# Implementation of FizzBuzz using Tensorflow

**Hrishikesh Nitturkar**
University at Buffalo
hnitturk@buffalo.edu

## Abstract

In this paper, we try to compare two approaches for FizzBuzz implementation logic - Software 1.0 and Software 2.0, where Software 1.0 is a traditional approach to problem solving and Software 2.0 involves training the machine to understand the logic and make a prediction about a value in the future. For this purpose, we use the tensorflow library in python for fast numerical computing. Various functions like relu, matmul etc., help us to train the machine to predict fizzbuzz equivalents of the input numerical values.

## 1 Introduction

In this scenario, we are training the machine to learn the algorithm of FIZZBUZZ implementation. The logic of FIZZBUZZ problem is simple. If a number is divisible by 3, then Fizz is returned. If it is divisible by 5,then BUZZ is printed. If it divisible by both then fizzbuzz is printed and if it is not divisible by both then Other is printed. We are using different packages from python such as tensorflow, keras, pandas and matplot library.

### 1.1 Tensorflow

Tensorflow is a Python foundation library for fast numerical computing created and released by Google. It can be used to create learning models directly or by using wrapper libraries that simplify the process built on top on tensorflow.

Tensorflow works on Python 2.7 and Python 3.3+. It consists of three layers - input layers, hidden layer where computation nodes are present and the output layer. Each layer has one or more nodes that are linked from input to computation layer to the output node with a weight associated to each combination, commonly known as edge. It has three main components.

**Node**: Nodes perform computations on the data that passes through them and have zero or more input and outputs. Data that passes through the nodes are knoown as tensors, which are multidimensional arrays of real values. In our case, the inputs are the numerical values in the training data.
**Edge**: The graph defines the flow of data from the input layer to the nodes and then to the output layer. Each edge is associated with a weight that gives us th probability of the input belonging to a particular layer in the probability.
**Operation**: An operation is a named abstract computation which can take input attributes and produce output attributes. For example, you could define an add or multiply operation.

### 1.2 Keras

Keras is an open source library for deep learning that can run on top of tensorflow. The core data structure of Keras is a model, which means a way of stacking the layers in a model. For this algorithm, we use the to_categorical() function from np.utils package from keras. The to_categorical() converts the encoded labels to their equivalent binary classes.

### 1.3 Pandas

Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. This library provides a very powerful tool, DataFrame, that is used to store and manipulate large data with indexing and also, tools for reading and writing data from csv files, databases etc. Various features like intelligent data alignment, handling of missing data, reshaping of data, slicing, fancy indexing, subsetting, data set size mutability, merging and joining of data sets, hierarchical axis indexing, time series functionality and is optimized for higher performance.

### 1.4 Matplot library

Matplotlib is Python 2D plotting library that is used to plot graphs to compare the accuracy of the algorithm for various instances when different hyper-parameters are modified in the our experiment.

## 2 Description

Any machine learning algorithm consists of 3 main layers. They are input layer, the hidden layer where the neurons are present and output layer.

### 2.1 Input Layer

The input layer consists of various input nodes that are provided to the machine for processing. Each input is assigned a weight and bias and provided to a neuron in the network. This weight is an assumption for accuracy which is changed overtime when the data is passed forward and backward to get accurate weights to make better predictions. The weight can be corresponded to an Edge or a link between an input node and a neuron in the hidden layer. Each input is passed through different or all computation nodes with different weights or Edges that can give out different outputs. The output values are sent back through the network with different weights to manipulate the probability of an input belonging to a particular FizzBuzz layer.

**Epoch**: When all the data is passed through the neural network forward and backward and corresponding weights are calculated for one time, it is known as one **Epoch**.

**Batch**: Since the whole data is too large, it is divided into *batches* or *parts* or *sets*.

**Iterations** are the number of batches in one complete Epoch

**Batch Size**: The number of inputs in a single batch is known as its batch size.

Only one Epoch generally *does not* cover all the probabilities. In other words, only one Epoch gives weights that are less accurate. Hence it is important to increase the number of epochs and also *shuffle* the data in the batches of an epoch to increase the probabilities of the outputs, thus increasing efficiency of the algorithm. It is important to note that very large number of epochs may cause the algorithm to capture the noise in the data. Hence number of epochs should be regulated to give an accurate prediction.

### 2.2 Neural Network or Hidden Layer

The second layer is the neural network layer which is hidden from the user that processes the inputs and gives out probabilities for each input towards a specific output. Each neuron in the hidden layer processes the input based on some activation function.

**Activation Function**: An activation function of a node is used to provide a weight to the output of an input value that denotes the probability of the prediction. There are different types of activation functions that are used in machine learning such as

**Sigmoid Function**: The sigmoid function values exists between 0 and 1. Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.We can find the slope of the curve at any two points. The logistic sigmoid can cause the function to get stuck at a point in training.

It is denoted by

$$\boxed{\phi(z) = \frac{1}{1+e^{-}z}} \tag{1}$$

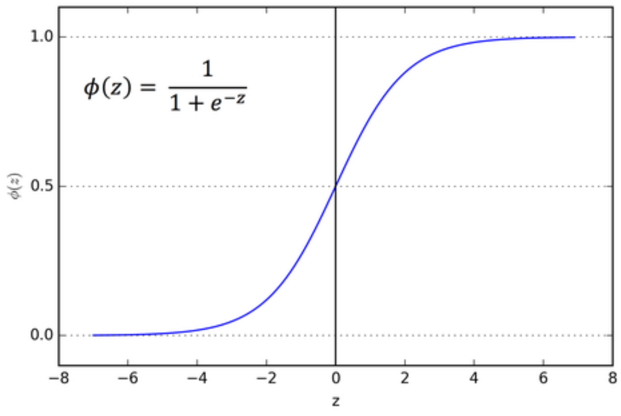

Figure 1: Sigmoid Function Curve

**Softmax activation function**: Softmax activation squashes the input into an output array of values between 0 and 1 and also, the sum of these values is equal to 1. This is done so that the values are easy to handle and it becomes easy to gain confidence over the values by manipulating them and making a prediction based on the highest value in the array.

$$\begin{bmatrix}1.2\\0.9\\0.4\end{bmatrix} \rightarrow \boxed{\text{Softmax}} \rightarrow \begin{bmatrix}0.46\\0.34\\0.20\end{bmatrix}$$

Figure 2: Softmax Function

Softmax activation function is denoted by

$$\boxed{\sigma(a)_j = \frac{e^{a_i}}{\sum_{k=1}^{K} e^{a_j}}} \tag{2}$$

**Rectified Linear Unit(ReLU) activation function**: Rectified Linear Unit(ReLU) activates the neuron based on the probability. ReLU function are the *most widely used* activation functions. ReLU is defined by

$$\boxed{A(x) = max(0, x)} \tag{3}$$

which means, for every negative value, the corresponding neuron is not activated since the output is 0. Since the number of neurons firing is less, the overhead decreases thus increasing efficiency. But when using ReLU, the gradient can go towards 0. For activation values near this region, the weights may not be adjusted during the descent. Hence the neurons that were not utilized before may never be utilized again. This is *dying ReLU* problem.

**Leaky ReLU**: To overcome this problem, Leaky ReLU was proposed. It simply proposed to convert the horizontal component of ReLU to non-horizontal component.

There are many such activation functions for various problems such as Binary Step function, TanH function, Gaussain function etc. We are using ReLU activation function to decrease the overhead of calculations since we have a large dataset to process.

Activation functions or model training algorithms use two types of parameters,
i. **Model parameters** are variable parameters that cannot be assumed. They are calculated as part of the algorithm and are saved as part of the learned model. They are required for the model to make predictions on other data. These parameters determine the skill of the model for the program.

ii. **Model Hyperparameters** are the parameters whose values are assumed before the learning process begins. Since these parameters cannot be derived to increase efficiency, they are to be varied by trail and error method and a relatioship has to be formed to minimize the error function as much as pssible. In other words, hyper parameters are a configuration for the activation functions that have to varied over time for increased accuracy.

Examples of Model Hyperparameters are

1. learning_rate
2. number_of_epochs
3. batch_size
4. number_of_hidden_neuron_layers

### 2.3 Output Layer

The third layer consists of all the possible outputs(in our case, the four fizzbuzz labels). Each neuron gives a weight to output of a input value using an activation function. Based on this output value or weight, a decision is made to connect an input to a particular output. But this is not the final value. This value is changed over time during all the epochs to manipulate it in such a way that the prediction made by the machine is correct.

Each output node has corresponding weight from each neuron. If the output for a specific input is different from expected, the weights need to be adjusted on both sides. The change to be made (increase/decrease) in the weights is given by error function. An error function or loss function or cost function is given by the formula : Real Output - Predicted Output i.e.

$$\boxed{\nabla w = y - \hat{y}} \tag{4}$$

It is important to minimize this function so that the accuracy of the algorithm is increased. The output of error function is used to adjust the weight for the next epoch so that the weights are balanced well and the next epoch gives weights that are more accurate than previous epoch to provide a prediction. A Gradient Descent Optimizer is used to minimize this error function. When output values are plotted as a graph, we take careful and calculated steps to modify the weights to determine the best possible minimum value (minima) of curve to increase the accuracy of the weights. This minimum value is called Global Cost Minimum. It is calculated by the formula

$$\boxed{\theta_j = \theta_j - \alpha \frac{\delta}{\delta\theta_i} J(\theta_0, \theta_1) \qquad for(j = 0 and j = 1)} \tag{5}$$

where α is learning rate, which denotes at what pace should we modify the weights to reach the minimum value. This leads us to linear regression. We have to solve the partial derivative of the cost function to have a working linear function:

$$\boxed{\frac{\delta}{\delta\theta_i}} \tag{6}$$

Caution must be taken that te learning rate is not too small nor is it too big. If it is too small, it might take a lot of time to reach the minimum value of the curve. And if it is too big, we might overshoot and miss the minima and it might not be possible to get to the minimum. Hence it is very important to change the learning rate of an algorithm to reduce the error function value and get best weights or probabilities.

Pictorial representation of the curve can be given by:

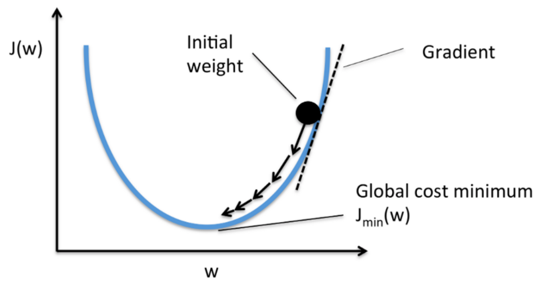

Figure 3: Gradient Descent Optimizer

We are using a Softmax Cross entropy loss function, or log loss that gives a probability value between 0 and 1 for an output of the classification model where 1 denotes perfect classification or match. The log loss function increases when the probability value moves away from the predicted value. As the predicted probability approaches the value 1, log loss decreases. If the predicted probability decreases more, log loss increases again rapidly. Hence log loss penalizes both errors but especially the predictions that are confident and wrong.

## 3 Algorithm or Experiment Execution

In our task of FizzBuzz , there are various hyperparameters such as Number of hidden layers, learning rate, batch size and number of epochs and model parameters such as training accuracy, right and wrong outputs. As discusses earlier, we can change only the hyper parameters since model parameters are learned by the algorithm.
Let us start with running the algorithm using values,

1. LEARNING_RATE = 0.05
2. NUMBER_OF_EPOCHS = 5000
3. BATCH_SIZE = 128
4. NUMBER_OF_HIDDEN_NEURON_LAYERS = 100

Initially the weights are initialized to a normal distribution since the mean of a normal distribution is close to 0. In the hidden layer, ReLU activation is used since we are dealing with huge data and it reduces the overhead by turning off the neurons whose output value was 0. This is done by calling the tf.nn.relu() function from tensorflow package. Then the cross entropy error function is defined using the tensorflow softmax cross entropy function by providing output layer labels are parameters. Then the model is trained using a Gradient Descent Optimizer that uses learning rate as a parameter and the error function is minimized.

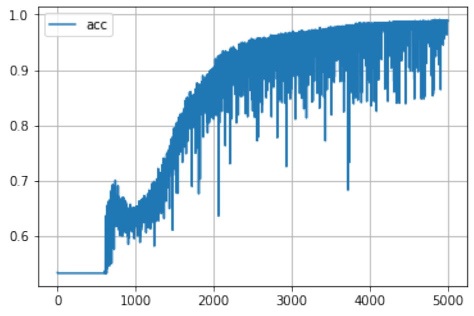

Figure 4: Training Accuracy when number_of_epochs = 5000

As shown in the graph, acc denotes the training accuracy, which is increased and clearly covers most of the points for accuracy. It is to be noted that in the beginning the accuracy did not fluctuate more, but near the 600th epoch mark, the algorithm got significant output values. The accuracy of the algorithm increased more after completing the 5000 epochs.

When the batch size is decreased by half i.e. batch_size = 64, the time taken to complete processing was 41 seconds. But there was a steady increase in the accuracy. This proves that smaller batch sizes are processed more thoroughly. The training accuracy was 92%.

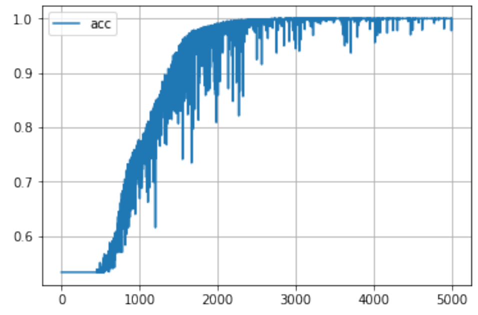

Figure 5: Training Accuracy when batch_size = 64

When the number of epochs are increased to 10000 with batch size 64, the time taken is considerable larger because processing the data for 10000 times requires a lot of time but the accuracy increased significantly in the early stages of training. And the testing accuracy was 93%.

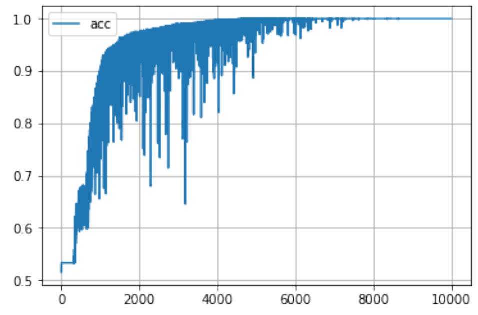

Figure 6: Training Accuracy when number_of_epochs = 10000

When the number of hidden neuron layers increased to 200, the time taken to process the whole data is the same but the testing accuracy is increased to 98%. This proves that when the number of neurons is increased to process the data the accuracy of the model is increased.

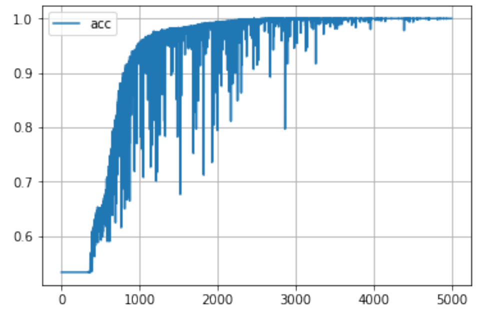

Figure 7: Training Accuracy when number_of_hidden_neuron_layers = 200

When the learning rate is increased, the training model accuracy is increased to the highest value which later diminishes towards the end of the epochs i.e. the end of processing. The accuracy is decreased to 91

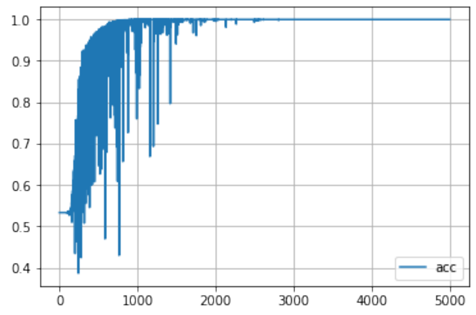

Figure 8: Training Accuracy when learning_rate=0.15

## 4 Observations

Increase in number of epochs result in higher accuracy since processing the data more number of times while shuffling the data gives more preferable weights to the inputs and outputs and as a result, more favorable probabilities are assigned to the outputs that can result in increased accuracy. When the number of epochs are decreased, the data processing degrades and this results in underfitting of the curve. Thus accuracy is decreased. When the epochs are increased to a very large value, the algorithm starts capturing the noise or useless data in the dataset. This inturn decreases the accuracy of the algorithm. Hence it is important to manipulate the number of epochs carefully.

When the size of each batch is decreased, it is observed that the accuracy is increased though the time taken to process all the data is increased. this is because, when the batch size is decreased, less number of inputs are passed a time which result in a thorough processingand increase in more accurate probabilities.

Learning rate helps in determining the minimum value of the error functions through which the weights are manipulated. When the learning rate is decreased, the gradient descent optimizer uses small steps to reach the minimum. This results in determining the value verynear to the minima of the curve. Hence the accuracy of the model is increased. When the learning rate is increased, the gradient descent optimizer shoots with larger steps which mostly results in overshooting and thus the accuracy is decreased.

When the number of neurons are increased, a raise in accuracy is observed. THis is because when the neurons are increased, the data is passed through more neurons and thus the data is associated with more number of weights which is processed more number of times and thus the output values are high for each value indicating increase in accuracy.

To achieve the best data accuracy in best amount of time, it is necessary to vary the hyperparameters and track the accuracy of outputs. Testing out all the combinations of the hyperparameters gives a very low error value, if not 0, which denotes that the resulting accuracy of the algorithm is the best.

## Acknowledgments

This report is supported by the entire teaching faculty of CSE Department for the course CSE 574 - Introduction to Machine Learning for Fall 2018 semester

## References

[1] ShareLaTex concepts for various mathematical representation retrieved from *https://www.sharelatex.com/*

[2] Machine Learning: Cost Function and Gradient Descent Optimization concepts, retrieved from *https://medium.com/@lachlanmiller_52885/machine-learning-week-1-cost-function-gradient-descent-and-univariate-linear-regression-8f5fe69815fd*

[3] ReLU and Softmax Activation Function retrieved from *https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions*