Assignment 5（Writeup）

Created By:

Group: Chengyin Liu(cl93), Ran Jin(Oliver)(rj23)

COMP 540 – Statistical Machine Learning
Rice University

Problem 1)

a) In neural networks, the number of parameters controls the size of the hypothesis space. Thus, more parameters are structured if there exist more hidden layers. We know that a shallow network has less hidden layers than a deep network has. Thus, deep network would have the model learn more detailed relationships within the data and how the features interact with each other on a non-linear basis.

(Reference:
https://stats.stackexchange.com/questions/222883/why-are-neural-networks-becoming-deeper-but-not-wider)

b) The concept of leaky ReLU is a way to eliminate the dying ReLU problem. Essentially, a leaky ReLU is written as:

$$f(x) = \begin{cases} x, x > 0 \\ \alpha x, x \leq 0 \end{cases}$$

A leaky ReLU is when x $\leq$ 0, instead of having the function going to zero(This is what regular ReLU function does), will it have the function to be some small slope α (e.g 0.05 or so).

(Reference:
https://ayearofai.com/rohan-4-the-vanishing-gradient-problem-ec68f76ffb9b)

c)

	AlexNet	VGG-Net	GoogleNet	ResNet
Years introduced	2012	2014	2015	2015
Number of layers	8	19	22	152
Data Augmentation	Yes	Yes	Yes	Yes
Inception(NIN)	No	No	Yes	No
Dropout	Yes	Yes	Yes	Yes
Top 5 error	16.4%	7.3%	6.7%	3.57%
Convolutional layers	5	16	21	151
Size of fully connected layers	4096,4096,1000	4096,4096,1000	1000	1000

Overall, we see that as newer networks were introduced, the number of layers and convolutional layers have increased which is the defining characteristic for each network. And we see that as number of layers and convolutional layers increase, the top 5 error has decreased from 16.4% to 3.57%.

Problem 2)

a) Since log is a concave function, we have

$$H(S) = H\left(\frac{p}{p+n}\right) = \frac{p}{p+n} log_2\left(\frac{p+n}{p}\right) + \frac{n}{p+n} log_2(\frac{p+n}{n})$$

$$\leq log_2(\frac{p}{p+n} \cdot \frac{p+n}{p} + \frac{n}{p+n} \cdot \frac{p+n}{n}) = log_2(2) = 1$$

Finally, when $n = p$, we have $H(S) = \frac{1}{2} log_2(2) + \frac{1}{2} log_2(2) = 1$

b)

$$Misclassification\ A: \frac{100 + 100}{400 + 400} = \frac{1}{4}$$

$$Misclassification\ B: (200)/(400 + 400) = 1/4$$

$$H(D) = -\frac{1}{2} log(\frac{1}{2}) - \frac{1}{2} log(\frac{1}{2}) = 1$$

$$Entropy\ A:\ H(D_1) = \frac{-1}{4} log(\frac{1}{4}) - \frac{1}{4} log(\frac{1}{4}) = 0.811$$

$$H(D_2) = \frac{-1}{4} log(\frac{1}{4}) - \frac{1}{4} log(\frac{1}{4}) = 0.811$$

Thus, entropy gain for A: $1 - \frac{1}{2} * 0.811 - \frac{1}{2} * 0.811 = 0.189$

$$Entropy\ B: H(D_1) = \frac{-1}{3} log(\frac{1}{3}) - \frac{2}{3} log(\frac{2}{3}) = 0.918$$

$$H(D_2) = -\log(-1) = 0$$

Thus, entropy gain for B: $1 - \frac{3}{4} * 0.918 = 0.3115 > 0.189$, thus, model B is the preferred split.

$$G(D) = 2 * \frac{1}{2} * \frac{1}{2} = \frac{1}{2}$$

Gini Index A: $G(D_1) = 2 * \frac{1}{4} * \frac{3}{4} = \frac{3}{8}$, $G(D_2) = 2 * \frac{1}{4} * \frac{3}{4} = \frac{3}{8}$, thus gini index for A is

$$\frac{1}{2} - \frac{1}{2} * \frac{3}{8} + \frac{1}{2} * \frac{3}{8} = \frac{1}{8}$$

Gini Index B: $G(D_1) = 2 * \frac{1}{3} * \frac{2}{3} = \frac{4}{9}$, $G(D_2) = 0$, thus gini index for B is

$$\frac{1}{2} - \frac{3}{4} * \frac{4}{9} + \frac{1}{4} * 0 = \frac{1}{6} > \frac{1}{8}$$

Thus, model B is the preferred split.

c)

The misclassification rate will not increase when splitting a feature.

Suppose there exists a parent node P with C children, where each of them has $a_i\ positive\ examples, and\ b_i\ negative\ examples\ and\ i = 1, \dots, C.$

Then we have the error rate for parent E_P and children E_C:

$$E_P = \frac{\min(\sum_{i=1}^{C} a_i, \sum_{i=1}^{C} b_i)}{\sum_{i=1}^{C}(a_i + b_i)}, E_C = \frac{\sum_{i=1}^{C} \min(a_i, b_i)}{\sum_{i=1}^{C}(a_i + b_i)}$$

Since the denominators are the same, now we want to show that the nominator for children is less than or equal to the nominator for parent in order to show that $E_P \geq E_C$ always holds which is an indication such that the misclassification rate will not increase when splitting a feature.

We know that $\min(a_i, b_i) \leq a_i$, $\min(a_i, b_i) \leq b_i$,

then $\min(\sum_{i=1}^{C} a_i, \sum_{i=1}^{C} b_i) \leq \sum_{i=1}^{C} a_i$, $\min(\sum_{i=1}^{C} a_i, \sum_{i=1}^{C} b_i) \leq \sum_{i=1}^{C} b_i$,

then we say that $\min\left(\sum_{i=1}^{C} a_i, \sum_{i=1}^{C} b_i\right) \leq \sum_{i=1}^{C} \min(a_i, b_i)$ which is what we desired to show and that concludes that the misclassification rate will not increase when splitting a feature.

Problem 3)

a) $E_{bag} = E_X[\varepsilon_{bag}(x)^2] = E_X\left[\left(\left(\frac{1}{L}\sum_{l=1}^{L}\left(f(x) + \varepsilon_l(x)\right)\right) - f(x)\right)^2\right]$

$$= E_X\left[\left(\frac{1}{L}\sum_{l=1}^{L}\left(\varepsilon_l(x)\right)\right)^2\right] = \frac{1}{L^2}E_X\left[\left(\sum_{l=1}^{L}\left(\varepsilon_l(x)\right)\right)^2\right]$$

$$= \frac{1}{L^2}E_X\left[\sum_{l=1}^{L}\varepsilon_l{}^2(x) + \sum_{1<i,j<L\ (i\neq j)}\varepsilon_i(x)\varepsilon_j(x)\right]$$

$$= \frac{1}{L^2}E_X\left(\sum_{l=1}^{L}\varepsilon_l{}^2(x)\right) + \frac{1}{L^2}E_X\left(\sum_{1<i,j<L\ (i\neq j)}\varepsilon_i(x)\epsilon_j(x)\right)$$

$$= \frac{1}{L^2}\sum_{l=1}^{L}E_X\left(\varepsilon_l{}^2(x)\right) + \frac{1}{L^2}\sum_{1<i,j<L\ (i\neq j)}E_X\left(\varepsilon_i(x)\varepsilon_j(x)\right)$$

$$= \frac{1}{L^2} \sum_{l=1}^{L} E_X\left(\varepsilon_l{}^2(x)\right)$$

$$= \frac{1}{L} E_{av}$$

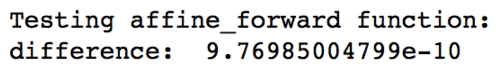
b) Since $E_{bag} = (\frac{1}{L})^2 E_X[(\sum_{l=1}^{L} \varepsilon_l\,(x))^2]$,

Then $(\sum_{l=1}^{L} \frac{1}{L} \varepsilon_l(x))^2 \leq \sum_{l=1}^{L}(\frac{1}{L} \varepsilon_l(x)^2)$ by Jeason's inequity,

Then $E_X[(\sum_{l=1}^{L} \frac{1}{L} \varepsilon_l(x))^2] \leq E_X[\sum_{l=1}^{L}(\frac{1}{L} \varepsilon_l(x)^2)] = \sum_{l=1}^{L} E_X\,[(\frac{1}{L} \varepsilon_l(x)^2)] = E_{av}$

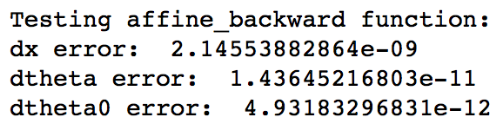
Problem 4)

Problem 4.1.1: Affine layer: forward



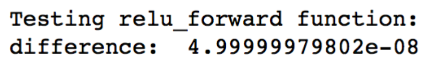
```
Testing affine_forward function:
difference:  9.76985004799e-10
```

Problem 4.1.2: Affine layer: backward



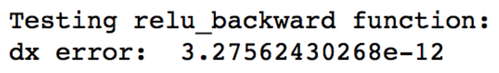
```
Testing affine_backward function:
dx error:  2.14553882864e-09
dtheta error:  1.43645216803e-11
dtheta0 error:  4.93183296831e-12
```

Problem 4.1.3: ReLU layer: forward



```
Testing relu_forward function:
difference:  4.99999979802e-08
```

Problem 4.1.4: ReLU layer: backward



```
Testing relu_backward function:
dx error:  3.27562430268e-12
```

Sandwich layers

```
Testing affine_relu_backward:
dx error:  5.6044169871e-10
dtheta error:  3.11121874094e-10
dtheta0 error:  3.31345931733e-11
```

Loss layers: softmax and SVM

```
Testing svm_loss:
loss:  8.99924379
dx error:  8.18289447289e-10

Testing softmax_loss:
loss:  2.30250987844
dx error:  8.75383352652e-09
```

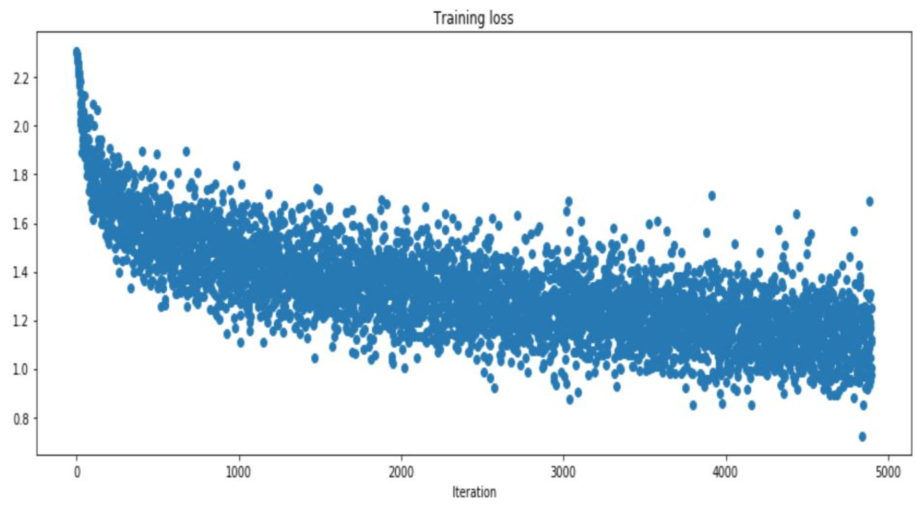
Problem 4.1.5: Two layer network

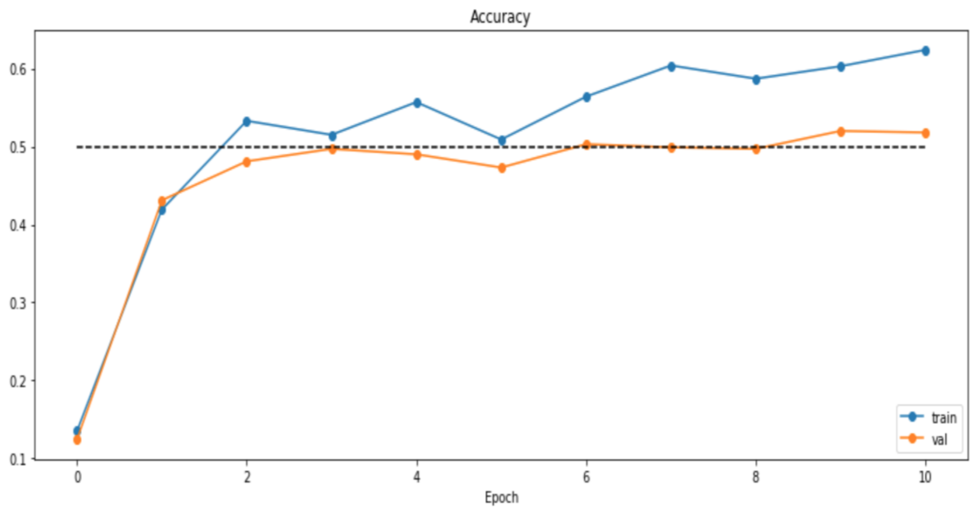
We set the parameters as below,

```
sgd_solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-3,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
sgd_solver.train()
pass
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
theta1 relative error: 1.52e-08
theta1_0 relative error: 8.37e-09
theta2 relative error: 3.30e-10
theta2_0 relative error: 2.14e-10
Running numeric gradient check with reg =  0.7
theta1 relative error: 2.53e-07
theta1_0 relative error: 1.56e-08
theta2 relative error: 2.85e-08
theta2_0 relative error: 9.09e-10
```

Problem 4.1.6: Overfitting a two layer network





Problem 4.1.7: Multilayer network

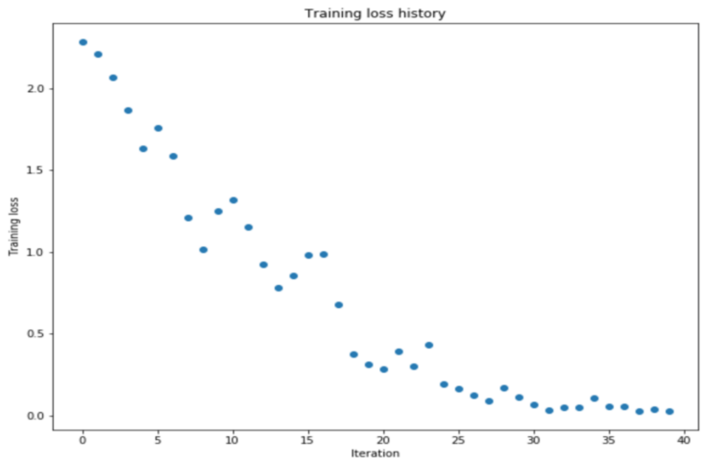
Yes, the initial loss is around lnC = 2.303, where C=10.

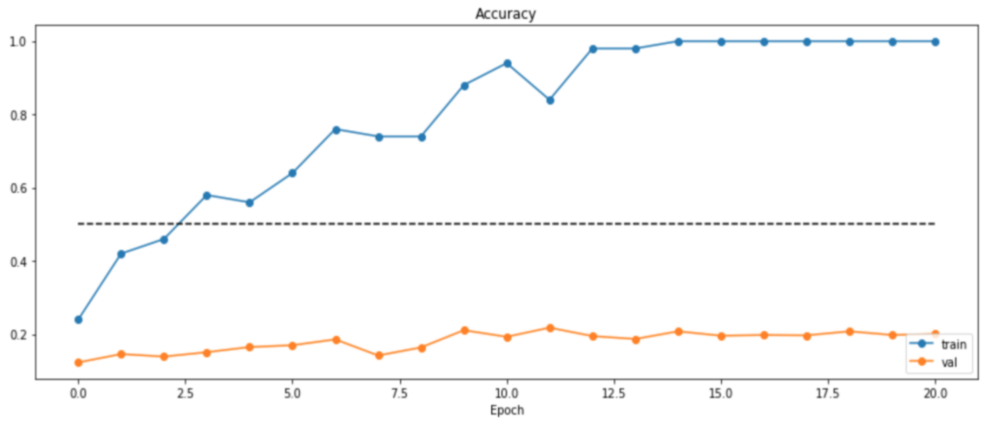
```
Running check with reg =  0
Initial loss:  2.30560881632
theta1 relative error: 1.85e-07
theta1_0 relative error: 8.76e-09
theta2 relative error: 4.79e-07
theta2_0 relative error: 2.59e-09
theta3 relative error: 1.05e-06
theta3_0 relative error: 1.61e-10
Running check with reg =  3.14
Initial loss:  5.9601191892
theta1 relative error: 5.95e-09
theta1_0 relative error: 8.40e-09
theta2 relative error: 1.25e-07
theta2_0 relative error: 6.48e-09
theta3 relative error: 1.00e+00
theta3_0 relative error: 2.11e-10
```

Problem 4.1.8: Overfitting a three layer network

`weight_scale = 1e-2, learning_rate = 1e-2`

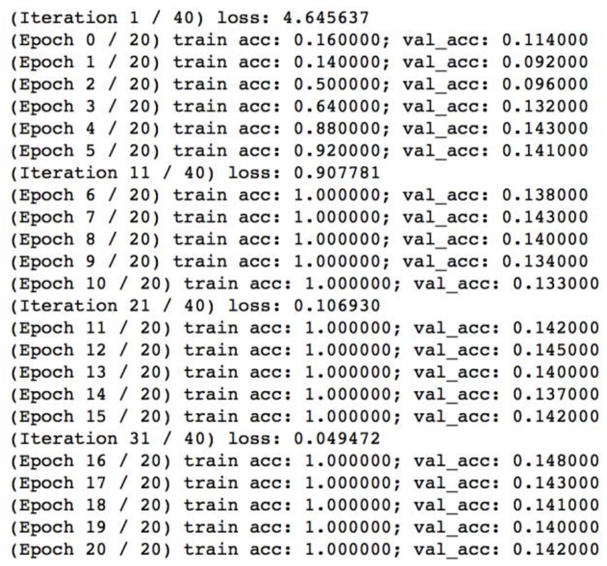
```
(Iteration 1 / 40) loss: 2.285701
(Epoch 0 / 20) train acc: 0.240000; val_acc: 0.123000
(Epoch 1 / 20) train acc: 0.420000; val_acc: 0.146000
(Epoch 2 / 20) train acc: 0.460000; val_acc: 0.139000
(Epoch 3 / 20) train acc: 0.580000; val_acc: 0.151000
(Epoch 4 / 20) train acc: 0.560000; val_acc: 0.165000
(Epoch 5 / 20) train acc: 0.640000; val_acc: 0.170000
(Iteration 11 / 40) loss: 1.319077
(Epoch 6 / 20) train acc: 0.760000; val_acc: 0.186000
(Epoch 7 / 20) train acc: 0.740000; val_acc: 0.142000
(Epoch 8 / 20) train acc: 0.740000; val_acc: 0.164000
(Epoch 9 / 20) train acc: 0.880000; val_acc: 0.211000
(Epoch 10 / 20) train acc: 0.940000; val_acc: 0.193000
(Iteration 21 / 40) loss: 0.280645
(Epoch 11 / 20) train acc: 0.840000; val_acc: 0.218000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.195000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.187000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.208000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.196000
(Iteration 31 / 40) loss: 0.067947
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.198000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.197000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.208000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.198000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.201000
```



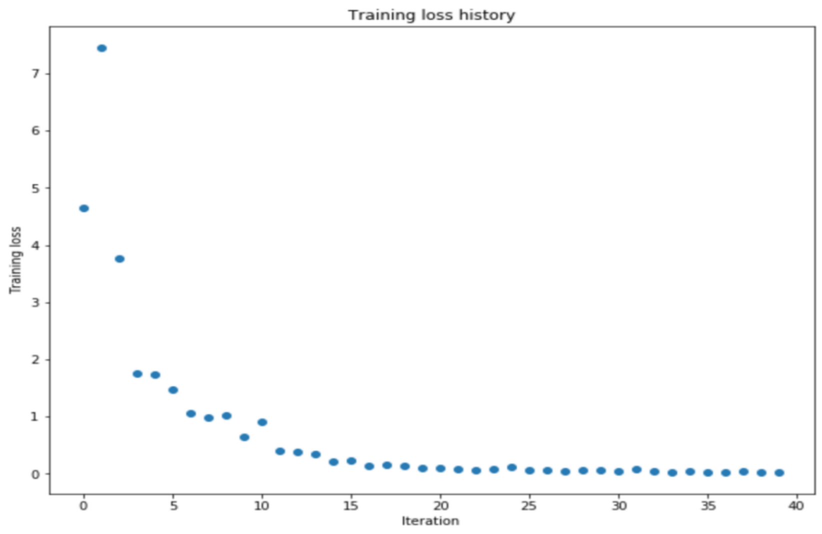


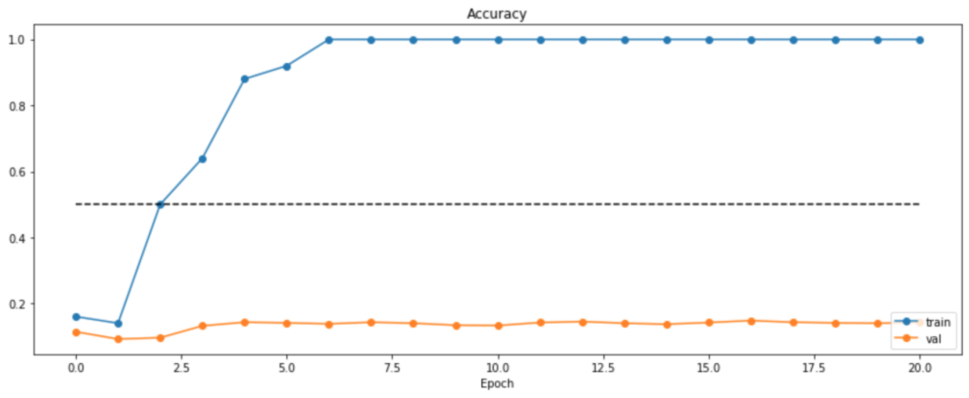
Problem 4.1.9: Overfitting a five layer network

`learning_rate = 1e-2, weight_scale = 5e-2`



```
(Iteration 1 / 40) loss: 4.645637
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.114000
(Epoch 1 / 20) train acc: 0.140000; val_acc: 0.092000
(Epoch 2 / 20) train acc: 0.500000; val_acc: 0.096000
(Epoch 3 / 20) train acc: 0.640000; val_acc: 0.132000
(Epoch 4 / 20) train acc: 0.880000; val_acc: 0.143000
(Epoch 5 / 20) train acc: 0.920000; val_acc: 0.141000
(Iteration 11 / 40) loss: 0.907781
(Epoch 6 / 20) train acc: 1.000000; val_acc: 0.138000
(Epoch 7 / 20) train acc: 1.000000; val_acc: 0.143000
(Epoch 8 / 20) train acc: 1.000000; val_acc: 0.140000
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.134000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.133000
(Iteration 21 / 40) loss: 0.106930
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.142000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.145000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.140000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.137000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.142000
(Iteration 31 / 40) loss: 0.049472
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.148000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.143000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.141000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.140000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.142000
```

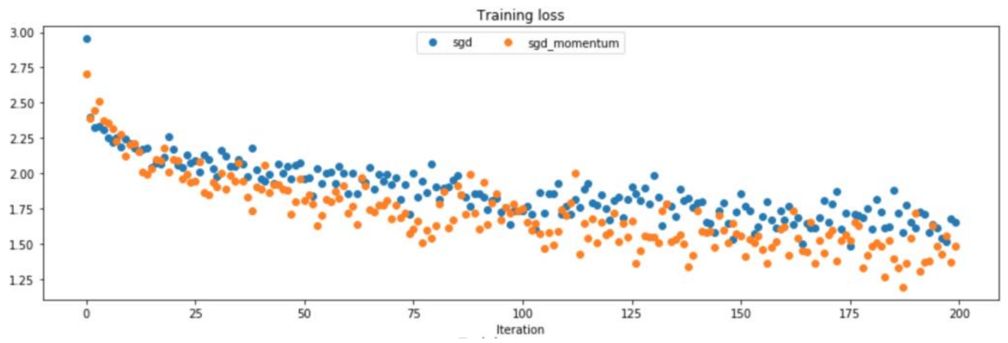


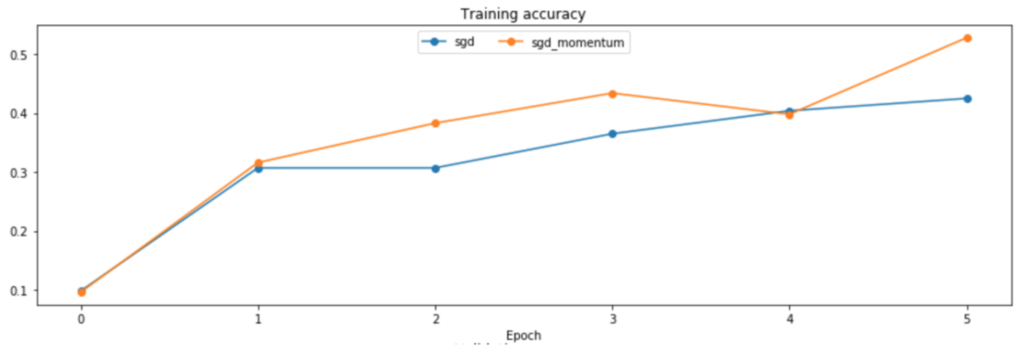


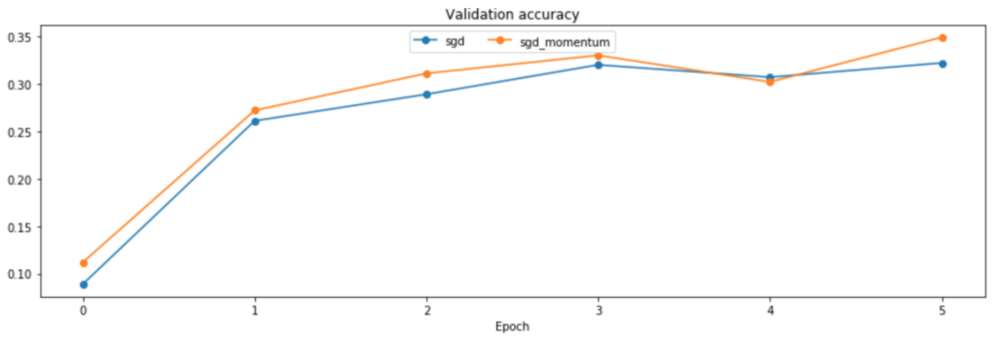
Overall, with the parameters that we have selected shown above, we did not notice that if one runs faster than the other.

Problem 4.1.10: SGD+Momentum

```
next_theta error:  8.88234703351e-09
velocity error:  4.26928774328e-09
```







Problem 4.1.11: RMSProp

```
next_theta error:  9.52468751104e-08
cache error:  2.64779558072e-09
```

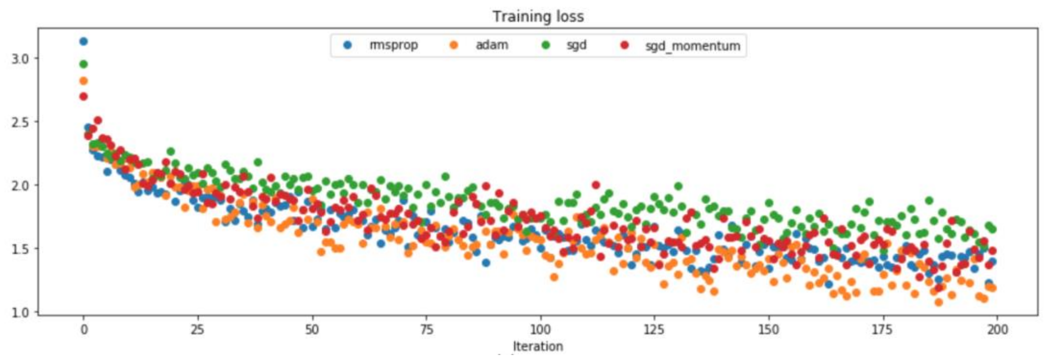
Problem 4.1.12: Adam

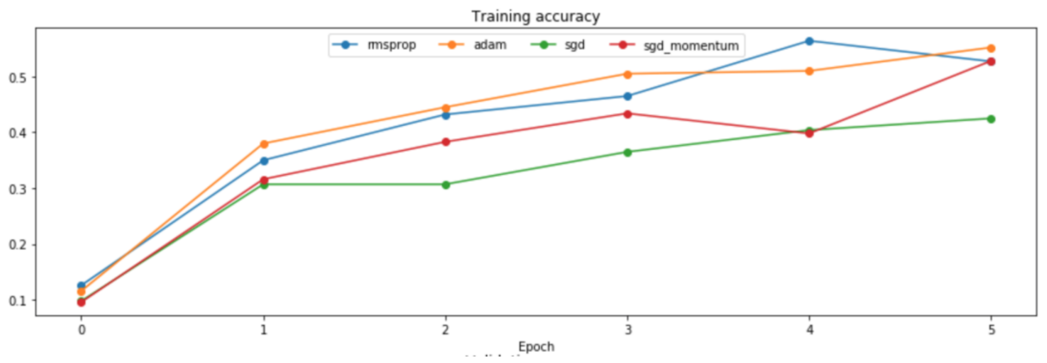
```
next_theta error:  1.13956917985e-07
v error:  4.20831403811e-09
m error:  4.21496319311e-09
```

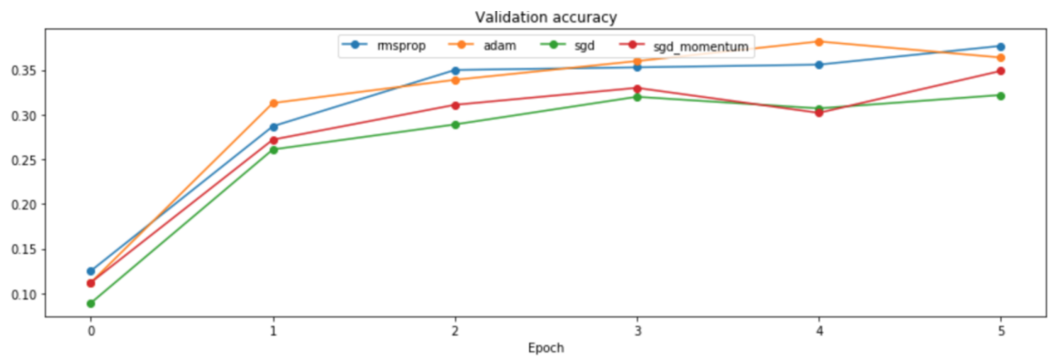
Problem 4.1.13: Comparison between different model

```
running with  adam
(Iteration 1 / 200) loss: 2.826789
(Epoch 0 / 5) train acc: 0.116000; val_acc: 0.112000
(Iteration 11 / 200) loss: 2.134827
(Iteration 21 / 200) loss: 1.983768
(Iteration 31 / 200) loss: 1.919874
(Epoch 1 / 5) train acc: 0.380000; val_acc: 0.313000
(Iteration 41 / 200) loss: 1.753019
(Iteration 51 / 200) loss: 1.885133
(Iteration 61 / 200) loss: 1.694751
(Iteration 71 / 200) loss: 1.714472
(Epoch 2 / 5) train acc: 0.445000; val_acc: 0.339000
(Iteration 81 / 200) loss: 1.596062
(Iteration 91 / 200) loss: 1.631651
(Iteration 101 / 200) loss: 1.568558
(Iteration 111 / 200) loss: 1.454471
(Epoch 3 / 5) train acc: 0.505000; val_acc: 0.360000
(Iteration 121 / 200) loss: 1.628023
(Iteration 131 / 200) loss: 1.298055
(Iteration 141 / 200) loss: 1.471206
(Iteration 151 / 200) loss: 1.418290
(Epoch 4 / 5) train acc: 0.510000; val_acc: 0.382000
(Iteration 161 / 200) loss: 1.309487
(Iteration 171 / 200) loss: 1.418249
(Iteration 181 / 200) loss: 1.209670
(Iteration 191 / 200) loss: 1.133905
(Epoch 5 / 5) train acc: 0.552000; val_acc: 0.364000
```

```
running with  rmsprop
(Iteration 1 / 200) loss: 3.133482
(Epoch 0 / 5) train acc: 0.126000; val_acc: 0.125000
(Iteration 11 / 200) loss: 2.062934
(Iteration 21 / 200) loss: 1.868280
(Iteration 31 / 200) loss: 1.876734
(Epoch 1 / 5) train acc: 0.350000; val_acc: 0.287000
(Iteration 41 / 200) loss: 1.818280
(Iteration 51 / 200) loss: 1.668685
(Iteration 61 / 200) loss: 1.717317
(Iteration 71 / 200) loss: 1.879672
(Epoch 2 / 5) train acc: 0.432000; val_acc: 0.350000
(Iteration 81 / 200) loss: 1.550333
(Iteration 91 / 200) loss: 1.589690
(Iteration 101 / 200) loss: 1.506552
(Iteration 111 / 200) loss: 1.564783
(Epoch 3 / 5) train acc: 0.465000; val_acc: 0.353000
(Iteration 121 / 200) loss: 1.374621
(Iteration 131 / 200) loss: 1.640120
(Iteration 141 / 200) loss: 1.490481
(Iteration 151 / 200) loss: 1.339503
(Epoch 4 / 5) train acc: 0.564000; val_acc: 0.356000
(Iteration 161 / 200) loss: 1.497039
(Iteration 171 / 200) loss: 1.387342
(Iteration 181 / 200) loss: 1.520318
(Iteration 191 / 200) loss: 1.449462
(Epoch 5 / 5) train acc: 0.527000; val_acc: 0.377000
```







Problem 4.2.1: Dropout forward pass

```
Running tests with p =  0.3
Mean of input:  9.99403005524
Mean of train-time output:  10.0021284944
Mean of test-time output:  9.99403005524
Fraction of train-time output set to zero:  0.299484
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.6
Mean of input:  9.99403005524
Mean of train-time output:  9.97760023661
Mean of test-time output:  9.99403005524
Fraction of train-time output set to zero:  0.600632
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.75
Mean of input:  9.99403005524
Mean of train-time output:  10.0093968697
Mean of test-time output:  9.99403005524
Fraction of train-time output set to zero:  0.749672
Fraction of test-time output set to zero:  0.0
```

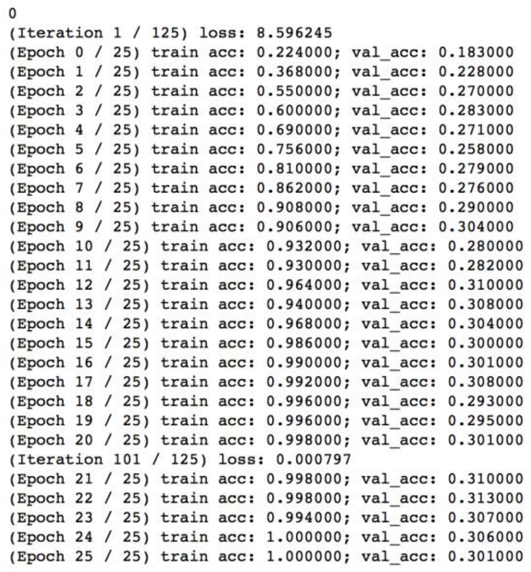
Problem 4.2.2: Dropout backward pass

```
dx relative error:  1.8929064223e-11
```

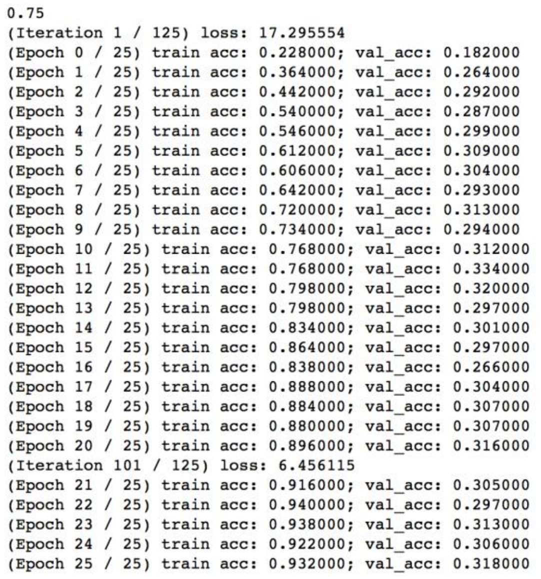
Problem 4.2.3: Fully connected nets with dropout

```
Running check with dropout =  0
Initial loss:  2.3051948274
theta1 relative error: 2.53e-07
theta1_0 relative error: 2.94e-06
theta2 relative error: 1.50e-05
theta2_0 relative error: 5.05e-08
theta3 relative error: 2.75e-07
theta3_0 relative error: 1.17e-10

Running check with dropout =  0.25
Initial loss:  2.31086540779
theta1 relative error: 3.40e-07
theta1_0 relative error: 6.89e-08
theta2 relative error: 2.91e-07
theta2_0 relative error: 3.77e-09
theta3 relative error: 1.77e-07
theta3_0 relative error: 1.68e-10

Running check with dropout =  0.5
Initial loss:  2.30243758771
theta1 relative error: 4.55e-08
theta1_0 relative error: 1.87e-08
theta2 relative error: 2.97e-08
theta2_0 relative error: 5.05e-09
theta3 relative error: 4.34e-07
theta3_0 relative error: 7.49e-11
```

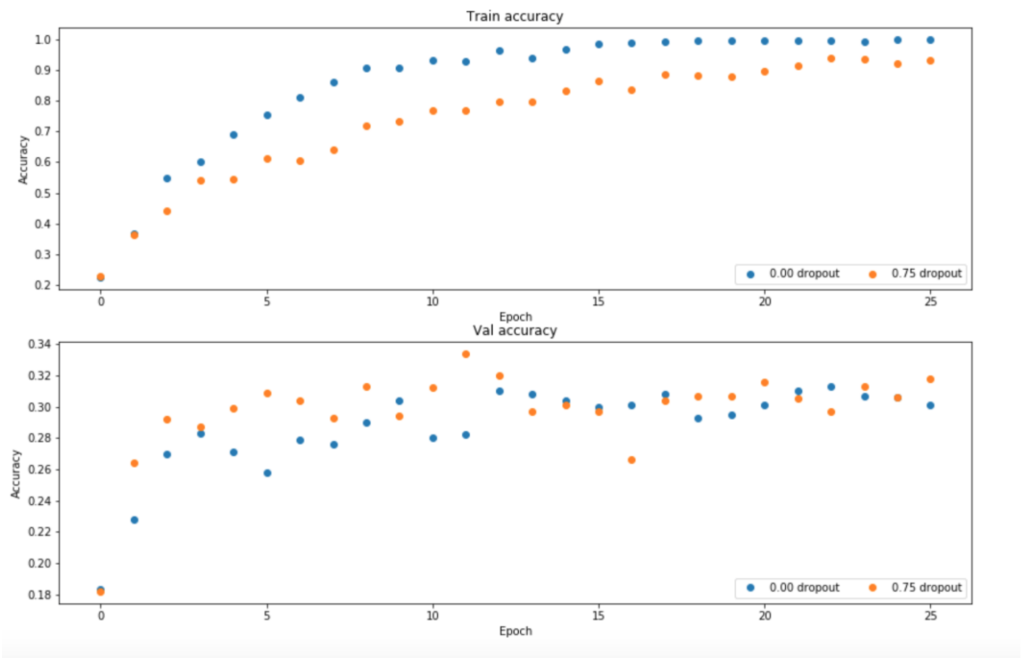
Problem 4.2.4: Experimenting with fully connected nets with dropout



```
0
(Iteration 1 / 125) loss: 8.596245
(Epoch 0 / 25) train acc: 0.224000; val_acc: 0.183000
(Epoch 1 / 25) train acc: 0.368000; val_acc: 0.228000
(Epoch 2 / 25) train acc: 0.550000; val_acc: 0.270000
(Epoch 3 / 25) train acc: 0.600000; val_acc: 0.283000
(Epoch 4 / 25) train acc: 0.690000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.756000; val_acc: 0.258000
(Epoch 6 / 25) train acc: 0.810000; val_acc: 0.279000
(Epoch 7 / 25) train acc: 0.862000; val_acc: 0.276000
(Epoch 8 / 25) train acc: 0.908000; val_acc: 0.290000
(Epoch 9 / 25) train acc: 0.906000; val_acc: 0.304000
(Epoch 10 / 25) train acc: 0.932000; val_acc: 0.280000
(Epoch 11 / 25) train acc: 0.930000; val_acc: 0.282000
(Epoch 12 / 25) train acc: 0.964000; val_acc: 0.310000
(Epoch 13 / 25) train acc: 0.940000; val_acc: 0.308000
(Epoch 14 / 25) train acc: 0.968000; val_acc: 0.304000
(Epoch 15 / 25) train acc: 0.986000; val_acc: 0.300000
(Epoch 16 / 25) train acc: 0.990000; val_acc: 0.301000
(Epoch 17 / 25) train acc: 0.992000; val_acc: 0.308000
(Epoch 18 / 25) train acc: 0.996000; val_acc: 0.293000
(Epoch 19 / 25) train acc: 0.996000; val_acc: 0.295000
(Epoch 20 / 25) train acc: 0.998000; val_acc: 0.301000
(Iteration 101 / 125) loss: 0.000797
(Epoch 21 / 25) train acc: 0.998000; val_acc: 0.310000
(Epoch 22 / 25) train acc: 0.998000; val_acc: 0.313000
(Epoch 23 / 25) train acc: 0.994000; val_acc: 0.307000
(Epoch 24 / 25) train acc: 1.000000; val_acc: 0.306000
(Epoch 25 / 25) train acc: 1.000000; val_acc: 0.301000
```



```
0.75
(Iteration 1 / 125) loss: 17.295554
(Epoch 0 / 25) train acc: 0.228000; val_acc: 0.182000
(Epoch 1 / 25) train acc: 0.364000; val_acc: 0.264000
(Epoch 2 / 25) train acc: 0.442000; val_acc: 0.292000
(Epoch 3 / 25) train acc: 0.540000; val_acc: 0.287000
(Epoch 4 / 25) train acc: 0.546000; val_acc: 0.299000
(Epoch 5 / 25) train acc: 0.612000; val_acc: 0.309000
(Epoch 6 / 25) train acc: 0.606000; val_acc: 0.304000
(Epoch 7 / 25) train acc: 0.642000; val_acc: 0.293000
(Epoch 8 / 25) train acc: 0.720000; val_acc: 0.313000
(Epoch 9 / 25) train acc: 0.734000; val_acc: 0.294000
(Epoch 10 / 25) train acc: 0.768000; val_acc: 0.312000
(Epoch 11 / 25) train acc: 0.768000; val_acc: 0.334000
(Epoch 12 / 25) train acc: 0.798000; val_acc: 0.320000
(Epoch 13 / 25) train acc: 0.798000; val_acc: 0.297000
(Epoch 14 / 25) train acc: 0.834000; val_acc: 0.301000
(Epoch 15 / 25) train acc: 0.864000; val_acc: 0.297000
(Epoch 16 / 25) train acc: 0.838000; val_acc: 0.266000
(Epoch 17 / 25) train acc: 0.888000; val_acc: 0.304000
(Epoch 18 / 25) train acc: 0.884000; val_acc: 0.307000
(Epoch 19 / 25) train acc: 0.880000; val_acc: 0.307000
(Epoch 20 / 25) train acc: 0.896000; val_acc: 0.316000
(Iteration 101 / 125) loss: 6.456115
(Epoch 21 / 25) train acc: 0.916000; val_acc: 0.305000
(Epoch 22 / 25) train acc: 0.940000; val_acc: 0.297000
(Epoch 23 / 25) train acc: 0.938000; val_acc: 0.313000
(Epoch 24 / 25) train acc: 0.922000; val_acc: 0.306000
(Epoch 25 / 25) train acc: 0.932000; val_acc: 0.318000
```



From the graph above, we see that the network with dropout converge much faster. And we also see that the shapes of curves for validation and training with dropout seem to be less volatile (less variance) than the ones without dropout. The model without dropout tends to over-fit the dataset, thus performs better on training set. Therefore, the graph shows that the model without dropout has better training accuracy, but its validation accuracy is lower than the model with dropout. However, the network with dropout is a better fit for the real model. Thus, it has a higher validation accuracy.

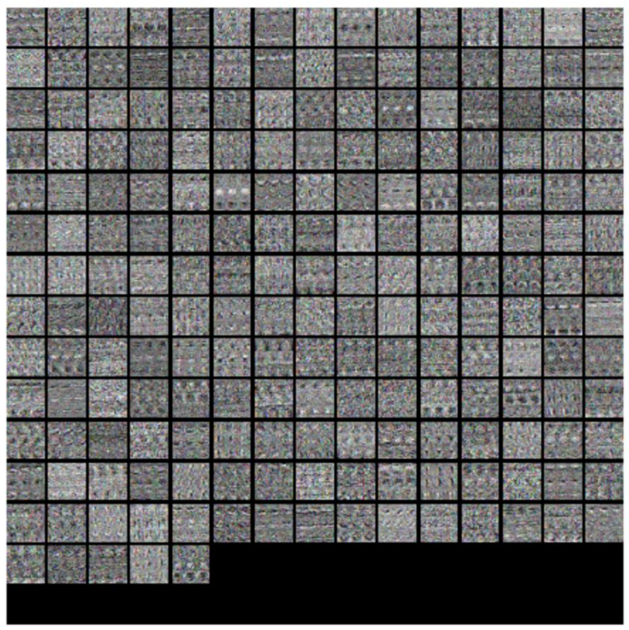
Problem 4.3: Training a fully connected network for the CIFAR-10 dataset with dropout
With parameters selected as below,

```
learning_rate = 1e-3
weight_scale = 1e-2
```

```
solver = Solver(model, training_data,
                print_every=100, num_epochs=20, batch_size=512,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                })
```

We get

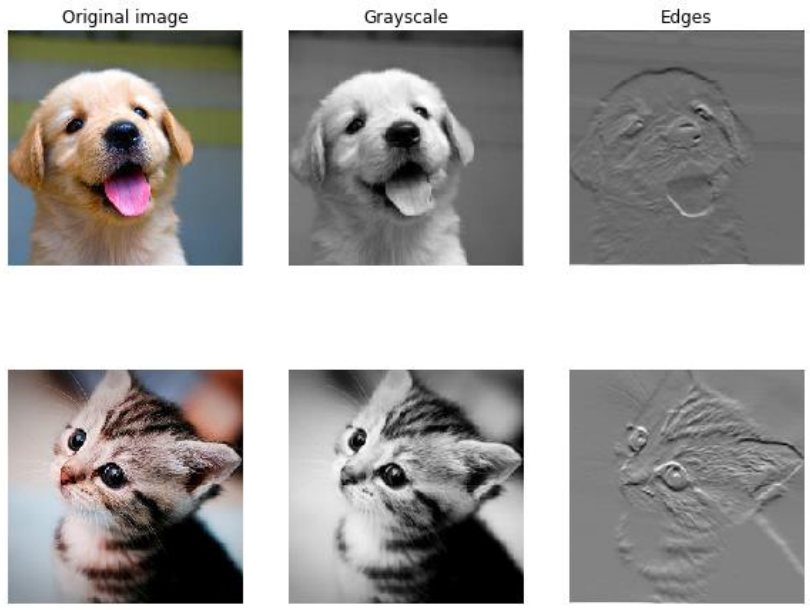
```
Validation set accuracy:  0.53
Test set accuracy:  0.529
```



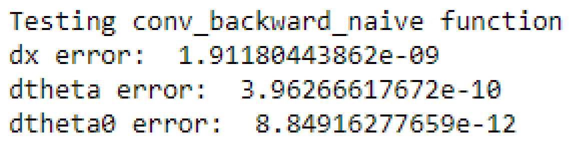
Problem 4.4.1: Convolution: naive forward pass

```
Testing conv_forward_naive
difference:  2.21214764175e-08
```

Aside: Image processing via convolutions

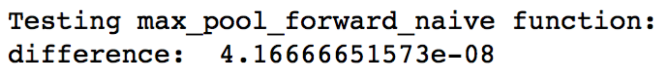


Problem 4.4.2: Convolution: naive backward pass



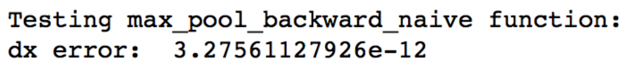
```
Testing conv_backward_naive function
dx error:  1.91180443862e-09
dtheta error:  3.96266617672e-10
dtheta0 error:  8.84916277659e-12
```

Problem 4.4.3: Max pooling: naive forward pass



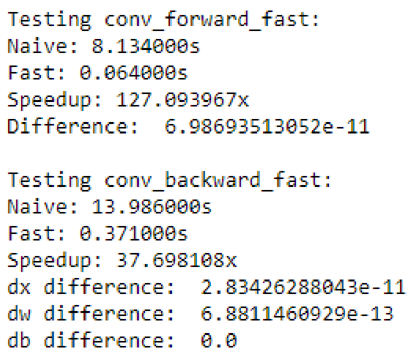
```
Testing max_pool_forward_naive function:
difference:  4.16666651573e-08
```

Problem 4.4.4: Max pooling: naive backward pass

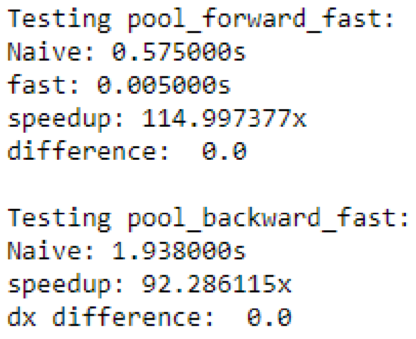


```
Testing max_pool_backward_naive function:
dx error:  3.27561127926e-12
```

Fast layers

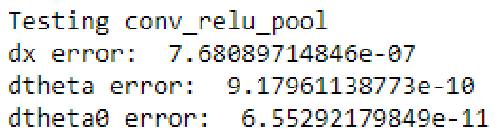


```
Testing conv_forward_fast:
Naive: 8.134000s
Fast: 0.064000s
Speedup: 127.093967x
Difference:  6.98693513052e-11

Testing conv_backward_fast:
Naive: 13.986000s
Fast: 0.371000s
Speedup: 37.698108x
dx difference:  2.83426288043e-11
dw difference:  6.8811460929e-13
db difference:  0.0
```

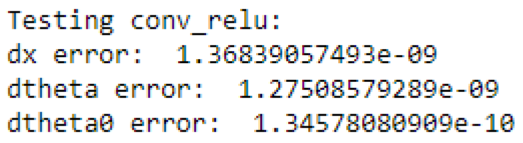


```
Testing pool_forward_fast:
Naive: 0.575000s
fast: 0.005000s
speedup: 114.997377x
difference:  0.0

Testing pool_backward_fast:
Naive: 1.938000s
speedup: 92.286115x
dx difference:  0.0
```

Convolutional sandwich layers



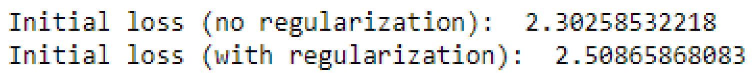
```
Testing conv_relu_pool
dx error:  7.68089714846e-07
dtheta error:  9.17961138773e-10
dtheta0 error:  6.55292179849e-11
```



```
Testing conv_relu:
dx error:  1.36839057493e-09
dtheta error:  1.27508579289e-09
dtheta0 error:  1.34578080909e-10
```

Problem 4.4.5: Three layer convolutional neural network

Loss computation:



```
Initial loss (no regularization):  2.30258532218
Initial loss (with regularization):  2.50865868083
```

Gradient Check:

```
theta1 max relative error: 1.712796e-04
theta1_0 max relative error: 3.801874e-05
theta2 max relative error: 2.740106e-03
theta2_0 max relative error: 1.399381e-07
theta3 max relative error: 1.465232e-04
theta3_0 max relative error: 1.568466e-09
```

Overfit small data:

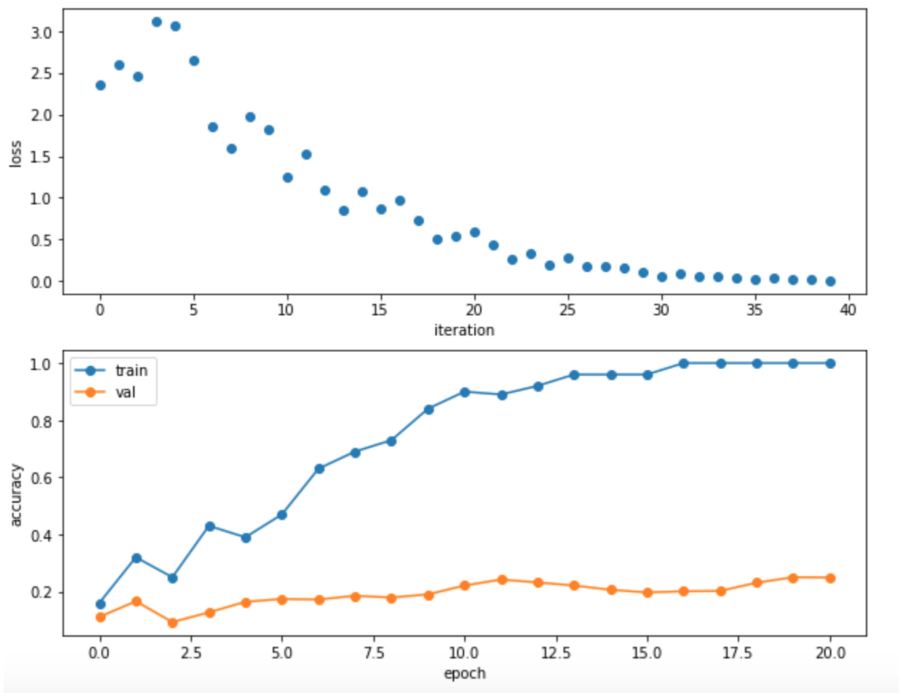
With the following being the parameters selected,

```
solver = Solver(model, small_data,
                num_epochs=20, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
```

We get the overfitting model

```
(Iteration 1 / 40) loss: 2.472078
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.119000
(Iteration 2 / 40) loss: 3.779697
(Epoch 1 / 20) train acc: 0.250000; val_acc: 0.145000
(Iteration 3 / 40) loss: 2.161449
(Iteration 4 / 40) loss: 2.501547
(Epoch 2 / 20) train acc: 0.210000; val_acc: 0.112000
(Iteration 5 / 40) loss: 2.517067
(Iteration 6 / 40) loss: 2.201610
(Epoch 3 / 20) train acc: 0.450000; val_acc: 0.182000
(Iteration 7 / 40) loss: 1.872595
(Iteration 8 / 40) loss: 1.803886
(Epoch 4 / 20) train acc: 0.430000; val_acc: 0.167000
(Iteration 9 / 40) loss: 1.607517
(Iteration 10 / 40) loss: 1.740199
(Epoch 5 / 20) train acc: 0.560000; val_acc: 0.157000
(Iteration 11 / 40) loss: 1.576530
(Iteration 12 / 40) loss: 1.332522
(Epoch 6 / 20) train acc: 0.630000; val_acc: 0.183000
(Iteration 13 / 40) loss: 1.281972
(Iteration 14 / 40) loss: 1.036954
(Epoch 7 / 20) train acc: 0.760000; val_acc: 0.213000
(Iteration 15 / 40) loss: 0.880180
(Iteration 16 / 40) loss: 0.990210
(Epoch 8 / 20) train acc: 0.790000; val_acc: 0.205000
(Iteration 17 / 40) loss: 0.644071
(Iteration 18 / 40) loss: 0.779442
(Epoch 9 / 20) train acc: 0.840000; val_acc: 0.204000
(Iteration 19 / 40) loss: 0.558558
(Iteration 20 / 40) loss: 0.510775
(Epoch 10 / 20) train acc: 0.900000; val_acc: 0.215000
```

```
(Iteration 21 / 40) loss: 0.410603
(Iteration 22 / 40) loss: 0.485505
(Epoch 11 / 20) train acc: 0.920000; val_acc: 0.219000
(Iteration 23 / 40) loss: 0.286831
(Iteration 24 / 40) loss: 0.166471
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.207000
(Iteration 25 / 40) loss: 0.248834
(Iteration 26 / 40) loss: 0.091540
(Epoch 13 / 20) train acc: 0.930000; val_acc: 0.209000
(Iteration 27 / 40) loss: 0.242292
(Iteration 28 / 40) loss: 0.107951
(Epoch 14 / 20) train acc: 0.970000; val_acc: 0.201000
(Iteration 29 / 40) loss: 0.167010
(Iteration 30 / 40) loss: 0.092082
(Epoch 15 / 20) train acc: 0.990000; val_acc: 0.198000
(Iteration 31 / 40) loss: 0.032871
(Iteration 32 / 40) loss: 0.043531
(Epoch 16 / 20) train acc: 0.990000; val_acc: 0.211000
(Iteration 33 / 40) loss: 0.065602
(Iteration 34 / 40) loss: 0.052366
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.210000
(Iteration 35 / 40) loss: 0.030620
(Iteration 36 / 40) loss: 0.016364
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.206000
(Iteration 37 / 40) loss: 0.025416
(Iteration 38 / 40) loss: 0.017261
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.210000
(Iteration 39 / 40) loss: 0.014098
(Iteration 40 / 40) loss: 0.016695
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.217000
```

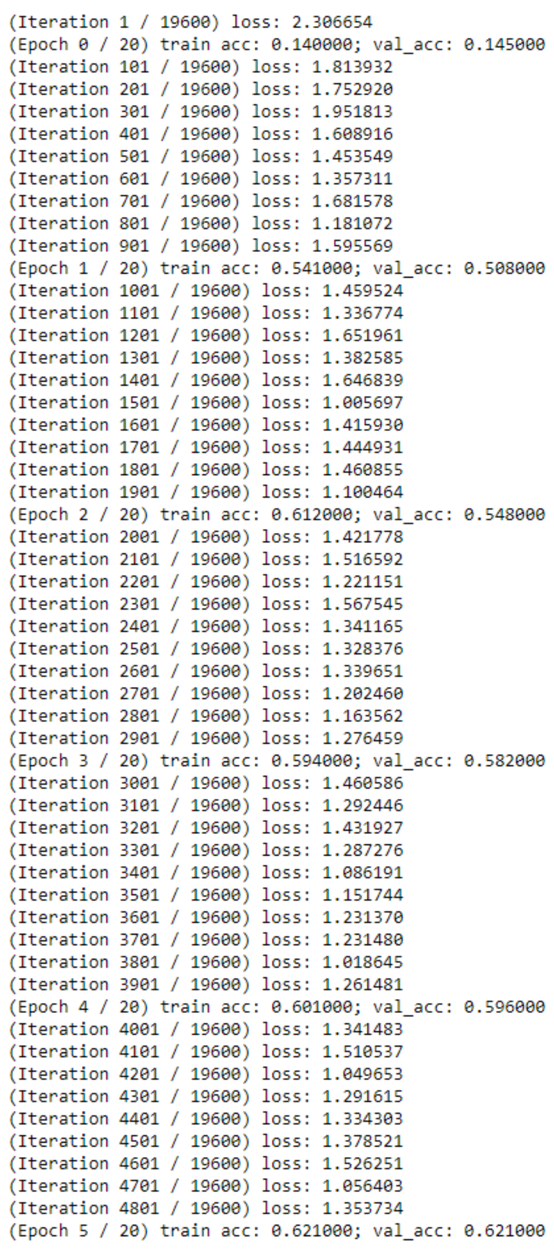


Problem 4.4.6: Train the CNN on the CIFAR-10 data

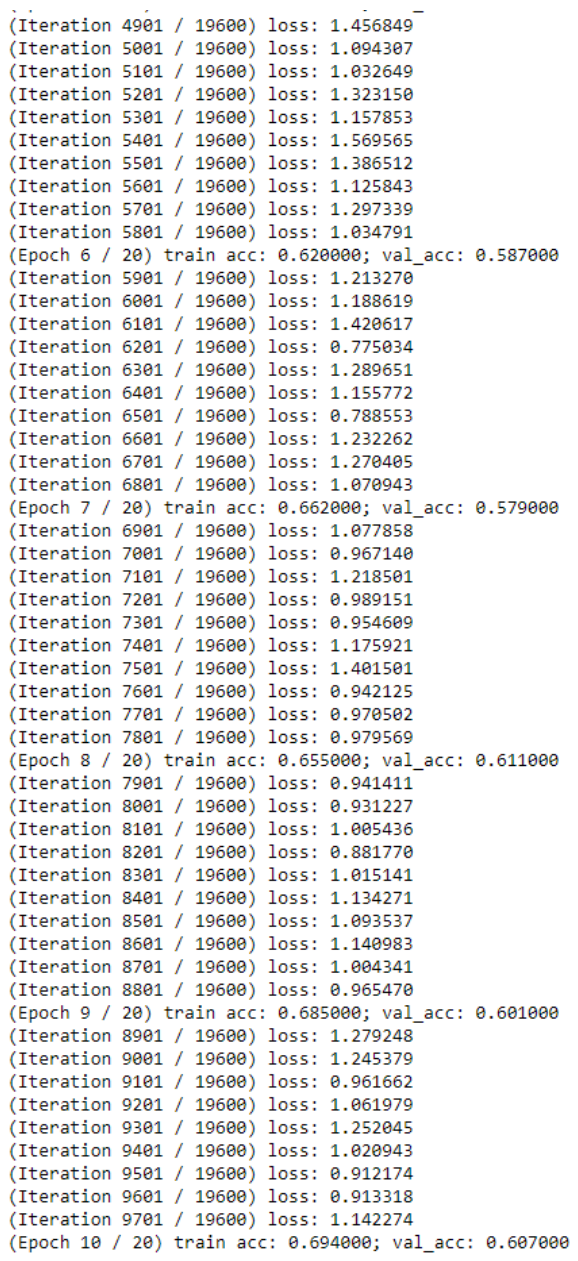
With the following being the parameters selected,

```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001, num_filters=64, filter_size=3
)

solver = Solver(model, data,
                num_epochs=20, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=100)
solver.train()
```

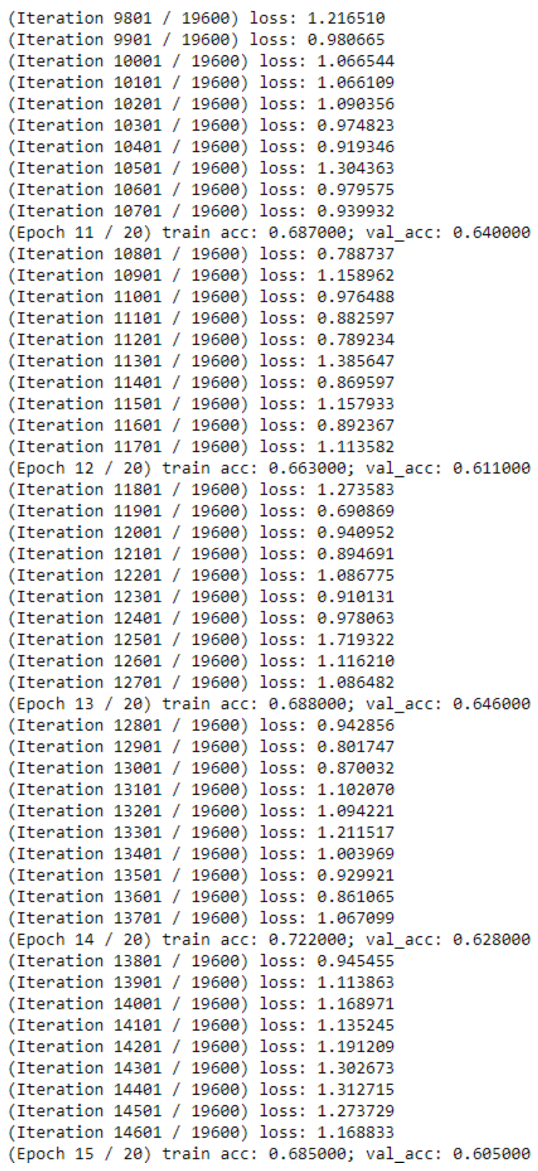
We get the result of our model:



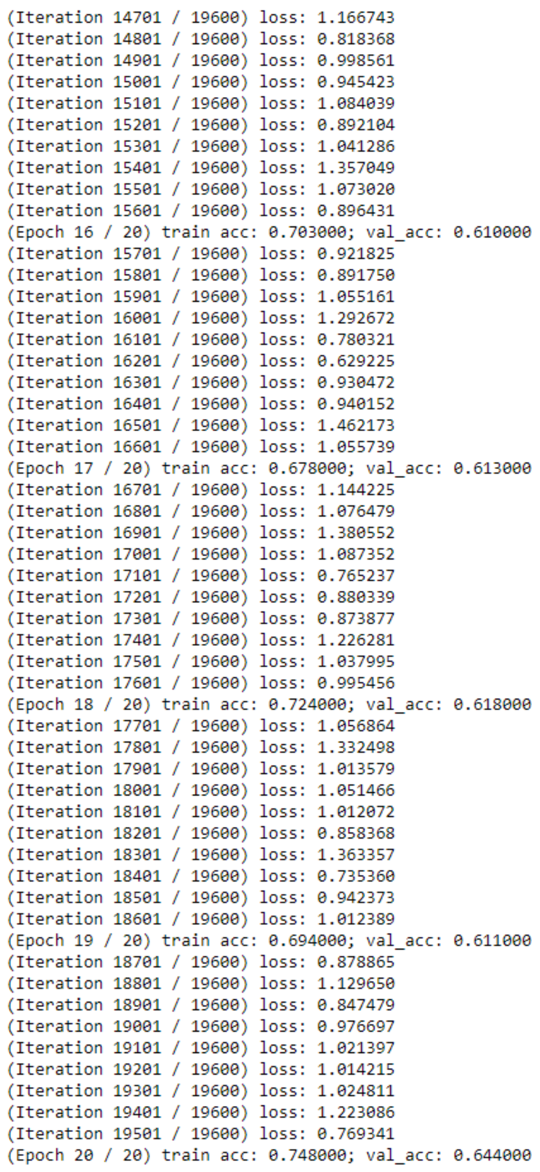
```
(Iteration 1 / 19600) loss: 2.306654
(Epoch 0 / 20) train acc: 0.140000; val_acc: 0.145000
(Iteration 101 / 19600) loss: 1.813932
(Iteration 201 / 19600) loss: 1.752920
(Iteration 301 / 19600) loss: 1.951813
(Iteration 401 / 19600) loss: 1.608916
(Iteration 501 / 19600) loss: 1.453549
(Iteration 601 / 19600) loss: 1.357311
(Iteration 701 / 19600) loss: 1.681578
(Iteration 801 / 19600) loss: 1.181072
(Iteration 901 / 19600) loss: 1.595569
(Epoch 1 / 20) train acc: 0.541000; val_acc: 0.508000
(Iteration 1001 / 19600) loss: 1.459524
(Iteration 1101 / 19600) loss: 1.336774
(Iteration 1201 / 19600) loss: 1.651961
(Iteration 1301 / 19600) loss: 1.382585
(Iteration 1401 / 19600) loss: 1.646839
(Iteration 1501 / 19600) loss: 1.005697
(Iteration 1601 / 19600) loss: 1.415930
(Iteration 1701 / 19600) loss: 1.444931
(Iteration 1801 / 19600) loss: 1.460855
(Iteration 1901 / 19600) loss: 1.100464
(Epoch 2 / 20) train acc: 0.612000; val_acc: 0.548000
(Iteration 2001 / 19600) loss: 1.421778
(Iteration 2101 / 19600) loss: 1.516592
(Iteration 2201 / 19600) loss: 1.221151
(Iteration 2301 / 19600) loss: 1.567545
(Iteration 2401 / 19600) loss: 1.341165
(Iteration 2501 / 19600) loss: 1.328376
(Iteration 2601 / 19600) loss: 1.339651
(Iteration 2701 / 19600) loss: 1.202460
(Iteration 2801 / 19600) loss: 1.163562
(Iteration 2901 / 19600) loss: 1.276459
(Epoch 3 / 20) train acc: 0.594000; val_acc: 0.582000
(Iteration 3001 / 19600) loss: 1.460586
(Iteration 3101 / 19600) loss: 1.292446
(Iteration 3201 / 19600) loss: 1.431927
(Iteration 3301 / 19600) loss: 1.287276
(Iteration 3401 / 19600) loss: 1.086191
(Iteration 3501 / 19600) loss: 1.151744
(Iteration 3601 / 19600) loss: 1.231370
(Iteration 3701 / 19600) loss: 1.231480
(Iteration 3801 / 19600) loss: 1.018645
(Iteration 3901 / 19600) loss: 1.261481
(Epoch 4 / 20) train acc: 0.601000; val_acc: 0.596000
(Iteration 4001 / 19600) loss: 1.341483
(Iteration 4101 / 19600) loss: 1.510537
(Iteration 4201 / 19600) loss: 1.049653
(Iteration 4301 / 19600) loss: 1.291615
(Iteration 4401 / 19600) loss: 1.334303
(Iteration 4501 / 19600) loss: 1.378521
(Iteration 4601 / 19600) loss: 1.526251
(Iteration 4701 / 19600) loss: 1.056403
(Iteration 4801 / 19600) loss: 1.353734
(Epoch 5 / 20) train acc: 0.621000; val_acc: 0.621000
```



```
(Iteration 4901 / 19600) loss: 1.456849
(Iteration 5001 / 19600) loss: 1.094307
(Iteration 5101 / 19600) loss: 1.032649
(Iteration 5201 / 19600) loss: 1.323150
(Iteration 5301 / 19600) loss: 1.157853
(Iteration 5401 / 19600) loss: 1.569565
(Iteration 5501 / 19600) loss: 1.386512
(Iteration 5601 / 19600) loss: 1.125843
(Iteration 5701 / 19600) loss: 1.297339
(Iteration 5801 / 19600) loss: 1.034791
(Epoch 6 / 20) train acc: 0.620000; val_acc: 0.587000
(Iteration 5901 / 19600) loss: 1.213270
(Iteration 6001 / 19600) loss: 1.188619
(Iteration 6101 / 19600) loss: 1.420617
(Iteration 6201 / 19600) loss: 0.775034
(Iteration 6301 / 19600) loss: 1.289651
(Iteration 6401 / 19600) loss: 1.155772
(Iteration 6501 / 19600) loss: 0.788553
(Iteration 6601 / 19600) loss: 1.232262
(Iteration 6701 / 19600) loss: 1.270405
(Iteration 6801 / 19600) loss: 1.070943
(Epoch 7 / 20) train acc: 0.662000; val_acc: 0.579000
(Iteration 6901 / 19600) loss: 1.077858
(Iteration 7001 / 19600) loss: 0.967140
(Iteration 7101 / 19600) loss: 1.218501
(Iteration 7201 / 19600) loss: 0.989151
(Iteration 7301 / 19600) loss: 0.954609
(Iteration 7401 / 19600) loss: 1.175921
(Iteration 7501 / 19600) loss: 1.401501
(Iteration 7601 / 19600) loss: 0.942125
(Iteration 7701 / 19600) loss: 0.970502
(Iteration 7801 / 19600) loss: 0.979569
(Epoch 8 / 20) train acc: 0.655000; val_acc: 0.611000
(Iteration 7901 / 19600) loss: 0.941411
(Iteration 8001 / 19600) loss: 0.931227
(Iteration 8101 / 19600) loss: 1.005436
(Iteration 8201 / 19600) loss: 0.881770
(Iteration 8301 / 19600) loss: 1.015141
(Iteration 8401 / 19600) loss: 1.134271
(Iteration 8501 / 19600) loss: 1.093537
(Iteration 8601 / 19600) loss: 1.140983
(Iteration 8701 / 19600) loss: 1.004341
(Iteration 8801 / 19600) loss: 0.965470
(Epoch 9 / 20) train acc: 0.685000; val_acc: 0.601000
(Iteration 8901 / 19600) loss: 1.279248
(Iteration 9001 / 19600) loss: 1.245379
(Iteration 9101 / 19600) loss: 0.961662
(Iteration 9201 / 19600) loss: 1.061979
(Iteration 9301 / 19600) loss: 1.252045
(Iteration 9401 / 19600) loss: 1.020943
(Iteration 9501 / 19600) loss: 0.912174
(Iteration 9601 / 19600) loss: 0.913318
(Iteration 9701 / 19600) loss: 1.142274
(Epoch 10 / 20) train acc: 0.694000; val_acc: 0.607000
```



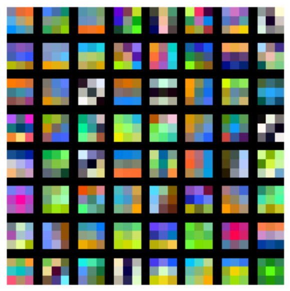
```
(Iteration 9801 / 19600) loss: 1.216510
(Iteration 9901 / 19600) loss: 0.980665
(Iteration 10001 / 19600) loss: 1.066544
(Iteration 10101 / 19600) loss: 1.066109
(Iteration 10201 / 19600) loss: 1.090356
(Iteration 10301 / 19600) loss: 0.974823
(Iteration 10401 / 19600) loss: 0.919346
(Iteration 10501 / 19600) loss: 1.304363
(Iteration 10601 / 19600) loss: 0.979575
(Iteration 10701 / 19600) loss: 0.939932
(Epoch 11 / 20) train acc: 0.687000; val_acc: 0.640000
(Iteration 10801 / 19600) loss: 0.788737
(Iteration 10901 / 19600) loss: 1.158962
(Iteration 11001 / 19600) loss: 0.976488
(Iteration 11101 / 19600) loss: 0.882597
(Iteration 11201 / 19600) loss: 0.789234
(Iteration 11301 / 19600) loss: 1.385647
(Iteration 11401 / 19600) loss: 0.869597
(Iteration 11501 / 19600) loss: 1.157933
(Iteration 11601 / 19600) loss: 0.892367
(Iteration 11701 / 19600) loss: 1.113582
(Epoch 12 / 20) train acc: 0.663000; val_acc: 0.611000
(Iteration 11801 / 19600) loss: 1.273583
(Iteration 11901 / 19600) loss: 0.690869
(Iteration 12001 / 19600) loss: 0.940952
(Iteration 12101 / 19600) loss: 0.894691
(Iteration 12201 / 19600) loss: 1.086775
(Iteration 12301 / 19600) loss: 0.910131
(Iteration 12401 / 19600) loss: 0.978063
(Iteration 12501 / 19600) loss: 1.719322
(Iteration 12601 / 19600) loss: 1.116210
(Iteration 12701 / 19600) loss: 1.086482
(Epoch 13 / 20) train acc: 0.688000; val_acc: 0.646000
(Iteration 12801 / 19600) loss: 0.942856
(Iteration 12901 / 19600) loss: 0.801747
(Iteration 13001 / 19600) loss: 0.870032
(Iteration 13101 / 19600) loss: 1.102070
(Iteration 13201 / 19600) loss: 1.094221
(Iteration 13301 / 19600) loss: 1.211517
(Iteration 13401 / 19600) loss: 1.003969
(Iteration 13501 / 19600) loss: 0.929921
(Iteration 13601 / 19600) loss: 0.861065
(Iteration 13701 / 19600) loss: 1.067099
(Epoch 14 / 20) train acc: 0.722000; val_acc: 0.628000
(Iteration 13801 / 19600) loss: 0.945455
(Iteration 13901 / 19600) loss: 1.113863
(Iteration 14001 / 19600) loss: 1.168971
(Iteration 14101 / 19600) loss: 1.135245
(Iteration 14201 / 19600) loss: 1.191209
(Iteration 14301 / 19600) loss: 1.302673
(Iteration 14401 / 19600) loss: 1.312715
(Iteration 14501 / 19600) loss: 1.273729
(Iteration 14601 / 19600) loss: 1.168833
(Epoch 15 / 20) train acc: 0.685000; val_acc: 0.605000
```



```
(Iteration 14701 / 19600) loss: 1.166743
(Iteration 14801 / 19600) loss: 0.818368
(Iteration 14901 / 19600) loss: 0.998561
(Iteration 15001 / 19600) loss: 0.945423
(Iteration 15101 / 19600) loss: 1.084039
(Iteration 15201 / 19600) loss: 0.892104
(Iteration 15301 / 19600) loss: 1.041286
(Iteration 15401 / 19600) loss: 1.357049
(Iteration 15501 / 19600) loss: 1.073020
(Iteration 15601 / 19600) loss: 0.896431
(Epoch 16 / 20) train acc: 0.703000; val_acc: 0.610000
(Iteration 15701 / 19600) loss: 0.921825
(Iteration 15801 / 19600) loss: 0.891750
(Iteration 15901 / 19600) loss: 1.055161
(Iteration 16001 / 19600) loss: 1.292672
(Iteration 16101 / 19600) loss: 0.780321
(Iteration 16201 / 19600) loss: 0.629225
(Iteration 16301 / 19600) loss: 0.930472
(Iteration 16401 / 19600) loss: 0.940152
(Iteration 16501 / 19600) loss: 1.462173
(Iteration 16601 / 19600) loss: 1.055739
(Epoch 17 / 20) train acc: 0.678000; val_acc: 0.613000
(Iteration 16701 / 19600) loss: 1.144225
(Iteration 16801 / 19600) loss: 1.076479
(Iteration 16901 / 19600) loss: 1.380552
(Iteration 17001 / 19600) loss: 1.087352
(Iteration 17101 / 19600) loss: 0.765237
(Iteration 17201 / 19600) loss: 0.880339
(Iteration 17301 / 19600) loss: 0.873877
(Iteration 17401 / 19600) loss: 1.226281
(Iteration 17501 / 19600) loss: 1.037995
(Iteration 17601 / 19600) loss: 0.995456
(Epoch 18 / 20) train acc: 0.724000; val_acc: 0.618000
(Iteration 17701 / 19600) loss: 1.056864
(Iteration 17801 / 19600) loss: 1.332498
(Iteration 17901 / 19600) loss: 1.013579
(Iteration 18001 / 19600) loss: 1.051466
(Iteration 18101 / 19600) loss: 1.012072
(Iteration 18201 / 19600) loss: 0.858368
(Iteration 18301 / 19600) loss: 1.363357
(Iteration 18401 / 19600) loss: 0.735360
(Iteration 18501 / 19600) loss: 0.942373
(Iteration 18601 / 19600) loss: 1.012389
(Epoch 19 / 20) train acc: 0.694000; val_acc: 0.611000
(Iteration 18701 / 19600) loss: 0.878865
(Iteration 18801 / 19600) loss: 1.129650
(Iteration 18901 / 19600) loss: 0.847479
(Iteration 19001 / 19600) loss: 0.976697
(Iteration 19101 / 19600) loss: 1.021397
(Iteration 19201 / 19600) loss: 1.014215
(Iteration 19301 / 19600) loss: 1.024811
(Iteration 19401 / 19600) loss: 1.223086
(Iteration 19501 / 19600) loss: 0.769341
(Epoch 20 / 20) train acc: 0.748000; val_acc: 0.644000
```

After several trials on the combination of the parameters, here we got a 0.644 validation accuracy on the CIFAR-10 dataset.

Visualize Filters

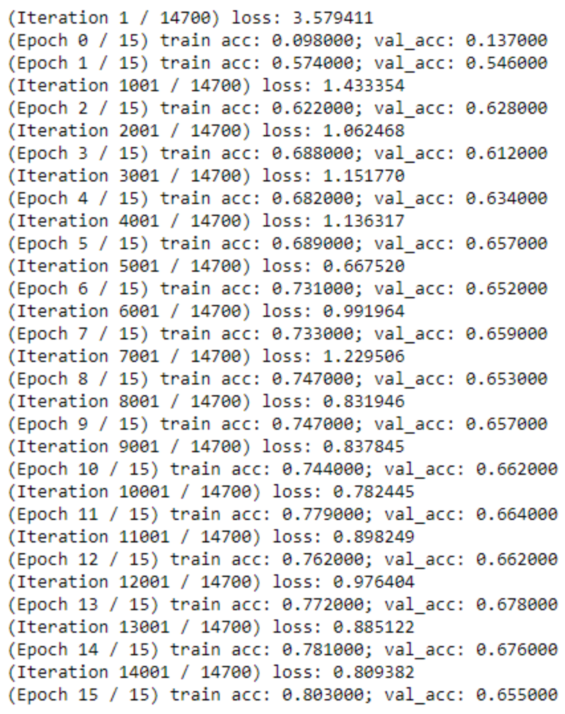


Extra credit:

Problem 4.4.7: Train the best model for CIFAR-10 dataset

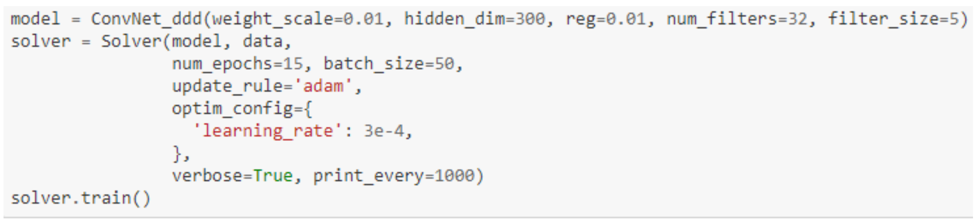
I implemented and tested an alternative network architecture as suggested: [conv-relu-pool] * N - conv - relu - [affine] * M - [softmax]. In my implementation, this model contains the following layers: conv-relu-pool, conv-relu, [affine] * 2-softmax.

The result I got is as below, the best validation accuracy we got is 0.678 using this model.



```
(Iteration 1 / 14700) loss: 3.579411
(Epoch 0 / 15) train acc: 0.098000; val_acc: 0.137000
(Epoch 1 / 15) train acc: 0.574000; val_acc: 0.546000
(Iteration 1001 / 14700) loss: 1.433354
(Epoch 2 / 15) train acc: 0.622000; val_acc: 0.628000
(Iteration 2001 / 14700) loss: 1.062468
(Epoch 3 / 15) train acc: 0.688000; val_acc: 0.612000
(Iteration 3001 / 14700) loss: 1.151770
(Epoch 4 / 15) train acc: 0.682000; val_acc: 0.634000
(Iteration 4001 / 14700) loss: 1.136317
(Epoch 5 / 15) train acc: 0.689000; val_acc: 0.657000
(Iteration 5001 / 14700) loss: 0.667520
(Epoch 6 / 15) train acc: 0.731000; val_acc: 0.652000
(Iteration 6001 / 14700) loss: 0.991964
(Epoch 7 / 15) train acc: 0.733000; val_acc: 0.659000
(Iteration 7001 / 14700) loss: 1.229506
(Epoch 8 / 15) train acc: 0.747000; val_acc: 0.653000
(Iteration 8001 / 14700) loss: 0.831946
(Epoch 9 / 15) train acc: 0.747000; val_acc: 0.657000
(Iteration 9001 / 14700) loss: 0.837845
(Epoch 10 / 15) train acc: 0.744000; val_acc: 0.662000
(Iteration 10001 / 14700) loss: 0.782445
(Epoch 11 / 15) train acc: 0.779000; val_acc: 0.664000
(Iteration 11001 / 14700) loss: 0.898249
(Epoch 12 / 15) train acc: 0.762000; val_acc: 0.662000
(Iteration 12001 / 14700) loss: 0.976404
(Epoch 13 / 15) train acc: 0.772000; val_acc: 0.678000
(Iteration 13001 / 14700) loss: 0.885122
(Epoch 14 / 15) train acc: 0.781000; val_acc: 0.676000
(Iteration 14001 / 14700) loss: 0.809382
(Epoch 15 / 15) train acc: 0.803000; val_acc: 0.655000
```

I also tried some combinations of the parameters, like decreasing the filter size, increasing the number of filters, changing the learning rate, and increasing the hidden dimension. The result shows that it can achieve better performance on our training and validation dataset. The parameters leading to the above result is



```
model = ConvNet_ddd(weight_scale=0.01, hidden_dim=300, reg=0.01, num_filters=32, filter_size=5)
solver = Solver(model, data,
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 3e-4,
                },
                verbose=True, print_every=1000)
solver.train()
```

Of course, we can add more [conv-relu-pool] layers or [affine] layers to this net to get a better performance, but it costs much longer training time.