Software 1.0 vs Software 2.0

Shruti Bendale
Person Number: 50289048,
School of Engineering and Applied Sciences,
University at Buffalo, Buffalo, NY, 14214
shrutitu@buffalo.edu

Abstract

The purpose of this project is to compare the logic-based approach (Software 1.0) and the machine learning approach (Software 2.0) to problem solving. The objective is to get familiar with python and machine learning frameworks like tensorflow and keras by implementing the FizzBuzz problem. The two problem solving approaches are then compared to generate the accuracy of the machine learning model. Some hyperparameters are then adjusted to show how the choice of hyperparameters affect the accuracy of the model.

1. Introduction

Software 1.0 vs Software 2.0:

The Software 1.0 is what we're familiar with. It is written in languages such as Python, C, Java, C++ etc. It uses the standard logic and consists of explicitly programmed instructions to the computer [1]. All the instructions are explicitly programmed manually.

Software 2.0, on the other hand, can be written in much more abstract, human unfriendly language, such as the weights of a neural network. No human is involved in writing this code because there are a lot of weights involved [1]. Machine learning frameworks like tensorflow, PyTorch, Gluon, sklearn, etc. are available for creating neural network models that can be trained to solve complex problems. 'tf.keras' is Tensorflow's implementation of Keras which provides high-level API to build machine learning models. After a model is constructed, it is trained on training and validation datasets. The performance of this trained model is then compared with the performance of software 1.0 implementation to measure the accuracy of the model.

Hyperparameters:

A hyperparameter is a configuration whose value cannot be estimated from data. They are often specified manually by searching for the best values using trial and error method. Hyperparameters have to be tuned to fit the model for generating maximum accuracy. Some examples of hyperparameters are the learning rate, batch size, activation functions, dropout, number of epochs, loss function, regularizer, number of layers in the model, etc. The values of the hyperparameters are set manually before learning as contrast to the 'parameters' where the values are learned during the training (E.g.: weights).

2. Experimentation

For the purpose of this project, some hyperparameters were chosen to observe the change in accuracy of the model for different values of the hyperparameters. The hyperparameters that were experimented with are:

i. Learning rate
ii. Activation Functions
iii. Dropout
iv. Optimizer

2.1. Learning Rate (lr):

Learning rate is a hyper -parameter that controls how much we are adjusting the weights of our network with respect the loss gradient [3]. The accuracies observed for different learning rates with a RMSProp optimizer are specified in Table 1.1.

Table 2.1. learning rates comparison

Learning Rate	Accuracy
0.0001	53
0.001	81
0.002	86
0.003	82
0.004	81
0.005	84
0.01	72
0.1	53
0.2	53

a. When learning rate is very less, say 0.0001, the accuracy was 53%. The graph of validation accuracy(val_acc) and testing accuracy(acc) against the number of epochs is showed in fig. 2.1.

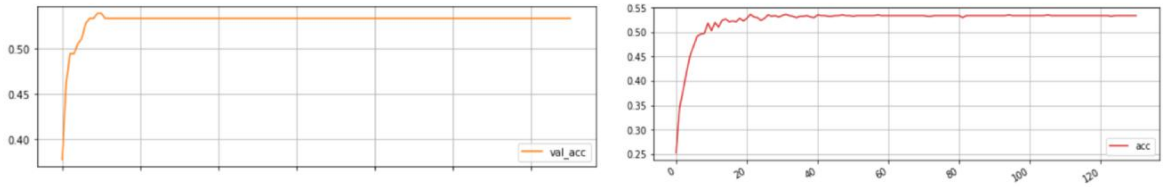

Fig. 2.1: lr=0.0001

b. When learning rate is very high, say 0.1, the accuracy was 53%. The graph of validation accuracy(val_acc) and testing accuracy(acc) against the number of epochs is showed in fig. 2.2.

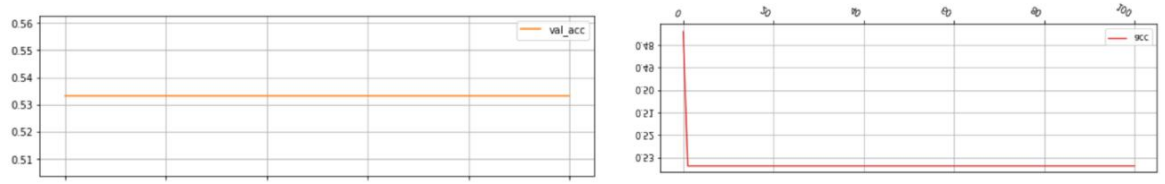

Fig. 2.2: lr=0.1

c. Learning rate of 0.002 gave the best accuracy i.e. 86%. The graph of validation accuracy(val_acc) and testing accuracy(acc) against the number of epochs is showed in fig. 2.3.

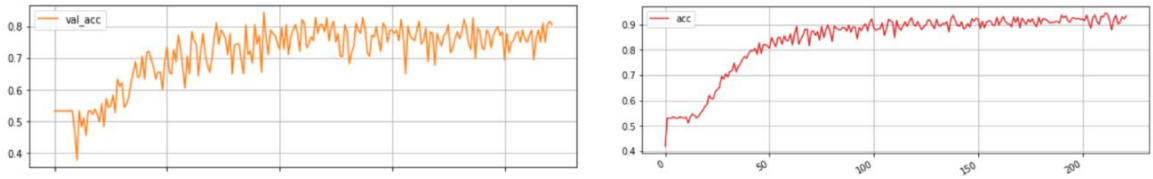

Fig. 2.3: lr=0.001

To summarize the observations, when the learning rate is very less, the model will take a large number of epochs to converge. But here, since the patience is set to 'early patience', it stops the training beforehand as no significant change in accuracy is observed.

When the learning rate is very large, we might overshoot the local minimum and the model will fail to converge or to even diverge.

2.2. Activation function:

The main purpose of activation functions is to convert input signals to a neuron to output signal. This output can be used as the input of the next layer.
Some of the most common activation functions are:

i. **Relu** (Rectifier Linear unit)
f(x)=max (0, x)

ii. **Sigmoid**

$$f(x) = \frac{1}{1+e^{-x}}$$

iii. **Softmax**

$$f(x) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

iv. **Tanh**

$$f(x) = \frac{1-e^{-x}}{1+e^{-x}}$$

Following are some combinations of activation functions experimented with to observe the changes in accuracy:

a. Relu for first layer, softmax for second layer gave 83% accuracy for the model. This is the combination of activation functions that gave the best accuracy. (Refer Fig. 2.4)

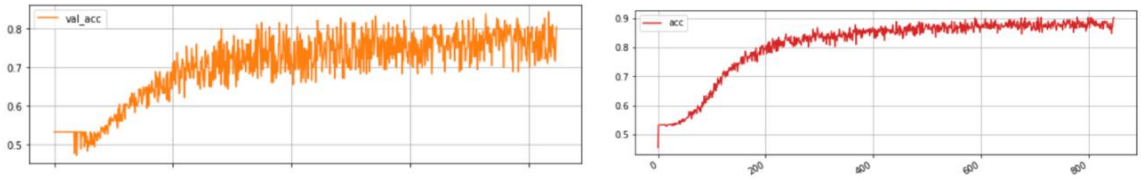

Fig. 2.4

b. Sigmoid activation function for first layer, softmax for second layer gave 53% accuracy. (Refer Fig. 2.5)

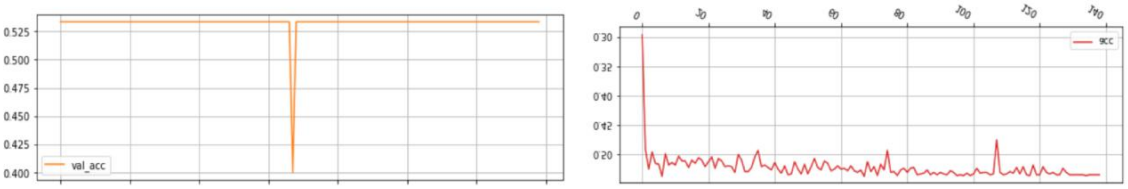

Fig. 2.5

c. tanh for the first layer, softmax activation function for the second layer gave the accuracy of 14%. (Refer Fig. 2.6)

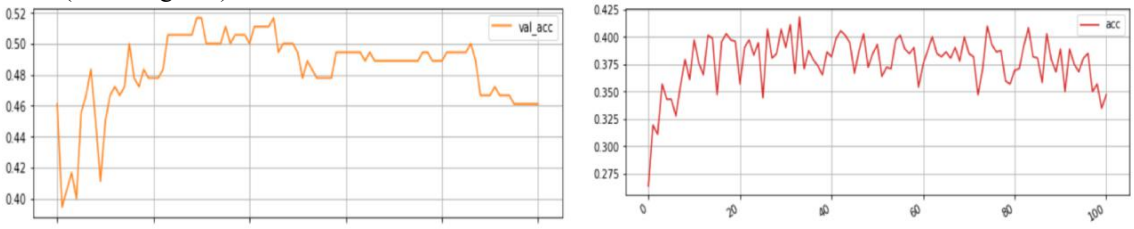

Fig. 2.6

2.3. Dropout:

The term "dropout" refers to dropping out units (both hidden and visible) in a neural network. It is done to prevent overfitting. Dropout is an approach to regularization in neural networks which helps reducing interdependent learning amongst the neurons [4]. The following observations were made when the optimizer was RMSprop and the **number of neurons in hidden layer was 2048**.

a. Dropout=0.001 gives accuracy of 97%. (Refer Fig. 2.7)

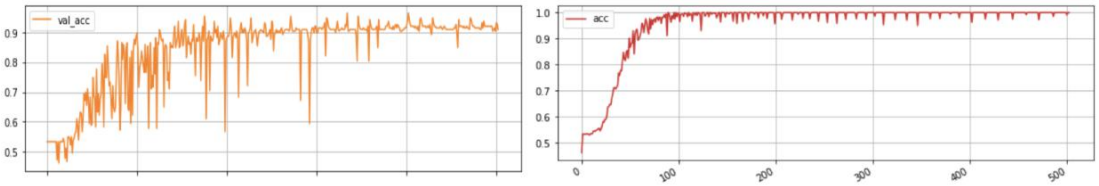

Fig. 2.7

b. Dropout=0.5 gives accuracy of 92%. (Refer Fig. 2.8)

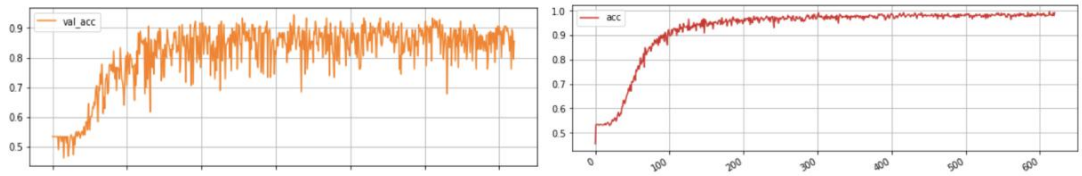

Fig. 2.8

c. Dropout=0.9 gives accuracy of 66%. (Refer Fig. 2.9)

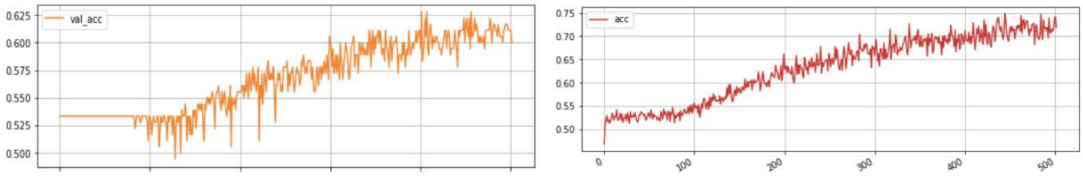

Fig. 2.9

Low dropouts give a higher accuracy. This is because more percentage of neurons are dropped when higher dropout rate is used.

2.4. Optimizers:

Several optimization algorithms have been created over the years. These algorithms use different equations to adjust the model's parameters.
Some of the optimizers available in 'keras.optimizers'are:

a. RMSprop (Accuracy 86%)
RMSprop and Adam give the highest accuracy for the FizzBuzz problem.

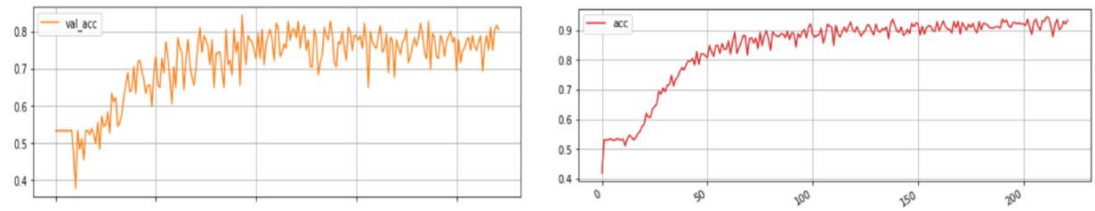

Fig. 2.10

b. SGD (accuracy 5%)

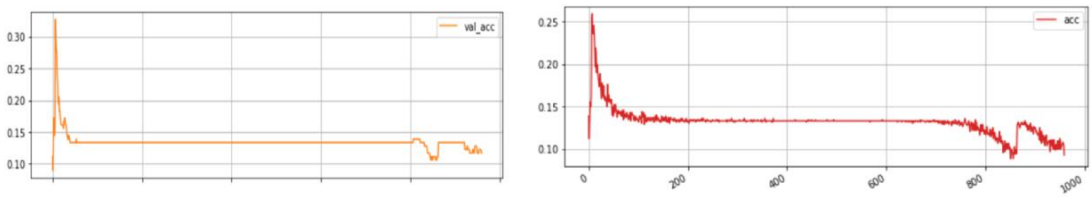

Fig. 2.11

c. Adagrad (Accuracy 53%)

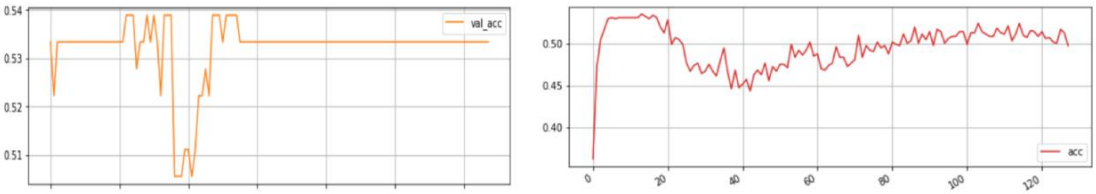

Fig. 2.12

The parameters of these optimizers need to be set according to the model. Some of the parameters common to most the optimizers are clipnorm, clipvalue, learning rate, momentum, decay.

3. Conclusion

Every problem needs a model optimized for that particular problem. The values of the hyperparameters need to be adjusted to fine tune the model to generate highest accuracy for that problem. This can be done either by using hyperparameters of similar models and adjusting them to fit your problem or by trial and error. The combination of hyperparameters that generate the highest accuracy is used for training the model.

4. References

[1]https://medium.com/@karpathy/software-2-0-a64152b37c35
[2]https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/
[3]https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10
[4]https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5
[5] https://keras.io/optimizers/