Assignment 1: Implementing MLP using Numpy

Deep Learning - 20/11/2022

Ángel A. Gil - 2788988

1 Brief introduction

In this assignment, we will implement backpropagation for a simple feed-forward network, with a cross-entropy loss function. We will first implement it entirely from scratch, using only basic python primitives. Then, we will vectorize the forward and backward passes using NumPy and perform some experiments.

The report is divided into two main sections. The first one is devoted to responding to the first six questions included in the assignment, and the second one reveals the findings for the final part.

2 Working out the questions

Question 1.

We are given the following definitions for the softmax activation function and the cross-entropy loss:

$$y_i = \frac{exp\ o_i}{\sum_j exp\ o_j}, \qquad l = -log\ y_c$$

where x is a given instance with a true class c, o_i are the linear (non-activated) outputs of the network, y_i are the softmax-activated nodes (where i ranges over the number of classes), l is the loss and the logarithm is base e. The local derivatives are obtained as follows:

- $\frac{\partial l}{\partial y_i} = \frac{\partial \sum_j -log\ y_j}{\partial y_i} = -\sum_j \frac{\partial log\ y_j}{\partial y_i} = -\frac{1}{y_i}$
- $\frac{\partial y_i}{\partial o_j} = \frac{\partial \frac{exp\ o_i}{\sum_k exp\ o_k}}{\partial o_j} = \begin{cases} y_i(1-y_i) & \text{if } i = j \ ^{*1)} \\ -y_i y_j & \text{if } i \neq j \ ^{*2)} \end{cases}$

1) $\frac{exp\ o_i \cdot \sum_k exp\ o_k - exp\ o_i \cdot exp\ o_i}{\sum_k exp\ o_k \cdot \sum_k exp\ o_k} = \frac{exp\ o_i}{\sum_k exp\ o_k} \cdot \frac{\sum_k exp\ o_k - exp\ o_i}{\sum_k exp\ o_k} =$
$= y_i(1-y_i)$

2) $\frac{0 - exp\ o_i \cdot exp\ o_j}{\sum_k exp\ o_k \cdot \sum_k exp\ o_k} = -\frac{exp\ o_i}{\sum_k exp\ o_k} \cdot \frac{exp\ o_j}{\sum_k exp\ o_k} = -y_i y_j$

Question 2.

The derivative $\frac{\partial l}{\partial o_i}$ is determined as follows:

$$\frac{\partial l}{\partial o_i} = \frac{\partial l}{\partial y_j}\frac{\partial y_j}{\partial o_i} = \begin{cases} -\frac{1}{y_j} \cdot y_j(1-y_j) = y_j - 1 & \text{if } j = i \\ -\frac{1}{y_j} \cdot (-y_j y_i) = y_i & \text{if } j \neq i \end{cases}$$

If we work further with the expressions, we can get an unique formula for the derivative: $\boxed{\frac{\partial l}{\partial o_i} = y_i - \delta_i}$ where δ_i is 0 for all values except when $i = c$, where it adopts the value 1.

Working out this derivative is not strictly necessary because we can compute any global derivative as the multiplication of the local derivatives of which it is composed. This way, once we have obtained $\frac{\partial l}{\partial y_i}$ and $\frac{\partial y_i}{\partial o_j}$, we already have $\frac{\partial l}{\partial o_i}$ implicitly.

Question 3.

We now implement the network drawn in the assignment, perform one forward pass, up to the loss on the target value, and one backward pass (excluding the update of the weights, since it is straightforward). For the shake of brevity, I will only show a caption of the backward pass and the computed derivatives on the weights:

```
y_d  = [-1/y[i] for i in range(dim_out)]
o_d  = [y[i] for i in range(dim_out)]
o_d[c]-=1

v_d = np.zeros((dim_hid,dim_out))
h_d  = np.zeros(dim_hid)
for j in range(dim_out):
    for i in range(dim_hid):
        v_d[i][j] += o_d[j] * h[i]
        h_d[i] += o_d[j] * v[i][j]
b2_d = o_d
k_d  = [h_d[i]*h[i]*(1-h[i]) for i in range(dim_hid)]

w_d  = np.zeros((dim_in,dim_hid))
for j in range(dim_hid):
    for i in range(dim_in):
        w_d[i][j] = k_d[j] * x[i]
b1_d = k_d
```

```
v_d= [[-0.4404  0.4404] [-0.4404  0.4404] [-0.4404  0.4404]]
b2_d= [-0.5, 0.5]
w_d= [[ 0.  0.  0.] [-0. -0. -0.]]
b1_d= [0.0, 0.0, 0.0]
```

Note: all of the captions showed this report are exported from a Jupyter notebook.

Question 4.

The next step is to implement a training loop for our network and show that the training loss drops as training progresses. For this question, I have set the initial bias weights to 0 and the regular weights to random values from $\mathcal{N}(0,1)$. There are two important remarks to my implementation of this part:

- I have chosen stochastic gradient descent (SGD) but, for the plot of the loss curve, I have grouped the loss in batches of 100 iterations, using a variable called *update_step*. The reasoning behind this is that plotting the loss for each iteration resulted in a too noisy plot, where the outliers (the times where the classification strongly opted for a wrong label, resulting in a high loss) took over the graphic, making it unattractive.
- I opted for a learning rate α that adapts on the fly with a simple formula: $\alpha = \alpha * \frac{actual}{previous}$ where *actual* is the mean loss of the last 100 iterations and *previous*, that of the 100 before. This way, α decreases if the results keep getting better and it increases otherwise. It also makes use of the variable *update_step*.

The initial α was set to 0.1 and the number of epochs to 5. Figure 1 (both the prompt output and the graph) illustrates the results:

```
mean loss of epoch 1 : 0.1613
        mean loss of the last 100 iters: 0.0708

mean loss of epoch 2 : 0.1002
        mean loss of the last 100 iters: 0.0635

mean loss of epoch 3 : 0.0932
        mean loss of the last 100 iters: 0.0612

mean loss of epoch 4 : 0.0899
        mean loss of the last 100 iters: 0.0605

mean loss of epoch 5 : 0.0882
        mean loss of the last 100 iters: 0.0599

* alpha started as 0.1, but now alpha = 0.0011
```

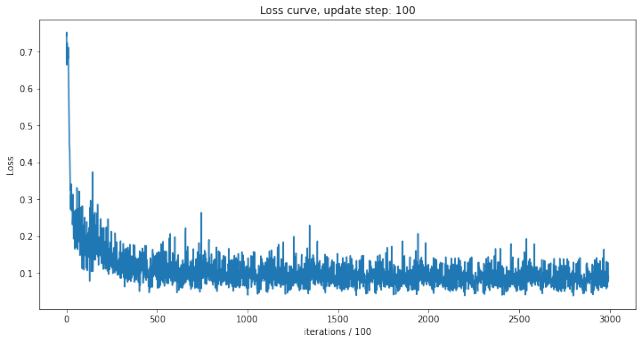

Figure 1: Loss curve with SGD after 5 epochs. The total number of iterations is 300k, so the loss vector has 3k entries.

Question 5.

In this question, we extend our neural network to dimensions 784, 300 and 10 for the input, hidden and output layer respectively. The main difference are that the input data has to be normalized beforehand and that we have to work out a vectorized version of the neural network. The former can be easily done by dividing every input vector by 255, since the values for MNIST data range from 0 to 255. The latter can be worked out taking advantage of *numpy*. In order to keep it brief, I will only show the results for this Question. In Q6 I will show how I worked out the vectorized gradient descent, although in that case it is slightly more complex because it is a batched version.

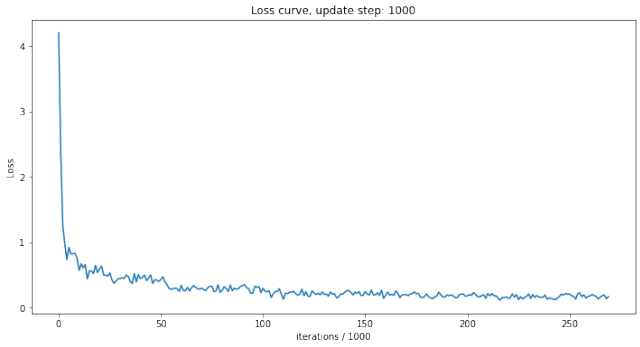

Figure 2: Loss curve with vectorized SGD after 5 epochs. The loss vector has 300k/1k = 300 entries.

```
mean loss of epoch 1 : 0.6391
        mean loss of the last 1000 iters: 0.3983

mean loss of epoch 2 : 0.2785
        mean loss of the last 1000 iters: 0.2438

mean loss of epoch 3 : 0.2107
        mean loss of the last 1000 iters: 0.2056

mean loss of epoch 4 : 0.1840
        mean loss of the last 1000 iters: 0.1930

mean loss of epoch 5 : 0.1644
        mean loss of the last 1000 iters: 0.1344

* alpha started as 0.1, but now alpha = 0.0067
```

For this question I decided to go for an even smoother loss curve, losing sensibility in exchange, since the update step is now 1000 (ten times higher than for the previous question). Alpha is still dynamic and the number of epochs has been set to 5 again. We can see that the tendency of the curve is pretty similar in both cases.

With regards to the loss, we can see that it is slightly higher than the loss for the binary case. This is expected since now there are 10 possible outputs instead of 2.

Question 6.

In this question, we are asked to Work out and implement the vectorized version of a batched forward and backward. For the shake of brevity, I will only show how to get to V^{∇} and b_2^{∇} and report the implementation of the backward pass:

$$V_{i,j}^{\nabla} = \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial o_j}\frac{\partial o_j}{\partial v_{ij}} = \Sigma_{ij} O_{jk}^{\nabla}\frac{\partial [Vh+b_2]_{ij}}{\partial V_{ij}} = \Sigma_{ij} O_{jk}^{\nabla}\frac{\partial [Vh]_{ij}}{\partial V_{ij}} = \Sigma_{ij} O_{jk}^{\nabla}\frac{\partial V_{ij}h_{ik}}{\partial V_{ij}} = \Sigma_{ij} O_{jk}^{\nabla} h_{ij} = h^T \cdot O^{\nabla}$$

Note: I seem to always do the maths with inverted dimensions for some reason, pardon me if the result should be indeed $V^{\nabla} = O^{\nabla} h$, I sticked to my representation.

For b_2 we proceed analogously and get to $b_2^{\nabla} = O^{\nabla} \cdot 1^T$
The backward pass would then look as follows:

```
# Backward pass (Mini)batch gradient descent
y_d  = -1/y
o_d  = y
for i in range(batch_size):
    o_d[i][c[i]]-=1
v_d = np.matmul(h.T,o_d)
b2_d = np.matmul(o_d.T,np.ones(batch_size))
h_d = np.matmul(o_d,v.T)
k_d = h_d * h * (1-h)
w_d = np.matmul(x.T,k_d)
b1_d = np.matmul(k_d.T,np.ones(batch_size))

# updating the weights. We want the average of the gradients.
w  -= alpha * w_d  / batch_size
b1 -= alpha * b1_d / batch_size
v  -= alpha * v_d  / batch_size
b2 -= alpha * b2_d / batch_size
```

To end this question, Figure 3 reports the results obtained when training the neural network with a minibatch gradient descent of size 50. Alpha has been set to 0.1 static this time, and the number of epochs is 20.

```
mean loss of epoch 1 : 1.6446
        mean loss of the last 50 iters: 0.8350

mean loss of epoch 5 : 0.3680
mean loss of epoch 10 : 0.2264
mean loss of epoch 15 : 0.1656
mean loss of epoch 20 : 0.1301
        mean loss of the last 50 iters: 0.1104
```

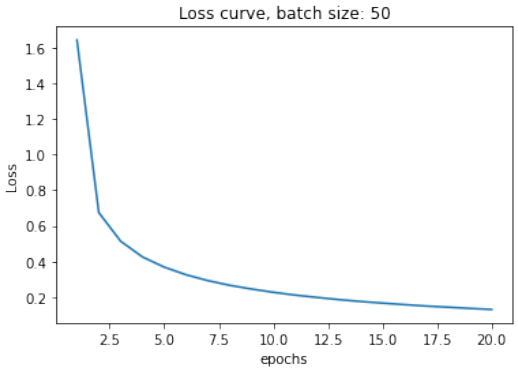

Figure 3: Loss curve with minibatch gradient descent for 20 epochs.

Although there are 4 times more epochs, the time required for the neural network to finish training was less than with the SGD of Q5. The mean loss of the final epoch is also lower, which means that it probably outperforms the stochastic version. In order to be able to confirm this claim, we should run a statistical test with multiple runs, but that is not the objective of this assignment.

3 Final training and conclusions

In this final section we conduct the final experiments, perform various analysis and choose a final hyper-parameter set for our neural network. In all of the subsections, we have used SGD and we have limited the training to 5 epochs. In this section I also implemented the method $shuffle$ in order to get a random ordered vectors $(xtrain, ytrain)$ for each epoch.

Question 7.1.

Although running the experiment multiple times can yield slightly different results, Figure 4 is a good representative of what happens.

The validation set starts with a better mean loss because we compute it after training the neural network for that epoch. After that, the mean for the training set gets better than that of the validation set, since our neural network gets more specialized on the particular training set, which can result on not trustworthy results.

Furthermore, the validation set spikes slightly in the final epoch for some runs (although not in this particular graph) due to an overspecialization of the network on the training set.

```
Epoch 1 , mean loss of training data: 0.8518 | mean loss of validation data: 0.4219
Epoch 2 , mean loss of training data: 0.3485 | mean loss of validation data: 0.4075
Epoch 3 , mean loss of training data: 0.2298 | mean loss of validation data: 0.2392
Epoch 4 , mean loss of training data: 0.1847 | mean loss of validation data: 0.2367
Epoch 5 , mean loss of training data: 0.1413 | mean loss of validation data: 0.1796
```

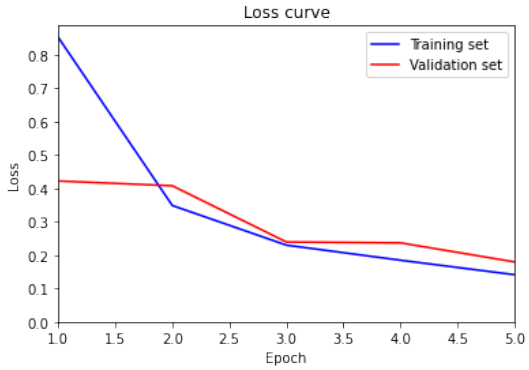

Figure 4: Training set vs Validation set

Question 7.2.

Instead of making the plot for each iteration (for the shake of visualization as spotted in Question 4), I have made use of the variable $update\_step$ once again. I chose to use a step size of 1000 and 100 in order to be able to compare. Figure 5 shows the average and a standard deviation of the loss for 3 runs of the neural network with the dynamic α initialized in 0.1.

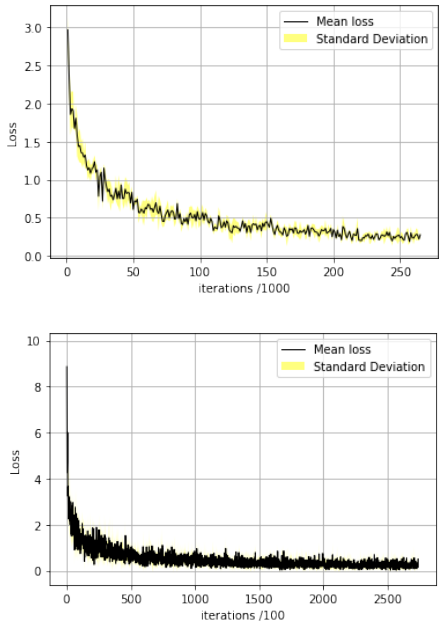

Figure 5: Mean and std for loss grouped by 1000 (upper) and 100 (lower) iterations.

As we can see, the standard deviation is not really high when the loss is grouped by 1000 iterations. For a step size of 100 the difference is hard to tell because of the fluctuation in the loss vector. This problem would increase the lesser the step size because the fluctuation in the (mean) loss vector would get higher. In general,

the standard deviation could be high at the beginning, since for a particular iteration and training point the NN might have totally different outcomes but, overall, the loss curve will behave similarly independently of the random initialisation, thus decreasing the variability towards the end of the training.

Question 7.3.

The set of 6 different α chosen for the experiment has been $\alpha = [0.1, 0.03, 0.01, 0.003, 0.001, 0.0001]$. The average results after 3 runs with each value are the following

```
alpha =    0.1. Mean loss training data: 0.1653 | mean loss validation data: 0.2278
alpha =   0.03. Mean loss training data: 0.0847 | mean loss validation data: 0.1911
alpha =   0.01. Mean loss training data: 0.1377 | mean loss validation data: 0.2346
alpha =  0.003. Mean loss training data: 0.2910 | mean loss validation data: 0.3656
alpha =  0.001. Mean loss training data: 0.5329 | mean loss validation data: 0.5392
alpha = 0.0001. Mean loss training data: 1.5868 | mean loss validation data: 1.3857
```

α yields the best results with the values 0.01 and 0.03, being the latter the best performing among the six of them. $\alpha = 0.1$ seems to be too aggressive for this architecture and data size, while the smaller values are too slow.

Question 7.4.

As spotted in the previous Question, the most suitable α from the analyzed set seems to be 0.003. Figure 6 shows the accuracy obtained through the 5 epochs for both the training and the validation set. The best accuracy is found in the last epoch for both the training and validation sets, which means that there is still room for improvement. These are 96.79% for the training data and 96.32% for the validation set.

```
Epoch 1 . Accuracy on training set: 87.92% | Accuracy on validation set: 93.83%
Epoch 2 . Accuracy on training set: 93.67% | Accuracy on validation set: 92.19%
Epoch 3 . Accuracy on training set: 95.11% | Accuracy on validation set: 95.78%
Epoch 4 . Accuracy on training set: 96.05% | Accuracy on validation set: 95.44%
Epoch 5 . Accuracy on training set: 96.79% | Accuracy on validation set: 96.32%
```

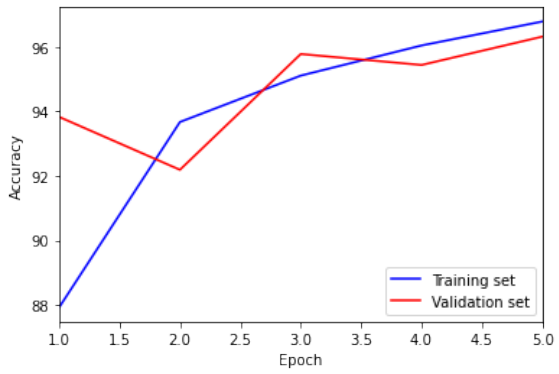

Figure 6: Accuracy obtained for $\alpha = 0.03$.