ML 574 Project 1.1

Daniel Amirtharaj

damirtha@buffalo.edu
UB Person Number 50291137

1 Projrect overview

This project attempts to emulate a *fizzbuzz* program using the software 1.0 and the software 2.0 approaches. The 1.0 model uses simple logical instructions and branches to classify the input data, while the 2.0 model using a dataset learns to classify inputs to their corresponding outputs using machine learning algorithms. The main focus of this project is on the software 2.0 model.

Objective

The objective here is to implement both software models 1.0 and 2.0 for the *fizzbuzz* program using Python. The following are the tasks involved.

1. Implement software 1.0 using traditional logic and branch statements.
2. Generate training and test datasets for software 2.0 using the traditional software 1.0 model.
3. Train software 2.0 using a machine learning algorithm such as Neural networks, using a framework such as Tensorflow.
4. Test the model with the test data and observe performance metrics such as accuracy or mean and variances of accuracy across different parameter initializations.
5. Change parameters such as learning rate, number of epochs or the number of hidden layers and neurons in the network and observe changes in performance, and finally pick model setting with best performance.
6. Using the learned parameters, generate an output file giving the correct classification for any range of inputs.

2 Analysis

2.1 The *fizzbuzz* problem

Fizzbuzz is a simple game created to teach children simple divisions and math operations. A person counts from 1 and the next person calls 2 and so on, until a number divisible by 3 or 5 arrives. When such a number arrives the person calls *fizz* if the number is divisible by 3 and calls *buzz* if the number is divisible by 5 and calls *fizzbuzz* if the number is divisible by both 3 and 5.

2.2 Algorithms and Techniques

The software 1.0 model uses simple iterations using for loops/ while loops and conditional branches using if-else blocks to classify the input data.

The software model 2.0 which is the project's main focus, uses machine learning algorithms to map the inputs to their output labels by fitting linear/non-linear functions with several unknown parameters. The goal of the algorithm is to best estimate parameters of the function that will allow it to efficiently map the inputs to their corresponding output labels.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

Neural Networks The problem given is a supervised classification problem. Supervised, since we already know the output labels, and classification because the output labels are discrete. To solve this problem any classification algorithm can be used. A Neural network has been used here to classify the data. Neural networks is a highly efficient and robust algorithm used widely today. It emulates the human brain in terms of how we learn, and process data. Neural networks can represent complex non-linear functions and have applications in complex learning problems such as computer vision. They are flexible and can also be configured to give better results depending on the problem requirements.

Configuration of Neural Networks The following are parameters of the Neural network classifier that can be tuned to enhance performance,

- Neural network model parameters and properties
  1. Number of hidden layers
  2. Number of neurons in each layer
  3. Activation function of each layer
  4. Optimization algorithm
  5. Error function
- Training parameters
  1. Batch size
  2. Learning rate
  3. Number of epochs
- Pre-processing parameters

  Input data can be processed or transformed to another form in order to improve the effectiveness of the algorithm. In general, inputs with multiple uncorrelated features can better train the algorithm.

2.3 Performance metric

Accuracy will be the performance metric to evaluate the efficiency of the machine learning algorithm. Accuracy will be measured on the testing dataset and can be defined in terms of the percentage of correct outputs that the algorithm can produce when compared with our reference test output labels already available in the testing dataset.

$$Accuracy(\%) = (No.\ of\ correct\ outputs/Total\ number\ of\ test\ examples) * 100$$

Initial parameters, i.e weights of the network/ algorithm will be chosen randomly and later fine-tuned to fit the dataset. Since different initial parameters will result in slightly different performances, *mean* and *variance* of the accuracy will also be calculated, to give a clear picture of the algorithm's performance.

3 Methodology

3.1 Data Generation

The software model 1.0 is used to generate the training and test datasets. For inputs ranging between 101 and 1000 corresponding output classifications of 'fizz', 'buzz', 'fizzbuzz' or 'others' are recorded and saved in the training.csv file. For inputs ranging between 1 and 100, the same classification is done and stored in the testing.csv file.

3.2 Data Preprocessing

Each input sample is transformed to a corresponding 10 bit binary representation, which will be fed to Neural network as 10 different features of the same input. This is implemented by creating a list

of 10 binary bit features for every input sample of both the training and test datasets. The output labels are also encoded to 4 different numeric values and bit representations, with each representation corresponding to one of the labels.

3.3 Classification training

This is implemented in Python using the Tensorflow framework and follows the sequence mentioned below.

1. Two Tensorflow placeholders for input and output are declared with appropriate sizes and data types. These placeholders will not hold any value now, but will dynamically be assigned tensors when a Tensorflow session is run.
2. Parameters such as number of neurons in each hidden layer, number of hidden layers, learning rate are set, and the weights of the Neural network are initialized using a normal distribution, with tensor size set according to the size of the layers the weight will be acting on.
3. Connections between the layers are established by an activation of the matrix multiplications of the input layers and input weights to the layer. All the initial layers will use a sigmoid/ tanh /leaky ReLU /ReLU activation function in order to be able to fit complex non-linear functions, while the final layer will use a softmax activation in order to output the probabilities of each class for a corresponding input.
4. A loss function and an optimizer are selected in order to tune the learning algorithm to the training dataset. Here the cross entropy loss function is used with an optimizer algorithm (SGD/ AdaGrad/ Adam/ RMSprop) to arrive at the Neural network weights that will fit the data best, given other hyper-parameters such as learning rate, batch size and number of epochs. Different optimizers follow different approaches on how back-propagation is achieved in the network.
5. Training starts here, number of epochs and number of batches(initally 3000, 100) are set and a TensorFlow session is invoked. The input dataset is split into batches in a random fashion and is passed to the TensorFlow session, simultaneously invoking the optimizer. The loop iterates till the number of epochs specified is reached and then the session terminates.
6. Metrics such as training accuracy and loss are collected after each epoch to observe and evaluate the performance of the model, and a graph to visualize how the network is converging is plotted to help understand the networks quality.
7. Steps 5 and 6 will be repeated a few more times for the same network setting to observe different accuracies that result due to random weight initializations, and the mean and variances of these accuracies will be recorded.
8. Different network settings and hyper-parameters are then tried to observe and record the network setting that gives the best performance, and an output file which gives the classification predicted by the best network setting for a given input is recorded.

3.4 Testing

Now that the model has been trained with the input dataset, its performance on a dataset it has not encountered yet will be able to give a good idea of how good the model is at generalizing and predicting outputs of new unseen samples.

The testing dataset is passed to the TensorFlow session, and a prediction is obtained for each sample. The prediction is taken to be the class which outputs the highest value, which is the probability of that class being the correctly predicted class (this is because of the softmax activation done at the last layer).

The predictions are then compared with the actual output labels in the testing dataset, and the accuracy of the model is calculated. The same is repeated a few more times with different weight initializations to count for the changes in performance that arise due to their random initializations.

3.5 Refinement

The hyper-parameters and network settings such as learning rate, number of epochs and batch size, number of neurons in each layer, number of layers, activation functions, optimization algorithm can be altered and refined to obtain a better model with better performance over the testing dataset. These can be manually altered to find what works best for the problem.

4 Results

The Neural network was trained over different network settings and hyper-parameters to see what works best for the given problem and dataset. Each network setting was trained multiple times over different random initializations to ensure a suitable optima was indeed obtained legitimately by the algorithm and not by accident.

4.1 Optimization Algorithm

In order to select the best optimization algorithm for the network, several Neural network algorithms were chosen to optimize the loss function, and their performances were measured.

The Neural network was run with the ReLU activation function and 1 hidden layer, 100 neurons in the hidden layer, learning rate 0.05 and with batch size 100, 3000 epochs. The following table gives the mean accuracy, variance of accuracy and accuracies of each class of each algorithm while keeping number of nodes, layers and the learning rate constant. The same network was trained 5 times, and the performance metrics were obtained from all 5 trainings.

Optimizer	Accuracy (%) - Mean	Variance	Fizz	Buzz	FizzBuzz
SGD	87	50.8	68.14	82.85	93.33
Adam	70	32	57	68.57	76.66
RMSProp	61.2	12.16	44.44	45.71	96.66
Adagrad	94.8	12.56	95.52	95.71	100
Adadelta	53	0	0	0	0

Table 1. Accuracy measures on the Neural network taken for different optimizer algorithms using the *ReLU* activation function, expressed as percentages.

The AdaGrad Optimizer (Adaptive Gradient) gave the best accuracy given other network and hyper-parameter constraints. Other Optimizers like Adam and RMSProp although are good algorithms, fail to converge with the constraints imposed on the network. On changing the learning rate, these algorithms perform better as can be seen in Table 2.

Optimizer	Accuracy (%) - Mean	Variance	Fizz	Buzz	FizzBuzz
Adam	91.2	19.36	81.48	85.71	100
RMSProp	61.2	12.16	44.44	45.71	96.66

Table 2. Accuracy measures on the Neural network taken for Adam and RMSProp using the *ReLU* activation function, with learning rate 0.008, expressed as percentages.

It can be observed that on changing the learning rate these algorithms converge and perform better. But the AdaGrad Optimizer still performs better and will be used for the remainder of this project.

4.2 Activation Function

Since the AdaGrad Optimizer gave the best performance as the optimizer, the same algorithm was used with different activation functions to see which activation function worked best to fit the given dataset. The Neural network was run with AdaGrad optimizer and the following configurations, 1 hidden layer, 100 neurons in the hidden layer, learning rate 0.05 and with batch size 100, 3000 epochs, to identify the best activation function for this problem. Table 3. gives the accuracy of each algorithm with these network settings and hyper-parameters.

Optimizer	Accuracy (%) - Mean	Variance	Fizz	Buzz	FizzBuzz
Sigmoid	53	0	0	0	0
ReLU	94.8	12.56	95.52	95.71	100
Leaky ReLU	90.2	63.76	81.48	88.57	93.33
tanH	53	0	0	0	0

Table 3. Accuracy measures on the Neural network taken for different activation functions using the *AdaGrad* Optimizer, express as percentages.

ReLU followed by leaky ReLU gave the best results here. The sigmoid and tanh activations were not able to learn anything useful from the data.

4.3 Number of nodes and layers

Since AdaGrad did best with the ReLU activation function, the number of nodes and layers were changed on this setting to see how many neurons the network will need to perform well. The Neural network was run with AdaGrad optimizer with the ReLU activation function and the following configurations, learning rate 0.05 and with batch size 100, 3000 epochs. Table 4. gives the accuracy this network setting with different number of neurons and different number of hidden layers.

No. of layers	Neurons per layer	Accuracy (%) - Mean	Variance	Fizz	Buzz	FizzBuzz
1	100	94.8	12.56	95.52	95.71	100
1	150	97.4	1.04	92.59	97.14	100
1	250	97	1.6	92.59	100	96.66
2	100	95.6	12.16	89.62	94.28	93.33
2	150	97	4	88.88	100	100
2	250	97.4	0.24	91.11	100	100

Table 4. Accuracy measures on the Neural network taken for different layers and number of neurons, expressed as percentages.

All the networks performed similarly, but the Network with 2 layers, each with 250 nodes outperformed the other models with a 97.4% accuracy and low variance of 0.24.

4.4 Learning Rate

The optimizer, activation function and the number of layers and neurons had thus been been determined. The learning rate was then modified to see if it causes the algorithm to perform better. Although the Adagrad Optimizer adapts the learning rate as it progresses, causing the learning rate to fall with each iteration, the initial learning rate is set manually and the optimal value of this initial learning rate must be found.

Learning Rate	Accuracy (%) - Mean	Variance	Fizz	Buzz	FizzBuzz
0.005	53.6	1.44	2.22	0	0
0.01	96.0	4.8	87.04	97.14	96.66
0.04	96.6	1.84	87.40	100	100
0.05	97.4	0.24	91.11	100	100
0.06	97.6	0.64	91.11	100	100
0.07	97.8	0.96	92.59	100	100
0.08	97.8	1.76	92.59	100	100
0.09	97.2	0.56	89.62	100	100
0.1	97	0	89.62	100	100

Table 5. Accuracy measures on the Neural network taken for different learning parameters, expressed as percentages.

The Neural network was run with AdaGrad optimizer with the ReLU activation function and the following configurations, 2 hidden layers with 250 nodes each and with batch size 100, 3000 epochs

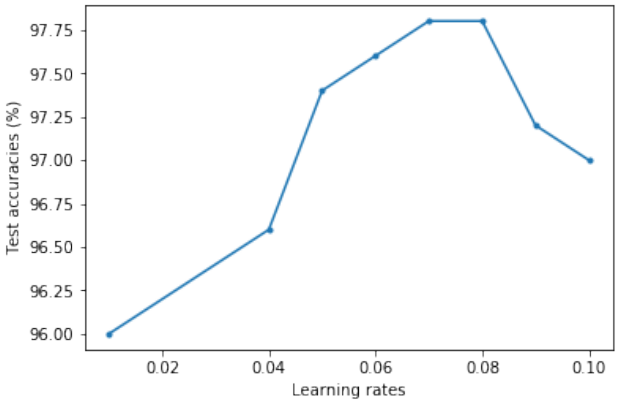


Figure 1: Accuracies of the Neural networks plotted as a function of learning rates

for this problem. Table 5. gives the accuracy of each network setting against the learning rates while keeping batch size, number of epochs, number of hidden layers and neurons, and other hyper-parameters constant. Figure 1 gives the plot of accuracy over learning rate.
It can be observed here that learning rates between 0.06 and 0.08 help the algorithm perform well with good accuracy, and the weights converge well without too many oscillations.

4.5 Batch Size and Number of Epochs

Now that most of the Neural network settings and learning rate have been determined, the batch size and number of Epochs is determined to finally decide on all the Network configurations that would give the best result.

The Neural network was run with AdaGrad optimizer with the RelU activation function and the following configurations, 2 hidden layers with 150 nodes each and learning rate 0.07. Table 7. gives the accuracy of each network setting against the batch size while keeping Epochs at 3000 and Table 8. number of epochs with batch size kept at 150.

Batch size	Accuracy (%) - Mean	Variance	Fizz	Buzz	FizzBuzz
30	97.4	1.04	91.11	98.57	100
50	97.8	0.16	91.85	100	100
100	97.8	0.96	92.59	100	100
150	97.8	0.16	92.59	100	100
200	96.8	1.36	88.88	100	100

Table 7. Accuracy measures on the Neural network taken for different Batch sizes, expressed as percentages.

The batch size of 150 gave the best performance, with low accuracy variance and better *fizz* accuracy as well.

Epochs	Accuracy (%) - Mean	Variance	Fizz	Buzz	FizzBuzz
500	96.4	3.44	91.85	94.28	89.99
1000	96.6	3.84	88.88	98.57	96.66
1500	97.6	2.23	91.11	100	100
2000	96.6	1.04	88.14	100	100
3000	97.8	0.16	92.59	100	100

Table 8. Accuracy measures on the Neural network taken for different Epochs, expressed as percentages.

While changing the number of epochs does not change the mean accuracy much, it does affect the variance for different values, since different random initializations will converge at different rates. It can be observed that the network run over 3000 epochs gives the best accuracy with minimum variance.

4.6 *Model A* - Best Network Setting

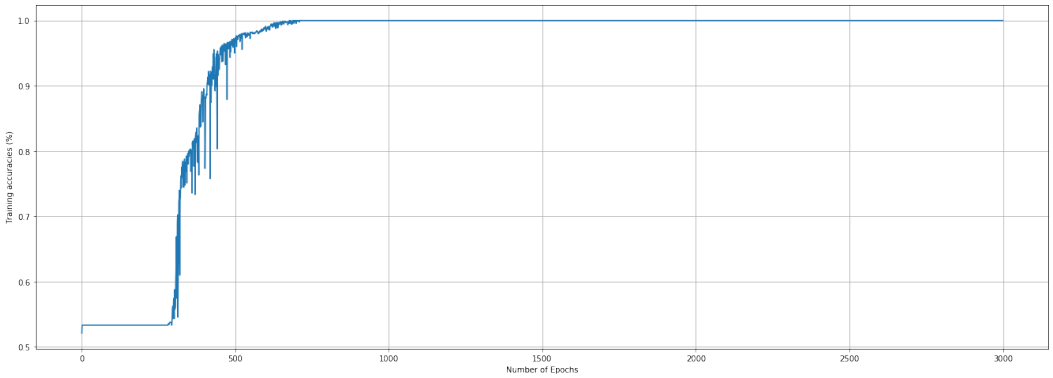
After changing and tweaking different network settings and hyper-parameters, the best configuration for the network for the given dataset was obtained. Let this be *Model A*. *Model A* has the following configuration, it uses the AdaGrad Optimizer, ReLU activation function, has 2 hidden layers with 250 neurons each, learning rate 0.07, number of batches 150 and number of epochs 3000. Performance of this model is given in Table 9. and Table 10. Table 9. gives the accuracy obtained for this model over 5 different random initializations of the networks weights.

Metric	**Model A.0**	**Model A.1**	**Model A.2**	**Model A.3**	**Model A.4**
Accuracy	98	98	98	97	98

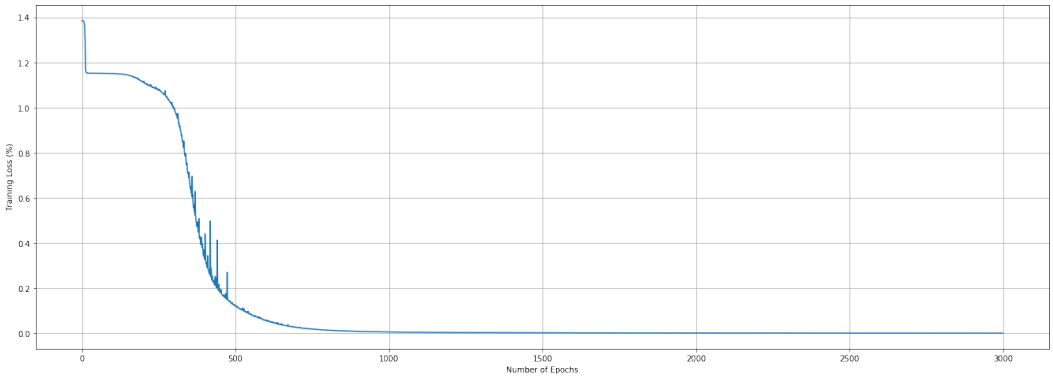
Table 9. Accuracy taken for each random weight initialization for *Model A*, with mean accuracy (%) = 97.8 and variance = 0.16

Accuracy (%) - Mean	**Variance**	**Fizz**	**Buzz**	**FizzBuzz**
97.8	0.16	92.59	100	100

Table 10. Average accuracy measures on *Model A*, expressed as percentages.



(a) Training Accuracy of *model A* over 3000 epochs



(b) Loss function of *Model A* over 3000 epochs

Figure 2: Plot of Accuracy and loss function for Model A

It can be observed in the plots above, that AdaGrad had converged at around 500 to 1000 epochs, and its training loss had been minimized and training accuracy maximized. The number of epochs can be

decreased without a significant change in accuracy, but this would affect and increase the variance in accuracy for the model over different random weight initializations.

4.7 Output

The output of the program using *Model A* for the test input data was recorded in the output.csv file that is generated each time the program is run. It records the output only for the best accuracy found while iterating over different weight initializations, and records the predicted output labels over the testing input data fed to the algorithm.

4.8 Conclusion

The best accuracy measured so far is on *Model A*, with a mean accuracy of 97.8% and low variance over different random weight initializations. The AdaGrad Optimizer along with the ReLU activation function performed really well for this dataset.

References

[1] Tensorflow documentation and API reference.
`https://www.tensorflow.org/api_docs/python/tf`

[2] FizzBuzz code posted in UB learns.

[3] Numpy and pandas documentations.
`https://docs.scipy.org/doc/numpy/reference/`,
`http://pandas.pydata.org/pandas-docs/stable/`

[4] Matplotlib implementation sample.
`https://matplotlib.org/gallery/lines_bars_and_markers/simple_plot.html`

[5] Gradient Descent Algorithms, by Sebastian Ruder.
`http://ruder.io/optimizing-gradient-descent/`