MNIST Classification Using Multi-layered Perceptrons

Bùi Quang Long

February 2020

Contents

1 Formula

1.1 Definition

TODO : Chinh lai x, X, w, W, a, A clarify that we are using (mini) Batch GD L d^l d^L z z_i^l a a_i^l $x(+bias) = a0$ w First, let us define some mathematic symbols:

- We are analysing the process of only **ONE** sample from the training data, then make conclusion for the whole dataset. In this assignment, we will use batch Gradient Descent. Thus (x, y) can be considered as a sample and (X, Y) can be considered as training input and output.
- L: number of layers (includes hidden layers and output layer). In this assignment, $L = 2$.
- d^l: number of hidden units in layer l. In this assignment, $d^L = 10$.

- x: In this assignment we will consider x as a bias-added input. Hence, X is our bias-added training input.
- z^l: Values of layer l before activating.
- a^l: Values of layer l after activating. Thus,
  - $a^1 = x$
  - $z^l = w_l^T a^{l-1} \quad (*)$
  - $a^l = f(z^l) = f(w^{(l)T} a^{l-1}) \quad (**)$
  - a, z are vectors, whereas A, Z are matrices.
- w^l: weight matrix of layer l. Hence, W will stand for our set of weight matrices.

1.2 Forward Propagation

In this process, we calculate the values of each layer. For $l = 1..L$,

$$z^l = w_l^T a^{l-1}$$

$$a^l = f(z^l) = f(w^{(l)T} a^{l-1})$$

In this assignment, activation funcion f is:

- Sigmoid for $l = 1..L-1$: $\quad a_i^l = f(z_i^l) = \frac{e^{z_i^l}}{1+e^{z_i^l}}$
- Softmax for $l = L$: $\quad a_i^L = f(z_i^L) = \frac{exp(z_i^L)}{\sum_{k=1}^{d^L} exp(z_k^L)}$
  Since $exp(x)$ easily gets overflow when x reach higher values, we will use a **more stable version of softmax**, which is:

$$f(z_i) = \frac{exp(z_i - M)}{\sum_{k=1}^{K} exp(z_k - M)}, \quad where\ M = max(z_i), i = 1..K$$

1.3 Lost function (cross-entropy)

In this assignment, in order to measure distance between the predicted and the target, we will use a cross entropy - a common loss function for classification problems. To avoid misunderstanding, let us define J as the loss function instead of L:

$$J(y, a^L) = -\sum_{i=0}^{d^L} y_i log(a_i^L)$$

Here y is one-hot encoded label: $y = [0, 0, ...1, ...0]$, where $y_k = 1$ means the corresponding digit illustrated by that image is k. (Notice that $d^L = 10$ in this assignment)

1.4 Back Propagation

Our goal is to find set of weight matrices W via gradient descent with 2 steps in each iteration:

- Calculate output layer a^L and loss function J
- Calculate new set of weight maxtrices. Hence, for $l = 1..L$,

$$W^l_{new} = W^l - \eta \nabla_{W^l} J + \gamma(W^l - W^l_{old})$$

In order to derive $\nabla_{W^l} J$, we will take a look at **only one parameter** in one weight matrix. (W^l_{ij})

$$\frac{\partial J}{\partial w^l_{ij}} = \frac{\partial J}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{ij}} \quad (using \ \textbf{chain rule} \ and \ (*))$$

Let us denote $\frac{\partial J}{\partial z^l_j} = e^l_j$ because the **activation function is different** between output layer (softmax) and hidden layer (sigmoid). We obtain:

$$\frac{\partial J}{\partial w^l_{ij}} = e^l_j a^{l-1}_i \quad (***)$$

Now we will derive this in 2 cases : output layer and hidden layer.

1.4.1 Output layer

Now we take a look at the loss function J of **one** given sample in the dataset:

$$J(W) = J(W, x, y) = -\sum_{j=1}^{d^L} y_j log(a^L_j)$$

$$J(W) = -\sum_{j=1}^{d^L} y_j log(\frac{exp(z^L_j)}{\sum_{k=1}^{d^L} exp(z^L_k)}) = -\sum_{j=1}^{d^L} y_j z^L_j + log(\sum_{k=1}^{d^L} exp(z^L_k))$$

$$e^L_j = \frac{\partial J(W)}{\partial z^L_j} = -\sum_{j=1}^{d^L} y_j + \frac{exp(z^L_j)}{\sum_{k=1}^{d^L} exp(z^L_k)} = -y_j + a^L_j$$

Let us look back to equation $(***)$:

$$\frac{\partial J}{\partial W^L_{ij}} = e^L_j a^{L-1}_i = (-y_j + a^L_j) a^{L-1}_i$$

Now if we apply this to the whole dataset (using batch GD), we obtain:

$$\nabla_{W^L} J = \frac{\partial J}{\partial W^L} = A^{L-1}(A^L - Y)^T$$

1.4.2 Hidden layer

$$e_j^l = \frac{\partial J}{\partial z_j^l} = \frac{\partial J}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}$$

- $\frac{\partial a_j^l}{\partial z_j^l} = f'(z_j^l) = \frac{exp(z_j^l)}{(1+exp(z_j^l))^2}$ (because $a^l = sigmoid(z^l)$
  In order to simplify the function, we will keep using $f'(z_j^l)$.

- $\frac{\partial J}{\partial a_j^l} = \sum_{k=1}^{d^{l+1}} \frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial a_j^l} = \sum_{k=1}^{d^{l+1}} e_k^{l+1} w_{jk}^{l+1} = w_{j^{th}row}^{l+1} e^{l+1}$
  (Due to ($z_k^{l+1} = w_{k^{th}col}^{(l+1)T} a^l$), a_j^l contributes to all z_k^{l+1} with $k = 1..d^{l+1}$)

  - ($w_{k^{th}col}^{(l+1)T}$) here means the k^{th} col of W^{l+1}
  - ($w_{j^{th}row}^{(l+1)T}$) here means the j^{th} row of W^{l+1}

  Let us combine the above two together,

$$e_j^l = (w_{j^{th}row}^{l+1} e^{l+1}) f'(z_j^l)$$

  Let us look back to equation ($***$):

$$\frac{\partial J}{\partial w_{ij}^l} = e_j^l a_i^{l-1} = \left[(w_{j^{th}row}^{l+1} e^{l+1}) f'(z_j^l)\right] a_i^{l-1}$$

Now if we apply this to the whole dataset (using batch GD), we obtain:

$$\nabla_{W^l} J = \frac{\partial J}{\partial W^l} = A^{l-1}(W^{l+1}E^{l+1} \odot f'(Z^l))^T$$

2 Implementation, experiment and report

2.1 Notes for my implementation

- I use 5000 samples for experiments, thus our training set will have 4000 samples and our test set will have 1000 samples.
- I also implement a class called MyMLP with functions quite similar to MLP in sklearn.
- Most of the experiments I have scaled input data to the range [0,1]. For that reason, my bias added to the input is a vector with values equal to 0.1, not 1.0.

2.2 Different hidden dimension

Exp 1: I set $\eta = 0.1, \gamma = 0.9$ and number of epochs $= 200$ and study a list of different hidden dimension: $[100, 200, 500, 1000, 1500, 2000]$.

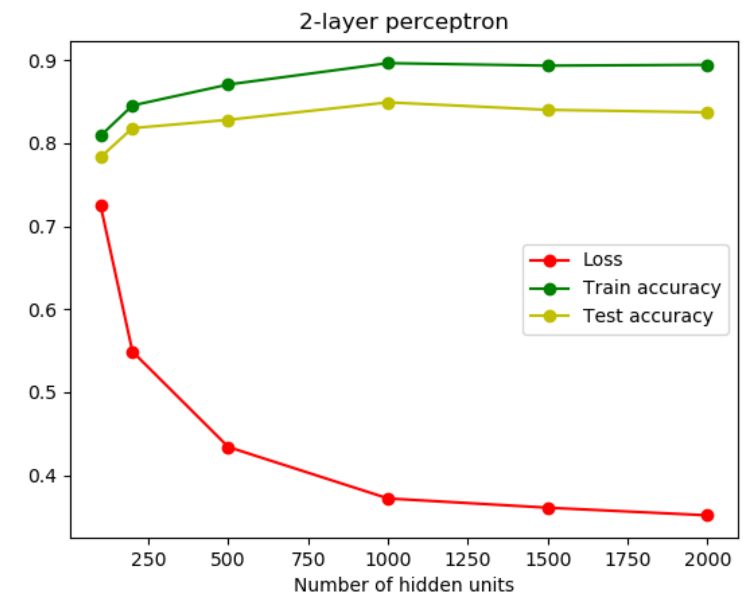

Figure 1: Data preparation

The number of hidden dimensions indicate how "**complex**" the parameters are used to learn the training data. It's good to have a complex enough set of parameters, but it will have a negative effect if it is too complex, which often called "**overfit**". In the experiment above, when the number of hidden units increase, we can see that the gap between the training accuracy and the test accuracy is larger and larger. It means our model can deal with the already known data very well, but worse when dealing with new data. To tackle overfitting problem, we can use **K-fold cross-validation** and the **"rule of thumb"** when choosing the number of hidden units.

2.3 Different epochs

Exp 2: I set $\eta = 0.1, \gamma = 0.9$ and number of hidden dimension $= 100$ and study a list of different epochs: $[50, 100, 200, 500, 2000]$.

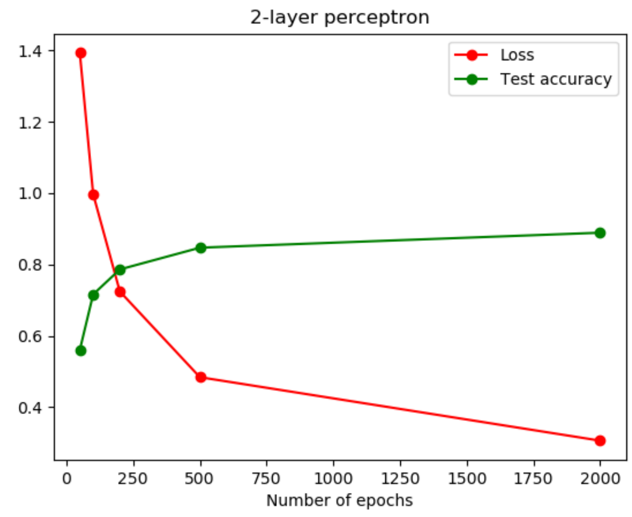

Figure 2: Data preparation

To me, hidden units and number of epochs are not specifically related to each other.

2.4 Different input scaling

Exp 3: I set $\eta = 0.1 \gamma = 0.9$, number of hidden dimension = 100, number of epochs = 100 and compare 2 cases: scaling the input to $[0, 1]$ range and keeping the input intact. The test accuracy for 2 cases are 72% and 70% respectively. The scaling process did affected the result and it will be clearer in larger dataset: Scaling the input to $[0, 1]$ will improve the model.

2.5 Different learning rate

Exp 4: I set $\gamma = 0.9$, number of hidden dimension = 100, number of epochs = 5000 and study a list of different learning rate: $[0.01, 0.05, 0.1, 0.5, 1]$.

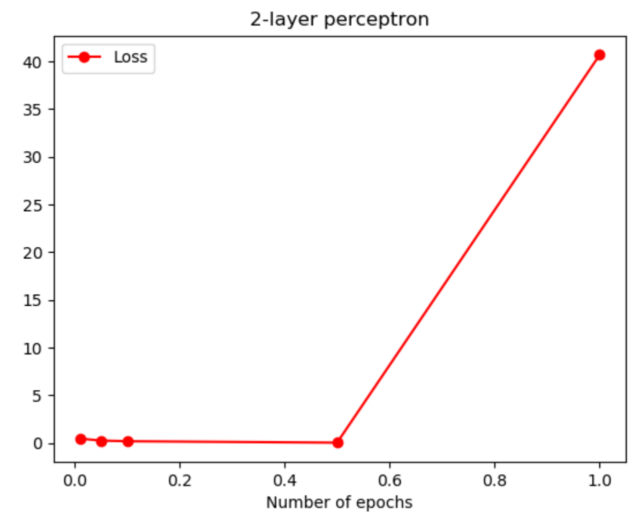

Figure 3: Data preparation

2.6 Different momentum factors

Exp 5: I set $\eta = 0.1$, number of hidden dimension $= 100$, number of epochs $= 200$ and study a list of different momentum: $np.linspace(0.0, 0.9, 10)$.

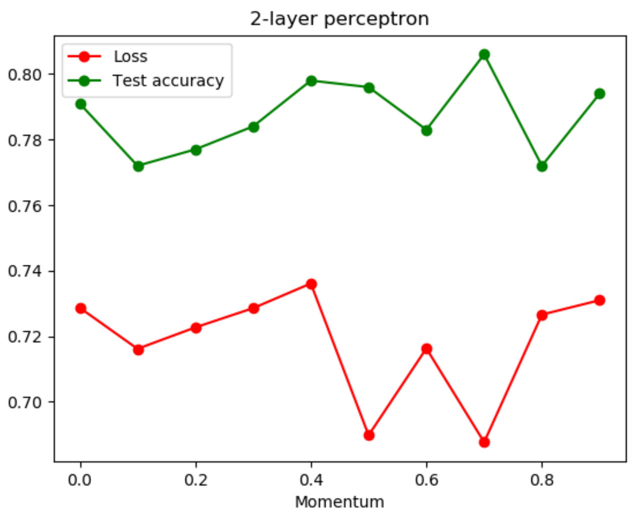

Figure 4: Data preparation

The momentum will affect our model:

- It will increase the learning speed thus lead to a reduction in the number of epochs. What makes it different from learning rate is that momentum make use of previous information.
- It will help the Gradient Descent "jump" over the local minima of the loss function, thus sometimes improves the result. If not, the result (here is accuracy score) doesn't change.