Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer and Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

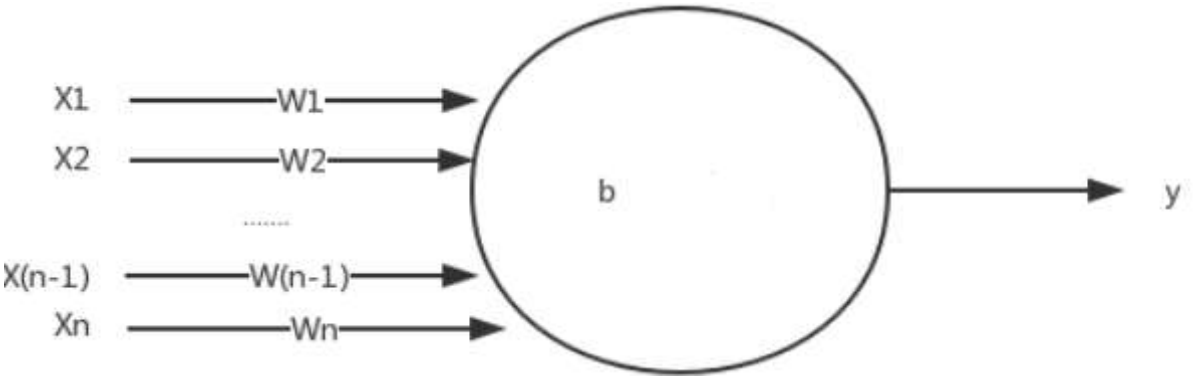
Affine layer: foward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

解释：

输入为(x,w,b), 意义如下图。



那么在不加激活函数的情况下输出为

```
$$ y = \sum_1^n{x_i*w_i}+b$$
```

则代码如下：

```
###########################################################################
# TODO: Implement the affine forward pass. Store the result in out. You   #
# will need to reshape the input into rows.                               #
###########################################################################
N = x.shape[0]
X = x.reshape(N, -1) #传进来的x这个多维数组重新组成N*D的矩阵，这样与D*M的w相乘得到上面表达式的结果
out = X.dot(w) + b
###########################################################################
    #                             END OF YOUR CODE                            #
###########################################################################
```

```
# Test the affine_forward function

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]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

解释：

1) 对输出层(第l层)，计算残差：

$\delta^{(l)} = \frac{\partial J(W,b)}{\partial z^{(l)}}$（不同损失函数，结果不同，这里不给出具体形式）

3) 对于l-1, l-2 , ... , 2的隐藏层，计算：

$\delta^{(l)} = \frac{\partial J}{\partial z^{(l)}} = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} = ((W^{(l)})^T \delta^{(l+1)}) \cdot f'(z^{(l)})$ 4) 计算各层参数W{(l)}、b{(l)}偏导数：

$\nabla_{W^{(l)}} J(W,b) = \frac{\partial J(W,b)}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial W^{(l)}} = \delta^{(l+1)} (a^{(l)})^T \nabla_{b^{(l)}} J(W,b) = \frac{\partial J(W,b)}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial b^{(l)}} = \delta^{(l+1)}$

所以编程实现的时候，先要分清楚该层的输入、输出即能正确编程实现,如：

$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$

$a^{(l+1)} = f(z^{(l+1)})$ 注意$out_diff = \frac{\partial J}{\partial z^{(l+1)}}$ 是上一层（Softmax 或 Sigmoid/ReLU的 in_diff）已经求得的，对于本层有：

$in\_diff = \frac{\partial J}{\partial a^{(l)}} = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial a^{(l)}} = W^T * out\_diff$

$W\_diff = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial W^{(l)}} = out\_diff * in^T$

$b\_diff = \frac{\partial J}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial b^{(l)}} = out\_diff * 1$

代码如下：

```
N = x.shape[0]
X = x.reshape(N, -1)
dx = dout.dot(w.T)   # (N,D)
dx = dx.reshape(x.shape) # (N,d1,...,d_k)
dw = X.T.dot(dout)   # (D, M)
db = np.sum(dout, axis=0) # (1,M)
```

```
# Test the affine_backward function
np.random.seed(231)
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)

# The error should be around e-10 or less
print('Testing 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))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

解释：

ReLU 这个函数比较简单，如下： $f(x) = max(x, 0)$ 所以对于前向传播： 就是将输出中小于0的值变成0就行了。

对于反向传播： 在数学上ReLU这个函数在x=0处不可微，但是在深度学习中我们将其函数置为1.所以有

```
$$（x>=0): f^/(x)=1$$
$$ (x<0): f^/(x)=0$$
```

上式的值再乘上上一层传过来的 derivatives 即可。 所以代码有： 前向传播：

```
  out = np.maximum(0, x)
```

反向传播：

```
 dx = dout * (x >= 0)
```

```
# Test the relu_forward function

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,       ]])

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

```
np.random.seed(231)
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)

# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

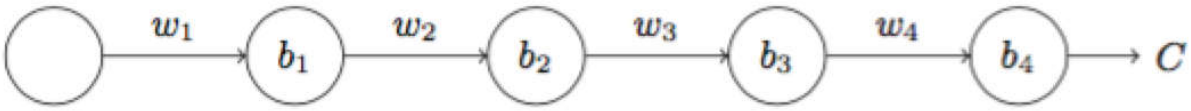
Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

1. Sigmoid
2. ReLU
3. Leaky ReLU

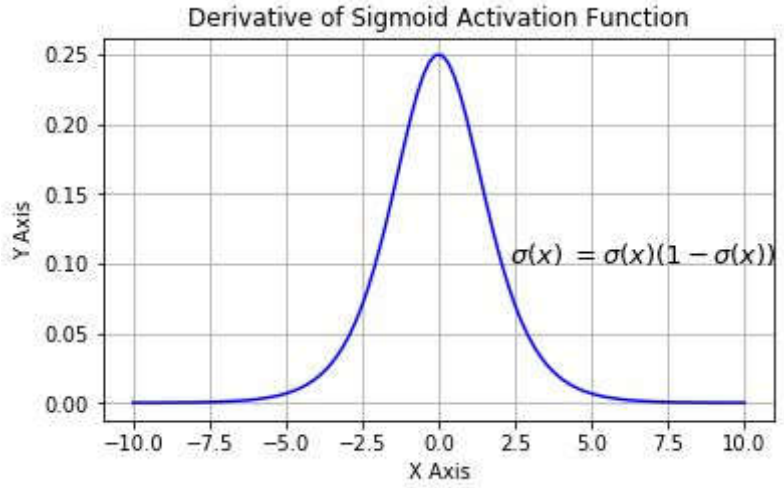
Answer:

Sigmoid函数 会导致梯度消失问题。考虑如下图的一维神经网络



这里，w1,w2...是权重，而b1,b2,...是偏置，C是某个loss function。第j个神经元的输出$a_j = \sigma(z_j)$,其中σ是Sigmoid激活函数，而$z_j = w_j * a_{j-1} + b_j$是神经元的带权输出。这里C是强调网络输出$a_4$的代价函数。如果实际输出越接近目标输出，那么代价会变低；相反则变高。

我们将整个网络的梯度表达式写出来：$\frac{\partial C}{\partial b_1} = \sigma'(z_1)w_2\sigma'(z_2)w_3\sigma'(z_3)w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}$ 除了最后一项，这个表达式是一系列形如$w_j\sigma'(z_j)$的乘积。为了理解每一项的行为，先看下面的sigmoid函数导数的图像：



导数在$\sigma'(0) = 1/4$时达到最高。现在如果我们使用标准方法来初始化网络权重，那么会使用一个均值为0标准差为1的高斯分布。因为所有权重|w|<1。那么$w_j{\sigma}^/(z_j)<1/4$，那么所有这些项乘起来最终结果会指数级下降到0，这就是梯度消失的原因。只有当输入可以满足让w在训练时能增加，使得$w_j{\sigma}^/(z_j)>1$才不会出现梯度消失现象。

综上，从导数图像我们可以看出来当输入x过大或者过小就会出现梯度消失的现象

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
import numpy as np
np.random.seed(231)
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)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs231n/layers.py`.

You can make sure that the implementations are correct by running the following:

```
np.random.seed(231)
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)

# Test svm_loss function. Loss should be around 9 and dx error should be around the order of e-9
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)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

解释：

这题要求用前面实现的接口实现两层神经网络，两层神经网络的结构是 输入x ->(第一层+ReLU)->（输出层，不加激活函数）

初始化权重和阈值部分没什么好说，按要求初始化就行。

1.对于前向传播部分，直接调用一次有ReLU的前向传播函数和一次没有ReLu的前向传播函数即可。 代码：

```
layer1_out,layer1_cache = affine_relu_forward(X,self.params['W1'],self.params['b1'])
layer2_out,layer2_cache = affine_forward(layer1_out,self.params['W2'],self.params['b2'])
scores = layer2_out
```

对于计算loss， 这里要求除了调用softmax_loss这个已有的函数计算出softmax的loss之外，还要加入L2 正则化（权重衰减）。 L2正则化是为了解决过拟合现象的手段。L2正则化就是在代价函数后面再加上一个正则化项：

$C = C_0 + \frac{\lambda}{2n}\sum_w W^2$ C0代表原始的代价函数，后面那一项就是L2正则化项，它是这样来的：所有参数w的平方的和，除以训练集的样本大小n。λ就是正则项系数，权衡正则项与C0项的比重。另外还有一个系数1/2，1/2经常会看到，主要是为了后面求导的结果方便，后面那一项求导会产生一个2，与1/2相乘刚好凑整。 求导后如下：

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}.$$

可以看到只对w有影响。

W更新公式：

$$\begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w \\ &= \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}. \end{aligned}$$

所以接下来求loss 和反向传播的代码如下，注意调用反向传播的结果，输出层的是调用没有激活函数版的，第一层调用的是有relu激活版的反向传播函数：

```
loss, dx_loss = 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'])  #loss后+的一段就是正则化项

d1_out, dw2, db2 = affine_backward(dx_loss, layer2_cache)
grads['W2'] = dw2 + self.reg * self.params['W2']       # 这里正则化的梯度只对W更新有影响，下同。
grads['b2'] = db2

dx1, dw1, db1 = affine_relu_backward(d1_out, layer1_cache)
grads['W1'] = dw1 + self.reg * self.params['W1']
grads['b1'] = db1
```

```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
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'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)


  for name in sorted(grads):
```

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

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.52e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 3.12e-07
W2 relative error: 7.98e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

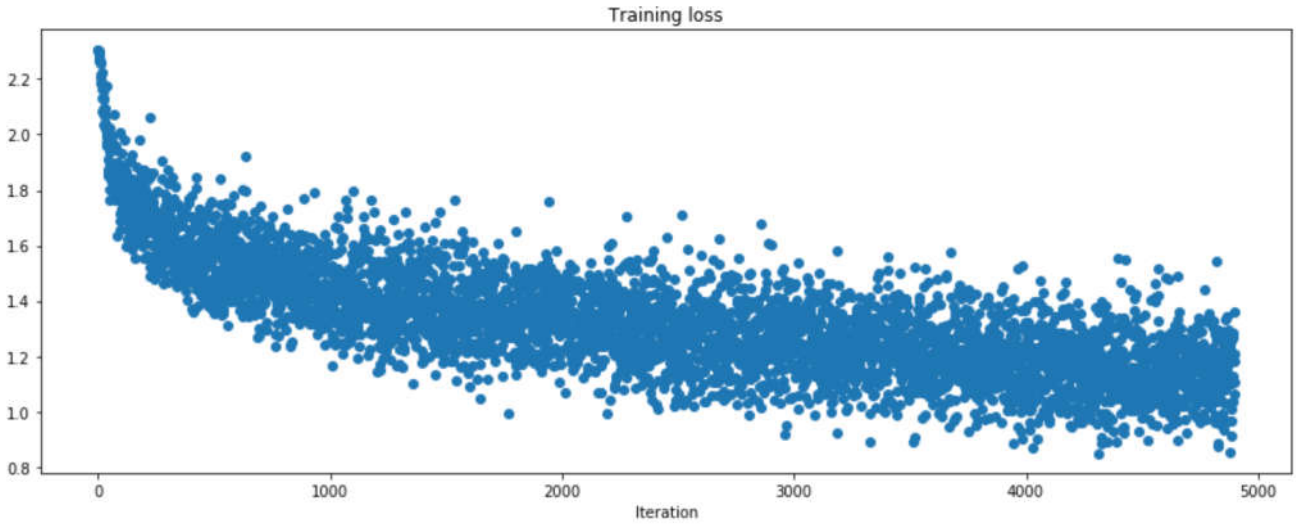
Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

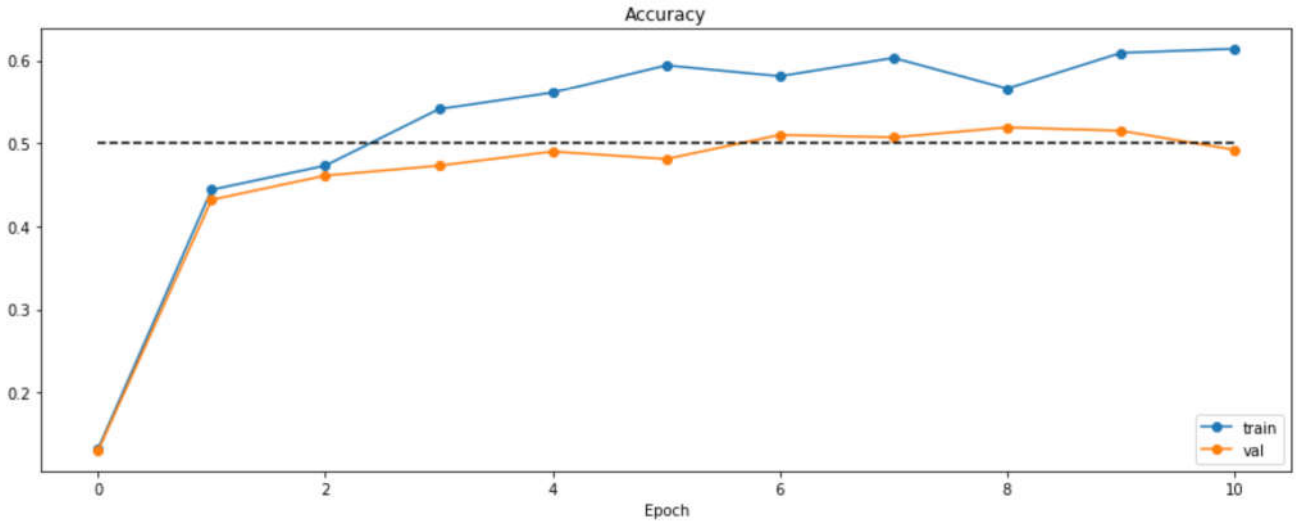
```
model = TwoLayerNet()
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                    'learning_rate': 1e-3,},
                lr_decay=0.95,
                num_epochs=10, batch_size=100,
                print_every=100)
solver.train()
scores = solver.model.loss(data['X_test'])
y_pred = np.argmax(scores, axis=1)
acc = np.mean(y_pred == data['y_test'])
print("test acc: ",acc)
##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
```

```
(Iteration 1 / 4900) loss: 2.304305
(Epoch 0 / 10) train acc: 0.132000; val_acc: 0.130000
(Iteration 101 / 4900) loss: 1.862001
(Iteration 201 / 4900) loss: 1.755473
(Iteration 301 / 4900) loss: 1.872329
(Iteration 401 / 4900) loss: 1.466895
(Epoch 1 / 10) train acc: 0.444000; val_acc: 0.432000
(Iteration 501 / 4900) loss: 1.621252
(Iteration 601 / 4900) loss: 1.542340
(Iteration 701 / 4900) loss: 1.444864
(Iteration 801 / 4900) loss: 1.431592
(Iteration 901 / 4900) loss: 1.401742
(Epoch 2 / 10) train acc: 0.473000; val_acc: 0.461000
(Iteration 1001 / 4900) loss: 1.420819
(Iteration 1101 / 4900) loss: 1.551001
(Iteration 1201 / 4900) loss: 1.145786
(Iteration 1301 / 4900) loss: 1.174424
(Iteration 1401 / 4900) loss: 1.199727
(Epoch 3 / 10) train acc: 0.541000; val_acc: 0.473000
(Iteration 1501 / 4900) loss: 1.327703
(Iteration 1601 / 4900) loss: 1.221111
(Iteration 1701 / 4900) loss: 1.286650
(Iteration 1801 / 4900) loss: 1.167255
(Iteration 1901 / 4900) loss: 1.299780
(Epoch 4 / 10) train acc: 0.561000; val_acc: 0.490000
(Iteration 2001 / 4900) loss: 1.462216
(Iteration 2101 / 4900) loss: 1.268536
(Iteration 2201 / 4900) loss: 1.500084
(Iteration 2301 / 4900) loss: 1.355347
(Iteration 2401 / 4900) loss: 1.298523
(Epoch 5 / 10) train acc: 0.594000; val_acc: 0.481000
(Iteration 2501 / 4900) loss: 1.211618
(Iteration 2601 / 4900) loss: 1.566974
(Iteration 2701 / 4900) loss: 1.121811
(Iteration 2801 / 4900) loss: 1.209136
(Iteration 2901 / 4900) loss: 1.376604
(Epoch 6 / 10) train acc: 0.581000; val_acc: 0.510000
(Iteration 3001 / 4900) loss: 1.451466
(Iteration 3101 / 4900) loss: 1.048938
(Iteration 3201 / 4900) loss: 1.292317
(Iteration 3301 / 4900) loss: 1.134088
(Iteration 3401 / 4900) loss: 1.274798
(Epoch 7 / 10) train acc: 0.603000; val_acc: 0.507000
(Iteration 3501 / 4900) loss: 1.193176
(Iteration 3601 / 4900) loss: 1.029356
(Iteration 3701 / 4900) loss: 0.981899
(Iteration 3801 / 4900) loss: 1.045425
(Iteration 3901 / 4900) loss: 1.345838
(Epoch 8 / 10) train acc: 0.566000; val_acc: 0.519000
(Iteration 4001 / 4900) loss: 1.206046
(Iteration 4101 / 4900) loss: 1.187770
(Iteration 4201 / 4900) loss: 1.168217
(Iteration 4301 / 4900) loss: 1.160490
```

```
(Iteration 4401 / 4900) loss: 1.145806
(Epoch 9 / 10) train acc: 0.609000; val_acc: 0.515000
(Iteration 4501 / 4900) loss: 1.144793
(Iteration 4601 / 4900) loss: 1.485305
(Iteration 4701 / 4900) loss: 1.084970
(Iteration 4801 / 4900) loss: 1.129719
(Epoch 10 / 10) train acc: 0.614000; val_acc: 0.492000
test acc:  0.508
```

```
# Run this cell to visualize training loss and train / val accuracy

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')
plt.gcf().set_size_inches(15, 12)
plt.show()
```





解释

这是两层神经网络跑出来的结果，我们可以看到loss在不断下降，训练集正确率在0.6左右，还有上升的趋势，但是val的正确率在0.5左右，有下降的趋势。这是因为我们的网络结构还比较简单，学习能力不够强，泛化能力也不够，所以在训练集和测试集的正确率都只有一半多一点。

Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization; we will add those features soon.

解释：

多层网络版，这个就是将两层神经网络的代码多加几层，最后一层为输出层不变，前面的层用数组管理即可。前向传播和反向传播过程都在前面说过了，这里不再重复。

Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-7 or less.

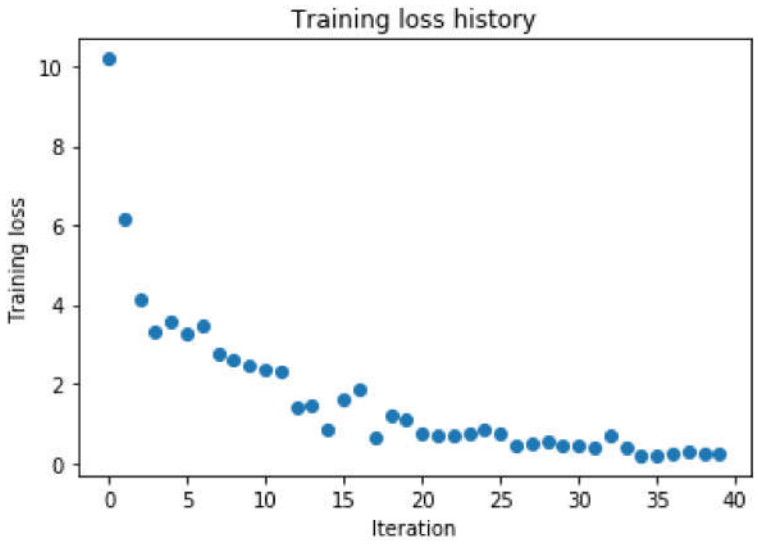
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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)

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  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])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 7.36e-09
W2 relative error: 6.87e-08
W3 relative error: 3.48e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the learning rate and initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

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 = 3e-2
learning_rate = 4e-4
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()
```

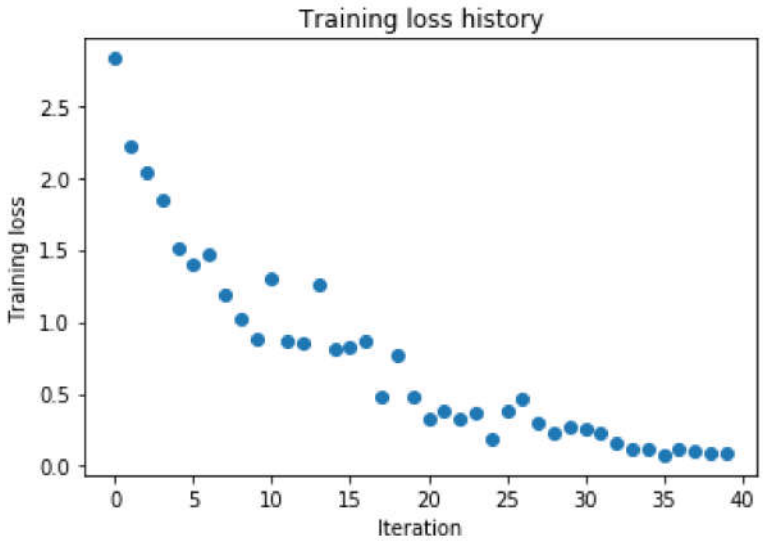
```
(Iteration 1 / 40) loss: 10.195766
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.115000
(Epoch 1 / 20) train acc: 0.200000; val_acc: 0.121000
(Epoch 2 / 20) train acc: 0.300000; val_acc: 0.133000
(Epoch 3 / 20) train acc: 0.280000; val_acc: 0.131000
(Epoch 4 / 20) train acc: 0.440000; val_acc: 0.138000
(Epoch 5 / 20) train acc: 0.480000; val_acc: 0.128000
(Iteration 11 / 40) loss: 2.385841
(Epoch 6 / 20) train acc: 0.580000; val_acc: 0.134000
(Epoch 7 / 20) train acc: 0.620000; val_acc: 0.142000
(Epoch 8 / 20) train acc: 0.660000; val_acc: 0.146000
(Epoch 9 / 20) train acc: 0.720000; val_acc: 0.143000
(Epoch 10 / 20) train acc: 0.780000; val_acc: 0.154000
(Iteration 21 / 40) loss: 0.763662
(Epoch 11 / 20) train acc: 0.780000; val_acc: 0.158000
(Epoch 12 / 20) train acc: 0.800000; val_acc: 0.154000
(Epoch 13 / 20) train acc: 0.840000; val_acc: 0.150000
(Epoch 14 / 20) train acc: 0.860000; val_acc: 0.156000
(Epoch 15 / 20) train acc: 0.920000; val_acc: 0.155000
(Iteration 31 / 40) loss: 0.418163
(Epoch 16 / 20) train acc: 0.920000; val_acc: 0.155000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.149000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.149000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.149000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.151000
```



Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

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'],
}

learning_rate = 1e-2
weight_scale = 4e-2
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()
```

```
(Iteration 1 / 40) loss: 2.826414
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.100000
(Epoch 1 / 20) train acc: 0.340000; val_acc: 0.135000
(Epoch 2 / 20) train acc: 0.520000; val_acc: 0.106000
(Epoch 3 / 20) train acc: 0.580000; val_acc: 0.096000
(Epoch 4 / 20) train acc: 0.660000; val_acc: 0.124000
(Epoch 5 / 20) train acc: 0.740000; val_acc: 0.151000
(Iteration 11 / 40) loss: 1.297868
(Epoch 6 / 20) train acc: 0.780000; val_acc: 0.138000
(Epoch 7 / 20) train acc: 0.720000; val_acc: 0.138000
(Epoch 8 / 20) train acc: 0.880000; val_acc: 0.163000
(Epoch 9 / 20) train acc: 0.900000; val_acc: 0.131000
(Epoch 10 / 20) train acc: 0.940000; val_acc: 0.174000
(Iteration 21 / 40) loss: 0.323658
(Epoch 11 / 20) train acc: 0.940000; val_acc: 0.152000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.162000
(Epoch 13 / 20) train acc: 0.880000; val_acc: 0.158000
(Epoch 14 / 20) train acc: 0.980000; val_acc: 0.151000
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.159000
(Iteration 31 / 40) loss: 0.254242
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.167000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.166000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.157000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.162000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.166000
```



解释：

这里要求我们调参使得 3层网络 和 5层网络的结果过拟合。所谓过拟合就是在训练集上有很高的正确率但是在测试集上正确率极低。

常见过拟合原因：

（1）建模样本选取有误，如样本数量太少，选样方法错误，样本标签错误等，导致选取的样本数据不足以代表预定的分类规则；

（2）样本噪音干扰过大，使得机器将部分噪音认为是特征从而扰乱了预设的分类规则；

（3）假设的模型无法合理存在，或者说是假设成立的条件实际并不成立；

（4）参数太多，模型复杂度过高；
（5）对于决策树模型，如果我们对于其生长没有合理的限制，其自由生长有可能使节点只包含单纯的事件数据(event)或非事件数据(no event)，使其虽然可以完美匹配（拟合）训练数据，但是无法适应其他数据集。
（6）对于神经网络模型：

```
  a)对样本数据可能存在分类决策面不唯一，随着学习的进行,, BP算法使权值可能收敛过于复杂的决策面；<br>
  b)权值学习迭代次数足够多(Overtraining)，拟合了训练数据中的噪声和训练样例中没有代表性的特征。<br>
```

解决方法:
（1）在神经网络模型中，可使用权值衰减的方法，即每次迭代过程中以某个小因子降低每个权值。
（2）选取合适的停止训练标准，使对机器的训练在合适的程度；
（3）保留验证数据集，对训练成果进行验证；
（4）获取额外数据进行交叉验证；
（5）正则化，即在进行目标函数或代价函数优化时，在目标函数或代价函数后面加上一个正则项，一般有L1正则与L2正则等。

在这两个例子中，迭代次数不变，那么我采取的策略是将学习率调大，这样网络的更新变快，更容易快速接近局部最优解。

Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

Answer:

5层网络比3层网络更敏感，因为层数更多，模型更复杂，参数空间更大。那么梯度消失的现象更容易出现。局部最优情况比3层网络要多。我们将weight_scale放大，这样可以让初始权重变大，能够搜索到更多的最优情况。

Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file `cs231n/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

解释：

如果把要优化的目标函数看成山谷的话，可以把要优化的参数看成滚下山的石头，参数随机化为一个随机数可以看做在山谷的某个位置以0速度开始往下滚。目标函数的梯度可以看做给石头施加的力，由力学定律知：**F=m∗a**，所以梯度与石头下滚的加速度成正比。因而，梯度直接影响速度，速度的累加得到石头的位置，对这个物理过程进行建模，可以得到参数更新过程为：

```
v = momentum * v - learning_rate * dx  # integrate velocity
x += v  # integrate position
```

代码中v指代速度，其计算过程中有一个超参数momentum，称为动量（momentum）。虽然名字为动量，其物理意义更接近于摩擦，其可以降低速度值，降低了系统的动能，防止石头在山谷的最底部不能停止情况的发生。动量的取值范围通常为[0.5, 0.9, 0.95, 0.99]，一种常见的做法是在迭代开始时将其设为0.5，在一定的迭代次数（epoch）后，将其值更新为0.99。

在实践中，一般采用SGD+momentum的配置，相比普通的SGD方法，这种配置通常能极大地加快收敛速度

按照上述公式直接实现代码如下：

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

```
from cs231n.optim import sgd_momentum

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    ]])

# Should see relative errors around e-8 or less
print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

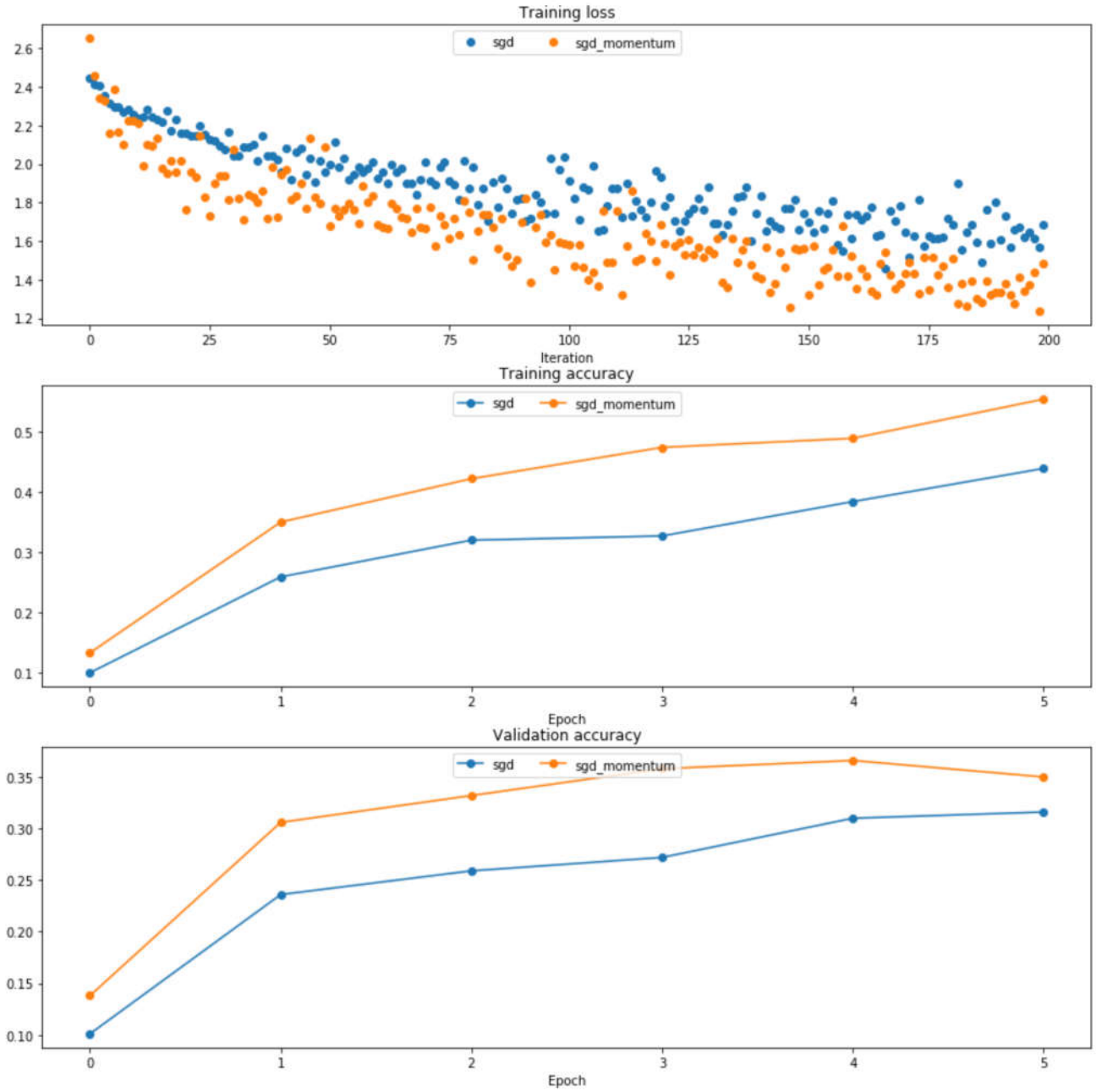
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

```
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 = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  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_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.443281
(Epoch 0 / 5) train acc: 0.100000; val_acc: 0.101000
(Iteration 11 / 200) loss: 2.236031
(Iteration 21 / 200) loss: 2.158782
(Iteration 31 / 200) loss: 2.042411
(Epoch 1 / 5) train acc: 0.259000; val_acc: 0.236000
(Iteration 41 / 200) loss: 1.958406
(Iteration 51 / 200) loss: 1.996430
(Iteration 61 / 200) loss: 1.926050
(Iteration 71 / 200) loss: 2.013198
(Epoch 2 / 5) train acc: 0.320000; val_acc: 0.259000
(Iteration 81 / 200) loss: 1.986519
(Iteration 91 / 200) loss: 1.821577
(Iteration 101 / 200) loss: 1.913309
(Iteration 111 / 200) loss: 1.875194
(Epoch 3 / 5) train acc: 0.327000; val_acc: 0.272000
(Iteration 121 / 200) loss: 1.785762
(Iteration 131 / 200) loss: 1.691236
(Iteration 141 / 200) loss: 1.832861
(Iteration 151 / 200) loss: 1.701464
(Epoch 4 / 5) train acc: 0.384000; val_acc: 0.310000
(Iteration 161 / 200) loss: 1.735170
(Iteration 171 / 200) loss: 1.644769
(Iteration 181 / 200) loss: 1.684796
(Iteration 191 / 200) loss: 1.605482
(Epoch 5 / 5) train acc: 0.439000; val_acc: 0.316000

running with  sgd_momentum
(Iteration 1 / 200) loss: 2.651982
(Epoch 0 / 5) train acc: 0.133000; val_acc: 0.138000
(Iteration 11 / 200) loss: 2.210676
(Iteration 21 / 200) loss: 1.764062
(Iteration 31 / 200) loss: 2.077861
(Epoch 1 / 5) train acc: 0.350000; val_acc: 0.306000
(Iteration 41 / 200) loss: 1.942479
(Iteration 51 / 200) loss: 1.680975
(Iteration 61 / 200) loss: 1.684854
(Iteration 71 / 200) loss: 1.664756
(Epoch 2 / 5) train acc: 0.422000; val_acc: 0.332000
(Iteration 81 / 200) loss: 1.505645
(Iteration 91 / 200) loss: 1.696198
(Iteration 101 / 200) loss: 1.580990
(Iteration 111 / 200) loss: 1.754374
(Epoch 3 / 5) train acc: 0.474000; val_acc: 0.358000
(Iteration 121 / 200) loss: 1.585163
(Iteration 131 / 200) loss: 1.534794
(Iteration 141 / 200) loss: 1.408238
(Iteration 151 / 200) loss: 1.323209
(Epoch 4 / 5) train acc: 0.489000; val_acc: 0.366000
(Iteration 161 / 200) loss: 1.352253
(Iteration 171 / 200) loss: 1.434986
(Iteration 181 / 200) loss: 1.509994
```

```
(Iteration 191 / 200) loss: 1.332932
(Epoch 5 / 5) train acc: 0.554000; val_acc: 0.350000
```

```
D:\Python\Anaconda3\lib\site-packages\matplotlib\cbook\deprecation.py:106:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new instance will always be
created and returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



解释

从上面的结果图我们可以看出，使用sgd+momentum的更新方法比只使用sgd的收敛速度要快得多

RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

解释：

RMSProp：
1.AdaGrad算法的改进。鉴于神经网络都是非凸条件下的，RMSProp在非凸条件下结果更好，改变梯度累积为指数衰减的移动平均以丢弃遥远的过去历史。
2.经验上，RMSProp被证明有效且实用的深度学习网络优化算法。
相比于AdaGrad的历史梯度： $\boldsymbol{r} \leftarrow \boldsymbol{r}+\boldsymbol{g} \odot \boldsymbol{g}$ RMSProp增加了一个衰减系数来控制历史信息的获取多少：
$\boldsymbol{r} \leftarrow \rho \boldsymbol{r}+(1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$ 算法如下：

算法 8.5 RMSProp 算法

Require: 全局学习率 ϵ，衰减速率 ρ

Require: 初始参数 $\boldsymbol{\theta}$

Require: 小常数 δ，通常设为 10^{-6}（用于被小数除时的数值稳定）

初始化累积变量 $\boldsymbol{r}=0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)} ; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \rho \boldsymbol{r}+(1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$ 与AdaGrad的唯一不同

计算参数更新：$\Delta \boldsymbol{\theta}=-\frac{\epsilon}{\sqrt{\delta+\boldsymbol{r}}} \odot \boldsymbol{g}$ （$\frac{1}{\sqrt{\delta+\boldsymbol{r}}}$ 逐元素应用）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}+\Delta \boldsymbol{\theta}$

end while

根据公式直接写代码就行，如下（其中config['cache']对应图中的累积变量r）：

```
config['cache'] = config['decay_rate'] * config['cache'] + (1 - config['decay_rate']) * (dw**2)
next_w = w - config['learning_rate'] * dw / (np.sqrt(config['cache']) + config['epsilon'])
```

Adam

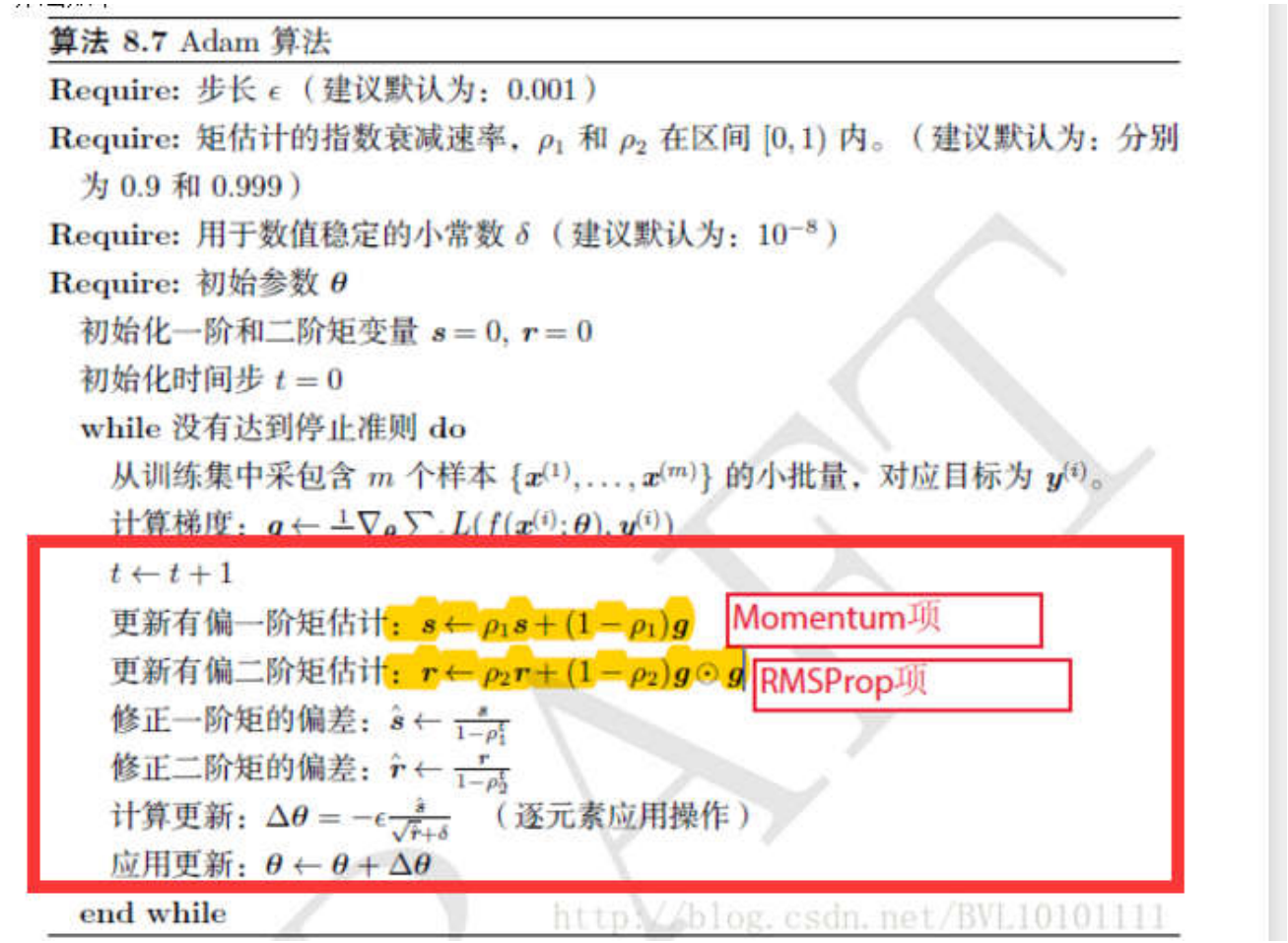
1.Adam算法可以看做是修正后的Momentum+RMSProp算法

2.动量直接并入梯度一阶矩估计中（指数加权）

3.Adam通常被认为对超参数的选择相当鲁棒

4.学习率建议为0.001

算法如下：



根据算法直接写出代码（对应上图红框）：

```
config['t'] += 1
config['m'] = config['beta1'] * config['m'] + (1 - config['beta1']) * dw
config['v'] = config['beta2'] * config['v'] + (1 - config['beta2']) * (dw**2)
mb = config['m'] / (1 - config['beta1']**config['t'])
vb = config['v'] / (1 - config['beta2']**config['t'])
next_w = w - config['learning_rate'] * mb / (np.sqrt(vb) + config['epsilon'])
```

```
# Test RMSProp implementation
from cs231n.optim import rmsprop

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)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

```
# Test Adam implementation
from cs231n.optim import adam

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)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))
```

```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

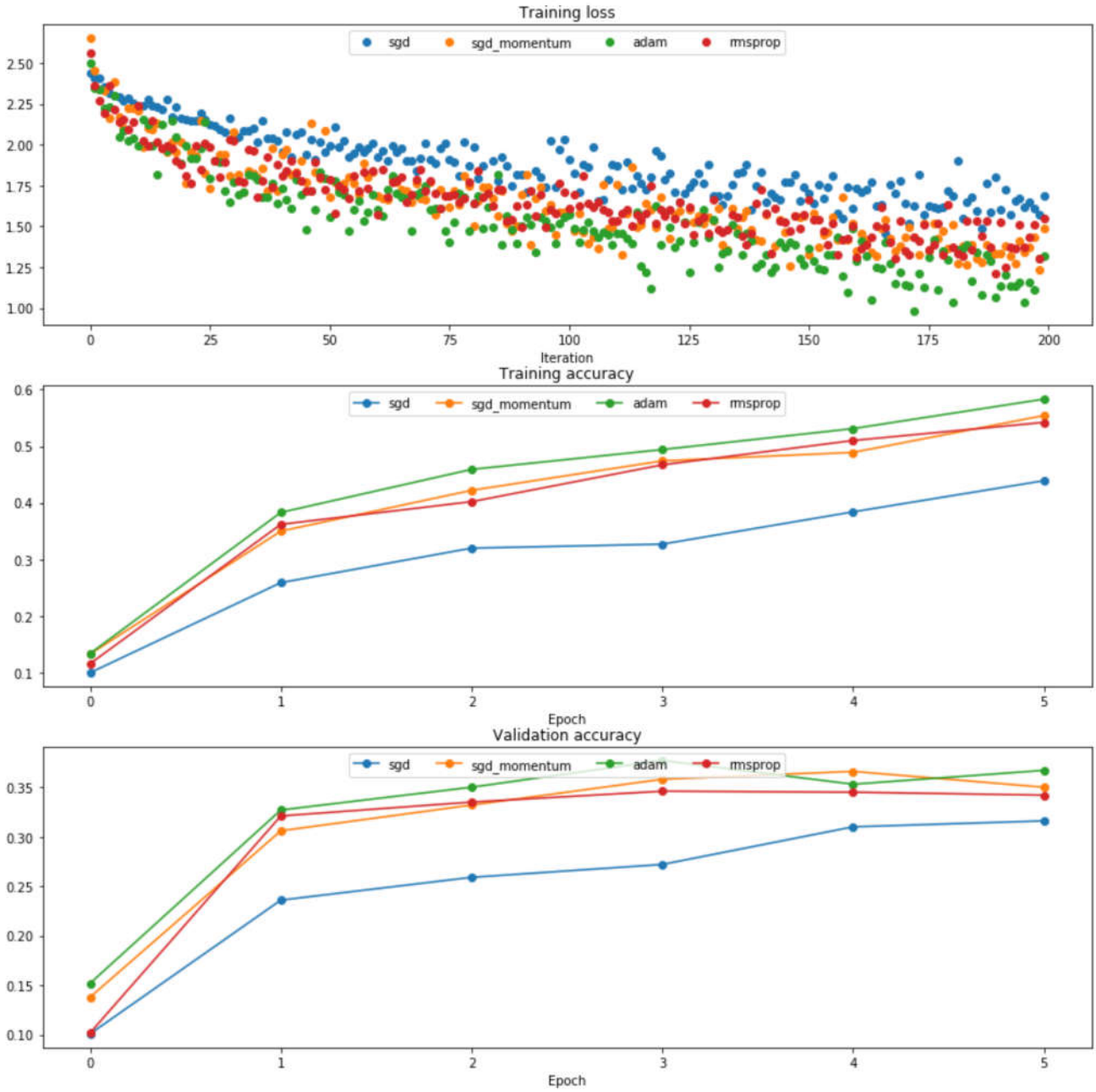
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  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_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  adam
(Iteration 1 / 200) loss: 2.499732
(Epoch 0 / 5) train acc: 0.134000; val_acc: 0.152000
(Iteration 11 / 200) loss: 1.995475
(Iteration 21 / 200) loss: 1.995232
(Iteration 31 / 200) loss: 1.773736
(Epoch 1 / 5) train acc: 0.383000; val_acc: 0.327000
(Iteration 41 / 200) loss: 1.732933
(Iteration 51 / 200) loss: 1.559354
(Iteration 61 / 200) loss: 1.602810
(Iteration 71 / 200) loss: 1.748136
(Epoch 2 / 5) train acc: 0.459000; val_acc: 0.350000
(Iteration 81 / 200) loss: 1.639627
(Iteration 91 / 200) loss: 1.503361
(Iteration 101 / 200) loss: 1.573046
(Iteration 111 / 200) loss: 1.456679
(Epoch 3 / 5) train acc: 0.494000; val_acc: 0.377000
(Iteration 121 / 200) loss: 1.576555
(Iteration 131 / 200) loss: 1.542742
(Iteration 141 / 200) loss: 1.274752
(Iteration 151 / 200) loss: 1.365536
(Epoch 4 / 5) train acc: 0.531000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.284861
(Iteration 171 / 200) loss: 1.141643
(Iteration 181 / 200) loss: 1.032143
(Iteration 191 / 200) loss: 1.138360
(Epoch 5 / 5) train acc: 0.583000; val_acc: 0.367000

running with  rmsprop
(Iteration 1 / 200) loss: 2.559746
(Epoch 0 / 5) train acc: 0.116000; val_acc: 0.102000
(Iteration 11 / 200) loss: 2.238213
(Iteration 21 / 200) loss: 1.810098
(Iteration 31 / 200) loss: 2.022241
(Epoch 1 / 5) train acc: 0.362000; val_acc: 0.321000
(Iteration 41 / 200) loss: 1.808871
(Iteration 51 / 200) loss: 1.786286
(Iteration 61 / 200) loss: 1.573671
(Iteration 71 / 200) loss: 1.719619
(Epoch 2 / 5) train acc: 0.402000; val_acc: 0.335000
(Iteration 81 / 200) loss: 1.641362
(Iteration 91 / 200) loss: 1.496705
(Iteration 101 / 200) loss: 1.609680
(Iteration 111 / 200) loss: 1.531539
(Epoch 3 / 5) train acc: 0.467000; val_acc: 0.346000
(Iteration 121 / 200) loss: 1.562758
(Iteration 131 / 200) loss: 1.667923
(Iteration 141 / 200) loss: 1.725374
(Iteration 151 / 200) loss: 1.662205
(Epoch 4 / 5) train acc: 0.510000; val_acc: 0.345000
(Iteration 161 / 200) loss: 1.313356
(Iteration 171 / 200) loss: 1.351694
(Iteration 181 / 200) loss: 1.374806
```

```
(Iteration 191 / 200) loss: 1.525517
(Epoch 5 / 5) train acc: 0.542000; val_acc: 0.342000
```

```
D:\Python\Anaconda3\lib\site-packages\matplotlib\cbook\deprecation.py:106:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new instance will always be
created and returned.  Meanwhile, this warning can be suppressed, and the future behavior
ensured, by passing a unique label to each axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



Inline Question 3:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

Answer:

Adagrad自适应地为各个参数分配不同学习率的算法。其公式如下： $\Delta x_t = -\frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2 + \epsilon}} g_t$ 其中g_t是当前的梯度，连加和开根号都是元素级别的运算。eta 是初始学习率，由于之后会自动调整学习率，所以初始值就不像之前的算法那样重要了。而ϵ是一个比较小的数，用来保证分母非0。 其含义是，对于每个参数，随着其更新的总距离增多，其学习速率也随之变慢。 Adam不会有这个问题。Adma更新步长与梯度大小无关。

Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################
best_model = FullyConnectedNet([100, 100, 100], weight_scale=1e-2)
solver = Solver(best_model, data, lr_decay=0.95,
                num_epochs=10, batch_size=100, print_every=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                })
solver.train()
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 4900) loss: 2.300708
(Epoch 0 / 10) train acc: 0.133000; val_acc: 0.140000
(Iteration 101 / 4900) loss: 1.813058
(Iteration 201 / 4900) loss: 1.619161
(Iteration 301 / 4900) loss: 1.512859
(Iteration 401 / 4900) loss: 1.535831
(Epoch 1 / 10) train acc: 0.463000; val_acc: 0.442000
(Iteration 501 / 4900) loss: 1.472093
(Iteration 601 / 4900) loss: 1.606569
(Iteration 701 / 4900) loss: 1.547652
(Iteration 801 / 4900) loss: 1.524029
(Iteration 901 / 4900) loss: 1.511513
(Epoch 2 / 10) train acc: 0.501000; val_acc: 0.487000
(Iteration 1001 / 4900) loss: 1.499424
(Iteration 1101 / 4900) loss: 1.304338
(Iteration 1201 / 4900) loss: 1.566519
(Iteration 1301 / 4900) loss: 1.289329
(Iteration 1401 / 4900) loss: 1.400862
(Epoch 3 / 10) train acc: 0.509000; val_acc: 0.474000
(Iteration 1501 / 4900) loss: 1.428173
(Iteration 1601 / 4900) loss: 1.623116
(Iteration 1701 / 4900) loss: 1.465420
(Iteration 1801 / 4900) loss: 1.246377
(Iteration 1901 / 4900) loss: 1.286566
(Epoch 4 / 10) train acc: 0.515000; val_acc: 0.498000
(Iteration 2001 / 4900) loss: 1.240812
(Iteration 2101 / 4900) loss: 1.506252
(Iteration 2201 / 4900) loss: 1.176224
(Iteration 2301 / 4900) loss: 1.380327
(Iteration 2401 / 4900) loss: 1.233783
(Epoch 5 / 10) train acc: 0.561000; val_acc: 0.491000
(Iteration 2501 / 4900) loss: 1.289683
(Iteration 2601 / 4900) loss: 1.251189
(Iteration 2701 / 4900) loss: 1.139643
(Iteration 2801 / 4900) loss: 1.283152
(Iteration 2901 / 4900) loss: 1.125279
(Epoch 6 / 10) train acc: 0.570000; val_acc: 0.505000
(Iteration 3001 / 4900) loss: 1.142210
(Iteration 3101 / 4900) loss: 1.243314
(Iteration 3201 / 4900) loss: 1.027143
(Iteration 3301 / 4900) loss: 1.102593
(Iteration 3401 / 4900) loss: 1.372353
(Epoch 7 / 10) train acc: 0.579000; val_acc: 0.512000
(Iteration 3501 / 4900) loss: 1.240291
(Iteration 3601 / 4900) loss: 1.109637
(Iteration 3701 / 4900) loss: 1.054912
(Iteration 3801 / 4900) loss: 1.292929
(Iteration 3901 / 4900) loss: 1.281895
(Epoch 8 / 10) train acc: 0.602000; val_acc: 0.506000
(Iteration 4001 / 4900) loss: 1.129552
(Iteration 4101 / 4900) loss: 1.002371
(Iteration 4201 / 4900) loss: 1.082399
(Iteration 4301 / 4900) loss: 1.116727
```

```
(Iteration 4401 / 4900) loss: 1.030965
(Epoch 9 / 10) train acc: 0.601000; val_acc: 0.525000
(Iteration 4501 / 4900) loss: 1.120818
(Iteration 4601 / 4900) loss: 0.936371
(Iteration 4701 / 4900) loss: 1.108228
(Iteration 4801 / 4900) loss: 0.987017
(Epoch 10 / 10) train acc: 0.646000; val_acc: 0.519000
```

Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

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

```
Validation set accuracy:  0.525
Test set accuracy:  0.519
```

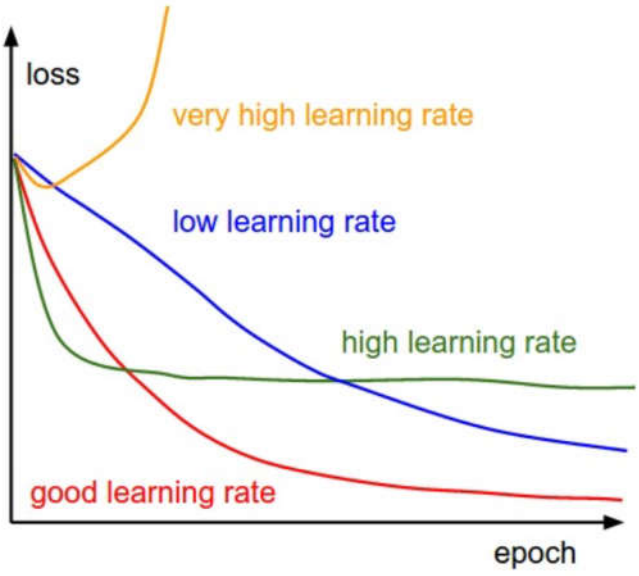
附上实战心得（不算报告，方便自己以后看）：

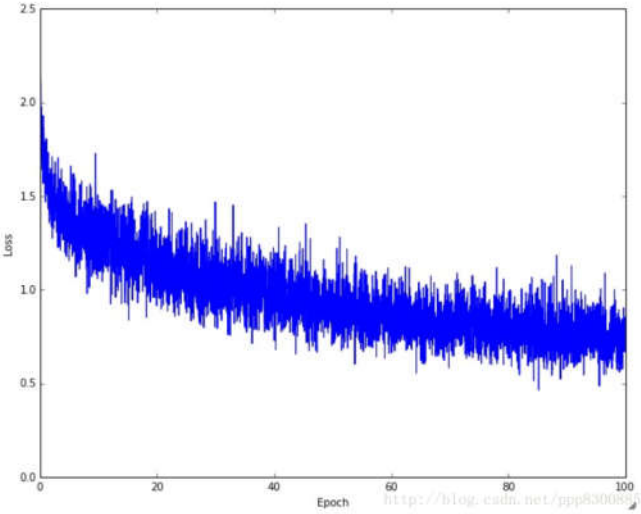
1： 如何测试我们实现的模型有效性：小数据集过拟合： 如何判断我们的方法到能不能work，根据notebook里的提示，可以在一个很小的数据集上跑几个epoch，观察我们的模型是否能够对训练集很好的过拟合，而测试集准确率很低，具体是：在训练集这个100张图片里，是否能够实现99%-100%准确率的判断，而在测试集中另外的100张图片有比较低的准确率（10%左右），这里的模型指我们手动实现的算法

2： 如何判断对层的实现是否正确：梯度检查： 在将我们的算法应用到正式的数据集上之前，需要对实现的层进行解析梯度和数值梯度的比较，具体方法在notebook里，数值梯度是微调1e-6参数获得的差值除以改变量1e-6得到的，而解析梯度是我们实现代码反向传播的输出（因为我们对层的更新都是根据求导法则来的，所以是梯度的解析值），将这两个梯度值比较，观察相对误差，能判断对这个层的实现有没有问题，相对误差在1e-7或者更小是很好的结果，若相对误差达到了1e-2，通常你的实现就有问题。但是，网络越深，相对误差会累计，在10层的网络里若有1e-2的相对误差，那也是可以的。除了梯度检查，notebook中提供了参考值用来检查你实现的权值更新准则如SGD+Momentum、RMSProp、Adam

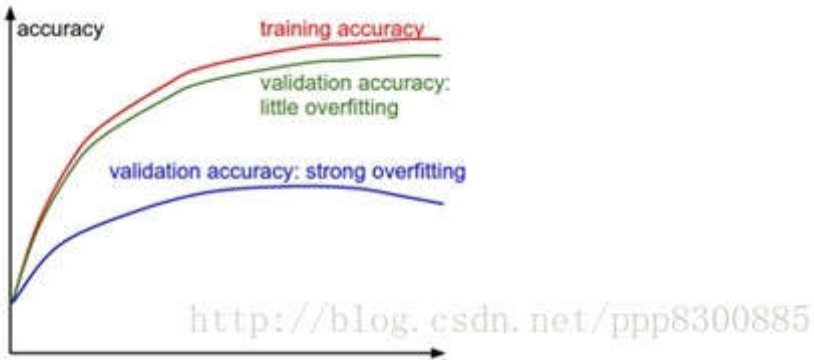
3： 如何判断权值和偏移量参数的初始化是否正确： 初始化对训练过程也是非常重要的，所以我们需要对训练过程进行检查，当正则强度为0时，对于CIFAR-10的softmax输出的分类器（初始权值w为0.01量级的随机数，初始偏移b为0），一般初始的loss function的值为2.302，这是因为初始时分类器对每一类的概率期望为0.1（共有10类），因此softmax损失函数值为对于正确分类概率的负对数：-ln(0.1)=2.302

4： 如何判断学习率是否合适： 在训练过程中，需要对loss值进行实时地打印，可以判断当前训练的状态：高的学习率高会使损失值下降很快，然后停止在一个比较高的位置（相对最优），这是因为参数每次更新过大，导致在最优点附近震荡，但始终无法达到最优点，而过高的学习率会直接使损失值递增。过于低的学习率会导致损失值下降很慢，训练过程太长，引用笔记中的一张图来理解：



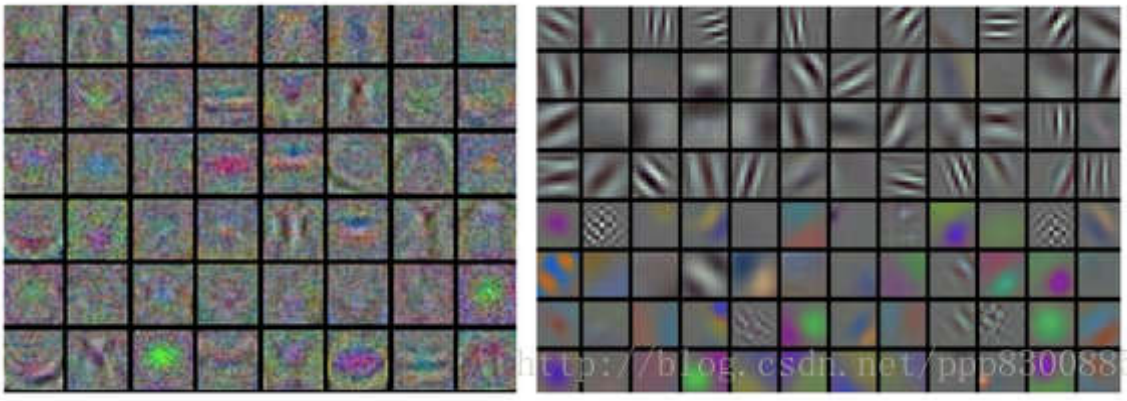


5： 如何判断模型的过拟合程度： 在训练过程中，我们还需要对每个epoch中的训练集和测试集的准确率进行打印，能够确定模型是否过拟合或者欠拟合，若训练集准确率一直大幅度高于验证集，说明此时模型过拟合，对训练集有过好的分类能力导致无法在验证集上进行比较好的分类，解决的方法可以增大正则化强度，如增大L2正则惩罚，增加dropout的随机失活率等。如果训练集一直小幅度低于验证集，说明此时稍微过拟合，而如果训练集和验证集的准确率不相上下，说明此时模型有点欠拟合，没有很好地学习到特征，此时可以调整模型参数如层的深度等，引用笔记中的一张图说明：



6： 如何判断训练中出现的梯度消散问题： 我们知道，当网络的层数过于深以后，会出现梯度消散的情况，也就是回传到前几层的梯度值很小，导致前面几层的参数无法更新。对此，我们可以打印前几层网络权值参数w的更新比例，经验结论是这个更新比例在1e-3比较比较好，若这个值太大，说明学习率太高；若这个值很小到1e-7，说明参数w基本上不会变，发生了梯度消散，解决方法为：1）使用Batch Normalization归一化每层之间的输出，2）激活函数改用线性ReLU，3）还有可能是学习率太低，4）减少网络层数

7： 如何判断训练过程是否稳定和有效： 若数据为图像数据，那么可以把第一层的权重进行可视化，观察模型是否学习到了比较的好的特征，notebook里内置了相关可视化的方法，若特征图中颜色杂乱无规律且充满噪音，说明训练过程未收敛（学习率太高）或者正则化惩罚不够，引用笔记中的图来解释，下图中的右图为比较好的特征，平滑而且种类繁多，说明训练过程有效且稳定



8： 如何进行有效的数据预处理： 在实际应用中CNN比较多的是减均值法和归一化，其他的处理方法为PCA和白化（Whitening）：PCA能消除数据的相关性，使数据的分布在基准值上；白化则可看成是把数据在各个特征方向上进行拉伸压缩变化，使之服从均值为零的高斯分布，具体参考[知乎]（https://zhuanlan.zhihu.com/p/21560667?refer=intelligentunit）