Deep Learning Practical 1

Andrei Marius Sili[*]
Master's in Artificial Intelligence
University van Amsterdam
andrei.sili@student.uva.nl

1 MLP Back Prop and NumPy implementation

1.1 Analytic derivation of gradients

1.1.1 Part a)

In this part I use $\odot$ to denote element-wise multiplication and $diag(v)$ to denote the matrix that is composed by the elements of v on the diagonal and zeros elsewhere. The indicator function is denoted by $\mathbb{1}(expr)$ and is applied element-wise for Boolean expression involving a vector.

[*]Student ID: 11659416

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

$$
\begin{aligned}
\left(\frac{\partial L}{\partial \mathbf{x}^{(N)}}\right)_i &= \frac{\partial L}{\partial x_i^{(N)}} \\
&= \frac{\partial}{\partial x_i}\left(-\sum_j t_j \log x_j\right) \\
&= -\frac{t_i}{x_i} \\
&= (\mathbf{t} \odot \mathbf{r})_i^T \text{ where } r_i = \frac{1}{x_i} \\
\left(\frac{\partial \mathbf{x}^{(N)}}{\partial \tilde{\mathbf{x}}^{(N)}}\right)_{ij} &= \frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} \\
&= \frac{\partial}{\partial \tilde{x}_j^{(N)}}\left(\frac{e^{\tilde{x}_i^{(N)}}}{\sum_k e^{\tilde{x}_k^{(N)}}}\right) \\
&= \frac{\delta_{ij} e^{\tilde{x}_i^{(N)}} \Sigma - e^{\tilde{x}_i^{(N)}} e^{\tilde{x}_j^{(N)}}}{\Sigma^2} \text{ where } \Sigma = \sum_k e^{\tilde{x}_k^{(N)}} \\
&= x_i^{(N)}\left(\delta_{ij} - x_j^{(N)}\right) \\
&= \left(diag\left(\mathbf{x^{(N)}}\right) - \mathbf{x^{(N)}}\mathbf{x^{(N)}}^T\right)_{ij} \\
\left(\frac{\partial \mathbf{x}^{(l)}}{\partial \tilde{\mathbf{x}}^{(l)}}\right)_{ij} &= \frac{\partial x_i^{(l)}}{\partial \tilde{x}_j^{(l)}} \\
&= \mathbb{1}(i=j)\,\mathbb{1}(\tilde{x}_j > 0) \\
&= (diag(\mathbb{1}(\tilde{\mathbf{x}} > 0)))_{ij} \text{ where } \mathbb{1} \text{ is the indicator function} \\
\left(\frac{\partial \mathbf{x}^{(l)}}{\partial \tilde{\mathbf{b}}^{(l)}}\right)_{ij} &= \frac{\partial x_i^{(l)}}{\partial b_j^{(l)}} \\
&= \delta_{ij} = (\mathbf{I})_{ij} \\
\left(\frac{\partial \tilde{\mathbf{x}}^{(l)}}{\partial \mathbf{x}^{(l-1)}}\right)_{ij} &= \frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} \\
&= \frac{\partial}{\partial x_j^{(l-1)}}\left(\sum_n w_{in}^{(l)} x_n^{(l-1)} + b_n^{(l)}\right) = w_{ij}^{(l)} \\
&= \left(\mathbf{W}^{(l)}\right)_{ij} \\
\left(\frac{\partial \tilde{\mathbf{x}}^{(l)}}{\partial \mathbf{W}^{(l)}}\right)_{ijk} &= \frac{\partial \tilde{\mathbf{x}}_i{}^{(l)}}{\partial \mathbf{W}_{jk}^{(l)}} \\
&= \mathbb{1}(1=j)\,\mathbb{1}(n=k)\,x_n^{l-1} \\
&= (\mathbf{P})_{ijk}
\end{aligned}
$$

1.1.2 Part b)

$$
\begin{aligned}
\frac{\partial L}{\partial \tilde{\mathbf{x}}^{(N)}} &= \frac{\partial L}{\partial \mathbf{x}^{(N)}} \frac{\partial \mathbf{x}^{(N)}}{\partial \tilde{\mathbf{x}}^{(N)}} \\
&= \frac{\partial L}{\partial \mathbf{x}^{(N)}} \left(diag\left(\mathbf{x^{(N)}} \right) - \mathbf{x^{(N)}} \mathbf{x^{(N)}}^T \right) \\
\frac{\partial L}{\partial \tilde{\mathbf{x}}}^{(l)} &= \frac{\partial L}{\partial \mathbf{x}^{(l)}} \frac{\partial \mathbf{x}^{(l)}}{\partial \tilde{\mathbf{x}}^{(l)}} \\
&= \frac{\partial L}{\partial \mathbf{x}^{(l)}} \left(diag\left(\mathbb{1}\left(\tilde{\mathbf{x}^{(l)}} > 0 \right) \right) \right) \\
&= \frac{\partial L}{\partial \mathbf{x}^{(l)}} \odot \mathbb{1}\left(\tilde{\mathbf{x}}^{(l)} > 0 \right) \\
\frac{\partial L}{\partial \mathbf{x}}^{(l)} &= \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l+1)}} \frac{\partial \tilde{\mathbf{x}}^{(l+1)}}{\partial \mathbf{x}^{(l)}} \\
&= \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l+1)}} \mathbf{W}^{(l)} \\
\frac{\partial L}{\partial \mathbf{b}}^{(l)} &= \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l)}} \frac{\partial \tilde{\mathbf{x}}^{(l)}}{\partial \mathbf{b}^{(l)}} \\
&= \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l)}} \mathbf{I} = \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l)}} \\
\frac{\partial L}{\partial \mathbf{W}^{(l)}} &= \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l)}} \frac{\partial \tilde{\mathbf{x}}^{(l)}}{\partial \mathbf{W}^{(l)}} \\
&= \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l)}} \mathbf{P} = \frac{\partial L}{\partial \tilde{\mathbf{x}}^{(l)}} \mathbf{x}^T
\end{aligned}
$$

Please note, for the last derivative expression

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}}$$

The simplification is possible because of the following:

$$
\begin{aligned}
\left(\frac{\partial L}{\partial \mathbf{W}^{(l)}} \right)_{jk} &= \sum_i \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \frac{\partial \tilde{x}_i^{(l)}}{\partial w_{jk}^{(l)}} \\
&= \sum_i \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \delta_{ij} \tilde{x}_k \\
&= \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \tilde{x}_k
\end{aligned}
$$

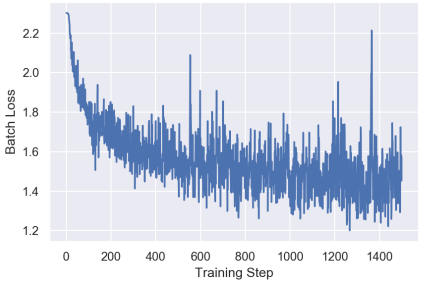
1.1.3 Part c)

If a batch size different than 1 is used, the following will happen:

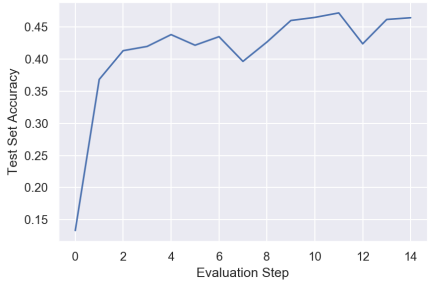
- Derivatives with respect to inputs, $\mathbf{X}, \tilde{\mathbf{X}}$, will have an extra dimension, the batch dimension.
- The gradients with respect to the model parameters, $\mathbf{W}^{(l)}$, $\mathbf{b}^{(l)}$, will be summed across the batch dimension when updating weights.
- The variance of the gradient will be reduced, as the size of the batch increases (unrelated to the equations, per say).

1.2 NumPy implementation

Figures 1a and 1b show the loss and accuracy curves. Note that the loss is computed on the training set per batch at each iteration, while the accuracy is computed at a default step interval of 100 on the test set.



(a) Training Set Batch Loss. *Training steps are on the horizontal axis, and the average cross entropy loss over a batch is represented on the vertical axis. The downward trend is indicates that the network is learning.*



(b) Test Set Full Accuracy. *Evaluation steps are on the horizontal axis, and the accuracy over the entire test set is represented on the vertical axis. The increasing performance shows that our surrogate objective (the loss) is a good approximation of the task and that learning is taking place.*

2 PyTorch MLP

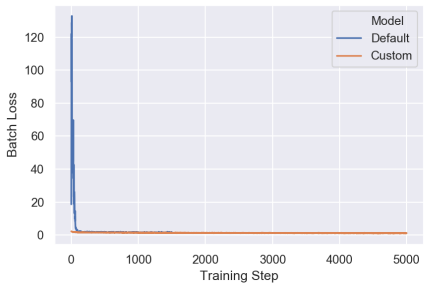
The baseline PyTorch implementation obtains similar results to the NumPy implementation, with the same hyper-parameters(depth, width, choice of optimiser, etc.) with a final test set accuracy of around 0.43. I do observe, however, a great spike in the initial loss values using SGD which I cannot explain. To improve performance I have taken a sequential approach.

Firstly, I used Adam optimiser with default parameters. This circumvents the shortcomings of SGD, particularly in areas of the search space that are shaped like ravines. This is due to a combination of using momentum to boost small gradients and and simulated annealing to reduce large noisy gradients. This resulted in a slight increase to the overall test set performance to around 0.46.

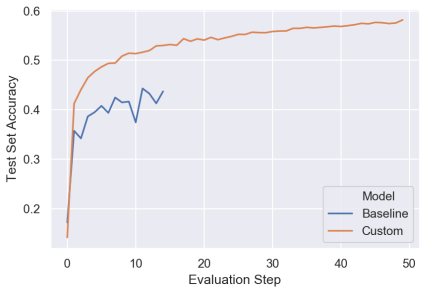
Secondly, I added Batch Normalisation on the outputs of each linear layer in order to reduce internal co-variate shift and allow the network to more easily adapt its weights by modifying the affine transformation parameters. This stabilised and boosted learning substantially, with the first evaluation already reaching an accuracy of 0.41 and levelling around 0.53 after 15 evaluation steps.

Thirdly, I added dropout layers on on inputs to each linear layer. This serves as a regularisation technique, since half of the inputs are set to zero, encouraging the network to distribute reliance equally among layer parameters. In effect, this ensures that the network does not become overly reliant on some features and ignorant of others, leading to less overfitting on the training set. This also improved accuracy by about 2 percentage points.

Finally, as I noticed that in all of these experiments, the accuracy was showing an increasing trend still, and was not plateauing, I decided to increase the training steps to 5000. I also increased the network depth and width, with 3 linear layers as opposed to just 1, and with 1024, 512, and 256 neurons respectively. The choice for this architecture was motivated by the intuition that the network would be able to first learn lower-level features in the first layer that it can then combine into more robust and abstract representation in the following layers. The results were positive, peaking at around 0.58 in the final evaluation step and still showing an increasing trend. Figures 2a and 2b show the the loss and accuracy curves, compared to the baseline model.



(a) Training Batch Loss. *As can be seen, training is much more stable, with the loss curve seeming almost flat compared to the spiky loss curve of the baseline model.*



(b) Test Full Accuracy. *Using Batch Normalisation, Adam, Dropout and a bigger network results in dramatic improvements both in overall performance and the speed of convergence.*

Figure 2: Optimised MLP Training Loss and Test Accuracy

3 Custom Module: Batch Normalisation

3.1 Automatic differentiation

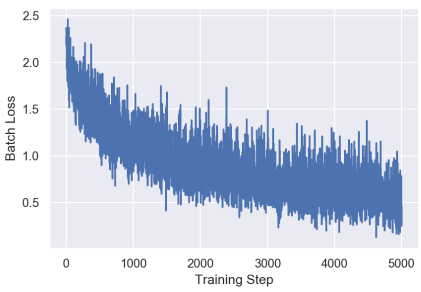
3.2 Manual implementation of backward pass

3.2.1 Part a)

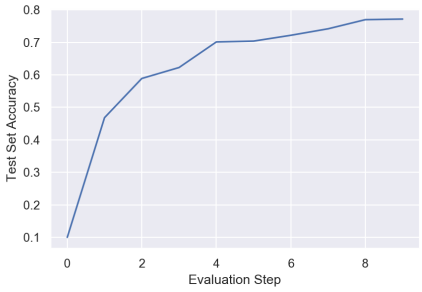
$$
\begin{aligned}
\left(\frac{\partial L}{\partial \boldsymbol{\gamma}}\right)_j &= \sum_s \sum_i \frac{\partial L}{\partial y_i^{(s)}} \frac{\partial y_i^{(s)}}{\partial \gamma_j} \\
&= \sum_s \frac{\partial L}{\partial y_i^{(s)}} \frac{\partial y_j^{(s)}}{\partial \gamma_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^{(s)}} \hat{x}_j^{(s)} \\
\left(\frac{\partial L}{\partial \boldsymbol{\beta}}\right)_j &= \sum_s \sum_i \frac{\partial L}{\partial y_i^{(s)}} \frac{\partial y_i^{(s)}}{\partial \beta_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^{(s)}} \\
\left(\frac{\partial L}{\partial \hat{\mathbf{x}}}\right)_j &= \sum_s \sum_i \frac{\partial L}{\partial y_i^{(s)}} \frac{\partial y_i^{(s)}}{\partial \hat{x}_j^{(s)}} \\
\left(\frac{\partial L}{\partial \mathbf{x}}\right)_j^r &= \sum_s \sum_i \frac{\partial L}{\partial y_i^{(s)}} \frac{\partial y_i^{(s)}}{\partial x_j^{(r)}} \\
&= \sum_s \sum_i \frac{\partial L}{\partial y_i^{(s)}} \left(\frac{\partial y_i^{(s)}}{\partial \hat{x}_j^{(r)}} \frac{\partial \hat{x}_j^{(s)}}{\partial x_j^{(r)}} + \frac{\partial y_i^{(s)}}{\partial \mu_j} \frac{\partial \mu_j}{\partial x_j^{(r)}} + \frac{\partial y_i^{(s)}}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{\partial x_j^{(r)}} \right) \\
&= \frac{B \frac{\partial L}{\partial \hat{x}_j^{(r)}} - \sum_s \frac{\partial L}{\partial \hat{x}_j^{(s)}} - \hat{x}_j^{(r)} \sum_s \frac{\partial L}{\partial \hat{x}^{(s)}{}_j} \hat{x}_j^{(s)}}{B\sqrt{\sigma^2 + \epsilon}}
\end{aligned}
$$

4 PyTorch CNN

Indeed, using the VGG network dramatically increased performance. Figures 3a 3b show the loss and accuracy curves as always.



(a) Training Batch Loss



(b) Test Full Accuracy

Figure 3: Optimised MLP Training Loss and Test Accuracy. *Performance improves much faster and reaches much higher peaks as opposed to the best version of MLP. Test Accuracy plateaued at around* 0.78.