之前的作业中，我们已经完成了一个两层的全连接神经网络的设计，但是有些简单，并且还没有模块化，因为那里的损失函数和梯度我们是用一个函数来计算出来的。因此，我们希望可以设计更复杂的网络，以便于我们可以完成不同类型层的设计，然后将它们集成到不同结构的模型中。

主要内容如下：

注意，这里我们提到的层数，是不将输入层算入的。

1 基础层

在`layers.py`文件中，定义了我们接下来所需要的层，我们来逐个进行分析。

1.1 全连接层

1.1.1 前向传播

前向传播的计算类似这样：

```
def layer_forward(x,w):
    z=   #中间值
    out=  #输出值
    cache=(x,w,z,out)  #反向传播计算梯度时需要的参数
    return out cache
```

代码如下：

```
def affine_forward(x,w,b):

    '''x:(N,d_1,…,d_k) x_rsp中我们会将x转化为(N,D)，其中D=d_1*…*d_k;  w:(D,M);   b:(M,);   out:(N,M);    cache:(x,w,b)'''

    out=None
    N=x.shape[0]
    x_rsp=x.reshape(N,-1)
    out=x_rsp.dot(w)+b
    cache=(x,w,b)

    return out,cache
```

我们可以检验下计算的误差，检验代码如下：

```
num_inputs=2
input_shape=(4,5,6)
output_dim=3
input_size=num_inputs*np.prod(input_shape)
weight_size=output_dim*np.prod(input_shape)

x=np.linspace(-0.1,0.5,num=input_size).reshape(num_inputs,*input_shape)
w=np.linspace(-0.2,0.3,num=weight_size).reshape(np.prod(input_shape),output_dim)
b=np.linspace(-0.3,0.1,num=output_dim)

out,_=affine_forward(x,w,b)
correct_out=np.array([[ 1.49834967,  1.70660132,  1.91485297],
                      [ 3.25553199,  3.5141327,   3.77273342]])

print('testing affine_forward function:')
```

```
print('difference:',rel_error(out,correct_out))
```

输出结果如下：

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

误差竟然达到e-10，还是挺不错的。接下来我们就继续计算反向传播。

1.1.2 反向传播

反向传播接收来自上一层输出的梯度值和 `cache` 值，返回输入和权重的梯度，形式如下：

```
def layer_backward(dout,cache):
    x,w,z,out=cache
    dx=
    dw=
    return dx dw
```

实现代码如下：

```
def affine_backward(dout,cache):
    '''dout:(N,M);   x:(N,d_1,…d_k) x_rsp中我们会将x转化为(N,D)，其中D=d_1*…*d_k;   w:(D,M);   dx:(D,d_1,…d_k);   dw:(D,M);   db:(M,)'''
    x,w,b=cache
    dx,dw,db=None,None,None
    N=x.shape[0]
    x_rsp=x.reshape(N,-1)
    dx=dout.dot(w.T)
    dx=dx.reshape(*x.shape)
    dw=x_rsp.T.dot(dout)
    db=np.sum(dout,axis=0)
    return dx,dw,db
```

注：如果不清楚反向传播过程中如何利用利用链式法则进行计算，可以参阅我们上一次的作业
同样，下面我们来检验下：

```
x=np.random.randn(10,2,3)
```

```
w=np.random.randn(6,5)
b=np.random.randn(5)
dout=np.random.randn(10,5)

dx_num=eval_numerical_gradient_array(lambda x:affine_forward(x,w,b)[0],x,dout)
dw_num=eval_numerical_gradient_array(lambda w:affine_forward(x,w,b)[0],w,dout)
db_num=eval_numerical_gradient_array(lambda b:affine_forward(x,w,b)[0],b,dout)

_,cache=affine_forward(x,w,b)
dx,dw,db=affine_backward(dout,cache)

print('test affine_backward function:')
print('dx error:',rel_error(dx_num,dx))
print('dw error:',rel_error(dw_num,dw))
print('db error:',rel_error(db_num,db))
```

输出结果如下：

```
test affine_backward function:
dx error: 2.50214010994e-10
dw error: 9.11372672216e-11
db error: 3.57180037616e-11
```

bingo，误差还是很小的。这样我们就完成了一层映射层的前向和反向传播两个过程。

1.2 Relu层

1.2.1 前向传播

代码如下：

```
def relu_forward(x):
    out=None
    out=x*(x>=0)
    cache=x
    return out,cache
```

检验代码如下：

```
x=np.linspace(-0.5,0.5,num=12).reshape(3,4)
out,_=relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,
   ],
                        [ 0.,          0.,          0.04545455,
 0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,
  ]])
print('testing relu_forward function:')
print('difference:',rel_error(out,correct_out))
```

输出结果如下：

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

1.2.2 反向传播

代码如下：

```
def relu_backward(dout,cache):
    dx,x=None,cache
    dx=(x>=0)*dout
    return dx
```

检验代码如下：

```
x=np.random.randn(10,10)
dout=np.random.randn(*x.shape)

dx_num=eval_numerical_gradient_array(lambda x:relu_forward(x)[0],x,dout)
_,cache=relu_forward(x)
dx=relu_backward(dout,cache)

print('testing relu_backward functin:')
print('dx error:',rel_error(dx_num,dx))
```

输出结果如下：

```
testing relu_backward functin:
```

```
dx error: 3.27561649471e-12
```

到现在，我们差不多已经完成了基础层的构建。但是，实践中，一般映射层后面会接Relu层，我们这里把两层合为一层来看看：

```
def affine_relu_forward(x,w,b):
    a,fc_cache=affine_forward(x,w,b)
    out,relu_cache=relu_forward(a)
    cache=(fc_cache,relu_cache)
    return out,cache

def affine_relu_backward(dout,cache):
    fc_cache,relu_cache=cache
    da=relu_backward(dout,relu_cache)
    dx,dw,db=affine_backward(da,fc_cache)
    return dx,dw,db
```

梯度检验代码如下：

```
x=np.random.randn(2,3,4)
w=np.random.randn(12,10)
b=np.random.randn(10)
dout=np.random.randn(2,10)

out,cache=affine_relu_forward(x,w,b)
dx,dw,db=affine_relu_backward(dout,cache)

dx_num=eval_numerical_gradient_array(lambda x:affine_relu_forward(x,w,b)[0],x,dout)
dw_num=eval_numerical_gradient_array(lambda w:affine_relu_forward(x,w,b)[0],w,dout)
db_num=eval_numerical_gradient_array(lambda b:affine_relu_forward(x,w,b)[0],b,dout)

print('testing affine_relu_forward:')
print('dx_error:',rel_error(dx_num,dx))
print('dw error:',rel_error(dw_num,dw))
print('db error:',rel_error(db_num,db))
```

运行结果如下：

```
testing affine_relu_forward:
```

```
dx_error: 1.58523685005e-10
dw error: 3.68009862446e-10
db error: 1.04868325611e-10
```

1.3 损失函数层

在上一个作业中，我们已经实现了svm和softmax线性分类器，现在我们来实现了svm和softmax层。

代码如下：

```
def svm_loss(x,y):
    ''':param x: (N,C)，x[i,j]表示第i个输入是第j个的分数   :param y:(N,)，x[i]的标签    :return:loss，dx'''
    N=x.shape[0]
    correct_class_scores=x[np.arange(N),y]
    margins=np.maximum(0,x-correct_class_scores[:,np.newaxis]+1.0)
    margins[np.arange(N),y]=0
    loss=np.sum(margins)/N

    num_pos=np.sum(margins>0,axis=1)
    dx=np.zeros_like(x)
    dx[margins>0]=1
    dx[np.arange(N),y]-=num_pos
    dx/=N
    return loss,dx

def softmax_loss(x,y):
    probs=np.exp(x-np.max(x,axis=1,keepdims=True))
    probs/=np.sum(probs,axis=1,keepdims=True)
    N=x.shape[0]
    loss=-np.sum(np.log(probs[np.arange(N),y]))/N
    dx=probs.copy()
    dx[np.arange(N),y]-=1
    dx/=N
    return loss,dx
```

我们检验下我们做的是不是正确，代码如下：

```
num_classes,num_inputs=10,50
x=0.001*np.random.randn(num_inputs,num_classes)
y=np.random.randint(num_classes,size=num_inputs)

```

```
dx_num=eval_numerical_gradient(lambda x:svm_loss(x,y)[0],x,verbose=False)
loss,dx=svm_loss(x,y)
print('testing svm_loss:')
print('loss:',loss)
print('dx error:',rel_error(dx_num,dx))

dx_num=eval_numerical_gradient(lambda x:softmax_loss(x,y)
[0],x,verbose=False)
loss,dx=softmax_loss(x,y)

print('\ntesting softmax_loss:')
print('loss:',loss)
print('dx error:',rel_error(dx_num,dx))
```

运行结果如下：

```
testing svm_loss:
loss: 9.00034570116
dx error: 1.40215660067e-09

testing softmax_loss:
loss: 2.30262010084
dx error: 8.19589239336e-09
```

OK，现在我们已经完成了基础层的构建，下面我们就可以通过这些层的连接来组成我们想要的结构。

2 两层神经网络

下面我们就使用我们原来构建的基础层来搭建一个两层的神经网络。
我们定义了一个 `TwoLayerNet` 代的类来实现一个两层神经网络的搭建，代码如下：

```
class TwoLayerNet(object):
    def
__init__(self,input_dim=3*32*32,hidden_dim=100,num_classes=10,weight_scal
e=1e-3,reg=0.0):
        self.params={}
        self.reg=reg

 self.params['W1']=weight_scale*np.random.rand(input_dim,hidden_dim)
        self.params['b1']=np.zeros(hidden_dim)
```

```
 self.params['W2']=weight_scale*np.random.randn(hidden_dim,num_classes)
        self.params['b2']=np.zeros(num_classes)

    def loss(self,X,y=None):
        scores=None

 ar1_out,ar1_cache=affine_relu_forward(X,self.params['W1'],self.params['b1'])

 a2_out,a2_cache=affine_forward(ar1_out,self.params['W2'],self.params['b2'])
        scores=a2_out

        if y is None:
            return scores

        loss,grads=0,{}

        loss,dscores=softmax_loss(scores,y)

 loss=loss+0.5*self.reg*np.sum(self.params['W1']*self.params['W1'])+0.5*self.reg*np.sum(self.params['W2']*self.params['W2'])
        dx2,dw2,db2=affine_backward(scores,a2_cache)
        grads['W2']=dw2+self.reg*self.params['W2']
        grads['b2']=db2

        dx1,dw1,db1=affine_relu_forward(dx2,ar1_cache)
        grads['W1']=dw1+self.reg*self.params['W1']
        grads['b1']=db1

        return loss,grads
```

基本就同前面的内容，这里我们就不对代码进行解释了。
同样，我们来检验下我们搭建的网络。代码如下：

```
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-2
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print ('Testing initialization … ')
```

```
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print ('Testing test-time forward pass … ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434,
15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128,
15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822,
15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print ('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time
loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization
loss'

for reg in [0.0, 0.7]:
  print ('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)
```

输出结果如下：

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 1.22e-08
W2 relative error: 3.35e-10
b1 relative error: 8.37e-09
b2 relative error: 2.53e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.53e-07
W2 relative error: 7.98e-08
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

我们不使用正则化系数和正则化系数为0.7，参数的误差都是很小的，所以我们在前面基础层的基础之上组合的两层神经网络还是不错的。

3 Solver

在之前的作业中，训练模型的logic是和模型在一起的，接下来我们将把它单独作为一个 `Solver` 类，并对上一节的两层神经网络进行训练。
在Keras框架里面这个应该是叫 `compile` 。
哈哈，代码有点多，这里我就不贴出来了（毕竟这是大神造好的轮子），只是给大家简单介绍下这个代码的主要功能，更详细的介绍可以详见博客。

该类用于接收数据与标签，对权值进行相应求解，我们可以在这个类中调整一些超参数以达到较好的训练效果。

下面我们就边训练边对这个类进行解释。
我们来看下训练代码：

```
data={}
data=
{'X_train':X_train,'y_train':y_train,'X_val':X_val,'y_val':y_val,'X_test'
:X_test,'y_test':y_test}
model=TwoLayerNet(reg=1e-1)
solver=None

solver=Solver(model,data,update_rule='sgd',
              optim_config={'learning_rate':1e-3},
              lr_decay=0.8,num_epochs=10,batch_size=100,print_every=100)
solver.train()
```

```
scores=model.loss(data['X_test'])
y_pred=np.argmax(scores,axis=1)
acc=np.mean(y_pred==data['y_test'])
print('test acc: %f' % acc)
```

代码解释：这里我们就看下 `Solver` 类所需要的参数。 `model` 是我们定义的模型； `data` 是我们的数据（x需要是(N_train, d_1, …, d_k) 数组的形式，y需要是(N_train,)）； `update_relu` 是我们的所选择的优化器（如sgd、adam等）； `optim_config` 所需要的参数（sgd只需要学习率这一个参数）； `lr_decay` 学习率衰减，负责在每个epoch之后更新学习率； `num_epochs` 轮数，也就是我们的数据需要被训练多少次； `batch_size` 批尺寸，也就是我们每次需要从训练集中选取多少数据进行训练； `print_every` 每n次迭代，打印损失函数值。

我们这里解释下迭代次数这个概念，假设我们有49000个数据， `batch_size=100` ，那么这一轮的迭代次数就是49000/100=490次，而我们的 `num_epochs=10` ，所以总的迭代次数就是490×10=49000

我们看下输出结果，是不是和我们分析的是一样的

```
(Iteration 1 / 4900) loss: 2.318766
(Epoch 0 / 10) train acc: 0.111000; val_acc: 0.122000
(Iteration 101 / 4900) loss: 1.736319
(Iteration 201 / 4900) loss: 1.761152
(Iteration 301 / 4900) loss: 1.580906
(Iteration 401 / 4900) loss: 1.779731
(Epoch 1 / 10) train acc: 0.473000; val_acc: 0.456000
(Iteration 501 / 4900) loss: 1.633682
(Iteration 601 / 4900) loss: 1.375359
(Iteration 701 / 4900) loss: 1.574423
(Iteration 801 / 4900) loss: 1.643599
(Iteration 901 / 4900) loss: 1.484345
(Epoch 2 / 10) train acc: 0.478000; val_acc: 0.479000
(Iteration 1001 / 4900) loss: 1.517680
(Iteration 1101 / 4900) loss: 1.571929
(Iteration 1201 / 4900) loss: 1.315616
(Iteration 1301 / 4900) loss: 1.367084
(Iteration 1401 / 4900) loss: 1.478173
(Epoch 3 / 10) train acc: 0.513000; val_acc: 0.491000
(Iteration 1501 / 4900) loss: 1.525170
(Iteration 1601 / 4900) loss: 1.507696
(Iteration 1701 / 4900) loss: 1.310258
(Iteration 1801 / 4900) loss: 1.307820
(Iteration 1901 / 4900) loss: 1.420765
(Epoch 4 / 10) train acc: 0.562000; val_acc: 0.492000
```

```
(Iteration 2001 / 4900) loss: 1.346618
(Iteration 2101 / 4900) loss: 1.368148
(Iteration 2201 / 4900) loss: 1.273601
(Iteration 2301 / 4900) loss: 1.255306
(Iteration 2401 / 4900) loss: 1.434615
(Epoch 5 / 10) train acc: 0.563000; val_acc: 0.503000
(Iteration 2501 / 4900) loss: 1.320840
(Iteration 2601 / 4900) loss: 1.451932
(Iteration 2701 / 4900) loss: 1.319160
(Iteration 2801 / 4900) loss: 1.269273
(Iteration 2901 / 4900) loss: 1.351908
(Epoch 6 / 10) train acc: 0.543000; val_acc: 0.513000
(Iteration 3001 / 4900) loss: 1.277055
(Iteration 3101 / 4900) loss: 1.254344
(Iteration 3201 / 4900) loss: 1.386045
(Iteration 3301 / 4900) loss: 1.359556
(Iteration 3401 / 4900) loss: 1.220027
(Epoch 7 / 10) train acc: 0.579000; val_acc: 0.542000
(Iteration 3501 / 4900) loss: 1.334799
(Iteration 3601 / 4900) loss: 1.099247
(Iteration 3701 / 4900) loss: 1.287256
(Iteration 3801 / 4900) loss: 1.196124
(Iteration 3901 / 4900) loss: 1.221174
(Epoch 8 / 10) train acc: 0.604000; val_acc: 0.526000
(Iteration 4001 / 4900) loss: 1.317148
(Iteration 4101 / 4900) loss: 1.267705
(Iteration 4201 / 4900) loss: 1.209211
(Iteration 4301 / 4900) loss: 1.043793
(Iteration 4401 / 4900) loss: 1.346820
(Epoch 9 / 10) train acc: 0.605000; val_acc: 0.523000
(Iteration 4501 / 4900) loss: 1.176413
(Iteration 4601 / 4900) loss: 1.316058
(Iteration 4701 / 4900) loss: 1.181584
(Iteration 4801 / 4900) loss: 1.182825
(Epoch 10 / 10) train acc: 0.601000; val_acc: 0.522000
test acc: 0.522000
```

每`print_every`次输出我们的loss值，每轮输出我们的准确率，最后得到我们在测试集上的准确率：0.522。

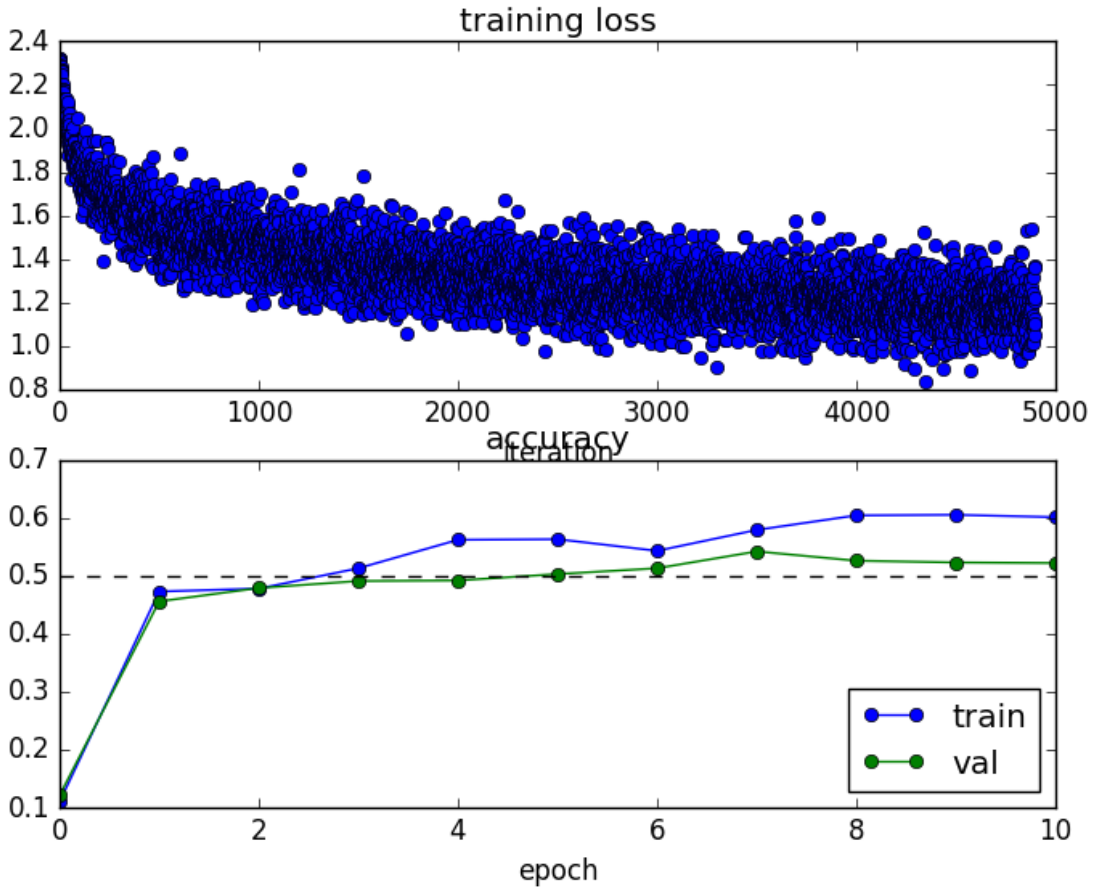
就是这么简单，如果你熟悉深度学习的一种框架的话，其实会觉得很容易的。

我们来可视化下我们的过程（loss值和acc），代码如下：

```
plt.subplot(2,1,1)
```

```
plt.title('training loss')
plt.plot(solver.loss_history,'o')
plt.xlabel('iteration')

plt.subplot(2,1,2)
plt.title('accuracy')
plt.plot(solver.train_acc_history,'-o',label='train')
plt.plot(solver.val_acc_history,'-o',label='val')
plt.plot([0.5]*len(solver.val_acc_history),'k--')
plt.xlabel('epoch')
plt.legend(loc='lower right')#图例位置
plt.gcf().set_size_inches(15,12)
plt.show()
```

运行结果如下：



为了让我们的网络可以多增加几层，我们将这里尝试不同的优化器，当然下次作业中我们可以尝试Dropout和Batch Normalization。

4 多层神经网络

上面的网络只有2层，如果我们要实现4层，甚至是5层、6层呢？我们来定义一个多层的神经网络，代码如下：

```
class FullyConnectedNet(object):

    #A fully-connected neural network with an arbitrary number of hidden layers,
    #ReLU nonlinearities, and a softmax loss function. This will also implement
    #dropout and batch normalization as options. For a network with L layers,
    #the architecture will be

    #{affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

    #where batch normalization and dropout are optional, and the {…} block is
    #repeated L - 1 times.

    #Similar to the TwoLayerNet above, learnable parameters are stored in the
    #self.params dictionary and will be learned using the Solver class.


    def __init__(self, hidden_dims, input_dim=3 * 32 * 32, num_classes=10,
                 dropout=0, use_batchnorm=False, reg=0.0,
                 weight_scale=1e-2, dtype=np.float32, seed=None):

        #Initialize a new FullyConnectedNet.

        #Inputs:
        #- hidden_dims: A list of integers giving the size of each hidden layer.
        #- input_dim: An integer giving the size of the input.
        #- num_classes: An integer giving the number of classes to classify.
        #- dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
         # the network should not use dropout at all.
        #- use_batchnorm: Whether or not the network should use batch normalization.
        #- reg: Scalar giving L2 regularization strength.
        #- weight_scale: Scalar giving the standard deviation for random
         # initialization of the weights.
        #- dtype: A numpy datatype object; all computations will be performed using
```

```
        # this datatype. float32 is faster but less accurate, so you
should use
        # float64 for numeric gradient checking.
      #- seed: If not None, then pass this random seed to the dropout
layers. This
        # will make the dropout layers deteriminstic so we can gradient
check the
        # model.

      self.use_batchnorm = use_batchnorm
      self.use_dropout = dropout > 0
      self.reg = reg
      self.num_layers = 1 + len(hidden_dims)
      self.dtype = dtype
      self.params = {}


 ############################################################################
#####
      # TODO: Initialize the parameters of the network, storing all
values in    #
      # the self.params dictionary. Store weights and biases for the
first layer #
      # in W1 and b1; for the second layer use W2 and b2, etc. Weights
should be #
      # initialized from a normal distribution with standard deviation
equal to  #
      # weight_scale and biases should be initialized to zero.
        #
      #
         #
      # When using batch normalization, store scale and shift
parameters for the #
      # first layer in gamma1 and beta1; for the second layer use
gamma2 and     #
      # beta2, etc. Scale parameters should be initialized to one and
shift      #
      # parameters should be initialized to zero.
         #

 ############################################################################
#####
      layer_input_dim = input_dim
      for i, hd in enumerate(hidden_dims):
          self.params['W%d' % (i + 1)] = weight_scale *
np.random.randn(layer_input_dim, hd)
```

```
            self.params['b%d' % (i + 1)] = weight_scale * np.zeros(hd)
            if self.use_batchnorm:
                self.params['gamma%d' % (i + 1)] = np.ones(hd)
                self.params['beta%d' % (i + 1)] = np.zeros(hd)
            layer_input_dim = hd
        self.params['W%d' % (self.num_layers)] = weight_scale *
np.random.randn(layer_input_dim, num_classes)
        self.params['b%d' % (self.num_layers)] = weight_scale *
np.zeros(num_classes)
        # pass

 ##############################################################################
#####
        #                            END OF YOUR CODE
          #

 ##############################################################################
#####

        # When using dropout we need to pass a dropout_param dictionary
to each
        # dropout layer so that the layer knows the dropout probability
and the mode
        # (train / test). You can pass the same dropout_param to each
dropout layer.
        self.dropout_param = {}
        if self.use_dropout:
            self.dropout_param = {'mode': 'train', 'p': dropout}
            if seed is not None:
                self.dropout_param['seed'] = seed

        # With batch normalization we need to keep track of running
means and
        # variances, so we need to pass a special bn_param object to
each batch
        # normalization layer. You should pass self.bn_params[0] to the
forward pass
        # of the first batch normalization layer, self.bn_params[1] to
the forward
        # pass of the second batch normalization layer, etc.
        self.bn_params = []
        if self.use_batchnorm:
            self.bn_params = [{'mode': 'train'} for i in
range(self.num_layers - 1)]

        # Cast all parameters to the correct datatype
```

```
        for k, v in self.params.iteritems():
            self.params[k] = v.astype(dtype)

    def loss(self, X, y=None):

        #Compute loss and gradient for the fully-connected net.
        #Input / output: Same as TwoLayerNet above.

        X = X.astype(self.dtype)
        mode = 'test' if y is None else 'train'

        # Set train/test mode for batchnorm params and dropout param
since they
        # behave differently during training and testing.
        if self.dropout_param is not None:
            self.dropout_param['mode'] = mode
        if self.use_batchnorm:
            for bn_param in self.bn_params:
                bn_param['mode'] = mode

        scores = None

 ############################################################################
#####
        # TODO: Implement the forward pass for the fully-connected net,
computing  #
        # the class scores for X and storing them in the scores
variable.          #
        #
           #
        # When using dropout, you'll need to pass self.dropout_param to
each       #
        # dropout forward pass.
           #
        #
           #
        # When using batch normalization, you'll need to pass
self.bn_params[0] to #
        # the forward pass for the first batch normalization layer, pass
          #
        # self.bn_params[1] to the forward pass for the second batch
normalization #
        # layer, etc.
           #
```

```
124
 ############################################################################
#####
125         layer_input = X
126         ar_cache = {}
127         dp_cache = {}
128
129         for lay in range(self.num_layers - 1):
130             if self.use_batchnorm:
131                 layer_input, ar_cache[lay] =
affine_bn_relu_forward(layer_input,
132
 self.params['W%d' % (lay + 1)],
133
 self.params['b%d' % (lay + 1)],
134
 self.params['gamma%d' % (lay + 1)],
135
 self.params['beta%d' % (lay + 1)],
136
 self.bn_params[lay])
137             else:
138                 layer_input, ar_cache[lay] =
affine_relu_forward(layer_input, self.params['W%d' % (lay + 1)],
139
self.params['b%d' % (lay + 1)])
140
141             if self.use_dropout:
142                 layer_input, dp_cache[lay] =
dropout_forward(layer_input, self.dropout_param)
143
144         ar_out, ar_cache[self.num_layers] = affine_forward(layer_input,
self.params['W%d' % (self.num_layers)],
145
self.params['b%d' % (self.num_layers)])
146         scores = ar_out
147         # pass
148
 ############################################################################
#####
149         #                             END OF YOUR CODE
           #
150
 ############################################################################
#####
151
```

```
        # If test mode return early
        if mode == 'test':
            return scores

        loss, grads = 0.0, {}

 ############################################################################
#####
        # TODO: Implement the backward pass for the fully-connected net.
Store the #
        # loss in the loss variable and gradients in the grads
dictionary. Compute #
        # data loss using softmax, and make sure that grads[k] holds the
gradients #
        # for self.params[k]. Don't forget to add L2 regularization!
          #
        #
           #
        # When using batch normalization, you don't need to regularize
the scale   #
        # and shift parameters.
           #
        #
           #
        # NOTE: To ensure that your implementation matches ours and you
pass the   #
        # automated tests, make sure that your L2 regularization
includes a factor #
        # of 0.5 to simplify the expression for the gradient.
           #

 ############################################################################
#####
        loss, dscores = softmax_loss(scores, y)
        dhout = dscores
        loss = loss + 0.5 * self.reg * np.sum(
            self.params['W%d' % (self.num_layers)] * self.params['W%d' %
(self.num_layers)])
        dx, dw, db = affine_backward(dhout, ar_cache[self.num_layers])
        grads['W%d' % (self.num_layers)] = dw + self.reg *
self.params['W%d' % (self.num_layers)]
        grads['b%d' % (self.num_layers)] = db
        dhout = dx
        for idx in range(self.num_layers - 1):
            lay = self.num_layers - 1 - idx - 1
```

```
            loss = loss + 0.5 * self.reg * np.sum(self.params['W%d' % (lay + 1)] * self.params['W%d' % (lay + 1)])
            if self.use_dropout:
                dhout = dropout_backward(dhout, dp_cache[lay])
            if self.use_batchnorm:
                dx, dw, db, dgamma, dbeta = affine_bn_relu_backward(dhout, ar_cache[lay])
            else:
                dx, dw, db = affine_relu_backward(dhout, ar_cache[lay])
            grads['W%d' % (lay + 1)] = dw + self.reg * self.params['W%d' % (lay + 1)]
            grads['b%d' % (lay + 1)] = db
            if self.use_batchnorm:
                grads['gamma%d' % (lay + 1)] = dgamma
                grads['beta%d' % (lay + 1)] = dbeta
            dhout = dx
        # pass

        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        return loss, grads
```

注意：我们这里其实也引入了Dropout层和BN层，但是本次作业可以先将其忽略，我们会在后面两次作业中分开叙述。

下面我们先对我们构建的网络进行梯度检验，代码如下：

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print ('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print ('Initial loss: ', loss)

```

```
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print ('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

运行结果如下：

```
Running check with reg = 0
Initial loss: 2.30497257983
W1 relative error: 1.54e-06
W2 relative error: 2.86e-07
W3 relative error: 2.27e-06
b1 relative error: 4.41e-08
b2 relative error: 1.33e-09
b3 relative error: 1.57e-10
Running check with reg = 3.14
Initial loss: 6.93678451752
W1 relative error: 3.74e-08
W2 relative error: 2.29e-06
W3 relative error: 1.87e-08
b1 relative error: 2.71e-08
b2 relative error: 6.51e-09
b3 relative error: 2.65e-10
```

误差不大，很好。

多层网络模型已经构建完成，下面我们可以按照和两层网络同样的方式对多层网络进行训练和预测了。

4.1 多层网络训练

我们先使用50个训练集来训练一个三层的神经网络，代码如下：

```
num_train=50
small_data={
    'X_train':data['X_train'][:num_train],
    'y_train':data['y_train'][:num_train],
    'X_val':data['X_val'],
    'y_val':data['y_val'],
}

```

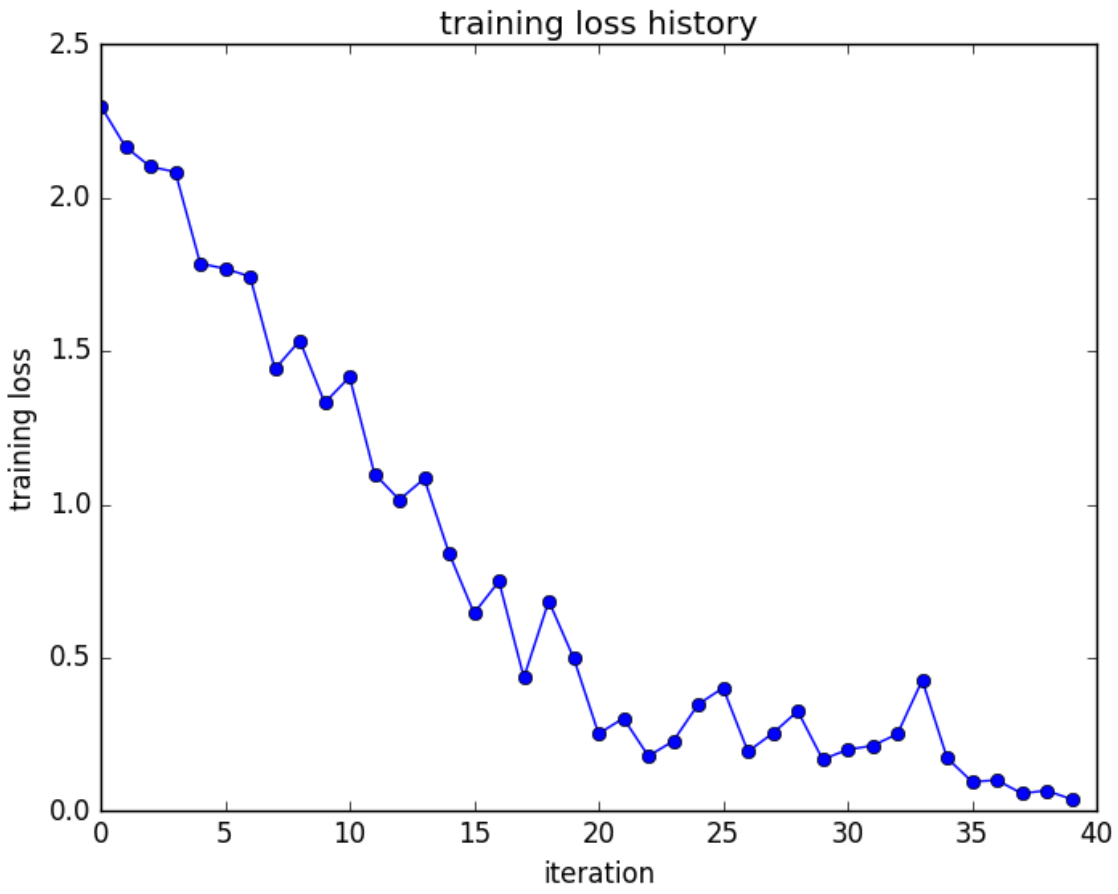
```
weight_scale=1e-2
learning_rate=8e-3
model=FullyConnectedNet([100,100],weight_scale=weight_scale,dtype=np.float64)
solver=Solver(model,small_data,print_every=10,num_epochs=20,batch_size=25,update_rule='sgd',optim_config={'learning_rate':learning_rate,})
solver.train()

plt.plot(solver.loss_history,'-o')
plt.title('training loss history')
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.show()
```

我们可以看到，多层网络通过接收隐含层的神经元个数来确定层数的，比如[100,100]表示有两个隐含层，每个隐含层的神经元个数为100。

输出结果如下：

```
(Iteration 1 / 40) loss: 2.300085
(Epoch 0 / 20) train acc: 0.300000; val_acc: 0.149000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.162000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.155000
(Epoch 3 / 20) train acc: 0.480000; val_acc: 0.167000
(Epoch 4 / 20) train acc: 0.460000; val_acc: 0.168000
(Epoch 5 / 20) train acc: 0.540000; val_acc: 0.195000
(Iteration 11 / 40) loss: 1.415692
(Epoch 6 / 20) train acc: 0.780000; val_acc: 0.197000
(Epoch 7 / 20) train acc: 0.840000; val_acc: 0.195000
(Epoch 8 / 20) train acc: 0.760000; val_acc: 0.177000
(Epoch 9 / 20) train acc: 0.880000; val_acc: 0.208000
(Epoch 10 / 20) train acc: 0.900000; val_acc: 0.202000
(Iteration 21 / 40) loss: 0.252448
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.196000
(Epoch 12 / 20) train acc: 0.940000; val_acc: 0.189000
(Epoch 13 / 20) train acc: 0.880000; val_acc: 0.193000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.188000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.192000
(Iteration 31 / 40) loss: 0.200646
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.202000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.193000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.190000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.196000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.204000
```

我们得到的loss曲线图如下：



上面损失函数有些震荡，可能是学习率有点大。

如果五层神经网络呢？效果会不会好些呢？我们来看下

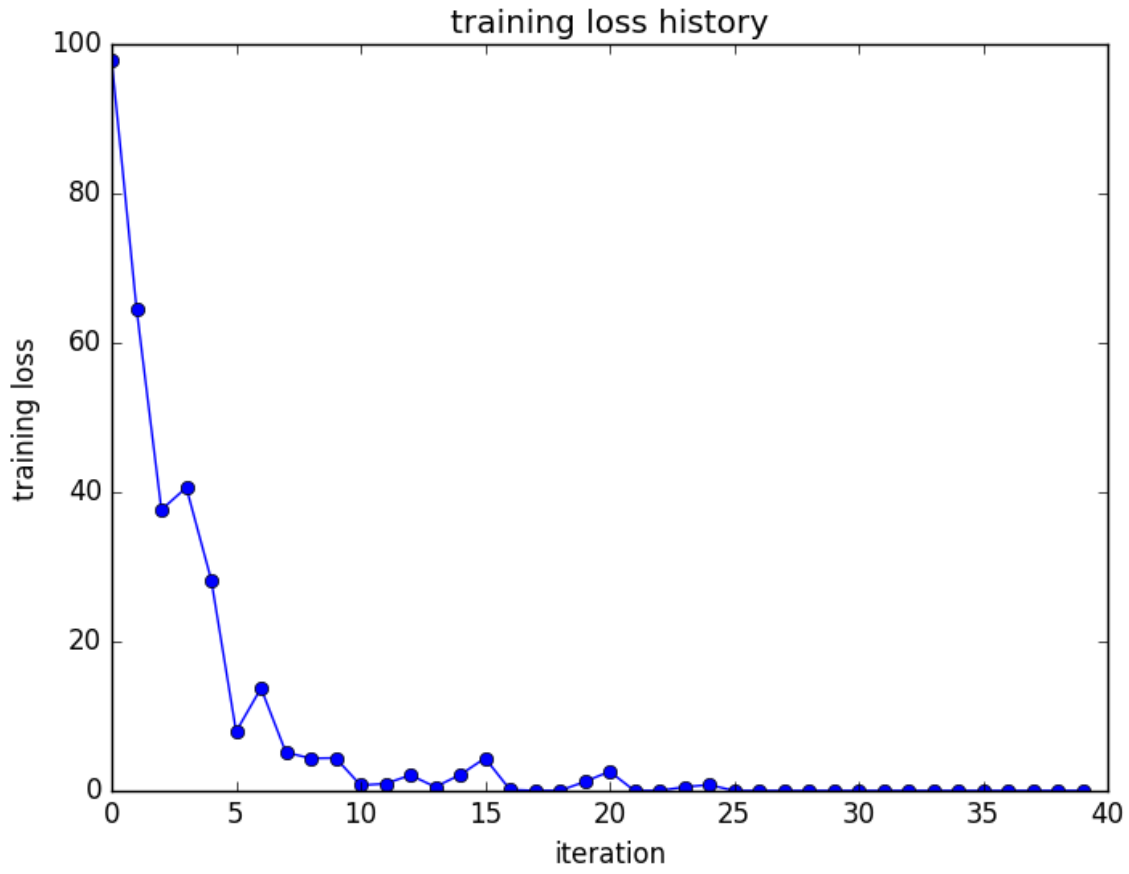
```
num_train=50
small_data={
    'X_train':data['X_train'][:num_train],
    'y_train':data['y_train'][:num_train],
    'X_val':data['X_val'],
    'y_val':data['y_val'],
}

weight_scale=1e-2
learning_rate=3e-4
model=FullyConnectedNet([100,100,100,100],weight_scale=weight_scale,dtype=np.float64)
solver=Solver(model,small_data,print_every=10,num_epochs=20,batch_size=25,update_rule='sgd',optim_config={'learning_rate':learning_rate,})
solver.train()

plt.plot(solver.loss_history,'-o')
plt.title('training loss history')
```

```
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.show()
```

运行结果如下：

```
(Iteration 1 / 40) loss: 97.854722
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.114000
(Epoch 1 / 20) train acc: 0.300000; val_acc: 0.104000
(Epoch 2 / 20) train acc: 0.380000; val_acc: 0.107000
(Epoch 3 / 20) train acc: 0.660000; val_acc: 0.090000
(Epoch 4 / 20) train acc: 0.720000; val_acc: 0.103000
(Epoch 5 / 20) train acc: 0.780000; val_acc: 0.098000
(Iteration 11 / 40) loss: 0.731547
(Epoch 6 / 20) train acc: 0.780000; val_acc: 0.097000
(Epoch 7 / 20) train acc: 0.940000; val_acc: 0.099000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.098000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.109000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.105000
(Iteration 21 / 40) loss: 2.570933
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.102000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.102000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.106000
(Iteration 31 / 40) loss: 0.000013
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.105000
```

loss曲线图如下：



哇塞，loss已经收敛了，最后已经趋近于0了，很不错！

我们看下两次的对比：

1. 降低了学习率；学习率太大可能出现梯度爆炸，loss值为nan的情况啊
2. 增加了网络层数。

当然，层数多了，对参数的初始化也是更敏感的。

5 更新规则

在前面几章，我们对参数的更新都是采用随机梯度下降法，也就是沿着负梯度方向改变参数，但是实际中还有很多更新方法，如SGD+Momentum，我们就来看看不同的更新规则会不会给结果带来不一样的影响。

5.1 SGD+Momenrum

我们直接上代码：

```
def sgd_momentum(w, dw, config=None):

    #Performs stochastic gradient descent with momentum.
```

```
    #config format:
    #- learning_rate: Scalar learning rate.
    #- momentum: Scalar between 0 and 1 giving the momentum value.
    #  Setting momentum = 0 reduces to sgd.
    #- velocity: A numpy array of the same shape as w and dw used to
store a moving
    #  average of the gradients.

    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('momentum', 0.9)
    v = config.get('velocity', np.zeros_like(w))

    next_w = None

 ###########################################################################
#####
    # TODO: Implement the momentum update formula. Store the updated
value in   #
    # the next_w variable. You should also use and update the velocity v.
      #

 ###########################################################################
#####
    v = config['momentum'] * v - config['learning_rate'] * dw
    next_w = w + v
    # pass

 ###########################################################################
#####
    #                             END OF YOUR CODE
       #

 ###########################################################################
#####
    config['velocity'] = v

    return next_w, config
```

该优化器接受两个参数 `momentum` 动量， `learning_rate` 学习率。

同样，我们来检验下：

```
N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
```

```
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])
```

结果如下：

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

误差小于1e-8，还是很成功的。

5.2 自适应优化器

上面这两种优化器，都是学习率固定的。下面我们来看下两种自适应的优化器。

5.2.1 RMSProp

实现代码如下：

```
def rmsprop(x, dx, config=None):

    #Uses the RMSProp update rule, which uses a moving average of squared
gradient
    #values to set adaptive per-parameter learning rates.
    #config format:
    #- learning_rate: Scalar learning rate.
    #- decay_rate: Scalar between 0 and 1 giving the decay rate for the
squared
    #  gradient cache.
```

```
    #- epsilon: Small scalar used for smoothing to avoid dividing by
zero.
    #- cache: Moving average of second moments of gradients.

    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('decay_rate', 0.99)
    config.setdefault('epsilon', 1e-8)
    config.setdefault('cache', np.zeros_like(x))

    next_x = None

 ################################################################################
#####
    # TODO: Implement the RMSprop update formula, storing the next value
of x   #
    # in the next_x variable. Don't forget to update cache value stored
in      #
    # config['cache'].
       #

 ################################################################################
#####
    config['cache'] = config['decay_rate'] * config['cache'] + (1 -
config['decay_rate']) * (dx ** 2)
    next_x = x - config['learning_rate'] * dx / (np.sqrt(config['cache'])
+ config['epsilon'])
    # pass

 ################################################################################
#####
    #                             END OF YOUR CODE
       #

 ################################################################################
#####

    return next_x, config
```

该优化器的参数有`learning_rate` `decay_rate` `epsilon`，这里设置的默认值都是论文中提供的，建议不进行修改。

5.2.3 Adam

实现代码如下：

```
def adam(x, dx, config=None):

    #Uses the Adam update rule, which incorporates moving averages of
both the
    #gradient and its square and a bias correction term.
    #config format:
    #- learning_rate: Scalar learning rate.
    #- beta1: Decay rate for moving average of first moment of gradient.
    #- beta2: Decay rate for moving average of second moment of gradient.
    #- epsilon: Small scalar used for smoothing to avoid dividing by
zero.
    #- m: Moving average of gradient.
    #- v: Moving average of squared gradient.
    #- t: Iteration number.

    if config is None: config = {}
    config.setdefault('learning_rate', 1e-3)
    config.setdefault('beta1', 0.9)
    config.setdefault('beta2', 0.999)
    config.setdefault('epsilon', 1e-8)
    config.setdefault('m', np.zeros_like(x))
    config.setdefault('v', np.zeros_like(x))
    config.setdefault('t', 0)

    next_x = None

 ##############################################################################
#####
    # TODO: Implement the Adam update formula, storing the next value of
x in   #
    # the next_x variable. Don't forget to update the m, v, and t
variables     #
    # stored in config.
      #

 ##############################################################################
#####
    config['t'] += 1
    config['m'] = config['beta1'] * config['m'] + (1 - config['beta1']) *
dx
    config['v'] = config['beta2'] * config['v'] + (1 - config['beta2']) *
(dx ** 2)
    mb = config['m'] / (1 - config['beta1'] ** config['t'])
    vb = config['v'] / (1 - config['beta2'] ** config['t'])
```

```
    next_x = x - config['learning_rate'] * mb / (np.sqrt(vb) + config['epsilon'])
    # pass

 ##############################################################################
    #                             END OF YOUR CODE                               #
       #
 ##############################################################################

    return next_x, config
```

该优化器我们需要设置的超参数有 `learning_rate` `beta1` `beta2` `epsilon` 实际中，我们也是只需要调整学习率这一个超参数就可以的。

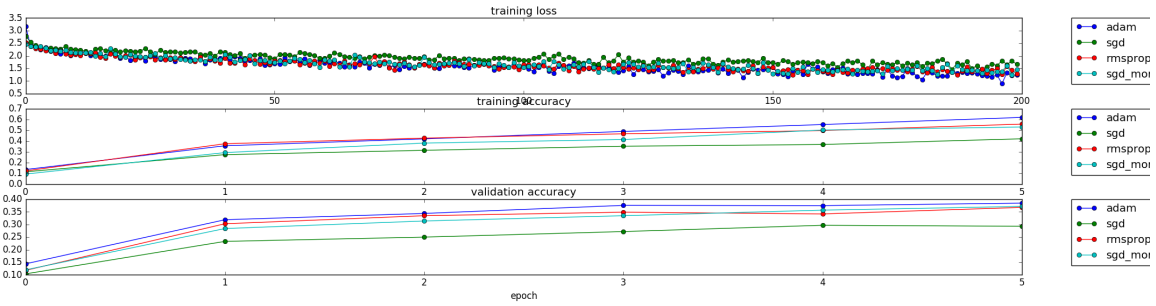
5.3 四种优化器的比较

下面我们同样比较下这四种优化器的收敛效果。
代码如下：

```
num_train=4000
small_data={
    'X_train':data['X_train'][:num_train],
    'y_train':data['y_train'][:num_train],
    'X_val':data['X_val'],
    'y_val':data['y_val'],
}
solvers={}

learning_rates={'sgd':1e-2,'sgd_momentum':1e-2,'rmsprop':1e-4,'adam':1e-3}
for update_relu in ['sgd','sgd_momentum','rmsprop','adam']:
    print('running with',update_relu)

    model=FullyConnectedNet([100,100,100,100,100],weight_scale=5e-2)


 solver=Solver(model,small_data,num_epochs=5,batch_size=100,update_rule=update_relu,
                  optim_config={'learning_rate':learning_rates[update_relu]},verbose=True)
    solvers[update_relu]=solver
```

```
    solver.train()
    print()

plt.subplot(311)
plt.title('training loss')
plt.xlabel('iteration')

plt.subplot(312)
plt.title('training accuracy')
plt.xlabel('epoch')

plt.subplot(313)
plt.title('validation accuracy')
plt.xlabel('epoch')

for update_relu,solver in solvers.items():
    plt.subplot(311)
    plt.plot(solver.loss_history,'-o',label=update_relu)
    plt.legend( bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

    plt.subplot(312)
    plt.plot(solver.train_acc_history,'-o',label=update_relu)
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

    plt.subplot(313)
    plt.plot(solver.val_acc_history,'-o',label=update_relu)
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)



plt.gcf().set_size_inches(15,15)
plt.show()
```

结果如下



我们可以看到，Adam的收敛效果是最好的。

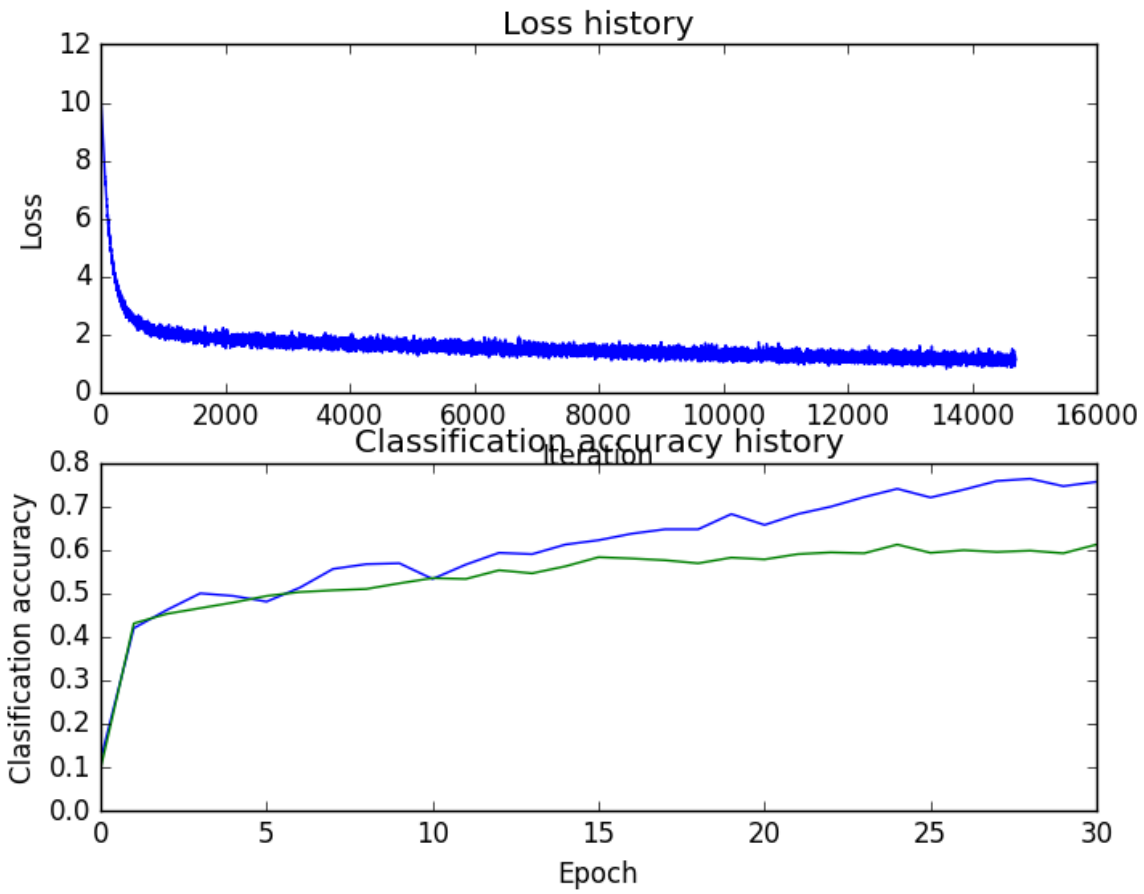
6 网络训练

下面我们就使用adam优化器来训练我们的网络，对cifar10 得到更高的准确率。
代码如下：

```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# batch normalization and dropout useful. Store your best model in the         #
# best_model variable.                                                         #
################################################################################
X_val= data['X_val']
y_val= data['y_val']
X_test= data['X_test']
y_test= data['y_test']

learning_rate = 3.1e-4
weight_scale = 2.5e-2 #1e-5
model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
                weight_scale=weight_scale, dtype=np.float64, dropout=0.25, use_batchnorm=True, reg=1e-2)
solver = Solver(model, data,
                print_every=500, num_epochs=30, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=0.9
         )

solver.train()
scores = model.loss(data['X_test'])
y_pred = np.argmax(scores, axis = 1)
acc = np.mean(y_pred == data['y_test'])
print ('test acc: %f' %(acc))
best_model = model

plt.subplot(2, 1, 1)
plt.plot(solver.loss_history)
plt.title('Loss history')
```

```
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, label='train')
plt.plot(solver.val_acc_history, label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.show()
```

结果如下：



7 预测

我们使用上章保存的最好模型来完成预测工作，并最终输出准确率。
代码如下：

```
y_test_pred = np.argmax(best_model.loss(X_test), axis=1)
y_val_pred = np.argmax(best_model.loss(X_val), axis=1)
print ('Validation set accuracy: ', (y_val_pred == y_val).mean())
print ('Test set accuracy: ', (y_test_pred == y_test).mean())
```

输出结果如下：

```
Test set accuracy: 0.548
```

注意：我们这里加入加入了Dropout和BN，使得结果较好，这两个的作用我们会在接下来的一次作业中进行展示。

8 总结

在实践中，3层的神经网络会比2层的神经网络的表现好，然而继续加深（做到4、5、6层）很少有很大的帮助。卷积神经网络的情况却不同，在卷积神经网络中，对于一个良好的识别系统来说，深度是一个极端重要的因素（比如数十（以10为数量级）个可学习的层）。对于该现象的一种解释观点是：因为图像拥有层次化结构（比如脸是由眼睛等组成，眼睛又是由边缘组成），所以多层处理对于这种数据就有直观意义。