Implementation of a Deep Learning Regression Algorithm

Abstract: Following the exponential rise of computing power, Machine Learning algorithms have become an increasingly effective tool in various areas of research. This paper outlines the mathematical foundation for a Deep Learning ML algorithm, and subsequently describes the process of building a regressive Feed Forward Deep Learning ML Model from first principles. A toy model for finding the roots (both real and imaginary) of a quadratic equation is then described, and this model's performance is evaluated.

1. Introduction:

REMOVED

2. Theory:

Feed Forward Neural Networks, like the one described here, are in a class of machine learning algorithms classified as supervised algorithms. They are called this, as opposed to unsupervised algorithms, because they require a "training" period beforehand. This training period consists of giving the model multiple input,output value pairs for a given f(x) that you would like the model to fit to. The Neural Network then adjusts its own parameters to generalise a solution to the function that relates the inputs to the outputs presented to it.

2.1. Biases, Weights, Neurons and Layers:

The building blocks of neural networks are called neurons. These neurons are organised into layers as in figure 2.1 .

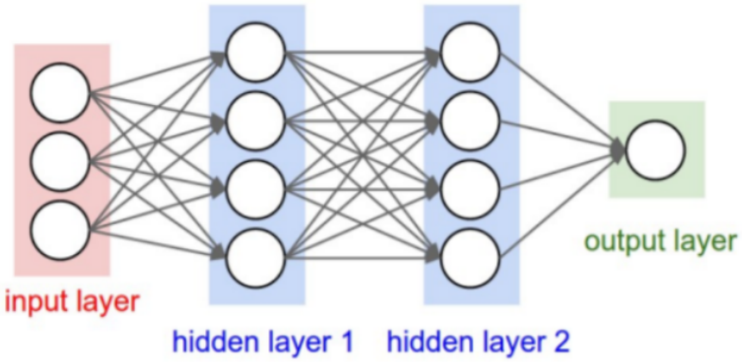

Figure 2.1: Note that every neuron will accept an input from every single neuron in the previous layer. [2]

Each neuron is assigned a bias and a weight on initialisation at random. This bias and weight is what determines the output of the neuron. Take a neuron in the input layer for example. This neuron has a bias b_{ij}, a weight w_{ij} ,one input x_i and an output y_{ij}. Where j represents the layer (1 for the first layer, 2 for the second etc..), and i represents its position in the layer (1 for the top neuron, 2 for the middle and 3 for the bottom).Thus the output of the neuron y_{11}, for example, is calculated as:

$$y_{11} = w_{11}x_1 + b_1 \qquad 2.1.1$$

This is a simple case when there is only one input into the neuron. This output is of course propagated to every neuron in the subsequent layer. In fact, every neuron in each layer will propagate its output to every neuron in the next layer. This means, that for the example in figure 2.1, a neuron in a subsequent layer will have multiple inputs, and we can generalise eq 2.1.1 as [3]:

$$y_{ij} = \sum_j w_{ij}y_{i-1,j} + b_{ij} \qquad 2.1.2$$

Where we have used the outputs from the previous layer ($y_{i-1,j}$) as the input. From this, we can see that each layer does not necessarily have the same number of inputs as outputs. For example, the first layer in Figure 2.1 takes 3 inputs, and has 4 outputs per neuron. This is represented formally as a layer with shape $(3, 4)$, and following the same for the following layers we get layers with shape $(4, 4)$, $(4, 4)$and $(4, 1)$ for layers 2, 3 , and 4 respectively. The total shape of a neural network is called its architecture, and as an example is given by:
$[(3, 4), (4, 4), (4, 4), (4, 1)]$, for Figure 2.1.

So, the initial input is propagated through the neurons through what is called a forward pass of the neural network, using the equations defined above, till it gets to the output layer, These weights and biases that each neuron has is what is tweaked in the training process so that by the end of the training process, a forward pass through this neural network gives the required output.

2.2 Activation Functions:

Now using the model of inputs and outputs described in question 2.1, the neural network can learn to approximate any linear function (given enough layers, which is dependent on the complexity of the problem) . However, if we would like to approximate nonlinear functions, which most real world problems, or problems we would even need to use a neural network on, are, we will need to introduce some nonlinearity to the model. Fortunately, this is very simple. This can be accomplished by simply wrapping the output of a neuron with what is called an activation function, which is a non-linear mapping of the output y_{ij} to another output Y_{ij}. There are many such functions, but one commonly used for regression, and the one used here because of its speed, simplicity, and effectiveness is the Rectified Linear Activation Function:

$$Y_{ij} = max(0, y_{ij}) \quad 2.2$$

Now the network is poised to represent any arbitrary nonlinear function!

2.3 Loss Functions:

Next, we need an estimate of how well the model is performing to be able to push it in the right direction. This is accomplished by using a loss function, which is an estimate of how close the model's output is to the desired output. Tweaking weights and biases to minimise this is the key to how neural networks work. A popular loss function for regression problems is the Mean squared Error. This is the average, of the square, of the differences between the output layer and the true values :

$$L = \frac{1}{n} \sum_{i}^{n} (Y_i - \hat{Y}_i)^2 \quad 2.3$$

Where L is the loss, n is the number of outputs, Y_i are the predicted values and $\hat{Y}_i$ are the true values. This loss is then averaged for all examples given to a function after each round of training (epoch) before being propagated backwards.

2.4 Backpropagation and Gradient Descent

Once the inputs have been propagated forward to the output layer and the loss has been calculated. This information can then be propagated backwards to tweak the weights and biases and minimise the cost function. To do this, we need to know the impact each weight and bias has on the loss. This is nothing more than the partial derivative of the loss function with respect to the weights and biases respectively.

Using the chain rule on equations 2.1.2 , 2.2 and 2.3. We get the partial derivatives to be [4]:

1. For $\sum_j w_{ij} y_{i-1,j} + b_{ij} > 0$:

   a. $\frac{dL}{db_{ij}} = \frac{1}{n} \sum_i^n -2(\hat{Y}_i - max(0, \sum_j w_{ij} y_{i-1,j} + b_{ij})$ 2.4.1

   b. $\frac{dL}{dw_{ij}} = \frac{1}{n} \sum_i^n -2(\hat{Y}_i - max(0, \sum_j w_{ij} y_{i-1,j} + b_{ij}) y_{ij}$ 2.4.2

2. Else:

   a. $\frac{dL}{db_{ij}} = 0$ 2.4.3

   b. $\frac{dL}{dw_{ij}} = 0$ 2.4.4

Now we can adjust the weights and biases with this information. The simplest way to do this is called gradient descent, which is moving the weights and biases in the direction that will minimise the Loss Function, ie:

$$w^{t+1}_{ij} = w^{t}_{ij} - \eta \frac{dL}{dw^{t}_{ij}} \quad 2.4.5$$

$$b^{t+1}_{ij} = b^{t}_{ij} - \eta \frac{dL}{db^{t}_{ij}} \quad 2.4.6$$

Where the t superscript represents the epoch of that weight or bias, and η is an adjustable variable between 0 and 1 called the “learning rate” which governs how aggressively the weight or bias is tweaked in the direction of the gradient each epoch.

2.5 Optimisation and Decaying Moments:

While the traditional method of gradient descent does minimise the loss function, it can take a very long time. For this reason, algorithms called optimisers that make use of additional machine learning concepts are typically used to speed up the process.

The optimiser I have used for the toy model presented here is the Adaptive Momentum, or Adam, optimiser [5]. This optimiser makes use of both the moving average of all previous gradients (momentum), as well as the current gradient in terms m and v, initialized as 0, for a general parameter P_{ij} (either a weight or bias), defined as:

$$m^{t}_{ij} = B_1 m^{t-1}_{ij} + (1 - B_1)\frac{dL}{dP^{t}_{ij}} \quad 2.5.1$$

$$v^{t}_{ij} = B_2 v^{t-1}_{ij} + (1 - B_2)\left(\frac{dL}{dP^{t}_{ij}}\right)^2 \quad 2.5.2$$

Where B_1 and B_2 are decay terms, most commonly set to 0.9 and 0.999 respectively. With these terms defined, the optimisation algorithm reads:

$$P^{t}_{ij} = P^{t-1}_{ij} - A\hat{m}^{t}_{ij}/(\sqrt{\hat{v}}^{t}_{ij} + e') \quad 2.5.3$$

Where A is the analog of the learning rate, called the step size, e' a small number (typically 10^{-8}) to avoid division by zero, and $\hat{m}$, $\hat{v}$ are just the terms m and v corrected as:

$$\hat{m} = \frac{m}{1-B_1(t)} \quad 2.5.4$$

$$\hat{v} = \frac{v}{1-B_2(t)} \quad 2.5.5$$

Where the $B_1(t)$ and $B_2(t)$ terms are just B_1 and B_2 raised to the power of t, not to be confused with superscript t which represents the epoch that the value was calculated at.

That's it! With these four pieces (A model architecture, a loss function, an activation function, and an optimisation algorithm) we can build any neural network.

3. Solving the Quadratic:

Now we have all the pieces to build a neural network and test it on an example. In this section, I will lay out the procedure used to train a model to solve the quadratic equation. There are two variations of this problem that were attempted. The first, simpler, variation was to train the model to only find the maximal real root of a solution. Ie, all quadratic polynomials without a real root were discarded, and for quadratic polynomials that have two real roots, only the biggest one was taken. The second variation was to find all the roots of a quadratic equation, regardless if they were complex or not.

3.1 Procedure:

3.1.1 Maximal Real Root:

The architecture of this model was 3 layers, one input one hidden and one output. The function that this model is trying to fit to is of course $\frac{-b+\sqrt{b^2-4ac}}{2a}$, for $b^2 - 4ac > 1$. Clearly from this we can see that the input layer must take in the triplet (a, b, c) and output one root. So the input layer must have an input size of 3, and the final layer must have an output size of 1. For this variation of the problem, I have chosen 64 as both the input and the output size of the hidden layer.

So the final architecture of this neural network is $[(3, 64), (64, 64), (64, 1)]$ for the input, hidden and output layers respectively. To introduce non linearity into the model, the first two layers were wrapped with a Relu activation function. The final layer was not because this would set all negative roots to 0, which of course is not the output we want for a negative root.

Using numpy, the training data was created by first sampling 3000 random triplets (a, b, c). Then calculating their roots, and keeping the maximum root for triplets that had a real root. This model was subsequently trained for 10,000 epochs.

3.1.2 All Roots:

The procedure carried out here is almost the same with two minor changes:

1. Since we now have two roots, that can also be imaginary. The output layer is stretched to a size of 4. This is so that we can separate the roots into real and imaginary parts. So for example, for a triplet (a, b, c) with solutions $(d + i * e, f + i * g)$, where d,e,f and g are real numbers, and i represents the complex variable, the desired output would be $[d, e, f, g]$ (four outputs).
2. All polynomials with no roots are discarded, and all polynomials with one root $(d + i * e)$ are padded with an extra solution $(0, 0)$ so that the desired output now looks like $[d, e, 0, 0]$.

So the final architecture of this neural network is $[(3, 64), (64, 64), (64, 4)]$ for the input, hidden and output layers respectively, and the model is trained for 10,000 epochs.

3.2 Evaluation and Results:

3.2.1 Maximal Real Root:

For regression problems such as this one, there is no real metric for accuracy. For this reason, I have used a custom one. The accuracy metric used is:

$$\frac{1}{n}\sum_{i=1}^{n}\left|\frac{(Y_i-\hat{Y}_i)}{Y_i}\right| \qquad 3.2.1$$

Equation 3.2.1 represents how far off, on average, the predicted value is from the true value, as a percentage of the true value. When the predicted values are on average more than 100% of the original value off, the accuracy is recorded as 0.

With this metric, this simple 3 layer model was able to achieve ~94 % accuracy on the training set with only a few thousand examples and 10,000 epochs.

To see if a model has truly generalised a solution, we need to evaluate its performance on a test set with (a, b, c) triplets it has not encountered before. A test set ~600 new examples was created to test this model, and the model achieved an accuracy of ~91%.

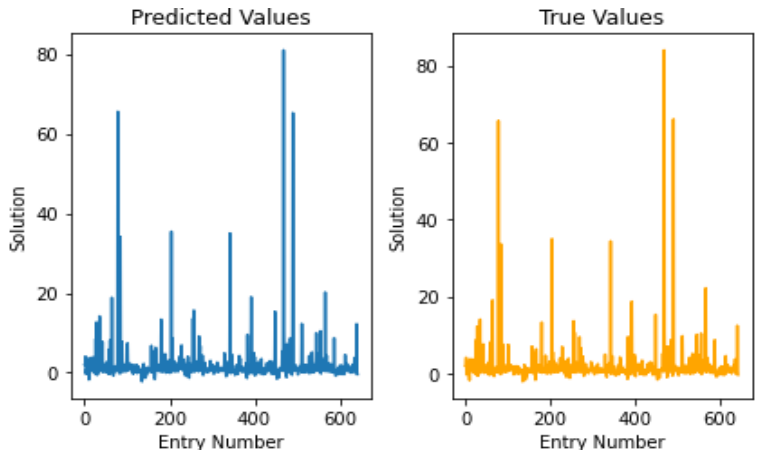

Figure 3.1: Each entry represents a unique (a,b,c) triplet

3.2.2 All Roots:

The accuracy metric applied here follows equation 3.2.1. Using this metric, the model was trained for 10,000 epochs on 3000 examples and was able to achieve ~88% accuracy on the training set and ~83% accuracy on the test set.

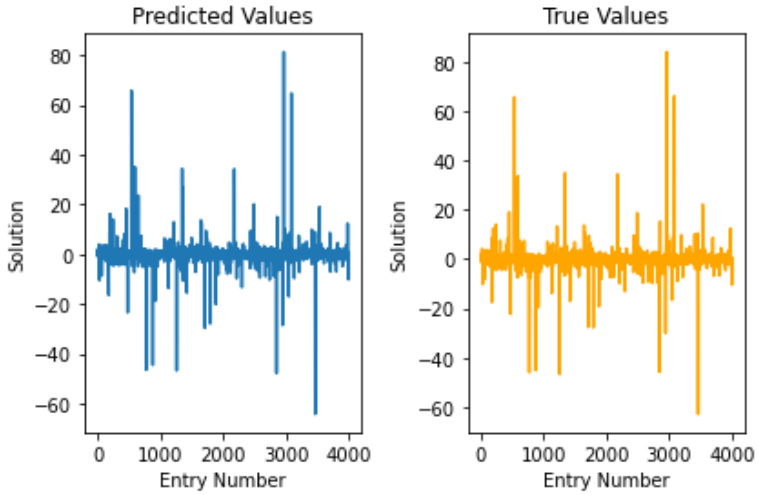

Figure 3.2: There exists 4 entries for each (a,b,c) triplet,4000 outputs for 1000 total unique (a,b,c) triplets [1]

This model was tending in the direction of the solution even when training was stopped. This indicates that its accuracy can be pushed further if given more time to train. To test this, 100,000 training examples were cycled for a total of 1,000,00 epochs. This got the model to reach an accuracy of ~95% accuracy. Training this model for 1,000,000 epochs will take a few hours on most devices, so for this reason, a pre-trained model has been attached with the code as well, and code/instructions to import and use it in python is at the end of the main code write up.

Conclusion:

Using only one hidden layer, and a few thousand examples, a simple neural network like this one was capable of coming within striking distance of generalising a solution for the quadratic. With the employment of more machine learning techniques, parameter tuning, and even additional

layers, the performance of a neural network can increase dramatically. For this, however, pre-built libraries would serve much better as they are highly optimised for speed.

The Feed Forward Neural Network described here is only one of many neural networks that can be built, and a huge class of neural networks exist that are specialised in solving various types of problems. Similar neural networks, with small tweaks in the loss function or the definition of its neuron, can be made to accomplish classification, or deal with time series data, for example. Machine learning is evolving quickly, and providing various powerful tools that can be used for physics. As such, it I believe it is a vital tool that many physicists will benefit from having in their tool box

References

[1] HEPML-Livingreview (2021). [https://iml-wg.github.io/HEPML-LivingReview/]

[2] J. et al McGonagle. Feedforward neural networks(2021). [https://brilliant.org/wiki/feedforward-neural-networks/#multi-layer-perceptron]

[3] Y. et al. Bengio. Introduction to multi-layer feed-forward neural networks. IEEE Transactions on Neural Networks and Learning Systems, 5(2):157–166, 1994.

[4] T. Parr and H. Jeremy, *The Matrix Calculus You Need For Deep Learning* (San Fransisco, 2018).

[5] D. Kingma and J. Lei Ba, ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION (2015).