

Machine Learning

Introduction

Machine Learning

Machine Learning的本质就是寻找一个function

- 首先要做的是明确你要找什么样的function，大致上分为以下三类：
 - Regression——让机器输出一个数值，如预测PM2.5
 - Classification——让机器做选择题
 - 二元选择题——binary classification，如用RNN做文本语义的分析，是正面还是负面
 - 多元选择题——multi-class classification，如用CNN做图片的多元分类
 - Generation——让机器去创造、生成
 - 如用seq2seq做机器翻译
 - 如用GAN做二次元人物的生成
- 其次是要告诉机器你想要找什么样的function，分为以下三种方式：
 - Supervised Learning：用labeled data明确地告诉机器你想要的理想的正确的输出是什么
 - Reinforcement Learning：不需要明确告诉机器正确的输出是什么，而只是告诉机器它做的好还是不好，引导它自动往正确的方向学习
 - Unsupervised Learning：给机器一堆没有标注的数据
- 接下来就是机器如何去找出你想要的function
当机器知道要找什么样的function之后，就要决定怎么去找这个function，也就是使用loss去衡量一个function的好坏
 - 第一步，给定function寻找的范围
 - 比如Linear Function、Network Architecture都属于指定function的范围
两个经典的Network Architecture就是RNN和CNN
 - 第二步，确定function寻找的方法
 - 主要的方法就是gradient descent以及它的扩展
可以手写实现，也可以用现成的Deep Learning Framework，如PyTorch来实现

前沿研究

- Explainable AI
举例来说，对猫的图像识别，Explainable AI要做的就是让机器告诉我们为什么它觉得这张图片里的东西是猫
- Adversarial Attack
现在的图像识别系统已经相当的完善，甚至可以在有诸多噪声的情况下也能成功识别，而Adversarial Attack要做的事情是专门针对机器设计噪声，刻意制造出那些对人眼影响不大，却能够对机器进行全面干扰使之崩溃的噪声图像
- Network Compression

你可能有一个识别准确率非常高的model，但是它庞大到无法放到手机、平板里面，而Network Compression要做的事情是压缩这个硕大无比的network，使之能够成功部署在手机甚至更小的平台上

- Anomaly Detection

如果你训练了一个识别动物的系统，但是用户放了一张动漫人物的图片进来，该系统还是会把这张图片识别成某种动物，因此Anomaly Detection要做的事情是，让机器知道自己无法识别这张图片，也就是能不能让机器知道“我不知道”

- Transfer Learning (即Domain Adversarial Learning)

学习的过程中，训练资料和测试资料的分布往往是相同的，因此能够得到比较高的准确率，比如黑白的手写数字识别。但是在实际场景的应用中，用户给你的测试资料往往和你用来训练的资料很不一样，比如一张彩色背景分布的数字图，此时原先的系统的准确率就会大幅下降。

而Transfer Learning要做的事情是，在训练资料和测试资料很不一样的情况下，让机器也能学到东西

- Meta Learning

Meta Learning的思想就是让机器学习该如何学习，也就是Learn to learn。

传统的机器学习方法是人所设计的，是我们赋予了机器学习的能力；而Meta Learning并不是让机器直接从我们指定好的function范围内去学习，而是让它自己有能力自己去设计一个function的架构，然后再从这个范围内学习到最好的function。我们期待用这种方式让机器自己寻找到那个最合适的model，从而得到比人类指定model的方法更为有效的结果。

传统：我们指定model->机器从这个model中学习出best function

Meta：我们教会机器设计model的能力->机器自己设计model->机器从这个model中学习出best function

原因：人为指定的model实际上效率并不高，我们常常见到machine在某些任务上的表现比较好，但是这是它花费大量甚至远超于人类所需的时间和资料才能达到和人类一样的能力。相当于我们指定的model直接定义了这是一个天资不佳的机器，只能通过让它勤奋不懈的学习才能得到好的结果；由于人类的智慧有限无法设计高效的model才导致机器学习效率低下，因此Meta learning就期望让机器自己去定义自己的天赋，从而具备更高效的学习能力。

- Life-long Learning

一般的机器学习都是针对某一个任务设计的model，而Life-long Learning想要让机器能够具备终身学习的能力，让它不仅能够学会处理任务1，还能接着学会处理任务2、3...也就是让机器成为一个全能型人才

Regression

3 Steps

- 定义一个model即function set
- 定义一个goodness of function，损失函数去评估该function的好坏
- 找一个最好的function

Step 1: Model

Linear Model

$$y = b + w \cdot X_{cp}$$

y 代表进化后的cp值， X_{cp} 代表进化前的cp值， w 和**b**代表未知参数，可以是任何数值

根据不同的 w 和**b**，可以确定不同的无穷无尽的function，而 $y = b + w \cdot X_{cp}$ 这个抽象出来的式子就叫做model，是以上这些具体化的function的集合，即function set

实际上这是一种**Linear Model**，我们可以将其扩展为：

$$y = b + \sum w_i x_i$$

x_i: an attribute of input X (x_i is also called **feature**, 即特征值)

w_i: weight of x_i

b: bias

Step 2: Goodness of Function

x^i : 用上标来表示一个完整的object的编号， x^i 表示第*i*只宝可梦(下标表示该object中的component)

\hat{y}^i : 用 \hat{y} 表示一个real data的标签，上标为*i*表示是第*i*个object

由于regression的输出值是scalar，因此 \hat{y} 里面并没有component，只是一个简单的数值；但是未来如果考虑structured Learning的时候，我们output的object可能是有structured的，所以我们还是会需要用上标下标来表示一个完整的output的object和它包含的component

Loss function

The loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set.

为了衡量function set中的某个function的好坏，我们需要一个评估函数，即Loss function，损失函数。Loss function是一个function的function

$$L(f) = L(w, b)$$

Input: a function; output: how bad/good it is

由于 $f: y = b + w \cdot x_{cp}$ ，即 f 是由**b**和 w 决定的，因此Loss function实际上是在衡量一组参数的好坏

最常用的方法就是采用类似于方差和的形式来衡量参数的好坏，即预测值与真值差的平方和

这里真正的数值减估测数值的平方，叫做估测误差，Estimation error，将10个估测误差合起来就是loss function

$$L(f) = L(w, b) = \sum_{n=1}^{10} (\hat{y}^n - (b + w \cdot x_{cp}^n))^2$$

如果 $L(f)$ 越大，说明该function表现得越不好； $L(f)$ 越小，说明该function表现得越好

Step 3: Best Function

挑选最好的function写成formulation/equation就是：

$$f^* = \arg \min_f L(f)$$

或者是

$$w^*, b^* = \arg \min_{w,b} L(w, b) = \arg \min_{w,b} \sum_{n=1}^{10} (\hat{y}^n - (b + w \cdot x_{cp}^n))^2$$

使 $L(f) = L(w, b)$ 最小的 f 或 (w, b) ，就是我们要找的 f^* 或 (w^*, b^*)

Gradient Descent

只要 $L(f)$ 是可微分的，Gradient Descent都可以拿来处理 f ，找到表现比较好的parameters

单个参数的问题

以只带单个参数 w 的Loss Function $L(w)$ 为例

首先保证 $L(w)$ 是可微的

我们的目标就是找到这个使Loss最小的 $w^* = \arg \min_w L(w)$ ，实际上就是寻找切线 L 斜率为0的global minima，但是存在一些local minima极小值点，其斜率也是0

有一个暴力的方法是，穷举所有的 w 值，去找到使loss最小的 w^* ，但是这样做是没有效率的；而gradient descent就是用来解决这个效率问题的

- 首先随机选取一个初始的点 w^0 （当然也不一定要随机选取，如果有办法可以得到比较接近 w^* 的表现得比较好的 w^0 当初始点，可以有效地提高查找 w^* 的效率）
- 计算 L 在 $w = w^0$ 的位置的微分，即 $\frac{dL}{dw}|_{w=w^0}$ ，几何意义就是切线的斜率
- 如果切线斜率是negative，那么就应该使 w 变大，即往右踏一步；如果切线斜率是positive，那么就应该使 w 变小，即往左踏一步，每一步的步长step size就是 w 的改变量

w 改变量step size的大小取决于两件事

- 一是现在的微分值 $\frac{dL}{dw}$ 有多大，微分值越大代表现在在一个越陡峭的地方，那它要移动的距离就越大，反之就越小；
- 二是一个常数项 η ，被称为**learning rate**，即学习率，它决定了每次踏出的step size不只取决于现在的斜率，还取决于一个事先就定好的数值，如果learning rate比较大，那每踏出一步的时候，参数 w 更新的幅度就比较大，反之参数更新的幅度就比较小

如果learning rate设置的大一些，那机器学习的速度就会比较快；但是learning rate如果太大，可能就会跳过最合适的global minima的点

- 因此每次参数更新的大小是 $\eta \frac{dL}{dw}$ ，为了满足斜率为负时 w 变大，斜率为正时 w 变小，应当使原来的 w 减去更新的数值，即

$$\begin{aligned} w^1 &= w^0 - \eta \frac{dL}{dw}|_{w=w^0} \\ w^2 &= w^1 - \eta \frac{dL}{dw}|_{w=w^1} \\ w^3 &= w^2 - \eta \frac{dL}{dw}|_{w=w^2} \\ &\dots \\ w^{i+1} &= w^i - \eta \frac{dL}{dw}|_{w=w^i} \\ \text{if } \frac{dL}{dw}|_{w=w^i} &== 0 \text{ stop} \end{aligned}$$

w^i 对应的斜率为0，我们找到了一个极小值local minima。

这就出现了一个问题，当微分为0的时候，参数就会一直卡在这个点上没有办法再更新了，因此通过gradient descent找出来的solution其实并不是最佳解global minima

但幸运的是，在linear regression上，是没有local minima的，因此可以使用这个方法

两个参数的问题

今天要解决的关于宝可梦的问题，是含有two parameters的问题，即 $(w^*, b^*) = \arg \min_{w,b} L(w, b)$

当然，它本质上处理单个参数的问题是一样的

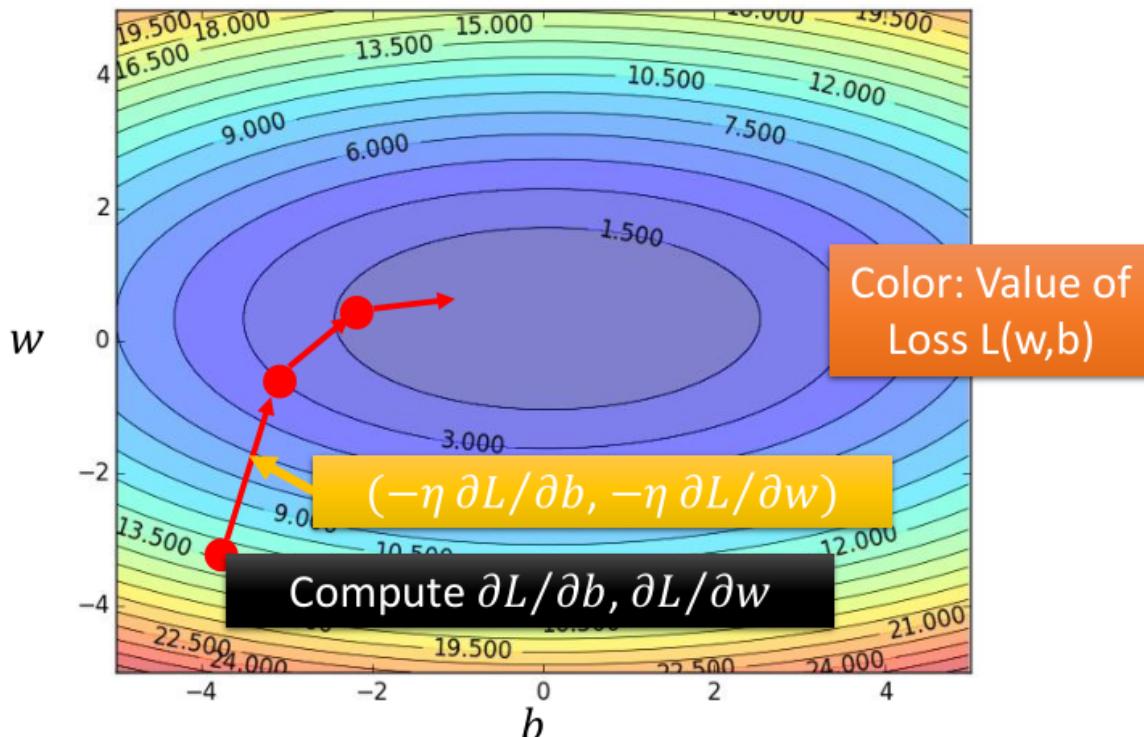
- 首先，也是随机选取两个初始值， w^0 和 b^0
- 然后分别计算 (w^0, b^0) 这个点上， L 对 w 和 b 的偏微分，即 $\frac{\partial L}{\partial w}|_{w=w^0, b=b^0}$ 和 $\frac{\partial L}{\partial b}|_{w=w^0, b=b^0}$
- 更新参数，当迭代跳出时， (w^i, b^i) 对应着极小值点

$$\begin{aligned} w^1 &= w^0 - \eta \frac{\partial L}{\partial w}|_{w=w^0, b=b^0} & b^1 &= b^0 - \eta \frac{\partial L}{\partial b}|_{w=w^0, b=b^0} \\ w^2 &= w^1 - \eta \frac{\partial L}{\partial w}|_{w=w^1, b=b^1} & b^2 &= b^1 - \eta \frac{\partial L}{\partial b}|_{w=w^1, b=b^1} \\ &\dots \\ w^{i+1} &= w^i - \eta \frac{\partial L}{\partial w}|_{w=w^i, b=b^i} & b^{i+1} &= b^i - \eta \frac{\partial L}{\partial b}|_{w=w^i, b=b^i} \\ \text{if } \frac{\partial L}{\partial w} &= 0 \text{ and } \frac{\partial L}{\partial b} = 0 \text{ stop} \end{aligned}$$

实际上， L 的gradient就是微积分中的那个梯度的概念，即

$$\nabla L = \left[\begin{array}{c} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{array} \right]_{gradient}$$

每次计算得到的梯度gradient，就是由 $\frac{\partial L}{\partial b}$ 和 $\frac{\partial L}{\partial w}$ 组成的vector向量，就是该点等高线的法线方向；而 $(-\eta \frac{\partial L}{\partial b}, -\eta \frac{\partial L}{\partial w})$ 的作用就是让原先的 (w^i, b^i) 朝着gradient的反方向前进，其中learning rate的作用是每次更新的跨度(对应图中红色箭头的长度)；经过多次迭代，最终gradient达到极小值点



这里两个方向的learning rate必须保持一致，这样每次更新坐标的step size是等比例缩放的，保证坐标前进的方向始终和梯度下降的方向一致；否则坐标前进的方向将会发生偏移

local minima

gradient descent有一个令人担心的地方，它每次迭代完毕，寻找到的梯度为0的点必然是极小值点 local minima；却不一定是最小值点 global minima

这会造成一个问题，如果loss function长得比较坑坑洼洼（极小值点比较多），而每次初始化 w^0 的取值又是随机的，这会造成每次gradient descent停下来的位置都可能是不同的极小值点

而且当遇到梯度比较平缓（gradient ≈ 0 ）的时候，gradient descent也可能会效率低下甚至可能会stuck也就是说通过这个方法得到的结果，是看人品的

但是在linear regression里，loss function实际上是**convex**的，是一个凸函数，是没有local optimal 局部最优解的，它只有一个global minima，visualize出来的图像就是从里到外一圈一圈包围起来的椭圆形的等高线，因此随便选一个起始点，根据gradient descent最终找出来的，都会是同一组参数

Overfitting

随着 $(x_{cp})^i$ 的高次项的增加，对应的average error会不断地减小

实际上这件事情非常容易解释，实际上低次的式子是高次的式子的特殊情况（令高次项 $(x_{cp})^i$ 对应的 w_i 为0，高次式就转化成低次式）

在gradient descent可以找到best function的前提下（多次式为Non-linear model，存在local optimal，gradient descent不一定能找到global minima）function所包含的项的次数越高，越复杂，error在training data上的表现就会越来越小

但是，我们关心的不是model在training data上的error表现，而是model在testing data上的error表现，在training data上，model越复杂，error就会越低；但是在testing data上，model复杂到一定程度之后，error非但不会减小，反而会暴增。

从含有 $(x_{cp})^4$ 项的model开始往后的model，testing data上的error出现了大幅增长的现象，通常被称为**Overfitting**

原来的loss function只考虑了prediction error，即 $\sum_n (\hat{y}^n - (b + \sum w_i x_i))^2$

regularization则是在原来的loss function的基础上加上了一项 $\lambda \sum (w_i)^2$ ，就是把这个model里面所有的 w_i 的平方和用 λ 加权

也就是说，我们期待参数 w_i 越小甚至接近于0的function。为什么呢？

因为参数值接近0的function，是比较平滑的；所谓的平滑的意思是，当今天的输入有变化的时候，output对输入的变化是比较不敏感的

举例来说，对 $y = b + \sum w_i x_i$ 这个model，当input变化 Δx_i ，output的变化就是 $w_i \Delta x_i$ ，也就是说，如果 w_i 越小越接近0的话，输出对输入就越不sensitive，我们的function就是一个越平滑的function；

我们没有把bias b这个参数考虑进去的原因是，bias的大小跟function的平滑程度是没有关系的，bias值的大小只是把function上下移动而已

如果我们有一个比较平滑的function，由于输出对输入是不敏感的，测试的时候，一些noises噪音对这个平滑的function的影响就会比较小，而给我们一个比较好的结果

这里的 λ 需要我们手动去调整以取得最好的值

λ 值越大代表考虑smooth的那个regularization那一项的影响力越大，我们找到的function就越平滑

当我们的 λ 越大的时候，在training data上得到的error其实是越大的，但是这件事情是非常合理的，因为当 λ 越大的时候，我们就越倾向于考虑w的值而越少考虑error的大小

但是有趣的是，虽然在training data上得到的error越大，但是在testing data上得到的error可能会是比较小的；当然 λ 太大的时候，在testing data上的error就会越来越大

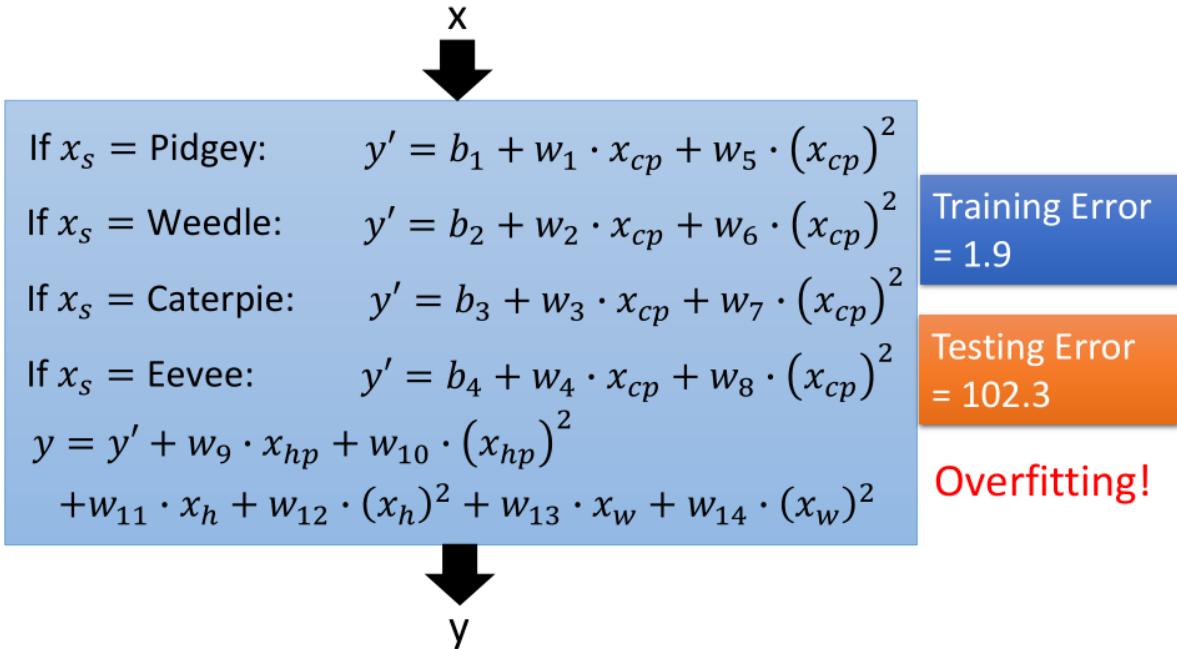
我们喜欢比较平滑的function，因为它对noise不那么sensitive；但是我们又不喜欢太平滑的function，因为它就失去了对data拟合的能力；而function的平滑程度，就需要通过调整 λ 来决定

Conclusion

根据已有的data特点(labeled data，包含宝可梦及进化后的cp值)，确定使用supervised learning监督学习
根据output的特点(输出的是scalar数值)，确定使用regression回归(linear or non-linear)

Back to step 1: Redesign the Model Again

考虑包括进化前cp值、species、hp等各方面变量属性以及高次项的影响，我们的model可以采用：



Back to step 2: Regularization

而为了保证function的平滑性，不overfitting，应使用regularization，即 $L = \sum_{i=1}^n (\hat{y}^i - y^i)^2 + \lambda \sum_j (w_j)^2$

注意bias b对function平滑性无影响

Back to step 3

利用gradient descent对regularization版本的loss function进行梯度下降迭代处理，每次迭代都减去L对该参数的微分与learning rate之积，假设所有参数合成一个vector： $[w_0, w_1, w_2, \dots, w_j, \dots, b]^T$ ，那么每次梯度下降的表达式如下：

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_j} \\ \vdots \\ \frac{\partial L}{\partial b} \end{bmatrix}_{gradient}$$

gradient descent :

$$\begin{bmatrix} w'_0 \\ w'_1 \\ w'_2 \\ \vdots \\ w'_j \\ \vdots \\ b' \end{bmatrix}_{L=L'} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_j \\ \vdots \\ b \end{bmatrix}_{L=L_0} - \eta \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_j} \\ \vdots \\ \frac{\partial L}{\partial b} \end{bmatrix}_{L=L_0}$$

当梯度稳定不变时，即 ∇L 为0时，gradient descent便停止，此时如果采用的model是linear的，那么vector必然落于global minima处；如果采用的model是Non-linear的，vector可能会落于local minima处，此时需要采取其他办法获取最佳的function

假定我们已经通过各种方法到达了global minima的地方，此时的vector: $[w_0, w_1, w_2, \dots, w_j, \dots, b]^T$ 所确定的那个唯一的function就是在该 λ 下的最佳 f^* ，即loss最小

这里 λ 的最佳数值是需要通过我们不断调整来获取的，因此令 λ 等于0, 10, 100, 1000, ...不断使用gradient descent或其他算法得到最佳的parameters $[w_0, w_1, w_2, \dots, w_j, \dots, b]^T$ ，并计算出这组参数确定的function f^* 对training data和testing data上的error值，直到找到那个使testing data的error最小的 λ
 $\lambda=0$ 就是没有使用regularization时的loss function

Where does the error come from?

Estimator

\hat{y} 表示那个真正的function，而 f^* 表示这个 \hat{f} 的估测值estimator

就好像在打靶， \hat{f} 是靶的中心点，收集到一些data做training以后，你会得到一个你觉得最好的function即 f^* ，这个 f^* 落在靶上的某个位置，它跟靶中心有一段距离，这段距离就是由bias和variance决定的

实际上对应着物理实验中系统误差和随机误差的概念，假设有n组数据，每一组数据都会产生一个相应的 f^* ，此时bias表示所有 f^* 的平均落靶位置和真值靶心的距离，variance表示这些 f^* 的集中程度

Bias and Variance of Estimator

假设独立变量为x(这里的x代表每次独立地从不同的training data里训练找到的 f^*)，那么

总体期望 $E(x) = u$ ； 总体方差 $Var(x) = \sigma^2$

用样本均值 \bar{x} 估测总体期望 u

由于我们只有有限组样本 $\{x^1, x^2, \dots, x^N\}$ ，故样本均值 $\bar{x} = \frac{1}{N} \sum_{i=1}^N x^i \neq \mu$ ； 样本均值的期望

$E(\bar{x}) = E\left(\frac{1}{N} \sum_{i=1}^N x^i\right) = \mu$ ； 样本均值的方差 $Var(\bar{x}) = \frac{\sigma^2}{N}$

样本均值 \bar{x} 的期望是总体期望 μ , 也就是说 \bar{x} 是按概率对称地分布在总体期望 μ 的两侧的; 而 \bar{x} 分布的密集程度取决于 N , 即数据量的大小, 如果 N 比较大, \bar{x} 就会比较集中, 如果 N 比较小, \bar{x} 就会以 μ 为中心分散开来

综上, 样本均值 \bar{x} 以总体期望 μ 为对称分布, 可以用来估测总体期望 μ

用样本方差 s^2 估测总体方差 σ^2

由于我们只有有限组样本 $\{x^1, x^2, \dots, x^N\}$, 故样本均值 $\bar{x} = \frac{1}{N} \sum_{i=1}^N x^i$; 样本方差 $s^2 = \frac{1}{N-1} \sum_{i=1}^N (x^i - \bar{x})^2$; 样本方差的期望 $E(s^2) = \frac{N-1}{N} \sigma^2 \neq \sigma^2$

同理, 样本方差 s^2 以总体方差 σ^2 为对称分布, 可以用来估测总体方差 σ^2 , 而 s^2 分布的密集程度也取决于 N

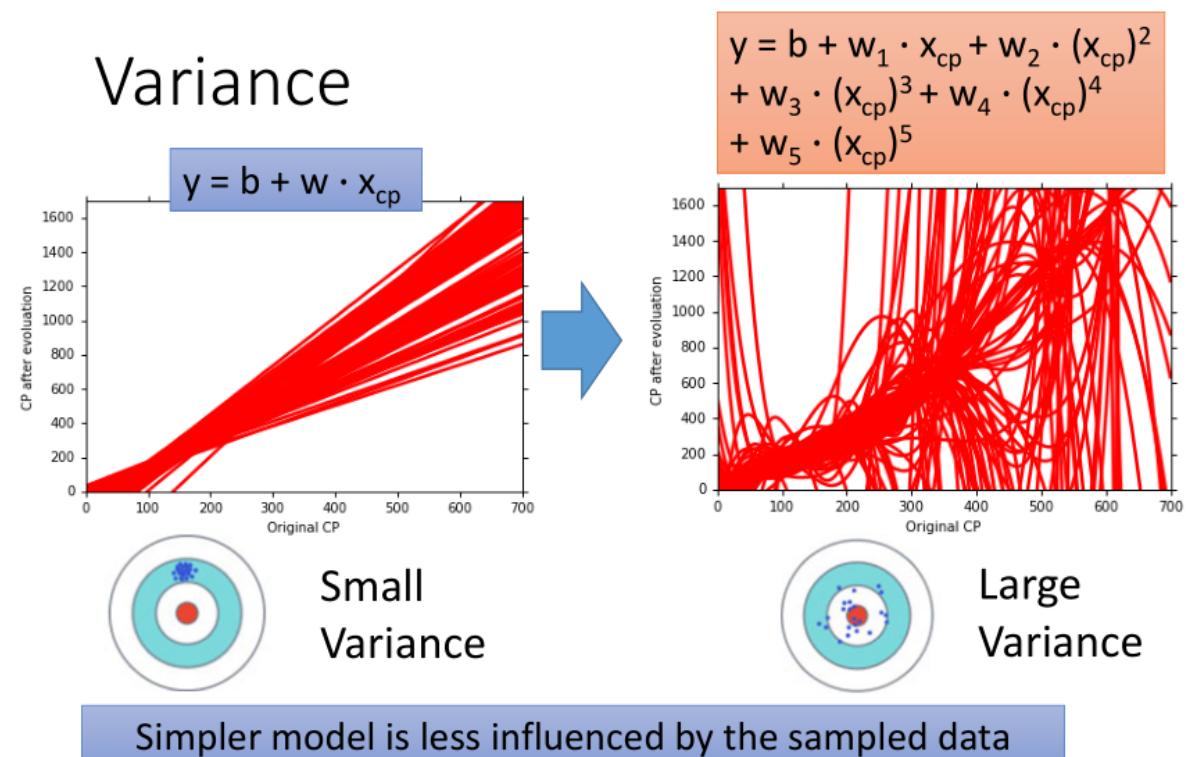
现在我们要估测的是靶的中心 \hat{f} , 每次 collect data 训练出来的 f^* 是打在靶上的某个点; 产生的 error 取决于:

- 多次实验得到的 f^* 的期望 \bar{f} 与靶心 \hat{f} 之间的 bias—— $E(f^*)$, 可以形象地理解为瞄准的位置和靶心的距离的偏差
- 多次实验的 f^* 之间的 variance—— $Var(f^*)$, 可以形象地理解为多次打在靶上的点的集中程度

Error

Variance

f^* 的 variance 是由 model 决定的, 一个简单的 model 在不同的 training data 下可以获得比较稳定分布的 f^* , 而复杂的 model 在不同的 training data 下的分布比较杂乱 (如果 data 足够多, 那复杂的 model 也可以得到比较稳定的分布)



Consider the extreme case $f(x) = c$

如果采用比较简单的 model, 那么每次在不同 data 下的实验所得到的不同的 f^* 之间的 variance 是比较小的, 就好像说, 你在射击的时候, 每次击中的位置是差不多的, 就如同下图中的 linear model, 100 次实验找出来的 f^* 都是差不多的

但是如果model比较复杂，那么每次在不同data下的实验所得到的不同的 f^* 之间的variance是比较大的，它的散布就会比较开，就如同图中含有高次项的model，每一条 f^* 都长得不太像，并且散布得很开

那为什么比较复杂的model，它的散布就比较开呢？比较简单的model，它的散布就比较密集呢？

原因其实很简单，其实前面在讲regularization正规化的时候也提到了部分原因。简单的model实际上就是没有高次项的model，或者高次项的系数非常小的model，这样的model表现得相当平滑，受到不同的data的影响是比较小的

举一个很极端的例子，我们的整个model(function set)里面，只有一个function: $f=c$ ，这个function只有一个常数项，因此无论training data怎么变化，从这个最简单的model里找出来的 f^* 都是一样的，它的variance就是等于0

Bias

bias是说，我们把所有的 f^* 平均起来得到 $E(f^*) = \bar{f}^*$ ，这个 \bar{f}^* 与真值 \hat{f} 有多接近

当然这里会有一个问题是说，总体的真值 \hat{f} 我们根本就没有办法知道，因此这里只是假定了一个 \hat{f}

当model比较简单的时候，每次实验得到的 f^* 之间的variance会比较小，这些 f^* 会稳定在一个范围内，但是它们的平均值 \bar{f} 距离真实值 \hat{f} 会有比较大的偏差；

而当model比较复杂的时候，每次实验得到的 f^* 之间的variance会比较大，实际体现出来就是每次重新实验得到的 f^* 都会与之前得到的有较大差距，但是这些差距较大的 f^* 的平均值 \bar{f} 却和真实值 \hat{f} 比较接近

也就是说，复杂的model，单次实验的结果是没有太大参考价值的，但是如果把考虑多次实验的结果的平均值，也许会对最终的结果有帮助

这里的单次实验指的是，用一组training data训练出model的一组有效参数以构成 f^* （每次独立实验使用的training data都是不同的）

因此

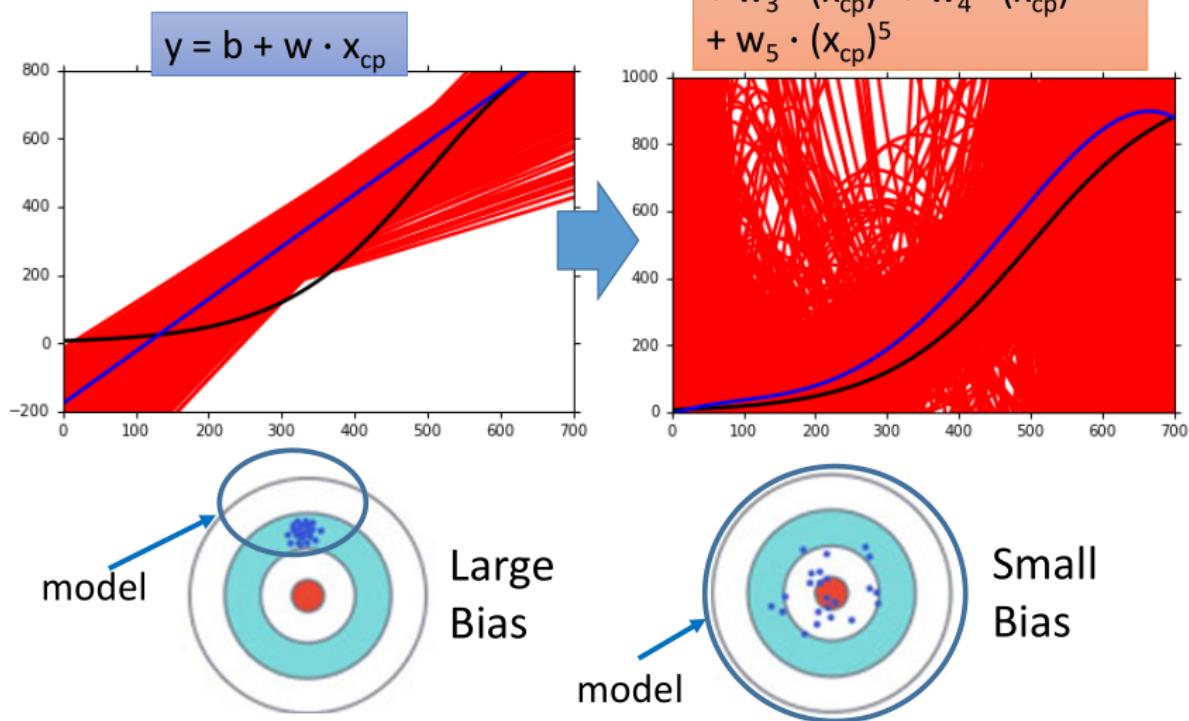
- 如果是一个比较简单的model，那它有比较小的variance和比较大的bias。每次实验的 f^* 都比较集中，但是他们平均起来距离靶心会有一段距离（比较适合实验次数少甚至只有单次实验的情况）
- 如果是一个比较复杂的model，每次实验找出来的 f^* 都不一样，它有比较大的variance但是却有比较小的bias。每次实验的 f^* 都比较分散，但是他们平均起来的位置与靶心比较接近（比较适合多次实验的情况）

Why?

实际上我们的model就是一个function set，当你定好一个model的时候，实际上就已经定好这个function set的范围了，那个最好的function只能从这个function set里面挑出来

如果是一个简单的model，它的function set的space是比较小的，这个范围可能根本就没有包含你的target；如果这个function set没有包含target，那么不管怎么sample，平均起来永远不可能是target

Bias

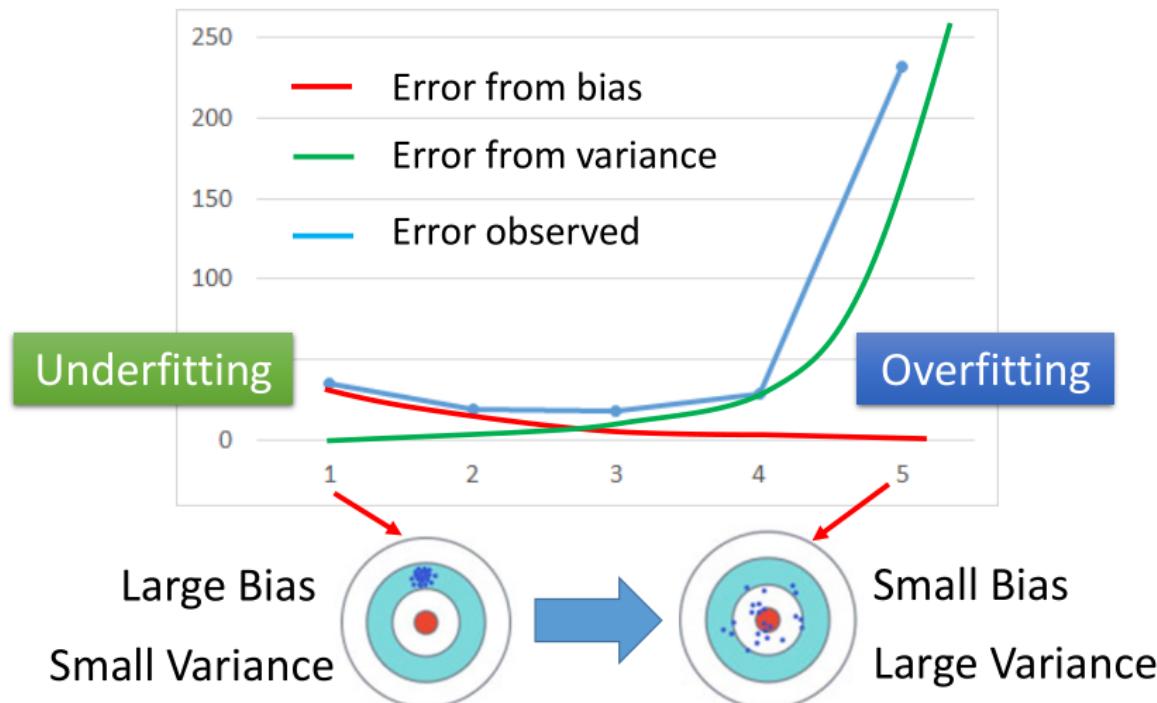


如果这个model比较复杂，那么这个model所代表的function set的space是比较大的，那它就很有可能包含target

只是它没有办法找到那个target在哪，因为你给的training data不够，你给的training data每一次都不一样，所以它每一次找出来的 f^* 都不一样，但是如果他们是散布在这个target附近的，那平均起来，实际上就可以得到和target比较接近的位置

Bias vs Variance

由前面的讨论可知，比较简单的model，variance比较小，bias比较大；而比较复杂的model，bias比较小，variance比较大



$$\text{error}_{\text{observed}} = \text{error}_{\text{variance}} + \text{error}_{\text{bias}}$$

可以发现，随着model的逐渐复杂：

- bias逐渐减小，bias所造成的error也逐渐下降，也就是打靶的时候瞄得越来越准
- variance逐渐变大，variance所造成的error也逐渐增大，也就是虽然瞄得越来越准，但是每次射出去以后，你的误差是越来越大的
- 当bias和variance这两项同时被考虑的时候，也就是实际体现出来的error的变化
- 实际观测到的error先是减小然后又增大，因此实际error为最小值的那个点，即为bias和variance的error之和最小的点，就是表现最好的model
- 如果实际error主要来自于variance很大，这个状况就是**overfitting**过拟合；如果实际error主要来自于bias很大，这个状况就是**underfitting**欠拟合。

这就是我们之前要先计算出每一个model对应的error，再挑选error最小的model的原因

只有这样才能综合考虑bias和variance的影响，找到一个实际error最小的model

Where does the error come from?

当你自己在做research的时候，你必须要搞清楚，手头上的这个model，它目前主要的error是来源于哪里；你觉得你现在的问题是bias大，还是variance大

你应该先知道这件事情，你才能知道你的future work，你要improve你的model的时候，你应该要走哪一个方向

- 如果model没有办法fit training data的examples，代表bias比较大，这时是underfitting
形象地说，就是该model找到的 f^* 上面并没有training data的大部分样本点，代表说这个model跟正确的model是有一段差距的，所以这个时候是bias大的情况，是underfitting
- 如果model可以fit training data，在training data上得到小的error，但是在testing data上，却得到一个大的error，代表variance比较大，这时是overfitting

遇到bias大或variance大的时候，你其实是要用不同的方式来处理它们

What to do with large bias?

bias大代表，你现在这个model里面可能根本没有包含你的target， \hat{f} 可能根本就不在你的function set里对于error主要来自于bias的情况，是由于该model (function set) 本来就不好，collect更多的data是没有用的，必须要从model本身出发redesign，重新设计你的model

For bias, redesign your model

- Add more features as input
比如pokemon的例子里，只考虑进化前cp值可能不够，还要考虑hp值、species种类...作为model新的input变量
- Add more features as input

What to do with large variance?

- More data
 - Very effective, but not always practical
 - 如果是5次式，找100个 f^* ，每次实验我们只用10只宝可梦的数据训练model，那我们找出来的100个 f^* 的分布就会杂乱无章；但如果每次实验我们用100只宝可梦的数据训练model，那我们找出来的100个 f^* 的分布就非常地集中

- 增加data是一个很有效控制variance的方法，假设你variance太大的话，collect data几乎是一个万能的东西，并且它不会伤害你的bias。但是它存在一个很大的问题是，实际上并没有办法去collect更多的data
- 如果没有办法collect更多的data，其实有一招，根据你对这个问题的理解，自己去generate更多“假的”data
 - 比如手写数字识别，因为每个人手写数字的角度都不一样，那就把所有training data里面的数字都左转 15° ，右转 15°
 - 比如做火车的影像辨识，只有从左边开过来的火车影像资料，没有从右边开过来的火车影像资料，实际上可以把每张图片都左右颠倒，就generate出右边的火车数据了，这样就多了一倍data出来
 - 比如做语音辨识的时候，只有男生说的“你好”，没有女生说的“你好”，那就用男生的声音用一个变声器把它转化一下，这样男女生的声音就可以互相转化，这样data就多了
 - 比如现在你只有录音室里录下的声音，但是detection实际要在真实场景下使用的，那你就去真实场景下录一些噪音加到原本的声音里，就可以generate出符合条件的data
- Regularization
 - 在loss function里面再加一个与model高次项系数相关的term，它会希望你的model里高次项的参数越小越好，也就是说希望你今天找出来的曲线越平滑越好；这个新加的term前面可以有一个weight，代表你希望你的曲线有多平滑
 - 加了regularization后，一些怪怪的、很不平滑的曲线就不会再出现，所有曲线都集中在比较平滑的区域；增加weight可以让曲线变得更平滑
 - 加了regularization以后，因为你强迫所有的曲线都要比较平滑，所以这个时候也会让你的variance变小。**但regularization是可能会伤害bias的，因为它实际上调整了function set的space范围，变成它只包含那些比较平滑的曲线**，这个缩小的space可能没有包含原先在更大space内的 \hat{f} ，因此伤害了bias，所以当你做regularization的时候，需要调整regularization的weight，在variance和bias之间取得平衡

Model Selection

我们现在会遇到的问题往往是这样：我们有很多个model可以选择，还有很多参数可以调，比如regularization的weight，那通常我们是在bias和variance之间做一些trade-off

我们希望找一个model，它variance够小，bias也够小，这两个合起来给我们最小的testing data的error

Cross Validation

你要做的事情是，把你的training set分成两组：

- 一组是真正拿来training model的，叫做training set(训练集)
- 另外一组不拿它来training model，而是拿它来选model，叫做validation set(验证集)

先在training set上找出每个model最好的function f^* ，然后用validation set来选择你的model

也就是说，你手头上有3个model，你先把这3个model用training set训练出三个 f^* ，接下来看一下它们在validation set上的performance

假设现在model 3的performance最好，那你可以直接把这个model 3的结果拿来apply在testing data上

如果你担心现在把training set分成training和validation两部分，感觉training data变少的话，可以这样做：已经从validation决定model 3是最好的model，那就定住model 3不变(function的表达式不变)，然后使用全部的数据去更新model 3表达式的参数

这个时候，如果你把这个训练好的model的 f^* apply到public testing set上面，虽然这么做，你得到的error表面上看起来是比较大的，但是这个时候你在public set上的error才能够真正反映你在private set上的error

当你得到public set上的error的时候（尽管它可能会很大），不建议回过头去重新调整model的参数，因为当你再回去重新调整什么东西的时候，你就又会把public testing set的bias给考虑进去了，这就又回到围绕着有偏差的testing data做model的优化。这样的话此时你在public set上看到的performance就没有办法反映实际在private set上的performance了。因为你的model是针对public set做过优化的，虽然public set上的error数据看起来可能会更好看，但是针对实际未知的private set，这个“优化”带来的可能是反作用，反而会使实际的error变大

因此这里只是说，你要keep in mind，benchmark corpus上面所看到的testing的performance，说不定是别人特意调整过的，并且testing set与实际的数据也会有偏差，它的error，肯定是大于它在real application上应该有的值

比如说你现在常常会听到说，在image lab的那个corpus上面，error rate都降到3%，已经超越人类了。但是真的是这样子吗？如果已经用testing data调过参数了，你把那些model真的apply到现实生活中，它的error rate肯定是大于3%的。

N-fold Cross Validation

如果你不相信某一次分train和validation的结果的话，那你就分很多种不同的样子

比如说，如果你做3-fold的validation，把training set分成三份

你每一次拿其中一份当做validation set，另外两份当training；分别在每个情境下都计算一下3个model的error，然后计算一下它的average error；然后你会发现在这三个情境下的average error，是model 1最好

然后接下来，你就把用整个完整的training data重新训练一遍model 1的参数；然后再去testing data上test

原则上是，如果你少去根据public testing set上的error调整model的话，那你在private testing set上面得到的error往往是比较接近public testing set上的error的

Gradient Descent

Gradient Descent

$$\theta^* = \arg \min_{\theta} L(\theta)$$

L : loss function

θ : parameters(上标表示第几组参数，下标表示这组参数中的第几个参数)

Suppose that θ has two variables $\{\theta_1, \theta_2\}$

Randomly start at $\theta^0 = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix}$

计算 θ 处的梯度： $\nabla L(\theta) = \begin{bmatrix} \partial L(\theta_1) / \partial \theta_1 \\ \partial L(\theta_2) / \partial \theta_2 \end{bmatrix}$

$$\begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\theta_1^0) / \partial \theta_1 \\ \partial L(\theta_2^0) / \partial \theta_2 \end{bmatrix} \Rightarrow \theta^1 = \theta^0 - \eta \nabla L(\theta^0)$$

$$\begin{bmatrix} \theta_1^2 \\ \theta_2^2 \end{bmatrix} = \begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\theta_1^1) / \partial \theta_1 \\ \partial L(\theta_2^1) / \partial \theta_2 \end{bmatrix} \Rightarrow \theta^2 = \theta^1 - \eta \nabla L(\theta^1)$$

在整个gradient descent的过程中，梯度不一定是递减的，但是沿着梯度下降的方向，函数值loss一定是递减的（如果学习率足够小），且当gradient=0时，loss下降到了局部最小值，梯度下降法指的是函数值loss随梯度下降的方向减小

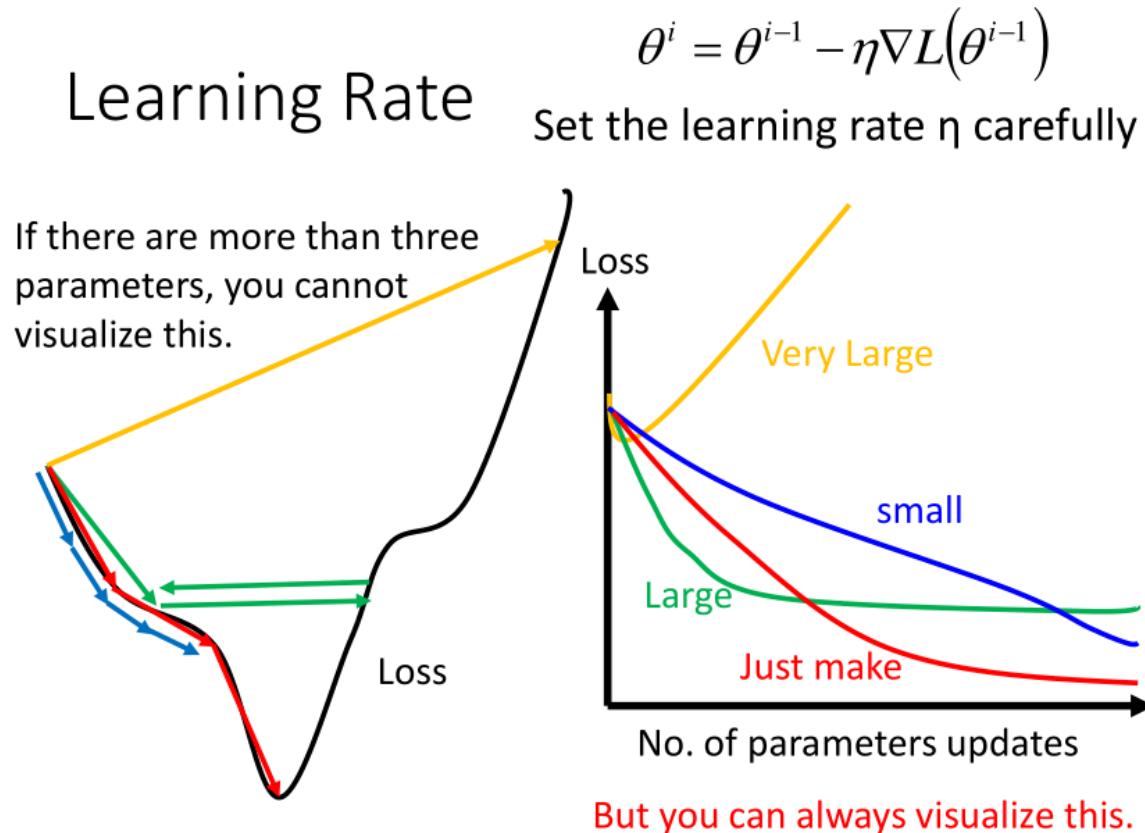
初始随机在三维坐标系中选取一个点，这个三维坐标系的三个变量分别为 $(\theta_1, \theta_2, loss)$ ，我们的目标是找到最小的那个loss也就是三维坐标系中高度最低的那个点，而gradient梯度（Loss等高线的法线方向）可以理解为高度上升最快的那个方向，它的反方向就是梯度下降最快的那个方向

于是每次update沿着梯度反方向，update的步长由梯度大小和learning rate共同决定，当某次update完成后，该点的gradient=0，说明到达了局部最小值

Tip 1: Tuning your learning rates

- 如果learning rate刚刚好，就可以顺利地到达到loss的最小值
- 如果learning rate太小的话，虽然最后能够走到local minimal的地方，但是它可能会走得非常慢，以至于你无法接受
- 如果learning rate太大，它的步伐太大了，它永远没有办法走到特别低的地方，可能永远在这个“山谷”的口上振荡而无法走下去
- 如果learning rate非常大，可能一瞬间就飞出去了，结果会造成update参数以后，loss反而会越来越大

当参数有很多个的时候(>3)，其实我们很难做到将loss随每个参数的变化可视化出来（因为最多只能可视化出三维的图像，也就只能可视化三维参数），但是我们可以把update的次数作为唯一的一个参数，将loss随着update的增加而变化的趋势给可视化出来



所以做gradient descent一个很重要的事情是，要把不同的learning rate下，loss随update次数的变化曲线给可视化出来，它可以提醒你该如何调整当前的learning rate的大小，直到出现稳定下降的曲线

Adaptive Learning rates

显然这样手动地去调整learning rates很麻烦，因此我们需要有一些自动调整learning rates的方法

最基本、最简单的大原则是：learning rate通常是随着参数的update越来越小的

因为在起始点的时候，通常是离最低点是比较远的，这时候步伐就要跨大一点；而经过几次update以后，会比较靠近目标，这时候就应该减小learning rate，让它能够收敛在最低点的地方

举例：假设到了第t次update，此时 $\eta^t = \eta / \sqrt{t+1}$

这种方法使所有参数以同样的方式同样的learning rate进行update，而最好的状况是每个参数都给它不同的learning rate去update

Adagrad

Divide the learning rate of each parameter by the root mean square(方均根) of its previous derivatives

Adagrad就是将不同参数的learning rate分开考虑的一种算法（adagrad算法update到后面速度会越来越慢，当然这只是adaptive算法中最简单的一种）

这里的w是function中的某个参数，t表示第t次update， g^t 表示Loss对w的偏微分，而 σ^t 是之前所有Loss对w偏微分的方均根(根号下的平方均值)，这个值对每一个参数来说都是不一样的

Adagrad

$$\begin{aligned} w^1 &= w^0 - \frac{\eta^0}{\sigma^0} \cdot g^0 \quad \sigma^0 = \sqrt{(g^0)^2} \\ w^2 &= w^1 - \frac{\eta^1}{\sigma^1} \cdot g^1 \quad \sigma^1 = \sqrt{\frac{1}{2}[(g^0)^2 + (g^1)^2]} \\ w^3 &= w^2 - \frac{\eta^2}{\sigma^2} \cdot g^2 \quad \sigma^2 = \sqrt{\frac{1}{3}[(g^0)^2 + (g^1)^2 + (g^2)^2]} \\ &\dots \\ w^{t+1} &= w^t - \frac{\eta^t}{\sigma^t} \cdot g^t \quad \sigma^t = \sqrt{\frac{1}{1+t} \sum_{i=0}^t (g^i)^2} \end{aligned}$$

由于 η^t 和 σ^t 中都有一个 $\sqrt{\frac{1}{1+t}}$ 的因子，两者相消，即可得到adagrad的最终表达式：

$$w^{t+1} = w^t - \frac{\eta}{\sqrt{\sum_{i=0}^t (g^i)^2}} \cdot g^t$$

Contradiction

Adagrad的表达式 $w^{t+1} = w^t - \frac{\eta}{\sqrt{\sum_{i=0}^t (g^i)^2}} \cdot g^t$ 里面有一件很矛盾的事情：

我们在做gradient descent的时候，希望的是当梯度值即微分值 g^t 越大的时候（此时斜率越大，还没有接近最低点）更新的步伐要更大一些，但是Adagrad的表达式中，分母表示梯度越大步伐越小，分子却表示梯度越大步伐越大，两者似乎相互矛盾

Intuitive Reason $\eta^t = \frac{\eta}{\sqrt{t+1}}$ $g^t = \frac{\partial C(\theta^t)}{\partial w}$

- How surprise it is 反差

特別大

g^0	g^1	g^2	g^3	g^4
0.001	0.001	0.003	0.002	0.1
g^0	g^1	g^2	g^3	g^4
10.8	20.9	31.7	12.1	0.1

特別小

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^t (g^i)^2}} g^t \longrightarrow \text{造成反差的效果}$$

在一些paper里是这样解释的：Adagrad要考虑的是，这个gradient有多surprise，即反差有多大，假设t=4的时候 g^4 与前面的gradient反差特别大，那么 g^t 与 $\sqrt{\frac{1}{t+1} \sum_{i=0}^t (g^i)^2}$ 之间的大小反差就会比较大，它们的商就会把这一反差效果体现出来

同时，gradient越大，离最低点越远这件事情在有多个参数的情况下是不一定成立的

实际上，对于一个二次函数 $y = ax^2 + bx + c$ 来说，最小值点的 $x = -\frac{b}{2a}$ ，而对于任意一点 x_0 ，它迈出最好的步伐长度是 $|x_0 + \frac{b}{2a}| = |\frac{2ax_0 + b}{2a}|$ (这样就一步迈到最小值点了)，联系该函数的一阶和二阶导数 $y' = 2ax + b$ 、 $y'' = 2a$ ，可以发现the best step is $|\frac{y'}{y''}|$ ，也就是说他不仅跟一阶导数(gradient)有关，还跟二阶导数有关

再来看看Adagrad的表达式： $w^{t+1} = w^t - \frac{\eta}{\sqrt{\sum_{i=0}^t (g^i)^2}} \cdot g^t$

g^t 就是一次微分，而分母中的 $\sum_{i=0}^t (g^i)^2$ 反映了二次微分的大小，所以Adagrad想要做的事情就是在不增加任何额外运算的前提下，想办法去估测二次微分的值

Tip 2: Stochastic Gradient Descent

随机梯度下降的方法可以让训练更快速，传统的gradient descent的思路是看完所有的样本点之后再构建loss function，然后去update参数；而stochastic gradient descent的做法是，看到一个样本点就update一次，因此它的loss function不是所有样本点的error平方和，而是这个随机样本点的error平方

Tip 3: Mini-batch Gradient Descent

这里有一个秘密，就是我们在做deep learning的gradient descent的时候，并不会真的去minimize total loss，那我们做的是什么呢？我们会把Training data分成一个一个的batch，比如说你的Training data一共有万张image，每次random选100张image作为一个batch

- 像gradient descent一样，先随机initialize network的参数
- 选第一个batch出来，然后计算这个batch里面的所有element的total loss， $L' = l^1 + l^{31} + \dots$ ，接下来根据 L' 去update参数，也就是计算 L' 对所有参数的偏微分，然后update参数

注意： L' 不是全部data的total loss

- 再选择第二个batch，现在这个batch的total loss是 $L'' = l^2 + l^{16} + \dots$ ，接下来计算 L'' 对所有参数的偏微分，然后update参数
- 反复做这个process，直到把所有的batch通通选过一次，所以假设你有100个batch的话，你就把这个参数update 100次，把所有batch看过一次，就叫做一个epoch
- 重复epoch的过程，所以你在train network的时候，你会需要好几个epoch，而不是只有一个epoch

整个训练的过程类似于stochastic gradient descent，不是将所有数据读完才开始做gradient descent的，而是拿到一部分数据就做一次gradient descent

Batch size and Training Speed

batch size太小会导致不稳定，速度上也没有优势

前面已经提到了，stochastic gradient descent速度快，表现好，既然如此，为什么我们还要用Mini-batch呢？这就涉及到了一些实际操作上的问题，让我们必须去用Mini-batch

举例来说，我们现在有50000个examples，如果我们把batch size设置为1，就是stochastic gradient descent，那在一个epoch里面，就会update 50000次参数；如果我们把batch size设置为10，在一个epoch里面，就会update 5000次参数

看上去stochastic gradient descent的速度貌似是比较快的，它一个epoch更新参数的次数比batch size等于10的情况下要快了10倍，但是我们好像忽略了一个问题，我们之前一直都是下意识地认为不同batch size的情况下运行一个epoch的时间应该是相等的，然后我们才去比较每个epoch所能够update参数的次数，可是它们又怎么可能相等的呢？

实际上，当你batch size设置不一样的时候，一个epoch需要的时间是不一样的，以GTX 980为例

- case 1：如果batch size设为1，也就是stochastic gradient descent，一个epoch要花费166秒，接近3分钟
- case 2：如果batch size设为10，那一个epoch是17秒

也就是说，当stochastic gradient descent算了一个epoch的时候，batch size为10的情况已经算了近10个epoch了

所以case 1跑一个epoch，做了50000次update参数的同时，case 2跑了十个epoch，做了近 $5000 * 10 = 50000$ 次update参数；你会发现batch size设1和设10，update参数的次数几乎是一样的

如果不同batch size的情况，update参数的次数几乎是一样的，你其实会想要选batch size更大的情况，相较于batch size=1，你会更倾向于选batch size=10，因为batch size=10的时候，是会比较稳定的，因为由更大的数据集计算的梯度能够更好的代表样本总体，从而更准确的朝向极值所在的方向

我们之前把gradient descent换成stochastic gradient descent，是因为后者速度比较快，update次数比较多，可是现在如果你用stochastic gradient descent并没有觉得有多快，那你为什么不选一个update次数差不多，又比较稳定的方法呢？

batch size会受到GPU平行加速的限制，太大可能导致在train的时候卡住

上面例子的现象产生的原因是我们在用了GPU，用了平行运算，所以batch size=10的时候，这10个example其实是同时运算的，所以你在一个batch里算10个example的时间跟算1个example的时间几乎可以是一样的

那你可能会问，既然batch size越大，它会越稳定，而且还可以平行运算，那为什么不把batch size变得超级大呢？这里有两个claim：

- 第一个claim就是，如果你把batch size开到很大，最终GPU会没有办法进行平行运算，它终究是有自己的极限的，也就是说它同时考虑10个example和1个example的时间是一样的，但当它考虑10000个example的时候，时间就不可能还是跟1个example一样，因为batch size考虑到硬件限制，是没有办法无穷尽地增长的
- 第二个claim是说，如果把batch size设的很大，在train gradient descent的时候，可能跑两下你的network就卡住了，就陷到saddle point或者local minima里面去了

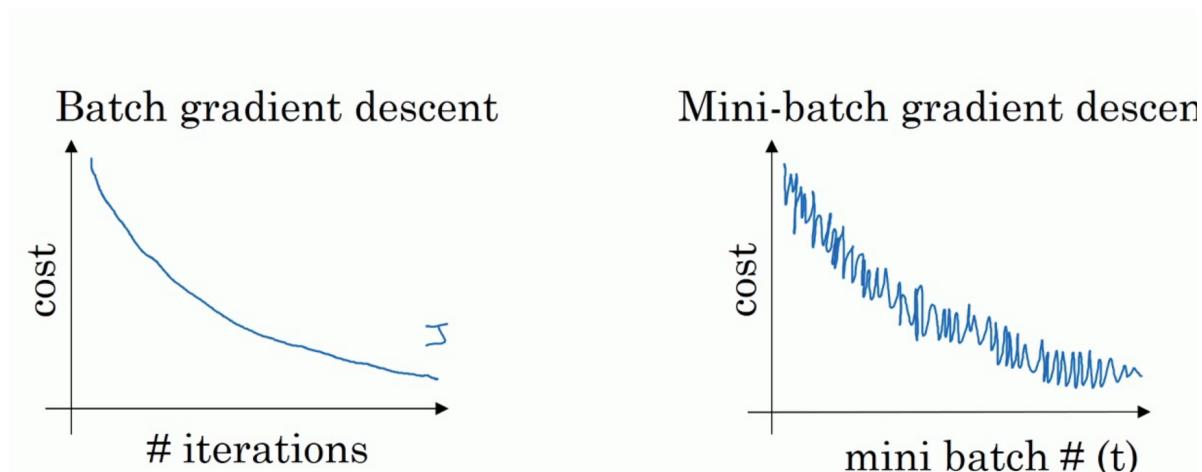
因为在neural network的error surface上面，如果你把loss的图像可视化出来的话，它并不是一个convex的optimization problem，不会像理想中那么平滑，实际上它会有很多的坑坑洞洞

如果你用的batch size很大，甚至是Full batch，那你走过的路径会是比较平滑连续的，可能这一条平滑的曲线在走向最低点的过程中就会在坑洞或是缓坡上卡住了；但是，如果你的batch size没有那么大，意味着你走的路线没有那么的平滑，有些步伐走的是随机性的，路径是会有一些曲折和波动的

可能在你走的过程中，它的曲折和波动刚好使得你绕过了那些saddle point或是local minima的地方；或者当你陷入不是很深的local minima或者没有遇到特别麻烦的saddle point的时候，它步伐的随机性就可以帮你跳出这个gradient接近于0的区域，于是你更有可能真的走向global minima的地方

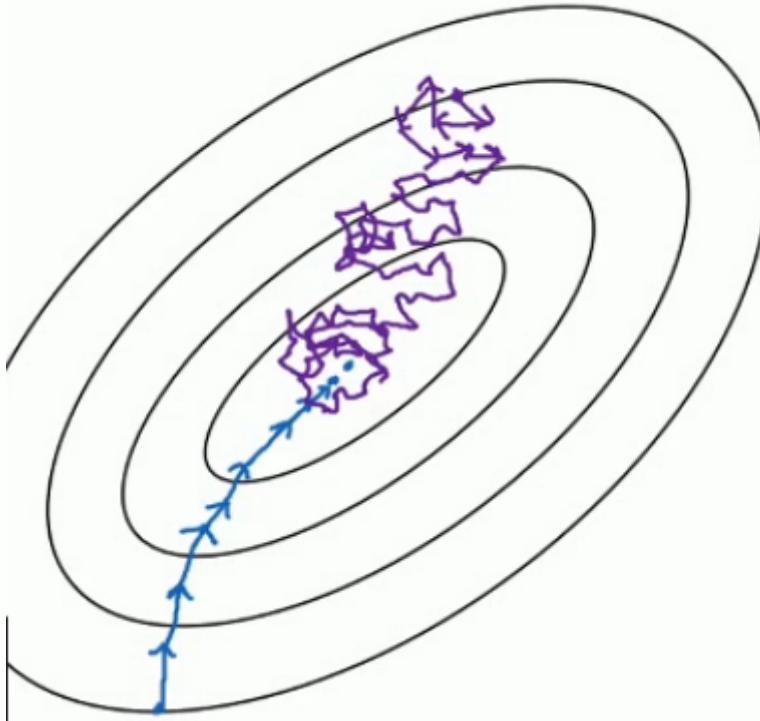
而对于Full batch的情况，它的路径是没有随机性的，是稳定朝着目标下降的，因此在这个时候去train neural network其实是有问题的，可能update两三次参数就会卡住，所以mini batch是有必要的

如下图，左边是full batch(拿全部的Training data做一个batch)的梯度下降效果，可以看到每一次迭代成本函数都呈现下降趋势，这是好的现象，说明我们w和b的设定一直再减少误差，这样一直迭代下去我们就可以找到最优解；右边是mini batch的梯度下降效果，可以看到它是上下波动的，成本函数的值有时高有时低，但总体还是呈现下降的趋势，这个也是正常的，因为我们每一次梯度下降都是在min batch上跑的而不是在整个数据集上，数据的差异可能会导致这样的波动(可能某段数据效果特别好，某段数据效果不好)，但没关系，因为它整体是呈下降趋势的



把下面的图看做是梯度下降空间：蓝色部分是full batch而紫色部分是mini batch，就像上面所说的mini batch不是每次迭代损失函数都会减少，所以看上去好像走了很多弯路，不过整体还是朝着最优解迭代的，而且由于mini batch一个epoch就走了5000步（5000次梯度下降），而full batch一个epoch只有一步，所以虽然mini batch走了弯路但还是会快很多

而且，就像之前提到的那样，mini batch在update的过程中，步伐具有随机性，因此紫色的路径可以在一定程度上绕过或跳出 saddle point、local minima这些gradient趋近于0的地方；而蓝色的路径因为缺乏随机性，只能按照既定的方式朝着目标前进，很有可能就在中途被卡住，永远也跳不出来了



当然，如果batch size太小，会造成速度不仅没有加快反而会导致下降的曲线更加不稳定的情况产生

因此batch size既不能太大，因为它会受到硬件GPU平行加速的限制，导致update次数过于缓慢，并且由于缺少随机性而很容易在梯度下降的过程中卡在saddle point或是local minima的地方；

而且batch size也不能太小，因为它会导致速度优势不明显的情况下，梯度下降曲线过于不稳定，算法可能永远也不会收敛

Speed - Matrix Operation

整个network，不管是Forward pass还是Backward pass，都可以看做是一连串的矩阵运算的结果

那今天我们就来比较batch size等于1(stochastic gradient descent)和10(mini batch)的差别

如下图所示，stochastic gradient descent就是对每一个input x 进行单独运算；而mini batch，则是把同一个batch里面的input全部集合起来，假设现在我们的batch size是2，那mini batch每一次运算的input就是把黄色的vector和绿色的vector拼接起来变成一个matrix，再把这个matrix乘上 w_1 ，你就可以直接得到 z^1 和 z^2

这两件事在理论上运算量是一样的，但是在实际操作上，对GPU来说，在矩阵里面相乘的每一个element都是可以平行运算的，所以图中stochastic gradient descent运算的时间反而会变成下面mini batch使用GPU运算速度的两倍，这就是为什么我们要使用mini batch的原因

Speed - Matrix Operation

- Why mini-batch is faster than stochastic gradient descent?

Stochastic Gradient Descent

$$z^1 = W^1 \quad x \quad z^1 = W^1 \quad x \quad \dots$$

Mini-batch

$$\begin{matrix} z^1 & z^1 \\ \boxed{\quad \quad} & \boxed{\quad \quad} \end{matrix} = \begin{matrix} W^1 \\ \text{matrix} \end{matrix} \quad \begin{matrix} x & x \\ \boxed{\quad \quad} & \boxed{\quad \quad} \end{matrix}$$

Practically, which one is faster?

所以，如果你买了GPU，但是没有使用mini batch的话，其实就不会有多少加速的效果。

Tip 4: Feature Scaling

特征缩放，当多个特征的分布范围很不一样时，最好将这些不同feature的范围缩放成一样

$y = b + w_1 x_1 + w_2 x_2$, 假设 x_1 的值都是很小的，比如1,2...； x_2 的值都是很大的，比如100,200...

此时去画出loss的error surface，如果对 w_1 和 w_2 都做一个同样的变动 Δw ，那么 w_1 的变化对 y 的影响是比较小的，而 w_2 的变化对 y 的影响是比较大的

对于error surface表示， w_1 对 y 的影响比较小，所以 w_1 对loss是有比较小的偏微分的，因此在 w_1 的方向上图像是比较平滑的； w_2 对 y 的影响比较大，所以 w_2 对loss的影响比较大，因此在 w_2 的方向上图像是比较sharp的

如果 x_1 和 x_2 的值，它们的scale是接近的，那么 w_1 和 w_2 对loss就会有差不多的影响力，loss的图像接近于圆形，那这样做对gradient descent有什么好处呢？

对于长椭圆形的error surface，如果不使用Adagrad之类的方法，是很难搞定它的，因为在像 w_1 和 w_2 这样不同的参数方向上，会需要不同的learning rate，用相同的学习率很难达到最低点

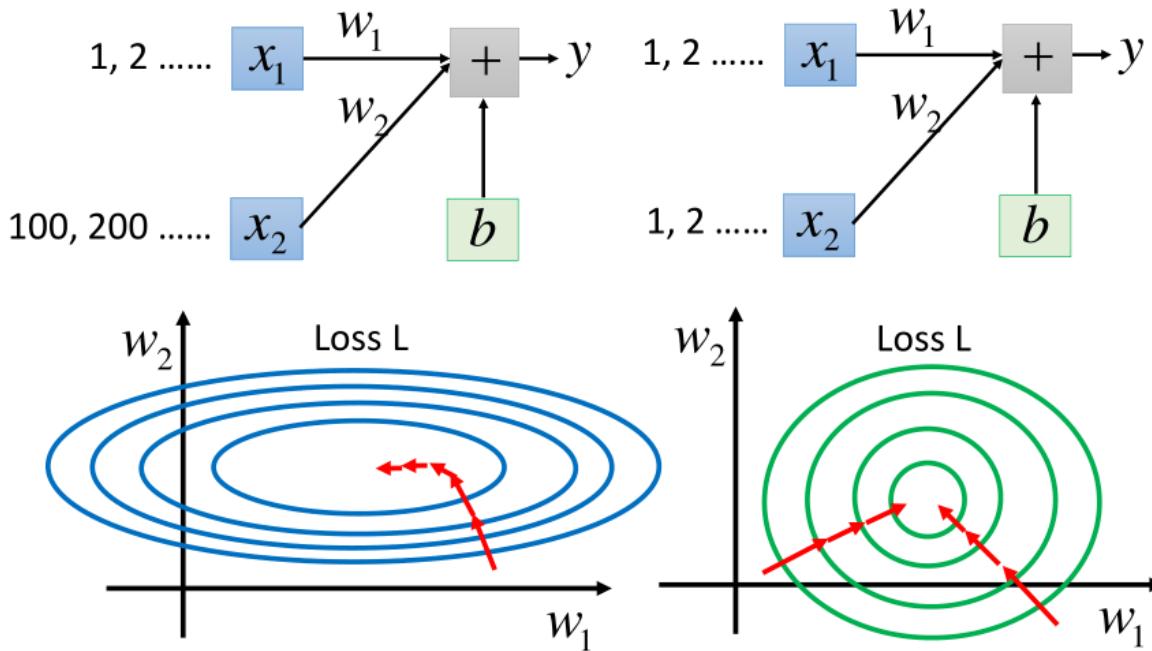
如果有scale的话，loss在参数 w_1 、 w_2 平面上的投影就是一个正圆形，update参数会比较容易

而且gradient descent的每次update并不都是向着最低点走的，每次update的方向是顺着等高线的方向（梯度gradient下降的方向），而不是径直走向最低点；

但是当经过对input的scale使loss的投影是一个正圆的话，不管在这个区域的哪一个点，它都会向着圆心走。因此feature scaling对参数update的效率是有帮助的。

Feature Scaling

$$y = b + w_1x_1 + w_2x_2$$

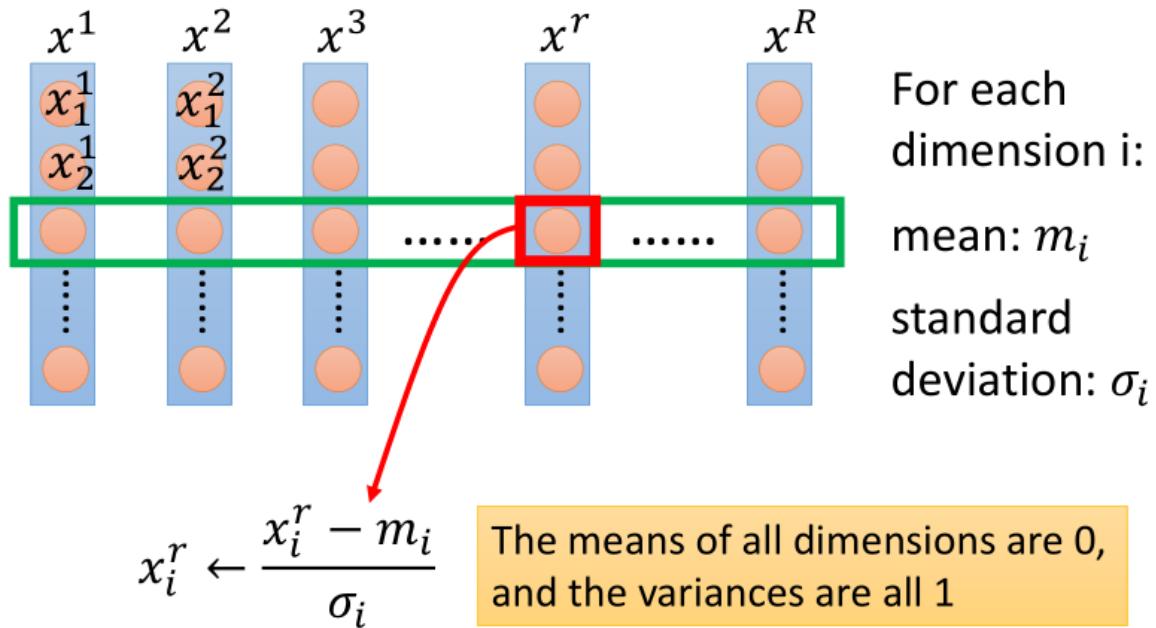


假设有R个example(上标i表示第i个样本点), $x^1, x^2, x^3, \dots, x^r, \dots x^R$, 每一筆example, 它里面都有一组feature(下标j表示该样本点的第j个特征)

对每一个dimension i, 都去算出它的平均值mean m_i , 以及标准差standard deviation σ_i

对第r个example的第i个component, 减掉均值, 除以标准差, 即 $x_i^r = \frac{x_i^r - m_i}{\sigma_i}$

实际上就是将每一个参数都归一化成标准正态分布, 即 $f(x_i) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x_i^2}{2}}$, 其中 x_i 表示第i个参数



Gradient Descent Theory

When solving: $\theta^* = \arg \min_{\theta} L(\theta)$ by gradient descent

Each time we update the parameters, we obtain θ that makes $L(\theta)$ smaller.

$$L(\theta^0) > L(\theta^1) > L(\theta^2) > \dots$$

Is this statement correct?

不正确

Taylor Series

泰勒表达式: $h(x) = \sum_{k=0}^{\infty} \frac{h^{(k)}(x_0)}{k!} (x - x_0)^k = h(x_0) + h'(x_0)(x - x_0) + \frac{h''(x_0)}{2!}(x - x_0)^2 + \dots$

When x is close to x_0 : $h(x) \approx h(x_0) + h'(x_0)(x - x_0)$

同理, 对于二元函数, when x and y is close to x_0 and y_0 :

$$h(x, y) \approx h(x_0, y_0) + \frac{\partial h(x_0, y_0)}{\partial x}(x - x_0) + \frac{\partial h(x_0, y_0)}{\partial y}(y - y_0)$$

Formal Derivation

对于loss图像上的某一个点(a,b), 如果我们想要找这个点附近loss最小的点, 就可以用泰勒展开的思想

假设用一个red circle限定点的范围, 这个圆足够小以满足泰勒展开的精度, 那么此时我们的loss function就可以化简为:

$$L(\theta) \approx L(a, b) + \frac{\partial L(a, b)}{\partial \theta_1}(\theta_1 - a) + \frac{\partial L(a, b)}{\partial \theta_2}(\theta_2 - b)$$

$$\text{令 } s = L(a, b), u = \frac{\partial L(a, b)}{\partial \theta_1}, v = \frac{\partial L(a, b)}{\partial \theta_2}$$

$$\text{则 } L(\theta) \approx s + u \cdot (\theta_1 - a) + v \cdot (\theta_2 - b)$$

假定red circle的半径为d, 则有限制条件: $(\theta_1 - a)^2 + (\theta_2 - b)^2 \leq d^2$

此时去求 $L(\theta)_{min}$, 这里有个小技巧, 把 $L(\theta)$ 转化为两个向量的乘积:

$$u \cdot (\theta_1 - a) + v \cdot (\theta_2 - b) = (u, v) \cdot (\theta_1 - a, \theta_2 - b) = (u, v) \cdot (\Delta \theta_1, \Delta \theta_2)$$

当向量 $(\theta_1 - a, \theta_2 - b)$ 与向量 (u, v) 反向, 且刚好到达red circle的边缘时, $L(\theta)$ 最小

$(\theta_1 - a, \theta_2 - b)$ 实际上就是 $(\Delta \theta_1, \Delta \theta_2)$, 于是 $L(\theta)$ 局部最小值对应的参数为中心点减去gradient的加权

用 η 去控制向量的长度

$$\begin{bmatrix} \Delta \theta_1 \\ \Delta \theta_2 \end{bmatrix} = -\eta \begin{bmatrix} u \\ v \end{bmatrix} \Rightarrow \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} - \eta \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial L(a, b)}{\partial \theta_1} \\ \frac{\partial L(a, b)}{\partial \theta_2} \end{bmatrix}$$

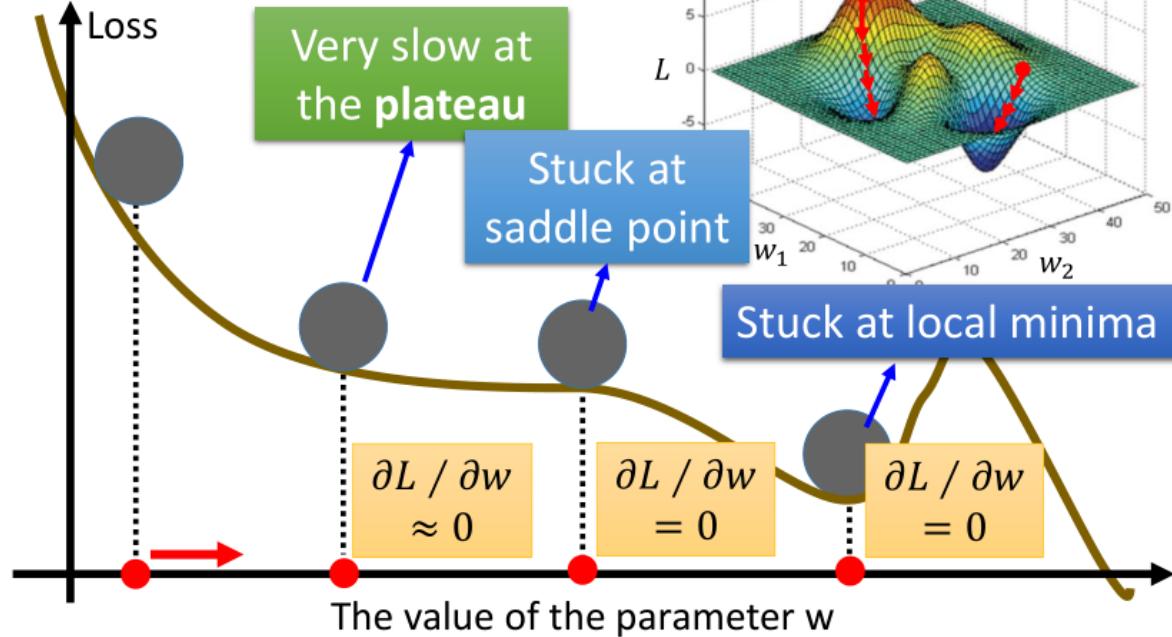
这就是gradient descent在数学上的推导, 注意它的重要前提是, 给定的那个红色圈圈的范围要足够小, 这样泰勒展开给我们的近似才会更精确, 而 η 的值是与圆的半径成正比的, 因此理论上learning rate要无穷小才能够保证每次gradient descent在update参数之后的loss会越来越小, 于是当learning rate没有设置好, 泰勒近似不成立, 就有可能使gradient descent过程中的loss没有越来越小

当然泰勒展开可以使用二阶、三阶乃至更高阶的展开, 但这样会使得运算量大大增加, 反而降低了运行效率

More Limitation of Gradient Descent

gradient descent的限制是，它在gradient即微分值接近于0的地方就会停下来，而这个地方不一定是global minima，它可能是local minima，可能是saddle point鞍点，甚至可能是一个loss很高的plateau平缓高原

More Limitation of Gradient Descent



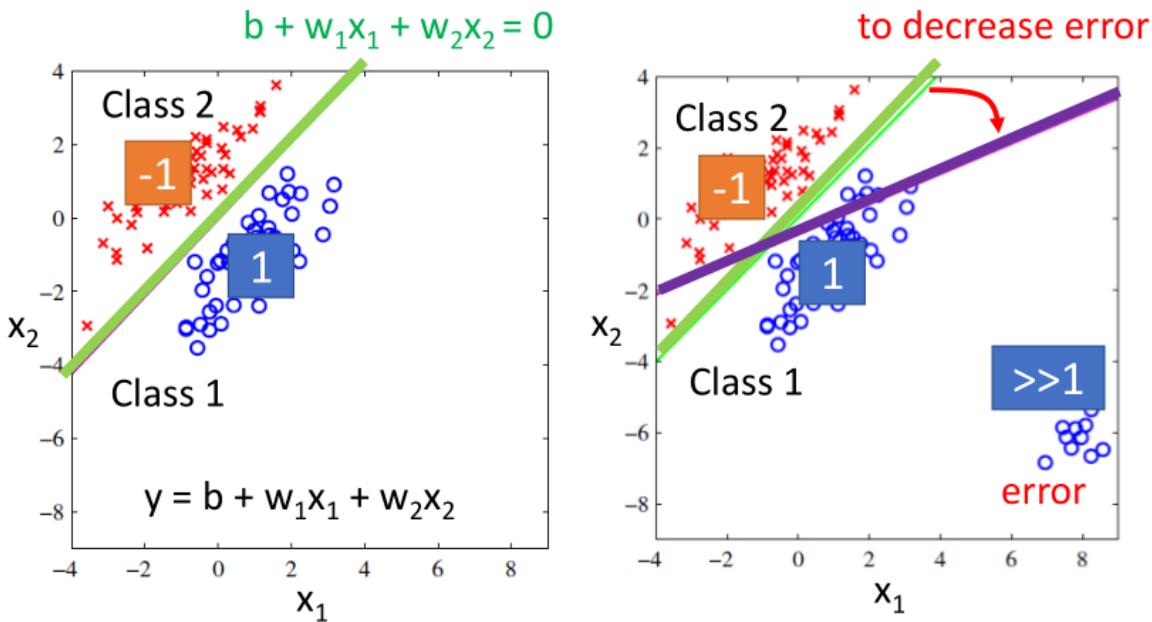
Classification: Probabilistic Generative Model

Classification

分类问题是找一个function，它的input是一个object，它的输出是这个object属于哪一个class
要想把一个东西当做function的input，就需要把它数值化

How to classification

Classification as Regression?



Penalize to the examples that are “too correct” ... (Bishop, P186)

- Multiple class: Class 1 means the target is 1; Class 2 means the target is 2; Class 3 means the target is 3 problematic
- Penalize to the examples that are “too correct”
- 如果是多元分类问题，把class 1的target当做是1，class 2的target当做是2，class 3的target当做是3的做法是错误的，因为当你这样做的时候，就会被Regression认为class 1和class 2的关系是比较接近的，class 2和class 3的关系是比较接近的，而class 1和class 3的关系是比较疏远的；但是当这些class之间并没有什么特殊的关系的时候，这样的标签用Regression是没有办法得到好的结果的。

Ideal Alternatives

Regression的output是一个real number，但是在classification的时候，它的output是discrete(用来表示某一个class)

Function(Model)

我们要找的function $f(x)$ 里面会有另外一个function $g(x)$ ，当我们的input x 输入后，如果 $g(x) > 0$ ，那 $f(x)$ 的输出就是class 1，如果 $g(x) < 0$ ，那 $f(x)$ 的输出就是class 2，这个方法保证了function的output都是离散的表示class的数值

之前不是说输出是1,2,3...是不行的吗，注意，那是针对Regression的loss function而言的，因为Regression的loss function是用output与“真值”的平方和作为评判标准的，这样输出值(3,2)与(3,1)之间显然是(3,2)关系更密切一些，为了解决这个问题，我们只需要重新定义一个loss function即可

Loss function

我们可以把loss function定义成 $L(f) = \sum_n \delta(f(x^n) \neq \hat{y}^n)$ ，即这个model在所有的training data上predict预测错误的次数，也就是说分类错误的次数越少，这个function表现得就越好

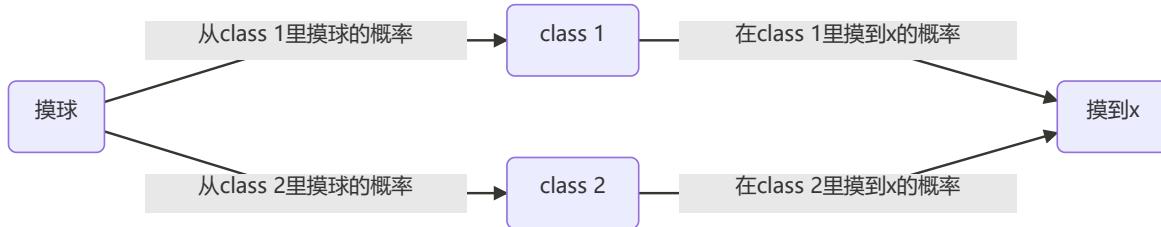
但是这个loss function没有办法微分，无法用gradient descent的方法去解的，当然有Perceptron、SVM这些方法可以用，但这里先用另外一个solution来解决这个问题

Generative model

假设我们考虑一个二元分类的问题，我们拿到一个input x ，想要知道这个 x 属于class 1或class 2的概率

实际上就是一个贝叶斯公式， x 属于class 1的概率就等于class 1自身发生的概率乘上在class 1里取出 x 这种颜色的球的概率除以在class 1和 class 2里取出 x 这种颜色的球的概率（后者是全概率公式）

贝叶斯公式=单条路径概率/所有路径概率和



- x 属于Class 1的概率为第一条路径除以两条路径和: $P(C_1|x) = \frac{P(C_1)P(x|C_1)}{P(C_1)P(x|C_1)+P(C_2)P(x|C_2)}$
- x 属于Class 2的概率为第二条路径除以两条路径和: $P(C_2|x) = \frac{P(C_2)P(x|C_2)}{P(C_1)P(x|C_1)+P(C_2)P(x|C_2)}$

因此我们想要知道 x 属于class 1或是class 2的概率，只需要知道4个值： $P(C_1), P(x|C_1), P(C_2), P(x|C_2)$ ，我们希望从Training data中估测出这四个值

这一整套想法叫做Generative model，因为如果你可以计算出每一个 x 出现的概率，就可以用这个distribution分布来生成 x 、sample x 出来

Prior Probability

$P(C_1)$ 和 $P(C_2)$ 这两个概率，被称为Prior，计算这两个值还是比较简单的

假设我们还是考虑二元分类问题，Water and Normal type with ID < 400 for training, rest for testing，如果想要严谨一点，可以在Training data里面分一部分validation出来模拟testing的情况。

在Training data里面，有79只水系宝可梦，61只一般系宝可梦，那么 $P(C_1) = 79/(79 + 61) = 0.56$ ， $P(C_2) = 61/(79 + 61) = 0.44$

现在的问题是，怎么得到 $P(x|C_1)$ 和 $P(x|C_2)$ 的值

Probability from Class

Gaussian Distribution

这里 u 表示均值， Σ 表示方差，那高斯函数的概率密度函数则是：

Input: vector x , output: probability of sampling x

The shape of the function determines by mean u and covariance matrix Σ

$$f_{u,\Sigma}(x) = \frac{1}{(2\pi)^{\frac{D}{2}}} \frac{1}{|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-u)^T \Sigma^{-1} (x-u)}$$

同样的 Σ ，不同的 u ，概率分布最高点的地方是不一样的；同理，如果是同样的 u ，不同的 Σ ，概率分布最高点的地方是一样的，但是分布的密集程度是不一样的。

那就从这79个已有的点找出Gaussian，只需要去估测出这个Gaussian的均值 u 和协方差 Σ 即可

Maximum Likelihood

估测 u 和 Σ 的方法就是**Maximum Likelihood**, 极大似然估计的思想是, 找出最特殊的那对 u 和 Σ , 从它们共同决定的高斯函数中再次采样出79个点, 使得到的分布情况与当前已知79点的分布情况相同发生的可能性最大

极大似然函数 $L(u, \Sigma) = f_{u,\Sigma}(x^1) \cdot f_{u,\Sigma}(x^2) \dots f_{u,\Sigma}(x^{79})$, 实际上就是该事件发生的概率就等于每个点都发生的概率之积, 我们只需要把每一个点的data代进去, 就可以得到一个关于 u 和 Σ 的函数, 分别求偏导, 解出微分是0的点, 即使L最大的那组参数, 便是最终的估测值, 通过微分得到的高斯函数 u 和 Σ 的最优解如下:

$$u^*, \Sigma^* = \arg \max_{u, \Sigma} L(u, \Sigma)$$

$$u^* = \frac{1}{79} \sum_{n=1}^{79} x^n \quad \Sigma^* = \frac{1}{79} \sum_{n=1}^{79} (x^n - u^*)(x^n - u^*)^T$$

当然如果你不愿意去求微分的话, 这也可以当做公式来记忆(u^* 刚好是数学期望, Σ^* 刚好是协方差)

数学期望: $u = E(X)$, 协方差: $\Sigma = cov(X, Y) = E[(X - u)(Y - u)^T]$, 对同一个变量来说, 协方差为 $cov(X, X) = E[(X - u)(X - u)^T]$

Do Classification

根据 $P(C_1|x) = \frac{P(C_1)P(x|C_1)}{P(C_1)P(x|C_1) + P(C_2)P(x|C_2)}$, 只要带入某一个input x , 就可以通过这个式子计算出它属于 class 1 的机率

$$f_{\mu^1, \Sigma^1}(x) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma^1|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu^1)^T (\Sigma^1)^{-1} (x - \mu^1) \right\}$$

$$\mu^1 = \begin{bmatrix} 75.0 \\ 71.3 \end{bmatrix} \quad \Sigma^1 = \begin{bmatrix} 874 & 327 \\ 327 & 929 \end{bmatrix}$$

$$P(C_1|x) = \frac{P(x|C_1)P(C_1)}{P(x|C_1)P(C_1) + P(x|C_2)P(C_2)}$$

$$f_{\mu^2, \Sigma^2}(x) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma^2|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu^2)^T (\Sigma^2)^{-1} (x - \mu^2) \right\}$$

$$\mu^2 = \begin{bmatrix} 55.6 \\ 59.8 \end{bmatrix} \quad \Sigma^2 = \begin{bmatrix} 847 & 422 \\ 422 & 685 \end{bmatrix}$$

$P(C_1) = 79 / (79 + 61) = 0.56$
 $P(C_2) = 61 / (79 + 61) = 0.44$

If $P(C_1|x) > 0.5 \rightarrow x \text{ belongs to class 1 (Water)}$

Modifying Model

其实之前使用的model是不常见的, 你不会经常看到给每一个Gaussian都有自己的mean和covariance, 比如我们的class 1用的是 u_1 和 Σ_1 , class 2用的是 u_2 和 Σ_2

比较常见的做法是, 不同的class可以share同一个covariance matrix

其实variance是跟input的feature size的平方成正比的, 所以当feature的数量很大的时候, Σ 大小的增长是可以非常快的, 在这种情况下, 给不同的Gaussian以不同的covariance matrix, 会造成model的参数太多, 而参数多会导致该model的variance过大, 出现overfitting的现象, 因此对不同的class使用同一个 covariance matrix, 可以有效减少参数

此时就把 u_1 、 u_2 和共同的 Σ 一起去合成一个极大似然函数，此时可以发现，得到的 u_1 和 u_2 和原来一样，还是各自的均值，而 Σ 则是原先两个 Σ_1 和 Σ_2 的加权 $\Sigma = \frac{79}{140}\Sigma_1 + \frac{61}{140}\Sigma_2$

看一下结果，class 1和class 2在没有共用covariance matrix之前，它们的分界线是一条曲线，正确率只有54%；如果共用covariance matrix的话，它们之间的分界线就会变成一条直线，这样的model，我们也称之为linear model（尽管Gaussian不是linear的，但是它分两个class的boundary是linear）

如果我们考虑所有的feature，并共用covariance的话，原来的54%的正确率就会变成73%。但是为什么能做到这样子，我们是很难分析的，因为这是在高维空间中发生的事情，我们很难知道boundary到底是怎么切的，但这就是machine learning它fancy的地方，人没有办法知道怎么做，但是machine可以帮助我们做出来

Three Steps of classification

- Find a function set(model)

prior probability $P(C)$ 和probability distribution $P(x|C)$ 就是model的参数

当posterior Probability $P(C|x) > 0.5$ 的话，就output class 1，反之就output class 2

- Goodness of function

对于Gaussian distribution这个model来说，我们要评价的是决定这个高斯函数形状的均值 u 和协方差 Σ 这两个参数的好坏，而极大似然函数 $L(u, \Sigma)$ 的输出值，就评价了这组参数的好坏

- Find the best function

找到的那个最好的function，就是使 $L(u, \Sigma)$ 值最大的那组参数，实际上就是所有样本点的均值和协方差

$$u^* = \frac{1}{n} \sum_{i=0}^n x^i \quad \Sigma^* = \frac{1}{n} \sum_{i=0}^n (x^i - u^*)(x^i - u^*)^T$$

这里上标*i*表示第*i*个点，这里x是一个features的vector，用下标来表示这个vector中的某个feature

Probability distribution

Why Gaussian distribution

你可以选择自己喜欢的Probability distribution概率分布函数，如果你选择的是简单的分布函数（参数比较少），那你的bias就大，variance就小；如果你选择复杂的分布函数，那你的bias就小，variance就大，那你就可以用data set来判断一下，用什么样的Probability distribution作为model是比较好的

Naive Bayes Classifier

我们可以考虑这样一件事情，假设 $x = [x_1 \ x_2 \ x_3 \ \dots \ x_k \ \dots]$ 中每一个dimension x_k 的分布都是相互独立的，它们之间的covariance都是0，那我们就可以把x产生的机率拆解成 x_1, x_2, \dots, x_k 产生的机率之积

这里每一个dimension的分布函数都是一维的Gaussian distribution，如果这样假设的话，等于是说，原来那多维度的Gaussian，它的covariance matrix变成是diagonal，在不是对角线的地方，值都是0，这样就可以更加减少需要的参数量，就可以得到一个更简单的model

我们把上述这种方法叫做**Naive Bayes Classifier**，如果真的明确了所有的feature之间是相互独立的，是不相关的，使用朴素贝叶斯分类法的performance是会很好的

如果这个假设是不成立的，那么Naive Bayes Classifier的bias就会很大，它就不是一个好的classifier（朴素贝叶斯分类法本质就是减少参数）

总之，寻找model总的原则是，尽量减少不必要的参数，但是必然的参数绝对不能少

那怎么去选择分布函数呢？有很多时候凭直觉就可以看出来，比如某个feature是binary的，它代表是或不是，这个时候就不太可能是高斯分布了，而很有可能是Bernoulli distributions

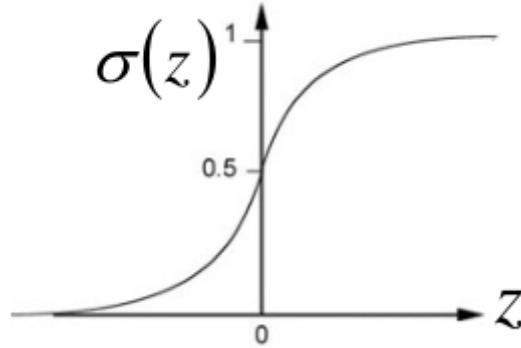
Posterior Probability

接下来我们来分析一下这个表达式，会发现一些有趣的现象

表达式上下同除以分子，令 $z = \ln \frac{P(C_2)P(x|C_2)}{P(C_1)P(x|C_1)}$

$$\begin{aligned} P(C_1|x) &= \frac{P(C_1)P(x|C_1)}{P(C_1)P(x|C_1)+P(C_2)P(x|C_2)} \\ &= \frac{1}{1+\frac{P(C_2)P(x|C_2)}{P(C_1)P(x|C_1)}} = \frac{1}{1+exp(-z)} = \sigma(z) \end{aligned}$$

得到 $\sigma(z) = \frac{1}{1+e^{-z}}$ ，这个function叫做sigmoid function



其中，Sigmoid函数是已知函数，因此我们来推导一下z的具体形式

$$\begin{aligned} P(C_1 | x) &= \sigma(z) \text{ sigmoid} \quad z = \ln \frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)} \\ z &= \ln \frac{P(x|C_1)}{P(x|C_2)} + \ln \frac{P(C_1)}{P(C_2)} \quad \frac{P(C_1)}{P(C_2)} = \frac{\frac{N_1}{N_1+N_2}}{\frac{N_2}{N_1+N_2}} = \frac{N_1}{N_2} \\ P(x | C_1) &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma^1|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu^1)^T (\Sigma^1)^{-1} (x - \mu^1) \right\} \\ P(x | C_2) &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma^2|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu^2)^T (\Sigma^2)^{-1} (x - \mu^2) \right\} \\ \ln \frac{P(x|C_1)}{P(x|C_2)} &= \ln \frac{\frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma^1|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu^1)^T (\Sigma^1)^{-1} (x - \mu^1) \right\}}{\frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma^2|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu^2)^T (\Sigma^2)^{-1} (x - \mu^2) \right\}} \\ &= \ln \frac{|\Sigma^2|^{1/2}}{|\Sigma^1|^{1/2}} \exp \left\{ -\frac{1}{2} \left[(x - \mu^1)^T (\Sigma^1)^{-1} (x - \mu^1) - (x - \mu^2)^T (\Sigma^2)^{-1} (x - \mu^2) \right] \right\} \\ &= \ln \frac{|\Sigma^2|^{1/2}}{|\Sigma^1|^{1/2}} - \frac{1}{2} \left[(x - \mu^1)^T (\Sigma^1)^{-1} (x - \mu^1) - (x - \mu^2)^T (\Sigma^2)^{-1} (x - \mu^2) \right] \\ (x - \mu^1)^T (\Sigma^1)^{-1} (x - \mu^1) &= x^T (\Sigma^1)^{-1} x - x^T (\Sigma^1)^{-1} \mu^1 - (\mu^1)^T (\Sigma^1)^{-1} x + (\mu^1)^T (\Sigma^1)^{-1} \mu^1 \\ &= x^T (\Sigma^1)^{-1} x - 2(\mu^1)^T (\Sigma^1)^{-1} x + (\mu^1)^T (\Sigma^1)^{-1} \mu^1 \\ (x - \mu^2)^T (\Sigma^2)^{-1} (x - \mu^2) &= x^T (\Sigma^2)^{-1} x - 2(\mu^2)^T (\Sigma^2)^{-1} x + (\mu^2)^T (\Sigma^2)^{-1} \mu^2 \\ z &= \ln \frac{|\Sigma^2|^{1/2}}{|\Sigma^1|^{1/2}} - \frac{1}{2} x^T (\Sigma^1)^{-1} x + (\mu^1)^T (\Sigma^1)^{-1} x - \frac{1}{2} (\mu^1)^T (\Sigma^1)^{-1} \mu^1 \\ &\quad + \frac{1}{2} x^T (\Sigma^2)^{-1} x - (\mu^2)^T (\Sigma^2)^{-1} x + \frac{1}{2} (\mu^2)^T (\Sigma^2)^{-1} \mu^2 + \ln \frac{N_1}{N_2} \end{aligned}$$

当 Σ_1 和 Σ_2 共用一个 Σ 时，经过化简相消 Σ 就变成了一个linear的function， x 的系数是一个vector w ，后面的一大串数字其实就是一个常数项 b

$$P(C_1|x) = \sigma(z)$$

$$\begin{aligned} z &= \ln \frac{|\Sigma^2|^{1/2}}{|\Sigma^1|^{1/2}} - \frac{1}{2} x^T (\Sigma^1)^{-1} x + (\mu^1)^T (\Sigma^1)^{-1} x - \frac{1}{2} (\mu^1)^T (\Sigma^1)^{-1} \mu^1 \\ &\quad + \frac{1}{2} x^T (\Sigma^2)^{-1} x - (\mu^2)^T (\Sigma^2)^{-1} x + \frac{1}{2} (\mu^2)^T (\Sigma^2)^{-1} \mu^2 + \ln \frac{N_1}{N_2} \end{aligned}$$

$$\Sigma_1 = \Sigma_2 = \Sigma$$

$$z = \frac{(\mu^1 - \mu^2)^T \Sigma^{-1} x - \frac{1}{2} (\mu^1)^T \Sigma^{-1} \mu^1 + \frac{1}{2} (\mu^2)^T \Sigma^{-1} \mu^2 + \ln \frac{N_1}{N_2}}{w^T b}$$

$$P(C_1|x) = \sigma(w \cdot x + b) \quad \text{How about directly find } w \text{ and } b?$$

In generative model, we estimate $N_1, N_2, \mu^1, \mu^2, \Sigma$

Then we have w and b

$P(C_1|x) = \sigma(w \cdot x + b)$ 这个式子就解释了，当class 1和class 2共用 Σ 的时候，它们之间的boundary会是linear的

在Generative model里面，我们做的事情是，我们用某些方法去找出 $N_1, N_2, u_1, u_2, \Sigma$ ，找出这些后算出 w 和 b ，把它们代进 $P(C_1|x) = \sigma(w \cdot x + b)$ ，就可以算概率，但是，当你看到这个式子的时候，你可能会有一个直觉的想法，为什么要这么麻烦呢？我们的最终目标都是要找一个vector w 和constant b ，我们何必先去搞个概率，算出一些 u, Σ 什么的，然后再回过头来又去算 w 和 b ，这不是舍近求远吗？

所以我们能不能直接把 w 和 b 找出来呢？

Logistic Regression

Step 1: Function Set

在Classification这一章节，我们讨论了如何通过样本点的均值 u 和协方差 Σ 来计算 $P(C_1), P(C_2), P(x|C_1), P(x|C_2)$ ，进而利用 $P(C_1|x) = \frac{P(C_1)P(x|C_1)}{P(C_1)P(x|C_1) + P(C_2)P(x|C_2)}$ 计算得到新的样本点 x 属于class 1的概率，由于是二元分类，属于class 2的概率 $P(C_2|x) = 1 - P(C_1|x)$ 。

可知 $P(C_1|x) = \sigma(z) = \frac{1}{1+e^{-z}}$, $z = \ln \frac{P(C_2)P(x|C_2)}{P(C_1)P(x|C_1)}$ 。

之后我们推导了在Gaussian distribution下考虑class 1和class 2共用 Σ ，可以得到一个线性的 z （很多其他的Probability model经过化简以后也都可以得到同样的结果）

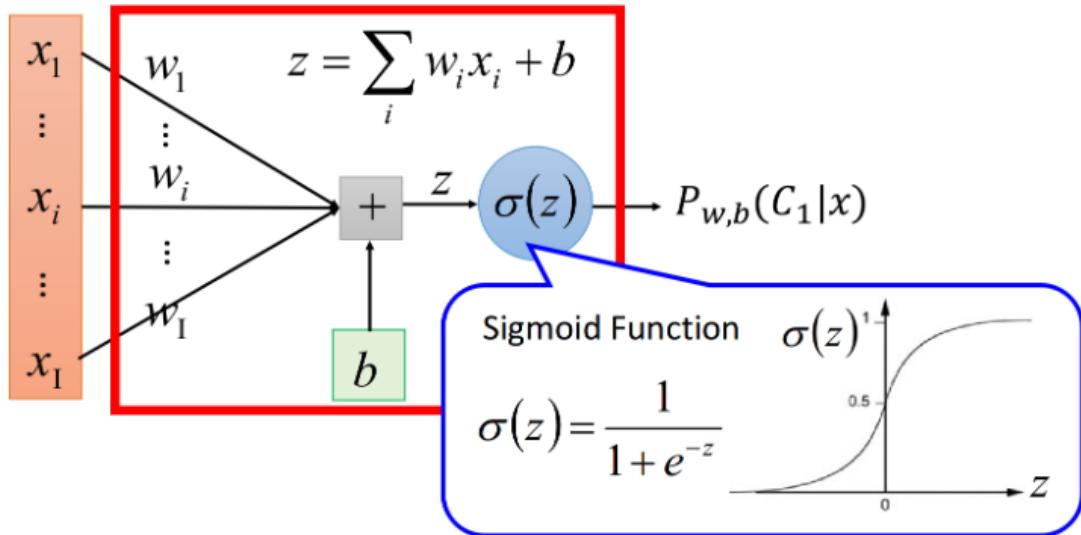
$$\begin{aligned} P_{w,b}(C_1|x) &= \sigma(z) = \frac{1}{1+e^{-z}} \\ z &= w \cdot x + b = \sum_i w_i x_i + b \end{aligned}$$

这里的w和x都是vector，两者的乘积是inner product，从上式中我们可以看出，现在这个model (function set) 是受w和b控制的，因此我们不必要再去像前面一样计算一大堆东西，而是用这个全新的由w和b决定的model——Logistic Regression

因此Function Set为： $f_{w,b}(x) = P_{w,b}(C_1|x) = \sigma(\sum_i w_i x_i + b)$

w_i : weight, b : bias, $\sigma(z)$: sigmoid function, x_i : input

Step 1: Function Set



Step 2: Goodness of a Function

现在我们有N笔Training data，每一笔data都要标注它是属于哪一个class

假设这些Training data是从我们定义的posterior Probability中产生的，而w和b就决定了这个posterior Probability，那我们就可以去计算某一组w和b去产生这N笔Training data的概率，利用极大似然估计的思想，最好的那组参数就是有最大可能性产生当前N笔Training data分布的 w^* 和 b^*

似然函数只需要将每一个点产生的概率相乘即可，注意，这里假定是二元分类，class 2的概率为1减去class 1的概率

$$L(w, b) = f_{w,b}(x^1) f_{w,b}(x^2) (1 - f_{w,b}(x^3)) \cdots f_{w,b}(x^N)$$

由于 $L(w, b)$ 是乘积项的形式，为了方便计算，我们将上式做个变换：

$$\begin{aligned} w^*, b^* &= \arg \max_{w,b} L(w, b) = \arg \min_{w,b} (-\ln L(w, b)) \\ -\ln L(w, b) &= -\ln f_{w,b}(x^1) \\ &\quad -\ln f_{w,b}(x^2) \\ &\quad -\ln(1 - f_{w,b}(x^3)) \\ &\quad \dots \end{aligned}$$

为了统一格式，这里将Logistic Regression里的所有Training data都打上0和1的标签，即output $\hat{y} = 1$ 代表class 1，output $\hat{y} = 0$ 代表class 2，于是上式进一步改写成：

$$\begin{aligned} -\ln L(w, b) &= -[\hat{y}^1 \ln f_{w,b}(x^1) + (1 - \hat{y}^1) \ln(1 - f_{w,b}(x^1))] \\ &\quad -[\hat{y}^2 \ln f_{w,b}(x^2) + (1 - \hat{y}^2) \ln(1 - f_{w,b}(x^2))] \\ &\quad -[\hat{y}^3 \ln f_{w,b}(x^3) + (1 - \hat{y}^3) \ln(1 - f_{w,b}(x^3))] \\ &\quad \dots \end{aligned}$$

现在已经有了统一的格式，我们就可以把要minimize的对象写成一个summation的形式：

$$-\ln L(w, b) = \sum_n -[\hat{y}^n \ln f_{w,b}(x^n) + (1 - \hat{y}^n) \ln(1 - f_{w,b}(x^n))]$$

这里 x^n 表示第n个样本点， \hat{y}^n 表示第n个样本点的class标签（1表示class 1, 0表示class 2），最终这个summation的形式，里面其实是两个Bernoulli distribution的cross entropy。

$$\begin{aligned} p(x=1) &= \hat{y}^n & q(x=1) &= f(x^n) \\ p(x=0) &= 1 - \hat{y}^n & q(x=0) &= 1 - f(x^n) \end{aligned}$$

假设有如上两个distribution p和q，它们的交叉熵就是 $H(p, q) = -\sum_x p(x) \ln(q(x))$

cross entropy的含义是表达这两个distribution有多接近，如果p和q这两个distribution一模一样的话，那它们算出来的cross entropy就是0，而这里 $f(x^n)$ 表示function的output， \hat{y}^n 表示预期的target，因此交叉熵实际上表达的是希望这个function的output和它的target越接近越好

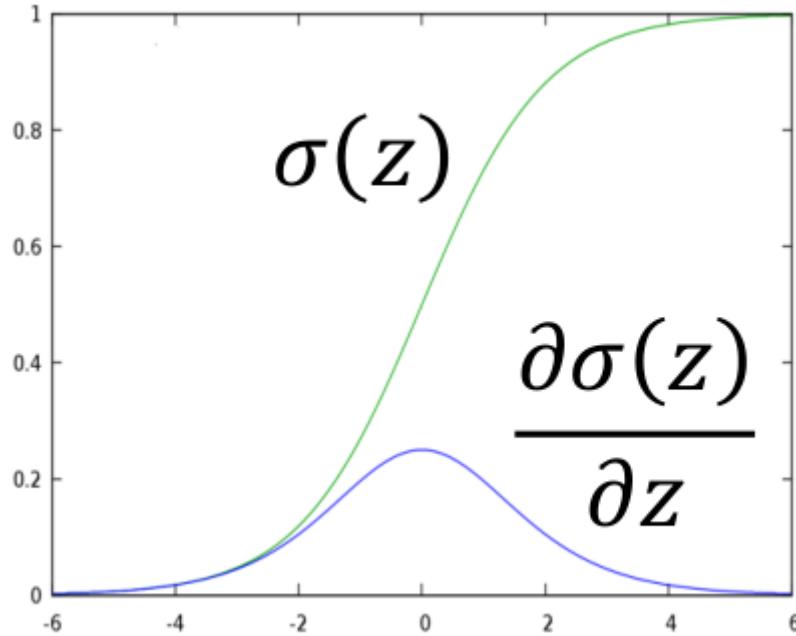
总之，我们要找的参数实际上就是：

$$w^*, b^* = \arg \max_{w,b} L(w, b) = \arg \min_{w,b} (-\ln L(w, b)) = \sum_n -[\hat{y}^n \ln f_{w,b}(x^n) + (1 - \hat{y}^n) \ln(1 - f_{w,b}(x^n))]$$

Step 3: Find the best function

实际上就是去找到使loss function即交叉熵之和最小的那组参数 w^*, b^* 就行了，这里用gradient descent的方法进行运算就可以

sigmoid function的微分: $\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$



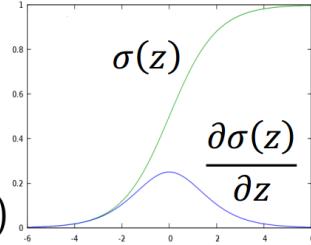
先计算 $-\ln L(w, b) = \sum_n -[\hat{y}^n \ln f_{w,b}(x^n) + (1 - \hat{y}^n) \ln(1 - f_{w,b}(x^n))]$ 对 w_i 的偏微分，这里 \hat{y}^n 和 $1 - \hat{y}^n$ 是常数先不用管它，只需要分别求出 $\ln f_{w,b}(x^n)$ 和 $\ln(1 - f_{w,b}(x^n))$ 对 w_i 的偏微分即可，整体推导过程如下：

Step 3: Find the best function

$$\frac{\partial \ln L(w, b)}{\partial w_i} = \sum_n - \left[\hat{y}^n \frac{\partial \ln f_{w,b}(x^n)}{\partial w_i} + (1 - \hat{y}^n) \frac{\partial \ln(1 - f_{w,b}(x^n))}{\partial w_i} \right]$$

$$\frac{\partial \ln f_{w,b}(x)}{\partial w_i} = \frac{\partial \ln f_{w,b}(x)}{\partial z} \frac{\partial z}{\partial w_i}$$

$$\frac{\partial \ln \sigma(z)}{\partial z} = \frac{1}{\sigma(z)} \frac{\partial \sigma(z)}{\partial z} = \frac{1}{\sigma(z)} \sigma(z)(1 - \sigma(z))$$



$$\frac{\partial \ln(1 - f_{w,b}(x))}{\partial w_i} = \frac{\partial \ln(1 - f_{w,b}(x))}{\partial z} \frac{\partial z}{\partial w_i}$$

$$\frac{\partial \ln(1 - \sigma(z))}{\partial z} = -\frac{1}{1 - \sigma(z)} \frac{\partial \sigma(z)}{\partial z} = -\frac{1}{1 - \sigma(z)} \sigma(z)(1 - \sigma(z))$$

$$f_{w,b}(x) = \sigma(z)$$

$$= 1/(1 + \exp(-z))$$

$$z = w \cdot x + b = \sum_i w_i x_i + b$$

将得到的式子进行进一步化简，可得：

$$\begin{aligned} \frac{\partial \ln L(w, b)}{\partial w_i} &= \sum_n - \left[\hat{y}^n \frac{\partial \ln f_{w,b}(x^n)}{\partial w_i} + (1 - \hat{y}^n) \frac{\partial \ln(1 - f_{w,b}(x^n))}{\partial w_i} \right] \\ &= \sum_n - \left[\hat{y}^n \underbrace{(1 - f_{w,b}(x^n))}_{\text{Larger difference, larger update}} x_i^n - (1 - \hat{y}^n) \underbrace{f_{w,b}(x^n)}_{\text{Larger difference, larger update}} x_i^n \right] \\ &= \sum_n - [\hat{y}^n - f_{w,b}(x^n)] x_i^n \\ &\quad w_i \leftarrow w_i - \eta \sum_n - (\hat{y}^n - f_{w,b}(x^n)) x_i^n \end{aligned}$$

我们发现最终的结果竟然异常的简洁，gradient descent每次update只需要做：

$$w_i = w_i - \eta \sum_n -(\hat{y}^n - f_{w,b}(x^n))x_i^n$$

$$b = b - \eta \sum_n -(\hat{y}^n - f_{w,b}(x^n))$$

那这个式子到底代表着什么意思呢？现在你的update取决于三件事：

- learning rate, 是你自己设定的
- x_i , 来自于data
- $\hat{y}^n - f_{w,b}(x^n)$, 代表function的output跟理想target的差距有多大, 如果离目标越远, update的步伐就要越大

Logistic Regression v.s. Linear Regression

我们可以把逻辑回归和之前讲的线性回归做一个比较

Compare In Step 1

Logistic Regression是把每一个feature x_i 加权求和, 加上bias, 再通过sigmoid function, 当做function的output

因为Logistic Regression的output是通过sigmoid function产生的, 因此一定是介于0~1之间; 而Linear Regression的output并没有通过sigmoid function, 所以它可以是任何值

Compare In Step 2

在Logistic Regression中, 我们定义的loss function, 即要去minimize的对象, 是所有example的output($f(x^n)$)和实际target(\hat{y}^n)在Bernoulli distribution下的cross entropy总和

而在Linear Regression中, loss function的定义相对比较简单, 就是单纯的function的output($f(x^n)$)和实际target(\hat{y}^n)在数值上的平方和的均值

这里可能会有一个疑惑, 为什么Logistic Regression的loss function不能像linear Regression一样用square error来表示呢? 后面会有进一步的解释

Compare In Step 3

神奇的是, Logistic Regression和Linear Regression的 w_i update的方式是一模一样的

<u>Logistic Regression</u>	<u>Linear Regression</u>
Step 1: $f_{w,b}(x) = \sigma\left(\sum_i w_i x_i + b\right)$ Output: between 0 and 1	$f_{w,b}(x) = \sum_i w_i x_i + b$ Output: any value
Training data: (x^n, \hat{y}^n) Step 2: \hat{y}^n : 1 for class 1, 0 for class 2 $L(f) = \sum_n l(f(x^n), \hat{y}^n)$	Training data: (x^n, \hat{y}^n) \hat{y}^n : a real number $L(f) = \frac{1}{2} \sum_n (f(x^n) - \hat{y}^n)^2$
Logistic regression: $w_i \leftarrow w_i - \eta \sum_n - (\hat{y}^n - f_{w,b}(x^n)) x_i^n$	
Step 3: Linear regression: $w_i \leftarrow w_i - \eta \sum_n - (\hat{y}^n - f_{w,b}(x^n)) x_i^n$	
Cross entropy: $l(f(x^n), \hat{y}^n) = - [\hat{y}^n \ln f(x^n) + (1 - \hat{y}^n) \ln(1 - f(x^n))]$	

Logistic Regression + Square Error?

之前提到了，为什么Logistic Regression的loss function不能用square error来描述呢？

Logistic Regression + Square Error

Step 1: $f_{w,b}(x) = \sigma\left(\sum_i w_i x_i + b\right)$

Step 2: Training data: (x^n, \hat{y}^n) , \hat{y}^n : 1 for class 1, 0 for class 2

$$L(f) = \frac{1}{2} \sum_n (f_{w,b}(x^n) - \hat{y}^n)^2$$

Step 3:

$$\frac{\partial (f_{w,b}(x) - \hat{y})^2}{\partial w_i} = 2(f_{w,b}(x) - \hat{y}) \frac{\partial f_{w,b}(x)}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= 2(f_{w,b}(x) - \hat{y}) f_{w,b}(x)(1 - f_{w,b}(x)) x_i$$

$\hat{y}^n = 1$ If $f_{w,b}(x^n) = 1$ (close to target) $\rightarrow \partial L / \partial w_i = 0$

If $f_{w,b}(x^n) = 0$ (far from target) $\rightarrow \partial L / \partial w_i = 0$

$\hat{y}^n = 0$ If $f_{w,b}(x^n) = 1$ (far from target) $\rightarrow \partial L / \partial w_i = 0$

If $f_{w,b}(x^n) = 0$ (close to target) $\rightarrow \partial L / \partial w_i = 0$

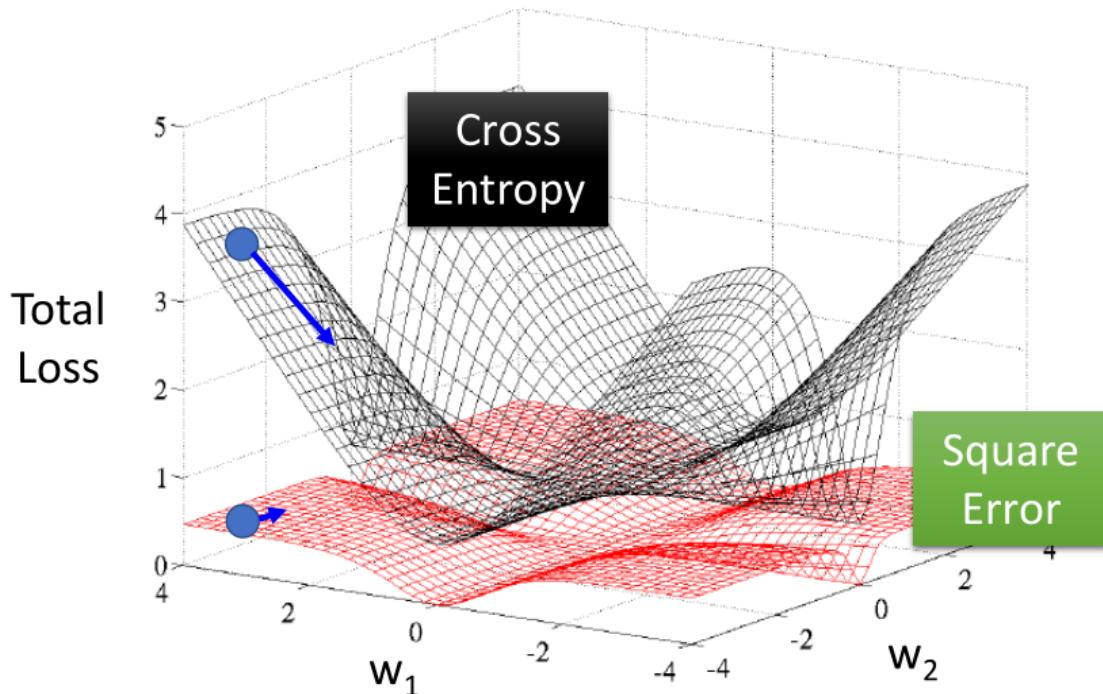
现在会遇到一个问题：

如果第n个点的目标target是class 1, $\hat{y}^n = 1$, 此时如果function的output $f_{w,b}(x^n) = 1$ 的话, 得到的微分 $\frac{\partial L}{\partial w_i}$ 为0; 但是当function的output $f_{w,b}(x^n) = 0$ 的时候, 微分 $\frac{\partial L}{\partial w_i}$ 也是0

如果举class 2的例子, 得到的结果与class 1是一样的

Cross Entropy v.s. Square Error

如果我们把参数的变化对total loss作图的话, loss function选择cross entropy或square error, 参数的变化跟loss的变化情况可视化出来如下所示:



假设中心点就是距离目标很近的地方, 如果是cross entropy的话, 距离目标越远, 微分值就越大, 参数update的时候变化量就越大, 迈出去的步伐也就越大

但当你选择square error的时候, 过程就会很卡, 因为距离目标远的时候, 微分也是非常小的, 移动的速度是非常慢的, 我们之前提到过, 实际操作的时候, 当gradient接近于0的时候, 其实就很有可能会停下来, 因此使用square error很有可能在一开始的时候就卡住不动了, 而且这里也不能随意地增大learning rate, 因为在做gradient descent的时候, 你的gradient接近于0, 有可能离target很近也有可能很远, 因此不知道learning rate应该设大还是设小

综上, 尽管square error可以使用, 但是会出现update十分缓慢的现象, 而使用cross entropy可以让你的Training更顺利

Discriminative v.s. Generative

Logistic Regression的方法, 我们把它称之为discriminative的方法

而我们用Gaussian来描述posterior Probability这件事, 我们称之为Generative的方法

实际上它们用的model(function set)是一模一样的, 都是 $P(C_1|x) = \sigma(w \cdot x + b)$, 如果是用Logistic Regression的话, 可以用gradient descent的方法直接去把b和w找出来; 如果是用Generative model的话, 我们要先去算 u_1, u_2, Σ^{-1} , 然后算出b和w

你会发现用这两种方法得到的b和w是不同的，尽管我们的function set是同一个，但是由于做了不同的假设，最终从同样的Training data里找出来的参数会是不一样的

这是因为在Logistic Regression里面，我们没有做任何实质性的假设，没有对Probability distribution有任何的描述，我们就是单纯地去找b和w

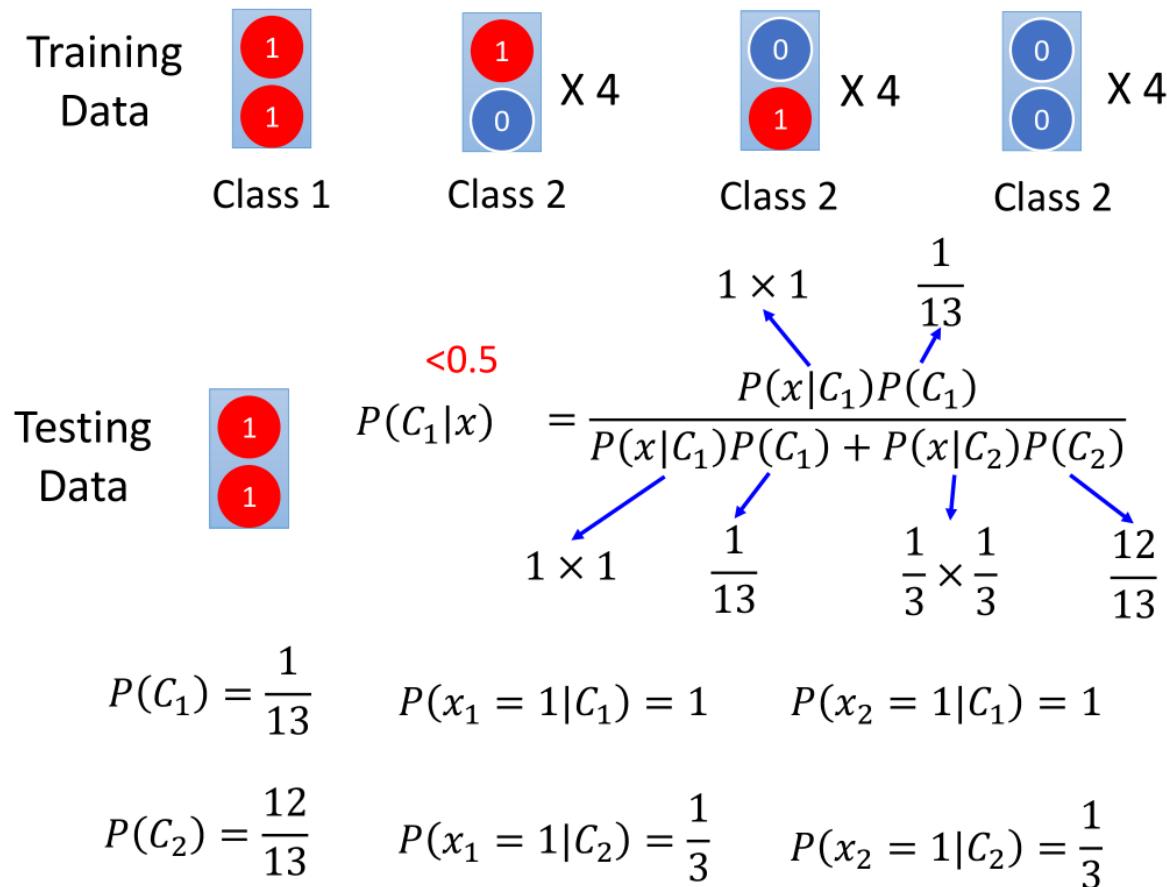
而在Generative model里面，我们对Probability distribution是有实质性的假设的，之前我们假设的是 Gaussian，甚至假设在相互独立的前提下是否可以是Naive Bayes，根据这些假设我们才找到最终的b和w

哪一个假设的结果是比较好的呢？实际上Discriminative的方法常常会比Generative的方法表现得更好，这里举一个简单的例子来解释一下

Example

假设总共有两个class，有这样的Training data：每一笔data有两个feature，总共有 $1+4+4+4=13$ 笔data

如果我们的testing data的两个feature都是1，凭直觉来说会认为它肯定是class 1，但是如果用Naive Bayes的方法(朴素贝叶斯假设所有的feature相互独立，方便计算)，得到的结果又是怎样的呢？



通过Naive Bayes得到的结果竟然是这个测试点属于class 2的可能性更大，这跟我们的直觉比起来是相反的

实际上我们直觉认为两个feature都是1的测试点属于class 1的可能性更大是因为我们潜意识里认为这两个feature之间是存在某种联系的

但是对Naive Bayes来说，它是不考虑不同dimension之间的correlation，Naive Bayes认为在dimension相互独立的前提下，class 2没有sample出都是1的数据，是因为sample的数量不够多，如果sample够多，它认为class 2观察到都是1的数据的可能性会比class 1要大

Naive Bayes认为从class 2中找到样本点x的概率是x中第一个feature出现的概率与第二个feature出现的概率之积： $P(x|C_2) = P(x_1 = 1|C_2) \cdot P(x_2 = 1|C_2)$

但是我们的直觉告诉自己，两个feature之间肯定是有某种联系的， $P(x|C_2)$ 不能够那么轻易地被拆分成两个独立的概率乘积，也就是说Naive Bayes自作聪明地多假设了一些条件

所以，Generative model和discriminative model的差别就在于，Generative的model它有做了某些假设，假设你的data来自于某个概率模型；而Discriminative的model是完全不作任何假设的

通常脑补不是一件好的事情，因为你给你的data强加了一些它并没有告诉你的属性，但是在data很少的情况下，脑补也是有用的，discriminative model并不是在所有的情况下都可以赢过Generative model，discriminative model是十分依赖于data的，当data数量不足或是data本身的label就有一些问题，那Generative model做一些脑补和假设，反而可以把data的不足或是有问题部分的影响给降到最低

在Generative model中，priors probabilities和class-dependent probabilities是可以拆开来考虑的，以语音辨识为例，现在用的都是neural network，是一个discriminative的方法，但事实上整个语音辨识的系统是一个Generative的system，DNN只是其中的一块

它需要算一个prior probability是某一句话被说出来的机率，而想要estimate某一句话被说出来的机率并不需要有声音的data，去互联网上爬取大量文字就可以计算出某一段文字出现的机率，这个就是language model，prior的部分只用文字data来处理，而class-dependent的部分才需要声音和文字的配合，这样的处理可以把prior estimate更精确

Generative model的好处是，它对data的依赖并没有像discriminative model那么严重，在data数量少或者data本身就存在noise的情况下受到的影响会更小，而它还可以做到Prior部分与class-dependent部分分开处理，如果可以借助其他方式提高Prior model的准确率，对整一个model是有所帮助的

而Discriminative model的好处是，在data充足的情况下，它训练出来的model的准确率一般是比Generative model要来的高的

Benefit of generative model

- With the assumption of probability distribution, less training data is needed
- With the assumption of probability distribution, more robust to the noise
- Priors and class-dependent probabilities can be estimated from different sources.

Multi-class Classification

Softmax

之前讲的都是二元分类的情况，这里讨论一下多元分类问题，其原理的推导过程与二元分类基本一致

假设有三个class： C_1, C_2, C_3 ，每一个class都有自己的weight和bias，这里 w_1, w_2, w_3 分别代表三个vector， b_1, b_2, b_3 分别代表三个const，input x 也是一个vector

softmax的意思是对最大值做强化，因为在做第一步的时候，对 z 取exponential会使大的值和小的值之间的差距被拉得更开，也就是强化大的值

我们把 z_1, z_2, z_3 丢进一个softmax的function，softmax做的事情是这样三步：

- 取exponential，得到 $e^{z_1}, e^{z_2}, e^{z_3}$
- 把三个exponential累计求和，得到total sum = $\sum_{j=1}^3 e^{z_j}$
- 将total sum分别除去这三项(归一化)，得到 $y_1 = \frac{e^{z_1}}{\sum_{j=1}^3 e^{z_j}}$ 、 $y_2 = \frac{e^{z_2}}{\sum_{j=1}^3 e^{z_j}}$ 、 $y_3 = \frac{e^{z_3}}{\sum_{j=1}^3 e^{z_j}}$

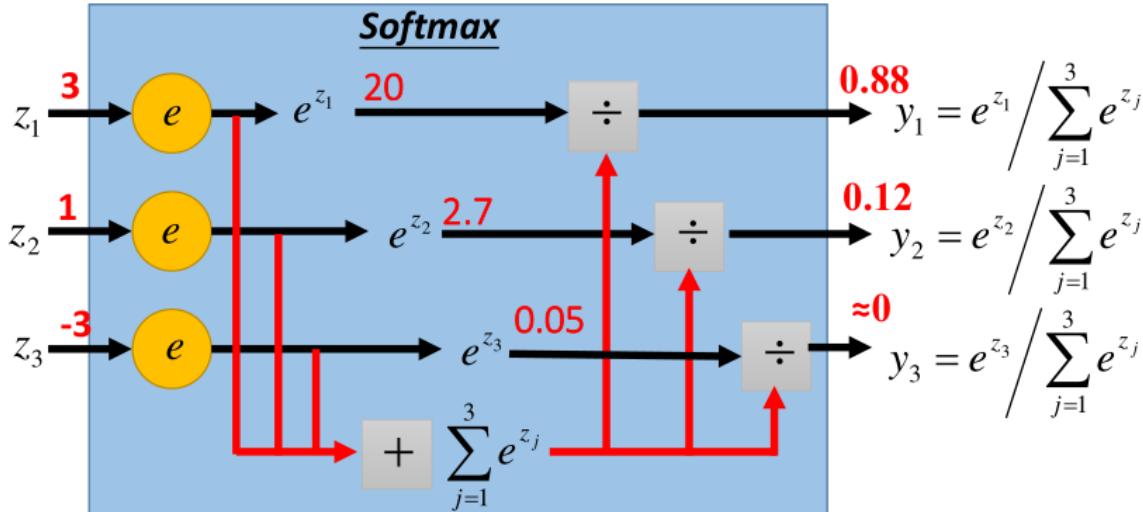
Multi-class Classification (3 classes as example)

$$\begin{array}{ll} C_1: w^1, b_1 & z_1 = w^1 \cdot x + b_1 \\ C_2: w^2, b_2 & z_2 = w^2 \cdot x + b_2 \\ C_3: w^3, b_3 & z_3 = w^3 \cdot x + b_3 \end{array}$$

Probability:

- $1 > y_i > 0$
- $\sum_i y_i = 1$

$$y_i = P(C_i | x)$$



原来的output z 可以是任何值，但是做完softmax之后，你的output y_i 的值一定是介于0~1之间，并且它们的和一定是1， $\sum_i y_i = 1$ ，以上图为例， y_i 表示input x 属于第*i*个class的概率，比如属于Class 1的概率是 $y_1 = 0.88$ ，属于Class 2的概率是 $y_2 = 0.12$ ，属于Class 3的概率是 $y_3 = 0$

而softmax的output，就是拿来当z的posterior probability

假设我们用的是Gaussian distribution (共用covariance)，经过一般推导以后可以得到softmax的function

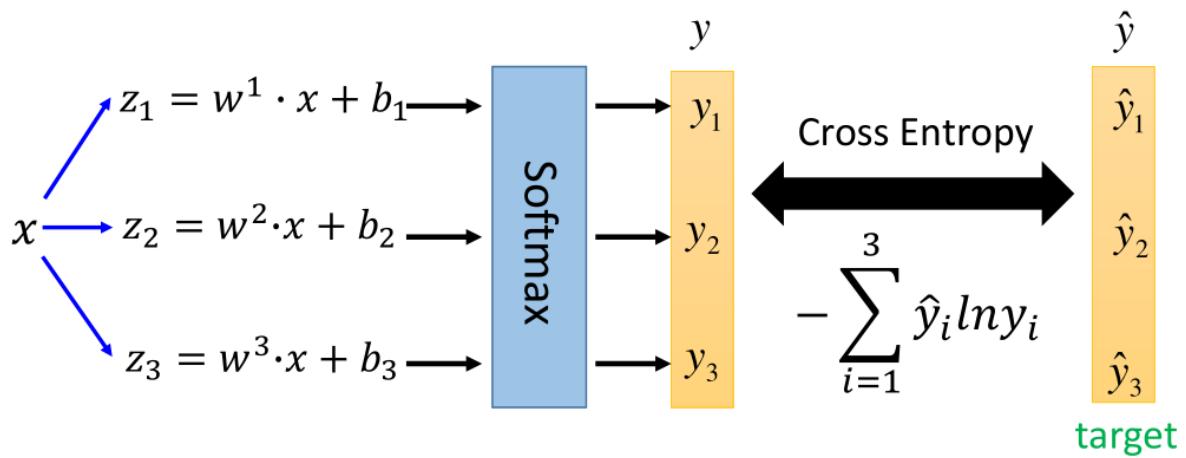
同样从information theory也可以推导出softmax function，Maximum entropy本质内容和Logistic Regression是一样的，它是从另一个观点来切入为什么我们的classifier长这样子

Multi-class Classification

如下图所示，input x 经过三个式子分别生成 z_1, z_2, z_3 ，经过softmax转化成output y_1, y_2, y_3 分别是这三个class的posterior probability，由于 $\text{summation}=1$ ，因此做完softmax之后就可以把y的分布当做是一个probability contribution

我们在训练的时候还需要有一个target，因为是三个class，output是三维的，对应的target也是三维的，为了满足交叉熵的条件，target \hat{y} 也必须是probability distribution，这里我们不能使用1,2,3作为class的区分，为了保证所有class之间的关系是一样的，这里使用类似于one-hot编码的方式，即

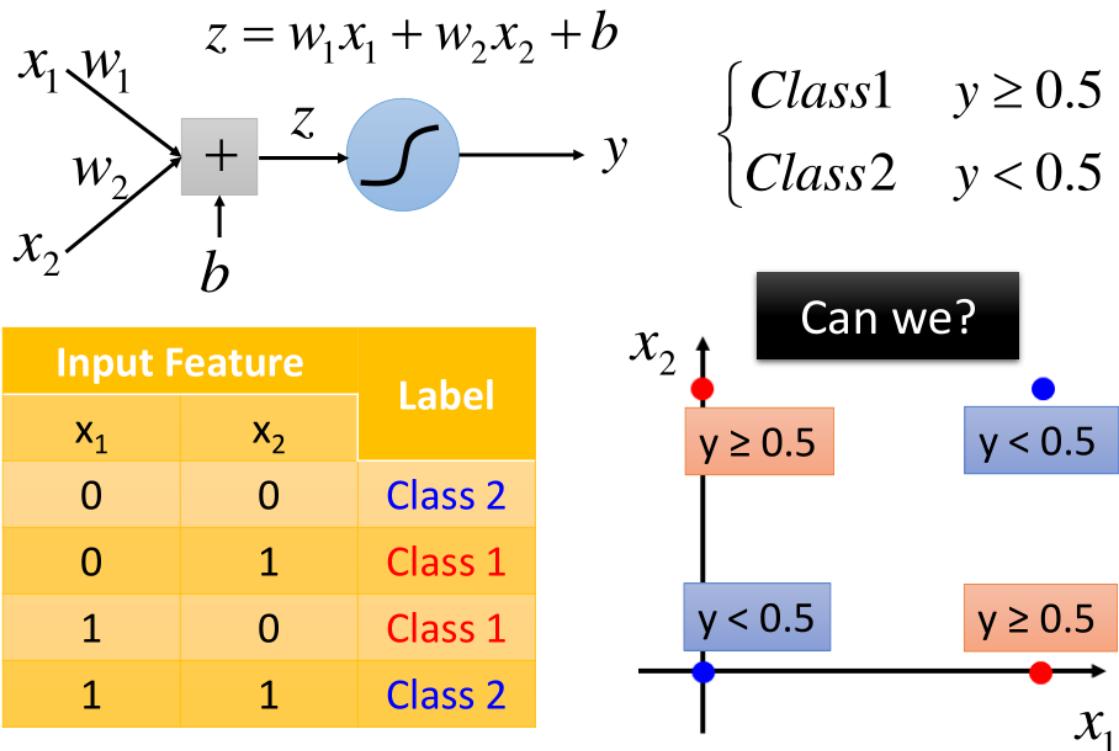
$$\hat{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}_{x \in \text{class1}} \quad \hat{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}_{x \in \text{class2}} \quad \hat{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}_{x \in \text{class3}}$$



这个时候就可以计算output y 和target \hat{y} 之间的交叉熵，即 $-\sum_{i=1}^3 \hat{y}_i \ln y_i$ ，同二元分类一样，多元分类问题也是通过极大似然估计法得到最终的交叉熵表达式的，这里不再赘述

Limitation of Logistic Regression

Logistic Regression其实有很强的限制，给出下图的例子中的Training data，想要用Logistic Regression对它进行分类，其实是做不到的



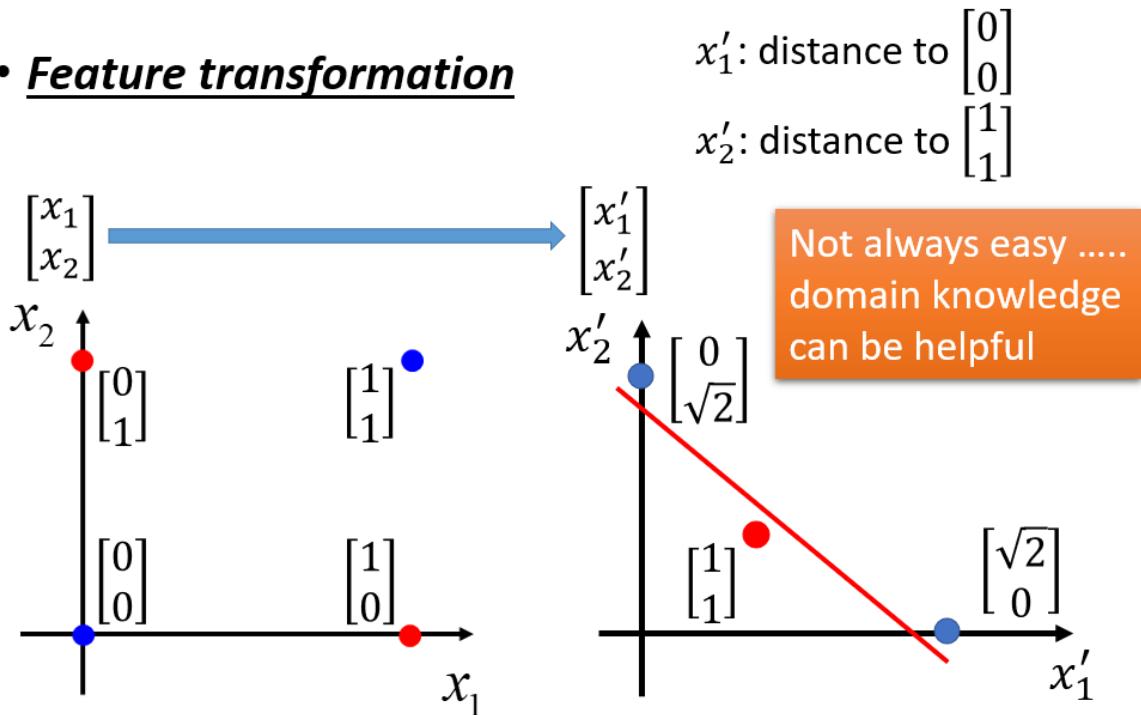
因为Logistic Regression在两个class之间的boundary就是一条直线，但是在这个平面上无论怎么画直线都不可能把图中的两个class分隔开来

Feature Transformation

如果坚持要用Logistic Regression的话，有一招叫做Feature Transformation，原来的feature分布不好划分，那我们可以将之转化以后，找一个比较好的feature space，让Logistic Regression能够处理

假设这里定义 x'_1 是原来的点到 $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ 之间的距离， x'_2 是原来的点到 $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ 之间的距离，重新映射之后如下图右侧(红色两个点重合)，此时Logistic Regression就可以把它们划分开来

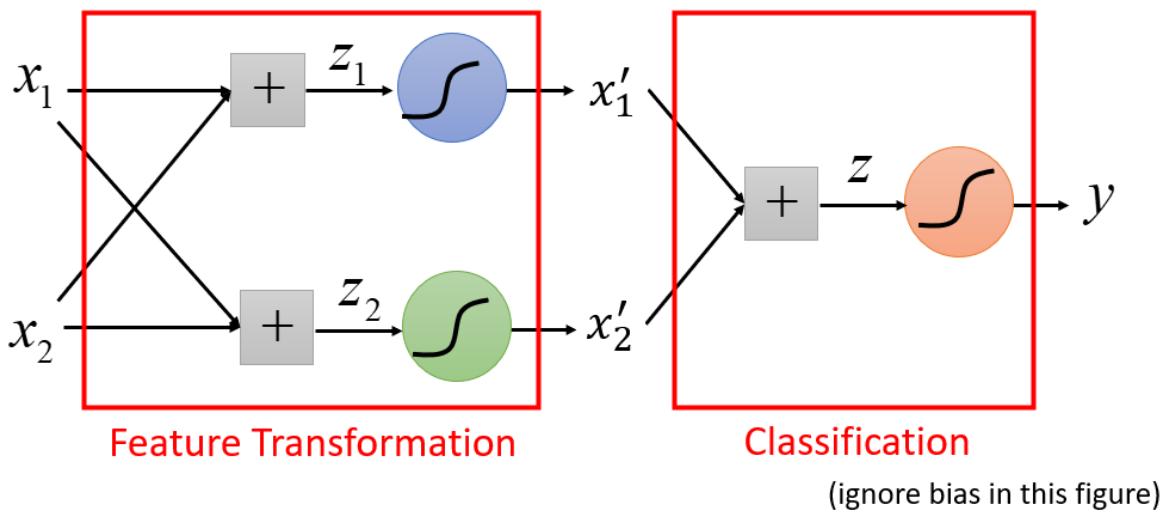
- ***Feature transformation***



但麻烦的是，我们并不知道怎么做feature Transformation，如果在这上面花费太多的时间就得不偿失了，于是我们会希望这个Transformation是机器自己产生的，怎么让机器自己产生呢？我们可以让很多 Logistic Regression cascade(连接)起来

我们让一个input x 的两个feature x_1, x_2 经过两个Logistic Regression的transform，得到新的feature x'_1, x'_2 ，在这个新的feature space上，class 1和class 2是可以用一条直线分开的，那么最后只要再接另外一个Logistic Regression的model（对它来说， x'_1, x'_2 才是每一个样本点的feature，而不是原先的 x_1, x_2 ），它根据新的feature，就可以把class 1和class 2分开

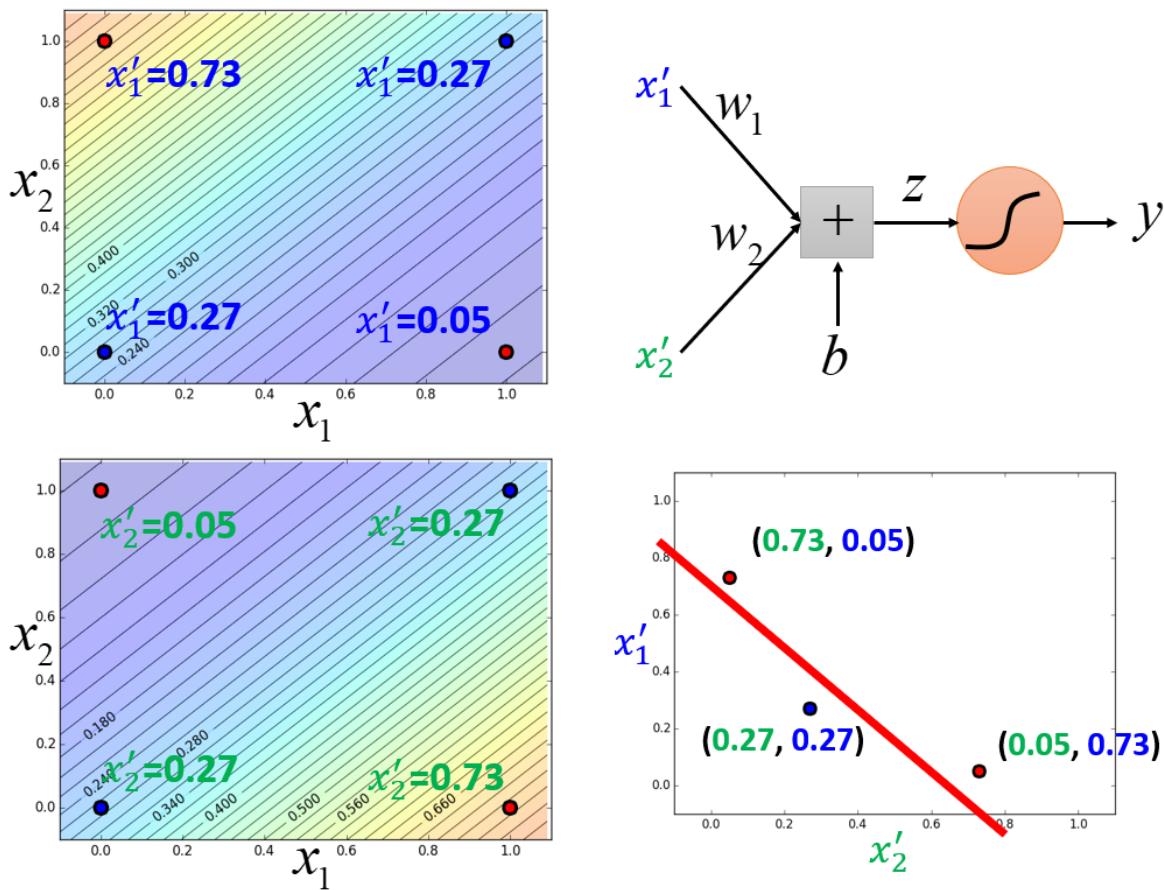
- **Cascading logistic regression models**



因此整个流程是，先用n个Logistic Regression做Feature Transformation（n为每个样本点的feature数量），生成n个新的feature，然后再用一个Logistic Regression作classifier

Logistic Regression的boundary一定是一条直线，具体的分布是由Logistic Regression的参数决定的，直线是由 $b + \sum_i^n w_i x_i = 0$ 决定的（二维feature的直线画在二维平面上，多维feature的直线则是画在多维空间上）

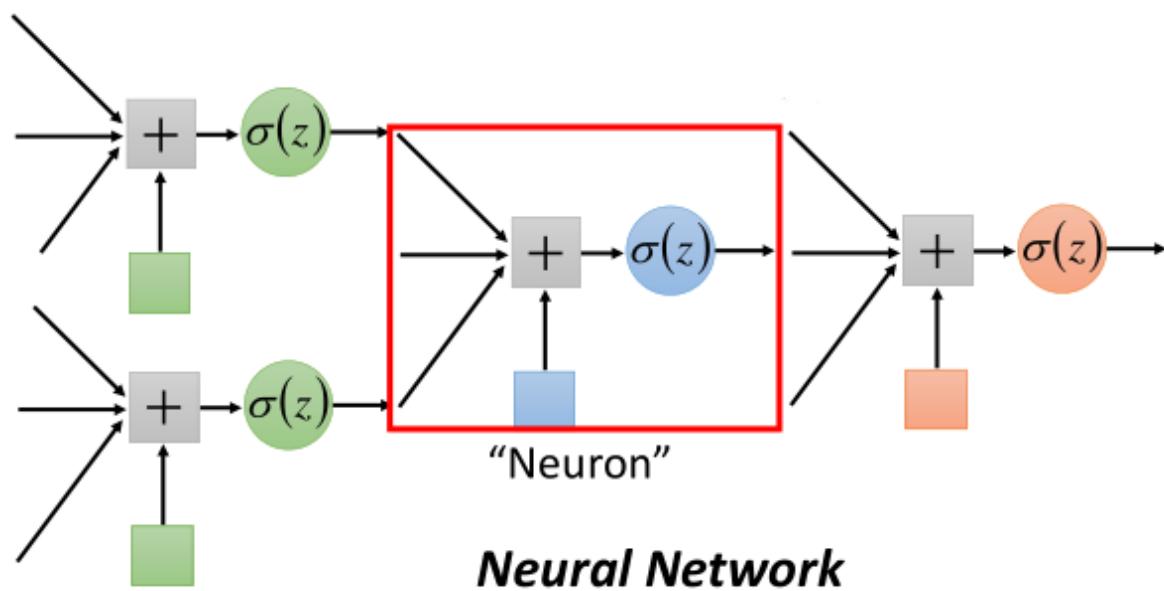
下图是二维feature的例子，分别表示四个点经过transform之后的 x'_1 和 x'_2 ，在新的feature space中可以通过最后的Logistic Regression划分开来



注意，这里的Logistic Regression只是一条直线，它指的是属于这个类或不属于这个类这两种情况，因此最后的这个Logistic Regression是跟要检测的目标类相关的

当只是二元分类的时候，最后只需要一个Logistic Regression即可，当面对多元分类问题，需要用到多个Logistic Regression来画出多条直线划分所有的类，每一个Logistic Regression对应它要检测的那个类

通过上面的例子，我们发现，多个Logistic Regression连接起来会产生powerful的效果，我们把每一个Logistic Regression叫做一个neuron (神经元)，把这些Logistic Regression串起来所形成的network，就叫做Neural Network，这个东西就是Deep Learning。



Support Vector Machine

SVM = Hinge Loss + Kernel Method

Hinge Loss

Binary Classification

先回顾一下二元分类的做法，为了方便后续推导，这里定义data的标签为-1和+1

- 当 $f(x) > 0$ 时， $g(x) = 1$ ，表示属于第一类别；当 $f(x) < 0$ 时， $g(x) = -1$ ，表示属于第二类别
 - 原本用 $\sum \delta(g(x^n) \neq \hat{y}^n)$ ，不匹配的样本点个数，来描述loss function，其中 $\delta = 1$ 表示 x 与 \hat{y} 相匹配，反之 $\delta = 0$ ，但这个式子不可微分，无法使用梯度下降法更新参数
- 因此使用近似的可微分的 $l(f(x^n), \hat{y}^n)$ 来表示损失函数

Binary Classification

x^1	x^2	x^3	...
\hat{y}^1	\hat{y}^2	\hat{y}^3	...

$$\hat{y}^n = +1, -1$$

- Step 1: Function set (Model)

$$g(x) = \begin{cases} f(x) > 0 & \text{Output} = +1 \\ f(x) < 0 & \text{Output} = -1 \end{cases}$$

- Step 2: Loss function:

$$L(f) = \sum_n \delta(g(x^n) \neq \hat{y}^n)$$

The number of times g get incorrect results on training data.

- Step 3: Training by gradient descent is difficult

Gradient descent is possible if $g(*)$ and $\delta(*)$ is differentiable

下图中，横坐标为 $\hat{y}^n f(x)$ ，我们希望横坐标越大越好：

- 当 $\hat{y}^n > 0$ 时，希望 $f(x)$ 越正越好
- 当 $\hat{y}^n < 0$ 时，希望 $f(x)$ 越负越好

纵坐标是loss，原则上，当横坐标 $\hat{y}^n f(x)$ 越大的时候，纵坐标loss要越小，横坐标越小，纵坐标loss要越大

ideal loss

在 $L(f) = \sum_n \delta(g(x^n) \neq \hat{y}^n)$ 的理想情况下，如果 $\hat{y}^n f(x) > 0$ ，则loss=0，如果 $\hat{y}^n f(x) < 0$ ，则loss=1，

如下图中加粗的黑线所示，可以看出该曲线是无法微分的，因此我们要另一条近似的曲线来替代该损失函数

Step 2: Loss function

$$g(x) = \begin{cases} f(x) > 0 & \text{Output} = +1 \\ f(x) < 0 & \text{Output} = -1 \end{cases}$$

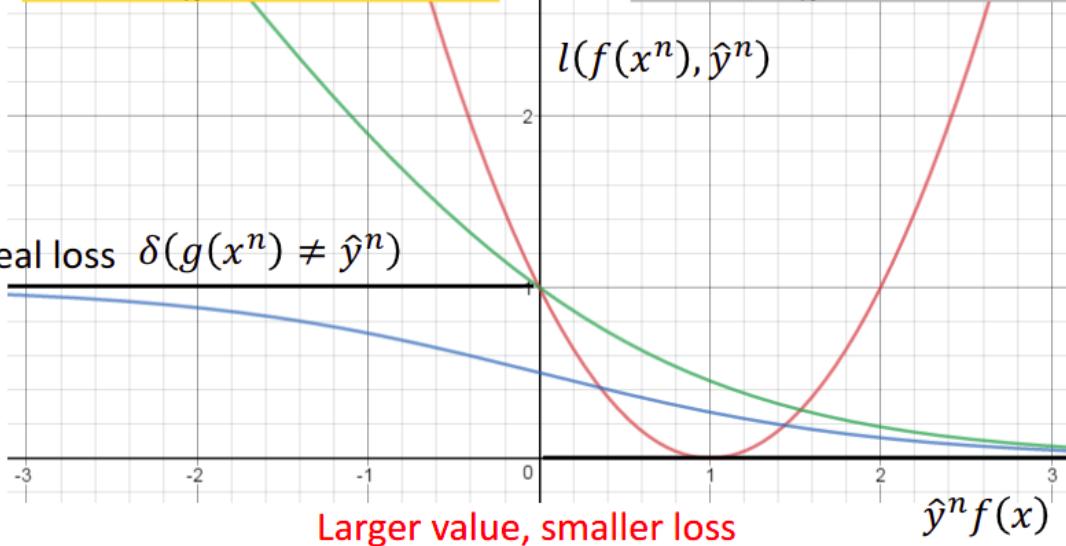
Ideal loss:

$$L(f) = \sum_n \delta(g(x^n) \neq \hat{y}^n)$$

Approximation:

$$L(f) = \sum_n l(f(x^n), \hat{y}^n)$$

Ideal loss $\delta(g(x^n) \neq \hat{y}^n)$



square loss

下图中的红色曲线代表了square loss的损失函数: $l(f(x^n), \hat{y}^n) = (\hat{y}^n f(x^n) - 1)^2$

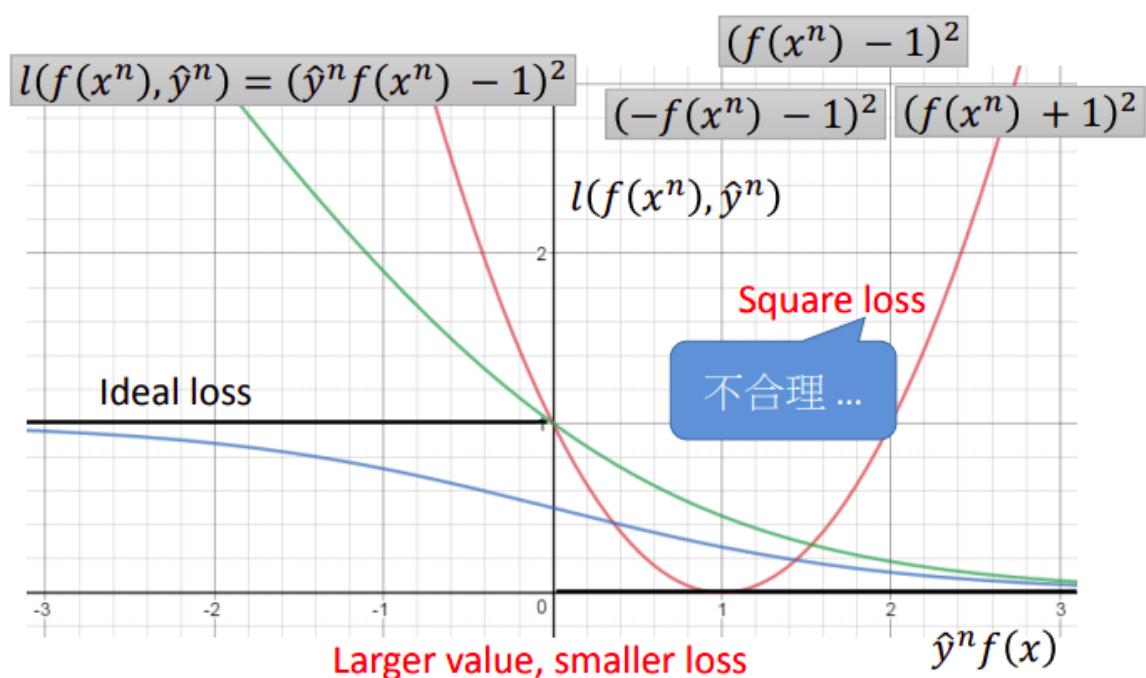
- 当 $\hat{y}^n = 1$ 时, $f(x)$ 与 1 越接近越好, 此时损失函数化简为 $(f(x^n) - 1)^2$
- 当 $\hat{y}^n = -1$ 时, $f(x)$ 与 -1 越接近越好, 此时损失函数化简为 $(f(x^n) + 1)^2$
- 但实际上整条曲线是不合理的, 它会使得 $\hat{y}^n f(x)$ 很大的时候有一个更大的 loss

Step 2: Loss function

Square Loss:

If $\hat{y}^n = 1$, $f(x)$ close to 1

If $\hat{y}^n = -1$, $f(x)$ close to -1



sigmoid + square loss

此外蓝线代表sigmoid+square loss的损失函数: $l(f(x^n), \hat{y}^n) = (\sigma(\hat{y}^n f(x^n)) - 1)^2$

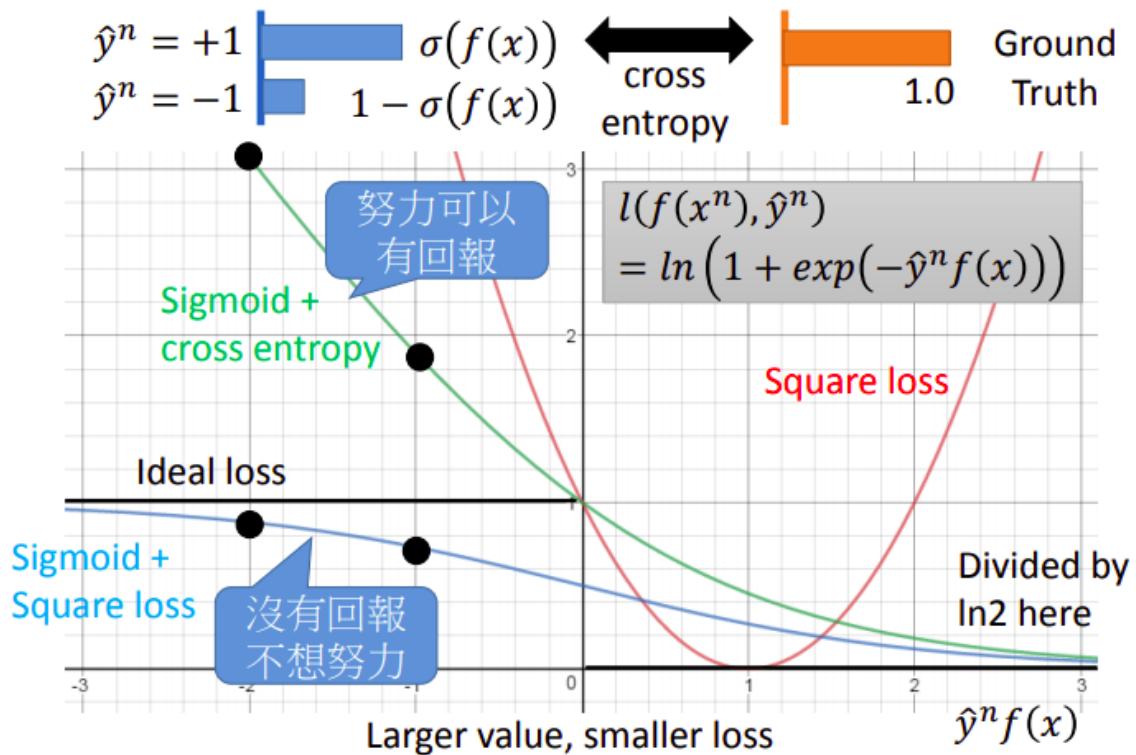
- 当 $\hat{y}^n = 1$ 时, $\sigma(f(x))$ 与1越接近越好, 此时损失函数化简为 $(\sigma(f(x)) - 1)^2$
- 当 $\hat{y}^n = -1$ 时, $\sigma(f(x))$ 与0越接近越好, 此时损失函数化简为 $(\sigma(f(x)))^2$
- 在逻辑回归的时候实践过, 一般square loss的方法表现并不好, 而是用cross entropy会更好

sigmoid + cross entropy

绿线则是代表了sigmoid+cross entropy的损失函数: $l(f(x^n), \hat{y}^n) = \ln(1 + e^{-\hat{y}^n f(x)})$

- $\sigma(f(x))$ 代表了一个分布, 而Ground Truth则是真实分布, 这两个分布之间的交叉熵, 就是我们要去 minimize的loss
- 当 $\hat{y}^n f(x)$ 很大的时候, loss接近于0
- 当 $\hat{y}^n f(x)$ 很小的时候, loss特别大
- 下图是把损失函数除以 $\ln 2$ 的曲线, 使之变成ideal loss的upper bound, 且不会对损失函数本身产生影响
- 我们虽然不能minimize理想的loss曲线, 但我们可以minimize它的upper bound, 从而起到最小化 loss的效果

Step 2: Loss function Sigmoid + cross entropy (logistic regression)



cross entropy v.s. square error

为什么cross entropy要比square error要来的有效呢?

- 我们期望在极端情况下, 比如 \hat{y}^n 与 $f(x)$ 非常不匹配导致横坐标非常负的时候, loss的梯度要很大, 这样才能尽快地通过参数调整回到loss低的地方
- 对sigmoid+square loss来说, 当横坐标非常负的时候, loss的曲线反而是平缓的, 此时去调整参数值对最终loss的影响其实并不大, 它并不能很快地降低

形象来说就是, “没有回报, 不想努力”

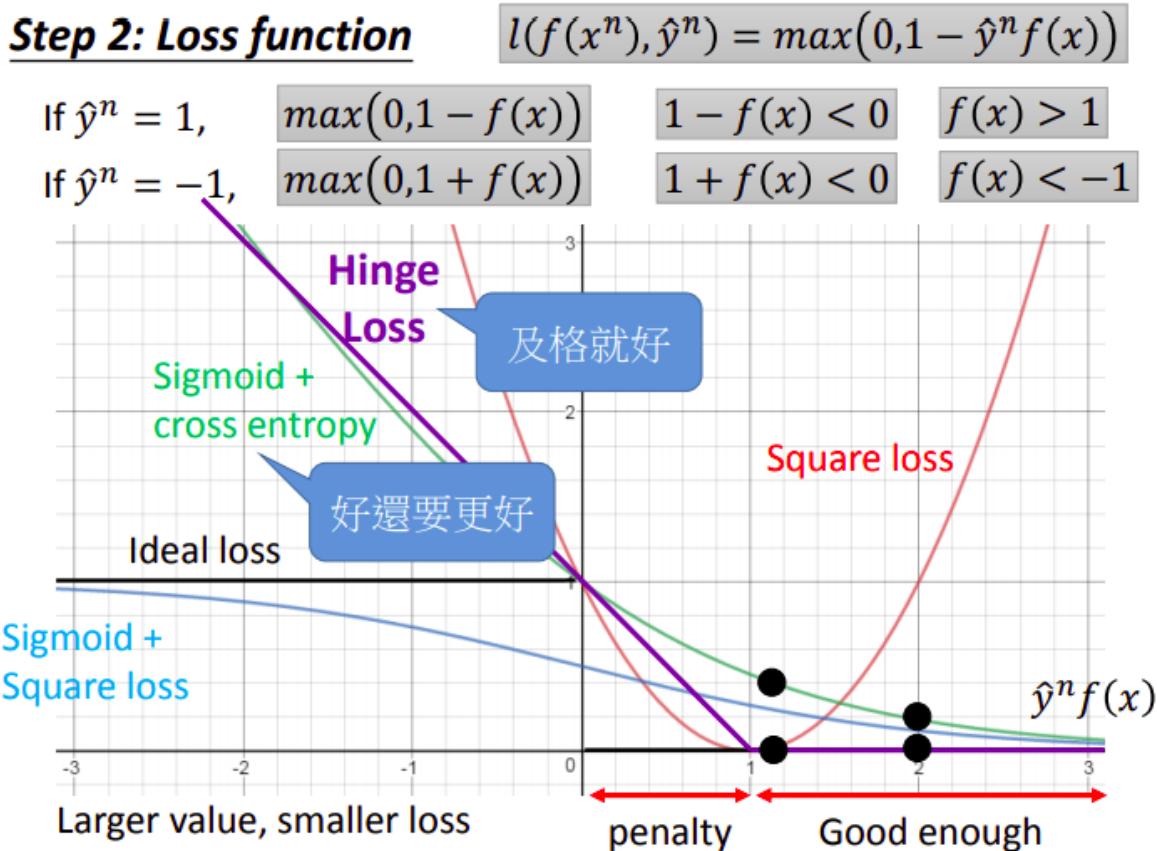
- 而对cross entropy来说，当横坐标非常负的时候，loss的梯度很大，稍微调整参数就可以往loss小的地方走很大一段距离，这对训练是友好的

形象来说就是，“努力可以有回报”

Hinge Loss

紫线代表了hinge loss的损失函数： $l(f(x^n), \hat{y}^n) = \max(0, 1 - \hat{y}^n f(x))$

- 当 $\hat{y}^n = 1$ ，损失函数化简为 $\max(0, 1 - f(x))$
 - 此时只要 $f(x) > 1$, loss就会等于0
- 当 $\hat{y}^n = -1$ ，损失函数化简为 $\max(0, 1 + f(x))$
 - 此时只要 $f(x) < -1$, loss就会等于0
- 总结一下，如果label为1，则当 $f(x) > 1$ ，机器就认为loss为0；如果label为-1，则当 $f(x) < -1$ ，机器就认为loss为0，因此该函数并不需要 $f(x)$ 有一个很大的值



在紫线中，当 $\hat{y}^n f(x) > 1$ ，则已经实现目标，loss=0；当 $\hat{y}^n f(x) > 0$ ，表示已经得到了正确答案，但Hinge Loss认为这还不够，它需要你继续往1的地方前进

事实上，Hinge Loss也是Ideal loss的upper bound，但是当横坐标 $\hat{y}^n f(x) > 1$ 时，它与Ideal loss近乎是完全贴近的

比较Hinge loss和cross entropy，最大的区别在于他们对待已经做得好的样本点的态度，在横坐标 $\hat{y}^n f(x) > 1$ 的区间上，cross entropy还想要往更大的地方走，而Hinge loss则已经停下来了，就像一个的目标是“还想要更好”，另一个的目标是“及格就好”

在实作上，两者差距并不大，而Hinge loss的优势在于它不怕outliers，训练出来的结果鲁棒性(robust)比较强

Linear SVM

model description

在线性的SVM里，我们把 $f(x) = \sum_i w_i x_i + b = w^T x$ 看做是向量 $\begin{bmatrix} w \\ b \end{bmatrix}$ 和向量 $\begin{bmatrix} x \\ 1 \end{bmatrix}$ 的内积，也就是新的 w 和 x ，这么做可以把bias项省略掉

在损失函数中，我们通常会加上一个正规项，即 $L(f) = \sum_n l(f(x^n), \hat{y}^n) + \lambda \|w\|_2$

这是一个convex的损失函数，好处在于无论从哪个地方开始做梯度下降，最终得到的结果都会在最低处，曲线中一些折角处等不可微的点可以参考NN中relu、maxout等函数的微分处理

Linear SVM

Compared with logistic regression,
linear SVM has different loss function

Deep version: Yichuan Tang , "Deep
Learning using Linear Support
Vector Machines", ICML 2013
Challenges in Representation
Learning Workshop

- Step 1: Function (Model)

$$f(x) = \sum_i w_i x_i + b = \begin{bmatrix} w \\ b \end{bmatrix} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix} = w^T x$$

- Step 2: Loss function
- convex
- $$L(f) = \sum_n l(f(x^n), \hat{y}^n) + \lambda \|w\|_2$$
- $$l(f(x^n), \hat{y}^n) = \max(0, 1 - \hat{y}^n f(x))$$

- Step 3: gradient descent?

Recall relu, maxout network

对比Logistic Regression和Linear SVM，两者唯一的区别就是损失函数不同，前者用的是cross entropy，后者用的是Hinge loss

事实上，SVM并不局限于Linear，尽管Linear可以带来很多好的特质，但我们完全可以在一个Deep的神经网络中使用Hinge loss的损失函数，就成为了Deep SVM，其实Deep Learning、SVM这些方法背后的精神都是相通的，并没有那么大的界限

gradient descent

尽管SVM大多不是用梯度下降训练的，但使用该方法训练确实是可行的，推导过程如下：

Linear SVM – gradient descent

Ignore regularization for simplicity

$$L(f) = \sum_n l(f(x^n), \hat{y}^n) \quad l(f(x^n), \hat{y}^n) = \max(0, 1 - \hat{y}^n f(x^n))$$

$$\frac{\partial l(f(x^n), \hat{y}^n)}{\partial w_i} = \frac{\partial l(f(x^n), \hat{y}^n)}{\partial f(x^n)} \boxed{\frac{\partial f(x^n)}{\partial w_i}} x_i^n \quad \boxed{f(x^n)} = w^T \cdot x^n$$

$$\frac{\partial \max(0, 1 - \hat{y}^n f(x^n))}{\partial f(x^n)} = \begin{cases} -\hat{y}^n & \text{if } \hat{y}^n f(x^n) < 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial L(f)}{\partial w_i} = \sum_n \frac{-\delta(\hat{y}^n f(x^n) < 1) \hat{y}^n x_i^n}{c^n(w)} \quad w_i \leftarrow w_i - \eta \sum_n c^n(w) x_i^n$$

another formulation

前面列出的式子可能与你平常看到的SVM不大一样，这里将其做一下简单的转换

$$\text{对 } L(f) = \sum_n \max(0, 1 - \hat{y}^n f(x)) + \lambda \|w\|_2$$

用 $L(f) = \sum_n \epsilon^n + \lambda \|w\|_2$ 来表示，其中 $\epsilon^n = \max(0, 1 - \hat{y}^n f(x^n))$

对 $\epsilon^n \geq 0$ 、 $\epsilon^n \geq 1 - \hat{y}^n f(x)$ 来说，它与上式是不同的，因为 \max 得到的 ϵ^n 是二选一，而 \geq 得到的 ϵ^n 则大多都可以

但是当加上取 loss function $L(f)$ 最小化这个条件时， \geq 就要取到等号，两者就是等价的

Linear SVM – another formulation

Minimizing loss function L:

$$L(f) = \sum_n \varepsilon^n + \lambda \|w\|_2$$

$$\varepsilon^n = \max(0, 1 - \hat{y}^n f(x))$$

||

ε^n : slack variable
Quadratic programming problem

$$\varepsilon^n \geq 0$$

$$\varepsilon^n \geq 1 - \hat{y}^n f(x) \rightarrow \hat{y}^n f(x) \geq 1 - \varepsilon^n$$

此时该表达式就和你熟知的SVM一样了：

$$L(f) = \sum_n \varepsilon^n + \lambda \|w\|_2, \text{ 且 } \hat{y}^n f(x) \geq 1 - \varepsilon^n$$

其中 \hat{y}^n 和 $f(x)$ 要同号， ε^n 要大于等于0，这里 ε^n 的作用就是放宽1的margin，也叫作松弛变量slack variable

这是一个QP问题Quadratic programming problem，可以用对应方法求解，当然前面提到的梯度下降法也可以解

Kernel Method

Linear combination of data points

你要先说服你自己一件事：实际上我们找出来的可以minimize损失函数的参数，其实就是data的线性组合

$$w^* = \sum_n \alpha_n^* x^n$$

你可以通过拉格朗日乘数法去求解前面的式子来验证，这里试图从梯度下降的角度来解释：

观察 w 的更新过程 $w = w - \eta \sum_n c^n(w) x^n$ 可知，如果 w 被初始化为0，则每次更新的时候都是加上data point x 的线性组合，因此最终得到的 w 依旧会是 x 的Linear Combination

而使用Hinge loss的时候， $c^n(w)$ 或者说 α_n^* 往往会被0（如果作用在 $\max=0$ 的区域），SVM解出来的 α_n 是sparse的，因为有很多 x^n 的系数微分为0，这意味着即使从数据集中把这些 x^n 的样本点移除掉，对结果也是没有影响的，这可以增强系统的鲁棒性

不是所有的 x^n 都会被加到 w 里去，而被加到 w 里的那些 x^n ，才是会决定model和parameter样子的数据点，就叫做**support vector**

Dual Representation

$$w^* = \sum_n \alpha_n^* x^n \quad \text{Linear combination of data points}$$

α_n^* may be sparse $\rightarrow x^n$ with non-zero α_n^* are support vectors

$$\left. \begin{array}{l} w_1 \leftarrow w_1 - \eta \sum_n c^n(w) x_1^n \\ \vdots \\ w_i \leftarrow w_i - \eta \sum_n c^n(w) x_i^n \\ \vdots \\ w_k \leftarrow w_k - \eta \sum_n c^n(w) x_k^n \end{array} \right\} \begin{array}{l} \text{If } w \text{ initialized as } \mathbf{0} \\ w \leftarrow w - \eta \sum_n c^n(w) x^n \\ c^n(w) \\ = \frac{\partial l(f(x^n), \hat{y}^n)}{\partial f(x^n)} \quad \text{Hinge loss: usually zero} \\ \text{c.f. for logistic regression, it is always non-zero} \end{array}$$

而在传统的cross entropy的做法里，每一笔data对结果都会有影响，因此鲁棒性就没有那么好

redefine model and loss function

知道 w 是 x^n 的线性组合之后，我们就可以对原先的SVM函数进行改写：

$$\begin{aligned} w &= \sum_n \alpha_n x^n = X\alpha \\ f(x) &= w^T x = \alpha^T X^T x = \sum_n \alpha_n (x^n \cdot x) \end{aligned}$$

这里的 x 表示新的data， x^n 表示数据集中已存在的所有data，由于很多 α_n 为0，因此内积的计算量并不是很大

Dual Representation

$$w = \sum_n \alpha_n x^n = X\alpha \quad X = \begin{bmatrix} x^1 & x^2 & \dots & x^N \end{bmatrix} \quad \alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix}$$

$w = X\alpha$

Step 1: $f(x) = w^T x \rightarrow f(x) = \alpha^T X^T x$

$$f(x) = \sum_n \alpha_n (x^n \cdot x)$$

$$= \sum_n \alpha_n K(x^n, x)$$

$$\begin{bmatrix} \alpha_1 & \dots & \alpha_N \end{bmatrix}$$

$$\begin{bmatrix} x^1 \cdot x \\ x^2 \cdot x \\ \vdots \\ x^N \cdot x \end{bmatrix}$$

接下来把 x^n 与 x 的内积改写成**Kernel function**的形式: $x^n \cdot x = K(x^n, x)$

此时model就变成了 $f(x) = \sum_n \alpha_n K(x^n, x)$, 未知的参数变成了 α_n

现在我们的目标是, 找一组最好的 α_n , 让loss最小, 此时损失函数改写为:

$$L(f) = \sum_n l(\sum_{n'} \alpha_{n'} K(x^{n'}, x^n), \hat{y}^n)$$

从中可以看出, 我们并不需要真的知道 x 的vector是多少, 需要知道的只是 x 跟另外一个vector z 之间的内积值 $K(x, z)$, 也就是说, 只要知道 $K(x, z)$ 的值, 就可以去对参数做优化了, 这招就叫做**Kernel Trick**

只要满足 w 是 x^n 的线性组合, 就可以使用Kernel Trick, 所以也可以有Kernel based Logistic Regression, Kernel based Linear Regression

Step 1: $f(x) = \sum_n \alpha_n K(x^n, x)$

Step 2, 3: Find $\{\alpha_1^*, \dots, \alpha_n^*, \dots, \alpha_N^*\}$, minimizing loss function L

$$L(f) = \sum_n l(f(x^n), \hat{y}^n)$$

$$= \sum_n l\left(\sum_{n'} \alpha_{n'} K(x^{n'}, x^n), \hat{y}^n\right)$$

We don't really need to know vector x
We only need to know the inner project between a pair of vectors x and z
 $K(x, z)$

Kernel Trick

Kernel Trick

linear model会有很多的限制，有时候需要对输入的feature做一些转换之后，才能用linear model来处理

假设现在我们的data是二维的， $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ ，先要对它做feature transform，然后再去应用Linear SVM

如果要考虑特征之间的关系，则把特征转换为 $\phi(x) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$ ，此时Kernel function就变为：

$$K(x, z) = \phi(x) \cdot \phi(z) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \cdot \begin{bmatrix} z_1^2 \\ \sqrt{2}z_1z_2 \\ z_2^2 \end{bmatrix} = (x_1z_1 + x_2z_2)^2 = (\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} z_1 \\ z_2 \end{bmatrix})^2 = (x \cdot z)^2$$

Kernel Trick

Directly computing $K(x, z)$ can be faster than “feature transformation + inner product” sometimes.

Kernel trick is useful when we transform all x to $\phi(x)$

$$\begin{aligned} K(x, z) &= \phi(x) \cdot \phi(z) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \cdot \begin{bmatrix} z_1^2 \\ \sqrt{2}z_1z_2 \\ z_2^2 \end{bmatrix} \\ x &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= x_1^2 z_1^2 + 2x_1 x_2 z_1 z_2 + x_2^2 z_2^2 \end{aligned}$$

$$\begin{aligned} \phi(x) &= \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \\ &= (x_1 z_1 + x_2 z_2)^2 = \left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \right)^2 \\ &= (x \cdot z)^2 \end{aligned}$$

可见，我们对 x 和 z 做特征转换 $\phi(x)$ +内积，就等同于在原先的空间上先做内积再平方，在高维空间里，这种方式可以有更快的速度和更小的运算量

Kernel Trick

Directly computing $K(x, z)$ can be faster than “feature transformation + inner product” sometimes.

$$K(x, z) = (x \cdot z)^2 \\ = (x_1 z_1 + x_2 z_2 + \dots + x_k z_k)^2$$

$$= \underline{x_1}^2 \underline{z_1}^2 + \underline{x_2}^2 \underline{z_2}^2 + \dots + \underline{x_k}^2 \underline{z_k}^2$$

$$+ 2\underline{x_1} \underline{x_2} \underline{z_1} \underline{z_2} + 2\underline{x_1} \underline{x_3} \underline{z_1} \underline{z_3} + \dots$$

$$+ 2\underline{x_2} \underline{x_3} \underline{z_2} \underline{z_3} + 2\underline{x_2} \underline{x_4} \underline{z_2} \underline{z_4} + \dots$$

$$= \phi(x) \cdot \phi(z)$$

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} \quad z = \begin{bmatrix} z_1 \\ \vdots \\ z_k \end{bmatrix}$$

$$\phi(x) = \begin{bmatrix} x_1^2 \\ \vdots \\ x_k^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \vdots \\ \sqrt{2}x_2x_3 \\ \vdots \end{bmatrix}$$

Radial Basis Function Kernel

在Radial Basis Function Kernel中, $K(x, z) = e^{-\frac{1}{2}\|x - z\|_2}$, 如果x和z越像, Kernel的值越大。实际上也可以表示为 $\phi(x) \cdot \phi(z)$, 只不过 $\phi(*)$ 的维数是无穷大的, 所以我们直接使用Kernel trick计算, 其实就等同于在无穷多维的空间中计算两个向量的内积

将Kernel展开成无穷维如下:

Radial Basis Function Kernel

$$K(x, z) = \exp\left(-\frac{1}{2}\|x - z\|_2\right) = \phi(x) \cdot \phi(z)?$$

$$= \exp\left(-\frac{1}{2}\|x\|_2 - \frac{1}{2}\|z\|_2 + x \cdot z\right) \quad \phi(*) \text{ has inf dim!!!}$$

$$= \exp\left(-\frac{1}{2}\|x\|_2\right) \exp\left(-\frac{1}{2}\|z\|_2\right) \exp(x \cdot z) = C_x C_z \exp(x \cdot z)$$

$$= C_x C_z \sum_{i=0}^{\infty} \frac{(x \cdot z)^i}{i!} = C_x C_z + C_x C_z (x \cdot z) + C_x C_z \frac{1}{2} (x \cdot z)^2 \dots$$

$$[C_x] \cdot [C_z] \quad \left[\begin{array}{c} C_x x_1 \\ C_x x_2 \\ \vdots \end{array} \right] \cdot \left[\begin{array}{c} C_z z_1 \\ C_z z_2 \\ \vdots \end{array} \right] \quad \frac{1}{\sqrt{2}} \left[\begin{array}{c} C_x x_1^2 \\ \vdots \\ \sqrt{2} C_x x_1 x_2 \\ \vdots \end{array} \right] \cdot \frac{1}{\sqrt{2}} \left[\begin{array}{c} C_z z_1^2 \\ \vdots \\ \sqrt{2} C_z z_1 z_2 \\ \vdots \end{array} \right]$$

把与 x 相关的无穷多项串起来就是 $\phi(x)$, 把与 z 相关的无穷多项串起来就是 $\phi(z)$, 也就是说, 当你使用RBF Kernel的时候, 实际上就是在无穷多维的平面上做事情, 当然这也意味着很容易过拟合

Sigmoid Kernel

Sigmoid Kernel: $K(x, z) = \tanh(x \cdot z)$, $\tanh(x \cdot z)$ 是哪两个 high dimension vector 做 Inner Product 的结果, 自己回去用 Taylor Expansion 展开来看就知道了

如果使用的是Sigmoid Kernel, 那model $f(x)$ 就可以被看作是只有一层hidden layer的神经网络, 其中 $x^1 \sim x^n$ 可以被看作是neuron的weight, 变量 x 乘上这些weight, 再通过Hyperbolic Tangent 激活函数, 最后全部乘上 $\alpha^1 \sim \alpha^n$ 做加权和, 得到最后的 $f(x)$

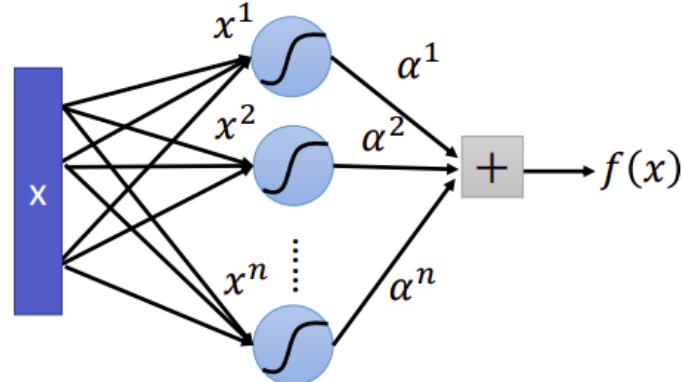
Sigmoid Kernel

$$K(x, z) = \tanh(x \cdot z)$$

- When using sigmoid kernel, we have a 1 hidden layer network.

$$f(x) = \sum_n \alpha_n K(x^n, x) = \sum_n \alpha^n \tanh(x^n \cdot x)$$

The weight of each neuron is a data point
The number of support vectors is the number of neurons.



其中neuron的数目, 由support vector的数量决定

Design Kernel Function

既然有了Kernel Trick, 其实就可以直接去设计Kernel Function, 它代表了投影到高维以后的内积, 类似于相似度的概念

我们完全可以不去管 x 和 z 的特征长什么样, 因为用低维的 x 和 z 加上 $K(x, z)$, 就可以直接得到高维空间中 x 和 z 经过转换后的内积, 这样就省去了转换特征这一步

当 x 是一个有结构的对象, 比如不同长度的sequence, 它们其实不容易被表示成vector, 我们不知道 x 的样子, 就更不用说 $\phi(x)$ 了, 但是只要知道怎么计算两者之间的相似度, 就有机会把这个Similarity当做Kernel来使用

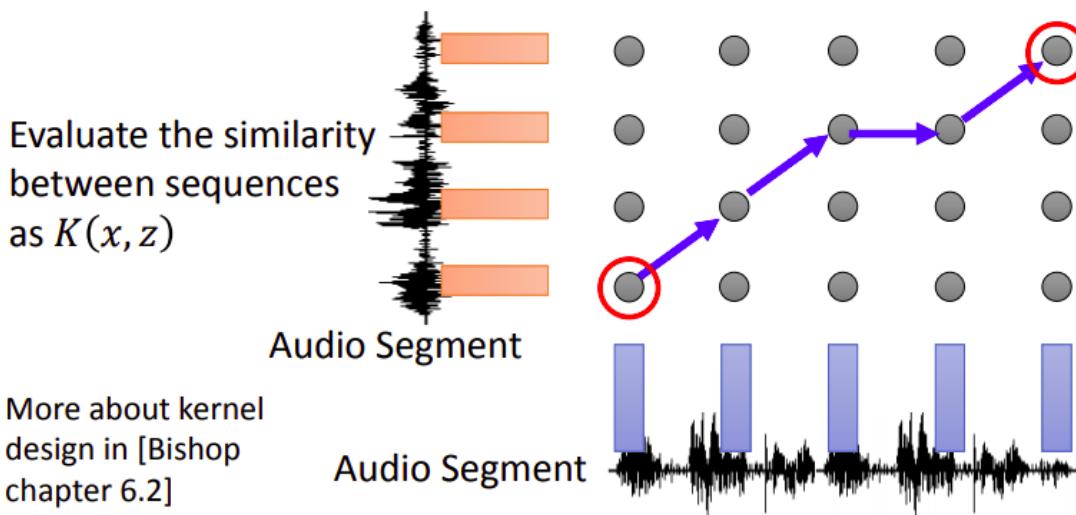
我们随便定义一个Kernel Function, 其实并不一定能够拆成两个向量内积的结果, 但有Mercer's theory可以帮助你判断当前的function是否可拆分

下图是直接定义语音vector之间的相似度 $K(x, z)$ 来做Kernel Trick的示例:

You can directly design $K(x, z)$ instead of considering $\phi(x), \phi(z)$

When x is structured object like sequence, hard to design $\phi(x)$

$K(x, z)$ is something like similarity (Mercer's theory to check)



Hiroshi Shimodaira, Ken-ichi Noma, Mitsuru Nakai, Shigeki Sagayama, "Dynamic Time-Alignment Kernel in Support Vector Machine", NIPS, 2002

Marco Cuturi, Jean-Philippe Vert, Oystein Birkenes, Tomoko Matsui, A kernel for time series based on global alignments, ICASSP, 2007

SVM related methods

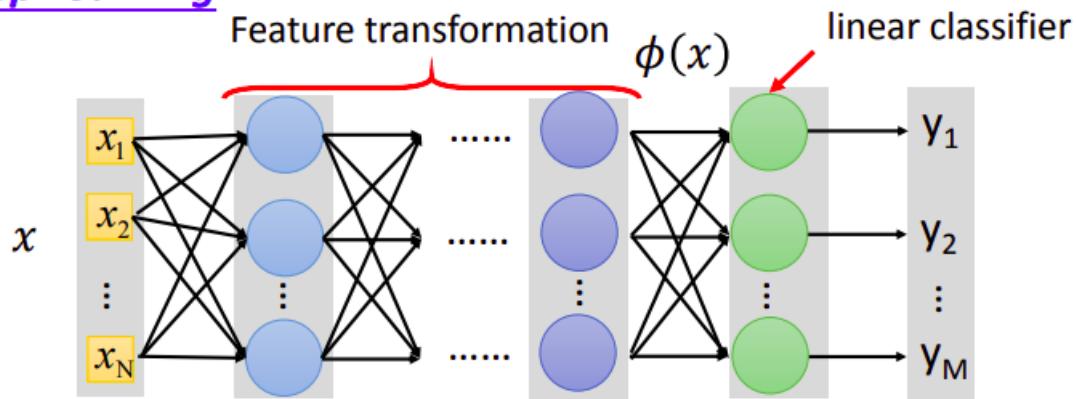
- Support Vector Regression(SVR)
 - [Bishop chapter 7.1.4]
- Ranking SVM
 - [Alpaydin, Chapter 13.11]
- One-class SVM
 - [Alpaydin, Chapter 13.11]

SVM vs Deep Learning

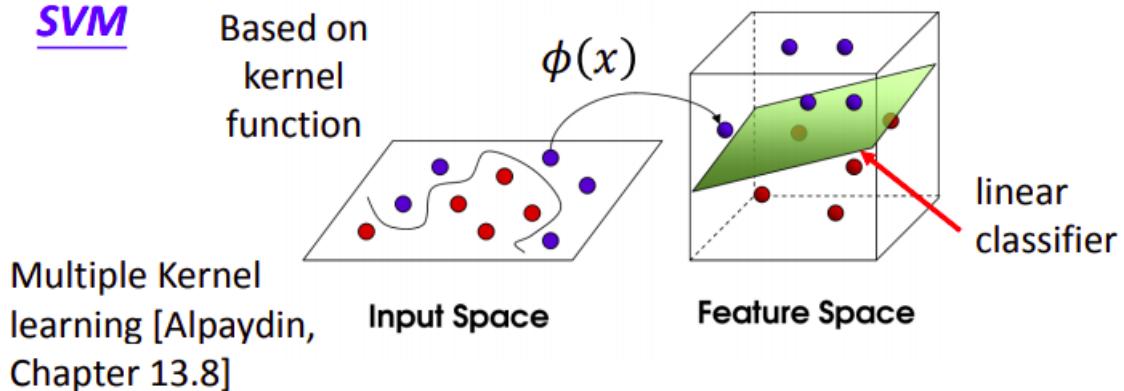
这里简单比较一下SVM和Deep Learning的差别:

- deep learning的前几层layer可以看成是在做feature transform，而后几层layer则是在做linear classifier
- SVM也类似，先用Kernel Function把feature transform到高维空间上，然后再使用linear classifier
在SVM里一般Linear Classifier都会采用Hinge Loss

Deep Learning



SVM



事实上SVM的Kernel是 learnable 的，但是它没有办法 learn 的像 Deep Learning 那么多。

你可以做的是你有好几个不同的 kernel，然后把不同 kernel combine 起来，它们中间的 weight 是可以 learn 的。

当你只有一个 kernel 的时候，SVM 就好像是只有一个 Hidden Layer 的 Neural Network，当你把 kernel 在做 Linear Combination 的时候，它就像一个有两个 layer 的 Neural Network

Ensemble

Ensemble的方法就是一种团队合作，好几个模型一起上的方法。

Framework of Ensemble

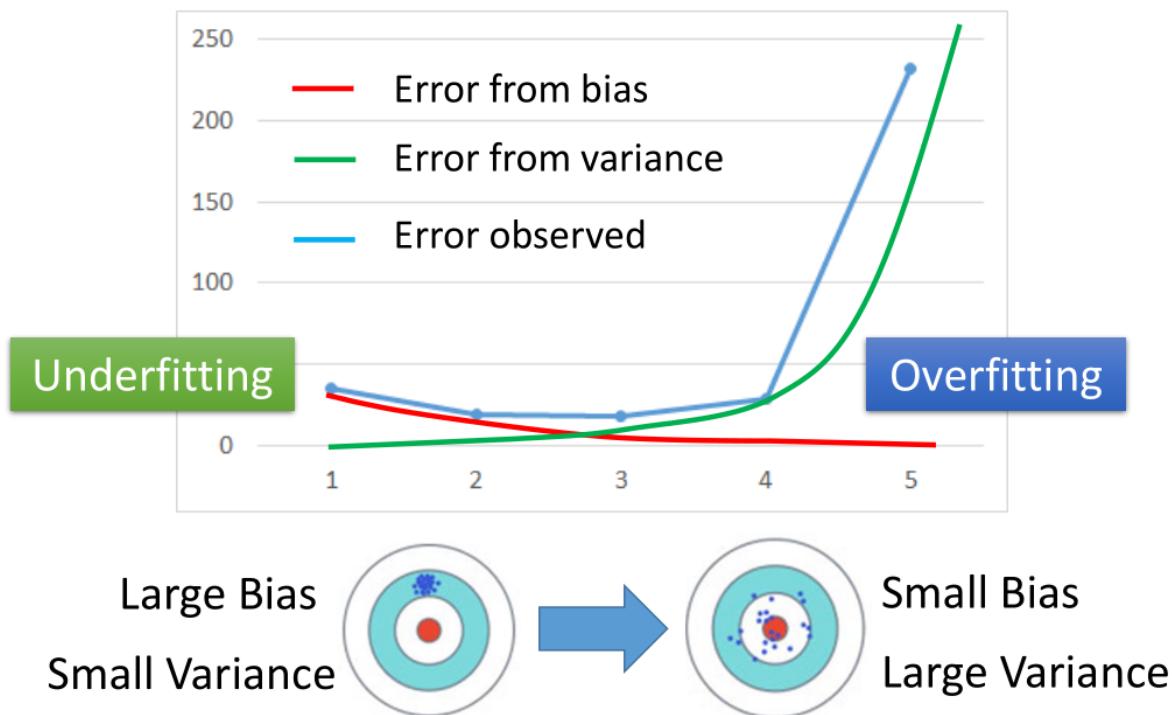
Get a set of classifiers

第一步：通常情况是有很多的classifier，想把他们集合在一起发挥更强大的功能，这些classifier一般是 diverse的，这些classifier有不同的属性和不同的作用。就像moba游戏中每个人都有自己需要做的工作。

Aggregate the classifiers (properly)

第二步：就是要把classifier用比较好的方法集合在一起，就好像打团的时候输出和肉都站不同的位置。通常用ensemble可以让我们的表现提升一个档次，在kaggle之类的比赛中，你有一个好的模型，你可以拿到前几名，但你要夺得冠军你通常会需要 ensemble。

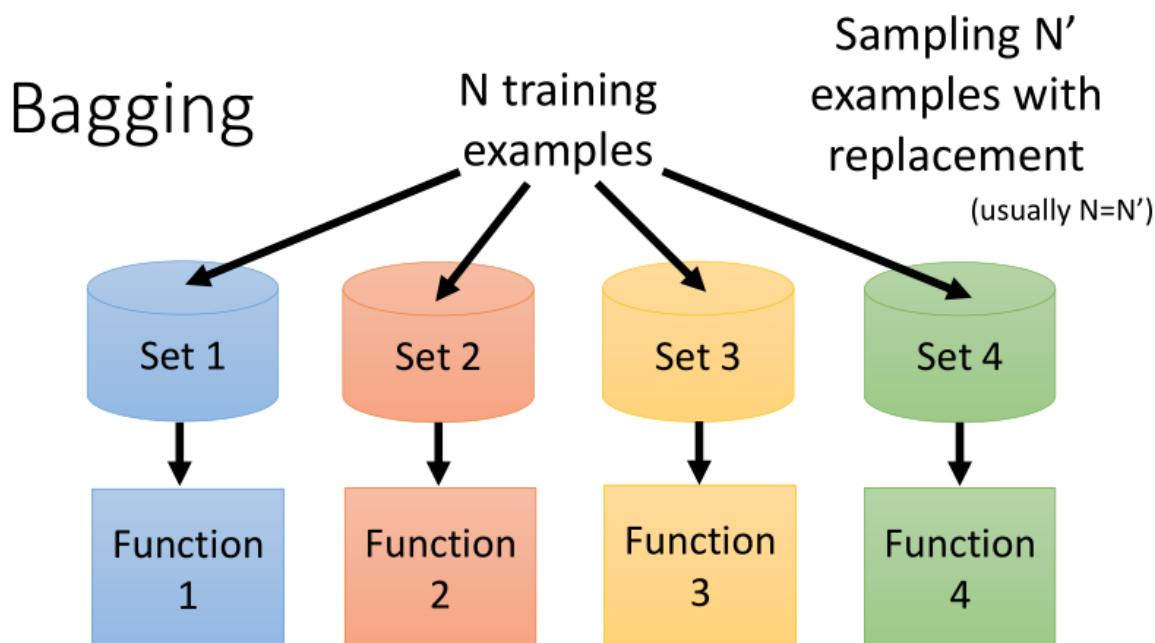
Bagging



我们先来回顾一下bias和variance，对于简单的模型，我们会有比较大的bias但是有比较小的variance，如果是复杂的模型，则有比较小的bias但是有比较大的variance。在这两者的组合下，我们最后的误差（蓝色的线）会随着模型复杂度的增加，先下降后逐渐上升。

如果一个复杂的模型就会有很大的variance。这些模型的variance虽然很大，但是bias是比较小的，所以我们可以把不同的模型都集合起来，把输出做一个平均，得到一个新的模型 \hat{f} ，这个结果可能和正确的答案就是接近的。Bagging就是要体现这个思想。

Bagging就是我们自己创造出不同的dataset，再用不同的dataset去训练一个复杂的模型，每个模型独自拿出来虽然方差很大，但是把不同的方差大的模型集合起来，整个的方差就不会那么大，而且偏差也会很小。



怎么自己制造不同的 data 呢？

假设现在有 N 笔 Training Data, 对这 N 笔 Training Data 做 Sampling, 从这 N 笔 Training Data 里面每次取 N' 笔 data 组成一个新的 Data Set。

通常在做 Sampling 的时候会做 replacement, 抽出一笔 data 以后会再把它放到 pool 里面去, 那所以通常 N' 可以设成 N。所以把 N' 设成 N, 从 N 这个 Data Set 里面做 N 次的 Sample with replacement, 得到的 Data Set 跟原来的这 N 笔 data 并不会一样, 因为你可能会反复抽到同一个 example。

总之我们就用 sample 的方法建出好几个 Data Set。每一个 Data Set 都有 N' 笔 Data, 每一个 Data Set 里面的 Data 都是不一样的。

接下来你再用一个复杂的模型去对这四个 Data Set 做 Learning, 就找出了四个 function。接下来在 testing 的时候, 就把一笔 testing data 丢到这四个 function 里面, 再把得出来的结果作平均或者是作 Voting。通常就会比只有一个 function 的时候 performance 还要好, Variance 会比较小, 所以你得到的结果会是比较 robust 的, 比较不容易 Overfitting。

如果做的是 regression 方法的时候, 你可能会用 average 的方法来把四个不同 function 的结果组合起来, 如果是分类问题的话可能会用 Voting 的方法把四个结果组合起来。

注意一下, 当你的 model 很复杂的时候、担心它 Overfitting 的时候才做 Bagging。

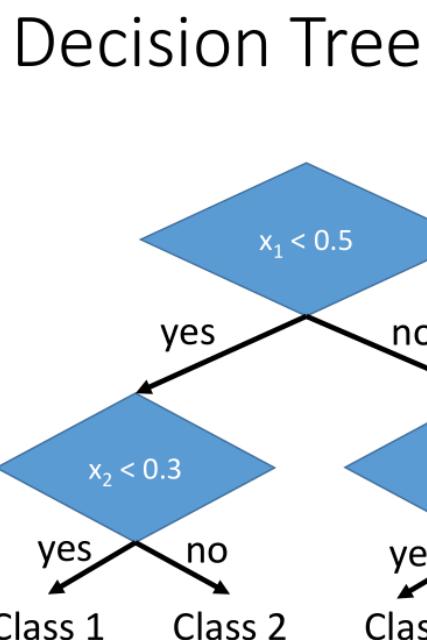
做 Bagging 的目的是为了要减低 Variance, 你的 model Bias 已经很小但 Variance 很大, 想要减低 Variance 的时候, 你才做 Bagging。

This approach would be helpful when your model is complex, easy to overfit.

所以适用做 Bagging 的情况是, 你的 Model 本身已经很复杂, 在 Training Data 上很容易就 Overfit, 这个时候你会想要用 Bagging。

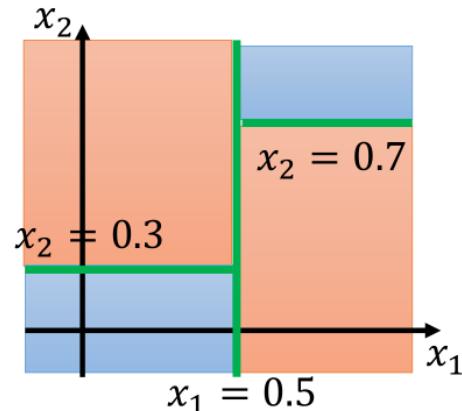
举例来说 Decision Tree 就是一个非常容易 Overfit 的方法。所以 Decision Tree 很需要做 Bagging。Random Forest 就是 Decision Tree 做 Bagging 的版本。

Decision Tree



Can have more complex questions

Assume each object x is represented by a 2-dim vector $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



The questions in training

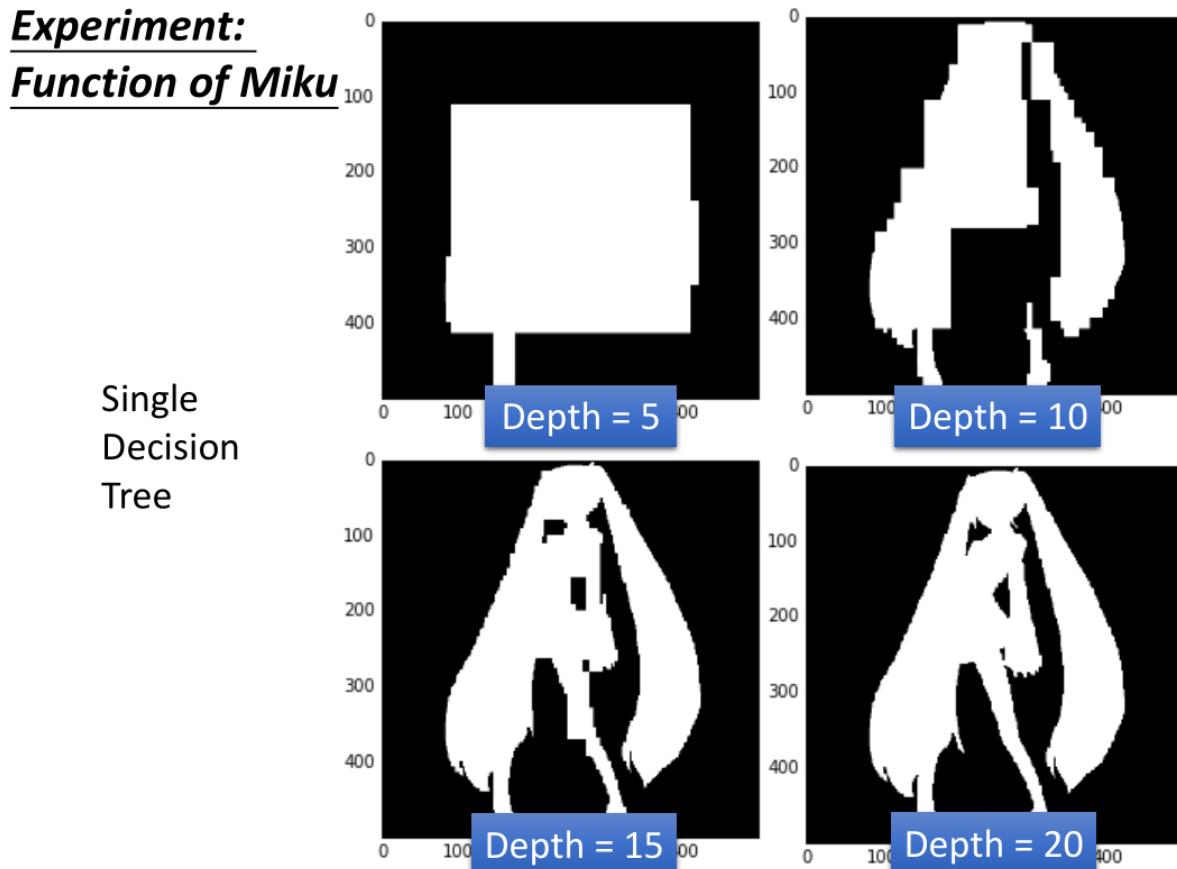
number of branches,
Branching criteria,
termination criteria,
base hypothesis

假设给定的每个Object有两个feature，我们就用这个training data建立一颗树，如果 x_1 小于0.5就是yes（往左边走），当 x_1 大于0.5就是no（往右边走），接下来看 x_2 ，当 x_2 小于0.3时就是class 1（对应坐标轴图中左下角的蓝色）当大于0.3时候就是class 2（红色）；对右边的当 x_2 小于0.7时就是红色，当 x_2 大于0.7就是蓝色。这是一个比较简单的例子，其实可以同时考虑多个dimension，变得更复杂。

做决策树时会有很多地方需要注意：比如每个节点分支的数量，用什么样的criterion来进行分支，什么时候停止分支，有哪些可以问的问题等等，也是有很多参数要调。

Experiment: Function of Miku

描述：输入的特征是二维的，其中class 1分布的和初音的样子是一样的。我们用决策树对这个问题进行分类。



上图可以看到，深度是5的时候效果并不好，图中白色的就是class 1，黑色的是class 2.当深度是10的时候有一点初音的样子，当深度是15的时候，基本初音的轮廓就出来了，但是一些细节还是很奇怪（比如一些凸起来的边角）当深度是20的时候，就可以完美的把class 1和class 2的位置区别开来，就可以完美地把初音的样子勾勒出来了。对于决策树，理想的状况下可以达到错误是0的时候，最极端的就是每一笔data point就是很深的树的一个节点，这样正确率就可以达到100%（树够深，决策树可以做出任何的function）但是决策树很容易过拟合，如果只用决策树一般很难达到好的结果。

Random Forest

Random Forest

train	f_1	f_2	f_3	f_4
x^1	O	X	O	X
x^2	O	X	X	O
x^3	X	O	O	X
x^4	X	O	X	O

- Decision tree:
 - Easy to achieve 0% error rate on training data
 - If each training example has its own leaf
- Random forest: Bagging of decision tree
 - Resampling training data is not sufficient
 - Randomly restrict the features/questions used in each split
- Out-of-bag validation for bagging
 - Using $RF = f_2 + f_4$ to test x^1
 - Using $RF = f_2 + f_3$ to test x^2
 - Using $RF = f_1 + f_4$ to test x^3
 - Using $RF = f_1 + f_3$ to test x^4

Out-of-bag (OOB) error
Good error estimation
of testing set

传统的随机森林是通过之前的重采样的方法做，但是得到的结果是每棵树都差不多（效果并不好）。比较typical 的方法是在每一次要产生 Decision Tree 的 branch 要做 split 的时候，都 random 的决定哪一些 feature 或哪一些问题是不能用。这样就能保证就算用同样的dataset，每次产生的决策树也会是不一样的，最后把所有的决策树的结果都集合起来，就会得到随机森林。

如果是用Bagging的方法的话，用**out-of-bag**可以做验证。用这个方法可以不用把label data划分成 training set和validation set，一样能得到同样的效果。

具体做法：假设我们有training data是 x^1, x^2, x^3, x^4 ， f_1 我们只用第一笔和第二笔data训练（上图中圆圈表示训练，叉表示没训练）， f_2 我们只用第三笔第四笔data训练， f_3 用第一，第三笔data训练， f_4 表示用第二，第四笔data训练，我们知道，在训练 f_1 和 f_4 的时候没用用到 x^1 ，所以我们就可以用 f_1 和 f_4 Bagging的结果在 x^1 上面测试他们的表现。

同理，我们可以用 f_2 和 f_3 Bagging的结果来测试 x^2 ，用 f_1 跟 f_4 Bagging 的结果 test x_3 ，用 f_1 跟 f_3 Bagging 的结果 test x_4 。

接下来再把 x_1 跟 x_4 的结果把它做平均，算一下 error rate 就得到 Out-of-bag 的 error。虽然我们没有明确的切出一个验证集，但是我们做测试的时候所有的模型并没有看过那些测试的数据。所有这个输出的error也是可以作为反映测试集结果的估测效果。

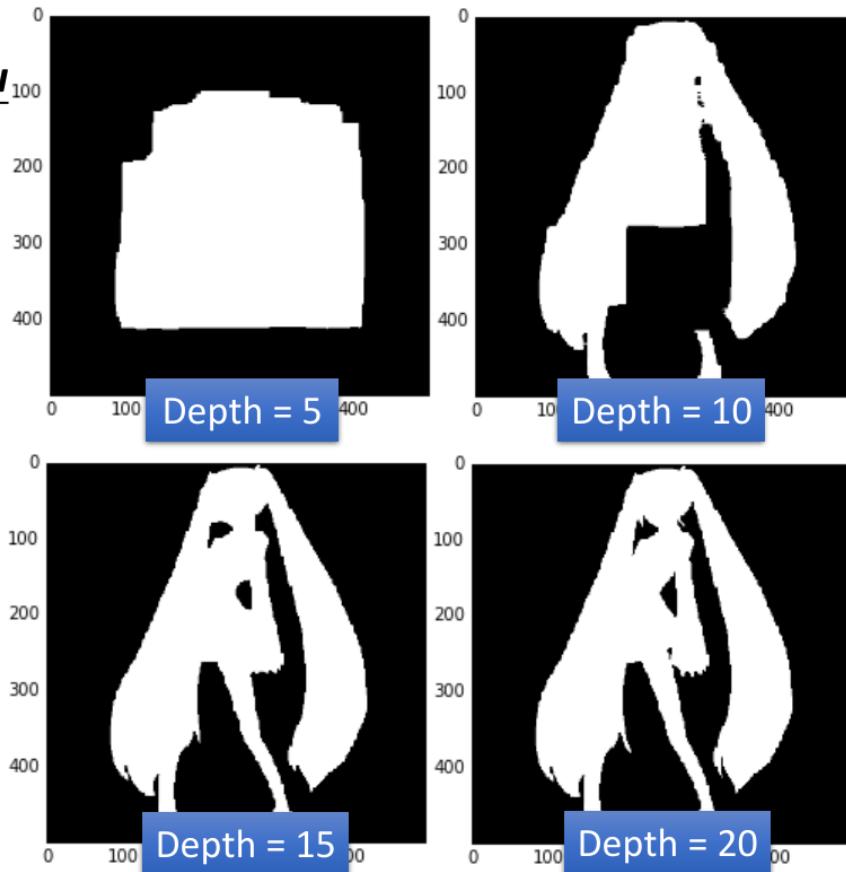
接下来是用随机森林做的实验结果：

Experiment:

Function of Miku

Random
Forest

(100 trees)



强调一点是做Bagging并不会使模型能更fit data，所以用深度为5的时候还是不能fit出那个function，就是5颗树的一个平均，相当于得到一个比较平滑的树。当深度是10的时候，大致的形状能看出来了，当15的时候效果就还不错，但是细节没那么好，当20 的时候就可以完美的把初音分出来。

Boosting

Boosting

Training data:

$$\{(x^1, \hat{y}^1), \dots, (x^n, \hat{y}^n), \dots, (x^N, \hat{y}^N)\}$$

$\hat{y} = \pm 1$ (binary classification)

- Guarantee:
 - If your ML algorithm can produce classifier with error rate smaller than 50% on training data
 - You can obtain 0% error rate classifier after boosting.
- Framework of boosting
 - Obtain the first classifier $f_1(x)$
 - Find another function $f_2(x)$ to help $f_1(x)$
 - However, if $f_2(x)$ is similar to $f_1(x)$, it will not help a lot.
 - We want $f_2(x)$ to be complementary with $f_1(x)$ (How?)
 - Obtain the second classifier $f_2(x)$
 - Finally, combining all the classifiers
- The classifiers are learned sequentially.

Boosting是用在很弱的模型上的，当我们有很弱的模型的时候，不能fit我们的data的时候，我们就可以用Boosting的方法。

Boosting有一个很强的guarantee：假设有一个ML的algorithm，它可以给你一个错误率高过50%的classifier，只要能够做到这件事，Boosting这个方法可以保证最后把这些错误率仅略高于50%的classifier组合起来以后，它可以让错误率达到0%。

Boosting的结构：

- 首先要找一个分类器 $f_1(x)$
- 接下再找一个辅助 $f_1(x)$ 的分类器 $f_2(x)$ （注意 $f_2(x)$ 如果和 $f_1(x)$ 很像，那么 $f_2(x)$ 的帮助效果就不好，所以要尽量找互补的 $f_2(x)$ ，能够弥补 $f_1(x)$ 没办法做到的事情）
- 得到第二个分类器 $f_2(x)$
-
- 最后就结合所有的分类器得到结果

要注意的是在做Boosting的时候，classifier的训练是有顺序的（sequential），要先找 f_1 才知道怎么找跟 f_1 互补的 f_2 ，所以它是有顺序的找。在Bagging的时候，每一个classifier是没有顺序的

How to obtain different classifiers?

- Training on different training data sets
- How to have different training data sets
 - Re-sampling your training data to form a new set
 - Re-weighting your training data to form a new set
 - In real implementation, you only have to change the cost/objective function

(x^1, \hat{y}^1, u^1)	$u^1 = \cancel{1}$	0.4
(x^2, \hat{y}^2, u^2)	$u^2 = \cancel{1}$	2.1
(x^3, \hat{y}^3, u^3)	$u^3 = \cancel{1}$	0.7

$$L(f) = \sum_n l(f(x^n), \hat{y}^n)$$

$$L(f) = \sum_n u^n l(f(x^n), \hat{y}^n)$$

制造不同的训练数据来得到不同的分类器

用重采样的方法来训练数据得到新的数据集；用重新赋权重的方法来训练数据得到新的数据集。

上图中用 u 来代表每一笔data的权重，可以通过改变weight来制造不同的data，举例来说就是刚开始都是1，第二次就分别改成0.4, 2.1, 0.7，这样就制造出新的data set。在实际中，就算改变权重，对训练没有太大影响。在训练时，原来的loss function是 $L(w) = \sum_n l(f(x^n), \hat{y}^n)$ ，其中 l 可以是任何不同的function，只要能衡量 $f(x^n)$ 和 \hat{y}^n 之间的差距就行，然后用gradient descent的方法来最小化这个 L （total loss function）。当加上权重后，变成了 $L(w) = \sum_n u^n l(f(x^n), \hat{y}^n)$ ，相当于就是在原来的基础上乘以 u 。这样从loss function来看，如果有一笔data的权重比较重，那么在训练的时候就会被多考虑一点。

Adaboost

- Idea: training $f_2(x)$ on the new training set that fails $f_1(x)$
- How to find a new training set that fails $f_1(x)$?

ε_1 : the error rate of $f_1(x)$ on its training data

$$\varepsilon_1 = \frac{\sum_n u_1^n \delta(f_1(x^n) \neq \hat{y}^n)}{Z_1} \quad Z_1 = \sum_n u_1^n \quad \varepsilon_1 < 0.5$$

Changing the example weights from u_1^n to u_2^n such that

$$\frac{\sum_n u_2^n \delta(f_1(x^n) \neq \hat{y}^n)}{Z_2} = 0.5$$

The performance of f_1 for new weights would be random.

Training $f_2(x)$ based on the new weights u_2^n

想法：先训练好一个分类器 $f_1(x)$, 要找一组新的training data, 让 $f_1(x)$ 在这组data上的表现很差, 然后让 $f_2(x)$ 在这组training data上训练。

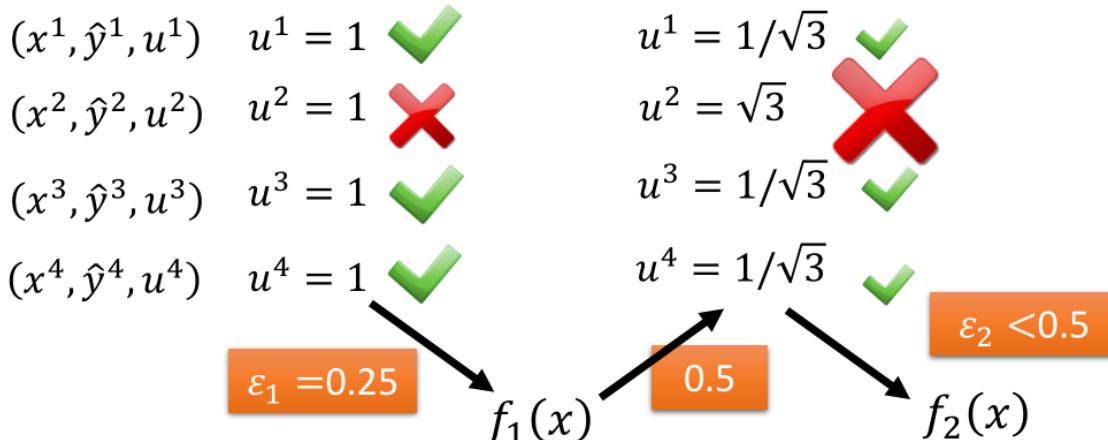
怎么找一个新的训练数据集让 $f_1(x)$ 表现差?

上图中的 ε_1 就是训练数据的error rate, 这个就是对所有训练的样本求和, $\delta(f_1(x^n) \neq \hat{y}^n)$ 是计算每笔的 training sample 分类正确与否, 用0来表示正确, 用1来表示错误, 乘以一个 weight u , 然后做 normalization, 这个 Z_1 对所有的 weight 标准化, 这里的 $\varepsilon_1 < 0.5$

然后我们想要用 u_2 作为权重的数据来进行计算得到 error rate, 在新的权重上, $f_1(x)$ 的表现就是随机的, 恰好等于 0.5, 接下来我们拿这组新的训练数据集再去训练 $f_2(x)$, 这样的 $f_2(x)$ 和 $f_1(x)$ 就是互补的。

Re-weighting Training Data

- Idea: training $f_2(x)$ on the new training set that fails $f_1(x)$
- How to find a new training set that fails $f_1(x)$?



假设我们上面的四组训练数据, 权重就是 u_1 到 u_4 , 并且每个初始值都是 1, 我们现在用这四组训练数据去训练一个模型 $f_1(x)$, 假设 $f_1(x)$ 只分类正确其中的三笔训练数据, 所以 $\varepsilon_1 = 0.25$

然后我们改变每个权重, 把对的权重改小一点, 把第二笔错误的权重改大一点, $f_1(x)$ 在新的训练数据集上表现就会变差 $\varepsilon_1 = 0.5$ 。

然后在得到的新的训练数据集上训练得到 $f_2(x)$, 这个 $f_2(x)$ 训练完之后得到的 ε_2 会比 0.5 小。

- Idea: **training $f_2(x)$ on the new training set that fails $f_1(x)$**
- How to find a new training set that fails $f_1(x)$?

$$\left\{ \begin{array}{l} \text{If } x^n \text{ misclassified by } f_1 (f_1(x^n) \neq \hat{y}^n) \\ \quad u_2^n \leftarrow u_1^n \text{ multiplying } d_1 \quad \text{increase} \\ \text{If } x^n \text{ correctly classified by } f_1 (f_1(x^n) = \hat{y}^n) \\ \quad u_2^n \leftarrow u_1^n \text{ devided by } d_1 \quad \text{decrease} \end{array} \right.$$

f_2 will be learned based on example weights u_2^n

What is the value of d_1 ?

假设训练数据 x^n 会被 $f_1(x)$ 分类错, 那么就把第 n 笔 data 的 u_1^n 乘上 d_1 变成 u_2^n , 这个 d_1 是大于 1 的值

如果 x^n 正确的被 $f_1(x)$ 分类的话, 那么就用 u_1^n 除以 d_1 变成 u_2^n

$f_2(x)$ 就会在新的权重 u_2^n 上进行训练。

Re-weighting Training Data

$$\begin{aligned} \varepsilon_1 &= \frac{\sum_n u_1^n \delta(f_1(x^n) \neq \hat{y}^n)}{Z_1} & Z_1 &= \sum_n u_1^n \\ \frac{\sum_n u_2^n \delta(f_1(x^n) \neq \hat{y}^n)}{Z_2} &= 0.5 & f_1(x^n) \neq \hat{y}^n & u_2^n \leftarrow u_1^n \text{ multiplying } d_1 \\ && f_1(x^n) = \hat{y}^n & u_2^n \leftarrow u_1^n \text{ devided by } d_1 \\ &= \sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1 & &= \sum_{f_1(x^n) \neq \hat{y}^n} u_2^n + \sum_{f_1(x^n) = \hat{y}^n} u_2^n \\ &= \sum_n u_2^n & &= \sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1 + \sum_{f_1(x^n) = \hat{y}^n} u_1^n / d_1 \\ & & & \frac{\sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1 + \sum_{f_1(x^n) = \hat{y}^n} u_1^n / d_1}{\sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1} = 2 \end{aligned}$$

分类错误的 $f_1(x^n) \neq \hat{y}^n$ 对应的 u_1^n 就乘上 d_1 ;

Z_2 就等于 $\sum_n u_2^n$, 也等于分类错误和分类正确的两个 u_1^n 的权重和。

所以结合一下然后再取个倒数, 就可以得到图中最后一个式子。

$$\varepsilon_1 = \frac{\sum_n u_1^n \delta(f_1(x^n) \neq \hat{y}^n)}{Z_1} \quad Z_1 = \sum_n u_1^n$$

$$\frac{\sum_n u_2^n \delta(f_1(x^n) \neq \hat{y}^n)}{Z_2} = 0.5 \quad \begin{array}{ll} f_1(x^n) \neq \hat{y}^n & u_2^n \leftarrow u_1^n \text{ multiplying } d_1 \\ f_1(x^n) = \hat{y}^n & u_2^n \leftarrow u_1^n \text{ devided by } d_1 \end{array}$$

$$\frac{\sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1 + \sum_{f_1(x^n) = \hat{y}^n} u_1^n / d_1}{\sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1} = 2 \quad \frac{\sum_{f_1(x^n) = \hat{y}^n} u_1^n / d_1}{\sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1} = 1$$

$$\sum_{f_1(x^n) = \hat{y}^n} u_1^n / d_1 = \sum_{f_1(x^n) \neq \hat{y}^n} u_1^n d_1 \quad \frac{1}{d_1} \sum_{f_1(x^n) = \hat{y}^n} u_1^n = d_1 \sum_{f_1(x^n) \neq \hat{y}^n} u_1^n$$

$$\varepsilon_1 = \frac{\sum_{f_1(x^n) \neq \hat{y}^n} u_1^n}{Z_1} \quad Z_1(1 - \varepsilon_1)$$

$$\sum_{f_1(x^n) \neq \hat{y}^n} u_1^n = Z_1 \varepsilon_1 \quad Z_1(1 - \varepsilon_1) / d_1 = Z_1 \varepsilon_1 d_1$$

$$d_1 = \sqrt{(1 - \varepsilon_1) / \varepsilon_1} > 1$$

最后得到的结果是 $d_1 = \sqrt{(1 - \varepsilon_1) / \varepsilon_1}$

然后用这个 d_1 去乘或者除权重，就能得到让 $f_2(x)$ 表现不好的新的训练数据集

由于 ε_1 小于 0.5，所以 d_1 大于 1

Algorithm for AdaBoost

- Giving training data
 $\{(x^1, \hat{y}^1, u_1^1), \dots, (x^n, \hat{y}^n, u_1^n), \dots, (x^N, \hat{y}^N, u_1^N)\}$
 - $\hat{y} = \pm 1$ (Binary classification), $u_1^n = 1$ (equal weights)
- For $t = 1, \dots, T$:
 - Training weak classifier $f_t(x)$ with weights $\{u_t^1, \dots, u_t^N\}$
 - ε_t is the error rate of $f_t(x)$ with weights $\{u_t^1, \dots, u_t^N\}$
 - For $n = 1, \dots, N$:
 - If x^n is misclassified classified by $f_t(x)$: $\hat{y}^n \neq f_t(x^n)$
 - $u_{t+1}^n = u_t^n \times d_t = u_t^n \times \exp(\alpha_t)$ $d_t = \sqrt{(1 - \varepsilon_t) / \varepsilon_t}$
 - Else:
 - $u_{t+1}^n = u_t^n / d_t = u_t^n \times \exp(-\alpha_t)$ $\alpha_t = \ln \sqrt{(1 - \varepsilon_t) / \varepsilon_t}$

给定一笔训练数据以及其权重，设置初始的权重为 1，接下来用不同的权重来进行很多次迭代训练弱分类器，然后再把这些弱的分类器集合起来就变成一个强的分类器。

其中在每次迭代中，每一笔训练数据都有其对应的权重 u_t^n ，用每个弱分类器对应的权重训练出每个弱分类器 $f_t(x)$ ，计算 $f_t(x)$ 在各自对应权重中的错误率 ε_t 。

然后就可以重新给训练数据赋权值，如果分类错误的数据，就用原来的 u_t^n 乘上 d_t 来更新其权重，反之就把原来的 u_t^n 除以 d_t 得到一组新的权重，然后就继续在下一次迭代中继续重复操作。（其中 $d_t = \sqrt{(1 - \varepsilon_t) / \varepsilon_t}$ ）

或者对 d_t 我们还可以用 $\alpha_t = \ln\sqrt{(1 - \varepsilon)/\varepsilon}$ 来代替，这样我们就可以直接统一用乘的形式来更新 u_t^n ，变成了乘以 $\exp(\alpha_t)$ 或者乘以 $\exp(-\alpha_t)$

这里用 $-\hat{y}^n f_t(x^n)$ 来取正负号（当分类错误该式子就是正的，分类正确该式子就是负的），这样表达式子就会更加简便。

$$u_{t+1}^n \leftarrow u_t^n \times \exp(-\hat{y}^n f_t(x^n) \alpha_t)$$

- We obtain a set of functions: $f_1(x), \dots, f_t(x), \dots, f_T(x)$
- How to aggregate them?

- Uniform weight:

- $H(x) = \text{sign}(\sum_{t=1}^T f_t(x))$

- Non-uniform weight:

- $H(x) = \text{sign}(\sum_{t=1}^T \alpha_t f_t(x))$

Smaller error ε_t ,
larger weight for
final voting

$$\alpha_t = \ln\sqrt{(1 - \varepsilon_t)/\varepsilon_t} \quad \varepsilon^t = 0.1 \quad \varepsilon^t = 0.4$$

$$u_{t+1}^n = u_t^n \times \exp(-\hat{y}^n f_t(x^n) \alpha_t) \quad \alpha^t = 1.10 \quad \alpha^t = 0.20$$

经过刚才的训练之后我们就得到了 $f_1(x)$ 到 $f_T(x)$

一般有两种方法进行集合：

Uniform weight:

我们把T个分类器加起来，看其结果是正的还是负的（正的就代表class 1，负的就代表class 2），这样可以但不是最好的，因为分类器中有好有坏，如果每个分类器的权重都一样的，显然是不合理的。

Non-uniform weight:

在每个分类器前都乘上一个权重 α_t ，然后全部加起来后取结果的正负号，这种方法就能得到比较好的结果。

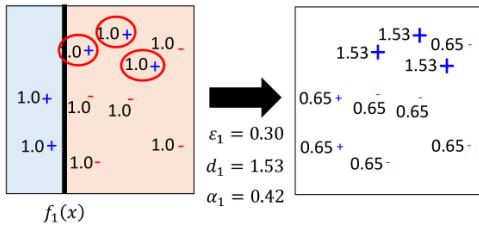
这里的 $\alpha_t = \ln\sqrt{(1 - \varepsilon)/\varepsilon}$ ，从后面的例子可以看到，错误率比较低的 $\varepsilon_t = 0.1$ 得到的 $\alpha_t = 1.10$ 就比较大；反之，如果错误率比较高的 $\varepsilon_t = 0.4$ 得到的 $\alpha_t = 0.20$ 就比较小。

错误率比较小的分类器，最后在最终结果的投票上会有比较大的权重。

Toy example

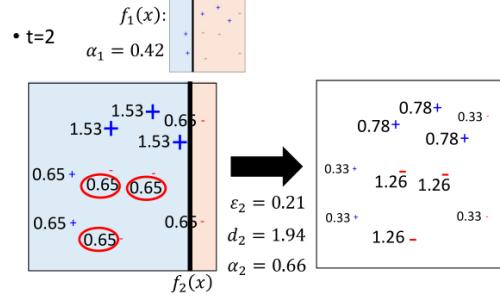
Toy Example T=3, weak classifier = decision stump

• t=1



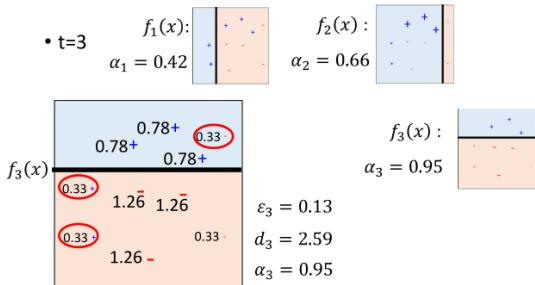
Toy Example T=3, weak classifier = decision stump

• t=2



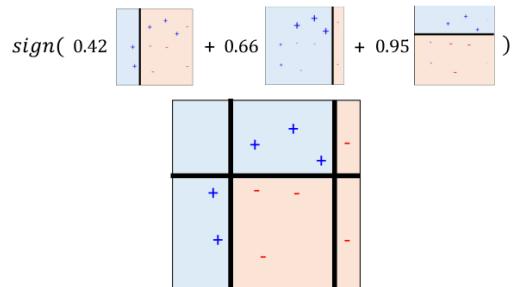
Toy Example T=3, weak classifier = decision stump

• t=3



Toy Example

• Final Classifier: $H(x) = \text{sign}(\sum_{t=1}^T \alpha_t f_t(x))$



Decision stump, 决策树桩：假设所有的特征都分布在二维平面上，在二维平面上选一个维度切一刀，其中一边为class 1，另外一边就当做class 2。

上图中t=1时，我们先用decision stump找一个 $f_1(x)$ ，左边就是正类，右边就是负类，其中会发现有三笔data是错误的，所以能得到错误率是0.3， $d_1=1.53$ (训练数据更新的权重), $\alpha_1=0.42$ (在最终结果投票的权重)，然后改变每笔训练数据的权重。

t=2和t=3按照同样的步骤，就可以得到第二和第三个分类器。由于设置了三次迭代，这样训练就结束了，用之前每个分类器乘以对应的权重，就可以得到最终分类器。

这个三个分类器把平面分割成六个部分，左上角三个分类器都是蓝色的，那就肯定就蓝色的。

上面中间部分第一个分类器是红色的，第二个第三个是蓝色的，但是后面两个加起来的权重比第一个大，所以最终中间那块是蓝色的。

对于右边部分，第一个第二个分类器合起来的权重比第三个蓝色的权重大，所以就是红色的。

下面部分也是按照同样道理，分别得到蓝色，红色和红色。

所以这三个弱分类器其实都会犯错，但是我们把这三个整合起来就能达到100%的正确率了。

Proof

$$H(x) = \text{sign}\left(\sum_{t=1}^T \alpha_t f_t(x)\right) \quad \alpha_t = \ln \sqrt{(1 - \epsilon_t) / \epsilon_t}$$

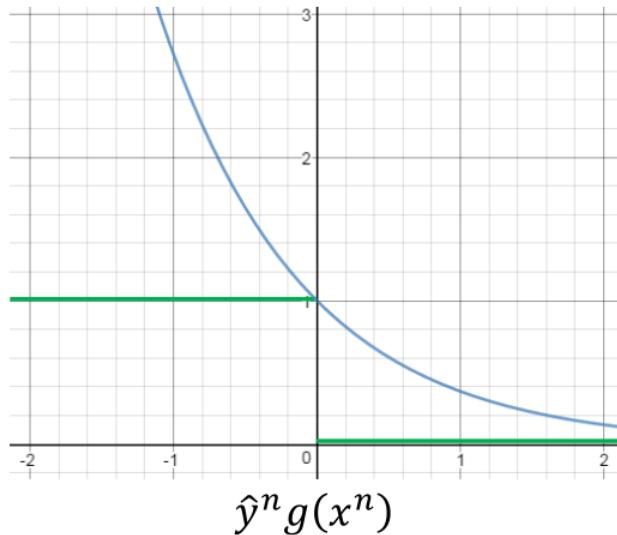
上式中的 $H(x)$ 是最终分类结果的表达式， α_t 是权重， ϵ_t 是错误率。

Proof: As we have more and more $f(t)$ (T increases), $H(x)$ achieves smaller and smaller error rate on training data.

- Final classifier: $H(x) = \text{sign}(\sum_{t=1}^T \alpha_t f_t(x))$
- $\alpha_t = \ln \sqrt{(1 - \varepsilon_t)/\varepsilon_t}$

Training Data Error Rate

$$\begin{aligned}
 &= \frac{1}{N} \sum_n \delta(H(x^n) \neq \hat{y}^n) \\
 &= \frac{1}{N} \sum_n \delta(\hat{y}^n g(x^n) < 0) \\
 &\leq \frac{1}{N} \sum_n \exp(-\hat{y}^n g(x^n))
 \end{aligned}$$



先计算总的训练数据集的错误率，也就是 $\frac{1}{N} \sum_n \delta(H(x^n) \neq \hat{y}^n)$ 其中 $H(x^n) \neq \hat{y}^n$ 得到的就是 1，反之如果 $H(x^n) = \hat{y}^n$ 就是 0。

进一步，可以把 $H(x^n) \neq \hat{y}^n$ 写成 $\hat{y}^n g(x^n) < 0$ 。如果 $\hat{y}^n g(x^n)$ 是同号的代表是正确的，如果是异号就代表分类错误的。整个错误率有一个 upper bound 就是 $\frac{1}{N} \sum_n \exp(-\hat{y}^n g(x^n))$

上图中横轴是 $\hat{y}^n g(x^n)$ ，绿色的线代表的是 δ 的函数，蓝色的是 $\exp(-\hat{y}^n g(x^n))$ 也就是绿色函数的上限。

我们要证明 upper bound 会越来越小

Training Data Error Rate

$$\leq \frac{1}{N} \sum_n \exp(-\hat{y}^n g(x^n)) = \frac{1}{N} Z_{T+1}$$

$$\begin{aligned}
 g(x) &= \sum_{t=1}^T \alpha_t f_t(x) \\
 \alpha_t &= \ln \sqrt{(1 - \varepsilon_t)/\varepsilon_t}
 \end{aligned}$$

Z_t : the summation of the weights of training data for training f_t

$$\text{What is } Z_{T+1} = ? \quad Z_{T+1} = \sum_n u_{T+1}^n$$

$$\left. \begin{array}{l} u_1^n = 1 \\ u_{t+1}^n = u_t^n \times \exp(-\hat{y}^n f_t(x^n) \alpha_t) \end{array} \right\} u_{T+1}^n = \prod_{t=1}^T \exp(-\hat{y}^n f_t(x^n) \alpha_t)$$

$$\begin{aligned}
 Z_{T+1} &= \sum_n \prod_{t=1}^T \exp(-\hat{y}^n f_t(x^n) \alpha_t) \\
 &= \sum_n \exp \left(-\hat{y}^n \sum_{t=1}^T f_t(x^n) \alpha_t \right)
 \end{aligned}$$

上式证明中，思路是先求出 Z_{T+1} （也就是第 $T+1$ 次训练数据集权重的和），就等于 $\sum_n u_{T+1}^n$

而 u_{t+1}^n 与 u_t^n 有关系，通过 u_{t+1}^n 在图中的表达式

能得到 u_{T+1}^n 就是T次连乘的 $\exp(-\hat{y}^n f_t(x^n) \alpha_t)$ ，也就是 u_{T+1}^n ，然后在累加起来得到 Z_{T+1}

同时把累乘放到 \exp 里面去变成了累加，由于 \hat{y}^n 是迭代中第n笔的正确答案，所以和累乘符号没有关系

就会发现后面的 $\sum_{t=1}^T f_t(x^n) \alpha_t$ 恰好等于图片最上面的 $g(x)$ 。

这样就说明了，训练数据的权重的和会和训练数据的错误率有关系。接下来就是证明权重的和会越来越小就可以了。

Training Data Error Rate

$$\leq \frac{1}{N} \sum_n \exp(-\hat{y}^n g(x^n)) = \frac{1}{N} Z_{T+1}$$

$$g(x) = \sum_{t=1}^T \alpha_t f_t(x)$$

$$\alpha_t = \ln \sqrt{(1 - \varepsilon_t) / \varepsilon_t}$$

$$Z_1 = N \text{ (equal weights)}$$

$$Z_t = \underline{Z_{t-1} \varepsilon_t} \exp(\alpha_t) + \underline{Z_{t-1} (1 - \varepsilon_t)} \exp(-\alpha_t)$$

Misclassified portion in Z_{t-1} Correctly classified portion in Z_{t-1}

$$= Z_{t-1} \varepsilon_t \sqrt{(1 - \varepsilon_t) / \varepsilon_t} + Z_{t-1} (1 - \varepsilon_t) \sqrt{\varepsilon_t / (1 - \varepsilon_t)}$$

$$= Z_{t-1} \times 2 \sqrt{\varepsilon_t (1 - \varepsilon_t)} \quad Z_{T+1} = N \prod_{t=1}^T 2 \sqrt{\varepsilon_t (1 - \varepsilon_t)}$$

$$\text{Training Data Error Rate} \leq \prod_{t=1}^T \frac{2 \sqrt{\varepsilon_t (1 - \varepsilon_t)}}{<1} \quad \text{Smaller and smaller}$$

Z_1 的权重就是每一笔初试权重的和N，然后这里的 Z_t 就是要根据 Z_{t-1} 来求出；

对于分类正确的，用 Z_{t-1} 乘以 $\exp(\alpha_t)$ 乘以 ε_t ，对于分类错误的就乘以 $\exp(-\alpha_t)$ 再乘以 $1 - \varepsilon_t$ 。

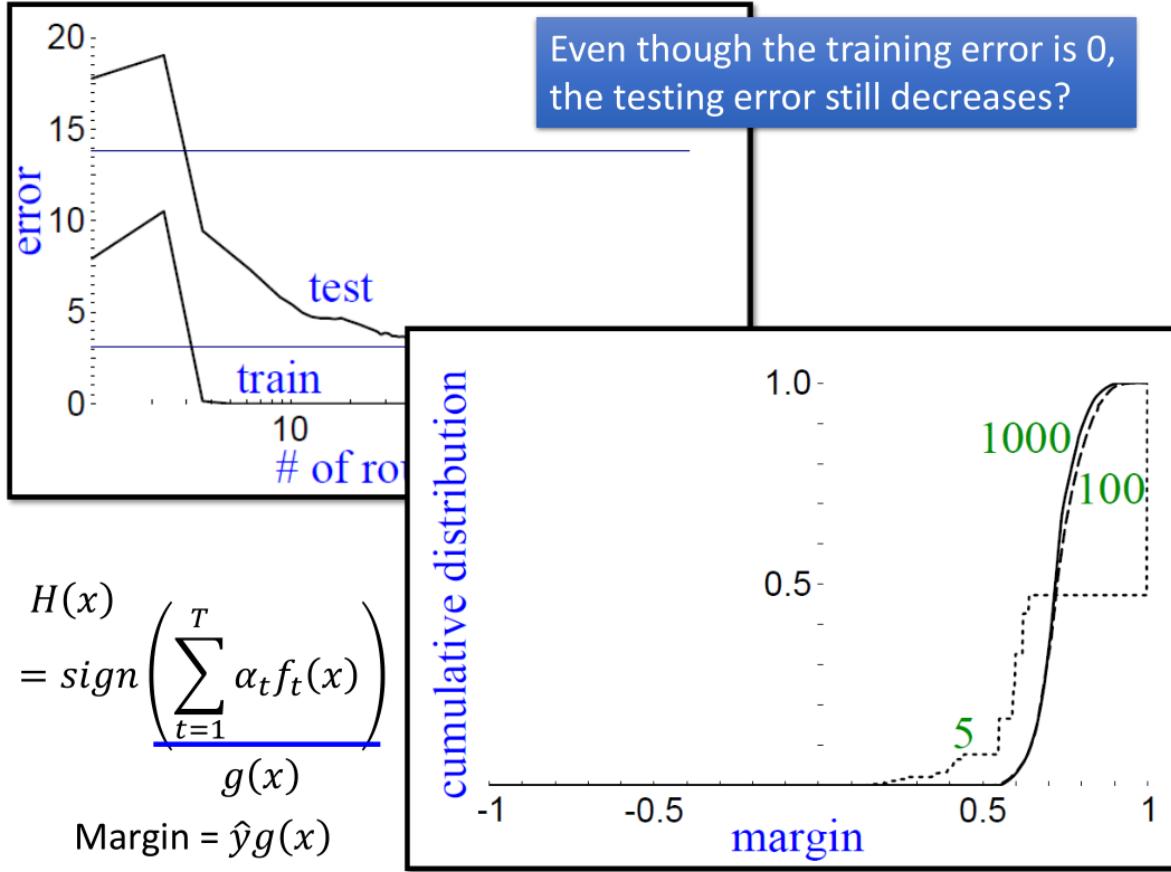
然后再把 α_t 代入到这个式子中化简得到 $Z_{t-1} \times 2 \sqrt{\varepsilon_t (1 - \varepsilon_t)}$

其中， ε_t 是错误率，肯定小于0.5，所以 $2 \sqrt{\varepsilon_t (1 - \varepsilon_t)}$ 当 $\varepsilon_t = 0.5$ 时，最大值为1，所以 Z_t 小于等于 Z_{t-1} 。

Z_{T+1} 就是N乘以T个 $2 \sqrt{\varepsilon_t (1 - \varepsilon_t)}$ 连乘。

这样的一来训练数据的错误率的upper bound就会越来越小。

Margin



其中图中x轴是训练的次数，y轴是错误大小，从这张图我们发现训练数据集上的错误率其实很快就变成了0，但是在 testing data 上的 error 仍然可以继续下降。

我们把 $\hat{y}g(x)$ 定义为margin，我们希望它们是同号，同时不只希望它同号，希望它相乘以后越大越好

原因：图中是5, 100, 1000个权重的分类器结合在一起时margin的分布图，当5个分类器结合的时候，其实margin已经大于0了，但是当增加弱分类器的数量的时候，margin还会一直变大，增加 margin 的好处是让你的方法比较 robust，可以在 testing set 上得到比较好的 performance。

为什么margin会增加？

Large Margin?

$$H(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t f_t(x) \right)$$

$$g(x)$$

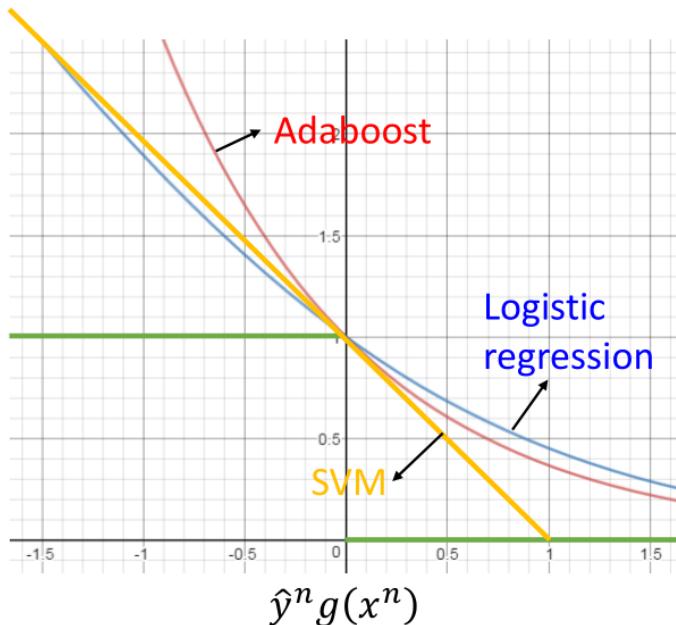
Training Data Error Rate =

$$= \frac{1}{N} \sum_n \delta(H(x^n) \neq \hat{y}^n)$$

$$\leq \frac{1}{N} \sum_n \exp(-\hat{y}^n g(x^n))$$

$$= \prod_{t=1}^T 2 \sqrt{\epsilon_t (1 - \epsilon_t)}$$

Getting smaller and smaller as T increase



该图是 $\hat{y}^n g(x^n)$ 的函数图像，红色的线就是AdaBoost的目标函数，从图中可以看出AdaBoost的在为 $\hat{y}^n g(x^n) > 0$ 时，error 并不是 0，它可以把 $\hat{y}^n g(x^n)$ 再往右边推然后得到更小的 error，依然能不断的下降，也就是让 $\hat{y}^n g(x^n)$ (margin)能不断增大，得到更小的错误。

Logistic Regression和SVM也可以做到同样的效果。

Experiment: Function of Miku

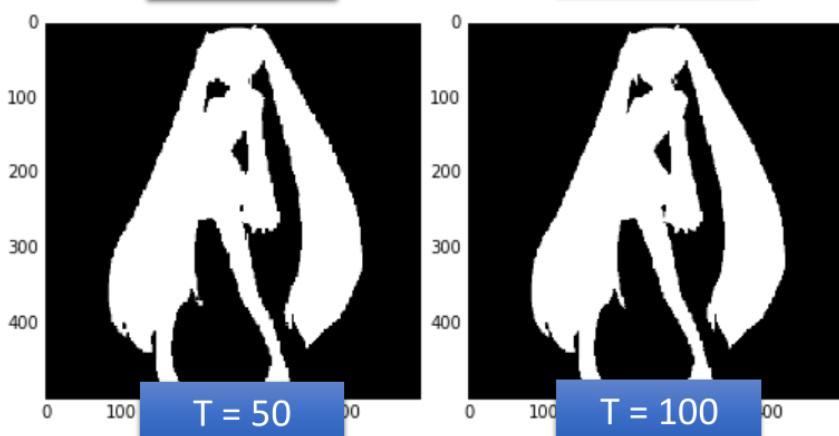
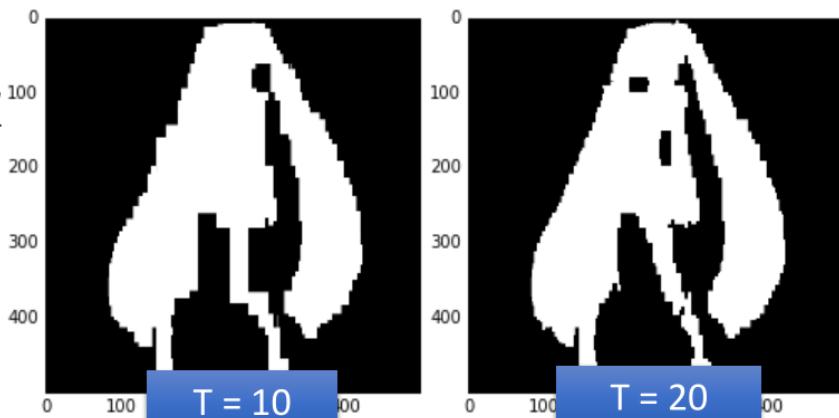
Experiment:

Function of Miku

Adaboost

+Decision Tree

(depth = 5)



本来深度是5的决策树是不能做好初音的分类（只能通过增加深度来进行改进），但是现在有了AdaBoost的决策树是互补的，所以用AdaBoost就可以很好的进行分类。T代表AdaBoost运行次数，图中可知用AdaBoost，100棵树就可以很好的对初音进行分类。

Gradient Boosting

Initial function $g_0(x) = 0$

For t = 1 to T:

- Find a function $f_t(x)$ and α_t to improve $g_{t-1}(x)$
 - $g_{t-1}(x) = \sum_{i=1}^{t-1} \alpha_i f_i(x)$
 - $g_t(x) = g_{t-1}(x) + \alpha_t f_t(x)$

Output: $H(x) = \text{sign}(g_T(x))$

What is the learning target of $g(x)$?

$$\text{Minimize } L(g) = \sum_n l(\hat{y}^n, g(x^n)) = \sum_n \exp(-\hat{y}^n g(x^n))$$

Gradient Boosting是Boosting的更泛化的一个版本。

具体步骤：

- 初始化一个 $g_0(x) = 0$,
- 现在进行很多次的迭代，找到一组 $f_t(x)$ 和 α_t 来共同改进 $g_{t-1}(x)$
 - $g_{t-1}(x)$ 就是之前得到所有的 $f(x)$ 和 α 乘积的和
 - 把找到的一组 $f_t(x)$ 和 α_t 相乘（与 $g_{t-1}(x)$ 互补）加上原来的 $g_{t-1}(x)$ 得到新的 $g_t(x)$ ，这样 $g_t(x)$ 就比原来的 $g_{t-1}(x)$ 更好
- 经过T次迭代，得到的 $H(x)$

这里的cost function是 $L(g) = \sum_n l(\hat{y}^n, g(x^n))$ ，其中 l 用来衡量 \hat{y}^n 和 $g(x^n)$ 的差异（比如说可以用 Cross Entropy 或 Mean Square Error 等等）这里定义成了 $\exp(-\hat{y}^n g(x^n))$ 。

接下来我们要最小化损失函数，我们就需要用梯度下降来更新每个 $g(x)$

- Find $g(x)$, minimize $L(g) = \sum_n \exp(-\hat{y}^n g(x^n))$
 - If we already have $g(x) = g_{t-1}(x)$, how to update $g(x)$?

Gradient Descent:

$$g_t(x) = g_{t-1}(x) - \eta \frac{\partial L(g)}{\partial g(x)} \Big|_{g(x) = g_{t-1}(x)}$$

$\sum_n \exp(-\hat{y}^n g_{t-1}(x^n))(\hat{y}^n)$

$$g_t(x) = g_{t-1}(x) + \alpha_t f_t(x)$$

从梯度下降角度考虑：上图式子中，我们需要用函数 $g(x)$ 对 $L(g)$ 求梯度，然后用这个得到的梯度去更新 g_{t-1} 得到新的 g_t

这里对 $L(g)$ 求梯度的函数 $g(x)$ 就是可以想成每一点就是一个参数，那其实 $g(x)$ 就是一个 vector
 $\begin{bmatrix} g(x_1) \\ g(x_2) \\ \dots \end{bmatrix}$ ，通过调整参数就能改变函数的形状，这样就可以对 $L(g)$ 做偏微分。

从Boosting角度考虑，红色框的两部分应该是同方向的，如果 $f_t(x)$ 和其方向是一致的话，那么就可以把 $f_t(x)$ 加上 $g_{t-1}(x)$ ，就可以让新的损失减少。

我们希望 $f_t(x)$ 和 $\sum_n \exp(-\hat{y}^n g_t(x^n))(\hat{y}^n)$ 方向越一致越好。所以我们希望maximize两个式子相乘，保证这两个式子方向一致。

对于得到的新式子，可以想成对每一笔 training data 都希望 \hat{y} 跟 f_t 他们是同号的，然后每一笔 training data 前面都乘上了一个 weight $\exp(-\hat{y}^n g_{t-1}(x^n))$

经过计算之后发现这个权重恰好就是AdaBoost上的权重

$$f_t(x) \longleftrightarrow \sum_n \exp(-\hat{y}^n g_t(x^n))(\hat{y}^n)$$

Same direction

We want to find $f_t(x)$ maximizing

$$\sum_n \underbrace{\exp(-\hat{y}^n g_{t-1}(x^n))}_{\text{example weight } u_t^n} (\hat{y}^n) f_t(x^n)$$

Minimize Error
Same sign

$$u_t^n = \exp(-\hat{y}^n g_{t-1}(x^n)) = \exp\left(-\hat{y}^n \sum_{i=1}^{t-1} \alpha_i f_i(x^n)\right)$$

$$= \prod_{i=1}^{t-1} \exp(-\hat{y}^n \alpha_i f_i(x^n))$$

Exactly the weights we obtain
in Adaboost

这里找出来的 $f_t(x)$, 其实也就是AdaBoost找出来的 $f_t(x)$, 所以用AdaBoost找一个弱的分类器 $f_t(x)$ 的时候, 就相当于用梯度下降更新损失, 值得损失会变小。

- Find $g(x)$, minimize $L(g) = \sum_n \exp(-\hat{y}^n g(x^n))$

$$g_t(x) = g_{t-1}(x) + \alpha_t f_t(x)$$

α_t is something like learning rate

Find α_t minimizing $L(g_{t+1})$

$$\begin{aligned} L(g) &= \sum_n \exp(-\hat{y}^n (g_{t-1}(x) + \alpha_t f_t(x))) \\ &= \sum_n \exp(-\hat{y}^n g_{t-1}(x)) \exp(-\hat{y}^n \alpha_t f_t(x)) \\ &= \sum_{\hat{y}^n \neq f_t(x)} \exp(-\hat{y}^n g_{t-1}(x^n)) \exp(\alpha_t) \\ &\quad + \sum_{\hat{y}^n = f_t(x)} \exp(-\hat{y}^n g_{t-1}(x^n)) \exp(-\alpha_t) \end{aligned}$$

Find α_t such that

$$\frac{\partial L(g)}{\partial \alpha_t} = 0$$

$$\alpha_t = \ln \sqrt{(1 - \varepsilon_t) / \varepsilon_t}$$

Adaboost!

Gradient Boosting 里面, $f_t(x)$ 是一个 classifier, 在找 $f_t(x)$ 的过程中运算量可能就是很大的, 甚至如果 $f_t(x)$ 是个 Neural Network, 要把 $f_t(x)$ 找出来的时候本身就需要很多次的 Gradient Descent 的 iteration。

由于求 $f_t(x)$ 是很不容易才找到的, 所以我们这里就会给 $f_t(x)$ 配一个最好的 α_t , 把 $f_t(x)$ 的价值发挥到最大。

α_t 有点像学习率, 但是这里我们固定 $f_t(x)$, 穷举所有的 α_t , 找到一个 α_t 使得 $g_t(x)$ 的损失更小。

实际中不可能穷举, 就是求解一个 optimization 的 problem, 找出一个 α_t , 让 $L(g)$ 最小, 这里用计算偏微分的方法求极值。巧合的是找出来的 α_t 就是 $\alpha_t = \ln \sqrt{(1 - \varepsilon_t) / \varepsilon_t}$ 。

所以 Adaboost 整件事情, 就可以想成它也是在做 Gradient Descent。只是 Gradient 是一个 function。

Gradient Boosting 有一个好的地方是, 可以任意更改 Objective Function, 创造出不一样的 Boosting。

Stacking

为了让 performance 再提升, 就要把四个人的 model combine 起来, 把一笔数据 x 输入到四个不同的模型中, 然后每个模型输出一个 y , 然后用 Majority Vote 决定出最好的 (对于分类问题)。

但是有个问题就是并不是所有系统都是好的, 有些系统会比较差, 但是如果采用之前的设置低权重的方法又会伤害小毛的自尊心, 这样我们就提出一种方法:

Stacking



as new feature

把得到的system 的 output 当做feature输入到一个classifier 中，然后再决定最终的结果。

这个最终的 classifier 就不需要太复杂，最前面如果都已经用好几个 Hidden Layer 的 Neural Network 了，也许 final classifier 就不需要再好几个 Hidden Layer 的 Neural Network，它可以只是 Logistic Regression 就行了。

那在做这个实验的时候要注意，我们会把有 label 的 data 分成 training set 跟 validation set。在做 Stacking 的时候要把 training set 再分成两部分，一部分的 training set 拿来 learn 这些 classifier，另外一部分的 training data 拿来 learn 这个 final classifier。

有的要做 Stacking 的前面 classifier，它可能只是 fit training data 的 overfit model。如果 final classifier 的 training data 跟这些 system 用的 training data 是同一组的话，就会因为这个 model 在 training set 上正确率很高而给其很高的权重。所以在 train final classifier 的时候必须要用另外一笔 training data 来 train final classifier，不能跟前面 train system 的 classifier 一样。

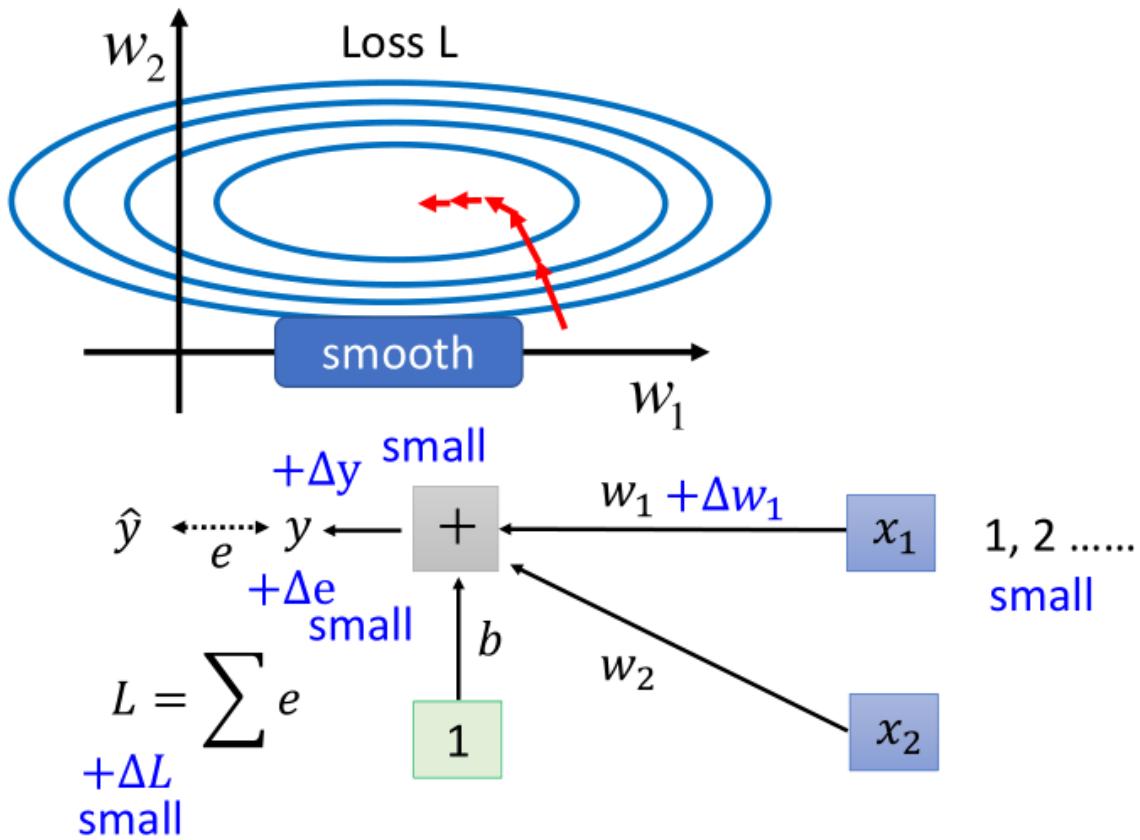
Batch Normalization

很快地介绍一下 Batch Normalization 这个技术。

Changing Landscape

我们之前讲过 error surface 如果很崎岖的时候，它比较难 train，那我们能不能够直接把山铲平，让它变得比较好 train 呢？Batch Normalization 就是其中一个，把山铲平的想法。

我们在讲 optimization 的时候，我们一开始就跟大家讲说，不要小看 optimization 这个问题，有时候就算你的 error surface 是 convex，它就是一个碗的形状，都不见得很好 train。



那我们举的例子就是，假设你的两个参数，它们对 Loss 的斜率差别非常大，在 w_1 这个方向上面，你的斜率变化很小，在 w_2 这个方向上面斜率变化很大，你今天如果是固定的 learning rate，你可能很难得到好的结果，所以我们才说你需要 adaptive 的 learning rate，你需要用 Adam 等等比较进阶的 optimization 的方法，才能够得到好的结果。

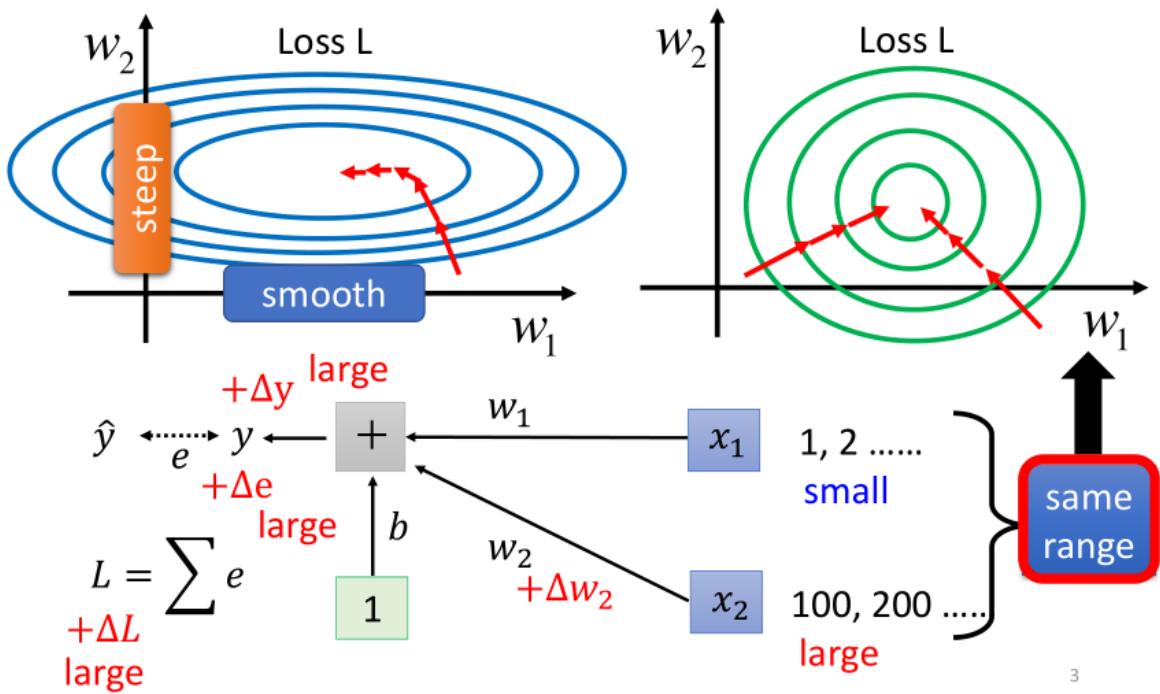
那现在我们要从另外一个方向想，直接把难做的 error surface 把它改掉，看能不能够改得好做一点。

那在做这件事之前，也许我们第一个要问的问题就是，有这一种状况， w_1 跟 w_2 它们的斜率差很多的这种状况，到底是从什么地方来的。

那我们这边就是举一个例子，假设我现在有一个非常非常非常简单的 model，它的输入是 x_1 跟 x_2 ， x_1 跟 x_2 它对应的参数就是 w_1 跟 w_2 ，它是一个 linear 的 model，没有 activation function， w_1 乘 x_1 ， w_2 乘 x_2 加上 b 以后就得到 y ，然后会计算 y 跟 \hat{y} 之间的差距当做 e ，把所有 training data 的 e 加起来呢，就是你的 Loss，你希望去 minimize 你的 Loss。

那什么样的状况我们会产生像上面这样子，比较不好 train 的 error surface 呢？

当我们对 w_1 有一个小小的改变，比如说加上 Δw_1 的时候，那这个 L 也会有一个改变，那什么时候 w_1 的改变会对 L 的影响很小呢，什么时候 w_1 这边的变化，它在 error surface 上的斜率会很小呢？



3

一个可能性是当你的 input 很小的时候，假设 x_1 的值都很小，假设 x_1 的值在不同的 training example 里面，它的值都很小。那因为 x_1 是直接乘上 w_1 ，如果 x_1 的值都很小， w_1 有一个变化的时候，它对 y 的影响也是小的，对 e 的影响也是小的，它对 L 的影响就会是小的。

所以如果 w_1 接的 input 它的值都很小，那就会产生这边这样的 case，你在 w_1 上面的变化对大 L 的影响是小的。

反之呢，如果今天是 x_2 的话，假设 x_2 它的值都很大，那假设 x_2 的值都很大，当你的 w_2 有一个小小的变化的时候，虽然 w_2 这个变化可能很小，但是因为它乘上了 x_2 ， x_2 的值很大，那 y 的变化就很大，那 e 的变化就很大，那 L 的变化就会很大，就会导致我们在 w 这个方向上，做变化的时候，我们把 w 改变一点点，那我们的 error surface 就会有很大的变化。

所以你发现说，既然在这个 linear 的 model 里面，当我们 input 的 feature，每一个 dimension 的值，它的 scale 差距很大的时候，我们就可能产生像这样子的 error surface，就可能产生不同方向，它的斜率非常不同的，它的坡度非常不同的 error surface。所以我们有没有可能给不同的 dimension，feature 里面不同的 dimension，让它有同样的数值的范围？

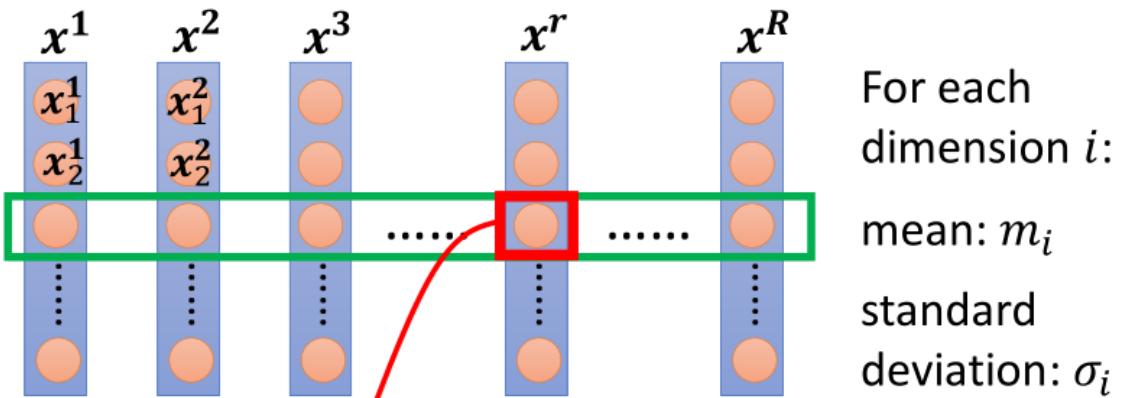
如果我们可以给不同的 dimension，同样的数值范围的话，那我们可能就可以制造比较好的 error surface，让 training 变得比较容易一点。

那怎么让不同的 dimension，有类似的有接近的数值的范围呢，其实有很多不同的方法。那这些不同的方法，往往就合起来统称为 Feature Normalization。

Feature Normalization

那我以下所讲的方法只是，Feature Normalization 的一种可能性，它并不是 Feature Normalization 的全部。

你可以说假设 x_1 到 x_R ，是我们所有的训练数据的 feature vector，我们把所有训练数据的 feature vector，统统都集合起来，那每一个 vector 呢， x_1 里面就 x 上标 1 下标 1，代表 x_1 的第一个 element， x 上标 2 下标 1，就代表 x_2 的第一个 element，以此类推。



$$\tilde{x}_i^r \leftarrow \frac{x_i^r - m_i}{\sigma_i}$$

The means of all dims are 0, and the variances are all 1

In general, feature normalization makes gradient descent converge faster.

4

那我们把不同 feature vector, 同一个 dimension 里面的数值, 把它取出来, 然后去计算某一个 dimension 的 mean。

那我们现在计算的是第 i 个 dimension, 而它的 mean 就是 m_i 。

我们计算第 i 个 dimension 的, standard deviation, 我们用 σ_i 来表示它。

那接下来我们就可以做一种 normalization, 那这种 normalization 呢, 其实叫做标准化, 其实叫 standardization, 不过我们这边呢, 就等一下都统称 normalization 就好了。

那我们怎么做 normalization? 我们就是把这个 x , 把这边的某一个数值减掉 mean, 除掉 standard deviation, 得到新的数值叫做 \tilde{x} 。

得到新的数值以后, 再把新的数值塞回去。

我们用这个 tilde, 来代表有被 normalize 后的数值。

做完 normalize 以后, 这个 dimension 上面的数值就会平均是 0, 然后它的 variance 就会是 1, 所以这一排数值, 它的分布就都会在 0 上下。

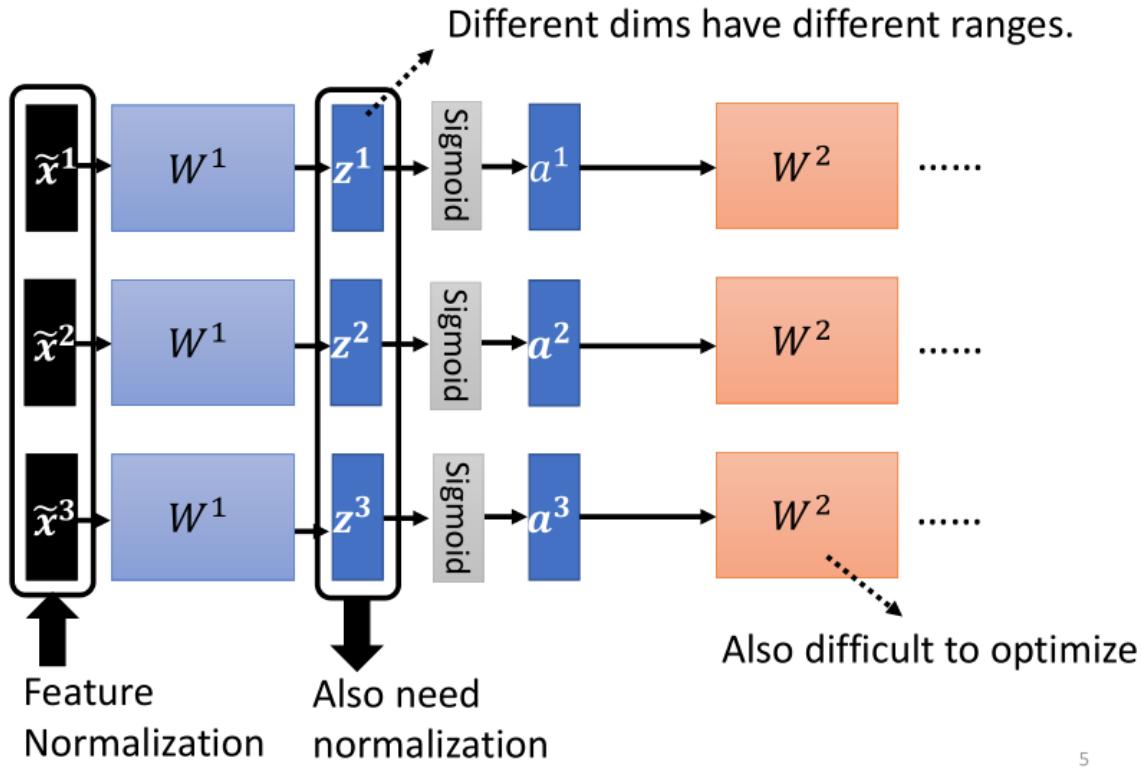
那你对每一个 dimension, 每一个 dimension, 都做一样的 normalization, 把他们变成 mean 接近 0, variance 是 1, 那你就会发现说所有的数值, 所有 feature 不同 dimension 的数值, 都在 0 上下, 那你可能就可以制造一个, 比较好的 error surface。

所以像这样子 Feature Normalization 的方式, 往往对你的 training 有帮助, 它可以让你在做 gradient descent 的时候, 这个 gradient descent, 它的 Loss 收敛更快一点, 可以让你的 gradient descent, 它的训练更顺利一点。这个是 Feature Normalization。

Considering Deep Learning

当然 Deep Learning 可以做 Feature Normalization, 得到 \tilde{x} 以后, 把 \tilde{x}_1 通过第一个 layer 得到 z_1 , 那你有可能通过 activation function, 不管是选 Sigmoid 或者 ReLU 都可以。然后再得到 a_1 , 然后再通过下一层等等, 那就看你有几层 network 你就做多少的运算。所以每一个 x 都做类似的事情。

但是如果我们要进一步来想的话，对 w_2 来说，这边的 $a_1 a_3$ 这边的 $z_1 z_3$ ，其实也是另外一种 input，如果这边 \tilde{x} ，虽然它已经做 normalize 了，但是通过 w_1 以后它就没有做 normalize，如果 \tilde{x} 通过 w_1 得到是 z_1 ，而 z_1 不同的 dimension 间，它的数值的分布仍然有很大的差异的话，那我们要 train w_2 第二层的参数，会不会也有困难呢？所以这样想起来，我们也应该要对这边的 a 或对这边的 z ，做 Feature Normalization。对 w_2 来说，这边的 a 或这边的 z 其实也是一种 feature，我们应该要对这些 feature 也做 normalization。



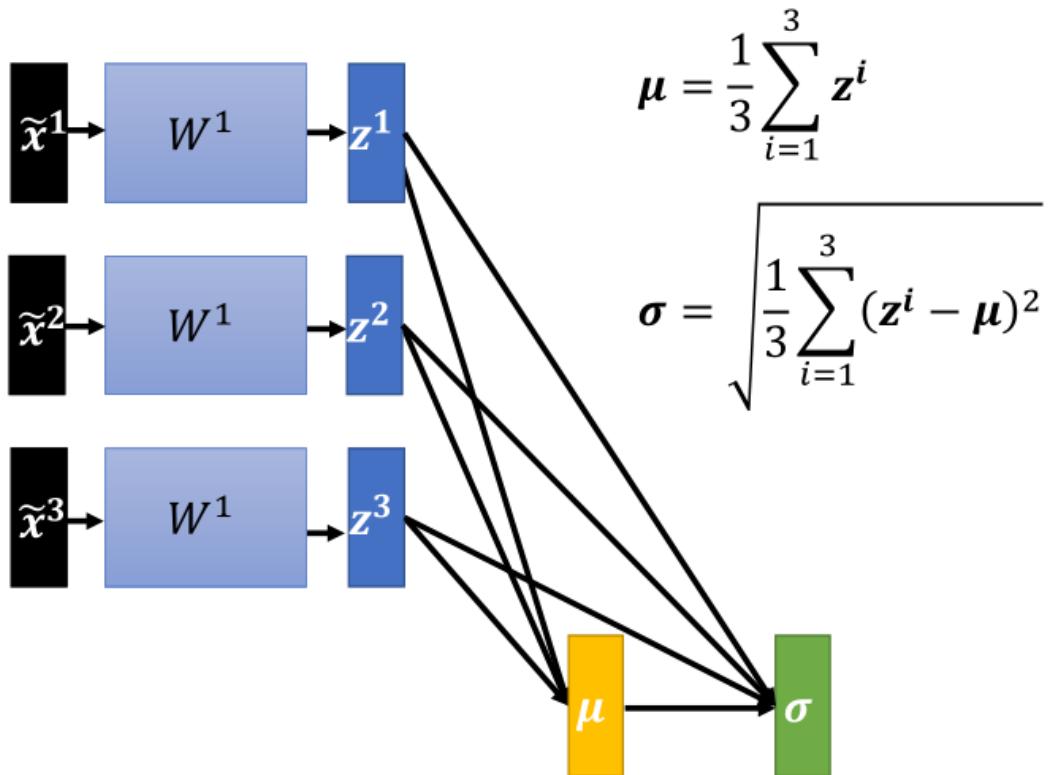
5

但这边有人就会问一个问题，应该要在 activation function 之前，做 normalization，还是要在 activation function 之后，做 normalization 呢？

在实践上这两件事情其实差异不大，所以你对 z 做 Feature Normalization，或对 a 做 Feature Normalization，其实都可以。

那如果你选择的是 Sigmoid，那可能比较推荐对 z 做 Feature Normalization，因为 Sigmoid 是一个 s 的形状，那它在 0 附近斜率比较大，所以如果你对 z 做 Feature Normalization，把所有的值都挪到 0 附近，那你到时候算 gradient 的时候，算出来的值会比较大。

那不过因为你不见得是跟 sigmoid，所以你也不一定要把 Feature Normalization 放在 z 这个地方，如果是别的，也许你选 a 也会有好的结果，也说不定。In general 而言，这个 normalization，要放在 activation function 之前，或之后都是可以的，在实践上，可能没有太大的差别。



那我们这边就是对 z ，做一下 Feature Normalization。那怎么对 z 做 Feature Normalization 呢？那你就把 z ，想成是另外一种 feature

我们这边有 $z_1 z_2 z_3$ ，我们就把 $z_1 z_2 z_3$ 拿出来，算一下它的 mean，standard deviation。

这个 notation 有点 abuse，这边的平方就是指，对每一个 element 都去做平方，然后再开根号，这边开根号指的是对每一个 element，向量里面的每一个 element，都去做开根号，得到 σ 。

就把这三个 vector，里面的每一个 dimension，都去把它的 μ 算出来，把它的 σ 算出来。从 $z_1 z_2 z_3$ ，算出 μ ，算出 σ 。接下来呢，你就把这边的每一个 z ，都去减掉 μ 除以 σ ，你把 z_i 减掉 μ ，除以 σ ，就得到 \tilde{z}_i 。

那这边的 μ 跟 σ ，它都是向量，所以这边这个除，它的 notation 有点 abuse。这边的除的意思是说，element wise 的相除，就是 z_i 减 μ ，它是一个向量，所以分子的地方是一个向量，分母的地方也是一个向量，把这个两个向量，它们对应的 element 的值相除，是我这边这个除号的意思。这边得到 \tilde{z} 。

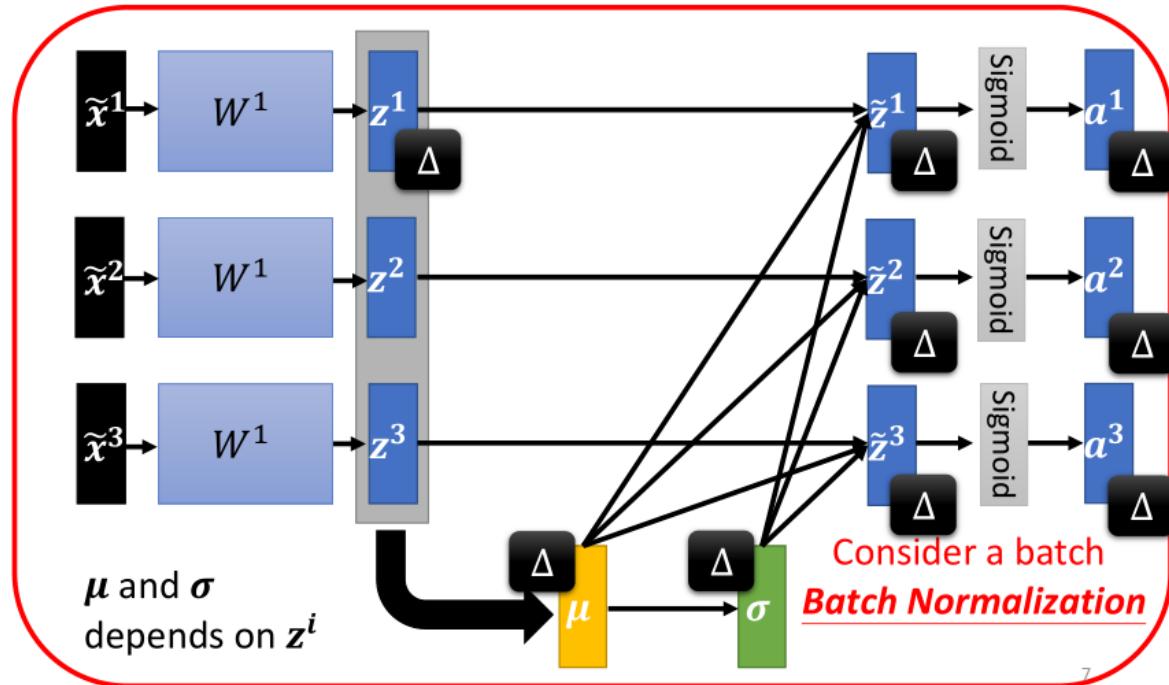
所以我们就是把 z_i 减 μ 除以 σ ，做 Feature Normalization 得到 \tilde{z}_i 。

那接下来通过 activation function，得到其他 vector，然后再通过，再去通过其他 layer 等等，这样就可以了。这样你就等于对 $z_1 z_2 z_3$ ，做了 Feature Normalization，变成 \tilde{z}_i 。

Considering Deep Learning

This is a large network!

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$



在这边有一件有趣的事情，这件事情是这样子的。这边的 μ 跟 σ ，它们其实都是根据 z_1 z_2 z_3 算出来的，所以这边 z_1 ，它本来，如果我们没有做 Feature Normalization 的时候，你改变了 z_1 的值，你会改变这边 a 的值，但是现在，当你改变 z_1 的值的时候， μ 跟 σ 也会跟着改变， μ 跟 σ 改变以后， z_2 的值 a_2 的值， z_3 的值 a_3 的值，也会跟着改变。所以之前，我们每一个 \tilde{x}_1 \tilde{x}_2 \tilde{x}_3 ，它是独立分开处理的，但是我们在做 Feature Normalization 以后，这三个 example，它们变得彼此关联了，我们这边 z_1 只要有改变，接下来 z_2 , a_2 , z_3 , a_3 ，也都会跟着改变。

所以当你有做 Feature Normalization 的时候，你要把这一整个 process，就是有收集一堆 feature，把这堆 feature 算出 μ 跟 σ 这件事情，当做是 network 的一部分。

也就是说，你现在有一个比较大的 network，你之前的 network，都只吃一个 input，得到一个 output，现在你有一个比较大的 network，这个大的 network，它是吃一堆 input，用这堆 input 在这个 network 里面，要算出 μ 跟 σ ，然后接下来产生一堆 output。

那这一段只可会意不可言传这样子，不知道你听不听得懂这一段的意思。就是现在不是一个 network 处理一个 example，而是有一个巨大的 network，它处理一把 example，用这把 example，还要算个 μ 跟 σ ，得到一把 output。

那这边就会有一个问题了，因为你的训练资料里面，你的 data 非常多，现在一个 data set，benchmark corpus 都上百万笔资料，你哪有办法一次把上百万笔资料，丢到一个 network 里面。那你那个 GPU 的 memory，根本没有办法，把它整个 data set 的 data 都 load 进去。

Batch normalization

所以怎么办？在实作的时候，你不会让这一个 network 考虑，整个 training data 里面的所有 example。你只会考虑一个 batch 里面的 example。

举例来说，你 batch 设 64，那你这个巨大的 network，就是把 64 笔 data 读进去，算这 64 笔 data 的 μ ，算这 64 笔 data 的 σ ，对这 64 笔 data 都去做 normalization。

因为我们在实作的时候，我们只对一个 batch 里面的 data，做 normalization，所以这招叫做 **Batch Normalization**。

这个就是你常常听到的，Batch Normalization。

那这个 Batch Normalization，显然有一个问题就是，你一定要有一个够大的 batch，你才算得出 μ 跟 σ 。

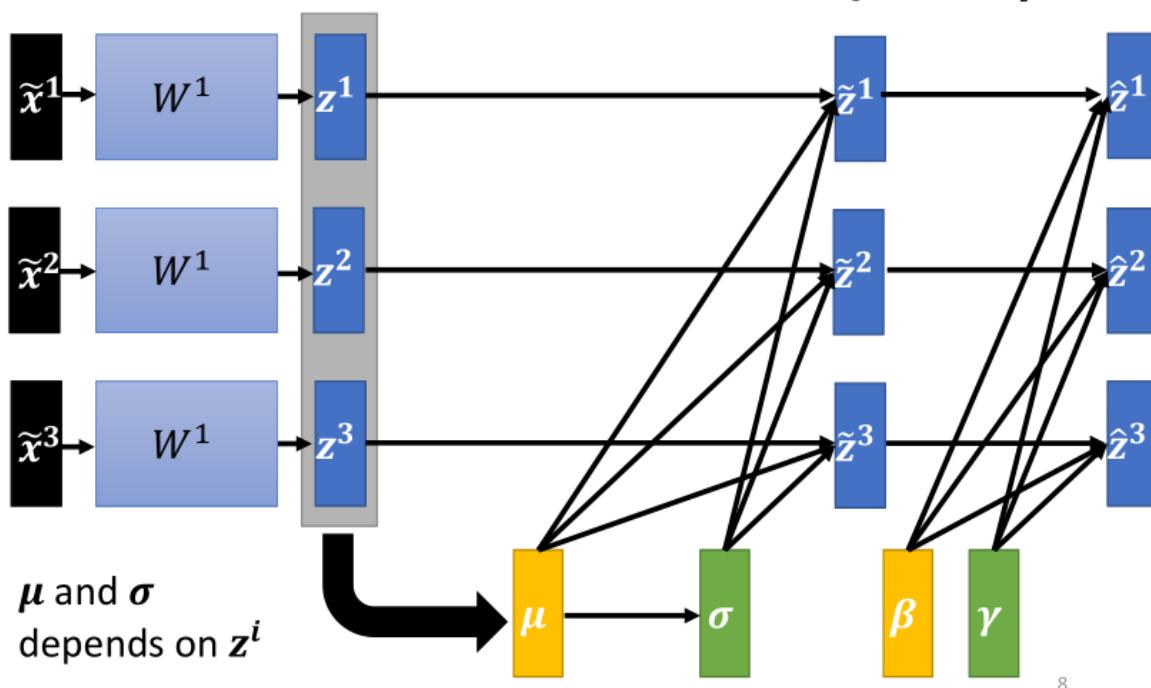
假设你今天，你 batch size 设 1，那你就没有什么 μ 或 σ 可以算，你就会有问题，所以这个 Batch Normalization，是适用于 batch size 比较大的时候。

因为 batch size 如果比较大，也许这个 batch size 里面的 data，就足以表示，整个 corpus 的分布，那这个时候你就可以，把这个本来要对整个 corpus，做 Feature Normalization 这件事情，改成只在一个 batch 做 Feature Normalization 作为 approximation。

Batch normalization

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$

$$\hat{z}^i = \gamma \odot \tilde{z}^i + \beta$$



那在做 Batch Normalization 的时候，往往还会有这样的设计，你算出这个 \tilde{z} 以后，接下来你会把这个 \tilde{z} ，再乘上另外一个向量，叫做 γ ，这个 γ 也是一个向量，所以你就是把 \tilde{z} 跟 γ 做 element wise 的相乘，再加上 β 这个向量，得到 \hat{z} ，而 β 跟 γ ，你要把它想成是 network 的参数，它是另外再被 learn 出来的。

那为什么要加上 β 跟 γ 呢？那是因为有人可能会觉得说，如果我们做 normalization 以后，那这边的 \tilde{z} ，它的平均呢，就一定是 0，今天如果平均是 0 的话，就是给那 network 一些限制，那也许这个限制会带来什么负面的影响，所以我们把 β 跟 γ 加回去，然后让 network 呢，现在它的 hidden layer 的 output 呢，不需要平均是 0。如果它想要平均不是 0 的话，它就自己去 learn 这个 β 跟 γ ，来调整一下输出的分布，来调整这个 \hat{z} 的分布。

但讲到这边又会有人问说，刚才不是说做 Batch Normalization 就是，为了要让每一个不同的 dimension，它的 range 都是一样，我们才做这个 normalization 吗，现在如果加去乘上 γ ，再加上 β ，把 γ 跟 β 加进去，这样不会不同 dimension 的分布，它的 range 又都不一样了吗？

有可能，但是你实际上在训练的时候，你会把这个 γ 的初始值就都设为 1

所以 γ 是一个里面的值，一开始其实是一个里面的值，全部都是 1 的向量，那 β 是一个里面的值，全部都是 0 的向量。

所以让你的 network 在一开始训练的时候，每一个 dimension 的分布，是比较接近的。

也许训练到后来，你已经训练够长的一段时间，已经找到一个比较好的 error surface，走到一个比较好的地方以后，那再把 γ 跟 β 慢慢地加进去。所以加 Batch Normalization，往往对你的训练是有帮助的。

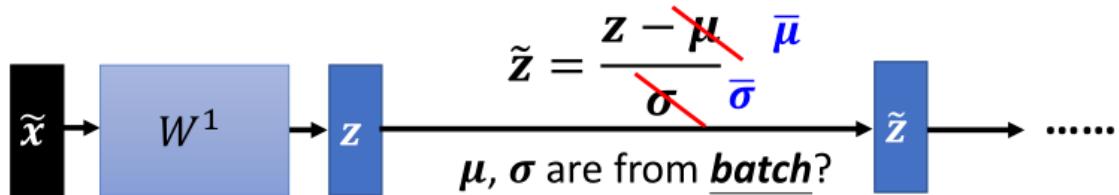
Testing

那接下来就要讲 testing 的部分了，刚才讲的都是 training 的部分，还没有讲到 testing 的部分 testing，有时候又叫 inference，所以有人在文件上看到有人说做个 inference，inference 指的就是 testing。

这个 Batch Normalization 在 inference，或是 testing 的时候呢，会有问题，会有什么样的问题呢？

假设你真的有系统上线，你是一个真正的在线的 application，你可以说，比如说你的 batch size 设 64，我一定要等 64 笔数据都进来，我才一次做运算吗？这显然是不行的。如果你是一个在线的服务，一笔资料进来，你就要每次都做运算，你不能等说，我累积了一个 batch 的资料，才开始做运算。

但是在做 Batch Normalization 的时候，我们今天，一个 \tilde{x} ，一个 normalization 过的 feature 进来，然后你有一个 z ，要减掉 μ 跟除 σ ，那这个 μ 跟 σ ，是用一个 batch 的资料算出来的。但如果今天在 testing 的时候，根本就没有 batch，那我们要怎么算这个 μ ，跟怎么算这个 σ 呢？



We do not always have batch at testing stage.

Computing the moving average of μ and σ of the batches during training.

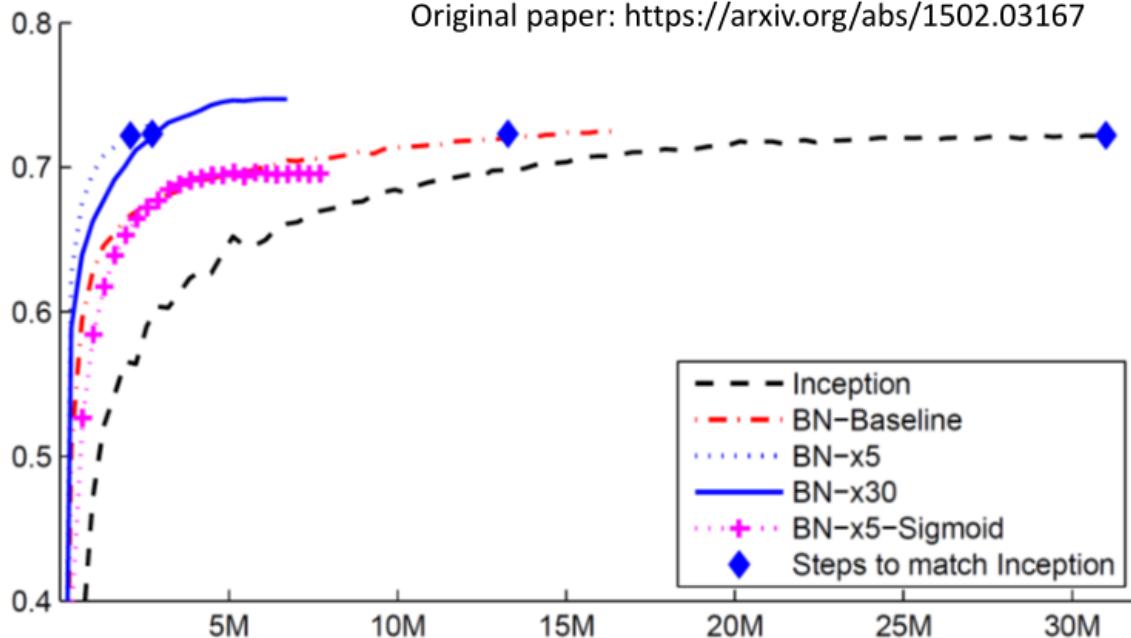
$$\mu^1 \quad \mu^2 \quad \mu^3 \quad \dots \quad \mu^t$$

$$\bar{\mu} \leftarrow p\bar{\mu} + (1-p)\mu^t$$

这个实作上的解法是这个样子的，如果你看那个 PyTorch 的话呢，Batch Normalization 在 testing 的时候，你并不需要做什么特别的处理，PyTorch 帮你处理好了。

PyTorch 是怎么处理这件事的呢？如果你有在做 Batch Normalization 的话，在 training 的时候，你每一个 batch 计算出来的 μ 跟 σ ，他都会拿出来算 moving average，什么意思呢，你每一次取一个 batch 出来的时候，你就会算一个 μ^1 ，取第二个 batch 出来的时候，你就算个 μ^2 ，一直到取第 t 个 batch 出来的时候，你就算一个 μ^t ，接下来你会算一个 moving average，也就是呢，你会把你现在算出来的 μ 的一个平均值，叫做 $\bar{\mu}$ ，乘上某一个 factor，那这也是一个常数，是一个 constant，也是一个 hyper parameter，也是需要调的那种。在 PyTorch 里面，我记得 p 就设 0.1，然后加上 1 减 p ，乘上 μ^t ，然后来更新你的 μ 的平均值。

最后在 testing 的时候，你就不用算 batch 里面的 μ 跟 σ 了，因为 testing 的时候，在真正 application 上，也没有 batch 这个东西，你就直接拿就是 μ 跟 σ 在训练的时候，得到的 moving average， $\bar{\mu}$ 跟 $\bar{\sigma}$ ，来取代这边的 μ 跟 σ ，这个就是 Batch Normalization，在 testing 的时候的运作方式。



那这个是从 Batch Normalization，原始的文献上面截出来的一个实验结果，那在原始的文献上还讲了很多其他的东西。

举例来说，我们今天还没有讲的是，Batch Normalization 用在 CNN 上，要怎么用呢？那你自己去读一下原始的文献，里面会告诉你说，Batch Normalization 如果用在 CNN 上，应该要长什么样子。

那这个是原始文献上面截出来的一个数据，这个横轴，代表的是训练的过程，纵轴代表的是 validation set 上面的 accuracy，那这个黑色的虚线是没有做 Batch Normalization 的结果，它用的是 inception 的 network，就是某一种 network 架构，也是以 CNN 为基础的 network 架构。

总之黑色的这个虚线，它代表没有做 Batch Normalization 的结果，如果有做 Batch Normalization，你会得到红色的这一条虚线，你会发现说，红色这一条虚线，它训练的速度，显然比黑色的虚线还要快很多，虽然最后收敛的结果，你只要给它足够的训练的时间，可能都跑到差不多的 accuracy，但是红色这一条虚线，可以在比较短的时间内，就跑到一样的 accuracy。那这边这个蓝色的菱形，代表说这几个点的那个 accuracy 是一样的。红色的相较于没有做 Batch Normalization 只需要一半或甚至更少的时间，就跑到同样的正确率了。

粉红色的线是 sigmoid function，我们一般都会选择 ReLU，而不是用 sigmoid function，因为 sigmoid function，它的 training 是比较困难的。

但是这边想要强调的点是说，就算是 sigmoid 比较难搞的，加 Batch Normalization 还是 train 的起来。作者说 sigmoid 不加 Batch Normalization，根本连 train 都 train 不起来。

蓝色的实线跟这个蓝色的虚线，是把 learning rate 设比较大一点， $\times 5$ 就是 learning rate 变原来的 5 倍， $\times 30$ 就是 learning rate 变原来的 30 倍。那因为如果你做 Batch Normalization 的话，那你的 error surface，会比较平滑 比较容易训练。所以你就可以把你的 learning rate 呢，设大一点。

那这边有个不好解释的奇怪的地方，就是不知道为什么，learning rate 设 30 倍的时候，是比 5 倍差。作者也没有解释，做 deep learning 就是有时候会产生这种怪怪的，不知道怎么解释的现象就是了，不过作者就是照实，把他做出来的实验结果，呈现在这个图上面。

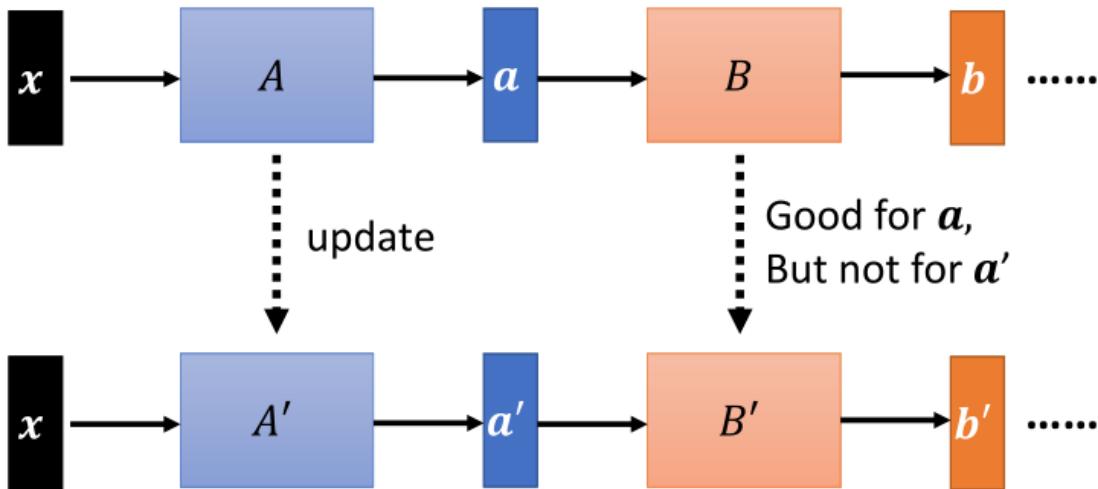
Internal Covariate Shift?

接下来的问题就是，Batch Normalization，它为什么会有帮助呢？在原始的 Batch Normalization 那篇 paper 里面，他提出来一个概念，叫做 internal 的 covariate shift。

covariate shift 这个词汇是原来就有的，internal covariate shift，我认为是 Batch Normalization 的作者自己发明的，他认为今天在 train network 的时候，会有以下这个问题。

How Does Batch Normalization Help Optimization?

<https://arxiv.org/abs/1805.11604>



Batch normalization make a and a' have similar statistics.
Experimental results do not support the above idea.

11

这个问题是这样，network 有很多层， x 通过第一层以后 得到 a ， a 通过第二层以后 得到 b ，那我们今天计算出 gradient 以后，把 A update 成 A' ，把 B 这一层的参数 update 成 B' 。

但是作者认为，现在我们在把 B update 到 B' 的时候，那我们在计算 B ，update 到 B' 的 gradient 的时候，这个时候前一层的参数是 A ，或者是前一层的 output 是小 a ，那当前一层从 A 变成 A' 的时候，它的 output 就从小 a 变成小 a' ，但是我们计算这个 gradient 的时候，我们是根据这个 a 算出来的，所以这个 update 的方向，也许它适合用在 a 上，但不适合用在 a' 上面。

那如果说 Batch Normalization 的话，因为我们每次都有做 normalization，我们就会让 a 跟 a' 呢，它的分布比较接近，也许这样就会对训练有帮助。

但是有一篇 paper 叫 How Does Batch Normalization Help Optimization，就打脸了internal covariate shift 的这一个观点，这篇 paper 从各式各样的方向来告诉你说 internal covariate shift 首先它不一定是 training network 时候的一个问题，然后 Batch Normalization，它会比较好，可能不见得是因为它解决了 internal covariate shift。

那在这篇 paper 里面，他做了很多很多的实验，比如说他比较了训练的时候，这个 a 的分布的变化发现，不管有没有做 Batch Normalization，它的变化都不大。然后他又说就算是变化很大，对 training 也没有太大的伤害。然后他又说，不管你是根据 a 算出来的 gradient，还是根据 a' 算出来的 gradient，方向居然都差不多。

所以他告诉你说，internal covariate shift，可能不是 training network 的时候，最主要的问题。它可能也不是 Batch Normalization 会好的一个的关键，那有关更多的实验，你就自己参见这篇文章。

为什么 Batch Normalization 会比较好呢？那在这篇 How Does Batch Normalization，Help Optimization 这篇论文里面，他从实验上，也从理论上，至少支持了 Batch Normalization，可以改变 error surface，让 error surface 比较不崎岖这个观点。所以这个观点是有理论的支持，也有实验的左证的。

Experimental results (and theoretically analysis) support batch normalization change the landscape of error surface.

and 12 of Appendix B.) This suggests that the positive impact of BatchNorm on training might be somewhat serendipitous. Therefore, it might be valuable to perform a principled exploration of the design space of normalization schemes as it can lead to better performance.

serendipitous (偶然的)

penicillin



这篇文章里面，作者说，如果我们要让 network，这个 error surface 变得比较不崎岖，其实不见得要做 Batch Normalization，感觉有很多其他的方法都可以让 error surface 变得不崎岖，那他就试了一些其他的方法，发现说跟 Batch Normalization performance 也差不多，甚至还稍微好一点。

所以他就讲了下面这句感叹，他觉得 positive impact of batchnorm on training might be somewhat serendipitous

什么是 serendipitous 呢？这个字眼可能可以翻译成偶然的，但偶然并没有完全表达这个词汇的意思，这个词的意思是说，你发现了一个什么意料之外的东西。举例来说，青霉素就是意料之外的发现，有一个人叫做弗莱明，然后他本来想要那个，培养一些葡萄球菌，然后但是因为他实验没有做好，他的那个葡萄球菌被感染了，有一些霉菌掉到他的培养皿里面，然后发现那些培养皿，那些霉菌呢，会杀死葡萄球菌，所以他就发现了青霉素，所以这是一种偶然的发现。

那这篇文章的作者也觉得，Batch Normalization 也像是盘尼西林一样，是一种偶然的发现，但无论如何，它是一个有用的方法。

To learn more

那其实 Batch Normalization，不是唯一的 normalization，normalization 的方法有一把，那这边就是列出了几个比较知名的参考。

- Batch Renormalization
 - <https://arxiv.org/abs/1702.03275>
- Layer Normalization
 - <https://arxiv.org/abs/1607.06450>
- Instance Normalization
 - <https://arxiv.org/abs/1607.08022>
- Group Normalization
 - <https://arxiv.org/abs/1803.08494>
- Weight Normalization
 - <https://arxiv.org/abs/1602.07868>
- Spectrum Normalization
 - <https://arxiv.org/abs/1705.10941>

Deep Learning

Deep Learning

Ups and downs of Deep Learning

- 1958: Perceptron (linear model), 感知机的提出
 - 和Logistic Regression类似，只是少了sigmoid的部分
- 1969: Perceptron has limitation, from MIT
- 1980s: Multi-layer Perceptron, 多层感知机
 - Do not have significant difference from DNN today
- 1986: Backpropagation, 反向传播
 - Hinton propose的Backpropagation
 - 存在problem: 通常超过3个layer的neural network, 就train不出好的结果
- 1989: 1 hidden layer is "good enough", why deep?
 - 有人提出一个理论: 只要neural network有一个hidden layer, 它就可以model出任何的function, 所以根本没有必要叠加很多个hidden layer, 所以Multi-layer Perceptron的方法又坏掉了, 这段时间Multi-layer Perceptron这个东西是受到抵制的
- 2006: RBM initialization(breakthrough), Restricted Boltzmann Machine
 - Deep learning = another Multi-layer Perceptron ? 在当时看来, 它们的不同之处在于在做gradient descent的时候选取初始值的方法如果是用RBM, 那就是Deep learning; 如果没有用RBM, 就是传统的Multi-layer Perceptron
 - 那实际上, RBM用的不是neural network base的方法, 而是graphical model, 后来大家试验得多了发现RBM并没有什么太大的帮助, 因此现在基本上没有人使用RBM做initialization了
 - RBM最大的贡献是, 它让大家重新对Deep Learning这个model有了兴趣 (石头汤)
- 2009: GPU加速的发现
- 2011: start to be popular in speech recognition, 语音识别领域
- 2012: win ILSVRC image competition, Deep learning开始在图像领域流行

实际上, Deep learning跟machine learning一样, 也是“大象放进冰箱”三个步骤

Step 1: define a set of function

Step 2: goodness of function

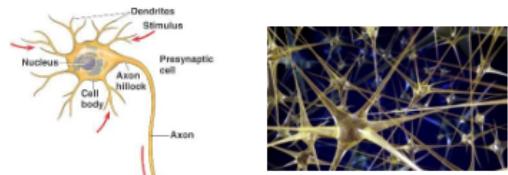
Step 3: pick the best function

Neural Network

Concept

把多个Logistic Regression前后connect在一起, 然后把一个Logistic Regression称之为neuron, 整个称之为neural network

Neural Network



Neural Network

Different connection leads to different network structures

Network parameter θ : all the weights and biases in the “neurons”

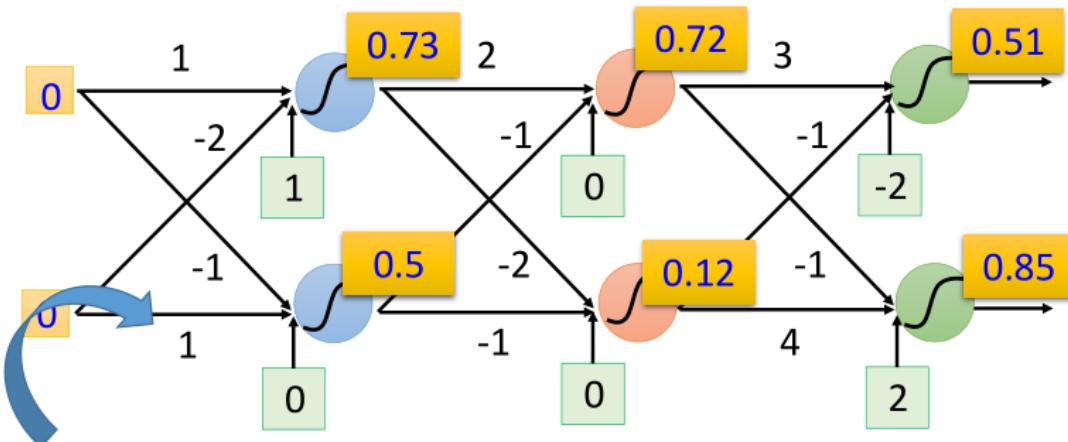
我们可以用不同的方法连接这些neuron，就可以得到不同的structure，neural network里的每一个Logistic Regression都有自己的weight和bias，这些weight和bias集合起来，就是这个network的parameter，我们用 θ 来描述

Fully Connect Feedforward Network

那该怎么把它们连接起来呢？这是需要你手动去设计的，最常见的连接方式叫做Fully Connect Feedforward Network (全连接前馈网络)

如果一个neural network里面的参数weight和bias已知的话，它就是一个function，它的input是一个vector，output是另一个vector，这个vector里面放的是样本点的feature，vector的dimension就是feature的个数

Fully Connect Feedforward Network



This is a function.
Input vector, output vector

$$f\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 0.62 \\ 0.83 \end{bmatrix} \quad f\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0.51 \\ 0.85 \end{bmatrix}$$

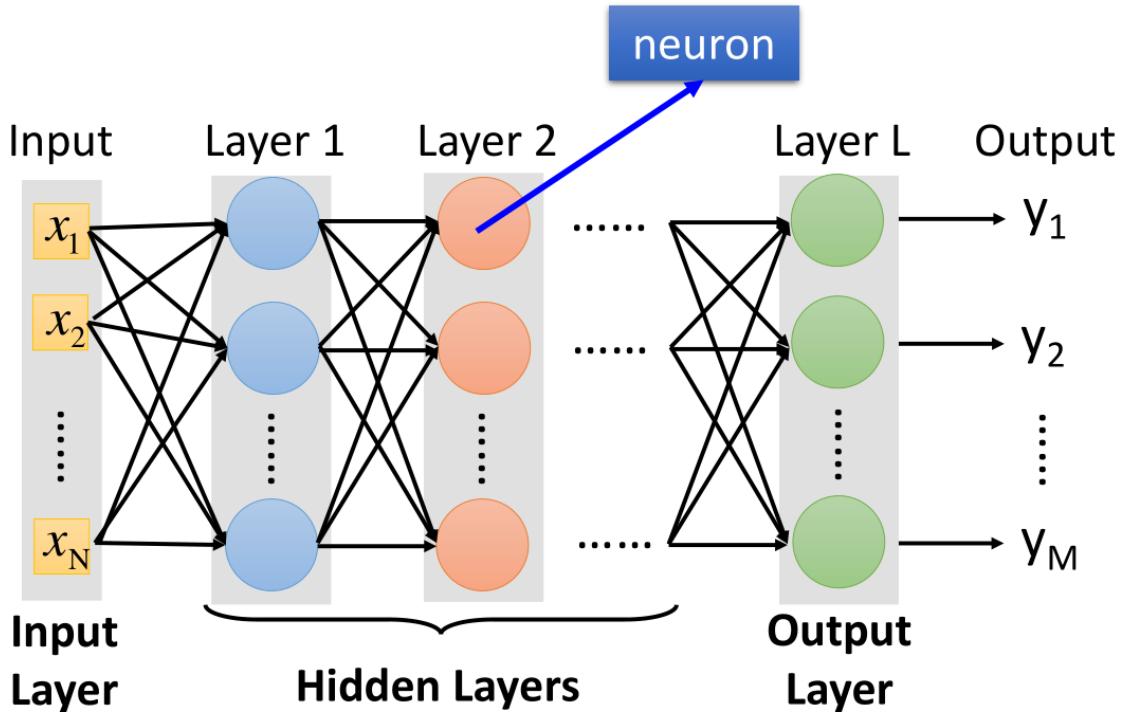
Given network structure, define *a function set*

如果我们还不知道参数，只是定出了这个network的structure，只是决定好这些neuron该怎么连接在一起，这样的一个network structure其实是define了一个function set(model)，我们给这个network设不同的参数，它就变成了不同的function，把这些可能的function集合起来，就是function set

只不过我们用neural network决定function set的时候，这个function set是比较大的，它包含了很多 Logistic Regression、Linear Regression没有办法包含的function

下图中，每一排表示一个layer，每个layer里面的每一个球都代表一个neuron。因为layer和layer之间，所有的neuron都是两两连接，所以它叫**Fully connected**的network；因为现在传递的方向是从layer 1->2->3，由后往前传，所以它叫做**Feedforward network**

- layer和layer之间neuron是两两互相连接的，layer 1的neuron output会连接给layer 2的每一个neuron作为input
- 对整个neural network来说，它需要一个input，这个input就是一个feature的vector，而对layer 1的每一个neuron来说，它的input就是input layer的每一个dimension
- 最后那个layer L，由于它后面没有接其它东西了，所以它的output就是整个network的output
- 这里每一个layer都是有名字的
 - input的地方，叫做**input layer**，输入层(严格来说input layer其实不是一个layer，它跟其他layer不一样，不是由neuron所组成的)
 - output的地方，叫做**output layer**，输出层
 - 其余的地方，叫做**hidden layer**，隐藏层
- 每一个neuron里面的sigmoid function，在Deep Learning中被称为**activation function**，事实上它不见得一定是sigmoid function，还可以是其他function (sigmoid function是从Logistic Regression迁移过来的，现在已经较少在Deep learning里使用了)
- 有很多层layers的neural network，被称为**DNN(Deep Neural Network)**



那所谓的deep，是什么意思呢？有很多层hidden layer，就叫做deep，具体的层数并没有规定，现在只要是neural network base的方法，都被称为Deep Learning。

使用了152个hidden layers的Residual Net(2015)，不是使用一般的Fully Connected Feedforward Network，它需要设计特殊的special structure才能训练这么深的network

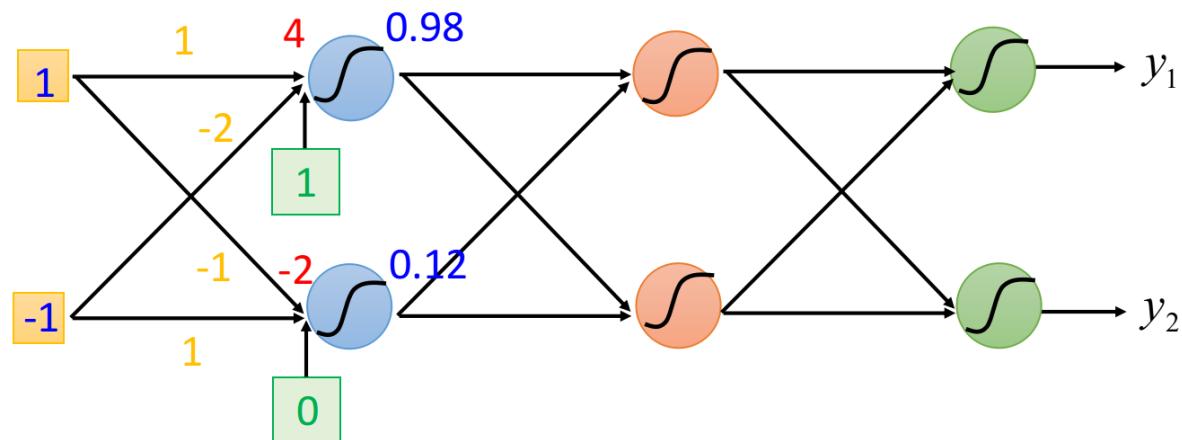
Matrix Operation

network的运作过程，我们通常会用Matrix Operation来表示，以下图为例，假设第一层hidden layers的两个neuron，它们的weight分别是 $w_1 = 1, w_2 = -2, w'_1 = -1, w'_2 = 1$ ，那就可以把它们排成一个matrix: $\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix}$ ，而我们的input又是一个 $2*1$ 的vector: $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$ ，将w和x相乘，再加上bias的vector: $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ ，就可以得到这一层的vector z，再经过activation function得到这一层的output

这里还是用Logistic Regression迁移过来的sigmoid function作为运算

$$\sigma\left(\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} 4 \\ -2 \end{bmatrix}\right) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

Matrix Operation



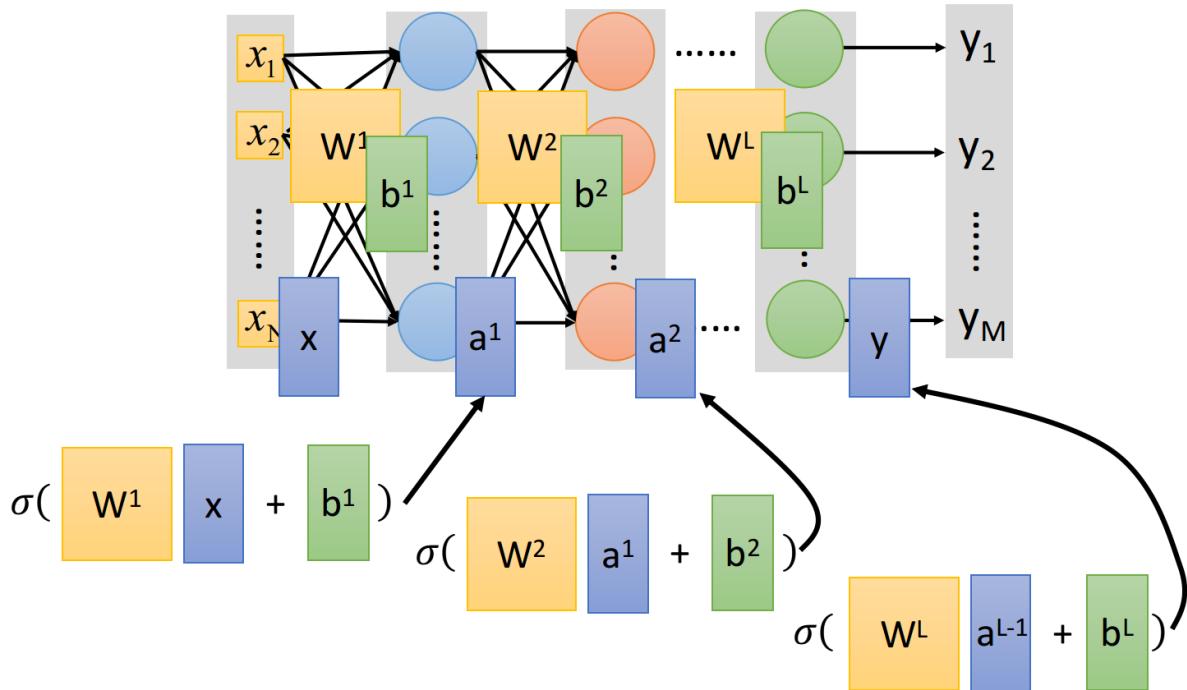
$$\sigma \left(\underbrace{\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{\begin{bmatrix} 4 \\ -2 \end{bmatrix}} \right) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

这里我们把所有的变量都以matrix的形式表示出来，注意 W^i 的matrix，每一行对应的是一个neuron的weight，行数就是neuron的个数，列数就是feature的数量

input x, bias b和output y都是一个列向量，行数是feature的个数，也是neuron的个数

neuron的本质就是把feature transform到另一个space

Neural Network



$$y = f(x)$$

Using parallel computing techniques
to speed up matrix operation

$$= \sigma(W^L \dots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \dots + b^L)$$

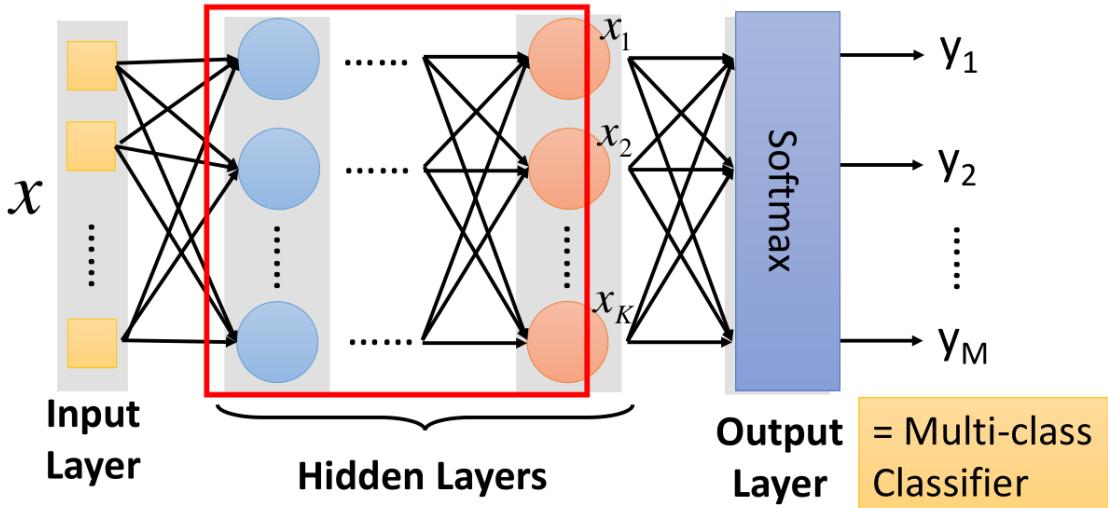
把这件事情写成矩阵运算的好处是，可以用GPU加速，GPU对matrix的运算是比CPU要来的快的，所以我们写neural network的时候，习惯把它写成matrix operation，然后call GPU来加速它

Output Layer

我们可以把hidden layers这部分，看做是一个feature extractor，这个feature extractor就replace了我们之前手动做feature engineering, feature transformation这些事情，经过这个feature extractor得到的output, x_1, x_2, \dots, x_k 就可以被当作一组新的feature

output layer做的事情，其实就是一个Multi-class classifier，它是拿经过feature extractor转换后的那一组比较好的feature（能够被很好地separate）进行分类的，由于我们把output layer看做是一个Multi-class classifier，所以我们在最后一个layer加上softmax

Feature extractor replacing feature engineering



Example Application

Handwriting Digit Recognition

Step 1: Neural Network

这里举一个手写数字识别的例子，input是一张image，对机器来说一张image实际上就是一个vector，假设这是一张 16×16 的image，那它有256个pixel，对machine来说，它是一个256维的vector，image中的每一个都对应到vector中的一个dimension，简单来说，我们把黑色的pixel的值设为1，白色的pixel的值设为0

而neural network的output，如果在output layer使用了softmax，那它的output就是一个突出极大值的Probability distribution，假设我们的output是10维的话（10个数字，0~9），这个output的每一维都对应到它可能是某一个数字的机率，实际上这个neural network的作用就是计算image成为10个数字的机率各自有多少，机率最大（softmax突出极大值的意义所在）的那个数字，就是机器的预测值

在手写字体识别的demo里，我们唯一需要的就是一个function，这个function的input是一个256的vector，output是一个10维的vector，这个function就是neural network（这里我们用简单的Feedforward network）

input固定为256维，output固定为10维的feedforward neural network，实际上这个network structure就已经确定了一个function set(model)的形状，在这个function set里的每一个function都可以拿来做手写数字识别

接下来我们要做的事情是用gradient descent去计算出一组参数，挑一个最适合拿来做手写数字识别的function

所以这里很重要的一件事情是，我们要对network structure进行design，之前在做Logistic Regression或者是Linear Regression的时候，我们对model的structure是没有什么好设计的，但是对neural network来说，我们现在已知的constraint只有input是256维，output是10维，而中间要有几个hidden layer，每个layer要有几个neuron，都是需要我们自己去设计的，它们近乎是决定了function set长什么样子

如果你的network structure设计的很差，这个function set里面根本就没有好的function，那就会像大海捞针一样，结果针并不在海里

input、output的dimension，加上network structure，就可以确定一个model的形状，前两个是容易知道的，而决定这个network的structure则是整个Deep Learning中最为关键的步骤

input 256维, output 10维, 以及自己design的network structure 决定了function set(model)

Q: How many layers? How many neurons for each layer?

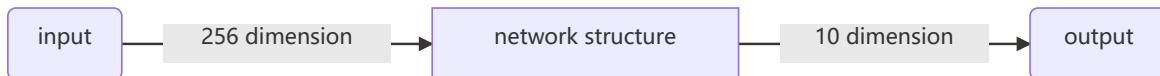
- Trial and Error + Intuition, 试错和直觉, 有时需要domain knowledge; 非deep的model, 做feature transform, 找好的feature; deep learning不需要找好的feature, 但是需要design network structure, 让machine自己找好的feature

Q: 有人可能会问, 机器能不能自动地学习network的structure?

- 其实是可以的, 基因演算法领域是有很多的technique是可以让machine自动地去找出network structure, 只不过这些方法目前没有非常普及

Q: 我们可不可以自己去design一个新的network structure, 比如说可不可以不要Fully connected layers(全连接层), 自己去DIY不同layers的neuron之间的连接?

- 当然可以, 一个特殊的接法就是CNN(Convolutional Neural Network)



Step 2: Goodness of function

定义一个function的好坏, 由于现在我们做的是一个Multi-class classification, 所以image为数字1的label "1"告诉我们, 现在的target是一个10维的vector, 只有在第一维对应数字1的地方, 它的值是1, 其他都是0

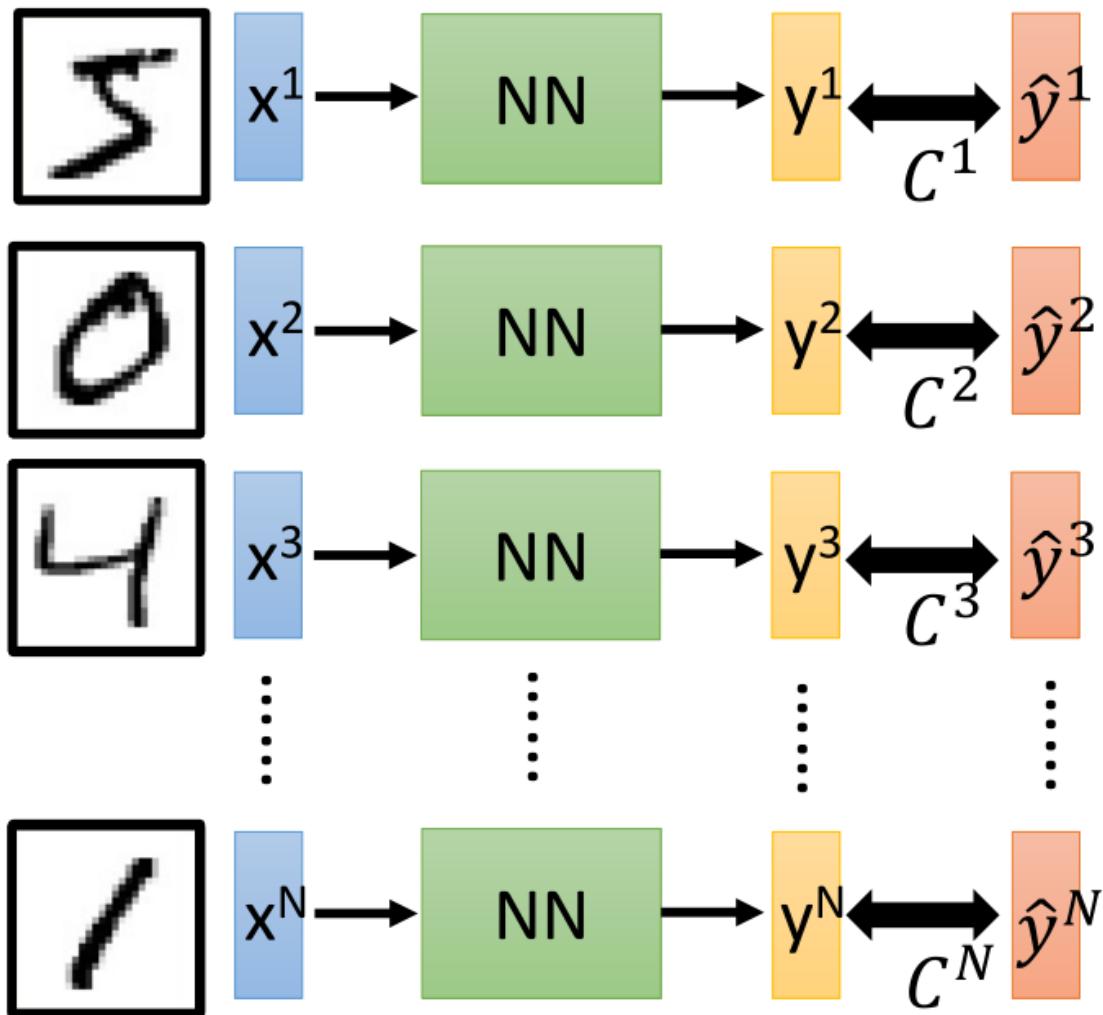
input这张image的256个pixel, 通过这个neural network之后, 会得到一个output, 称之为 y ; 而从这张image的label中转化而来的target, 称之为 \hat{y} , 有了output y 和target \hat{y} 之后, 要做的事情是计算它们之间的cross entropy, 这个做法跟我们之前做Multi-class classification的时候是一模一样的

$$\text{Cross Entropy} : C(y, \hat{y}) = - \sum_{i=1}^{10} \hat{y}_i \ln y_i$$

Step 3: Pick the best function

接下来就去调整参数, 让这个cross entropy越小越好, 当然整个training data里面不会只有一笔data, 你需要把所有data的cross entropy都sum起来, 得到一个total loss $L = \sum_{n=1}^N C^n$, 得到loss function之后你要做的事情是找一组network的parameters: θ^* , 它可以minimize这个total loss, 这组parameter 对应的function就是我们最终训练好的model

For all training data ...

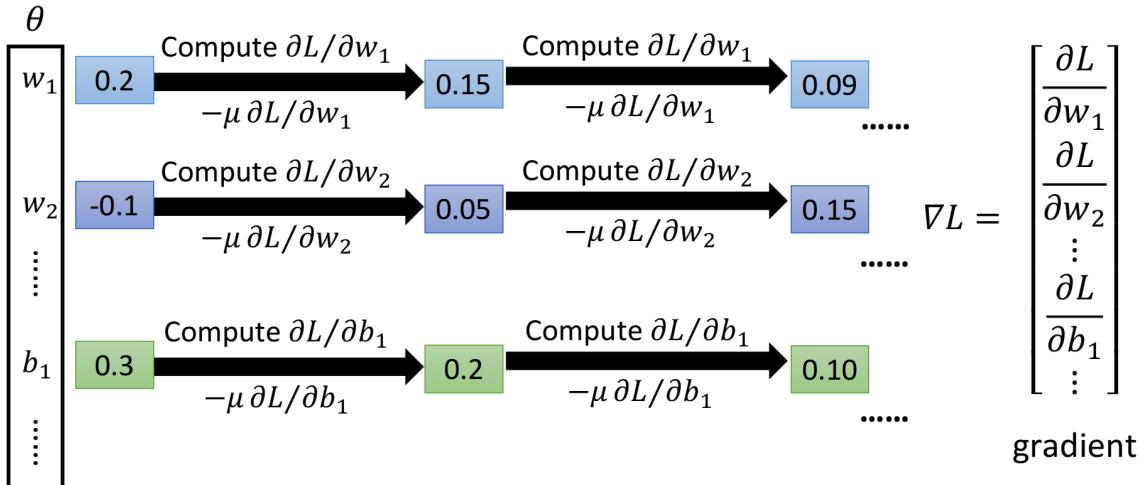


那怎么去找这个使total loss minimize的 θ^* 呢？使用的方法就是我们的老朋友Gradient Descent

实际上在deep learning里面用gradient descent，跟在linear regression里面使用完全没有什么差别，只是function和parameter变得更复杂了而已，其他事情都是一模一样的

现在你的 θ 里面是一大堆的weight、bias参数，先random找一个初始值，接下来去计算每一个参数对total loss的偏微分，把这些偏微分全部集合起来，就叫做gradient，有了这些偏微分以后，你就可以更新所有的参数，都减掉learning rate乘上偏微分的值，这个process反复进行下去，最终找到一组好的参数，就做完deep learning的training了

Gradient Descent



所以，其实deep learning就是这样子了，就算是alpha go，也是用gradient descent train出来的，可能在你的想象中它有多么得高大上，实际上就是在用gradient descent这样朴素的方法

Toolkit

你可能会问，这个gradient descent的function式子到底是长什么样子呢？之前我们都是一步一步地把那个算式推导出来的，但是在neural network里面，有成百上千个参数，如果要一步一步地人工推导并求微分的话是比较困难的，甚至是不可行的

其实，在现在这个时代，我们不需要像以前一样自己去implement Backpropagation，因为有太多太多的toolkit可以帮你计算Backpropagation，比如tensorflow、pytorch

注：Backpropagation就是算微分的一个比较有效的方式

Why Deep?

最后还有一个问题，为什么我们要deep learning？一个很直觉的答案是，越deep，performance就越好，一般来说，随着deep learning中的layers数量增加，error率不断降低

但是，稍微有一点machine learning常识的人都会觉得太surprise，因为本来model的parameter越多，它cover的function set就越大，它的bias就越小，如果今天你有足够的training data去控制它的variance，一个比较复杂、参数比较多的model，它performance比较好，是很正常的

那变deep有什么特别了不起的地方？

甚至有一个Universality Theorem是这样说的，任何连续的function，它的input是一个N维的vector，output是一个M维的vector，它都可以用一个hidden layer的neural network来表示，只要你这个hidden layer的neuron够多，它可以表示成任何的function，既然一个hidden layer的neural network可以表示成任何的function，而我们在做machine learning的时候，需要的东西就只是一个function而已，那做deep有什么特殊的意义呢？

所以有人说，deep learning就只是一个噱头而已，因为做deep感觉比较潮

如果你只是增加neuron把它变宽，变成fat neural network，那就感觉太“虚弱”了，所以我们要做deep learning，给它增加layers而不是增加neuron。

真的是这样吗？Why “Deep” neural network not “Fat” neural network?

后面会解释这件事情

Design network structure V.s. Feature Engineering

其实network structure的design是一件蛮难的事情，我们到底要怎么决定layer的数目和每一个layer的neuron的数目呢？

这个只能够凭着经验和直觉、多方面的尝试，有时候甚至会需要一些domain knowledge（专业领域的知识），从非deep learning的方法到deep learning的方法，并不是说machine learning比较简单，而是我们把一个问题转化成了另一个问题

本来不是deep learning的model，要得到一个好的结果，往往需要做feature engineering，也就是做feature transform，然后找一组好的feature

一开始学习deep learning的时候，好像会觉得deep learning的layers之间也是在做feature transform，但实际上在做deep learning的时候，往往不需要一个好的feature，比如说在做影像辨识的时候，你可以把所有的pixel直接丢进去

在过去做图像识别，你是需要对图像抽取出一些人定的feature出来的，这件事情就是feature transform，但是有了deep learning之后，你完全可以直接丢pixel进去硬做

但是，今天deep learning制造了一个新的问题，它所制造的问题就是，你需要去design network的structure，所以你的问题从本来的如何抽取feature转化成怎么design network structure，所以deep learning是不是真的好用，取决于你觉得哪一个问题比较容易

如果是影像辨识或者是语音辨识的话，design network structure可能比feature engineering要来的容易，因为，虽然我们人都会看、会听，但是这件事情，它太过潜意识了，它离我们意识的层次太远，我们无法意识到，我们到底是怎么做语音辨识这件事情，所以对人来说，你要抽一组好的feature，让机器可以很方便地用linear的方法做语音辨识，其实是很难的，因为人根本就不知道好的feature到底长什么样子；所以还不如design一个network structure，或者是尝试各种network structure，让machine自己去找出好的feature，这件事情反而变得比较容易，对影像来说也是一样的

有这么一个说法：deep learning在NLP上面的performance并没有那么好。语音辨识和影像辨识这两个领域是最早开始用deep learning的，一用下去进步量就非常地惊人，比如错误率一下子就降低了20%这样，但是在NLP上，它的进步量似乎并没有那么惊人，甚至有很多做NLP的人，现在认为说deep learning不见得那么work，这个原因可能是，人在做NLP这件事情的时候，由于人在文字处理上是比较强的，比如叫你设计一个rule去detect一篇document是正面的情绪还是负面的情绪，你完全可以列表，列出一些正面情绪和负面情绪的词汇，然后看这个document里面正面情绪的词汇出现的百分比是多少，你可能就可以得到一个不错的结果。所以NLP这个task，对人来说是比较容易设计rule的，你设计的那些ad-hoc（特别的）的rule，往往可以得到一个还不错的结果，这就是为什么deep learning相较于NLP传统的方法，觉得没有像其他领域一样进步得那么显著（但还是有一些进步的）

长久而言，可能文字处理中会有一些隐藏的资讯是人自己也不知道的，所以让机器自己去学这件事情，还是可以占到一些优势，眼下它跟传统方法的差异看起来并没有那么的惊人，但还是有进步的

Backpropagation

Backpropagation（反向传播），就是告诉我们用gradient descent来train一个neural network的时候该怎么做，它只是求微分的一种方法，而不是一种新的算法

Gradient Descent

gradient descent的使用方法，跟前面讲到的linear Regression或者是Logistic Regression是一模一样的，唯一的区别就在于当它用在neural network的时候，network parameters $\theta = w_1, w_2, \dots, b_1, b_2, \dots$ 里面可能会有将近million个参数

所以现在最大的困难是，如何有效地把这个近百万维的vector给计算出来，这就是Backpropagation要做的事情，所以Backpropagation并不是一个和gradient descent不同的training的方法，它就是gradient descent，它只是一个比较有效率的算法，让你在计算这个gradient的vector的时候更有效率

Chain Rule

Backpropagation里面并没有什么高深的数学，你唯一需要记得的就只有Chain Rule (链式法则)

$$\begin{aligned} \text{case1 : } & y = g(x) \quad z = h(y) \\ & \Delta x \rightarrow \Delta y \rightarrow \Delta z \\ & \frac{dz}{dx} = \frac{dy}{dx} \frac{dy}{dz} \\ \text{case2 : } & x = g(s) \quad y = h(s) \quad z = k(x, y) \\ & \Delta s \rightarrow \Delta x \rightarrow \Delta z \quad \Delta s \rightarrow \Delta y \rightarrow \Delta z \\ & \frac{dz}{ds} = \frac{\partial z}{\partial x} \frac{dx}{ds} + \frac{\partial z}{\partial y} \frac{dy}{ds} \end{aligned}$$

对整个neural network，我们定义了一个loss function: $L(\theta) = \sum_{n=1}^N C^n(\theta)$ ，它等于所有training data的loss之和

我们把training data里任意一个样本点 x^n 代到neural network里面，它会output一个 y^n ，我们把这个output跟样本点本身的label标注的target \hat{y}^n 作cross entropy，这个交叉熵定义了output y^n 和target \hat{y}^n 之间的距离 $C^n(\theta)$ ，如果cross entropy比较大的话，说明output和target之间距离很远，这个network的parameter的loss是比较大的，反之则说明这组parameter是比较好的。

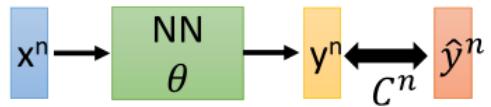
然后summation over所有training data的cross entropy $C^n(\theta)$ ，得到total loss $L(\theta)$ ，这就是我们的loss function，用这个 $L(\theta)$ 对某一个参数w做偏微分，表达式如下：

$$\frac{\partial L(\theta)}{\partial w} = \sum_{n=1}^N \frac{\partial C^n(\theta)}{\partial w}$$

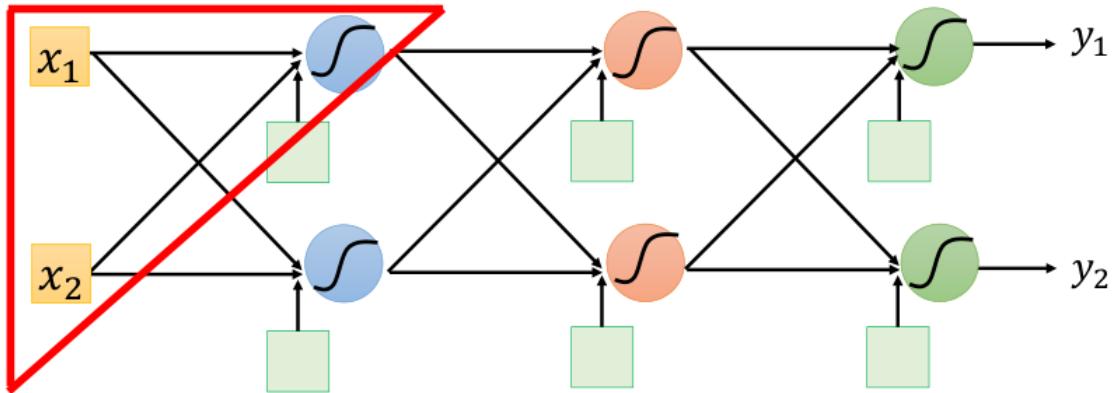
这个表达式告诉我们，只需要考虑如何计算对某一笔data的 $\frac{\partial C^n(\theta)}{\partial w}$ ，再将所有training data的cross entropy对参数w的偏微分累计求和，就可以把total loss对某一个参数w的偏微分给计算出来

我们先考虑某一个neuron，假设只有两个input x_1, x_2 ，通过这个neuron，我们先得到 $z = b + w_1 x_1 + w_2 x_2$ ，然后经过activation function从这个neuron中output出来，作为后续neuron的input，再经过了非常多非常多的事情以后，会得到最终的output y_1, y_2

Backpropagation



$$L(\theta) = \sum_{n=1}^N C^n(\theta) \rightarrow \frac{\partial L(\theta)}{\partial w} = \sum_{n=1}^N \frac{\partial C^n(\theta)}{\partial w}$$



现在的问题是这样： $\frac{\partial C}{\partial w}$ 该怎么算？按照chain rule，可以把它拆分成两项， $\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w} \frac{\partial C}{\partial z}$ ，这两项分别去把它计算出来。前面这一项是比较简单的，后面这一项是比较复杂的

计算前面这一项 $\frac{\partial z}{\partial w}$ 的这个process，我们称之为**Forward pass**；而计算后面这项 $\frac{\partial C}{\partial z}$ 的process，我们称之为**Backward pass**

Forward pass

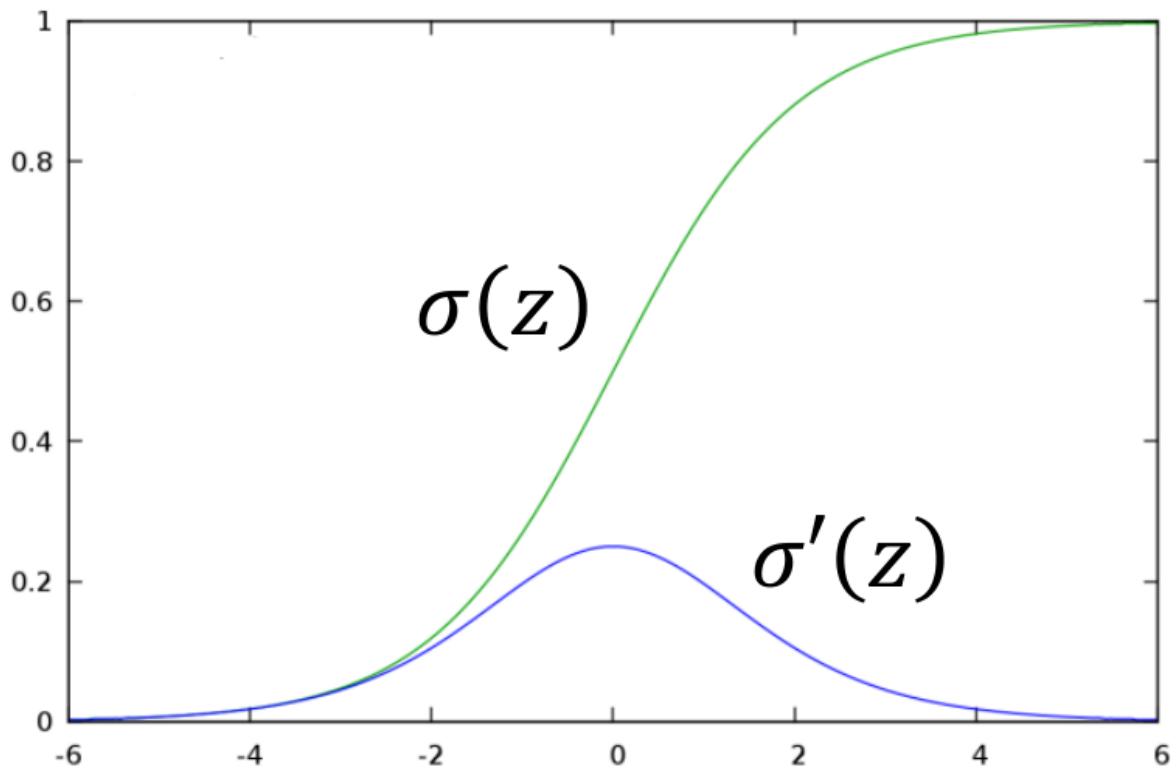
先考虑 $\frac{\partial z}{\partial w}$ 这一项，完全可以秒算出来， $\frac{\partial z}{\partial w_1} = x_1, \frac{\partial z}{\partial w_2} = x_2$

它的规律是这样的：求 $\frac{\partial z}{\partial w}$ ，就是看w前面连接的是什么，那微分后的 $\frac{\partial z}{\partial w}$ 值就是什么，因此只要计算出neural network里面每一个neuron的output就可以知道任意的z对w的偏微分

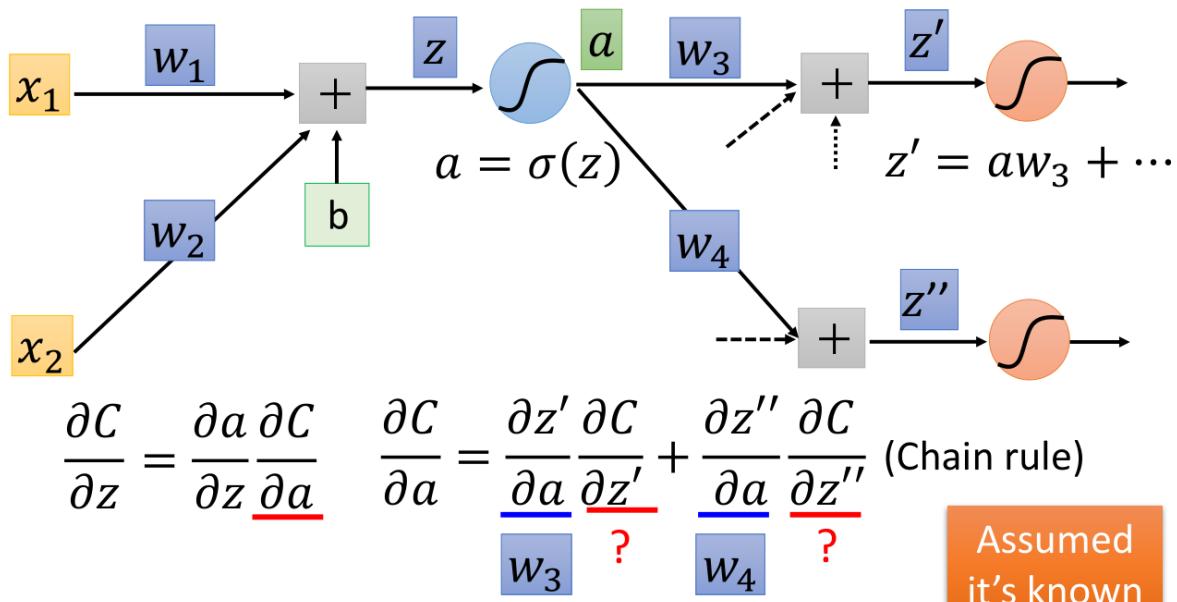
- 比如input layer作为neuron的输入时， w_1 前面连接的是 x_1 ，所以微分值就是 x_1 ； w_2 前面连接的是 x_2 ，所以微分值就是 x_2
- 比如hidden layer作为neuron的输入时，那该neuron的就是前一层neuron的output，于是 $\frac{\partial z}{\partial w}$ 的值就是前一层的z经过activation function之后输出的值

Backward pass

再考虑 $\frac{\partial C}{\partial z}$ 这一项，它是比较复杂的，这里我们假设activation function是sigmoid function



Compute $\partial C / \partial z$ for all activation function inputs z



我们的 z 通过activation function得到 a , 这个neuron的output是 $a = \sigma(z)$, 接下来这个 a 会乘上某一个weight w_3 , 再加上其它一大堆的value得到 z' , 它是下一个neuron activation function的input, 然后 a 又会乘上另一个weight w_4 , 再加上其它一堆value得到 z'' , 后面还会发生很多很多其他事情

不过这里我们就只先考虑下一步会发生什么事情:

$$\frac{\partial C}{\partial z} = \frac{\partial a}{\partial z} \frac{\partial C}{\partial a}$$

这里的 $\frac{\partial a}{\partial z}$ 实际上就是activation function的微分 (在这里就是sigmoid function的微分), 接下来的问题是 $\frac{\partial C}{\partial a}$ 应该长什么样子呢? a 会影响 z' 和 z'' , 而 z' 和 z'' 会影响 C , 所以通过chain rule可以得到

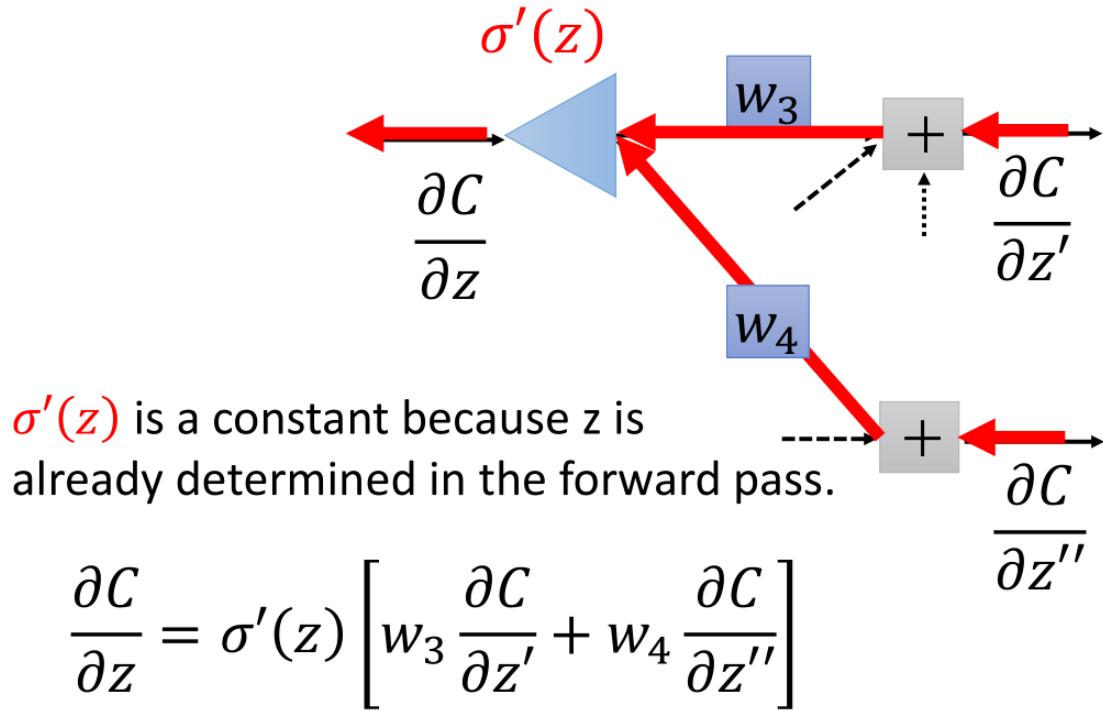
$$\frac{\partial C}{\partial a} = \frac{\partial z'}{\partial a} \frac{\partial C}{\partial z'} + \frac{\partial z''}{\partial a} \frac{\partial C}{\partial z''}$$

这里的 $\frac{\partial z'}{\partial a} = w_3$, $\frac{\partial z''}{\partial a} = w_4$, 那 $\frac{\partial C}{\partial z'}$ 和 $\frac{\partial C}{\partial z''}$ 又该怎么算呢? 这里先假设我们已经通过某种方法把 $\frac{\partial C}{\partial z'}$ 和 $\frac{\partial C}{\partial z''}$ 这两项给算出来了, 然后回过头去就可以把 $\frac{\partial C}{\partial z}$ 给轻易地算出来

$$\frac{\partial C}{\partial z} = \frac{\partial a}{\partial z} \frac{\partial C}{\partial a} = \sigma'(z)[w_3 \frac{\partial C}{\partial z'} + w_4 \frac{\partial C}{\partial z''}]$$

这个式子还是蛮简单的, 然后, 我们可以从另外一个观点来看待这个式子

你可以想象说, 现在有另外一个neuron, 它不在我们原来的network里面, 在下图中它被画成三角形, 这个neuron的input就是 $\frac{\partial C}{\partial z'}$ 和 $\frac{\partial C}{\partial z''}$, 那input $\frac{\partial C}{\partial z'}$ 就乘上 w_3 , input $\frac{\partial C}{\partial z''}$ 就乘上 w_4 , 它们两个相加再乘上 activation function的微分 $\sigma'(z)$, 就可以得到output $\frac{\partial C}{\partial z}$



这张图描述了一个新的“neuron”, 它的含义跟图下方的表达式是一模一样的, 作这张图的目的是为了方便理解

值得注意的是, 这里的 $\sigma'(z)$ 是一个constant常数, 它并不是一个function, 因为 z 其实在计算forward pass的时候就已经被决定好了, z 是一个固定的值

所以这个neuron其实跟我们之前看到的sigmoid function是不一样的, 它并不是把input通过一个non-linear进行转换, 而是直接把input乘上一个constant $\sigma'(z)$, 就得到了output, 因此这个neuron被画成三角形, 代表它跟我们之前看到的圆形的neuron的运作方式是不一样的, 它是直接乘上一个constant (这里的三角形有点像电路里的运算放大器op-amp, 它也是乘上一个constant)

现在我们最后需要解决的问题是, 怎么计算 $\frac{\partial C}{\partial z'}$ 和 $\frac{\partial C}{\partial z''}$ 这两项, 假设有两个不同的case:

Case 1: Output Layer

假设蓝色的这个neuron已经是hidden layer的最后一层了, 也就是说连接在 z' 和 z'' 后的这两个红色的neuron已经是output layer, 它的output就已经是整个network的output了, 这个时候计算就比较简单

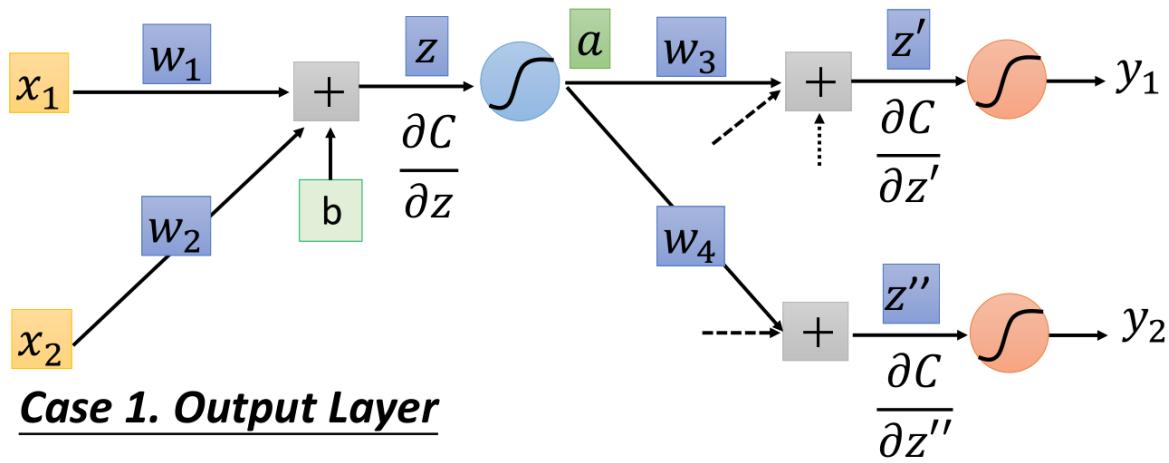
$$\frac{\partial C}{\partial z'} = \frac{\partial y_1}{\partial z'} \frac{\partial C}{\partial y_1}$$

其中 $\frac{\partial y_1}{\partial z'}$ 就是output layer的activation function (softmax) 对 z' 的偏微分

而 $\frac{\partial C}{\partial y_1}$ 就是 loss 对 y_1 的偏微分，它取决于你的 loss function 是怎么定义的，也就是你的 output 和 target 之间是怎么 evaluate 的，你可以用 cross entropy，也可以用 mean square error，用不同的定义， $\frac{\partial C}{\partial y_1}$ 的值就不一样

这个时候，你就已经可以把 C 对 w_1 和 w_2 的偏微分 $\frac{\partial C}{\partial w_1}$ 、 $\frac{\partial C}{\partial w_2}$ 算出来了

Compute $\frac{\partial C}{\partial z}$ for all activation function inputs z



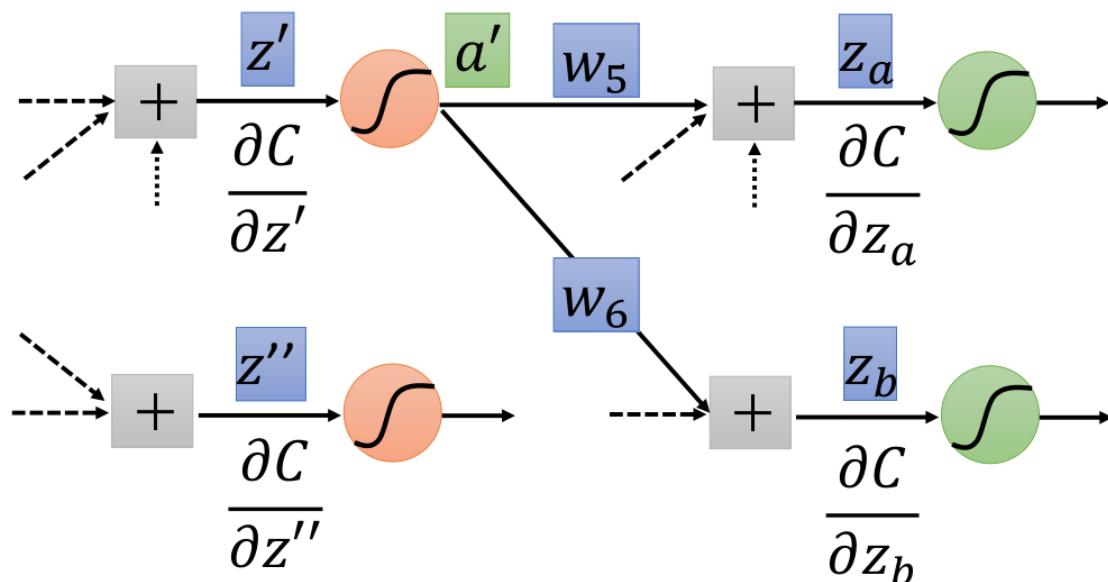
Case 1. Output Layer

$$\frac{\partial C}{\partial z'} = \frac{\partial y_1}{\partial z'} \frac{\partial C}{\partial y_1} \quad \frac{\partial C}{\partial z''} = \frac{\partial y_2}{\partial z''} \frac{\partial C}{\partial y_2}$$

Done!

Case 2: Not Output Layer

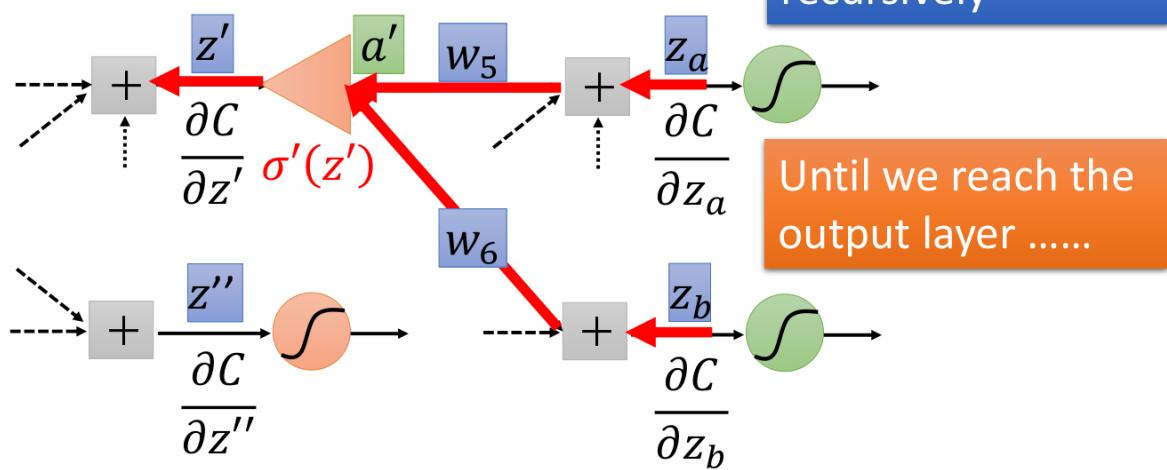
假设现在红色的neuron并不是整个network的output，那 z' 经过红色neuron的activation function 得到 a' ，然后 output a' 和 w_5 、 w_6 相乘并加上一堆其他东西分别得到 z_a 和 z_b ，如下图所示



根据之前的推导证明类比，如果知道 $\frac{\partial C}{\partial z_a}$ 和 $\frac{\partial C}{\partial z_b}$ ，我们就可以计算 $\frac{\partial C}{\partial z'}$ ，如下图所示，借助运算放大器的辅助理解，将 $\frac{\partial C}{\partial z_a}$ 乘上 w_5 和 $\frac{\partial C}{\partial z_b}$ 乘上 w_6 的值加起来再通过 op-amp，乘上放大系数 $\sigma'(z')$ ，就可以得到 output $\frac{\partial C}{\partial z'}$

$$\frac{\partial C}{\partial z'} = \sigma'(z') [w_5 \frac{\partial C}{\partial z_a} + w_6 \frac{\partial C}{\partial z_b}]$$

Case 2. Not Output Layer



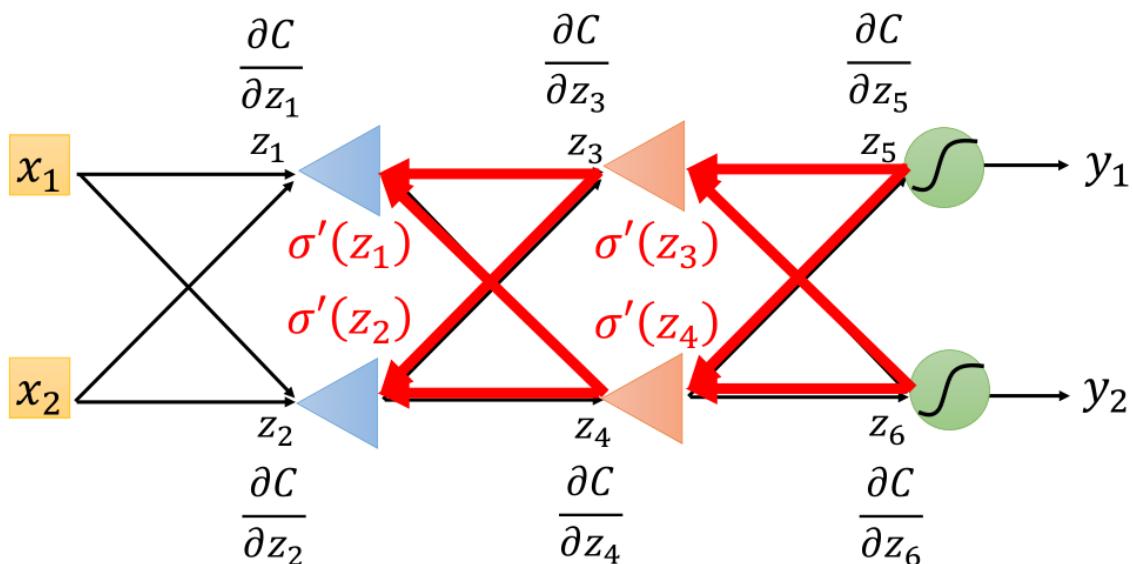
知道 z' 和 z'' 就可以知道 z , 知道 z_a 和 z_b 就可以知道 z' , ..., 现在这个过程就可以反复进行下去, 直到找到output layer, 我们可以算出确切的值, 然后再一层一层反推回去

你可能会想说, 这个方法听起来挺让人崩溃的, 每次要算一个微分的值, 都要一路往后走, 一直走到network的output, 如果写成表达式的话, 一层一层往后展开, 感觉会是一个很可怕的式子, 但是实际上并不是这个样子做的

你只要换一个方向, 从output layer的 $\frac{\partial C}{\partial z}$ 开始算, 你就会发现它的运算量跟原来的network的Feedforward path其实是一样的

假设现在有6个neuron, 每一个neuron的activation function的input分别是 z_1 、 z_2 、 z_3 、 z_4 、 z_5 、 z_6 , 我们要计算 C 对这些 z 的偏微分, 按照原来的思路, 我们想要知道 z_1 的偏微分, 就要去算 z_3 和 z_4 的偏微分, 想要知道 z_3 和 z_4 的偏微分, 就要去计算两遍 z_5 和 z_6 的偏微分, 因此如果我们是从 z_1 、 z_2 的偏微分开始算, 那就没有效率

但是, 如果你反过来先去计算 z_5 和 z_6 的偏微分的话, 这个process, 就突然之间变得有效率起来了, 我们先去计算 $\frac{\partial C}{\partial z_5}$ 和 $\frac{\partial C}{\partial z_6}$, 然后就可以算出 $\frac{\partial C}{\partial z_3}$ 和 $\frac{\partial C}{\partial z_4}$, 最后就可以算出 $\frac{\partial C}{\partial z_1}$ 和 $\frac{\partial C}{\partial z_2}$, 而这一整个过程, 就可以转化为op-amp运算放大器的那张图



这里每一个op-amp的放大系数就是 $\sigma'(z_1)$ 、 $\sigma'(z_2)$ 、 $\sigma'(z_3)$ 、 $\sigma'(z_4)$ ，所以整个流程就是，先快速地计算出 $\frac{\partial C}{\partial z_5}$ 和 $\frac{\partial C}{\partial z_6}$ ，然后再把这两个偏微分的值乘上路径上的weight汇集到neuron上面，再通过op-amp的放大，就可以得到 $\frac{\partial C}{\partial z_3}$ 和 $\frac{\partial C}{\partial z_4}$ 这两个偏微分的值，再让它们乘上一些weight，并且通过一个op-amp，就得到 $\frac{\partial C}{\partial z_1}$ 和 $\frac{\partial C}{\partial z_2}$ 这两个偏微分的值，这样就计算完了，这个步骤，就叫做Backward pass

在做Backward pass的时候，实际上的做法就是建另外一个neural network，本来正向neural network里面的activation function都是sigmoid function，而现在计算Backward pass的时候，就是建一个反向的neural network，它的activation function就是一个运算放大器op-amp，要先算完Forward pass后，才算得出来

每一个反向neuron的input是loss C 对后面一层layer的 z 的偏微分 $\frac{\partial C}{\partial z}$ ，output则是loss C 对这个neuron的 z 的偏微分 $\frac{\partial C}{\partial z}$ ，做Backward pass就是通过这样一个反向neural network的运算，把loss C 对每一个neuron的 z 的偏微分 $\frac{\partial C}{\partial z}$ 都给算出来

如果是正向做Backward pass的话，实际上每次计算一个 $\frac{\partial C}{\partial z}$ ，就需要把该neuron后面所有的 $\frac{\partial C}{\partial z}$ 都给计算一遍，会造成很多不必要的重复运算，如果写成code的形式，就相当于调用了很多次重复的函数；而如果是反向做Backward pass，实际上就是把这些调用函数的过程都变成调用值的过程，因此可以直接计算出结果，而不需要占用过多的堆栈空间

Summary

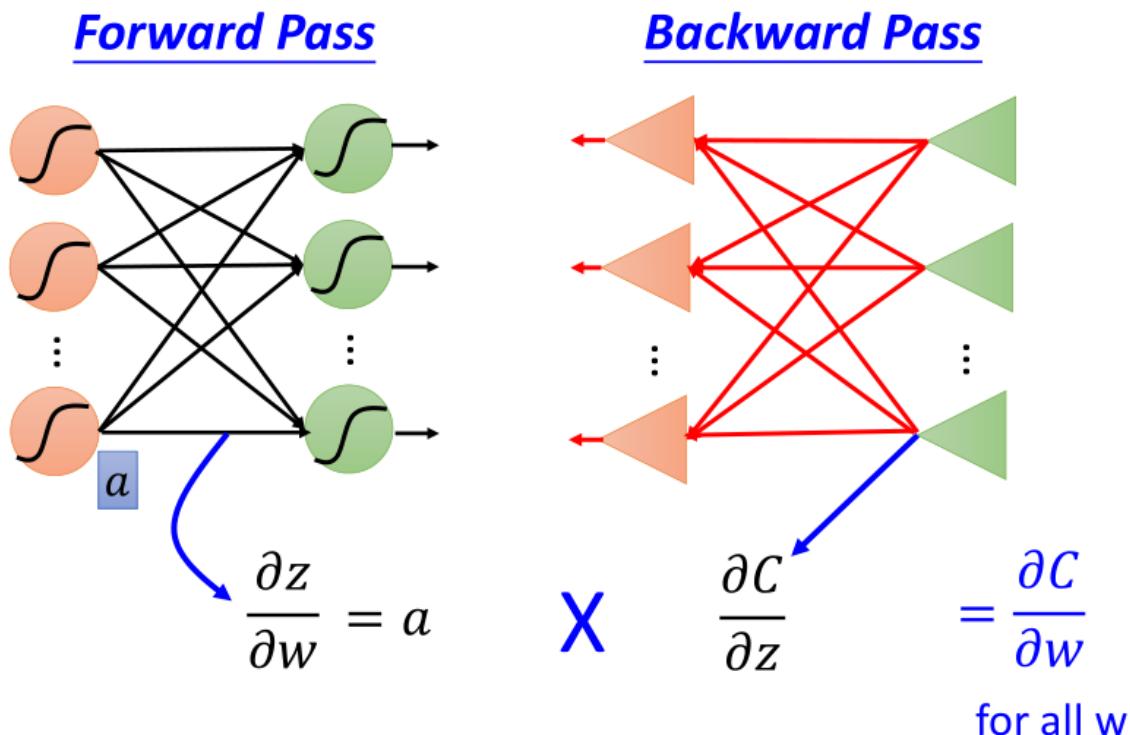
最后，我们来总结一下Backpropagation是怎么做的

Forward pass，每个neuron的activation function的output，就是它所连接的weight的 $\frac{\partial z}{\partial w}$

Backward pass，建一个与原来方向相反的neural network，它的三角形neuron的output就是 $\frac{\partial C}{\partial z}$

把通过forward pass得到的 $\frac{\partial z}{\partial w}$ 和通过backward pass得到的 $\frac{\partial C}{\partial z}$ 乘起来就可以得到 C 对 w 的偏微分 $\frac{\partial C}{\partial w}$

$$\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w} \Big|_{forward \ pass} \cdot \frac{\partial C}{\partial z} \Big|_{backward \ pass}$$



Tips for Deep Learning

- 在training set上准确率不高:
 - new activation function: ReLU、Maxout
 - adaptive learning rate: Adagrad、RMSProp、Momentum、Adam
- 在testing set上准确率不高
 - Early Stopping、Regularization or Dropout

Recipe of Deep Learning

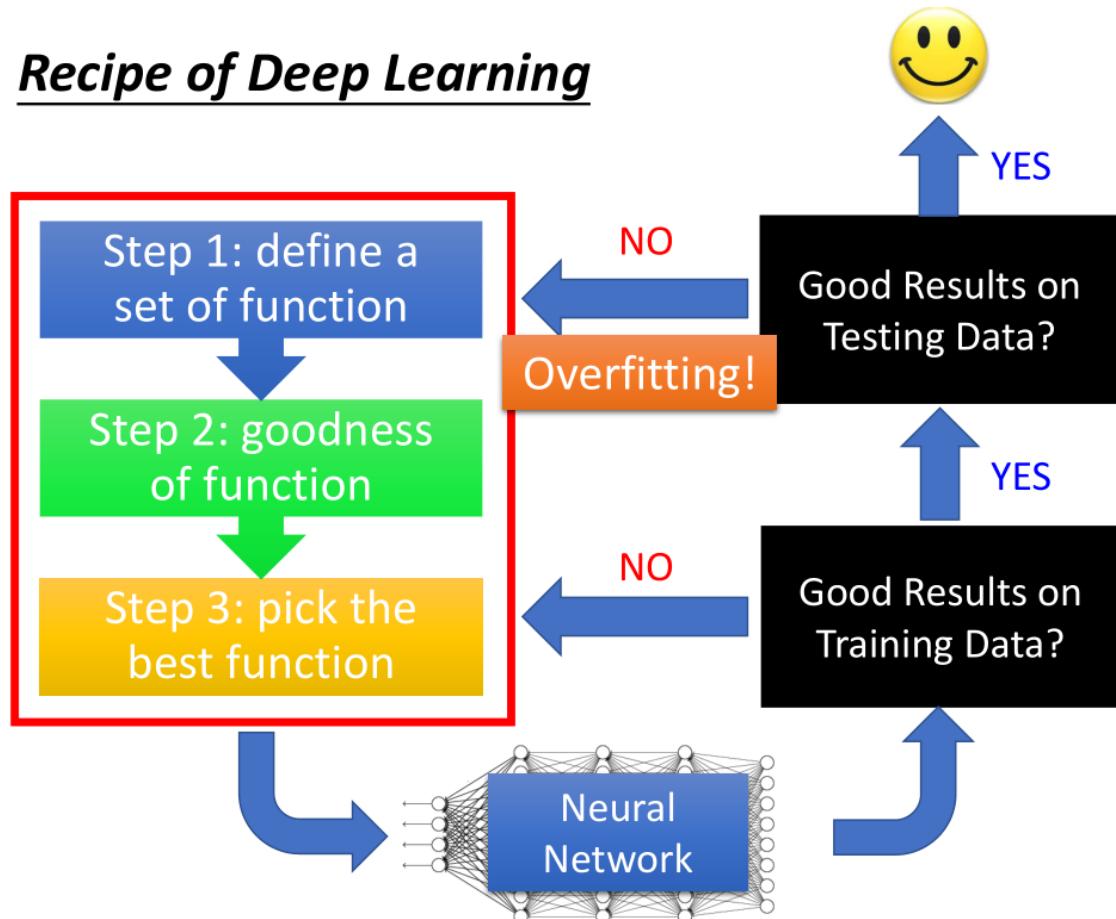
3 step of deep learning

Recipe, 配方、秘诀，这里指的是做deep learning的流程应该是什么样子

我们都已经知道了deep learning的三个步骤

- define the function set(network structure)
- goodness of function(loss function -- cross entropy)
- pick the best function(gradient descent -- optimization)

做完这些事情以后，你会得到一个更好的neural network，那接下来你要做什么事情呢？



Good Results on Training Data?

你要做的第一件事是，提高model在training set上的正确率

先检查training set的performance其实是deep learning一个非常unique的地方，如果今天你用的是k-nearest neighbor或decision tree这类非deep learning的方法，做完以后你其实会不太想检查training set的结果，因为在training set上的performance正确率就是100%，没有什么好检查的

有人说deep learning的model里这么多参数，感觉很容易overfitting的样子，但实际上这个deep learning的方法，它才不容易overfitting，我们说的overfitting就是在training set上performance很好，但在testing set上performance没有那么好

只有像k nearest neighbor, decision tree这类方法，它们在training set上正确率都是100，这才是非常容易overfitting的，而对deep learning来说，overfitting往往不会是你遇到的第一个问题

因为你在training的时候，deep learning并不是像k nearest neighbor这种方法一样，一训练就可以得到非常好的正确率，它有可能在training set上根本没有办法给你一个好的正确率，所以，这个时候你要回头去检查在前面的step里面要做什么样的修改，好让你在training set上可以得到比较高的正确率

Good Results on Testing Data?

接下来你要做的事是，提高model在testing set上的正确率

假设现在你已经在training set上得到好的performance了，那接下来就把model apply到testing set上，我们最后真正关心的，是testing set上的performance，假如得到的结果不好，这个情况下发生的才是Overfitting，也就是在training set上得到好的结果，却在testing set上得到不好的结果

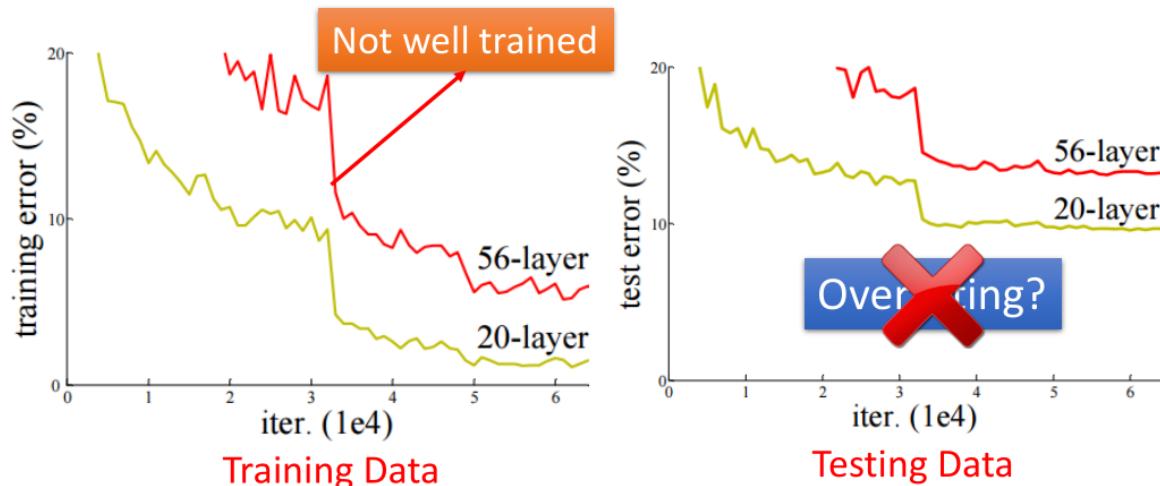
那你要回过头去做一些事情，试着解决overfitting，但有时候你加了新的technique，想要overcome overfitting这个problem的时候，其实反而会让training set上的结果变坏；所以你在做完这一步的修改以后，要先回头去检查新的model在training set上的结果，如果这个结果变坏的话，你就要从头对network training的过程做一些调整，那如果你同时在training set还有testing set上都得到好结果的话，你就成功了，最后就可以把你的系统真正用在application上面了

Do not always blame overfitting

不要看到所有不好的performance就归责于overfitting

先看右边testing data的图，横坐标是model做gradient descent所update的次数，纵坐标则是error rate（越低说明model表现得越好），黄线表示的是20层的neural network，红色表示56层的neural network

你会发现，这个56层network的error rate比较高，它的performance比较差，而20层network的performance则是比较好的，有些人看到这个图，就会马上得到一个结论：56层的network参数太多了，56层果然没有必要，这个是overfitting。但是，真的是这样子吗？



你在说结果是overfitting之前，有检查过training set上的performance吗？对neural network来说，在training set上得到的结果很可能会像左边training error的图，也就是说，20层的network本来就要比56层的network表现得更好，所以testing set得到的结果并不能说明56层的case就是发生了overfitting

在做neural network training的时候，有太多太多的问题可以让你的training set表现的不好，比如说我们有local minimum的问题，有saddle point的问题，有plateau的问题...

所以这个56层的neural network，有可能在train的时候就卡在了一个local minimum的地方，于是得到了一个差的参数，但这并不是overfitting，而是在training的时候就没有train好

有人认为这个问题叫做underfitting，但我的理解，underfitting的本意应该是指这个model的complexity不足，这个model的参数不够多，所以它的能力不足以解出这个问题；但这个56层的network，它的参数是比20层的network要来得多的，所以它明明有能力比20层的network要做的更好，却没有得到理想的结果，这种情况不应该被称为underfitting，其实就只是没有train好而已

Conclusion

当你在deep learning的文献上看到某种方法的时候，永远要想一下，这个方法是要解决什么样的问题，因为在deep learning里面，有两个问题：

- 在training set上的performance不够好
- 在testing set上的performance不够好

当有一个方法propose（提出）的时候，它往往只针对这两个问题的其中一个来做处理，举例来说，deep learning有一个很潮的方法叫做dropout，那很多人就会说，哦，这么潮的方法，所以今天只要看到performance不好，我就去用dropout；

但是，其实只有在testing的结果不好的时候，才可以去apply dropout，如果你今天的问题只是training的结果不好，那你去apply dropout，只会越train越差而已

所以，你**必须要先想清楚现在的问题到底是什么，然后再根据这个问题去找针对性的方法**，而不是病急乱投医，甚至是盲目诊断

下面我们分别从Training data和Testing data两个问题出发，来讲述一些针对性优化的方法

Good Results on Training Data?

如何在Training data上得到更好的performance，分为两个模块，New activation function和Adaptive Learning Rate

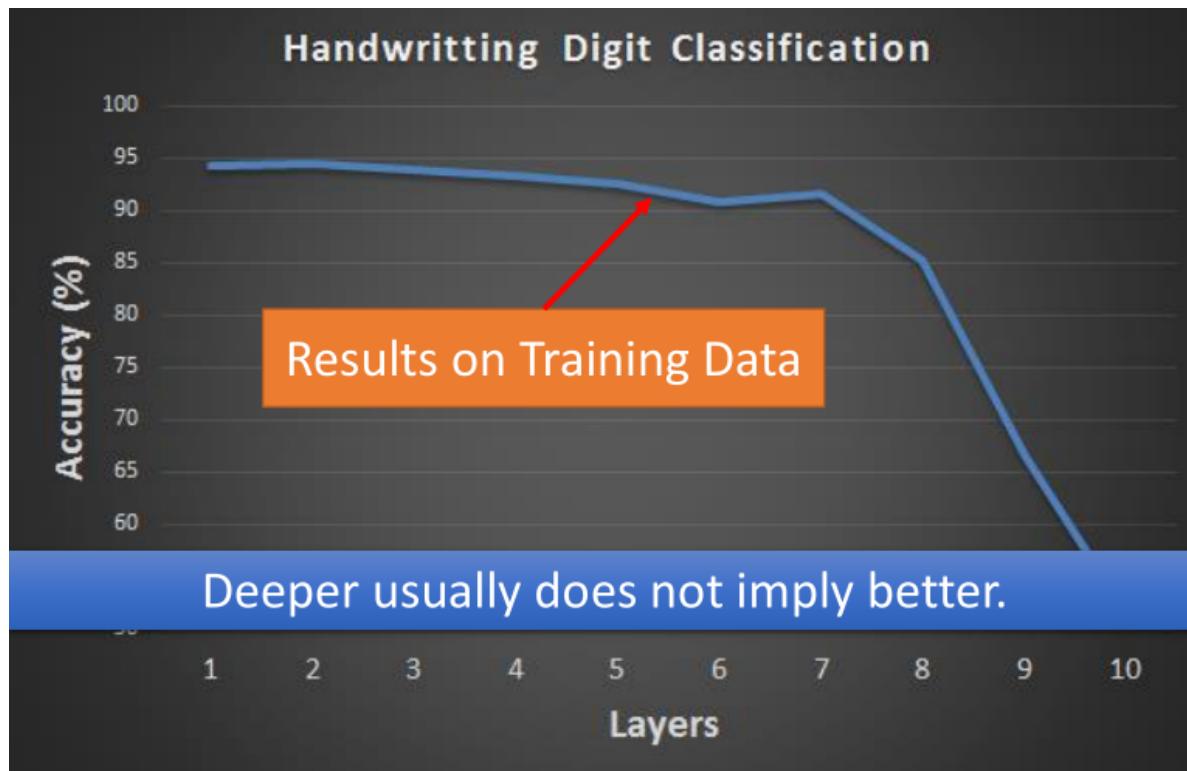
New activation function

activation function

如果你今天的training结果不好，很有可能是因为你的network架构设计得不好。举例来说，可能你用的activation function是对training比较不利的，那你就尝试着换一些新的activation function，也许可以带来比较好的结果

在1980年代，比较常用的activation function是sigmoid function，如果现在我们使用sigmoid function，你会发现deeper不一定imply better，在MNIST手写数字识别上training set的结果，当layer越来越多的时候，accuracy一开始持平，后来就掉下去了，在layer是9层、10层的时候，整个结果就崩溃了

但注意9层、10层的情况并不能被认为是因为参数太多而导致overfitting，实际上这只是training set的结果，你都不知道testing的情况，又哪来的overfitting之说呢？



Vanishing Gradient Problem

上面这个问题的原因不是overfitting，而是Vanishing Gradient（梯度消失），解释如下：

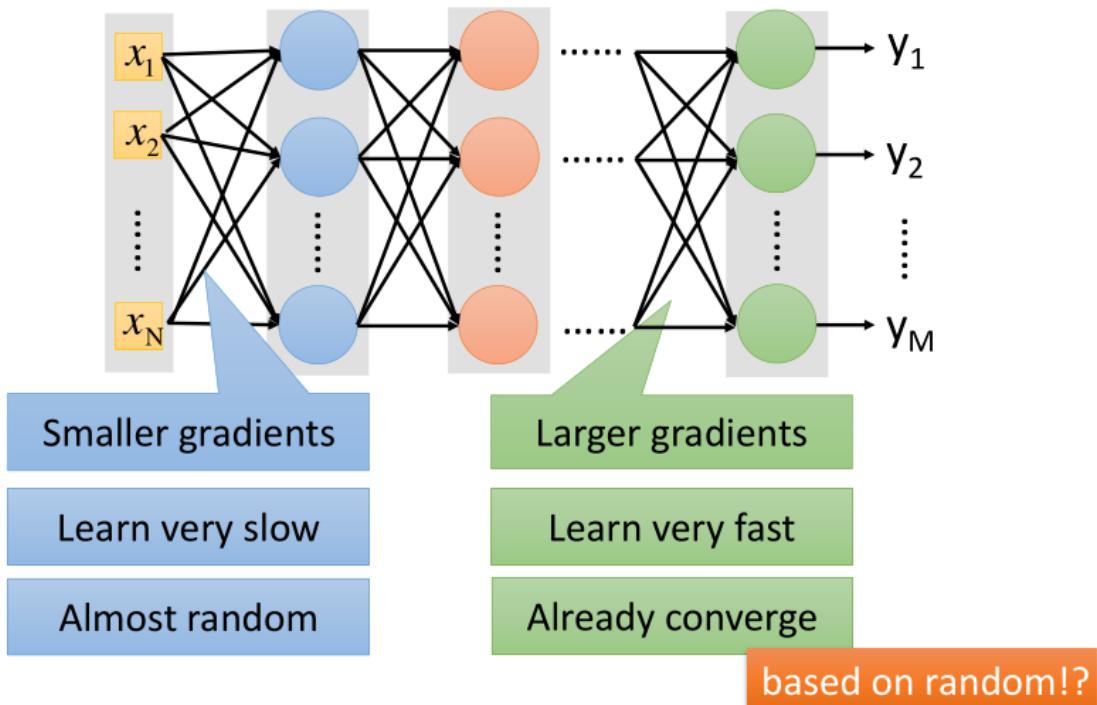
当你把network叠得很深的时候，在靠近input的地方，这些参数的gradient（即对最后loss function的微分）是比较小的；而在比较靠近output的地方，它对loss的微分值会是比较大的

因此当你设定同样learning rate的时候，靠近input的地方，它参数的update是很慢的；而靠近output的地方，它参数的update是比较快的

所以在靠近input的地方，参数几乎还是random的时候，output就已经根据这些random的结果找到了一个local minima，然后就converge(收敛)了

这个时候你会发现，参数的loss下降的速度变得很慢，你就会觉得gradient已经接近于0了，于是把程序停掉了，由于这个converge，是几乎base on random的参数，所以model的参数并没有被训练充分，那在training data上得到的结果肯定是很差的

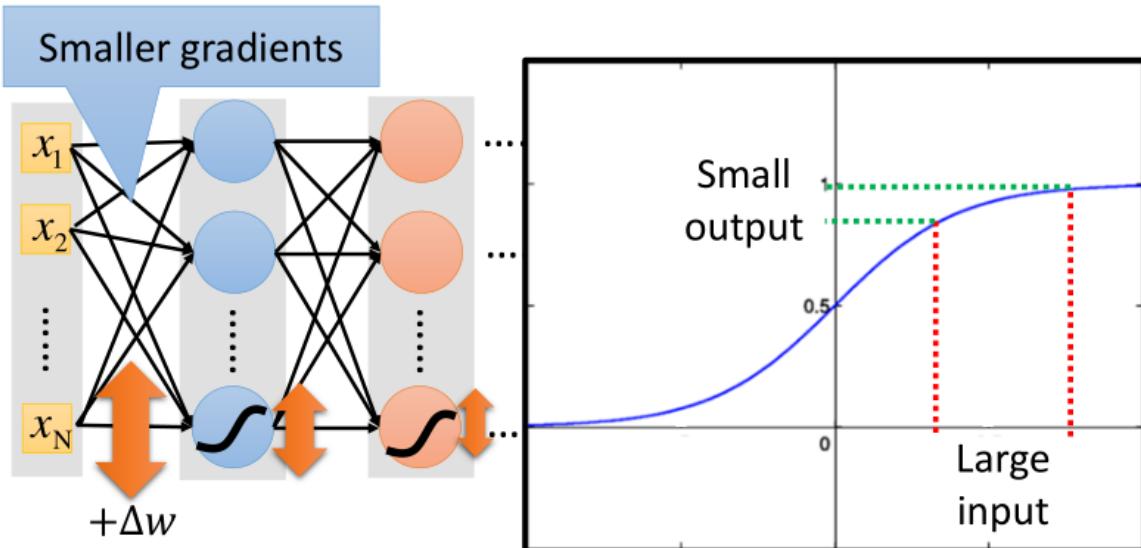
Vanishing Gradient Problem



为什么会有这个现象发生呢？如果你自己把Backpropagation的式子写出来的话，就可以很轻易地发现用sigmoid function会导致这件事情的发生；但是，我们今天不看Backpropagation的式子，其实从直觉上来想你也可以了解这件事情发生的原因

某一个参数 w 对total loss l 的偏微分，即gradient $\frac{\partial l}{\partial w}$ ，它直觉的意思是说，当我今天把这个参数做小小的变化的时候，它对这个loss 的影响有多大；那我们就把第一个layer里的某一个参数 w 加上 Δw ，看看对network的output和target之间的loss有什么样的影响

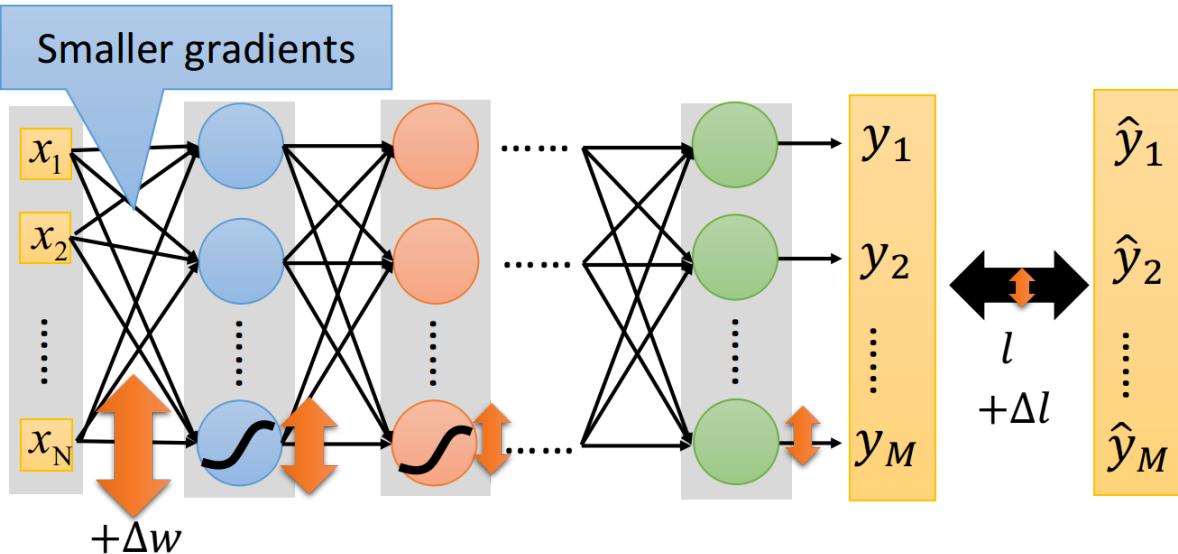
Δw 通过sigmoid function之后，得到output是会变小的，改变某一个参数的weight，会对某个neuron的output值产生影响，但是这个影响是会随着层数的递增而衰减的



sigmoid function的形状如图所示，它会把负无穷大到正无穷大之间的值都硬压到0~1之间，把较大的input压缩成较小的output

因此即使 Δw 值很大，但每经过一个sigmoid function就会被缩小一次，所以network越深， Δw 被衰减的次数就越多，直到最后，它对output的影响就是比较小的，相应的也导致input对loss的影响会比较小，于是靠近input的那些weight对loss的gradient $\frac{\partial l}{\partial w}$ 远小于靠近output的gradient

Vanishing Gradient Problem



Intuitive way to compute the derivatives ... $\frac{\partial l}{\partial w} = ? \frac{\Delta l}{\Delta w}$

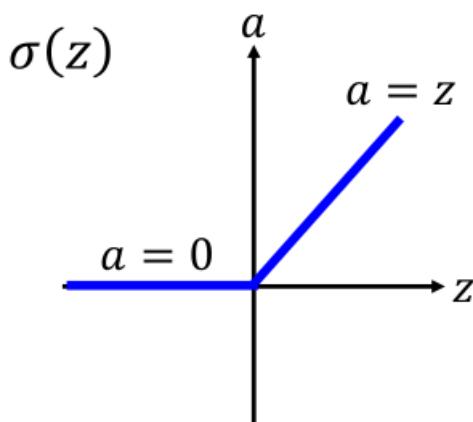
那怎么解决这个问题呢？比较早年做法是去train RBM，做layer-wise pre-training，它的精神就是，先把第一个layer train好，再去train第二个，然后再第三个...所以最后你在做Backpropagation的时候，尽管第一个layer几乎没有被train到，但一开始在做pre-train的时候就已经把它给pre-train好了，这就是RBM做pre-train有用的原因。可以在一定程度上解决问题

但其实改一下activation function可能就可以handle这个问题了

ReLU

现在比较常用的activation function叫做Rectified Linear Unit (整流线性单元函数，又称修正线性单元)，它的缩写是ReLU，该函数形状如下图所示， z 为input， a 为output，如果input>0则output = input，如果input<0则output = 0

Rectified Linear Unit (ReLU)



Reason:

1. Fast to compute
2. Biological reason
3. Infinite sigmoid with different biases
4. Vanishing gradient problem

[Xavier Glorot, AISTATS'11]
 [Andrew L. Maas, ICML'13]
 [Kaiming He, arXiv'15]

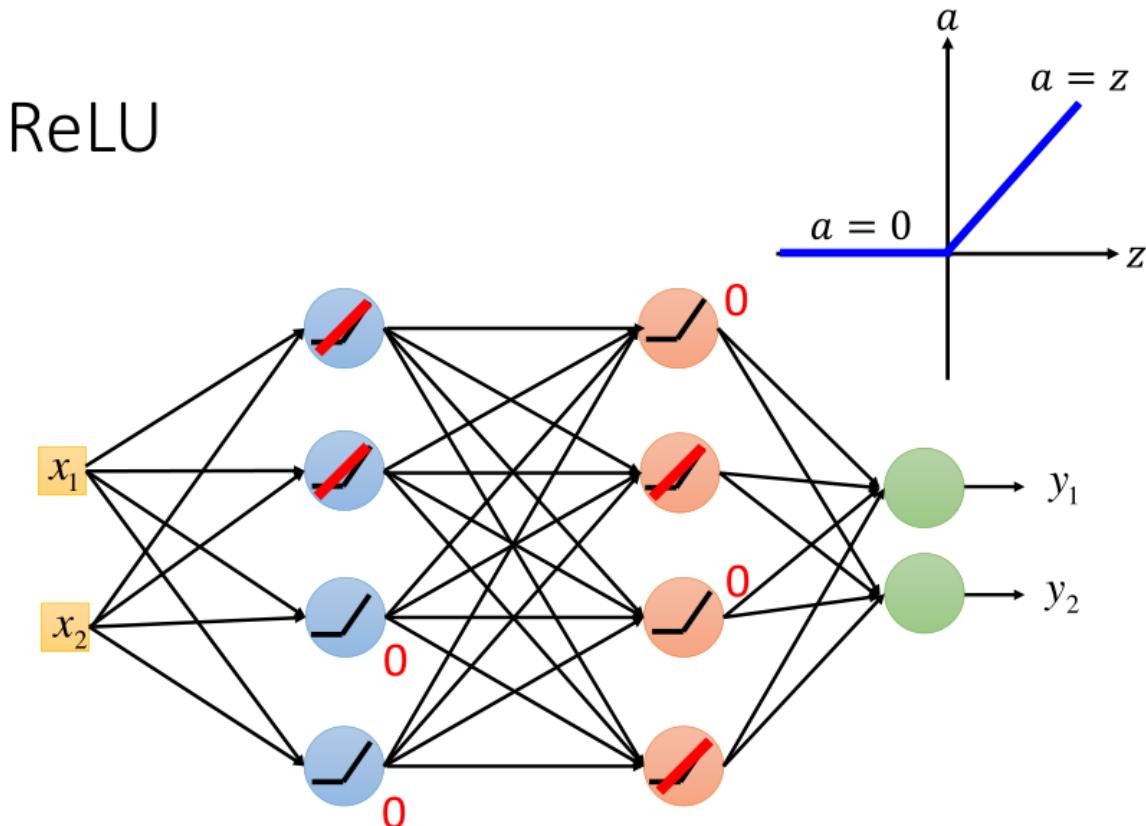
选择ReLU的理由如下：

- 跟sigmoid function比起来，ReLU的运算快很多
- ReLU的想法结合了生物上的观察 (Andrew)，跟人脑的神经脉冲很像，当 $z < 0$ 时，神经元是没有信号的。但是在sigmoid中，当 $z = 0$ 时，神经元输出为0.5，就是说神经元无论何时将会处于亢奋的状态，这与实际情况是相违背的
- 无穷多bias不同的sigmoid function叠加的结果会变成ReLU (Hitton)
- ReLU可以处理Vanishing gradient的问题 (the most important reason)

Handle Vanishing gradient problem

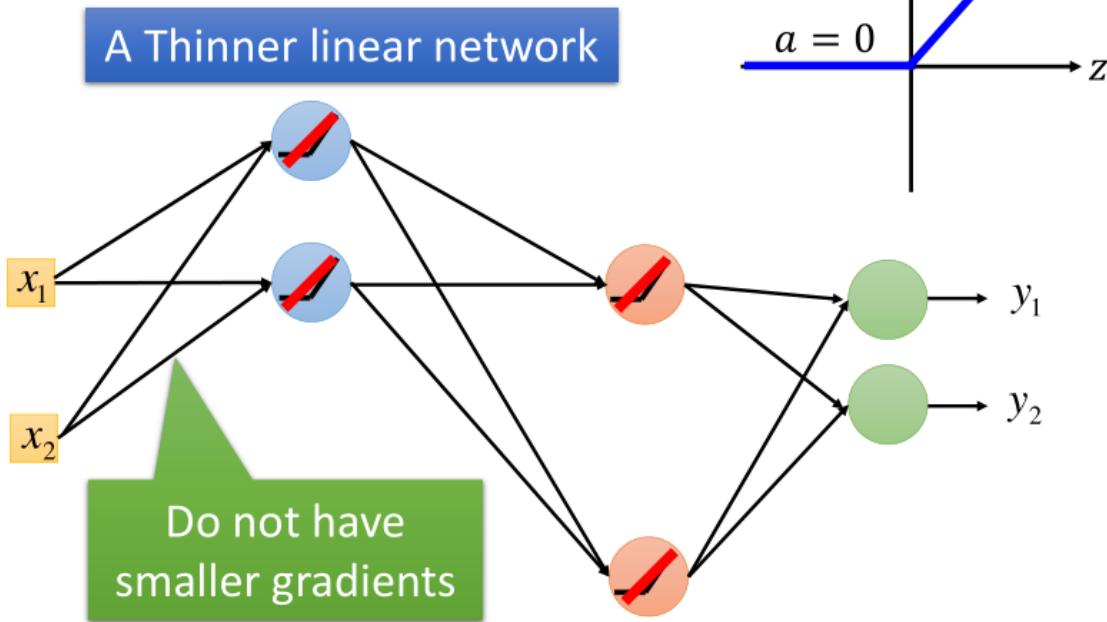
下图是ReLU的neural network，以ReLU作为activation function的neuron，它的output要么等于0，要么等于input

当output=input的时候，这个activation function就是linear的；而output=0的neuron对整个network是没有任何作用的，因此可以把它们从network中拿掉



拿掉所有output为0的neuron后如下图所示，此时整个network就变成a thinner linear network，linear的好处是，output=input，不会像sigmoid function一样使input产生的影响逐层递减

ReLU



Q: 这里就会有一个问题，我们之所以使用deep learning，就是因为想要一个non-linear、比较复杂的function，而使用ReLU不就会让它变成一个linear function吗？这样得到的function不是会变得很弱吗？

A: 其实，使用ReLU之后的network整体来说还是non-linear的，如果你对input做小小的改变，不改变neuron的operation region的话，那network就是一个linear function；但是，如果你对input做比较大的改变，导致neuron的operation region被改变的话，比如从output=0转变到了output=input，network整体上就变成了non-linear function

这里的region是指input $z < 0$ 和 $z > 0$ 的两个范围

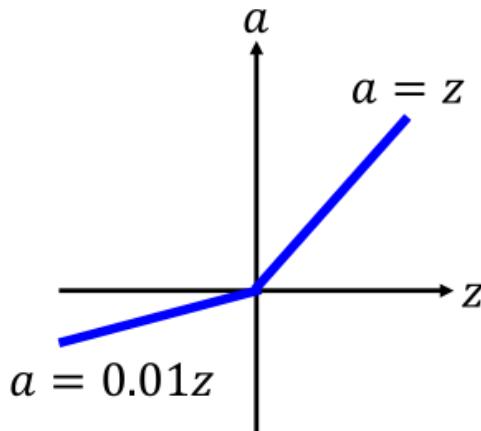
Q: 还有另外一个问题，我们对loss function做gradient descent，要求neural network是可以做微分的，但ReLU是一个分段函数，它是不能微分的（至少在 $z=0$ 这个点是不可微的），那该怎么办呢？

A: 在实际操作上，当region的范围处于 $z > 0$ 时，微分值gradient就是1；当region的范围处于 $z < 0$ 时，微分值gradient就是0；当 z 为0时，就不要管它

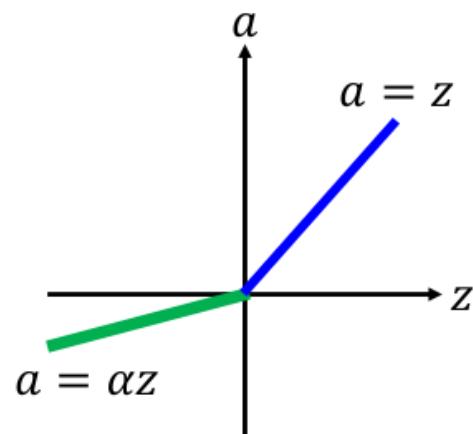
ReLU-variant

其实ReLU还存在一定的问题，比如当input<0的时候，output=0，此时微分值gradient也为0，你就没有办法去update参数了，所以我们应该让input<0的时候，微分后还能有一点点的值，比如令 $a = 0.01z$ ，这个东西就叫做Leaky ReLU

Leaky ReLU



Parametric ReLU



α also learned by gradient descent

既然 a 可以等于 $0.01z$, 那这个 z 的系数可不可以是 0.07 、 0.08 之类呢? 所以就有人提出了**Parametric ReLU**, 也就是令 $a = \alpha z$, 其中 α 并不是固定的值, 而是network的一个参数, 它可以通过training data学出来, 甚至每个neuron都可以有不同的 α 值

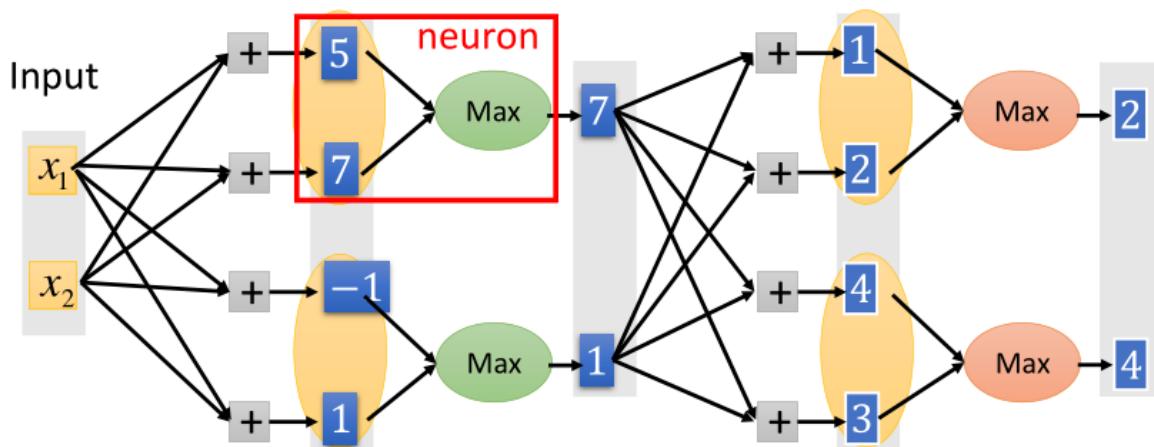
这个时候又有人想, 为什么一定要是ReLU这样子呢, activation function可不可以有别的样子呢? 所以后来有了一个更进阶的想法, 叫做**Maxout network**

Maxout

Maxout的想法是, 让network自动去学习它的activation function, 那Maxout network就可以自动学出ReLU, 也可以学出其他的activation function, 这一切都是由training data来决定的

假设现在有input x_1, x_2 , 它们乘上几组不同的weight分别得到5,7,-1,1, 这些值本来是不同neuron的input, 它们要通过activation function变为neuron的output; 但在Maxout network里, 我们事先决定好将某几个“neuron”的input分为一个group, 比如5,7分为一个group, 然后在这个group里选取一个最大值7作为output

- Learnable activation function [Ian J. Goodfellow, ICML'13]



You can have more than 2 elements in a group.

这个过程就好像在一个layer上做Max Pooling一样，它和原来的network不同之处在于，它把原来几个“neuron”的input按一定规则组成了一个group，然后并没有使它们通过activation function，而是选取其中的最大值当做这几个“neuron”的output

当然，实际上原来的“neuron”早就已经不存在了，这几个被合并的“neuron”应当被看做一个新的neuron，这个新的neuron的input是原来几个“neuron”的input组成的vector，output则取input的最大值，而并非由activation function产生

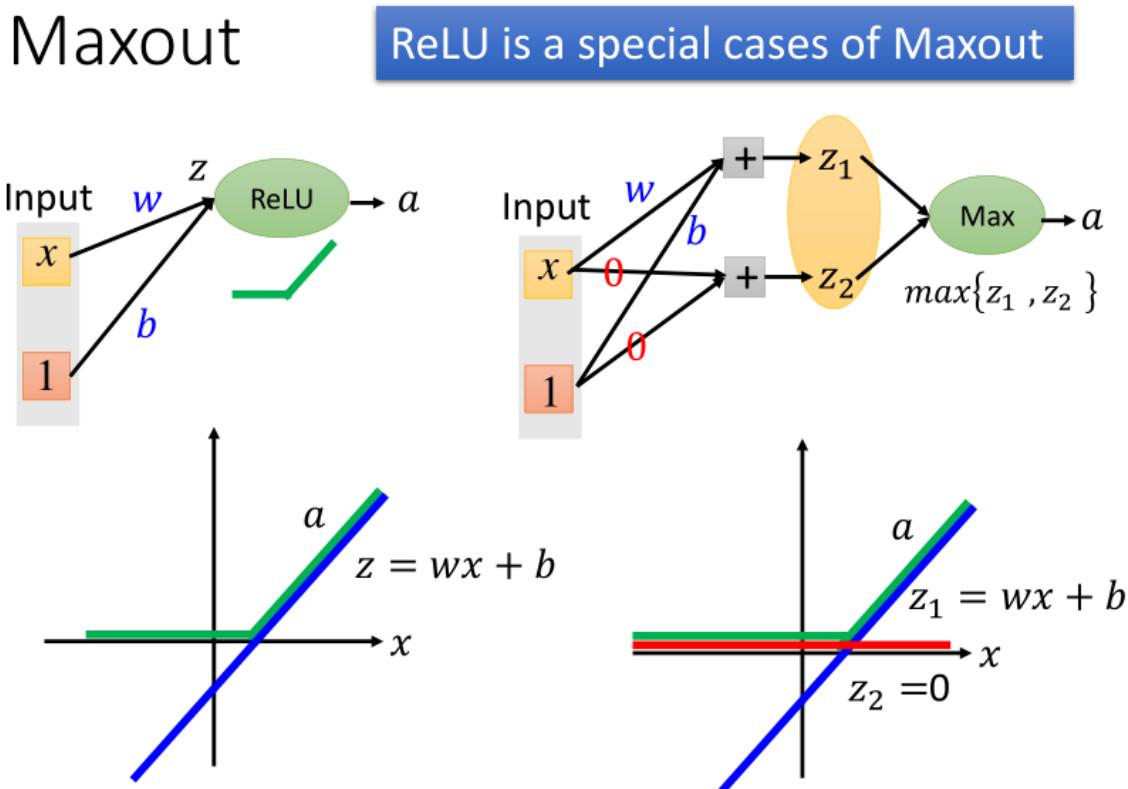
在实际操作上，几个element被分为一个group这件事情是由你自己决定的，它就是network structure里一个需要被调的参数，不一定要跟上图一样两个分为一组

Maxout → ReLU

Maxout是如何模仿出ReLU这个activation function的呢？

下图左上角是一个ReLU的neuron，它的input x 会乘上neuron的weight w ，再加上bias b ，然后通过activation function-ReLU，得到output a

- neuron的input为 $z = wx + b$ ，为下图左下角紫线
- neuron的output为 $a = z (z > 0); a = 0 (z < 0)$ ，为下图左下角绿线



如果我们使用的是上图右上角所示的Maxout network，假设 z_1 的参数 w 和 b 与ReLU的参数一致，而 z_2 的参数 w 和 b 全部设为0，然后做Max Pooling，选取 z_1, z_2 较大值作为 a

- neuron的input为 $[z_1 \ z_2]$
 - $z_1 = wx + b$ ，为上图右下角紫线
 - $z_2 = 0$ ，为上图右下角红线
- neuron的output为 $\max [z_1 \ z_2]$ ，为上图右下角绿线

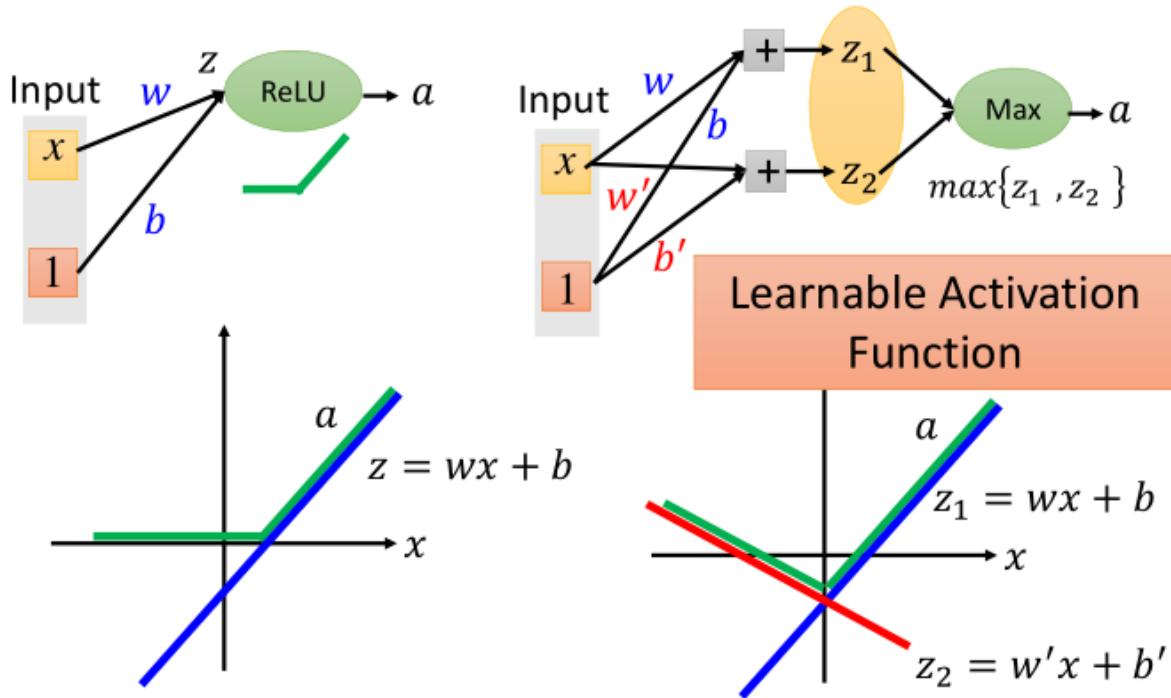
你会发现，此时ReLU和Maxout所得到的output是一模一样的，它们是相同的activation function

Maxout → More than ReLU

除了ReLU， Maxout还可以实现更多不同的activation function

比如 z_2 的参数w和b不是0，而是 w', b' ，此时

- neuron的input为 $[z_1 z_2]$
 - $z_1 = wx + b$, 为下图右下角紫线
 - $z_2 = w'x + b'$, 为下图右下角红线
- neuron的output为 $\max [z_1 z_2]$, 为下图右下角绿线

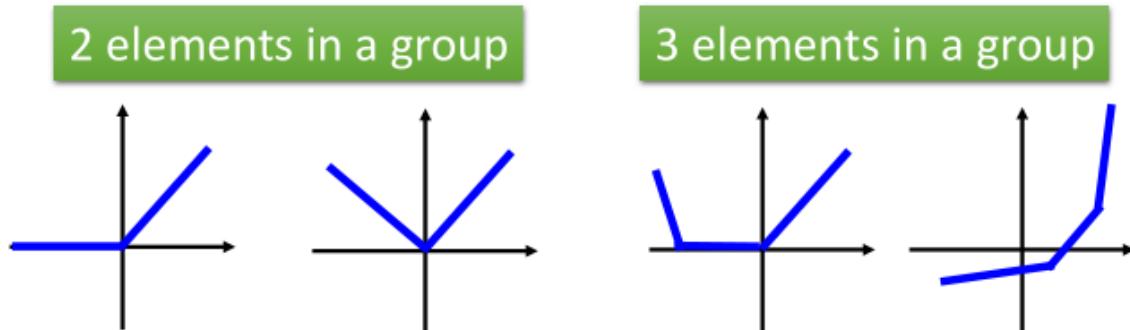


这个时候你得到的activation function的形状(绿线形状)，是由network的参数 w, b, w', b' 决定的，因此它是一个**Learnable Activation Function**，具体的形状可以根据training data去generate出来

Property

Maxout可以实现任何piecewise linear convex activation function（分段线性凸激活函数），其中这个activation function被分为多少段，取决于你把多少个element z 放到一个group里，下图分别是2个element一组和3个element一组的activation function的不同形状

- Learnable activation function [Ian J. Goodfellow, ICML'13]
 - Activation function in maxout network can be any piecewise linear convex function
 - How many pieces depending on how many elements in a group

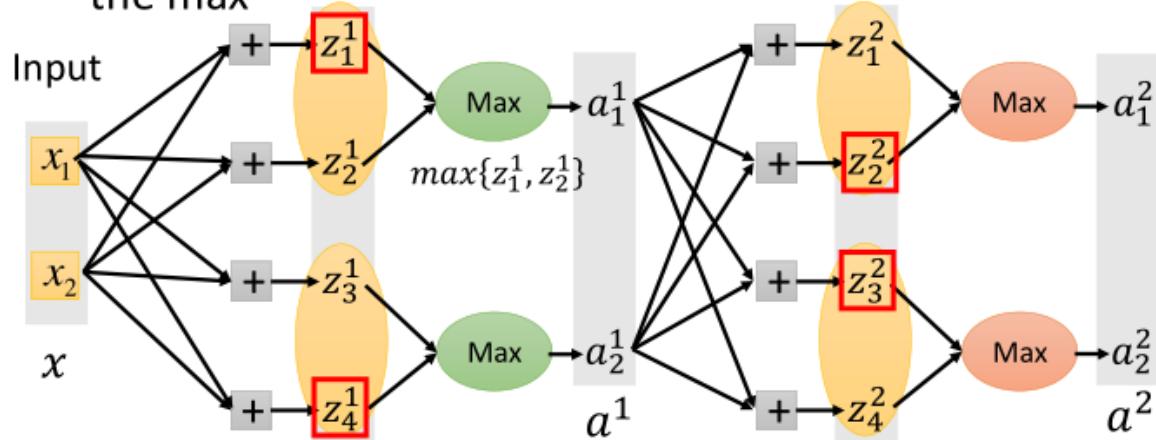


How to train Maxout

接下来我们要面对的是，怎么去train一个Maxout network，如何解决Max不能微分的问题

假设在下面的Maxout network中，红框内为每个neuron的output

- Given a training data x , we know which z would be the max

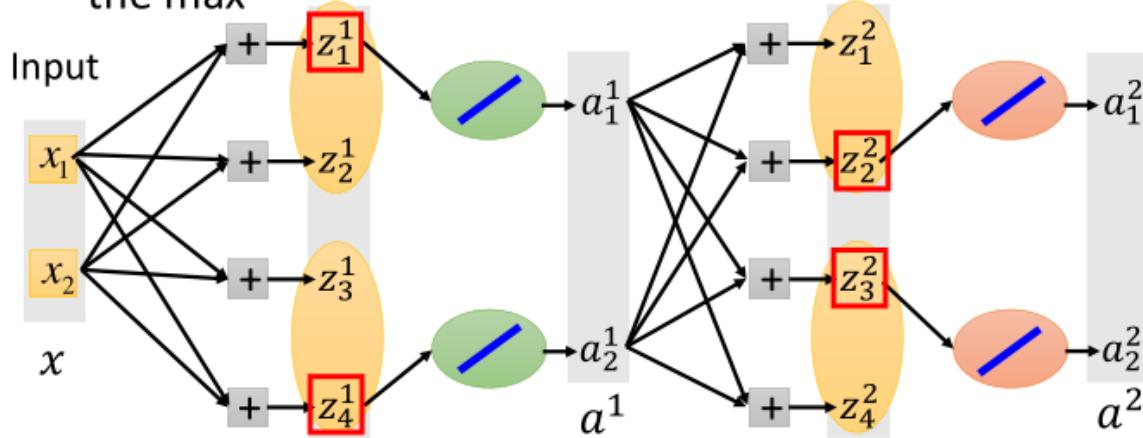


其实Max operation就是linear的operation，只是它仅接在前面这个group里的某一个element上，因此我们可以把那些并没有被Max连接到的element通通拿掉，从而得到一个比较细长的linear network

实际上我们真正训练的并不是一个含有max函数的network，而是一个化简后如下图所示的linear network；当我们还没有真正开始训练模型的时候，此时这个network含有max函数无法微分，但是只要真的丢进去了一笔data，network就会马上根据这笔data确定具体的形状，此时max函数的问题已经被实际数据给解决了，所以我们完全可以根据这笔training data使用Backpropagation的方法去训练被network留下来的参数

所以我们担心的max函数无法微分，它只是理论上的问题；在具体的实践上，我们完全可以先根据data把max函数转化为某个具体的函数，再对这个转化后的thinner linear network进行微分

- Given a training data x , we know which z would be the max



- Train this thin and linear network

Different thin and linear network for different examples

这个时候你也许会有一个问题，如果按照上面的做法，那岂不是只会train留在network里面的那些参数，剩下的参数该怎么办？那些被拿掉的直线（weight）岂不是永远也train不到了吗？

其实这也只是个理论上的问题，在实际操作上，我们之前已经提到过，每个linear network的structure都是由input的那一笔data来决定的，当你input不同data的时候，得到的network structure是不同的，留在network里面的参数也是不同的，由于我们有很多很多笔training data，所以network的structure在训练中不断地变换，实际上最后每一个weight参数都会被train到

所以，我们回到Max Pooling的问题上来，由于Max Pooling跟Maxout是一模一样的operation，既然如何训练Maxout的问题可以被解决，那训练Max Pooling又有什么困难呢？

Max Pooling有关max函数的微分问题采用跟Maxout一样的方案即可解决

Adaptive learning rate

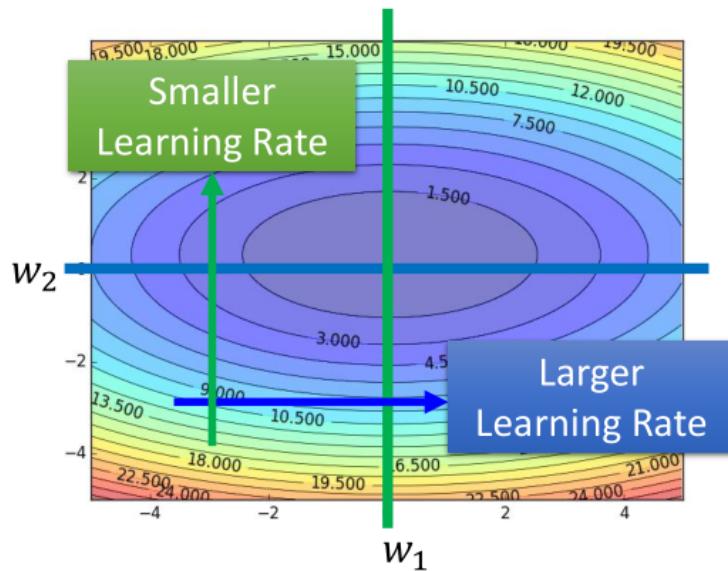
Review - Adagrad

我们之前已经了解过Adagrad的做法，让每一个parameter都要有不同的learning rate，

$$w^{t+1} = w^t - \frac{\eta}{\sqrt{\sum_{i=0}^t (g^i)^2}} \cdot g^t$$

Adagrad的精神是，假设我们考虑两个参数 w_1, w_2 ，如果在 w_1 这个方向上，平常的gradient都比较小，那它是比较平坦的，于是就给它比较大的learning rate；反过来说，在 w_2 这个方向上，平常gradient都比较大，那它是比较陡峭的，于是给它比较小的learning rate

Review



Adagrad

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^t (g^i)^2}} g^t$$

Use first derivative to estimate second derivative

但我们实际面对的问题，很有可能远比Adagrad所能解决的问题要来的复杂，我们之前做Linear Regression的时候，我们做optimization的对象，也就是loss function，它是convex的形状；但实际上我们在做deep learning的时候，这个loss function可以是任何形状

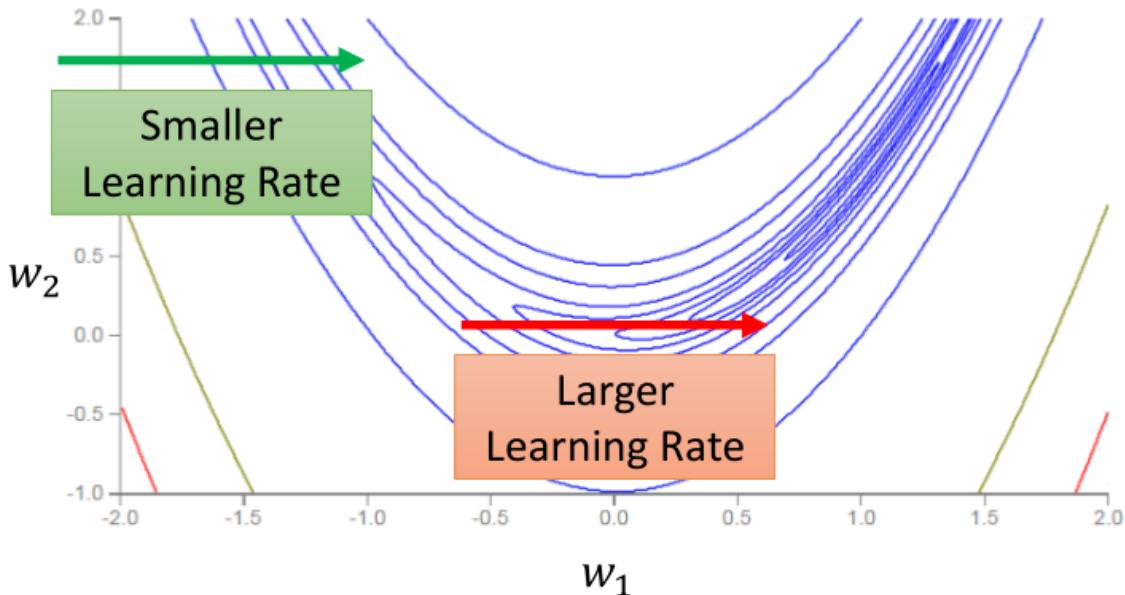
RMSProp

learning rate

loss function可以是任何形状，对convex loss function来说，在每个方向上它会一直保持平坦或陡峭的状态，所以你只需要针对平坦的情况设置较大的learning rate，对陡峭的情况设置较小的learning rate即可

但是在下图所示的情况下，即使是在同一个方向上(如 w_1 方向)，loss function也有可能一会儿平坦一会儿陡峭，所以你要随时根据gradient的大小来快速地调整learning rate

Error Surface can be very complex when training NN.



所以真正要处理deep learning的问题，用Adagrad可能是不够的，你需要更dynamic的调整learning rate的方法，所以产生了Adagrad的进阶版——**RMSProp**

RMSProp还是一个蛮神奇的方法，因为它并不是在paper里提出来的，而是Hinton在mooc的course里面提出来的一个方法，所以需要cite的时候，要去cite Hinton的课程链接

how to do RMSProp

RMSProp的做法如下：

我们的learning rate依旧设置为一个固定的值 η 除掉一个变化的值 σ ，这个 σ 等于上一个 σ 和当前梯度 g 的加权方均根（特别的是，在第一个时间点， σ^0 就是第一个算出来的gradient值 g^0 ），即：

$$w^{t+1} = w^t - \frac{\eta}{\sigma^t} g^t$$

$$\sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

这里的 α 值是可以自由调整的，RMSProp跟Adagrad不同之处在于，Adagrad的分母是对过程中所有的 gradient 取平方和开根号，也就是说 Adagrad 考虑的是整个过程平均的 gradient 信息；

而 RMSProp 虽然也是对所有的 gradient 进行平方和开根号，但是它用一个 α 来调整对不同 gradient 的使用程度，比如你把 α 的值设的小一点，意思就是你更倾向于相信新的 gradient 所告诉你的 error surface 的平滑或陡峭程度，而比较无视于旧的 gradient 所提供给你的 information

$$\begin{aligned}
 w^1 &\leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 & \sigma^0 &= g^0 \\
 w^2 &\leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 & \sigma^1 &= \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2} \\
 w^3 &\leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 & \sigma^2 &= \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2} \\
 &\vdots && \\
 w^{t+1} &\leftarrow w^t - \frac{\eta}{\sigma^t} g^t & \sigma^t &= \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}
 \end{aligned}$$

Root Mean Square of the gradients
with previous gradients being decayed

所以当你做RMSProp的时候，一样是在算gradient的root mean square，但是你可以给现在已经看到的gradient比较大的weight，给过去看到的gradient比较小的weight，来调整对gradient信息的使用程度

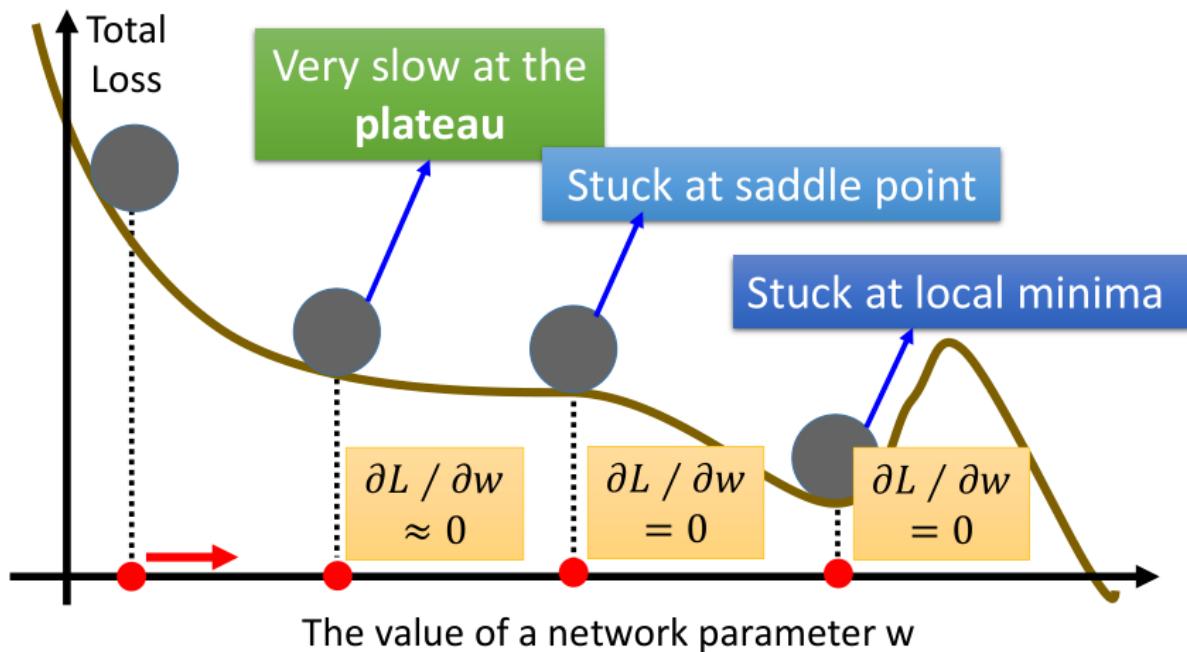
Momentum

optimization - local minima?

除了learning rate的问题以外，在做deep learning的时候，也会出现卡在local minimum、saddle point或是plateau的地方，很多人都会担心，deep learning这么复杂的model，可能非常容易就会被卡住了

但其实Yann LeCun在07年的时候，就提出了一个蛮特别的说法，他说你不要太担心local minima的问题，因为一旦出现local minima，它就必须在每一个dimension都是下图中这种山谷的低谷形状，假设山谷的低谷出现的概率为p，由于我们的network有非常非常多的参数，这里假设有1000个参数，每一个参数都要位于山谷的低谷之处，这件事发生的概率为 p^{1000} ，当你的network越复杂，参数越多，这件事发生的概率就越低

Hard to find optimal network parameters

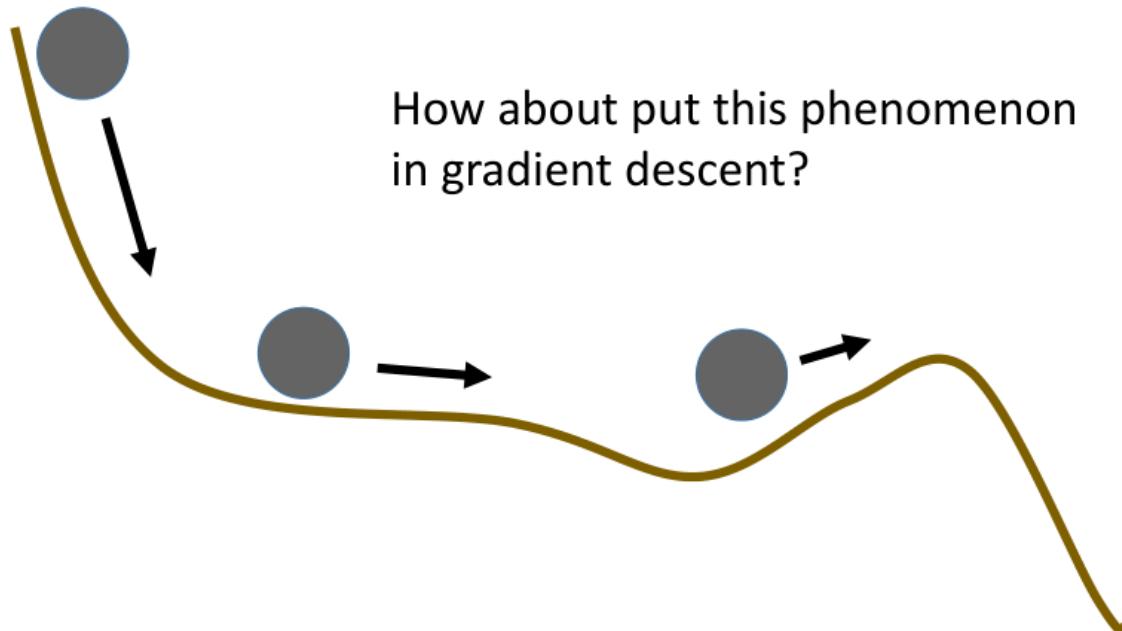


所以在一个很大的neural network里面，其实并没有那么多的local minima，搞不好它看起来其实是很平滑的，所以当你走到一个你觉得是local minima的地方被卡住了，那它八成就是global minima，或者是很接近global minima的地方

where is Momentum from

有一个heuristic (启发性) 的方法可以稍微处理一下上面所说的“卡住”的问题，它的灵感来自于真实世界。假设在有一个球从左上角滚下来，它会滚到plateau的地方、local minima的地方，但是由于惯性它还会继续往前走一段路程，假设前面的坡没有很陡，这个球就很有可能翻过山坡，走到比local minima还要好的地方

• Momentum



所以我们要做的，就是把**惯性**塞到gradient descent里面，这件事情就叫做**Momentum**

How to do Momentum

当我们在gradient descent里加上Momentum的时候，每一次update的方向，不再只考虑gradient的方向，还要考虑上一次update的方向，那这里我们就用一个变量 v 去记录前一个时间点update的方向

随机选一个初始值 θ^0 ，初始化 $v^0 = 0$ ，接下来计算 θ^0 处的gradient，然后我们要移动的方向是由前一个时间点的移动方向 v^0 和gradient的反方向 $\nabla L(\theta^0)$ 来决定的，即

$$v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$$

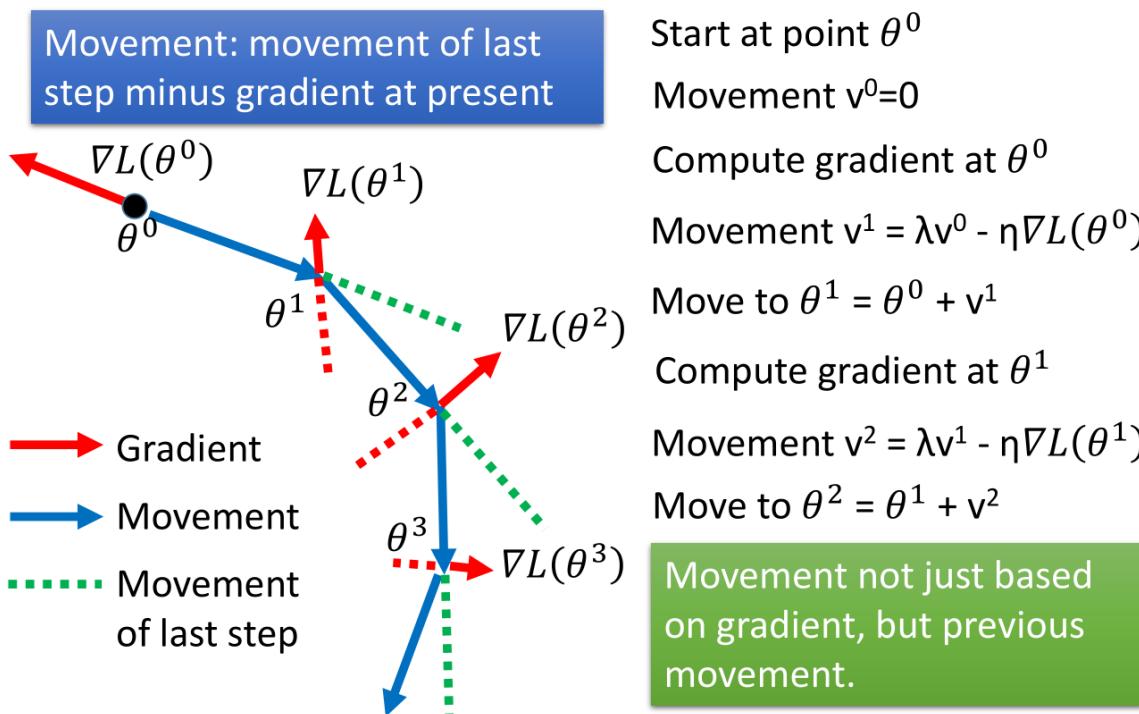
这里的 λ 也是一个手动调整的参数，它表示惯性对前进方向的影响有多大

接下来我们第二个时间点要走的方向 v^2 ，它是由第一个时间点移动的方向 v^1 和gradient的反方向 $\nabla L(\theta^1)$ 共同决定的

λv 是图中的绿色虚线，它代表由于上一次的惯性想要继续走的方向

$\eta \nabla L(\theta)$ 是图中的红色虚线，它代表这次gradient告诉你所要移动的方向

它们的矢量和就是这一次真实移动的方向，为蓝色实线



gradient告诉我们走红色虚线的方向，惯性告诉我们走绿色虚线的方向，合起来就是走蓝色的方向

我们还可以用另一种方法来理解Momentum这件事，其实你在每一个时间点移动的步伐 v^i ，包括大小和方向，就是过去所有gradient的加权和

具体推导如下图所示，第一个时间点移动的步伐 v^1 是 θ^0 处的gradient加权，第二个时间点移动的步伐 v^2 是 θ^0 和 θ^1 处的gradient加权和...以此类推；由于 λ 的值小于1，因此该加权意味着越是之前的gradient，它的权重就越小，也就是说，你更在意的是现在的gradient，但是过去的所有gradient也要对你现在update的方向有一定程度的影响力，这就是Momentum

Movement: movement of last step minus gradient at present

v^i is actually the weighted sum of all the previous gradient:
 $\nabla L(\theta^0), \nabla L(\theta^1), \dots \nabla L(\theta^{i-1})$

$$v^0 = 0$$

$$v^1 = -\eta \nabla L(\theta^0)$$

$$v^2 = -\lambda \eta \nabla L(\theta^0) - \eta \nabla L(\theta^1)$$

⋮

Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

Movement $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Movement $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

Movement not just based on gradient, but previous movement

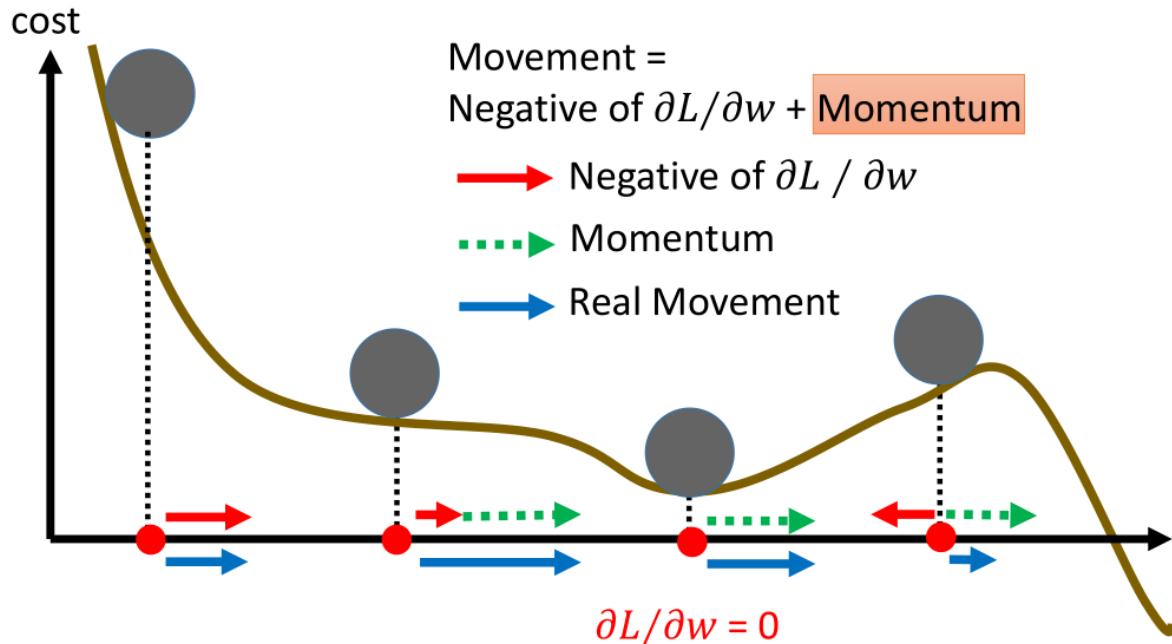
如果你对数学公式不太喜欢的话，那我们就从直觉上来看一下加入Momentum之后是怎么运作的

在加入Momentum以后，每一次移动的方向，就是negative的gradient加上Momentum建议我们要走的方向，Momentum其实就是上一个时间点的movement

下图中，红色实线是gradient建议我们走的方向，直观上看就是根据坡度要走的方向；绿色虚线是Momentum建议我们走的方向，实际上就是上一次移动的方向；蓝色实线则是最终真正走的方向

Momentum

Still not guarantee reaching global minima, but give some hope



如果我们今天走到local minimum的地方，此时gradient是0，红色箭头没有指向，它就会告诉你就停在这里吧，但是Momentum也就是绿色箭头，它指向右侧就是告诉你之前是要走向右侧的，所以你仍然应该要继续往右走，所以最后你参数update的方向仍然会继续向右；你甚至可以期待Momentum比较强，惯性的力量可以支撑着你走出这个谷底，去到loss更低的地方

Adam

其实RMSProp加上Momentum，就可以得到Adam

根据下面的paper来快速描述一下Adam的algorithm：

- 先初始化 $m_0 = 0$, m_0 就是Momentum中, 前一个时间点的movement

再初始化 $v_0 = 0$, v_0 就是RMSProp里计算gradient的root mean square的 σ

最后初始化 $t = 0$, t用来表示时间点

- 先算出gradient g_t

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

- 再根据过去要走的方向 m_{t-1} 和gradient g_t , 算出现在要走的方向 m_t ——Momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

- 然后根据前一个时间点的 v_{t-1} 和gradient g_t 的平方, 算一下放在分母的 v_t ——RMSProp

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- 接下来做了一个原来RMSProp和Momentum里没有的东西, 就是bias correction, 它使 m_t 和 v_t 都除上一个值, 分母这个值本来比较小, 后来会越来越接近于1 (原理详见paper)

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- 最后做update, 把Momentum建议你的方向 \hat{m}_t 乘上learning rate α , 再除掉RMSProp normalize后建议的learning rate分母, 然后得到update的方向

$$\theta_t = \theta_{t-1} - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Adam

RMSProp + Momentum

Algorithm 1: Adam, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1]$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector) → for momentum

$v_0 \leftarrow 0$ (Initialize 2nd moment vector) → for RMSprop

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Good Results on Testing Data?

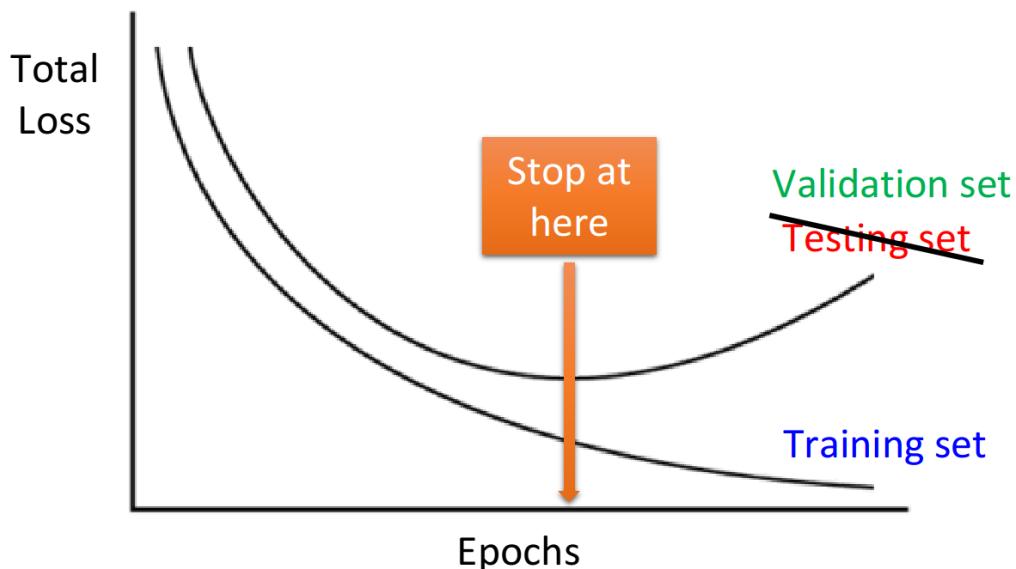
在Testing data上得到更好的performance，分为三个模块，Early Stopping和Regularization是很typical的做法，它们不是特别为deep learning所设计的；而Dropout是一个蛮有deep learning特色的做法

Early Stopping

假设你今天的learning rate调的比较好，那随着训练的进行，total loss通常会越来越小，但是Training set和Testing set的情况并不是完全一样的，很有可能当你在Training set上的loss逐渐减小的时候，在Testing set上的loss反而上升了

所以，理想上假如你知道testing data上的loss变化情况，你会在testing set的loss最小的时候停下来，而不是在training set的loss最小的时候停下来；但testing set实际上是未知的东西，所以我们需要用validation set来替代它去做这件事情

Early Stopping



Keras: <http://keras.io/getting-started/faq/#how-can-i-interrupt-training-when-the-validation-loss-isnt-decreasing-anymore>

很多时候，我们所讲的“testing set”并不是指代那个未知的数据集，而是一些已知的被你拿来做测试之用的数据集，比如kaggle上的public set，或者是你自己切出来的validation set

Regularization

regularization就是在原来的loss function上额外增加几个term，比如我们要minimize的loss function原先应该是square error或cross entropy，那在做Regularization的时候，就在后面加一个Regularization的term

L2 regularization

regularization term可以是参数的L2 norm，所谓的L2 norm，就是把model参数集 θ 里的每一个参数都取平方然后求和，这件事被称作L2 regularization，即

$$L2 \text{ regularization} : \|\theta\|_2 = (w_1)^2 + (w_2)^2 + \dots$$

Regularization

- New loss function to be minimized
 - Find a set of weight not only minimizing original cost but also close to zero

$$L'(\theta) = \underbrace{L(\theta)}_{\text{Original loss}} + \lambda \frac{1}{2} \|\theta\|_2^2 \rightarrow \text{Regularization term}$$

$\theta = \{w_1, w_2, \dots\}$

L2 regularization:

(e.g. minimize square error, cross entropy ...) $\|\theta\|_2^2 = (w_1)^2 + (w_2)^2 + \dots$

(usually not consider biases)

通常我们在做regularization的时候，新加的term里是不会考虑bias这一项的，因为加regularization的目的是为了让我们的function更平滑，而bias通常是跟function的平滑程度没有关系的

你会发现我们新加的regularization term $\lambda \frac{1}{2} \|\theta\|_2^2$ 里有一个 $\frac{1}{2}$ ，由于我们是要对loss function求微分的，而新加的regularization term是参数 w_i 的平方和，对平方求微分会多出来一个系数2，我们的 $\frac{1}{2}$ 就是用来和这个2相消的

L2 regularization具体工作流程如下：

- 我们加上regularization term之后得到了一个新的loss function: $L'(\theta) = L(\theta) + \lambda \frac{1}{2} \|\theta\|_2^2$
- 将这个loss function对参数 w_i 求微分，gradient: $\frac{\partial L'}{\partial w_i} = \frac{\partial L}{\partial w_i} + \lambda w_i$
- 然后update参数 w_i : $w_i^{t+1} = w_i^t - \eta \frac{\partial L'}{\partial w_i} = w_i^t - \eta (\frac{\partial L}{\partial w_i} + \lambda w_i^t) = (1 - \eta \lambda) w_i^t - \eta \frac{\partial L}{\partial w_i}$

如果把这个推导出来的式子和原式作比较，你会发现参数 w_i 在每次update之前，都会乘上一个 $(1 - \eta \lambda)$ ，而 η 和 λ 通常会被设为一个很小的值，因此 $(1 - \eta \lambda)$ 通常是一个接近于1的值，比如0.99

也就是说，regularization做的事情是，每次update参数 w_i 之前，不分青红皂白就先对原来的 w_i 乘个0.99，这意味着，随着update次数增加，参数 w_i 会越来越接近于0

Q: 你可能会问，要是所有的参数都越来越靠近0，那最后岂不是 w_i 通通变成0，得到的network还有什么用？

A: 其实不会出现最后所有参数都变为0的情况，因为通过微分得到的 $\eta \frac{\partial L}{\partial w_i}$ 这一项是会和前面 $(1 - \eta \lambda) w_i^t$ 这一项最后取得平衡的

使用L2 regularization可以让weight每次都变得更小一点，这就叫做**Weight Decay**(权重衰减)

L1 regularization

除了L2 regularization中使用平方项作为new term之外，还可以使用L1 regularization，把平方项换成每一个参数的绝对值，即

$$||\theta||_1 = |w_1| + |w_2| + \dots$$

Q: 你的第一个问题可能会是，绝对值不能微分啊，该怎么处理呢？

A: 实际上绝对值就是一个V字形的函数，在V的左边微分值是-1，在V的右边微分值是1，只有在0的地方是不能微分的，那真的走到0的时候就胡乱给它一个值，比如0，就OK了

如果w是正的，那微分出来就是+1，如果w是负的，那微分出来就是-1，所以这边写了一个w的sign function，它的意思是说，如果w是正数的话，这个function output就是+1，w是负数的话，这个function output就是-1

L1 regularization的工作流程如下：

- 我们加上regularization term之后得到了一个新的loss function: $L'(\theta) = L(\theta) + \lambda \frac{1}{2} ||\theta||_1$
- 将这个loss function对参数 w_i 求微分: $\frac{\partial L'}{\partial w_i} = \frac{\partial L}{\partial w_i} + \lambda sgn(w_i)$
- 然后update参数 w_i :

$$w_i^{t+1} = w_i^t - \eta \frac{\partial L'}{\partial w_i} = w_i^t - \eta \left(\frac{\partial L}{\partial w_i} + \lambda sgn(w_i^t) \right) = w_i^t - \eta \frac{\partial L}{\partial w_i} - \eta \lambda sgn(w_i^t)$$

这个式子告诉我们，每次update的时候，不管三七二十一都要减去一个 $\eta \lambda sgn(w_i^t)$ ，如果w是正的，sgn是+1，就会变成减一个positive的值让你的参数变小；如果w是负的，sgn是-1，就会变成加一个值让你的参数变大；总之就是让它们的绝对值减小至接近于0

L1 v.s. L2

我们来对比一下L1和L2的update过程：

$$\begin{aligned} L1 : w_i^{t+1} &= w_i^t - \eta \frac{\partial L}{\partial w_i} - \eta \lambda sgn(w_i^t) \\ L2 : w_i^{t+1} &= (1 - \eta \lambda) w_i^t - \eta \frac{\partial L}{\partial w_i} \end{aligned}$$

L1和L2，虽然它们同样是让参数的绝对值变小，但它们做的事情其实略有不同：

- L1使参数绝对值变小的方式是每次update减掉一个固定的值
- L2使参数绝对值变小的方式是每次update乘上一个小于1的固定值

因此，当参数w的绝对值比较大的时候，L2会让w下降得更快，而L1每次update只让w减去一个固定的值，train完以后可能还会有很多比较大的参数

当参数w的绝对值比较小的时候，L2的下降速度就会变得很慢，train出来的参数平均都是比较小的，而L1每次下降一个固定的value，train出来的参数是比较sparse的，这些参数有很多是接近0的值，也会有很大的值

在的CNN的task里，用L1做出来的效果是比较合适的，是比较sparse的

Weight Decay

之前提到了Weight Decay，那实际上我们在人脑里面也会做Weight Decay

下图分别描述了，刚出生的时候，婴儿的神经是比较稀疏的；6岁的时候，就会有很多很多的神经；但是到14岁的时候，神经间的连接又减少了，所以neural network也会跟我们人有一些很类似的事情，如果有一些weight你都没有去update它，那它每次都会越来越小，最后就接近0然后不见了

这跟人脑的运作，是有异曲同工之妙

some tips

在deep learning里面，regularization虽然有帮助，但它的重要性往往没有SVM这类方法中来得高

因为我们在做neural network的时候，通常都是从一个很小的、接近于0的值开始初始参数的，而做update的时候，通常都是让参数离0越来越远，但是regularization要达到的目的，就是希望我们的参数不要离0太远

如果你做的是Early Stopping，它会减少update的次数，其实也会避免你的参数离0太远，这跟regularization做的事情是很接近的

所以在neural network里面，regularization的作用并没有SVM中来的重要，SVM其实是explicitly把regularization这件事情写在了它的objective function（目标函数）里面，SVM是要去解一个convex optimization problem，因此它解的时候不一定会有iteration的过程，它不会有Early Stopping这件事，而是一步就可以走到那个最好的结果了，所以你没有办法用Early Stopping防止它离目标太远，你必须要把regularization explicitly加到你的loss function里面去

在deep learning里面，regularization虽然有帮助，但是重要性相比在其他方法中没有那么高，regularization的帮助没有那么显著。

Early Stop可以决定什么时候training停下来，因为我们初试参数都是给一个很小的接近0的值，做update的时候，让参数离0越来越远，而regularization做的是让参数不要离0太远，因此regularization和减少update次数（Early Stop）的效果是很接近的。

因此在Neural Network里面，regularization虽然也有帮助，但是帮助没有那么重要，没有重要到SVM中那样。因为SVM的参数是一步走到结果，没有Early Stop

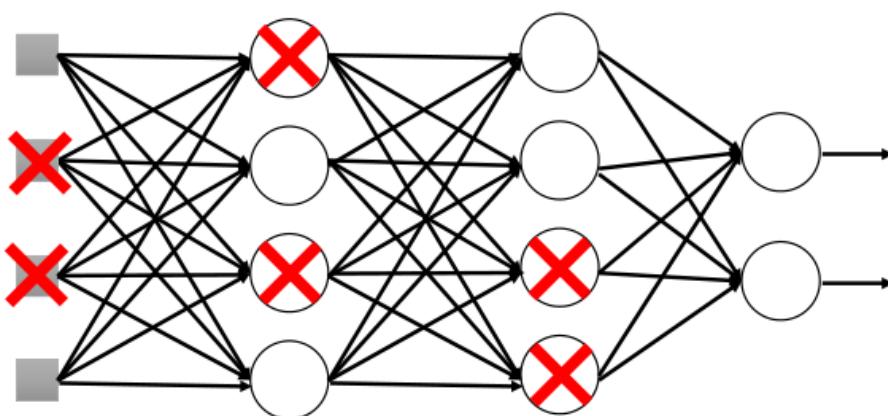
Dropout

How to do Dropout

Training

在training的时候，每次update参数之前，我们对每一个neuron（也包括input layer的“neuron”）做sampling，每个neuron都有p%的机率会被丢掉，如果某个neuron被丢掉的话，跟它相连的weight也都被丢掉

Training:



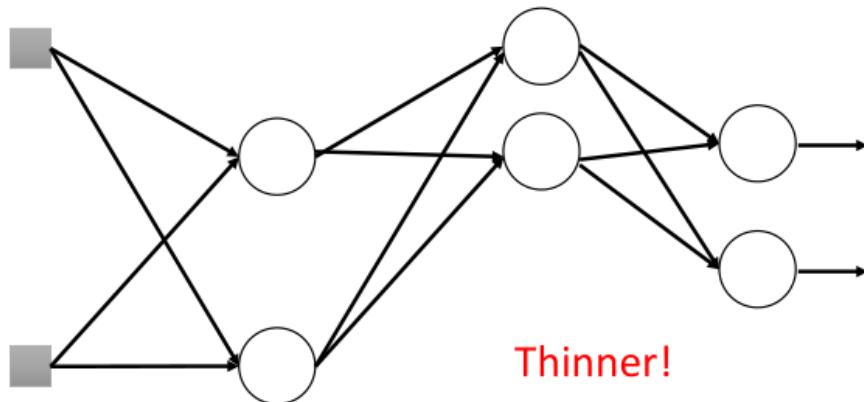
➤ Each time before updating the parameters

- Each neuron has p% to dropout

实际上就是每次update参数之前都通过抽样只保留network中的一部分neuron来做训练

做完sampling以后，network structure就会变得比较细长了，然后你再去train这个细长的network

Training:



➤ Each time before updating the parameters

- Each neuron has p% to dropout
→ **The structure of the network is changed.**
- Using the new network for training

For each mini-batch, we resample the dropout neurons

每次update参数之前都要做一遍sampling，所以每次update参数的时候，拿来training的network structure都是不一样的；你可能会觉得这个方法跟前面提到的Maxout会有一点像，但实际上，Maxout是每一笔data对应的network structure不同，而Dropout是每一次update的network structure都是不同的（每一个mini-batch对应着一次update，而一个mini-batch里含有很多笔data）

当你在training的时候使用dropout，得到的performance其实是会变差的，因为某些neuron在training的时候莫名其妙就会消失不见，但这并不是问题

因为Dropout真正要做的事情，就是要让你在training set上的结果变差，但是在testing set上的结果是变好的

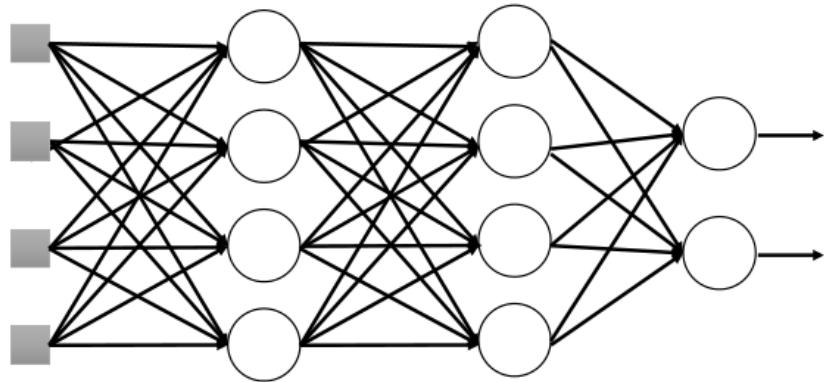
所以如果你今天遇到的问题是在training set上得到的performance不够好，你再加dropout，就只会越做越差

不同的problem需要用不同的方法去解决，而不是胡乱使用，dropout就是针对testing set的方法，当然不能够拿来解决training set上的问题

Testing

在使用dropout方法做testing的时候要注意两件事情：

Testing:



➤ No dropout

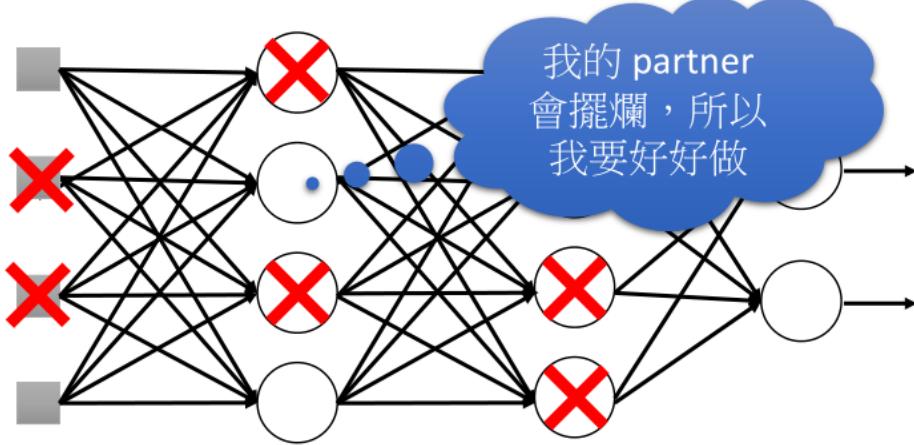
- If the dropout rate at training is $p\%$,
all the weights times $1-p\%$
 - Assume that the dropout rate is 50%.
If a weight $w = 1$ by training, set $w = 0.5$ for testing.
- testing的时候不做dropout，所有的neuron都要被用到
 - 假设在training的时候，dropout rate是 $p\%$ ，从training data中被learn出来的所有weight都要乘上 $(1-p\%)$ 才能被当做testing的weight使用

Intuitive Reason

直觉的想法是这样子：在training的时候，会丢掉一些neuron，就好像是你要练轻功的时候，会在脚上绑一些重物；然后，你在实际战斗的时候，就是实际testing的时候，是没有dropout的，就相当于把重物拿下来，所以你就会变得很强

另一个直觉的理由是这样，neural network里面的每一个neuron就是一个学生，那大家被连接在一起就是大家听说要组队做final project，那在一个团队里总是有人会拖后腿，就是他会dropout，所以假设你觉得自己的队友会dropout，这个时候你就会想要好好做，然后去carry这个队友，这就是training的过程

那实际在testing的时候，其实大家都有好好做，没有人需要被carry，由于每个人都比一般情况下更努力，所以得到的结果会是更好的，这也就是testing的时候不做dropout的原因



- When teams up, if everyone expect the partner will do the work, nothing will be done finally.
- However, if you know your partner will dropout, you will do better.
- When testing, no one dropout actually, so obtaining good results eventually.

为什么training和testing使用的weight是不一样的呢？

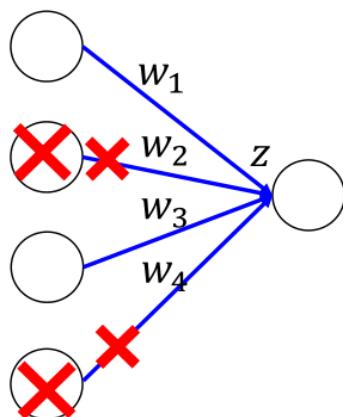
直觉的解释是这样的：

假设现在的dropout rate是50%，那在training的时候，你总是期望每次update之前会丢掉一半的neuron，就像下图左侧所示，在这种情况下你learn好了一组weight参数，然后拿去testing

但是在testing的时候是没有dropout的，所以如果testing使用的是和training同一组weight，那左侧得到的output z 和右侧得到的output z' ，它们的值其实是会相差两倍的，即 $z' \approx 2z$ ，这样会造成testing的结果与training的结果并不match，最终的performance反而会变差

Training of Dropout

Assume dropout rate is 50%



Testing of Dropout

No dropout

Weights from training

$$0.5 \times w_1 \rightarrow z' \approx 2z$$

$$0.5 \times w_2$$

$$0.5 \times w_3$$

$$0.5 \times w_4$$

Weights multiply $1-p\%$

$$\rightarrow z' \approx z$$

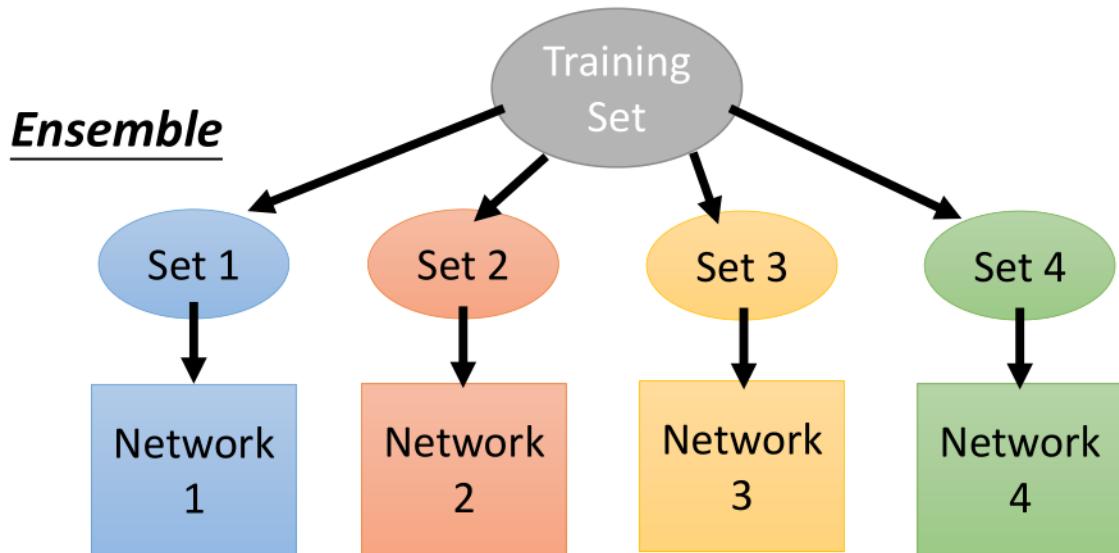
那这个时候，你就需要把右侧testing中所有的weight乘上0.5，然后做normalization，这样 z 就会等于 z' ，使得testing的结果与training的结果是比较match的

Dropout is a kind of ensemble

在文献上有很多不同的观点来解释为什么dropout会work，其中一种比较令人信服的解释是：dropout是一种终极的ensemble的方法

Ensemble

ensemble的方法在比赛的时候经常用得到，它的意思是说，我们有一个很大的training set，那你每次都只从这个training set里面sample一部分的数据出来，像下图一样，抽取了set 1, set 2, set 3, set 4



Train a bunch of networks with different structures

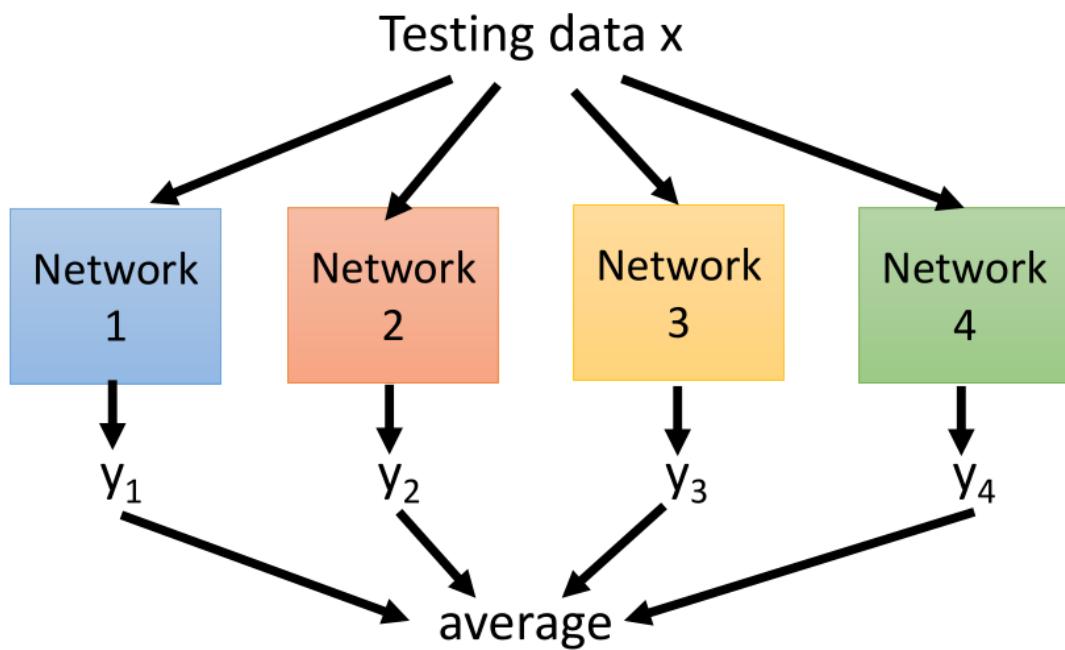
我们之前在讲bias和variance的trade off的时候说过，打靶有两种情况：

- 一种是因为bias大而导致打不准（参数过少）
- 另一种是因为variance大而导致打不准（参数过多）

假设我们今天有一个很复杂的model，它往往是bias比较准，但variance很大的情况，如果你有很多个笨重复杂的model，虽然它们的variance都很大，但最后平均起来，结果往往就会很准

所以ensemble做的事情，就是利用这个特性，我们从原来的training data里面sample出很多subset，然后train很多个model，每一个model的structure甚至都可以不一样；在testing的时候，丢一笔testing data进来，使它通过所有的model，得到一大堆的结果，然后把这些结果平均起来当做最后的output

Ensemble



如果你的model很复杂，这一招往往是很有效的，random forest也是实践这个精神的一个方法，也就是如果你用一个decision tree，它就会很弱，也很容易overfitting，而如果采用random forest，它就没有那么容易overfitting

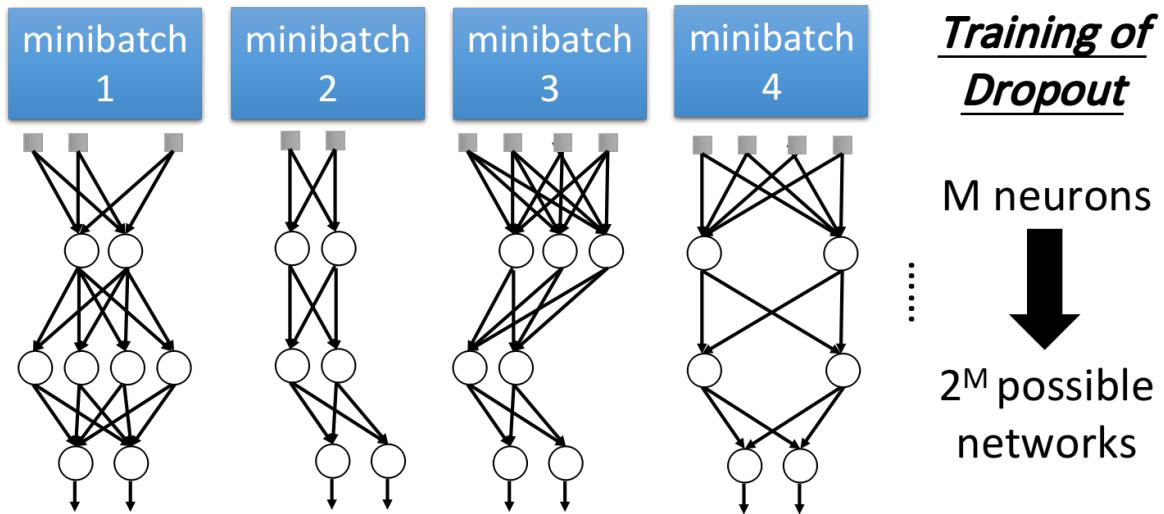
Dropout is a kind of ensemble

在training network的时候，每次拿一个mini-batch出来就做一次update，而根据dropout的特性，每次update之前都要对所有的neuron进行sample，因此每一个mini-batch所训练的network都是不同的

假设我们有 M 个neuron，每个neuron都有可能drop或不drop，所以总共可能的network数量有 2^M 个；所以当你在做dropout的时候，相当于是在用很多个mini-batch分别去训练很多个network（一个mini-batch设置为100笔data）

做了几次update，就相当于train了几个不同的network，最多可以训练到 2^M 个network

Dropout is a kind of ensemble.



- Using one mini-batch to train one network
- Some parameters in the network are shared

每个network都只用一个mini-batch的数据来train，可能会让人感到不安，一个batch才100笔data，怎么train一个network呢？

其实没有关系，因为这些不同的network之间的参数是shared，也就是说，虽然一个network只能用一个mini-batch来train，但同一个weight可以在不同的network里被不同的mini-batch train，所以同一个weight实际上是被所有没有丢掉它的network一起share的，它是拿所有这些network的mini-batch合起来一起train的结果

那按照ensemble这个方法的逻辑，在testing的时候，你把那train好的一大把network通通拿出来，然后把手上这一笔testing data丢到这把network里面去，每个network都给你吐出一个结果来，然后你把所有的结果平均起来，就是最后的output

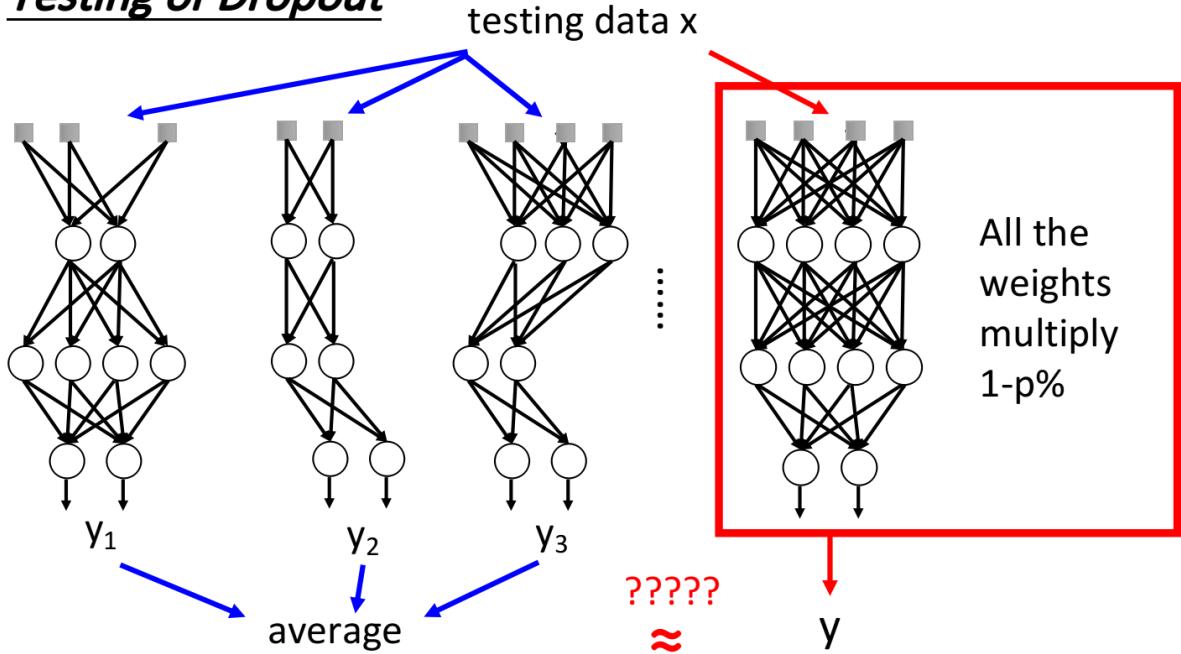
但是在实际操作上，如下图左侧所示，这一把network实在太多了，你没有办法每一个network都丢一个input进去，再把它们的output平均起来，这样运算量太大了

所以dropout最神奇的地方是，当你并没有把这些network分开考虑，而是用一个完整的network，这个network的weight是用之前那一把network train出来的对应weight乘上(1-p%)，然后再把手上这笔testing data丢进这个完整的network，得到的output跟network分开考虑的ensemble的output，是惊人的相近

也就是说下图左侧ensemble的做法和右侧dropout的做法，得到的结果是approximate(近似的)

Dropout is a kind of ensemble.

Testing of Dropout



这里用一个例子来解释：

我们train一个下图右上角所示的简单的network，它只有一个neuron，activation function是linear的，并且不考虑bias，这个network经过dropout训练以后得到的参数分别为 w_1, w_2 ，那给它input x_1, x_2 ，得到的output就是 $z = w_1 x_1 + w_2 x_2$

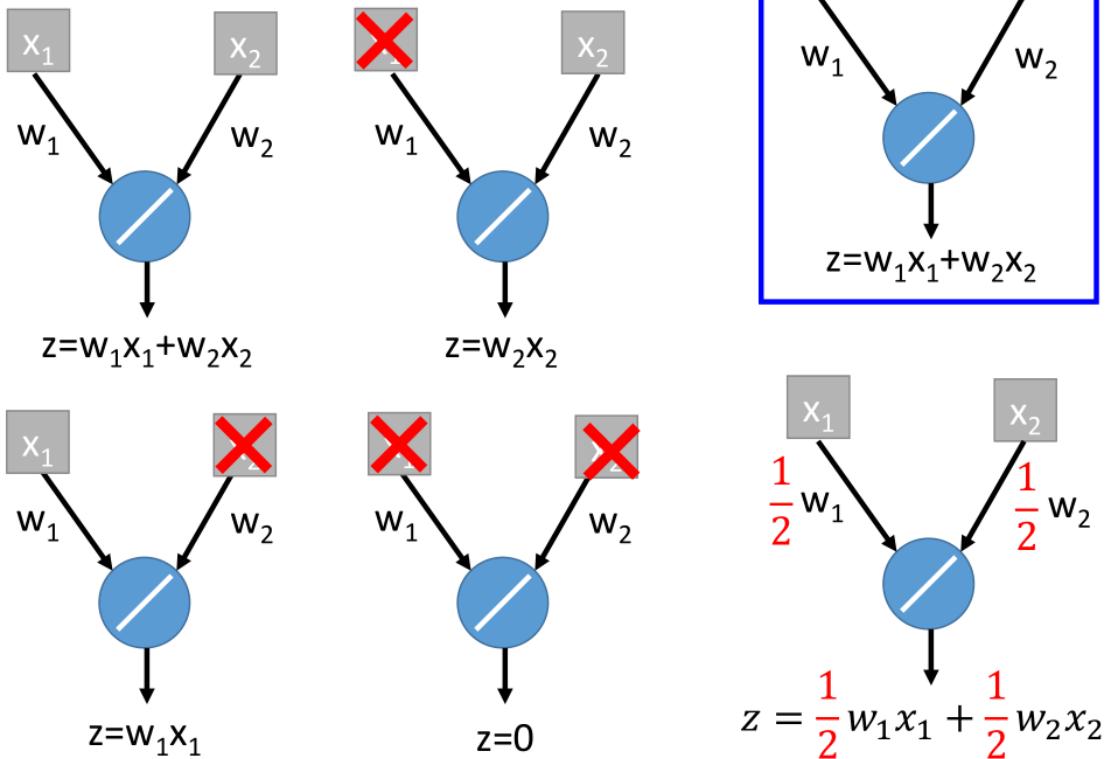
如果我们今天要做ensemble的话，theoretically就是像下图这么做，每一个neuron都有可能被drop或不drop，这里只有两个input的neuron，所以我们一共可以得到 $2^2=4$ 种network

我们手上这笔testing data x_1, x_2 丢到这四个network中，分别得到4个output：

$w_1 x_1 + w_2 x_2, w_2 x_2, w_1 x_1, 0$ ，然后根据ensemble的精神，把这四个network的output通通都average起来，得到的结果是 $\frac{1}{2}(w_1 x_1 + w_2 x_2)$

那根据dropout的想法，我们把从training中得到的参数 w_1, w_2 乘上(1-50%)，作为testing network里的参数，也就是 $w'_1, w'_2 = (1 - 50\%)(w_1, w_2) = 0.5w_1, 0.5w_2$

Testing of Dropout



这边想要呈现的是，在这个最简单的case里面，用不同的network structure做ensemble这件事情，跟我们用一整个network，并且把weight乘上一个值而不做ensemble所得到的output，其实是一样的

值得注意的是，只有是linear的network，才会得到上述的等价关系，如果network是非linear的，ensemble和dropout是不equivalent的

但是，dropout最后一个很神奇的地方是，虽然在non-linear的情况下，它是跟ensemble不相等的，但最后的结果还是会work

如果network很接近linear的话，dropout所得到的performance会比较好，而ReLU和Maxout的network相对来说是比较接近于linear的，所以我们通常会把含有ReLU或Maxout的network与Dropout配合起来使用

Why Deep Learning?

Shallow v.s. Deep

Deep is Better?

我们都知道deep learning在很多问题上的表现都是比较好的，越deep的network一般都会有更好的performance

那为什么会这样呢？有一种解释是：

- 一个network的层数越多，参数就越多，这个model就越复杂，它的bias就越小，而使用大量的data可以降低这个model的variance，performance当然就会更好

若随着layer层数从1到7，得到的error rate不断地降低，所以有人就认为，deep learning的表现这么好，完全就是用大量的data去硬train一个非常复杂的model而得到的结果

既然大量的data加上参数足够多的model就可以实现这个效果，那为什么一定要用DNN呢？我们完全可以用一层的shallow neural network来做同样的事情，理论上只要这一层里neuron的数目足够多，有足够的参数，就可以表示出任何函数；那DNN中deep的意义何在呢？

Fat + Short v.s. Thin + Tall

其实深和宽这两种结构的performance是会不一样的，这里我们就拿下面这两种结构的network做一下比较：

值得注意的是：如果要给Deep和Shallow的model一个公平的评比，你就要故意调整它们的形状，让它们的参数是一样多的，在这个情况下Shallow的model就会是一个矮胖的model，Deep的model就会是一个瘦高的model

在这个公平的评比之下，得到的结果如下图所示：

左侧表示的是deep network的情况，右侧表示的是shallow network的情况，为了保证两种情况下参数的数量是比较接近的，因此设置了右侧一层3772个neuron和一层4634个neuron这两种size大小，它们分别对应比较左侧每层两千个neuron共五层和每层两千个neuron共七层这两种情况下的network，此时它们的参数数目是接近的（注意参数数目和neuron的数目并不是等价的）

Layer X Size	Word Error Rate (%)	Layer X Size	Word Error Rate (%)
1 X 2k	24.2		
2 X 2k	20.4		
3 X 2k	18.4		
4 X 2k	17.8		
5 X 2k	17.2	1 X 3772	22.5
7 X 2k	17.1	1 X 4634	22.6
		1 X 16k	22.1

Why?

Seide, Frank, Gang Li, and Dong Yu. "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks." *Interspeech*. 2011.

这个时候你会发现，在参数数量接近的情况下，只有1层的network，它的error rate是远大于好几层的network的；这里甚至测试了一层16k个neuron大小的shallow network，把它跟左侧也是只有一层，但是没有那么宽的network进行比较，由于参数比较多所以才略有优势；但是把一层16k个neuron大小的shallow network和参数远比它少的2*2k大小的deep network进行比较，结果竟然是后者的表现更好

也就是说，只有1层的shallow network的performance甚至都比不过很多参数比它少但层数比它多的deep network，这是为什么呢？

有人觉得deep learning就是一个暴力碾压的方法，我可以弄一个很大很大的model，然后collect一大堆的数据，就可以得到比较好的performance

但根据上面的对比可知，deep learning显然是在结构上存在着某种优势，不然无法解释它会比参数数量相同的shallow learning表现得更好这个现象

Modularization

introduction

DNN结构一个很大的优势是， Modularization(模块化)，它用的是结构化的架构

就像写程序一样， shallow network实际上就是把所有的程序都写在了同一个main函数中，所以它去检测不同的class使用的方法是相互独立的；而deep network则是把整个任务分为了一个个小任务，每个小任务又可以不断细分下去，以形成modularization

在DNN的架构中，实际上每一层layer里的neuron都像是在解决同一个级别的任务，它们的output作为下一层layer处理更高级别任务的数据来源，低层layer里的neuron做的是对不同小特征的检测，高层layer里的neuron则根据需要挑选低层neuron所抽取出来的不同小特征，去检测一个范围更大的特征；neuron就像是一个个classifier，后面的classifier共享前面classifier的参数

这样做的好处是，低层的neuron输出的信息可以被高层不同的neuron重复使用，而并不需要像shallow network一样，每次在用到的时候都要重新去检测一遍，因此大大降低了程序的复杂度，做 modularization 的好处是，让我们的模型变简单了，我们是把本来的比较复杂的问题，变得比较简单。所以，当我们把问题变简单的时候，就算 training data 没有那么多，我们也可以把这个 task 做好

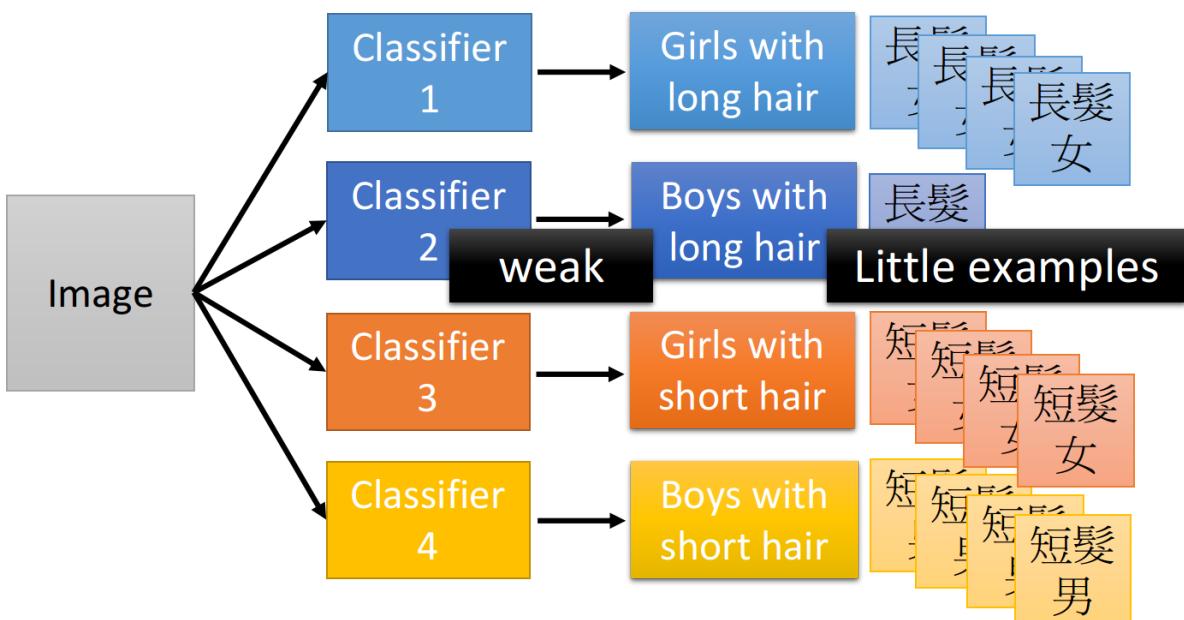
example

这里举一个分类的例子，我们要把input的人物分为四类：长头发女生、长头发男生、短头发女生、短头发男生

如果按照shallow network的想法，我们分别独立地train四个classifier(其实就相当于训练四个独立的 model)，然后就可以解决这个分类的问题；但是这里有一个问题，长头发男生的数据是比较少的，没有太多的training data，所以，你train出来的classifier就比较weak，去detect长头发男生的performance就比较差

Modularization

- Deep → Modularization



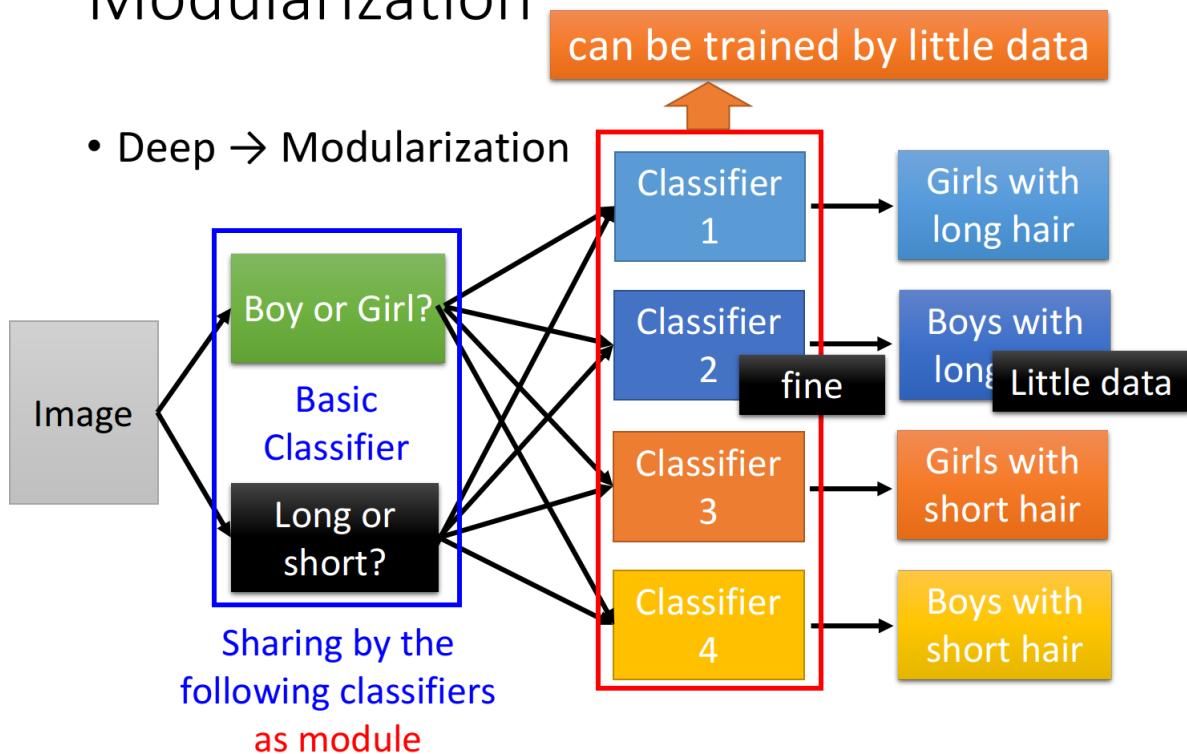
但其实我们的input并不是没有关联的，长头发的男生和长头发的女生都有一个共同的特征，就是长头发，因此如果我们分别独立地训练四个model作为分类器，实际上就是忽视了这个共同特征，也就是没有高效地用到data提供的全部信息，这恰恰是shallow network的弊端

而利用modularization的思想，使用deep network的架构，我们可以训练一个model作为分类器就可以完成所有的任务，我们可以把整个任务分为两个子任务：

- Classifier 1：检测是男生或女生
- Classifier 2：检测是长头发或短头发

虽然长头发的男生data很少，但长头发的人的data就很多，经过前面几层layer的特征抽取，就可以头发的数据全部都丢给Classifier 2，把男生或女生的数据全部都丢给Classifier 1，这样就真正做到了充分、高效地利用数据，Each basic classifier can have sufficient training examples，最终的Classifier再根据Classifier 1和Classifier 2提供的信息给出四类人的分类结果，

Modularization



你会发现，经过层层layer的任务分解，其实每一个Classifier要做的事情都是比较简单的，又因为这种分层的、模组化的方式充分利用了data，并提高了信息利用的效率，所以只要用比较少的training data就可以把结果train好

Deep → modularization

做modularization的好处是把原来比较复杂的问题变得简单，比如原来的任务是检测一个长头发的女生，但现在你的任务是检测长头发和检测性别，而当检测对象变简单的时候，就算training data没有那么多，我们也可以把这个task做好，并且所有的classifier都用同一组参数检测子特征，提高了参数使用效率，这就是modularization、这就是模块化的精神

由于deep learning的deep就是在做modularization这件事，所以它需要的training data反而是比较少的，这可能会跟你的认知相反，AI=big data+deep learning，但deep learning其实是为了解决less data的问题才提出的

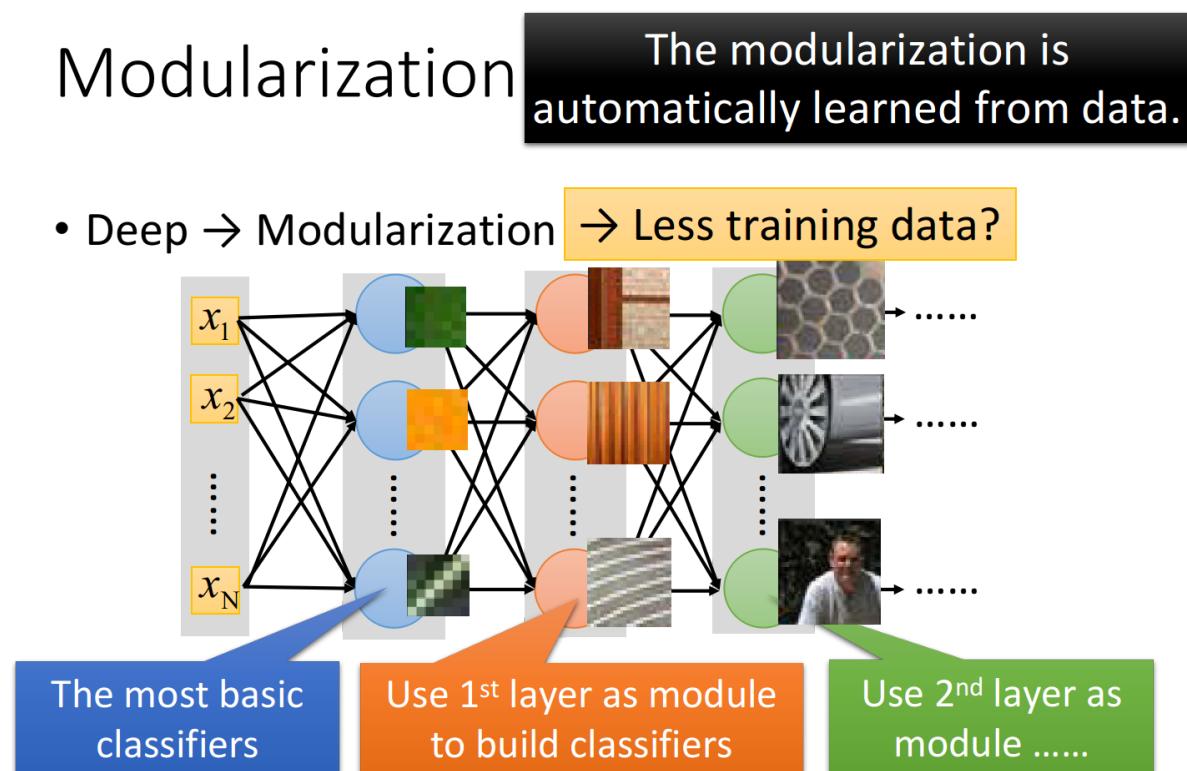
这边要强调的是，在做deep learning的时候，怎么做模块化这件事情是machine自动学到的，也就是说，第一层要检测什么特征、第二层要检测什么特征...这些都不是人为指定的，人只有定好有几层layer、每层layer有几个neuron，剩下的事情都是machine自己学到的

传统的机器学习算法，是人为地根据domain knowledge指定特征来进行提取，这种指定的提取方式，甚至是提取到的特征，也许并不是实际最优的，所以它的识别成功率并没有那么高；但是如果提取什么特征、怎么提取这件事让机器自己去学，它所提取的就会是那个最优解，因此识别成功率普遍会比人为指定要来的高

Modularization - Image

每一个neuron其实就是一个basic的classifier：

- 第一层neuron，它是一个最basic的classifier，检测的是颜色、线条这样的小特征
- 第二层neuron是比较复杂的classifier，它用第一层basic的classifier的output当作input，也就是把第一层的classifier当作module，利用第一层得到的小特征分类出不同样式的花纹
- 而第三层的neuron又把第二层的neuron当作它module，利用第二层得到的特征分类出蜂窝、轮胎、人
- 以此类推



Reference: Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014* (pp. 818–833)

Modularization - Speech

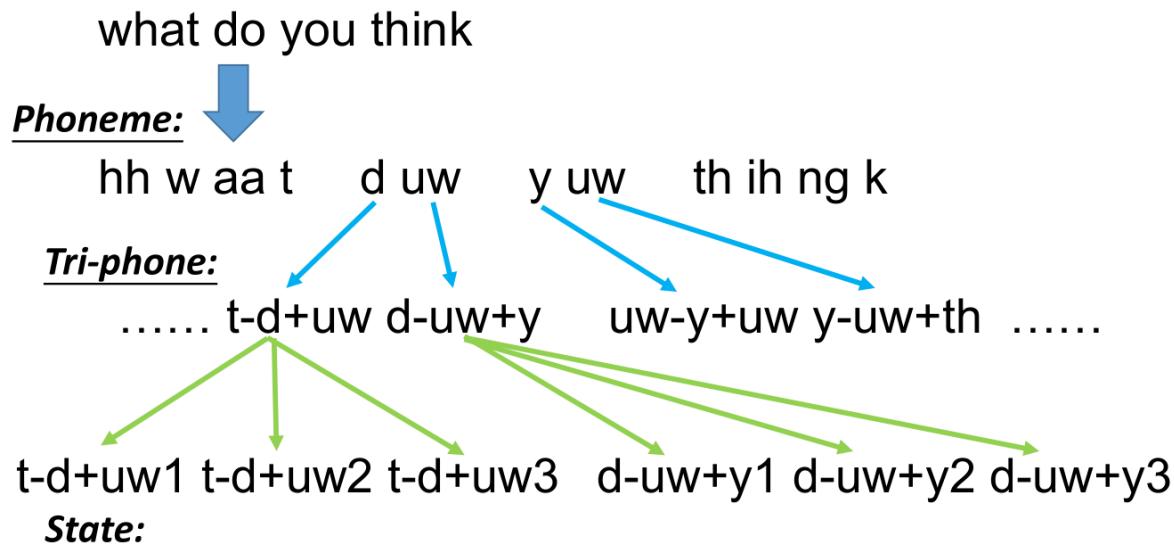
前面讲了deep learning的好处来自于modularization(模块化)，可以用比较efficient的方式来使用data和参数，这里以语音识别为例，介绍DNN的modularization在语音领域的应用

The hierarchical structure of human languages

当你说what do you think的时候，这句话其实是由一串phoneme所组成的，所谓phoneme，中文翻成音素，它是由语言学家制订的人类发音的基本单位，what由4个phoneme组成，do由两个phoneme组成，you由两个phoneme组成，等等

同样的phoneme可能会有不太一样的发音，当你发d uw和y uw的时候，心里想要发的都是uw，但由于人类发音器官的限制，你的phoneme发音会受到前后的phoneme所影响；所以，为了表达这一件事情，我们会给同样的phoneme不同的model，这个东西就叫做tri-phone

一个phoneme可以拆成几个state，我们通常就定成3个state



以上就是人类语言的基本构架

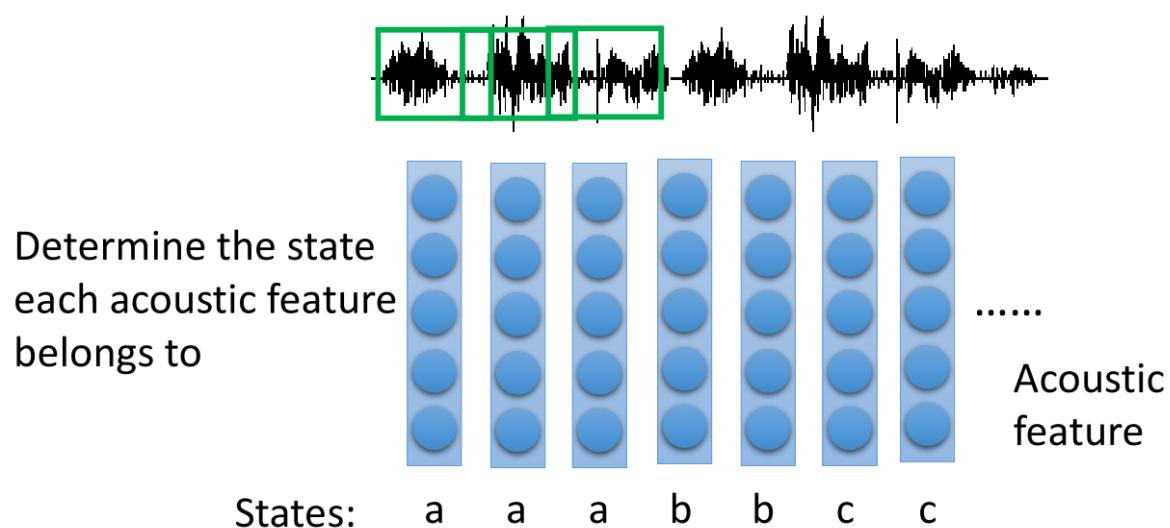
The first stage of speech recognition

语音辨识的过程其实非常复杂，这里只是讲语音辨识的第一步

你首先要做的事情是把acoustic feature(声学特征)转成state，这是一个单纯的classification的problem

大致过程就是在一串wave form(声音信号)上面取一个window(通常不会取太大，比如250个mini second大小)，然后用acoustic feature来描述这个window里面的特性，每隔一个时间段就取一个window，一段声音信号就会变成一串vector sequence，这个就叫做acoustic feature sequence

- Classification: input → acoustic feature, output → state



你要建一个Classifier去识别acoustic feature属于哪个state，再把state转成phoneme，然后把phoneme转成文字，接下来你还要考虑同音异字的问题...

这里不会详细讲述整个过程，而是想要比较一下过去在用deep learning之前和用deep learning之后，在语音辨识上的分类模型有什么差异

Classification

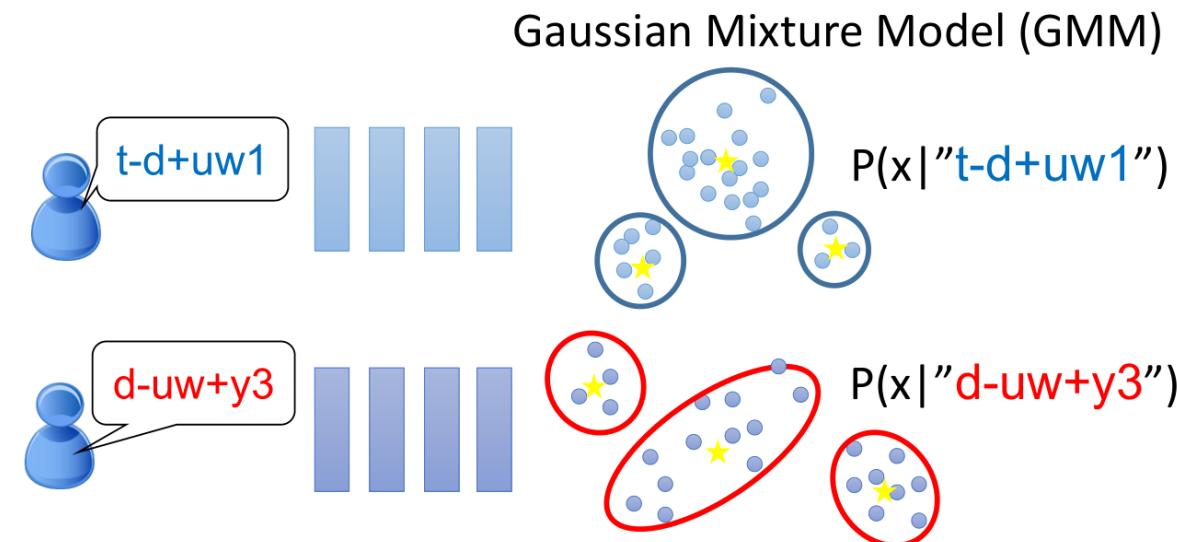
HMM-GMM

传统的方法叫做HMM-GMM

GMM，即Gaussian Mixture Model，它假设语音里的每一个state都是相互独立的（跟前面长头发的shallow例子很像，也是假设每种情况相互独立），因此属于每个state的acoustic feature都是stationary distribution（静态分布）的，因此我们可以针对每一个state都训练一个GMM model来识别

但这个方法其实不太现实，因为要列举的model数目太多了，一般语言中英文都有30几、将近40个phoneme，那这边就假设是30个，而在tri-phone里面，每一个phoneme随着context的不同又有变化，假设tri-phone的形式是a-b-c，那总共就有 $30 \times 30 \times 30 = 27000$ 个tri-phone，而每一个tri-phone又有三个state，每一个state都要用一个GMM来描述，那参数实在是太多了

- Each state has a stationary distribution for acoustic features



在有deep learning之前的传统处理方法是，让一些不同的state共享同样的model distribution，这件事情叫做Tied-state，实际操作上就把state当做pointer，不同的pointer可能会指向同样的distribution，所以有一些state的distribution是共享的，具体哪些state共享distribution则是由语言学等专业知识决定

那这样的处理方法太粗糙了，所以又有人提出了subspace GMM，它里面其实就有modularization、有模块化的影子

它的想法是，我们先找一个Gaussian pool（里面包含了很多不同的Gaussian distribution），每一个state的information就是一个key，它告诉我们这个state要从Gaussian pool里面挑选哪些Gaussian出来

比如有某一个state 1，它挑第一、第三、第五个Gaussian；另一个state 2，它挑第一、第四、第六个Gaussian；如果你这样做，这些state有些时候就可以share部分的Gaussian，有些时候又可以完全不share Gaussian，至于要share多少Gaussian，这都是可以从training data中学出来的

HMM-GMM的方法，默认把所有的phone或者state都看做是无关联的，对它们分别训练independent model，这其实是inefficient的，它没有充分利用data提供的信息

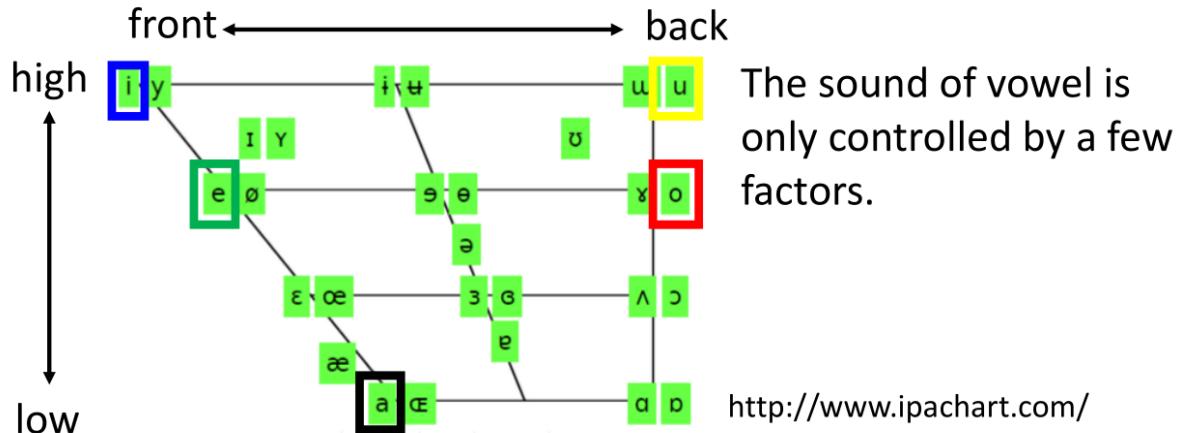
对人类的声音来说，不同的phoneme都是由人类的发音器官所generate出来的，它们并不是完全无关的，下图画出了人类语言里面所有的元音，这些元音的发音其实就只受到三件事情的影响：

- 舌头的前后位置
- 舌头的上下位置
- 嘴型

比如图中所标英文的5个元音a, e, i, o, u, 当你发a到e到i的时候, 舌头是由下往上; 而i跟u, 则是舌头放在前面或放在后面的差别; 在图中同一个位置的元音, 它们舌头的位置是一样的, 只是嘴型不一样

- In HMM-GMM, all the phonemes are modeled independently

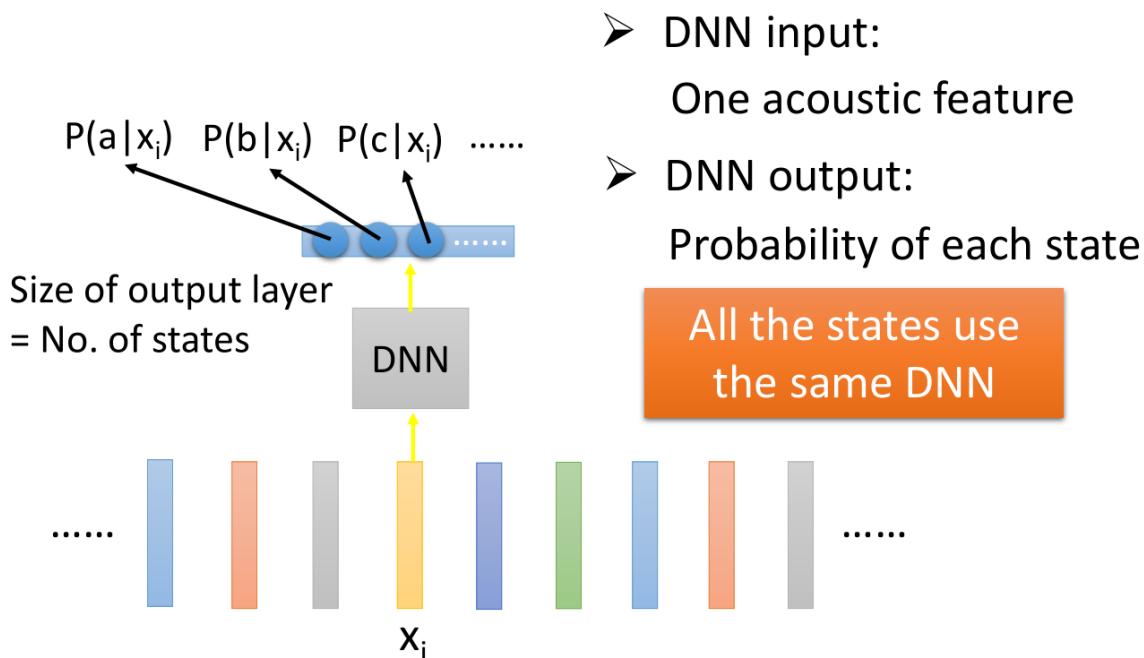
- Not an effective way to model human voice



DNN

如果采用deep learning的做法, 就是去learn一个deep neural network, 这个deep neural network的input是一个acoustic feature, 它的output就是该feature属于某个state的概率, 这就是一个简单的classification problem

那这边最关键的一点是, 所有的state识别任务都是用同一个DNN来完成的; 值得注意的是DNN并不是因为参数多取胜的, 实际上在HMM-GMM里用到的参数数量和DNN其实是差不多的, 区别只是GMM用了很多很小的model, 而DNN则用了一个很大的model



DNN把所有的state通通用同一个model来做分类, 会是一种比较有效率的做法, 解释如下

我们拿一个hidden layer出来, 然后把这个layer里所有neuron的output降维到2维得到下图, 每个点的颜色对应着input a, e, i, o, u, 神奇的事情发生了: 降维图上这5个元音的分布跟右上角元音位置图的分布几乎是一样的

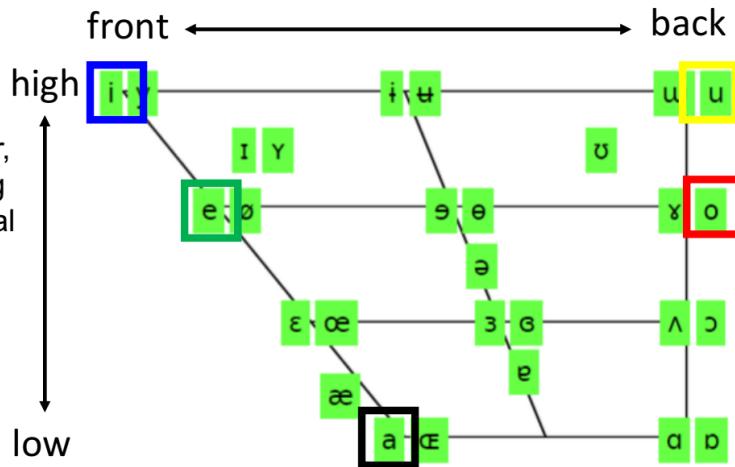
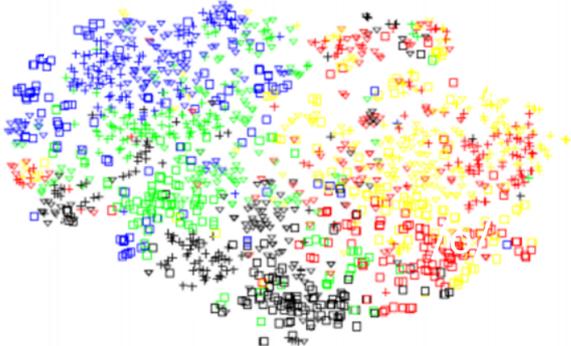
因此，DNN并不是马上就去检测发音是属于哪一个phone或哪一个state，比较lower的layer会先观察人是用什么样的方式在发这个声音，人的舌头位置应该在哪里，是高是低，是前是后；接下来的layer再根据这个结果，去决定现在的发音是属于哪一个state或哪一个phone

这些lower的layer是一个人类发音方式的detector，而所有phone的检测都share这同一组detector的结果，因此最终的这些classifier是share了同一组用来detect发音方式的参数，这就做到了模块化，同一个参数被更多的地方share，因此显得更有效率

Modularization

Vu, Ngoc Thang, Jochen Weiner, and Tanja Schultz. "Investigating the Learning Effect of Multilingual Bottle-Neck Features for ASR." *Interspeech*. 2014.

Output of hidden layer reduce to two dimensions



- The lower layers detect the manner of articulation
- All the phonemes share the results from the same set of detectors.
- Use parameters effectively

Result

这个时候就可以来回答Why Deep中提到的问题了

Universality Theorem告诉我们任何的continuous的function都可以用一层足够宽的neural network来实现，在90年代，这是很多人放弃做deep learning的一个原因

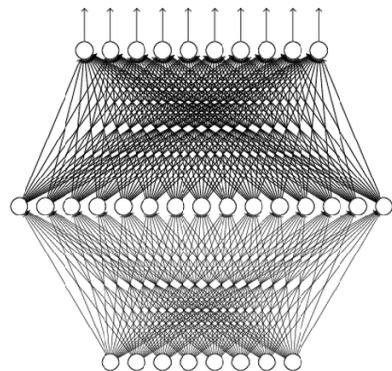
但是这个理论只告诉了我们可能性，却没有说明这件事的效率问题；根据上面的几个例子我们已经知道，只用一个hidden layer来描述function其实是没有效率的；当你用multi-layer，用hierarchy structure来描述function的时候，才会是比较有效率的

Universality Theorem

Any continuous function f

$$f : R^N \rightarrow R^M$$

Can be realized by a network
with one hidden layer
(given **enough** hidden neurons)



Reference for the reason:
<http://neuralnetworksanddeeplearning.com/chap4.html>

Yes, shallow network can represent any function.

However, using deep structure is more effective.

Analogy

下面用逻辑电路和剪窗花的例子来更形象地描述Deep和shallow的区别

Logic Circuit

逻辑电路其实可以拿来类比神经网络

- Logic circuits consists of **gates**; Neural network consists of **neurons**
- A two layers of logic gates can represent any Boolean function; 有一个hidden layer的 network(input layer+hidden layer共两层)可以表示任何continuous function
 - 逻辑门只要根据input的0、1状态和对应的output分别建立起门电路关系即可建立两级电路
- 实际设计电路的时候，为了节约成本，会进行多级优化，建立起hierarchy架构，如果某一个结构的逻辑门组合被频繁用到的话，其实在优化电路里，这个组合是可以被多个门电路共享的，这样用比较少的逻辑门就可以完成一个电路；在deep neural network里，践行modularization的思想，许多neuron作为子特征检测器被多个classifier所共享，本质上就是参数共享，就可以用比较少的参数就完成同样的function

比较少的参数意味着不容易overfitting，用比较少的数据就可以完成同样任务

剪窗花

我们之前讲过这个逻辑回归的分类问题，可能会出现下面这种linear model根本就没有办法分类的问题，而当你加了hidden layer的时候，就相当于做了一个feature transformation，把原来的 x_1, x_2 转换到另外一个平面，变成 x'_1, x'_2

你会发现，在例子中通过这个hidden layer的转换，其实就好像把原来这个平面按照对角线对折了一样，对折后两个蓝色的点就重合在了一起，这个过程跟剪窗花很像：

- 我们在做剪窗花的时候，每次把色纸对折，就相当于把原先的这个多维空间对折了一次来提高维度

- 如果你在某个地方戳一个洞，再把色纸打开，你折了几折，在对应的这些地方就都会有一个洞；那你在高维空间上的某一个点，就相当于展开后空间上的许多点，由于可以对这个空间做各种各样的对折和剪裁，所以二维平面上无论多少复杂的分类情况，经过多次折叠，不同class最后都可以在一个高维空间上以比较明显的方式被分隔开来

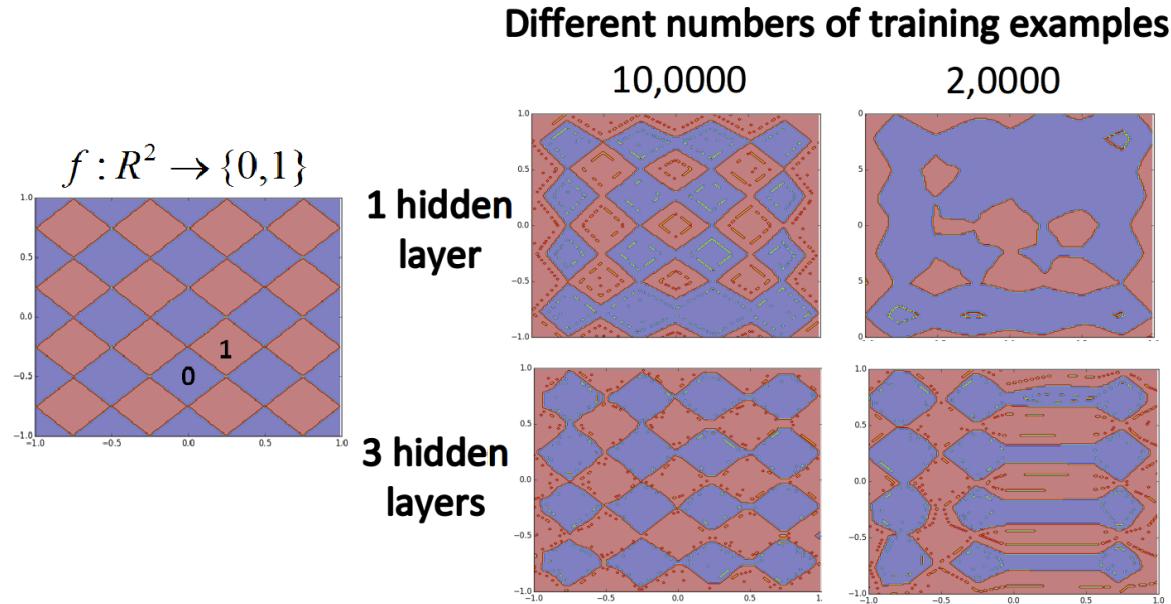
这样做既可以解决某些情况下难以分类的问题，又能够以比较有效率的方式充分利用data（比如高维空间上的1个点等于二维空间上的5个点，相当于1笔data发挥出5笔data的作用），deep learning是更有效率的利用data

下面举了一个小例子：

左边的图是training data，右边则是1层hidden layer与3层hidden layer的不同network的情况对比，这里已经控制它们的参数数量趋于相同，试验结果是，当training data为10万笔的时候，两个network学到的样子是比较接近原图的，而如果只给2万笔training data，1层hidden layer的情况就完全崩掉了，而3层hidden layer的情况会比较好一些，它其实可以被看作是剪窗花的时候一不小心剪坏了，然后展开得到的结果

关于如何得到model学到的图形，可以用固定model的参数，然后对input进行梯度下降，最终得到结果

More Analogy - Experiment



End-to-end Learning

Introduction

所谓的End-to-end learning，指的是只给model input和output，而不告诉它中间每一个function要怎么分工，让它自己去学会知道在生产线的每一站，自己应该要做什么事情；在DNN里，就是叠一个很深的neural network，每一层layer就是生产线上的一站

Speech Recognition

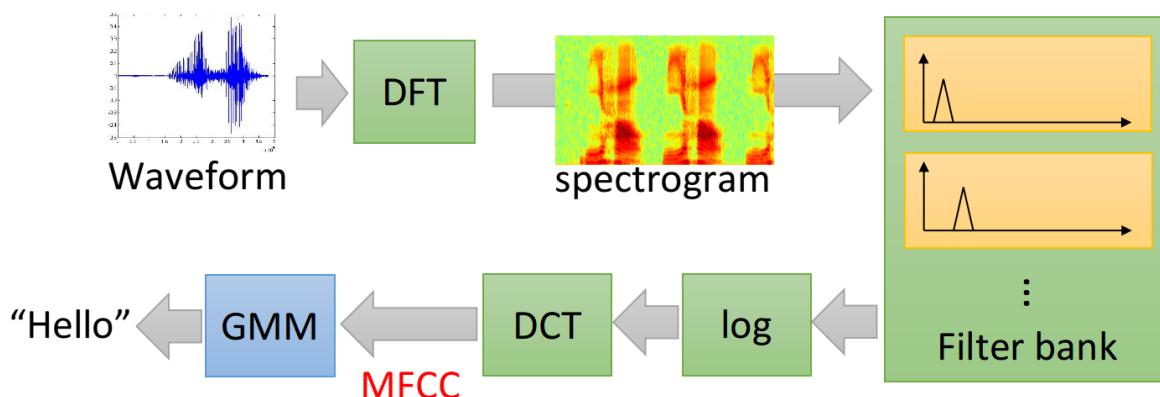
End-to-end Learning在语音识别上体现的非常明显

在传统的Speech Recognition里，只有最后GMM这个蓝色的block，才是由training data学出来的，前面绿色的生产线部分都是由过去的古圣先贤手动制订出来的，其实制订的这些function非常非常的强，可以说是增一分则太肥，减一分则太瘦这样子，以至于在这个阶段卡了将近20年

后来有了deep learning，我们就可以用neural network把DCT离散余弦变换取代掉，甚至你从 spectrogram开始都拿deep neural network取代掉，也可以得到更好的结果，如果你分析DNN的 weight，它其实可以自动学到要做filter bank这件事情（filter bank是模拟人类的听觉器官所制定出来的 filter）

End-to-end Learning - Speech Recognition

- Shallow Approach



Each box is a simple function in the production line:

 :hand-crafted :learned from data

那能不能够叠一个很深很深的neural network，input直接就是time domain上的声音信号，而output直接就是文字，中间完全不要做Fourier transform之类？

目前的结果是，它学到的极限也只是做到与做了Fourier transform的结果打平而已。Fourier transform很强，但是已经是信号处理的极限了，machine做的事情就很像是在做Fourier transform，但是只能做到一样好，没有办法做到更好

有关End-to-end Learning在Image Recognition的应用和Speech Recognition很像，这里不再赘述

Complex Task

那deep learning还有什么好处呢？

有时候我们会遇到非常复杂的task：

- 有时候非常像的input，它会有很不一样的output

比如在做图像辨识的时候，下图这个白色的狗跟北极熊其实看起来是很像的，但是你的machine要有能力知道，看到左边这张图要output狗，看到右边这张图要output北极熊

- 有时候看起来很不一样的input，output其实是一样的

比如下面这两个方向上看到的火车，横看成岭侧成峰，尽管看到的很不一样，但是你的machine要有能力知道这两个都是同一种东西

- Very similar input, different output



- Very different input, similar output



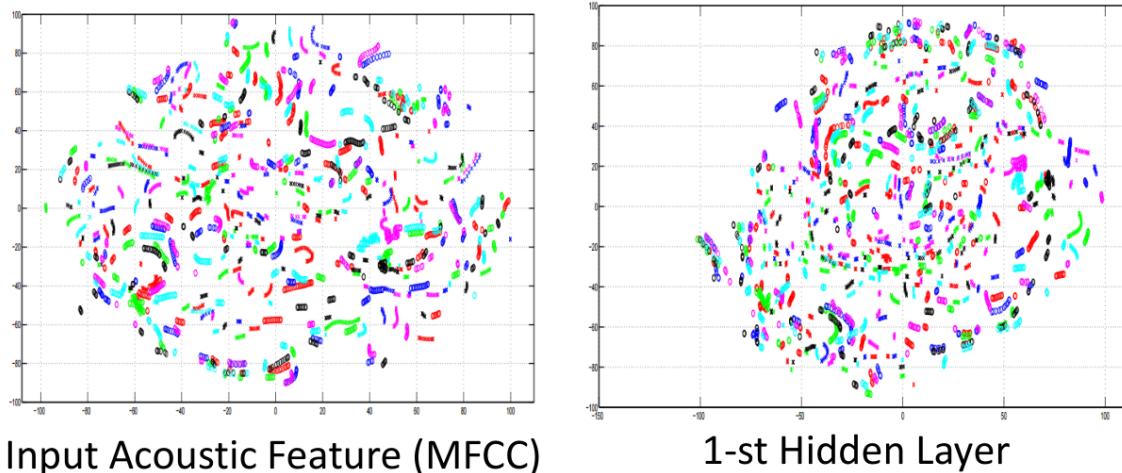
如果你的network只有一层的话，就只能做简单的transform，没有办法把一样的东西变得很不一样，把不一样的东西变得很像；如果要实现这些，就需要做很多层次的转换

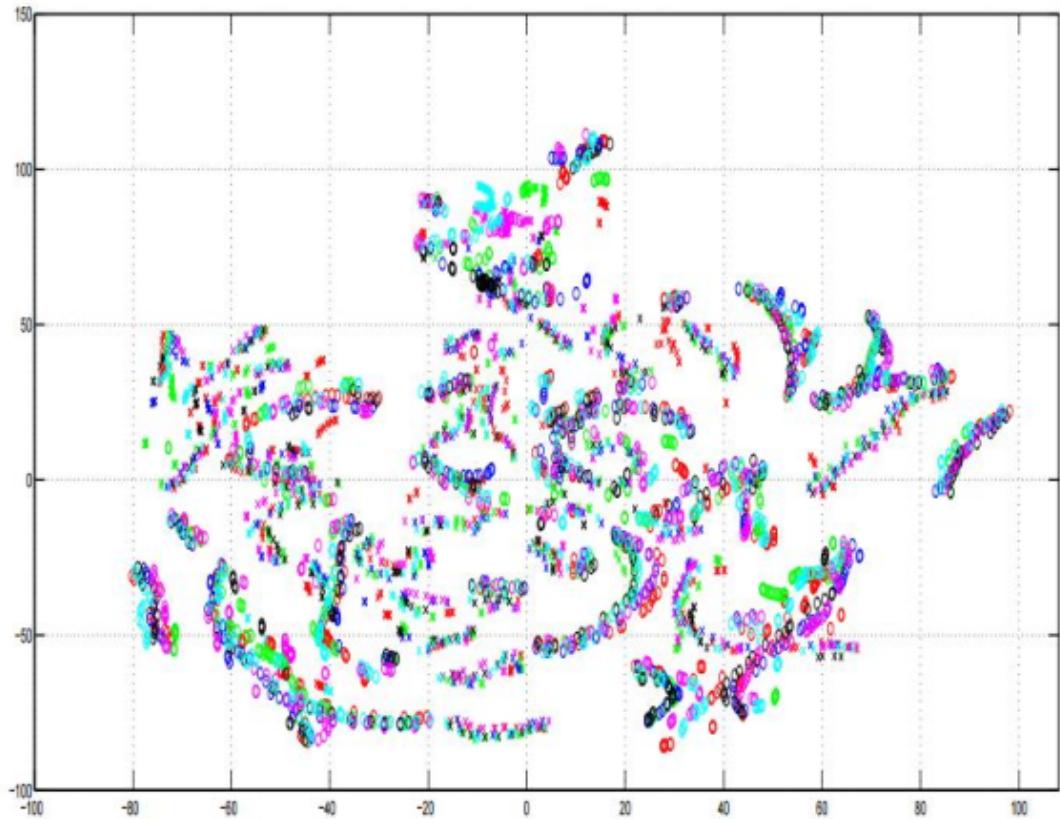
以语音识别为例，把MFCC投影到二维平面，不同颜色代表不同人说的同一句话，第一个隐藏层输出还是很不一样，第八个隐藏层输出，不同人说的同样的句子，变得很像，经过很多的隐藏层转换后，就把他们map在一起了。

Complex Task ...

A. Mohamed, G. Hinton, and G. Penn, "Understanding how Deep Belief Networks Perform Acoustic Modelling," in ICASSP, 2012.

- Speech recognition: Speaker normalization is automatically done in DNN





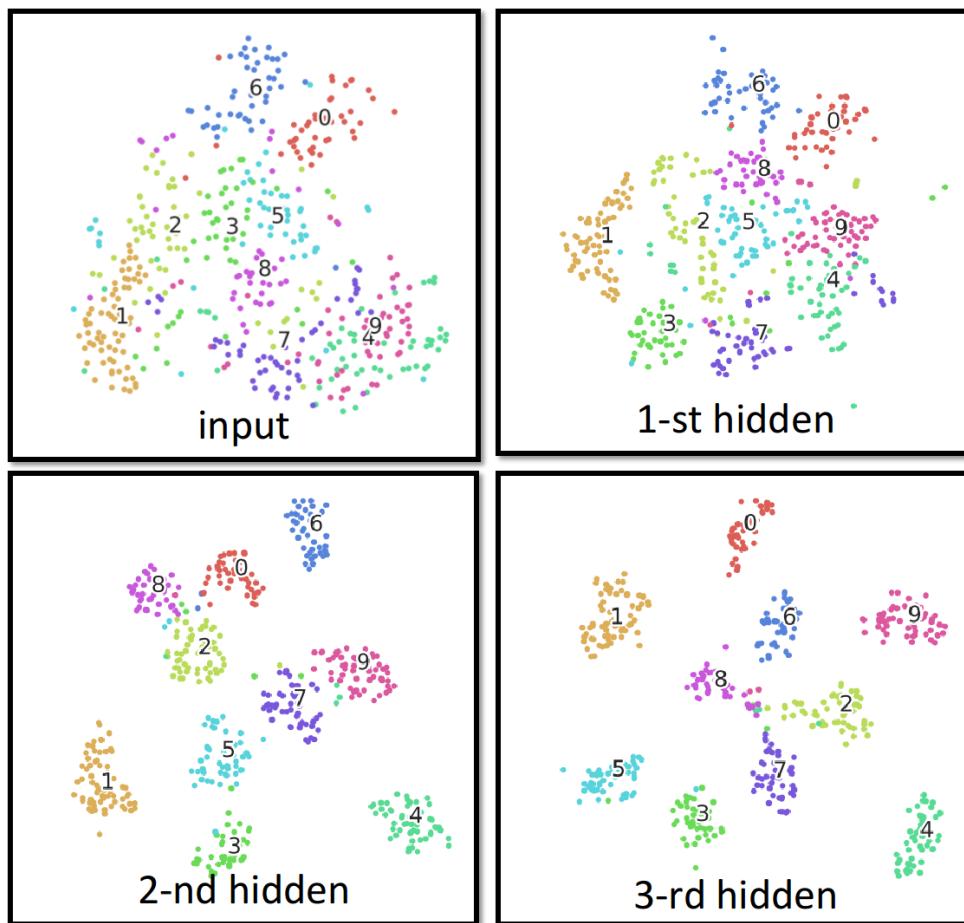
8-th Hidden Layer

这里以MNIST手写数字识别为例，展示一下DNN中，在高维空间上对这些Complex Task的处理能力

如果把 28×28 个pixel组成的vector投影到二维平面上就像左上角所示，你会发现4跟9的pixel几乎是叠在一起的，因为4跟9很像，都是一个圈圈再加一条线，所以如果你光看input的pixel的话，4跟9几乎是叠在一起的，你几乎没有办法把它分开

但是，等到第二个、第三个layer的output，你会发现4、7、9逐渐就被分开了，所以使用deep learning的deep，这也是其中一个理由

MNIST



Conclusion

- 考虑input之间的内在关联，所有的class用同一个model来做分类
- modularization思想，复杂问题简单化，把检测复杂特征的大任务分割成检测简单特征的小任务
- 所有的classifier使用同一组参数的子特征检测器，共享检测到的子特征
- 不同的classifier会share部分的参数和data，效率高
- 联系logic circuit和剪纸的例子
- 多层hidden layer对complex问题的处理上比较有优势

To learn more ...

Do Deep Nets Really Need To Be Deep? (by Rich Caruana)

<http://research.microsoft.com/apps/video/default.aspx?id=232373&r=1>

Deep Learning: Theoretical Motivations (Yoshua Bengio)

http://videolectures.net/deeplearning2015_bengio_theoretical_motivations/

Connections between physics and deep learning

<https://www.youtube.com/watch?v=5MdSE-N0bxS>

Why Deep Learning Works: Perspectives from Theoretical Chemistry

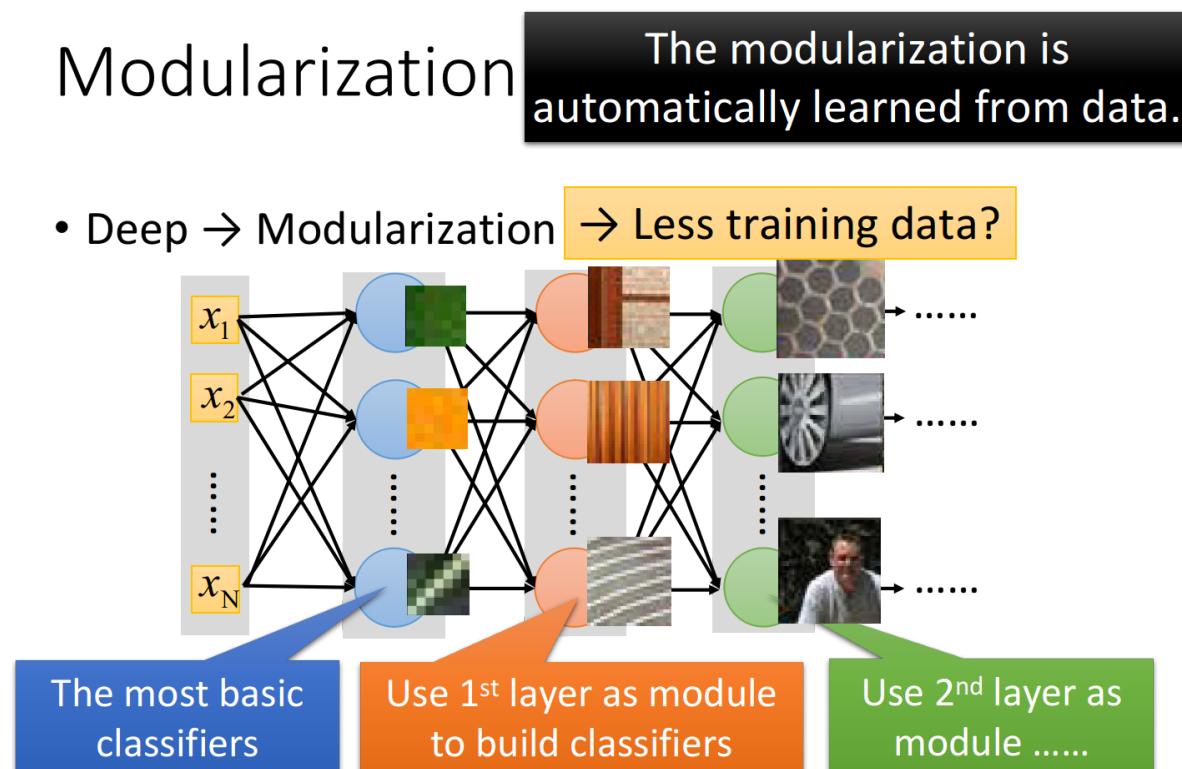
<https://www.youtube.com/watch?v=kIbKHIPbxiU>

Convolutional Neural Network

CNN v.s. DNN

我们当然可以用一般的neural network来做影像处理，不一定要用CNN，比如说，你想要做图像的分类，那你就去train一个neural network，它的input是一张图片，你就用里面的pixel来表示这张图片，也就是一个很长很长的vector，而output则是由图像类别组成的vector，假设你有1000个类别，那output就有1000个dimension

但是，我们现在会遇到的问题是这样子：实际上，在train neural network的时候，我们会有一种期待说，在这个network structure里面的每一个neuron，都应该代表了一个最基本的classifier；事实上，在文献上，根据训练的结果，也有很多人得到这样的结论，举例来说，下图中：



Reference: Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014* (pp. 818–833)

- 第一个layer的neuron，它就是最简单的classifier，它做的事情就是detect有没有绿色出现、有没有黄色出现、有没有斜的条纹出现等等
- 那第二个layer，它做的事情是detect更复杂的东西，根据第一个layer的output，它如果看到直线横线，就是窗框的一部分；如果看到棕色的直条纹就是木纹；看到斜条纹加灰色的，这个有可能是很多东西，比如说，轮胎的一部分等等
- 再根据第二个hidden layer的output，第三个hidden layer会做更复杂的事情，比如它可以知道说，当某一个neuron看到蜂巢，它就会被activate；当某一个neuron看到车子，它就会被activate；当某一个neuron看到人的上半身，它就会被activate等等

那现在的问题是这样子：当我们直接用一般的fully connected的feedforward network来做图像处理的时候，往往需要太多的参数

举例来说，假设这是一张100*100的彩色图片，它的分辨率才100*100，那这已经是很小张的image了，然后你需要把它拉成一个vector，总共有100*100*3个pixel（如果是彩色的图的话，每个pixel其实需要3个value，即RGB值来描述它的），把这些加起来input vector就已经有三万维了；如果input vector是三万维，又假设hidden layer有1000个neuron，那仅仅是第一层hidden layer的参数就已经有30000*1000个了，这样就太多了

所以，CNN做的事情其实是，来简化这个neural network的架构，我们根据自己的知识和对图像处理的理解，一开始就把某些实际上用不到的参数给过滤掉

我们一开始就想一些办法，不要用fully connected network，而是用比较少的参数，来做图像处理这件事情，所以CNN其实是比一般的DNN还要更简单的

虽然CNN看起来，它的运作比较复杂，但事实上，它的模型比DNN还要更简单，我们就是用prior knowledge，去把原来fully connected的layer里面的一些参数拿掉，就变成CNN

Why CNN for Image?

为什么我们有可能把一些参数拿掉？为什么我们有可能只用比较少的参数就可以来做图像处理这件事情？下面列出三个对影像处理的观察，这也是CNN架构提出的基础所在

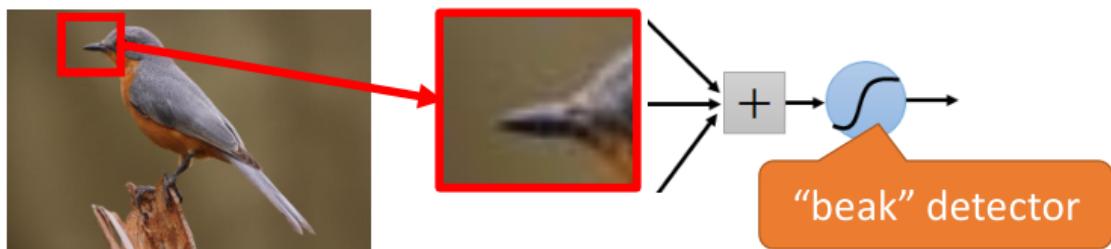
Some patterns are much smaller than the whole image

在影像处理里面，如果在network的第一层hidden layer里，那些neuron要做的事情是侦测有没有一种东西、一种pattern（图案样式）出现，那大部分的pattern其实是比整张image要小的，所以对一个neuron来说，想要侦测有没有某一个pattern出现，它其实并不需要看整张image，只需要看这张image的一小部分，就可以决定这件事情了

举例来说，假设现在我们有一张鸟的图片，那第一层hidden layer的某一个neuron的工作是，检测有没有鸟嘴的存在（你可能还有一些neuron侦测有没有鸟嘴的存在、有一些neuron侦测有没有爪子的存在、有一些neuron侦测有没有翅膀的存在、有没有尾巴的存在，之后合起来，就可以侦测，图片中有没有一只鸟），那它其实并不需要看整张图，因为，其实我们只要给neuron看个小的区域，它其实就可以知道说，这不是一个鸟嘴，对人来说也是一样，只要看这个小的区域你就会知道说这是鸟嘴，所以，**每一个neuron其实只要连接到一个小块的区域就好，它不需要连接到整张完整的图，因此也对应着更少的参数**

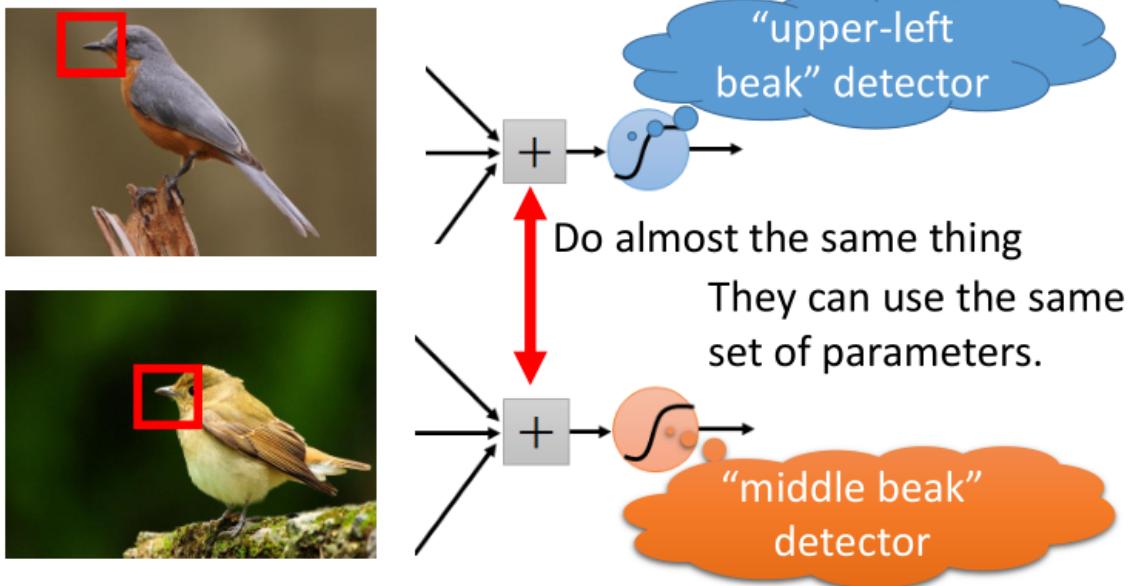
A neuron does not have to see the whole image to discover the pattern.

Connecting to small region with less parameters



The same patterns appear in different regions

同样的pattern，可能会出现在image的不同部分，但是它们有同样的形状、代表的是同样的含义，因此它们也可以用同样的neuron、同样的参数，被同一个detector检测出来



举例来说，图中分别有一个处于左上角的鸟嘴和一个处于中央的鸟嘴，但你并不需要训练两个不同的detector去专门侦测左上角有没有鸟嘴和中央有没有鸟嘴这两件事情，这样做太冗余了，我们要cost down(降低成本)，我们并不需要有两个neuron、两组不同的参数来做duplicate的事情，所以我们可以要求这些功能几乎一致的neuron共用一组参数，它们share同一组参数就可以帮助减少总参数的量

Subsampling the pixels will not change the object

我们可以对一张image做subsampling，假如你把它奇数行、偶数列的pixel拿掉，image就可以变成原来的十分之一大小，而且并不会影响人对这张image的理解，对你来说，下面两张大小不一的image看起来不会有什么太大的区别，你都可以识别里面有什么物件，因此subsampling对图像辨识来说，可能是没有太大的影响的

所以，我们可以利用subsampling这个概念把image变小，从而减少需要用到的参数量

The whole CNN structure

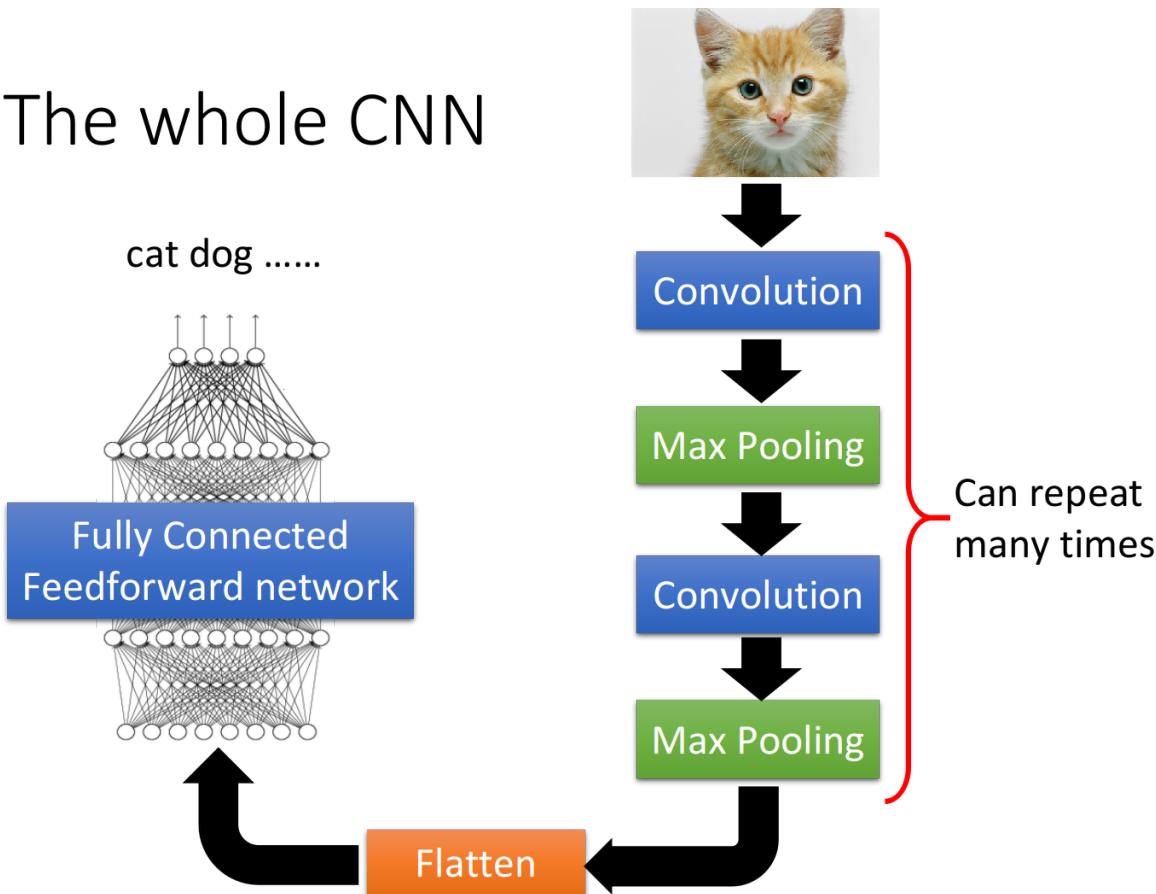
整个CNN的架构是这样的：

首先，input一张image以后，它会先通过Convolution的layer，接下来做Max Pooling这件事，然后再去做Convolution，再做Max Pooling...

这个process可以反复进行多次（重复次数需要事先决定），这就是network的架构，就好像network有几层一样，你要做几次convolution，做几次Max Pooling，在定这个network的架构时就要事先决定好

当你做完先前决定的convolution和max pooling的次数后，你要做的事情是Flatten，做完flatten以后，你就把Flatten output丢到一般的Fully connected network里面去，最终得到影像辨识的结果

The whole CNN



我们基于之前提到的三个对影像处理的观察，设计了CNN这样的架构，第一个是要侦测一个pattern，你不需要看整张image，只要看image的一个小部分；第二个是同样的pattern会出现在一张图片的不同区域；第三个是我们可以对整张image做subsampling

前面两个property，是用convolution的layer来处理的；最后这个property，是用max pooling来处理的

Convolution

假设现在我们network的input是一张 6×6 的image，图像是黑白的，因此每个pixel只需要用一个value来表示，而在convolution layer里面，有一堆Filter，这边的每一个Filter，其实就等同于是在Fully connected layer里的一个neuron

Property 1

每一个Filter其实就是一个matrix，这个matrix里面每一个element的值，就跟那些neuron的weight和bias一样，是network的parameter，它们具体的值都是通过Training data学出来的，而不是人去设计的

所以，每个Filter里面的值是什么，要做什么事情，都是自动学习出来的，图中每一个filter是 3×3 的size，意味着它就是在侦测一个 3×3 的pattern，当它侦测的时候，并不会去看整张image，它只看一个 3×3 范围内的pixel，就可以判断某一个pattern有没有出现，这就考虑了property 1

Property 2

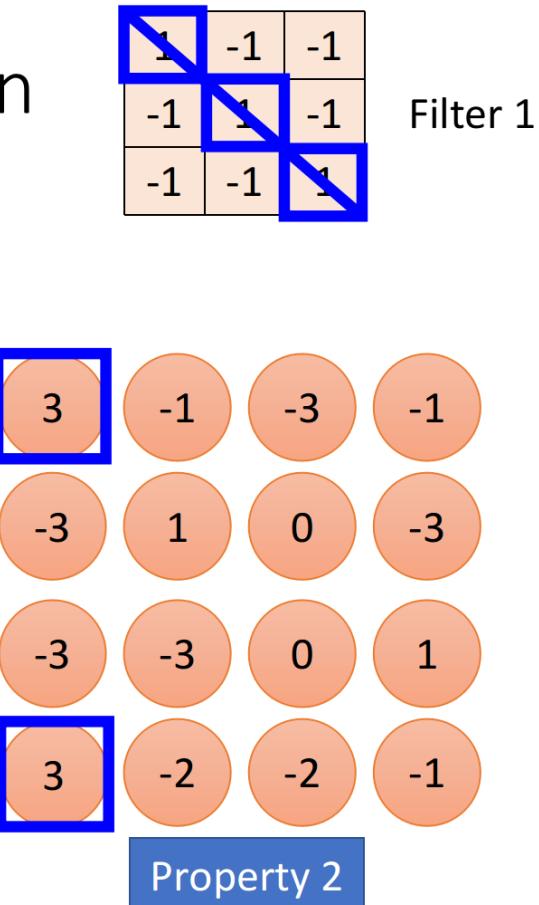
这个filter是从image的左上角开始，做一个slide window，每次向右挪动一定的距离，这个距离就叫做stride，由你自己设定，每次filter停下的时候就跟image中对应的 3×3 的matrix做一个内积(相同位置的值相乘并累计求和)，这里假设stride=1，那么我们的filter每次移动一格，当它碰到image最右边的时候，就从下一行的最左边开始重复进行上述操作，经过一整个convolution的过程，最终得到下图所示的红色的 4×4 matrix

CNN – Convolution

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image



观察上图中的Filter 1，它斜对角的地方是1,1,1，所以它的工作就是detect有没有连续的从左上角到右下角的1,1,1出现在这个image里面，检测到的结果已在上图中用蓝线标识出来，此时filter得到的卷积结果的左上和左下得到了最大的值，这就代表说，该filter所要侦测的pattern出现在image的左上角和左下角

同一个pattern出现在image左上角的位置和左下角的位置，并不需要用到不同的filter，我们用filter 1就可以侦测出来，这就考虑了property 2

Feature Map

在一个convolution的layer里面，它会有一打filter，不一样的filter会有不一样的参数，但是这些filter做卷积的过程都是一模一样的，你把filter 2跟image做完convolution以后，你就会得到另外一个蓝色的4*4 matrix，那这个蓝色的4*4 matrix跟之前红色的4*4 matrix合起来，他们就叫做**Feature Map**，有多少个filter，对应就有多少个映射后的image，filter的数量等于feature map的数量

CNN – Convolution

-1	1	-1
-1	1	-1
-1	1	-1

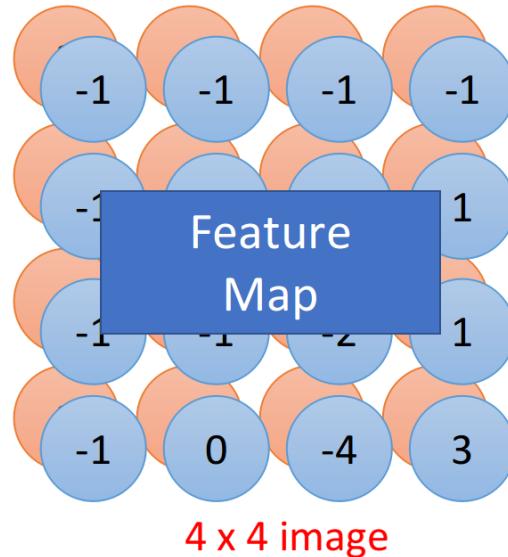
Filter 2

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Do the same process for every filter



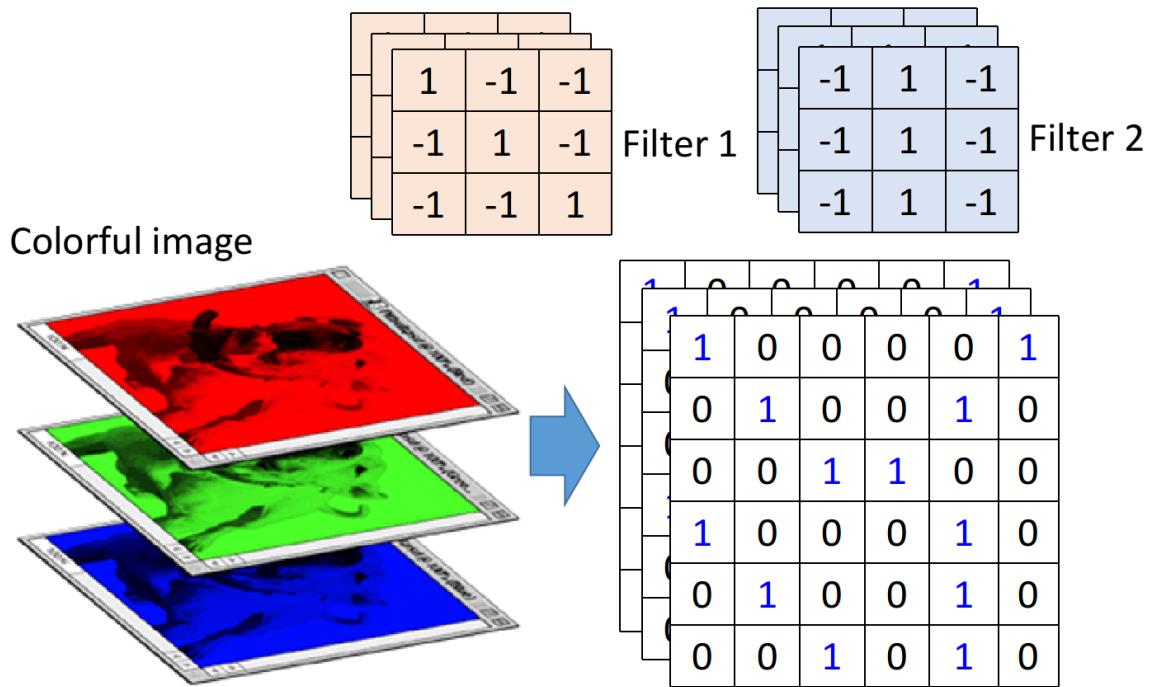
CNN对不同scale的相同pattern的处理上存在一定的困难，由于现在每一个filter size都是一样的，这意味着，如果你今天有同一个pattern，它有不同的size，有大的鸟嘴，也有小的鸟嘴，CNN并不能够自动处理这个问题；

DeepMind曾经发过一篇paper，提到了当你input一张image的时候，它在CNN前面，再接另外一个network，这个network做的事情是，它会output一些scalar，告诉你说，它要把这个image的里面的哪些位置做旋转、缩放，然后，再丢到CNN里面，这样你其实会得到比较好的performance

Colorful image

刚才举的例子是黑白的image，所以你input的是一个matrix，如果今天是彩色的image会怎么样呢？我们知道彩色的image就是由RGB组成的，所以一个彩色的image，它就是好几个matrix叠在一起，是一个立方体，如果我今天要处理彩色的image，要怎么做呢？

CNN – Colorful image



这个时候你的filter就不再是一个matrix了，它也会是一个立方体，如果你今天是RGB这三个颜色来表示一个pixel的话，那你的input就是 $3 \times 6 \times 6$ ，你的filter就是 $3 \times 3 \times 3$ ，你的filter的高就是3，在做convolution的话，就是将filter的9个值和image的9个值做内积，不是把每一个channel分开来算，而是合在一起来算，一个filter就考虑了不同颜色所代表的channel，具体操作为做内积，并且三层的结果相加，得到一个scalar，因此一个filter可以得到一个feature map，并且层数只能为1层

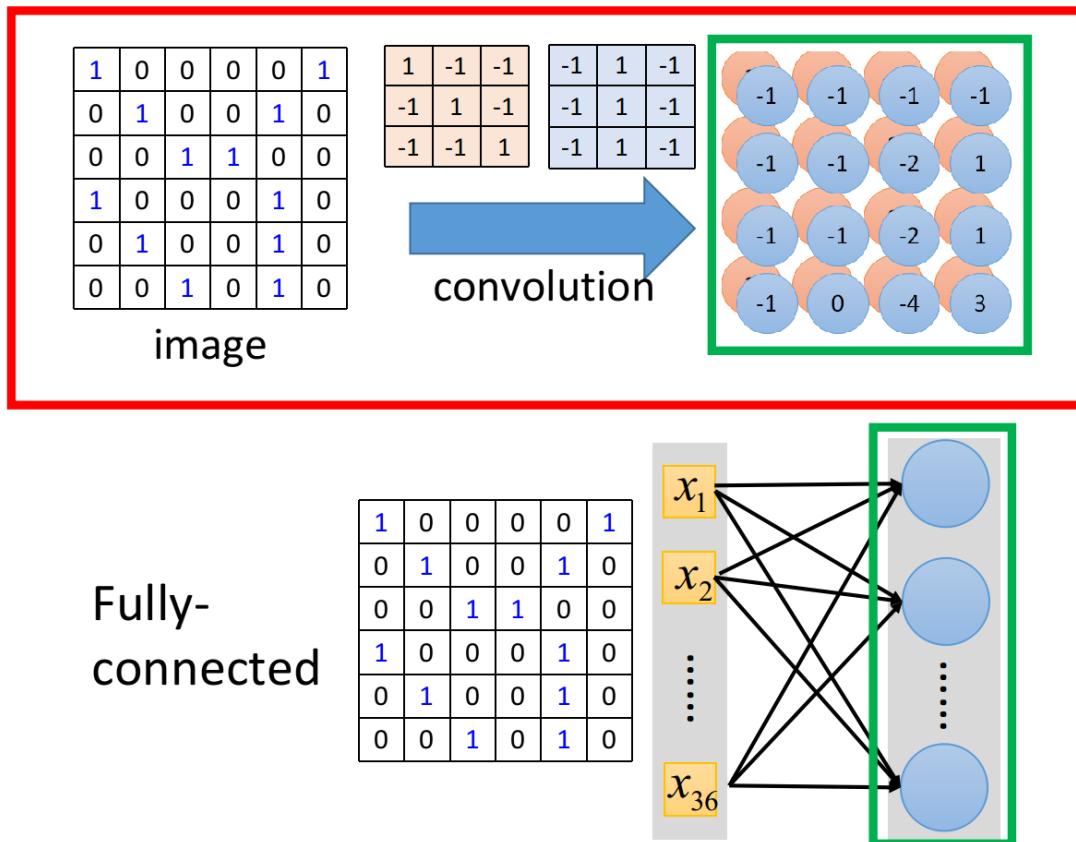
图中的这种情况，输出的feature map有2个channel，分别是filter 1和filter 2与原图卷积得到的矩阵。

Convolution v.s. Fully connected

接下来要讲的是，convolution跟fully connected有什么关系，你可能觉得说，它是一个很特别的operation，感觉跟neural network没半毛钱关系，其实，它就是一个neural network

convolution这件事情，其实就是fully connected的layer把一些weight拿掉而已，下图中绿色方框标识出的feature map的output，其实就是hidden layer的neuron的output

Convolution v.s. Fully Connected



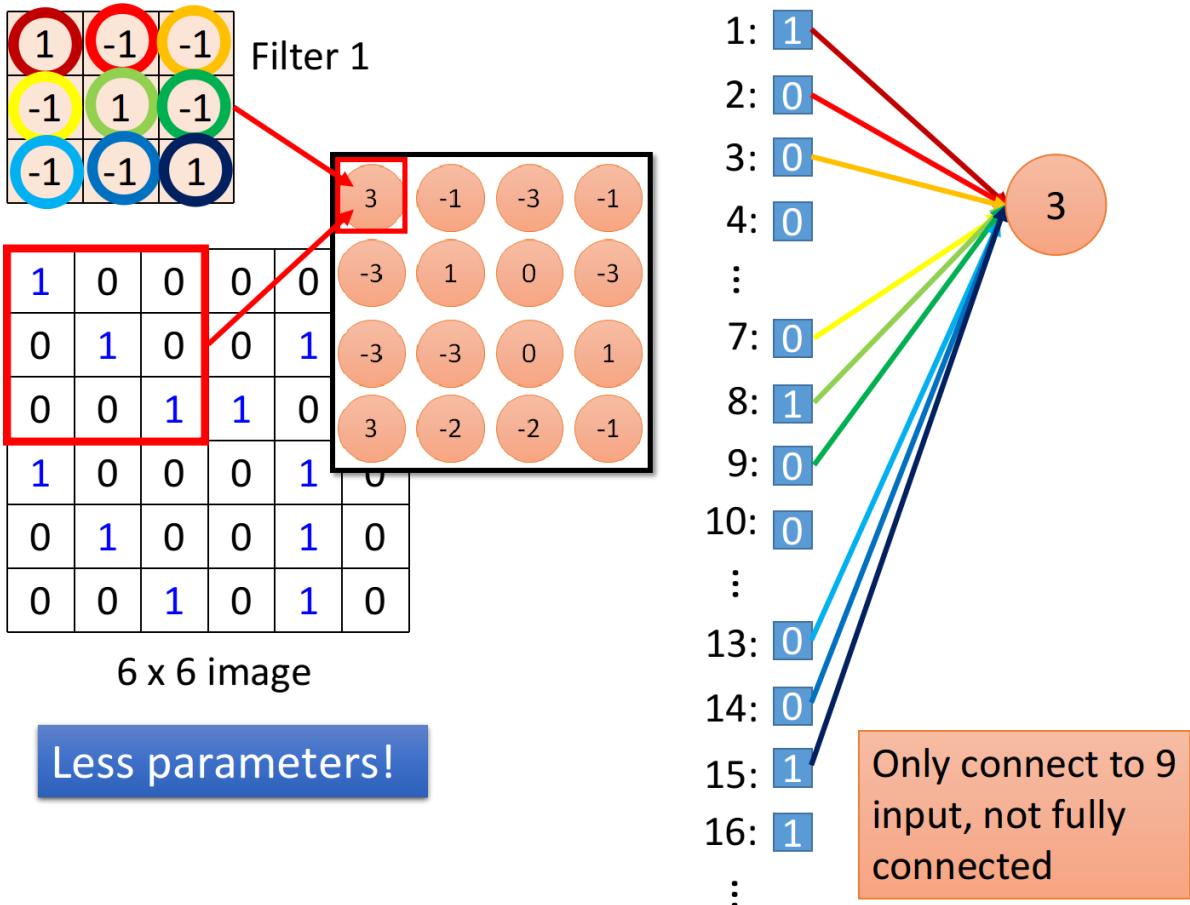
接下来我们来解释这件事情：

如下图所示，我们在做convolution的时候，把filter放在image的左上角，然后再去做inner product，得到一个值3；这件事情等同于，我们现在把这个image的 6×6 的matrix拉直变成右边这个用于input的vector，然后，你有一个neuron，这些input经过这个neuron之后，得到的output是3

那这个neuron的output怎么来的呢？这个neuron实际上就是由filter转化而来的，我们把filter放在image的左上角，此时filter考虑的就是和它重合的9个pixel，假设你把这一个 6×6 的image的36个pixel拉成直的vector作为input，那这9个pixel分别就对应着右侧编号1, 2, 3的pixel，编号7, 8, 9的pixel跟编号13, 14, 15的pixel

如果我们说这个filter和image matrix做inner product以后得到的output 3，就是input vector经过某个neuron得到的output 3的话，这就代表说存在这样一个neuron，这个neuron带weight的连线，就只连接到编号为1, 2, 3, 7, 8, 9, 13, 14, 15的这9个pixel而已，而这个neuron和这9个pixel连线上所标注的weight就是filter matrix里面的这9个数值

作为对比，Fully connected的neuron是必须连接到所有36个input上的，但是，我们现在只用连接9个input，因为我们知道要detect一个pattern，不需要看整张image，看9个input pixel就够了，所以当我们这么做的时候，就用了比较少的参数



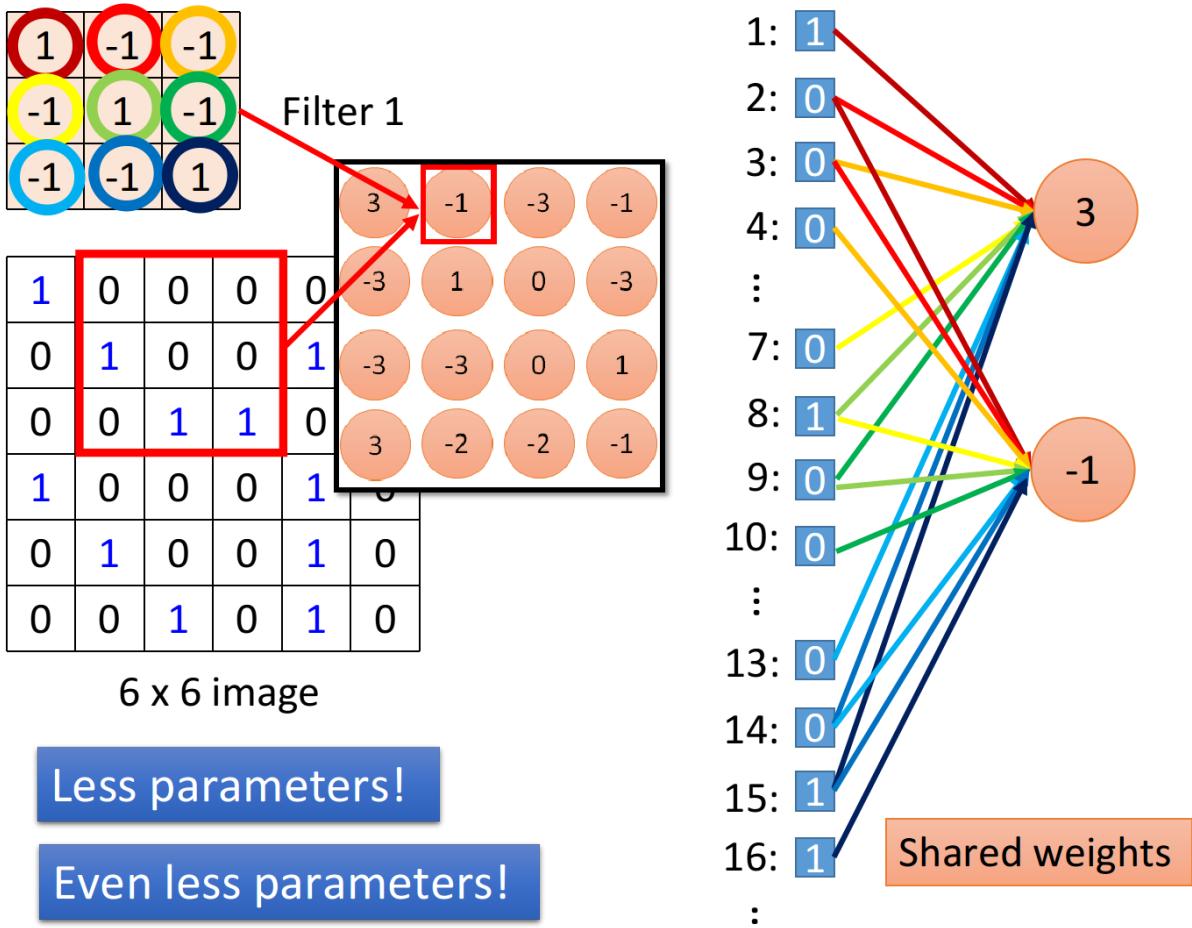
当我们把filter做stride = 1的移动的时候，会发生什么事呢？此时我们通过filter和image matrix的内积得到另外一个output值-1，我们假设这个-1是另外一个neuron的output，那这个neuron会连接到哪些input呢？下图中这个框起来的地方正好就对应到pixel 2, 3, 4, pixel 8, 9, 10跟pixel 14, 15, 16

你会发现output为3和-1的这两个neuron，它们分别去检测在image的两个不同位置上是否存在某个pattern，因此在Fully connected layer里它们做的是两件不同的事情，每一个neuron应该都有自己独立的weight

但是，当我们做这个convolution的时候，首先我们把每一个neuron前面连接的weight减少了，然后我们强迫某些neuron（比如图中output为3和-1的两个neuron），它们一定要共享一组weight

虽然这两个neuron连接到的pixel对象各不相同，但它们用的weight都必须是一样的，等于filter里面的元素值

这件事情就叫做weight share，当我们做这件事情的时候，用的参数，又会比原来更少



因此我们可以这样想，有这样一些特殊的neuron，它们只连接着9条带weight的线（ $9=3 \times 3$ 对应着filter的元素个数，这些weight也就是filter内部的元素值，上图中圆圈的颜色与连线的颜色一一对应）

当filter在image matrix上移动做convolution的时候，每次移动做的事情实际上是去检测这个地方有没有某一种pattern，对于Fully connected layer来说，它是对整张image做detection的，因此每次去检测image上不同地方有没有pattern其实是不同的事情，所以这些neuron都必须连接到整张image的所有pixel上，并且不同neuron的连线上的weight都是相互独立的

对于convolution layer来说，首先它是对image的一部分做detection的，因此它的neuron只需要连接到image的部分pixel上，对应连线所需要的weight参数就会减少；

其次由于是用同一个filter去检测不同位置的pattern，所以这对convolution layer来说，其实是同一件事情，因此不同的neuron，虽然连接到的pixel对象各不相同，但是在“做同一件事情”的前提下，也就是用同一个filter的前提下，这些neuron所使用的weight参数都是相同的，通过这样一种weight share的方式，再次减少network所需要用到的weight参数

CNN的本质，就是减少参数的过程

Training

看到这里你可能会问，这样的network该怎么搭建，又该怎么去train呢？

首先，第一件事情就是这都是用toolkit做的，所以你大概不会自己去写；如果你要自己写的话，它其实跟原来的Backpropagation用一模一样的做法，只是有一些weight永远是0，你就不去train它，它就永远是0

然后，怎么让某些neuron的weight值永远都是一样呢？你就用一般的Backpropagation的方法，对每个weight都去算出gradient，再把本来要tight在一起、要share weight的那些weight的gradient平均，然后，让他们update同样值就ok了

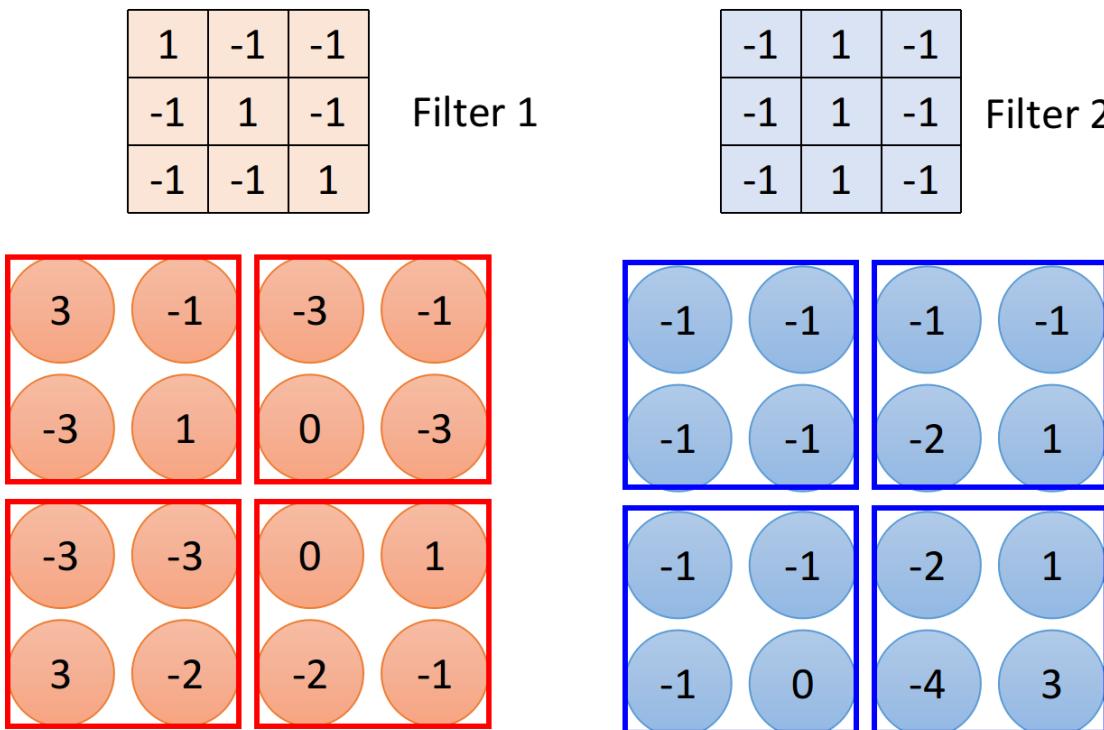
Max Pooling

Operation of max pooling

相较于convolution, max pooling是比较简单的, 它就是做subsampling, 根据filter 1, 我们得到一个4*4的matrix, 根据filter 2, 你得到另外一个4*4的matrix, 接下来, 我们要做什么事呢?

我们把output四个分为一组, 每一组里面通过选取平均值或最大值的方式, 把原来4个value合成一个value, 这件事情相当于在image每相邻的四块区域内都挑出一块来检测, 这种subsampling的方式就可以让你的image缩小!

CNN – Max Pooling

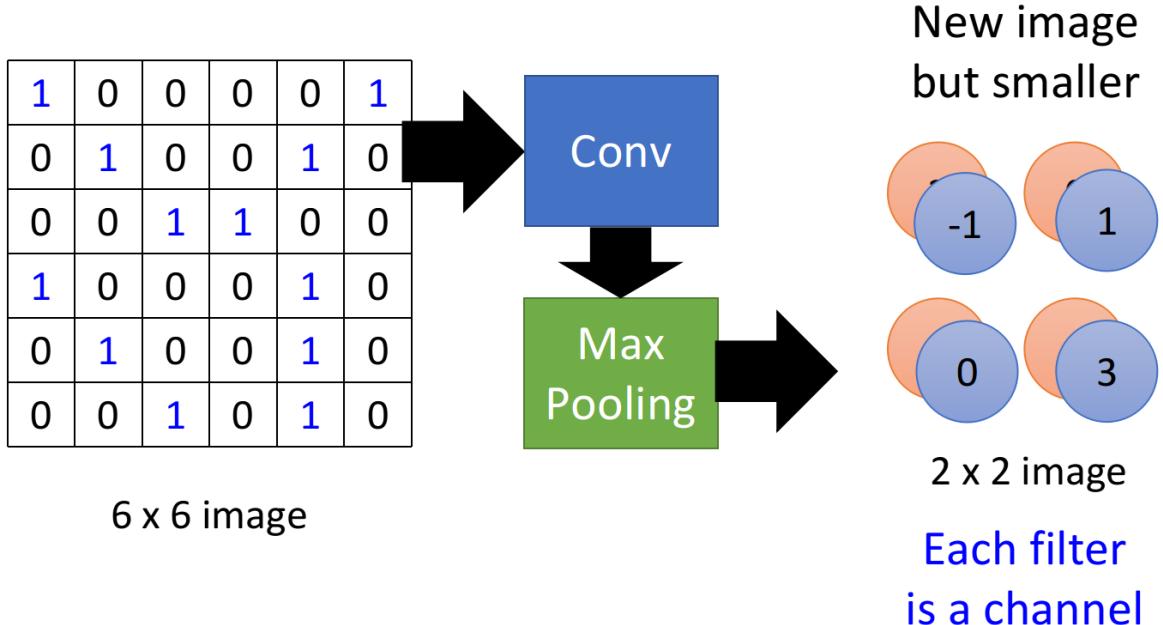


讲到这里你可能会有一个问题, 如果取Maximum放到network里面, 不就没法微分了吗? max这个东西, 感觉是没有办法对它微分的啊, 其实是可以的, 类比Maxout network, 你就知道怎么用微分的方式来处理它

The whole CNN

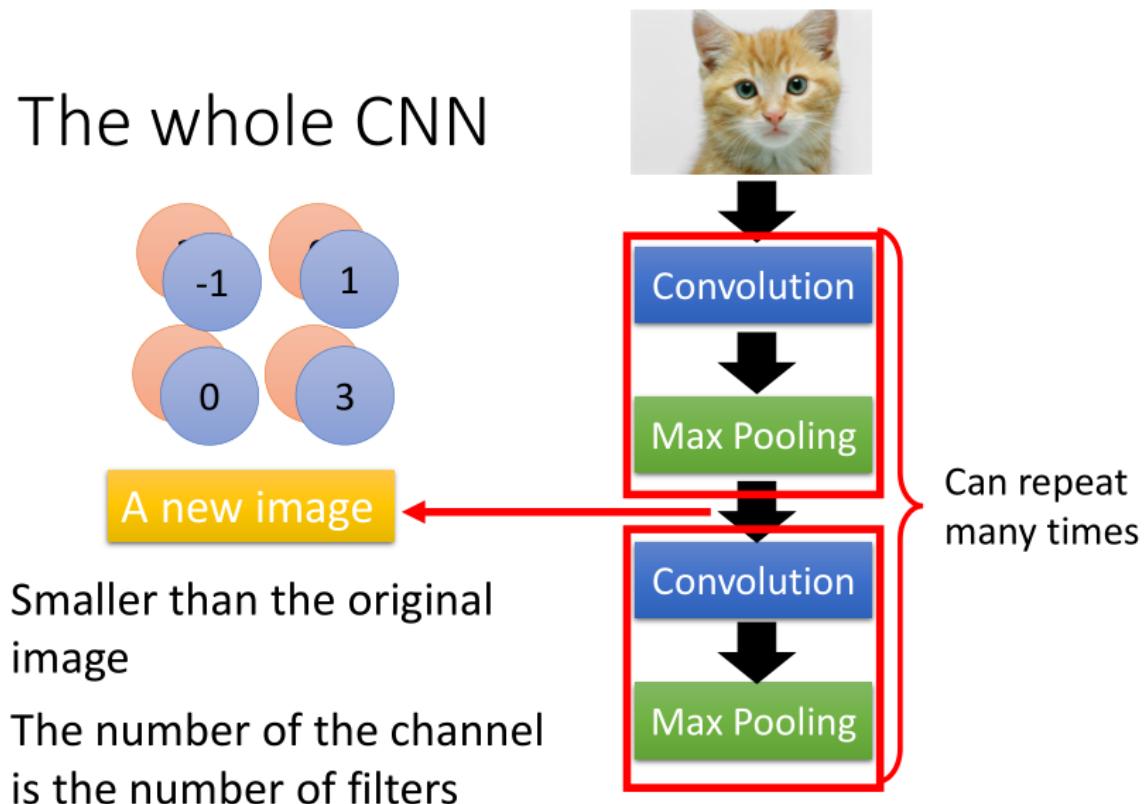
做完一次convolution加一次max pooling, 我们就把原来6*6的image, 变成了一个2*2的image; 至于这个2*2的image, 它每一个pixel的深度, 也就是每一个pixel用几个value来表示, 就取决于你有几个filter, 如果你有50个filter, 就是50维, 像下图中是两个filter, 对应的深度就是二维, 得到结果就是一个new smaller image, 一个filter就代表了一个channel。

CNN – Max Pooling



所以，这是一个新的比较小的image，它表示的是不同区域上提取到的特征，实际上不同的filter检测的是该image同一区域上的不同特征属性，所以每一层channel代表的是一种属性，一块区域有几种不同的属性，就有几层不同的channel，对应的就会有几个不同的filter对其进行convolution操作，**Each filter is a channel**

The whole CNN

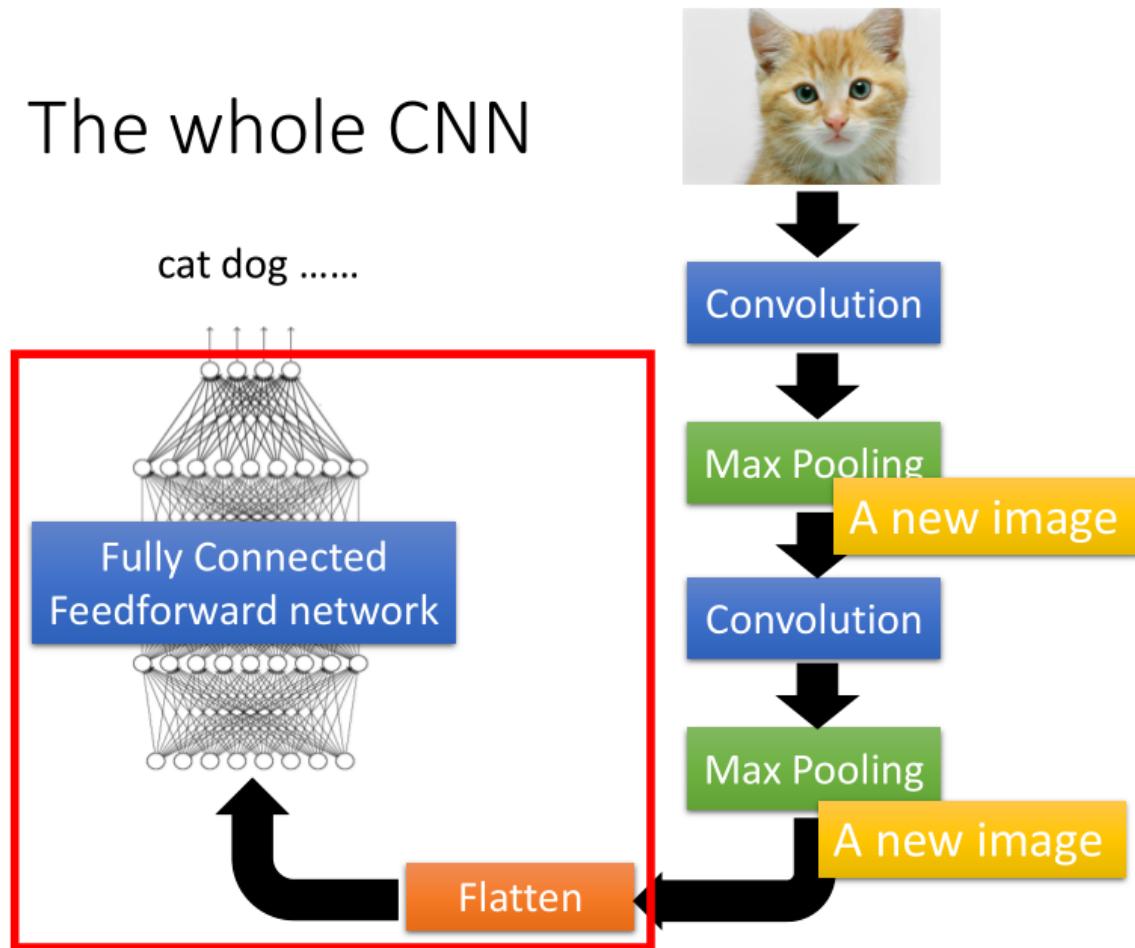


这件事情可以repeat很多次，你可以把得到的这个比较小的image，再次进行convolution和max pooling的操作，得到一个更小的image，依次类推

有这样一个问题：假设我第一个convolution有25个filter，通过这些filter得到25个feature map，然后repeat的时候第二个convolution也有25个filter，那这样做完，我是不是会得到 25^2 个feature map？

其实不是这样的，你这边做完一次convolution，得到25个feature map之后再做一次convolution，还是会得到25个feature map，因为convolution在考虑input的时候，是会考虑深度的，它并不是每一个channel分开考虑，而是一次考虑所有的channel，所以，你convolution这边有多少个filter，再次output的时候就会有多少个channel，**The number of the channel is the number of filters**，只不过下一次convolution时，25个filter都是一个立方体，它的高有25个value那么高

The whole CNN



这件事可以repeat很多次，通过一个convolution + max pooling就得到新的image。它是一个比较小的image，可以把这个小的image，做同样的事情，再次通过convolution + max pooling，将得到一个更小的image。

filter

- 假设我们input是一个 $1*28*28$ 的image
- 通过25个filter的convolution layer以后你得到的output，会有25个channel，又因为filter的size是 $3*3$ ，因此如果不考虑image边缘处的处理的话，得到的channel会是 $26*26$ 的，因此通过第一个convolution得到 $25*26*26$ 的cubic image
- 接下来就是做Max pooling，把 $2*2$ 的pixel分为一组，然后从里面选一个最大的组成新的image，大小为 $25*13*13$
- 再做一次convolution，假设这次选择50个filter，每个filter size是 $3*3$ 的话，output的channel就变成有50个，那 $13*13$ 的image，通过 $3*3$ 的filter，就会变成 $11*11$ ，因此通过第二个convolution得到 $50*11*11$ 的image
- 再做一次Max Pooling，变成 $50*5*5$

在第一个convolution里面，每一个filter都有9个参数，它就是一个 3×3 的matrix；但是在第二个convolution layer里面，虽然每一个filter都是 3×3 ，但它其实不是 3×3 个参数，因为它的input是一个 $25 \times 13 \times 13$ 的cubic，这个cubic的channel有25个，所以要用同样高度的cubic filter对它进行卷积，于是我们的filter实际上是一个 $25 \times 3 \times 3$ 的cubic，所以第二个convolution layer这边每个filter共有225个参数

通过两次convolution和max pooling的组合，最终的image变成了 $50 \times 5 \times 5$ 的size，然后使用Flatten将这个image拉直，变成一个1250维的vector，再把它丢到一个Fully Connected Feedforward network里面，network structure就搭建完成了

看到这里，你可能会有一个疑惑，第二次convolution的input是 $25 \times 13 \times 13$ 的cubic，用50个 3×3 的filter卷积后，得到的输出时应该是50个cubic，且每个cubic的尺寸为 $25 \times 11 \times 11$ ，那么max pooling把长宽各砍掉一半后就是 $50 \times 5 \times 5$ 的cubic，那flatten后不应该就是 $50 \times 25 \times 5 \times 5$ 吗？

其实不是这样的，在第二次做convolution的时候，我们是用 $25 \times 3 \times 3$ 的cubic filter对 $25 \times 13 \times 13$ 的cubic input进行卷积操作的，filter的每一层和input cubic中对应的每一层（也就是每一个channel），它们进行内积后，还要把cubic的25个channel的内积值进行求和，作为这个“neuron”的output，它是一个scalar，这个cubic filter对整个cubic input做完一遍卷积操作后，得到的是一层scalar，然后有50个cubic filter，对应着50层scalar，因此最终得到的output是一个 $50 \times 11 \times 11$ 的cubic

这里的关键是filter和image都是cubic，每个cubic filter有25层高，它和同样有25层高的cubic image做卷积，并不是单单把每个cubic对应的channel进行内积，还会把这些内积求和，最终变为1层

因此两个矩阵或者tensor做了卷积后，不管之前的维数如何，都会变为一个scalar

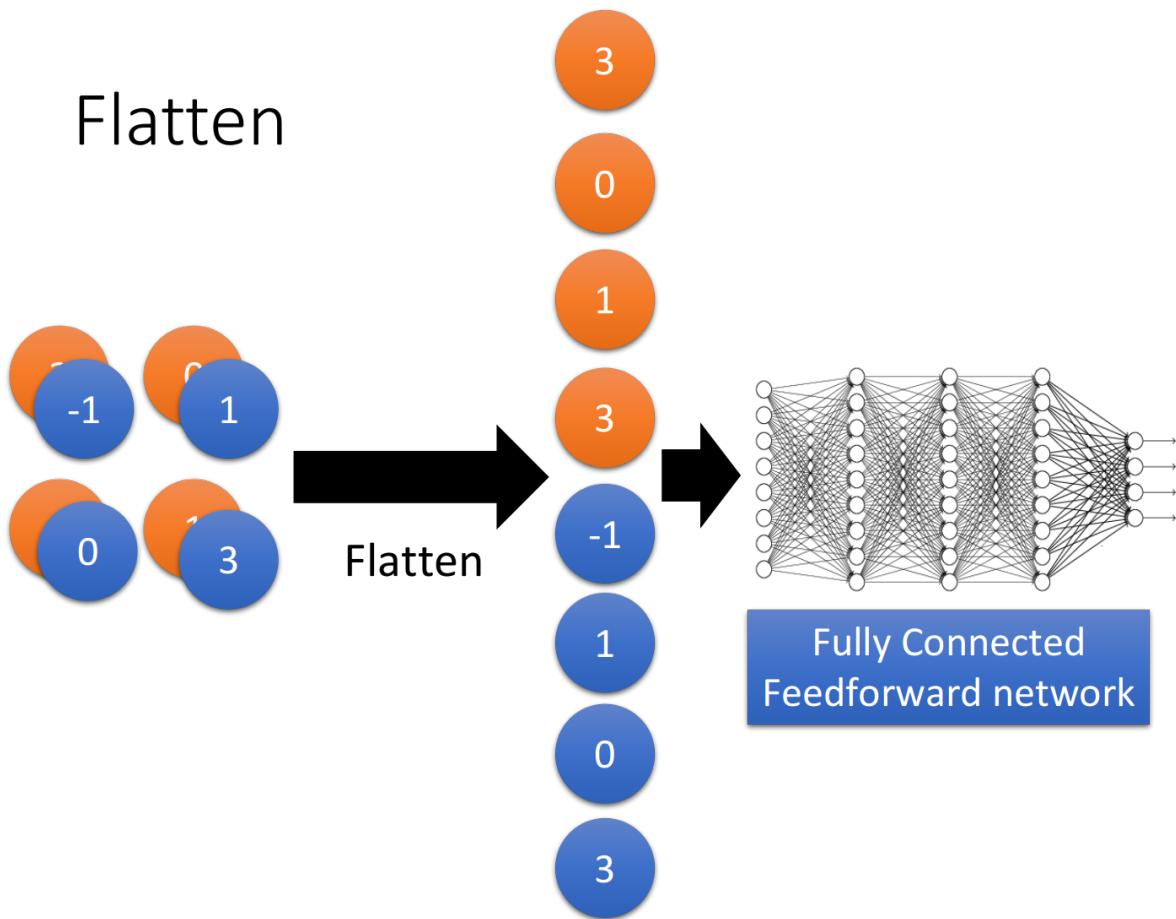
故如果有50个Filter，无论input是什么样子的，最终的output还是会是50层

Flatten

做完convolution和max pooling之后，就是Flatten和Fully connected Feedforward network的部分

Flatten的意思是，把左边的feature map拉直，然后把它丢进一个Fully connected Feedforward network，然后就结束了，也就是说，我们之前通过CNN提取出了image的feature，它相较于原先一整个image的vector，少了很大一部分内容，因此需要的参数也大幅度地减少了，但最终，也还是要丢到一个Fully connected的network中去做最后的分类工作

Flatten



What does CNN learn?

如果今天有一个方法，它可以让你轻易地理解为什么这个方法会下这样的判断和决策的话，那其实你会觉得它不够intelligent；它必须要是你无法理解的东西，这样它才够intelligent，至少你会感觉它很intelligent

所以，大家常说deep learning就是一个黑盒子，你learn出来以后，根本就不知道为什么是这样子，于是你会感觉它很intelligent，但是其实还是有很多方法可以分析的，今天我们就来示范一下怎么分析CNN，看一下它到底学到了什么

要分析第一个convolution的filter是比较容易的，因为第一个convolution layer里面，每一个filter就是一个 3×3 的matrix，它对应到 3×3 范围内的9个pixel，所以你只要看这个filter的值，就可以知道它在detect什么东西，因此第一层的filter是很容易理解的

但是你比较没有办法想像它在做什么事情的，是第二层的filter，它们是50个同样为 3×3 的filter，但是这些filter的input并不是pixel，而是做完convolution再做Max pooling的结果，因此filter考虑的范围并不是 $3 \times 3 = 9$ 个pixel，而是一个长宽为 3×3 ，高为25的cubic，filter实际在image上看到的范围是远大于9个pixel的，所以你就算把它的weight拿出来，也不知道它在做什么

那我们怎么来分析一个filter它做的事情是什么呢？你可以这样做：

我们知道在第二个convolution layer里面的50个filter，每一个filter的output就是一个 11×11 的matrix，假设我们现在把第k个filter的output拿出来，如下图所示，这个matrix里的每一个element，我们叫它 a_{ij}^k ，上标k表示这是第k个filter，下标 ij 表示它在这个matrix里的第i个row，第j个column

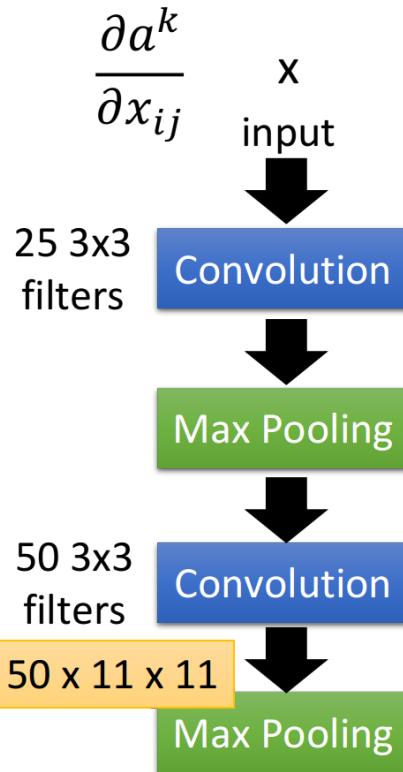
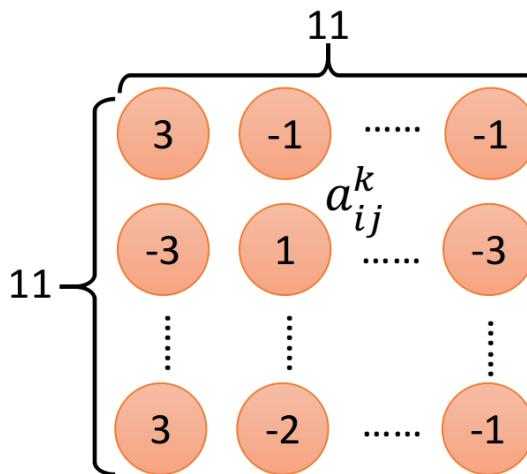
What does CNN learn?

The output of the k-th filter is a 11×11 matrix.

Degree of the activation of the k-th filter:

$$a^k = \sum_{i=1}^{11} \sum_{j=1}^{11} a_{ij}^k$$

$$x^* = \operatorname{argmax}_x a^k \text{ (gradient ascent)}$$



接下来我们define一个 a^k 叫做**Degree of the activation of the k-th filter**, 这个值表示现在的第k个filter, 它有多被activate, 直观来讲就是描述现在input的东西跟第k个filter有多接近, 它对filter的激活程度有多少

第k个filter被启动的degree a^k 就定义成, 它与input进行卷积所输出的output里所有element的summation, 以上图为例, 就是这 11×11 的output matrix里所有元素之和, 用公式描述如下:

$$a^k = \sum_{i=1}^{11} \sum_{j=1}^{11} a_{ij}^k$$

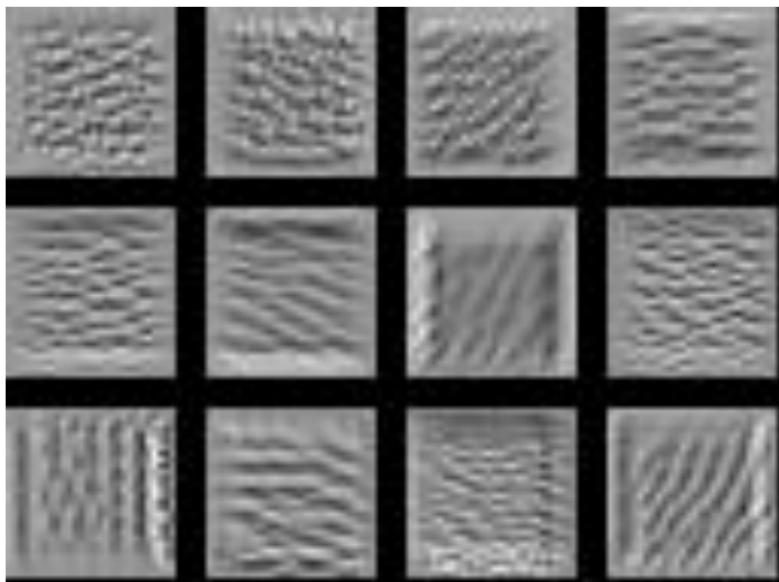
也就是说, 我们input一张image, 然后把这个filter和image进行卷积所output的 11×11 个值全部加起来, 当作现在这个filter被activate的程度

接下来我们要做的事情是这样子, 我们想要知道第k个filter的作用是什么, 那我们就要找一张image, 这张image可以让第k个filter被activate的程度最大; 于是我们现在要解的问题是, 找一个image x , 它可以让我们定义的activation的degree a^k 最大, 即:

$$x^* = \operatorname{arg} \max_x a^k$$

之前我们求minimize用的是gradient descent, 那现在我们求Maximum用gradient ascent就可以做到这件事了

仔细一想这个方法还是颇为神妙的, 因为我们现在是把input x 作为要找的参数, 对它去用gradient descent或ascent进行update, 原来在train CNN的时候, input是固定的, model的参数是要用gradient descent去找出来的; 但是现在这个立场是反过来的, 在这个task里面model的参数是固定的, 我们要用gradient ascent去update这个 x , 让它可以使degree of activation最大



For each filter

上图就是得到的结果，50个filter理论上可以分别找50张image使对应的activation最大，这里仅挑选了其中的12张image作为展示，这些image有一个共同的特征，它们里面都是一些**反复出现的某种texture(纹路)**，比如说第三张image上布满了小小的斜条纹，这意味着第三个filter的工作就是detect图上有没有斜条纹，要知道现在每个filter检测的都只是图上一个小小的范围而已，所以图中一旦出现一个小小的斜条纹，这个filter就会被activate，相应的output也会比较大，所以如果整张image上布满这种斜条纹的话，这个时候它会最兴奋，filter的activation程度是最大的，相应的output值也会达到最大

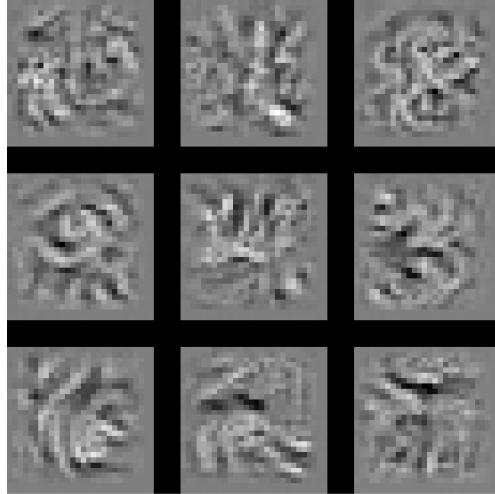
因此每个filter的工作就是去detect某一种pattern，detect某一种线条，上图所示的filter所detect的就是不同角度的线条，所以今天input有不同线条的话，某一个filter会去找到让它兴奋度最高的匹配对象，这个时候它的output就是最大的

我们做完convolution和max pooling之后，会将结果用Flatten展开，然后丢到Fully connected的neural network里面去，之前已经搞清楚了filter是做什么的，那我们也想要知道在这个neural network里的每一个neuron是做什么的，所以就对刚才的做法如法炮制

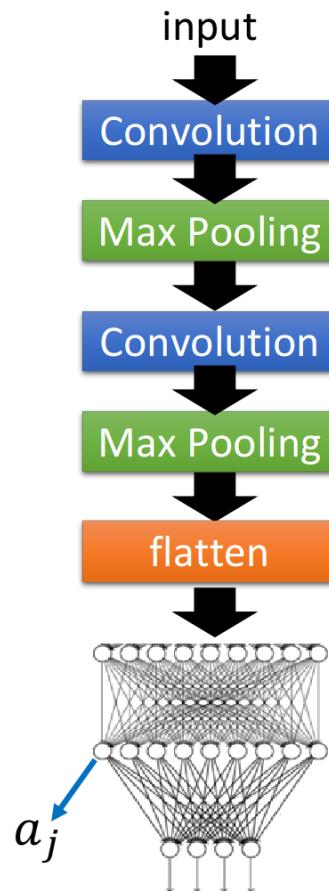
What does CNN learn?

Find an image maximizing the output of neuron:

$$x^* = \operatorname{argmax}_x a_j$$



Each figure corresponds to a neuron



我们定义第j个neuron的output就是 a_j , 接下来就用gradient ascent的方法去找一张image x , 把它丢到neural network里面就可以让 a_j 的值被maximize, 即:

$$x^* = \operatorname{argmax}_x a_j$$

找到的结果如上图所示, 同理这里仅取出其中的9张image作为展示, 你会发现这9张图跟之前filter所观察到的情形是很不一样的, 刚才我们观察到的是类似纹路的东西, 那是因为每个filter考虑的只是图上一部分的vision, 所以它detect的是一种texture;

但是在做完Flatten以后, 每一个neuron不再是只看整张图的一小部分, 它现在的工作是看整张图, 所以对每一个neuron来说, 让它最兴奋的、activation最大的image, 不再是texture, 而是一个完整的图形, 虽然它侦测的不是完整的数字, 但是是比较大的pattern。

接下来我们考虑的是CNN的output, 由于是手写数字识别的demo, 因此这里的output就是10维, 我们把某一维拿出来, 然后同样去找一张image x , 使这个维度的output值最大, 即

$$x^* = \operatorname{argmax}_x y^i$$

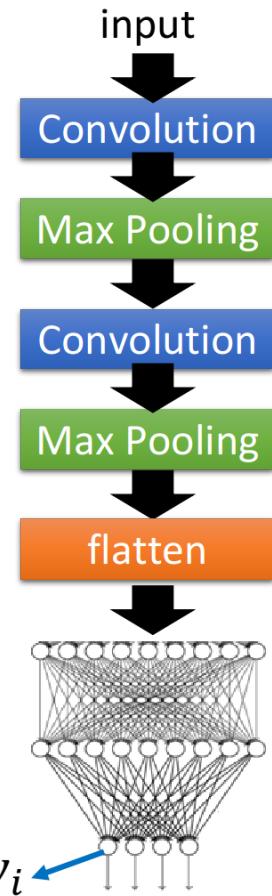
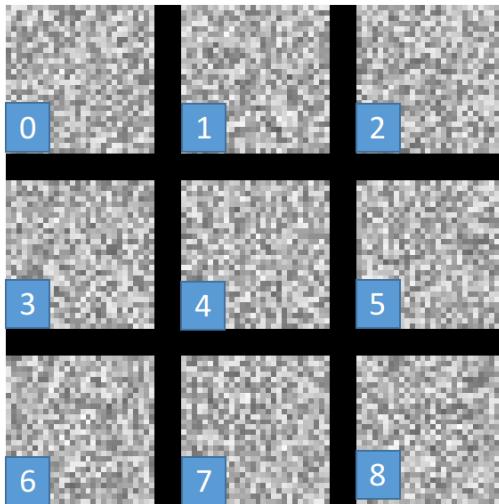
你可以想一想, 既然现在每一个output的每一个dimension就对应到一个数字, 那如果我们去找一张image x , 它可以让对应到数字1的那个output layer的neuron的output值最大, 那这张image显然应该看起来会像是数字1, 你甚至可以期待, 搞不好用这个方法就可以让machine自动画出数字

但实际上, 我们得到的结果是这样子, 如下图所示

What does CNN learn?

$$x^* = \operatorname{argmax}_x y^i$$

Can we see digits?



Deep Neural Networks are Easily Fooled

<https://www.youtube.com/watch?v=M2IebCN9Ht4>

上面的每一张图分别对应着数字0-8，你会发现，可以让数字1对应neuron的output值最大的image其实长得一点也不像1，就像是电视机坏掉的样子，为了验证程序有没有bug，这里又做了一个实验，把上述得到的image真的作为testing data丢到CNN里面，结果classify的结果确实还是认为这些image就对应着数字0-8

所以今天这个neural network，它所学到的东西跟我们人类一般的想象认知是不一样的

那我们有没有办法，让上面这个图看起来更像数字呢？想法是这样的，我们知道一张图是不是一个数字，它会有一些基本的假设，比如这些image，你不知道它是什么数字，你也会认为它显然就不是一个digit，因为人类手写出来的东西就不是长这个样子的，所以我们要对这个x做一些regularization，我们要对找出来的x做一些constraint，我们应该告诉machine说，虽然有一些x可以让你的y很大，但是它们不是数字

那我们应该加上什么样的constraint呢？最简单的想法是说，画图的时候，白色代表的是有墨水、有笔画的地方，而对于一个digit来说，整张image上涂白的区域是有限的，像上面这些整张图都是白白的，它一定不会是数字

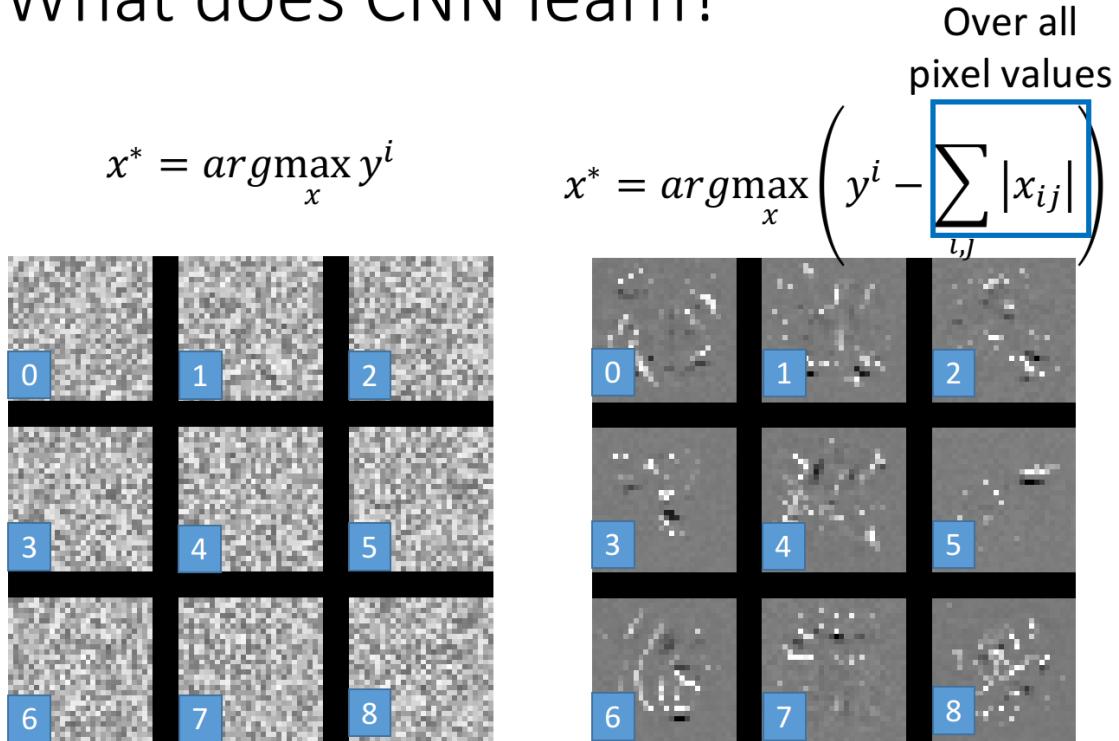
假设image里的每一个pixel都用 x_{ij} 表示，我们把所有pixel值取绝对值并求和，也就是 $\sum_{i,j} |x_{ij}|$ ，这一项其实是之前提到过的L1的regularization，再用 y^i 减去这一项，得到

$$x^* = \operatorname{arg max}_x (y^i - \sum_{i,j} |x_{ij}|)$$

这次我们希望再找一个input x，它可以让 y^i 最大的同时，也要让 $|x_{ij}|$ 的summation越小越好，也就是说我们希望找出来的image，大部分的地方是没有涂颜色的，只有少数数字笔画在的地方才有颜色出现

加上这个constraint以后，得到的结果会像下图右侧所示一样，已经隐约有些可以看出来是数字的形状了

What does CNN learn?



如果再加上一些额外的constraint, 比如你希望相邻的pixel是同样的颜色等等, 你应该可以得到更好的结果

Deep Dream

其实, 这就是Deep Dream的精神, Deep Dream是说, 如果你给machine一张image, 它会在这个image里面加上它看到的东西

怎么做这件事情呢? 你就找一张image丢到CNN里面去, 然后你把某一个convolution layer里面的filter或是fully connected layer里的某一个hidden layer的output拿出来, 它其实是一个vector; 接下来把本来是positive的dimension值调大, negative的dimension值调小, 也就是让正的更正, 负的更负, 然后把它作为新的image的目标

总体来说就是使它们的绝对值变大, 然后用gradient descent的方法找一张image x , 让它通过这个hidden layer后的output就是你调整后的target, 这么做的目的就是, **让CNN夸大化它看到的东西**——make CNN exaggerates what it sees

也就是说, 如果某个filter有被activate, 那你让它被activate的更剧烈, CNN可能本来看到了某一样东西, 那现在你就让它看起来更像原来看到的东西, 这就是所谓的**夸大化**

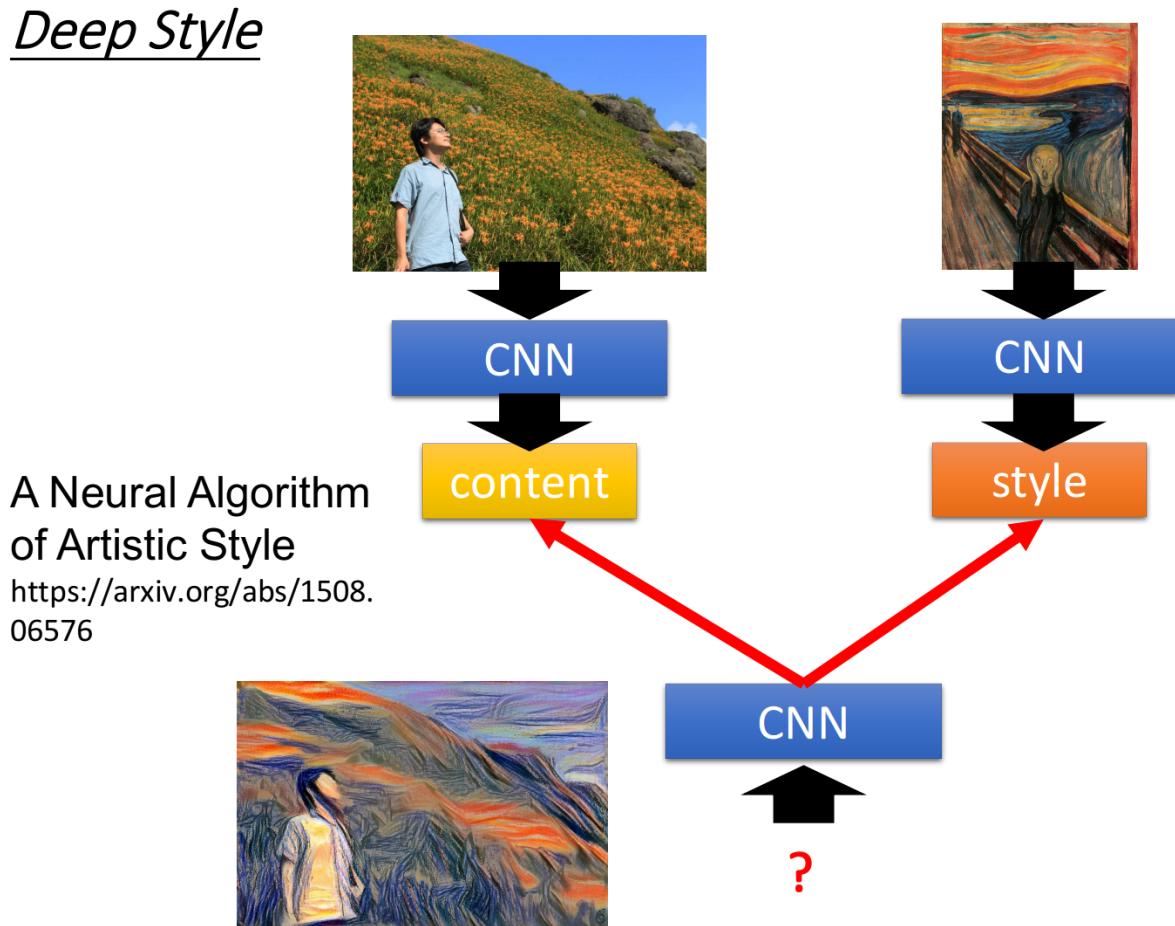
如果你把上面这张image拿去做Deep Dream的话, 你看到的结果就会好像背后有很多念兽, 比如像上图右侧那一只熊, 它原来是一个石头, 对机器来说, 它看这张图的时候, 本来就觉得这个石头有点像熊, 所以你就更强化这件事, 让它看起来真的就变成了一只熊, 这个就是Deep Dream

Deep Style

Deep Dream还有一个进阶的版本, 就叫做Deep Style, 如果今天你input一张image, Deep Style做的事情就是让machine去修改这张图, 让它有另外一张图的风格, 如下所示

实际上机器做出来的效果惊人的好, 具体的做法参考reference: [A Neural Algorithm of Artistic Style](#)

这里仅讲述Deep Style的大致思路，你把原来的image丢给CNN，得到CNN filter的output，代表这样image里面有什么样的content，然后你把呐喊这张图也丢到CNN里面得到filter的output，注意在这时我们并不在意一个filter output的value到底是什么，我们真正在意的是，filter和filter的output之间的correlation，这个correlation代表了一张image的style



接下来你就再用一个CNN去找一张image，**这张image的content像左边的图片**，比如这张image的filter output的value像左边的图片；同时让**这张image的style像右边的图片**，所谓的style像右边的图片是说，这张image output的filter之间的correlation像右边这张图片

最终你用gradient ascent找到一张image，同时可以maximize左边的content和右边的style，它的样子就像上图左下角所示

Application

Playing Go

What does CNN do in Playing Go

CNN可以被运用到不同的应用上，不只是影像处理，比如出名的AlphaGo

想要让machine来下围棋，不见得要用CNN，其实一般typical的neural network也可以帮我们做到这件事情

你只要learn一个network，也就是找一个function，它的input是棋盘当前局势，output是你下一步根据这个棋盘的盘势而应该落子的位置，这样其实就可以让machine学会下围棋了，所以用fully connected的feedforward network也可以做到让machine下围棋这件事情

也就是说，你只要告诉它input是一个 19×19 的vector，vector的每一个dimension对应到棋盘上的某一个位置，如果那一个位置有一个黑子的话，就是1，如果有白子的话，就是-1，反之呢，就是0，所以如果你把棋盘描述成一个 19×19 的vector，丢到一个fully connected的feedforward network里，output也是 19×19 个dimension，每一个dimension对应到棋盘上的一个位置，那machine就可以学会下围棋了。

但实际上如果我们采用CNN的话，会得到更好的performance，我们之前举的例子都是把CNN用在图像上面，也就是input是一个matrix，而棋盘其实可以很自然地表示成一个 19×19 的matrix，那对CNN来说，就是直接把它当成一个image来看待，然后再output下一步要落子的位置，具体的training process是这样的：

你就搜集很多棋谱，比如说初手下在5之五，次手下在天元，然后再下在5之五，接下来你就告诉machine说，看到落子在5之五，CNN的output就是天元的地方是1，其他的output是0；看到5之五和天元都有子，那你的output就是5之五的地方是1，其他都是0。

上面是supervised的部分，那其实呢Alpha Go还有reinforcement learning的部分，后面会讲到。

Why CNN for Playing Go

自从AlphaGo用了CNN以后，大家都觉得好像CNN应该很厉害，所以有时候如果你没有用CNN来处理问题，人家就会来问你；比如你去面试的时候，你的论文里面没有用CNN来处理问题，面试的人可能不知道CNN是什么，但是他就会问你说为什么不用CNN呢，CNN不是比较强吗？这个时候如果你真的明白了为什么要用CNN，什么时候才要用CNN这个问题，你就可以直接给他怼回去。

那什么时候我们可以用CNN呢？你要有image该有的那些特性，也就是上一篇文章开头所说的，根据观察到的三个property，我们才设计出了CNN这样的network架构：

- **Some patterns are much smaller than the whole image**
- **The same patterns appear in different regions**
- **Subsampling the pixels will not change the object**

CNN能够应用在AlphaGo上，是因为围棋有一些特性和图像处理是很相似的。

在property 1，有一些pattern是比整张image要小得多，在围棋上，可能也有同样的现象，比如一个白子被3个黑子围住，如果下一个黑子落在白子下面，就可以把白子提走；只有另一个白子接在下面，它才不会被提走。

那现在你只需要看这个小小的范围，就可以侦测这个白子是不是属于被叫吃的状态，你不需要看整个棋盘，才知道这件事情，所以这件事情跟image有着同样的性质；在AlphaGo里面，它第一个layer其实都是用 5×5 的filter，显然做这个设计的人，觉得围棋上最基本的pattern可能都是在 5×5 的范围内就可以被侦测出来。

在property 2，同样的pattern可能会出现在不同的region，在围棋上也可能有这个现象，像这个叫吃的pattern，它可以出现在棋盘的左上角，也可以出现在右下角，它们都是叫吃，都代表了同样的意义，所以你可以用同一个detector，来处理这些在不同位置的同样的pattern。

所以对围棋来说呢，它在第一个observation和第二个observation是有这个image的特性的，但是，让我们没有办法想通的地方，就是第三点。

我们可以对一个image做subsampling，你拿掉奇数行、偶数列的pixel，把image变成原来的 $1/4$ 的大小也不会影响你看这张图的样子，基于这个观察才有了Max pooling这个layer；但是，对围棋来说，它可以做这件事情吗？比如说，你对一个棋盘丢掉奇数行和偶数列，那它还和原来是同一个吗？显然不是的。

如何解释在棋盘上使用Max Pooling这件事情呢？有一些人觉得说，因为AlphaGo使用了CNN，它里面有可能用了Max pooling这样的构架，所以，或许这是它的一个弱点，你要针对这个弱点攻击它，也许就可以击败它。

AlphaGo的paper内容不多，只有6页左右，它只说使用了CNN，却没有在正文里面仔细地描述它的CNN构架，但是在这篇paper长长附录里，其实是有描述neural network structure的。

它是这样说的，input是一个 $19 \times 19 \times 48$ 的image，其中 19×19 是棋盘的格局，对Alpha来说，每一个位置都用48个value来描述，这是因为加上了domain knowledge，它不只是描述某位置有没有白子或黑子，它还会观察这个位置是不是处于叫吃的状态等等

- Subsampling the pixels will not change the object



Max Pooling

How to explain this???

Neural network architecture. The input to the policy network is a $19 \times 19 \times 48$ image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23×23 image, then convolves k filters of kernel size 5×5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21×21 image, then convolves k filters of kernel size 3×3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1×1 with stride 1, with a different bias for each position, and applies a softmax function. The Alpha Go does not use Max Pooling Extended Data Table 3 additionally show the results of training with $k = 128, 256$ and 384 filters.

先用一个hidden layer对image做zero padding，也就是把原来 19×19 的image外围补0，让它变成一张 23×23 的image，然后使用 k 个 5×5 的filter对该image做convolution，stride设为1，activation function用的是ReLU，得到的output是 21×21 的image；接下来使用 k 个 3×3 的filter，stride设为1，activation function还是使用ReLU，...

你会发现这个AlphaGo的network structure一直在用convolution，其实根本就没有使用Max Pooling，原因并不是疏失了什么之类的，而是根据围棋的特性，我们本来就不需要在围棋的CNN里面，用Max pooling这样的构架

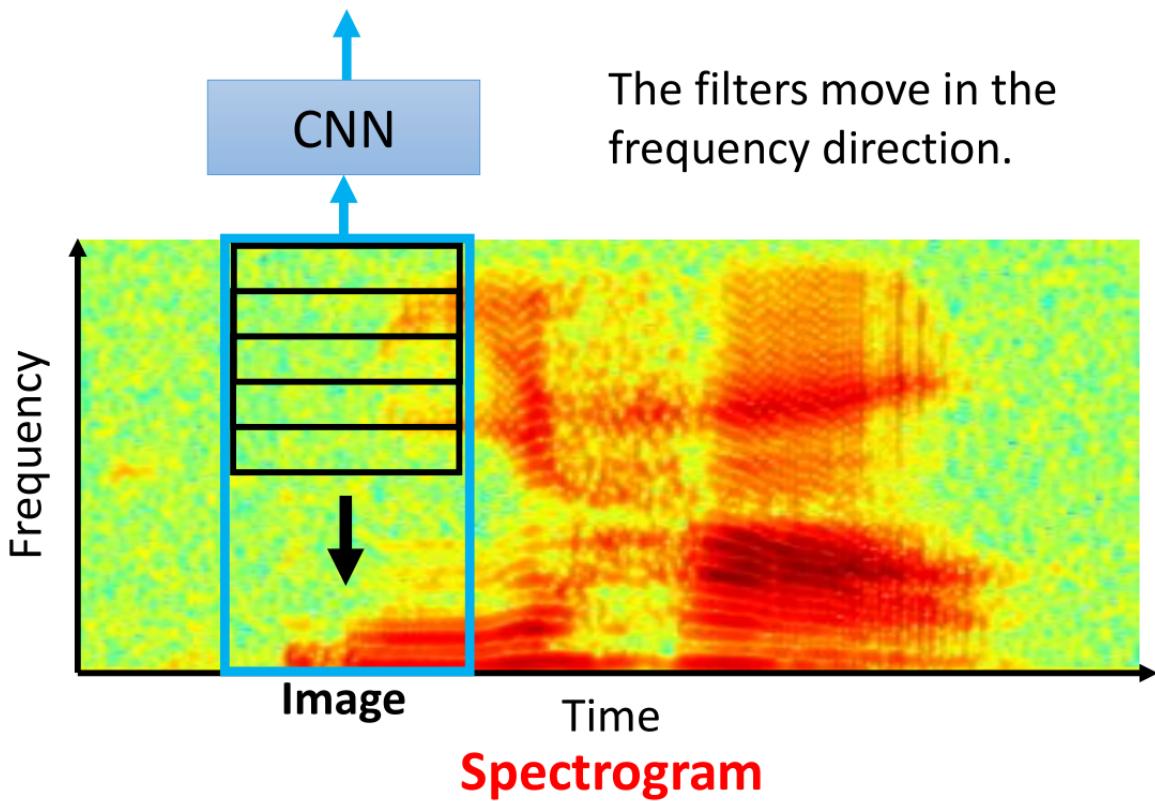
举这个例子是为了告诉大家：neural network架构的设计，是应用之道，存乎一心

Speech

CNN也可以用在很多其他的task里面，比如语音处理上，我们可以把一段声音表示成spectrogram，spectrogram的横轴是时间，纵轴则是这一段时间里声音的频率

下图中是一段“你好”的音频，偏红色代表这段时间里该频率的energy是比较大的，也就对应着“你”和“好”这两个字，也就是说spectrogram用颜色来描述某一个时刻不同频率的能量

我们也可以让机器把这个spectrogram就当作一张image，然后用CNN来判断说，input的这张image对应着什么样的声音信号，那通常用来判断结果的单位，比如phoneme，就是类似音标这样的单位



这边比较神奇的地方就是，当我们把一段spectrogram当作image丢到CNN里面的时候，在语音上，我们通常只考虑在frequency(频率)方向上移动的filter，我们的filter就像上图这样，是长方形的，它的宽就跟image的宽是一样的，并且filter只在Frequency即纵坐标的方向上移动，而在时间的序列上移动

这是因为在语音里面，CNN的output后面都还会再接别的东西，比如接LSTM之类，所以你在CNN里面再考虑一次时间的information其实没有什么特别的帮助，但是为什么在频率上的filter有帮助呢？

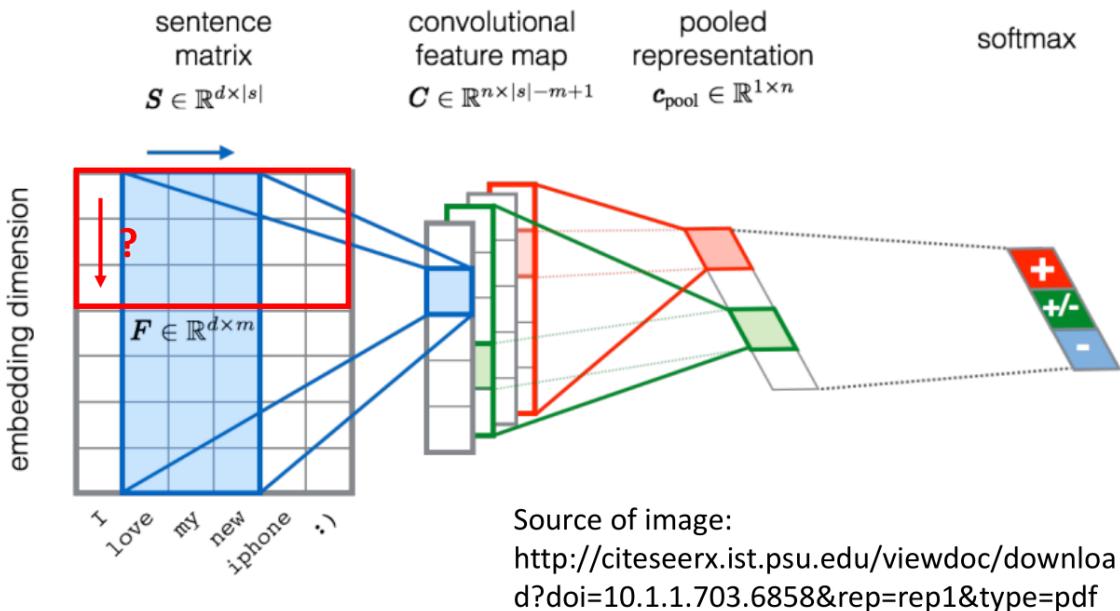
我们用CNN的目的是为了用同一个filter把相同的pattern给detect出来，在声音讯号上，虽然男生和女生说同样的话看起来这个spectrogram是非常不一样的，但实际上他们的不同只是表现在一个频率的shift而已，男生说的你好跟女生说的你好，它们的pattern其实是一样的，比如pattern是spectrogram变化的情形，男生女生的声音的变化情况可能是一样的，它们的差别可能只是所在的频率范围不同而已，所以filter在frequency的方向上移动是有效的，在time domain上移动是没有帮助的。

所以，这又是另外一个例子，当你把CNN用在一个Application的时候呢，你永远要想一想这个Application的特性是什么，根据这个特性你再去design network的structure，才会真正在理解的基础上去解决问题

Text

CNN也可以用在文字处理上，假设你的input是一个word sequence，你要做的事情是让machine侦测这个word sequence代表的意思是positive的还是negative的

首先你把这个word sequence里面的每一个word都用一个vector来表示，vector代表的这个word本身的semantic，那如果两个word本身含义越接近的话，它们的vector在高维的空间上就越接近，这个东西就叫做word embedding



把一个sentence里面所有word的vector排在一起，它就变成了一张image，你把CNN套用到这个image上，那filter的样子就是上图蓝色的matrix，它的高和image的高是一样的，然后把filter沿着句子里词汇的顺序来移动，每个filter移动完成之后都会得到一个由内积结果组成的vector，不同的filter就会得到不同的vector，接下来做Max pooling，然后把Max pooling的结果丢到fully connected layer里面，你就会得到最后的output

与语音处理不同的是，在文字处理上，filter只在时间的序列（按照word的顺序，蓝色的方向）上移动，而不在这个embedding dimension上移动

因为在word embedding里面，不同dimension是independent的，它们是相互独立的，不会出现有两个相同的pattern的情况，所以在这个方向上面移动filter，是没有意义的

所以这又是另外一个例子，虽然大家觉得CNN很powerful，你可以用在各个不同的地方，但是当你应用到一个新的task的时候，你要想一想这个新的task在设计CNN的构架的时候，到底该怎么做

Reference

如果你想知道更多visualization的事情，以下是一些reference

- The methods of visualization in these slides
 - <https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html>
- More about visualization
 - <http://cs231n.github.io/understanding-cnn/>
- Very cool CNN visualization toolkit
 - <http://yosinski.com/deepvis>
 - <http://scs.ryerson.ca/~aharley/vis/conv/>

如果你想要用Deep Dream的方法来让machine自动产生一个digit，这件事是不太成功的，但是有很多其它的方法，可以让machine画出非常清晰的图。这里列了几个方法，比如说：PixelRNN，VAE，GAN等进行参考。

- PixelRNN
 - <https://arxiv.org/abs/1601.06759>
- Variation Autoencoder(VAE)
 - <https://arxiv.org/abs/1312.6114>
- Generative Adversarial Network(GAN)

- <https://arxiv.org/abs/1406.2661>

Recurrent Neural Network

Introduction

Slot Filling

How to represent each word as a vector?

- 1-of-N encoding
- Beyond 1-of-N encoding
 - Dimension for “Other”
 - Word hashing

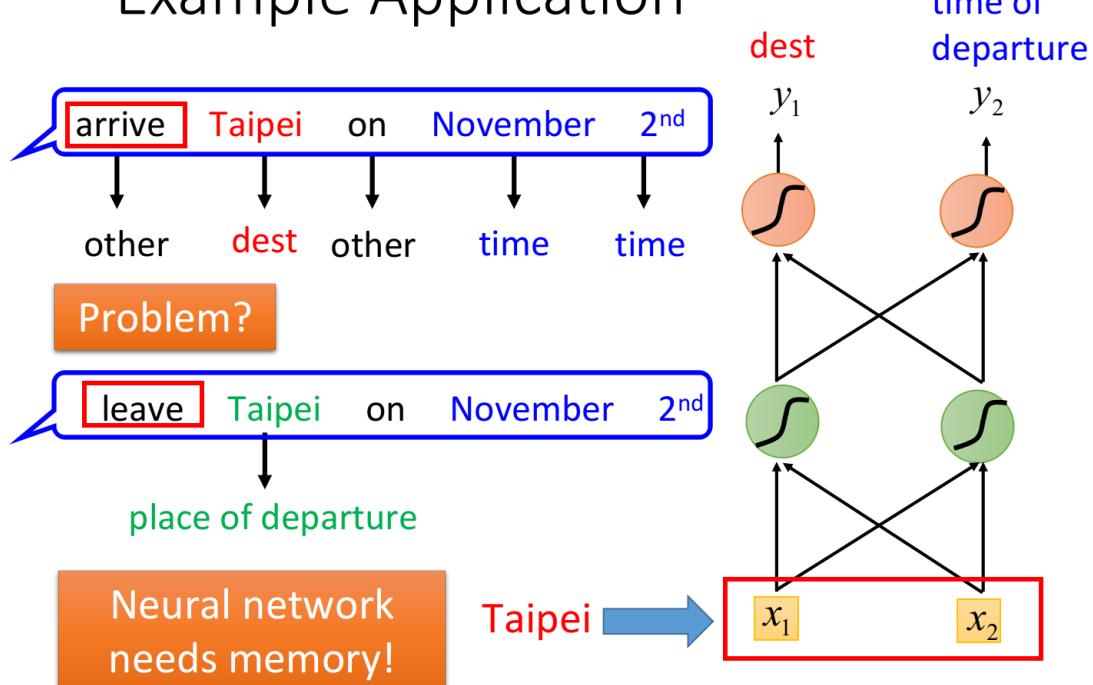
在智能客服、智能订票系统中，往往需要slot filling技术，它会分析用户说出的语句，将时间、地址等有效的关键词填到对应的槽上，并过滤掉无效的词语

Solving slot filling by Feedforward network?

- Input: a word (Each word is represented as a vector)
- Output: Probability distribution that the input word belonging to the slots

但这样做会有一个问题，句子中“arrive”和“leave”这两个词汇，它们都属于“other”，这时对NN来说，输入是相同的，它没有办法区分出“Taipei”是出发地还是目的地

Example Application



这个时候我们就希望神经网络是有记忆的，如果NN在看到“Taipei”的时候，还能记住之前已经看过的“arrive”或是“leave”，就可以根据上下文得到正确的答案

这种有记忆力的神经网络，就叫做**Recurrent Neural Network(RNN)**

在RNN中，hidden layer每次产生的output a_1 、 a_2 ，都会被存到memory里，下一次有input的时候，这些neuron就不仅会考虑新输入的 x_1 、 x_2 ，还会考虑存放在memory中的 a_1 、 a_2

Example

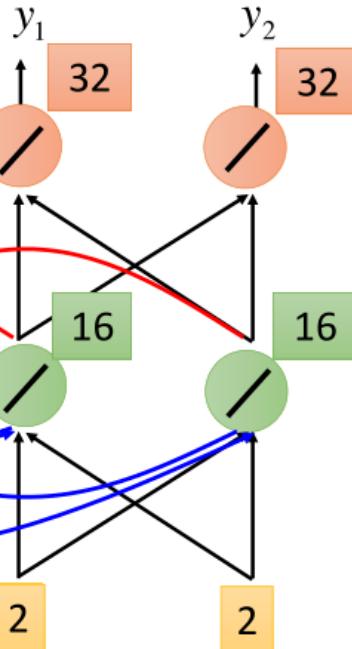
Input sequence: $\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} \dots \dots$

output sequence: $\begin{bmatrix} 4 \\ 4 \end{bmatrix} \begin{bmatrix} 12 \\ 12 \end{bmatrix} \begin{bmatrix} 32 \\ 32 \end{bmatrix}$

Changing the sequence order will change the output.

All the weights are “1”, no bias
All activation functions are linear

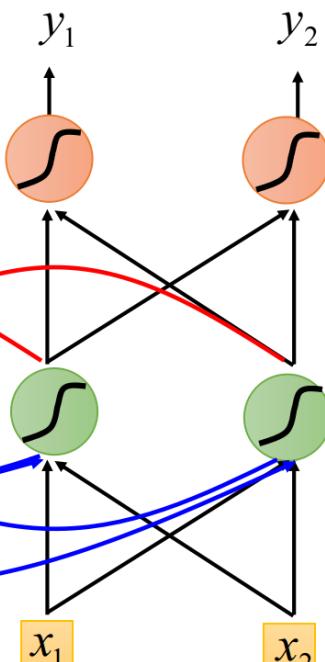
注：在input之前，要先给内存里的 a_i 赋初始值，比如0



Recurrent Neural Network (RNN)

The output of hidden layer are stored in the memory.

Memory can be considered as another input.



注意到，每次NN的输出都要考虑memory中存储的临时值，而不同的输入产生的临时值也尽不相同，因此改变输入序列的顺序会导致最终输出结果的改变，Changing the sequence order will change the output

Slot Filling with RNN

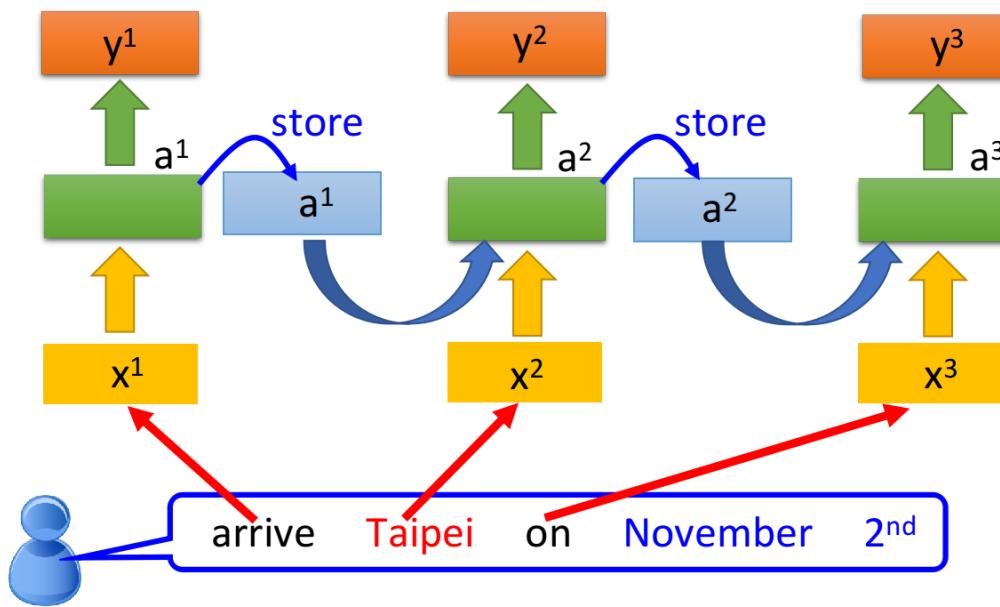
用RNN处理Slot Filling的流程举例如下：

- “arrive”的vector作为 x^1 输入RNN，通过hidden layer生成 a^1 ，再根据 a^1 生成 y^1 ，表示“arrive”属于每个slot的概率，其中 a^1 会被存储到memory中
- “Taipei”的vector作为 x^2 输入RNN，此时hidden layer同时考虑 x^2 和存放在memory中的 a^1 ，生成 a^2 ，再根据 a^2 生成 y^2 ，表示“Taipei”属于某个slot的概率，此时再把 a^2 存到memory中
- 依次类推

RNN

The same network is used again and again.

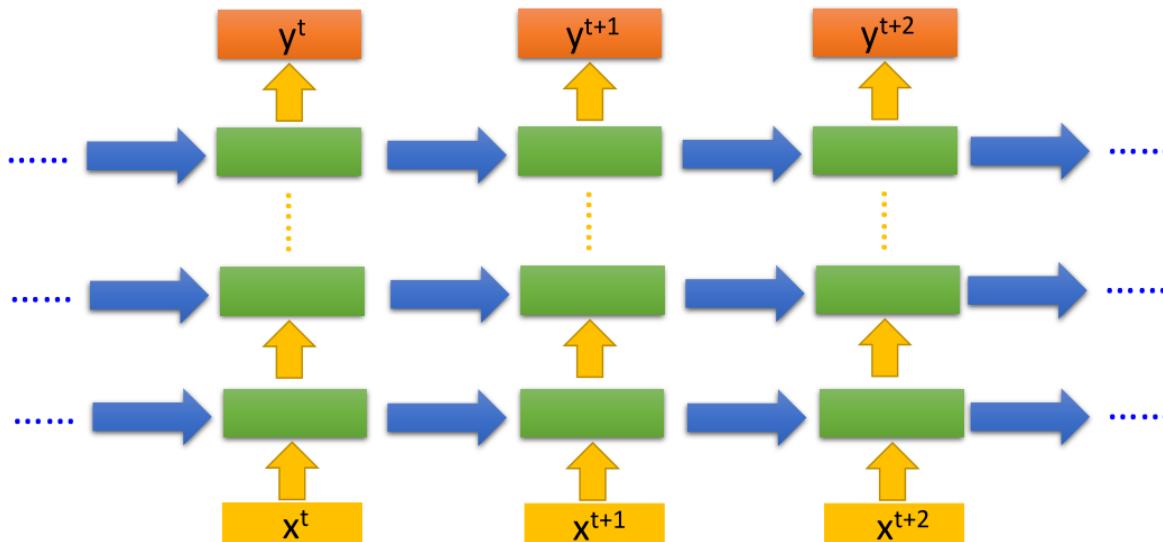
Probability of “arrive” in each slot Probability of “Taipei” in each slot Probability of “on” in each slot



注意：上图为同一个RNN在三个不同时间点被分别使用了三次，并非是三个不同的NN

这个时候，即使输入同样是“Taipei”，我们依旧可以根据前文的“leave”或“arrive”来得到不一样的输出

Deeper RNN

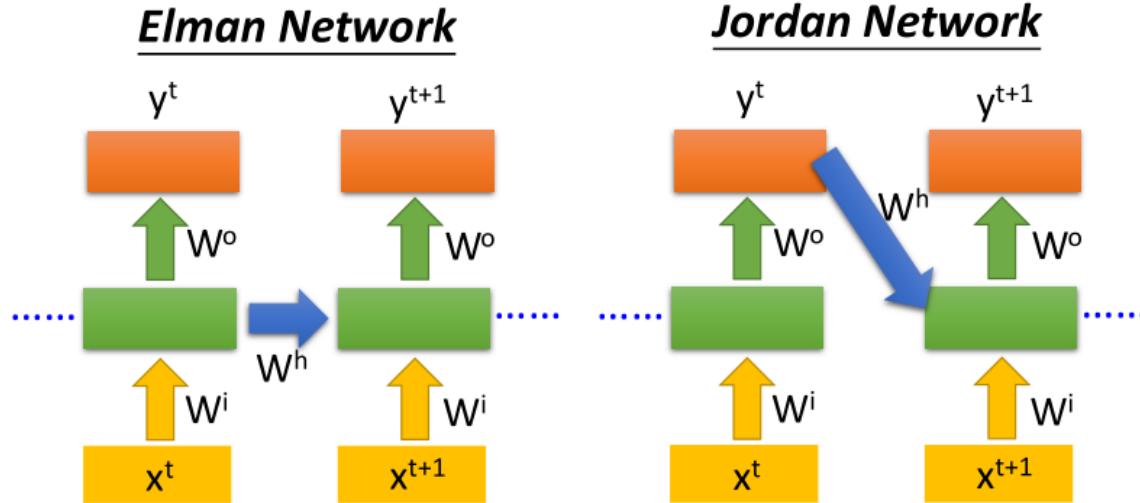


Elman Network & Jordan Network

RNN有不同的变形：

- Elman Network：将hidden layer的输出保存在memory里
- Jordan Network：将整个neural network的输出保存在memory里

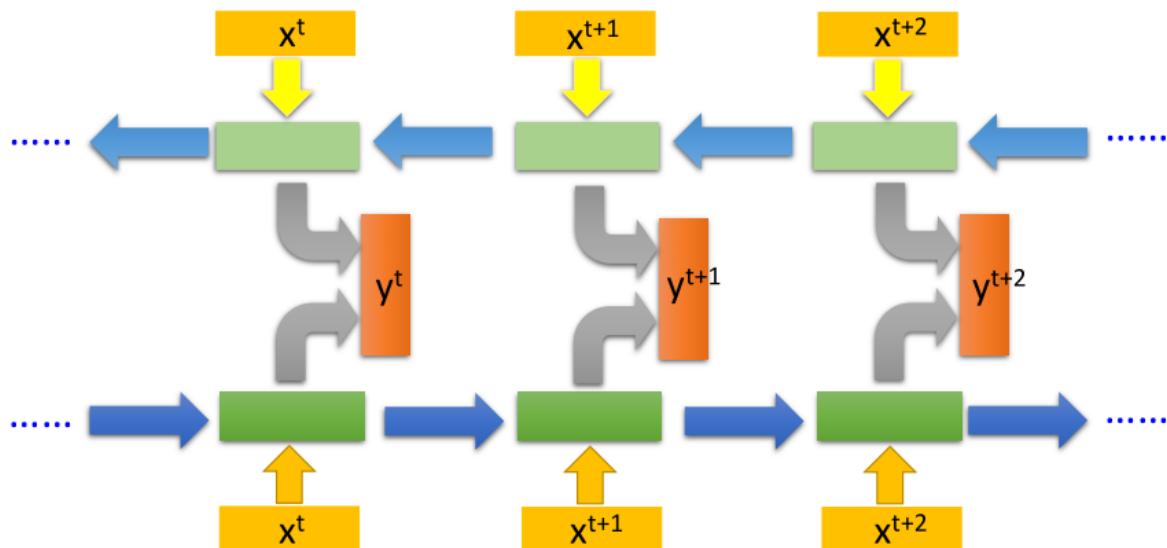
由于hidden layer没有明确的训练目标，而整个NN具有明确的目标， y 是有target的，所以可以比较清楚放在memory里面是什么样的东西。因此Jordan Network的表现会更好一些



Bidirectional RNN

RNN 还可以是双向的，你可以同时训练一对正向和反向的RNN，把它们对应的hidden layer x^t 拿出来，都接给一个output layer，得到最后的 y^t

使用Bi-RNN的好处是，NN在产生输出的时候，它能够看到的范围是比较广的，RNN在产生 y^{t+1} 的时候，它不只看了从句首 x^1 开始到 x^{t+1} 的输入，还看了从句尾 x^n 一直到 x^{t+1} 的输入，这就相当于RNN在看了整个句子之后，才决定每个词汇具体要被分配到哪一个槽中，这会比只看句子的前一半要更好



LSTM

前文提到的RNN只是最简单的版本，并没有对memory的管理多加约束，可以随时进行读取，而现在常用的memory管理方式叫做长短期记忆Long Short-term Memory简称LSTM

可以被理解为比较长的短期记忆，因此是short-term

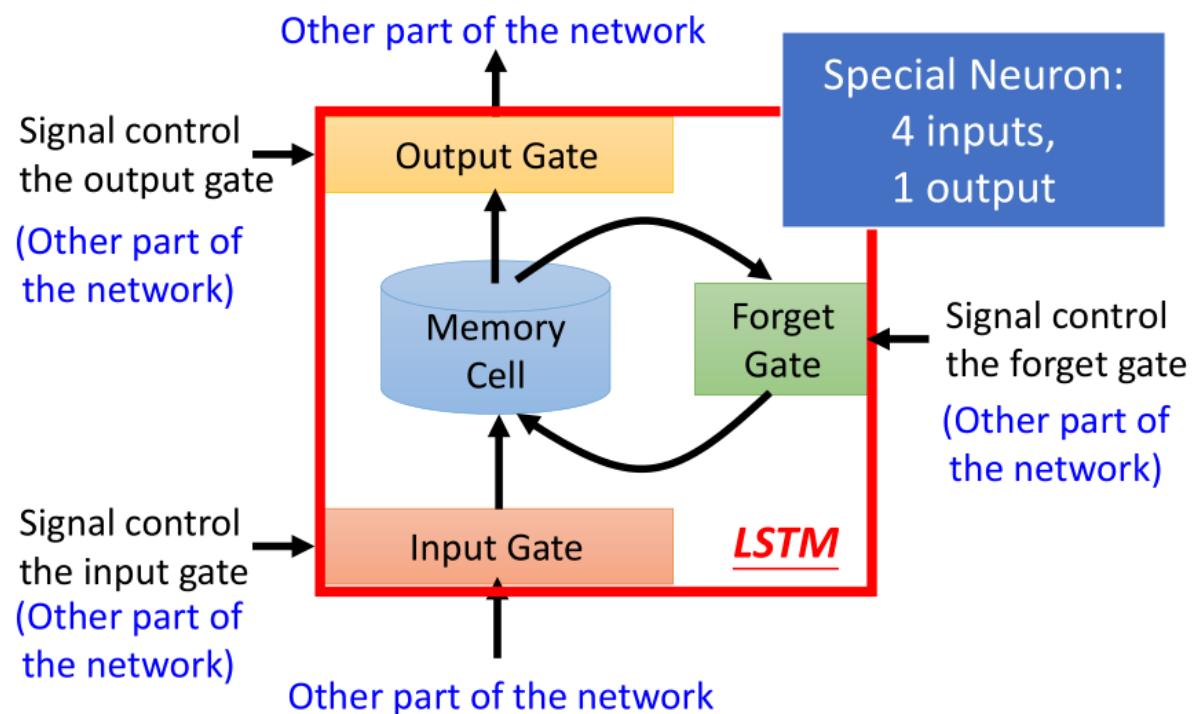
Three-gate

LSTM有三个gate：

- 当某个neuron的输出想要被写进memory cell，它就必须先经过一道叫做**input gate**的闸门，如果input gate关闭，则任何内容都无法被写入，而关闭与否、什么时候关闭，都是由神经网络自己学习到的
- output gate决定了外界是否可以从memory cell中读取值，当**output gate**关闭的时候，memory里面的内容同样无法被读取，同样关闭与否、什么时候关闭，都是由神经网络自己学习到的
- forget gate**则决定了什么时候需要把memory cell里存放的内容忘记清空，什么时候依旧保存

整个LSTM可以看做是4个input，1个output：

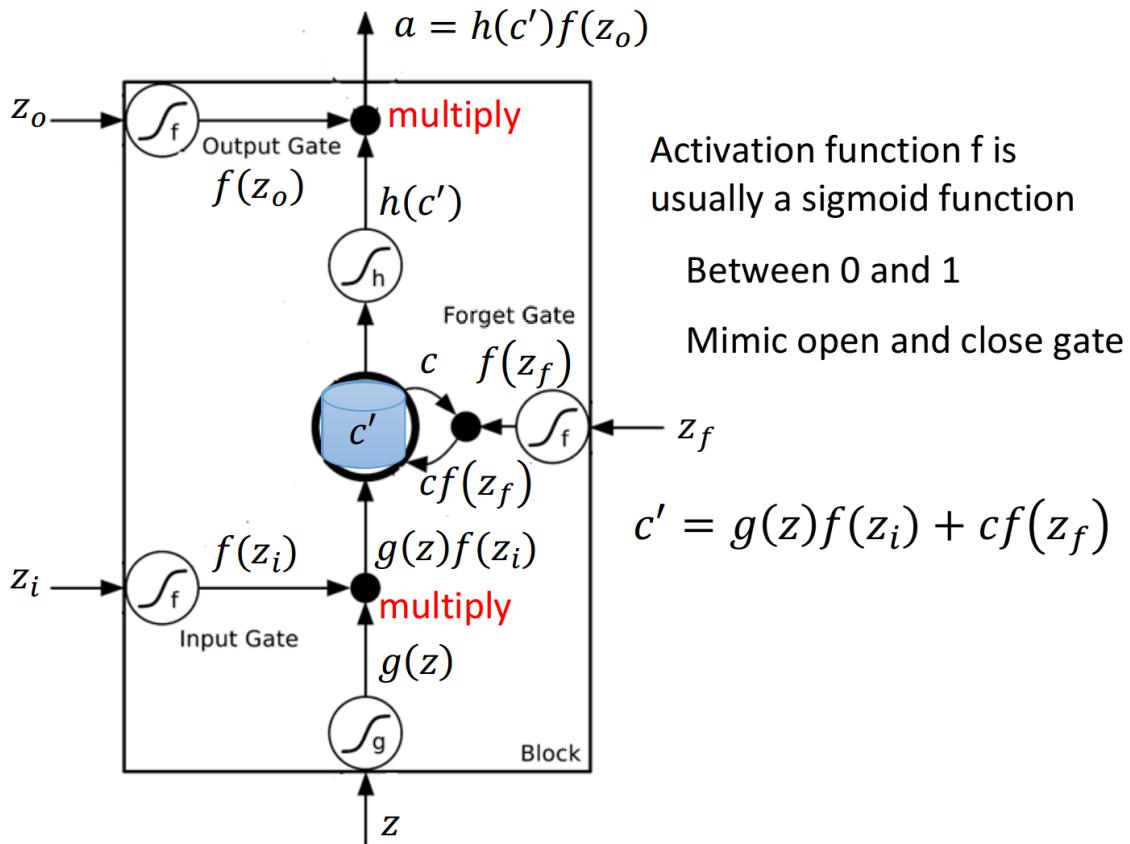
- 4个input=想要被存到memory cell里的值+操控input gate的信号+操控output gate的信号+操控forget gate的信号
- 1个output=想要从memory cell中被读取的值



Memory Cell

如果从表达式的角度看LSTM，它比较像下图中的样子

- z 是想要被存到cell里的输入值
- z_i 是操控input gate的信号
- z_o 是操控output gate的信号
- z_f 是操控forget gate的信号
- a 是综合上述4个input得到的output值



把 z 、 z_i 、 z_o 、 z_f 通过activation function, 分别得到 $g(z)$ 、 $f(z_i)$ 、 $f(z_o)$ 、 $f(z_f)$

其中对 z_i 、 z_o 和 z_f 来说, 它们通过的激活函数 $f()$ 一般会选sigmoid function, 因为它的输出在0~1之间, 代表gate被打开的程度

令 $g(z)$ 与 $f(z_i)$ 相乘得到 $g(z) \cdot f(z_i)$, 然后把原先存放在cell中的 c 与 $f(z_f)$ 相乘得到 $cf(z_f)$, 两者相加得到存在memory中的新值 $c' = g(z) \cdot f(z_i) + cf(z_f)$

- 若 $f(z_i) = 0$, 则相当于没有输入, 若 $f(z_i) = 1$, 则相当于直接输入 $g(z)$
- 若 $f(z_f) = 1$, 则保存原来的值 c 并加到新的值上, 若 $f(z_f) = 0$, 则旧的值将被遗忘清除

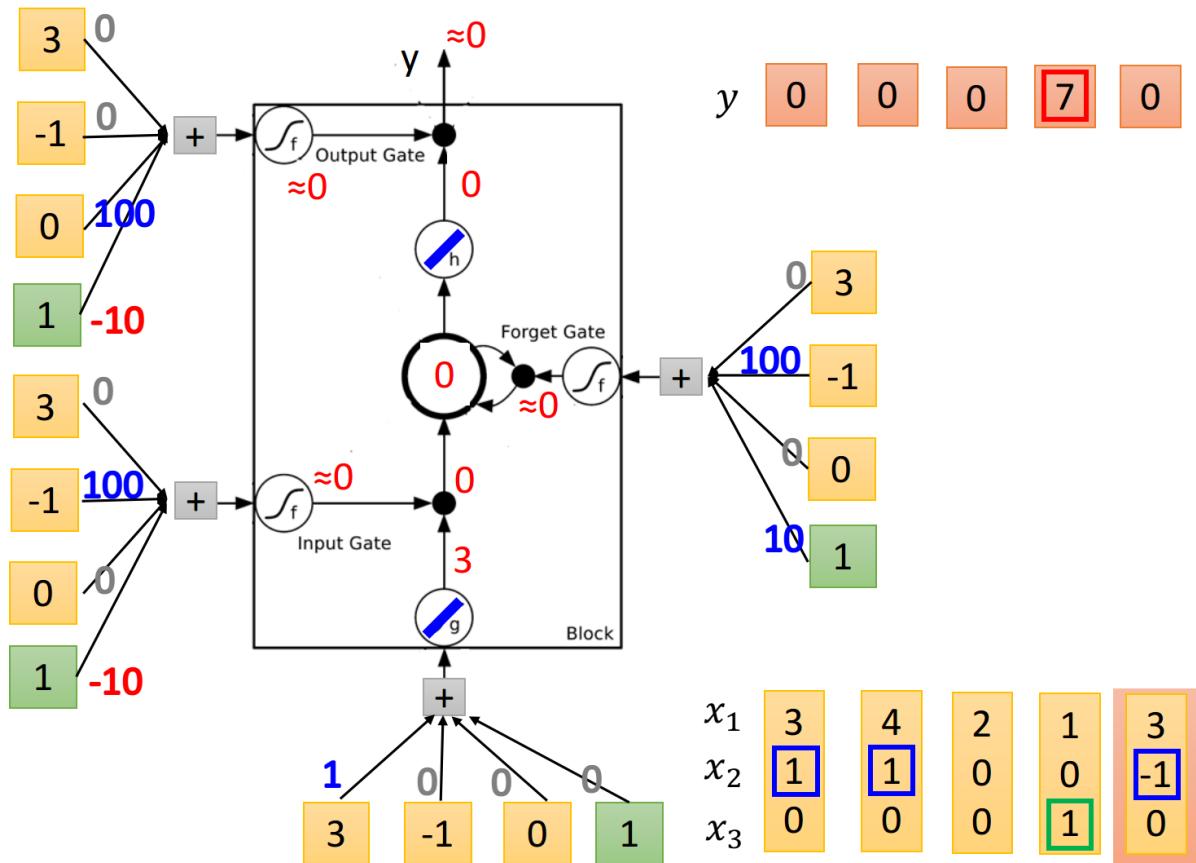
从中也可以看出, forget gate的逻辑与我们的直觉是相反的, 控制信号打开表示记得, 关闭表示遗忘

此后, c' 通过激活函数得到 $h(c')$, 与output gate的 $f(z_o)$ 相乘, 得到输出 $a = h(c')f(z_o)$

LSTM Example

下图演示了一个LSTM的基本过程, x_1 、 x_2 、 x_3 是输入序列, y 是输出序列, 基本原则是:

- 当 $x_2 = 1$ 时, 将 x_1 的值写入memory
- 当 $x_2 = -1$ 时, 将memory里的值清零
- 当 $x_3 = 1$ 时, 将memory里的值输出
- 当neuron的输入为正时, 对应gate打开, 反之则关闭

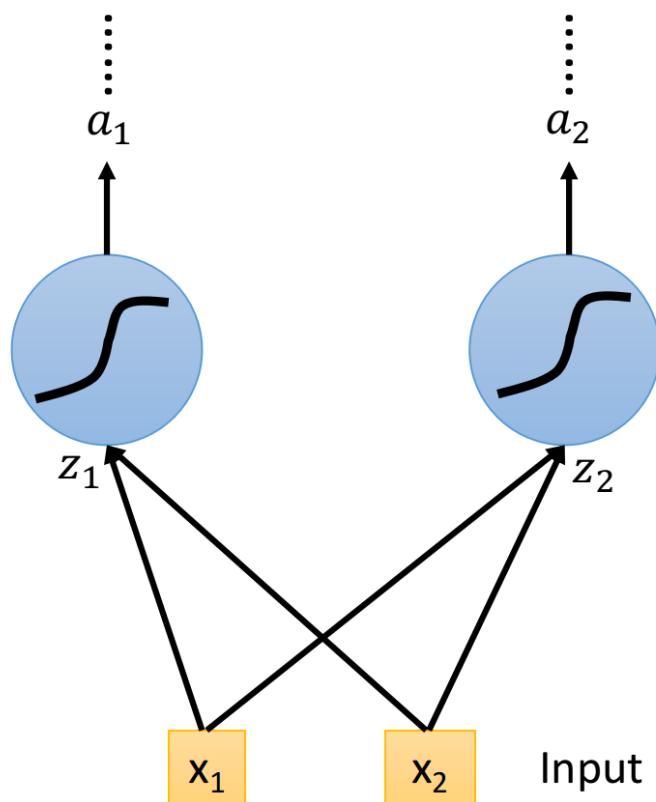


LSTM Structure

你可能会觉得上面的结构与平常所见的神经网络不太一样，实际上我们只需要把LSTM整体看做是下面的一个neuron即可

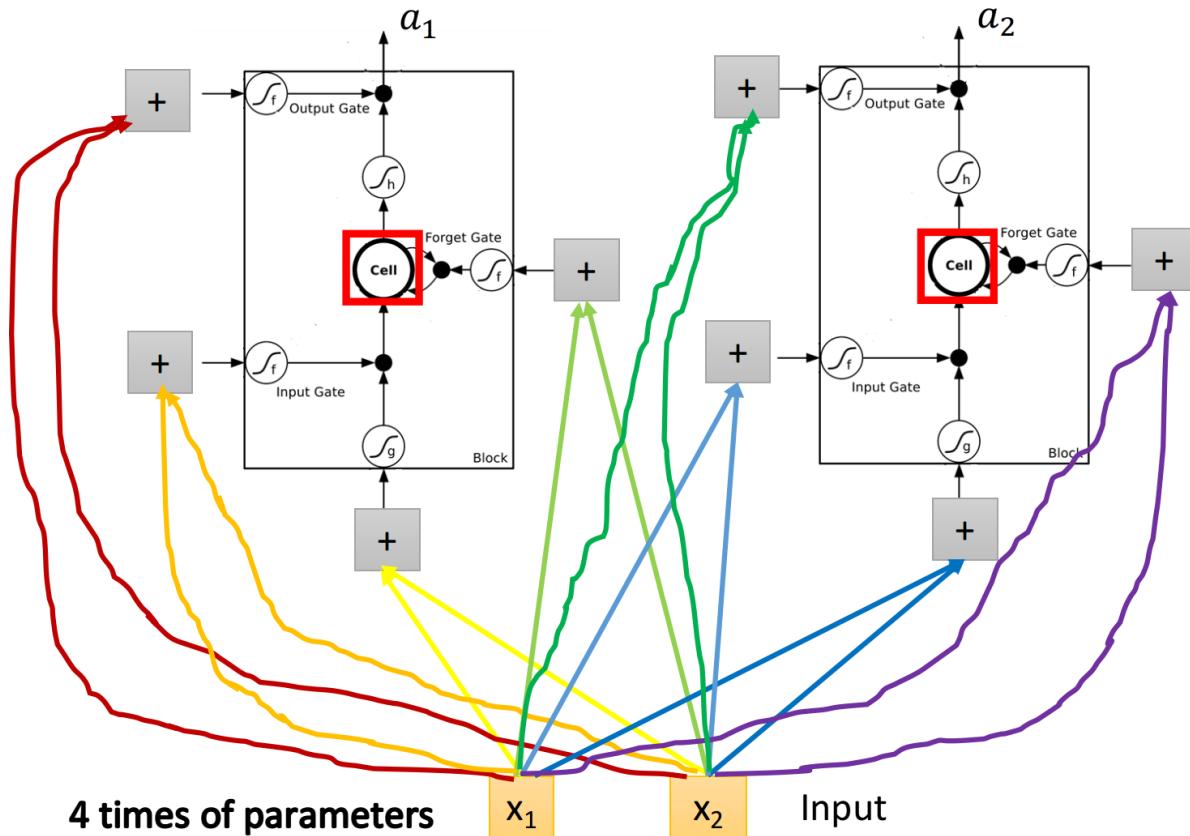
Original Network:

➤ Simply replace the neurons with LSTM



假设目前我们的hidden layer只有两个neuron，则结构如下图所示：

- 输入 x_1 、 x_2 会分别乘上四组不同的weight，作为neuron的输入以及三个状态门的控制信号
- 在原来的neuron里，1个input对应1个output，而在LSTM里，4个input才产生1个output，并且所有的input都是不相同的
- 从中也可以看出LSTM所需要的参数量是一般NN的4倍



LSTM

从上图中你可能看不出LSTM与RNN有什么关系，接下来我们用另外的图来表示它

假设我们现在有一整排的LSTM作为neuron，每个LSTM的cell里都存了一个scalar值，把所有的scalar连接起来就组成了一个vector c^{t-1}

在时间点 t ，输入了一个vector x^t ，它会乘上一个matrix，通过转换得到 z ，而 z 的每个dimension就代表了操控每个LSTM的输入值，同理经过不同的转换得到 z^i 、 z^f 和 z^o ，得到操控每个LSTM的门信号

假设我们现在有一整排的neuron 假设有一整排的LSTM，那这一整排的LSTM里面，每一个LSTM的cell，它里面都存了一个scalar，把所有的scalar接起来，它就变成一个vector，这边写成 c^{t-1} ，那你就可以想成这边每一个memory它里面存的scalar，就是代表这个vector里面的一个dimension，现在在时间点 t ，input一个vector x^t ，这个vector，它会先乘上一个linear的transform，乘上一个matrix，变成另外一个vector z ，这个 z 也是一个vector， z 这个vector的每一个dimension，就操控每一个LSTM的input，所以 z 它的dimension就正好是LSTM的memory cell的数目。那这个 z 的第一维就丢给第一个cell，第二维就丢给第二个cell，以此类推。

x^t 会再乘上另外一个transform，得到 z^i ，然后这个 z^i 呢，它的 dimension 也跟 cell 的数目一样， z^i 的每一个 dimension，都会去操控一个 input gate，所以 z^i 的第一维就是，去操控第一个 cell 的 input gate，第二维，就是操控第二个 cell 的 input gate，最后一维，就是操控最后一个 cell 的 input gate

那 forget gate 跟 output gate 也是一样，把 x^t 乘上一个 transform，得到 z^f ， z^f 会去操控每一个 forget gate，然后 x^t 乘上另外一个 transform，得到 z^o ， z^o 会去操控每一个 cell 的 output gate

所以我们要把 x^t 乘上 4 个不同的 transform，得到 4 个不同的 vector，这 4 个 vector 的 dimension，都跟 cell 的数目是一样的，那这 4 个 vector 合起来，就会去操控这些 memory cell 的运作

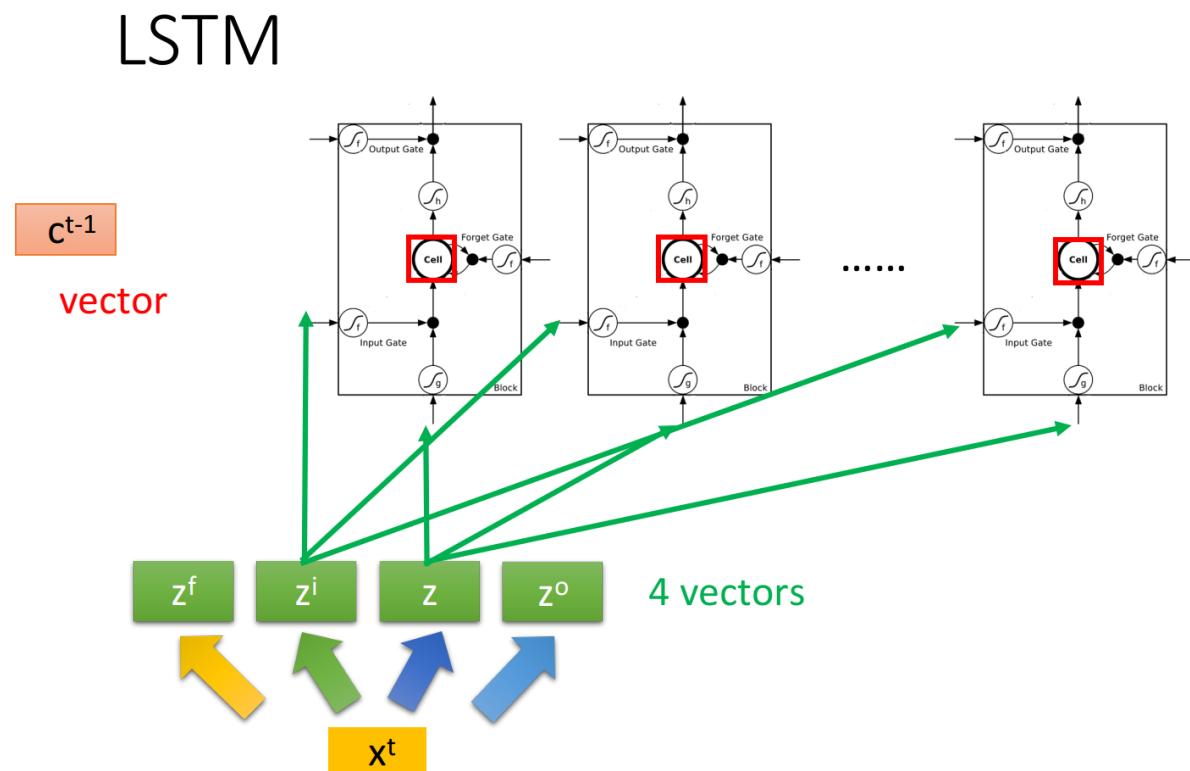
那我们知道一个 memory cell 就是长这样，那现在 input 分别是 z, z^i, z^f, z^o ，那注意一下这 4 个 z 其实都是 vector，丢到 cell 里面的值，其实只是每一个 vector 的一个 dimension，因为每一个 cell 它们 input 的 dimension 都是不一样的，所以它们 input 的值都会是不一样的

但是，所有的 cell 是可以共同一起被运算的。怎么一起共同被运算呢？我们说 z 要乘上 z^i ，要把 z^i 先通过 activation function，然后把它跟 z 相乘，所以我们就把 z^i 先通过 activation function，跟 z 相乘，这个乘是 element-wise 的相乘，好那这个 z^f 也要通过 forget gate 的 activation function， z^f 通过这个 activation function，它跟之前已经存在 cell 里面的值相乘，然后接下来呢，也要把这两个值加起来，你就是把 z^i 跟 z 相乘的值加上 z^f ，跟 c^{t-1} 相乘的值，把他们加起来。

那 output gate， z^o 通过 activation function，然后把这个 output 跟相加以后的结果，再相乘，最后就得到最后的 output 的 y ，这个时候相加以后的结果，也就是 memory 里面存的值，也就是 c^t ，那这个 process 呢，就反复地继续下去，在下一个时间点，input x^{t+1} ，然后你把 z 跟 input gate 相乘，你把 forget gate 跟存在 memory 里面的值相乘，然后再把这个值跟这个值加起来，再乘上 output gate 的值，然后得到下一个时间点的输出...

这个不是 LSTM 的最终型态，这个只是一个 simplified 的 version，真正的 LSTM 会怎么做它会把这个 hidden layer 的输出把它接进来，当作下一个时间点的 input，也就是说，下一个时间点操控这些 gate 的值，不是只看，那个时间点的 input x ，也看前一个时间点的 output h ，然后其实还不只这样，还会加一个东西，叫 peephole

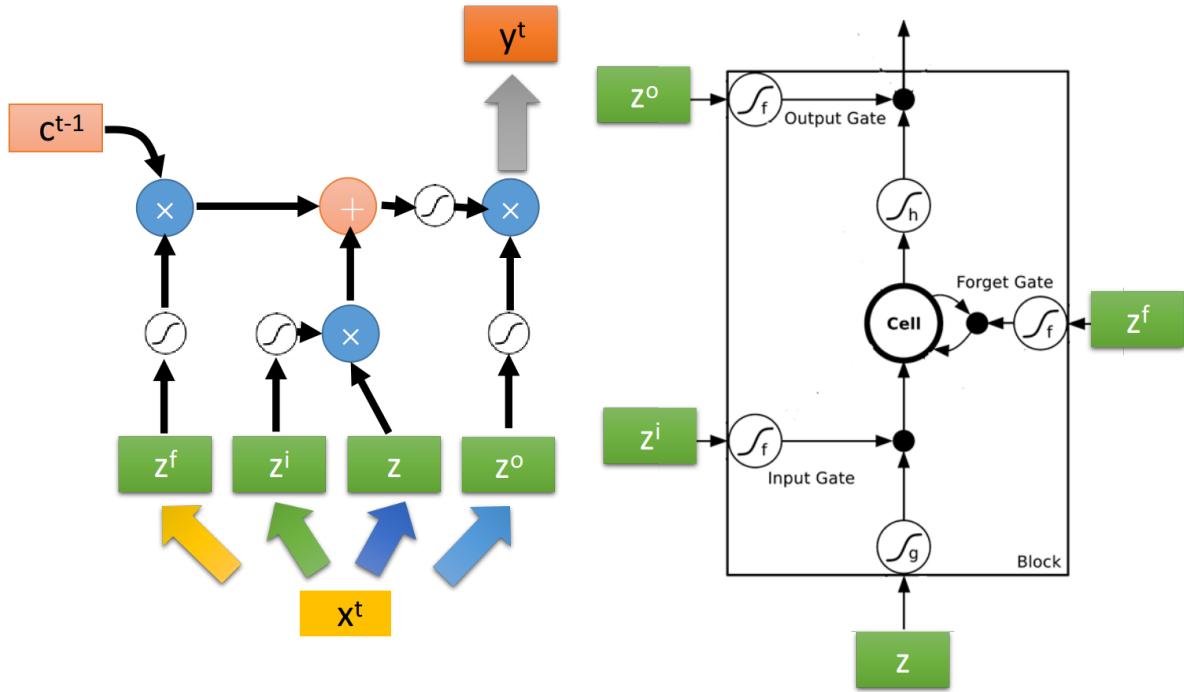
这个 peephole 就是把存在 memory cell 里面的值，也拉过来，所以在操纵 LSTM 的 4 个 gate 的时候，你是同时考虑了 x, h, c ，你把这 3 个 vector 并在一起，乘上 4 个不同的 transform，得到这 4 个不同的 vector，再去操控 LSTM



下图是单个LSTM的运算情景，其中LSTM的4个input分别是 z, z^i, z^f 和 z^o 的其中1维，每个LSTM的cell所得到的input都是各不相同的，但它们却是可以一起共同运算的，整个运算流程如下图左侧所示：

$f(z^f)$ 与上一个时间点的cell值 c^{t-1} 相乘，并加到经过input gate的输入 $g(z) \cdot f(z^i)$ 上，得到这个时刻cell中的值 c^t ，最终再乘上output gate的信号 $f(z^o)$ ，得到输出 y^t

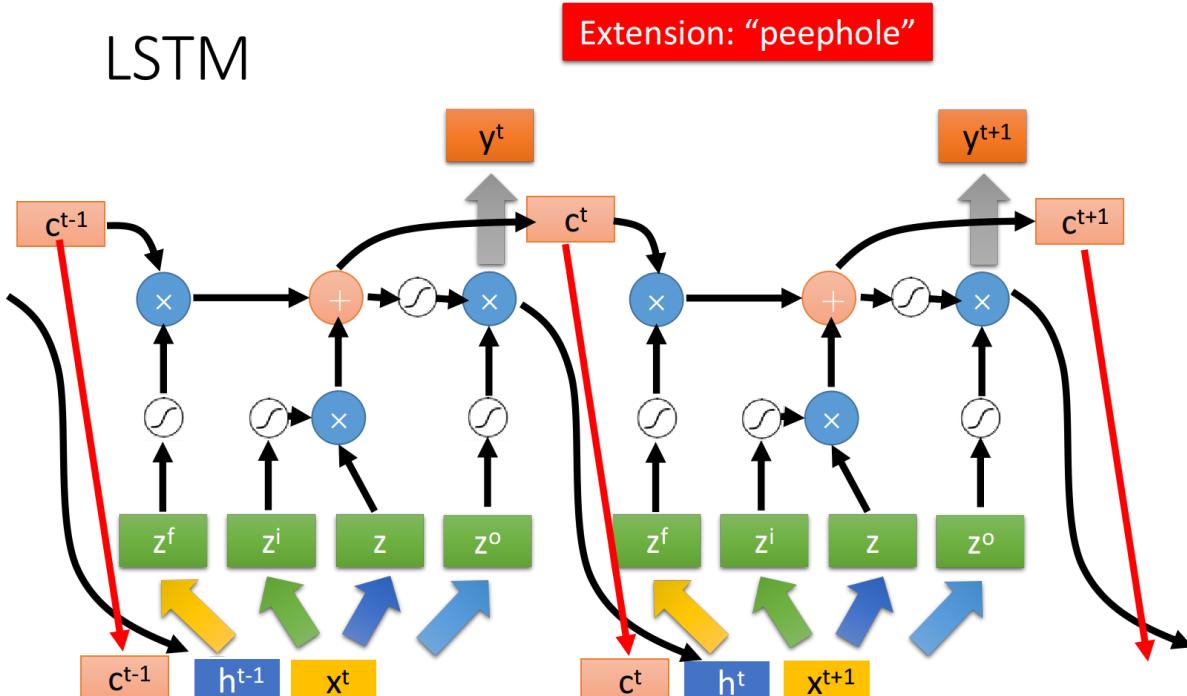
LSTM



上述的过程反复进行下去，就得到下图中各个时间点上，LSTM值的变化情况，其中与上面的描述略有不同的是，这里还需要把hidden layer的最终输出 y^t 以及当前cell的值 c^t 都连接到下一个时间点的输入上

因此在下一个时间点操控这些gate值，不只是看输入的 x^{t+1} ，还要看前一个时间点的输出 h^t 和cell值 c^t ，你需要把 x^{t+1} 、 h^t 和 c^t 这3个vector并在一起，乘上4个不同的转换矩阵，去得到LSTM的4个输入值 z^f 、 z^i 、 z^f 、 z^o ，再去对LSTM进行操控

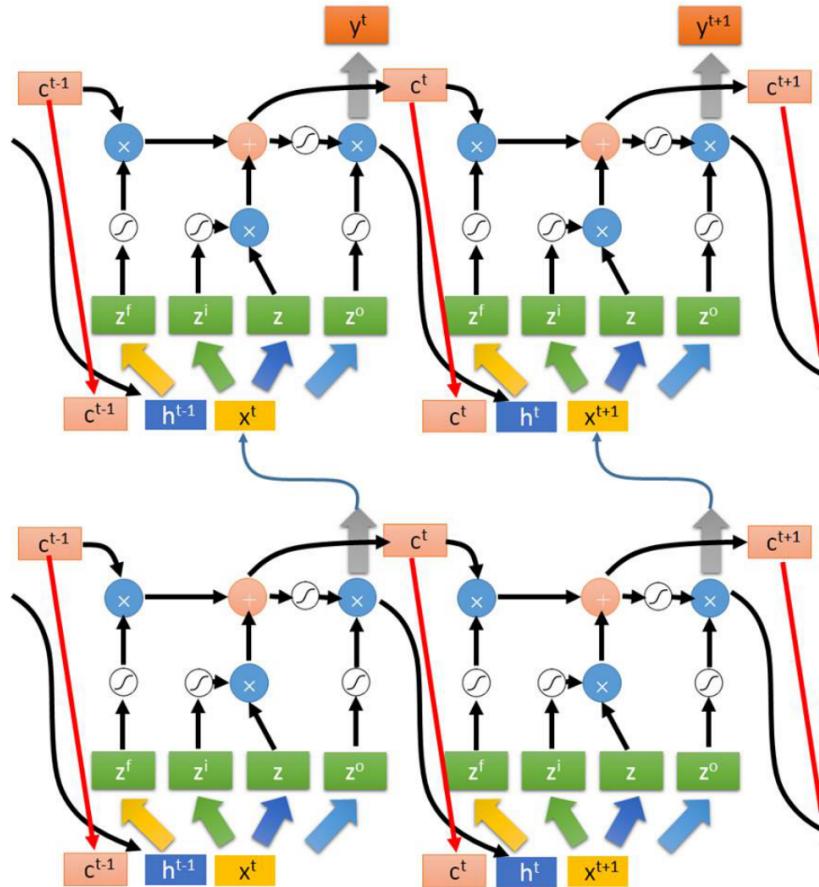
注意：下图是同一个LSTM在两个相邻时间点上的情况



上图是单个LSTM作为neuron的情况，事实上LSTM基本上都会叠多层，如下图所示，左边两个LSTM代表了两层叠加，右边两个则是它们在下一个时间点的状态

Multiple-layer LSTM

This is quite standard now.



Learning Target

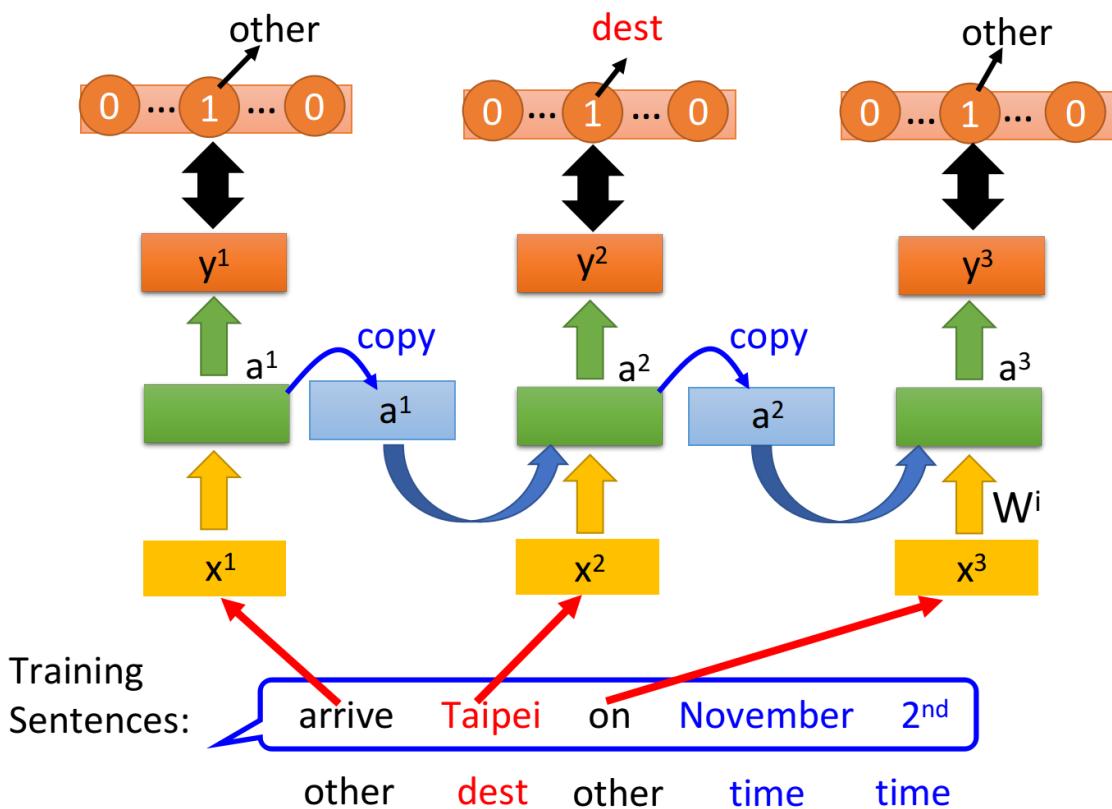
Loss Function

依旧是Slot Filling的例子，我们需要把model的输出 y^i 与映射到slot的reference vector求交叉熵，比如“Taipei”对应到的是“dest”这个slot，则reference vector在“dest”位置上值为1，其余维度值为0

RNN的output和reference vector的cross entropy之和就是损失函数，也是要minimize的对象

需要注意的是，word要依次输入model，比如“arrive”必须要在“Taipei”前输入，不能打乱语序

Learning Target

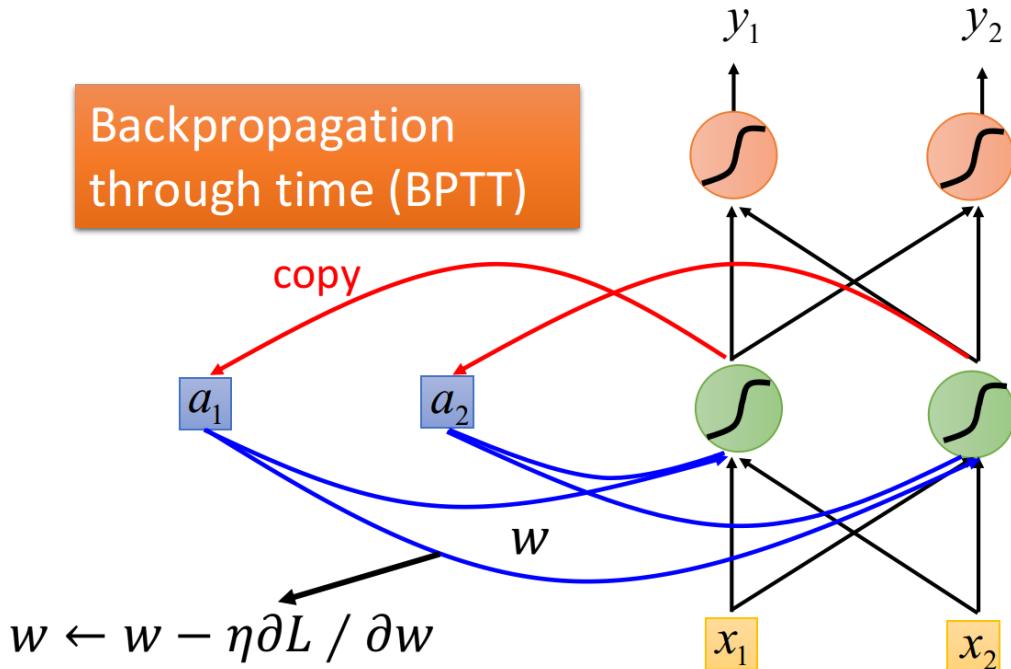


Training

有了损失函数后，训练其实也是用梯度下降法，为了计算方便，这里采取了反向传播(Backpropagation)的进阶版，Backpropagation through time，简称BPTT算法

BPTT算法与BP算法非常类似，只是多了一些时间维度上的信息，这里不做详细介绍

Learning



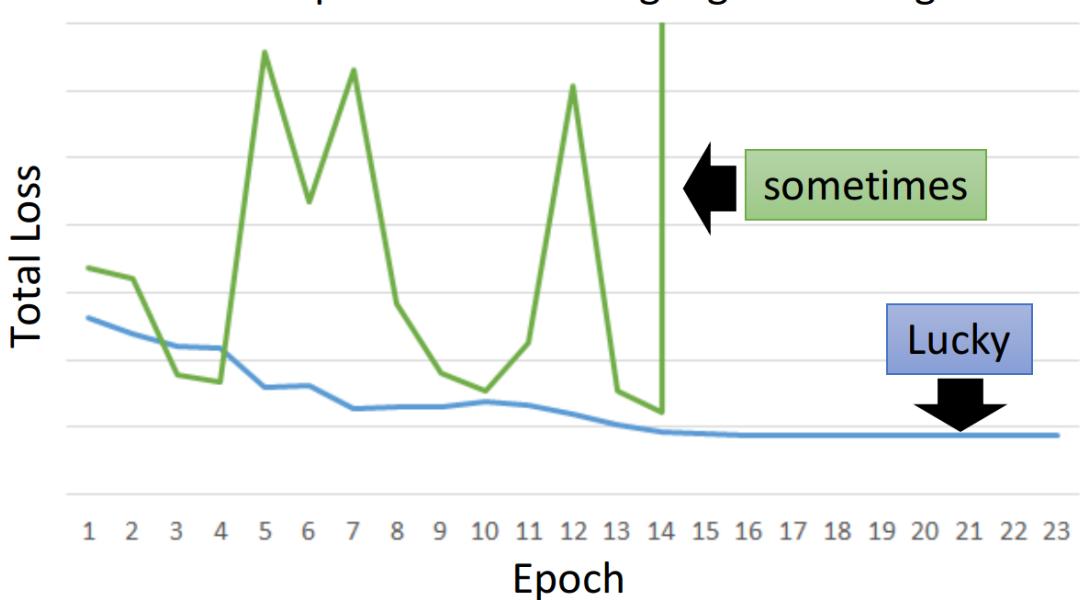
不幸的是，RNN的训练并没有那么容易

我们希望随着epoch的增加，参数的更新，loss应该要像下图的蓝色曲线一样慢慢下降，但在训练RNN的时候，你可能会遇到类似绿色曲线一样的学习曲线，loss剧烈抖动，并且会在某个时刻跳到无穷大，导致程序运行失败

Unfortunately

- RNN-based network is not always easy to learn

Real experiments on Language modeling



Error Surface

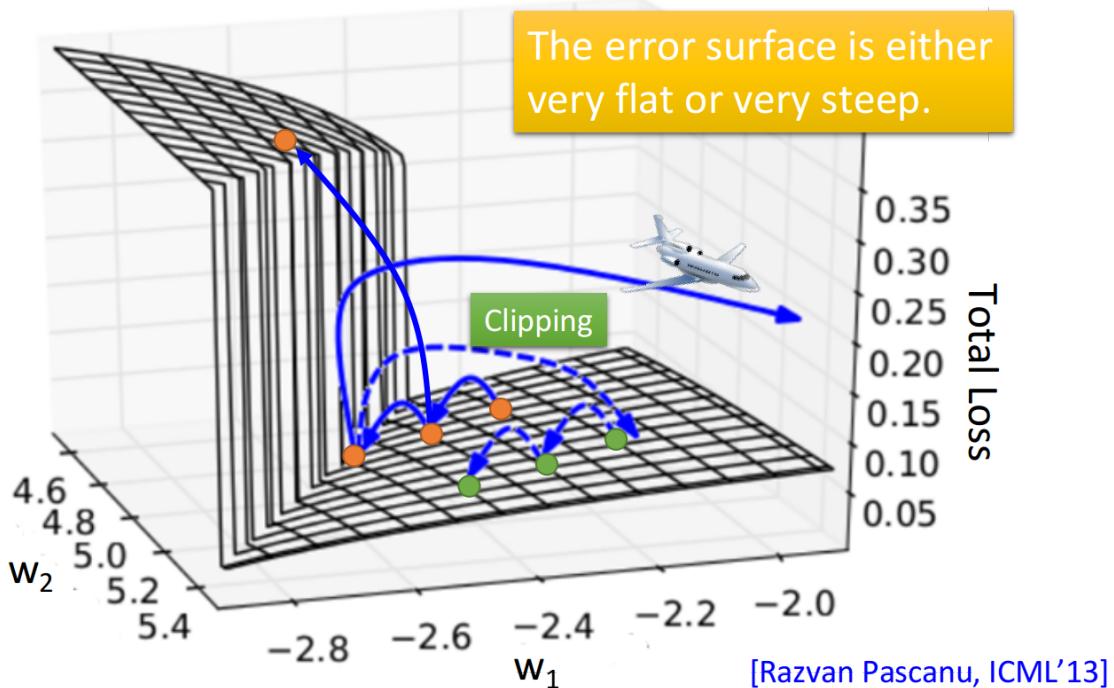
分析可知，RNN的error surface，即loss由于参数产生的变化，是非常陡峭崎岖的

下图中， z 轴代表loss， x 轴和 y 轴代表两个参数 w_1 和 w_2 ，可以看到loss在某些地方非常平坦，在某些地方又非常的陡峭

如果此时你的训练过程类似下图中从下往上的橙色的点，它先经过一块平坦的区域，又由于参数的细微变化跳上了悬崖，这就会导致loss上下抖动得非常剧烈

如果你的运气特别不好，一脚踩在悬崖上，由于之前一直处于平坦区域，gradient很小，你会把参数更新的步长(learning rate)调的比较大，而踩到悬崖上导致gradient突然变得很大，这会导致参数一下子被更新了一个大步伐，导致整个就飞出去了，这就是学习曲线突然跳到无穷大的原因

The error surface is rough.



想要解决这个问题，就要采用Clipping方法，当gradient即将大于某个threshold的时候，就让它停止增长，比如当gradient大于15的时候就直接让它等于15

为什么RNN会有这种奇特的特性呢？下图给出了一个直观的解释：

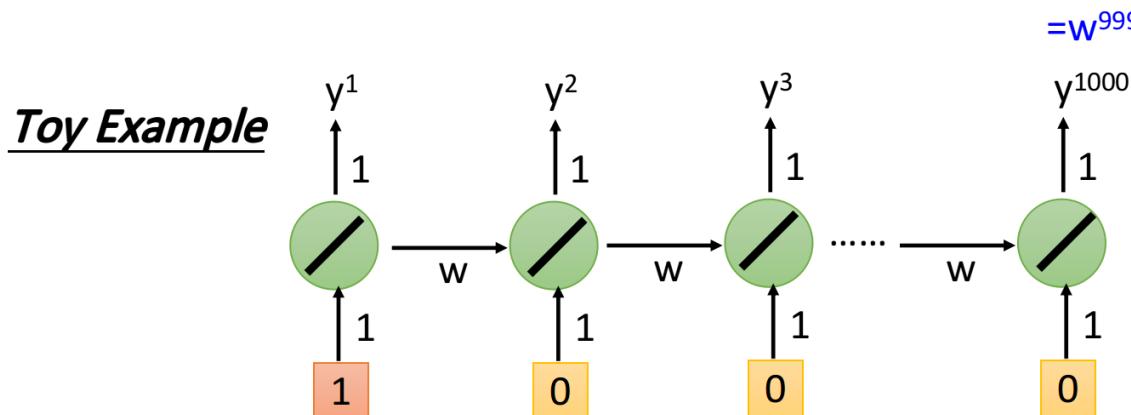
假设RNN只含1个neuron，它是linear的，input和output的weight都是1，没有bias，从当前时刻的memory值接到下一时刻的input的weight是 w ，按照时间点顺序输入 $[1, 0, 0, 0, \dots, 0]$

当第1个时间点输入1的时候，在第1000个时间点，RNN输出的 $y^{1000} = w^{999}$ ，想要知道参数 w 的梯度，只需要改变 w 的值，观察对RNN的输出有多大的影响即可：

- 当 w 从 $1 \rightarrow 1.01$ ，得到的 y^{1000} 就从1变到了20000，这表示 w 的梯度很大，需要调低学习率
- 当 w 从 $0.99 \rightarrow 0.01$ ，则 y^{1000} 几乎没有变化，这表示 w 的梯度很小，需要调高学习率
- 从中可以看出gradient时大时小，error surface很崎岖，尤其是在 $w = 1$ 的周围，gradient几乎是突变的，这让我们很难去调整learning rate

Why?

$w = 1 \rightarrow y^{1000} = 1$	$w = 1.01 \rightarrow y^{1000} \approx 20000$	Large $\partial L / \partial w$	Small Learning rate?
$w = 0.99 \rightarrow y^{1000} \approx 0$	$w = 0.01 \rightarrow y^{1000} \approx 0$	small $\partial L / \partial w$	Large Learning rate?



因此我们可以解释，RNN 会不好训练的原因，并不是来自于 activation function。而是来自于它有 time sequence，同样的 weight，在不同的时间点被反复的，不断的被使用。

从memory接到neuron输入的参数 w ，在不同的时间点被反复使用， w 的变化有时候可能对RNN的输出没有影响，而一旦产生影响，经过长时间的不断累积，该影响就会被放得无限大，因此RNN经常会遇到这两个问题：

- 梯度消失(gradient vanishing)，一直在梯度平缓的地方停滞不前
- 梯度爆炸(gradient explode)，梯度的更新步伐迈得太大导致直接飞出有效区间

Help Techniques

有什么技巧可以帮我们解决这个问题呢？LSTM就是最广泛使用的技巧，它会把error surface上那些比较平坦的地方拿掉，从而解决梯度消失(gradient vanishing)的问题，但它无法处理梯度崎岖的部分，因而也就无法解决梯度爆炸的问题(gradient explode)

但由于做LSTM的时候，大部分地方的梯度变化都很剧烈，因此训练时可以放心地把learning rate设的小一些

Q: 为什么要把RNN换成LSTM？

A: LSTM可以解决梯度消失的问题

Q: 为什么LSTM能够解决梯度消失的问题？

A: RNN和LSTM对memory的处理其实是不一样的：

- 在RNN中，每个新的时间点，memory里的旧值都会被新值所覆盖
- 在LSTM中，每个新的时间点，memory里的值会乘上 $f(g_f)$ 与新值相加

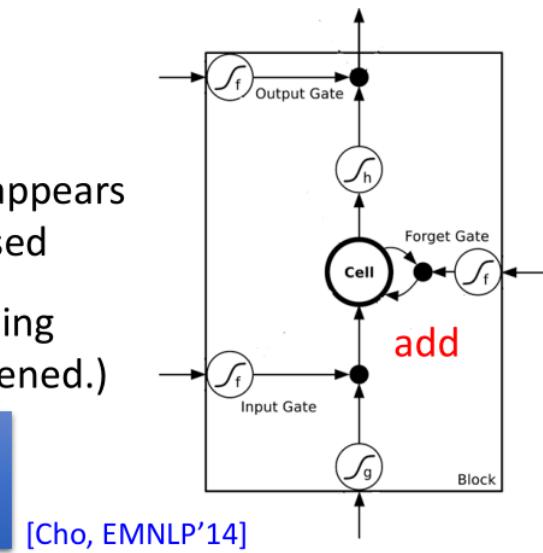
对RNN来说， w 对memory的影响每次都会被清除，而对LSTM来说，除非forget gate被打开，否则 w 对memory的影响就不会被清除，而是一直累加保留，因此它不会有梯度消失的问题

那你可能会想说，现在有 forget gate 啊，事实上 LSTM 在 97 年就被 proposed 了，LSTM 第一个版本就是为了解决 gradient vanishing 的问题，所以它是没有 forget gate 的，forget gate 是后来才加上去的。那甚至现在有一个传言是，你在训练 LSTM 时，不要给 forget gate 特别大的 bias，你要确保 forget gate 在多数的情况下是开启的，在多数情况下都不要忘记

• Long Short-term Memory (LSTM)

- Can deal with gradient vanishing (not gradient explode)
 - Memory and input are added
 - The influence never disappears unless forget gate is closed
 - No Gradient vanishing (If forget gate is opened.)

Gated Recurrent Unit (GRU): simpler than LSTM



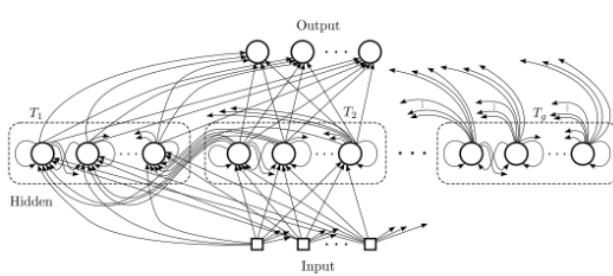
[Cho, EMNLP'14]

另一个版本GRU (Gated Recurrent Unit)，只有两个gate，需要的参数量比LSTM少，鲁棒性比LSTM好，performance与LSTM差不多，不容易过拟合，它的基本精神是旧的不去，新的不来，GRU会把input gate和forget gate连起来，当forget gate把memory里的值清空时，input gate才会打开，再放入新的值

当 input gate 被打开的时候，forget gate 就会被自动的关闭，就会自动忘记存在 memory 里面的值。当 forget gate 没有要忘记值，input gate 就会被关起来，也就是你要把存在 memory 里面的值清掉，才可以把新的值放进来。

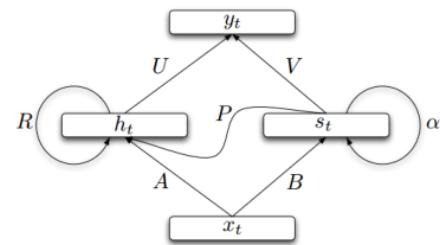
此外，还有很多技术可以用来处理梯度消失的问题，比如Clockwise RNN、SCRN等

Clockwise RNN



[Jan Koutnik, JMLR'14]

Structurally Constrained Recurrent Network (SCRN)



[Tomas Mikolov, ICLR'15]

Vanilla RNN Initialized with Identity matrix + ReLU activation function [Quoc V. Le, arXiv'15]

➤ Outperform or be comparable with LSTM in 4 different tasks

More Applications

在Slot Filling中，我们输入一个word vector输出它的label，除此之外RNN还可以做更复杂的事情

Sentiment Analysis

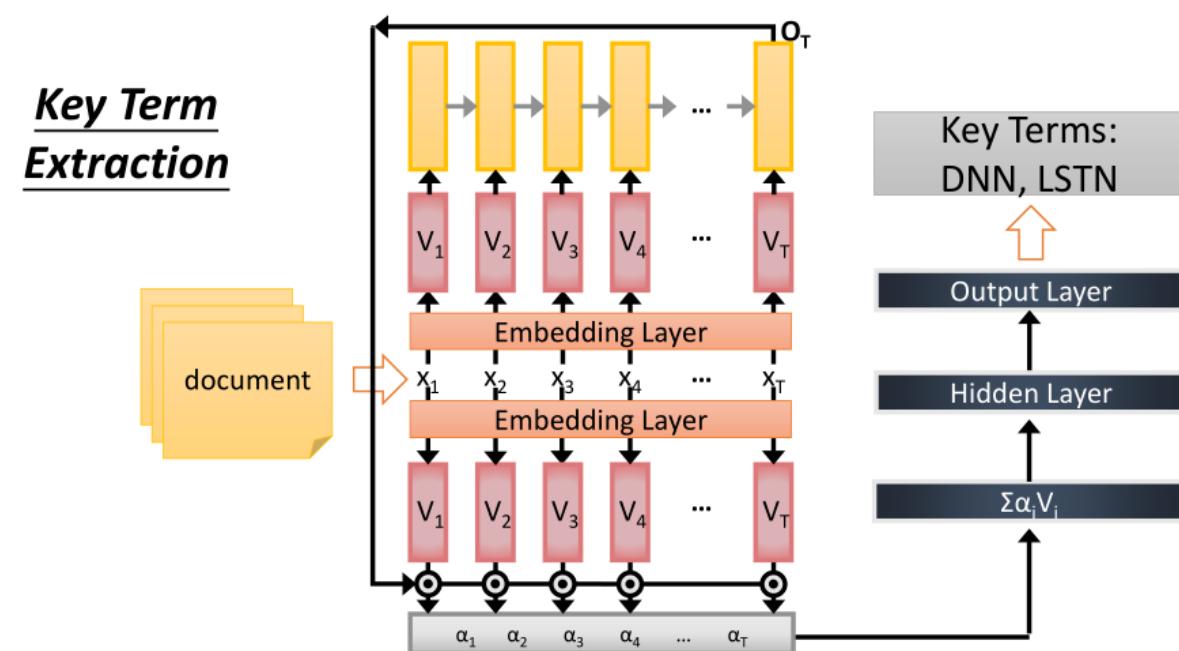
Many to one: Input is a vector sequence, but output is only one vector

语义情绪分析，我们可以把某影片相关的文章爬下来，并分析其正面情绪or负面情绪

RNN的输入是字符序列，在不同时间点输入不同的字符，并在最后一个时间点把hidden layer拿出来，再经过一系列转换，可以得到该文章的语义情绪的prediction

Key term Extraction

关键词分析，RNN可以分析一篇文章并提取出其中的关键词，这里需要把含有关键词标签的文章作为RNN的训练数据



Speech Recognition

Many to Many (Output is shorter): Both input and output are both sequences, but the output is shorter.

以语音识别为例，输入是一段声音信号，每隔一小段时间就用1个vector来表示，因此输入为vector sequence，而输出则是character sequence

如果依旧使用Slot Filling的方法，只能做到每个vector对应1个输出的character，识别结果就像是下图中的“好好好棒棒棒棒棒棒”，但这不是我们想要的，可以使用Trimming的技术把重复内容消去，剩下“好棒”

Output: “好棒” (character sequence)

Problem?

Why can't it be
“好棒棒”

Input:

好 好 好 棒 棒 棒 棒 棒

Trimming

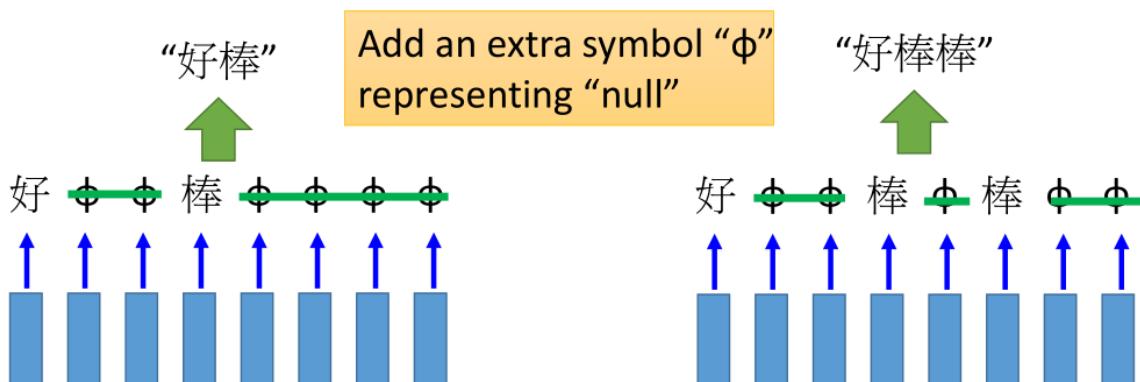
(vector
sequence)



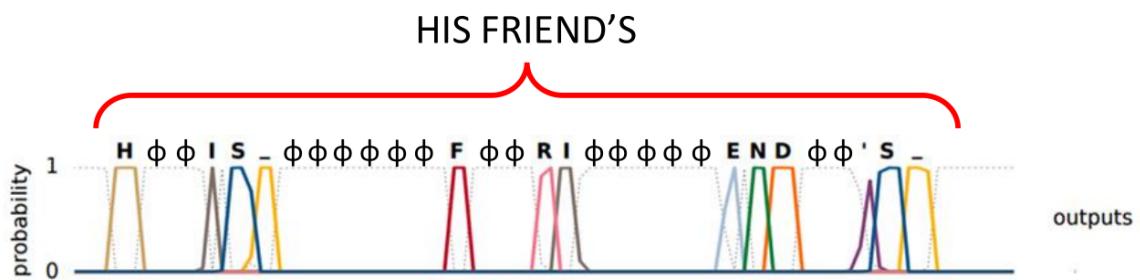
但“好棒”和“好棒棒”实际上是不一样的，如何区分呢？

需要用到CTC算法，它的基本思想是，输出不只是字符，还要填充NULL，输出的时候去掉NULL就可以得到叠字的效果

- Connectionist Temporal Classification (CTC) [Alex Graves, ICML'06][Alex Graves, ICML'14][Haşim Sak, Interspeech'15][Jie Li, Interspeech'15][Andrew Senior, ASRU'15]



下图是CTC的示例，RNN的输出就是英文字母+NULL，Google的语音识别系统据说就是用CTC实现的



Graves, Alex, and Navdeep Jaitly. "Towards end-to-end speech recognition with recurrent neural networks." *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. 2014.

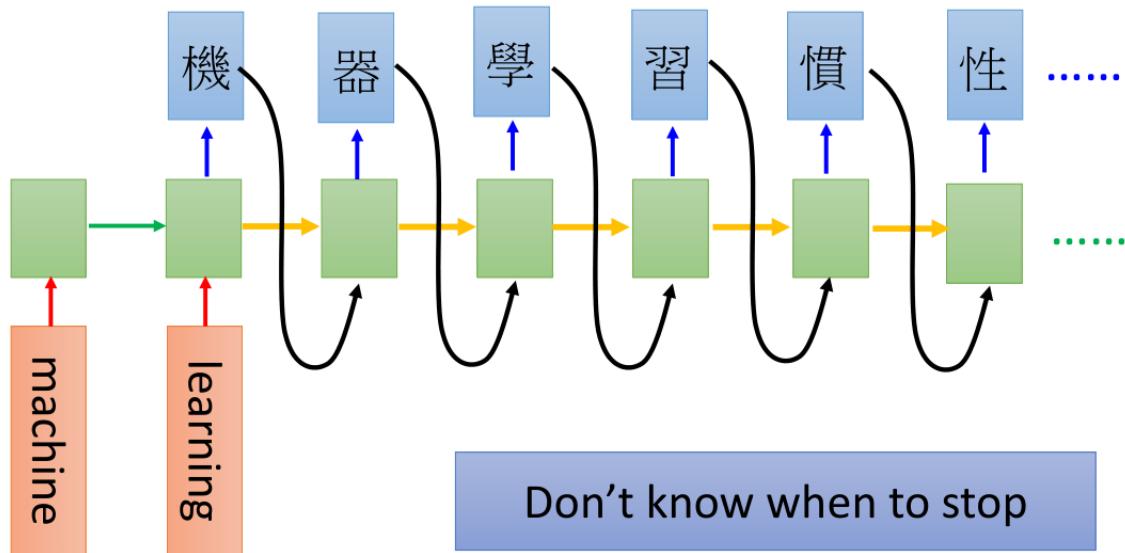
Sequence to Sequence Learning

Many to Many (No Limitation): Both input and output are both sequences with different lengths.

在CTC中，input比较长，output比较短；而在Seq2Seq中，并不确定谁长谁短

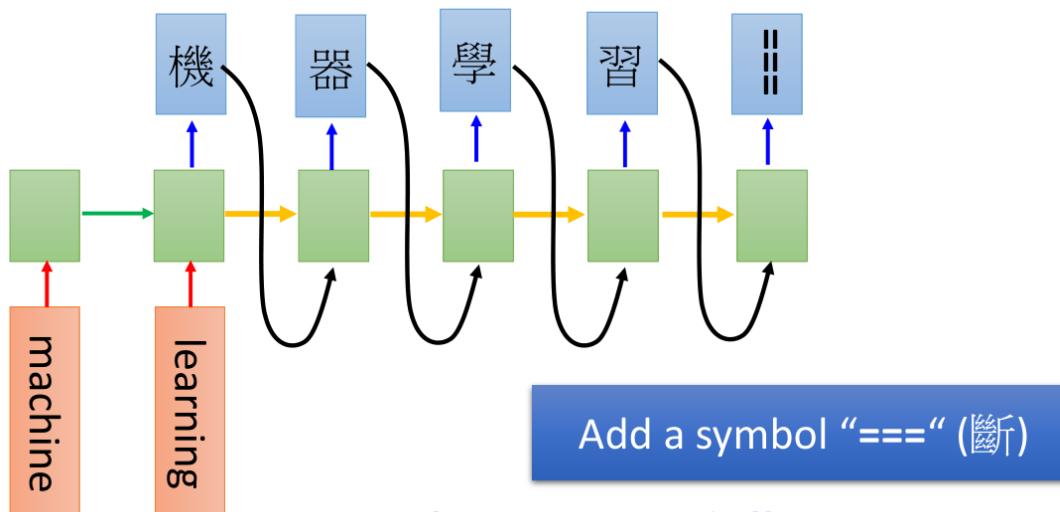
比如现在要做机器翻译，将英文的word sequence翻译成中文的character sequence

假设在两个时间点分别输入“machine”和“learning”，则在最后1个时间点memory就存了整个句子的信息，接下来让RNN输出，就会得到“机”，把“机”当做input，并读取memory里的值，就会输出“器”，依次类推，这个RNN甚至会一直输出，不知道什么时候会停止



怎样才能让机器停止输出呢？

可以多加一个叫做“断”的symbol “==”，当输出到这个symbol时，机器就停止输出

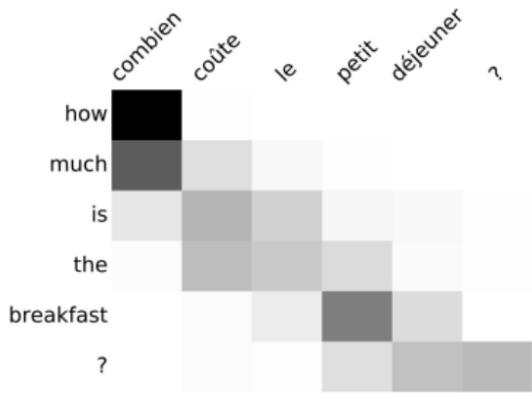


[Ilya Sutskever, NIPS'14][Dzmitry Bahdanau, arXiv'15]

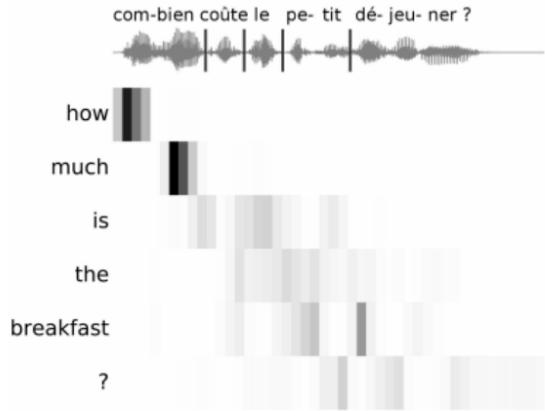
具体的处理技巧这里不再详述

Machine Translation

一种语言的声音讯号翻译成另一种语言的文字，很神奇的可以work



(a) Machine translation alignment



(b) Speech translation alignment

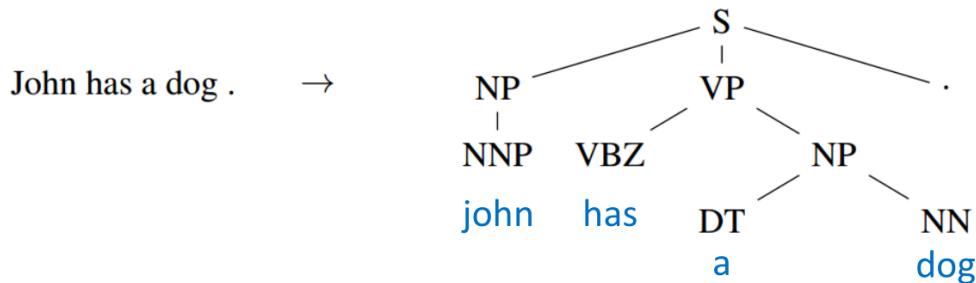
Figure 1: Alignments performed by the attention model during training

Syntactic Parsing

Sequence-to-sequence还可以用在句法解析上，让机器看一个句子，它可以自动生成Syntactic parsing tree

过去，你可能要用 structure learning 的技术才能够解这一个问题，但现在有了 sequence to sequence 的技术以后，只要把这个树形图，描述成一个 sequence，直接 learn 一个 sequence to sequence 的 model，output 直接是这个 Syntactic 的 parsing tree

- Syntactic parsing



John has a dog . → (S (NP NNP)_{NP} (VP VBZ (NP DT NN)_{NP})_{VP} .)_S

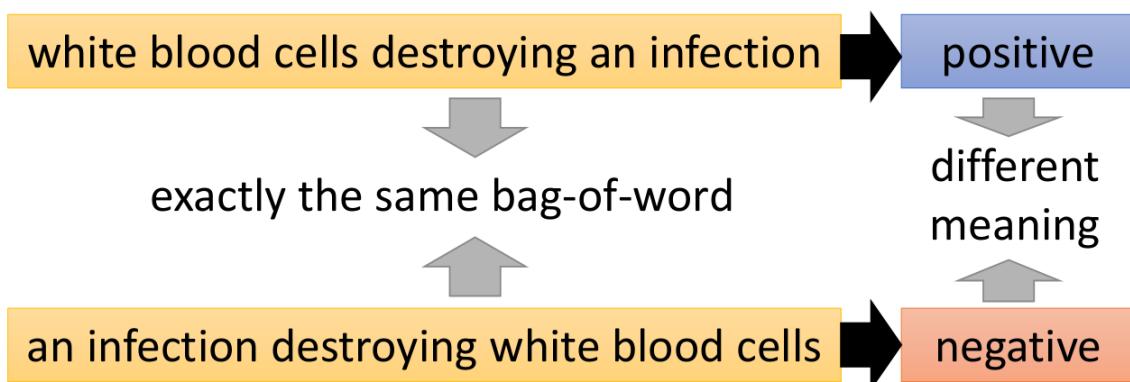
Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, Geoffrey Hinton,
Grammar as a Foreign Language, NIPS 2015

Sequence-to-sequence for Auto-encoder - Text

如果用bag-of-word来表示一篇文章，就很容易丢失词语之间的联系，丢失语序上的信息

比如“白血球消灭了感染病”和“感染病消灭了白血球”，两者bag-of-word是相同的，但语义却是完全相反的

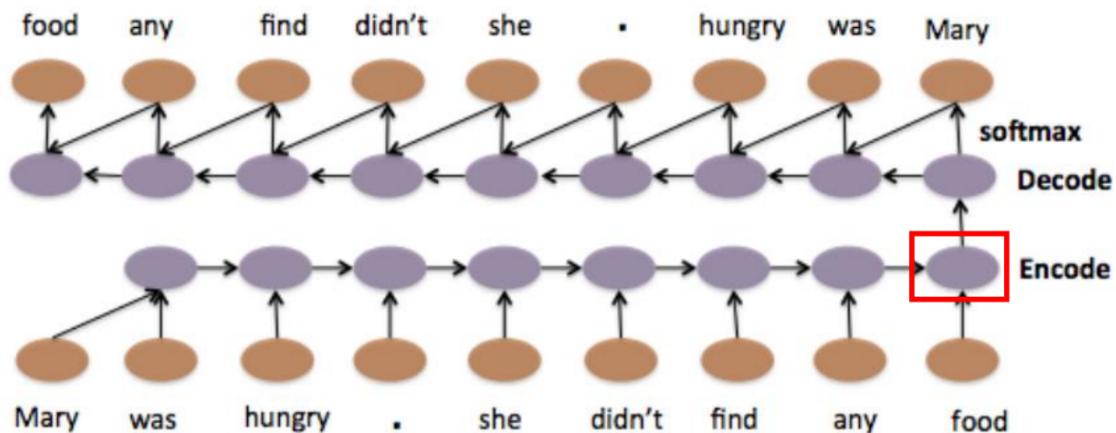
- To understand the meaning of a word sequence, the order of the words can not be ignored.



这里就可以使用Sequence-to-sequence Auto-encoder，在考虑了语序的情况下，把文章编码成vector，只需要把RNN当做编码器和解码器即可

我们输入word sequence，通过RNN变成embedded vector，再通过另一个RNN解压回去，如果能够得到一模一样的句子，则压缩后的vector就代表了这篇文章中最重要的信息

如果是用 Seq2Seq auto encoder，input 跟 output 都是同一个句子。如果你用 skip-thought 的话，output target会是下一个句子。如果是用 Seq2Seq auto encoder，通常你得到的 code 比较容易表达文法的意思。如果你要得到语意的意思，用 skip-thought 可能会得到比较好结果。



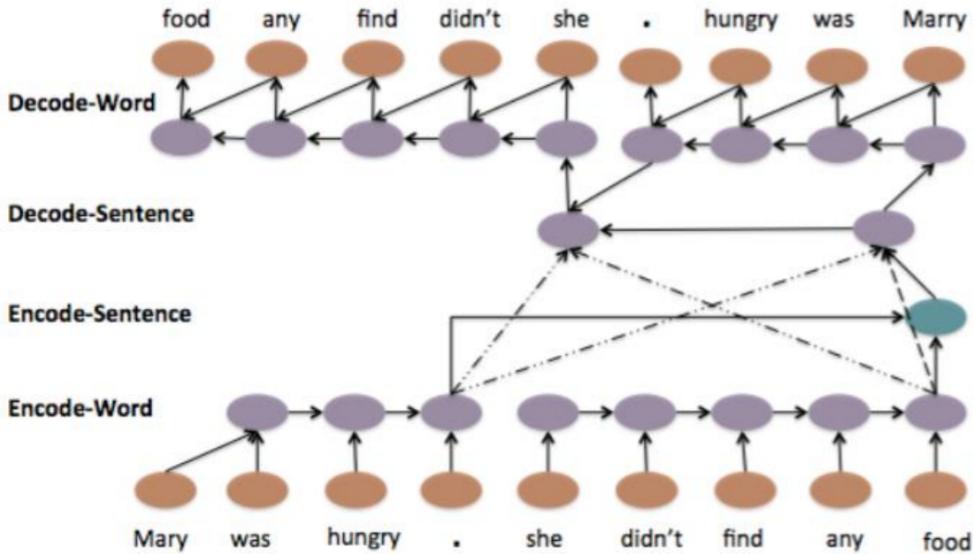
Li, Jiwei, Minh-Thang Luong, and Dan Jurafsky. "A hierarchical neural autoencoder for paragraphs and documents." *arXiv preprint arXiv:1506.01057*(2015).

这个结构甚至可以是 Hierarchy 的，你可以每一个句子都先得到一个 vector
再把这些 vector 加起来，变成一个整个document high level 的 vector

再用这个 document high level 的 vector去产生一串 sentence 的 vector

再根据每一个 sentence vector去解回 word sequence

所以这是一个 4 层的 LSTM，你从 word 变成 sentence sequence，再变成 document level 的东西，再解回 sentence sequence，再解回 word sequence



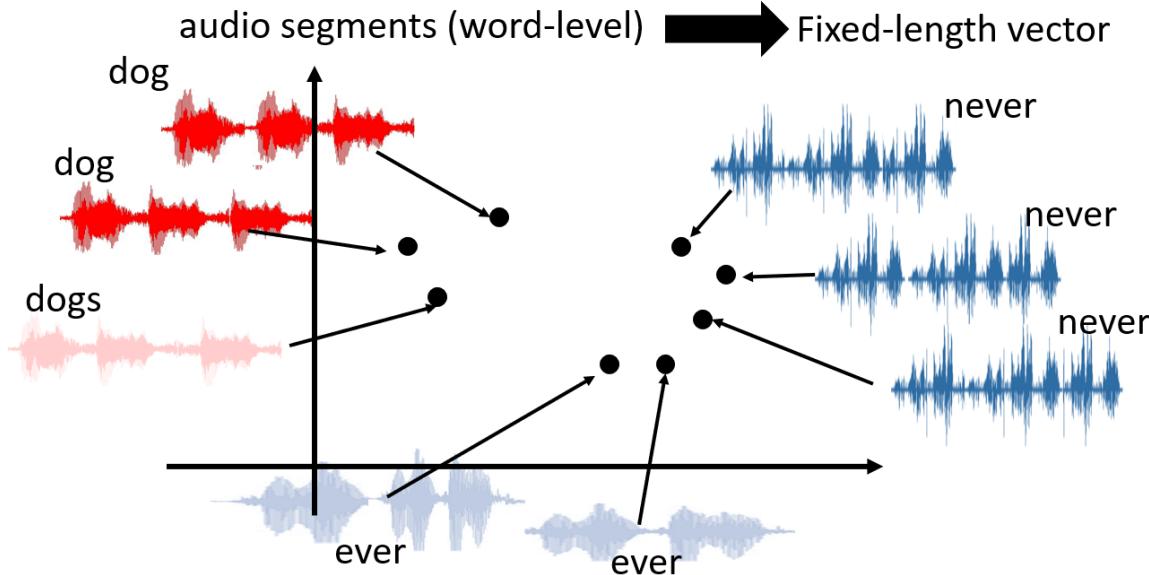
Li, Jiwei, Minh-Thang Luong, and Dan Jurafsky. "A hierarchical neural autoencoder for paragraphs and documents." *arXiv preprint arXiv:1506.01057*(2015).

Sequence-to-sequence for Auto-encoder - Speech

Sequence-to-sequence Auto-encoder还可以用在语音处理上，它可以把一段 audio segment 变成一段 fixed length 的 vector

比如说这边有一堆声音讯号，它们长长短短的都不一样，你把它们变成 vector 的话，可能 dog/dogs 的 vector 比较接近，可能 never/ever 的 vector 是比较接近的

- Dimension reduction for a sequence with variable length



Yu-An Chung, Chao-Chung Wu, Chia-Hao Shen,
Hung-Yi Lee, Lin-Shan Lee, Audio Word2Vec: Unsupervised Learning of Audio Segment Representations using Sequence-to-sequence Autoencoder, Interspeech 2016

这种方法可以把声音信号都转化为低维的vector，并通过计算相似度来做语音搜索，不需要做语音识别，直接比对声音讯号的相似度即可。

Audio archive divided into variable-length audio segments

Off-line



Audio
Segment to
Vector

Similarity

On-line

Search Result

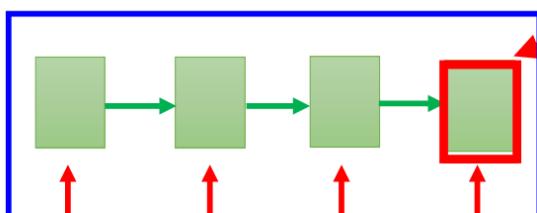
如何把audio segment变成vector呢? 先把声音信号转化成声学特征向量(acoustic features), 再通过RNN编码, 最后一个时间点存在memory里的值就代表了整个声音信号的信息



The values in the memory
represent the whole audio
segment

The vector we want

How to train RNN Encoder?

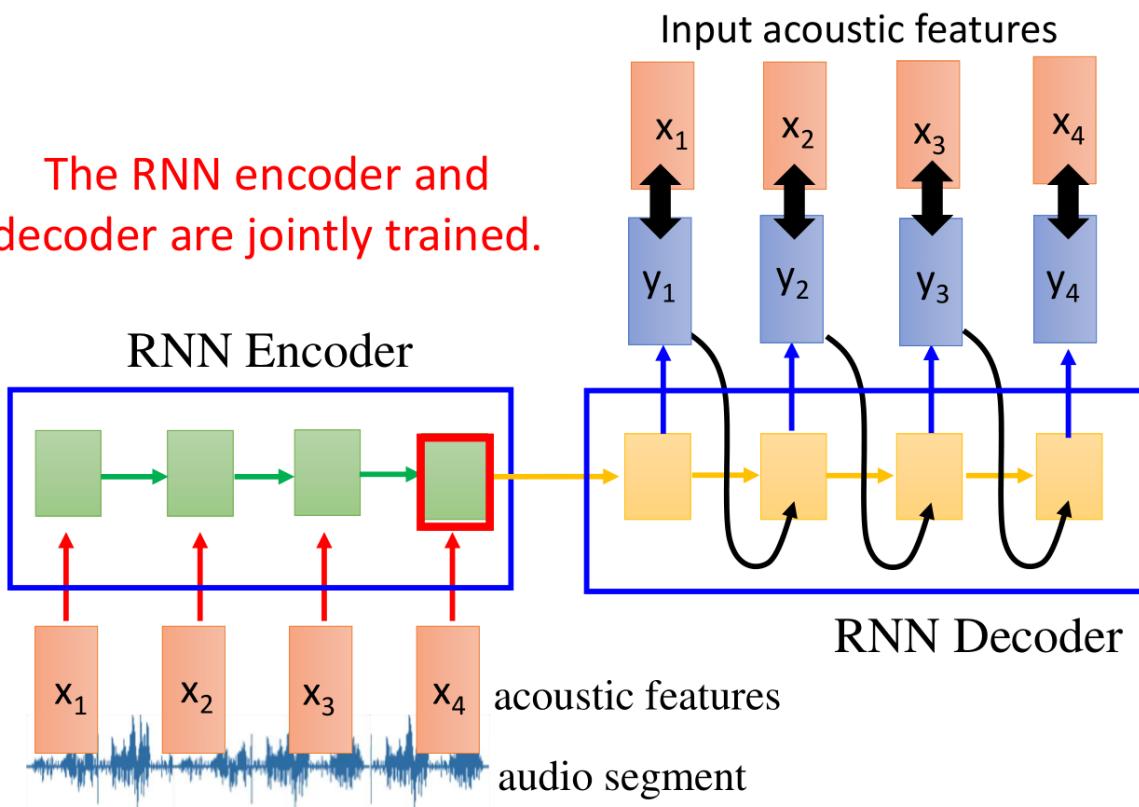


acoustic features

audio segment

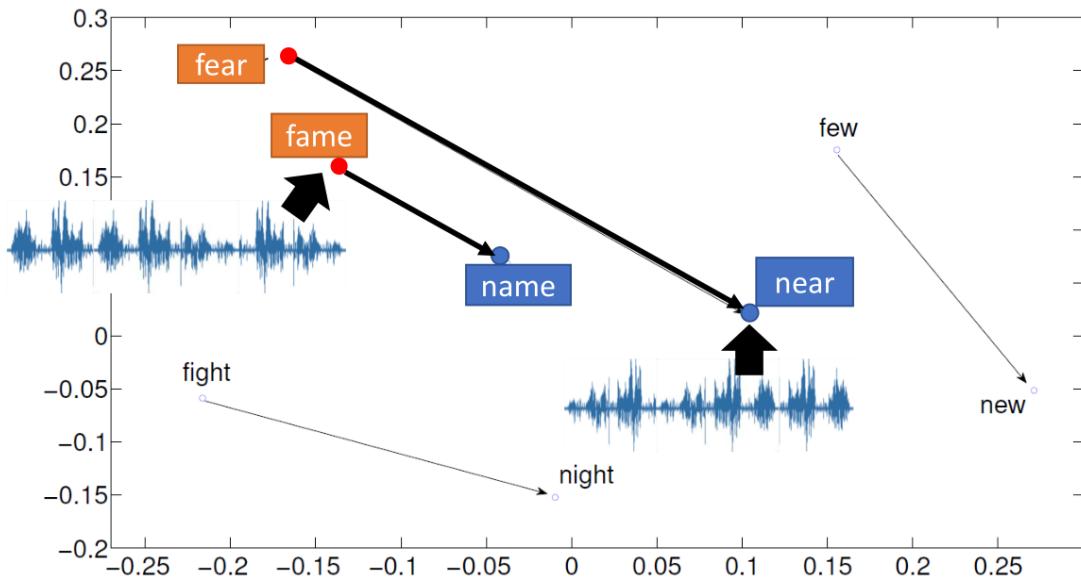
为了能够对该神经网络训练, 还需要一个RNN作为解码器, 得到还原后的 y_i , 使之与 x_i 的差距最小

The RNN encoder and decoder are jointly trained.



最后得到vector的可视化

- Visualizing embedding vectors of the words

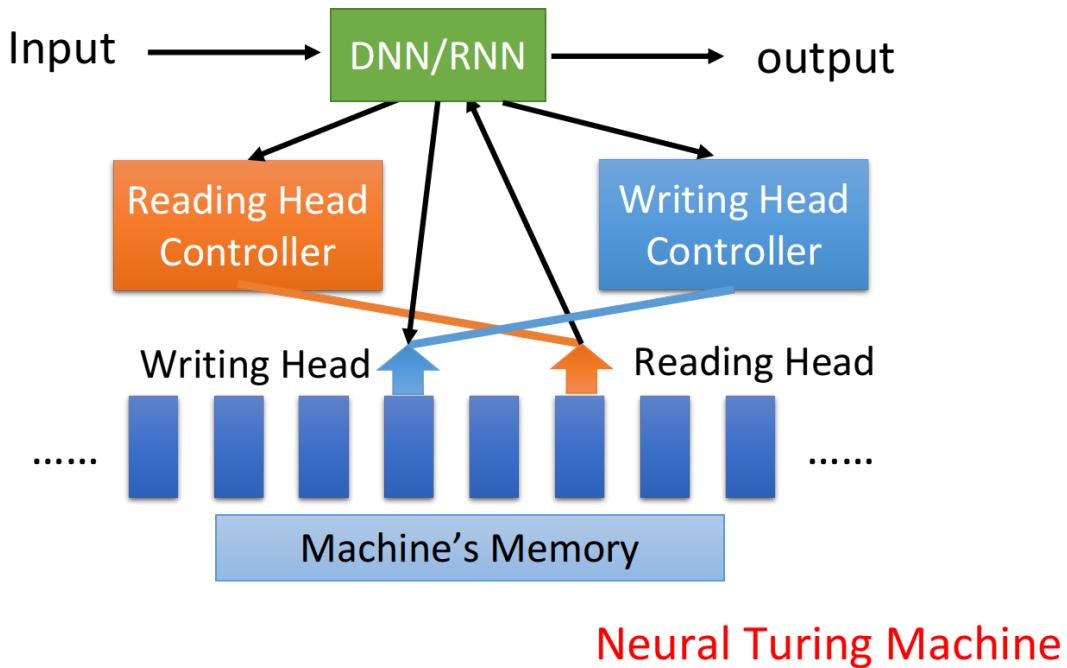


Attention-based Model

除了RNN之外，Attention-based Model也用到了memory的思想

机器会有自己的记忆池，神经网络通过操控读写头去读或者写指定位置的信息，这个过程跟图灵机很像，因此也被称为neural turing machine

Attention-based Model v2



这种方法通常用在阅读理解上，让机器读一篇文章，再把每句话的语义都存到不同的vector中，接下来让用户向机器提问，神经网络就会去调用读写头的中央处理器，取出memory中与查询语句相关的信息，综合处理之后，可以给出正确的回答

Semi-supervised Learning

Semi-supervised Learning

Introduction

Supervised learning: $(x^r, \hat{y}^r)_{r=1}^R$

- training data中，共有R笔data，每一笔data都有input x^r 和对应的output \hat{y}^r

Semi-supervised Learning: $\{(x^r, \hat{y}^r)\}_{r=1}^R + \{x^u\}_{u=R}^{R+U}$

- training data中，部分data没有标签，只有input x^u ，没有output
- 通常遇到的场景是，无标签的数据量远大于有标签的数据量，即 $U >> R$
- semi-supervised learning分为以下两种情况：

- Transductive Learning: unlabeled data is the testing data

即，把testing data当做无标签的training data使用，适用于事先已经知道testing data的情况（一些比赛的时候）

值得注意的是，这种方法使用的仅仅是testing data的feature，而不是label，因此不会出现直接对testing data做训练而产生cheating的效果

- Inductive Learning: unlabeled data is not the testing data

即，不把testing data的feature拿去给机器训练，适用于事先并不知道testing data的情况（更普遍的情况）

- 为什么要做semi-supervised learning？

实际上我们从来不缺data，只是缺有label的data，就像你可以拍很多照片，但它们一开始都是没有标签的

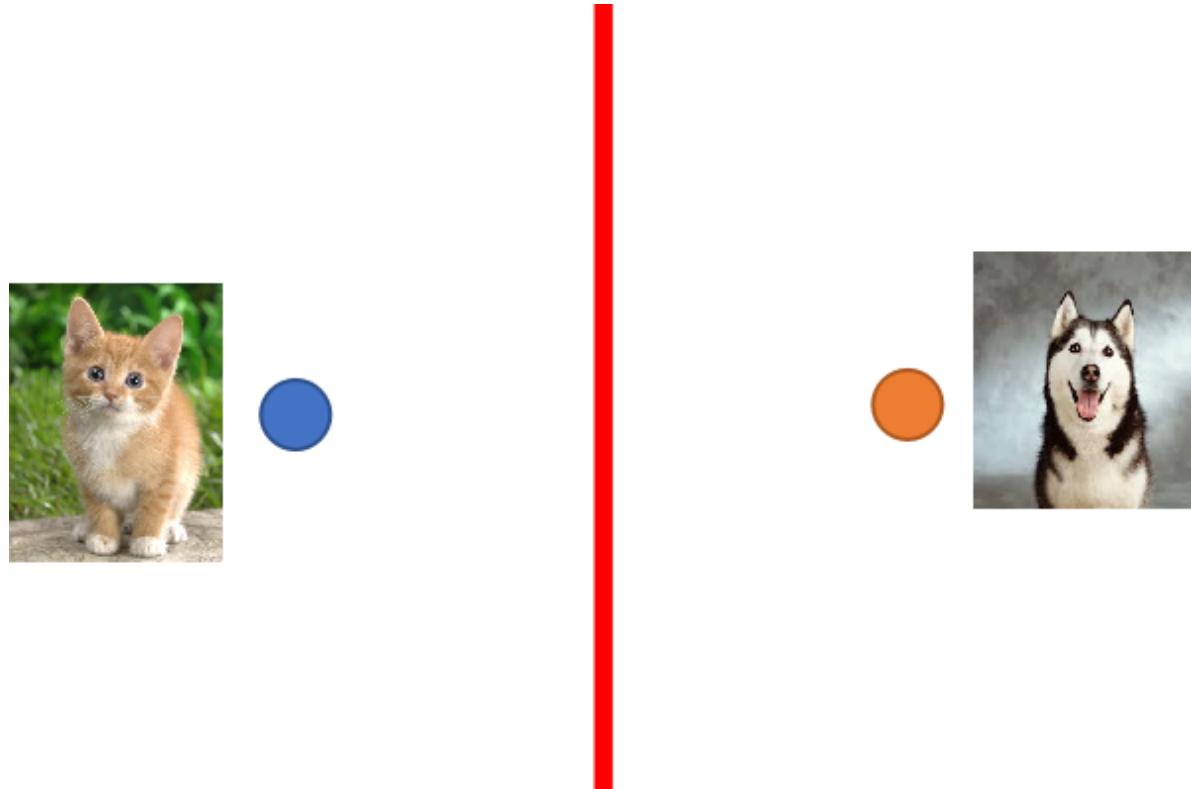
Why semi-supervised learning help?

为什么semi-supervised learning会有效呢？

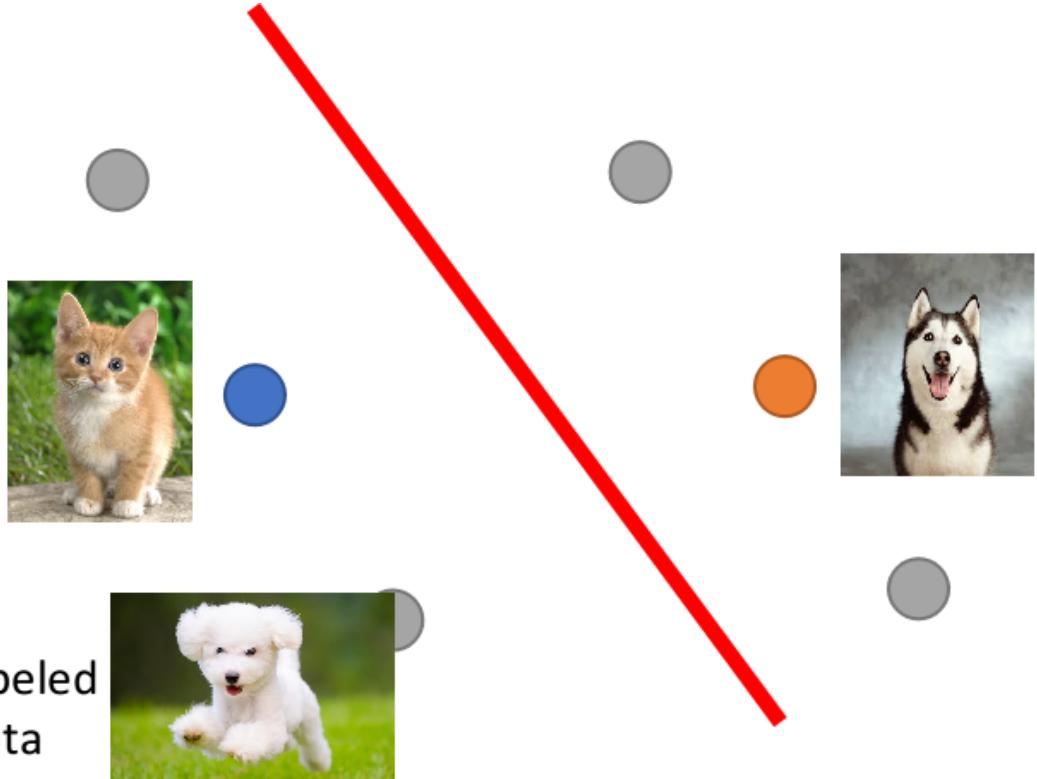
The distribution of the unlabeled data tell us something.

unlabeled data虽然只有input，但它的**分布**，却可以告诉我们一些事情

以下图为例，在只有labeled data的情况下，红线是二元分类的分界线



但当我们加入unlabeled data的时候，由于**特征分布**发生了变化，分界线也随之改变



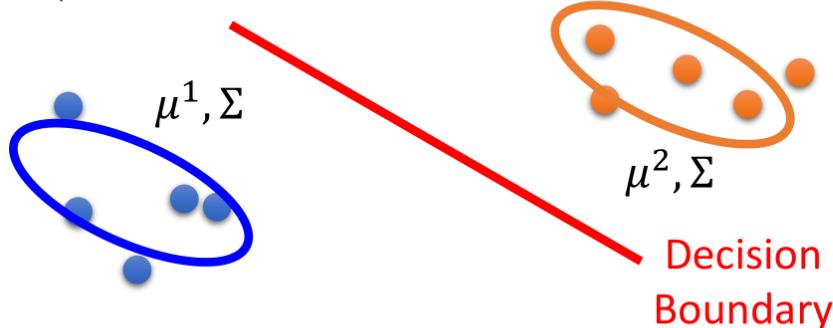
semi-supervised learning的使用往往伴随着假设，而该假设的合理与否，决定了结果的好坏程度；比如上图中的unlabeled data，它显然是一只狗，而特征分布却与猫被划分在了一起，很可能是由于这两张图片的背景都是绿色导致的，因此假设是否合理显得至关重要

Semi-supervised Learning for Generative Model

Supervised Generative Model

事实上，在监督学习中，我们已经讨论过概率生成模型了，假设class 1和class 2的分布分别为 $mean_1 = u^1, covariance_1 = \Sigma$ 、 $mean_2 = u^2, covariance_2 = \Sigma$ 的高斯分布，计算出Prior Probability后，再根据贝叶斯公式可以推得新生成的x所属的类别

- Given labelled training examples $x^r \in C_1, C_2$
 - looking for most likely prior probability $P(C_i)$ and class-dependent probability $P(x|C_i)$
 - $P(x|C_i)$ is a Gaussian parameterized by μ^i and Σ



With $P(C_1), P(C_2), \mu^1, \mu^2, \Sigma$

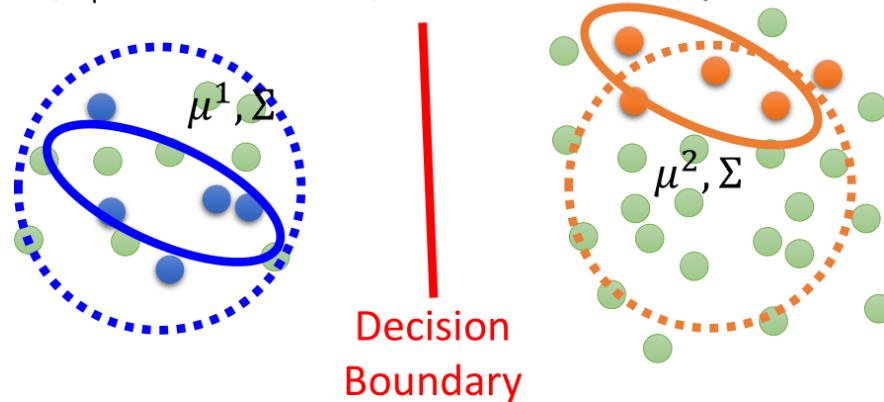
$$P(C_1|x) = \frac{P(x|C_1)P(C_1)}{P(x|C_1)P(C_1) + P(x|C_2)P(C_2)}$$

Semi-supervised Generative Model

如果在原先的数据下多了unlabeled data (下图中绿色的点) , 它就会影响最终的决定, 你会发现原先的 u, Σ 显然是不合理的, 新的 u, Σ 需要使得样本点的分布更接近下图虚线圆所标出的范围, 除此之外, 右侧的Prior Probability会给人一种比左侧大的感觉 (右侧样本点变多了)

此时, unlabeled data对 $P(C_1), P(C_2), u^1, u^2, \Sigma$ 都产生了一定程度的影响, 划分两个class的decision boundary也会随之发生变化

- Given labelled training examples $x^r \in C_1, C_2$
 - looking for most likely prior probability $P(C_i)$ and class-dependent probability $P(x|C_i)$
 - $P(x|C_i)$ is a Gaussian parameterized by μ^i and Σ



The unlabeled data x^u help re-estimate $P(C_1), P(C_2), \mu^1, \mu^2, \Sigma$

讲完了直观上的解释, 接下来进行具体推导(假设做二元分类):

- 先随机初始化一组参数: $\theta = \{P(C_1), P(C_2), u^1, u^2, \Sigma\}$
- Step 1: 利用初始model计算每一笔unlabeled data x^u 属于class 1的posterior probability $P_\theta(C_1|x^u)$
- Step 2: update model

如果不考虑unlabeled data, 则先验概率显然为属于class 1的样本点数 N_1 /总的样本点数 N , 即
 $P(C_1) = \frac{N_1}{N}$

而考虑unlabeled data时, 分子还要加上所有unlabeled data属于class 1的概率和, 此时它们被看作小数, 可以理解为按照概率一部分属于 C_1 , 一部分属于 C_2

$$P(C_1) = \frac{N_1 + \sum_{x^u} P(C_1|x^u)}{N}$$

同理, 对于均值, 原先的mean $u_1 = \frac{1}{N_1} \sum_{x^r \in C_1} x^r$ 加上根据概率对 x^u 求和再归一化的结果即可

$$u_1 = \frac{1}{N_1} \sum_{x^r \in C_1} x^r + \frac{1}{\sum_{x^u} P(C_1|x^u)} \sum_{x^u} P(C_1|x^u)x^u$$

剩余的参数同理, 接下来就有了一组新的参数 θ'

于是回到step 1->step 2->step 1循环

Semi-supervised Generative Model

The algorithm converges eventually, but the initialization influences the results.

- Initialization: $\theta = \{P(C_1), P(C_2), \mu^1, \mu^2, \Sigma\}$

- E** • Step 1: compute the posterior probability of unlabeled data

$$P_\theta(C_1|x^u) \quad \text{Depending on model } \theta$$

- M** • Step 2: update model

$$P(C_1) = \frac{N_1 + \sum_{x^u} P(C_1|x^u)}{N} \quad \begin{array}{l} N: \text{total number of examples} \\ N_1: \text{number of examples belonging to } C_1 \end{array}$$

$$\mu^1 = \frac{1}{N_1} \sum_{x^r \in C_1} x^r + \frac{1}{\sum_{x^u} P(C_1|x^u)} \sum_{x^u} P(C_1|x^u)x^u \quad \dots$$

Back to step 1

- 理论上该方法保证是可以收敛的，而一开始给 θ 的初始值会影响收敛的结果，类似gradient descent
- 上述的step 1就是EM algorithm里的E, step 2则是M

以上的推导基于的基本思想是，把unlabeled data x^u 看成是可以划分的，一部分属于 C_1 ，一部分属于 C_2 ，此时它的概率 $P_\theta(x^u) = P_\theta(x^u|C_1)P(C_1) + P_\theta(x^u|C_2)P(C_2)$ ，也就是 C_1 的先验概率乘上 C_1 这个class产生 x^u 的概率+ C_2 的先验概率乘上 C_2 这个class产生 x^u 的概率

实际上我们在利用极大似然函数更新参数的时候，就利用了该拆分的结果：

$$\log L(\theta) = \sum_{x^r} \log P_\theta(x^r, \hat{y}^r) + \sum_{x^u} \log P_\theta(x^u)$$

Why?

$$\theta = \{P(C_1), P(C_2), \mu^1, \mu^2, \Sigma\}$$

- Maximum likelihood with labelled data Closed-form solution

$$\log L(\theta) = \sum_{x^r} \log P_\theta(x^r, \hat{y}^r)$$

$$\begin{aligned} & P_\theta(x^r, \hat{y}^r) \\ &= P_\theta(x^r | \hat{y}^r) P(\hat{y}^r) \end{aligned}$$

- Maximum likelihood with labelled + unlabeled data

$$\log L(\theta) = \sum_{x^r} \log P_\theta(x^r, \hat{y}^r) + \sum_{x^u} \log P_\theta(x^u) \quad \begin{array}{l} \text{Solved} \\ \text{iteratively} \end{array}$$

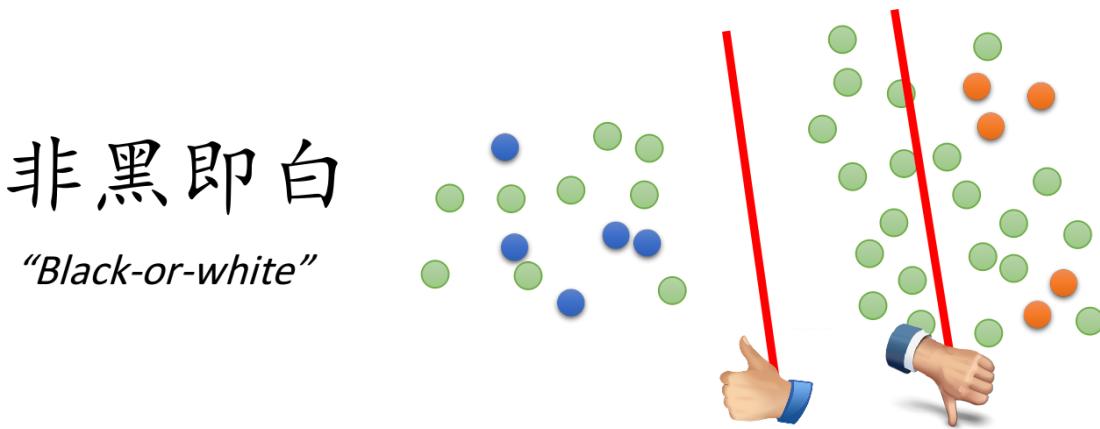
$$P_\theta(x^u) = P_\theta(x^u | C_1)P(C_1) + P_\theta(x^u | C_2)P(C_2)$$

(x^u can come from either C_1 and C_2)

Low-density Separation Assumption

接下来介绍一种新的方法，它基于的假设是Low-density separation

通俗来讲，就是这个世界是非黑即白的，在两个class的交界处data的密度(density)是很低的，它们之间会有一道明显的鸿沟，此时unlabeled data(下图绿色的点)就是帮助你在原本正确的基础上挑一条更好的boundary



Self Training

low-density separation最具代表性也最简单的方法是**self training**

- 先从labeled data去训练一个model f^* ，训练方式没有限制
- 然后用该 f^* 去对unlabeled data打上label, $y^u = f^*(x^u)$, 也叫作pseudo label
- 从unlabeled data中拿出一些data加到labeled data里，至于data的选取需要你自己设计算法来挑选
- 回头再去训练 f^* ，循环即可

注：该方法对Regression是不适用的

该方法与之前提到的generative model还是挺像的，区别在于：

- Self Training使用的是hard label：假设一笔data强制属于某个class
- Generative Model使用的是soft label：假设一笔data可以按照概率划分，不同部分属于不同class

如果我们使用的是neural network的做法， θ^* 是从labeled data中得到的一组参数，此时丢进来一个unlabeled data x^u ，通过 $f_{\theta^*}^*$ ()后得到 $\begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}$ ，即它有0.7的概率属于class 1, 0.3的概率属于class 2

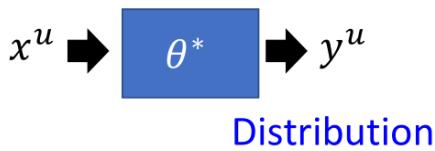
- 如果此时使用hard label，则 x^u 的label被转化成 $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- 如果此时使用soft label，则 x^u 的label依旧是 $\begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}$

可以看到，在neural network里使用soft label是没有用的，因为把原始的model里的某个点丢回去重新训练，得到的依旧是同一组参数，实际上low density separation就是通过强制分类 (hard label) 来提升分类效果的方法

Entropy-based Regularization

该方法是low-density separation的进阶版，你可能会觉得hard label这种直接强制性打标签的方式有些太武断了，而entropy-based regularization则做了相应的改进： $y^u = f_{\theta^*}^*(x^u)$ ，其中 y^u 是一个**概率分布 (distribution)**

由于我们不知道unlabeled data x^u 的label到底是什么，但如果通过entropy-based regularization得到的分布集中在某个class上的话，那这个model就是好的，而如果分布是比较分散的，那这个model就是不好的，如下图所示：



Entropy of y^u :
Evaluate how concentrate
the distribution y^u is



$$E(y^u) = 0$$

$$E(y^u) = - \sum_{m=1}^5 y_m^u \ln(y_m^u)$$

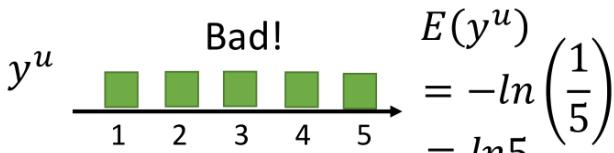
As small as possible



$$E(y^u) = 0$$

$$L = \sum_{x^r} C(y^r, \hat{y}^r) \quad \boxed{\text{labelled data}}$$

$$+ \lambda \sum_{x^u} E(y^u) \quad \boxed{\text{unlabeled data}}$$



$$\begin{aligned} E(y^u) &= -\ln\left(\frac{1}{5}\right) \\ &= \ln 5 \end{aligned}$$

接下来的问题是，如何用数值的方法来evaluate distribution的集中(好坏)与否，要用到的方法叫entropy，一个distribution的entropy可以告诉你它的集中程度：

$$E(y^u) = - \sum_{m=1}^5 y_m^u \ln(y_m^u)$$

对上图中的第1、2种情况，算出的 $E(y^u) = 0$ ，而第3种情况，算出的 $E(y^u) = -\ln(\frac{1}{5}) = \ln(5)$ ，可见entropy越大，distribution就越分散；entropy越小，distribution就越集中

因此我们的目标是在labelled data上分类要正确，在unlabeled data上，output的entropy要越小越好，此时就要修改loss function

- 对labelled data来说，它的output要跟正确的label越接近越好，用cross entropy表示如下：

$$L = \sum_{x^r} C(y^r, \hat{y}^r)$$

- 对unlabeled data来说，要使得该distribution(也就是output)的entropy越小越好：

$$L = \sum_{x^u} E(y^u)$$

- 两项综合起来，可以用weight来加权，以决定哪个部分更为重要一些

$$L = \sum_{x^r} C(y^r, \hat{y}^r) + \lambda \sum_{x^u} E(y^u)$$

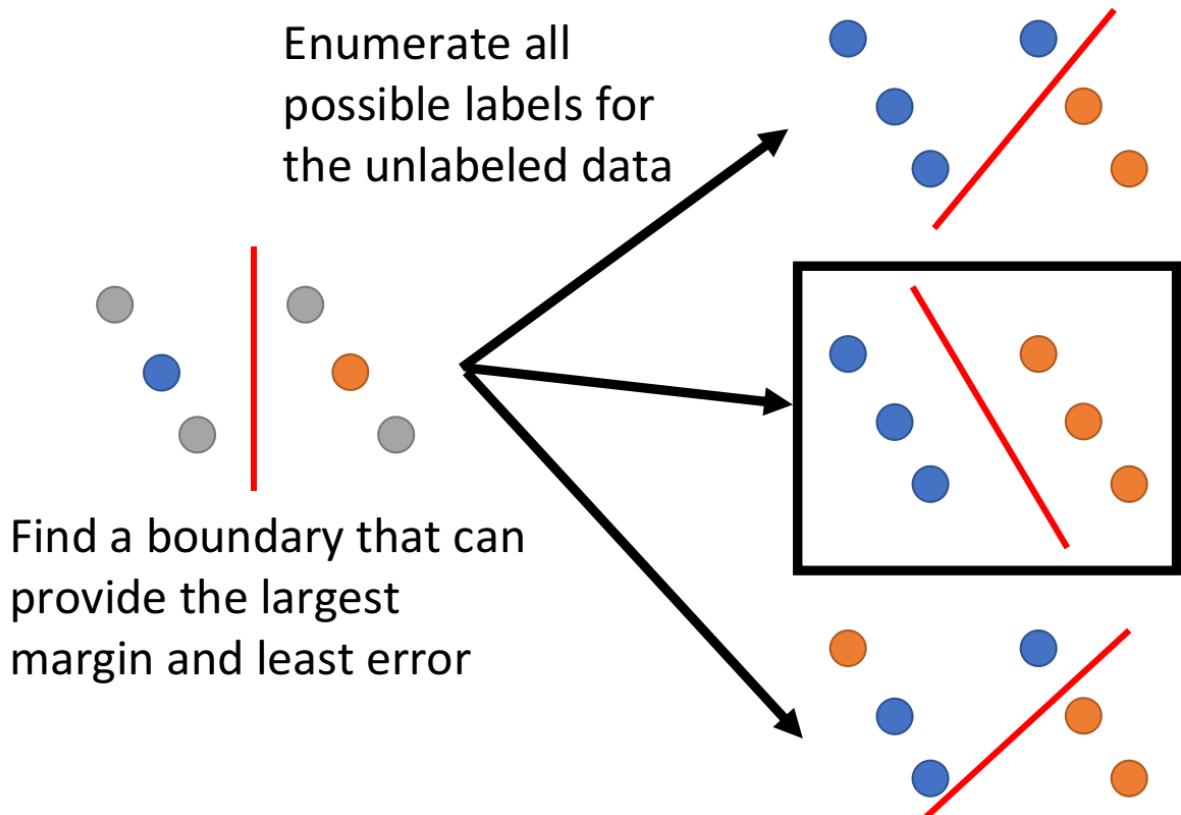
可以发现该式长得很像regularization，这也就是Entropy-based Regularization的名称由来

Semi-supervised SVM

SVM要做的是，给你两个class的数据，去找一个boundary：

- 要有最大的margin，让这两个class分的越开越好
- 要有最小的分类错误

对unlabeled data穷举所有可能的label，下图中列举了三种可能的情况；然后对每一种可能的结果都去算SVM，再找出可以让margin最大，同时又minimize error的那种情况，下图中是用黑色方框标注的情况



Thorsten Joachims, "Transductive Inference for Text Classification using Support Vector Machines", ICML, 1999

当然这么做会存在一个问题，对于 n 笔unlabeled data，意味着即使在二元分类里也有 2^n 种可能的情况，数据量大的时候，几乎难以穷举完毕，上面给出的paper提出了一种approximate的方法，基本精神是：一开始你先得到一些label，然后每次改一笔unlabeled data的label，看看可不可以让你的objective function变大，如果变大就去改变该label，具体内容详见paper

Smoothness Assumption

Concepts

smoothness assumption的基本精神是：近朱者赤，近墨者黑

粗略的定义是相似的 x 具有相同的 \hat{y} ，精确的定义是：

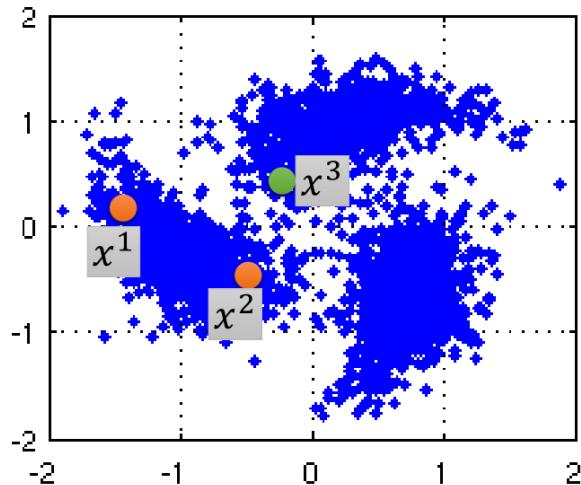
- x 的分布是不平均的，在某些地方很集中，某些地方很分散
- 如果 x^1 和 x^2 在一个high density region上很接近的话，那么 \hat{y}^1 和 \hat{y}^2 就是相同的

也就是这两个点可以在样本点高密度集中分布的区域块中有一条可连接的路径，即 connected by a high density path

假设下图是data的分布， x^1, x^2, x^3 是其中的三笔data，如果单纯地看 x 的相似度，显然 x^2 和 x^3 更接近一些，但对于smoothness assumption来说， x^1 和 x^2 是处于同一块区域的，它们之间可以有一条相连的路径；而 x^2 与 x^3 之间则是“断开”的，没有high density path，因此 x^1 与 x^2 更“像”

- Assumption: “similar” x has the same \hat{y}
- More precisely:
 - x is not uniform.
 - If x^1 and x^2 are close in a high density region, \hat{y}^1 and \hat{y}^2 are the same.

connected by a high density path



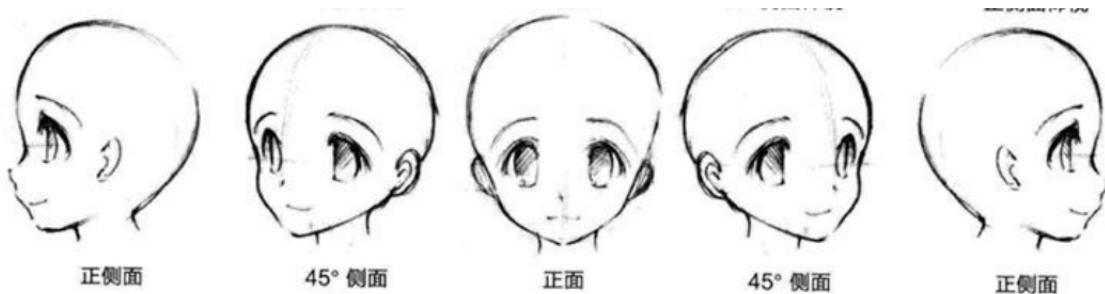
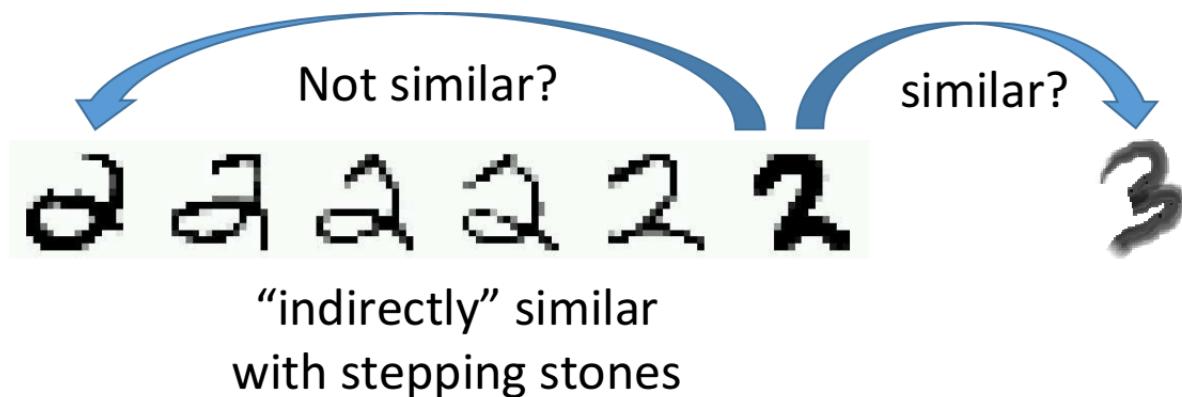
Source of image:
<http://hips.seas.harvard.edu/files/pinwheel.png>

x^1 and x^2 have the same label
 x^2 and x^3 have different labels

Digits Detection

以手写数字识别为例，对于最右侧的2和3以及最左侧的2，显然最右侧的2和3在pixel上相似度更高一些；但如果把所有连续变化的2都放进来，就会产生一种“不直接相连的相似”，根据Smoothness Assumption的理论，由于2之间有连续过渡的形态，因此第一个2和最后一个2是比较像的，而最右侧2和3之间由于没有过渡的数据，因此它们是比较不像的

人脸的过渡数据也同理



File Classification

Smoothness Assumption在文件分类上是非常有用的

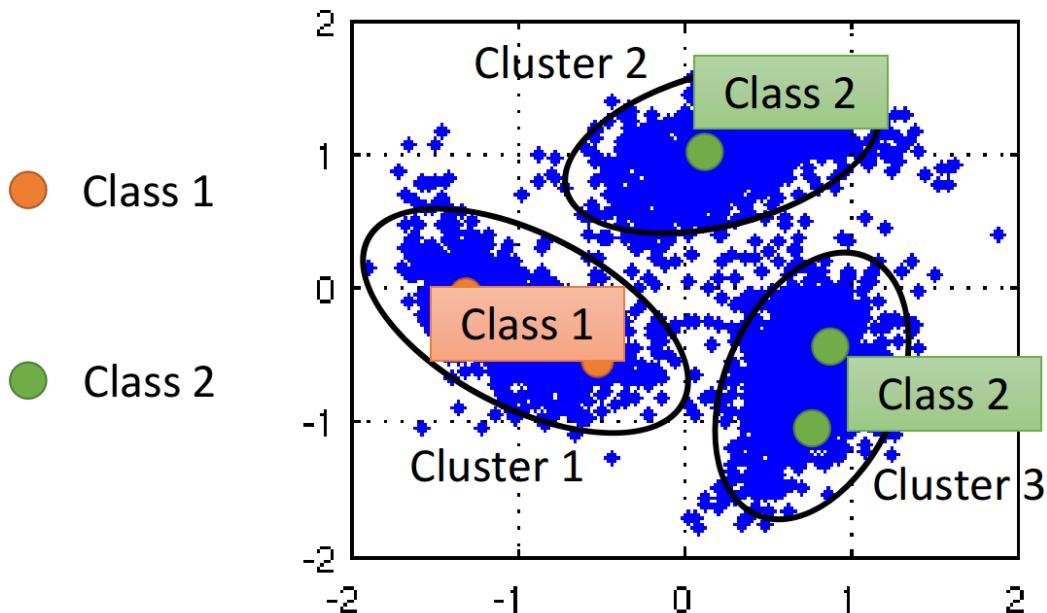
假设对天文学(astronomy)和旅行(travel)的文章进行分类，它们各自有专属的词汇，此时如果unlabeled data与label data的词汇是相同或重合(overlap)的，那么就很容易分类；但在真实的情况下，unlabeled data和labeled data之间可能没有任何重复的words，因为世界上的词汇太多了，sparse的分布很难会使overlap发生

但如果unlabeled data足够多，就会以一种相似传递的形式，建立起文档之间相似的桥梁

Cluster and then label

在具体实现上，有一种简单的方法是cluster and then label，也就是先把data分成几个cluster，划分class之后再拿去训练，但这种方法不一定会得到好的结果，因为它的假设是你可以把同一个class的样本点cluster在一起，而这其实是没那么容易的

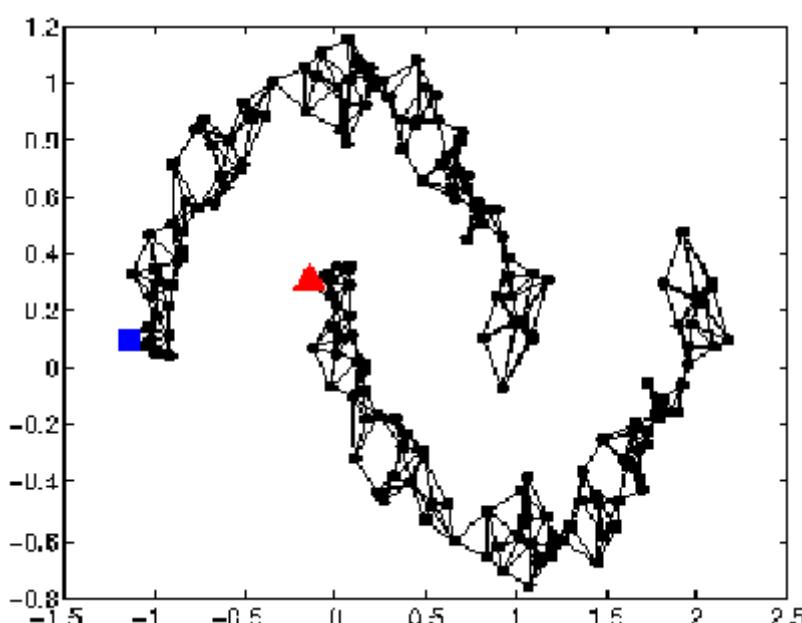
对图像分类来说，如果单纯用pixel的相似度来划分cluster，得到的结果一般都会很差，你需要设计一个很好的方法来描述image(类似Deep Autoencoder的方式来提取feature)，这样cluster才会有效果



Using all the data to learn a classifier as usual

Graph-based Approach

之前讲的是比较直觉的做法，接下来引入Graph Structure来表达connected by a high density path这件事



Represented the data points as a graph, 有时候建立vertex之间的关系是比较容易的，比如网页之间的链接关系、论文之间的引用关系；但有时候需要你自己去寻找vertex之间的关系，建立graph

graph的好坏，对结果起着至关重要的影响，而如何build graph却是一件heuristic的事情，需要凭着经验和直觉来做

- 首先定义两个object x^i, x^j 之间的相似度 $s(x^i, x^j)$

如果是基于pixel的相似度，performance可能会不太好；建议使用autoencoder提取出来的feature来计算相似度，得到的performance会好一些

- 算完相似度后，就可以建graph了，方式有很多种：

- k nearest neighbor：假设 $k=3$ ，则每个point与相似度最接近的3个点相连

- e -Neighborhood：每个point与相似度超过某个特定threshold e 的点相连

- 除此之外，还可以给Edge特定的weight，让它与相似度 $s(x^i, x^j)$ 成正比

- 建议用Gaussian Radial Basis Function来确定相似度： $s(x^i, x^j) = e^{-\gamma \|x^i - x^j\|^2}$

这里 x^i, x^j 均为vector，计算它们的Euclidean Distance(欧几里得距离)，乘一个参数后再取exponential

- 至于取exponential，经验上来说通常是可以帮助提升performance的，在这里只有当 x^i, x^j 非常接近的时候，similarity才会大；只要距离稍微远一点，similarity就会下降得很快，变得很小

- 使用exponential的Gaussian Radial Basis Function可以做到只有非常近的两个点才能相连，稍微远一点就无法相连的效果，避免了下图中跨区域相连的情况

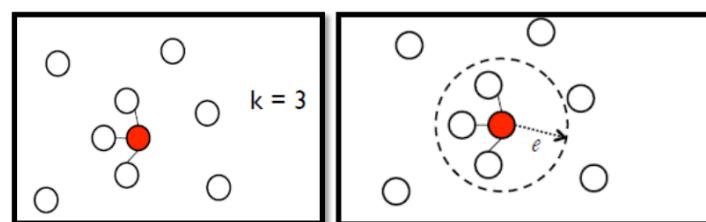
Graph-based Approach - Graph Construction

The image is from the tutorial slides of Amarnag Subramanya and Partha Pratim Talukdar

- Define the similarity $s(x^i, x^j)$ between x^i and x^j

- Add edge:

- K Nearest Neighbor

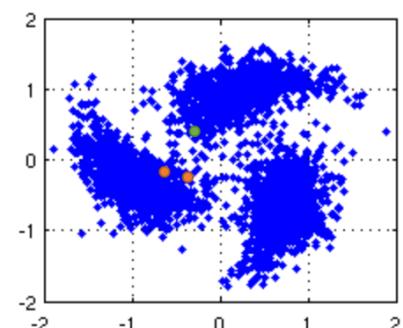


- e -Neighborhood

- Edge weight is proportional to $s(x^i, x^j)$

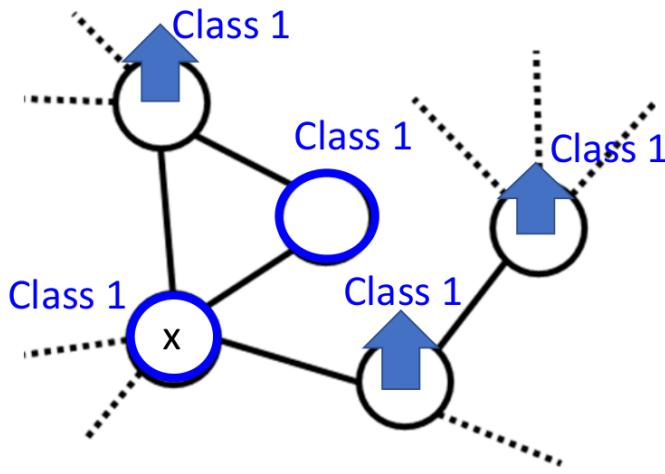
Gaussian Radial Basis Function:

$$s(x^i, x^j) = \exp(-\gamma \|x^i - x^j\|^2)$$



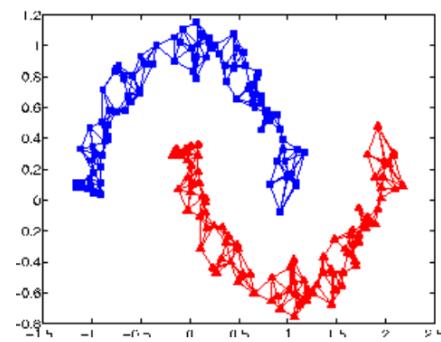
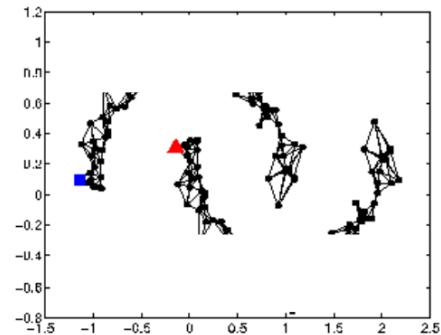
graph-based approach的基本精神是，在graph上已经有一些labeled data，那么跟它们相连的point，属于同一类的概率就会上升，每一笔data都会去影响它的邻居，而graph带来的最重要的好处是，这个影响是会随着edges传递出去的，即使有些点并没有真的跟labeled data相连，也可以被传递到相应的属性

比如下图右下，如果graph建的足够好，那么两个被分别label为蓝色和红色的点就可以传递完两张完整的图；从下图右上中我们也可以看出，如果想要让这种方法生效，收集到的数据一定要足够多，否则可能传递到一半，graph就断掉了，information的传递就失效了



The labelled data influence their neighbors.

Propagate through the graph



介绍了如何定性使用graph，接下来介绍一下如何定量使用graph

定量的使用方式是定义label的smoothness，下图中，edge上的数字是weight， x^i 表达data， y^i 表示data的label，计算smoothness的方式为：

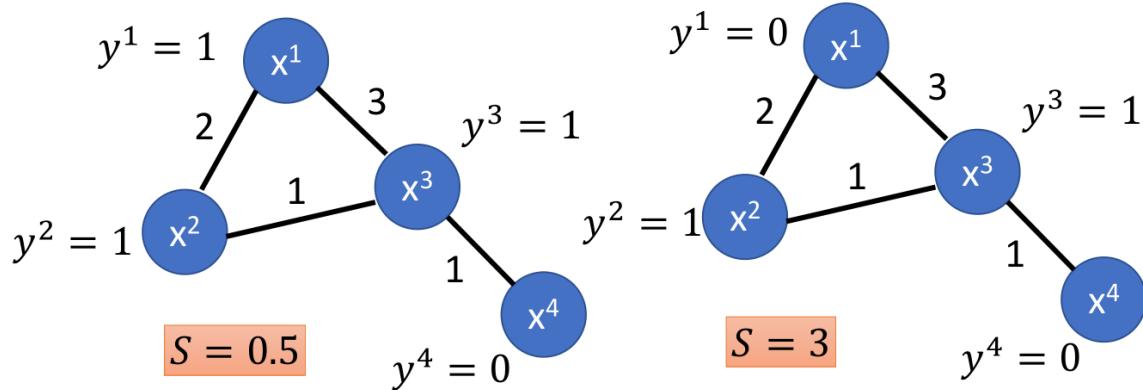
$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (y^i - y^j)^2$$

我们期望smooth的值越小越好

- Define the smoothness of the labels on the graph

$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (y^i - y^j)^2$$

Smaller means smoother
For all data (no matter labelled or not)



当然上面的式子还可以化简，如果把labelled data和unlabelled data的y组成一个(R+U)-dim vector，即

$$\mathbf{y} = [\dots y^i \dots y^j]^T$$

于是smooth可以改写为：

$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (y^i - y^j)^2 = \mathbf{y}^T L \mathbf{y}$$

其中 L 为 $(R+U) \times (R+U)$ matrix, 成为**Graph Laplacian**, 定义为 $L = D - W$

- W : 把data point两两之间weight的关系建成matrix, 代表了 x^i 与 x^j 之间的weight值
- D : 把 W 的每一个row上的值加起来放在该行对应的diagonal上即可, 比如 $5=2+3, 3=2+1, \dots$

- Define the smoothness of the labels on the graph

$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (y^i - y^j)^2 = \mathbf{y}^T L \mathbf{y}$$

\mathbf{y} : $(R+U)$ -dim vector

$$\mathbf{y} = [\dots y^i \dots y^j \dots]^T$$

L : $(R+U) \times (R+U)$ matrix

Graph Laplacian	$W = \begin{bmatrix} 0 & 2 & 3 & 0 \\ 2 & 0 & 1 & 0 \\ 3 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	$D = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
------------------------	--	--

对 $S = \mathbf{y}^T L \mathbf{y}$ 来说, y 是label, 是neural network的output, 取决于neural network的parameters, 因此要在原来仅针对labeled data的loss function中加上这一项, 得到:

$$L = \sum_{x^r} C(y^r, \hat{y}^r) + \lambda S$$

λS 实际上也象征着一个regularization term

训练目标:

- labeled data的cross entropy越小越好(neural network的output跟真正的label越接近越好)
- smooth S 越小越好(neural network的output, 不管是labeled还是unlabeled, 都要符合Smoothness Assumption的假设)

具体训练的时候, 不一定只局限于neural network的output要smooth, 可以对中间任意一个hidden layer 加上smooth的限制

- Define the smoothness of the labels on the graph

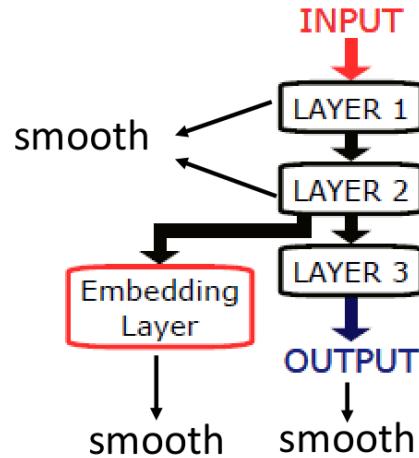
$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (y^i - y^j)^2 = \mathbf{y}^T L \mathbf{y}$$

Depending on network parameters

$$L = \sum_{x^r} C(y^r, \hat{y}^r) + \lambda S$$

As a regularization term

J. Weston, F. Ratle, and R. Collobert, “Deep learning via semi-supervised embedding,” ICML, 2008



Better Representation

Better Representation的精神是，去芜存菁，化繁为简

我们观察到的世界是比较复杂的，而在它的背后其实是有一些比较简单的东西，在操控着这个复杂的世界，所以只要你能够看透这个世界的假象，直指它的核心的话，就可以让training变得比较容易

Find the latent factors behind the observation

The latent factors (usually simpler) are better representations

算法具体思路和内容unsupervised learning中介绍

Explainable Machine Learning

Explainable Machine Learning

机器不但要知道，还要告诉我们它为什么会知道。

Local Explanation——Why do you think this image is a cat?

Global Explanation——What do you think a “cat” looks like?

Why we need Explainable ML?

- 用机器来协助判断简历
 - 具体能力？还是性别？
- 用机器来协助判断犯人是否可以假释
 - 具体证据？还是肤色？
- 金融相关的决策常常依法需要提供理由
 - 为什么拒绝了某个人的贷款？
- 模型诊断：到底机器学到了什么

- 不能只看正确率吗？今天我们看到各式各样机器学习非常强大的力量，感觉机器好像非常的聪明，过去有一只马叫做汉斯，它非常的聪明，聪明到甚至可以做数学。举例来说：你跟它讲根号9是多少，它就会敲它的马蹄，大家欢呼道，这是一只会算数学的马。可以解决根号的问题，大家都觉得非常的惊叹。后面就有人怀疑说：难道汉斯真的这么聪明吗？在没有任何的观众的情况下，让汉斯自己去解决一个数学题目，这时候它就会一直踏它的马蹄，不停的不停。这是为什么呢？因为它之前学会了观察旁观人的反应，它知道什么时候该停下来。它可能不知道自己在干什么，它也不知道数学是什么，但是踏对了正确的题目就有萝卜吃，它只是看了旁边人的反应得到了正确的答案。今天我们看到种种机器学习的成果，难道机器真的有那么的聪明吗？会不会它会汉斯一样用了奇怪的方法来得到答案的。

We can improve ML model based on explanation.

当我们可以做可解释的机器学习模型时，我们就能做模型诊断，就可以知道机器到底学到了什么，是否和我们的预期相同。准确率不足以让我们精确调整模型，只有当我们知道 why the answer is wrong, so i can fix it.

My Point of View

- Goal of ML Explanation ≠ you completely know how the ML model work (Not necessary)
 - Human brain is also a Black Box!
 - People don't trust network because it is Black Box, but you trust the decision of human!
- Goal of ML Explanation is —— Make people (your customers, your boss, yourself) comfortable.
- Personalized explanation in the future

可解释机器学习的目标，不需要真正知道模型如何工作，只需要给出可信服的解释，让人满意就行。对此还可以针对不同人的接受能力给出不同层次的解释。

Interpretable v.s. Powerful

- Some models are intrinsically interpretable.
 - For example, linear model (from weights, you know the importance of features)
 - But...not very powerful.
- Deep network is difficult to interpret.
 - Deep network is a black box.
 - But it is more powerful than linear model...
 - Because deep network is a black box, we don't use it. (这样做是不对的，削足适履，Let's make deep network interpretable.)
- Are there some models interpretable and powerful at the same time?
 - How about decision tree?

模型的可解释性和模型的能力之间有矛盾。

一些模型，比如线性模型，可解释性很好，但效果不佳。而深度网络，虽然能力一流，但缺乏可解释性。

我们的目标不是放弃复杂的模型，直接选择可解释性好的模型，而是让能力强的模型具有更好的解释性，去尝试解释复杂模型。

同时具有强大能力和可解释性的，是决策树。但决策树结构如果很复杂，那么可解释性也会很差。(森林)

Local Explanation: Explain the Decision

假设我们的Object x 有N个components $\{x_1, \dots, x_n, \dots, x_N\}$, Image: pixel, segment, etc; Text: a word

We want to know the importance of each components for making the decision.

Idea

Removing or modifying the values of the components, observing the change of decision.

如果有Large decision change, 那么就可以得到important component

比如找一个图片，把一个灰色的方块放在在图片中任意一个位置。当灰色方块位于某个位置导致机器判断改变，那么我们就把这个区域看为重要的component。注意，覆盖图片的方块的颜色、大小都是需要人工调整的参数，会影响结果，这其实是至关重要crucial的。甚至选择不同的参数可以得到不同的结果。

$$\{x_1, \dots, x_n, \dots, x_N\} \rightarrow \{x_1, \dots, x_n + \Delta x, \dots, x_N\}$$

$$y_k \rightarrow y_k + \Delta y$$

y_k : the prob of the predicted class of the model

对于每一个输 x 将其某个维度加上一个小小的扰动，然后观察模型判断结果和原判断结果的差值，根据这个扰动造成的结果差值来了解机器对图片中哪些像素比较敏感。影响可以用 $|\frac{\Delta y}{\Delta x}|$ 来表示，计算方式就

是偏微分： $|\frac{\partial \Delta y_k}{\partial \Delta x_n}|$

得到的图称为：Saliency Map，亮度代表了偏微分的绝对值，也是pixel的重要性

Karen Simonyan, Andrea Vedaldi, Andrew Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR, 2014

To Learn More.....

Grad-CAM

SmoothGrad

Layer-wise Relevance Propagation

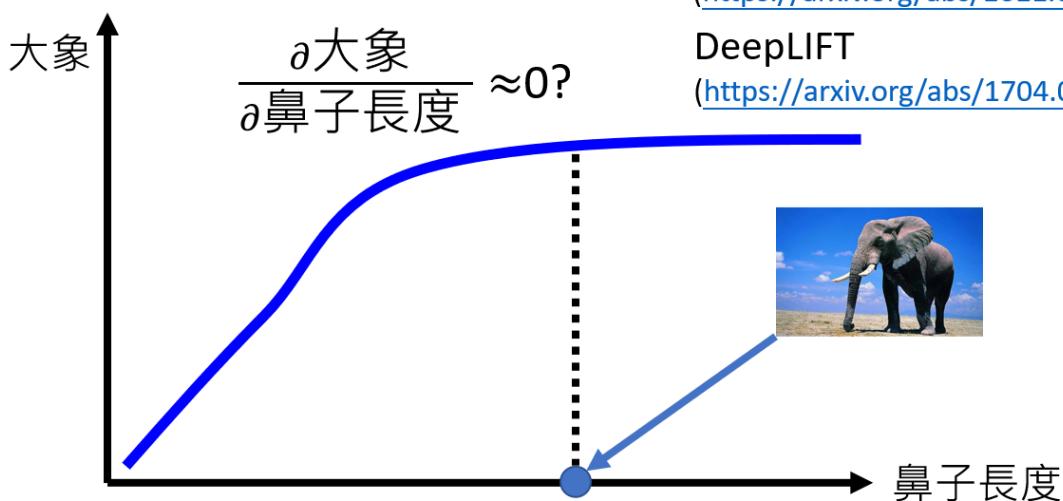
Guided Backpropagation

Limitation of Gradient based Approaches

Gradient Saturation

刚才我们用gradient的方法判断一个pixel或者一个segment是否重要，但是梯度有一个限制就是gradient saturation (梯度饱和)。我们如果通过鼻子的长度来判断一个生物是不是大象，鼻子越长是大象的信心分数越高，但是当鼻子长到一定程度信心分数就不太会变化了，信心分数就封顶了。鼻子长到一定程度，就可以确定这是一只大象，但是导数却是0，得出鼻子不重要这样的结论，明显是不对的。

- Gradient Saturation



To deal with this problem:

Integrated gradient
(<https://arxiv.org/abs/1611.02639>)

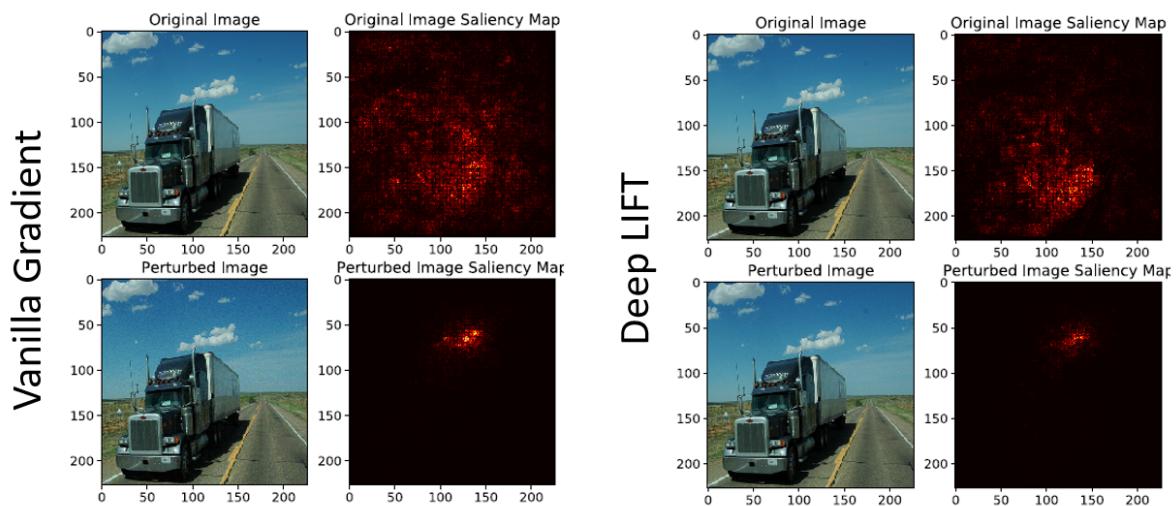
DeepLIFT
(<https://arxiv.org/abs/1704.02685>)



Attack Interpretation

攻击机器学习的解释是可能的。我们可以加一些神奇的扰动到图片上，人眼虽然辨识不出来，也不改变分类结果，但是模型聚焦的判定点却变了。

- It is also possible to attack interpretation...



The noise is small, and do not change the classification results.

Case Study: Pokémon v.s. Digimon

通过Saliency Map，看到机器更在意边缘，而非宝可梦的本体

All the images of Pokémon are PNG, while most images of Digimon are JPEG. PNG文件透明背景，读档后背景是黑的。Machine discriminate Pokémon and Digimon based on Background color.

This shows that **explainable ML is very critical**.

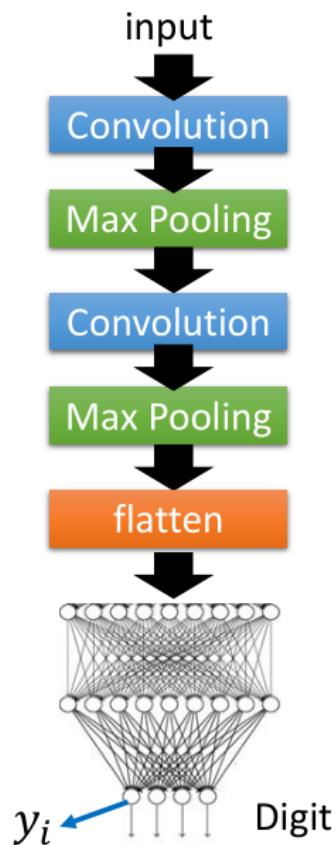
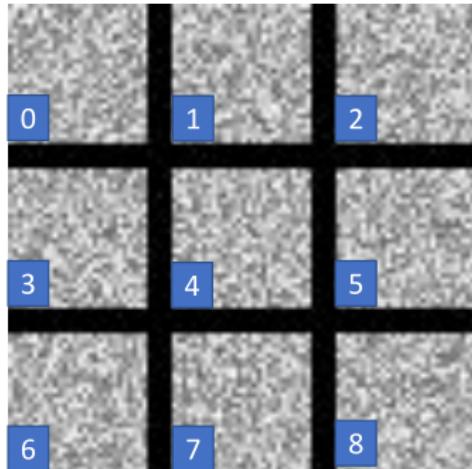
Global Explanation: Explain the whole model

Activation Maximization

这块内容之前有讲过，就是找到一个图片，让某个隐藏层的输出最大，例如在手写数字辨识中，找出机器认为理想的数字如下

Activation Minimization (review)

$$x^* = \arg \max_x y_i \quad \text{Can we see digits?}$$



Deep Neural Networks are Easily Fooled

<https://www.youtube.com/watch?v=M2lebCN9Ht4>

若不加限制，得到的结果经常会像是杂乱雪花，加一些额外的限制，使图像看起来更像数字（或其他正常图片）。我们不仅要maximize output的某个维度，还要加一个限制函数 $R(x)$ 让 x 尽可能像一个数字图片，这个函数输入是一张图片，输出是这张图片有多像是数字。

那这个限制函数 $R(x)$ 要怎么定义呢，定义方式有很多种，这里就简单的定义为所有pixel的和的数值，这个值越小说明黑色RGB(0,0,0)越多，这个图片就越像是一个数字，由于这里数字是用白色的笔写的，所以就加一个负号，整体去maximize y_i 与 $R(x)$ 的和，形式化公式就变成了：

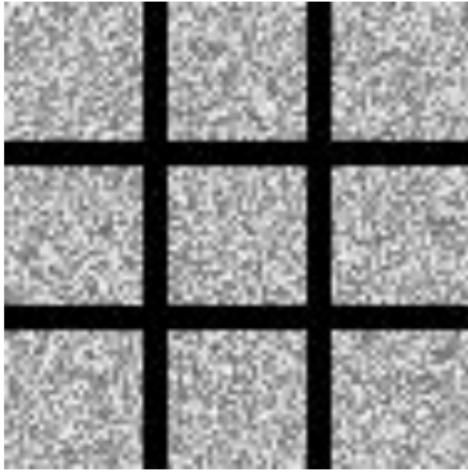
$$x^* = \arg \max_x (y_i + R(x))$$

得到的结果如下图所示，比加限制函数之前要好很多了，从每个图片中多少能看出一点数字的形状了，尤其是6那一张。

Activation Minimization (review)

Find the image that maximizes class probability

$$x^* = \arg \max_x y_i$$

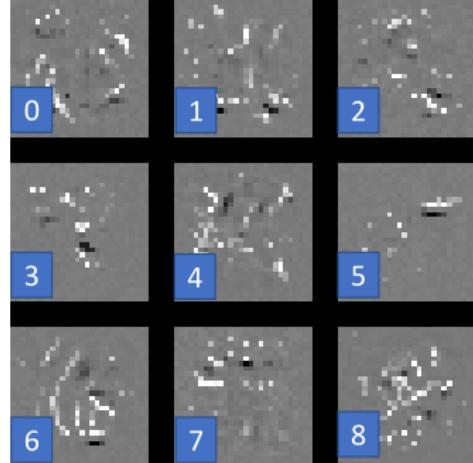


The image also looks like a digit.

$$x^* = \arg \max_x y_i + R(x)$$

$$R(x) = - \sum_{i,j} |x_{ij}|$$

How likely
x is a digit



With several regularization terms, and hyperparameter tuning

在更复杂的模型上，比如要在ImgNet这个大语料库训练出的模型上做上述的事情，在不加限制函数的情况下将会得到同样杂乱无章的结果。而且在这种复杂模型中要想得到比较好的Global Explanation结果，往往要加上更多、更复杂、更精妙的限制函数。

<https://arxiv.org/abs/1506.06579>

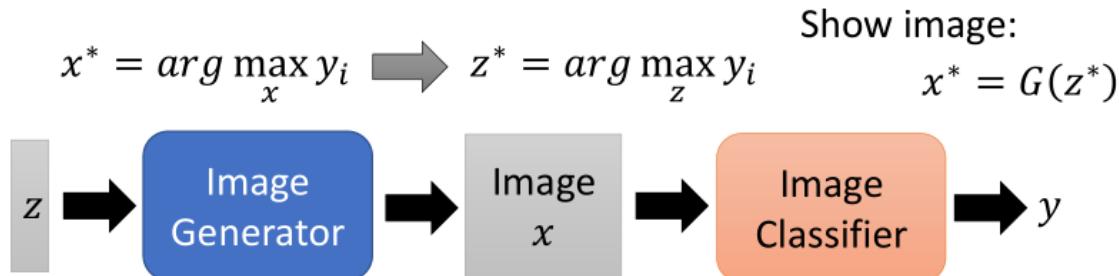
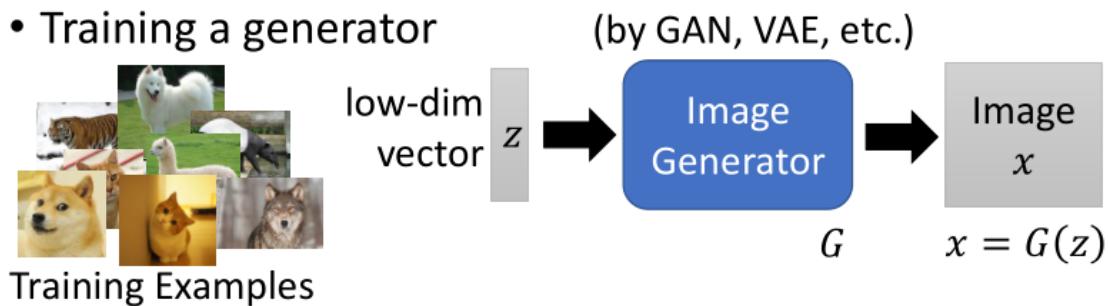
Constraint from Generator

上面提到要对图片做一个限制，比较像一个图片，这个事情可以用Generator来做。

我们将方法转变为：找一个 z ，将它输入到图片生成器中产生一个图片，再将这个图片输入原来的图片分类器，得到一个类别判断 y ，我们同样要maximize y_i . 其实可以考虑为我们把图片生成器和图片分类器连接在一起，fix住两者的参数，通过梯度下降的方法不断改变 z ，最终找到一个合适的 z^* 可以maximize y_i . 之后我们只要将这个 z^* 丢入图片生成器就可以拿到图片 x^* 了。形式化表述为：

$$z^* = \arg \max_x y_i$$

调整输入低维向量 z 得到理想的 y ，把 z 通过generator就可以得到理想的image

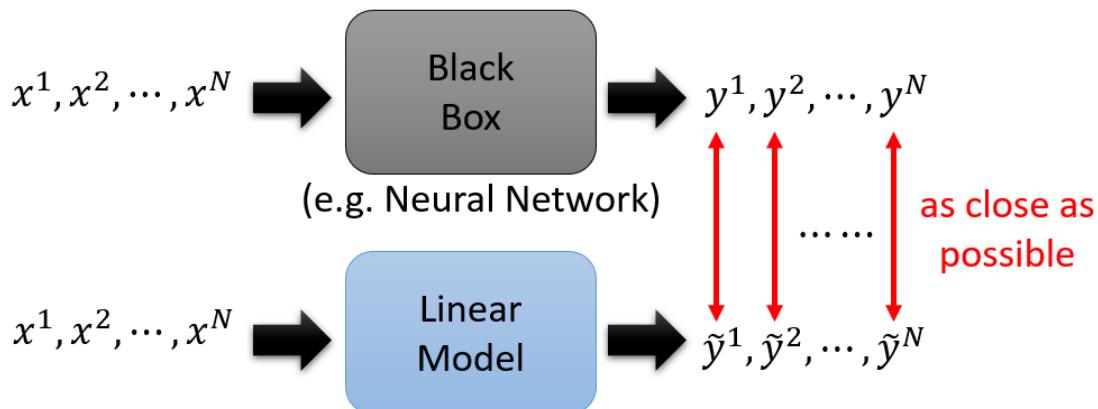


这里和GAN里面的discriminator不一样，discriminator只会判断generator生成的图片好或者不好，这里是生成某一个类型的图片。并且generator是固定的，我们调整的是低维向量z。

Using A Model to Explain Another

还有一种很神奇的方法用于解释模型，那就是用可解释的模型去模拟不可解释的模型，然后通过可解释的模型给出解释。

Some models are easier to Interpret. Using an interpretable model to mimic the behavior of an uninterpretable model.

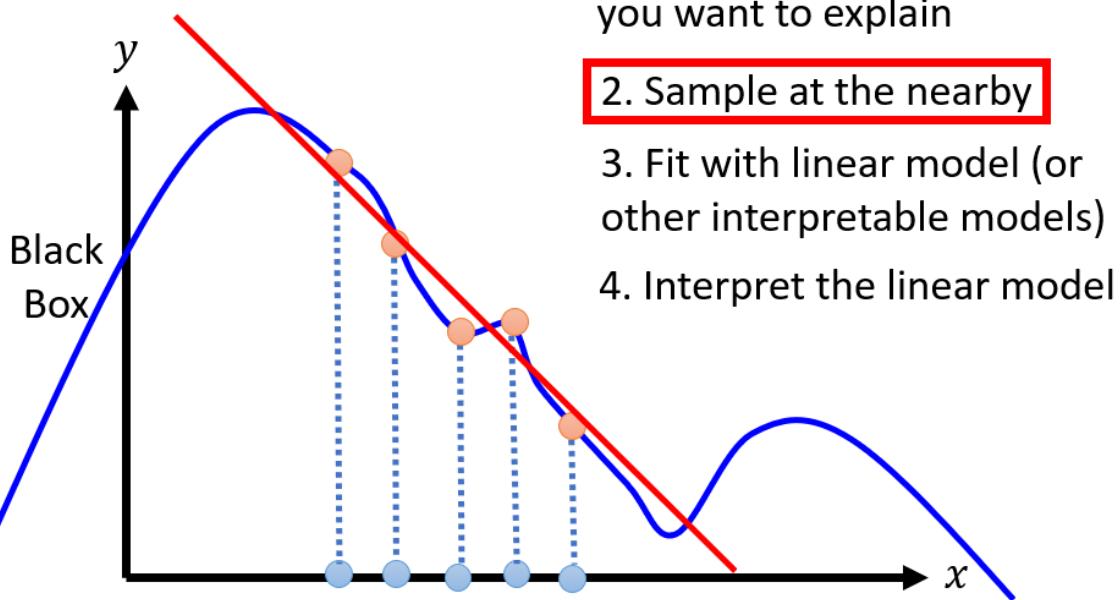


Problem: Linear model cannot mimic neural network ...

However, it can mimic a local region.

具体思路也很简单，就是喂给目标模型一些样本并记录输出作为训练集，然后用这个训练集训练一个解释性良好的模型对目标模型进行解释。然而可解释性好的模型往往是简单的模型，可解释性差的模型往往是复杂的模型，用一个简单模型去模拟一个复杂模型无疑是困难的。上图就是一个示例。用线性模型去模拟一个神经网络效果肯定是很差的，但是我们可以做的是局部的模拟。

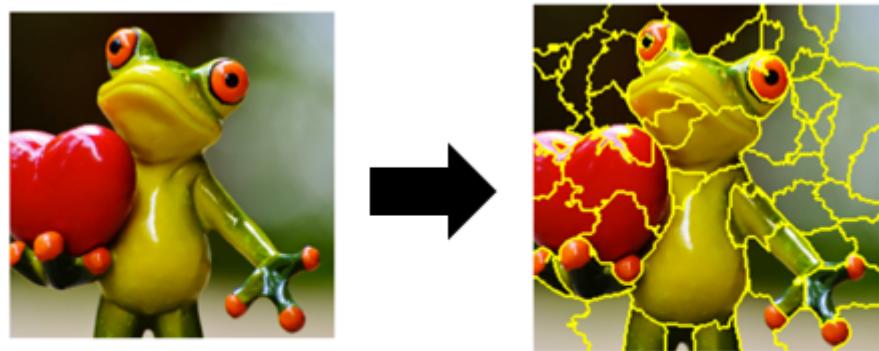
Local Interpretable Model-Agnostic Explanations (LIME)



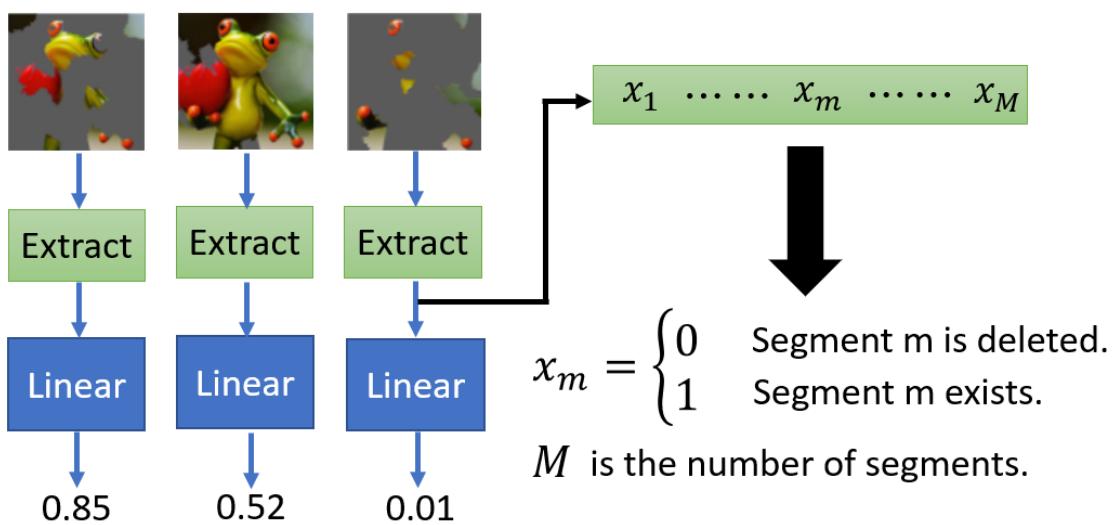
也就是说当我们给定一个输入的时候，我们要在这个输入附近找一些数据点，然后用线性模型进行拟合。那么当输入是图片的时候，什么样的数据算是这张图片附近的数据点呢？这需要不断调整参数。

LIME - Image

1. Given a data point you want to explain
2. Sample at the nearby
 - o Each image is represented as a set of superpixels (segments).
 - o Randomly delete some segments.
 - o Compute the probability of "frog" by black box



3. Fit with linear (or interpretable) model



4. Interpret the model you learned



Extract

Linear

0.85

$$y = w_1x_1 + \cdots + w_mx_m + \cdots + w_Mx_M + b$$

$$x_m = \begin{cases} 0 & \text{Segment } m \text{ is deleted.} \\ 1 & \text{Segment } m \text{ exists.} \end{cases}$$

M is the number of segments.

If $w_m \approx 0 \rightarrow$ segment m is not related to "frog"

If w_m is positive

\rightarrow segment m indicates the image is "frog"

If w_m is negative

\rightarrow segment m indicates the image is not "frog"

举例：

1. 首先现有一张要解释的图片，我们想知道模型为什么把它认作树蛙

2. 在这张图附近sample一些数据

我们通常会用一些toolkit把图片做一下切割，然后随机的丢掉一些图块，得到的新图片作为原图片附近区域中的数据点。

3. 把上述这些图片输入原黑盒模型，得到黑盒的输出

4. 用线性模型（或者其他可解释模型）fit上述数据

在上面的例子中我们在做图片辨识任务，此时我们可能需要在将图片丢到线性模型之前先做一个特征抽取。

5. 解释你的线性模型

如上图所示，当线性模型种某个特征维度对应的weight：

- 趋近于零，说明这个segment对模型判定树蛙不重要
- 正值，说明这个segment对模型判定树蛙有作用
- 负值，说明这个segment对模型判定树蛙有反作用

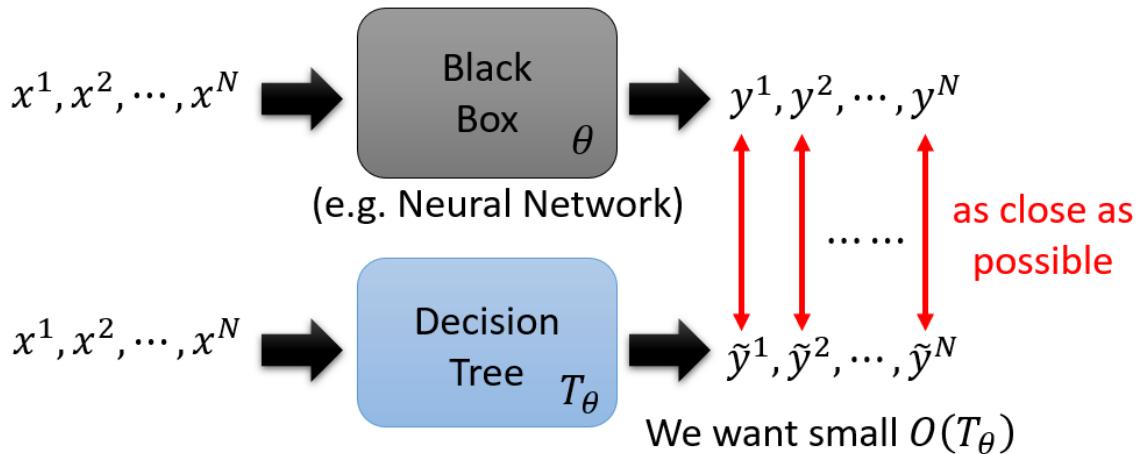
Decision Tree

下面用Decision Tree来代替上面的线性模型做解释工作。

Decision Tree

$O(T_\theta)$: how complex T_θ is
e.g. average depth of T_θ

- Using an interpretable model to mimic the behavior of an uninterpretable model.



Problem: We don't want the tree to be too large.

Decision Tree足够复杂时，理论上完全可以模仿黑盒子的结果，但是这样Decision Tree会非常非常复杂，那么Decision Tree本身又变得没法解释了，因此我们不想Decision Tree太过复杂。这里我们用 $O(T_\theta)$ 表示决策树的复杂度，定义方式可以自选，比如树的平均深度。

要考虑决策树的复杂度，因此要在损失函数中加一个正则项： $\theta = \underset{\theta}{\operatorname{argmin}} L(\theta) + \lambda O(T_\theta)$

Decision Tree – Tree regularization

<https://arxiv.org/pdf/1711.06178.pdf>

- Train a network that is easy to be interpreted by decision tree.

T_θ : tree mimicking network with parameters θ
 $O(T_\theta)$: how complex T_θ is

$$\theta^* = \arg \min_{\theta} L(\theta) + \lambda O(T_\theta)$$

Original loss function for training network

Preference for network parameters

→ Tree Regularization

Is the objective function with tree regularization differentiable? No! Check the reference for solution.

但是这个正则项没法做偏导，所以没有办法做GD。

解决方法：<https://arxiv.org/pdf/1711.06178.pdf>

train一个神奇的神经网络，我们给它一个神经网络的参数，它就可以输出一个数值，来表示输入的参数转成Decision Tree后Decision Tree的平均深度。

中心思想是用一个随机初始化的结构简单的Neural Network，找一些NN转成DT，算出平均深度，就有了训练data，训练后可以模拟出决策树的平均深度，然后用Neural Network替换上面的正则项，Neural Network是可以微分的，然后就可以Gradient Descent了。

Adversarial Attack

Adversarial Attack

Motivation

We seek to deploy machine learning classifiers not only in the labs, but also in real world.

The classifiers that are robust to noises and work “most of the time” is not sufficient. We want the classifiers that are robust to the inputs that are built to fool the classifier.

Especially useful for spam classification, malware detection, network intrusion detection, etc.

光是强还不够，还要应对人类的恶意攻击。

攻击是比较容易的，多数machine learning的model其实相当容易被各式各样的方法攻破，防御目前仍然是比较困难的。

Attack

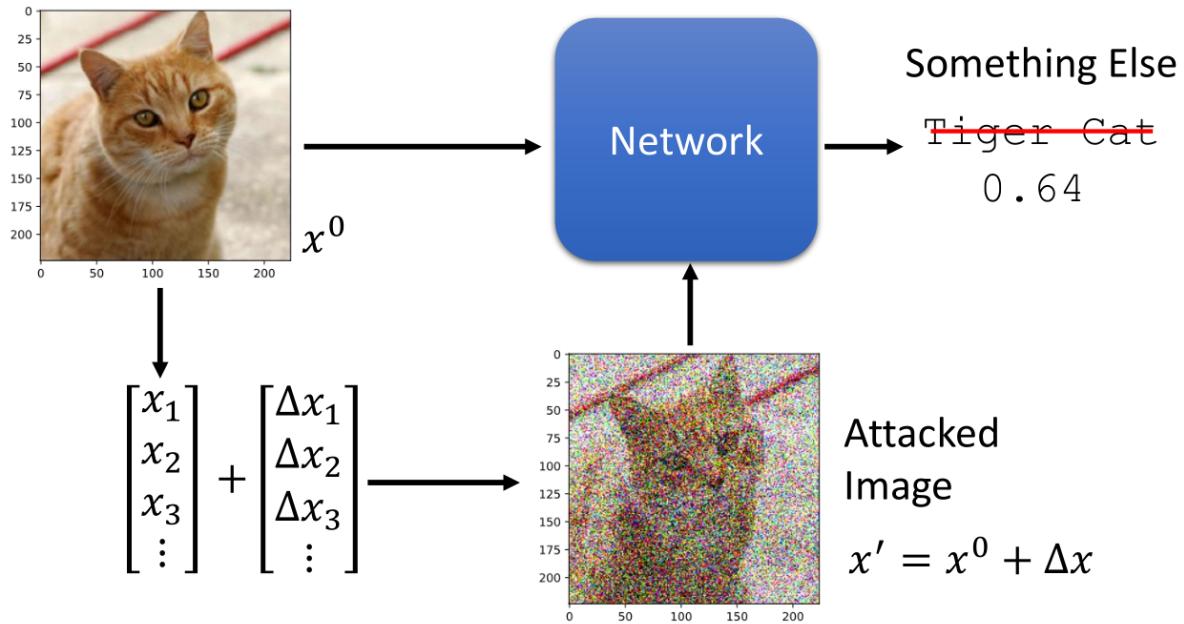
What do we want to do?

给当前的machine输入一个图片，输出Tiger Cat的信息分数为0.64，你不会说它是一个很差的model，你会说它是一个做的还不错的model。

我们现在想要做的是：往图片上面加上一些杂讯，这些杂讯不是从gaussion distribution中随机生成的（随机生成的杂讯，没有办法真的对Network造成太大的伤害），这是种很特殊的讯号，把这些很特殊的讯号加到一张图片上以后，得到稍微有点不一样的图片。将这张稍微有点不一样的图片丢到Network中，Network会得到非常不一样的结果。

本来的图片叫做 x_0 ，而现在是在原来的 x_0 上加上一个很特别的 Δx ，得到一张新的图片 x' （ $x' = x_0 + \Delta x$ ）。然后将 x' 丢到Network中，原来看到 x_0 时，Network会输出是一张Tiger Cat，但是这个Network看到时输出一个截然不同的答案（Attacked image, Else），那么这就是所谓攻击所要做的事情。

Original Image



Loss Function for Attack

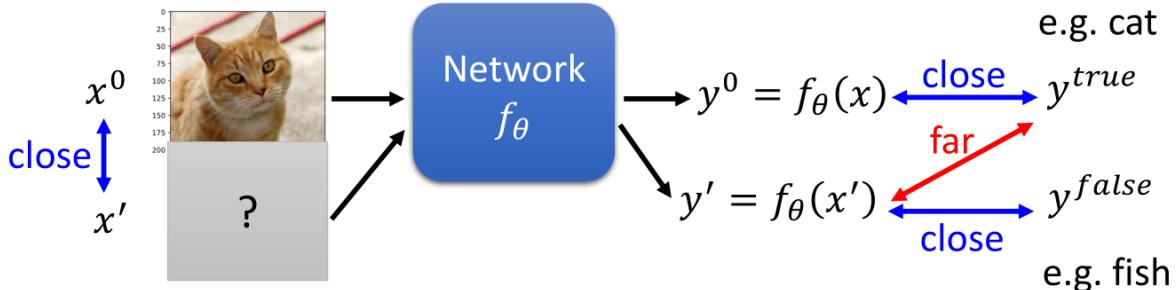
即找一张图片，使得loss(cross-entropy loss)越大越好，此时网络的参数训练完了，不能改变，而是只改变输入，使我们找到这样一张图片，能够让结果“越错越好”，离正确答案越远越好。

普通的训练模型， x^0 不变，希望找到一个 θ ，使cross-entropy越小越好

Non-targeted Attack， θ 不变，希望找到一个 x' ，使得cross-entropy越大越好，cross-entropy越大Loss越小

Targeted Attack，希望找到一个 x' ，使得cross-entropy越大越好，同时与目标（错误）标签越近越好

输入不能改变太大， x^0 和 x' 越接近越好



- Training:** $L_{train}(\theta) = C(y^0, y^{true})$ x fixed
- Non-targeted Attack:** $L(x') = -C(y', y^{true})$ θ fixed
- Targeted Attack:**

$$L(x') = -C(y', y^{true}) + C(y', y^{false})$$
- Constraint:** $d(x^0, x') \leq \varepsilon$ 不要被發現

Constraint

$distance(x^0, x')$ 由任务的不同而不同，主要取决于对人来说什么样的文字、语音、图像信号是相似的。

L-infinity稍微更适合用于影像的攻击上面

Constraint $d(x^0, x') \leq \varepsilon$

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \\ \vdots \end{bmatrix} - \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \Delta x_3 \\ \vdots \end{bmatrix}$$

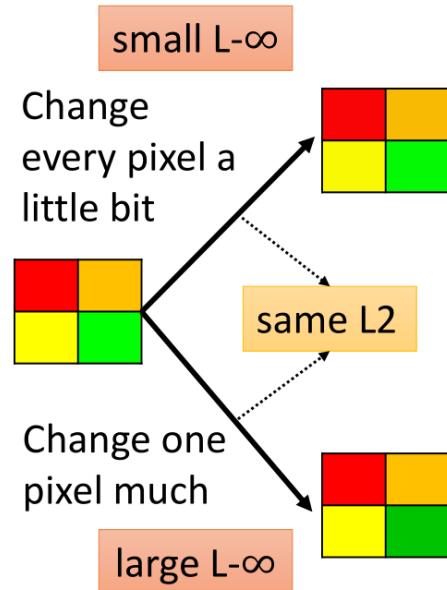
$$x' \quad x^0 \quad \Delta x$$

- L2-norm

$$\begin{aligned} d(x^0, x') &= \|x^0 - x'\|_2 \\ &= (\Delta x_1)^2 + (\Delta x_2)^2 + (\Delta x_3)^2 \dots \end{aligned}$$

- L-infinity

$$\begin{aligned} d(x^0, x') &= \|x^0 - x'\|_\infty \\ &= \max\{\Delta x_1, \Delta x_2, \Delta x_3, \dots\} \end{aligned}$$



How to Attack

损失函数的梯度为 (此时变量为 x)

$$\nabla L(x) = \begin{bmatrix} \frac{\partial L(x)}{\partial x_1} \\ \frac{\partial L(x)}{\partial x_2} \\ \frac{\partial L(x)}{\partial x_3} \\ \vdots \end{bmatrix}_{gradient}$$

Gradient Descent, 但是加上限制, 判断 x^t 是否符合限制, 如果不符合, 要修正 x^t

修正方法: 穷举 x^0 附近的所有 x , 用距离 x^t 最近的点来取代 x^t

How to Attack

Just like training a neural network,
but network parameter θ is
replaced with input x'

$$x^* = \arg \min_{d(x^0, x') \leq \varepsilon} L(x')$$

- Gradient Descent (Modified Version)

Start from original image x^0

For $t = 1$ to T

$$x^t \leftarrow x^{t-1} - \eta \nabla L(x^{t-1})$$

If $d(x^0, x^t) > \varepsilon$

$$x^t \leftarrow fix(x^t)$$

`def fix(x^t)`

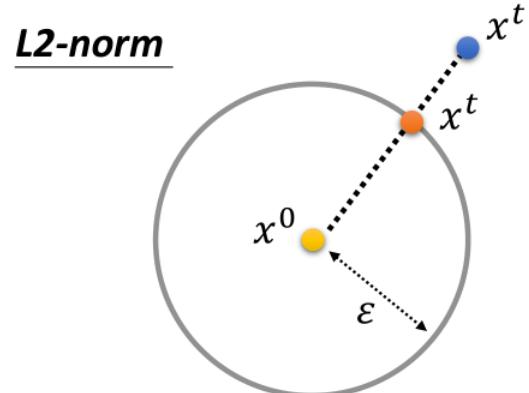
For all x fulfill
 $d(x^0, x) \leq \varepsilon$
Return the one
closest to x^t

How to Attack

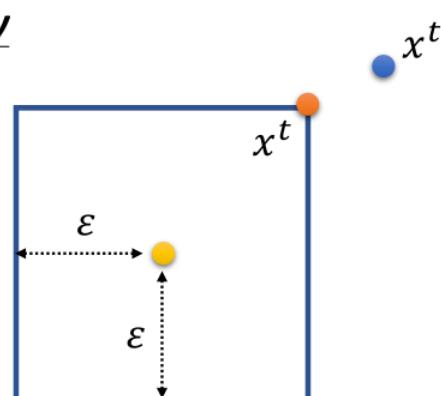
`def fix(x^t)`

For all x fulfill
 $d(x^0, x) \leq \varepsilon$

Return the one
closest to x^t



L-infinity



Example

使用一个猫，可以让它输出海星

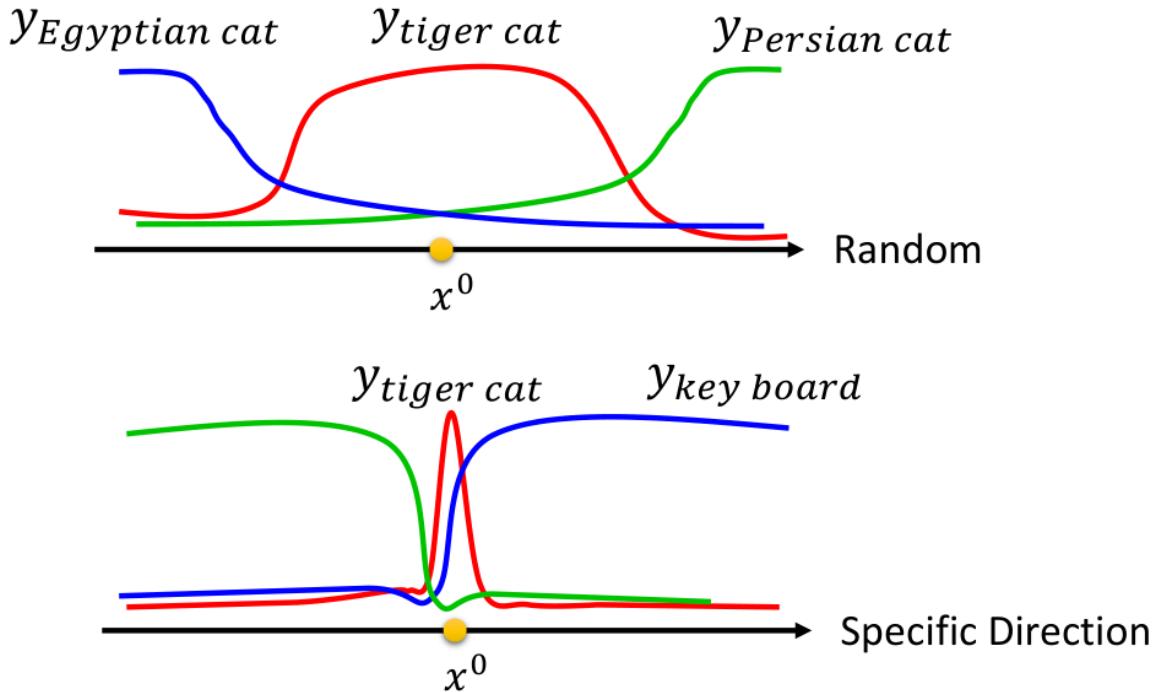
两张图片的不同很小，要相减后乘以50倍才能看出

是否是因为网络很弱呢？随机加噪声，但是对结果的影响不大；加的噪声逐渐增大以后会有影响

What happened?

高维空间中，随机方向与特定方向的 y 表现不同，因此一般的杂讯没有效果，特定的杂讯会有效果。

你可以想象 x^0 是在非常高维空间中的一个点，你把这个点随机的移动，你会发现多数情况下在这个点的附近很大一个范围内都是能正确辨识的空间，当你把这个点推到的很远的时候才会让机器识别成类似的事物，当你把它推的非常远时才会被辨识成不相关的事物。但是，有一些神奇维度，在这些神奇的维度上 x^0 的正确区域只有非常狭窄的一点点，我们只要将 x^0 推离原值一点点，就会让模型输出产生很大的变化。



这种现象不止存在于Deep Model中，一般的Model也会被攻击。

Attack Approaches

- FGSM
- Basic iterative method
- L-BFGS
- Deepfool
- JSMA
- C&W
- Elastic net attack
- Spatially Transformed
- One Pixel Attack
- only list a few

Fast Gradient Sign Method (FGSM)

攻击方法的主要区别在于Different optimization methods和Different constraints

FGSM是一种简单的model attack方法，梯度下降的时候计算的梯度，如果是负的，则直接为-1，如果是正的，则直接为+1。所以 x^0 的更新要么为 $-\varepsilon$ ，要么为 $+\varepsilon$ ，只更新一次就结束了。

这个算法的思想就是只攻击一次就好（减去或者加上 ε ）。多攻击几次确实会更好，所以FGSM有一个进阶方法Iterative fast gradient sign method (I-FGSM)

$$x^* = \arg \min_{d(x^0, x') \leq \varepsilon} L(x')$$

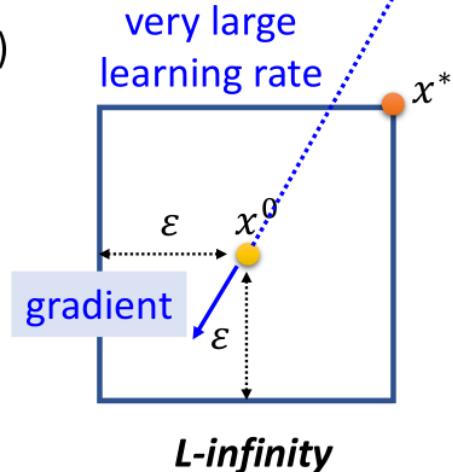
Different optimization methods
Different constraints

- Fast Gradient Sign Method (FGSM)

$$x^* \leftarrow x^0 - \varepsilon \Delta x$$

$$\Delta x = \begin{bmatrix} sign(\partial L / \partial x_1) \\ sign(\partial L / \partial x_2) \\ sign(\partial L / \partial x_3) \\ \vdots \end{bmatrix}$$

only have +1 or -1



FGSM只在意gradient的方向，不在意它的大小。假设constraint用的是L-infinity， FGSM相当于设置了一个很大的learning rate，导致可以马上跳出范围，再拉回来。

White Box v.s. Black Box

In the previous attack, we fix network parameters θ to find optimal x' .

To attack, we need to know network parameters θ

- This is called **White Box Attack**.

Are we safe if we do not release model?

- You cannot obtain model parameters in most on-line API.

No, because **Black Box Attack** is possible.

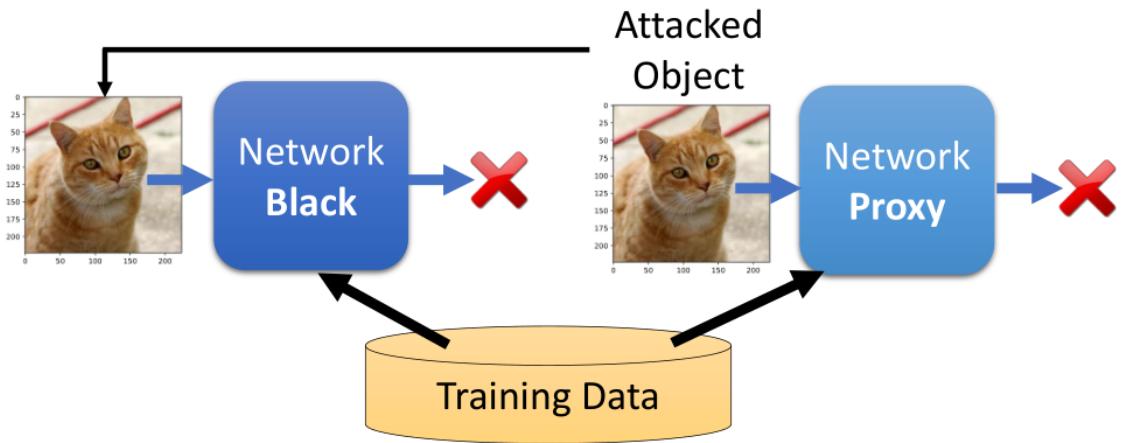
Black Box Attack

If you have the **training data** of the target network

- Train a **proxy network** yourself
- Using the proxy network to generate attacked objects

Otherwise, obtaining input-output pairs from target network

如图中描述的是模型辨识正确的概率，也就是攻击失败的概率。上述五种神经网络的架构是不一样的，但是我们可以看到即使是不同架构的模型攻击成功的概率也是非常高的，而相同的架构的模型攻击成功率则明显是更高的。



Black

Proxy

	ResNet-152	ResNet-101	ResNet-50	VGG-16	GoogLeNet
ResNet-152	4%	13%	13%	20%	12%
ResNet-101	19%	4%	11%	23%	13%
ResNet-50	25%	19%	5%	25%	14%
VGG-16	20%	16%	15%	1%	7%
GoogLeNet	25%	25%	17%	19%	1%

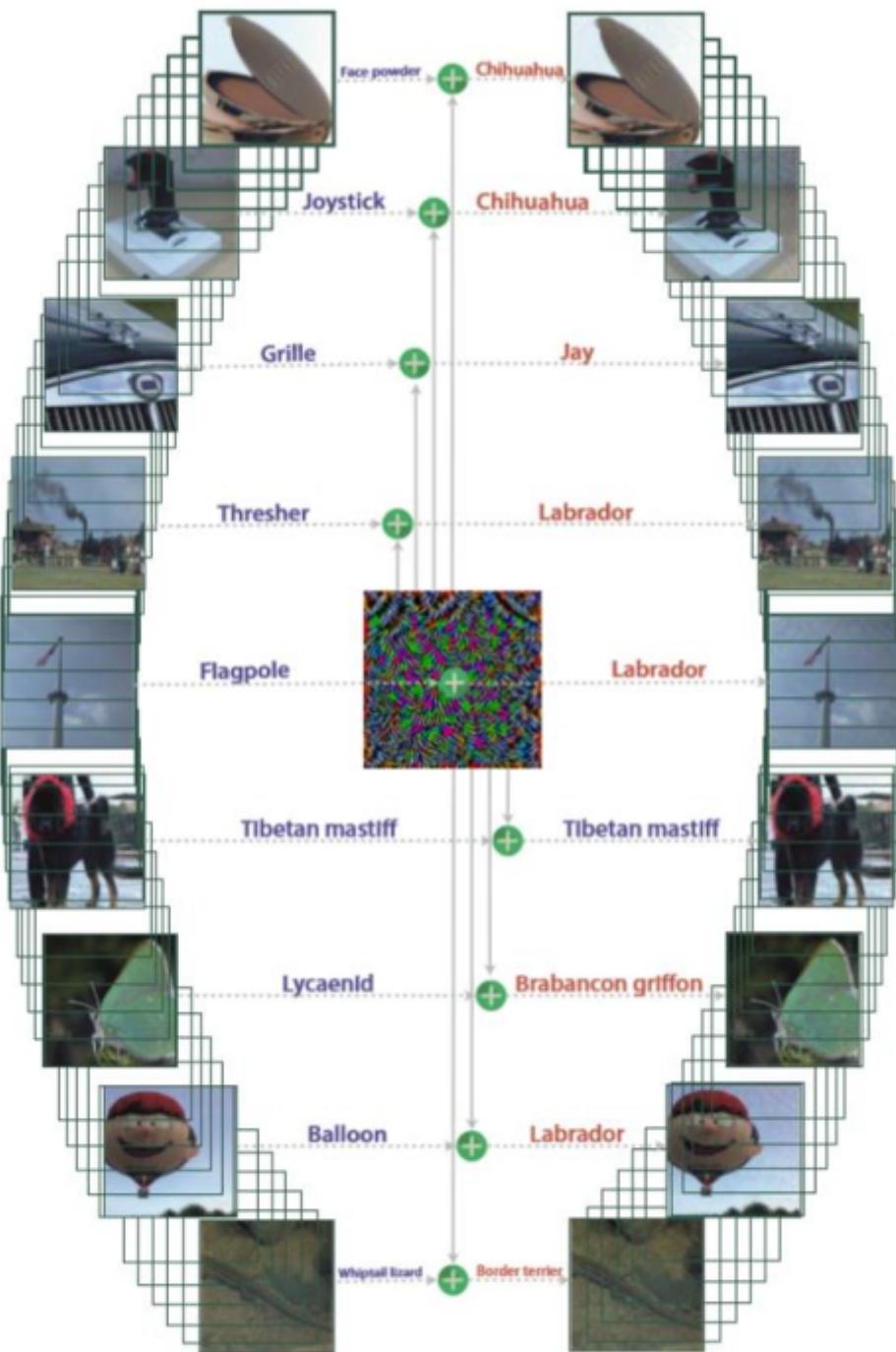
<https://arxiv.org/pdf/1611.02770.pdf>

Universal Adversarial Attack

核心精神是找一个通用的攻击向量，将其叠加到任意样本上都会让模型辨识出错。

<https://arxiv.org/abs/1610.08401>

这件事做成以后，你可以想象，只要在做辨识任务的摄像机前面贴一张噪声照片，就可以让所有结果都出错。另外，这个通用攻击甚至也可以做上述的黑盒攻击。

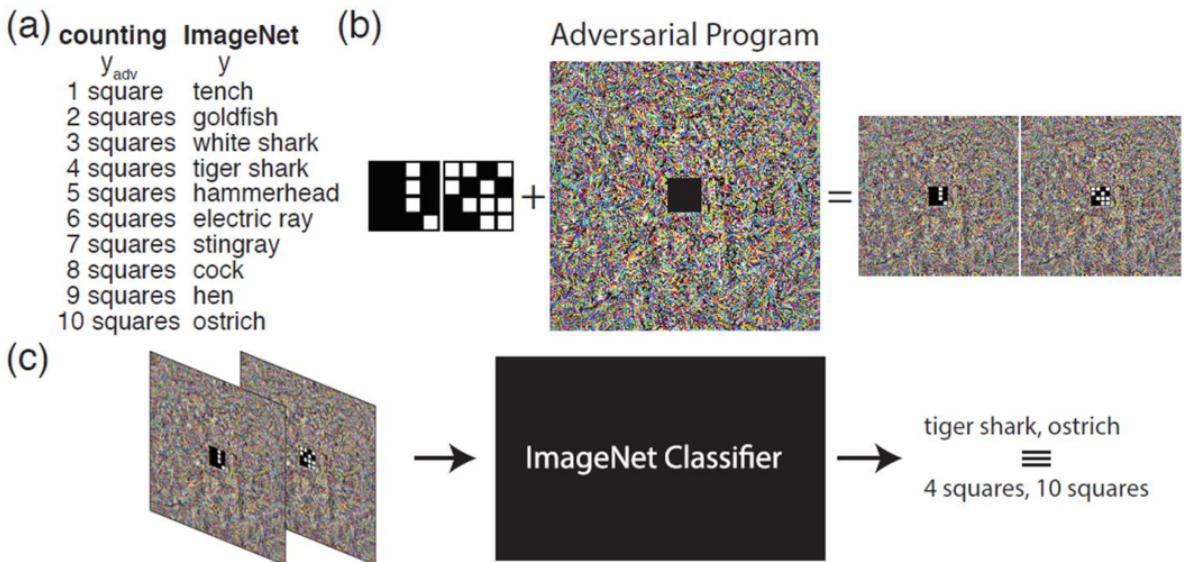


Black Box Attack is also possible!

Adversarial Reprogramming

这个攻击的核心精神是：通过找一些噪音，让机器的行为发生改变，达到重编程实现其他功能的效果。改变原来network想要做的事情，例如从辨识动物变成数方块

当我们把图片中间贴上上图中那种方块图，机器就会帮我们数出途中方块的个数，如果有一个方块会输出tench，有两个方块就输出goldfish... 这件事还挺神奇的，因为我们并没有改变机器的任何参数，我们只是用了和前述相同的方法，找了一个噪声图片，然后把要数方块的图贴在噪声上，输入模型就会让模型帮我们数出方块的个数。具体方法细节参考引用文章。



Gamaleldin F. Elsayed, Ian Goodfellow, Jascha Sohl-Dickstein, "Adversarial Reprogramming of Neural Networks", ICLR, 2019

Attack in the Real World

我们想知道上述的攻击方法是否能应用在现实生活中，上述的所有方法中加入的噪声其实都非常的小，在数字世界中这些噪声会对模型的判别造成很大影响似乎是很合理的，但是在真实的世界中，机器是通过一个小相机看世界的，这样的小噪声通过相机以后可能就没有了。通过镜头是否会识别到那些微小的杂讯呢？实验把攻击的图像打印出来，然后用摄像头识别。证明这种攻击时可以在现实生活中做到的。

对人脸识别上进行攻击，可以把噪声变成实体眼镜，带着眼镜就可以实现攻击。

需要确保：多角度都是成功的；噪声不要有非常大的变化（比如只有1像素与其他不同），要以色块方式呈现，这样方便摄像头能看清；不用打印机打不出来的颜色。

1. An attacker would need to find perturbations that generalize beyond a single image.
2. Extreme differences between adjacent pixels in the perturbation are unlikely to be accurately captured by cameras.
3. It is desirable to craft perturbations that are comprised mostly of colors reproducible by the printer

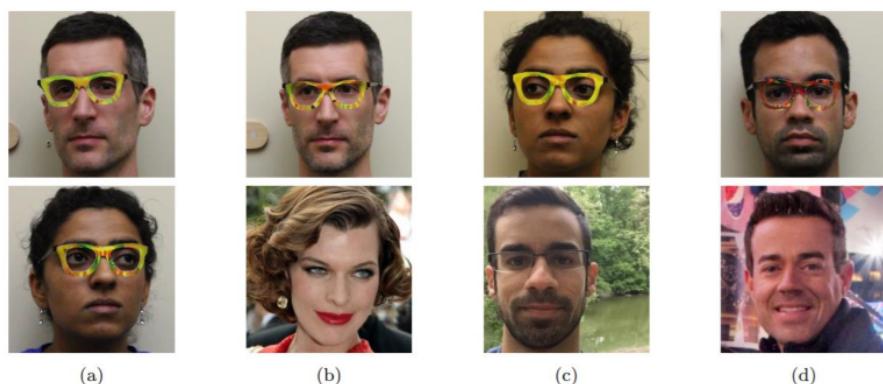


Figure 4: Examples of successful impersonation and dodging attacks. Fig. (a) shows S_A (top) and S_B (bottom) dodging against DNN_B . Fig. (b)–(d) show impersonations. Impersonators carrying out the attack are shown in the top row and corresponding impersonation targets in the bottom row. Fig. (b) shows S_A impersonating Milla Jovovich (by Georges Biard / CC BY-SA / cropped from <https://goo.gl/GlsWIC>); (c) S_B impersonating S_C ; and (d) S_C impersonating Carson Daly (by Anthony Quintano / CC BY / cropped from <https://goo.gl/VfnDct>).

在不同角度变成限速45的标志

Distance/Angle	Subtle Poster	Subtle Poster Right Turn	Camouflage Graffiti	Camouflage Art (LISA-CNN)	Camouflage Art (GTSRB-CNN)
5' 0°					
5' 15°					
10' 0°					
https://arxiv.org/abs/1707.08945					
10' 30°					
40' 0°					
Targeted-Attack Success	100%	73.33%	66.67%	100%	80%

Beyond Images

You can attack audio

- https://nicholas.carlini.com/code/audio_adversarial_examples/
- <https://adversarial-attacks.net>

You can attack text

<https://arxiv.org/pdf/1707.07328.pdf>

Article: Super Bowl 50

Paragraph: “Peyton Manning became the first quarterback ever to lead two different teams to multiple Super Bowls. He is also the oldest quarterback ever to play in a Super Bowl at age 39. The past record was held by John Elway, who led the Broncos to victory in Super Bowl XXXIII at age 38 and is currently Denver’s Executive Vice President of Football Operations and General Manager. Quarterback Jeff Dean had jersey number 37 in Champ Bowl XXXIV.”

Question: “What is the name of the quarterback who was 38 in Super Bowl XXXIII?”

Original Prediction: John Elway

Prediction under adversary: Jeff Dean

Defense

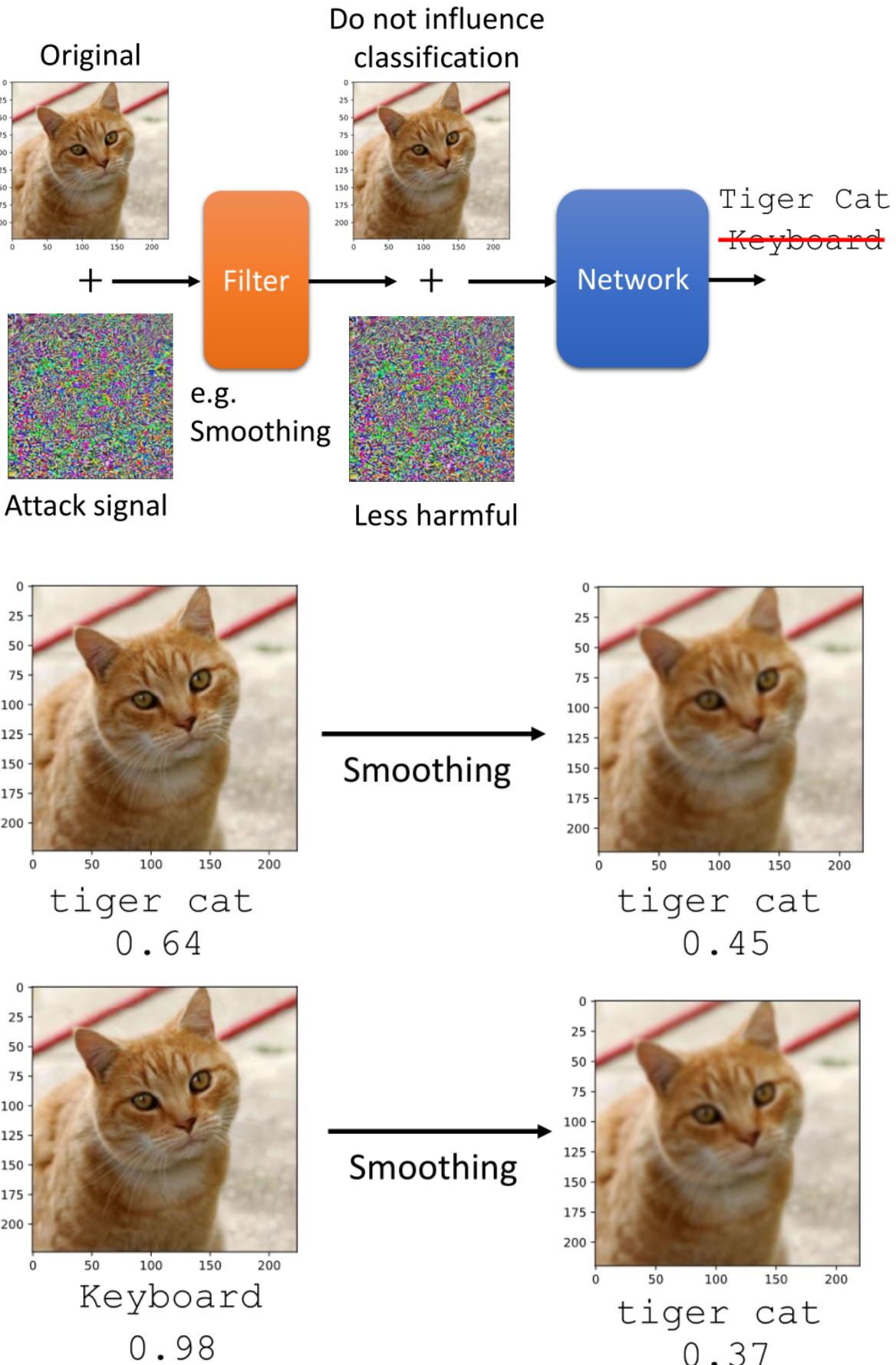
Adversarial Attack **cannot** be defended by weight regularization, dropout and model ensemble.

Two types of defense

- Passive defense: Finding the attached image without modifying the model
 - Special case of Anomaly Detection, 找出加入攻击的图片, 不修改网络
- Proactive defense: Training a model that is robust to adversarial attack

Passive Defense

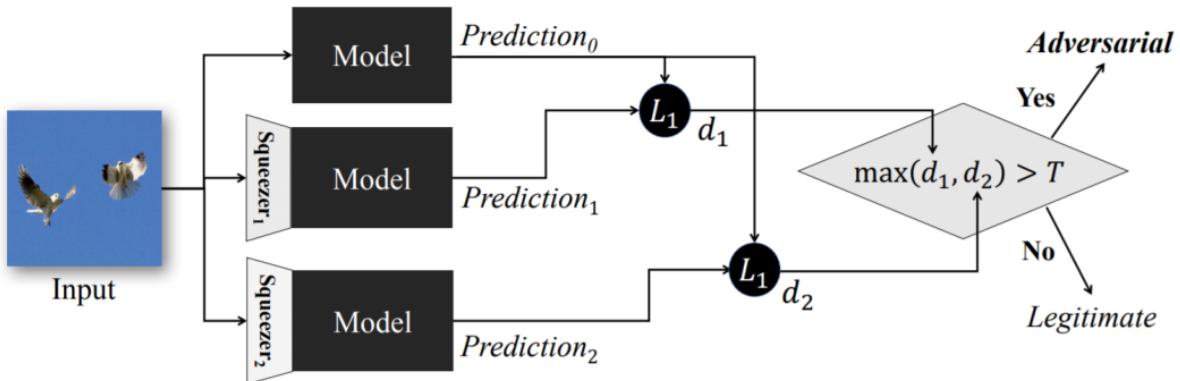
加入filter, 比如smoothing。



解释是，找攻击信号时，实际上只有某一种或者某几种攻击信号能够让攻击成功。虽然这种攻击信号能够让model失效，但是只有某几个方向上的攻击信号才能work。一旦加上一个filter，讯号改变了，攻击便失效。

Feature Squeeze

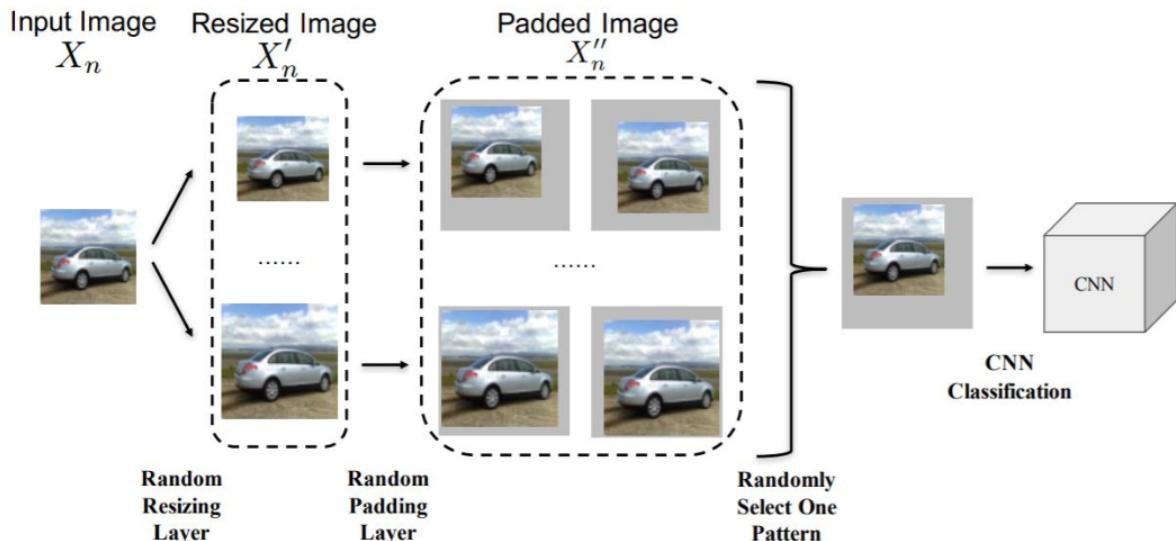
先用正常的model做一个prediction，再用model之前加上squeeze模块的model做一个prediction，如果这两个prediction差距很大，说明这个image被攻击过。



<https://arxiv.org/abs/1704.01155>

Randomization at Inference Phase

还可以在原图基础上做一点小小的缩放，一些小padding，让攻击杂讯与原来不太一样，让攻击杂讯失效。



<https://arxiv.org/abs/1711.01991>

但问题是这样在model之前加“盾牌”的方法，有一个隐患是，如果，“盾牌”的机制被泄露出去，那么攻击仍然很有可能成功。（把filter想象成network的第一个layer，再去训练攻击杂讯即可）

Proactive Defense

精神：训练NN时，找出漏洞，补起来。

假设train T个iteration，在每一个iteration中，利用attack algorithm找出每一张图片的attack image，在把这些attack image标上正确的label，再作为training data，加入训练。这样的方法有点像data augmentation。

为什么需要进行T个iteration？因为加入新的训练数据后，NN的结构改变，会有新的漏洞出现。

This method would stop algorithm A, but is still vulnerable for algorithm B.

Defense今天仍然是个很困难，尚待解决的问题。

Given training data $X = \{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$

Using X to train your model

For $t = 1$ to T

For $n = 1$ to N 找出漏洞

This method would stop
algorithm A, but is still
vulnerable for algorithm B.

Find adversarial input \tilde{x}^n given x^n
by an attack algorithm

Using algorithm A

We have new training data different in each iteration

$X' = \{(\tilde{x}^1, \hat{y}^1), (\tilde{x}^2, \hat{y}^2), \dots, (\tilde{x}^N, \hat{y}^N)\}$ Data Augmentation

Using both X' to update your model 把洞補起來

##

Network Compression

Network Compression

由于未来我们的模型有可能要运行在很多类似手机，手表，智能眼镜，无人机上，这些移动终端的算力和存储空间有限，因此要对模型进行压缩。当然也可以根据具体的硬件平台定制专门的模型架构，但这不是本课的重点。

Network Pruning

Network can be pruned

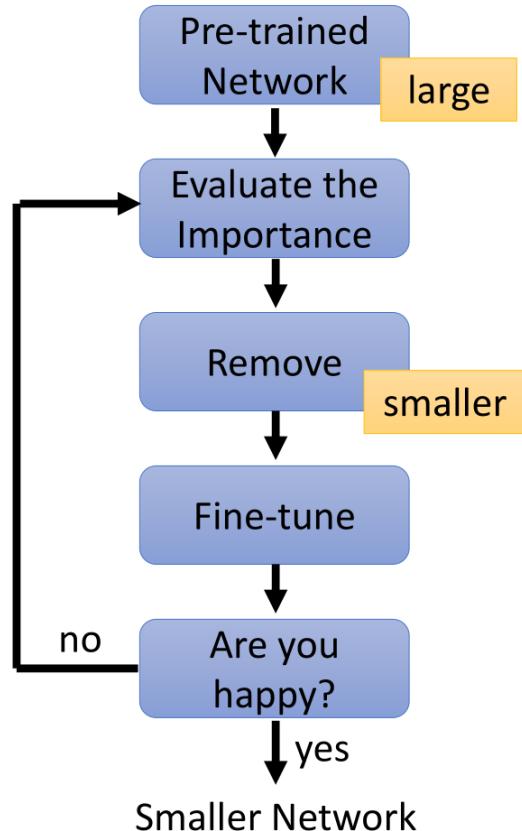
Networks are typically over-parameterized (there is significant redundant weights or neurons)

Prune them!

先要训练一个很大的模型，然后评估出重要的权重或神经元，移除不重要的权重或神经元，After pruning, the accuracy will drop (hopefully not too much)，要对处理后的模型进行微调，进行recovery把移除的损伤拿回来，若不满意就循环到第一步。注意：Don't prune too much at once, or the network won't recover.

Network Pruning

- Importance of a weight:
L1, L2
- Importance of a neuron:
the number of times it wasn't zero on a given data set
- After pruning, the accuracy will drop (hopefully not too much)
- Fine-tuning on training data for recover
- Don't prune too much at once, or the network won't recover.



怎么判断哪些参数是冗余或者不重要的呢?

- 对权重(weight)而言，我们可以通过计算它的L1、L2值来判断重要程度
- 对neuron而言，我们可以给出一定的数据集，然后查看在计算这些数据集的过程中neuron参数为0的次数，如果次数过多，则说明该neuron对数据的预测结果并没有起到什么作用，因此可以去除。

Why Pruning?

How about simply train a smaller network?

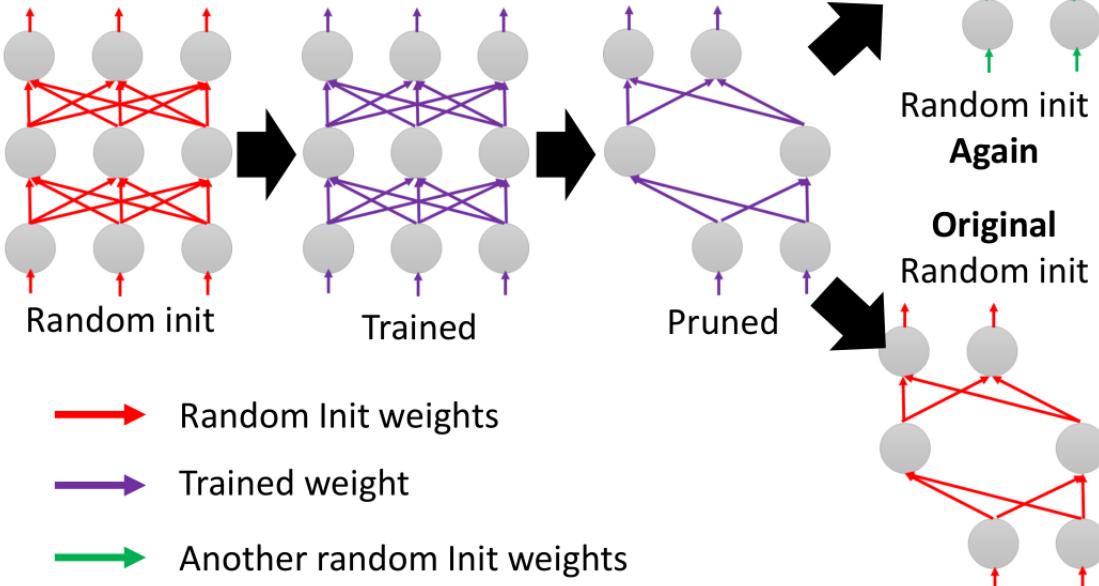
- It is widely known that smaller network is more difficult to learn successfully.
 - Larger network is easier to optimize? 更容易找到global optimum
https://www.youtube.com/watch?v=_VuWvQUMQVk
- Lottery Ticket Hypothesis
<https://arxiv.org/abs/1803.03635>

首先看最左边的网络，它表示大模型，我们随机初始化它权重参数（红色）。然后我们训练这个大模型得到训练后的模型以及权重参数（紫色）。最后我们对训练好的大模型做pruning得到小模型。作者把小模型拿出来后随机初始化参数（绿色，右上角），结果发现无法训练。然后他又把最开始的大模型的随机初始化的weight复制到小模型上（即把对应位置的权重参数复制过来，右下角）发现可以train出好的结果。

就像我们买彩票，买的彩票越多，中奖的机率才会越大。而最开始的大模型可以看成是由超级多的小模型组成的，也就是对大模型随机初始化参数会得到各种各样的初始化参数的小模型，有的小模型可能train不起来，但是有的就可以。大的Network比较容易train起来是因为，其中只要有小的Network可以train起来，大的Network就可以train起来。对大的Network做pruning，得到了可以train起来的小Network，因此它的初始参数是好的参数，用此参数去初始化，就train起来了。

Why Pruning?

Lottery Ticket Hypothesis



- Rethinking the Value of Network Pruning

这个文献提出了不同见解，其实直接train小的network也可以得到好的结果，无需初始化参数。Scratch-B比Scratch-E训练epoch多一些，可以看到比微调的结果要好。

- <https://arxiv.org/abs/1810.05270>

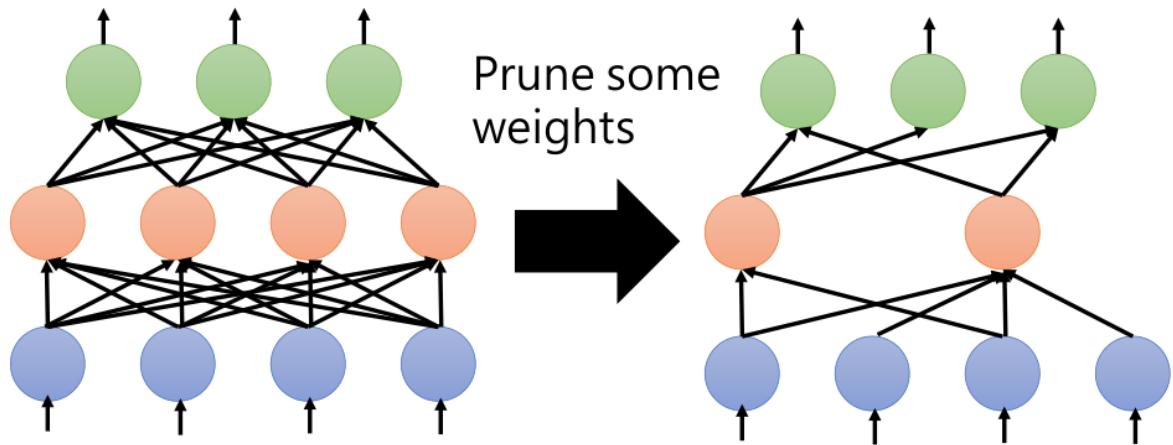
Dataset	Model	Unpruned	Pruned Model	Fine-tuned	Scratch-E	Scratch-B
CIFAR-10	VGG-16	93.63 (± 0.16)	VGG-16-A	93.41 (± 0.12)	93.62 (± 0.11)	93.78 (± 0.15)
	ResNet-56	93.14 (± 0.12)	ResNet-56-A	92.97 (± 0.17)	92.96 (± 0.26)	93.09 (± 0.14)
	ResNet-110	93.14 (± 0.24)	ResNet-110-A	93.14 (± 0.16)	93.25 (± 0.29)	93.22 (± 0.22)
ImageNet	ResNet-34	73.31	ResNet-34-A	72.56	72.77	73.03
			ResNet-34-B	72.29	72.55	72.91

- Real random initialization, not original random initialization in “Lottery Ticket Hypothesis”
- Pruning algorithms could be seen as performing network architecture search

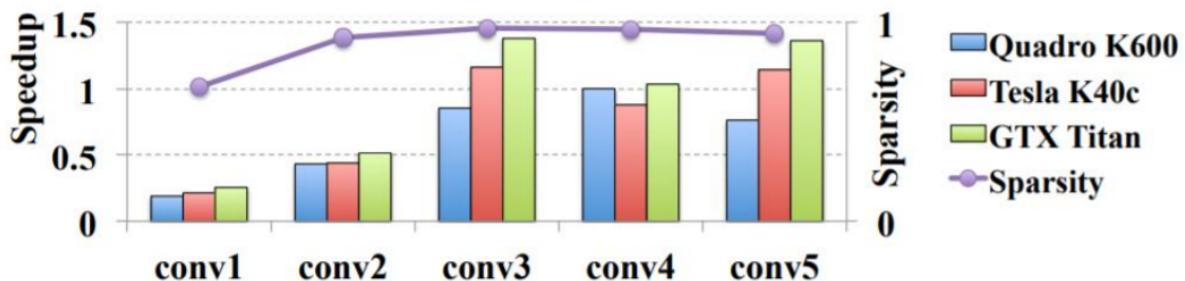
Practical Issue

Weight pruning

- The network architecture becomes irregular.
- Hard to implement, hard to speedup
- 可以补0，但是这种方法使得算出来Network的大小与原Network的大小一样。



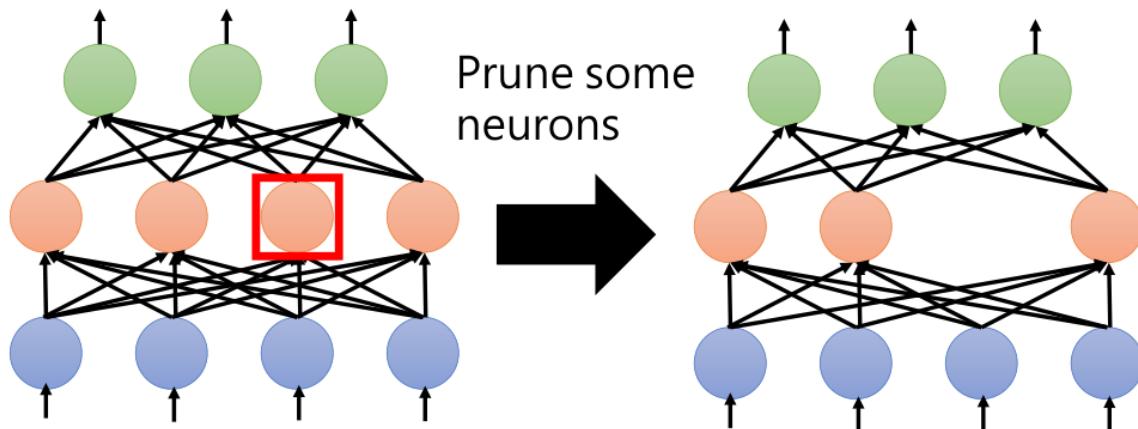
- 模型参数去掉了95%，最后的准确率只降低了2%左右，说明weight pruning的确能够在保证模型准确率的同时减少了模型大小，但是由于模型不规则，GPU加速没有效率，模型计算速度并没有提速，甚至有时使得速度降低了。



<https://arxiv.org/pdf/1608.03665.pdf>

Neuron pruning

- The network architecture is regular.
- Easy to implement, easy to speedup

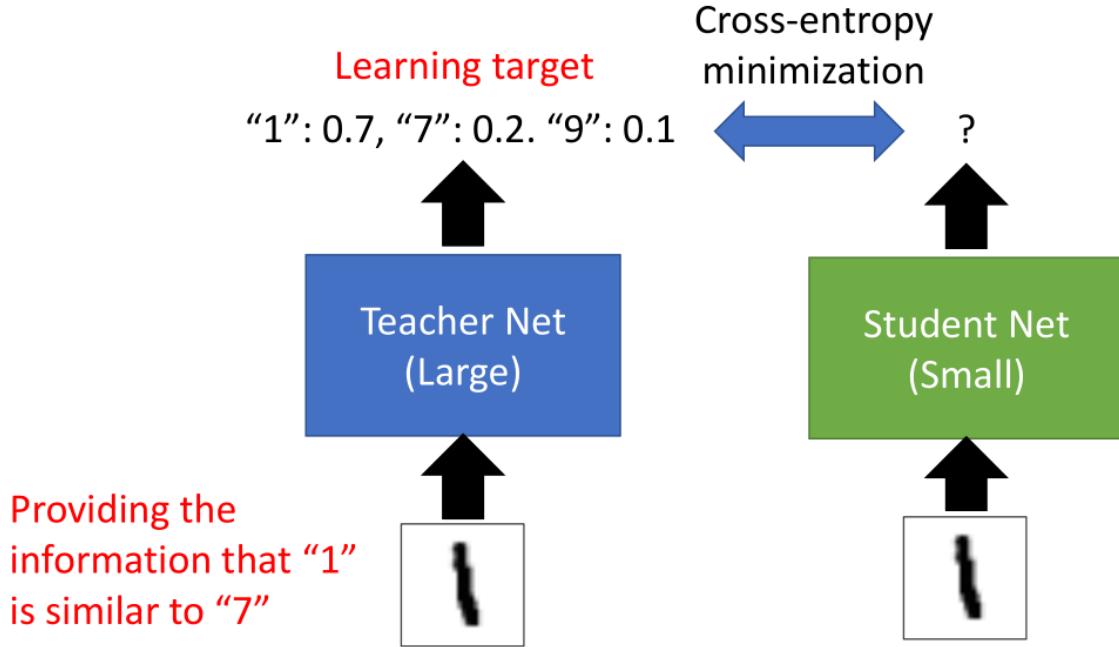


Knowledge Distillation

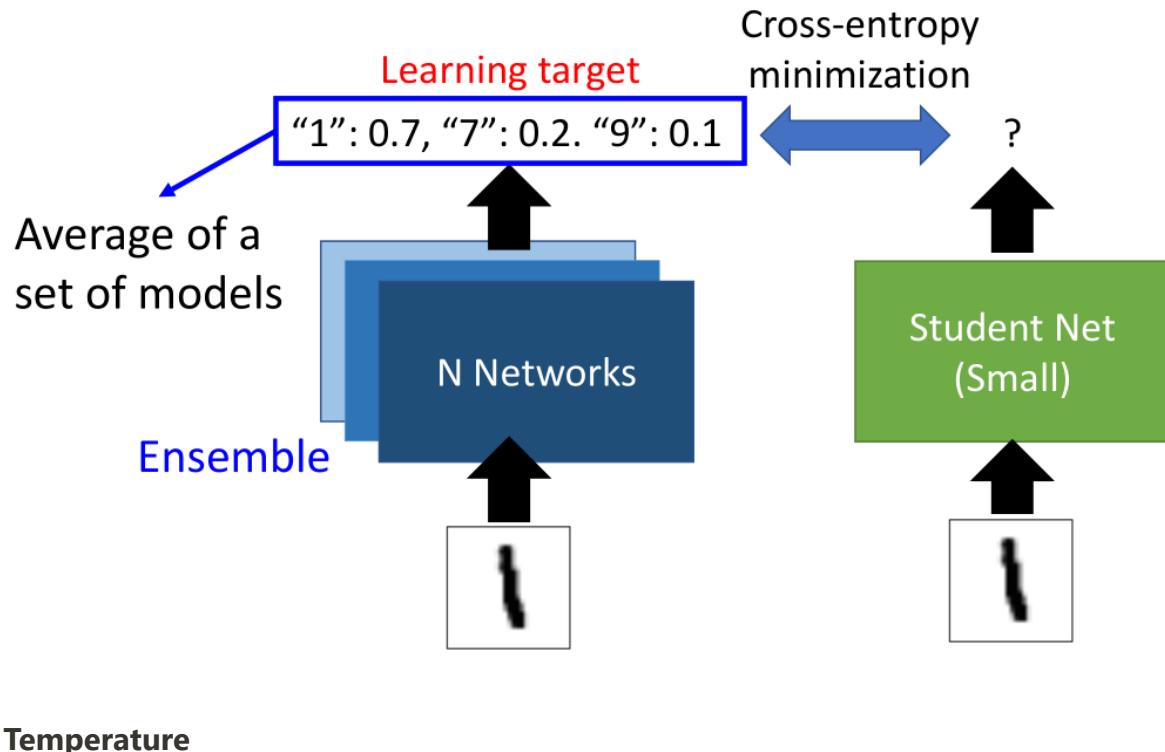
我们可以使用一个student network来学习teacher network的输出分布，并计算两者之间的cross-entropy，使其最小化，从而可以使两者的输出分布相近。teacher提供了比label data更丰富的资料，比如teacher net不仅给出了输入图片和1很像的结果，还说明了1和7长得像，1和9长得像；所以，student跟着teacher net学习，是可以得到更多的information的。

Knowledge Distillation

Knowledge Distillation
<https://arxiv.org/pdf/1503.02531.pdf>
Do Deep Nets Really Need to be Deep?
<https://arxiv.org/pdf/1312.6184.pdf>



可以让多个老师出谋划策来教学生，即通过Ensemble来进一步提升预测准确率。



$$y_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \quad T = 100 \quad \rightarrow \quad y_i = \frac{\exp(x_i/T)}{\sum_j \exp(x_j/T)}$$

$x_1 = 100$	$y_1 = 1$	$x_1/T = 1$	$y_1 = 0.56$
$x_2 = 10$	$y_2 \approx 0$	$x_2/T = 0.1$	$y_2 = 0.23$
$x_3 = 1$	$y_3 \approx 0$	$x_3/T = 0.01$	$y_3 = 0.21$

在多类别分类任务中，我们用到的是softmax来计算最终的概率，但是这样有一个缺点，因为使用了指数函数，如果在使用softmax之前的预测值是 $x_1 = 100, x_2 = 10, x_3 = 1$ ，那么使用softmax之后三者对应的概率接近于 $y_1 = 1, y_2 = 0, y_3 = 0$ ，这样小模型没有学到另外两个分类的信息，和直接学习label没有区别。

引入了一个新的参数Temperature，通过T把y的差距变小了，导致各个分类都有机率，小模型学习的信息就丰富了。T需要自行调整。

Parameter Quantization

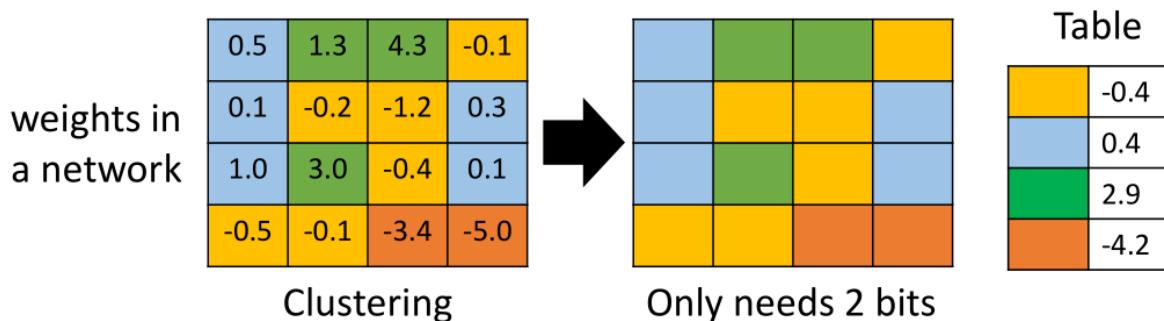
Using less bits to represent a value

- 一个很直观的方法就是使用更少bit来存储数值，例如一般默认是32位，那我们可以用16或者8位来存数据。

Weight clustering

- 最左边表示网络中正常权重矩阵，之后我们对这个权重参数做聚类（比如用kmean），比如最后得到了4个聚类，那么为了表示这4个聚类我们只需要2个bit，即用00,01,10,11来表示不同聚类。之后每个聚类的值就用均值来表示。这样的缺点是误差可能会比较大。每个参数不用保存具体的数值，而是只需要保存参数所属的类别即可。

- 1. Using less bits to represent a value
- 2. Weight clustering



- 3. Represent frequent clusters by less bits, represent rare clusters by more bits
 - e.g. Huffman encoding

Represent frequent clusters by less bits, represent rare clusters by more bits

- Huffman Encoding
 - 对常出现的聚类用少一点的bit来表示，而那些很少出现的聚类就用多一点的bit来表示。

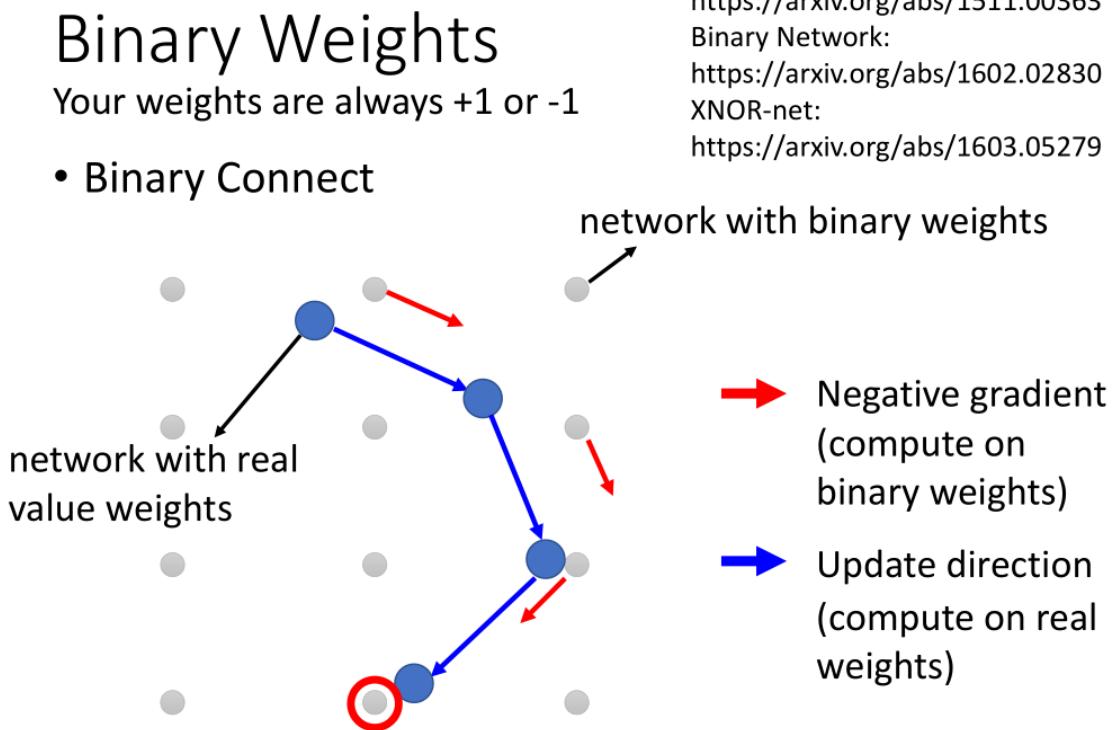
Binary Weights

- Binary Connect
 - 一种更加极致的思路来对模型进行压缩，Your weights are always +1 or -1
 - 简单介绍一下Binary Connect的思路，如下图示，灰色节点表示一组binary weights，蓝色节点是一组real value weights

我们先计算出和蓝色节点最接近的灰色节点，并计算出其梯度方向（红色剪头）。

蓝色节点按照红色箭头方向更新，而不是按照自身的梯度方向更新。

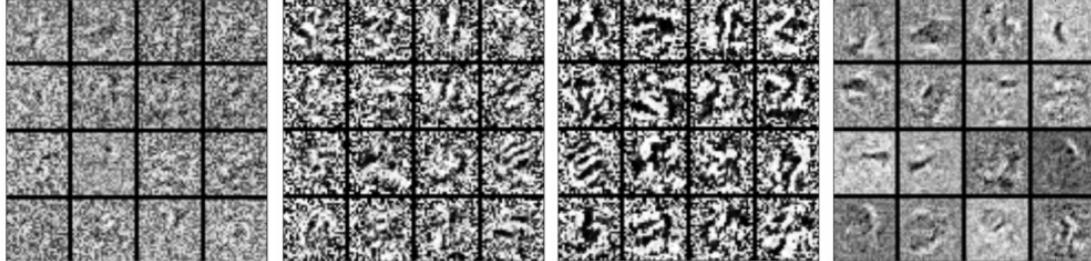
最后在满足一定条件后(例如训练至最大epoch)，蓝色节点会停在一个灰色节点附近，那么我们就使用该灰色节点的weights为Network的参数。



- Binary Connect的效果并没有因为参数只有1/-1而坏掉，相反，Binary Connect有点像regularization，虽然效果不如Dropout，但是比正常的Network效果还要稍微好点

Binary Connect

Method	MNIST	CIFAR-10	SVHN
No regularizer	$1.30 \pm 0.04\%$	10.64%	2.44%
BinaryConnect (det.)	$1.29 \pm 0.08\%$	9.90%	2.30%
BinaryConnect (stoch.)	$1.18 \pm 0.04\%$	8.27%	2.15%
50% Dropout	$1.01 \pm 0.04\%$		

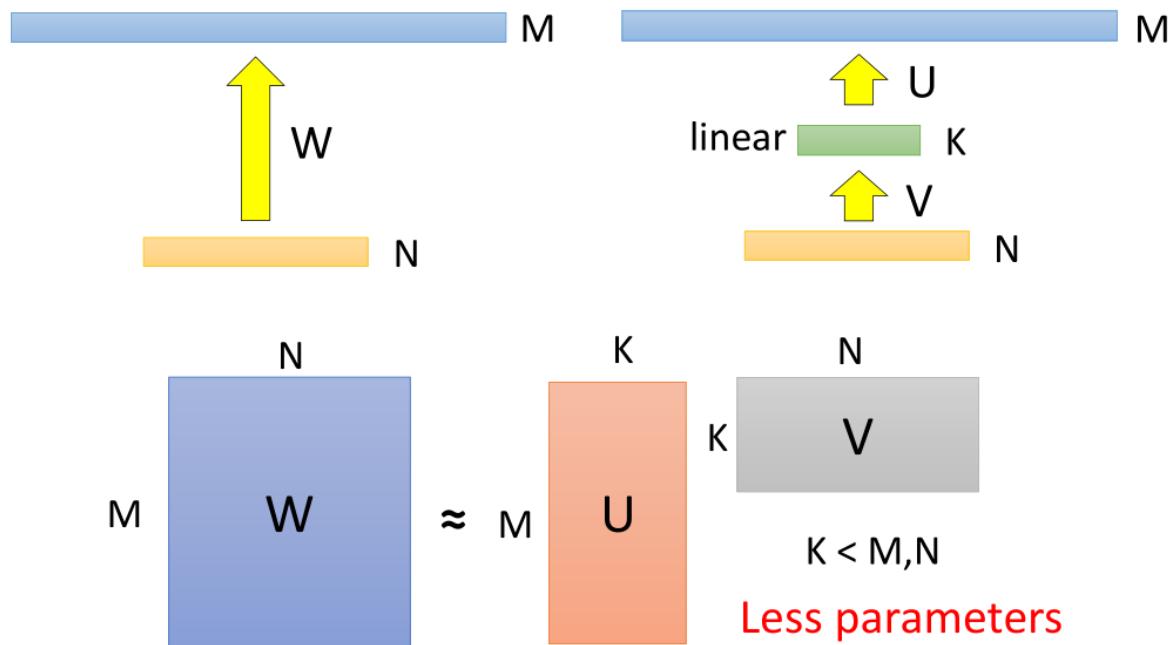


<https://arxiv.org/abs/1511.00363>

Architecture Design

Low rank approximation

下图是低秩近似的简单示意图，左边是一个普通的全连接层，可以看到权重矩阵大小为 $M \times N$ ，而低秩近似的原理就是在两个全连接层之间再插入一层 K 。很反直观，插入一层后，参数还能变少？没错，的确变少了，我们可以看到新插入一层后的参数数量为 $K \times (M+N)$ ，若让 K 不要太大，就可以减少 Network 的参数量。

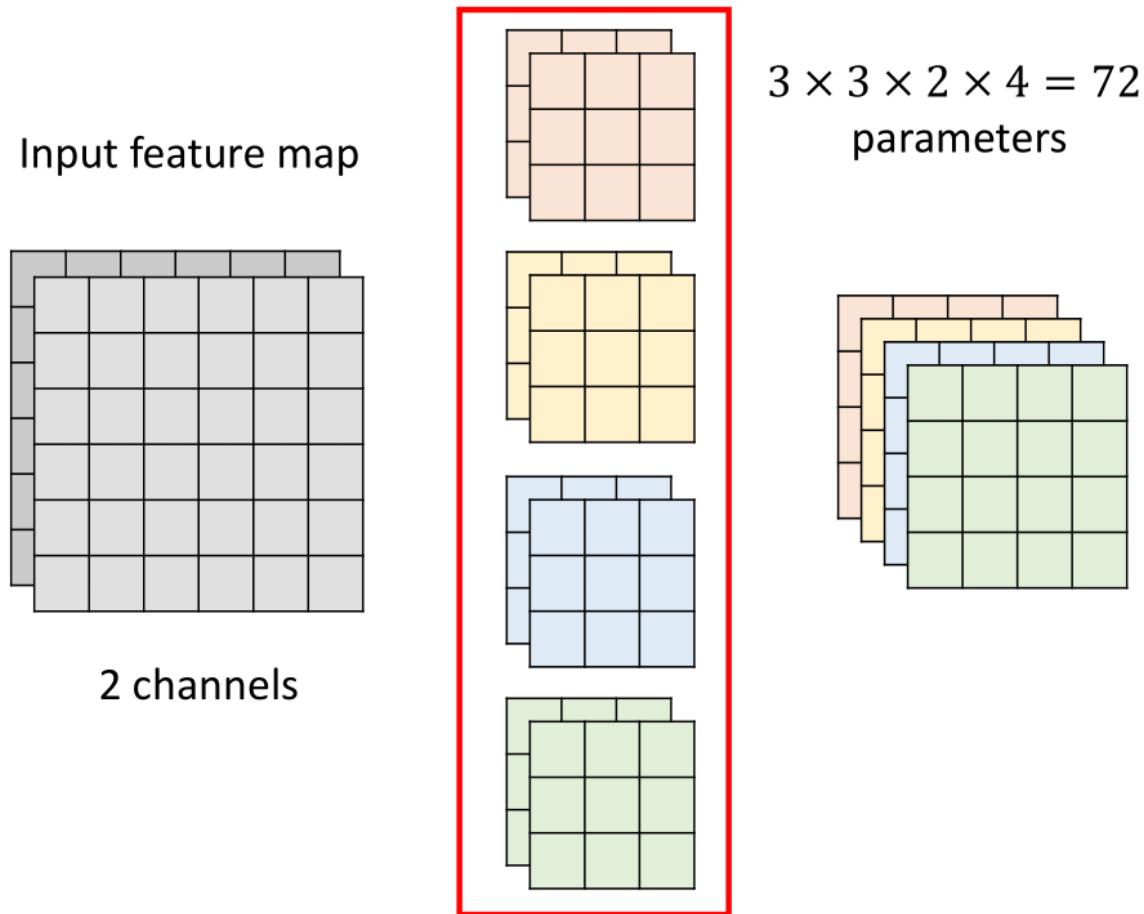


但是低秩近似之所以叫低秩，是因为原来的矩阵的秩最大可能是 $\min(M, N)$ ，而新增一层后可以看到矩阵 U 和 V 的秩都是小于等于 K 的，我们知道 $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$ ，所以相乘之后的矩阵的秩一定还是小于等于 K 。

因此会限制 Network 的参数，限制原来 Network 所做到的事情。

Standard CNN

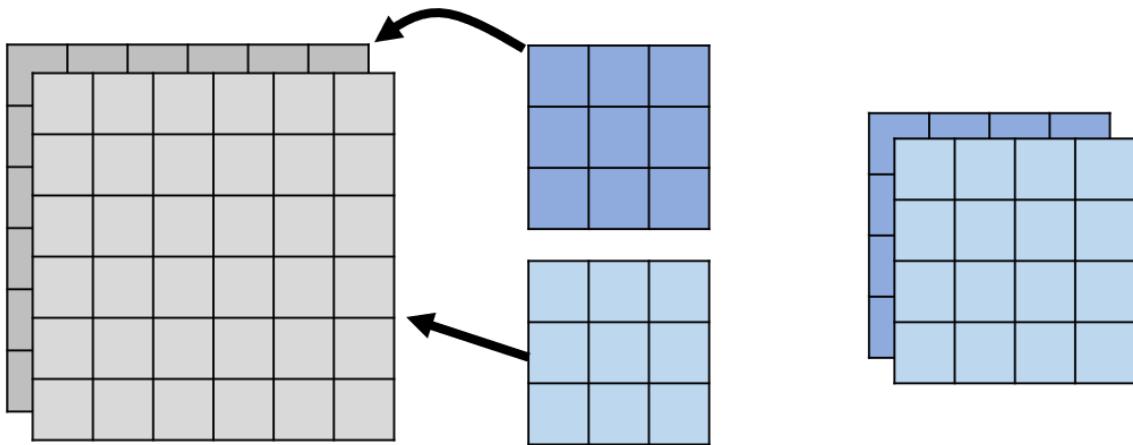
看一下标准卷积所需要的参数量。如下图示，输入数据由两个 6×6 的feature map组成，之后用4个大小为 3×3 的卷积核做卷积，最后的输出特征图大小为 $4 \times 4 \times 4$ 。每个卷积核参数数量为 $2 \times 3 \times 3 = 18$ ，所以总共用到的参数数量为 $4 \times 18 = 72$ 。



Depthwise Separable Convolution

Depthwise Convolution

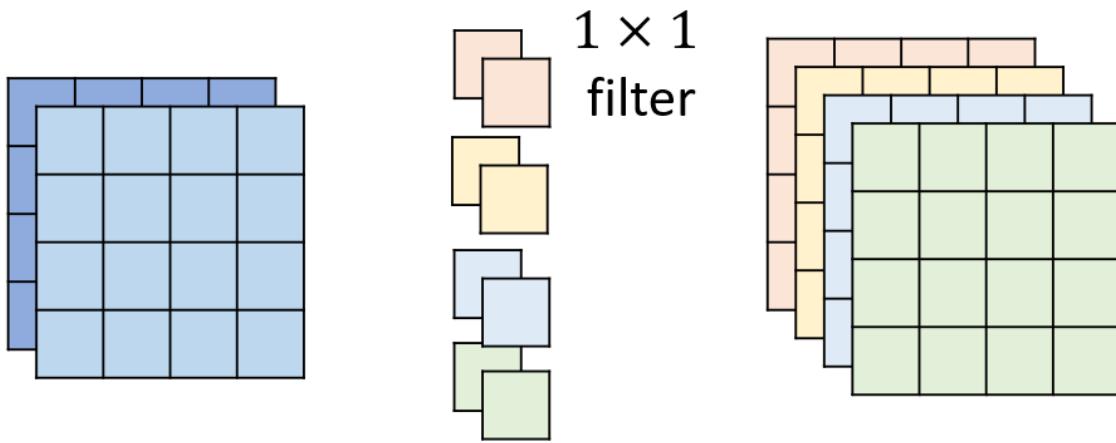
首先是输入数据的每个通道只由一个二维的卷积核负责，即卷积核通道数固定为1，这样最后得到的输出特征图等于输入通道。



- Filter number = Input channel number
- Each filter only considers one channel.
- The filters are $k \times k$ matrices
- There is no interaction between channels.

Pointwise Convolution

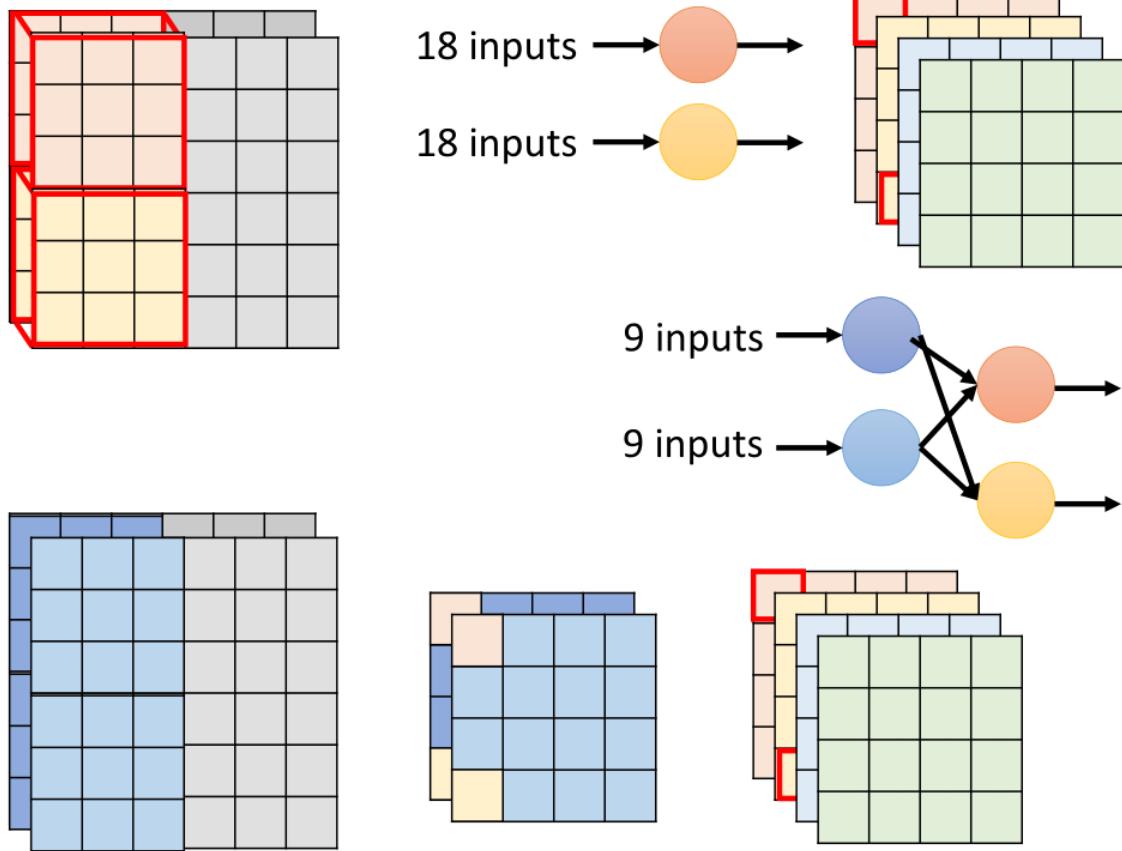
因为第一步得到的输出特征图是用不同卷积核计算得到的，所以不同通道之间是独立的，因此我们还需要对不同通道之间进行关联。为了实现关联，在第二步中使用了 1×1 大小的卷积核，通道数量等于输入数据的通道数量。另外 1×1 卷积核的数量等于预期输出特征图的数量，在这里等于4。最后我们可以得到和标准卷积一样的效果，而且参数数量更少， $3 \times 3 \times 2 = 18$ 、 $2 \times 4 = 8$ ，相比72个参数，合起来只用了26个参数。



比较

与上文中插入一层linear hidden layer的思想相似，在其中插入feature map，使得参数减少

把filter当成神经元，可以发现Depthwise Separable Convolution是对普通Convolution filter的拆解。共用第一层filter的参数，第二层才用不同的参数。即不同filter间共用同样的参数。



下面我们算一下标准卷积和Depthwise Separable卷积参数量关系

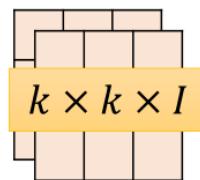
假设输入特征图通道数为 I , 输出特征图通道数为 O , 卷积核大小为 $k \times k$ 。

标准卷积参数量: $(k \times k \times I) \times O$

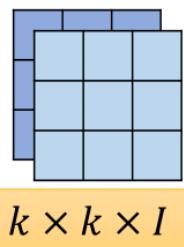
Depthwise Separable Convolution参数量: $k \times k \times I + I \times O$

两者相除, 一般来说 O 很大, 因此考虑后一项, $k=3$ 时, 参数量变成原来的 $\frac{1}{9}$

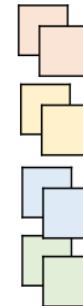
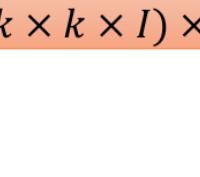
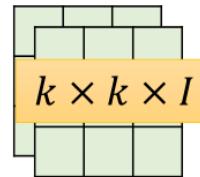
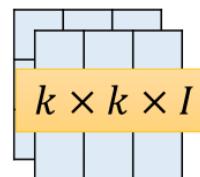
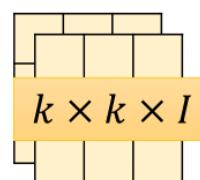
I : number of input channels



O : number of output channels



$k \times k$: kernel size



$I \times O$

$k \times k \times I + I \times O$

$(k \times k \times I) \times O$

$$\frac{k \times k \times I + I \times O}{k \times k \times I \times O}$$

$$= \frac{1}{O} + \frac{1}{k \times k}$$

To learn more

这样的设计广泛运用在各种号称比较小的网络上面

- SqueezeNet
 - <https://arxiv.org/abs/1602.07360>
- MobileNet
 - <https://arxiv.org/abs/1704.04861>
- ShuffleNet
 - <https://arxiv.org/abs/1707.01083>
- Xception
 - <https://arxiv.org/abs/1610.02357>

Dynamic Computation

Can network adjust the computation power it need? 该方法的主要思路是如果目前的资源充足（比如你的手机电量充足），那么算法就尽量做到最好，比如训练更久，或者训练更多模型等；反之，如果当前资源不够（如电量只剩10%），那么就先求有，再求好，先算出一个过得去的结果。

Possible Solutions

1. Train multiple classifiers

比如说我们提前训练多种网络，比如大网络，中等网络和小网络，那么我们就可以根据资源情况来选择不同的网络。但是这样的缺点是我们需要保存多个模型。

2. Classifiers at the intermedia layer

当资源有限时，我们可能只是基于前面几层提取到的特征做分类预测，但是一般而言这样得到的结果会打折扣，因为前面提取到的特征是比较细腻度的，可能只是一些纹理，而不是比较高层次抽象的特征。

左下角的图表就展示了不同中间层的结果比较，可以看到DenseNet和ResNet越靠近输入，预测结果越差。

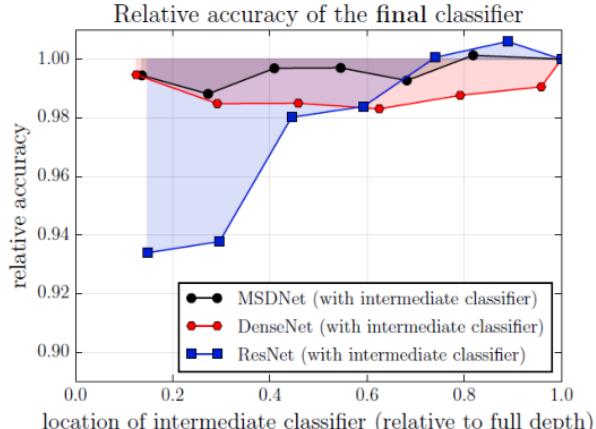
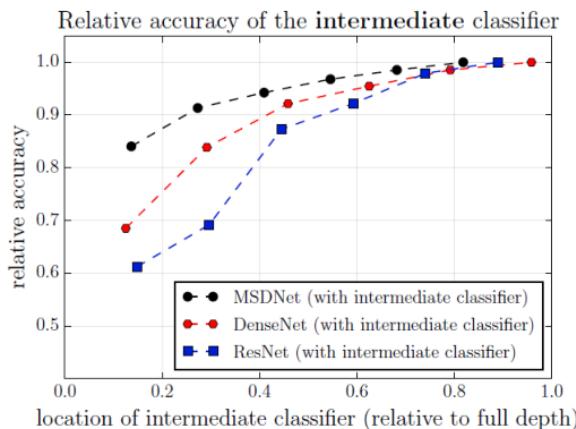
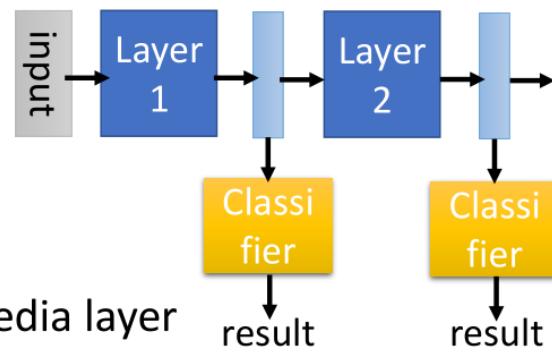
右下角的图则展示了在不同中间层插入分类器对于模型训练的影响，可以看到插入的分类器越靠近输入层，模型的性能越差。

因为一般而言，前面的网络结构负责提取浅层的特征，但是当我们在前面就插入分类器后，那么分类器为了得到较好的预测结果会强迫前面的网络结构提取一些复杂的特征，进而扰乱了后面特征的提取。

具体的解决方法可以阅读Multi-Scale Dense Networks

Possible Solutions

- 1. Train multiple classifiers
- 2. Classifiers at the intermedia layer



<https://arxiv.org/abs/1703.09844>

##

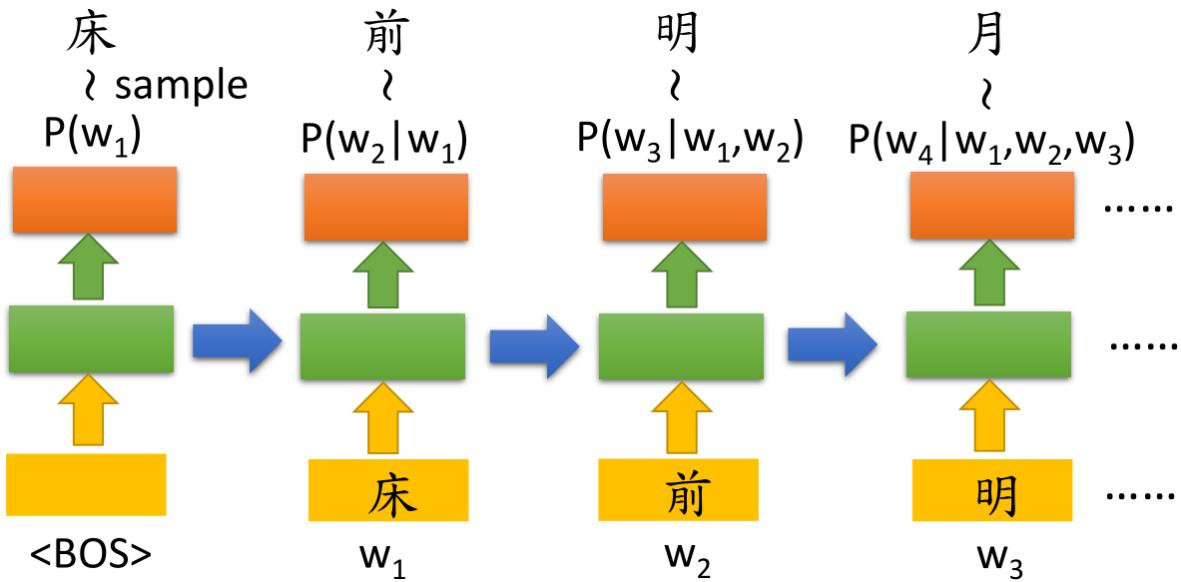
Seq2Seq

Conditional Generation by RNN & Attention

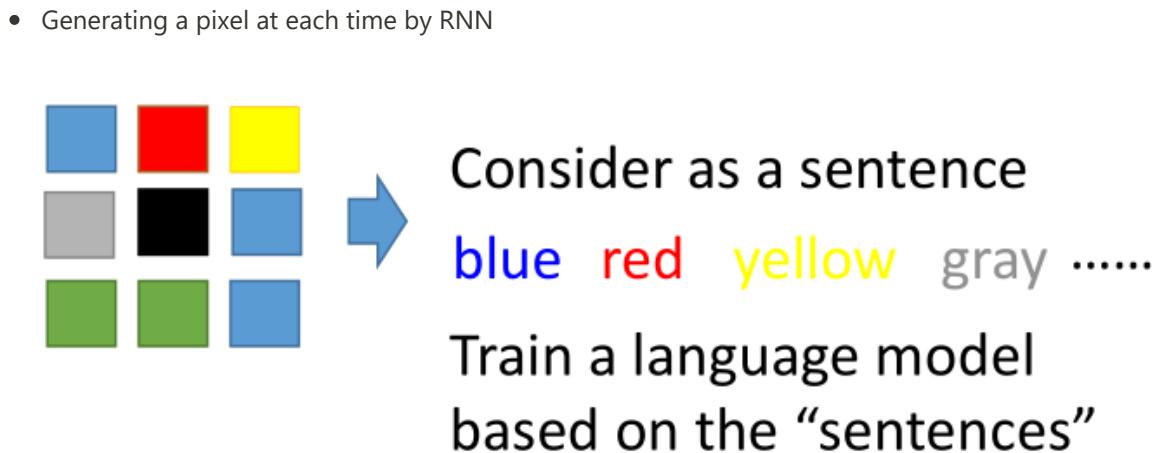
Generation

Sentences are composed of characters/words

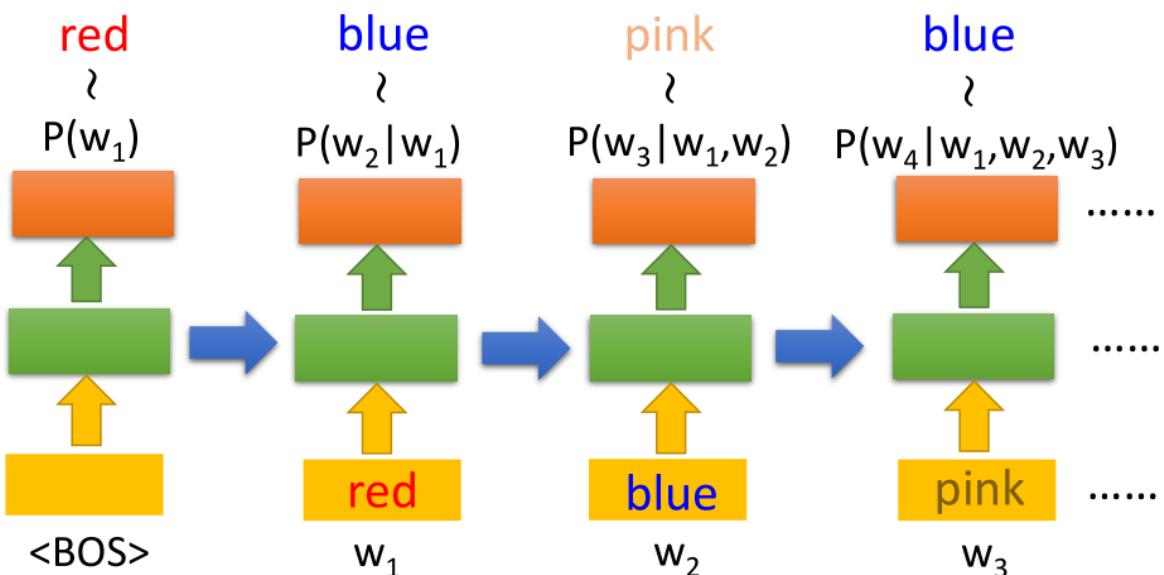
- 英文characters: a-z, word: 用空格分割；中文word: 有意义的最小单位，如“葡萄”，characters: 单字“葡”
- Generating a character/word at each time by RNN



Images are composed of pixels

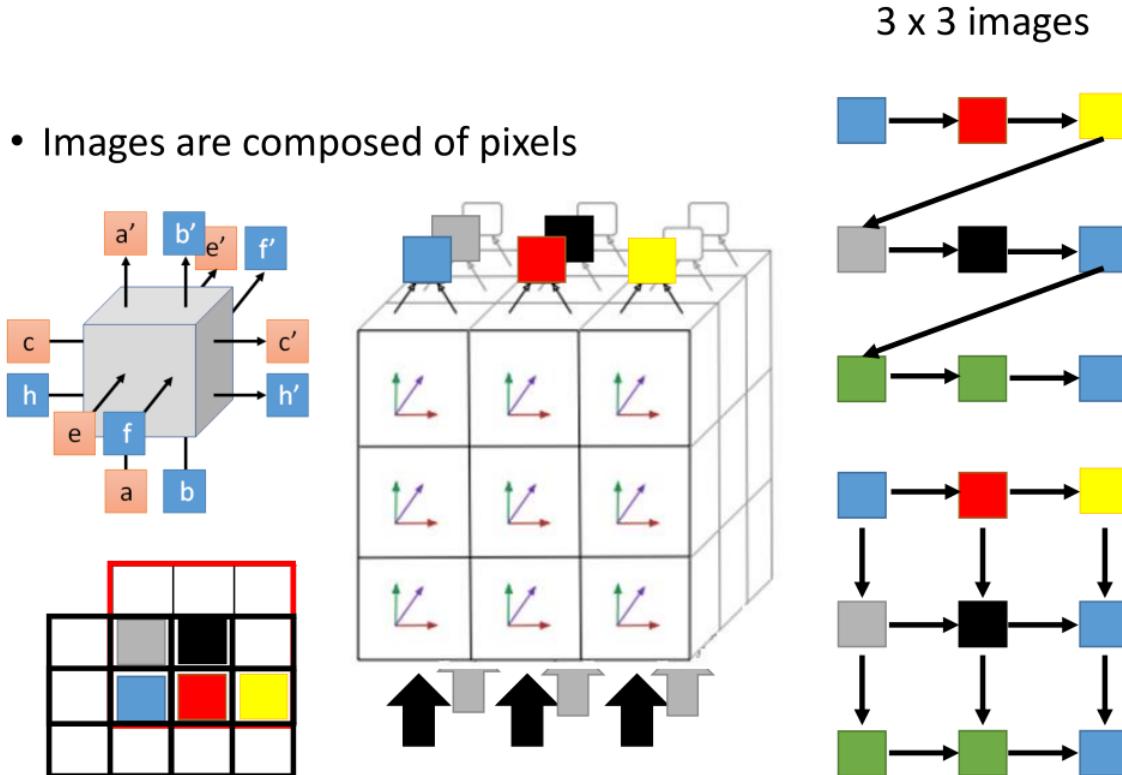


同样道理也可以用来生成一张照片，只要将每一个Pixel想成是一个Word，给模型一个BOS讯号，它就会开始生成颜色。



一般生成照片的时候如果单纯的按序生成可能会无法考量到照片之间的几何关系，但如果在生成Pixel的同时可以考量周围Pixel的话，那就可以有好的生成，可以利用3D Grid-LSTM

首先convolution filter在左下角计算，经过3D Grid-LSTM得到蓝色。filter右移一格，这时候会考虑蓝色，而3D Grid-LSTM的输入会往三个维度丢出，因此在计算第二个颜色的时候它会考虑到左边蓝色那排，得到红色。相同方式再一次得到黄色。filter往上一格移至左边起始点，同时会考量蓝色，才产生灰色。filter右移一格，这时候的filter计算涵盖了灰、蓝、红三个资讯，得到黑色。



其他例子

Image

- Aaron van den Oord, Nal Kalchbrenner, Koray Kavukcuoglu, Pixel Recurrent Neural Networks, arXiv preprint, 2016
- Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu, Conditional Image Generation with PixelCNN Decoders, arXiv preprint, 2016

Video

- Aaron van den Oord, Nal Kalchbrenner, Koray Kavukcuoglu, Pixel Recurrent Neural Networks, arXiv preprint, 2016

Handwriting

- Alex Graves, Generating Sequences With Recurrent Neural Networks, arXiv preprint, 2013

Speech

- Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu, WaveNet: A Generative Model for Raw Audio, 2016

Conditional Generation

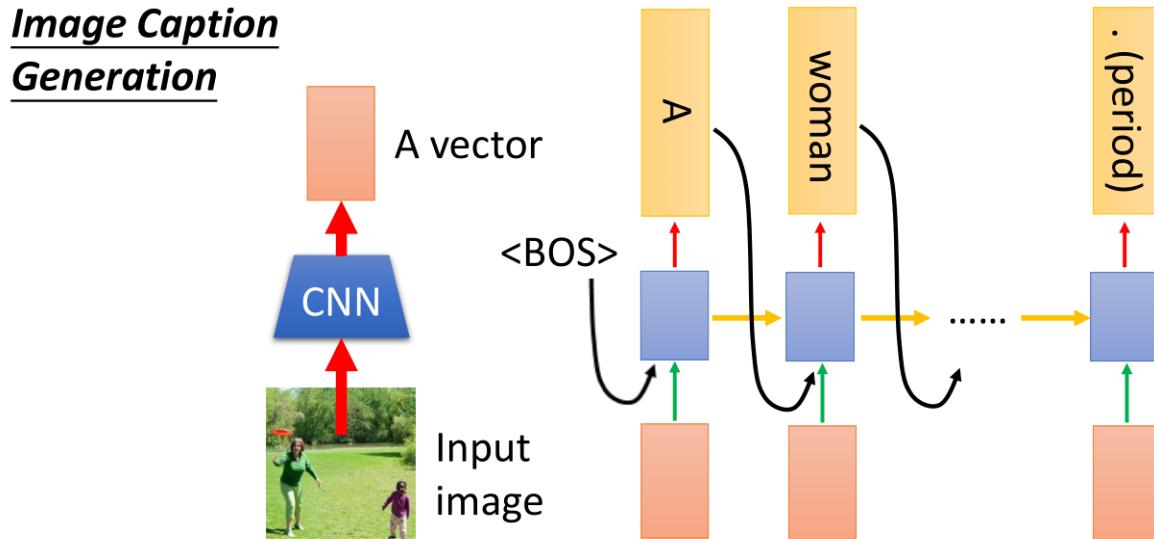
We don't want to simply generate some random sentences.

Generate sentences based on conditions.

单纯的利用RNN来产生句子那可能是不足的，因为它可以胡乱产生合乎文法的句子，因此我们希望模型可以根据某些条件来产生句子，也许给一张照片由机器来描述照片，或是像聊天机器人，给一个句子，机器回一个句子。

Represent the input condition as a vector, and consider the vector as the input of RNN generator.

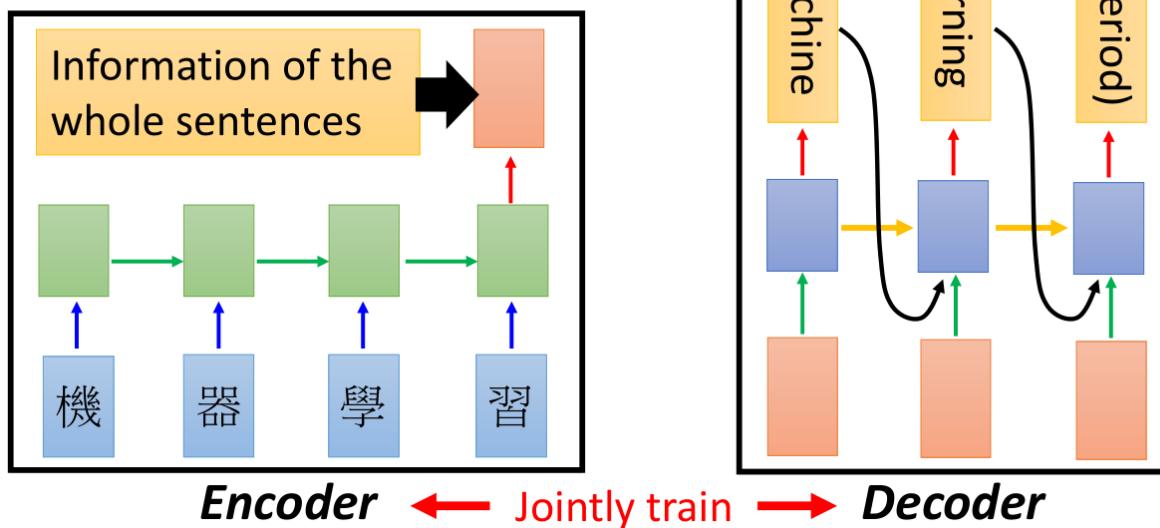
将照片通过一个CNN产生一个Vector，在每一个时间点都将该Vector输入RNN，这样子每次产生的Word都会被该照片的Vector给影响，就不会是胡乱产生句子了。



相同做法也可以应用在机器翻译与聊天机器人上，例如机器翻译中把“机器学习”这个句子表示成一个vector（先通过另一个RNN，抽出最后一个时间点Hidden Layer的Output）丢到一个可以产生英语句子的RNN里即可。

Represent the input condition as a vector, and consider the vector as the input of RNN generator

E.g. Machine translation / Chat-bot



前半部分称为Encoder，后半部分称为Decoder，两边可以结合一起训练参数，这种方式称为**Sequence-to-Sequence Learning**。资料量大时可以两边学习不同参数，资料量小时也可以共用参数（不容易过拟合）。

Need to consider longer context during chatting.

在聊天机器人中状况比较复杂，举例来说，机器人说了Hello之后，人类说了Hi，这时候机器人再说一次Hi就显的奇怪。因此机器必需要可以考虑比较长的资讯，它必需知道之前已经问过什么，或使用者说过什么。

我们可以再加入一个RNN来记忆对话，也就是双层的Encoder。首先，机器说的Hello跟人类回复的Hi都先变成一个code，接着第二层RNN将过去所有互动记录读一遍，变成一个code，再把这个结果做为后面Decoder的输入。

Attention

Dynamic Conditional Generation

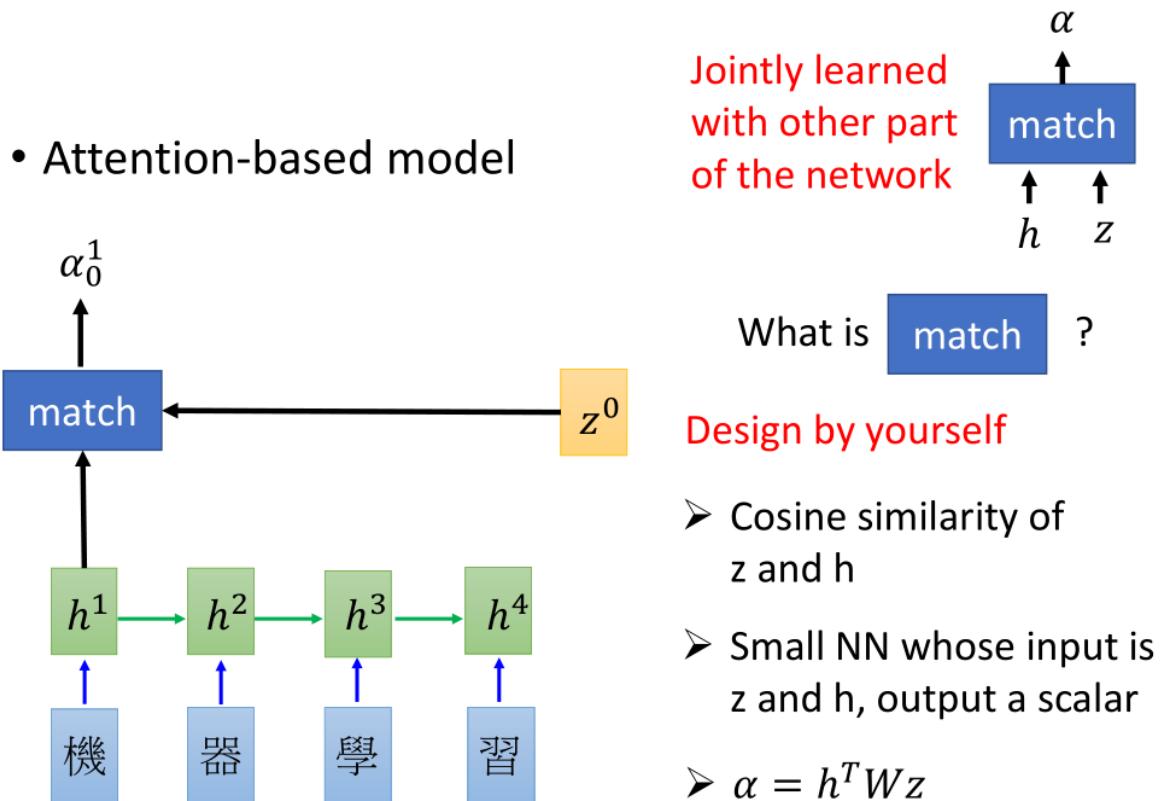
Dynamic Conditional Generation可以让机器考虑仅需要的information，让Decoder在每个时间点看到的information都是不一样的。

Attention-based model

Machine Translation

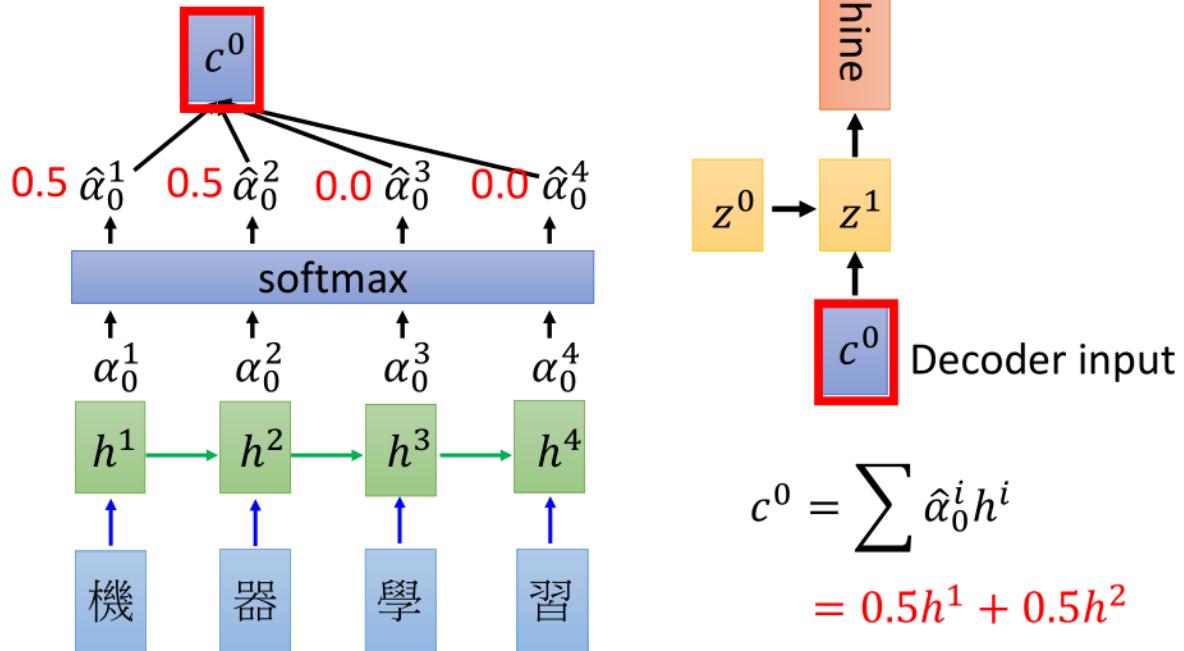
下图中的match是一个function，可以自己设计，“机”，“器”，“学”，“习”通过RNN得到各自的vector， z_0 是一个parameter。 α_0^1 ，上标1表示 z_0 和 h_1 算匹配度，下标0表示时间是0这个时间点， α 的值表示匹配程度。

match function可以自己设计，无论是那种方式，如果match方法中有参数，要和模型一起训练jointly train



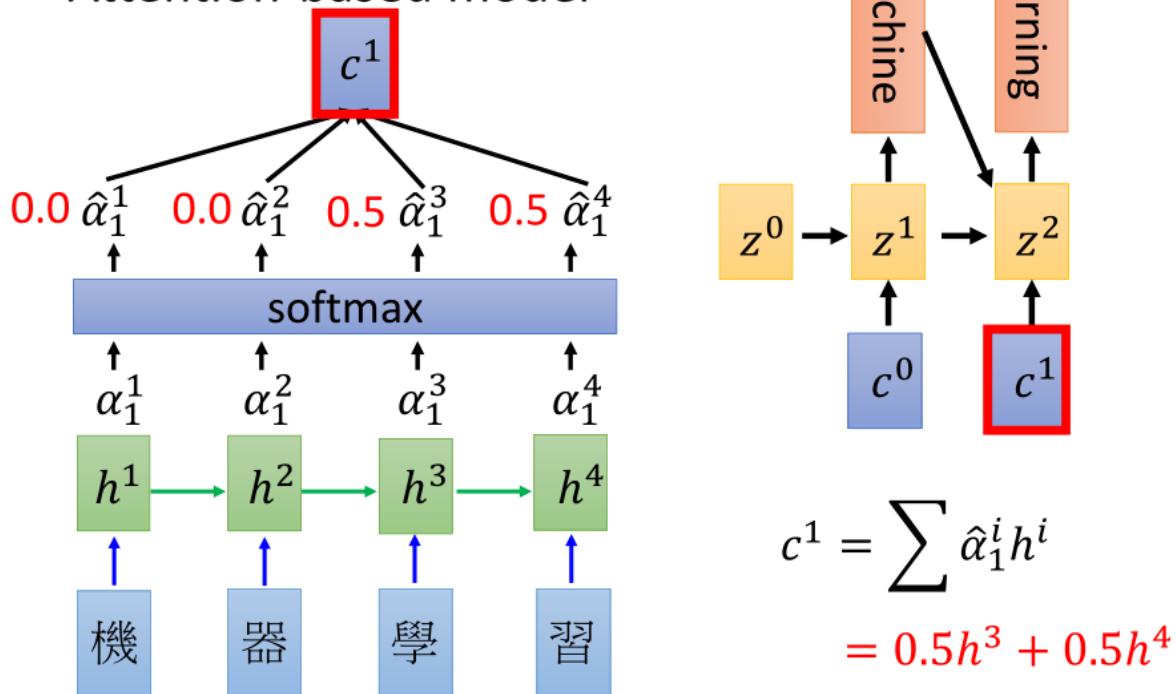
对每个输入都做match后，分别得到各自的 α ，然后softmax得到 $\hat{\alpha}$ ， c^0 为加权的句子表示

- Attention-based model



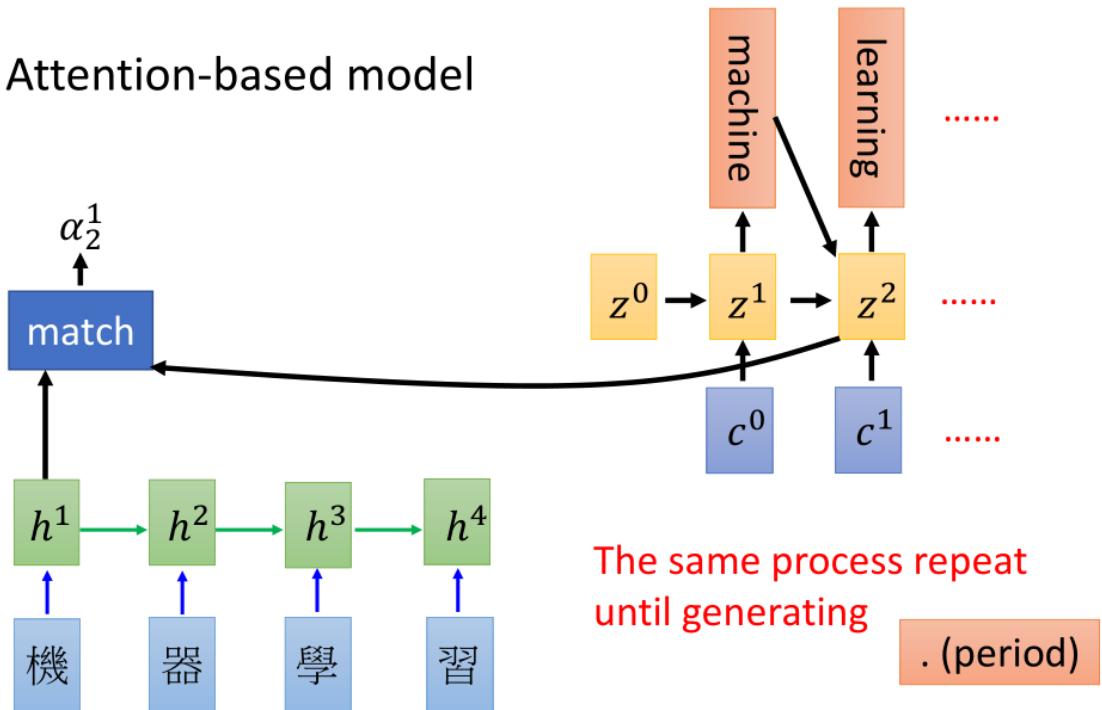
z_1 可以是 c^0 丢到 RNN 以后, hidden layer 的 output, 也可以是其他。 z_1 再去算一次 match 的分数

- Attention-based model



得到的 c^1 就是下一个 decoder 的 input, 此时只关注“学”, “习”。

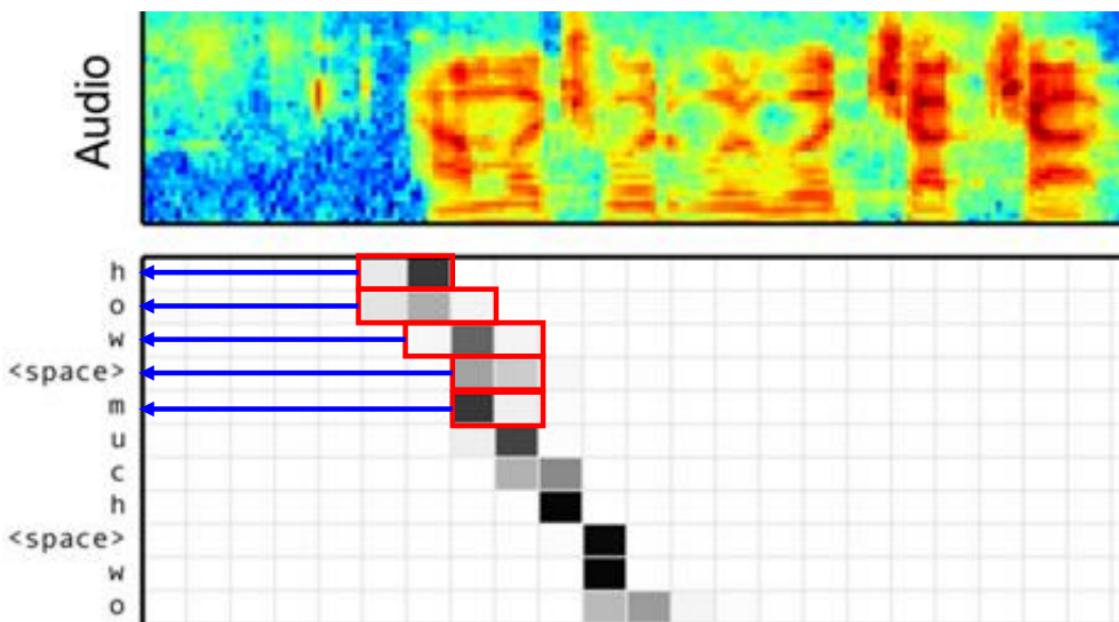
- Attention-based model



得到的 z_2 再去match得到 c^2 ，直到生成句号。

Speech Recognition

input声音讯号，output是文字，甚至可以产生空格，但是seq2seq的方法效果目前还不如传统做法，完全不需要human knowledge是它的优点。

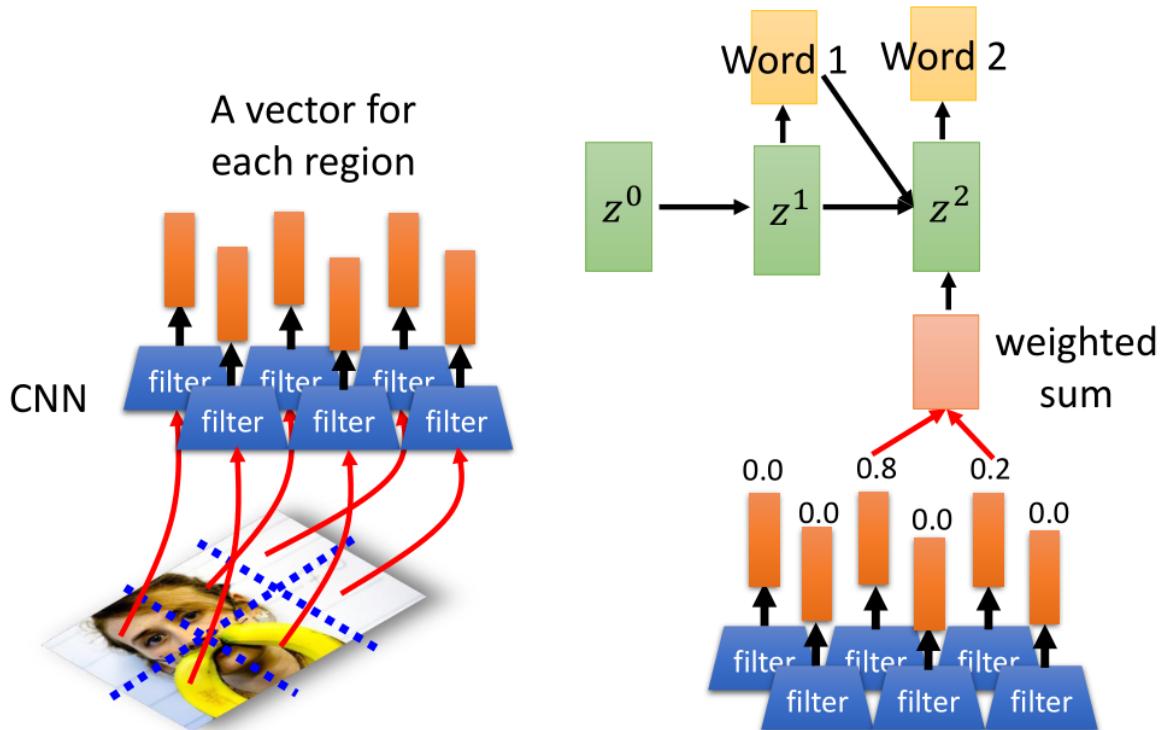


Model	Clean WER	Noisy WER
CLDNN-HMM [22]	8.0	8.9
LAS	14.1	16.5
LAS + LM Rescoring	10.3	12.0

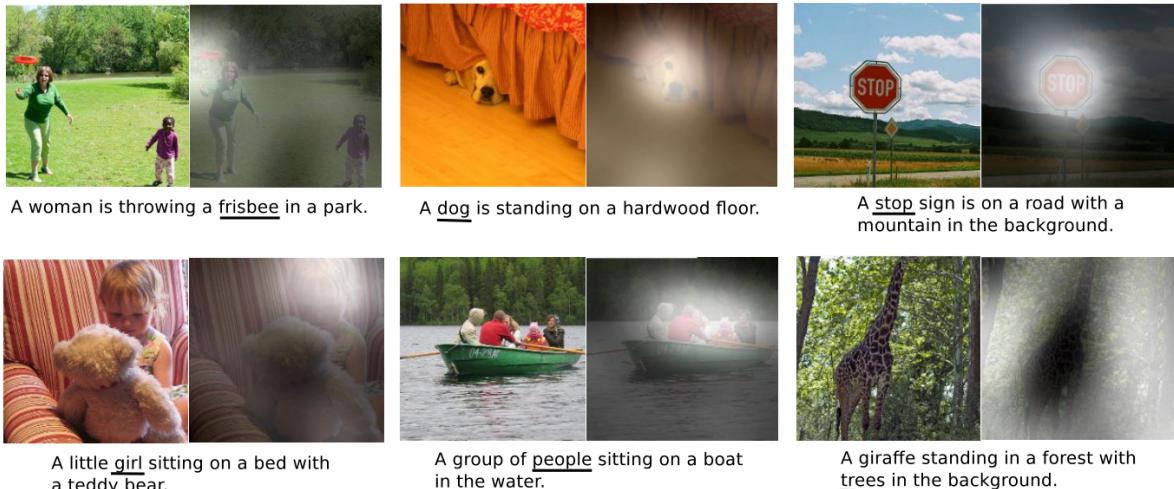
William Chan, Navdeep Jaitly, Quoc V. Le, Oriol Vinyals, "Listen, Attend and Spell", ICASSP, 2016

Image Caption Generation

用一组vector描述image, 比如用CNN filter的输出



产生划线词语时, attention的位置如图光亮处



Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, Yoshua Bengio, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML, 2015

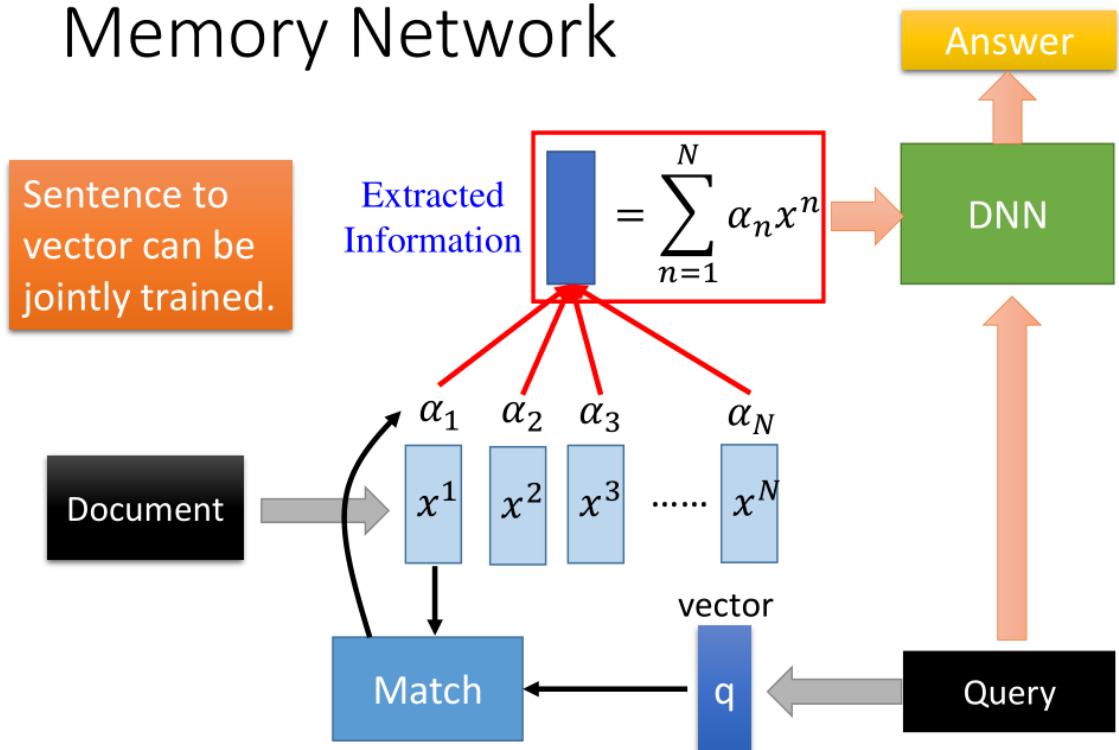
当然也有不好的例子, 但是从attention里面可以了解到它为什么会犯这些错误

从一段影像中也可以进行文字生成

Memory Network

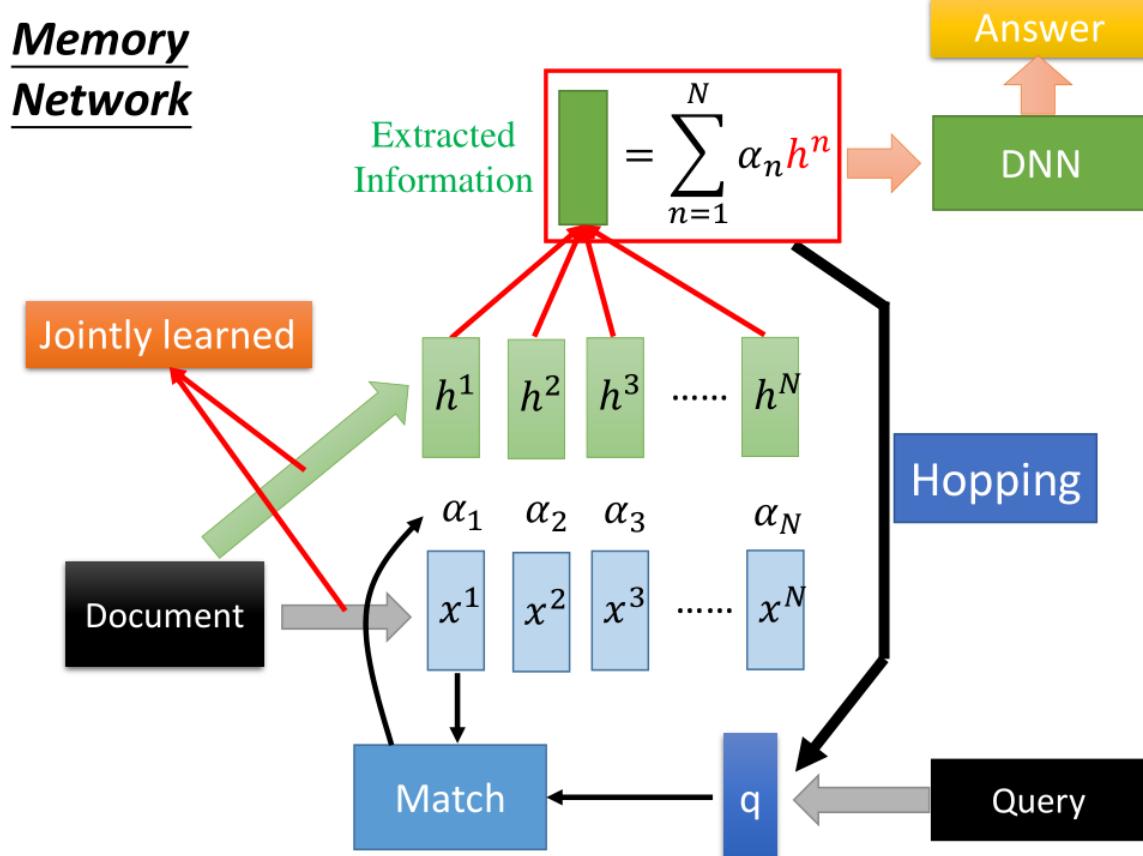
Memory Network是在memory上面做attention, Memory Network最早也是最常见的应用是在阅读理解上面, 如下图所示: 文章里面有很多句子组成, 每个句子表示成一个vector, 假设有N个句子, 向量表示成 x^1, x^2, \dots, x^N , 问题也用一个向量描述出来, 接下来算问题与句子的匹配分数 α , 做加权和, 然后丢进DNN里面, 就可以得到答案了。

Memory Network



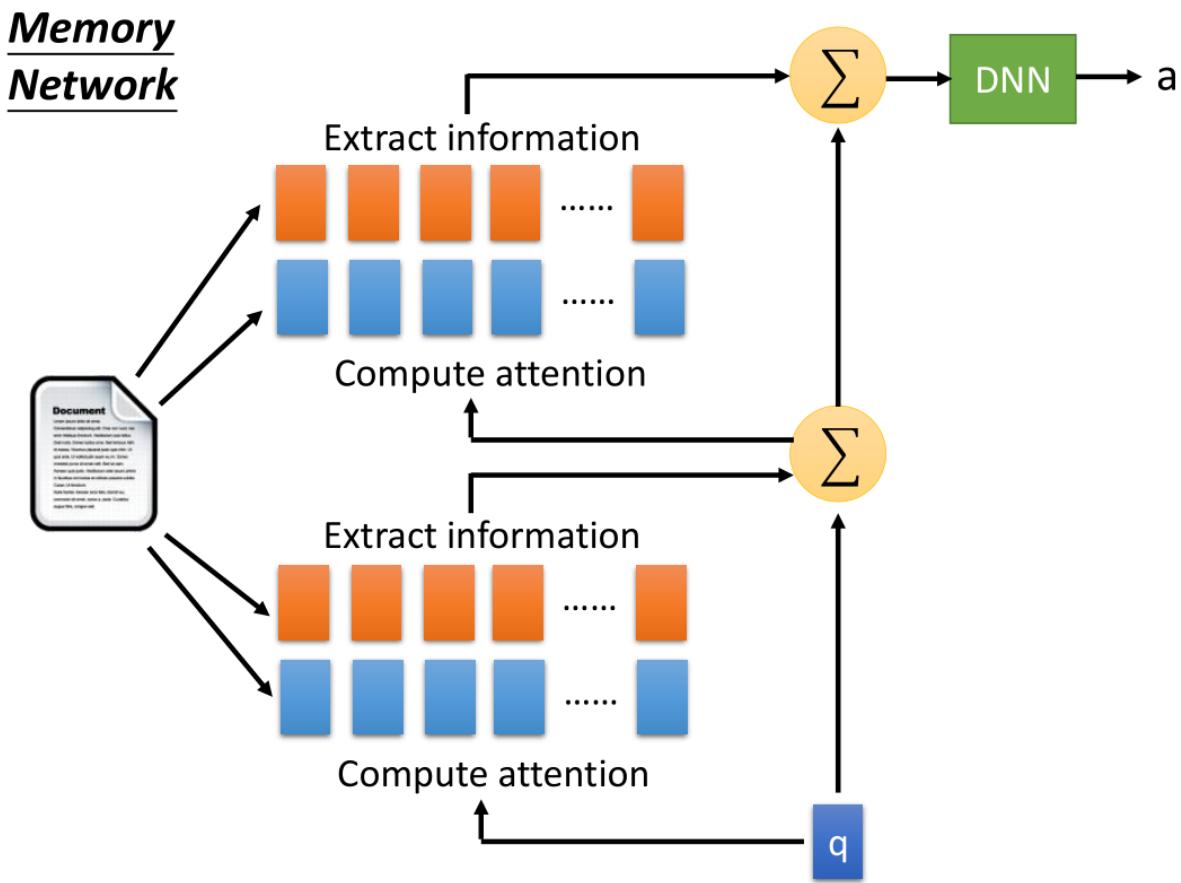
Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, Rob Fergus, "End-To-End Memory Networks", NIPS, 2015

Memory Network有一个更复杂的版本，这个版本是这样的，算match的部分跟抽取information的部分不见得是一样的，如果他们是不一样的，其实你可以得到更好的performance。把文档中的同一句子用两组向量表示；Query对x这一组vector算Attention，但是它是用h这一组向量来表示information，把这些Attention乘以h的加和得到提取的信息，放入DNN，得到答案。



通过Hopping可以反复进行运算，会把计算出的 extracted information 和 query 加在一起，重新计算 match score，然后又再算出一次 extracted information，就像反复思考一样。

如下图所示，我们用蓝色的vector计算attention，用橙色的vector做提取information。蓝色和蓝色，橙色和橙色的vector不一定是一样的。以下是两次加和后丢进去DNN得到答案。所以整件事情也可以看作两个layer的神经网络。



Neural Turing Machine

刚刚的 memory network 是在 memory 里面做 attention，并从 memory 中取出 information。而 Neural Turing Machine 还可以根据 match score 去修改 memory 中的内容。

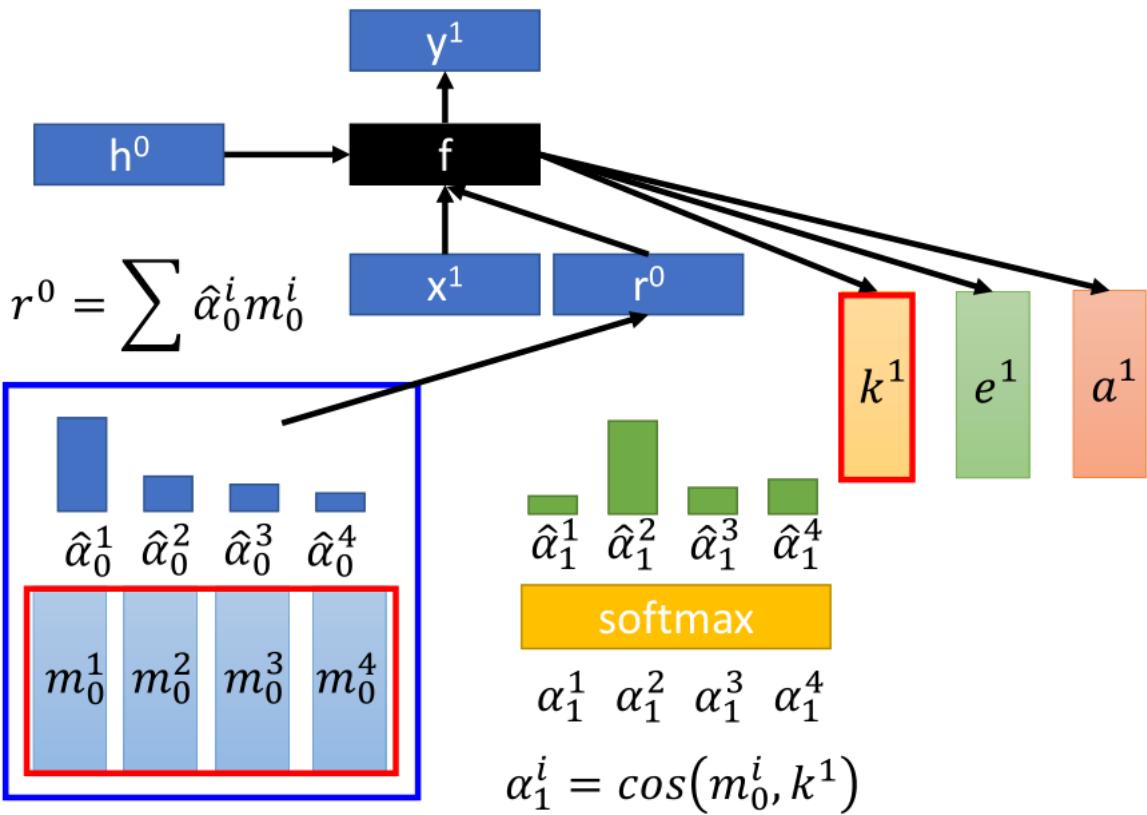
m_0^i 初始 memory sequence 的第 i 个 vector

α_0^i 第 i 个 vector 初始的 attention weight

r^0 初始的 extracted information

f 可以是 DNN\LSTM\GRU...，会 output 三个vector k, e, a

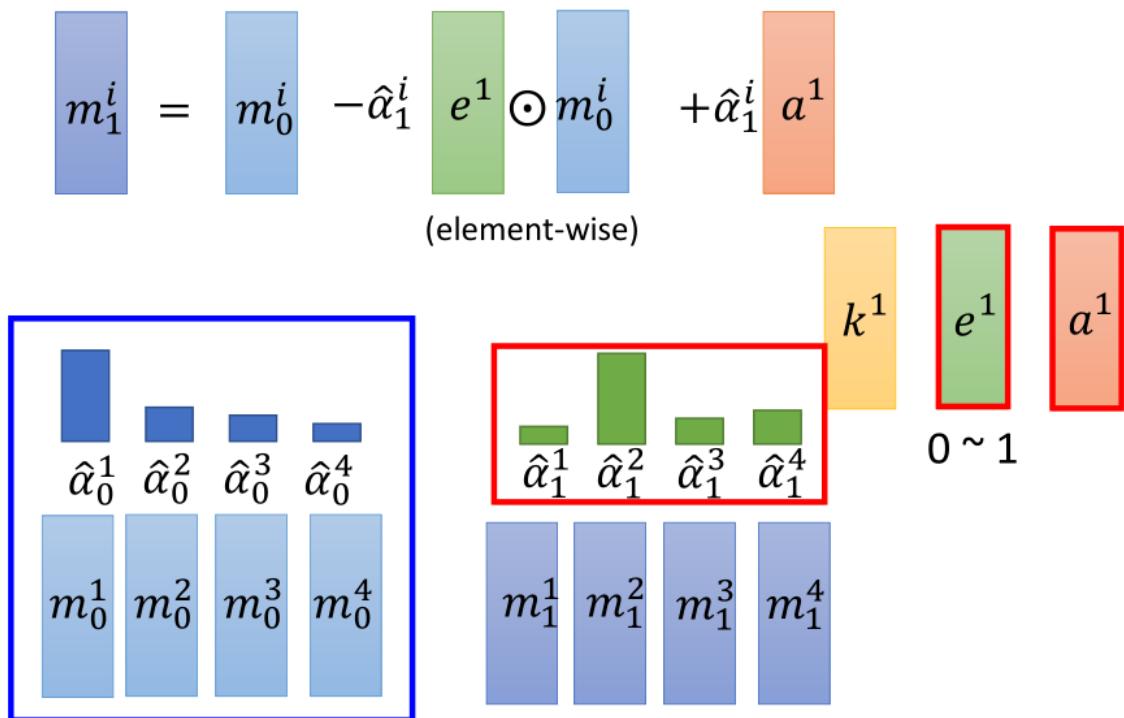
用 k, m 一起计算出 match score 得到 α_1^i ，然后 softmax，得到新的 attention $\hat{\alpha}_1^i$ ，计算 match score 的流程如下图



e, a 的作用分别是把之前的 memory 清空(erase), 以及 写入新的 memory

e 的每个 dimension 的 output 都介于 0~1 之间

m_1^i 就是新的 memory, 更新后再计算match score, 更新 attention weight, 计算 r^1 , 用于下一时刻模型输入



Tips for Generation

Attention

我们今天要做video的generation，我们给machine看一段如下的视频，如果你今天用的是Attention-based model的话，那么machine在每一个时间点会给video里面的每一帧(每一张image)一个attention，那我们用 α_t^i 来代表attention，上标*i*代表是第*i*个component，下标*t*代表是时间点。那么下标是1的四个 α 会得到 w_1 ，下标是2的四个 α 会得到 w_2 ，下标是3的四个 α 会得到 w_3 ，下标是4的四个 α 会得到 w_4 。

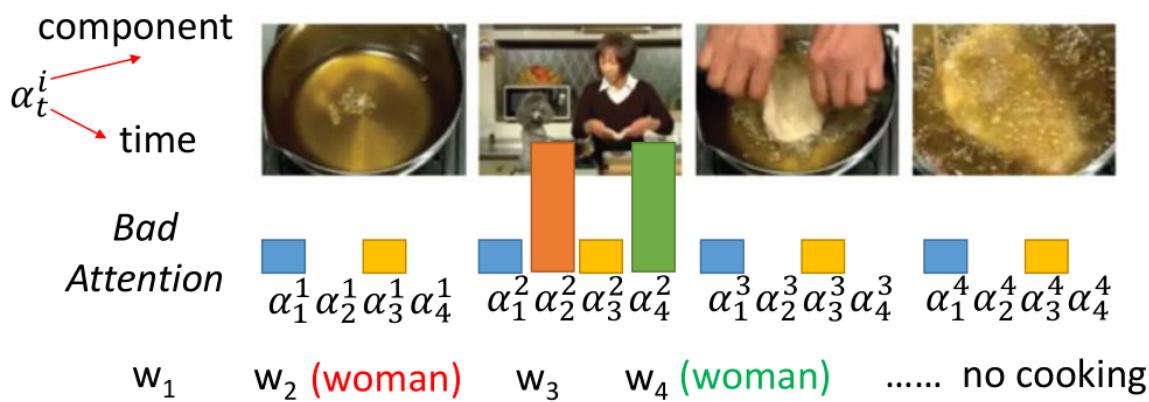
可能会有Bad Attention，如图所示，在得到第二个词的时候，attention(柱子最高的地方)主要在woman那儿，得到第四个词的时候，attention主要也在woman那儿，这样得到的不是一个好句子。

一个好的attention应该cover input所有的帧，而且每一帧的cover最好不要太多。最好的是：每一个input组件有大概相同attention权重。举一个最简单的例子，在本例中，希望在处理过程中所有attention的加和接近于一个值： τ ，这里的 τ 是类似于learning rate的一个参数。用这个正则化的目的是可以调整比较小的attention，使得整个的performance达到最好。

不要让attention过度关注于一个field，可以设置一个regularization term，使attention可以关注到其它的field。相当于加大其它component的权重。

Attention

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, Yoshua Bengio, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML, 2015



Good Attention: each input component has approximately the same attention weight

$$\text{E.g. Regularization term: } \sum_i \left(\tau - \sum_t \alpha_t^i \right)^2$$

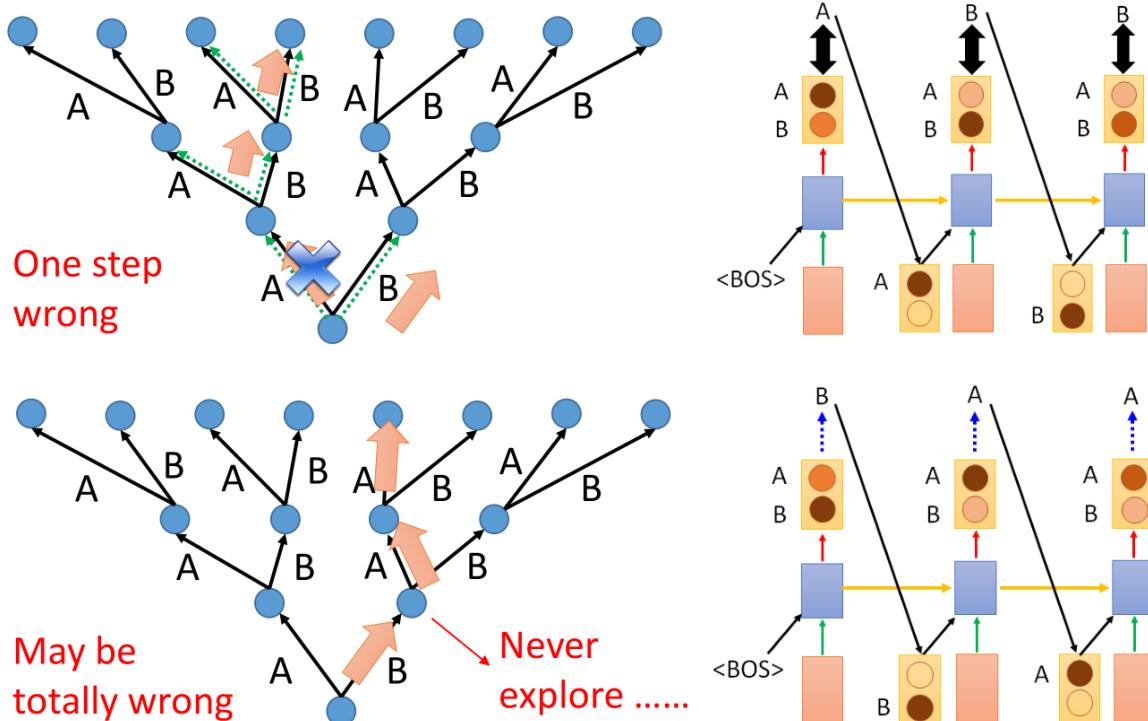
For each component Over the generation

Mismatch between Train and Test

我们做的是把condition和begin of sentence丢进去，然后output一个distribution，颜色越深表示产生的机率越大，再把产生的output作为下一个的input。

这里有一个注意的是，在training的时候，RNN 每个 step 的 input 都是正确答案 (reference)，然而 testing 时，RNN 每个 step 的 input 是它上个 step 的 output (from model)，可能输出与正确不同，这称为 Exposure Bias。

曝光误差简单来讲是因为文本生成在训练和推断时的不一致造成的。不一致体现在推断和训练时使用的输入不同，在训练时每一个词输入都来自真实样本，但是在推断时当前输入用的是上一个词的输出。



一步錯，步步錯

Modifying Training Process?

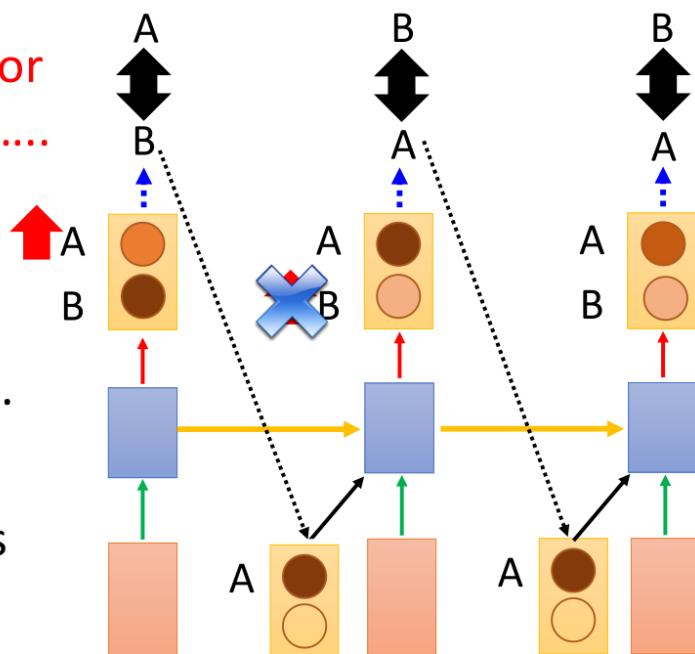
如果把 training 的 process 改成：把上个 step 的 output 当成下个 step 的 input，听起来似乎很合理，不过实际上不太容易 train 起来。比如下图：在 training 的时候，与 reference 不同，假设你的 gradient 告诉你要使 A 上升，第二个输出时使 B 上升，如果让 A 的值上升，它的 output 就会改变，即第二个时间点的 input 就会不一样，那它之前学的让 B 上升就没有意义了，可能反而会得到奇怪的结果。

When we try to decrease the loss for both step 1 and 2

Training is matched to testing.

In practice, it is hard to train in this way.

Reference



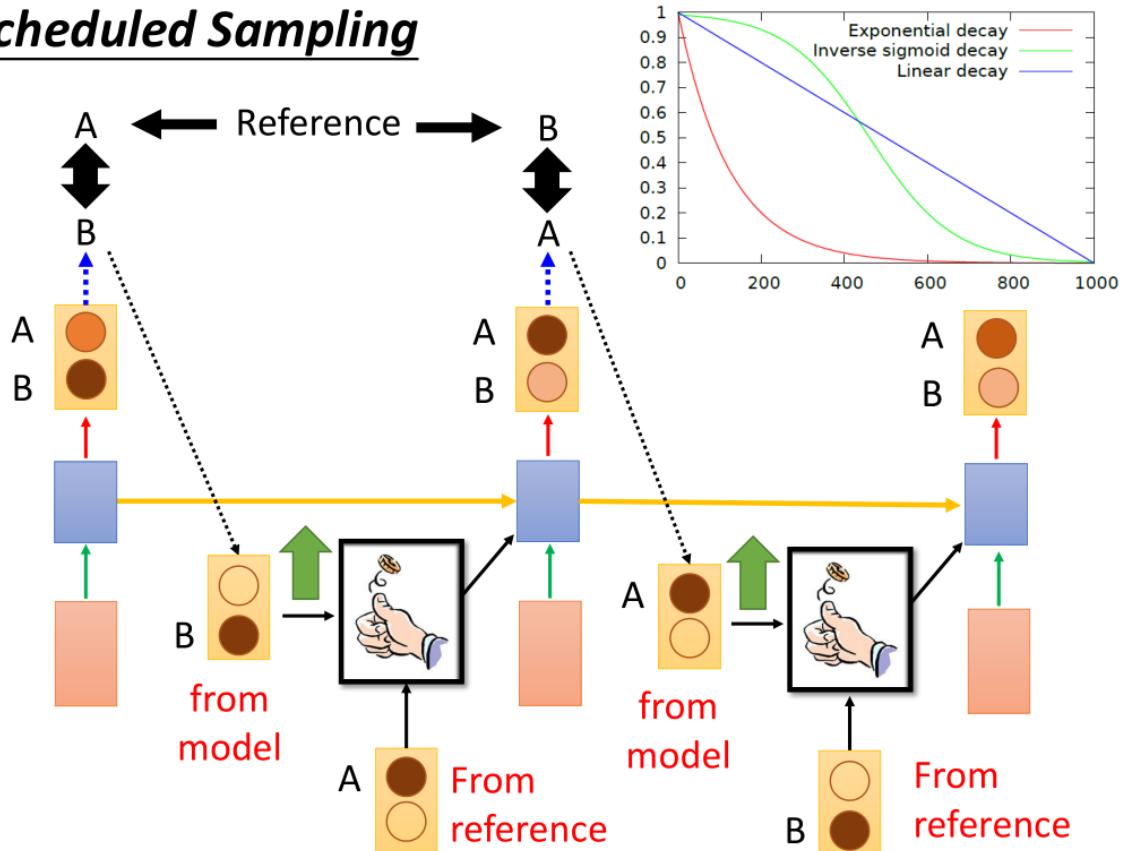
Scheduled Sampling

Scheduled Sampling通过修改我们的训练过程来解决上面的问题，一开始我们只用真实的句子序列进行训练，而随着训练过程的进行，我们开始慢慢加入模型的输出作为训练的输入这一过程。

我们纠结的点就是到底下一个时间点的input到底是从模型的output来呢，还是从reference来呢？这个Scheduled Sampling方法就说给他一个概率，概率决定以哪个作为input

一开始我们只用真实的句子序列（reference）进行训练，而随着训练过程的进行，我们开始慢慢加入模型的输出作为input这一过程。如果这样train的话，就可能得到更好的效果。

Scheduled Sampling



- Caption generation on MSCOCO

	BLEU-4	METEOR	CIDER
Always from reference	28.8	24.2	89.5
Always from model	11.2	15.7	49.7
Scheduled Sampling	30.6	24.3	92.1

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, Noam Shazeer, Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks, arXiv preprint, 2015

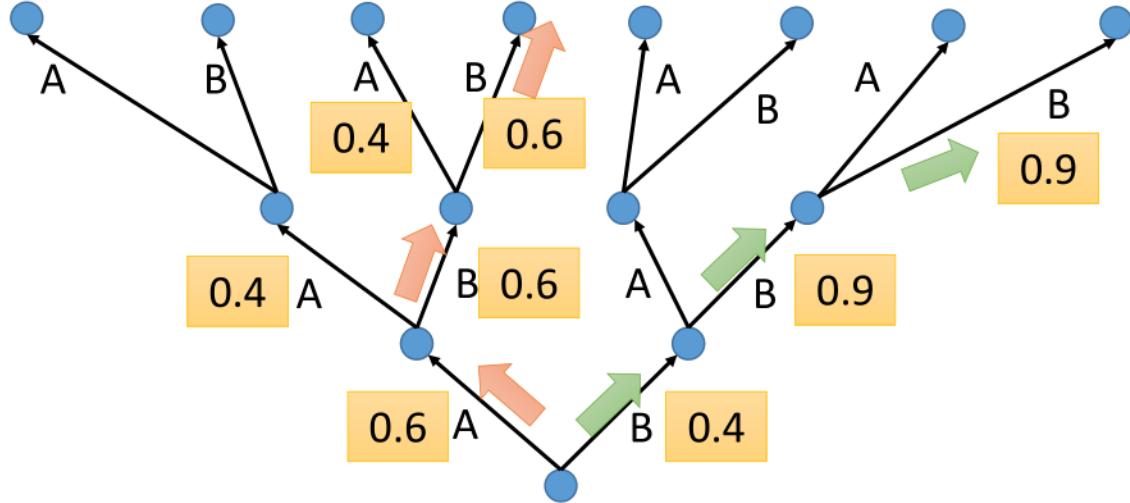
Beam Search

Beam Search是一个介于greedy search和暴力搜索之间的方法。第一个时间点都看，然后保留分数最高的k个，一直保留k个。

贪心搜索：直接选择每个输出的最大概率；暴力搜索：枚举当前所有所有出现的情况，从而得到需要的情况。

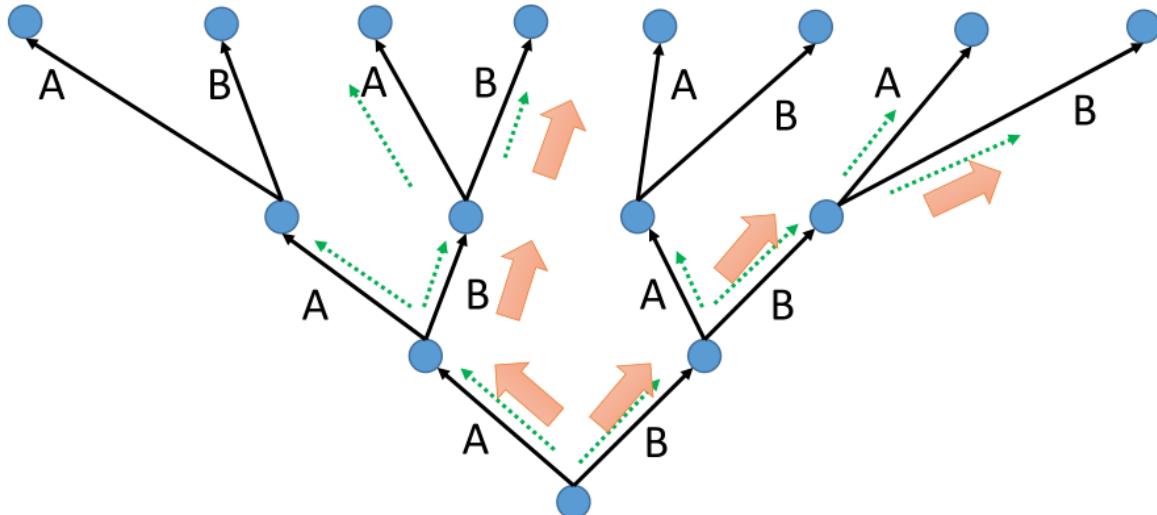
The green path has higher score.

Not possible to check all the paths



Keep several best path at each step

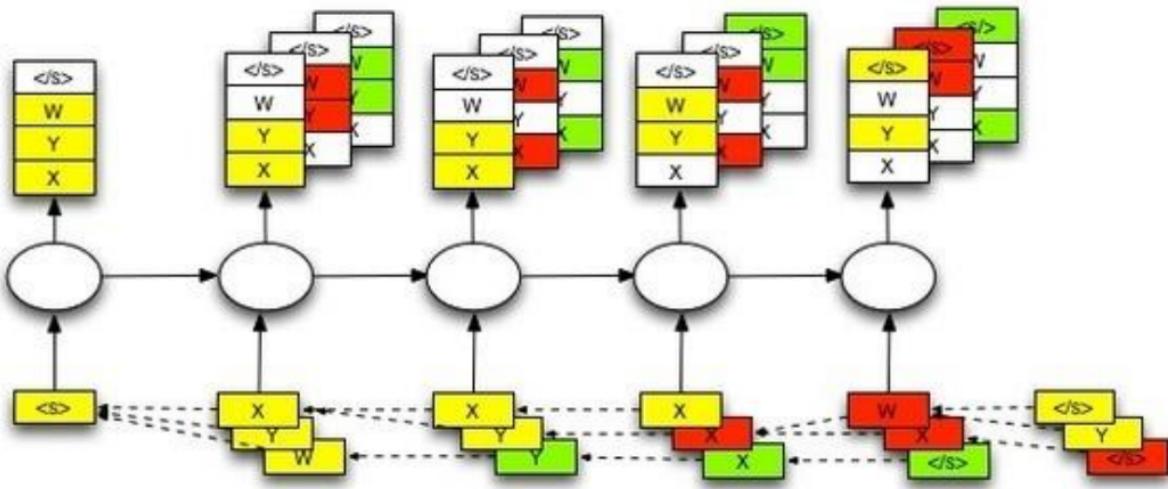
Beam Search方法是指在某个时间只pick几个分数最高的路径；选择一个beam size，即选择size个路径最佳。在搜索的时候，设置Beam size = 2，就是每一次保留分数最高的两条路径，走到最后的时候，哪一条分数最高，就输出哪一条。如下图所示：一开始，可以选择A和B两条路径，左边的第一个A点有两条路径，右边第一个B点有两条路径，此时一共有四条路径，选出分数最高的两条，再依次往下走。



下一张图是如果使用beam search的时候，应该是怎么样的；

假设世界上只有三个词XYW和一个代表句尾的符号s，我们选择size=3，每一步都选最佳的三条路径。

输出分数最高的三个X,Y,W，再分别将三个丢进去，这样就得到三组不同的distribution（一共4*3条路径），选出分数最高的三条放入.....



Better Idea?

之前 Scheduled Sampling 要解决的 problem 为何不直接把 RNN 每个 step 的 output distribution 当作下一个 step 的 input 就好了呢? 有很多好处

- training 的时候可以直接 BP 到上一个 step
- testing 的时候可以不用考虑多个 path, 不用做 beam search , 直接 output distribution

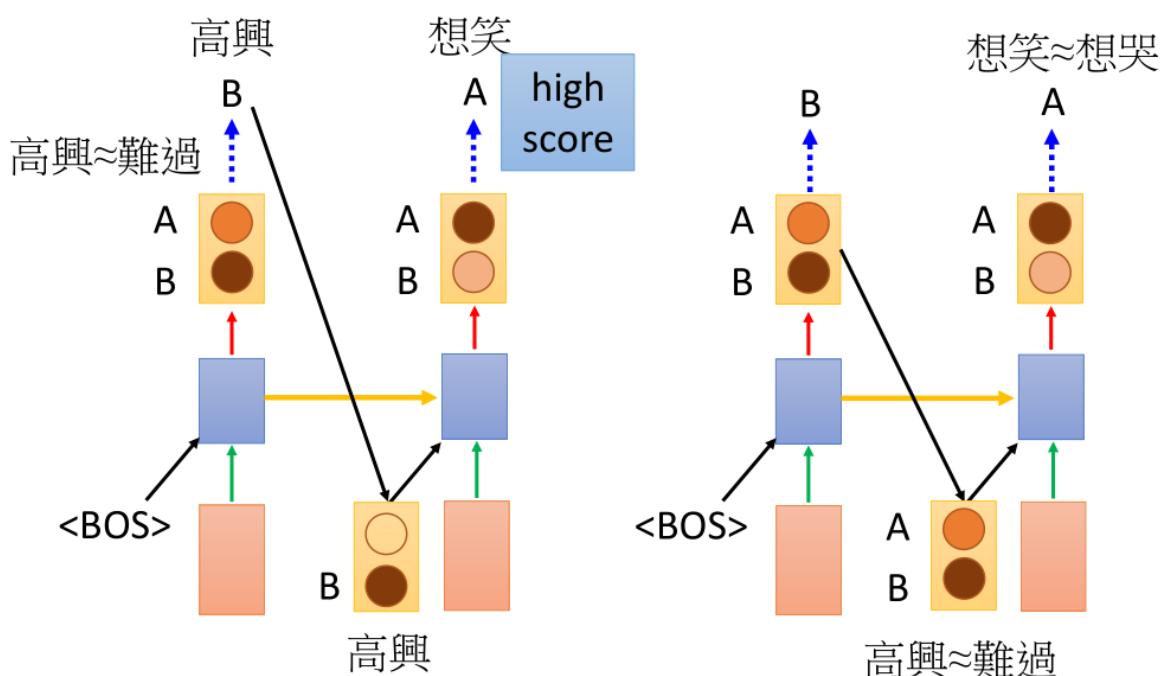
老师直觉这个做法会变糟, 原因如下:

如下图所示, 对于左边, 高兴和难过的机率几乎是一样的, 所以我们现在选择高兴丢进去, 后面接想笑的机率会很大; 而对于右边, 高兴和难过的机率几乎是一样的, 想笑和想哭的机率几乎是一样的, 那么就可能出现高兴想哭和难过想笑这样的输出, 产生句子杂糅。

Better Idea?

U: 你覺得如何?

M: 高興想笑 or 難過想哭



Object level v.s. Component level

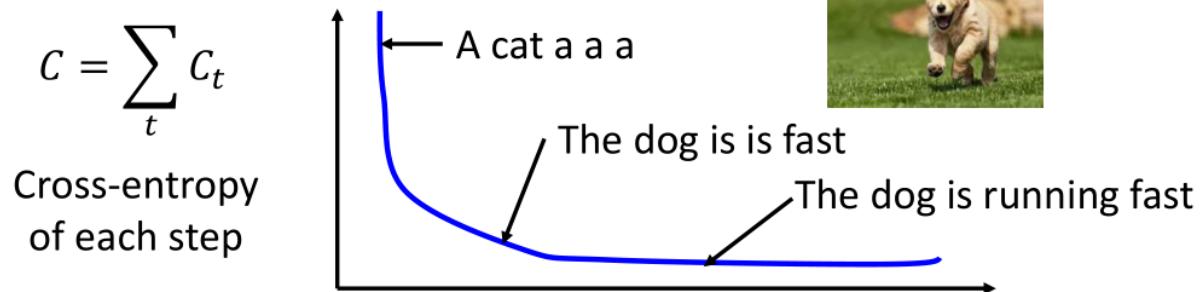
我们现在要生成的是整个句子，而不是单一的词语，所以我们在考量生成的结果好不好时候，我们应该看一个整个句子，而不是看单一的词汇。

举例来说，The dog is is fast，loss很小，但是效果并不好。用object level criterion可以根据整个句子的好坏进行评价。

但是使用这样的object level criterion是没办法做 gradient descent 的，因为 $R(y, \hat{y})$ 看的是 RNN 的 hard output，即使生成 word 的机率有改变，只要最终 output 的 y 一样，那 $R(y, \hat{y})$ 就不会变，也就是 gradient 仍然是 0，无法更新。而cross entropy在调整参数的时候，是会变的，所以可以做GD

- Minimizing the error defined on component level is not equivalent to improving the generated objects

Ref: The dog is running fast



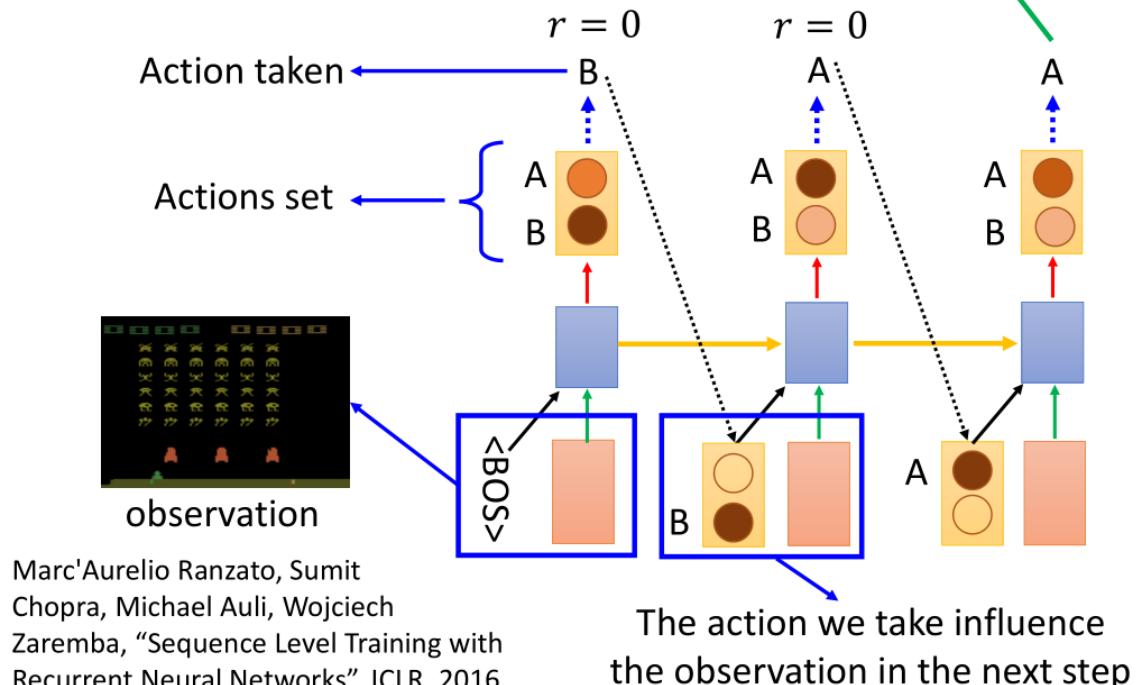
Optimize object-level criterion instead of component-level cross-entropy. object-level criterion: $R(y, \hat{y})$ Gradient Descent?
 y : generated utterance, \hat{y} : ground truth

Reinforcement learning?

RL 的 reward，基本只有最后才会拿到，可以用这招来 maximize 我们的 object level criterion

前面的r都是0，只有最后一个参数通过调整得到一个reward。

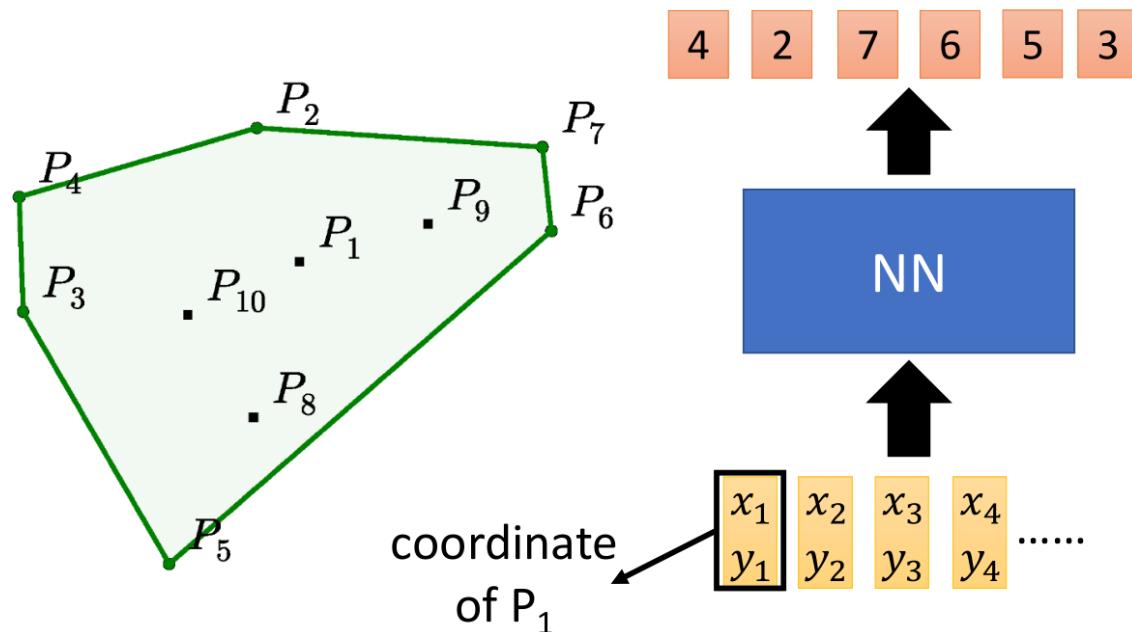
Reinforcement learning?



Pointer Network

Pointer Network最早是用来解决演算法(电脑算数学的学问)的问题。

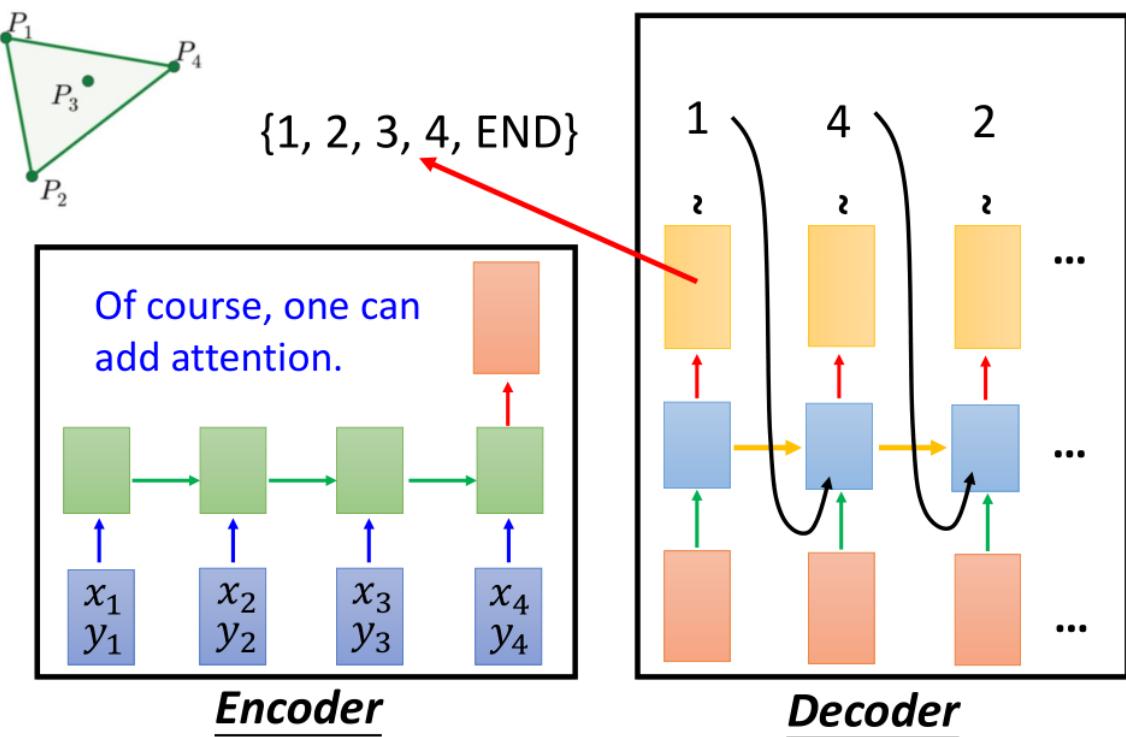
举个例子：如下图所示，给出十个点，让你连接几个点能把所有的点都包含在区域内。拿一个NN出来，它的input就是10个坐标，我们期待它的输出是427653，就可以把十个点圈起来。那就要准备一大堆的训练数据；这个 Neural Network 的输入是所有点的坐标，输出是构成凸包的点的合集。



如何求解Pointer Network?

输入一排sequence，输出另一个sequence，理论上好像是可以用Seq2Seq解决的。那么这个 Network 可以用 seq2seq 的模式么？

Sequence-to-sequence?



答案是不行的，因为，我们并不知道输出的数据的多少。更具体地说，就是在 encoder 阶段，我们只知道这个凸包问题的输入，但是在 decoder 阶段，我们不知道我们一共可以输出多少个值。

举例来说，第一次我们的输入是 50 个点，我们的输出可以是 0-50 (0 表示 END)；第二次我们的输入是 100 个点，我们的输出依然是 0-50，这样的话，我们就没办法输出 51-100 的点了。

为了解决这个问题，我们可以引入 Attention 机制，让 network 可以动态决定输出有多大。

现在用 Attention 模型把这个 sequence 读取进来，第 1-4 个点 h^1, h^2, h^3, h^4 ，在这边我们要加一个特殊的点 $h^0 = \{x_0, y_0\}$ ，代表 END 的点。

接下来，我们就采用我们之前讲过的 attention-based model，初始化一个 key，即为 z^0 ，然后用这个 key 去做 attention，用 z^0 对每一 input 做 attention，每一个 input 都产生有一个 Attention Weight。

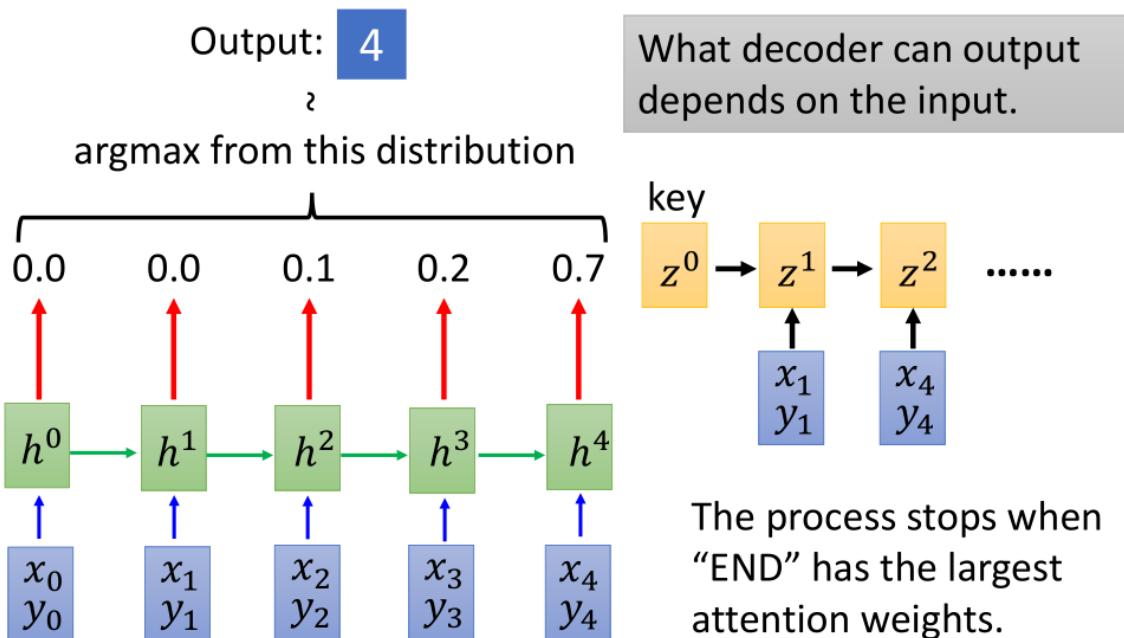
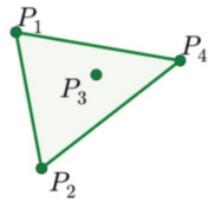
举例来说，在 (x^1, y^1) 这边 attention 的 weight 是 0.5，在 (x^2, y^2) 这边 attention 的 weight 是 0.3，在 (x^3, y^3) 这边 attention 的 weight 是 0.2，在 (x^4, y^4) 这边 attention 的 weight 是 0，在 (x^0, y^0) 这边 attention 的 weight 是 0.0。

这个 attention weight 就是我们输出的 distribution，我们根据这个 weight 的分布取 argmax，在这里，我取到 (x^1, y^1) ，output 1，然后得到下一个 key，即 z^1 ；同理，算权重，取 argmax，得到下一个 key，即 z^2 ，循环以上。

这样 output set 会跟随 input 变化

Pointer Network

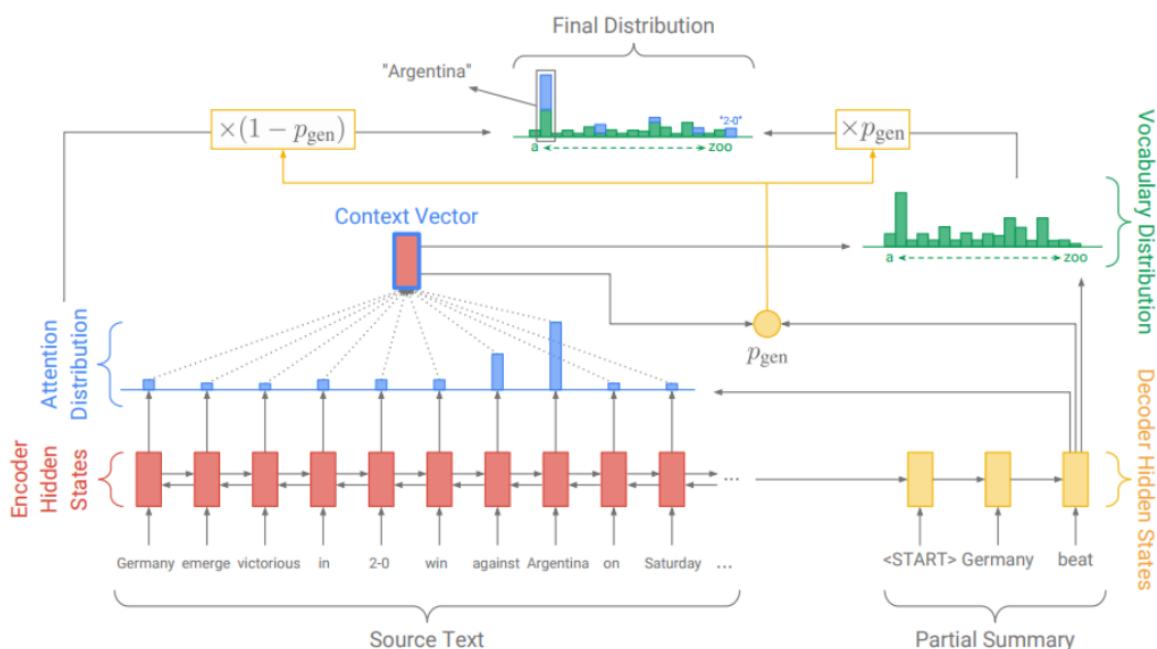
x_0
 y_0 : END



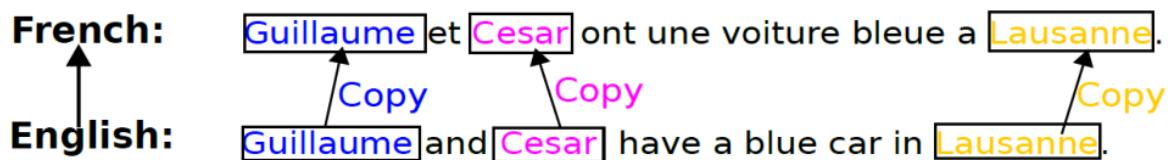
Applications

传统的seq2seq模型是无法解决输出序列的词汇表会随着输入序列长度的改变而改变的问题的，如寻找凸包等。因为对于这类问题，输出往往是输入集合的子集（输出严重依赖输入）。基于这种特点，考虑能不能找到一种结构类似编程语言中的指针，每个指针对应输入序列的一个元素，从而我们可以直接操作输入序列而不需要特意设定输出词汇表。

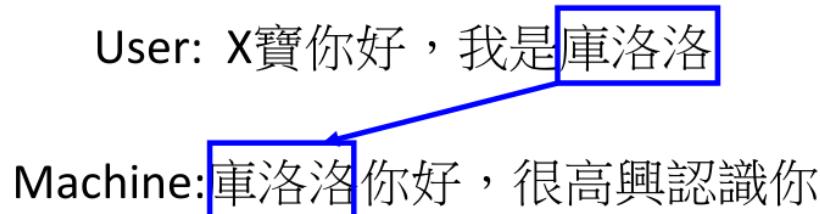
Summarization



Machine Translation



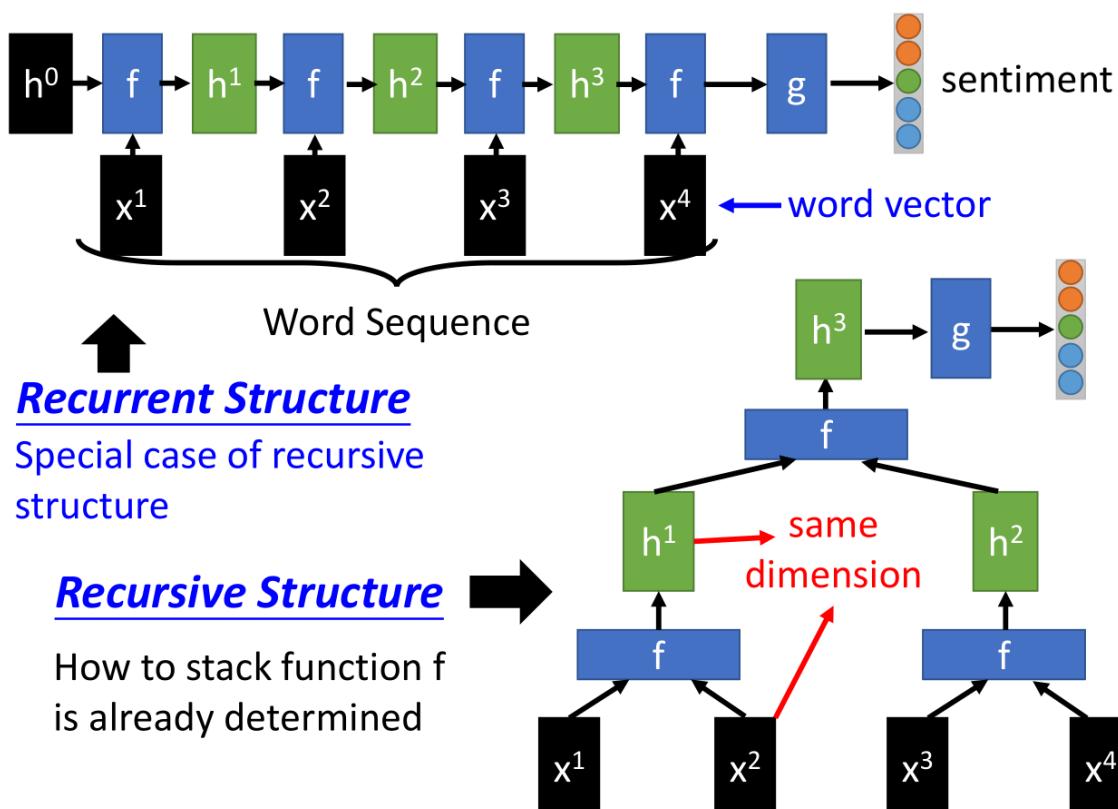
Chat-bot



Recursive Structure

RNN是Recursive Network的subset

Application: Sentiment Analysis



从RNN来看情绪分析的案例，将Word Sequence输入NN，经过相同的function-f最后经过function-g得到结果。

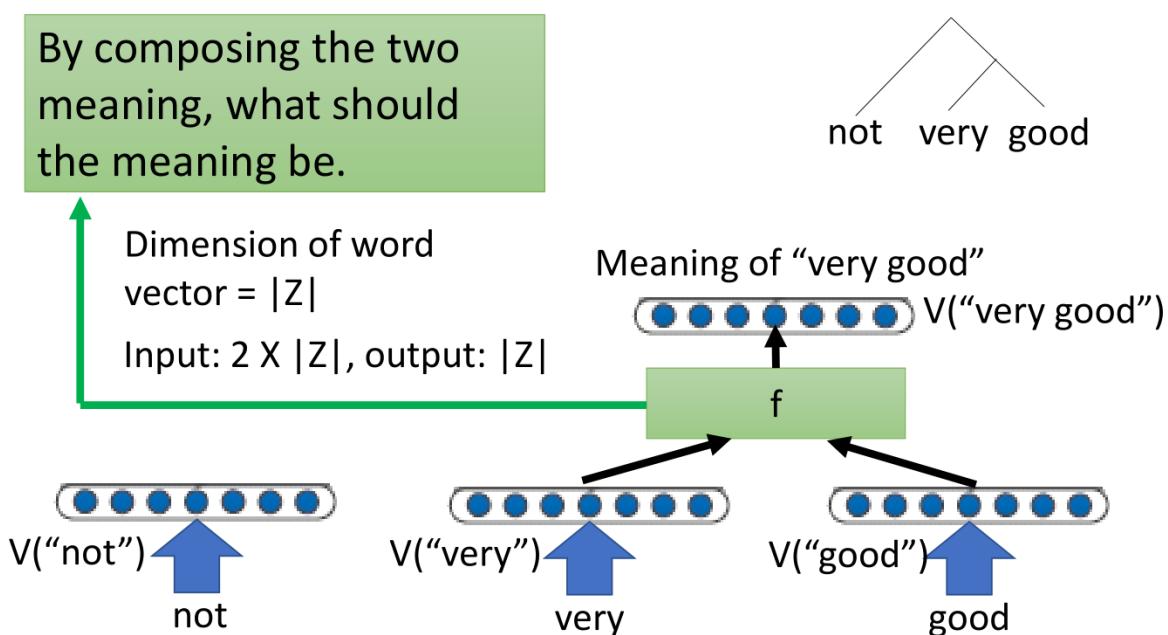
如果是Recursive Network的话，必需先决定这4个input word的structure，上图中，我们 x_1, x_2 关联得到 h_1 ， x_3, x_4 关联得到 h_2

x, h 的维度必需要相同(因为用的是同一个f)。

Recursive Model

输入和输出vector维度一致

Recursive Model

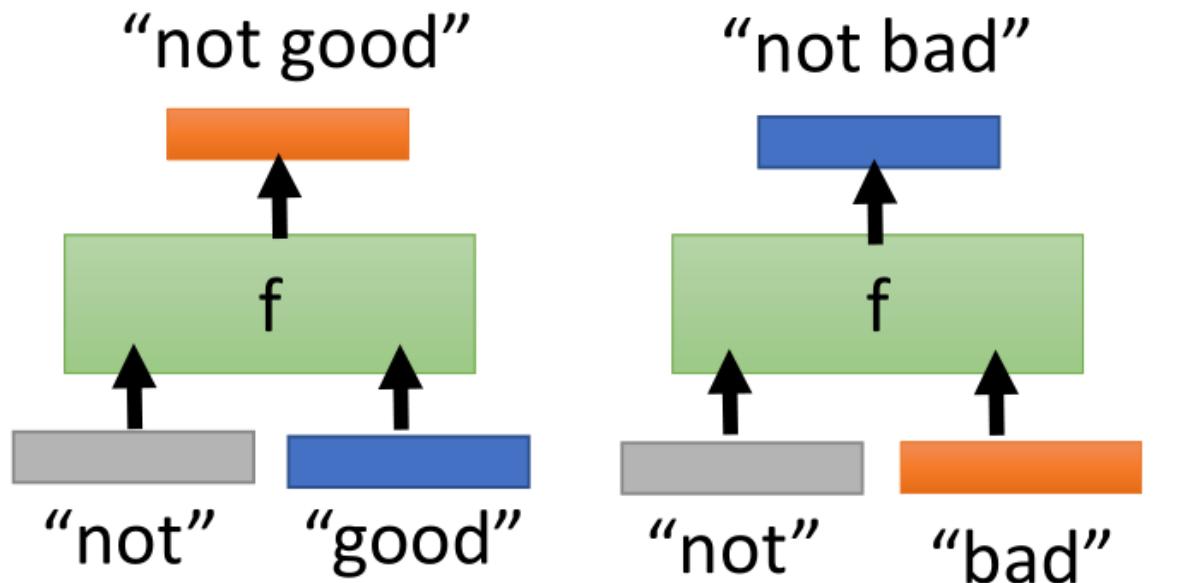


中间的 f 是一个复杂的neural network, 而不是两个单词vector的简单相加。

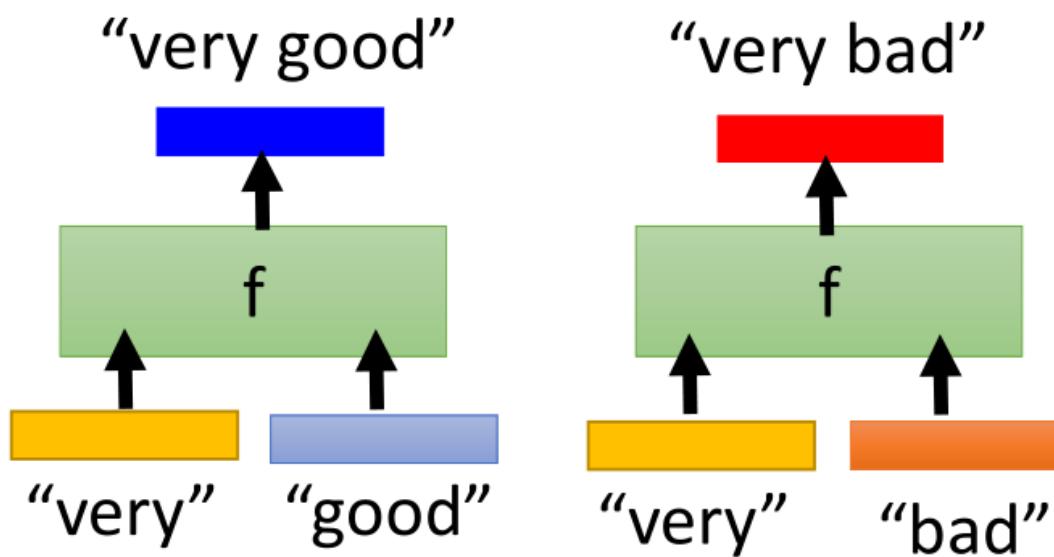
$$V(w_A w_B) \neq V(w_A) + V(w_B)$$

"good": positive, "not": neutral, "not good": negative

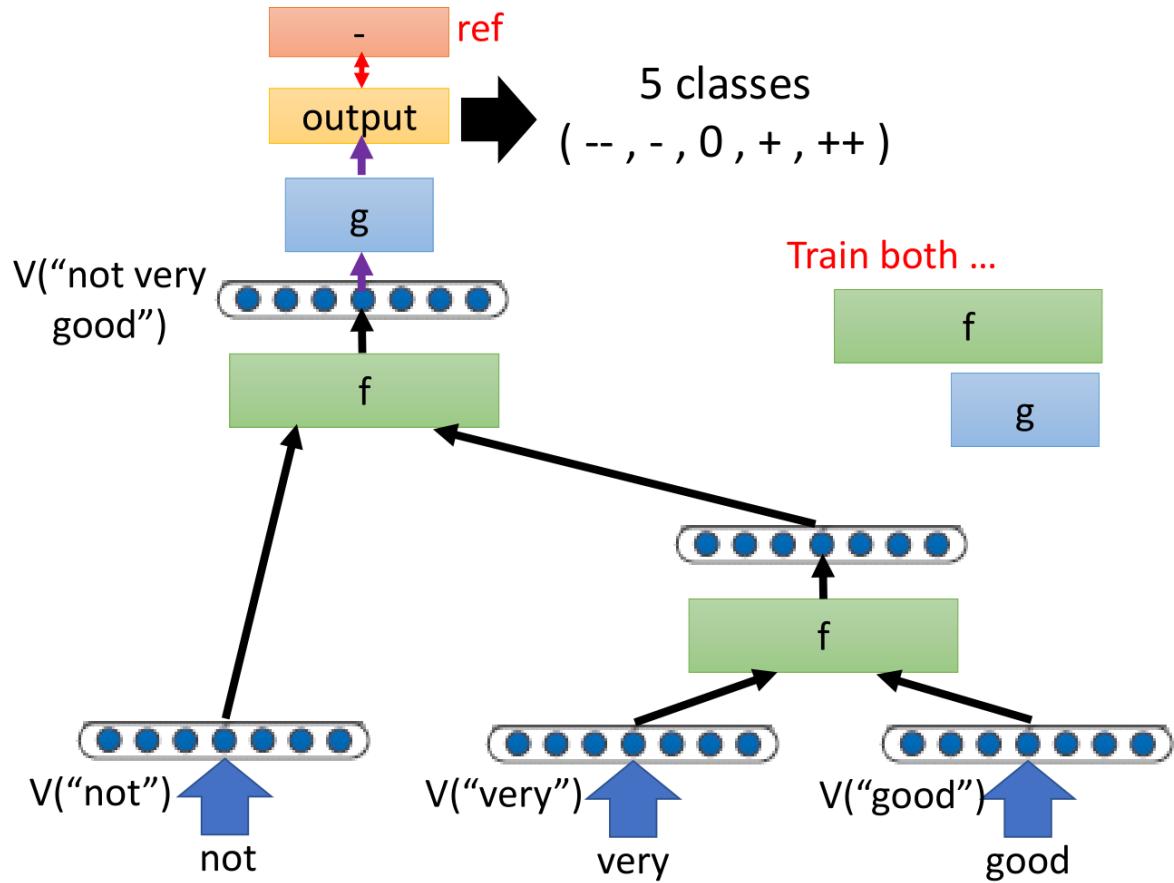
"棒": positive, "好棒": positive, "好棒棒": negative



: “reverse” another input
“not”

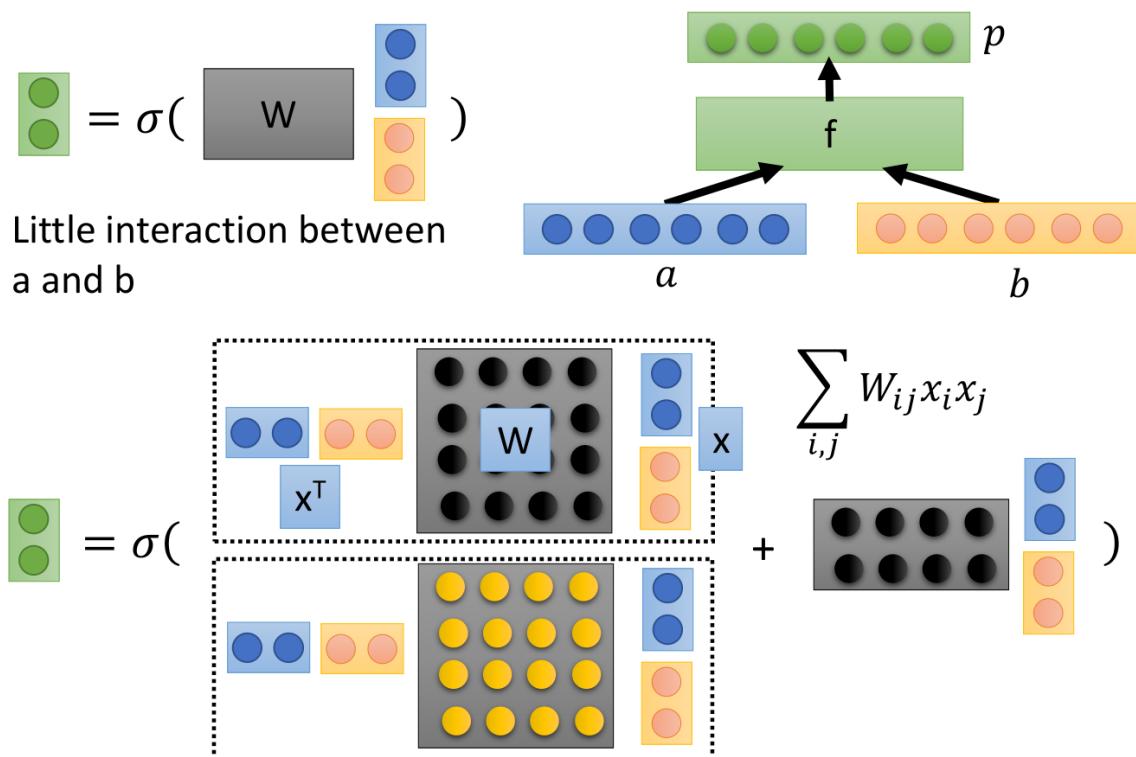


: “emphasize” another input
“very”



function- f 可以很简单，单纯的让 a,b 两个向量相加之后乘上权重 W ，但这么做可能无法满足我们上说明的需求期望，或者很难达到理想的效果。

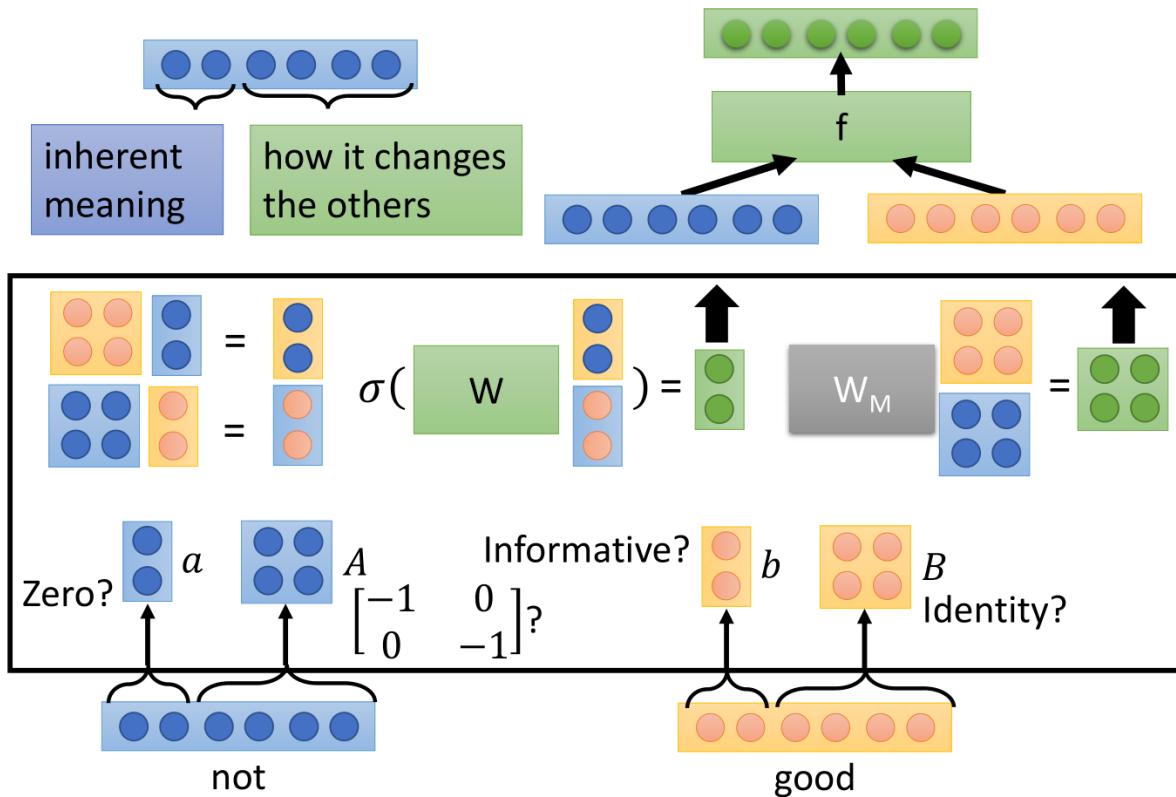
当然也可以自己设计，比如：我们要让 a,b 两个向量是有相乘的关联，因此调整为下所示，两个向量堆叠之后转置 X^T 乘上权重 W 再乘上 X ，它的计算逻辑就是将两个元素相乘 $x_i x_j$ 之后再乘上相对应的权重索引元素值 W_{ij} 做sum，这么计算之后得到的是一个数值，而后面所得项目是一个 2×1 矩阵，无法相加，因此需要重复一次，要注意两个W颜色不同代表的是不同的权重值。



Matrix-Vector Recursive Network

它的word vector有两个部分：一个是包含自身信息的vector，另一个是包含影响其他词关系的matrix。

经过如图所示计算，得到输出：两个绿色的点的matrix代表not good本身的意思，四个绿色的点是要影响别人的matrix，再把它们拉平拼接起来得到output。

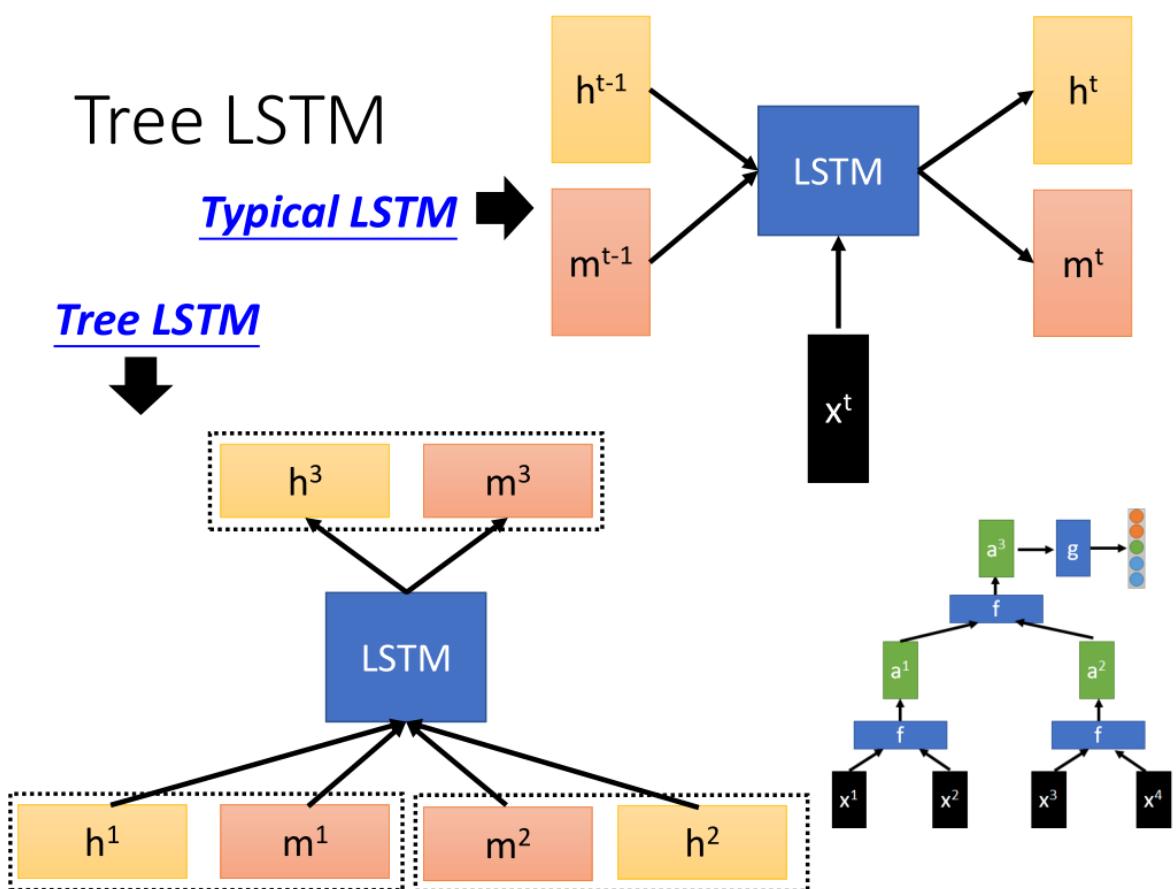


Tree LSTM

Typical LSTM: h和m的输入对应相应的输出，但是h的输入输出差别很大，m的差别不大

Tree LSTM: 就是把f换成LSTM

Tree LSTM



More Applications

如果处理的是sequence，它背后的结构你是知道的，就可以使用Recursive Network，若结构越make sense（比如数学式），相比RNN效果越好

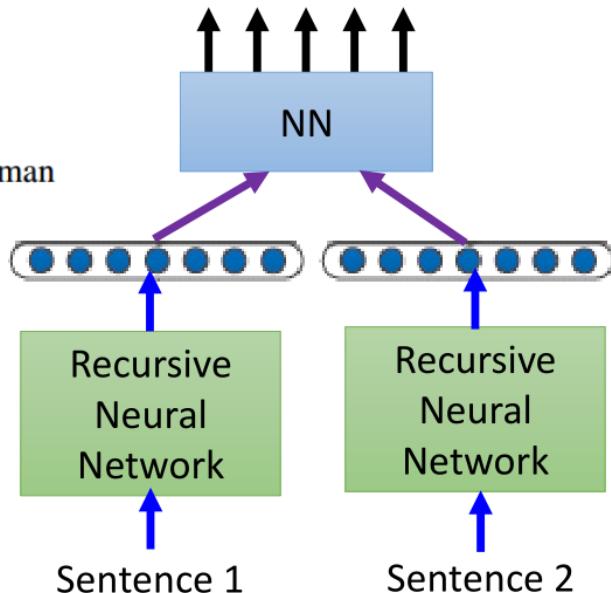
检测两个句子是不是同一个意思，把两个句子分别得到embedding，然后再丢到f，把output输入到NN里面来预测

- Sentence relatedness

a woman is slicing potatoes

- 4.82 a woman is cutting potatoes
- 4.70 potatoes are being sliced by a woman
- 4.39 tofu is being sliced by a woman

Tai, Kai Sheng, Richard Socher, and Christopher D. Manning. "Improved semantic representations from tree-structured long short-term memory networks." *arXiv preprint arXiv:1503.00075* (2015).



Transformer

Seq2seq model with "Self-attention"

处理Seq2seq问题时一般会首先想到RNN，但是RNN的问题在于无论使用单向还是双向RNN都无法并行运算，输出一个值必须等待其依赖的其他部分计算完成。

为了解决并行计算的问题，可以尝试使用CNN来处理。如下图，使用CNN时其同一个卷积层的卷积核的运算是可以并行执行的，但是浅层的卷积核只能获取部分数据作为输入，只有深层的卷积层的卷积核才有可能会覆盖到比较广的范围的数据，因此CNN的局限性在于无法使用一层来输出考虑了所有数据的输出值。

Self-Attention

self-attention可以取代原来RNN做的事情。输入是一个sequence，输出是另一个sequence。

它和双向RNN相同，每个输出也看过整个输入sequence，特别的地方是，输出是同时计算的。

You can try to replace any thing that has been done by RNN with self-attention.

步骤

首先input是 x^1 到 x^4 ，是一个sequence。

每个input先通过一个embedding $a^i = Wx^i$ 变成 a^1 到 a^4 ，然后丢进self-attention layer。

self-attention layer里面，每一个input分别乘上三个不同的transformation (matrix)，产生三个不同的向量。这个三个不同的向量分别命名为q、k、v。

- q 代表query，to match others，把每个input a^i 都乘上某个matrix W^q ，得到 q^i (q^1 到 q^4)
- k 代表key，to be matched，每个input a^i 都乘上某个matrix W^k ，得到 k^i (k^1 到 k^4)
- v 代表information，information to be extracted，每个input a^i 都乘上某个matrix W^v ，得到 v^i (v^1 到 v^4)

现在每个时刻，每个 a^i 都有q、k、v三个不同的向量。

接下来要做的事情是拿每一个q对每个k做attention。attention是吃两个向量，output一个分数，告诉你这两个向量有多匹配，至于怎么吃这两个向量，则有各种各样的方法，这里我们采用Scaled Dot-Product Attention， q^1 和 k^i 做点积，然后除以 \sqrt{d} 。

- 把 q^1 拿出来，对 k^i 做attention得到一个分数 $\alpha_{1,i}$

d 是q跟k的维度，因为q要跟k做点积，所以维度是一样的。因为dim越大，点积结果越大，因此除以一个 \sqrt{d} 来平衡。

- 接下来 $\alpha_{1,i}$ 通过一个softmax layer 得到 $\hat{\alpha}_{1,i}$
- 然后拿 $\hat{\alpha}_{1,i}$ 分别和每一个 v^i 相乘。即 $\hat{\alpha}_{1,1}$ 乘上 v^1 ， $\hat{\alpha}_{1,2}$ 乘上 v^2 ， ..., 最后相加起来。等于 v^1 到 v^4 拿 $\hat{\alpha}_{1,i}$ 做加权求和，得到 b^1 。

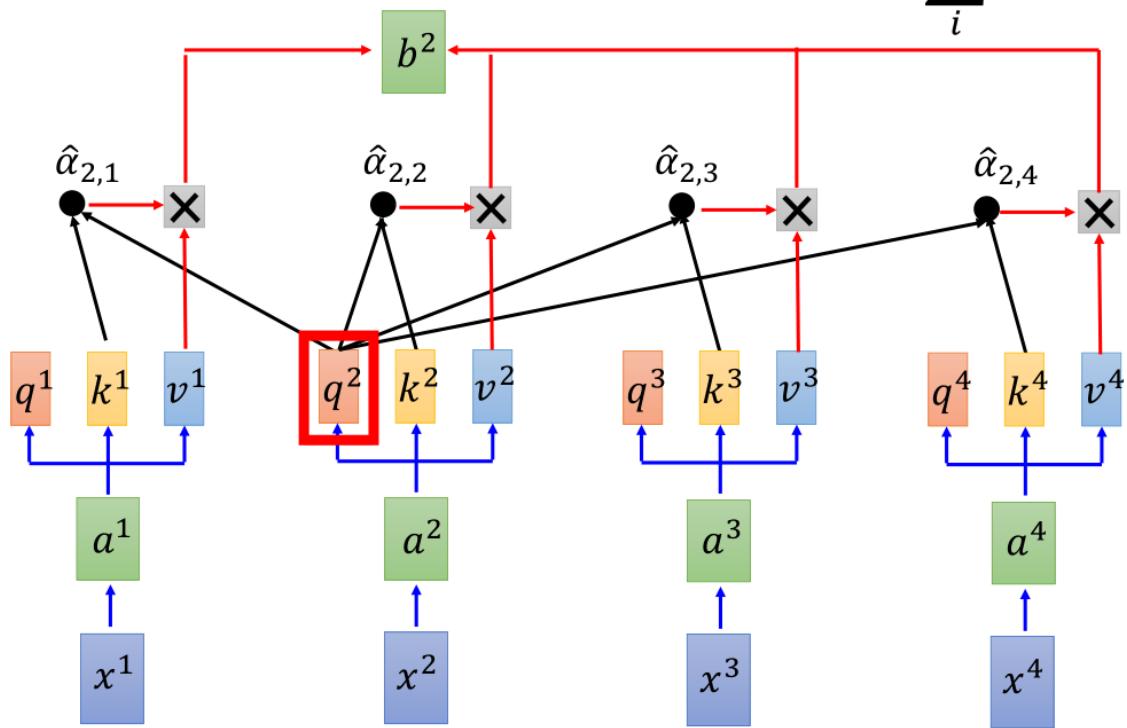
现在就得到了sequence的第一个向量 b^1 。在产生 b^1 的时候，用了整个sequence的信息，看到了 a^1 到 a^4 的信息。如果你不想考虑整个句子的信息，只想考虑局部信息，只要让 $\hat{\alpha}_{1,i}$ 的值为0，意味着不会考虑 a^i 的信息。如果想考虑某个 a^i 的信息，只要让对应的 $\hat{\alpha}_{1,i}$ 有值即可。所以对self-attention来说，只要它想看，就能用attention看到，只要自己学习就行。

在同一时间，就可以用同样的方式，把 b^2, b^3, b^4 也算出来。

Self-attention

拿每個 query q 去對每個 key k 做 attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$



矩阵运算

Self-attention

$$\begin{matrix} q^1 & q^2 & q^3 & q^4 \end{matrix} = \begin{matrix} W^q \\ Q \end{matrix} \quad \begin{matrix} a^1 & a^2 & a^3 & a^4 \end{matrix} = \begin{matrix} I \\ a^1 & a^2 & a^3 & a^4 \end{matrix}$$

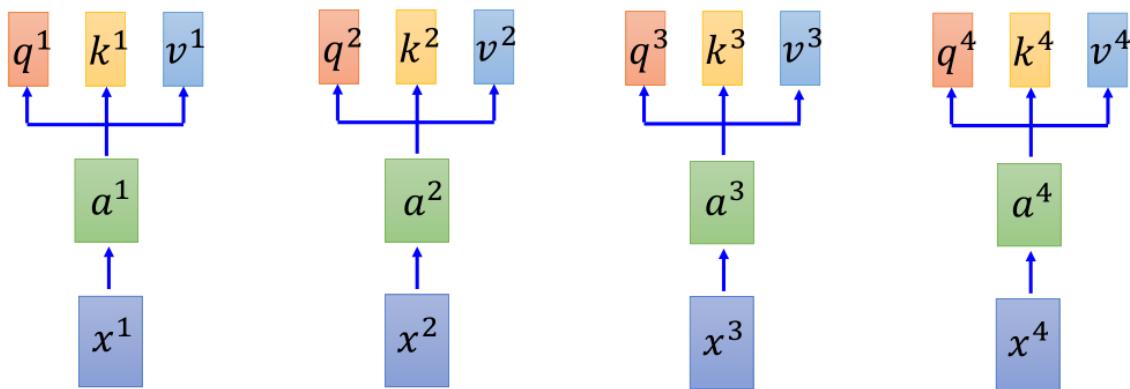
$$q^i = W^q a^i$$

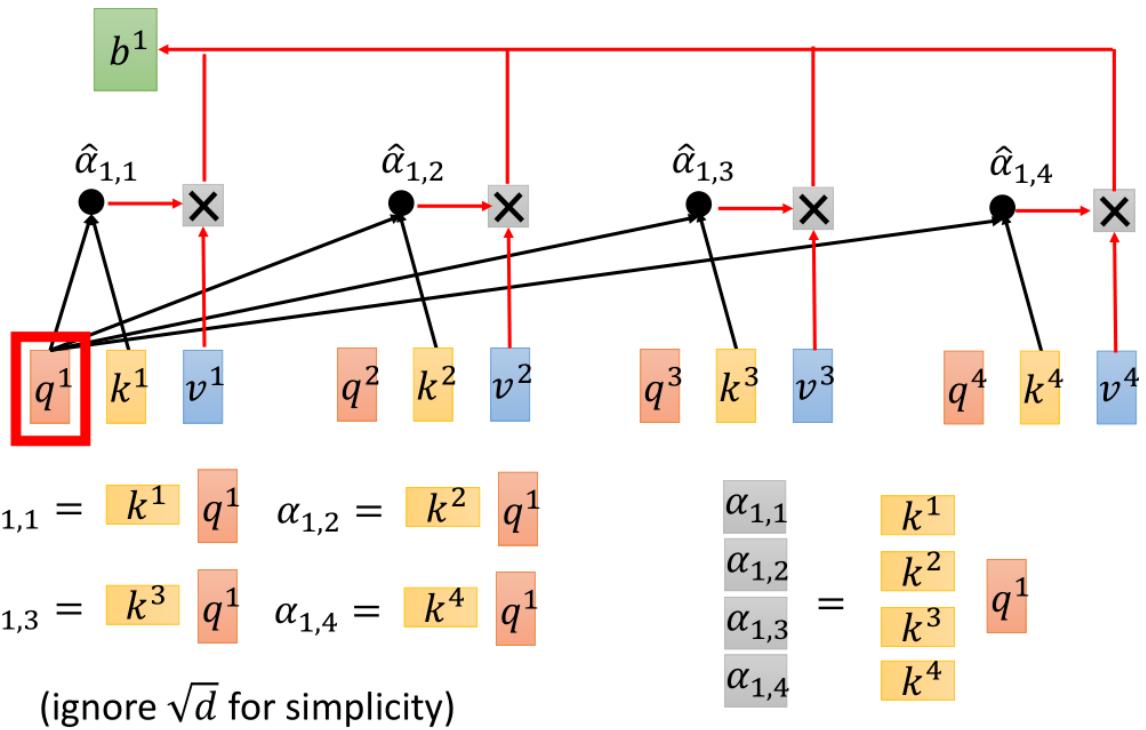
$$\begin{matrix} k^1 & k^2 & k^3 & k^4 \end{matrix} = \begin{matrix} W^k \\ K \end{matrix} \quad \begin{matrix} a^1 & a^2 & a^3 & a^4 \end{matrix} = \begin{matrix} I \\ a^1 & a^2 & a^3 & a^4 \end{matrix}$$

$$k^i = W^k a^i$$

$$v^i = W^v a^i$$

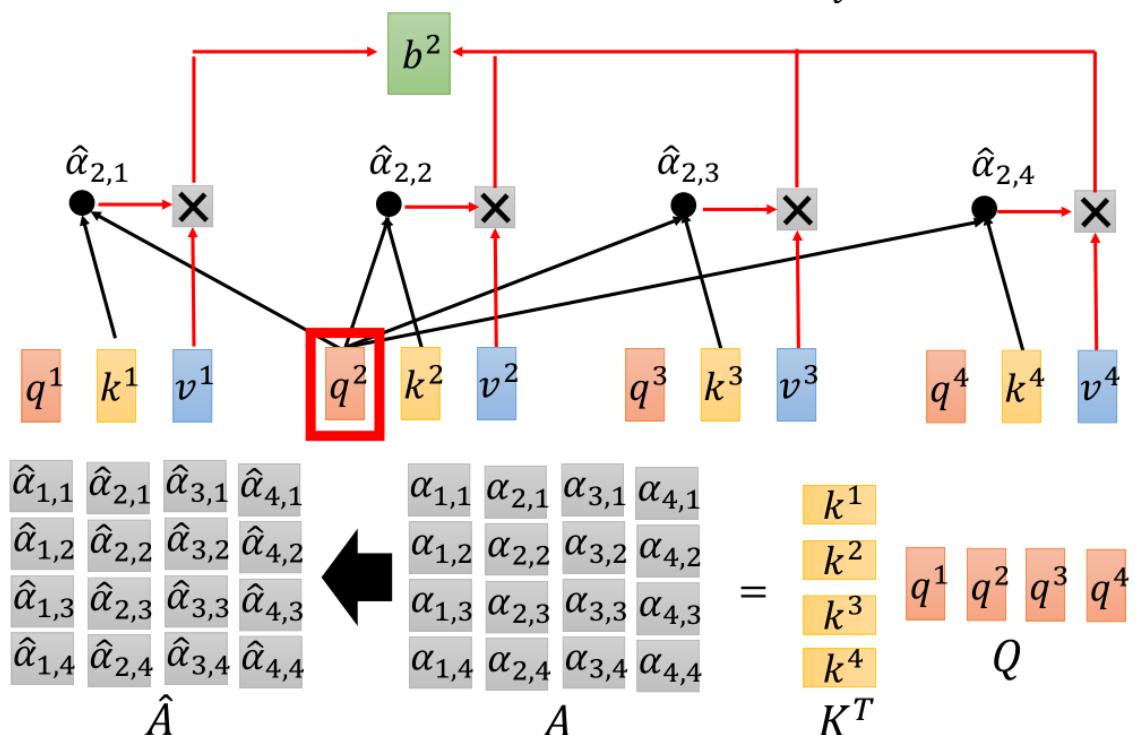
$$\begin{matrix} v^1 & v^2 & v^3 & v^4 \end{matrix} = \begin{matrix} W^v \\ V \end{matrix} \quad \begin{matrix} a^1 & a^2 & a^3 & a^4 \end{matrix} = \begin{matrix} I \\ a^1 & a^2 & a^3 & a^4 \end{matrix}$$





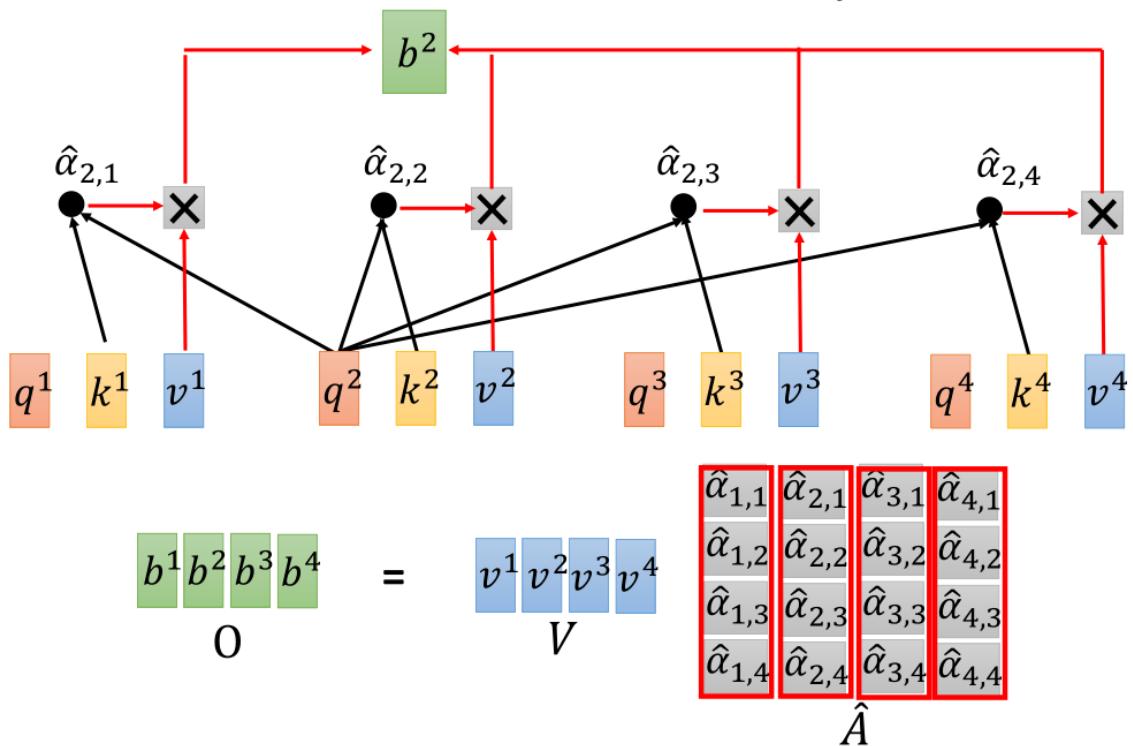
Self-attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$



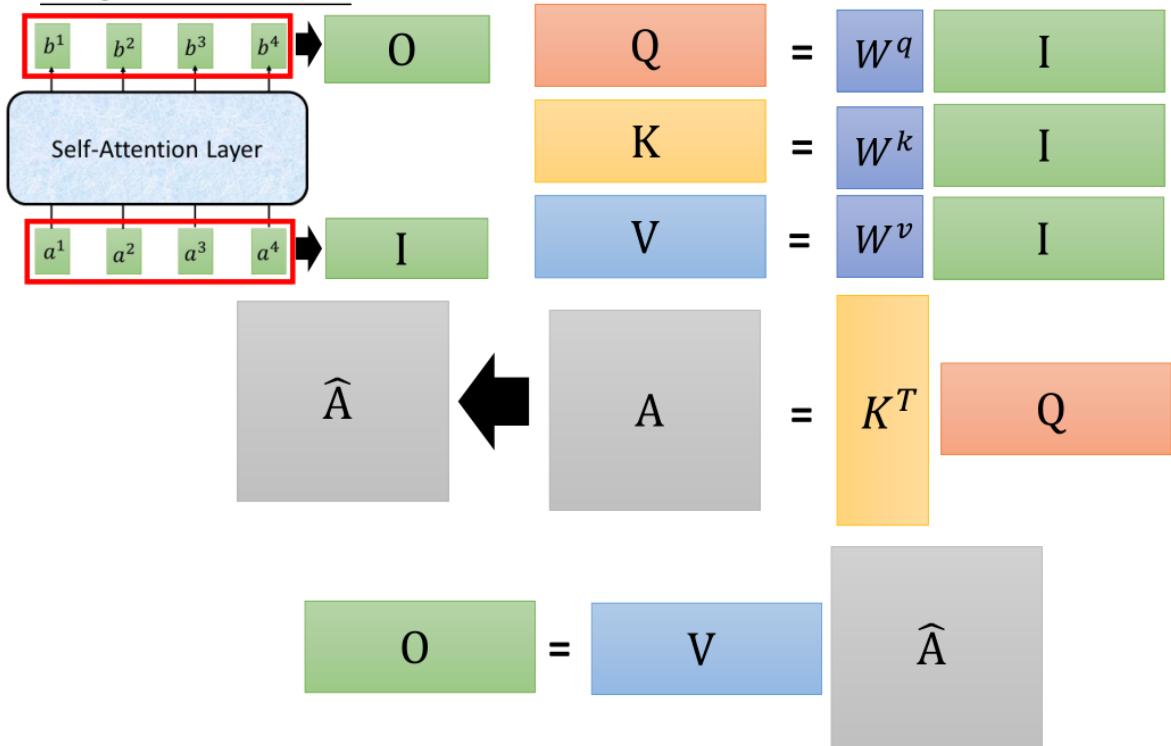
Self-attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$



反正就是一堆矩阵乘法，用 GPU 可以加速

Self-attention



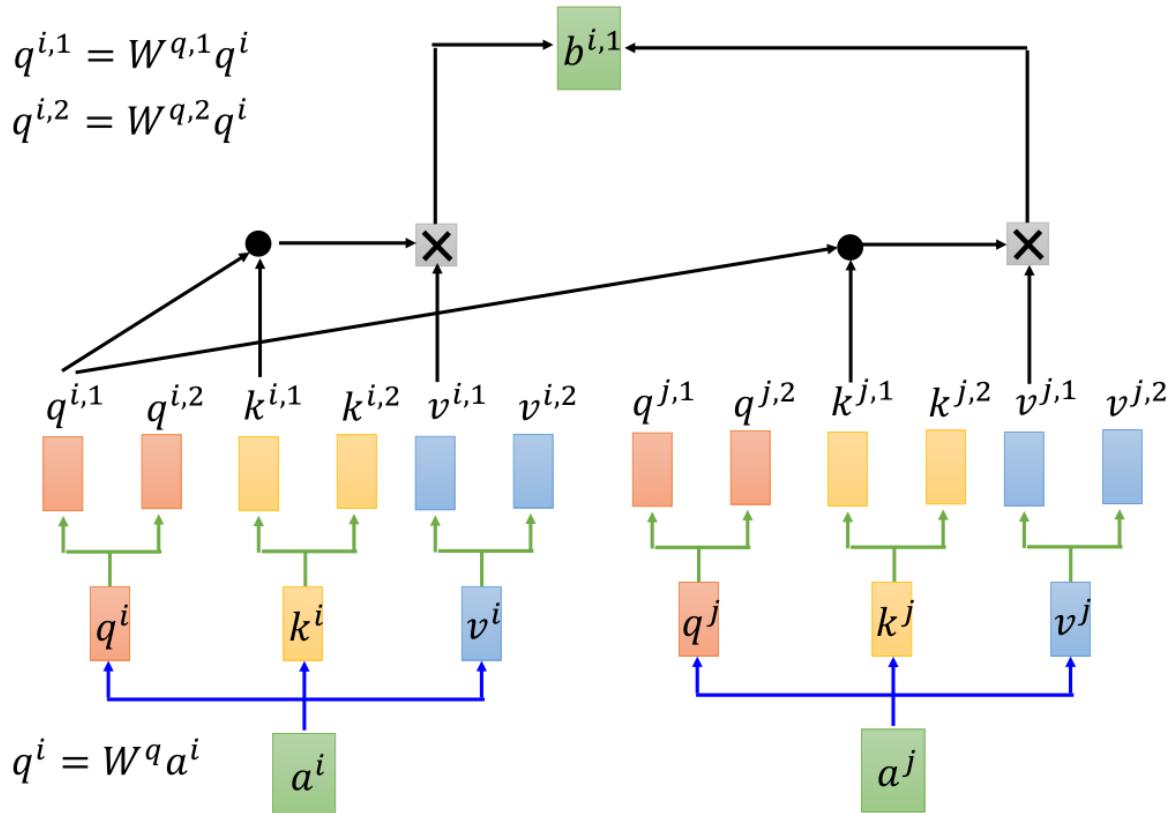
Multi-head Self-attention

(2 heads as example)

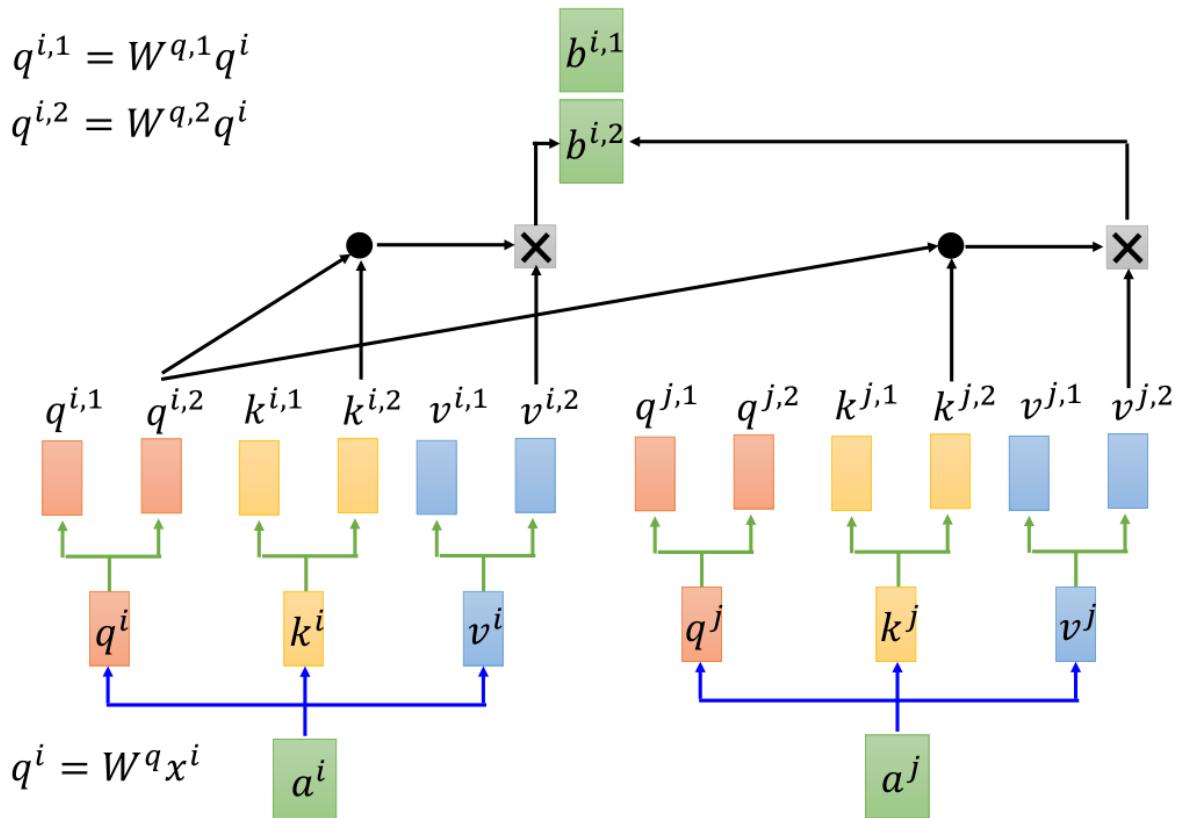
在2个head的情况下，你会进一步把 q^i 分裂，变成 $q^{i,1}$ 、 $q^{i,2}$ ，做法是 q^i 可以乘上两个矩阵 $W^{q,1}$ 、 $W^{q,2}$ 。

k^i 、 v^i 也一样，产生 $k^{i,1}$ 、 $k^{i,2}$ 和 $v^{i,1}$ 、 $v^{i,2}$ 。

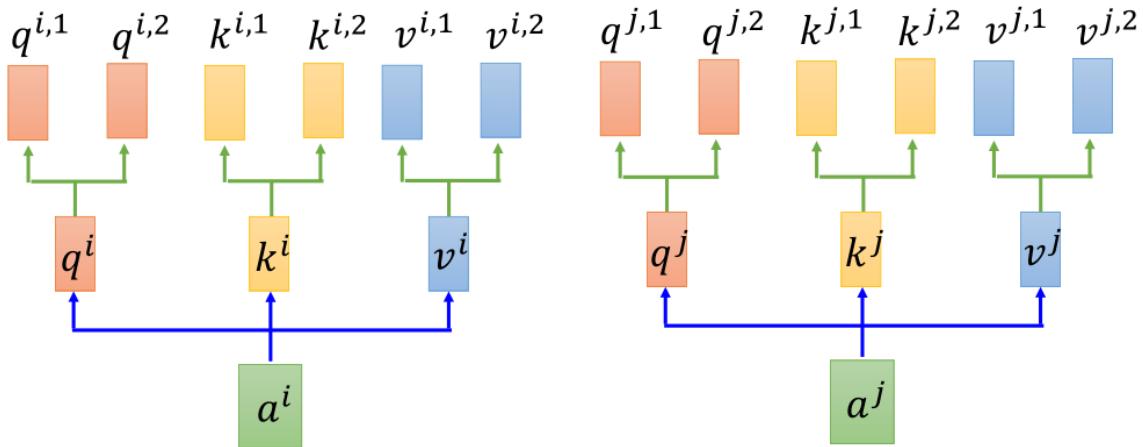
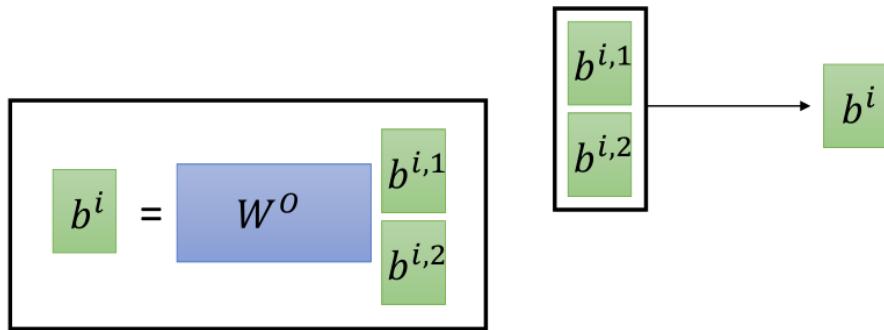
但是现在 $q^{i,1}$ 只会对 $k^{i,1}$ 、 $k^{j,1}$ (同样是第一个向量)做attention，然后计算出 $b^{i,1}$



$q^{i,2}$ 只会对 $k^{i,2}$ 、 $k^{j,2}$ 做attention，然后得到 $b^{i,2}$ 。



然后把 $b^{i,1}$ 、 $b^{i,2}$ 接在一起，如果你还想对维度做调整，那么再乘上一个矩阵 W^O 做降维就可以了。



有可能不同的head关注的点不一样，比如有的head想看的是local（短期）的信息，有的head想看的是global（长期）的信息。有了Multi-head之后，每个head可以各司其职，自己做自己想做的事情。

当然head的数目是可以自己调的，比如8个head，10个head等等都可以。

Positional Encoding

No position information in self-attention.

但是这个显然不是我们想要的，我们希望把input sequence的顺序考虑进self-attention layer里去。

在原始的paper中说，在把 x^i 变成 a^i 后，还要加上一个 e^i （要跟 a^i 的维度相同）， e^i 是人工设定的，不是学出来的，代表了位置的信息，所以每个位置都有一个不同的 e^i 。比如第一个位置为 e^1 ，第二个位置为 e^2 ……。

把 e^i 加到 a^i 后，接下来的步骤就跟之前的一样。

通常这里会想，为什么 e^i 跟 a^i 是相加，而不是拼接，相加不是把位置信息混到 a^i 里去了吗

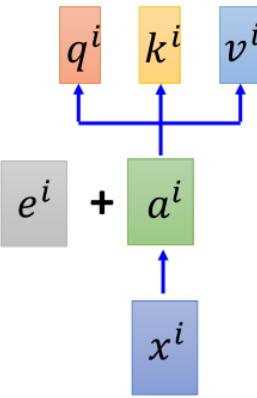
我们可以想象，给 x^i 再添加一个one-hot向量 p^i （代表了位置信息）， p^i 是一个很长的向量，位置i为1，其他为0。

x^i, p^i 拼接后乘上一个矩阵 W ，你可以想像为等于把 W 拆成两个矩阵 W^I, W^P ，之后 W^I 跟 x^i 相乘+ W^P 跟 p^i 相乘。而 W^I 跟 x^i 相乘部分就是 a^i ， W^P 跟 p^i 相乘部分是 e^i ，那么就是 $a^i + e^i$ ，所以相加也是说得通的。

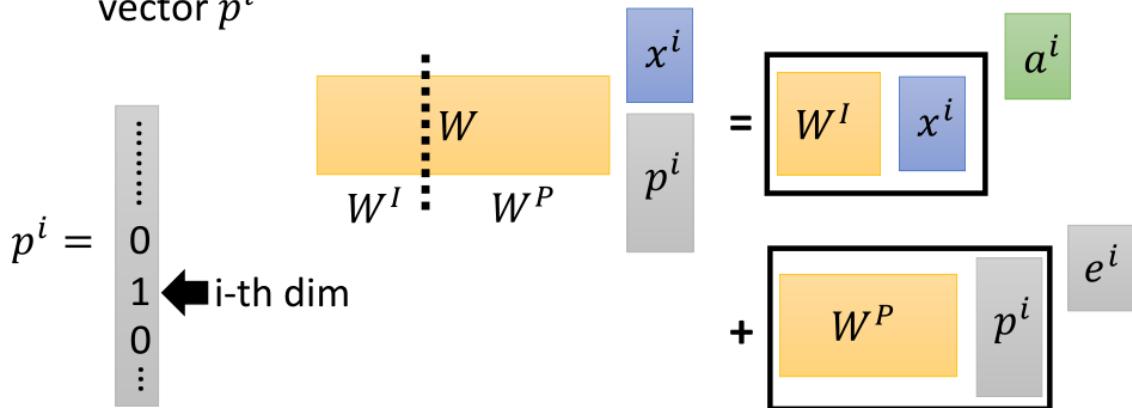
$$W \begin{pmatrix} x^i \\ p^i \end{pmatrix} = (W^I \quad W^P) \begin{pmatrix} x^i \\ p^i \end{pmatrix} = \underbrace{W^I x^i}_{a^i} + \underbrace{W^P p^i}_{e^i}$$

W^P 是可以学习的，但是有人做过实验，学出来的 W^P 效果并不如手动设定好。

Positional Encoding



- No position information in self-attention.
- Original paper: each position has a unique positional vector e^i (not learned from data)
- In other words: each x^i appends a one-hot vector p^i



Seq2seq with Attention

Encode: 所有word两两之间做attention，有三个attention layer

Decode: 不只 attend input 也会attend 之前已经输出的部分

More specifically, to compute the next representation for a given word - "bank" for example - the Transformer compares it to every other word in the sentence. The result of these comparisons is an attention score for every other word in the sentence. These attention scores determine how much each of the other words should contribute to the next representation of "bank". In the example, the disambiguating "river" could receive a high attention score when computing a new representation for "bank". The attention scores are then used as weights for a weighted average of all words' representations which is fed into a fully-connected network to generate a new representation for "bank", reflecting that the sentence is talking about a river bank.

The animation above illustrates how we apply the Transformer to machine translation. Neural networks for machine translation typically contain an encoder reading the input sentence and generating a representation of it. A decoder then generates the output sentence word by word while consulting the representation generated by the encoder. The Transformer starts by generating initial representations, or embeddings, for each word. These are represented by the unfilled circles. Then, using self-attention, it aggregates information from all of the other words, generating a new representation per word informed by the entire context, represented by the filled balls. This step is then repeated multiple times in parallel for all words, successively generating new representations.

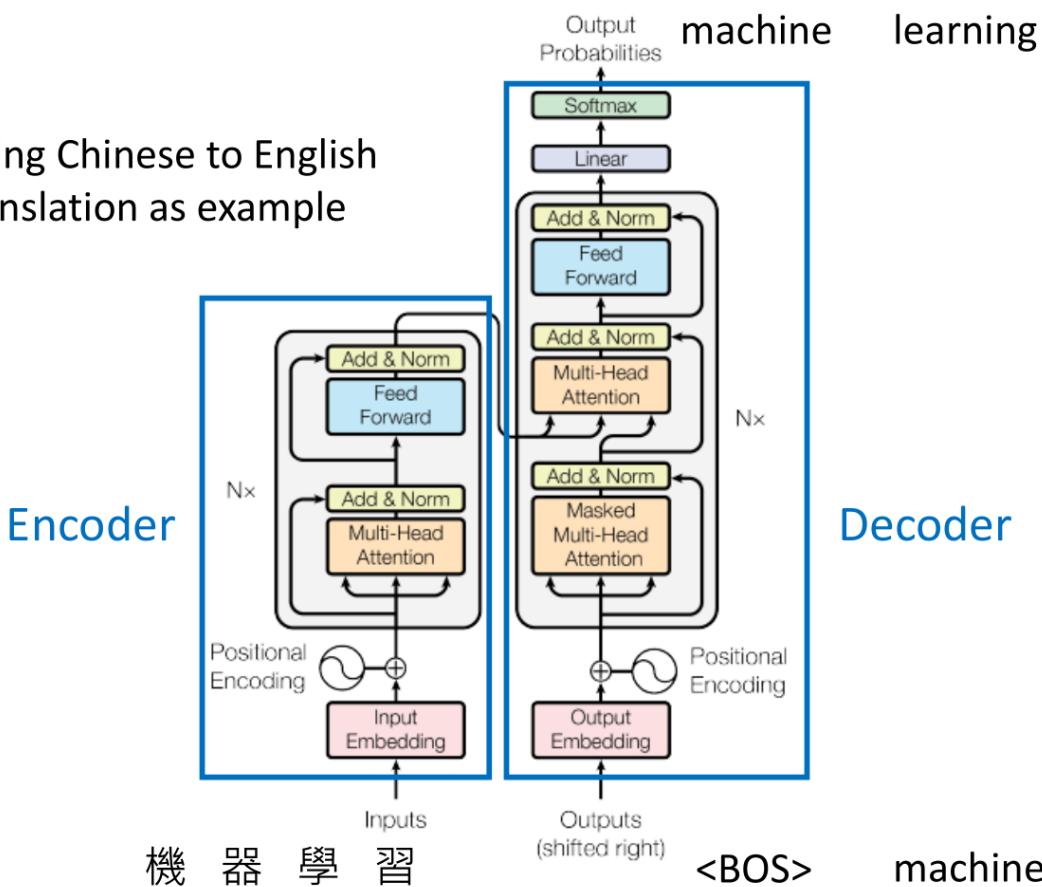
The decoder operates similarly, but generates one word at a time, from left to right. It attends not only to the other previously generated words, but also to the final representations generated by the encoder.

Transformer

Using Chinese to English translation as example

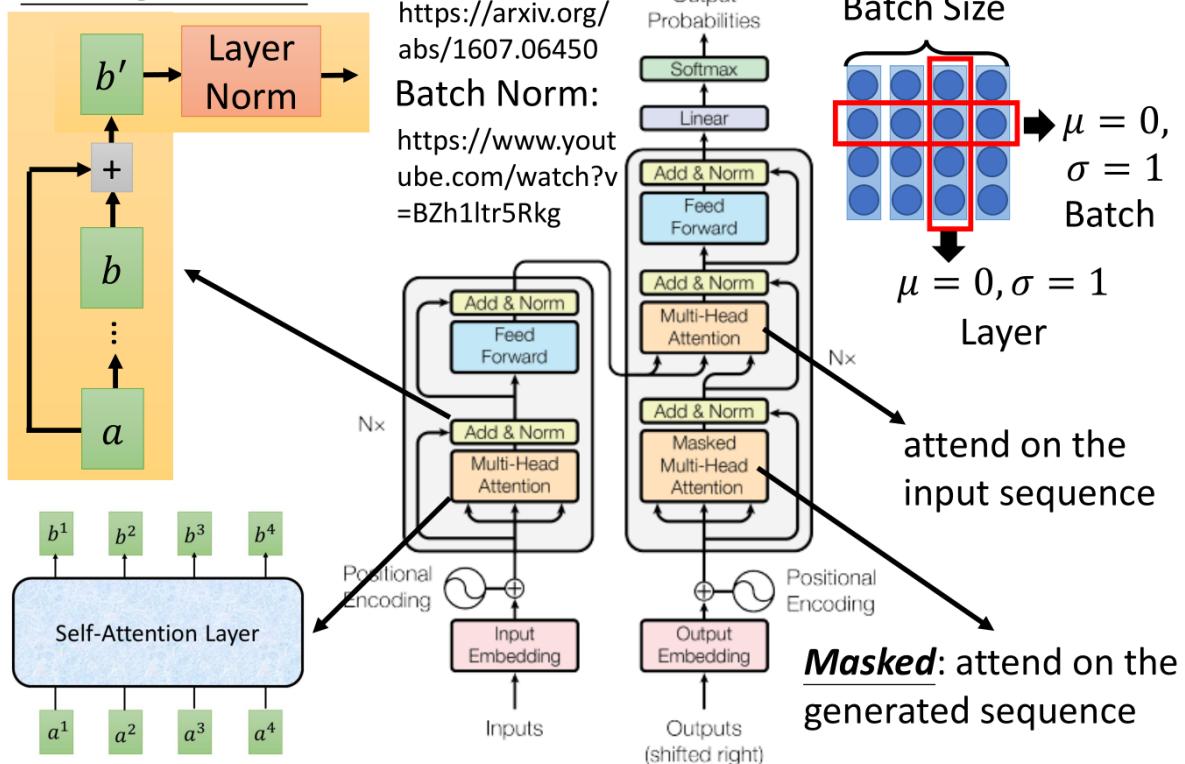
左半部是encoder，右半部是decoder。encoder的输入是机器学习（一个中文的character），decoder先给一个begin token，然后输出machine，在下一个时间点把machine作为输入，输出learning，这个翻译的过程就结束了。

Using Chinese to English translation as example



接下来看里面的每个layer在干什么。

Transformer



Encoder:

input 通过一个 input embedding layer 变成一个向量，然后加上位置 encoding 向量

然后进入灰色的 block，这个 block 会重复多次

- 第一层是 Multi-Head Attention，input 一个 sequence，输出另一个 sequence

- 第二层是Add&Norm
 - 把Multi-Head Attention的input a和output b加起来得到b' (Add)
 - 把b'再做Layer Norm，上图右上方所示为Batch Norm（行，n个样本的同一个维度标准化），和Layer Norm（列，1个样本的n个维度标准化）。一般Layer Norm会搭配RNN使用，所以这里也使用Layer Norm
- 第三层是Feed Forward，会把sequence 的每个b'向量进行处理
- 第四层是另一个Add&Norm

最终输出的是输入信号的向量表示

Decoder:

decoder的input是前一个时间点产生的output，通过output embedding，再加上位置encoding变成一个向量，然后进去灰色的block，灰色block同样会重复多次

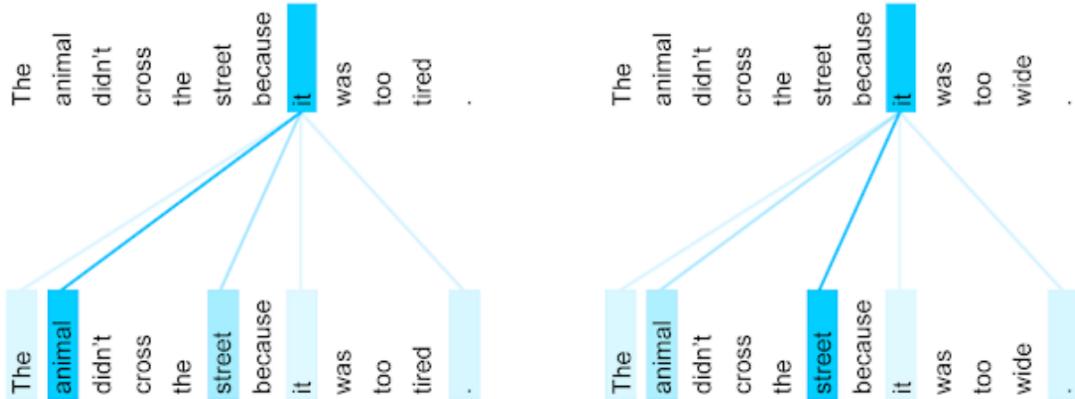
- 第一层是Masked Multi-Head Attention，Masked的意思是，在做self-attention的时候，这个decoder只会attend到已经产生的sequence，因为没有产生的部分无法做attention
- 第二层是Add&Norm
- 第三层是Multi-Head Attention，attend的是encoder部分的输出和第二层的输出结果
- 第四层是Add&Norm
- 第五层是Feed Forward
- 第六层是Add&Norm

不再循环后，进行Linear，最后再进行softmax

Attention Visualization

Transformer paper最后附上了一些attention的可视化，每两个word之间都会有一个attention。attention权重越大，线条颜色越深。

现在input一个句子，在做attention的时候，你会发现it attend到animal；但是把tired换成wide，it会attend到street。

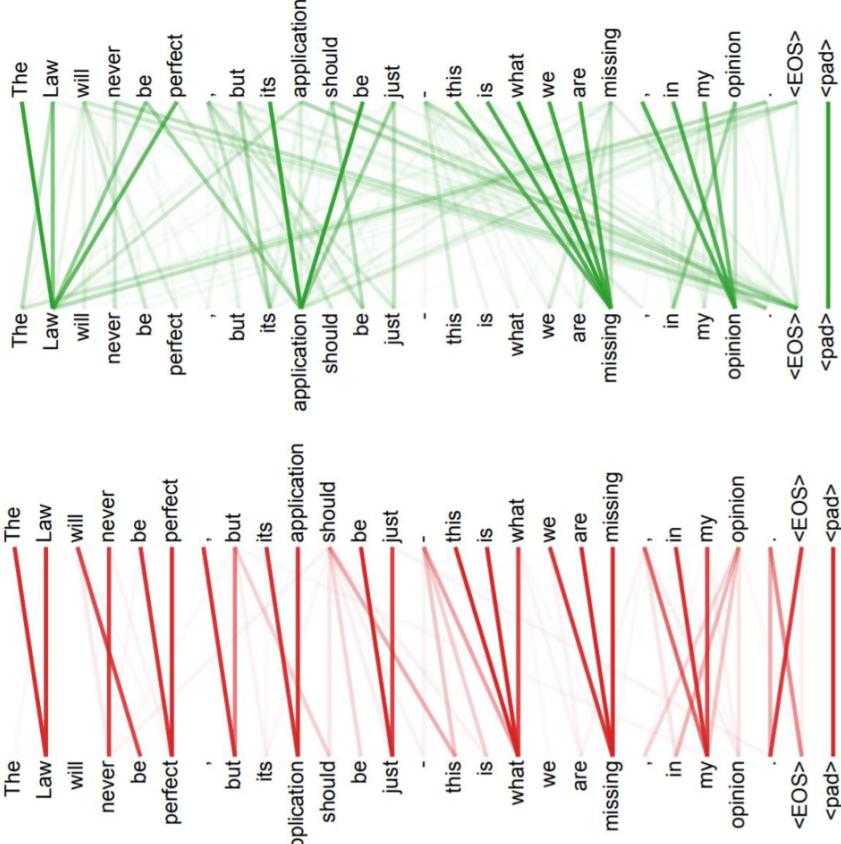


The encoder self-attention distribution for the word "it" from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

<https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html>

对于Multi-head attention，每一组 q, k 都做不同的事情，这里会发现，确实是这样。如下图所示，上面的部分可以看出这个head主要关注比较长序列（global）的信息，而下面的head比较关注距自己相近的序列（local）的信息，说明使用多个head时不同的head通过学习会关注不同的信息。

Multi-head Attention



Example Application

使用Transformer可以做多文档摘要，通过训练一个Summarizer来输入一个文档的集合然后输出这些文档的摘要。<https://arxiv.org/abs/1801.10198>

Transformer很好地解决了输入序列长度较大的情况，而向RNN中输入长序列结果通常不会好。

Universal Transformer

将Transformer在深度上随时间循环使用，即重复使用相同的网络结构。

<https://ai.googleblog.com/2018/08/moving-beyond-translation-with.html>

Self-Attention GAN

将Transformer用在影像上，用每一个pixel去attention其他pixel，这样可以考虑到比较global的信息。

<https://arxiv.org/abs/1805.08318>

Unsupervised Learning

Word Embedding

本文介绍NLP中词嵌入(Word Embedding)相关的基础知识，基于降维思想提供了count-based和prediction-based两种方法，并介绍了该思想在机器问答、机器翻译、图像分类、文档嵌入等方面的应用

Introduction

词嵌入(word embedding)是降维算法(Dimension Reduction)的典型应用

那如何用vector来表示一个word呢?

1-of-N Encoding

最传统的做法是1-of-N Encoding, 假设这个vector的维数就等于世界上所有单词的数目, 那么对每一个单词来说, 只需要某一维为1, 其余都是0即可; 但这会导致任意两个vector都是不一样的, 你无法建立起同类word之间的联系

Word Class

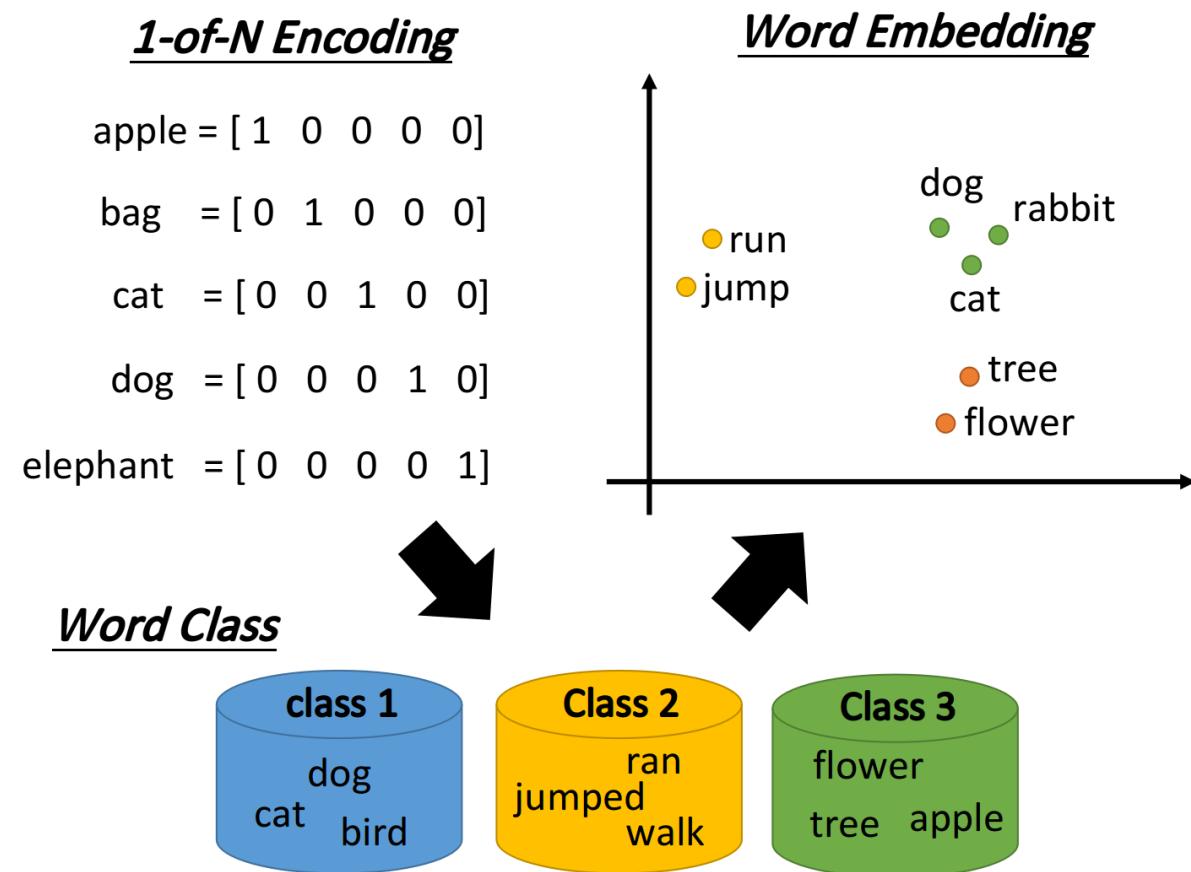
还可以把有同样性质的word进行聚类(clustering), 划分成多个class, 然后用word所属的class来表示这个word, 但光做clustering是不够的, 不同class之间关联依旧无法被有效地表达出来

Word Embedding

词嵌入(Word Embedding)把每一个word都投影到高维空间上, 当然这个空间的维度要远比1-of-N Encoding的维度低, 假如后者有10w维, 那前者只需要50~100维就够了, 这实际上也是Dimension Reduction的过程

类似语义(semantic)的词汇, 在这个word embedding的投影空间上是比较接近的, 而且该空间里的每一维都可能有特殊的含义

假设词嵌入的投影空间如下图所示, 则横轴代表了生物与其它东西之间的区别, 而纵轴则代表了会动的东西与静止的东西之间的差别



word embedding是一个无监督的方法(unsupervised approach), 只要让机器阅读大量的文章, 它就可以知道每一个词汇embedding之后的特征向量应该长什么样子

Machine learns the meaning of words from reading a lot of documents without supervision

我们的任务就是训练一个neural network, input是词汇, output则是它所对应的word embedding vector, 实际训练的时候我们只有data的input, 该如何解这类问题呢?

之前提到过一种基于神经网络的降维方法, Auto-Encoder, 就是训练一个model, 让它的输入等于输出, 取出中间的某个隐藏层就是降维的结果, 自编码的本质就是通过自我压缩和解压的过程来寻找各个维度之间的相关信息。但word embedding这个问题是不能用Auto-encoder来解的, 因为输入的向量通常是1-of-N encoding, 各维无关, 很难通过自编码的过程提取出什么有用信息。

Word Embedding

基本精神就是, 每一个词汇的含义都可以根据它的上下文来得到

A word can be understood by its context

比如机器在两个不同的地方阅读到了“马英九宣誓就职”、“蔡英文宣誓就职”, 它就会发现“马英九”和“蔡英文”前后都有类似的文字内容, 于是机器就可以推测“马英九”和“蔡英文”这两个词汇代表了可能有同样地位的东西, 即使它并不知道这两个词汇是人名

怎么用这个思想来找出word embedding的vector呢? 有两种做法:

- Count based
- Prediction based

Count based

假如 w_i 和 w_j 这两个词汇常常在同一篇文章中出现(co-occur), 它们的word vector分别用 $V(w_i)$ 和 $V(w_j)$ 来表示, 则 $V(w_i)$ 和 $V(w_j)$ 会比较接近

假设 $N_{i,j}$ 是 w_i 和 w_j 这两个词汇在相同文章里同时出现的次数, 我们希望它与 $V(w_i) \cdot V(w_j)$ 的内积越接近越好, 这个思想和之前的文章中提到的矩阵分解(matrix factorization)的思想其实是一样的

这种方法有一个很代表性的例子是Glove Vector

Prediction based

how to do prediction

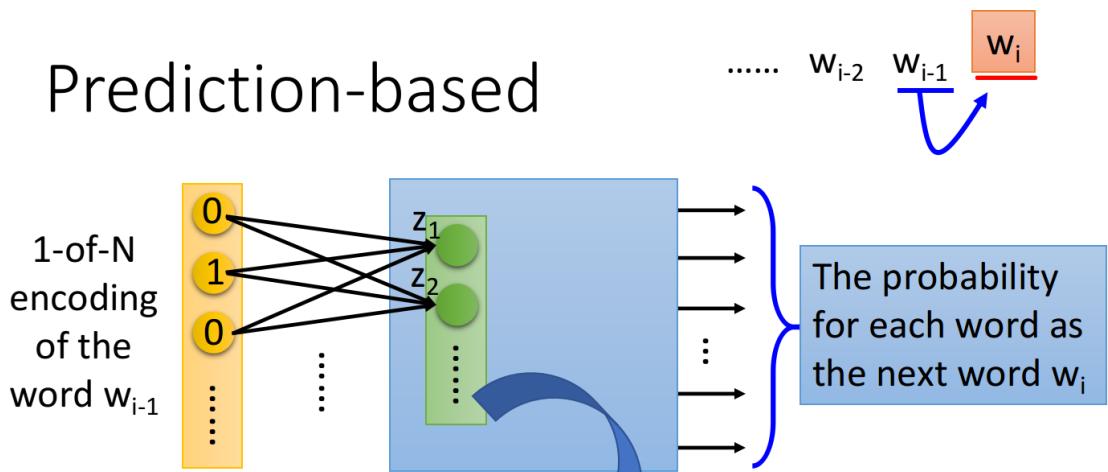
给定一个sentence, 我们要训练一个神经网络, 它要做的就是根据当前的word w_{i-1} , 来预测下一个可能出现的word w_i 是什么

假设我们使用1-of-N encoding把 w_{i-1} 表示成feature vector, 它作为neural network的input。output的维数和input相等, 只不过每一维都是小数, 代表在1-of-N编码中该维为1其余维为0所对应的word会是下一个word w_i 的概率

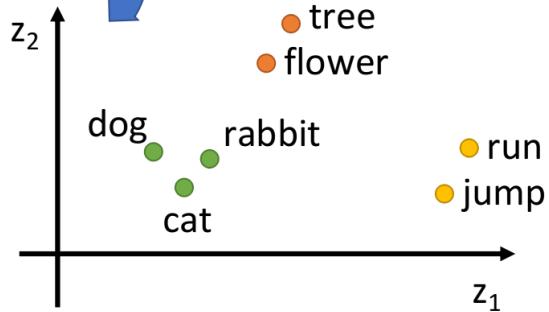
把第一个hidden layer的input z_1, z_2, \dots 拿出来, 它们所组成的Z就是word的另一种表示方式, 当我们input不同的词汇, 向量Z就会发生变化

也就是说, 第一层hidden layer的维数可以由我们决定, 而它的input又唯一确定了一个word, 因此提取出第一层hidden layer的input, 实际上就得到了一组可以自定义维数的Word Embedding的向量

Prediction-based



- Take out the input of the neurons in the first layer
- Use it to represent a word w
- Word vector, word embedding feature: $V(w)$



Why prediction works

prediction-based方法是如何体现根据词汇的上下文来了解该词汇的含义这件事呢？

假设在两篇文章中，“蔡英文”和“马英九”代表 w_{i-1} ，“宣誓就职”代表 w_i ，我们希望对神经网络输入“蔡英文”或“马英九”这两个词汇，输出的vector中对应“宣誓就职”词汇的那个维度的概率值是高的

为了使这两个不同的input通过NN能得到相同的output，就必须在进入hidden layer之前，就通过weight的转换将这两个input vector投影到位置相近的低维空间上

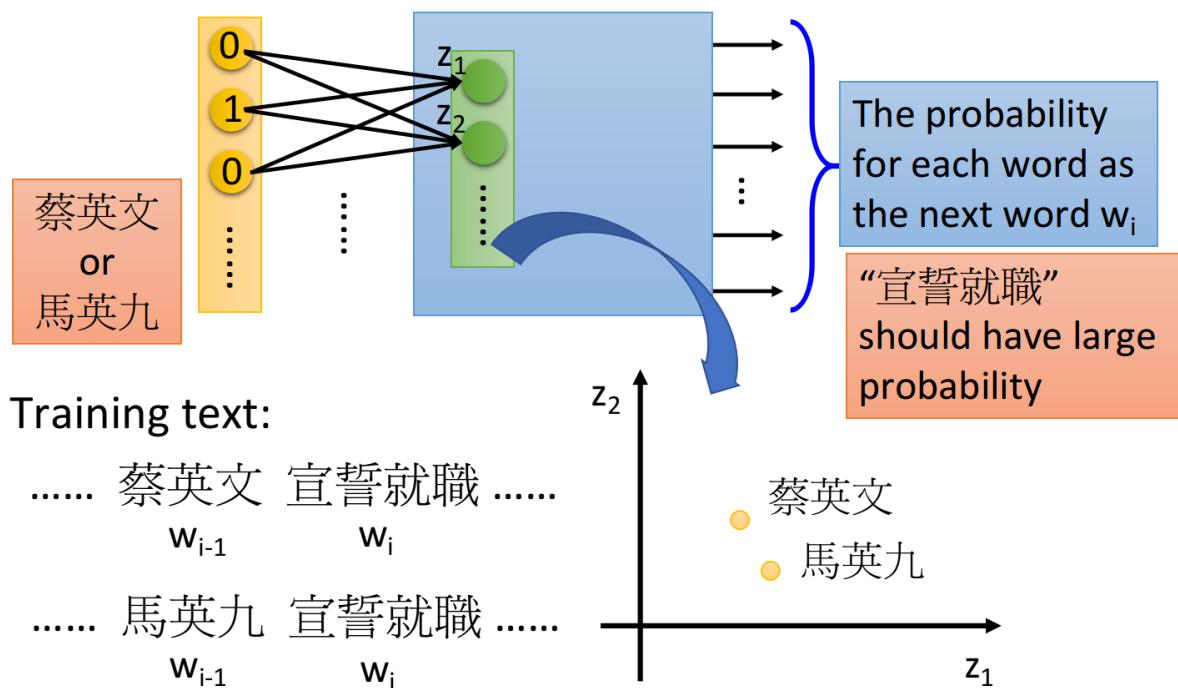
也就是说，尽管两个input vector作为1-of-N编码看起来完全不同，但经过参数的转换，将两者都降维到某一个空间中，在这个空间里，经过转换后的new vector 1和vector 2是非常接近的，因此它们同时进入一系列的hidden layer，最终输出时得到的output是相同的

因此，词汇上下文的联系就自动被考虑在这个prediction model里面

总结一下，对1-of-N编码进行Word Embedding降维的结果就是神经网络模型第一层hidden layer的输入向量 $[z_1 \ z_2 \ \dots]^T$ ，该向量同时也考虑了上下文词汇的关联，我们可以通过控制第一层hidden layer的大小从而控制目标降维空间的维数

Prediction-based

You shall know a word
by the company it keeps



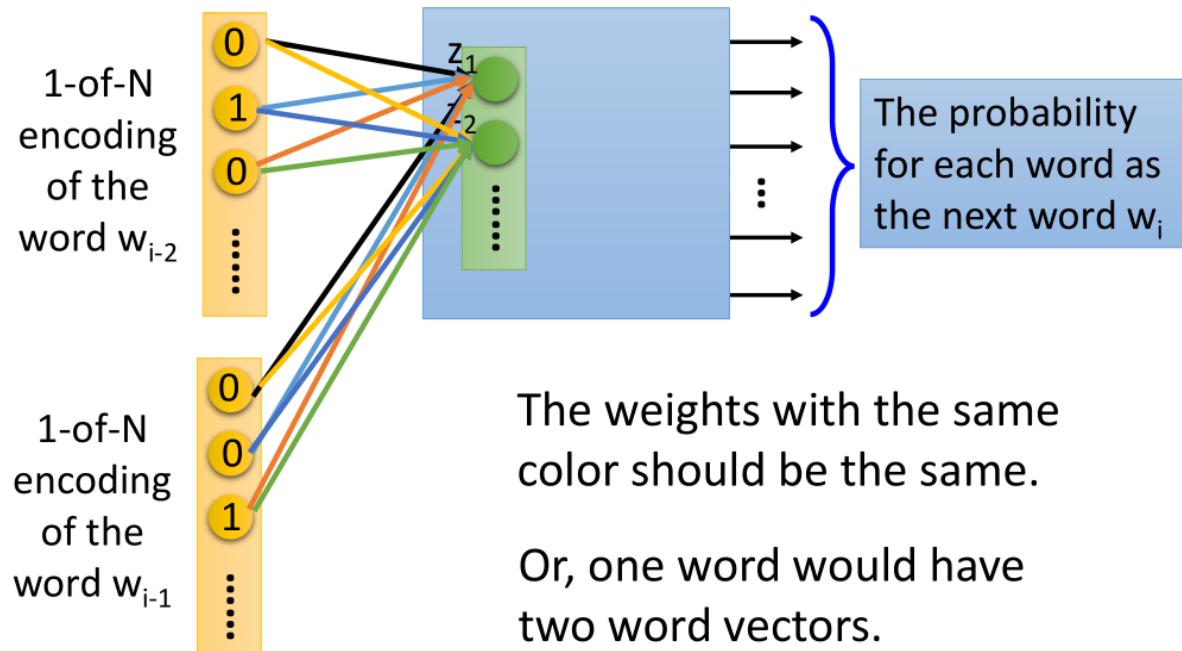
Sharing Parameters

你可能会觉得通过当前词汇预测下一个词汇这个约束太弱了，由于不同词汇的搭配千千万万，即便是人也无法准确地给出下一个词汇具体是什么

你可以扩展这个问题，使用10个及以上的词汇去预测下一个词汇，可以帮助得到较好的结果

这里用2个词汇举例，如果是一般神经网络，我们直接把 w_{i-2} 和 w_{i-1} 这两个vector拼接成一个更长的vector作为input即可

但实际上，我们希望和 w_{i-2} 相连的weight与和 w_{i-1} 相连的weight是tight在一起的，简单来说就是 w_{i-2} 与 w_{i-1} 的相同dimension对应到第一层hidden layer相同neuron之间的连线拥有相同的weight，在下图中，用同样的颜色标注相同的weight：



如果我们不这么做，那把同一个word放在 w_{i-2} 的位置和放在 w_{i-1} 的位置，得到的Embedding结果是会不一样的，把两组weight设置成相同，可以使 w_{i-2} 与 w_{i-1} 的相对位置不会对结果产生影响

除此之外，这么做还可以通过共享参数的方式有效地减少参数量，不会由于input的word数量增加而导致参数量剧增

Formulation

假设 w_{i-2} 的1-of-N编码为 x_{i-2} , w_{i-1} 的1-of-N编码为 x_{i-1} , 维数均为 $|V|$, 表示数据中的words总数

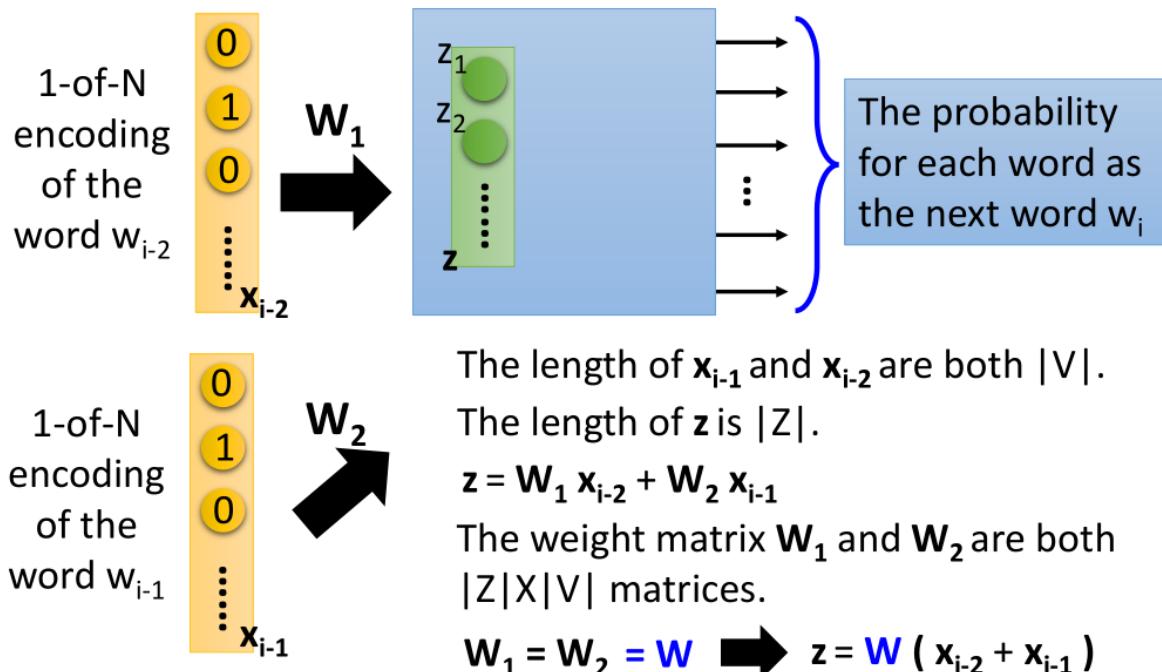
hidden layer的input为向量 z , 长度为 $|Z|$, 表示降维后的维数

$$z = W_1 x_{i-2} + W_2 x_{i-1}$$

其中 W_1 和 W_2 都是 $|Z| \times |V|$ 维的weight matrix, 它由 $|Z|$ 组 $|V|$ 维的向量构成, 第一组 $|V|$ 维向量与 $|V|$ 维的 x_{i-2} 相乘得到 z_1 , 第二组 $|V|$ 维向量与 $|V|$ 维的 x_{i-2} 相乘得到 z_2 , ..., 依次类推

我们强迫让 $W_1 = W_2 = W$, 此时 $z = W(x_{i-2} + x_{i-1})$

因此，只要我们得到了这组参数 W , 就可以与1-of-N编码 x 相乘得到word embedding的结果 z



In Practice

那在实际操作上，我们如何保证 W_1 和 W_2 一样呢？

以下图中的 w_i 和 w_j 为例，我们希望它们的weight是一样的：

- 首先在训练的时候就要给它们一样的初始值
- 然后分别计算loss function C 对 w_i 和 w_j 的偏微分，并对其进行更新

$$w_i = w_i - \eta \frac{\partial C}{\partial w_i}$$

$$w_j = w_j - \eta \frac{\partial C}{\partial w_j}$$

这个时候你就会发现， C 对 w_i 和 w_j 的偏微分是不一样的，这意味着即使给了 w_i 和 w_j 相同的初始值，更新过一次之后它们的值也会变得不一样，因此我们必须保证两者的更新过程是一致的，即：

$$w_i = w_i - \eta \frac{\partial C}{\partial w_i} - \eta \frac{\partial C}{\partial w_j}$$

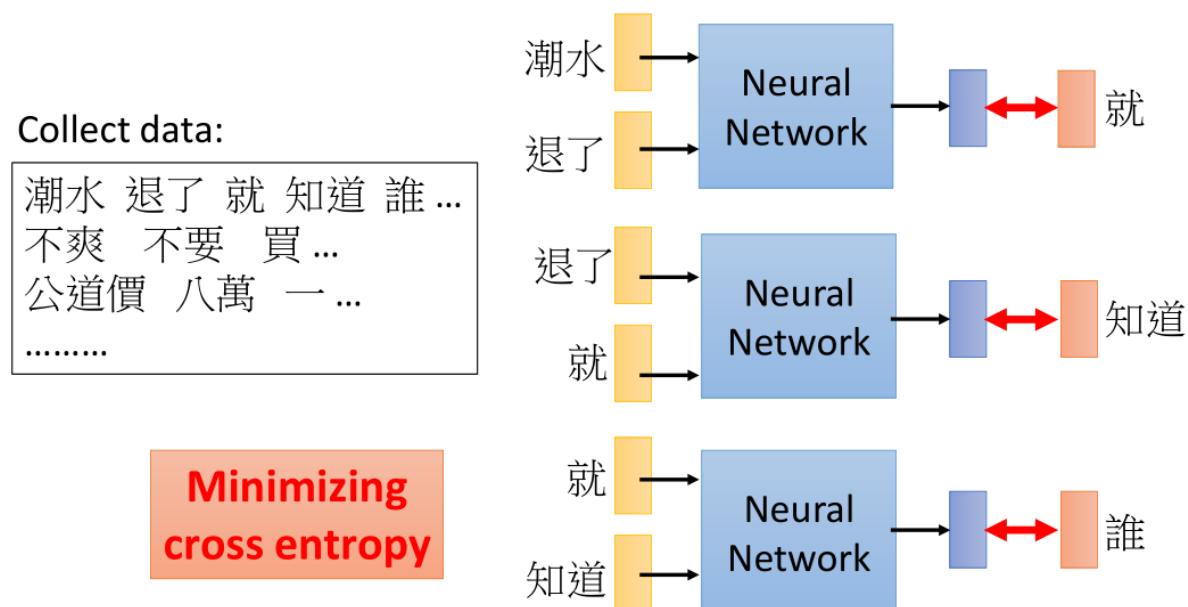
$$w_j = w_j - \eta \frac{\partial C}{\partial w_j} - \eta \frac{\partial C}{\partial w_i}$$

- 这个时候，我们就保证了 w_i 和 w_j 始终相等：

- w_i 和 w_j 的初始值相同
- w_i 和 w_j 的更新过程相同

如何去训练这个神经网络呢？注意到这个NN完全是unsupervised，你只需要上网爬一下文章数据直接喂给它即可

比如喂给NN的input是“潮水”和“退了”，希望它的output是“就”，之前提到这个NN的输出是一个由概率组成的vector，而目标“就”是只有某一维为1的1-of-N编码，我们希望minimize它们之间的cross entropy，也就是使得输出的那个vector在“就”所对应的那一维上概率最高



Various Architectures

除了上面的基本形态，Prediction-based方法还可以有多种变形

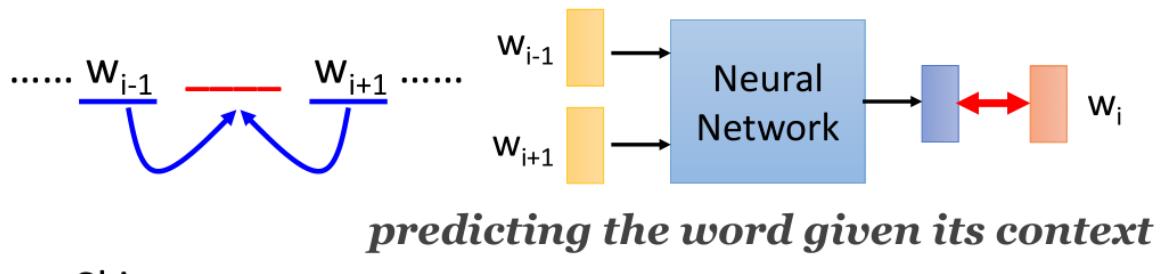
- CBOW(Continuous bag of word model)

拿前后的词汇去预测中间的词汇

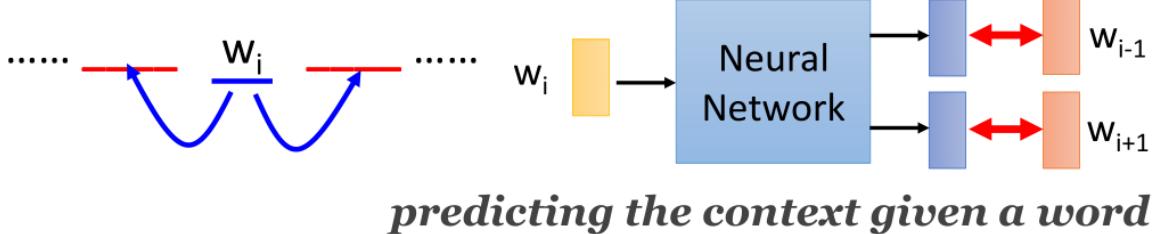
- Skip-gram

拿中间的词汇去预测前后的词汇

- Continuous bag of word (CBOW) model



- Skip-gram



Others

假设你有读过word vector的文献的话，你会发现这个neural network其实并不是deep的，它就只有一个linear的hidden layer

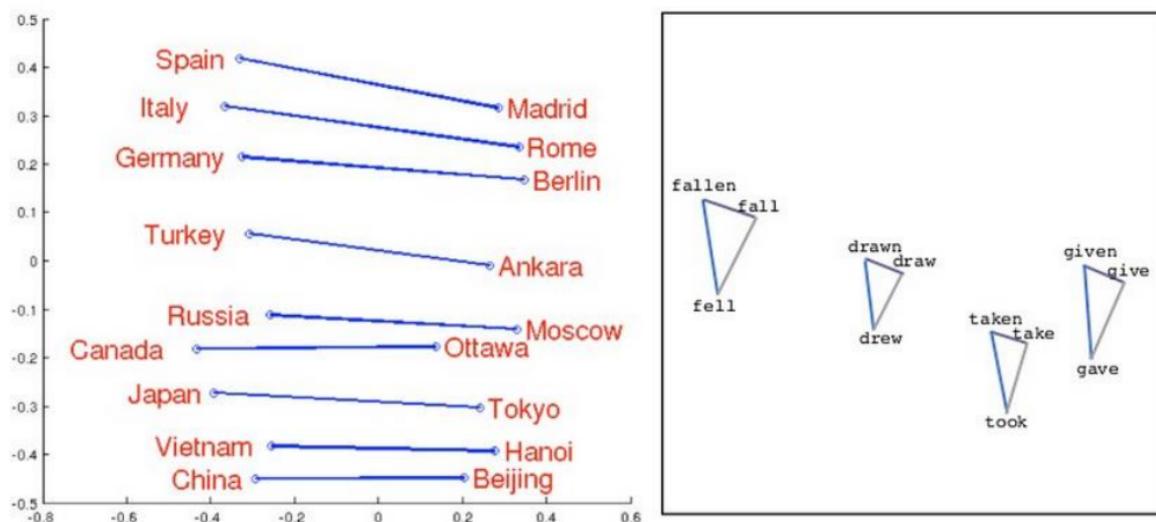
我们把1-of-N编码输入给神经网络，经过weight的转换得到Word Embedding，再通过第一层hidden layer就可以直接得到输出

其实过去有很多人使用过deep model，但这个task不用deep就可以实现，这样做既可以减少运算量，跑大量的data，又可以节省下训练的时间(deep model很可能需要长达好几天的训练时间)

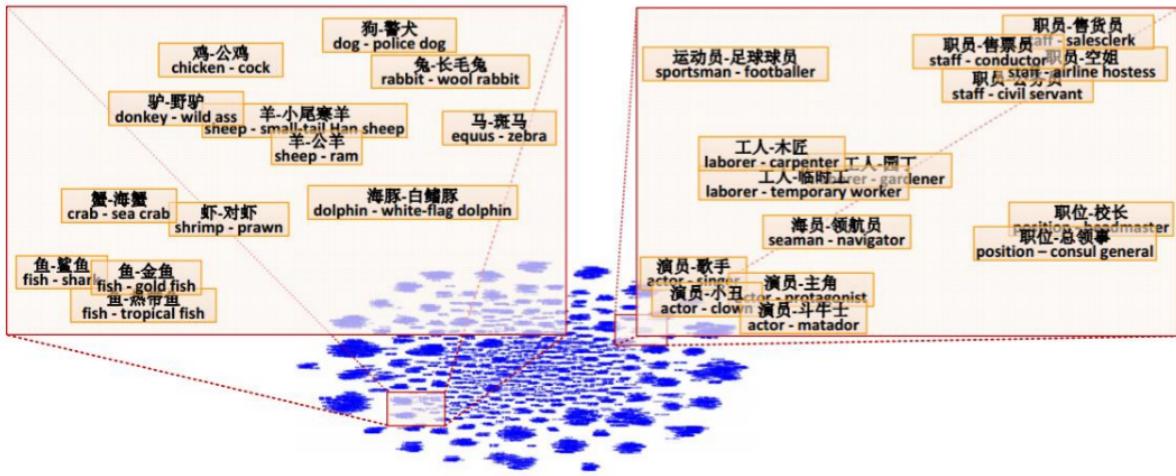
Application

Word Embedding

从得到的word vector里，我们可以发现一些原本并不知道的word与word之间的关系



把word vector两两相减，再投影到下图中的二维平面上，如果某两个word之间有类似包含于的相同关系，它们就会被投影到同一块区域



Fu, Ruiji, et al. "Learning semantic hierarchies via word embeddings." *Proceedings of the 52th Annual Meeting of the Association for Computational Linguistics: Long Papers*. Vol. 1. 2014.

利用这个概念，我们可以做一些简单的推论：

- 在word vector的特征上， $V(Rome) - V(Italy) \approx V(Berlin) - V(Germany)$
- 此时如果有人问“罗马之于意大利等于柏林之于？”，那机器就可以回答这个问题

因为德国的vector会很接近于“柏林的vector-罗马的vector+意大利的vector”，因此机器只需要计算 $V(Berlin) - V(Rome) + V(Italy)$ ，然后选取与这个结果最接近的vector即可

$$\begin{aligned} & V(Germany) \\ \bullet \text{ Characteristics} \quad & \approx V(Berlin) - V(Rome) + V(Italy) \end{aligned}$$

$$V(hotter) - V(hot) \approx V(bigger) - V(big)$$

$$V(Rome) - V(Italy) \approx V(Berlin) - V(Germany)$$

$$V(king) - V(queen) \approx V(uncle) - V(aunt)$$

• Solving analogies

$$Rome : Italy = Berlin : ?$$

$$\text{Compute } \underline{V(Berlin) - V(Rome) + V(Italy)}$$

Find the word w with the closest $V(w)$

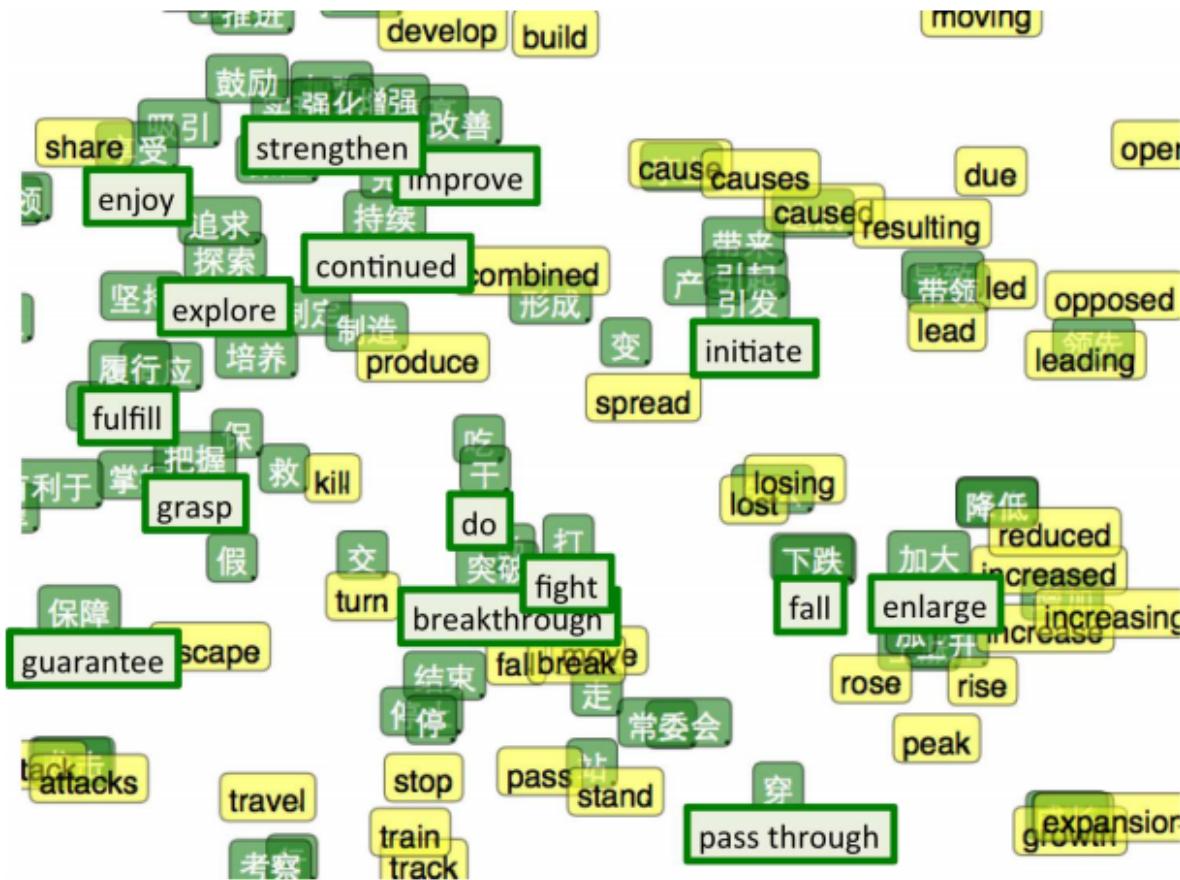
Multi-lingual Embedding

此外，word vector还可以建立起不同语言之间的联系

如果你要用上述方法分别训练一个英文的语料库(corpus)和中文的语料库，你会发现两者的word vector之间是没有任何关系的，因为Word Embedding只体现了上下文的关系，如果你的文章没有把中英文混合在一起使用，机器就没有办法判断中英文词汇之间的关系

但是，如果你知道某些中文词汇和英文词汇的对应关系，你可以先分别获取它们的word vector，然后再去训练一个模型，把具有相同含义的中英文词汇投影到新空间上的同一个点

接下来遇到未知的新词汇，无论是中文还是英文，你都可以采用同样的方式将其投影到新空间，就可以自动做到类似翻译的效果



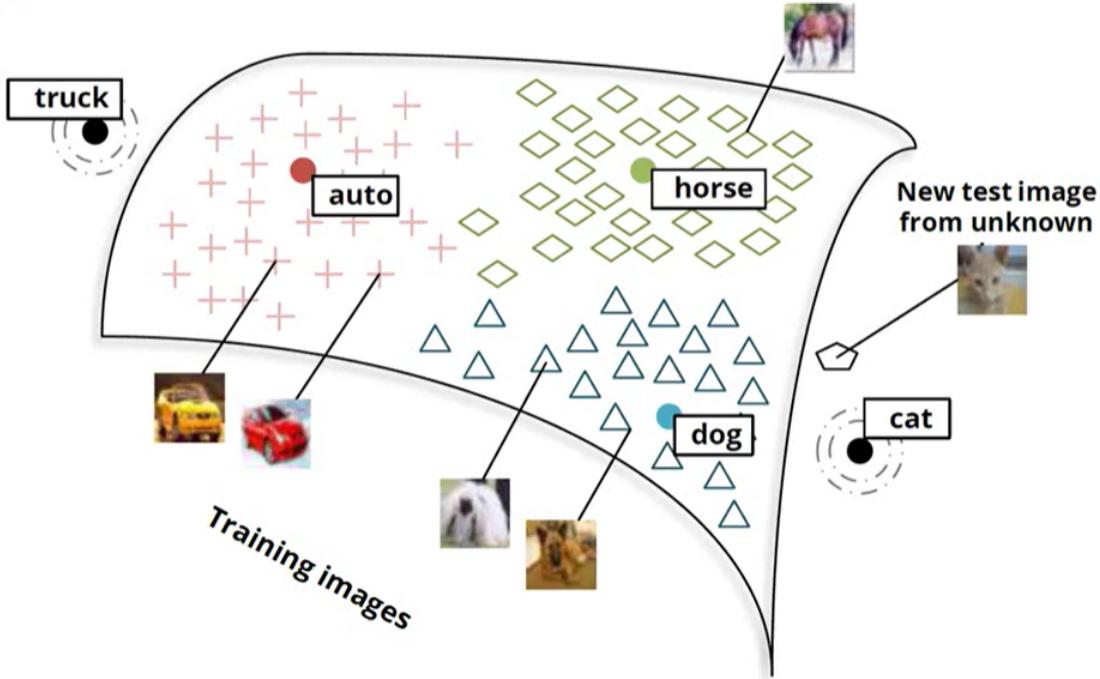
参考文献：Bilingual Word Embeddings for Phrase-Based Machine Translation, Will Zou, Richard Socher, Daniel Cer and Christopher Manning, EMNLP, 2013

Multi-domain Embedding

在这个word embedding 不局限于文字，你可以对影像做embedding。举例：我们现在已经找好一组 word vector, dog vector, horse vector, auto vector, cat vector 在空间中是这个样子。接下来，你 learn一个model，input一张image，output是跟word vector一样dimension的vector。你会希望说，狗的 vector分布在狗的周围，马的vector散布的马的周围，车辆的vector散布在auto的周围，你可以把影像的 vector project到它们对应的word vector附近。

假设有一张新的image进来(它是猫，但是你不知道它是猫)，你通过同样的projection把它project这个 space以后。神奇的是，你发现它就可能在猫的附近，machine就会知道这是个猫。我们一般做影像分类的时候，你的machine很难去处理新增加的，它没有看过的图片。如果你用这个方法的话，就算有一张 image，在training的时候你没有看到过的class。比如说猫这个image，从来都没有看过，但是猫这个 image project到cat附近的话，你就会说，这张image叫做cat。

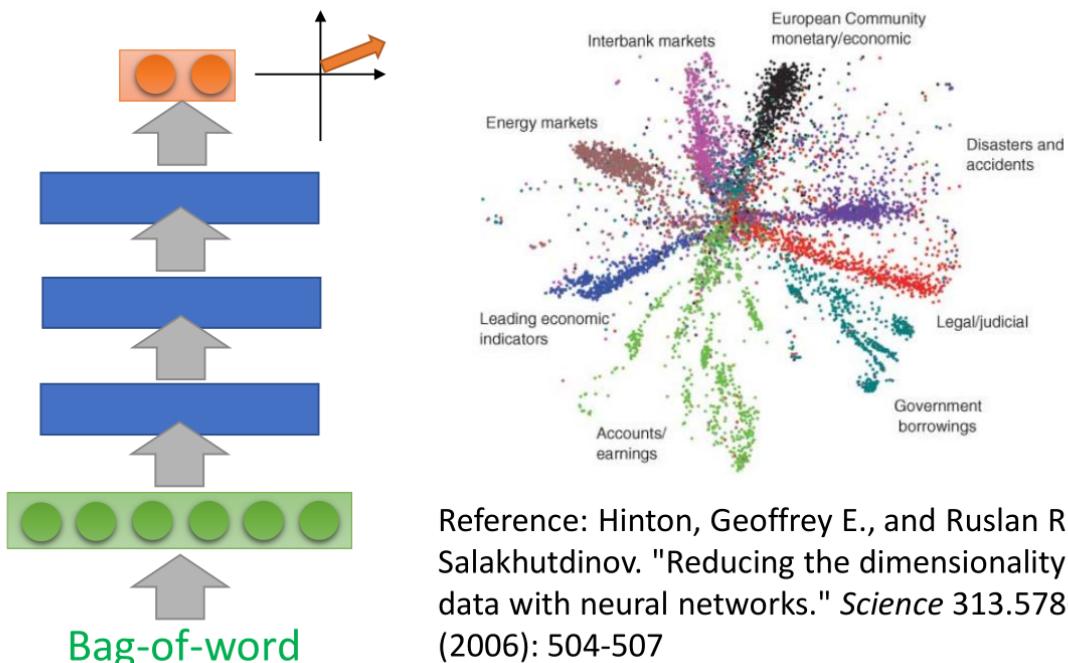
如果你可以做到这件事的话，就好像是machine阅读了大量的文章以后，它知道说：每一个词汇它是什么意思。先通过阅读大量的文章，先了解词汇之间的关系，接下来再看image的时候，会根据它阅读的知识去match每一个image所该对应的位置。这样就算它没有看过的东西，它也有可能把它的名字叫出来。



Document Embedding

除了Word Embedding，我们还可以对Document做Embedding

最简单的方法是把document变成bag-of-word，然后用Auto-encoder就可以得到该文档的语义嵌入(Semantic Embedding)，但光这么做是不够的

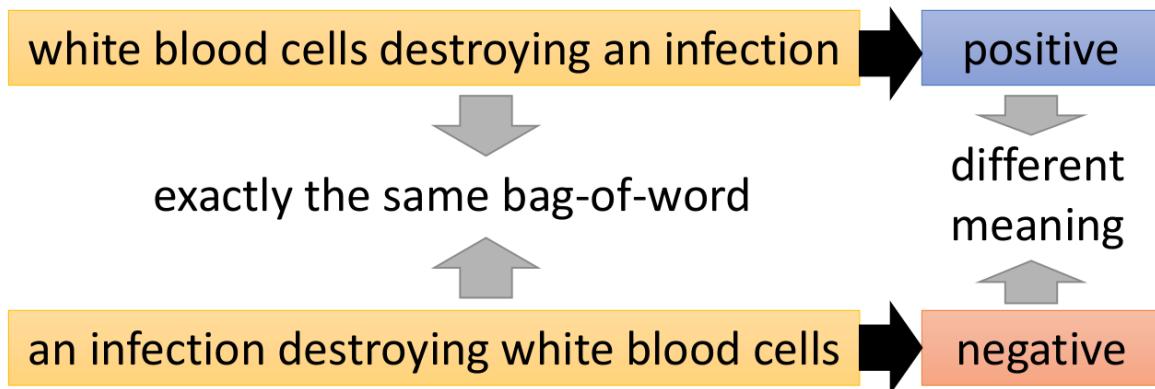


Reference: Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313.5786 (2006): 504-507

词汇的顺序代表了很重要的含义，两句话相同但语序不同的话可能会有完全不同的含义，比如

- 白血球消灭了传染病——正面语义
- 传染病消灭了白血球——负面语义

- To understand the meaning of a word sequence, the order of the words can not be ignored.



想要解决这个问题，具体可以参考下面的几种处理方法 (Unsupervised) :

- **Paragraph Vector:** Le, Quoc, and Tomas Mikolov. "Distributed Representations of Sentences and Documents." ICML, 2014
- **Seq2seq Auto-encoder:** Li, Jiwei, Minh-Thang Luong, and Dan Jurafsky. "A hierarchical neural autoencoder for paragraphs and documents." arXiv preprint, 2015
- **Skip Thought:** Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, Sanja Fidler, "Skip-Thought Vectors" arXiv preprint, 2015.

Principle Component Analysis

Unsupervised Learning

无监督学习(Unsupervised Learning)可以分为两种：

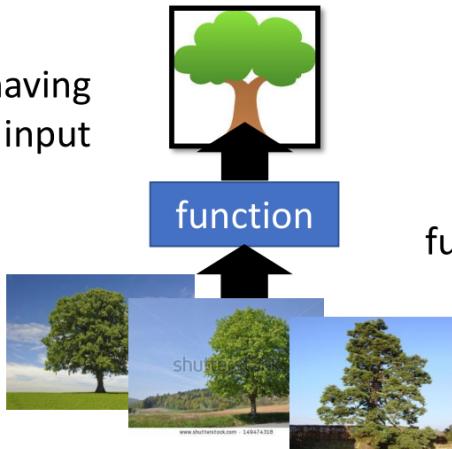
- 化繁为简
 - 聚类(Clustering)
 - 降维(Dimension Reduction)
- 无中生有(Generation)

对于无监督学习(Unsupervised Learning)来说，我们通常只会拥有 (x, \hat{y}) 中的 x 或 \hat{y} ，其中：

- **化繁为简**就是把复杂的input变成比较简单的output，比如把一大堆没有打上label的树图片转变为一棵抽象的树，此时training data只有input x ，而没有output \hat{y}
- **无中生有**就是随机给function一个数字，它就会生成不同的图像，此时training data没有input x ，而只有output \hat{y}

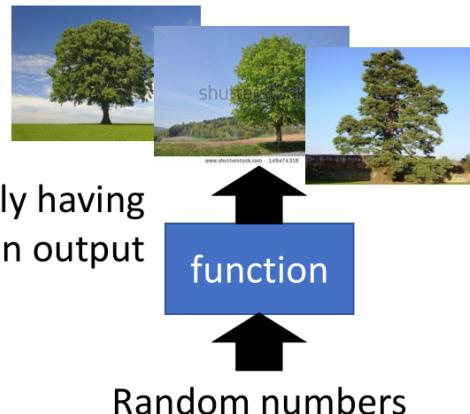
- Dimension Reduction
(化繁為簡)

only having
function input



- Generation (無中生有)

only having
function output



Clustering

聚类，顾名思义，就是把相近的样本划分为同一类，比如对下面这些没有标签的image进行分类，手动打上cluster 1、cluster 2、cluster 3的标签，这个分类过程就是化繁为简的过程

一个很critical的问题：我们到底要分几个cluster？

K-means

最常用的方法是**K-means**：

- 我们有一大堆的unlabeled data $\{x^1, \dots, x^n, \dots, x^N\}$, 我们要把它划分为K个cluster
- 对每个cluster都要找一个center $c^i, i \in \{1, 2, \dots, K\}$, initial的时候可以从training data里随机挑K个object x^n 出来作为K个center c^i 的初始值
- Repeat
 - 遍历所有的object x^n , 并判断它属于哪一个cluster, 如果 x^n 与第i个cluster的center c^i 最接近, 那它就属于该cluster, 我们用 $b_i^n = 1$ 来表示第n个object属于第i个cluster, $b_i^n = 0$ 表示不属于
 - 更新center: 把每个cluster里的所有object取平均值作为新的center值, 即 $c^i = \sum_{x^n} b_i^n x^n / \sum_{x^n} b_i^n$

注：如果不是从原先的数据集里取center的初始值，可能会导致部分cluster没有样本点

HAC

HAC, 全称Hierarchical Agglomerative Clustering, 层次聚类

假设现在我们有5个样本点，想要做clustering：

- build a tree:

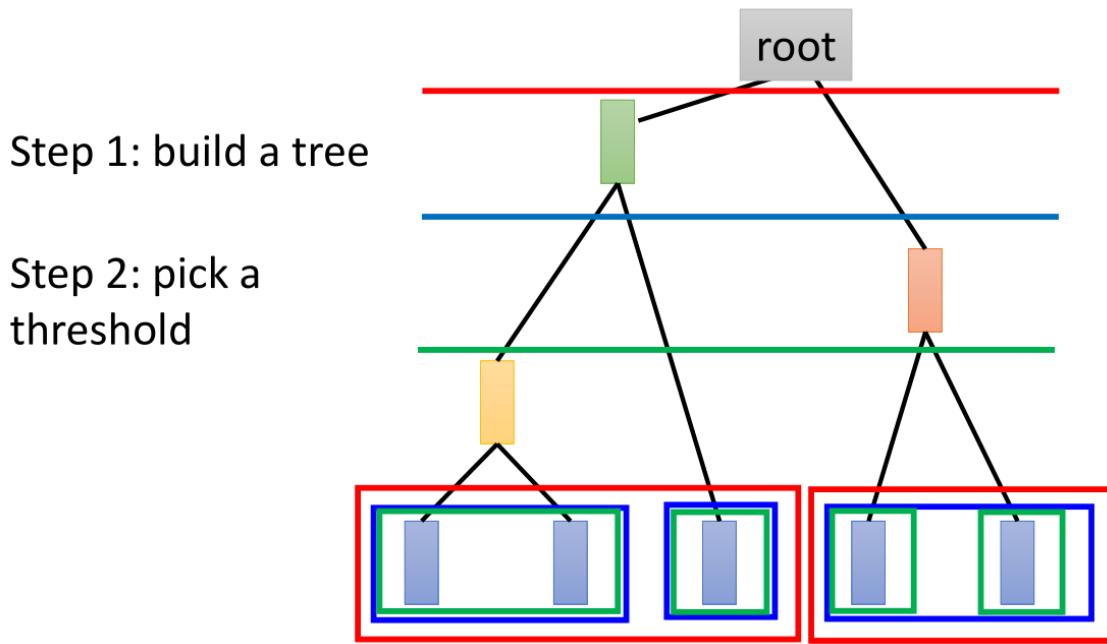
整个过程类似建立Huffman Tree, 只不过Huffman是依据词频, 而HAC是依据相似度建树

- 对5个样本点两两计算相似度, 挑出最相似的一对, 比如样本点1和2
- 将样本点1和2进行merge (可以对两个vector取平均), 生成代表这两个样本点的新结点
- 此时只剩下4个结点, 再重复上述步骤进行样本点的合并, 直到只剩下1个root结点

- pick a threshold:

选取阈值, 形象来说就是在构造好的tree上横着切一刀, 相连的叶结点属于同一个cluster

下图中，不同颜色的横线和叶结点上不同颜色的方框对应着切法与cluster的分法



HAC和K-means最大的区别在于如何决定cluster的数量，在K-means里，K的值是要你直接决定的；而在HAC里，你并不需要直接决定分多少cluster，而是去决定这一刀切在树的哪里

Dimension Reduction

clustering的缺点是**以偏概全**，它强迫每个object都要属于某个cluster

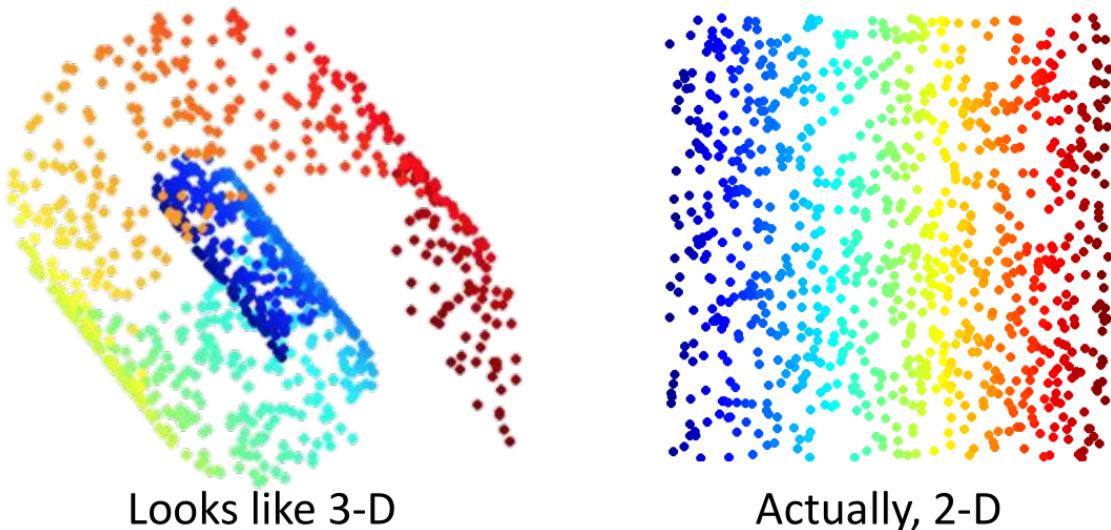
但实际上某个object可能拥有多种属性，或者多个cluster的特征，如果把它强制归为某个cluster，就会失去很多信息；我们应该用一个vector来描述该object，这个vector的每一维都代表object的某种属性，这种做法就叫做Distributed Representation，或者说，Dimension Reduction

如果原先的object是high dimension的，比如image，那现在用它的属性来描述自身，就可以使之从高维空间转变为低维空间，这就是所谓的**降维(Dimension Reduction)**

Why Dimension Reduction Help?

接下来我们从另一个角度来看为什么Dimension Reduction可能是有用的

假设data为下图左侧中的3D螺旋式分布，你会发现用3D的空间来描述这些data其实是很浪费的，因为我们完全可以把这个卷摊平，此时只需要用2D的空间就可以描述这个3D的信息



如果以MNIST(手写数字集)为例，每一张image都是 28×28 dimension，但我们反过来想，大多数 28×28 dimension的vector转成image，看起来都不会像是一个数字，所以描述数字所需要的dimension可能远比 28×28 要来得少。

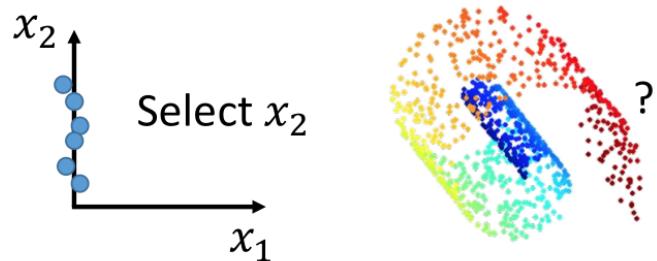
举一个极端的例子，对于只是存在角度差异的image，我们完全可以用某张image旋转的角度 θ 来描述，也就是说，我们只需要用 θ 这1个dimension就可以描述原先 28×28 dims的图像

How to do Dimension Reduction?

在Dimension Reduction里，我们要找一个function，这个function的input是原始的x，output是经过降维之后的z

最简单的方法是**Feature Selection**，即直接从原有的dimension里拿掉一些直观上就对结果没有影响的dimension，就做到了降维，比如下图中从 x_1, x_2 两个维度中直接拿掉 x_1 ；但这个方法不总是有用，因为很多情况下任何一个dimension其实都不能被拿掉。

Feature selection



Principle component analysis (PCA)

[Bishop, Chapter 12]

$$z = Wx$$

另一个常见的方法叫做**PCA**(Principle Component Analysis)

PCA认为降维就是一个很简单的linear function，它的input x和output z之间是linear transform，即 $z = Wx$ ，PCA要做的，就是根据一大堆的x把W给找出来(z未知)

PCA

为了简化问题，这里我们假设 z 是1维的vector，也就是把 x 投影到一维空间

注： w_i 为行向量， x_i 为列向量，下文中 $w_i \cdot x_i$ 表示的是矢量内积，而 $(w^i)^T x_i$ 表示的是矩阵相乘

$z_1 = w_1 \cdot x_1$ ，为Scalar，其中 w_1 表示 W 的第一个row vector，假设 w_1 的长度为1，即 $\|w_1\|_2 = 1$ ，那 w_1 跟 x_1 做内积得到的 z_1 意味着： x_1 是高维空间中的一个点， w_1 是高维空间中的一个vector，此时 z_1 就是 x_1 在 w_1 上的投影，投影的值就是 w_1 和 x 的inner product

$$(w_1)(x_1 \ x_2 \ \cdots \ x_N) = (z_1 \ z_2 \ \cdots \ z_N)$$

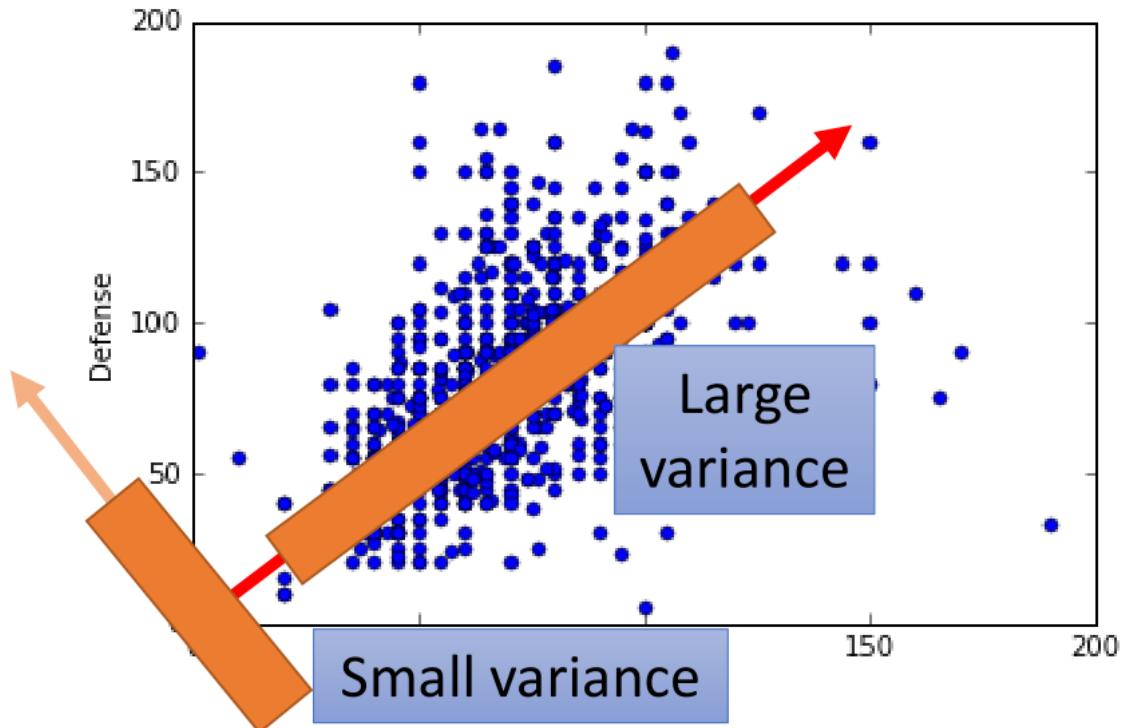
那我们到底要找什么样的 w_1 呢？

假设我们现在已有的宝可梦样本点分布如下，横坐标代表宝可梦的攻击力，纵坐标代表防御力，我们的任务是把这个二维分布投影到一维空间上

我们希望选这样一个 w_1 ，它使得 x 经过投影之后得到的 z 分布越大越好，也就是说，经过这个投影后，不同样本点之间的区别，应该仍然是可以被看得出来的，即：

- 我们希望找一个projection的方向，它可以让projection后的variance越大越好
- 我们不希望projection使这些data point通通挤在一起，导致点与点之间的奇异度消失
- 其中，variance的计算公式： $Var(z) = \frac{1}{N} \sum_z (z - \bar{z})^2$, $\|w_1\|_2 = 1$, \bar{z} 是 z 的平均值

下图给出了所有样本点在两个不同的方向上投影之后的variance比较情况



当然我们不可能只投影到一维空间，我们还可以投影到更高维的空间

对 $z = Wx$ 来说：

- $z_1^1 = w_1 \cdot x_1$ (注意是内积)，表示 x_1 在 w_1 方向上的投影，为Scalar
- $z_1^2 = w_2 \cdot x_1$ ，表示 x_1 在 w_2 方向上的投影
- ...

z_1^1, z_1^2, \dots 串起来就得到列向量 z_1 ，而 w_1, w_2, \dots 分别是 W 的第1,2,...个row，需要注意的是，这里的 w_i 必须相互正交，此时 W 是正交矩阵(orthogonal matrix)，如果不加以约束，则找到的 w_1, w_2, \dots 实际上是相同的值

两个矩阵相乘的意义是将右边矩阵中的每一列向量变换到左边矩阵中每一行向量为基所表示的空间中去。

如果我们有M个N维向量，想将其变换为由R个N维向量表示的新空间中，那么首先将R个基按行组成矩阵A，然后将向量按列组成矩阵B，那么**两矩阵的乘积AB就是变换结果**，其中AB的第m列为A中第m列变换后的结果。我们可以将一N维数据变换到更低维度的空间中去，变换后的维度取决于基的数量。因此这种矩阵相乘的表示也可以表示降维变换。

PCA的数学原理

$$\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_R \end{pmatrix} \begin{pmatrix} x_1 & x_2 & \cdots & x_N \end{pmatrix} = \begin{pmatrix} w_1 \cdot x_1 & w_1 \cdot x_2 & \cdots & w_1 \cdot x_N \\ w_2 \cdot x_1 & w_2 \cdot x_2 & \cdots & w_2 \cdot x_N \\ \vdots & \vdots & \ddots & \vdots \\ w_R \cdot x_1 & w_R \cdot x_2 & \cdots & w_R \cdot x_N \end{pmatrix} = \begin{pmatrix} z_1 & z_2 & \cdots & z_N \end{pmatrix}$$

PCA

$$z = Wx$$

Reduce to 1-D:

$$z_1 = w^1 \cdot x$$

$$z_2 = w^2 \cdot x$$

$$W = \begin{bmatrix} (w^1)^T \\ (w^2)^T \\ \vdots \end{bmatrix}$$

Orthogonal matrix

Project all the data points x onto w^1 , and obtain a set of z_1

We want the variance of z_1 as large as possible

$$Var(z_1) = \frac{1}{N} \sum_{z_1} (z_1 - \bar{z}_1)^2 \quad \|w^1\|_2 = 1$$

We want the variance of z_2 as large as possible

$$Var(z_2) = \frac{1}{N} \sum_{z_2} (z_2 - \bar{z}_2)^2 \quad \|w^2\|_2 = 1$$

$$w^1 \cdot w^2 = 0$$

Lagrange multiplier

求解PCA，实际上已经有现成的函数可以调用，此外你也可以把PCA描述成neural network，然后用gradient descent的方法来求解，这里主要介绍用拉格朗日乘数法(Lagrange multiplier)求解PCA的数学推导过程

$$z_1 = w^1 \cdot x$$

PCA

$$\bar{z}_1 = \frac{1}{N} \sum z_1 = \frac{1}{N} \sum w^1 \cdot x = w^1 \cdot \frac{1}{N} \sum x = w^1 \cdot \bar{x}$$

$$\begin{aligned}
 Var(z_1) &= \frac{1}{N} \sum_{z_1} (z_1 - \bar{z}_1)^2 & (a \cdot b)^2 &= (a^T b)^2 = a^T b a^T b \\
 &= \frac{1}{N} \sum_x (w^1 \cdot x - w^1 \cdot \bar{x})^2 & &= a^T b (a^T b)^T = a^T b b^T a \\
 &= \frac{1}{N} \sum (w^1 \cdot (x - \bar{x}))^2 & & \\
 &= \frac{1}{N} \sum (w^1)^T (x - \bar{x})(x - \bar{x})^T w^1 & & \\
 &= (w^1)^T \left[\frac{1}{N} \sum (x - \bar{x})(x - \bar{x})^T \right] w^1 & & \\
 &= (w^1)^T Cov(x) w^1 & S = Cov(x)
 \end{aligned}$$

Find w^1 maximizing

$$(w^1)^T S w^1$$

$$\|w^1\|_2 = (w^1)^T w^1 = 1$$

Find w^1 maximizing $(w^1)^T S w^1$ $(w^1)^T w^1 = 1$

$S = Cov(x)$ Symmetric Positive-semidefinite
(non-negative eigenvalues)

Using Lagrange multiplier [Bishop, Appendix E]

$$g(w^1) = (w^1)^T S w^1 - \alpha((w^1)^T w^1 - 1)$$

$$\left. \begin{array}{l} \partial g(w^1)/\partial w_1^1 = 0 \\ \partial g(w^1)/\partial w_2^1 = 0 \\ \vdots \end{array} \right\} \quad \begin{array}{l} S w^1 - \alpha w^1 = 0 \\ S w^1 = \alpha w^1 \quad w^1 : \text{eigenvector} \\ (w^1)^T S w^1 = \alpha (w^1)^T w^1 \\ = \alpha \quad \text{Choose the maximum one} \end{array}$$

w^1 is the eigenvector of the covariance matrix S

Corresponding to the largest eigenvalue λ_1

Calculate w^1

注：根据PPT， w^1 为列向量， z_1 和 x 为多个列向量

- 首先计算出 \bar{z}_1 ：

$$z_1 = w^1 \cdot x$$

$$\bar{z}_1 = \frac{1}{N} \sum z_1 = \frac{1}{N} \sum w^1 \cdot x = w^1 \cdot \frac{1}{N} \sum x = w^1 \cdot \bar{x}$$

- 然后计算maximize的对象 $Var(z_1)$:

其中 $Cov(x) = \frac{1}{N} \sum (x - \bar{x})(x - \bar{x})^T$

$$\begin{aligned} Var(z_1) &= \frac{1}{N} \sum_{z_1} (z_1 - \bar{z}_1)^2 \\ &= \frac{1}{N} \sum_x (w^1 \cdot x - w^1 \cdot \bar{x})^2 \\ &= \frac{1}{N} \sum (w^1 \cdot (x - \bar{x}))^2 \\ &= \frac{1}{N} \sum (w^1)^T (x - \bar{x})(x - \bar{x})^T w^1 \\ &= (w^1)^T \frac{1}{N} \sum (x - \bar{x})(x - \bar{x})^T w^1 \\ &= (w^1)^T Cov(x) w^1 \end{aligned}$$

- 当然这里想要求 $Var(z_1) = (w^1)^T Cov(x) w^1$ 的最大值，还要加上 $\|w^1\|_2 = (w^1)^T w^1 = 1$ 的约束条件，否则 w^1 可以取无穷大
- 令 $S = Cov(x)$ ，它是：

- 对称的(symmetric)
- 半正定的(positive-semidefinite)
- 所有特征值(eigenvalues)非负的(non-negative)
- 目标：maximize $(w^1)^T S w^1$ ，条件： $(w^1)^T w^1 = 1$
- 使用拉格朗日乘数法，利用目标和约束条件构造函数：

$$g(w^1) = (w^1)^T S w^1 - \alpha((w^1)^T w^1 - 1)$$

- 对 w^1 这个vector里的每一个element做偏微分：

$$\begin{aligned} \partial g(w^1) / \partial w_1^1 &= 0 \\ \partial g(w^1) / \partial w_2^1 &= 0 \\ \partial g(w^1) / \partial w_3^1 &= 0 \\ &\dots \end{aligned}$$

- 整理上述推导式，可以得到：

$$S w^1 = \alpha w^1$$

其中， w^1 是 S 的特征向量(eigenvector)

- 注意到满足 $(w^1)^T w^1 = 1$ 的特征向量 w^1 有很多，我们要找的是可以maximize $(w^1)^T S w^1$ 的那一个，于是利用上一个式子：

$$(w^1)^T S w^1 = (w^1)^T \alpha w^1 = \alpha (w^1)^T w^1 = \alpha$$

- 此时maximize $(w^1)^T S w^1$ 就变成了maximize α ，也就是当 S 的特征值 α 最大时对应的那个特征向量 w^1 就是我们要找的目标
- 结论： w^1 是 $S = Cov(x)$ 这个matrix中的特征向量，对应最大的特征值 λ_1

Calculate w^2

在推导 w^2 时，相较于 w^1 ，多了一个限制条件： w^2 必须与 w^1 正交(orthogonal)

目标：maximize $(w^2)^T S w^2$ ，条件： $(w^2)^T w^2 = 1, (w^2)^T w^1 = 0$

- 同样是用拉格朗日乘数法求解，先写一个关于 w^2 的function，包含要maximize的对象，以及两个约束条件

$$g(w^2) = (w^2)^T S w^2 - \alpha((w^2)^T w^2 - 1) - \beta((w^2)^T w^1 - 0)$$

- 对 w^2 的每个element做偏微分：

$$\partial g(w^2)/\partial w_1^2 = 0$$

$$\partial g(w^2)/\partial w_2^2 = 0$$

$$\partial g(w^2)/\partial w_3^2 = 0$$

...

- 整理后得到：

$$Sw^2 - \alpha w^2 - \beta w^1 = 0$$

- 上式两侧同乘 $(w^1)^T$, 得到：

$$(w^1)^T Sw^2 - \alpha(w^1)^T w^2 - \beta(w^1)^T w^1 = 0$$

- 其中 $\alpha(w^1)^T w^2 = 0, \beta(w^1)^T w^1 = \beta$,

而由于 $(w^1)^T Sw^2$ 是vector×matrix×vector=scalar, 因此在外面套一个transpose不会改变其值, 因此该部分可以转化为：

注：S是symmetric的，因此 $S^T = S$

$$\begin{aligned} (w^1)^T Sw^2 &= ((w^1)^T Sw^2)^T \\ &= (w^2)^T S^T w^1 \\ &= (w^2)^T Sw^1 \end{aligned}$$

我们已经知道 w^1 满足 $Sw^1 = \lambda_1 w^1$, 代入上式:

$$\begin{aligned} (w^1)^T Sw^2 &= (w^2)^T Sw^1 \\ &= \lambda_1 (w^2)^T w^1 \\ &= 0 \end{aligned}$$

- 因此有 $(w^1)^T Sw^2 = 0, \alpha(w^1)^T w^2 = 0, \beta(w^1)^T w^1 = \beta$, 又根据

$$(w^1)^T Sw^2 - \alpha(w^1)^T w^2 - \beta(w^1)^T w^1 = 0$$

可以推得 $\beta = 0$

- 此时 $Sw^2 - \alpha w^2 - \beta w^1 = 0$ 就转变成了 $Sw^2 - \alpha w^2 = 0$, 即

$$Sw^2 = \alpha w^2$$

- 由于S是symmetric的, 因此在不与 w_1 冲突的情况下, 这里 α 选取第二大的特征值 λ_2 时, 可以使 $(w^2)^T Sw^2$ 最大
- 结论: w^2 也是 $S = Cov(x)$ 这个matrix中的特征向量, 对应第二大的特征值 λ_2

Decorrelation

神奇之处在于 $Cov(z) = D$, 即z的covariance是一个diagonal matrix

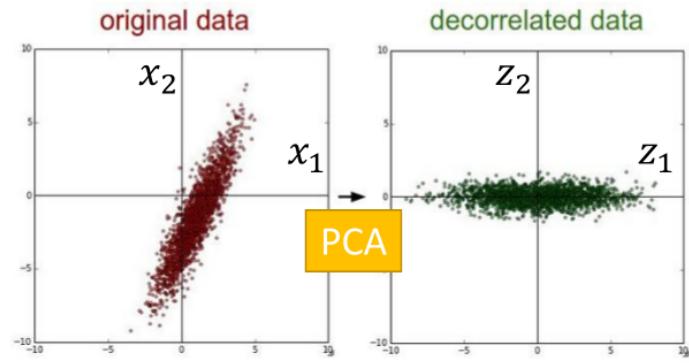
如果你把原来的input data通过PCA之后再给其他model使用, 其它的model就可以假设现在的input data它的dimension之间没有decorrelation。所以它就可以用简单的model处理你的input data, 参数量大大降低, 相同的数据量可以得到更好的训练结果, 从而可以避免overfitting的发生

PCA - decorrelation

$$z = Wx$$

$$\text{Cov}(z) = D$$

Diagonal matrix



$$\text{Cov}(z) = \frac{1}{N} \sum (z - \bar{z})(z - \bar{z})^T = WSW^T \quad S = \text{Cov}(x)$$

$$= WS [w^1 \quad \dots \quad w^K] = W [S w^1 \quad \dots \quad S w^K]$$

$$= W [\lambda_1 w^1 \quad \dots \quad \lambda_K w^K] = [\lambda_1 W w^1 \quad \dots \quad \lambda_K W w^K]$$

$$= [\lambda_1 e_1 \quad \dots \quad \lambda_K e_K] = D \quad \text{Diagonal matrix}$$

Reconstruction Component

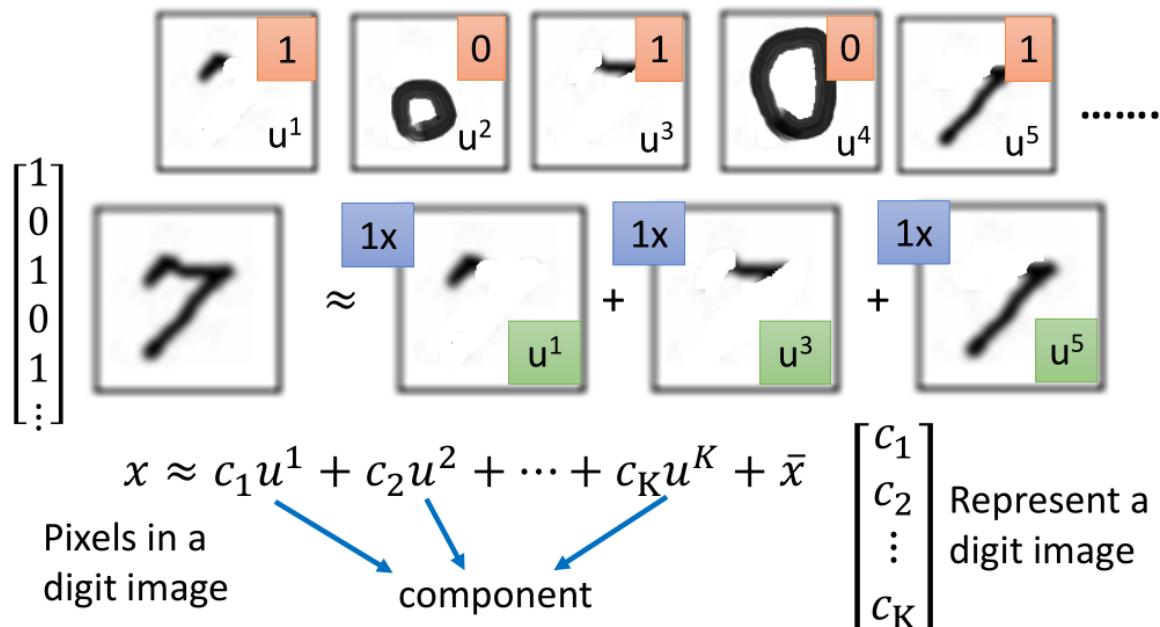
假设我们现在考虑的是手写数字识别，这些数字是由一些类似于笔画的basic component组成的，本质上就是一个vector，记做 u_1, u_2, u_3, \dots ，以MNIST为例，不同的笔画都是一个 28×28 维的vector，把某几个vector加起来，就组成了一个 28×28 维的digit

写成表达式就是： $x \approx c_1 u^1 + c_2 u^2 + \dots + c_k u^k + \bar{x}$

其中 x 代表某张digit image中的pixel，它等于 k 个component的加权和 $\sum c_i u^i$ 加上所有image的平均值 \bar{x}

比如 7 就是 $x = u^1 + u^3 + u^5$ ，我们可以用 $[c_1 \ c_2 \ c_3 \dots c_k]^T$ 来表示一张digit image，如果component的数目 k 远比pixel的数目要小，那这个描述就是比较有效的

Basic Component:



实际上目前我们并不知道 $u^1 \sim u^K$ 具体的值，因此我们要找这样 k 个vector，使得 $x - \bar{x}$ 与 \hat{x} 越接近越好：

$$x - \bar{x} \approx c_1 u^1 + c_2 u^2 + \dots + c_k u^k = \hat{x}$$

而用未知component来描述的这部分内容，叫做Reconstruction error，即 $\|(x - \bar{x}) - \hat{x}\|$

接下来我们就要去找k个vector u^i 去minimize这个error：

$$L = \min_{u^1, \dots, u^k} \sum \|(x - \bar{x}) - (\sum_{i=1}^k c_i u^i)\|_2$$

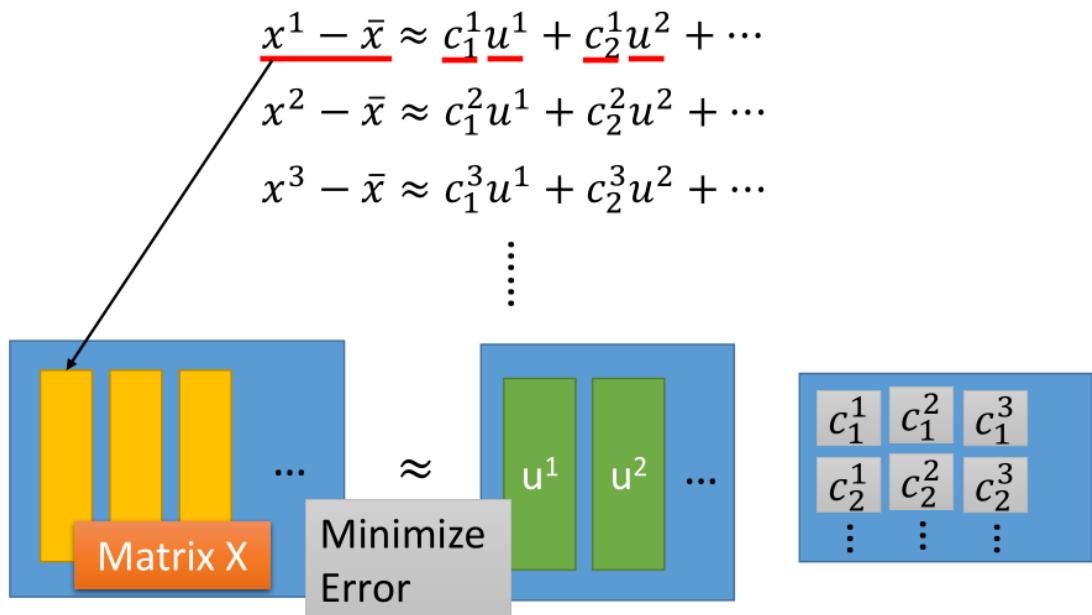
回顾PCA， $z = Wx$

实际上我们通过PCA最终解得的 $\{w^1, w^2, \dots, w^k\}$ 就是使reconstruction error最小化的 $\{u^1, u^2, \dots, u^k\}$ ，简单证明如下：

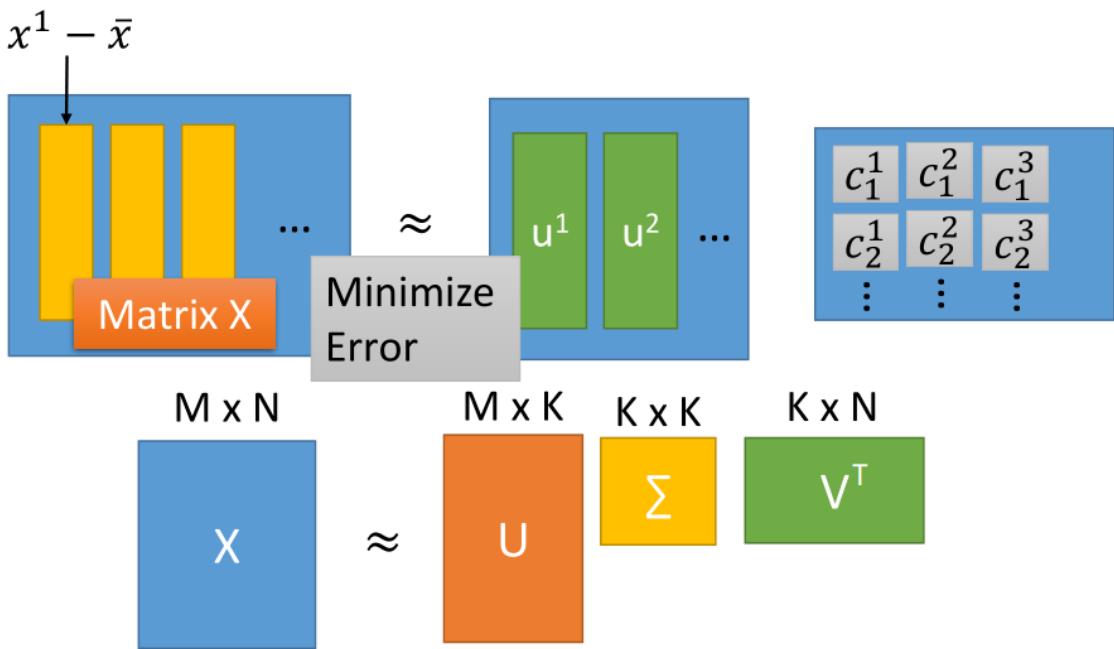
- 我们将所有的 $x^i - \bar{x} \approx c_1^i u^1 + c_2^i u^2 + \dots$ 都用下图中的矩阵相乘来表示，我们的目标是使等号两侧矩阵之间的差距越小越好，把 u^1, u^2, \dots 看作一行。

$$x - \bar{x} \approx c_1 u^1 + c_2 u^2 + \dots + c_K u^K = \hat{x}$$

Reconstruction error: $\|(x - \bar{x}) - \hat{x}\|_2$ Find $\{u^1, \dots, u^K\}$ minimizing the error



- 可以使用SVD将每个matrix $X_{m \times n}$ 都拆成matrix $U_{m \times k}, \Sigma_{k \times k}, V_{k \times n}$ 的乘积，其中k为component的数目
- 使用SVD拆解后的三个矩阵相乘的结果，是跟等号左边的矩阵X最接近的，此时U就对应着 u^i 那部分的矩阵， $\Sigma \cdot V$ 就对应着 c_k^i 那部分的矩阵
- 根据SVD的结论，组成矩阵U的k个列向量(标准正交向量, orthonormal vector)就是 XX^T 最大的k个特征值(eigenvalue)所对应的特征向量(eigenvector)，而 $\frac{1}{N}XX^T$ 实际上就是x的covariance matrix，因此U就是PCA的k个解
- 因此我们可以发现，通过PCA找出来的Dimension Reduction的transform w ，实际上就是把X拆解成能够最小化Reconstruction error的component，通过PCA所得到的 w^i 就是component u^i ，而Dimension Reduction的结果就是参数 c_i
- 简单来说就是，用PCA对x进行降维的过程中，我们要找的投影方式 w^i 就相当于恰当的组件 u^i ，投影结果 z^i 就相当于这些组件各自所占的比例 c_i
- PCA求解关键在于求解协方差矩阵 $\frac{1}{N}XX^T$ 的特征值分解，SVD关键在于 XX^T 的特征值分解。



K columns of U : a set of orthonormal eigen vectors corresponding to the K largest eigenvalues of XX^\top

This is the solution of PCA

- 下面的式子简单演示了将一个样本点 x 划分为 k 个组件的过程，其中 $[c_1 \ c_2 \ \dots \ c_k]^T$ 是每个组件的比例；把 x 划分为 k 个组件即从 n 维投影到 k 维空间， $[c_1 \ c_2 \ \dots \ c_k]^T$ 也是投影结果

注： x 和 u_i 均为 n 维列向量

$$x = [u_1 \ u_2 \ \dots \ u_k] \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} u_1^1 \ u_2^1 \ \dots \ u_k^1 \\ u_1^2 \ u_2^2 \ \dots \ u_k^2 \\ \vdots \\ u_1^n \ u_2^n \ \dots \ u_k^n \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix}$$

$$\begin{bmatrix} (u_1)^T \\ (u_2)^T \\ \vdots \\ (u_k)^T \end{bmatrix} x = Iz \quad (z = Wx, \text{ 证明 SVD 得到的 } u \text{ 就是 PCA 求到的 } w, \text{ 对 } x \text{ 进行 SVD 即可得到第一个式子})$$

Neural Network

现在我们已经知道，用PCA找出来的 $\{w^1, w^2, \dots, w^k\}$ 就是 k 个component $\{u^1, u^2, \dots, u^k\}$

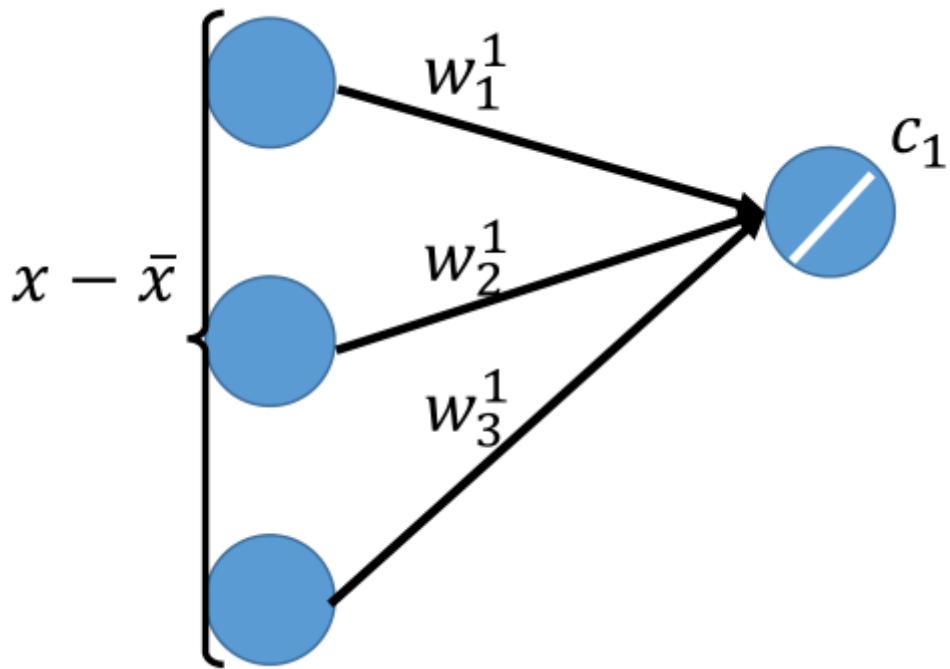
而 $\hat{x} = \sum_{k=1}^K c_k w^k$ ，我们要使 \hat{x} 与 $x - \bar{x}$ 之间的差距越小越好，我们已经根据SVD找到了 w^k 的值，而对每个不同的样本点，都会有一组不同的 c_k 值

在PCA中我们已经证得， $\{w^1, w^2, \dots, w^k\}$ 这 k 个vector是标准正交化的(orthonormal)，因此：

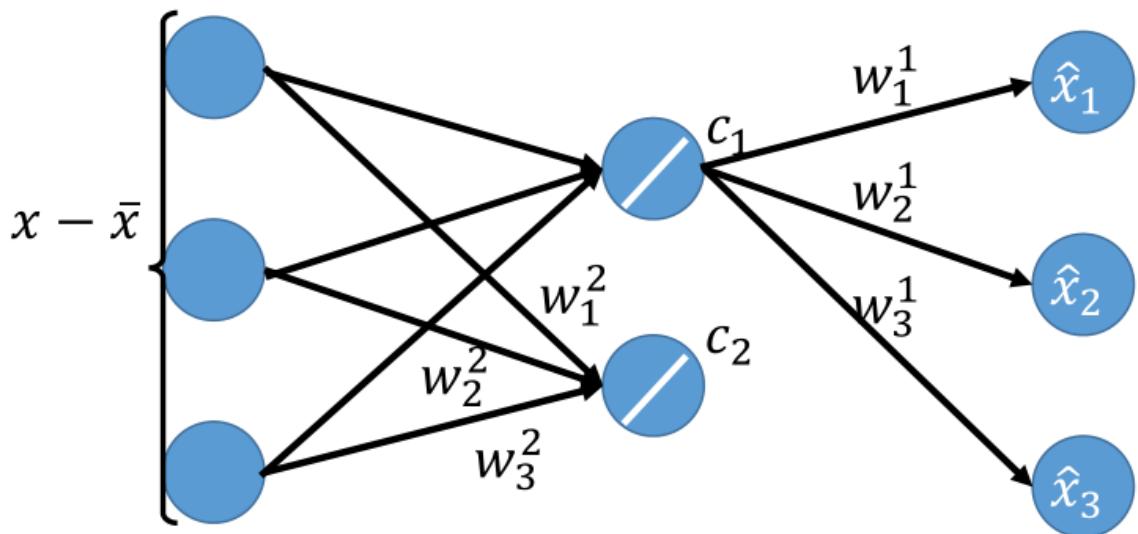
$$c_k = (x - \bar{x}) \cdot w^k \quad (\text{内积})$$

这个时候我们就可以使用神经网络来表示整个过程，假设 x 是3维向量，要投影到 $k=2$ 维的component上：

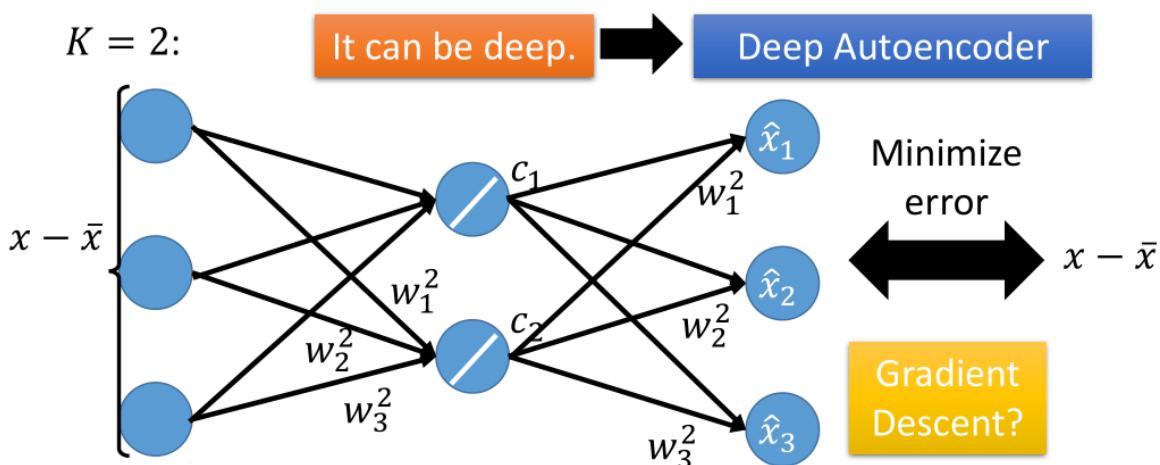
- 对 $x - \bar{x}$ 与 w^k 做inner product的过程中， $x - \bar{x}$ 在3维空间上的坐标就相当于neuron的input，而 w_1^1, w_2^1, w_3^1 则是neuron的weight，表示在 w^1 这个维度上投影的参数，而 c_1 则是这个neuron的output，表示在 w^1 这个维度上投影的坐标值；对 w^2 也同理



- 得到 c_1 之后，再让它乘上 w^1 ，得到 \hat{x} 的一部分



- 对 c_2 进行同样的操作，乘上 w^2 ，贡献 \hat{x} 的剩余部分，此时我们已经完整计算出 \hat{x} 三个分量的值



- 此时，PCA就被表示成了只含一层hidden layer的神经网络，且这个hidden layer是线性的激活函数，训练目标是让这个NN的input $x - \bar{x}$ 与output \hat{x} 越接近越好，这件事就叫做**Autoencoder**
- PCA looks like a neural network with one hidden layer (linear activation function)
- 注意，通过PCA求解出的 w^i 与直接对上述的神经网络做梯度下降所解得的 w^i 是会不一样的，因为PCA解出的 w^i 是相互垂直的(orthonormal)，而用NN的方式得到的解无法保证 w^i 相互垂直，NN无法做到Reconstruction error比PCA小，因此：
 - 在linear的情况下，直接用PCA找 W 远比用神经网络的方式更快速方便
 - 用NN的好处是，它可以使用不止一层hidden layer，它可以做**deep autoencoder**

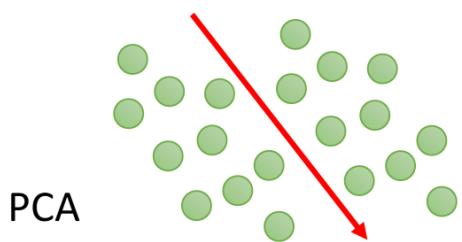
Weakness

PCA有很明显的弱点：

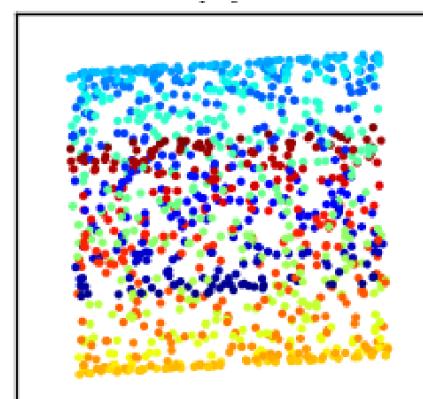
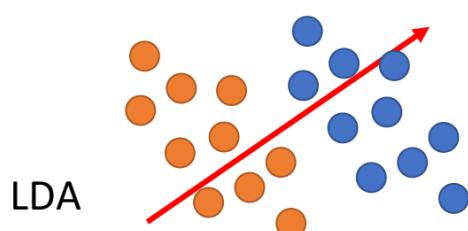
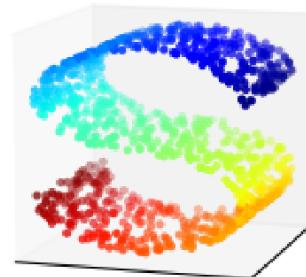
- 它是**unsupervised**的，如果我们要将下图绿色的点投影到一维空间上，PCA给出的从左上到右下的划分很有可能使原本属于蓝色和橙色的两个class的点被merge在一起；
LDA是考虑了labeled data之后进行降维的一种方式，属于**supervised**
- 它是**linear**的，对于下图中的彩色曲面，我们期望把它平铺拉直进行降维，但这是一个non-linear的投影转换，PCA无法做到这件事情，PCA只能做到把这个曲面打扁压在平面上，类似下图，而无法把它拉开
对类似曲面空间的降维投影，需要用到non-linear transformation (non-linear dimension reduction)

Weakness of PCA

• Unsupervised



• Linear



http://www.astroml.org/book_figures/chapter7/fig_S_manifold_PCA.html

Application

Pokémon

用PCA来分析宝可梦的数据

假设总共有800只宝可梦，每只都是一个六维度的样本点，即vector={HP, Atk, Def, Sp Atk, Sp Def, Speed}，接下来的问题是，我们要投影到多少维的空间上？要多少个component就好像是neural network要几个layer，每个layer要有几个neural一样，所以这是你要自己决定的。

如果做可视化分析的话，投影到二维或三维平面可以方便人眼观察。

一个常见的方法是这样的：我们去计算每一个principle components的 λ (每一个principle component就是一个eigenvector，一个eigenvector对应到一个eigenvalue λ)。这个eigenvalue代表principle component去做dimension reduction的时候，在principle component的那个dimension上，它的variance有多大(variance就是 λ)。

今天这个宝可梦的数据总共有6维，所以covariance matrix是有6维。你可以找出6个eigenvector，找出6个eigenvalue。现在我们来计算一下每个eigenvalue的ratio(每个eigenvalue除以6个eigenvalue的总和)，得到的结果如图。

How many principle components? $\frac{\lambda_i}{\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 + \lambda_5 + \lambda_6}$

	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6
ratio	0.45	0.18	0.13	0.12	0.07	0.04

Using 4 components is good enough

可以从这个结果看出来：第五个和第六个principle component的作用是比较小的，你用这两个dimension来做projection的时候project出来的variance是很小的，代表说：现在宝可梦的特性在第五个和第六个principle component上是没有太多的information。所以我们今天要分析宝可梦data的话，感觉只需要前面四个principle component就好了。

我们实际来分析一下，做PCA以后得到四个principle component就是这个样子，每一个principle component就是一个vector，每一个宝可梦是用6维的vector来描述。

如果你要产生一只宝可梦的时候，每一个宝可梦都是由这四个vector做linear combination，

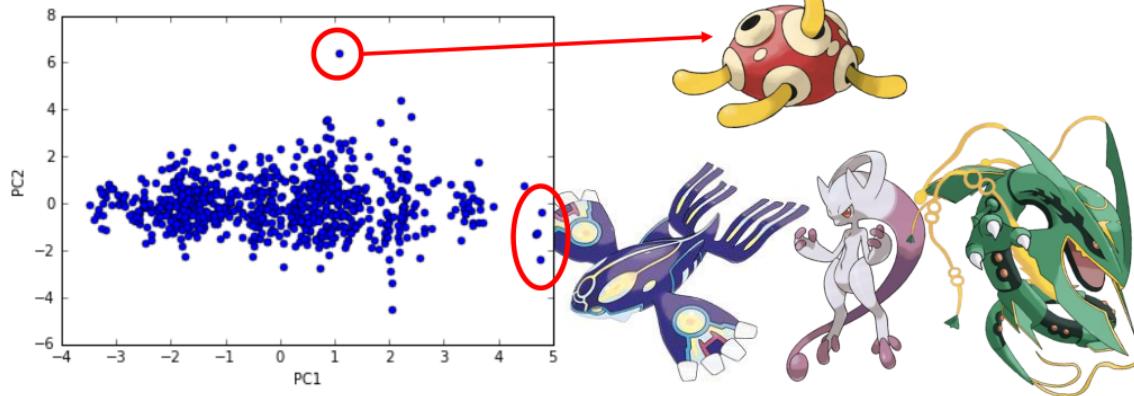
新的维度本质上就是旧的维度的加权矢量和，下图给出了前4个维度的加权情况，从PC1到PC4这4个principle component都是6维度加权的vector，它们都可以被认为是某种组件，大多数的宝可梦都可以由这4种组件拼接而成，也就是用这4个6维的vector做linear combination的结果

我们来看每一个principle component做的事情是什么：

- 对第一个vector PC1来说，每个值都是正的，在选第一个principle component的时候，你给它的weight比较大，那这个宝可梦的六维都是强的，所以这第一个principle component就代表了这一只宝可梦的强度。
- 对第二个vector PC2来说，防御力Def很大而速度Speed很小，你给第二个principle component一个weight的时候，你会增加那只宝可梦的防御力但是会减低它的速度。
- 如果将宝可梦仅仅投影到PC1和PC2这两个维度上，则降维后的二维可视化图像如下图所示：
从该图中也可以得到一些信息：
 - 在PC2维度上特别大的那个样本点刚好对应着海龟，确实是防御力且速度慢的宝可梦

- 在PC1维度上特别大的那三个样本点则对应着盖欧卡、超梦等综合实力很强的宝可梦

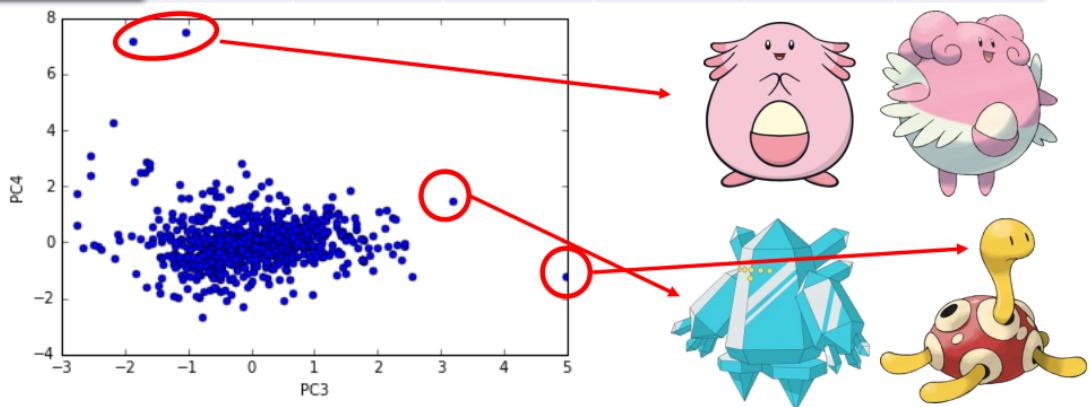
	HP	Atk	Def	Sp Atk	Sp Def	Speed	
PC1	0.4	0.4	0.4	0.5	0.4	0.3	強度
PC2	0.1	0.0	0.6	-0.3	0.2	-0.7	
PC3	-0.5	-0.6	0.1	0.3	0.6	-0.6	防禦(犧牲速度)
PC4	0.7	-0.4	-0.4	0.1	-0.6	-0.3	



- 对第三个principle component来说，sp Def很大而HP和Atk很小，这个组件是用生命力和攻击力来换取特殊防御力。
- 对第四个vector PC4来说，HP很大而Atk和Def很小，这个组件是用攻击力和防御力来换取生命力
- 同样将宝可梦只投影到PC3和PC4这两个维度上，则降维后得到的可视化图像如下图所示：
该图同样可以告诉我们一些信息：

- 在PC3维度上特别大的样本点依旧是普普，第二名是冰柱机器人，它们的特殊防御力都比较高
- 在PC4维度上特别大的样本点则是吉利蛋和幸福蛋，它们的生命力比较强

	HP	Atk	Def	Sp Atk	Sp Def	Speed	
PC1	0.4	0.4	0.4	0.5	0.4	0.3	
PC2	0.1	0.0	0.6	-0.3	0.2	-0.7	
PC3	-0.5	-0.6	0.1	0.3	0.6	-0.6	特殊防禦(犧牲攻擊和生命)
生命力強	0.7	-0.4	-0.4	0.1	0.2	-0.3	



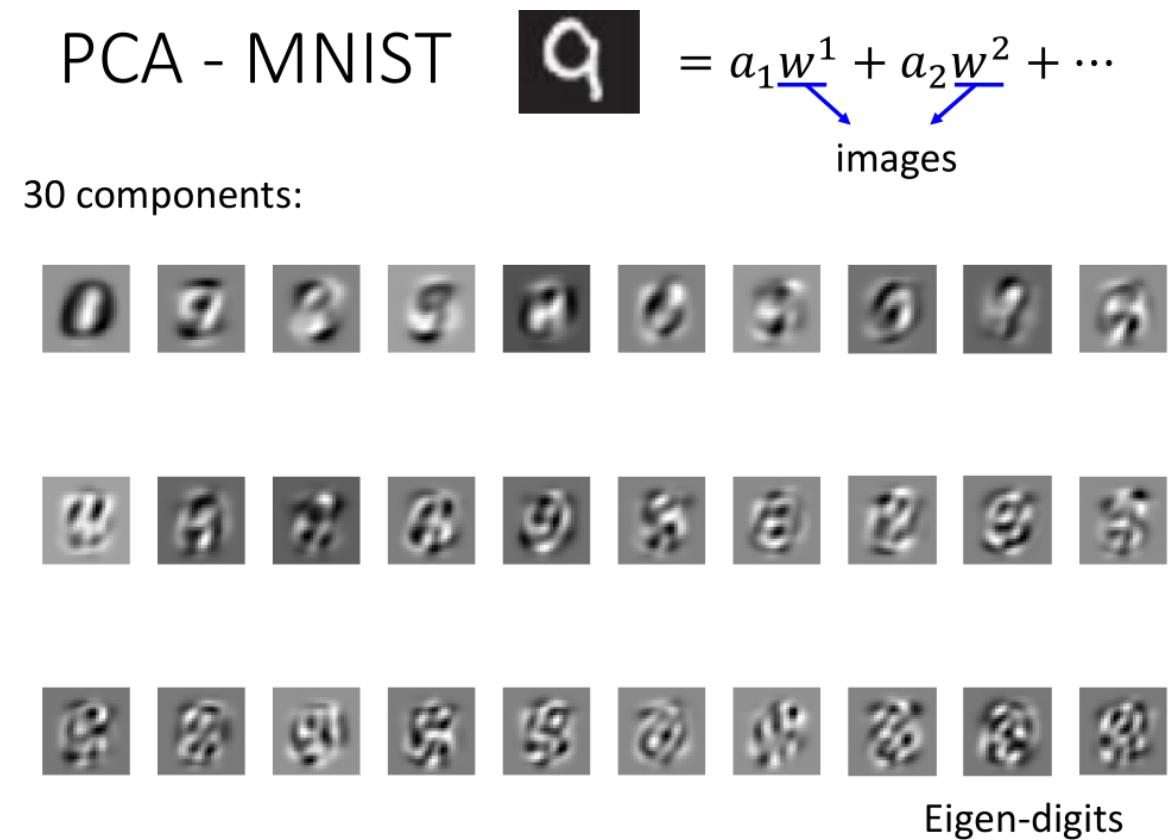
MNIST

我们拿它来做手写数字辨识的话，我们可以把每一张数字都拆成component乘以weight，加上另外一个component乘以weight，每一个component是一张image(28* 28的vector)。

$$\text{digit image} = a_1 w^1 + a_2 w^2 + \dots$$

我们现在来画前PCA得到的前30个component的话，你得到的结果是这样子的(如图所示)，你用这些component做linear combination，你就得到所有的digit(0-9)，所以这些component就叫做Eigen digits(这些component其实都是covariance matrix的eigenvector)

注：PCA就是求 $Cov(x) = \frac{1}{N} \sum (x - \bar{x})(x - \bar{x})^T$ 的前30个最大的特征值对应的特征向量



Face

同理，通过PCA找出人脸的前30个principle component，得到的结果是这样子的。这些叫做Eigen-face。你把这些脸做linear combination以后就可以得到所有的脸。但是这边跟我们预期的有些是不一样的，因为现在我们找出来的不是component，我们找出来的每一个图都几乎是完整的脸。

PCA - Face



30 components:



<http://www.cs.unc.edu/~lazebnik/research/spring08/assignment3.html> Eigen-face

What happens to PCA

在对MNIST和Face的PCA结果展示的时候，你可能会注意到我们找到的组件好像并不算是组件，比如MNIST找到的几乎是完整的数字雏形，而Face找到的也几乎是完整的人脸雏形，但我们预期的组件不应该是类似于横折撇捺，眼睛鼻子眉毛这些吗？

如果你仔细思考了PCA的特性，就会发现得到这个结果是可能的

$$\text{image} = a_1 w^1 + a_2 w^2 + \dots$$

注意到linear combination的weight a_i 可以是正的也可以是负的，因此我们可以通过把组件进行相加或相减来获得目标图像，这会导致你找出来的component不是基础的组件，但是通过这些组件的加加减减肯定可以获得基础的组件元素

NMF

Introduction

如果你要一开始就得到类似笔画这样的基础组件，就要使用NMF(non-negative matrix factorization)，非负矩阵分解的方法

PCA可以看成对原始矩阵 X 做SVD进行矩阵分解，但并不保证分解后矩阵的正负，实际上当进行图像处理时，如果部分组件的matrix包含一些负值的话，如何处理负的像素值也会成为一个问题(可以做归一化处理，但比较麻烦)

而NMF的基本精神是，强迫使所有组件和它的加权值都必须是正的，也就是说**所有图像都必须由组件叠加得到**：

- Forcing a_1, a_2, \dots be non-negative
 - additive combination
- Forcing w_1, w_2, \dots be non-negative

- More like “parts of digits”

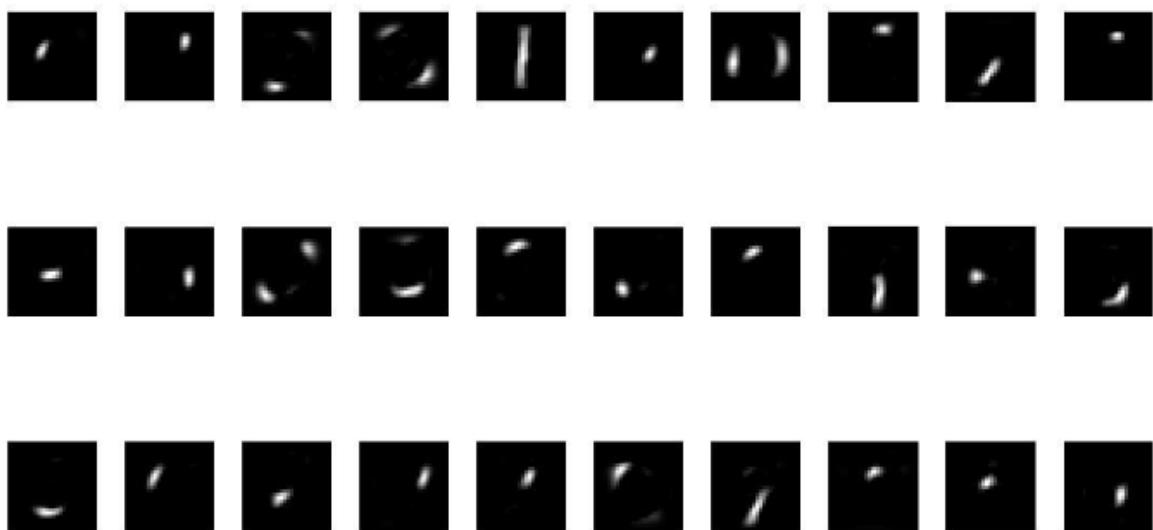
注：关于NMF的具体算法内容可参考paper(公众号回复“NMF”获取pdf)：

Daniel D. Lee and H. Sebastian Seung. "Algorithms for non-negative matrix factorization." Advances in neural information processing systems. 2001.

MNIST

在MNIST数据集上，通过NMF找到的前30个组件如下图所示，可以发现这些组件都是由基础的笔画构成：

NMF on MNIST



Face

在Face数据集上，通过NMF找到的前30个组价如下图所示，相比于PCA这里更像是脸的一部分

NMF on Face



More Related Approaches

降维的方法有很多，这里再列举一些与PCA有关的方法：

- Multidimensional Scaling (**MDS**) [Alpaydin, Chapter 6.7]

MDS不需要把每个data都表示成feature vector，只需要知道特征向量之间的distance，就可以做降维

一般教科书举的例子会说：我现在一堆城市，你不知道如何把城市描述成vector，但你知道城市跟城市之间的距离(每一笔data之间的距离)，那你就可以画在二维的平面上。

其实MDS跟PCA是有一些关系的，如果你用某些特定的distance来衡量两个data point之间的距离的话，你做MDS就等于做PCA。

其实PCA有个特性是：它保留了原来在高维空间中的距离（在高维空间的距离是远的，那么在低维空间中的距离也是远的，在高维空间的距离是近的，那么在低维空间中的距离也是近的）

- **Probabilistic PCA** [Bishop, Chapter 12.2]

PCA概率版本

- **Kernel PCA** [Bishop, Chapter 12.3]

PCA非线性版本

- Canonical Correlation Analysis (**CCA**) [Alpaydin, Chapter 6.9]

CCA常用于两种不同的data source的情况，假如说你要做语音辨识，两个source（一个是声音讯号，另一个是嘴巴的image，可以看到这个人的唇形）把这两种不同的source都做dimension reduction，那这个就是CCA。

- Independent Component Analysis (**ICA**)

ICA常用于source separation，PCA找的是正交的组件，而ICA则只需要找“独立”的组件即可

- Linear Discriminant Analysis (**LDA**) [Alpaydin, Chapter 6.8]

LDA是supervised的方式

Matrix Factorization

■

Introduction

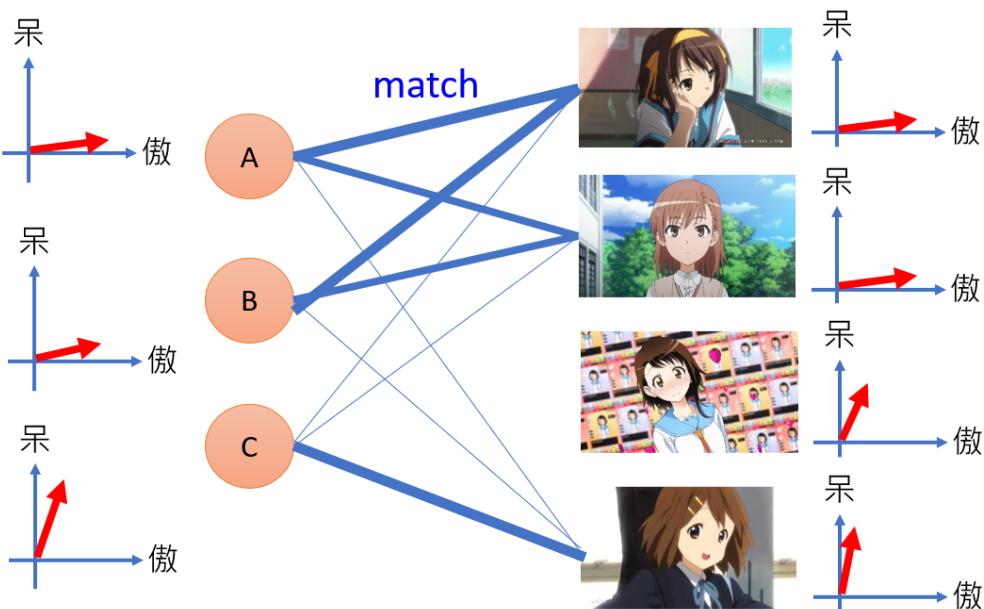
接下来介绍**矩阵分解**的思想：有时候存在两种object，它们之间会受到某种共同**潜在因素**(latent factor)的操控，如果我们找出这些潜在因素，就可以对用户的行为进行预测，这也是**推荐系统**常用的方法之一

假设我们现在去调查每个人购买的公仔数目，ABCDE代表5个人，每个人或者每个公仔实际上都是有着傲娇的属性或天然呆的属性

我们可以用vector去描述人和公仔的属性，如果某个人的属性和某个公仔的属性是match的，即他们背后的vector很像(内积值很大)，这个人就会偏向于拥有更多这种类型的公仔

Otakus v.s. No. of Figures

The factors are latent.



Matrix Expression

但是，我们没有办法直接观察某个人背后这些潜在的属性，也不会有人在意一个肥宅心里想的是什么；我们同样也没有办法直接得到动漫人物背后的属性；

我们目前有的，只是动漫人物和人之间的关系，即每个人已购买的公仔数目，我们要通过这个关系去推测出动漫人物与人背后的潜在因素(latent factor)

我们可以把每个人的属性用vector r^A 、 r^B 、 r^C 、 r^D 、 r^E 来表示，而动漫人物的属性则用vector r^1 、 r^2 、 r^3 、 r^4 来表示，购买的公仔数目可以被看成是matrix X ，对 X 来说，行数为人数，列数为动漫角色的数目

做一个假设：matrix X 里的每个element，都是属于人的vector和属于动漫角色的vector的内积

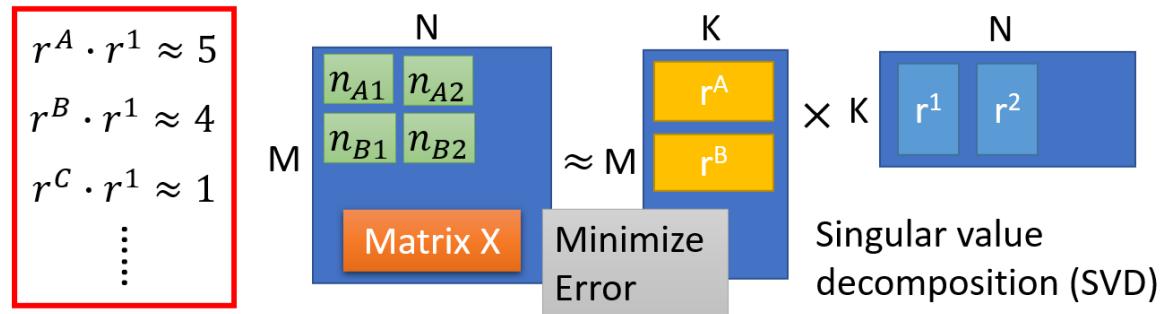
比如， $r^A \cdot r^1 \approx 5$ ，表示 r^A 和 r^1 的属性比较贴近

接下来就用下图所示的矩阵相乘的方式来表示这样的关系，其中 K 为latent factor的数量，这是未知的，需要你自己去调整选择

我们要找一组 $r^A \sim r^E$ 和 $r^1 \sim r^4$, 使得右侧两个矩阵相乘的结果与左侧的matrix X 越接近越好, 可以使用SVD的方法求解

		r^1	r^2	r^3	r^4
r^A	A	5	3	0	1
r^B	B	4	3	0	1
r^C	C	1	Matrix X		
r^D	D	1	0	4	4
r^E	E	0	1	5	4

No. of Otakus = M No. of characters = N No. of latent factor = K



Prediction

但有时候, 部分的information可能是会missing的, 这时候就难以用SVD精确描述, 但我们可以使用梯度下降的方法求解, loss function如下:

$$L = \sum_{(i,j)} (r^i \cdot r^j - n_{ij})^2$$

其中 r^i 值的是人背后的latent factor, r^j 指的是动漫角色背后的latent factor, 我们要让这两个vector的内积与实际购买该公仔的数量 n_{ij} 越接近越好, 这个方法的关键之处在于, 计算上式时, 可以跳过missing的数据, 最终通过gradient descent求得 r^i 和 r^j 的值

r^j	r^1	r^2	r^3	r^4	
r^i					
r^A	A	5 n_{A1}	3	?	1
r^B	B	4	3	?	1
r^C	C	1	1	?	5
r^D	D	1	?	4	4
r^E	E	?	1	5	4

$$r^A \cdot r^1 \approx 5$$

$$r^B \cdot r^1 \approx 4$$

$$r^C \cdot r^1 \approx 1$$

⋮

Minimizing

$$L = \sum_{(i,j)} (r^i \cdot r^j - n_{ij})^2$$

Only considering the defined value

Find r^i and r^j by gradient descent

假设latent factor的数目等于2，则人的属性 r^i 和动漫角色的属性 r^j 都是2维的vector，这里实际进行计算后，把属性中较大值标注出来，可以发现：

- 人：A、B属于同一组属性，C、D、E属于同一组属性
- 动漫角色：1、2属于同一组属性，3、4属于同一组属性
- 结合动漫角色，才可以分析出动漫角色的第一个维度是天然呆属性，第二个维度是傲娇属性
- 接下来就可以预测未知的值，只需要将人和动漫角色的vector做内积即可

这也是推荐系统的常用方法

	r^1	r^2	r^3	r^4
r^A	A 5	3	-0.4	1
r^B	B 4	3	-0.3	1
r^C	C 1	1	2.2	5
r^D	D 1	0.6	4	4
r^E	E 0.1	1	5	4

Assume the dimensions of r are all 2 (there are two factors)

A	0.2	2.1
B	0.2	1.8
C	1.3	0.7
D	1.9	0.2
E	2.2	0.0

1 (春日)	0.0	2.2
2 (炮姐)	0.1	1.5
3 (姐寺)	1.9	-0.3
4 (小唯)	2.2	0.5

More about Matrix Factorization

实际上除了人和动漫角色的属性之外，可能还存在其他因素操控购买数量这一数值，因此我们可以将式子更精确地改写为：

$$r^A \cdot r^1 + b_A + b_1 \approx 5$$

其中 b_A 这个Scalar表示A这个人本身有多喜欢买公仔， b_1 这个Scalar则表示这个动漫角色本身有多让人想要购买，这些内容是跟属性vector无关的，此时Minimizing的loss function被改写为：

$$L = \sum_{(i,j)} (r^i \cdot r^j + b_i + b_j - n_{ij})^2$$

当然你也可以加上一些regularization去对结果做约束

Paper Ref: Matrix Factorization Techniques For Recommender Systems

Latent Semantic Analysis

如果把matrix factorization的方法用在topic analysis上，就叫做LSA(Latent semantic analysis)，潜在语义分析

Matrix Factorization for Topic analysis

- Latent semantic analysis (LSA)

	Doc 1	Doc 2	Doc 3	Doc 4
投资	5	3	0	1
股票	4	0	0	1
总统	1	1	0	5
选举	1	0	0	4
立委	0	1	5	4

character→document,
otakus→word

Number in Table:

Term frequency
(weighted by inverse
document frequency)

Latent factors are topics
(财经、政治.....)

- Probability latent semantic analysis (PLSA)

• Thomas Hofmann, Probabilistic Latent Semantic Indexing, SIGIR, 1999

- latent Dirichlet allocation (LDA)

• David M. Blei, Andrew Y. Ng, Michael I. Jordan, Latent Dirichlet Allocation, Journal of Machine Learning Research, 2003

把刚才的动漫人物换成文章，把刚才的人换成词汇，table里面的值就是term frequency（词频），把这个term frequency乘上一个weight代表说这个term本身有多重要。

怎样evaluation一个term重不重要呢？常用的方式是：inverse document frequency（计算某一个词汇在整个paper有多少比率的document涵盖这个词汇，假如说，某一个词汇，每个document都有，那它的inverse document frequency就很小，代表着这个词汇的重要性是低的，假设某个词汇只有某一篇document有，那它的inverse document frequency就很大，代表这个词汇的重要性是高的。在各种文章中出现次数越多的词汇越不重要，出现次数越少则越重要。）

在这个task里面，如果你今天把这个matrix做分解的话，你就会找到每一个document背后那个latent factor，那这边的latent factor是什么呢？可能指的是topic（主题），这个topic有多少是跟财经有关的，有多少是跟政治有关的。document1跟document2有比较多的“投资，股票”这样的词汇，那document1跟document2的latent factor有比较高的可能性是比较偏向“财经”的

topic analysis的方法多如牛毛，基本的精神是差不多的(有很多各种各样的变化)。常见的是Probability latent semantic analysis (PLSA)和latent Dirichlet allocation (LDA)。注意这跟之前在machine learning讲的LDA是完全不一样的东西。

Neighbor Embedding

介绍非线性降维的一些算法，包括局部线性嵌入LLE、拉普拉斯特征映射和t分布随机邻居嵌入t-SNE，其中t-SNE特别适用于可视化的应用场景

PCA和Word Embedding介绍了线性降维的思想，而Neighbor Embedding要介绍的是非线性的降维

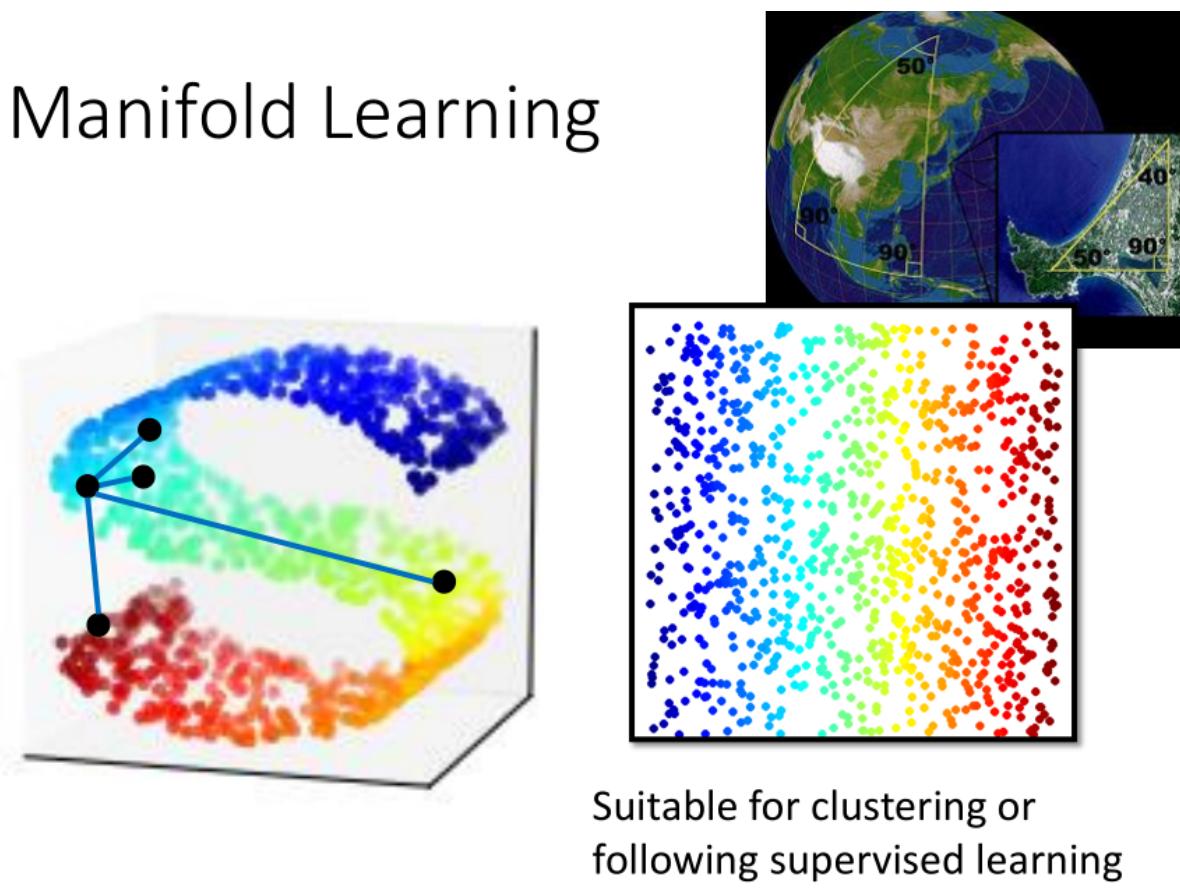
Manifold Learning

我们知道data point可能是在高维空间里面的一个manifold，也就是说：data point的分布其实是在低维的一个空间里，只是被扭曲地塞到高维空间里面。

讲到manifold，常常举的例子是地球，地球的表面就是一个manifold（一个二维的平面，被塞到一个三维的空间里面）。

在manifold里面只有很近距离的点，欧氏距离Euclidean distance才会成立，如果距离很远的时候，欧式几何不一定成立。

所以manifold learning要做的事情是把S型的这块东西展开，把塞到高维空间的低维空间摊平。摊平的好处就是：把这个塞到高维空间里的manifold摊平以后，那我们就可以在这个manifold上面用Euclidean distance来算点和点之间的距离，描述样本点之间的相似程度，这会对接下来你要做supervised learning都是会有帮助的。



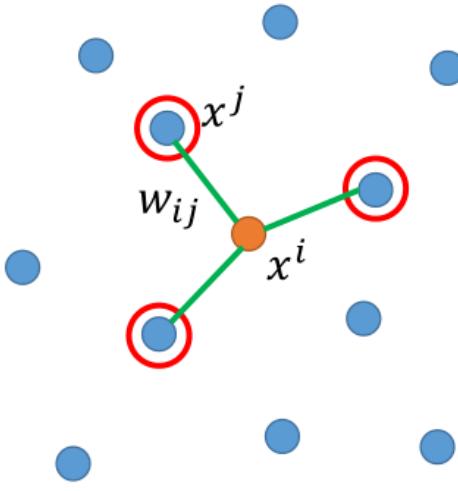
Locally Linear Embedding

局部线性嵌入，locally linear embedding，简称LLE

在原来的空间里面，有某一个点叫做 x^i ，我们先选出 x^i 的neighbor叫做 x^j 。接下来我们找 x^i 跟 x^j 之间的关系，它们之间的关系我们写作 w_{ij} 。

我们假设说：每一个 x^i 都是可以用它的neighbor做linear combination以后组合而成，这个 w_{ij} 是拿 x^j 组成 x^i 的时候，linear combination的weight。因此找点与点的关系 w_{ij} 这个问题就转换成，找一组使得所有样本点与周围点线性组合的差距能够最小的参数 w_{ij} 。那找这一组 w_{ij} 要如何做呢，我们现在找一组 w_{ij} ， x^i 减掉summation over w_{ij} 乘以 x^j 的L2-Norm越接近越好，然后summation over所有的data point i。

$$\sum_i ||x^i - \sum_j w_{ij}x^j||_2$$



w_{ij} represents the relation between x^i and x^j

Find a set of w_{ij} minimizing

$$\sum_i \left\| x^i - \sum_j w_{ij} x^j \right\|_2$$

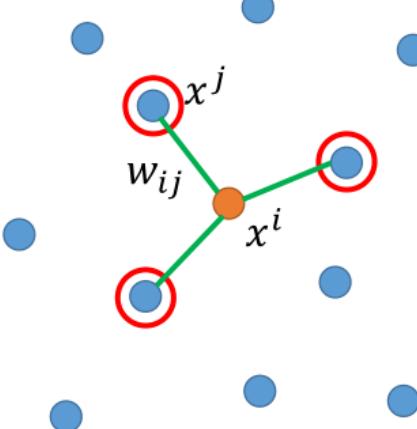
Then find the dimension reduction results z^i and z^j based on w_{ij}

接下来就要做Dimension Reduction, 把 x^i 和 x^j 降维到 z^i 和 z^j , 并且保持降维前后两个点之间的关系 w_{ij} 是不变的

LLE

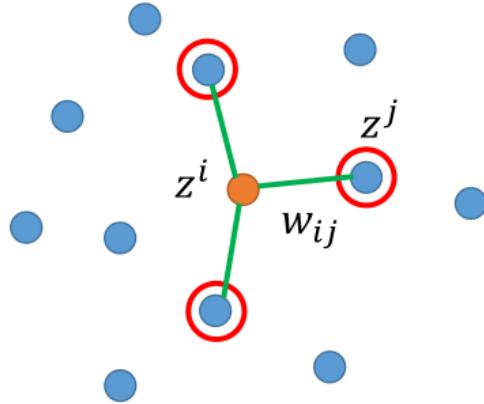
Find a set of z^i minimizing

Keep w_{ij} unchanged



Original Space

$$\sum_i \left\| z^i - \sum_j w_{ij} z^j \right\|_2$$



New (Low-dim) Space

LLE的具体做法如下：

- 在原先的高维空间中找到 x^i 和 x^j 之间的关系 w_{ij} 以后就把它固定住
- 使 x^i 和 x^j 降维到新的低维空间上的 z^i 和 z^j
- z^i 和 z^j 需要minimize下面的式子：

$$\sum_i \|z^i - \sum_j w_{ij} z^j\|_2$$

- 即在原本的空间里， x^i 可以由周围点通过参数 w_{ij} 进行线性组合得到，则要求在降维后的空间里， z^i 也可以用同样的线性组合得到

实际上，LLE并没有给出明确的降维函数，它没有明确地告诉我们怎么从 x^i 降维到 z^i ，只是给出了降维前后的约束条件。它并没有一个明确的function告诉你说我们如何来做dimension reduction，不像我们在做auto encoding的时候，你learn出一个encoding的network，你input一个新的data point，然后你就得到dimension结果。在LLE里面，你并没有找一个明确的function告诉我们，怎么样从一个 x 变到 z ， z 完全就是另外凭空找出来的。

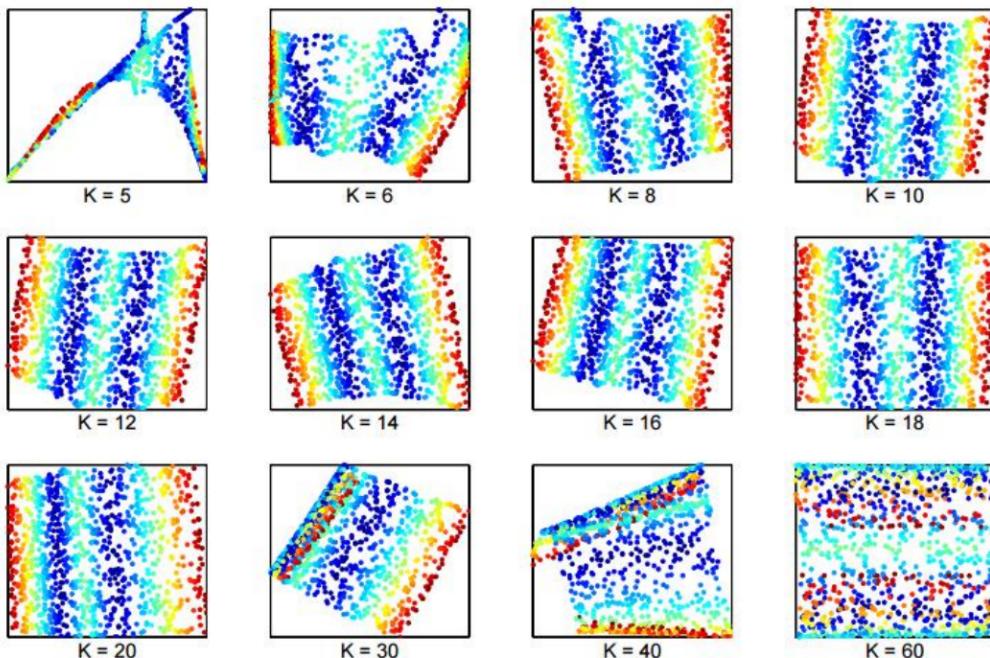
在实际应用LLE的时候，LLE要好好的调neighbor，neighbor的数目要刚刚好，对 x^i 来说，需要选择合适的邻居点数目K才会得到好的结果

下图给出了原始paper中的实验结果，K太小或太大得到的结果都不太好。

为什么k太大，得出的结果也不好呢？因为我们之前的假设是Euclidean distance只是在很近的距离里面可以这样想，当k很大的时候，你会考虑很远的点，所以你不应该把它考虑进来，你的k要选一个适当的值。注意到在原先的空间里，只有距离很近的点之间的关系需要被保持住，如果K选的很大，就会选中一些由于空间扭曲才导致距离接近的点，而这些点的关系我们并不希望在降维后还能被保留。

LLE

Lawrence K. Saul, Sam T. Roweis, “Think Globally, Fit Locally: Unsupervised Learning of Low Dimensional Manifolds”, JMLR, 2013



Laplacian Eigenmaps

Introduction

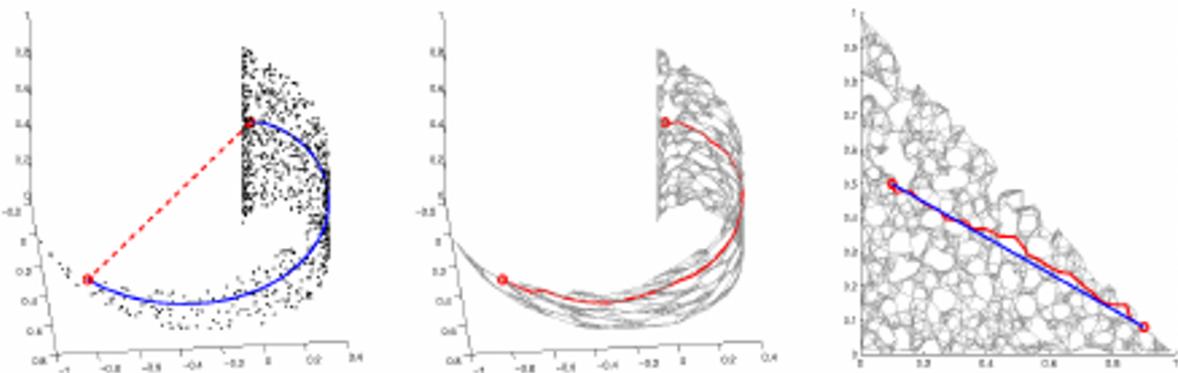
另一种方法叫拉普拉斯特征映射，Laplacian Eigenmaps

之前在semi-supervised learning有提到smoothness assumption，即我们仅知道两点之间的欧氏距离是不够的，还需要观察两个点在high density区域下的距离，如果两个点之间有high density connection，那它们才是真正的很接近。

我们依据某些规则把样本点建立graph，把比较近的点连起来，变成一个graph，那么smoothness的距离就可以被graph上面的connection来approximate

- Graph-based approach

Distance defined by graph approximate the distance on manifold



Construct the data points as a graph

Review for Smoothness Assumption

简单回顾一下在semi-supervised里的说法：如果两个点 x^1 和 x^2 在高密度区域上是相近的，那它们的label y^1 和 y^2 很有可能是一样的

$$L = \sum_{x^r} C(y^r, \hat{y}^r) + \lambda S$$

$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (y^i - y^j)^2 = y^T L y$$

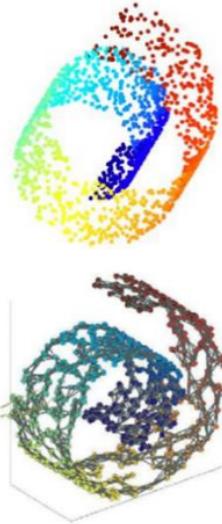
其中 $C(y^r, \hat{y}^r)$ 表示labeled data项， λS 表示unlabeled data项，它就像是一个regularization term，用于判断我们当前得到的label是否是smooth的

其中如果点 x^i 与 x^j 是相连的，则 $w_{i,j}$ 等于相似度，否则为0， S 的表达式希望在 x^i 与 x^j 很接近，相似度 $w_{i,j}$ 很大的情况下，而label差距 $|y^i - y^j|$ 越小越好，同时也是对label平滑度的一个衡量

Laplacian Eigenmaps

$$w_{i,j} = \begin{cases} \text{similarity} & \text{If connected} \\ 0 & \text{otherwise} \end{cases}$$

- *Review in semi-supervised learning:* If x^1 and x^2 are close in a high density region, \hat{y}^1 and \hat{y}^2 are probably the same.



$$L = \sum_{x^r} C(y^r, \hat{y}^r) + \lambda S$$

As a regularization term

$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (y^i - y^j)^2 = \mathbf{y}^T L \mathbf{y}$$

S evaluates how smooth your label is

L: (R+U) x (R+U) matrix

Graph Laplacian

$$L = D - W$$

Application in Unsupervised Task

降维的基本原则：如果 x^i 和 x^j 在 high density 区域上是相近的，即相似度 $w_{i,j}$ 很大，则降维后的 z^i 和 z^j 也需要很接近，总体来说就是让下面的式子尽可能小

$$S = \sum_{i,j} w_{i,j} \|z^i - z^j\|_2$$

这里的 $w_{i,j}$ 表示 x^i 与 x^j 这两点的相似度

如果说 x^1, x^2 在 high desity region 是 close 的，那我们就希望 z^1, z^2 也是相近的。如果 x^i, x^j 两个 data point 很像，那 z^i, z^j 做完 dimension reduction 以后距离就很近，反之 $w_{i,j}$ 很小，距离要怎样都可以。

但光有上面这个式子是不够的，假如令所有的 z 相等，比如令 $z^i = z^j = 0$ ，那上式就会直接停止更新

在 semi-supervised 中，如果所有 label z^i 都设成一样，会使得 supervised 部分的 $\sum_{x^r} C(y^r, \hat{y}^r)$ 变得很大，因此 loss 就会很大，但在这里少了 supervised 的约束，因此我们需要给 z 一些额外的约束：

- 假设降维后 z 所处的空间为 M 维，则 $\{z^1, z^2, \dots, z^N\} = R^M$ ，我们希望降维后的 z 占据整个 M 维的空间，而不希望它分布在一个比 M 更低维的空间里，
- 最终解出来的 z 其实就是 Graph Laplacian L 比较小的特征值所对应的特征向量

这也是 Laplacian Eigenmaps 名称的由来，我们找的 z 就是 Laplacian matrix 的特征向量

如果通过拉普拉斯特征映射找到 z 之后再对其利用 K-means 做聚类，就叫做谱聚类 (spectral clustering)

- *Dimension Reduction*: If x^1 and x^2 are close in a high density region, z^1 and z^2 are close to each other.

$$S = \frac{1}{2} \sum_{i,j} w_{i,j} (z^i - z^j)^2$$

Any problem? How about $z^i = z^j = \mathbf{0}$?

Giving some constraints to z :

If the dim of z is M , $\text{Span}\{z^1, z^2, \dots, z^N\} = \mathbb{R}^M$

Spectral clustering: clustering on z

Belkin, M., Niyogi, P. Laplacian eigenmaps and spectral techniques for embedding and clustering. *Advances in neural information processing systems*. 2002

t-SNE

t-SNE, 全称为T-distributed Stochastic Neighbor Embedding, t分布随机邻居嵌入

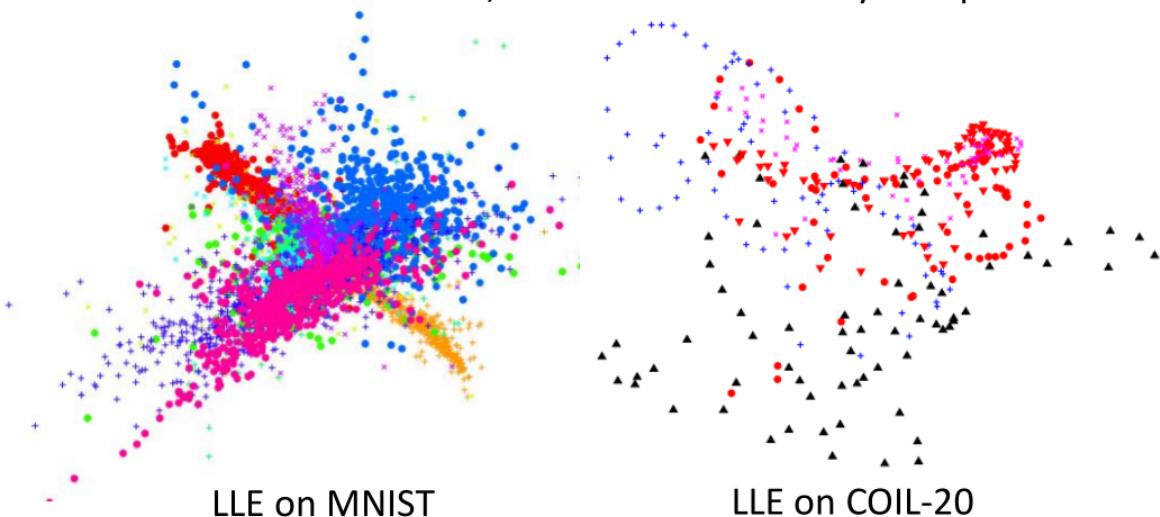
Shortage in LLE

前面的方法只假设了相邻的点要接近，却没有假设不相近的点要分开

所以在MNIST使用LLE会遇到下图的情形，它确实会把同一个class的点都聚集在一起，却没有办法避免不同class的点重叠在一个区域，这就会导致依旧无法区分不同class的现象

COIL-20数据集包含了同一张图片进行旋转之后的不同形态，对其使用LLE降维后得到的结果是，同一个圆圈代表同张图像旋转的不同姿态，但许多圆圈之间存在重叠

- Problem of the previous approaches
- Similar data are close, but different data may collapse



How t-SNE works

做t-SNE同样要降维，把原来的data point x 变成low dimension vector z ，在原来 x 的分布空间上，我们需要计算所有 x^i 与 x^j 之间的相似度 $S(x^i, x^j)$

然后需要将其做归一化： $P(x^j|x^i) = \frac{S(x^i, x^j)}{\sum_{k \neq i} S(x^i, x^k)}$ ，即 x^j 与 x^i 的相似度占所有与 x^i 相关的相似度的比例

将 x 降维到 z ，同样可以计算相似度 $S'(z^i, z^j)$ ，并做归一化： $Q(z^j|z^i) = \frac{S'(z^i, z^j)}{\sum_{k \neq i} S'(z^i, z^k)}$

t-SNE

$x \longrightarrow z$

Compute similarity between all pairs of x : $S(x^i, x^j)$

$$P(x^j|x^i) = \frac{S(x^i, x^j)}{\sum_{k \neq i} S(x^i, x^k)}$$

Compute similarity between all pairs of z : $S'(z^i, z^j)$

$$Q(z^j|z^i) = \frac{S'(z^i, z^j)}{\sum_{k \neq i} S'(z^i, z^k)}$$

Find a set of z making the two distributions as close as possible

$$\begin{aligned} L &= \sum_i KL(P(*|x^i)||Q(*|z^i)) \\ &= \sum_i \sum_j P(x^j|x^i) \log \frac{P(x^j|x^i)}{Q(z^j|z^i)} \end{aligned}$$

这里的归一化是有必要的，因为我们无法判断在 x 和 z 所在的空间里， $S(x^i, x^j)$ 与 $S'(z^i, z^j)$ 的范围是否是一致的，需要将其映射到一个统一的概率区间。

我们希望找到的投影空间 z ，可以让 $P(x^j|x^i)$ 和 $Q(z^j|z^i)$ 的分布越接近越好

所以我们要做的事情就是找一组 z ，它可以做到， x^i 对其他point的distribution跟 z^i 对其他point的distribution，这样的distribution之间的KL距离越小越好，然后summation over 所有的data point，使得这这个值 L 越小越好。

用于衡量两个分布之间相似度的方法就是**KL散度(KL divergence)**，我们的目标就是让 L 越小越好：

$$\begin{aligned} L &= \sum_i KL(P(*|x^i)||Q(*|z^i)) \\ &= \sum_i \sum_j P(x^j|x^i) \log \frac{P(x^j|x^i)}{Q(z^j|z^i)} \end{aligned}$$

KL Divergence

这里简单补充一下KL散度的基本知识

KL 散度，最早是从信息论里演化而来的，所以在介绍 KL 散度之前，我们要先介绍一下信息熵，信息熵的定义如下：

$$H = - \sum_{i=1}^N p(x_i) \cdot \log p(x_i)$$

其中 $p(x_i)$ 表示事件 x_i 发生的概率，信息熵其实反映的就是要表示一个概率分布所需要的平均信息量
在信息熵的基础上，我们定义KL散度为：

$$\begin{aligned} D_{KL}(p||q) &= \sum_{i=1}^N p(x_i) \cdot (\log p(x_i) - \log q(x_i)) \\ &= \sum_{i=1}^N p(x_i) \cdot \log \frac{p(x_i)}{q(x_i)} \end{aligned}$$

$D_{KL}(p||q)$ 表示的就是概率 q 与概率 p 之间的差异，很显然，KL散度越小，说明概率 q 与概率 p 之间越接近，那么预测的概率分布与真实的概率分布也就越接近

How to use

t-SNE会计算所有样本点之间的相似度，运算量会比较大，当在data point比较多的时候跑起来效率会比较低

常见的做法是对原先的空间用类似PCA的方法先做一次降维，然后用t-SNE对这个简单降维空间再做一次更深层次的降维，以期减少运算量。比如说：原来的dimension很大，不会直接从很高的dimension直接做t-SNE，因为这样计算similarity时间会很长，通常会先用PCA做降维，降到50维，再用t-SNE降到2维，这个是比较常见的做法。

值得注意的是，t-SNE的式子无法对新的样本点进行处理，一旦出现新的 x^i ，就需要重新跑一遍该算法，所以**t-SNE通常不是用来训练模型的，它更适合用于做基于固定数据的可视化。**

t-SNE常用于将固定的高维数据可视化到二维平面上。你有一大堆的 x 是high dimension，你想要它在二维空间的分布是什么样子，你用t-SNE，t-SNE会给你往往不错的结果。

Similarity Measure

如果根据欧氏距离计算降维前的相似度，往往采用**RBF function** (Radial Basis Function)
 $S(x^i, x^j) = e^{-\|x^i - x^j\|_2}$ ，这个表达式的好处是，只要两个样本点的欧氏距离稍微大一些，相似度就会下降得很快

在t-SNE之前，有一个方法叫做SNE：dimension reduction以后的space，它选择的measure跟原来的space是一样的 $S'(z^i, z^j) = e^{-\|z^i - z^j\|_2}$ 。

对t-SNE来说，它在降维后的新空间所采取的相似度算法是与之前不同的，它选取了**t-distribution**中的一种，即 $S'(z^i, z^j) = \frac{1}{1 + \|z^i - z^j\|_2}$

以下图为例，假设横轴代表了在原先 x 空间上的欧氏距离 $\|x^i - x^j\|_2$ 或者做降维之后在 z 空间上的欧氏距离 $\|z^i - z^j\|_2$ ，红线代表RBF function，是降维前的分布；蓝线代表了t-distribution，是降维后的分布

你会发现，降维前后相似度从RBF function到t-distribution：

- 如果原先两个点距离(Δx)比较近，则降维转换之后，如果要维持原先的相似度，它们的距离依旧是比较接近的
- 如果原先两个点距离(Δx)比较远，则降维转换之后，如果要维持原先的相似度，它们的距离会被拉得更远

Ignore σ for simplicity

t-SNE –Similarity Measure

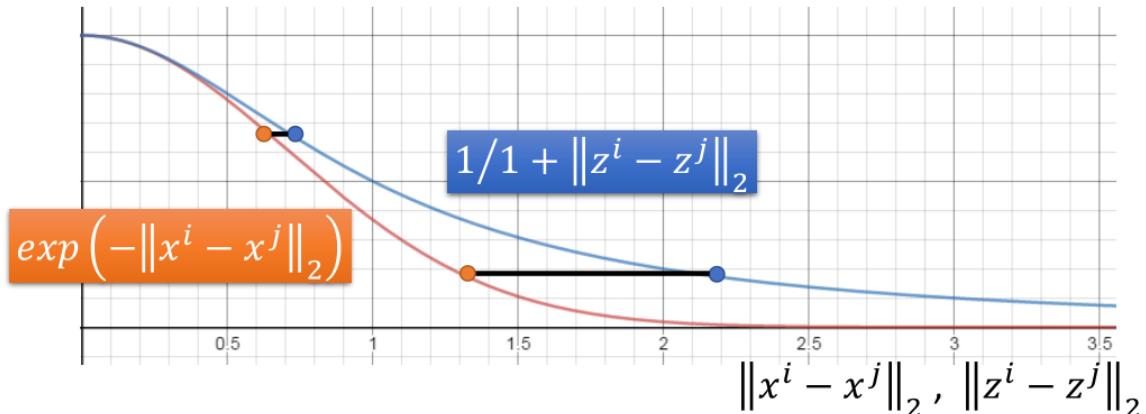
SNE:

$$S(x^i, x^j) = \exp(-\|x^i - x^j\|_2)$$

$$= \exp(-\|x^i - x^j\|_2)$$

t-SNE:

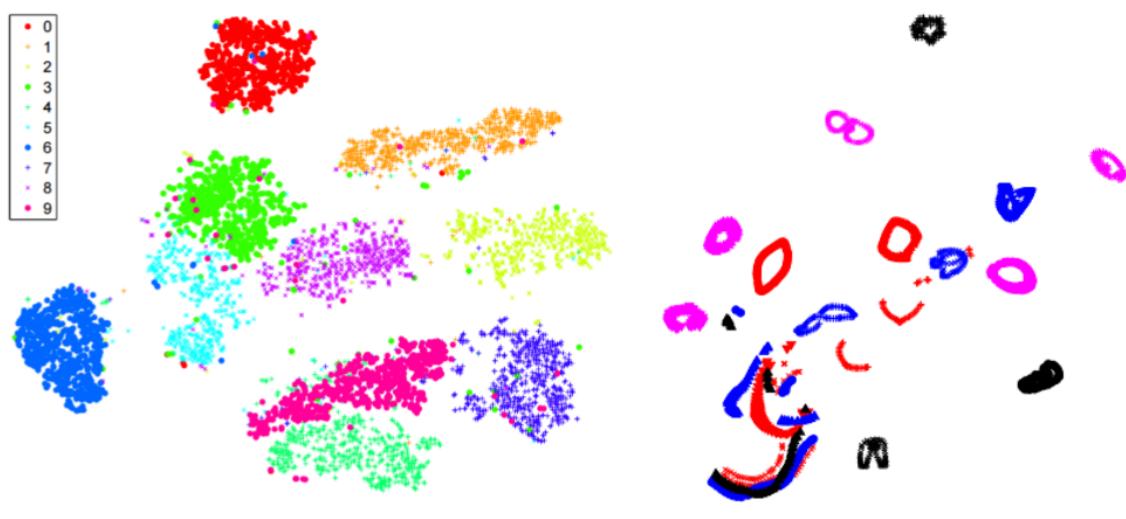
$$S'(z^i, z^j) = 1 / (1 + \|z^i - z^j\|_2)$$



也就是说t-SNE可以聚集相似的样本点，同时还会放大不同类别之间的距离，从而使得不同类别之间的分界线非常明显，特别适用于可视化。

下图则是对MNIST和COIL-20先做PCA降维，再做t-SNE降维可视化的结果，t-SNE画出来的图往往长的这样，它会把你的data point 聚集成一群一群的，只要你的data point 离的比较远，那做完t-SNE之后，就会强化，变得更远了。

- Good at visualization



如图为t-SNE的动画。因为这是利用gradient descent 来train的，所以你会看到随着iteration process，点会被分的越来越开。

Conclusion

小结一下，本文主要介绍了三种非线性降维的算法：

- LLE(Locally Linear Embedding)，局部线性嵌入算法，主要思想是降维前后，每个点与周围邻居的线性组合关系不变， $x^i = \sum_j w_{ij}x^j$ 、 $z^i = \sum_j w_{ij}z^j$
- Laplacian Eigenmaps，拉普拉斯特征映射，主要思想是在high density的区域，如果 x^i 、 x^j 这两个点相似度 $w_{i,j}$ 高，则投影后的距离 $\|z^i - z^j\|_2$ 要小
- t-SNE(t-distribution Stochastic Neighbor Embedding)，t分布随机邻居嵌入，主要思想是，通过降维前后计算相似度由RBF function转换为t-distribution，在聚集相似点的同时，拉开不相似点的距离，比较适合用在数据固定的可视化领域

Deep Auto-encoder

文本介绍了自编码器的基本思想，与PCA的联系，从单层编码到多层的变化，在文字搜索和图像搜索上的应用，预训练DNN的基本过程，利用CNN实现自编码器的过程，加噪声的自编码器，利用解码器生成图像等内容

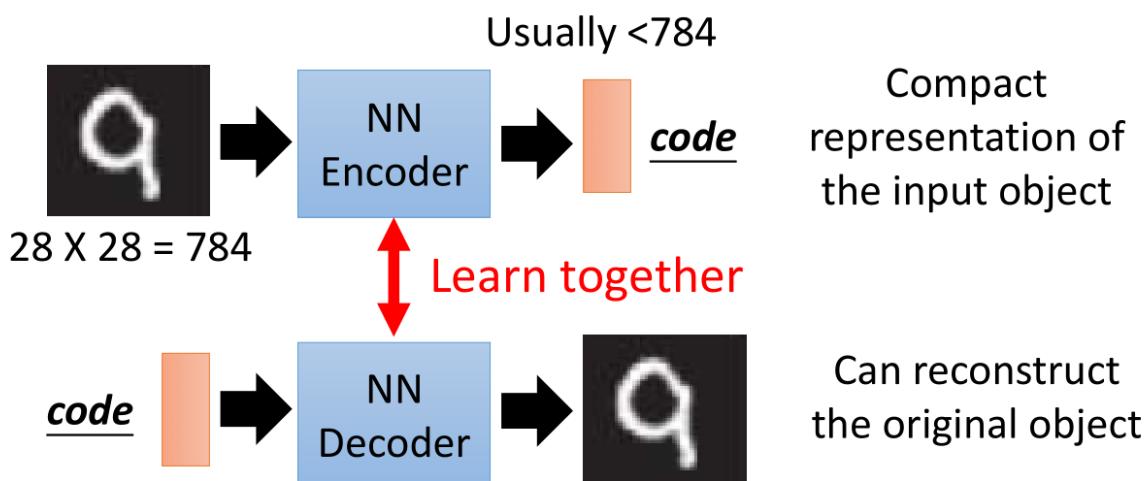
自动编码器的想法是这样子的：我们先去找一个encoder，这个encoder input一个东西(假如说，我们来做NMIST的话，就是input一张digit，它是784维的vector)，这个encoder可能就是一个neural network，它的output就是code(这个code远比784维要小的，类似压缩的效果)，这个coder代表了原来input一张image compact representation。

但是现在问题是：我们现在做的是Unsupervised learning，你可以找到一大堆的image当做这个NN encoder的input，但是我们不知道任何的output。你要learn一个network，只有一个input，你没有办法learn它。那没有关系，我们要做另外一件事情：想要learn一个decoder，decoder做的事情就是：input一个vector，它就通过这个NN decoder，它的output就是一张image。但是你也没有办法train一个NN decoder，因为你只要output，没有input。

这两个network，encoder decoder单独你是没有办法去train它。但是我们可以把它接起来，然后一起train。也就是说：接一个neural network，input一张image，中间变成code，再把code变成原来的image。这样你就可以把encoder跟decoder一起学，那你就同时学出来了。

Auto-encoder本质上就是一个自我压缩和解压的过程，我们想要获取压缩后的code，它代表了对原始数据的某种紧凑精简的有效表达，即降维结果，这个过程中我们需要：

- Encoder(编码器)，它可以把原先的图像压缩成更低维度的向量
- Decoder(解码器)，它可以把压缩后的向量还原成图像



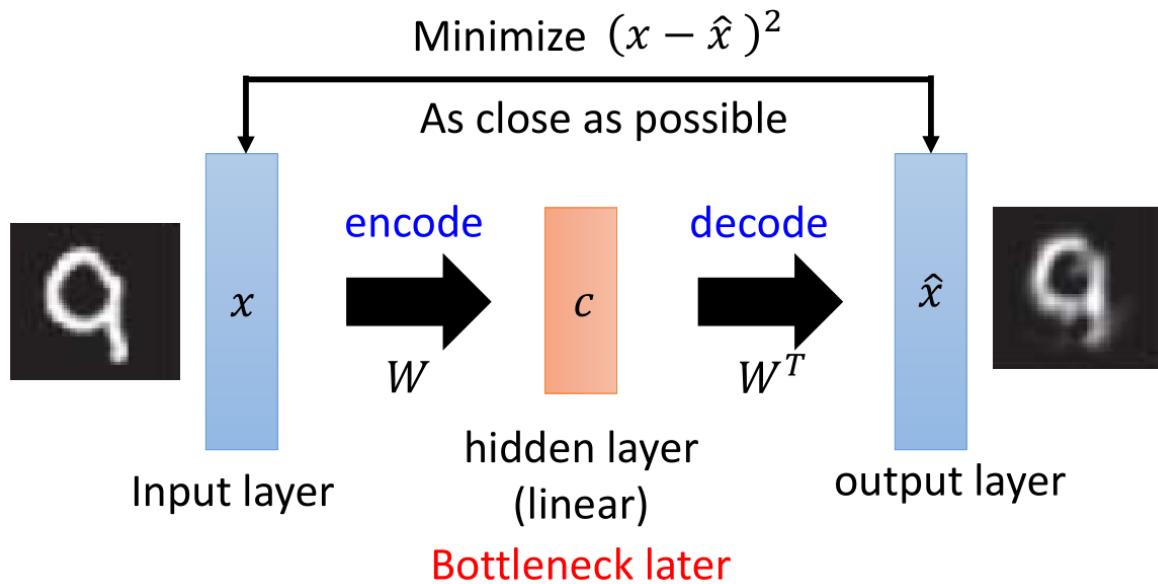
Encoder和Decoder单独拿出一个都无法进行训练，我们需要把它们连接起来，这样整个神经网络的输入和输出都是我们已有的图像数据，就可以同时对Encoder和Decoder进行训练，而降维后的编码结果就可以从最中间的那层hidden layer中获取

Compare with PCA

那我们刚才在PCA里面看过非常类似的概念，我们讲过：PCA其实在做的事情是：input一张image x (在刚才的例子里面，我们会让 $x - \bar{x}$ 当做input，这边我们把减掉 \bar{x} 省略掉，省略掉并不会太奇怪，因为通常在做NN的时候，你拿到的数据其实会normalize，其实你的data mean是为0，所以就不用再去减掉mean)，把 x 乘以一个weight，通过NN一个layer得到component weight c ， c 乘以matrix w 的transpose得到 \hat{x} 。 \hat{x} 是根据这些component的reconstruction的结果。

实际上PCA用到的思想与Auto-encoder非常类似，PCA的过程本质上就是按组件拆分，再按组件重构的过程

在PCA中，我们先把均一化后的 x 根据组件 W 分解到更低维度的 c ，然后再将组件权重 c 乘上组件的转置 W^T 得到重组后的 \hat{x} ，同样我们期望重构后的 \hat{x} 与原始的 x 越接近越好



Output of the hidden layer is the code

如果把这个过程看作是神经网络，那么原始的 x 就是input layer，重构 \hat{x} 就是output layer，中间组件分解权重 c 就是hidden layer，在PCA中它是linear的，我们通常又叫它瓶颈层(Bottleneck layer)。你可以用gradient descent来解PCA。

hidden layer的output就是我们要找的那些code。由于经过组件分解降维后的 c ，维数要远比输入输出层来得低，因此hidden layer实际上非常窄，因而有瓶颈层的称呼。

对比于Auto-encoder，从input layer到hidden layer的按组件分解实际上就是编码(encode)过程，从hidden layer到output layer按组件重构实际上就是解码(decode)的过程。

这时候你可能会想，可不可以用更多层hidden layer呢？答案是肯定的

Deep Auto-encoder

Multi Layer

对deep的自编码器来说，实际上就是通过多级编码降维，再经过多级解码还原的过程

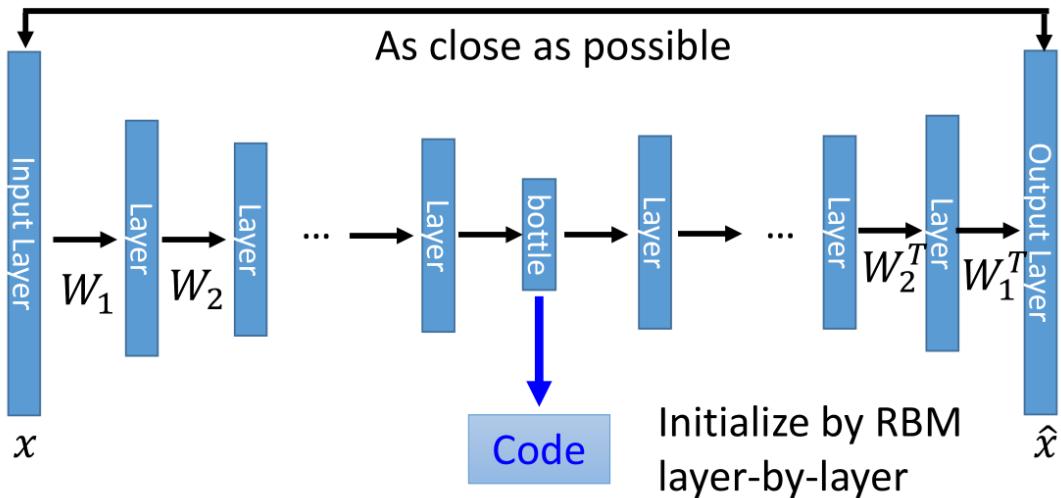
此时：

- 从input layer x 到bottleneck layer的部分都属于Encoder
- 从bottleneck layer到output layer \hat{x} 的部分都属于Decoder
- bottleneck layer的output就是自编码结果code

Deep Auto-encoder

Symmetric is not necessary.

- Of course, the auto-encoder can be deep



Reference: Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313.5786 (2006): 504-507

注意到，如果按照PCA的思路，则Encoder的参数 W_i 需要和Decoder的参数 W_i^T 保持一致的对应关系，这样做的好处是，可以节省一半的参数，降低overfitting的概率

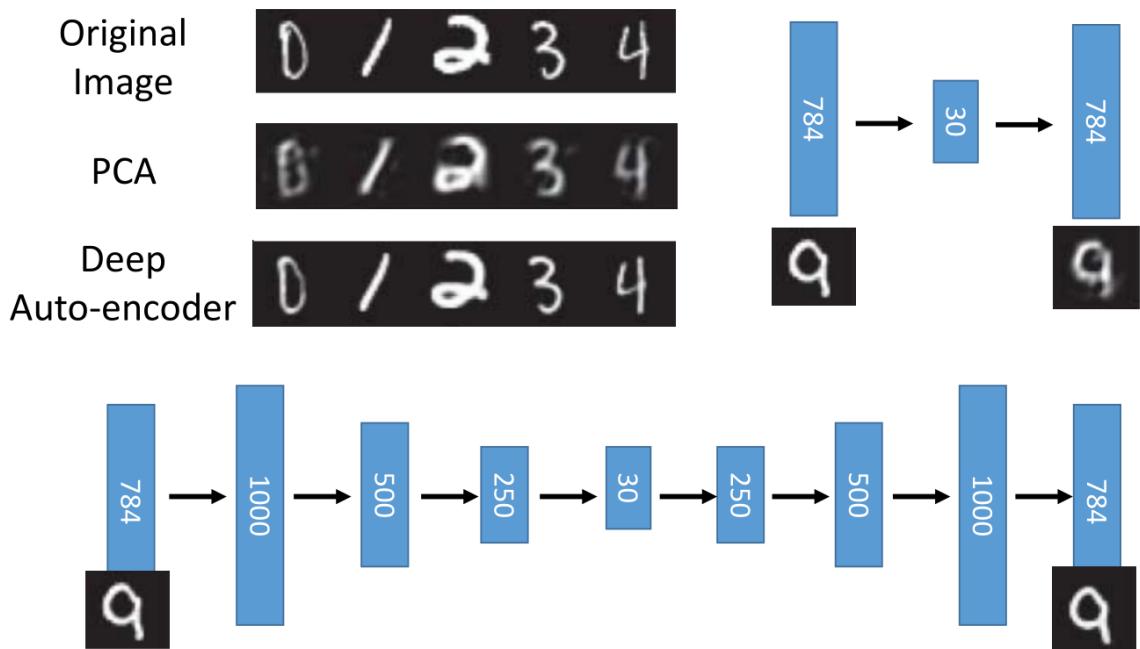
但这件事情并不是必要的，实际操作的时候，你完全可以对神经网络用Backpropagation直接train下去，而不用保持编码器和解码器的参数一致

Visualize

下图给出了Hinton分别采用PCA和Deep Auto-encoder对手写数字进行编码解码后的结果。

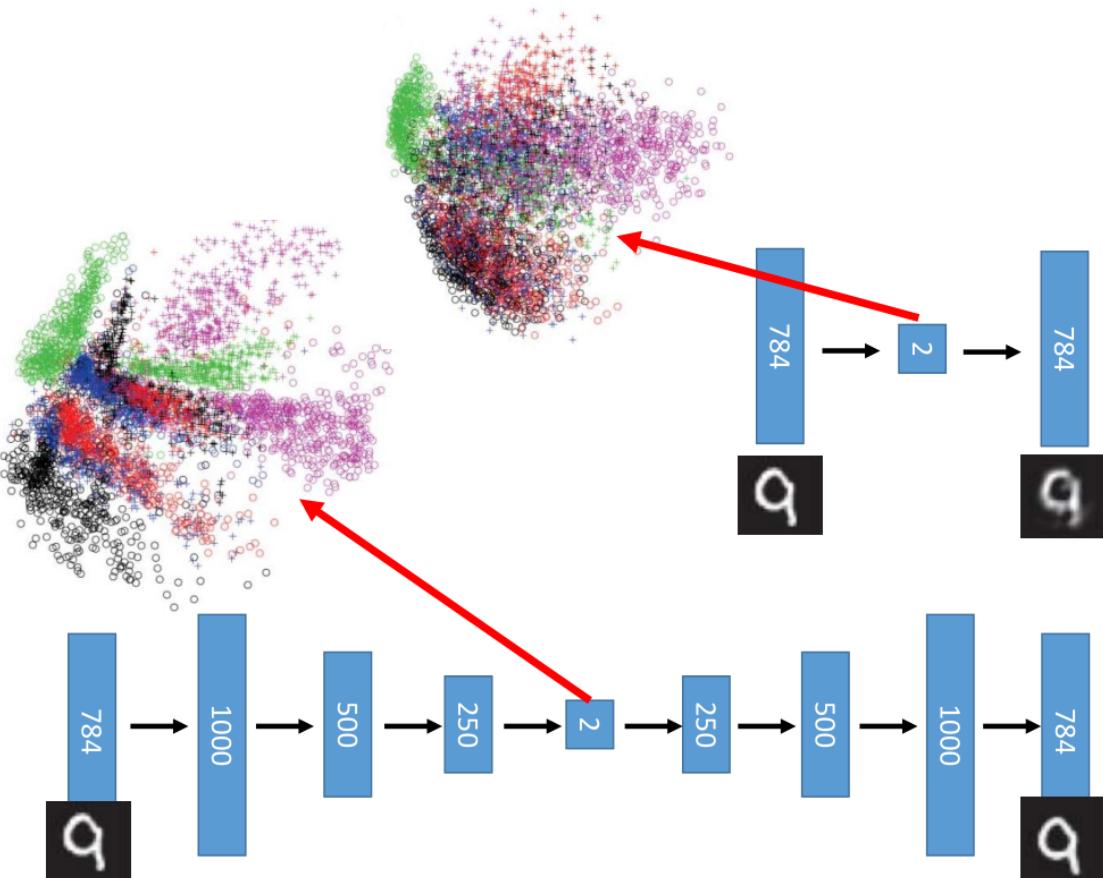
original image做PCA，从784维降到30维，然后从30维reconstruction回784维，得到的image差不多，可以看出它是比较模糊的。

如果是用deep encoder的话，784维先扩为1000维，再不断下降，下降到30维（你很难说为什么它会设计成这样子），然后再把它解回来。你会发现，如果用的是deep Auto-encoder的话，它的结果看起来非常的好。



如果将其降到2维平面做可视化，不同颜色代表不同的数字，可以看到

- 通过PCA降维得到的编码结果中，不同颜色代表的数字被混杂在一起
- 通过Deep Auto-encoder降维得到的编码结果中，不同颜色代表的数字被分散成一群一群的



Text Retrieval

Auto-encoder也可以用在文字处理上，比如说：我们把一篇文章压成一个code。

比如我们要做文字检索，很简单的一个做法是Vector Space Model，把每一篇文章都表示成空间中的一个vector

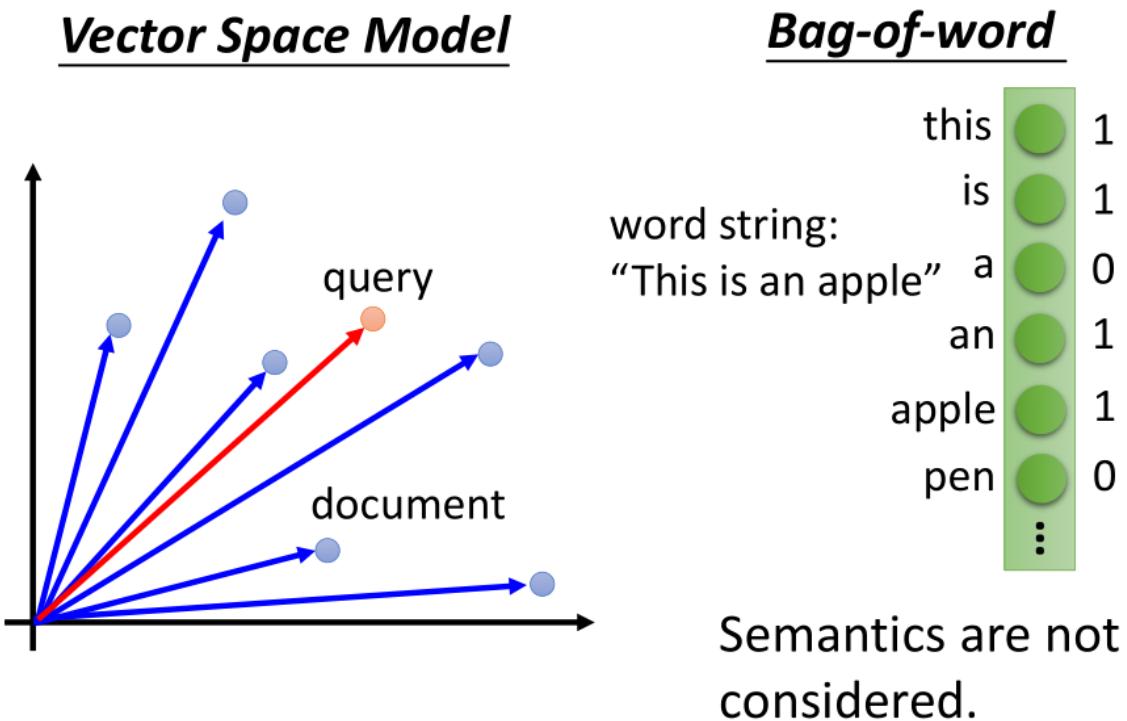
假设查询者输入了某个词汇，那我们就把该查询词汇也变成空间中的一个点，并计算query和每一篇document之间的内积(inner product)或余弦相似度(cos-similarity)

注：余弦相似度有均一化的效果，可能会得到更好的结果

下图中跟query向量最接近的几个向量的cosine-similarity是最大的，于是可以从这几篇文章中去检索
实际上这个模型的好坏，就取决于从document转化而来的vector的好坏，它是否能够充分表达文章信息

Bag-of-word

把一个document表示成一个vector，最简单的表示方法是Bag-of-word，维数等于所有词汇的总数，某
一维等于1则表示该词汇在这篇文章中出现，此外还可以根据词汇的重要性将其加权；但这个模型是非常
weak的，它没有考虑任何Semantics相关的东西，对它来说每个词汇都是相互独立的。



Auto-encoder

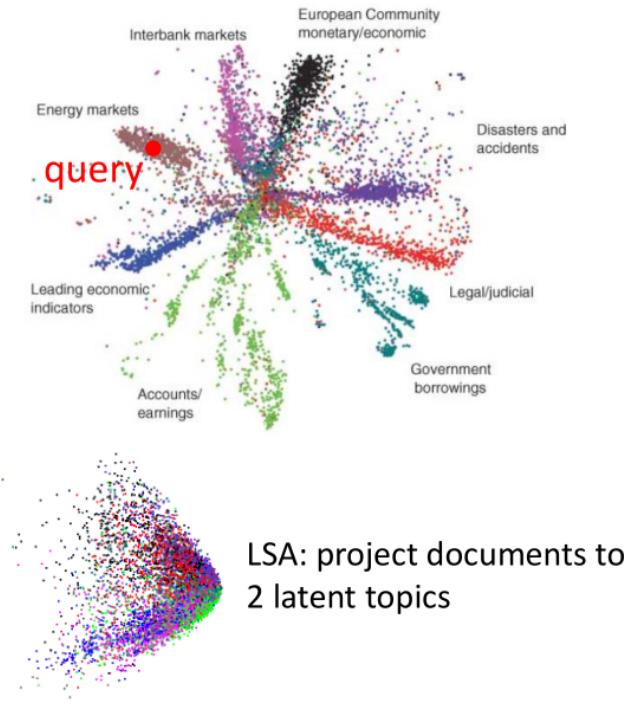
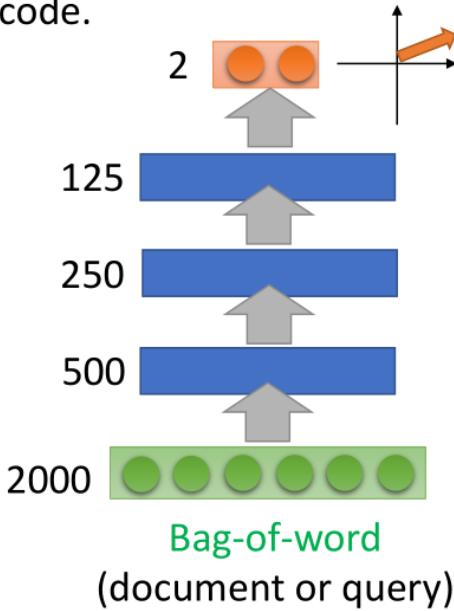
我们可以把它作为Auto-encoder的input，通过降维来抽取有效信息，以获取所需的vector

同样为了可视化，这里将Bag-of-word降维到二维平面上，下图中每个点都代表一篇文章，不同颜色则代
表不同的文章类型

我们可以用Auto-encoder让语义被考虑进来，举例来说，你learn一个Auto-encoder，它的input就是一
个document 或一个query，通过encoder把它压成二维。

每一个点代表一个document，不同颜色代表document属于哪一类。今天要做搜寻的时候，今天输入一
个词汇，那你就把那个query也通过这个encoder把它变为一个二维的vector。假设query落在某一类，你
就可以知道这个query与哪一类有关，就把document retrieve出来。

The documents talking about the same thing will have close code.



在矩阵分解(Matrix Factorization)中，我们介绍了LSA算法，它可以用来寻找每个词汇和每篇文章背后的隐藏关系(vector)，在这里我们采用LSA，并使用二维latent vector来表示每篇文章。

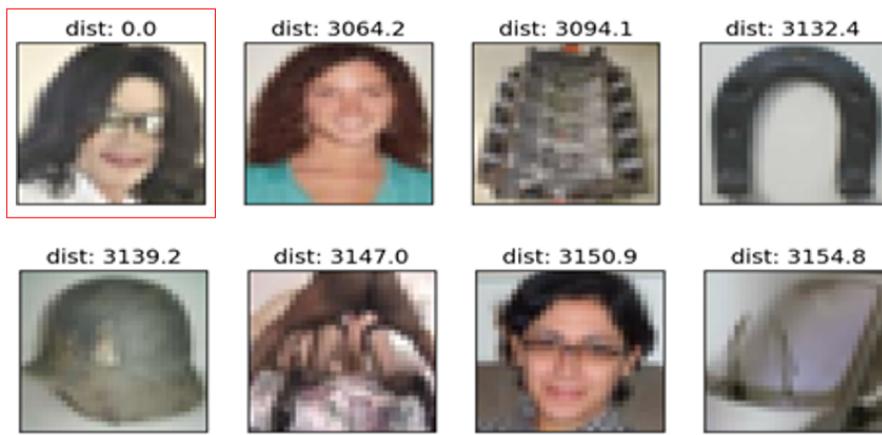
Auto-encoder的结果是相当惊人的。则如果用LSA的话，得不到类似的结果。

Similar Image Search

Auto-encoder同样可以被用在图像检索上

image search最简单的做法就是直接对image query与database中的图片计算pixel的相似度，并挑出最像的图片，但这种方法的效果是不好的，因为单纯的pixel所能够表达的信息太少了。

Retrieved using Euclidean distance in pixel intensity space



(Images from Hinton's slides on Coursera)

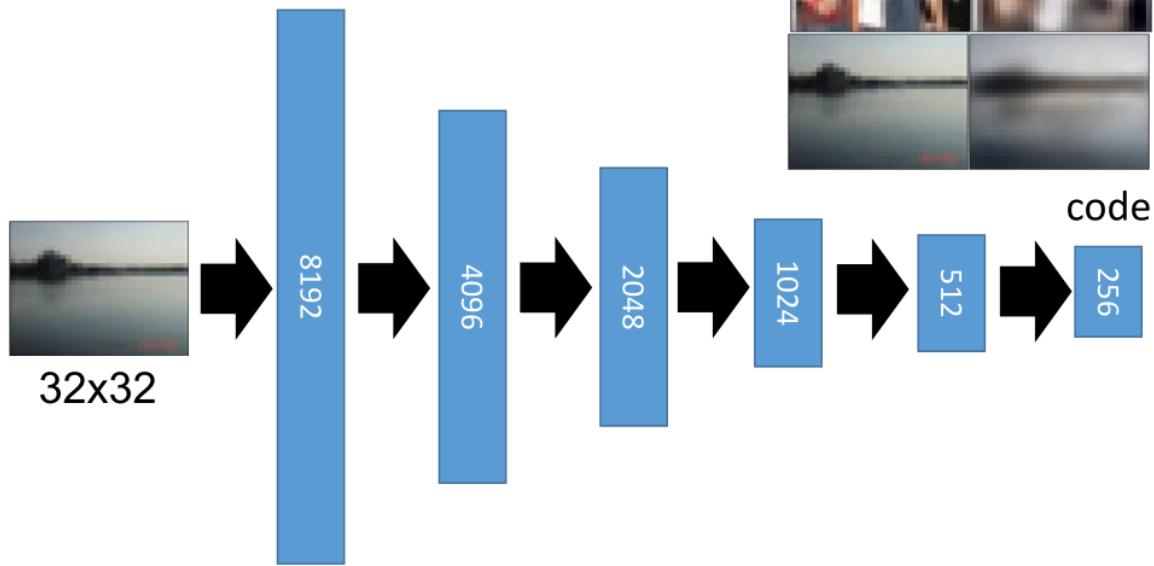
Reference: Krizhevsky, Alex, and Geoffrey E. Hinton. "Using very deep autoencoders for content-based image retrieval." *ESANN*. 2011.

我们需要使用Auto-encoder对图像进行降维和特征提取，把每一张image变成一个code，然后再code上面去做搜寻，在编码得到的code所在空间做检索。

learn一个Auto-encoder是unsupervised，所以你要多少data都行（supervised是很缺data的，unsupervised是不缺data的）

input一张 32×32 的image，每一个pixel用RGB来表示($32 \times 32 \times 3$)，变成8192维，然后dimension reduction变成4096维，最后一直变为256维，你用256维的vector来描述这个image。然后你把这个code再通过另外一个decoder（形状反过来，变成原来的image），它的reconstruction是右上角如图。

Auto-encoder – Similar Image Search



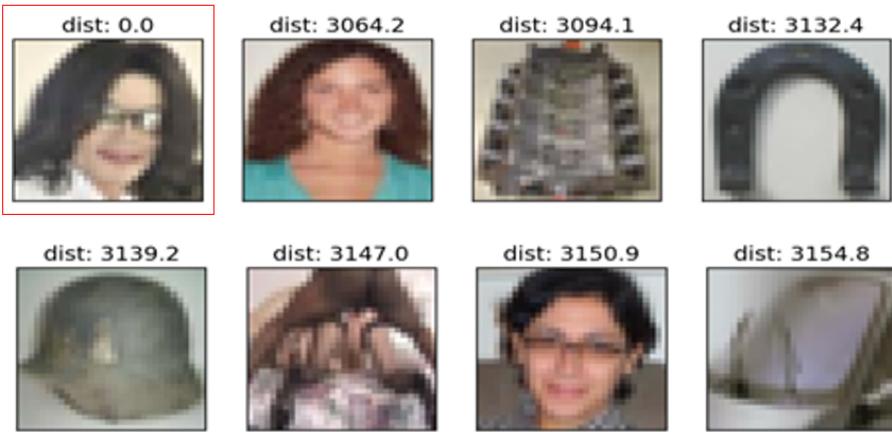
(crawl millions of images from the Internet)

这么做的好处如下：

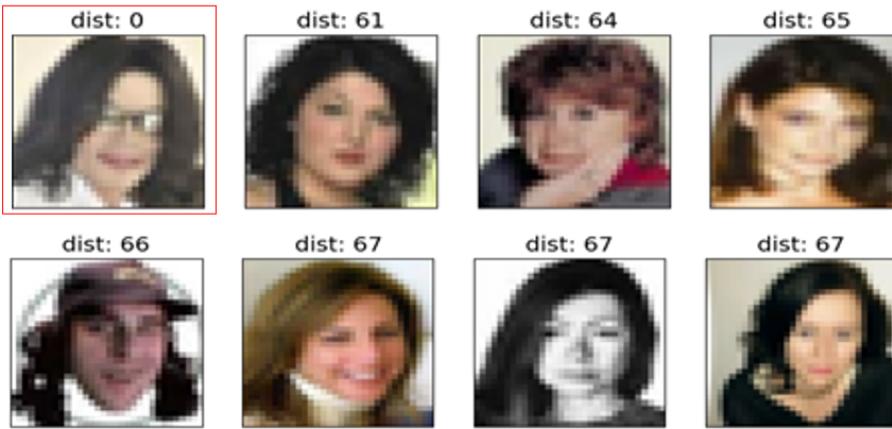
- Auto-encoder可以通过降维提取出一张图像中最有用的特征信息，包括pixel与pixel之间的关系
- 降维之后数据的size变小了，这意味着模型所需的参数也变少了，同样的数据量对参数更少的模型来说，可以训练出更精确的结果，一定程度上避免了过拟合的发生
- Auto-encoder是一个无监督学习的方法，数据不需要人工打上标签，这意味着我们只需简单处理就可以获得大量的可用数据

如果你不是在pixel上算相似度，是在code上算相似度的话，你就会得到比较好的结果。举例来说：你是用Jackson当做image的话，你找到的都是人脸，相比之前的结果进步了一些。可能这个image在pixel label上面看起来是不像的，但是你通过很多的hidden layer把它转成code的时候，在那个256维的空间上看起来是像的，可能在投影空间中某一维就代表了人脸的特征，因此能够被检索出来。

Retrieved using Euclidean distance in pixel intensity space



retrieved using 256 codes



Pre-training

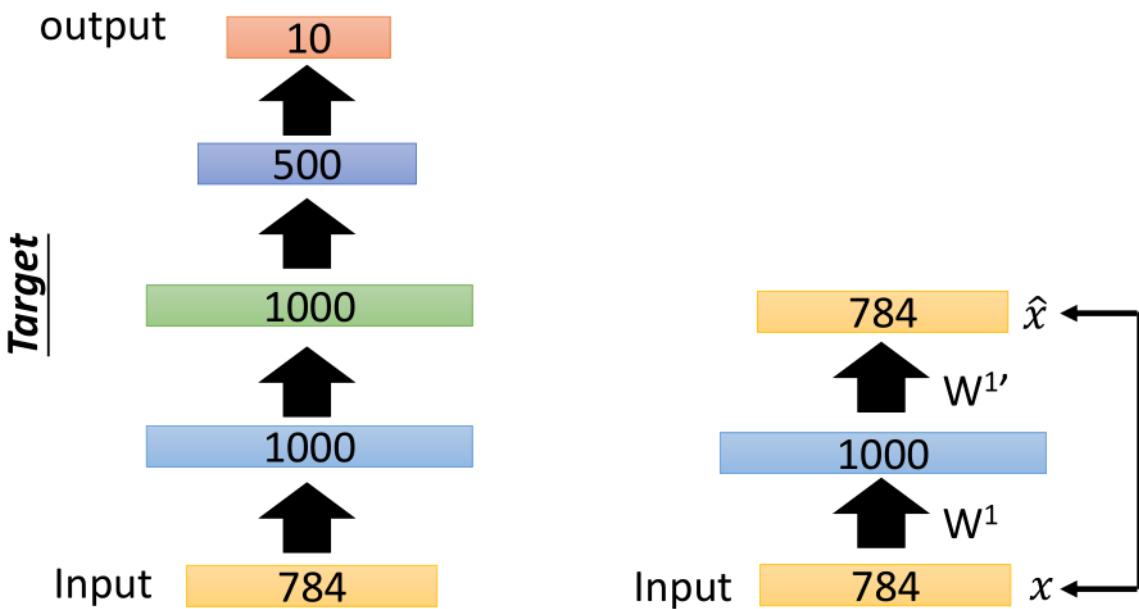
在训练神经网络的时候，我们一般都会对如何做参数的initialization比较困扰，预训练(pre-training)是一种寻找比较好的参数initialization的方法，而我们可以用Auto-encoder来做pre-training

以MNIST数据集为例，我们使用的neural network input 784维，第一个hidden layer是1000维，第二个hidden layer是1000维，第三个hidden layer是500维，然后到10维。

我们对每层hidden layer都做一次auto-encoder，使每一层都能够提取到上一层最佳的特征向量

Greedy Layer-wise Pre-training

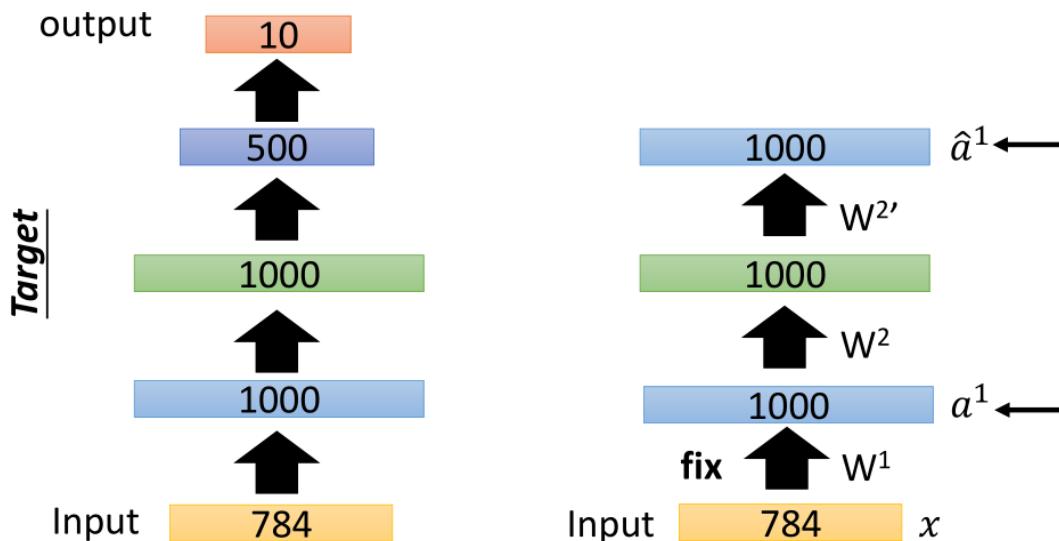
那我做Pre-training的时候，我先train一个Auto-encoder，这个Auto-encoder input784维，中间有1000维的vector，然后把它变回784维，我期望input 跟output越接近越好。



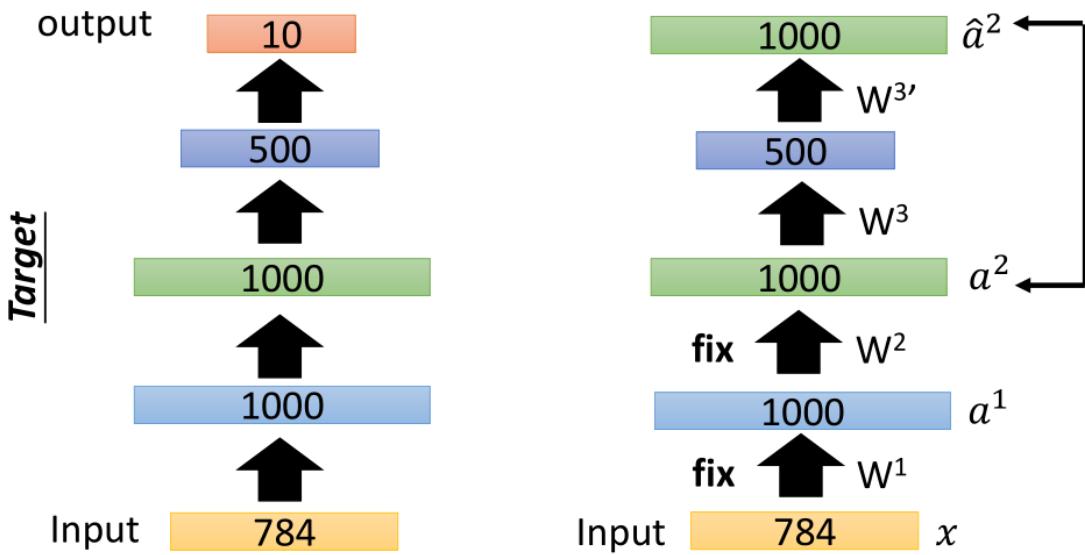
在做这件事的时候，你要稍微小心一点，我们一般做Auto-encoder的时候，你会希望你的coder要比dimension还要小。比dimension还要大的话，你会遇到的问题是：它突然就不learn了，把784维直接放进去，得到一个接近identity的matrix。

所以你今天发现你的hidden layer比你的input还要大的时候，你要加一个很强的regularization在1000维上，你可以对这1000维的output做L1的regularization，可以希望说：这1000维的output里面，只有某几维是可以有值的，其他维要必须为0。这样你就可以避免Auto-encoder直接把input背起来再输出的问题。总之你今天的code比你input还要大，你要注意这种问题。

- 首先使input通过一个如上图的Auto-encoder，input784维，code1000维，output784维，learn参数，当该自编码器训练稳定后（它会希望input跟output越接近越好），就把参数 W^1 fix住。然后将数据集中所有784维的图像都转化为1000维的vector
- 接下来再让这些1000维的vector通过另外一个Auto-encoder，input1000维，code1000维，output1000维learn参数，当其训练稳定后，再把参数 W^2 固定住，对数据集再做一次转换



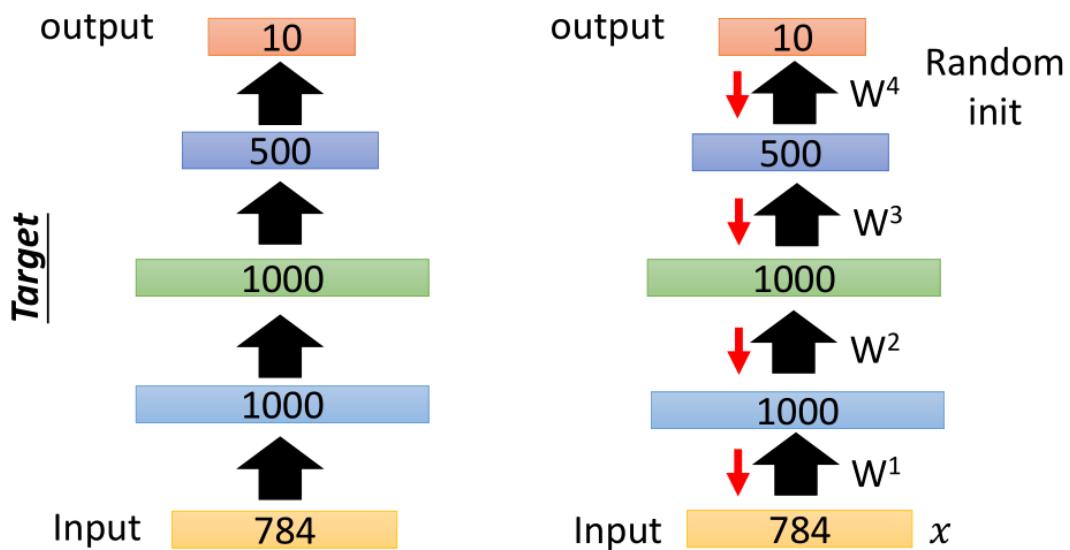
- 接下来再用转换后的数据集去训练第三个Auto-encoder，input1000维，code500维，output1000维，训练稳定后固定 W^3 ，数据集再次更新转化为500维



- 此时三个隐藏层的参数 W^1 、 W^2 、 W^3 就是训练整个神经网络时的参数初始值
- 然后random initialization最后一个隐藏层到输出层之间的参数 W^4
- 再用backpropagation去调整一遍参数，因为 W^1 、 W^2 、 W^3 都已经是很好的weight了，这里只是做微调，因此这个步骤称为**Find-tune**

- Greedy Layer-wise Pre-training again

Find-tune by backpropagation



pre-training在过去learn一个deep neural network还是很需要的，不过现在neural network不需要pre-training往往都能train的起来。由于现在训练机器的条件比以往更好，因此pre-training并不是必要的，但它也有自己的优势。

如果你今天有很多的unlabeled data，少量的labeled data，你可以用大量的unlabeled data先去把 W^1 、 W^2 、 W^3 learn 好，最后再用labeled data去微调 W^1 ~ W^4 即可。所以pre-training在大量的unlabeled data时还是有用的。

De-noising Auto-encoder

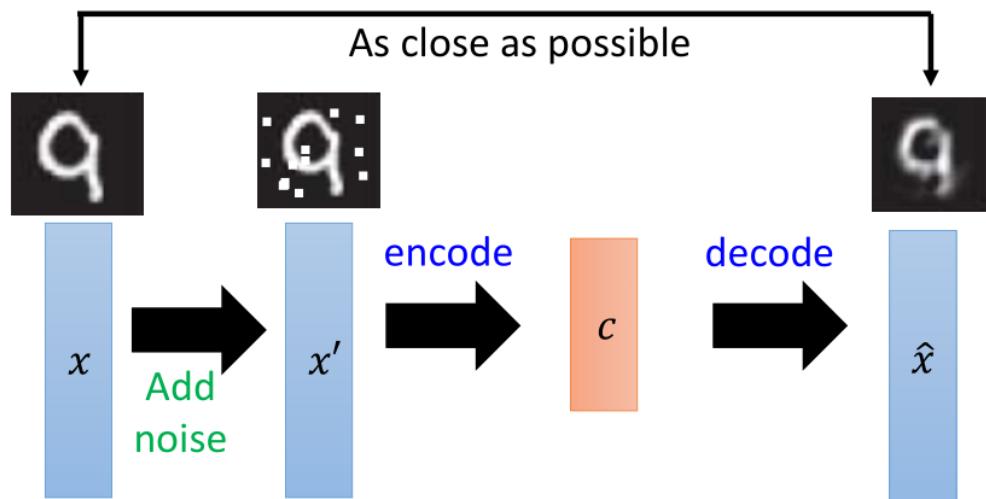
去噪自编码器的基本思想是，把输入的 x 加上一些噪声(noise)变成 x' ，再对 x' 依次做编码(encode)和解码(decode)，得到还原后的 y

值得注意的是，一般的自编码器都是让输入输出尽可能接近，但在去噪自编码器中，我们的目标是让解码后的 y 与加噪声之前的 x 越接近越好

这种方法可以增加系统的鲁棒性，因为此时的编码器Encoder不仅仅是在学习如何做编码，它还学习到了如何过滤掉噪声这件事情

Auto-encoder

- De-noising auto-encoder



Vincent, Pascal, et al. "Extracting and composing robust features with denoising autoencoders." ICML, 2008.

Contractive Auto-encoder

收缩自动编码器的基本思想是，在做encode编码的时候，要加上一个约束，它可以使得：当input有变化的时候，对code的影响是被minimize的。

这个描述跟去噪自编码器很像，只不过去噪自编码器的重点在于加了噪声之后依旧可以还原回原先的输入，而收缩自动编码器的重点在于加了噪声之后能够保持编码结果不变。

Restricted Boltzmann Machine

还有很多non-linear 的 dimension reduction的方法，比如Restricted Boltzmann Machine，它不是NN

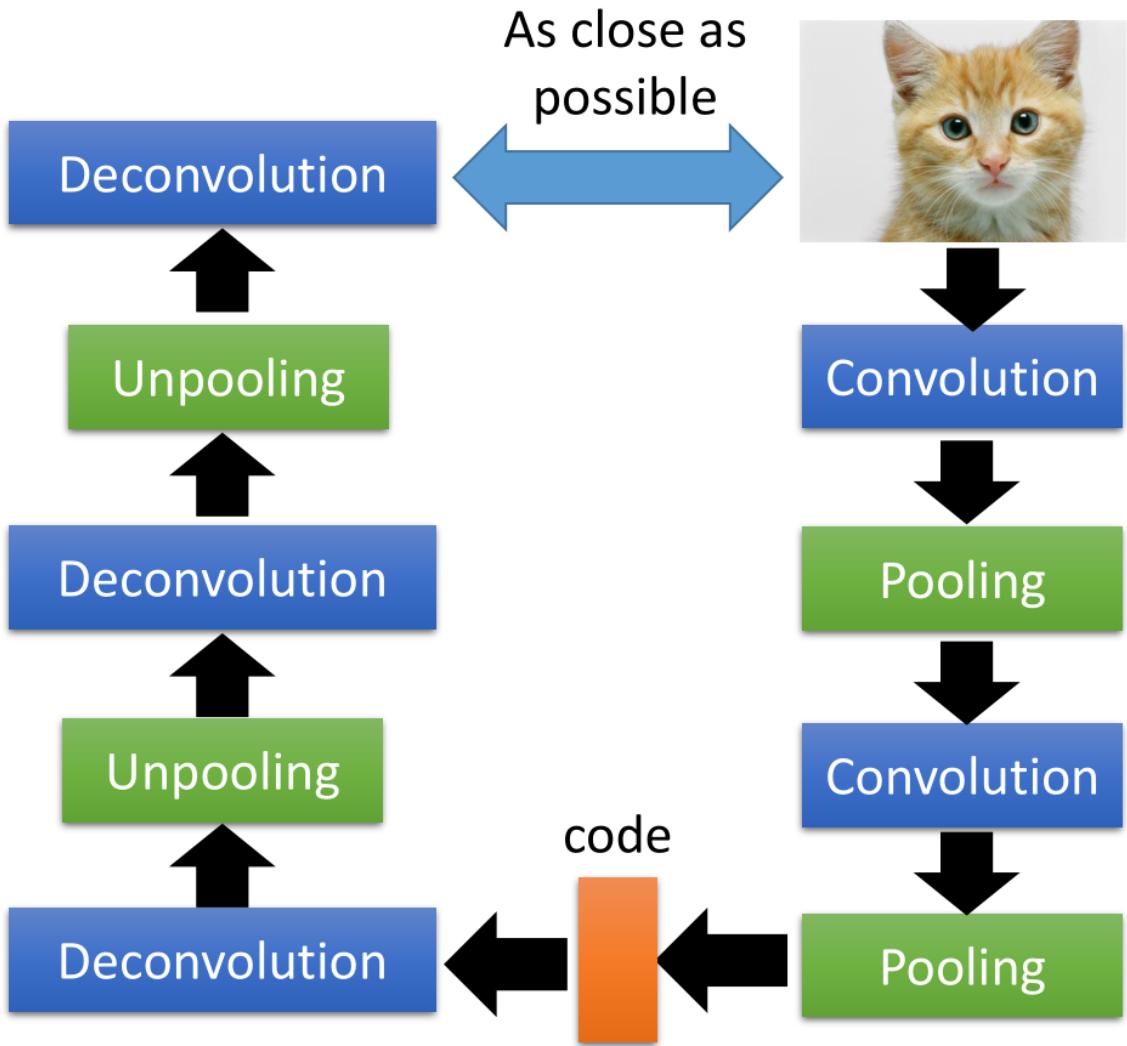
Deep Belief Network

和RBM一样，只是看起来比较像NN，但是并不是NN

Auto-encoder for CNN

处理图像通常都会用卷积神经网络CNN，它的基本思想是交替使用卷积层和池化层，让图像越来越小，最终展平，这个过程跟Encoder编码的过程其实是类似的。

理论上要实现自编码器，Decoder只需要做跟Encoder相反的事即可，那对CNN来说，解码的过程也就变成了交替使用去卷积层和去池化层即可



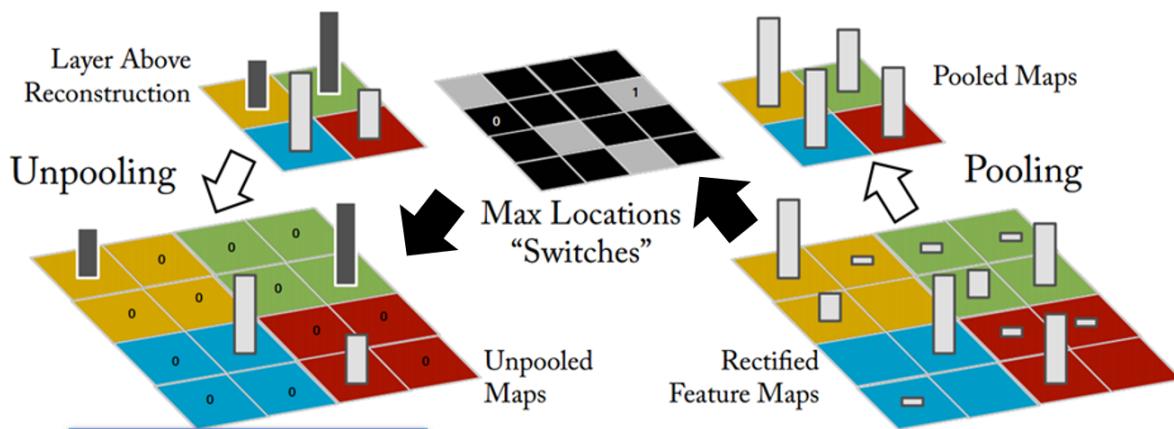
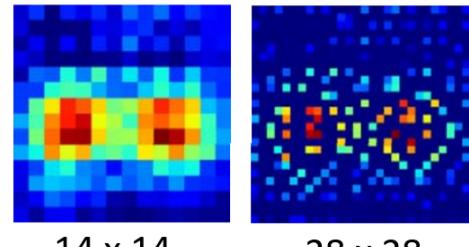
那什么是去卷积层(Deconvolution)和去池化层(Unpooling)呢?

Unpooling

做pooling的时候，假如得到一个 4×4 的matrix，就把每4个pixel分为一组，从每组中挑一个最大的留下，此时图像就变成了原来的四分之一大小

如果还要做Unpooling，就需要提前记录pooling所挑选的pixel在原图中的位置，下图中用灰色方框标注

CNN -Unpooling



Source of image :
https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/image_segmentation.html

然后做Unpooling，就要把当前的matrix放大到原来的四倍，也就是把 2×2 matrix里的pixel按照原先记录的位置插入放大后的 4×4 matrix中，其余项补0即可。

做完unpooling以后，比较小的image会变得比较大，比如说：原来是 $14 * 14$ 的image会变成 $28 * 28$ 的image。你会发现说：它就是把原来的 $14 * 14$ 的image做一下扩散，在有些地方补0。

当然这不是唯一的做法，在Keras中，pooling并没有记录原先的位置，做Unpooling的时候就是直接把pixel的值复制四份填充到扩大后的matrix里即可

Deconvolution

实际上，Deconvolution就是convolution

这里以一维的卷积为例，假设输入是5维，过滤器(filter)的大小是3

卷积的过程就是每三个相邻的点通过过滤器生成一个新的点，如下图左侧所示

在你的想象中，去卷积的过程应该是每个点都生成三个点，不同的点对生成同一个点的贡献值相加；但实际上，这个过程就相当于在周围补0之后再次做卷积，如下图右侧所示，两个过程是等价的

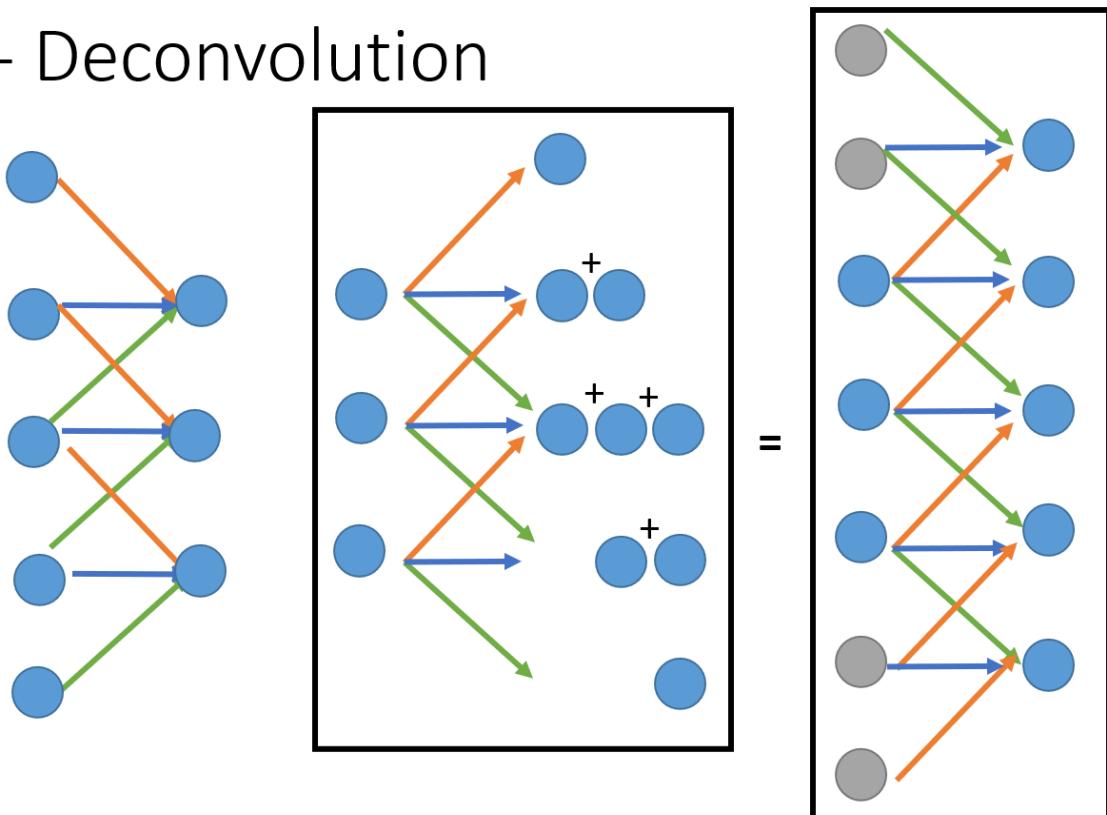
卷积和去卷积的过程中，不同点在于，去卷积需要补零且过滤器的weight与卷积是相反的：

- 在卷积过程中，依次是橙线、蓝线、绿线weight
- 在去卷积过程中，依次是绿线、蓝线、橙线weight

因此在实践中，做Deconvolution的时候直接对模型加卷积层即可

CNN - Deconvolution

Actually, deconvolution is convolution.



Seq2Seq Auto-encoder

在之前介绍的自编码器中，输入都是一个固定长度的vector，但类似文章、语音等信息实际上不应该单纯被表示为vector，那会丢失很多前后联系的信息。比如说语音（一段声音讯号有长有短），文章（你可能用bag-of-word变成一个vector，但是你会失去词汇和词汇之间的前后关系，是不好的）

Seq2Seq就是为了解决这个问题提出的，具体内容在RNN部分已经介绍

Generate

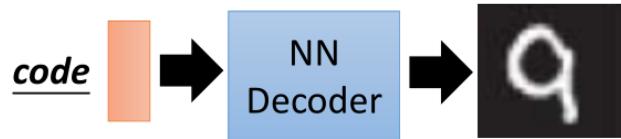
在用自编码器的时候，通常是获取Encoder之后的code作为降维结果，但实际上Decoder也是有作用的，我们可以拿它来生成新的东西

以MNIST为例，训练好Encoder之后，取出其中的Decoder，输入一个随机的code，就可以生成一张图像。

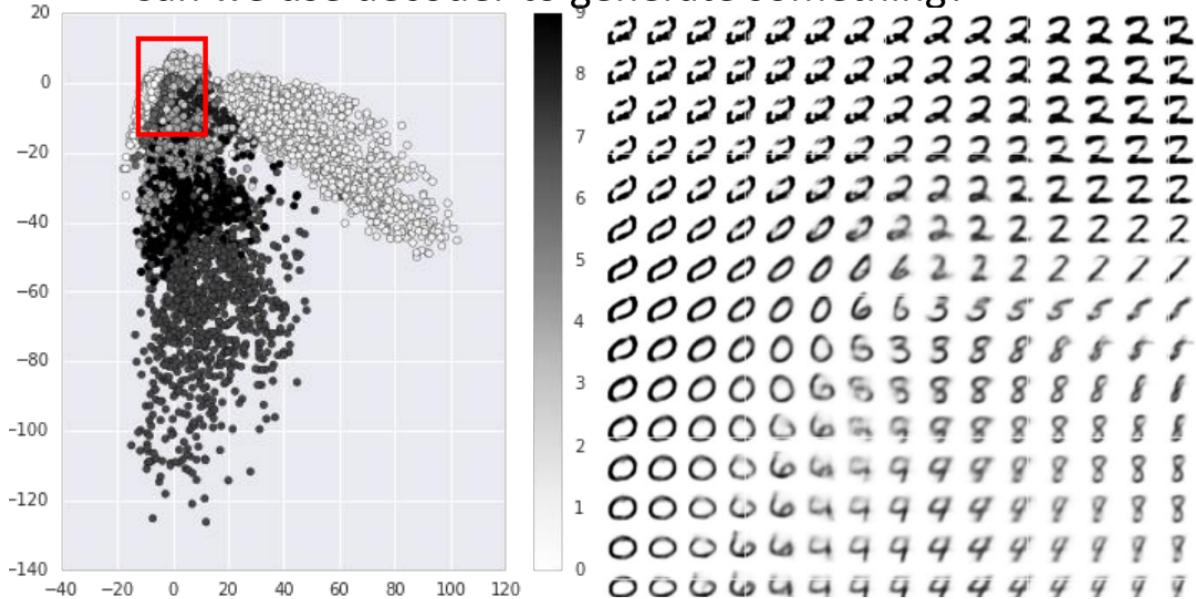
把每一张 28×28 维的image，通过hidden layer，把它project到2维，2维再通过一个hidden layer解回原来的image。在Encoder的部分，2维的vector画出来如下图左，不同颜色的点代表不同的数字。

然后在红色方框中，等间隔的挑选2维向量丢进Decoder中，就会生成许多数字的图像。这些2维的vector，它不见得是某个原来的image就是对应的vector。我们发现在红框内，等距离的做sample，得到的结果如下图右。在没有image对应的位置，画出的图像怪怪的。

Next



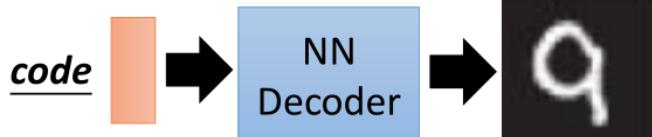
- Can we use decoder to generate something?



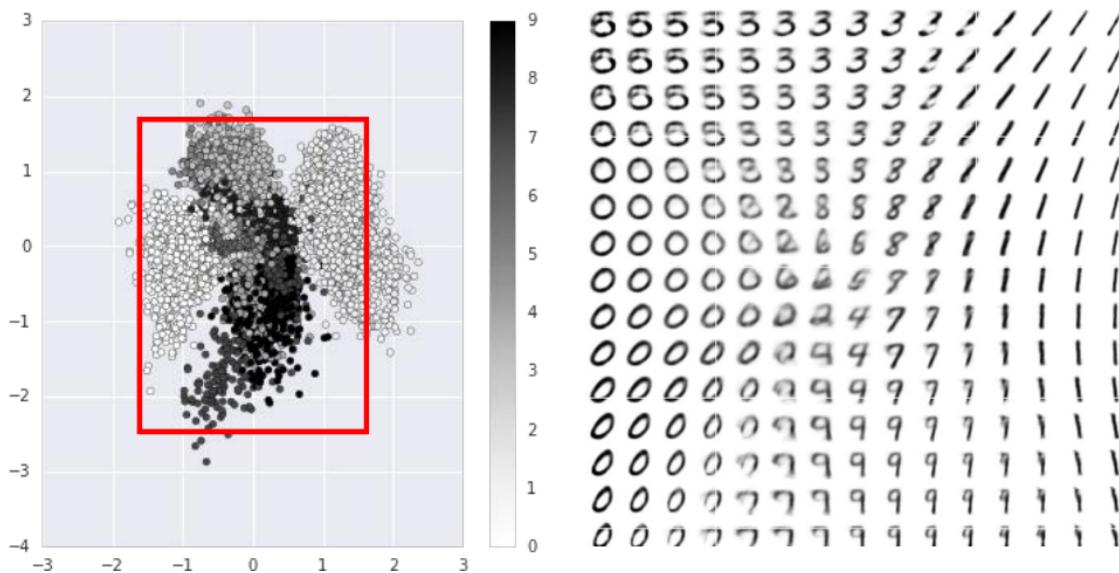
此外，我们还可以对code加L2 regularization，以限制code分布的范围集中在0附近，此时就可以直接以0为中心去随机采取样本点，再通过Decoder生成图像。

观察生成的数字图像，可以发现这两个dimension是有意义的，横轴的维度表示是否含有圆圈，纵轴的维度表示是否倾斜。

Next



- Can we use decoder to generate something?



More About Auto-encoder

Auto-encoder主要包含一个编码器（Encoder）和一个解码器（Decoder），通常它们使用的都是神经网络。Encoder接收一张图像（或是其他类型的数据，这里以图像为例）输出一个vector，它也可称为Embedding、Latent Representation或Latent code，它是关于输入图像的表示；然后将vector输入到Decoder中就可以得到重建后的图像，希望它和输入图像越接近越好，即最小化重建误差（reconstruction error），误差项通常使用的平方误差。

More than minimizing reconstruction error

What is good embedding?

An embedding should represent the object.

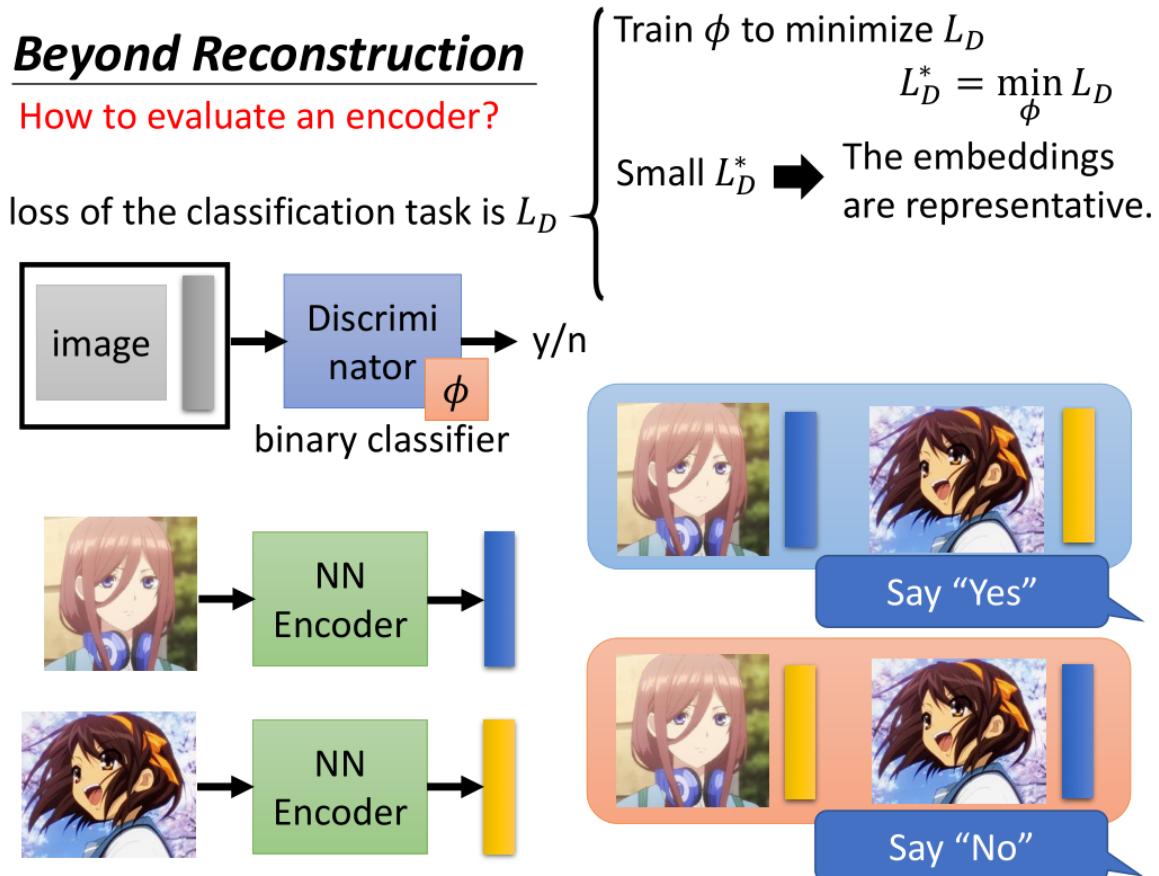
最直观的想法是它应该包含了关于输入的关键信息，从中我们就可以大致知道输入是什么样的。

Beyond Reconstruction

除了使用重建误差来驱动模型训练外，可以使用其他的方式来衡量Encoder是否学到了关于输入的重要表征吗？

假设我们现在有两类动漫人物的图像，一类是三九，一类是凉宫春日。如果将三九的图像丢给Encoder后，它就会给出一个蓝色的Embedding；如果Encoder接收的是凉宫春日的图像，它就会给出一个黄色的Embedding。那么除了Encoder之外，还有一个Discriminator（可以看作Binary Classifier），它接收图像和Embedding，然后给出一个结果表示它们是否是两两对应的。

如果是三九和蓝色的Embedding、凉宫春日和黄色的Embedding，那么Discriminator给出的就是YES；如果它们彼此交换一下，Discriminator给出的就应该是NO。

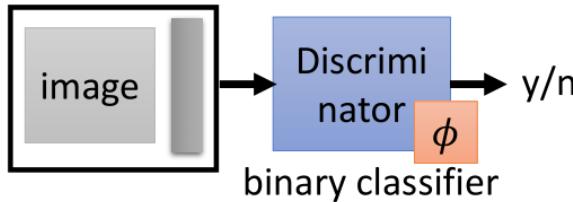


借助GAN的思想，我们用 ϕ 来表述Discriminator，希望通过训练最小化D的损失函数 $L_D^* = \arg \min_{\phi} L_D$ ，得到最小的损失值 L_D^* 。如果 L_D^* 的值比较小，就认为Encoder得到的Embedding很有代表性；相反 L_D^* 的值很大时，就认为得到的Embedding不具有代表性。

Beyond Reconstruction

How to evaluate an encoder?

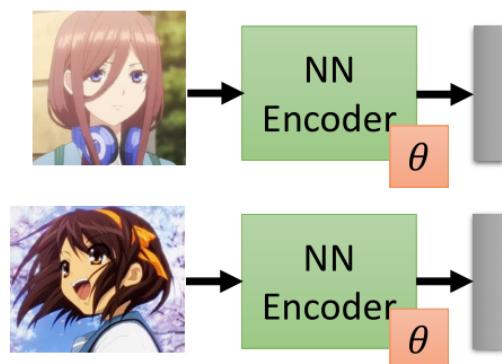
loss of the classification task is L_D



$\left\{ \begin{array}{l} \text{Train } \phi \text{ to minimize } L_D \\ L_D^* = \min_{\phi} L_D \\ \text{Small } L_D^* \rightarrow \text{The embeddings are representative.} \\ \text{Large } L_D^* \rightarrow \text{Not representative} \end{array} \right.$
 Train θ to minimize L_D^*

$$\theta^* = \arg \min_{\theta} L_D^*$$

$$= \arg \min_{\theta} \min_{\phi} L_D$$

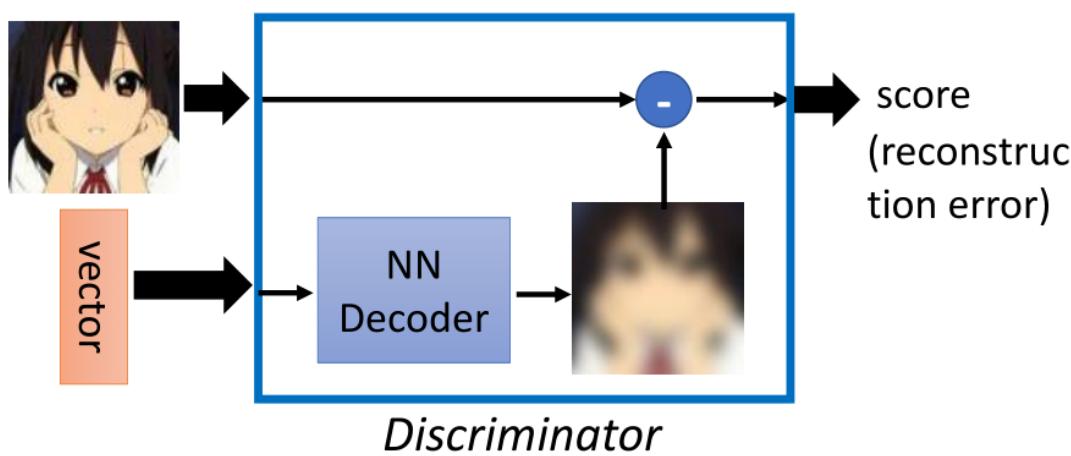
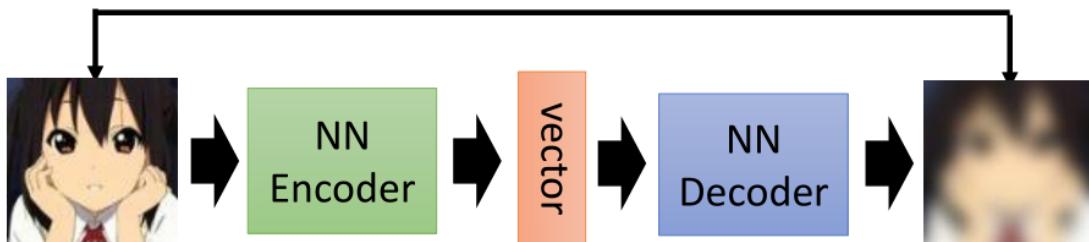


Train the encoder θ and discriminator ϕ to minimize L_D
 Deep InfoMax (DIM)
 (c.f. training encoder and decoder to minimize reconstruction error)

如果用 θ 表示Encoder, Train the encoder θ and discriminator ϕ to minimize L_D , 即
 $\theta^* = \arg \min_{\theta} \min_{\phi} L_D$, 这样的方法也称为Deep InfoMax(DIM)。这个和training encoder and decoder to minimize reconstruction error的思想其实是差不多的。

Typical auto-encoder is a special case。Discriminator接收一个图像和vector的组合, 然后给出一个判断它们是否是配对的分数。在Discriminator的内部先使用Decoder来解码vector生成一个重建的图像, 然后和输入图像相减, 得到score。只不过这种情况下不考虑negative, 只判断有多相似。

As close as possible



Sequential Data

Skip thought

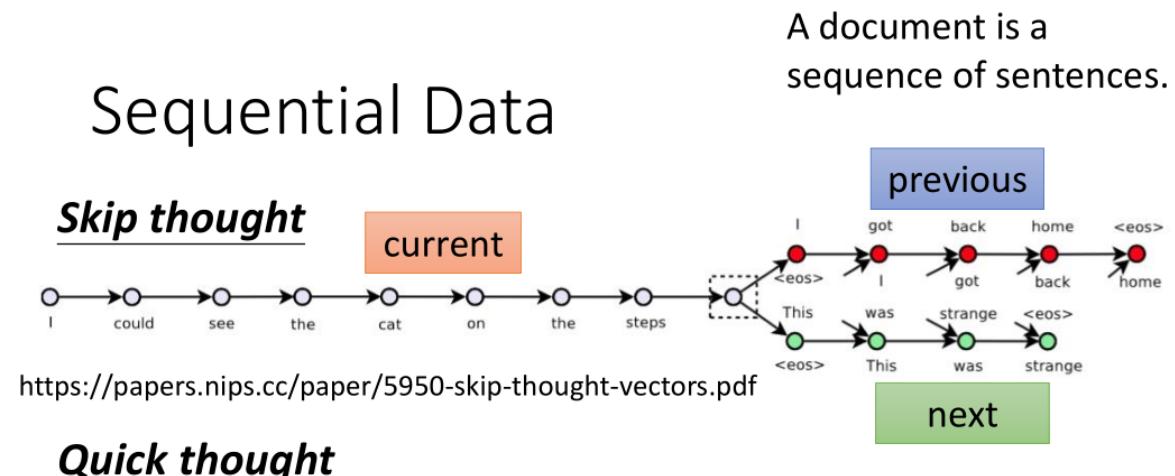
Skip thought就是根据中间句来预测上下句。模型在大量的文档数据上训练结束后，Encoder接收一个句子，然后给出输入句子的上一句和下一句是什么。这个模型训练过程和训练word embedding很像，因为训练word embedding的时候有这么一个原则，就是两个词的上下文很像的时候，这两个词的embedding就会很接近。换到句子的模型上，如果两个句子的上下文很像，那么这两个句子的embedding就应该很接近。

这个东西多少钱？答：10元；这个东西多贵？答：10元。发现答案一样，所以问句的embedding是很接近的。

Quick thought

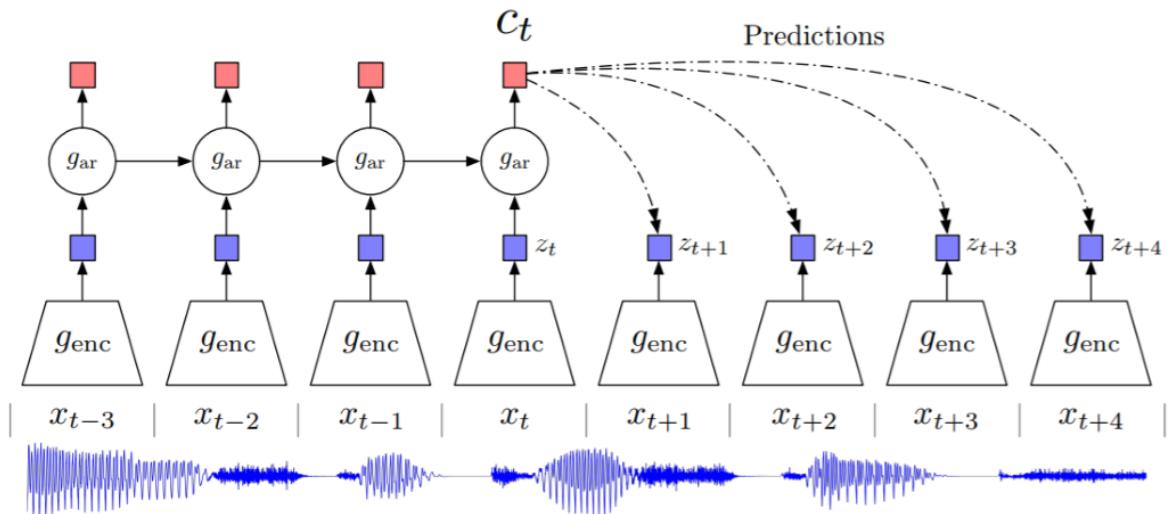
由于Skip thought要训练encoder和decoder，训练速度比较慢，因此有出现一个改进版本Quick thought，顾名思义就是训练速度上很快。

Quick thought不使用Decoder，而是使用一个辅助的分类器。它将当前的句子、当前句子的下一句和一些随机采样得到的句子分别送到Encoder中得到对应的Embedding，然后将它们丢给分类器。因为当前的句子的Embedding和它下一句的Embedding应该是越接近越好，而它和随机采样句子的Embedding应该差别越大越好，因此分类器应该可以判断出哪一个Embedding代表的是当前句子的下一句。



Contrastive Predictive Coding (CPC)

这个模型和Quick thought的思想是很像的，它接收一段序列数据，得到Embedding，然后用它预测接下来数据的Embedding。模型结构如下所示，具体内容可见原论文。

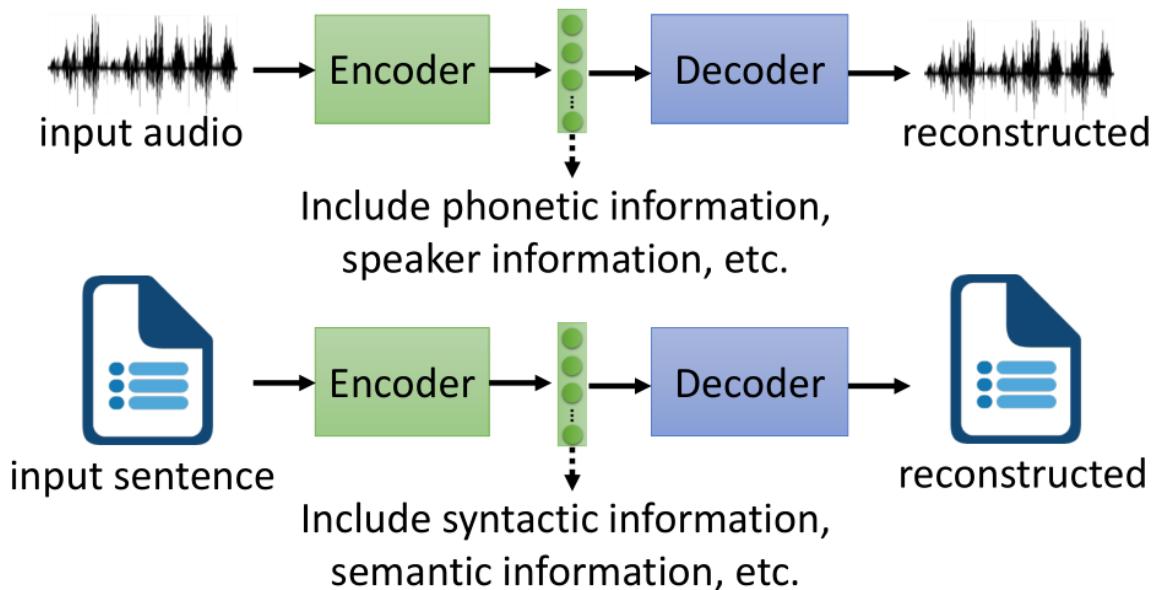


<https://arxiv.org/pdf/1807.03748.pdf>

More interpretable embedding

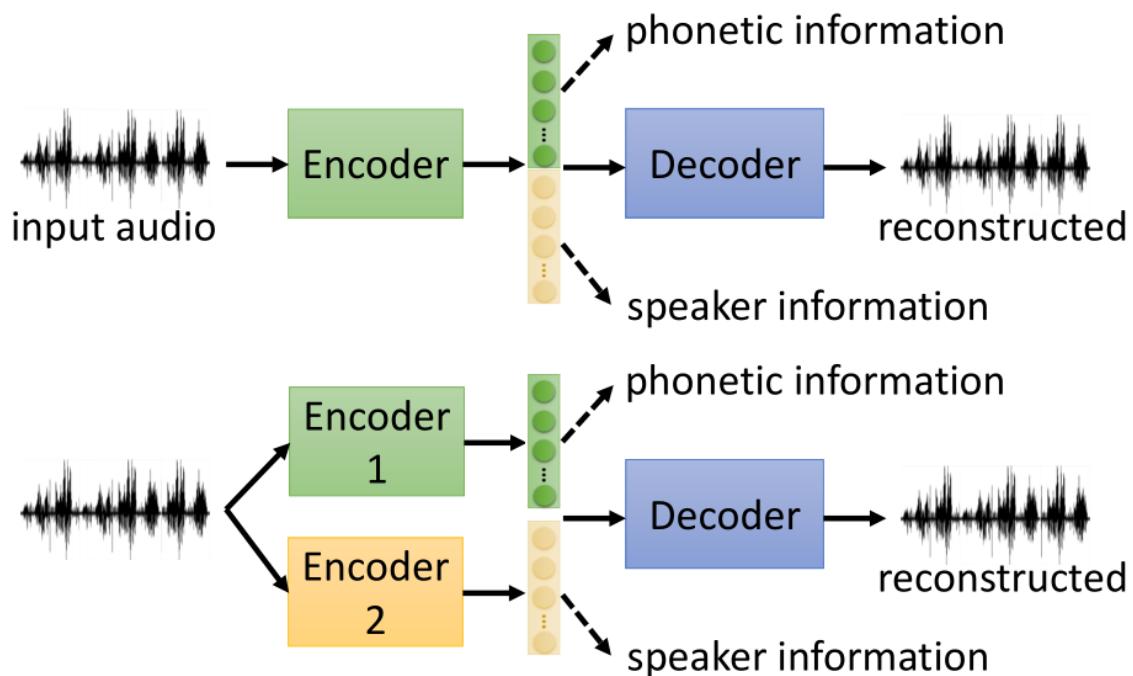
Feature Disentangle

An object contains multiple aspect information



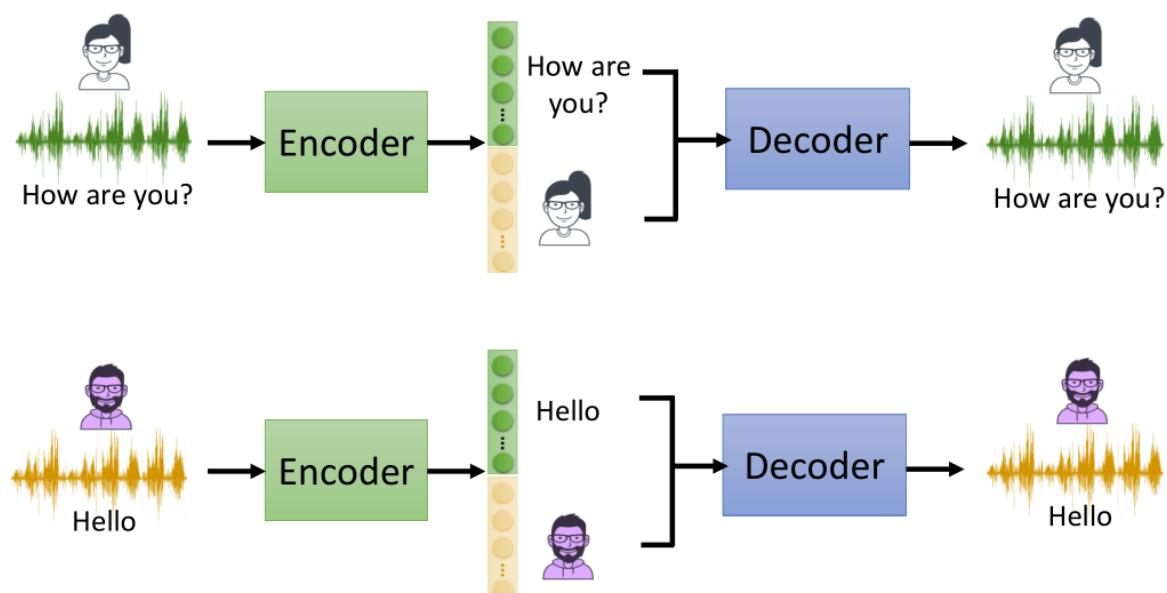
现在我们只用一个向量来表示一个object，我们是无法知道向量的哪些维度包含哪些信息，例如哪些维度包含内容信息，哪些包含讲话人信息等。也就是说这些信息是交织在一起的，我们希望模型可以帮我们把这些信息disentangle开来。

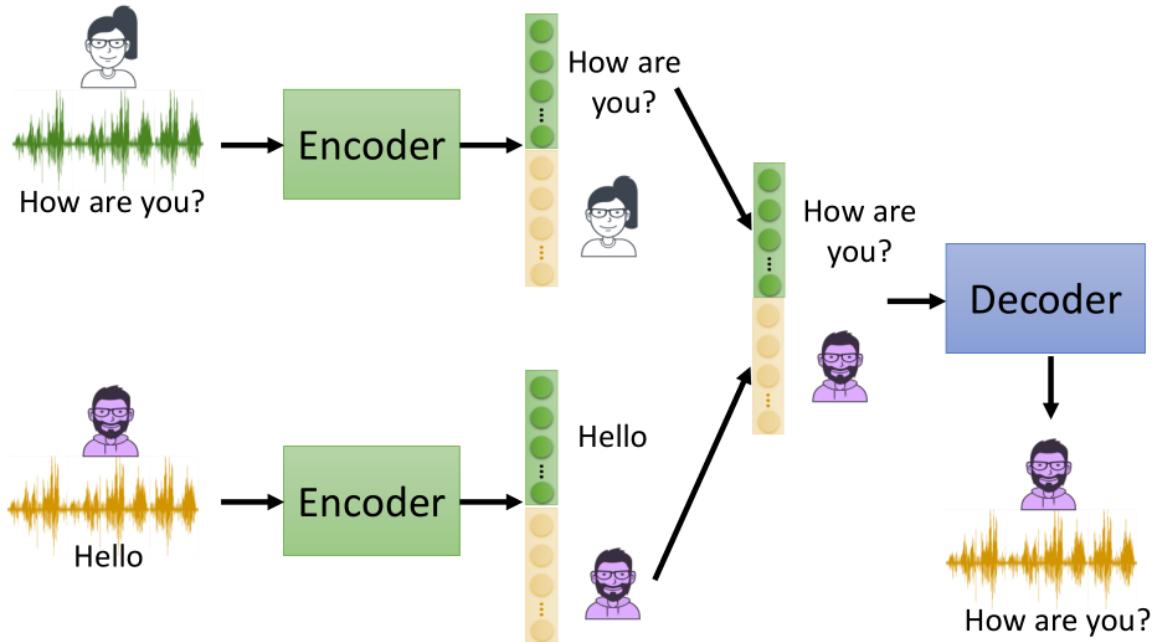
我们以声音讯号为例，假设通过Encoder得到的Embedding是一个100维的向量，它只包含内容和讲话者身份两种信息。我们希望经过不断的训练，它的前50维代表内容信息，后50维代表讲话者的身份信息。可以用1个Encoder，也可以训练两个encoder分别抽取不同内容，然后把两个部分拼接起来，才能还原原来的内容。



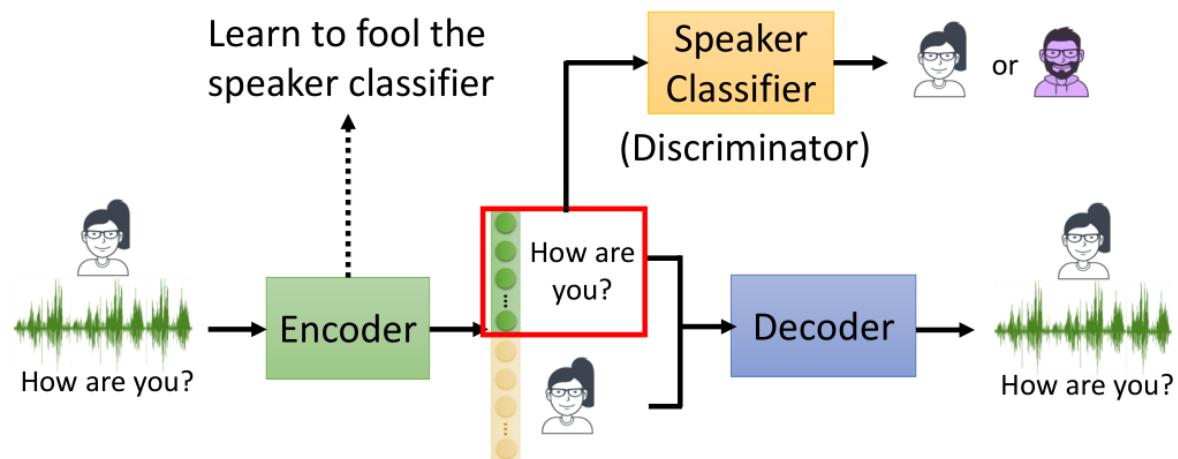
Voice Conversion

The same sentence has different impact when it is said by different people.





Adversarial Training



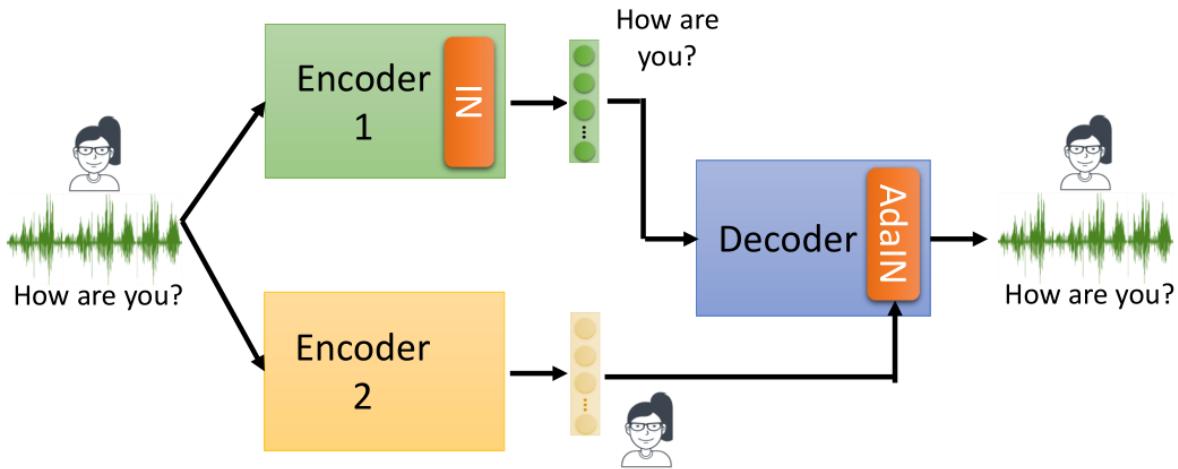
Speaker classifier and encoder are learned iteratively

一种方法就是使用GAN的思想，我们在Encoder-Decoder架构中引入一个Classifier，通过Embedding某个具体的部分判断讲话者身份，通过不断地训练，希望Encoder得到的Embedding可以骗过Classifier，就是要使得不能让Classifier分辨出语者的性别，那么那个具体的部分就不包含讲话者的信息。

在实作过程中，通常是利用GAN来完成这个过程，也就是把Encoder看做Generator，把Classifier看做Discriminator。Speaker classifier and encoder are learned iteratively.

Designed Network Architecture

使用两个Encoder来分别得到内容信息和讲话者身份信息的Embedding，在Encoder中使用instance normalization，然后将得到的两个Embedding结合起来送入Decoder重建输入数据，除了将两个Embedding直接组合起来的方式，还可以在Decoder中使用Adaptive instance normalization。



IN = instance normalization (remove global information)

AdaIN = adaptive instance normalization
(only influence global information)

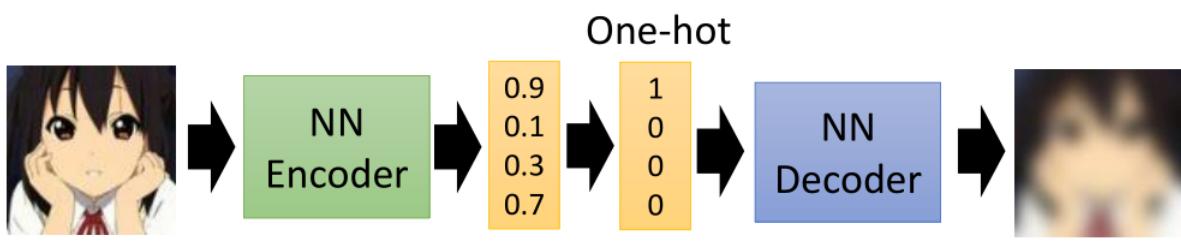
Discrete Representation

Easier to interpret or clustering

- Easier to interpret or clustering

non differentiable

<https://arxiv.org/pdf/1611.01144.pdf>



通常情况下，Encoder输出的Embedding都是连续值的向量，但如果可以将其转换为离散值的向量，例如one-hot向量或是binary向量，我们就可以更加方便的解读Embedding的哪一部分表示什么信息。

当然此时不能直接使用反向传播来训练模型，一种方式就是用强化学习来进行训练。

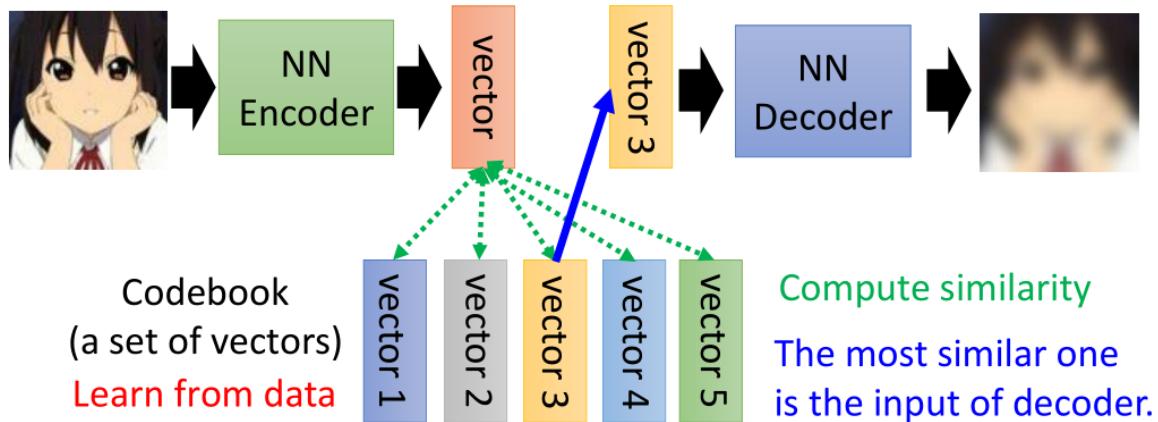
当然，上面两个离散向量的模型比较起来，个人觉得Binary模型要好，原因有两点：同样的类别Binary需要的参数量要比独热编码少，例如1024个类别Binary只需要10维即可，独热需要1024维；使用Binary模型可以处理在训练数据中未出现过的类别。

Vector Quantized Variational Auto-encoder (VQVAE)

基于这样的想法就出现了一种方法叫VQVAE，它引入了一个Codebook（内容是学出来的）。先用Encoder抽取为连续型的vector；再用vector与Codebook中的离散变量进行相似度计算，哪一个和输入更像，就将其丢给Decoder重建输入。

<https://arxiv.org/abs/1711.00937>

- Vector Quantized Variational Auto-encoder (VQVAE)



For speech, the codebook represents phonetic information

<https://arxiv.org/pdf/1901.08810.pdf>

上面的模型中，如果输入的是语音信号，那么不是Discrete的语者信息和噪音信息会被过滤掉，比较容易保留去辨识的内容和资讯。因为上面的Codebook中保存的是离散变量，而声音里面有关文字的内容信息是一个个的token，是容易用离散向量来表示的，其他不是Discrete的信息会被过滤掉。

Sequence as Embedding

seq2seq2seq auto-encoder.

Using a sequence of words as latent representation.

一篇文章经过encoder得到一串文字，然后这串文字再通过decoder还原回文章。

但是这个机器抽取出的sequence是看不懂的，是机器自己的暗号。

Sequence as Embedding

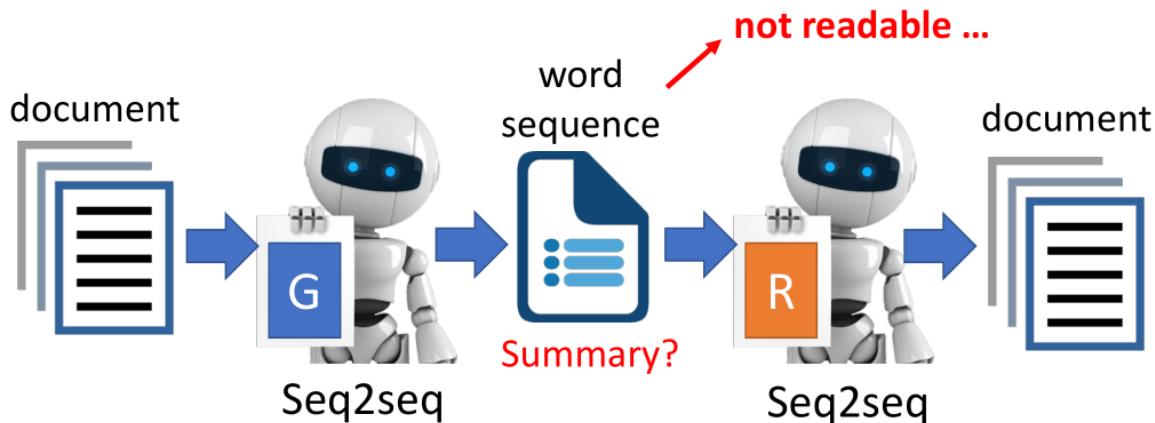
<https://arxiv.org/abs/1810.02851>

Only need a lot
of documents to
train the model

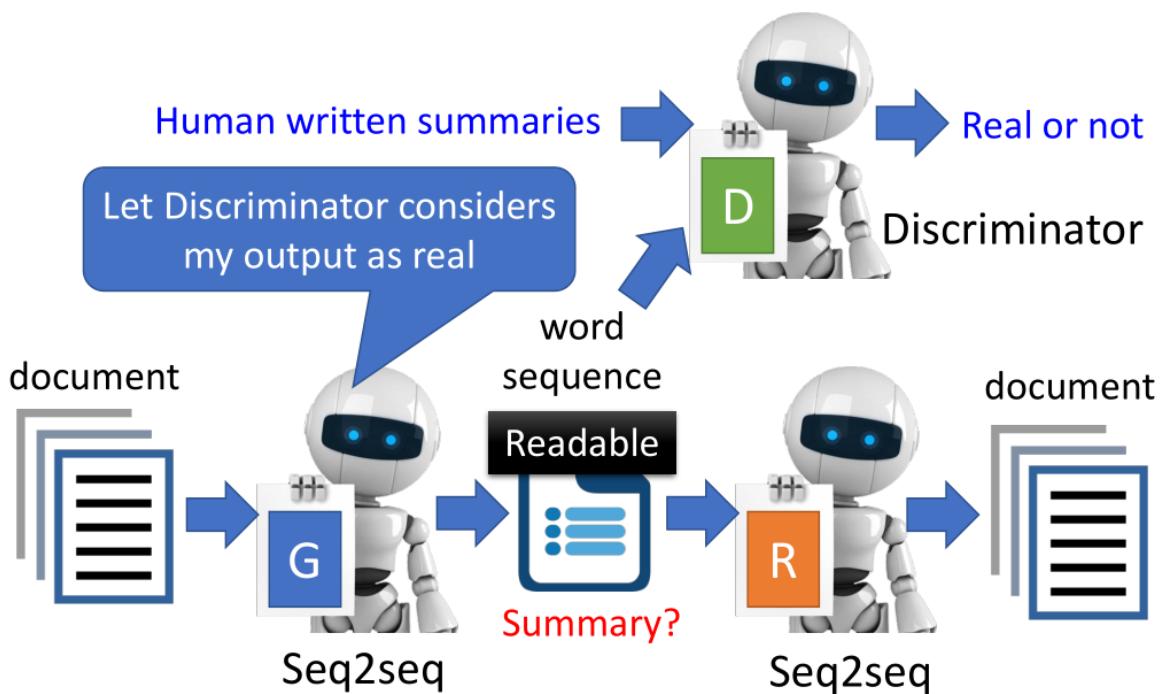


This is a seq2seq2seq auto-encoder.

Using a sequence of words as latent representation.

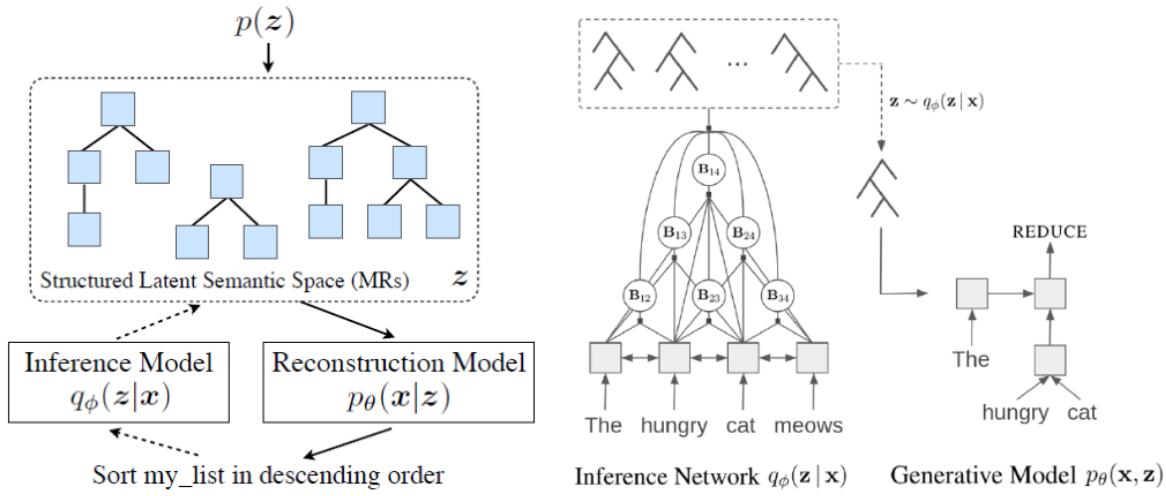


可以采用GAN的概念，训练一个Discriminator，判断是否是人写的句子。使得中间的seq可读。



不能微分，实际上用RL train encoder和decoder，loss当作reward。

Tree as Embedding



<https://arxiv.org/abs/1806.07832>

<https://arxiv.org/abs/1904.03746>

Concluding Remarks

More than minimizing reconstruction error

- Using Discriminator
- Sequential Data

More interpretable embedding

- Feature Disentangle
- Discrete and Structured

BERT

Representation of Word

1-of-N Encoding

最早的词表示方法。它就是one-hot encoding 没什么好说的，就是说如果词典中有N个词，就用N维向量表示每个词，向量中只有一个位置是1，其余位置都是0。但是这么做词汇之间的关联没有考虑。

Word Class

根据词的类型划分，但是这种方法还是太粗糙了，举举例说dog、cat和bird都是动物，它们应该是同类。但是动物之间也是有区别的，如dog和cat是哺乳类动物，和鸟类还是有些区别的。

Word Embedding

有点像是soft 的word class，我们用一个向量来表示一个单词，向量的每一个维度表示某种意思，相近的词汇距离较近，如cat和dog。

A word can have multiple senses

Have you paid that *money* to the **bank** yet ?

It is safest to deposit your *money* in the **bank**.

The victim was found lying dead on the *river bank*.

They stood on the *river bank* to fish.

The four **word tokens** have the same **word type**.

In typical word embedding, each word type has an embedding.

bank一词在上述前两个句子与后两个句子中的token是不一样的，但是type是一样的。也就是说同样的词有存在不用语义的情况，而词嵌入会为同一个词只唯一确定一个embedding。

那我们能不能标记一词多义的形式呢？可以尝试的解决方案是为**bank**这个词设置2个不同的embedding，但是确定一个词有几个词义是困难的，可以参看以下句子：

The hospital has its own blood **bank**.

The third sense or not?下个句子中的**bank**又有了不同的词义，这个词义可以看做一个新的词义，也可以看做与“银行”词义相同，因此机械地确定一个词有几个词义是困难的，因为很多词的意义是微妙的。

此时我们需要根据上下文来计算对应单词的embedding结果，这种技术称之为**Contextualized Word Embedding**（语境词嵌入）。

Embeddings from Language Model (ELMO)

ELMO是一个RNN-based Language Model，训练的方法就是找一大堆的句子，也不需要做标注，然后做上图所示的训练。

RNN-based Language Model 的训练过程就是不断学习预测下一个单词是什么。举例来说，你要训练模型输出“潮水退了就知道谁没穿裤子”，你教model，如果看到一个开始符号，就输出潮水，再给它潮水，就输出退了，再给它退了，就输出就.....学完以后你就有Contextualized Word Embedding，我们可以把RNN 的hidden layer 拿出来作为Embedding。为什么说这个hidden layer 做Embedding 就是Contextualized 呢，因为RNN中每个输出都是结合前面所有的输入做出的。

我们做了正反双向的训练，最终的word embedding 是把正向的RNN 得到的token embedding 和反向RNN 得到的token embedding 接起来作为最终的Contextualized Word Embedding。

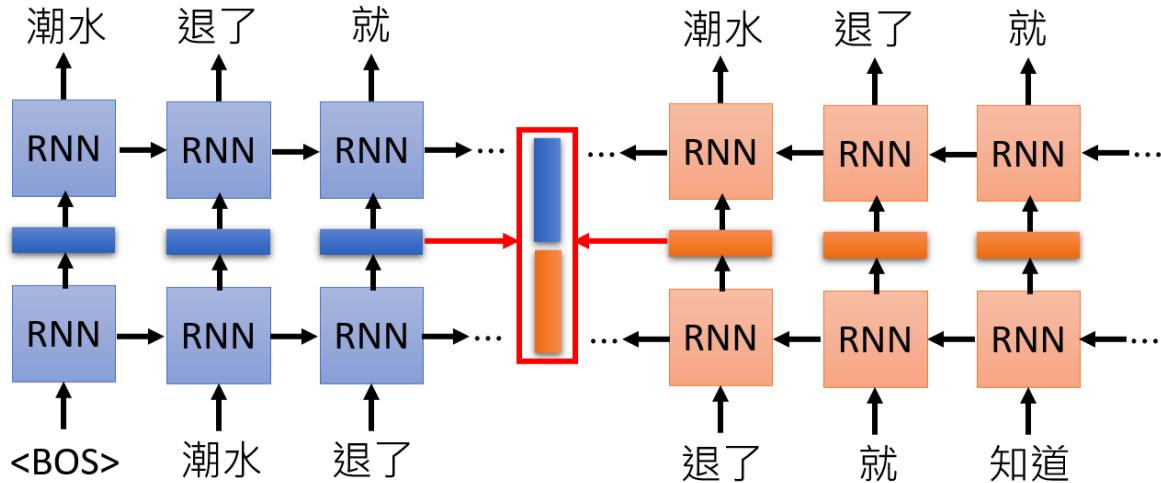
Embeddings from Language

Model (ELMO)

<https://arxiv.org/abs/1802.05365>

- RNN-based language models (trained from lots of sentences)

e.g. given “潮水 退了 就 知道 谁 没穿 裤子”

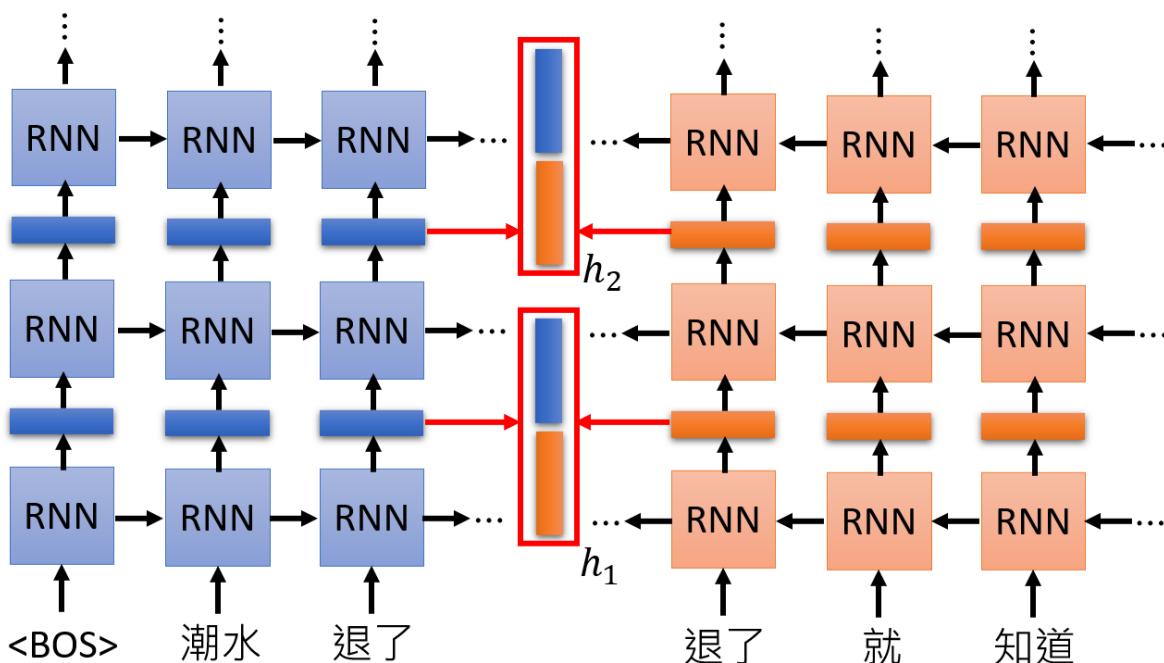


该过程也是可以deep的，如下图的网络结构，每一个隐藏层都会输出一个词的embedding，它们全部都会被使用到。

ELMO

Each layer in deep LSTM can generate a latent representation.

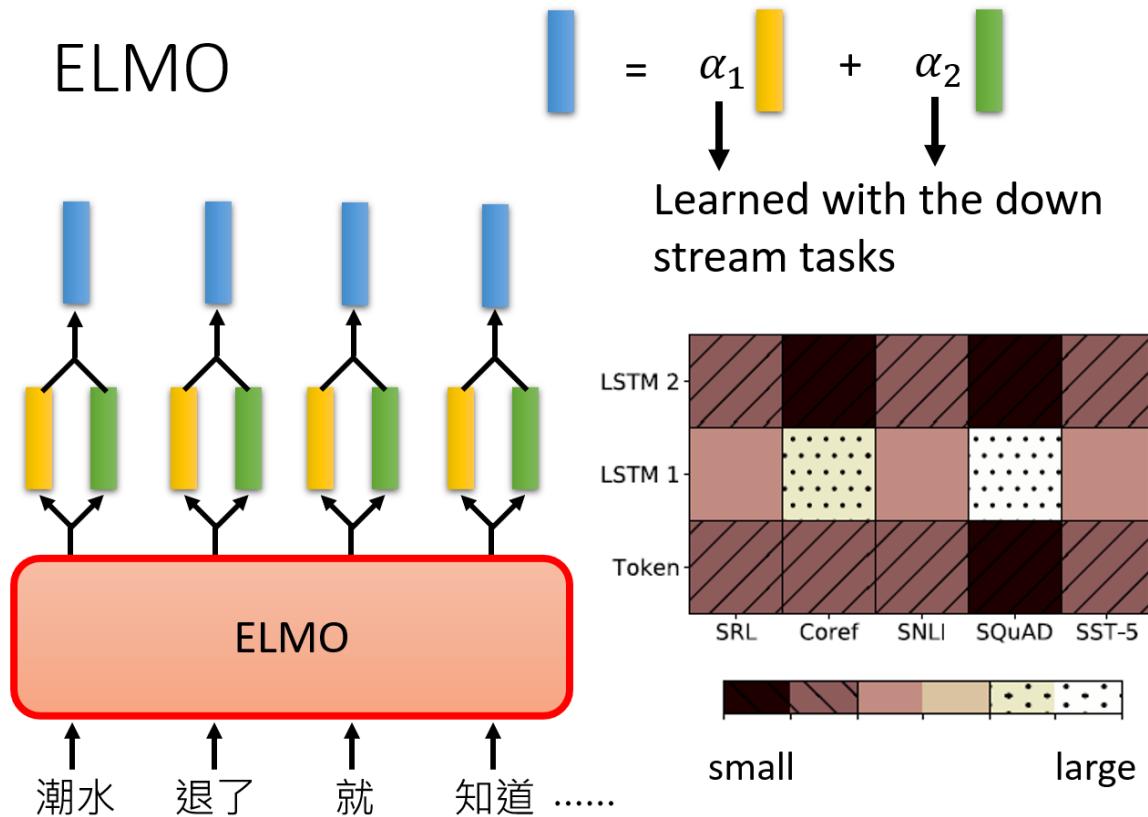
Which one should we use???



ELMO会将每个词输出多个embedding，这里我们假设LSTM叠两层。ELMO会用做weighted sum，weight是根据你做的下游任务训练出来的，下游任务就是说用ELMO做SRL (Sematic Role Labeling 语义角色标注)、Coref (Coreference resolution 共指解析)、SNLI (Stanford Natural Language Inference 自然语言推理)、SQuAD (Stanford Question Answering Dataset)、SST-5 (5分类情感分析数据集) 等等。

具体来说，你要先train好ELMO，得到每个token 对应的多个embedding，然后决定你要做什么task，然后在下游task 的model 中学习weight α_1 和 α_2 的值。

原始ELMO的paper 中给出了图中的实验结果，Token是说没有做Contextualized Embedding之前的原始向量，LSTM-1、LSTM-2是EMLO的两层得到的embedding，然后根据下游5个task 学出来的weight 的比重情况。我们可以看出Coref 和SQuAD 这两个任务比较看重LSTM-1抽出的embedding，而其他task 都比较平均的看了三个输入。



Bidirectional Encoder Representations from Transformers (BERT)

BERT = Encoder of Transformer

BERT其实就是Transformer的Encoder，可以从大量没有注释的文本中学习

BERT会输入一些词的序列然后输出每个词的一个embedding。

需要注意，实际操作中如果训练中文，应该以中文的字作为输入。因为中文的词语很难穷举，但字的穷举相对容易，常用汉字约4000个左右，而词的数量非常多，使用词作为输入可能导致输入向量维度非常高 (one-hot)，所以也许使用字作为基本单位更好。

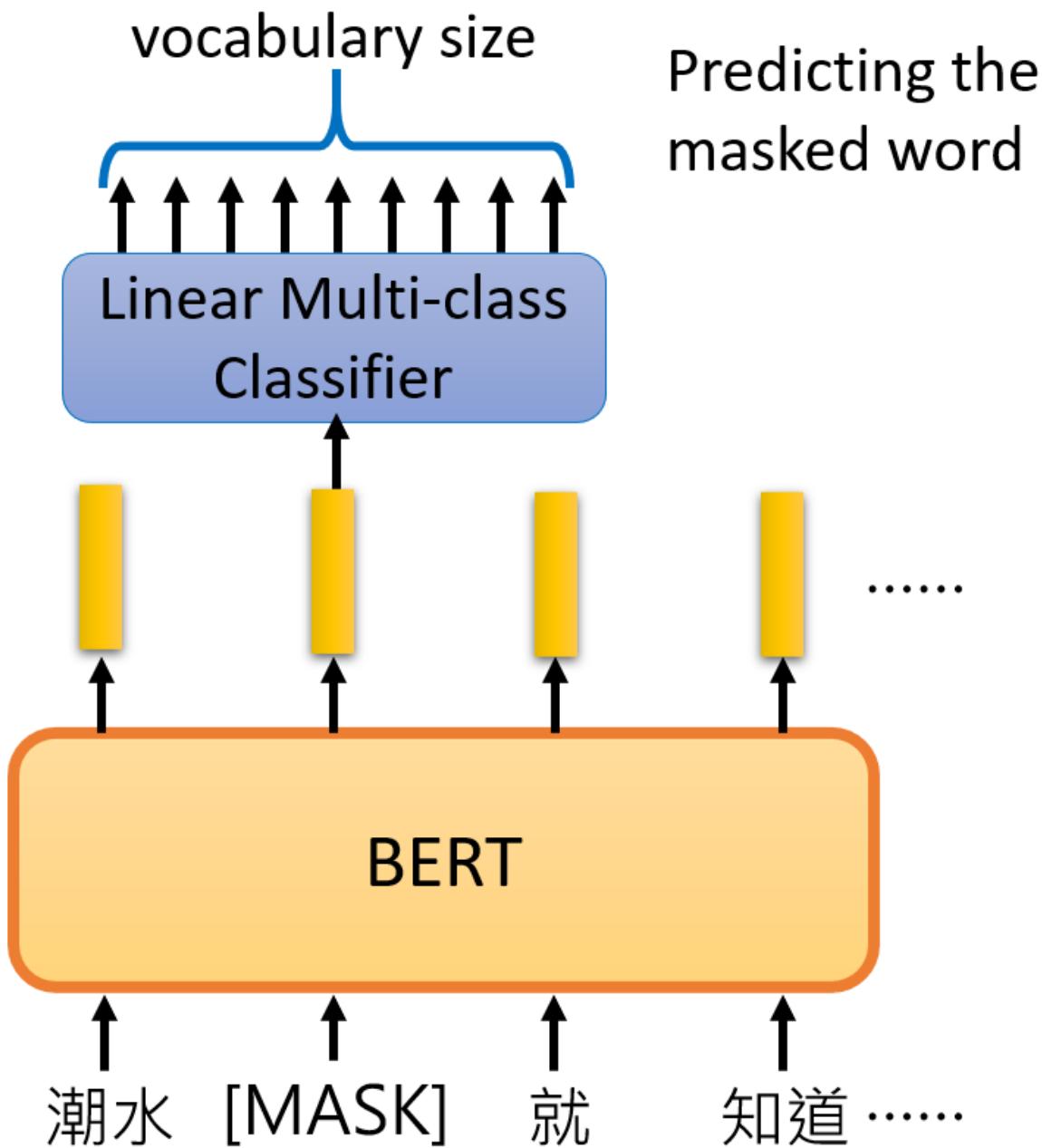
Training of BERT

paper上提及的BERT的训练方法有两种，**Masked LM** 和**Next Sentence Prediction**。

Masked LM

Masked LM这种训练方式指的是在词序列中每个词汇有15%的机率被一个特殊的token[MASK]遮盖掉，得到被遮盖掉的词对应输出的embedding后，使用一个Linear Multi-class Classifier来预测被遮盖掉的词是哪一个，由于Linear Multi-class Classifier的能力很弱，所以训练得到的embedding会是一个非常好的表示。

BERT的embedding是什么样子的呢？如果两个词填在同一个地方没有违和感，那它们就有类似的embedding，代表他们的语义是类似的。



Next Sentence Prediction

给BERT两个句子，然后判断这两个句子是不是应该接在一起。

具体做法是，[SEP]符号告诉BERT句子交接的地方在哪里，[CLS]这个符号通常放在句子开头，将其通过BERT得到的embedding输入到简单的Linear Binary Classifier中，Linear Binary Classifier判断当前这两个句子是不是应该接在一起。

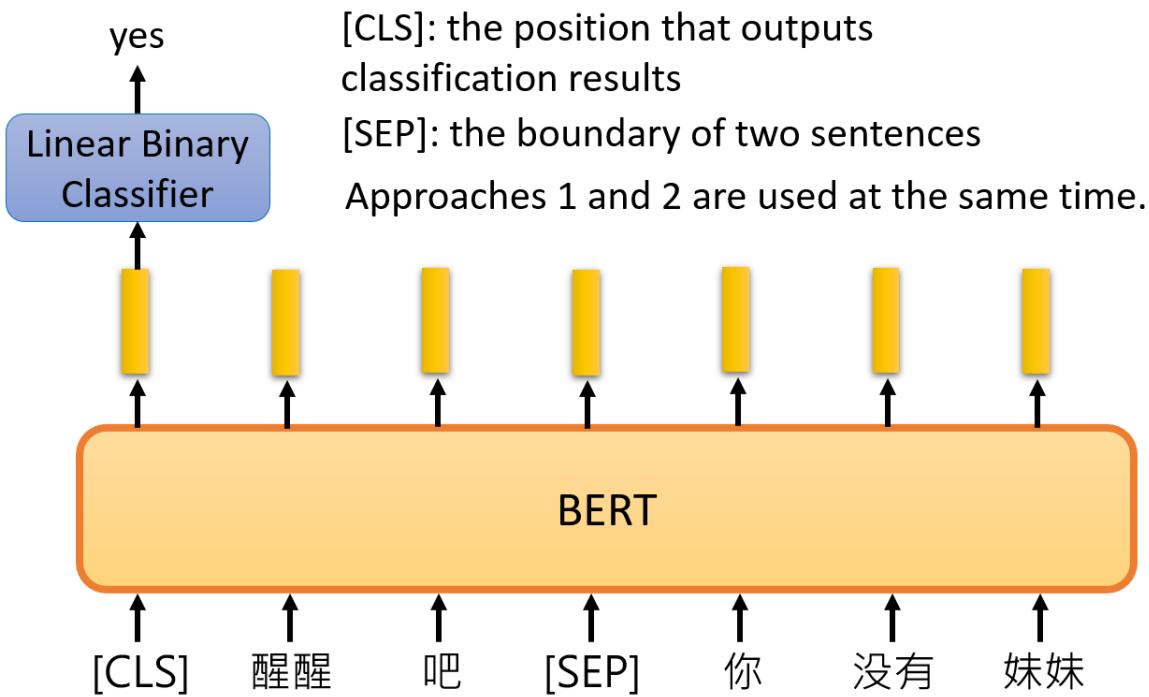
你可能会疑惑[CLS]难道不应该放在句末，让BERT看完整个句子再做判断吗？

BERT里面一般用的是Transformer的Encoder，也就是说它做的是self-attention，self-attention layer不受位置的影响，它会看完整个句子，所以一个token放在句子的开头或者结尾是没有差别的。

上述两个方法中，Linear classifier 是和BERT一起训练的。

两个方法在文献上是同时使用的，让BERT的输出去解这两个任务的时候会得到最好的训练效果。

Approach 2: Next Sentence Prediction



How to use BERT

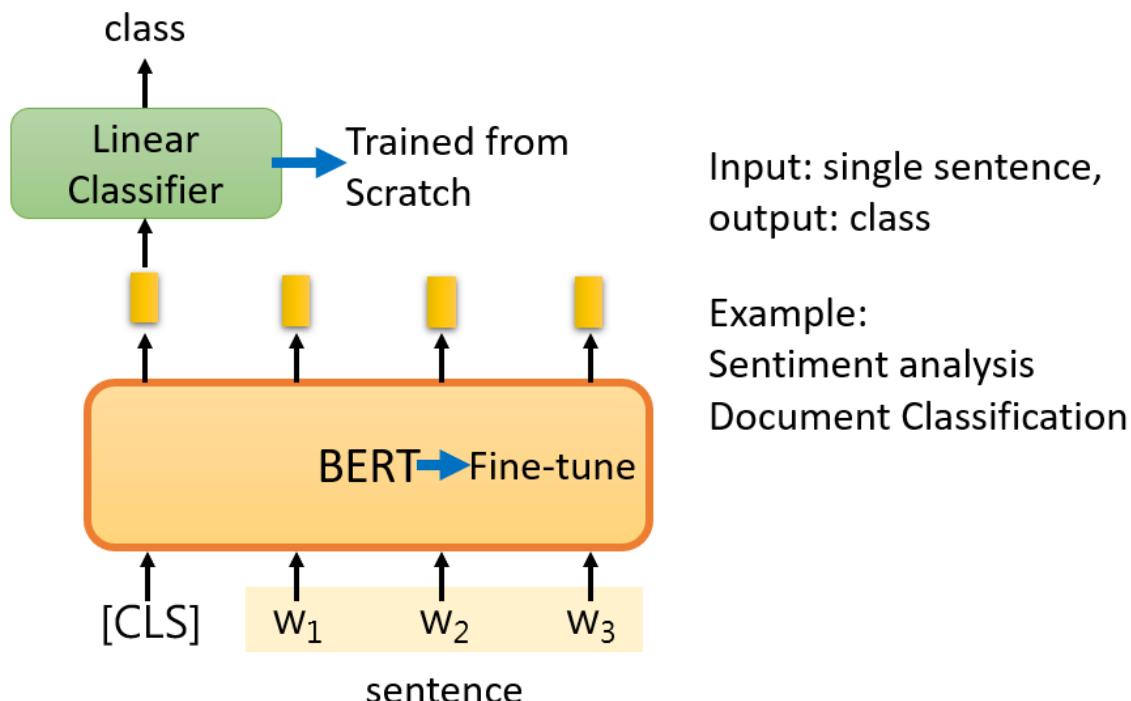
你可以把BERT当作一个抽embedding 的工具，抽出embedding 以后去做别的task。

但是在BERT的paper中是把BERT和down stream task 一起做训练。

Case 1

输入一个sentence 输出一个class，有代表性的任务有情感分析，文章分类等。

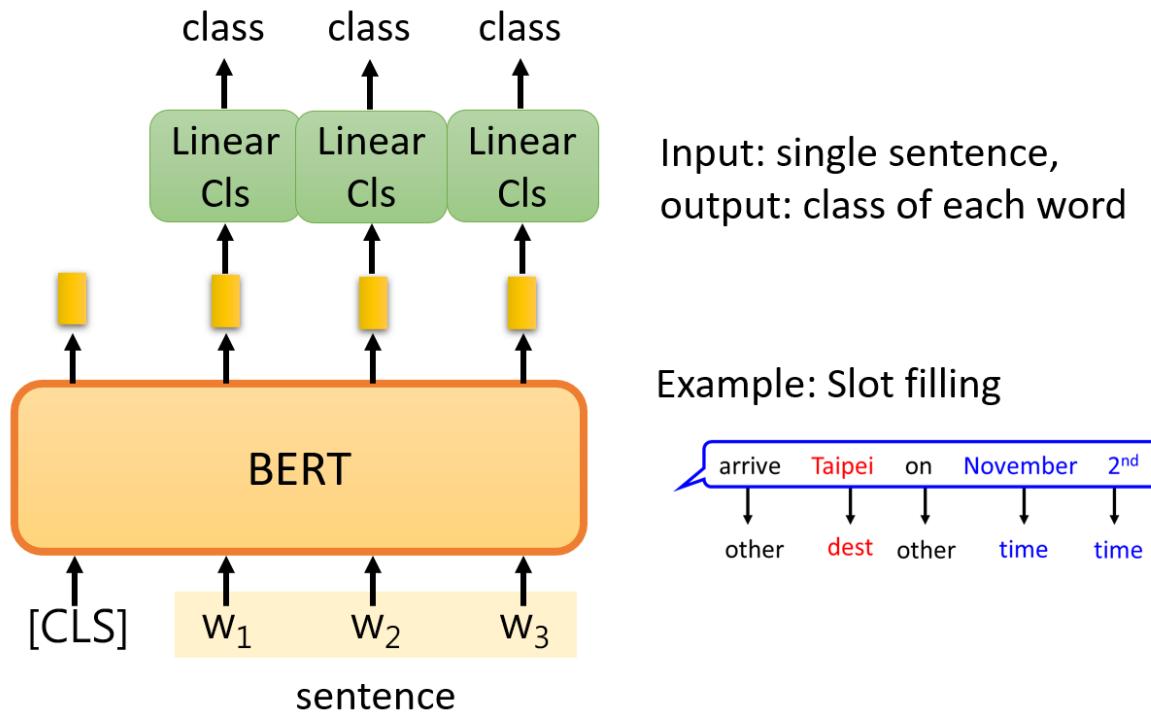
以句子情感分析为例，你找一堆带有情感标签的句子，丢给BERT，再句子开头设一个判断情感的符号[CLS]，把这个符号通过BERT的输出丢给一个线性分类器做情感分类。线性分类器是随机初始化参数，再用你的训练资料train 的，这个过程中也可以对BRET进行fine-tune，也可以fix住BRET的参数。



Case 2

输入：一个句子；输出：词的类别；例子：槽位填充

将句子输入到BERT，将每个token对应输出的embedding输入到一个线性分类器中进行分类。线性分类器要从头开始训练，BERT的参数只需要微调即可。

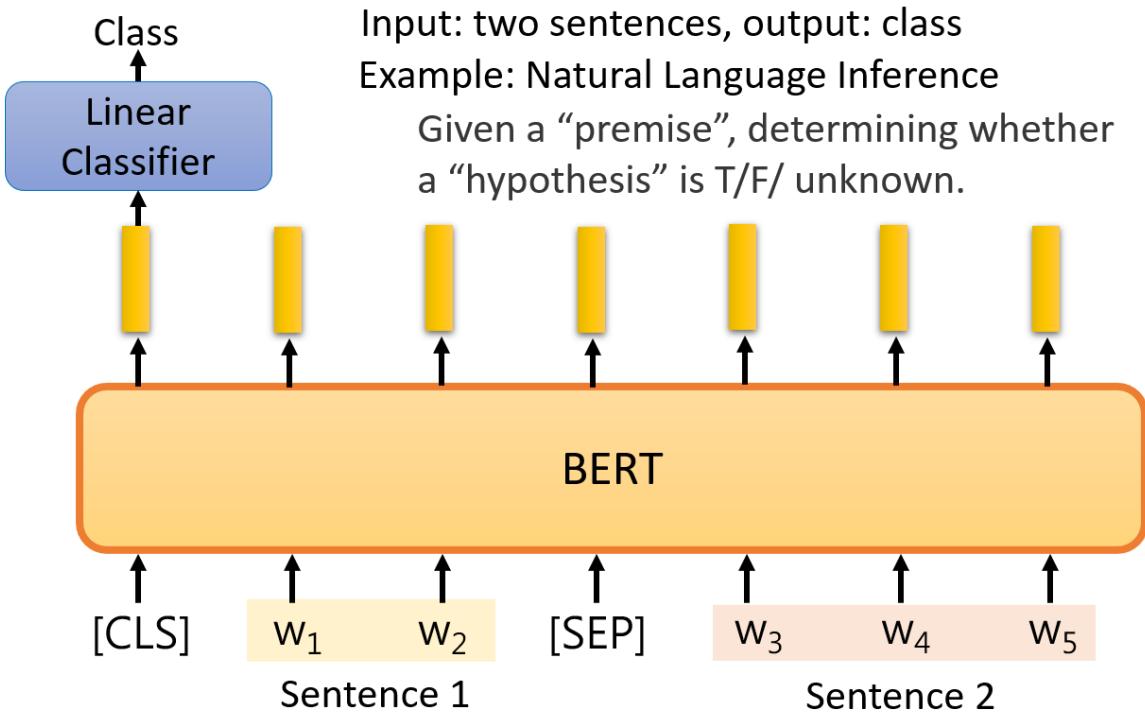


Case 3

输入：两个句子；输出：类别；例子：自然语言推理

可以将两个句子输入到BERT，两个句子之间添加一个token[SEP]，这两个句子分别是premise和hypothesis，第一个token设置为[CLS]表示句子的分类。

将第一个token对应输出的embedding输入到一个线性分类器中进行分类，分类结果代表在该假设下该推断是true (entailment), false (contradiction), or undetermined (neutral)。线性分类器要从头开始训练，BERT的参数只需要微调即可。



Case 4

BERT还可以用来做Extraction-based Question Answering，也就是阅读理解，如下图所示，给出一篇文章然后提问一个问题，BERT就会给出答案，前提是答案在文中出现。

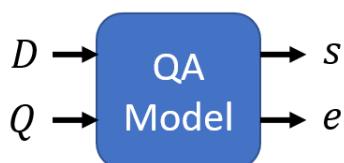
模型输入Document D 有N个单词，Query Q有M个单词，模型输出答案在文中的起始位置和结束位置： s 和 e 。举例来说，图中第一个问题的答案是gravity，是Document 中第17个单词；第三个问题的答案是within a cloud，是Document 中第77到第79个单词。

怎么用BERT解这个问题呢？

- Extraction-based Question Answering (QA) (E.g. SQuAD)

Document: $D = \{d_1, d_2, \dots, d_N\}$

Query: $Q = \{q_1, q_2, \dots, q_M\}$



output: two integers (s, e)

Answer: $A = \{d_s, \dots, d_e\}$

In meteorology, precipitation is any product of the condensation of 17 atmospheric water vapor that falls under gravity. The main forms of precipitation include drizzle, rain, sleet, snow, graupel and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals within a cloud. Short, intense periods of rain 77 79 atterations are called "showers".

What causes precipitation to fall?

gravity $s = 17, e = 17$

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?

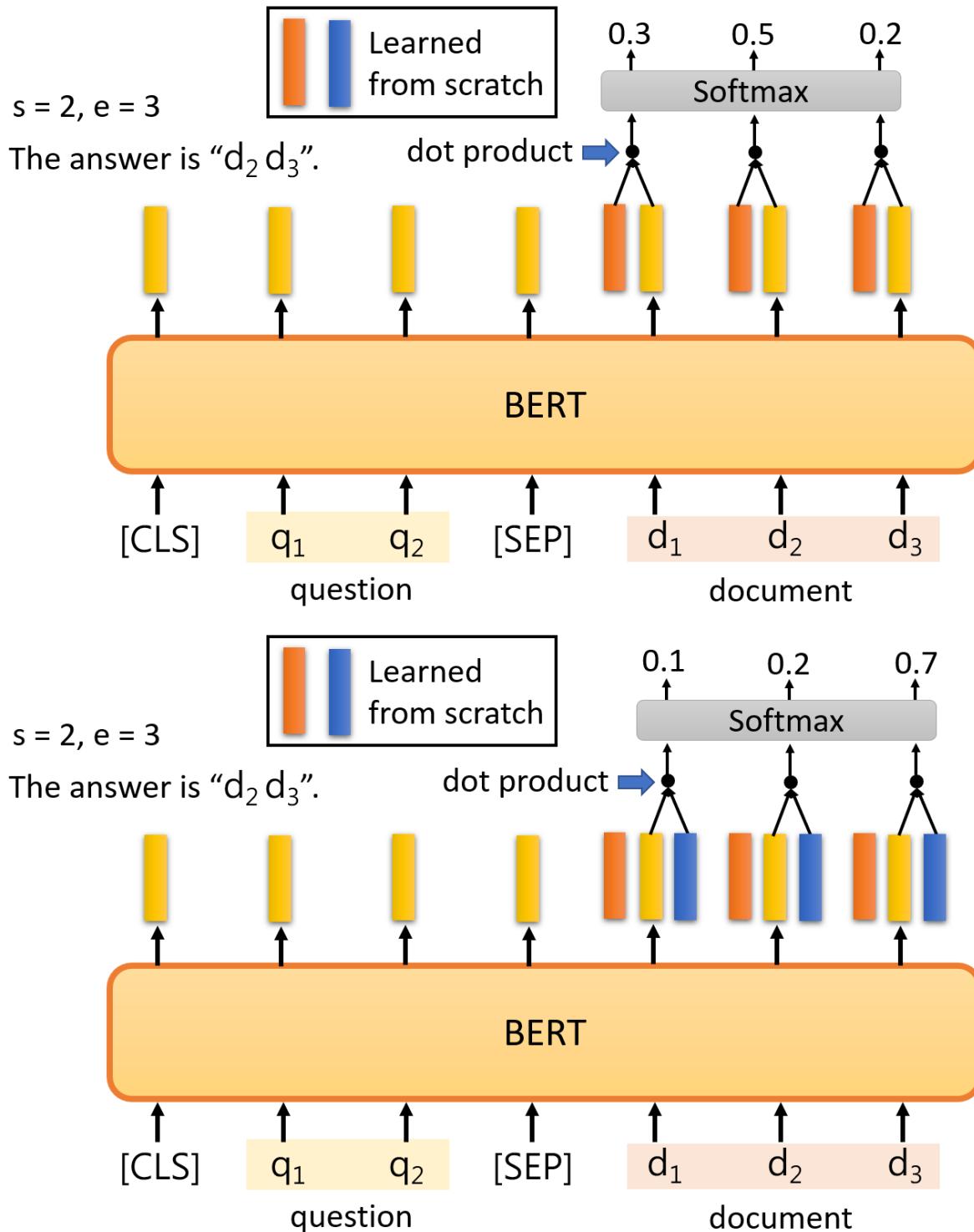
graupel

Where do water droplets collide with ice crystals to form precipitation?

within a cloud $s = 77, e = 79$

通过BERT后，Document 中每个词都会有一个向量表示，然后你再去learn 两个向量，得到图中红色和蓝色向量，这两个向量的维度和BERT的输出向量相同，红色向量和Document 中的词汇做点积得到一堆数值，把这些数值做softmax 最大值的位置就是 s ，同样的蓝色的向量做相同的运算，得到 e ：如果 e 落在 s 的前面，有可能就是无法回答的问题。

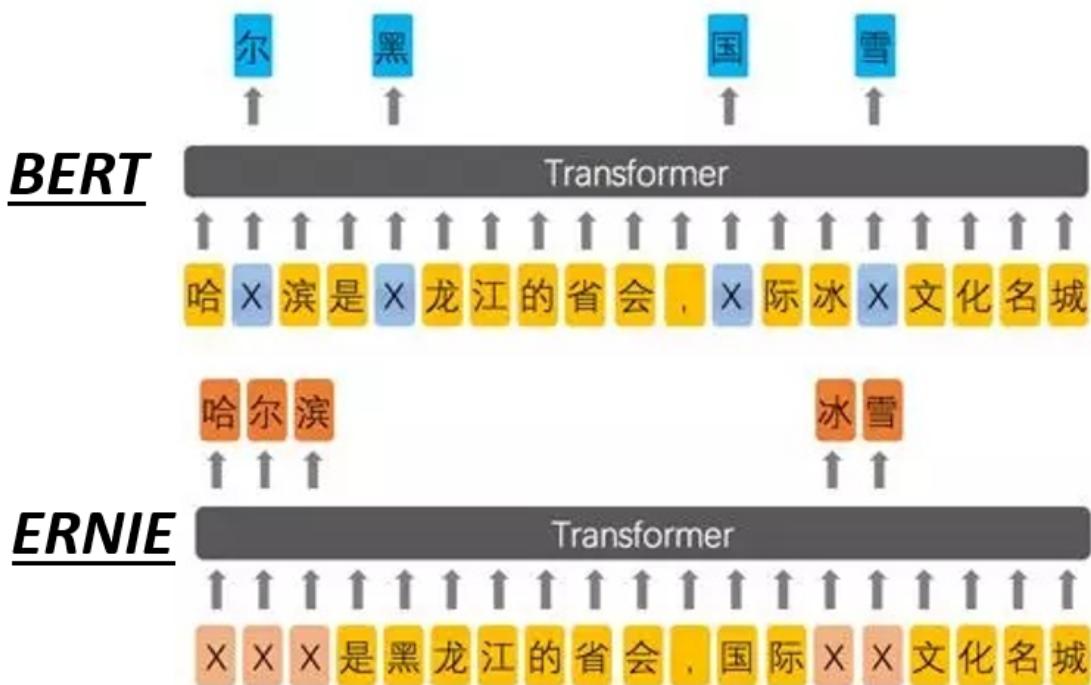
这个训练方法，你需要label很多data，每个问题的答案都需要给出在文中的位置。两个向量是从头开始学出来的，BERT只要fine-tune就好。



Enhanced Representation through Knowledge Integration (ERNIE)

ERNIE是类似BERT的模型，是专门为中文设计的，如果使用BERT的第一种训练方式时，一次只会盖掉一个字，对于BERT来说是非常好猜到的，因此ERNIE会一次盖掉中文的一个词。

- Designed for Chinese



What does BERT learn?

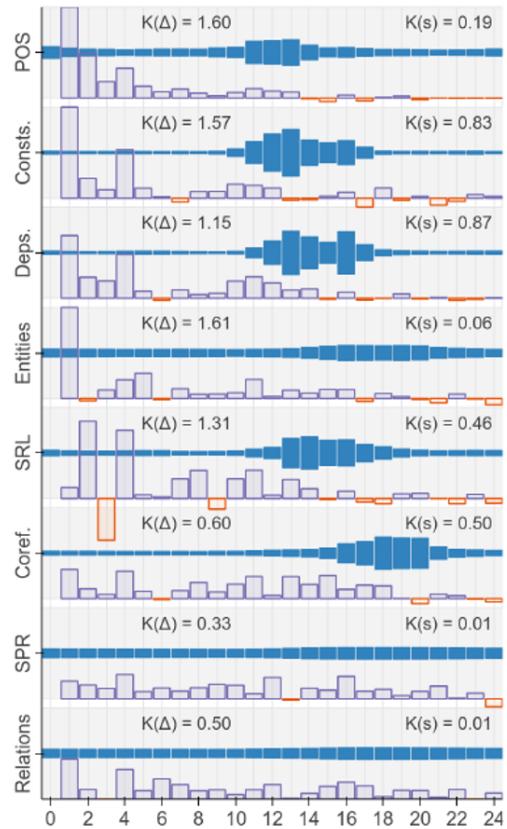
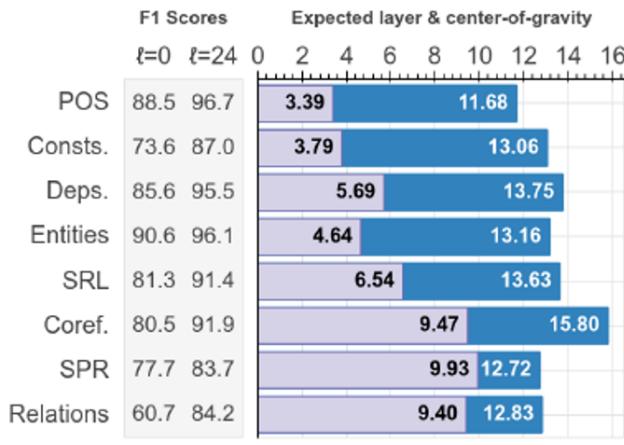
思考一下BERT每一层都在做什么，列出两个reference 给大家做参考。

假如我们的BERT有24层，单纯用BERT做embedding，用得到的词向量做下游任务。down stream task有POS、Consts等等，实验把BERT的每一层的Contextualized Embedding 抽出来做weighted sum，然后通过下游任务learn出weight，看最后learn出的weight的情况，就可以知道这个任务更需要那些层的vector。

图中右侧蓝色的柱状图，代表通过不同任务learn出的BERT各层的weight，POS是做词性标注任务，会更依赖11-13层；Coref是做分析代词指代，会更依赖BERT高层的向量（17-20层）；而SRL语义角色标注就比较平均地依赖各层抽出的信息。前三个任务都是文法相关的，因此更需要前面几层。若任务更困难，通常会需要比较深层抽出的embedding。

What does BERT learn?

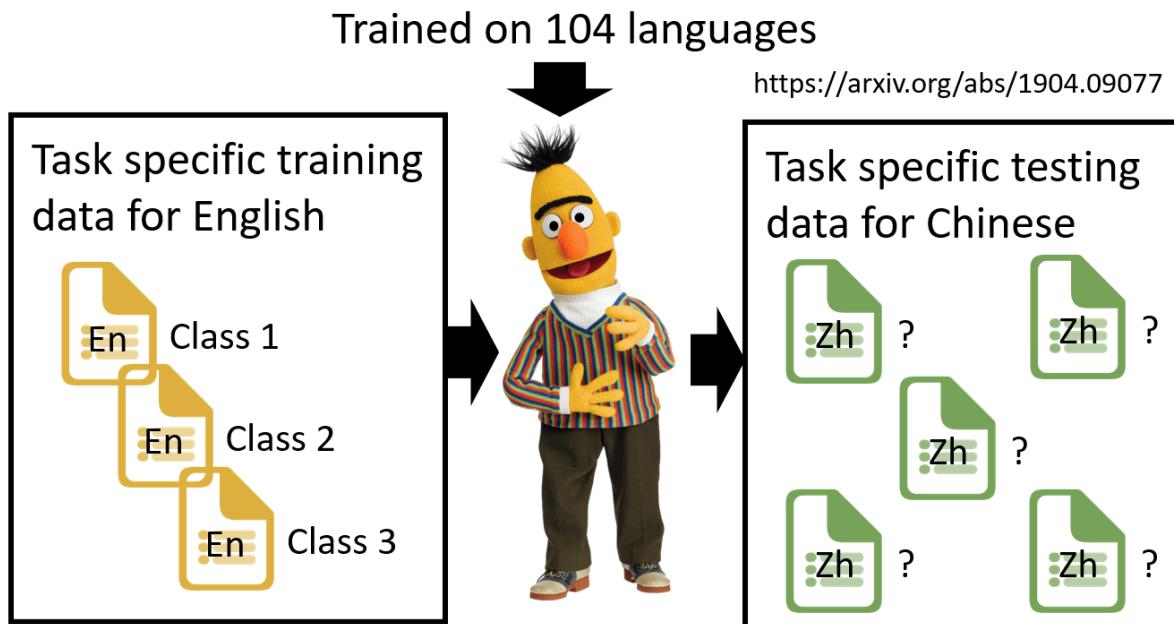
<https://arxiv.org/abs/1905.05950>
<https://openreview.net/pdf?id=SJzSgnRcKX>



Multilingual BERT

用104种语言的文本资料给BERT学习，虽然BERT没看过这些语言之间的翻译，但是它看过104种语言的文本资料以后，它似乎自动学会了不同语言之间的对应关系。

所以，如果你现在要用这个预训练好的BERT去做文章分类，你只要给他英文文章分类的label data set，它学完之后，竟然可以直接去做中文文章的分类。



Generative Pre-Training (GPT)

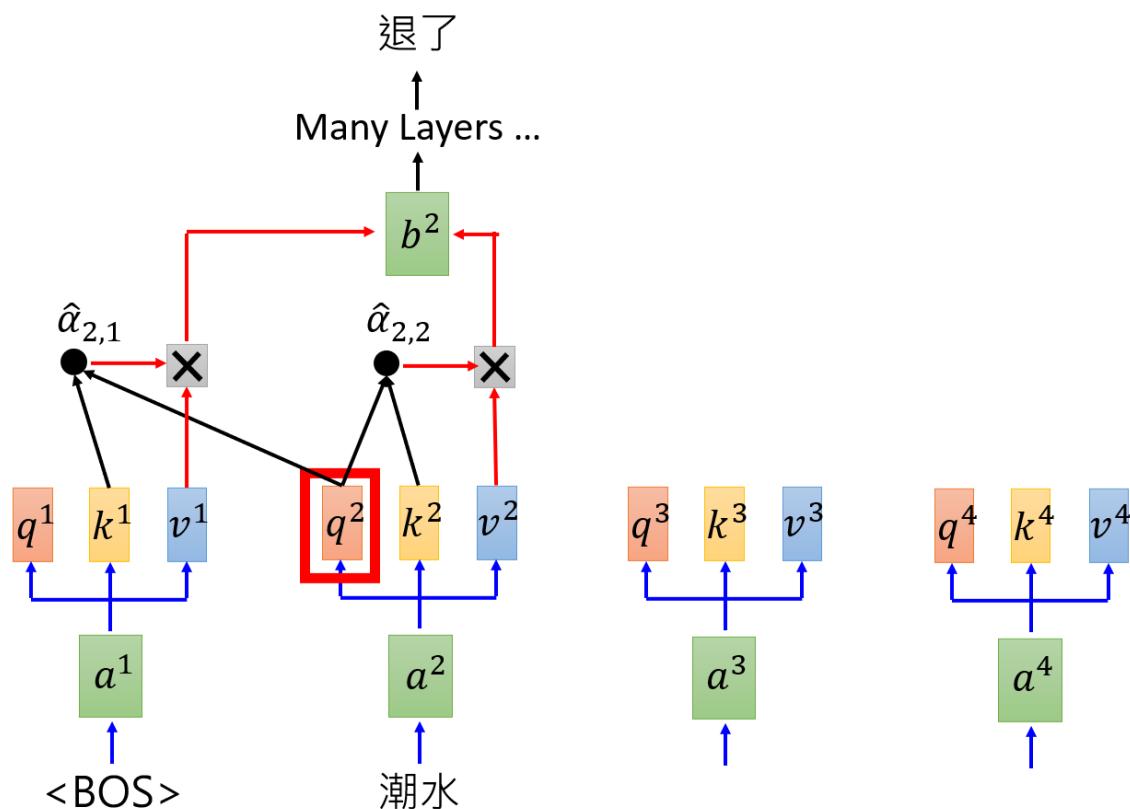
GPT-2是OpenAI 做的，OpenAI 用GPT-2 做了一个续写故事的例子，他们给机器看第一段，后面都是机器脑补出来的。机器生成的段落中提到了独角兽和安第斯山，所以现在都拿独角兽和安第斯山来隐喻GPT。

OpenAI担心GPT-2最大的模型过于强大，可能会被用来产生假新闻这种事上，所以只发布了GPT-2的小模型。

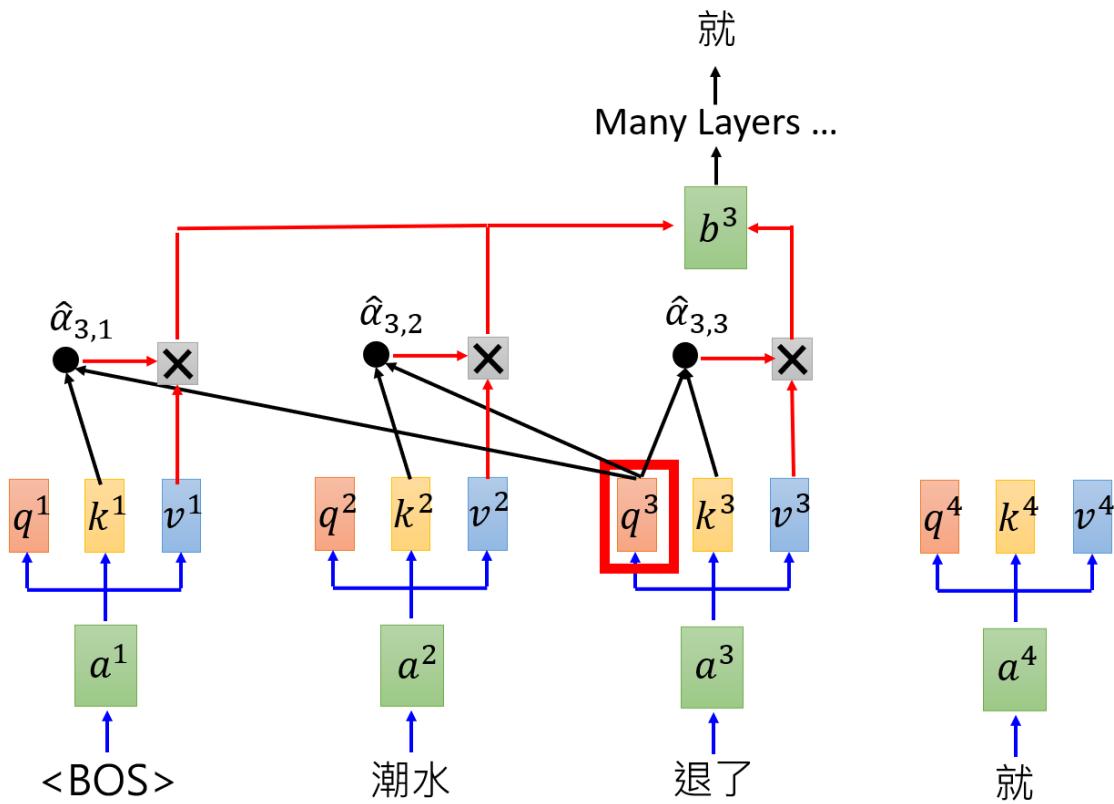
有人用GPT-2的公开模型做了一个在线[demo](#)。

我们上面说BERT是Transformer 的Encoder， GPT其实是Transformer 的Decoder。

GPT和一般的Language Model 做的事情一样，就是你给他一些词汇，它预测接下来的词汇。举例来说，如上图所示，把“潮水”的 q 拿出来做self-attention，然后做softmax 产生 $\hat{\alpha}$ ，再分别和 v 做相乘求和得到 b ，self-attention 可以有很多层（ b 是vector，上面还可以再接self-attention layer），通过很多层以后要预测“退了”这个词汇。



预测出“退了”以后，把“退了”拿下来，做同样的计算，预测“就”这个词汇，如此往复。



Zero-shot Learning?

GPT-2是一个巨大的预训练模型，它可以在没有更多训练资料的情况下做以下任务：

- **Reading Comprehension**

BERT也可以做Reading Comprehension，但是BERT需要新的训练资料train 线性分类器，对BERT本身进行微调。而GPT可以在没有训练资料的情况下做这个任务。

给GPT-2一段文章，给出一个问题，再写一个A:，他就会尝试做出回答。下图是GPT-2在CoQA上的结果，最大的GPT-2可以和DrQA达到相同的效果，不要忘了GPT-2在这个任务上是zero-shot learning，从来没有人教过它做QA。

- **Summarization**

给出一段文章加一个too long don't read 的缩写"TL;DR:" 就会尝试总结这段文字。

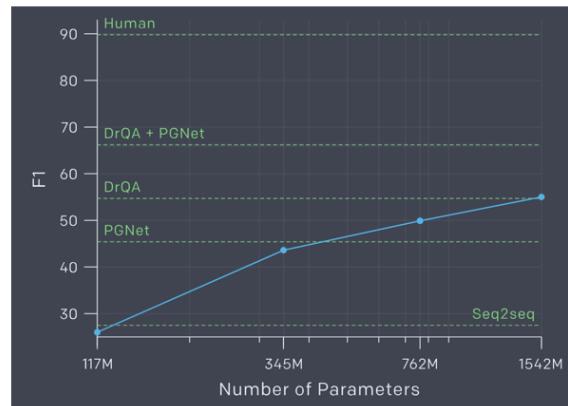
- **Translation**

以上图所示的形式给出一段英文=对应的法语，这样的例子，然后机器就知道要给出第三句英文的法语翻译。

其实后两个任务效果其实不是很好，Summarization就像是随机生成的句子一样。

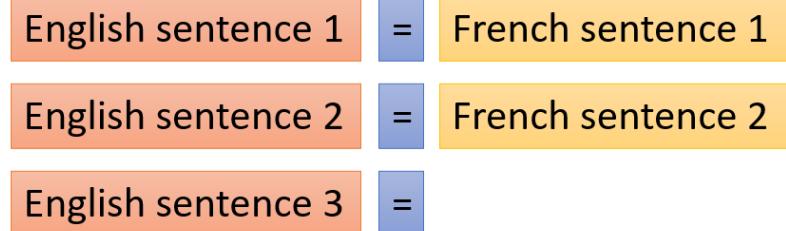
- **Reading Comprehension**

$d_1, d_2, \dots, d_N,$
 "Q:", $q_1, q_2, \dots, q_N,$
 "A:"



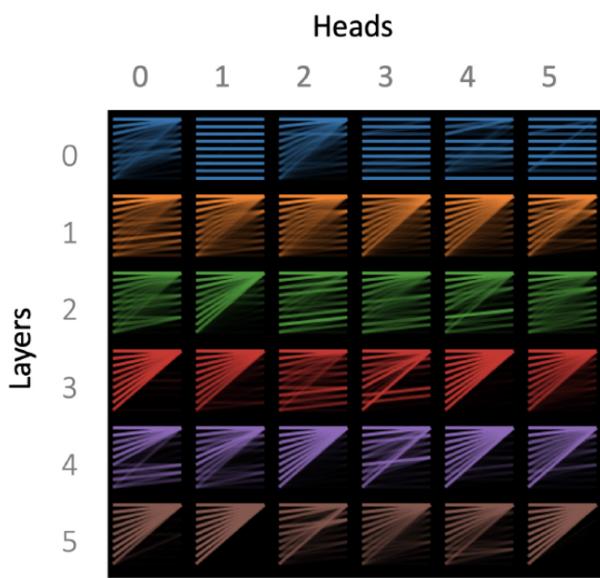
- **Summarization** $d_1, d_2, \dots, d_N, \text{"TL;DR:"}$

- **Translation**

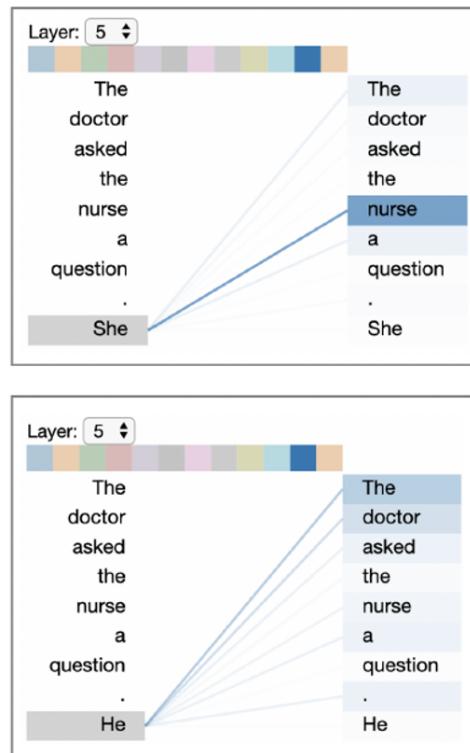


Visualization

Visualization



<https://arxiv.org/abs/1904.02679>
 (The results below are from GPT-2)



有人分析了一下GTP-2的attention做的事情是什么。

上图右侧的两列，GPT-2中左列词汇是下一层的结果，右列是前一层需要被attention的对象，我们可以观察到，She 是通过nurse attention 出来的，He是通过doctor attention 出来的，所以机器学到了某些词汇是和性别有关系的（虽然它大概不知道性别是什么）。

上图左侧，是对不同层的不同head 做一下分析，你会发现一个现象，很多不同的词汇都要attend 到第一个词汇。一个可能的原因是，如果机器不知道应该attend 到哪里，或者说不需要attend 的时候就attend 在第一个词汇。如果真是这样的话，以后我们未来在做这种model 的时候可以设一个特别的token，当机器不知道要attend到哪里的时候就attend到这个特殊token上。

Unsupervised Learning: Generation

本文将简单介绍无监督学习中的生成模型，包括PixelRNN、VAE

Introduction

正如Richard Feynman所说，“*What I cannot create, I do not understand*”，我无法创造的东西，我也无法真正理解，机器可以做猫狗分类，但却不一定知道“猫”和“狗”的概念，但如果机器能自己画出“猫”来，它或许才真正理解了“猫”这个概念

这里将简要介绍：PixelRNN、VAE和GAN这三种方法

PixelRNN

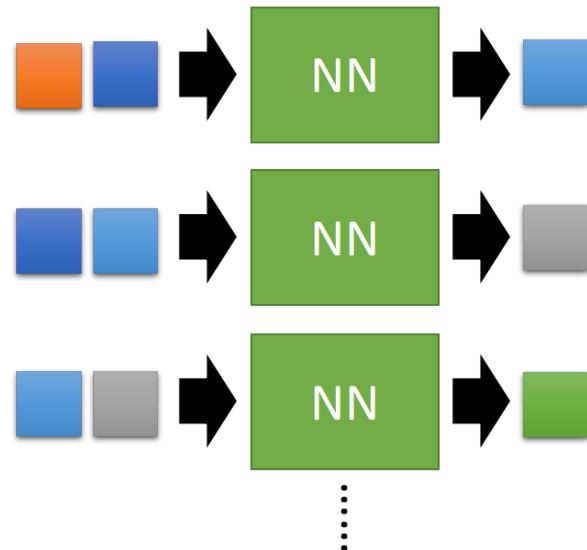
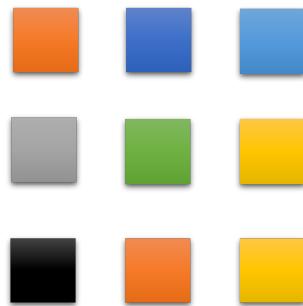
Introduction

RNN可以处理长度可变的input，它的基本思想是根据过去发生的所有状态去推测下一个状态

PixelRNN的基本思想是每次只画一个pixel，这个pixel是由过去所有已产生的pixel共同决定的

- **Image generation**

E.g. 3 x 3 images



Can be trained just with a large collection of images without any annotation

这个方法也适用于语音生成，可以用前面一段的语音去预测接下来生成的语音信号

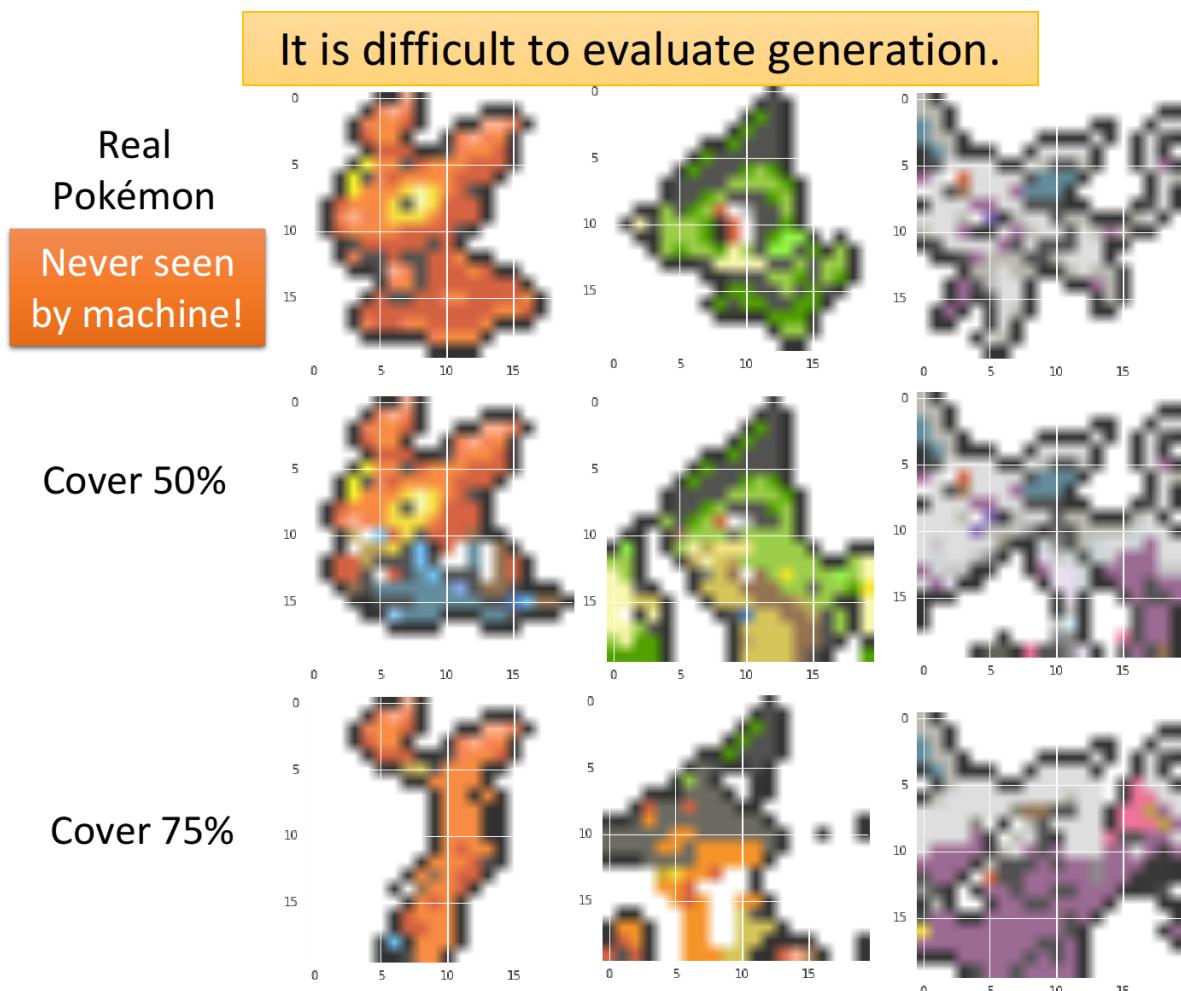
总之，这种方法的精髓在于根据过去预测未来，画出来的图一般都是比较清晰的

pokemon creation

用这个方法去生成宝可梦，有几个tips：

- 为了减少运算量，将 40×40 的图像截取成 20×20
 - 如果将每个pixel都以[R, G, B]的vector表示的话，生成的图像都是灰蒙蒙的，原因如下：
 - 亮度比较高的图像，一般都是RGB值差距特别大而形成的，如果各个维度的值大小比较接近，则生成的图像偏向于灰色
 - 如果用sigmoid function，最终生成的RGB往往都是在0.5左右，导致色彩度不鲜艳
 - 解决方案：将所有色彩集合成为一个1-of-N编码，由于色彩种类比较多，因此这里先对类似的颜色做clustering聚类，最终获得了167种色彩组成的向量
- 我们用这样的向量去表示每个pixel，可以让生成的色彩比较鲜艳

使用PixelRNN训练好模型之后，给它看没有被放在训练集中的3张图像的一部分，分别遮住原图的50%和75%，得到的原图和预测结果的对比如下：



Variational Autoencoder(VAE)

Introduction

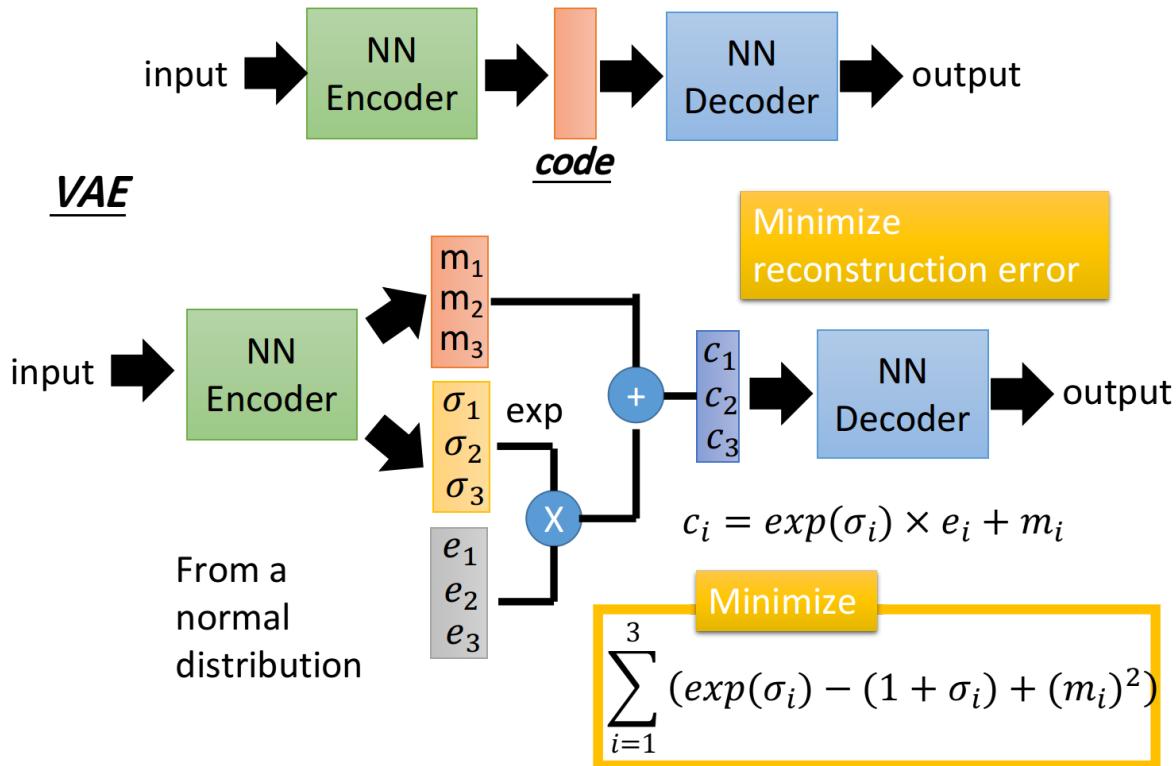
前面的文章中已经介绍过Autoencoder的基本思想，我们拿出其中的Decoder，给它随机的输入数据，就可以生成对应的图像

但普通的Decoder生成效果并不好，VAE可以得到更好的效果

在VAE中，code不再直接等于Encoder的输出，这里假设目标降维空间为3维，那我们使Encoder分别输出 m_1, m_2, m_3 和 $\sigma_1, \sigma_2, \sigma_3$ ，此外我们从正态分布中随机取出三个点 e_1, e_2, e_3 ，将下式作为最终的编码结果：

$$c_i = e^{\sigma_i} \cdot e_i + m_i$$

Auto-encoder



此时，我们的训练目标不仅要最小化input和output之间的差距，还要同时最小化下式：

$$\sum_{i=1}^3 (\exp(\sigma_i) - (1 + \sigma_i) + (m_i)^2)$$

与PixelRNN不同的是，VAE画出的图一般都是不太清晰的，但使用VAE可以在某种程度上控制生成的图像

Pokémon Creation

假设我们将这个VAE用在pokemon creation上面。

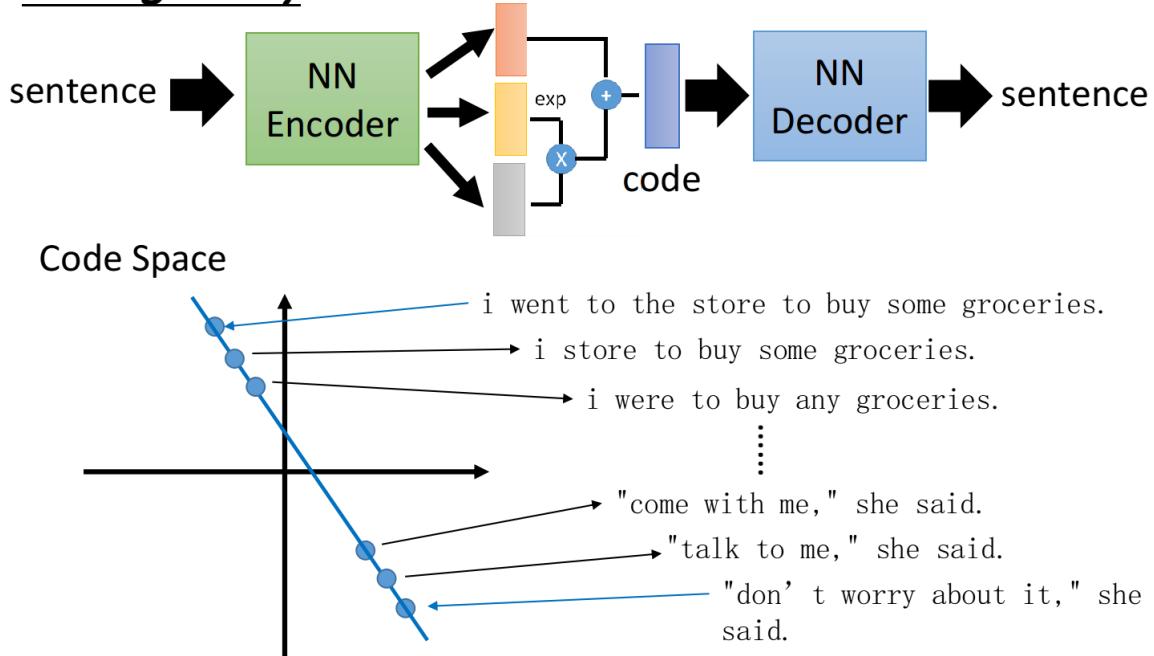
那我们在train的时候，input一个pokemon，然后你output一个的pokemon，然后learn出来的这个code就设为10维。learn好这个pockmon的VAE以后，我么就把decoder的部分拿出来。因为我们现在有一个decoder，可以input一个vector。所以你在input的时候你可以这样做：我现在有10维的vector，我固定其中8维只选其中的二维出来，在这两维dimension上面散不同的点，然后把每个点丢到decoder里面，看它合出来的image长什么样子。

那如果我们做这件事情的话，你就可以看到说：这个code的每一个dimension分别代表什么意思。如果我们可以解读code每一个dimension代表的意思，那以后我们就可以把code当做拉杆一样可以调整它，就可以产生不同的pokemon。

Write Poetry

VAE还可以用来写诗，我们只需要得到某两句话对应的code，然后在降维后的空间中得到这两个code所在点的连线，从中取样，并输入给Decoder，就可以得到类似下图中的效果

Writing Poetry



Why VAE?

VAE和传统的Autoencoder相比，有什么优势呢？

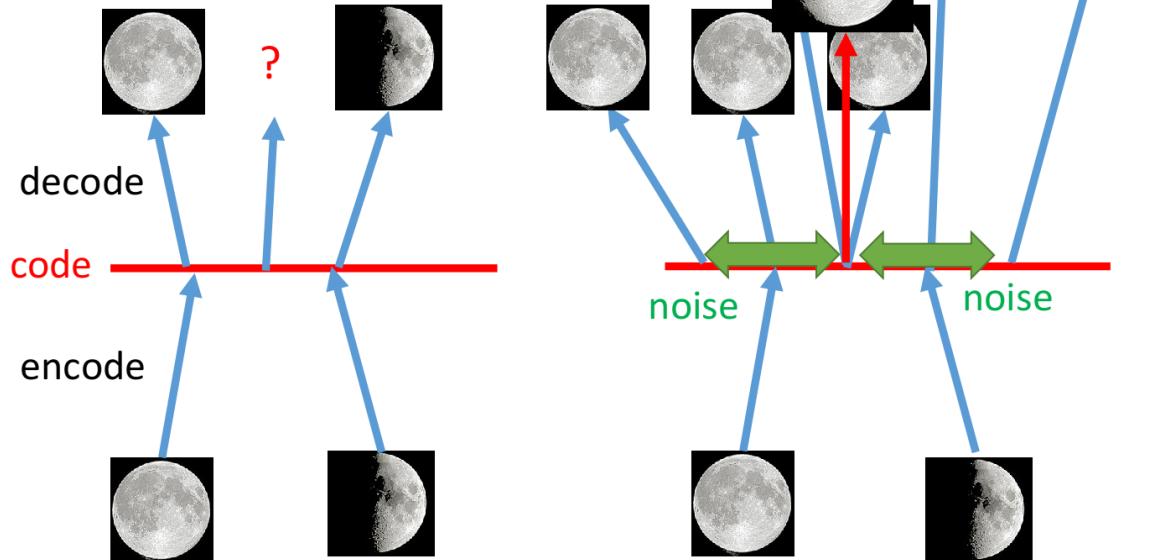
事实上，VAE就是加了噪声noise的Autoencoder，它的抗干扰能力更强，过渡生成能力也更强

对原先的Autoencoder来说，假设我们得到了满月和弦月的code，从两者连线中随机获取一个点并映射回原来的空间，得到的图像很可能是完全不一样的东西。

而对VAE来说，它要保证在降维后的空间中，加了noise的一段范围内的所有点都能够映射到目标图像，如下图所示，当某个点既被要求映射到满月、又被要求映射到弦月，VAE training的时候你要minimize mean square，所以这个位置最后产生的图会是一张介于满月和半月的图。所以你用VAE的话，你从你的code space上面去sample一个code再产生image的时候，你可能会得到一个比较好的image。如果是原来的auto-encoder的话，得到的都不像是真实的image。

Why VAE?

Intuitive Reason



再回过来头看VAE的结构，其中：

- m_i 其实就代表原来的code
- c_i 则代表加了noise以后的code
- σ_i 代表了noise的variance，描述了noise的大小，这是由NN学习到的参数
注：使用 e^{σ_i} 的目的是保证variance是正的
- e_i 是正态分布中随机采样的点

注意到，损失函数仅仅让input和output差距最小是不够的，因为variance是由机器自己决定的，如果不加以约束，它自然会去让variance=0，这就跟普通的Autoencoder没有区别了

额外加的限制函数解释如下：

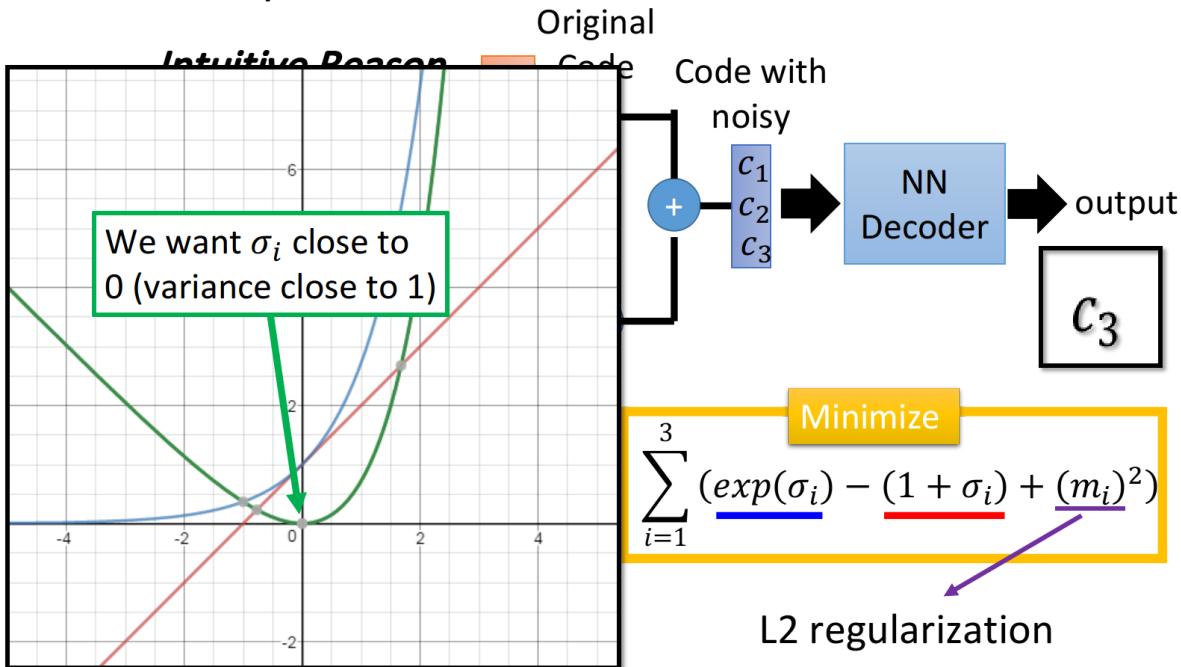
下图中，蓝线表示 e^{σ_i} ，红线表示 $1 + \sigma_i$ ，两者相减得到绿线

绿线的最低点 $\sigma_i = 0$ ，则variance $e^{\sigma_i} = 1$ ，此时loss最低

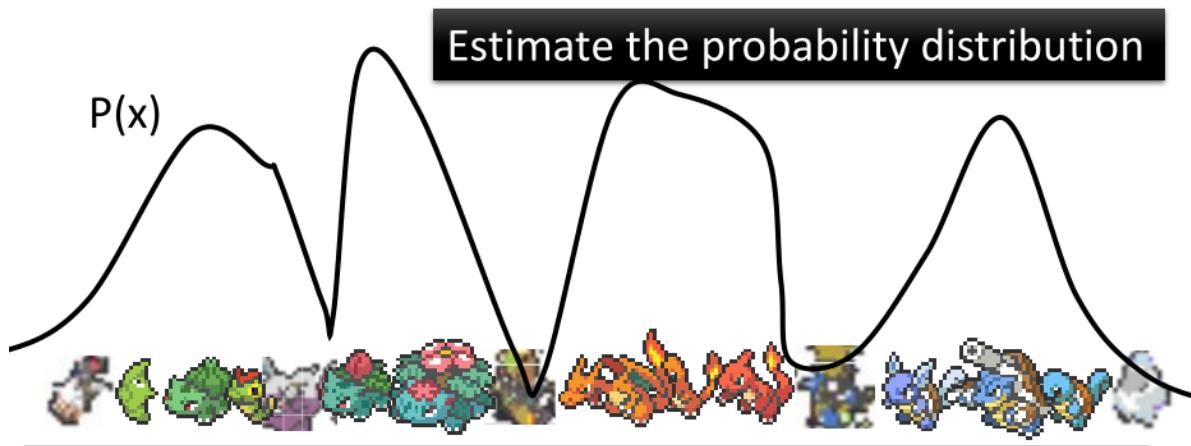
而 $(m_i)^2$ 项则是对code的L2 regularization，让它比较sparse，不容易过拟合，比较不会 learn 出太多 trivial 的 solution

Why VAE?

What will happen if we only minimize reconstruction error?



刚才是比较直观的理由，正式的理由这样的，以下是paper上比较常见的说法。



Each Pokémon is a point x in the space

回归到我们要做的事情是什么，你要 machine generate 这个pokemon的图，那每一张pokemon的图都可以想成是高维空间中的一个点。一张 image，假设它是 20×20 的 image，它在高维的空间中就是一个 20×20 ，也就是一个 400 维的点。我们这边写做 x ，虽然在图上，我们只用一维来描述它，但它其实是一个高维的空间。那我们现在要做的事情其实就是 estimate 高维空间上面的机率分布， $P(x)$ 。只要我们能够 estimate 出这个 $P(x)$ 的样子，注意，这个 x 其实是一个 vector，我们就可以根据这个 $P(x)$ ，去 sample 出一张图。那找出来的图就会像是宝可梦的样子，因为你取 $P(x)$ 的时候，机率高的地方比较容易被 sample 出来，所以，这个 $P(x)$ 理论上应该是在有宝可梦的图的地方，它的机率是大的；如果是一张怪怪的图的话，机率是低的。如果我们今天能够 estimate 出这一个 probability distribution那就结束了。

Gaussian Mixture Model

那怎么 estimate 一个 probability 的 distribution 呢?

Gaussian Mixture Model

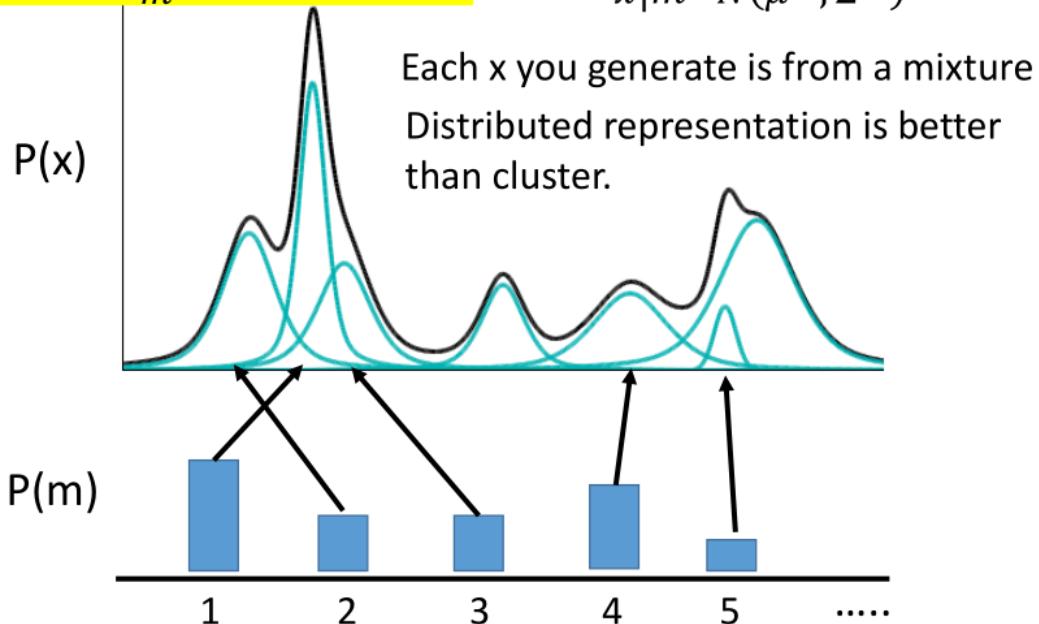
$$P(x) = \sum_m P(m)P(x|m)$$

How to sample?

$$m \sim P(m) \text{ (multinomial)}$$

m is an integer

$$x|m \sim N(\mu^m, \Sigma^m)$$



我们可以用 Gaussian mixture model。我们现在有一个 distribution, 它长这个样子, 黑色的、很复杂。我们说这个很复杂的黑色 distribution, 它其实是很多的 Gaussian。这一边蓝色的代表有很多的 Gaussian 用不同的 weight 叠合起来的结果。假设你今天 Gaussian 的数目够多, 你就可以产生很复杂的 distribution。所以, 虽然黑色很复杂, 但它背后其实是有很多 Gaussian 叠合起来的结果。根据每一个 Gaussian 的 weight 去决定你要从哪一个 Gaussian sample data, 然后, 再从你选择的那个 Gaussian 里面 sample data。如果你的gaussion数目够多, 你就可以产生很复杂的distribution, 公式为

$$P(x) = \sum_m P(m)P(x|m)。$$

如果你要从 $p(x)$ sample 出一个东西的时候, 你先要决定你要从哪一个 gaussian sample 东西, 假设现在有 100 gaussian, 你根据每个 gaussian 的 weight 去决定你要从哪一个 gaussian sample data。所以你要咋样从一个 gaussian mixture model sample data 呢? 首先你有一个 multinomial distribution, 你从 multinomial distribution 里面决定你要 sample 哪一个 gaussian, m 代表第几个 gaussian, 它是一个 integer。你决定好你要从哪一个 m sample gaussian 以后, , 你有了 m 以后就可以找到 μ^m, σ^m (每一个 gaussian 有自己的 μ^m, σ^m) , 根据 μ^m, σ^m 就可以 sample 一个 x 出来。所以 $p(x)$ 写为 summation over 所有的 gaussian 的 weight 乘以 sample 出 x 的机率。

每一个 x 都是从某一个 mixture 被 sample 出来的, 这件事情其实就很像是做 classification 一样。我们每一个所看到的 x , 它都是来自于某一个分类。但是我们之前有讲过说: 把 data 做 cluster 是不够的, 更好的表示方式是用 distributed representation, 也就是说每一个 x 它并不是属于某一个 class, 而是它有一个 vector 来描述它的各个不同的特性。所以 VAE 就是 gaussian mixture model 的 distributed representation 的版本。

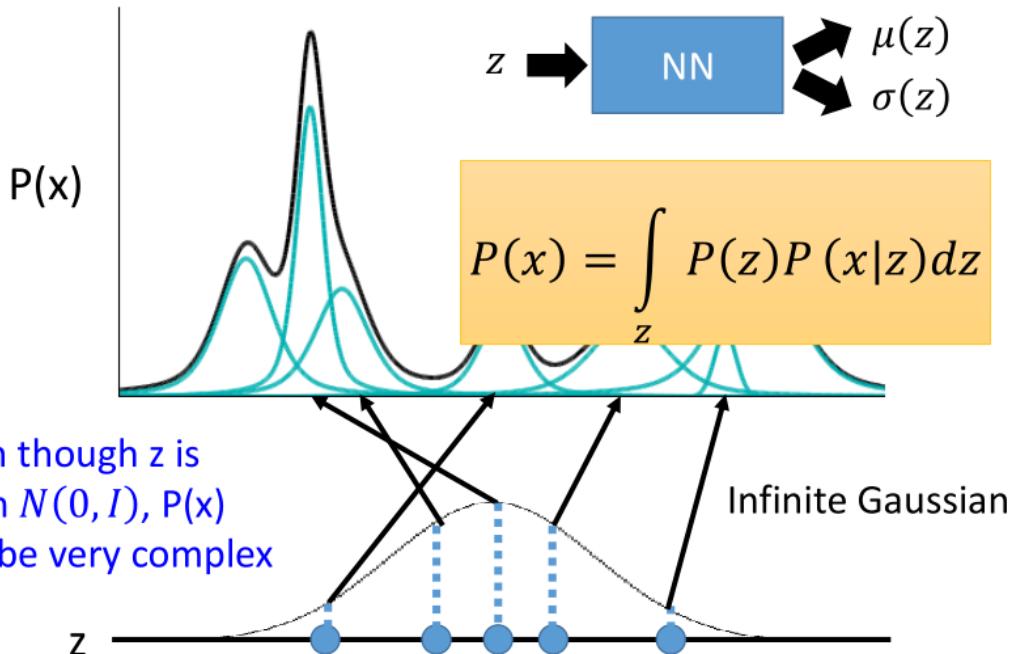
VAE

$$z \sim N(0, I)$$

z is a vector from normal distribution

$$x|z \sim N(\mu(z), \sigma(z))$$

Each dimension of z
represents an attribute



首先我们要sample一个 z ，这个 z 是从normal distribution中sample出来的。这个vector z 的每一个dimension就代表了某种attribute，如图中所示，假设是 z 是一维的，实际上 z 可能是一个10维的、100维的vector。到底有几维，是由你自己决定。接下来你Sample z 以后，根据 z 你可以决定 $\mu(z), \sigma(z)$ ，你可以决定gaussian的 μ, σ 。刚才在gaussian model里面，你有10个mixture，那你就有了10个 μ, σ ，但是在这个地方，你的 z 有无穷多的可能，所以你的 $\mu(z), \sigma(z)$ 也有无穷多的可能。那咋样找到这个 $\mu(z), \sigma(z)$ 呢？做法是：假设 $\mu(z), \sigma(z)$ 都来自于一个function，你把 z 带到产生 μ 的这个function $N(\mu(z), \sigma(z))$ ， $\mu(z)$ 代表说：现在如果你的attribute是 z 的时候，你在 x space上面的 μ 是多少。同理 $\sigma(z)$ 代表说： σ 是多少

其实 $P(x)$ 是这样产生的：在 z 这个space上面，每一个点都有可能被sample到，只不过是中间这些点被sample出来的机率比较大。当你sample出来点以后，这个point会对应到一个guassian。至于一个点对应到什么样的gaussian，它的 μ, σ 是多少，是由某一个function来决定的。所以当gaussian是从normal distribution所产生的时候，就等于你有无穷多个gaussian。

另外一个问题是：我们怎么知道每一个 z 应该对应到什么样的 μ, σ （这个function如何去找）。我们知道 neural network 就是一个function，所以你就可以说：我就是在train一个neural network，这个neural network的input就是 z ，它的output就是两个vector ($\mu(z), \sigma(z)$)。

$P(x)$ 的distribution为 $P(x) = \int_z P(z)P(x|z)dz$

那你可能会困惑，为什么是gaussian呢？你可以假设任何形状的，这是你自己决定的。你可以说每一个attribute的分布就是gaussian，因为极端的case总是少的，比较没有特色的东西总是比较多的。你不用担心如果假设gaussian会不会对 $P(x)$ 带来很大的限制：NN是非常powerful的，NN可以represent任何的function。所以就算你的 z 是normal distribution，最后的 $P(x)$ 最后也可以是很复杂的distribution。

Maximizing Likelihood

$p(z)$ is a normal distribution, $x|z$ 表示我们先知道 z 是什么，然后我们就可以决定 x 是从什么样子的 mean 跟 variance 的 Gaussian里面被 sample 出来的， $\mu(z), \sigma(z)$ 是等待被找出来的。

但是，问题是要怎么找呢？它的criterion就是maximizing the likelihood，我们现在手上已经有一笔data x ，你希望找到一组 μ 的function和 σ 的function，它可以让你现在已经有的image x ，它的 $p(x)$ 取log之后相加被maximize。 z 通过一个NN产生这个 μ 跟 σ ，所以我们要做的事情就是，调整NN里面的参数（每个neural的weight bias），使得likehood可以被maximize。

引入另外一个distribution，叫做 $q(z|x)$ 。也就是我们有另外一个 NN' ，input一个 x 以后，它会告诉你说：对应在 z 这个space上面的 μ', σ' (给它 x 以后，它会决定这个 z 要从什么样的 μ', σ' 被sample出来)。这个 NN' 就是VAE里的Encoder，前面说的 NN 就是Decoder

Maximizing Likelihood

$$P(x) = \int_z P(z)P(x|z)dz$$

$$L = \sum_x \log P(x)$$

$P(z)$ is normal distribution

$$x|z \sim N(\mu(z), \sigma(z))$$

$\mu(z), \sigma(z)$ is going to be estimated

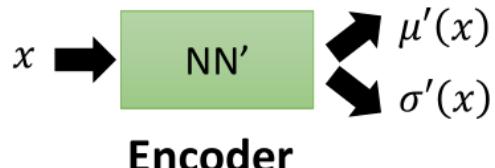
Maximizing the likelihood of the observed x

Tuning the parameters to maximize likelihood L



We need another distribution $q(z|x)$

$$z|x \sim N(\mu'(x), \sigma'(x))$$



$\log P(x) = \int_z q(z|x) \log P(x) dz$ ，对任何distribution $q(z|x)$ 都成立。因为这个积分是跟 $P(x)$ 无关的，然后就可以提出来，积分的部分就会变成1，所以左式就等于右式。

由条件概率 $P(A|B) = P(AB)/P(B)$ ，得到第二行。log 中的式子拆开，得到第三行。右边这一项，它代表了一个 KL divergence。KL divergence 代表的是这两个 distribution 相近的程度，如果 KL divergence 它越大代表这两个 distribution 越不像，这两个 distribution 一模一样的时候，KL divergence 会是0。所以，KL divergence 它是一个距离的概念，它衡量了两个 distribution 之间的距离。最小为0。左边一项经过变化，得到 L 的 lower bound L_b 。

Maximizing Likelihood

$$P(x) = \int_z P(z)P(x|z)dz$$

$P(z)$ is normal distribution

$$x|z \sim N(\mu(z), \sigma(z))$$

$\mu(z), \sigma(z)$ is going to be estimated

$$L = \sum_x \log P(x)$$

Maximizing the likelihood of the observed x

$$\log P(x) = \int_z q(z|x) \log P(x) dz$$

$$= \int_z q(z|x) \log \left(\frac{P(z,x)}{P(z|x)} \right) dz = \int_z q(z|x) \log \left(\frac{P(z,x)}{q(z|x)} \frac{q(z|x)}{P(z|x)} \right) dz$$

$$= \int_z q(z|x) \log \left(\frac{P(z,x)}{q(z|x)} \right) dz + \int_z q(z|x) \log \left(\frac{q(z|x)}{P(z|x)} \right) dz$$

$$KL(q(z|x)||P(z|x)) \geq 0$$

$$\geq \int_z q(z|x) \log \left(\frac{P(x|z)P(z)}{q(z|x)} \right) dz$$

lower bound L_b

我们要maximize的对象是由这两项加起来的结果，在 L_b 这个式子中， $p(z)$ 是已知的，我们不知道的是 $p(x|z)$ 跟 $q(z|x)$ 。我们本来要做的事情是要找 $p(x|z)$ ，让likelihood越大越好，现在我们要做的事情变成要找 $p(x|z)$ 跟 $q(z|x)$ ，让 L_b 越大越好。

如果我们只找 $p(x|z)$ ，然后去maximizing L_b 的话，那因为你要找的这个 likelihood，它是 L_b 的upper bound，所以，你增加 L_b 的时候，你有可能会增加你的 likelihood。但是，你不知道你的这个 likelihood跟你的 lower bound 之间到底有什么样的距离。你希望做到的事情是当你的 lower bound 上升的时候，你的 likelihood 是会比 lower bound 高，然后你的 likelihood 也跟着上升。但是，你有可能会遇到一个比较糟糕的状况是你的 lower bound 上升的时候，likelihood 反而下降。虽然，它还是 lower bound，它还是比 lower bound 大，但是，它有可能下降。因为根本不知道它们之间的差距是多少。

所以，引入 q 这一项呢，其实可以解决刚才说的那一个问题。因为 $likelihood = L_b + KL divergence$ 。如果你今天去这个调 $q(z|x)$ ，去 maximize L_b 的话，会发生什么事呢？首先 q 这一项跟 $\log P(x)$ 是一点关系都没有的， $\log P(x)$ 只跟 $P(x|z)$ 有关，所以，这个值是不变的，蓝色这一条长度都是一样的。我们现在去 maximize L_b ，maximize L_b 代表说你 minimize 了 KL divergence，也就是说你会让你的 lower bound 跟你的这个 likelihood越来越接近，假如你固定住 $P(x|z)$ 这一项，然后一直去调 $q(z|x)$ 这一项的话，让这个 L_b 一直上升，最后这一个 KL divergence 会完全不见。

假如你最后可以找到一个 q ，它跟这个 $p(z|x)$ 正好完全 distribution 一模一样的话，你就会发现说你的 likelihood 就会跟lower bound 完全停在一起，它们就完全是一样大。这个时候呢，如果你再把 lower bound 上升的话，因为你的 likelihood 一定要比 lower bound 大。所以这个时候你的 likelihood 你就可以确定它一定会上升。所以，这个就是引入 q 这一项它有趣的地方。

一个副产物，当你在 maximize q 这一项的时候，你会让这个 KL divergence 越来越小，你会让这个 $q(z|x)$ 跟 $P(z|x)$ 越来越接近。

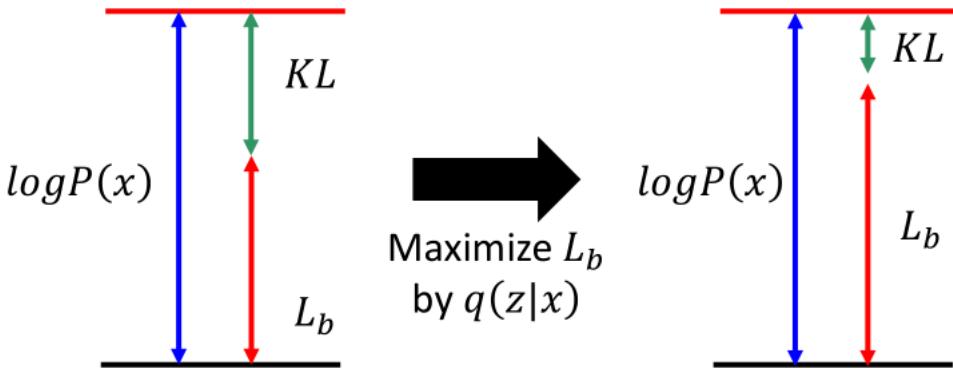
所以我们接下要做的事情就是找 $P(x|z)$ and $q(z|x)$ ，可以让 L_b 越大越好。让 L_b 越大越好就等同于我们可以让 likelihood 越来越大，而且你顺便会找到 q 可以去 approximation of $p(z|x)$

Maximizing Likelihood

$$\log P(x) = L_b + KL(q(z|x)||P(z|x))$$

$$L_b = \int_z q(z|x) \log \left(\frac{P(x|z)P(z)}{q(z|x)} \right) dz$$

Find $P(x|z)$ and $q(z|x)$
maximizing L_b



$q(z|x)$ will be an approximation of $p(z|x)$ in the end

对于 L_b log 里面相乘，拆开，得到 $P(z)$ 跟 $q(z|x)$ 的 KL divergence

$$L_b = \int_z q(z|x) \log \left(\frac{P(z,x)}{q(z|x)} \right) dz = \int_z q(z|x) \log \left(\frac{P(x|z)P(z)}{q(z|x)} \right) dz$$

$$= \int_z \underbrace{q(z|x) \log \left(\frac{P(z)}{q(z|x)} \right) dz}_{-KL(q(z|x)||P(z))} + \int_z q(z|x) \log P(x|z) dz$$

$$z|x \sim N(\mu'(x), \sigma'(x))$$



Connection with Network

q 是一个 neural network，当你给 x 的时候，它会告诉你 $q(z|x)$ 是从什么样的 mean 跟 variance 的 Gaussian 里面 sample 出来的。所以，我们现在如果你要 minimize 这个 $P(z)$ 跟 $q(z|x)$ 的 KL divergence 的话，你就是去调 output 让它产生的 distribution 可以跟这个 normal distribution 越接近越好。minimize 这一项其实是我们刚才在 reconstruction error 外加的那一项 $\sum_{i=1}^3 (\exp(\sigma_i) - (1 + \sigma_i) + (m_i)^2)$ ，它要做的事情就是 minimize KL divergence，希望 $q(z|x)$ 的 output 跟 normal distribution 是接近的。

Connection with Network

Minimizing $KL(q(z|x)||P(z))$

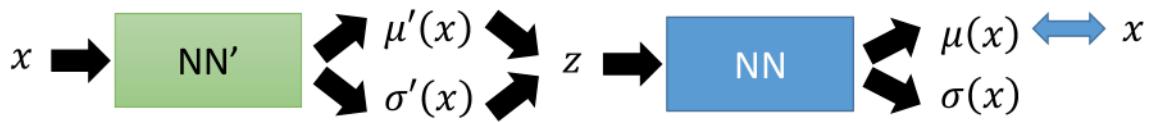
$$\text{Minimize} \quad \sum_{i=1}^3 (\exp(\sigma_i) - (1 + \sigma_i) + (m_i)^2)$$



(Refer to the Appendix B of the original VAE paper)

Maximizing

$$\int_z q(z|x) \log P(x|z) dz = E_{q(z|x)}[\log P(x|z)]$$



This is the auto-encoder

另外一项是要这个积分的意思就是

你可以想象，我们有一个 $\log P(x|z)$ ，然后，它用 $q(z|x)$ 来做 weighted sum。所以，你可以把它写成 $[\log P(x|z)]$ 根据 $q(z|x)$ 的期望值

这个式子的意思就好像是说：给我们一个 x 的时候，我们去根据这个 $q(z|x)$ ，这个机率分布去 sample 一个 data，然后，要让 $\log P(x|z)$ 的机率越大越好。那这一件事情其实就 Auto-encoder 在做的事情。

怎么从 $q(z|x)$ 去 sample 一个 data 呢？你就把 x 丢到 neural network 里面去，它产生一个 mean 跟一个 variance，根据这个 mean 跟 variance，你就可以 sample 出一个 z 。

你已经根据现在的 x sample 出一个 z ，接下来，你要 maximize 这一个 z ，产生这个 x 的机率。

这个 z 产生这个 x ，是把这个 z 丢到另外一个 neural network 里面去，它产生一个 mean 跟 variance，要怎么让这个 NN output 所代表 distribution 产生 x 的机率越大越好呢？假设我们无视 variance 这一件事情的话，因为在一般实作里面你可能不会把 variance 这一件事情考虑进去。你只考虑 mean 这一项的话，那你要做的事情就是：让这个 mean 跟你的 x 越接近越好。你现在是一个 Gaussian distribution，那 Gaussian distribution 在 mean 的地方机率是最高的。所以，如果你让这个 NN output 的这个 mean 正好等于你现在这个 data x 的话，这一项 $\log P(x|z)$ 它的值是最大的。

所以，现在这整个 case 就变成说，input 一个 x ，然后，产生两个 vector，然后 sample 产生一个 z ，再根据这个 z ，你要产生另外一个 vector，这个 vector 要跟原来的 x 越接近越好。这件事情其实就是 Auto-encoder 在做的事情。所以这两项合起来就是刚才我们前面看到的 VAE 的 loss function。

problems of VAE

VAE其实有一个很严重的问题就是：它从来没有真正学过如何产生一张看起来像真的image，它学到的东西是：它想要产生一张image，跟我们在database里面某张image越接近越好。

但它不知道的是：我们 evaluate 它产生的 image 跟 database 里面的相似度的时候 (MSE 等等)，decoder output 跟真正的 image 之间有一个 pixel 的差距，不同的 pixel 落在不同的位置会得到非常不一样的结果。假设这个不一样的 pixel 落在 7 的尾部 (让 7 比较长一点)，跟落在另外一个地方 (右边)。你一眼就看出说：右边这是怪怪的 digit，左边这个搞不好是真的。但是对 VAE 来说都是一个 pixel 的差异，对它来说这两张

image是一样的好或者是一样的不好。

所以VAE学的只是怎么产生一张image跟database里面的一模一样，从来没有想过：要真的产生可以一张以假乱真的image。所以你用VAE来做training的时候，其实你产生出来的image往往都是database里面的image linear combination而已。因为它从来都没有想过要产生一张新的image，它唯一做的事情就是希望它产生的 image 跟 data base 的某张 image 越像越好，模仿而已。

GAN

GAN，对抗生成网络，是近两年非常流行的神经网络，基本思想就像是天敌之间相互竞争，相互进步

GAN由生成器(Generator)和判别器(Discriminator)组成：

- 对判别器的训练：把生成器产生的图像标记为0，真实图像标记为1，丢给判别器训练分类
- 对生成器的训练：input: Vectors from a distribution，调整生成器的参数，使产生的图像能够骗过判别器
- 每次训练调整判别器或生成器参数的时候，都要固定住另一个的参数

In practical

- GANs are difficult to optimize.
- No explicit signal about how good the generator is
 - In standard NNs, we monitor loss
 - In GANs, we have to keep “well-matched in a contest”
- When discriminator fails, it does not guarantee that generator generates realistic images
 - Just because discriminator is stupid
 - Sometimes generator find a specific example that can fail the discriminator
- Making discriminator more robust may be helpful.

GAN的问题：没有明确的训练目标，很难调整生成器和判别器的参数使之始终处于势均力敌的状态，当两者之间的loss很小的时候，并不意味着训练结果是好的，有可能它们两个一起走向了一个坏的极端，所以在训练的同时还要有人在旁边关注着训练的情况

Anomaly Detection

Anomaly Detection

异常探测就是要让机器知道它不知道这件事

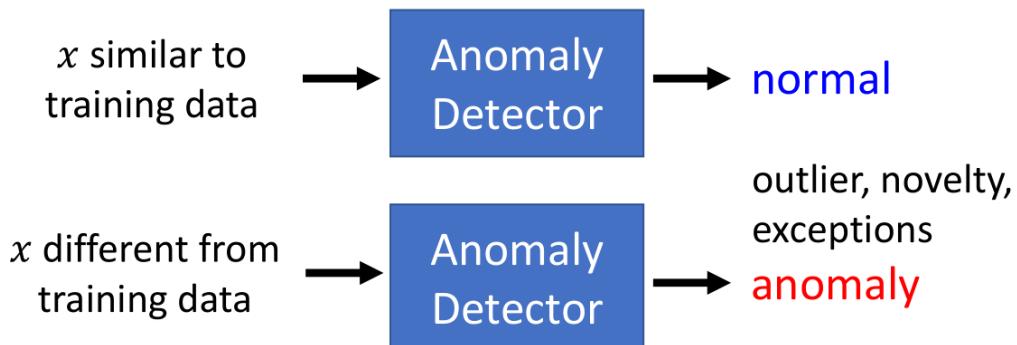
Problem Formulation

异常侦测的问题通常formulation成这样，假设我们现在有一堆训练数据 (x^1, x^2, \dots, x^N) ，(在这门课里面，我们通常用上标来表示一个完整的东西，用下标来表示一个完整东西的其中一部分)。我们现在要找到一个function，这个function要做的事情是：检测输入x的时，决定现在输入的x到底跟我们的训练数据是相似还是不相似的。

我们一直在用Anoamly这个词汇，可能会让某些同学觉得机器在做Anoamly Detector都是要Detector不好的结果，因为异常这个词汇显然通常代表的是负面意思。其实Anoramly Detector并不一定是找不好的结果，只是找跟训练数据不一样的数据。所以我们找出结果不见得是异常的数据，你会发现Anoamly Detector在不同的领域里面有不同名字。有时候我们会叫它为“Outlier Detector, Novelty Detector, Exceprions Detector”

总之我们要找的是跟训练数据不一样的数据，但至于什么叫做similar，这就是Anomaly Detector需要探讨的问题。不同的方法就有不同的方式来定义什么叫做“像”、什么叫做“不像”。

- Given a set of training data $\{x^1, x^2, \dots, x^N\}$
- We want to find a function detecting input x is similar to training data or not.



Different approaches use different ways to determine the similarity.

What is Anomaly?

这里我要强调一下什么叫做异常，机器到底要看到什么就是Anomaly。其实是取决于你提供给机器什么样的训练数据。

假设你提供了很多的雷丘作为训练数据，皮卡丘就是异常的。若你提供了很多的皮卡丘作为训练数据，雷丘就是异常的。若你提供很多的宝可梦作为训练数据，这时数码宝贝就是异常的。

Applications

Fraud Detection

异常侦测有很多的应用，你可以应用到诈骗侦测（Fraud Detection）。训练数据是正常的刷卡行为，收集很多的交易记录，这些交易记录视为正常的交易行为，若今天有一笔新的交易记录，就可以用异常检测的技术来侦测这笔交易记录是否有盗刷的行为。（正常的交易金额比较小，频率比较低，若短时间内有非常多的高额消费，这可能是异常行为）

Network Intrusion Detection

异常侦测还可以应用到网络系统的入侵侦测，训练数据是正常连线。若有一个新的连线，你希望用Anomaly Detection让机器自动决定这个新的连线是否为攻击行为

Cancer Detection

异常侦测还可以应用到医疗（癌细胞的侦测），训练数据是正常细胞。若给一个新的细胞，让机器自动决定这个细胞是否为癌细胞。

Binary Classification?

我们咋样去做异常侦测这件事呢？很直觉的想法就是：若我们现在可以收集到很多正常的资料 $\{x^1, x^2, \dots, x^N\}$ ，我们可以收集到很多异常的资料 $\{\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^N\}$ 。我们可以将normal data当做一个Class (Class1)，anomaly data当做另外一个Class (Class2)。我们已经学过了binary classification，这时只需要训练一个binary classifier，然后就结束了。

这个问题其实并没有那么简单，因为不太容易把异常侦测视为一个binary classification的问题。为什么这样说呢？

假设现在有一笔正常的训练数据是宝可梦，只要不是宝可梦就视为是异常的数据，这样不只是数码宝贝是异常数据，凉宫春日也是异常数据，茶壶也是异常的数据。不属于宝可梦的数据太多了，不可能穷举所有不是宝可梦的数据。根本没有办法知道整个异常的数据 (Class2) 是怎样的，所以不应该将异常的数据视为一个类别，应为它的变化太大了。这是第一个不能将异常侦测视为二元分类的原因。

第二个原因是：很多情况下不太容易收集到异常的资料，收集正常的资料往往比较容易，收集异常的资料往往比较困难。对于刚才的诈欺侦测例子而言，你可以想象多数的交易通常都是正常的，很难找到异常的交易。这样就造成异常侦测不是一个单纯的二元分类问题，需要想其它的方法，它是一个独立的研究主题，仍然是一个尚待研究的问题。

Categories

接下来对异常侦测做一个简单的分类

With labels

一类，是不只有训练数据 $\{x^1, x^2, \dots, x^N\}$ ，同时这些数据还具有label $\{\hat{y}^1, \hat{y}^2, \dots, \hat{y}^N\}$ 。用这样的数据集可以train出一个classifier，让机器通过学习这些样本，以预测出新来的样本的label，但是我们希望分类器有能力知道新给样本不属于原本的训练数据的任何类别，它会给新样本贴上“unknown”的标签。训练classifier可以用generative model、logistic regression、deep learning等方法，你可以从中挑一个自己喜欢的算法train 出一个classifier。

上述的这种类型的任务，train出的classifier 具有看到不知道的数据会标上这是未知物的能力，这算是异常检测的其中一种，又叫做Open-set Recognition。我们希望做分类的时候模型是open 的，它可以辨识它没看过的东西，没看过的东西它就贴一个“unknown”的标签。

Without labels

另一类，所有训练数据都是没有label 的，这时你只能根据现有资料的特征去判断，新给的样本跟原先的样本集是否相像。这种类型的数据又分成两种情况：

- Clean：手上的样本是干净的（所有的训练样本都是正常的样本）
- Polluted：手上的样本已经被污染（训练样本已经被混杂了一些异常的样本，更常见）

情况二是更常见的，对于刚才的诈欺检测的例子而言，银行收集了大量的交易记录，它把所有的交易记录都当做是正常的，然后告诉机器这是正常交易记录，然后希望机器可以借此检测出异常的交易。但所谓的正常的交易记录可能混杂了异常的交易，只是银行在收集资料的时候不知道这件事。所以我们更多遇到的是：手上有训练样本，但我没有办法保证所有的训练样本都是正常的，可能有非常少量的训练样本是异常的。

Case 1: With Classifier

现在给定的例子是要侦测一个人物是不是来自辛普森家庭，可以看出 x^1, x^2, x^3, x^4 是来自辛普森家庭（辛普森家庭的人有很明显的特征：脸是黄色的，嘴巴像似鸭子），同时也可以看出凉宫春日显然不是来自辛普森家庭。

假设我们收集的辛普森家庭的人物都具有标注（霸子，丽莎，荷马，美枝），有了这些训练资料以后就可以训练出一个辛普森家庭成员的分类器。我们就可以给分类器看一张照片，它就可以判断这个照片中的人物是辛普森家庭里面的哪个人物。

How to use the Classifier

现在我们想做的事情是根据这个分类器来进行异常侦测，判断这个人物是否来自辛普森家庭。

我们原本是使用分类器来进行分类，现在希望分类器不仅可以来自分类，还会输出一个数值，这个数值代表信心分数（Confidence score），然后根据信心分数做异常侦测这件事情。

定义一个阈值称之为 λ ，若信心分数大于 λ 就说明是来自于辛普森家庭。若信心分数小于 λ 就说明不是来自于辛普森家庭

How to estimate Confidence

咋样可以得到信心分数呢？若我们将图片输入辛普森家庭的分类器中，若分类器非常的肯定这个图片到底是谁，输出的信心分数就会非常的高。当我们把图片输入分类器时，分类器的输出是一个机率分布（distribution），所以将一张图片输入分类器时，分类器会给事先设定的标签一个分数。

如图所示，将“霸子”图片输入分类器，分类器就会给“霸子”一个很高的分数。

但你若给它一张很奇怪的图片（凉宫春日），这时输出的分数会特别的平均，代表机器是没有信心的。若输出特别平均，那这张图片就是异常的图片

刚才讲的都是定性的分析，现在需要将定性分析的结果化为信心分数。

一个非常直觉的方法就是将分类器的分布中最高数值作为信心分数，所以上面那张图输出的信心分数为0.97（霸子），下面那张图输出的信心分数为0.26（凉宫春日）

根据信心分数来进行异常检测不是唯一的方法，因为输出的是distribution，那么就可以计算entropy。entropy越大就代表输出越平均，代表机器没有办法去肯定输出的图片是哪个类别，表示输出的信心分数是比较低。总之我们有不同的方法根据分类器决定它的信心分数。

现在我输入一张训练资料没有的图片（荷马），分类器输出荷马的信心分数是1.00；输入霸子的图片，分类器输出霸子的信心分数为0.81，输出郭董的信心分数为0.12；输入三玖的图片，分类器输出柯阿三的信心分数为0.34，输出陈趾鹹的信心分数为0.31，输出鲁肉王的信心分数为0.10。

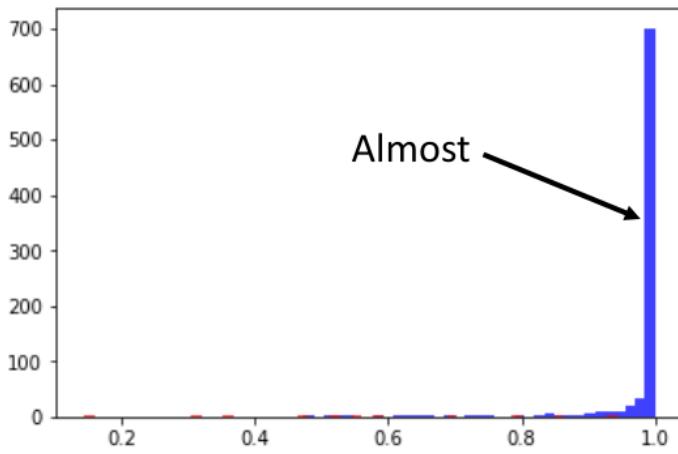
以上都是动漫人物，现在输入一张真人的图片，分类器输出柯阿三的信心分数为0.63，输出宅神的信心分数为0.08，输出小丑阿基的信心分数为0.04，输出孔龙金的信心分数为0.03。

我们可以发现，如果输入的是辛普森家庭的人物，分类器输出比较高信心分数。如果输入不是辛普森家庭的任务，分类器输出的信心分数是比较低。

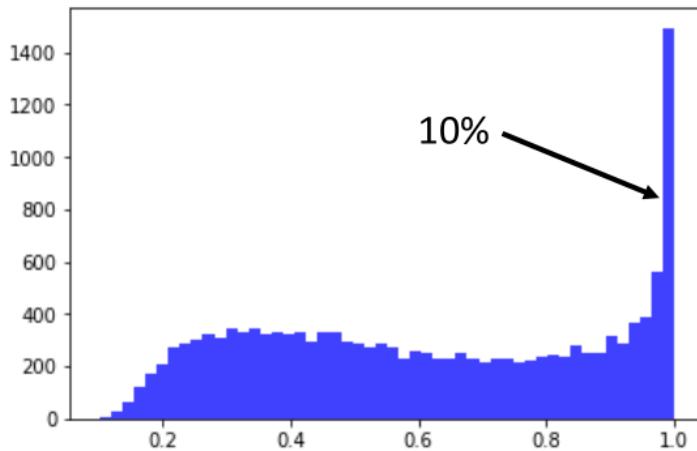
但是也有一些例外，比如输入凉宫春日的图片，分类器输出柯阿三的信心分数为0.99。

若输入大量的训练资料输入至分类器中，输出的信心分数分布如图所示。几乎所有的辛普森家庭的人物输入分类器中，无论是否辨识有错都会给出一个较高的信心分数。

但还是发现若辨识有错误会得到较低的信心分数，如图所示的红色点就是辨识错误图片的信心分数的分布。蓝色区域分布相较于红色区域集中在1的地方，有很高的信心分数认为是辛普森家庭的人物。



Confidence score distribution for *characters from Simpsons*



Confidence score distribution for *anime characters*



若输入其它动画的人物图片，其分类器输出的信心分数如题所示，我们会发现有1/10的图片的信心分数比较高（不是辛普森家庭的人物，但给了比较高的分数），多数的图片得到的信心分数比较低。

Network for Confidence Estimation

刚才是比较直观的给出了一个信心分数，你可能会觉得这种方法会让你觉得非常弱，不过刚才那种非常简单的方法其实在实作上往往还可以有不错的结果。

若你要做异常侦测的问题，现在有一个分类器，这应该是你第一个要尝试的baseline。虽然很简单，但不见得结果表现会很差。

也有更好的方法，比如你训练一个neuron network时，可以直接让neuron network输出信心分数，具体细节可参考：Terrance DeVries, Graham W. Taylor, Learning Confidence for Out-of-Distribution Detection in Neural Networks, arXiv, 2018

Example Framework

Training Set

我们有大量的训练资料，且训练资料具有标注（辛普森家庭哪个人物），因此我们可以训练一个分类器。不管用什么方法，可以从分类器中得到对所有图片的信心分数。然后就根据信心分数建立异常侦测的系统，若信心分数高于某个阈值（threshold）时就认为是正常，若低于某个阈值（threshold）时就认为是异常。

Dev Set

在之前的课程中已经讲了Dev Set的概念，需要根据Dev Set调整模型的超参数（hyperparameter），才不会过拟合。

在异常侦测的任务里面我们的Dev Set，不仅是需要大量的images，还需要被标注这些图片是来自辛普森家庭的人物还是不是来自辛普森家庭的人物。需要强调的是在训练时所有的资料都是来自辛普森家庭的人物，标签是来自辛普森家庭的哪一个人物。

但是我们要做Dev Set时，Dev Set要模仿测试数据集（testing Set），Dev Set要的并不是一张图片（辛普森家庭的哪一个人物），而应该是：辛普森家庭的人物和不是辛普森家庭的人物。

有了Dev Set以后，我们就可以把我们异常侦测的系统用在Dev Set，然后计算异常侦测系统在Dev Set上的performance是多少。你能够在Dev Set衡量一个异常侦测系统的performance以后，你就可以用Dev Set调整阀值（threshold），找出让最好的阀值（threshold）。

Testing Set

决定超参数以后（hyperparameters），就有了一个异常侦测的系统，你就可以让它上线。输入一张图片，系统就会决定是不是辛普森家庭的人物。

Evaluation

接下来要讲的是：如何计算一个异常侦测系统的性能好坏？现在有100张辛普森家庭人物的图片和5张不是辛普森家庭人物的图片。如图所示，辛普森家庭是用蓝色来进行表示，你会发现基本都集中在高分的区域。5张不是辛普森家庭的图片用红色来表示。

你会发现图的左边有一个辛普森家庭人物的分数是非常低的，在异常侦测时机器显然会在这里犯一个错误，认为它不是辛普森家庭人物，这张图片是辛普森家庭的老爷爷。

第一个图片是看起来像安娜贝尔的初音，第二张图片是小樱，第三张图片也是小樱，第四张图是凉宫春日，第五张图是魔法少女。我们会发现这个魔法少女的信心分数非常的高（0.998），事实上在这个bar中有百分之七十五的信心分数都高于0.998，多数辛普森家庭人物得到的信心分数都是1。

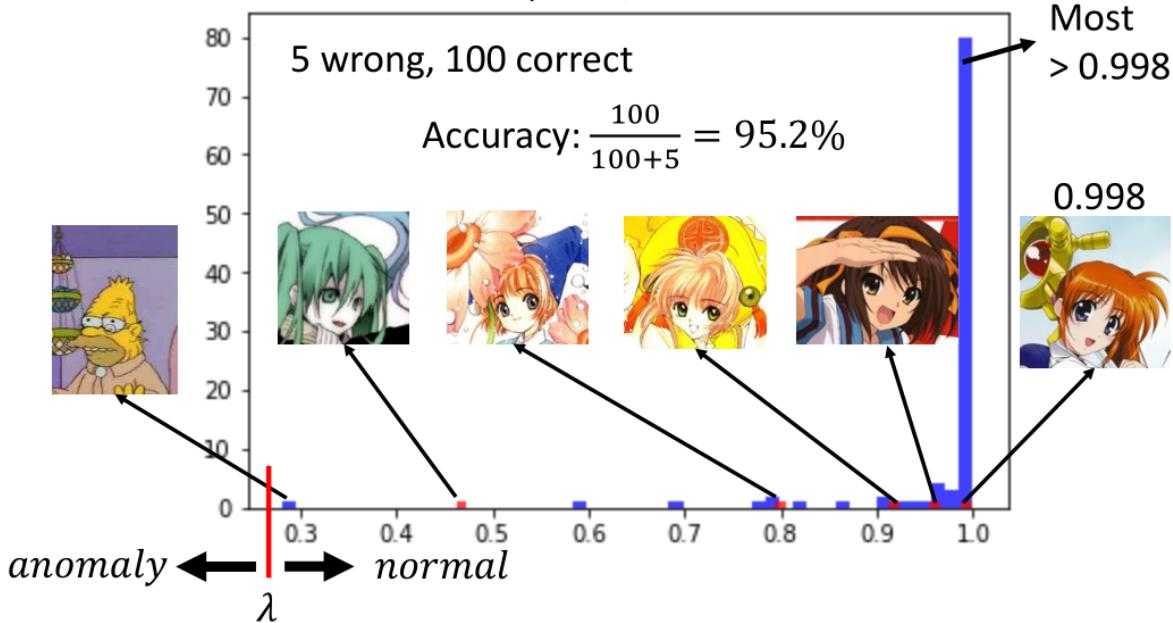
很多人在实作时，发现这张异常的图片却给到了0.998很高的分数。但你发现那些正常的图片往往得到更高的分数。虽然这些异常的图片可以得到很高的分数，但如果正常图片的分数那么高，还是可以得到较好的异常侦测的结果。

Accuracy is not a good measurement!

Evaluation

A system can have high accuracy, but do nothing.

100 Simpsons, 5 anomalies



我们咋样来评估一个异常侦测系统的好坏呢？我们知道异常侦测其实是一个二元分类 (binary classification) 的问题。在二元分类中我们都是用正确率来衡量一个系统的好坏，但是在异常侦测中正确率并不是一个好的评估系统的指标。你可能会发现一个系统很可能有很高的正确率，但其实这个系统什么事都没有做。为什么这样呢？因为在异常侦测的问题中正常的数据和异常的数据之间的比例是非常悬殊的。在这个例子里面，我们使用了正常的图片有一百张，异常的图片有五张。

通常来说正常的资料和异常的资料之间的比例是非常悬殊的，所以只用准确率衡量系统的好坏会得到非常奇怪的结果的。

在如图所示的例子中，我们认为有一个异常侦测的系统，它的 λ 设为0.3以下。 λ 以上认为是正常的， λ 以下认为是异常的。这时你会发现这个系统的正确率是95.2%，由于异常资料很少，所以正确率仍然是很高的。所以**异常侦测问题中不会用正确率来直接当做评估指标**。

首先我们要知道在异常侦测中有两种错误：一种错误是异常的资料被判断为正常的资料，另外一种是正常的资料被判为异常的资料。假设我们将 λ 设为0.5（0.5以上认为是正常的资料，0.5以下认为是异常的资料），这时就可以计算机器在这两种错误上分别犯了多少错误。

对于所有异常的资料而言，有一笔资料被侦测出来，其余四笔资料没有被侦测为异常的资料。对于所有正常的资料而言，只有一笔资料被判断为异常的资料，其余的九十九笔资料被判断为正常的资料。这时我们会说机器有一个false alarm (正常的资料被判断为异常的资料) 错误，有四个missing (异常的资料却没有被侦测出来) 错误。

若我们将阀值 (threshold) 切在比0.8稍高的部分，这时会发现在五张异常的图片中，其中有两张认为是异常的图片，其余三种被判断为正常的图片；在一百张正确的图片中，其中有六张图片被认为是异常的图片，其余九十四张图片被判断为正常的图片。

哪一个系统比较好呢？其实你是很难回答这个问题。有人可能会很直觉的认为：当阀值为0.5时有五个错误，阀值为0.8时有九个错误，所以认为左边的系统好，右边的系统差。

但其实一个系统是好还是坏，取决于你觉得false alarm比较严重还是missing比较严重。

	Anomaly	Normal		Anomaly	Normal
Detected	1	1	Detected	2	6
Not Det	4	99	Not Det	3	94

Cost = 104

Cost = 401

Cost = 603

Cost = 306

Cost	Anomaly	Normal	Cost	Anomaly	Normal
Detected	0	100	Detected	0	1
Not Det	1	0	Not Det	100	0

Cost Table A

Cost Table B

Some evaluation metrics consider the ranking

For example, Area under ROC curve

所以你在做异常侦测时，可能有一个Cost Table告诉你每一种错误有多大的Cost。若没有侦测到资料就扣一分，若将正确的资料被误差为错误的资料就扣100分。若你是使用这样的Cost来衡量系统的话，左边的系统会被扣104分，右边的系统会被扣603分。所以你会认为左边的系统较好。若Cost Table为Cost Table B时，异常的资料没有被侦测出来就扣100分，将正常的资料被误判为错误的资料就扣1分，计算出来的结果会很不一样。

在不同的情景下、不同的任务，其实有不同的Cost Table：假设你要做癌症检测，你可能就会比较倾向想要用右边的Cost Table。因为一个人没有癌症却被误判为有癌症，顶多就是心情不好，但是还可以接受。若一个人其实有癌症，但没有检查出来，这时是非常严重的，这时的Cost也是非常的高。

这些Cost要给出来，其实是要问你现在是什么样的任务，根据不同的任务有不同的Cost Table。所以根据右边的Cost Table，左边的Cost为401分，右边的Cost为306分，所以这时右边的系统较好。

其实还有很多衡量异常检测系统的指标，有一个常用的指标为AUC (Area under ROC curve)。若使用这种衡量的方式，你就不需要决定阀值 (threshold)，而是看你将测试集的结果做一个排序 (高分至低分)，根据这个排序来决定这个系统好还是不好。

如果我们直接用一个分类器来侦测输入的资料是不是异常的，这并不是一种很弱的方法，但是有时候无法给你一个perfect的结果，我们用这个图来说明用classifier做异常侦测时有可能会遇到的问题。

假设现在做一个猫和狗的分类器，将属于的一类放在一边，属于狗的一类放在一边。若输入一笔资料即没有猫的特征也没有狗的特征，机器不知道该放在哪一边，就可能放在这个boundary上，得到的信息分数就比较低，你就知道这些资料是异常的。

你有可能会遇到这样的状况：有些资料会比猫更像猫（老虎），比狗还像狗（狼）。机器在判断猫和狗时是抓一些猫的特征跟狗的特征，也许老虎在猫的特征上会更强烈，狼在狗的特征上会更强烈。对于机器来说虽然有些资料在训练时没有看过（异常），但是它有非常强的特征会给分类器很大的信心看到某一种类别。

在解决这个问题之前我想说辛普森家庭人物脸都是黄的，如果用侦测辛普森家庭人物的classifier进行侦测时，会不会看到黄脸的人信心分数会不会暴增呢？所以将三玖的脸涂黄，结果侦测为是宅神，信心分数为0.82；若再将其头发涂黄，结果侦测为丽莎，信心分数为1.00。若将我的脸涂黄，结果侦测为丽莎，信心分数为0.88。

当然有些方法可以解这个问题，这里列一些文献给大家进行参考。其中的一个解决方法是：假设我们可以收集到一些异常的资料，我们可以教机器看到正常资料时不要只学会分类这件事情，要学会一边做分类一边看到正常的资料信心分数就高，看到异常的资料就要给出低的信心分数。

但是会遇到的问题是：很多时候不容易收集到异常的数据。有人就想出了一个神奇的解决方法就是：既然收集不到异常的资料，那我们就通过Generative Model来生成异常的资料。这样你可能遇到的问题是：若生成的资料太像正常的资料，那这样就不是我们所需要的。所以还要做一些特别的constraint，让生成的资料有点像正常的资料，但是又跟正常的资料又没有很像。接下来就可以使用上面的方法来训练你的classifier。

Case 2: Without Labels

Twitch Plays Pokémon

接下来我们再讲第二个例子，在第二个例子中我们没有classifier，我们只能够收集到一些资料，但没有这些资料的label

这是一个真实的Twitch Plays Pokemon例子，这个的例子是这样的：有人开了一个宝可梦的游戏，全世界的人都可以连接一起玩这个宝可梦的游戏。右边的框是每一个人都在输入指令同时操控这个游戏，这个游戏最多纪录好像是有八万人同时玩这个游戏。当大家都在同时操作同一个角色时，玩起来其实是相当崩溃的。

人们玩的时候就非常的崩溃，那么崩溃的原因是什么呢？可能是因为有网络小白（Troll）。有一些人根本就不会玩，所以大家都无法继续玩下去；或者觉得很有趣；或者是不知名的恶意，不想让大家结束这个游戏。人们相信有一些小白潜藏在人们当中，他们想要阻挠游戏的进行。

所以我们就用异常侦测的技术，假设多数的玩家都是想要破关的（训练资料），我们可以从多数玩家的行为知道正常玩家的行为是怎样的，然后侦测出异常的玩家（网络小白）。至于侦测出来给网路小白做什么，还需要待讨论的问题。有人说：小白只是人们的幻想，为什么这么说呢？

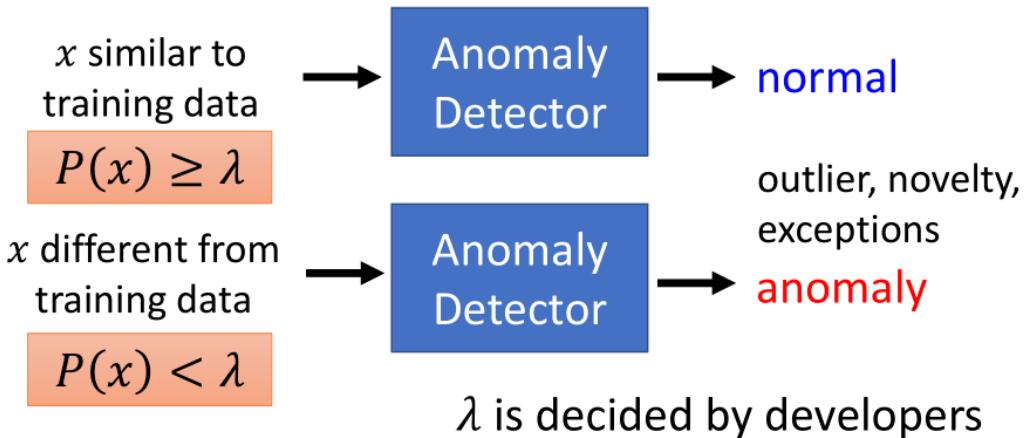
也许实际上根本就没有在阻挠这个游戏的进行，只是因为大家同时玩这个游戏时，大家的想法会是不一样的，这样玩起来会非常的卡。甚至有人说若没有网络小白，大家也无法玩下去，因为在这个游戏里面同时可能会有非常多的指令被输入，而系统pick一个指令，所以多数时你的指定根本就没有被选到。如果所有人的想法都是一致的（输入同一个指令），结果某一个人的指令被选到。那你可能就会觉得这有什么好玩的，反正我又没有操控那个角色。所以大家相信有网络小白的存在，大家一起联手起来抵挡网络小白的攻击，这会让你觉得最后系统没有选到我，但是至少降低了小白被选到的可能，所以大家可以继续玩下去。

我们需要一些训练的资料，每一个x代表一个玩家，如果我们使用machine learning的方法来求解这个问题，首先这个玩家能够表示为feature vector。向量的第一维可以是玩家说垃圾话的频率，第二维是统计玩家无政府状态发言。

Problem Formulation

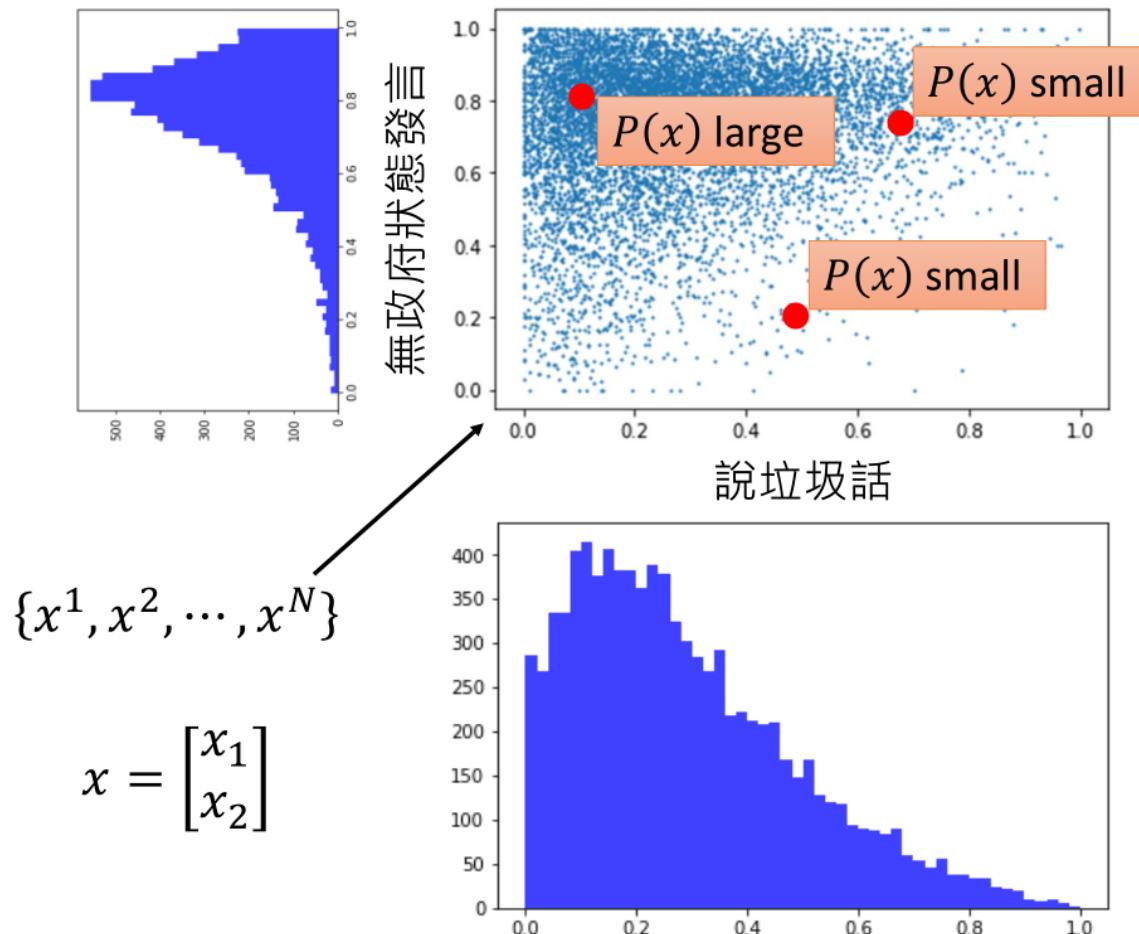
Generated from $P(x)$

- Given a set of training data $\{x^1, x^2, \dots, x^N\}$
- We want to find a function detecting input x is *similar* to training data or not.



我们刚刚可以根据分类器的conference来判断是不是异常的资料，但我们现在只有大量的训练资料，没有label。我们在没有classifier的情况下可以建立一个模型，这个模型是告诉我们 $P(x)$ 的机率有多大。（根据这些训练资料，我们可以找出一个机率模型，这个模型可以告诉我们某一种行为发生的概率多大）。如果有一个玩家的机率大于某一个阀值（threshold），我们就说他是正常的；如果机率小于某一个阀值（threshold），我们就说他是异常的。

如图这是一个真实的资料，假设每一个玩家可以用二维的向量来描述（一个是说垃圾话的机率，一个是无政府状态发言的机率）。



如果我们只看说垃圾话的那一维如图所示，会发现并不是完全不说垃圾话的是最多的。很多人可能会想说大多数人是在民主状态下发言的（民主时比较想发言，无政府混乱时不想发言），但是实际上统计的结果跟直觉的想象是不一样的，如图所示大多数人有八成的机率是在无政府状态下发言的，因为这个游戏多数时间是在无政府状态下进行的。游戏进行到某一个地方以后，官方决定加入民主机制（20秒内最多人选择的那一个行为是控制角色所采取的决策，这个听起来好像是很棒的主意，马上就遭到了大量的反对。在游戏里面要投票选择民主状态还是无政府状态，若很多人选择无政府状态，就会进入无政府状态。

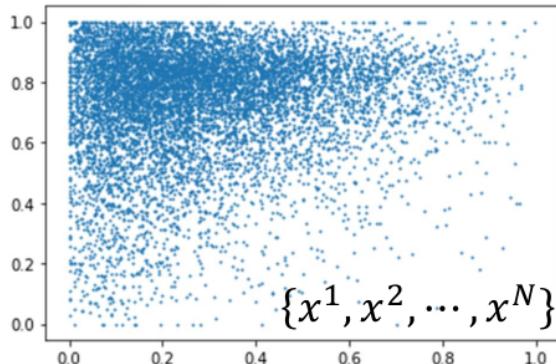
事实上很多人强烈支持无政府状态，强烈反对民主状态，所以这个游戏多数是在无政府状态下进行。假设一个玩家不考虑自己是要在什么状态下发言，大多数人有八成的机率是在无政府下进行发言，有人甚至觉得多数小白是在民主状态下发言，因为进入了民主状态，所以他要多发言才能够让他的行为被选中，所以小白会特别喜欢在民主状态下发言。

我们还没有讲任何的机率模型，从这个图上可以很直觉的看出一个玩家落在说垃圾的话机率低，通常在无政府状态下发言，这个玩家有可能是一个正常的玩家。假设有玩家落在有五成左右的机率说垃圾话，二成的机率在无政府状态下发言；或者落在有七成左右的机率说垃圾话，七成的机率在无政府状态下发言，显然这个玩家比较有可能是一个不正常的玩家。我们直觉上明白这件事情，但是我们仍然希望用一个数值化的方法告诉我们玩家落在哪里会更加的异常。

Maximum Likelihood

我们需要likelihood这个概念：我们收集到n笔资料，假设我们有一个probability density function $f_\theta(x)$ ，其中 θ 是这个probability density function的参数（ θ 的数值决定这个probability density function的形状）， θ 必须要从训练资料中学习出来。假设这个probability density function生成了我们所看到的数据，而我们所做的事情就是找出这个probability density function究竟长什么样子。

- Assuming the data points is sampled from a probability density function $f_\theta(x)$
 - θ determines the shape of $f_\theta(x)$
 - θ is unknown, to be found from data



$$L(\theta) = f_\theta(x^1)f_\theta(x^2) \cdots f_\theta(x^N)$$

Likelihood

$$\theta^* = \arg \max_{\theta} L(\theta)$$

likelihood的意思是：根据probability density function产生如图所示的数据机率有多大。若严格说的话， $f_\theta(x)$ 并不是机率，它的output是probability density；输出的范围也并不是(0,1)，有可能大于1。

x^1 根据probability density function产生的机率 $f_\theta(x^1)$ ，乘以 x^2 根据probability density function产生的机率 $f_\theta(x^2)$ ，一直乘到 x^N 根据probability density function产生的机率，得到的结果就是likelihood。

likelihood的可能性显然是由 θ 控制的，选择不同的 θ 就有不同的probability density function，就可以算出不同的likelihood。

而我们现在并不知道这个 θ 是多少，所以我们找一个 θ^* ，使得算出来的likelihood是最大的。

Gaussian Distribution

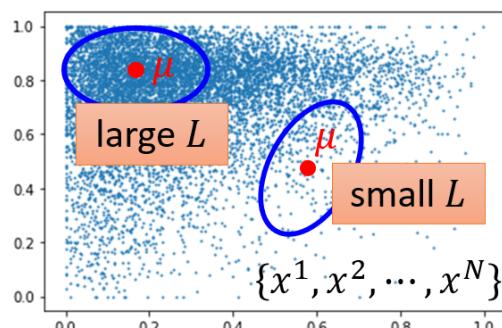
Gaussian Distribution

D is the dimension of x

$$f_{\mu, \Sigma}(x) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right\}$$

Input: vector x, output: probability of sampling x

θ which determines the shape of the function are **mean μ** and **covariance matrix Σ**



$$L(\theta) = f_\theta(x^1)f_\theta(x^2) \cdots f_\theta(x^N)$$

$$L(\mu, \Sigma) = f_{\mu, \Sigma}(x^1)f_{\mu, \Sigma}(x^2) \cdots f_{\mu, \Sigma}(x^N)$$

$$\theta^* = \arg \max_{\theta} L(\theta)$$

$$\rightarrow \mu^*, \Sigma^* = \arg \max_{\mu, \Sigma} L(\mu, \Sigma)$$

$$\mu^* = \frac{1}{N} \sum_{n=1}^N x^n = \begin{bmatrix} 0.29 \\ 0.73 \end{bmatrix}$$

$$\Sigma^* = \frac{1}{N} \sum_{n=1}^N (x - \mu^*)(x - \mu^*)^T = \begin{bmatrix} 0.04 & 0 \\ 0 & 0.03 \end{bmatrix}$$

第二项分母为 Determinant 行列式

一个常用的probability density function就是Gaussian Distribution，你可以将这个公式想象为一个function，这个function就是输入一个vector x ，输出为这个vector x 被sample到的机率。这个function由两个参数（ μ 和covariance matrix Σ ）所控制，它们相当于我们刚才所讲的 θ 。这个Gaussian function的形状由 μ 和covariance matrix Σ 所控制。将 θ 替换为 μ, Σ ，不同的 μ, Σ 就有不同的probability density function。

假设如图所示的数据是由左上角的 μ 来生成的，它的likelihood是比较大的（Gaussian function的特性就是在 μ 附近时data被sample的机率很高）。假设这个 μ 远离高密度，若从这个 μ 被sample出来的资料应该落在这个蓝色圈圈的范围内，但是资料没有落在这个蓝色圈圈的范围内，显然这样计算出来的likelihood是比较低的。

所以我们要做的事情就是穷举所有的 μ, Σ ，观察哪个 μ, Σ 计算的likelihood最大，那这个 μ, Σ 就是我们要找的 μ^*, Σ^* 。得到 μ^*, Σ^* 以后就可以知道产生这笔资料的Distribution的形状。

但是往往有同学问的问题是：为什么要用Gaussian Distribution，为什么不用其它的Distribution。最简答的答案是：我选别的Distribution，你也会问同样的问题。Gaussian是真的常用，这是一个非常非常强的假设，因为你的资料分布可能根本就不是Gaussian，有可能你会做的更好，但不是我们这门课要讲的范围。

如果 $f_\theta(x)$ 是一个非常复杂的function（network），而操控这个network的参数有非常大量，那么就不会有那么强的假设了，就会有多的自由去选择function来产生资料。这样就不会限制在看起来就不像Gaussian产生的资料却硬要说Gaussian产生的资料。因为我们这门课还没有讲到其它进阶的生成模型，所以现在用Gaussian Distribution来当做我们资料是由Gaussian Distribution所产生的。

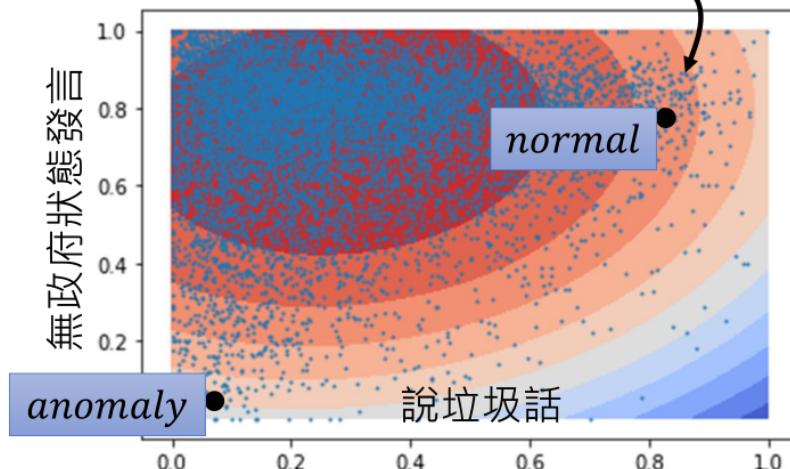
μ^*, Σ^* 可以代入相应的公式来解这个公式， μ^* 等于将所有的training data x 做平均，结果为 $\begin{bmatrix} 0.29 \\ 0.73 \end{bmatrix}$ ， Σ^* 等于将 $x - \mu^*$ 乘以 $x - \mu^*$ 的转置，然后做平均，得到的结果为 $\begin{bmatrix} 0.04 & 0 \\ 0 & 0.03 \end{bmatrix}$

$$f_{\mu^*, \Sigma^*}(x) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma^*|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu^*)^T \Sigma^{*-1} (x - \mu^*) \right\}$$

$$\mu^* = \begin{bmatrix} 0.29 \\ 0.73 \end{bmatrix} \quad \Sigma^* = \begin{bmatrix} 0.04 & 0 \\ 0 & 0.03 \end{bmatrix}$$

$f(x) = \begin{cases} \text{normal}, & f_{\mu^*, \Sigma^*}(x) > \lambda \\ \text{anomaly}, & f_{\mu^*, \Sigma^*}(x) \leq \lambda \end{cases}$	λ is a contour line
---	-----------------------------

The colors represents the value of $f_{\mu^*, \Sigma^*}(x)$

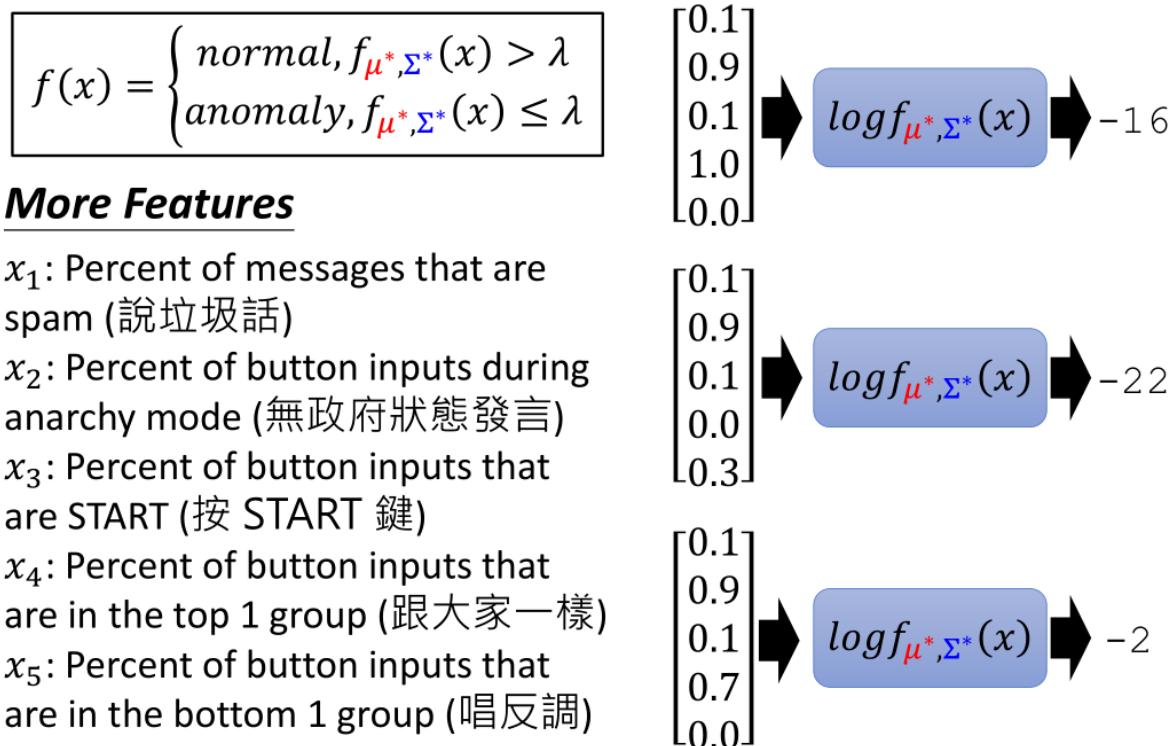


我们根据如图所示的资料找出了 μ^* 和 Σ^* ，接下来就可以做异常侦测了。将 μ^*, Σ^* 代入probability density function，若大于某一个阈值（threshold）就说明是正常的，若小于某一个阈值（threshold）就说明是异常的。

每一笔资料都可以代入probability density function算出一个数值，结果如图所示。若落在颜色深的红色区域，就说明算出来的数值越大，越是一般的玩家，颜色浅的蓝色区域，就说明这个玩家的行为越异常。其中 λ 就是如图所示的等高线的其中一条，若有一个玩家落在很喜欢说垃圾话，多数喜欢在无政府状态下发言的区域，就说明是一个正常的玩家。若有一个玩家落在很少说垃圾话，特别喜欢在民主时发言，就说明是一个异常的玩家。

More Features

machine learning最厉害的就是让machine做，所以你要选择多少feature都可以，把能想到觉得跟判断玩家是正常的还是异常的feature加进去。



有了这些feature以后就训练训练出 μ^* , Σ^* ，然后创建一个新的玩家代入这个function，就可以知道这个玩家算出来的分数有多高（对这个function进行log变化，因为一般function计算出来的分数会比较小）。

假设输入的这个玩家有0.1 percent说垃圾话，0.9 percent无政府状态下发言，0.1 percent按START键，1.0 percent跟大家一样，0.0 percent唱反调，这个玩家计算出来的likelihood为-16。

假设输入的这个玩家有0.1 percent说垃圾话，0.9 percent无政府状态下发言，0.1 percent按START键，0.0 percent跟大家一样，0.3 percent唱反调，这个玩家计算出来的likelihood为-22。

假设输入的这个玩家有0.1 percent说垃圾话，0.9 percent无政府状态下发言，0.1 percent按START键，0.7 percent跟大家一样，0.0 percent唱反调，这个玩家计算出来的likelihood为-2。

我们可以看到第一个和第三个玩家除了第四个特征都一样，但是第一个玩家和大家的选择完全一样，第三个玩家和大家的选择在大多数情况下是相同的，这时第一个得到的分数反而低，是因为机器会觉得如果你和所有人完全一样这件事就是很异常的。

Outlook: Auto-encoder

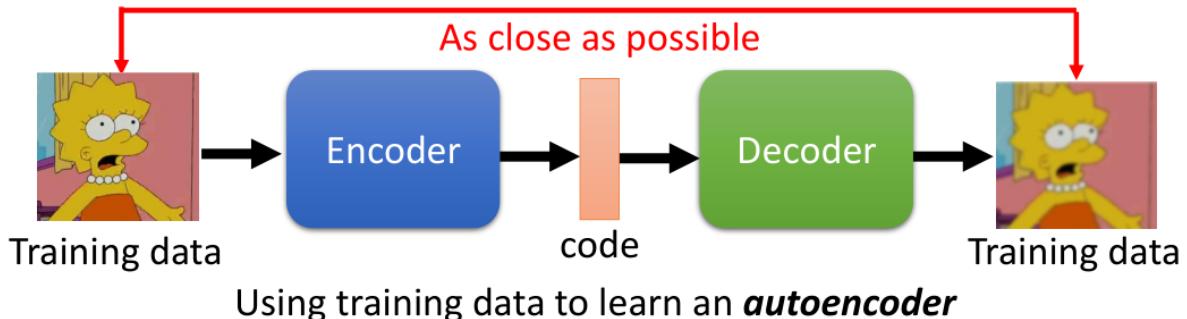
上述是用生成模型 (Generative Model) 来进行异常侦测这件事情，我们也可以使用Auto-encoder来做这件事情。

我们把所有的训练资料训练一个Encoder，Encoder所做的事情是将输入的图片（辛普森）变为code（一个向量），Decoder所做的事情是将code解回原来的图片。训练时Encoder和Decoder是同时训练，训练目标是希望输入和输出越接近越好。

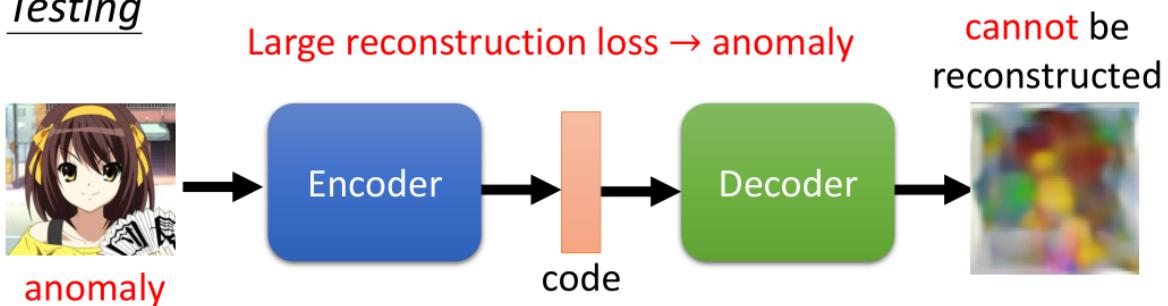
测试时将一个图片输入Encoder，Decoder还原原来的图片。如果这张图片是一个正常的照片，很容易被还原为正常的图片。因为Auto-encoder训练时输入的都是辛普森家庭的图片，那么就特别擅长还原辛普森家庭的图片。

但是若你输入异常的图片，通过Encoder变为code，再通过Decoder将code解回原来的图片时，你会发现无法解回原来的图片。解回来的图片跟输入的图片差很多时，这时你就可以认为这是一张异常的图片。

Training



Testing

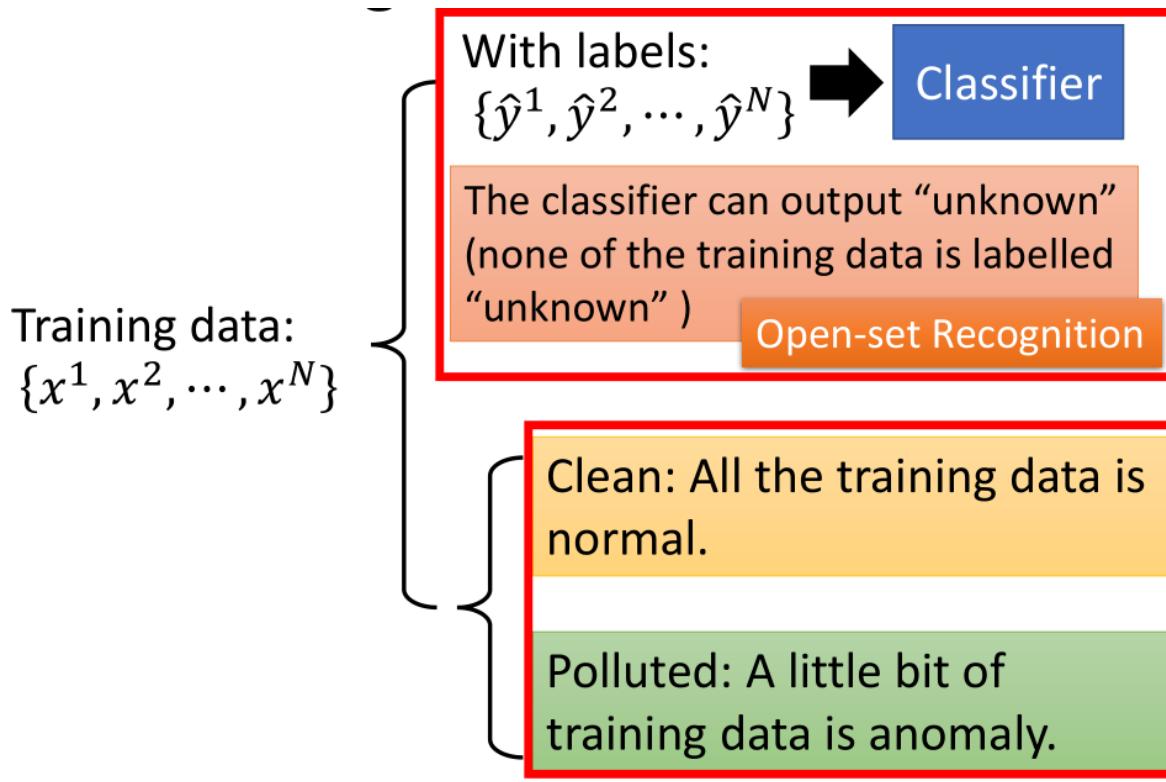


More ...

machine learning中也有其它做异常侦测的方法，比如One-class SVM，只需要正常的资料就可以训练SVM，然后就可以区分正常的还是异常的资料。

Isolated Forest，它所做的事情跟One-class SVM所做的事情很像（给出正常的训练进行训练，模型会告诉你异常的资料是什么模样）。

Concluding Remarks



Generative Adversarial Network

Generative Adversarial Network

Three Categories of GAN

Typical GAN

Typical GAN要做的事情是找到一个Generator，Generator就是一个function。这个function的输入是random vector，输出是我们要这个Generator生成的图片。

举例来说：假设要机器做一个动画的Generator，那么就要收集很多动画人物的头像，然后将这些动画人物的头像输入至Generator去训练，Generator就会学会生成二次元人脸的图像。

那怎么训练这个Generator 呢，模型的架构是什么样子的呢？

最基本的GAN就是对Generator输入vector，输出就是我们要它生成的东西。我们以二次元人物为例，假设我们要机器画二次元人物，那么输出就是一张图片（高维度的向量）。Generator就像吃一个低维度的向量，然后输出一个高维度的向量。

GAN有趣的地方是：我们不只训练了Generator，同时在训练的过程中还会需要一个Discriminator。Discriminator的输入是一张图片，输出是一个分数。这个分数代表的含义是：这张输入的图片像不像我们要Generative产生的二次元人物的图像，如果像就给高分，如果不像就给低分。

Algorithm

接下来要讲的是Generator和Discriminator是怎样训练出来的，Generator和Discriminator都是neuron network，而它们的架构是什么样的，这取决于想要做的任务。举例来说：你要generator产生一张图片，那显然generator里面有很多的deconvolution layer。若要generator产生一篇文章或者句子，显然要使用RNN。

我们今天不讲generator跟discriminator的架构，这应该是取决于你想要做什么样的事情。

generator跟discriminator是某种network，训练neuron network时要随机初始化generator和discriminator的参数。

在初始化参数以后，在每一个training iteration要做两件事

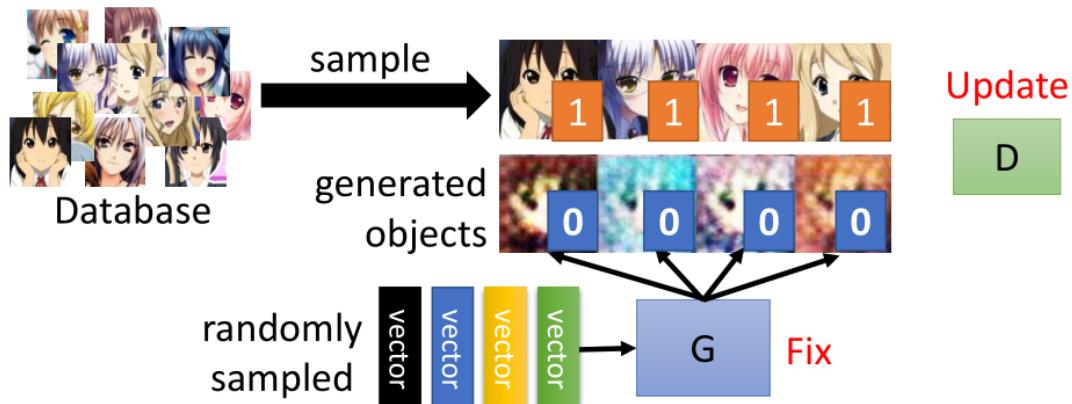
step1: 固定generator，然后只训练discriminator（看到好的图片给高分，看到不好的图片给低分）。从一个资料库（都是二次元的图片）随机取样一些图片输入 discriminator，对于discriminator来说这些都是好的图片。因为generator的参数都是随机给定的，所以给generator一些向量，输出一些根本不像二次元的图像，这些对于discriminator来说就是不好的图片。接下来就可以教discriminator若看到上面的图片就输出1，看到下面的图片就输出0。训练discriminato的方式跟我们一般训练neuron network的方式是一样的。

Algorithm

- Initialize generator and discriminator
- In each training iteration:



Step 1: Fix generator G, and update discriminator D



Discriminator learns to assign high scores to real objects and low scores to generated objects.

有人会把discriminator当做回归问题来做，看到上面的图片输出1，看到下面的图片就输出0也可以。有人把discriminator当做分类的问题来做，把上面好的图片当做一类，把下面不好的图片当做另一类也可以。训练discriminator没有什么特别之处，跟我们训练neuron network或者binary classifier是一样的。唯一不同之处就是：假设我们用训练binary classifier的方式来训练discriminator时，不一样的就是binary classifier其中class的资料不是人为标注好的，而是机器自己生成。

step2: 固定住discriminator，只更新generator。

一般我们训练network是minimize 人为定义的loss function，在训练generator时，generator学习的对象不是人定的loss function/objective function，而是discriminator。你可以认为discriminator就是定义了某一种loss function，等于机器自己学习的loss function。

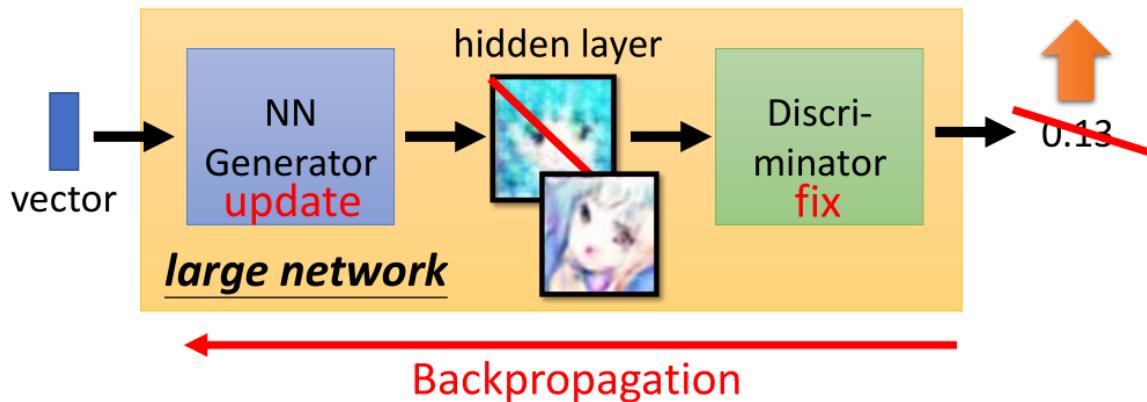
generator学习的目标就是为了骗过discriminator，我们让generator产生一些图片，在将这些图片输入进discriminator，然后discrimination就会给出这些图片一些分数。

接下来我们把generator和discriminator串在一起视为一个巨大的network。这个巨大的network输入是一个向量，输出是一个分数，在这个network中间hidden layer的输出可以看做是一张图片。

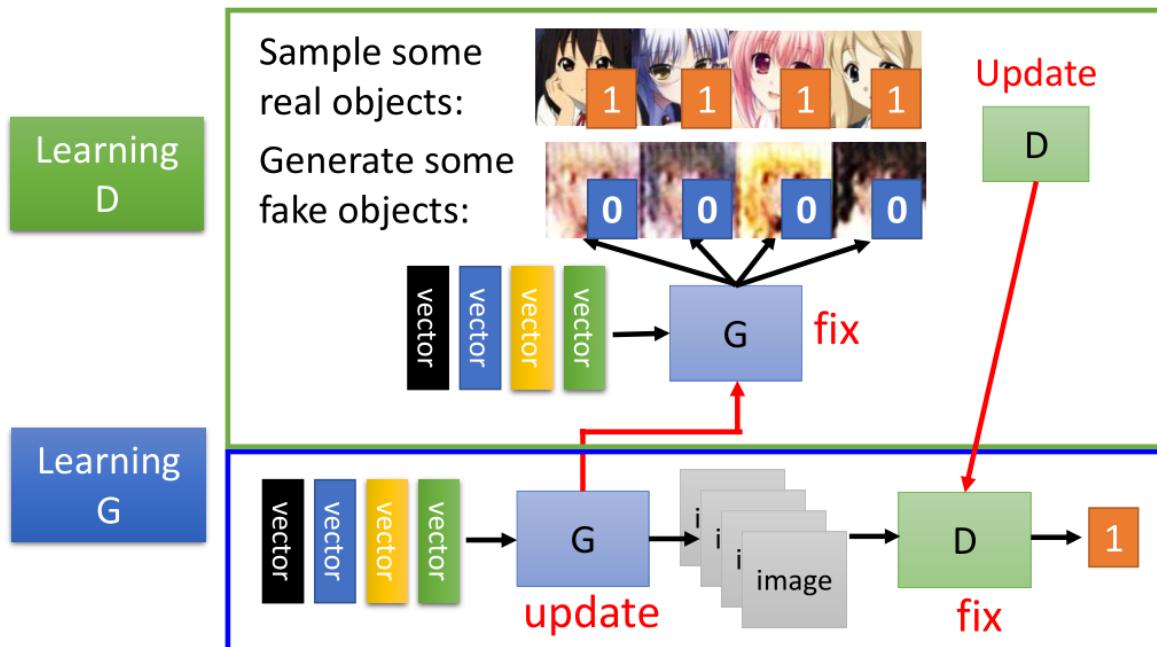
我们训练的目标是让输出越大越好，训练时依然会做backpropagation，只是要固定住discriminator，只是调整generator。调整完generator后输出会发生改变，generator新的输出会让discriminator给出高的分数。

Step 2: Fix discriminator D, and update generator G

Generator learns to “fool” the discriminator



实际上在训练时，训练discriminator一次，然后训练一次generator（固定discriminator），接下来由新的generator产生更多需要被标为0的图片，再去调discriminator，再调generator.....generator和discriminator交替训练，就做二次元人物的生成。网络上最好的二次元人物生成的结果。



GAN is hard to train...

众所周知，GAN这个技术是比较难train起来的，所以有很多人提出了更好的训练GAN的方法，WGAN, improve WGAN...

Conditional GAN

刚才是让机器随机的产生一些东西，这些不见得是我们想要的。我们更多的想要控制机器产生出来的东西。

我们可以训练一个Generator，这个generator的输入是一段文字，输出是这段文字对应的图像。举例来说：我们现在输入“Girl with red hair”，它就输出一个。根据某一个输入产生对应输出的generator被叫做Conditional GAN。

Text-to-Image

Traditional supervised approach

现在用文字产生影像作为示例，如何可以训练一个network根据文字来产生图像，最直觉的方法就是使用supervised learning。假设可以收集到文字跟影像之间的对应关系（这些图片有人标识每张图片对应的文字是什么），接下来就可以完全的套用传统的supervised learning的方法。直接训练一个network，它的输入是一段文字，输出是一张图像，希望这个输出跟原始的图像越接近越好。可以用这种方法直接训练，看到文字产生图像。

过去用这种方法（看到文字产生图像）来训练，训练出来的结果并不太好。为什么呢？举例说明：假设要机器学会画火车，但是训练资料里面有很多不同形态的火车，当network输入火车时它的正确答案有好几个，对于network来说会产生它们的平均作为输出。如果用supervised learning的方法产生出来的图像往往是非常模糊。

Conditional GAN

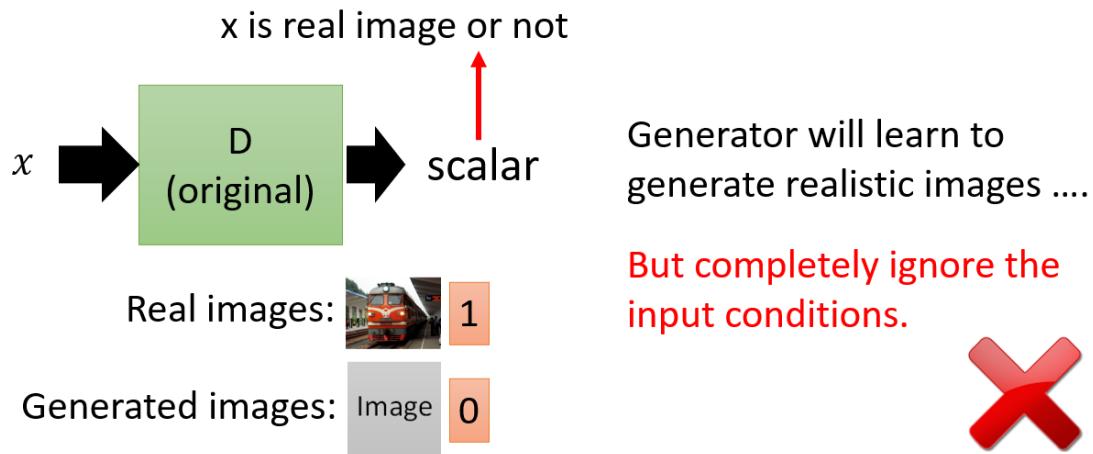
所以我们需要新的技术（Conditional GAN）来做根据文字产生图像，对于Conditional GAN，我们也需要文字与图像的对应关系（supervised learning），但是它跟一般的supervised learning的训练目标是不一样的。事实上Conditional GAN也可以在unsupervised learning的情况下进行训练，后面的内容会涉及到。

Conditional GAN是跟着discriminator来进行学习的，要注意的是：在做Conditional GAN时跟一般的GAN是不一样的。

在第一部分讲一般的GAN时，discriminator是输入一张图片，然后判断它好还是不好。现在在Conditional GAN的情况下，discriminator只输入一张图片会遇到的问题是：对于discriminator想要骗过generator太容易了，它只要永远产生好的图像就行了。举例来说：永远产生一只很可爱的猫就可以骗过discriminator（对于discriminator来说那只猫是好的图片），然后就结束了。Generator就会学到不管现在输入什么样的文字，就一律忽视，都产生猫就好了，这显然不是我们想要的。



[Scott Reed, et al, ICML, 2016]



在进行Conditional GAN时往往有两个输入，discriminator应该同时看generator的输入（文字）和输出（图像），然后输出一个分数，这个分数同时代表两个含义。第一个含义是两个输入有多匹配，第二个含义是输入图像有多好。

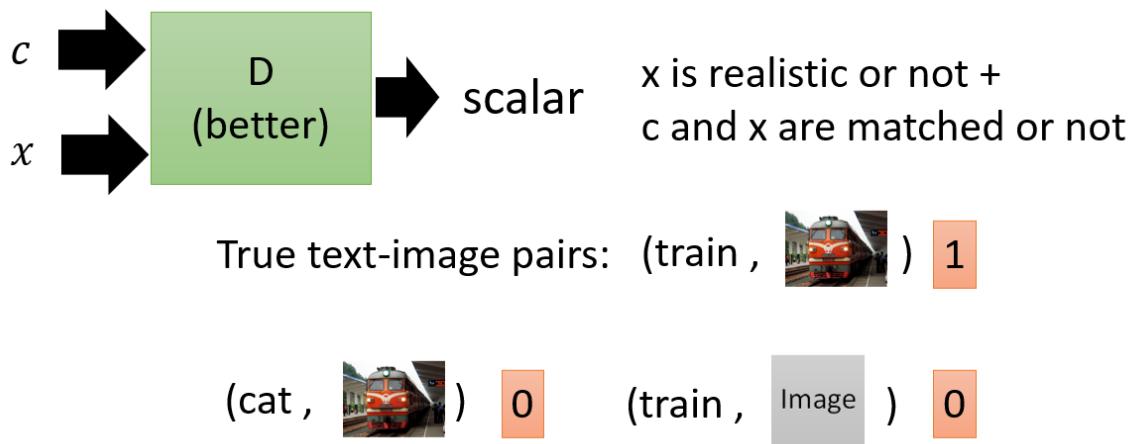
训练discriminator时要给它一些好的输入，这些是要给高分的。可以从datasets里面sample出文字与图像，告诉discriminator看到这些文字和图像应该给出高分。

对于另外一个case而言，什么情况会给出低分呢？

按照我们在第一部分讲一般GAN的想法，我们可能就是把文字输入generator，让他产生一些图片，这些文字和generator产生出来的图像要给予低分。

光是这么做discriminator会学到判断现在的输入是好还是不好，不管文字的部分只管图像的部分，这显然不是我们想要的。

所以在做Conditional GAN时，要给低分的case是要有两种。一个是跟一般的GAN一样是用generator生成图片；另外一个是从资料库里面sample出一些好的图片，但是给这些好的图片一些错误的文字。这时discriminator就会学到：并不是所有好的图片都是对的，如果好的图片对应都错误的文字它也是不好的。这样discriminator就会学懂文字与图像之间应该要有什么样的关系。



Sound-to-image

其实上述使用Conditional GAN根据文字生成影像的应用已经满坑满谷了，其实只要你有pair data你都可以尝试使用Conditional GAN，这里实作了一个例子，训练了一个Generator，输入声音，可以输出对应的画面。

训练Conditional GAN必须要有pair data，影像跟声音的对应关系其实并不难收集，我们可以收集到很多的video，将video中的audio部分拿出来就是声音，将image frame部分拿出来就是image，这样就有了pair data，就可以训练network。举例来说：听到狗叫声，就画一只狗出来。

当听到第一行第一列声音时，机器觉得它是一条小溪；听到第二行第一列的声音时，机器觉得它是在海上奔驰的快艇；我现在担心机器是不是背了一些资料库里面的图片，并没有学会声音跟影像之间的关系，所以我们把声音调大，然后就产生了一条越来越大的瀑布。我们把快艇的声音调大，产生的就是快艇在海上快速的奔驰，水花就越来越多。机器有时候产生的结果也会非常的差。

Image-to-label

我们刚才尝试了用文字产生影像，现在反过来想，用影像来产生文字。我们将影像用在Multi-label image Classifier上，给机器看一张图片，让机器告诉我们图片有哪些物件，比如有球，球棒等等，正确答案不只有一个。我们在课堂里讲的分类问题，每一个输入都只属于每一个类别。但是在Multi-label image classifier的情况下，同一张图片可以同时属于不同的类别。

一张图片可以同时属于不同类别这件事，我们可以想成是一个生成的问题，现在Multi-label image Classifier是一个Conditional Generator，它的输入是一张图片（图片是condition），label是Generator输出。接下来就当做Conditional GAN进行训练。

	F1	MS-COCO	NUS-WIDE
The classifiers can have different architectures.	VGG-16	56.0	33.9
The classifiers are trained as conditional GAN.	+GAN	60.4	41.2
	Inception	62.4	53.5
	+GAN	63.8	55.8
Conditional GAN outperforms other models designed for multi-label.	Resnet-101	62.8	53.1
	+GAN	64.0	55.4
	Resnet-152	63.3	52.1
	+GAN	63.9	54.1
[Tsai, et al., submitted to ICASSP 2019]	Att-RNN	62.1	54.7
	RLSD	62.0	46.9

这些是一些实验的结果，其中使用F1 score来当做评价指标（分数越高代表分类越正确）。我们试了不同的architecture（从VGG-16到Resnet-152），假设现在不是用一般的训练法（nn输出和ground truth越接近越好），而是用Conditional GAN的方法时，你会发现在不同的network架构下，结果都是比较好的。为什么加上GAN的方法会比较好呢？因为加上GAN以后会考虑label和label之间的dependence。

Talking Head

根据一张图片跟人脸landmarks去产生另外一张人脸，也就是说你现在可以做：给它一张蒙娜丽莎的脸，然后在画一些人脸的landmarks，这样你就可以让蒙娜丽莎开始讲话。

<https://arxiv.org/abs/1905.08233>

Unsupervised Conditional GAN

刚才在讲Conditional GAN时我们需要输入和输出之间的对应关系，但是事实上有机会在不知道输入和输出之间的对应关系的情况下，可以教机器怎样将输入的内容转化为输出。这个技术最常见到的应用是风格转化。

Cycle GAN

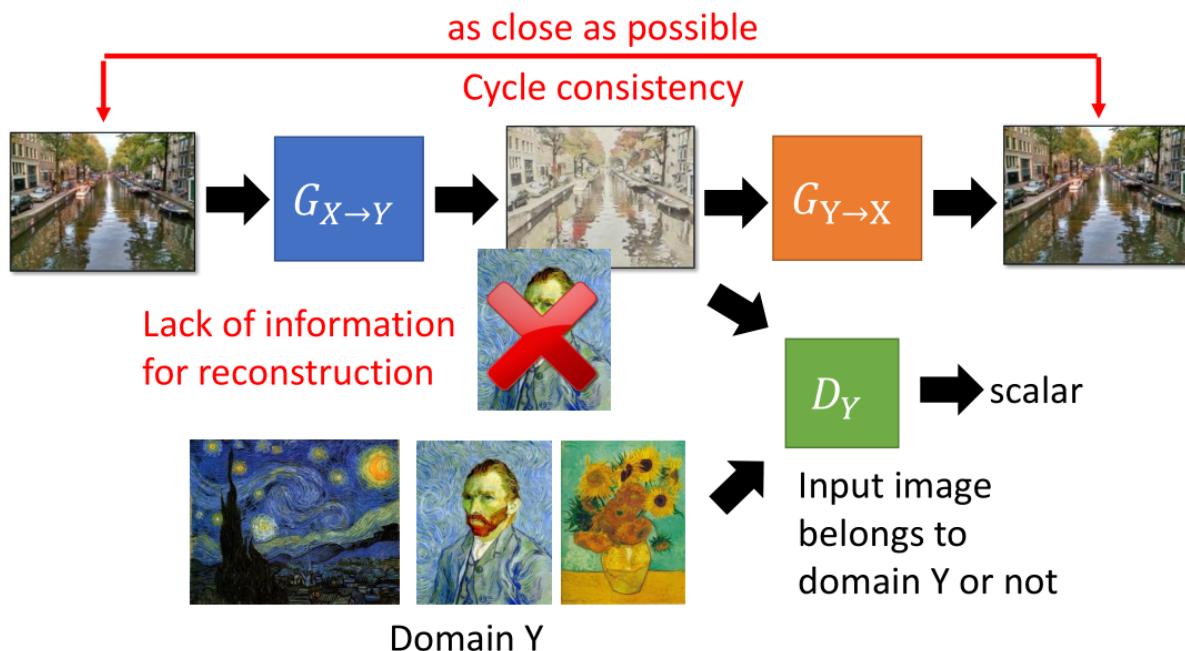
Cycle GAN想要做的事情是：训练一个generator，输入domain X（真实的画作），输出是梵高的画作。除了训练一个generator还需要训练一个discriminator，discriminator做的事情是：看很多梵高的画作，看到梵高的画作就给高分，看到不是梵高的画作就给低分。

generator为了要骗过discriminator，那我们期待产生出来的照片像是梵高的画作，光是这样做是不够的，因为generator会很快发现discriminator看的就是它的输出，那么generator就直接产生梵高的画作，完全无视输入的画作，只要骗过discriminator整个训练就结束了。

为了解决这个问题我们还需要再加上一个generator，这个generator要做的事情是根据第一个generator的输出还原原来的输入（输入一张Domain X，第一个generator将其转化为Domain Y的图，第二个generator在将其还原为Domain X，两者越接近越好）

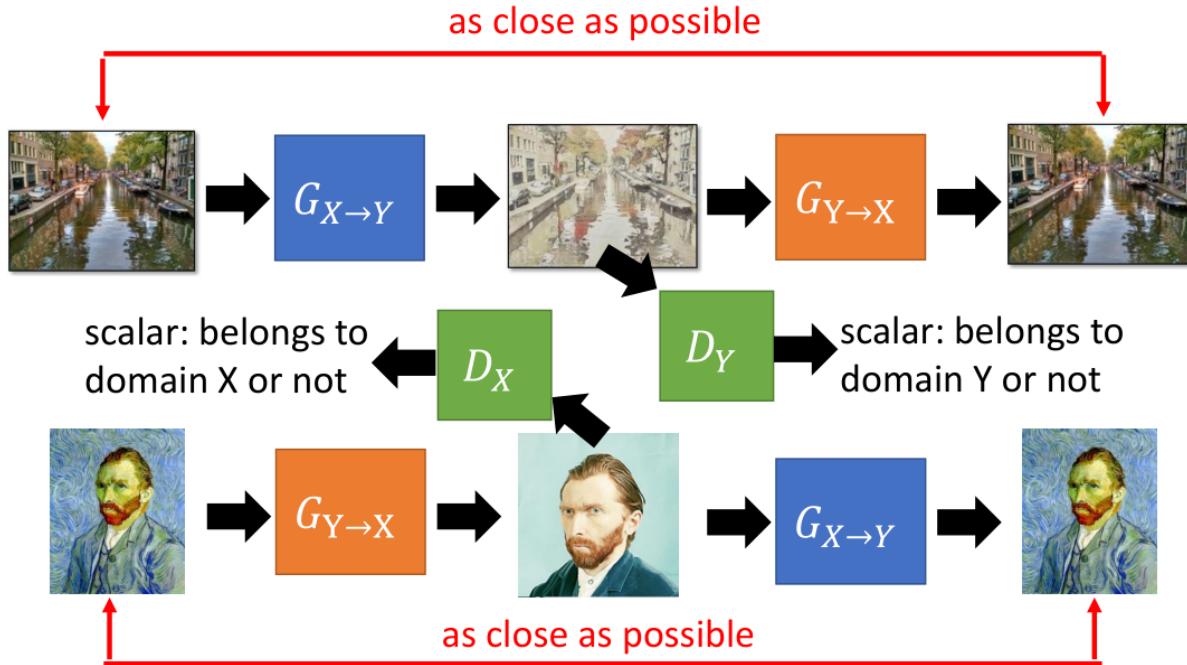
[Jun-Yan Zhu, et al., ICCV, 2017]

Cycle GAN



现在加上了这个限制，第一个generator就不能够尽情的骗过discriminator，不能够直接产生梵高的自画像。如果直接转换为梵高的自画像，那么第二个generator就无法将梵高的自画像还原。所以第一个generator想办法将Domain Y，但是原来图片最重要的资讯仍然被保留了下来，那么第二个generator才会有办法将其还原。输出跟输入越接近越好，这件事叫做Cycle consistency。

Cycle GAN其实是可以双向的，我们刚才讲的是先将Domain X转换为Domain Y，再将Domain Y转换为Domain X。现在我们可以将Domain Y转换为Domain X，然后再用一个蓝色的generator将Domain X转换为Domain Y，让输出和输入越接近越好。



同样的技术不只是用在影像上，可以应用在其它领域上。举例来说：假设Domain X和Domain Y分别是不同人讲的话（语音），那就可以做语音的风格转换。假设Domain X和Domain Y是两种不同的风格文字，那就可以做文字的风格转化。假设我们把Domain X替换成负面的句子，将Domain Y换成正面的句子，我们就可以训练一个generator，可以将负面的句子转换为正面的句子。

如果直接将影像的技术套用到文字上是会有些问题，如果generator在做文字风格转换的时候，输入是一个句子，输出也是一个句子。如果输入和输出分别是句子的话，这时应该使用Seq2seq model作为网络架构。在训练时仍然很期待使用Backpropagation将Seq2seq和discriminator串在一起，然后使用backpropagation固定discriminator，只训练generator，希望输出的分数越接近越好。

但在文字上没有办法直接使用bachpropagation，因为Seq2seq model输出的是离散的seq，是一串token。原来我们将generator的输看做是一个巨大network的hidden layer，那是因为在影像上generator的输出是连续的，但是在文字上generator的输出是离散的（不能微分）。

如何解决呢，在文献上有各式各样的解决办法。

Three Categories of Solutions

Gumbel-softmax: [Matt J. Kusner, et al, arXiv, 2016]

Continuous Input for Discriminator: [Sai Rajeswar, et al., arXiv, 2017][Ofir Press, et al., ICML workshop, 2017][Zhen Xu, et al., EMNLP, 2017][Alex Lamb, et al., NIPS, 2016][Yizhe Zhang, et al., ICML, 2017]

Reinforcement Learning: [Yu, et al., AAAI, 2017][Li, et al., EMNLP, 2017][Tong Che, et al, arXiv, 2017][Jiaxian Guo, et al., AAAI, 2018][Kevin Lin, et al, NIPS, 2017][William Fedus, et al., ICLR, 2018]

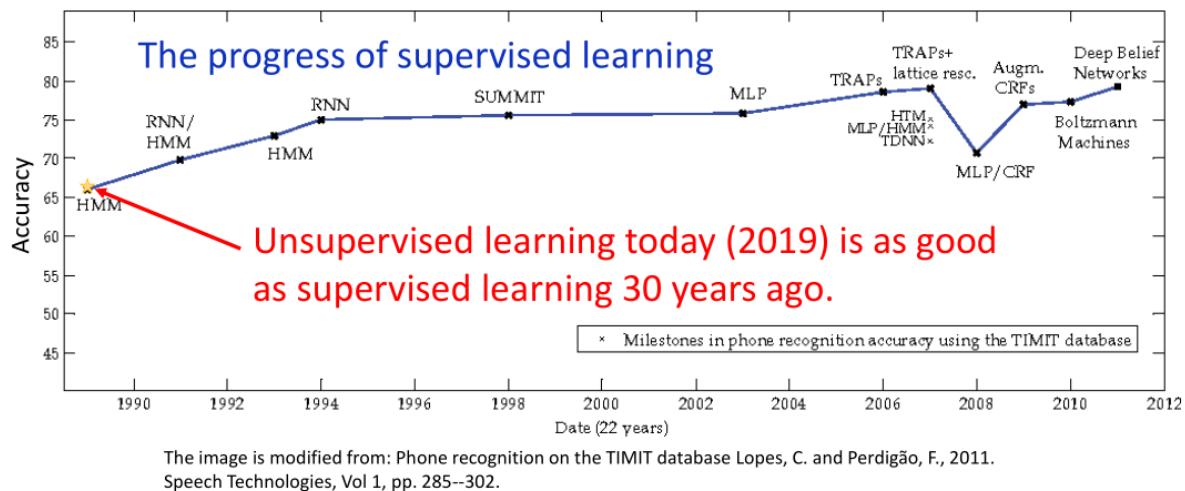
Speech Recognition

Unsupervised Conditional GAN其实除了风格转换以外还可以做其它的事情，比如说：Unsupervised 语音辨识。我们在做语音辨识时通常是supervised，也就是说我们要给机器一大堆的句子，还要给除句子对应的文字是什么，这样的句子收集上万小时，然后期待机器自动学会语音辨识。但是世界上语言有7000多种，其实不太可能为每一种语言都收集训练资料。

所以能不能想象语音辨识能不能是Unsupervised的，也就是说我们收集一大堆语言，一大堆文字，但是我们没有收集语言跟文字之间的对应关系。机器做的就是听一大堆人讲话，然后上面读一大堆文章，然后期待机器自动学会语音辨识。

现在有很多人很笼统的问一个问题：语音辨识可以做到什么样的错误率、正确率，这个问题我都是没有办法去回答的，这个问题就好像是你的数学会考多少分，都没有说是考哪一门数学。所以你今天要问一个语言辨识系统的正确率有多少，你应该问是拿什么样的资料去测试它。

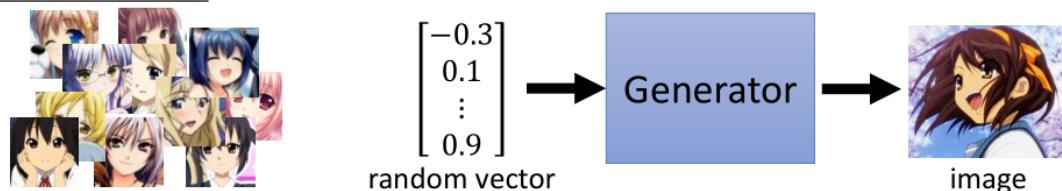
这张图讲的是supervised learning过去变化的情形（纵轴是正确率，横轴是时间），Unsupervised learning（没有提供给机器文字与语音之间的对应关系）可以做到跟三十年前supervised learning的结果一样好。



Concluding Remarks

Three Categories of GAN

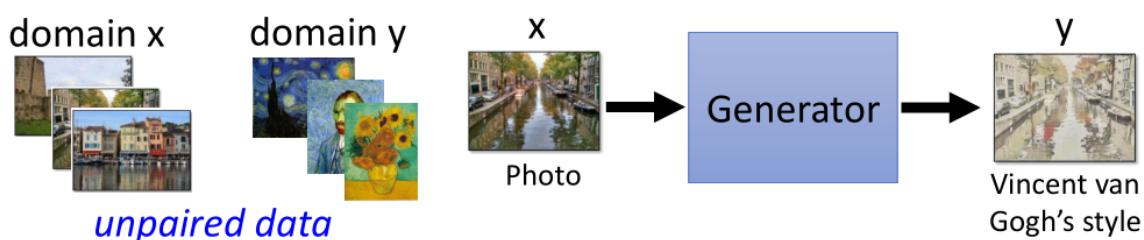
1. Typical GAN



2. Conditional GAN



3. Unsupervised Conditional GAN



Introduction of Generative Adversarial Network

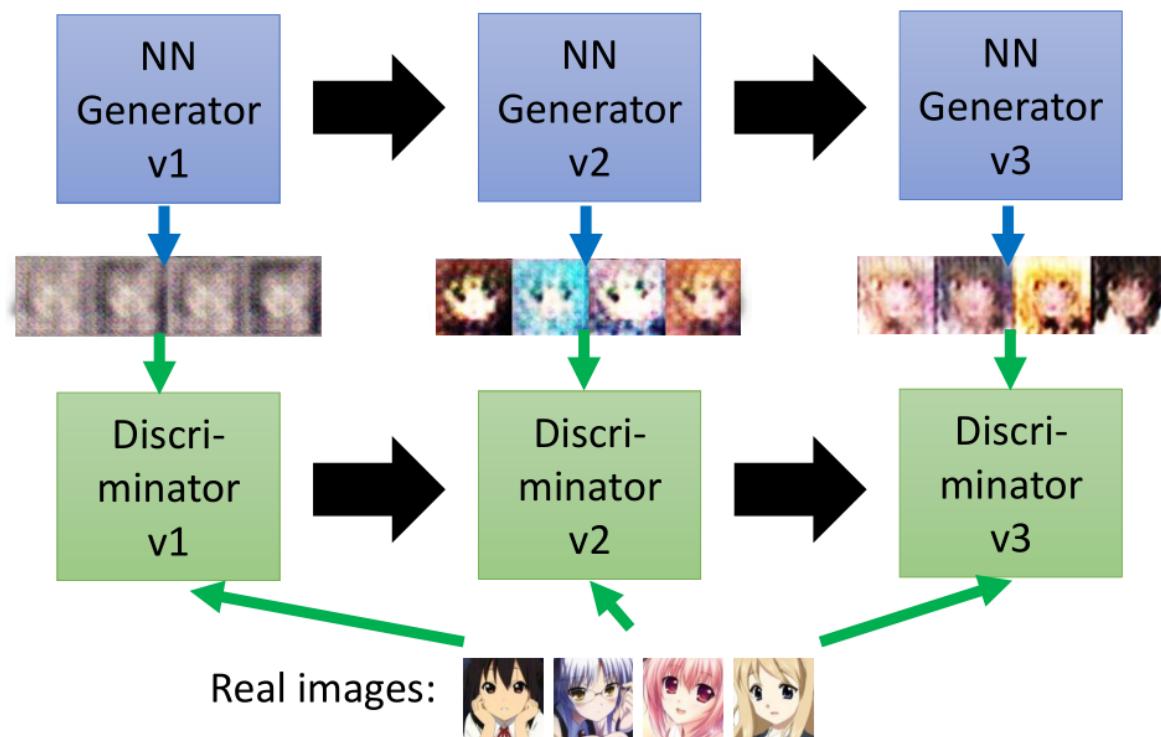
Basic Idea of GAN

Generator v.s. Discriminator

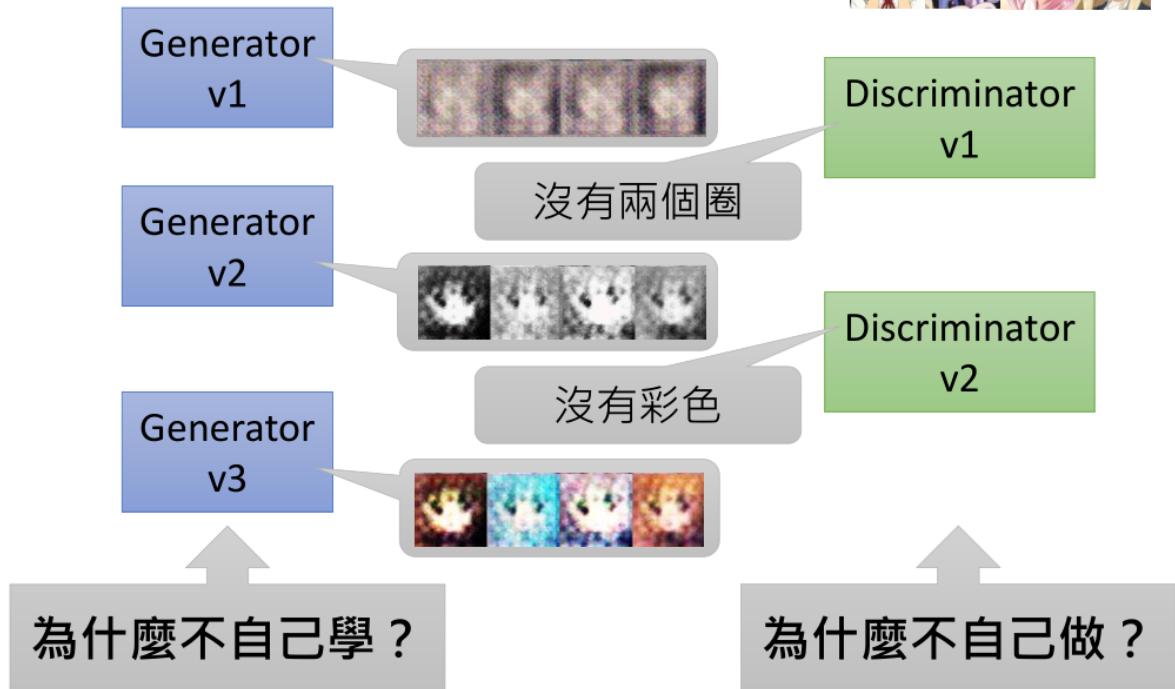
名为敌人，实为朋友

Basic Idea of GAN

This is where the term
“*adversarial*” comes from.
You can explain the process
in different ways.....



Basic Idea of GAN (和平的比喻)



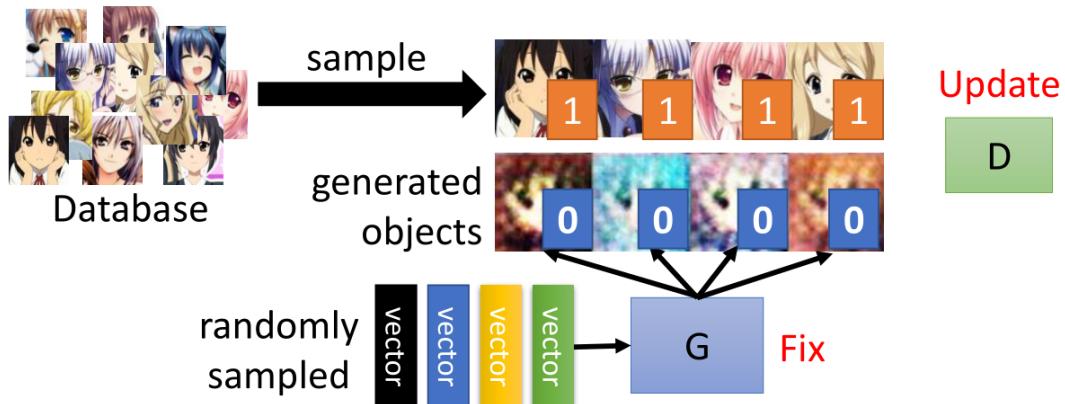
Algorithm

Algorithm

- Initialize generator and discriminator
- In each training iteration:

G D

Step 1: Fix generator G, and update discriminator D



Discriminator learns to assign high scores to real objects and low scores to generated objects.