

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：

1 基础层

1.1 全连接层

1.1.1 前向传播

1.1.2 反向传播

1.2 Relu层

1.2.1 前向传播

1.2.2 反向传播

1.3 损失函数层

2 两层神经网络

3 Solver

4 多层神经网络

4.1 多层网络训练

5 更新规则

5.1 SGD+Momenrum

5.2 自适应优化器

5.2.1 RMSProp

5.2.3 Adam

5.3 四种优化器的比较

6 网络训练

7 预测

8 总结

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：

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

代码如下：

```
1 def affine_forward(x,w,b):
2
3     '''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)'''
4
5     out=None
6     N=x.shape[0]
7     x_rsp=x.reshape(N,-1)
8     out=x_rsp.dot(w)+b
9     cache=(x,w,b)
10
11     return out,cache
```

我们可以检验下计算的误差，检验代码如下：

```
1 num_inputs=2
2 input_shape=(4,5,6)
3 output_dim=3
4 input_size=num_inputs*np.prod(input_shape)
5 weight_size=output_dim*np.prod(input_shape)
6
7 x=np.linspace(-0.1,0.5,num=input_size).reshape(num_inputs,*input_shape)
8 w=np.linspace(-0.2,0.3,num=weight_size).reshape(np.prod(input_shape),output_dim)
9 b=np.linspace(-0.3,0.1,num=output_dim)
10
11 out,_=affine_forward(x,w,b)
12 correct_out=np.array([[ 1.49834967,  1.70660132,  1.91485297],
13                       [ 3.25553199,  3.5141327,  3.77273342]])
14
15 print('testing affine_forward function:')
```

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

输出结果如下：

```
1 testing affine_forward function:
2 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：

```
1 def layer_backward(dout,cache):
2     x,w,z,out=cache
3     dx=
4     dw=
5     return dx,dw
```

实现代码如下：

```
1 def affine_backward(dout,cache):
2     '''dout:(N,M); x:(N,d_1,...d_k) x_rsp中我们会将x转化为(N,D)，其中
3     D=d_1*...*d_k; w:(D,M); dx:(D,d_1,...d_k); dw:(D,M); db:(M,)'''
4     x,w,b=cache
5     dx,dw,db=None,None,None
6     N=x.shape[0]
7     x_rsp=x.reshape(N,-1)
8     dx=dout.dot(w.T)
9     dx=dx.reshape(*x.shape)
10    dw=x_rsp.T.dot(dout)
11    db=np.sum(dout,axis=0)
12    return dx,dw,db
```

注：如果不清楚反向传播过程中如何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：

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

```

2 w=np.random.randn(6,5)
3 b=np.random.randn(5)
4 dout=np.random.randn(10,5)
5
6 dx_num=eval_numerical_gradient_array(lambda x:affine_forward(x,w,b)
7 [0],x,dout)
8 dw_num=eval_numerical_gradient_array(lambda w:affine_forward(x,w,b)
9 [0],w,dout)
10 db_num=eval_numerical_gradient_array(lambda b:affine_forward(x,w,b)
11 [0],b,dout)
12
13 _,cache=affine_forward(x,w,b)
14 dx,dw,db=affine_backward(dout,cache)
15
16 print('test affine_backward function:')
17 print('dx error:',rel_error(dx_num,dx))
18 print('dw error:',rel_error(dw_num,dw))
19 print('db error:',rel_error(db_num,db))

```

输出结果如下：

```

1 test affine_backward function:
2 dx error: 2.50214010994e-10
3 dw error: 9.11372672216e-11
4 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：

```

1 def relu_forward(x):
2     out=None
3     out=x*(x>=0)
4     cache=x
5     return out,cache

```

检验代码如下：

```

1 x=np.linspace(-0.5,0.5,num=12).reshape(3,4)
2 out,_=relu_forward(x)
3 correct_out = np.array([[ 0.,          0.,          0.,          0.,
4                          [ 0.,          0.,          0.04545455,
5                          0.13636364,],
6                          [ 0.22727273,  0.31818182,  0.40909091,  0.5,
7                          ]])
6 print('testing relu_forward function:')
7 print('difference:',rel_error(out,correct_out))

```

输出结果如下：

```

1 testing relu_forward function:
2 difference: 4.99999979802e-08

```

1.2.2 反向传播

代码如下：

```

1 def relu_backward(dout,cache):
2     dx,x=None,cache
3     dx=(x>=0)*dout
4     return dx

```

检验代码如下：

```

1 x=np.random.randn(10,10)
2 dout=np.random.randn(*x.shape)
3
4 dx_num=eval_numerical_gradient_array(lambda x:relu_forward(x)[0],x,dout)
5 _,cache=relu_forward(x)
6 dx=relu_backward(dout,cache)
7
8 print('testing relu_backward functin:')
9 print('dx error:',rel_error(dx_num,dx))

```

输出结果如下：

```

1 testing relu_backward functin:

```

```
2 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：

```
1 def affine_relu_forward(x,w,b):
2     a,fc_cache=affine_forward(x,w,b)
3     out,relu_cache=relu_forward(a)
4     cache=(fc_cache,relu_cache)
5     return out,cache
6
7 def affine_relu_backward(dout,cache):
8     fc_cache,relu_cache=cache
9     da=relu_backward(dout,relu_cache)
10    dx,dw,db=affine_backward(da,fc_cache)
11    return dx,dw,db
```

梯度检验代码如下：

```
1 x=np.random.randn(2,3,4)
2 w=np.random.randn(12,10)
3 b=np.random.randn(10)
4 dout=np.random.randn(2,10)
5
6 out,cache=affine_relu_forward(x,w,b)
7 dx,dw,db=affine_relu_backward(dout,cache)
8
9 dx_num=eval_numerical_gradient_array(lambda x:affine_relu_forward(x,w,b)
10 [0],x,dout)
11 dw_num=eval_numerical_gradient_array(lambda w:affine_relu_forward(x,w,b)
12 [0],w,dout)
13 db_num=eval_numerical_gradient_array(lambda b:affine_relu_forward(x,w,b)
14 [0],b,dout)
15
16 print('testing affine_relu_forward:')
17 print('dx error:',rel_error(dx_num,dx))
18 print('dw error:',rel_error(dw_num,dw))
19 print('db error:',rel_error(db_num,db))
```

运行结果如下：

```
1 testing affine_relu_forward:
```

```
2 dx_error: 1.58523685005e-10
3 dw_error: 3.68009862446e-10
4 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：

```
1 def svm_loss(x,y):
2     ''' :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'''
3     N=x.shape[0]
4     correct_class_scores=x[np.arange(N),y]
5     margins=np.maximum(0,x-correct_class_scores[:,np.newaxis]+1.0)
6     margins[np.arange(N),y]=0
7     loss=np.sum(margins)/N
8
9     num_pos=np.sum(margins>0,axis=1)
10    dx=np.zeros_like(x)
11    dx[margins>0]=1
12    dx[np.arange(N),y]-=num_pos
13    dx/=N
14    return loss,dx
15
16 def softmax_loss(x,y):
17     probs=np.exp(x-np.max(x,axis=1,keepdims=True))
18     probs/=np.sum(probs,axis=1,keepdims=True)
19     N=x.shape[0]
20     loss=-np.sum(np.log(probs[np.arange(N),y]))/N
21     dx=probs.copy()
22     dx[np.arange(N),y]-=1
23     dx/=N
24     return loss,dx
```

我们检验下我们做的是不是正确，代码如下：

```
1 num_classes,num_inputs=10,50
2 x=0.001*np.random.randn(num_inputs,num_classes)
3 y=np.random.randint(num_classes,size=num_inputs)
4
```

```

5 dx_num=eval_numerical_gradient(lambda x:svm_loss(x,y)[0],x,verbose=False)
6 loss,dx=svm_loss(x,y)
7 print('testing svm_loss:')
8 print('loss:',loss)
9 print('dx error:',rel_error(dx_num,dx))
10
11 dx_num=eval_numerical_gradient(lambda x:softmax_loss(x,y)
12 [0],x,verbose=False)
13 loss,dx=softmax_loss(x,y)
14 print('\ntesting softmax_loss:')
15 print('loss:',loss)
16 print('dx error:',rel_error(dx_num,dx))

```

运行结果如下：

```

1 testing svm_loss:
2 loss: 9.00034570116
3 dx error: 1.40215660067e-09
4
5 testing softmax_loss:
6 loss: 2.30262010084
7 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：

```

1 class TwoLayerNet(object):
2     def
3     __init__(self,input_dim=3*32*32,hidden_dim=100,num_classes=10,weight_scale=1e-3,reg=0.0):
4         self.params={}
5         self.reg=reg
6
7         self.params['W1']=weight_scale*np.random.rand(input_dim,hidden_dim)
8         self.params['b1']=np.zeros(hidden_dim)

```



```

7         self.params['W2']=weight_scale*np.random.randn(hidden_dim,num_classes)
8         self.params['b2']=np.zeros(num_classes)
9
10        def loss(self,X,y=None):
11            scores=None
12
13            ar1_out,ar1_cache=affine_relu_forward(X,self.params['W1'],self.params['b
14            1'])
15
16            a2_out,a2_cache=affine_forward(ar1_out,self.params['W2'],self.params['b2
17            '])
18
19            scores=a2_out
20
21            if y is None:
22                return scores
23
24            loss,grads=0,{ }
25
26            loss,dscores=softmax_loss(scores,y)
27
28            loss=loss+0.5*self.reg*np.sum(self.params['W1']*self.params['W1'])+0.5*s
29            elf.reg*np.sum(self.params['W2']*self.params['W2'])
30            dx2,dw2,db2=affine_backward(scores,a2_cache)
31            grads['W2']=dw2+self.reg*self.params['W2']
32            grads['b2']=db2
33
34            dx1,dw1,db1=affine_relu_forward(dx2,ar1_cache)
35            grads['W1']=dw1+self.reg*self.params['W1']
36            grads['b1']=db1
37
38            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：

```

1  N, D, H, C = 3, 5, 50, 7
2  X = np.random.randn(N, D)
3  y = np.random.randint(C, size=N)
4
5  std = 1e-2
6  model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C,
7                        weight_scale=std)
8
9  print ('Testing initialization ... ')

```

```

9 W1_std = abs(model.params['W1'].std() - std)
10 b1 = model.params['b1']
11 W2_std = abs(model.params['W2'].std() - std)
12 b2 = model.params['b2']
13 assert W1_std < std / 10, 'First layer weights do not seem right'
14 assert np.all(b1 == 0), 'First layer biases do not seem right'
15 assert W2_std < std / 10, 'Second layer weights do not seem right'
16 assert np.all(b2 == 0), 'Second layer biases do not seem right'
17
18 print ('Testing test-time forward pass ... ')
19 model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
20 model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
21 model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
22 model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
23 X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
24 scores = model.loss(X)
25 correct_scores = np.asarray(
26     [[11.53165108, 12.2917344, 13.05181771, 13.81190102, 14.57198434,
27       15.33206765, 16.09215096],
28      [12.05769098, 12.74614105, 13.43459113, 14.1230412, 14.81149128,
29       15.49994135, 16.18839143],
30      [12.58373087, 13.20054771, 13.81736455, 14.43418138, 15.05099822,
31       15.66781506, 16.2846319 ]])
32 scores_diff = np.abs(scores - correct_scores).sum()
33 assert scores_diff < 1e-6, 'Problem with test-time forward pass'
34
35 print ('Testing training loss (no regularization)')
36 y = np.asarray([0, 5, 1])
37 loss, grads = model.loss(X, y)
38 correct_loss = 3.4702243556
39 assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time
40 loss'
41
42 model.reg = 1.0
43 loss, grads = model.loss(X, y)
44 correct_loss = 26.5948426952
45 assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization
46 loss'
47
48 for reg in [0.0, 0.7]:
49     print ('Running numeric gradient check with reg = ', reg)
50     model.reg = reg
51     loss, grads = model.loss(X, y)

```

输出结果如下：

```
1 Testing initialization ...
2 Testing test-time forward pass ...
3 Testing training loss (no regularization)
4 Running numeric gradient check with reg = 0.0
5 W1 relative error: 1.22e-08
6 W2 relative error: 3.35e-10
7 b1 relative error: 8.37e-09
8 b2 relative error: 2.53e-10
9 Running numeric gradient check with reg = 0.7
10 W1 relative error: 2.53e-07
11 W2 relative error: 7.98e-08
12 b1 relative error: 1.56e-08
13 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：

```
1 data={}
2 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}
3 model=TwoLayerNet(reg=1e-1)
4 solver=None
5
6 solver=Solver(model,data,update_rule='sgd',
7               optim_config={'learning_rate':1e-3},
8               lr_decay=0.8,num_epochs=10,batch_size=100,print_every=100)
9 solver.train()
```

```

10 scores=model.loss(data['X_test'])
11 y_pred=np.argmax(scores,axis=1)
12 acc=np.mean(y_pred==data['y_test'])
13 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

```

1 (Iteration 1 / 4900) loss: 2.318766
2 (Epoch 0 / 10) train acc: 0.111000; val_acc: 0.122000
3 (Iteration 101 / 4900) loss: 1.736319
4 (Iteration 201 / 4900) loss: 1.761152
5 (Iteration 301 / 4900) loss: 1.580906
6 (Iteration 401 / 4900) loss: 1.779731
7 (Epoch 1 / 10) train acc: 0.473000; val_acc: 0.456000
8 (Iteration 501 / 4900) loss: 1.633682
9 (Iteration 601 / 4900) loss: 1.375359
10 (Iteration 701 / 4900) loss: 1.574423
11 (Iteration 801 / 4900) loss: 1.643599
12 (Iteration 901 / 4900) loss: 1.484345
13 (Epoch 2 / 10) train acc: 0.478000; val_acc: 0.479000
14 (Iteration 1001 / 4900) loss: 1.517680
15 (Iteration 1101 / 4900) loss: 1.571929
16 (Iteration 1201 / 4900) loss: 1.315616
17 (Iteration 1301 / 4900) loss: 1.367084
18 (Iteration 1401 / 4900) loss: 1.478173
19 (Epoch 3 / 10) train acc: 0.513000; val_acc: 0.491000
20 (Iteration 1501 / 4900) loss: 1.525170
21 (Iteration 1601 / 4900) loss: 1.507696
22 (Iteration 1701 / 4900) loss: 1.310258
23 (Iteration 1801 / 4900) loss: 1.307820
24 (Iteration 1901 / 4900) loss: 1.420765
25 (Epoch 4 / 10) train acc: 0.562000; val_acc: 0.492000

```

```
26 (Iteration 2001 / 4900) loss: 1.346618
27 (Iteration 2101 / 4900) loss: 1.368148
28 (Iteration 2201 / 4900) loss: 1.273601
29 (Iteration 2301 / 4900) loss: 1.255306
30 (Iteration 2401 / 4900) loss: 1.434615
31 (Epoch 5 / 10) train acc: 0.563000; val_acc: 0.503000
32 (Iteration 2501 / 4900) loss: 1.320840
33 (Iteration 2601 / 4900) loss: 1.451932
34 (Iteration 2701 / 4900) loss: 1.319160
35 (Iteration 2801 / 4900) loss: 1.269273
36 (Iteration 2901 / 4900) loss: 1.351908
37 (Epoch 6 / 10) train acc: 0.543000; val_acc: 0.513000
38 (Iteration 3001 / 4900) loss: 1.277055
39 (Iteration 3101 / 4900) loss: 1.254344
40 (Iteration 3201 / 4900) loss: 1.386045
41 (Iteration 3301 / 4900) loss: 1.359556
42 (Iteration 3401 / 4900) loss: 1.220027
43 (Epoch 7 / 10) train acc: 0.579000; val_acc: 0.542000
44 (Iteration 3501 / 4900) loss: 1.334799
45 (Iteration 3601 / 4900) loss: 1.099247
46 (Iteration 3701 / 4900) loss: 1.287256
47 (Iteration 3801 / 4900) loss: 1.196124
48 (Iteration 3901 / 4900) loss: 1.221174
49 (Epoch 8 / 10) train acc: 0.604000; val_acc: 0.526000
50 (Iteration 4001 / 4900) loss: 1.317148
51 (Iteration 4101 / 4900) loss: 1.267705
52 (Iteration 4201 / 4900) loss: 1.209211
53 (Iteration 4301 / 4900) loss: 1.043793
54 (Iteration 4401 / 4900) loss: 1.346820
55 (Epoch 9 / 10) train acc: 0.605000; val_acc: 0.523000
56 (Iteration 4501 / 4900) loss: 1.176413
57 (Iteration 4601 / 4900) loss: 1.316058
58 (Iteration 4701 / 4900) loss: 1.181584
59 (Iteration 4801 / 4900) loss: 1.182825
60 (Epoch 10 / 10) train acc: 0.601000; val_acc: 0.522000
61 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：

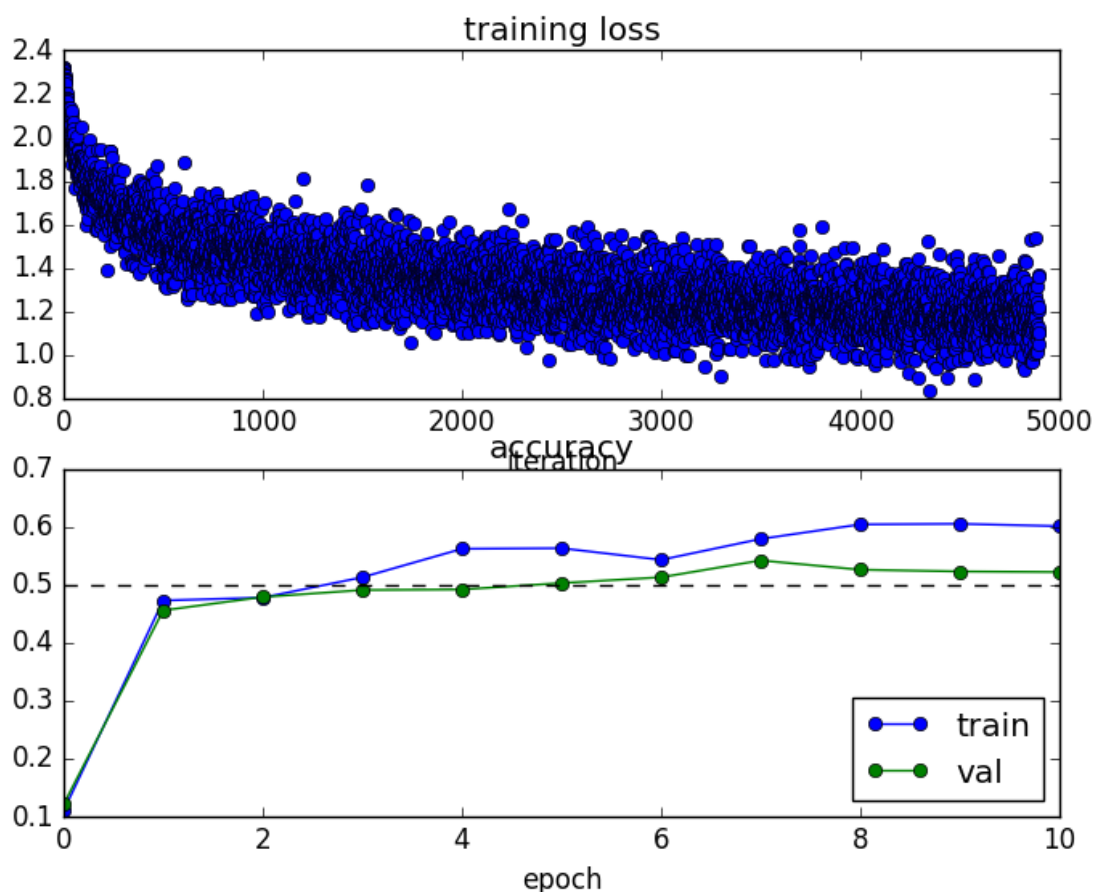
```
1 plt.subplot(2,1,1)
```

```

2 plt.title('training loss')
3 plt.plot(solver.loss_history, 'o')
4 plt.xlabel('iteration')
5
6 plt.subplot(2,1,2)
7 plt.title('accuracy')
8 plt.plot(solver.train_acc_history, '-o', label='train')
9 plt.plot(solver.val_acc_history, '-o', label='val')
10 plt.plot([0.5]*len(solver.val_acc_history), 'k--')
11 plt.xlabel('epoch')
12 plt.legend(loc='lower right')#图例位置
13 plt.gcf().set_size_inches(15,12)
14 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：

```

1 class FullyConnectedNet(object):
2
3     #A fully-connected neural network with an arbitrary number of hidden
4     layers,
5     #ReLU nonlinearities, and a softmax loss function. This will also
6     implement
7     #dropout and batch normalization as options. For a network with L
8     layers,
9     #the architecture will be
10
11     #{affine - [batch norm] - relu - [dropout]} x (L - 1) - affine -
12     softmax
13
14     #where batch normalization and dropout are optional, and the {...}
15     block is
16     #repeated L - 1 times.
17
18     #Similar to the TwoLayerNet above, learnable parameters are stored
19     in the
20     #self.params dictionary and will be learned using the Solver class.
21
22     def __init__(self, hidden_dims, input_dim=3 * 32 * 32,
23     num_classes=10,
24     dropout=0, use_batchnorm=False, reg=0.0,
25     weight_scale=1e-2, dtype=np.float32, seed=None):
26
27     #Initialize a new FullyConnectedNet.
28
29     #Inputs:
30     #- hidden_dims: A list of integers giving the size of each
31     hidden layer.
32     #- input_dim: An integer giving the size of the input.
33     #- num_classes: An integer giving the number of classes to
34     classify.
35     #- dropout: Scalar between 0 and 1 giving dropout strength. If
36     dropout=0 then
37     # the network should not use dropout at all.
38     #- use_batchnorm: Whether or not the network should use batch
39     normalization.
40     #- reg: Scalar giving L2 regularization strength.
41     #- weight_scale: Scalar giving the standard deviation for random
42     # initialization of the weights.
43     #- dtype: A numpy datatype object; all computations will be
44     performed using

```

```

34         # this datatype. float32 is faster but less accurate, so you
        should use
35         # float64 for numeric gradient checking.
36         #- seed: If not None, then pass this random seed to the dropout
        layers. This
37         # will make the dropout layers deterministic so we can gradient
        check the
38         # model.
39
40         self.use_batchnorm = use_batchnorm
41         self.use_dropout = dropout > 0
42         self.reg = reg
43         self.num_layers = 1 + len(hidden_dims)
44         self.dtype = dtype
45         self.params = {}
46
47         #####
        #####
48         # TODO: Initialize the parameters of the network, storing all
        values in #
49         # the self.params dictionary. Store weights and biases for the
        first layer #
50         # in W1 and b1; for the second layer use W2 and b2, etc. Weights
        should be #
51         # initialized from a normal distribution with standard deviation
        equal to #
52         # weight_scale and biases should be initialized to zero.
        #
53         #
        #
54         # When using batch normalization, store scale and shift
        parameters for the #
55         # first layer in gamma1 and beta1; for the second layer use
        gamma2 and #
56         # beta2, etc. Scale parameters should be initialized to one and
        shift #
57         # parameters should be initialized to zero.
        #
58
        #####
        #####
59         layer_input_dim = input_dim
60         for i, hd in enumerate(hidden_dims):
61             self.params['W%d' % (i + 1)] = weight_scale *
        np.random.randn(layer_input_dim, hd)

```



```

62         self.params['b%d' % (i + 1)] = weight_scale * np.zeros(hd)
63         if self.use_batchnorm:
64             self.params['gamma%d' % (i + 1)] = np.ones(hd)
65             self.params['beta%d' % (i + 1)] = np.zeros(hd)
66             layer_input_dim = hd
67             self.params['W%d' % (self.num_layers)] = weight_scale *
np.random.randn(layer_input_dim, num_classes)
68             self.params['b%d' % (self.num_layers)] = weight_scale *
np.zeros(num_classes)
69             # pass
70
#####
#####
71             #                               END OF YOUR CODE
72             #
73
#####
#####
74             # When using dropout we need to pass a dropout_param dictionary
to each
75             # dropout layer so that the layer knows the dropout probability
and the mode
76             # (train / test). You can pass the same dropout_param to each
dropout layer.
77             self.dropout_param = {}
78             if self.use_dropout:
79                 self.dropout_param = {'mode': 'train', 'p': dropout}
80                 if seed is not None:
81                     self.dropout_param['seed'] = seed
82
83             # With batch normalization we need to keep track of running
means and
84             # variances, so we need to pass a special bn_param object to
each batch
85             # normalization layer. You should pass self.bn_params[0] to the
forward pass
86             # of the first batch normalization layer, self.bn_params[1] to
the forward
87             # pass of the second batch normalization layer, etc.
88             self.bn_params = []
89             if self.use_batchnorm:
90                 self.bn_params = [{'mode': 'train'} for i in
range(self.num_layers - 1)]
91
92             # Cast all parameters to the correct datatype

```

```

93         for k, v in self.params.iteritems():
94             self.params[k] = v.astype(dtype)
95
96     def loss(self, X, y=None):
97
98         #Compute loss and gradient for the fully-connected net.
99         #Input / output: Same as TwoLayerNet above.
100
101         X = X.astype(self.dtype)
102         mode = 'test' if y is None else 'train'
103
104         # Set train/test mode for batchnorm params and dropout param
since they
105         # behave differently during training and testing.
106         if self.dropout_param is not None:
107             self.dropout_param['mode'] = mode
108         if self.use_batchnorm:
109             for bn_param in self.bn_params:
110                 bn_param['mode'] = mode
111
112         scores = None
113
114         #####
115         # TODO: Implement the forward pass for the fully-connected net,
computing #
116         # the class scores for X and storing them in the scores
variable. #
117         #
118         # When using dropout, you'll need to pass self.dropout_param to
each #
119         # dropout forward pass.
#
120         #
121         # When using batch normalization, you'll need to pass
self.bn_params[0] to #
122         # the forward pass for the first batch normalization layer, pass
#
123         # self.bn_params[1] to the forward pass for the second batch
normalization #
124         # layer, etc.
#

```

```

124 #####
125 #####
126     layer_input = X
127     ar_cache = {}
128     dp_cache = {}
129
130     for lay in range(self.num_layers - 1):
131         if self.use_batchnorm:
132             layer_input, ar_cache[lay] =
133             affine_bn_relu_forward(layer_input,
134
135             self.params['W%d' % (lay + 1)],
136
137             self.params['b%d' % (lay + 1)],
138
139             self.params['gamma%d' % (lay + 1)],
140
141             self.params['beta%d' % (lay + 1)],
142
143             self.bn_params[lay])
144         else:
145             layer_input, ar_cache[lay] =
146             affine_relu_forward(layer_input, self.params['W%d' % (lay + 1)],
147
148             self.params['b%d' % (lay + 1)])
149
150         if self.use_dropout:
151             layer_input, dp_cache[lay] =
152             dropout_forward(layer_input, self.dropout_param)
153
154     ar_out, ar_cache[self.num_layers] = affine_forward(layer_input,
155     self.params['W%d' % (self.num_layers)],
156
157     self.params['b%d' % (self.num_layers)])
158     scores = ar_out
159     # pass
160
161 #####
162 #####
163     #
164     #
165
166 #####
167 #####
168
169 #####
170 #####
171
172 #####
173 #####
174
175 #####
176 #####
177
178 #####
179 #####
180
181 #####
182 #####
183
184 #####
185 #####
186
187 #####
188 #####
189
190 #####
191 #####
192
193 #####
194 #####
195
196 #####
197 #####
198
199 #####
200 #####
201
202 #####
203 #####
204
205 #####
206 #####
207
208 #####
209 #####
210
211 #####
212 #####
213
214 #####
215 #####
216
217 #####
218 #####
219
220 #####
221 #####
222
223 #####
224 #####
225
226 #####
227 #####
228
229 #####
230 #####
231
232 #####
233 #####
234
235 #####
236 #####
237
238 #####
239 #####
240
241 #####
242 #####
243
244 #####
245 #####
246
247 #####
248 #####
249
250 #####
251 #####
252
253 #####
254 #####
255
256 #####
257 #####
258
259 #####
260 #####
261
262 #####
263 #####
264
265 #####
266 #####
267
268 #####
269 #####
270
271 #####
272 #####
273
274 #####
275 #####
276
277 #####
278 #####
279
280 #####
281 #####
282
283 #####
284 #####
285
286 #####
287 #####
288
289 #####
290 #####
291
292 #####
293 #####
294
295 #####
296 #####
297
298 #####
299 #####
300
301 #####
302 #####
303
304 #####
305 #####
306
307 #####
308 #####
309
310 #####
311 #####
312
313 #####
314 #####
315
316 #####
317 #####
318
319 #####
320 #####
321
322 #####
323 #####
324
325 #####
326 #####
327
328 #####
329 #####
330
331 #####
332 #####
333
334 #####
335 #####
336
337 #####
338 #####
339
340 #####
341 #####
342
343 #####
344 #####
345
346 #####
347 #####
348
349 #####
350 #####
351
352 #####
353 #####
354
355 #####
356 #####
357
358 #####
359 #####
360
361 #####
362 #####
363
364 #####
365 #####
366
367 #####
368 #####
369
370 #####
371 #####
372
373 #####
374 #####
375
376 #####
377 #####
378
379 #####
380 #####
381
382 #####
383 #####
384
385 #####
386 #####
387
388 #####
389 #####
390
391 #####
392 #####
393
394 #####
395 #####
396
397 #####
398 #####
399
400 #####
401 #####
402
403 #####
404 #####
405
406 #####
407 #####
408
409 #####
410 #####
411
412 #####
413 #####
414
415 #####
416 #####
417
418 #####
419 #####
420
421 #####
422 #####
423
424 #####
425 #####
426
427 #####
428 #####
429
430 #####
431 #####
432
433 #####
434 #####
435
436 #####
437 #####
438
439 #####
440 #####
441
442 #####
443 #####
444
445 #####
446 #####
447
448 #####
449 #####
450
451 #####
452 #####
453
454 #####
455 #####
456
457 #####
458 #####
459
460 #####
461 #####
462
463 #####
464 #####
465
466 #####
467 #####
468
469 #####
470 #####
471
472 #####
473 #####
474
475 #####
476 #####
477
478 #####
479 #####
480
481 #####
482 #####
483
484 #####
485 #####
486
487 #####
488 #####
489
490 #####
491 #####
492
493 #####
494 #####
495
496 #####
497 #####
498
499 #####
500 #####
501
502 #####
503 #####
504
505 #####
506 #####
507
508 #####
509 #####
510
511 #####
512 #####
513
514 #####
515 #####
516
517 #####
518 #####
519
520 #####
521 #####
522
523 #####
524 #####
525
526 #####
527 #####
528
529 #####
530 #####
531
532 #####
533 #####
534
535 #####
536 #####
537
538 #####
539 #####
540
541 #####
542 #####
543
544 #####
545 #####
546
547 #####
548 #####
549
550 #####
551 #####
552
553 #####
554 #####
555
556 #####
557 #####
558
559 #####
560 #####
561
562 #####
563 #####
564
565 #####
566 #####
567
568 #####
569 #####
570
571 #####
572 #####
573
574 #####
575 #####
576
577 #####
578 #####
579
580 #####
581 #####
582
583 #####
584 #####
585
586 #####
587 #####
588
589 #####
590 #####
591
592 #####
593 #####
594
595 #####
596 #####
597
598 #####
599 #####
600
601 #####
602 #####
603
604 #####
605 #####
606
607 #####
608 #####
609
610 #####
611 #####
612
613 #####
614 #####
615
616 #####
617 #####
618
619 #####
620 #####
621
622 #####
623 #####
624
625 #####
626 #####
627
628 #####
629 #####
630
631 #####
632 #####
633
634 #####
635 #####
636
637 #####
638 #####
639
640 #####
641 #####
642
643 #####
644 #####
645
646 #####
647 #####
648
649 #####
650 #####
651
652 #####
653 #####
654
655 #####
656 #####
657
658 #####
659 #####
660
661 #####
662 #####
663
664 #####
665 #####
666
667 #####
668 #####
669
670 #####
671 #####
672
673 #####
674 #####
675
676 #####
677 #####
678
679 #####
680 #####
681
682 #####
683 #####
684
685 #####
686 #####
687
688 #####
689 #####
690
691 #####
692 #####
693
694 #####
695 #####
696
697 #####
698 #####
699
700 #####
701 #####
702
703 #####
704 #####
705
706 #####
707 #####
708
709 #####
710 #####
711
712 #####
713 #####
714
715 #####
716 #####
717
718 #####
719 #####
720
721 #####
722 #####
723
724 #####
725 #####
726
727 #####
728 #####
729
730 #####
731 #####
732
733 #####
734 #####
735
736 #####
737 #####
738
739 #####
740 #####
741
742 #####
743 #####
744
745 #####
746 #####
747
748 #####
749 #####
750
751 #####
752 #####
753
754 #####
755 #####
756
757 #####
758 #####
759
760 #####
761 #####
762
763 #####
764 #####
765
766 #####
767 #####
768
769 #####
770 #####
771
772 #####
773 #####
774
775 #####
776 #####
777
778 #####
779 #####
780
781 #####
782 #####
783
784 #####
785 #####
786
787 #####
788 #####
789
790 #####
791 #####
792
793 #####
794 #####
795
796 #####
797 #####
798
799 #####
800 #####
801
802 #####
803 #####
804
805 #####
806 #####
807
808 #####
809 #####
810
811 #####
812 #####
813
814 #####
815 #####
816
817 #####
818 #####
819
820 #####
821 #####
822
823 #####
824 #####
825
826 #####
827 #####
828
829 #####
830 #####
831
832 #####
833 #####
834
835 #####
836 #####
837
838 #####
839 #####
840
841 #####
842 #####
843
844 #####
845 #####
846
847 #####
848 #####
849
850 #####
851 #####
852
853 #####
854 #####
855
856 #####
857 #####
858
859 #####
860 #####
861
862 #####
863 #####
864
865 #####
866 #####
867
868 #####
869 #####
870
871 #####
872 #####
873
874 #####
875 #####
876
877 #####
878 #####
879
880 #####
881 #####
882
883 #####
884 #####
885
886 #####
887 #####
888
889 #####
890 #####
891
892 #####
893 #####
894
895 #####
896 #####
897
898 #####
899 #####
900
901 #####
902 #####
903
904 #####
905 #####
906
907 #####
908 #####
909
910 #####
911 #####
912
913 #####
914 #####
915
916 #####
917 #####
918
919 #####
920 #####
921
922 #####
923 #####
924
925 #####
926 #####
927
928 #####
929 #####
930
931 #####
932 #####
933
934 #####
935 #####
936
937 #####
938 #####
939
940 #####
941 #####
942
943 #####
944 #####
945
946 #####
947 #####
948
949 #####
950 #####
951
952 #####
953 #####
954
955 #####
956 #####
957
958 #####
959 #####
960
961 #####
962 #####
963
964 #####
965 #####
966
967 #####
968 #####
969
970 #####
971 #####
972
973 #####
974 #####
975
976 #####
977 #####
978
979 #####
980 #####
981
982 #####
983 #####
984
985 #####
986 #####
987
988 #####
989 #####
990
991 #####
992 #####
993
994 #####
995 #####
996
997 #####
998 #####
999
1000 #####

```

```

152         # If test mode return early
153         if mode == 'test':
154             return scores
155
156         loss, grads = 0.0, {}
157
158         #####
159         # TODO: Implement the backward pass for the fully-connected net.
160         # Store the #
161         # loss in the loss variable and gradients in the grads
162         # dictionary. Compute #
163         # data loss using softmax, and make sure that grads[k] holds the
164         # gradients #
165         # for self.params[k]. Don't forget to add L2 regularization!
166         #
167         # When using batch normalization, you don't need to regularize
168         # the scale #
169         # and shift parameters.
170         #
171         # NOTE: To ensure that your implementation matches ours and you
172         # pass the #
173         # automated tests, make sure that your L2 regularization
174         # includes a factor #
175         # of 0.5 to simplify the expression for the gradient.
176         #
177         #####
178
179         loss, dscores = softmax_loss(scores, y)
180         dhout = dscores
181         loss = loss + 0.5 * self.reg * np.sum(
182             self.params['W%d' % (self.num_layers)] * self.params['W%d' %
183             (self.num_layers)])
184         dx, dw, db = affine_backward(dhout, ar_cache[self.num_layers])
185         grads['W%d' % (self.num_layers)] = dw + self.reg *
186         self.params['W%d' % (self.num_layers)]
187         grads['b%d' % (self.num_layers)] = db
188         dhout = dx
189         for idx in range(self.num_layers - 1):
190             lay = self.num_layers - 1 - idx - 1

```

```

180         loss = loss + 0.5 * self.reg * np.sum(self.params['W%d' %
(181         (lay + 1)] * self.params['W%d' % (lay + 1)])
182         if self.use_dropout:
183             dhout = dropout_backward(dhout, dp_cache[lay])
184         if self.use_batchnorm:
185             dx, dw, db, dgamma, dbeta =
186             affine_bn_relu_backward(dhout, ar_cache[lay])
187         else:
188             dx, dw, db = affine_relu_backward(dhout, ar_cache[lay])
189             grads['W%d' % (lay + 1)] = dw + self.reg * self.params['W%d'
190             % (lay + 1)]
191             grads['b%d' % (lay + 1)] = db
192             if self.use_batchnorm:
193                 grads['gamma%d' % (lay + 1)] = dgamma
194                 grads['beta%d' % (lay + 1)] = dbeta
195             dhout = dx
196         # pass
197
198         #####
199         #####
200         #                                     END OF YOUR CODE
201         #
202         #####
203         #####
204
205         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：

```

1  N, D, H1, H2, C = 2, 15, 20, 30, 10
2  X = np.random.randn(N, D)
3  y = np.random.randint(C, size=(N,))
4
5  for reg in [0, 3.14]:
6      print ('Running check with reg = ', reg)
7      model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
8                                reg=reg, weight_scale=5e-2, dtype=np.float64)
9
10     loss, grads = model.loss(X, y)
11     print ('Initial loss: ', loss)
12

```

```

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

```

运行结果如下：

```

1 Running check with reg = 0
2 Initial loss: 2.30497257983
3 W1 relative error: 1.54e-06
4 W2 relative error: 2.86e-07
5 W3 relative error: 2.27e-06
6 b1 relative error: 4.41e-08
7 b2 relative error: 1.33e-09
8 b3 relative error: 1.57e-10
9 Running check with reg = 3.14
10 Initial loss: 6.93678451752
11 W1 relative error: 3.74e-08
12 W2 relative error: 2.29e-06
13 W3 relative error: 1.87e-08
14 b1 relative error: 2.71e-08
15 b2 relative error: 6.51e-09
16 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：

```

1 num_train=50
2 small_data={
3     'X_train':data['X_train'][:num_train],
4     'y_train':data['y_train'][:num_train],
5     'X_val':data['X_val'],
6     'y_val':data['y_val'],
7 }
8

```

```

9 weight_scale=1e-2
10 learning_rate=8e-3
11 model=FullyConnectedNet([100,100],weight_scale=weight_scale,dtype=np.float64)
12 solver=Solver(model,small_data,print_every=10,num_epochs=20,batch_size=25,update_rule='sgd',optim_config={'learning_rate':learning_rate,})
13 solver.train()
14
15 plt.plot(solver.loss_history,'-o')
16 plt.title('training loss history')
17 plt.xlabel('iteration')
18 plt.ylabel('training loss')
19 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。

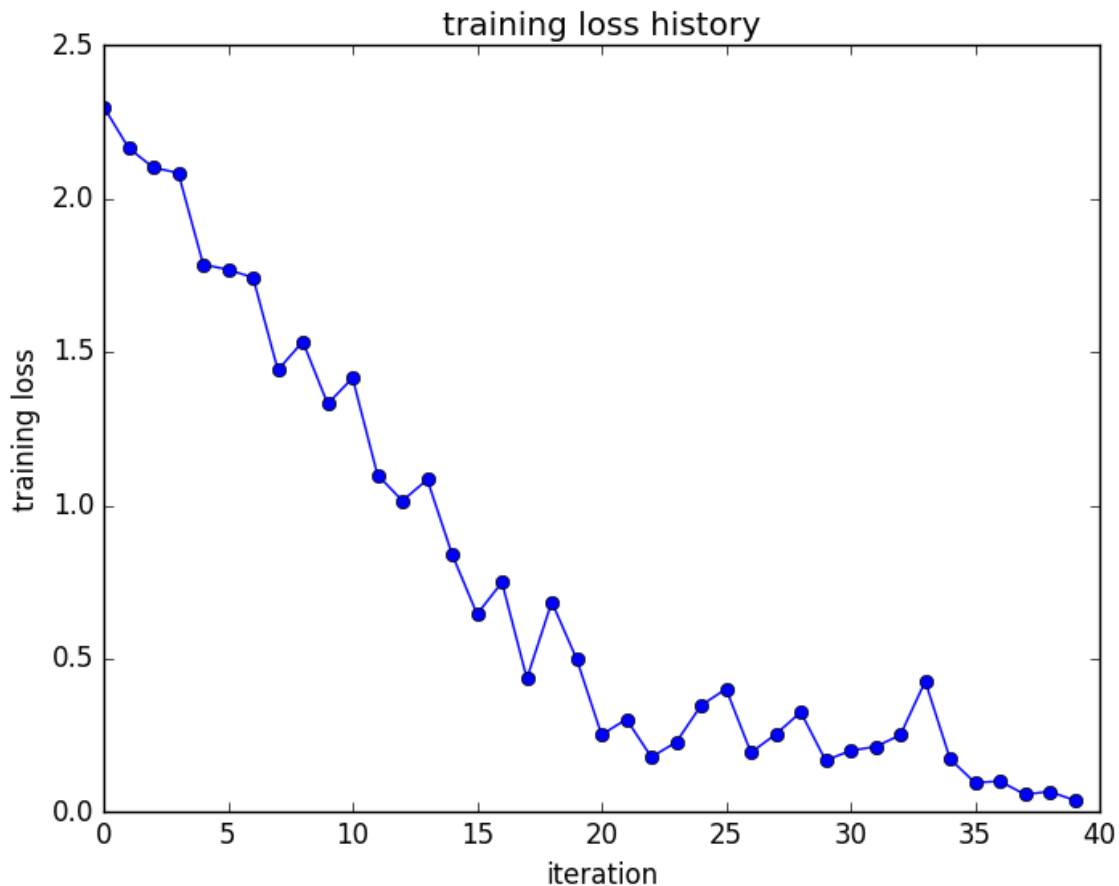
输出结果如下：

```

1 (Iteration 1 / 40) loss: 2.300085
2 (Epoch 0 / 20) train acc: 0.300000; val_acc: 0.149000
3 (Epoch 1 / 20) train acc: 0.380000; val_acc: 0.162000
4 (Epoch 2 / 20) train acc: 0.440000; val_acc: 0.155000
5 (Epoch 3 / 20) train acc: 0.480000; val_acc: 0.167000
6 (Epoch 4 / 20) train acc: 0.460000; val_acc: 0.168000
7 (Epoch 5 / 20) train acc: 0.540000; val_acc: 0.195000
8 (Iteration 11 / 40) loss: 1.415692
9 (Epoch 6 / 20) train acc: 0.780000; val_acc: 0.197000
10 (Epoch 7 / 20) train acc: 0.840000; val_acc: 0.195000
11 (Epoch 8 / 20) train acc: 0.760000; val_acc: 0.177000
12 (Epoch 9 / 20) train acc: 0.880000; val_acc: 0.208000
13 (Epoch 10 / 20) train acc: 0.900000; val_acc: 0.202000
14 (Iteration 21 / 40) loss: 0.252448
15 (Epoch 11 / 20) train acc: 0.960000; val_acc: 0.196000
16 (Epoch 12 / 20) train acc: 0.940000; val_acc: 0.189000
17 (Epoch 13 / 20) train acc: 0.880000; val_acc: 0.193000
18 (Epoch 14 / 20) train acc: 0.960000; val_acc: 0.188000
19 (Epoch 15 / 20) train acc: 1.000000; val_acc: 0.192000
20 (Iteration 31 / 40) loss: 0.200646
21 (Epoch 16 / 20) train acc: 1.000000; val_acc: 0.202000
22 (Epoch 17 / 20) train acc: 1.000000; val_acc: 0.193000
23 (Epoch 18 / 20) train acc: 1.000000; val_acc: 0.190000
24 (Epoch 19 / 20) train acc: 1.000000; val_acc: 0.196000
25 (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

```
1 num_train=50
2 small_data={
3     'X_train':data['X_train'][:num_train],
4     'y_train':data['y_train'][:num_train],
5     'X_val':data['X_val'],
6     'y_val':data['y_val'],
7 }
8
9 weight_scale=1e-2
10 learning_rate=3e-4
11 model=FullyConnectedNet([100,100,100,100],weight_scale=weight_scale,dtype
    =np.float64)
12 solver=Solver(model,small_data,print_every=10,num_epochs=20,batch_size=25
    ,update_rule='sgd',optim_config={'learning_rate':learning_rate,})
13 solver.train()
14
15 plt.plot(solver.loss_history,'-o')
16 plt.title('training loss history')
```



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

运行结果如下：

```
1 (Iteration 1 / 40) loss: 97.854722
2 (Epoch 0 / 20) train acc: 0.200000; val_acc: 0.114000
3 (Epoch 1 / 20) train acc: 0.300000; val_acc: 0.104000
4 (Epoch 2 / 20) train acc: 0.380000; val_acc: 0.107000
5 (Epoch 3 / 20) train acc: 0.660000; val_acc: 0.090000
6 (Epoch 4 / 20) train acc: 0.720000; val_acc: 0.103000
7 (Epoch 5 / 20) train acc: 0.780000; val_acc: 0.098000
8 (Iteration 11 / 40) loss: 0.731547
9 (Epoch 6 / 20) train acc: 0.780000; val_acc: 0.097000
10 (Epoch 7 / 20) train acc: 0.940000; val_acc: 0.099000
11 (Epoch 8 / 20) train acc: 0.940000; val_acc: 0.098000
12 (Epoch 9 / 20) train acc: 0.960000; val_acc: 0.109000
13 (Epoch 10 / 20) train acc: 0.960000; val_acc: 0.105000
14 (Iteration 21 / 40) loss: 2.570933
15 (Epoch 11 / 20) train acc: 0.960000; val_acc: 0.102000
16 (Epoch 12 / 20) train acc: 0.960000; val_acc: 0.102000
17 (Epoch 13 / 20) train acc: 1.000000; val_acc: 0.106000
18 (Epoch 14 / 20) train acc: 1.000000; val_acc: 0.106000
19 (Epoch 15 / 20) train acc: 1.000000; val_acc: 0.106000
20 (Iteration 31 / 40) loss: 0.000013
21 (Epoch 16 / 20) train acc: 1.000000; val_acc: 0.106000
22 (Epoch 17 / 20) train acc: 1.000000; val_acc: 0.106000
23 (Epoch 18 / 20) train acc: 1.000000; val_acc: 0.106000
24 (Epoch 19 / 20) train acc: 1.000000; val_acc: 0.106000
25 (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：

```
1 def sgd_momentum(w, dw, config=None):
2
3     #Performs stochastic gradient descent with momentum.
```

```

4     #config format:
5     #- learning_rate: Scalar learning rate.
6     #- momentum: Scalar between 0 and 1 giving the momentum value.
7     # Setting momentum = 0 reduces to sgd.
8     #- velocity: A numpy array of the same shape as w and dw used to
store a moving
9     # average of the gradients.
10
11     if config is None: config = {}
12     config.setdefault('learning_rate', 1e-2)
13     config.setdefault('momentum', 0.9)
14     v = config.get('velocity', np.zeros_like(w))
15
16     next_w = None
17
18     #####
19     # TODO: Implement the momentum update formula. Store the updated
value in #
20     # the next_w variable. You should also use and update the velocity v.
21     #
22     #####
23     v = config['momentum'] * v - config['learning_rate'] * dw
24     next_w = w + v
25     # pass
26
27     #####
28     #
29     #
30     #####
31     #
32     #
33     #####
34     config['velocity'] = v
35
36     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：

```

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

```

```

3 dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
4 v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
5
6 config = {'learning_rate': 1e-3, 'velocity': v}
7 next_w, _ = sgd_momentum(w, dw, config=config)
8
9 expected_next_w = np.asarray([
10     [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
11     [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
12     [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
13     [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096      ]])
14 expected_velocity = np.asarray([
15     [ 0.5406,      0.55475789,  0.56891579,  0.58307368,  0.59723158],
16     [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
17     [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
18     [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096      ]])

```

结果如下：

```

1 next_w error: 8.88234703351e-09
2 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：

```

1 def rmsprop(x, dx, config=None):
2
3     #Uses the RMSProp update rule, which uses a moving average of squared
    gradient
4     #values to set adaptive per-parameter learning rates.
5     #config format:
6     #- learning_rate: Scalar learning rate.
7     #- decay_rate: Scalar between 0 and 1 giving the decay rate for the
    squared
8     # gradient cache.

```

```

9      #- epsilon: Small scalar used for smoothing to avoid dividing by
zero.
10     #- cache: Moving average of second moments of gradients.
11
12     if config is None: config = {}
13     config.setdefault('learning_rate', 1e-2)
14     config.setdefault('decay_rate', 0.99)
15     config.setdefault('epsilon', 1e-8)
16     config.setdefault('cache', np.zeros_like(x))
17
18     next_x = None
19
20     #####
21     # TODO: Implement the RMSprop update formula, storing the next value
of x  #
22     # in the next_x variable. Don't forget to update cache value stored
in    #
23     # config['cache'].
    #
24     #####
25     config['cache'] = config['decay_rate'] * config['cache'] + (1 -
config['decay_rate']) * (dx ** 2)
26     next_x = x - config['learning_rate'] * dx / (np.sqrt(config['cache'])
+ config['epsilon'])
27     # pass
28
29     #####
30     #
    #
31     #####
32
33     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：

```

1 def adam(x, dx, config=None):
2
3     #Uses the Adam update rule, which incorporates moving averages of
    both the
4     #gradient and its square and a bias correction term.
5     #config format:
6     #- learning_rate: Scalar learning rate.
7     #- beta1: Decay rate for moving average of first moment of gradient.
8     #- beta2: Decay rate for moving average of second moment of gradient.
9     #- epsilon: Small scalar used for smoothing to avoid dividing by
    zero.
10    #- m: Moving average of gradient.
11    #- v: Moving average of squared gradient.
12    #- t: Iteration number.
13
14    if config is None: config = {}
15    config.setdefault('learning_rate', 1e-3)
16    config.setdefault('beta1', 0.9)
17    config.setdefault('beta2', 0.999)
18    config.setdefault('epsilon', 1e-8)
19    config.setdefault('m', np.zeros_like(x))
20    config.setdefault('v', np.zeros_like(x))
21    config.setdefault('t', 0)
22
23    next_x = None
24
25    #####
26    # TODO: Implement the Adam update formula, storing the next value of
    x in #
27    # the next_x variable. Don't forget to update the m, v, and t
    variables #
28    # stored in config.
29    #
30    #####
31
32    config['t'] += 1
33    config['m'] = config['beta1'] * config['m'] + (1 - config['beta1']) *
    dx
34    config['v'] = config['beta2'] * config['v'] + (1 - config['beta2']) *
    (dx ** 2)
35    mb = config['m'] / (1 - config['beta1'] ** config['t'])
36    vb = config['v'] / (1 - config['beta2'] ** config['t'])

```

```

34     next_x = x - config['learning_rate'] * mb / (np.sqrt(vb) +
config['epsilon'])
35     # pass
36
#####
37     #                                END OF YOUR CODE
#
38
#####
39
40     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：

```

1  num_train=4000
2  small_data={
3      'X_train':data['X_train'][:num_train],
4      'y_train':data['y_train'][:num_train],
5      'X_val':data['X_val'],
6      'y_val':data['y_val'],
7  }
8  solvers={}
9
10 learning_rates={'sgd':1e-2,'sgd_momentum':1e-2,'rmsprop':1e-4,'adam':1e-3}
11 for update_relu in ['sgd','sgd_momentum','rmsprop','adam']:
12     print('running with',update_relu)
13
14     model=FullyConnectedNet([100,100,100,100,100],weight_scale=5e-2)
15
16     solver=Solver(model,small_data,num_epochs=5,batch_size=100,update_rule=update_relu,
17                   optim_config=
18                   {'learning_rate':learning_rates[update_relu]},verbose=True)
19     solvers[update_relu]=solver

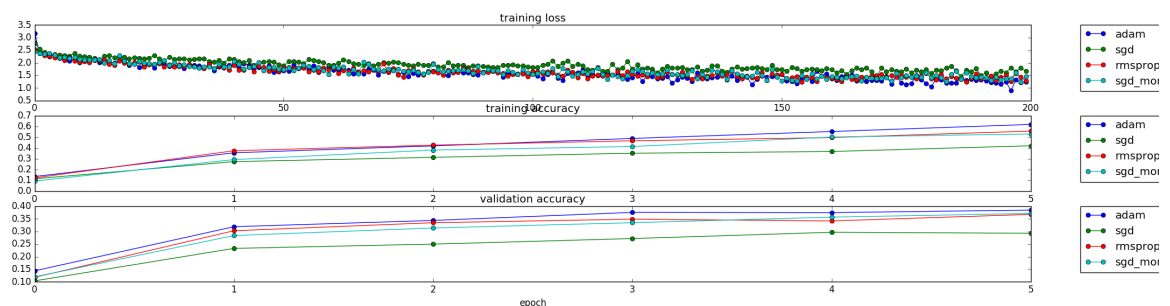
```

```

19     solver.train()
20     print()
21
22     plt.subplot(311)
23     plt.title('training loss')
24     plt.xlabel('iteration')
25
26     plt.subplot(312)
27     plt.title('training accuracy')
28     plt.xlabel('epoch')
29
30     plt.subplot(313)
31     plt.title('validation accuracy')
32     plt.xlabel('epoch')
33
34     for update_relu,solver in solvers.items():
35         plt.subplot(311)
36         plt.plot(solver.loss_history,'-o',label=update_relu)
37         plt.legend( bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
38
39         plt.subplot(312)
40         plt.plot(solver.train_acc_history,'-o',label=update_relu)
41         plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
42
43         plt.subplot(313)
44         plt.plot(solver.val_acc_history,'-o',label=update_relu)
45         plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
46
47
48
49 plt.gcf().set_size_inches(15,15)
50 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：

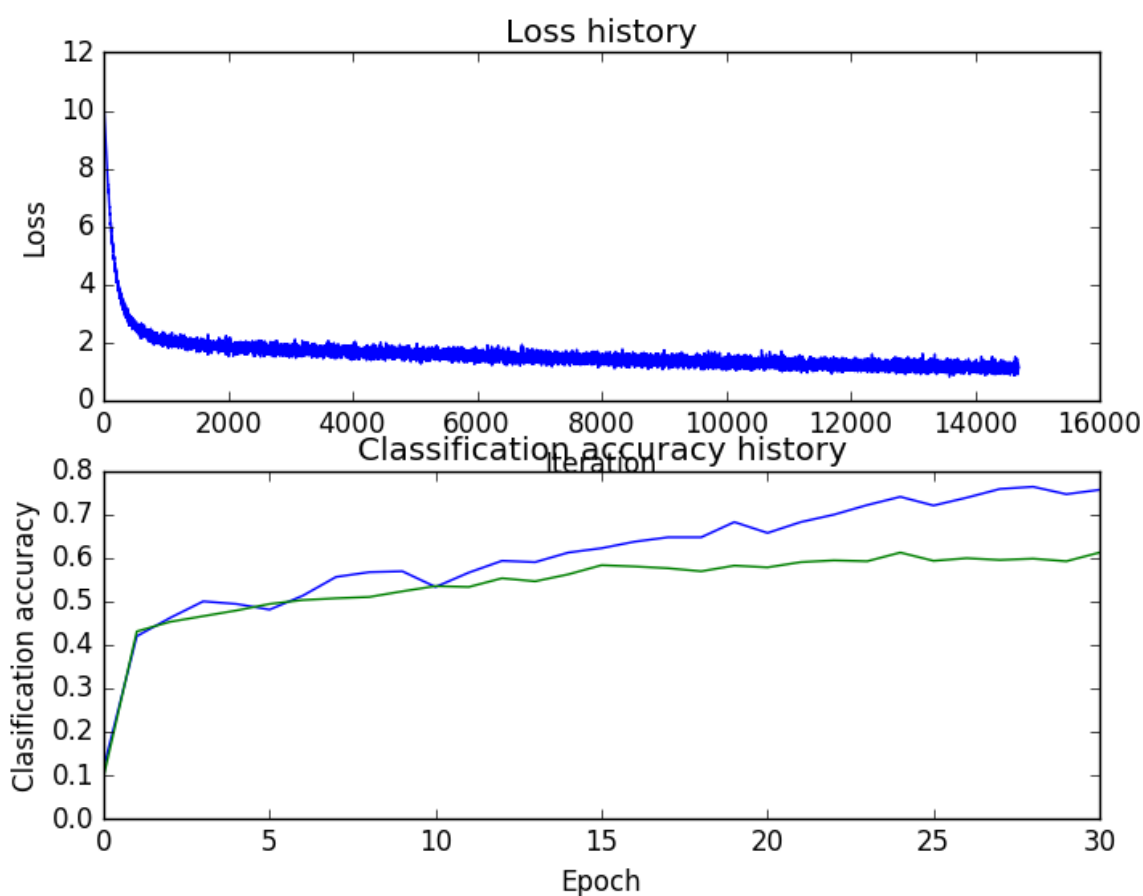
```
1 best_model = None
2 #####
3 # TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You
   might #
4 # batch normalization and dropout useful. Store your best model in the
   #
5 # best_model variable.
   #
6 #####
7 X_val= data['X_val']
8 y_val= data['y_val']
9 X_test= data['X_test']
10 y_test= data['y_test']
11
12 learning_rate = 3.1e-4
13 weight_scale = 2.5e-2 #1e-5
14 model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
15                             weight_scale=weight_scale, dtype=np.float64,
16                             dropout=0.25, use_batchnorm=True, reg=1e-2)
17 solver = Solver(model, data,
18                 print_every=500, num_epochs=30, batch_size=100,
19                 update_rule='adam',
20                 optim_config={
21                     'learning_rate': learning_rate,
22                 },
23                 lr_decay=0.9
24             )
25 solver.train()
26 scores = model.loss(data['X_test'])
27 y_pred = np.argmax(scores, axis = 1)
28 acc = np.mean(y_pred == data['y_test'])
29 print ('test acc: %f' %(acc))
30 best_model = model
31
32 plt.subplot(2, 1, 1)
33 plt.plot(solver.loss_history)
34 plt.title('Loss history')
```

```

35 plt.xlabel('Iteration')
36 plt.ylabel('Loss')
37
38 plt.subplot(2, 1, 2)
39 plt.plot(solver.train_acc_history, label='train')
40 plt.plot(solver.val_acc_history, label='val')
41 plt.title('Classification accuracy history')
42 plt.xlabel('Epoch')
43 plt.ylabel('Clasification accuracy')
44 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：

```

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

```

输出结果如下：

```
1 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。