Deep Learning
Assignment 1: MLPs, CNNs and Backpropagation

Daniel Daza
University of Amsterdam
`daniel.dazacruz@student.uva.nl`

Abstract

Neural networks are flexible function approximators that have a wide range of applications in artificial intelligence. In this work we derive the gradient equations that are needed to train a Multi-Layer Perceptron (MLP), and we implement and test its performance on the task of image classification. We additionally derive the gradient equations for the Batch-Normalization layer. Lastly, we implement a Convolutional Neural Network (CNN) and test it on the same task. We find that the MLP can generalize to samples outside the training set by using regularization techniques, albeit at a lower accuracy; and that networks with more parameters and appropriate prior structure, such as CNNs, yield better performance.

1 MLP, backpropagation and NumPy implementation

1.1 Analytical derivation of gradients

Question 1.1 a)

Cross-entropy loss

Since the loss only depends on x_i if $t_i = 1$, this gradient is a vector of zeros except in the position given by $\arg\max(t)$:

$$\left(\frac{\partial L}{\partial x^{(N)}}\right)_i = \begin{cases} 0 & \text{if } i \neq \arg\max(t) \\ \frac{\partial}{\partial x_i^{(N)}}\left(-\log x_i^{(N)}\right) & \text{if } i = \arg\max(t) \end{cases} \tag{1}$$

$$= \begin{cases} 0 & \text{if } i \neq \arg\max(t) \\ -\frac{1}{x_i^{(N)}} & \text{if } i = \arg\max(t) \end{cases} \tag{2}$$

Therefore, the gradient vector can be obtained by element-wise multiplication with the targets vector:

$$\frac{\partial L}{\partial x^{(N)}} = -t \odot \frac{1}{x^{(N)}} \tag{3}$$

Softmax layer

$$
\begin{aligned}
\left(\frac{\partial x^{(N)}}{\tilde{x}^{(N)}}\right)_{ij} &= \frac{\partial x_i^{(N)}}{\tilde{x}_j^{(N)}} && (4)\\
&= \frac{\partial}{\partial \tilde{x_j}^{(N)}}\left(\frac{\exp(\tilde{x}^{(N)})_i}{\sum_{k=1}^{d_N}\exp(\tilde{x}^{(N)})_k}\right) && (5)\\
&= \frac{\partial}{\partial \tilde{x_j}^{(N)}}\left(\frac{\exp(\tilde{x}^{(N)})_i}{Z}\right) && (6)\\
&= \frac{Z\frac{\partial}{\partial \tilde{x}_j}\exp(\tilde{x}^{(N)})_i - \exp(\tilde{x}^{(N)})_i\frac{\partial}{\partial \tilde{x}_j}Z}{Z^2} && (7)\\
&= \frac{Z\frac{\partial}{\partial \tilde{x}_j}\exp(\tilde{x}^{(N)})_i - \exp(\tilde{x}^{(N)})_i\frac{\partial}{\partial \tilde{x}_j}\sum_{k=1}^{d_N}\exp(\tilde{x}_k^{(N)})}{Z^2} && (8)\\
&= \frac{Z\frac{\partial}{\partial \tilde{x}_j}\exp(\tilde{x}^{(N)})_i - \exp(\tilde{x}^{(N)})_i\exp(\tilde{x}_j^{(N)})}{Z^2} && (9)\\
&= \frac{\frac{\partial}{\partial \tilde{x}_j}\exp(\tilde{x}^{(N)})_i}{Z} - \frac{\exp(\tilde{x}^{(N)})_i\exp(\tilde{x}_j^{(N)})}{Z^2} && (10)\\
&= \frac{\frac{\partial}{\partial \tilde{x}_j}\exp(\tilde{x}^{(N)})_i}{Z} - \frac{\exp(\tilde{x}^{(N)})_i}{Z}\frac{\exp(\tilde{x}_j^{(N)})}{Z} && (11)\\
&= \frac{\frac{\partial}{\partial \tilde{x}_j}\exp(\tilde{x}^{(N)})_i}{Z} - \text{softmax}(\tilde{x}^{(N)})_i\text{softmax}(\tilde{x}^{(N)})_j && (12)\\
&= \begin{cases} -\text{softmax}(\tilde{x}^{(N)})_i\text{softmax}(\tilde{x}^{(N)})_j \text{ if } i \neq j \\ \frac{\exp(\tilde{x}^{(N)})_j}{Z} - \text{softmax}(\tilde{x}^{(N)})_i\text{softmax}(\tilde{x}^{(N)})_j \text{ if } i = j \end{cases} && (13)\\
&= \begin{cases} -x_i^{(N)}x_j^{(N)} \text{ if } i \neq j \\ x_j^{(N)} - x_i^{(N)}x_j^{(N)} \text{ if } i = j \end{cases} && (14)\\
& && (15)
\end{aligned}
$$

The Jacobian is then:

$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \text{diag}(x^{(N)}) - x^{(N)}x^{(N)\top} \tag{16}$$

ReLU activation

$$
\begin{aligned}
\left(\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}}\right)_{ij} &= \frac{\partial}{\partial \tilde{x}_j^{(l)}}\max(0, \tilde{x}_i^{(l)}) && (17)\\
&= \begin{cases} 1 & \text{if } i = j \text{ and } \tilde{x}^{(l)} < 0 \\ 0 & \text{otherwise} \end{cases} && (18)
\end{aligned}
$$

If we let $\mathbb{I}[s]$ be an indicator variable equal to 1 when the element-wise statement s is true, the Jacobian can be written as follows:

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \text{diag}(\mathbb{I}[\tilde{x}^{(l)} > 0]) \tag{19}$$

Linear layer

$$
\begin{aligned}
\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} &= \frac{\partial}{\partial x^{(l-1)}}(W^{(l)}x^{(l-1)} + b^{(l)}) && (20)\\
&= W^{(l)} && (21)
\end{aligned}
$$

Let $W_{i:}^{(l)}$ be a slice containing the i-th row of $W^{(l)}$. Then we have:

$$\left(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}\right)_{ijk} = \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} \tag{22}$$

$$= \frac{\partial}{\partial W_{jk}^{(l)}}(W^{(l)}x^{(l-1)} + b^{(l)})_i \tag{23}$$

$$= \frac{\partial}{\partial W_{jk}^{(l)}}(W_{i:}^{(l)\top}x^{(l-1)} + b_i^{(l)}) \tag{24}$$

$$= \begin{cases} x_k^{(l-1)} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

Lastly,

$$\left(\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}\right)_{ij} = \frac{\partial}{\partial b_j^{(l)}}(W^{(l)}x^{(l-1)} + b^{(l)})_i \tag{26}$$

$$= \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{27}$$

Question 1.1 b)

Softmax backward

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}}\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} \tag{28}$$

$$= \frac{\partial L}{\partial x^{(N)}}\left(\text{diag}(x^{(N)}) - x^{(N)}x^{(N)\top}\right) \tag{29}$$

ReLU backward

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}}\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} \tag{30}$$

$$= \frac{\partial L}{\partial x^{(l)}}\text{diag}(\mathbb{I}[\tilde{x}^{(l)} > 0]) \tag{31}$$

$$= \frac{\partial L}{\partial x^{(l)}} \odot \mathbb{I}[\tilde{x}^{(l)} > 0] \tag{32}$$

Linear backward

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}}\frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}}$$

$$= \frac{\partial L}{\partial \tilde{x}^{(l+1)}}W^{(l+1)}$$

From the chain rule we have:

$$\left(\frac{\partial L}{\partial W^{(l)}}\right)_{ij} = \sum_k \frac{\partial L}{\partial \tilde{x}_k^{(l)}}\frac{\partial \tilde{x}_k^{(l)}}{\partial W_{ij}} \tag{33}$$

$$= \frac{\partial L}{\partial \tilde{x}_i^{(l)}}x_j^{(l-1)} \tag{34}$$

Therefore, the gradient with respect to $W^{(l)}$ can be written as an outer product:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} x^{(l-1)\top} \tag{35}$$

$$\left(\frac{\partial L}{\partial b^{(l)}}\right)_i = \sum_j \frac{\partial L}{\partial \tilde{x}_j^{(l)}} \frac{\partial \tilde{x}_j^{(l)}}{\partial b_i^{(l)}} \tag{36}$$

$$= \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \tag{37}$$

Therefore:

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \tag{38}$$

Question 1.1 c)

Let us assume that when using batches, the B samples and targets are arranged in the rows of matrices $X^{(0)}$ and T, and for any linear module the output is calculated as $X^{(l-1)}W^{(l)\top} + b^{(l)}$. In this case, the output of the module will be a matrix as well.

When using a single sample, we obtained the following gradient for the cross-entropy loss:

$$\frac{\partial L}{\partial x^{(N)}} = -t \odot \frac{1}{x^{(N)}}$$

This vector is used during backpropagation to calculate the next gradients. In the new batched formulation, it turns into is a matrix that we can arrange so that the number of rows equals the number of samples in the batch:

$$-T \odot \frac{1}{X^{(N)}} = \begin{bmatrix} -T_{1:} \odot \frac{1}{X_{1:}^{(N)}} \\ -T_{2:} \odot \frac{1}{X_{2:}^{(N)}} \\ \vdots \\ -T_{B:} \odot \frac{1}{X_{B:}^{(N)}} \end{bmatrix} \tag{39}$$

By matrix multiplication, every backward step of a module will then produce a matrix with one row per sample. When calculating the gradients of the parameters, each sample in the batch will produce a gradient matrix for $W^{(l)}$ and a gradient vector for $b^{(l)}$ via brodcasting of matrix multiplication. These gradient arrays can be aggregated across the batch axis before updating the parameters.

Question 1.2: NumPy implementation

The previous equations were used to implement an MLP with the NumPy library to solve an image classification problem with the CIFAR-10 dataset [1], with a training set of 50,000 images and 10,000 images for testing. We trained a 2-layer MLP with 100 units in the hidden layer and the ReLU activation function, a batch size of 200, Stochastic Gradient Descent (SGD) with a learning rate of 1×10^{-3}, for 1,500 steps. To evaluate the performance of the classifier we use the accuracy score.

We obtained an accuracy of 46.6% in the test set. During training, we observed that the test set obtained a value of 48.8% after 1,000 iterations, and later decreased, which indicates a sign of overfitting. This can also be seen in the loss and accuracy curves, shown in Figure 1, where a gap between the training and test curves starts to appear towards the end of training. We can also see that at 1,500 iterations the curves appear to stagnate, suggesting that training for more iterations would further decrease the performance on the test set due to overfitting.

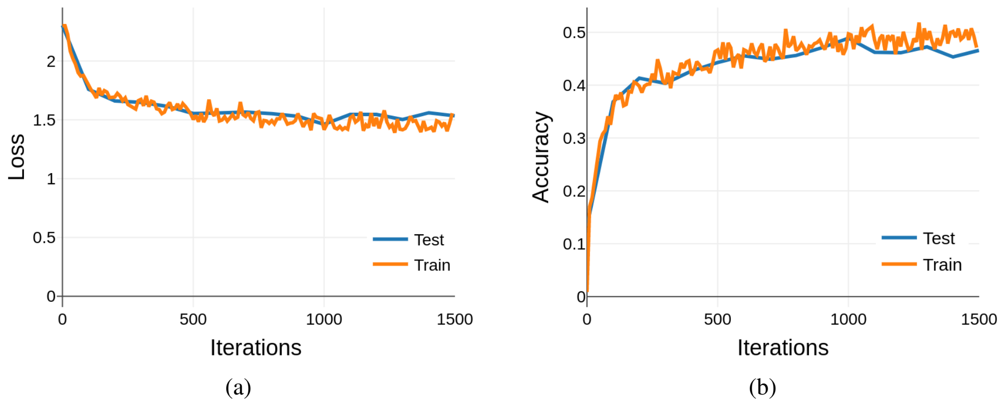


Figure 1: Curves for (a) the loss, and (b) accuracy on the training and test splits, when training an MLP for image classification with NumPy. Training values are averaged over 10 steps, and test values are measured every 100 steps.

Table 1: Hyperparameter settings for the grid search carried out for the PyTorch implementation of the MLP. The best values are shown for the model with the highest accuracy on the test set.

Hyperparameter	**Values**	**Best**
Units in the first layer	$\{400, 600, 1000\}$	600
Weight decay	$\{0.1, 0.01, 0.001\}$	0.1
Dropout	$\{0.1, 0.2, 0.5\}$	0.1
Learning rate	$\{1 \times 10^{-3}, 5 \times 10^{-4}, 1 \times 10^{-4}\}$	1×10^{-4}

2 PyTorch MLP

Question 2

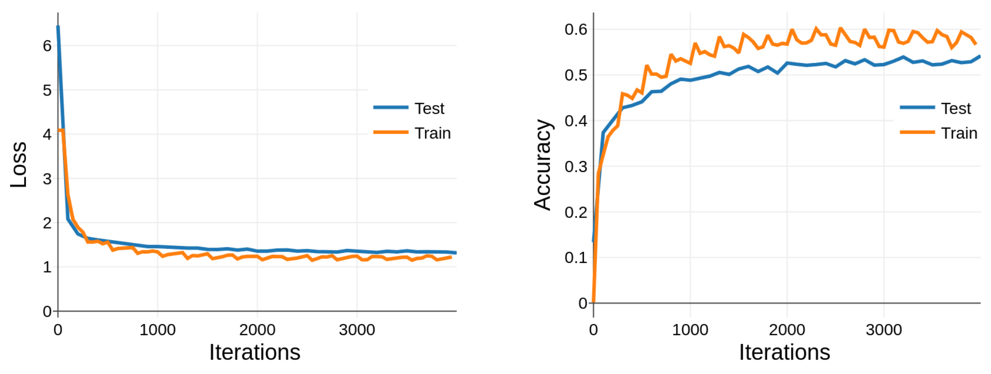
We implemented the same MLP using the PyTorch library, and obtained an accuracy of 43.7% with the same hyperparameters as the NumPy implementation. We attribute the difference to the initialization strategy, as in the NumPy implementation we initialize the weights from a Gaussian distribution with a mean of 0 and a standard deviation of 0.0001, whereas by default PyTorch uses the *He initialization* strategy [2].

In order to improve the performance of the model, we experiment with modifications to the architecture and training procedure. Increasing the number of parameters allows the model to represent more complex functions [3], and arranging these into different layers imposes a hierarchical inductive bias that has shown to improve the generalization properties of neural networks [4, 5]. However, the increased complexity of the model also increases its propensity to overfitting to the training data, and methods like regularizing the ℓ_2-norm of the weights, and dropout [6] have been proposed to overcome this outcome.

Another important component in training neural networks is the optimization algorithm used to minimize the loss. While our first experiments use SGD, other alternatives have been proposed that improve the convergence speed, such as the use of momentum [7, 8] and second order moment estimation, as used in the Adam algorithm [1].

Based on the previous reasons, we experiment with a model with three layers. We vary the number of units in the first layer, and we fix the second layer with 400 units. We add dropout to the output of the second layer, and we train the neural network with Adam, including weight decay regularization. We train for 4,000 iterations, which we observed was necessary to achieve a stable convergence of the training loss. To find the best set of hyperparameters, we run grid search according to the values listed in Table 1.

Table 1 also includes the best set of hyperparameters, which yielded an accuracy of 54.2% in the test set. We note that the best model is not the one with more parameters, but a model in the mid range that is also regularized both with weight decay and dropout. The loss and accuracy curves for



(a) (b)

Figure 2: Curves for (a) the loss, and (b) accuracy on the training and test splits, when training an MLP for image classification with PyTorch. Training values are averaged over 10 steps, and test values are measured every 100 steps.

this model are shown in Figure 2. We can see that the regularization methods effectively prevent the model from overfitting, as the gap between the training and test curves is controlled as the number of iteration increases. The accuracy curves also suggest that training for more iterations could further improve the performance on the test set.

3 Batch Normalization

We reproduce the forward propagation equations for the batch normalization layer here for future reference:

$$\mu_i = \frac{1}{B}\sum_{s=1}^{B} x_i^s \tag{40}$$

$$\sigma_i = \frac{1}{B}\sum_{s=1}^{B} (x_i^s - \mu_i)^2 \tag{41}$$

$$\hat{x}_i^s = \frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{42}$$

$$y_i^s = \gamma_i \hat{x}_i^s + \beta_i \tag{43}$$

Question 3.1

For this question, the forward propagation equations were implemented in PyTorch.

Question 3.2 a)

We now derive the backpropagation equations for the batch normalization layer. We are interested in the gradients of the loss with respect to γ, β, and x. We assume that the gradient with respect to the module output, y, is already provided by the layer following it.

$$\begin{aligned}
\left(\frac{\partial L}{\partial \gamma}\right)_j &= \frac{\partial L}{\partial \gamma_j} \\
&= \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \gamma_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} \frac{\partial y_j^s}{\partial \gamma_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} \frac{\partial}{\partial \gamma_j} (\gamma_j \hat{x}_j^s + \beta_j) \\
&= \sum_s \frac{\partial L}{\partial y_j^s} \hat{x}_j^s
\end{aligned}$$

$$\begin{aligned}
\left(\frac{\partial L}{\partial \beta}\right)_j &= \frac{\partial L}{\partial \beta_j} \\
&= \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial \beta_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} \frac{\partial y_j^s}{\partial \beta_j} \\
&= \sum_s \frac{\partial L}{\partial y_j^s} \frac{\partial}{\partial \beta_j} (\gamma_j \hat{x}_j^s + \beta_j) \\
&= \sum_s \frac{\partial L}{\partial y_j^s}
\end{aligned}$$

$$\begin{aligned}
\left(\frac{\partial L}{\partial x}\right)_{jk} &= \frac{\partial L}{\partial x_j^k} \\
&= \frac{\partial L}{\partial y_j^k} \frac{\partial y_j^k}{\partial \hat{x}_j^k} \frac{\partial \hat{x}_j^k}{\partial x_j^k} + \sum_s \frac{\partial L}{\partial y_j^s} \frac{\partial y_j^s}{\partial \hat{x}_j^s} \left(\frac{\partial \hat{x}_j^s}{\partial \mu_j} \frac{\partial \mu_j}{\partial x_j^k} + \frac{\partial \hat{x}_j^s}{\partial \sigma_j^2} \left[\frac{\partial \sigma_j^2}{\partial x_j^k} + \frac{\partial \sigma_j^2}{\partial \mu_j} \frac{\partial \mu_j}{\partial x_j^k} \right] \right) \qquad (44)
\end{aligned}$$

The derivatives required in the last equation are derived next:

$$\begin{aligned}
\frac{\partial y_j^k}{\partial \hat{x}_j^k} &= \frac{\partial}{\partial \hat{x}_j^k} (\gamma_j \hat{x}_j^k + \beta_j) \\
&= \gamma_j
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \hat{x}_j^k}{\partial x_j^k} &= \frac{\partial}{\partial x_j^k} \left(\frac{x_j^k - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \right) \\
&= \frac{1}{\sqrt{\sigma_j^2 + \epsilon}}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \hat{x}_j^s}{\partial \mu_j} &= \frac{\partial}{\partial \mu_j} \left(\frac{x_j^s - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \right) \\
&= -\frac{1}{\sqrt{\sigma_j^2 + \epsilon}}
\end{aligned}$$

$$\begin{aligned}\frac{\partial \mu_j}{\partial x_j^k} &= \frac{\partial}{\partial x_j^k}\frac{1}{B}\sum_s x_j^s \\ &= \frac{1}{B}\end{aligned}$$

$$\begin{aligned}\frac{\partial \hat{x}_j^s}{\partial \sigma_j^2} &= \frac{\partial}{\partial \sigma_j^2}\left(\frac{x_j^s-\mu_j}{\sqrt{\sigma_j^2+\epsilon}}\right) \\ &= -\frac{1}{2}(\sigma_j^2+\epsilon)^{-3/2}(x_j^s-\mu_j)\end{aligned}$$

$$\begin{aligned}\frac{\partial \sigma_j^2}{\partial x_j^k} &= \frac{\partial}{\partial x_j^k}\frac{1}{B}\sum_s (x_j^s-\mu_j)^2 \\ &= \frac{2}{B}(x_j^k-\mu_j)\end{aligned}$$

$$\begin{aligned}\frac{\partial \sigma_j^2}{\partial \mu_j} &= \frac{\partial}{\partial \mu_j}\frac{1}{B}\sum_s (x_j^s-\mu_j)^2 \\ &= -\frac{2}{B}\sum_s (x_j^s-\mu_j) \\ &= -\frac{2}{B}\sum_s x_s^j + \frac{2}{B}\sum_s \mu_j \\ &= -2\mu_j + 2\mu_j \\ &= 0\end{aligned}$$

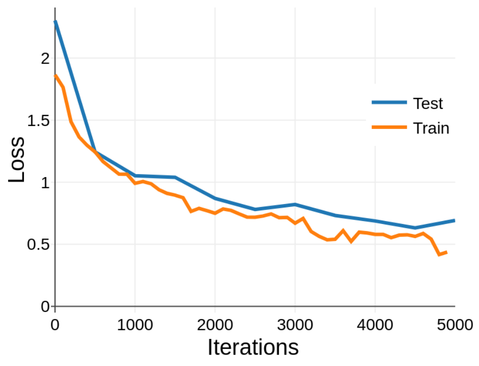
Since this last derivative is 0, we do not need to calculate $\partial \mu_j / \partial x_j^k$. Substituting the rest of the expressions in Eq. 44, we obtain

$$\begin{aligned}\frac{\partial L}{\partial x_j^k} &= \frac{\partial L}{\partial y_j^k}\frac{\gamma_j}{\sqrt{\sigma_j^2+\epsilon}} - \frac{\gamma_j}{B\sqrt{\sigma_j^2+\epsilon}}\sum_s \frac{\partial L}{\partial y_j^s}\left(1+\frac{(x_j^s-\mu_j)}{\sqrt{\sigma_j^2+\epsilon}}\frac{(x_j^k-\mu_j)}{\sqrt{\sigma_j^2+\epsilon}}\right) \\ &= \frac{\gamma_j}{\sqrt{\sigma_j^2+\epsilon}}\left(\frac{\partial L}{\partial y_j^k} - \frac{1}{B}\sum_s \frac{\partial L}{\partial y_j^s}(1+\hat{x}_j^s\hat{x}_j^k)\right) \\ &= \frac{\gamma_j}{\sqrt{\sigma_j^2+\epsilon}}\left(\frac{\partial L}{\partial y_j^k} - \frac{1}{B}\sum_s \frac{\partial L}{\partial y_j^s} - \frac{1}{B}\hat{x}_j^k\sum_s \frac{\partial L}{\partial y_j^s}\hat{x}_j^s\right) \\ &= \frac{\gamma_j}{B\sqrt{\sigma_j^2+\epsilon}}\left(B\frac{\partial L}{\partial y_j^k} - \frac{\partial L}{\partial \beta_j} - \hat{x}_j^k\frac{\partial L}{\partial \gamma_j}\right)\end{aligned}$$

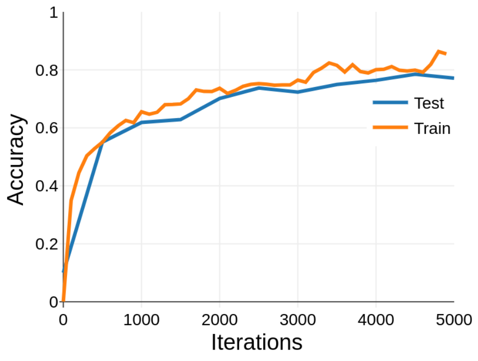
where in the last step we have substituted $\partial L/\partial \beta_j$ and $\partial L/\partial \gamma_j$, which can be computed first to be reused in the calculation of $\partial L/\partial x_j^k$.

Questions 3.2 b) and c)

For this part we implemented the forward and backward equations derived in the previous section, which were tested by checking the gradient against a gradient computed numerically.



(a)



(b)

Figure 3: Curves for (a) the loss, and (b) accuracy on the training and test splits for the CNN. Training values are averaged over 10 steps, and test values are measured every 100 steps.

4 PyTorch CNN

Question 4

We implemented a Convolutional Neural Network (CNN) based on the VGG network for image recognition [9]. It consists of a series of convolutional layers, batch normalization, ReLU and pooling layers. We train the network for 5,000 iterations, using the Adam optimizer with a learning rate of 1×10^{-4}.

To account for the randomness introduced in the initialization of parameters, we train the model with 10 different random seeds. The model achieves an average accuracy in the test set of 77.0% with a standard deviation of 0.6%, with a maximum value of 78.5%. We also train a model that does not include batch normalization, which achieves an accuracy of 73.8%, showing that this layer is effective when training large neural networks, as shown by Ioffe and Szegedy (2015) [10].

The loss and accuracy curves are shown in Figure 3 for the model with the highest accuracy. The lower loss and higher accuracy values, compared to the performance of the MLP, show that CNNs are better suited for the task by introducing appropriate prior structure that considers translational invariance and parameter sharing. The network yields high accuracy without overfitting, as shown by the small difference between training and test performance.

References

[1] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[3] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[4] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

[5] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[7] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[8] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[9] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.