XOR PREDICTION-NEURAL NETWORK IMPLEMENTATION USING NUMPY

Neural networks are a powerful tool when there is a need to make predictions based on a non-linear dataset. Predicting the outputs of XOR gate is an apt example of the same. NumPy is a powerful python library which along with its matrices and mathematical functions makes the implementation of this project on python a breeze.

The structure of the program is standard as for most simplified neural networks. It begins with random initialization of weights and biases matrices. The network contains 2 neurons in the input layer with 4 neurons in a single hidden layer and 1 neuron in output layer. The neurons act as the place where all the matrix functions are performed. Firstly, a feed-forward pass where matrix multiplications are performed with the random weights and the outputs are activated with sigmoid function. Further, the raw outputs are passed into a back-propagation pass. In this process, the derivative of loss function is calculated with respect to weights and the loss is minimized. The loss function is a function between the actual output and predicted output of the program. Loss functions are of various types and the one implemented here is the cross-entropy function. It is a useful function employing log functions for classification problems where the output ranges between 0 and 1. Further the weights and biases are updated using gradient descent which uses a learning rate which can be thought of as the size of the step taken for each descent. This process is repeated for a certain number of iterations called epochs so that the network can be trained. The values of weights, biases, learning rate and epochs are called hyperparameters which can be tuned so as to maximize the accuracy of the network.

Graphs and other function visualizations have been done using matplotlib which also indicates the graph between losses and epochs. The losses can be further reduced by using a deeper network with more layers and neurons and utilizing an optimizer like Adam optimizer which aids in a ‘smoother’ descent. The value of learning rate can also be tuned appropriately which yields fastest convergence.

An issue which is commonly observed in neural networks (which was also ran into during this project) is ‘overfitting’. In this scenario, the model ‘rote learns’ certain outputs which is caused when while descending during back-propagation, the model runs into a local minimum instead of the global minimum and assigns it as the minimum loss point. In this case, the model is ought to output wrong values. This problem can be fixed by carefully tuning the learning rate and alternatively an optimizer can also be used.

An interesting point to observe here is how the model is trained using only 4 datapoint (4 possible sets of inputs and outputs in a XOR gate) whereas in common applications, the datasets employed are huge. Interesting examples include MNIST dataset which contains 60,000 small square 28×28-pixel grayscale images of handwritten single digits between 0 and 9 and CIFAR-10 dataset which consists of 60000 32x32 color images in 10 classes, with 6000 images per class.