Deep Learning - Assignment 1

Davide Belli
11887532
University of Amsterdam
davide.belli@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Question 1.1 a)
Using $diag(x)$ notation to refer to a square diagonal matrix having the element of vector x on its diagonal. Also, the division between two vectors is to mean an element-wise division of their elements: $-\frac{t}{x^{(N)}} = \left\{ -\frac{t_i}{x_i^{(N)}} \right\}_i^S$.

Cross-Entropy

$$\frac{\partial L}{\partial x_i^{(N)}} = -\frac{\partial \sum_i^{d_N} t_i \log x_i^{(N)}}{\partial x_i^{(N)}} = -\frac{t_i}{x_i^{(N)}} \tag{1}$$

$$\frac{\partial L}{\partial x^{(N)}} = -\frac{t}{x^{(N)}} \tag{2}$$

Softmax

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \begin{cases} i \neq j \rightarrow \frac{-\exp \tilde{x}_i^{(N)} \exp \tilde{x}_j^{(N)}}{\left(\sum_i \exp \tilde{x}_i^{(N)}\right)^2} \\ i = j \rightarrow \frac{\exp \tilde{x}_i^{(N)}}{\sum_i \exp \tilde{x}_i^{(N)}} - \frac{\exp \tilde{x}_i^{(N)} \exp \tilde{x}_i^{(N)}}{\left(\sum_i \exp \tilde{x}_i^{(N)}\right)^2} \end{cases} \tag{3}$$

$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = diag(x^{(N)}) - x^{(N)} x^{(N)^T} \tag{4}$$

ReLU

$$\frac{\partial x^{(l)}}{\tilde{x}^{(l)}} = \frac{\partial \max(0, \tilde{x}^{(l)})}{\tilde{x}^{(l)}} = diag(\mathbb{1}_{(\tilde{x}^{(l)} > 0)}) \tag{5}$$

Linear

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} = \frac{\partial (W_{i,:}^l x^{(l-1)} + b_i^l)}{\partial x_j^{(l-1)}} = \frac{\partial W_{ij}^l x_j^{(l-1)}}{\partial x_j^{(l-1)}} = W_{ij}^l \tag{6}$$

$$\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} = W^l \tag{7}$$

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^l} = \frac{\partial W_{ij}^l x_j^{(l-1)} + b_i^l}{\partial W_{jk}^l} = \mathbb{1}_{(i=j)} x_k^{(l-1)} \tag{8}$$

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial W^l x^{(l-1)} + b^{(l)}}{\partial b^{(l)}} = \mathbb{I} \tag{9}$$

Question 1.1 b)

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \Big(diag(x^{(N)}) - x^{(N)} x^{(N)^T} \Big) \tag{10}$$

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} diag(\mathbb{1}_{(\tilde{x}^{(l)}>0)}) \tag{11}$$

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{l+1} \tag{12}$$

$$\frac{\partial L}{\partial W^l} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^l} = \sum_i \frac{\partial L}{\partial \tilde{x}_i^{(l)}} \frac{\partial \tilde{x}_i^{(l)}}{\partial W^l} = \frac{\partial L}{\partial \tilde{x}^{(l)}}^T x^{(l-1)^T} \tag{13}$$

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \tag{14}$$

Question 1.1 c)

Starting from the last layer (first in the chain of backpropagation), the Cross Entropy Loss is averaged across the batch to obtain a final scalar loss for the whole input batch. As a consequence, the gradients between the loss and network output have to be rescaled by dividing them by the batch size in order to account for this averaging. In the last layer of the network, the softmax, we have that our equations for backpropagation expand to 3d arrays. It is important to notice that multiplication are not done along the batch dimension, meaning that the gradients are propagated independently through the input elements in the batch without affecting each other. This is done practically using dimension broadcast in python before performing dot products between 3d arrays. Similarly for ReLU layers, the gradient for an activation with respect to a neuron is zero if they are not the same element in the batch, thus the matrix is 3d diagonal. Finally, for the linear layer we have that the bias and weights gradients are computed summing over the batch dimensions to have matrix with the same shape as the actual parameters. On the other side, the gradients with respect to the inputs keep the additional dimension for the batch, to be backpropagated to further layers in the computational graph The math and code behind this explanation can be better described by considering our python NumPy implementation.

1.2 NumPy implementation

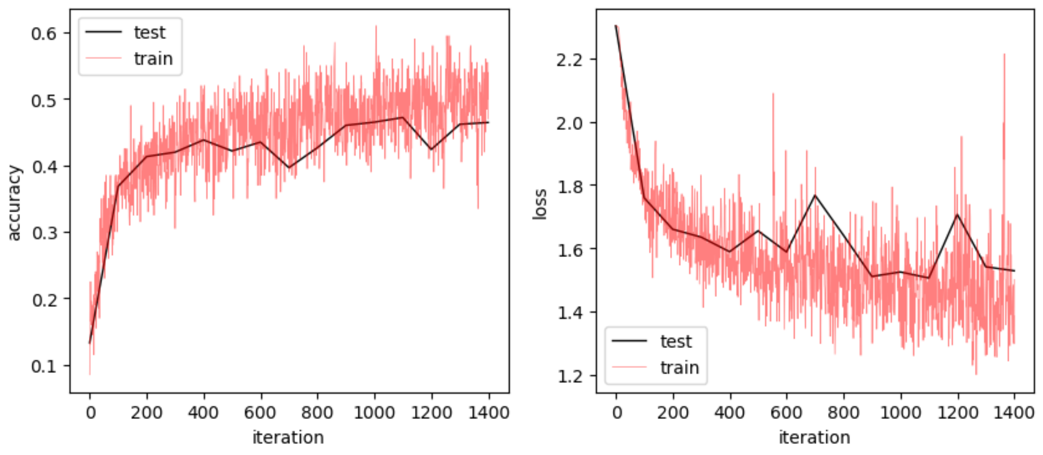
Question 1.2)

Our NumPy implementation was tested on various parameter settings. With the default settings, our best accuracy in the test set is 0.4720. For comparison, the accuracy in the last batch of the train set is 0.4700, which tells us that the model is not overfitting too much. This idea is confirmed looking at the plots of the accuracy and loss, where the training curves are only slightly better than the test ones. In Fig. 1.2 we show the training and test curves for accuracy and loss with this settings.

2 PyTorch MLP

Question 2

We included many modification in our initial implementation aiming to solve different problems. for different combinations of them we applied an automated grid search executed through a shell script



to find the best hyper-parameters. Here we list and explain which and how different modifications change the performance of our model, while later on we focus on the best configuration found so far. In the *out/* subdirectory of our project we include logs and plots for some sets of experiment we ran with various model configurations.

Linear Layers We experiment with a different number and size of linear layers. We started with introducing few small layers (of 10 to 50 neurons each), increasing then the number (e.g. two layers of 500 neurons) and depth (the final configuration has four layers of $1000, 800, 600, 400$ neurons. We also tried changing the initialization of the network from the PyTorch default. For example, we used the one previously implemented in NumPy, with mean 0 and std ranging between0.00001 and 0.1, and with biases set to zero. This changes resulted in similar or worse results in comparison to the default PyTorch implementation.

Learning rate We tried with different ranges of learning rate also in combination with different optimizers. We noticed how SGD usually works better with lower learning rates around 10^{-5}, while Adam performs significantly better around 10^{-4}. In general we tried more than 10 values ranging from 10^{-7} to 10^{-2}.

Batch Size We tried the following batch sizes: $[10, 20, 50, 100, 200, 300, 500, 600, 700, 800, 900, 1000, 1500, 2000, 3000, 4000, 5000]$. In general, lower batch size values didn't allow our model to converge to good optima, often not achieving even 0.4 accuracy regardless the rest of the configuration. Larger batch sizes were found to be better in our case, not being large enough to lose the noisy component in the stochastic update step that would result in getting stuck in local minimum points.

Optimizer We tried using both SGD and Adam optimizers. For the former, we found that including a momentum ranging between 0.8 and 0.9 was the best setting. Including Nesterov momentum didn't improve our results. We also tried to apply weight decay, but even with low decay values this configuration didn't help. For Adam, we found that the default values for betas were performing good enough in our case.

Regularization Term We tried including a regularization term in the loss, computing the L1 or L2 norms on the weight parameters. Since the norm itself is much larger than the Cross Entropy Loss, we used an alpha parameter to rescale it in the total loss. $\alpha = 10^{-4}$ resulted in a good tradeoff between the two components of the loss.

Dropout A very useful component we included in our final model was the Dropout layer after the ReLU layers. Once again, we applied grid search on this parameter in a range between 0 and 0.5 and found values ranging between 0.05 and 0.1 as the most effective ones.

Batch Normalization Finally, we tried applying Batch Normalization layers after the linear layers. This modification seem to yield worse results, so we decided to not include it in the final model.

Number of Iterations This is not really a modification of the model, but obviously we want our model to fully employ the training data to optimize the training. As we would expect, the model improves drastically in the first iterations while slowly converge to little to no improvements in the last part of the training. In our experiments we see that after 10,000 iterations using batches of 3000 elements there is almost no improvement.

In conclusion, our final model uses the following configuration:

Linear Layers 4 layers of size $[1000, 800, 600, 400]$ with the default PyTorch initialization.

Learning rate $\eta = 10^{-4}$

Batch Size 3000

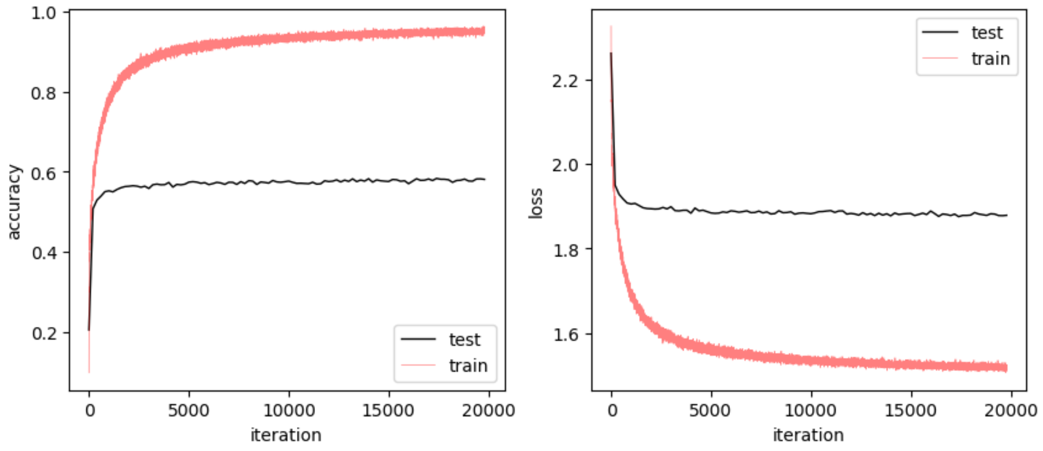
Optimizer Adam, with betas = $[0.9, 0.999]$

Regularization Term L2 Norm on linear layer weights, weighted with $\alpha = 10^{-4}$ in the total loss

Dropout Yes, with probability $p = 0.1$

Batch Normalization None

Number of Iterations $20,000$



In figure 2 we show the training and test curves for accuracy and loss using our PyTorch fine-tuned implementation. The best test accuracy we reach using the final model equals to 0.5833. We notice how the model is extremely overfitting on the training data. However, although tuning some hyperparameters in dropout, weight decay and batch normalization helped reducing this overfitting, the results in the test set dropped significantly. For this reason, we kept this configuration as our final best model. With a more in-depth search in the hyperparameter space we probably would have been able to find a configuration resulting in less overfitting and at the same time improvement (or at least no decrease) in the performance on the test set.

3 Custom Module: Batch Normalization

3.1 Automatic differentiation

Question 3.1
See python code.

3.2 Manual implementation of backward pass

Question 3.2 a)

$$\left(\frac{\partial L}{\partial \beta}\right) = \sum_s \sum_j \frac{\partial L}{\partial y_j^{(s)}} \frac{\partial y_j^{(s)}}{\partial \beta} = \sum_s \frac{\partial L}{\partial y^{(s)}} \tag{15}$$

$$\Big(\frac{\partial L}{\partial \gamma}\Big) = \sum_s \sum_j \frac{\partial L}{\partial y_j^{(s)}} \frac{\partial y_j^{(s)}}{\partial \gamma} = \sum_s \frac{\partial L}{\partial y^{(s)}} \hat{x} \tag{16}$$

Notice that every neuron in the batch normalization layer is not dependent on other neurons, regardless which element on the batches we are considering, namely: $\frac{\partial y_j^{(s)}}{\partial x_i^{(r)}} = 0$ for each $i \neq j$

$$\Big(\frac{\partial L}{\partial x}\Big)_i^r = \sum_s \sum_j \frac{\partial L}{\partial y_j^{(s)}} \frac{\partial y_j^{(s)}}{\partial x_i^{(r)}} = \sum_s \frac{\partial L}{\partial y_i^{(s)}} \frac{\partial y_i^{(s)}}{\partial x_i^{(r)}} \tag{17}$$

Applying the derivative quotient rule and chain rule in the numerator part:

$$\begin{aligned} \frac{\partial y_i^{(s)}}{\partial x_i^{(r)}} &= \frac{\partial}{\partial x_i^{(r)}} \Big(\gamma_i \frac{x_i^{(s)} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} - \beta_i \Big) = \\ &= \gamma_i \left(\frac{\sqrt{\sigma_i^2 + \epsilon} \Big(\mathbb{1}_{(r=s)} - \frac{1}{B} \Big) - \Big(x_i^{(s)} - \mu_i \Big) \frac{x_i^{(r)} - \mu_i}{B\sqrt{\sigma_i^2 + \epsilon}}}{\sigma_i^2 + \epsilon} \right) \end{aligned} \tag{18}$$

$$\begin{aligned} \Big(\frac{\partial L}{\partial x}\Big)_i^r &= \sum_s \frac{\partial L}{\partial y_i^{(s)}} \frac{\partial y_i^{(s)}}{\partial x_i^{(r)}} = \\ &= \sum_s \frac{\partial L}{\partial y_i^{(s)}} \gamma_i \left(\frac{\sqrt{\sigma_i^2 + \epsilon} \Big(\mathbb{1}_{(r=s)} - \frac{1}{B} \Big) - \Big(x_i^{(s)} - \mu_i \Big) \frac{x_i^{(r)} - \mu_i}{B\sqrt{\sigma_i^2 + \epsilon}}}{\sigma_i^2 + \epsilon} \right) = \\ &= \frac{\gamma_i}{\sqrt{\sigma_i^2 + \epsilon}} \left[\frac{\partial L}{\partial y_i^{(r)}} - \frac{1}{B} \sum_s \frac{\partial L}{\partial y_i^{(s)}} - \frac{\hat{x}_i^r}{B} \sum_s \Big(\frac{\partial L}{\partial y_i^{(s)}} \hat{x}_i^{(s)} \Big) \right] \end{aligned} \tag{19}$$

In matrix form (note that element-wise operations and dimension broadcast are considered to be used in the operations of this equation, check the Python implementation for more details):

$$\Big(\frac{\partial L}{\partial x}\Big) = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \left[\frac{\partial L}{\partial y} - \frac{1}{B} \sum_s \frac{\partial L}{\partial y^{(s)}} - \frac{\hat{x}}{B} \sum_s \Big(\frac{\partial L}{\partial y^{(s)}} \hat{x}^{(s)} \Big) \right] \tag{20}$$

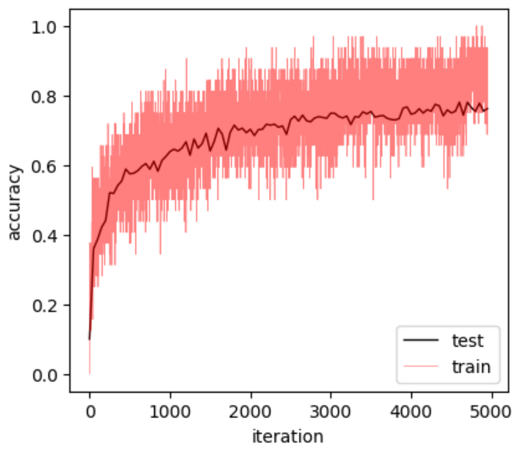
Question 3.2 b)
See python code.

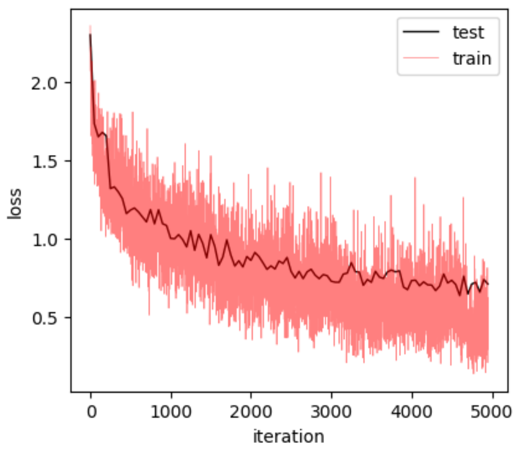
Question 3.2 c)
See python code.

4 PyTorch CNN

Question 4

In figure 4 we show the training and test curves for accuracy and loss. The best accuracy on the test set we found using the CNN architecture based on VGG-Net is 0.7805. We notice a drastic increase in performance with respect to DNNs explored in the previous sections. This behavior is well known, and it's due to the fact that CNNs are much better in capturing the 2D aspect intrinsic in images. In little more detail, the chain of convolutional layers in our network enables our model to detect a wider set of spatial features from high-level objects to basic low-level components and shapes like edges, corners and circular structures. This is done through convolutional filters (3×3 matrices in our case), which are learned from scratch during the training and result in an activation when the corresponding shape is present as input from the previous layer. Looking at the plots and comparing the training and





test curves, we see how the model doesn't seem to be overfitting a lot. We also notice that, even if improvements are relatively small, convergence is probably not reached yet. A longer training and an accurate tuning of the network's parameter would certainly improve even further the results obtained with this network.