

Deep Learning From Andrew Ng

第一课时：基本概念

1. 神经网络擅长于处理非结构化数据，大数据。
2. 逻辑回归：我的理解：线性回归+非线性函数—>广泛的应用于二分类问题

多重线性回归直接将 $w^T x + b$ 作为因变量，即 $y = w^T x + b$ ，而logistic回归则通过函数 L 将 $w^T x + b$ 对应一个隐状态 p ， $p = L(w^T x + b)$ ，然后根据 p 与 $1-p$ 的大小决定因变量的值（0或者1）。如果 L 是logistic函数，就是logistic回归，如果 L 是多项式函数就是多项式回归。

逻辑回归表达式：

$$\hat{y} = \sigma(w^T x + b)$$
$$\text{当 } \theta_0 = b; \theta_1 \dots \theta_{n_x} = w;$$
$$\hat{y} = \sigma(\theta^T x)$$

损失函数的求导转换为一个凸优化问题：

$$L(\hat{y}, y) = -y * \log(\hat{y}) - (1 - y) * \log(1 - \hat{y})$$

when $y = 1$, \hat{y} will be possibly big in order to make L smaller

when $y = 0$, \hat{y} will be possibly small in order to make L smaller

可以从上述这两个角度理解逻辑回归的损失函数

AndrewNg提供了另一种解释：

$$\text{if } y = 1 : p(y|x) = \hat{y}$$
$$\text{if } y = 0 : p(y|x) = 1 - \hat{y}$$
$$\hat{y} = p(y = 1|x)$$
$$p(y|x) = \hat{y}^y * (1 - \hat{y})^{1-y}$$

取对数： $L(\hat{y}, y) = -y * \log(\hat{y}) - (1 - y) * \log(1 - \hat{y})$

3. 梯度下降算法 本质是一个凸优化问题，求损失函数偏导，不断迭代，得到全局最优
4. 向量化的速度>for循环的速度

```
z = np.dot(w.T, X) + b # 充分利用CPU, GPU并行化计算
```

5. numpy Broadcasting

后缘长度相同，或者有1维度=1，可以进行广播

```
x = np.array([1, 2, 3])
```

```

>>> x

array([1, 2, 3])

>>> y = np.array([[1], [2], [3]])

>>> y

array([[1],
       [2],
       [3]])

>>> x+y

array([[2, 3, 4],
       [3, 4, 5],
       [4, 5, 6]])

>>> x*y

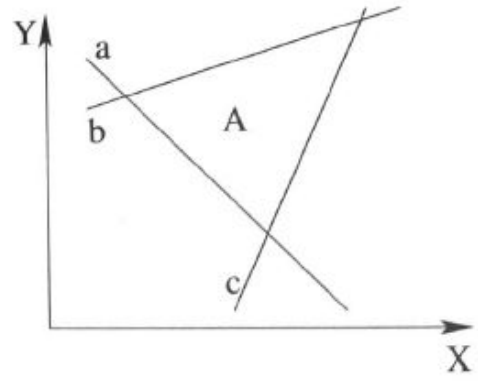
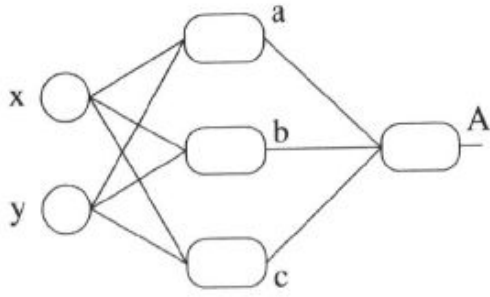
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])

>>> np.dot(x, y)

array([14])

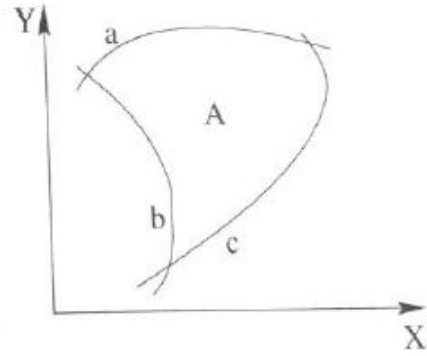
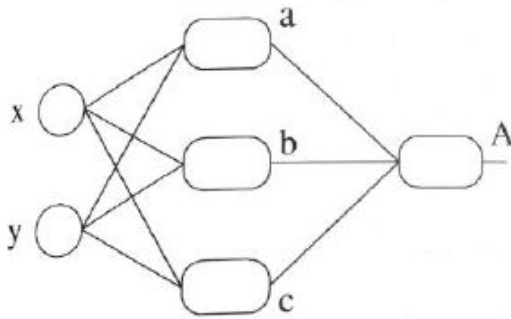
```

6. **不需要使用一维数组**即($n,$) 这种情况。使用assert语句去避免可能出现的问题。
7. **激活函数**：定义了给定输入后输出的集合。用来强化神经网络的表达能力
 当用**线性函数**时，所能表达的**区间范围有限**，从而划分的区间也有限



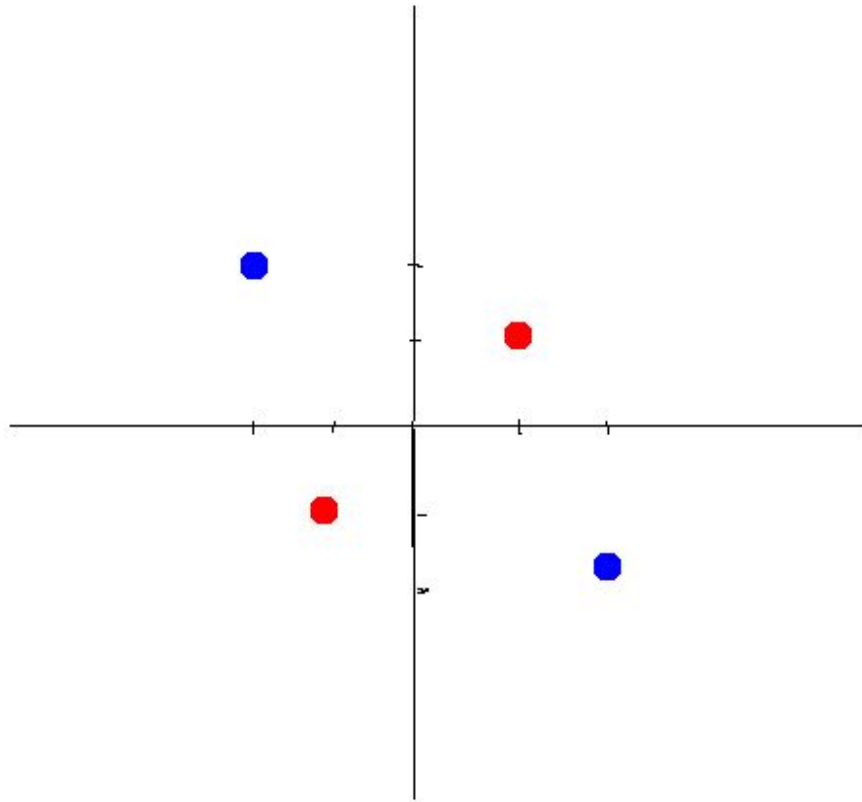
with step activation function

当用非线性函数时，表达能力增强



with sigmoid activation function

当遇到这种分类的时候，必须需要非线性函数作为划分

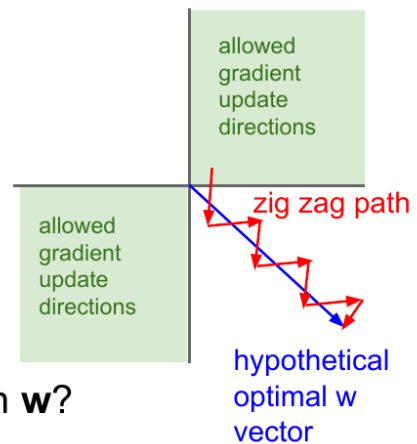


Sigmoid: 容易出现梯度弥散的情况，出现绝对值大的数，神经元无法更新

导致Zigzag现象

Consider what happens when the input to a neuron is always positive...

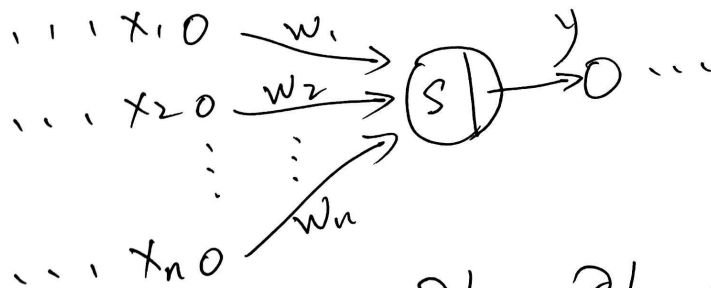
$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on \mathbf{w} ?

Always all positive or all negative :(
(this is also why you want zero-mean data!)

sigmoid 激活函数.



$$S = \sum_{i=1}^n w_i x_i$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_i}$$

$$= \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial S} \cdot \frac{\partial S}{\partial w_i}$$

$$\frac{\partial y}{\partial S} = \frac{d}{ds} \left(\frac{1}{1+e^{-s}} \right)$$

$$= \frac{\partial L}{\partial y} \cdot \underbrace{y(1-y)}_{>0} \cdot \underbrace{x_i}_{>0}$$

$$= y(1-y) > 0$$

恒大于0
可正可负 (x_i > 0)

故 $\frac{\partial L}{\partial w_i}$ 必然同号 (在二维空间来说,

即 $\frac{\partial L}{\partial w_i}$ 只能落在 I, II

象限)

<https://blog.csdn.net/edogamachi>

由此可知，所有的 w_i 进行梯度计算后，都是一个符号，那么梯度下降的方向过于单一，并且下降速率会受影响。

ReLU & Leaky ReLU: 为了解决ReLU的dead ReLU现象。这里选择一个数，让负数区域不在饱和和死掉。这里的斜率都是确定的。

8. 向量化的解释:

$$W^{[i]}.shape() = [n_i, n_{i-1}]$$

$$x = [x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(m)}] \quad x^{(i)} \text{ 以列向量形式排列即可}$$

$$W^{[1]}x = \begin{bmatrix} \dots \\ \dots \\ \dots \end{bmatrix} [x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(m)}]$$

$$= [W^{(1)}x^{(1)}, W^{(1)}x^{(2)}, W^{(1)}x^{(3)}, \dots, W^{(1)}x^{(m)}]$$

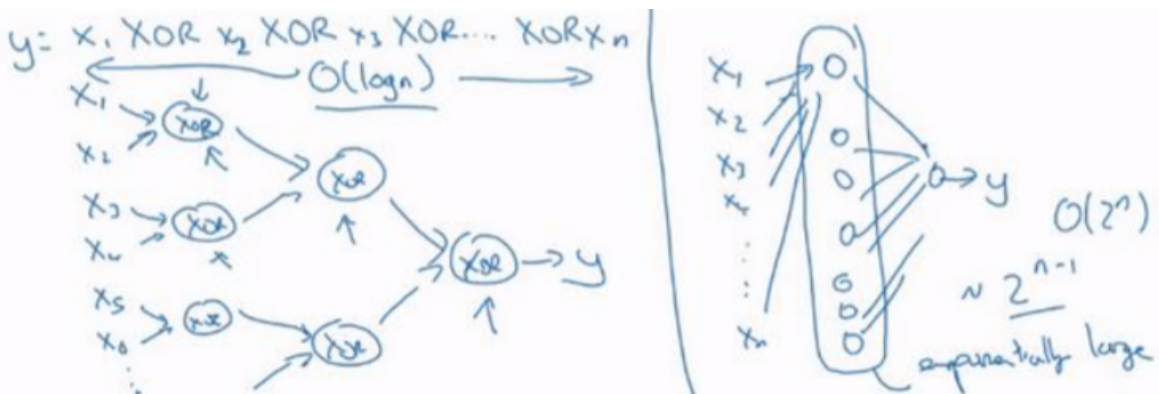
第二课时：模型调优

9. 随机初始化：对于权重 W 一定需要随机初始化，不然隐藏层的神经元都是一样的值
 b 来说，一般初始为0也可以

```
W = np.random.randn(2, 2) * 0.01 # 让一开始的数足够下，从而能在sigmoid上获取良好的学习率
```

10. 深层神经网络的优势在何处？

可以从利用神经网络学习异或来初步理解。



右侧为深度网络，层数 $\log(n)$ 左侧为单层网络，神经元数量为 2^n

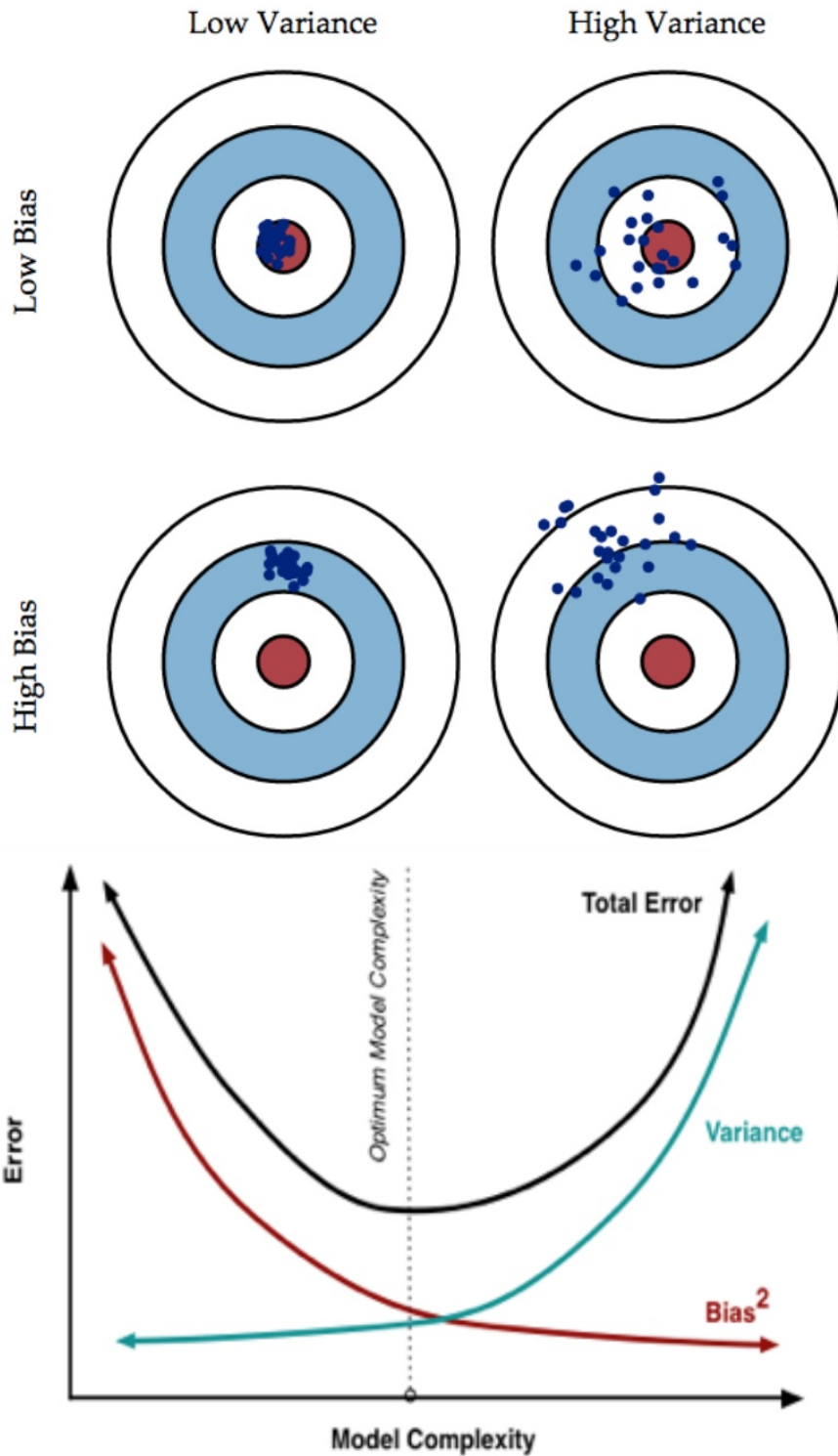
11. 偏差和方差

偏差度量了学习算法的期望预测与真实结果的偏离程度，即刻画了学习算法本身的拟合能力

方差度量了同样大小的训练集的变动所导致的学习性能的变化，即刻画了数据扰动所造成的影响

一般来说，训练集的误差高，那么高偏差；验证集的误差高，那么高方差。

准与确



12. Inverted Dropout(反向随机失活)

当模型使用了 dropout layer，训练的时候只有占比为 p 的隐藏层单元参与训练，那么在预测的时候，如果所有的隐藏层单元都需要参与进来，则得到的结果相比训练时平均要大 $1/p$ ，为了避免这种情况，就需要测试的时候将输出结果乘以 p 使下一层的输入规模保持不变。

而利用inverted dropout, 我们可以在训练的时候直接将dropout后留下的权重扩大 $1/p$ 倍, 这样就可以使结果的scale保持不变, 而在预测的时候也不用做额外的操作了, 更方便一些。

Dropout的一大缺点就是代价函数 J 不再被明确定义

13. 参数&超参数

经验性/推广性/变化性

14. 训练集(train)/验证集(dev)/测试集(test)

比例分配: 由数量而定

15. L2正则化

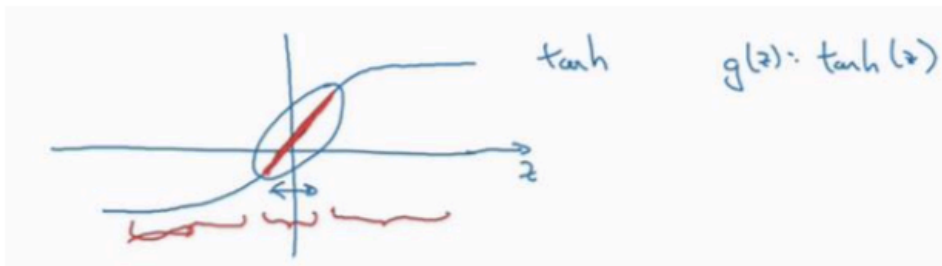
加上一个 $\frac{\lambda}{2m} * \sum_1^L \|W^{[l]}\|^2$ 这个矩阵范数 $\|W^{[l]}\|^2$ 被定义为矩阵中虽有元素的平方求和。

当权值更新时, W 会被乘以 $(1 - \alpha * \frac{\lambda}{m})$ 。也称之为权重衰减, 防止模型复杂度过高, 过拟合。

```
lambda / (2*m) * np.dot(w, w.T)
```

直观理解

特别是, 如果 z 的值最终在这个范围内, 都是相对较小的值, $g(z)$ 大致呈线性, 每层几乎都是线性的, 和线性回归函数一样。



第一节课我们讲过, 如果每层都是线性的, 那么整个网络就是一个线性网络, 即使是一个非常深的深层网络, 因具有线性激活函数的特征, 最终我们只能计算线性函数, 因此, 它不适用于非常复杂的决策, 以及过度拟合数据集的非线性决策边界, 如同我们在幻灯片中看到的过度拟合高方差的情况。

所以不会变得十分复杂

16. 其他的正则化方法

Early stopping

Data augmentation: by rotating the images.

17. 归一化输入

零均值 $\mu = \frac{1}{m} * \sum_{i=1}^m x^{(i)}$

之后, 归一化方差 $\sigma^2 = \frac{1}{m} * \sum_{i=1}^m (x^{(i)})^2$

理解归一化: 使得特征 x 处于统一范围, 从而学习速率加快

18. 权重初始化

随机权重初始化已经不够用了, 当遇到**梯度爆炸**, **梯度消失**的情况。

因此对于一些情况，随机后的值，需要再处理

ReLU: $np.sqrt(\frac{2}{n^{[L-1]}})$

Tanh: $np.sqrt(\frac{1}{n^{[L-1]}})$

Yoshua Bengio $np.sqrt(\frac{2}{n^{[L-1]}+n^{[l]}})$

19. 梯度检查

反向传播算法很难调试得到正确结果，尤其是当实现程序存在很多难于发现的bug时。举例来说，索引的缺位错误（off-by-one error）会导致只有部分层的权重得到训练（for(i=1; i<=m; ++i) 被漏写为 for(i=1; i<m; ++i)），再比如忘记计算偏置项。这些错误会使你得到一个看似十分合理的结果（但实际上比正确代码的结果要差）。因此，仅从计算结果上来看，我们很难发现代码中有什么东西遗漏了。

Check:

$$\frac{\|d\theta_{appro} - d\theta\|_2}{\|d\theta_{appro}\|_2 + \|d\theta\|_2}$$

这里运用到了欧几里得范数 当值 $< 10^{-7}$ 表示反向传播没问题

当值 $< 10^{-3}$ 就要担心是否存在bug了

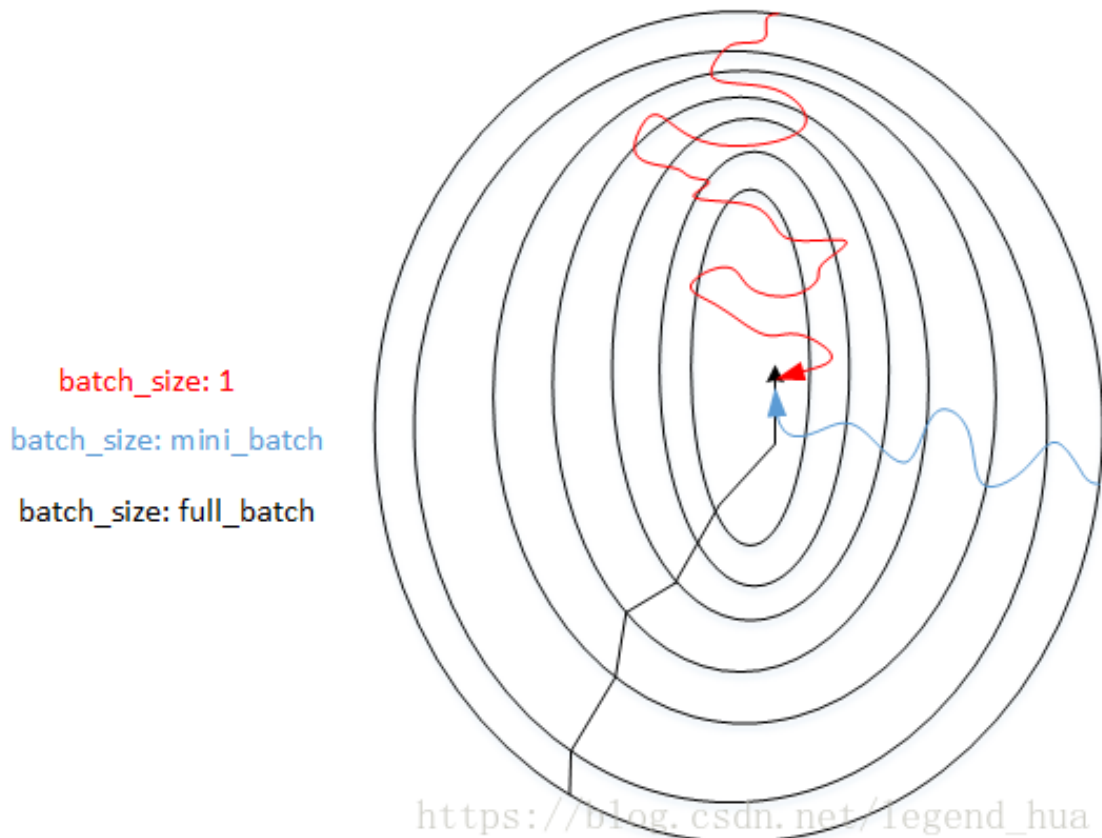
注意点

若用了正则化，包括正则项

关闭Dropout

若 W, b 很小的时候，梯度检查没问题，但变大后会出现问题。所以得训练一段时间后再进行梯度检查

20. SGD, Mini_Batch, Full_Batch



当使用Mini_Batch时，不会收敛，会在最优值处摆动，并且下降过程中有许多噪声

特殊地，当batch_size = 1时，为随机梯度下降SGD

Epoch 遍历一次数据集称之为一个Epoch，但因为有多个mini_batch，做了多个梯度下降。

SGD 噪声小（因为相应的学习率最小，不考虑学习率还是有很多噪声的），但失去了向量化的加速

Full_Batch 单次迭代时间过长。噪声小。

21. 选择对数作为标尺

比如对学习率 α 在**0.001~1**进行选择，不必要等间距选择，因为0.1~0.9就占用了90%的资源，而0.001-0.1同等重要。

可以使用指数作为标尺。 $0.001 = 10^{-3}$; $1 = 10^0$ 利用 `-3 * np.random.rand()` 即可

22. **随机搜索替代格子化的搜索** 在每个点上，每个超参数都有些不同，利于观察超参数对结果的影响。

23. **Batch**归一化和输入归一化的不同在于，你有时候不总希望把 $Z^{[l](i)}$ 变到0~1范围之间

操作是在计算 $z > a$ 的时候（计算激活函数之前）

由于要进行归一化，参数 b 显得不那么重要了可以把参数 b 暂时设置为0

此时 $\tilde{Z}^{[l]} = \gamma^{[l]} z^{[l]} + \beta^{[l]}$ 用两个参数再来确定

24. 测试时的**Batch Norm**：采用指数加权平均估算每一层的 $\tilde{Z}^{[l]}$

25. 梯度下降优化算法

什么时候优化？在计算完梯度，update_parameters时优化。

优化的思想？指数加权平均的思想。如果一个参数 w 在一个数值上下不断波动，这样的方法可以让平均值减缓它的波动，从而达到更快的下降。

优化算法：adam, Momentum算法。

26. Logistic回归的一般形式 softmax

Softmax层实际上在算最后一层激活函数时使用了归一化

方程为 $a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{j=1}^n t_j}$ 其中 $t = e^{z^{[l]}}$

方程式可表达为 $a^{[l]} = g^{[l]}(z^{[l]})$

softmax与**sigmoid**和**ReLU**的不同之处在于其由于要将可能的结果归一化，所以传入的是一个向量，输出的也是一个向量

我理解的是：**softmax**在**激活函数**的位置，用了**回归函数**

hardmax(处理方式类似于one-hot (把一个向量的最大元素映射为1，其余为0))

softmax(更为平和的概率映射方式)

最后有 z 了，只用输出最大值，不用softmax层？

第三课时：结构化机器学习项目

27. Classification of localization

$$y = \begin{bmatrix} pc \\ bx \\ by \\ w \\ h \\ c1 \\ c2 \\ c3 \end{bmatrix}$$

pc = 1 : there is object in images pc = 0: there is no object in images.

ci : the possibility of as a object.

27. sliding window algo



to predict every small box is a car or not, that is apply convnet on each small box.

Drawbacks: high complexity, computational cost.

28. Convolutional implementation on sliding window algorithm

more efficient, but the bounding box can be not very accurate.

我认为就是卷积神经网络，只不过把最后的全连接层，softmax也转化成了卷积层，从而每一个节点，代表一个位置的卷积结果。这样一次CNN计算就可以实现窗口滑动的所有子区域的分类预测。这其实是overfeat算法的思路。看了半天网上的不知道在说些啥。。。

这篇稍微讲的好些 <https://github.com/AlbertHG/Coursera-Deep-Learning-deeplearning.ai/tree/master/04-Convolutional%20Neural%20Networks/week3>

29. 目标检测策略

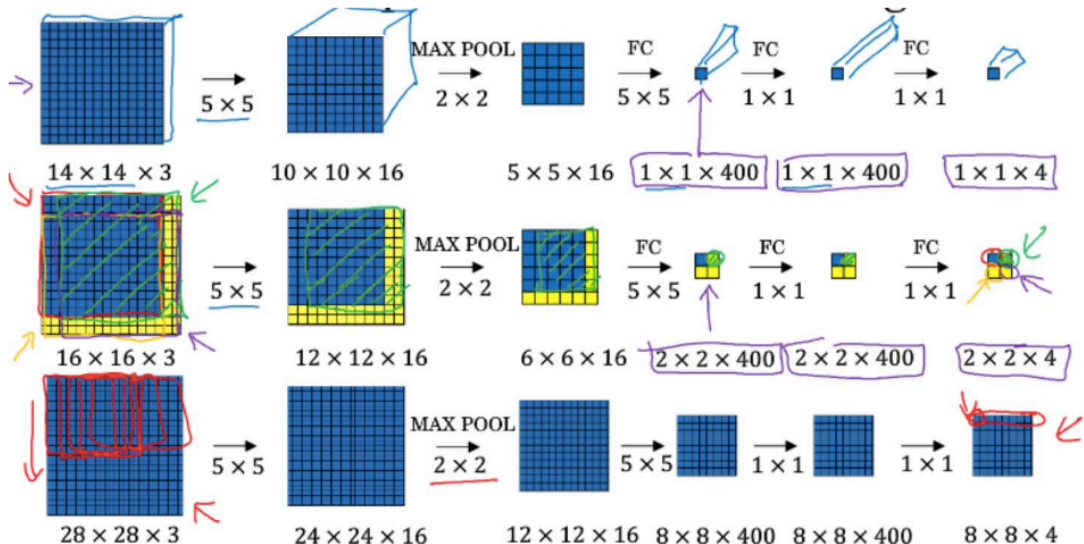
1. **Sliding windows** : Apply each window in convnet, to see if there's an object, and thus get its position
2. **Convolutional implementation of Sliding window** : First, transforming the full connect layer into convolutional layer. Then use convolutional filter to get the result. If lucky, we can get all results in different places in one computing.

如下图中间一行卷积的初始图，我们假设输入的图像是 $16 \times 16 \times 3$ 的，而窗口大小是 $14 \times 14 \times 3$ ，我们要做的是把蓝色区域输入卷积网络，生成0或1分类，接着向右滑动2个元素，形成的区域输入卷积网络，生成0或1分类，然后接着滑动，重复操作。我们在 $16 \times 16 \times 3$ 的图像上卷积了4次，输出了4个标签，我们会发现这4次卷积里很多计算是重复的。

而实际上，直接对这个 $16 \times 16 \times 3$ 的图像进行卷积，如下图中间一行的卷积的整个过程，这个卷积就是在计算我们刚刚提到的很多重复的计算，过程中蓝色的区域就是我们初始的时候用来卷积的第一块区域，到最后它变成了 $2 \times 2 \times 4$ 的块的左上角那一块，我们可以看到最后输出的 2×2 ，刚好就是4个输出，对应我们上面说的输出4个标签。

这两个过程刚好可以对应的上。所以我们不需要把原图分成四个部分，分为用卷积去检测，而是把它们作为一张图片输入给卷积网络进行计算，其中的公有区域可以共享很多计算。

同样的，当图片大小是 $28 \times 28 \times 3$ 的时候，CNN网络得到的输出层为 $8 \times 8 \times 4$ ，共64个窗口结果。



BTW, **Fast RCNN**利用了该策略，借鉴的**OverFeat**的思路

3. **Regional proposal** : 利用图像色块的分器，对窗口进行启发式搜索。降低了滑动窗口的运算复杂度。**RCNN**利用了该策略
4. **Bounding Boxes** : 对于每一个检测目标，找到他们的中心点，依此中心点找到cell，每一个cell产生**b**个**bounding boxes**，转换为向量就是

$$y = \begin{bmatrix} pc1 \\ bx1 \\ by1 \\ bw1 \\ bh1 \\ pc2 \\ bx2 \\ by2 \\ bw2 \\ bh2 \\ c1 \\ c2 \\ c3 \end{bmatrix}$$

根据**loss function**，**BBR**不断修正。

5. **Anchor Boxes** : 对于每一个检测目标, 找到他们的中心点, 依此中心点找到`cell`, 每一个`cell`产生`b`个`anchor boxes`, 对于检测目标, 求与每个对应`cell`的`anchor box`的`IoU`, 找到最大的, 即为对应的`box`。该方法为普通`Bounding Box`的改良版, 因为之前若一个`cell`有多个目标, 便无法检测。

6. 重温了逻辑回归, 上手了week1的编程题

几点值的注意:

1. 逻辑回归的方程式和损失函数
2. `plt`的使用
3. 建立模型的流程: 数据预处理, 参数初始化, 前向传播, 反向传播 (迭代得到答案)

7. 重温了浅层模型的建立, 关键在于几个公式

0. 初始化, `w, b`

```
W1 = np.random.randn(nh, nx)*0.001
b1 = np.zeros((nh, 0))
W2 = np.random.randn(ny, nh)*0.001
b2 = np.zeros((ny, 0))
```

1. 建立前向传播 **Forward propagation**

```
Z1 = np.dot(W1, X) + b1
A1 = np.tanh(Z1)
Z2 = np.dot(W2, A1) + b2
A2 = sigmoid(Z2)
```

2. 反向传播

```
#####Compute
Loss#####
m = Y.shape[1] # number of example
# Compute the cross-entropy cost
logprobs = np.multiply(np.log(A2),Y) + np.multiply((1-Y), (np.log(1-
A2)))
cost = -1/m * np.sum(logprobs)

#####Backward
propagate#####
dZ2 = A2 - Y
dW2 = 1/m * np.dot(dZ2, A1.T)
db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)
dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
dW1 = 1/m * np.dot(dZ1, X.T)
db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)
### END CODE HERE ###
grads = {"dW1": dW1,
         "db1": db1,
         "dW2": dW2,
         "db2": db2}
```

```
#####Update
params#####
W1 -= learning_rate * dW1
b1 -= learning_rate * db1
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
```

3. 预测

```
predictions = np.array( [1 if x >0.5 else 0 for x in
A2.reshape(-1,1)] ).reshape(A2.shape)

print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-
Y,1-predictions.T))/float(Y.size)*100) + '%')
```

4. Draw Graph

```
# Datasets
noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure
= load_extra_datasets()

datasets = {"noisy_circles": noisy_circles,
           "noisy_moons": noisy_moons,
           "blobs": blobs,
           "gaussian_quantiles": gaussian_quantiles}

i = 0
plt.figure(figsize=(8, 16))
for dataset in datasets:
    plt.subplot(4, 2, i+1)
    i += 1
    plt.title(dataset)

    X, Y = datasets[dataset]
    X, Y = X.T, Y.reshape(1, Y.shape[0])

    # make blobs binary
    if dataset == "blobs":
        Y = Y%2

    #### Draw scatter.
    plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);

    # Build a model with a n_h-dimensional hidden layer
    parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000,
print_cost=False)
```

```

##### Plot the decision boundary
plt.subplot(4, 2, i+1)
i += 1
##### Draw boundary.
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title(dataset + 'Classifier')

```

5. 深度网络建立总结：核心在于记忆化存储，反向传播所需要的值，避免重复求导，增大运算量。

反向传播公式

对 $W_{ji}^{[l]}$ 的求导是两次链式法则得到的

$$\frac{\partial E}{\partial w_{ji}^{[l]}} = \frac{\partial E}{\partial o_j^{[l]}} \frac{\partial o_j^{[l]}}{\partial net_j^{[l]}} \frac{\partial net_j^{[l]}}{\partial w_{ji}^{[l]}}$$

对于最右边一项：
$$\frac{\partial net_j^{[l]}}{\partial w_{ji}^{[l]}} = \frac{\partial}{\partial w_{ji}^{[l]}} \left(\sum_{k=0}^{n_{l-1}-1} o_k^{[l-1]} w_{jk}^{[l]} \right) = o_i^{[l-1]}$$

对于中间一项：纯粹是激活函数的偏导数

对于第一项：

若 $o^{[l]}$ 是输出层 := 直接求导就好了

若 $o^{[l]}$ 是中间网络一层：

$$\frac{\partial E}{\partial o_j^{[l]}} = \sum_{k=0}^{n_{l+1}-1} \left(\frac{\partial E}{\partial net_k^{[l+1]}} \frac{\partial net_k^{[l+1]}}{\partial o_j^{[l]}} \right) = \sum_{k=0}^{n_{l+1}-1} \left(\frac{\partial E}{\partial o_k^{[l+1]}} \frac{\partial o_k^{[l+1]}}{\partial net_k^{[l+1]}} \frac{\partial net_k^{[l+1]}}{\partial o_j^{[l]}} \right) = np.dot(w.T, dz)$$

前两项所有结点求 sum 后为 dz ，最后一项列出实际公式，就知道是 w 了

而 w, dz 是后一层的，所以这也是后一层算的

can be shortened as:
$$\frac{\partial E}{\partial w_{ji}} = o_i \delta_j$$
 [注释： δ_j 可以看作 dZ]

但在编程实现中是这样的思路：

$$\frac{\partial E}{\partial o_j^{[l]}}$$
 因为有保存为 dA ，直接用就好了，不用展开（核心）

$$\frac{\partial E}{\partial o_j^{[l]}} \frac{\partial o_j^{[l]}}{\partial net_j^{[l]}}$$
 作为 $dZ = dA * \sigma'(z)$

$$\frac{\partial net_j^{[l]}}{\partial w_{ji}^{[l]}}$$
 为 A_{prev}

反向传播公式理解

首先明确他是来干嘛的，求 dW, db 的。 dW, db 由于优化计算，从后向前计算。

dW 的计算需要用到两次链式法则，生成了其他项，又必须得对其他梯度的求导。

图解

计算 $\frac{\partial E}{\partial w_{ji}}$ 的公式

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial O_j^{(l)}} \cdot \frac{\partial O_j^{(l)}}{\partial net_j^{(l)}} \cdot \frac{\partial net_j^{(l)}}{\partial w_{ji}}$$

① $l-1$ 为输出层. $\frac{\partial E}{\partial O_j^{(l)}} = \frac{\partial E}{\partial O_j^{(l)}} \cdot \sigma_j'$
 $\frac{\partial E}{\partial O_j^{(l)}} = \frac{\partial E}{\partial net_j^{(l)}} \cdot \sigma_j'$
 $\frac{\partial E}{\partial net_j^{(l)}} = \sum_{k=0}^{K-1} \frac{\partial E}{\partial O_k^{(l-1)}} \cdot \frac{\partial O_k^{(l-1)}}{\partial net_j^{(l)}}$
 $= n.p.dot(w.T, dz)$

② $l-1$ 不为 output layer
 $\frac{\partial E}{\partial O_j^{(l)}} \cdot \frac{\partial O_j^{(l)}}{\partial net_j^{(l)}} = dA \cdot \sigma_j'$
 后一层算好的
 前一层为 dA , 和步骤 1 相同.

直接可用损失函数求导.

前一层在这里算

两种情况无区别)

实际编程例子

...

实际编程中思路:

将output层的dAL算出来, 很好算, 不需要递归, 对应上述(最左边一项)的第一种的情况

再算 $dZ = dAL * g'(Z)$ (一次链接法则)

$dW = (dE/dAL * dAL/dZ) * (dZ/dW) = dZ * A_pre.T$

db 同理

$dA_pre = dZ/dA_pre = W$

再算前一层:

$dZ = dA * g'(Z)$

...

计算 $dZ = dA$ (后一层算出来的, 这层的任务是求前一层的 dA) * $g'(Z)$

```
def sigmoid_backward(dA, cache):
```

```
    """
```

```
        Implement the backward propagation for a single SIGMOID unit.
```

```
    Arguments:
```

```
    dA -- post-activation gradient, of any shape
```

```
    cache -- 'Z' where we store for computing backward propagation efficiently
```

```
    Returns:
```

```
    dZ -- Gradient of the cost with respect to Z
```

```
    """
```

```
    Z = cache
```

```
    s = 1/(1+np.exp(-Z))
```

```
    dZ = dA * s * (1-s)
```

```
    assert (dZ.shape == Z.shape)
```

```
    return dZ
```

```
# relu 激活函数
```



```

def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation
    efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct
    object.

    # When z <= 0, you should set dz to 0 as well. 再乘以激活函数
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def linear_backward(dZ, cache):
    """
    为单层实现反向传播的线性部分（第L层）

    参数:
    dZ - 相对于（当前第l层的）线性输出的成本梯度
    cache - 来自当前层前向传播的值的元组（A_prev, W, b）

    返回:
    dA_prev - 相对于激活（前一层l-1）的成本梯度，与A_prev维度相同
    dW - 相对于w（当前层l）的成本梯度，与w的维度相同
    db - 相对于b（当前层l）的成本梯度，与b维度相同
    """

    A_prev, W, b = cache
    m = A_prev.shape[1]
    dW = np.dot(dZ, A_prev.T) / m
    db = np.sum(dZ, axis=1, keepdims=True) / m
    ##### !!!本层在这里计算前一层的网络的梯度值!!! #####
    dA_prev = np.dot(W.T, dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)
    ##### !!!dA_prev是需要进入前一层计算的，dW,db只是作为记录，方便更新!!! #####
    return dA_prev, dW, db

```

```

def linear_activation_backward(dA, cache, activation="relu"):
    """
    实现LINEAR-> ACTIVATION层的后向传播。

    参数:
        dA - 当前层l的激活后的梯度值
        cache - 我们存储的用于有效计算反向传播的值的元组 (值为
linear_cache, activation_cache)
        activation - 要在此层中使用的激活函数名, 字符串类型, ["sigmoid" |
"relu"]
    返回:
        dA_prev - 相对于激活 (前一层l-1) 的成本梯度值, 与A_prev维度相同
        dW - 相对于w (当前层l) 的成本梯度值, 与w的维度相同
        db - 相对于b (当前层l) 的成本梯度值, 与b的维度相同
    """
    linear_cache, activation_cache = cache # cache = A_prev, W, b, Z
    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache) #
A_prev, W, b
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db

def L_model_backward(AL, Y, caches):
    """
    对[LINER-> RELU] * (L-1) -> LINEAR -> SIGMOID组执行反向传播, 就
是多层网络的向后传播

    参数:
        AL - 概率向量, 正向传播的输出 (L_model_forward () )
        Y - 标签向量 (例如: 如果不是猫, 则为0, 如果是猫则为1), 维度为 (1, 数量)
        caches - 包含以下内容的cache列表:
            linear_activation_forward ("relu") 的cache, 不包含输出
层
            linear_activation_forward ("sigmoid") 的cache

    返回:
        grads - 具有梯度值的字典
            grads ["dA"+ str (l) ] = ...
            grads ["dW"+ str (l) ] = ...
            grads ["db"+ str (l) ] = ...
    """
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)

```

```

dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

current_cache = caches[L-1]
grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)]
= linear_activation_backward(dAL, current_cache, "sigmoid")

for l in reversed(range(L-1)):
    current_cache = caches[l]
    dA_prev_temp, dW_temp, db_temp =
linear_activation_backward(grads["dA" + str(l + 2)], current_cache,
"relu")
    grads["dA" + str(l + 1)] = dA_prev_temp
    grads["dW" + str(l + 1)] = dW_temp
    grads["db" + str(l + 1)] = db_temp

return grads

```

8. Regression 和 Classification 的区别

Supervised learning problems are categorized into "**regression**" and "**classification**" problems. In a **regression** problem, we are trying to predict results within a **continuous** output, meaning that we are trying to map input variables to some **continuous** function. In a **classification** problem, we are instead trying to predict results in a **discrete** output. In other words, we are trying to map input variables into **discrete** categories.

AndrewNg

一般来说，分类模型可以将回归模型离散化（logistic regression apply on linear regression），回归模型也可以将分类模型的输出连续化（概率Bayes）

参考：<https://www.zhihu.com/question/21329754>

9. 正交化

采用Andrew Ng 举的一个例子，你想要调整电视机的大小，比如是梯形的电视机，你想要调整上边或者下边的宽度；如果是长方形的电视机，你可能想要调整长度和宽度的比例。你希望调整的时候，不会影响到其他因素，因为这样能最有效的修改。如果会影响到其他的参数，你在更改长度时，电视机的上边变窄了，变成了梯形，这肯定不是我们想要的。

对于机器学习来说，参数的调整也是这样的意思。每个维度互不相关。

10. 单一数字评判标准

有时候我们会觉得一个标准无法衡量我们的模型。所以可能会综合多种指标进行评判。

比如F1分数 $F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$ 这里 P 指的是精度 R 指的是查全率

$$Precision = \frac{tp}{tp+fp}$$

$$Recall = \frac{tp}{tp+fn}$$

$$Accuracy = \frac{tp+fn}{tp+tn+fp+fn}$$

11. 可避免偏差: 贝叶斯最优错误率与训练集错误率的差值

引入了这个之后, 不再对比偏差和方差。评估可避免误差和方差的大小。

贝叶斯最优错误率一般通过人的最高正确率估计

12. 善于进行误差分析: 手动调出不符合要求的数据, 试图分析错误的原因, 是否有共同特征

13. 数据分布不匹配时对于偏差和方差的分析: 从训练集中划分出训练-测试集 (不进行训练) 通过比较验证集和训练-测试集的正确率。可以看出数据分布不匹配是否对模型预测有影响, 以及有多大的影响。

14. 处理数据不匹配的问题: 人工合成数据

15. 迁移学习

训练目标是B, 先训练A数据

主要因为下原因, B数据量小, A数据量大

A与B有着共同的特征。学习A的知识, 有助于模型能更快的学习B的知识

预训练完成之后, 一般只需要训练最后几层的数据即可。

16. 多任务学习 顾名思义: 一个模型同时处理多个任务, 比如同时完成行人, 汽车的检测。

多任务学习在各个子任务用共同特征的时候有意义, 如上所述, 行人和汽车都是在汽车公路上的, 都属于交通着一部分

17. 端到端模型 (end to end model)

以前是人为提取所要检测事物的特征

现在只需要放入 x, y 就好了。只要数据量够大, 那么模型具有自动学习的能力。

端到端指的是输入是原始数据, 输出是最后的结果, 原来输入端不是直接的原始数据, 而是在原始数据中提取的特征, 这一点在图像问题上尤为突出, 因为图像像素数太多, 数据维度高, 会产生维度灾难, 所以原来一个思路是手工提取图像的一些关键特征, 这实际就是一个降维的过程。那么问题来了, 特征怎么提? 特征提取的好坏异常关键, 甚至比学习算法还重要, 举个例子, 对一系列人的数据分类, 分类结果是性别, 如果你提取的特征是头发的颜色, 无论分类算法如何, 分类效果都不会好, 如果你提取的特征是头发的长短, 这个特征就会好很多, 但是还是会有错误, 如果你提取了一个超强特征, 比如染色体的数据, 那你的分类基本就不会错了。这就意味着, 特征需要足够的经验去设计, 这在数据量越来越大的情况下也越来越困难。于是就出现了端到端网络, 特征可以自己去学习, 所以特征提取这一步也就融入到算法当中, 不需要人来干预了。

18. 卷积操作 实际上就是 *element wise* 操作, 对应项相乘再加在一起

最后输出维度为 $(n - f + 1) * (n - f + 1)$

使用卷积器使得训练的参数变少:

1. 参数共享

2. 稀疏连接: (又称之为“局部连接”) 以往的神经元负责整个图像的检测, 神经元数量一多, 那么计算量也就增大了。现在每一个神经元 (卷积器) 只负责一部分图像, 减少了计算量。

(一般卷积器的大小远小于图像)。利用层数的增加 (卷积几次) 来学习更广阔, 更抽象的概念。

有两个缺点:

就是每一次的卷积后, 图像就会缩小

角落边缘的像素容易被忽视掉，因为他们往往只参与一次卷积

为了解决上述问题，我们需要padding

19. **Padding** 分为same, valid，前者保证卷积后大小相同，后者其实就是padding = 0

最后输出维度 $(n + 2p - f + 1) * (n + 2p - f + 1)$

20. **stride** 意为步长

现在每次卷积后的结果为 $\lfloor \frac{n+2p-f}{s} + 1 \rfloor * \lfloor \frac{n+2p-f}{s} + 1 \rfloor$

21. **三维卷积** 对于多通道的图像一般需要应用到三维卷积，并且卷积器的通道和图像通道数相同，一般应用多个卷积器来控制图像卷积后的通道数

22. **Pooling Layer** 计算公式和卷积一样，同样有stride, padding这样的操作

23. **ResNets** 在训练深度网络的时候存在梯度爆炸或者梯度消失的问题，可以通过残差块的跳跃连接来解决

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

插入 $a^{[l]}$ 正好是在线性激活之前，ReLU激活之后。

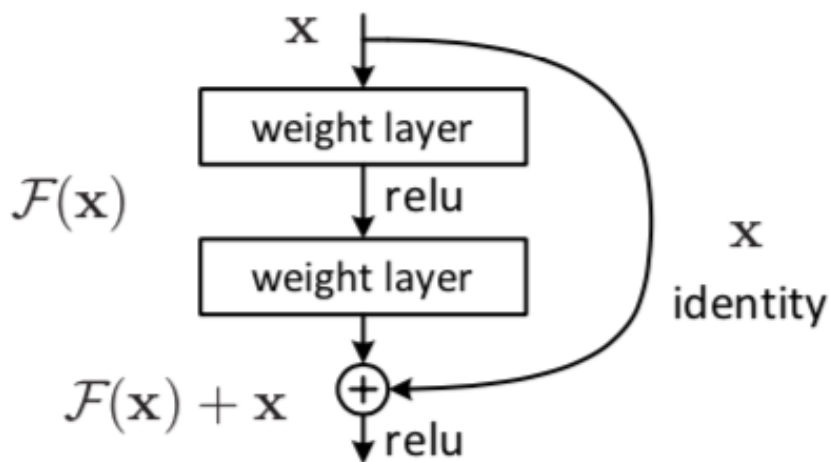


Figure 2. Residual learning: a building block.

在本文中，我们通过引入深度残差学习框架解决了退化问题。我们明确地让这些层拟合残差映射，而不是希望每个堆叠的层直接拟合期望的基础映射。形式上，将期望的基础映射表示为 $H(x)$ ，我们将堆叠的非线性层拟合另一个映射 $F(x) := H(x) - x$

$F(x) := H(x) - x$ 。原始的映射重写为 $F(x) + x$ 。我们假设残差映射比原始的、未参考的映射更容易优化。在极端情况下，如果一个恒等映射是最优的，那么将残差置为零比通过一堆非线性层来拟合恒等映射更容易。

原论文

解读：堆叠的非线性层指网络搭建，当网络到很深的时候，可能学不会什么东西，此时我们把他标记为 $F(x)$

，学习到的结果也是 $x_0 = x_1$ 一个恒等映射 $H(x)$ ，此时有 $H(x) = F(x)$ ，当我们加入shortcut单元，重写之前的恒等映射为 $F(x) + x$ 。虽然一样是恒等映射，但是由于此次直接将残差块置0比通过堆叠非线性层拟合更容易

AndrewNg的解释是他们学习恒等函数十分容易，并且很有可能在隐藏层学习到有用的东西，所以不会比之前的网络性能差

为什么 **ResNets** 可以解决梯度爆炸，梯度消失的问题？

反向传播推导

$$a^{[l]} = g(z^{[l]} + a^{[l-1]}) = g(w^{[l]} a^{[l-1]} + b^{[l]} + a^{[l-1]}) = g((w^{[l]} + 1)a^{[l-1]} + b^{[l]})$$

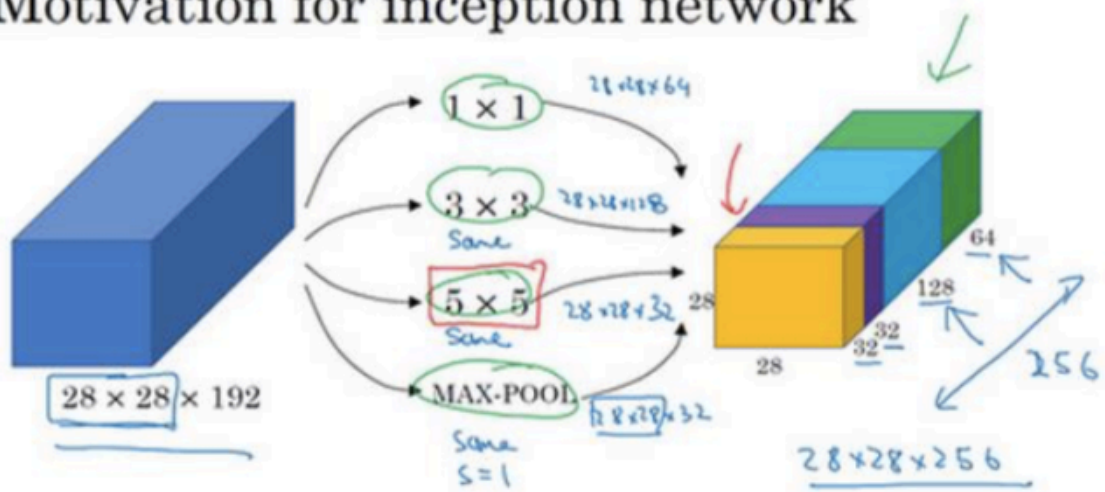
$$\frac{\partial J}{\partial z^{[l-1]}} = \frac{\partial J}{\partial z^{[l]}} \frac{\partial z^{[l]}}{\partial z^{[l-1]}}$$

$$\frac{\partial z^{[l]}}{\partial z^{[l-1]}} = (w^{[l-1]} + 1)\sigma'(z^{[l-1]}) \quad \text{由此避免了梯度消失}$$

48. **1 * 1 convolution** 可以降低通道数，**GoogleNet**应用了这种思想。

49. **GoogleNet: Inception** 自动学习卷积器的大小，是否添加池化层

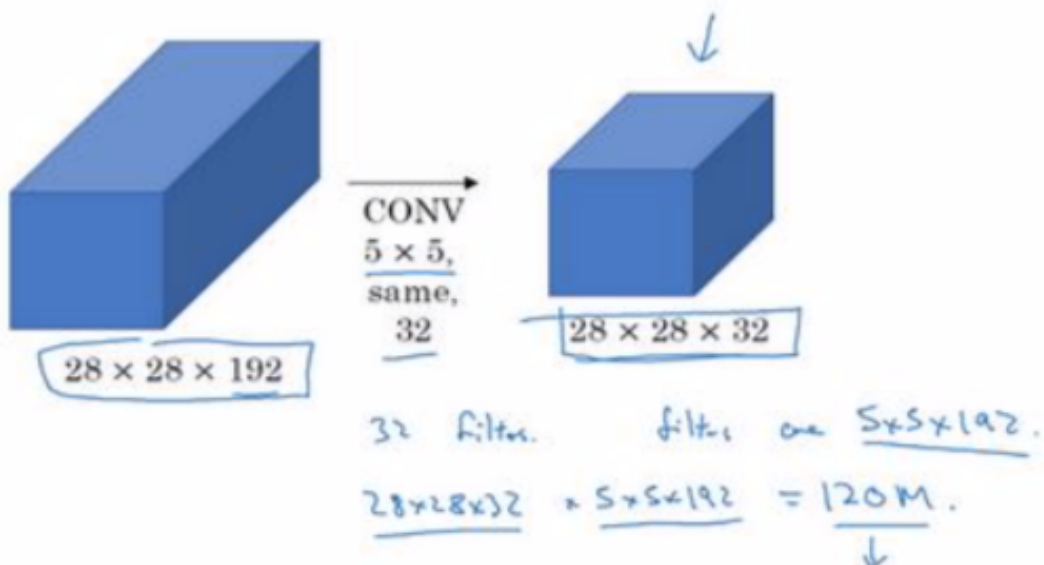
Motivation for inception network



让网络自己去学习采用怎样的过滤器组合。

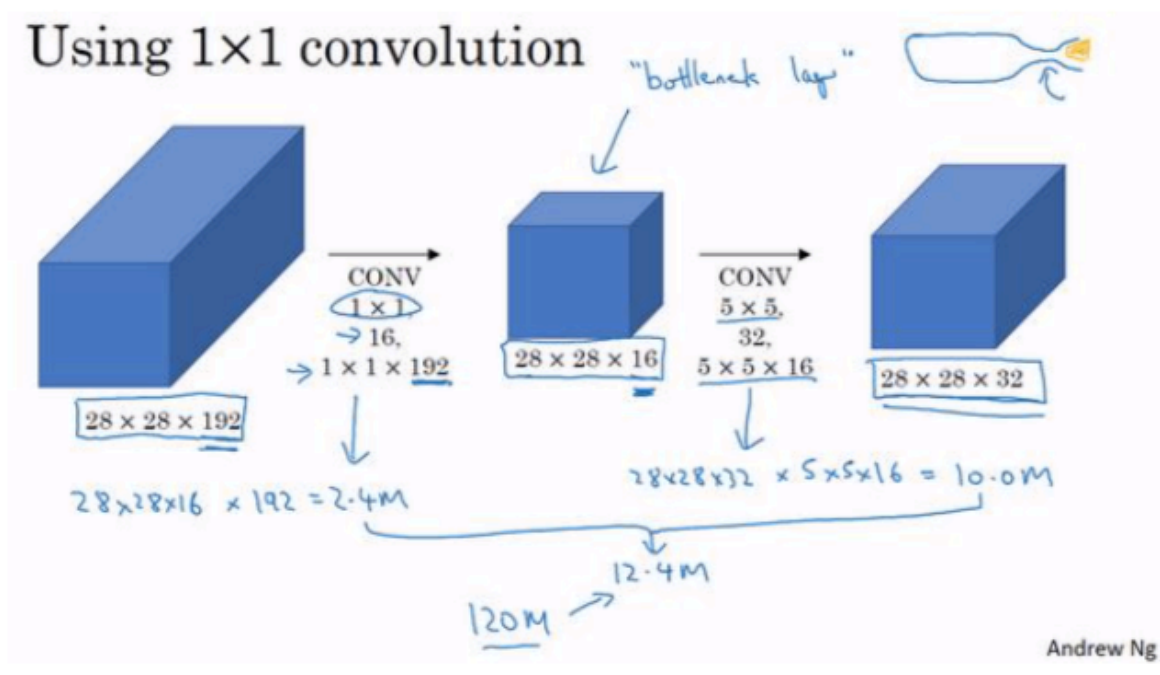
缺点：计算成本过高。 $cost = (28 * 28 * 32) * ((5 * 5) * 192) = 120M$

The problem of computational cost

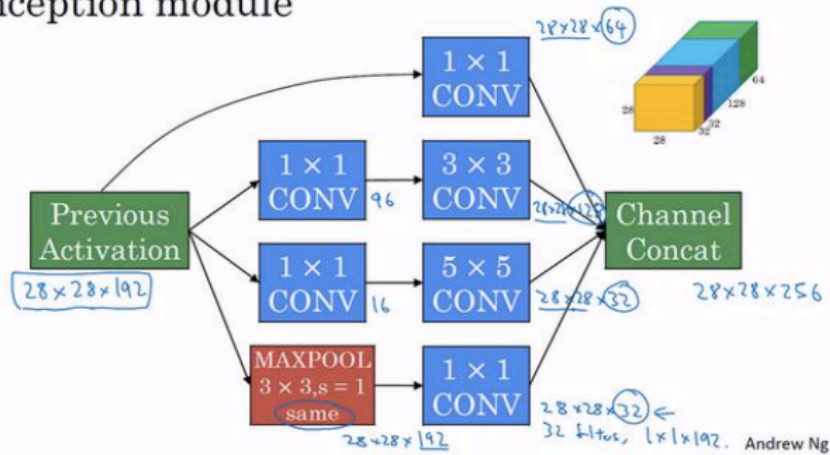


解决方法：利用大小为1的卷积器，引入**bottleneck**层

$$cost = 28 * 28 * 16 * 192 + 28 * 28 * 32 * 5 * 5 * 16 = 12.4M$$

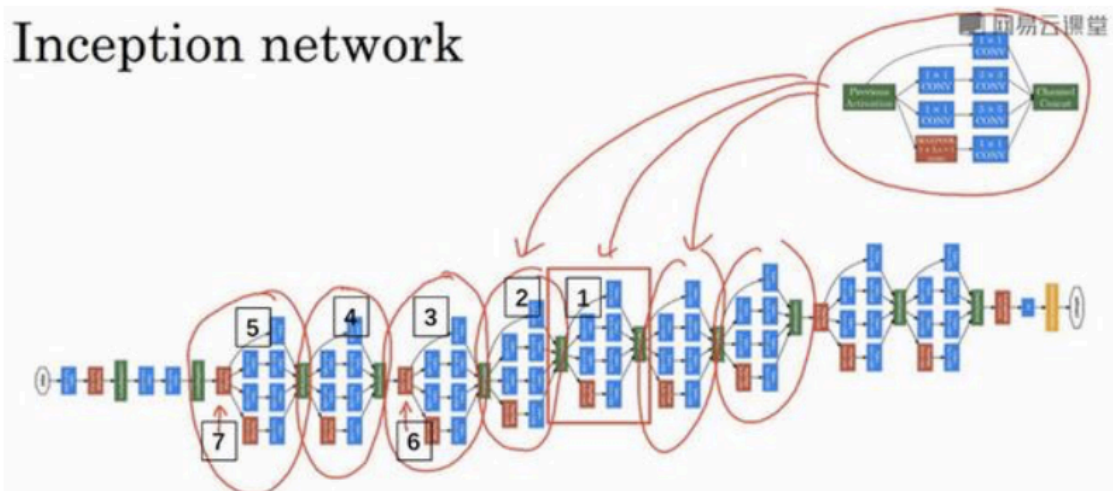


Inception module



最后，将这些方块全都连接起来。在这过程中，把得到的各个层的通道都加起来，最后得到一个 $28 \times 28 \times 256$ 的输出。通道连接实际就是之前视频中看到过的，把所有方块连接在一起的操作。这就是一个 **Inception** 模块，而 **Inception** 网络所做的就是将这些模块都组合到一起。

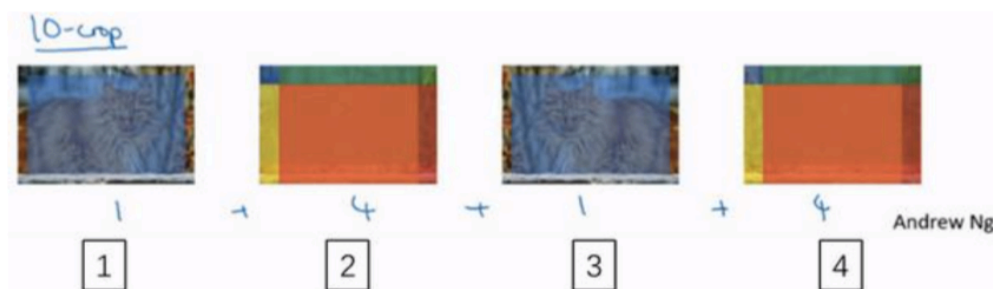
Inception network



50. **迁移学习** 采用别人预训练好的权重，若自己的训练集较少，那么，训练的层数也较少，可能只改变别人的softmax层。其他的层选择**freeze = 1**或者**trainableParameter = 0**这样的参数。若训练集大，那么相应的，得继续训练更多的层，冻结更少的层。
51. **数据增强** 利用**cpu线程**来实现诸如**颜色变换**，**10-crop**这样的变换

10-crop

举个例子，让我们看看猫的图片，然后把它复制四遍，包括它的两个镜像版本。有一种叫作**10-crop**的技术（**crop**理解为裁剪的意思），它基本上说，假设你取这个中心区域，裁剪，然后通过你的分类器去运行它，然后取左上角区域，运行你的分类器，右上角用绿色表示，左下方用黄色表示，右下方用橙色表示，通过你的分类器来运行它，然后对镜像图像做同样的事情对吧？所以取中心的**crop**，然后取四个角落的**crop**。



这是这里（编号1）和这里（编号3）就是中心**crop**，这里（编号2）和这里（编号4）就是四个角落的**crop**。如果把这些加起来，就会有10种不同的图像的**crop**，因此命名为**10-crop**。所以你要做的就是，通过你的分类器来运行这十张图片，然后对结果进行平均。如果你有足够的计算预算，你可以这么做，也许他们需要10个**crop**，你可以使用更多，这可能会让你在生产系统中获得更好的性能。如果是生产的话，我的意思还是实际部署用户的系统。但这是另一种技术，它在基准测试上的应用，要比实际生产系统中好得多。

52. One shot learning

对于人脸识别来说，输出的不应该是图像成为某个人的概率，因为，如果有新员工加入公司，那么输出就会加1，这时只能得重新训练神经网络，这不是一个行之有效的方法。

应该让深度网络去学习的是，**相似度**

我们的思想就是如果两张图片是同一个人，那么编码的距离就得缩小，如果不是一个人，那么编码的距离就得增大。

53. Triplet Loss

三元组损失 即，三张图片为一组，分别标记为**anchor, positive, negative**

损失函数定义为：

$$L(A, P, N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$$

如果前者 ≤ 0 ，那么损失函数为0，如果 > 0 那么损失函数为前者。

54. 将人脸识别转换为二分类问题

Loss function

$$\hat{y} = \sigma\left(\sum_{k=1}^{nx} w_k * |f(x_k^{(i)}) - f(x_k^{(j)})| + b\right)$$

$f(x)$ 为对图片 x 的编码

55. 神经风格转换

给定原始图像 C ，风格图像 S ，令生成成为 G 。可以定义代价函数为

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(C, S)$$

Content cost function

选择隐藏层 l 来计算代价函数，若 l 为浅层，这会是生成的图片像素十分接近于内容图像；若 l 为深层，他能学习到图像中一些更为抽象的特征。定义如下

$$J_{content} = \frac{1}{2} * \|a^{[l][C]} - a^{[l][G]}\|^2$$

Style cost function

想要将风格图像迁移到原始图像上，和**Content translation**不同，对风格图像的捕捉，主要捕捉的是图像块之间的相互性。定义如下

$$\text{原图像互关性} : G_{kk'}^{[l][S]} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{i,j,k}^{[l][S]} a_{i,j,k'}^{[l][S]}$$

$$\text{生成图像互关性} : G_{kk'}^{[l][G]} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{i,j,k}^{[l][G]} a_{i,j,k'}^{[l][G]}$$

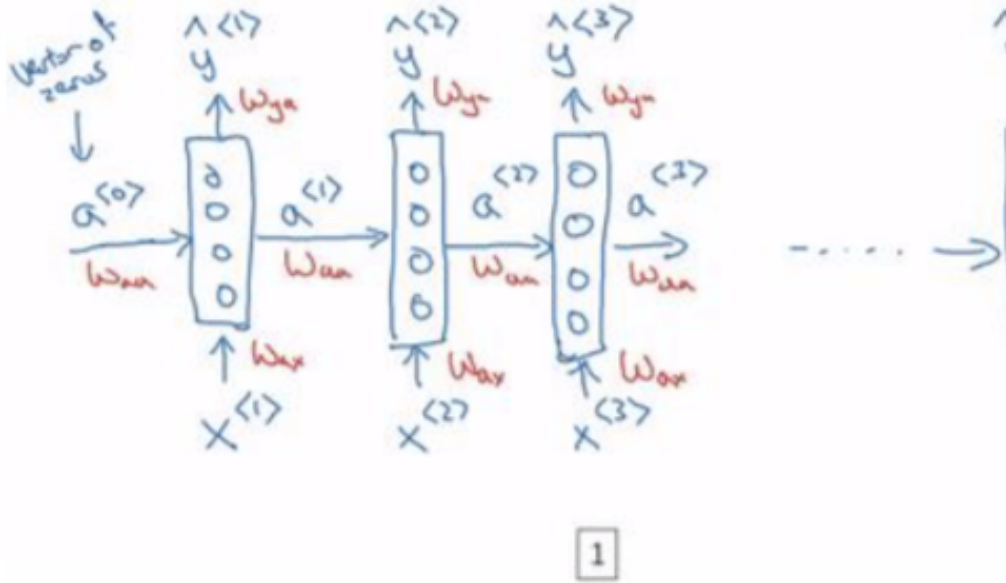
$$\text{所以} : J_{Style}^{[l]} = \frac{1}{2n_H^{[l]}n_W^{[l]}n_C^{[l]}} \sum_k \sum_{k'} (G_{kk'}^{[l][S]} - G_{kk'}^{[l][G]})$$

第五课 序列模型

56. **RNN的基本思路**：初始化零向量 $a^{<0>}$ ，按顺序对序列 x 有序输入，每一个输入都得到之前的激活值

缺点：只能接受前面的输入。前面的单元无法收到后面的激活信息。

Recurrent Neural Networks



57. RNN的前向, 反向传播

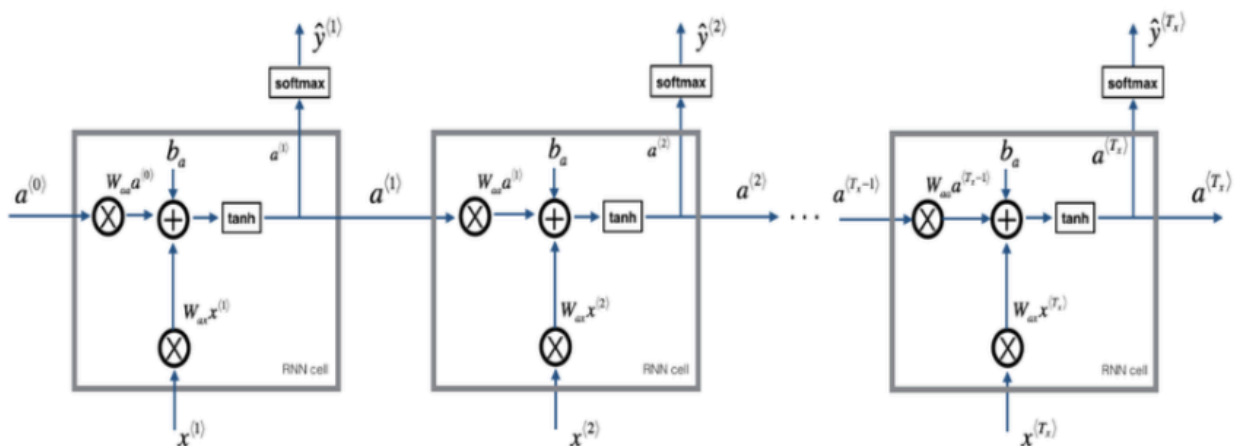
前向传播:

$$a^{<t>} = g(w_a \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} + b_a)$$

$$w_a = [w_{aa}, w_{ax}]$$

$$\hat{y}^{<t>} = g_2(w_y a^{<t>} + b_y)$$

RNN 前向传播示意图:



反向传播

Loss function

$$L(\hat{y}, y) = \sum_{t=1}^{Tx} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -(y^{<t>} \log \hat{y}^{<t>} + (1 - y^{<t>})(1 - \log \hat{y}^{<t>}))$$

公式

$a^{<t>}$ 在本层和下一层都有出现，所以求导两次，并将结果相加
求共享的参数如 W, U, b 都必须得知道 他们对隐层值的求导。

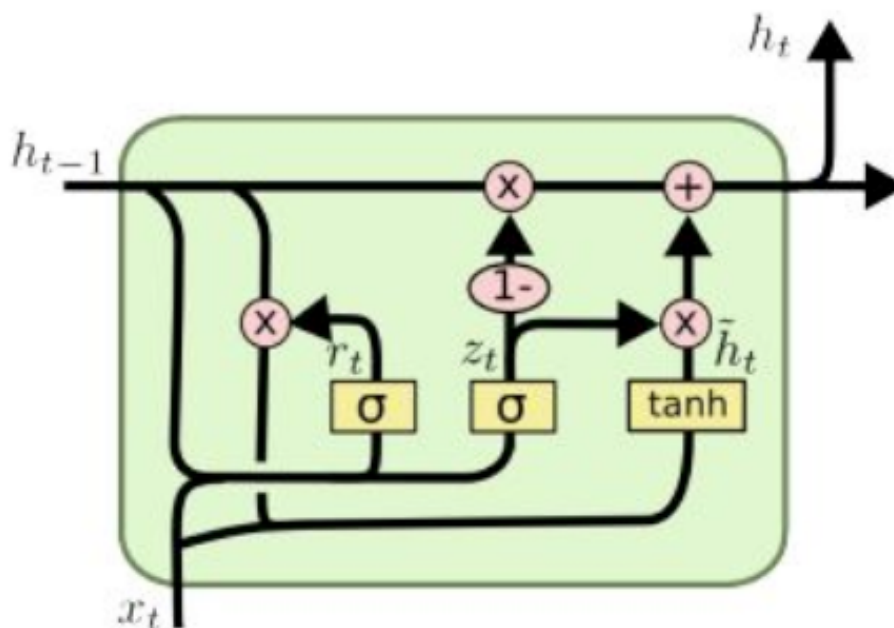
Reference:

<http://www.cnblogs.com/pinard/p/6509630.html>

Deep Learning p234

58. GRU

先看LSTM



$$z_t = \sigma(W_z[h_{t-1}, x_t])$$

$$r_t = \sigma(W_r[h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh(W[r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t)h_{t-1} + z_t\tilde{h}_t \quad z_t \text{ 控制遗忘那些信息}$$

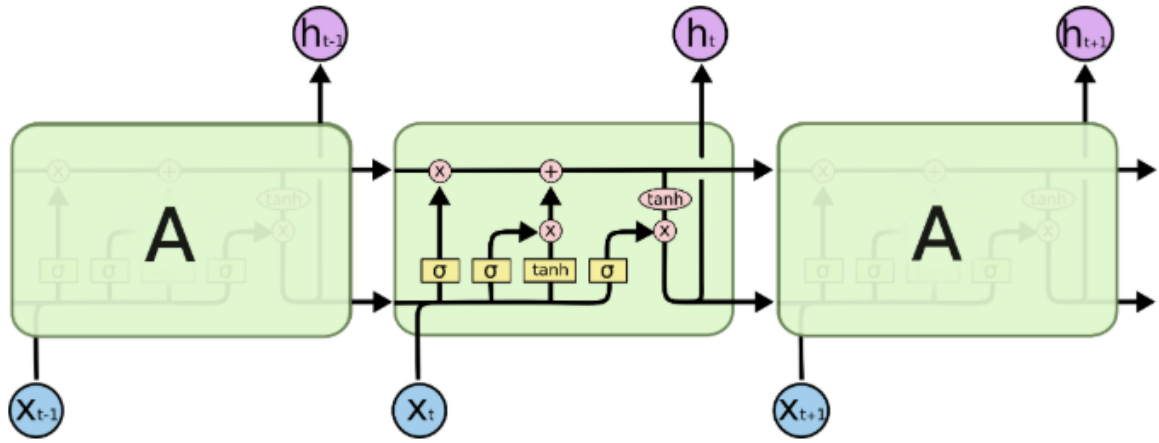
GRU很聪明的一点就在于，我们使用了同一个门控 z 就同时可以进行遗忘和选择记忆（LSTM则要使用遗忘门和信息选择门）。合并了 **hidden state** 和 **cell state**。

References

<https://zhuanlan.zhihu.com/p/32481747>

<https://zhuanlan.zhihu.com/p/34203833>

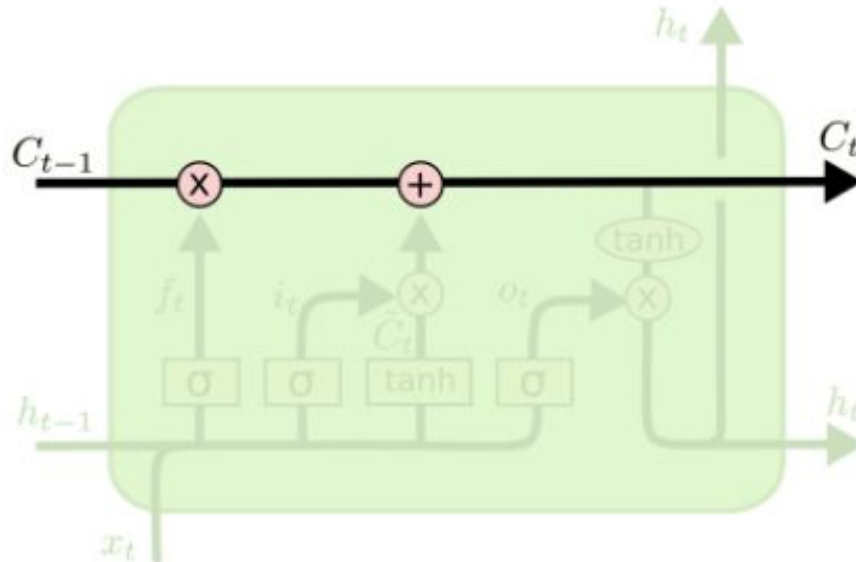
59. LSTM



The repeating module in an LSTM contains four interacting layers.

目标：解决梯度下降，梯度爆炸的问题

核心思想： h_t 还是作为输出，不过增加一个Cell state c_t 作为传送带一般，传递信息，信息流过这条线而改变是非常不容易的。同时也可以通过门控控制的思想，增减其中的信息。



结构：一个信息选择门，一个遗忘门，一个更新门。

先说几个门：

普通rnn的激活值在这里面变成cell state的一个副本： $\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

遗忘门： $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

选择门： $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

更新门： $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$

几个门的应用： f_t 用来对上一步传进来的信息做选择性忘记， i_t 对本轮产生的激活值选择性记忆。加和更新本轮的激活值

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

忘记上层的，记住这层的。

更新层：对得到的 c_t 更新 $h_t = o_t * \tanh(c_t)$

更新的是 c_t 的 filter version (放缩一下)

思考一下：

c_t 就是不过更新门的 h_t ，这样穿到下一层运算时，就能防止梯度消失了。

里面变量很多再整理下

x_t t 时间的输入

c_t t 时间的 *cell state* 贯穿其中

h_t t 时间的输出，也作为下一层的输入

至于为什么：还有待思考。

References

<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

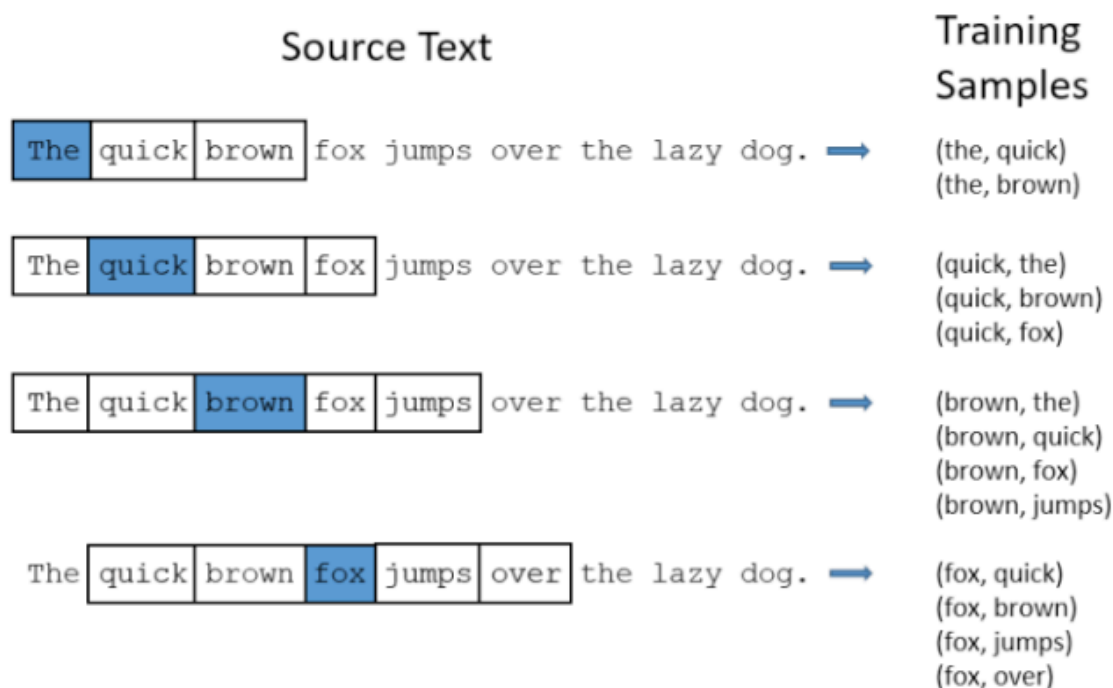
<https://zhuanlan.zhihu.com/p/34203833>

<https://zhuanlan.zhihu.com/p/32085405>

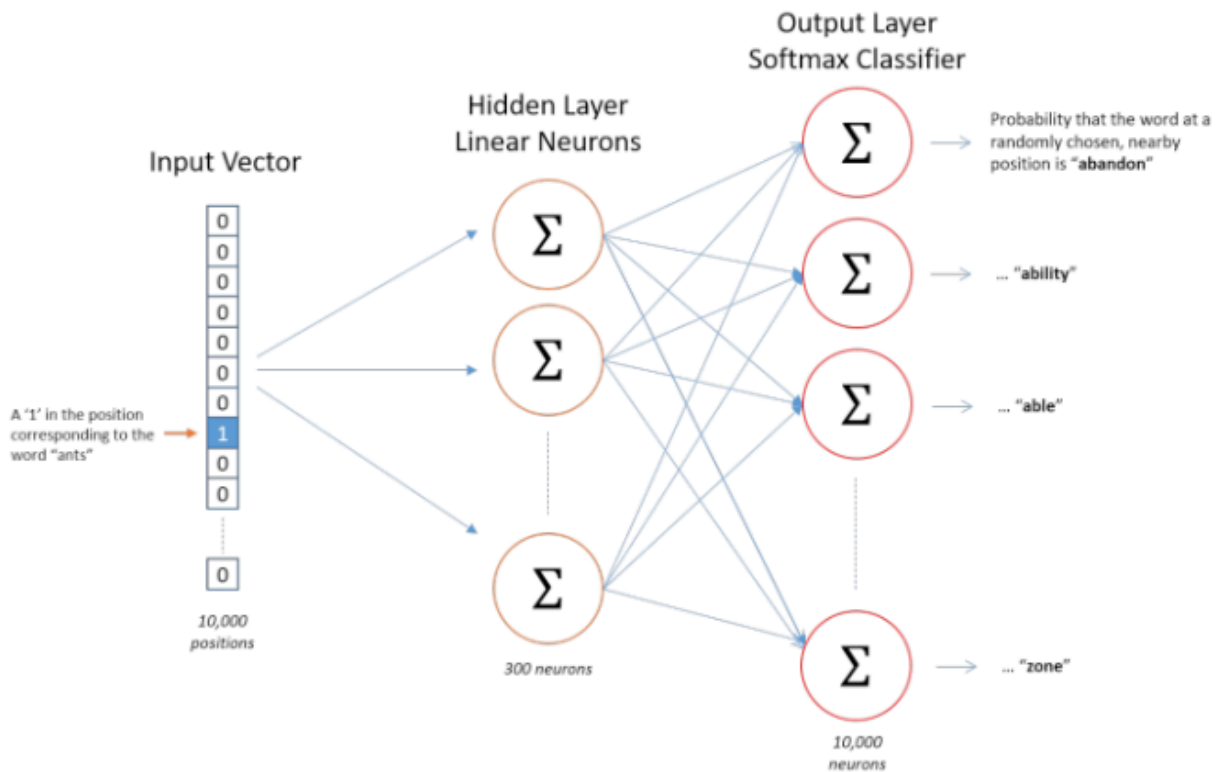
60. Word2Vec

思想： 将语料库里面找出许多单词对，一个作为输入，一个作为输出，训练得到合适的权重，这样，语义相似的单词，就按着比较近。另外，我们要的不是输出层，而是中间的隐层的参数。隐层称之为嵌入层，训练好后可以进行迁移学习

引入了 *skip-window*, *num-skips* 两个概念。



模型细节： 输入层时one-hot编码，假设为10000维，隐层大小为300维。我们想要让单词通过隐层的权重，降低维数。 $w^T x = h; w = [10000, 300]; x = [1, 10000]$



输入层到隐藏层具体选中细节：

$$[0 \ 0 \ 0 \ 1 \ 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \ 12 \ 19]$$

输出层的softmax：目的是用于训练隐层参数

Reference:

<https://yifdu.github.io/2018/12/05/Embedding%E5%B1%82/>

高效的训练：有负采样，对高频词抽样减少的方法，这里只介绍负采样

回忆一下，输入是10000维，输出是10000维，都是one hot，如果orange作为输入，希望输出为juice。即输出10000维向量中，只有juice那是1，其余是0，这是理想的训练输出结果。

负采样就是，选取一个正样本，几个负样本，做反向传播，其余不进行反向传播。大大减少的运算量。

如何选择负样本呢？基本上是随机选择，单词出现频率越高（语料库中），被抽到概率就越大。

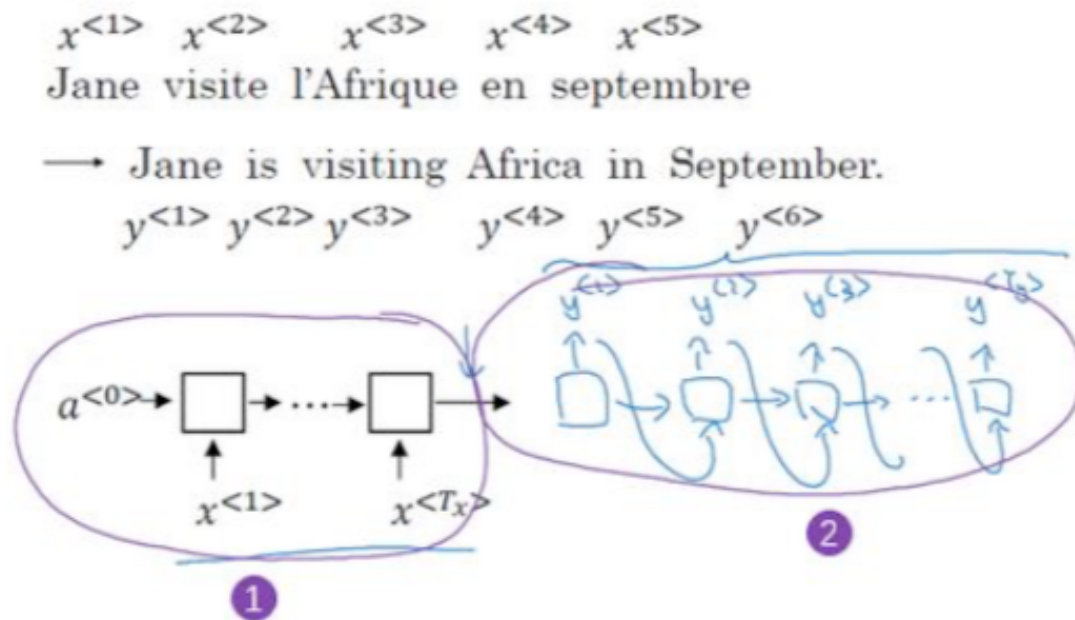
61. 词嵌入除偏

1. 中和与性别无关的词
2. 均衡与性别有关的词

62. Seq2seq and Image_seq

是一个**Encoder-Decoder**的网络。Encoder 中将一个可变长度的信号序列变为固定长度的向量表达，Decoder 将这个固定长度的向量变成可变长度的目标的信号序列（当输出为EOF时，截止）。Encoder将最后的hidden state输出，已包含所有之前的信息。其中的结点可以是**RNN, GRU, LSTM**。Image_seq就是把encoder转换成一些卷积层，全联接层即可。得到编码的图像向量。一般而言，文本处理和语音识别的Encoder部分通常采用RNN模型，图像处理的Encoder一般采用CNN模型。

Sequence to sequence model



63. 集束搜索：启发式的搜索类似于**IDA**，分支限界法。

BLEU：将单词，词组出现的次数加入打分系统中。

64. 注意力机制：

传统的编码器-解码器架构：

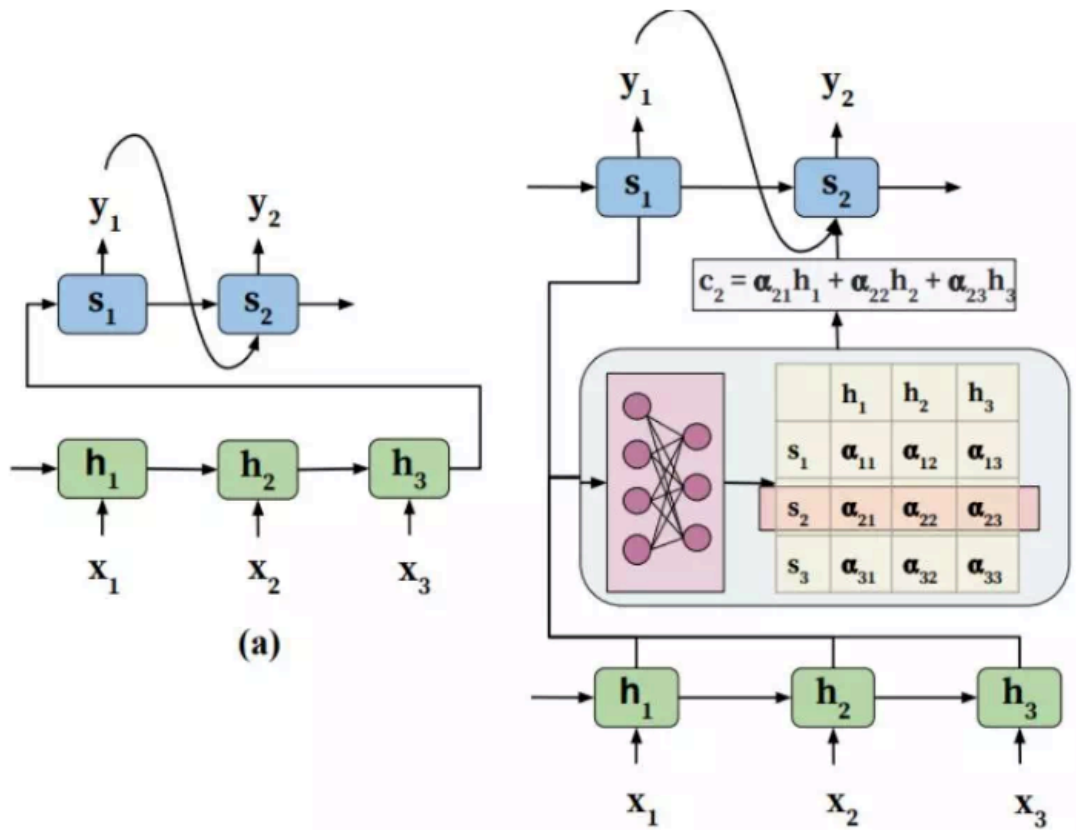


图2：编码器-解码器架构：(a) 为传统结构，(b) 为带注意力机制的结构。

可以看到传统的**Encoder**只向译码器传输一个向量，称之为中间语义向量 c

Encoder对 X 的非线性变换数学表达如下。最后的 c 包含了所有之前的信息，看起来还是很合理的。

$$c = G(x_1, x_2, x_3, \dots)$$

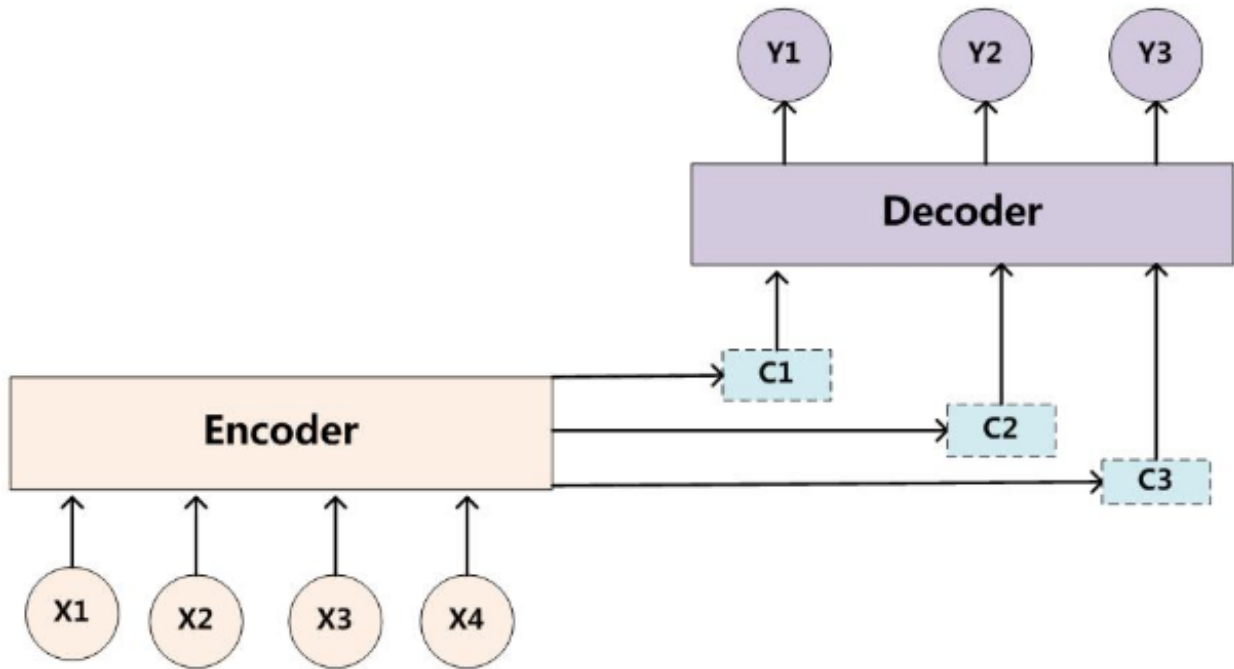
Decoder 根据语义 c 和生成的预测单词，来预测下一个单词，数学表达式如下：

$$y_i = f(c, y_1, y_2, y_3, \dots, y_{i-1})$$

然而缺点就是当语句过长，仅通过中间语义向量来表示，单词自身信息已经消失，会丢失掉更多细节。改进措施就是应用注意力机制。

AM

简略图如下：



$$y_i = f(c_i, y_0, y_1, y_2, y_{i-1})$$

那么语义向量怎么计算呢?

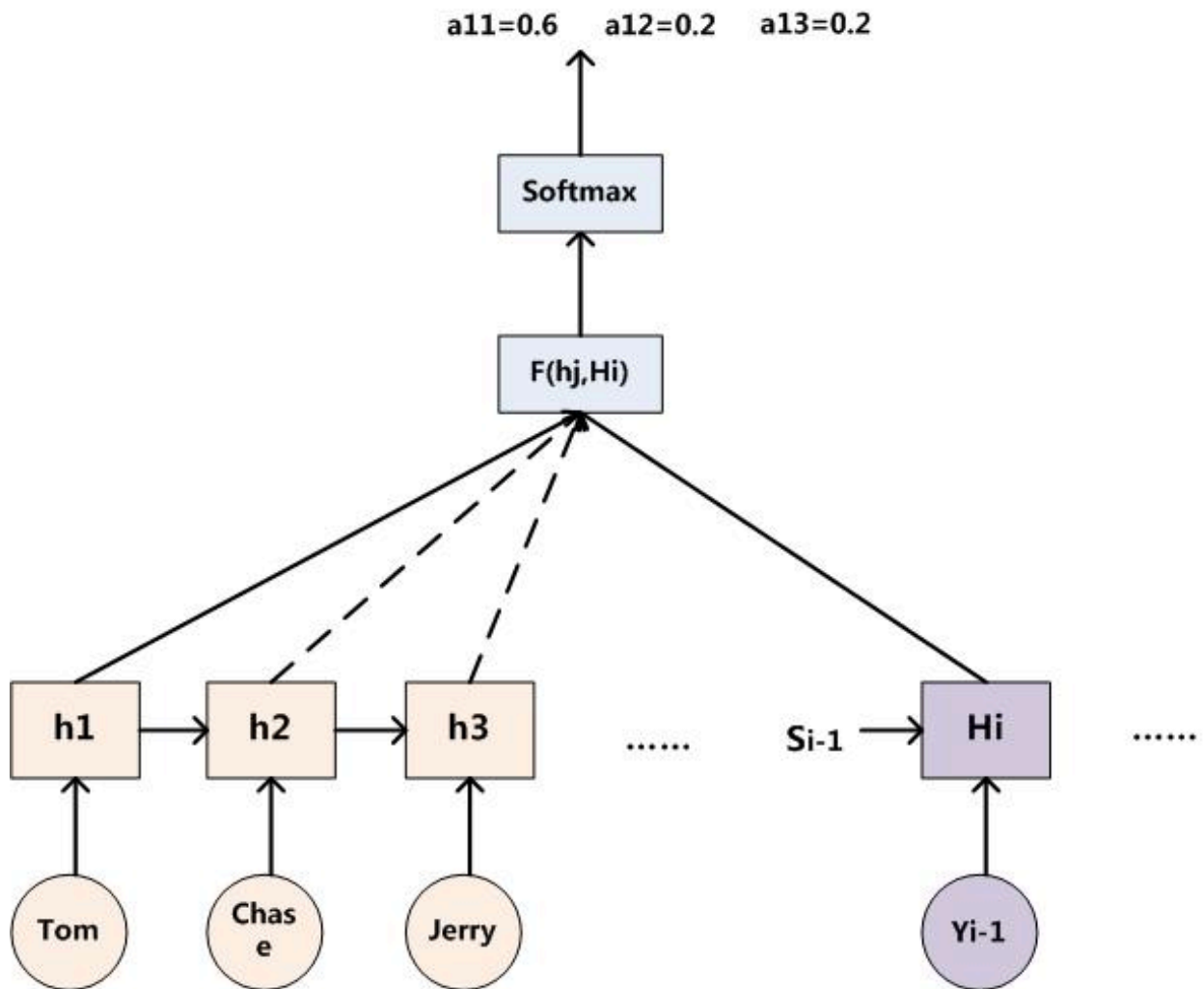
$$c_i = g(a_{i0}h_0, a_{i1}h_1, a_{i2}h_2, \dots, a_{ij}h_j)$$

g 一般是求和函数

所以 c_i 可以表示为：
$$c_i = \sum_{j=0}^{Tx} a_{ij}h_j \quad h_j = h(x_j)$$

注：一般 $h(x_j)$ 就是 *encoder* 的隐藏状态值

现在要求的就是分配概率的计算即 a_{ij}



该网络选取参数为 h_i , H_j , $F(h_i, H_j)$ 实际上是一个对齐函数，代表其对齐可能性。经 **Softmax** 层输出后，即可得到分配概率。

References:

<https://plmsmile.github.io/2017/10/10/attention-model/>

<https://zhuanlan.zhihu.com/p/37601161>

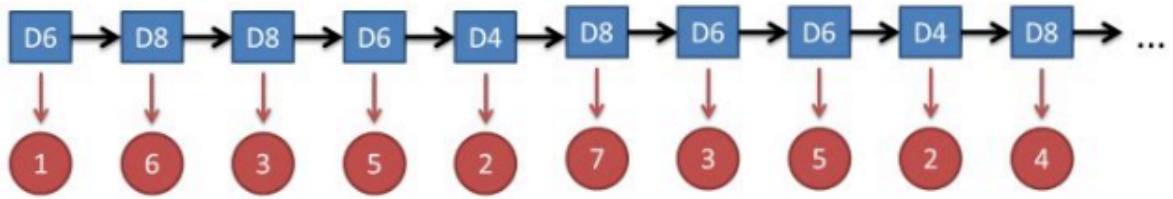
<https://www.jiqizhixin.com/articles/2019-04-10-10>

65. 语音识别

HMM(隐马尔可夫模型)

先引入几个概念：可观测状态（可见状态），预测状态（隐状态），转换概率（也称输出概率，为一个隐状态到一个可见状态的的概率）。可见状态链（可观测到的状态的序列），隐状态链（隐状态的序列）。

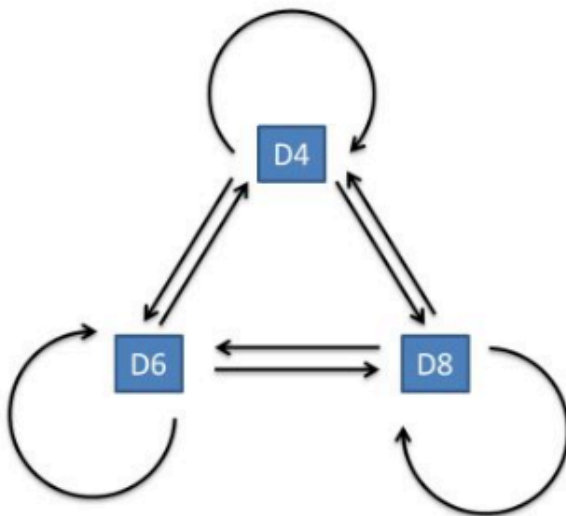
隐马尔可夫模型示意图



图例说明:



隐含状态转换关系示意图



在语音识别中，利用到了HMM去完成解码操作。

知道隐状态的个数，知道隐状态到可见状态的输出概率，知道可见状态链，求：隐状态链

在语音识别中，对应的就是：音素，部分因素，单词，代表隐状态，可观测的声学信息，如频率等为可见

状态。那么就要用到可见状态链去预测隐状态链。

Connectionist Temporal Classification (CTC)

HMM为语音模型，直接转换成文本效果不好，而CTC是一个End-To-End的结构，即，语音特征到文字串只有一个神经网络模型。

CTC的思想就是，允许空白符的输出，从而解决原本RNN模型中输入序列长，输出序列短的问题。

References

<https://www.zhihu.com/question/47642307>

66. 触发字检测

比如一个 *rnn* 模型，输入音频的特征向量后，会得到一些输出值 y_t ，我们在训练集中把触发字说完之后，将 y_t 标记为1，否则为0。这在训练上效果很好，但有一点就是训练集十分不平衡，0比1的数目多太多了。

一个简单粗暴的优化方法就是：比起在只在一个时间步内输出多个1，其实可以在输出变回0之前，多次输出1，这确实又些简单粗暴，不过确实很有效果。