CSE574 Assignment 1.1 Report

Fizz and Buzz

Tenzin Norden
UB person# 50096989
September 17, 2018

1 Model Description

The task of our model is to classify fizz buzz for numbers from 1 to 100. We have logic-based software 1.0 and machine learning based software 2.0 which uses neural network model.

1.1 Software 1.0

Software 1.0 is logic-based approach which prints Fizz if an integer is divisible by 3 and Buzz if an integer is divisible by 5. If an integer is divisible by both 3 and 5 or 15, it prints FizzBuzz and if it is not divisible by 3 or 5 or 15, it simply prints Other.

In the given software 1.0, a fizzbuzz(n) function is defined which performs the task, where 'n' can be any integer. It uses the modulo (%) operator to find the remainder to determine if an integer is divisible by 3, 5 or 15 with the use of if, elif and else statements. Software 1.0 gives an output with an accuracy of 100%.

1.2 Software 2.0

FizzBuzz software 2.0 model is based on machine learning and uses a simple one hidden layer neural network setting. FizzBuzz is a classification problem and this model is based on supervised learning as we know the answer of the output. The training data from 101-1001 with their expected output are provided so that the machine can learn from it and make predictions until the accuracy of its prediction or the performance reaches an acceptable level. Its performance is then tested on the testing data (1-100) by comparing the prediction with known outputs.

The input to the model is numbers and the output will be the 'fizz', 'buzz', 'fizzbuzz' or 'print as-is' based on the input. Following is how software 2.0 model description.

(a) In this model, the input numbers are turned into binary with 10 bits as vector activations. The testing data 1 – 100 and the training data 101-1000 with their labels in FizzBuzz representation are created. Then the data is processed by encoding the input as binaries of 10 bits and their labels in fizzbuzz representation in one-hot encoding of 4 bits.

(b) The model is set up in tensorflow which is two layers deep with one hidden layer and one output layer as shown in figure 1. Tensorflow placeholder for input variable with width 10 and output variable with width 4 is created. Number of neurons in the hidden layer is set to 100 and can be changed.

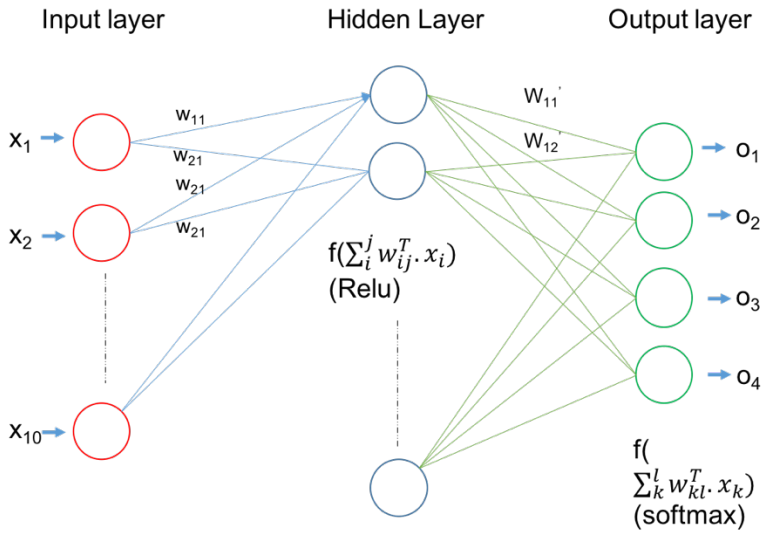

Figure 1. Two layers deep neural network with one hidden layer and 10 input neurons. Relu Activation function used at hidden layer and softmax activation is used at the output layer

(c) The weights for the input, hidden and output neurons are randomly initialized to a normal distribution. *In the hidden layer, an activation function is applied on the sum of the dot products of input vectors (x) and their corresponding weights (w^T), which gives the output of that layer.* An activation function is used to make neural network non-linear and limits the output signal to a finite value, for eg, between 0 and 1.

$$\mathrm{Relu(f)} = (\sum_{i}^{j} w_{ij}^{T}.x_i, 0)$$

In this model, **relu (rectified linear unit)** activation function is used in the hidden layer which is just f(x) = max(0, x). Relu limits the output to be either 0 or x and it can only be used in hidden layers of the neural network.

(d) The output of the hidden layer after applying relu function is set as input for the output layer of our neural network model. Once again, the input to this layer is multiplied with a set of random weights and their sum is fed into **softmax** activation which computes probabilities for different output classes.

(e) The output of the output layer is then compared with the actual known output. The error is computed using cross entropy error function as we have classification problem. Steps so far can be called forward propagation.

(f) **Backpropagation** is then performed using **GradientDescent** optimizer to minimize the error. It propagates backwards in the network to compute gradient of error function with respect to weights and then optimize weights using Gradient Descent optimization to reduce error. Gradient descent is then used to update and optimize weights until a global minimum of the error function is reached. Tensorflow framework does all of these automatically. **Our model trains via backpropagation.**

(f) **Learning rate** is a hyperparameter that determines the steps of a gradient descent. A very small learning will lead to slow convergence while large learning rate might overshoot the minima and can cause loss function to fluctuate and even diverge. In this project, we experiment the performance of the software 2.0 by changing the learning rate from 0.1 to 1.

(g) To train the model, we run the tensorflow model graph for 5000 epochs, where one **epoch** is equivalent to one forward and one backward propagation of all the training data. In this model, we use the **batch size**, the number of training data in one forward and backward propagation, as 128. So, the total number of iterations in the model is approximately 35,156 iterations. The output is then classified into four classes after all the iterations or training.

(h) After batch and epoch training, the testing is done for the testing data from 1 to 100. The output is decoded into fizz buzz label representation.

(i) The output label is then compared with the known label of the testing data and the accuracy is computed.

2 Experiment with software 2.0

I have experimented with the learning program by changing learning rate from 0.1 to 1, activation functions and optimizers to find the combination which gives the best performance. Hyperparameters such as number of epochs (5000), batch size (128) and error function (cross entropy) is kept constant for all experiments. I have also changed number of hidden layers in the network to find the best performance.

Since the weights are randomly initialized, accuracies are different every run. Therefore, all the accuracies plotted are averaged over five individual accuracy.

2.1 Tuning number of hidden Neurons

In the first experiment, the objective is to see how number of neurons in the hidden layer affect the output accuracy of software 2.0. In figure 1, I plotted the accuracy percentage of software 2.0 as a function of Learning Rate. Here the learning rate is changed from 0.1 to 1 with steps of 0.1 for different numbers of hidden neurons in the hidden layer of the neural network. The number of neurons in the hidden layer used are 10, 50, 100, 500 and 1000. Relu and GradientDescentOptimizers are used as activation function and optimizer respectively.

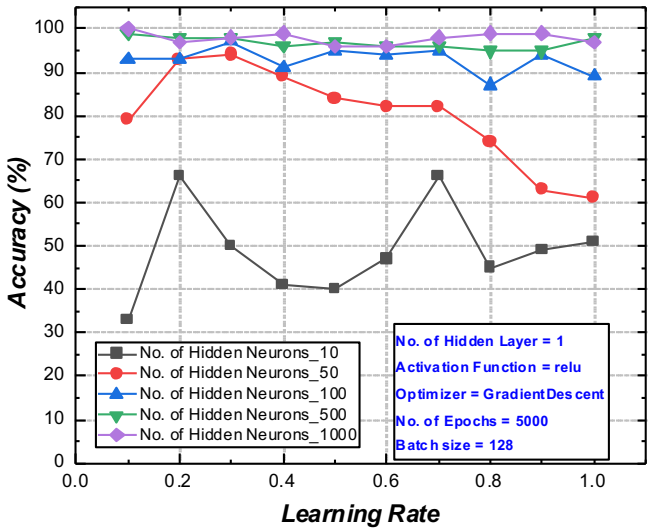

Figure 2. Accuracy vs Learning rate with different numbers of neurons in the hidden layer. Number of neurons in hidden layer: 10 (black), 50 (red), 100 (blue), 500 (green) and 1000 (purple).

From the figure, we can clearly see that the accuracy increases with increasing number of neurons in hidden layer. The hidden layer with 10 neurons showed the lowest and very scattered accuracies as compared with those of 100 and 1000. From this experiment, accuracy for 100 and 1000 neurons in the hidden layer is somewhat similar for all learning rate. I got the highest accuracy of 99% for 0.1 learning rate with 1000 neurons.

I found that a greater number of neurons in a hidden layer gives more combinations of input features which increases the accuracy. However, it is to be noted that using very large number of neurons at hidden layer might cause over fitting and it is not tested in this experiment.

2.2 Different optimizers for fixed activation function.

In experiment 2, accuracy as a function of learning rate for different optimizers is plotted by keeping relu activation function fixed as shown in Figure 2. For relu activation, it is evident that GradientDescent optimizer gives the best accuracy for the range of Learning rate from 0.1 to 1. Adagrad optimizer performs good for lower learning rate from 0.1 to 0.3 but the accuracy drops dramatically after that. Adadelta, Adam and Rmsprop gives very poor accuracy for all learning rates as shown in figure 2.

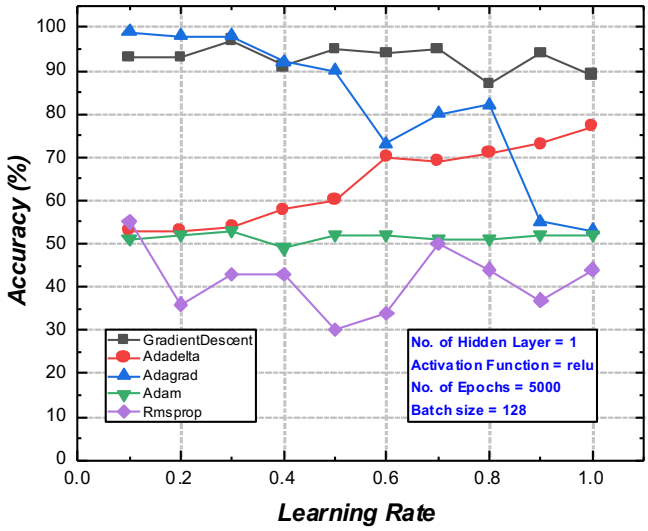

Figure 3. Accuracy vs Learning rate for different optimizers with relu as activation function.

2.3 Different Activation Functions

Here I changed the activation function in the hidden layer and observed how accuracy vs learning rate behaves for different activation function as shown in figure 4. I used relu, leaky relu, tanh, sigmoid, and selu activation functions to check the performance of the software. The accuracy using relu activation is far superior compared with the rest. Although leaky relu have similar accuracy as relu activation for smaller learning rates, its performance deteriorates for larger learning rates.

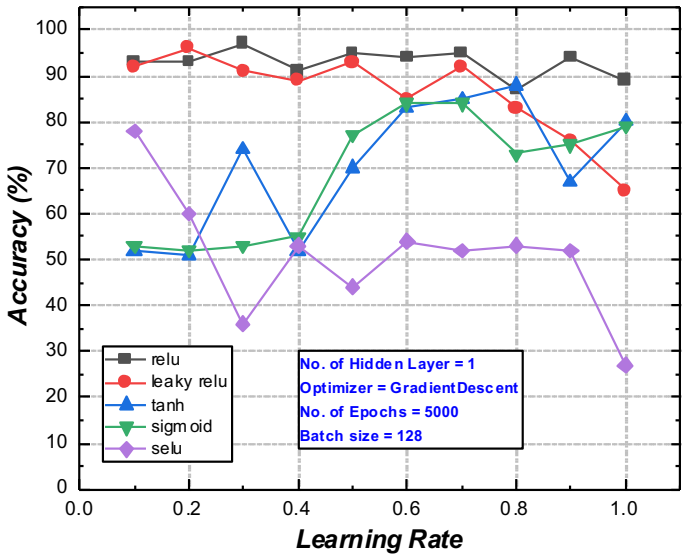

Figure 4. Accuracy vs learning rate with different activation functions and fixed GradientDescent optimizer.

2.4 Different combinations of Activation functions and optimizers

In this section, I have tried all combinations of activation functions and optimizers to find the best possible combination. In table 1, the network setting with the best performance at fixed learning rate of 0.1 is relu activation with Gradient Descent optimizer. Which is why I have used this combination for the next experiment with number of hidden layers.

Table 1. Accuracy for a series of activation function and optimizer at fixed learning rate = 0.1.

Activation Function \ Optimizer	GradientDescent	Adagrad	Adadelta	Adam	Rmsprop
Relu	95	93	53	48	52
Leaky Relu	90	93	53	50	80
Sigmoid	53	53	53	59	80
tanh	47	51	53	84	80
Selu	79	93	53	27	74

1 hidden layer; Number of Neurons = 100, Learning Rate = 0.1; No. of Epochs = 500; Batch size = 128

2.5 Different number of hidden layers

Finally, I want to see the performance with different number of hidden layers in the network. I have observed and plotted accuracy vs learning rate for one, two and three hidden layers as shown in figure 4. Each hidden layer has 100 neurons. The network with 1 hidden layer gives the most consistent accuracy over the range of the learning rate. Two and three hidden layers gives very high accuracy at smaller learning rates; however, the performance deteriorates for larger learning rates.

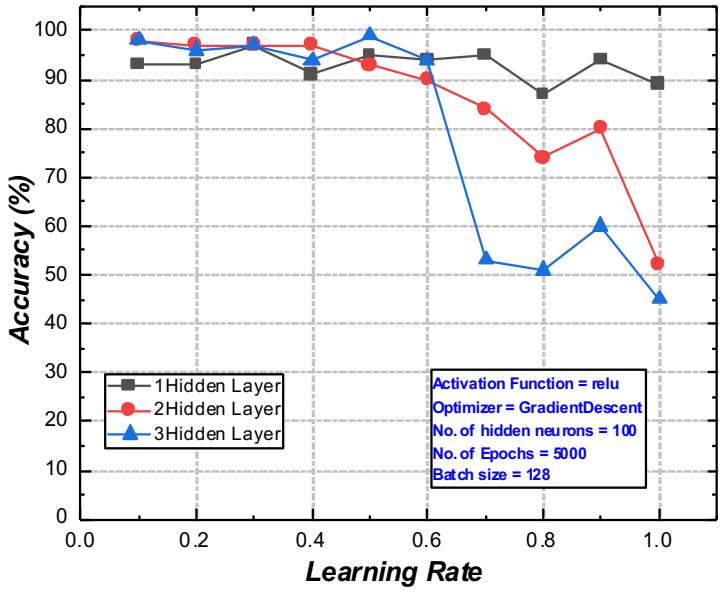

Figure 5. Accuracy vs learning rate with 1, 2 and 3 hidden layers.

3 Conclusion

To conclude, I have experimented with software 2.0 fizzbuzz by tuning various network settings and checking their performance as a function of learning rate.
For network settings with one hidden layer of 100 neurons, the best average performance over the range of learning rate from 0.1 to 1 is the combination of relu activation and Gradient Descent optimizer. For smaller learning rates, two and three hidden layers outperform the network with one hidden layer but it deteriorates for larger learning rates. For one hidden layer, the performance is best for 1000 neurons in the hidden layer.