Deep Learning 2018 - Assignment 1

David Rau (11725184)
University of Amsterdam
david.rau@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

Question 1.1 a

$$\frac{\partial L}{\partial x_i^{(N)}} = \frac{\partial - \log x^{(N)}_{\text{argmax}_t}}{\partial x_i^{(N)}} = \frac{1}{x_i^{(N)}} \mathbb{1}(i = \text{argmax}_t) \tag{1}$$

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\partial \frac{\exp(\tilde{x}_i^{(N)})}{\sum_{k=1}^{d_N} \exp(\tilde{x}_k^{(N)})}}{\partial \tilde{x_j}^{(N)}} = \frac{\tilde{x_i}^{(N)} \sum_{k=1}^{d_N} \exp(\tilde{x}_k^{(N)}) - \tilde{x_j}^{(N)} \tilde{x_i}^{(N)}}{\left(\sum_{k=1}^{d_N} \exp(\tilde{x}_k^{(N)}) \right)^2}$$

$$= \frac{\tilde{x_i}^{(N)}}{\sum_{k=1}^{d_N} \exp(\tilde{x}_k^{(N)}} \left(1 - \frac{\exp(\tilde{x}_j)^{(N)}}{\sum_{k=1}^{d_N} \exp(\tilde{x}_k^{(N)}} \right) = \text{Softmax}(\tilde{x}_i^{(N)}) \big(\mathbb{1}(i = j) - \text{Softmax}(\tilde{x}_j^{(N)}) \tag{2}$$

$$\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}} = \frac{\partial \max(0, x_i^{(l)})}{\partial \tilde{x}_j^{(l<N)}} = \mathbb{1}(i = j) \mathbb{1}(x_j^{(l)} > 0) \tag{3}$$

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} = \frac{\partial \sum_{k=1}^{L} W_{jk}^{(l)} x_i^{(l-1)} + b_i^{(l)}}{\partial x_j^{(l-1)}} = \mathbb{1}(i = j) W_{jk}^{(l)} \tag{4}$$

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \frac{\partial \sum_{k=1}^{L} W_{ik}^{(l-1)} x_i^{(l)} + b_i^{(l-1)}}{\partial W_{jk}^{(l)}} = \mathbb{1}(i = j) x_j^{(l)} \tag{5}$$

$$\frac{\partial \tilde{x_i}^{(l)}}{\partial b_i^{(l)}} = \frac{\partial \sum_{k=1}^{L} W_{ik}^{(l-1)} x_i^{(l)} + b_i^{(l-1)}}{\partial b_j^{(l)}} = \mathbb{1}(i = j) \tag{6}$$

Question 1.1 b

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \mathbb{1}_{N_D} x^{(N)} - x^{(N)} x^{(N)^T} \tag{7}$$

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l<N)}} \frac{\partial x^{(l<N)}}{\tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l<N)}} \cdot \mathbb{1}(x^{(l<N)} > 0) \tag{8}$$

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

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{(l)^T} \tag{9}$$

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = x^{(l)^T} \frac{\partial L}{\partial \tilde{x}^{(l)}} \tag{10}$$

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \mathbb{1}_{d_l} \tag{11}$$

Question 1.1 c

The equations for batch size $\neq 1$ remain mostly the same. However, as the derivative of the loss with respect to the previous layer is a matrix all derivatives have now to be taken with respect to the matrix. The loss of the Cross Entropy has now to be calculated with respect to the different batches (s):

$$\frac{\partial L_{\text{total}}}{\partial \tilde{x}^{(s,N)}} = \frac{\partial L_{\text{total}}}{L^s_{\text{individual}}} \frac{\partial L^s_{\text{individual}}}{\partial \tilde{x}^{(s,N)}} \tag{12}$$

where

$$\frac{\partial L_{\text{total}}}{L^s_{\text{individual}}} = \frac{1}{B} \tag{13}$$

1.2 NumPy implementation

Question 1.2

Since it wasn't explicitly stated which loss we were supposed to report I decided to show the batch loss and refer to in the following as 'loss'.

The accuracy and loss curves of the of the numpy implementation of the MLP can be found in Fig. 1. The model was trained with the default parameters. The top accuracy within the training was 47.25%. As it can be seen of the bottom of Fig. 1 the loss goes quickly down in the beginning and then stabilizes at around 1.5. The Accuracy curve of the whole validation set has a steep slope at the beginning and then saturates around 46%.

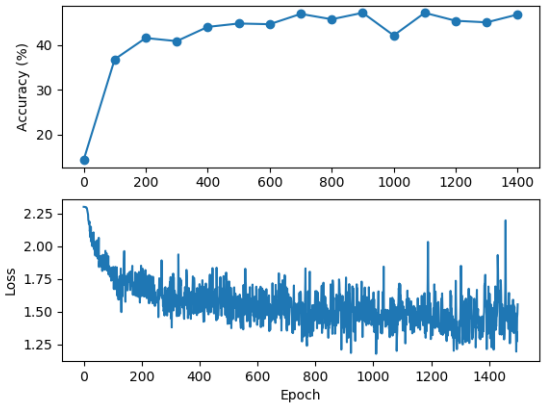

Figure 1: Accuracy (in %) and loss curves of the Numpy MLP for default values of parameters: batch size : 200, learning rate : 0.002 and dnn hidden units : 100. The top accuracy on the whole test set within the training was 47.24%.

2 PyTorch MLP

Question 2

First I ran the pytorch implementation with default configuration and reached 45.53% accuracy on the test set after 1500 epochs. Then I increased the hidden layers to two hidden layers with 100 units respectively to increase the model complexity in order to cope with the high dimensional input. This achieved a accuracy of 43.39% after 1500 epochs and 46.1% after 3500 epochs. Two hidden layers of size 200 achieved 47.82% after 3500 epochs and the accuracy stabilised around that value. In the following all models were therefore trained for 3500 epochs.
In order to cope with the problem that ReLUs die out in deeper networks I added a batch normalization layer before each linear layer. The accuracy went up to 54.03%. Because Batch normalization layers allow for bigger learning rates i changed the learning rate from 0.002 to 0.02 and retrained. This however couldn't improve the performance. As a last experiment I added a dropout layer with probability 0.2 to every linear layer this also couldn't improve the accuracy further probably because the learning is already saturated. However, by adding l1 weight regularization for the linear layers I could achieve the highest accuracy of 56.06%. The accuracy curve as well as the loss curve can be found in Fig. 2.

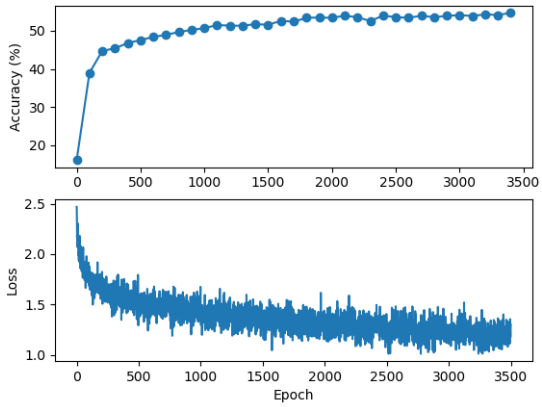

Figure 2: Accuracy (in %) and loss curves of the Pytorch MLP for parameters: batch size : 200, learning rate : 0.02, batch normalization, l1 regularization, dropout 0.2, with two hidden layers of size 200. The top accuracy on the whole test set within the training is 56.06%.

3 Custom Module: Batch Normalization

Question 3.2 a

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial \gamma_i \hat{x}_i^s + \beta_i}{\partial \gamma_j} = \sum_s \frac{\partial L}{\partial y_i^s}\mathbb{1}(i=j)\hat{x}_j^s \tag{14}$$

$$\left(\frac{\partial L}{\partial \beta}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial \gamma_i \hat{x}_i^s + \beta_i}{\partial \beta_j} = \sum_s \frac{\partial L}{\partial y_i^s}\mathbb{1}(i=j)1 \tag{15}$$

$$\left(\frac{\partial L}{\partial x}\right)_j^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial x_j^r} = \tag{16}$$

4 PyTorch CNN

Question 4

As the top accuracy (78%) on the validation set of the CNN in Fig. 3 suggests, CNNs are the appropriate network architecture to tackle this task. The CNN was trained with the default parameters:

batch size 32 and learning rate 0.0001. As it can be seen in Fig. 3 the batch loss stabilises below 1. Moreover, it could be interesting to examine the validation loss. If the validation loss would up while the batch loss goes still down the network is overfitting.

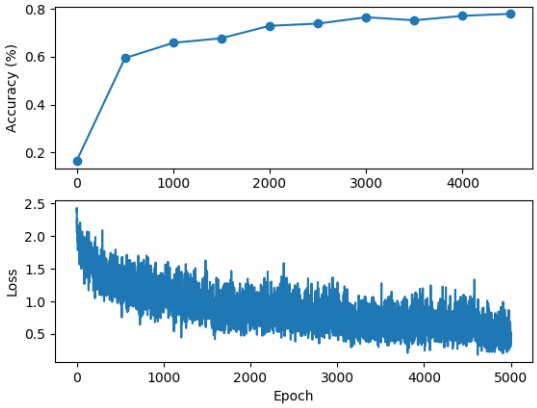

Figure 3: Accuracy (in %) and loss curves of the Pytorch ConvNet for default values of parameters: batch size : 32, learning rate : 0.0001. The top accuracy on the whole test set within the training is 78%.