivative near 1 may help as much as possible.

3] a) $|w\sigma'(wa+b)| \geq 1$

the reason $|w| \geq 4$ is because for a sigmoid activation function, $|\sigma'(wa+b)| \leq 1/4$

try plugging this in, $|w(\frac{1}{4})| \geq 1 \rightarrow |w| \geq 4$

b) to simplify choose $b=0$,

$$|w\sigma'(wa)| \geq 1 \rightarrow \sigma(wa) \geq \frac{1}{|w|}$$

$\rightarrow$ plug in $\frac{e^{-wa}}{(1+e^{-wa})^2} \geq \frac{1}{|w|}$

$\rightarrow$ solve for 0 since σ' is largest at 0 $\quad 0 \geq 1 + (2-|w|)e^{-wa} + (e^{-wa})^2$

$\rightarrow$ this is a quadratic eqn of the form $x^2 + (2-|w|)x + 1 \leq 0$ where $x = e^{-wa}$

$\rightarrow$ using an eqn solver, the discriminant is $\Delta = |w|(|w|-4)$

So for the range, $r_1 \leq x \leq r_2$

$\rightarrow$ plug in x and take log, $\ln r_1 \leq -wa \leq \ln r_2$

$\rightarrow$ Solving a bit more $-\frac{1}{w}\ln r_2 \leq a \leq -\frac{1}{w}\ln r_1$

if $w>0$ the width is, $\frac{1}{w}(-\ln r_1 + \ln r_2) = \frac{1}{w}\ln\frac{r_2}{r_1} = \frac{1}{|w|}\ln\frac{r_2}{r_1}$

if $w<0$ the width is, $-\frac{1}{w}(\ln r_2 - \ln r_1) = -\frac{1}{w}\ln\frac{r_2}{r_1} = \frac{1}{|w|}\ln\frac{r_2}{r_1}$

* cont on next page.

3b continued] for a quadratic function $r_1 r_2 = \frac{c}{a}$

→ Since $|w e^{i(\alpha+\beta)}| = 1$, $\rightarrow r_1 r_2 = 1 \rightarrow \frac{r_2}{r_1} = r_2^2$

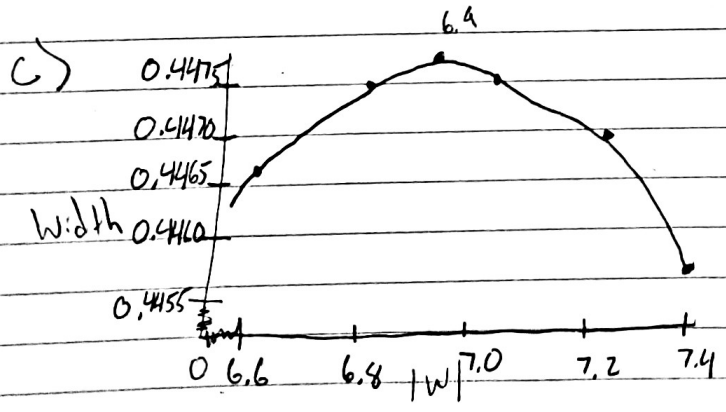
So for the width $\frac{1}{|w|}\ln(r_2^2) = \frac{2}{|w|}\ln r_2$

$r_2 = \frac{-b + \sqrt{\Delta}}{2a}$ → plug in Δ $r_2 = \frac{|w| - 2 + \sqrt{|w|(|w|-4)}}{2}$

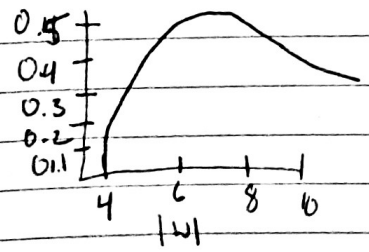
$r_2 = \frac{|w|\left(1 + \sqrt{1 - 4/|w|}\right)}{2} - 1$

So the max width $= \frac{2}{|w|}\ln\left(\frac{|w|\left(1+\sqrt{1-4/|w|}\right)}{2} - 1\right)$
→ plugging in

c)



* notice the range of $|w|$ is very small near the maximum and falls away quickly



4) a choice of $\alpha > 1$ would cause the learning to over-accelerate. this could cause the updates to be huge and miss the minimization. $\alpha < 0$ would cause the updates to possibly go in the wrong direction completely. going up the gradient and increasing the cost.

Problem 3

1)

Achieved a test accuracy of 99.00%
I used:
Epochs = 10
batch size = 100
learning rate = 0.1
Dropout = 0.4
L2 regularization
no momentum
no weight decay
and random initialization

I think most of the success likely came from the network setup. I used the recommended 2 convolutional layers with max pooling layers after each then 3 linear layers to add additional depth. In between the first and second linear layers I added the dropout and then a softmax layer at the end. The test accuracy does seem to depend a bit on initialization. In some later runs it did perform a bit worse for some of them. Also my loss calc looks a little weird being negative, but this is just because of the network. Most of the learning occurs in the earlier epochs. Below is a screenshot of that performance.

```
Epoch [9/10], Step [600/600], Loss: -0.9405
Epoch [10/10], Step [100/600], Loss: -0.9486
Epoch [10/10], Step [200/600], Loss: -0.9766
Epoch [10/10], Step [300/600], Loss: -0.9756
Epoch [10/10], Step [400/600], Loss: -0.9536
Epoch [10/10], Step [500/600], Loss: -0.9356
Epoch [10/10], Step [600/600], Loss: -0.9830
```

[15]:

```
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.9900
```

2)

Using the same hyperparameters as above I tested the L1 regularization.

L1 regularization lambda = 0.001 with random initialization

The L1 regularization generally did not help much especially with random initialization. It achieved an 87% test accuracy in this run.

```
Epoch [10/10], Step [100/600], Loss: -2.2675
Epoch [10/10], Step [200/600], Loss: -2.3384
Epoch [10/10], Step [300/600], Loss: -2.3046
Epoch [10/10], Step [400/600], Loss: -2.3611
Epoch [10/10], Step [500/600], Loss: -2.3442
Epoch [10/10], Step [600/600], Loss: -2.3783
```

.2]:

```
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.8700
```

L1 regularization lambda = 0.001 with L2 weights initialization

In this run, I initialized the L1 training with the L2 weights. This actually had quite a bit of improvement over the random initialized L1 and for the run also improved over the initial L2 training.

For overall performance, L2 regularization seems satisfactory but perhaps can be made more consistent by including L1 and initializing the weights with the L2.

```
Epoch [9/10], Step [500/600], Loss: -2.3912
Epoch [9/10], Step [600/600], Loss: -2.4291
Epoch [10/10], Step [100/600], Loss: -2.4471
Epoch [10/10], Step [200/600], Loss: -2.4638
Epoch [10/10], Step [300/600], Loss: -2.4835
Epoch [10/10], Step [400/600], Loss: -2.5216
Epoch [10/10], Step [500/600], Loss: -2.5383
Epoch [10/10], Step [600/600], Loss: -2.5567
```

]:
```python
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.9700
```

3)

a) Write-up at end of results graphs

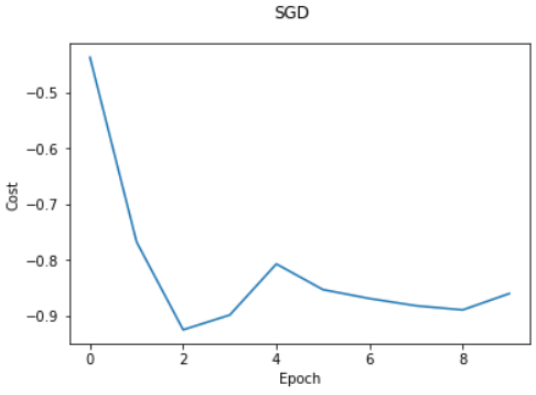
Results with SGD

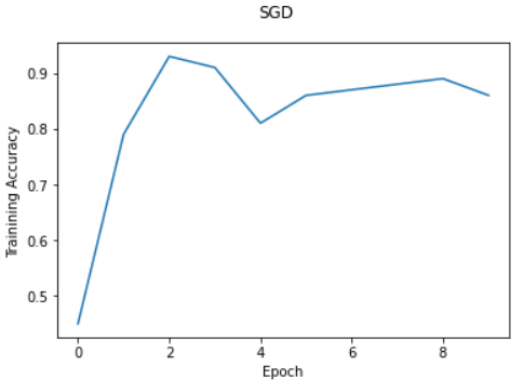
```
Epoch [10/10], Step [500/600], Loss: -0.8212
Epoch [10/10], Step [600/600], Loss: -0.8603
```

```python
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.8800
```





SGD with momentum = 0.1

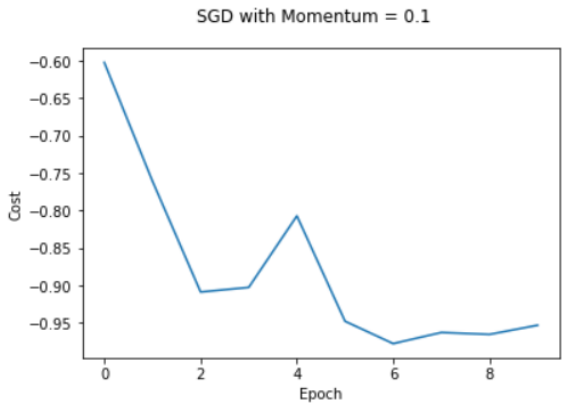
```
Epoch [10/10], Step [300/600], Loss: -0.9522
Epoch [10/10], Step [400/600], Loss: -0.9564
Epoch [10/10], Step [500/600], Loss: -0.9639
Epoch [10/10], Step [600/600], Loss: -0.9533
```

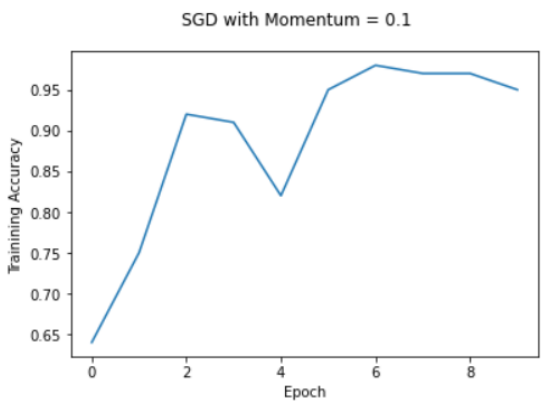
3]:

```python
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.9700
```





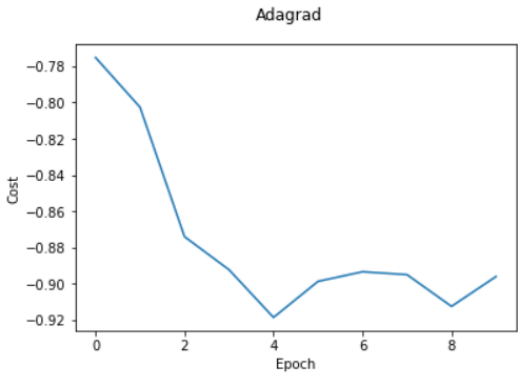
Adagrad with lr = 0.001

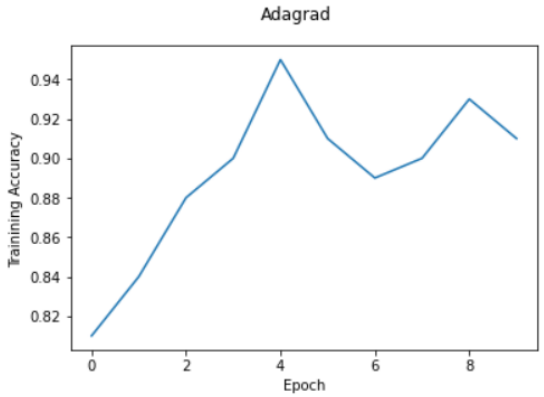
```
Epoch [10/10], Step [400/600], Loss: -0.9141
Epoch [10/10], Step [500/600], Loss: -0.9310
Epoch [10/10], Step [600/600], Loss: -0.8961
```

```python
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.9200
```





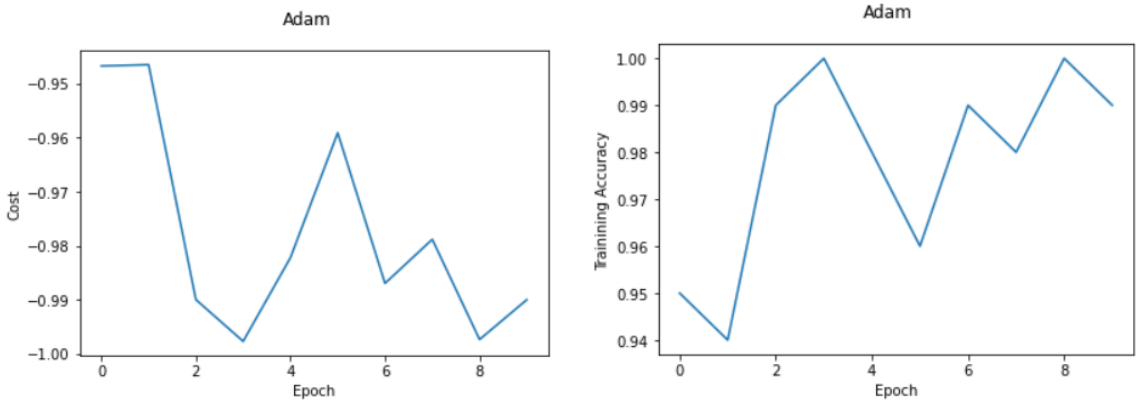
Adam with lr = 0.001

```
Epoch [10/10], Step [400/600], Loss: -0.9674
Epoch [10/10], Step [500/600], Loss: -0.9898
Epoch [10/10], Step [600/600], Loss: -0.9900
```

```
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.9900
```



The best result came from the Adam optimizer. On all of the graphs the test accuracy starts after the first step because of the way I collected the data. But you can see that Adam learns very quickly and maintains a high test and train accuracy. It was also robust to initialization. Adagrad performed okay but was worse than Adam. SGD with momentum has potential as the accuracy reached a fairly high value and learned consistently over training. It seemed to outperform simple SGD.

For these tests, I kept the hyperparameters the same as in part 1 of this problem except for some specific optimizer parameters which I will list below.

Parameter/ Optimizer	SGD	SGD w/ momentum	Adagrad	Adam
Learning rate	0.1	0.1	0.001	0.001
momentum	-	0.1	-	-

b)

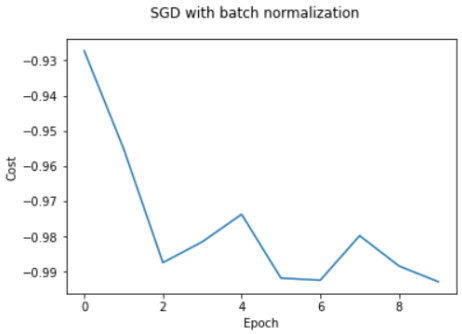
I tested this first on SGD just to see if the batch normalization could improve the worst result and it does dramatically. The train accuracy of SGD goes from about 88% to about 99% and makes the cost trend more consistently

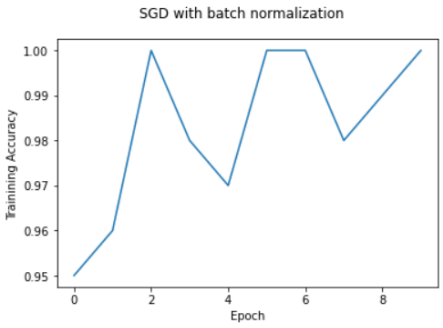
```
Epoch [10/10], Step [500/600], Loss: -0.9835
Epoch [10/10], Step [600/600], Loss: -0.9929
```

```
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 0.9900
```





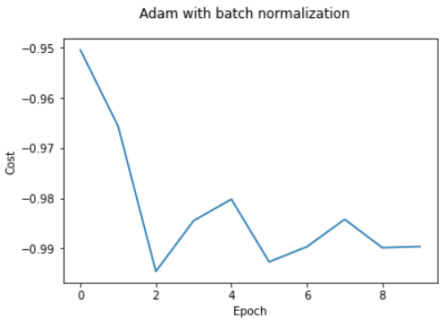
The best result of Adam with batch renormalization also saw some improvement in consistency and maintaining a high accuracy it seems. Note, the 1.0 test accuracy is likely due to rounding

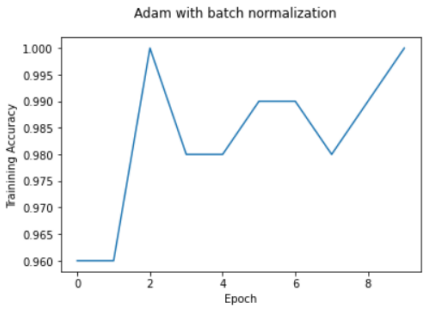
```
Epoch [10/10], Step [500/600], Loss: -0.9876
Epoch [10/10], Step [600/600], Loss: -0.9896
```

```
def test():
    # Test the model
    cnn.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = cnn(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
    print('Test Accuracy of the model on the 10000 test images: %.4f' % accuracy)

    pass
test()
```

```
Test Accuracy of the model on the 10000 test images: 1.0000
```



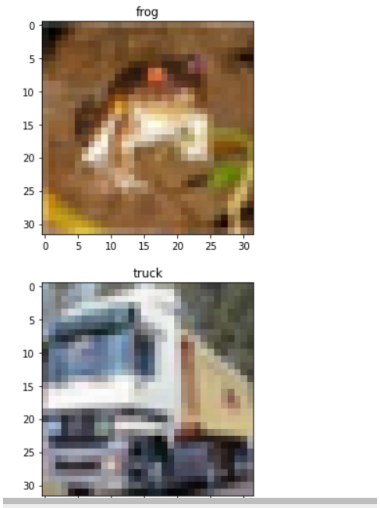


Problem 5

1)

First two image classes displayed below. You can see the full 10 if you run the .ipynb file.

```
Nimg = [0,1,2,3,4,5,6,7,8,9]
for i in range(len(Nimg)):
  plt.imshow(trainset.data[i])
  plt.title(trainset.classes[trainset.targets[i]])
  plt.show()
```



I used two data transformation strategies. Initially I just normalize the data. The first transform tested is to simply resize the images to the same pixel square and make it a tensor.

```
simpleTransform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor()
])
```

The second transformation is a bit more complex, code snippet is below.

```
preprocess = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

2)

Here is my image of corn.

```
from PIL import Image
myCorn = Image.open("corn_pic2.jpg")
myCorn
```

Correct identification:

```
imagenet_labels[probs.argmax()]
```

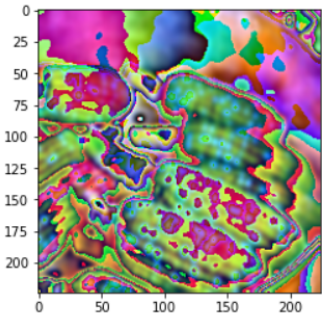
```
"987: 'corn',"
```

Trick Transformation (notice this is the preprocessing from 1):

```
preprocess = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    transforms.ColorJitter(),
    transforms.GaussianBlur(51)
])
myCornT = preprocess(myCorn)
```

```
plt.imshow(transforms.ToPILImage()(myCornT))
```

```
<matplotlib.image.AxesImage at 0x1e6838390d0>
```



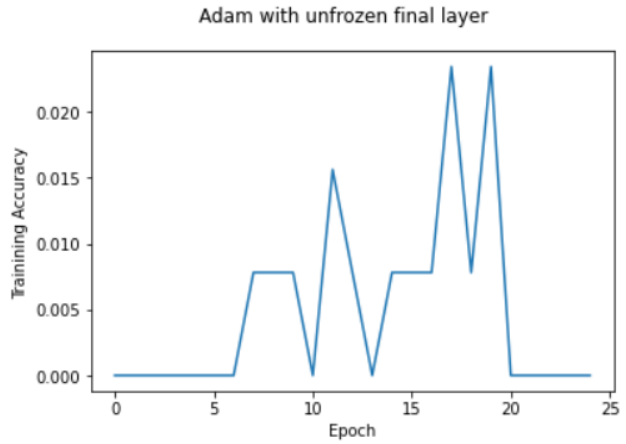

```
imagenet_labels[probs.argmax()]
```

```
"926: 'hot pot, hotpot',"
```

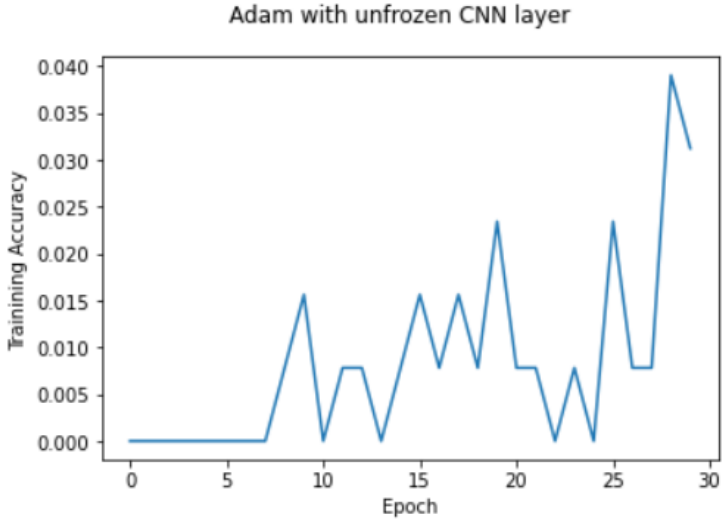
3)

The early stopping strategy implemented was checking whether the validation loss improved or not with a patience counter of 10. I used cross entropy loss and the Adam optimizer with a learning rate of 0.0001. This first set is with the frozen layers other than the final layer.

Notice that these graphs are the validation accuracy not the cost per epoch. The cost improvement is more smooth but stalls out.

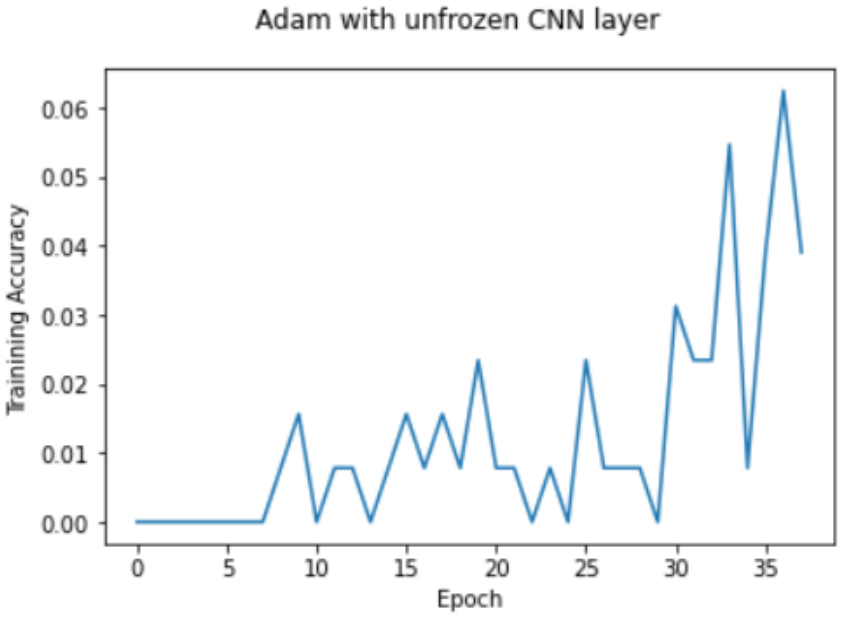


4)



Here you can see the improvement of unfreezing the CNN layer. Again I used the same hyperparameters as in 3).

Below I decided to increase the patience on my early stopping to 15 and was able to see a bit better accuracy by the end of training.


Adam with unfrozen CNN layer
Trainining Accuracy
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
5
10
15
20
25
30
35
Epoch