Part 1

In this section, three models were trained using the DIGITS data. The data was divided into 80% training set and 20% test set. First, a Logistic Regression model was trained. It had a training accuracy of 0.96 and test accuracy of 0.95. Next, the regularization process was applied. Both of the regularization, Ridge (L2) and Lasso (L1), were accompanied with a 5-fold cross validation to determine the best hyperparameter α.

For the Ridge model, the alpha values tested were [1e-15, 1e-10, 1e-8, 1e-4, 1e-3,1e-2, 1, 5, 10, 20]. After cross validation, it was found that the highest validation accuracy, 0.94, occurred at α=0.01. Applying this model for the test set gave an accuracy of 0.93.

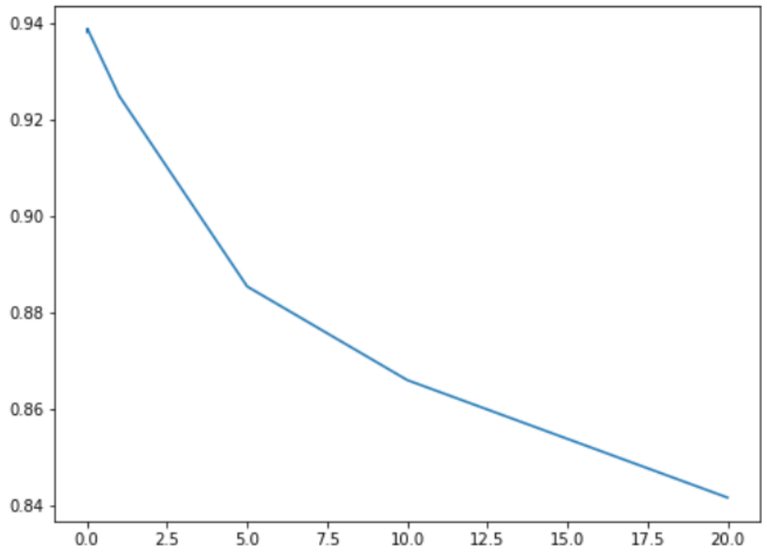


Figure 1. Ridge accuracy versus alpha values.

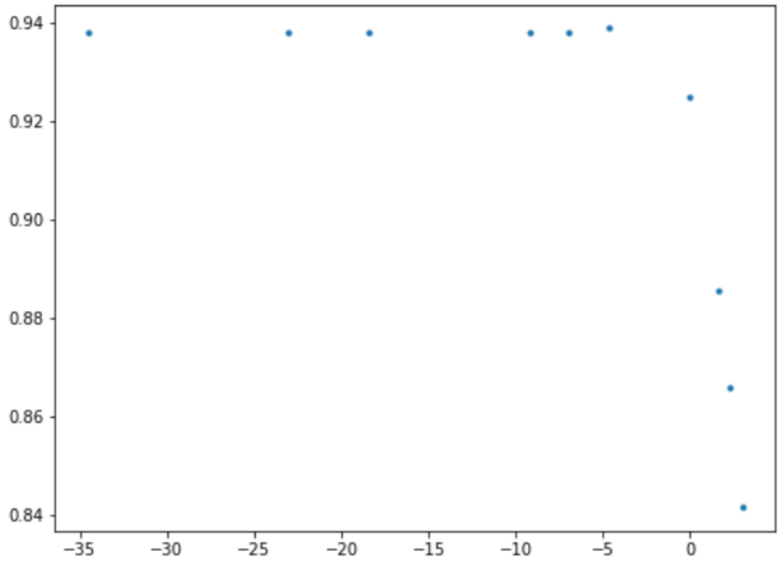


Figure 2. Accuracy---Log(alpha) as x axis

For Lasso, the alpha values tested were [1e-15, 1e-10, 1e-8, 1e-5,1e-4, 1e-3,1e-2, 1, 5, 10]. After cross validation, it was found that the highest validation accuracy, 0.956, occurred at α=1. Applying this model for the test set gave an accuracy of 0.953.

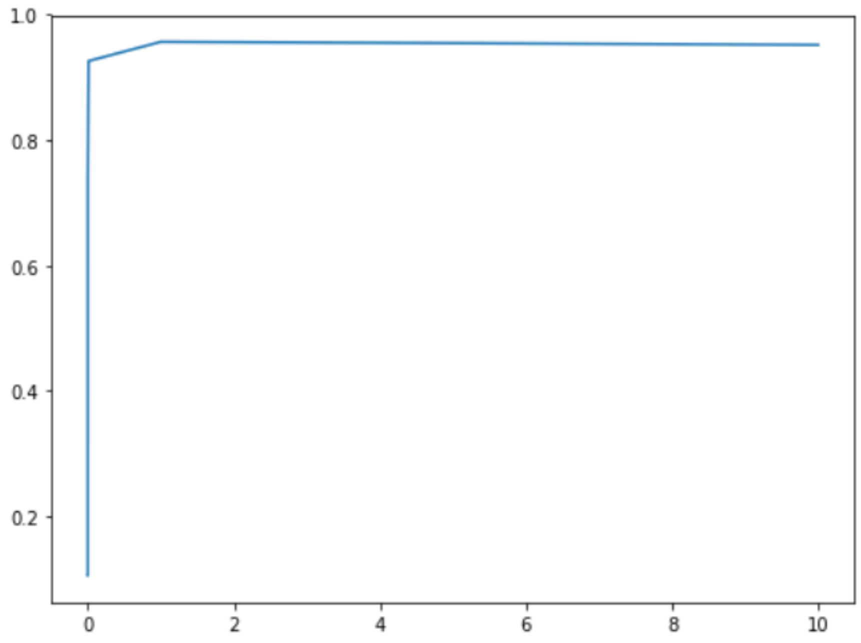


Figure 3. Lasso accuracy versus alpha values.

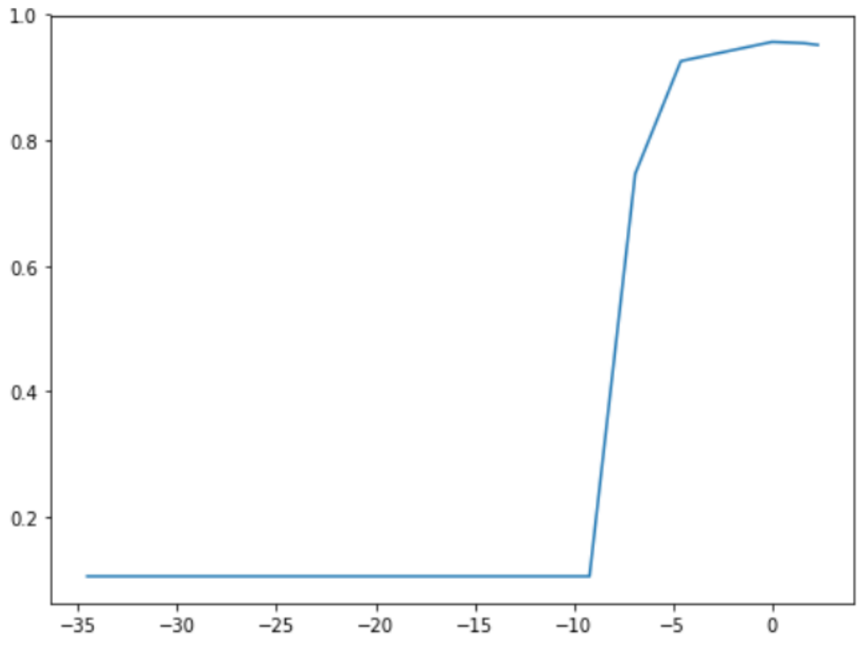


Figure 4. Accuracy -- log(alpha) as x axis

Part 2

1.

In this section, the logistic regression model (ten class) was trained in Tensorflow. The independent variables were color index in an 8*8 matrix and were used to predict digits from 0 to 9 in an image. 80% of the data were in training set and 20% were in test set. First, the data were reshaped into the correct formats (i.e. x in to (? , 64), y into (? , 10), where 10 is the number of labels 0 to 9.) using keras.

Next was to define variables and the loss function. I used x and y to denote input and output, W for weights, and b for bias. The loss function was Cross Entropy, which is $-\sum y * \log(y^{bar})$, where y denotes the actual values and y^{bar}denotes the predictions.

Batch gradient descent was used to train the model. It had the following parameters: batch size = 100, epochs =200, and learning rate =0.001. After training, the final model had a training accuracy of 0.983 and a validation accuracy of 0.969.

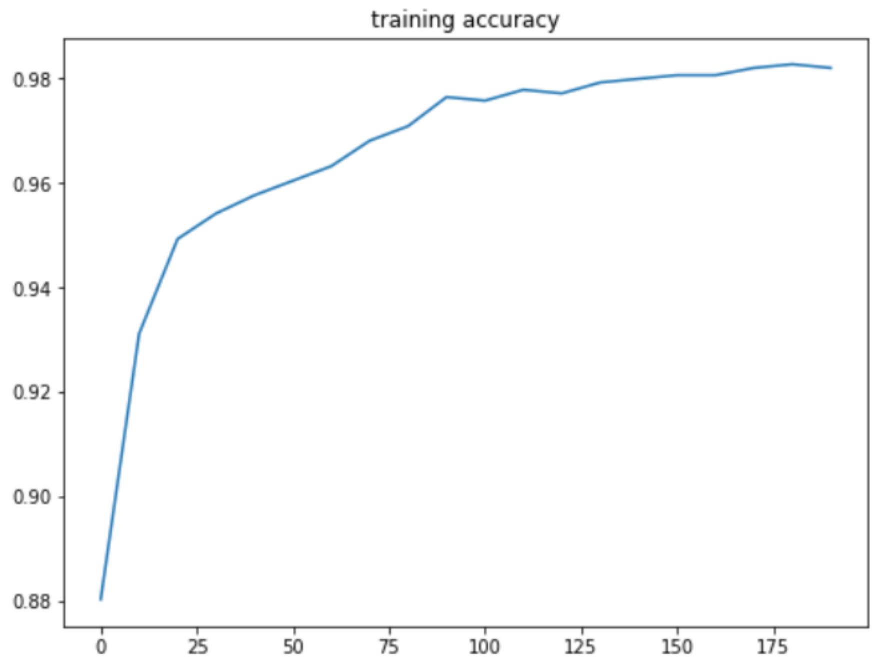


Figure 5. Training accuracy for epochs ----Logistic Regression

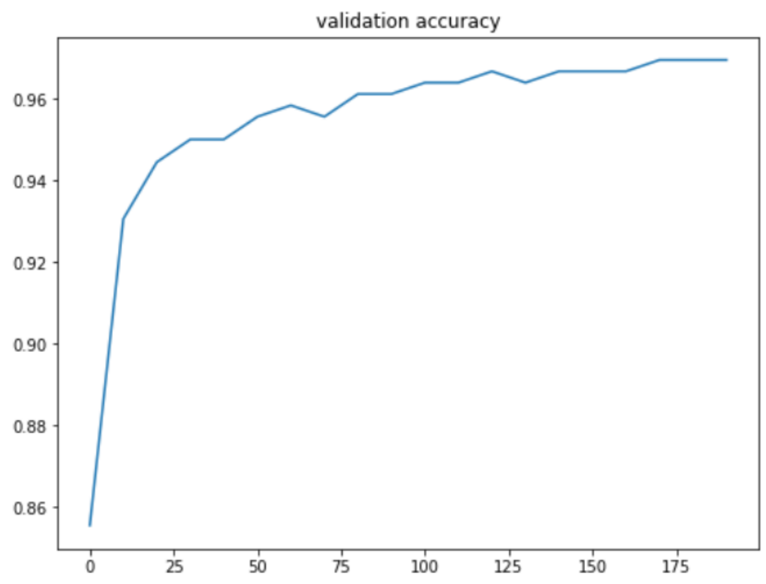


Figure 6. Validation accuracy for epochs ---Logistic Regression

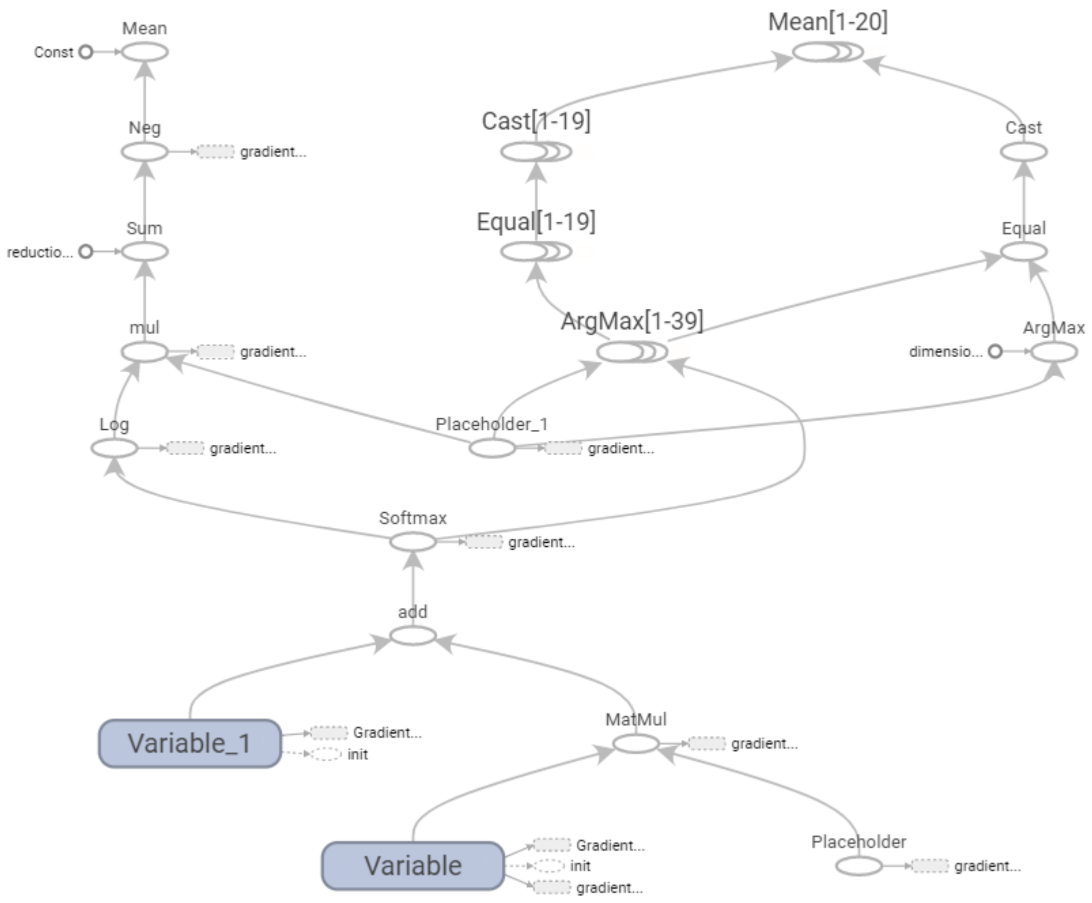


Figure 7. Tensorboard ---Logistic Regression

2.

In the neural network, the input data matrix x was normalized by dividing the maximum value in the matrix, which was 16, since the model would work better for features in the range 0 to 1. I defined X as the input and y as the prediction. each neuron had an output function Z= W*X +b, followed by a tanh activation function. There are three layers of neurons, respectively 300, 200, and 100. The dropout ratio was set as 10%. The gradient descent method was used to train the model and it had the following parameters: batch size =100, epochs =1000, learning rate= 0.001.

After training, the best model had a training accuracy of 0.98, and a validation accuracy of 0.955.

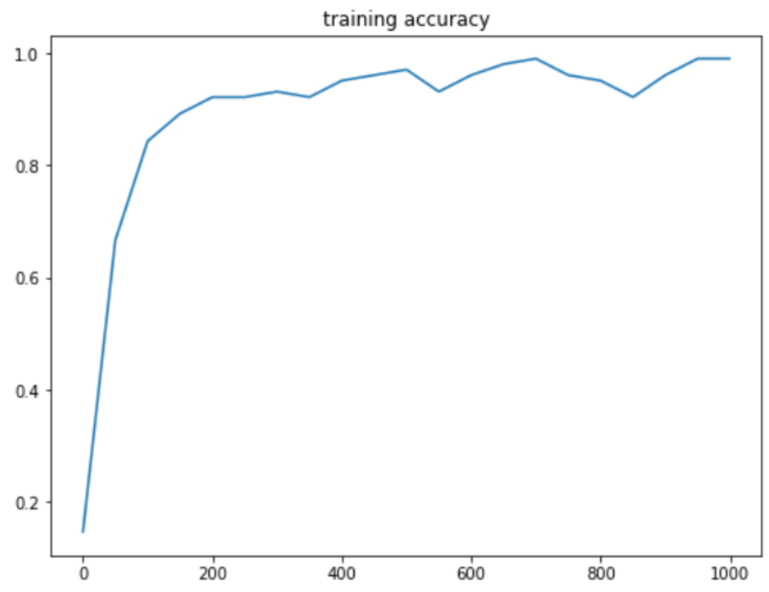


Figure 8. Training accuracy --- Neural network

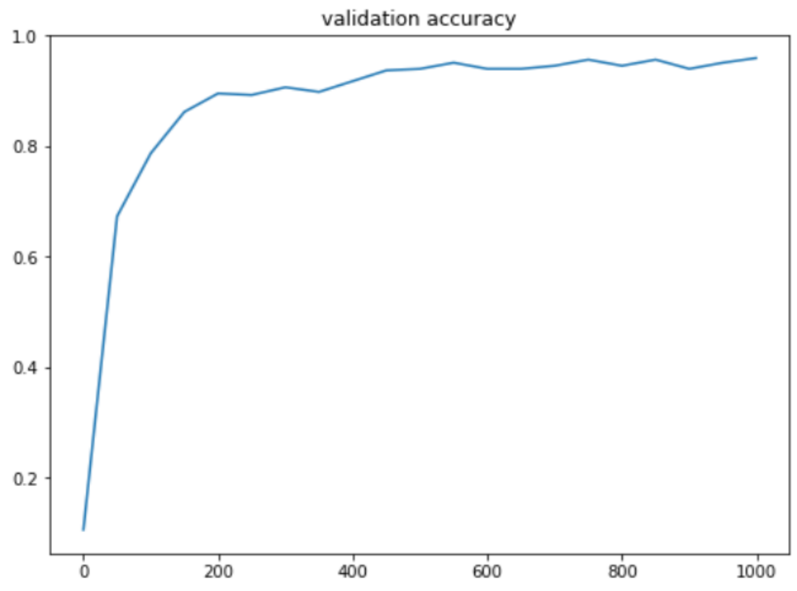


Figure 9. Validation accuracy ---Neural network

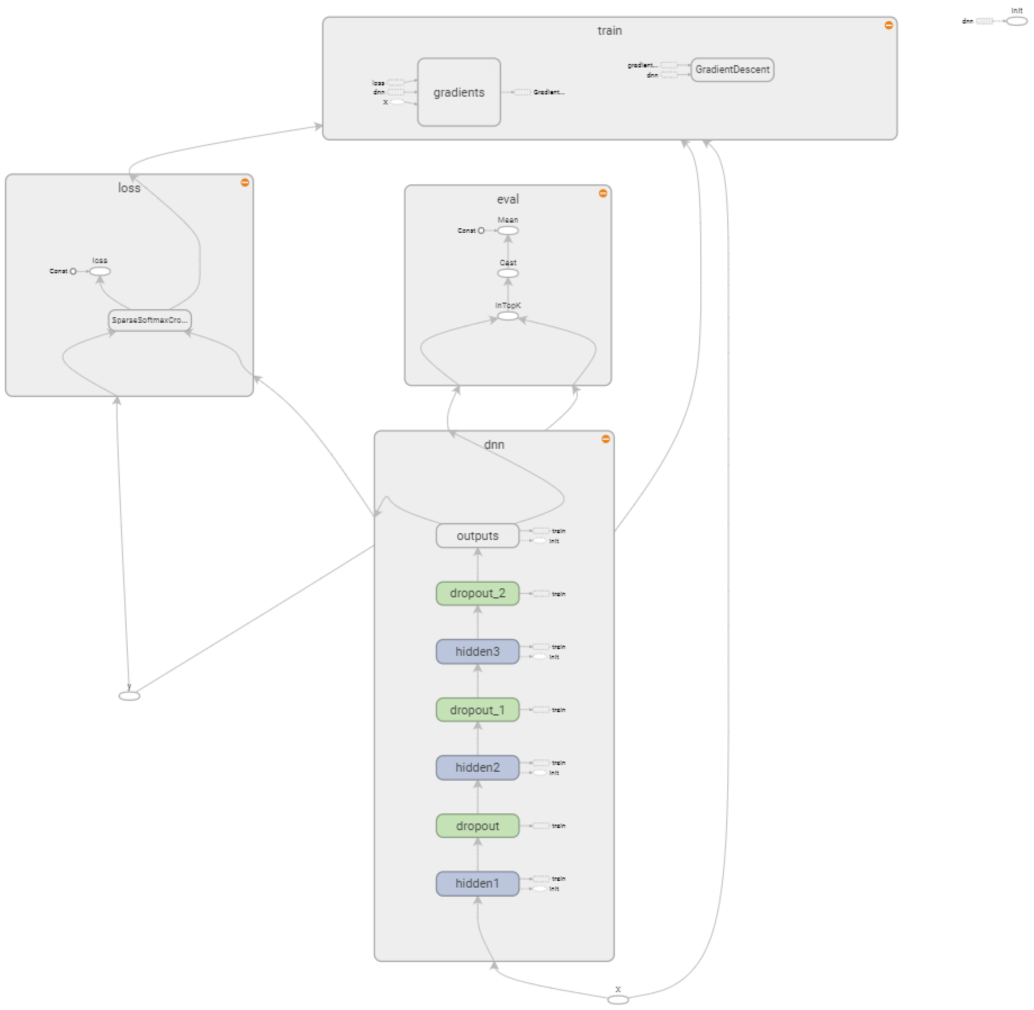


Figure 10. Tensorboard ---Part 2 Neural network.

The neural network model used more weight and bias terms, since between every two layers it requires a great number of terms (i.e. 64*300). It produced a higher training accuracy than logistic regression, but also a lower validation accuracy.

Part 3

1.

Saturated neurons are those who have taken the bounded values of their activation functions, such as infinity in tanh or a logistic function. Therefore, these neurons' activation function parameters are pre-determined. Hence, they are unable to "learn" from the back-propagation, damaging the capability for a neural network to learn from new input, because the number of neurons that can learn in each layer is limited. The reason that they could slow down the training process is that, when these saturated neurons have wrong initial values, it will take a long time for them to be changed to the values that actually solve the problem.

One way to solve this is by using Batch Normalization. It normalizes features that have different ranges to one specific range for all features and hidden layers. The outputs from the previous layer are divided into mini batches, then normalized by subtracting the batch mean and divided by the batch standard deviation. These outputs will have an effect on the weights in the next layer. However, instead of changing the weights, the outputs are denormalized----- they are multiplied by a standard deviation parameter (gamma) and added to a mean parameter (beta). Hence, for each activation, only these two parameters are changed instead of changing all the weights in the layer, so the overall stability of this layer is ensured.

During back-propagation, the activation parameter of each neutron is adjusted by the chain rule. The chain rule denotes that the overall gradient between input and output is simply the product of local gradient (the derivative of the local activation function) and the subsequent gradients (derivatives of the activation function in the other layers). Since the last step of batch normalization involves beta and gamma, the back-propagation will have initial values of beta and gamma, then propagates back to the input by multiplying the remaining local gradients. The final gradient will be the sum of gradient 1, which comes from beta, and gradient 2, which comes from gamma. In other words, the adjusted gradient is the sum of a function beta and a function of gamma, so changing only these parameters can still change the overall gradient.

Batch normalization has multiple effect on the learning process. First, it reduces the shifted amount of hidden layer values (covariance shift). Covariance shift are the shifted amount of data adjusted by the parameters and weights. Reducing the shift is important because, when learning from two very different inputs (i.e. a black-and-white image and a colorful image would have different distributions), a lower covariance shift allows the model to learn from the new input more quickly. Second, because the parameters have lower magnitudes, it creates a regularization effect which prevent overfitting. The additional regularization effect means that fewer dropouts can be used, so more information in the hidden layer is preserved.

2.

Activation functions are used to represent the relationship between input and output in a neural network model. To capture nontrivial relationships, non-linear activation functions are used.

Three common activation functions are logistic (sigmoid), Tanh and ReLU (Rectifier Linear Unit).

Sigmoid:

$\varphi(z) = \frac{1}{1+\exp(-z)}$

First derivative: $\frac{\exp(-z)}{(1+\exp(-z))^2} = \varphi(z)(1-\varphi(z))$

Tanh:

$$\varphi(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

First derivative: = derivative of $\frac{\sinh(z)}{\cosh(z)}$

$= (\cosh(z) * \frac{d}{dz}\sinh(z) - \sinh(z) * \frac{d}{dz}\cosh(z))/\cosh(z)\text{^}2$

$= \frac{\cosh(z)^2 - \sinh(z)^2}{\cosh(z)^2} = 1 - \tanh(z)^2 = 1 - \varphi(z)^2$

ReLU:

$\varphi(z) = \max(z)$

First derivative:$0\ for\ z \leq 0; 1\ for\ z > 0.$

Differences: Sigmoid and tanh convert features from a large range to between -1 and 1, while ReLU convert the negative values in features to 0.

3.

Improving the neural network model --- Batch normalization.

Batch normalization was done for each batch by first normalizing it by subtracting batch mean and divided by the batch standard deviation, both of which are obtained by computing the moving average. Then the values were scaled back by a parameter gamma, and slide by adding a parameter beta. The parameters were kept the same as in part 2 (batch size =100, epochs =1000, learning rate =0.001).

The final model had a training accuracy of 0.99, and test accuracy of 0.955 (It studied the training set better but no significant improvement to unseen data).

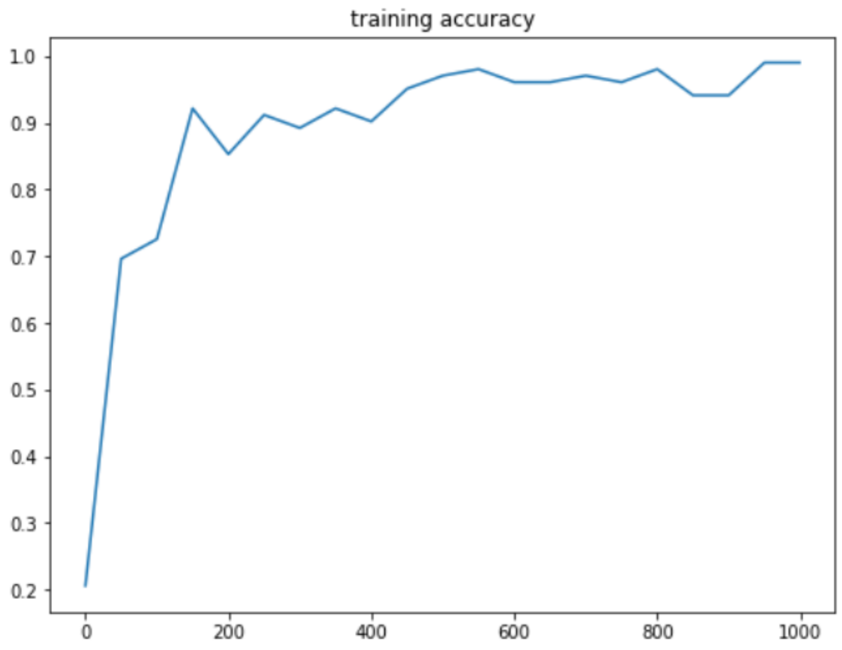


Figure 11. Training accuracy ---Batch normalization

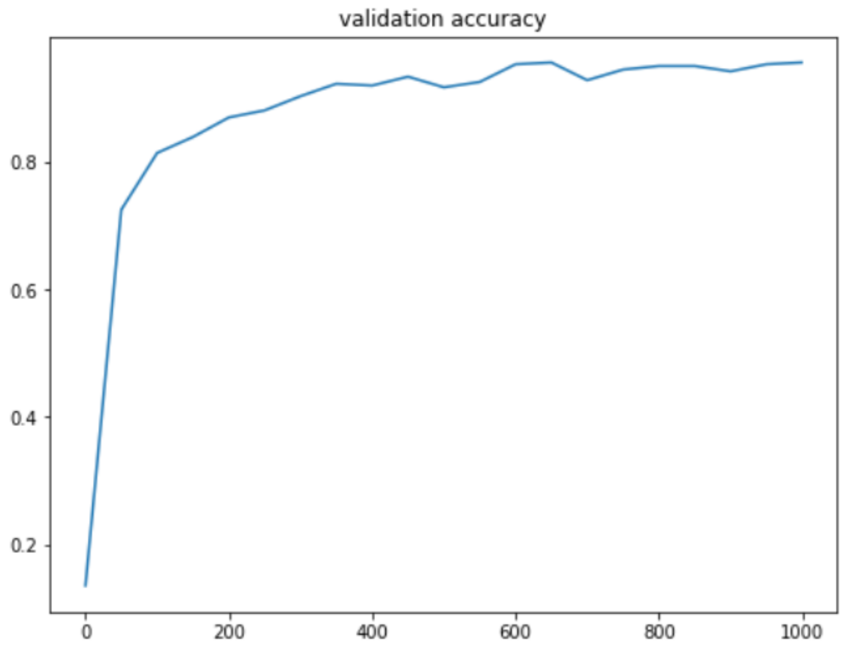


Figure 12. Validation accuracy ---Batch normalization

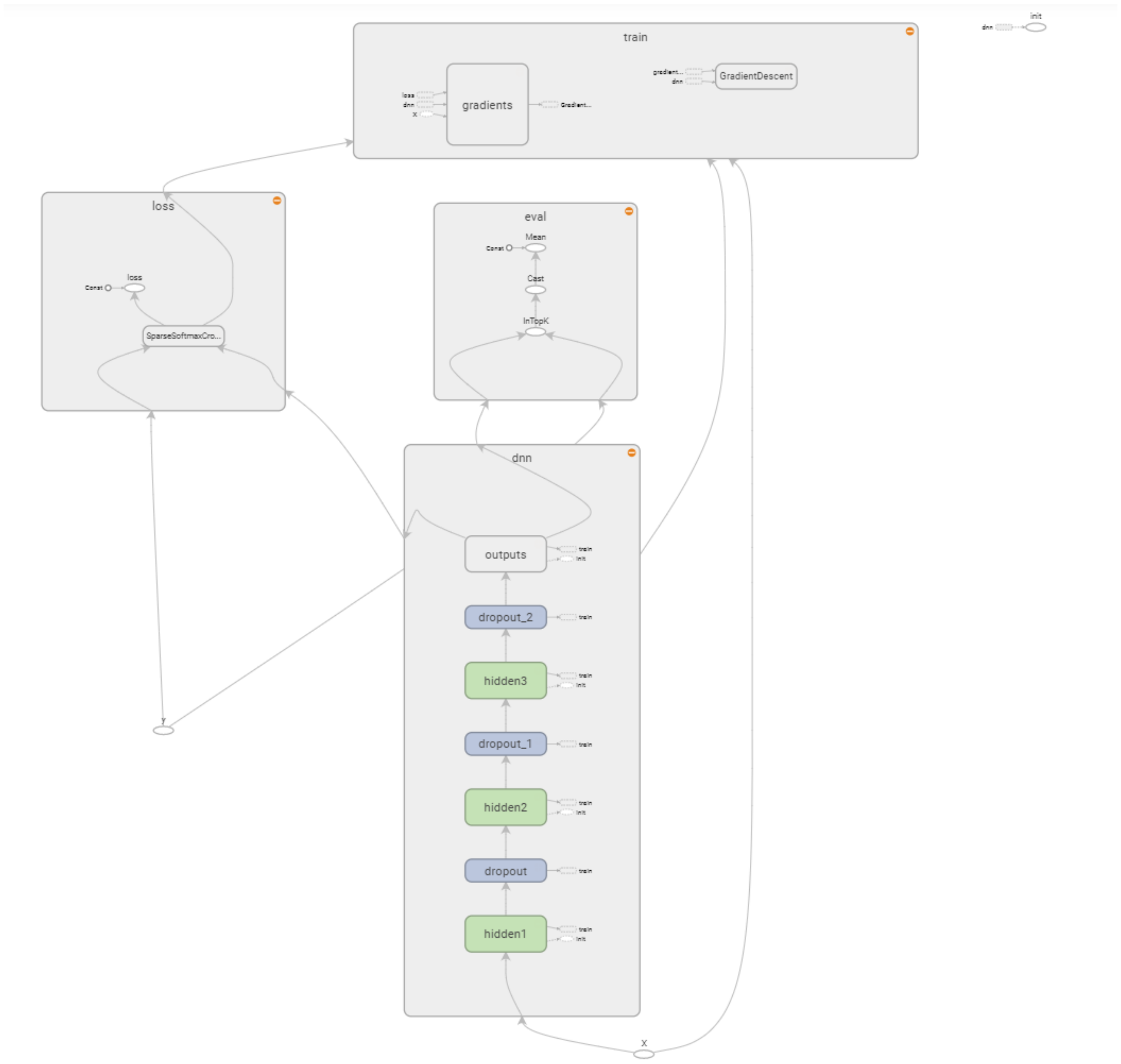
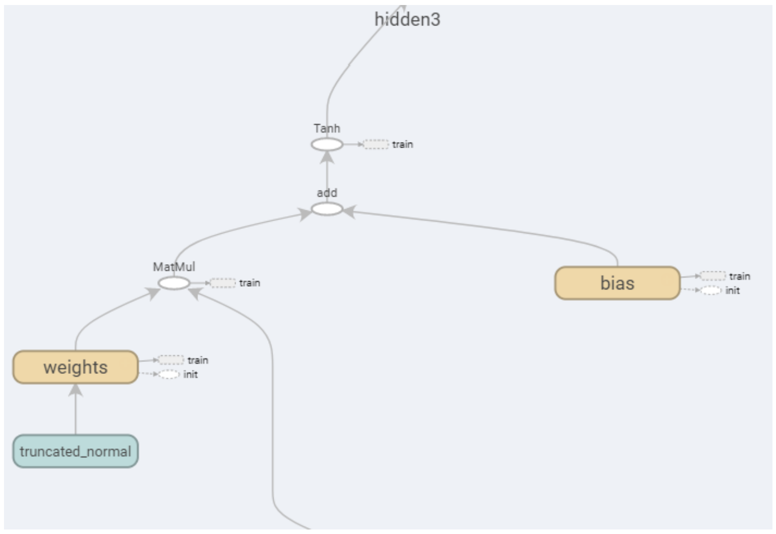
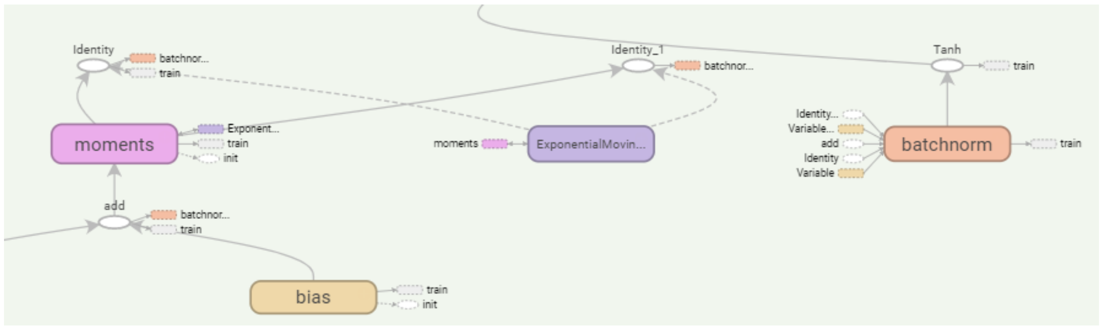
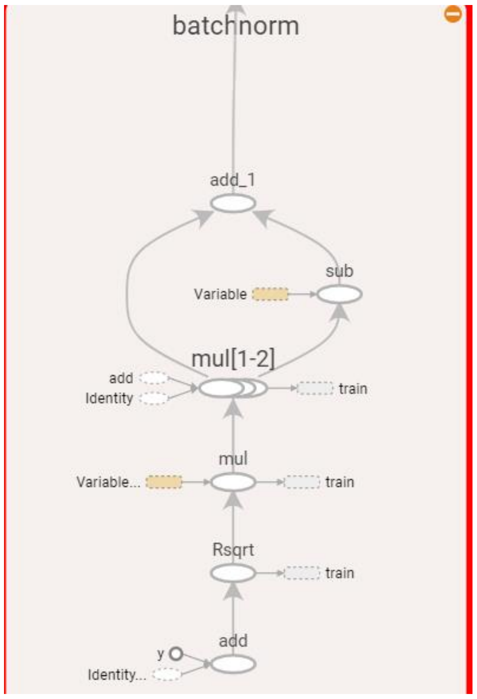


Figure 13. Tensorboard --- Part 3 neural network

This model is different inside each hidden layer. Instead of directly put into the activation function, the output has to go through the batch normalization process before reaching the activation function.


hidden3
Tanh
train
add
MatMul
train
bias
train
init
weights
train
init
truncated_normal


Identity
batchnor...
train
Identity_1
batchnor...
Tanh
train
moments
Exponent...
train
init
moments
ExponentialMovin...
Identity...
Variable...
add
Identity
Variable
batchnorm
train
add
batchnor...
train
bias
train
init


batchnorm
add_1
sub
Variable
mul[1-2]
add
Identity
train
mul
Variable...
train
Rsqrt
train
add
y
Identity...