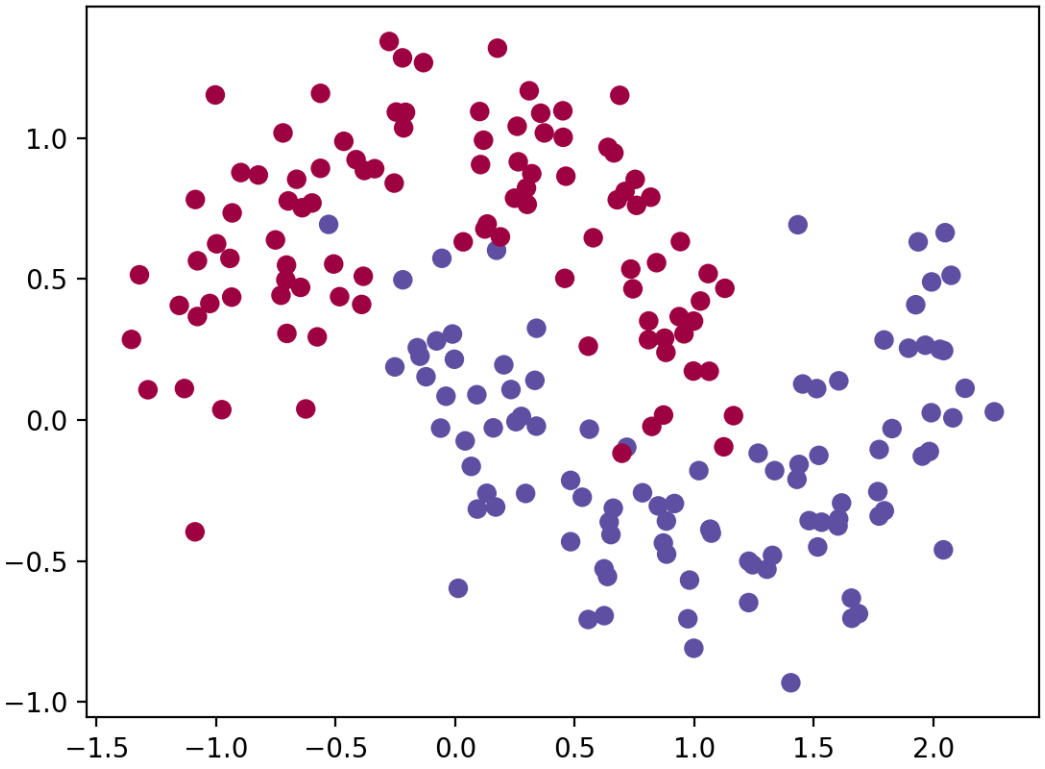
Part 1 Back Propagation in a Simple Neural Network

(a) Make Moon Dataset



(b) Derivatives of activation function

(1)Sigmoid activation:

g(z)=1/(1 + e^-z)=a

g'(z)= (1/(1 + e^-z)) * (1- 1/(1 + e^-z))=a*(1-a)

(2)Tanh activation:

g(z)=(e^z - e^-z)/(e^z + e^-z)=a

g'(z)=((e^z + e^-z)^2 – (e^z - e^-z)^2)/)/(e^z + e^-z)^2

=1-(g(z))^2=1-a^2

(3)Relu activation:

g(z)=max(0,z)

g'(z) = 0 if z<0 and g(z)=0

g'(z) =1 if z>0 and g(z)=z

(c) Derivative of backpropagation

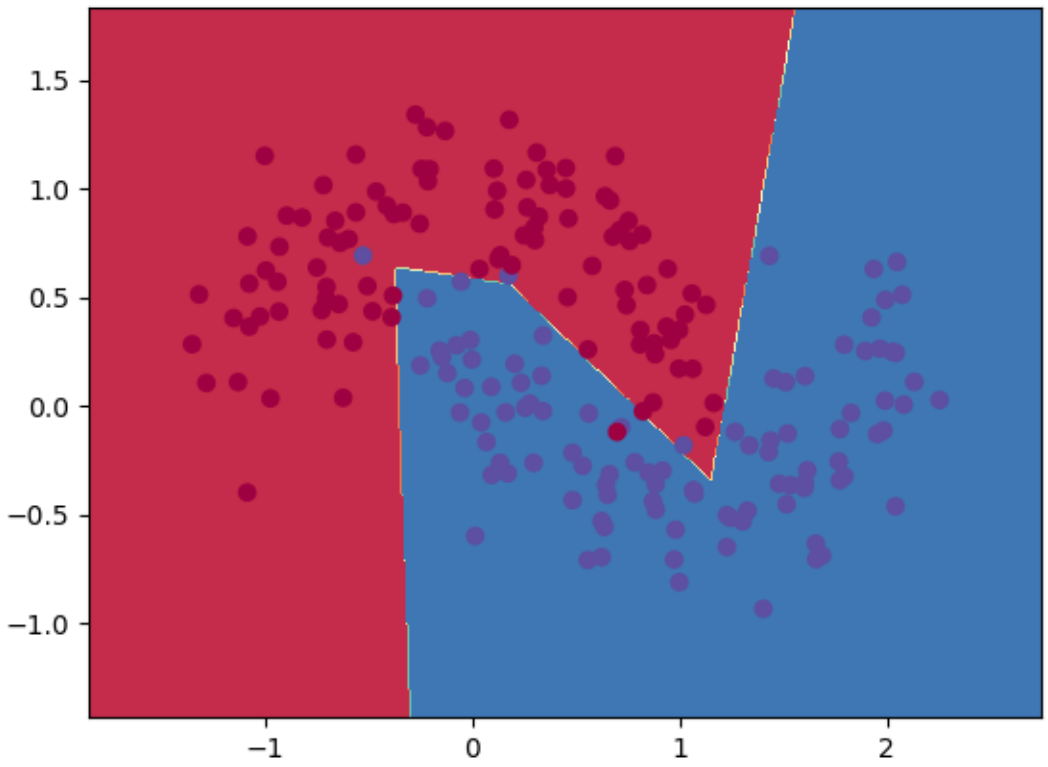
a2=e^f/sum of e^f

for loss function of i= -log(a2)

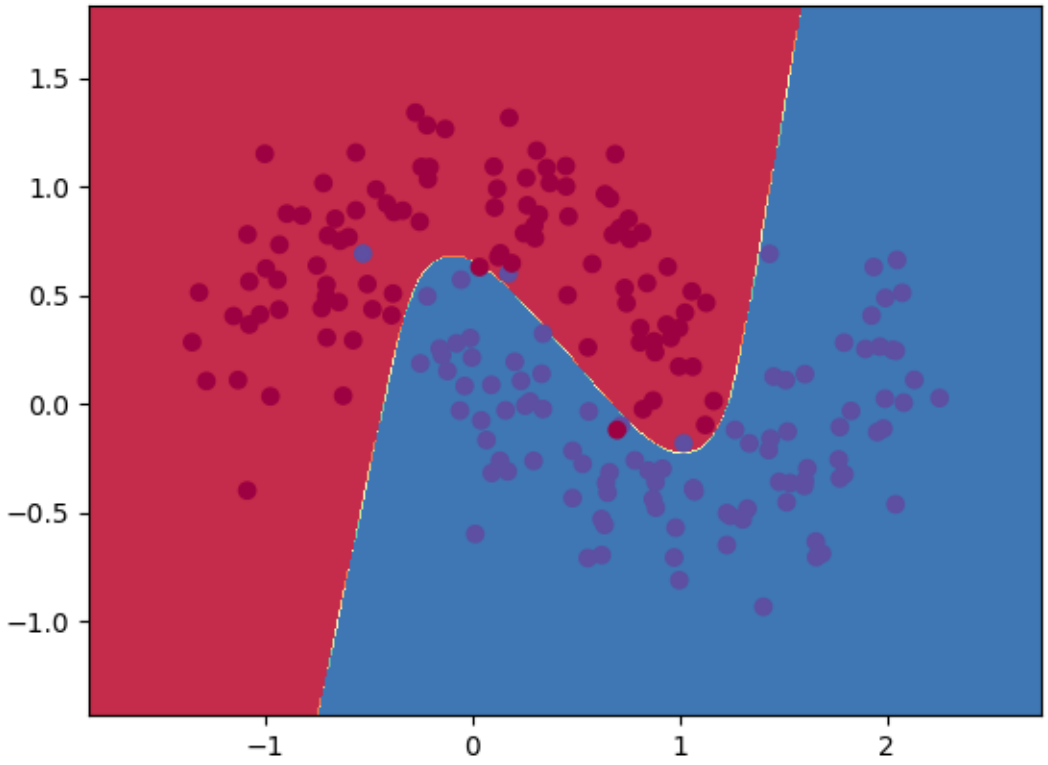
partial of L w.r.t z2:

= -1/a2 * (sum of e^f)^-2 *(e^f* sum of e^f-(e^f)^2)=a2-1
partial of L w.r.t W2 =dz2*dW2=1/m*a1 *(a2 - 1)
partial of L w.r.t a1 =(a2-1)*w2
partial of L w.r.t b2 =dz2=a2-1
partial of L w.r.t W1 =X^t ***** (a2-1)*w2* g'(z1)
partial of L w.r.t B1 = (a2-1)*w2**g'(z1)

(e)
1.
(a). Relu
NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3, nn_output_dim=2, actFun_type='relu')
Loss after iteration 19000: 0.071219



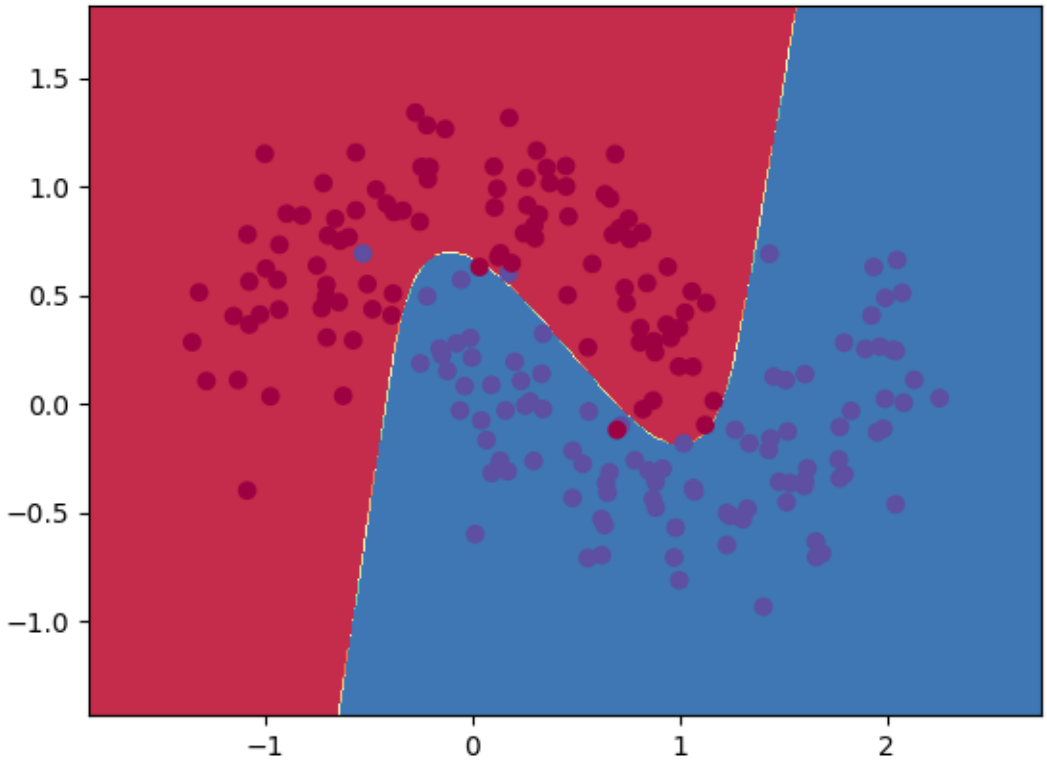
(b).Tanh
NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3, nn_output_dim=2, actFun_type='tanh')
Loss after iteration 19000: 0.070758



(c).Sigmoid

NeuralNetwork(nn_input_dim=2, nn_hidden_dim=3, nn_output_dim=2, actFun_type='sigmoid')

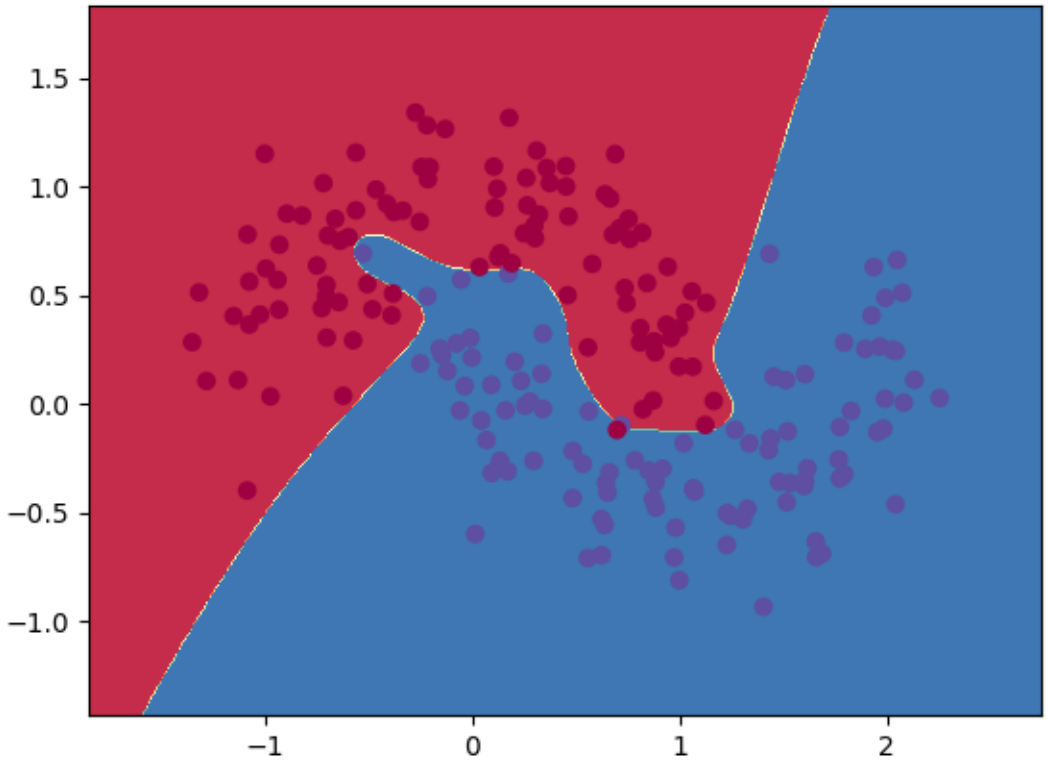
Loss after iteration 19000: 0.078155



Comparison:
From Loss after iterations, Tanh has smallest number and then Relu and then Sigmoid. There is just a tiny difference between Tanh and Relu, which could possiblely from randomness of weight.
From the graph, Relu has straight line which corresponds to the activation function of max(0,z). Tanh is centerd more on zero that mostly fall between -1 and 1. Sigmoid is more right skewed which is reasonable from exponential activation function.
In addition, I observe that Relu is computed quicker than Tanh and Sigmoid as it is numerically easier to compute Relu activation function and avoids extreme values,which could cause very small gradient and kill the process. That is the disadvantage of Sigmoid and Tanh compared to Relu.

(2)

Tanh
Loss after iteration 19000: 0.030908
Hidden units increase to 18
I



I increase hidden units to 18. The loss after iteration decrease dramatically.

Also visually, the decision boundary is less smooth and more curvy than the previous one. The increase of units give more explaining power but tend to over fit the training data.

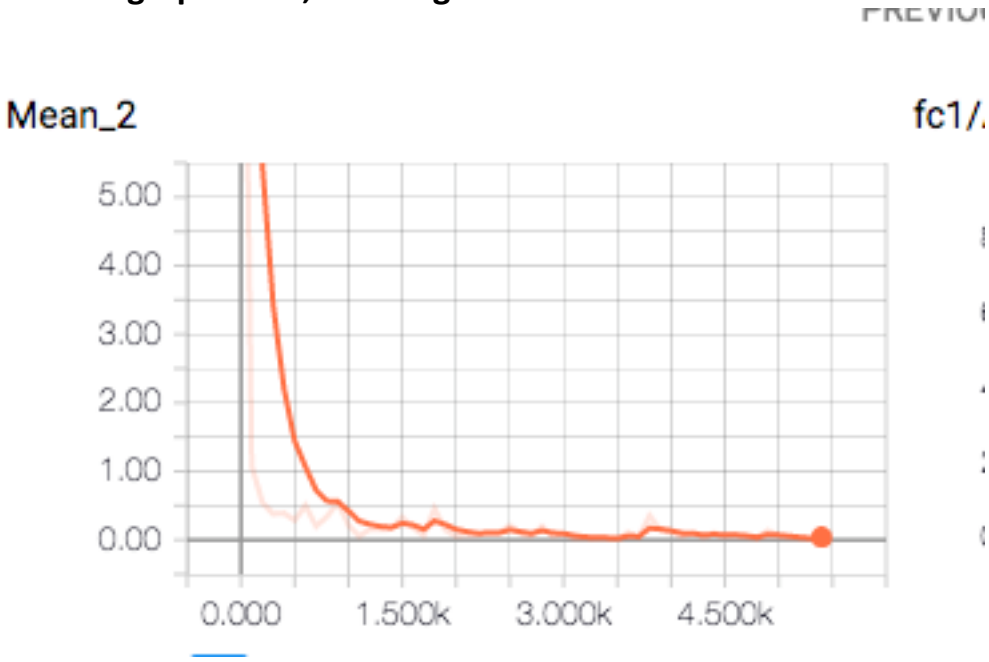
(f)

I write the code but do not have time to debug it. I apologize for that.

The approach I used is to make a cache file. Every time I feed forward, I will save relevant variables in cache in order to compute the gradient later when I go backward. I have attached the code for it. I choose this approach as it appears to be more intuitive for me.

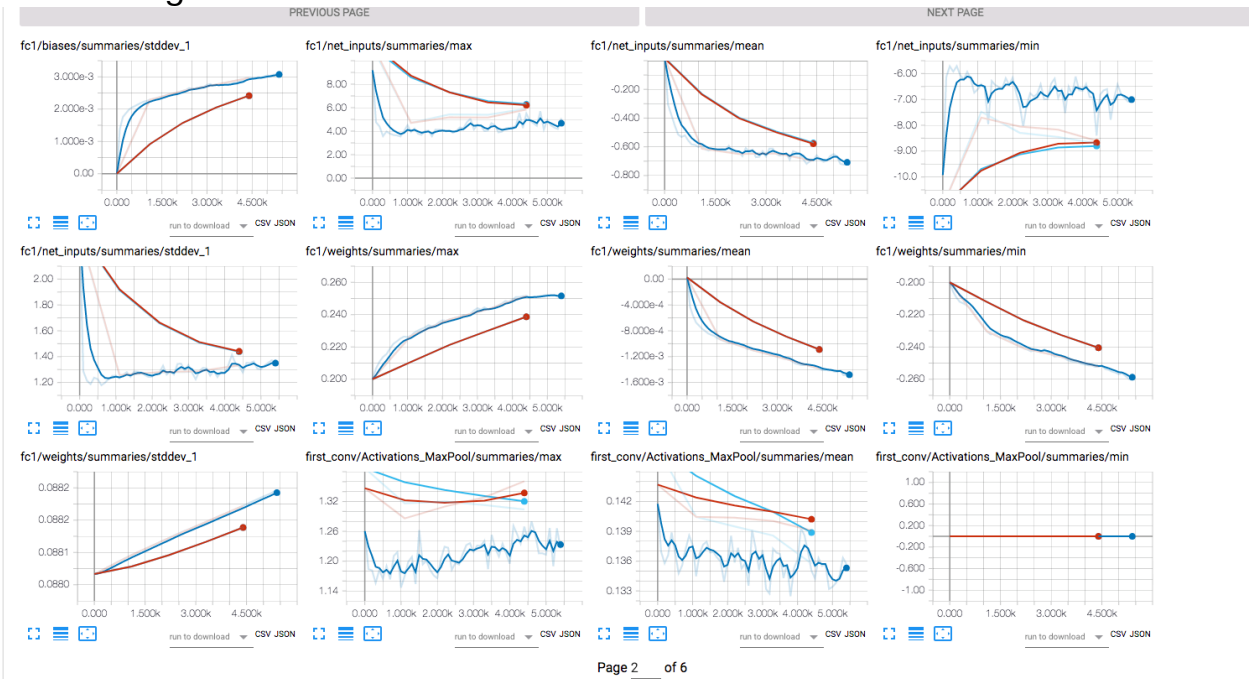
Part 2 Tensorflow

(a).6
test accuracy 0.9855
The training takes 978.913033 second to finish
This is the graph of a6, showing the mean.

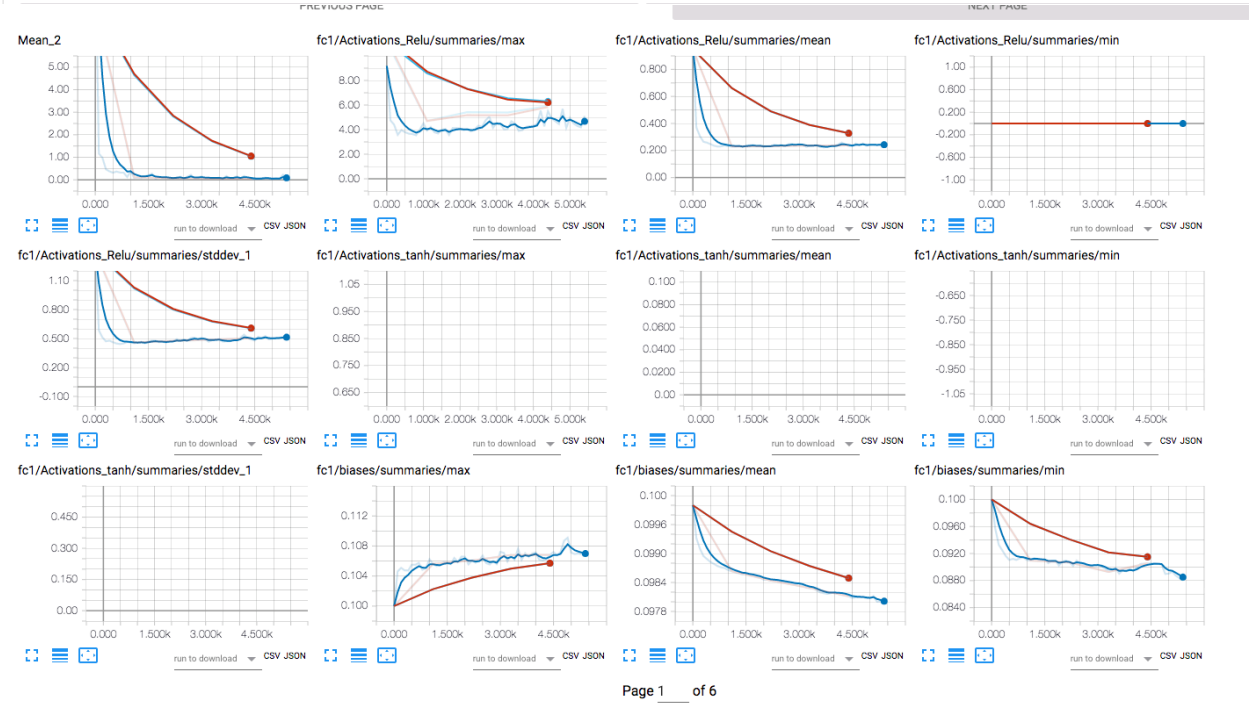


(b)
test accuracy 0.9876

The training takes 1403.682781 second to finish



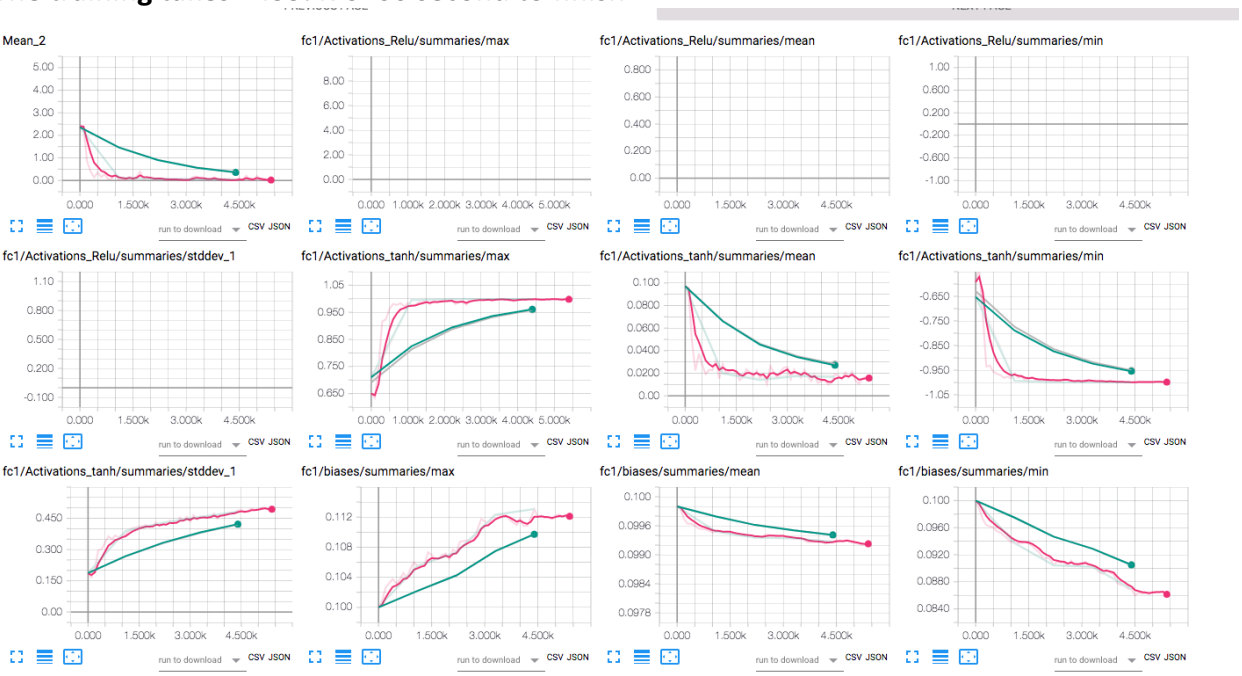
Page 2 of 6

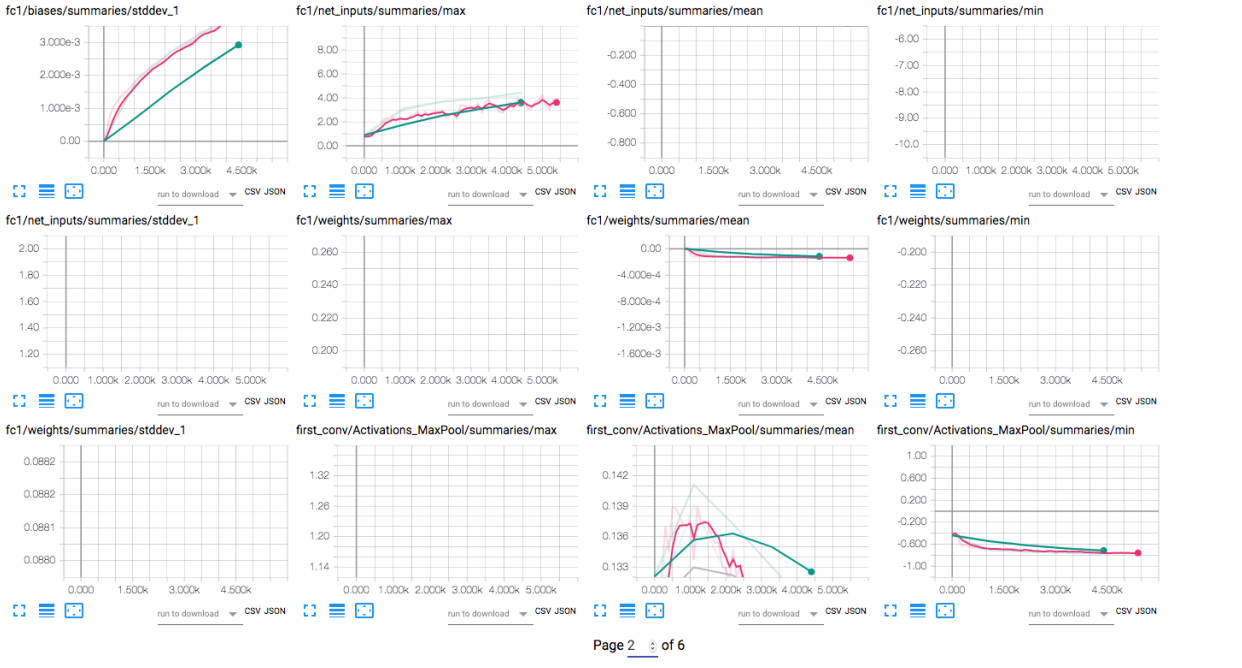


Page 1 of 6

(2c)

test accuracy 0.9871
The training takes 1435.470200 second to finish





Summary,
2b and 2c are experiencing longer time as I record the test and validate error.

Graph differs as I choose different algo and initialization method.