STAT540-HW5

Yunyi Lin(yl146) & Pei Zeng(pz14)

1 Deep neural networks

1.1 Why do deep networks typically outperform shallow networks?

Deep networks with multiple hidden layers can learn more detailed and more abstractions relationships within the data and how the features interact with each other on a non-linear level. Adding more layers, allows for more easy representation of the interactions within the input data, as well as allows for more abstract features to be learned and used as input into the next hidden layer. So deep networks typically outperform shallow networks.

1.2 What is leaky ReLU activation and why is it used?

The concept of leaky ReLU is when x < 0, it will have a small positive slope of 0.1. Its function is:

$$f(x) = max(0.1x, x)$$

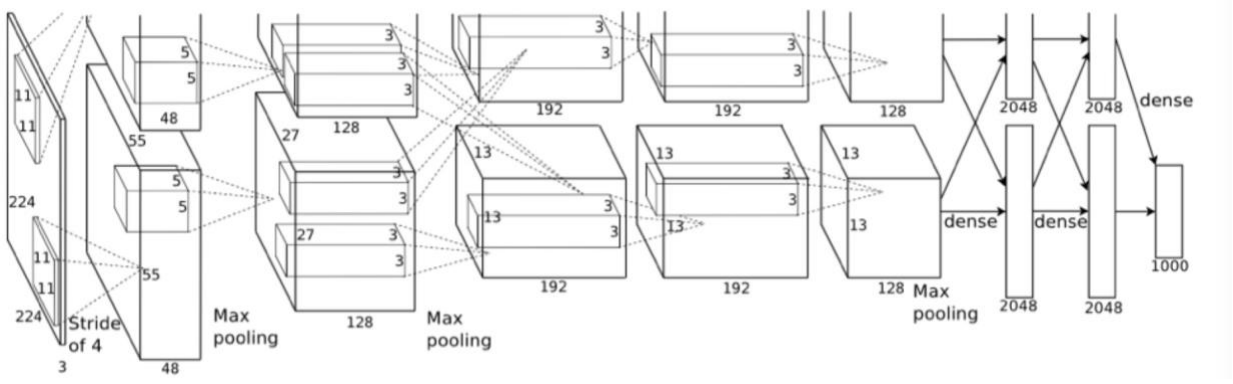
Leaky ReLU is used to describe the negetive part of x where the gradient vanishes in ReLU. In some degrees, it eliminates the dying ReLU problem.

1.3 In one or more sentences, and using sketches as appropriate, contrast: AlexNet, VGGNet, GoogleNet and ResNet. What is the one defining characteristic of each network?

AlexNet:

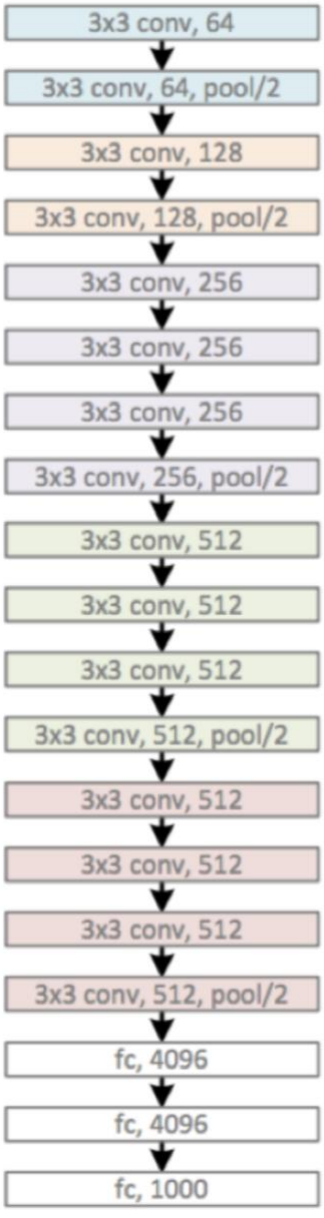
Compare to traditional CNN, AlexNet has some important differences:

(1)Data Augmentation. (2)Dropout. (3)ReLU Activation Function. (4)Local Response Normalization(LRN). (5)Overlapping Pooling. (6)Multi-GPU Parallel.



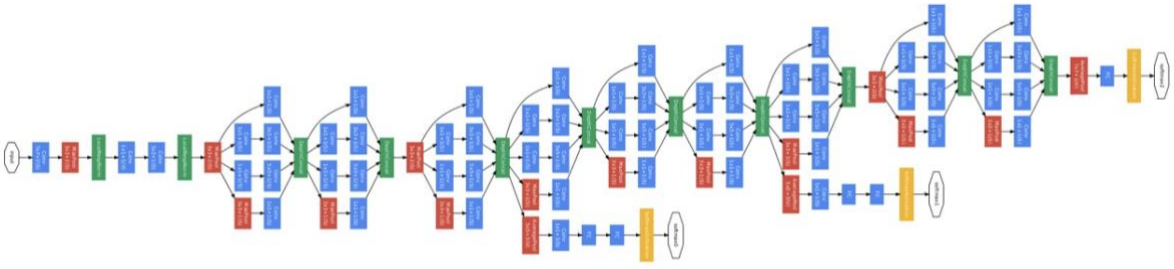
VGGNet:

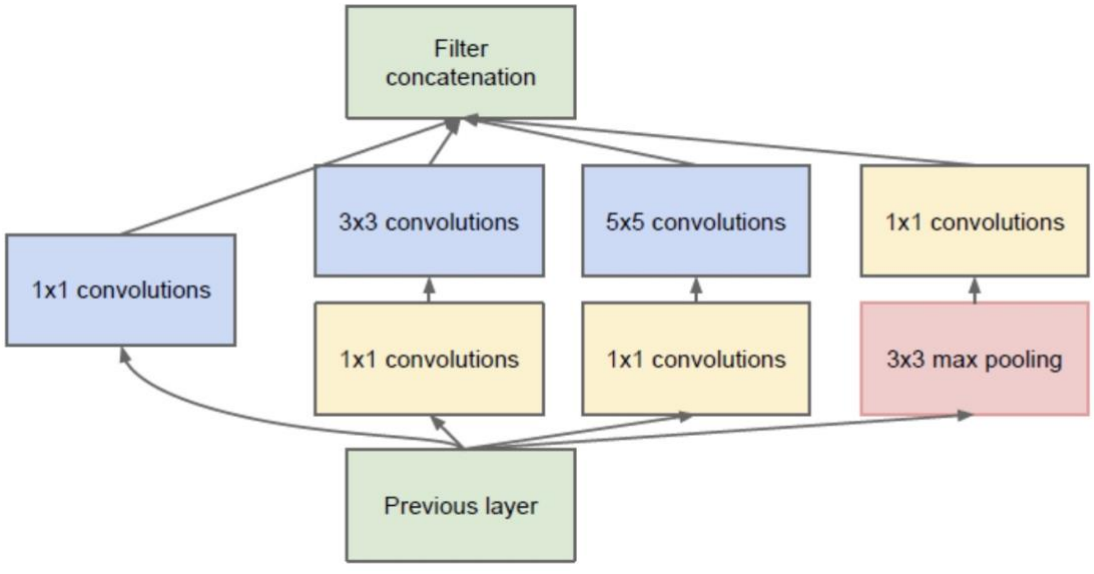
It has many continuous convolutions and need to deal with huge computation task.



GoogleNet:

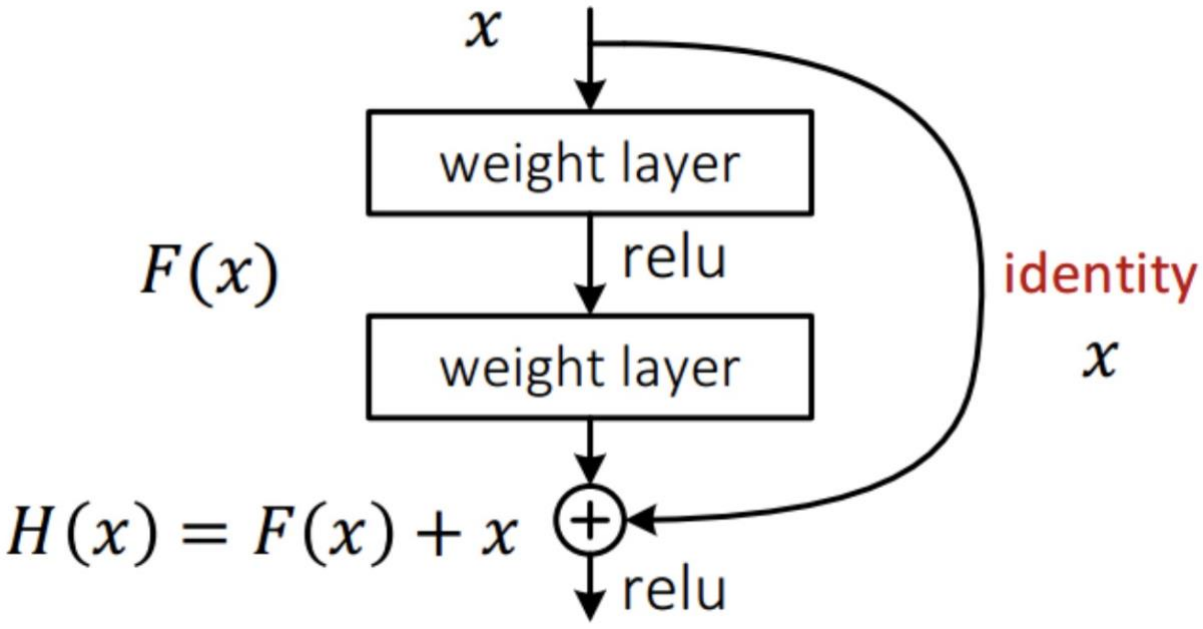
The main innovation of this algorithm is on its inception. This is a Network in Network structure, whose Nodes are become a network in this algorithm. By this, the width and depth of whole network structure could be expanded and make the performance better than former about two to three times.





ResNet:

The innovation of this algorithm is main on its residuals network, which is the training issue with highly deep layer. This algorithm learn from Highway Network, adding an specialized tunnel to make the input to output directly.



Citation:

1.http://www.cnblogs.com/52machinelearning/p/5821591.html

2.http://www.sohu.com/a/134347664_642762

2 Decision trees, entropy and information gain

2.1

$$H(s) = H(\frac{p}{p+n})$$

$$= \frac{p}{p+n} log_2(\frac{p+n}{p}) \frac{n}{p+n} log_2(\frac{p+n}{n})$$

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

$$= log_2(2) = 1$$

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

2.2

Misclassification rate of A $= \frac{1}{2}(1 - max(\frac{300}{400}, \frac{100}{400})) + \frac{1}{2}(1 - max(\frac{100}{400}, \frac{300}{400})) = \frac{1}{4}$
Misclassification rate of B $= \frac{3}{4}(1 - max(\frac{200}{600}, \frac{400}{600})) + \frac{1}{4}(1 - max(\frac{200}{200}, \frac{0}{200})) = \frac{1}{4}$
Entropy of A $= \frac{3}{4} log(3) - 1 \approx 0.1887$
Entropy of B $= \frac{3}{2} - \frac{3}{4} log(3) \approx 0.3113$
Gini index of A $= 2\frac{400}{800}\frac{400}{800} - 4\frac{400}{800}\frac{300}{400}\frac{100}{400} = \frac{1}{8}$
Gini index of B $= 2\frac{400}{800}\frac{400}{800} - 2\frac{600}{800}\frac{400}{600}\frac{200}{600} - 0 = \frac{1}{6}$
According to the reduction in cost using methods above, we prefer model B as our split model.

2.3 Can the misclassification rate ever increase when splitting on a feature? If so, give an example. If not, give a proof.

Yes, it is possible. Ex: consider a dataset with all 800 from class C_1 and 0 from class C_2. If a feature splits it into two leaves (200,200) and (200,200), then the misclassification rate will be larger.

3 Bagging

3.1

$$E_{bag} = E_X[\varepsilon_{bag}(x)^2]$$

$$= E_X[\{(\frac{1}{L}\sum_{l=1}^{L}(f(x) + \varepsilon_l(x))) - f(x)\}^2]$$

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

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

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

3.2

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

by Jeason's inequity, we have:

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

Take mean of both side, we have:

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

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

$$= E_{av}$$

4 Fully connected neural networks and convolutional neural networks

Problem 4.1 Fully connected feedforward neural networks: a modular approach

Problem 4.1.1: Affine layer: forward

Testing affine_forward function:
difference: 9.76984772881e-10

Problem 4.1.2: Affine layer: backward

Testing affine_backward function:
dx error: 9.40510302326e-11
dtheta error: 4.83474453381e-11
dtheta0 error: 3.62121786767e-11

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: 3.27563971569e-12

Sandwich layers

Testing affine_relu_backward:
dx error: 5.4113712782e-11
dtheta error: 1.74437470977e-08
dtheta0 error: 3.27559209021e-12

Loss layers: softmax and SVM

Testing svm_loss:
loss: 8.99899398558
dx error: 8.18289447289e-10

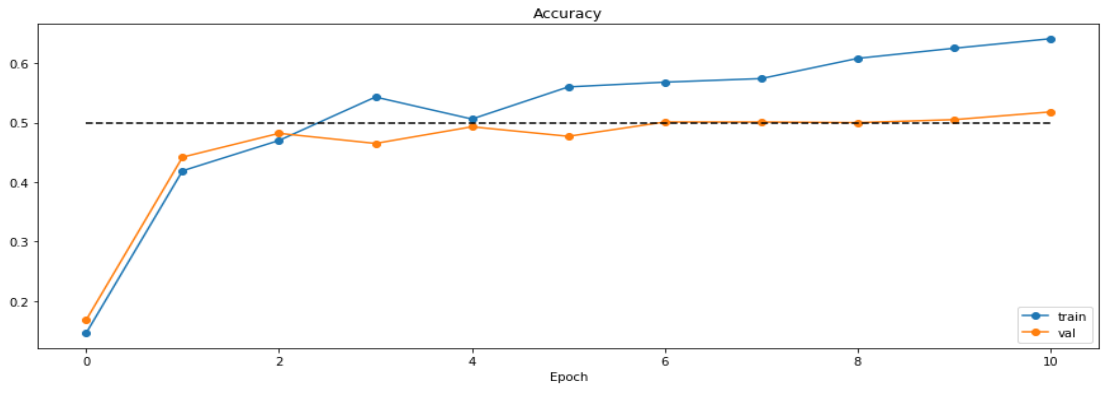
Testing softmax_loss:
loss: 2.30248491427
dx error: 9.96098395061e-09

Problem 4.1.5: Two layer network

Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
theta1 relative error: 1.22e-08
theta1_0 relative error: 6.55e-09
theta2 relative error: 3.48e-10
theta2_0 relative error: 4.33e-10
Running numeric gradient check with reg = 0.7
theta1 relative error: 8.18e-07
theta1_0 relative error: 1.09e-09
theta2 relative error: 2.85e-08
theta2_0 relative error: 9.09e-10

Problem 4.1.6: Overfitting a two layer network

We selected the size of the hidden layer =100, leaning rate = 1e-03, num_epochs=10 and batch_size = 100



Problem 4.1.7: Multilayer network

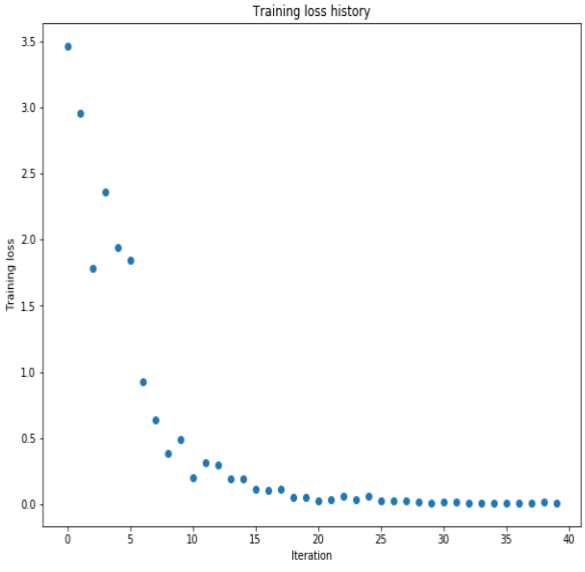
Initial loss and gradient check

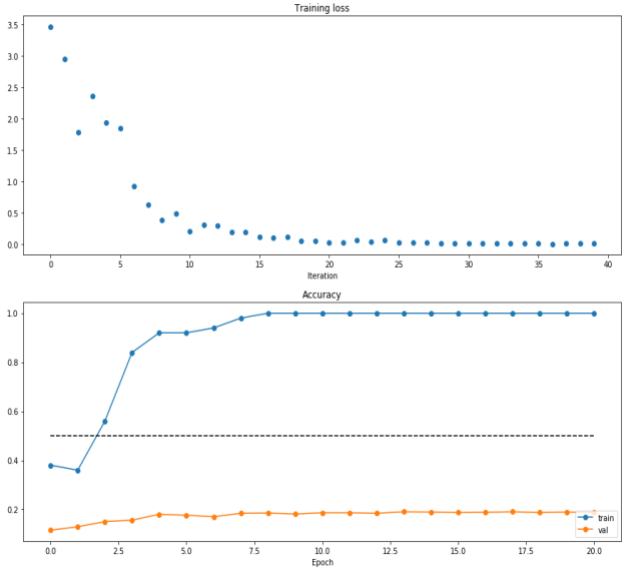
Running check with reg = 0
Initial loss: 2.30287680132
theta1 relative error: 8.23e-07
theta1_0 relative error: 1.25e-08
theta2 relative error: 1.74e-07
theta2_0 relative error: 7.56e-09
theta3 relative error: 7.36e-08
theta3_0 relative error: 8.00e-11
Running check with reg = 3.14
Initial loss: 6.88050676604
theta1 relative error: 1.76e-08
theta1_0 relative error: 3.50e-07
theta2 relative error: 4.65e-08
theta2_0 relative error: 1.98e-08
theta3 relative error: 1.00e+00
`theta3_0 relative error: 2.42e-10`

Yes, the initial loss looks reasonable, when reg = 0, initial loss = 2.303 ≈ ln C where C = 10. Moreover, the initial loss become larger as we add regularization.

Problem 4.1.8: Overfitting a three layer network

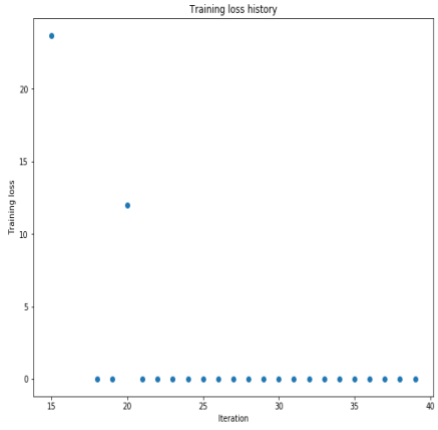
We selected the weight_scale = 2e-2 and learning rate = 1e-2

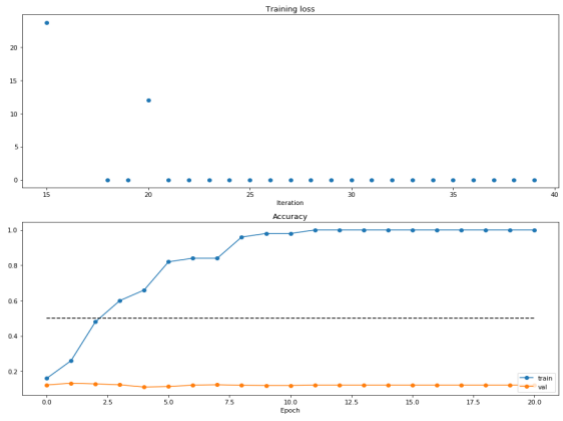




Problem 4.1.9: Overfitting a five layer network

We selected the weight_scale = 3e-1 and learning rate = 2e-5

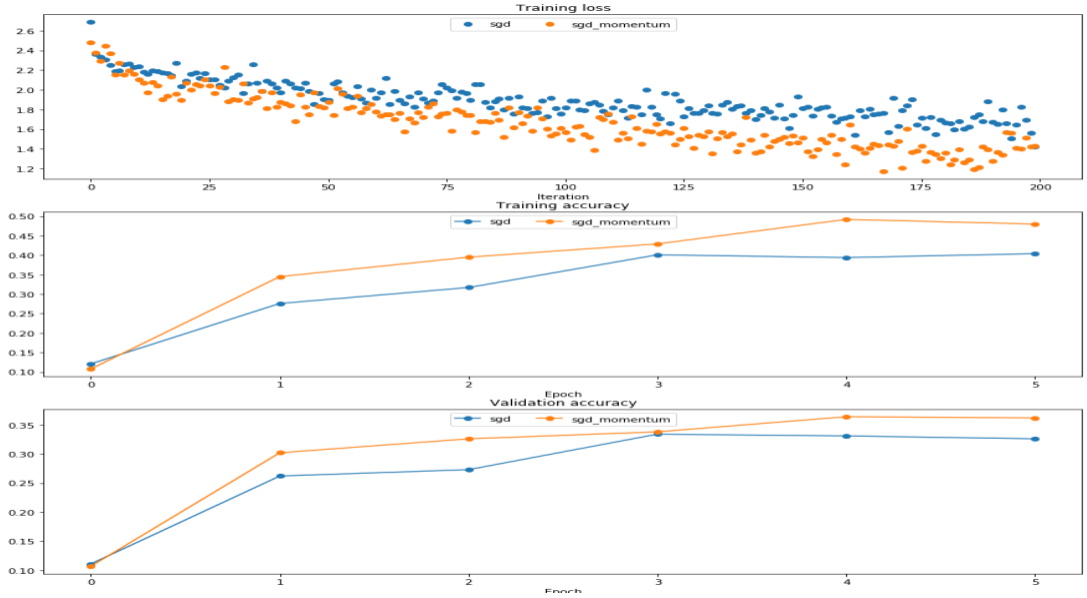




I think it's harder to train the five-layer net (tried 5 sets) than the three-layer net (tried 3 sets) since the five-layer net is deeper which makes the backward spread of the gradient become more difficult.

Problem 4.1.10: SGD+Momentum

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



The figure above shows the SGD + momentum update rule converge faster than SGD.

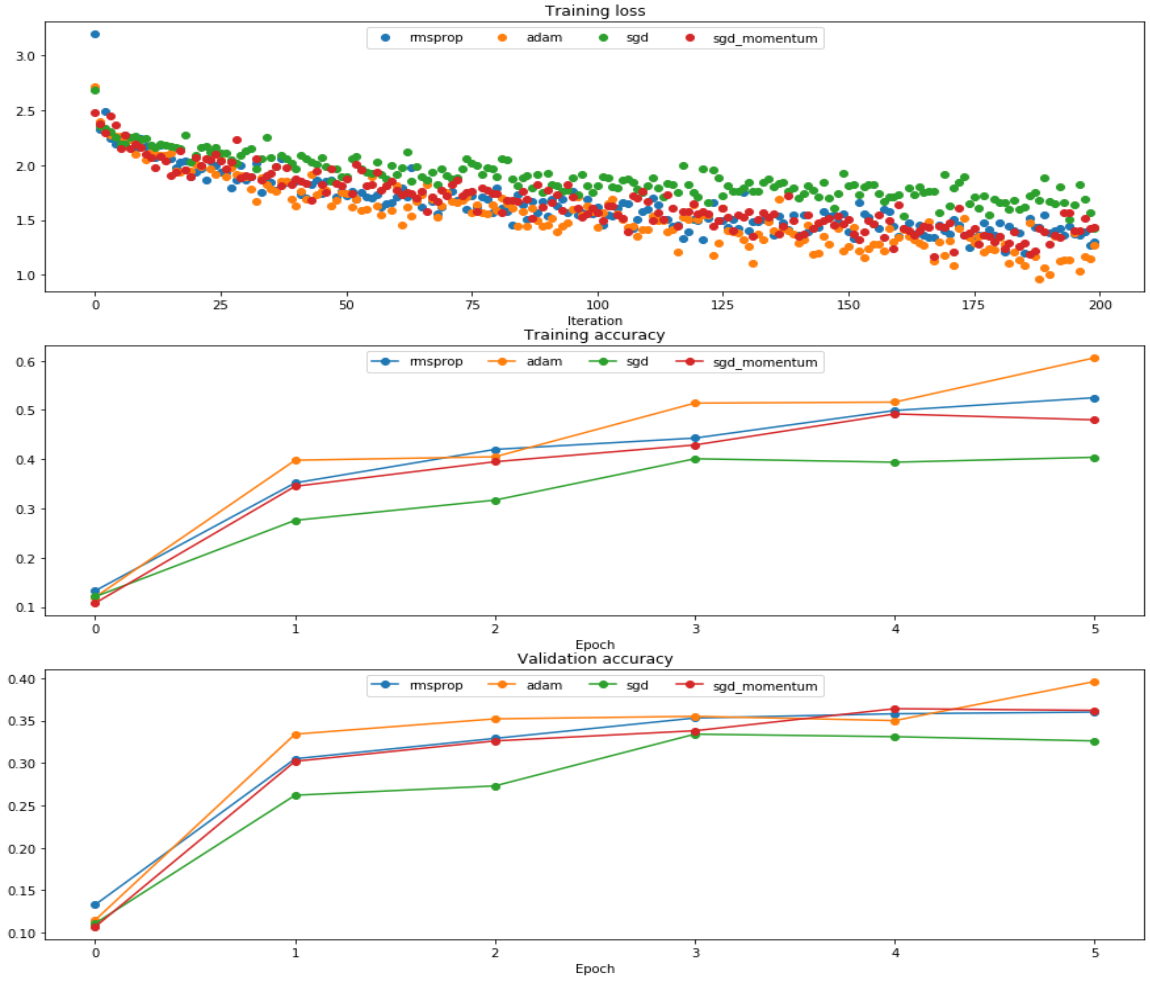
Problem 4.1.11: RMSProp

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

Problem 4.1.12: Adam

next_theta error: 1.13988746733e-07
v error: 4.20831403811e-09
m error: 4.21496319311e-09

We can see from the figure below that the converge speed is highest for adam and lowest for s gd.



Problem 4.2 Dropout

Problem 4.2.1: Dropout forward pass

Running tests with p = 0.3
Mean of input: 9.99975704885
Mean of train-time output: 10.0019121054
Mean of test-time output: 9.99975704885
Fraction of train-time output set to zero: 0.299856
Fraction of test-time output set to zero: 0.0

Running tests with p = 0.6
Mean of input: 9.99975704885
Mean of train-time output: 10.0144625909
Mean of test-time output: 9.99975704885
Fraction of train-time output set to zero: 0.59954
Fraction of test-time output set to zero: 0.0

Running tests with p = 0.75
Mean of input: 9.99975704885
Mean of train-time output: 9.97309830654
Mean of test-time output: 9.99975704885
Fraction of train-time output set to zero: 0.750632
Fraction of test-time output set to zero: 0.0

Problem 4.2.2: Dropout backward pass

dx relative error: 1.89290647574e-11

Problem 4.2.3: Fully connected nets with dropout

Running check with dropout = 0
Initial loss: 2.30304316117
theta1 relative error: 4.80e-07
theta1_0 relative error: 2.03e-08
theta2 relative error: 1.97e-07
theta2_0 relative error: 1.69e-09
theta3 relative error: 1.56e-07
theta3_0 relative error: 1.11e-10

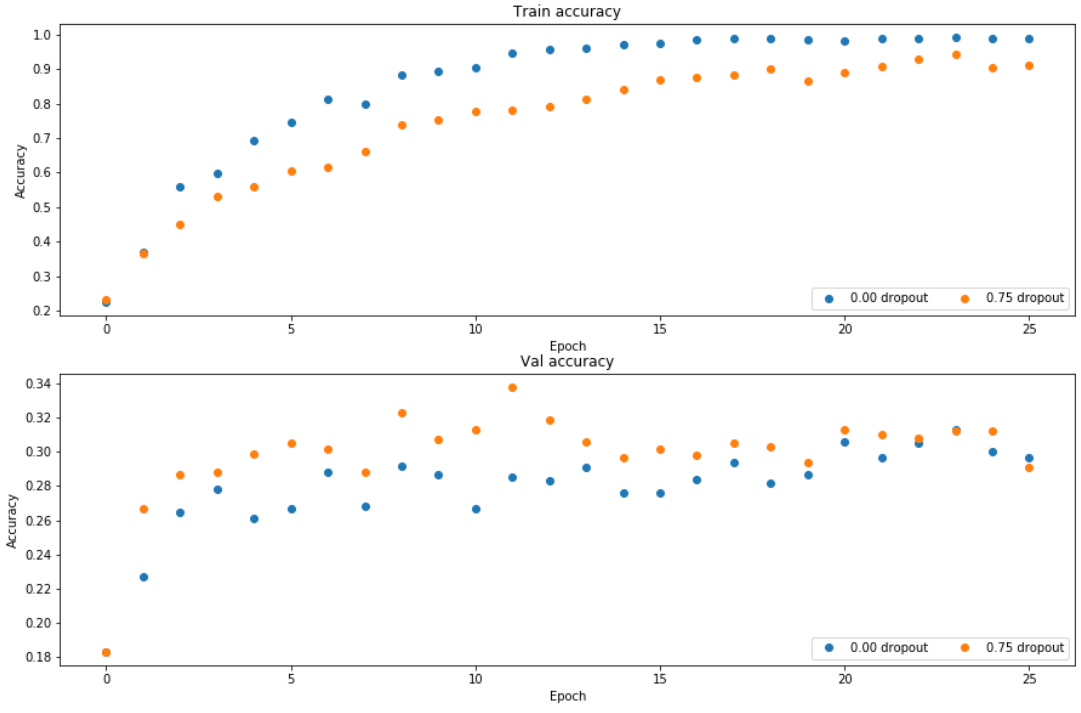
Running check with dropout = 0.25
Initial loss: 2.298956181
theta1 relative error: 2.37e-08
theta1_0 relative error: 1.36e-09
theta2 relative error: 1.86e-07
theta2_0 relative error: 2.71e-09
theta3 relative error: 1.89e-08

theta3_0 relative error: 1.89e-10

Running check with dropout = 0.5
Initial loss: 2.30424261716
theta1 relative error: 1.21e-07
theta1_0 relative error: 2.28e-08
theta2 relative error: 2.45e-08
theta2_0 relative error: 6.84e-10
theta3 relative error: 8.06e-07
theta3_0 relative error: 1.28e-10

Problem 4.2.4: Experimenting with fully connected nets with dropout

We can see from the figure below that when dropout is used, the training accuracy is lower while the validation accuracy is higher, since using dropout can prevent overfitting issue.



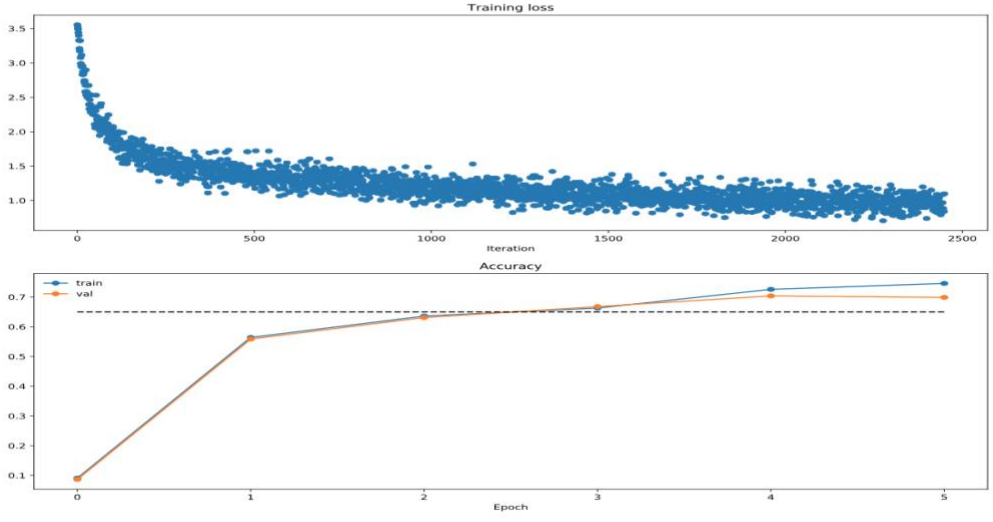
Problem 4.3: Training a fully connected network for the CIFAR-10 dataset with dropout

After run sets of different hyper parameters, the best model we chose is with learning_rate:0.0 001, dropout:0.1, reg:0.03 and weight_scales:0.01.

(Iteration 1 / 6520) loss: 3.959299
(Epoch 0 / 20) train acc: 0.159000; val_acc: 0.138000
(Iteration 201 / 6520) loss: 2.363343
(Epoch 1 / 20) train acc: 0.462000; val_acc: 0.453000
(Iteration 401 / 6520) loss: 2.110817
(Iteration 601 / 6520) loss: 1.920278
(Epoch 2 / 20) train acc: 0.473000; val_acc: 0.496000
(Iteration 801 / 6520) loss: 1.726907
(Epoch 3 / 20) train acc: 0.534000; val_acc: 0.495000
(Iteration 1001 / 6520) loss: 1.583383
(Iteration 1201 / 6520) loss: 1.629909
(Epoch 4 / 20) train acc: 0.555000; val_acc: 0.500000
(Iteration 1401 / 6520) loss: 1.551707
(Iteration 1601 / 6520) loss: 1.581095
(Epoch 5 / 20) train acc: 0.553000; val_acc: 0.511000
(Iteration 1801 / 6520) loss: 1.502301
(Epoch 6 / 20) train acc: 0.575000; val_acc: 0.525000
(Iteration 2001 / 6520) loss: 1.491617
(Iteration 2201 / 6520) loss: 1.444362
(Epoch 7 / 20) train acc: 0.578000; val_acc: 0.532000
(Iteration 2401 / 6520) loss: 1.504876
(Iteration 2601 / 6520) loss: 1.533616
(Epoch 8 / 20) train acc: 0.577000; val_acc: 0.532000
(Iteration 2801 / 6520) loss: 1.552167
(Epoch 9 / 20) train acc: 0.603000; val_acc: 0.527000
(Iteration 3001 / 6520) loss: 1.396799
(Iteration 3201 / 6520) loss: 1.449900
(Epoch 10 / 20) train acc: 0.580000; val_acc: 0.522000
(Iteration 3401 / 6520) loss: 1.424513
(Epoch 11 / 20) train acc: 0.640000; val_acc: 0.529000
(Iteration 3601 / 6520) loss: 1.387792
(Iteration 3801 / 6520) loss: 1.385131
(Epoch 12 / 20) train acc: 0.567000; val_acc: 0.537000
(Iteration 4001 / 6520) loss: 1.401319
(Iteration 4201 / 6520) loss: 1.369493
(Epoch 13 / 20) train acc: 0.641000; val_acc: 0.524000
(Iteration 4401 / 6520) loss: 1.386015
(Epoch 14 / 20) train acc: 0.641000; val_acc: 0.530000
(Iteration 4601 / 6520) loss: 1.316762

(Iteration 4801 / 6520) loss: 1.435967
(Epoch 15 / 20) train acc: 0.619000; val_acc: 0.565000
(Iteration 5001 / 6520) loss: 1.157730
(Iteration 5201 / 6520) loss: 1.263927
(Epoch 16 / 20) train acc: 0.641000; val_acc: 0.531000
(Iteration 5401 / 6520) loss: 1.406826
(Epoch 17 / 20) train acc: 0.639000; val_acc: 0.539000
(Iteration 5601 / 6520) loss: 1.403645
(Iteration 5801 / 6520) loss: 1.298459
(Epoch 18 / 20) train acc: 0.673000; val_acc: 0.539000
(Iteration 6001 / 6520) loss: 1.363221
(Epoch 19 / 20) train acc: 0.657000; val_acc: 0.542000
(Iteration 6201 / 6520) loss: 1.175109
(Iteration 6401 / 6520) loss: 1.232952
(Epoch 20 / 20) train acc: 0.657000; val_acc: 0.568000

We achieved > 50% accuracy on the validation set and our best validation accuracy achieved within 20 epochs is 56.8%.



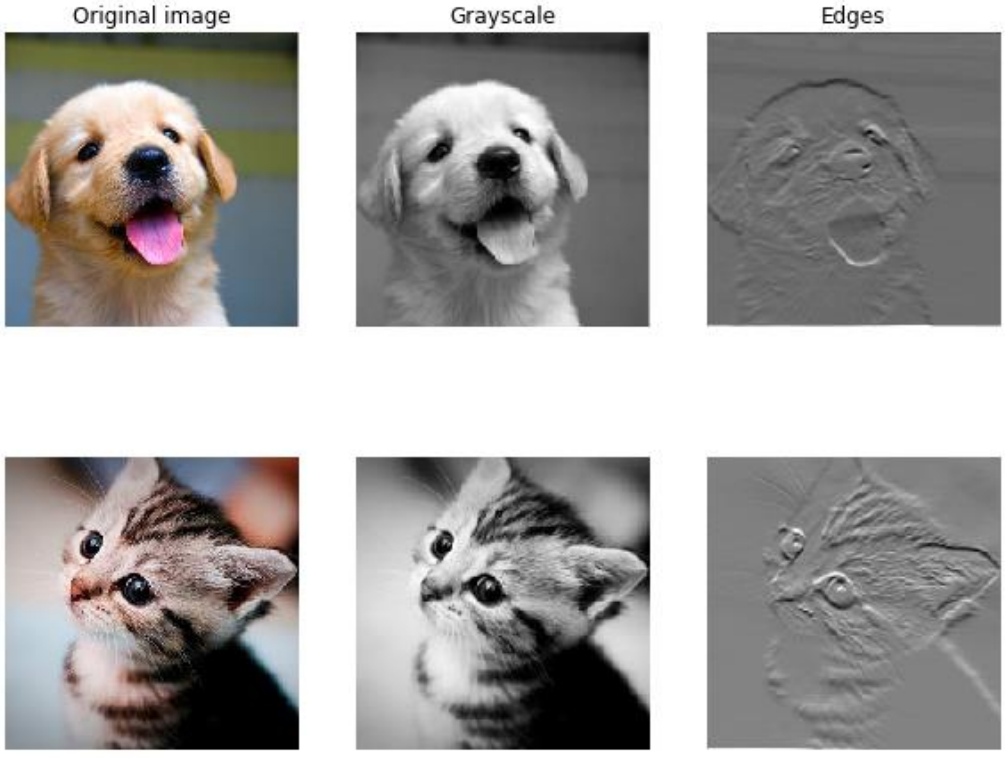
After run our best model on the validation and test sets, the following is out validation/test accuracy achieved:

Validation set accuracy: 0.568
Test set accuracy: 0.5

Problem 4.4 Convolutional neural networks

Problem 4.4.1: Convolution: naive forward pass

Testing conv_forward_naive
difference: 2.21214764175e-08



Problem 4.4.2: Convolution: naive backward pass

Testing conv_backward_naive function
dx error: 2.75331807334e-09
dtheta error: 1.80951825745e-10
dtheta0 error: 5.69908341076e-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: 3.27563635525e-12

Fast layers

Testing conv_forward_fast:
Naive: 5.493994s
Fast: 0.020059s
Speedup: 273.893490x
Difference: 1.92687594866e-11

Testing conv_backward_fast:
Naive: 8.741904s
Fast: 0.015537s
Speedup: 562.649853x
dx difference: 2.52358199906e-10
dw difference: 3.766295208e-12
db difference: 0.0

Testing pool_forward_fast:
Naive: 0.411300s
fast: 0.002900s
speedup: 141.833265x
difference: 0.0

Testing pool_backward_fast:
Naive: 0.426634s
speedup: 34.325699x
dx difference: 0.0

Convolutional sandwich layers

Testing conv_relu_pool
dx error: 9.36063841153e-08
dtheta error: 2.04776061435e-09
dtheta0 error: 2.24324063601e-11

Testing conv_relu:
dx error: 1.1917961062e-09
dtheta error: 6.45846455473e-10
dtheta0 error: 2.87857259536e-11

Problem 4.4.5: Three layer convolutional neural network

Testing the CNN: loss computation

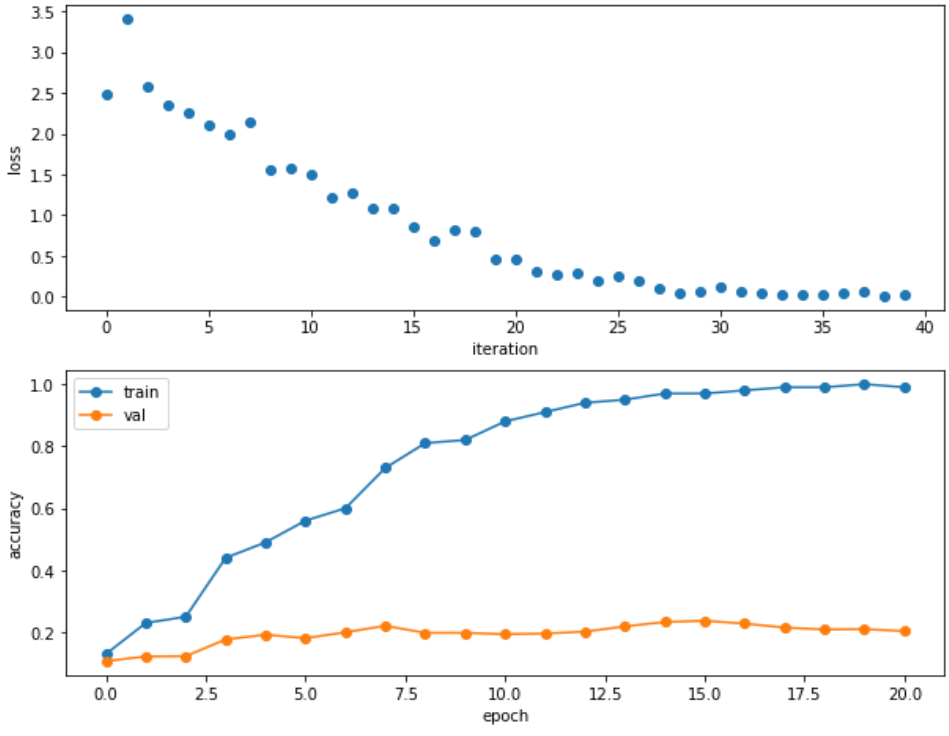
Initial loss (no regularization): 2.30258559901
Initial loss (with regularization): 2.50839094538

Testing the CNN: gradient check

theta1 max relative error: 1.733680e-04
theta1_0 max relative error: 2.222898e-05
theta2 max relative error: 4.631238e-04
theta2_0 max relative error: 1.658187e-07
theta3 max relative error: 2.855207e-05
theta3_0 max relative error: 1.077213e-09

Testing the CNN: overfit small data

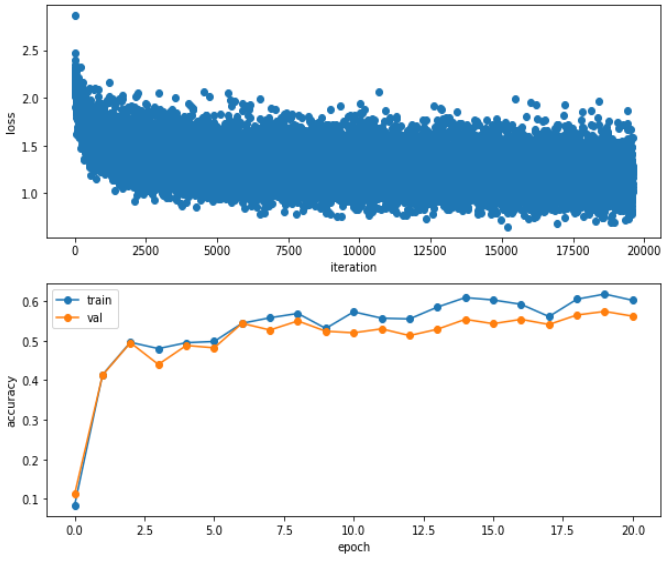
We can see from the plot that it's clear overfitting with very high training accuracy as well as very low validation accuracy.

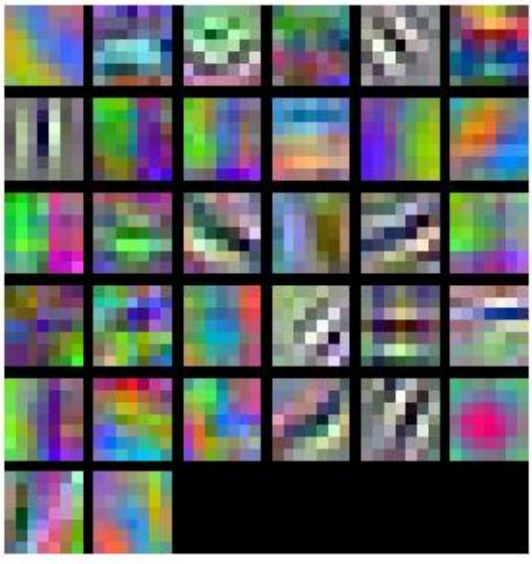


Problem 4.4.6: Train the CNN on the CIFAR-10 data

(Epoch 0 / 20) train acc: 0.084000; val_acc: 0.112000
(Epoch 1 / 20) train acc: 0.414000; val_acc: 0.413000
(Epoch 2 / 20) train acc: 0.496000; val_acc: 0.495000
(Epoch 3 / 20) train acc: 0.480000; val_acc: 0.440000
(Epoch 4 / 20) train acc: 0.495000; val_acc: 0.488000
(Epoch 5 / 20) train acc: 0.498000; val_acc: 0.482000
(Epoch 6 / 20) train acc: 0.544000; val_acc: 0.544000
(Epoch 7 / 20) train acc: 0.558000; val_acc: 0.527000
(Epoch 8 / 20) train acc: 0.569000; val_acc: 0.550000
(Epoch 9 / 20) train acc: 0.531000; val_acc: 0.524000
(Epoch 10 / 20) train acc: 0.573000; val_acc: 0.520000
(Epoch 11 / 20) train acc: 0.557000; val_acc: 0.530000
(Epoch 12 / 20) train acc: 0.555000; val_acc: 0.513000
(Epoch 13 / 20) train acc: 0.585000; val_acc: 0.529000
(Epoch 14 / 20) train acc: 0.609000; val_acc: 0.554000
(Epoch 15 / 20) train acc: 0.603000; val_acc: 0.543000
(Epoch 16 / 20) train acc: 0.592000; val_acc: 0.554000
(Epoch 17 / 20) train acc: 0.561000; val_acc: 0.541000
(Epoch 18 / 20) train acc: 0.605000; val_acc: 0.565000
(Epoch 19 / 20) train acc: 0.618000; val_acc: 0.574000
(Epoch 20 / 20) train acc: 0.602000; val_acc: 0.562000

We achieved > 50% accuracy on the validation set and our best validation accuracy achieved within 20 epochs is 57.4%. Moreover the figure below visualize the first-layer convolutional fil ters from the trained network



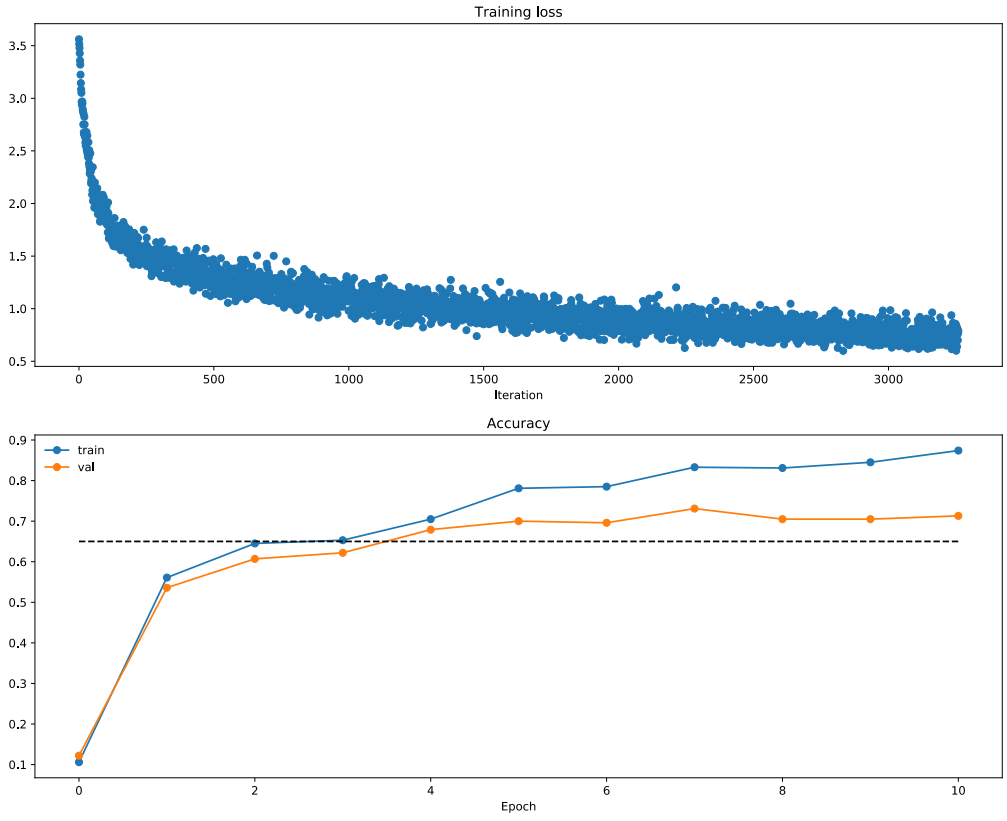


Extra credit: Problem 4.4.7: Train the best model for CIFAR-10 data

We chose the model with weight_scale=0.01, hidden_dim=300, reg=0.01, num_filters=32, filte r_size=5, learning_rate = 3e-4, batch_size = 100.

The network architecture is as below: (conv-relu-pool) – conv - conv – (affine-relu) – affine – softmax

(Iteration 1 / 3260) loss: 3.561390
(Epoch 0 / 10) train acc: 0.106000; val_acc: 0.122000
(Iteration 201 / 3260) loss: 1.475688
(Epoch 1 / 10) train acc: 0.561000; val_acc: 0.536000
(Iteration 401 / 3260) loss: 1.316818
(Iteration 601 / 3260) loss: 1.465381
(Epoch 2 / 10) train acc: 0.645000; val_acc: 0.607000
(Iteration 801 / 3260) loss: 1.063184
(Epoch 3 / 10) train acc: 0.653000; val_acc: 0.622000
(Iteration 1001 / 3260) loss: 1.173276
(Iteration 1201 / 3260) loss: 1.140105
(Epoch 4 / 10) train acc: 0.705000; val_acc: 0.679000
(Iteration 1401 / 3260) loss: 0.862790
(Iteration 1601 / 3260) loss: 0.861681
(Epoch 5 / 10) train acc: 0.781000; val_acc: 0.700000
(Iteration 1801 / 3260) loss: 0.811049
(Epoch 6 / 10) train acc: 0.785000; val_acc: 0.696000
(Iteration 2001 / 3260) loss: 0.917931
(Iteration 2201 / 3260) loss: 0.849614
(Epoch 7 / 10) train acc: 0.833000; val_acc: 0.731000
(Iteration 2401 / 3260) loss: 0.962506
(Iteration 2601 / 3260) loss: 0.917935
(Epoch 8 / 10) train acc: 0.831000; val_acc: 0.705000
(Iteration 2801 / 3260) loss: 0.779084
(Epoch 9 / 10) train acc: 0.845000; val_acc: 0.705000
(Iteration 3001 / 3260) loss: 0.779691
(Iteration 3201 / 3260) loss: 0.864678
(Epoch 10 / 10) train acc: 0.874000; val_acc: 0.713000



We achieved > 65% accuracy on the validation set and our best validation accuracy achieved within 5 epochs is 70.4%.