Deep Learning - Practical 1

Ard Snijders
University of Amsterdam
12854913

1 MLP backprop and NumPy Implementation

1.1 a) Linear Module

Derivative of loss w.r.t. weights:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{W}}$$

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{W}} \rightarrow \frac{\partial Y_{mn}}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \sum_k X_{mk} W_{kn}^T + B_{mn}$$

$$= \sum_k X_{mk} \frac{\partial W_{nk}}{\partial W_{ij}} + B_{mn} = \sum_k X_{mk} \delta_{ni} \delta_{kj} = X_{mj} \delta_{ni}$$

$$\frac{\partial L}{\partial W_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} X_{mj} \delta_{ni} = \sum_m \frac{\partial L}{\partial Y_{mi}} X_{mj} = \sum_m \left[\frac{\partial L}{\partial \mathbf{Y}} \right]_{mi} X_{mj}$$

$$= \sum_m \left[\left(\frac{\partial L}{\partial \mathbf{Y}} \right)^T \right]_{im} X_{mj} = \left[\left(\frac{\partial L}{\partial \mathbf{Y}} \right)^T \mathbf{X} \right]_{ij} \rightarrow \frac{\partial L}{\partial \mathbf{W}} = \left(\frac{\partial L}{\partial \mathbf{Y}} \right)^T \mathbf{X}$$

Derivative of loss w.r.t bias:

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{b}}$$

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{b}} \rightarrow \frac{\partial Y_{mn}}{\partial b_i} = \frac{\partial}{\partial b_i} \sum_k X_{mk} W_{kn}^T + B_{mn}$$

Since $B_{mn} = b_n$, we can substitute b_n for B_{mn} in the expression above. Evaluating the derivative w.r.t. b_i then yields:

$$\frac{\partial b_n}{\partial b_i} = \delta_{ni}$$

$$\frac{\partial L}{\partial b_i} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial b_i} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \delta_{ni} = \sum_m \frac{\partial L}{\partial Y_{mi}} \cdot 1 = \sum_m \frac{\partial L}{\partial Y_{mi}} [\mathbf{1}]_i = \left[\frac{\partial L}{\partial \mathbf{Y}} \mathbf{1} \right]_i$$

$$\frac{\partial L}{\partial \mathbf{b}} = \sum_i \frac{\partial L}{\partial b_i} = \sum_i \left[\frac{\partial L}{\partial \mathbf{Y}} \mathbf{1} \right]_i = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{1}, \text{ where } \mathbf{1} \text{ represents the ones vector.}$$

Derivative of loss w.r.t inputs

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}}\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$$

$$\frac{\partial Y_{mn}}{\partial X_{ij}} = \frac{\partial}{\partial X_{ij}}\sum_k X_{mk}W^T_{kn} + B_{mn}$$

$$= \sum_k \frac{\partial X_{mk}}{\partial X_{ij}}W^T_{kn} + B_{mn} = \sum_k \delta_{mi}\delta_{kj}W^T_{kn} = \delta_{mi}W^T_{jn}$$

$$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n}\frac{\partial L}{\partial Y_{mn}}\delta_{mi}W^T_{jn} = \sum_n \frac{\partial L}{\partial Y_{in}}W^T_{jn} = \sum_n \frac{\partial L}{\partial Y_{in}}W_{nj} = \left[\frac{\partial L}{\partial \mathbf{Y}}\mathbf{W}\right]_{ij}$$

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}}\frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}}\mathbf{W}$$

1.1 b) Activation Module

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial h(\mathbf{X})}\frac{\partial h(\mathbf{X})}{\partial \mathbf{X}}$$

$$\frac{\partial h(\mathbf{X}_{mn})}{\partial \mathbf{X}_{ij}} = \frac{\partial}{\partial \mathbf{X}_{ij}}h'(\mathbf{X}_{mn})\mathbf{X}_{mn} = \frac{\partial \mathbf{X}_{mn}}{\partial \mathbf{X}_{ij}}h'(\mathbf{X}_{mn}) = \delta_{mi}\delta_{nj}h'(\mathbf{X}_{mn})$$

$$\frac{\partial L}{\partial X_{ij}} = \sum_{m,n}\frac{\partial L}{\partial Y_{mn}}\frac{\partial Y_{mn}}{\partial X_{ij}} = \sum_{m,n}\frac{\partial L}{\partial Y_{mn}}\delta_{mi}\delta_{nj}h'(\mathbf{X}_{mn}) = \frac{\partial L}{\partial Y_{ij}}h'(\mathbf{X}_{ij}) = \left[\frac{\partial L}{\partial \mathbf{Y}}h'(\mathbf{X})\right]_{ij}$$

$$\frac{\partial L}{\partial \mathbf{X}} = \left[\frac{\partial L}{\partial \mathbf{Y}} \circ h'(\mathbf{X})\right]$$

1.1 1.1 c) i - Softmax Module

$$\frac{\partial L}{\partial \mathbf{X}} \rightarrow \left[\frac{\partial L}{\partial \mathbf{X}}\right]_{ij} = \frac{\partial L}{\partial X_{ij}} = \sum_{m,n}\frac{\partial L}{\partial Y_{mn}}\frac{\partial Y_{mn}}{\partial X_{ij}}.$$

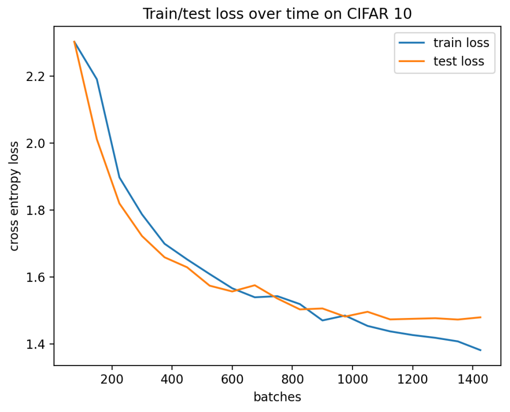
I'm quite terrible at derivatives so instead of deriving everything from scratch, I just took the "common" softmax derivative; $\sigma(1-\sigma)$. I then chose to re-cast that into an expression that accounts for the batch dimension by adding appropriate Kronecker deltas:

$$\frac{\partial Y_{mn}}{\partial X_{ij}} = \delta_{mi}\sigma(X)_{mn}(\delta_{nj} - \sigma(X)_{ij})$$

Since we're applying the softmax over a batch of datapoints, the first Kronecker delta δ_{mi} ensures that partial derivatives of output elements from datapoint m w.r.t. input elements from datapoint i, where $i \neq m$, will always evaluate to zero (as the elements of the input vector of one datapoint shouldn't affect the behaviour of the softmax output of another datapoint, or vice versa). Conversely, if $m = i$, we're considering the pre- and post-softmax row-vectors of the same datapoint, in which case computing the gradient does make sense.

Provided that we're then considering the output and input of the same datapoint (i.e. $m = i$; $\delta_{mi} = 1$), the form of the gradient of the softmax itself differs depending on whether $n = j$ or $n \neq j$:

$$\left[\frac{\partial \mathbf{Y}_{mn}}{\partial \mathbf{X}_{ij}}\right]_{n=j} = \sigma(X)_{mn}(1 - \sigma(X)_{ij}), \quad \left[\frac{\partial \mathbf{Y}_{mn}}{\partial \mathbf{X}_{ij}}\right]_{n\neq j} = \sigma(X)_{mn}(-\sigma(X)_{ij}).$$



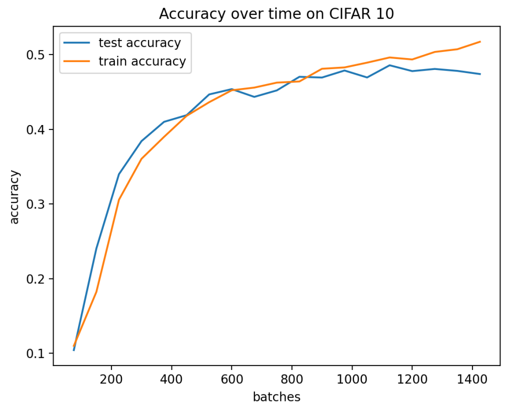


Figure 1: Left: loss for NumPy MLP on CIFAR-10 on training set and development set over time. Right: accuracy on the development set on CIFAR-10 over time. Accuracy on test set after 1400 batches: **0.4967**.

The second Kronecker delta δ_{nj}ensures that the appropriate form is evaluated for the given indices nj. This then yields the following single element derivative:

$$\sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \delta_{mi} \sigma(X)_{mn} (\delta_{nj} - \sigma(X)_{ij}) = \frac{\partial L}{\partial Y_{ij}} \sigma(X)_{ij} (1 - \sigma(X)_{ij})$$

ii - Categorical cross entropy loss module The categorical cross entropy loss can be expressed as

$$L = \frac{1}{S} \sum_i L_i = -\frac{1}{S} \sum_{ik} T_{ik} \log X_{ik}.$$

The derivative of the loss w.r.t. the ij-th element of $\mathbf{X}$ can be expressed as

$$\left[\frac{\partial L}{\partial \mathbf{X}}\right]_{ij} = -\frac{\partial}{\partial X_{ij}} \frac{1}{S} T_{ij} \log(X_{ij}) = -\frac{1}{S} \frac{T_{ij}}{X_{ij}}$$

$$\frac{\partial L}{\partial \mathbf{X}} = \sum_{i,j} \left[\frac{\partial L}{\partial \mathbf{X}}\right]_{ij} = \sum_{i,j} (-\frac{1}{S} \frac{T_{ij}}{X_{ij}}) = -\frac{1}{S} (\mathbf{T} \oslash \mathbf{X})$$

Where $\oslash$ denotes a Hadamard division. The gradient can therefore be computed by first performing element-wise division of $\mathbf{T}$ by $\mathbf{X}$, and then performing another element-wise division by S on the resulting matrix (Regarding the product with $-\frac{1}{S}$, I'm not sure about the notation for element-wise matrix-by-scalar-division, but I hope it's clear enough.)

1.2) NumPy implementation

Loss and accuracy curves can be seen in Figure 1. Training the model for 1400 batches of size 200 yields a reported 0.497 accuracy on the test set. As can be observed, both the loss and accuracy curves exhibit comparable trends up until the first 1000 batches or so; after this point, the training loss continues to steadily decline, whilst the test loss appears to converge to a steady state value. This is paired with a similar divergence in the train and test accuracy, with the accuracy on the training set following a continuous upward trend, whilst the test accuracy becomes relatively stable after 1000 batches. This could be explained by the model over-fitting on the training set, where better performance on the training set no longer translates to improved generalisation on the test set.

The overall accuracy is also somewhat poor, which is to be expected: by nature, the MLP architecture lacks the appropriate inductive bias to effectively generalise on unseen image data.

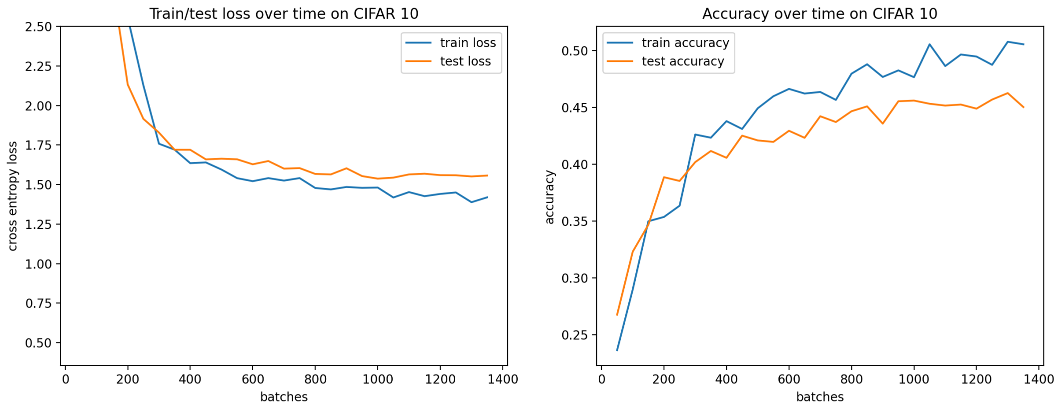


Figure 2: Left: loss on CIFAR-10 for Pytorch MLP with single layer of 100 units, pre-hyperparameter tuning. Right: accuracy on CIFAR-10 over time. Test set accuracy: 0.466.

2 Pytorch MLP

2.1 Experiments

First, we verified whether the Pytorch implementation was performing similarly to the NumPy implementation. To this end, a network was trained with an identical architecture - a single linear layer of 100 hidden units with ELU activation functions with alpha = 1, trained for 1400 batches of size 200, with a learning rate of 0.001, optimized via Minibatch Stochastic Gradient Descent. Loss and accuracy curves can be seen in Figure 2. Using this training configuration, an accuracy of 0.4612 was achieved on the test set, which is slightly lower than the NumPy MLP implementation, everything else equal.

Hyperparameter tuning In order to improve the classification performance, we conduct several experiments, modifying the network architecture, hyperparameter values, and optimization process, amongst other things. Ideally, one ought to observe effects for all possible combinations of settings before settling on the optimal choice for a particular parameter, since the effects of different settings are not independent, and different combinations could lead to different results. However, in the interest of time, this was considered beyond the scope of the assignment. Instead, we experiment with various parameters "one at a time", observing which value or choice leads to the best results, before moving on to the next parameter.

Adding depth to model architecture We first experiment with different network architectures, specifically, by adding additional linear layers, and experimenting with different numbers of hidden units per layer (all other parameters kept equal). The motivation for this choice was that possibly, making the network deeper would grant it greater expressiveness. In hindsight this might have not been a good choice; as can be seen in figure 2, the model is already over-fitting slightly, as indicated by the training accuracy steadily surpassing the test accuracy after around 400 batches. In this respect, we should possibly not expect gains from introducing additional complexity to the network via added depth.

Surprisingly, whilst introducing more complexity to the network lead to expected increases in divergence between the train and test accuracies (indicating an increase in over-fitting), adding more layers leads to a moderate increase in test accuracy, reporting 0.514 accuracy on the test set, all other settings kept at default. Here, it should be noted that the deeper networks do require more training data to reach results that are 'competitive' w.r.t the default MLP, requiring 4000 batches before surpassing the default MLP. However, even when accounting for this (by training the default, single layer MLP on the same amount of training data), there remains about a two percent difference between models. This suggests that adding depth alone positively impacts model generalisation. Based on these results, subsequent experiments were carried out with the best performing network architecture (as denoted by an asterix in Table 1).

Table 1: Effect of network depth on test accuracy, other settings kept at default.

`dnn_hidden_units`	Test accuracy
100	0.481
300, 100	0.483
300, 200, 100	0.490
250, 200, 150, 100	0.501
300, 250, 200, 150, 100	0.503
400, 300, 250, 200, 150, 100*	**0.514**

Learning rate Having determined an appropriate amount of hidden layers, we now consider the effect of adjusting the learning rate. Here, it was observed that a learning rate of 0.01 caused the network to cease learning altogether, with both train and test accuracy remaining near 0.10, which is close to a random guess. Possibly, this can be explained by the resulting weight updates being too large, thereby consistently 'overshooting' the minimum, as opposed to gradually converging to it. Conversely, a smaller learning rate of 0.0001 lead the network to converge significantly slower, requiring substantially more training data. Based on these (admittedly, crude) experiments, it appears that the default learning rate of 0.001 is most suitable.

Optimizer We also conduct a short experiment where we use a different optimizer. Here, we observed the effects of optimizing the network with Adam (where previously, SGD was used). This decision was primarily motivated by Adam being the current state-of-the-art optimizer. We initialize Adam with the default PyTorch parameters: `beta1` = 0.9, `beta2` = 0.999, `epsilon` = 1e-08, and zero weight decay. Results can be seen in Table 2. As can be observed, optimizing with Adam leads to improved generalisation and faster convergence, reaching an accuracy on the test set of 0.537 after having seen 4000 batches, a difference of approximately 4 percent compared to the same configuration trained with SGD.

Table 2: Effect of optimizer on train and test accuracy.

Optimizer	Train accuracy	Test accuracy
SGD	0.521	0.481
Adam	**0.770**	**0.537**

Batch-normalization Next, we introduce batch-normalization layers after each linear layer, which yields a test accuracy of 0.544, a slight increase compared to the previous configuration. For an explanation on batch normalization and why it can improve performance, please see section 3.2. In this case, the small increase in test performance could be understood to arise from the indirect regularising quality of batch normalization (which in turn results from the noisy batch statistics, which introduce a small amount of noise into the model). Possibly, the regularization induced by the batch normalization modules leads to slightly better generalisation, explaining the reported increase in test accuracy.

Table 3: Final model configuration.

DNN hidden units	400, 300, 250, 200, 150, 100
Learning rate	0.001
Optimizer	Adam
Max steps	2400

Final model See Table 3 for the best-performing model configuration. Note that the model was further augmented with Batch-normalization, as discussed previously, which is not indicated here. Loss and accuracy curves can be seen in Figure 3. As can be seen, the model appears to achieve its best accuracy on the test set after it has been trained on approximately 2400 batches - this point in training corresponds to a reported accuracy on the test set of 0.544. Beyond this point, we can see that the test loss gradually increases again, while the training loss further decreases. This is a clear example of over-fitting; as the model has seen more of the training data, it begins to over-fit

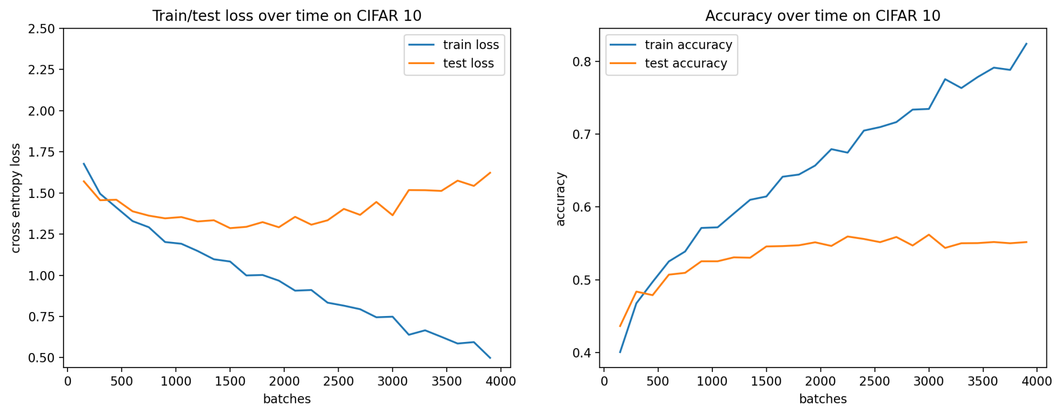


Figure 3: Left: loss on CIFAR-10 for Pytorch MLP with best-performing configuration. Right: accuracy on CIFAR-10 over time. Test set accuracy: 0.544.

(as reflected by the steady decrease in training loss / increase in accuracy on train set), at the cost of poorer generalisation.

An interesting observation here is that despite the model beginning to over-fit after only 500 batches, the increasing divergence between train and test loss is paired with an increase in test accuracy, suggesting that up to a point, over-fitting benefits generalisation. Furthermore, even as the model heavily over-fits on the training set (as indicated by its > 0.8 accuracy on the train set after 3500 batches), this does not appear to be paired with a decrease in test accuracy (despite the test loss exhibiting an upward trend between 2000 and 4000 batches).

In order to further optimize the performance of this model, several things could be considered. Particularly, given the complexity of the model, regularization techniques such as Drop-out might yield the largest gains - applying such techniques might curb the model's expressive power at the benefit of improved generalisation. Furthermore, one could perform a mini-ablation study where, instead of introducing the 'optimal' parameters in a successive, 'compounding' fashion, one could study the effects of different modifications in isolation (similarly to for instance, the experiments on different architecture depths, where results were compared to the base model). Possibly, this could provide us with insights on how different components affect model training and performance, and allow us to adjust the model configuration in a more informed way.

2.2) Tanh versus ELU

There are several factors tying into the choice between the Tanh and ELU activation functions. One of the benefits of the Tanh function is that it is zero-centered, meaning that its mean output is zero. Assuming the modules in our network are zero centered also, this guarantees strong gradients. The Tanh also has better output range compared to e.g. the sigmoid activation, with less 'positive' bias. However, the Tanh suffers from two drawbacks. First, the Tanh will saturate for extreme positive and negative values. The logits that arise from such saturation suggest a high degree of confidence, even when this is not necessarily the case. This is particularly problematic for intermediate layers (where Tanh activations are used predominantly), as these neurons ought to represent latent variables, and thus overconfidence might not be appropriate. Another disadvantage is that the Tanh always produces gradients that are < 1; particularly for deep networks, this could lead to a vanishing gradient problem, potentially causing 'early' layers to stop learning.

In these respects, the ELU might be a favourable choice; this activation is also zero-centered and produces strong gradients, but it suffers comparatively less from saturation and vanishing gradients. In practice, the only disadvantage of the ELU module concerns so-called "exploding" activations, given its output range of $[0, \infty]$; such behaviour might be particularly problematic in a network with a series of multiple ELU modules.

3 Custom Module: Layer Normalization

3.2 Manual implementation of backward pass - 3.2 d)

3.2 a) Derivative w.r.t. γ:

$$\frac{\delta L}{\delta \gamma} \to \frac{\delta L}{\delta \gamma_i} = \sum_{m,n} \frac{\delta L}{\delta Y_{mn}} \frac{\delta Y_{mn}}{\delta \gamma_i}$$

$$\frac{\delta Y_{mn}}{\delta \gamma_i} = \frac{\delta}{\delta \gamma_i} \gamma_m \hat{X}_{mn} + \beta_n = \delta_{mi} \hat{X}_{mn}$$

$$\sum_{m,n} \frac{\delta L}{\delta Y_{mn}} \delta_{mi} \hat{X}_{mn} = \sum_{n} \frac{\delta L}{\delta Y_{in}} \hat{X}_n = \left[\frac{\delta L}{\delta Y} \hat{X} \right]_i \to \frac{\delta L}{\delta \gamma} = \frac{\delta L}{\delta Y} \hat{X}$$

Derivative w.r.t. β:

$$\frac{\delta L}{\delta \beta} \to \frac{\delta L}{\delta \beta_i} = \sum_{m,n} \frac{\delta L}{\delta Y_{mn}} \frac{\delta Y_{mn}}{\delta \beta_i}$$

$$\frac{\delta Y_{mn}}{\delta \beta_i} = \frac{\delta}{\delta \beta_i} \gamma_m \hat{X}_{mn} + \beta_n = \delta_{ni}$$

$$\frac{\delta L}{\delta \beta_i} = \sum_{m,n} \frac{\delta L}{\delta Y_{mn}} \delta_{ni} = \sum_{m} \frac{\delta L}{\delta Y_{mi}} (1) = \left[\frac{\delta L}{\delta Y} \mathbf{1} \right]_i \to \frac{\delta L}{\delta \beta} = \frac{\delta L}{\delta Y} \mathbf{1}$$

Derivative w.r.t. X:

$$\frac{\delta L}{\delta X} = \frac{\delta L}{\delta Y} \frac{\delta Y}{\delta X}$$

$$\frac{\delta Y}{\delta X} = \left(\frac{\delta Y}{\delta \hat{X}} \frac{\delta \hat{X}}{\delta X} \right) + \left(\frac{\delta Y}{\delta \hat{X}} \left(\frac{\delta \hat{X}}{\delta \mu} \frac{\delta \mu}{\delta X} + \frac{\delta \hat{X}}{\delta \sigma^2} \frac{\delta \sigma^2}{\delta \mu} \frac{\delta \mu}{\delta X} \right) \right) + \left(\frac{\delta Y}{\delta X} \frac{\delta \hat{X}}{\delta \sigma^2} \frac{\delta \sigma^2}{\delta X} \right)$$

I derive all the single-element derivatives.

$$\frac{\delta \mu_s}{\delta X_{pq}} = \frac{\delta}{\delta X_{pq}} \frac{1}{M} \sum_i X_{si} = \frac{1}{M} \sum_i \delta_{sp} \delta_{iq} = \frac{1}{M} \delta_{sp}.$$

$$\begin{aligned} \frac{\delta \sigma_s^2}{\delta X_{pq}} &= \frac{\delta}{\delta X_{pq}} \frac{1}{M} \sum_i (X_{si} - \mu_s)^2 = \frac{1}{M} \sum_i 2(X_{si} - \mu_s) \frac{\delta}{\delta X_{pq}} (X_{si} - \mu_s) \\ &= \frac{1}{M} \sum_i 2(X_{si} - \mu_s)(\delta_{sp}\delta_{iq} - \frac{1}{M}\delta_{sp}) = \frac{2(X_{sq} - \mu_s)}{M} (\delta_{sp}). \end{aligned}$$

$$\begin{aligned} \frac{\sigma_s^2}{\mu_p} &= \frac{\delta}{\delta \mu_p} \frac{1}{M} \sum_i (X_{si} - \mu_s)^2 = \frac{1}{M} \sum_i 2(X_{si} - \mu_s) \frac{\delta}{\delta \mu_p} (X_{si} - \mu_s) \\ &= \frac{1}{M} \sum_i 2(X_{si} - \mu_s) - \delta_{sp} = \frac{2\sum_i (X_{si} - \mu_s)}{M} (-\delta_{sp}). \end{aligned}$$

$$\frac{\delta \hat{X}_{si}}{\delta \mu_p} = \frac{\delta}{\delta \mu_p} \frac{X_{si} - \mu_s}{\sqrt{\sigma_s^2 + \epsilon}} = \frac{\delta}{\delta \mu_p} (X_{si} - \mu_s)(\sigma_s^2 + \epsilon)^{-0.5} = -\delta_{sp} (\sigma_s^2 + \epsilon)^{-0.5} = \frac{-1}{\sqrt{\sigma_s^2 + \epsilon}} (\delta_{sp}).$$

$$\frac{\delta \hat{X}_{si}}{\delta X_{pq}} = \frac{\delta}{\delta X_{pq}} \frac{X_{si} - \mu_s}{\sqrt{\sigma_s^2 + \epsilon}} = \frac{\delta}{\delta X_{pq}} (X_{si} - \mu_s)(\sigma_s^2 + \epsilon)^{-0.5} = \delta_{sp} \delta_{iq} (\sigma_s^2 + \epsilon)^{-0.5} = \frac{1}{\sqrt{\sigma_s^2 + \epsilon}} (\delta_{sp} \delta_{iq}).$$

$$\frac{\delta \hat{X}_{si}}{\delta \sigma_p^2} = \frac{\delta}{\delta \sigma_p^2}(X_{si}-\mu_s)(\sigma_s^2+\epsilon)^{-0.5} = -0.5(X_{si}-\mu_s)(\sigma_s^2+\epsilon)^{-1.5}\delta_{sp} = -0.5(X_{si}-\mu_s)(\sigma_s^2+\epsilon)^{-1.5}(\delta_{sp}).$$

$$\frac{\delta Y_{si}}{\delta \hat{X}_{pq}} = \frac{\delta}{\delta \hat{X}_{pq}}\gamma_i \hat{X}_{si} + \beta_i = \delta_{sp}\delta_{iq}\gamma_i = \gamma_q(\delta_{sp}).$$

We can now evaluate the entire derivative;

$$\left(\frac{\delta Y}{\delta \hat{X}}\frac{\delta \hat{X}}{\delta X}\right) = \gamma_i \cdot \frac{1}{\sqrt{\sigma_s^2+\epsilon}},$$

$$\left(\frac{\delta Y}{\delta \hat{X}}\left(\frac{\delta \hat{X}}{\delta \mu}\frac{\delta \mu}{\delta X} + \frac{\delta \hat{X}}{\delta \sigma^2}\frac{\delta \sigma^2}{\delta \mu}\frac{\delta \mu}{\delta X}\right)\right) = \gamma_i \cdot \left(\frac{-1}{\sqrt{\sigma_s^2+\epsilon}} \cdot \frac{1}{M} + -0.5(X_{si}-\mu_s)(\sigma_s^2+\epsilon)^{-1.5} \cdot \frac{2(X_s-\mu_s)}{M} \cdot \frac{1}{M}\right)$$

$$\left(\frac{\delta Y}{\delta \hat{X}}\frac{\delta \hat{X}}{\delta \sigma^2}\frac{\delta \sigma^2}{\delta X}\right) = \gamma_i \cdot -0.5(X_{si}-\mu_s)(\sigma_s^2+\epsilon)^{-1.5} \cdot \frac{2(X_s-\mu_s)}{M}$$

Full disclosure: I got quite lost with this derivative (I also did not manage to implement the backward pass for the next question, in case you were looking for it).

3.2 d) - Batch Normalization v.s. Layer Normalization The technique of batch normalization was initially designed to account for the change in input distributions as arising from gradient updates in preceding layers, by normalizing the input distribution of each layer using mini-batch statistics. This yields zero-centered input distributions that remain stable over time, ensuring that neurons are activated in their optimal 'regimes', providing a more reliable learning signal through time. One explanation on why batch normalization is effective is that it dampens higher order layer interactions, simplifying the overall learning dynamics.

Since the batch normalization operation can be incorporated directly into the gradient update step, and since it has learnable parameters to shift and scale the input distribution, we can exercise more control over the activations in each layer. Because this can reduce the likelihood of vanishing or exploding gradients, we can afford to use higher learning rates, which can speed up the training process. It has also been suggested that batch normalization can act as a kind of model regularization, as the per-batch statistics are effectively noisy estimates of the true mean and variance, which would supposedly reduce over-fitting.

A disadvantage of batch normalization is that it requires large mini-batches to be effective; we cannot use a batch of size 1, as the variance will be 0; for small batches (e.g. < 32), the statistics are likely too noisy to provide adequate normalization. Another drawback is that batch normalization is not well-suited for recurrent neural networks, since we have to compute the batch statistics for each time-step, for each layer, complicating the training process. In addition, it requires us to store the past statistics of each time-step during training, which can be memory-intensive.

An alternative to batch normalization is layer normalization. The latter differs from the former in that it does not compute the mean and variance over the batch dimension, but rather over the feature dimension. Effectively, we are applying separate normalization to each input, independent of other inputs. A clear advantage of this is that we can choose to use smaller mini-batches without this (adversely) affecting the effectiveness of the normalization procedure (as would be the case with batch normalization).

Research has also shown that layer normalization is more effective in recurrent settings, allowing comparatively straightforward computations, whilst boasting stable hidden state dynamics for long sequences. However, it was also found that layer normalization is less effective in convolutional neural networks, since the statistics of hidden units for a given layer can vary heavily depending on the location of units with respect to the image (Hinton, 2016). This suggests that batch normalization is still a favourable choice when one looking to apply normalization in ConvNets.

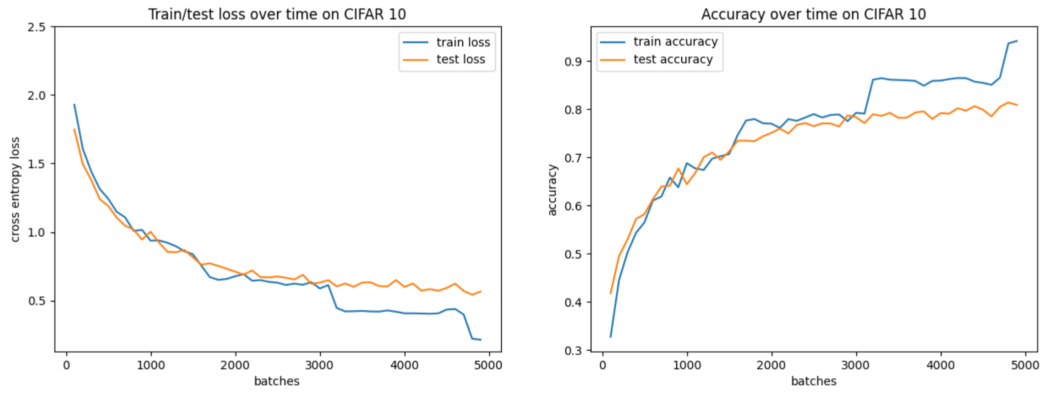


Figure 4: Left: loss on CIFAR-10 for Pytorch ConvNet. Right: accuracy on CIFAR-10 over time. Test set accuracy after 5000 batches: 0.811

4 PyTorch CNN

4 a) PyTorch ConvNet Results can be seen in Figure 4. As can be observed, the test loss curve exhibits a gradual decline over time, converging after 5000 batches, which translates to an accuracy on the test set of 0.811. From roughly 3000 batches onwards, these large gains in training accuracy no longer translate to equivalent increases in testing accuracy, suggesting that the model is nearing convergence. (I am aware of the step-wise behaviour of the train loss and I suspect it has something to do with Adam, but after asking on Piazza I was reassured that this doesn't require clarification, hence I do not discuss it)

4 b) Transfer Learning Code can be found in `train_finetuned.py`. Here, we fine-tune an instance of ResNet18 on the CIFAR-10 dataset. Particularly, we briefly explore how unfreezing a smaller/larger portion of the network translates to differences in performance.

In the first experiment, all parameters of the pre-trained model are kept fixed during training, except for the last linear layer (Figure 5). In the second experiment, we unfreeze the last twelve trainable parameters, thereby allowing us to fine-tune further on the CIFAR-10 data (Figure 6). Finally, in the third experiment we unfreeze the last 30 layers (Figure 7).

The results in Figure 5 suggest that only fine-tuning the last linear layer puts too much of a constraint on the model's ability to generalize on the CIFAR-10 data; after 4000 batches, it is already beginning to converge, albeit at a relatively poor 0.43 accuracy on the test set. Better results are achieved when we unfreeze more layers; unfreezing the last 10 layers yields an accuracy of 0.6239 after the same amount of batches, and unfreezing the last 30 layers yields an accuracy of 0.7515. As could be expected, there is a clear positive relationship between the proportion of unfrozen parameters and the model's ability to fine-tune to new data. It should be noted however that naturally, unfreezing more layers will lead to greater computational requirements, since more parameters need to be adjusted; greater fine-tuning ability can therefore be more costly.

References

Ba, J., Kiros, J., & Hinton, G.E. (2016). Layer Normalization. *ArXiv, abs/1607.06450.*

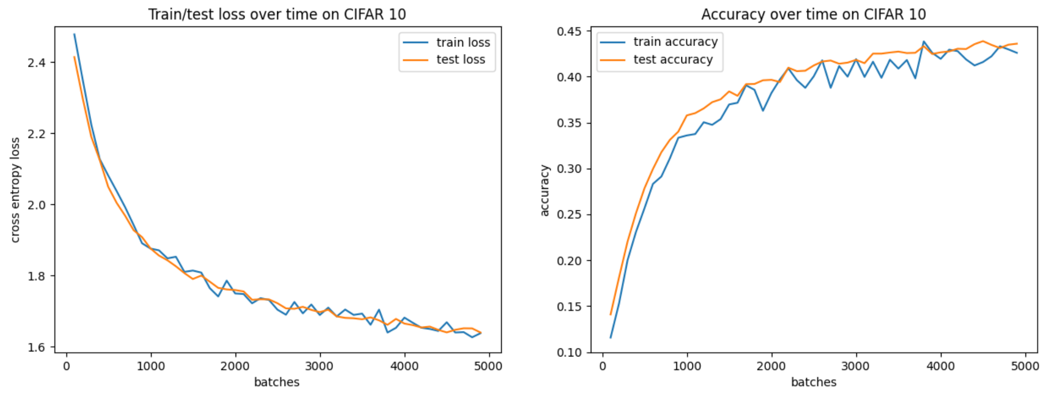


Figure 5: Left: loss on CIFAR-10 for fine-tuned ResNet18, all layers except linear layer kept frozen. Right: accuracy on CIFAR-10 over time. Test set accuracy after 5000 batches: 0.4383

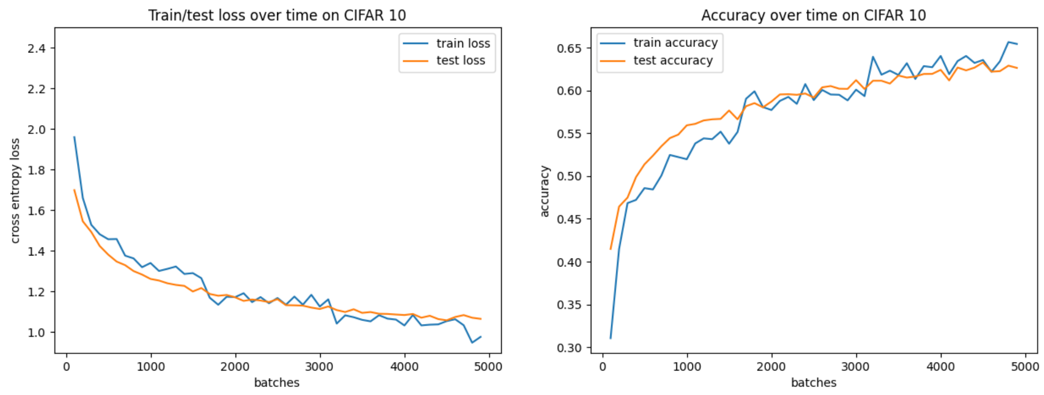


Figure 6: Left: loss on CIFAR-10 for fine-tuned ResNet18 with last 10 layers unfrozen. Right: accuracy on CIFAR-10 over time. Test set accuracy after 5000 batches: 0.6239

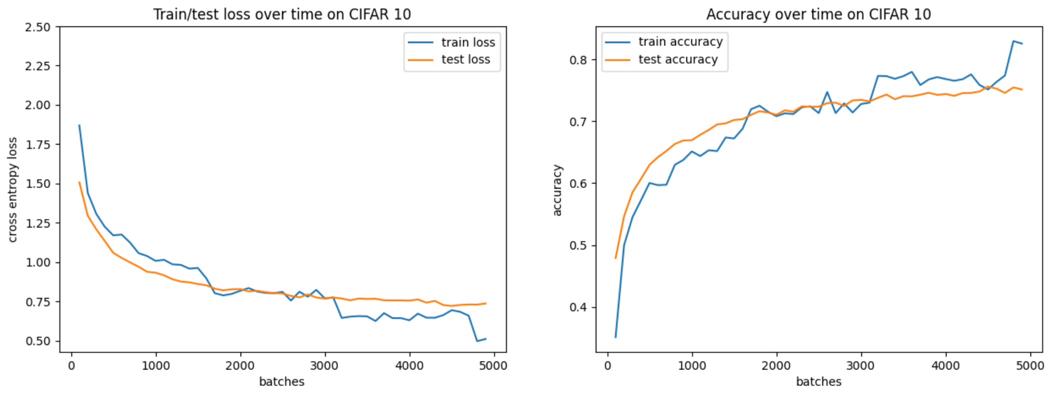


Figure 7: Left: loss on CIFAR-10 for fine-tuned ResNet18 with last 30 layers unfrozen. Right: accuracy on CIFAR-10 over time. Test set accuracy after 5000 batches: 0.7515