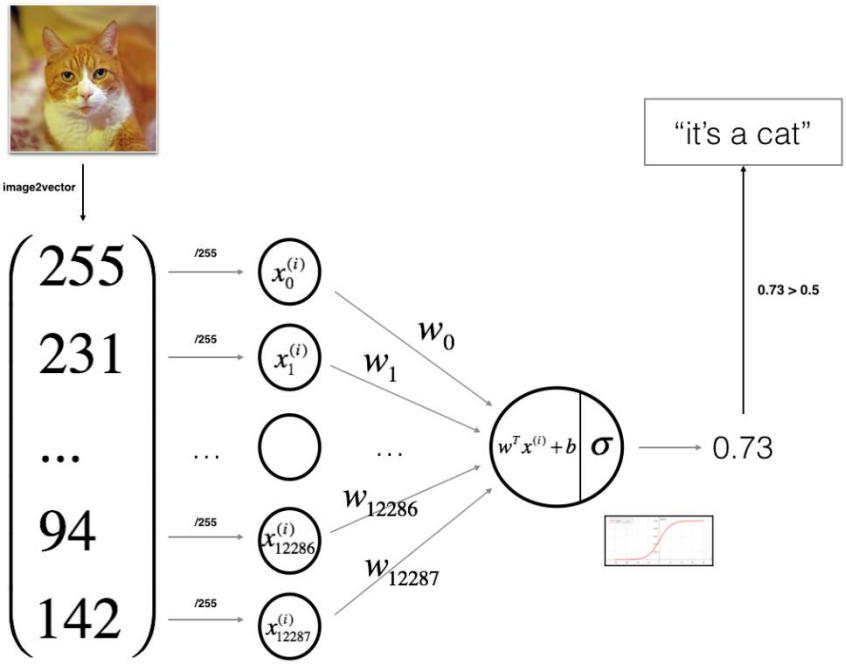
A. Logistic Regression in A Neural Network

罗杰斯特回归是最简单神经网络，如果不算上输入层的话只用一层，其基本结构如下所示：



在图像输入前我们先对其进行向量化操作，必要时我们也可以对其进行归一化操作（Normalizing）以加速梯度下降的速度。

1.1 Mathematical Expression of The Algorithm

To each $x^{(i)}$：

$$z^{(i)} = w^T x^{(i)} + b \tag{1.2.1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{1.2.2}$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \tag{1.2.3}$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{1.2.4}$$

1.2 Forward and Backward Propagation

Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, ..., a^{(m-1)}, a^{(m)})$
- You calculate the cost function: $J = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

Here are the two formulas you will be using:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A - Y)^T \tag{1.2.5}$$

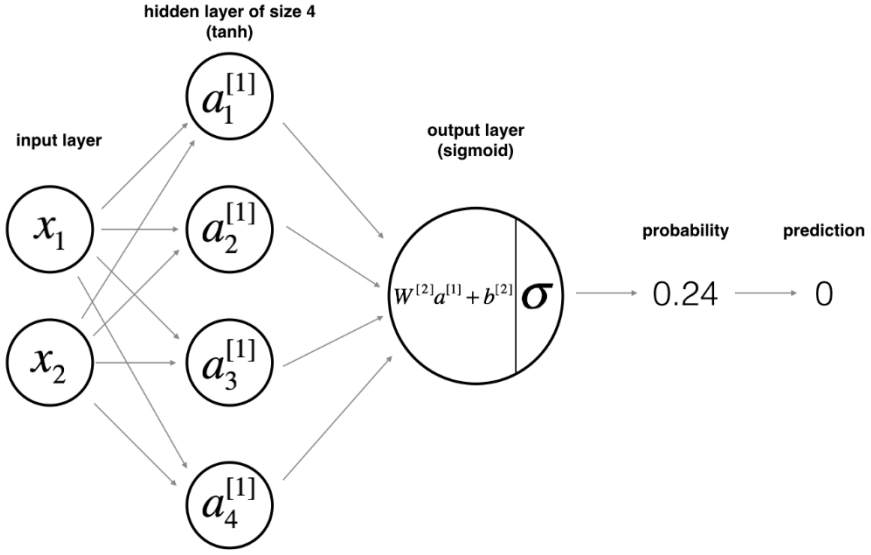
$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \tag{1.2.6}$$

For a parameter θ, the update rule is $\theta = \theta - \alpha d\theta$, where α is the learning rate.

B. Planar Data Classification With A Hidden Layer

2.1 Neural Network Model

我们这里设置了四个隐层，这个两层网络才是真正意义上的神经网络。



For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1](i)} \tag{1.3.1}$$

$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{1.3.2}$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2](i)} \tag{1.3.3}$$

$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{1.3.4}$$

$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{1.3.5}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right) \tag{1.3.6}$$

$$\frac{\partial \mathcal{J}}{\partial z_2^{(i)}} = \frac{1}{m}(a^{[2](i)} - y^{(i)}) \tag{1.3.8}$$

$$\frac{\partial \mathcal{J}}{\partial W_2} = \frac{\partial \mathcal{J}}{\partial z_2^{(i)}} a^{[1](i)T} \tag{1.3.9}$$

$$\frac{\partial \mathcal{J}}{\partial b_2} = \sum_i \frac{\partial \mathcal{J}}{\partial z_2^{(i)}} \tag{1.3.10}$$

$$\frac{\partial \mathcal{J}}{\partial z_1^{(i)}} = W_2^T \frac{\partial \mathcal{J}}{\partial z_2^{(i)}} * (1 - a^{[1](i)2}) \tag{1.3.11}$$

$$\frac{\partial \mathcal{J}}{\partial W_1} = \frac{\partial \mathcal{J}}{\partial z_1^{(i)}} X^T \tag{1.3.12}$$

$$\frac{\partial \mathcal{J}_i}{\partial b_1} = \sum_i \frac{\partial \mathcal{J}}{\partial z_1^{(i)}} \tag{1.3.13}$$

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]}a^{[1]T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]}A^{[1]T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T}dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T}dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]}x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]}X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Andrew Ng

其他比较容易证明，只证明 $\frac{\partial J}{\partial z_1}$，证明过程如下：

此处我们引入更加一般的情况我们 l 和 $l+1$ 的情况，做一下参数的定义，与吴恩达的定义有所区别：

- L：表示神经网络的层数；
- $m^{(l)}$：表示第 l 层神经元的个数；
- $f_l(\cdot)$：表示 l 层神经元的激活函数；
- $W^{(l)} \in \mathbb{R}^{m^{(l)} \times m^{l-1}}$：表示 $l-1$ 层到第 l 层的权重矩阵；
- $\mathbf{b}^{(l)} \in \mathbb{R}^{m^l}$：表示 $l-1$ 层到第 l 层的偏置；
- $\mathbf{z}^{(l)} \in \mathbb{R}^{m^l}$：表示 l 层神经元的净输入（净活性值）；
- $\mathbf{a}^{(l)} \in \mathbb{R}^{m^l}$：表示 l 层神经元的输出（活性值）。|

根据链式法则我们可以写出损失函数关于每个元素的偏导数：

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial w_{ij}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial w_{ij}^{(l)}} \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}},$$
$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}.$$

其中我们只需要计算其中的三项，便可以对其他逐个求解，因此可见在反向传播中求解顺序也异常重要。

（1）计算偏导数 $\frac{\partial \mathbf{z}^{(l)}}{\partial w_{ij}^{(l)}}$ 因 $\mathbf{z}^{(l)} = W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$，偏导数

$$\frac{\partial \mathbf{z}^{(l)}}{\partial w_{ij}^{(l)}} = \left[\frac{\partial z_1^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \frac{\partial z_{m^{(l)}}^{(l)}}{\partial w_{ij}^{(l)}}\right] \tag{4.48}$$
$$= \left[0, \cdots, \frac{\partial(\mathbf{w}_{i:}^{(l)}\mathbf{a}^{(l-1)} + b_i^{(l)})}{\partial w_{ij}^{(l)}}, \cdots, 0\right] \tag{4.49}$$
$$= \left[0, \cdots, a_j^{(l-1)}, \cdots, 0\right] \tag{4.50}$$
$$\triangleq \mathbb{I}_i(a_j^{(l-1)}) \quad \in \mathbb{R}^{m^{(l)}}, \tag{4.51}$$

其中 $\mathbf{w}_{i:}^{(l)}$ 为权重矩阵 $W^{(l)}$ 的第 i 行，$\mathbb{I}_i(a_j^{(l-1)})$ 表示第 j 个元素为 $a_j^{(l-1)}$，其余为 0 的行向量。

（2）计算偏导数 $\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}}$ 因为 $\mathbf{z}^{(l)}$ 和 $\mathbf{b}^{(l)}$ 的函数关系为 $\mathbf{z}^{(l)} = W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$，因此偏导数

$$\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} = \mathbf{I}_{m^{(l)}} \quad \in \mathbb{R}^{m^{(l)} \times m^{(l)}}, \tag{4.52}$$

为 $m^{(l)} \times m^{(l)}$ 的单位矩阵。

（3）计算误差项 $\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}$ 我们用 $\delta^{(l)}$ 来定义第 l 层神经元的误差项，

$$\delta^{(l)} = \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}} \in \mathbb{R}^{m^{(l)}}. \tag{4.53}$$

误差项 $\delta^{(l)}$ 来表示第 l 层神经元对最终损失的影响，也反映了最终损失对第 l 层神经元的敏感程度。误差项也间接反映了不同神经元对网络能力的贡献程度，从而比较好地解决了“贡献度分配问题”。

根据 $\mathbf{z}^{(l+1)} = W^{(l+1)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)}$，有

$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} = (W^{(l+1)})^{\mathrm{T}}. \tag{4.54}$$

根据 $\mathbf{a}^{(l)} = f_l(\mathbf{z}^{(l)})$，其中 $f_l(\cdot)$ 为按位计算的函数，因此有

$$\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} = \frac{\partial f_l(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}} \tag{4.55}$$

$$= \mathrm{diag}(f_l'(\mathbf{z}^{(l)})). \tag{4.56}$$

因此，根据链式法则，第 l 层的误差项为

$$\delta^{(l)} \triangleq \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}} \tag{4.57}$$

$$= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l+1)}} \tag{4.58}$$

$$= \mathrm{diag}(f_l'(\mathbf{z}^{(l)})) \cdot (W^{(l+1)})^{\mathrm{T}} \cdot \delta^{(l+1)} \tag{4.59}$$

$$= f_l'(\mathbf{z}^{(l)}) \odot \big((W^{(l+1)})^{\mathrm{T}}\delta^{(l+1)}\big), \tag{4.60}$$

其中 $\odot$ 是向量的点积运算符，表示每个元素相乘。

从公式 (4.60) 可以看出，第 l 层的误差项可以通过第 $l+1$ 层的误差项计算得到，这就是误差的反向传播。反向传播算法的含义是：第 l 层的一个神经元的误差项（或敏感性）是所有与该神经元相连的第 $l+1$ 层的神经元的误差项的权重和。然后，再乘上该神经元激活函数的梯度。

在计算出上面三个偏导数之后，公式 (4.46) 可以写为

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial w_{ij}^{(l)}} = \mathbb{I}_i(a_j^{(l-1)})\delta^{(l)} = \delta_i^{(l)} a_j^{(l-1)}. \tag{4.61}$$

进一步，$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ 关于第 l 层权重 $W^{(l)}$ 的梯度为

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial W^{(l)}} = \delta^{(l)}(\mathbf{a}^{(l-1)})^{\mathrm{T}}. \tag{4.62}$$

同理，$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ 关于第 l 层偏置 $\mathbf{b}^{(l)}$ 的梯度为

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}. \tag{4.63}$$

在计算出每一层的误差项之后，我们就可以得到每一层参数的梯度。因此，基于误差反向传播算法（Backpropagation，BP）的前馈神经网络训练过程可以分为以下三步：

1. 前馈计算每一层的净输入 $\mathbf{z}^{(l)}$ 和激活值 $\mathbf{a}^{(l)}$，直到最后一层；
2. 反向传播计算每一层的误差项 $\delta^{(l)}$；
3. 计算每一层参数的偏导数，并更新参数。

由此我们可以得到随机梯度下降的误差反向传播算法的具体训练过程：

算法 4.1: 基于随机梯度下降的反向传播算法

输入: 训练集 $\mathcal{D}=\{(\mathbf{x}^{(n)},y^{(n)})\}_{n=1}^{N}$, 验证集 $\mathcal{V}$, 学习率 α, 正则化系数 λ, 网络层数 L, 神经元数量 $m^{(l)}$, $1\leq l\leq L$.

随机初始化 $W,\mathbf{b}$;

repeat

  对训练集 $\mathcal{D}$ 中的样本随机重排序;

  for $n=1\cdots N$ **do**

    从训练集 $\mathcal{D}$ 中选取样本 $(\mathbf{x}^{(n)},y^{(n)})$;

    前馈计算每一层的净输入 $\mathbf{z}^{(l)}$ 和激活值 $\mathbf{a}^{(l)}$, 直到最后一层;

    反向传播计算每一层的误差 $\delta^{(l)}$; // 公式 (4.60)

    // 计算每一层参数的导数

    $\forall l,\quad \frac{\partial \mathcal{L}(\mathbf{y}^{(n)},\hat{\mathbf{y}}^{(n)})}{\partial W^{(l)}}=\delta^{(l)}(\mathbf{a}^{(l-1)})^{\mathrm{T}}$; // 公式 (4.62)

    $\forall l,\quad \frac{\partial \mathcal{L}(\mathbf{y}^{(n)},\hat{\mathbf{y}}^{(n)})}{\partial \mathbf{b}^{(l)}}=\delta^{(l)}$; // 公式 (4.63)

    // 更新参数

    $W^{(l)}\leftarrow W^{(l)}-\alpha(\delta^{(l)}(\mathbf{a}^{(l-1)})^{\mathrm{T}}+\lambda W^{(l)})$;

    $\mathbf{b}^{(l)}\leftarrow \mathbf{b}^{(l)}-\alpha\delta^{(l)}$;

  end

until 神经网络模型在验证集 $\mathcal{V}$ 上的错误率不再下降;

输出: $W,\mathbf{b}$

C. Improving Deep Neural Networks:Hyperparameter Tuning,Regularization and Optimization

3.1 Initialization

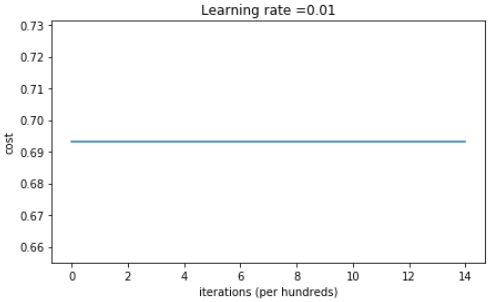
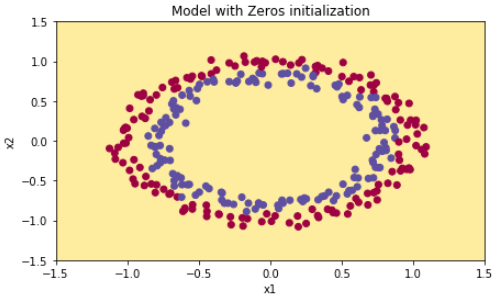
Zero initialization：意味着每层的每个神经元将会学到相同的东西，相当于训练一个一层网络

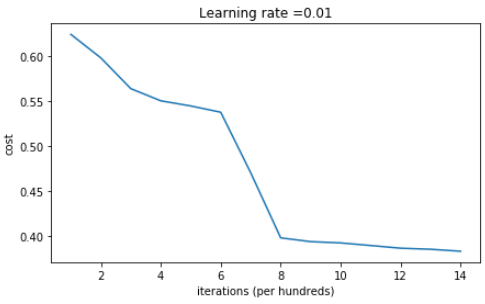
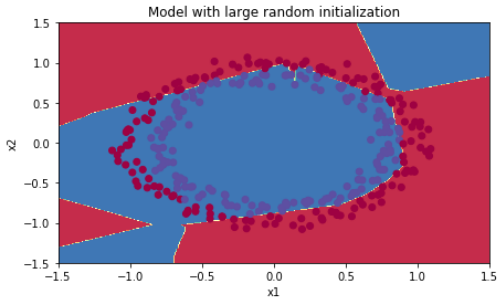
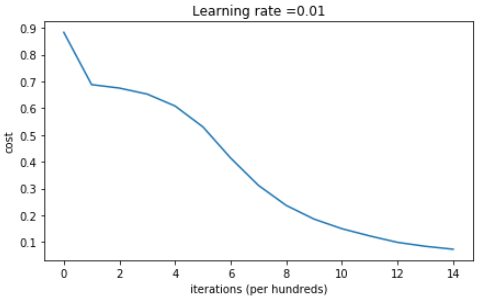
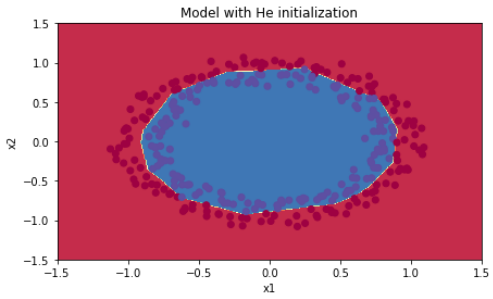
Random initialization：初始化的值需要尝试，一般情况下较小的初始化值表现更好

He initialization：对权重 *W[l]*乘以 *sqrt(2./layers_dims[l-1])*.)

3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

右上表我们可以看出 He initialization 的效果最好，尤其如果激活函数是 ReLU 函数时其效果更好。不同的初始化导致不同的结果；随机初始化保证了不同的隐层单元能够学到不同的事情；不要初始化太大的值。

The way of initialization	Plot of loss	Plot of decision boundary
Zero initailization		

Random initialization		
He initialization		

3.2 Regularization

深度学习中存在过拟合的原因是因为高方差，其中一个解决方法就是正则化。这个很容易理解，直观上说激活函数中，如果输出的 z 扩展为比较大或者比较小的数的话，激活函数开始变得非线性。如果 W 很小，则 z 很小，z 在小范围内移动，使得激活函数输出大致成线性。

L2 regularization：

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

我们有时也称其为权重衰减，我们都试图让它变得更小，实际上，相当于我们给矩阵 W 乘以$(1-a\lambda/m)$倍的权重，矩阵W减去$a\lambda/m$倍的它，也就是用这个系数$(1-a\lambda/m)$乘以矩阵W，该系数小于 1。对于其前向传播只需要修改 loss 函数计算公式即可，反向传播我们需要给 dW1, dW2 and dW3 增加正则项的梯度，其计算公式为：

$$(\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W).$$

Dropout Regularization:

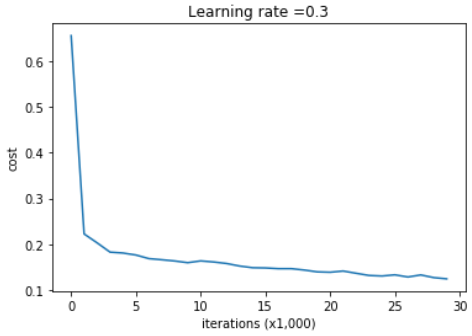
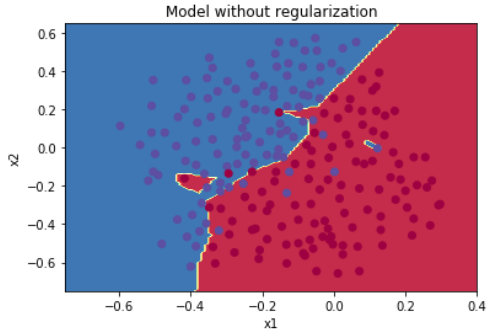
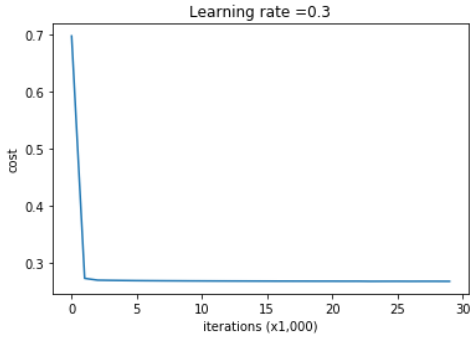
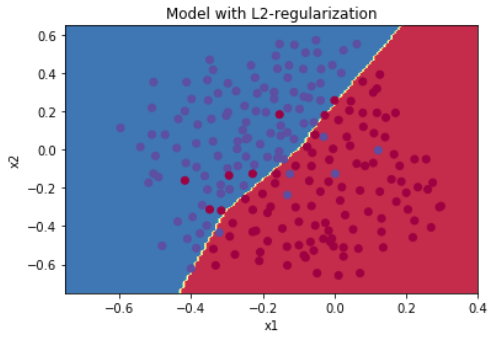
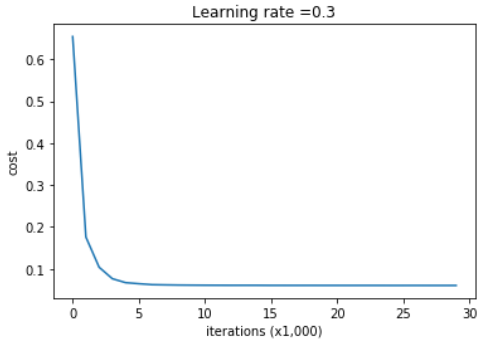
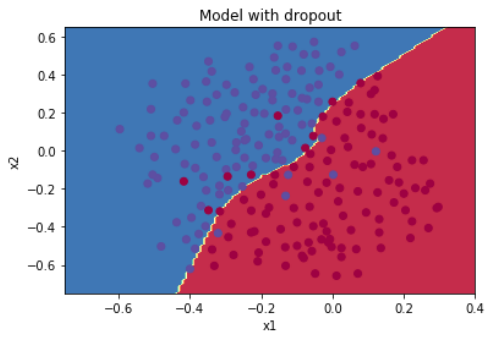
它随机的去掉每一次迭代中的一些神经元，我们前向传播和反向传播中据需要应用 dropout，因此我们在计算 A 或者 dA 时之前乘以 D 之后，我们需要对其再除以 keep_prob。此外，我们只在训练时使用 dropout 而不再测试时使用。下面阐述其前向和反向传播的基本步骤：

前向传播：
①我们实施 DROPOUT 通过一个随机设置的矩阵 D 来实现
②根据阈值，将 D 转换为 0-1 矩阵
③A=A*D
④A=A/KEEP_PRO

反向传播：
①我们实施 DROPOUT 通过一个随机设置的矩阵 D 来实现
②根据阈值，将 D 转换为 0-1 矩阵
③dA=dA*D
④dA = dA/KEEP_PRO

model	train accuracy	test accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

直观上理解：不要依赖于任何一个特征，因为该单元的输入可能随时被清除，因此该单元通过这种方式传播下去，并为单元的四个输入增加一点权重，通过传播所有权重，dropout 将产生收缩权重的平方范数的效果，和之前讲的 *L2* 正则化类似。正则化类似；实施 dropout 的结果实它会压缩权重，并完成一些预防过拟合的外层正则化； *L2* 对不同权重的衰减是不同的，它取决于倍增的大小。

The way of regularization	Plot of loss	Plot of decision boundary
None regularization		
L2 regularization		
Dropout Regularization		

3.3 Gradient Checking

前向传播容易计算的，因此我们假设前向传播是正确的，我们使用梯度检测来判断反向传播是否正确。此外梯度检测速度较慢，我们并不是在整个训练的过程中均使用它，而是一小段时间会使用它。还有需要注意的是，我们使用梯度检测时就不能再使用 dropout。下面我们给出了算法的思路：

对于一维的梯度下降，我们根据公式：

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

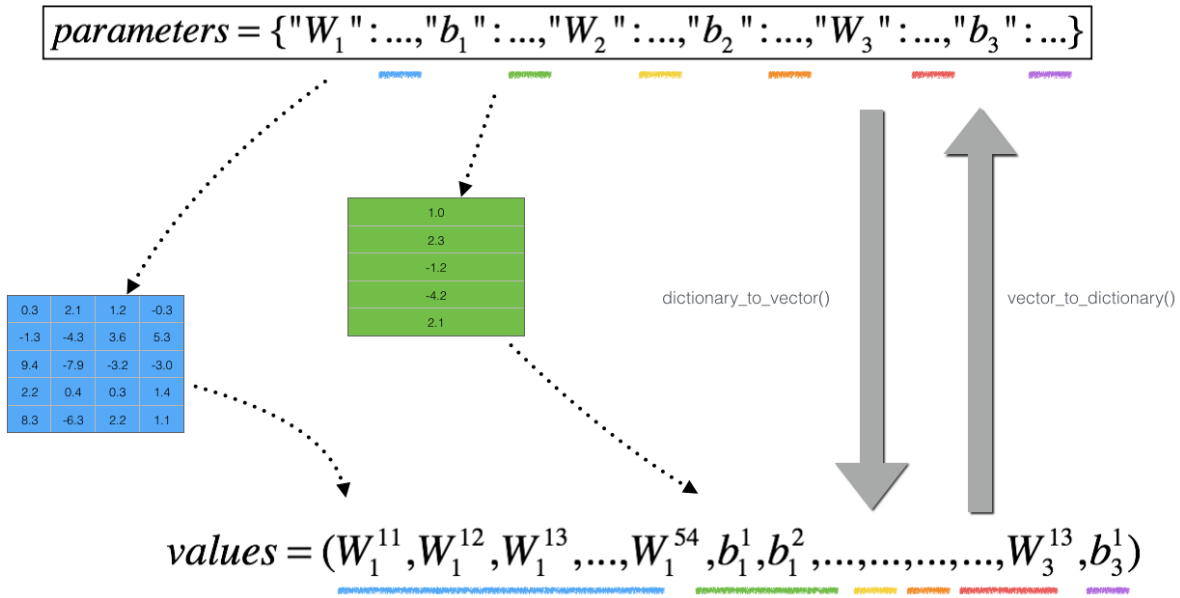
可见，我们需要计算五个参数：

1. $\theta^+ = \theta + \varepsilon$
2. $\theta^- = \theta - \varepsilon$
3. $J^+ = J(\theta^+)$
4. $J^- = J(\theta^-)$
5. $gradapprox = \frac{J^+ - J^-}{2\varepsilon}$

最终，根据此可以计算梯度和 gradapprox 之间的差别：

$$difference = \frac{|| \, grad - gradapprox \, ||_2}{|| \, grad \, ||_2 + || \, gradapprox \, ||_2}$$

对于 N 维的梯度下降，我们首先需要将 parameter 转换为 vector，示意图如图所示：



下面我们给出了整个算法的 pseudo-code，过程如下：

Algorithm 2.1.1 N-dimensional gradient checking

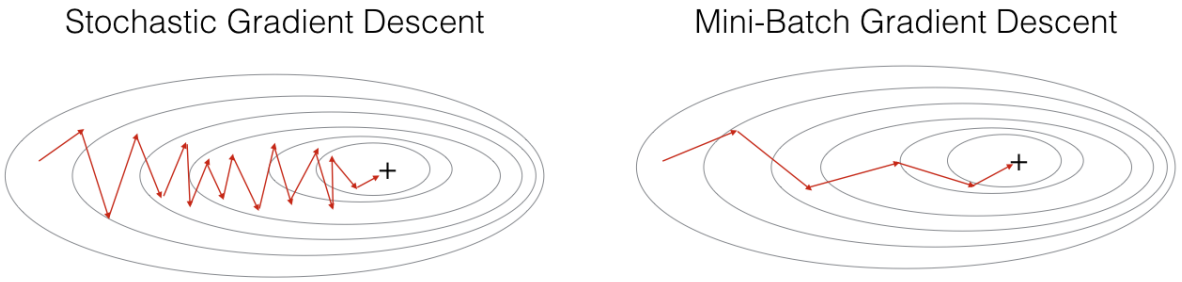
```
for i in num_parameters do
    Calculate: J_plus[i]
        Set θ+ to np.copy(parameters_values)
        Set θ+_i to θ+_i + ε
        Calculate J+_i using to forward_propagation_n(x, y, vector_to_dictionary(θ+))
    Calculate J_minus[i]:
        do the same thing with θ−
    Compute gradapprox[i] = (J+_i − J−_i) / 2ε
end for
```

3.4 Optimization

3.4.1 Gradient Descent

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]}$$

(Batch) Gradient Descent 和 Stochastic Gradient Descent 不同在于，随机梯度下降只使用一个训练事例，所以当训练集较大时，SGD 速度更快。但是 SGD 导致了更多的抖动为了达到平衡，可以从下图中看出：



3.4.2 Mini-Batch Gradient Descent

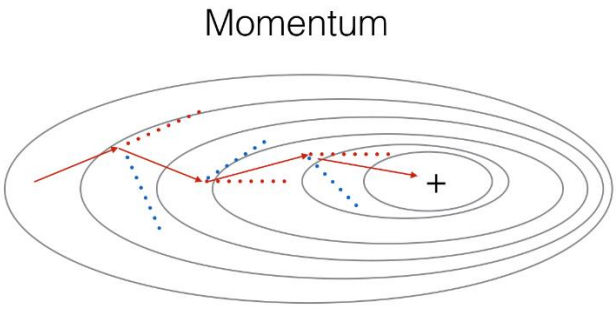
mini-batch 梯度下降法，顾名思义就是将训练集分割为小一点的子集进行训练，使用 batch 梯度下降法时，每次迭代你都需要历遍整个训练集，可以预期每次迭代成本都会下降，所以如果成本函数 J是迭代次数的一个函数，它应该会随着每次迭代而减少；使用 mini-batch 梯度下降法，如果你作出成本函数在整个过程中的图，则并不是每次迭代都是下降的。

用 mini-batch 梯度下降法，我们从这里开始，一次迭代这样做，两次，三次，四次，它不会总朝向最小值靠近，但它比随机梯度下降要更持续地靠近 最小值的方向，它也不一定在很小的范围内收敛或者波动。

其算法实现主要是两个步骤，第一步为 Shuffle，随机改变 X 和 Y 的一些列，第二步为 Partition 即进行划分。需要注意的是，如果大小不能整数划分，则使用向下取整。

3.4.3 Momentum

动量梯度下降法，运行速度几乎总是快于标准的梯度下降算法，简而言之，基本的想法就是计算梯度的指数加权平均数，并利用该梯度更新你的权重。动量梯度下降法考虑了过去的梯度来平滑更新，我们用变量 v 存储先前梯度的方向。下图中红线表示的是动量梯度下降法，而蓝色的线表示的是 mini-batch 梯度下降，可见动量梯度下降法更快：



动量梯度下降法的计算公式如下所示，它可以应用于 batch gradient descent, mini-batch gradient descent or stochastic gradient descent：

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

3.4.4 Adam

Adam 优化算法基本上就是将 Momentum 和 RMSprop 结合在一起，其计算公式如下所示，它需要两个变量 v 和 s 来存储过去梯度的平均指和平方值：

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1-\beta_1) \frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v_{dW^{[l]}}^{corrected} = \frac{v_{dW^{[l]}}}{1-(\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1-\beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s_{dW^{[l]}}^{corrected} = \frac{s_{dW^{[l]}}}{1-(\beta_1)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v_{dW^{[l]}}^{corrected}}{\sqrt{s_{dW^{[l]}}^{corrected}} + \varepsilon} \end{cases} \tag{2.2.5}$$

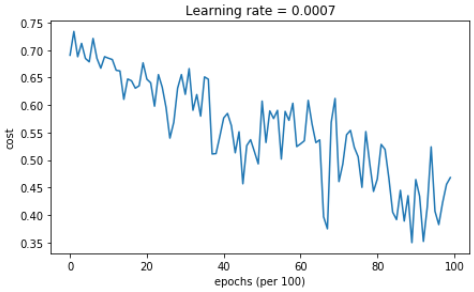
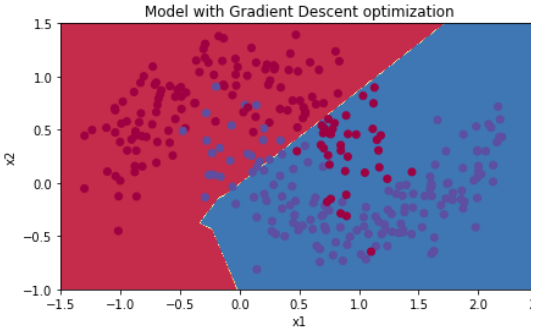
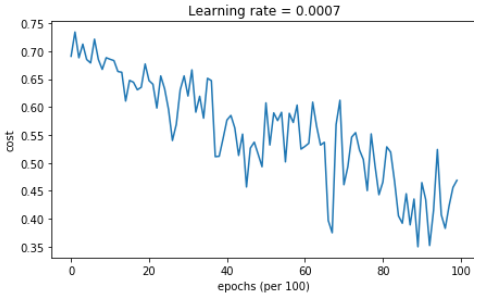
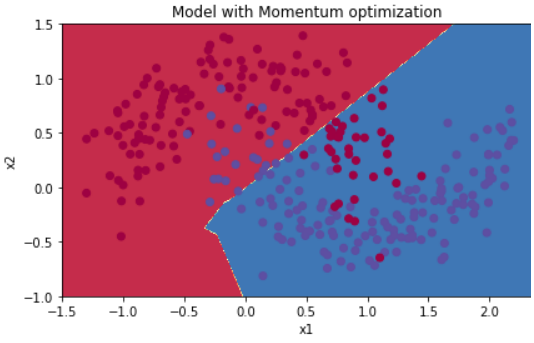
where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

我们在 moons 数据集上测试，得到了如下结果：

optimization method	accuracy	cost shape
Gradient descent	79.7%	oscillations
Momentum	79.7%	oscillations
Adam	94%	smoother

Momentum 通常是有用的，但是鉴于学习率较低和数据集过于简单，它的影响几乎可以忽略不计。也可以清晰地看出 Adam 梯度下降迅速趋于收敛，其优势主要体现在较少的内存要求，另一方面即使很少调整超参数，Adam 通常也能很好地工作。

The way of optimization	Plot of loss	Plot of decision boundary
Gradient descent		
Momentum		
Adam