Assignment 1: MLPs, CNNs and Backpropagation

Akash Raj Komarlu Narendra Gupta
11617586
University of Amsterdam
akashrajkn@gmail.com

Abstract

In this assignment, I explore image classification using Multi layer perceptrons (MLPs). In the first section, the gradients for MLP are derived and a NumPy implementation is tested on the CIFAR-10 dataset. In the second section, a PyTorch version of the same is implemented with different hyper parameter settings. In section 3, a custom Batch normalization module is implemented. Finally, a smaller version of the popular VGG network is explored.

Introduction

In this assignment, for the task of image classification I use the CIFAR-10 dataset[1]. It contains 60000 images each of size $32 * 32$ from 10 classes. The train-test split is $5 : 1$. Figure 1 shows a few images drawn at random from the dataset. There is no overlap between the classes (they are mutually exclusive).

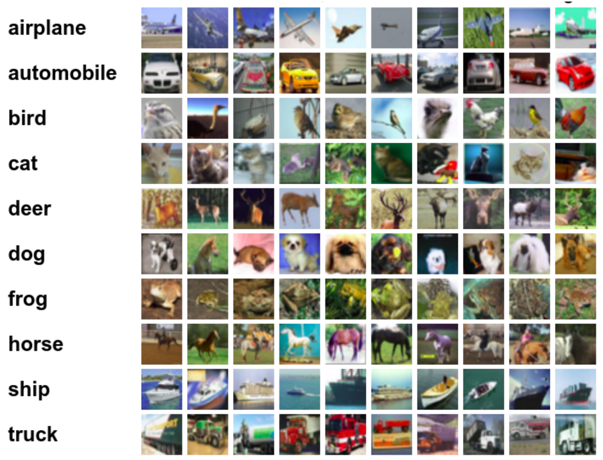


Figure 1: 10 random images in each class from the CIFAR-10 dataset.

1 MLP backprop and NumPy implementation

In this section, I first derive the gradients analytically and use them in deriving the back propagation equations. Then I implement the different modules and MLP in NumPy.

[1] https://www.cs.toronto.edu/~kriz/cifar.html

1.1 Analytical derivation of gradients

1.1.1 Gradient Computation of each module

Gradient of Loss with respect to its input: In matrix notation, this can be written as $\frac{\partial L}{\partial x^{(N)}} = \left(\frac{-t}{x^{(N)}}\right)^T$ (here division is taken as an element-wise operation). The derivation is shown below

$$\begin{aligned} \left(\frac{\partial L}{\partial x^{(N)}}\right)_i &= \frac{\partial L}{\partial x_i^{(N)}} \\ &= \frac{\partial(-\sum_i t_i \log x_i^{(N)})}{\partial x_i^{(N)}} \\ &= \frac{-t_i}{x_i^{(N)}} \end{aligned} \tag{1}$$

Gradient of softmax: In matrix notation, it can be written as $\frac{\partial x^{(N)}}{\partial \widetilde{x}^{(N)}} = \text{diag}(\widetilde{x}) - \widetilde{x}\widetilde{x}^T$, where 'diag' refers to the diagonal elements of the matrix. The component-wise derivation is shown below

$$\begin{aligned} \left(\frac{\partial x^{(N)}}{\partial \widetilde{x}^{(N)}}\right)_{ij} &= \frac{\partial x_i^{(N)}}{\partial \widetilde{x}_j^{(N)}} \\ &= \frac{\partial}{\partial \widetilde{x}_j^{(N)}} \left[\frac{\exp \widetilde{x}_j^{(N)}}{\sum_i \exp \widetilde{x}_i^{(N)}}\right] \\ &= \begin{cases} \frac{\exp \widetilde{x}_j^N \sum_i \exp \widetilde{x}_i^N - \exp \widetilde{x}_j^N \exp {x^N}_j}{\left[\sum_i \exp \widetilde{x}_i^N\right]^2}, & (i = j) \\ \frac{-\exp \widetilde{x}_i^N \exp \widetilde{x}_j^N}{\left[\sum_i \exp \widetilde{x}_i^N\right]^2}, & (i \neq j) \end{cases} \end{aligned} \tag{2}$$

Gradient of ReLU module,

$$\begin{aligned} \frac{\partial x^{(l<N)}}{\partial \widetilde{x}^{(l<N)}} &= \frac{\partial(\max(0, \widetilde{x}^{l<N}))}{\partial \widetilde{x}^{l<N}} \\ &= \begin{cases} 0 & (\widetilde{x}^l \leq 0) \\ 1 & (\widetilde{x}^l > 0) \end{cases} \end{aligned} \tag{3}$$

Gradient of Linear module: Each layer is parameterized by weights, $W^{(l)} \in \mathbb{R}^{d_l * d_{l-1}}$ and biases, $b^{(l)} \in \mathbb{R}^{d_l}$. In matrix form, it can be written as $\frac{\partial \widetilde{x}^l}{\partial \widetilde{x}^{(l-1)}} = W^{(l)}$

$$\begin{aligned} \left(\frac{\partial \widetilde{x}^l}{\partial \widetilde{x}^{(l-1)}}\right)_{ij} &= \frac{\partial \widetilde{x}_i^l}{\partial x_j^{(l-1)}} \\ &= \frac{\partial \sum_{k=1}^{l-1} w_{ik} x_k}{\partial x_j^{(l-1)}} \\ &= w_{ij} \\ \left(\frac{\partial \widetilde{x}^l}{\partial w^l}\right)_{ijk} &= \frac{\partial \widetilde{x}_i^l}{\partial w_{jk}^l} \\ &= \frac{\partial}{\partial w_{jk}^l} \left[\sum_{k=1}^{l-1} w_{ik} x_k + b_i\right] = x_k \\ \left(\frac{\partial \widetilde{x}^l}{\partial b^l}\right)_{ij} &= \frac{\partial \widetilde{x}_i^l}{\partial b_j^l} = 1 \end{aligned} \tag{4}$$

1.1.2 Gradients in backpropagation equations

Gradient of loss with respect to the softmax layer:

$$\begin{aligned} \frac{\partial L}{\partial \widetilde{x}^{(N)}} &= \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \widetilde{x}^{(N)}} \\ &= \frac{\partial L}{\partial x^{(N)}} \left[\text{diag}(\widetilde{x} - \widetilde{x}\widetilde{x}^{T}) \right] \end{aligned} \tag{5}$$

where, $\frac{\partial L}{\partial x^{(N)}} = \left(\frac{-t}{x}\right)^{T}$ (here division is taken as an element-wise operation).
Gradient of loss with respect to the ReLU module:

$$\begin{aligned} \frac{\partial L}{\partial \widetilde{x}^{(l<N)}} &= \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \widetilde{x}^{(l)}} \\ &= \begin{cases} \frac{\partial L}{\partial x^{(l)}} & , x^{(l)} > 0 \\ 0 & , x^{(l)} \leq 0 \end{cases} \end{aligned} \tag{6}$$

Gradient of loss with respect to the network parameters,

$$\begin{aligned} \frac{\partial L}{\partial x^{(l<N)}} &= \frac{\partial L}{\partial \widetilde{x}^{(l+1)}} \frac{\partial \widetilde{x}^{(l+1)}}{\partial x^{(l)}} \\ &= \frac{\partial L}{\partial x^{(l+1)}} W^{(l)T} \\ \frac{\partial L}{\partial W^{(l)}} &= \frac{\partial L}{\partial \widetilde{x}^{(l)}} \frac{\partial \widetilde{x}^{(l)}}{\partial W^{(l)}} \\ &= \frac{\partial L}{\partial \widetilde{x}^{(l)}} x^{(l-1)} \\ \frac{\partial L}{\partial b^{(l)}} &= \frac{\partial L}{\partial \widetilde{x}^{(l)}} \frac{\partial \widetilde{x}^{(l)}}{\partial b^{(l)}} \\ &= \frac{\partial L}{\partial \widetilde{x}^{(l)}} \end{aligned} \tag{7}$$

1.1.3 Batch size $\neq 1$

For batch size B, total loss can be written as,

$$L_{\text{total}} \left(\left\{ x^{(0),s} t^{s} \right\}_{s=1}^{B} \right) = \frac{1}{B} \sum_{s=1}^{B} L_{\text{individual}} \left(x^{(0),s} t^{s} \right) \tag{8}$$

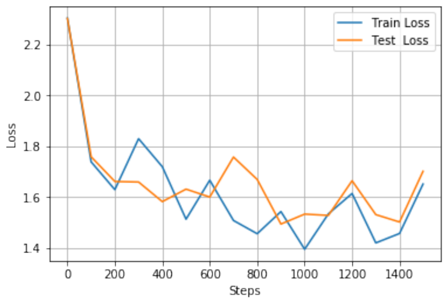
For the backpropagation equations, the total gradient would be the mean of individual gradients. We don't need to re-derive the backpropagation equations for mini-batches.

1.2 NumPy implementation

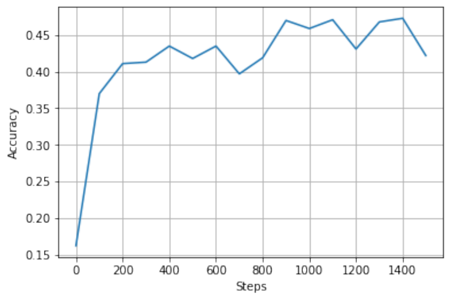
Using the above backpropagation equations, the corresponding NumPy modules were implemented. Using the default hyper-parameter settings (mentioned in Table 1), 42.2% accuracy can be achieved. Figure 2 shows the accuracy and loss curves over training steps. We can observe that though the loss (Cross Entropy) decreases, it has not stabilized and therefore requires a larger number of training steps.

2 PyTorch MLP

In this section, I implement a Multi layered perceptron in PyTorch. As a baseline, I use parameter values as mentioned in Table 1. The MLP has one hidden layer with 100 hidden units. The accuracy of the baseline model on `CIFAR-10` dataset is 43.2%. Figure 3 shows the accuracy and loss curves for the baseline model. We can observe that the loss (Cross Entropy) decreases rapidly within the first 200 steps of training.

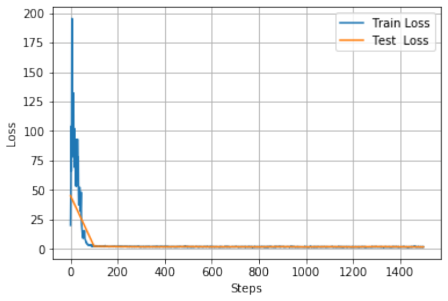


(a)

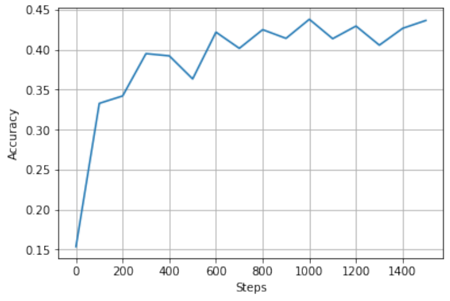


(b)

Figure 2: Figures show accuracies and loss curves for MLP implemented in NumPy.



(a)



(b)

Figure 3: The left figure shows the train and test losses at different steps for the baseline model. The right figure shows the accuracy on the test set for different steps.

The accuracy of the MLP greatly depends on the choice of hyper-parameter values. I performed several experiments to explore the effects of different parameter settings. In this section, I present a subset of the (parameter grid-search) experiments (appendix contains accuracies obtained for all the experiments). Table 2 shows the different values of hyper parameters considered during grid-search.

Hidden layers	**Learning rate**	**Activation**	**Steps**	**Batch size**	**SGD**	
					momentum	**weight decay**
[100]	0.002	ReLU	1500	200	0	0

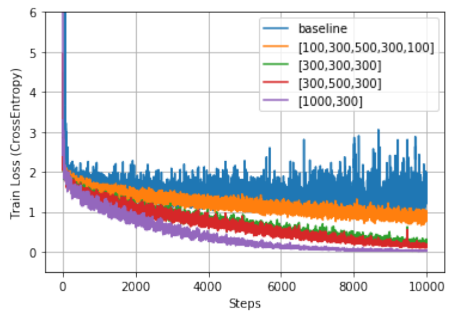
Table 1: Baseline parameter values

Parameter	**Values**
Architecture	$[100], [1000, 300], [300, 500, 300], [500, 500, 500], [100, 300, 500, 300, 100]$
Momentum	$0, 0.9$
Weight decay	$0, 0.005$
Batch size	$50, 100, 500$
Learning rate	$0.001, 0.002, 0.1$

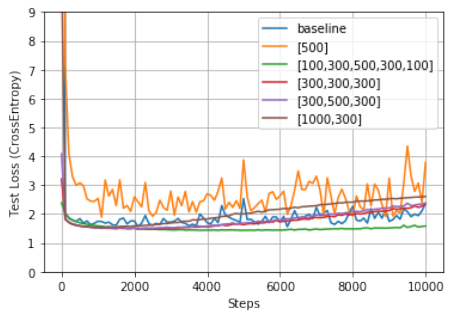
Table 2: Hyper-parameter settings considered

2.1 Hidden layers

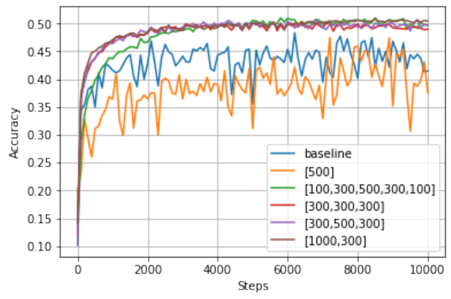
Different architectures (hidden layers) have been experimented with (The input is of dimension $32 * 32 * 3$). For the different hidden layers mentioned in Table 2, we plot the accuracy and loss



(a)



(b)



(c)

Figure 4: Effect of **Architecture** on accuracy and loss. Plots a) and b) show the train and test loss curves respectively. Plot c) shows the accuracy with respect to steps.

curves. Figure 4 shows the curves of interest when different architectures are used to train a MLP. We can see that by just increasing the hidden nodes without a change in the number of layers does not guarantee a better accuracy (100 hidden nodes does better than 500 hidden nodes). We can also observe that by increasing the number of layers, in general the accuracy increases which could be attributed to the network learning more complex features from the data. But introducing too many layers could be detrimental as seen in the case of $[100, 300, 500, 300, 100]$: a 5 layered MLP.

2.2 Weight Regularization

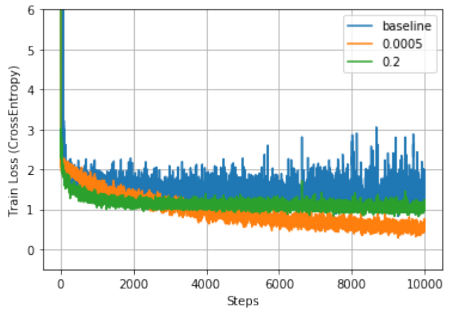
Vanilla Deep neural networks are prone to over-fitting (especially when data is scarce). This could potentially lead to over-confident wrong predictions. Here, to combat this problem, we introduce regularization in the form of weight decay (norm of the model params are optimized along with the objective function). I test with three settings of weight decay: {0, 0.0005, 0.2}. From Figure 5 we can observe that a weight decay factor of 0.2 helps in stabilizing the test loss and it reaches a high level of accuracy faster.

2.3 Batch Size

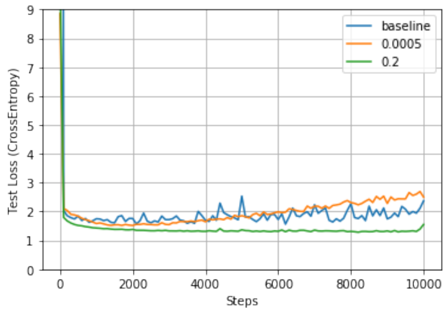
Smaller batch sizes tend to give only a rough approximation of the gradients compared to a larger batch size. The immediate effect of using a small batch size is that it takes a longer time to find a good solution. To test this, I experiment with 3 batch sizes: {50, 100, 500}. From the Figure 6 we can observe that train loss for the batch size of 50 takes very long to stabilize. We can also notice that using a larger batch size resulted in better accuracy. However more experiments need to be conducted to get a better idea about the effect of different batch sizes.

2.4 Momentum

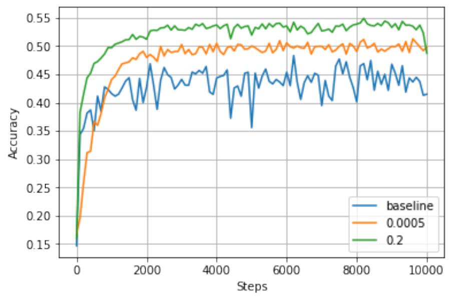
Momentum is a simple addition to the SGD algorithm that generally performs better than the vanilla-SGD. The addition of momentum results in a faster convergence by accelerating the gradients in the



(a)

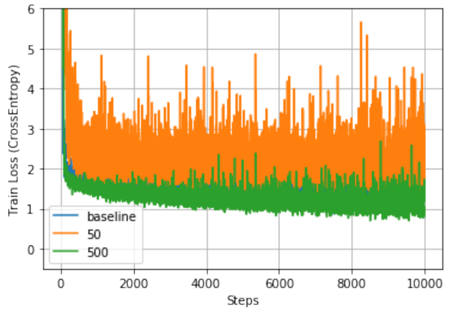


(b)

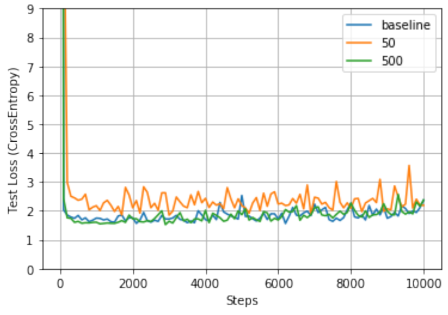


(c)

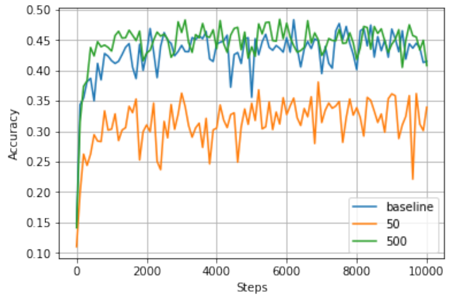
Figure 5: Effect of **Weight decay** on accuracy and loss. Plots a) and b) show the train and test loss curves respectively. Plot c) shows the accuracy with respect to steps.



(a)



(b)



(c)

Figure 6: Effect of **Weight decay** on accuracy and loss. Plots a) and b) show the train and test loss curves respectively. Plot c) shows the accuracy with respect to steps.

right direction. I test with 2 values of momentum: {0, 0.9}. The addition of momentum improved the convergence time for the best model.

2.5 Best Model

Among the tested models, the best performing model parameter setting is shown in Table 3. It gives an accuracy of 56.37%.

Hidden layers	Learning rate	Activation	Steps	Batch size	SGD	
					momentum	**weight decay**
[1000, 300]	0.002	ReLU	50000	200	0.9	0.2

Table 3: Best model parameter values

3 Custom Module: Batch Normalization

The backpropagation equations have been derived below[2],

$$\begin{aligned}\frac{\partial L}{\partial \gamma} &= \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma} \\ &= \sum_{i=1}^{B} \frac{\partial L}{\partial y_i^s}\hat{x}_i^s \\ \frac{\partial L}{\partial \beta} &= \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta} \\ &= \sum_{i=1}^{B} \frac{\partial L}{\partial y_i^s} \\ \frac{\partial L}{\partial \hat{x}_i^s} &= \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \hat{x}_i^s} \\ &= \frac{\partial L}{\partial y_i^s}\gamma\end{aligned} \tag{9}$$

To compute the gradient of L with respect to x_i^s,

$$\frac{\partial L}{\partial x_i^s} = \frac{\partial L}{\partial \hat{x}_i^s}\frac{\partial \hat{x}_i^s}{\partial x_i} + \frac{\partial L}{\partial \hat{x}_i^s}\frac{\partial \hat{x}_i^s}{\partial \mu_i} + \frac{\partial L}{\partial \sigma_i^2}\frac{\partial \sigma_i^2}{\partial \mu_i} \tag{10}$$

We know that $\hat{x}_i^s = \frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$. Therefore,

$$\frac{\partial \hat{x}_i^s}{\partial \mu_i} = \frac{-1}{\sqrt{\sigma_i^2 + \epsilon}} \tag{11}$$

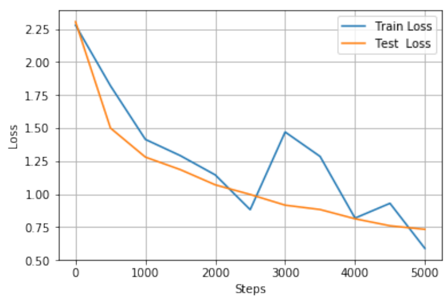
We also know that $\sigma_i^2 = \frac{1}{B}\sum_{s=1}^{B}(x_i^s - \mu_i)^2$. Using this,

$$\frac{\partial \sigma_i^2}{\partial \mu_i} = \frac{-2}{B}\sum_{s=1}^{B}(x_i^s - \mu_i) \tag{12}$$

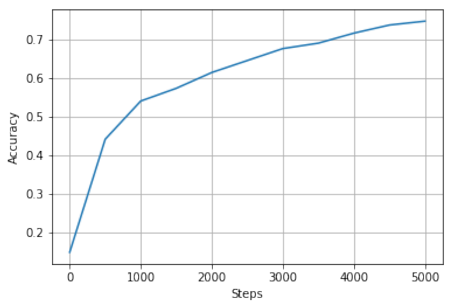
Rearranging, we have $\hat{x}_i^s = \frac{(x_i^s - \mu_i)}{\sqrt{\sigma_i^2 + \epsilon}}$.

$$\frac{\partial \hat{x}_i^s}{\partial \sigma_i^2} = \frac{-1}{2}\sum_{s=1}^{B}(x_i^s - \mu_i)\sqrt{(\sigma_i^2 + \epsilon)^{-1}} \tag{13}$$

[2]Note: I took the help of this resource: `https://kevinzakka.github.io/2016/09/14/batch_normalization/`



(a)



(b)

Figure 7: The left figure shows the train and test losses at different steps for the CNN model. The right figure shows the accuracy on the test set for different steps.

Using the above results and substituting into equation 10, we get,

$$\frac{\partial L}{\partial x_i} = \frac{1}{B}\left[\frac{\partial L}{\partial \hat{x}_i^s}\frac{B}{\sqrt{\sigma_i^s+\epsilon}} + \frac{\partial L}{\partial \mu_i} + (x_i^s-\mu_i)\frac{\partial L}{\partial \sigma_i^2}\right] \tag{14}$$

4 PyTorch CNN

Finally, a small version of the popular VGG network is implemented in PyTorch. A Convolutional neural network (CNN) is better suited for image classification because of the convolutional filters. Adam optimizer with default learning rate of 0.0001 is used and the batch size is 32. With these parameters, the CNN gives an accuracy of 74.8%. Figure!7 shows the accuracy and loss curves. Due to time constraints, the grid search over various parameter settings could not be performed.

5 Conclusion

In this assignment, we explored two architectures for the image classification task: CNN and MLP. The back propagation equations for an MLP were derived and a custom module was implemented in NumPy. Effect of different hyper-parameters were studied with a PyTorch version of the same. While CNN performs quite well in this task, an MLP struggles to achieve such an accuracy. A possible explanation is due to the convolutional filters which help in learning spatial information.