Assignment 1

Michele Vannucci (2819493)

October 2024

The code used to obtain the results of this report can be found here[1].

Answers

Part 1

question 1 Having defined the loss as

$$\text{loss} = \sum_i l_i \qquad\qquad l_i = \begin{cases} -\ln y_i & \text{if } c = i \\ 0 & \text{otherwise.} \end{cases}$$

The partial derivative $\frac{\partial \text{loss}}{\partial y_j}$ results in:

$$\frac{\partial \text{loss}}{\partial y_j} = \sum_i \frac{\partial l_i}{\partial y_j} = \begin{cases} -\frac{1}{y_j} & \text{if } c = j \\ 0 & \text{otherwise.} \end{cases} \quad \text{since: } \frac{\partial l_i}{\partial y_j} = \begin{cases} -\frac{1}{y_j} & \text{if } c = j = i \\ 0 & \text{otherwise.} \end{cases}$$

Using the multivariate chain-rule the derivation for $\frac{\partial \text{loss}}{\partial o_i}$ is as follows:

$$\frac{\partial \text{loss}}{\partial o_i} = \sum_j \frac{\partial \text{loss}}{\partial y_j} \cdot \frac{\partial y_j}{\partial o_i} \quad \rightarrow \quad \frac{\partial \text{loss}}{\partial o_i} = \frac{\partial \text{loss}}{\partial y_c} \cdot \frac{\partial y_c}{\partial o_i} \tag{1}$$

Now we have to distinguish two cases based on which o_i node we are calculating the derivative with respect to:

- If $i = c$, the derivation is as follows:

$$\begin{aligned} \frac{\partial \text{loss}}{\partial o_i} &= -\frac{1}{y_c} \cdot \frac{\partial \left[\frac{\exp o_i}{\sum_j \exp o_j} \right]}{\partial o_i} \\ &= -\frac{1}{y_c} \cdot \frac{\exp o_i \cdot \sum_j \exp o_j - \exp o_i \cdot \exp o_i}{(\sum_j \exp o_j)^2} \\ &= -\frac{1}{y_c} \cdot \frac{\exp o_i \cdot (\sum_j \exp o_j - \exp o_i)}{(\sum_j \exp o_j)^2} \\ &= -\frac{1}{y_c} \cdot y_c \cdot (1 - y_c) \end{aligned} \tag{2}$$

  From which we can say, more generally, $\frac{\partial y_i}{\partial o_i} = y_i \cdot (1 - y_c)$

[1] https://github.com/michelexyz/deeplearning_assignments

- While if $i \neq c$, the derivation is as follows:

$$\begin{aligned}\frac{\partial \text{loss}}{\partial o_i} &= -\frac{1}{y_c} \cdot \frac{\partial \left[\frac{\exp o_c}{\sum_j \exp o_j}\right]}{\partial o_i} \\ &= -\frac{1}{y_c} \cdot \frac{-\exp o_c \cdot \exp o_i}{(\sum_j \exp o_j)^2} \\ &= -\frac{1}{y_c} \cdot (-y_c) \cdot y_i \end{aligned} \tag{3}$$

  From which we can say, more generally, $\frac{\partial y_j}{\partial o_i} = -y_j \cdot y_i$, where $j \neq i$

question 2 Given the Equations 2 and 3 we can easily derive $\frac{\partial \text{loss}}{\partial o_i}$ cancelling out the y_c terms above and under the fraction line for both cases. By doing this we get:

$$\frac{\partial \text{loss}}{\partial o_i} = y_c - 1 \quad \text{if } i = c \qquad\qquad \frac{\partial \text{loss}}{\partial o_i} = y_i \quad \text{if } i \neq c$$

We don't necessarily need the formula of $\frac{\partial \text{loss}}{\partial o_i}$ because we already formulated symbolically its partial derivatives, $\frac{\partial \text{loss}}{\partial y_i}$ and $\frac{\partial y_i}{\partial o_j}$, which can then been computed numerically after the forward pass and multiplied together thanks to the chain rule to obtain $\frac{\partial \text{loss}}{\partial o_i}$.

question 3 I implemented the network in Python using the worked out the local derivatives, which are computed numerically at each stage of the network through list comprehension. The following is the code for the **forward pass**:

```
# first layer
k = [sum([x[i]*w[i][j] for i in range(len(x))]) + b[j] for j in
↪  range(len(b))]

# sigmoid layer
z = [1/(1+math.exp(-k[i])) for i in range(len(k))]

# second layer
o = [sum([z[i]*v[i][j] for i in range(len(z))]) + c[j] for j in
↪  range(len(c))]

# softmax layer
y = [math.exp(o[i])/sum([math.exp(o[j]) for j in range(len(o))]) for i in
↪  range(len(o))]

# cross entropy loss
loss = -sum([t[i]*math.log(y[i]) for i in range(len(t))])
```

And the code for the **backward pass**:

```
# derivative of loss function with respect to yi
dy = [(-1/y[i])*t[i] for i in range(len(y))]

# derivative of loss function with respect to oi
do = [sum([dy[j] * (y[i]*(1-y[i]) if j==i else -y[j]*y[i]) for j in
↪  range(len(dy))]) for i in range(len(o))]

# alternative way to calculate do
i_true = t.index(1) #index of true class
do_alt = [dy[i_true]  * (y[i]*(1-y[i]) if i==i_true else -y[i_true]*y[i])
↪  for i in range(len(o))]

assert do == do_alt

# derivative of loss function with respect to v
dv = [[do[j]*z[i] for j in range(len(do))] for i in range(len(z))]

# derivative of loss function with respect to ci
dc = [do[i]  for i in range(len(c))]

# derivative of loss function with respect to zi
dz = [sum([do[j]*v[i][j] for j in range(len(do))]) for i in range(len(v))]

# derivative of loss function with respect to ki
dk = [dz[i]*z[i]*(1-z[i]) for i in range(len(z))]

# derivative of loss function with respect to wi
dw = [[dk[j]*x[i] for j in range(len(dk))] for i in range(len(x))]

# derivative of loss function with respect to bi
db = [dk[i] for i in range(len(b))]
```

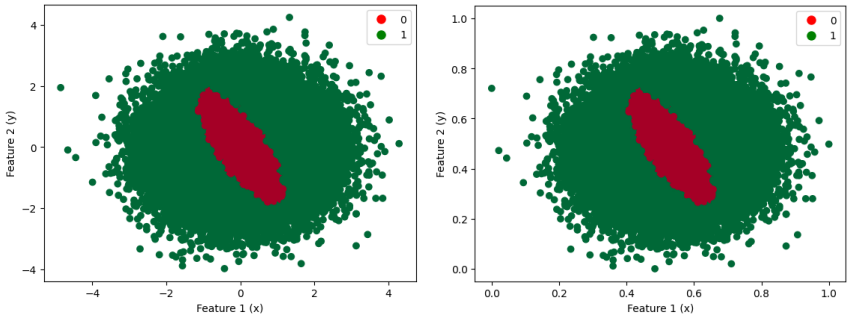
Here we can see that $\frac{\partial \text{loss}}{\partial o_i}$ can be computed also without performing the summation over all ∂y_j as we have seen in Eq. 1. Finally, the derivatives that we get after performing sequentially one backward and one forward pass are the following:

$$\left(\frac{\partial \text{loss}}{\partial v}, \frac{\partial \text{loss}}{\partial c}\right): \quad \left(\begin{bmatrix} -0.4404 & 0.4404 \\ -0.4404 & 0.4404 \\ -0.4404 & 0.4404 \end{bmatrix}, \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}\right)$$

$$\left(\frac{\partial \text{loss}}{\partial w}, \frac{\partial \text{loss}}{\partial b}\right): \quad \left(\begin{bmatrix} 0.0 & 0.0 & 0.0 \\ -0.0 & -0.0 & -0.0 \end{bmatrix}, \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}\right)$$

question 4 Plotting the synthetic data, we can observe in Fig. 1a that both features lie in the range $[-4; 4]$, to fix that we apply Min-max normalization to transpose the data in the $[0; 1]$ interval. Fig. 1b shows that the normalization maintains the proportions between the data points intact.

Finally, I run the data through the network using stochastic gradient descent for 30 epochs, after each epoch the data points were randomly shuffled. Fig. 2 shows the running mean loss of every epoch, we can see that after few epochs the loss then falls down and stabilizes on the optima.



(a) The original synthetic data (b) The normalized synthetic data

Figure 1: Plots of the synthetic data where the instances labeled with 1 (true class) are colored green and instances labeled with 0 are colored in red.

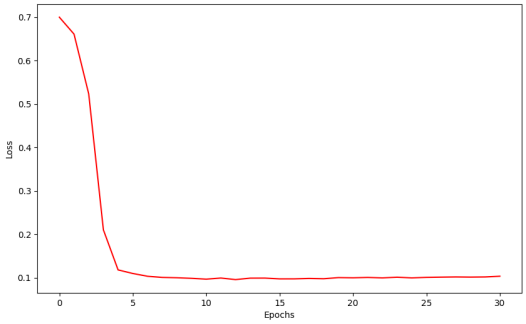


Figure 2: Running mean loss per epoch

question 5 For this question I normalized the MNIST data over all features since they are all pixel values in the same range as shown in Fig. 6 of the appendix, then I one-hot encoded the classes. Finally, I vectorized my implementation and applied stochastic gradient descent, in the appendix are more detail on the performance of the network.

question 6 A few changes to the code were required for the batched implementation, refer to the appendix for more details.

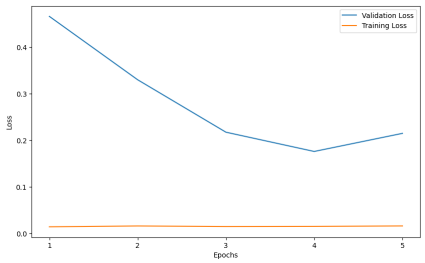
question 7 From the graph in Fig. 3a we can see the loss per epoch of both the training and the validation set over 5 epochs and a batch size, $b_s = 100$. We can observe that the training loss shrinks down after the first epoch already and stays around 0.01. On the other hand, the validation loss gets to its lowest at epoch 4 going slightly below 0.2. These results were obtained with a learning rate $\alpha = 0.1$.

This pattern and difference in performance for the training and validation set it's common in machine learning and it's explained by the fact that the model fits better to the data it has already seen, additionally, we can assume that after epoch 5 the network starts over-fitting as the validation loss seems to increase.

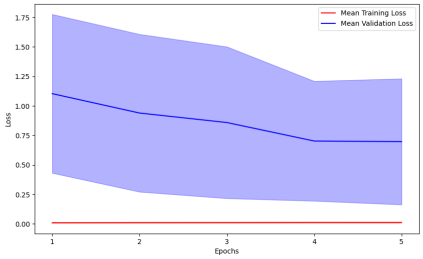
Finally, I run the same experiment (with $\alpha = 0.1$ and $b_s = 100$) with 5 different randomization seeds, to generate different weights each time. Figure 3b shows the results, we can see that the same pattern observed for the single run case mentioned above is present.

For point **3.** I trained the network by SGD with three different values of α : $\{0.1, 0.01, 0.001\}$. I chose these values since they have three different orders of magnitude . Figure 4 shows the results of this experiment.

Since $\alpha = 0.1$ seems to perform generally, this is the value I picked for the final model.



(a) Graph of the average loss per epoch over the entirety of the training and the validation set respectively.



(b) Graph of the average loss per epoch averaged out over 5 runs, the area around both the training and validation loss represent the standard deviation.

Figure 3: Graphs of the loss for the batched implementation, using a batch size of 100 and a learning rate $\alpha = 0.1$

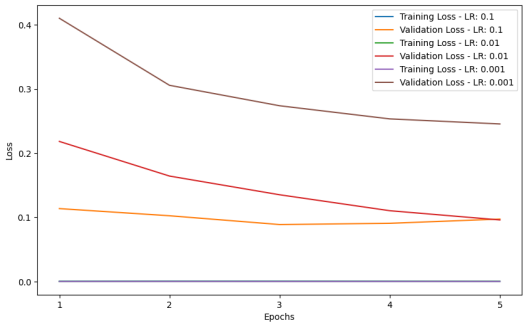


Figure 4: Comparison of the performance of the NN trained by SDG with different learning rates ($\{0.1, 0.01, 0.001\}$ and same initialization for the weights

For the latter I decide to use a batch size of 10 ($b_s = 10$) because even though the stochastic approach seemed to work better than the one with a larger batch size (as with $b_s = 100$, Fig. 3), calculating through batches is faster and seemed generally safer for the final test. Finally, I kept the number of epochs to 5 because all the graphs showed that the validation loss was at its lowest around that epoch. Figure 5 shows the resulting performance.

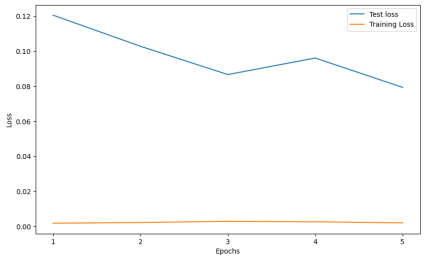
We have a final accuracy of $97,71\%$ and a loss that stabilizes at epoch 5 of 0.08. Fig 5a and 5b were generated after the same run, the test set has been used only once.

Appendix

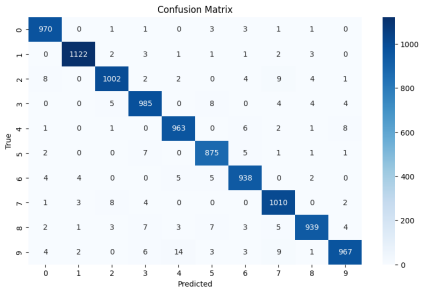
Question 5

Data visualization

Figure 6a shows the distribution of the pixel values of the MNIST dataset , we can see that there is peak at both ends of the histogram, this is explained by the fact that the numbers are mainly represented by white values (255) and the background by black values (0). All the values in between are present at the intersection between the two and represent different shades of gray. Aditionally, in Fig. 6b we can see the distribution of the 10 labels, except of a slightly higher frequency for class "1", all the classes have similar frequency therefore we don't have to apply any balancing.

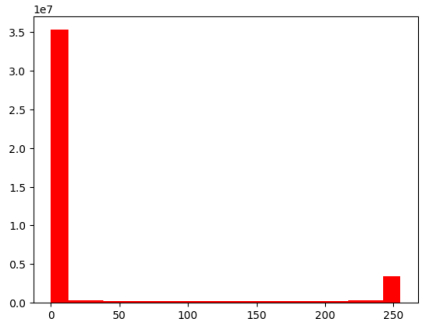


(a) Graph of the average loss per epoch over the entirety of the training and the test set respectively.

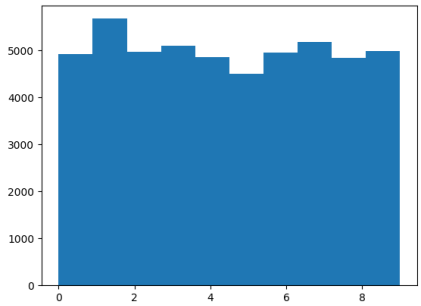


(b) Confusion matrix of the final model over the test data

Figure 5: Performances measures for the final model with $\alpha = 0.1$, $b_s = 10$ and $n_{epochs} = 5$



(a) Distribution of the pixel values of the MNIST dataset, all pixels of different positions have been aggregated together



(b) Histogram showing the distribution of the 10 classes of the MNIST dataset

Figure 6: Histograms of the MNIST data feature and labels respectively

Vectorized implementation

To vectorize the implementation, I used the numpy's `np.dot()` function because it has the same effect of `np.matmul()` for matrices of 2 dimensions and it's supposed to be faster (since also in the batched case my matrices won't go beyond the 2 dimensions, I used the same function there). The following is the code for the **forward pass**

```
def forward(self, X):

        assert X.shape[1] == self.W.shape[0], "X shape mismatch"
        k = np.dot(X, self.W) + self.b


        z = self.sigmoid(k)

        assert z.shape[1] == self.V.shape[0], "z shape mismatch"
        o = np.dot(z, self.V) + self.c


        y = self.softmax(o)
        return k, z, o, y
```

This function assumes that X is a 2d numpy matrix representing a horizontal vector, the same goes for the bias vectors. The code for the **backward pass**:

```
def backward(self, t, y, o, z, k, X):

        dy = (-1 / y)*t

        bool_t = t.astype(bool)

        do =  dy[bool_t] * (y*(1-y) *bool_t + (-y[bool_t]*y)*~bool_t)

        # derivative of loss function with respect to v
        dv = np.dot(z.T, do)

        assert dv.shape == self.V.shape, "dv shape mismatch"

        # derivative of loss function with respect to ci
        dc = do

        assert dc.shape == self.c.shape, "dc shape mismatch"

        # derivative of loss function with respect to zi
        dz = np.dot(self.V, do.T).T

        assert dz.shape == z.shape, "dz shape mismatch"

        # derivative of loss function with respect to ki
        dk = dz * z * (1 - z)

        dw = np.dot(X.T, dk)

        assert dw.shape == self.W.shape, "dw shape mismatch"

        db = dk

        return dw, db, dc, dv
```

Also this function assumes that all single-dimension vectors are horizontal, represented as 2d matrices with only one row. This was done to help visualize all the operations and to be consistent with the graph representation of the network, where the nodes of the same layer are distributed horizontally.

Question 6

For question 6 I had to make few changes to the code above since I could perform the operations in batches just by feeding an X matrix with multiple rows instead of just one (as it's done for SGD implementation). Doing that the gradients of the weights turn out to be the sum of all the gradients over the data points in the batch as expected.

I had to make few changes to the backward pass only. Firstly, I added a reshape function to `dy[bool_t]`, to make sure for it to be a bi-dimensional matrix for the calculation of $\frac{\partial \text{loss}}{\partial o_i}$:

```
do =  dy[bool_t].reshape(-1, 1) * (y*(1-y) *bool_t +
↪  (-y[bool_t].reshape(-1, 1)*y)*~bool_t)
```

Secondly, I summed the batched derivatives of nodes **k** and **o** along the first axis to get the sum of the derivatives of the single data points for the biases. The resulting gradient is then used to update them. The changed code within the backward pass is the following:

```
...
dc = do.sum(axis=0).reshape(1, -1)
...
db = dk.sum(axis=0).reshape(1, -1)
...
```

I also applied a reshape here to make sure the biases vectors are represented horizontally.

Question 7

To show better the training process during the first epoch, where the network sees the data for the first time, I plotted the running loss over the single datapoints using the SDG approach. The plot is visible in Figure 7.

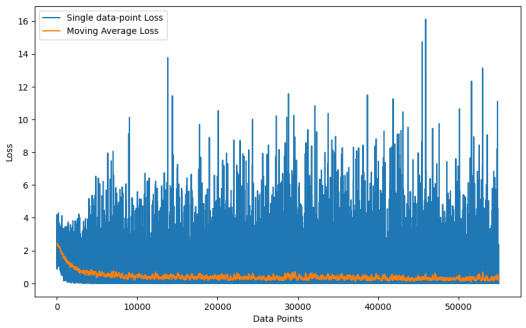


Figure 7: Loss values of the single data points within the first epoch of the training using SDG, with $\alpha = 0.01$.