1. Download assignment 2 from Stanford's CS231n:

Data is downloaded from given link.

2. Start with FullyConnectedNets.ipynb. Implement affine_forward function in cs231n/layers.py .

Before implementing the forward pass input is reshaped to be multiplied with weights. Then, bias is added to results to obtain final output.

3. Implement affine_backward function.

In order to calculate gradients backward function is implemented. In this function firstly gradient of x is calculated with mutliplication of output and transpose of weights. Similarly, gradient of weights are obtained by multiplication of reshaped x and output. Lastly, bias gradients are calculated by taking sum of them for each row.

4. Implement relu_forward and relu_backward funcions. Answer the inline question 1.

ReLU is non-linear activation function and it passes values zero as it is, but for the ones that are below zero it passes as zero. Thus, forward pass is implemented with numpy max function.

For backward calculation, because of structure of the ReLU activation function there is no partial derivative calculation. Instead it lets gradients pass backward if they are above 0 if they are not assigns 0 to them.

Inline question is answered at FullyConnectedNets.ipynb file

5. For affine_relu_forward and affine_relu_backward functions in cs231n/layer_utils.py and for the loss layers do not implement anything. Just run the cells. They come free. Enjoy.

Visualizations are given at these cells.

6. Now you will redo two layer implementation as you did in the previous assignment. Go to cs231n/classifiers/fc_net.py and complete TwoLayerNet class (__init__ and loss methods).

In order to run backward and forward passes at first weights and biases are initialized at init function. Then, in loss calculation first layer weights are given into ReLU activation function. Afterwards, result values are fed into affine forward pass to obtain outputs. After obtaining results these results are given into backward process to calculate gradients. After calculation of gradients with backward functions weights and biases are updated.

7. Go to cs231n/solver.py . Use it to train with different hyper-parameters and setup. You can achieve around 50% accuracy. Just do as best as you can. And plot your training curves.

Because of it was a lot of time consuming I had a chance to only test 2 different values for 2 parameters. These are:

- Learning Rate : [1e-4, 1e-5]
- Regularizer: [1e-5, 1e-6]

My best accuracy was 0.481 when learning rate = 1e-4 and regularizer = 1e-5. Plot of traning is given at Figure 1.

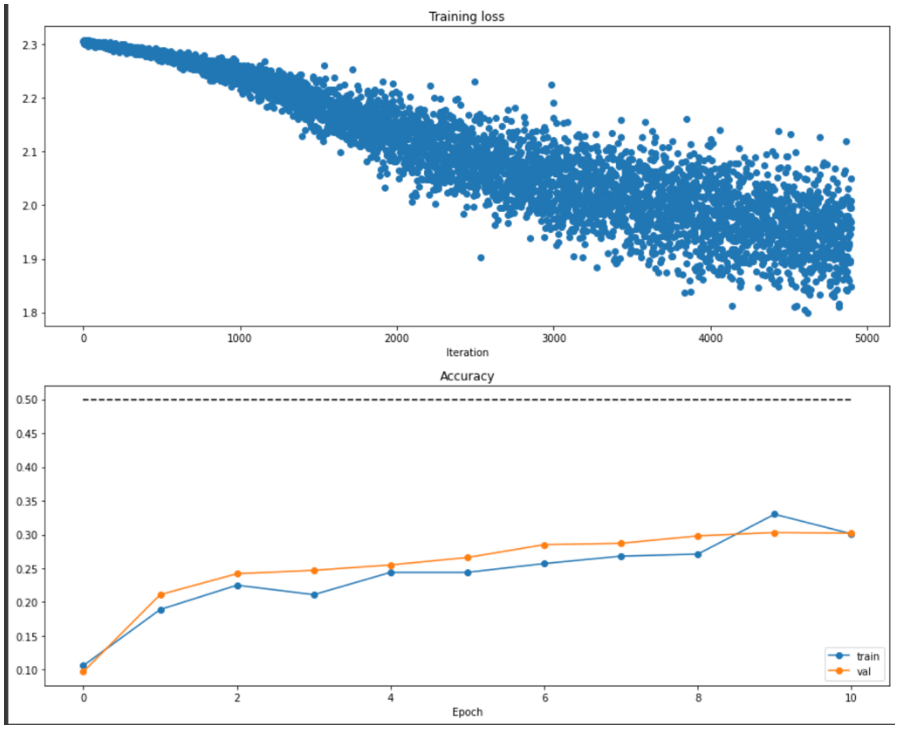

Figure 1: Loss and Accuracy Plot

8. Overfit the small dataset of 50 images with two different networks. Play with learning late and weight scale hyper-parameters to get around 100% accuracy on training set (overfitting). Answer inline question 2.

Model reached overfit when weight scale is 3e-2 and learning rate is 5e-3. Loss history plot is given at Figure 2.

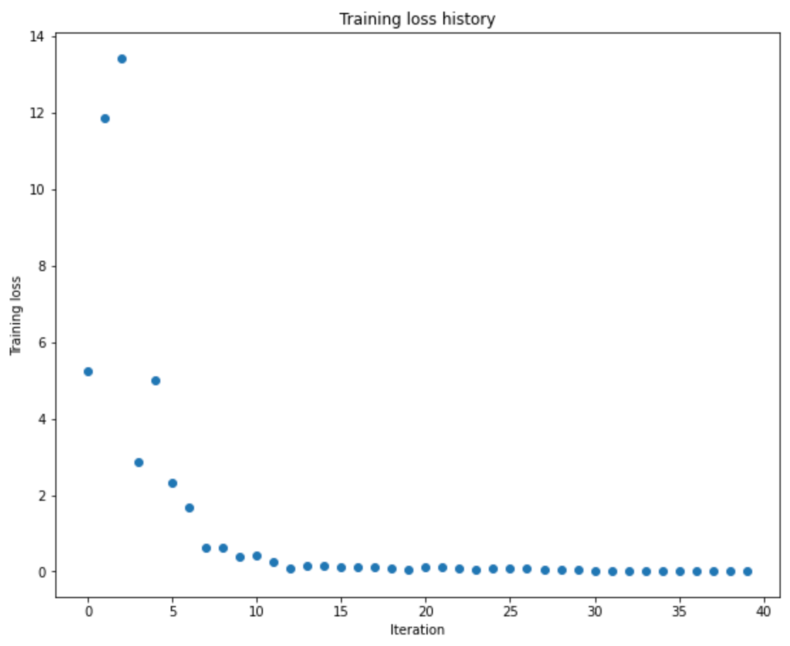

Figure 2: Overfitted Model Loss Plot

Inline question 2 is answered at FullyConnectedNets.ipynb

9**. Go cs231n/optim.py . Implement sgd_momentum, rmsprop and adam update rules. Answer inline question 3. (I know this step contains a lot. Do your best.)**

To implement rules empty sections in optim.py function is filled. Visualization of trains is shown at Figure 3.

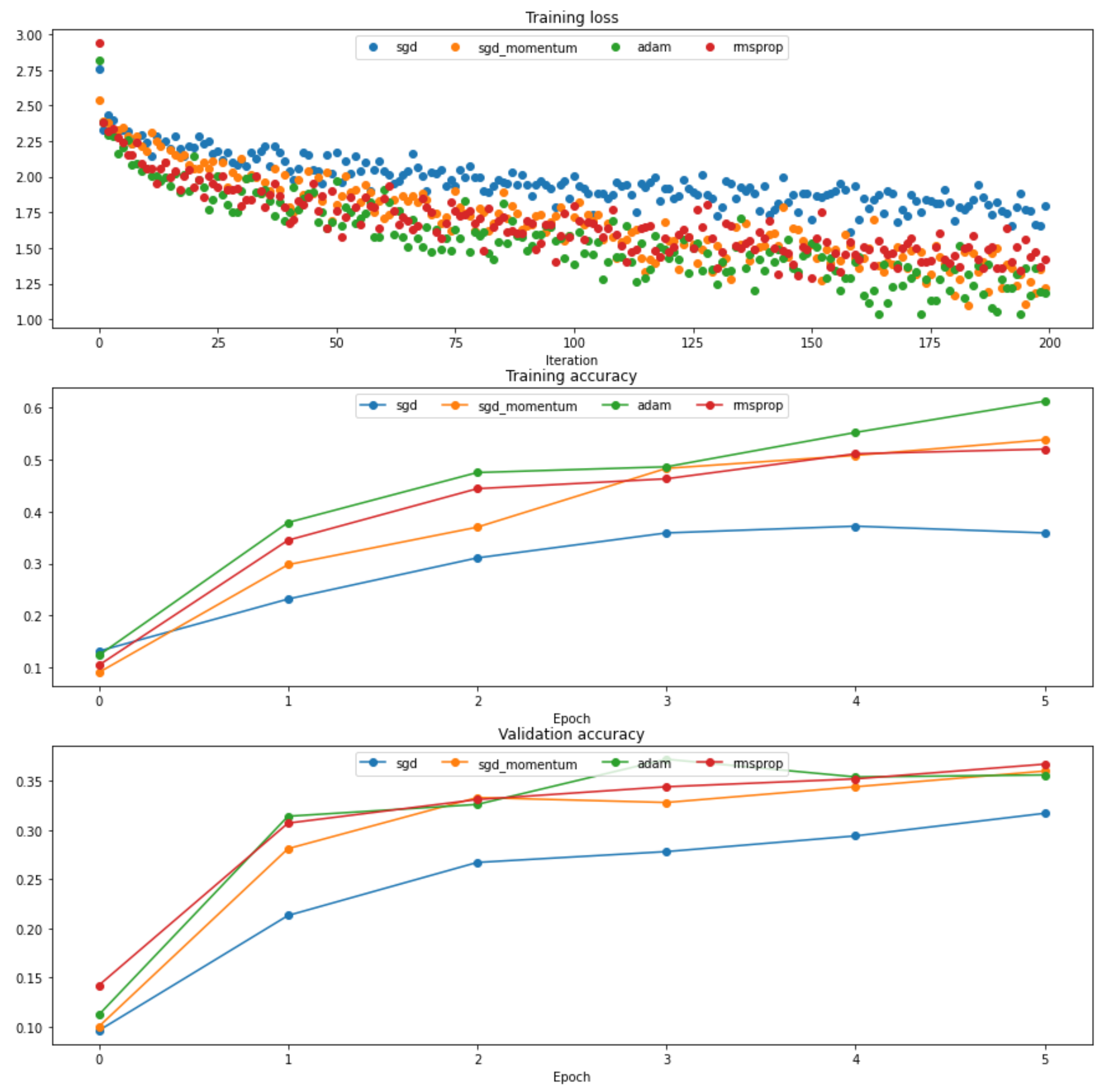

Figure 3: Training with Different Optimizers

Inline question 3 is answered in FullyConnectedNets.ipynb

10. Now using solver, train a better model. You will come back (or maybe not) this step after implementing batch normalization and dropout layers.

Training is done with parameters below:

- Update Rules : [SGD, Adam, RMSProp]
- Learning Rates : [1e-4, 1e-5]

- Layers : (20,30) (Only this combination is trained because of time consumption of training.
- Learning rate decay : 0.98

I could not achieve 0.55 accuracy in my tests. However, it is just about trial and error of input parameters.

11. Now open BatchNormalization.ipynb and go to cs231n/layer.py again. Implement batchnorm_forward, batchnorm_backward and batchnorm_backward_alt functions. (Yeah yeah, getting tough, I'm aware. Stay with me.)

Batch normalization solves a major problem called internal covariate shift. You can utilize a higher learning rate since it makes the data traveling between intermediate layers of the neural network look better. Since it has a regularizing impact, dropout is frequently eliminated. Batch normalization forward pass is implemented to the function.