

CS231N - Assignment 1 - Q4 - Two-Layer Neural Network

1. 编写: 郭承坤 观自在降魔 [Fanli SlyneD](#)
2. 校对: 毛丽
3. 总校对与审核: 寒小阳

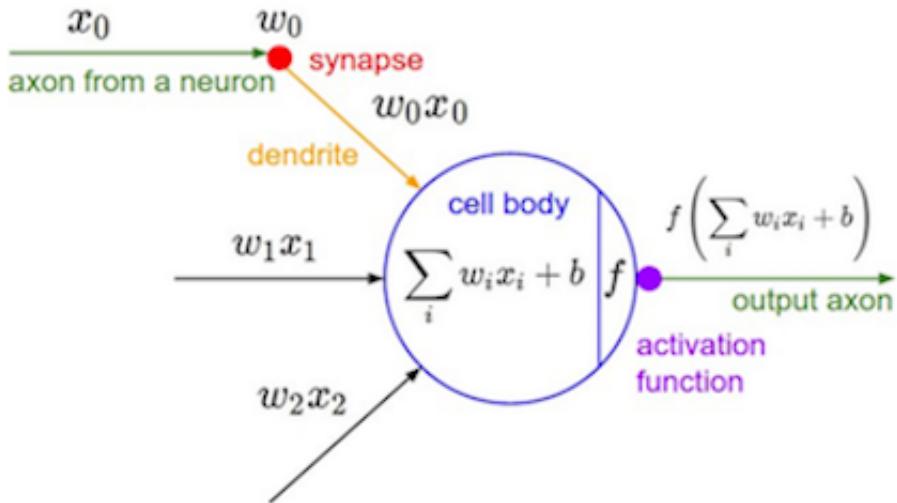
1 任务

在这个练习里, 我们将实现一个完全连接的神经网络分类器, 然后用CIFAR-10数据集进行测试。

2 知识点

2.1 神经元

要讲神经网络就得先从神经元开始说起, 神经元是神经网络的基本单位, 它的结构如下图所示:



神经元不管它的名字有多逼格, 它本质上不过是一个函数。就像每个函数都有输入和输出一样, 神经元的输入是一个向量, 里面包含了很多个 x_i (图左边), 输出的是一个值 (图右边)。神经元在这个过程中只做了两件事: 一是对输入向量做一次仿射变换 (线性变化+平移), 用公式表示就是

$$\sum_i w_i x_i + b$$

二是对仿射变换的结果做一次非线性变换, 即

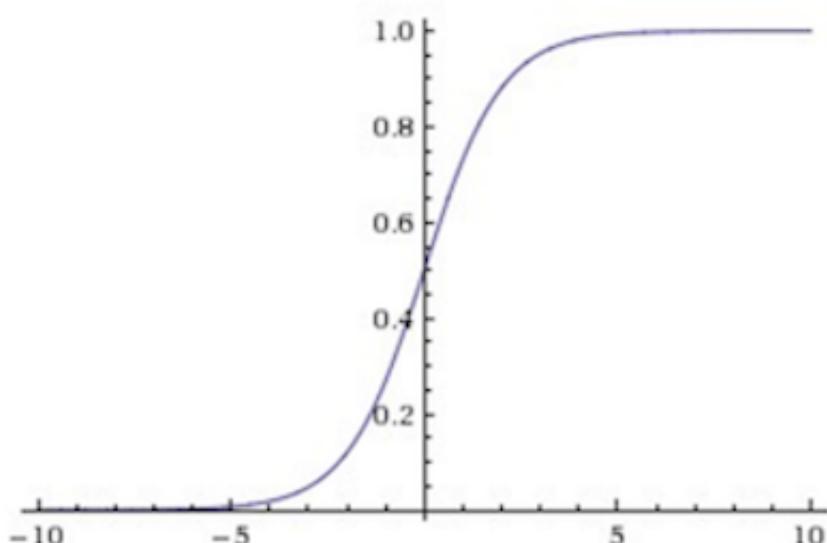
$$f(\sum_i w_i x_i + b)$$

这里的 f 就叫做激活函数。权值 w_i 和偏置项 b 是神经元特有的属性，事实上，不同神经元之间的差别主要就三点：权重值，偏置项，激活函数。总的来说，神经元的工作就是把所有的输入项 x_i 与其对应的权值相乘求和，然后加上偏置项 b ，之后通过一个激活函数求得最终的值，即这个神经元的输出。

激活函数有很多种，下面介绍一个叫sigmoid的激活函数：

$$\sigma(x) = 1/(1 + e^{-x})$$

这个函数很有意思，它把任何一个属于值域R的实数转换成0到1之间的数，这使得概率的诠释变得可能。但是在训练神经网络的时候，它有两个问题，一是当输入的仿射变换值太大或者太小的时候，那 f 关于 x 的梯度值就为0，这不利于后面提到的反向传播算法中梯度的计算。第二个问题在于它的值不是中心对称的，这也导致梯度更新过程中的不稳定。Sigmoid的函数图像大概长下面这样：



如果现在我们把实数1也放进输入的向量中得到 \vec{x} ，把偏置项放进权值向量中得到 \vec{w} ，那么包含sigmoid激活函数神经元的工作就可以简洁地表示为：

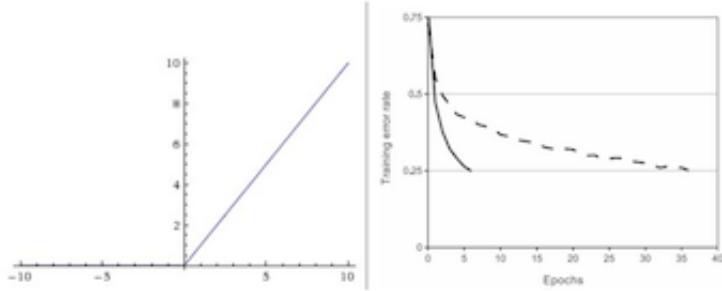
$$y = \text{sigmoid}(\vec{w}^T \cdot \vec{x})$$

聪明的你或许已经看出来：既然sigmoid函数的输出可以诠释为在0到1间的概率值大小，那么，单个神经元其实就可以当作一个二类线性分类器来使用。也就是说，假设我们输入的是一个图像向量，将它乘以权重向量之后再通过sigmoid函数，如果得到的值如果大于0.5，那这幅图片就属于这个类别，否则就不是。

实践中比较常用的是这个叫ReLU的激活函数：

$$f(x) = \max(0, x)$$

这个函数把输入的 x 与0做比较，取两者中更大的那个作为函数的输出值。它的图像表示如下：

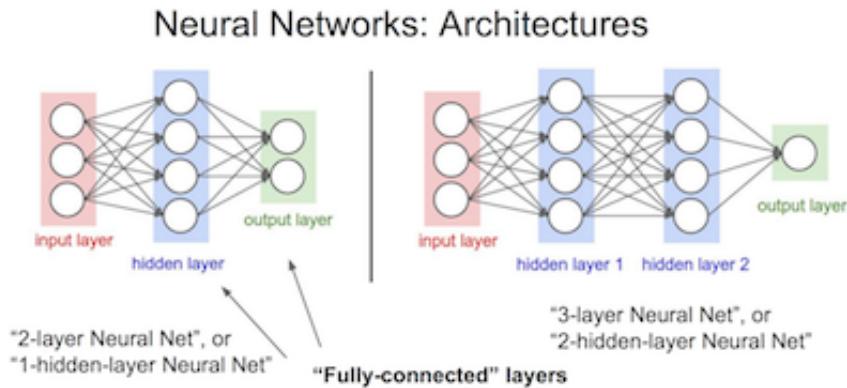


Left: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$. Right: A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

ReLU在实践中被证明的优点是它在随机梯度下降法中收敛效果非常好，此外它的梯度值非常简单。但它的缺点也非常明显，就是当它的梯度为0的时候，这个神经元就相当于把从后面传过来的梯度值都变成了0，使得某些变量在梯度下降法法中不再变化。cs231n官网笔记里有关于不同激活函数的详细介绍，大家有兴趣可以看一下：<http://cs231n.github.io/neural-networks-1/>。

2.2 神经网络

当我们把很多个神经元按照一定规则排列和连接起来，便得到了一个神经网络。如果网络中每层之间的神经元是两两连接的，这样的神经网络就叫做全连接神经网络（Fully-connected Neural Network）。下图展示了两个不同层数的全连接神经网络：



Fei-Fei Li & Andrej Karpathy & Justin Johnson Lecture 4 - 77 13 Jan 2016

图中的神经网络包含一个输入层（Input Layer）和输出层（Output Layer），中间的层叫隐藏层（Hidden Layer）。需要注意的是，输入层包含的并不是神经元，而是输入向量 \vec{x} ，上图输入层中圆圈的表示方法其实有些误导。输出层在最后起的作用相当于一个多类分类器，实践中经常采用Softmax分类器。而中间的隐藏层，无论有多少层，隐藏层们在一起做的事情相当于输入向量特征的提取。

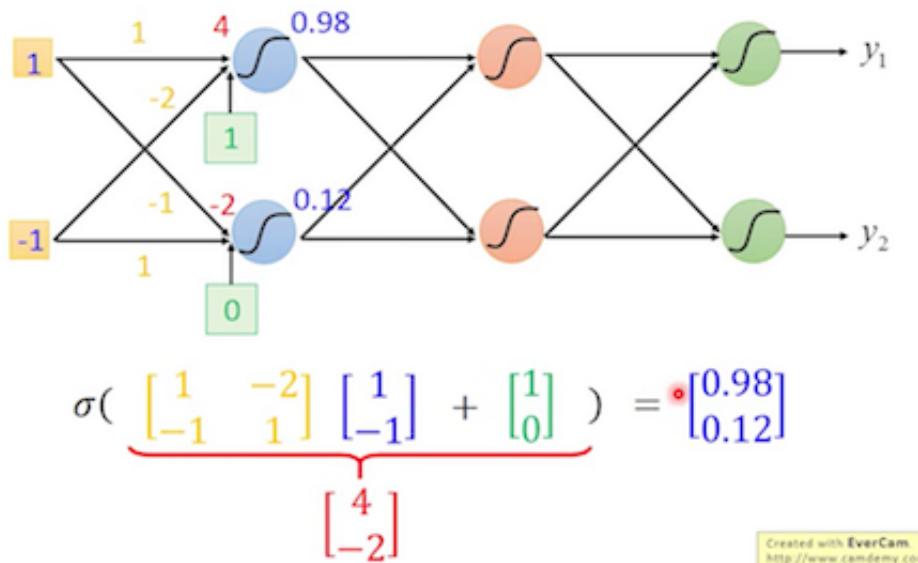
现在我们从计算的角度来分析下一层神经网络到底在做什么。我们知道一个神经元做的事情是把一个向量变成一个标量，那当我们把很多个神经元排列起来变成一个神经网络层的时候，这个层输出的结果应该是跟层里神经元数目相当的很多个标量，把这些标量排列起来就

成了向量。也就是说，一个神经网络层做的事情是把一个向量变成了另一个向量，这中间的计算过程无非就是矩阵运算再加一个激活函数。假设输入向量为 \vec{x} ，矩阵和向量的乘法运算可以表示为 $\mathbf{W}\vec{x}$ ，这里的矩阵 \mathbf{W} 是将层中的每一个神经元的转置权重向量 \vec{w}_i^T 由上到下排列得到，就像这样：

$$\mathbf{W} = \begin{bmatrix} \vec{w}_1^T \\ \vec{w}_2^T \\ \vec{w}_3^T \\ \vec{w}_4^T \end{bmatrix}$$

现在我们拿一个简单的例子来感受一下一个神经网络层在做的事情（图片来源于台湾李宏毅教授深度学习的课件）：

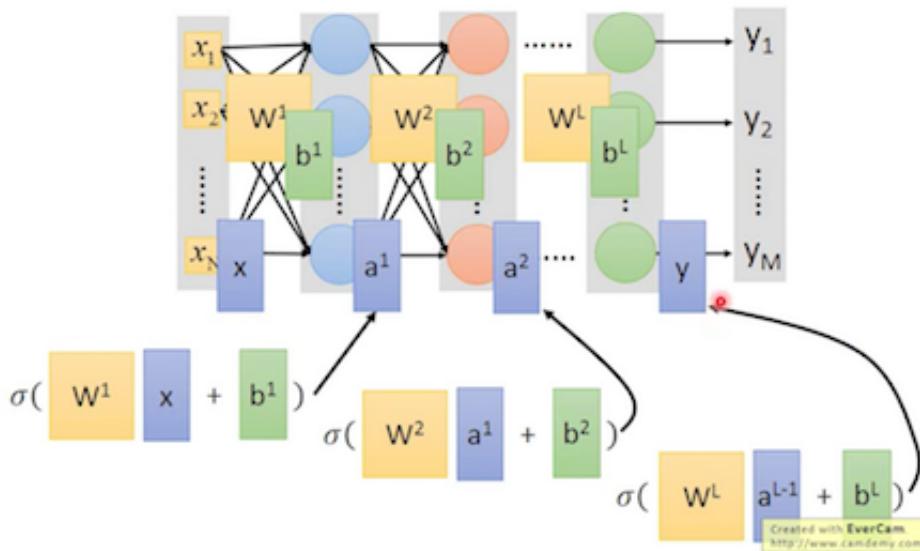
Matrix Operation



图中的网络一共有四层，两个隐藏层，一个输入层和一个输出层，每个隐藏层中又包含两个神经元。假设输入为向量 $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$ ，将隐藏层中每个神经元的权重都作为一行，那么第一个隐藏层的权重矩阵就是一个大小为 2×2 的矩阵（如图中黄色字体显示），权重矩阵与输入向量相乘再加上偏置向量得到的向量 $\begin{bmatrix} 4 \\ -2 \end{bmatrix}$ 是每个神经元对输入向量做仿射变换的结果，之后通过激活函数 σ 得到第一个隐藏层的输出 $\begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$ 。输出向量里的 0.98 代表输入向量经过第一个神经元变换后得到的值，0.12 输入向量通过第二个神经元后得到的值。之后隐藏层的运算可以以此类推。

所以，把整个神经网络看出是一个函数，这个函数本质上就是一连串的矩阵运算再加上激活函数的非线性变换，下图很好地给出了多层神经网络的计算过程：

Neural Network



图中的 w^i 是第*i*个隐藏层的权重矩阵， b^i 该层的权重向量， a^i 是该层的输出向量。输入向量 x 经过第一个隐藏层的变换得到 a^1 ，经过第二个隐藏层得到 a^2 ，直到最后一层得到输出向量 y 。

值得一提的是，神经网络的层数以及每一层神经元个数的设计是一门艺术，更多的还是依靠不断地尝试和经验。目前人们发现，对于全连接网络，三层神经网络训练效果往往比较好。但是在使用卷积神经网络解决图像相关的问题时，层数越多，能对图片这种复杂的数据表示得更充分，往往训练效果会越好。

2.3 反向传播算法

训练神经网络的步骤跟我们之前训练SVM以及Softmax分类器一样：首先我们有很多个训练样本以及每个样本对应的分类标签，在确定一个模型之后（你可以使用SVM，Softmax或者神经网络），我们可以得到用模型参数表示的损失函数，之后的目标是使损失函数值最小的一组模型参数，参数的训练用的一般是梯度下降法。下图很好地总结了梯度下降法的步骤（同样来自于台湾李宏毅教授深度学习的课件）：

Gradient Descent

Network parameters $\theta = \{w_1, w_2, \dots, b_1, b_2, \dots\}$

Starting Parameters $\theta^0 \longrightarrow \theta^1 \longrightarrow \theta^2 \longrightarrow \dots$

$$\nabla L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial w_0} \\ \frac{\partial L(\theta)}{\partial w_1} \\ \vdots \\ \frac{\partial L(\theta)}{\partial b_1} \\ \frac{\partial L(\theta)}{\partial b_2} \\ \vdots \end{bmatrix} \quad \begin{array}{ll} \text{Compute } \nabla L(\theta^0) & \theta^1 = \theta^0 - \eta \nabla L(\theta^0) \\ \text{Compute } \nabla L(\theta^1) & \theta^2 = \theta^1 - \eta \nabla L(\theta^1) \\ \text{Millions of parameters} \dots & \end{array}$$

To compute the gradients efficiently, we use backpropagation.

Created with EverCam

第一步是随便确定一组参数值，神经网络里的参数是每个神经元的权重和偏置值，用向量 θ^0 表示；第二步，计算损失函数在每一个参数上的偏导数，用梯度 $\nabla L(\theta^0)$ 来表示；第三步，用学习率和刚刚计算出来的梯度值更新第一组参数。之后重复这个步骤，直到参数值收敛。但神经网络中的参数量往往非常庞大，用这样的方法一个一个地去计算它们的梯度值效率很低，所以在这里我们的反向传播算法就登场了。

反向传播算法是一个非常高效获得函数梯度的算法。我们首先来看一个简单的例子：

Backpropagation: a simple example

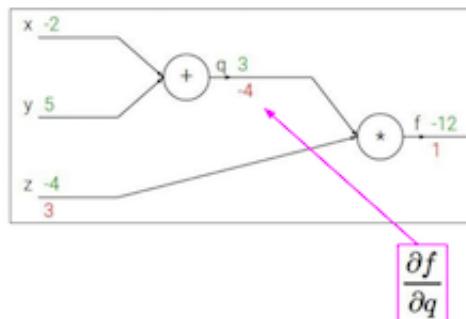
$$f(x, y, z) = (x + y)z$$

$$\text{e.g. } x = -2, y = 5, z = -4$$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

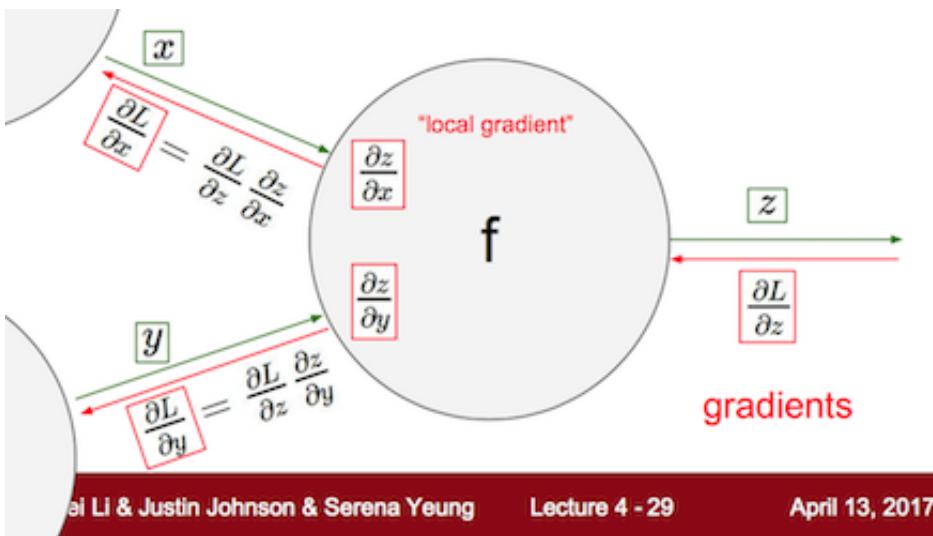
$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

$$\text{Want: } \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$



$$\frac{\partial f}{\partial q}$$

上图右上角的计算图表示函数 $f = (x + y) * z$ 的计算过程，每一个节点表示一个运算，节点左边线上的数字为一个节点的输入值，节点右边线上的数字为通过节点运算之后的输出值。梯度的计算分为两个步骤：先是用正向传播从左向右计算出每个节点的输出值，接着用反向传播从右到左计算出函数输出值关于每个变量的导数。从图中可以看到，当给出 $x = -2$, $y = 5$, $z = -4$ 的时候，计算图输出的结果为 -12。而图中线下的值表示的是导数，最右边的 1 是 f 对自己的导数，所以是 1。而之后的 -4 是 f 对 q 的导数，也就是 z 值 -4。想要求 f 对 x 的导数，需要用到链式法则。具体的公式在下图中可以看到：



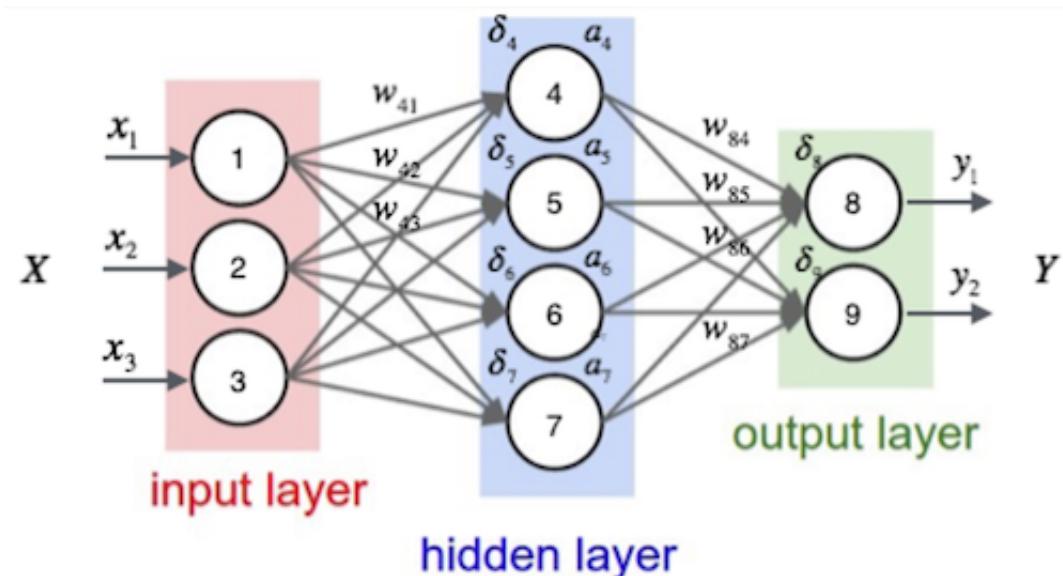
所以例子中 f 关于 x 的导数就是：

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

因为后者为1， 所以 $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} = -4$ 。

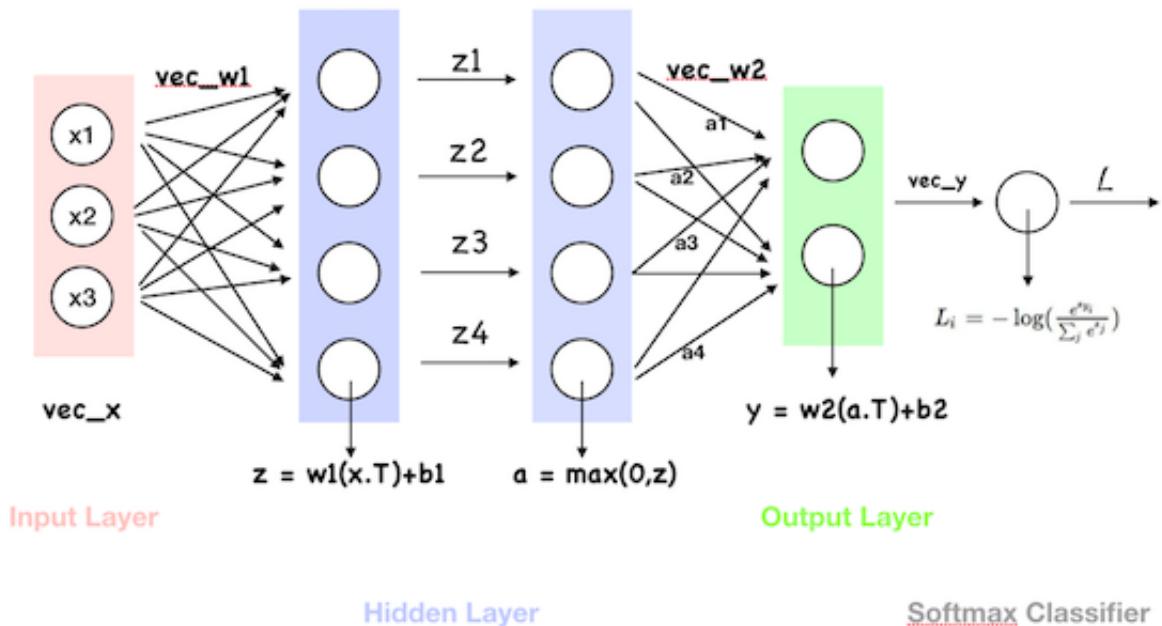
这种计算梯度的方法其实是非常优雅的，因为你发现计算图中的每一个节点之间都是相互独立的，只要给一个节点输入，它就会独立地计算出输出值以及输出值关于输入值的梯度（local gradient），所以当你把很多个相互独立的节点组织成一张计算图，然后给它一个起始值，计算图就会在自动得出梯度值。

理解了反向传播算法，我们回到神经网络的训练上。正如上面 $f = (x + y) * z$ 函数的计算图，我们的神经网络也是一个超大型的计算图，比如给出下图中的两层神经网络



我们可以画出该网络的计算图，如下图所示。需要注意的是，计算图里的原点并不表示一个神经元，它只是一种运算节点。为了保持跟神经网络的一致性，我将输入层也画了进去，但输入层里并没有任何运算，它的作用只是为下一层神经网络提供数据而已。因为一个神经网

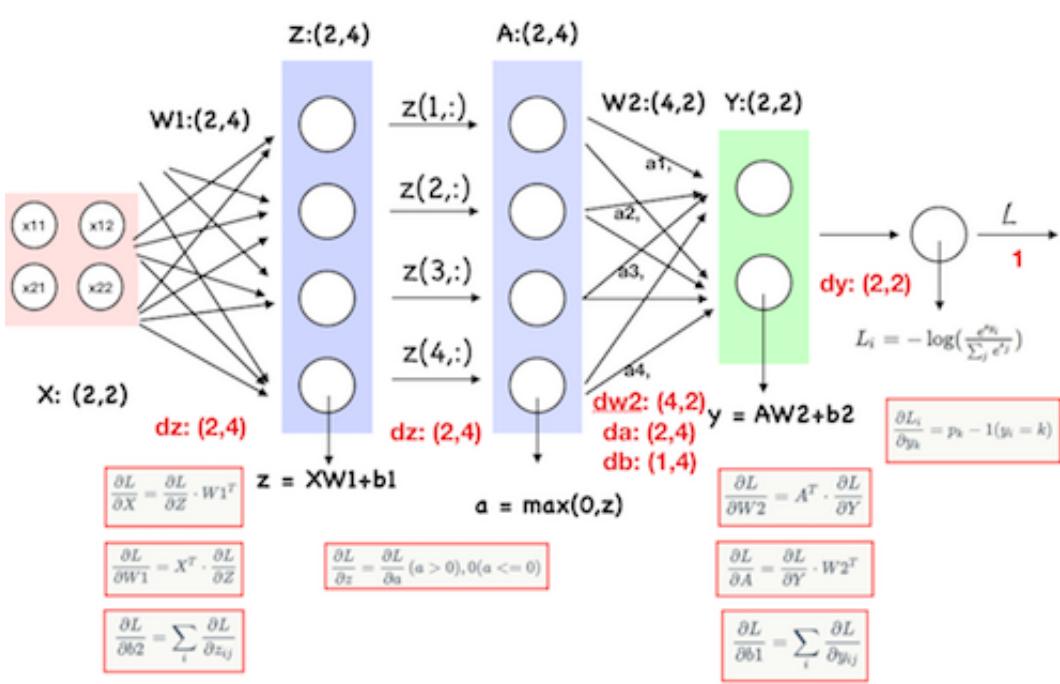
络层有很多个神经元，所以一层里会有很多个计算节点。图中下面的公式表示该层节点进行的运算。输出层之后的节点是softmax，这里需要把所有节点的输出值放进一个向量里作为softmax节点的输入，最终得到网络的损失值。



有了计算图，我们现在就可以开始求梯度了。从右到左，损失函数是一个标量，所以，标量关于自身的导数是1；之后的softmax函数的梯度是：

$$\frac{\partial L_i}{\partial f_k} = p_k - 1(y_i = k)$$

也就是说，在将输出向量归一化之后得到的概率值，除了正确类别对应的概率值要减去1，其他的都保持不变，得到的向量就会是损失值关于输出向量的梯度。之后根据链式法则，求得 y 关于 w_2 , b_2 以及 a 的导数，再将它们分别乘以softmax的梯度就得到损失值关于 w_2 , b_2 以及 a 的梯度了。这个做法跟我们上面在函数计算图中的求梯度方法是一样的，就是一步一步求回去。在具体训练的时候，我们往往会有许多个图像样本，它们是以矩阵的形式排列的，反向传播算法同样也适用这种情况，下图提供了一个总结：



3 代码解析

3.1 初始化代码

```

1. # 导入模块和库
2. import numpy as np
3. import matplotlib.pyplot as plt
4.
5. from cs231n.classifiers.neural_net import TwoLayerNet
6.
7. from __future__ import print_function
8.
9. %matplotlib inline
10. plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
11. plt.rcParams['image.interpolation'] = 'nearest'
12. plt.rcParams['image.cmap'] = 'gray'
13.
14.
15. # 自动加载外部的模块
16. # see http://stackoverflow.com/questions/1907993/autoreload-of-modules
17. %load_ext autoreload
18. %autoreload 2
19.
20.
21. def rel_error(x, y):
22.     """ returns relative error """
23.     return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))

```

3.2 加载测试数据

我们将使用文件cs231n / classifiers / neural_net.py中的类TwoLayerNet来表示神经网络中的实例。网络参数存储在实例变量self.params中，其中关键字是字符串参数名称，值是numpy数组。下面，我们将初始化一个的测试数据和测试模型用来帮助神经网络的实现。

```
1. # 创建一个小的网络和测试数据来检查你的实现。注意我们用了随机种子来帮助实现实验的可复现性。
2.
3.
4. input_size = 4
5. hidden_size = 10
6. num_classes = 3
7. num_inputs = 5
8.
9. def init_toy_model():
10.     np.random.seed(0)
11.     return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)
12.
13. def init_toy_data():
14.     np.random.seed(1)
15.     X = 10 * np.random.randn(num_inputs, input_size)
16.     y = np.array([0, 1, 2, 2, 1])
17.     return X, y
18.
19. net = init_toy_model()
20. X, y = init_toy_data()
```

3.3 前向传播：计算分值

打开文件cs231n/classifiers/neural_net.py，找到你们的TwoLayerNet.loss这个方法。它的函数跟你在SVM和Softmax练习中写的损失函数非常相似：它需要数据和权重来计算分值和损失值以及参数的梯度。实现第一部分的向前传播算法。使用权重和偏置项来计算所有输入数据的分值。

```
1. #插播neural.net.py文件中的第一部分代码
2. z1 = X.dot(W1) + b1
3. a1 = np.maximum(0, z1)
4. scores = a1.dot(W2) + b2
5. #插播结束
```

```

1. scores = net.loss(X)
2. print('Your scores:')
3. print(scores)
4. print()
5. print('correct scores:')
6. correct_scores = np.asarray([
7.     [-0.81233741, -1.27654624, -0.70335995],
8.     [-0.17129677, -1.18803311, -0.47310444],
9.     [-0.51590475, -1.01354314, -0.8504215 ],
10.    [-0.15419291, -0.48629638, -0.52901952],
11.    [-0.00618733, -0.12435261, -0.15226949]])
12. print(correct_scores)
13. print()
14.
15.
16. # 差值应该非常小, 小于1e-7
17. print('Difference between your scores and correct scores:')
18. print(np.sum(np.abs(scores - correct_scores)))

```

输出结果:

```

Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]
correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]
Difference between your scores and correct scores:3.68027207459e-08

```

3. 前向传播：计算损失值

在同一个函数中，实现第二部分中的数据和正则化损失值

```

1. #插播neural.net.py文件中的第二部分代码
2. exp_scores = np.exp(scores)
3. probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
4. #插播结束

```

```
1. loss, _ = net.loss(X, y, reg=0.05)
2. correct_loss = 1.30378789133
3.
4.
5. # 应该非常小, 小于1e-12
6. print('Difference between your loss and correct loss:')
7. print(np.sum(np.abs(loss - correct_loss)))
```

输出结果：

```
Difference between your loss and correct loss:
```

```
0.0189654196061
```

3.4 反向传播

实现函数中剩下的部分，计算关于变量W1, b1, W2和b2的损失梯度。如果你已经正确实现了前向传播，那么你现在就可以通过数值梯度检查的方法来调试你的反向传播。

```
1. # 插播neural.net.py文件中的第三部分代码
2. # 计算分值的梯度
3. dscores = probs
4. dscores[range(N),y] -= 1
5. dscores /= N
6.
7. # W2 和 b2
8. grads['W2'] = np.dot(a1.T, dscores)
9. grads['b2'] = np.sum(dscores, axis=0)
10.
11. # 反向传播里第二个隐藏层
12. dhidden = np.dot(dscores, W2.T)
13.
14. # 激活函数ReLU的梯度
15. dhidden[a1 <= 0] = 0
16.
17. # 关于W1和b1的梯度
18. grads['W1'] = np.dot(X.T, dhidden)
19. grads['b1'] = np.sum(dhidden, axis=0)
20.
21. # 加上正则化梯度的部分
22. grads['W2'] += reg * W2
23. grads['W1'] += reg * W1
24. #插播结束
```

```

1. from cs231n.gradient_check import eval_numerical_gradient
2.
3.
4. # 使用数值梯度法检查反向传播的实现。如果你的实现正确的话，每一个W1, W2, b1和b2的分
5. loss, grads = net.loss(X, y, reg=0.05)
6.
7.
8. # 这些都应该小于1e-8
9. for param_name in grads:
10.     f = lambda W: net.loss(X, y, reg=0.05)[0]
11.     param_grad_num = eval_numerical_gradient(f, net.params[param_name])
12.     print('%s max relative error: %e' % (param_name, rel_error(param_g

```

输出结果：

```

W1 max relative error: 4.090896e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447646e-11
b1 max relative error: 2.738421e-09

```

3.5 训练神经网络

跟SVM和Softmax分类器类似，我们将使用随机梯度下降法训练我们的神经网络。查看TwoLayerNet里的train函数，在相应的地方补充训练过程的代码。这个跟你在训练SVM和Softmax分类器的时候应该很相似。然后你还需要完成predict这个函数，这样在训练过程中，可以记录下预测正确率。

完成了这个代码之后，用下面的代码在你的测试数据上训练两层神经网络。你应该会得到一个小于0.2的损失值。

```

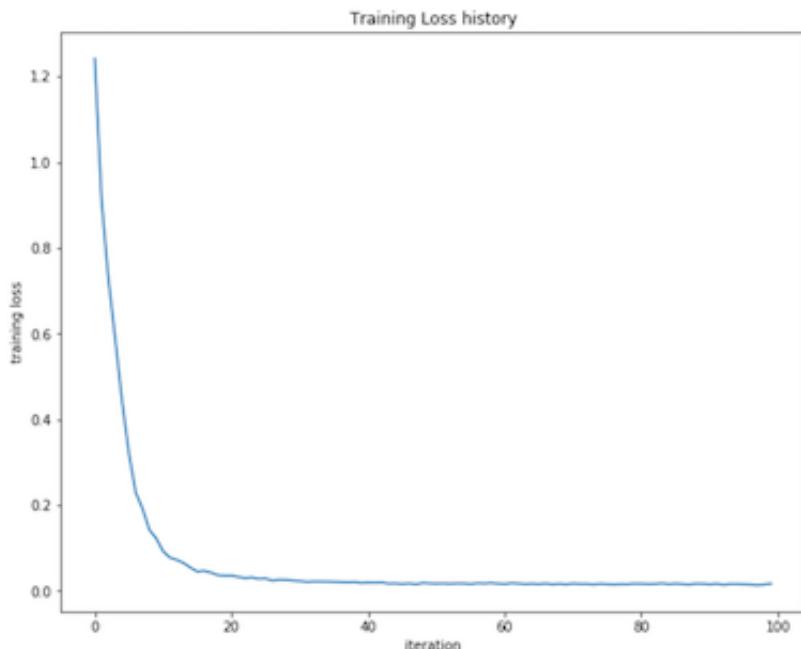
1. #下面插播neural.net.py文件里train函数的缺失代码
2. sample_indices = np.random.choice(np.arange(num_train), batch_size)
3. X_batch = X[sample_indices]
4. y_batch = y[sample_indices]
5. #####
6. self.params['W1'] += -learning_rate * grads['W1']
7. self.params['b1'] += -learning_rate * grads['b1']
8. self.params['W2'] += -learning_rate * grads['W2']
9. self.params['b2'] += -learning_rate * grads['b2']
10.
11. #下面是neural.net.py文件里predic函数的代码
12. z1 = X.dot(self.params['W1']) + self.params['b1']
13. a1 = np.maximum(0, z1) # pass through ReLU activation function
14. scores = a1.dot(self.params['W2']) + self.params['b2']
15. y_pred = np.argmax(scores, axis=1)
16. #插播结束

```

```
1. net = init_toy_model()
2. stats = net.train(X, y, X, y,
3. learning_rate=1e-1, reg=5e-6,
4. num_iters=100, verbose=False)
5.
6. print('Final training loss: ', stats['loss_history'][-1])
7.
8.
9. # 画出迭代过程的损失值变化图像
10. plt.plot(stats['loss_history'])
11. plt.xlabel('iteration')
12. plt.ylabel('training loss')
13. plt.title('Training Loss history')
14. plt.show()
```

输出结果：

```
Final training loss:  0.0171436435329
```



3.6 加载CIFAR-10图像数据

现在你已经成功实现了一个通过梯度检查并在测试数据集上工作的两层神经网络，现在是时候在我们最受欢迎的CIFAR-10数据集上训练模型了。

```
1. from cs231n.data_utils import load_CIFAR10
2.
3. def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test
4.         =100):
5.     """Load the CIFAR-10 dataset from disk and perform preprocessing to p
6.     it for the two-layer neural net classifier. These are the same ste
7.     we used for the SVM, but condensed to a single function.
8.     """
9.
10.    # 加载CIFAR-10数据
11.    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
12.    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
13.
14.
15.    # 从数据集中取数据子集用于后面的练习
16.    mask = list(range(num_training, num_training + num_validation))
17.    X_val = X_train[mask]
18.    y_val = y_train[mask]
19.    mask = list(range(num_training))
20.    X_train = X_train[mask]
21.    y_train = y_train[mask]
22.    mask = list(range(num_test))
23.    X_test = X_test[mask]
24.    y_test = y_test[mask]
25.
26.
27.    # 标准化数据：先求平均图像，再将每个图像都减去其平均图像
28.    mean_image = np.mean(X_train, axis=0)
29.    X_train -= mean_image
30.    X_val -= mean_image
31.    X_test -= mean_image
32.
33.
34.    # 将所有的图像数据都变成行的形式
35.    X_train = X_train.reshape(num_training, -1)
36.    X_val = X_val.reshape(num_validation, -1)
37.    X_test = X_test.reshape(num_test, -1)
38.
39.    return X_train, y_train, X_val, y_val, X_test, y_test
40.
41.
42. # 调用该函数以获取我们需要的数据，查看数据集的大小
43. X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
44. print('Train data shape: ', X_train.shape)
45. print('Train labels shape: ', y_train.shape)
46. print('Validation data shape: ', X_val.shape)
47. print('Validation labels shape: ', y_val.shape)
48. print('Test data shape: ', X_test.shape)
49. print('Test labels shape: ', y_test.shape)
```

输出结果：

```
Train data shape: (49000, 3072)
Train labels shape: (49000,)
Validation data shape: (1000, 3072)
Validation labels shape: (1000,)
Test data shape: (1000, 3072)
Test labels shape: (1000,)
```

3.7 训练神经网络

我们将使用包含momentum的随机梯度下降法来训练我们的神经网络。此外，每一次迭代之后，我们都学习率乘以递减率，所以我们的学习率会在优化的进程中成指数化递减。

```
1. input_size = 32 * 32 * 3
2. hidden_size = 50
3. num_classes = 10
4. net = TwoLayerNet(input_size, hidden_size, num_classes)
5.
6.
7. # 训练网络
8. stats = net.train(X_train, y_train, X_val, y_val,
9.                     num_iters=1000, batch_size=200,
10.                    learning_rate=1e-4, learning_rate_decay=0.95,
11.                   reg=0.25, verbose=True)
12.
13.
14. # 在验证集上进行预测
15. val_acc = (net.predict(X_val) == y_val).mean()
16. print('Validation accuracy: ', val_acc)
```

输出结果：

```
iteration 0 / 1000: loss 2.302762
iteration 100 / 1000: loss 2.302358
iteration 200 / 1000: loss 2.297404
iteration 300 / 1000: loss 2.258897
iteration 400 / 1000: loss 2.202975
iteration 500 / 1000: loss 2.116816
iteration 600 / 1000: loss 2.049789
iteration 700 / 1000: loss 1.985711
iteration 800 / 1000: loss 2.003726
iteration 900 / 1000: loss 1.948076
Validation accuracy: 0.287
```

3.8 调试训练结果

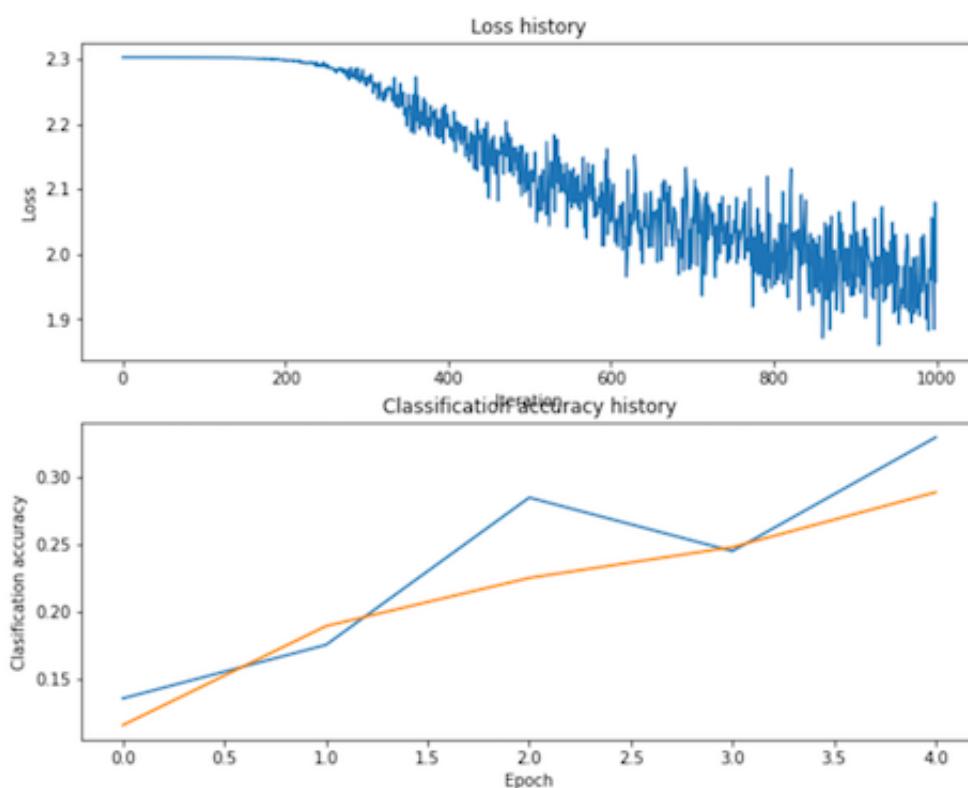
用我们提供的默认参数，你将在验证集上获得0.29的验证准确率。这还不够好。

了解错误的一个策略是绘制损失函数值以及在优化过程中训练集和验证集之间的准确性。

另一个策略是可视化在第一层神经网络中学到的权重。在大多数用视觉数据训练得到的神经网络中，第一层网络的权重通常会在可视化时显示出一些可见结构。

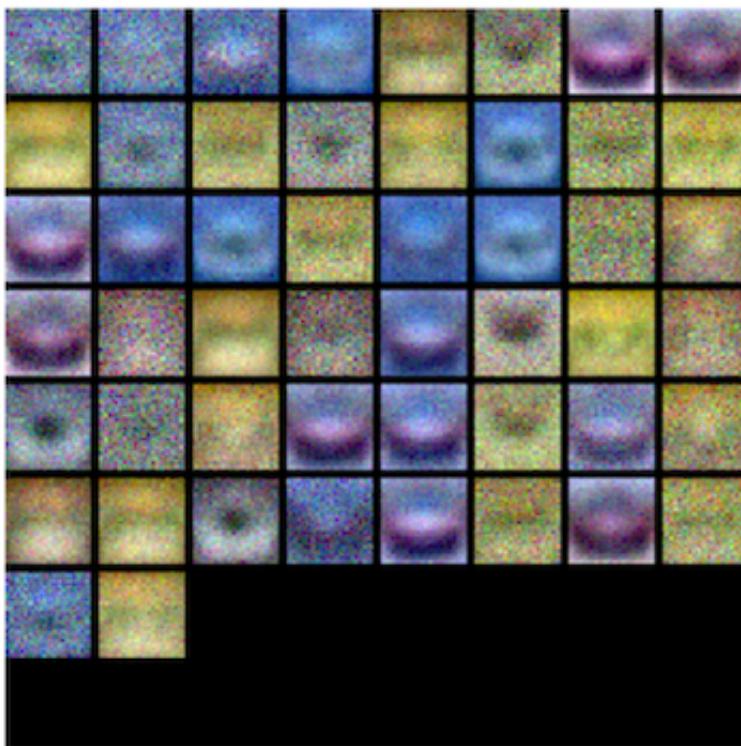
```
1. # 绘制损失值
2. plt.subplot(2, 1, 1)
3. plt.plot(stats['loss_history'])
4. plt.title('Loss history')
5. plt.xlabel('Iteration')
6. plt.ylabel('Loss')
7.
8.
9. plt.subplot(2, 1, 2)
10. plt.plot(stats['train_acc_history'], label='train')
11. plt.plot(stats['val_acc_history'], label='val')
12. plt.title('Classification accuracy history')
13. plt.xlabel('Epoch')
14. plt.ylabel('Classification accuracy')
15. plt.show()
```

输出结果：



```
1. from cs231n.vis_utils import visualize_grid
2.
3.
4. # 可视化网络的权重
5. def show_net_weights(net):
6.     W1 = net.params['W1']
7.     W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
8.     plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
9.     plt.gca().axis('off')
10.    plt.show()
11. show_net_weights(net)
```

输出结果：



3.9 调试你的超参数

哪里出错了？仔细看上面可视化的结果，我们看到损失值几乎是线性减少，这意味着我们的学习率可能设定得太低了。

此外，训练和验证准确率之间没有差距，表明我们使用的模型容量低，我们需要增加它的容量。另一方面，模型如果非常大的话，我们也将会看到更多的过拟合，这将在训练和验证准确率之间表现出很大的差距。

调参。调整超参数并且培养超参数是如何影响最终结果的直觉是训练神经网络的重要组成部分，因此我们希望你多尝试多实践。接下来，你将试验超参数的不同的值，包括隐藏层大小，学习率，训练次数和正则化强度。你也可以考虑调整学习率衰减参数，但你用默认值应该也能获得非常好的性能。

大概的结果。你的目标应该是在验证集上取得超过48%的分类准确度。我们最好的网络在验证集上的准确率超过了52%。

实验：你在本练习中的目标是使用全连接神经网络，尽可能地在CIFAR-10数据集中获得最好的分类结果。对于任何在测试集上获得超过52%结果的，每超出1%就会得到一分额外的奖励。大胆去实践你的技术吧（例如用PCA以降低维度，或添加dropout或向分类器添加其它feature等）。

```
1. best_net = None # store the best model into this
2. #将最好的模型放进这里
3.
4.
5. # 使用验证集调整超参数。在best_net变量中存储最好模型的模型参数。为了帮助调试你的网
6. 手动调整超参数可能很有趣，但你可能会发现编写代码自动扫描可能的超参数组合会更有用，就
7. best_val = -1
8. best_stats = None
9. learning_rates = [1e-2, 1e-3]
10. regularization_strengths = [0.4, 0.5, 0.6]
11. results = {}
12. iters = 2000 #100
13. for lr in learning_rates:
14.     for rs in regularization_strengths:
15.         net = TwoLayerNet(input_size, hidden_size, num_classes)
16.
17.         # Train the network
18.         stats = net.train(X_train, y_train, X_val, y_val,
19.                           num_iters=iters, batch_size=200,
20.                           learning_rate=lr, learning_rate_decay=0.95,
21.                           reg=rs)
22.
23.         y_train_pred = net.predict(X_train)
24.         acc_train = np.mean(y_train == y_train_pred)
25.         y_val_pred = net.predict(X_val)
26.         acc_val = np.mean(y_val == y_val_pred)
27.
28.         results[(lr, rs)] = (acc_train, acc_val)
29.
30.         if best_val < acc_val:
31.             best_stats = stats
32.             best_val = acc_val
33.             best_net = net
34.
35. # Print out results.
36. for lr, reg in sorted(results):
37.     train_accuracy, val_accuracy = results[(lr, reg)]
38.     print ("lr ", lr, "reg ", reg, "train accuracy: ", train_accuracy,
39.
40. print ("best validation accuracy achieved during cross-validation: ",
41. #####
42. #                                     END OF YOUR CODE
43. #####
```

输出结果：

```
lr 0.001 reg 0.4 train accuracy: 0.516408163265 val accuracy: 0.48
lr 0.001 reg 0.5 train accuracy: 0.530061222449 val accuracy: 0.479
lr 0.001 reg 0.6 train accuracy: 0.520591836735 val accuracy: 0.497
lr 0.01 reg 0.4 train accuracy: 0.100265306122 val accuracy: 0.087
lr 0.01 reg 0.5 train accuracy: 0.100265306122 val accuracy: 0.087
lr 0.01 reg 0.6 train accuracy: 0.100265306122 val accuracy: 0.087
best validation accuracy achieved during cross-validation: 0.497
```

```
1.
2. # 可视化最好的神经网络的权重
3. show_net_weights(best_net)
```

输出结果：



3.10 在测试集上测试

在调试完你的参数之后，你应该在测试集上测试你训练出的神经网络。你应该得到超过48%的准确率。每高于52%一个百分点，我们都将给予一分额外的分数奖励。

```
1. test_acc = (best_net.predict(X_test) == y_test).mean()
2. print('Test accuracy: ', test_acc)
```

输出结果：

```
Test accuracy: 0.474
```