# 38616A - Implement a Neural Network

Natalie Pham

February 2023

## 1 Introduction

In this report, I am going to implement a single layer neural network for binary classification from scratch, then validate it using a PyTorch implementation. The results will be discuss in section 3. Beyond the single layer neural network, this report will also explain the steps to derive mathematical formula in the forward and back-propagation stages (Section 2.1 and 2.2, respectively) and extend the model to two-layers neural networks (Section 4).

## 2 Methods

To implement a single layer neural network from scratch, I implement the forward and backward functions for `ReLU`, `LinearMap`, `SoftmaxCrossEntropyLoss`. These functions are used to construct a single layer neural network model in `SingleLayerMLP`. The model will be train and validate on the provided dataset, with 500 records in the train set and 200 records on the test (validate) set. The structure of the single layer neural network is as follow:

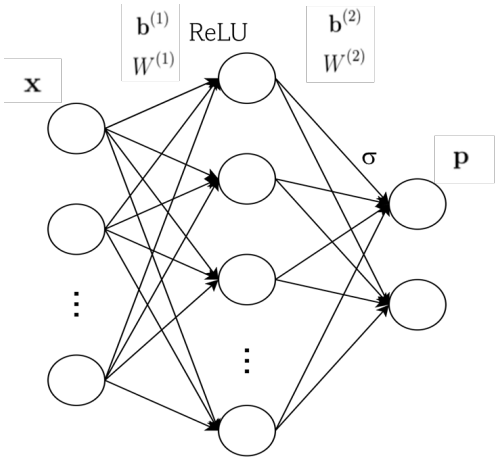


***Fig 1:*** *single layer neural network*

### 2.1 Mathematical Background

Denote $x \in \mathbb{R}^D$ as the input column vector, where $D = 10$ is the input dimension for the given sample dataset. Denote $b$ and $W$ the bias and weights terms for the linear transformation, where the superscript denotes the transformation index. There are $M$ classes ($M = 2$ in our example), so label vector $y \in \mathbb{R}^M$. Assuming there are $H$ hidden neurons.

For the single layer neural network, the input undergoes the following transformations in the forward pass. Firstly, the column vector $x$ goes through an linear transformation (`LinearMap`), followed by an element-wise `ReLU` function to add non-linearty:

$$l(x) = \text{Linear}_1(x) = W^{(1)}x + b^{(1)}$$

where (1) denotes the first hidden layer;

$$h = ReLU(x) = \max(0, x)$$

The output of *ReLU* is then feed to the next linear transfromation to get the logits $\tilde{x}$ :

$$\tilde{x} = W^{(2)}h + b^{(2)}$$

Applying softmax function on the output of the last linear transformation to yields the probability distribution over the predicted classes $p$:

$$p = \sigma(\tilde{x}) = softmax(\tilde{x}) = \frac{\exp\left(\max_i \tilde{x_i}\right)}{\sum_i exp(\tilde{x_i})}$$

To calculate loss in a binary classification problem, I use the cross entropy loss function, stated as

$$L(p, y) = Loss(p, y) = -\sum_{i=1}^{M} y_i \log(p_i)$$

## 2.2 Back-propagation

Compute the gradient for the `SoftmaxCrossEntropyLoss` backward pass:

$$\begin{aligned}\frac{\partial p_i}{\partial \tilde{x}_j} = \frac{\partial(\frac{\exp\left(\max_i \tilde{x_i}\right)}{\sum_k exp(\tilde{x_k})})}{\partial \tilde{x_j}} &= \delta_{ij}\frac{\exp\left(\max_i \tilde{x_i}\right)}{\sum_k exp(\tilde{x_k})} - \frac{\exp\left(\max_i \tilde{x_i}\right)}{(\sum_k exp(\tilde{x_k}))^2}\exp(\tilde{x_j}) \\ &= p_i(\delta_{ij} - p_j)\end{aligned}$$

where $\delta_{ij} = \begin{cases} 1 & , i = j \\ 0 & , i \neq j \end{cases}$

$$\begin{aligned}\frac{\partial L}{\partial \tilde{x_j}} = \sum_{i=1}^{M} \frac{\partial L}{\partial p_i}\frac{\partial p_i}{\partial \tilde{x_j}} &= \sum_{i=1}^{M} \frac{\partial(-\sum_{i=1}^{M} y_i \log(p_i))}{\partial p_i}\frac{\partial p_i}{\partial \tilde{x_j}} \\ &= \sum_{i=1}^{M} -y_i \frac{1}{p_i} p_i(\delta_{ij} - p_j) \\ &= p_j - y_j\end{aligned}$$

Compute the gradients for the `LinearMap` backward pass to update weights $W^{(i)}$ and biases $b^{(i)}$ for $i = 1, 2$

$$\frac{\partial \tilde{x_i}}{\partial b_j^{(2)}} = \delta_{ij}$$

$$\frac{\partial L}{\partial b_j^{(2)}} = \frac{\partial L}{\partial \tilde{x_j}}\frac{\partial \tilde{x_j}}{\partial b_j^{(2)}} = p_j - y_j$$

$$\frac{\partial \tilde{x_i}}{\partial h_j} = \frac{\partial(\sum_{k=1}^{H} W_{ik}^{(2)} h_k + b_i)}{\partial h_j} = W_{ij}^{(2)}$$

$$\frac{\partial L}{\partial h_j} = \sum_{i=1}^{M} \frac{\partial L}{\partial \tilde{x_j}}\frac{\partial \tilde{x_j}}{\partial h_j} = \sum_{i=1}^{M}(p_i - y_i)W_{ij}^{(2)}$$

$$\frac{\partial \tilde{x_k}}{\partial W_{ij}^{(2)}} = \frac{\partial(\sum_{k=1}^{H} W_{ik}^{(2)} h_k + b_i)}{\partial W_{ij}^{(2)}} = \sum_{n=1}^{H} \delta_{ik}\delta_{nj}h_n = \delta_{ki}h_j$$

$$\frac{\partial L}{\partial W_{ij}^{(2)}} = \sum_{i=1}^{M} \frac{\partial L}{\partial \tilde{x_j}}\frac{\partial \tilde{x_j}}{\partial W_{ij}^{(2)}} = \sum_{k=1}^{M}(p_k - y_k)\delta_{ki}h_j = (p_i - y_i)h_j$$

$$\frac{\partial h_i}{\partial l_j} = \delta_{ij}a \quad \text{where } a = \begin{cases} 1 & , l_j \geq 0 \\ 0 & , l_j < 0 \end{cases}$$

$$\frac{\partial L}{\partial l_i} = \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial l_i} = \frac{\partial L}{\partial h_i}a = \sum_{i=1}^{M}(p_i - y_i)W_{ij}^{(2)}a$$

# 3 Results and Discussion

In the set-up of the experiment, fixed the followings variables for both the self-implemented neural network and the Pytorch one:

- Number of nodes in hidden layers = 100
- Learning rate = 0.01
- Batch size = 64
- Epochs = 200

## 3.1 Neural network implementation from scratch

In this section, I present the results of my implementation of a single layer neural network from scratch.

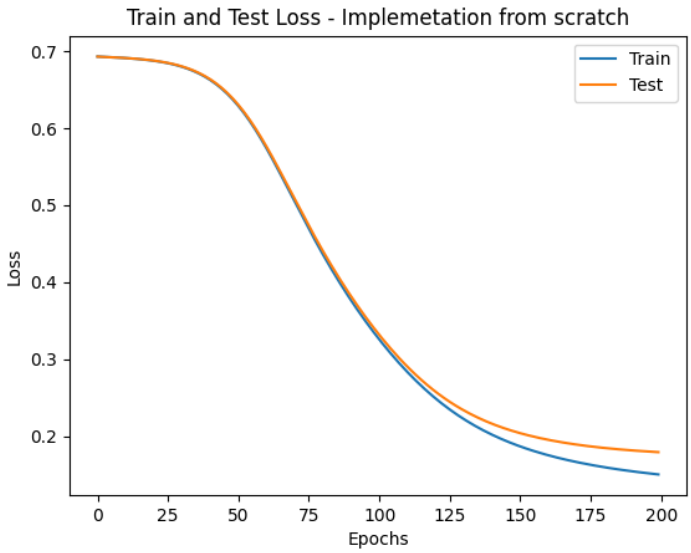


***Figure A:*** *training loss and test loss of my neural network implementation.*

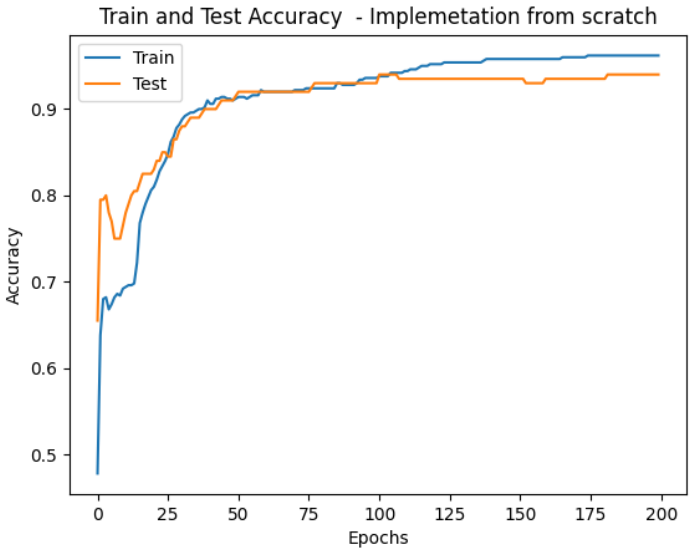


***Figure B:*** *training accuracy and test accuracy of my neural network implementation.*

### 3.2 Neural network implementation with Pytorch

Below are the results of a single layer neural network implemented using PyTorch.

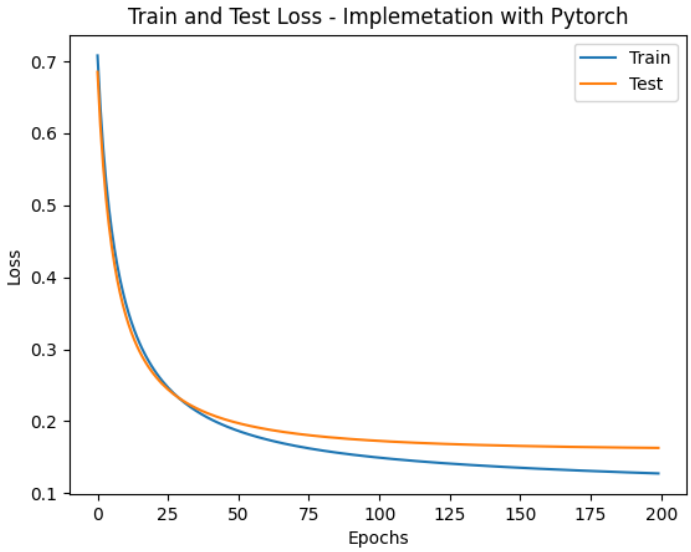


***Figure C:*** *training loss and test loss of the PyTorch implementation.*

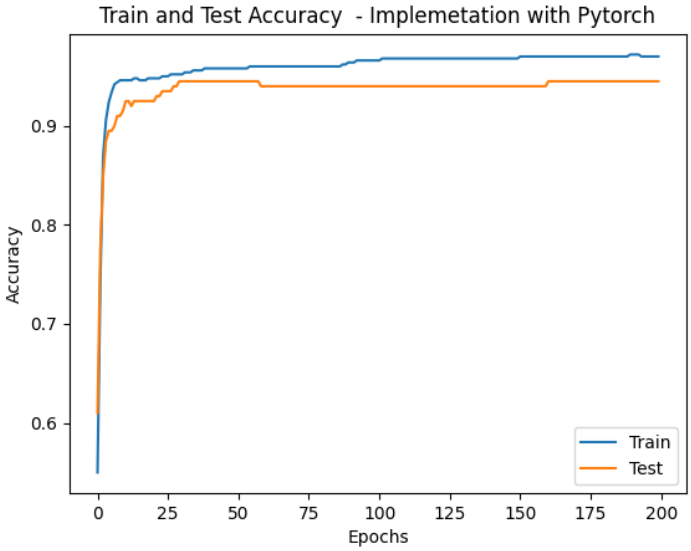


***Figure D:*** *training accuracy and test accuracy of the PyTorch implementation*

### 3.3 Discussion

Using the same parameters as given in `reference.py` (hidden dimensions = 100, learning rate = 0.01, batch size = 64, epochs = 200) for both implementations, I achieve the training and testing losses and accuraccies shown in above figures. My neural network implementation shows convergence at a slower rate compare to the Pytorch one at the first 100 epochs, but both implentations achieve training and testing accuracies above 90% after 50 epochs and converge to above 94% accuracy on test set within 200 epochs. Generally, both the Pytorch implementation and my implementation give similar results in both losses and accuracies on testing and training sets after 175 epochs. Also, from the figures, I observe that the neural network model is slightly over-fit after around 140 epochs for the implementation from scratch and after roughly 60 epochs for the implementation with Pytorch.

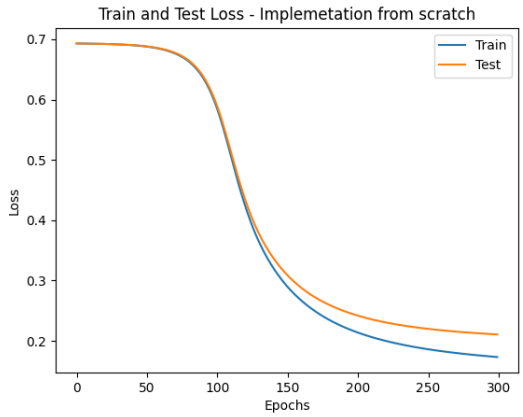


**Figure E:** training loss and test loss of my neural network implementation

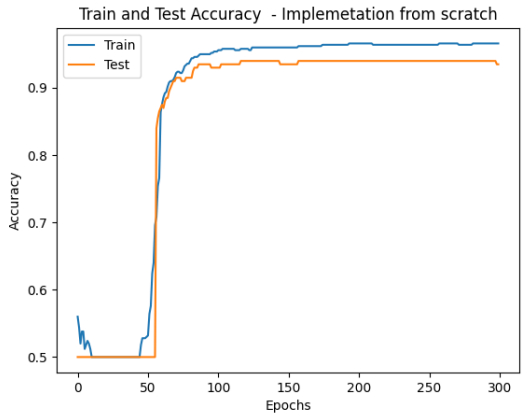


**Figure F:** training loss and test loss of my neural network implementation.

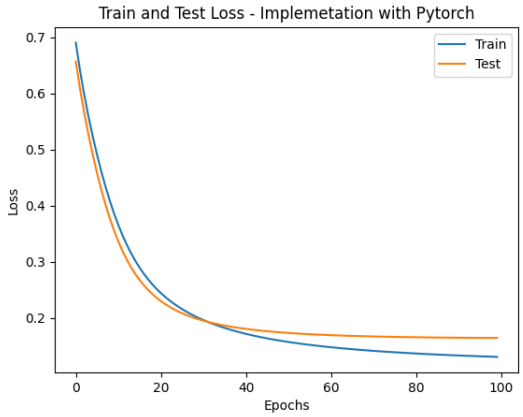


**Figure G:** training loss and test loss of the Pytorch implementation

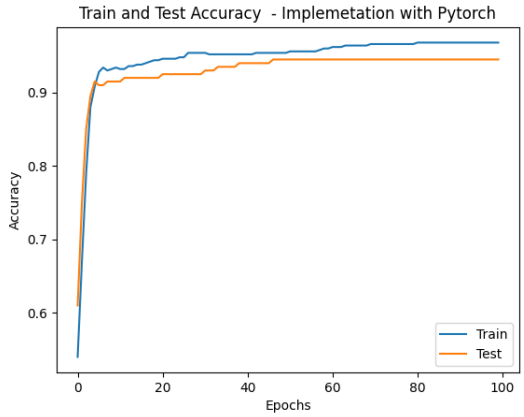


**Figure H:** training loss and test loss of the Pytorch implementation.

# 4 Two-layers Neural Network

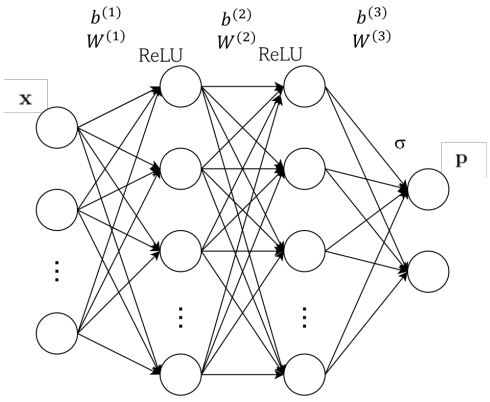


***Fig 2:*** *Two-layers neural network structure*

Observe that for the self-implemented two-layer neural network, the losses and accuracies of testing and training sets are both converged much slower compare to those of the Pytorch imlementation. While the Pytorch neural network model quickly achieves test and train accuracies above 90% within the first 20 epochs and reaches test

accuracy of 94% after 50 epochs, the self-implemented neural network model only converges after first 100 epochs and reaches test accuracy of 94% after 150 epochs.