CSC3022F

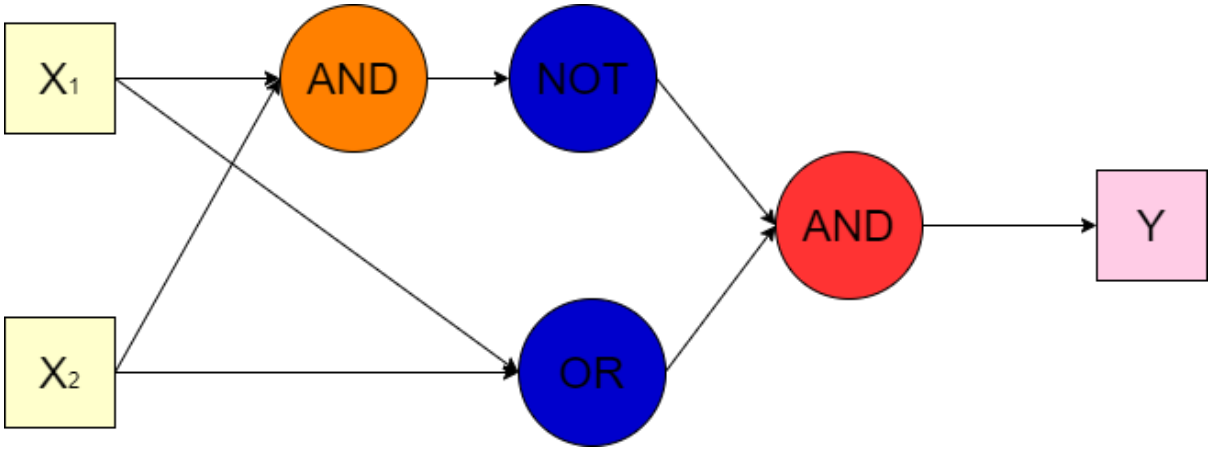
ML Assignment 3

LXXHSI007

Hsien Heng Lie

Part 1

The topology of the network



- I used a feed forward neural network that consists of 3 hidden layers.
- Each perceptron has their own unique bias.
- To create the XOR function:
    - Trained 3 perceptron's, AND, OR and NOT
    - From the XOR algebraic function,
      XOR = (A+B) * -(A*B)
      Also, as XOR = (OR) AND NOT(AND)
      I got the above topology.

Training data

For the training data, I generated a data set, that was bound to -0.25 to 0.25 for 0 and 0.75 to 1.25 for 1, using excel random number generator. Then I went through each data point and did binary calculation for the label set. The data sets are in AND.txt, OR.txt and NOT.txt files. To train the perceptron's using this data, the train option must be entered before inputting values for the XOR function. Random weights are first assigned to the perceptron's and will be adjusted as the perceptron's are trained.

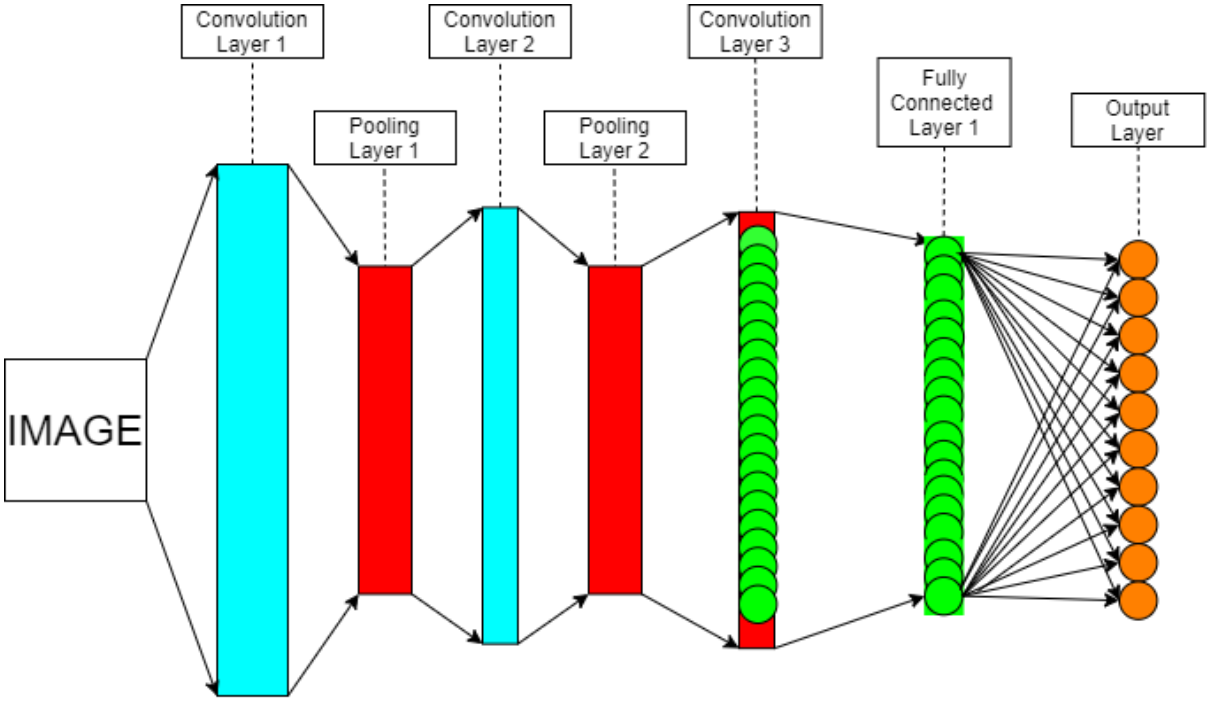
I have also made it possible to set seeded weights and save weights into a txt file. In the file, the first two weights are for the AND perceptron, then the next are for the OR, and finally the last is for NOT perceptron (the 0.0 weight is needed in the file even though the NOT perceptron only accepts one input).

Learning rates

The larger the learning rate the faster the training of each perceptron converged, but most of the time it will create a sub optimal weight set for that perceptron. The sub optimal solutions failed on fringe cases such as max value and min value inputs. I decided to use a 0.2 learning rate as I have 60 examples in each data set to train from. This is still not the best as I would need way more data examples to each the most optimal solution, but it is low enough to stop the weights from fluctuating too much between each example.

Part 2

The topology of the network



By using the LeNet-5 model, 8 layers in the network and includes nonlinear activation functions.

The reason for using the LeNet-5 CNN, is because it was designed for the MNIST dataset and was fairly simpler to create. Convolution operations have the advantage of weight sharing mechanism on the same feature maps reducing the number of parameters as well as other advantages above fully connected layers. The pooling layer gets the features from the convolution layers, and filters features to improve operability of the classification. After the pooling layer the fully connected layer pulls the feature maps into a one-dimensional vector. We give each vector a specific image class.

Layers:

1. Input layer:
    - 28 x 28 Image input
2. Convolution layer 1:
    - 6 convolution nodes of 5 x 5
    - 28 x 28 Feature map to each kernel
3. Pooling layer 1:
    - Outputs 6 feature graphs of size 14 x 14
    - Each cell in each feature map is connected 2x2 cells in the corresponding map in Convolution layer 1
4. Convolution layer 2:
    - 16 5x5 convolution nodes
    - Takes 6 inputs from the 6 features graphs from the Pooling layer 1
5. Pooling layer 2:

- Like Pooling layer 1

6. Convolution layer 3:
    - Fully connected to Pooling layer 2
    - 400 inputs from Pooling layer 2 (16*5*5 = 400 from the Pooling layer 2)
    - 120 nodes

7. Fully Connected layer 1:
    - Fully connected to Convolution layer 3
    - 120 inputs
    - 84 nodes/outputs
8. Output layer:
    - Fully connected to Fully Connected layer 1
    - 84 inputs
    - 10 nodes/outputs

Activation function:

I'm using the ReLu(Rectified Linear Unit) function for the activation function, this gives an output of 0 if x is negative otherwise gives 1. I used this as we are using a large data set and ReLu reduces the likelihood of the gradient to vanish. The bonus of being computational efficient was one of the reasons I used it.

Loss function:

I used the nn.CrossEtrophyloss(), Softmax Cross-Entrophy Loss. Softmax has 2 proterties that help with the prediction, being each value is between 0 and 1, and the sums of all values is always equal to 1. Since the prediction is based on probabilities, it makes the classification much more simpler to implement.

Optimizer:

I used the torch.optim.Adam, Adam optimizer. It is computationally efficient and very simple to implement.

Validation and training

I used the total correct predicted values divided by the total number of values to calculate the accuracy. In each training epoch if the accuracy of that epoch is the best accuracy of all the others it will save the model. This gets the best local minima and can potentially get the global minima if it is trained for more epochs. After all the epochs are completed, the best model will be returned.