comp540 Statistical Machine Learning HW5

Chuanwenjie Wei, Yiqing Lu

April 3, 2017

1 Deep neural networks

• Since deep networks have more layer than shallow networks, according the definition, deep networks could use the intermediate hidden layers and thus extracting or building better features than shallow models.

In addition, deep networks could implement functions with higher complexity than shallow ones. While there are studies that a shallow network can fit any function, it will need to be really fat, meaning that the number of parameters should increase a lot.[1] In general, the deeper network has stronger power and express more information and model relationship more accurately.

• Leaky ReLU. Leaky ReLUs are one attempt to fix the "dying ReLU" problem when a weight could be trained to zero. Instead of the function being zero when x $<$ 0, a leaky ReLU will instead have a small non-negative slope (of 0.01, or so).[2] That is, the function computes:

$$f(x) = \begin{cases} x & x > 0 \\ ax & x \leq 0 \end{cases}$$

In contrast to ReLU, in which the negative part is totally dropped, leaky ReLU assigns a non-zero slope to it, which could make the information in negative axis could be retained.

•Comparison:

[1] source: https://www.quora.com/What-is-the-difference-between-deep-and-shallow-neural-networks

[2] source: http://cs231n.github.io/neural-networks-1/

Model	AlexNet	VGG	GoogleNet	ResNet
Layers	**8**	**19**	**22**	**152**
Top-5 error	**16.4%**	**7.3%**	**6.7%**	**3.57%**
Data Augmentation	+	+	+	+
Inception(NIN)	-	-	+	-
Convolutional Layers	**5**	**16**	**21**	**151**
Filter Size	**11,5,3**	**3**	**7,1,3,5**	**7,1,3,5**
Full-connected layers	**3**	**3**	**1**	**1**
Size of Full-connected layers	**4096,4096,1000**	**4096,4096,1000**	**1000**	**1000**
Dropout	+	+	+	+
Local Response Normalization	+	-	+	-
Batch Normalization	-	-	-	+

2 Decision trees, entropy and information gain

2.1 Show the value of H(S)

Since

$$H(q) = -qlogq - (1-q)\,log\,(1-q)$$

So

$$\frac{\partial H}{\partial q} = -logq + log\,(1-q)\,,\ \frac{\partial H}{\partial^2 q} = -\frac{1}{q} - \frac{1}{1-q}$$

Since $q \in (0,1)$

So $\frac{\partial H}{\partial^2 q} < 0$, indicating the entropy is concave.

Thus we can get a global maximum by setting $\frac{\partial H}{\partial q} = 0$, which yiels $q = 0.5$

And $H(q)_{max} = H(0.5) = 1$ (Note that log is 2 base)

Since

$$H(S) = H(p/(p+n))$$

So the global maximum is derived by setting:

$$\frac{p}{p+n} = 0.5 \Longrightarrow p = n$$

2.2 Compare model A and B

•Misclassification Rate:

The misclassification rate of A is:

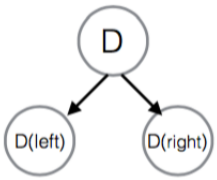
$$err_A = \frac{100+100}{800} = \frac{1}{4}$$

And the misclassification rate of B is:

$$err_B = \frac{200}{800} = \frac{1}{4}$$

•Entrophy Gain:

We know that:



So for model A,

$$H(D) = 1$$

$$H(D_1) = -\frac{300}{400} \times log\frac{3}{4} - \frac{100}{400} \times log\frac{1}{4} = 0.8113$$

$$H(D_2) = -\frac{100}{400} \times log\frac{1}{4} - \frac{300}{400} \times log\frac{3}{4} = 0.8113$$

Therefore,

$$Entrophy(A) = \frac{1}{2}H(D_1) + \frac{1}{2}H(D_2) = 0.8113$$

$$EntrophyGain(A) = H(D) - Entrophy(A) = 0.1887$$

For model B, we have:

$$H(D) = 1$$

$$H(D_1) = -\frac{200}{600} \times log\frac{2}{6} - \frac{400}{600} \times log\frac{4}{6} = 0.9131$$

$$H(D_2) = 0$$

So

$$Entrophy\,(B) \quad = \quad \frac{600}{800} \times H\,(D_1) + \frac{200}{800} \times H\,(D_2) = 0.6887$$

$$EntrophyGain\,(B) \quad = \quad H\,(D) - Entrophy\,(B) = 0.3113$$

•Gini Index:

For the whole dataset,

$$Gini \quad = \quad \frac{400}{800} \times \frac{400}{800} = \frac{1}{4}$$

And

$$Gini\,(A) \quad = \quad Gini\,(left) + Gini(right)$$

$$= \quad \frac{1}{2} \times \left(1 - \left(\frac{300}{400}\right)^2 - \left(\frac{100}{400}\right)^2\right) + \frac{1}{2} \times \left(1 - \left(\frac{100}{400}\right)^2 - \left(\frac{300}{400}\right)^2\right)$$

$$= \quad \frac{3}{8}$$

With the same procedure,

$$Gini\,(B) \quad = \quad \frac{600}{800} \times \left(1 - \left(\frac{400}{600}\right)^2 - \left(\frac{200}{600}\right)^2\right) + \frac{200}{800} \times \left(1 - \left(\frac{200}{200}\right)^2 - \left(\frac{0}{200}\right)^2\right) = \frac{1}{3}$$

2.3 The change of misclassification rate

No, it will not increase

Proof:

Suppose a parent node S have m children and each has p_k positive and n_k negative examples, respectively, for k=1...m.

Then we can derive the misclassification rate for the parent is:

$$Err_P \quad = \quad \frac{min\left(\sum_{k=1}^{m} p_k, \sum_{k=1}^{m} n_k\right)}{\sum_{k=1}^{m} (p_k + n_k)}$$

And the misclassification rate for all children is:

$$Err_C = \frac{\sum_{k=1}^{m} min(p_k, n_k)}{\sum_{k=1}^{m} (p_k + n_k)}$$

Since

$$min(p_k, n_k) \leq p_k,\ min(p_k, n_k) \leq n_k$$

So

$$\sum_{k=1}^{m} min(p_k, n_k) \leq \sum_{k=1}^{m} p_k$$

And

$$\sum_{k=1}^{m} min(p_k, n_k) \leq \sum_{k=1}^{m} n_k$$

Thus, we have:

$$\sum_{k=1}^{m} min(p_k, n_k) \leq min\left(\sum_{k=1}^{m} p_k, \sum_{k=1}^{m} n_k\right)$$

$$\Longrightarrow Err_C \leq Err_P$$

So splitting on a feature will not increase misclassification rate.

3 Bagging

• Since

$$\varepsilon_{bag}(x) = \frac{1}{L}\sum_{l=1}^{L}(f(x) + \varepsilon_l(x)) - f(x) = \frac{\sum_{l=1}^{L}\varepsilon_l(x)}{L}$$

And $\varepsilon_l(x) \sim N\left(0, \sigma_l^2\right)$

So $E(\varepsilon_{bag}(x)) = 0$

And we also know that $E(\varepsilon_m \varepsilon_l) = 0$, therefore,

$$E_{bag} = E\left(\varepsilon_{bag}(x)^2\right) = var\left(\varepsilon_{bag}(x)\right),$$

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

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

For E_{av}, since $E(\varepsilon_l(x)) = 0$, we have:

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

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

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

• Let $f(x) = x^2,\ \lambda_l = \frac{1}{L}\ for\ l = 1...L$

So we can get:

$$\sum_{l=1}^{L}\lambda_l f(\varepsilon_l) = \frac{1}{L}\sum_{l=1}^{L}\varepsilon_l^2$$

According to Jensen's inequality,

$$\sum_{l=1}^{L}\lambda_l f(\varepsilon_l) \geq f\left(\sum_{l=1}^{L}\lambda_l\varepsilon_l\right)$$

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

$$= \varepsilon_{bag}^2$$

So

$$\frac{1}{L}\sum_{l=1}^{L}\varepsilon_l^2 \quad \geq \quad \varepsilon_{bag}^2$$

Take expectation on both sides:

$$E\left(\frac{1}{L}\sum_{l=1}^{L}\varepsilon_l^2\right) \quad \geq \quad E\left(\varepsilon_{bag}^2\right)$$

$$\Longrightarrow \frac{1}{L}\sum_{l=1}^{L}E\left(\varepsilon_l^2\right) \quad \geq \quad E\left(\varepsilon_{bag}^2\right)$$

$$\Longrightarrow E_{av} \quad \geq \quad E_{bag}$$

4 Fully connected neural networks and convolutional neural networks

4.1 Fully connected feedforward neural networks: a modular approach

Problem 4.1.1: Affine layer: forward

An affine layer computes a linear function of inputs in a fully connected network, and the vectorized form for forward process is:

```
x_1 = np.reshape(x, [x.shape[0], -1]) # reshape x
out = np.dot(x_1, theta) + theta0
```

The test results in FullyConnectedNets.ipynb is:

```
Testing affine_forward function:
difference:  9.76984772881e-10
```

Problem 4.1.2: Affine layer: backward

The vectorized formulas for these three partial derivatives:

```
# 4-5 lines of code expected

x_1 = np.reshape(x, [x.shape[0], -1])

dx_1=np.dot(dout,theta.T)
dx=np.reshape(dx_1,x.shape)
dtheta=np.dot(x_1.T,dout)
dtheta0=np.sum(dout,axis=0)
```

The test result is:

```
Testing affine_backward function:
dx error:  2.58657848648e-09
dtheta error:  5.77899134366e-11
dtheta0 error:  1.0450925957e-11
```

Problem 4.1.3: ReLU layer: forward

A ReLU layer constitutes a non-linear layer in a fully-connected network, and the formula for forward process is:

```
out=np.where(x>0,x,0)
cache = x
return out, cache
```

Test result:

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

Problem 4.1.4: ReLU layer: backward

The relu backward propagate functions:

```
# 1 line of code expected. Hint: use
dx=np.where(x>0,dout,0)
```

Test result:

```
Testing relu_backward function:
dx error:  3.27561717971e-12
```

Note:

- **Sandwich layers:**

The test result for "sandwhich" layer:

```
Testing affine_relu_backward:
dx error:  3.21414432749e-10
dtheta error:  6.00637027995e-10
dtheta0 error:  8.64745236068e-12
```

- **Loss layers(softmax and SVM):**

Implementation results:

```
Testing svm_loss:
loss:  9.00007721663
dx error:  1.40215660067e-09

Testing softmax_loss:
loss:  2.30259322444
dx error:  7.51420916352e-09
```

Problem 4.1.5: Two layer network

In this part, we will implement a two layer fully connected network with architectural layers affine - relu - affine - softmax.

The code for this network is:

```
self.params['theta1']=weight_scale * np.random.randn(input_dim, hidden_dim)
self.params['theta1_0']=np.zeros(hidden_dim)
self.params['theta2']=weight_scale * np.random.randn(hidden_dim, num_classes)
self.params['theta2_0']=np.zeros(num_classes)
```

Implementation results tested with FullyConnectedNets.ipynb:

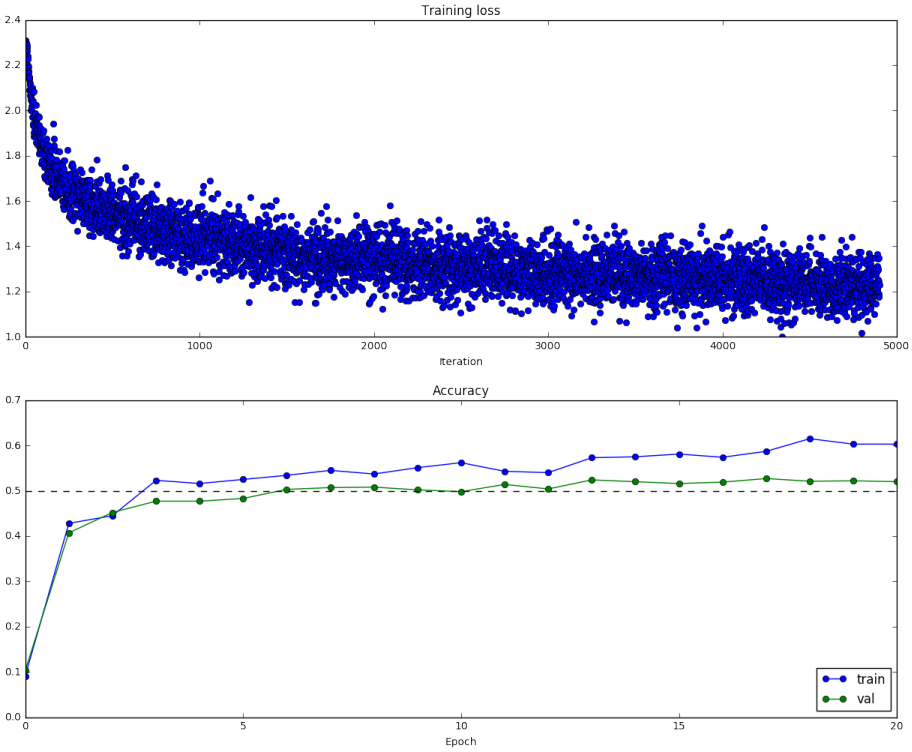
```
Running numeric gradient check with reg =  0.0
theta1 relative error: 1.22e-08
theta1_0 relative error: 6.55e-09
theta2 relative error: 3.48e-10
theta2_0 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
theta1 relative error: 8.18e-07
theta1_0 relative error: 1.09e-09
theta2 relative error: 2.85e-08
theta2_0 relative error: 9.09e-10
```

Problem 4.1.6: Overfitting a two layer network

With the hidden layers equal to 50, regulation term equals 0.1 and other paramters shown below:

```
model =  TwoLayerNet (hidden_dim=50,reg=0.1)
sgd_solver = Solver(model, data,
               update_rule='sgd',
               optim_config={
                 'learning_rate': 1e-3,
               },
               lr_decay=0.9,
               num_epochs=10, batch_size=200,
               print_every=200)
sgd_solver.train()
```

By doing so, We can achieve at least 50% accuracy after 10 epochs.



Problem 4.1.7: Multilayer network

Initialization, forward pass, and backward pass for a fully-connected network:

```
self.params['theta1'] = weight_scale * np.random.randn(input_dim, hidden_dims[0])
self.params['theta1_0'] = np.zeros(hidden_dims[0])
for i in range(1,len(hidden_dims)):
    self.params['theta' + str(i + 1)] = weight_scale * np.random.randn(hidden_dims[i - 1], hidden_dims[i])
    self.params['theta' + str(i + 1) + '_0'] = np.zeros(hidden_dims[i])
self.params['theta' + str(1 + len(hidden_dims))] = weight_scale * np.random.randn(hidden_dims[len(hidden_dims) - 1], num_classes)
self.params['theta'+ str(1 + len(hidden_dims)) + '_0'] = np.zeros(num_classes)
```

Check the initial loss and gradients:

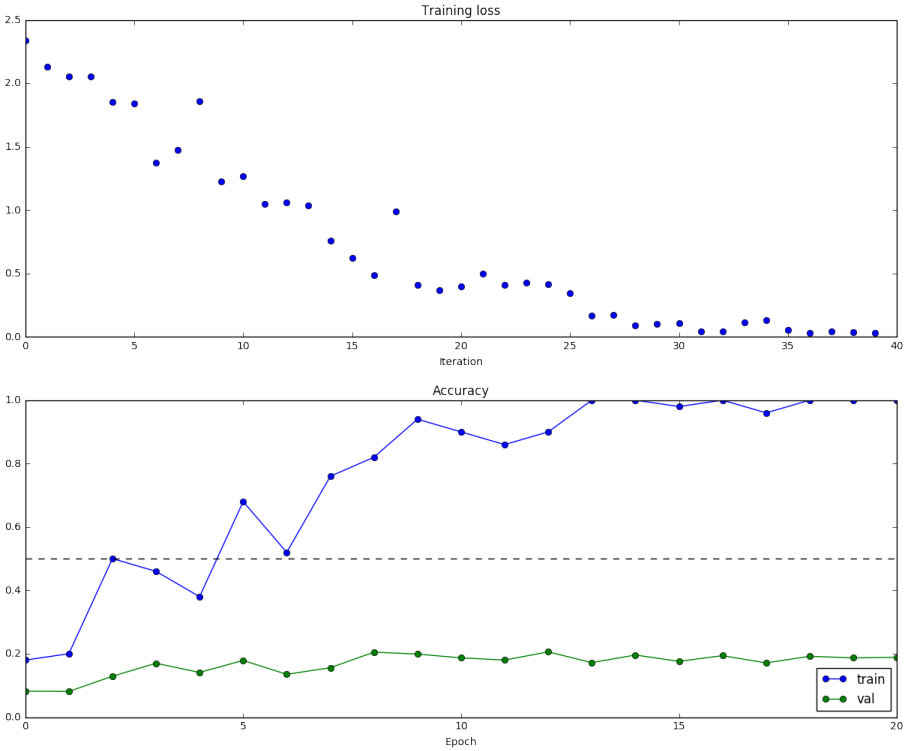
```
Running check with reg =  0
Initial loss:  2.30265446744
theta1 relative error: 8.95e-07
theta1_0 relative error: 9.90e-08
theta2 relative error: 1.35e-05
theta2_0 relative error: 3.93e-09
theta3 relative error: 1.44e-07
theta3_0 relative error: 9.09e-11
Running check with reg =  3.14
Initial loss:  7.08607690982
theta1 relative error: 4.40e-08
theta1_0 relative error: 4.05e-08
theta2 relative error: 4.92e-08
theta2_0 relative error: 4.55e-09
theta3 relative error: 1.00e+00
theta3_0 relative error: 3.69e-10
```

Problem 4.1.8: Overfitting a three layer network

We set both weight scale and learning rate to 1e-2, and try this three-layer network with 100 units in each hidden layer with 20 epochs:

```
Running check with reg =  0
Initial loss:  2.30265446744
theta1 relative error: 8.95e-07
theta1_0 relative error: 9.90e-08
theta2 relative error: 1.35e-05
theta2_0 relative error: 3.93e-09
theta3 relative error: 1.44e-07
theta3_0 relative error: 9.09e-11
Running check with reg =  3.14
Initial loss:  7.08607690982
theta1 relative error: 4.40e-08
theta1_0 relative error: 4.05e-08
theta2 relative error: 4.92e-08
theta2_0 relative error: 4.55e-09
theta3 relative error: 1.00e+00
theta3_0 relative error: 3.69e-10
```

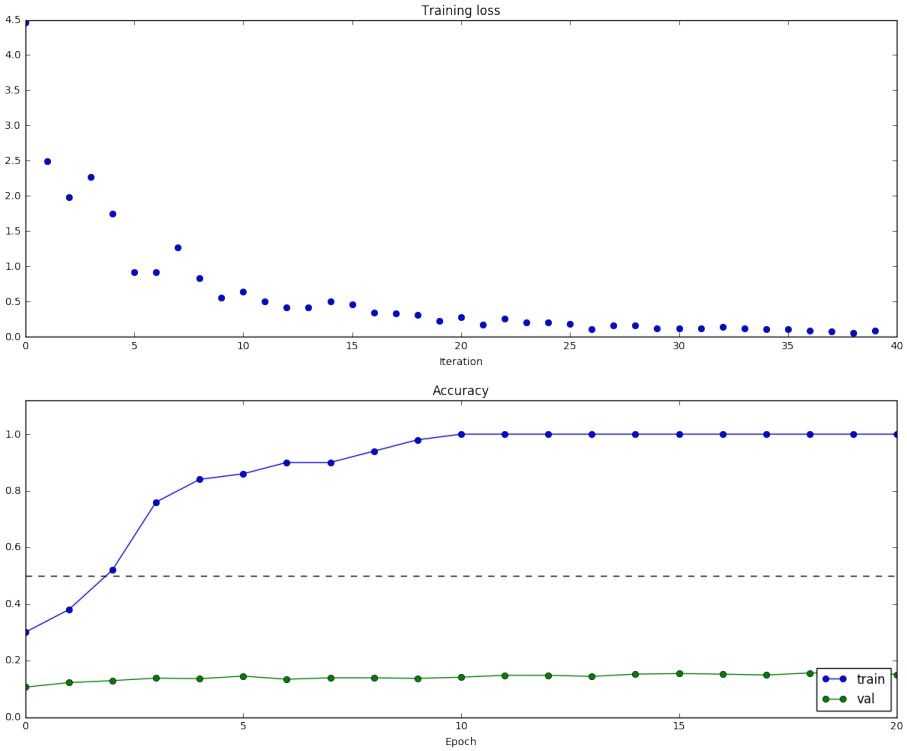
We can see that the accuracy in training set can reach 100% within 20 epochs. The training history can be plt below:



Problem 4.1.9: Overfitting a five layer network

With weight scale 5e-2, learning rate 5e-3, we can get the accuracy results and training history for this five-layer network with 100 units on each layer network as follows:

```
Running check with reg =  0
Initial loss:  2.30265446744
theta1 relative error: 8.95e-07
theta1_0 relative error: 9.90e-08
theta2 relative error: 1.35e-05
theta2_0 relative error: 3.93e-09
theta3 relative error: 1.44e-07
theta3_0 relative error: 9.09e-11
Running check with reg =  3.14
Initial loss:  7.08607690982
theta1 relative error: 4.40e-08
theta1_0 relative error: 4.05e-08
theta2 relative error: 4.92e-08
theta2_0 relative error: 4.55e-09
theta3 relative error: 1.00e+00
theta3_0 relative error: 3.69e-10
```



For the three-layer net, three sets of weight scaler and learning rate could achieve the training accuracy=1. For the five-layer net, two sets of weight scaler and learning rate could achieve the training accuracy=1. So, finding the parameter of weight scaler and learning rate for the five-layer net is more difficult than for the three-layer net.

•In the latter parts, we will try different update rules to make it easier to train deep networks.

Problem 4.1.10: SGD+Momentum

The SGD+momentum update rule:

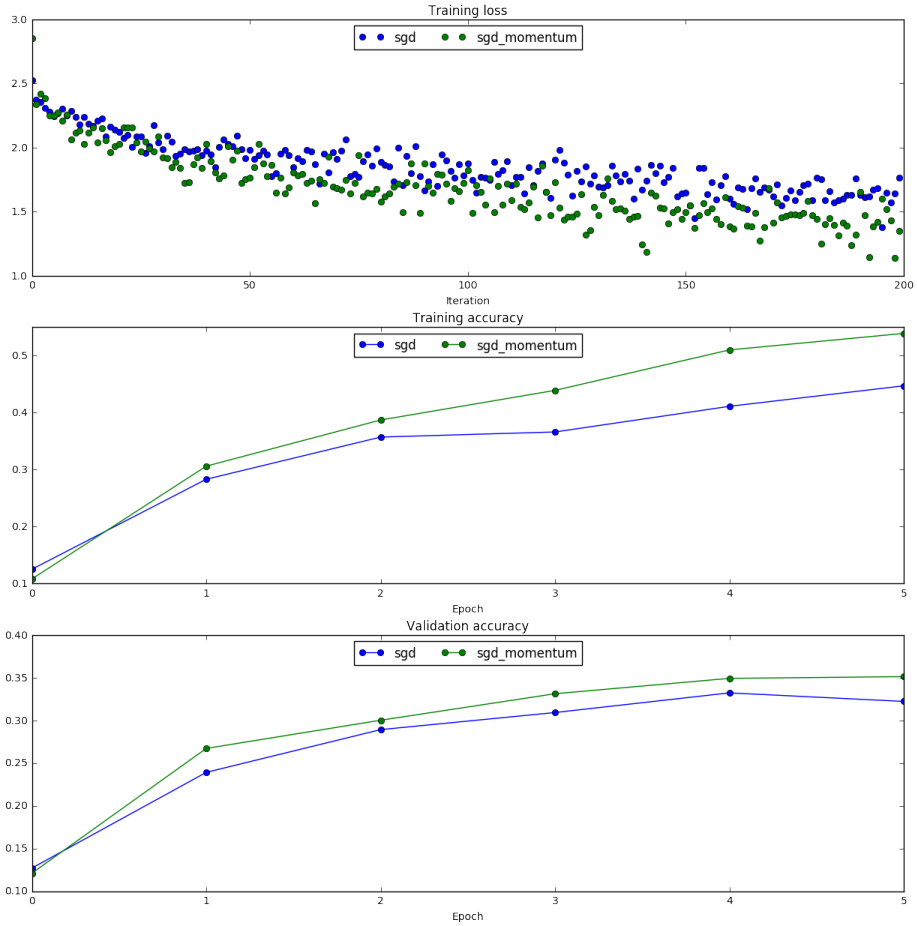
```
Running check with reg =  0
Initial loss:  2.30265446744
theta1 relative error: 8.95e-07
theta1_0 relative error: 9.90e-08
theta2 relative error: 1.35e-05
theta2_0 relative error: 3.93e-09
theta3 relative error: 1.44e-07
theta3_0 relative error: 9.09e-11
Running check with reg =  3.14
Initial loss:  7.08607690982
theta1 relative error: 4.40e-08
theta1_0 relative error: 4.05e-08
theta2 relative error: 4.92e-08
theta2_0 relative error: 4.55e-09
theta3 relative error: 1.00e+00
theta3_0 relative error: 3.69e-10
```

Check implementation:

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

And then we train a six-layer network with both SGD and SGD+momentum and found that SGD+momentum update rule converge much faster. The training and valdation accuracy are shown below:

source	epoch1	epoch2	epoch3	epoch4	epoch5
train-SGD	**0.283000**	**0.357000**	**0.366000**	**0.411000**	**0.447000**
val-SGD	**0.239000**	**0.289000**	**0.309000**	**0.332000**	**0.322000**
train-SGD+momentum	**0.306000**	**0.387000**	**0.439000**	**0.510000**	**0.539000**
val-SGD+momentum	**0.267000**	**0.300000**	**0.331000**	**0.349000**	**0.351000**



Problem 4.1.11: RMSProp

RMSProp update rule:

```
cache = config['decay_rate'] * config['cache'] + (1-config['decay_rate']) * dtheta ** 2
next_theta = theta - config['learning_rate'] * dtheta / (cache ** 0.5 + config['epsilon'])
config['cache'] = cache
```

Test RMSProp implementation:

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

Problem 4.1.12: Adam

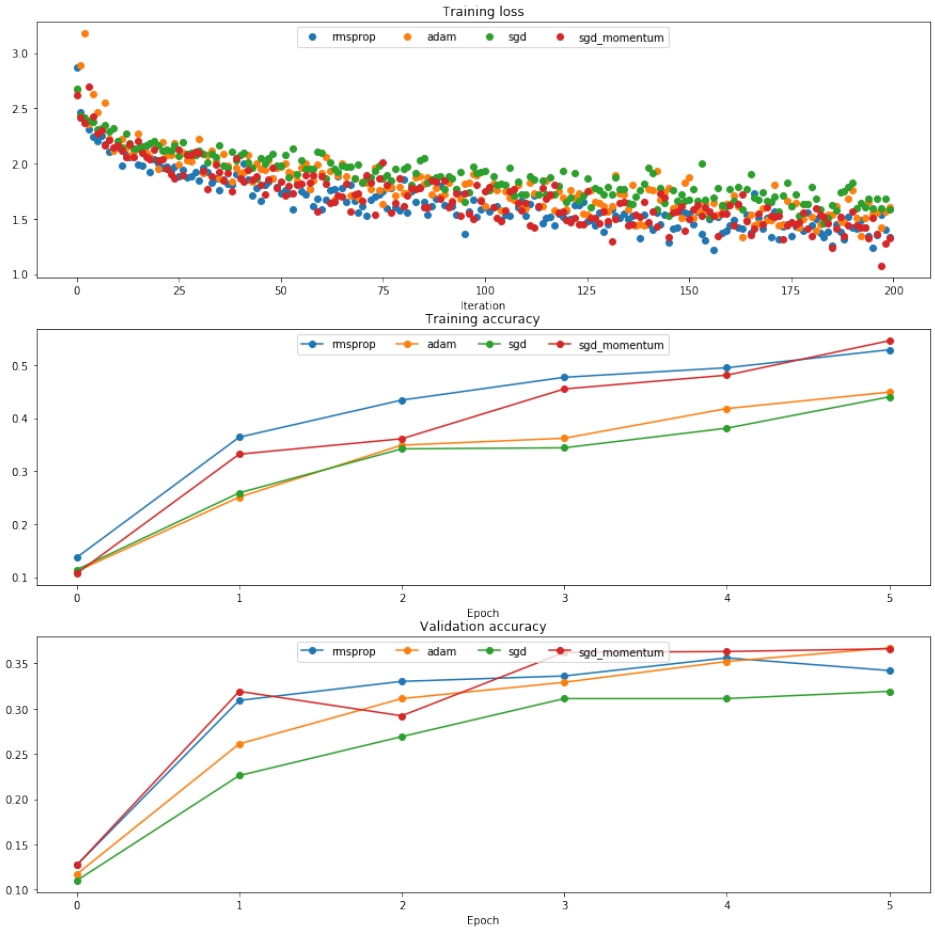
The update rule for Adam:

```
m = config['beta1'] * config['m'] + (1 - config['beta1']) * dtheta
v = config['beta2'] * config['v'] + (1 - config['beta2']) * dtheta ** 2
next_theta = theta - config['learning_rate'] * m / (v ** 0.5 + config['epsilon'])
config['m'] = m
config['v'] = v
```

Test Adam implementation(the hyperparameter 1 is typically chosen to be 0.9, 2 to be 0.999):

```
next_theta error:  0.207207036686
v error:  4.20831403811e-09
m error:  4.21496319311e-09
```

Problem 4.1.13: Comparison between different model



4.2 Dropout

Problem 4.2.1: Dropout forward pass

The function for dropout forward pass is:

```
if mode == 'train':
  mask=(np.random.rand(*x.shape)<(1-p))/(1-p)
  out = x*mask
elif mode == 'test':
  out=x
```

Run the cell in Dropout.ipynb we get:

```
Running tests with p =  0.3
Mean of input:  10.0010463333
Mean of train-time output:  9.98193775775
Mean of test-time output:  10.0010463333
Fraction of train-time output set to zero:  0.301448
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.6
Mean of input:  10.0010463333
Mean of train-time output:  9.98615630633
Mean of test-time output:  10.0010463333
Fraction of train-time output set to zero:  0.6006
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.75
Mean of input:  10.0010463333
Mean of train-time output:  9.99933906056
Mean of test-time output:  10.0010463333
Fraction of train-time output set to zero:  0.75004
Fraction of test-time output set to zero:  0.0
```

Problem 4.2.2: Dropout backward pass

The function for dropout backward pass:

```
if mode == 'train':
  dx = dout*mask
elif mode == 'test':
  dx = dout
```

Test result:

```
dx relative error:  1.89290450953e-11
```

Problem 4.2.3: Fully connected nets with dropout

Modified Implementation:

Forward pass:

```
if self.use_dropout==False:
  self.dropout_param = {'mode': 'train', 'p': 0}
caches={}
temp_i,cache=affine_relu_forward(X,self.params['theta1'],self.params['theta1_0'])
temp_i,drop_cache=dropout_forward(temp_i, self.dropout_param)
caches[0]={'cache':cache, 'drop_cache':drop_cache}

for i in range(1,self.num_layers-2):
  temp_i,cache=affine_relu_forward(temp_i,self.params['theta'+str(i+1)],self.params['theta'+str(i+1)+'_0'])
  temp_i,drop_cache=dropout_forward(temp_i, self.dropout_param )
  caches[i]={'cache':cache, 'drop_cache':drop_cache}

scores,cache=affine_forward(temp_i,self.params['theta'+str(self.num_layers-1)],self.params['theta'+str(self.num_layers-1)+'_0'])
caches[self.num_layers-2]={'cache':cache}
```

backward pass:

```
loss,d_loss=softmax_loss(scores, y)
loss=loss+0.5*self.reg*(np.sum(self.params['theta'+str(self.num_layers-1)]**2))
d_temp,grads['theta'+str(self.num_layers-1)],grads['theta'+str(self.num_layers-1)+'_0']=affine_backward(d_loss,caches[self.num_layers-
for i in range(self.num_layers-2,0,-1):
  loss=loss+0.5*self.reg*(np.sum(self.params['theta'+str(i)]**2))
  d_temp= dropout_backward(d_temp, caches[i-1]['drop_cache'] )
  d_temp,grads['theta'+str(i)],grads['theta'+str(i)+'_0']=affine_relu_backward(d_temp,caches[i-1]['cache'])
  grads['theta'+str(i)]=grads['theta'+str(i)]+self.reg*self.params['theta'+str(i)]
```

Numerically gradient-check implementation:

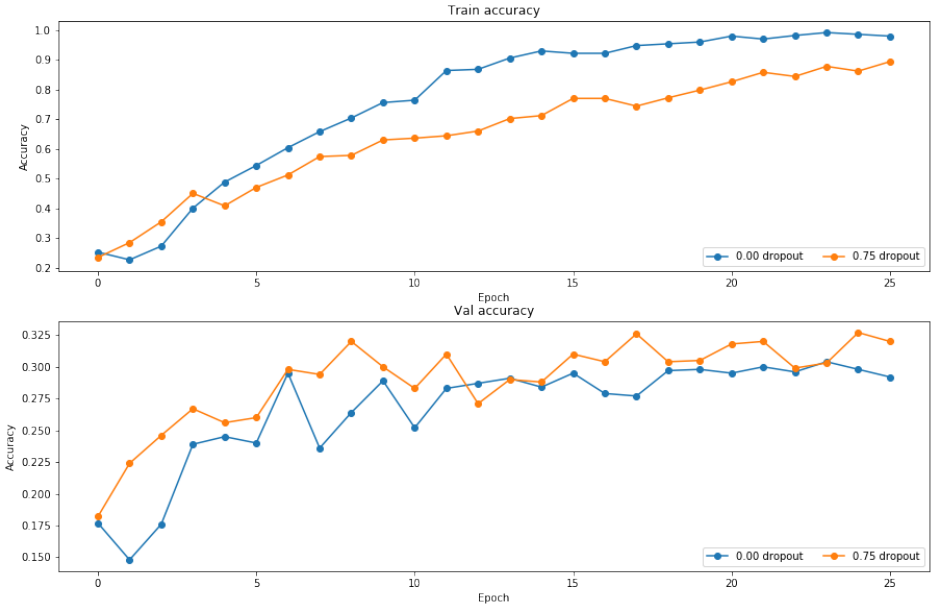
```
Running check with dropout =  0
Initial loss:  2.3051948274
theta1 relative error: 2.53e-07
theta1_0 relative error: 2.94e-06
theta2 relative error: 1.50e-05
theta2_0 relative error: 5.05e-08
theta3 relative error: 2.75e-07
theta3_0 relative error: 1.17e-10

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: Regularization experiment

We can see that with the same parameters(epcohs=25, batch size=100, learning rate=5e-4. optimizer=adam), the network with dropout converge much faster. And the accuracy for two networks after 25 epochs is:

	dropout=0	**dropout=0.75**
train	**0.980000**	**0.894000**
val	**0.292000**	**0.320000**

And learning history is shown below:

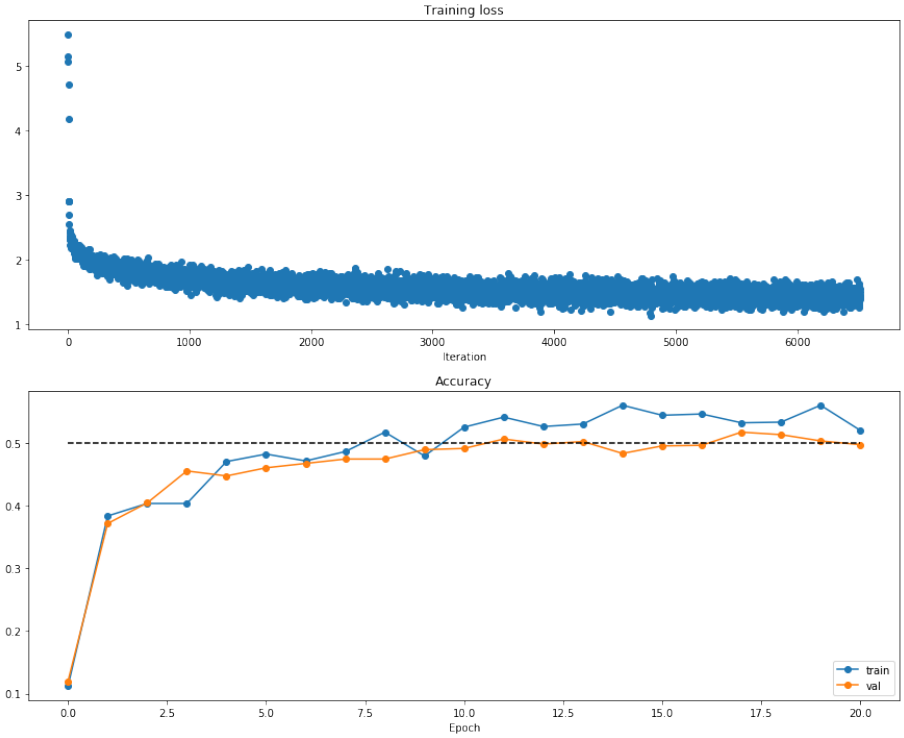


Problem 4.3: Training a fully connected network for the CIFAR-10 dataset with dropout

After trials, we set basic parameters as follows:

optimizer	epochs	learning rate	batch size	weight scale	reg	dropout	topology
Adam	25	1e-3	150	5e-2	1e-4	0.1	[100,100,100,100]

The training history is shown below:



And the accuracy we finally derived is:

```
Validation set accuracy:  0.518
Test set accuracy:  0.503
```

4.4 Convolutional neural networksProblem

4.4.1: Convolution: naive forward pass

```
stride = conv_param['stride']
pad = conv_param['pad']

m,C,H,W = x .shape
F,C,HH,WW = theta.shape


H_1 = 1 + (H + 2 * pad - HH) / stride
W_1 = 1 + (W + 2 * pad - WW) / stride
x_pad = np.pad(x,[(0,0),(0,0),(pad, pad),(pad, pad)], mode = 'constant')
out = np.zeros((m,F,H_1,W_1))

for i in range(m):
  for j in range(F):
    for k in range(H_1):
      for p in range(W_1):
        conv = x_pad[i,:,k * stride:k * stride + HH,p * stride:p * stride + WW]
        out[i,j,k,p] = np.sum(conv * theta[j]) + theta0[j]
```

Test result:

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

Problem 4.4.2: Convolution: naive backward pass

```
x,theta,theta0,conv_param=cache
m,C,H,W=x.shape
F,C,HH,WW=theta.shape

stride=conv_param['stride']
pad=conv_param['pad']

dx = np.zeros_like(x)
dtheta = np.zeros_like(theta)
dtheta0 = np.zeros_like(theta0)

H_1=1+(H+2*pad-HH)/stride
W_1=1+(W+2*pad-WW)/stride

x_pad=np.pad(x,[(0,0),(0,0),(pad, pad),(pad, pad)], mode='constant')
dx_pad=np.pad(dx,[(0,0),(0,0),(pad, pad),(pad, pad)], mode='constant')

for i in range(m):
  for j in range(F):
    for k in range(H_1):
      for p in range(W_1):
        dx_pad[i,:,k*stride:k*stride+HH,p*stride:p*stride+WW] += theta[j,:,:,:]*dout[i,j,k,p]
        dtheta[j,:,:,:] += x_pad[i,:,k*stride:k*stride+HH,p*stride:p*stride+WW]*dout[i,j,k,p]
dtheta0 = np.sum(dout ,axis=(0,2,3))
dx=dx_pad[:, :, pad:pad+H, pad:pad+W]
```

Test implementation:

```
Testing conv_backward_naive function
dx error:  7.10693878012e-09
dtheta error:  2.95776831088e-10
dtheta0 error:  6.38200035324e-12
```

Problem 4.4.3: Max pooling: naive forward pass

The function for forward pass for the max-pooling operation:

```
m,C,H,W=x.shape

stride=pool_param['stride']
pool_height=pool_param['pool_height']
pool_width=pool_param['pool_width']

H_1=1+(H-pool_height)/stride
W_1=1+(W-pool_width)/stride
out=np.zeros((m,C,H_1,W_1))

for i in range(m):
  for j in range(C):
    for k in range(H_1):
      for p in range(W_1):
        out[i,j,k,p] = np.max(x[i,j,k*stride:(k*stride+pool_height),p*stride:(p*stride+pool_width)])
```

Check implementation:

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

Problem 4.4.4: Max pooling: naive backward pass

The function for backward pass for the max-pooling operation:

```
x,pool_param=cache
m,C,H,W=x.shape

stride=pool_param['stride']
pool_height=pool_param['pool_height']
pool_width=pool_param['pool_width']
dx=np.zeros_like(x)

H_1=1+(H-pool_height)/stride
W_1=1+(W-pool_width)/stride
out=np.zeros((m,C,H_1,W_1))

for i in range(m):
  for j in range(C):
    for k in range(H_1):
      for p in range(W_1):
        matrix=x[i,j,k*stride:(k*stride+pool_height),p*stride:(p*stride+pool_width)]
        max_matrix=(np.max(matrix)==matrix)
        dx[i,j,k*stride:(k*stride+pool_height),p*stride:(p*stride+pool_width)]+=dout[i,j,k,p]*max_matrix
```

Check implementation:

```
Testing max_pool_backward_naive function:
dx error:  3.27562894085e-12
```

Problem 4.4.5: Three layer convolutional neural network

The detailed code of this three-layer CNN in is cnn.py and we will run the following things to help us debug.

•Testing the CNN: loss computation

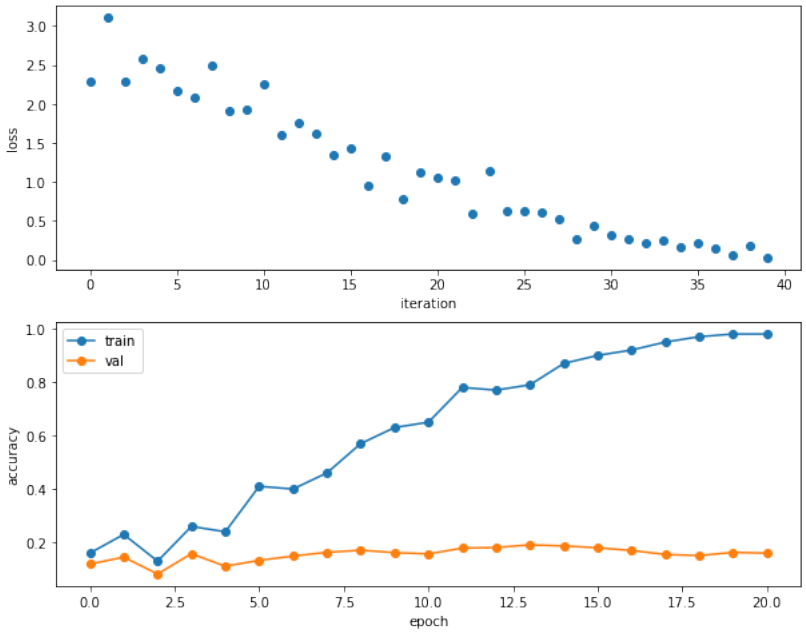
```
Initial loss (no regularization):  2.30258535358
Initial loss (with regularization):  2.33486022565
```

•Testing the CNN: gradient check

```
theta1 max relative error: 3.315313e-03
theta1_0 max relative error: 4.918253e-05
theta2 max relative error: 7.711371e-03
theta2_0 max relative error: 2.044220e-07
theta3 max relative error: 2.391497e-04
theta3_0 max relative error: 1.695955e-09
```

•Testing the CNN: overfit small data

The learning history can be plot below when we use the parameters given(epochs=20, batch size=50, learning rate=1e-3, weight scale=1e-2 optimizer='adam'):



We found that the network results in very high training accuracy and comparatively low validation accuracy, indicating the problem of overfitting.

Problem 4.4.6: Train this 3-layer CNN on the CIFAR-10 data

We set basic learning parameters as follows:

ooptimizer	epochs	learning rate	batch size	hidden dim	weight scale	reg	filter number	filter size
adam	20	1e-3	50	500	1e-2	1e-4	8	5

We can achieve >50% accuracy after 20 epoch:

```
(Epoch 20 / 20) train acc: 0.535000; val_acc: 0.506000
```

•Visualize Filters

We can visualize the first layer convolutional filters as following: