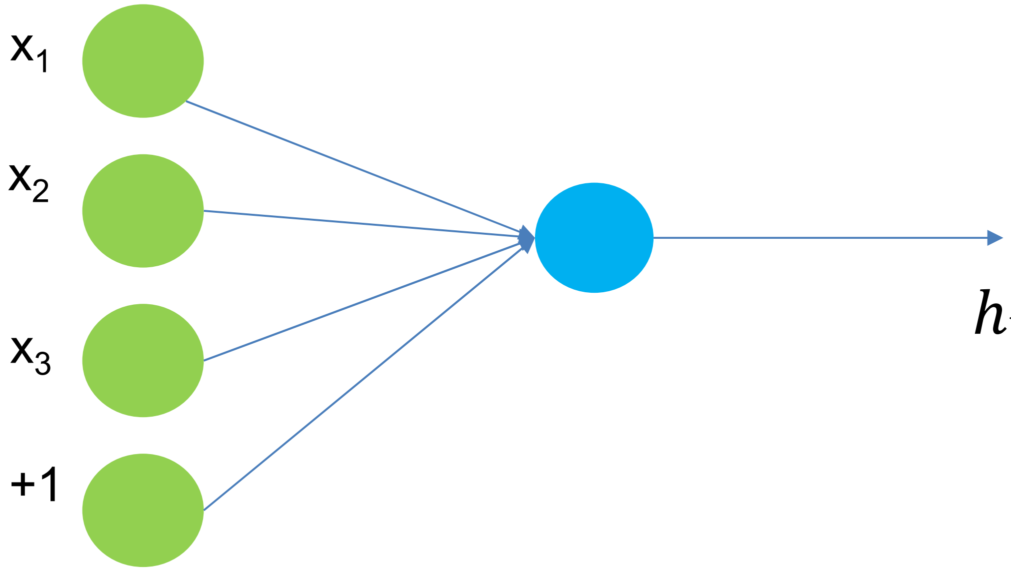
小型神经网络案例学习笔记

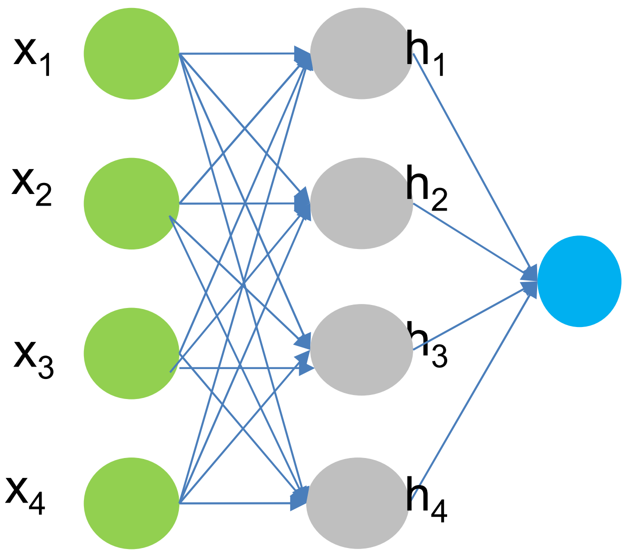
简单的线性分类器



$h_{W,b}(x)$

$$h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$$

具有一个隐含层的神经网络



$$h_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}\, x_3 + W_{14}\, x_4)$$

$$h_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}\, x_3 + W_{24}\, x_4)$$

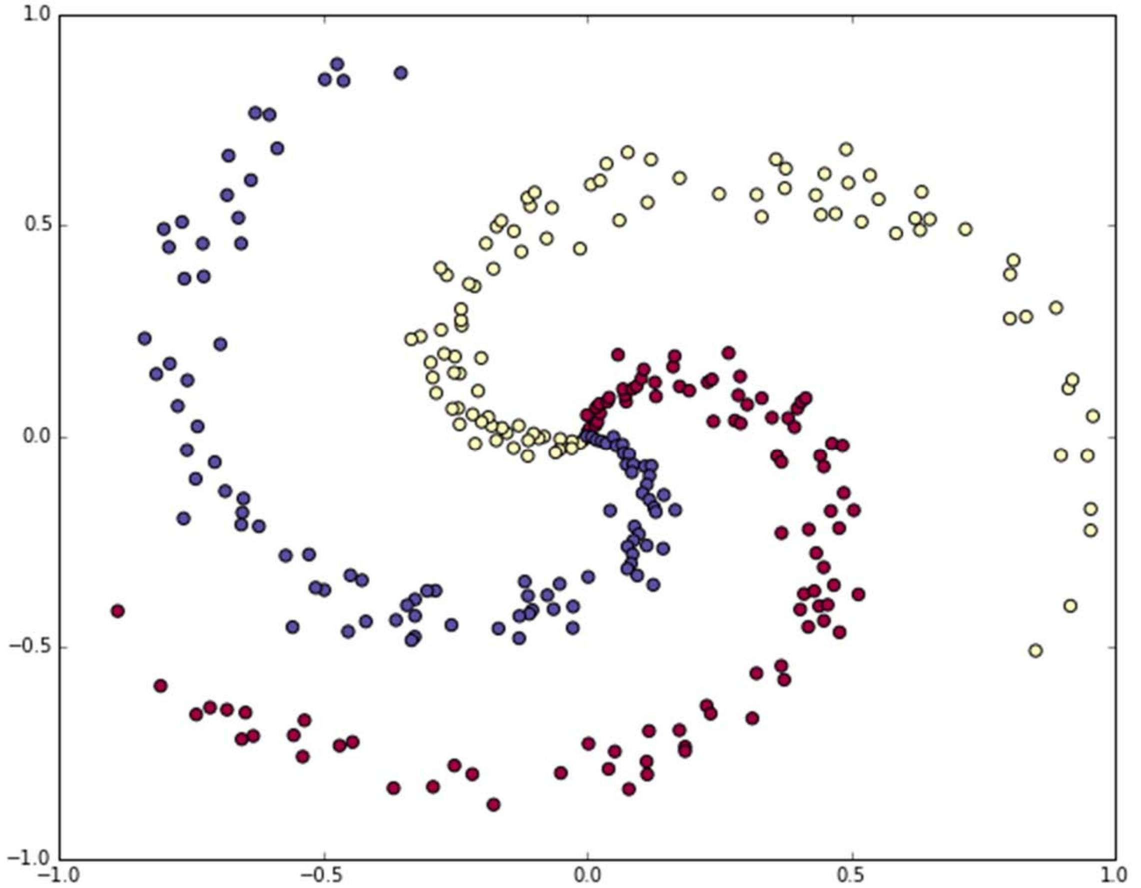
$$h_3 = f(W_{31}x_1 + W_{32}x_2 + W_{33}\, x_3 + W_{34}\, x_4)$$

$$h_4 = f(W_{41}x_1 + W_{42}x_2 + W_{43}\, x_3 + W_{44}\, x_4)$$

- 当将多个单元组合起来并具有分层结构时，就形成了神经网络模型。上图展示了一个具有一个隐含层的神经网络。

产生一些数据

- 首先生成一些不易线性分割的分类数据。本例中生成螺旋spiral dataset。



训练线性分类器
-初始化参数取值

- 线性分类器的参数包括一个权值矩阵 **W**以及一个偏置向量(bias vector) **b**. 首先需要初始化这些参数值为随机数：

```
# 随机初始化参数
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))
```

注意这里D=2是特征维度， K = 3 是类的个数.

训练线性分类器
–计算每个样本的类分数

- 使用矩阵乘积（matrix multiplication）计算每个样本的类分数：

  scores = np.dot(X, W) + b

- 本例中，X是300*2维，W是2*3维，因此结果是300*3维。每行表示该样本属于三个类的概率。B为1*3矩阵，分别加到每行上。

交叉熵损失函数

- 如果 **f** 是某样本 **i** 的类分数(class scores)数组，则Softmax classifier计算该样本的交叉熵损失函数为：$L_i = -\log\left(\frac{e^{f_{yi}}}{\sum_j e^{f_j}}\right)$
- Note: 分子中的 $\mathbf{f_{yi}}$ 表示预测结果中该样本 **i** 被预测为真实类的概率有多大；分母 $\mathbf{f_j}$ 表示该样本属于每个类 *j* 的概率。

- 对于这个表达式：数值总是在 0 和1之间.
- 当正确类的概率比较小 (near 0) 时，损失较大。相反，当正确类的概率较大，如接近1时，损失函数将较小，如接近0, 因为**log(1)=0**.

交叉熵损失函数

- 全部损失（full Softmax classifier loss ）是训练数据上的平均的交叉熵损失，并进行正则化regularization:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{data\ loss} + \underbrace{\frac{1}{2}\lambda \sum_k \sum_l W_{k,l}^2}_{regularization\ loss}\ \text{，其中} L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

- 损失函数的计算代码：

```
exp_scores = np.exp(scores)
#   probs 是个size为[300 x 3]的矩阵 。每行进行归一化（normalize ），表示该样本属于不同类的概率。每行的数值的和为1，相当于softmax操作。
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
# correct_logprobs是一个一维数组，包含每个样本所属正确类的概率。相当于Li
correct_logprobs = -np.log(probs[range(num_examples),y])
data_loss = np.sum(corect_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
```

反向传播梯度

- 在损失函数$L_i = -\log\left(\frac{e^{f_{yi}}}{\sum_j e^{f_j}}\right)$中，引入中间变量 **p**，则每个样本的损失为: $L_i = -\log(p_{y_i})$， $p_k = \frac{e^{f_k}}{\sum_j e^{f_j}}$
- 使用链式求导法则，求梯度（derive the gradient） $\frac{\partial L_i}{\partial f_k}$（k表示不同的类，需要对不同的k值进行求导），结果为：

  $$\frac{\partial L_i}{\partial f_k} = p_k - 1(y_i = k)$$
- 上述过程归结为如下代码（All of this boils down to the following code），即 在分数上的梯度dscores可以计算为 ：

```
dscores = probs    #  probs存储每个样本属于每个类的概率.
dscores[range(num_examples),y] -= 1
dscores /= num_examples
```

反向传播梯度

- $\frac{\partial L_i}{\partial f_k} = p_k - 1(y_i = k)$
- 假设我们计算得到的概率为 p = [0.2, 0.3, 0.5]，且正确的类是中间的那个 (with probability 0.3)．根据上述公式可得梯度 df = [0.2, -0.7, 0.5].

反向传播梯度

- 最后，由于scores = np.dot(X, W) + b，且刚刚算好了scores的梯度dscores（即损失函数对scores的梯度），所以可以反向传播（back propagate into ）给W and b:

```
dW = np.dot(X.T, dscores)
db = np.sum(dscores, axis=0, keepdims=True)
dW += reg*W   # 加上正则化梯度
```

Note：我们通过矩阵的乘法得到梯度部分，权重W的部分加上了正则化项的梯度。因为我们在设定正则化项的时候用了系数0.5 ($\frac{d}{dw}\left(\frac{1}{2}\lambda w^2\right) = \lambda w$).因此直接用$reg * W$就可以表示出正则化的梯度部分。.

参数更新

- 现在我们已经评价了每个参数如何影响损失函数。现在我们可以进行负梯度方向参数更新以降低损失

```
# perform a parameter update
W += -step_size * dW
b += -step_size * db
```

运行线性分类器

运行结果:

```
iteration 0: loss 1.096956
iteration 10: loss 0.917265
iteration 20: loss 0.851503
iteration 30: loss 0.822336
iteration 40: loss 0.807586
iteration 50: loss 0.799448
iteration 60: loss 0.794681
iteration 70: loss 0.791764
iteration 80: loss 0.789920
iteration 90: loss 0.788726
iteration 100: loss 0.787938
iteration 110: loss 0.787409
iteration 120: loss 0.787049
iteration 130: loss 0.786803
iteration 140: loss 0.786633
iteration 150: loss 0.786514
iteration 160: loss 0.786431
iteration 170: loss 0.786373
iteration 180: loss 0.786331
iteration 190: loss 0.786302
```

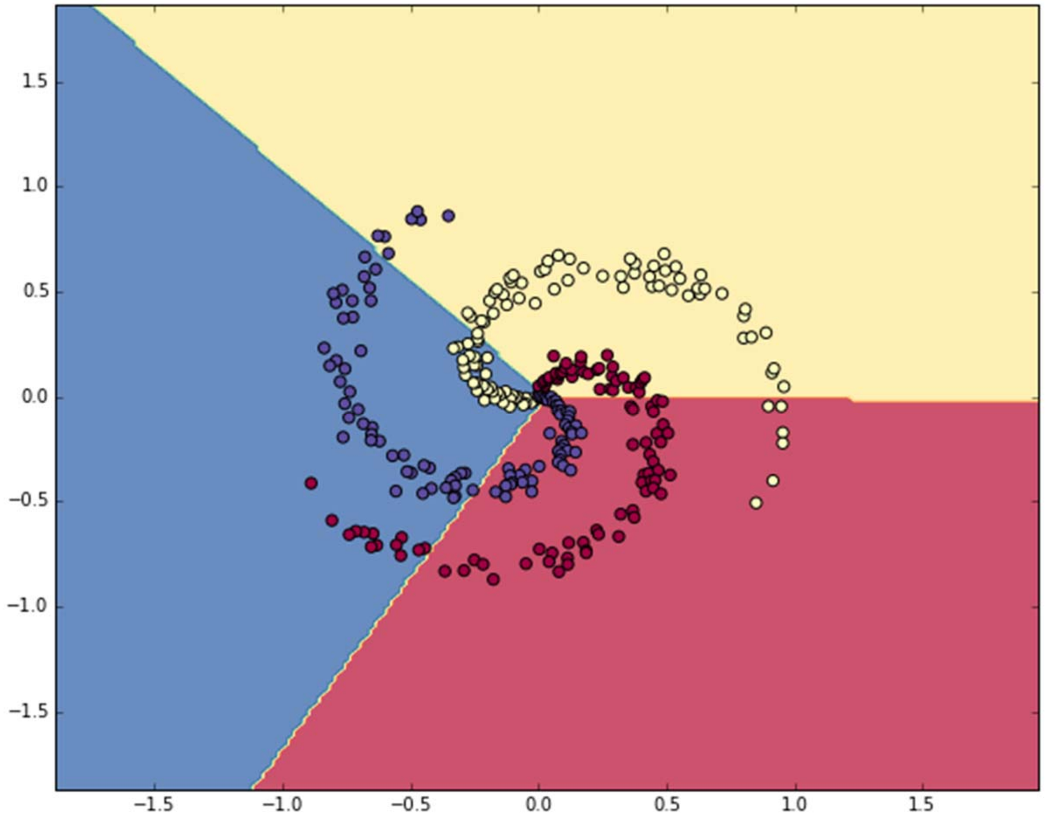
- 若干次循环后，结果大致收敛。

```
# evaluate training set accuracy
scores = np.dot(X, W) + b
predicted_class = np.argmax(scores, axis=1)
print 'training accuracy: %.2f' % np.mean(predicted_class == y))
```

- 输出结果可能不是很理想，这和使用的线性分类器有关。

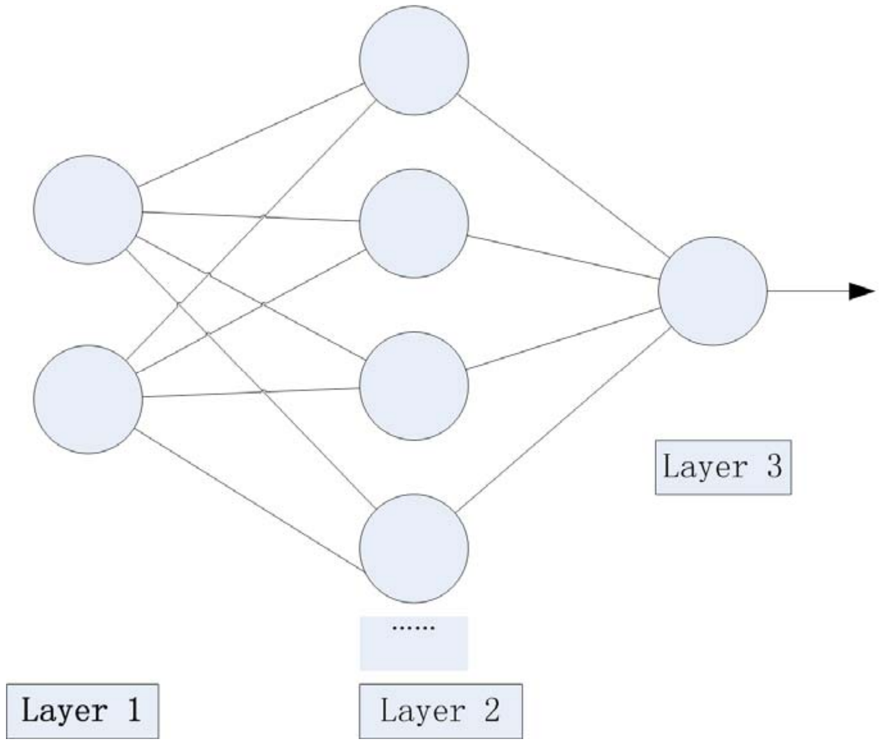
Putting it all together: Training a Softmax Classifier

- 可视化决策边界(decision boundaries)：

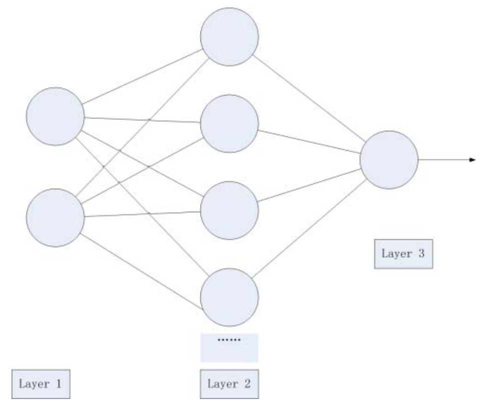


改造为一个 Neural Network

- 可以使用单隐藏层的神经网络：



改造为一个 Neural Network



- 用单隐藏层的神经网络，需要2层的权重和偏置：

```
# 随机初始化参数
h = 100 #  隐藏层的神经元个数
W = 0.01 * np.random.randn(D,h) #D=2为维数, 所以 2*100
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K) #k=3 为类数, 所以100*3
b2 = np.zeros((1,K))
```

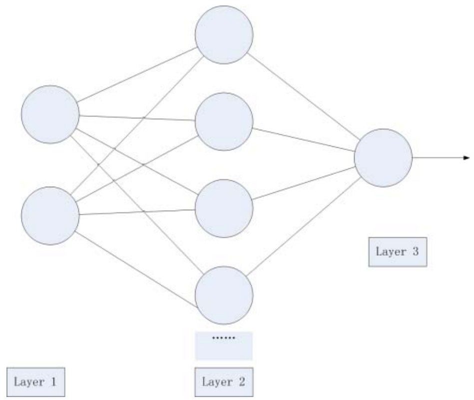
- 前向计算类分数的代码变为:

```
hidden_layer = np.maximum(0, np.dot(X, W) + b) #  ReLU activation
scores = np.dot(hidden_layer, W2) + b2
```

注意：（1）和之前线性分类器中的得分计算相比，多了一行代码计算，我们首先计算第一层神经网络结果，然后作为第二层的输入，计算最后的结果。

（2）这里使用的是 ReLu(Rectified Linear Units)激活函数，即f(x)=max(0,x)

改造为一个 Neural Network



- 计算loss以及梯度dscores：方法和原来一样
- 梯度更新：这里使用hidden_layer替换掉了之前的X:

```
# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, dscores)
db2 = np.sum(dscores, axis=0, keepdims=True)
```

- hidden_layer本身是一个包含其他参数和数据的函数，需要计算其梯度：:

```
dhidden = np.dot(dscores, W2.T)
```

改造为一个 Neural Network

- 现在得到了隐层输出的梯度，接着要反向传播到ReLU神经元。由于r=max(0,x)，有dr/dx=1(x>0)。用链式法则处理，因此有
  - 如果回传梯度大于0，经过ReLU之后，保持原有数值；
  - 如果回传梯度小于0，则数值为0。

```
# backprop the ReLU non-linearity#梯度回传经过ReLU
dhidden[hidden_layer <= 0] = 0
```

- 然后回到第一层，得到总权重和偏置梯度

```
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)
```

改造为一个 Neural Network

- 现在我们有了梯度 dW, db, dW2, db2，因此可以进行参数更新:

iteration 0: loss 1.098637
iteration 1000: loss 0.294416
iteration 2000: loss 0.266441
iteration 3000: loss 0.251507
iteration 4000: loss 0.248295
iteration 5000: loss 0.247106
iteration 6000: loss 0.246432
iteration 7000: loss 0.245957
iteration 8000: loss 0.245486
iteration 9000: loss 0.245220

- 此时通常能得到较高的准确率

改造为一个 Neural Network

- 再次可视化决策边界(decision boundaries)：