Machine Learning Project 1.1

Dhanashree Solanke
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
ddsolank@buffalo.edu

Abstract

Fizz Buzz is a group word game. Players take turns to count incrementally, replacing any number divisible by three with the word "fizz", and any number divisible by five with the word "buzz", and any number divided by both three and five “fizzbuzz” and the others we say the number as it is. In this project I have compared two approaches to this problem. First, the traditional coding approach and the second is the machine learning approach.

1 Software 1.0

The traditional approach has a simple logic using if else conditions. For each number we check if the number is divided by 3 and 5, if yes, we return class as “Fizzbuzz”, else we check if the number is divided by 3, if yes, we return class as “Fizz”, else we check if the number is divided by 5, if yes, we return class as “Buzz”, else we return class as “others”. The accuracy for this implementation is 100%.

2 Software 2.0

In this implementation, I have taken up the approach of machine learning to build a model which will return us the category of the number which are either fizz, buzz, fizzbuzz, or others. So, basically, we build up a classification model with Neural Networks with a set of training data from 101 to 1000 and a testing data 1 to 100 to achieve this.

2.1 Neural Networks

Neural nets are a means of doing machine learning, in which a computer learns to perform some task by analyzing training examples. Usually, the examples have been hand-labeled in advance.

Modeled loosely on the human brain, a neural net consists of thousands or even millions of simple processing nodes that are densely interconnected. Neural nets are organized into layers of nodes, and they’re “feed-forward,” meaning that data moves through them in only one direction. An individual node might be connected to several nodes in the layer beneath it, from which it receives data, and several nodes in the layer above it, to which it sends data.

2.2 Model

We start by creating a simple sequential neural network with three layers. The first would be the input layer, the hidden layer and the output layer.

Our input which is integer data is converted into binary inputs of size 10. As we have 101 to 1000 as our training set, 10 bits are enough to represent 1000 numbers. Thus, by encoding we

get an input vector of size 10. The middle layer consists of 100 neurons where every neuron is connected to the input and output layer. The output will have a vector of size 4 as output, where the class 1000 – other, 0100 – Buzz, 0010 – Fizz, 0001 – Fizzbuzz.

2.3 Activation Functions

Activation functions also known as transfer functions are applied at the end of the any neural network. It is used to determine the output of neural network like yes or no. It calculates a "weighted sum" of its input, adds a bias and then decides whether it should be "fired" or not It maps the resulting value from 0 to 1 or -1 to 1 depending on the function used. Here we have used ReLU and Softmax as our activation functions.

2.3.1 Rectified Linear Units

A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input. ReLUs' machinery is more like a real neuron in your body.

$F(x) = \max(0,x)$

ReLU activations are the simplest non-linear activation function we can use.

2.3.2 Softmax

The softmax function squashes the outputs of each unit to be between 0 and 1. It divides each output such that the total sum of the outputs is equal to 1. The output of the softmax function is equivalent to a categorical probability distribution, it tells you the probability that any of the classes are true.

2.3.3 Sigmoid Function

The sigmoid function exists between (0 to 1). I t is especially used for models where we must predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the one.

2.3.3 Tanh Function

Like the logistic sigmoid, the tanh function is also sigmoidal, but instead outputs values range from -1 to 1. Thus, strongly negative inputs to the tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs. This causes the neural network not to be stuck while training.

2.4 Epoch

One epoch is where an entire dataset is passes through the neural network once. Since, one epoch is too big to feed to the computer at once we divide it in several smaller batches. Here we have our batch size as 128. So, the propagation will happen 1000/128 times.

2.5 Optimization Functions

2.5.1 Stochastic gradient descent Optimizer

SGD is an optimization technique used to update the parameters of a model. It works in an iterative fashion. The first run adjusts the parameters a bit, and consecutive runs keep adjusting the parameters improving them. The learning rate used should not be too high or too low here.

2.5.2 Adagrad Optimizer

Adagrad is a gradient-based optimization that just adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. It is well-suited for dealing with sparse data.

2.5.3 Adadelta Optimizer

Adadelta is an extension to AdaGrad. It is a method that uses the magnitude of recent gradients and steps to obtain an adaptive step rate. An exponential moving average over the gradients and steps is kept; a scale of the learning rate is then obtained by their ration.

2.5.4 RmsProp Optimizer

RmsProp is also an extension of Gradient Descent. It restricts the oscillations in the vertical direction. Therefore, we can increase our learning rate and our algorithm could take larger steps in the horizontal direction converging faster

2.5.5 Adam Optimizer

ADAM is SGD algorithm in which the gradient used in each iteration is updated from the previous using a technique based in momenta. The update is a linear combination of the current stochastic gradient and the previous update.

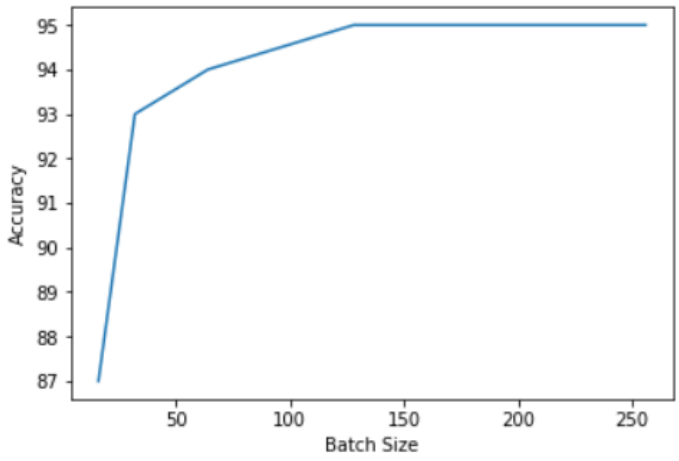
2.6 Cases

In below cases we compare the accuracy by changing the parameters, batch size, epochs, learning rate, etc.

2.6.1 Changing the Batch Sizes

The following parameters are kept constant with different batch sizes.

Input size =10 Epochs = 5000 Learning Rate = 0.05 Hidden layer neurons = 100 Batch Size – [256,128,64,32,16]

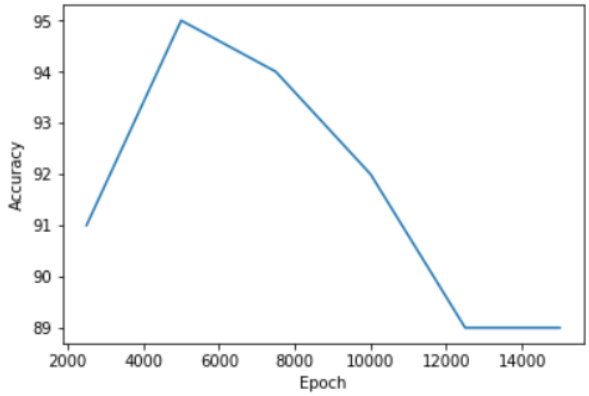


As we can see as the batch size decreases, the accuracy decreases as well. For our model 128 seems to be a best fit for batch size.

2.6.2 Changing the Epochs

The following parameters are kept constant while varying the epoch sizes.

Input size =10 Learning Rate = 0.05 Hidden layer neurons = 100
Batch Size = 16
Epoch = [2500,5000,7500,10000,12500,15000]

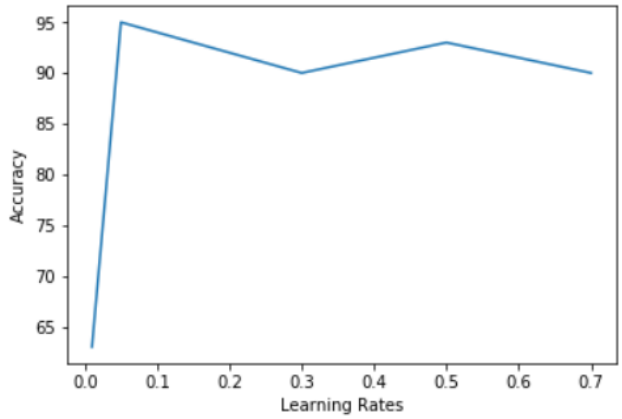


As the number of epochs increase, the computation time also increase, and the accuracy decreases. Hence, 5000 seems to be a best fit for our model.

2.6.3 Changing the Learning Rate

The following parameters are kept constant while varying the epoch sizes.

Input size =10 Epoch = 5000 Hidden layer neurons = 100 Batch Size = 16
Learning Rate = [0.01,0.05,0.3,0.5,0.7,0.9]

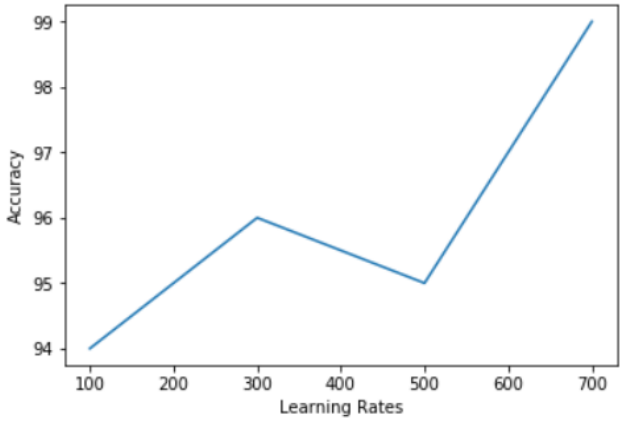


The learning rate was highest at 0.05 and 0.5. But it's better to keep it as 0.05 as the higher our learning rate is the higher chances there are to lose the optimum minima in gradient descent.

2.6.4 Changing the Number of Neurons

The following parameters are kept constant while varying the epoch sizes.

Input size =10 Epoch = 5000 Learning Rate = 0.05 Batch Size = 16
Hidden layer of neurons = [100,300,500,700]

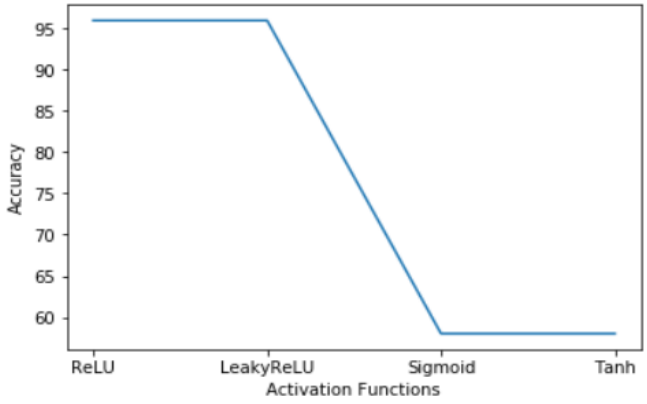


The highest accuracy is achieved when we increase the number of neurons, but the complexity of the model also increases. So, we keep the number of neurons as 30 to achieve the accuracy of 96%.

2.6.5 Changing the Activation function

The following parameters are kept constant while varying the epoch sizes.

Input size =10 Epoch = 5000 Learning Rate = 0.05 Batch Size = 16 Hidden layers of neurons = 100

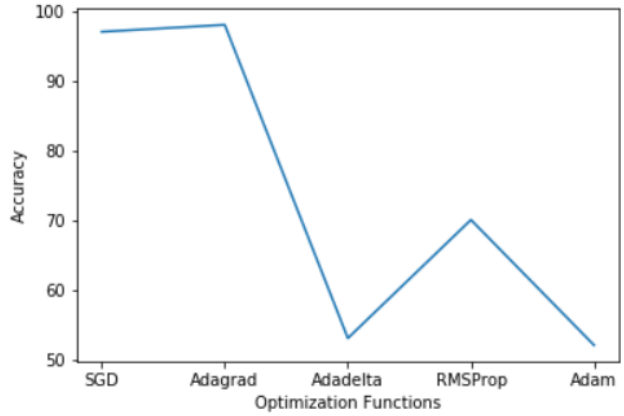


The accuracy of ReLU is more than the Sigmoid and Tanh. ReLU here is more computationally efficient to compute than Sigmoid like functions since ReLU just needs to pick max(0,x) and not perform expensive exponential operations as in Sigmoid. Networks with ReLU tend to show better convergence performance than sigmoid. As we have a multi-class classification problem, using ReLU is a best fit.

2.6.6 Changing the Optimization Function

The following parameters are kept constant while varying optimization functions.

Input size =10 Epoch = 5000 Learning Rate = 0.05 Batch Size = 16 Hidden layers of neurons = 100 Batch Size = 128 Activation Function = ReLU



Adagrad and SGD seems to have highest accuracies in this model. Since, our model is a simple network and trained on small dataset gradient descent would be the best one to choose.

3 Conclusion

The first approach as we saw has an accuracy of 100%.
In second approach a highest accuracy of 96% can be achieved by implementing a Sequential single layer neural network model with epoch of 1000, learning rate of 0.05, hidden number of neurons as 300, batch size as 128, activation function ReLU and Gradient Descent as Optimizer.

3 References

1. https://keras.io/activations/
2. https://www.quora.com/In-Keras-what-is-a-dense-and-a-dropout-layer
3. https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions
4. https://jovianlin.io/cat-crossentropy-vs-sparse-cat-crossentropy/
5. https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b
6. https://climin.readthedocs.io/en/latest/adadelta.html
7. http://ruder.io/optimizing-gradient-descent/index.html#adadelta
8. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6