Lect1. Neural Networks and Deep Learning

Chap1. Introduction to Deep Learning

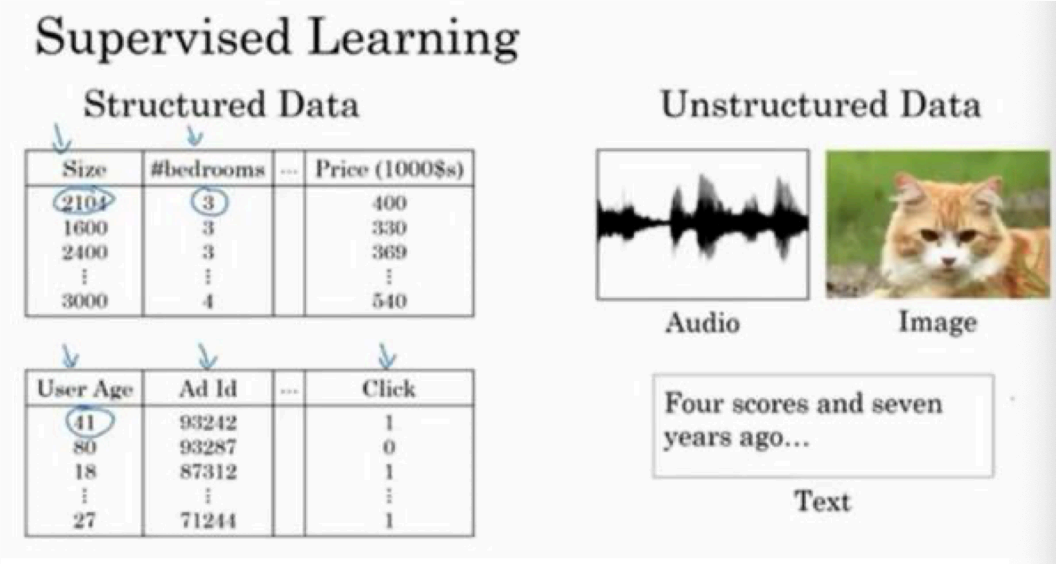
1.2 Introduction to Neural Network

1. **ReLU (Rectified Linear Unit)激活函数**：rectify（修正）可以理解成 $\max(0, x)$.

1.3 Supervised Learning with Neural Networks

1. 对于图像应用，经常使用卷积神经网络(Convolutional Neural Network, CNN)

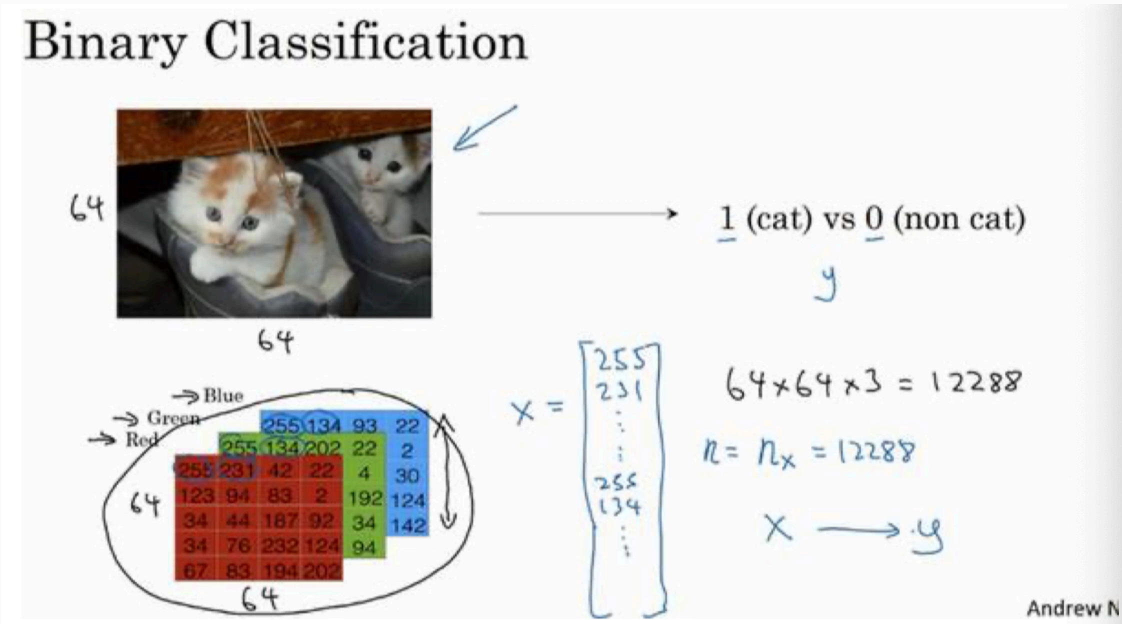
2. 对于序列数据（尤其是一维序列），例如音频、语言等，经常使用递归神经网络 (Recurrent Neural Network, RNN)
3. 结构化数据：每个特征都有一个很好的定义。
4. 非结构化数据：比如原始音频，或者你想要识别的图像或文本中的内容。这里的特征可能是图像中的像素值或文本中的单个单词。



Chap2. Basics of Neural Network Programming

2.1 Binary Classification 二分类

1. 逻辑回归 (logistic regression) 是一个是一个用于二分类 (Binary Classification) 的算法。
2. 使用 n_x 或 n 来表示输入特征向量 x 的维度。



3. 符号说明

- x : 表示一个 n_x 维数据，为输入数据，维度为 $(n_x, 1)$;
- y : 表述输出结果，取值为 $(0, 1)$;
- $(x^{(i)}, y^{(i)})$: 表示第 i 组数据，可能是训练数据，也可能是测试数据，此处默

认为训练数据；

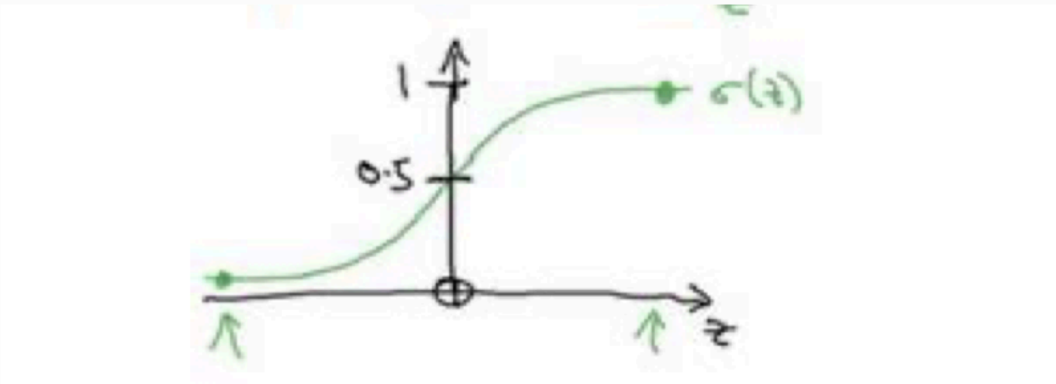
- $X=[x^{(1)},x^{(2)},\ldots,x^{(m)}]$: 表示所有训练数据集的输入值，放在一个 $n_x \times m$ 的矩阵中，其中 m 表示样本数目；
- $Y=\left[y^{(1)},y^{(2)},\ldots,y^{(m)}\right]$: 对应所有训练数据集的输出值，维度为 $1\times m$。

$$X=\left[x^{(1)}x^{(2)}\cdots x^{(m)}\right]$$

2.2 Logistic Regression 逻辑回归

1. $\hat{y}$: 对实际值 y 的估计，让 $\hat{y}$ 表示 y 等于 1 的一种可能性，前提条件是给定了输入特征 X。
2. w : 逻辑回归的参数，实际是特征权重，维度与特征向量相同，为 n_x 维向量。
3. b : 逻辑回归的实数参数，用来表示偏差。
4. **sigmoid函数**：

$$\sigma(z)=\frac{1}{1+e^{-z}}$$



定义 $\hat{y}=\sigma(\theta^T x)$ 的 sigmoid 函数来限定范围

2.3 Logistic Regression Cost Function 逻辑回归的代价函数

$$\hat{y}^{(i)}=\sigma\left(w^T x^{(i)}+b\right),\ \text{where } \sigma(z)=\frac{1}{1+e^{-z}}$$

$$\text{Given }\left\{\left(x^{(1)},y^{(1)}\right),\ldots,\left(x^{(m)},y^{(m)}\right)\right\},\ \text{want } \hat{y}^{(i)}\approx y^{(i)}$$

- 损失函数，又称误差函数，用来衡量算法的运行情况，**Loss function:** $L(\hat{y},y)$.
- 逻辑回归中的损失函数是：$L(\hat{y},y)=-y\log(\hat{y})-(1-y)\log(1-\hat{y})$
- 逻辑回归中的代价函数

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L\left(\hat{y}^{(i)},y^{(i)}\right)=\frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}\log\hat{y}^{(i)}-\left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right)$$

2.4 Gradient Descent 梯度下降法

在测试集上，通过最小化代价函数（成本函数）$J(w,b)$ 来训练的参数 w 和 b.

假设我们要求 $f(x_1,x_2)$ 的最小值，起始点为 $x^{(1)}=\left(x_1^{(1)},x_2^{(1)}\right)$，则在 $x^{(1)}$ 点处的梯度为$\nabla\left(f\left(x^{(1)}\right)\right)=\left(\frac{\partial f}{\partial x_1^{(1)}},\frac{\partial f}{\partial x_2^{(1)}}\right)$，我们可以进行第一次梯度下降来更新 x:

$$x^{(2)}=x^{(1)}-\alpha * \nabla f\left(x^{(1)}\right)$$

其中，α 表示学习率 (**learning rate**)，用于控制步长 (**step**)，这样我们就得到了下一个点 $x^{(2)}$，重复上面的步骤，直到函数收敛，此时可认为函数取得了最小值。在实际应用中，我们可以设置一个精度 ϵ， 当函数在某一点的梯度的模小于 ϵ 时，就可以终止迭代。

逻辑回归的代价函数（成本函数）$J(w, b)$ 有两个参数：

$$w:=w-\alpha \frac{\partial J(w, b)}{\partial w}$$

$$b:=b-\alpha \frac{\partial J(w, b)}{\partial b}$$

2.9 Logistic Regression Gradient Descent 逻辑回归中的梯度下降

假设现在只考虑单个样本的情况，单个样本的代价函数定义如下：

$$L(a, y)=-(y \log (a)+(1-y) \log (1-a))$$

其中 a 是逻辑回归的输出， y 是样本的标签值。

$$z=w^{T} x+b$$

$$\hat{y}=a=\sigma(z)$$

$$\mathcal{L}(a, y)=-(y \log (a)+(1-y) \log (1-a))$$

通过微积分可得到

$$\frac{d L(a, y)}{d a}=-y / a+(1-y) /(1-a)$$

$$\frac{d L(a, y)}{d z}=\left(\frac{d L}{d a}\right) \cdot\left(\frac{d a}{d z}\right)=\left(-\frac{y}{a}+\frac{(1-y)}{(1-a)}\right) \cdot(a \cdot(1-a))=a-y$$

单个样本的梯度下降算法：

1. 计算 $dz=(a-y)$
2. 计算 $dw_1=x_1 \cdot dz$, $dw_2=x_2 \cdot dz$, $db=dz$
3. 梯度下降： $w_1:=w_1-\alpha dw_1$, $w_2:=w_2-\alpha dw_2$

2.10 Gradient Descent on m Examples m个样本的梯度下降

一步梯度下降代码流程

```
J = 0; dw1 = 0; dw2 = 0; db = 0;
for i = 1 to m
    z(i) = wx(i) + b;
    a(i) = sigmoid(z(i));
    J += -[y(i)log(a(i)) + (1 - y(i))log(1 - a(i))];
    dz(i) = a(i) - y(i);
```

```
    dw1 += x1(i)dz(i);
    dw2 += x2(i)dz(i);
    db += dz(i);
J /= m;
dw1 /= m;
db /= m;
w -= alpha*dw;
b -= alpha*db;
```

2.11 Vectorization 向量化

非向量化方法计算 $z = w^T x + b$

```
z = 0;
for i in range(n_x)
    z += w[i]*x[i];
z += b;
```

使用向量化直接计算 $w^T x + b$

```
z = np.dot(w,b) + b
```

Chap3. Shallow Neural Networks 浅层神经网络

3.1 Neural Network Overview

公式 3.3:

$$\left.\begin{array}{r} x \\ W^{[1]} \\ b^{[1]} \end{array}\right\} \implies z^{[1]} = W^{[1]}x + b^{[1]} \implies a^{[1]} = \sigma(z^{[1]})$$

公式 3.4:

$$\left.\begin{array}{r} x \\ dW^{[1]} \\ db^{[1]} \end{array}\right\} \impliedby dz^{[1]} = d(W^{[1]}x + b^{[1]}) \impliedby d\alpha^{[1]} = d\sigma(z^{[1]})$$

3.6 Activation Functions 激活函数

1. $a = \tan(z)$
2. $a = \tanh(z)$

- $g\left(z^{[1]}\right) = \tanh\left(z^{[1]}\right)$ 的效果总是优于 **sigmoid** 函数，因为函数值域在 -1 和 $+1$ 之间的激活函数，其均值是更接近零的。

- 二分类问题是一个例外，对于输出层，因为 y 的值是 0 或 1，所以想让 $\hat{y}$ 的数值介于 0 和 1 之间，而不是在 -1 和 $+1$ 之间。所以需要用 **sigmoid** 激活函数。
- **sigmoid** 函数和 **tanh** 函数两者共同的缺点是，在 z 特别大或者特别小的情况下，导数的梯度或者函数的斜率会变得特别小，最后就会接近于 0，导致降低梯度下降的速度。
- 修正线性单元函数 **ReLu**： $a = \max(0, z)$
- **Leaky ReLu**: 当 z 为负值时，函数轻微倾斜。
- **ReLu**函数优点：
  1. 在 z 的区间变动很大的情况下，激活函数的导数或者激活函数的斜率都会远大于 0，在程序实现就是一个 **if-else** 语句，而 **sigmoid** 函数需要进行浮点四则运算，在实践中， 使用 **ReLu** 激活函数神经网络通常会比使用 **sigmoid** 或者 **tanh** 激活函数学习的更快。
  2. **sigmoid** 和 **tanh** 函数的导数在正负饱和区的梯度都会接近于 0，这会造成梯度弥散，而 **ReLu** 和 **Leaky ReLu** 函数大于 0 部分都为常数，不会产生梯度弥散现象。
- 不能再隐藏层用线性激活函数，唯一可以用线性激活函数的通常就是输出层。

3.8 Derivatives of Activation Functions

1. **sigmoid activation function**

$$a = g(z) = \frac{1}{1 + e^{-Z}}$$

$$g(z)' = \frac{d}{dz} g(z) = a(1 - a)$$

2. **Tanh activation function**

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\frac{d}{\mathrm{d}z} g(z) = 1 - (\tanh(z))^2$$

3. **Rectified Linear Unit**

$$g(z) = \max(0, z)$$

$$g(z)' = \begin{cases} 0 & \text{if} \quad z < 0 \\ 1 & \text{if} \quad z > 0 \\ undefined & \text{if} \quad z = 0 \end{cases}$$

4. **Leaky Linear Unit (Leaky ReLu)**

$$g(z) = \max(0.01z, z)$$

$$g(z)' = \begin{cases} 0.01 & \text{if} \quad z < 0 \\ 1 & \text{if} \quad z > 0 \\ undefined & \text{if} \quad z = 0 \end{cases}$$

3.9 Gradient Descent for Neural Networks

- 二分类任务成本函数

$$J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right) = \frac{1}{m} \sum_{i=1}^{n} L(\hat{y}, y)$$

- 正向传播方程

  (1) $z^{[1]} = W^{[1]} x + b^{[1]}$

  (2) $a^{[1]} = \sigma\left(z^{[1]}\right)$

  (3) $z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

  (4) $a^{[2]} = g^{[2]}\left(z^{[z]}\right) = \sigma\left(z^{[2]}\right)$

- 反向传播方程

  1. $dz^{[2]} = A^{[2]} - Y, Y = \begin{bmatrix} y^{[1]} & y^{[2]} & \cdots & y^{[m]} \end{bmatrix}$
  2. $dW^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$
  3. $db^{[2]} = \frac{1}{m} np \cdot \operatorname{sum}\left(dz^{[2]}, \text{ axis } = 1, \text{ keepdims } = \text{True}\right)$
  4. $dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'} * z^{[1]}$
  5. $dW^{[1]} = \frac{1}{m} dz^{[1]} x^{T}$
  6. $db^{[1]} = \frac{1}{m} np.\operatorname{sum}\left(dz^{[1]}, axis = 1, keepdims = True\right)$

3.11 Random Initialization

- 权重随机初始化是很重要的，对于逻辑回归，把权重初始化为 0 是可以的，但对于一个神经网络，把权重或者参数全部初始化为 0，那么梯度下降就不再有作用。
- 随机初始化参数：`W[1] = np.random.randn(2, 2)`（生成高斯分布），通常再乘上一个小的数，比如 0.01，这样把它初始化为一个很小的随机数。b 没有这个对称问题（称为 **symmetry breaking problem**），所以可以把 b 初始化为 0

```
W[1] = np.random.randn(2, 2) * 0.01;
b[1] = np.zeros((2, 1));
W[2] = np.zeros(2, 2) * 0.01;
b[2] = 0;
```

- 如果 w 很大，那么可能最终停在 z 很大的值，这回造成 **tanh/sigmoid** 激活函数饱和在龟速的学习上。若不用 **sigmoid/tanh** 就不存在这样的问题。

Chap4. Deep Neural Networks

4.2 Forward and Backward Propagation

1. 前向传播：输入 $a^{[l-1]}$，输入 $a^{[l]}$，缓存为 $z^{[l]}$

$$z[l]=W[l] \cdot A[l-1]+b[l]$$

$$A^{[l]}=g^{[l]}\left(Z^{[l]}\right)$$

2. 反向传播：输入为 $da^{[l]}$，输出为 $da^{[l-1]}, dw^{[l]}, db^{[l]}$

$$(1)\ dZ^{[l]}=dA^{[l]} * g^{[l]\prime}\left(Z^{[l]}\right)$$

$$(2)\ dW^{[l]}=\frac{1}{m} dZ^{[l]} \cdot A^{[l-1] T}$$

$$(3)\ db^{[l]}=\frac{1}{m} np.\operatorname{sum}\left(dz^{[l]}, axis=1, \text{ keepdims }=\text{True}\right)$$

$$(4)\ dA^{[l-1]}=W^{[l] T} \cdot dZ^{[l]}$$

4.3 Getting your Matrix Dimensions Right

做深度神经网络的反向传播时，一定要确认所有的矩阵维数是前后一致的，可以大 大提高代码通过率。

1. $w^{[l]}:\left(n^{[l]}, n^{[l-1]}\right)$
2. $b^{[l]}:\left(n^{[l]}, 1\right)$
3. $z^{[l]}, a^{[l]}:\left(n^{[l]}, 1\right)$

4.7 Parameters vs. Hyperparameters

- 什么是超参数
    - learning rate a 学习率
    - iterations 梯度下降法循环的数量
    - L 隐藏层的数目
    - $n^{[l]}$ 隐藏层单元数目
    - activation function 激活函数的选择

Lect2. Improving Deep Neural Networks-Hyperparameter tuning, Regularization and Optimization

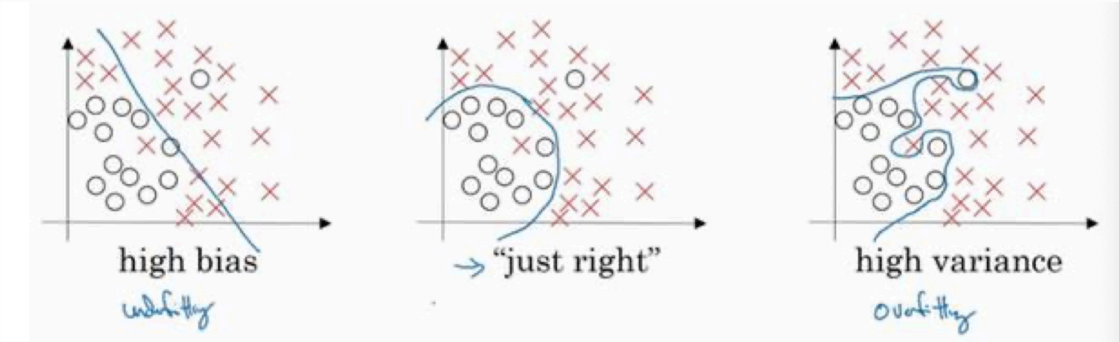
Chap1. Practical Aspects of Deep Learning

1.1 Train/Dev/Test Sets

- 训练数据的划分
    - 训练集
    - 验证集（简单交叉验证集）
    - 测试集

- 在训练集上训练，尝试不同的模型框架，在验证集上评估这些模型，然后迭代并选出适用的模型。
- 在机器学习中，如果只有一个训练集和一个验证集，而没有独立的测试集，遇到这种情 况，训练集还被人们称为训练集，而验证集则被称为测试集，不过在实际应用中，人们只是 把测试集当成简单交叉验证集使用，并没有完全实现该术语的功能。

1.2 Bias/Variance 偏差/方差



1. 如果给这个数据集拟合一条直线，可能得到一个逻辑回归拟合，但它并不能很好地拟合该数据，这是高偏差(**high bias**)的情况，我们称为"欠拟合"(**underfitting**)
2. 如果我们拟合一个非常复杂的分类器，比如深度神经网络或含有隐藏单元的神经网络，可能就非常适用于这个数据集，但是这看起来也不是一种很好的拟合方式分类器方差较高（**high variance**），数据过度拟合（**overfitting**）。
3. 在两者之间，可能还有一些像图中这样的，复杂程度适中，数据拟合适度的分类器，这个数据拟合看起来更加合理，我们称之为"适度拟合"（**just right**）是介于过度拟合和欠拟合中间的一类。

1.4 Regularization

- 深度学习中解决过拟合问题——高方差的两个解决方法：
  - 正则化
  - 准备更多数据
- 正则化有助于避免过度拟合，减少网络误差。
- $L2$ 范数：$\|x\|_2 = \sqrt{\sum_i x_i^2}$
- 逻辑回归函数中加入正则化

$$\min_{w,b} J(w,b), \quad w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

$$L2 \ \ regulation : \|w\|_2^2 = \sum_{y=1}^{n_x} w_j^2 = w^T w$$

- 只正则化参数 w : 因为 w 通常是一个高维参数矢量，已经可以表达高偏差问题，w 可能包含有很多参数，我们不可能拟合所有参数，而 b 只是单个数字。

- $L1$ 正则化：

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}\|w\|_1=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j|$$

- 如果用 $L1$ 正则化，那么 w 最终会是稀疏的，也就是说 w 向量中优很多 0，
- λ 是正则化参数，通常用验证集或交叉验证集来配置这个参数，尝试各种各样的数据，寻找最好的参数，我们要考虑训练集之间的权衡，把参数设置为较小值，这样可以避免过拟合，所以 λ 是个需要调整的超参。
- 弗罗贝尼乌斯范数 **Frobenius Norm**

$$\|w^{[l]}\|_F^2=\sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(w_{ij}^{[l]})$$

- 使用 **Frobenius Norm** 实现梯度下降

  用 **backprop** 计算出 dW 的值，**backprop** 会给出 J 对 W 的偏导数，实际上是 $W^{[l]}$，然后$W^{[l]}=W^{[l]}-\alpha dW^{[l]}$

$$dW^{[l]}=(\text{from backprop})+\frac{\lambda}{m}w^{[l]},\ \frac{\partial J}{\partial w^{[l]}}=\partial w^{[l]}$$

$$W^{[l]}=W^{[l]}-\alpha\left[(\text{from backprop})+\frac{\lambda}{m}W^{[l]}\right]$$

$$=W^{[l]}-\frac{\alpha\lambda}{m}W^{[l]}-\alpha(\text{from backprop})$$

- $L2$ 正则化也称为"权重衰减”

1.5 Why Regularization Reduces Overfitting?

- 直观上理解就是如果正则化参数 λ 设置得足够大，权重矩阵 W 被设置为接近于 0 的值，即把多隐藏单元的权重设为 0，于是基本上消除了这些隐藏单元的许多影响。
- 但是 λ 会存在一个中间值，于是会有一个接近“Just Right”的中间状态。

1.6 Dropout Regularization 随机失活正则化

1. **Inverted Dropout** 反向随机失活