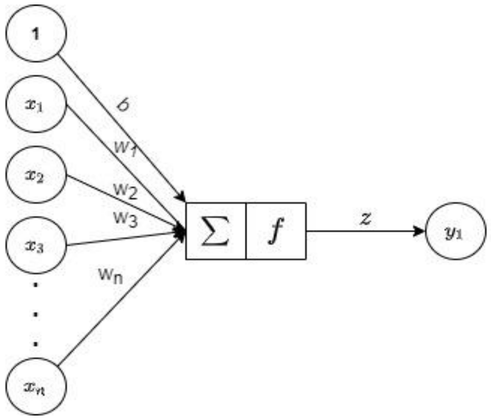
I. Introduction

Generally, in the ML world, a feedforward neural network architecture is referred to as a Multi-layer perceptron (MLP) it is mainly composed of an input layer associated with one or more hidden and output layers.

A closer look at a specific hidden or output layer shows the computation process operated once the input is received from the previous layer. it appears in the figure below that, the node computes the weighted sum of the received input following by the application of the non-linear activation function over the weighted sum and finally passes the result to the next connected node. The iteration of this process over the entire network results in the final output.



the basic transformation that takes place between layers, is simply a change of one vector space to another. The following formula results in a dot product of vectors on n inputs.

$$z = f(b + x \cdot w) = f(b + \sum_{i=1}^{n}(x_i w_i)); \{x \in d_{(1*n)}, w \in d_{(n*1)}, b \in d_{(1*1)}, z \in d_{(1*1)}\}$$

Where:

z represents the output of the node of the neuron;

x represents the input signal;

f represents the non-linear activation function;

b represents the bias of the neuron;

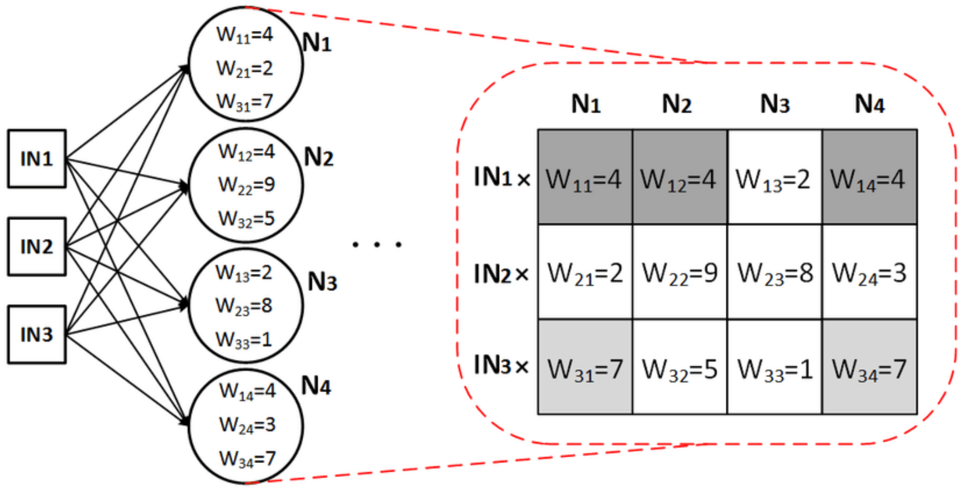
Let describe in more detail the basics building blocks of a Neural Network.

1. Inputs / Outputs

The Input/ Output can be of any form (pictures, text, etc.) although, the language the machine understands is 0 and 1 thus, any input given to the machine will be preprocessed and transform into a format the computer understands. In our case, the input is a 2D image that will transform into a matrix form and pass down to our network and the Output will be the image associated with the class it belongs to.

2. Weights

Weight is a connection between neurons that carries a value. The higher the value, the larger the weight, and the more importance we attach to the neuron on the input side of the weight. Also, in math and programming, we view the weights in a matrix format. Let's illustrate with an image.



IN1
IN2
IN3
N1: $W_{11}=4$, $W_{21}=2$, $W_{31}=7$
N2: $W_{12}=4$, $W_{22}=9$, $W_{32}=5$
N3: $W_{13}=2$, $W_{23}=8$, $W_{33}=1$
N4: $W_{14}=4$, $W_{24}=3$, $W_{34}=7$

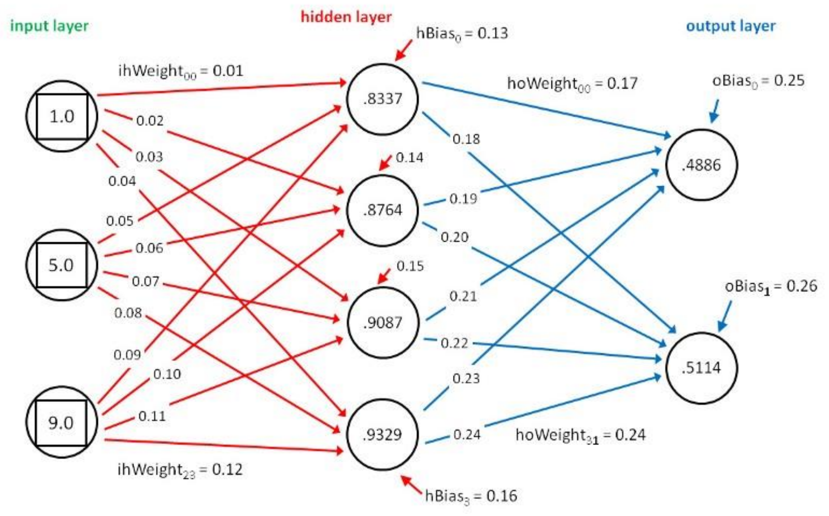
	N1	N2	N3	N4
IN1 ×	$W_{11}=4$	$W_{12}=4$	$W_{13}=2$	$W_{14}=4$
IN2 ×	$W_{21}=2$	$W_{22}=9$	$W_{23}=8$	$W_{24}=3$
IN3 ×	$W_{31}=7$	$W_{32}=5$	$W_{33}=1$	$W_{34}=7$

the input layer has 3 neurons and the very next layer (a hidden layer) has 4. We can create a matrix of 3 rows and 4 columns and insert the values of each weight in the matrix as done above. This matrix would be called W1. In the case where we have more layers, we would have more weight matrices, W2, W3, etc.

In general, if a layer L has N neurons and the next layer L+1 has M neurons, the weight matrix is an N-by-M matrix (N rows and M columns).

3. Bias

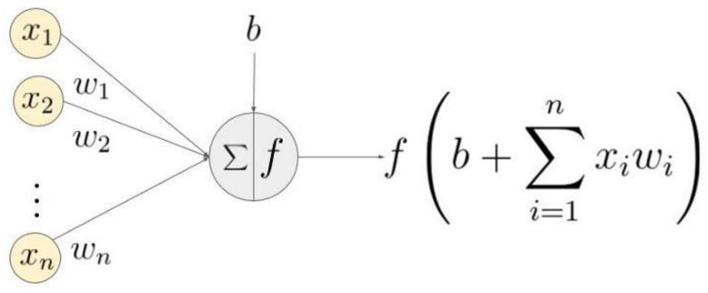
The bias is also a weight. Imagine you're thinking about a situation (trying to make a decision). You have to think about all possible (or observable) factors. But what about parameters you haven't come across? What about factors you haven't considered? In a Neural Net, we try to cater to these unforeseen or non-observable factors. This is the bias. Every neuron that is not on the input layer has a bias attached to it, and the bias, just like the weight, carries a value. The image below is a good illustration.



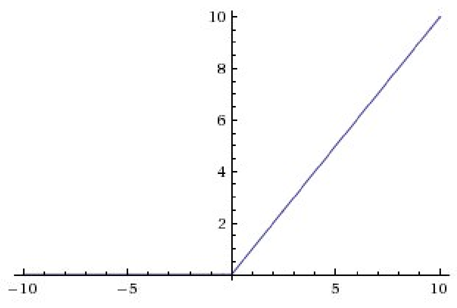
Just like weights can be viewed as a matrix, biases can also be seen as matrices with 1 column (a vector if you please). As an example, the bias for the hidden layer above would be expressed as [[0.13], [0.14], [0.15], [0.16]].

4. Activation function

An activation Function takes the sum of weighted input (w1*x1 + w2*x2 + w3*x3 +…. + wi*xi+1*b) as an argument and returns the output of the neuron.



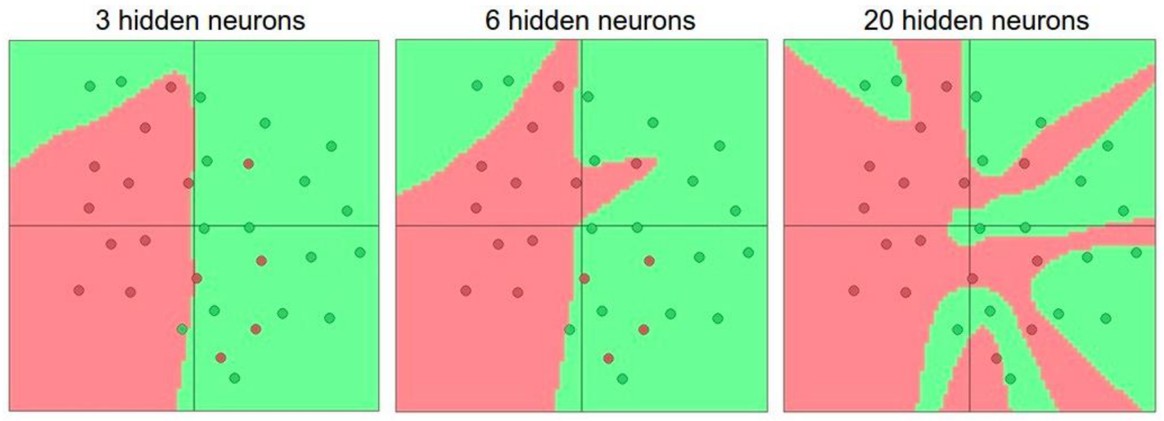
There are several ways our neuron can make a decision, several choices of activation functions. In our case, we used Rectified Linear Units (ReLU). With ReLU, we ensure our output doesn't go below zero (or negative). Therefore, if z is greater than zero, our output remains z, else if z is negative, our output is zero. The formula is f(z) = max (0, z). Graphically it looks like this



Activation functions serve two primary purposes:

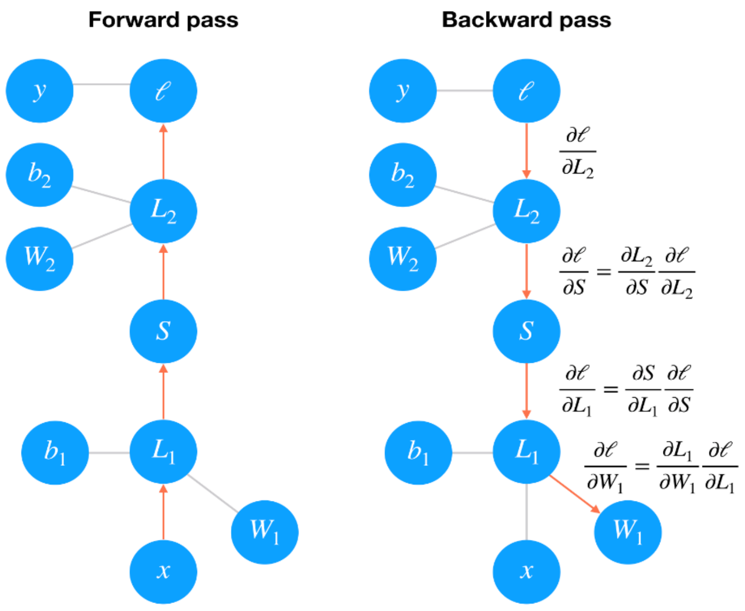
- Help a model account for interaction effects: It is when one variable A affects a prediction differently depending on the value of B.
- Help a model account for non-linear effects: This just means that if I graph a variable on the horizontal axis and my predictions on the vertical axis, it isn't a straight line.

Non-linearities needed to learn complex (non-linear) representations of data, otherwise the NN would be just a linear function



II. Forward Propagation (Pass), Back Propagation, Gradient Descent, and Epochs

- A neural network takes several inputs, processes it through multiple neurons from multiple hidden layers, and returns the result using an output layer. This result estimation process is technically known as "*Forward Propagation*".
- Next, we compare the result with the actual output. The task is to make the output of the network as close to the actual (desired) output. Each of these neurons is contributing some error to the final output. We try to minimize the value/ weight of neurons that are contributing more to the error and this happens while traveling back to the neurons of the neural network and finding where the error lies. This process is known as "Backward Propagation". Back-propagation (BP) algorithms work by determining the loss (or error) at the output and then propagating it back into the network. The weights are updated to minimize the error resulting from each neuron. Subsequently, the first step in minimizing the error is to determine the gradient (Derivatives) of each node w.r.t. the final output.
- By minimizing the loss with respect to the network parameters, we can find configurations where the loss is at a minimum and the network can predict the correct labels with high accuracy. We find this minimum using the Gradient Descent process. The gradient is the slope of the loss function and points in the direction of the fastest change. To get to the minimum in the least amount of time, we then want to follow the gradient (downwards).


Forward pass
Backward pass
y
ℓ
b_2
L_2
W_2
S
b_1
L_1
W_1
x
$\frac{\partial \ell}{\partial L_2}$
$\frac{\partial \ell}{\partial S} = \frac{\partial L_2}{\partial S}\frac{\partial \ell}{\partial L_2}$
$\frac{\partial \ell}{\partial L_1} = \frac{\partial S}{\partial L_1}\frac{\partial \ell}{\partial S}$
$\frac{\partial \ell}{\partial W_1} = \frac{\partial L_1}{\partial W_1}\frac{\partial \ell}{\partial L_1}$

In the forward pass through the network, our data and operations go from bottom to top. We pass the input x through a linear transformation L_1 with weights W_1 and biases b_1. The output then goes through the sigmoid operation S and another linear transformation L_2. Finally, we calculate the loss l. We use the loss as a measure of how bad the network's predictions are. The goal then is to adjust the weights and biases to minimize the loss.

To train the weights with gradient descent, we propagate the gradient of the loss backward through the network. Each operation has some gradient between the inputs and outputs. As we send the gradients backward, we multiply the incoming gradient with the gradient for the operation. Mathematically, this is just calculating the gradient of the loss with respect to the weights using the chain rule.

$$\frac{\partial l}{\partial W_1} = \frac{\partial L_1}{\partial W_1}\frac{\partial S}{\partial L_1}\frac{\partial L_2}{\partial S}\frac{\partial l}{\partial L_2}$$

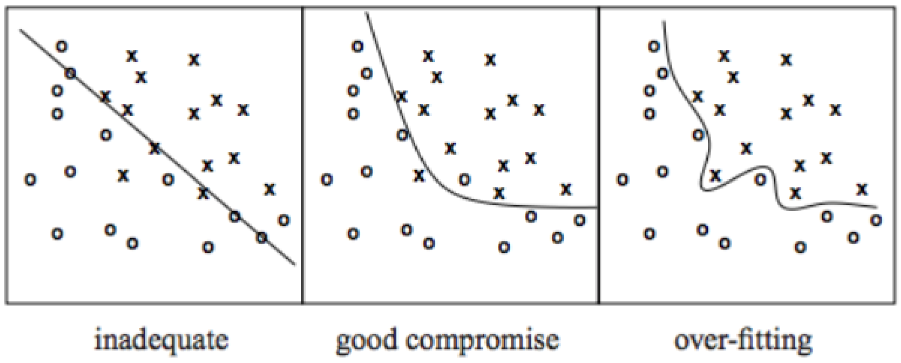
We update our weights using this gradient with some learning rate α.

$$W_1^{'} = W_1 - \alpha\frac{\partial l}{\partial W_1}$$

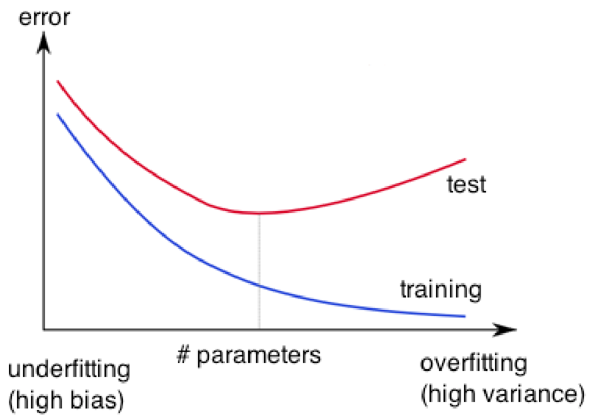
The learning rate α is set such that the weight update steps are small enough that the iterative method settles at a minimum.

- This one round of forwarding and backpropagation iteration is known as one training iteration aka “Epoch”.

When training our network, we must **avoid overfitting**.



Overfitting happens when Learned hypothesis (model) may fit the training data very well, even outliers (noise) but fail to generalize to new examples (test data). Graphically it looks like this

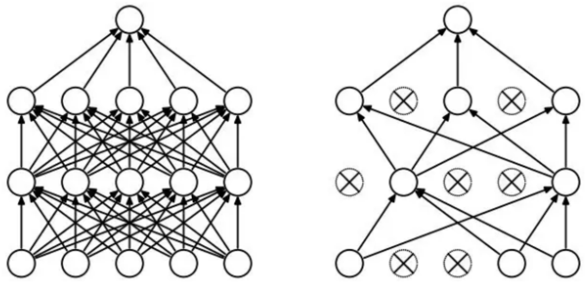


In order to prevent the model to overfit, few techniques can be applied. They are often referred to as regularization methods. Among those we will present two of them.

Regularization methods

- **Dropout**

Randomly drop units (along with their connections) during training. Each unit it retained with a fixed probability p, independent of other units also the hyper-parameter p to be chosen (tuned).

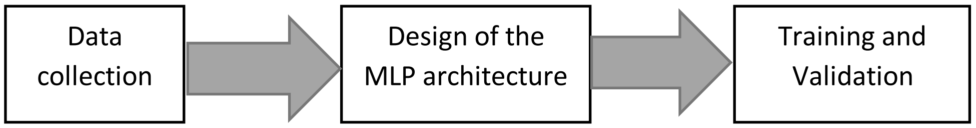


- **Early-stopping**

This approach uses the validation error to decide when to stop training. Stop when monitored quantity has not improved after n subsequent epochs, n is called patience.

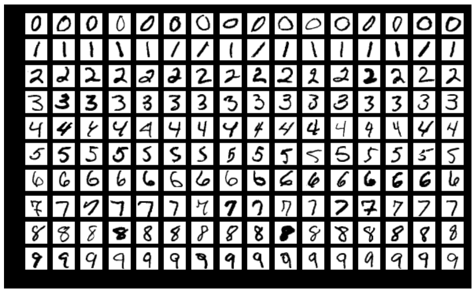
III. Implementation approach

Our approach to design the MLP consists of the steps presented below:

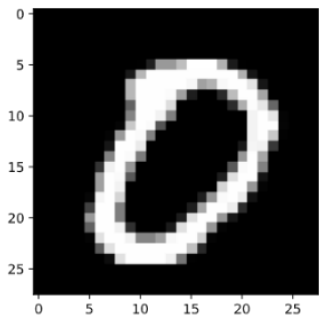


❖ Data collection

The dataset represents not only the first but also a very crucial part in the process of realizing creating our model. The MNIST of handwritten digits contains 60,000 training images and 10,000 test images each of size 28 x 28 pixels, including numbers 0- 9 total 10 categories.

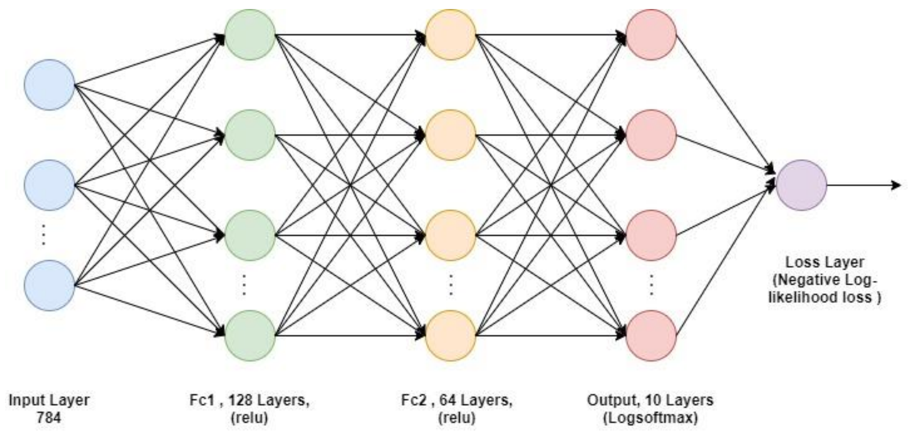


The dataset is available online and can be easily obtained. PyTorch provides an easy implementation to download train and test data. Although, we still need to preprocess the data before feeding it onto our model. We perform two basic transformations: transforms.ToTensor()-- to convert the image into a tensor and transform.Normalize() -- to normalize the tensor with a mean and standard deviation given as the two parameters respectively. This processing will be applied to the downloaded dataset both training and test data. An example of the data is shown below.



❖ Design of the MLP architecture

Our proposed neural network is composed of four layers (input layer, hidden layer 1, hidden layer 2, and output later) with fc1(fully connected 1) denoting the hidden layer 1, fc2(fully connected 2) represents the hidden layer 2, and the output layer. The input layer consists of 784 input units which are just (28 × 28) since our input tensor must be a 1D tensor. Fc1 has a tensor shape of (784, 128). Also, fc2 has a tensor shape of (128, 64) and finally from the fc3 with (64, 10). The figure below shows the architecture of the neural network explained above.


Input Layer 784
Fc1 , 128 Layers, (relu)
Fc2 , 64 Layers, (relu)
Output, 10 Layers (Logsoftmax)
Loss Layer (Negative Log-likelihood loss)

There are two layers with ReLU activation. The ReLU (Rectified Linear Unit) function define as follow, $f(x) = \max(0, x)$.This function simply returns the received input x whenever the input is negative ($x > 0$) and returns 0 when it is negative ($x < 0$).

The output layer is a linear layer with LogSoftmax activation since we are dealing with different classes and we want the prediction to be based on one of the classes it means we are taking a discrete probability distribution over the classes. What this does is squish each input between 0 and 1 and normalizes the values to give you a proper probability distribution where the probabilities sum up to one.

$$LogSoft\max(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

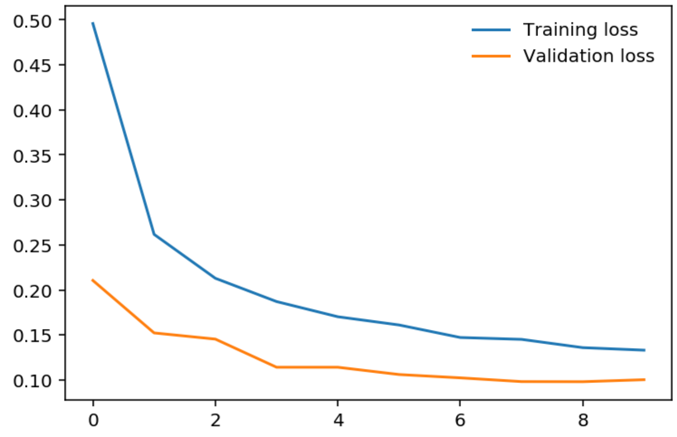
We also define Negative log-likelihood loss along with logSoftmax() performs the cross-entropy loss within our neural network.

❖ Training and Validation

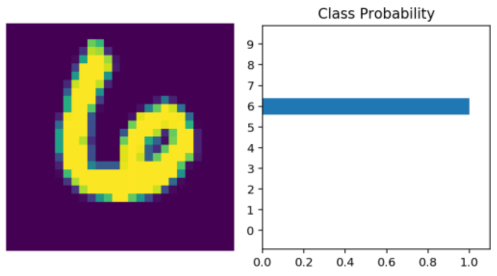
During the training, the model performs a forward pass through the network, it iterates over the training set and updates the weights which eventually, results in some running and training loss. By performing gradient descent and updating the weights by back-propagation, the model gets better at performing the classification. Therefore, in each epoch, we will be observing a gradual decrease in training loss. The table below is a summary of the training parameters.

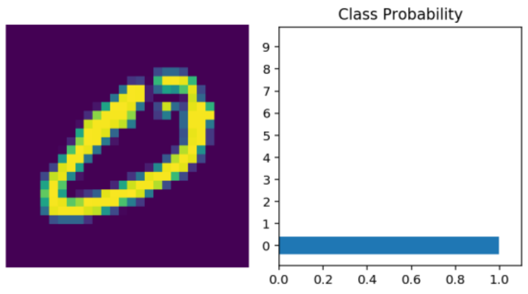
Training Parameters	
Batch_size	64
Learning rate	Lr = 0.001
Epoch	10
Optimizer	Adam
Shuffle	True
Activation function	Relu() and LogSoftmax()
Dropout	True
Framework	PyTorch

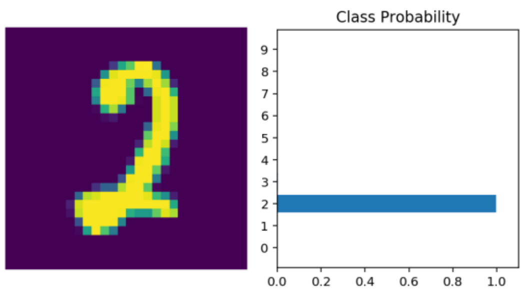
From the graph below we see that the model converges, it does not overfit, because we used the dropout technique, thus as the number of epoch increase, the training loss decreases.

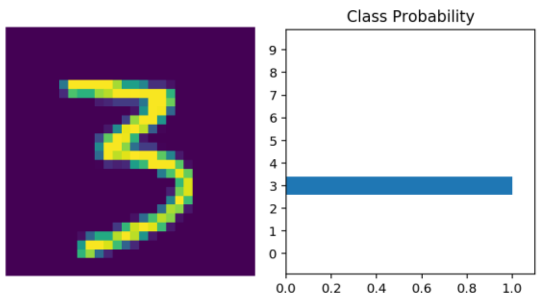


The accuracy is calculated based on the top-class predictions made by our network compared to the actual labels. Our network got an accuracy of **96.97%** (Shown in the code). Here are examples of inference after training.





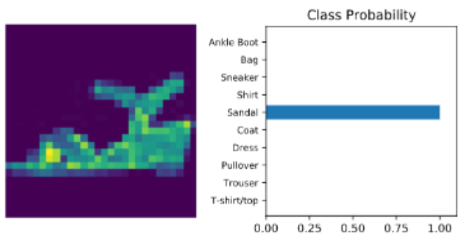


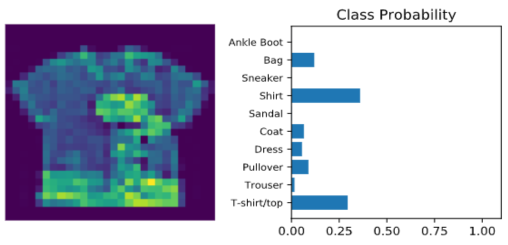


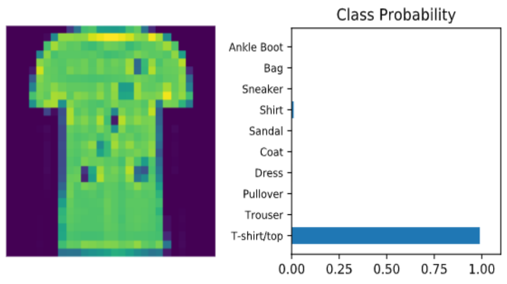
The model could perfectly identify the handwritten digits.

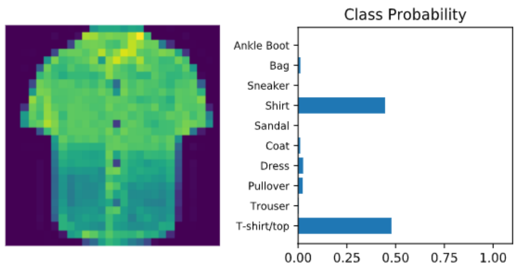
IV. Conclusion

The MNIST dataset is quite simple, therefore the model could easily pick up patterns within the data and correctly classify the digits. Therefore, we implemented quite similar architecture (with more layers) to test our new model on the Fashion MNIST dataset which shows a more complex structure than MNIST handwritten dataset and we had an accuracy of **86,67%.**


Class Probability
Ankle Boot
Bag
Sneaker
Shirt
Sandal
Coat
Dress
Pullover
Trouser
T-shirt/top


Class Probability
Ankle Boot
Bag
Sneaker
Shirt
Sandal
Coat
Dress
Pullover
Trouser
T-shirt/top


Class Probability
Ankle Boot
Bag
Sneaker
Shirt
Sandal
Coat
Dress
Pullover
Trouser
T-shirt/top


Class Probability
Ankle Boot
Bag
Sneaker
Shirt
Sandal
Coat
Dress
Pullover
Trouser
T-shirt/top