Project 1.1: Software 1.0 Versus Software 2.0 for FizzBuzz

Arinjoy Bhattacharya
Department of Physics
University at Buffalo, SUNY
Buffalo, NY 14214
Person # 50168806
arinjoyb@buffalo.edu

Abstract

The FizzBuzz problem has been solved using a logical approach and a neural network algorithm in this report. On tweaking various hyperparameters like learning rate, activation function, number of hidden layers, optimization method, various accuracies were obtained and reported. It has been shown after various trails that 'relu' and 'Adagrad' can be used for maximum accuracy.

1 Introduction

FizzBuzz is a very easy and commonly issued problem, but solving it using machine learning approach provides the basic understanding of neural networks. In this report, two different methods of solving the problem has been shown and compared the results. For a huge data set it is recommended to use a machine learning approach since the testing accuracy can be easily tweaked in such an algorithm.

2 Model

2.1 Model: Software 1.0

For Software 1.0 of our project we implement a logic based approach thus resulting in a 100% accuracy every time. But with a huge set of data we might need to use Software 2.0 since the cross entropy loss function can be used to more or less return an accurate value of FizzBuzz.

2.2 Model: Software 2.0

Software 2.0 implements a multilayer neural network to tackle the FizzBuzz problem. A neural network can be thought of as a network of "neurons" which are organized in layers. In our model we have three layers, an input layer, a hidden layer and an output layer. The inputs are multiplied with the weights from each layer to form the hidden layer. They are then subsequently trained with the activation function and multiplied with respective weights to get the output. The output is then regularized with the optimization method. More complex systems will have more layers of neurons, some having increased layers of input neurons and output neurons.

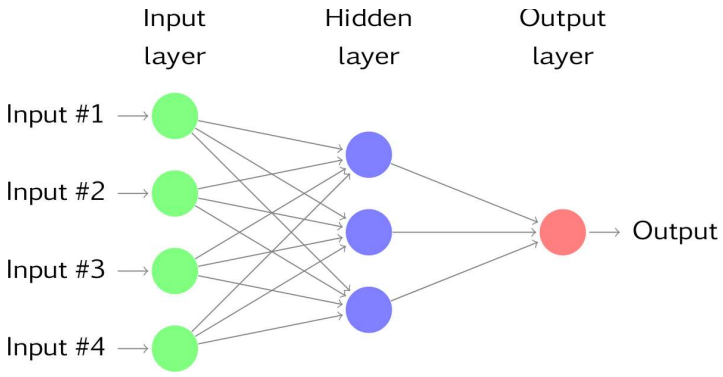

Figure 1: Schematic of the neural network in our project.

2.2.1 Model: Neural Networks

Neural networks are trained iteratively using optimization techniques like gradient descent. After each cycle of training, an error metric is calculated based on the difference between prediction and target. The derivatives of this error metric are calculated and propagated back through the network using a technique called backpropagation. Each neuron's coefficients (weights) are then adjusted relative to how much they contributed to the total error. This process is repeated iteratively until the network error drops below an acceptable threshold.

2.2.2 Model: Neurons

The inputs to each node are combined using a weighted linear combination. The result is then modified by a nonlinear function before being output. Mathematically, it can be shown by the equation.

$$z_j = b_j + \sum_{i=1}^{n} w_{i,j} x_i$$

, where $w_{i,j}$ represents the weight vector, x_i represent the input vector from each neuron and z_j gives the values in the hidden layer.

2.2.3 Model: Layers

There are mostly speaking three layers in the network, the input layer, the hidden layer and the output layer. In the hidden layer, this is then modified using a nonlinear function such as a relu,

$$f(x) = x^+ = \max(0, x)$$

to give the input for the next layer. This tends to reduce the effect of extreme input values, thus making the network somewhat potent to arbitrary values. The parameters $b_1, b_2, b_3, w_{1,2}$ ... are "learned" by training the data. The values of the weights are often confined to prevent them from blowing up. The parameter that restricts the weights in the code is known as the 'learning rate', which varies from 0.1 to 1.

2.2.4 Model: Weights

The weights take arbitrary values to commence with, and these are then updated utilizing the observed data. Thereupon, there's a part of unpredictability within the predictions made by a neural network. Therefore, the network is typically trained many times considering totally different random beginning points, and also the results area unit averaged. The number of hidden layers, and the number of neurons in each hidden layer, must be specified in advance.

2.2.5 Model: Activation functions

Activation functions takes the input from the hidden layer and modify the values to tune the model. Activation functions give neural networks to model complex non-linear relationships. By modifying inputs with non-linear functions neural networks can model highly complex relationships between features. Popular activation functions include relu and leaky relu.

2.2.6 Model: Loss functions

After the hidden layer, a loss function is implemented to check how well the model performs for a particular set of parameters. In the provided code, it is a cross entropy function whose slope gives us how the model actually is performing. So, depending on the performance of the model, the parameters are modified by the loss function.

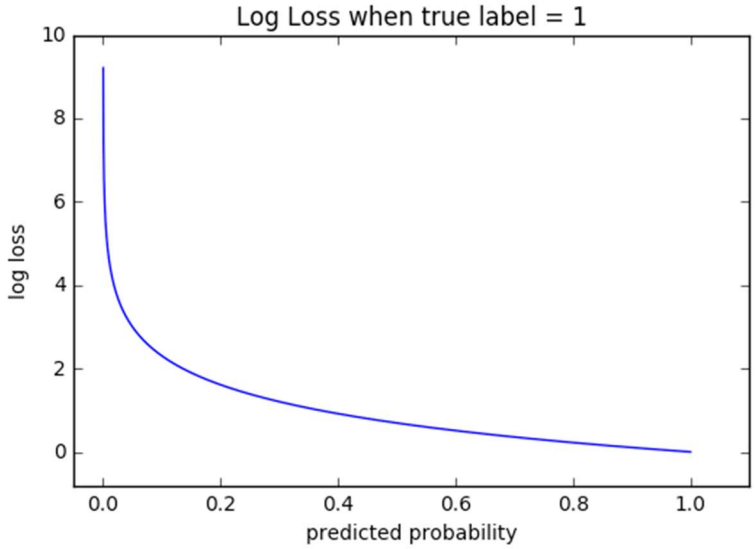

Figure 2: Range of possible loss values for cross entropy for a particular set of parameters.

2.2.7 Model: Optimization Methods

Optimization algorithms are used to back process the model and reduce the error in accuracy rate of the model. The method used in the provided code is gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

$$\hat{C} = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2$$

2.2.8 Model: Forward and Backward Propagation

Essentially speaking, the two methods of propagation are complimentary to one another. The forward propagation involves calculating weights and multiplying them with the input, and finally using the activation function to pass it to the output layer. In case of backpropagation each weight in the network are adjusted in proportion to how much it contributes to overall error. The error function if differentiated with respect to particular hyperparameters and equated to zero to get our best probable accuracy.

3 Experiment

For the experiment, the neural network was trained with various hyper parameters to measure the testing accuracy of a given data set. The tensorflow version of the code was

utilized for the above said purpose. Some of them include number of layers in the network, different number of neurons in each layer of the network (hidden layer here), different optimization methods, different learning rates, different activation functions. They are sequentially explained with plots below.

3.1 Experiment: different number of hidden layers in the network

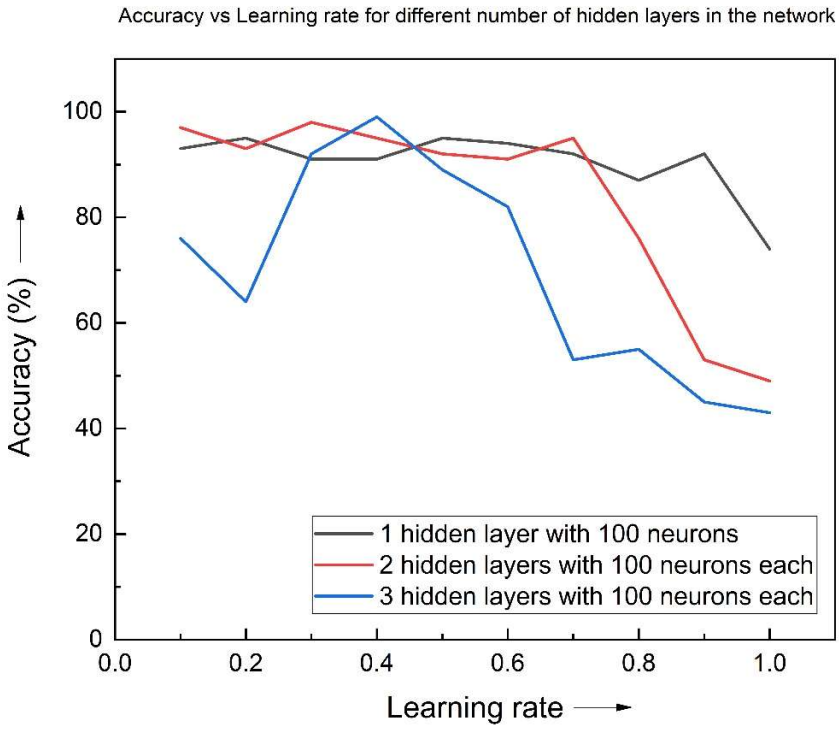

Figure 3: Training Accuracy Vs Learning rate for different number of hidden layers in the network.

From Fig 3 shown above, it clearly shows that increasing the number of layers don't work out too good for the testing accuracy of our model. The learning rate, while training the model was varied from 0.1 to 1 and the testing accuracy based on a given data set was measured. The activation function was chosen to be "relu" and the Optimization Method as "stochastic gradient descent", epochs kept at 5000. The accuracy increases when the number of Hidden layers are increased from 1 to 2 or 0 to 1. But further increase in hidden layers may cause bad things to happen and it cannot be generalized that accuracy will increase proportionately with increase in number of hidden layers. From the plot, in a two-layer model, the accuracy increases at first and then fails at higher learning rate, but in the three-layer model accuracy fails miserably. This might be caused since the Back-Propagation algorithm become less effective which causes the test set error to shoot up when more hidden layers are used.

3.2 Experiment: different number of neurons in the hidden layer of the network

As we see from the figure below, increasing the number of neurons helps accuracy by a lot. All the parameters were kept constant as the previous experiment just the number of neurons were increased. With increasing neurons, the accuracy went up higher and became more constant. It works by eliminating redundant nodes with iterations.

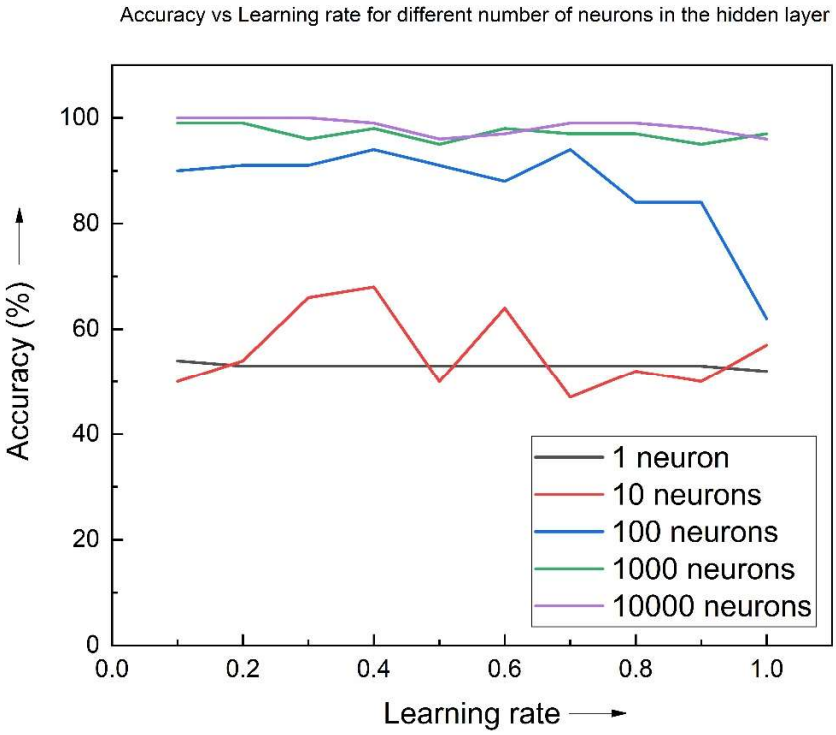

Figure 4: Training Accuracy Vs Learning rate for different number of neurons in the hidden layer of the network

3.3 Experiment: different optimization methods in the network

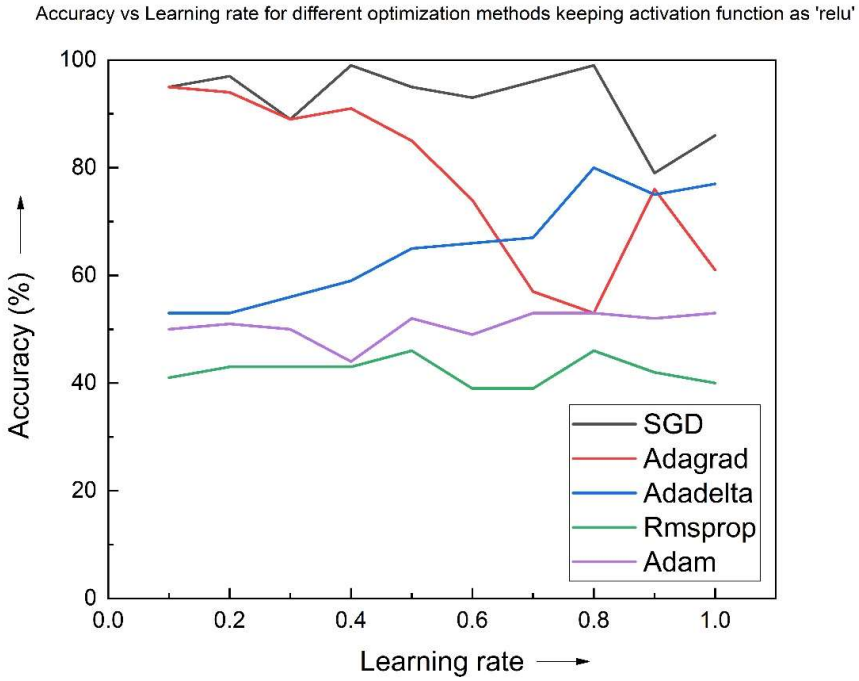

Figure 5: Training Accuracy Vs Learning rate for different optimization method in the network

Optimization methods are important because they help train the model by reducing the least square error. The list of optimization methods includes Stochastic gradient descent, Adagrad, Adadelta, Rmsprop and Adam. Among all of them, as evident from the figure above SGD is the most efficient in training the model since it reduces the loss function after each iteration.

3.4 Experiment: different activation functions in the network

Figure 6: Training Accuracy Vs Learning rate for different activation functions in the network

The hidden layer uses the activation functions to calculate the output from the input layer. Tuning it might result in some interesting observations. But from the figure above it is safe to assume that relu has one of the highest accuracy among the commonly used functions.

Since, there are a varied permutation of hyperparameters to be tweaked so as to experiment with in the model, it is always a good practice to try out all combinations to find out the perfect parameter.

3.5 Experiment: the perfect parameter

The learning rate in the table below was kept at 0.1 and number of epochs at 5000. It is hence suggested to try the following parameters for the maximum accuracy:

- Optimizer method: Adagrad
- Activation function: relu
- Number of hidden layers: 1
- Number of neurons in the hidden layer: 10000

Attached with the submission is the code with aforesaid parameters.

Table 1: Comparison of testing accuracy for various activation functions and Optimization methods.

	SGD	Adagrad	Adadelta	Rmsprop	Adam
relu	92%	96%	53%	49%	50%
sigmoid	53%	53%	53%	81%	80%
tanh	52%	50%	53%	83%	86%
leaky relu	92%	96%	53%	43%	90%

3 Conclusion

The Tensorflow version of the code can be simply used to tune the hyperparameters of the code to get various level of accuracies. It can be also checked for overfitting and training the model better by comparing the training accuracy with the testing accuracy.

Acknowledgments

I thank the professor Dr. Srihari and the TAs for helping me complete this report.

References

[1] CM Bishop, TM Mitchell. (2014) *Pattern Recognition and Machine Learning*, Springer.