MLPs, CNNs and Backpropagation

Tarun Krishna
University of Amsterdam
11593040
tarun.krishna@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

$$\texttt{LinearModule.forward: } \left(x^{l-1}, W^l, b^l\right) \rightarrow \tilde{x}^l$$
$$\texttt{ReLUModule.forward: } \left(\tilde{x}^l\right) \rightarrow x^l$$
$$\texttt{SoftmaxModule.forward: } \left(\tilde{x}^N\right) \rightarrow x^N$$
$$\texttt{CrossEntropyModule.forward: } \left(\tilde{x}^N, t\right) \rightarrow \mathcal{L}(x^N, t)$$

1.1.1 Compute the gradients

I will first compute the scalar gradients and subsequently will reframe them into matrix/vector calculation.

1. $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^N}$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_m^N} &= -\sum_{i=1}^{n_{class}} t_i \log\left(x_i^N\right) \\
&= -\frac{\partial}{\partial x_m^N}\Big[\sum_{i \neq m}^{n_{class}} t_i \log\left(x_i^N\right)\Big] - \frac{\partial}{\partial x_m^N}\left[t_m \log x_m^N\right] \\
&= \begin{cases} 0, & \text{if} \quad i \neq m \\ -\dfrac{t_m}{x_m^N}, & \text{if} \quad i = m \end{cases} \\
&= -\frac{t_m}{x_m^N}
\end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^N} = \begin{bmatrix} -\frac{t_1}{x_1^N} \\ \cdot \\ \cdot \\ \cdot \\ -\frac{t_{n_{class}}}{x_{n_{class}}^N} \end{bmatrix}.$$

2. $\frac{\partial \mathbf{x}^N}{\partial \tilde{\mathbf{x}}^N}$

The gradient would be a matrix of size $d_N \times d_N$ as $(d_N, 1)$ is the dimensionality of both $\mathbf{x}^N$ and $\partial \tilde{\mathbf{x}}^N$

$$\begin{aligned}\frac{\partial x_i^N}{\partial \tilde{x}_m^N} &= \frac{\partial}{\partial \tilde{x}_m^N}\left[\frac{\exp \tilde{x}_i^N}{\sum_{j=1} \exp \tilde{x}_j^N}\right] \\ &= \begin{cases} x_i^N(1-x_i^N), & \text{if} \quad i=m \\ -x_i^N x_m^N, & \text{otherwise} \end{cases} \\ &= x_i^N(\delta_{i,m} - x_m^N) \quad ,\delta_{i,m}=1 \text{ when } i=m, \text{ else } 0\end{aligned}$$

$\frac{\partial \mathbf{x}^N}{\partial \tilde{\mathbf{x}}^N} = \mathbf{x}^N \mathbf{e}^T \odot (\mathbf{I} - \mathbf{e}(\mathbf{x}^N)^T)$ where $\mathbf{e}$ = vector of ones of same dimensionality as $\mathbf{x}^N$ i.e $(d_N, 1)$.
So this will give us a matrix of the form:

$$\frac{\partial \mathbf{x}^N}{\partial \hat{\mathbf{x}}^{(N)}} = \begin{bmatrix} x_1^N(1-x_1^N) & -x_1^N x_2^N & \dots & -x_1^N x_{d_N}^N \\ -x_2^N x_1^N & x_2^N(1-x_2^N) & \dots & -x_1^N x_{d_N}^N \\ \vdots & \vdots & \vdots & \vdots \\ -x_{d_N}^N x_1^N & -x_1^N x_2^N & \dots & -x_{d_N}^N(1-x_{d_N}^N) \end{bmatrix}$$

3. $\frac{\partial \mathbf{x}^{(l<N)}}{\partial \tilde{\mathbf{x}}^{(l<N)}}$

Again the gradient of a vector with respect to a vector would be a matrix of size $d_l \times d_l$.

$$\begin{aligned}\frac{\partial x_i^l}{\partial \tilde{x}_j^l} &= \frac{\partial}{\partial \tilde{x}_j^l}\big[max(0, \tilde{x}_i^l)\big] \\ &= \frac{\partial\big[max(0, \tilde{x}_i^l)\big]}{\partial \tilde{x}_i^l}\frac{\partial \tilde{x}_i^l}{\partial \tilde{x}_j^l} \\ &= \begin{cases} 0, & \text{if} \quad i \neq j \\ \underbrace{\frac{\partial\big[max(0, \tilde{x}_i^l)\big]}{\partial \tilde{x}_i^l}}_{\text{Derivative of ReLU}}, & \text{otherwise} \end{cases} \\ &= \delta_{i,j}\theta[\tilde{x}^l]_j\end{aligned}$$

$\frac{\partial \mathbf{x}^{(l<N)}}{\partial \tilde{\mathbf{x}}^{(l<N)}} = \mathbf{I}\theta[\tilde{x}^l]$. Where $\theta[\tilde{x}^l]$ is a vector $\begin{bmatrix} \frac{\partial\big[max(0,\tilde{x}_1^l)\big]}{\partial \tilde{x}_1^l} \\ \frac{\partial\big[max(0,\tilde{x}_2^l)\big]}{\partial \tilde{x}_2^l} \\ . \\ . \\ . \\ \frac{\partial\big[max(0,\tilde{x}_h^l)\big]}{\partial \tilde{x}_h^l} \end{bmatrix}$ *h* denotes number of hidden units in *l* and $\mathbf{I}$ identity matrix, so does the dimension of $\frac{\partial \mathbf{x}^{(l<N)}}{\partial \tilde{\mathbf{x}}^{(l<N)}}$ will be $h \times h$. Also, $\frac{\partial \max(0,x)}{\partial x} = \begin{cases} 1, & \text{if} \quad x > 0 \\ 0, & \text{otherwise} \end{cases}$

4. $\frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{x}^{l-1}}$

$$\begin{aligned}\frac{\partial \tilde{x}_i^l}{\partial x_j^{l-1}} &= \frac{\partial\big[\sum_{h=1} W_{ih}^l x_h^{l-1} + b_i^l\big]}{\partial x_j^{l-1}} \\ &= W_{ij}^l\end{aligned}$$

$\frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{x}^{l-1}} = \mathbf{W}^l$. Where dimensions of $\mathbf{W}^l$ is $d_l \times d_{l-1}$. Which is again a matrix that one should expect.

5. $\frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{W}^l}$

$$\begin{aligned}\frac{\partial \tilde{x_i}^l}{\partial W_{jk}^l} &= \frac{\partial \big[\sum_{h=1} W_{ih}^l x_h^{l-1} + b_i^l\big]}{\partial W_{jk}^l} \\ &= \begin{cases} x_k^{l-1}, & \text{if} \quad j = i \\ 0, & \text{otherwise} \end{cases} \\ &= \delta_{i,j} x_k^{l-1}\end{aligned}$$

This $\frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{W}^l}$ be a 3 dimensional matrix but can be represented into 2D matrix with dimension same as $\mathbf{W^1}$ with rows as defined above.

6. $\frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{b}^l}$

$$\begin{aligned}\frac{\partial \tilde{x}_i^l}{\partial b_j^l} &= \begin{cases} 1, & \text{if} \quad i = j \\ 0, & \text{otherwise} \end{cases} \\ &= \delta_{i,j}\end{aligned}$$

$\frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{b}^l} = \mathbf{I}$, as the gradient will be a matrix.

1.1.2 Using the gradients of the modules calculate the gradients

Assuming all the gradients derive so far are matrix or *column vector*.

1. $\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{x}}^N} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^N} \frac{\partial \mathbf{x}^N}{\partial \tilde{\mathbf{x}}^N}$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \tilde{x}_j^N} &= \sum_i \frac{\partial \mathcal{L}}{\partial x_i^N} \frac{\partial x_i^N}{\partial \tilde{x}_j^N} \\ &= \sum_i \frac{\partial \mathcal{L}}{\partial x_i^N} x_i^N (\delta_{i,j} - x_j^N) \\ &= \big(\frac{\partial \mathbf{x}^N}{\partial \tilde{\mathbf{x}}^N} \frac{\partial \mathcal{L}}{\partial \mathbf{x}^N}\big)_j\end{aligned}$$

All the derivative $\frac{\partial \mathbf{x}^N}{\partial \tilde{\mathbf{x}}^N}$ is matrix of $d_N \times d_N$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^N}$ is a column vector with size $(d_N, 1)$. So based on that $\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{x}}^N}$ this could be formulated into simple matrix multiplication of a matrix and a column vector to gives a column vector of size $(d_N, 1)$. A complete matrix multiplication can be shown as:

$$\begin{bmatrix} x_1^N(1-x_1^N) & -x_1^N x_2^N & \dots & -x_1^N x_{d_N}^N \\ -x_2^N x_1^N & x_2^N(1-x_2^N) & \dots & -x_1^N x_{d_N}^N \\ \vdots & \vdots & \vdots & \vdots \\ -x_{d_N}^N x_1^N & -x_1^N x_2^N & \dots & -x_{d_N}^N(1-x_{d_N}^N) \end{bmatrix} \begin{bmatrix} -\frac{t_1}{x_1^N} \\ \cdot \\ \cdot \\ \cdot \\ -\frac{t_{n_{d_N}}}{x_{d_N}^N} \end{bmatrix}$$

2. $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l<N)}}$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_i^{(l<N)}} &= \sum_j \frac{\partial L}{\partial \tilde{x}_j^{l+1}} \frac{\partial \tilde{x}_j^{l+1}}{\partial x_i^l} \\ &= \sum_j \frac{\partial L}{\partial \tilde{x}_j^{l+1}} W_{ji}^{l+1}\end{aligned}$$

Well this could be realized in vector notation as $(\mathbf{W}^{l+1})^T \frac{\partial L}{\partial \tilde{\mathbf{x}}^{l+1}}$. Where $\frac{\partial L}{\partial \tilde{\mathbf{x}}^{l+1}}$ would be received from the previous layers which is again a column vector.

3. $\frac{\partial L}{\partial \tilde{\mathbf{x}}^{l<N}}$

$$\begin{aligned}\frac{\partial L}{\partial \tilde{x}_j^{l<N}} &= \sum_i \underbrace{\frac{\partial L}{\partial x_i^{l<N}}}_{\text{derived just above}} \frac{\partial x_i^{l<N}}{\partial \tilde{x}_j^{l<N}} \\ &= \sum_i \frac{\partial L}{\partial x_i^{l<N}} \delta_{i,j} \theta[\tilde{x}^l]_j \\ &= \frac{\partial L}{\partial x_j^{l<N}} \theta[\tilde{x}^l]_j\end{aligned}$$

Hence $\frac{\partial L}{\partial \tilde{\mathbf{x}}^{l<N}}$ can be written in matrix form as $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l<N)}} \odot \theta[\tilde{x}^l]$ where $\theta[\tilde{x}^l]$ is derivative of ReLU, also this can be written completely in form of matrix multiplication form as $(\mathbf{W}^{l+1})^T \frac{\partial L}{\partial \tilde{\mathbf{x}}^{l+1}} \odot \theta[\tilde{x}^l]$.

4. $\frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \tilde{\mathbf{x}}^l} \frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{W}^l}$

$$\begin{aligned}\frac{\partial L}{\partial W_{jk}^l} &= \sum_i \frac{\partial L}{\partial \tilde{x}_i^l} \frac{\partial \tilde{x}_i^l}{\partial W_{jk}^l} \\ &= \sum_i \frac{\partial L}{\partial \tilde{x}_i^l} \delta_{j,i} x_k^{l-1} \\ &= \frac{\partial L}{\partial \tilde{x}_j^l} x_k^{l-1} \\ &= (\frac{\partial L}{\partial \tilde{\mathbf{x}}^l} \otimes (\mathbf{x}^{l-1}))_{jk}\end{aligned}$$

In matrix/vector form it can be written as $\frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \tilde{\mathbf{x}}^l} \otimes (\mathbf{x}^{l-1}) = \frac{\partial L}{\partial \tilde{\mathbf{x}}^l} (\mathbf{x}^{l-1})^T$ (just the expanded form of outer product to avoid confusion). Also I'm considering my $\frac{\partial L}{\partial \tilde{\mathbf{x}}^l}$ and $\mathbf{x}^{l-1}$ as a column vector.

5. $\frac{\partial L}{\partial \mathbf{b}^l} = \frac{\partial L}{\partial \tilde{\mathbf{x}}^l} \frac{\partial \tilde{\mathbf{x}}^l}{\partial \mathbf{b}^l}$

$$\begin{aligned}\frac{\partial L}{\partial b_j^l} &= \sum_i \frac{\partial L}{\partial \tilde{x}_i^l} \frac{\partial \tilde{x}_i^l}{\partial b_j^l} \\ &= \sum_i \frac{\partial L}{\partial \tilde{x}_i^l} \delta_{i,j} \\ &= \frac{\partial L}{\partial \tilde{x}_j^l}\end{aligned}$$

$\frac{\partial L}{\partial \mathbf{b}^l} = \frac{\partial L}{\partial \tilde{\mathbf{x}}^l}$

1.1.3 Argue how the backpropagation equations derived above change if a batchsize $B \neq 1$ is used.

$L_{total} = \frac{1}{B}\sum_s^B \mathcal{L}_{individual}\left(x^{(0),s}, t_s\right)$

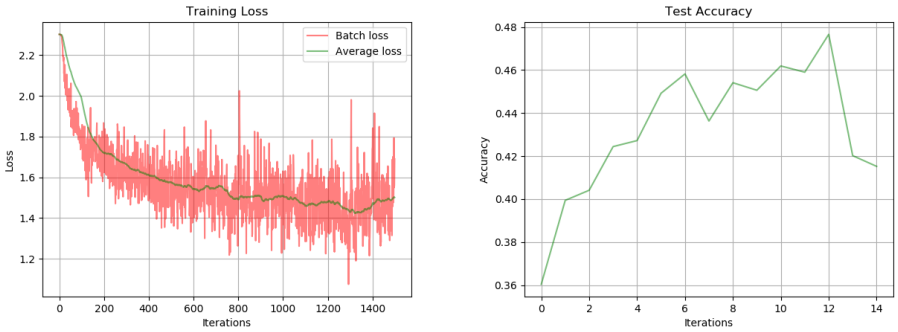
Now we want to calculate derivative of $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^N}$ we need to do it over batches :

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial (x_i^k)^N} &= \frac{1}{B}\partial \frac{\left(\sum_s^B \mathcal{L}_{individual}\left(x^{(0),s}, t_s\right)\right)}{\partial L^k}\frac{\partial L^k}{\partial (x_i^k)^N} \\ &= -\frac{1}{B}\sum_s \delta_{k,s}\frac{\partial L^k}{\partial (x_i^k)^N} \\ &= -\frac{1}{B}\frac{t_i^k}{x_i^k}\end{aligned}$$

Here I have considered $\mathbf{x}_N$ as a matrix of size $B \times d_N$ and accordingly I have computed the gradient. B is my batch size and d_N is number of hidden units in my last layer. Each loss here remains independent over batches. Rest of the gradients in the model can be calculated in the same way as we derived it analytically in previous sections but this time they will be divided over by batch size as the gradient that flows from last layeroutput layer is averaged as shown above.

1.2 NumPy implementation

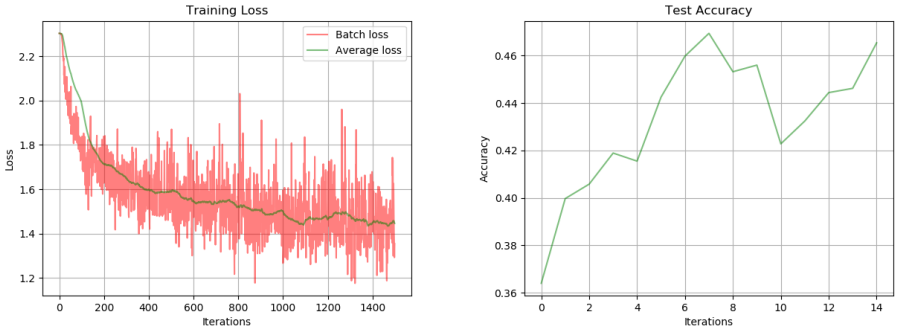
Accuracy and Loss curves for default parameters Figure 1.



(a) Training Loss (b) Test Accuracy

Figure 1: Accuracy and Loss Curves for NumPy implementation

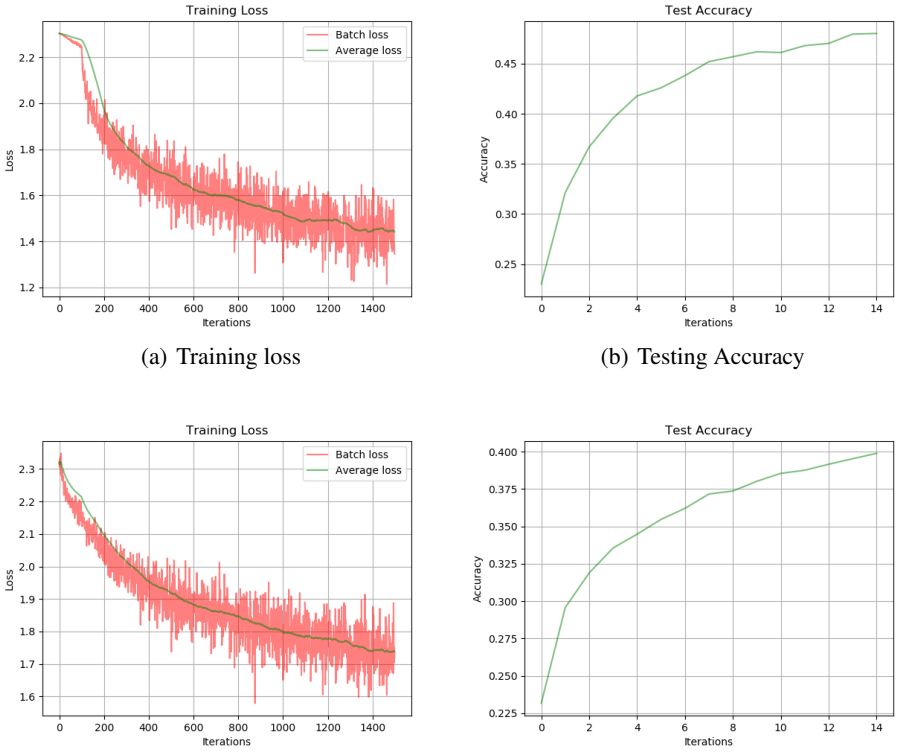
2 PyTorch MLP



(a) Training Loss (b) Test Accuracy

Figure 2: Accuracy and Loss Curves for PyTorch implementation

In this section we initially compare our `numpy` implementation with `pytorch` implementation in order to validate the correctness of our code. In Figure 1 and Figure 2 depicts the accuracy and loss curves for both the implementations `numpy` and `PyTorch` respectively. Both the implementation achieves the similar accuracy of 0.46 on testing data.



(a) Training loss (b) Testing Accuracy

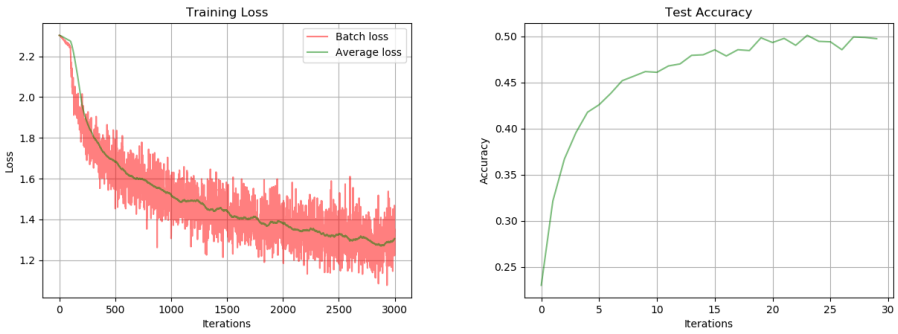
(c) Training loss (d) Testing Accuracy

Figure 3: TEST 1: Batch Normalization: `Top-row`: weight initialization using normal distribution, `Bottom-row`: random weight initialization

After initial validation of our implementation. We further moved on to experimenting with model parameters to get more insights about the model.

1. **Experiment 1**. To start with, we initially inserted *Batch normalization* to our existing `PyTorch` implementation i.e with default parameters as given. Including *Batch normalization* boosted our accuracy by 2% i.e reaching over 0.48 as depicted in Figure 3(a), 3(b)having faster convergence as compared to previous results in Figure 1, 2. This comes to no surprise why including *Batch normalization* to the existing model boosted the results and also providing better convergence rate. Here, we tested *Batch normalization* with random weight initialization included in the *nn.Parameters* itself and another initialization as given in the assignment i.e using normal distribution. Results for both the initialization is depicted in Figure 3(a), 3(b) and Figure 3(c), 3(d). Normal weight initialization performs fairly better than random weight initialization as depicted in Figure 3.

2. **Experiment 2** In this experiment we build upon the finding in the previous experiments i.e now we have *Batch normalization* and our weights are normally initialized as given in the assignment. In this experiment we simply ran our model upto 3000 iteration instead of 1500 iterations which was default in all our previous implementations. The intuition behind increasing the training iteration comes from the results depicted in Figure 3(b) where accuracy is still increasing. As a result we again got a increase in our accuracy, reaching 0.50 which is again a 2% increase as compared to experiment 1. The plot for loss and accuracy is depicted in Figure 4.



(a) Training Loss (b) Testing Accuracy

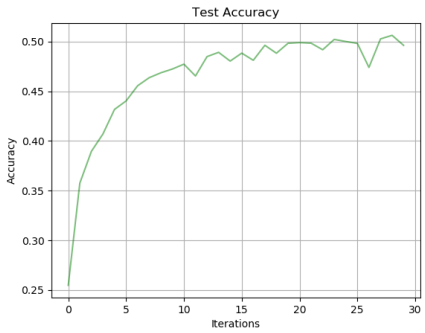
Figure 4: TEST 2: Increasing training iterations

3. **Experiment 3** In this experiment we played around with varying learning rates. Unlike our previous implementations where we used default learning rates.

   Just to mention each of the experiments are build upon the findings of our previous experiments. i.e now we have *Batch normalization*, training iterations now set to 3000 and weight initialization is still default i.e using normal distribution. Well with the introduction of *Batch normalization* it has given us some freedom to increase the learning rate. Because Batch normalization affects the gradient flow itself; it reduces the dependence on the scale of the parameters and the initial values, and it prevents the network from getting stuck in saturated modes caused by certain non-lineararities. It appears that batch normalization acts as a kind of regularize. Exploiting this use fullness we experimented with learning rate 0.01 and 0.1 as compared to our default learning rate which was 0.002. Figure 5(a), 5(b) are corresponding to 0.01 and 5(c), 5(d) are corresponding to 0.1. Accuracy's for both the learning rates is better than previous models, i.e now reaching over 0.50 but model with learning rate of 0.1 perform better getting accuracy over 0.51 Figure 5(c), 5(d).

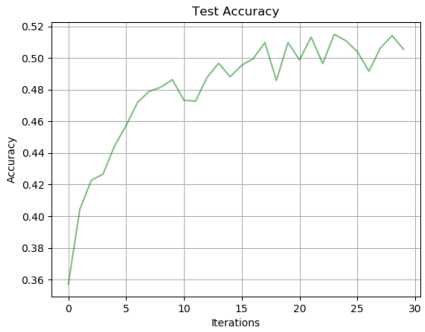
4. **Experiment 4**: Further we try to change our optimizer that is instead of using SGD we are using Adam optimizer with varying learning rates. With Adam optimizer we further experiment with varying learning rates because using the best performing learning rate that we found in our previous experiment i.e 0.1 fails miserably with Adam optimizer Figure 6(a),6(b). From our results as depicted in Figure 6 we see that as we decrease our learning rate we see a boost in our performance. We see in Figure 6(e), 6(f) we perform much better reaching over 0.53 which is baseline to compare with, as given in the assignment. It seems

(a) Training Loss for $lr = 0.01$



(b) Testing Accuracy for $lr = 0.01$

(c) Training Loss for $lr = 0.1$



(d) Testing Accuracy for $lr = 0.1$

Figure 5: TEST 3: Learning Rate

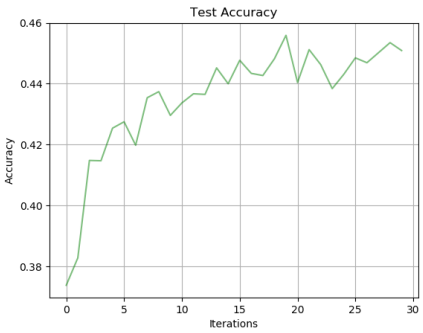
like the default learning rate works best for Adam optimizer achieving better of all the experiments so far.

5. **Experiment 5**. Finally we try to increase the depth in our model. We introduced 3 hidden layers each with 300 units of hidden neurons with $lr = 0.02$, Adam optimizer and also increasing the training iteration upto 6000 steps. With these we attain a testing accuracy of over 0.54 as shown in Figure 7. Well given the models complexity and the improvement it makes is still under performing considering the accuracy versus computation trade off we went through. However, it is still the best performing network [1]. Also there could be many experiments possible, as one could also perform experiment with different learning rate after certain time steps.

In the series of experiments we played with different parameters. Our best performing model which attains a accuracy just over 0.54 is based on 3 hidden layers each with 300 hidden units. We also got a accuracy of just over 0.53 using single hidden layer with 100 hidden units with Adam optimizer as described in earlier experiments.

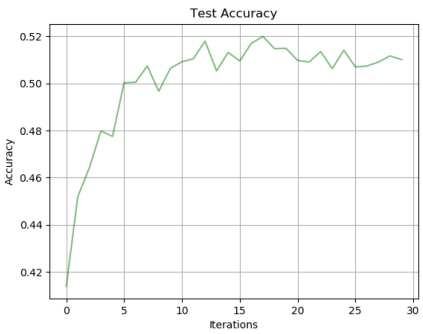
[1] train_mlp_pytorch.py contains the best performing model

(a) Training Loss for $lr = 0.1$

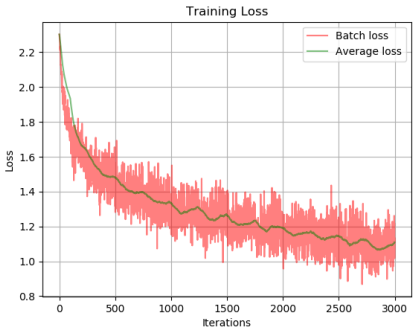


(b) Testing accuracy for $lr = 0.1$

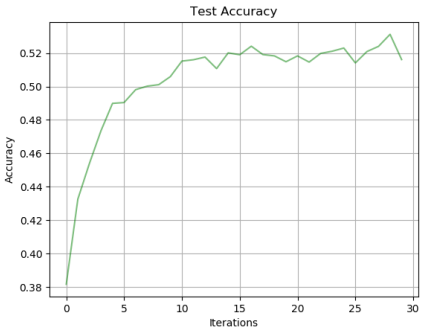
(c) Training Loss for $lr = 0.01$



(d) Testing accuracy for $lr = 0.01$

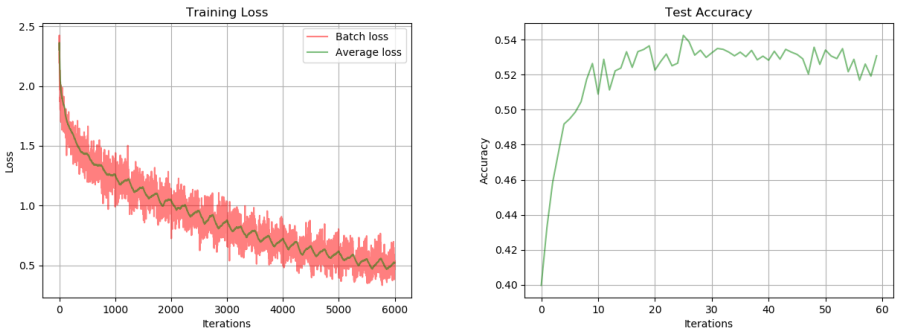


(e) Training Loss for $lr = 0.002$



(f) Testing accuracy for $lr = 0.002$

Figure 6: TEST 4: Optimizer and varying learning rate



(a) Training loss (b) Testing accuracy

Figure 7: TEST 5: Increasing depth of the network

3 Custom Module: Batch Normalization

3.1 Manual implementation of backward pass

3.1.1 Compute the backpropagation equations for the batch normalization operation, that is, compute

$$\mu_i = \frac{1}{B}\sum_{s=1}^{B} x_i^s \tag{1}$$

$$\sigma_i^2 = \frac{1}{B}\sum_{s=1}^{B}(x_i^s - \mu_i)^2 \tag{2}$$

1. $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$

$$\begin{aligned}\frac{\partial L}{\partial x_j^r} &= \sum_s\sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial x_j^r} \\ &= \sum_s\sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \hat{x}_s^i}\frac{\partial \hat{x}_s^i}{\partial x_j^r} \\ &= \sum_s\sum_i \frac{\partial L}{\partial y_i^s}\gamma_i\frac{\partial \hat{x}_s^i}{\partial x_j^r}\end{aligned}$$

We will first derive the partial derivatives for $\frac{\partial \hat{x}_s^i}{\partial x_j^r}$ and will plug the in the above equation.

$$\begin{aligned}\frac{\partial \hat{x}_s^i}{\partial x_j^r} &= \frac{\partial}{\partial x_j^r}\left(\frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2}}\right) \quad \rightarrow \text{removing } \epsilon \text{ to make calculation easy} \\ &= \frac{\left(\delta_{r,s}\delta_{i,j} - \frac{1}{B}\delta_{i,j}\right)\sqrt{\sigma_i^2} - \frac{\partial\sqrt{\sigma_i^2}}{\partial x_j^r}\left(x_i^s - \mu_i\right)}{\sigma_i^2} \\ &= \frac{\left(\delta_{r,s}\delta_{i,j} - \frac{1}{B}\delta_{i,j}\right)\sqrt{\sigma_i^2} - \frac{1}{B}(\sigma_i^{-1/2})(x_i^r - \mu_i)\delta_{i,j}\left(x_i^s - \mu_i\right)}{\sigma_i^2}\end{aligned}$$

Putting this in our main equation.

$$\begin{aligned}\frac{\partial L}{\partial x_j^r} &= \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \gamma_i \frac{\left(\delta_{r,s}\delta_{i,j} - \frac{1}{B}\delta_{i,j}\right)\sqrt{\sigma_i^2} - \frac{1}{B}(\sigma_i^{-1/2})(x_i^r - \mu_i)\delta_{i,j}\left(x_i^s - \mu_i\right)}{\sigma_i^2} \\ &= \frac{\partial L}{\partial y_j^r}\gamma_j \sigma_j^{-1/2} - \frac{1}{B}\gamma_j \sum_s \frac{\partial L}{\partial y_j^s}\sigma_j^{-1/2} - \frac{1}{B}\sigma_j^{-3/2}(x_j^r - \mu_j)\gamma_j \sum_s \frac{\partial L}{\partial y_j^s}(x_j^s - \mu_j) \\ &= \frac{1}{B}\gamma_j(\sigma_j^2 + \epsilon)^{-1/2}\left(B\frac{\partial L}{\partial y_j^r} - \sum_s \frac{\partial L}{\partial y_j^s} - (\sigma_j^2 + \epsilon)^{-1}(x_j^r - \mu_j)\sum_s \frac{\partial L}{\partial y_j^s}(x_j^s - \mu_j)\right)\end{aligned}$$

2. $\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta}$

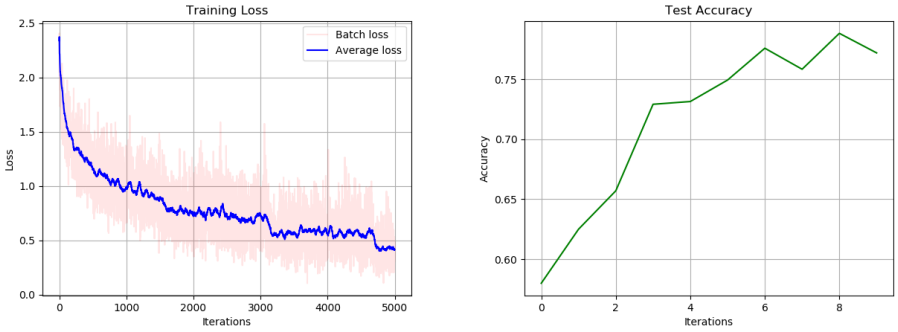
$$\begin{aligned}\frac{\partial L}{\partial \beta_j} &= \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} \\ &= \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s}\delta_{ij} \\ &= \sum_s \frac{\partial \mathcal{L}}{\partial y_j^s}\end{aligned}$$

3. $\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y}\frac{y}{\partial \gamma}$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \gamma_j} &= \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} \\ &= \sum_s \sum_i \frac{\partial \mathcal{L}}{\partial y_i^s}\hat{x}_i^s\delta_{ij} \\ &= \sum_s \frac{\partial \mathcal{L}}{\partial y_j^s}(x_j^s - \mu_j)(\sigma_j^2 + \epsilon)^{-1/2}\end{aligned}$$

4 PyTorch CNN

Undoubtedly, classification based on conv nets outperforms each of the experiment performed so far. This is very much validate by loss and accuracy curves in Figure 8.



(a) Training loss (b) Test Accuracy

Figure 8: Loss and accuracy curve of Conv-net on cifar10.