过拟合、欠拟合和Dropout

过拟合现象

- 出现过拟合，得到的模型在训练集上的准确率很高，但在真实的场景中识别率确很低。
- 也就是模型在真实场景下的预测效果不好

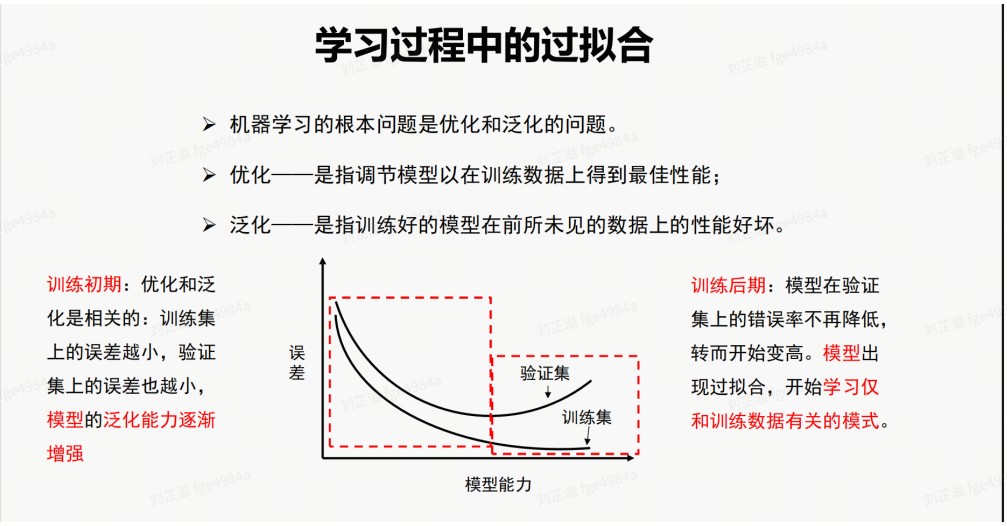
过拟合与欠拟合

过拟合——是指学习时选择的模型所包含的参数过多，以至于出现这一模型对已知数据预测的很好，但对未知数据预测得很差的现象。这种情况下模型可能只是记住了训练集数据，而不是学习到了数据特征。

欠拟合——模型描述能力太弱，以至于不能很好地学习到数据中的规律。产生欠拟合的原因通常是模型过于简单。

学习过程中的过拟合

- 优化：更好地拟合训练样本，令损失趋近于0，模型预测值正确
- 训练集优化模型参数；验证集用来度量模型泛化能力



- 过拟合产生的根本原因：学习到的数据集中训练数据中的特性在真实世界并不普遍存在，令模型不具有泛化能力

解决欠拟合

- 增加网络复杂度：增加网络层数，增加神经元的个数

应对过拟合

- 我们设计网络一般会朝过拟合方向设计网络，因而可能出现过拟合；我们需要考虑如何应对过拟合

应对过拟合

最优方案——获取更多的训练数据

次优方案——调节模型允许存储的信息量或者对模型允许存储的信息加以约束，该类方法也称为正则化。

- 调节模型大小
- 约束模型权重，即权重正则化（常用的有L1、L2正则化）
- 随机失活（Dropout）

L2正则化

- 权值分散可以抑制过拟合：依赖输入图像的所有特征，而不是仅仅依赖某几个特征
- **决策面会变得简单平滑**，不会学习到任意形状的决策面

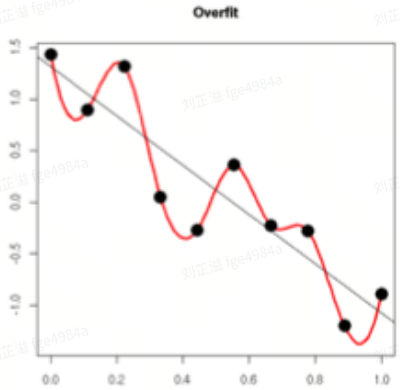
L2正则化

$$L(W) = \underbrace{\frac{1}{N}\sum_{i} L_i\left(f(x_i, W), y_i\right)}_{\text{数据损失}} + \underbrace{\lambda R(W)}_{\text{权重正则损失}}$$

L2正则损失：$R(W) = \sum_{k}\sum_{l} W_{k,l}^2$

L2正则损失对于大数值的权值向量进行严厉惩罚，鼓励更加分散的权重向量，使模型倾向于使用所有输入特征做决策，此时的模型泛化性能好！

过拟合的时候，拟合函数的系数往往非常大，为什么？如下图所示，过拟合，就是拟合函数需要顾忌每一个点，最终形成的拟合函数波动很大。在某些很小的区间里，函数值的变化很剧烈。这就意味着函数在某些小区间里的导数值（绝对值）非常大，由于自变量值可大可小，所以只有系数足够大，才能保证导数值很大。



而正则化是通过约束参数的范数使其不要太大，所以可以在一定程度上减少过拟合情况。

随机失活(Dropout)

随机失活（Dropout）

- 随机失活：让隐层的神经元以一定的概率不被激活。
- 实现方式：训练过程中，对某一层使用Dropout，就是随机将该层的一些输出舍弃(输出值设置为0），这些被舍弃的神经元就好像被网络删除了一样。
- 随机失活比率（ Dropout ratio）：是被设为 0 的特征所占的比例，通常在 0.2——0.5 范围内。

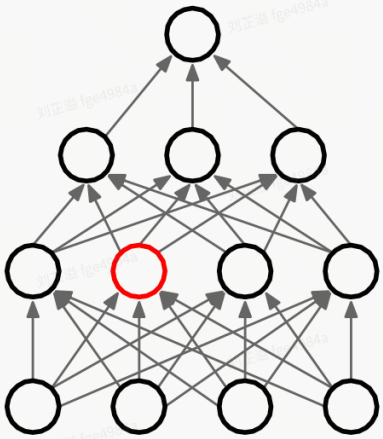
例:假设某一层对给定输入样本的返回值应该是向量：[0.2, 0.5, 1.3, 0.8, 1.1]。
使用Dropout后，这个向量会有几个随机的元素变成：[0, 0.5, 1.3, 0, 1.1]。

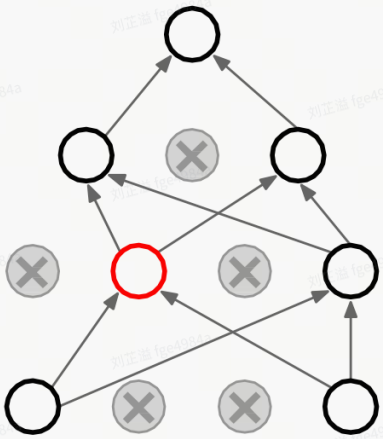
随机失活（Dropout）

随机失活为什么能够防止过拟合呢？

解释1：随机失活使得每次更新梯度时参与计算的网络参数减少了，降低了模型容量，所以能防止过拟合。

解释2：
随机失活鼓励权重分散，从这个角度来看随机失活也能起到正则化的作用，进而防止过拟合。



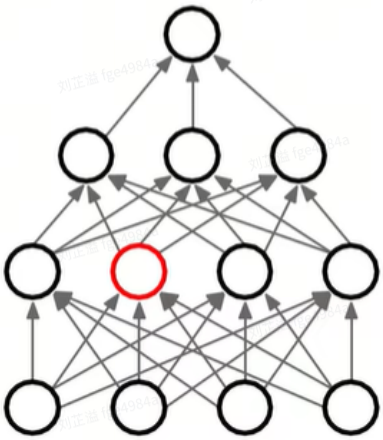


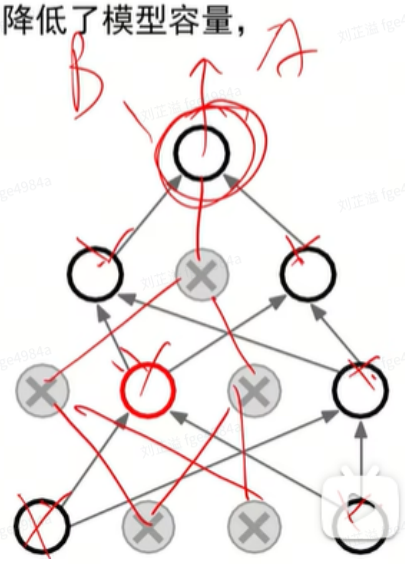
随机失活为什么能够防止过拟合呢？

解释1：随机失活使得每次更新梯度时参与计算的网络参数减少了，降低了模型容量，所以能防止过拟合。

解释2：随机失活鼓励权重分散，从这个角度来看随机失活也能起到正则化的作用，进而防止过拟合。

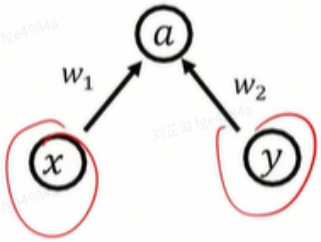
解释3：Dropout可以看作模型集成





随机失活的应用

存在的问题：



测试阶段：$E[a] = w_1x + w_2y$

训练阶段：$E[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + w_2y) + \frac{1}{4}(0x + 0y) = \frac{1}{2}(w_1x + w_2y)$

随机失活的应用

三层神经网络示例：

```
p = 0.5 #神经元保持激活状态的概率，该值越高失活的单元就越少
def train(X):
    H1 = np.maximum(0, np.dot(W1,X) + b1)
    U1 = np.random.rand(*H1.shape) < p #第一层的mask
    H1 *= U1 #第一层dropout操作
    H2 = np.maximun(0, np.dot(W2,H1) + b2)
    U2 = np.random.rand(*H2.shape) < p #第二层的mask
    H2 *= U2 #第二层dropout操作
    out = np.dot(W3,H2) + b3

def predict(X):
    H1 = np.maximum(0, np.dot(W1,X) + b1) * p
    H2 = np.maximun(0, np.dot(W2,H1) + b2) * p
    out = np.dot(W3,H2) + b3
```

随机失活的应用

三层神经网络示例：

```
p = 0.5 #神经元保持激活状态的概率，该值越高失活的单元就越少
def train_new(X):
    H1 = np.maximum(0, np.dot(W1,X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p #第一层的mask,注意除了p
    H1 *= U1
    H2 = np.maximun(0, np.dot(W2,H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p #第二层的mask,注意除了p
    H2 *= U2
    out = np.dot(W3,H2) + b3

def predict_new(X):
    H1 = np.maximum(0, np.dot(W1,X) + b1)
    H2 = np.maximun(0, np.dot(W2,H1) + b2)
    out = np.dot(W3,H2) + b3
```