ECE 498NSU/598NSG:
Deep Learning in Hardware Fall 2020

HW 3:

Abhi Kamboj

Table of Contents:

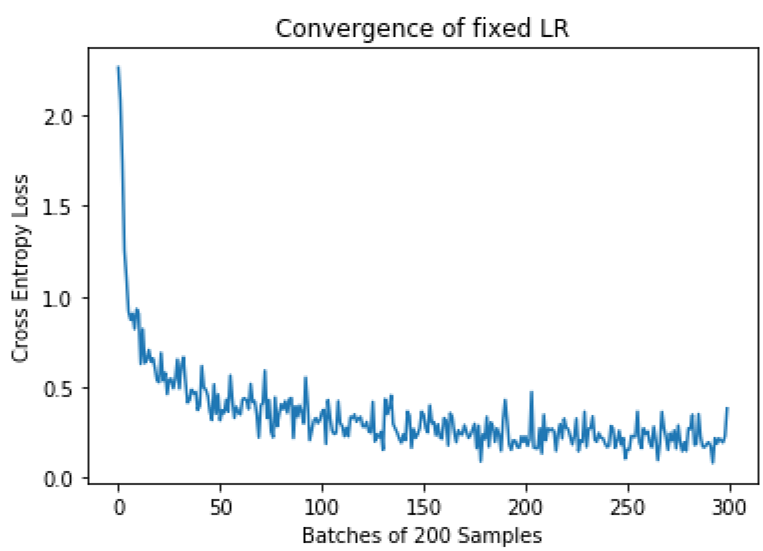
Problem 1:

NOTE: My computer has quite a slow CPU, so it took longer than 10 minutes to run through all 60,000 MNIST images. On some of the plots, I show less than 60,000 samples, however, convergence can still be seen. This is also the reason I plot cross entropy loss instead of test error, because running the test images every epoch takes even longer, to get an error.

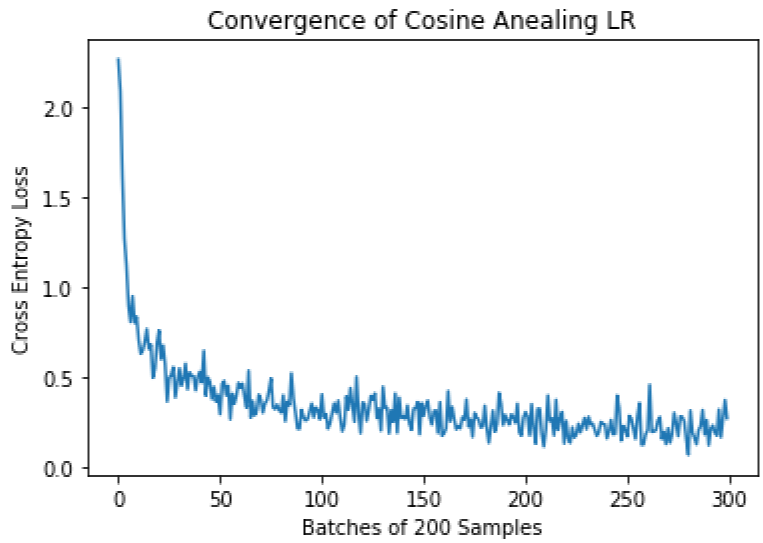
1.1

Note: In the following graphs I labedled the x-axis as “Batches fo 200 Samples.” What I meant was the algorithm plots the average loss after 200 samples (you could say that is the epoch size, or ensemble average size), and only one sample is being run through the network at a time. Since there are 60,000 images, there were 300 sets of the 200 sample average losses.

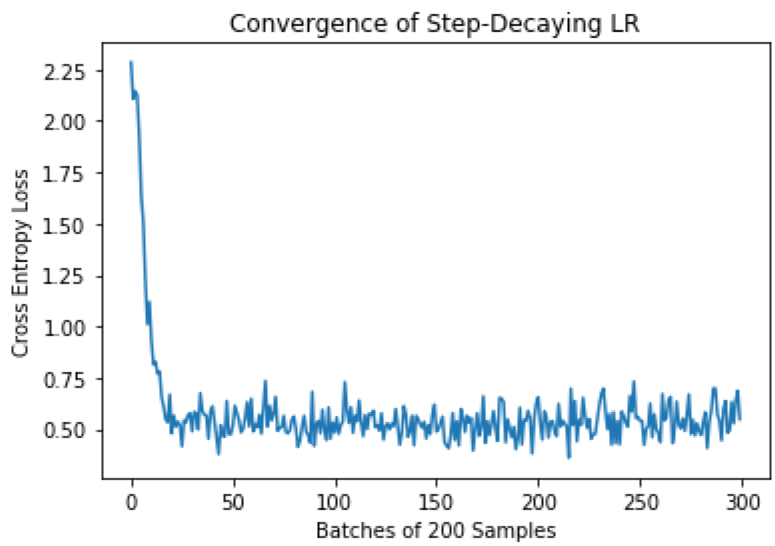
Part A: Fixed Learning Rate of 0.001



Part B: Starting with Learning_Rate=.001, T_max=.1



Part C: Starting at Learning_Rate=.01, step_size=100, gamma=.9



Note: Although I plotted Loss, I tested the model afterwards and received a little higher than 90% accuracy, so I knew these models were learning.

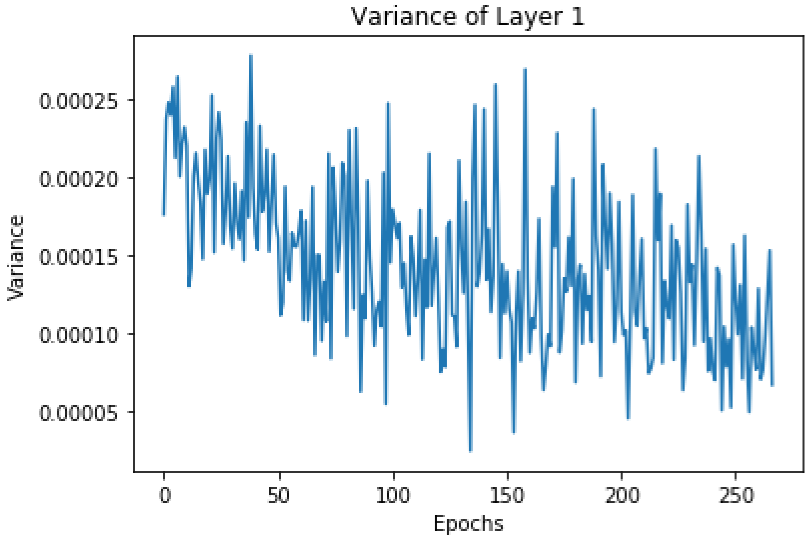
1.2

We are working with a networks with the following layers:

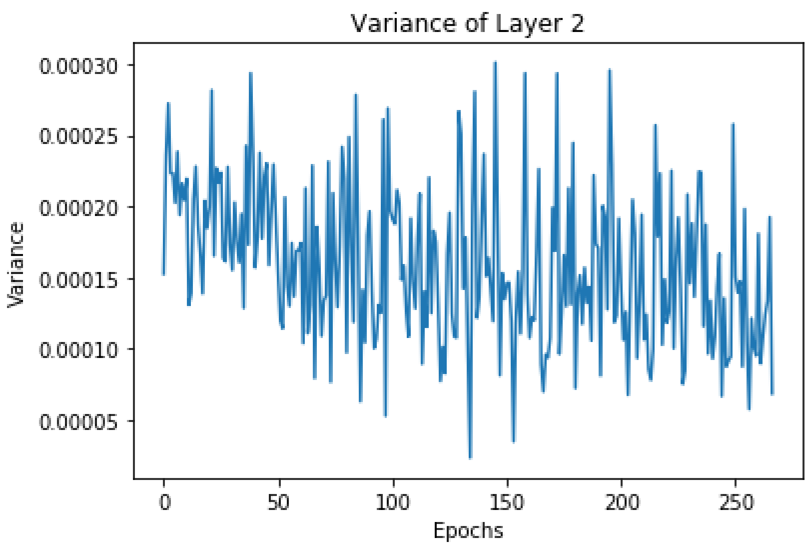
```
nn.Linear(784,512), #each image is 28x28 pixels
nn.ReLU(),
nn.Linear(512,256),
nn.ReLU(),
nn.Linear(256, 256),
nn.ReLU(),
nn.Linear(256, 10)
```

With an input of 784. We train with 200 samples being in an epoch.

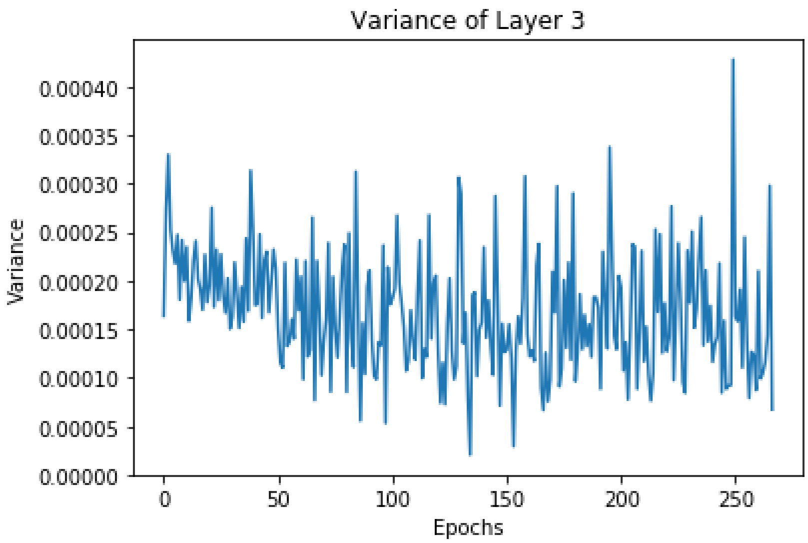
Linear Layer 1:
Max Variance 0.00027849411170818766
Min Variance 2.4069021423730302e-05



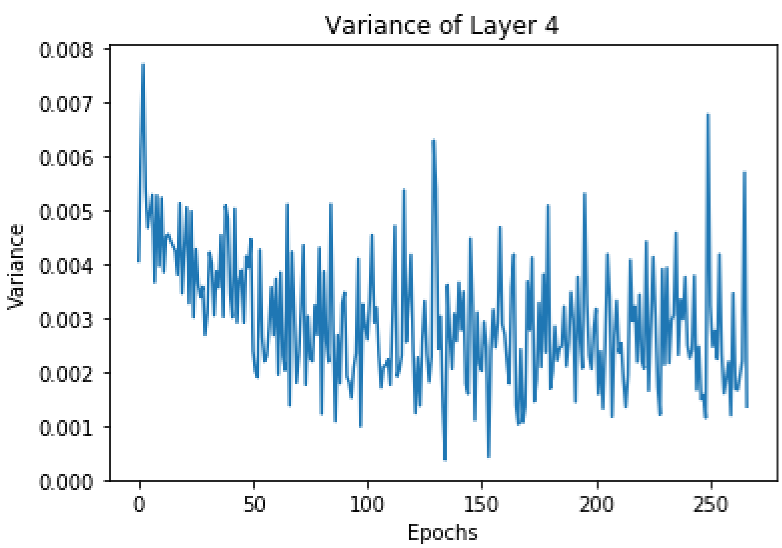
Linear Layer 2:
Max Variance 0.00030124222271816758
Min Variance 2.3078324989063e-05



Linear Layer 3:
Max Variance 0.0004279109396935473
Min Variance 2.0742843151225246e-05



Linear Layer 4:
Max Variance 0.007700157304747548
Min Variance 0.0003670435271409234



1.3

1.3

To obtain close-to-minimal quantization, we use criteria 2 and 3:

Clipping level: $r_{G_l(w)} \geq 2\sigma^{(max)}_{G_l(w)}$

Quantization stepsize: $\Delta_{G_l(w)} < \dfrac{\sigma^{(min)}_{G_l(w)}}{4}$

Since the min and max values for σ^2 was determined in part 1.2 we use those values to calculate this for each of the layers.

And to compute the precision Bg, I did log_2(clipping/step)+1

Layer 1:

- Clipping Level: 0.033376286017582996
- Stepsize: 6.0172553559325756e-06
- Bg: 12.0

Layer 2:

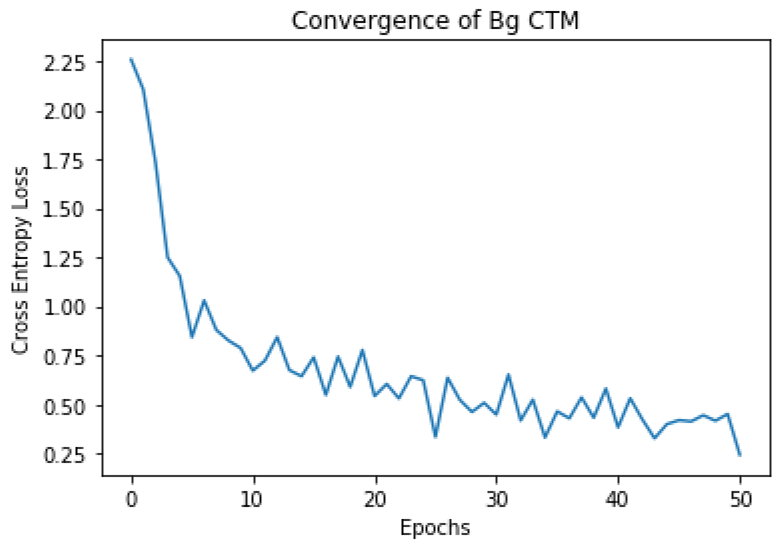
- Clipping Level: 0.034712662080668824
- Stepsize: 5.76958124726575e-06
- Bg: 13.0

Layer 3:

- Clipping Level: 0.04137201661478673
- Stepsize: 5.1857107878063114e-06
- Bg: 13.0

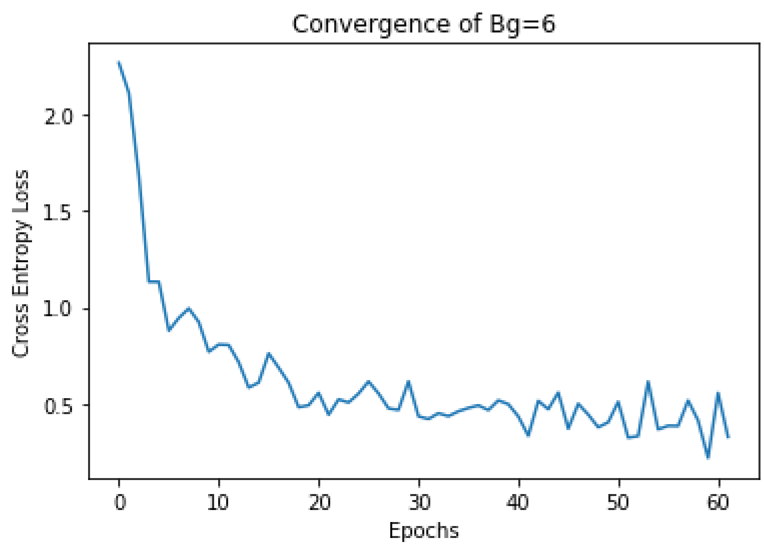
Layer 4:

- Clipping Level: 0.17550108039265797
- Stepsize: 9.176088178523085e-05
- Bg: 11.0

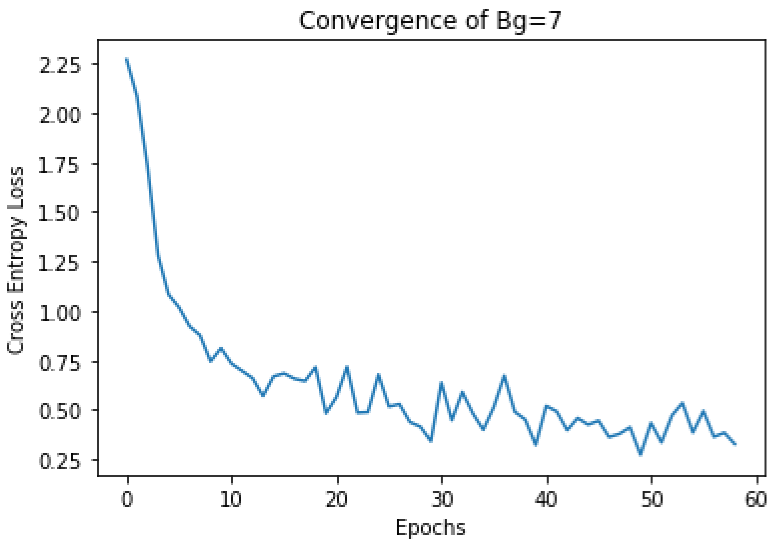


1.4

A) 6 bits



B) 7 bits



We can see that since we determined in part 1.3 that larger precisions were needed for the weight gradients to be close-to-minimal, the convergence took longer for these smaller precisions.

Problem 2:

2.1

3.1

For a convolutional layer in forward propagation the response is

(1) $y_l = W_l x_l + b_l$ where W_l is $d \times n$ matrix of all the filters as rows

x_l is a $k^2 c \times 1$ vector of k^2 colocated pixels in the input features across c channels.

b is a vector of biases, and y is response of a pixel for the different filter.

Let elements of W_l and x_l are mutually independent and share the same distribute, and let w_l and x_l be mutually independent then: $Var[y_l] = n_l Var[w_l x_l]$

where y_l, w_l and x_l represent random variables of elements in y_l, W_L and x_l described in eq. (1)

$$Var[y_l] = n_l Var[w_l x_l]$$

$\Downarrow$ $\rightarrow$ here we let w_l have zero mean.

$$Var[y_l] = n_l Var[w_l] E[x_l^2]$$

$E[x_l^2] = \frac{1}{2} Var[y_{l-1}]$ because we let w_{l-1} have a symmetric distribution around zero and $b_{l-1} = 0$

hence y_{l-1} has zero mean + symmetric distribution

Notice the factor of $\frac{1}{2}$ because of the ReLU activation:

ReLu: $x_l = \max(0, y_{l-1})$

$$Var[y_l] = \frac{1}{2} n_l Var[w_l] Var[y_{l-1}]$$

now if we apply this across L layers:

$$\therefore Var[y_L] = Var[y_1] \prod_{l=2}^{L} \frac{1}{2} n_l Var[w_l]$$

2.2

3.2

Far backward propogation we have:

$\Delta x_l = \hat{W}_l \Delta y_l$, where Δx_l and Δy_l denote gradients $\frac{\partial \varepsilon}{\partial x}$ and $\frac{\partial \varepsilon}{\partial y}$ respectively.

Δy_l represents $k \times k$ pixels in d channels

Δx_l represents a $c \times 1$ vector of channels

$\hat{W}$ is a $c \times \hat{n}$ matrix where $\hat{n} = k^2 d$

We assume w_l and Δy_l are independent of each other, hence. Δx_l has zero mean when w_l is initialized by a symmetric distribution with zero mean ($\Delta x_l = \hat{W}_l \Delta y_l$

$\Rightarrow E[\Delta x_l] = E[w_l]E[\Delta y_l) = 0$ when $E[w_l] = 0$)

For backward propogation we have $\Delta y_l = f'(y_l)\Delta x_{l+1}$ where $f'(y_l)$ is derivative of relu which is zero one with probability $\frac{1}{2}$. Then:

$$E[\Delta y_l] = E[\Delta x_{l+1}]/2 = 0 \Rightarrow E[(\Delta y_l)^2] = Var[\Delta y_l]$$

thus: $Var[\Delta x_l] = \hat{n}_l Var[w_l] Var[\Delta y_l]$ for each layer l

since $\Delta y_l = f'(y_l)\Delta x_{l+1}$ and $f'(y_l)$ is nonzero $\frac{1}{2}$ the time we have $Var[\Delta y_l] = \frac{1}{2} Var[\Delta x_{l+1})$

$$\Rightarrow Var[\Delta x_l] = \frac{1}{2}\hat{n}_l Var[w_l] Var[\Delta x_{l+1}]$$

for all the layers we have:

$$Var[\Delta x_2] = Var[\Delta x_{L+1}) \prod_{l=2}^{L} \frac{1}{2}\hat{n}_l Var[w_l]$$

2.3

In a convolutional layer, how are the forward dot-product length nl and backward dot-product length ˆnl related?

In convolution, the forward dot product length applies a k-by-k kernel filter across *c* different channels. Applying this filter creates a new output channel. After applying *d* filters, *c* input channels turns into d output channels. Hence during forward propagation we have n=*c*k*k* length for the dot product and we do the dot product for every pixel *d* times. For backwards propagation, we use the *d* output channels and go backwards resulting in *c* input channels. As a result we do the dot product of length n_hat =*d*k*k* and we do *c* of those dot procuts. Note: the overall dimension of the weight vector W doesn’t change, it’s always c*d*k*k, it’s just reshaped from d-by-n in forward propagation to c-by-n_hat in backwards propagation.

2.4

Explain how the He initialization prevents the vanishing or explosion of activations in the forward propagation.

The intuition behind He initialization is to control the variance of the activation responses in each layer. From part 2.1 we can see that the overall variance of the layers is dependent on the variance of the weights. We want to prevent the initial input signals magnitudes from reducing or magnifying exponentially, so we need to restrict the variance of the activations. From 2.1, we can see that this can be done by setting the value in the product term to 1 and then the Variance of the final layer will be the same as the variance of the initial layer and we will not have to explode or vanishing activations.
He initialization:

$$\frac{1}{2} n_l Var[w_l] = 1, \quad \forall l.$$

2.5

Explain how the He initialization prevents the vanishing or explosion of gradients in the backward propagation (HINT: the answer to this question is not the same as that of the previous one).

Now we have a similar idea 2.4, except we are discussing the gradients. We know that the gradient of the outputs after the first activation (delta_x_2) is dependent on all the following layers' gradients, so there must be some way to write its variance in terms of the following layers' gradients. We determined this in part 2.2 and we can see here again that we can restrict the variance of the weights during initialization to hold the variance of the first layer's gradient to be the same as the variance of the last layer's gradient if we keep the value of the term in the product as 1. This means that the gradients could not explode or vanish with increasingly large or small values.
He initialization:

$$\frac{1}{2} \hat{n}_l Var[w_l] = 1, \quad \forall l.$$

Problem 3:

3.1

$$\gamma \frac{\langle w, x \rangle - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \qquad W = \begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \end{pmatrix} \qquad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$= \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \begin{pmatrix} w_1 x_1 + w_2 x_2 - \mu \\ w_3 x_1 + w_4 x_2 - \mu \end{pmatrix} + \beta$$

$$= \langle \hat{w}, x \rangle + b \implies \boxed{\begin{aligned} \hat{w} &= \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} w \\ b &= \beta - \frac{\mu}{\sqrt{\sigma^2 + \epsilon}} \end{aligned}}$$

$$\hat{x} = \frac{x - E(x)}{\sqrt{\mathrm{Var}[x] + \epsilon}}$$

, for inference $E(x)$ and variance (x) are statistics across all samples not just a minibatch, so they can be absorbed into the weights as shown above.

3.2

Problem 3.2

In [40]:

```python
#we assume w is given as (nSamples,channels,H,W), where H,W is the 2d feature map
#for 2d convultion the normalization is taken across the channels, since filterweights are shared across the features.

def find_w_b(w,gamma,beta,mu,std,epsilon):
    #reshapes w_scalar to apply accross channels
    w_scalar = gamma/np.sqrt(std**2 + epsilon)
    return w*w_scalar.reshape(1,-1,1,1), beta - mu/np.sqrt(std**2 + epsilon)

#Example:
w = np.ones((2,4,2,2))
gamma = np.array([1,2,3,4])
beta = np.ones(4)*3
mu = np.ones(4)*2
std = np.ones(4)*4
epsilon = np.ones(4)*84

w_hat, b = find_w_b(w,gamma,beta,mu,std,epsilon)
print("w_hat:\n",w_hat)
print("b:\n",b)
```

```
w_hat:
 [[[[0.1 0.1]
   [0.1 0.1]]

  [[0.2 0.2]
   [0.2 0.2]]

  [[0.3 0.3]
   [0.3 0.3]]

  [[0.4 0.4]
   [0.4 0.4]]]


 [[[0.1 0.1]
   [0.1 0.1]]

  [[0.2 0.2]
   [0.2 0.2]]

  [[0.3 0.3]
   [0.3 0.3]]

  [[0.4 0.4]
   [0.4 0.4]]]]
b:
 [2.8 2.8 2.8 2.8]
```

3.3

I added 3 Batch normalization layers between the linear layer and Relu layers as follows:

```
self.engine = nn.Sequential(
        nn.Linear(784,512), #each image is 28x28 pixels
        nn.BatchNorm1d(512),
        nn.ReLU(),
        nn.Linear(512,256),
        nn.BatchNorm1d(256),
        nn.ReLU(),
        nn.Linear(256, 256),
        nn.BatchNorm1d(256),
        nn.ReLU(),
        nn.Linear(256, 10)
    )
```

Each Batch Norm Layer stores parameters gamma and beta for each of the features so we have 2*(512+256+256) = 2048 new parameters. I ran the algoirhtms with Batches of 5 and displayed the average loss of each batch every 40 batches (ie. ensemble average every 40 batches):