Project Report for Perception Assignment

Aryan Kayande

Abstract

This report presents the design of my neural network which predicts the depth of an object (cone) based on its bounding box details. The model does not use any popular AI/ML libraries, rather it employs advanced ML algorithms right from scratch. This method aims to reduce latency while maintaining accuracy.

1 Introduction

Yolov5 provides bounding box data in the form of (class, (x, y, w, h), confidence). Here, w and h represent normalised bounding box height and width. It was observed that w, h and depth (y) are highly correlated and thus, the complete bounding box data was fed into the neural network to start predicting depths and provide redundancy over curve fitting methods. This method of estimation is much more robust to box detection errors compared to other methods.

2 Design Motivation

As mentioned above, popular AI/ML libraries were not encouraged for the designing and development of this model, so I have used a structure similar to the internal workings of such libraries. The basic design overview consists of 3 parts : class Value, class MLP (Neuron and Layer also) and actual implementation. The model architecture consists of 1 input layer with 2 neurons, and 1 output layer of 1 neuron (depth). Along with this, the model contains 1 hidden layer of 4 neurons, thus making the model structure : 2 - 4 - 1. Since the number of training examples was not too large (441 samples out of which 264 (60 %) for training), having a simple and small model worked.

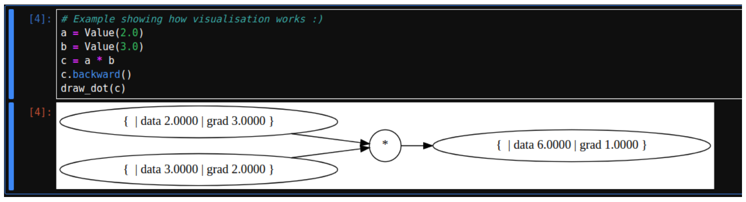
3 Class Value

3.1 Motivation behind Value

When we use numbers in Python, the typical data types that we use are float or int. However, they lack customizability. I build the class Value for storing several attributes of each value, like label, operands and gradients. Several functions are written in this class to simulate their corresponding mathematical operations in Python, so that the objects of this class can be dealt with in a similar fashion to Python floats/int.

3.2 Attributes

Each object (Value) contains ‘data’, the actual floating point value it is supposed to hold. Then, it also contains ‘grad’, which signifies the gradient. During backpropagation, this value eventually determines the amount (direction, to be precise) by which the weights/biases of the network will change. Then, each object also has a ‘children’ attribute for storing the immediate children values that resulted in the current value and also a ‘_op’ for storing the immediate operation that created the value. Lastly, each instance has a ‘label’ for distinctively representing and identifying each Value. Below is a visualisation for a Value :



```
[4]: # Example showing how visualisation works :)
     a = Value(2.0)
     b = Value(3.0)
     c = a * b
     c.backward()
     draw_dot(c)
[4]:
```

{ | data 2.0000 | grad 3.0000 }
{ | data 3.0000 | grad 2.0000 }
*
{ | data 6.0000 | grad 1.0000 }

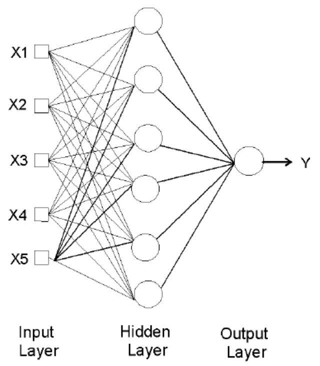
3.3 Functions

The class contains various functions simulating several mathematical operations like addition, multiplication, exponentiation and many more. Each function calculates the respective operative value on the operand's 'data' attribute, and then creates a Value object with the resultant value. Along with this, each function has a '_backward()' function inside it, to calculate the gradient while going backwards through the operation. There are three different activation functions (sigmoid, ReLU and tanh) designed in a similar fashion, to be used later in the model. Lastly, the class contains a 'backward' function, which sorts (topologically) all the children values of the concerned Value node and calculates the _backward (and gradient alongside) for each of them.

4 Class MLP (Neuron, Layer)

4.1 Class Neuron

This class creates nin number of Neurons, each having weights with values between (-1) and (1). When an instance is called, the activation of this neuron is calculated by doing w * x + b and then using activation function tanh on the obtained result. Another important feature of this class is the function 'parameters' which returns all the Values used for the particular Neuron. This turns out to be pretty useful later during backpropagation.

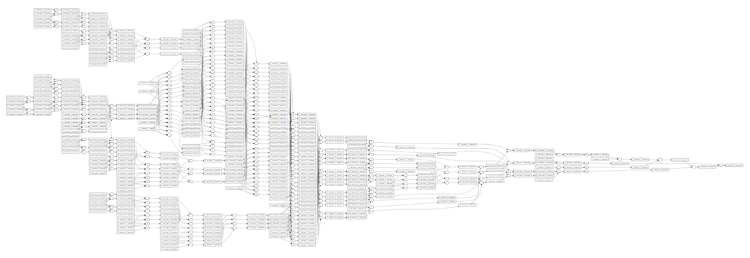


4.2 Class Layer

As the name suggests, this class created a layer with nin inputs and nout outputs. When an instance is called, it further calls the Neuron class and thus the layer is "activated". Similar to Neuron, it also contains a function parameters(), returning a list of parameters of all the neurons in the Layer.

4.3 Class MLP

The class MLP (Multi Layer Perceptron) takes in a list of nins as input and then creates layers of the specified shape. Every time, the class Layer is called and each layer further gets activated. The parameters() function returns the parameters of all the neurons in all the layers of the MLP.
(Below is a visual representation of how the final loss in a MLP might look like, more on that later)



5 Input, Shuffling and Dividing

5.1 Taking Input

The dataset consists of a list of lists named 'bbox', which, essentially, is the output from Yolov5. Each list within consists of 6 elements : (class, (x, y, w, h), confidence). The attributes of our interest are w and h which are used for the training purposes. The dataset also consists of another list : 'depth_values'

5.2 Shuffling and Dividing

The dataset is then split into training and validation sets of size 40 % : 60 % of the total dataset. This accounts to 264 and 177 samples respectively.

Then the dataset is shuffled to create randomness and remove any sort of bias that may affect the training process.

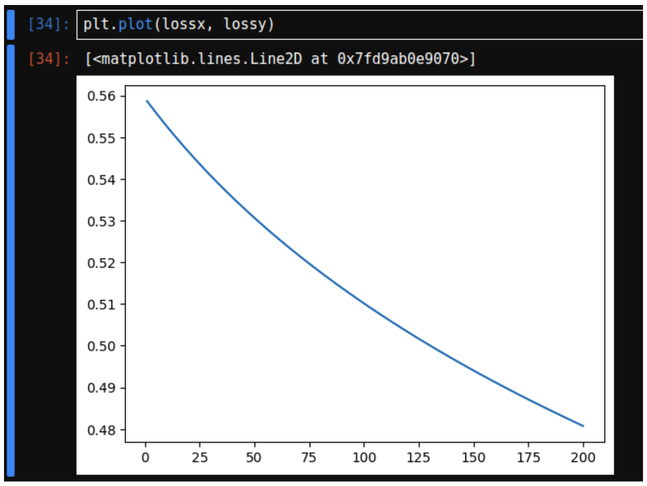
5.3 Normalising depths

The depth values are pretty randomised, ranging from 0.0 to 17.09. So, I thought it was a better idea to normalise these values to squeeze them into range 0 - 1, so that the process of training becomes easier. The method used for normalising is min-max normalisation. This made the process more efficient and stable. After prediction, values are denormalized again.

6 Training

The training process is pretty straight forward, the model predicts what it thinks are the correct values of the depths (at that moment) and then calculates the net loss, which is the summation of all mean squared errors. Then, we initiate the backpropagation by calling 'loss.backward()' and we achieve the values of the gradients of all parameters in the Network. We change the parameter's values and repeat the process, thus lowering the loss every time. This method ensures that the values predicted by the model approach as close as possible to the actual depth values.

Graph of loss v/s epoch :



```
[34]: plt.plot(lossx, lossy)
[34]: [<matplotlib.lines.Line2D at 0x7fd9ab0e9070>]
```

7 Results

The training loss / number of training samples value stooped to as low as 0.15 and the same value of validation loss lowered to 0.21. While checking the values of the model's prediction (denormalized) and actual depth values, the model gave satisfactory results over both training set and validation set.

8 Miscellaneous

8.1 Function draw_dot()

Well, this function is not really 'mine'. This was adapted from Andrej Karpathy's micrograd video (in which he uses this to visualise loss for a MLP he built). The image used above is also from a model I built during one of his lectures :)

8.2 Denormalisation

The values on which the model was predicted were normalised between 0 and 1, so, obviously, the predicted values of the model would also be between 0 and 1. The dénormalisation process is the exact reverse of normalisation :

$$Y_{actual} = Y_{normalised} * (Y_{max} - Y_{min}) + Y_{min}$$

8.3 Choice for model architecture

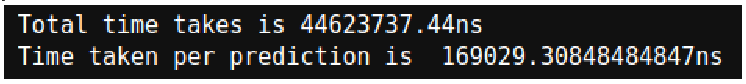
As mentioned in Design Motivation, the number of samples wasn't too large, so I opted for a smaller model. I tried out many models, having a variety of neurons and layers, but eventually got the best results through this model.

8.4 Choice for Activation function

Well, no real 'reason' as such! Tanh() worked pretty fine. The only problem which I faced was the Vanishing Gradient Problem. Essentially, after epoch 2, the gradients were slowly approaching 0 and thus no learning was taking place. I tried using ReLU and leaky ReLU, but it proved no avail. The issue was in fact caused by not normalising the depth values and upon doing the normalisation the issue seemed to have been solved, and I returned back to Tanh().

8.5 Latency

The model takes around 0.16 millisec per prediction, as shown below



```
Total time takes is 44623737.44ns
Time taken per prediction is  169029.30848484847ns
```

9 Conclusion

The development of a network from scratch represents a significant achievement in understanding the fundamental principles and implementation of Machine Learning algorithms. This project provided a comprehensive learning experience and I look forward to taking part in more such projects :)