



DEEP LEARNING

Author: Zixuan Xia

目 录

1 深度学习介绍	1
1.1 基本概念简介	1
1.1.1 模型	2
1.1.2 损失函数	2
1.1.3 优化	3
1.2 神经网络与深度学习	4
1.2.1 全连接神经网络	4
1.2.2 深度神经网络与特征工程	5
1.2.3 反向传播	6
1.3 深度学习总体攻略	7
1.3.1 模型偏差	8
1.3.2 优化问题	8
1.3.3 过拟合	9
1.3.4 验证集与交叉验证	10
1.3.5 深度学习总体流程	11
2 线性神经网络	12
2.1 线性回归	12
2.1.1 最简单的回归——线性回归	12
2.1.2 从线性回归到多项式回归	13
2.1.3 解决过拟合的办法	14
2.2 线性分类	16
2.2.1 分类问题的输入	16
2.2.2 分类问题的步骤	17
2.2.3 探究：为什么分类问题也是线性模型	21
2.3 逻辑回归（Logistic Regression）	22
2.3.1 逻辑回归的步骤	22
2.3.2 逻辑回归 v.s 线性回归	23
2.3.3 多分类问题（Multi-class Classification）	24
2.3.4 从逻辑回归到深度学习	25
3 卷积神经网络（Convolutional Neural Network）	27
3.1 背景	27
3.2 卷积神经网络如何减少参数？	28
3.2.1 感受野(Receptive Field)	28
3.2.2 步长(stride)和补值(padding)	29
3.2.3 参数共享与滤波器(Filter)	29
3.3 卷积神经网络的计算和结构	30
3.3.1 从滤波器到卷积层计算	30
3.3.2 池化层(Pooling Layer)	31

3.3.3 卷积神经网络的架构	32
3.4 卷积神经网络的应用与优缺点	33
3.4.1 最知名的应用——Alpha Go	33
3.4.2 卷积神经网络的利与弊	34
3.5 从 LeNet 到现代卷积神经网络	34
3.5.1 卷积神经网络的“开山之作”——LeNet	35
3.5.2 深度卷积神经网络 (AlexNet)	36
3.5.3 使用块的网络 (VGG)	38
3.5.4 含并行连接的网络 (GoogLeNet)	39
3.5.5 残差网络 (ResNet)	40
4 神经网络训练指南	44
4.1 训练前——我们应该做哪些准备？	44
4.1.1 激活函数的选择	44
4.1.2 数据预处理	47
4.1.3 权重初始化	48
4.2 训练中——梯度下降如何做得更好？	50
4.2.1 局部最小值(local minima) v.s. 鞍点(saddle point)	50
4.2.2 批次(Batch) v.s. 动量(Momentum)	51
4.2.3 自动调整学习率 (Adaptive Learning Rate)	53
4.2.4 损失函数也可能有影响	57
4.2.5 批次标准化(Batch Normalization)	59
4.3 训练后——如何让训练精益求精？	62
4.3.1 正则化 (Regularization)	62
4.3.2 超参数调优	63
4.3.3 模型集成和迁移学习	64
5 循环神经网络 (Recurrent Neural Network)	66
5.1 背景	66
5.2 循环神经网络 (RNN)	67
5.2.1 简单 RNN 的工作流程	67
5.2.2 简单 RNN 的训练	68
5.2.3 简单 RNN 训练的困难	69
5.3 长短期记忆神经网络 (LSTM)	70
5.3.1 门控记忆元构成的 RNN——LSTM 介绍	70
5.3.2 LSTM 的计算与架构	71
5.3.3 LSTM 为什么比简单 RNN 更好	72
5.4 循环神经网络的应用	73
5.4.1 多对一序列 (Many to one)	73
5.4.2 多对多序列 (Many to Many)	74
5.4.3 “超越序列” (Beyond Sequence) 的应用	75
5.4.4 自编码器上的应用	75

6 自注意力机制 (Self-attention)	77
6.1 输入是序列的情况 (Sequence as Input)	77
6.1.1 输入是向量组 (Vector Set) 的例子	77
6.1.2 输出可能长什么样?	78
6.2 自注意力机制的提出与介绍	79
6.2.1 背景——序列标注的解决办法	79
6.2.2 自注意力机制的运作	81
6.2.3 多头注意力机制 (Multi-head Self-attention)	83
6.3 自注意力机制的应用	84
6.3.1 在语音上的应用	84
6.3.2 自注意力机制 v.s 卷积神经网络	85
6.3.3 自注意力机制 v.s 循环神经网络	87
6.4 序列到序列模型 (Seq2Seq) 及应用	88
6.4.1 语音识别、机器翻译与语音翻译和合成	88
6.4.2 聊天机器人 (Chat Bot) 与问答任务 (QA)	89
6.4.3 Seq2Seq 的一些新奇应用	91
6.5 自注意力机制经典模型——Transformer	92
6.5.1 Transformer 编码器 (Encoder)	92
6.5.2 Transformer 解码器 (Decoder)	94
6.5.3 编码器-解码器注意力	98
6.5.4 Transformer 的训练	99
7 图神经网络	103
7.1 图神经网络基本介绍	103
7.1.1 为什么需要图结构?	103
7.1.2 图结构的选择	104
7.1.3 从图结构到图神经网络	104
7.2 图神经网络的经典模型	105
7.2.1 图卷积神经网络 (GCN)	106
7.2.2 从 GCN 到 GraphSAGE	107
7.2.3 图注意力网络 (GAT)	109
7.3 图机器学习的应用	110
7.3.1 节点级别任务应用	110
7.3.2 边级别任务应用	111
7.3.3 图级别任务	111
8 生成式模型 (Generative Model)	113
8.1 背景	113
8.1.1 将网络当成生成器来用	113
8.1.2 为什么需要分布?	113
8.1.3 生成模型的分类	114
8.2 生成对抗网络 (GAN) 介绍	114

8.2.1 生成对抗网络的组成	115
8.2.2 生成对抗网络的基本算法	116
8.2.3 生成对抗网络的训练目标	117
8.3.1 GAN 训练困难的本质原因	118
8.3.2 GAN 目标函数带来的困难及改进技巧	119
8.3.3 如何评估生成器的好坏	122
8.4 GAN 的常见变种	124
8.4.1 条件型生成 (Conditional GAN)	124
8.4.2 Cycle GAN	126
9 自监督学习 (Self-supervised Learning)	129
9.1 BERT 的模型结构	129
9.1.1 BERT 模型究竟在做什么事	129
9.1.2 BERT 预训练 (Pretrain) 与微调 (fine-tune)	131
9.1.3 使用 BERT 的 4 种情况	131
9.1.4 BERT 有用的原因	134
9.2 BERT 的奇闻异事	136
9.2.1 意想不到的 BERT 应用	136
9.2.2 BERT 在多语言上的应用	137
9.3 生成式预训练 (GPT)	139
9.3.1 GPT 的框架概念	139
9.3.2 GPT 的使用	139
9.4 自监督学习的其他应用	141
10 自编码器 (Autoencoder)	142
10.1 自编码器的概念	142
10.1.1 从另一个角度认识自监督学习	142
10.1.2 为什么需要自编码器？	143
10.1.3 自编码器不是一个新的概念	143
10.2 自编码器的结构	144
10.2.1 栈式自编码器	144
10.2.2 稀疏自编码器	145
10.2.3 去噪自编码器	146
10.3 自编码器的应用	146
10.3.1 特征解耦	146
10.3.2 离散潜在表征	148
10.3.3 自编码器的其它应用	149
11 其它生成式模型	152
11.1 变分自编码器 (VAE)	152
11.1.1 变分自编码器的引入	152
11.1.2 变分自编码器的模型架构	153
11.1.3 变分自编码器的训练原理	155

11.1.4 VAE v.s. GAN	157
11.2 基于流（Flow-based）的生成式模型	158
11.2.1 流模型的神奇之处	158
11.2.2 数学背景	158
11.2.3 流模型的训练	160
11.3 生成式模型综合应用——VITS 模型	162
11.3.1 变分推断	163
11.3.2 对齐估计	163
11.3.3 对抗训练	165
11.3.4 VITS 的应用	165
12 机器学习的可解释性	166
12.1 可解释性的基本概念	166
12.1.1 什么是可解释性	166
12.1.2 为什么需要可解释性	166
12.1.3 可解释性 v.s 能力	167
12.2 局部可解释性（Local Explanation）	168
12.2.1 基本思想	168
12.2.2 移除组成要素（遮挡法）	168
12.2.3 改变梯度	169
12.3 全局可解释性（Global Explanation）	170
12.3.1 反向寻找理想输入	171
12.3.2 添加正则化项	171
12.4 扩展与小结	172
13 机器学习中的攻击与防御	174
13.1 模型攻击	174
13.1.1 模型攻击简介	174
13.1.2 如何进行网络攻击	176
13.1.3 最简单的攻击方式	177
13.2 其它类型的攻击	178
13.2.1 黑盒攻击	178
13.2.2 单像素攻击	179
13.2.3 通用对抗攻击	180
13.2.4 现实世界中的攻击	180
13.3 机器学习中的防御	182
13.3.1 机器学习中的被动防御	182
13.3.2 主动防御	184
14 迁移学习（Transfer Learning）	185
14.1 模型微调（Model Fine-tune）	186
14.1.1 技巧 1——保守训练（Conservative Training）	186
14.1.2 技巧 2——层迁移（Layer Transfer）	187

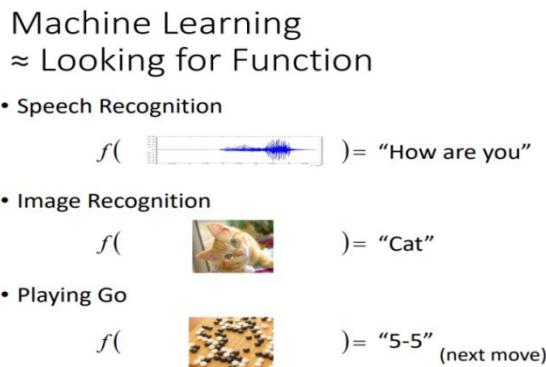
14.2 多任务学习 (Multi-task Learning)	188
14.2.1 多任务学习的概念	188
14.2.2 多任务学习的基本框架	188
14.2.3 多任务学习的应用	189
14.3 域对抗训练	190
14.3.1 域对抗训练的提出背景	190
14.3.2 域对抗训练使用的原因	190
14.3.3 域对抗训练的组成	191
14.4 零样本学习 (Zero-shot Learning)	192
14.4.1 用属性表示类别	193
14.4.2 属性嵌入	193
14.4.3 零样本学习的应用	193
15 强化学习 (Reinforcement Learning)	195
15.1 强化学习基本概念	195
15.1.1 强化学习的本质	195
15.1.2 强化学习的三个步骤	196
15.1.3 强化学习 v.s GAN	197
15.2 评价行为的标准	197
15.2.1 版本 0	197
15.2.2 版本 1	198
15.2.3 版本 2	199
15.2.4 版本 3	199
15.3 策略梯度算法	200
15.3.1 策略梯度的流程	200
15.3.2 同策略 (On-policy) v.s. 异策略 (Off-policy)	201
15.4 行动者-批评者算法	202
15.4.1 什么是批评者 (Critic)	202
15.4.2 批评者是怎么被训练出来的?	202
15.4.3 版本 3.5 和版本 4	204
15.4.4 Actor-Critic 的训练技巧	205
15.5 本章小结	205
16 终生学习	206
16.1 终生学习基本介绍	206
16.1.1 什么是终生学习?	206
16.1.2 终生学习的难点	207
16.2 终生学习的评估标准	209
16.2.1 模型准确率	209
16.2.2 记忆能力	209
16.2.3 迁移能力	210
16.3 终生学习的研究方向	210

16.3.1 选择性的突触可塑性	211
16.3.2 权重设置的解决	211
16.3.3 其它方法	213
17 元学习（Meta Learning）	214
17.1 元学习的概念	214
17.1.1 元（Meta）的含义	214
17.1.2 元学习的步骤	214
17.1.3 元学习与机器学习的比较	216
17.2 元学习的实例算法	218
17.2.1 从迁移学习（Transfer Learning）到 MAML	218
17.2.2 MAML 与预训练的区别	219
17.2.3 元学习算法的评价标准	220
17.2.4 MAML 算法和 Reptile 算法	221
18 网络压缩	223
18.1 网络剪枝	223
18.1.1 网络剪枝的框架	223
18.1.2 网络剪枝的两种形式	224
18.1.3 为什么需要剪枝	225
18.2 其他技术	226
18.2.1 知识蒸馏	226
18.2.2 参数量化	227
18.2.3 架构设计	228
18.2.4 动态计算	229
总结	232

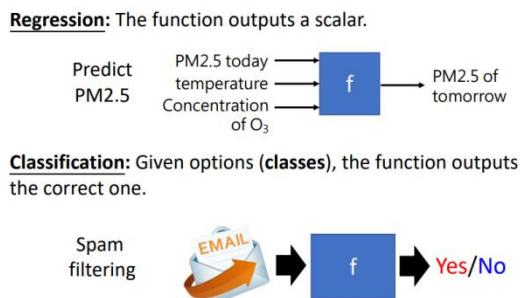
1 深度学习介绍

1.1 基本概念简介

什么是机器学习（Machine Learning）？一个通俗易懂且简单的解释是，机器学习约等于寻找一个函数。



我们可以首先把机器学习大致分为两类，一类是回归问题(Regression)，另一类是分类问题(Classification)，对于回归问题，它需要寻找的函数的输出是一个标量，而分类问题是说，人类事先选择一些选项 (classes) ，函数需要从这些选项中选出最合适的选项，作为我们的分类。



当然，机器学习肯定不止回归和分类这两个任务。事实上，在机器学习领域中，仍然存在一个“黑暗大陆”，叫做结构式学习 (structured learning)，即机器能够产生一些有结构的东西。

那么机器怎么去找一个函数呢？主要是分为以下三个步骤。



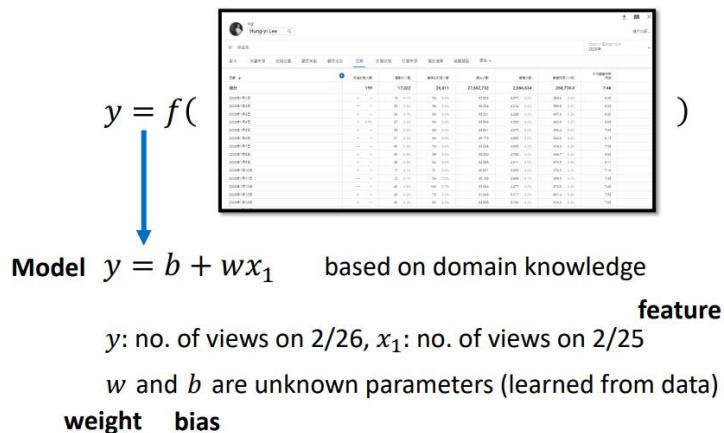
上述三个步骤合称为机器学习的训练过程(training), 在这里, 我们可以初步认为: 对于一个机器学习问题, 都包含模型, 损失函数(loss function)和优化算法这三个内容, 它们被称为机器学习的三要素。

1.1.1 模型

模型的定义, 从本质上来说就是定义一个含有未知参数的函数, 这个函数通常是基于我们的领域知识(domain knowledge)进行适当地猜测得出的。比如说, 对于一个线性回归问题, 我们会很自然地猜测预测量与已知的数据输入特征之间呈线性关系, 即:

$$y = \mathbf{w} \cdot \mathbf{x} + b$$

在上面的公式中, w 和 b 分别是模型的权重(weight)和偏差(bias), 它们合起来就是模型的未知参数(unknown parameters), 当然在实际工作中, 你的模型不太可能会这么简单, 你将会接触到一些更复杂的模型。



1.1.2 损失函数

我们在上一小节刚刚定义了模型, 但是我们并不知道模型的好坏。事实上, 损失函数可以帮助我们进行评判。损失函数是一个以模型中的未知参数为输入的函数, 它的输出将直接/间接反映出模型的好坏(goodness of the model)。

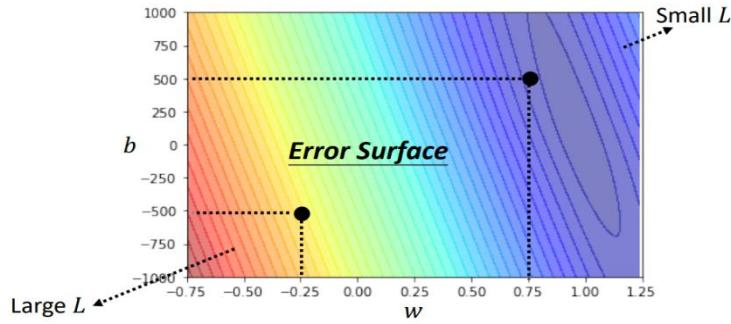
在机器学习中, 我们会根据数据的特点来选择合适的损失函数, 常见的选择有平均绝对值误差 (Mean Absolute Error, MAE), 均方误差 (Mean Square Error, MSE), 交叉熵 (cross-entropy) 等, 不同的损失函数的定义会得到不同的结果。

$$e = |y - \hat{y}| \quad L \text{ is mean absolute error (MAE)}$$

$$e = (y - \hat{y})^2 \quad L \text{ is mean square error (MSE)}$$

If y and \hat{y} are both probability distributions \rightarrow Cross-entropy

我们发现，当未知参数不同时，即便我们选用的模型是相同的，损失函数的效果也会不同，通常这一系列的损失函数所呈现出来的是一组等高线，我们将其称为误差曲面(Error Surface)，如下图所示：



由于损失函数代表着训练的误差，我们肯定是希望损失函数的值越小越好，换句话说，我们需要找到这样的一组未知参数，使得以这组未知参数为输入的损失函数达到最小值，这本质上是一个优化的问题，因此我们需要一些优化算法来进行求解。

1.1.3 优化

最后一步就是求解一个优化的问题，即找到一组(w, b)，使得损失函数的值最小，这个问题可以用数学语言表达如下：

$$(\mathbf{w}^*, \mathbf{b}^*) = \arg \min_{\mathbf{w}, \mathbf{b}} L(\mathbf{b}, \mathbf{w})$$

机器学习/深度学习中，大多数流行的优化算法通常是基于一种叫梯度下降(gradient descent)的基本方法，梯度下降算法的流程是，随机选择一组初始的未知参数，计算损失函数在当前位置的梯度并进入迭代：

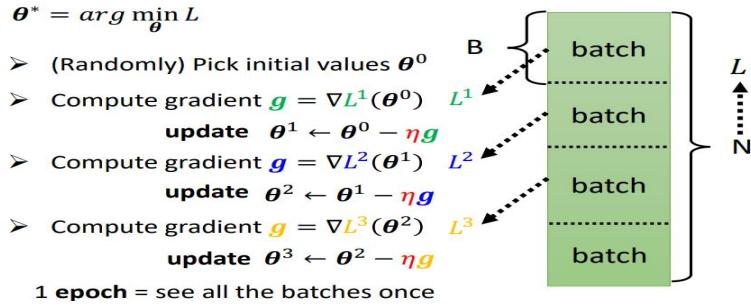
- (Randomly) Pick initial values w^0, b^0
- Compute

$\frac{\partial L}{\partial w} |_{w=w^0, b=b^0}$
 $\frac{\partial L}{\partial b} |_{w=w^0, b=b^0}$

$w^1 \leftarrow w^0 - \eta \frac{\partial L}{\partial w} |_{w=w^0, b=b^0}$
 $b^1 \leftarrow b^0 - \eta \frac{\partial L}{\partial b} |_{w=w^0, b=b^0}$
- Can be done in one line in most deep learning frameworks
- Update w and b interatively

在梯度下降算法中，我们需要设置一个学习率(learning rate)，每次迭代中，未知参数的梯度下降的步长取决于学习率的设置，这种由人为设定的参数被称为超参(hyperparameters)。

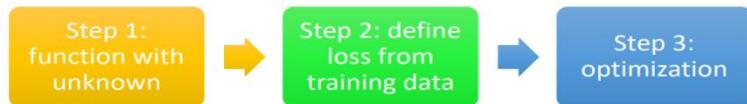
实际上，在我们做梯度下降 t 时，会遇到一个较大规模的资料，那么我们的梯度的计算的代价可能有些高，因此，一般可以将这 N 个大规模资料分成一个一个的 batch，每一个 batch 中有 B 个资料(随机分组)，这样的方法叫做 mini-batch 梯度下降(MBGD, Mini-Batch Gradient Descent)。



1.2 神经网络与深度学习

在本书中，无论是监督学习还是无监督学习，亦或是强化学习，我们都需要用到神经网络(neural network)的相关技术，人们将这种利用神经网络进行的机器学习称为深度学习(deep learning)。

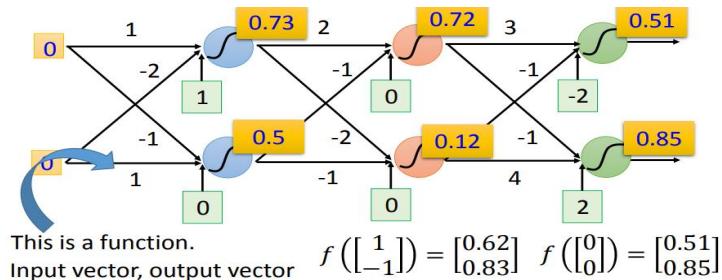
深度学习的步骤与机器学习一样，都可以分为以下三步：



当然，在深度学习中，第一步中定义的函数就叫做神经网络，它是由一系列逻辑回归(Logistic Regression)连接组成的，其中每个逻辑回归被称为神经元(neuron)，事实上，对于神经元的连接方式，同样是可以由人们自主决定的。

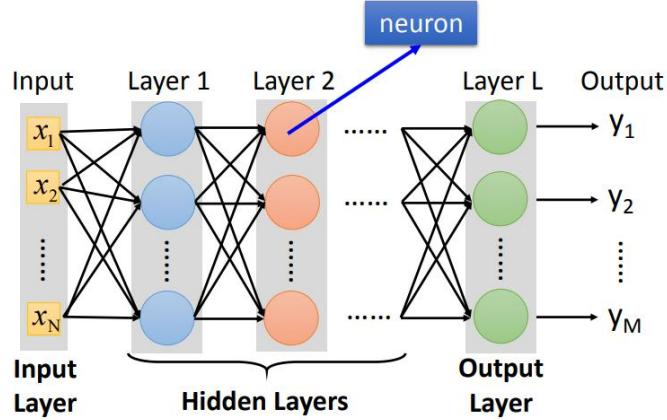
1.2.1 全连接神经网络

最常见的连接方式叫做全连接前馈神经网络(fully-connected feed-forward network)，对于每一组神经元，都有一组权重和偏差，在这个网络中，最左边是我们的输入数据，假设网络中的所有权重和偏差都是已知的，我们可以逐层计算出每一层的值，最后一层就是神经网络的输出。因此，一个神经网络就是一个向量到向量(vector to vector)的函数。



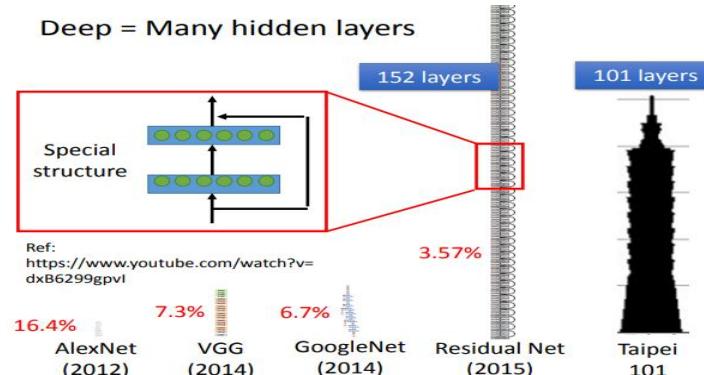
Given network structure, define **a function set**

更一般地，我们可以将这个神经网路表示成如下形式，其中第一层我们称为输入层（input layer），最后一层称为输出层（output layer），中间的层数被称为隐藏层(hidden layer)。这个网络之所以叫做 fully-connected 是因为在相邻的两层中，所有的神经元都是相连的，而之所以叫做 feed-forward，是因为这个网络是按照从第 1 层到第 L 层的顺序进行传播的。



1. 2. 2 深度神经网络与特征工程

那么为什么叫做深度学习呢？事实上，deep learning 中的 deep 意味着有很多层隐藏层，常见的深度神经网络(DNN, deep neural network)的应用如下：



但在我们的实际应用中，并非需要一味地将我们的网络叠得尽可能深，因为在有些情况下，当我们的网络叠得更深时，在 Youtube 点播预测的案例中，我们发现，当在 3 层之后再叠一层，在已知的资料上，4 层比 3 层好，但是在未知资料上，4 层反而更差了。这种现象被称为过拟合（overfitting）。

	1 layer	2 layer	3 layer	4 layer
2017 – 2020	0.28k	0.18k	0.14k	0.10k
2021	0.43k	0.39k	0.38k	0.44k

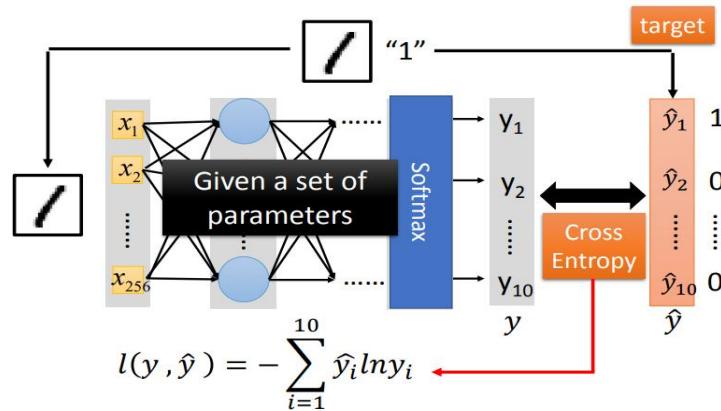
Better on training data, worse on unseen data

→ Overfitting

在实际应用中，我们的神经网络的结构是由自己决定的。因为我们仅对输入层和输出层的维度有着明确的要求，我们通常把隐藏层的设计称为特征工程（feature engineering）。

1.2.3 反向传播

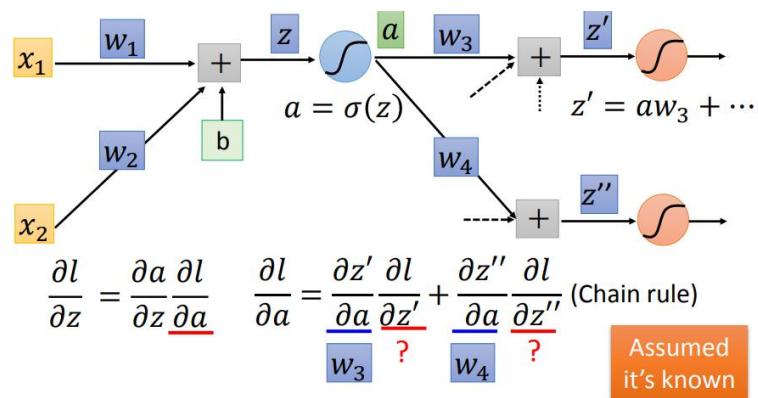
在定义完函数后，和机器学习一样，需要定义一个损失函数。假设给定一组参数，以多目标分类为例，我们通常会用输出值和目标值去做一个交叉熵(cross-entropy)，这是因为在这个分类问题中，我们通常是根据概率的大小来决定某个输入到底属于哪一类的，具体细节我们会在后面章节中进行介绍。



对于这个优化问题，它的求解方法和之前没有本质上的区别。但是对于 DNN 这种数据规模较大的东西，有一种更高效的梯度计算的方法被称为反向传播（backpropagation）。

字面上理解，反向传播就是从输出层向输入层传播的过程，对于一个神经网络，需要对每一层的 w 和 b 进行偏导数的计算。

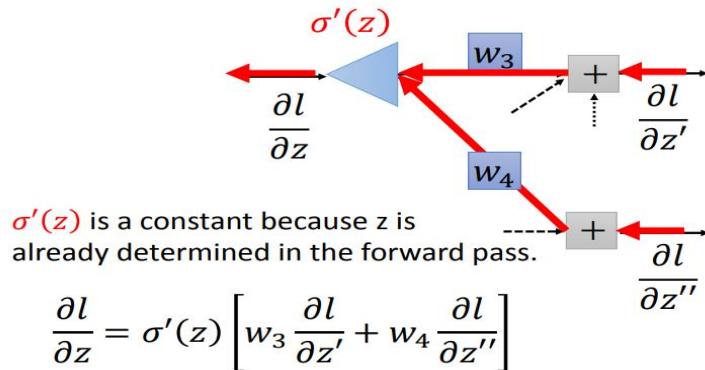
通常这样的计算是不太方便的，因为隐藏层的 w 和 b 是不与损失函数直接相关的，这个时候，我们需要利用链式法则，我们可以根据某一层隐藏层的参数计算出下一层的激活函数，再用其对 w/b 求偏导，这个步骤被叫前向传播(forward pass)，根据链式法则，我们只需要计算损失函数对之前得出的激活函数 activate function 的偏导，再将其与之前的结果相乘，就得到了我们需要计算的梯度，而这一步骤由于是与神经网络的方向相反的，我们称之为 backward pass。



实际上，每一层都可以生成一系列新的 w 和一系列新的 z，因此在每一个隐藏层中，都会存在前向传播的计算。而对于反向传播，直接求偏导可能比较困难，因此我们需要借助链式法则：

$$\frac{\partial C}{\partial z} = \frac{\partial a}{\partial z} \cdot \frac{\partial C}{\partial a}, \quad \frac{\partial C}{\partial a} = \frac{\partial z'}{\partial a} \cdot \frac{\partial C}{\partial z'} + \frac{\partial z''}{\partial a} \cdot \frac{\partial C}{\partial z''}$$

这个 backward 的计算看起来有些难以理解，我们可以换一个角度去看待，我们最后一步的偏导数的值看作一个假想的神经元作为输入，如下图所示，这两种表达式的结果是等价的。



综上所述，我们在进行神经网络的梯度计算时，只需要建立一层一层的假想的反向神经元，因此我们把这个过程叫做反向传播。

在实操过程中，我们可以根据上述的原理（本质就是链式法则）来写出代码计算梯度，但是这样其实还是有一点繁琐，在现阶段流行的深度学习框架中，官方已经写好了反向传播的代码，在实际项目中直接使用即可。

1.3 深度学习总体攻略

在本节中，我们将介绍一些深度学习任务中会出现的一些常见问题。在接手一个机器学习/深度学习的任务时，你将遇到一些数据，它们被划分成训练数据（training data）和测试数据（testing data），其数据格式分别如下所示：

Training Data: $\{(x^1, \hat{y}^1), \dots, (x^N, \hat{y}^N)\}$, **Testing Data:** $\{x^{N+1}, \dots, x^{N+M}\}$

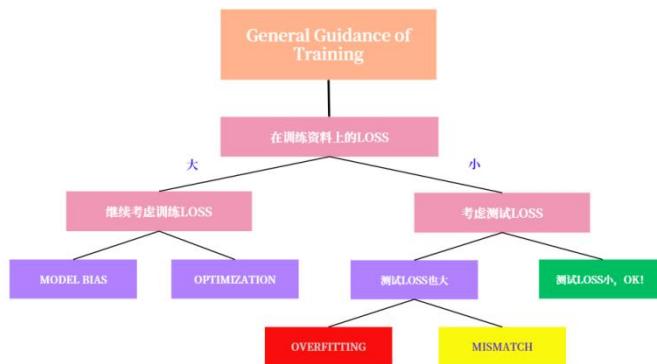
事实上，当你的任务不同时，训练数据中的 x 和 y 也会不同，常见的一些深度学习任务如下图所示：

<u>Speech Recognition</u>	<u>Image Recognition</u>
$x:$	$\hat{y}:$ phoneme
$x:$	$\hat{y}:$ soup
<u>Speaker Recognition</u>	<u>Machine Translation</u>
$x:$	$x:$ 痛みを知れ
$\hat{y}:$ John (speaker)	$\hat{y}:$ 了解痛苦吧

当你接触到一个全新的深度学习问题时，你该如何将它做得更好？首先你需要检查训练资料上的损失函数，这一点是很多人忽略掉的，这些人总是习惯性地只检查测试集上的损失函数，并将损失函数过大全部归因于过拟合，但事实上并非如此！

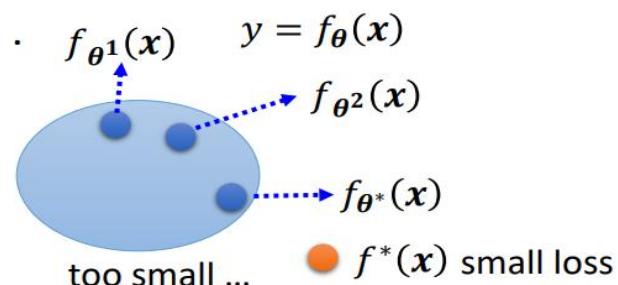
需要注意的是，过拟合是损失函数在训练数据上表现得好，在测试数据上表现得不好的一种现象，因此当我们的损失函数在训练集上表现得同样糟糕时，那就不是过拟合的问题了。

综上所述，整个流程如下图所示，在本节中，我们将分别对深度学习中所可能出现的三个问题及其解决方案进行初步讨论：



1.3.1 模型偏差

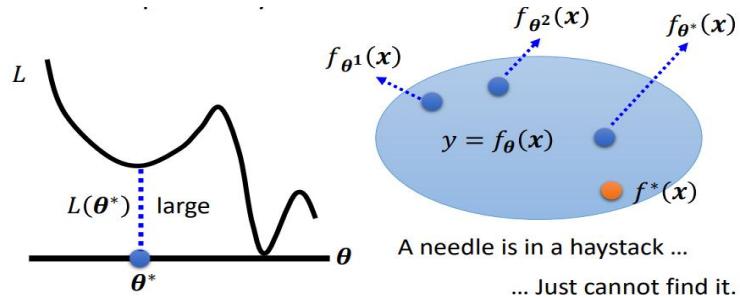
从流程图中可知，当你的模型在训练资料上的损失函数很大时，这可能是模型偏差的原因，换句话说，你在这个问题中选择的模型太过简单，以至于无论你用这个给模型选择什么样的参数 θ ， $f(\theta)$ 都不会变得很小：



对于这个问题，你的解决方案是选择一个更有弹性的模型，比如你可以增加模型的特征，或者是用深度学习模型（使用更多的神经元）等。

1.3.2 优化问题

当然，在训练资料上出现很大的 loss 值并不总是因为模型偏差的原因，另一种可能是优化没有做好，这可能是因为在做梯度下降时， $L(\theta)$ 卡在了局部最小（local minima）的点，但实际上我们需要让这个优化问题停在全局最小（global minima）。

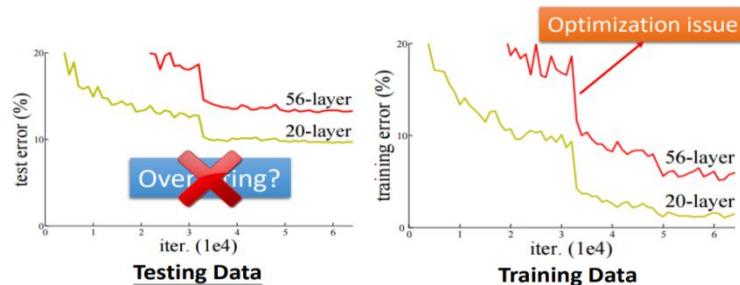


对于优化问题的解决方法，就是换一些更强大的优化器去尝试求解，深度学习中，常见的优化算法有 Adam, RMSProp, AdaGrad 等，这些将在后续章节中有所提及。

1.3.3 过拟合

最后一个问题是过拟合，过拟合问题出现在，给定一批训练资料和测试资料，你的 loss 函数在训练资料上很小，但是在测试资料上很大。一般地，更有弹性的模型更容易发生过拟合的现象，但我们一定需要注意，在判断是否是过拟合之前，首先需要检查训练资料上的 loss 函数。

下面是何恺明 ResNet 论文(Ref: <http://arxiv.org/abs/1512.03385>) 中的一张图，在这张图中，我们可以看出，使用 56 层的网络的误差更大并非是因为过拟合，因为它在训练资料上同样误差也很大，这就是优化算法出现的问题。



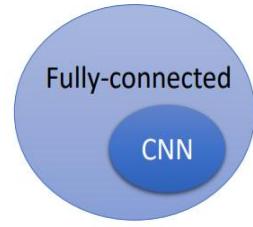
对于过拟合问题，主要有三种解决方案，第一种也是最有效的一种解决方案是增加训练资料，这是很容易理解的，因为当你选择的数据越多，它更容易拟合出一条更接近的曲线。

第二种方法就是数据增广 (Data Augmentation)，这个技术在图像处理中经常被使用，比如我们可以将图像进行左右翻转，当然，增广的数据也一定要遵循一定的规则的，比如在下图中，我们可以对猫进行左右翻转，放缩，但是不能上下颠倒，因为颠倒之后，机器有很大概率将不会识别出来它是一只猫。



最后一种方法就是增加对模型的限制，这些限制完全可以凭借自己对模型的理解和意愿，常见的处理手法如下：

- Less parameters, sharing parameters
- Less features
- Early stopping
- Regularization
- Dropout



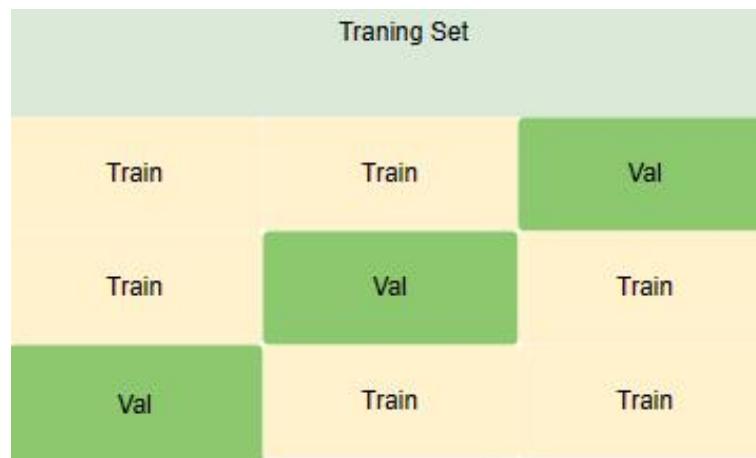
1.3.4 验证集与交叉验证

当你利用攻略以及上述方法对模型进行完调整后，假如你对你目前的工作还是不够满意，或许以下方法可以帮你更进一步。

在之前的工作中，我们仅仅接触到了训练集和测试集，但在很多实际工作中，我们会将原先的训练集进行进一步划分：



为了使模型尽可能接近模型在测试集上的表现，我们采用 K 交叉验证 (K-fold Validation) 的方法，具体做法是将原始训练集分为 K 份，依次选取其中的一份作为验证集，其余的 K-1 份作为新的训练集，这样会得到 K 个模型。这 K 个模型分别在验证集中评估结果，最后的误差加和平均就得到交叉验证误差。交叉验证有效利用了有限的数据，并且评估结果能够尽可能接近模型在测试集上的表现，可以做为模型优化的指标使用。具体做法如下：



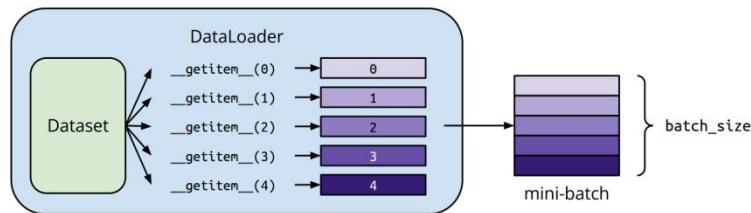
当然，K 交叉验证并不适合数据规模太大的情况，因为它会在原先训练的基础上，增加 K 倍的训练成本（时间），但如果数据规模较小时，其他方法无法继续提升性能，这也是一个可观的优化策略。

1.3.5 深度学习总体流程

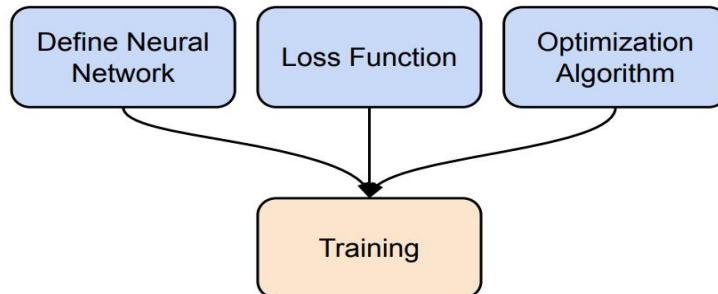
在本章的最后，我们将对深度学习的总体流程进行介绍。在前面的章节中，我们已经了解了深度学习训练的步骤以及可能出现的常见的问题。我们接下来给出一个完整的深度学习步骤。

首先，对于一个完整的深度学习任务，我们需要收集数据，数据可以是一些知名的公开数据集，比如 MNIST, CIFAR10 等，也可以是自己的私域数据集，不管是公开数据集还是私域数据集，我们都需要将它导入，并且运用 Python 的一些库进行数据预处理，其中，最常见的数据预处理方法就是将它分成很多批，然后随机打乱。

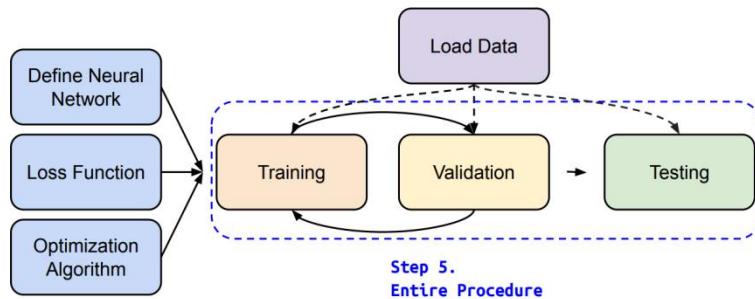
```
dataset = MyDataset(file)  
  
dataloader = DataLoader(dataset, batch_size=5, shuffle=False)
```



接下来，我们需要进行定义神经网络，选择损失函数以及用梯度下降求解最小损失函数的优化问题，这其实就是我们之前讲过的深度学习训练的三部曲，在此我们不再赘述，流程图如下所示：



当我们完成训练工作之后，我们需要对模型进行测试，测试本质上就是要将模型使用在它之前从未看到过的数据上并对其性能（准确率等）进行测试。当然，我们在前文也提到了，为了对模型的能力进行初步评估并降低出现过拟合等风险，我们会讲训练拆分成训练+验证两部分。因此，整体的深度学习训练流程（pipeline）如下：



2 线性神经网络

在介绍深度神经网络之前，我们来看一看最简单的模型——线性模型（Linear Model）。在机器学习中，线性回归（Linear Regression）和逻辑回归（Logistics Regression）都可以看作是线性模型。我们将在本章中结合案例分别进行介绍。

2.1 线性回归

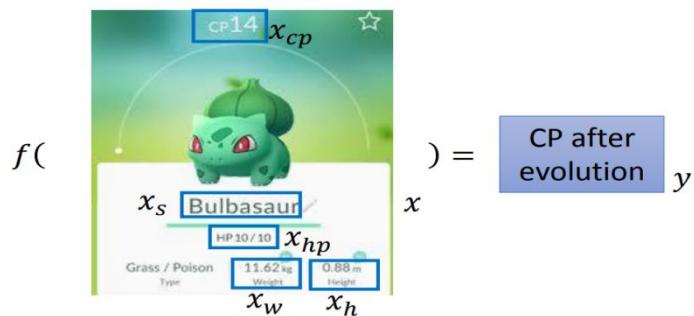
我们在第一章曾说过，机器学习的本质就是寻找一个函数，函数的两个要素是自变量与因变量，而回归（regression）是能为一个或多个自变量与因变量之间的关系建模的一类方法。事实上，回归经常用来表示输入和输出之间的关系。

在机器学习领域中，回归问题大多数都与预测（prediction）相关，因此回归问题通常输出的是一个变量，常见的应用场景如下所示：



接下来，我们通过一个具体案例——宝可梦 CP 值预测来了解一下回归问题。

- Estimating the Combat Power (CP) of a pokemon after evolution

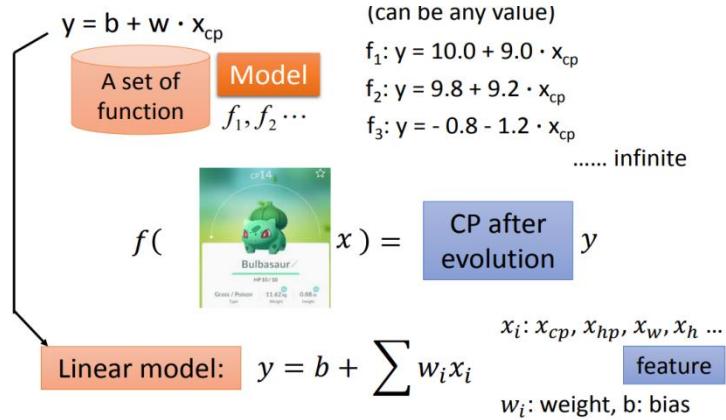


2.1.1 最简单的回归——线性回归

对于这个问题，我们用机器学习的三个步骤去解决。第一步是寻找一个合适的模型，我们假设宝可梦进化前后的 CP 值成线性关系，即假设这个模型为：

$$y = \mathbf{w} \cdot \mathbf{x} + b$$

其中， w 和 b 可以是任意选取的，而我们事先并不知道最优的 w , b 具体是多少，因此我们会得到一个函数集合（function set）。

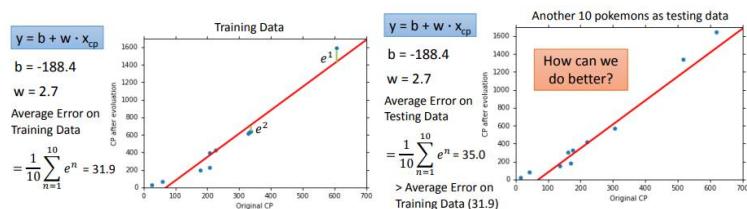


接下来，我们需要评判一个函数的“优劣程度”，即利用上一章讲的损失函数，在这里我们采用的是均方误差损失函数，公式如下：

$$L(\mathbf{w}, \mathbf{b}) = \sum_{i=1}^n (\hat{y}^i - (\mathbf{b} + \mathbf{w} \cdot \mathbf{x}_{CP}))^2$$

最后一步就是选择一个损失最小的函数，得到最终线性回归中最优的 w 和 b ，求解做法同样也是梯度下降法。

上述过程我们在第一章其实有所提及，接下来我们看看结果，我们可以通过梯度下降求得 $(\mathbf{w}^*, \mathbf{b}^*) = (2.7, -188.4)$ ，运用到训练资料和测试资料得到结果如下：



2.1.2 从线性回归到多项式回归

对于我们的线性回归，我们发现，在训练资料上的损失函数值比测试资料上要高，这其实是不好的，为了让模型做得更好，我们考虑重新设计一个模型。

我们首先考虑将模型加入一个二次项，模型表达式如下：

$$y = \mathbf{b} + \mathbf{w}_1 \cdot \mathbf{x}_{CP} + \mathbf{w}_2 \cdot \mathbf{x}_{CP}^2$$

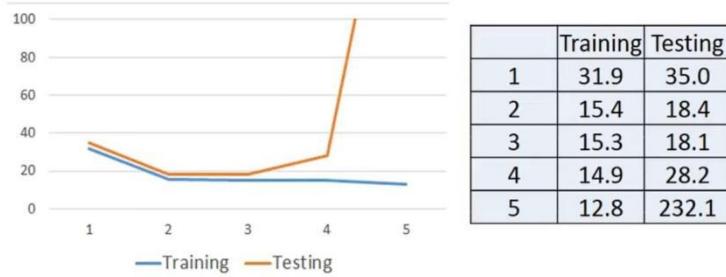
得到的训练 loss 是 15.4，测试 loss 是 18.4，由此我们可以猜测可以加入三次项，四次项等等，来使模型变得更好。

事实上，对于加入了二次项及以上的模型，我们将它称为多项式回归。需要注意的是，多项式回归模型也是线性的，因为我们只有一个单变量 x_{CP} ，考虑一个 m 次多项式的回归，我们可以将模型改写成如下形式：

$$y = \mathbf{w} \cdot \mathbf{x} + \mathbf{b}$$

其中， $\mathbf{w} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_m]^T$, $\mathbf{x} = [\mathbf{x}_{CP}, \mathbf{x}_{CP}^2, \dots, \mathbf{x}_{CP}^m]$ ，因此，其本质也是一个线性模型。对于宝可梦 CP 值预测问题，李宏毅老师做出了一个实验，即使用了二次-五次多项式分别进

行预测，结果如下图所示：

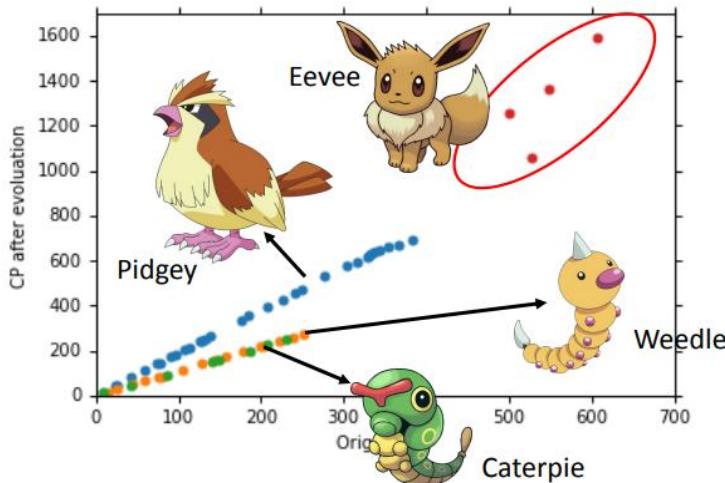


实验表明，当模型变得越来越复杂时，模型在训练数据上会表现得越来越好。但是我们发现，一个更复杂的模型并不能保证在测试集上获得越来越好的结果。这种现象一般被称为过拟合（over-fitting），因此我们需要一个更好的模型。

2.1.3 解决过拟合的办法

在上一小节中，当我们收集 10 只宝可梦并用多项式回归进行预测时，出现了过拟合的现象。我们在这一小节试图对过拟合进行解决。

第一种解决的办法是收集更多的数据，我们现在收集 60 只宝可梦，把原始的 CP 值和进化之后的 CP 值作图，我们发现，存在着另外一种关系，它不能用简单的多项式关系来表达，如下图所示：



实际上，这个隐藏因素是宝可梦的物种，在上图中，不同物种的宝可梦用了不同颜色的点进行表示，因此，我们可以根据物种重新设计模型，假设 x_s 是宝可梦的物种，设计如下：

If $x_s = \text{Pidgey}$:	$y = b_1 + w_1 \cdot x_{cp}$
If $x_s = \text{Weedle}$:	$y = b_2 + w_2 \cdot x_{cp}$
If $x_s = \text{Caterpie}$:	$y = b_3 + w_3 \cdot x_{cp}$
If $x_s = \text{Eevee}$:	$y = b_4 + w_4 \cdot x_{cp}$

但是我们可以对上述模型进行重写，可以将判断语句用一个 0-1 函数 $\delta(x)$ 进行替换。

$$\delta(x_s = \text{Pidgey}) = \begin{cases} 1 & \text{if } x_s = \text{Pidgey} \\ 0 & \text{otherwise} \end{cases}$$

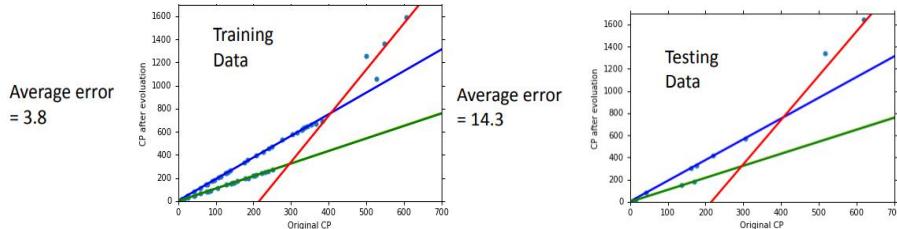
$$\text{If } x_s = \text{Pidgey}$$

$$y = b_1 + w_1 \cdot x_{cp}$$

模型如下所示：

$$y = b_1 \cdot \delta(x_s = \text{Pidgey}) + w_1 \cdot \delta(x_s = \text{Pidgey})x_{cp} \\ + b_2 \cdot \delta(x_s = \text{Weedle}) + w_2 \cdot \delta(x_s = \text{Weedle})x_{cp} \\ + b_3 \cdot \delta(x_s = \text{Caterpie}) + w_3 \cdot \delta(x_s = \text{Weedle})x_{cp} + \dots$$

我们发现，改进后的模型在训练资料上变得非常好，训练 loss 只有 3.8，而在测试资料上的 loss 也比之前要好，结果如下图所示：



但是这里还有不能拟合得很好的点，还有其他潜在因素影响预测结果，比如宝可梦的体重，身高，HP 值等。我们将所有的因素加入模型，结果发现，训练资料上确实误差变小了，但测试资料上又出现了过拟合的现象：

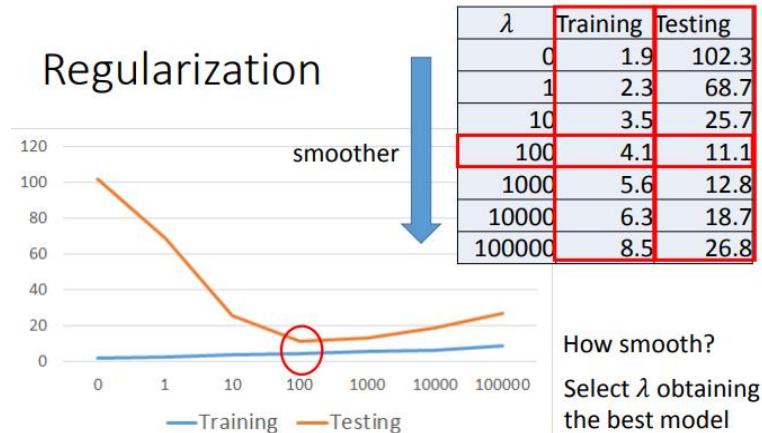
If $x_s = \text{Pidgey}$:	$y' = b_1 + w_1 \cdot x_{cp} + w_5 \cdot (x_{cp})^2$	Training Error = 1.9 Testing Error = 102.3 Overfitting!
If $x_s = \text{Weedle}$:	$y' = b_2 + w_2 \cdot x_{cp} + w_6 \cdot (x_{cp})^2$	
If $x_s = \text{Caterpie}$:	$y' = b_3 + w_3 \cdot x_{cp} + w_7 \cdot (x_{cp})^2$	
If $x_s = \text{Eevee}$:	$y' = b_4 + w_4 \cdot x_{cp} + w_8 \cdot (x_{cp})^2$	
$y = y' + w_9 \cdot x_{hp} + w_{10} \cdot (x_{hp})^2$ $+ w_{11} \cdot x_h + w_{12} \cdot (x_h)^2 + w_{13} \cdot x_w + w_{14} \cdot (x_w)^2$		

实际上，这边还有一个解决办法，叫做正则化（Regulation），它其实是在损失函数中，加入一些辅助的项，从而让我们找到更好的函数，我们可以将原来的损失函数改写成下面的形式：

$$L(\mathbf{w}, \mathbf{b}) = \sum_{i=1}^n (\hat{y}^i - (\mathbf{b} + \mathbf{w} \cdot \mathbf{x}_{CP}))^2 + \lambda \sum (\mathbf{w}_i)^2$$

我们期待 w_i 越小越好，因为参数越小，函数越平滑（smooth）。平滑的意思是说，当输入有变化时，输出对于输入的变化是比较不敏感的。

事实上，人们更喜欢一个平滑的函数，这是因为平滑的函数能够减小在测试的时候噪声对输入的干扰，从而能得到一个更好的结果。我们选择了不同的正则化系数 λ 在宝可梦数据上进行了实验，实验结果如下：



假设我们找到的系数是比较小的，当 λ 越大时，说明考虑平滑项的影响力越大，找到的函数越平滑。我们发现，随着 λ 的增大，在训练集上的误差越来越大。而这件事其实是合理的，因为我们会越来越倾向于考虑平滑项而非损失函数本身。但是有趣的是，在测试集上，得到的误差可能会变小，但是 λ 太大时，测试集上的误差又会变大。因此我们可以看出，我们喜欢比较平滑的函数，但不喜欢太平滑的函数，换言之，正则化系数 λ 在深度学习中，也是一个常见的超参数，通常设置成 100 比较合适。

2.2 线性分类

在本章一开始，我们曾讲过，大部分的预测型问题都是回归问题，但并非所有预测都是回归问题。在本节中，我们将介绍分类问题，它的目标是预测数据属于一组类别中的哪一个类别。同样，我们根据一个宝可梦分类的问题对它进行介绍。



2.2.1 分类问题的输入

在这个问题中，我们大致的步骤是输入一只宝可梦，输出它对应的属性分类。

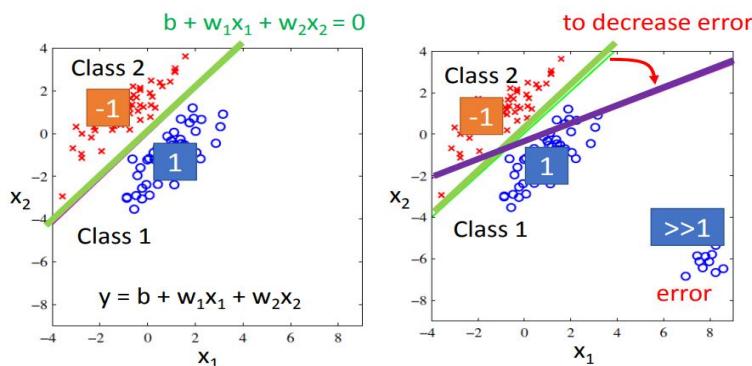
$$f(\text{ Pikachu }) = \text{ Lightning } \quad f(\text{ Squirtle }) = \text{ Water } \quad f(\text{ Venusaur }) = \text{ Grass }$$

那么我们该如何把宝可梦当作输入呢？一种常见的方法是将宝可梦数值化，我们知道一只宝可梦有很多属性，比如 HP, Attack, Defense, Speed 等。我们可以用一个向量将这些数值属性存起来。这样做好处是，每一只宝可梦都能几乎被独立表示出来，因为两只所有能力值完全一样的宝可梦应该是不存在的（可能存在，因为我没玩过游戏）。

- **Total**: sum of all stats that come after this, a general guide to how strong a pokemon is **320**
- **HP**: hit points, or health, defines how much damage a pokemon can withstand before fainting **35**
- **Attack**: the base modifier for normal attacks (eg. Scratch, Punch) **55**
- **Defense**: the base damage resistance against normal attacks **40**
- **SP Atk**: special attack, the base modifier for special attacks (e.g. fire blast, bubble beam) **50**
- **SP Def**: the base damage resistance against special attacks **50**
- **Speed**: determines which pokemon attacks first each round **90**

2.2.2 分类问题的步骤

接下来，我们来讨论一下如何做分类问题。考虑一个二分类问题，假如我们将这个分类问题当成回归问题来做，那么输出是一个标量，我们假设类别 1 输出为 1，类别 2 输出为 -1。我们可以计算回归问题的输出，以 0 为分界，如果我们的回归模型输出的值大于 0，就将它归为第一类，否则，归为第二类。

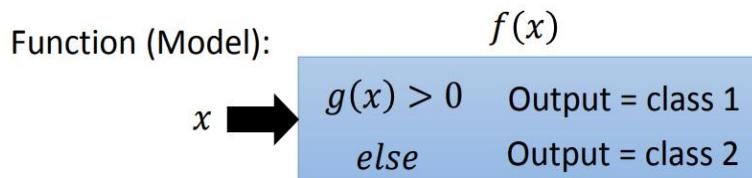


这件事情可能是可行的，如上图左所示，绿色的线是函数值等于 0 的部分，也就是第一类和第二类的分界线。但是这件事情也有一些问题，因为我们希望，不管你的样本数据的输出是在哪一类，它都应该与 1 或者 -1 越接近越好，对于一些远大于 1 的输出和远小于 -1 的输出，我们同样认为这是错误的数据。为了减少这样的数据，我们会将绿色的线往下移，变成紫色的线，因此回归问题与分类问题中的好的函数是不一样的。

实际上，这件事还有另外一个潜在的问题，假设我们现在在做一个三分类问题，按照二分类类似的方法，我们规定类别 1 输出 1，类别 2 输出 2，类别 3 输出 3。这样其实潜在定义了类别 1 和 2 比较相近，类别 1 和 3 相差较大，但事实上并没有这样的关系。

因此，我们不建议用回归问题的揭发来做分类问题。理想的做法如下：

首先我们需要找一个函数，同样以二分类为例，输入一个 x ，我们的函数 $f(x)$ 中可以内嵌一个函数 $g(x)$ 进行判断，假设 $g(x) > 0$ ，输出为第一类，否则输出为第二类。



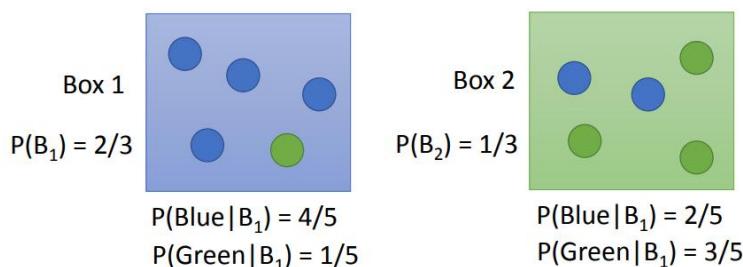
第二步是确定损失函数。对于分类问题，我们可以用分类样本正确的个数来定义损失函数，定义方法如下：

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

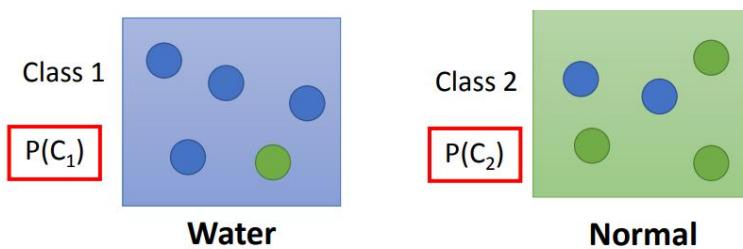
The number of times f
get incorrect results on
training data.

接下来，我们需要根据损失函数找到最好的函数，常见的办法有感知机（Perceptron）和支持向量机（SVM），我们今天会讲到另外一种方法，可以用几率的办法来看待。

回想我们学过的概率问题，假设有两个盒子，盒子 1 和 2，分别各自装有蓝色和绿色小球，现已知从盒子 1 中抽出球的概率，从盒子 2 中抽出球的概率以及从盒子 1 (2) 中抽出蓝球，绿球的概率。我们就可以通过贝叶斯公式计算抽出蓝（绿）球来自于盒子 1 (2) 中的概率。



基于上述例子，我们可以将盒子 1 和盒子 2 看作水系和普通系，假设我们将 140 只宝可梦数据作为训练资料，有 79 只水系，61 只普通系的宝可梦。那么，我们可以将分类问题看作，抽取一个样本，它来自于水系/普通系的概率。



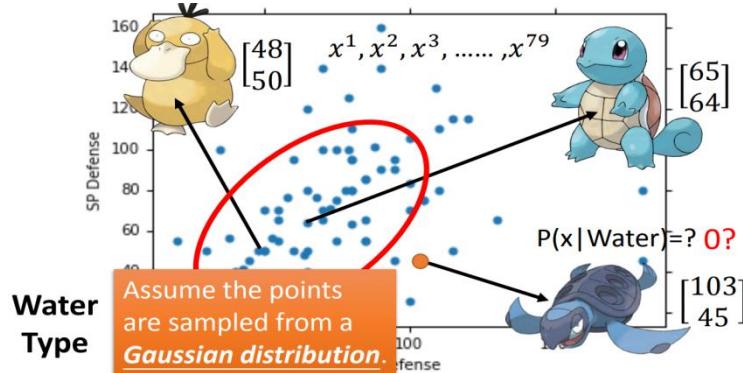
Water and Normal type with ID < 400 for training,
rest for testing

Training: 79 Water, 61 Normal

接下来的问题是，假设我们拿到了一只不存在于训练资料的宝可梦（一只海龟），如何判断出它是水系还是普通系的呢？

$$P(x|C_1) = ? \quad P(\text{Sea turtle} | \text{Water}) = ?$$

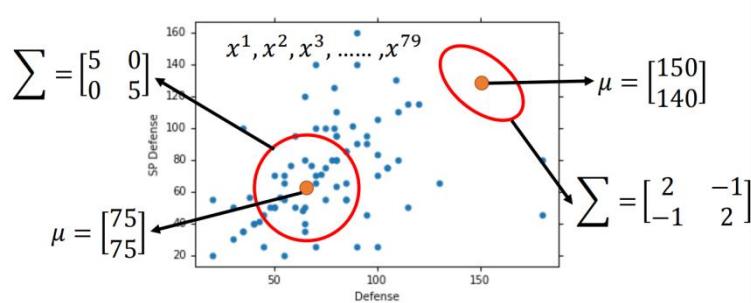
为了简化问题，我们只考虑宝可梦的 Defense 和 SP Defense 两个属性，我们画出图像，发现海龟样本在图中是不存在的，那是否概率为 0 呢？显然不是！因为海龟只是在这一笔训练资料中不存在，它也是宝可梦中的精灵之一。



我们假设上图中的点遵循高斯分布（Gaussian Distribution），其概率密度函数如下：

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

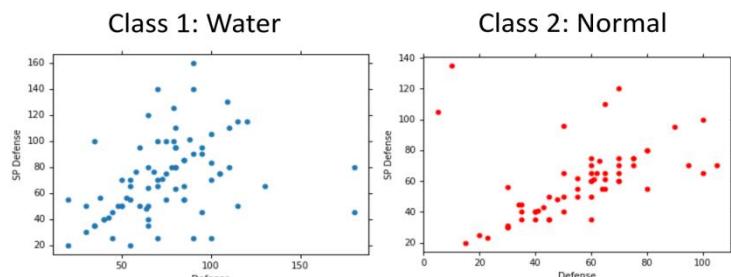
其中， μ 代表均值， Σ 代表协方差矩阵（Covariance Matrix），该高斯分布函数由二者共同决定，因此我们需要从训练资料中找到 μ 和 Σ ，为了实现上述功能，我们利用极大似然估计（Maximum Likelihood）的思想定义一个函数。



我们知道，对于水系精灵的 79 个点，我们可以从任意的高斯分布中取样出来（如上图所示），因此我们可以将 79 个高斯分布乘起来，得到如下的函数：

$$L(\mu, \Sigma) = \prod_{i=1}^{79} f_{\mu, \Sigma}(x^i)$$

而极大似然估计的思想在于，需要找到最优的 μ^* 和 Σ^* ，使得上述函数取得最大值。事实上，不管你之前是否了解过高斯分布，你都能很自然地猜测上述函数取平均值的时候取得最大值。我们可以从几何上直观地看出，你的样本点越接近于中心越好。

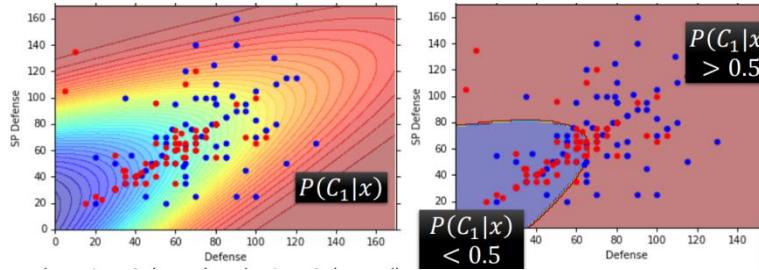


$$\mu^1 = \begin{bmatrix} 75.0 \\ 71.3 \end{bmatrix} \quad \Sigma^1 = \begin{bmatrix} 874 & 327 \\ 327 & 929 \end{bmatrix} \quad \mu^2 = \begin{bmatrix} 55.6 \\ 59.8 \end{bmatrix} \quad \Sigma^2 = \begin{bmatrix} 847 & 422 \\ 422 & 685 \end{bmatrix}$$

我们在上面的计算中得到了两组不同的 (μ, Σ) , 对应两个函数, 它其实代表的就是在两类中抽到样本 x 的概率, 可表示为: $P(x|C_i), i = 1, 2$ 。这其实是先验概率 (Prior Probability) , 根据贝叶斯公式, 我们可以计算出后验概率 (Posterior Probability) :

$$P(C_i|x) = \frac{P(x|C_i)P(C_i)}{P(x|C_1)P(C_1) + P(x|C_2)P(C_2)}, i = 1, 2$$

那么分类问题的思想就是, 如果 $P(C_i|x)$ 大于 0.5, 就归类于第 i 类, 否则归为另一类, 具体的结果如下图所示:



但是我们发现结果并不是很好, 准确率仅为 47%, 我们将宝可梦所有的 7 个数值属性全部加入模型中, 得到的准确率仍然只有 54%, 因此我们可以对上述函数进行进一步处理。对于例子中的两类高斯密度函数, 都有不同的 Σ 。为了简化计算参数, 我们统一使用一样的协方差矩阵 Σ 。使用更少的参数, 也一定程度上能减少函数过拟合的情况。

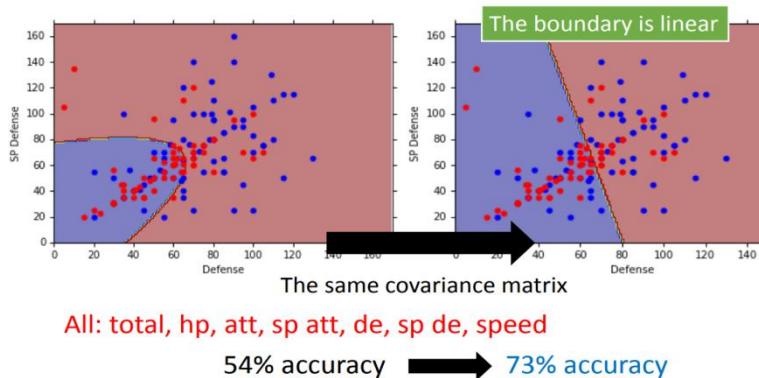
当使用了一样的 Σ 时, 似然估计函数也需要进行修改, 我们需要将剩下 61 个普通系的高斯分布也乘到原来的函数中, 计算方法如下:

$$L(\mu^1, \mu^2, \Sigma) = \prod_{i=1}^{79} f_{\mu^1, \Sigma}(x^i) \times \prod_{j=1}^{61} f_{\mu^2, \Sigma}(x^{79+j})$$

其中, μ^1, μ^2 的计算方法与之前不同, 唯一的不同是 Σ 的计算, 我们可以用一个加权平均的算法来进行计算:

$$\Sigma = \frac{79}{140} \times \Sigma^1 + \frac{61}{140} \times \Sigma^2$$

在统一了 Σ 之后, 准确率瞬间有了提高, 从 54% 提高到了 73%。



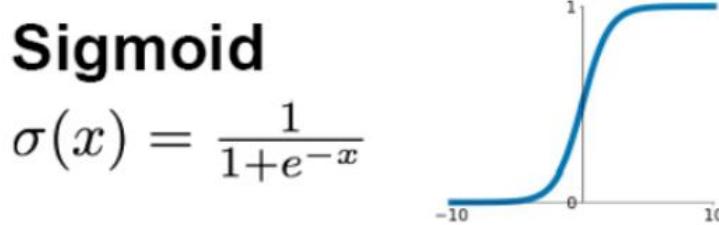
以上就是分类问题的主要步骤。当然, 在实际问题中, 我们更常见的是一些多分类问题 (Multi-class Classification), 它仍然可以用线性模型进行求解, 我们将在后面的小节中进行介绍。

2.2.3 探究：为什么分类问题也是线性模型

事实上，对于上述问题，当我们统一了协方差矩阵 Σ 之后，分界线变成线性的了。考虑后验概率 $P(C_1|x)$ ，我们可以将上下同时除以分子，得到改写后的表达式如下所示：

$$P(C_1|x) = \frac{1}{1 + \frac{P(x|C_2)P(C_2)}{P(x|C_1)P(C_1)}}$$

我们将上式中的红色部分改写 $\exp(-z)$ 的形式，就可以将上述后验概率转化成一个 sigmoid 函数的形式：



对于 z ，我们进一步展开如下：

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

在上式中，我们的 $P(C_1)$ 和 $P(C_2)$ 就分别代表第一类和第二类中出现的次数，因此可以看作一个常量，我们接下来将高斯分布代入前面这一项，结果如下：

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

由于统一了 Σ 之后，我们又可以进一步进行简化：

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

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

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

综上所述， $P(C_1|x)$ 可以写成一个 $\text{sigmoid}(wx + b)$ 的形式，故而它也是线性模型。

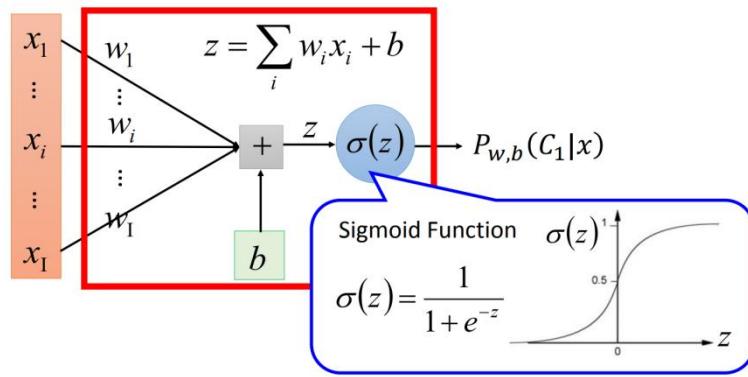
2.3 逻辑回归 (Logistic Regression)

在上一节中的分类问题中，我们引出了 sigmoid 函数，而这个 sigmoid 函数其实还有另一个用处，就是逻辑回归。在本节中我们将对逻辑回归的步骤进行介绍，并与线性回归进行比较，最后再对多目标分类（Multi-class Classification）进行介绍。

2.3.1 逻辑回归的步骤

与线性回归一样，逻辑回归同样分为三个步骤。

第一步是定义一个函数集合，我们希望找到一个关于 w 和 b 的函数，它其实是我们上一节分类问题中讲到的后验概率，同样，我们可以将这个概率函数看成一个 sigmoid 函数。



第二步是构造一个损失函数来评估我们第一步定义的函数好坏。在这边，假设我们有 N 笔训练资料，并且每一个训练数据都与其应该被分为的类别一一对应。同时，我们假设这些数据是由第一步中的后验概率产生的。

Training Data	x^1	x^2	x^3	x^N
	C_1	C_1	C_2		C_1

因此，我们的需要去计算产生全部 N 笔训练资料的概率，根据上图的训练资料，我们就可以给出如下函数：

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

给出损失函数，我们就需要取求解最优的 w 和 b ，使得损失函数最小，但是根据上一节的内容我们知道， $L(w, b)$ 是一个似然估计函数，因此它不能当作损失函数，同时连乘函数对于梯度计算也不是很方便，因此我们需要对损失函数进行一个数学上的变换，即对上式两边同时取负对数，得到损失函数如下：

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

对于上述表达式，我们还可以进一步进行改写，假如我们给第 i 笔训练资料的类别用 \hat{y}^i 进行编号，并且如果训练资料 x^i 属于第一类， \hat{y}^i 就为 1，如果属于第二类， \hat{y}^i 就为 0，我们可以将这一项改写成如下形式：

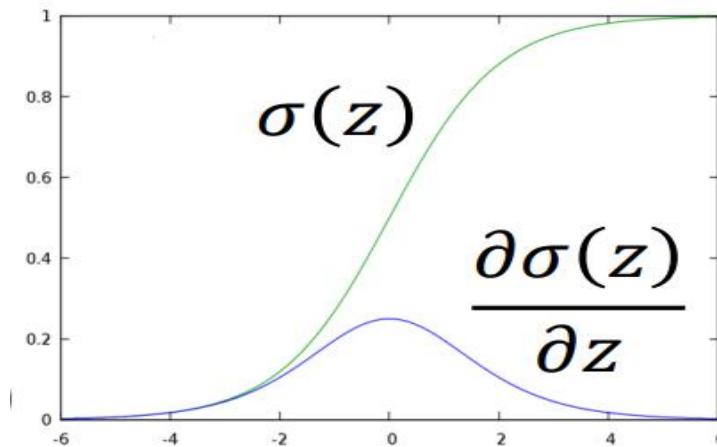
$$-[\hat{y}^i \cdot \ln f_{w,b}(x^i) + (1 - \hat{y}^i) \cdot \ln(1 - f_{w,b}(x^i))]$$

对于上述表达式，其实是一种名叫交叉熵（cross-entropy）的函数，它代表的含义是两个伯努利分布之间的接近程度，定义如下图所示：

Distribution p: $p(x=1) = \hat{y}^n$ $p(x=0) = 1 - \hat{y}^n$		Distribution q: $q(x=1) = f(x^n)$ $q(x=0) = 1 - f(x^n)$
---	---	---

$$H(p, q) = - \sum_x p(x) \ln(q(x))$$

第三步就是要对第二步中的损失函数求解优化问题，这个问题用梯度下降计算即可，需要注意的是，sigmoid 函数有一个奇妙的特性，假设我们有一个 sigmoid 函数： $\sigma(z)$ ，它关于 z 的微分为： $\sigma(z) \cdot (1 - \sigma(z))$ ，图像如下：



根据一系列计算，我们可以得到梯度下降的权重更新如下，我们发现，这个公式与线性回归的梯度下降中的权重更新的公式是一样的：

$$\begin{aligned} \text{cost } & L(\vec{w}, b) = -\frac{1}{m} \sum_{i=1}^m [\hat{y}^{(i)} \log(f_{\vec{w}, b}(\vec{x}^{(i)})) + (1 - \hat{y}^{(i)}) \log(1 - f_{\vec{w}, b}(\vec{x}^{(i)}))] \\ \text{repeat } & \left. \begin{array}{l} w_j = w_j - \alpha \frac{\partial}{\partial w_j} L(\vec{w}, b) \\ b = b - \alpha \frac{\partial}{\partial b} L(\vec{w}, b) \end{array} \right\} \text{ simultaneous updates} \end{aligned}$$

$$\frac{\partial}{\partial w_j} L(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^m (f_{\vec{w}, b}(\vec{x}^{(i)}) - \hat{y}^{(i)}) x_j^{(i)}$$

$$\frac{\partial}{\partial b} L(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^m (f_{\vec{w}, b}(\vec{x}^{(i)}) - \hat{y}^{(i)})$$

2.3.2 逻辑回归 v.s 线性回归

接下来，我们来对比一下 Logistic 回归与线性回归。首先是第一步定义函数，逻辑回归定义的函数是 sigmoid 函数，它的输出值介于 0 和 1 之间，而线性回归中定义的函数关于 x 是线性的，它可以输出任何值。

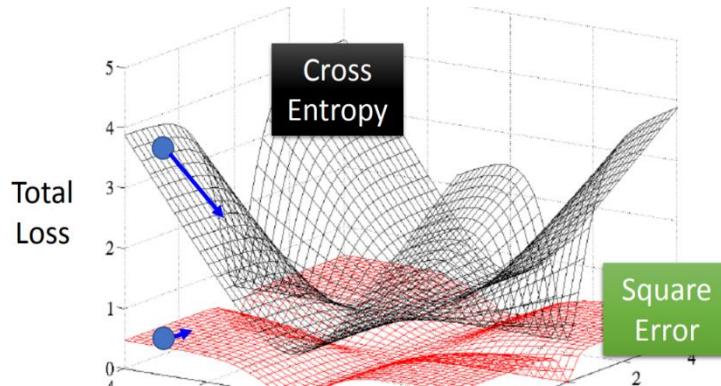
<u>Logistic Regression</u>	<u>Linear Regression</u>
$f_{w,b}(x) = \sigma\left(\sum_i w_i x_i + b\right)$	$f_{w,b}(x) = \sum_i w_i x_i + b$
Output: between 0 and 1	Output: any value

第二步是确立损失函数，对于逻辑回归，它的损失函数是交叉熵函数，而对于线性回归，我们用一个均方误差（MSE）即可。那么我们能不能将均方误差用在逻辑回归呢？答案是：不太好。事实上，当我们用均方误差作为分类问题的损失函数时，其实会出现如下问题：

$$\begin{aligned} \text{If } f_{w,b}(x^n) = 1 \text{ (far from target)} &\rightarrow \partial L / \partial w_i = 0 & \text{If } f_{w,b}(x^n) = 1 \text{ (close to target)} &\rightarrow \partial L / \partial w_i = 0 \\ \text{If } f_{w,b}(x^n) = 0 \text{ (close to target)} &\rightarrow \partial L / \partial w_i = 0 & \text{If } f_{w,b}(x^n) = 0 \text{ (far from target)} &\rightarrow \partial L / \partial w_i = 0 \end{aligned}$$

如上图所示，假设我们的 x^n 属于第一类，那么 $f_{w,b}(x^n)$ 为 1 是接近目标的，为 0 离目标较远，反之如果 x^n 属于第二类，那么 $f_{w,b}(x^n)$ 为 0 是接近目标的，为 1 离目标较远。但不管距离目标远近，梯度算出来都是 0，这样我们的损失函数就没什么作用了。

事实上，我们可以把交叉熵和平方误差在同一个平面中画出来，对于交叉熵函数，距离目标越远，梯度下降的步伐会变大，而对于平方误差，有可能会在距离目标比较远的地方梯度下降的算法卡住，可能没有办法比较有效地进行参数更新。



而对于第三步，在之前我们也进行了一些数学上的证明，我们发现逻辑回归和线性回归在做梯度下降时，参数更新的公式是一样的。因此，逻辑回归与线性回归的主要区别还是在损失函数的选择上，这一点我们也会在后续章节再次进行介绍。

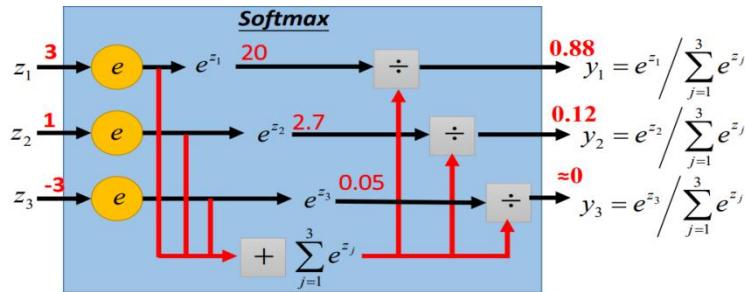
2.3.3 多分类问题 (Multi-class Classification)

我们在之前主要讨论的是二分类问题，但在深度学习中，我们通常接触的经典案例，比如图像分类，一般都是有多个类别的，这一问题称为多分类问题。

那这一问题该如何处理呢？以三分类问题为例，当我们遇到多分类问题时，我们需要将 sigmoid 函数用 softmax 函数进行替换，那什么是 softmax 函数呢？Softmax 函数实际上是 sigmoid 函数的一种多元化推广，它同样可以将一组任意实数转换为表示概率分布的 0, 1 之间的实数，考虑三个类： C_1, C_2, C_3 ，它们分别有与自身对应的 w , b 和 z ，如下所示：

$$\begin{array}{lll} C_1: w^1, b_1 & z_1 = w^1 \cdot x + b_1 & \text{Probability:} \\ C_2: w^2, b_2 & z_2 = w^2 \cdot x + b_2 & \blacksquare 1 > y_i > 0 \\ C_3: w^3, b_3 & z_3 = w^3 \cdot x + b_3 & \blacksquare \sum_i y_i = 1 \end{array} \quad y_i = P(C_i | x)$$

我们将 z_i 做如下变换：首先概率需要大于 0，因此需要将 z_i 经过一个 \exp 运算，其次，所有类别的概率总和为 1，因此需要再做一次归一化，变换流程如下所示：

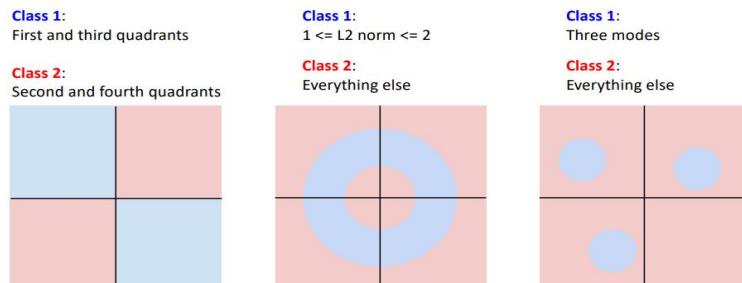


同样，我们仍然可以运用交叉熵作为损失函数，那这边对于多分类问题， \hat{y} 就是一个独热向量（one-hot vector），即当 x 属于第 i 类时，向量有且仅有第 i 维是 1，其余元素全是 0。接下来的事情就与前文所讲的基本相似了。

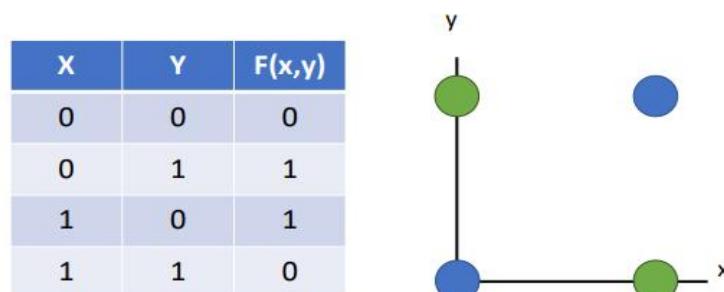
2.3.4 从逻辑回归到深度学习

我们在本书 1.2 中介绍深度学习时曾经说过，深度学习是由一系列逻辑回归组成的，但是当时我们甚至连逻辑回归本身都没有介绍，可能大家现在对深度学习和神经元还是有点懵懵懂懂。接下来，我们将从逻辑回归开始，向大家正式引入深度学习，作为本章最后一节的内容。

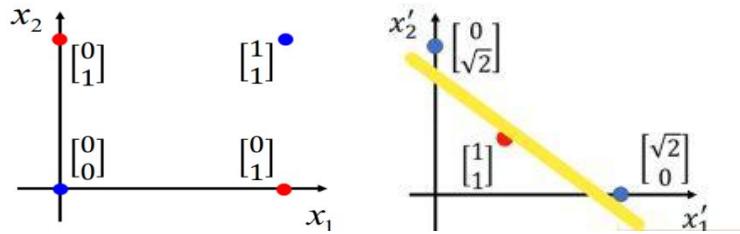
事实上，逻辑回归是有很强的限制的。我们知道，逻辑回归也是一个线性分类器（Linear Classifier），而对于线性分类器，我们需要用一条直线将不同的类别划分开来。但是以下例子对于线性分类器都是比较难处理的（hard cases）。



但大部分人或许更喜欢下面这个例子，考虑四个点： $[0, 0], [0, 1], [1, 0], [1, 1]$ ，将这四个点花在二维坐标系中，按照对角线进行分颜色染色。我们发现，不存在一条直线能将这两类点完全分隔开来。

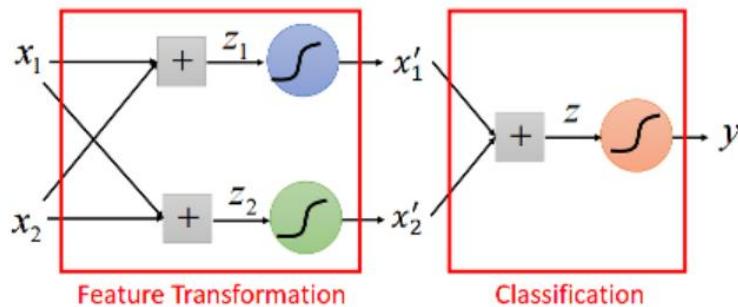


当然，对于上述例子，假如我们非用逻辑回归做不可，其实也不是不行。常见的做法是特征变换（Feature Transformation），即当原来的特征不好做时，可以通过某种变换，让原来的特征能够被逻辑回归处理。

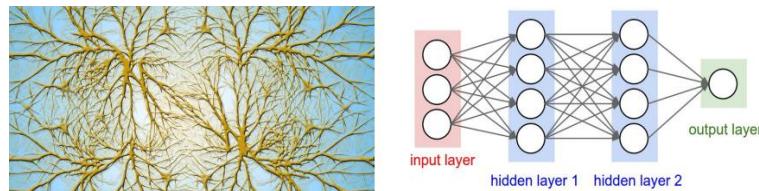


但并不是所有问题都能通过这样的方法进行求解的，因为假如我们能够通过自己的智慧来找到一个合适的特征变换，那人工智能的“智能”二字何在呢？

事实上，对于一些比较复杂的情况，我们通常会利用级联逻辑回归（Cascading Logistic Regression）来让机器自动产生特征变换，从而能够求解线性分类问题，如下图所示：



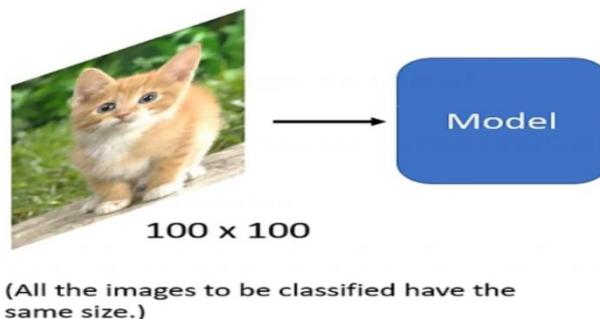
我们发现，把多个逻辑回归级联起来会产生非常不错的效果，而每个逻辑回归与生物细胞中的神经元长得很像，因此我们将每一个逻辑回归称为一个神经元（neuron），而由多个逻辑回归连接起来的架构称为神经网络（neural network），这就是神经网络与深度学习的由来。



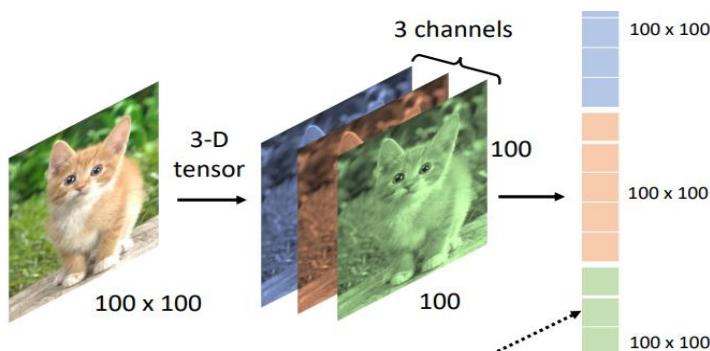
3 卷积神经网络 (Convolutional Neural Network)

3.1 背景

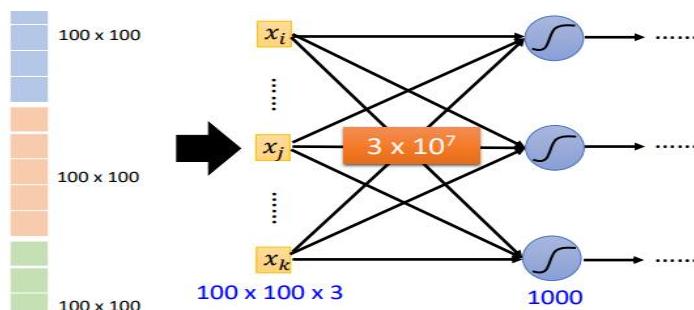
在图片分类中，我们需要给机器一张图片，并让它决定这张图片中有什么东西，在这个工作中，我们假设所有输入的图片的大小都是一致的（如果图片有大有小，将图片进行缩放即可）。



对于计算机来说，一张图片是一个三维的张量（Tensor），其三维分别代表着图片的宽（w），高（h）以及 Channel 的数目（n）。事实上，一张彩色的图片的每一个像素（pixel）都是由 R, G, B 三个颜色所组成的，我们在分类的过程中需要将图片所对应的三维张量进行拉直，变成 n 个 $w \times h$ 的二维张量，并将它们丢入神经网络中作为输入即可。



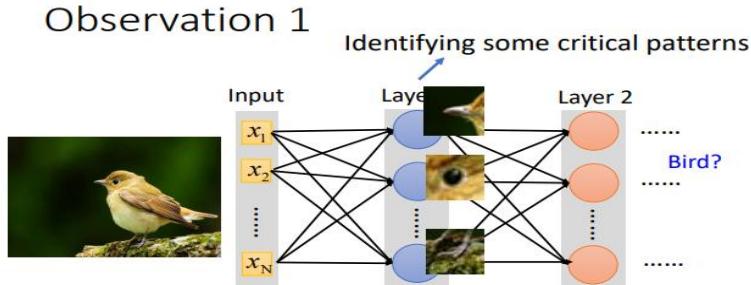
考虑神经网络的架构，事实上，对于上述向量，如果将它丢入全连接神经网络，整个网络的特征的长度就是 $100 * 100 * 3$ ，假设第一层神经元的数目有 1000 个，那么第一层的 weight 就有个，这个数量级无疑是庞大的。换言之，假设我们需要识别的物种在地球上的总数都没有超过 30M，似乎使用神经网络显得意义不大了。



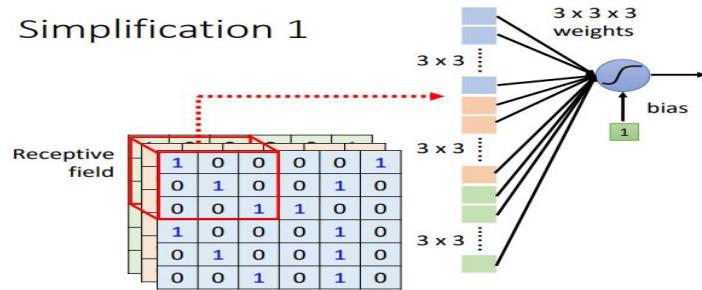
3.2 卷积神经网络如何减少参数?

3.2.1 感受野 (Receptive Field)

事实上，针对这些图片的特性，人们做出了一系列的观察，从而能够帮助我们减少参数数量，从而简化我们的训练任务。

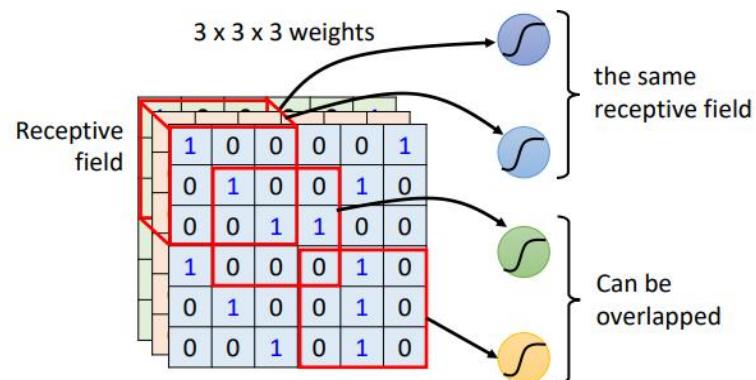


第一个观察是识别一些关键的特征，比如说我们需要识别一只鸟，在隐藏层中，假如有一个神经元看到了鸟嘴，鸟的眼睛和鸟爪这些特征，将它们综合起来，我们就可以说看到了一只鸟。于是我们可以让机器通过判断指定的特征是否出现来得出结论，这样就没有必要将整张图片作为输入了。



针对这种观察，人们提出了一个全新的网络架构，与我们之前介绍的神经网络不同，在该架构中，通常会设定一块被称为感受野 (Receptive Field) 的区域，每一个神经元只关心一个与之对应的感受野即可。

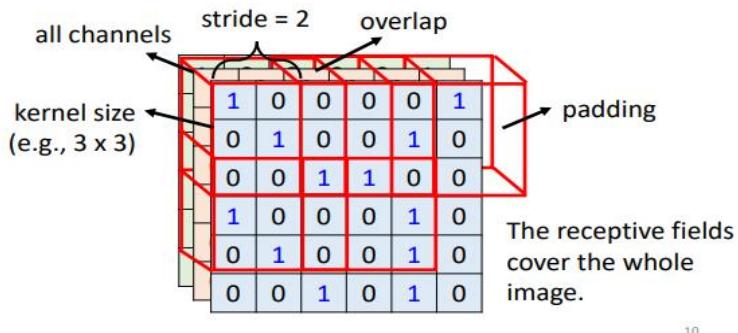
感受野的选取是人为决定的，不同的神经元所对应的感受野可以是不同大小的，也可以重叠，可以是同一个，甚至可以只关注某些指定的通道。



尽管如此，但它仍然是有一些经典的设计方法，最经典的安排方式是考虑所有的通道（channel），因此我们在描述它的感受野时，只需要描述其宽和高即可，它们合起来称为核大小（Kernel Size），一般设置成 3×3 ，此外，一般来说，对于同一个感受野，会有多个神经元对它进行“守备”（通常设为64）。

3.2.2 步长(stride)和补值(padding)

接下来我们需要考虑的是不同的感受野之间的关系，通常人们会把左上角的感受野往右移一点，从而制造出一个新的守备范围，这个移动的量被称为步长（stride），它是一个超参，通常设置为1或者2。

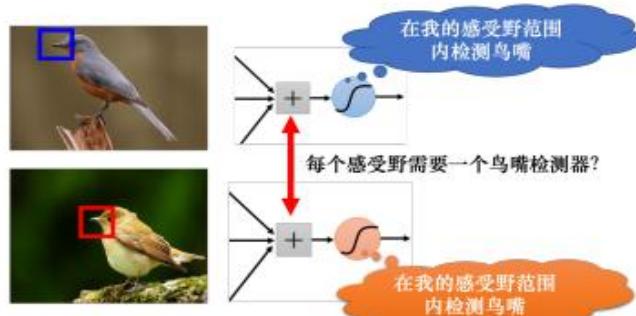


在实际问题中，我们希望不同的感受野之间有高度的重叠，从而确保不会遗漏掉一些重要特征，但是这样我们仍然会产生一些新的问题。以上图为例，当我们根据上图所设置的步长对感受野进行移动时，第二次移动就会发现有资源浪费（overlap）的情况，我们当然不能舍弃掉这一个感受野，因为舍弃它们就相当于舍弃掉了图片的一部分特征，在这里的做法是补值（padding）。

当然，除了横移，我们还需要沿着另一个方向移动，最终我们需要将整个图像完全被覆盖，因此整张图片的每一个像素都能被感受野覆盖，即存在神经元在侦测它。

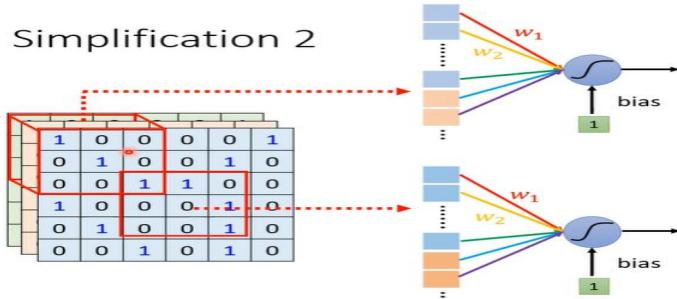
3.2.3 参数共享与滤波器(Filter)

继续观察图片的特征我们发现，同样的特征会出现在图片的不同区域。那这会出现什么样的问题呢？假如两张鸟的图片中，鸟嘴出现在图片的不同的位置，那么这两张鸟嘴的感受野就会不同，但它们确实是一样的东西。

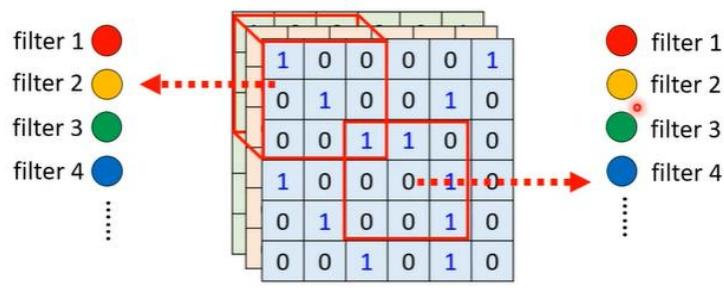


对于上述问题，我们可以用一个很形象的比喻来说明问题：假如电气学院，计算机学院和自动化学院的培养方案中都有程序设计课程，原先的想法就是在每一个学院中分别开设程序设计的课程，但其实这样就有点浪费教学资源了，我们完全可以通过大班教学来完成这一门课的教学任务。在卷积神经网络中，这一做法称为参数共享（Parameter Sharing）。

回到原来的问题，参数共享实际上就是将多个表示同一个特征的感受野所对应的神经元的权重设置成同一个值。



需要注意的是，尽管这两个感受野对应的神经元的参数是一模一样的，但它们的输出也是不一样的，这是因为它们的输入是不一样的，因此，很自然地可以想到，对于同一个感受野上的不同的神经元，通常是不会让它们共享参数的。

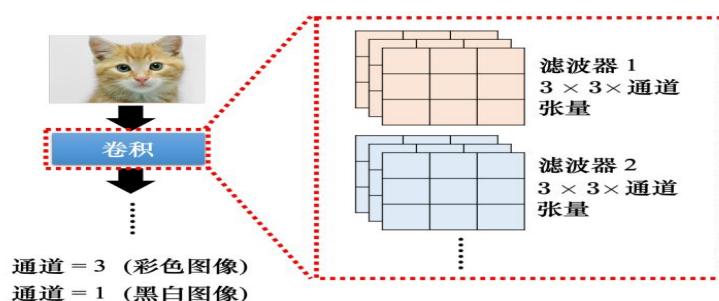


事实上，每个感受野都有一组神经元，通常人们共享参数的方法是将两个感受野中的神经元一一对应共享，这样保证了每一个感受野有且仅有组参数，这样一组参数通常被称为滤波器（filter）。

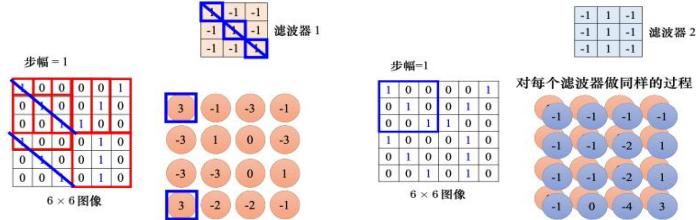
3.3 卷积神经网络的计算和结构

3.3.1 从滤波器到卷积层计算

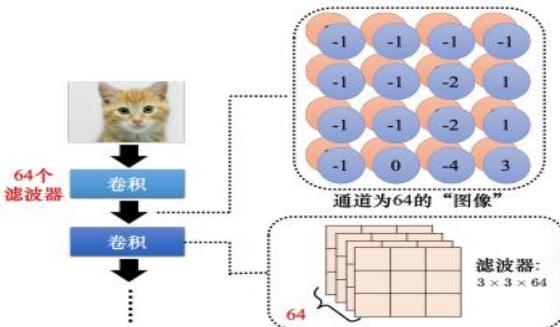
卷积层里面有很多滤波器，这些滤波器的大小是 $3 \times 3 \times$ 通道大小，每一个滤波器的作用是从图片中抓取某一个特征。



那么这些滤波器是怎么抓取特征的呢？以下面这个例子进行说明：假设有一个 6×6 的图片，将每一个滤波器放在图片的左上角，将滤波器与图像中的 3×3 的“子矩阵”进行内积的运算，并且进行横向和纵向移动，这实际上就是前文中所说的步长，直到滤波器移动到右下角为止，具体过程如下图所示：



对每一个滤波器进行上述过程的计算，得到一系列的数字，这群数字被称为特征图（Feature Map），它将图片通过卷积计算得出的。我们可以将这个特征图看成一个新的“图片”，只是这个图片的通道不是 RGB 或者黑白，其图片的通道数与滤波器的数目相等。



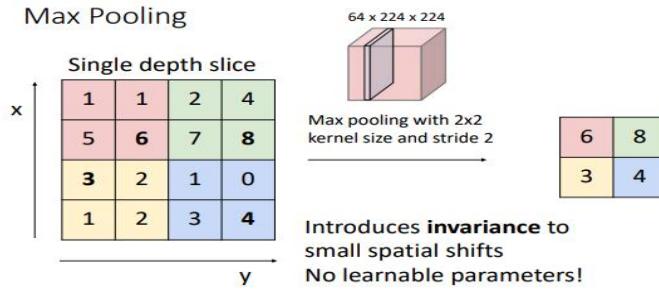
在实际操作中，这样的一系列运算被称为卷积层（Convolutional Layer），我们可以通过叠多层的网络来进行实操，这样的优点是可以让我们看到更多的特征，比如说，在第二层，当我们进行完了一系列运算后，我们可以看到的图片从原先的 3×3 扩充到了 5×5 ，以此类推。

3.3.2 池化层 (Pooling Layer)

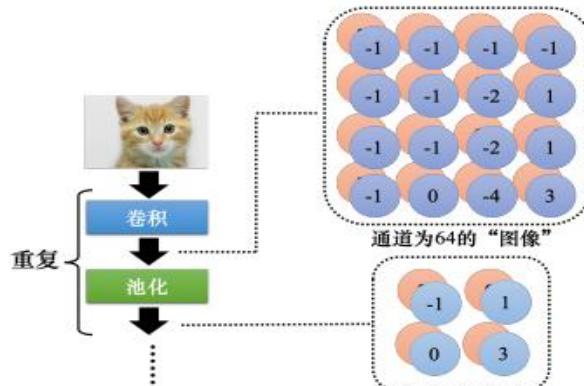
实际上，CNN 在做影像辨识时还有另一个常用的东西--池化（Pooling），这个技术来自于以下观察，即将一张比较大的图片做子采样（sub-sampling），换言之将一张图片缩小，仍然不应该影响原先识别的结果：



池化的方法有很多种，其中一种方法叫做最大池化（Max Pooling），具体做法是将每个滤波器产生的一组数字进行分组，并取得其中最大的池化值作为保留值，其他特征值全部舍弃。



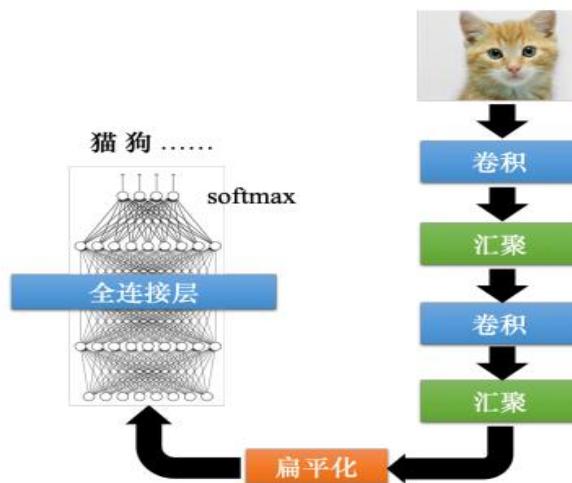
当然池化的方法还有很多种，总体来说，人们在做完卷积操作之后，会加上一个池化层，其作用是将原始的图片变小，在实操中，卷积+池化会被重复使用：



但实际上池化也不是必须的，假如我们在识别一张很细微的图片，对它做子采样肯定是对我们的识别结果产生一定影响的，因此，随着近年来计算能力越来越强，人们也会尝试将卷积层从头到尾使用，而放弃池化。

3.3.3 卷积神经网络的架构

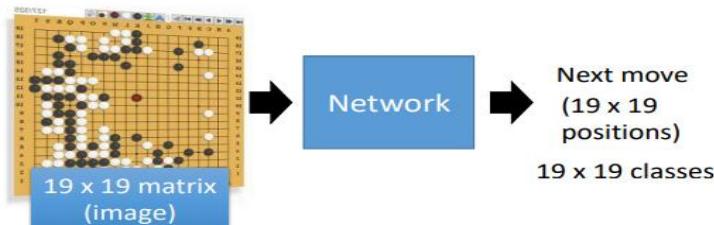
综上所述，一个经典的图像识别的模型架构如下，首先将图片经过一系列的卷积和池化（可有可无）的运算，在最后一次运算完成之后，需要将最终的输出的数值拉直成一个长的向量，并将其丢入全连接层，得到最终的结果。



3.4 卷积神经网络的应用与优缺点

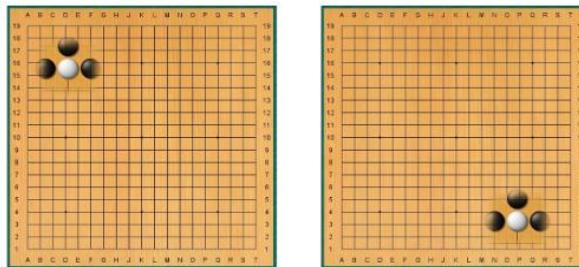
3.4.1 最知名的应用——Alpha Go

除了图像识别的任务，CNN 还有另外一个常见的应用，就是下围棋（AlphaGo），实际上，下围棋就是一个分类的问题，它的网络输入就是整个棋盘上的黑白子的位置，我们需要输出的是下一步需要落子的位置。我们需要将棋盘表示成一个 19×19 维的向量，假如某一个位置放的是黑子，记为 1，白子记为 -1，没有放置任何棋子记为 0.



对于上述问题，用 CNN 解决的效果会比全连接神经网络更好，这是因为我们完全可以将这个棋盘看作一张图片，其解析度为 19×19 （很小），在 AlphaGo 的原始论文中，用 48 个数字（channels）来描述棋盘上的每一个位置（pixels）。

既然下围棋的问题可以用 CNN 来解决，说明棋盘和图片有相似之处，其相似之处主要有两点，第一点是一些重要的特征是在很小的区域就可以被识别，第二点是相同的特征可能会出现在不同的区域。



但是棋盘与图片仍然有一些不同，就是图片是可以做子采样的，而一旦截取了棋盘的一部分，整个棋局是必然会受到影响的，实际上，作为下围棋问题中最出名的问题，AlphaGo 的网络结构如下，我们发现 Alpha Go 是没有使用池化这个技术的，由此可见池化并非 CNN 要做的事情。

• Subsampling the pixels will not change the object
→ Pooling How to explain this???

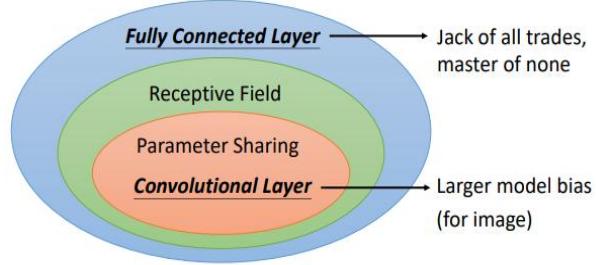
Neural network architecture. The input to the policy network is a $19 \times 19 \times 48$ image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23×23 image, then convolves k filters of kernel size 5×5 with stride 1 with the input image and applies a rectifier 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 match version of AlphaGo used $k = 192$ filters; Fig. 2b and Extended Data Table 1 show 256 and 384 filters.

Alpha Go does not use Pooling

当然 CNN 也有一些其他的应用，比如在 Speech 和 NLP 中都有一定的应用，这些在这里就不再赘述，详见 cs224n 和李宏毅老师的 NLP 课程，当然需要提及的是，在 NLP 或是语音辨识的任务中，Receptive Fields 的设计肯定是不能照搬图片上的方法的。

3.4.2 卷积神经网络的利与弊

在上述的问题中，我们首先使用的是全连接神经网络进行图像识别与分类，然后我们通过逐步优化依次引出了感受野和参数共享的概念，它们合并在一起被称为卷积层。

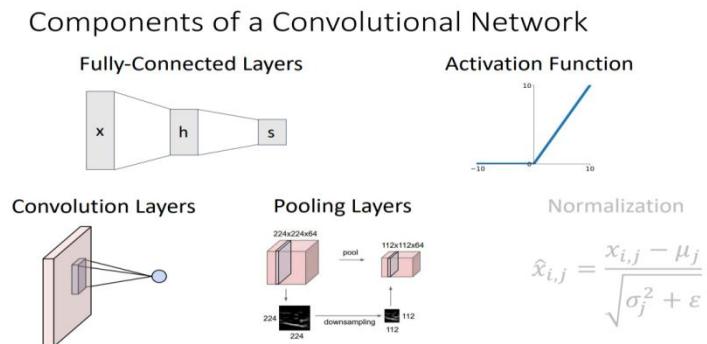


很容易发现，CNN 的模型偏差（model bias）很大，虽然我们在前文中说这是一件不好的事情，但在图像辨识的问题中，CNN 是完全够用的，因为感受野和参数共享的技术都是为特定的图像数据设定的，也就是说，这个大的模型偏差完全是在图像这个数据的接受范围之内（但如果应用在图像之外的任务，就需要谨慎了！），此外，大的模型偏差可以有效防止过拟合的发生。

当然 CNN 也是有一定的局限性的，比如说，CNN 是没法处理图片的旋转/缩放，解决办法主要有两种，第一种就是在 CNN 实操中，使用数据增广将图片进行旋转或者缩放，并将它们全部丢入网络中，第二种就是更换能够处理旋转/缩放的网络架构，比如说 Spatial Transformer Layer。

3.5 从 LeNet 到现代卷积神经网络

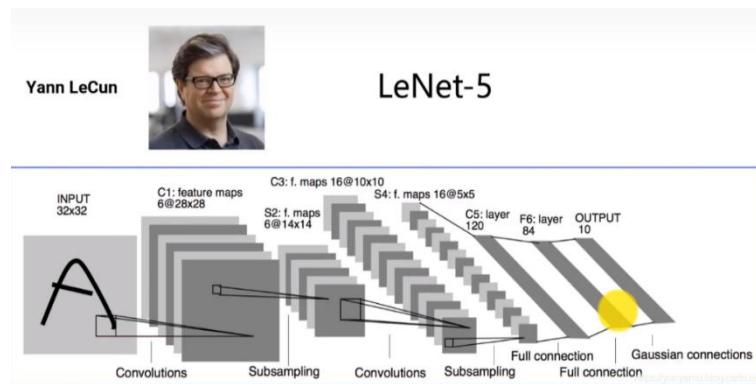
在上一章中，我们学习了卷积神经网络的基本内容，接下来我们需要学习如何去使用卷积神经网络来处理诸如图像识别之类的问题。事实上，工程实践中的卷积神经网络本质上可以认为由以下几个部分组成：输入-->卷积计算-->激励-->池化-->输出。



在本节中，我们将从最经典的 LeNet 开始讲起，并引入现代的常用于实践的（深度）卷积神经网络，比如 AlexNet，ResNet，VGG 等以及它们的架构，事实上，这些网络都或多或少活跃在诸如 ImageNet 竞赛之类的比赛之中，而它们也是依靠这个竞赛“一战成名”（注意：ImageNet 竞赛到了 2017 年以后就没了）。

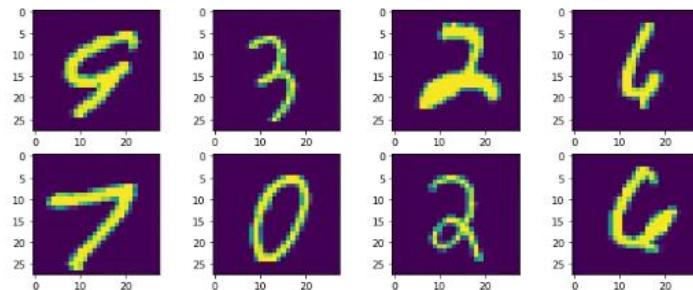
3.5.1 卷积神经网络的“开山之作”——LeNet

在 CNN 领域中，最经典的架构是由 Yann LeCun 在 1998 年提出的 LeNet-5，这个网络可以称得上是 CNN 的“开山之作”，该网络是第一个被广泛应用于数字图像识别的神经网络之一，也是深度学习领域的里程碑之一。



LeNet-5 的基本结构包括 7 层网络结构（不含输入层），其中包括 2 个卷积层、2 个降采样层（池化层）、2 个全连接层和输出层。

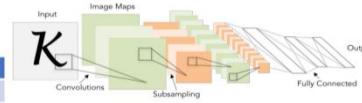
对于输入层，LeNet 接收的是 32*32 的手写数字图像。在实际应用中，我们通常会对输入图像进行预处理，例如对像素值进行归一化，以加快训练速度和提高模型的准确性。



总体来看，LeNet 由卷积编码器（两个卷积层组成）和全连接层密集块（三个全连接层）组成，每个卷积块中的基本单元是一个卷积层、一个 sigmoid 激活函数和平均汇聚层（值得注意的是，虽然最大汇聚层更有效一点，但是在当时并没有这个概念），具体网络参数如下图所示：

Example: LeNet-5

Layer	Output Size	Weight Size
Input	1 x 28 x 28	
Conv ($C_{out}=20$, $K=5$, $P=2$, $S=1$)	20 x 28 x 28	20 x 1 x 5 x 5
ReLU	20 x 28 x 28	
MaxPool($K=2$, $S=2$)	20 x 14 x 14	
Conv ($C_{out}=50$, $K=5$, $P=2$, $S=1$)	50 x 14 x 14	50 x 20 x 5 x 5
ReLU	50 x 14 x 14	
MaxPool($K=2$, $S=2$)	50 x 7 x 7	
Flatten	2450	
Linear (2450 -> 500)	500	2450 x 500
ReLU	500	
Linear (500 -> 10)	10	500 x 10



As we go through the network:

Spatial size **decreases**
(using pooling or strided conv)

Number of channels **increases**
(total "volume" is preserved!)

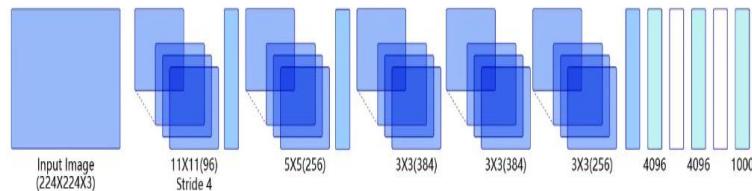
为了将卷积块的输出传递给稠密块，我们必须在小批量中展平每个样本。换言之，我们将这个四维输入转换成全连接层所期望的二维输入。这里的二维表示的第一个维度索引小批量中的样本，第二个维度给出每个样本的平面向量表示。LeNet 的稠密块有三个全连接层，分别有 120、84 和 10 个输出，输出层的 10 维对应于最后输出结果的数量。

在 LeNet 提出后，卷积神经网络在计算机视觉和机器学习领域中很有名气。但卷积神经网络并没有主导这些领域，这是因为虽然 LeNet 在小数据集上取得了很好的效果，但是在更大、更真实的数据集上训练卷积神经网络的性能和可行性还有待研究。

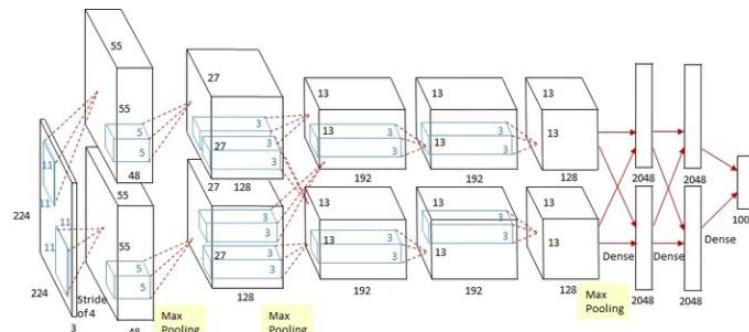
接下来我们将沿着 ImageNet 竞赛的时间线，将那些在该竞赛中具有出众表现亦或是具有划时代意义的卷积神经网络拿出来介绍。

3.5.2 深度卷积神经网络 (AlexNet)

到了 2012 年，一名叫做 Alex 的大佬在 ImageNet 竞赛中使用了一个全新的结构，并在竞赛中一举夺魁，这就是大名鼎鼎的 AlexNet，



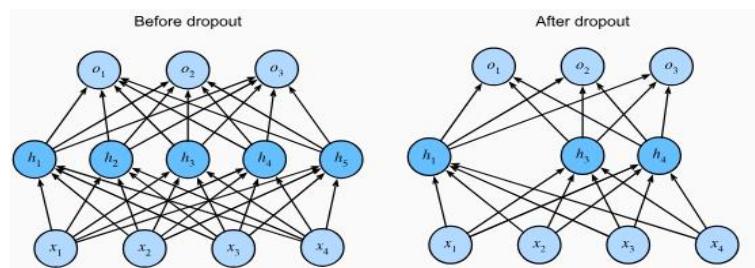
AlexNet 与 LeNet 结构相似，然而与 LeNet 相比，它具有更深的网络结构，包含 5 层卷积和 3 层全连接层，其中，每个卷积层都包含卷积核、偏置项、ReLU 激活函数和局部响应归一化 (LRN) 模块。第 1、2、5 个卷积层后面都跟着一个最大池化层，后三个层为全连接层。最终输出层为 softmax，将网络输出转化为概率值，用于预测图像的类别。具体架构如下所示：



AlexNet 的创新之处在于：首先，AlexNet 是首个真正意义上的深度卷积神经网络，它的深度达到了当时先前神经网络的数倍。通过增加网络深度，AlexNet 能够更好地学习数据集的特征，从而提高了图像分类的精度。

其次，AlexNet 首次使用了修正线性单元（ReLU）这一非线性激活函数。相比于传统的 sigmoid 和 tanh 函数，ReLU 能够在保持计算速度的同时，有效地解决了梯度消失问题，从而使得训练更加高效。

此外，为了防止过拟合，AlexNet 采取了三种办法，第一是数据增广，通过对图像进行旋转、翻转、裁剪等变换，增加训练数据的多样性，提高模型的泛化能力。第二是局部响应归一化（LRN）的使用，它的做法是将卷积层中的每一个卷积核对应的特征图归一化，目的是为了抑制邻近神经元的响应，从而增强了神经元的较大响应。第三是采用了 Dropout 技术，在训练过程中随机删除一定比例的神经元，强制网络学习多个互不相同的子网络，从而提高网络的泛化能力。



最后，AlexNet 可以说是首次提出了用最大池化代替平均池化的方法，从而避免了平均池化层的模糊化的效果，并且步长比池化核的尺寸小，同时池化层的输出之间的重叠能够提升特征的丰富性。重叠池化可以避免过拟合，这个策略贡献了 0.3% 的 Top-5 错误率。并且，AlexNet 在使用 GPU 进行训练时，可将卷积层和全连接层分别放到不同的 GPU 上进行并行计算，从而大大加快了训练速度。像这种大规模 GPU 集群进行分布式训练的方法在后来的深度学习中也得到了广泛的应用。这一点在 AlexNet 的原始论文中也有所提及。

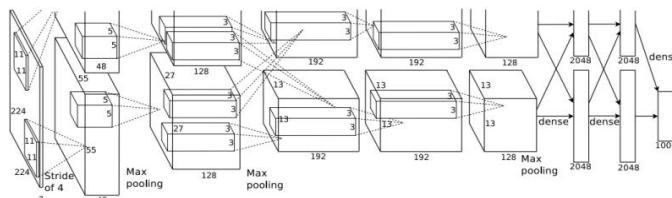
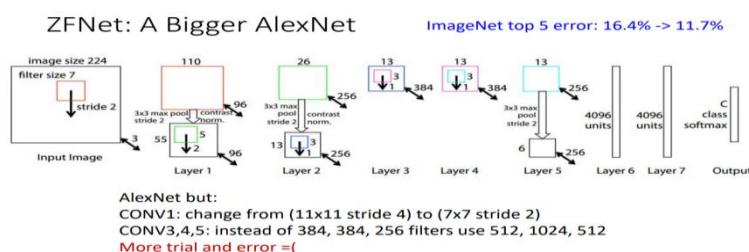


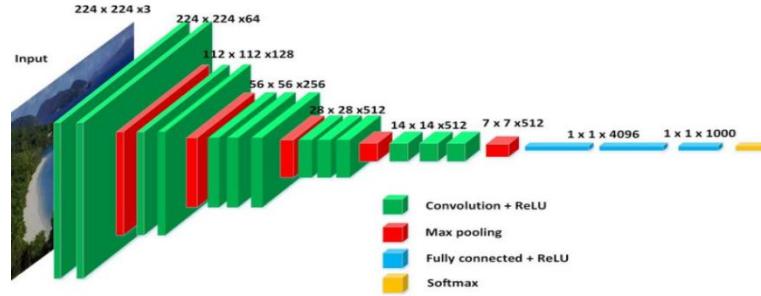
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

AlexNet 的网络架构可以说是启发性的，这不仅因为它“首创式”地提出了诸如 ReLU，Max Pooling 和双 GPU 训练的方法，还因为仅仅一年之后，Zeller 等人就在 AlexNet 的基础上，对网络架构进行了升级，并于当年(2013 年)获得了 ImageNet 的冠军，这个网络被称为 ZFNet。

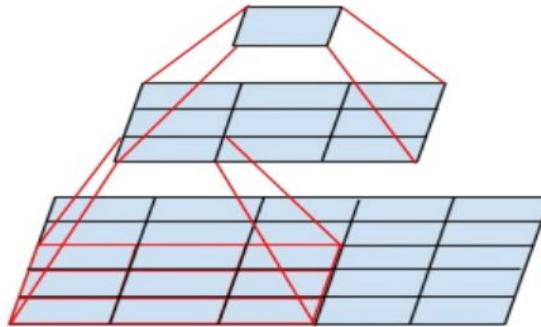


3.5.3 使用块的网络 (VGG)

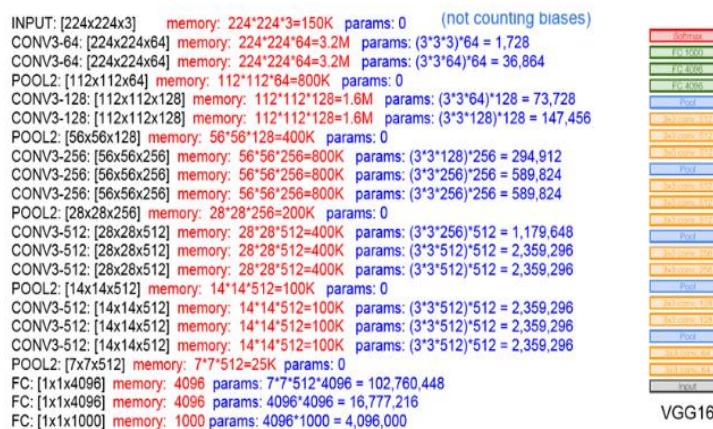
到了 2014 年，ImageNet 就已经进入了“神仙打架”的时代，这是因为当年竞赛的冠军并没有以往的“压倒性优势”，作为 2014 年的 ImageNet 竞赛亚军得主，VGG 的影像力同样非凡响。



VGG 是 Oxford 的 Visual Geometry Group 的组提出的。该网络的主要工作是证明了增加网络的深度能够在一定程度上影响网络最终的性能。VGG 有两种结构，分别是 VGG16 和 VGG19，两者并没有本质上的区别，只是网络深度不一样。

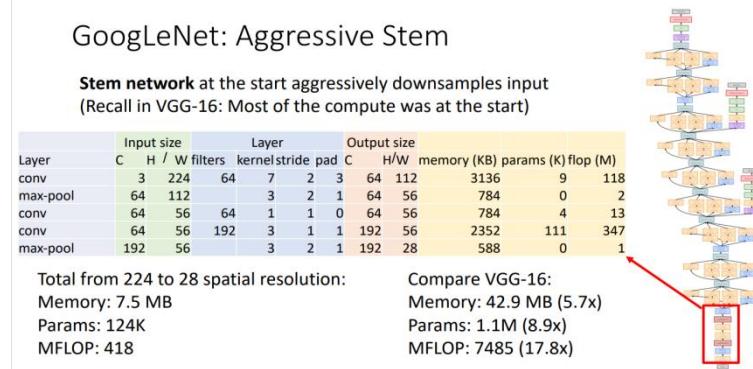


以 VGG16 为例，该网络有 13 层卷积和 3 层全连接层。与其他网络不同的是，VGG 总是严格采用 3×3 的卷积层和池化层来提取特征。简单来说，3 个 3×3 卷积核可以代替 7×7 卷积核，2 个 3×3 卷积核可以代替 5×5 卷积核，这样做的主要目的是在保证具有相同的感受野的条件下，提升了网络的深度，在一定程度上提升了神经网络的效果，VGG16 的网络结构及参数如下图所示：

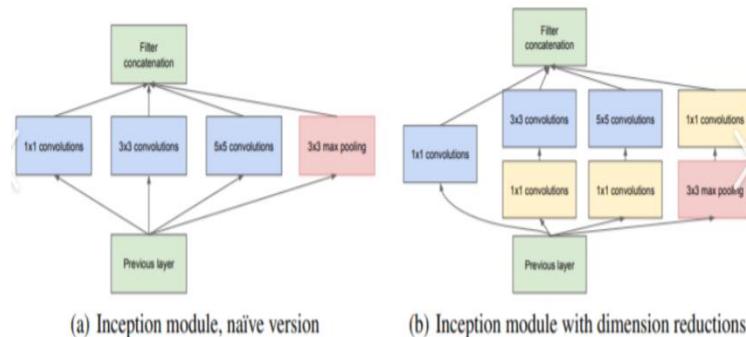


3.5.4 含并行连接的网络 (GoogLeNet)

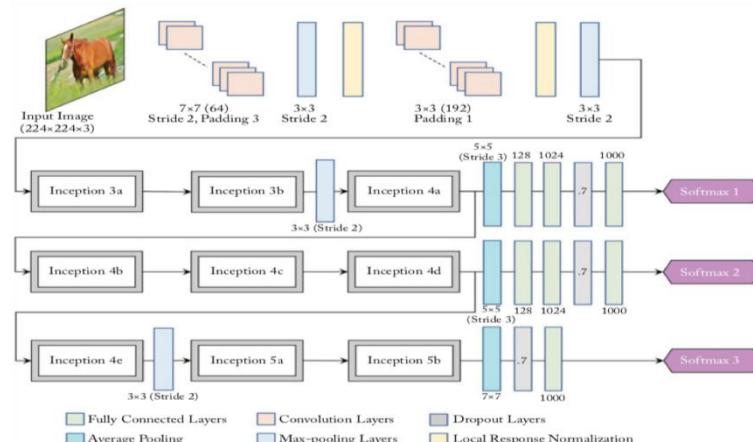
相比于同年屈居亚军的 VGG，作为冠军的 GoogLeNet 一定是有自身的过人之处的，相比于 VGG，GoogLeNet 的关注点在于减少参数，内存和计算量。



为了解决这个问题，GoogLeNet 提出了一种被称为 Inception 块 (Inception Block) 的方案，实际上，Inception-v1 的提出希望解决的本质是神经网络中可能产生的过拟合以及计算量过大的问题。作者希望达到一个稀疏性于密集计算的折衷，在 Inception 设计中，体现为较多的使用了 1×1 的卷积核。



GoogLeNet一共使用9个Inception块和全局平均汇聚层的堆叠来生成其估计值。Inception块之间的最大汇聚层可降低维度。第一个模块类似于 AlexNet 和 LeNet，Inception 块的组合从 VGG 继承，全局平均汇聚层避免了在最后使用全连接层。



3.5.5 残差网络 (ResNet)

压轴出场的著名的现代 CNN 架构就是大名鼎鼎的何恺明大神提出的 ResNet，它的引用量目前已经达到了惊人的 10W+次，ResNet 是由何恺明等 CV 界巨神于 2015 年提出来的，它不仅获得了当年 CVPR 最佳论文奖，还在诸如 ImageNet 等多个竞赛中夺冠。

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
`{kahe, v-xiangz, v-shren, jiansun}@microsoft.com`

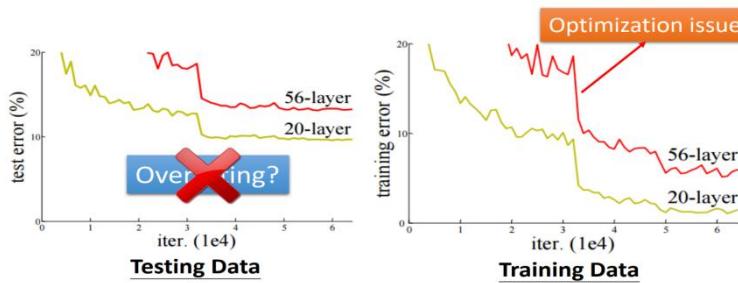
事实上，从 LeNet 到 AlexNet，再到 VGG 和 GoogLeNet，网络的层数不断在变深，但从第一章的案例中我们会发现，随着网络深度加深，深度学习网络模型的精确率并不会一直上升，而是会趋于饱和并下降。

	1 layer	2 layer	3 layer	4 layer
2017 – 2020	0.28k	0.18k	0.14k	0.10k
2021	0.43k	0.39k	0.38k	0.44k

对于 ResNet 来说，它本质上解决的贡献是要解决随着网络深度的增加，模型效果反倒变差的问题。

注意，与上图中的例子不同的是，ResNet 所处理的问题并不是过拟合，而是一种优化（在 cs231n 的课程中，Justin Johnson 称其为 underfitting）。残差网络的目的在于使深层网络表达的最优点至少不差于浅层网络。

- Gaining the insights from comparison

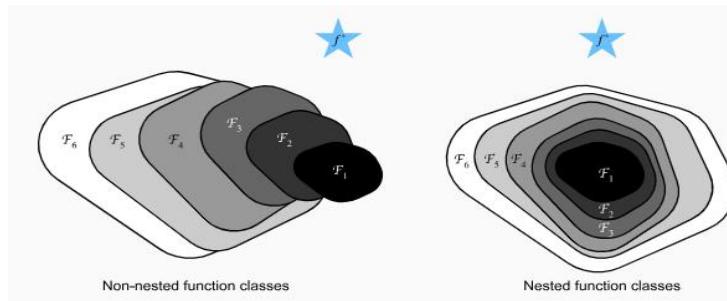


我们在第一章就讨论过机器学习/深度学习的本质就是寻找一个函数。现在，我们假设有一类特定的神经网络结构的集合 F ，它包含学习率和其他超参数的设置。机器学习/深度学习的寻找出来的最优函数 f^* 需要在这个函数集合 F 中，才能轻而易举地通过训练进行问题的求解。

但在大部分情况下，事情并不会那么顺利，很有可能最优函数 f^* 并不在集合中，因此我们需要尝试重新在 F 中寻找一个函数，使得它与 f^* 尽可能接近。

那么，我们应该怎样寻找这样的函数呢？唯一合理的可能是：需要一个更大的集合 F' 。假如 $F \subsetneq F'$ ，则无法保证新的函数集合能做的更好，甚至有可能会做的更糟。事实上，如下图所示，对于非嵌套函数（non-nested functions），较复杂的函数集合离最优值越来越远，而

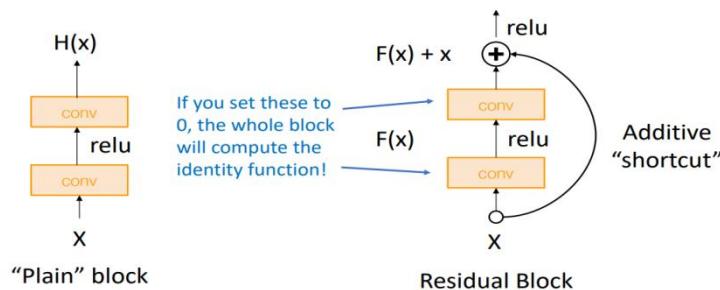
对于嵌套函数（nested functions）： $F_1 \subseteq F_2 \subseteq \dots$ 可以避免上述问题。



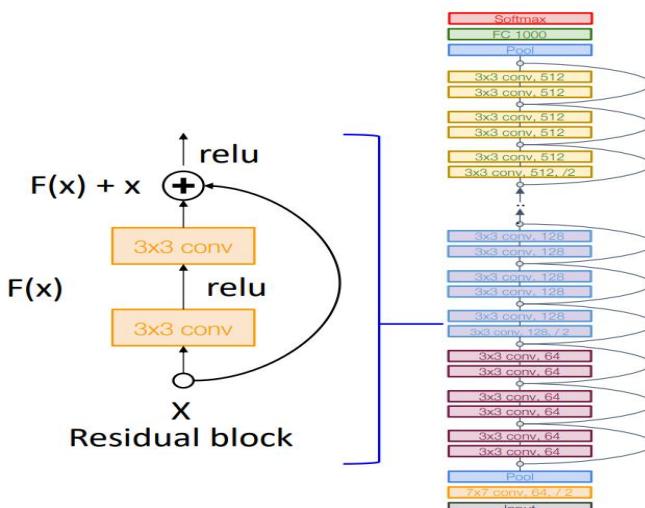
因此，对于深层网络，最简单的想法是建立恒等映射（identity function），即将多个浅层网络进行迭代生成更深层次的网络，但实践表明，恒等映射不太可能是最优操作，不过如果最优函数更接近于恒等映射而不是零映射，则求解器应该更容易通过恒等映射找到扰动，而不是将其作为新的函数学习。

在 ResNet 这篇论文中，研究者采用了一种名为残差学习的方法来训练每一个级联层，其数学表达式的定义如下：

$$y = F(x, \{W_i\}) + W_s \cdot x$$



事实上，以上公式可以由具有“捷径连接”（shortcut connections）的前馈神经网络来实现，捷径连接只需执行恒等映射，并将其输出添加到堆叠层的输出中，同时不会增加额外的参数和计算复杂性。整个网络仍然可以通过反向传播的 SGD 进行端对端训练，并且可以在不修改求解器的情况下使用公共库（例如 Caffe）轻松实现。



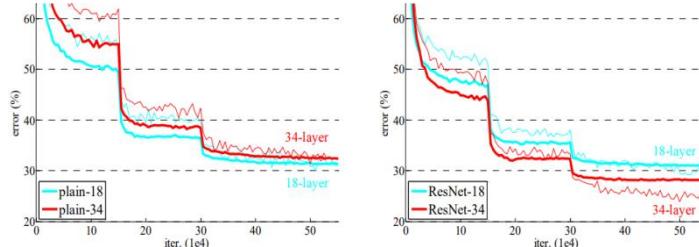
接下来，我们来讨论 ResNet，如上图所示，一个 ResNet 由一系列的残差块组成，对于普通的网络结构而言，它主要是受到 VGG 的启发，每个残差块有两个 3×3 的滤波器，基于

上述普通网络，ResNet 插入了捷径连接，将网络变成其对应的残差版本。当输入和输出具有相同的尺寸时，直接使用恒等捷径连接即可。当尺寸增加时，可以用以下两种解决办法处理，其一是捷径连接仍然执行恒等映射，为增加尺寸填充零以增加维度。该方法不会引入额外的参数；另一种是投影捷径方式用于匹配维度（通过 1×1 卷积完成）。对于这两个选项，当捷径连接跨越两种尺寸的特征图时，使用步幅为 2 处理。ResNet 的参数如下图所示：

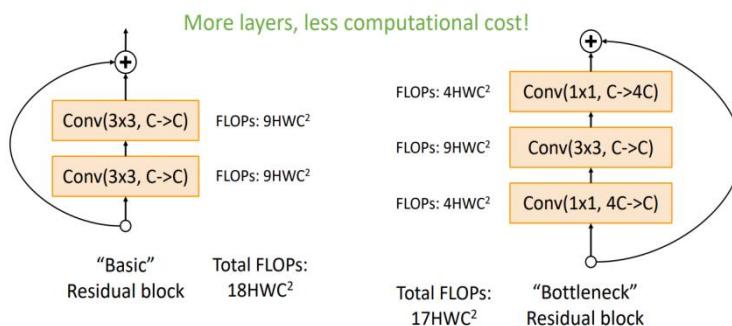
Layer	Input size		Layer			Output size		params (k)	flop (M)	
	C	H/W	filters	kernel	stride	pad	C	H/W		
conv	3	224	64	7	2	3	64	112	3136	9 118
max-pool	64	112			3	2	164	56	784	0 2

作为深度卷积神经网络，ResNet 同样是可以叠很多层的。事实上，在 PyTorch 中有一些预训练（Pretrained）的 ResNet，比如 ResNet18，ResNet34，ResNet50 等等。而为什么选择 18，34，50 等数字作为预训练 ResNet 的层数呢？其实这也是何恺明团队探索出来的。

对于上述网络，他们对于 18 层和 34 层的普通网络和残差网络分别做了实验，实验结果表明，较深的 34 层普通网比较浅的 18 层普通网具有更高的验证误差，而对于残差网络，其学习的情况相反，34 层 ResNet 比 18 层 ResNet (2.8%) 好。更重要的是，34 层 ResNet 展示的训练误差相当低，并且可以泛化到验证数据。这表明在这种情况下，退化问题得到了很好的解决，而且从增加的深度中获得了准确性的提高。



我们从算法的效率上分析，假如 ResNet 一共有 N 层，其每秒浮点运算次数（FLOPS）与层数 N 成正比 ($N \cdot H \cdot W \cdot C^2$)，当 ResNet 的层数叠到足够深时（临界 50 层），Kaiming 等人很自然想到引入了一种能减少计算开销的结构的必要性，于是瓶颈架构（Bottleneck）横空出世，这也是 Resnet 网络 50 层以下和 50 层以上最本质区别。



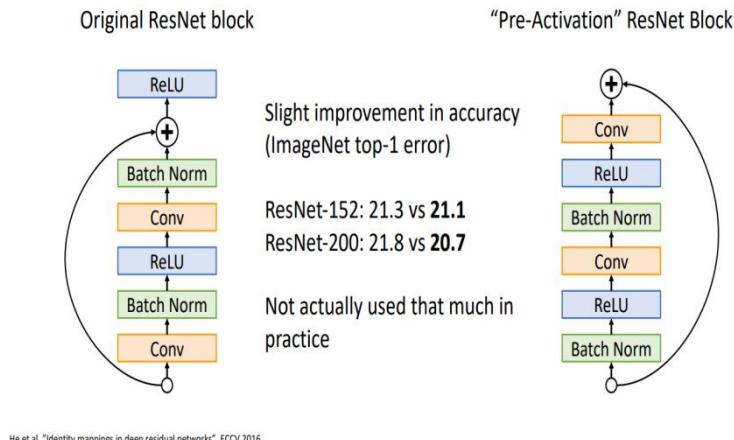
对于每个残差函数，我们使用 3 层而不是 2 层。三层是 $1 \times 1, 3 \times 3$ 和 1×1 的卷积，其中 1×1 层负责减小和增加（恢复）尺寸，使得 3×3 层成为具有较小输入/输出尺寸的瓶颈。

50 层 ResNet：我们用这个 3 层瓶颈块替换 34 层网络中的所有的 2 层块，产生 50 层 ResNet，这个模型有 38 亿 FLOPs。而对于更深的网络，比如 101 层和 152 层的 ResNet，只需要使用

更多的 3 层瓶颈块即可，其复杂性远低于 VGG16/19。

Block type	Stem	Stage 1		Stage 2		Stage 3		Stage 4		FC layers	ImageNet GFLOP	top-5 error
		layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	blocks			
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6
ResNet-50	Bottle	1	3	9	4	12	6	18	3	9	1	3.8
ResNet-101	Bottle	1	3	9	4	12	23	69	3	9	1	7.6
ResNet-152	Bottle	1	3	9	8	24	36	108	3	9	1	5.94

事实上，网络结构仍然可以被进一步改进，有一篇关于残差网络的后续论文将 Batch Norm Layer 和 ReLU 函数的顺序做了一些微调，这可以让其在 ImageNet 上的工作变得更好一点。



ResNet 除了在 ImageNet 竞赛中大放异彩，还在 CIFAR-10 和 MICRO-COCO 数据集中取得了斐然的效果，它可以被用于 ImageNet 检测，ImageNet 定位，COCO 检测和 COCO 分割，总之，这是一个非常强大的现代卷积神经网络。

当然，随着时代的发展，现在衍生出了更多更强大的现代卷积神经网络，比如稠密网络（DenseNet），MobileNet，EfficientNet 等等，感兴趣的读者可以自行探究。

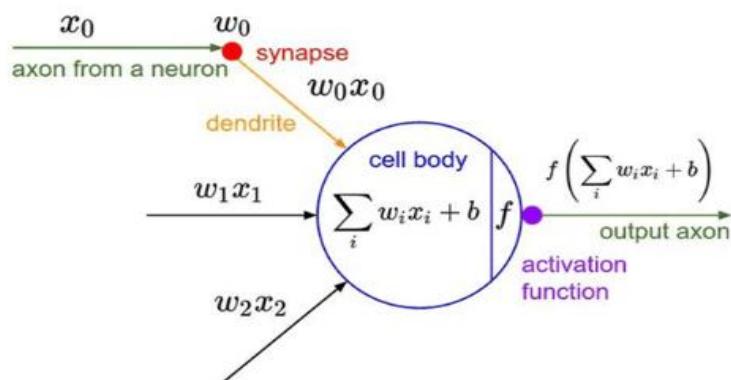
4 神经网络训练指南

在本章中，我们将对神经网络的训练进行进一步介绍。我们经常能听到有人将深度学习称为“炼丹”。事实上，对于深度学习的训练，我们可以采用一些小技巧（trick）来让我们的模型表现更好，我们可以将这些技巧按照训练的时间节点分类：训练前（Before Training），训练中（During Training）以及训练后（After Training），并沿着这条时间轴对一些常见的技巧进行介绍。

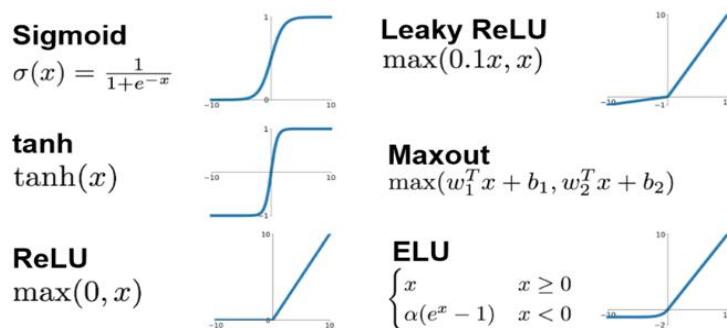
4.1 训练前——我们应该做哪些准备？

4.1.1 激活函数的选择

我们在前面几章介绍了全连接神经网络和卷积神经网络。我们知道，无论是全连接层还是卷积层，我们都需要讲输入数据与权重相乘后累加的结果送到一个非线性函数（non-linear function）中，我们通常将这个非线性函数称为激活函数（Activate Function）。



在深度学习中比较常见的激活函数如下图所示，我们将按照顺序分别进行介绍。



第一个函数是 sigmoid 函数，我们在第二章讲线性模型时就已经对它进行了一个初步的了解，我们了解了 sigmoid 函数的数学表达式以及其导函数的特点，在这里我们不再赘述。

我们还了解到，sigmoid 函数可以将输入压缩到一个 $[0, 1]$ 的区间作为输出。事实上，这一点在早期神经网络中是很重要的一点，因为它可以对神经元的激活频率给出一个很好的解释：激活频率介于完全不激活（0）与假定最大频率处的完全饱和（saturated）的激活（1）之间。

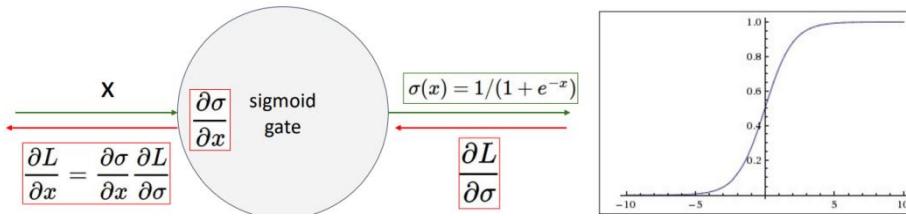
然而现在 Sigmoid 函数已经很少使用了，因为它有三个主要缺点：

首先，我们发现，Sigmoid 函数的梯度很有可能会消失，考虑 $\sigma(x) = 1/(1 + e^{-x})$ ，由数学知识我们可知：

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) \cdot (1 - \sigma(x))$$

$$\lim_{x \rightarrow -\infty} \sigma(x) = 0, \quad \lim_{x \rightarrow +\infty} \sigma(x) = 1$$

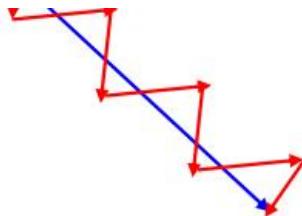
因此，当输入 x 过大或过小时，sigmoid 函数的梯度接近于 0，因此当我们在做反向传播时，局部梯度要与 sigmoid 函数的梯度相乘。因此，如果局部梯度非常小，那么相乘的结果也会接近零，这会“杀死”梯度，几乎就没有信号通过神经元传到权重再到数据了。



其次，Sigmoid 函数的输出不是以零为中心的（Zero-centered），这一问题主要影响的是梯度下降的运作。事实上，考虑一个多层神经网络：

$$h_i^{(l)} = \sum_j w_{i,j}^{(l)} \sigma(h_j^{(l-1)}) + b_i^{(l)}$$

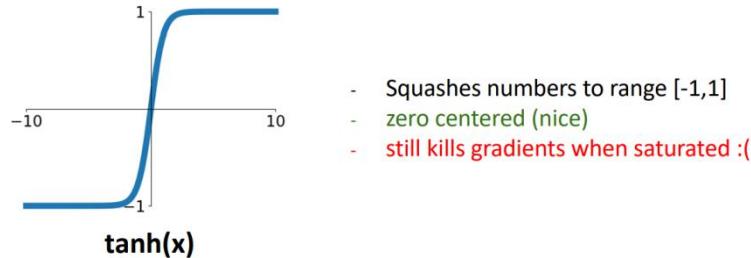
其中， $h_i^{(l)}$ 是这个神经网络第 l 层隐藏层中的第 i 个元素，我们发现，如果隐藏层的输出恒大于 0，那么关于 w 的梯度恒为正或者恒为负（符号根据反向传播初始回传梯度确定），也就是说，梯度下降时会朝着同一个方向改变，这将会导致梯度下降权重更新时出现 Z 字型的下降，如下图所示：



当然，这个问题并没有第一个问题严重，它是很容易被解决的，我们只需要将数据分成若干批，并将它们的梯度相加，就会出现不同的正负。此外，sigmoid 函数的第三个问题是： $\exp()$ 指数运算的代价比较高，综上所述，我们可以对 sigmoid 函数的问题总结如下：

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. $\exp()$ is a bit compute expensive

接下来，人们试图对 sigmoid 函数做一些改进，其中的一个变形就是 tanh 函数。我们从上面的图中不难发现，tanh 函数可以看成对一个长得像 sigmoid 的函数进行上下变换得到的函数，上下变换可以有效地使输出以零为中心，但与 sigmoid 函数一样，当 tanh 函数饱和时，同样会出现梯度消失的问题。

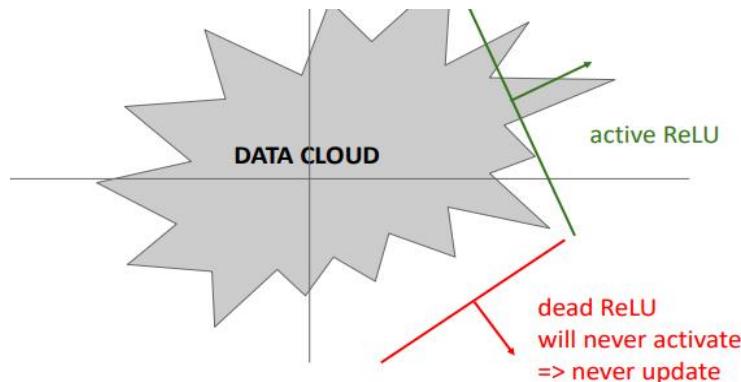


接下来，人们对激活函数进行进一步的改进，于是提出了 ReLU (Rectified Linear Unit) 激活函数，它的表达式如下：

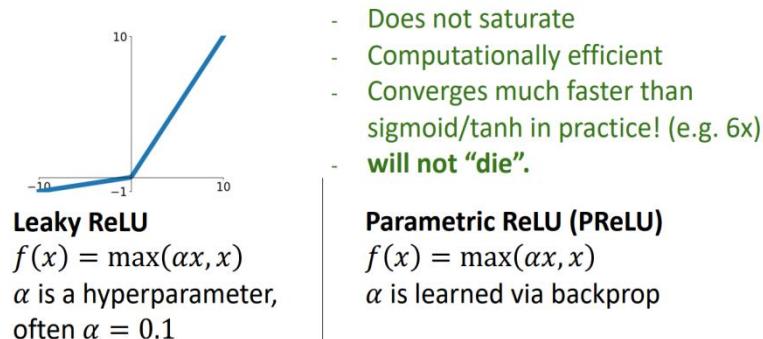
$$f(x) = \max(0, x)$$

换言之，这个激活函数是一个关于 0 的阈值，它相比于 sigmoid 函数和 tanh 函数的优点在于，首先，ReLU 只有 x 轴负半轴会饱和，并且它对于随机梯度下降的收敛有巨大的加速作用（Krizhevsky 等的论文提出有 6 倍之多），此外，ReLU 函数避免了指数运算等耗费计算资源的操作，只对一个矩阵进行阈值计算。

但是，ReLU 函数也有一定的缺点，首先它的输出并不以零为中心，其次，当 x 小于 0 时，函数值为 0，因此同样有可能出现梯度消失的问题，如下图所示：



对于这种问题，通常我们会通过在初始化 ReLU 时，设置一个小小的正数偏置（比如 0.01）进行避免，当然，我们还可以对 ReLU 函数进行改进，最常见的一种改进方法是使用一种名叫 Leaky ReLU 的函数，它通过设置一个超参数 α ，通常设置为 0.1，从而保证当 x 小于 0 时，梯度不会直接变为 0。有些研究表明这个激活函数表现不错，但其效果不是很稳定，于是何恺明等人在 2015 年发布的论文 Delving Deep into Rectifiers 中介绍了一种新的方法 PReLU，其表达式与 Leaky ReLU 相同，但其中的超参数 α 时通过反向传播不断学习的，如下图所示：

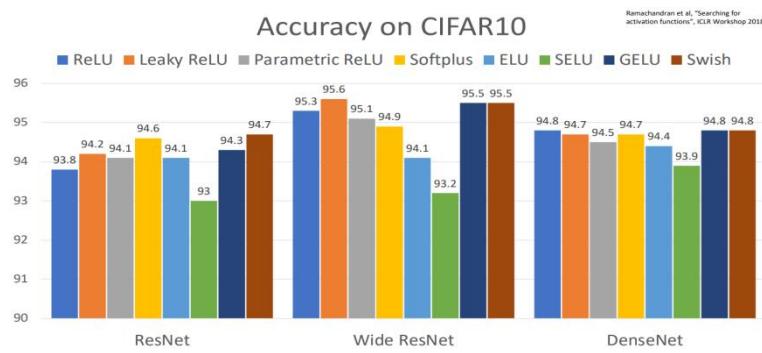


当然，后来人们也提出了一些其他类型的单元，它们对于权重和数据的内积结果不再使用线性函数的形式，一个比较流行的选择时 Maxout，它其实是对 ReLU 和 leaky ReLU 的一般化归纳，其表达式如下：

$$f(x) = \max(w_1^T x + b_1, w_2^T x + b_2)$$

ReLU 和 Leaky ReLU 都是这个公式的特殊情况，因为 w 和 b 的取值都是任意的，这样 Maxout 神经元就拥有了 ReLU 的所有优点，而同时也避免了 ReLU 的死亡单元。然而它的参数数量是 ReLU 的两倍，可能会导致更大的计算量。

此外，人们针对 ReLU 还提出了一些其他的变种，比如 ELU，SELU，GELU，它们的数学表达式和梯度求解就显得复杂许多。而这些复杂的激活函数可能会用在更复杂的模型中，比如 GELU 函数通常就作为 BERT，GPT 等大模型（后续章节我们会进行介绍）的激活函数。

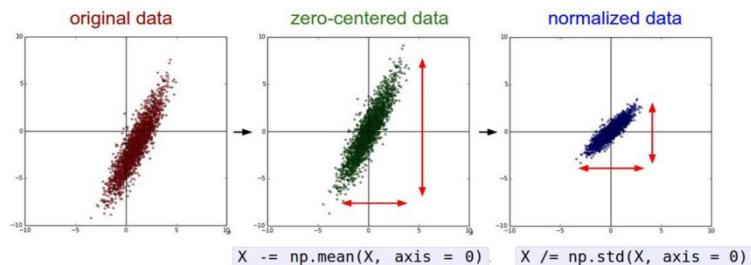


那么，我们在实做中该选择哪种激活函数呢？其实我们不用想得太复杂，只需要使用 ReLU 即可，上图是各种激活函数作用在不同的神经网络上的结果，我们采用 CIFAR10 数据集进行测试，我们发现 ReLU 函数已经做得足够好了，但如果你想要追求那 0.1% 的准确率的话，可以尝试用 Leaky ReLU，而对于 ELU，SELU，GELU，其实并不建议使用，因为它们的计算量比 ReLU 和 Leaky ReLU 要大得多。Anyway, do not use sigmoid or tanh!!!

4.1.2 数据预处理

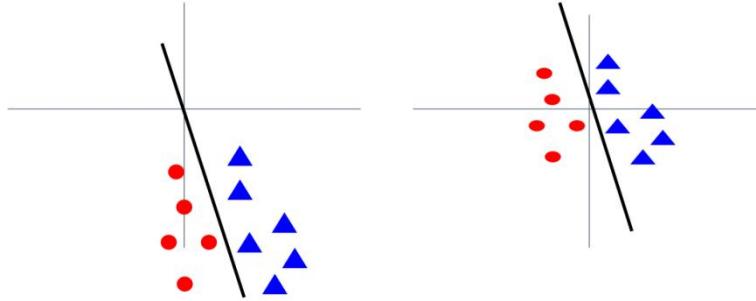
我们在训练神经网络之前，最开始需要对数据进行预处理。我们在上一小节讨论激活函数中发现，如果输出不以零为中心，可能会引发梯度下降的一些小问题。为了避免一些问题，我们可以对数据进行预处理（Data Preprocessing）。

第一种数据预处理的方法是减均值，这样做的好处是通过对数据中每个独立特征减去平均值，使得数据的中心在每个维度上都被迁移到了原点。接下来，我们可以对数据进行归一化，即在每个维度都除以其标准差，得到的数据如下图所示：



那我们为什么需要做归一化呢？假设我们在做一个分类问题，如下图所示，在归一化之

前，分类的损失函数值会对权重矩阵的变化很敏感，这对优化问题的求解会产生一些问题。而在归一化处理之后，损失函数对 w 的微小改变不那么敏感，更容易优化。

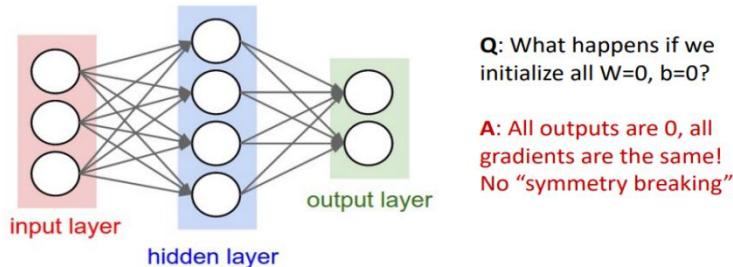


事实上，对于经典的神经网络，通常会采用一些数据处理，例如在 AlexNet 中，采用减去平均图像的方法，VGGNet 中使用的是每个通道减去各通道的均值，ResNet 中使用的是每个通道减去各个通道的均值，并除以各通道标准差。

4.1.3 权重初始化

我们在之前梯度下降中了解到，对于初始的 w ，是可以人为决定的。事实上，我们把这一操作称为权重初始化（weight initialization），初始化网络参数是训练神经网络里非常重要的一步，其方法也是不同的。

对于这一问题，人们很自然地想到将权重初始化为 0，这种方法被称为全零初始化，但实际上这种初始化的方法是不好的：因为网络中的每个神经元都计算出同样的输出，然后它们就会在反向传播中计算出同样的梯度，从而进行同样的参数更新。换句话说，如果权重被初始化为同样的值，神经元之间就失去了不对称性，从而只能学到同一个特征。



事实上，权重初始值要非常接近 0 又不能等于 0，解决方法就是将权重初始化为很小的数值，以此来打破对称性。其思路是：如果神经元刚开始的时候是随机且不相等的，那么它们将计算出不同的更新，并将自身变成整个网络的不同部分。

这种做法对于浅层神经网络是可行的，但随着网络层次的增加，在反向传播中，我们会得到非常小的梯度，这可能会最终引起梯度消失，如下图所示：

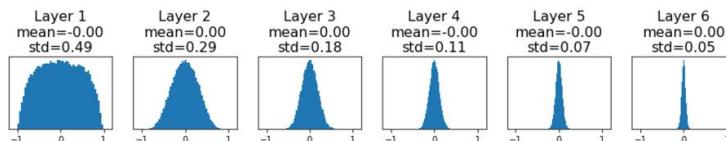
```

dims = [4096] * 7      Forward pass for a 6-layer
hs = []                  net with hidden size 4096
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)

All activations tend to zero for
deeper network layers
Q: What do the gradients
dL/dW look like?  

A: All zero, no learning =

```



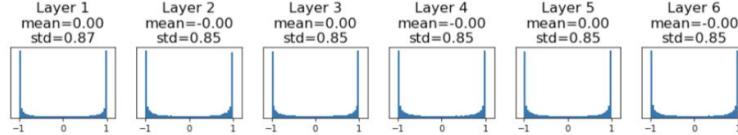
我们试图对上述问题进行改变，于是我们想到了将初始化的权重的标准差从原先的 0.01 增加到 0.05，并同样使用正态分布进行初始化。我们发现，尽管效果与使用 0.01 相反，但同样会导致梯度趋近于 0，这样也无法运作梯度下降了。

```

dims = [4096] * 7      Increase std of initial weights
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)

```

All activations saturate
Q: What do the gradients look like?
A: Local gradients all zero, no learning =(



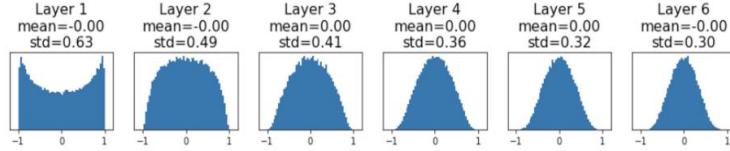
综上所述，初始化的幅度既不能太小，也不能太大，最理想的情况是可以让初始化实现“自我调节”，“Xavier”初始化就是一个很好地办法，它的思想是“根据输入维度设置幅度值，并通过将输入维度的平方根放在分母上实现自适应调整。

```

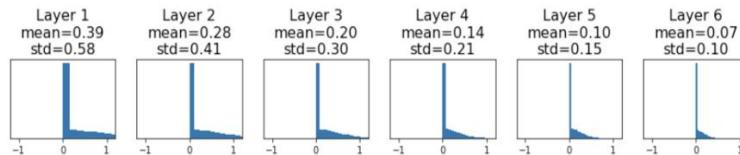
dims = [4096] * 7          "Xavier" initialization:
hs = []                      std = 1/sqrt(Din)
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)

```

"Just right": Activations are nicely scaled for all layers!
For conv layers, Din is kernel_size² * input_channels



但这个方法仍然有一些问题，因为我们在之前讲过，实做中我们应该选择 ReLU 作为激活函数，而不是 tanh，而我们在将 ReLU 作用到上述初始化过程中，会发现我们的激活函数的输出值再次倒向了 0，从而导致无法学习。



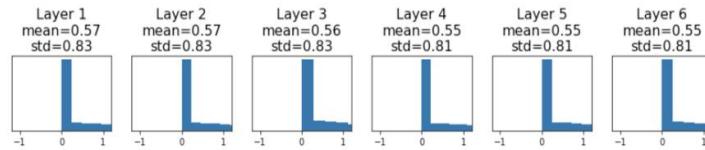
于是，何恺明提出了一种初始化方法，名叫 Kaiming 初始化(Kaiming/MSRA Initialization)，这种初始化方法只需要将标准差乘一个根号 2 的系数即可。

```

dims = [4096] * 7  ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)

```

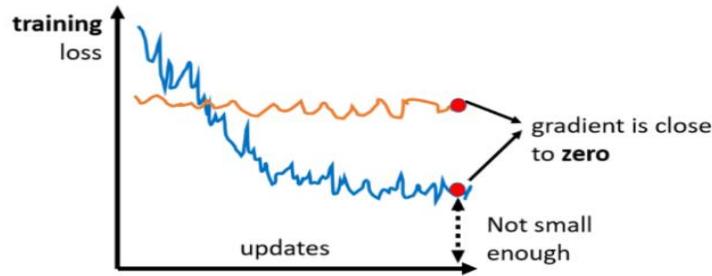
"Just right" – activations nicely scaled for all layers



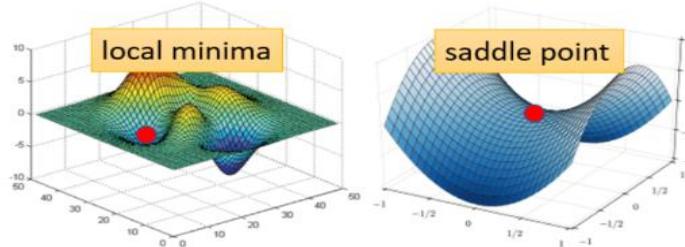
4.2 训练中——梯度下降如何做得更好?

4.2.1 局部最小值 (local minima) v.s 鞍点 (saddle point)

当在做优化问题时，随着参数的更新，损失函数确实在不断下降，但有时候它们的 loss 与一些更窄的网络甚至是线性模型相比，并没有发挥出更深的网络的力量。也有可能发现从一开始不管如何更新参数，loss 都掉不下去。这是为什么呢？



实际上，这是因为梯度已经接近 0 了，有人会说优化失效是因为卡在了局部最小值，但实际上，除了局部最小值，在鞍点 (saddle point) 处的梯度也是 0。而局部最小值点和鞍点合称为临界点 (critical point)。因此，我们只能说，当优化失效时，梯度卡在了临界点。



当然，我们还是有必要知道梯度是卡在局部最小值点还是卡在鞍点的，这是因为如果是卡在局部最小值点，那可能就没有路可以走了，而卡在鞍点时，只要我们能够用一些方法“逃离”，那还是有办法能让 loss 继续下降的。

那么如何判断梯度是卡在局部最小值点还是鞍点呢？实际上，对于一个损失函数，考虑它的泰勒展开式，这边我们不需要展开太多，到二阶即可：

$$L(\theta) \approx L(\theta') + (\theta - \theta')^T g + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$$

其中， g 是一个向量， H 是一个矩阵，满足如下公式：

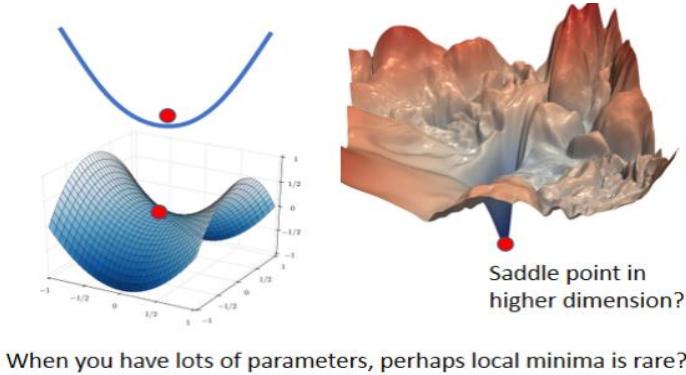
$$g = \nabla L(\theta'), g_i = \frac{\partial L(\theta')}{\partial \theta_i}$$

$$H_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} L(\theta')$$

当梯度走到临界点时，这意味着一阶导数 g 已经变为 0，我们只需要考虑 Hessian 矩阵即可。而我们知道，对于任意向量 v ，若均有 $v^T H v > (<) 0$ ，则称 H 为正（负）定的，判断方法是计算矩阵 H 的特征值是否全部为正（负），而对于这种情况，可以推出： $L(\theta) > (<) L(\theta')$

对于附近均成立，则可以知道梯度卡在了局部最小值点，而如果特征值有正有负，则说明卡在了鞍点。

当然实际上，我们不会这样计算，因为一个神经网络有很多参数，计算 Hessian 矩阵的特征值也是不现实的。但是我们也不需要担心，因为当我们把视角放到更高的维度的话，我们其实可以发现原先你所认为的局部最小值点，其实也是鞍点。

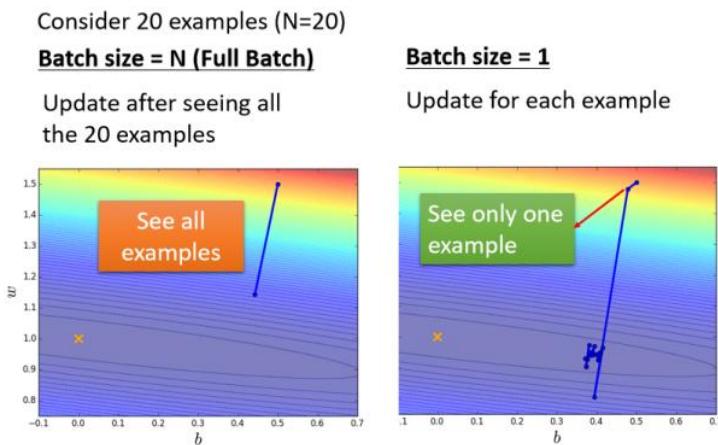


当然这一猜想是有人做过实验进行验证的，实验表明，局部最小值并没有那么常见，多数的时候，你觉得你训练到一个地方，梯度真的已经很小了，参数也不再更新了，大概率还是因为梯度卡在了一个鞍点的位置。

4.2.2 批次 (Batch) v.s. 动量 (Momentum)

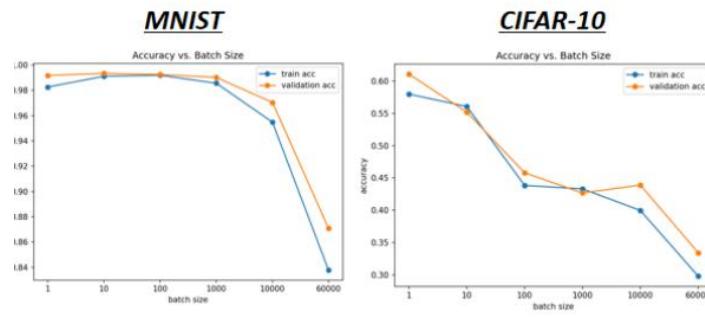
在梯度下降算法中，由于数据量大部分时候都很大，我们并不可能对所有的数据进行微分计算，处理的方法是把所有的数据分成一个一个的批次 (Batch)，并在每一次迭代中随机抽取一个批次做梯度下降。

那么这样的做法对我们的训练有什么帮助呢？考虑一个有 20 笔训练资料的样例，我么分别采用 full-batch 和 mini-batch 的方法进行训练，得到的结论是：full-batch 的冷却时间很长但是十分强大，而 mini-batch 的冷却时间很短，但会产生一些噪声。



但实际上，对于现在的科技而言，批次设的很大的参数更新时间也未必会长，这就是并行计算 (parallel computing) 的威力所在。当然，我们还是不能把批次大小设置地大得夸张，因为小批次的噪声其实是对我们的训练和测试有帮助的。换言之，随着批次大小 (batch

size) 的增加，训练结果和验证结果都会变差。

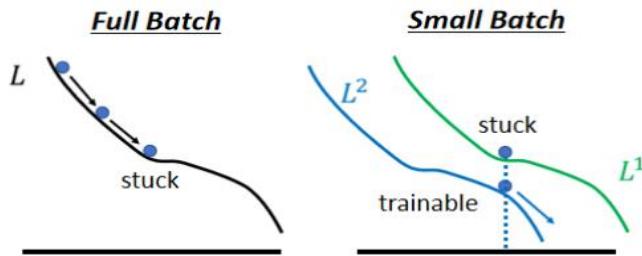


- Smaller batch size has better performance
- What's wrong with large batch size? Optimization Issue

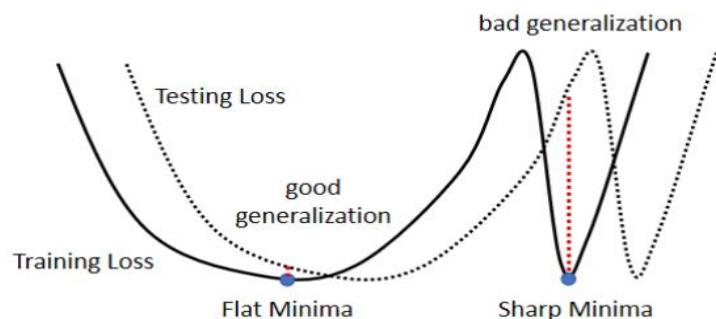
那么为什么出现噪声是一件不那么糟糕的事情呢？这可能体现在以下两个方面：

首先，”noisy update”对于训练是更有帮助的，这是因为如果我们采用 full-batch 做梯度下降的话，走到一个临界点（不管是 local minima 还是 saddle point）就会停下来，而对于小批次寻来你来说，损失函数的个数和批次的数目是一样的，因此当一个损失函数的参数不再更新时，我们可以选择跳出来换一个 batch 值或者换一个函数。

- Smaller batch size has better performance
- “Noisy” update is better for training



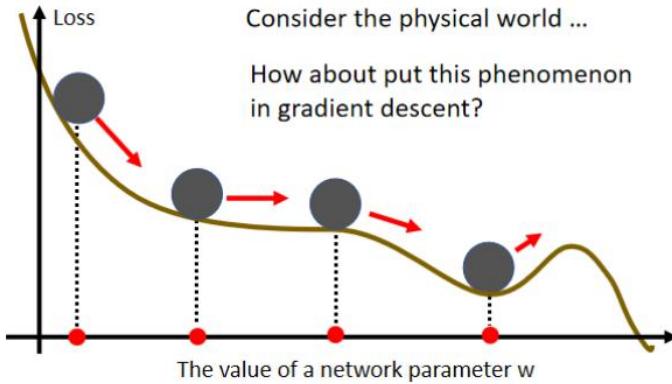
其次，”noisy update”能让模型泛化（model generalization）变得更好，事实上，在梯度下降寻找局部最小值时，局部最小值点也有宽窄之分。这完全不难理解，因为地势也有平缓之分，在 flat minima 处的 testing loss 与 training loss 是很接近的，而在 sharp minima 处，差距会很夸张。因此，拿整批数据做梯度下降无法得知寻找到的局部最小值点是 flat 还是 sharp 的，而小批次训练就能比较好地解决这一问题。



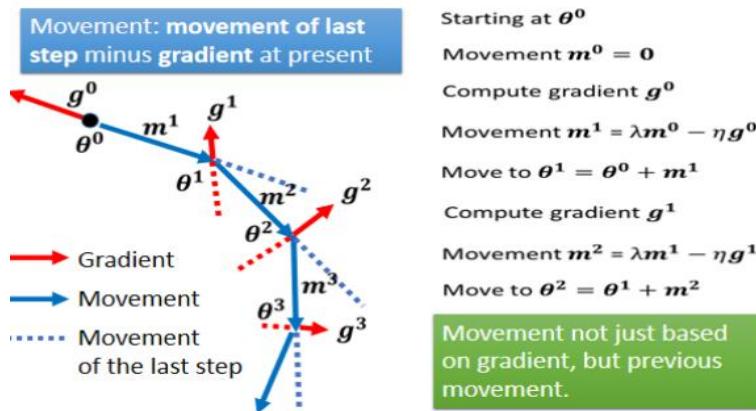
因此，批次大小（batch size）成为了深度学习中又一个重要的超参数。

当然，除了批次大小的设置，还有另一种可以对抗“saddle point or local minima”的技

术，叫做动量（momentum），形象地理解，就相当于我们人为给正在滚下坡的小球一个外力（动量）。



那么我们就很容易理解，动量经常是用在梯度下降里面，于是有人就提出了一个（Vanila）梯度下降+动量的算法，流程如下：

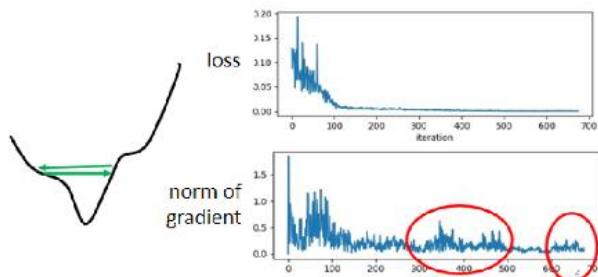


不难发现，当我们的梯度已经更新到了一个临界点时，加入的动量可能可以帮助它跳出去，这或许能帮助我们找到更好的局部最小值。

4.2.3 自动调整学习率 (Adaptive Learning Rate)

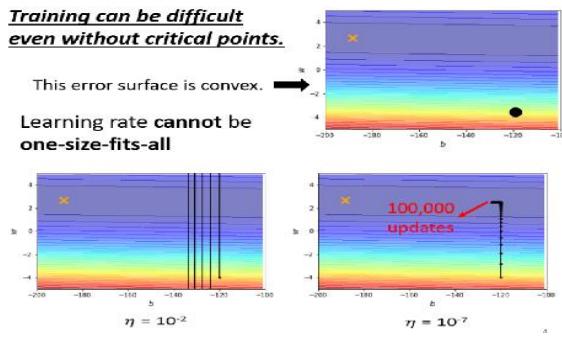
在之前，我们介绍了临界点对于训练的影响，但即便不在临界点，训练依旧困难。

事实上，当我们走到临界点时，意味着梯度已经走到了 0，所以我们的训练会“卡住”。但是你们有没有真的确认过，当你的 loss 不再下降的时候，你的梯度真的很小吗？在下面这个例子中，我们发现，当 loss 不再下降时，你的梯度并没有变得很小。

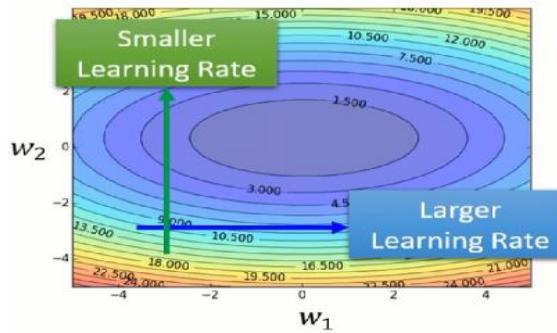


这样的结果其实也不难想到，也许你遇到的情况是，你的梯度由于走到了一个“山谷”

而来回震荡，这个时候，loss 也是无法下降的。



事实上，在实际问题中，如果你想用最原始的梯度下降算法去逼近临界点，几乎是不可能做到的。因为往往在梯度很大的时候，loss 就掉了下去。这是为什么呢？原因其实很简单：假如我们使用的学习率是固定的，那在梯度下降的过程中，有时会步幅过大，有时会步幅过小。这是因为我们的学习率显然无法适配每一个参数。



那么我们就需要定制化我们的学习率，最基本的自动调整的思想是这样的，考虑我们最原始的梯度下降公式：

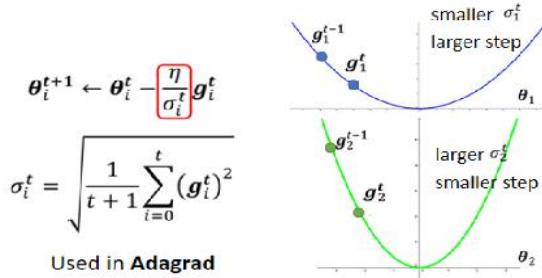
$$\theta_i^{t+1} \leftarrow \theta_i^t - \eta g_i^t$$

$$g_i^t = \frac{\partial L}{\partial \theta_i} |_{\theta=\theta^t}$$

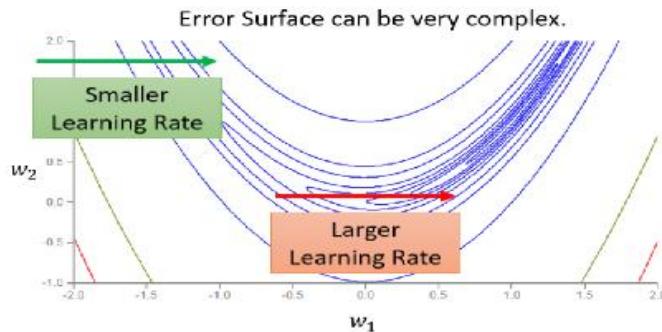
在参数更新的公式中，我们将学习率 η 改成 $\frac{\eta}{\sigma_i^t}$ 即可，这其实就是均方根(root mean square)的思想，算法如下所示：

$$\begin{aligned}
 & \text{Root Mean Square} \quad \theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t \\
 & \theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0 \quad \sigma_i^0 = \sqrt{(g_i^0)^2} = |g_i^0| \\
 & \theta_i^2 \leftarrow \theta_i^1 - \frac{\eta}{\sigma_i^1} g_i^1 \quad \sigma_i^1 = \sqrt{\frac{1}{2}[(g_i^0)^2 + (g_i^1)^2]} \\
 & \theta_i^3 \leftarrow \theta_i^2 - \frac{\eta}{\sigma_i^2} g_i^2 \quad \sigma_i^2 = \sqrt{\frac{1}{3}[(g_i^0)^2 + (g_i^1)^2 + (g_i^2)^2]} \\
 & \vdots \\
 & \theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t \quad \sigma_i^t = \sqrt{\frac{1}{t+1} \sum_{i=0}^t (g_i^i)^2}
 \end{aligned}$$

这一招在实做中被用在一个叫 AdaGrad 的方法中，而这一招可以做到当坡度比较大时，学习率就减小，当坡度比较小时，学习率就增大。事实上，当我们有了 σ 这一项，就可以随着梯度的不同，来实现调节我们的学习率。



当然，AdaGrad 仍然有一些问题，因为即便是同一个参数，它需要的学习率也会随着时间而改变。以下面这个误差表面为例：



那么我们期待找到这样的一个自适应学习率（adaptive learning rate）的方法，能够实现即便是同一个参数同一个方向，学习率也是可以动态调整的，于是有一个新招叫做 RMSProp。

RMSProp 其实是对 AdaGrad 的一个改进，因为它沿用了之前的 root mean square，只是在 AdaGrad 中，每一个梯度都具有同等的重要性，而在 RMSProp 中，我们需要用一个参数： α 来决定 root mean square 和梯度的权重，算法如下：

RMSProp	$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$
$\theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0$	$\sigma_i^0 = \sqrt{(g_i^0)^2}$
$\theta_i^2 \leftarrow \theta_i^1 - \frac{\eta}{\sigma_i^1} g_i^1$	$0 < \alpha < 1$
$\theta_i^3 \leftarrow \theta_i^2 - \frac{\eta}{\sigma_i^2} g_i^2$	$\sigma_i^1 = \sqrt{\alpha(\sigma_i^0)^2 + (1-\alpha)(g_i^1)^2}$
⋮	
$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$	$\sigma_i^t = \sqrt{\alpha(\sigma_i^{t-1})^2 + (1-\alpha)(g_i^t)^2}$

在今天我们常用的一个优化器，叫做 Adam 优化器，其实它就是由 RMSProp+Momentum 结合而成的一个优化器。其使用动量作为参数更新方向，并且能够自适应调整学习率。

Adam: RMSProp + Momentum

```

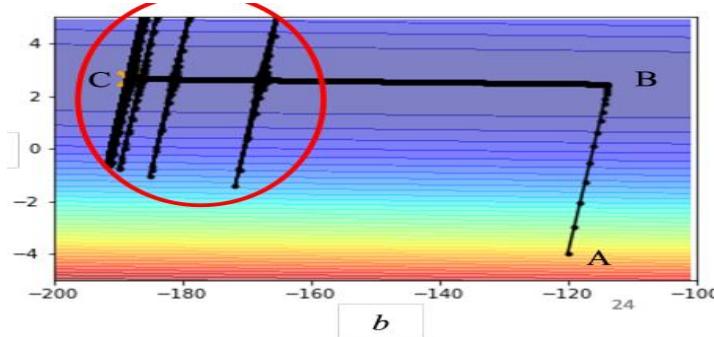
Algorithm 1: Adam, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation,  $g_t^2$  indicates the elementwise square  $g_t \odot g_t$ . Good default settings for the tested machine learning problems are  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\epsilon = 10^{-8}$ . All operations on vectors are element-wise. With  $\beta_1^t$  and  $\beta_2^t$  we denote  $\beta_1$  and  $\beta_2$  to the power  $t$ .
Require:  $\alpha$ : Stepsize
Require:  $\beta_1, \beta_2 \in [0, 1]$ : Exponential decay rates for the moment estimates
Require:  $f(\theta)$ : Stochastic objective function with parameters  $\theta$ 
Require:  $\theta_0$ : 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  $\theta_t$  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  $\theta_t$  (Resulting parameters)

```

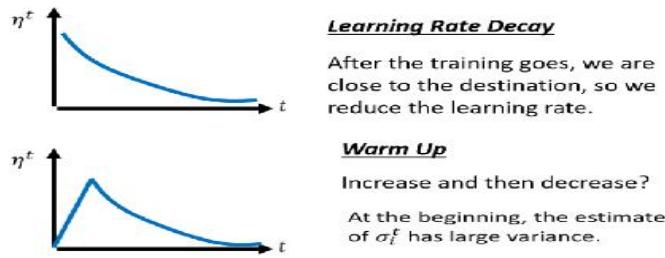
那对于 Adam 优化器的使用，其实也不需要太担心。PyTorch 里面已经写好了 Adam 优化器，这个优化器里面有一些超参数需要人为决定，但是往往用 PyTorch 预设的参数就足够好了。

那么我们来看一下，当我们用了自适应学习率方法之后，我们的误差表面会变成什么样子。在这边我们以最简单的 AdaGrad 为例。

我们发现，一开始优化很顺利，而在左转的时候，有了 AdaGrad，梯度并不会卡住，可以继续走到接近终点的位置，但是走到终点附近突然“爆炸”了。这是因为在转弯过后，梯度变得很小，因此我们累积了很多很小的 σ ，于是在 AdaGrad 的公式中，梯度就会突然爆炸一下。



通过学习率调度（learning rate scheduling）可以解决这个问题。在之前的学习率调整方法中， η 是一个固定的值。而在学习率调度中， η 与时间有关。最简单的一个方法叫学习率衰减（Learning Rate Decay）：随着参数不断更新， η 越来越小。



Please refer to RAdam <https://arxiv.org/abs/1908.03265>

除了学习率下降以外，还有另外一个经典的学习率调度的方式——预热（warmup）。预热的方法是让学习率先变大后变小，至于变到多大、变大的速度、变小的速度是超参数。

残差网络里面有预热的，在残差网络里面，学习率先设置成 0.01，再设置成 0.1，并且其论文还特别说明，一开始用 0.1 反而训练不好。除了残差网络，BERT 和 Transformer 的训练也都使用了预热。

We further explore $n = 18$ that leads to a 110-layer ResNet. In this case, we find that the initial learning rate of 0.1 is slightly too large to start converging⁵. So we use 0.01 to warm up the training until the training error is below 80% (about 400 iterations), and then go back to 0.1 and continue training. The rest of the learning schedule is as done previously. This 110-layer network converges well (Fig. 6, middle). It has fewer parameters than other deep and thin

⁵With an initial learning rate of 0.1, it starts converging (<90% error) after several epochs, but still reaches similar accuracy.

Residual Network

<https://arxiv.org/abs/1512.03385>

5.3 Optimizer

We used the Adam optimizer [17] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula:

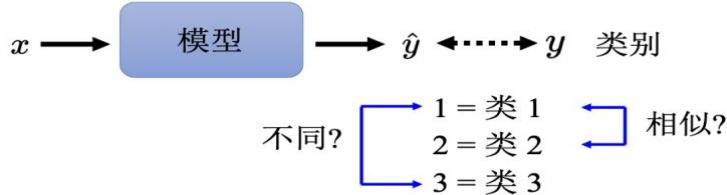
$$lrate = d_{\text{model}}^{-0.5} \cdot \min(\text{step_num}^{-0.5}, \text{step_num} \cdot \text{warmup_steps}^{-1.5}) \quad (3)$$

This corresponds to increasing the learning rate linearly for the first `warmup_steps` training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used `warmup_steps` = 4000.

Transformer <https://arxiv.org/abs/1706.03762>

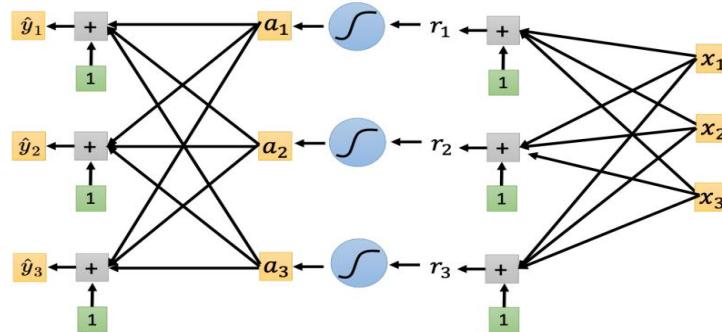
4.2.4 损失函数也可能有影响

我们在这里简单复习一下之前讲过的分类问题。我们在第 2 章曾说过，分类也不是不可以当作回归问题来看。输入 x 之后，得到的输出 \hat{y} 与实际的类编号 y 进行比较即可。



我们当时简略地说明了一下这一做法可能出现的问题，事实上，我们将分类当作回归问题时，不同的类在某种情况下可能会比较相近，但假设在上述分类问题中，三个类之间本身并没有特定的关系时，我们就需要采用独热编码（one-hot vector）来表示每一个类，比如上面的三类分别为： $[1, 0, 0]^T$, $[0, 1, 0]^T$, $[0, 0, 1]^T$ 。

独热向量的一个好处在于，如果用独热向量计算距离的话，类两两之间的距离都是一样的。那么如果目标 y 是一个有三个元素的向量，那对于我们的神经网络，也需要输出三个数字才行，那其实只需要将原先的步骤重复三次即可：



按照上面的流程，我们得到一个与回归问题类似的函数：

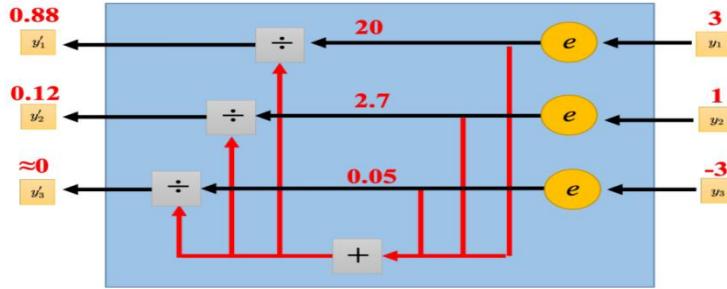
$$\mathbf{y} = \mathbf{b} + \mathbf{W} \cdot \sigma(\mathbf{b} + \mathbf{W} \cdot \mathbf{x})$$

需要注意的是，这里的 y 是一个向量，但实际上在分类问题中，往往会将 y 通过一个

softmax 函数得到一个 \hat{y}' ，而实际计算的距离也是 \hat{y}' 与 \hat{y} 之间的距离。

$$\text{label } \hat{y} \longleftrightarrow \hat{y}' = \text{softmax}(y)$$

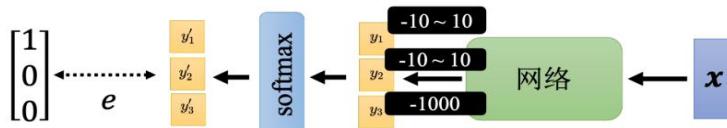
那么为什么需要加入一个 softmax 呢？实际上，softmax 有一个归一化的过程，它将多个神经元的输出，映射到 $(0,1)$ 区间内，可以看成概率来理解，从而来进行多分类。



在得到 \hat{y}' 之后，我们应该如何计算距离呢？在回归问题中，我们常用均方误差来计算，但是在分类问题中，选择交叉熵作为损失函数更加常用，其表达式如下：

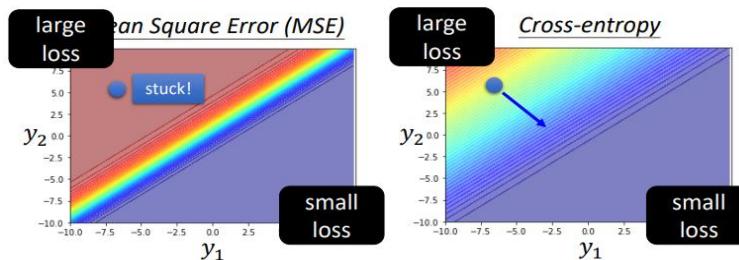
$$L = - \sum_i \hat{y}_i \ln y'_i$$

交叉熵的式子其实是从极大似然估计（maximum-likelihood）推出来的。接下来从优化的角度来说明相较于均方误差，交叉熵是被更常用在分类上。假设我们需要做一个 3 分类问题，并且假设正确答案就是 $[1, 0, 0]^T$



假设 y_3 被设置成了 -1000 ，那么经过 softmax 之后，其结果很接近 0，因此我们只需要看其他两个类别的变化即可，假设 y_1, y_2 的变化都是介于 -10 与 10 之间，考虑均方误差和交叉熵的误差表面。

当 y_1 大，而 y_2 小时，不管是均方误差还是交叉熵都会产生很小的 loss 值，反之 loss 会比较大，因此我们需要期待训练走到右下角。那假设我们从左上角开始走，均方误差的梯度在 loss 很大的时候就会卡住，因为它的梯度很平坦，因此如果采用均方误差作为损失函数来训练分类问题时，会有很大概率训练不起来。



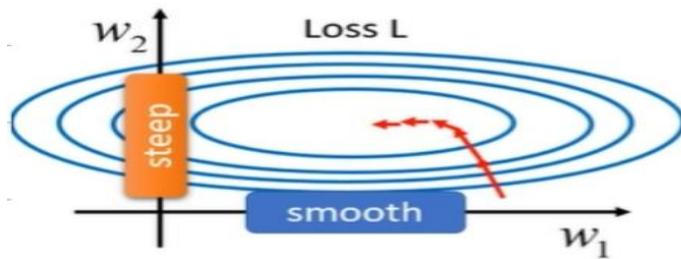
当然，上面的讨论仅仅是讨论了损失函数对于分类的影响，但假设你今天有足够的优

化器（比如 Adam），那采用均方误差也是有训练起来的可能性的。当然，在今天，人们肯定不再会选择用 MSE 作为分类问题的损失函数，因为交叉熵损失函数肉眼可见得更好。

4.2.5 批次标准化 (Batch Normalization)

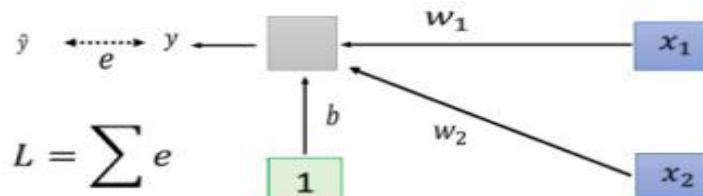
我们之前讲到，如果误差表面很崎岖，那就比较难训练。那么有没有一种办法，可以直接将其“铲平”，让它更容易训练呢？实际上，批次标准化 (Batch Normalization) 就是一个类似的想法。

千万不要小看这样的一个优化的问题！有时候就算误差表面是凸 (convex) 的，它就是一个碗的形状，都不一定很好训练。如下图所示，假设两个参数对损失的斜率差别非常大，如果是固定的学习率，是很难得到好的结果的。

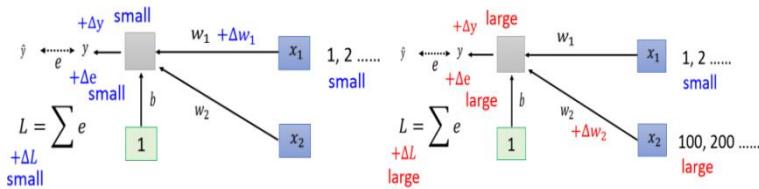


如果这个问题出现在前面的话，我们会采用自适应学习率的方法进行处理。在这里我们从另外一个角度出发，如果我们能将这个难做的误差表面改掉，看看能不能变得好做一点。那么，在做这件事之前，我们需要考虑 w_1 和 w_2 它们斜率差得很多的情况是怎么来的呢？

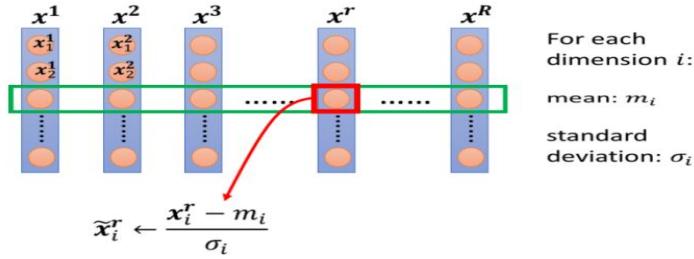
下图是一个非常简单的模型，输入是 x_1 和 x_2 ，其对应的权重分别是 w_1 和 w_2 ，这是一个简单的线性模型。



那假设 w_1 有一个变化，我们令其为 Δw_1 ，当 $w_1 \rightarrow w_1 + \Delta w_1$ 时， y 和 L 均会产生一些变化，而这个变化的大小与 x_1 的大小是直接相关的。换言之，假如我们的输入： x_1 很小时， w_1 对应的斜率就会很平缓。反之，对于 x_2 和 w_2 同理。

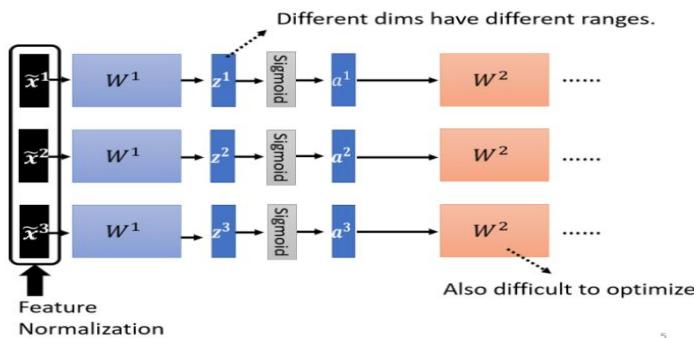


从上面的分析我们知道，可以通过给输入的值限定一个范围，从而能够对它们的斜率进行控制。这种将不同的维度限制在同一个范围的一类方法，叫做特征归一化 (Feature Normalization)，其方法如下图所示：



特征归一化实际上就是一个标准化的过程，这个做法的好处在于，让所有维度的均值和方差分别为 0 和 1，从而制造出一个更好的误差曲面，同时，在这样一个更好的误差曲面下做梯度下降，Loss 收敛的速度会更快，梯度下降也会变得更加顺利。

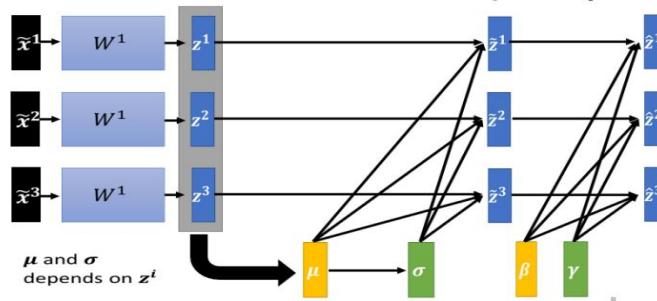
接下来我们考虑更深层次的网络（深度学习），我们将 \tilde{x} 记为归一化的特征，把它丢到深度网络里面，去做接下来的计算和训练。 \tilde{x}_1 通过第一层得到 z^1 ，依次类推。现在的一个问题是，我们的 x 虽然已经做了归一化，但经过与 W 相乘之后得到的 z ，并没有进行归一化，那么这样就会在对第二层参数训练产生困难！



我们当然可以仿照对于输入的方法类似进行归一化，但对于深度神经网络来说，这样的迭代计算是比较繁琐的。于是，就有了另外一个解决方法，即考虑某一批次，并对这一批数据进行归一化，这个方法叫做批次归一化（Batch Normalization），公式如下：

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

其中， β 和 γ 可以想成是网络的参数，需要另外再被学习出来，在 PyTorch 中，我们可以将 γ 设置成 `torch.ones`，而将 β 设置成 `torch.zeros`。

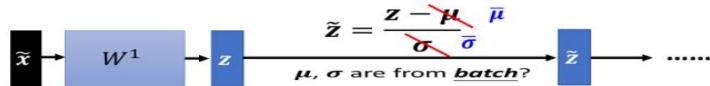


所以让网络在一开始训练的时候，每一个维度的分布，是比较接近的，也许训练到后来，已经训练够长的一段时间，已经找到一个比较好的误差表面，走到一个比较好的地方以后，再把 β 和 γ 慢慢地加进去，所以批量归一化，往往对训练是有帮助的。

批量归一化在测试的时候会有什么样的问题呢？在测试的时候，我们一次会把所有的测试的数据给你，所以你确实也可以在测试的数据上面，制造一个一个批量。但是假设系统上

线，做一个真正的线上的应用，比如批量大小设 64，我一定要等 64 笔数据都进来，才做一次做运算，这显然是不行的。

那我们该如何计算 μ 和 σ 呢？实际上，PyTorch 已经有一种特殊的处理方法，叫做移动平均（moving average），计算方法如下：



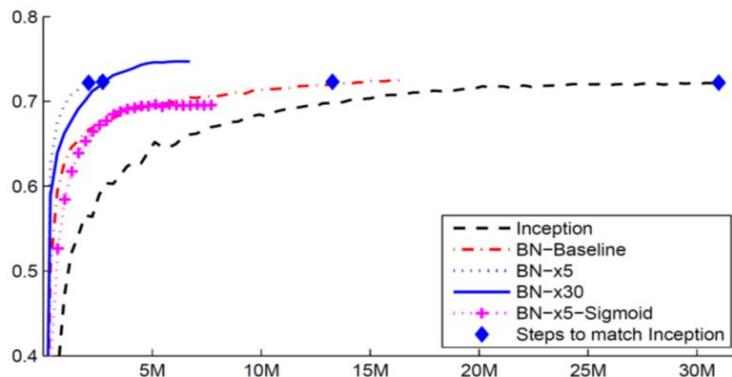
We do not always have **batch** at testing stage.

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

$$\begin{aligned} \mu^1 & \quad \mu^2 & \quad \mu^3 & \quad \dots & \quad \mu^t \\ \bar{\mu} \leftarrow p\bar{\mu} + (1-p)\mu^t \end{aligned}$$

在 PyTorch 中，上面的 p 也是一个超参，通常被设置成 0.1，这就是批量归一化在测试的时候的运作方式。

接下来我们来看一个文献中的结果，它反映了是否做批量归一化对于不同网络的训练准确率的影响。我们发现，所有的线在最后会收敛于一个差不多水准的准确率，但是红色的这条线收敛的速度更快，而粉色的这条线同样也使用了批量归一化，但它用到的是 Sigmoid 函数，它的训练是比较困难的。因此，我们一般选择 ReLU。



为什么批量归一化会比较好呢，那在这篇“How Does Batch Normalization Help Optimization?”这篇论文里面，他从实验和理论上，至少支持批量归一化可以改变误差表面，让误差表面比较不崎岖这个观点。事实上，如果要让网络误差表面变得比较不崎岖，其实不一定做批量归一化，还有很多其他的方法都可以让误差表面变得不崎岖，这篇论文就试了一些其他的方法，发现跟批量归一化表现也差不多，甚至还稍微好一点，这篇论文的作者也觉得批量归一化是一种偶然的发现，但无论如何，其是一个有用的方法。其实批量归一化不是唯一的归一化，还有很多归一化方法，比如层归一化（Layer Normalization, [\[1607.06450\] Layer Normalization \(arxiv.org\)](#)），组归一化（Group Normalization, [\[1803.08494\] Group Normalization \(arxiv.org\)](#)），权重归一化（Weight Normalization, [\[1602.07868\] Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks \(arxiv.org\)](#)）等，感兴趣的读者可以自行阅读相关论文。

4.3 训练后——如何让训练精益求精？

在完成训练之后，你可能希望自己训练的模型表现更进一步，事实上，我们在之前也讲过一些办法，比如激活函数选择 ReLU，优化器选择 Adam 等，但如果你在做了这些之后仍然感觉不是很满意的话，或许以下方法可以帮到你。

4.3.1 正则化 (Regularization)

我们在第二章介绍线性模型时，曾讲过一种 L2 正则化的方法来避免过拟合，它其实是在损失函数中将权重向量的某个范数，比如 $\|w^2\|$ (L2 范数) 作为惩罚项添加到最小化损失中。事实上，这只是正则化的一种参数化形式，需要我们对损失（目标）函数的形式进行改变。事实上，还存在着一些其他的正则化，我们将在本章中依次进行介绍。

首先，我们来对损失函数的正则化进行进一步讨论。事实上，除了在损失函数中引入 L2 正则化，还有一些其他的方法，比如 L1 正则化，以及 L1 和 L2 相结合的正则化（又被称为 Elastic net），当然，通常我们使用 L2 正则化即可。

In common use:

$$\text{L2 regularization} \quad R(W) = \sum_k \sum_l W_{k,l}^2 \quad (\text{Weight decay})$$

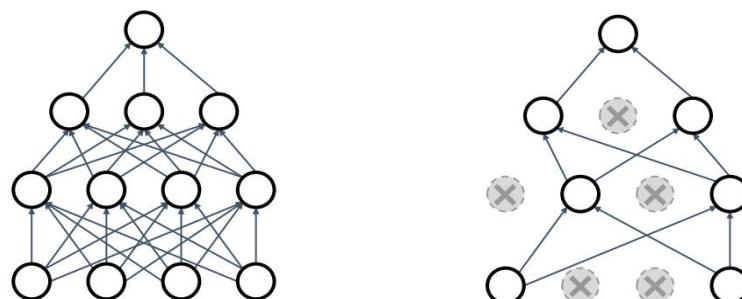
$$\text{L1 regularization} \quad R(W) = \sum_k \sum_l |W_{k,l}|$$

$$\text{Elastic net (L1 + L2)} \quad R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

第二种方法是通过修改网络架构本身来实现正则化。事实上，我们使用正则化的想法是想得到一个更平滑的函数。早在 1995 年，Bishop 在其论文中用数学证明了“要求函数平滑”与“要求函数对输入的随机噪声具有适应性”之间的联系。

而在 2014 年，Srivastava 等人在论文《Dropout: A Simple Way to Prevent Neural Networks from Overfitting》中提出了一个想法：在训练过程中，计算后续层之前向网络的每一层注入噪声，这一想法成为暂退法，而之所以被称为暂退法，是因为从表面上看是在训练过程中丢弃 (drop out) 一些神经元。

我们在 AlexNet 的介绍中对 Dropout 这项技术有所提及。事实上，Dropout 也是从 2012 年的 AlexNet 开始出现于大众的视野中的。Dropout 的思想是：在每一个前向传播中，随机丢弃了一定比例的神经元，这个比例系数用概率 p 来表示，这也是一个超参数，通常设置为 0.5。

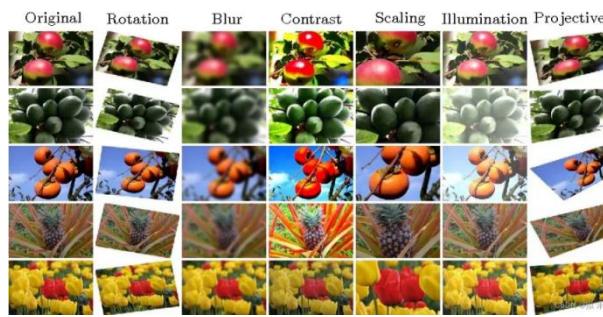


那么，为什么 Dropout 正则化可以解决过拟合问题呢？一个直观的解释是，我们在 Dropout 中丢弃了一部分的神经元，减少了参数数目，从而降低了模型的复杂度，解决过拟合。此外，还有一种解释是，使用 Dropout 正则化后，每一个节点都无法保证会留在网络中，

因此我们就不会给某一个神经元赋予太大的权重，从而起到了与 L2 正则化类似的效果。



在本节的最后，我们将讨论另外一种正则化，即增强传入网络进行训练的数据，这种正则化的方法我们之前也曾有所提及，叫做数据增强（Data Augmentation）。以图片数据为例，我们可以通过将原始的图片数据进行旋转，缩放，平移，仿射变换等方法进行数据增强，如下：



当然，数据增强也可以应用于图像以外的数据，比如文字，语音等，通过数据增强，可以生成具有多样性的训练样本，从而提高模型的鲁棒性，同时，也能使我们的模型在面对那些之前从未见过的数据时仍然具有可观的表现，这也就是模型的泛化（Generalization）能力的体现。

4.3.2 超参数调优

使模型获得更好表现的第二种方法是超参数调优。我们经常能从一些从事深度学习相关研究/工作的人们口中听说“调参”，这里的“调参”，其实就是调整模型的超参数，从而确定使模型性能最大化的超参数的正确组合，需要调节的超参数大致如下：



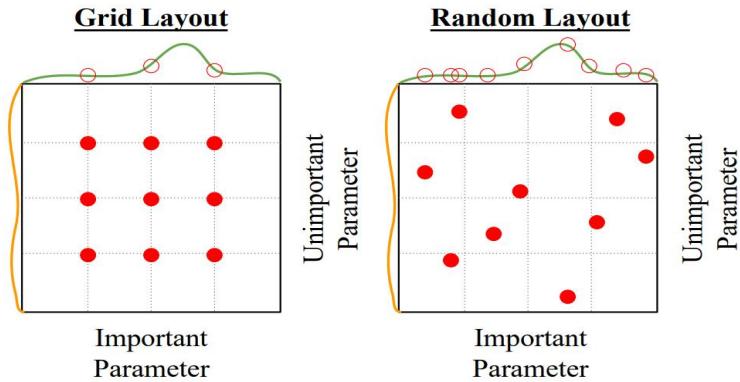
常见的超参数调优的方法主要有手动搜索，网格搜索（Grid Search）和随机搜索（Random

Search) 三种方法，而手动搜索显然不太适合我们现在的人工智能和深度学习的任务，因此，我们主要对后两种超参数搜索方法进行介绍。

首先，我们来看一下什么是网格搜索。网格搜索 (Grid Search) 是一种穷举搜索方法，它通过遍历超参数的所有可能组合来寻找最优超参数。常见的参数网格为学习率和权重衰退，每两个横 (纵) 坐标之间一般相差为 10 倍，常见的选择如下所示：

Weight Decay: $[1 \times 10^{-4}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}]$

Learning Rate: $[1 \times 10^{-4}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}]$



网格搜索的缺点是计算量较大，当超参数的数量和候选值较多时，搜索空间会急剧增大，导致搜索效率低下。于是就有了第二种方法，叫做随机搜索 (Random Search)，它通过随机采样超参数的组合来寻找最优超参数。与网格搜索相比，随机搜索不会遍历所有可能的超参数组合，而是在超参数空间中随机抽取一定数量的组合进行评估。

实验表明，随机搜索优于网格搜索。Bergstra 和 Bengio 在文章 *Random Search for Hyper-Parameter Optimization* 中说“随机选择比网格化的选择更加有效，而且在实践中也更容易实现”。通常，有些超参数比其余的更重要，通过随机搜索，可以让你更精确地发现那些比较重要的超参数的好数值。

4.3.3 模型集成和迁移学习

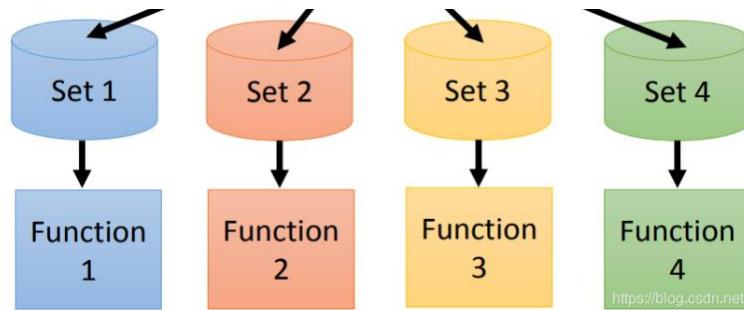
假如你已经“尽自己所能”完成了上面的步骤，但你仍然想让你的模型更进一步。模型集成 (Model Ensembles) 会是一个可行的选择，因为它可以让你的模型获得 2% 的 Bonus 准确率。

模型集成主要分为两步，首先我们需要独立地对不同模型进行训练，其次，我们在测试时可以对其结果取平均，具体做法是：取预测概率分布的平均值，然后利用 `argmax` 函数进行操作。主要的模型集成方法有：Bagging 和 Boosting，我们接下来将对这两种方法分别进行介绍。

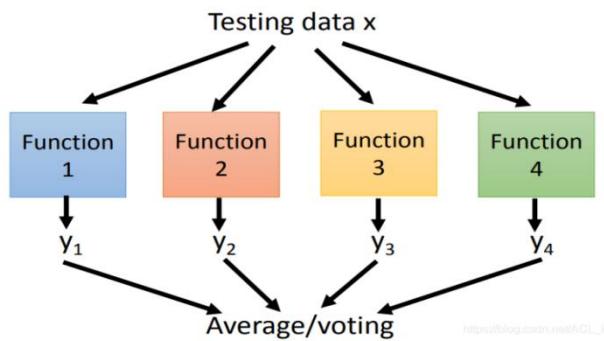
我们先来看一下什么是 Bagging。我们知道，对于一个模型，模型越复杂（越高次），方差会变得越大，bias 会变得越小。误差会先下降然后再上升，但假如我们把不同模型的输出做平均，得到新的模型的输出，这时的预测值就会和实际值比较接近。Bagging 做的事情就和这个类似。其具体做法如下：

对于训练部分，我们通过取样的方法将训练集分为 4 个不同的训练集，并以这 4 个数据

集训练出 4 个不同的模型，如下图所示：

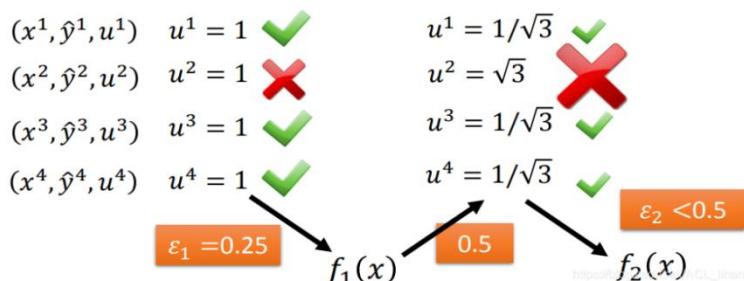


在上一步训练出 4 个模型之后，将测试数据输入到这 4 个模型中，得出 4 个结果，对于这 4 个结果，我们针对不同类型的问题有着不同的处理方法，如果是回归问题，就将这 4 个结果取平均得到最终的结果；如果是分类问题，就进行投票，看看哪一个类别会获得最多的票数，并将它作为最终的分类结果。



第二种常见的模型集成的方法是 Boosting，事实上，Bagging 方法通常被应用到强的模型中，目的是为了防止过拟合，而 Boosting 方法通常被用到一些比较弱的模型中，使它的拟合能力更强。

Boosting 方法的本质是不断学习，不断提升，因此，在 Boosting 的使用过程中，每一轮我们都需要去寻找一对“互补”的分类器。Boosting 的神奇之处在于，即便你所使用的模型有超过 50% 的错误率，在进行 Boosting 之后，错误率可以达到 0%。



Boosting 中有代表性的是 AdaBoost (Adaptive boosting) 算法，其算法思路如下：刚开始训练时对每一个训练例赋相等的权重，然后用该算法对训练集训练 t 轮，每次训练后，对训练失败的训练例赋以较大的权重，也就是让学习算法在每次学习以后更注意学错的样本，从而得到多个预测函数。

最后一个技巧是迁移学习 (Transfer Learning)，这一技巧在实做中经常会使用，我们在后续章节会单独拿出来介绍。

5 循环神经网络 (Recurrent Neural Network)

5.1 背景

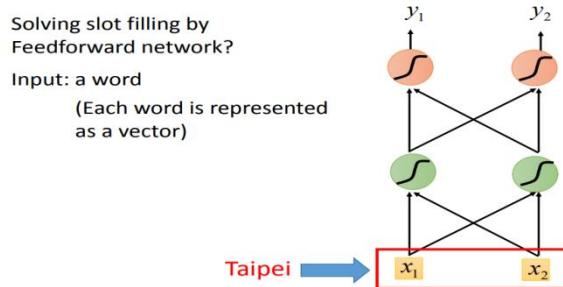
现在，智能机器人变得越来越流行，理解一段文字的一种方法是标记那些对句子有意义的单词或记号。在自然语言处理领域，这个问题被称为槽填充（Slot Filling）。

- Slot Filling



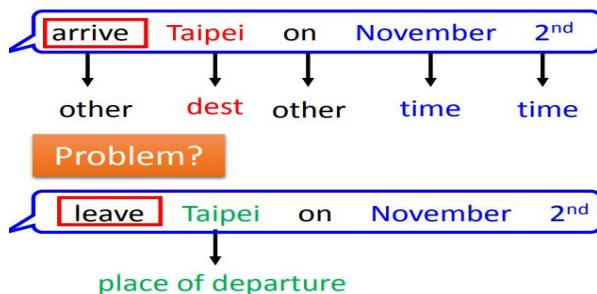
在这个订票系统的例子中，机器人要找出输入如句子的有用的信息（destination, time of arrival），然后输出要回答的答案。

对于上述问题，可以使用之前提及的前馈神经网络来解决，以上面的这句话为例：将输入的单词（比如“Taipei”）用向量来表示，输出是1个概率分布，表示输入的单词属于各个槽的概率。



那么，如何用向量表示1个单词呢？方法有很多。比如1-of-N Encoding(又称为独热编码，one-hot Encoding)，如下图所示。设定1个词汇表，向量的维度与词汇表的大小相同，并且二者是一一对应的，向量中，所对应单词的维度的值为1、其它维度的值为0，这种表示方法存在一定的问题，因为不是所有词汇都是见过的。

当然，前馈神经网络也存在着另外一个问题，以下面这两句话为例：

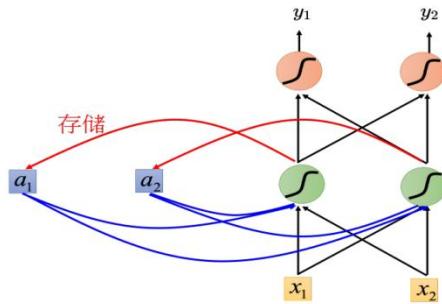


Taipei在这两句话中既是 destination 又是 place of departure，那么在之后的语境中，什么

时候我们应该将 Taipei 按 destination 处理，什么时候按 place of departure 处理呢？换言之，神经网络是需要记忆的。

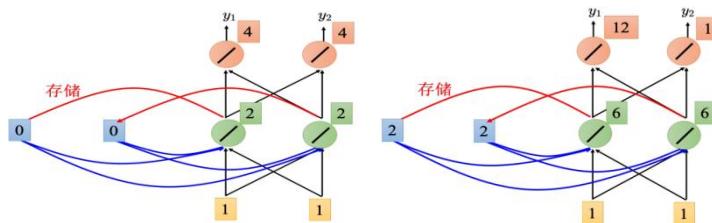
5.2 循环神经网络 (RNN)

事实上，我们在上一节提出的这种有记忆的神经网络称为循环神经网络 (Recurrent Neural Network, RNN)。在 RNN 中，每一个隐藏层中的神经元产生的输出都会被存到记忆单元中。而在下一轮中，除了输入的 x_1, x_2 之外，那些被存在记忆单元中的值也会影响输出值，换言之，记忆单元中的值也可以看作 RNN 的另一个输入。

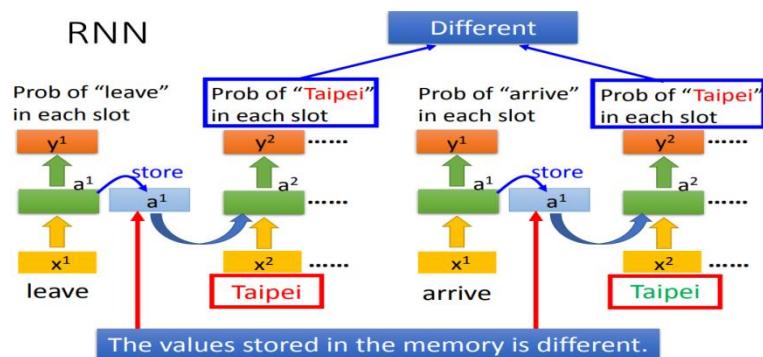


5.2.1 简单 RNN 的工作流程

接下来我们来大致了解一下 RNN 是如何运作的，以一个简单的线性 RNN 为例，假设所有的权重均为 1（没有偏置），当我们一开始输入 $[1, 1]^T$ 时，由于一开始神经网络中并没有记忆，它的初始化为 $[0, 0]^T$ ，因此绿色神经元的值为： $[2, 2]^T$ ，而输出值为 $[4, 4]^T$ 。

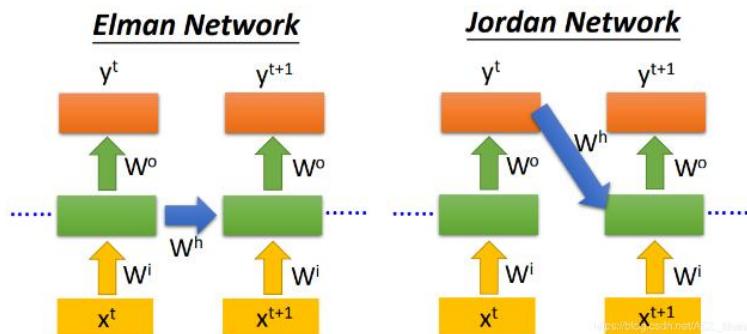


而在第二步中，绿色神经元中的值会被存储到记忆单元中，因此现在的记忆单元中存储的值为 $[2, 2]^T$ ，此时如果再次输入 $[1, 1]^T$ ，那么现在绿色神经元的值就变为 $[6, 6]^T$ ，而输出变为了 $[12, 12]^T$ 。以此类推，我们其实不难发现，在做循环神经网络时，它会考虑序列的顺序，输入序列调换顺序之后输出不同。

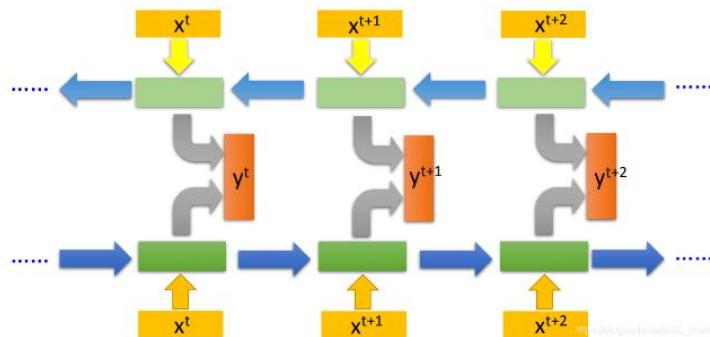


那么实际上，当我们在处理槽填充时，对于之前提到的两句话，尽管 Taipei 这个单词是一样的，但记忆单元会记住上下文中的一些关键词，比如 arrive, leave 等等。而存在记忆元里面的值不同，所以隐藏层的输出会不同，所以最后的输出也就会不一样。

当然，RNN 的架构是可以任意设计的，类似全连接神经网络，我们也可以将 RNN 叠很多层。事实上，刚才讲的 RNN 是其中一种结构，叫做 Elman Network，它只需要我们将隐藏层的值存起来，再在下一个时间点拿出来计算即可。而还有一种 RNN 叫做 Jordan Network，传说中它可以更好的表现，那它的做法是把输出的值存起来，下次用到再读出来。



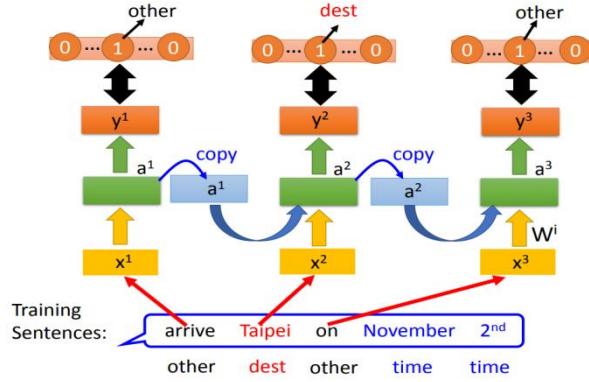
在实做中还有一种能获得更好性能的 RNN，叫做双向 RNN (Bidirectional RNN)，它的特殊之处在于，在任意时刻，它对于全部数据的记忆都有两个方向，也就是说，在任意时刻 t ，它的输出 y^t 都是由两个神经网络决定的，分别是从 $x^1, \dots, x^t, \dots, x^N$ (正向) 以及 $x^N, \dots, x^t, \dots, x^1$ (逆向)。



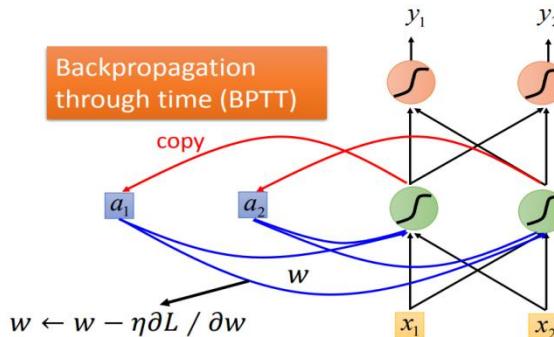
双向 RNN 的好处是，当神经网络产生输出的时候，它考虑的范围比较广。比如输入句子中间的词语进去，之前 RNN 只考虑了这个词语前面句子的部分。而双向 RNN 是考虑了句子前面和句子后面的部分，所以它的准确率会更高。

5.2.2 简单 RNN 的训练

经过上面的介绍，我们已经对 RNN 有了一个大概的了解，接下来就要看下它是如何训练的。我们仍然以槽填充为例，需要定义一个损失函数来评估模型的好坏。定义方法如下：给定一个输入的句子，我们将句子的每一个单词丢入循环神经网络进行训练，得到对应的一个输出： y_i ，并将其与相对应的参考向量进行交叉熵的计算。RNN 的损失函数输出和参考向量的交叉熵的和就是要最小化的对象。

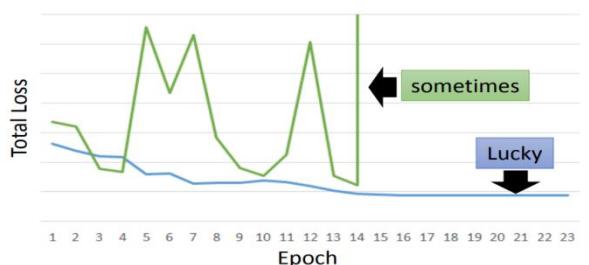


对于这个优化问题，RNN 也采用梯度下降进行求解。但是在循环神经网络中，计算梯度并不是简单的反向传播这么简单，在 RNN 中，采用的是一种随时间反向传播(BackPropagation Through Time, BPTT) 的方法，它和反向传播其实是很类似的，只是循环神经网络需要在时间序列上运作，所以在 RNN 中需要考虑时间上的信息，如下图所示：

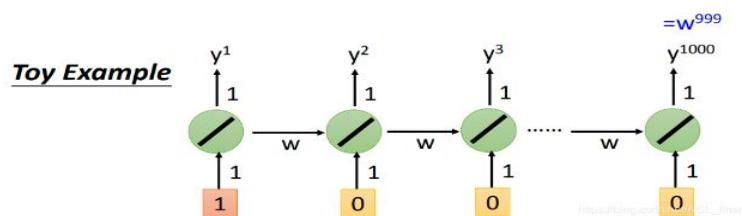


5.2.3 简单 RNN 训练的困难

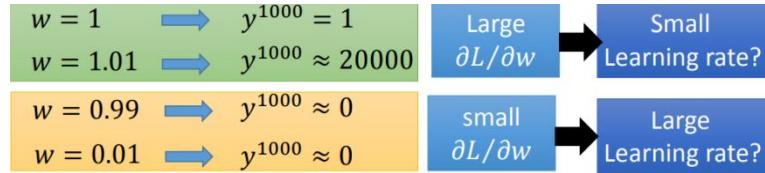
但实际上，RNN 的训练是比较困难的。对于 Gradient Descent 的问题，我们期待 Learning Curve 与蓝色的线是类似的，但在一些 Language Modeling 中，有些曲线会产生上下振荡的趋势，就如同绿颜色的线一样。



事实上，有人发现 RNN 的误差表面是粗糙的，这是因为在 RNN 中，有很多东西在不同时间内被反复使用。举一个很简单的例子，假设我们有一个“世界上最简单的”RNN 如下图所示：



这个网络很简单。只有一个神经元。这个神经元的输出会作为下一个神经元的输入，加上下一时间点的输入，一起被输入到下一时间点的神经元中。并且所有权重都是 1，那么如果我们输入 $[1, 0, \dots, 0]$ ，那时间点为 1000 的输出就是 w^{999} 。而就算 w 在训练过程中只有 0.01 的变化量，我们都会有一下两种情形：



由于这里有两种变化情况，所以不能很死板地说，用大的学习率或者小的学习率就是好的。

所以 RNN 不好训练的原因是：它有时间顺序，同样的权重在不同的时间点会被反复使用多次，从而导致梯度消失（gradient vanishing）或是梯度爆炸（gradient explode）的问题。

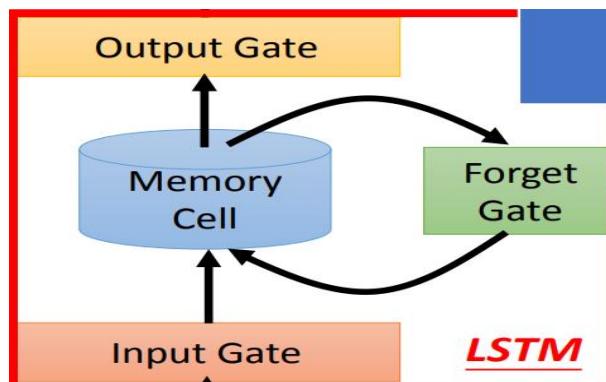
那么，如何解决以上问题呢？广泛被拿出来解决的方法是使用现代循环神经网络，其中最常见的现代循环神经网络是长短期记忆网络（Long Short-Term Memory Network, LSTM），我们将在下一节对 LSTM 进行介绍。

5.3 长短期记忆神经网络（LSTM）

5.3.1 门控记忆元构成的 RNN——LSTM 介绍

我们在上一小节中介绍了简单循环神经网络，事实上，我们之前提到的记忆元是最单纯的，可以随时把值存到记忆元去，也可以把值读出来。但最常用的记忆元是长短期记忆网络（Long Short-Term Memory Network, LSTM）。

LSTM 包括 3 个门，分别是：输入门（Input Gate），输出门（Output Gate）和遗忘门（Forget Gate），以及 1 个记忆单元（Memory Cell），Input Gate 是决定神经元的 output 要不要被保存到 Memory Cell（由 network 自己学习并决定是否打开阀门），Forget Gate 决定 Memory Cell 里面的东西要不要删掉（由网络自己学习并自己决定是否需要忘记这些内容），而 Output Gate 决定神经元能不能从 Memory Cell 读取之前保存的东西（由网络自己学习并决定是否打开阀门）。其结构如下图所示：



整个 LSTM 可以看成有 4 个输入、1 个输出。在这 4 个输入中，一个是想要被存在

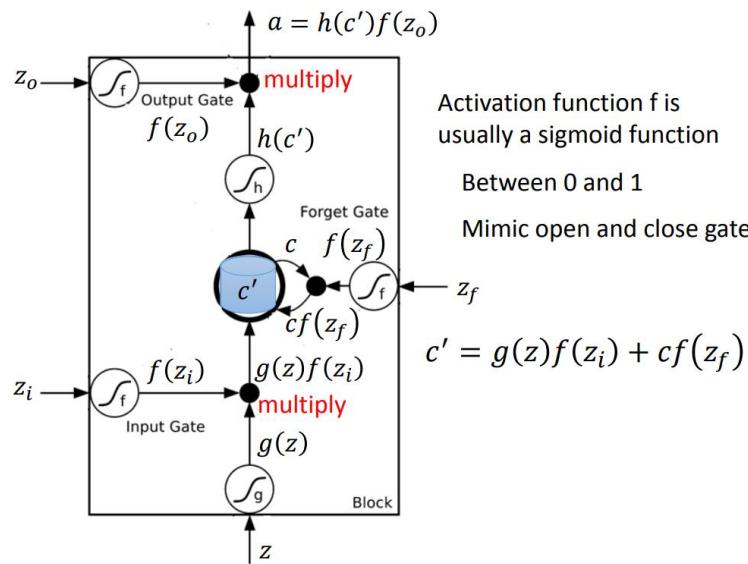
记忆元的值，但不一定能存进去，还有操控输入门的信号、操控输出门的信号、操控遗忘门的信号，有着四个输入但它只会得到一个输出。

5.3.2 LSTM 的计算与架构

在 LSTM 中，记忆单元的计算公式为：

$$c' = g(z)f(z_i) + cf(z_f)$$

如图所示，底下输入的是 z ，遗忘门，输入们和输出门均有一个与之对应的操纵其的数值： z_f, z_i, z_o 。而这三个操纵值均会通过一个激活函数 f ，通常选取 sigmoid 函数，从而确保输出的值在 0-1 之间，结构如下图所示：



这边需要注意的是。LSTM 中的遗忘门的开关是反直觉的，当我们将遗忘门打开时，代表“记得”，而关闭则代表“遗忘”。接下来，为了方便理解，我们可以做一个“人体 LSTM”来方便我们理解这个算法。

如下图所示，网络里面只有一个 LSTM 的单元，输入都是三维的向量，输出都是一维的输出。这三维的向量跟输出还有记忆元的关系是这样的。假设 x_2 的值为 1， x_1 的值会被写到记忆元里，而若为 -1，则会重置记忆元。假设 x_3 的值为 1 时，才会把输出打开，才能看到输出，看到记忆元的数字。

x_1	0	0	3	3	7	7	0	6
x_2	1	3	2	4	2	1	-1	6
x_3	0	0	0	0	0	1	0	1
y	0	0	0	0	0	7	0	6

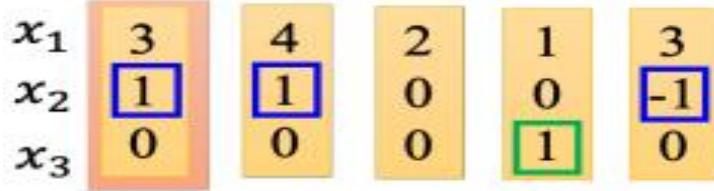
When $x_2 = 1$, add the numbers of x_1 into the memory

When $x_2 = -1$, reset the memory

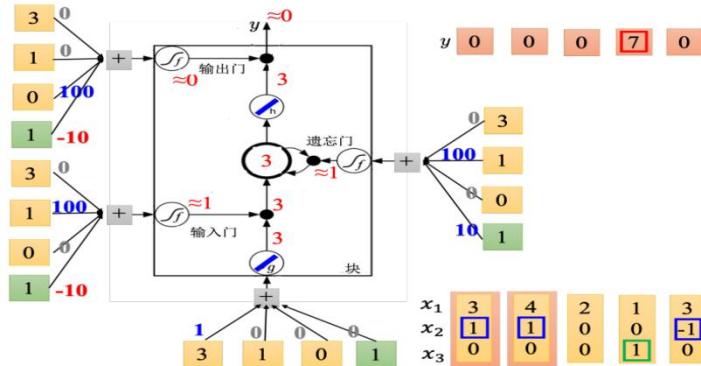
When $x_3 = 1$, output the number in the memory.

接下来我们举一个 LSTM 实际运算的例子，对于下图中的一个记忆元，其四个输入标量

是由输入的三维向量乘以线性变换 (linear transform) 后所得到的结果。假设 (x_1, x_2, x_3) 的参数 (weight, bias) 均是事先知道的。给定一系列输入，LSTM 的运算如下：



为了简化计算，假设 g 和 h 都是线性的。第一步输入的向量为： $[3, 1, 0]^T$ ，因此输入的值为 $3*1=3$ ，而此时输入门经过 sigmoid 运算约等于 1（被打开），遗忘门也约等于 1，也被打开，而输出门接近于 0，可以看作被关闭。因此，3 无法通过，输出值仍然为 0，结果如下：

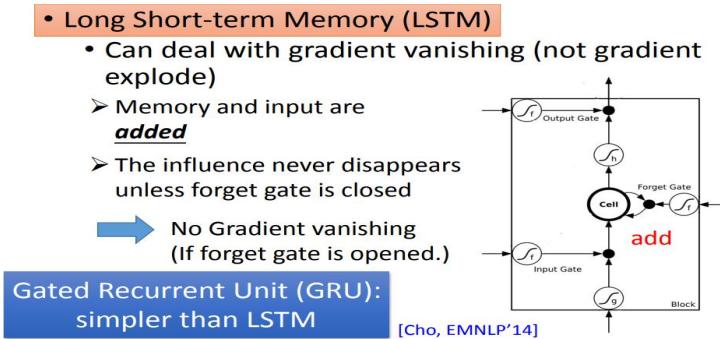


依次输入剩余的 4 个向量，根据上述运算，我们最终可以得到输出向量。需要注意的是，相比于一般的神经网络，LSTM 需要 4 倍的参数，当然，LSTM 的结构也是可以由人为决定的，我们同样可以得到多层的 LSTM，也可以使用双向 LSTM (Bi-LSTM)。

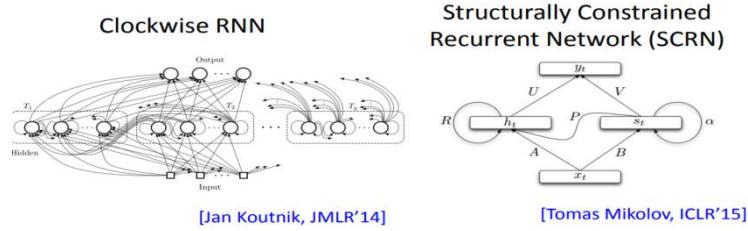
5.3.3 LSTM 为什么比简单 RNN 更好

我们在前文中说到，为了解决简单 RNN 可能出现的梯度爆炸或者梯度消失的问题，通常采用的办法是使用 LSTM。那为什么 LSTM 是一个不错的处理方式呢？事实上，LSTM 主要是将那些平坦的地方拿掉，因此它只能用于解决梯度消失的问题，而不能解决梯度爆炸的问题。

实际上，LSTM 之所以可以处理梯度消失的问题，与其结构有关。在 RNN 中，对于每个时间点，memory 都会被覆盖更新，而在 LSTM 中，memory 和 input 是相加的，因此，weight 对于 memory 一旦造成影响，那这个影响是会一直存在的，除非 forget gate 关闭。



当然，我们还可以进一步对 LSTM 进行参数计算层面上的优化，由于 LSTM 有 3 个门，参数量较大，而 Gated Recurrent Unit (GRU) 只有 2 个门，需要的参数量比较少，所以它在训练的时候是比较鲁棒的。如果训练 LSTM 的时候，过拟合的情况很严重，可以尝试用 GRU 替代。

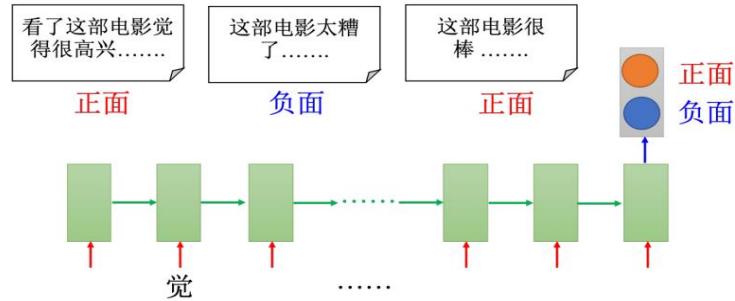


当然还有其他技术可以处理梯度消失的问题。比如顺时针循环神经网络 (clockwise RNN) 或结构约束的循环网络 (Structurally Constrained Recurrent Network, SCRN) 等等。

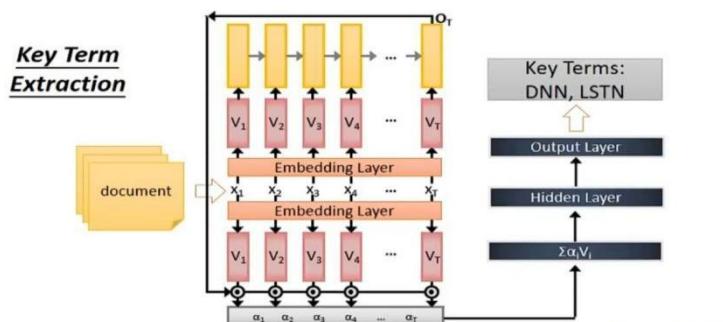
5.4 循环神经网络的应用

5.4.1 多对一序列 (Many to one)

RNN 的输入可以是一个向量序列，输出是一个向量。对于这样的问题，我们称其为“多对一”的问题，它的一个典型应用是情感分析 (Sentiment Analysis)，即输入 1 篇文章或 1 句话等 (1 个 vector sequence)，输出其情感倾向 (分类或者回归，比如超好、好、普通、差、超差、[-1,1])。

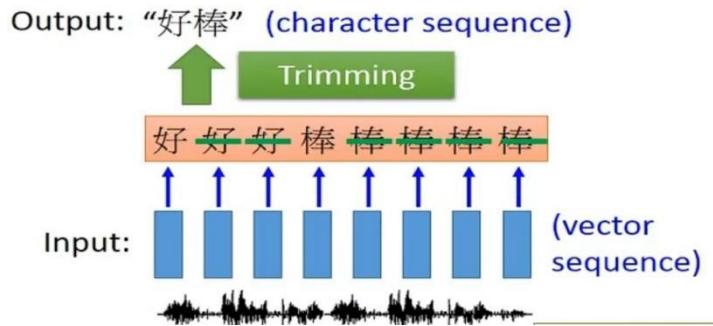


情感分析是一个分类问题，但是因为输入是序列，所以用 RNN 来处理。此外，RNN 还可以用来做关键词提取 (Key Term Extraction)，即给机器看一个文章，机器要预测出这篇文章有哪些关键词。如下图所示，如果能够收集到一些训练数据 (一些文档，这些文档都有标签，哪些单词是对应的，那就可以直接训练一个 RNN)，那这个 RNN 把文档当做输入，通过嵌入层 (embedding layer)，用 RNN 把这个文档读过一次，把出现在最后一个时间点的输出拿过来做注意力，可以把这样的信息抽出来再丢到前馈神经网络得到最后的输出。

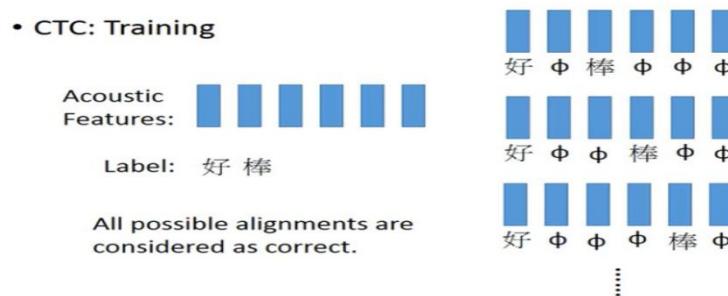


5.4.2 多对多序列 (Many to Many)

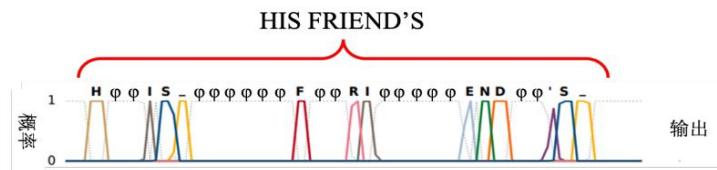
RNN 的输入和输出都可以是序列，只不过在“多对多”的任务中，输出序列长度更短。对于这类任务，语音辨识 (Speech Recognition) 是最典型的一个例子。在语音辨识的任务中，输入是 1 段声音信号，每隔 1 小段时间（通常很短，比如 0.01 秒）就用 1 个向量表示，输出是 1 段文字。因此输入是 1 个向量序列，而输出是 1 个字符序列。



但是如果我们采用了上面的技术，会产生另外一个问题，我们无法识别“好棒棒”，因为我们将多余的“棒”字都给删掉了，解决方法叫做 CTC，它的做法是加入一个新的 NULL 字符，可以在输出中填充 NULL，最终输出时删除 NULL 即可。

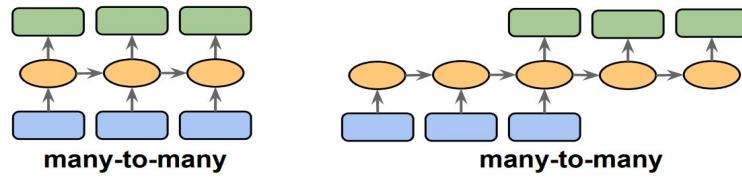


在做英文识别的时候，RNN 输出目标就是字符（英文的字母 + 空白）。直接输出字母，然后如果字和字之间有边界，就自动有空白。如下图所示，第一帧是输出 H，第二帧是输出 null，第三帧是输出 null，第四帧是输出 I 等等。如果我们看到输出是这样子话，最后把“null”的地方拿掉，这句话的识别结果就是“HIS FRIEND’ S”。我们不需要告诉机器说：“HIS”是一个单词，“FRIEND’s”是一个单词，机器通过训练数据会自己学到这件事情。如果用 CTC 来做语音识别，就算是有某一个单词在训练数据中从来没有出现过（比如英文中的人名或地名），机器也是有机会把它识别出来。

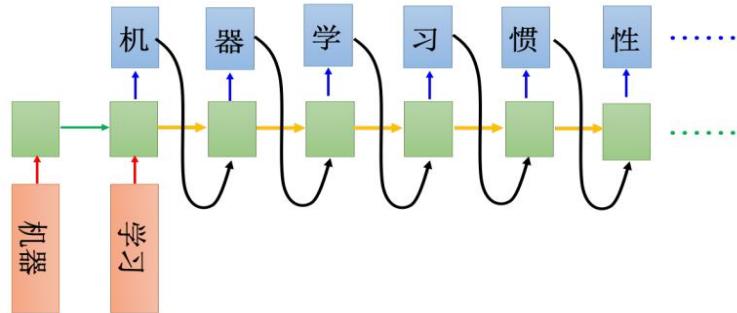


另外一个 RNN 的应用叫做序列到序列 (Sequence to Sequence, Seq2Seq) 学习，在序列到序列学习里面，RNN 的输入跟输出都是序列 (但是两者的长度是不一样的)。在 CTC 中，输入比较长，输出比较短。在这边我们要考虑的是不确定输入跟输出谁比较长谁比较短。比如机器翻译 (machine translation)，输入英文单词序列把它翻译成中文的字符序列。英文和

中文序列的长短是未知的。



假如输入“机器学习”，然后用 RNN 读过去，然后在最后一个时间点，这个记忆元里面就存了所有输入序列的信息，如下图所示。

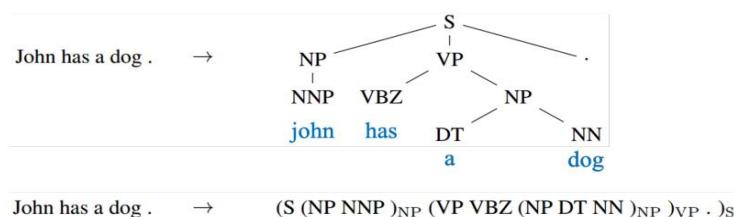


在上面的过程中，机器相当于是一个一个地将文字“吐出来”，让机器吐一个字符（机），然后就让它输出下一个字符，把之前的输出出来的字符当做输入，再把记忆元里面的值读进来，它就会输出“器”，依次类推。那怎么让机器停止生成文字呢？我们只需要加入一个特殊的表示“断”的符号即可。

添加符号“==”（断）

5.4.3 “超越序列”（Beyond Sequence）的应用

序列到序列的技术也能用到句法解析（syntactic parsing），即让机器看一个句子，得到句子结构树。如下图所示，只要把树状图描述成一个序列，比如：“John has a dog.”，序列到序列学习直接学习一个序列到序列模型，其输出直接就是句法解析树，这个是可以训练的起来的。LSTM 的输出的序列也是符合文法结构，左、右括号都有。



5.4.4 自编码器上的应用

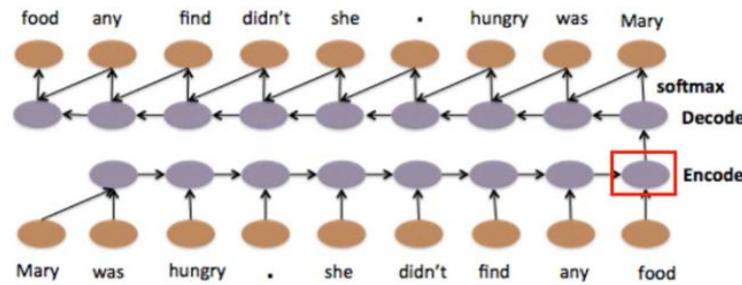
此外，序列到序列的技术还能应用在自编码器（Autoencoder）中。考虑一段文档，在之前我们曾经提到，要想将这个文档表示成一个向量，往往会用词袋（Bag-of-Words, BoW）的方法。但是这个方法的问题是，往往会忽略掉单词序列的信息。举例来说，有一个单词序列是“white blood cells destroying an infection”，另外一个单词序列是：“an infection

destroying white blood cells”，这两句话的意思完全相反的。但是我们用词袋的方法来描述的话，他们的词袋完全是一样的。它们里面有完全一摸一样的六个单词，因为单词的顺序是不一样的，所以他们的意思一个变成正面的，一个变成负面的，他们的意思是不一样的。

white blood cells destroying an infection → 正面

an infection destroying white blood cells → 负面

那么在 A Hierarchical Neural Autoencoder for Paragraphs and Documents 这篇文章中，作者提出：可以使用一种名叫 Seq2Seq Autoencoder 的结构，在考虑语序的情况下把文章编码成向量，只需要将 RNN 作为编码器和解码器即可。

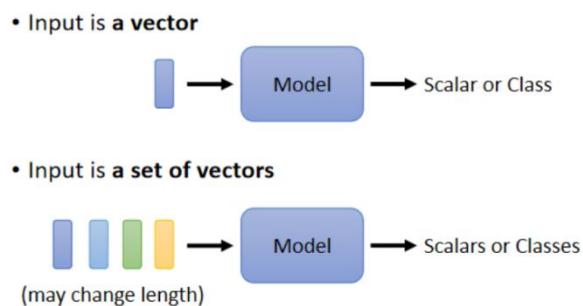


如上图所示，将输入 RNN 的单词编码成 embedded 的向量，然后再通过另 1 个 RNN 解码，如果解码后能得到一模一样的句子，则编码得到的向量就表示了这个单词序列中最重要的信息。

6 自注意力机制 (Self-attention)

6.1 输入是序列的情况 (Sequence as Input)

在图像识别中，我们曾经做过这样的处理：将所有输入的图像全部变成同等大小。但如果问题变得更复杂的话，输入是一组向量，并且输入的向量的数量是会改变的，即每次模型输入的序列长度都不一样，这个时候应该要怎么处理呢？我们通过具体的例子来讲解处理方法。

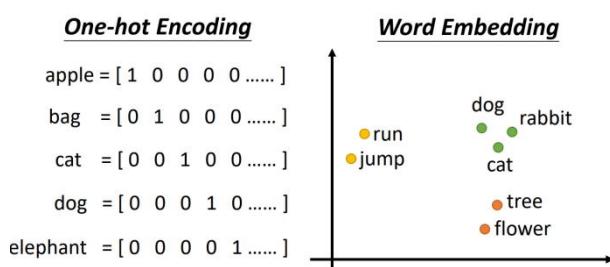


6.1.1 输入是向量组 (Vector Set) 的例子

第一个例子是文字处理，假设我们的神经网络中输入是一个句子，每一个句子的长度都不一样，每个句子的词汇的数目都不一样。

如果把一个句子里面的每一个词汇都描述成一个向量，用向量来表示，模型的输入就是一个向量序列，而且该向量序列的大小每次都不一样（句子的长度不一样，向量序列的大小就不一样）。将词汇表示成一个向量最简单的方法就是独热编码（One-hot Encoding）。

One-hot Encoding 的表示方法是和单词数量有关的，因为对于每一个单词，我们都需要产生一个独一无二的编码，而一般某种语言（英语）有上百万个单词，而用上百万的维度的 one-hot vector 来表示一个单词的参数未免也有些太多了，而且这种表示方法无法体现出单词与单词之间的关系。那么在这里，有另外一种表示方法叫做词嵌入（Word Embedding），如下图所示：



当然，声音讯号也可以看作一个向量序列作为输入。我们会将一段声音讯号取一个范围，这个范围叫做一个窗口，把这个窗口里面的资讯描述成一个向量，在语音中，我们把这个向量叫做一个桢（Frame），通常窗口的长度就是 25 毫秒。



为了要描述一整段的声音信号，我们会把这个窗口往右移一点，通常移动的大小是 10 毫秒。（注意：有人可能会问为什么窗口的长度是 25 毫秒，窗口移动的大小是 10 毫秒？在这边给出的回答是：这些都是古圣先贤实践得出的最佳结果，无需多问！）。

总之，一段声音信号就是用一串向量来表示，而因为每一个窗口，他们往右移都是移动 10 毫秒，所以一秒钟的声音信号有 100 个向量，所以一分钟的声音信号就有 6000 个向量。所以语音它里面包含的信息量其实是非常可观的，声音信号也是一堆向量。



一个图（graph）也是一堆向量。如图所示，社交网络是一个图，在社交网络上面每一个节点就是一个人。每一个节点可以看作是一个向量。每一个人的简介里面的信息（性别、年龄、工作等等）都可以用一个向量来表示。所以一个社交网络可以看做是一堆的向量所组成的。

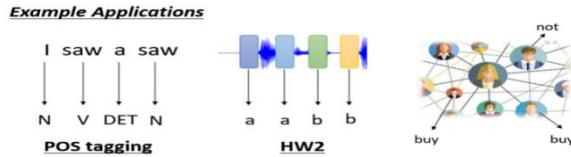
6.1.2 输出可能长什么样？

在上一小节我们介绍了序列可能出现的一些常见的输入，那对于这些输入来说，输出其实有以下三种可能。

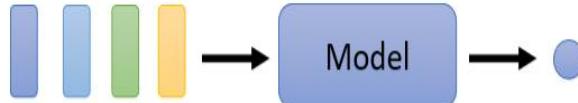
第一种可能是：每一个向量都有一个对应的标签。换言之，对于一个有着 N 个向量作为输入的模型，其输出也有一一对应的 N 个标签，而对于标签而言，假如它是一个标量，那就是一个回归问题；假如它是一个类别，那就是一个分类问题。



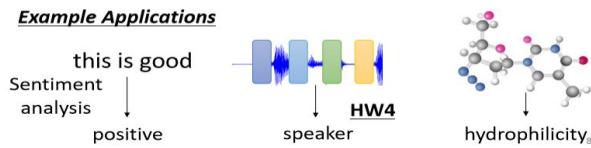
举个例子，如下图所示，在文字处理上，假设我们要做的是词性标注（Part-Of-Speech tagging, POS tagging）。机器会自动决定每一个词汇的词性，判断该词是名词还是动词还是形容词等等。现在有一个句子：I saw a saw，这句话的意思是我看到一个锯子，第二个 saw 是名词锯子。所以机器要知道，第一个 saw 是个动词，第二个 saw 是名词，每一个输入的词汇都要有一个对应的输出的词性。这个任务就是输入跟输出的长度是一样的情况，属于第一个类型的输出。如果是语音，一段声音信号里面有一串向量。每一个向量都要决定它是哪一个音标。这不是真正的语音识别，这是一个语音识别的简化版。如果是社交网络，给定一个社交网络，模型要决定每一个节点有什么样的特性，比如某个人会不会买某个商品，这样我们才知道要不要推荐某个商品给他。以上就是举输入跟输出数量一样的例子，这是第一种可能的输出。



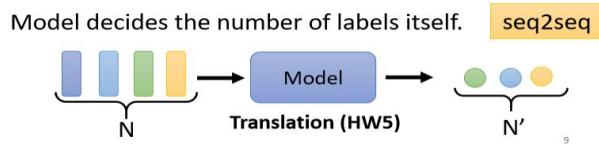
第二种可能的输出是：整个序列只需要输出一个标签即可，如下图所示：



我们同样以文字，语音和图分别举例。对于文字来说，一个典型的应用是做情感分析（Sentiment Analysis），即给机器看一段话，它要决定说这段话是正面的还是负面的。如果是语音的例子的话呢，这个模型常用于语音辨认，即机器听一段声音，再决定是谁讲的这个声音。如果是图，比如给定一个分子，预测该分子的亲水性。



最后一种可能的输出是：机器需要自己决定标签数量，这种任务也被称为序列到序列（Sequence-to-Sequence，简称 seq2seq）。

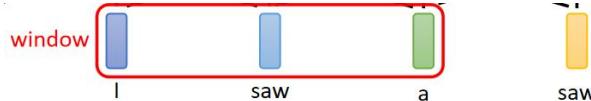


翻译就是序列到序列的任务，因为输入输出是不同的语言，它们的词汇的数量本来就不会一样多。真正的语音识别输入一句话，输出一段文字，其实也是一个序列到序列的任务。

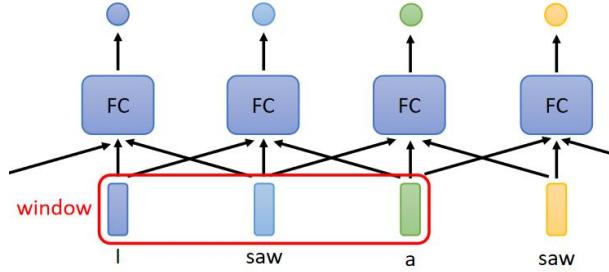
6.2 自注意力机制的提出与介绍

6.2.1 背景——序列标注的解决办法

在上一小节提出的三种可能的输出中，我们主要关注第一种，即输入和输出一样的情况。这一现象也被称为序列标注（Sequence Labeling）。



那么我们该如何解决这一问题呢？最直觉的想法是采用一个全连接神经网络。但采用这种方法会有很大的瑕疵！以上面这句话为例，在全连接层中，我们无法处理（saw 的）一词多义的问题。

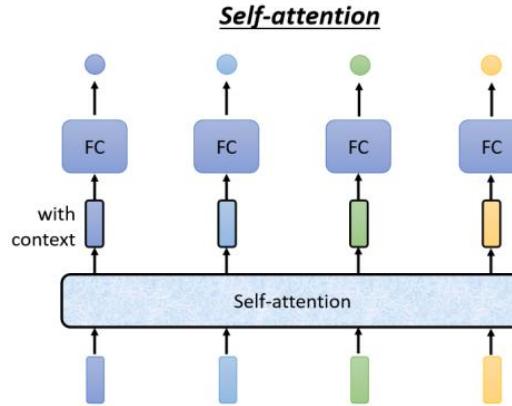


因此，我们的网络需要考虑更多的信息，比如上下文，做法是把每个向量的前后几个向量都“串”起来，一起输入到全连接网络即可。

但是这样的方法还是存在极限，因为即使我们的上下文考虑得再多，也是会有一定限制的。那这边真正的问题是，如果今天我们有某一个任务，不是考虑一个窗口就可以解决的，而是要考虑一整个序列，那要怎么办呢？

有人可能会想说这个很容易，我就把窗口开大到可以把整个序列盖住就可以了。但问题又出现了，序列的长度是有长有短的。如果你今天说我真的要开一个窗口，把整个序列盖住，那你可能要统计一下你的训练资料，然后看你的训练资料里面，最长的序列有多长然后开一个比最长的序列还长的窗口才有可能把整个序列盖住。

但这种做法就意味着，全连接网络需要非常多的参数，可能不只运算量很大，还容易出现过拟合。事实上，如果想要更好地考虑整个输入序列的信息，就要用到自注意力模型（Self-attention Model）。

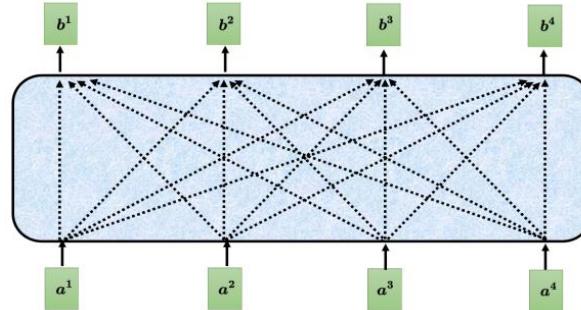


自注意力模型会将整个序列的数据作为输入，输入几个向量，它就输出几个向量。上图中输入 4 个向量，它就输出 4 个向量。而这 4 个向量都是考虑整个序列以后才得到的，所以输出的向量有一个黑色的框，代表它不是一个普通的向量，它是考虑了整个句子以后才得到的信息。接着再把考虑整个句子的向量丢进全连接网络，再得到输出。因此全连接网络不是只考虑一个非常小的范围或一个小的窗口，而是考虑整个序列的信息，再来决定现在应该要输出什么样的结果。

需要注意的是，Self-attention 并不是只能用一次，它可以叠很多次，如下图所示，自注意力模型的输出通过全连接网络以后，得到全连接网络的输出。全连接网络的输出再做一次自注意力模型，再重新考虑一次整个输入序列的数据，将得到的数据输入到另一个全连接网络，就可以得到最终的结果。全连接网络和自注意力模型可以交替使用。全连接网络专注于处理某一个位置的信息，自注意力把整个序列信息再处理一次。

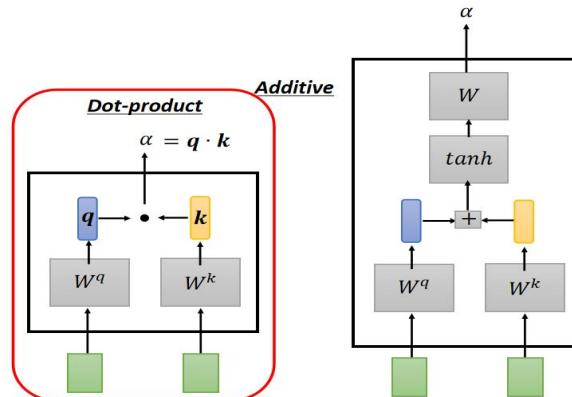
6.2.2 自注意力机制的运作

接下来我们来讨论一下自注意力模型的运作方式。其输入是一串的向量，这个向量可能是整个网络的输入，也可能是某个隐藏层的输出，在这里我们用 α 来表示。输入一组向量 α ，目标是输出一组向量 b ，其中，对于任意的 $b^i, i = 1, 2, 3, 4$ ，都是考虑了整个 α 向量得出的。

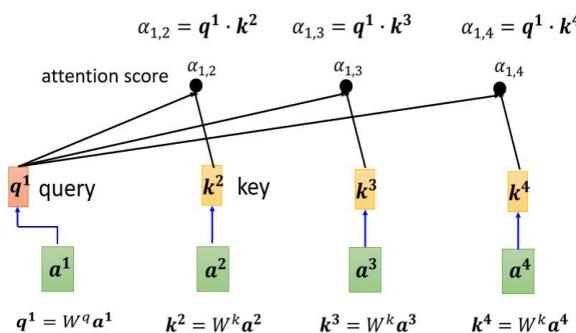


那么 b^i 是如何产生的呢？以第一个向量 b^1 为例（其余类似），第一步是根据 a^1 找出输入序列中与其相关的向量，每个向量与 a^1 的关联程度可以由一个数值 α 来表示。这个数值的计算方法如下：

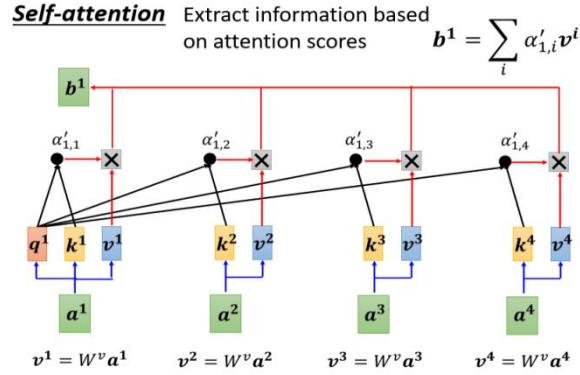
比较常见的计算方法是点乘（Dot-product）的方法：将左边这个向量乘上矩阵 W^q 得到向量 q ，右边这个向量乘上矩阵 W^k 得到向量 k ，随后我们将向量 q 与 k 进行点乘，就得到了 α 的值，当然还有一些其他的计算方法，如下图所示：



那么，我们需要把 α^1 分别与 $\alpha^2, \alpha^3, \alpha^4$ 计算它们之间的关联性。这边的做法是将 α^1 乘上矩阵 W^q 得到一个查询向量（query vector），而对于其他几个乘上 W^k 得到键向量（key vector），计算这两个向量的内积就可以得到 $\alpha_{1,j} (j = 2, 3, 4)$ ，我们将其称为注意力分数（Attention Score）。



计算出 α^1 跟每一个向量的关联性以后，会接入一个 softmax 函数，得到 $\alpha' = softmax(\alpha)$ ，我们需要根据 α' 去抽取序列中重要的资讯。具体做法是：将 $\alpha^i (i = 1, 2, 3, 4)$ 这边每一个向量，乘上矩阵 W^v 得到新的向量 $v^i (i = 1, 2, 3, 4)$ 。那么 b^1 的计算方法为： $\sum_i (\alpha'_{1,i} \cdot v^i)$



我们可以从矩阵的角度来理解这一机制的运算。事实上，对于 $\alpha^i, i = 1, 2, 3, 4$ 而言，每一个向量都需要与 W^q 相乘得到一个键向量： q^i 。那么 q^1, q^2, q^3, q^4 按照列合并可以得到一个矩阵 \mathbf{Q} ，同理， k^1, k^2, k^3, k^4 按照行合并可以得到一个矩阵 \mathbf{K} 。通过两个矩阵的相乘就得到注意力的分数，我们将其存入矩阵 \mathbf{A} 中。

$\alpha'_{1,1}$	$\alpha'_{2,1}$	$\alpha'_{3,1}$	$\alpha'_{4,1}$
$\alpha'_{1,2}$	$\alpha'_{2,2}$	$\alpha'_{3,2}$	$\alpha'_{4,2}$
$\alpha'_{1,3}$	$\alpha'_{2,3}$	$\alpha'_{3,3}$	$\alpha'_{4,3}$
$\alpha'_{1,4}$	$\alpha'_{2,4}$	$\alpha'_{3,4}$	$\alpha'_{4,4}$

←

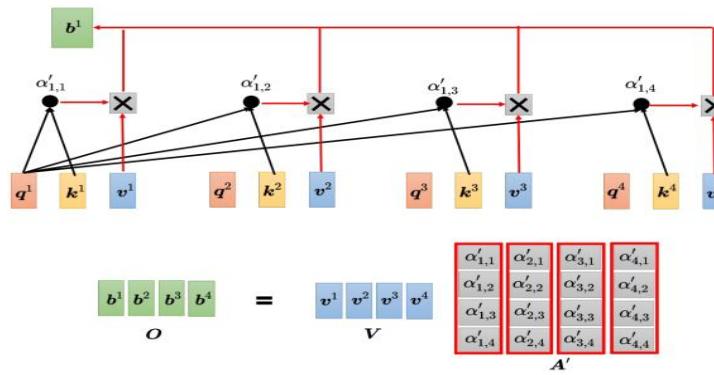
$\alpha_{1,1}$	$\alpha_{2,1}$	$\alpha_{3,1}$	$\alpha_{4,1}$
$\alpha_{1,2}$	$\alpha_{2,2}$	$\alpha_{3,2}$	$\alpha_{4,2}$
$\alpha_{1,3}$	$\alpha_{2,3}$	$\alpha_{3,3}$	$\alpha_{4,3}$
$\alpha_{1,4}$	$\alpha_{2,4}$	$\alpha_{3,4}$	$\alpha_{4,4}$

A' softmax A K^T

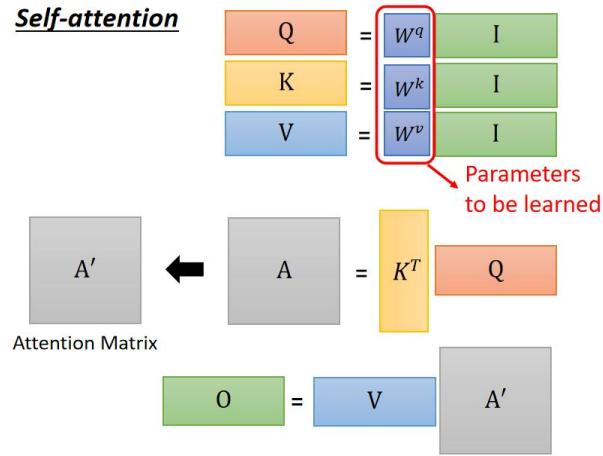
$$= \begin{matrix} (k^1)^T \\ (k^2)^T \\ (k^3)^T \\ (k^4)^T \end{matrix} \quad \begin{matrix} q^1 \\ q^2 \\ q^3 \\ q^4 \end{matrix}$$

Q

接下来，对于注意力分数矩阵 \mathbf{A} ，我们进行一次 softmax 函数的操作，得到一个新的矩阵 $A' = softmax(A)$ ，具体操作如上图。计算出 A' 之后，还需要把 v^1 到 v^4 乘上对应的 α' 再相加得到 b ，那实际上，我们同样可以将 v^1 到 v^4 按照列拼起来得到一个矩阵 \mathbf{V} 。因此，自注意力机制又相当于一个矩阵乘法，如下所示：

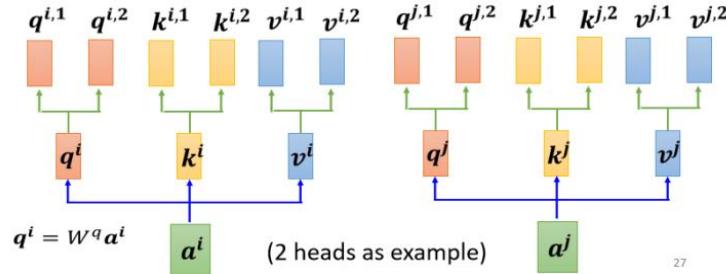


事实上，从矩阵乘法的角度来看待 Self-attention 的优点是，我们只需要学习三个矩阵参数： W^Q, W^K, W^V 即可，而其余参数都是人为设定好的，不需要通过训练数据学习。整个 Self-attention 的结构如下：

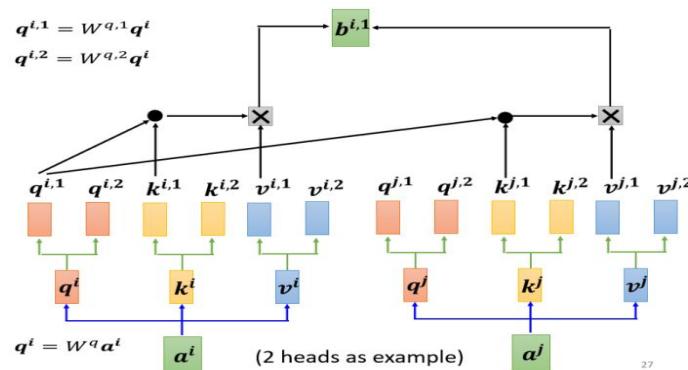


6.2.3 多头注意力机制 (Multi-head Self-attention)

自注意力有一个进阶的版本——多头自注意力 (Multi-head Self-attention)。多头自注意力的使用是非常广泛的，有一些任务，比如翻译、语音识别，用比较多的头可以得到比较好的结果。至于需要用多少的头，这个又是另外一个超参数。为什么需要比较多的头呢？在使用自注意力计算相关性的时候，就是用 q 去找相关的 k 。但是相关有很多种不同的形式，所以也许可以有多个 q ，不同的 q 负责不同种类的相关性，这就是多头注意力。

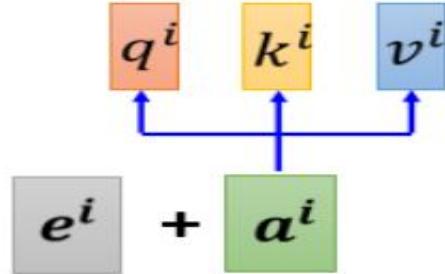


那么多头注意力机制是怎么运作的呢？以两个头为例，我们先将 a 乘上一个矩阵 q ，然后将结果乘上两个矩阵： q^1, q^2 (k, v 同理)。这里在计算注意力分数的时候，上标为 1 的 q, k, v 互相进行运算，并进行类似的 softmax 和矩阵运算，具体运算如下所示：



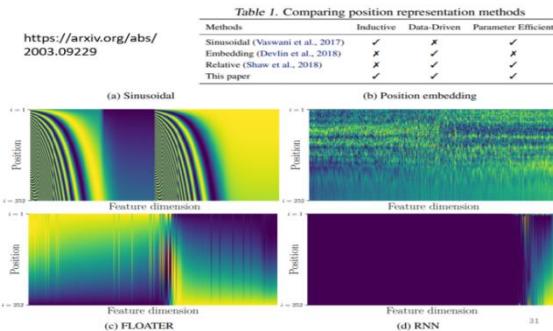
那讲到目前为止，你会发现自注意力层少了一个也许很重要的信息，即位置的信息。对一个自注意力层而言，每一个输入是出现在序列的最前面还是最后面，它是完全没有这个信息的。

事实上，对 Self-attention 而言，位置 1、位置 2、位置 3 跟位置 4 没有任何差别，这四个位置的操作是一模一样的。但在有些特殊的任务（比如：词性标注）中，我们又需要用到这样的信息（举例：动词基本不放在句首）。



因此做自注意力的时候，如果我们觉得位置的信息很重要，需要考虑位置信息时，就要用到位置编码（positional encoding）。如上图所示，位置编码为每一个位置设定一个向量，即位置向量（positional vector），用 e^i 表示，其中 i 表示位置。

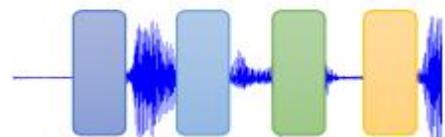
那在最早的 Transformer 那篇论文中，每一列就代表一个 e ，作者希望模型在处理输入的时候，它可以知道现在的输入的位置的信息。那这种位置向量的设定是人为的，当然人为设定也会出现很多问题。事实上在最早的“Attention Is All You Need”论文中，其位置向量是通过正弦函数和余弦函数所产生的，避免了人为设定向量固定长度的尴尬。总之位置编码的方法是各式各样的，也是一个值得研究的学问。



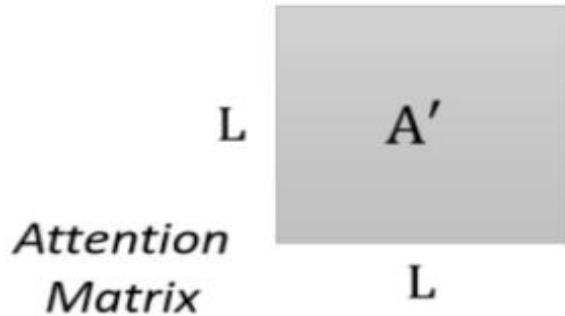
6.3 自注意力机制的应用

6.3.1 在语音上的应用

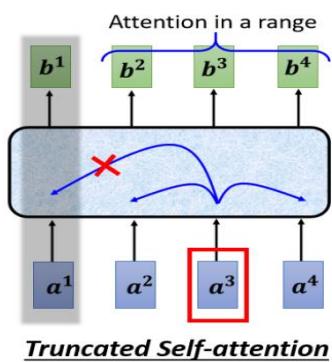
在做语音相关的任务时，我们也可以使用自注意力机制，不过可能需要做一些改动。因为一般的语音讯号非常长，假如我们不加变动地将其排成一个向量，那这个向量的维度会非常高。



需要注意的是，在我们的注意力矩阵（attention matrix）的计算中，其计算的复杂度与向量维度的平方是成正比的，也就是说，我们需要很大的内存才能将这些数据存储下来。



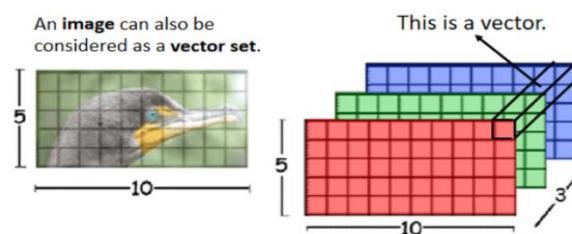
那么在语音注意力模型中，有一个改动叫做截断自注意力（Truncated Self-attention），它的意思是说，我们在做 Self-attention 的时候，不看一整句话，只看一小部分即可，而这个一小部分的窗口的设定，也是人为决定的。



当然这个范围的决定，也取决于你对这个问题的理解，比如我们在做语音辨识的时候，只需要前后一小段的资讯就可以进行判断。总之，使用截断注意力机制缩小了注意力模型考察的范围，从而加快运算的速度。

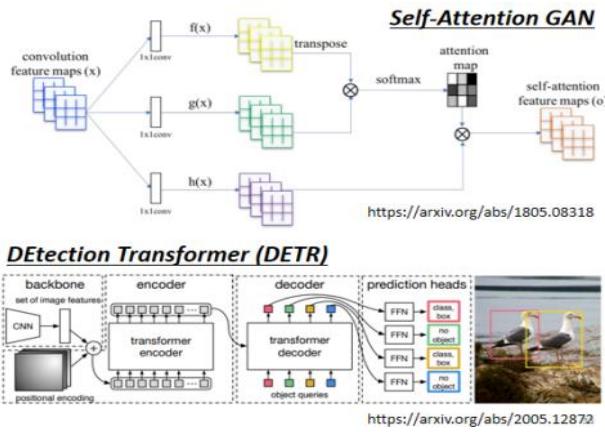
6.3.2 自注意力机制 v.s 卷积神经网络

当然，自注意力机制也可以用在影像上，你可能会觉得奇怪，因为在一开始我们提到自注意力机制模型时有一个前提，就是输入的是一个向量集合。



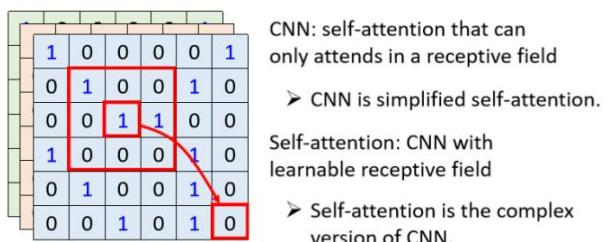
Source of image: https://www.researchgate.net/figure/Color-image-representation-and-RGB-matrix_fig15_282798184

那实际上，一张图片也可以看作一个向量集合。以上图为例，这是一张解析度为 5*10 的图片。在之前对图像的处理中，我们是将这一张图像拉长成一个大的张量，但是我们也可以将每一个位置的像素，看作是一个三维的向量。因此，整张图片就是一个 5*10 的向量集合。



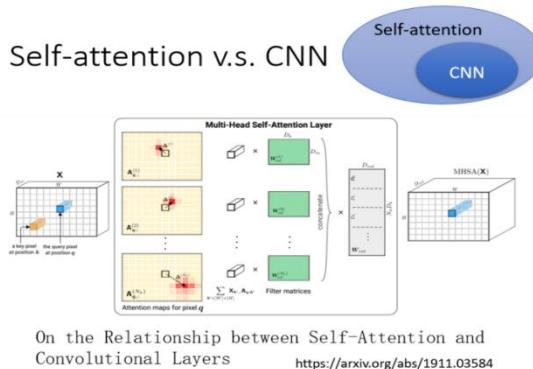
所以我们其实可以换一个角度，你完全可以用自注意力机制来处理一张图片，应用如上图所示。

实际上，自注意力机制与卷积神经网络是有一定关联的，假如我们今天是在用自注意力模型处理一张图片，如下图所示，假设红色框框是我们正在处理的一个像素，它可以生成一个查询向量，而其余地方的像素产生的是键向量。那在这里，我们也是需要进行内积的计算，这其实与卷积神经网络中的滤波器（filter）和感受野（receptive field）的思想是类似的。只不过自注意力模型考虑的是整张图片的信息，而滤波器只考虑其对应的感受野内的信息。



因此，我们可以说，卷积神经网络是一种简化版本的自注意力模型。对于自注意力模型来说，它的感受野的范围不再是人为划定的，而是通过机器自动学习出来的。

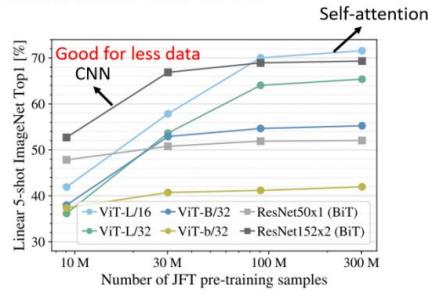
其实有一篇论文探究过二者的关系，名叫 *On the Relationship between Self-attention and Convolutional Layers*，在这篇论文中，它用数学的方式严谨阐述了二者之间的关系，即：CNN 是 Self-attention 的一个特例，假如设定合适的参数，二者的功能可以是一模一样的。



On the Relationship between Self-Attention and Convolutional Layers
<https://arxiv.org/abs/1911.03584>

换言之，Self-attention 可以看成更加灵活的 CNN，因此，Self-attention 需要更多的数据，否则就会更容易引起过拟合，实验对比如下：

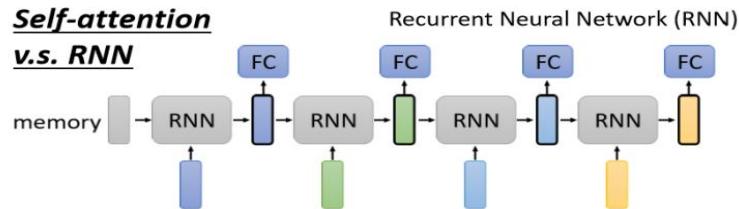
Self-attention v.s. CNN



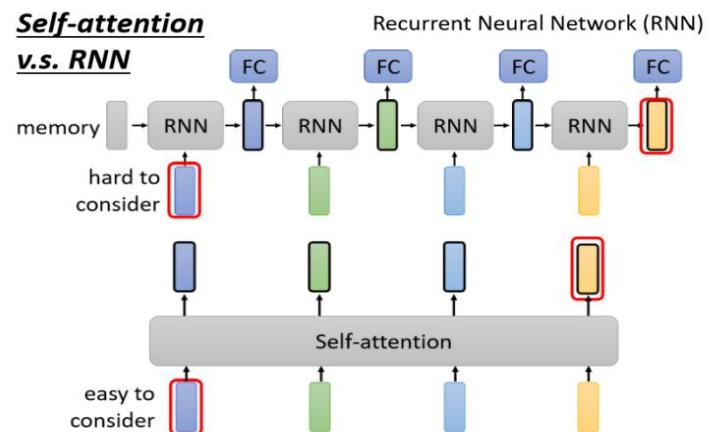
An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale
<https://arxiv.org/pdf/2010.11929.pdf>

6.3.3 自注意力机制 v.s. 循环神经网络

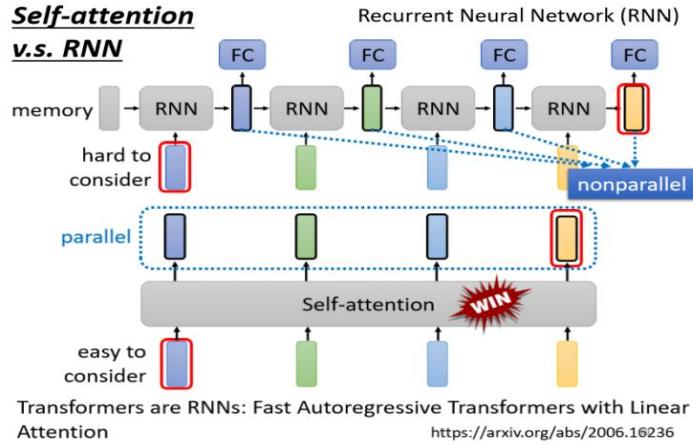
我们来比较一下自注意力跟循环神经网络（Recurrent Neural Network, RNN）。目前，循环神经网络的角色很大一部分都可以用自注意力来取代了。但循环神经网络跟自注意力一样，都是要处理输入是一个序列的状况。



循环神经网络和自注意力机制做的事情其实也非常像，它们的输入都是一个向量序列。但是二者有着明显的不同，RNN 只会考虑左（右）或是两边的向量（这取决于你的 RNN 是单向的还是双向的），但不管你的 RNN 是单向还是双向，它都没有办法考虑太远。而 Self-attention 中每一个向量考虑了整个输入序列。因此二者最本质的区别在于：RNN 必须把最左边的输入存入内存中然后一路带到最右边才能被最后的输出考虑，有这种序列的信息；而对于 Self-attention 的每一个向量对于所有的输入都是一视同仁的，产生这种“天涯若比邻”的效果。



此外，在处理上，RNN 不能并行化处理，而 self-attention 可以并行化输出。所以在运算速度上 Self-attention 会比 RNN 更有效率。



因此在今天，很多的应用都往往把 RNN 的架构，逐渐改成 Self-attention 的架构了。当然，Self-attention 同样可以用在图（Graph）上，这其实就衍生出了一个新的领域，叫做图神经网络（Graph Neural Network，GNN），我们将在下一章进行介绍。

6.4 序列到序列模型（Seq2Seq）及应用

Transformer 是一个序列到序列（Sequence-to-Sequence，Seq2Seq）的模型，我们先介绍下序列到序列模型。序列到序列模型输入和输出都是一个序列，输入与输出序列长度之间的关系有两种情况。第一种情况下，输入跟输出的长度一样；第二种情况下，机器决定输出的长度。

在本小节中，我们首先来介绍一下序列到序列模型的应用。

6.4.1 语音识别、机器翻译与语音翻译和合成

我们之前讲过输入为一个序列时，输出一共有三种可能。而在上一章讲自注意力模型的时候，我们只拿出了输入，输出序列长度相等的情况进行讨论。而对于序列到序列的问题，我们是不知道输出序列的长度的，这需要机器自己决定。

事实上，序列到序列模型的主要应用有以下三个：

1. 语音识别：输入是声音信号，输出是语音识别的结果，即输入的这段声音信号所对应的文字。输入的声音信号的长度是 T ，但是我们无法根据 T 得到输出序列的长度 N ，而输出序列长度是根据机器听这段声音信号的内容所决定的，从而决定输出的语音识别结果。



2. 机器翻译：机器输入一个语言的句子，输出另外一个语言的句子。输入句子的长度是

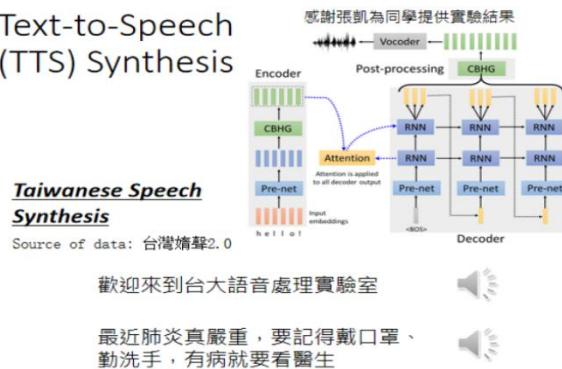
N , 输出句子的长度是 N' , 同样, 我们是无法控制输出句子的长度的, 这和需要翻译的语言有关。举个例子, 假设我们输入中文“机器学习”, 如果我们做的是中英翻译, 那输出为英文就是“Machine Learning”。



3. 语音翻译: 我们对机器说一句话, 比如“machine learning”, 机器直接把听到的英语的声音信号翻译成中文。事实上, 语音翻译就是语音识别和机器翻译的结合。那我们为什么还需要把它单独拿出来讲呢? 这是因为世界上很多语言是没有文字的, 无法做语音识别。因此需要对这些语言做语音翻译, 直接把它翻译成文字。



4. 语音合成 (Text-To-Speech, TTS), 某种程度上可以理解成语音翻译的逆运算。我们如果是一个模型, 输入台语声音, 输出中文的文字, 那就是语音辨识; 反过来输入文字, 输出声音讯号, 就是语音合成。

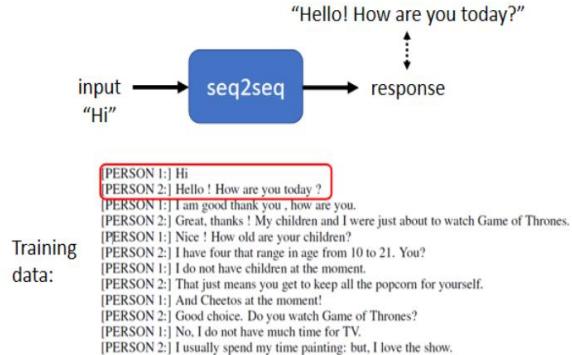


这边需要跟大家说明一下, 现在还没有真的做端到端 (end-to-end) 的模型, 以闽南语的语音合成为例, 其使用的模型还是分成两阶, 首先模型会先把中文的文字转成闽南语的拼音, 再把闽南语的拼音转成声音信号。从闽南语的拼音转成声音信号这一段, 是通过序列到序列模型实现的。

6. 4. 2 聊天机器人 (Chat Bot) 与问答任务 (QA)

除了语音以外, 文本也很广泛的使用了序列到序列模型。比如用序列到序列模型可用来训练一个聊天机器人。聊天机器人就是我们对它说一句话, 它要给出一个回应。因为聊天机器人的输入输出都是文字, 文字是一个向量序列, 所以可用序列到序列的模型来做一个聊天机器人。我们可以收集大量人的对话 (比如电视剧、电影的台词等等), 如图所示, 假设在

对话里面有出现，一个人说：“Hi”，另外一个人说：“Hello! How are you today?”。我们可以教机器，看到输入是“Hi”，输出就要跟“Hello! How are you today?”越接近越好。



事实上，序列到序列模型在自然语言处理（Natural Language Processing, NLP）的领域的应用很

广泛，而很多自然语言处理的任务都可以想成是问答（Question Answering, QA）的任务。



Question Answering，就是给机器读一段文字，然后你问机器一个问题，希望他可以给你一个正确的答案。以下是一些常见的 QA 任务的应用：

1. 翻译：机器读的文章是一个英语句子，问题是这个句子的德文翻译是什么？输出的答案就是德文。
2. 自动做摘要：给机器读一篇长的文章，让它把长的文章的重点找出来，即给机器一段文字，问题是这段文字的摘要是什么。
3. 情感分析（sentiment analysis）：机器要自动判断一个句子是正面的还是负面的。如果把情感分析看成是问答的问题，问题是给定句子是正面还是负面的，希望机器给出答案。



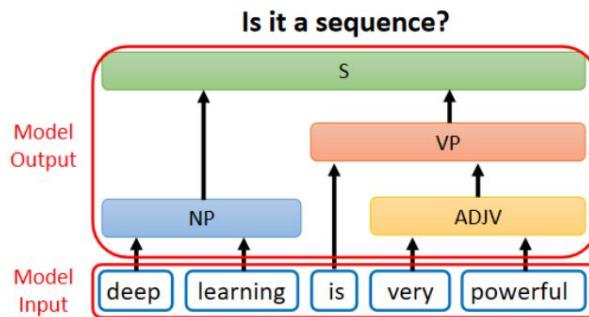
因此，各式各样的自然语言处理的问题往往都可以看作是问答的问题，而问答的问题可以用序列到序列模型来解。

但需要注意的是，往往对于 NLP 的任务，针对各种不同的任务定制的模型往往比只用序列到序列模型的模型更好。举例来说，谷歌 Pixel 4 手机用于语音识别的模型不是序列到序列模型，而是另一种基于 RNN 的模型模型，这种模型是为了语音的某些特性所设计的，表现更好。

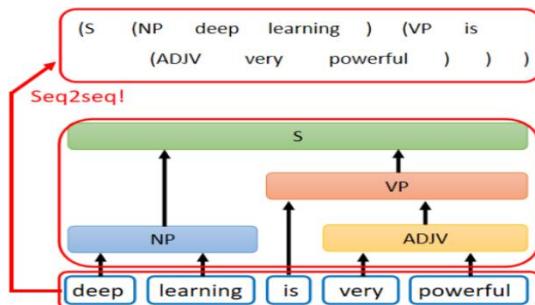
6.4.3 Seq2Seq 的一些新奇应用

Seq2Seq 这个模型的强大之处在于，有些看起来你认为和这个模型八竿子打不着的事情，我们也可以讲它转换为 Seq2Seq 来做。

第一个例子是句法分析 (syntactic parsing)，如下图所示，给机器一段文字：比如“deep learning is very powerful”，机器要产生一个句法的分析树，即句法树 (syntactic tree)。通过句法树告诉我们 deep 加 learning 合起来是一个名词短语，very 加 powerful 合起来是一个形容词短语，形容词短语加 is 以后会变成一个动词短语，动词短语加名词短语合起来是一个句子。



在句法分析的任务中，输入是一段文字，输出是一个树状的结构，而一个树状的结构可以看成一个序列，该序列代表了这个树的结构，如下图所示。把树的结构转成一个序列以后，我们就可以用序列到序列模型来做句法分析，具体可参考论文“Grammar as a Foreign Language”。



另一个意想不到的应用是多标签分类 (multi-label classification) 任务。我们需要区别一下 Multi-class classification 和 Multi-label classification，Multi-class 只会输出一个得分最高的类别，而 Multi-label 中，一个对象可以被分到多个类别中，如下图所示：

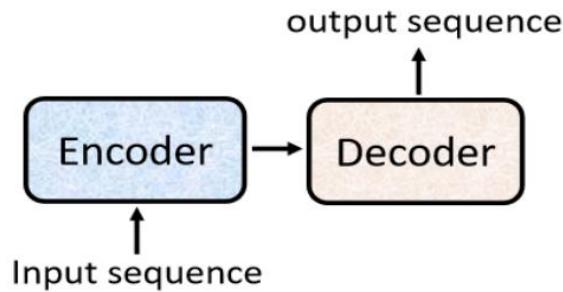


那么，对于这种输出是由机器决定的任务，我们统统可以用序列到序列模型硬做，类似

地,对于目标检测(Object Detection)的问题,我们也可以这样处理,处理方法参见“End-to-End Object Detection with Transformers”这篇论文。

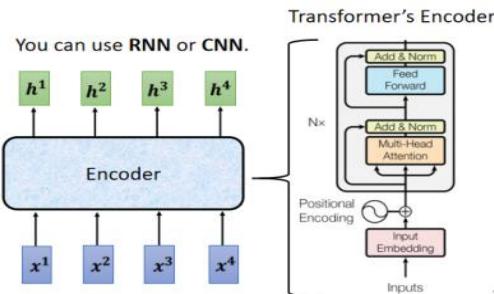
6.5 自注意力机制经典模型——Transformer

在前面的介绍中,我们通过各种各样的应用了解到了 Seq2Seq 的强大之处。在本节中,我们来介绍一下 Seq2Seq 的架构,而在序列到序列模型中,现在最典型的就是 Transformer,其有一个编码器架构和一个解码器架构,如下图所示:



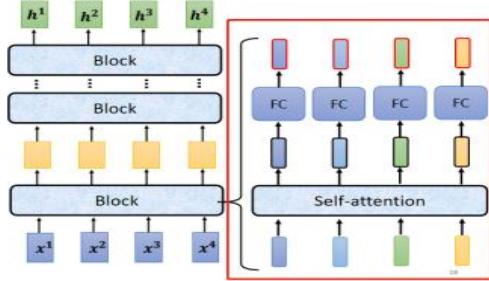
6.5.1 Transformer 编码器 (Encoder)

首先,我们来介绍一下 Transformer 中的编码器 (Encoder) 部分。Encoder 要做的事情,就是给一排向量,输出另外一排向量。事实上,自注意力、循环神经网络、卷积神经网络都能输入一排向量,输出一排向量。Transformer 的编码器使用的是自注意力,输入一排向量,输出另外一个同样长度的向量。



编码器里面会分成很多的块 (block),每一个块都是输入一排向量,输出一排向量。输入一排向量到第一个块,第一个块输出另外一排向量,以此类推,最后一个块会输出最终的向量序列。

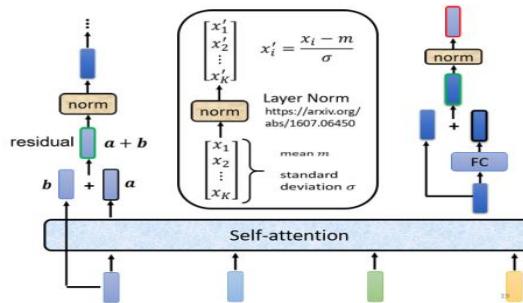
需要注意的是,Transformer 的块 (block) 并不是神经网络中的层 (layer)。事实上,一个编码器的块所做的事情是好几层神经元所做的:首先需要输入一排向量,记为 V^{in} ,随后做一次自注意力机制计算,输出另外一排向量,记为 V^{out1} 。随后,我们将 V^{out1} 丢入一层全连接层中得到新的输出: V^{out2} ,而这个输出就是整个编码器的输出,流程如下图所示:



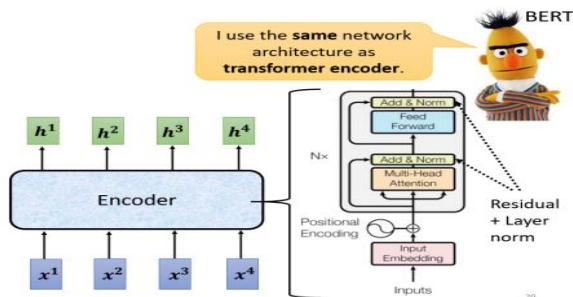
事实上在原来的 Transformer 里面做的事情是更复杂的。Transformer 里面加入了残差连接（residual connection）的设计，这是为了改善模型的梯度流（gradient flow），我们需要将 v_{out1} 中的向量加上 v_{in} 中的向量得到新的输出。在得到残差的结果以后，再做层归一化（layer normalization）。层归一化比批量归一化（batch normalization）更简单，不需要考虑批量（batch）的信息，而批量归一化需要考虑批量的信息。层归一化输入一个向量，输出另外一个向量。层归一化会计算输入向量的均值（mean）和标准差（standard deviation），公式如下：

$$x_i' = \frac{x_i - m}{\sigma}$$

而在接下来一层的全连接层中，同样也有类似的残差连接架构，那我们最终的输出是在全连接层+残差连接得到的输出的基础上，再进行一次层归一化的结果，如下图所示：



综上所述，下图给出了一个 Transformer 的编码器结构，图中的 $N \times$ 代表重复 N 次。当然，在实际的编码器架构中，输入向量还需要加入一个位置编码（positional encoding），这其实是因为如果只用自注意力，没有未知的信息，所以需要加上位置信息。



这个复杂的块，在后续一个非常重要的模型：BERT 中还会再次使用。当然，Transformer 的编码器其实不一定要这样设计，论文“On Layer Normalization in the Transformer Architecture”提出了另一种设计，结果比原始的 Transformer 要好。原始的 Transformer 的架构并不是一个最优的设计，我们永远可以思考看看有没有更好的设计方式。

那么，为什么 Transformer 中做的是 Layer Normalization 而不是 Batch Normalization 呢？在 CNN 中，我们经常使用 Batch Normalization，但是对 Transformer 这种需要处理序列类型

的数据而言，Layer Normalization 可能会更有效。

- On Layer Normalization in the Transformer Architecture
- <https://arxiv.org/abs/2002.04745>

- PowerNorm: Rethinking Batch Normalization in Transformers
- <https://arxiv.org/abs/2003.07845>



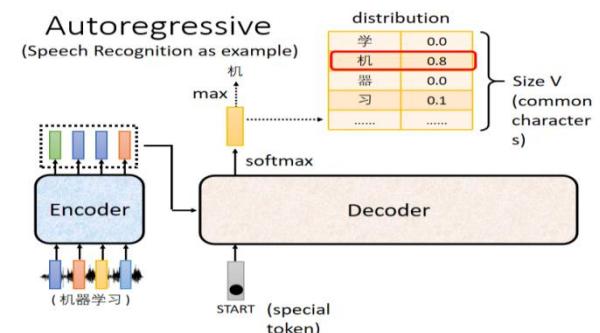
论文“PowerNorm: Rethinking Batch Normalization in Transformers”解释了在 Transformers 里面批量归一化不如层归一化的原因，并提出能量归一化（power normalization）。能量归一化跟层归一化性能差不多，甚至好一点。在这里，我问询了 GPT 砖家这个问题，并且阅读了这篇 paper，总结出了如下表格：

问题	BN	LN	解决——PN
批次相关性	依赖于批次内的统计数据	无相关性依赖	幂次方法动态调整计算方式，降低依赖，提高对变长序列适应能力
样本效率	小批次或数据稀缺表现不佳	独立进行标准化	引入可调整的幂次参数，提高对少量样本的效率，减少对大批次依赖
位置敏感性	不同位置序列，特征影响不同	计算方式一致	调整BN统计计算方式，尝试解决不同位置特征处理的一致性问题

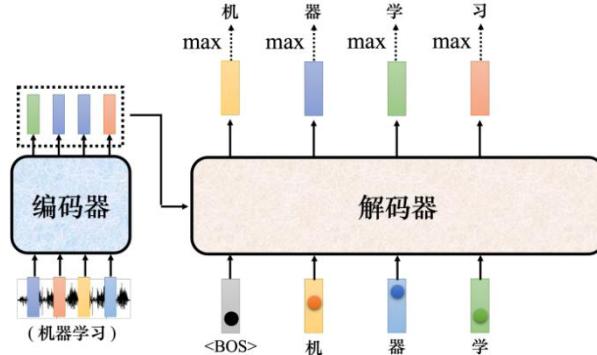
6.5.2 Transformer 解码器 (Decoder)

讲完 Transformer 的编码器部分，我们开始讨论解码器 (Decoder) 部分。对于 Transformer 的解码器，分为自回归和非自回归两种，虽然 Transformer 中最常用的还是自回归解码器 (Autoregressive Decoder)，但我们还是会讨论两种解码器分别进行讨论。

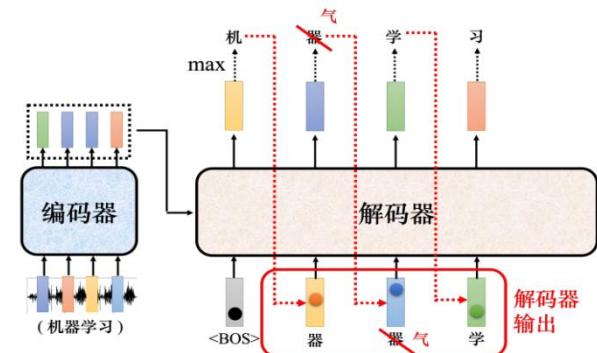
我们以语音识别的例子来说明什么是 Autoregressive。Autoregressive 是什么意思呢？就是指前一时刻的输出，作为下一时刻的输入。如下图所示，把一段声音（“机器学习”）输入编码器，输出会变成一排向量。解码器产生语音识别的结果，它把编码器的输出先“读”进去。要让解码器产生输出，首先要先给它一个代表开始的特殊符号 <BOS>，即 Begin Of Sequence，这是一个特殊的词元 (token)，它是在本来解码器中多加入的。在 NLP 的问题中，每一个 token 可以用一个独热向量表示，接下来解码器会“吐”出一个向量，该向量的长度跟词表的大小是一样的。在产生这个向量之前，跟做分类一样，通常会先进行一个 softmax 操作。这个向量里面的分数是一个分布，该向量里面的值全部加起来，总和是 1。这个向量会给每一个中文字一个分，分数最高的中文字就是最终的输出。“机”的分数最高，所以“机”就当做是解码器的第一个输出。



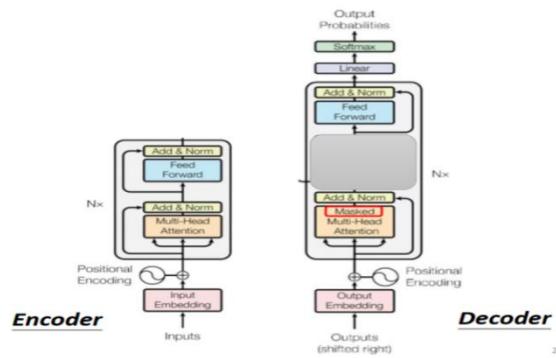
接下来把“机”当成解码器新的输入。根据两个输入：`<BOS>`和“机”，输出一个蓝色的向量。蓝色的向量里面会给出每一个中文字的分数，假设“器”的分数最高，“器”就是输出。解码器接下来会拿“器”当作输入，其看到了`<BOS>`、“机”、“器”、“学”，可能就输出“学”。解码器看到`<BOS>`、“机”、“器”、“学”，它会输出一个向量。这个向量里面“习”的分数最高的，所以它就输出“习”。这个过程就反复地持续下去。



但是这样的解码器也会有自己的问题，由于它会把自己的输出当做接下来的输入，因此在产生一个句子的时候，它有可能看到错误的东西。如果解码器有语音识别的错误，它把机器的“器”识别错成天气的“气”，接下来解码器会根据错误的识别结果产生它想要产生的期待是正确的输出，这会造成误差传播(error propagation)的问题，一步错导致步步错，接下来可能无法再产生正确的词汇。

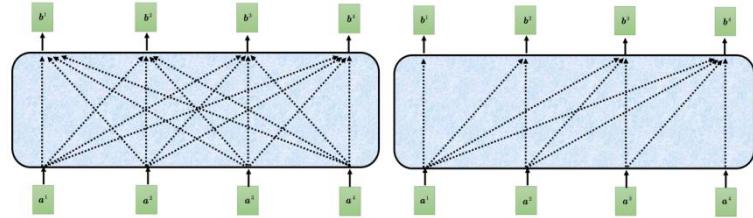


我们暂时忽略这个问题，先来看看解码器的架构，将编码器和解码器放在一起对比，如下图所示，除去灰色方框遮住的部分，解码器和编码器的结构非常相似。只不过解码器在最后需要进行一次 softmax 输出概率分布。

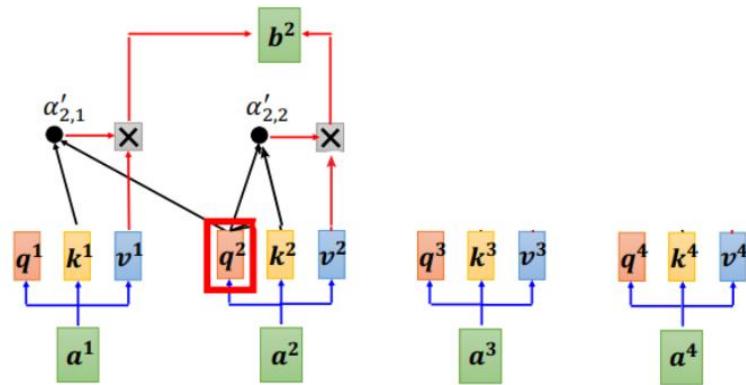


此外，解码器采用的是掩蔽自注意力（masked self-attention），通过一个掩码（mask）来阻止每个位置选择其后面的输入信息。掩蔽自注意力机制与普通的自注意力机制的区别在于，以之前我们在介绍自注意力机制时所举的例子为例：一般的自注意力机制中，对每一个

输出 \mathbf{b}^j , 都会考虑每一个输入 a^i , 而掩蔽自注意力机制中, 输出 \mathbf{b}^j 只会考虑输入 a^i , 其中 $i \leq j$, 如下图所示。

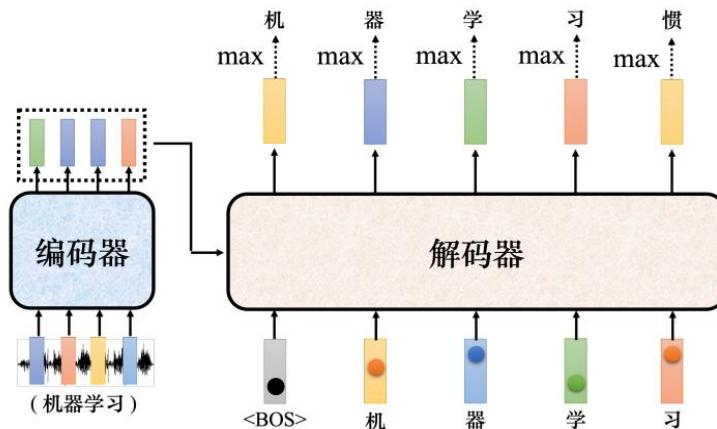


讲的更具体一点, 当我们需要计算 b^2 时, 只需要考虑 a^1 和 a^2 及其相关的 Q, K, V 矩阵进行运算即可。那为什么我们需要采用掩蔽注意力机制呢? 这件事情其实非常直觉, 因为我们的解码器是一个一个产生的: $a^1 \rightarrow a^2 \rightarrow a^3 \rightarrow a^4$, 因此我们只能考虑左边的东西, 无法考虑右边的东西, 这与编码器中一次性把 a^1 到 a^4 全部读入是不同的。

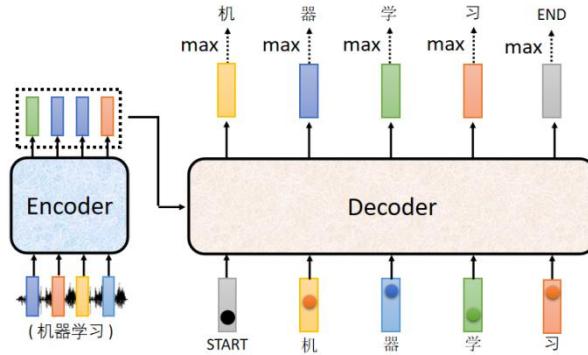


Why masked? Consider how does decoder work

了解了解码器的运作方式之后, 这里其实还有一个关键的问题: 实际应用中输入跟输出长度的关系是非常复杂的, 我们无法从输入序列的长度知道输出序列的长度, 因此解码器必须决定输出的序列的长度。给定一个输入序列, 机器可以自己学到输出序列的长度。但在目前的解码器运作的机制里面, 机器不知道什么时候应该停下来, 这和文字接龙的游戏很相像。



那怎么让这个过程停下来呢? 需要特别准备一个特别的符号 $<\text{EOS}>$ 。产生完“习”以后, 再把“习”当作解码器的输入以后, 解码器就要能够输出 $<\text{EOS}>$, 整个解码器产生序列的过程就结束了。



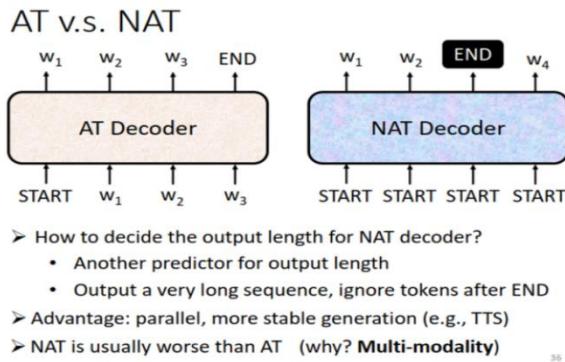
细心的你可能会发现，自回归解码器中还有一个缩放单元，事实上，缩放点乘注意力机制是 Transformer 模型的重要组件，公式如下：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

采用这样一个缩放单元的目的是，防止当 query 和 key 的维度 d_k 较大时，引起 softmax 输出分布在 0 和 1 两端，从而导致的梯度过小而无法训练的情况。在 Transformer 的原始论文中，作者们的解释如下：

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k , the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients⁴. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

讲完自回归解码器，我们来简单讲一下另一种解码器——非自回归解码器（Non-autoregressive Decoder）。为方便起见，在深度学习中，自回归解码器通常被称为 AT，而非自回归解码器通常被称为 NAT。



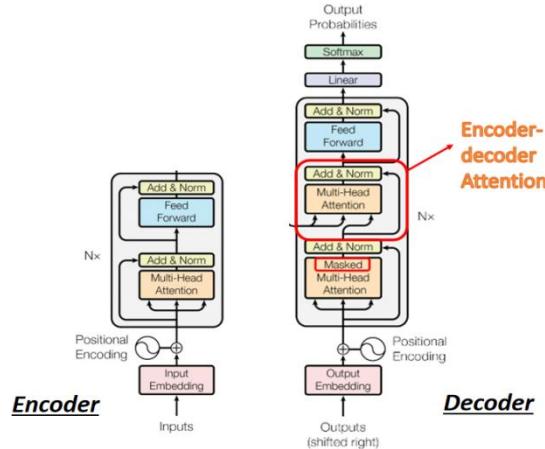
如上图所示，我们可以对比 AT (autoregressive) 与 NAT (Non-autoregressive) 的结构示意图。可以看到，与 AT 不同的是，NAT 并不使用之前时刻的输出，而是一次输入一组特殊的词元。那么，输入多长合适呢？有多种方法。比如，一种方法是把编码器的输出送入一个分类器，预测解码器输出序列的长度，进而也输入相同长度的特殊词元。另一种方法如下图所示，输入一个很长的序列，看输出什么时候出现“stop”词元，截取其之前的序列作为输出。

NAT 的第一个好处是可以支持并行化，因为 NAT 的句子产生与句子的长度无关，而对于 AT 来说，输出长度为 N 的句子就需要做 N 次解码，所以 NAT 的速度会更快。此外，NAT 能够控制它输出的长度，比如在语音合成 (TTS) 任务中，按前面提到的方法一，把编码器的

输出送入一个分类器并预测解码器输出序列的长度。通过改变这个分类器预测的长度，可以调整生成语音的语速。例如，设置输出序列长度 $\times 2$ ，语速就可以慢一倍。

6.5.3 编码器-解码器注意力

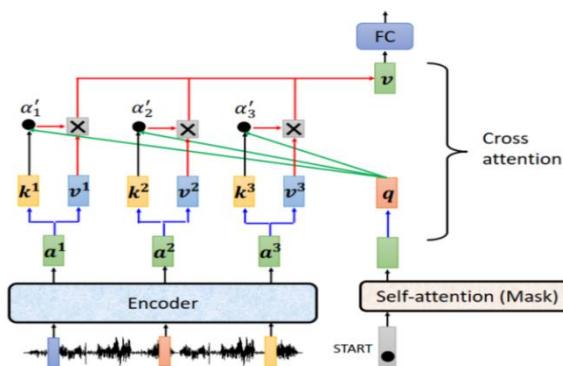
我们刚才在讲解码器架构时，有一个灰色方框当时被我们当成一个“黑盒”来看待。而这个“黑盒”其实就是连接编码器和解码器的桥梁，我们将其称为编码器-解码器注意力（Encoder-decoder Attention）。



编码器-解码器注意力的运作方法是这样的，如下图所示，编码器输入一排向量，输出 a^1, a^2, a^3 ，接下来解码器会先解码`<bos>`，经过掩蔽自注意力得到一个向量，把这个向量乘上一个矩阵做一次 transform，得到一个 query vector: q ， a^1, a^2, a^3 也都分别产生键，记为: k^1, k^2, k^3 ，我们将 q 与 k^1, k^2, k^3 分别相乘得到注意力分数: $\alpha^1, \alpha^2, \alpha^3$ ，接下来做一次 softmax 得到: $\alpha'_1, \alpha'_2, \alpha'_3$ ，接下来，我们需要利用如下公式计算 v ，它其实是一个加权和的计算：

$$v = \alpha'_1 \cdot v_1 + \alpha'_2 \cdot v_2 + \alpha'_3 \cdot v_3$$

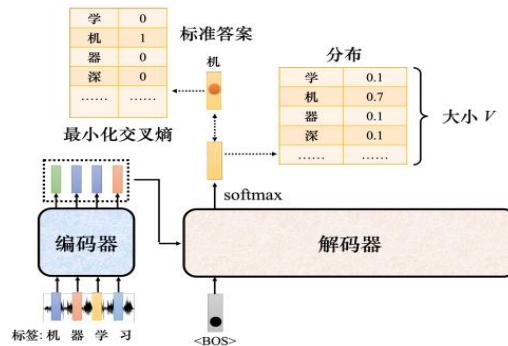
v 接下来会“丢”给全连接网络，这个步骤 q 来自于解码器， k 跟 v 来自于编码器，该步骤就叫做编码器-解码器注意力，所以解码器就是凭借着产生一个 q ，去编码器这边抽取信息出来，当做接下来的解码器的全连接网络（fully-connected network）的输入。



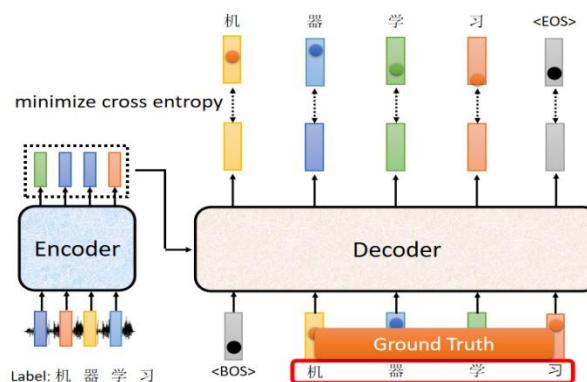
对于整个编码器-解码器注意力模型，解码器会不断地将产生的内容作为新的输出，从而进行上述类似的 q, k, v 的运算。编码器和解码器都有很多层，在原始论文中解码器是拿编码器最后一层的输出。但不一定要这样，有很多人尝试了各式各样的 Cross Attention 的方式，这也是一个值得研究的话题。

6.5.4 Transformer 的训练

以上讲的是 Transformer 训练完之后的推理 (inference, 相当于测试 testing) 相关的东西，接下来，我们需要了解一下具体的训练过程，以语音辨识为例，我们需要收集一大堆声音讯号，并需要相关的数据处理人员进行辨识。如下图所示，Transformer 应该要学到听到“机器学习”的声音信号，它的输出就是“机器学习”这四个中文字。把<BOS>丢给编码器的时候，其第一个输出应该要跟“机”越接近越好。而解码器的输出是一个概率的分布，这个概率分布跟“机”这个字所表示成的独热向量越接近越好。因此我们会去计算标准答案 (ground truth) 跟分布之间的交叉熵，希望值越小越好。每一次解码器在产生一个中文字的时候做了一次类似分类的问题。假设中文的词汇数量有 V 个，分类的类别就是 V 个。



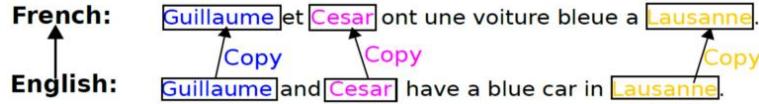
所以实际训练时，输出应该是“机器学习”。解码器的四次输出应该分别就是“机”、“器”、“学”、“习”这四个中文字的独热向量，输出跟这四个字的独热向量越接近越好。在训练的时候，每一个输出跟其对应的正确答案都有一个交叉熵。当然，还需要一个<EOS>作为训练结束的标识符。实际上，在训练的时候，告诉解码器在已经有<BOS>、“机”的情况下，要输出“器”，有<BOS>、“机”、“器”的情况下输出“学”，有 <BOS>、“机”、“器”、“学”的情况下输出“习”，有<BOS>、“机”、“器”、“学”、“习”的情况下，输出<EOS>。在解码器训练的时候，在输入的时候给它正确的答案，这称为教师强制 (teacher forcing)。



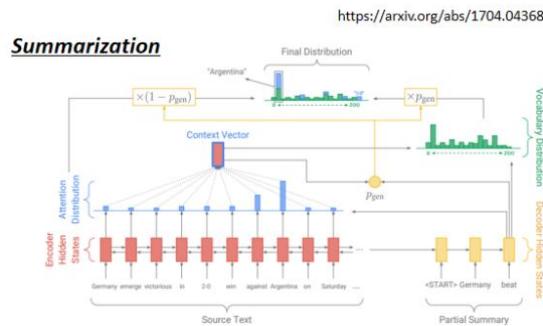
在本章最后，我们将简单介绍一下训练序列到序列模型的一些技巧。

第一个技巧是复制机制 (copy mechanism)。对很多任务而言，解码器没有必要自己创造输出，它可以从输入的东西里面复制一些东西。以聊天机器人为例，用户对机器说：“你好，我是库洛洛”。机器应该回答：“库洛洛你好，很高兴认识你”。事实上“库洛洛”对机器来说可能很难在训练数据里面出现，所以它不太可能正确地产生输出。但是假设我们不

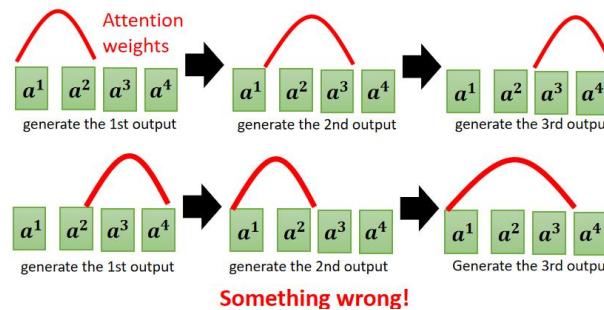
要求机器产生“库洛洛”，而是看到输入的时候说“我是某某某”，就直接把“某某某”复制出来，说“某某某你好”。这种机器的训练会比较容易得到正确的结果，所以对话任务可能需要复制机制。机器不需要从头去创造这一段文字，只需从用户的输入去复制一些词汇当做输出即可。



在做摘要的时候，我们可能更需要复制的技巧。做摘要需要搜集大量的文章，每一篇文章都有人写的摘要，训练一个序列到序列的模型就结束了。要训练机器产生合理的句子，通常需要百万篇文章，这些文章都要有人标的摘要。在做摘要的时候，很多的词汇就是直接从原来的文章里面复制出来的，所以对摘要任务而言，从文章里面直接复制一些信息出来是一个很关键的能力，最早有从输入复制东西的能力的模型叫做指针网络（pointer network），后来还有一个变形叫做复制网络（copy network）。

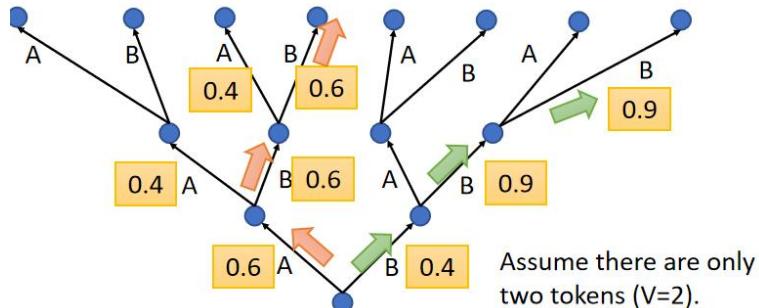


第二个技巧是引导注意力（Guided Attention），它是让注意力机制的计算按照一定顺序来进行。比如在做语音合成时，注意力机制的计算应该从左向右推进，如下图中前三幅图所示。如果注意力机制的计算顺序错乱，如下图中后三幅图所示，那就说明出了错误。因此，如果我们对这个问题本身就已经有理解，不如就直接把这个限制放进训练里。



第三个技巧是束搜索（Beam Search），这其实是一个最优路径的问题。每次解码器输出一个变量，假设输出词汇库只有 A, B 两个词汇。每一次都选择最大概率的作为输出，如下图中红色路径所示，这就是 Greedy Decoding。同时，解码器的每个输出又是下一时刻输入，如果我们从整个序列的角度考虑，可能第一次不选最大概率，后面的输出概率（把握）都很大，整体更佳，如下图中绿色路径所示。

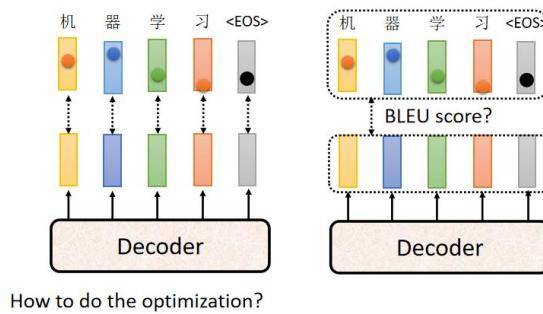
The red path is **Greedy Decoding**.
 The green path is the best one.
 Not possible to check all the paths ... → Beam Search



那么，我们应该如何找到这条绿色的路径呢？最直觉的办法就是暴力穷举搜索，但实际上并没有办法穷举所有可能的路径，因为每一个转折点的选择太多了，走两三步以后，就会无法穷举。

而束搜索就是一个不错的优化方法，其思想是用比较有效的方法找一个近似解。这种优化方法可以说是“时灵时不灵”的，假设任务的答案非常明确，比如语音识别，通常束搜索就会比较有帮助。但如果任务需要机器发挥一点创造力，束搜索比较没有帮助。

此外，我们还可以用强化学习“硬 train”我们的任务。通常在 Transformer 相关的项目中，评估的标准用的是 BLEU (BiLingual Evaluation Understudy) 分数。其做法是解码器先产生一个完整的句子，再去跟正确的答案一整句做比较，拿两个句子之间做比较算出 BLEU 分数。在训练的时候，每一个词汇是分开考虑的，最小化的是交叉熵，但是最小化交叉熵与最大化 BLEU 分数并没有什么关系。而且在做验证的时候，我们挑选的是 BLEU 分数最高的模型，而不是交叉熵最低的模型。

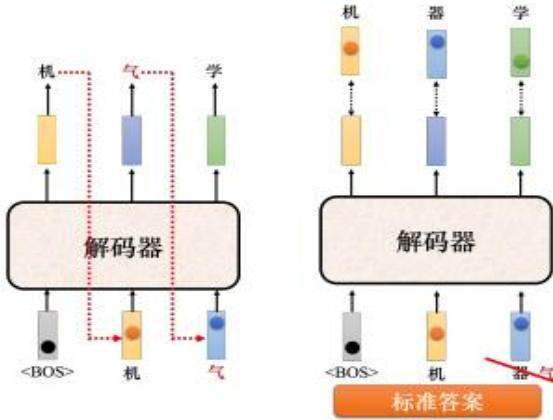


How to do the optimization?

When you don't know how to optimize, just use reinforcement learning (RL)! <https://arxiv.org/abs/1511.06732>

那你可能会说，为什么我们不能直接将 BLEU score 相关的东西当作损失函数呢？这是因为，BLEU score 本身很复杂，它是不能微分的。这边就教大家一个秘诀：当你在不知道如何求解优化问题时，用 RL “硬 train” 一发就完事了，我们可以将这个无法优化的损失函数当作强化学习的 Reward，把解码器当做是 Agent，这就把问题转换成了强化学习 (Reinforcement Learning)。我们将在后续章节对这类问题进行介绍，这里只是说明一下 Transformer 中的一个训练小技巧。

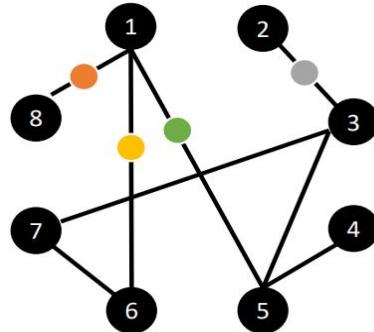
最后，我们可以来解决一下前文中所说的 error propagation 的问题。事实上，测试的时候，解码器看到的是自己的输出，因此它会看到一些错误的东西。但是在训练的时候，解码器看到的是完全正确的，这种不一致的现象叫做曝光偏差 (exposure bias)。



有一个可以思考的方向是：给解码器的输入加一些错误的东西，不要给解码器都是正确的答案，偶尔给它一些错的东西，它反而会学得更好，这一招叫做计划采样（scheduled sampling）。但是计划采样会伤害到 Transformer 的平行化的能力，所以 Transformer 的计划采样另有招数，其跟原来最早提在这个 LSTM 上被提出来的招数也不太一样。详见论文“Scheduled Sampling for Transformers”、“Parallel Scheduled Sampling”。

7 图神经网络

在上一讲我们曾提到，自注意力机制可以应用在图（Graph）上，将二者相结合就是图神经网络（Graph Neural Network, GNN）。在这个问题中，我们需要将序列转化为图结构，如下图所示。



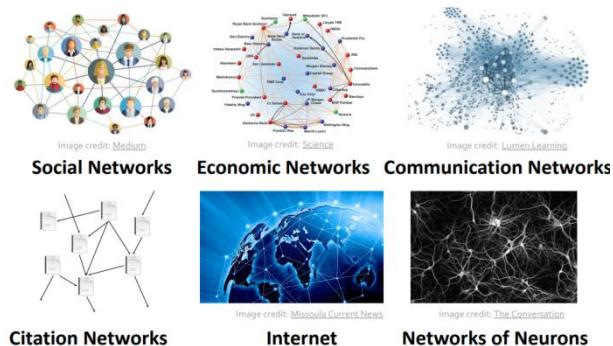
在本章中，我们将从为什么需要图结构作为切入点，分别讲解 GCN (Graph Convolutional Network)，GraphSAGE 以及 GAT 等图神经网络。当然鉴于我们的篇幅有限，并不能对图神经网络的专题进行太多深究，因此感兴趣的读者可以自行学习斯坦福大学的 CS224W (<https://snap.stanford.edu/class/cs224w-2023/>) 课程，本章的介绍也将部分参考这门课程的资料。

7.1 图神经网络基本介绍

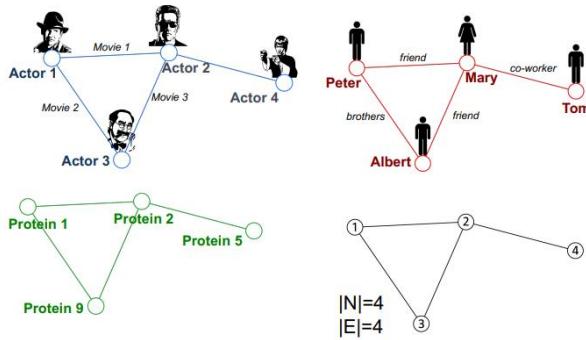
7.1.1 为什么需要图结构？

在了解图神经网络之前，我们首先来了解一下为什么需要“图”。从英译的角度来看，图神经网络中的“图”的英文是”Graph”而不是”Picture”或者”Image”。由此看来，图神经网络中的“图”是一种结构（和数据结构类似）而不是我们之前讲的图像。

事实上，图这种结构的好处在于，它是一种很好的描述领域知识的方式，我们可以用图这种结构来构建不同结点之间的关系，利用图进行建模的应用如下图所示：



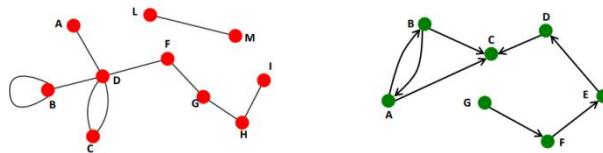
由此我们可以看出，图是一种解决关系问题的通用语言，在机器学习中，将问题抽象成图，可以实现利用同一种机器学习算法解决多个问题的功能。



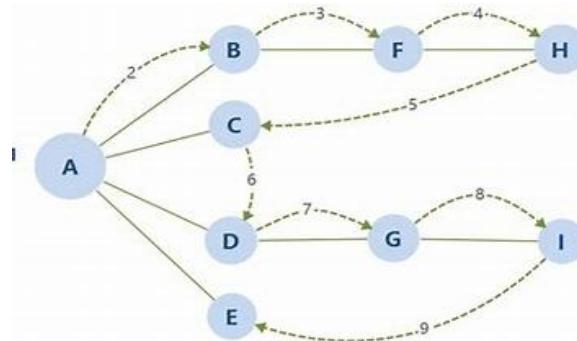
7.1.2 图结构的选择

在介绍图结构表示之前，我们首先来回顾一下图这种数据结构。一个图（Graph）是由若干顶点（Vertex, V）和边（Edge, E）组成的，因此，我们可以用顶点和边来表示一个图： $G = (V, E)$ 。

在实际任务中，图的建立是比较复杂的，我们需要考虑用什么东西作为图的顶点，用什么东西作为图的边。事实上，图的边作为顶点之间的关系，有不同的设计方法，比如我们可以将图的边设置成带方向的，这样的图被称为有向图（Directed Graph），它可以被用在社交媒体（比如推特）的关注中，因为社交媒体的关注并不一定是相互的；还可以将边设置为不带方向的，这样的图被称为（Undirected Graph），它可以被用于表示社交媒体的好友中，因为在社交媒体中，好友关系一定是相互的。



此外，对于这些关系，我们还可以给图的一些边设置一些权重（Weight），这样的图叫做带权图（Weighted Graph），它的典型应用是可以表示两个点之间的距离，如下图所示。

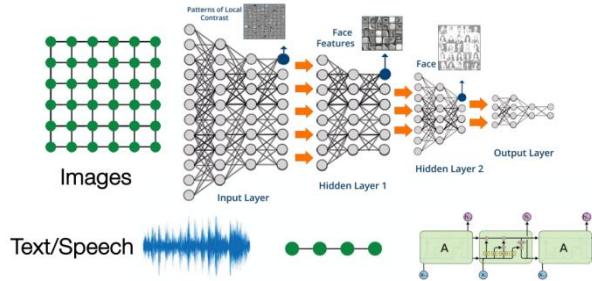


因此，在图相关的问题中，对某一领域或问题选择合适的网络表示方法会决定我们能不能成功使用网络，我们只需要根据自己对于问题的理解（或者说是 Domain Knowledge）进行建模即可。

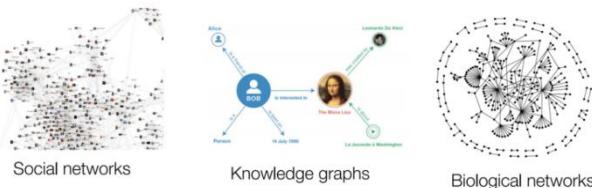
7.1.3 从图结构到图神经网络

本质上来说，世界上大部分数据都是网络拓扑结构，因此，将图结构应用到深度学习中是一个很重要的方向。事实上，随着机器学习，深度学习的发展，图像，语音，文本数据的

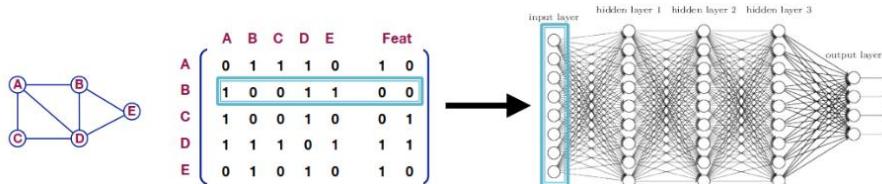
处理技术已经愈发成熟。然而这些数据都是很简单的序列数据，而神经网络是很擅长处理这种类型的数据的。



然而现实世界中并不是所有的事物都可以表示成一个序列或者一个网格，例如社交网络、知识图谱、复杂的文件系统等（如下图所示），也就是说很多事物都是非结构化的。相比于简单的文本和图像，这种网络类型的数据处理起来非常复杂，因此有人开始产生了将深度学习的技术运用到图这种结构中，因此促成了图神经网络的出现和发展。



那么和一般的神经网络相比，图神经网络有什么特别之处呢？相比于最基本的全连接神经网络。图神经网络中多了一个用来表示图的邻接矩阵 A ，一个最简单的运作方式是直接将邻接矩阵和特征合并在一起运用在神经网络上，如下图所示：



当然这种方法显然不是很好的，因为它需要 $O(|V|)$ 的参数，并且它不适用于不同大小的图，因此还有一些其它的处理方法，我们将信息的表达形式转化称为聚合（Aggregation），在图神经网络中，还有一些更好的聚合方式，我们将在接下来依次进行介绍。

7.2 图神经网络的经典模型

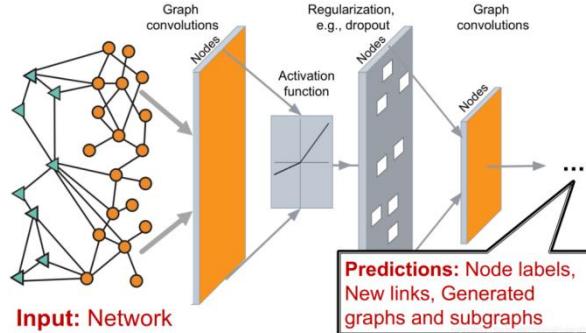
在上一节中，我们提到了一种最简单的信息聚合方式。事实上，信息聚合有两种主流方法：

1. 把 CNN 的方法泛化，考虑节点周边的邻居关系进行空间上的卷积。
2. 利用卷积本质是频域上的滤波这一特性，在频域上操作

基于上述两种技术，人们依次提出了一些图神经网络的模型：GCN, GraphSAGE 以及 GAT，我们将在本节中按顺序进行介绍。

7.2.1 图卷积神经网络 (GCN)

谈到图神经网络时，我们很自然地能想到 GCN。事实上，GCN 可谓是图神经网络的“开山之作”，由于其在各种任务和应用中具有简单性和有效性，因此 GCN 成为了目前非常流行的图神经网络架构。



对于一个图 $G = (V, E)$ ，输入 X 是一个 $N \times D$ 维的矩阵，表示每个节点的特征，同时还有邻接矩阵 A ，我们希望得到一个 $N \times F$ 的特征矩阵 Z 表示学习到的每个节点的特征表示，其中 F 是最终我们希望得到的维度。对于一个 L 层的神经网络来说，可以用如下数学公式表达：

$$H^{l+1} = f(H^l, A), l = 0, 1, \dots, L-1, H^0 = X, H^L = Z$$

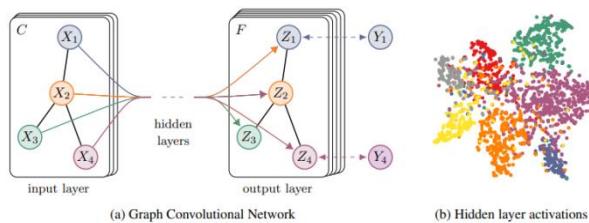
传统的图卷积做法可以看作对每一个节点都加上邻居节点的信息，我们的操作可以转化为：

$$f(H^l, A) = \sigma(AH^lW^l)$$

上式的每个节点都结合了相邻节点的信息，但是由于邻接矩阵的对角线元素全为 0，因此我们需要对其进行一些改进：首先我们需要加上一个单位矩阵 I ，即给邻接矩阵加上一个环，相当于结合了节点自身的信息；其次我们希望矩阵 A 的每一行的和均为 1，因此我们需要对该矩阵进行正则化，于是 GCN 的 Layer-wise Propagation Rule 如下：

$$H^{l+1} = f(H^l, A) = \sigma(D^{-\frac{1}{2}}\hat{A}D^{-\frac{1}{2}}H^lW^l), \hat{A} = A + I$$

GCN 的网络架构如下图所示：



那么 GCN 的效果如何呢？在 GCN 的原始论文 (<https://arxiv.org/pdf/1609.02907.pdf>) 中，作者将 GCN 放到节点分类任务上，分别在 Citeseer、Cora、Pubmed、NELL 等数据集上进行实验，相比于传统方法提升还是很显著的，这很有可能是得益于 GCN 善于编码图的结构信息，能够学习到更好的节点表示。

Dataset	Type	Nodes	Edges	Classes	Features	Label rate
Citeseer	Citation network	3,327	4,732	6	3,703	0.036
Cora	Citation network	2,708	5,429	7	1,433	0.052
Pubmed	Citation network	19,717	44,338	3	500	0.003
NELL	Knowledge graph	65,755	266,144	210	5,414	0.001

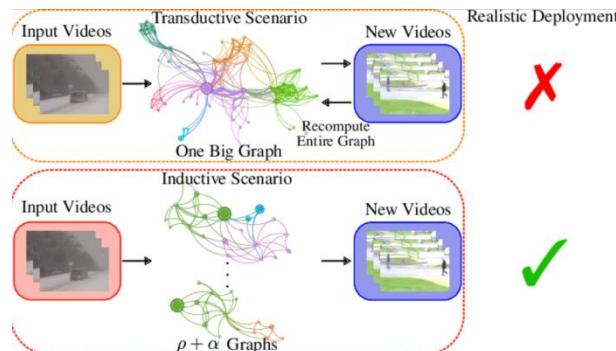
Method	Citeseer	Cora	Pubmed	NELL
ManiReg [3]	60.1	59.5	70.7	21.8
SemiEmb [28]	59.6	59.0	71.1	26.7
LP [32]	45.3	68.0	63.0	26.5
DeepWalk [22]	43.2	67.2	65.3	58.1
ICA [18]	69.1	75.1	73.9	23.1
Planetoid* [29]	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN (this paper)	70.3 (7s)	81.5 (4s)	79.0 (38s)	66.0 (48s)
GCN (rand. splits)	67.9 ± 0.5	80.1 ± 0.5	78.9 ± 0.7	58.4 ± 1.7

当然，GCN 也是有一些缺点的。首先，GCN 每一轮的迭代都要对全图的节点进行更新，当图的规模很大时，都需要将整张图放入内存，十分耗时，难以扩展到大规模网络。其次，GCN 每次学到的是每个节点的一个唯一确定的，如果要产生新的表示，需要从头开始训练，很难落地在需要快速生成未知节点表示的业务上。

7.2.2 从 GCN 到 GraphSAGE

为了解决 GCN 的这两个问题，后来人们提出了一种在大图上的归纳表示学习方法，也就是 GraphSAGE。在介绍 GraphSAGE 之前，我们先来了解一下归纳式学习(Inductive Learning)和直推式学习 (Transductive Learning)。

对于图这种数据结构，每一个节点可以通过边的关系利用其他节点的信息，这就导致一个问题，GCN 输入了整个图，训练节点收集邻居节点信息的时候，用到了测试和验证集的样本，这种学习方法叫直推式学习。



然而，我们所接触到的大部分机器学习问题都是刻意将样本集分为训练/验证/测试，并且训练的时候只用训练样本，这种方法叫归纳式学习。GraphSAGE 就是一种归纳式学习的方法，而事实上，当我们在图这种数据结构上运用归纳式学习的方法时，可以处理图中新来的节点，具体做法是利用已知节点的信息为未知节点生成嵌入 (embedding)。

在 GraphSAGE 这个归纳式学习的框架中，可以大致分为两个部分——采样和聚合，采样是指如何对邻居的个数进行采样，聚合是指拿到邻居节点的嵌入之后如何汇聚这些嵌入以更新自己的嵌入信息。GraphSAGE 学习的过程如下图所示：

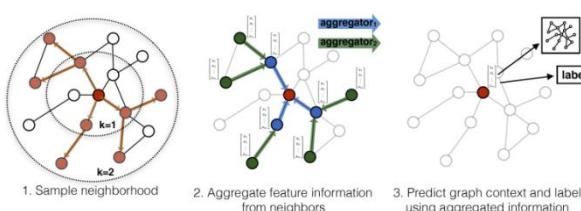


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

接下来，我们详细的说明一个训练好的 GraphSAGE 是如何给一个新的节点生成嵌入的（即一个前向传播的过程），其算法伪代码如下所示：

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K ; weight matrices $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$; non-linearity σ ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, \dots, K\}$; neighborhood function $\mathcal{N}: v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations \mathbf{z}_v for all $v \in \mathcal{V}$

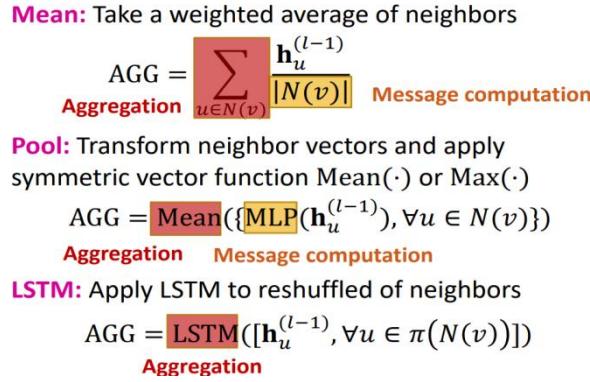
```

1  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$  ;
2 for  $k = 1 \dots K$  do
3   for  $v \in \mathcal{V}$  do
4     for  $u \in \mathcal{N}(v)$  do
5        $\mathbf{h}_{N(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ ;
6      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{N(v)}^k))$ 
7   end
8    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$ 
9 end
9  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$ 

```

那么，GraphSAGE 的采样是如何进行的呢？GraphSAGE 是采用定长抽样的方法，具体来说，定义需要的邻居个数 S ，然后采取“有放回”的重采样/负采样方法达到 S ，从而保证每个节点（采样后的）邻居个数一致，这样是为了把多个节点以及它们的邻居拼接成张量送到 GPU 中进行批训练。

而 GraphSAGE 的聚合方法主要有三种，分别是 Mean Aggregate，LSTM Aggregate 以及 Pooling Aggregate，如下图所示：



到此为止，整个模型的架构就讲完了，那么 GraphSAGE 是如何学习聚合器的参数以及权重矩阵 W 呢？这里需要分类讨论：如果任务是监督学习，可以使用每个节点的预测标签和真实标签的交叉熵作为损失函数；如果是在无监督的情况下，可以假设相邻的节点的嵌入表示尽可能相近，因此可以设计出如下的损失函数：

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})),$$

Nearby nodes → **Disparate nodes →**
Similar representation **Highly distinct representation**

那么 GraphSAGE 的实际效果如何呢？论文的作者同样在 Citation、Reddit、PPI 数据集上分别给出了无监督和完全有监督的结果，相比于传统方法提升还是很明显，如下所示：

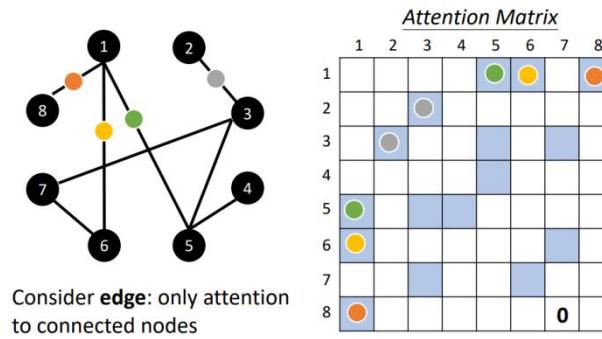
Name	Citation		Reddit		PPI	
	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	0.908	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	0.907	0.954	0.482	0.612
GraphSAGE-pool	0.798	0.839	0.892	0.948	0.502	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

当然，GraphSAGE 也有一些缺点，每个节点有很多邻居，GraphSAGE 的采样没有考虑

到不同邻居节点的重要性不同，而且聚合计算的时候邻居节点的重要性和当前节点也是不同的，于是便衍生出了我们需要讲的最后一个经典模型——GAT。

7.2.3 图注意力网络（GAT）

我们在上一章介绍自注意力机制时，就曾提到了自注意力机制可以应用在图结构中。事实上，这种神经网络被称为图注意力网络（GAT），其提出是为了解决 GNN 聚合邻居节点的时候没有考虑到不同的邻居节点重要性不同的问题，GAT 借鉴了 Transformer 的思想，引入掩码自注意力机制，在计算图中的每个节点的表示的时候，会根据邻居节点特征的不同来为其分配不同的权值。



GAT 使用了类似自注意力机制的方法来计算节点，其做法是：首先计算当前节点及其邻接节点的注意力分数，然后用这个分数乘上每个节点的特征，累加起来经过一个非线性映射，作为当前节点的特征。具体来说，对于一个输入的图，其图注意力层（Graph Attention Layer）如下所示：

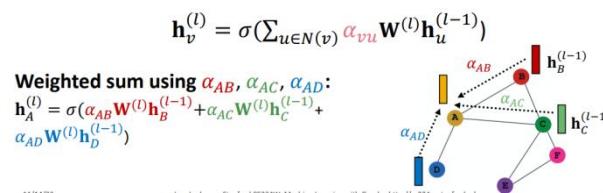
$$\mathbf{h} = \{\overrightarrow{\mathbf{h}_1}, \dots, \overrightarrow{\mathbf{h}_N}\}, \quad \overrightarrow{\mathbf{h}_i} \in \mathbb{R}^F$$

$$e_{ij} = \alpha(\overrightarrow{W\mathbf{h}_i}, \overrightarrow{W\mathbf{h}_j}), \quad \alpha_{ij} = \text{softmax}(e_{ij})$$

其中， α 采用了单层的前馈神经网络实现，计算过程如下：

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\vec{a}^T |W\vec{\mathbf{h}}_i| |W\vec{\mathbf{h}}_j|))}{\sum_{k \in N_i} \exp(\text{LeakyReLU}(\vec{a}^T |W\vec{\mathbf{h}}_i| |W\vec{\mathbf{h}}_k|))}$$

计算完注意力分数之后，我们需要根据计算好的注意力系数，对特征进行加权求和，从而得到某个节点聚合其邻居节点信息的新的表示，计算过程如下：



当然，上面的计算方法还是略微有些“单薄”，因此，为了提高模型的拟合能力，还引入了多头的注意力机制，即同时使用多个 W^k 计算注意力分数，然后将计算出的结果合并，如下图所示：

$$\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$$

$$\begin{aligned}\mathbf{h}_v^{(l)}[1] &= \sigma(\sum_{u \in N(v)} \alpha_{vu}^1 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}) \\ \mathbf{h}_v^{(l)}[2] &= \sigma(\sum_{u \in N(v)} \alpha_{vu}^2 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}) \\ \mathbf{h}_v^{(l)}[3] &= \sigma(\sum_{u \in N(v)} \alpha_{vu}^3 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})\end{aligned}$$

此外，由于 GAT 结构的特性，GAT 无需使用预先构建好的图，因此 GAT 既适用于直推式学习，又适用于归纳式学习，将 GAT 分别作用在直推式学习和归纳式学习的任务上，GAT 论文作者发现：无论是在直推式学习（下图左）还是在归纳式学习（下图右）的任务上，GAT 的效果都要优于传统方法的结果，效果如下图所示：

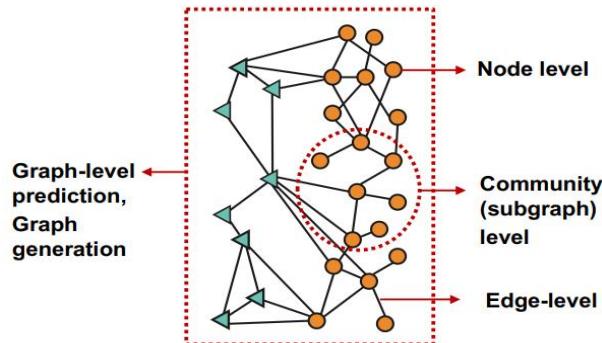
Transductive				Inductive	
Method	Cora	Citeseer	Pubmed	Method	PPI
MLP	55.1%	46.5%	71.4%	Random	0.396
ManiReg (Belkin et al., 2006)	59.5%	60.1%	70.7%	MLP	0.422
SemiEmb (Weston et al., 2012)	59.0%	59.6%	71.7%	GraphSAGE-GCN (Hamilton et al., 2017)	0.500
LP (Zhu et al., 2003)	68.0%	45.3%	63.0%	GraphSAGE-mean (Hamilton et al., 2017)	0.598
DeepWalk (Perozzi et al., 2014)	67.2%	43.2%	65.3%	GraphSAGE-LSTM (Hamilton et al., 2017)	0.612
ICA (Lu & Getoor, 2003)	75.1%	69.1%	73.9%	GraphSAGE-pool (Hamilton et al., 2017)	0.600
Planetoid (Yang et al., 2016)	75.7%	64.7%	77.2%		
Chebyshev (Defferrard et al., 2016)	81.2%	69.8%	74.4%		
GCN (Kipf & Welling, 2017)	81.5%	70.3%	79.0%		
MoNet (Monti et al., 2016)	81.7 ± 0.5%	—	78.8 ± 0.3%	GraphSAGE*	0.768
GCN-64*	81.4 ± 0.5%	70.9 ± 0.5%	79.0 ± 0.3%	Const-GAT (ours)	0.934 ± 0.006
GAT (ours)	83.0 ± 0.7%	72.5 ± 0.7%	79.0 ± 0.3%	GAT (ours)	0.973 ± 0.002

到此为止，我们介绍完了 GNN 中最经典的几个模型 GCN、GraphSAGE、GAT，在本章最后，我们来简单讨论一下图机器学习的经典应用。

7.3 图机器学习的应用

我们在第一章就知道，一个机器学习的任务主要可以分为回归和分类两大任务，同理，在图机器学习中，也可以按照这两大类对任务进行分类。

然而，由于图结构的复杂性，人们又将图机器学习任务按照图的结构进行分类，即：节点级别（Node Level）任务，边级别（Edge Level）任务，社区/子图级别（Community/Subgraph Level）任务以及图级别（Graph Level）任务，如下图所示：

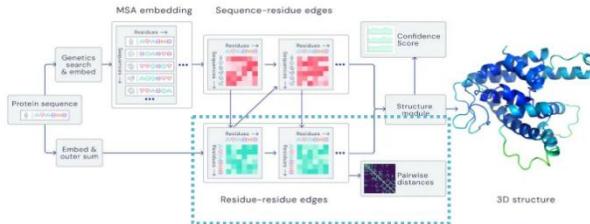


在本节中，我们将依次对这四种任务的典型应用进行介绍。

7.3.1 节点级别任务应用

解决蛋白质折叠问题——AlphaFold 是节点级别任务的一个经典应用，大致来说就是蛋白质由一系列氨基酸（氨基酸链 chains of amino acids 或 amino acid residues）结合而成，这些

氨基酸之间的交互使其形成不同的折叠方式，组成三维蛋白质结构（这个组合方式很复杂，但是一系列氨基酸交互之后形成的结构就是唯一的，所以理论上可以预测出最终结果）。学习任务就是输入一系列氨基酸，预测蛋白质的 3D 结构。

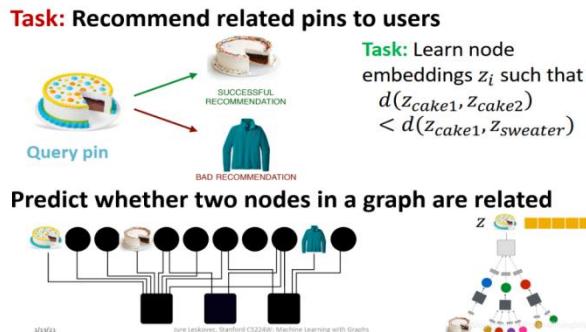


AlphaFold 将一个被折叠的蛋白质视作空间图（Spatial Graph），residue 视作节点，在相近的 residue 之间建立边，形成图结构，搭建深度学习模型，预测节点在空间中的位置（也就是蛋白质的三维结构）。

7.3.2 边级别任务应用

在这一小节中，我们将介绍两个边级别任务的典型应用。第一个应用是推荐系统，其任务是向用户推荐物品，当然在图机器学习中，推荐系统是“基于图”的，我们将这种推荐系统称为 PinSage。

PinSage 的做法是在网站 Pinterest 上给用户推荐相关图片。节点是图片和用户，组成如左下角所示的二分图。图中的 d 是嵌入向量之间的距离，即任务目标是使相似节点嵌入之间的距离比不相似节点嵌入之间的距离更小。



另外一个典型的应用是在同时吃 2 种药的情况下预测药的副作用。在日常生活中，很多人需要同时吃多种药来治疗多种病症。因此该任务的输入是一对药物，我们需要利用图机器学习来预测其有害副作用。预测结果如下图所示，其中有一部分已经有论文证明存在。

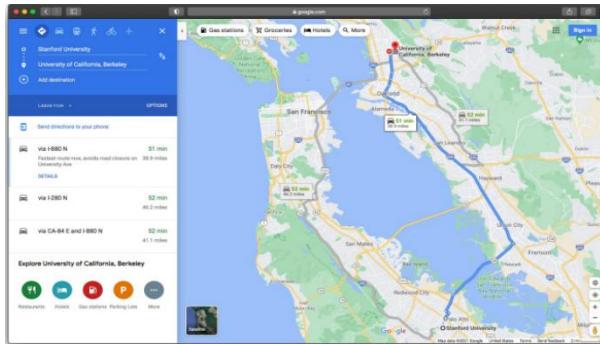
Rank	Drug c	Drug d	Side effect r	Evidence found
1	Pyrimethamine	Aliskiren	Sarcoma	Stage <i>et al.</i> 2015
2	Tigecycline	Bimatoprost	Autonomic neuropathy	
3	Omeprazole	Dacarbazine	Telangiectases	
4	Tolcapone	Pyrimethamine	Breast disorder	Bicker <i>et al.</i> 2017
5	Minoxidil	Paricalcitol	Cluster headache	
6	Omeprazole	Amoxicillin	Renal tubular acidosis	Russo <i>et al.</i> 2016
7	Anagrelide	Azelaic acid	Cerebral thrombosis	
8	Atorvastatin	Amlodipine	Muscle inflammation	Banakh <i>et al.</i> 2017
9	Aliskiren	Tioconazole	Breast inflammation	Parving <i>et al.</i> 2012
10	Estradiol	Nadolol	Endometriosis	

7.3.3 图级别任务

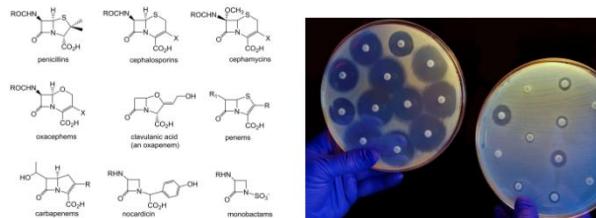
鉴于子图级别任务和图级别的任务都是和“图”相关的，我们在本章的最后将二者的一

些典型应用放在一起进行介绍。

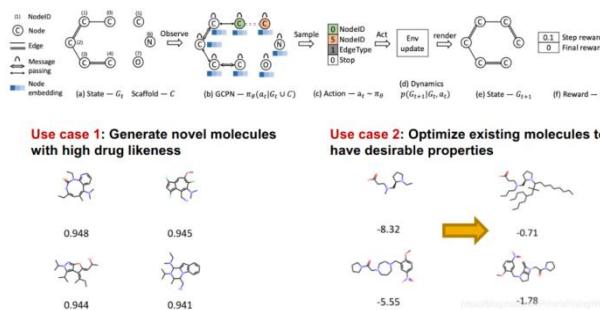
对于子图级别的任务，一个典型的应用是交通预测，我们可以将路段建模成图，在每个子图上建立预测模型。这一做法已经较为成功地应用在了在谷歌地图中。



而对于图级别的任务，可以将其分为三大类，第一类是图分类（Graph Classification），其典型的应用是药物发现，即用图神经网络的图分类任务来从一系列备选图（分子被表示为图，节点是原子，边是化学键）中预测最有可能是抗生素的分子。



第二类是图生成（Graph Generation），其典型应用是分子生成/优化（Molecule Generation / Optimization），如下图所示：



最后一类是图演化（Graph Evolution），它将整个物质表示为图（proximity graph），并利用图神经网络来预测粒子的下一步活动，例如，组成一个新位置或者新图。



以上就是本章对图神经网络的介绍，当然图神经网络还有一些其它结构（比如：Diff Pool, GIN）和其它应用，感兴趣的读者可以自行探究。

8 生成式模型 (Generative Model)

8.1 背景

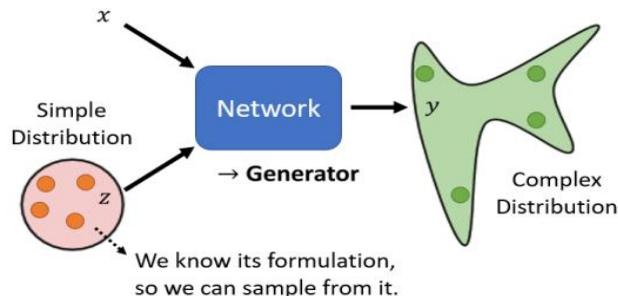
8.1.1 将网络当成生成器来用

在之前，我们所学的网络都是一个函数，输入一个 X 得到一个相对应的输出 Y ，我们已经学过了不同形式的输入，比如输入是一张图片，一个序列，而输出可以是一个数值，一个类别，也可以是一个序列，如下图左所示。在本章，我们要学习一个新的主题，即我们可以将神经网络当成一个生成器(generator)来使用。



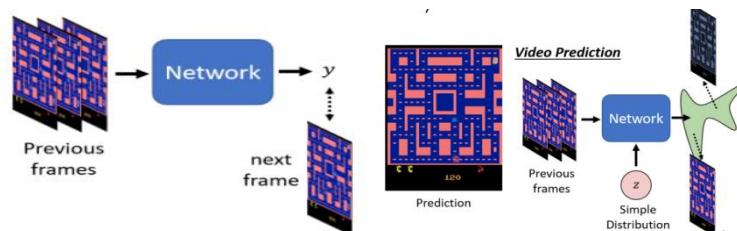
这种做法的特殊之处在于，我们需要在网络的输入中，加入一个随机变量，而它是从某一个概率分布采样出来的。因此，现在的网络的输出是由和同时作用来决定的，如上图右所示。

这个分布的形式可以是自己决定的，随着采样到的不同，的输出也不一样，,所以这个时候我们的网络输出不再是单一一个固定的东西，而变成了一个复杂的概率分布。这种可以输出一个概率的神经网络，我们就将其称为生成器（generator）。



8.1.2 为什么需要分布？

那么，为什么我们需要生成器的输出是一个分布呢？我们以一个例子进行说明，假如我们在做视频预测（video prediction）的任务，也就是给机器一段影片，然后让它预测接下来会发生什么。当我们用以前学过的监督学习的方法，训练方法为：给定一些之前的桢作为输入，输出下一帧的图像。

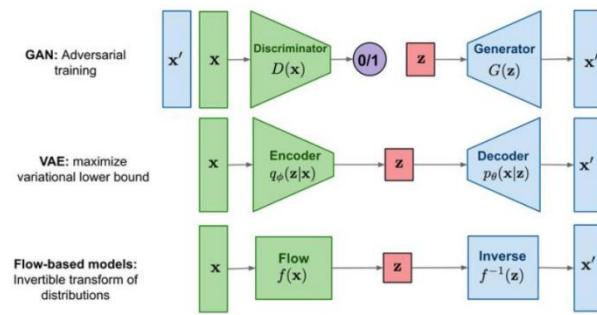


但我们发现这个网络的输出会出现不同的可能性，比如当精灵到了转角时，我们发现向左转时对的，向右转也是对的，所以这时候我们需要一个“两面讨好”的网络，它同时包含了向左转和向右转的可能性，这时就需要一个样本分布来建立这些可能性，如上图右所示。

那么，在什么时候我们需要用到分布呢？实际上，对于一些需要创造力的任务时我们会需要用到，因为当我们在创造一个东西时，每个人都有不同的答案，这个时候我们就需要生成式的模型（Generative Model）。

8.1.3 生成模型的分类

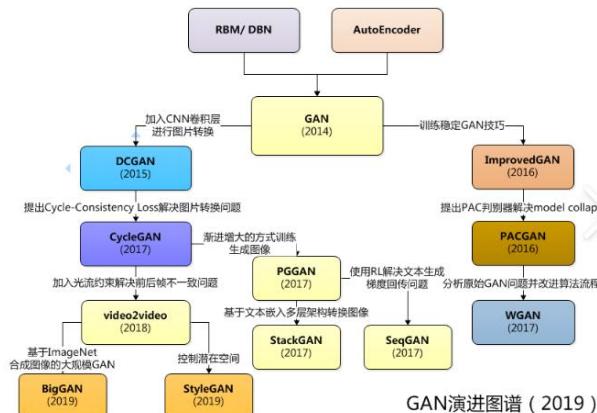
截至目前，学术界比较著名的有三大类生成模型：分别是生成对抗网络（Generative Adversarial Network, GAN），变分自编码器（Variational AutoEncoder）和基于流的生成式模型（Flow-based Generative Model），如下图所示：



对于 VAE 和 Flow-based 生成式模型，我们将在后续章节中进行介绍，因为它们其实是将无监督学习（Unsupervised Learning）运用到生成式模型中的应用，因此我们需要一些前置，即我们要在了解完什么是监督/自监督/无监督学习之后才能对这两个模型有一个更好的了解。我们在本章中主要对 GAN 及其常见的变体进行介绍。

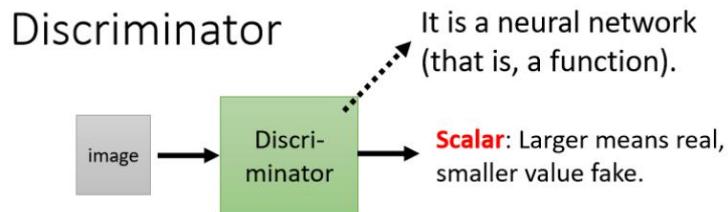
8.2 生成对抗网络（GAN）介绍

对于生成式模型，其中一个非常知名的，就是生成对抗网络（Generative Adversarial Network, GAN）。事实上，生成对抗网络有很多不同的变种，我们将在了解完基本的 GAN 之后，对于一些常见的变种进行介绍。



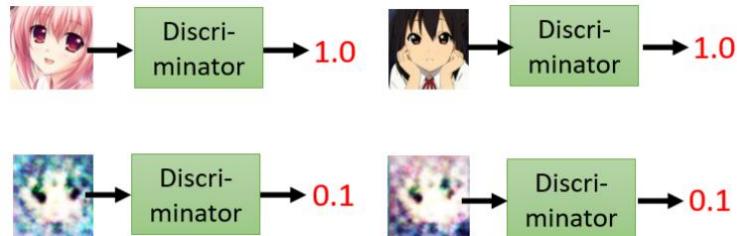
8.2.1 生成对抗网络的组成

生成对抗网络（GAN）的主要结构由一个生成器 G（Generator）和一个判别器 D（Discriminator）组成，那我们刚才大概了解过生成器是什么东西，我们接下来主要介绍一下判别器。



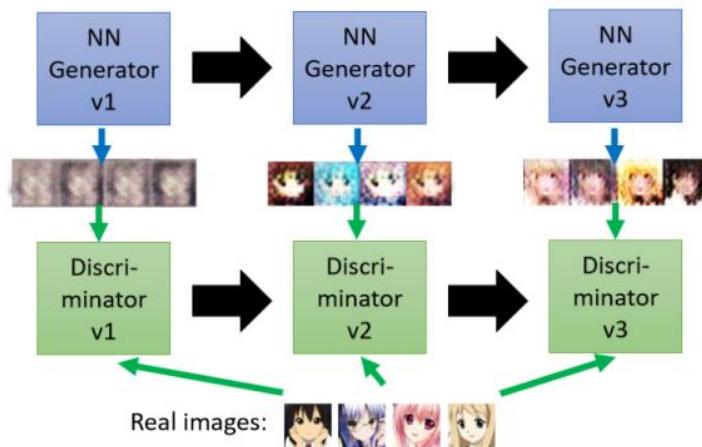
判别器会拿一张图片作为输入，它的输出是一个数值。这个判别器本身也是一个神经网络。输入一张图片，它的输出是一个标量，这个标量越大就代表着输入的这张图片越真实。

通常，我们将标量的值会定在 0-1 之间。而至于判别器的架构与生成器一样，都是可以人为设计的。



那么为什么同时需要生成器和判别器呢？实际上，GAN 中的 A 代表着 Adversarial，也就是对抗的意思。生成器和判别器其实可以看作一对“天敌”。以画二次元人物为例，生成器需要去画出二次元的人物，而判别器会根据生成的图片进行判分，学习过程是下面这个样子的。

第一代生成器的参数是随机的，这是因为它完全不知道该怎么画，而第一代的判别器会根据实际图像对生成的画像进行评分并反馈给生成器，而生成器会根据判别器的反馈进行参数更新。以此类推，具体流程如下图所示：



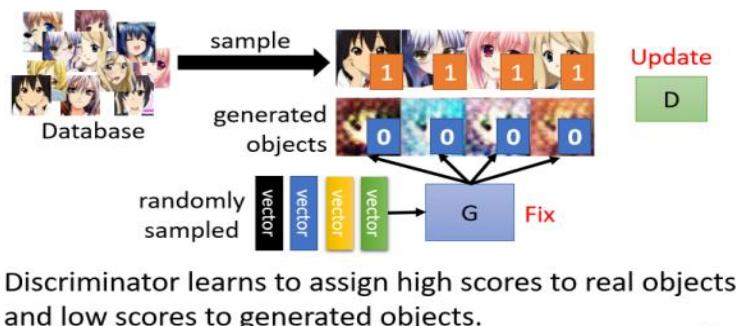
换言之，生成器和判别器可以说是“互相内卷”的关系，它们都是会根据互相的反馈不断改进自己完成进化的。

8.2.2 生成对抗网络的基本算法

以下就是正式来讲一下生成对抗网络的算法。生成器跟判别器就是两个网络，在训练前，你要先初始化它的参数，所以我们这边就假设生成器与判别器的参数都已经被初始化了。

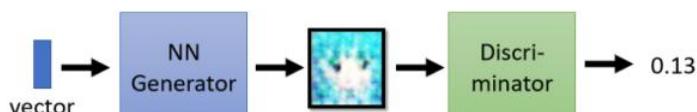
第一步是固定生成器 G，更新判别器 D。判别器训练的目标是要分辨真正的二次元人物与生成器产生出来的二次元人物之间的差异。讲得更具体一点，你可能会把这些真正的人物都标 1，生成器产生出来的图片都标 0。

Step 1: Fix generator G, and update discriminator D

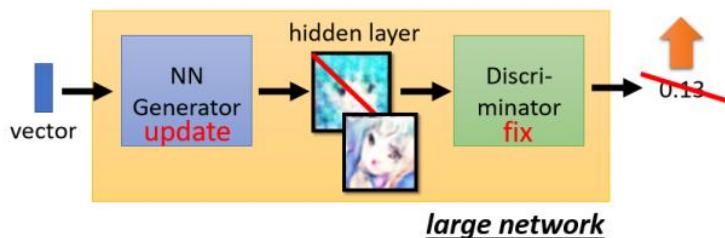


第二步是固定判别器，训练生成器。通俗易懂地来讲，我们需要让生成器想办法去骗过判别器。

Generator learns to “fool” the discriminator

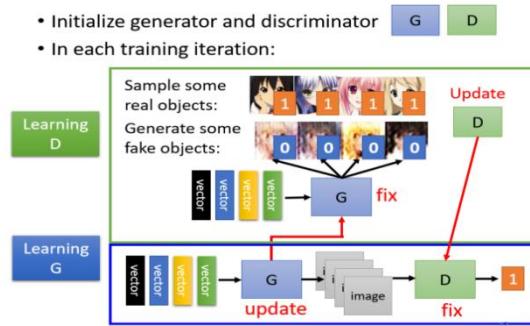


我们让生成器产生一些图片，在将这些图片输入进判别器，然后判别器就会给出这些图片一些分数。接下来我们把生成器和判别器串在一起视为一个巨大的网络，它的输入是一个向量，输出是一个分数，在这个网络中间隐藏层的输出可以看做是一张图片。



我们训练的目标是让输出越大越好，训练时依然会做梯度下降，只是要固定住判别器对应的维度，只是调整生成器对应的维度，调整过后输出会发生改变，生成器产生的新的输出

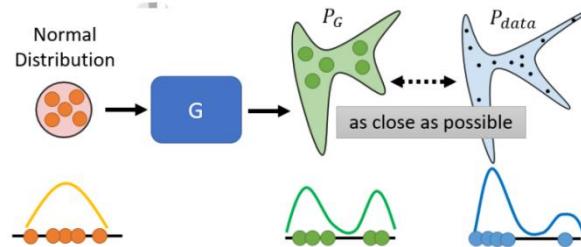
会让判别器给出高的分数。



那么接下来，我们需要反复训练，即将上述地两个步骤重复执行，并期待我们的生成器和判别器越做越好。

8.2.3 生成对抗网络的训练目标

我们知道，当我们确定一个神经网络时，需要决定损失函数，那么 GAN 的损失函数是什么样的呢？它该如何运作梯度下降算法呢？实际上，我们在 GAN 中需要最小化的是生成器 G 产生的分布： P_G 与实际数据的分布： P_{data} 之间的差异。换言之，我们需要让 P_G 和 P_{data} 越接近越好。



这个优化问题的表达式如下：

$$G^* = \arg \min_G \text{Div}(P_G, P_{data})$$

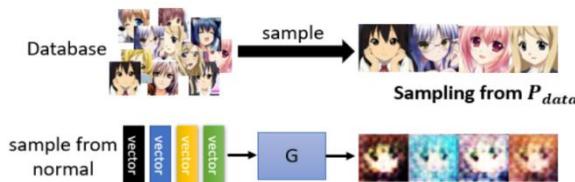
我们发现，损失函数是可以计算的，而散度（Divergence，比如：KL Divergence，JS Divergence）在数学上是一串十分复杂的积分式子，它们几乎是无法计算的。

而 GAN 的神奇之处在于，它可以突破我们不知道如何计算散度的限制。事实上，GAN 告诉我们，只要我们能从 P_G 和 P_{data} 中取样出东西，就有办法计算散度。

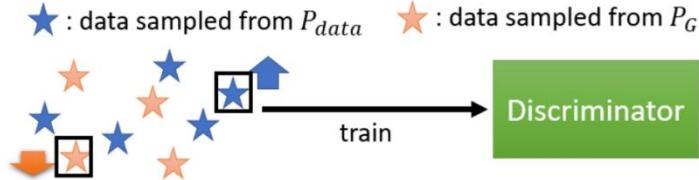
那么该如何进行取样呢？对于 P_{data} 而言，我们只需要从图库里随机取样即可得到，而对于 P_G 而言，我们需要把从标准概率分布取样出来的向量丢入生成器即可。

$$G^* = \arg \min_G \text{Div}(P_G, P_{data})$$

Although we do not know the distributions of P_G and P_{data} , we can sample from them.



接下来的一步是判别器的作用，它将完成在只知道采样的前提下，对散度进行计算的操作。事实上，判别器会按照如下的评分机制进行运作：当判别器看到一个很接近真实的数据时，就会打出一个高分，而看到一个几乎“肉眼可见”是生成出来的数据时，就会打出一个低分。



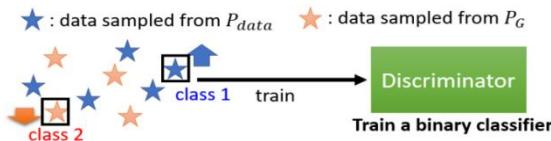
那么实际上，你可以把上述过程同样当作一个优化的问题，我们可以用这个判别器去最大化某一个函数，需要注意的是：当我们需要最大化某一个函数时，我们将其称为目标函数（Objective Function），当我们需要最小化某一个函数时，我们将其称为损失函数，对于这个问题，目标函数如下所示：

$$\text{Training: } D^* = \arg \max_D V(D, G)$$

Objective Function for D

$$V(G, D) = E_{y \sim P_{data}}[\log D(y)] + E_{y \sim P_G}[\log(1 - D(y))]$$

当然实际上，我们也没必要将这个目标函数写成上面这种形式。事实上，由于有人猜测GAN的目标函数的灵感是从二分类问题中得到的，我们不妨将其写成一个类似于交叉熵的形式。所以，当我们将这个目标函数两边同时取负号时，就等价于训练了一个二分类器。



8.3 GAN 的训练技巧及性能评估

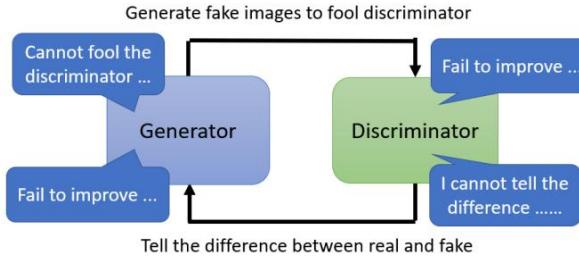
8.3.1 GAN 训练困难的本质原因

GAN 本身是很难训练的，常言道“*No Pain, no gain*”，有些研究生成式模型的人们将这句英文名言改编成了“*NO PAIN, NO GAN*”，由此可见，训练 GAN 是十分痛苦的。



GAN 训练困难的原因主要有两点，我们在这一小节主要聚焦于本质原因。事实上，不管

我们在训练 GAN 中用了些什么技巧（后面我们会有所提及），GAN 的训练并不会变得简单。原因在于：生成器总是在尝试“骗过”判别器，而判别器总是在告诉生成器真实数据和生成数据之间的差异。我们曾说过，生成器和判别器是“相互促进，相互内卷”的状态，那假如其中一方发生了什么问题停止了训练，另一方也会跟着变差。



到目前为止，我们已经见过很多不同类型的网络，我们没有办法保证在训练过程中，损失函数一定会一直下降，通常我们的做法是调整我们的超参数。但是在 GAN 中，生成器和判别器之间的“互动”是自动的，因此我们没有办法在每次训练判别器就调一次参。

因此我们只能祈祷在训练过程中，生成器和判别器能够一直做到“棋逢对手”，这样才能让我们的 GAN 继续训练下去。这就是 GAN 训练本质上的困难之处。

8.3.2 GAN 目标函数带来的困难及改进技巧

GAN 训练困难的第二个原因在于数学上的原因。我们进一步探究 GAN 的目标函数发现，这个目标函数其实是和概率统计中的散度 (Divergence) 有关的，而对于最基本的 GAN，所用的散度是 JS 散度。

为了探究为什么目标函数会给 GAN 的训练制造出困难，我们首先来了解一下 GAN 的数学原理。在这里，我们仅仅对原始 GAN 的理论进行探究，文章链接如下：[1406.2661.pdf \(arxiv.org\)](https://arxiv.org/pdf/1406.2661.pdf)。

Generative Adversarial Nets

Ian J. Goodfellow, Jean Pouget-Abadie^{*}, Mehdi Mirza, Bing Xu, David Warde-Farley,
 Sherjil Ozair,[†] Aaron Courville, Yoshua Bengio[†]
 Département d'informatique et de recherche opérationnelle
 Université de Montréal
 Montréal, QC H3C 3J7

首先，我们有一个数据集的分布： $P_{data}(x)$ ，其中 x 是一个真实的图片，在 GAN 中，生成器可以生成一个参数为 θ 的分布。值得一提的是，生成器的模型可以是任意的，当然，在现在的处理中，我们更倾向于用神经网络来行使生成器的功能，因为它能形成比高斯模型复杂得多的分布。

接下来，我们需要在原始得数据中进行采样： $\{x^1, x^2, \dots, x^m\}$ ，这时候在生成器中得到一个似然，于是我们需要用到极大似然估计推导如下：

$$\theta^* = \arg \max_{\theta} \prod_{i=1}^m P_G(x^i; \theta)$$

对两边同时取对数，我们可以发现极大似然估计与最小化 KL 散度得计算殊途同归：

$$\begin{aligned}
& \arg \max_{\theta} \sum_{i=1}^m \log P_G(x^i; \theta) \\
& \approx \arg \max_{\theta} E_{x \sim P_{\text{data}}} [\log P_G(x; \theta)] \\
& = \arg \max_{\theta} \int P_{\text{data}}(x) \log P_G(x; \theta) dx \\
& - \int P_{\text{data}}(x) \log P_{\text{data}}(x) dx = \arg \min_{\theta} \text{KL}(P_{\text{data}} || P_G)
\end{aligned}$$

这里注意， P_G 可以是任何模型，而在这里我们选择神经网络的原因是，由于激活函数的存在，它可以拟合任意的分布，实际上，GAN 的基本公式如下：

Example Objective Function for D

$$V(G, D) = E_{x \sim P_{\text{data}}} [\log D(x)] + E_{x \sim P_G} [\log(1 - D(x))]$$

↑
(G is fixed)

Training: $D^* = \arg \max_D V(D, G)$ The maximum objective value
[Goodfellow, et al., NIPS, 2014] is related to JS divergence.

在这里，给定一个 G ，最大似然 V 就表示了生成器输出和样本之前的差异，在这个过程中调整的是 D 的参数，然后找到一个最好的 G 使得这个最大值最小。第一项表示真实数据，此时 D 要让这一项的 $D(x)$ 尽量大，后一项是 G 产生的数据， $D(x)$ 则要尽量小。

首先，对于给定的 G ，求解最优的 D :

$$\begin{aligned}
V &= E_{x \sim P_{\text{data}}} [\log D(x)] + E_{x \sim P_G} [\log(1 - D(x))] \\
&= \int_x P_{\text{data}}(x) \log D(x) dx + \int_x P_G(x) \log(1 - D(x)) dx \\
&= \int [P_{\text{data}}(x) \log D(x) + P_G(x) \log(1 - D(x))] dx
\end{aligned}$$

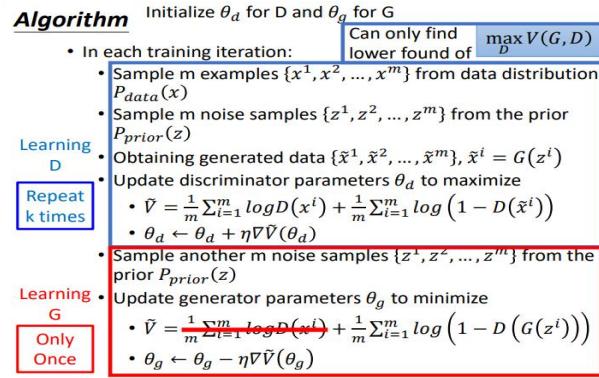
在这里我们要找的是 D 可以让其他项用常数代替，则所求的是：

$$P_{\text{data}}(x) \log D(x) + P_G(x) \log(1 - D(x))$$

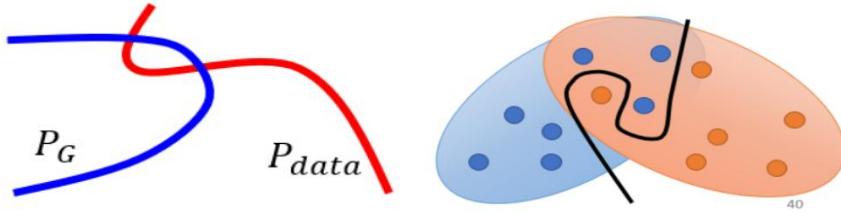
令所求为 $f(D)$ ，对其关于 D 求微分，并算出最优值为： $D^* = \frac{P_{\text{data}}(x)}{P_{\text{data}}(x) + P_G(x)}$ ，代入原式有：

$$\begin{aligned}
\max_D V(G, D) &= V(G, D^*) \quad D^*(x) = \frac{P_{\text{data}}(x)}{P_{\text{data}}(x) + P_G(x)} \\
&= -2 \log 2 + \int_x P_{\text{data}}(x) \log \frac{P_{\text{data}}(x)}{(P_{\text{data}}(x) + P_G(x))/2} dx \\
&\quad + \int_x P_G(x) \log \frac{P_G(x)}{(P_{\text{data}}(x) + P_G(x))/2} dx \\
&= -2 \log 2 + \text{KL}\left(P_{\text{data}} \parallel \frac{P_{\text{data}} + P_G}{2}\right) + \text{KL}\left(P_G \parallel \frac{P_{\text{data}} + P_G}{2}\right) \\
&= -2 \log 2 + 2 \text{JSD}(P_{\text{data}} || P_G) \quad \text{Jensen-Shannon divergence}
\end{aligned}$$

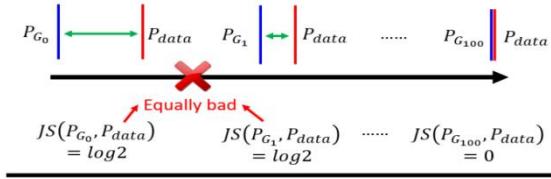
因此，我们可以说，在这个问题中，最大化目标函数与 JS 散度正相关，而 GAN 的训练步骤如下：



事实上，JS 散度存在一些问题，这是因为在很多情况下， P_G 与 P_{data} 并不重叠，而对于分布不重叠的两类，可以将数据（图片）看成高维空间中的一个低维的 manifold。不难看出，除非它们完全重合，否则它们的相交范围是可以完全忽略的。而即使这两个样本有重叠，但是我们在散度计算中采样的点不够多，那么实际上也相当于没有重叠，两种情况分别入下图（左），（右）所示。



而几乎没有重叠这件事情，对 JS 散度的影响是很大的，因为对于任意分布不重叠的两类 C_1, C_2 ，均有： $JS(C_1, C_2) \equiv \log 2$ 。这意味着，判别器始终可以准确分开这两类，从而导致生成器无法知道训练是否带来结果的提升，训练学不到东西。

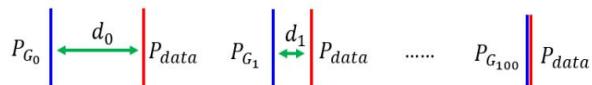


Intuition: If two distributions do not overlap, binary classifier achieves 100% accuracy.

The accuracy (or loss) means nothing during GAN training.

那既然是 JS 散度的问题，我们能不能换一种衡量两个分布相似程度的方式呢？于是，就有人提出了一种全新的度量方式，名叫 Wasserstein Distance。

Wasserstein Distance 的计算方法就好比是在“开一个推土机”，假设我们有两个分布：P 和 Q，那你把 P 想成是一堆土，把 Q 想成是你要把土堆放的目的地，那这个推土机把 P 这边的土，挪到 Q 所移动的平均距离，就是 Wasserstein Distance。直观上而言，它带来的改进是很大的，以上面那个情况为例，第二张图比第一张图有着明显的改进，因为“挪动”的距离变短了。



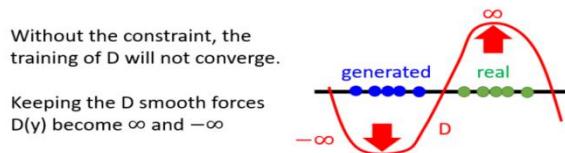
当然在实际情况下，平均距离并不会像上面那么直观容易计算，这时 GAN 的神奇之处再一次体现，有人提出了 GAN 的一种变体，名叫 WGAN，而 WGAN 实际上就是将 GAN 中

的 JS 散度替换成 Wasserstein Distance， WGAN 中的优化问题如下：

$$\max_{D \in 1-\text{Lipschitz}} \{E_{y \sim P_{\text{data}}}[D(y)] - E_{y \sim P_G}[D(y)]\}$$

D has to be smooth enough.

其中， D 是有一定限制的，它需要是一条平滑的曲线，因为如果不加这一条限制的话，训练将永远不会收敛。



那么如何做到这样的限制呢？最早 WGAN 论文给出了一个比较粗糙的处理，处理方法为始终将训练的参数设置在一个关于原点对称的区间之内。

Weight Clipping [Martin Arjovsky, et al., arXiv, 2017]

Force the parameters w between c and $-c$
After parameter update, if $w > c$, $w = c$;
if $w < -c$, $w = -c$

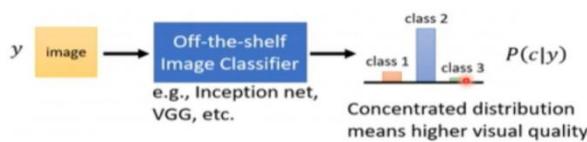
当然这样的处理方法并不能保证判别器成为 1-Lipschitz 函数，在 Spectral Normalization for Generative Adversarial Networks [1802.05957] Spectral Normalization for Generative Adversarial Networks (arxiv.org) 这篇论文中，作者提出了一个观点，即如果需要将 GAN 训练得足够好，可能需要做一下 Spectral Normalization。

8.3.3 如何评估生成器的好坏

既然我们已经知道 GAN 训练的好坏和生成器有关，那我们就需要一个机制来评估生成器的好坏。

最直觉的做法也许是用人眼看，事实上很长一段时间，尤其是人们刚开始研究生成式技术的时候，人们都是用人眼看，然后直接在论文最后放几张图片来评估生成器的好坏，我们可以发现比较早年的 GAN 的论文中，没有准确度等等的数字结果。但是这样显然是不行的，因为这种方法是不客观的，也不会很稳定。

那么后来人们在一些指定研究领域内提出了衡量标准，比如在处理图像时，我们可以跑一个影像辨识系统，输入是一张图片，输出是一个概率分布，这个概率分布代表说这张图片是猫的概率、狗的概率、斑马的概率等等。如果这个机率的分布越集中，就代表现在产生的图片可能越好。

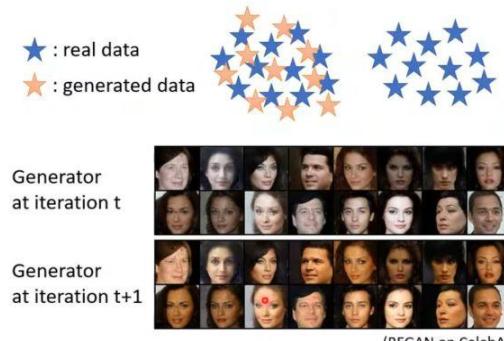


当然这样的做法是远远不够的，事实上，通过影像辨识系统判断产生出来的模型的好坏，

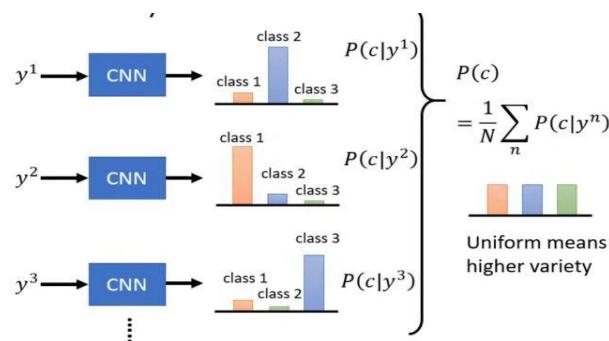
会引发一种名为模式崩塌 (Mode Collapse) 的问题。它的意思是说，在训练 GAN 的过程中，你会发现生成的图片来来去去就是这么几张，虽然结果肯定是对的，而且将图片单拎出来你可能会觉得还不错，但是这显然会影响模型的多样性。而且在实做中甚至不需要借助其他工具你都能发现这样的问题：当你发现你的生成器总是生成同一张图片或者同一类图片时，你甚至自己都不愿意承认它是一个好的生成器。



但是还有一种是人们很难侦测出来的问题叫做模式丢失 (Mode Dropping)，它的意思是说：GAN 能很好地生成训练集中的数据，但难以生成非训练集的数据，换言之，你的 GAN “缺乏想象力”，它产生出来的资料，只有真实资料的一部分，单纯看产生出来的资料，你可能会觉得还不错，而且分布的这个多样性也够，但你不知道真实资料的多样性的分布其实是更大的。

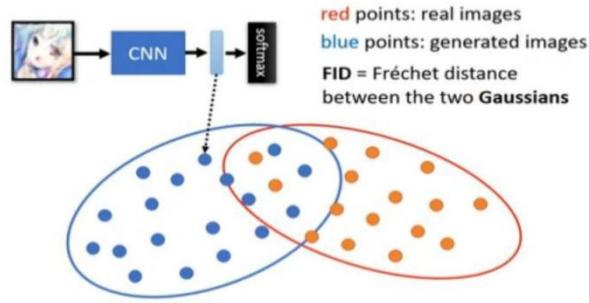


虽然存在以上的问题，但是我们需要去度量现在我们的生成器它产生出来的图片到底多样性够不够。如下图所示，有一个做法是借助我们之前介绍过的图像分类，把一系列图片都丢到图像分类里面，看它被判断成哪一个类别。每张图片都会给我们一个分布，我们将所有的分布平均起来，接下来看看平均的分布长什么样子。如果平均的分布非常集中，就代表现在多样性不够，如果平均的分布非常平坦，就代表现在多样性够了。



过去有一个常被使用的分数，叫做 Inception Score，缩写为 IS。其顾名思义就是用 CNN 中的 Inception 网络来做评估，Good quality 和 large diversity 代表着高的 Inception Score。事

实际上，现在有一种更新的评估方式，叫做 Fréchet Inception distance (FID)。具体做法是将生成的图片丢入 Inception 网络中，并让 Inception 网络输出它的类别。在这里需要注意的是，我们并不关注最后输出的类别，而是拿出进入 softmax 之前的隐藏层的向量，假设真实的图片跟产生出来的图片都服从高斯分布，然后去计算这两个分布之间的 Fréchet 的距离。两个分布间的距离越小越好，距离越小越代表这两组图片越接近，也就是产生出来的品质越高。



当然 FID 也有一定的问题，最直观的问题就是如果真实图片和生成图片不服从高斯分布怎么办？第二个问题是如果要准确的得到网络的分布，那需要产生大量的采样样本才能做到，需要一点运算量。

事实上，FID 算是目前比较常用的度量方式。但是仍然还存在一些问题，因为我们要求 FID 距离越小越好。那这边假如 GAN 生成的图片与真实图片长得一模一样，FID 距离是 0，这应该是 FID 能达到的理论上的最小值。那么有人会说，那 GAN 的存在还有什么用，我直接拿训练数据集里的图片就好了啊，生成器还有什么用？



因此，有时候我们也不能完全看 FID 的分数，FID 分数低，也未必代表这是一个好的生成器。这边有一篇论文，名叫”Pros and cons of GAN evaluation measures”，里面提出了 27 种衡量 GAN 的标准，有兴趣的读者可以自行查看这篇论文。

8.4 GAN 的常见变种

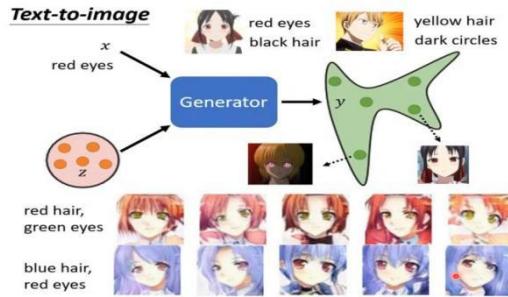
8.4.1 条件型生成 (Conditional GAN)

我们之前讲的 GAN 中的生成器，它输入都是一个随机的分布而已，它没有输入任何的条件，这个不一定非常有用。现在我们想要进一步操控生成器的输出，做法是给它一个条件 x ，输出 y 根据输入 z 和条件 x 共同决定。

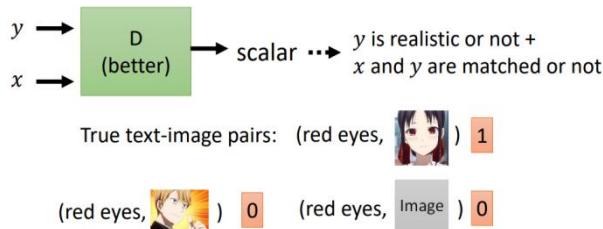
那这样的条件式生成有什么样的应用呢？比如可以做文字对图片的生成。这其实是一个监督学习的问题，我们需要一些有标签的数据，也需要去收集一些人脸的图片，然后这些人

脸需要有文字的描述。以下图的样本为例，有样本“红眼睛，黑头发”的样本，还有“黄头发，有黑眼圈”的样本。

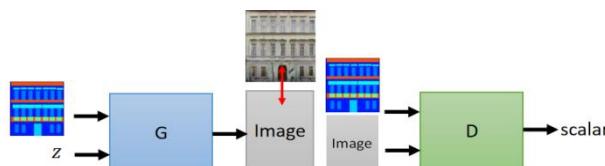
因此，在文字生成图片的任务中，条件 x 就是一段文字，输入一段文字，生成器产生一张根据文字描述的图片。那这段文字该怎么输入到生成器中呢？处理方法其实取决于你自己，过去常用的方法是 RNN，今天你也许可以用 Transformer 或者更先进的模型。总之，只要你的生成器能读出文字即可。



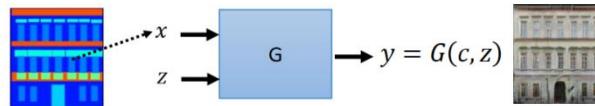
那具体应该如何操作呢？事实上，现在我们已经有了两个东西，一个是在最原始的 GAN 中就存在的采样出来的分布 z ，另一个是条件 x （在本例中就是一段文字），然后它会产生输出 y ，也就是一张图片。对于这个问题，条件 GAN 需要在最原始的 GAN 的基础上做一些改进，因为它不仅需要看输出的 y 好不好，还需要看 y 与条件 x 是不是适配，具体的做法是把文字和图像标注为成对的数据即可，所以当看到这些真正的成对资料，就给它 1 分，看到红色眼睛但是文字叙述是黑色头发，就给它 0 分，看到黑色头发但是文字叙述是红色眼睛，也给它 0 分，这样就可以训练辨别器了，如下图所示。



那其实在实际操作中，只是拿这样的负样本对和正样本对来训练辨别器，其实得到的结果往往不够好。往往还需要加上一种不好的状况：已经产生好的图片但是文字叙述配不上。所以通常会把我们的训练资料拿出来，然后故意把文字跟图片乱配，或者故意配一些错的，然后告诉判别器看到这种状况，也输出不匹配。经过反复训练，最后才会得到好的结果，这样才是一个条件 GAN 的完整训练。

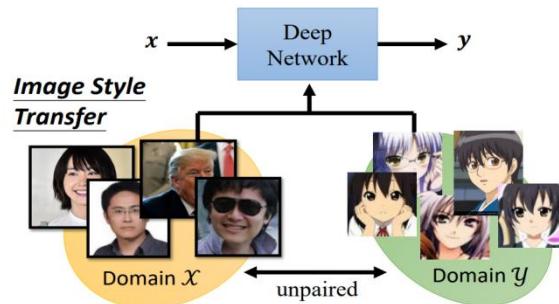


当然 Conditional GAN 还有一些其它应用，比如 Image Translation (Pix2Pix)，它其实是输入图像产生图像，同样也可以输入一段语音输出一个图像，感兴趣的读者可以自行探索。

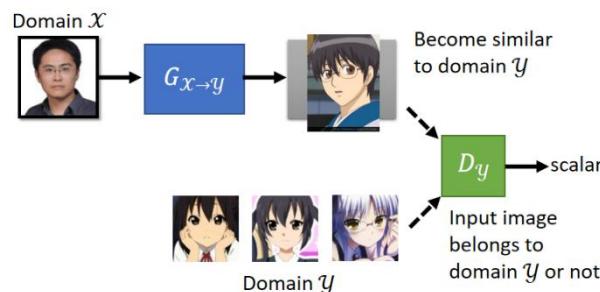


8.4.2 Cycle GAN

在本章的最后，我们要介绍 GAN 的另一个有趣的应用，就是将 GAN 应用到无监督学习上。那什么是无监督学习呢？我们在之后会进行详细的讲解。在这里，我们权且将它理解为：输入和输出并不会在任务中成对呈现出来。最典型的一个例子是图像风格迁移（Image Style Transfer），这个任务有一些真实的图片，也有一些特定风格的图像（比如：二次元的头像），我们希望把真实的图片转换成二次元风格的图片。在这个例子中，真人和与之对应的二次元头像显然不会成对出现，因为如果我们事先去根据真人图片画出二次元头像的话，训练的代价就太昂贵了。那么，在这种情况下，GAN 会是一个不错的解决办法，它能在这种完全没有成对资料的情况下进行学习。

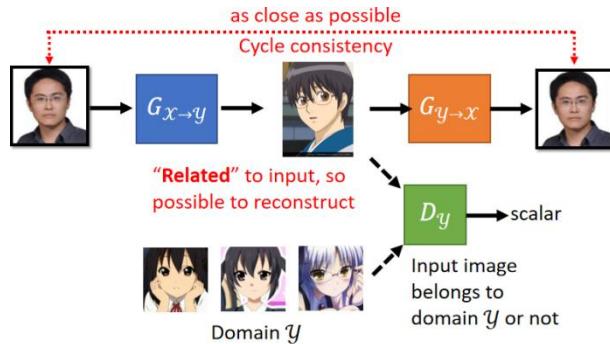


我们当然可以照搬前文所述的方法并做一些改动来解决这个问题，但是仔细想想，这么做好像是不够的。回想一下，我们需要机器生成的是一张 y 域的图，但是我们没有对输出的 y 域的图和输入之间的关系做任何的限制，所以生成器也许就把这张图片当作一个符合高斯分布的噪音，然后不管你输入什么它都无视它，只要判别器觉得它做得很好就结束了。但这并不是我们想要的，因为我们既希望输入一张真实的照片，产生出来的图片跟输入的照片有关系，又希望它是符合 y 域的分布的。

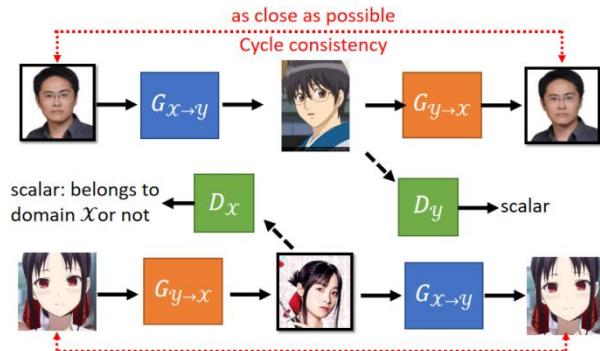


那么我们该如何“强化”输入和输出之间的关系呢？有些人可能会想到用条件型 GAN 的方法，但需要注意的是，条件型 GAN 中的数据也是成对出现的，而在我们现在的问题中

是没有成对数据的。为了解决这个问题，我们可以使用 Cycle GAN，中文翻译为循环生成对抗网络。



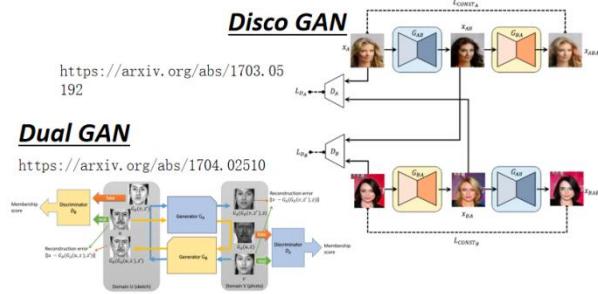
我们之所以将这个模型称为“Cycle”，是因为它其实存在一个闭环。在 Cycle GAN 中，我们会训练两个生成器。第一个生成器它的工作是把 x 域的图变成 y 域的图，第二个生成器它的工作是看到一张 y 域的图，把它还原回 x 域的图。在训练的时候，我们会增加一个额外的目标，就是我们希望输入一张图片，其从 x 域转成 y 域以后，要从 y 域转回原来一模一样的 x 域的图片。就这样经过两次转换以后，输入跟输出要越接近越好，或者说两张图片对应的两个向量之间的距离越接近越好。



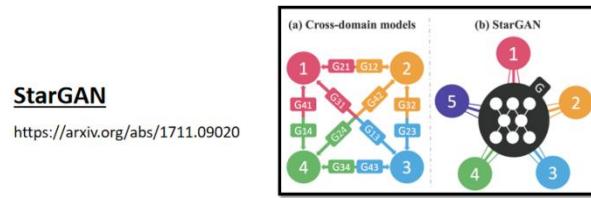
另外还有一个问题，我们需要只保证第一个生成器的输出和输入有一定的关系，但是我们怎么知道这个关系是所需要的呢？机器自己有没有可能学到很奇怪的转换并且满足 cycle 的一致性呢？比如一个很极端的例子，假设第一个生成器学到的是把图片左右翻转，那只要第二个生成器学到把图片左右翻转就可以还原了啊。这样的话，第一个生成器学到的东西跟输入的图片完全没有关系，但是第二个生成器还是可以还原回原来的图片。

所以如果我们做 Cycle GAN，用 cycle 的一致性的话似乎没有办法保证我们输入跟输出的人脸看起来很像，因为也许机器会学到很奇怪的转换，反正只要第二个生成器可以转得回来就好了。面对这个问题目前确实没有什么特别好的解法，但实际上使用 Cycle GAN 的时候，这种状况没有那么容易出现。实际上使用 Cycle GAN 时，输入跟输出往往真的就会看起来非常像，甚至实际应用时就算没有第二个生成器，不用 Cycle GAN，使用一般的 GAN 替代这种图片风格转换的任务，往往效果也很好。因为网络其实非常“懒惰”，输入一个图片它往往就想输出默认的与输入很像的东西，它不太想把输入的图片做太复杂的转换。所以在实际应用中，Cycle GAN 的效果往往非常好，而且输入跟输出往往真的就会看起来非常像，或许只是改变了风格而已。

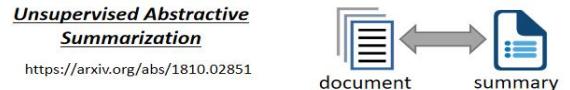
除了 Cycle GAN 以外，还有很多其他的可以做风格转换的 GAN，比如 Disco GAN、Dual GAN 等等，如下图所示。这些 GAN 的架构都是类似的。



此外还有另外一个更进阶的可以做影像风格转换的版本，叫做 StarGAN。Cycle GAN 只能在两种风格间做转换，而 StarGAN 厉害的地方是它可以在多种风格间做转换。其结构和文章链接如下所示：

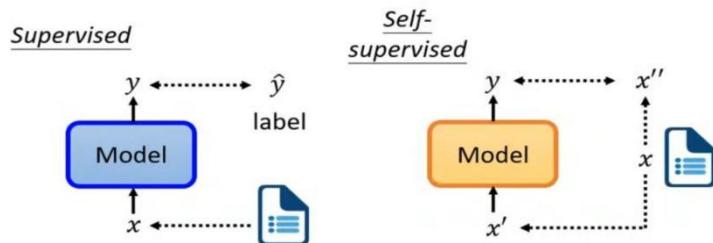


当然，对于风格迁移（Style Transfer）的问题，我们不仅可以将其用在图像上。我们也可以做文字风格迁移，比如将一句负面的句子转化正面的。当然还有一些其他应用，比如让机器学会把长的文章变成简短的摘要，让机器学会把中文翻成英文，甚至可以进行语音辨识，也就是让机器听一些声音，然后学会把声音转成文字。当然还有很多很多的有趣的应用等待大家去探索。

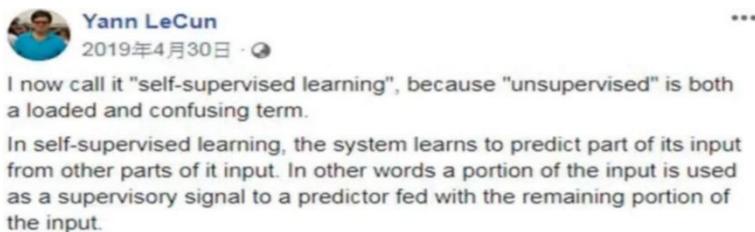


9 自监督学习 (Self-supervised Learning)

在本章，我们将对自监督学习 (Self-supervised Learning) 进行介绍。事实上，我们在前几章已经对监督学习 (Supervised Learning) 有了一定的了解，无论是卷积神经网络 (CNN)，循环神经网络 (RNN) 还是 Transformer，它们的学习任务都是监督学习。它们的共性是：在模型训练期间使用带标签的数据。而自监督学习的含义是：在没有标签的情况下，机器自己想办法“监督自己”的任务。



当然，由于这一类问题也是没有标签的一种学习，我们也可以将它归为无监督学习 (Unsupervised Learning) 这一类中，但是在 2019 年，Yann LeCun 曾公开表示将这一类问题定义为自监督学习，这是因为无监督学习的“家族”过于庞大，为了使定义更清晰，他将自监督学习单独地划分了出来。



自监督学习的模型大多都是以芝麻街的角色命名，比如：ELMo，BERT，GPT 等，在本章中，我们主要对 BERT 和 GPT 进行介绍。

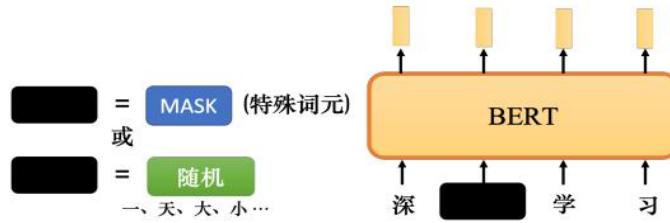
9.1 BERT 的模型结构

9.1.1 BERT 模型究竟在做什么事

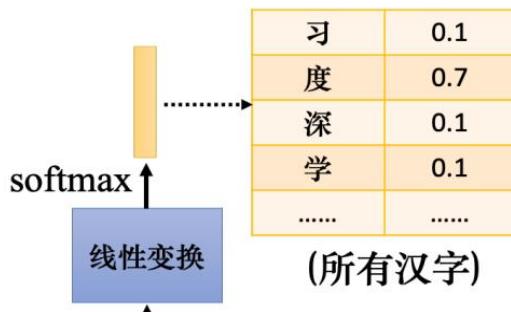
在本章的开头，我们对监督学习，自监督学习和无监督学习进行了一个抽象的介绍，接下来，我们以 BERT 这个模型进行说明。BERT 这个模型本质上就是 Transformer 的一个 Encoder。它实际上一共做了两件事，第一件事是 Masking Input，第二件事是 Next Sentence Prediction。

因为 BERT 本质上是 Transformer 的编码器，因此输入和输出的序列长度是一样的。Masking Input 的意思是说，BERT 会随机“盖住”一些词元 (token)，盖住的方法有以下两种：

1. 将盖住的词元标记为一个特殊的符号，记为 MASK
2. 在语料库中随机选择一个词元。



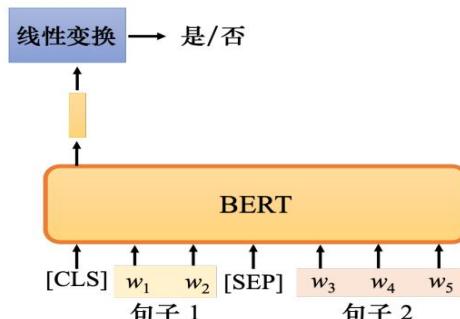
这两种方法都可以使用，使用哪种方法也是随机确定的。所以在 BERT 训练的时候，应输入一个句子需要做两次决定，首先随机决定遮盖哪些汉字，之后，再决定如何进行掩码。



在完成掩码之后，我们会得到一个输出序列，这个序列是一个长度与输入相同的向量，将输出向量经过一个线性变换（Linear Transform）和 softmax 之后，得到一个概率分布，它包含了所有可能的被遮盖住的词元的值。其中，概率最高的就应该是被盖起来的词元的值，过程如上图所示。事实上，我们可以将 BERT 训练的过程看作一个多分类问题，它的训练目标就是最小化预测结果与已知真实值（ground truth）之间的交叉熵。

另一件事是 Next Sentence Prediction (NSP)。我们可以通过在互联网上谱曲一些语料作为数据库，然后从其中拿出两个句子。这两个句子由一对特殊符号<[CLS], [SEP]>分隔开，这是基于 BERT 的特殊的输入结构决定的。事实上，在 BERT 训练之前，我们需要用一个函数来获取输入序列的词元及其段索引。

我们将整个输入丢入 BERT 中，其输出理论上是一个序列，现在我们只考虑输出序列中与输入中[CLS]符号对应的输出，并将其进行一个线性变换，并让它做一个 Yes/No 的二分类问题来预测这两个句子是否相接。



但是后来的研究返现，NSP 任务对 BERT 的下游任务没什么帮助，在 RoBERTa 的文章中有提到这个观点。可能的原因是这个任务太简单了，因此在训练 BERT 完成 NSP 任务时，

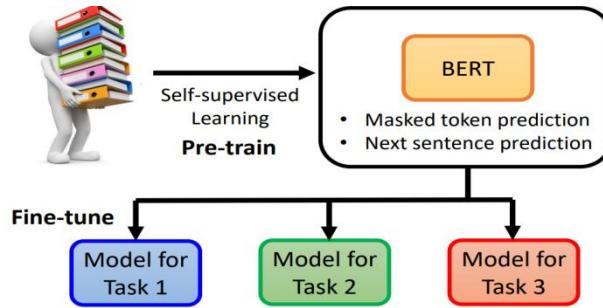
可能并没有学到太多东西。实际上，似乎另外一种方法是叫做 Sentence order prediction(SOP)，它比 NSP 在文献上看来似乎更有用，因为它不仅需要判断这两句话是否相连，还需要决定它们出现的顺序。这一应用在一种名为 ALBERT 的模型中被使用，该模型是 BERT 的进阶版。

9.1.2 BERT 预训练 (Pretrain) 与微调 (fine-tune)

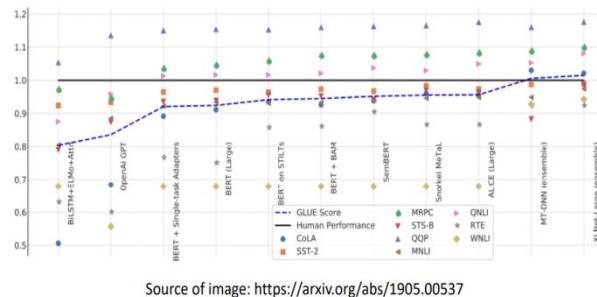
我们已经了解到了 BERT 主要做的两件事。换言之，我们可以这样理解，BERT 已经学会了怎么做“完型填空”。但现在的问题在于，假如我们现在的任务不是“填空题”，那应该如何处理呢？

事实上，BERT 的神奇之处在于，在训练模型学会填空这项任务之后，当我们面对其他任务的时候，BERT 仍然适用。这些真正使用 BERT 的任务叫做 BERT 的下游任务(Downstream Task)。当然，对于不同的下游任务，使用 BERT 的超参数概率是不同的。

如图所示，我们可以将整个 BERT 分成上下两个部分，对于下面的部分，即将 BERT 分化并用于各种任务称为微调 (fine-tune)，而与微调相对应的，在其之前的过程称为预训练 (Pre-train)。



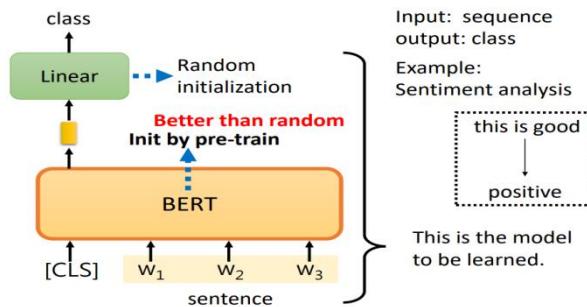
在谈如何对 BERT 进行微调之前，我们需要了解 BERT 的能力。通常，我们会把 BERT 在多个下游任务上进行测试，而不会测试其在单个任务上的能力，可以让 BERT 分化做各种任务来查看它在每个任务上的正确率，再取个平均值。对模型进行测试的不同任务的这种集合，可以将其称为任务集，任务集中最著名的标杆(基准测试)称为通用语言理解评估(General Language Understanding Evaluation, GLUE)，它包括文本相似度，自然语言推理，问答推理等 9 个任务，当然它也有中文版本。



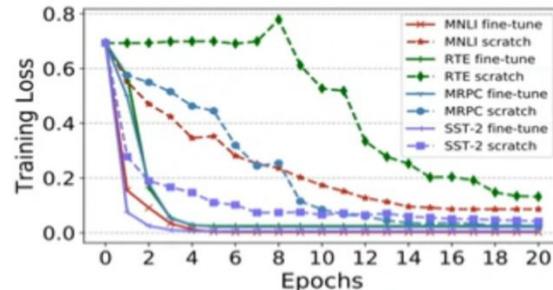
9.1.3 使用 BERT 的 4 种情况

在 BERT 及其下游任务中，我们可以将使用 BERT 的情况分为以下 4 种：

第一种情况是情感分析，它的输入是一个序列，输出一个类别，这实际上是一个分类问题。在情感分析任务中，将[CLS]词元对应的输出向量分别做线性变换和 softmax，得到 softmax 的类别。但是，必须要有下游任务的标注数据。换句话说，BERT 没有办法从头开始解决情感分析问题，其仍然需要提供很多句子以及它们的正面或负面标签来训练 BERT 模型。一言以蔽之，完整的情感分析模型包括 BERT 和线性变换的部分。而对于这样一个模型，我们在之前需要将所有参数随机初始化，再在训练过程中不断地做梯度下降。但在这个模型中，我们只需要随机初始化线性变换的参数，而对于 BERT 中的参数，我们只需要拿出它在预训练“做填空题”时的参数即可，这样做的原因有两点：一是可以减少随机化参数的个数，二是 BERT 预训练初始化的参数表现比随机初始化要更好。

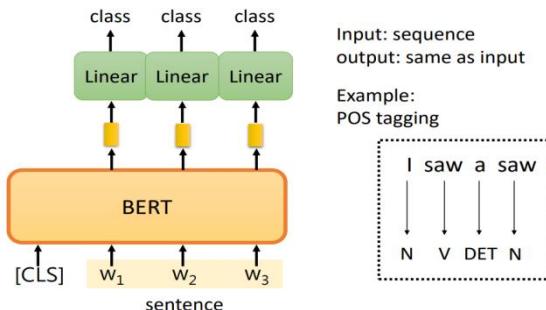


这当然不是我们瞎吹的，它是有一定研究依据的。研究表明：fine-tune 的 BERT 模型训练效果比 training from scratch 的效果更好。



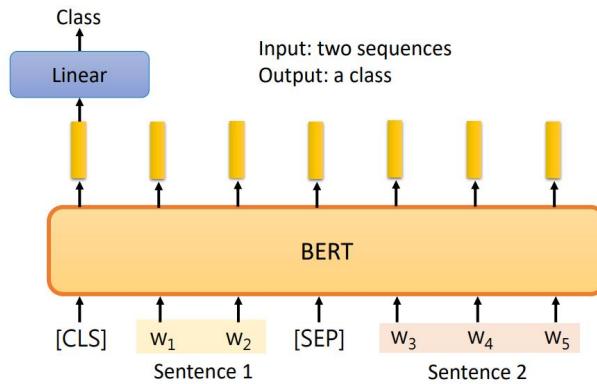
需要注意的是，BERT 虽然在预训练的时候是无监督学习，但它在执行下游任务时需要有标注的数据，所以整个 BERT 合起来是半监督学习（Semi-supervised Learning）。

第二种情况是词性标注（POS tagging），输入是一个序列，输出是一个与输入序列长度一样的序列。在词性标注的任务中，我们将每个单词对应的输出向量分别进行线性变换和 softmax，得到每一个单词的词性，其余与情况 1 完全一样。



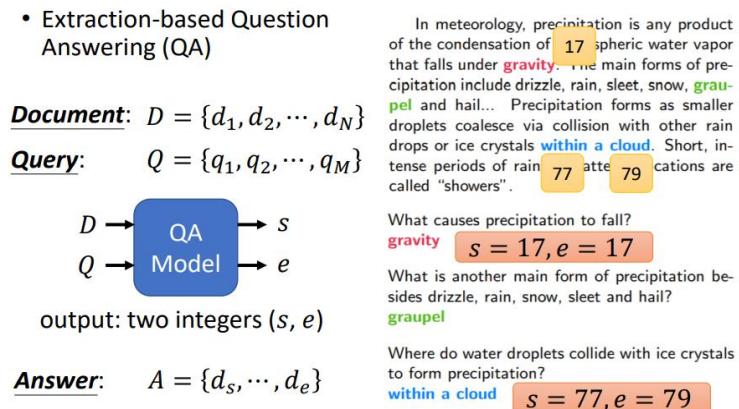
第三个情况是自然语言推理（NLI），它的输入是两个句子，输出一个类别。NLI 任务是在某个前提（premise）下，输出某个假设（hypothesis）与前提的关系是什么：是矛盾（contradiction），蕴涵（entailment），还是中立（neutral）。这个应用场景很常见，比如立

场分析。

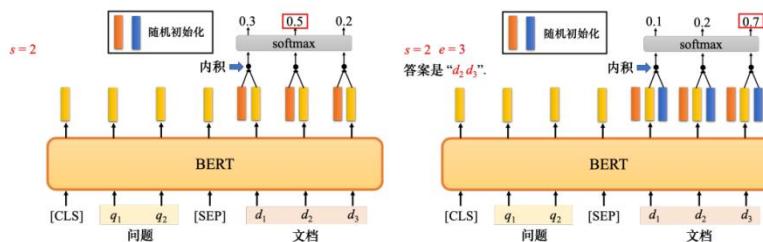


那 BERT 如何解这个问题呢？与 BERT 的基本任务 NSP 一样，输入两个句子，同样还是拿出[CLS]的词元对应的输出向量分别进行线性变换和 softmax，输出对应的类别。

最后一个情况是基于抽取的问答（Extraction-based Question Answering），那什么是“基于抽取”呢？意思就是说，答案必须来自于文章。如下图所示，输入的 document 中有 N 个单词，输入的 query(question) 有 M 个单词，最后输出两个正整数 s,e，分别代表答案在 document 中的开始（s for start）和结束（e for end）位置。

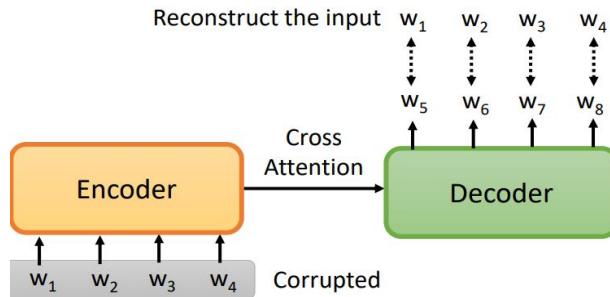


当然我们不可能对这个问题 train from scratch, 那如何利用 BERT 进行训练呢？首先我们需要和上述任务一样，用[SEP]将 Query 和 Document 分割开来，并随机初始化两组向量，将第一组随机初始化的向量与 document 对应的输出向量做内积，其结果进行 softmax 得到概率分布。选择最大概率所在的位置作为起始位置，对于第二组随机初始化的向量，做同样的操作寻找到的位置作为结束位置。

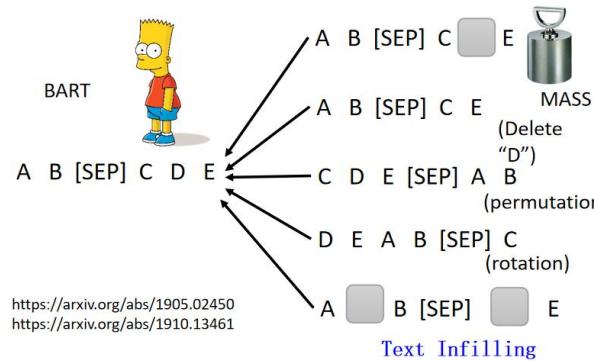


以上任务均不包括 Seq2Seq 模型。我们知道，BERT 只有预训练编码器，那有没有办法预训练 Seq2Seq 的解码器呢？其实是有的，如下图所示，图中有一个编码器和一个解码器。输入是一串句子，输出是一串句子。将它们与中间的交叉注意力（cross attention）连接起来，

然后对编码器的输入做一些扰动来损坏它。编码器看到弄坏的结果，解码器需要试图还原。



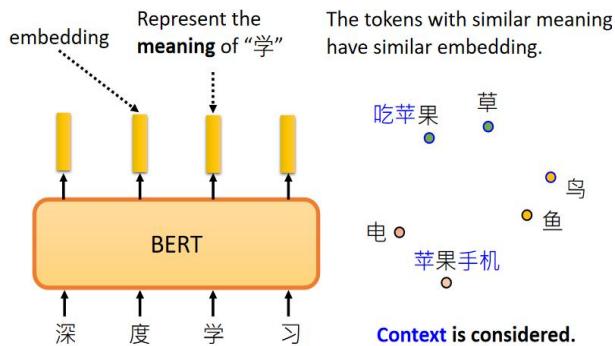
弄坏句子的方法有很多种，有一篇名为 MASS 的论文提出，弄坏句子的方法可以和 BERT 做填空题时一样，将一些地方遮盖住。当然后续也有一篇名为 BART 的论文提出了更多的破坏方法，比如删除，打乱句子顺序以及把单词的顺序做旋转，也可以将这些方法综合使用。



当然你可能想做一个实验看看到底哪种方法能最好地进行“搞破坏”，但是 Google 告诉你已经不需要做了，有一篇名为 T5 的论文(Transfer Text-to-Text Transformer)在 C4 的 Corpus 上进行了训练。感兴趣的读者可以自行阅读查看结果。

9.1.4 BERT 有用的原因

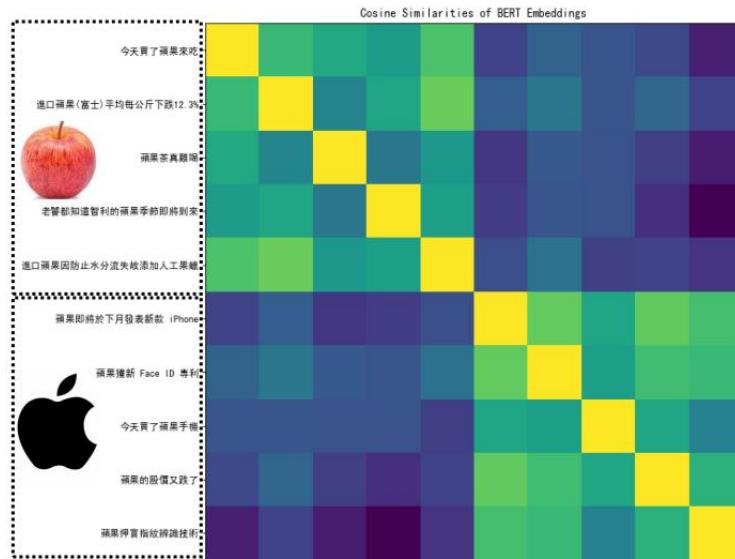
为什么 BERT 有用？最常见的解释是，当输入一串文字时，每个文字都有一个对应的向量，这个向量称为嵌入（Embedding）。如下图所示，这个向量很特别，因为它代表了输入字的意思。我们把这些字对应的向量一起画出来并计算它们之间的距离，意思越相似的字距离越近。



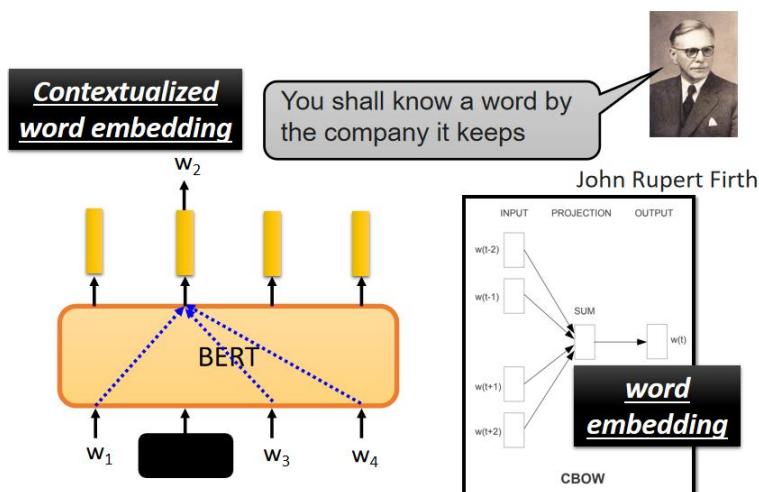
事实上，如上图右侧所示，中文有着“一词多义”的特点，这也是世界上很多语言的特

点。但是 BERT 在这里就会考虑上下文，以上图为例，“吃苹果”中的“果”与“草”更接近，而“苹果手机”中的“果”与“电”更接近。

当然，上面的是一个“toy example”，在实做中，我们会收集一些语料库，其中有很多句子，我们会将这些句子放入 BERT 中，再去计算指定词元对应的词嵌入，由于编码器中有注意力机制，根据每一个词元的上下文得到的向量也是不一样的。接下来，计算这些向量之间的余弦相似度，如下图所示：



为什么 BERT 可以输出代表输入字意思的向量？1960 年代的语言学家 John Rupert Firth 提出了一个假设，他说要知道一个词的意思，就要看这个词的 company，也就是经常和它一起出现的词汇（上下文）。一个词的意思取决于它的上下文。以苹果中的“果”为例，如果它经常与吃、树等一起出现，它可能指的是可以吃的苹果；如果经常与电、专利、股价等一起出现，可能指的是苹果公司。因此，可以从上下文中推断出单词的意思。而 BERT 在学习填空的过程中所做的，也许就是学习从上下文中提取信息。



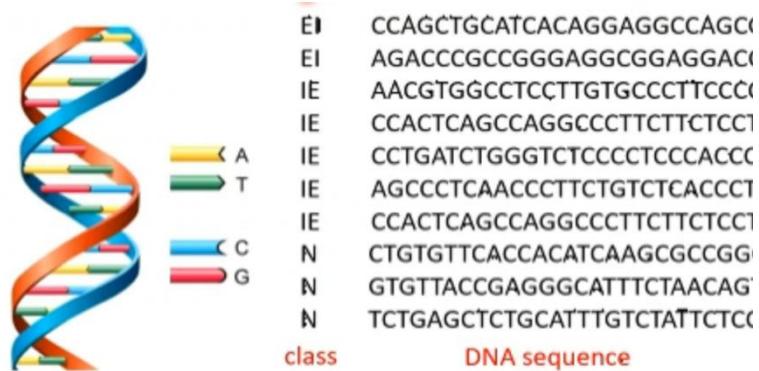
事实上这样的技术在 BERT 出现之前就被使用了，有一种技术是词嵌入(word embedding)，词嵌入中有一种技术称为连续词袋（Continuous Bag Of Words, CBOW）模型，它所做的与 BERT 完全相同，把中间挖空，预测空白处的内容。这个模型非常简单，只是一个线性模型，

这主要是因为当时技术受限，没有那么多的资源去用深度学习做这件事。因此，BERT 可以看作于一个深度版本的 CBOW 模型。与传统的 CBOW 模型不同的是，BERT 还可以根据不同的上下文从相同的词汇中产生不同的嵌入，因为它是词嵌入的高级版本，考虑了上下文。BERT 抽取的这些向量或嵌入也称为语境化的词嵌入（contextualized word embedding）。

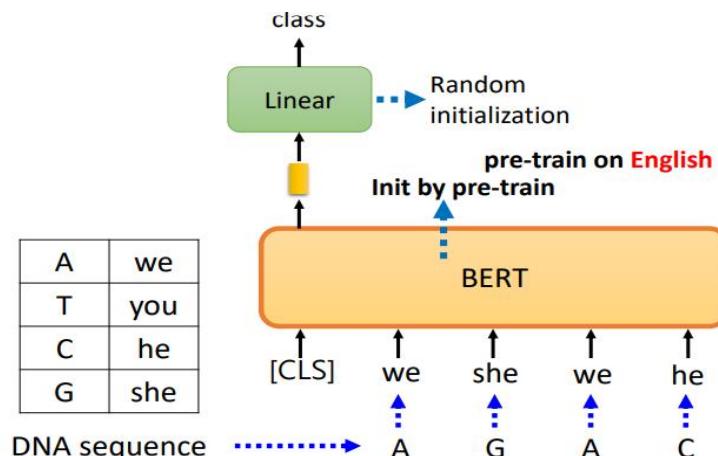
9.2 BERT 的奇闻异事

9.2.1 意想不到的 BERT 应用

我们在之前了解了 BERT 的主要应用，但实际上，BERT 的能量远超乎你想象。训练在文字上的 BERT 也可以用来对蛋白质、DNA 和音乐进行分类。以 DNA 链的分类问题为例，DNA 由脱氧核苷酸组成，脱氧核苷酸由碱基、脱氧核糖和磷酸构成，其中碱基有 4 种：腺嘌呤（A）、鸟嘌呤（G）、胸腺嘧啶（T）和胞嘧啶（C）。给定一条 DNA，尝试确定该 DNA 属于哪个类别（EI、IE 和 N 是 DNA 的类别）。总之，这是一个分类问题，只需用训练数据和标注数据来训练 BERT 就可以了。



如图所示，DNA 可以用 ATCG 表示，如下图所示，每个字母可以对应到一个英语单词，例如，“A”是“we”，“T”是“you”，“C”是“he”，“G”是“she”，当然你也可以自己随机产生。我们将 DNA 序列重新用构造的单词丢入 BERT 中，这就与我们之前所做的事情看起来就一样了。



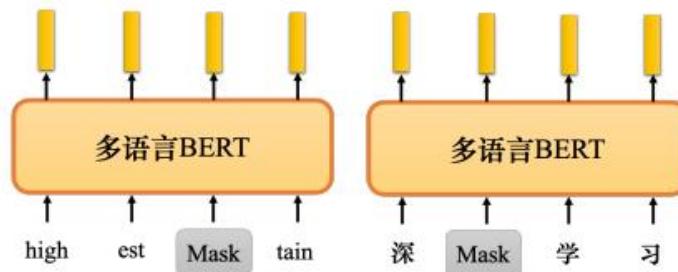
你可能不知道这个实验到底是在做什么，因为这个实验翻译出来的文本真的是不知所云，但有人就将这个问题利用 BERT “硬 Train”了一发，发现效果也是比“train from scratch”要

好很多。这个实验可能表明，虽然 BERT 可以学到语义，但假如我们给到它一个不知所云的句子，它依然可以有所表现，所以也许 BERT 的能力并不完全来自他看得懂文章这件事，可能还有其他原因。例如我们可以猜测，BERT 可能本质上只是一组比较好的初始化参数，它不一定与语义有关，也许这组初始参数比较适合训练大型模型，这个问题需要进一步的研究来回答。总而言之，BERT 是一个有很大研究空间的东西。

	Protein			DNA				Music
	localization	stability	fluorescence	H3	H4	H3K9ac	Splice	composer
specific	69.0	76.0	63.0	87.3	87.3	79.1	94.1	-
BERT	64.8	74.5	63.7	83.0	86.2	78.3	97.5	55.2
re-emb	63.3	75.4	37.3	78.5	83.7	76.3	95.6	55.2
rand	58.6	65.8	27.5	75.6	66.5	72.8	95	36

9.2.2 BERT 在多语言上的应用

BERT 还有很多其他的变种，比如多语言 BERT（multi-lingual BERT）。如下图所示，多语言 BERT 使用中文、英文、德文、法文等多种语言进行训练 BERT 做填空题。谷歌发布的多语言 BERT 使用 104 种不同的语言进行训练，所以它可以做 104 种语言的填空题。



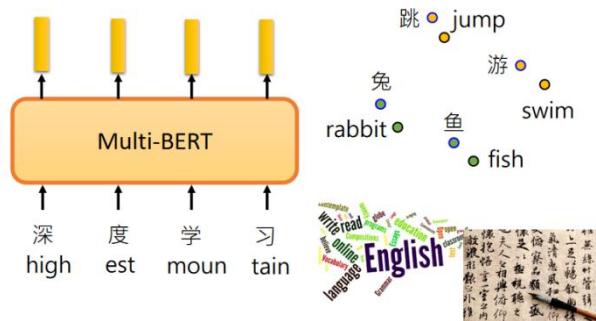
多语言 BERT 有一个神奇的功能，如果用英文问答数据训练它，它会自动学习如何做中文问答，下面是一个真实的实验，选取的英文数据集是 SQuAD，中文数据集是 DRCD。我们发现，在 BERT 没出现之前的结果并不出众，效果最好的是 QANet 模型，达到了 78.1 的 F1 Score，但是用最一般的 BERT，F1 Score 就飙升到了 89.1，而使用中英结合的多语言 BERT，F1 Score 还能进一步涨到 90.1，而值得注意的是，这与人类的表现已经相差无几了。而更值得注意的是，它从来没有看到过中文，相当于“裸考”！

Model	Pre-train	Fine-tune	Test	EM	F1
BERT	none	Chinese		66.1	78.1
	Chinese	Chinese		82.0	89.1
	104 languages	Chinese	Chinese	81.2	88.7
		English		63.3	78.8
		Chinese + English		82.6	90.1

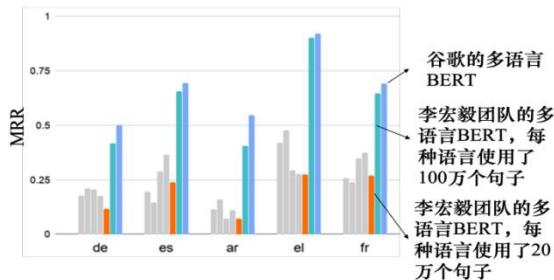
F1 score of Human performance is 93.30%

那为什么会这么神奇呢？一个可能的解释是：对于多语言的 BERT，不同的语言的差异不大。不管是中文还是英文，对于意思相同的词，它们的词嵌入离得很近。也许多语言 BERT

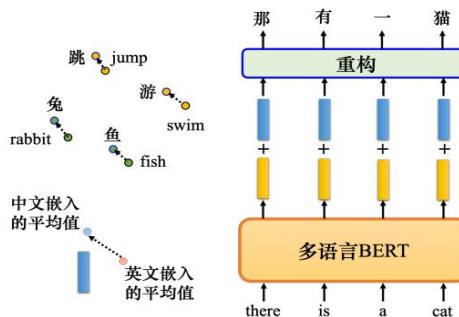
在看过大量语言的过程中自动学会了这件事情。



我们可以做一些验证，验证的指标叫做 Mean Reciprocal Rank (MRR)，MRR 的值越高，代表不同语言的 Alignment 更好。谷歌曾经就做过这样的实验，而李宏毅团队也曾做过这样的实验，但是效果远远不如 Google，后来李宏毅团队开始尝试加大资料量，从原先的 200k 的语料加大到了 1000k，海岸效果已经和 Google 的差不太多了。由此可见，资料量和算力是做这件事情的主要影响因素。

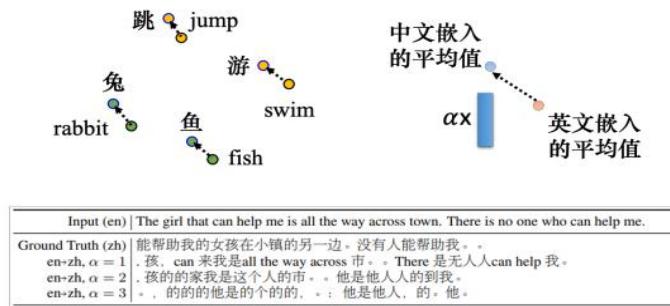


到现在为止，事情好像还是有些奇怪。有人可能会问，既然中英文似乎没什么差别，那为什么多语言 BERT 在学习填空的时候，并没有在填空中中英混用呢？那这就意味着，BERT 一定知道一些语言的信息。那些来自不同语言的符号毕竟还是不同的，它不会完全抹掉语言信息，语言信息并没有隐藏很深，把所有的英文的 Embedding 和中文的 Embedding 平均起来，它们之间的差值就是中英文之间的差异。现在它就可以做一件神奇的事情。如下图所示，将一句英文句子丢入 BERT，再加上中英文语言的差异，就可以重构出这句话的中文版本。



这意味着，多语言 BERT 可以做一个很棒的无监督翻译，如下图所示，把“The girl that can help me is all the way across town. There is no one who can help me.”这句话扔进多语种 BERT，再把蓝色的向量加到 BERT 的嵌入上，本来 BERT 读到的是英文句子的嵌入，加上蓝色向量，BERT 会觉得它读到的是中文的句子。当然，从结果中我们可以看出，这不是一个很好的翻译，但这件事可以表明，BERT 可以某种程度上做到无监督词元级翻译（unsupervised

token-level translation）。至于它为什么还不够好，根本原因还是在于多语言 BERT 表面上看起来把不同语言、同样意思的单词拉得很近，但是语言的信息还是藏在多语言 BRRT 里面。

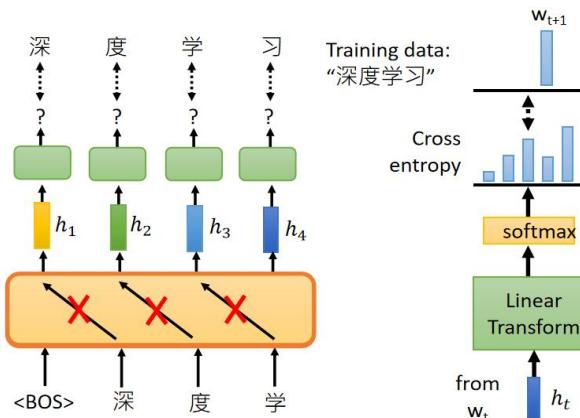


9.3 生成式预训练 (GPT)

在自监督学习中，除了 BERT 系列的模型，还有一个非常有名的模型——GPT 系列的模型。

9.3.1 GPT 的框架概念

我们在前面介绍了 BERT，BERT 的本质是做“填空题”，而 GPT 模型可以对这个问题做更进一步的事情，即预测下一个词元。如下图所示，假如我们有一笔资料是“深度学习”，给 GPT 输入<BOS>词元，GPT 会输出一个嵌入，接下来用这个嵌入去预测下一个应该出现的词元，在这个句子里面，根据这笔训练数据，下一个应该出现的词元是“深”，依次类推，我们接下来将<BOS>和“深”输入给 GPT，下一个应该出现的是“度”…



事实上，这个部分的具体操作和一般的分类问题是差不多的，对于一个词嵌入 h ，GPT 将它经过一个线性变换和一个 softmax，并计算输出的分布跟正确答案的交叉熵，使这个交叉熵越小越好。

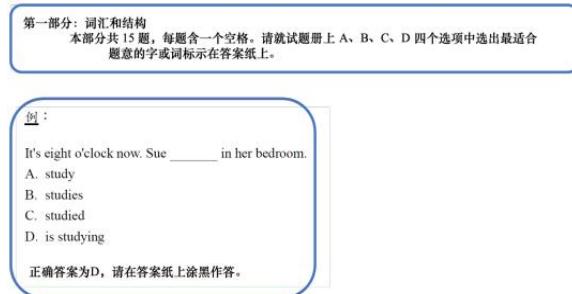
这边需要注意的是，GPT 的模型是建立在 Transformer 的解码器的基础上，不过与 Transformer 解码器有所不同的是，GPT 会做掩码注意力（Masked-attention），也就是说，给定 <BOS> 预测“深”的时候，不会看到接下来出现的词汇，给 GPT “深”要预测“度”的时候，其也不会看到接下来要输入的词汇，以此类推。

9.3.2 GPT 的使用

那么，我们在实际问题中会怎样使用 GPT 呢？对于这个比 BERT 还要庞大的模型，我们

当然不会只用一笔句子训练 GPT，实做中，我们会用成千上万个句子来训练模型。那如何将这一功能用在下游的任务，比如 QA 或是其他 NLP 任务上呢？

我们当然可以让 GPT 仿照 BERT 运作的方法去解决这样的下游任务，但 GPT 论文中并没有这么做，原因主要有两点，首先，BERT 已经用过这个方法了，其次，GPT 模型太大了，大到连微调可能都有困难。



因此，在 GPT 论文中提出了一种更疯狂的做法，这个做法和人类学习的方法可能更接近：如上图所示，假设考生正在进行一次考试，需要做一道选择题，人们肯定是期待考生能通过一个范例和答案，举一反三地进行作答。



我们期待 GPT 能像上面这个考生一样进行一些指定任务的学习，比如我们期待它在做机器翻译时，我们并没有“教”它怎么做翻译，只是给了几个范例，并且给出一个全新的单词，期待 GPT 能通过文字补全来实现翻译的效果。这种学习方法，由于样本比较少，我们将它称为少样本学习（Few-shot Learning），在少样本学习中，我们是不会运作梯度下降算法的。

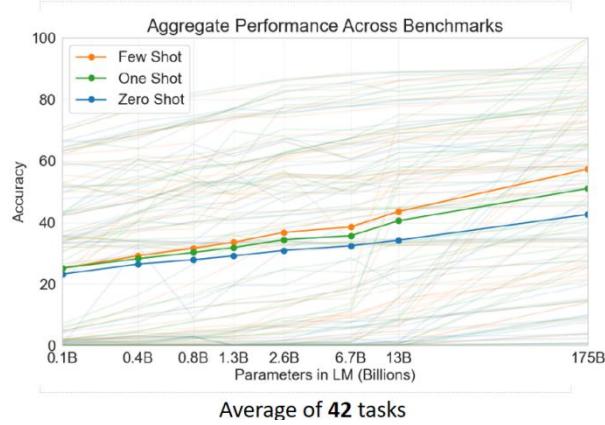


当然，你可以试图去做一些更疯狂的是，比如我们将样本设置为 1，这种做法叫做单样本学习（One-shot Learning），如上图所示，我们只给出了一组英文和法文的对照，期待机器能实现英法互译。

我们还可以更疯狂一点，即不给出例子，期待机器直接完成这一工作，这种做法叫零样本学习（Zero-shot Learning），这一内容在后续迁移学习（Transfer Learning）章节中会进一步进行介绍。



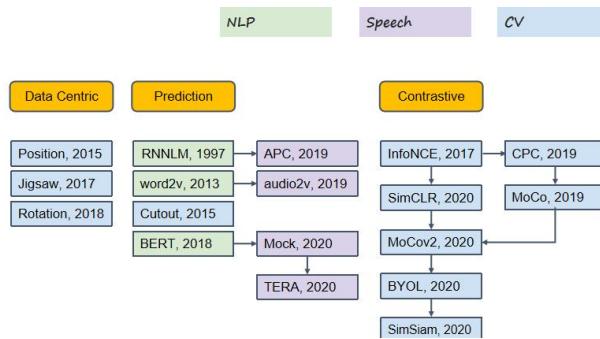
当然你可能会觉得奇怪，因为你可能会觉得少样本学习，尤其是零样本学习，似乎和你所认为的机器学习有点违背了。那么 GPT 真的能完成这样的任务吗？这是一个见仁见智的问题，它不是完全不可能答对，但相较于微调模型，正确率肯定是偏低的，GPT 系列的作者曾经做过一些实验，即将少样本，单样本和零样本学习运用在 42 个任务中，准确率如下所示：



我们发现，对于某些任务，GPT 似乎真的学会了，例如加减法，GPT 可以得到两个数字相加的正确结果；但是有些任务 GPT 可能怎么学都学不会，例如一些跟逻辑推理有关的任务，它的结果就不如人意。

9.4 自监督学习的其他应用

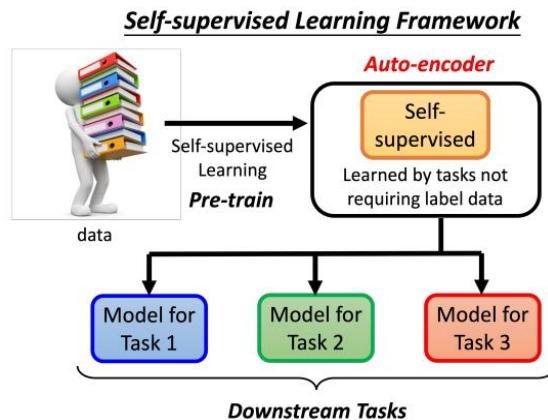
我们在本章中主要讲解了 BERT 和 GPT 两个模型，它们其实基本都是以用在文字上为主的，但需要注意的是，自监督学习不仅仅可以用在文字上。因此在本章的最后，我们来简单地介绍一些自监督学习在其他领域上的应用。



自监督学习不仅可以用在文字上，还可以用在语音和计算机视觉上。计算机视觉中比较典型的模型是 SimCLR 和 BYOL。在语音也可以使用自监督学习的概念，可以试着训练语音版的 BERT，语音版的 BERT 同样也是做“填空题”，只不过我们需要把原先的文字换成语音讯号即可。事实上，语音版的 BERT 都已经有很多相关的研究成果了，我们给出了各个方向自监督学习的一些模型和应用，感兴趣的读者可以自行探究。

10 自编码器 (Autoencoder)

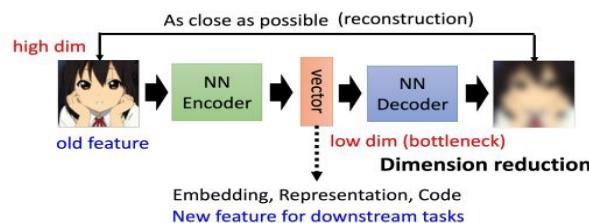
之前我们学习自监督模型的时候，就有看到过下面这个图，先预训练一个预训练模型(不用标签的，自监督学习，例如 Bert、GPT)，然后再在这个模型的基础上去做下游任务。事实上，在 BERT 和 GPT 等自监督学习模型问世之前，自监督学习的概念就已经被提出，而那时主流的模型就是自编码器（Autoencoder），它所做的事情与 BERT 几乎相差无几，如下图所示：



10.1 自编码器的概念

10.1.1 从另一个角度认识自监督学习

我们在上一章曾用文字的例子对自监督学习进行了介绍，事实上，自监督学习可以用在不同类型的数据（比如：文字，图片，语音等）中，因此，在了解自编码器的原理之前，我们以图片为输入的例子，重新对自监督学习进行一个简单的认识。

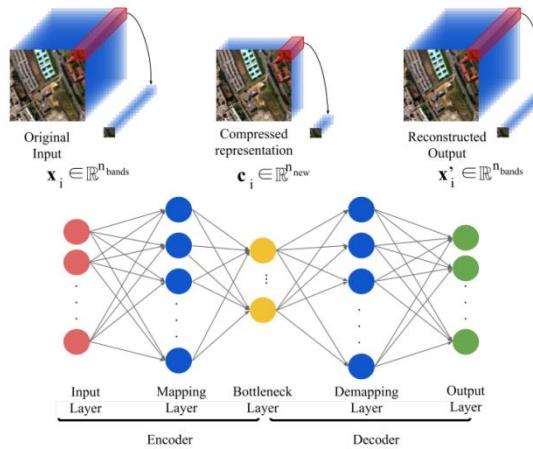


如上图所示，假设我们有大量的图片数据。在自编码器中有两个网络，其中一个叫编码器，另一个叫解码器（与我们之前讲的 Transformer 模型类似），它们是两个完全不同的网络，主要的功能是：编码器读入一张图片，输出一个向量；随后解码器将编码器输出的向量作为输入，并产生一张新的图片作为最终的输出。

我们训练的目标是希望编码器的输入跟解码器的输出越接近越好。事实上，我们可以将输入的图片和输出的图片均看成高维的向量，要判断两个图片的接近程度，我们可以计算与图片相对应的向量之间的距离作为衡量指标。我们发现，这其实跟前面讲的 Cycle GAN 模型是类似的，这样的操作也被称为重构（Re-construction）。

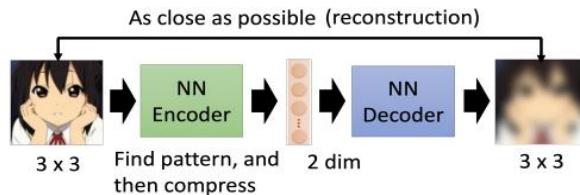
当然，在实做中，图片向量的维度比较高，我们会采取一些降维的方法，比如传统的 PCA，t-SNE 降维方法等，我们在这里并不会对这些技术进行介绍。我们只需要知道，编码器能够帮助我们实现向量的降维，在完成后续任务时，我们使用的是完成降维后的向量，如下图所

示：

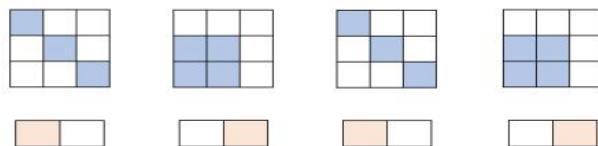


10.1.2 为什么需要自编码器？

自编码器到底好在哪里？当我们把一个高维度的图片，变成一个低维度的向量的时候，到底可以带来什么样的帮助呢？



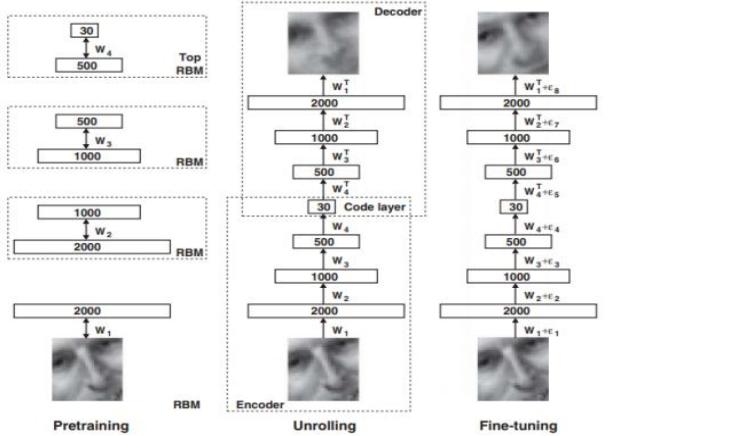
如上图所示，假设输入的图片和最终输出的图片的维度都是 3×3 ，而我们的自编码器的维度为 2，也就是说我们需要用一个二维的向量来描述一张图片。这种做法起到了一个“化繁为简”的作用，换言之，我们可以用比较少的训练数据在下游任务中取得较好的结果。



那为什么图片数据可以这样做呢？事实上对于图像来说，数据的变化其实是有限的：并不是所有的 3×3 的矩阵都是图片，可能我们收集到的图片的变化只有几种。因此在实做中，我们只需要找到数据有限的变化，就可以只用较低维度的编码器来表示较高维的数据。

10.1.3 自编码器不是一个新的概念

事实上，自编码器从来都不是一个新的概念，它具有很长的历史。深度学习之父 Hinton 早在 2006 年的 Science 论文中就提到了自编码器的概念，只是那个时候用的网络跟今天我们用的有很多不一样的地方，当时所提出的自编码器的结构如下图所示：



当时人们还并不认为深度的神经网络是能够训练起来的，换句话说就是把网络叠很多很多层，然后每一层一起训练的方法是不太可能成功的，因此人们普遍认为每一层应该分开训练，所以 Hinton 用的是一个叫做受限玻尔兹曼机（Restricted Boltzmann Machine，RBM）的技术。

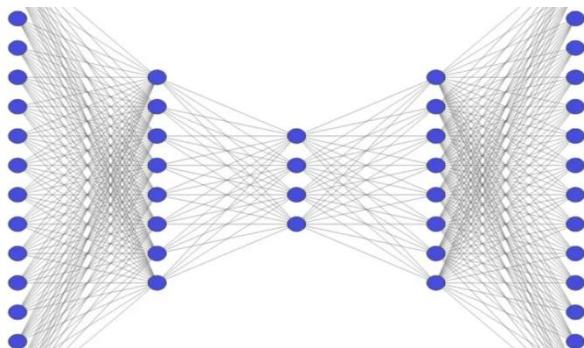
虽然 RBM 并不属于深度学习技术，而且就连 Hinton 本人都在 2012 年的论文中也“隐晦”地暗示了大家这一技术已经过时了。但它的一些思想还是有价值的，首先它可以说是比较早地提出了自编码器的思想，其次，它对于后续栈式自编码(SAE)的提出有着一定的启发。

10.2 自编码器的结构

在 10.1.3 介绍 RBM 时，我们曾说过 RBM 对于栈式自编码器（Stacked Autoencoder, SAE）有着一定的启发性意义。在本节中，我们将对不同的自编码器进行简单介绍，它们分别是：栈式自编码器，稀疏自编码器（Sparsified Autoencoder）和去噪自动编码器（Denoising Autoencoder）。事实上，还有一种自编码器叫变分自编码器（Variational Autoencoder, VAE），我们将在下一章：“更深层次的生成式模型”中进行介绍。

10.2.1 栈式自编码器

第一种自编码器是栈式自编码器，它也叫深度自编码器（Deep Autoencoder），从这个别名我们可以很自然地理解这一结构：它是在简单自动编码器的基础上增加其隐藏层的深度，从而获得更好的特征提取能力和训练效果。



如上图所示，栈式自编码器的结构是关于隐藏层对称的，通常编码器和解码器的层数是一样的，并且它们从图中直观上看，是左右对称的。这种技术称为权重捆绑，它的优点是可以使得模型的参数减半，加快训练速度，降低过拟合的风险。

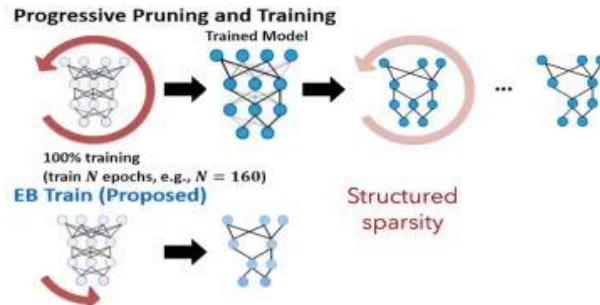
但这件事情并不是必要的，实际操作的时候，你完全可以对神经网络进行直接训练而不用保持编码器和解码器的参数一致。那这种自编码器的有什么用处呢？Hinton 分别采用了 PCA 和 Deep Autoencoder 的技术对手写数字进行编码解码，得到的结果如下图所示。我们发现，深度自编码器对于数据的还原效果比 PCA 要好。



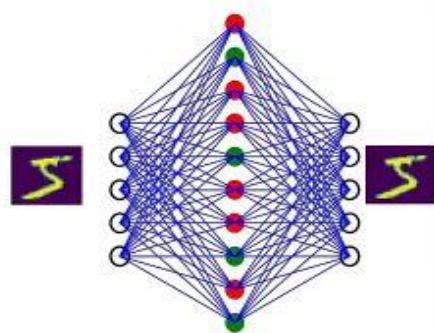
10.2.2 稀疏自编码器

与栈式自编码器一样，稀疏自编码器的英文缩写也是 SAE (Sparsified Autoencoder)，它其实也是在普通自编码器 (AE) 的基础之上进行了一些改动：即增加了“稀疏”的约束，使得神经网络在隐藏层神经元较多的情况下依然能够提取样本的特征和结构。

关于“稀疏”的解释如下：我们在第四讲已经学过了 Sigmoid 神经元的激活态（输出接近 1）和抑制态（输出接近 0），那么稀疏性限制的意思是使得神经网络中大部分的神经元的状态为抑制的一种约束或规则。

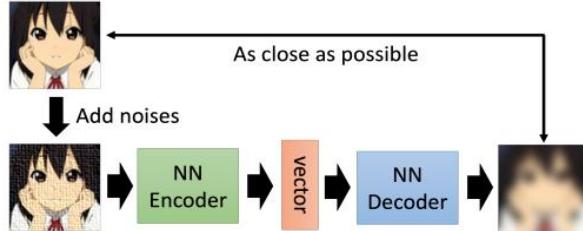


这种做法有很多优点：首先可以极大地降低计算量，其次，因为叙述编码只有少数的非零元素，相当于将一个输入样本表示为少数几个相关的特征，这样我们可以更好地描述特征，可以增强模型的可解释性。最后它可以实现特征的自动选择：我们可以通过只选择与输入样本相关性最小的特征来更好地表示输入样本，降低噪声并减轻过拟合。



10.2.3 去噪自编码器

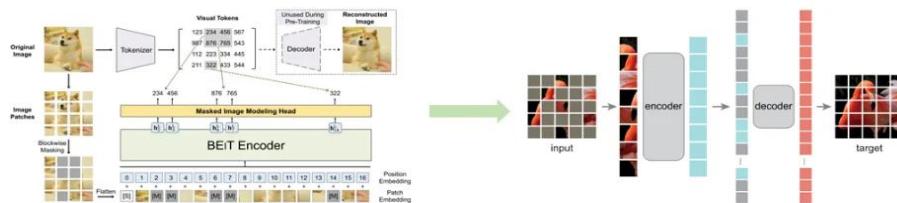
自编码器还有一个变种，叫去噪自编码器。如下图所示，它就是把原来需要输入到编码器中的图片加上一些噪声，其余流程与最原始的自编码器一致，从而试图还原原来的图片。



对于去噪自编码器的输入，我们加上了一些噪声。因此在还原图片时，我们还需要让这个自编码器学会将噪声去除。换言之，在这种自编码器中，编码器看到的是有噪声的图片，而解码器还原的目标是没有噪声的图片。

那为什么输入数据需要噪声呢？举一个简单的例子：在人类的感知过程中，某些模态的信息对结果的判断影响并不大。比如一块圆形的饼干和一块方形的饼干，在认知中同属于饼干这一类，因此形状对我们判断是否是饼干没有太大作用，也就是噪声。如果不能将形状数据去除掉，可能会产生“圆饼干是饼干，方饼干就不是饼干”的问题（过拟合）。

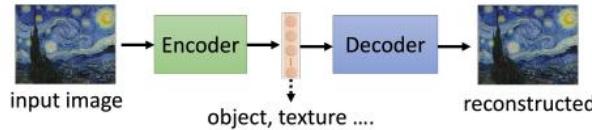
其实去噪自编码器也不算是很新的技术，至少在 2008 年的时候，就已经有相关的论文了。如果读者看今天的 BERT 模型的话，其实也可以把它看成一个去噪自编码器。我们知道 BERT 模型有一些 Masking Input，那这些 Masking 就可以看作自编码器的噪声。我们将 BERT 编码器接入一个线性模型，即解码器，它所做的事情是还原原来的句子，也就是把“填空题”盖住的地方还原回来。所以我们可以认为，BERT 其实就是一个去噪的自编码器，并且需要注意的是，解码器的部分不一定要是线性的，这样的模型叫做掩码自编码器（Masked Autoencoder，MAE）



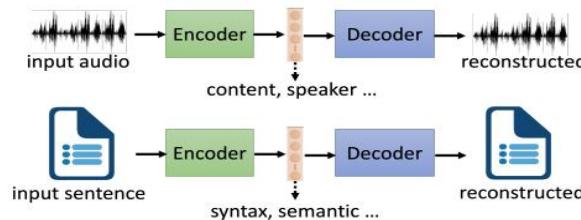
10.3 自编码器的应用

10.3.1 特征解耦

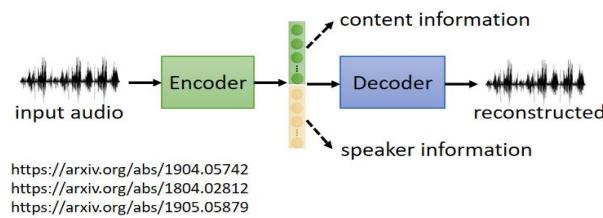
我们将在本节中将对自编码器的常见应用进行介绍。第一种常见的应用是特征解耦（Feature Disentanglement）。解耦的意思是将一堆原本纠缠在一起的东西解开。那么为什么会有特征解耦这个话题呢？回想一下自编码器所做的事，以图片数据为例，自编码器会将图片转化成编码，随后将编码还原成图片，如下图所示：



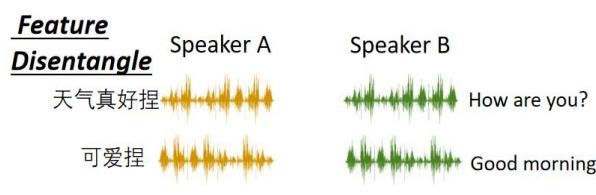
既然编码可以还原成图片，那就说明编码包含图片中的信息，比如图片中的色泽，纹理等。其实这项技术不仅可以用在图片上，也可以用在文字和语音数据中。但不管编码器被使用在哪种数据上，我们只需要知道它包含数据自身的一些特征即可。



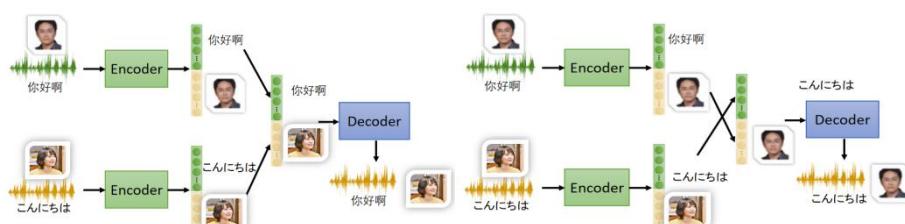
知道了这件事对我们了解特征解耦这个东西是很有帮助的。事实上，特征解耦期待做的事情是：在训练一个自编码器时，我们有办法知道编码（或者也叫表征 Representation/嵌入 Embedding）的哪些维度分别代表哪些信息。换言之，我们需要做到将特征和内容分离。



特征解耦的一个常见应用是音色转换（Voice Conversion）。我们知道，在很久以前的柯南的领结变声器中就已经做到了这一功能，但当时对于音色转换实现的效率是比较低的，因为我们需要找到两个不同的说话人念出相同的内容。而这种收集数据的方法显然有一定难度，因为有可能说话者的语言不同，无法念出相同的内容。而在特征解耦的帮助下，两个说话人可以读出不同的内容来实现音色转换。

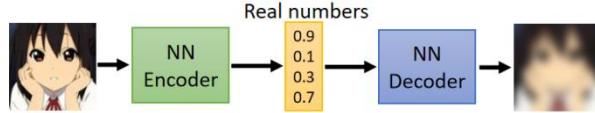


音色转换的流程如下：假如我们让李宏毅老师说一句中文，让新垣结衣说一句日文，将这两句话分别丢入编码器中，特征解耦会将这句话分为说话人特征向量（黄色部分）和说话内容向量（绿色部分）当我们李宏毅老师的说话内容向量和新垣结衣的说话特征向量相结合输入到解码器中，会得到新垣结衣说中文的一句话（如下图左所示），反之我们会得到一个李宏毅老师说的日语句子（如下图右所示）。

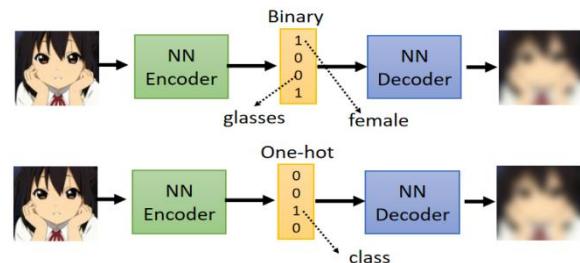


10.3.2 离散潜在表征

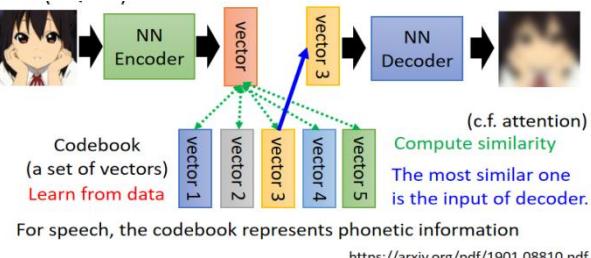
自编码器的另一个有意思的应用是离散潜在表征（Discrete Latent Representation）。到目前为止我们都假设嵌入是一个向量，它由一串实数组成。那它可不可以是一些其他东西呢，比如二进制编码等。



这当然是可以的，使用二进制编码作为嵌入向量的好处在于，可以用 0/1 代表一个指定维度的特征有无。当然我们也可以用其他方式来表示这个嵌入，比如可以采用独热向量（One-hot vector）的形式来表示嵌入，这或许可以让机器在没有标注数据的情况下自动学会分类。



在这些离散的表征技术中，最知名的就是矢量量化变分自编码器（Vector Quantized Variational Auto-Encoder, VQVAE）。它的运作原理是：输入一张图片，然后编码器输出一个向量。接下来会有一个 Codebook，它是由一排向量组成的。我们需要计算编码器输出的向量与 Codebook 中的向量的相似度（与自注意力机制类似），如下图所示，我们将相似度最大的向量从 Codebook 中拿出来作为解码器的输入，并最终输出一张图片。

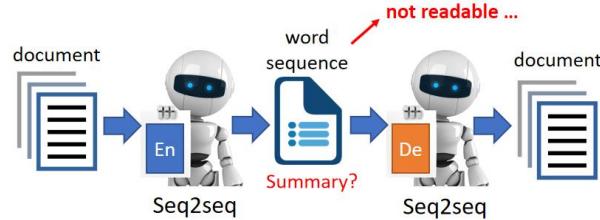


这种做法的好处是可以产生离散的潜在表征，也就是说解码器的输入一定是 Codebook 种的其中一个向量，即解码器输出的可能性是有限的（与 Codebook 中的向量个数相等）。这样的技术通常可以应用在语音中，因为不管是中文还是英文，汉语拼音和英文音标发音的可能性都是有限的，在语音任务中，Codebook 可以学到一个语言最基本的发音单位（Phonetic），而这个 Codebook 里面每一个向量，它就对应到某一个发音如下图所示。

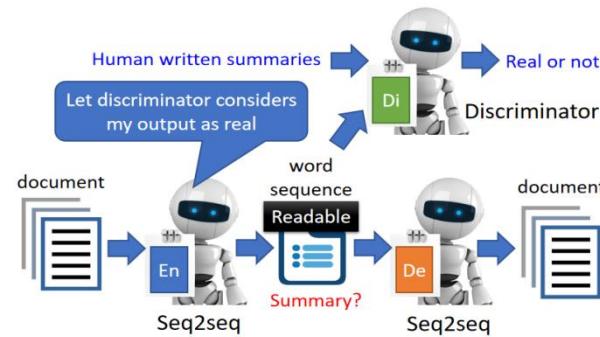
Consonants	p, b, t, d, k, g, tS, dZ, f, v, T, D, s, z, S, Z, h, m, n, N, l, r, w, j, 5 (-dark-l)
Vowels	i:, I, e, {, V, A:, Q, O:, U, u:, @, 3:, aI, OI, eI, @U, aU, e@, I@, U@
Silence	-

当然我们还可以有更疯狂的想法：比如这个表征一定要是向量的形式吗，能不能是一段文字？答案是可以的。如下图所示，我们可以做一个文字的自编码器，这其实与语音/图片的自编码器没什么不同，只不过我们将嵌入从原先的向量变成一段文字，它的好处在于，也许

这串文字就是文章的摘要。如果把一篇文章丢到编码器里面，它输出一串文字，而这串文字，可以通过解码器还原成原来的文章，那代表说这段文字，是这篇文章的精华，即最关键的内容，或者说就是摘要。显然，在这个问题中，编码器和解码器都应该是序列到序列的模型。



从上面的分析我们可以猜测，如果这个训练是可行的，那机器实现自己做摘要这件事将会变得非常简单。实际上，实验表明这样的训练是有难度的，因为这两个编码器和解码器之间会“发明”自己的“暗号”，而这个“暗号”是我们人类看不懂的，因此就算它产生的“暗号”是输入文章的摘要，我们也无法分辨。



那这个时候我们应该怎么办呢？我们可以在编码器和解码器之间加上一个判别器，也就是利用 GAN 的方法进行训练。因为判别器可以看过人类写过的句子，因此我们期待编码器和解码器之间的文字要像 GAN 一样，试图骗过判别器，这样或许可以产生一个人类能看得懂的一篇摘要。

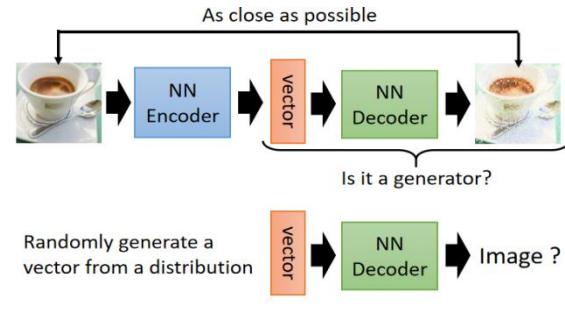
那这个网络该怎么训练呢？我们发现，这个网络和 CycleGAN 很像，事实上，这个网络的本质就是 CycleGAN，只是我们是从自编码器的角度来理解而已。而对于这个 CycleGAN 的网络，我们在之前有所提及，直接利用 RL “硬做” 即可。

10.3.3 自编码器的其它应用

自编码器其实还可以做更多的应用，前面主要都在讲编码器，其实解码器也有一定的作用。

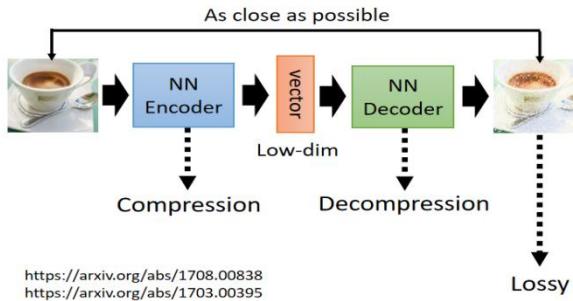
首先是应用在生成器（Generator）上，我们在离散潜在表征中讲过一个机器摘要的例子，它其实本质上就是 CycleGAN。事实上自编码器在生成式模型中还有其它的应用。如下图所示：把解码器拿出来就相当于一个生成器。而生成器就是要输入一个向量，然后输出一个东西，比如说一张图片，而解码器的原理也是类似的，因此解码器也可以当做一个生成器来用。我们可以从一个已知的分布（比如高斯分布）中采样一个向量喂给解码器，然后看看它能不能输出一张图。实际上在前面讲到生成模型的时候有提到除了 GAN 以外的另外两种生成模型，其中一个就叫做变分自编码器（VAE，Variational Auto-Encoder）。顾名思义显然可以看出它其实跟自编码器有很大的关系，实际上它就是把自编码器中的解码器拿出来当生成器

来用，那实际上它还有做一些其他的事情，感兴趣的读者可以自行研究。

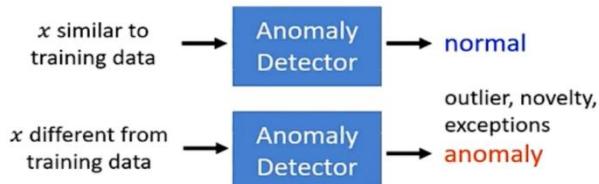


With some modification, we have **variational auto-encoder (VAE)**.

此外，自编码器还可以应用在图像压缩中。当我们处理图片时，如果图片太大了，回想我们之前讲过的自编码器原理：图片这种高维矩阵可以由一个低维自编码器进行重构。我们完全可以把这个向量看作是一个压缩的结果。所以编码器做的事情，就是压缩，对应解码器做的事情就是解压缩。只是这个压缩与 JPEG 图片一样，会发生失真现象，因为在训练自编码器的时候我们没有办法做到输入的图片跟输出的图片是完全一模一样的。

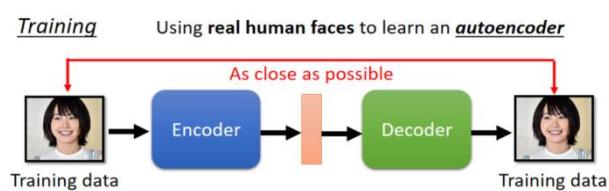


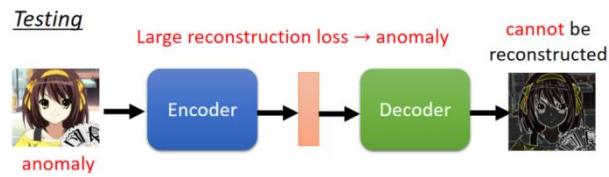
最后一个应用是近些年来流行起来的异常检测（Anomaly Detection），即判断输入与训练集中的数据是否相似。（这个相似没有明确的定义，根据应用情景会有所不同）。



根据上述定义描述，我们很可能将这类问题归为二分类问题，但实际上这样处理是不可行的。因为在通常情况下大部分收集到的数据都是正常的，而异常数据往往只有很小的一部分，所以这不是一个简单的二分类问题，此时自编码器就有了“用武之地”。

以人脸检测为例，在训练时我们搜集了很多人脸的图像，将它们依次经过编码器和解码器，使得自编码器具备合成真人图像的能力。如下图所示，在测试的时候，如果输入的是二次元图片而不是真人图片，那么在经过编码器和解码器后是合成不了正常的图片的。只有输入真人图像，才能有一个正常的合成。





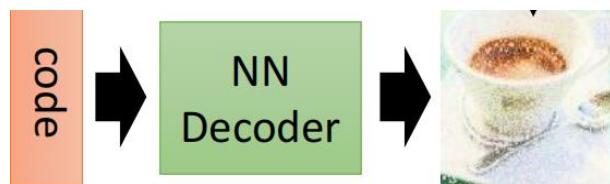
以上就是对于自编码器的简单介绍，当然自编码器还有一些其它应用，感兴趣的读者可以自行研究。

11 其它生成式模型

在本章我们将进一步对生成式模型进行介绍。我们在第 7 讲曾经提到过三种经典的生成式模型：GAN，VAE 和 FLOW，并且在当时我们对于 GAN 进行了较为详细的介绍。在本章中，我们将对其它两种经典的生成式模型 VAE 和 FLOW 进行介绍。此外，在本章的最后，我们将对融合了三种经典的生成式模型的一个语音合成模型——VITS 模型进行介绍。

11.1 变分自编码器（VAE）

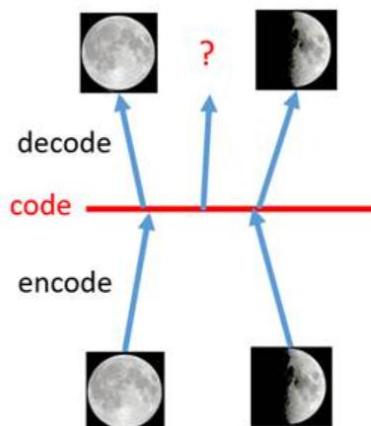
我们首先来介绍一种基于变分思想的深度学习的生成模型——Variational autoencoder，简称 VAE。在上一讲中我们曾对自编码器（AE）进行了一定的了解。我们知道，在处理图片数据时，可以将一个随机的编码输入到解码器中，期待输出一张完整的图像。



但实际上这么做的效果不一定好，有一种方法叫变分自编码器（VAE），它的结构与自编码器十分相像，只不过在中间加入了一些小技巧能够使图像重构的工作变得更好。编码器不直接输出编码，而是先输出两个向量，它们的维度与编码的维度一样。

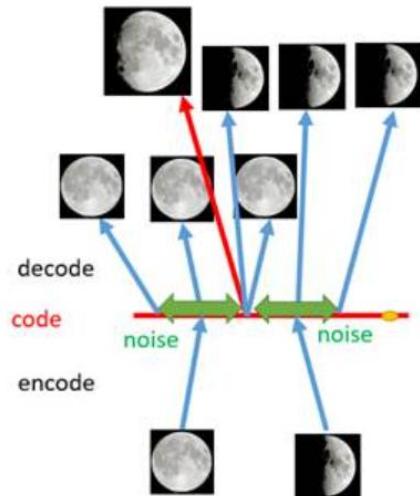
11.1.1 变分自编码器的引入

我们首先从一个直观的方式来讨论需要变分自编码器的原因。如下图所示，假设两张训练图片：一张是全月图，一张是半月图，经过训练我们的自编码器模型已经能无损地还原这两张图片。接下来，我们在编码空间上两张图片的编码点中间处取一点，然后将这一点交给解码器，我们希望新的生成图片是一张清晰的图片（类似 3/4 全月图）。但实际的结果是，生成图片是模糊且无法辨认的乱码图。

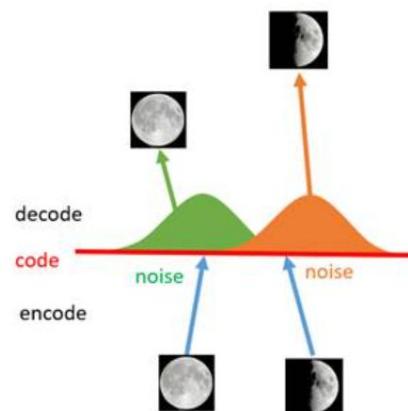


对于上述问题，一个合理的解释是，因为编码和解码的过程使用了深度神经网络，这是一个非线性的变换过程，所以在编码空间上点与点之间的迁移是非常没有规律的。那么应该

如何解决这样的问题呢？我们可以引入噪音，使得图片的编码区域得到扩大，从而掩盖掉失真的空白编码点。如下图所示，我们现在给两张图片编码的时候加上一点噪音，使得每张图片的编码点出现在绿色箭头所示范围内，于是在训练模型的时候，绿色箭头范围内的点都有可能被采样到，这样解码器在训练时会把绿色范围内的点都尽可能还原成和原图相似的图片。然后我们可以关注之前那个失真点，现在它处于全月图和半月图编码的交界上，于是解码器希望它既要尽量相似于全月图，又要尽量相似于半月图，因此的还原结果就是两种图的折中（3/4 全月图）。



由此我们发现，给编码器增添一些噪音，可以有效覆盖失真区域。不过这还并不充分，因为在上图的距离训练区域很远的黄色点处，它依然不会被覆盖到，仍是个失真点。为了解决这个问题，我们可以试图把噪音无限拉长，使得对于每一个样本，它的编码会覆盖整个编码空间，不过我们得保证，在原编码附近编码的概率最高，离原编码点越远，编码概率越低。在这种情况下，图像的编码就由原先离散的编码点变成了一条连续的编码分布曲线，如下图所示。



那么上述的这种将图像编码由离散变为连续的方法，就是变分自编码的核心思想。

11.1.2 变分自编码器的模型架构

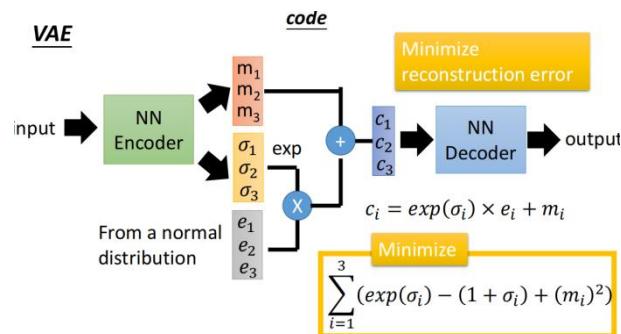
我们在本节一开始说过，变分自编码器通过对自编码器的网络结构做了一些具有技巧性

的改动，即编码器通过输出两个向量来取代原先的编码。

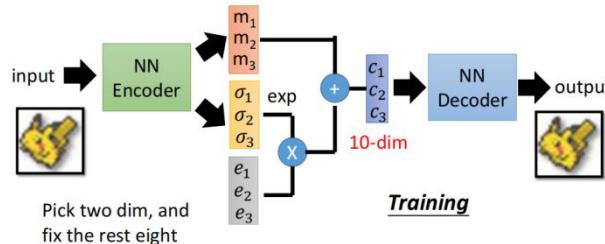
用一个具体的例子来说明上述流程：假设输出的编码是三维的，那编码器生成的两个向量也是三维的，不妨记为： $\mathbf{m} = (\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3)$ 和 $\boldsymbol{\sigma} = (\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2, \boldsymbol{\sigma}_3)$ 。此外我们还需要用正态分布（normal distribution）来生成另外一个向量，记为 $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$ ，接下来我们把 $\sigma_1, \sigma_2, \sigma_3$ 分别进行指数 \exp 运算，并且与 e_1, e_2, e_3 相乘，最后加上 m_1, m_2, m_3 ，得到最终的向量 $\mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$ ，其中对于 $i \in \{1, 2, 3\}$ ，均有：

$$\mathbf{c}_i = \exp(\sigma_i) \times \mathbf{e}_i + \mathbf{m}_i$$

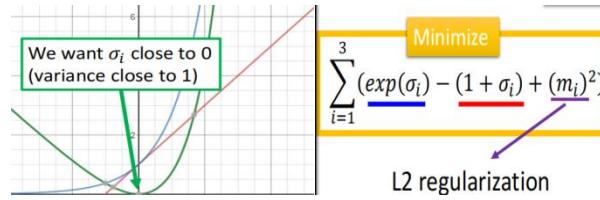
VAE 的做法是将上面描述的向量 \mathbf{c} 作为解码器的输入，以此希望解码器能实现最小化重构误差（Minimize reconstruction error）的功能。此外，对于这个重构误差，还需要添加如下图右下角所示的这一项。



我们来具体展开分析一下上面的表达式，向量 \mathbf{m} 代表的是原先编码器的输出编码，向量 \mathbf{c} 代表的是加上噪声之后的编码，解码器需要根据加上噪声之后的编码对输入图片进行重构。向量 $\boldsymbol{\sigma}$ 代表的是噪声的方差，而我们将它经过指数运算的原因仅仅是为了确保方差值是正数而已，向量 \mathbf{e} 是从一个正态分布中采样出来的值。

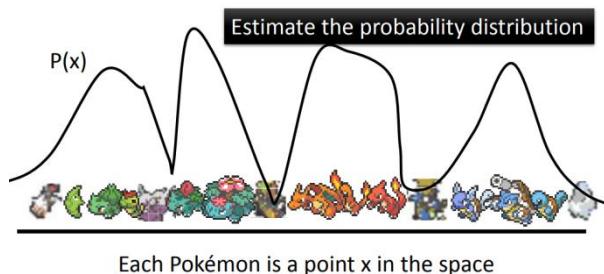


那么为什么我们不能只用上面的运算作为最小化重构误差呢？实践表明，经过噪声的编码重构出来的图片效果并不能达到预期效果，如上图所示。我们发现，噪声的方差是从编码器中产生的，也就是说在训练时噪声的方差是机器自己学到的参数。举个形象的例子，假设老师让同学们自己给自己打分，那同学们肯定给自己打满分，而编码器也是这样的，因为它肯定希望噪音对自身生成图片的干扰越小，那编码器干脆就直接将方差设置为 0 就可以达成目的了。因此，第二个损失函数首先有着限制编码器走向“极端赋值”的功能：我们可以观察第二部分的损失函数的数学表达式发现， $\exp(\sigma_i) - (1 + \sigma_i)$ 在 σ_i 为 0 时取最小值，因此就可以避免 σ_i 被赋值成负无穷大，其次，函数中的 \mathbf{m}_i^2 起到一个 L2 正则化的功能，从而避免过拟合。

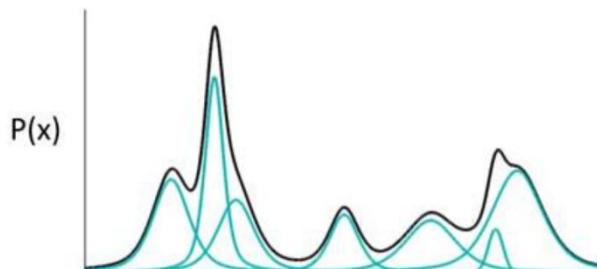


11.1.3 变分自编码器的训练原理

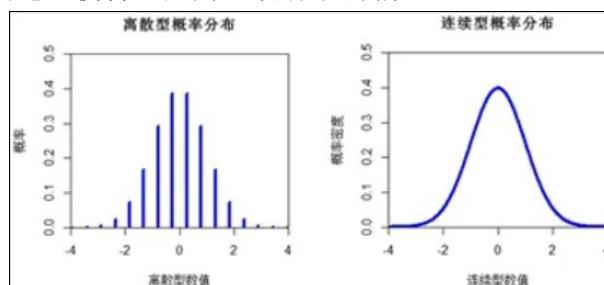
我们回归到我们原先需要做的事情——比如生成宝可梦的图片，我们可以将每一张图片想象成是高维空间中的一个点 x ，我们现在需要做的事是估计这个点的概率分布： $P(x)$ ，如下图所示：



那么我们该如何得到概率分布呢？我们可以用高斯混合模型（Gaussian Mixture Model），这个技术在语音中经常被使用。事实上，上图中的黑色曲线是由多个高斯分布叠加起来，并且这一理论是经过严格的数学证明的，也就是说，我们可以将 $P(x)$ 写成 $\sum_m P(m)P(x|m)$ 的形式，如下图所示，蓝色曲线代表一个个的高斯分布：



于是我们可以利用这一理论模型去考虑如何给数据进行编码。一种最直接的思路是，直接用每一组高斯分布的参数作为一个编码值实现编码。但事实上，这种编码方式是非常简单粗暴的，它对应的是我们之前提到的离散的、有大量失真区域的编码方式。于是我们需要对目前的编码方式进行改进，使得它成为连续有效的编码。



现在我们的编码换成一个连续变量 z ，我们规定 z 符合正态分布（实际上并不一定要选用，其他的连续分布都是可行的），对于每一个采样 z ，会有两个函数 μ 和 σ ，满足如下公式：

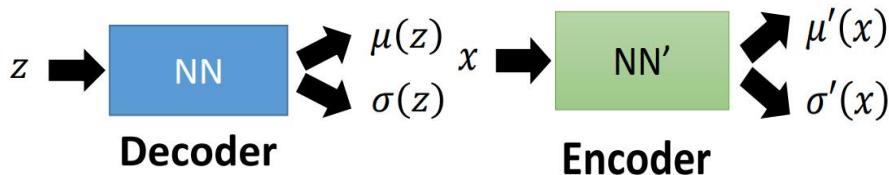
$$P(X) = \int P(z)P(X|z)dz$$

$$z \sim N(\mathbf{0}, \mathbf{1}), \quad x|z \sim N(\mu(z), \sigma(z))$$

对于这个问题，我们的目标函数需要利用极大似然估计，使得概率分布 x 极可能大，目标函数如下：

$$L = \sum_x \log P(x)$$

由于 $P(x)$ 的数学表达式比较复杂，我们需要借助两个神经网络来帮助我们求解：第一个神经网络用来求解 μ 和 σ 两个函数，这等价于求解 $P(x|z)$ ，我们将这个网络叫做解码器，与解码器对应的是编码器，因此我们需要另外一个概率分布 $q(z|x)$ ，其中 $z|x \sim N(\mu'(x), \sigma'(x))$ ，如下图所示：



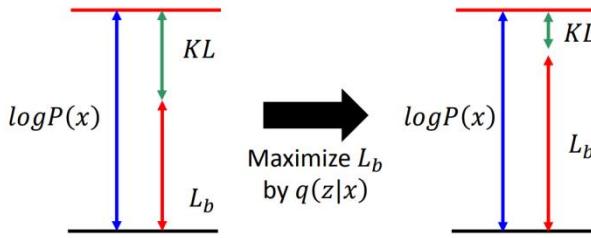
由于 $q(z|x)$ 的分布是任意的，我们可以将损失函数改写成以下形式：

$$\begin{aligned} \log P(x) &= \int q(z|x) \log P(x) dz = \int q(z|x) \cdot \log \left(\frac{P(z, x)}{P(z|x)} \right) dz = \int q(z|x) \cdot \log \left(\frac{P(z, x)}{q(z|x)} \cdot \frac{q(z|x)}{P(z|x)} \right) dz \\ &= \int q(z|x) \cdot \log \left(\frac{P(z, x)}{q(z|x)} \right) dz + \int q(z|x) \cdot \log \left(\frac{q(z|x)}{P(z|x)} \right) dz \end{aligned}$$

对于上式的第二项，其实就是 $q(z|x)$ 和 $P(z|x)$ 这两个概率分布所对应的 KL 散度（KL Divergence），它代表两个分布相近的程度，类似于距离，因此它是非负的，于是我们的目标函数有一个下界：

$$L_b = \int q(z|x) \cdot \log \left(\frac{P(z, x)}{q(z|x)} \right) dz$$

于是我们的目标变成了寻找 $P(x|z)$ 和 $q(z|x)$ ，来使得上述下界最大。

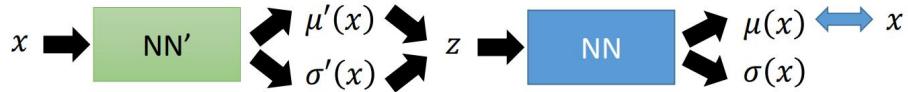


那么为什么我们需要多寻找一项 $q(z|x)$ 呢？因为如果我们只找 $P(x|z)$ 这一项来最大化下界 L_b 的话，我们不知道 L 与 L_b 之间的关系，可能会出现 L_b 上升但是 L 反而下降的情况，引入 $q(z|x)$ 这一项可以解决这个问题：因为首先 $q(z|x)$ 和 $\log P(x)$ 是没有关系的，所以不管 $q(z|x)$ 如何变化， $\log P(x)$ 是不变的，此时如果我们最大化 L_b 的话，会使得 L_b 和 L 越来越接近，而这两项的接近程度就是 KL 散度，当 KL 散度为 0 时，有： $L = L_b$ ，此时我们再增加 L_b 的话，就能确保 L 一定也会增大。

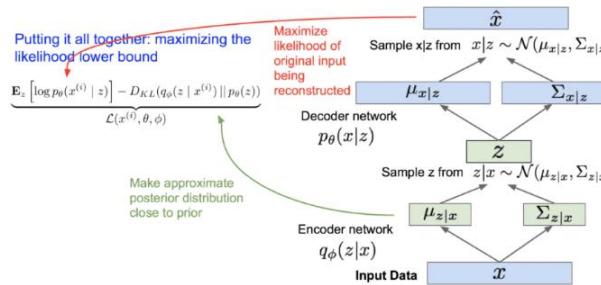
我们继续简化 L_b 如下：

$$L_b = \int q(z|x) \cdot \log \left(\frac{P(x|z)P(z)}{q(z|x)} \right) dz = \int q(z|x) \cdot \log \left(\frac{P(z)}{q(z|x)} \right) dz + \int q(z|x) \cdot \log P(x|z) dz$$

我们发现，第一项可以看成 $P(z)$ 和 $q(z|x)$ 的 KL 散度表达式，即： $\text{KL}(q(z|x)||P(z))$ ，当我们将这一散度最小化时，即代表将 $\sum(\exp(\sigma_i) - (1 + \sigma_i) + m_i^2)$ 最小化，而对于 L_b 中的 $\int q(z|x) \cdot \log P(x|z) dz$ 这一项，我们可以将它写成数学期望的形式： $E_{q(z|x)}[\log P(x|z)]$ ，这相当于从 $q(z|x)$ 这个分布中生成数据，使得 $\log P(x|z)$ 最大，这也就是自动编码器所做的事，即让输入和输出越接近越好。

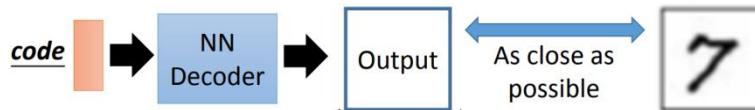


综上所示，VAE 的训练流程如下图所示：

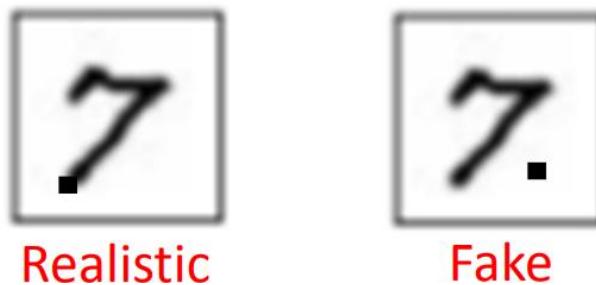


11.1.4 VAE v. s. GAN

既然我们已经在前面介绍了生成对抗网络 (GAN)，我们就可以来比较一下 VAE 和 GAN 的性能。事实上，VAE 有一个很严重的问题：它学习的目标是产生一张与数据集中的某张图片尽可能接近的图片，但它并没有真正去学如何产生一张看起来真实的图片。



我们可以用类似于最小均方误差 (MSE) 的方法来评估 VAE 产生的图片和数据集中图片的相似度。假设我们输出的图片和真的图片有一个像素点的差距，这个像素点在不同位置其实会有不同的结果，但是 VAE 无法看出来差异，也没有办法产生新的图片，而 GAN 可以很好地避免这一点。



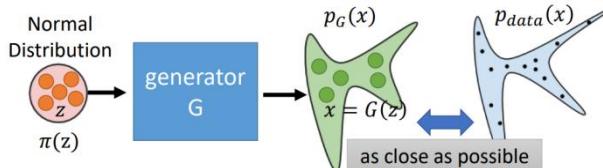
但是 VAE 也并非一无是处，因为就生成式模型而言，VAE 是一种“有迹可循”的方法，因为它使得查询推断成为了一种可能，如此能推断出类似于 $q(z|x)$ 的分布，这些东西对其他任务来说会是很有用的特征表征。

11.2 基于流 (Flow-based) 的生成式模型

我们先来讨论一下在之前介绍过的两种生成式模型：GAN 和 VAE。事实上，无论是 GAN 还是 VAE，它们都有一定的问题：GAN 的问题是难以训练，而 VAE 的问题是优化的是下界，并非最终目的，因此，我们引出了第三种生成式模型——基于流的生成式模型（Flow-based Generative Model）。

11.2.1 流模型的神奇之处

我们在第 8 讲曾讲过生成式对抗网络 (GAN) 由两个神经网络组成，一个是生成器 G ，另一个是判别器 D 。其中，生成器 G 定义了一个概率分布 P_G ，如下图所示：假如我们有一个生成器 G ，那么我们输入一个 z 就可以得到一个输出 x ，而输入 z 可以看成从简单的正态分布中采样出来的，而最终得到的 x 分布则可以认为和生成器 G 相关。我们可以将这个分布定义为： $P_G(x)$ ，我们可以将 x 称为观测变量，将 z 称为隐变量，其对于样本的生成是至关重要的。因此可以认为观测变量 x 的真实分布为 $P_{data}(x)$ 。



那么对于生成器 G ，我们的训练目标就是希望 $P_G(x)$ 和 $P_{data}(x)$ 越接近越好，那么对于这个极大似然估计的求解同样也可以转化为最小化 KL 散度（距离）的优化问题。对于这个问题，我们在 GAN 中曾讲过这是很难求解的，而基于流的生成式模型（Flow-based Generative Model）最厉害的地方在于它可以直接优化目标函数，而这是需要非常扎实的数学功底的，我们将在下一小节对 Flow-based 模型所需要的数学理论进行介绍。

11.2.2 数学背景

我们首先对 Flow-based 模型所用到的数学知识进行介绍。首先是雅可比矩阵（Jacobian Matrix），对于两个向量 $z = [z_1, z_2]^T$ 和 $x = [x_1, x_2]^T$ ，我们可以构造二者之间的函数关系：

$$x = f(z), z = f^{-1}(x)$$

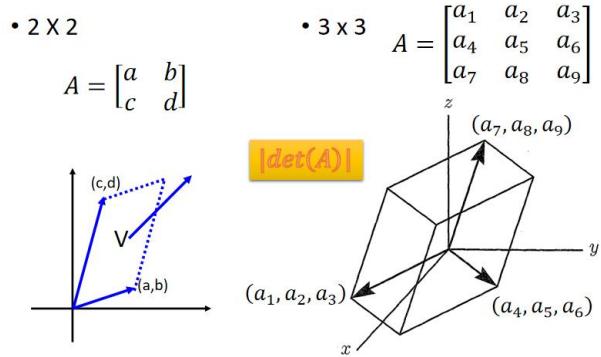
在向量微积分中，雅可比矩阵是一阶偏导数以一定方式排列成的矩阵，其行列式称为雅可比行列式，如下所示：

$$J_f = \begin{array}{c|cc} \text{input} & & \\ \hline \partial x_1 / \partial z_1 & \partial x_1 / \partial z_2 \\ \partial x_2 / \partial z_1 & \partial x_2 / \partial z_2 \end{array} \parallel \text{output}$$
$$J_{f^{-1}} = \begin{bmatrix} \partial z_1 / \partial x_1 & \partial z_1 / \partial x_2 \\ \partial z_2 / \partial x_1 & \partial z_2 / \partial x_2 \end{bmatrix}$$

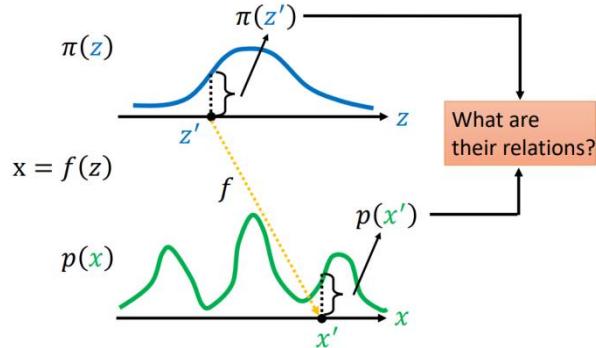
对于雅可比矩阵，存在一个重要的性质如下：

$$J_f \cdot J_{f^{-1}} = I, \det(J_f) \cdot \det(J_{f^{-1}}) = 1$$

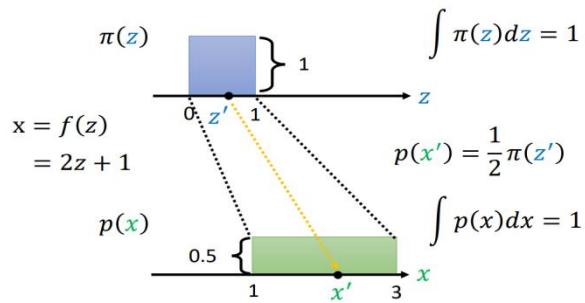
即它们互为逆矩阵，且行列式也存在互为倒数的关系。而行列式还有另外一个含义，就是将矩阵的每一行都当成一个向量，并在对应维度的空间中展开，那么形成的那个空间的“体积”就是行列式的绝对值，如下图的二维的面积和三维的体积：



根据前面的描述，我们已知了 z 的分布，假设当前也知道了 x 的分布，那么我们想要的是求出来生成器 G ，或者说求出来怎么从 z 的分布转换到 x 的分布，如下图所示：



举一个最简单的例子：假设当前 z 满足的分布为一个 0 到 1 之间的均匀分布，而 z 和 x 之间的关系已知为 $x = f(z) = 2z + 1$ ，那么就可以得到下面的图形。而由于两者都是概率分布，因此两者的积分都应该为 1（面积相同），因此可以解出来 x 的分布对应的高度为 0.5。

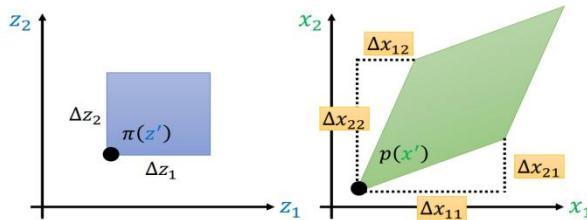


我们可以将这个问题推广到更复杂的情况，但是原理都是一样的，即两个概率分布的面积相同（为 1），我们可以在某个维度上的 z' 处取一个增量 Δz ，那么我们可以将其映射到 x 的分布上有 x' 和 Δx ，当 Δz 足够小时，可以使得在该段之内的 $p(z)$ 都相同，与之对应的 $p(x)$ 也是相同的，我们可以根据“面积相等”的原理得到如下公式：

$$p(x') = \pi(z') \cdot |\frac{dz}{dx}|$$

对于上面的公式，我们可以将其推广到二维的情况，那么“面积相等”的原理也同样被

推广成了“体积相等”的原理，如下图所示：



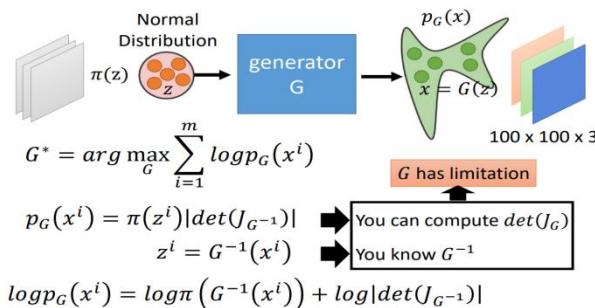
$$p(\mathbf{x}') \left| \det \begin{bmatrix} \Delta x_{11} & \Delta x_{21} \\ \Delta x_{12} & \Delta x_{22} \end{bmatrix} \right| = \pi(\mathbf{z}') \Delta z_1 \Delta z_2$$

我们发现，上述公式可以归纳成一个通式，即对于两个变量 x 和 z ，当二者之间存在一个函数关系时，它们的概率分布之间也存在着一个雅克比矩阵的行列式的绝对值作为系数的关系，这就是著名的随机变量的变量替换定理(Change of Variable Theorem)。

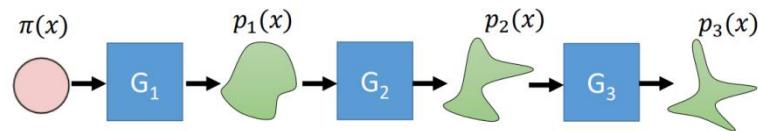
$$\begin{aligned} p(\mathbf{x}') |\det(J_f)| &= \pi(\mathbf{z}') \\ p(\mathbf{x}') &= \pi(\mathbf{z}') \left| \frac{1}{\det(J_f)} \right| \\ p(\mathbf{x}') &= \pi(\mathbf{z}') |\det(J_{f^{-1}})| \end{aligned}$$

11.2.3 流模型的训练

经过上面的各种推导，我们可以将目标函数进行转换：



此时，我们将目标函数转换成了上图中最下方的表达式，但在实做中，计算雅可比矩阵行列式的操作可能是非常耗时的，此外我们还需要计算生成器 G 的逆 G^{-1} ，这包含着一个潜在要求，即输入的维度和输出的维度必须是一样的，因此我们要巧妙地设计网络的架构，使其能够方便计算雅可比矩阵的行列式和生成器的逆。

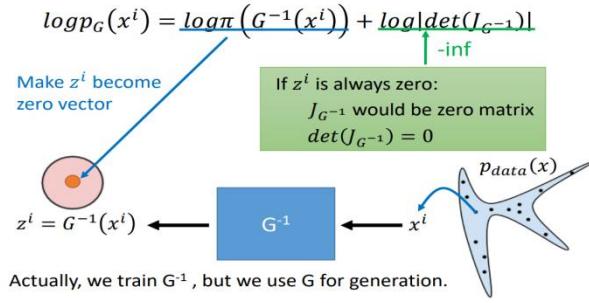


在实际的 Flow-based Model 中， G 可能不止一个。因为上述的条件意味着我们需要对 G 加上种种限制。那么单独一个加上各种限制就比较麻烦，我们可以将限制分散于多个 G ，再通过多个 G 的串联来实现，这也是称为流形的原因之一。

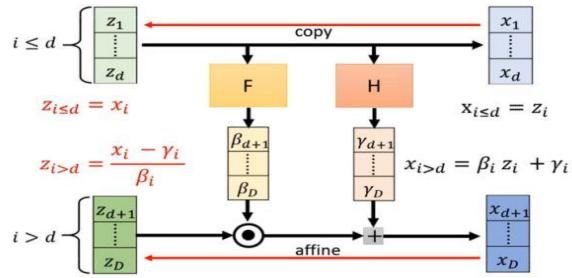
因此，我们需要最大化的目标函数变成了如下形式：

$$\log p_G(\mathbf{x}^i) = \log \pi(\mathbf{G}^{-1}(\mathbf{x}^i)) + \log |\det(\mathbf{J}_{\mathbf{G}^{-1}})|$$

可以发现上述要最大化的目标函数中只有 G^{-1} ，因此在训练时我们可以只训练这一项，因为我们在训练的时候就会从分布中采样得到 \mathbf{x} ，然后代入得到 \mathbf{z} ，其接受 \mathbf{x} 作为输入，输出为 \mathbf{z} ；而在训练完成后就将其反过来，接受 \mathbf{z} 作为输入，输出为 \mathbf{x} 。



事实上，我们在训练时还可以对计算进行简化，这种技术叫耦合层（Coupling Layer），它在 NICE (<https://arxiv.org/abs/1410.8516>) 和 Real NVP (<https://arxiv.org/abs/1605.08803>) 的论文中被使用，其思想如下：

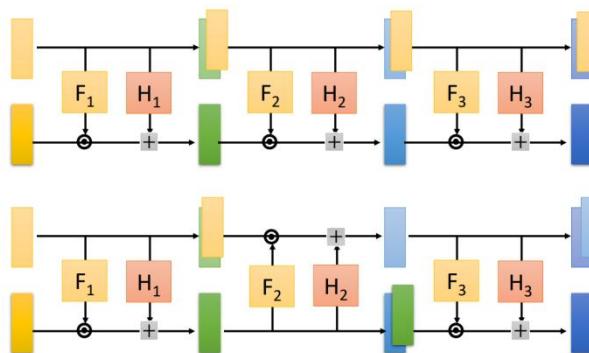


我们假设 z 和 x 之间满足某种关系，进行向量变换，如上图所示，黑色部分是正向的过程，但我们在训练时实际上只需要训练 G^{-1} 即可，因此我们只需要红色部分的逆向过程即可，而至于 F 和 H 两个函数，设置成多复杂都和我们的求解没有关系了。

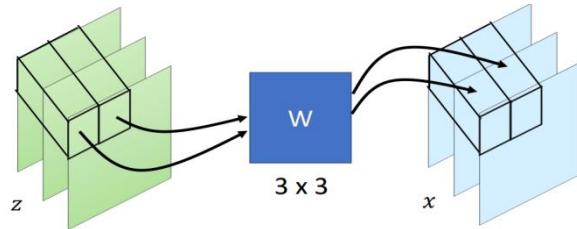
因此满足了上述关系之后，雅克比矩阵的计算就变得很方便了：

J_G $\begin{matrix} z_1 & \dots & z_d & z_{d+1} & \dots & z_D \end{matrix}$ $\begin{matrix} x_1 \\ \vdots \\ x_d \\ x_{d+1} \\ \vdots \\ x_D \end{matrix}$	$\det(J_G)$ $= \frac{\partial x_{d+1}}{\partial z_{d+1}} \frac{\partial x_{d+2}}{\partial z_{d+2}} \dots \frac{\partial x_D}{\partial z_D}$ $= \beta_{d+1} \beta_{d+2} \dots \beta_D$
$\begin{matrix} I \\ (\text{Identity}) \end{matrix}$ $\begin{matrix} O \\ (\text{zero}) \end{matrix}$ $I \text{ don't care.}$	$x_{i > d} = \beta_i z_i + \gamma_i$

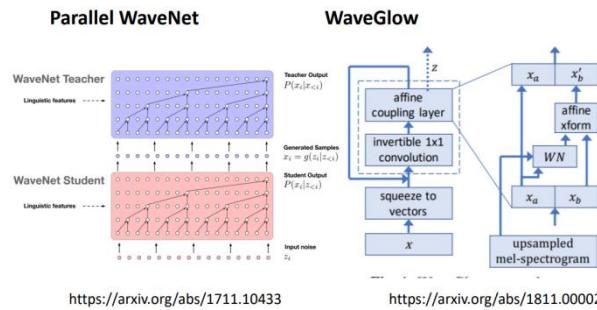
接下来我们就可以将多个耦合层串在一起，但如果正向直接串的话就会发现前 d 维度的值是直接拷贝的，从头到尾都相同，这并不是我们想要的结果，我们不是希望前 d 维度的值一直保持不变，因此我们的多个耦合层中会出现“正向串”（copy）和“反向串”（affine）均出现的情形，如下图所示：



对于这个 Flow 模型，除了耦合层之外，还有一个 1×1 的卷积层（Convolution Layer）使得它能获得很好的生成效果，如下图所示我们有输入 z 和输出 x ，由于我们需要产生图像，我们的输入中会有一系列的像素，而在图像生成的过程中，每一个像素都会与一个权重 3×3 的 W 进行相乘，得到输出 x 中与输入对应的像素，这一技术在 GLOW (<https://arxiv.org/abs/1807.03039>) 中被广泛应用。



我们对 Flow-based 的介绍就到此为止了，当然 Flow-based 生成式模型还有更多的应用，感兴趣的读者可以自行探究，下图给出了一些其它的应用及其对应的论文链接。

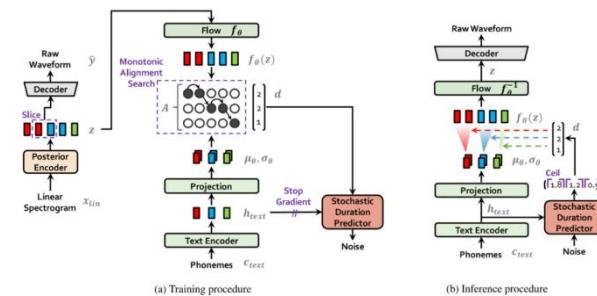


11.3 生成式模型综合应用——VITS 模型

在本章的最后，我们将介绍一个结合了三大生成式模型（GAN, VAE, FLOW）的模型，它是用于语音合成带有对抗学习的条件变分自编码器（Variational Inference with adversarial learning for end-to-end Text-to-Speech，简称 VITS）。

VITS 是一种结合变分推理（variational inference）、标准化流（normalizing flows）和对抗训练的高表现力语音合成模型。它通过隐变量而非频谱串联起来语音合成中的声学模型和声码器，在隐变量上进行随机建模并利用随机时长预测器，提高了合成语音的多样性，输入同样的文本，能够合成不同声调和韵律的语音。

VITS 主要分为三个模块：第一个是条件变分自编码器（Conditional VAE），它是我们之前所讲的 VAE 的变种，第二个是从变分推断中产生的对齐估计，第三个模块是生成对抗训练，其总体架构如下图所示：



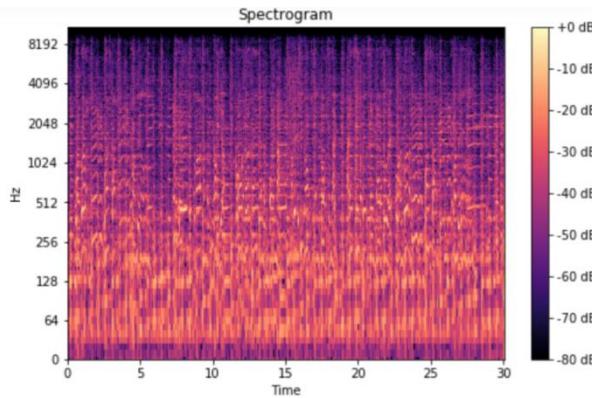
接下来，我们将对这三个模块分别进行介绍。

11.3.1 变分推断

我们在之前讲到 VAE 训练时曾提到目标函数有一个下界，因此，在 VITS 的第一个模块条件 VAE 中也存在着一个下界，称为最大化变分下界，也即 ELBO (Evidence Lower Bound)，它是一个不可解的对数似然，如下所示：

$$\log p_{\theta}(x|c) \geq E_{q_{\phi}(z|x)}[\log p_{\theta}(x|z) - \log \frac{q_{\phi}(z|x)}{p_{\theta}(z|c)}]$$

其中， $p_{\theta}(z|c)$ 表示给定条件 c 的潜在变量 z 的先验分布， $p_{\theta}(x|z)$ 是数据点 x 的似然函数。 $q_{\phi}(z|x)$ 是一个近似的后验分布。训练的损失就是 ELBO 的负值（最小化 ELMO 的负值也就是最大化 ELBO），它可以看作是重构损失 $-\log p_{\theta}(x|c)$ 和 KL 散度 $\log q_{\phi}(z|x) - \log p_{\theta}(z|c)$ 之和。



而至于 VITS 中的重构损失，它使用的是梅尔频谱而非原始波形，这是因为 VITS 在训练时实际还是会生成梅尔频谱以指导模型的训练：

$$L_{recon} = ||x_{mel} - \widehat{x}_{mel}||_1$$

至于 KL 散度的部分，其先验编码器的输入条件 c 由从文本中提取的音素 c_{text} 以及音素和潜在变量之间的对齐 A 组成，其中对齐矩阵 A 是一个尺寸为 $|c_{text}| \times |z|$ 的硬单调注意矩阵，表示每个输入音素扩展多长时间以与目标语音时间对齐。因为对齐没有真实标签，必须在每次训练迭代时估计对齐。在问题设置中，目标是为后编码器提供更高分辨率的信息。因此，使用目标语音 x_{lin} 的线性尺度频谱图而不是 mel 频谱图作为输入。KL 散度是：

$$L_{KL} = \log q_{\phi}(z|x_{lin}) - \log p_{\theta}(z|c_{text}, A), z \sim q_{\phi}(z|x_{lin}) = N(z; \mu_{\phi}(x_{lin}), \sigma_{\phi}(x_{lin}))$$

分解的正态分布用于参数化先验和后验编码器。发现增加先验分布的表现力对于生成真实样本很重要。因此，VITS 中使用了标准化流 (Normalizing Flow) f_{θ} ，它允许在分解的正态先验分布之上，按照变量变化的规则将简单分布可逆地转换为更复杂的分布：

$$p_{\theta}(z|c) = N(f_{\theta}(z); \mu_{\theta}(c), \sigma_{\theta}(c)) \left| \det \frac{\partial f_{\theta}(z)}{\partial z} \right|, \\ c = [c_{text}, A]$$

11.3.2 对齐估计

为了估计输入文本和目标语音之间的对齐 A ，VITS 论文参考了 GLOW-TTS 中提出的单调对齐搜索 (Monotonic Alignment Search, MAS) 方法，该方法基于搜索对齐的思想，可以

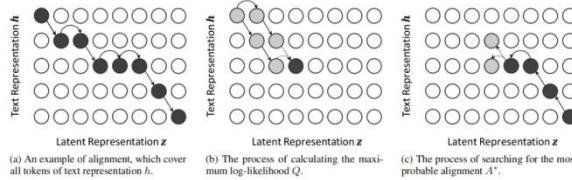
最大化由标准化流 f 参数化数据的可能性。

$$\mathbf{A} = \arg \max_{\hat{\mathbf{A}}} \log p(\mathbf{x} | \mathbf{c}_{text}, \hat{\mathbf{A}}) = \arg \max_{\hat{\mathbf{A}}} \log N(f(\mathbf{x}); \mu(\mathbf{c}_{text}, \hat{\mathbf{A}}), \sigma(\mathbf{c}_{text}, \hat{\mathbf{A}}))$$

其中由于人类按顺序阅读文本而不跳过任何单词，因此候选对齐被限制为单调且不可跳过。为了找到最佳对齐方式，Glow-TTS 中使用动态规划。在本方法的设置中直接应用 MAS 很困难，因为目标是 ELBO，而不是精确的对数似然。因此，本方法重新定义 MAS 以找到最大化 ELBO 的对齐方式，这简化为找到最大化潜在变量 z 的对数似然的对齐方式：

$$\begin{aligned} & \arg \max_{\hat{\mathbf{A}}} \log p_{\theta}(x_{mel} | z) - \log \frac{q_{\phi}(z | x_{lin})}{p_{\theta}(z | c_{text}, \hat{\mathbf{A}})} \\ &= \arg \max_{\hat{\mathbf{A}}} \log p_{\theta}(z | c_{text}, \hat{\mathbf{A}}) \\ &= \log N(f_{\theta}(z); \mu_{\theta}(c_{text}, \hat{\mathbf{A}}), \sigma_{\theta}(c_{text}, \hat{\mathbf{A}})) \end{aligned}$$

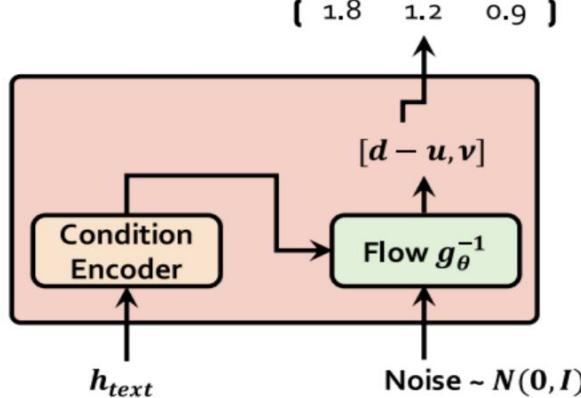
MAS 的算法流程如下图所示：



此外，Glow-TTS 中也使用了时长预测器（duration predictor），但它是静态的。为了模拟人说同样一句话会有不同的语速，VITS 中引入了随机时长预测器（stochastic duration predictor），它是一个基于流的生成模型，本可以直接优化似然函数。但实现起来很困难，这是因为：

1. 时长（Duration）是离散的整数，而 FLOW 要求输入是连续的，因此需要做反量化（dequantization）。
2. 时长是标量，不能在 FLOW 中像高维向量那样实现可逆变换。

因此，VITS 中使用了变分反量化（variational dequantization）和变分数据增广（variational data augmentation）技术来解决这两个问题，此外在训练过程中引入 stop gradient 算子，防止 duration predictor 的误差信息传播到其他模块，即让它单独训练，如下图所示：



在推理时，我们只给 duration predictor 文本的信息来生成浮点型的时长，然后向正无穷取整后得到 d 。

11.3.3 对抗训练

为了在我们的学习系统中采用对抗性训练（Adversarial Training），我们添加了一个判别器 D 用于区分解码器 G 生成的输出与真值波形 y。在这项工作中，VITS 使用了两种成功应用于语音合成的损失：最小二乘损失函数（least-squares loss function），用于对抗训练，额外的特征匹配损失（feature-matching loss），用于训练生成器（同 Hifi-GAN），公式如下：

$$L_{adv}(\mathbf{D}) = E_{(y, z)}[(\mathbf{D}(y) - \mathbf{1})^2 + (\mathbf{D}(G(z)))^2]$$

$$L_{adv}(G) = E_z[(D(G(z)) - \mathbf{1})^2]$$

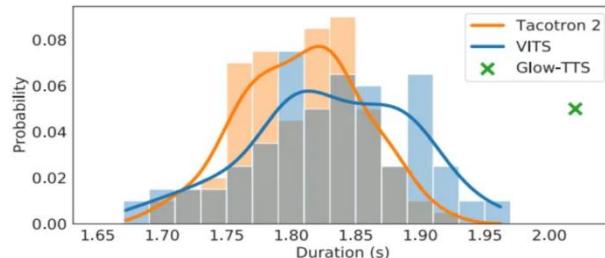
$$L_{fm}(G) = E_{(y, z)}[\sum_{l=1}^T \frac{1}{N_l} ||\mathbf{D}^l(y) - \mathbf{D}^l(G(z))||_1]$$

综上所述，整个 VITS 模型的 VAE 损失函数是由我们之前提到的五个损失函数相加组成，即：

$$L_{VAE} = L_{recon} + L_{KL} + L_{dur} + L_{adv}(G) + L_{fm}(G)$$

11.3.4 VITS 的应用

VITS 这个模型是非常强大的，它首先表现在语音合成的任务中，相比于之前的 GLOW-TTS，Hifi-GAN，Tacotron2 等模型，VITS 可以在不降低合成质量的基础之上，生成具有不同音高和节奏的语音，如下图所示：



此外，VITS 模型可以获得更快的速度：

Model	Speed (kHz)	Real-time
Glow-TTS + HiFi-GAN	606.05	$\times 27.48$
VITS	1480.15	$\times 67.12$
VITS (DDP)	2005.03	$\times 90.93$

当然对于这个模型，还存在着一些不同的变种，比如它可以和 BERT 相结合，它也可以用于 AI 音色转换的领域中，对于这些领域，感兴趣的读者可以自行探索，我们在本章中只是用 VITS 作为生成式模型的收尾。此外，李宏毅老师在今年新开了一门名叫“Generative AI”的课程，相信在这门课中会对生成式模型进行更多的介绍，敬请期待！



12 机器学习的可解释性

12.1 可解释性的基本概念

12.1.1 什么是可解释性

我们首先了解一下什么是可解释的人工智能（机器学习）。我们在前 11 章对很多模型进行了初步的认识，但我们有时候并不知道为什么需要用这些模型。因此人们开始希望机器能够既给出模型本身的用法，又能解释为什么需要用这个模型，这就是机器学习的可解释性。



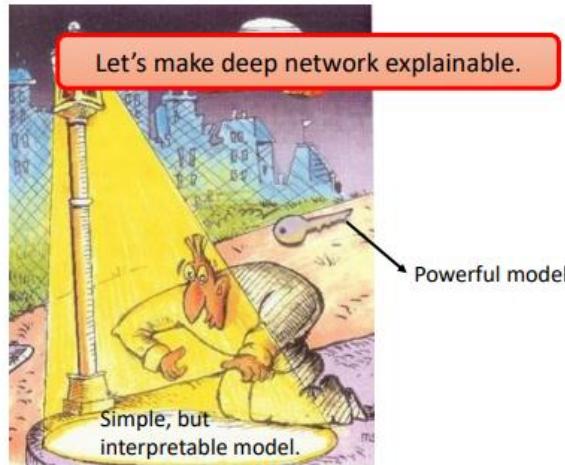
现阶段，可解释性可以大致分为两类：第一类是局部可解释性（local explanation），第二类是全局可解释性（global explanation）。以上图为例，前者主要需要做的是解释“计算机为什么觉得这张图片是一只猫”的问题，而后者需要解释“在计算机眼中猫是什么样”的问题。我们将在后续小节中对这两大类问题分别进行介绍。

12.1.2 为什么需要可解释性

事实上，今天在很多真实的应用中，可解释性的机器学习，或者说可解释性的模型往往是必须的。举例来说，银行今天可能会用机器学习的模型来判断要不要贷款给某一个客户，但是根据法律的规定银行作用机器学习模型来做自动的判断它必须要给出一个理由。所以这个时候，我们不是只训练机器学习模型就好，我们还需要机器学习的模型是具有解释力的。或者是说机器学习未来也会被用在医疗诊断上，但医疗诊断是人命关天的事情，如果机器学习的模型只是一个黑箱，不会给出诊断的理由的话，那我们又要怎么相信它做出的是正确的判断呢。现在也有人想把机器学习的模型用在法律上，比如说帮助法官判案，比如一个犯人能不能够被假释，但是我们怎么知道机器学习的模型它是公正的呢。我们又怎么知道它在做判断的时候没有种族歧视等其他的问题呢。所以我们希望机器学习的模型不只得到答案它还要给我们得到答案的理由。再更进一步，自动驾驶汽车未来可能会满街跑，但是当自动驾驶汽车突然急刹的时候导致车上的乘客受伤，那这个自动驾驶系统有没有问题呢？这也许取决于它急刹的理由，如果它是看到有一个老太太在过马路所以急刹，那也许它是对的，但是假设它只是无缘无故就突然发狂要急刹，那这个模型就有问题了。所以对它的种种的行为，种种的决策，我们希望知道决策背后的理由。更进一步，也许机器学习的模型如果具有解释力的话，那未来我们可以凭借着解释的结果再去修正我们的模型。

有人可能会想说我们今天之所以这么关注可解释性的机器学习的议题，也许是因为深度的网络本身就是一个黑箱。那我们能不能够用其它的机器学习的模型呢？如果不要用深度学习的模型，而改采用其他比较容易解释的模型会不会就不需要研究可解释性机器学习了呢。举例来说，假设我们都采用线性模型，它的解释的能力是比较强的，我们可以轻易地知道根据一个线性模型里面的每一个特征的权重，知道线性的模型在做什么事。所以训练完一个线

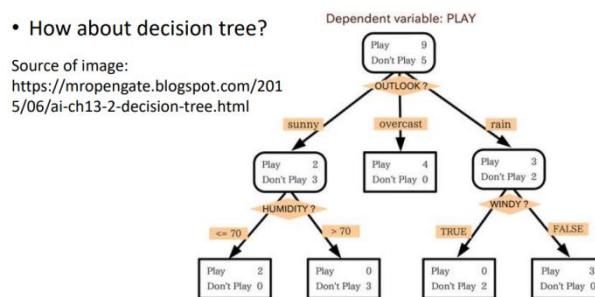
性模型后，我们可以轻易地知道它是怎么得到它的结果的。但是线性模型的问题是它没有非常地强大，它的表达能力是比较弱的。线性模型有很巨大地限制，所以我们才很快地进入了深度的模型。但是深度的模型它的坏处就是它不容易被解释，深度的网络它就是一个黑盒子，黑盒子里面发生了什么事情，我们很难知道。虽然它比线性的模型更好，但是它的解释的能力是远比线性的模型要差的。所以讲到这里，很多人就会得到一个结论，我们就不应该用这种深度的模型，我们不该用这些比较强大的模型，因为它们是黑盒子。但是这样的想法其实都是削足适履，我们因为一个模型它非常地强大，但是不容易被解释就放弃它吗？我们不是应该是想办法让它具有可以解释的能力吗？



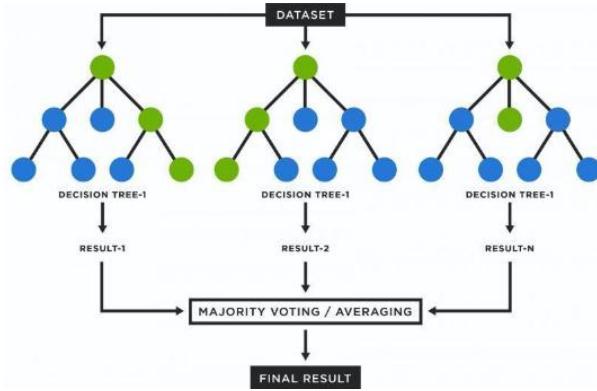
之前 Yann LeCun 讲了一个故事，有一个醉汉他在路灯下面找钥匙，大家问他说，你的钥匙掉在路灯下吗。他说不是，因为这边有光。所以我们坚持一定要用简单但是比较容易被解释的模型其实就好像是坚持一定要在路灯下面找钥匙一样。因为一个模型是有比较好可解释性的，虽然它比较不好但我们还是坚持要使用它，就好像一定要在路灯下面找钥匙一样。不知道说真实强大的模型，也许根本在路灯的范围之外。而我们现在要做的事情，就是改变路灯的范围，改变照明的方向，看能不能够让这些比较强大的模型可以被置于路灯之下，比较有可解释性。

12.1.3 可解释性 v.s 能力

那么一个模型是具有可解释性的，能否和这个模型的能力这一命题“画上等号”呢？答案是显然不能！这一答案已经可以从上一小节中 Yan LeCun 所讲的故事得出。但有些模型是可以兼顾可解释性和能力的，比如决策树（Decision Tree）。



当然实际上，在一般情况下决策树也不是很好解释，而且随着决策树结构变得越来越复杂，虽然能力会变得很强（在 Kaggle 等比赛中经常可以做到“大杀四方”），但是这时候我们用的技术已经不是决策树，而是随机森林（Random Forest），它其实是多棵决策树共同决定的结果。一棵决策树可以凭借着每一个节点的问题和答案知道它是怎么做出最终的判断的，但当你有一片森林的时候，你就很难知道说这一片森林是怎么做出最终的判断的。因此，这也会使可解释性的能力被削弱。



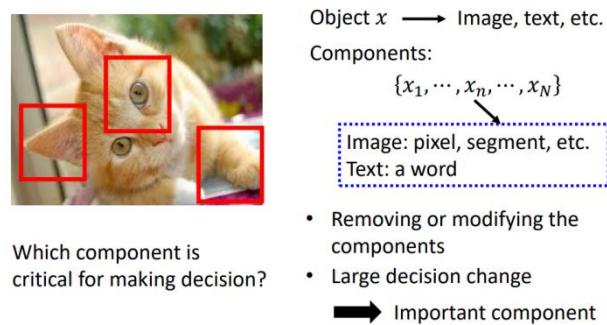
12.2 局部可解释性 (Local Explanation)

12.2.1 基本思想

我们在本章开头讲过，局部可解释性所需要解释的问题是“为什么计算机觉得这张图片是某种事物”类似的问题。我们知道，一张图片可以被分成很多部分，因此，局部可解释性的基本思想是：探索图片（或其它类型的数据）中的哪些组件对于识别数据起作用。

我们想要知道机器如何判断，这个问题其实就等价于机器把哪些特征看得很重要，这些重要的特征的出现极大程度上地影响了机器的判断。

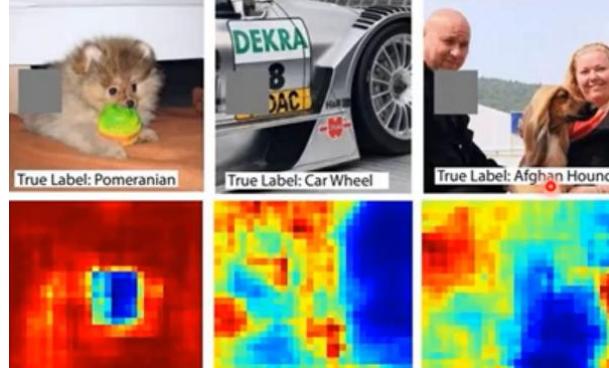
那怎么知道一个部分的重要性呢？基本的原则是我们把所有的部分都拿出来，然后把每一个部分做改造或者是删除。如果我们改造或删除某一个部分以后，网络的输出有了巨大的变化，那我们就知道这个部分是很重要的，基本思想如下图所示：



12.2.2 移除组成要素（遮挡法）

第一种方法是移除组成要素（遮挡法），我们还是以图像数据为例，在图像数据中的组

成要素是一块一块的区域（这一点我们在讲 CNN 时有所提及），那我们就可以移除一部分，然后来记录移除这个部分的影响。如下图，用灰色方块对图片中的每个部分依次进行遮挡，然后再进行预测，预测的效果在图的下方，与上图涂灰区域的中心对应，观察对识别结果的影响。



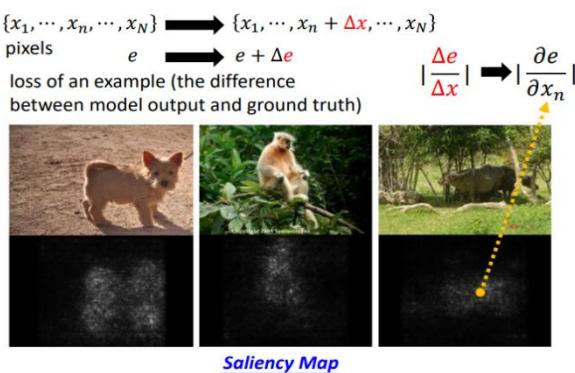
上图中蓝色区域说明此处的图像对图片识别起到重要作用，例如上面第三张图中，机器识别出来了狗。如果我们不采用这种方法，我们完全可以不信机器真的是看到了狗才识别出来的，而是把左边的人看成了狗；而采用这种方法，我们看出如果图片中抹去了狗的部分，那机器完全不会在这张图中找到狗，那就说明了机器确实是学到了狗的相关特征。

12.2.3 改变梯度

对于这个问题，还有一个更进阶的方法，即计算梯度。如下图所示，假设 wo 你有一张图片，将其分为 N 块区域，记为： x_1, \dots, x_N ，其中 x_i 代表一个像素（pixel）。接下来，我们去计算这张图片的损失函数 e ，即模型输出的结果与正确答案的差距（用交叉熵表示）。其数值越大，就代表在辨识的结果越差。

那我们如何知道每一个像素的重要性呢？我们可以将每一个像素做一个小小的变化——加上一个微小量 Δx ，然后再丢到模型里面看一下损失会有什么样的变化，损失的变化用 Δe 来表示。那么假如 Δe 趋近于 0，就代表这个位置，这个像素对于图像识别而言可能是不重要的；反之可能是重要的。

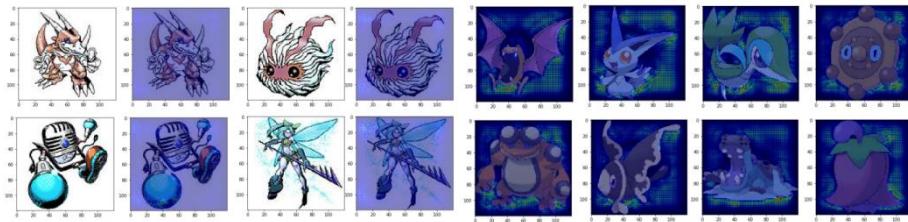
那么实际上，从数学的角度来看待这个问题，我们可以将 Δe 和 Δx 进行相除，其结果可以看作损失函数 e 在图片 x 的某一个维度的偏微分。当我们把每一个图片里面每一个像素它的这个比值都算出来后，我们就得到一个图，这个图就叫做显著图（Saliency Map）。



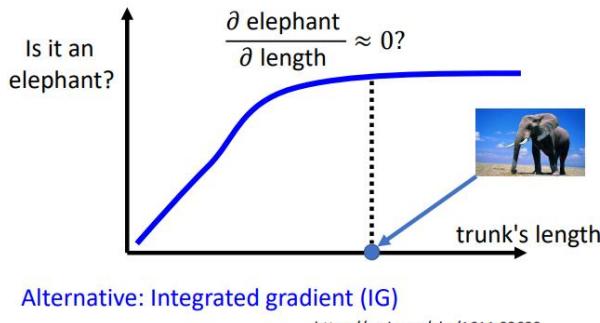
Karen Simonyan, Andrea Vedaldi, Andrew Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR, 2014

如上图所示，图片的上方是原始图片，下方就是显著图，越亮的点，就代表这个像素越

重要。



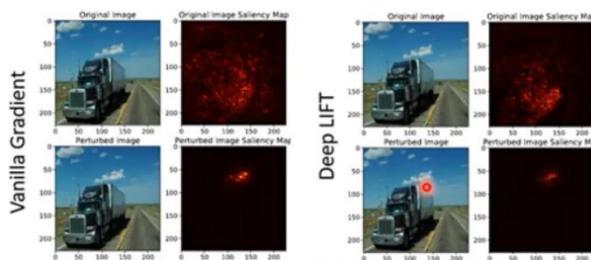
当然这种用梯度的方式去判断重要因素的方法有很多种，但同时它也有一些限制。首先会出现的问题叫做信息冗余现象，如下图所示，横轴是鼻子长度，而纵轴是判断为大象的概率。显然，鼻子长是大象的一个极其显著的特点，这里的曲线我们也可以确定机器是关注到这一点的了；但是鼻子长到足够明显了的时候，再长貌似就没有必要了，即信息冗余。如果我们在这个时候看梯度，那竟然几乎是 0，也就是梯度饱和，从而和我们的想法相矛盾，显然这是不合理的。那针对这一问题，下图右侧也给出了解决方案，感兴趣的读者可以阅读链接中的文章去探索。



Alternative: Integrated gradient (IG)

<https://arxiv.org/abs/1611.02639>

此外，使用显著图的方法解释 AI 很有可能会受到被攻击的风险（关于 AI 中的攻击与防御，我们将在下一节进行介绍）。如下图所示，我们就加入了一些肉眼看不到的噪声，也没有改变机器分类的结果，但机器所关注到的部分改变了很多，例如图中遭受攻击后，机器其实是通过看云来识别出了卡车。虽然我们无法解释这是为什么，但它确实发生了，这就说明了这种方法并不是完全可靠的，而且机器学到的内容可能还是很杂，不是我们想要的效果。



12.3 全局可解释性 (Global Explanation)

介绍完局部解释，我们接下来要讲全局的解释。我们前一小节讲的是局部的解释，也就是给机器一张照片，让它告诉我们说看到这张图片，它为什么觉得里面有一只猫。与此不同的是，全局解释并不是针对特定某一张照片来进行分析，而是把我们训练好的模型拿出来，

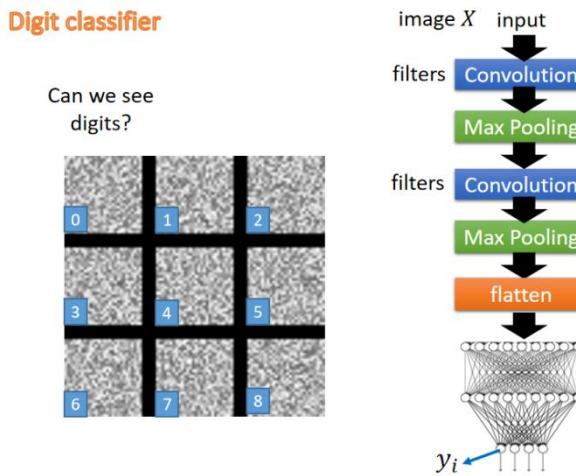
根据这个模型里面的参数去检查它的特性。也就是对这个网络而言，到底它所认为的猫长什么样子。

GLOBAL EXPLANATION: EXPLAIN THE WHOLE MODEL

Question: What does a "cat" look like?

12.3.1 反向寻找理想输入

在 CNN 中，其实就存在着全局可解释性的问题。假设我们是在识别 MNIST 数据集的手写数字，我们既然是想知道在机器眼中每个数字的样子，那我们就完全可以把结果固定，然后对输入通过梯度下降来得出，到底哪样的输入会使得结果最显著。



在卷积神经网络中，假设我们已经训练好了网络，并且 y_i 已经给定，那么全局解释所需要做的事情是找到一个最优解 X^* ，它满足让 y_i 越大越好，公式如下：

$$X^* = \arg \max_X y_i$$

12.3.2 添加正则化项

那如果我们希望看到的是比较像是人想像的数字应该要怎么办呢？方法是在解优化问题的时候，加上更多的限制。我们已经知道数字可能是长什么样子，所以要把这个限制加到这个优化的过程里。因此，我们现在所需要找到的最优解既需要满足让 y_i 越大越好，又需要让正则项分数 $R(X)$ 越大越好。

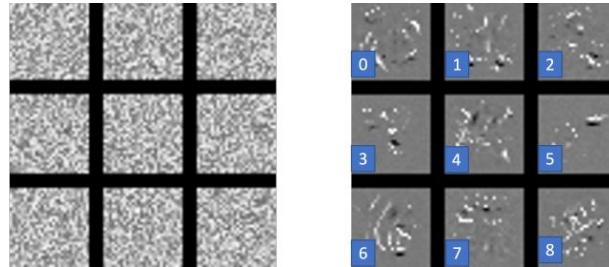
$$R(X) = - \sum_{i,j} |X_{ij}|$$

How likely
 X is a digit

在这里，正则化项 $R(x)$ 的作用是让 X 更像是一个数字，它的表达形式可以是多样的，上图只是其中一种表示形式，此时，我们的优化问题转化为了下面的形式：

$$X^* = \arg \max_X y_i + R(X)$$

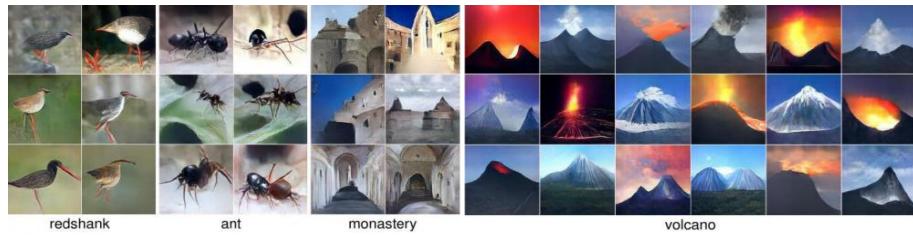
如下图所示，当我们将 $R(x)$ 设置成 X 的像素值的平方和时，我们看到的结果如下图所示，相比于左侧的最终图像分类器的输出，右侧的更加规则一些，但看起来还是不太像数字。我们还是需要增加更多的限制，当然这都是可以根据我们对于图像的了解去自己决定的。



当然这边还有另外一种正则化的方法——利用图像生成技术进行正则化。我们可以通过 GAN 或者 VAE 的方法训练一个图像生成器，对于这个图像生成器，它的输入是一个从高斯分布中采样出来的向量 z ，输出是图片 X 。假设我们的生成器用 G 来表示，那么这个生成器就相当于一个函数： $X = G(z)$ 。那么现在的优化问题同样可以转化为：

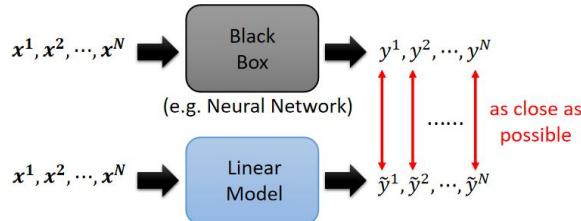
$$z^* = \arg \max_z y_i$$

我们将找到的最优值 z^* 丢到生成器 G 中，事实上，这样的效果是非常好的，因为它可以代表机器“脑子”中想象的内容了。



12.4 扩展与小结

那其实可解释性机器学习还有很多的技术，比如说我们可以用一些可解释性的模型来替代黑盒子模型。

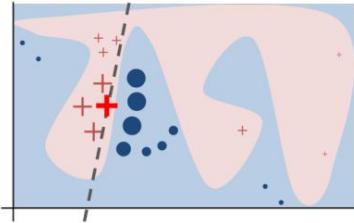


如上图所示，我们可以用一个简单的模型（线性模型）想办法去模仿复杂模型的行为，从而起到代替复杂模型的作用。

线性模型是最容易解释的模型，不论参数有多少个，每个参数的意义都是能很鲜明的说出这个对应项的情况。那现在我们的目标是使用相同的输入，使线性模型和神经网络有相近的输出。

然而实际上并不能使用线性模型来模拟整个神经网络，因为它过于简单，很多时候根本无法胜任这个任务，但可以用来模拟其中一个局部区域，因此就衍生出了一种名叫 LIME 的技术。

假设我们的输入是 x , 输出是 y , 神经网络可以看作是一个黑盒, 这个黑盒代表着 y 和 x 之间的某种关系, 由于我们不能用线性模型来模拟整个神经网络, 因此我们采用它来模拟其中一个局部的区域, LIME 就是这样的思想, 其步骤如下图所示:



Algorithm 1 Sparse Linear Explanations using LIME

```
Require: Classifier  $f$ , Number of samples  $N$ 
Require: Instance  $x$ , and its interpretable version  $x'$ 
Require: Similarity kernel  $\pi_x$ , Length of explanation  $K$ 
 $\mathcal{Z} \leftarrow \{\}$ 
for  $i \in \{1, 2, 3, \dots, N\}$  do
     $z'_i \leftarrow sample\_around(x')$ 
     $\mathcal{Z} \leftarrow \mathcal{Z} \cup \langle z'_i, f(z_i), \pi_x(z_i) \rangle$ 
end for
 $w \leftarrow K\text{-Lasso}(\mathcal{Z}, K)$   $\triangleright$  with  $z'_i$  as features,  $f(z_i)$  as target
return  $w$ 
```

以上就是关可解释性机器学习的介绍。这一章主要和大家介绍了可解释性机器学习的两个主流的技术, 局部的解释和全局的解释。局部的解释主要是通过对于一个特定的样本, 去找到一个和这个样本最相关的一些特征, 然后把这些特征拿出来, 去解释这个样本的分类结果。全局的解释主要是通过对于一个模型, 去找到一些和这个模型最相关的一些特征, 然后把这些特征拿出来, 去解释这个模型的行为。

13 机器学习中的攻击与防御

本章中，我们将对机器学习的攻击与防御进行介绍。我们之前已经训练了非常多各式各样的神经网络，这些神经网络都有非常高的正确率，我们期待可以把这些技术用在真正的应用上。但是把这些网络用在真正的应用上，仅仅提高它们的正确率是不够的。它们需要能够应付来自人类的恶意，也就是说，它们需要能够应付对抗攻击。有时候神经网络的工作是为了要检测一些有恶意的行为，它要检测的对象会去想办法骗过网络，所以我们不仅要在一般的情况下得到高的正确率，还需要它在有人试图想要欺骗它的情况下也得到高的正确率。举例来说，我们会使用神经网络来做电子邮件的过滤，检测它是不是垃圾邮件。而这个垃圾邮件的发信者也会想尽办法避免他的邮件被分类为垃圾邮件，这就是机器学习中的攻防。

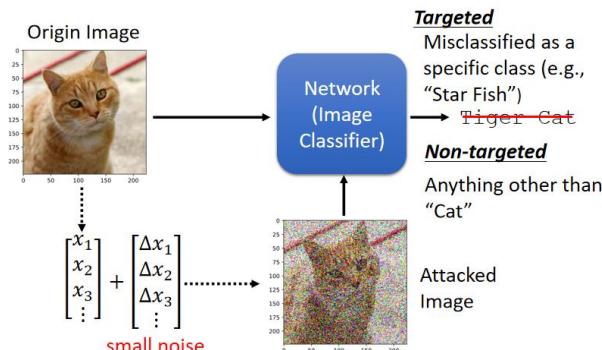


13.1 模型攻击

神经网络不仅是用在研究上，而且实际上是要用在各种有意义的应用中。因此模型如果仅仅是对噪声和大部分时间起作用是不够的，还要去对抗恶意攻击(不暴露出弱点)。所以对模型可抗性的研究有重要意义。

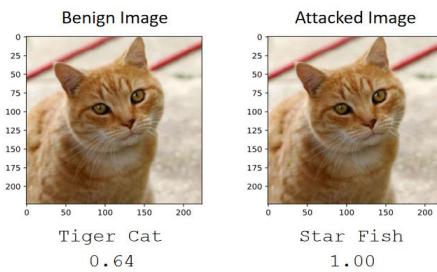
13.1.1 模型攻击简介

模型攻击就是向模型输入中加入一些人为处理过的噪声，以期待模型输出错误的结果。以图像模型为例，如下图所示，假如有一个做动物识别的神经网络，我们输入一张原始图片 x ，神经网络应该预测出这张图片到底是“老虎”还是“猫”。但是在我们给原始图片加上一个微小的噪声 Δx 后，通过网络预测出来的结果就可能会是其它结果了。



攻击大致上可以分成两种类型，一种是没有目标的攻击，也就是说我们只要受攻击图像的输出不是猫就算成功了。还有另外一种更困难的攻击，即有目标的攻击，我们希望受攻击图像的输出是狮子等等，也就是说我们希望网络不止它输出不能是猫，还要输出别的具体的东西，这样才算是成功攻击。

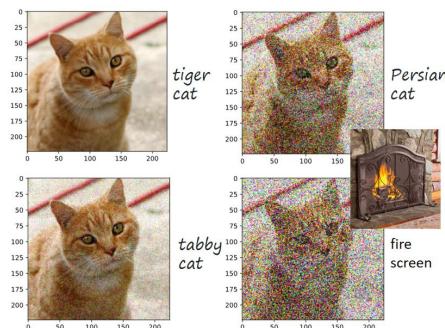
那么以上的攻击真的有办法做到吗？实际上是可以的，如下图所示，当我们将一张原始的动物图片丢入一个 ResNet-50 的图片分类器中时，它会输出它所认为的图片属于的具体类别（老虎猫，当然这张图片实际上未必是一只老虎猫），同时还会输出一个置信分数，这个置信分数是图像分类做完 softmax 之后的结果，也就是说图像分类器认为其所属于老虎猫的概率。



接下来我们将原始图像加入一些噪声，希望成功攻击的目标是把老虎猫变成海星。我们将加入噪声以后的图片(上图右)丢到 ResNet 中，得到的输出是海星，而且置信得分是 1.00，也就是说经过攻击后，机器十分确定这张图片是一只海星。因此，在加入噪声之后，我们的网络会非常肯定地将人类看不出答案的图片识别成一个错得非常离谱的结果。需要注意的是，这个案例并非特例，而且 ResNet-50 的模型也已经足够强大，我们也没有理由怀疑该模型的分类能力，或许你仍然会有一些疑惑，因为上图中左右两张图片看起来是一样的，但实际上它们还是有一些不一样的，当我们将其像素值进行相减，并乘上 50 倍，会得到一个如下图所示的噪声差距：

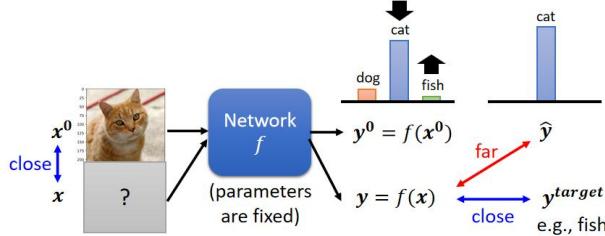


当然，上面的噪声是肉眼识别不出来的，如果我们加入的是人类显而易见的噪声，网络并不一定会犯错，如下图所示，随着噪声的不断添加，图像分类器虽然会犯错，比如识别出的猫的品种不同，以及将橘黄色的猫识别成火焰，但是这些错误都是情有可原的。



13.1.2 如何进行网络攻击

在讲为什么噪声会影响识别结果之前，我们来看看攻击究竟是如何做到的。如下图所示，假设我们有一个神经网络 f ，其输入是一张图片 x^0 ，输出一个分布，代表着这张图片所属每个类别的概率。在网络攻击的问题中，我们不讨论网络的参数部分。因此，我们不妨假设网络的参数是固定的。现在我们的目标是：找到一张新的图片 x ，使得 $f(x)$ 与正确答案 \hat{y} 的差距越大越好。



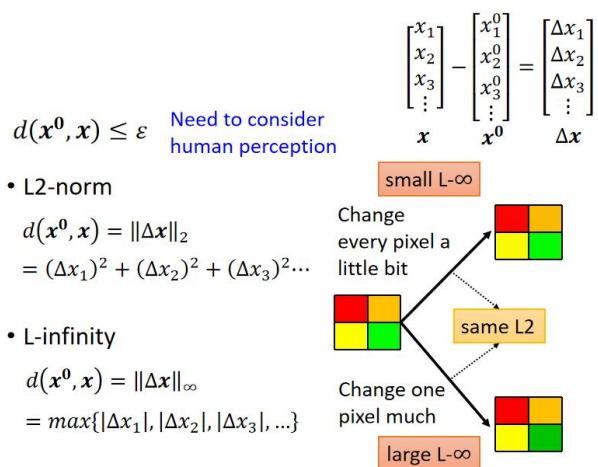
而对于上述问题，上一小节所介绍的有目标攻击和无目标攻击的损失函数是不同的：对于无目标攻击，我们只需要最大化输出值 y 和实际值 \hat{y} 的差距即可，具体做法是将损失函数定义为一个负的交叉熵的形式，即无目标攻击的损失函数如下：

$$L(x) = -e(y, \hat{y})$$

而对于有目标的攻击，我们会先设定好我们的目标，而我们除了希望输出值和实际值的差距越大越好之外，还希望我们的输出值和预先设置的目标值越相近越好，因此该问题的损失函数如下：

$$L(x) = -e(y, \hat{y}) + e(y, y_{target})$$

另外需要注意的是，我们希望找一个 x ，在最小化损失的同时还需要保证加入的噪声越小越好，也就是我们新找到的图片可以欺骗过网络的图片，并且尽量跟原来的图片要越相似越好。所以我们在解这个优化问题的时候，还会多加入一个限制，即 x 和 x^0 的差距要小于某个阈值 ϵ ，这个阈值是根据人类的感知能力来决定的，如果 x 和 x^0 的差距超过了该阈值，人类就会发现这是一张带有噪声的图片，所以我们要保证这个差距不要太。而阈值的定义方式可以用 L2 范数定义。

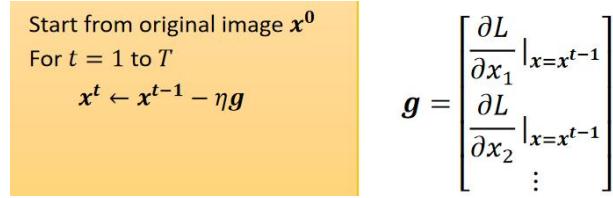


当然，我们还可以用无穷范数来定义这个距离，在上面图片识别例子中，我们更适合用无穷范数的定义方式。因为 L2 范式不能很好的描述人类感知的差别，它无法区分上图（右

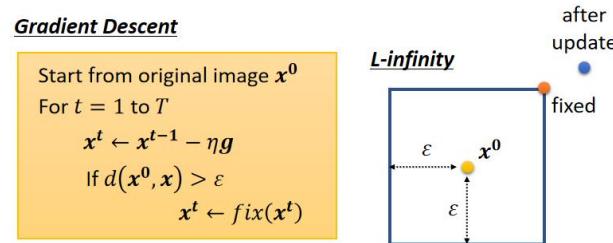
上与右下) 两张图片。如果有几个像素有明显的变化, 我们人眼还是能轻易捕捉到的。但是当图片不是很简单的时候, 整张图片每个像素都有很细微的变化, 我们肉眼几乎是完全检查不出来的。事实上不管距离函数选择的是 L2 范数还是无穷范数, 我们的优化问题均可写成如下形式:

$$\mathbf{x}^* = \arg \min_{d(\mathbf{x}^0, \mathbf{x}) \leq \varepsilon} L(\mathbf{x})$$

那么这个问题该如何求解呢? 假设我们的优化问题并没有距离的限制, 只需要用我们之前学过的梯度下降算法求解即可, 如下图所示:



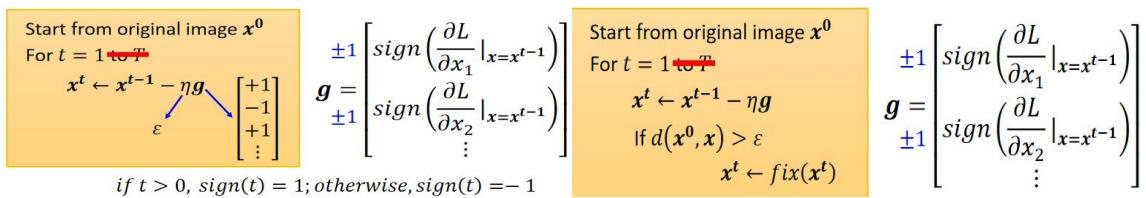
那这个距离限制怎么加进优化问题中呢? 其实就是在梯度更新以后再加一个限制即可。举例来说, 假设我们采用无穷范数来定义距离, 如下图所示。根据无穷范数的定义可知, 我们的 \mathbf{x} 只需要保证在下图中的方框中即可, 因此只要在梯度下降过程中, 如果更新梯度后 \mathbf{x} 落在了方框外面, 我们将其想办法拉回来即可, 具体操作为: 在方框里面找一个跟蓝色的点最近的位置, 然后把蓝色的点“拉进来”就结束了。



13.1.3 最简单的攻击方式

所谓的攻击还有很多不同的变形, 不过大同小异, 它们通常要么是限制不一样, 要么是优化的方法不一样。接下来再介绍一个最简单的攻击的方法——FGSM (Fast Gradient Sign Method)。

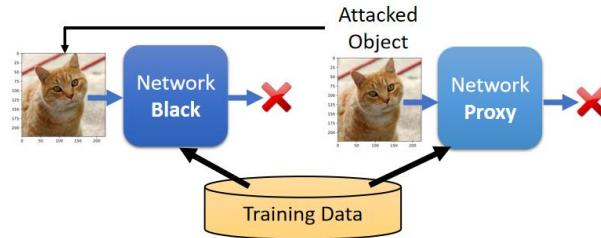
FGSM 与一般的梯度下降方法不同的是, 本来一般我们在做梯度下降的时候, 需要迭代更新参数很多次; 而在 FGSM 中, 我们只需要更新一次参数。具体来说, FGSM 将原始的梯度做了一个特殊的设计 (详见原始论文: <https://arxiv.org/abs/1412.6572>), 即对计算出来的梯度取一个符号函数, 对于学习率 η , 直接设置为我们之前定义的阈值 ε 。这样能保证更新后的 \mathbf{x} 一定落在方框的四个角落, 当我们将这个方法多迭代几次, 得到的结果会更好。



13.2 其它类型的攻击

13.2.1 黑盒攻击

事实上，在我们之前介绍的攻击方法中，模型的网络结构，输入输出，损失函数等都是事先就知道的，我们将这种类型的攻击称为白盒攻击。与之对应的攻击称为黑盒攻击，即我们事先不知道模型的参数，但是我们可以通过一些方法来反推出来。



在黑盒攻击中，如何将模型反推出来呢？假如我们知道该网络是用什么样的训练资料训练出来的，就可以训练一个代理网络（Proxy Network）来模仿我们要攻击的对象如果代理网络与要被攻击的网络有一定程度的相似的话，那我们只要对代理网络进行攻击，也许原始的网络也会被攻击成功，整个过程如上图所示。

那当我们拿不到训练资料的时候，还能继续做黑盒攻击吗？其实也是可行的。具体做法是接将一堆现有的数据（图片）丢进需要被攻击的模型观察其输出，然后再把输入输出的成对资料拿去训练一个模型的话，我们就有可能可以训练出一个类似的代理网络了，我们再对代理网络进行攻击即可。

黑盒攻击实际上还是很容易成功的，如下图所示，有 5 个不同的网络，ResNet-152、ResNet-101、ResNet-50、VGG-16 和 GoogLeNet。每一列代表要被攻击的网络，每一行代表代理网络。对角线的地方代表是代理网络和要被攻击的网络是一模一样的，因此对角线的地方其实是白盒攻击，而非对角线的地方均为黑盒攻击。作为神经网络的攻击方，我们希望表中的正确率越低越好。我们发现，黑盒攻击模型的正确率是比白盒攻击还要高的，但是其实这些正确率也都非常低（都低于 50%），所以显然黑盒攻击也有一定的成功的可能性。此外，我们很自然地能想到，黑盒攻击在无目标攻击上比较容易成功，而在有目标攻击上就没有那么容易成功了。

		Be Attacked				
		ResNet-152	ResNet-101	ResNet-50	VGG-16	GoogLeNet
Proxy	ResNet-152	0%	13%	18%	19%	11%
	ResNet-101	19%	0%	21%	21%	12%
	ResNet-50	23%	20%	0%	21%	18%
	VGG-16	22%	17%	17%	0%	5%
	GoogLeNet	39%	38%	34%	19%	0%

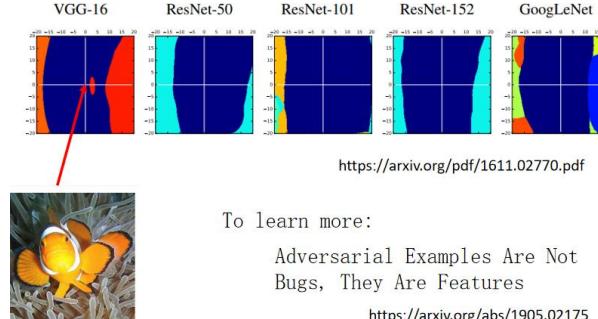
(lower accuracy → more successful attack)

此外，如果要增加黑盒攻击的成功率，我们可以使用集成攻击（Ensemble Attack）的方法，具体做法是：对于每个被攻击的网络，我们都可以集成其它 4 个网络进行黑盒攻击，得到的结果如下：

Ensemble Attack					
	ResNet-152	ResNet-101	ResNet-50	VGG-16	GoogLeNet
-ResNet-152	0%	0%	0%	0%	0%
-ResNet-101	0%	1%	0%	0%	0%
-ResNet-50	0%	0%	2%	0%	0%
-VGG-16	0%	0%	0%	6%	0%
-GoogLeNet	0%	0%	0%	0%	5%

13.2.2 单像素攻击

在介绍单像素攻击之前，我们先来讨论一下为什么黑盒攻击非常容易成功。实际上这个问题暂时没有一个标准答案（是一个很有潜力的研究方向），但目前有一个信服度比较高的结论，它基于下图的实验。



在上图中的论文中有一个实验，图中的原点代表一张小丑鱼的图片，横轴和纵轴分别是把这张图片往两个不同的方向移动——横轴代表着在 VGG-16 上面可以攻击成功的方向，纵轴表示一个随机的方向。我们发现，虽然横轴是让 VGG-16 这个模型可以攻击成功，但在其它几张图（模型）中，中间的深蓝色区域都是很相近的，而这个深蓝色的区域表示会被辨识成小丑鱼的图片的范围。也就是说如果把这个小丑鱼的图片加上一个噪声这个矩阵在高维的空间中横向移动，基本上网络还是会觉得它是小丑鱼的图片。但是如果是往可以攻击成功 VGG-16 的方向来移动的话，那基本上其他网络也是有很高的机率可以攻击成功的。对于小丑鱼这一个类别，它在攻击的方向上，可移动的范围特别窄，只要把这张图片稍微移动一下它就掉出会被辨识成小丑鱼的区域范围之外了。对每一个网络来说，攻击的方向对不同的网络影响都是很类似的。因此，有些人就认为对抗攻击会成功的原因，是来自于数据，当我们有足够的数据的时候，也许就有机会避免对抗攻击。当然这个只是一部分人的想法，即数据是造成攻击会成功的元凶。

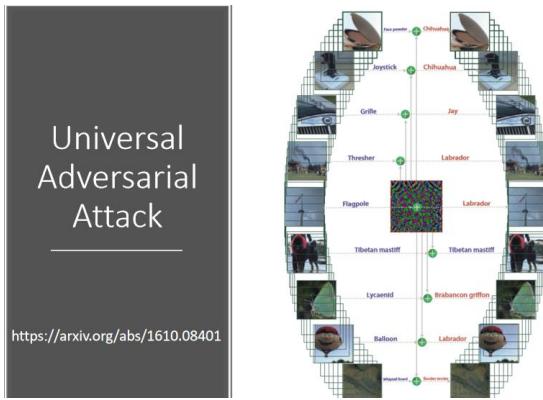
因此，我们希望攻击的信号越小越好，那具体可以小到什么程度呢？其实已经有人成功地做出了单像素攻击（One pixel Attack），也就是只能改变图片里面的一个像素而已。如下图所示，我们将所有的四张子图都对一个像素进行改变，并用红色圆圈圈起来。我们希望在改变这一个像素之后，图像识别系统的判断就必须要有错误，每一个图片下方黑色的部分代表的是攻击前的结果，蓝色代表是攻击后的识别结果。



13.2.3 通用对抗攻击

当然单像素攻击还是有一些局限，它的攻击并没有非常成功。比如上图左下角的茶壶，当我们改变一个像素进行攻击时，机器会把茶壶识别成摇杆。所以我们还是希望能够找到更好的攻击方式。

比单像素更好的攻击方式是通用对抗攻击，也就是说我们只要找到一个攻击信号，这个攻击信号可以攻击所有的图片。在之前对于多张图片，我们会分别找出不同的攻击信号来攻击不同的图片。但实际上我们不可能对于每一张图片都去找一个攻击信号，这个运算量可能会非常地大。因此，通用对抗攻击的目标就是找到一个通用的噪声进行攻击，从而让所有图片的识别结果出错。



13.2.4 现实世界中的攻击

我们前面介绍的所有攻击都发生在虚拟世界中，即将一张图片读入电脑之后才将噪声加上去。事实上，攻击也可能发生在现实世界中。以人脸识别为例，假如我们想对这个系统做一些攻击的话，按照以前的方法，我们需要对这个人脸识别系统进行访问，并在数据中加入噪声。但显然这样的攻击方法不是最好的，那么我们能否在现实世界中加入“噪声”进行攻击呢？答案是可以的，有篇文章发现可以制造神奇的眼镜，戴上这个眼镜以后我们就可以去欺骗人脸识别系统，如下图所示。这个眼镜看起来没有什么特别的，但是左边男人戴上这副眼镜以后人脸识别系统就会觉得他是右边这一个知名女艺人。



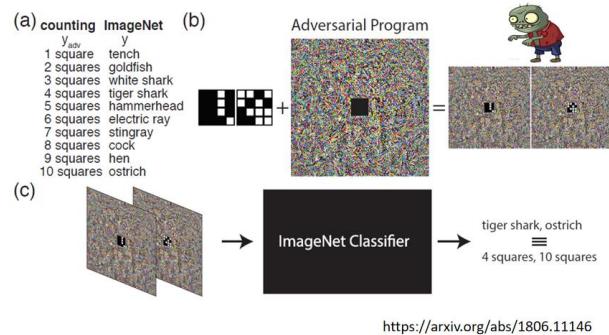
当然，在真实世界的攻击中，人们需要考虑的因素肯定更多的。首先，对于一个真实世界的物体，人们需要从多个角度进行观察，其次，在真实世界中很多事物都是有限的，比如摄像头的清晰度有限，因此它比虚拟世界的攻击会更加困难。

除了人脸识别系统可以被攻击之外，在现实世界中，交通公路标识牌也可以被攻击。如下图所示，我们可以在“STOP”标识牌上贴一些贴纸，这样我们的识别系统不管从哪种角度看这个“STOP”的标志牌，都会识别为限速 45 公里每小时，而不是之前的停车标志。



不过有的研究者认为，也许贴这种贴纸到路牌上面，所有人就都知道我们要做攻击了，或许过一阵子就有人将贴纸清理掉了。于是有人提出了第二种更隐蔽的攻击方式：将限速 35 千米每小时的数字“3”拉长一点，或许图像识别系统会将这个“3”识别成“8”。

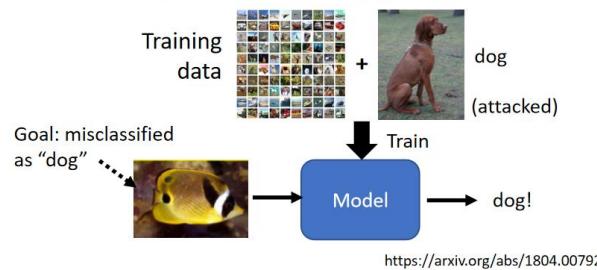
除了这些攻击方式之外，还有一些其它的攻击方式，比如对抗重编程（Adversarial Reprogramming），它会像一个僵尸一样，在原来的图像识别系统中“寄生”，让它做其原来不想做的事情。如下图所示，对抗重编程想做一个方块的识别系统，去数图片中方块的数量，但它不想训练自己的模型，它希望寄生在某一个已有的并且训练在 ImageNet 的模型上面。对抗重编程希望输入一张图片，这个图片里面如果有两个方块的时候，ImageNet 那个模型就要输出“Goldfish”（金鱼），如果 3 个方块就输出“White Shark”（白鲨），如果 4 个方块就输出“Tiger Shark”（老虎鲨），以此类推。这样它就可以操控这个 ImageNet 的模型，让它做它本来不想做的事情。具体的方法是就把要数方块的图片嵌入在这个噪声的中间，并且在这个方块的图片的周围加一些噪声，再把加噪声的图片丢到图像分类器里面，原来的图像分类器就会输出我们想要的结果。



<https://arxiv.org/abs/1806.11146>

此外，还有一种更加惊艳的攻击方式名叫后门攻击（Backdoor Attack），与之前在测试阶段进行攻击不同，它在模型的训练阶段就可以展开攻击。

- Attack happens at the training phase



<https://arxiv.org/abs/1804.00792>

举例来说，假设我们要让一张图片被识别错误，从鱼被误判为狗。最“愚蠢”的方法就是直接在训练集里面加很多鱼的图片，并且把鱼的图片都标注的为狗，但是这种方法显然是

行不通的，因为肯定会有对数据进行检查。因此，我们一定是需要拿“正确”的图片和标签进行训练的。之所以我们给“正确”二字打上了引号，是因为有些资料在人看起来是没有问题的，但实际上是有问题的，这些数据会让模型在训练的时候开一个后门，让模型在测试的时候识别错误，而且只会对某一张图片识别错误，对其他的图片识别没有影响。这种攻击方式是非常隐蔽的，因为我们的训练数据是正常的，而且我们的模型也是正常的，直到有人拿这张图片来攻击你的模型的时候，我们才会发现这个模型被攻击了。

当然这种攻击方式是非常危险的，比如说，人们可以利用一些手段获取到人脸数据集，并对一个人脸识别系统进行攻击，从而使用他人信息做一些违法的行为，这是非常可怕的。所以我们在使用别人的数据集的时候，一定要小心，要看一下这个数据集是不是有问题，是不是有“后门”。当然这个基于后门的攻击也不是那么容易成功的，里面会有很多的限制，比如模型和训练方式都会直接影响基于后门的方法能否成功。

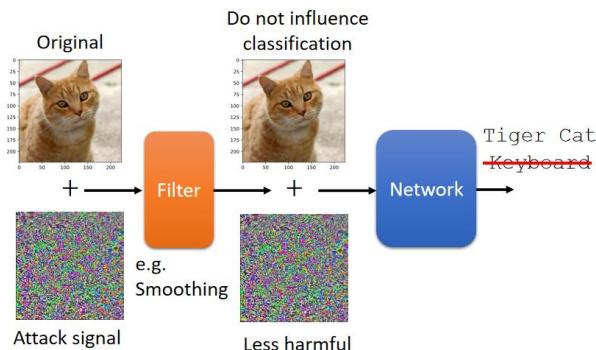
13.3 机器学习中的防御

本章到目前为止介绍的都是一些机器学习中的攻击方式，那么有了攻击方就一定会有防御方，在本章的最后，我们将对机器学习中的防御方式进行介绍。

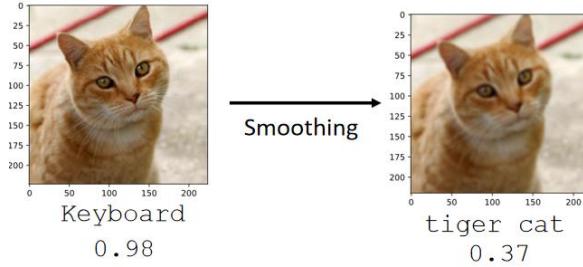


13.3.1 机器学习中的被动防御

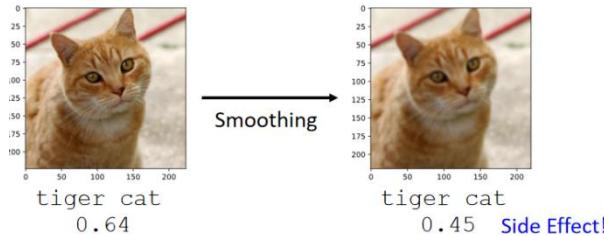
对于机器学习的攻击，人们最自然能想到的防御方式就是直接给模型加上一个“盾牌”，这种防御方式叫做被动防御。事实上，防御方式分为主动防御（Proactive Defense）和被动防御（Passive Defense）两种，我们将分别对这两种防御进行介绍。



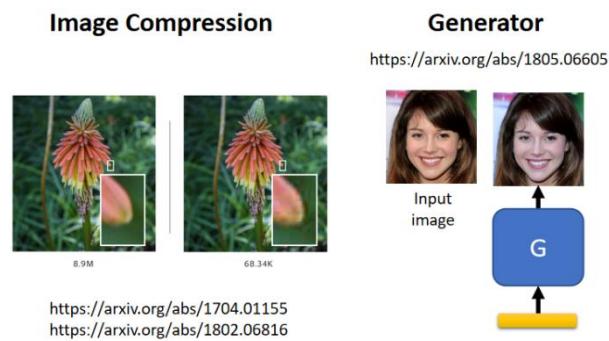
在被动防御中，“盾牌”就是一个滤波器（Filter）。我们知道加上攻击信号的图片可以“骗过”网络，但是这个滤波器可以削减攻击信号的威力，并且它不会对原来的图片产生影响。而这个滤波器的设计一般来说也非常简单，我们只需要把图片稍微做一些模糊化就可以得到非常好的防御效果。



如上图所示，左侧是系统将猫识别为键盘的图片，这是被攻击后产生的结果，当我们将这张图片做一个轻微的模糊化之后，我们就会发现识别结果变成是正确的老虎猫了。这种方法可行的原因在于，只有某一个方向上的某一种攻击信号才能够成功地对图像进行攻击，我们只需要在这个特定的噪声上做一些模糊化就可以使攻击不再奏效。



当然这个方法并不总是奏效的，它也有一些副作用，当我们在本来还没有完全被攻击的图片上做一些模糊化时，虽然还是可以正确识别，但是它的置信的分数就下降了。因此，对于模糊化这种技术，我们要有限制的去使用，使用的太过分的话它就会造成副作用，导致原来正常的图像也会识别出错。



事实上，除了模糊化的技术，还有一些其它的被动防御方法，有一系列的方法是直接对图像做压缩再解压缩。比如当我们把一张图片存储成 JPEG 格式后，它就会失真，那或许这种失真就可以让被攻击的图片失去它的攻击的威力；此外还有一种基于生成的方法，即给一张图片，然后让生成器生成一张和输入一模一样的图片。对生成器而言，它在训练的时候从来没有看过某些噪声，因此它只有很小的概率可以复现出可以攻击成功的噪声。这样我们的原始模型就不会被攻击了。

但是被动防御这个技术是有一个很大的弱点的。事实上，如果攻击方知道了防御方采用了模糊化的防御技术的话，那防御就失去作用了。因为我们完全可以将模糊化的图片当成神

经网络中多加入的第一层，并且在攻击时释放一个与防御相反的信号，就可以完美躲过防御方式了。因此，或许我们的防御可以“主动出击”，这就是我们接下来要介绍的另一种防御方式——主动防御。

13.3.2 主动防御

主动防御的思路是，在训练模型的时候，一开始就要训练一个比较不会被攻破的强鲁棒性的模型，这种的训练方式被称为对抗训练。对抗训练即在训练的时候，不要只用原始的训练数据，还需要加入一些攻击的数据。

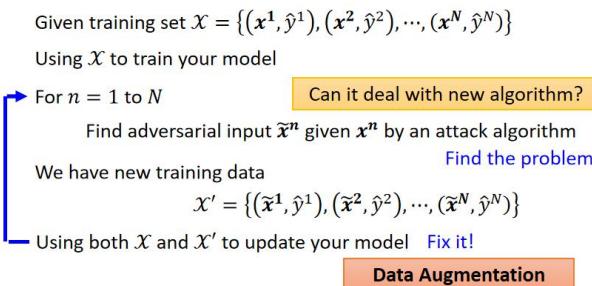
主动防御的做法十分自然：首先利用训练集训练出一个模型：

$$\mathbf{X} = \{(\mathbf{x}^1, \hat{\mathbf{y}}^1), (\mathbf{x}^2, \hat{\mathbf{y}}^2), \dots, (\mathbf{x}^N, \hat{\mathbf{y}}^N)\}$$

接下来在训练的阶段就对模型进行攻击，即对训练集 \mathbf{X} 中的数据制造一些攻击信号，得到一批用于攻击的数据 $\tilde{\mathbf{X}}$ ；然后给这些被攻击过的图片标上正确的标签，得到“新的训练资料” \mathbf{X}' ：

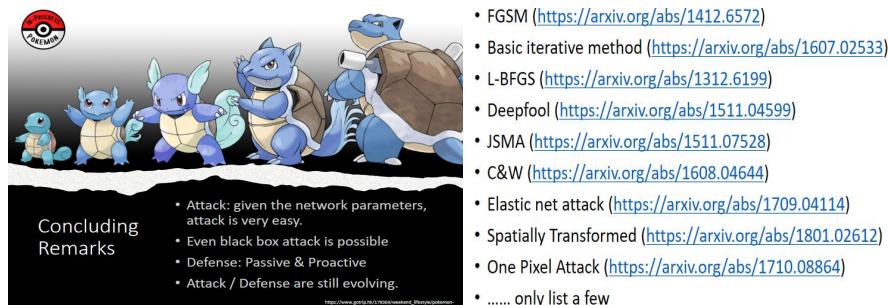
$$\mathbf{X}' = \{(\tilde{\mathbf{x}}^1, \hat{\mathbf{y}}^1), (\tilde{\mathbf{x}}^2, \hat{\mathbf{y}}^2), \dots, (\tilde{\mathbf{x}}^N, \hat{\mathbf{y}}^N)\}$$

我们将数据集 \mathbf{X} 和 \mathbf{X}' 结合在一起，用于更新模型，从而得到一个具有自我防御能力的模型，这种方法可以被视为一种数据增强的方法，所以有的研究者也会把对抗训练当做一个单纯数据增强的方法，就算没有人要攻击我们的模型，我们还可以用这样的方法产生更多的数据，并用于训练。这样可以让我们原始的模型，泛化性能更优。具体流程如下所示：



当然对抗训练也有一个非常大的缺点，就是它不一定能阻挡得住新的攻击方式，比如我们在寻找数据集 \mathbf{X}' 时所用的算法和实际攻击的算法不一样时，这样往往也可能让攻击方得逞；此外，对抗攻击需要非常大的运算资源去寻找这个被攻击过的数据集 \mathbf{X}' 。

总的来说，在机器学习模型中攻击远比防御容易，不管是主动防御还是被动防御目前都有相应的不足之处，因此学术界会不断有新的攻击和防御的方法被提出，它们仍然在以对手的方式独自进化中，感兴趣的读者可以进行探究。



14 迁移学习 (Transfer Learning)

我们在第 4 章曾对迁移学习 (Transfer Learning) 这一技术有所提及，但我们当时并没有具体展开介绍。事实上，迁移学习指的是，假设现在我们手上有一些与我们现在需要做的任务不相关的数据，我们试图利用“不相关”的数据尽可能完成手上的任务。



如上图所示，假如我们在做一个猫狗分类的问题，这种“不相关”可能体现在：一样的输入域，但是任务不同，如下图左所示：我们的输入域都是动物图片，但是任务的标签不同，因为我们需要分类大象和老虎。另一种可能是，任务相同，如下图右所示，我们仍然是要对猫狗进行分类，但输入数据的分布域不同，这体现在输入的是动物的卡通图片。



迁移学习把任务 A 开发的模型作为初始点，重新使用在为任务 B 开发模型的过程中。换言之，迁移学习是通过从已学习的相关任务中转移知识来改进学习的新任务。

之所以需要迁移学习是因为，在有些任务中，我们无法收集太多想要的数据，但是存在很多不直接相关的其他数据，如下图所示：

Task Considered	Data not directly related
Speech Recognition Taiwanese	English Chinese
Image Recognition Medical Images	
Text Analysis Specific domain	

其实在现实生活中，我们所说的“触类旁通”就有点迁移学习的意思，有时候，不同职业的人，比如运动员职业不同，但他们所干的工作可能都差不多，比如篮球运动员每天需要投 1000 次篮；而乒乓球运动员你每天要打几百筐球，虽然他们所做的事情看起来不一样，但本质上他们都是在加强自己的基本功，从而能在比赛中更有可能获得好成绩。

在本章中，我们主要把迁移学习分为 4 大类。在迁移学习中，有一些目标数据 (Target Data)，就是和你现在的任务有直接关系的数据，还有很多源数据 (Source Data)，就是和你现在的

任务没有直接关系的数据。我们可以按照源数据，目标数据是否有标签（labelled/unlabelled）分为如下 4 类，并且我们在这里对于 4 类迁移学习问题的具体做法给出一些大致的介绍。

		源数据	
		有标签	无标签
目标数据	有标签	微调（Fine-tune） 多任务学习（Multi-task Learning）	自学式学习（Self-taught Learning）
	无标签	域对抗训练（Domain-adversarial training） 零样本学习(Zero-Shot Learning)	自学式聚类（Self-taught Clustering）

我们在后续小节中将对微调，多任务学习，域对抗训练和零样本学习分别进行介绍，而至于自学式学习和自学式聚类，感兴趣的读者可以自行阅读以下两篇论文：

<https://ai.stanford.edu/%7Ehlee/icml07-selftaughtlearning.pdf> ——自学式学习

<http://www.machinelearning.org/archive/icml2008/papers/432.pdf#:~:text=Self-taught%20clustering%20is%20an%20instance%20of%20unsupervised%20transfer,auxiliary%20data%20can%20be%20differ-ent%20in%20topic%20distribution.> ——自学式聚类

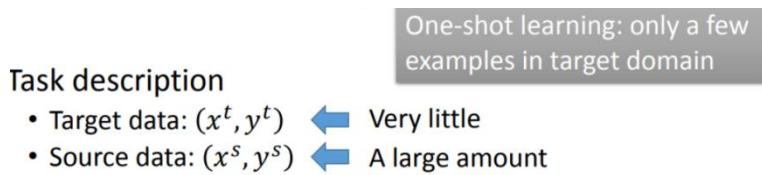
14.1 模型微调（Model Fine-tune）

如下图所示，假设有一组源数据和一组目标数据，它们都是有标签的，如下所示：

Source Data: (x^s, y^s)

Target Data: (x^t, y^t)

在上面的数据中，如果目标数据的量比较多的话，我们就可以直接使用目标数据，按照机器学习的方法去训练即可，因此我们假定源数据有很多，而目标数据很少，这时候需要用到迁移学习。需要注意的是，如果我们的目标域（Target Domain）中只有一点点例子，我们可以将这种问题称为一次性学习（One-shot Learning）。



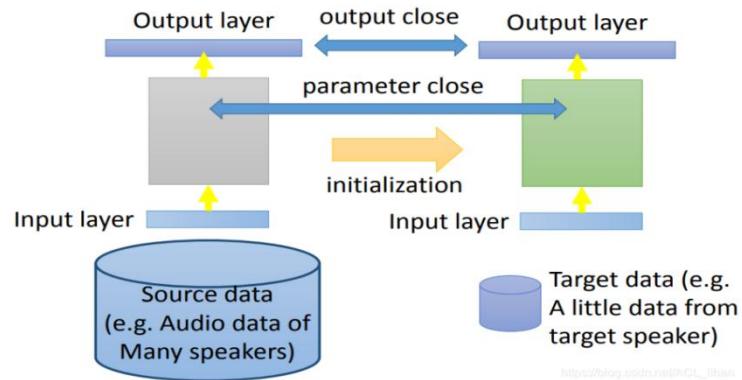
模型微调的做法其实非常简单：首先用源数据作为训练数据，去训练模型。其次，我们将第一步训练而出的模型的参数作为此时的初始值，改成用目标数据去训练模型。

这个方法很简单，但是要注意，如果目标数据的数据非常少，需要加一些技巧来防止模型发生过拟合。

14.1.1 技巧 1——保守训练（Conservative Training）

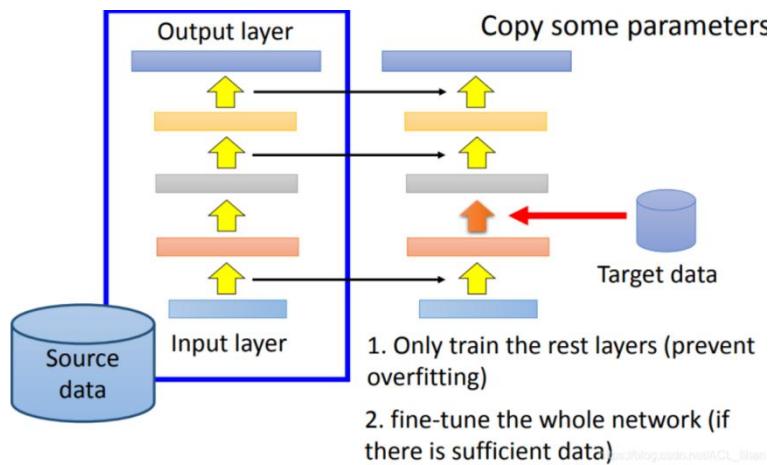
我们以语音识别问题为例来讲解保守训练。假设我们有大量的语音源数据以及一小部分

目标语音数据。在这种情况下，如果直接用目标语音数据进行训练会得到一个糟糕的结果。因此，我们在训练时可能会做得“保守”一点：即在训练时给模型加一些限制，让两边产生的模型不要相差太远。

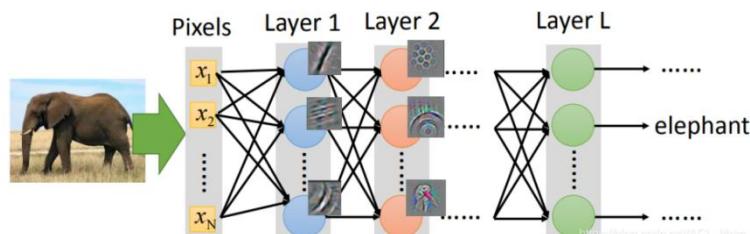


14.1.2 技巧 2——层迁移（Layer Transfer）

第二个技巧是层迁移（Layer Transfer），假如我们现在有一个用源数据训练好的模型，层迁移的做法是：将训练好的模型中的一些层复制到新的模型中，我们在新模型中只需要训练其他的层即可，这个做法只需要考虑非常少的参数，从而避免过拟合。当然如果你的目标数据足够多的话，也可以选择微调整整个网络。

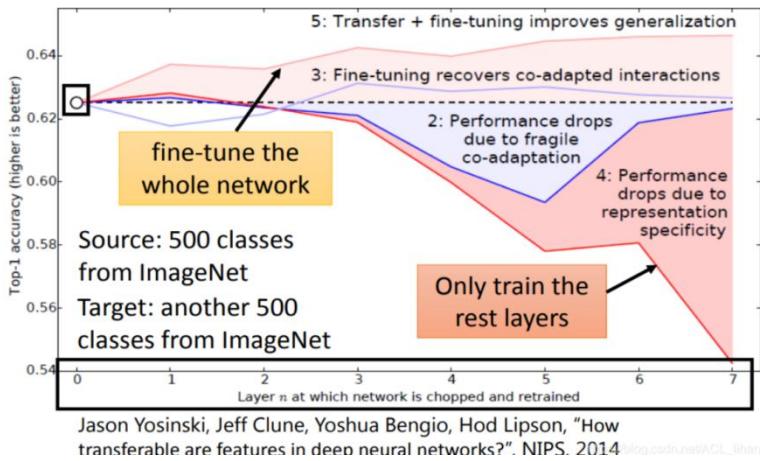


现在层迁移的问题在于，哪些层是可以被复制的呢？事实上，这一问题对于不同的问题答案是不同的。对于语音识别，通常复制后几层，因为靠近输入的几层会受不同人的声音信号所影响。而越到后面，和人的发音方法越不相关，所以可以保留。而对于图像识别，通常复制前面几层，因为靠近输入的几层只是识别一些简单的特征（直线、斜线等），所以不同的任务在前几层做的事情都差不多。



下图是将层迁移运用到 ImageNet 图像数据集中的实验结果，我们发现：复制越多层，效

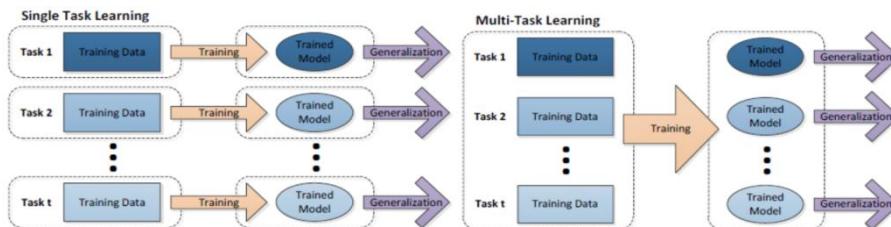
果慢慢变差。但是，如果我们在复制了不同层数的同时进行了微调，结果是随着复制层数的增多而越来越好的。总而言之，这个实验证明了，对于图像数据，我们通常只需要复制前面几层网络即可。



14.2 多任务学习 (Multi-task Learning)

14.2.1 多任务学习的概念

在汉语中，与“多”相对应的词是“单”。在了解多任务学习之前，我们先来看看什么是单任务学习。事实上，我们之前接触过的大部分深度学习问题都是单任务学习，比如情感分类，在单任务学习中，只有一个损失函数，它与我们的学习任务是对应的。



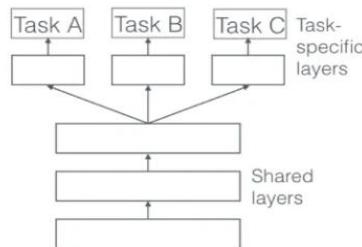
但是对于一些比较复杂的问题，假如我们采用单任务学习的方法去求解，我们很自然地会想到将其分解为简单且相互独立的子问题来单独解决，然后再合并结果，得到最初复杂问题的结果。这样做看似合理，其实是不正确的，因为现实世界中很多问题不能分解为一个一个独立的子问题，即使可以分解，各个子问题之间也是相互关联的，通过一些共享因素或共享表示（share representation）联系在一起。把现实问题当做一个个独立的单任务处理，忽略了问题之间所富含的丰富的关联信息。多任务学习就是为了解决这个问题而诞生的。

多任务学习的定义是：基于共享表示（shared representation），把多个相关的任务放在一起学习的一种机器学习方法。其目标是使多个任务在学习过程中，共享它们所学到的信息，从而能取得更好的泛化（Generalization）效果。

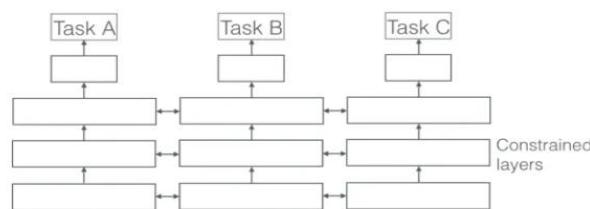
14.2.2 多任务学习的基本框架

多任务学习的本质是共享表示（Shared Representation），从而提高模型的泛化能力。对于多任务学习，有两种常见的共享方式：硬参数共享（Hard parameter Sharing）和软参数共

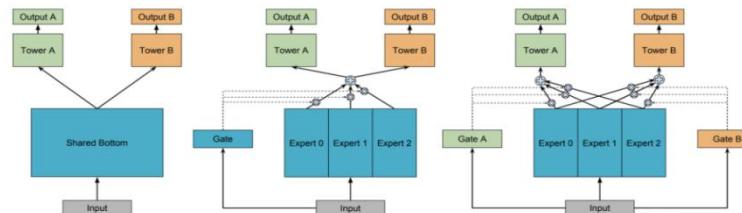
享 (Soft parameter Sharing)



第一种方法被称为硬参数共享 (Hard parameter sharing) , 如上图所示, 无论最后有多少个任务, 底层参数统一分享, 顶层参数各个模型各自独立。由于对于大部分参数进行了共享, 模型的过拟合概率会降低, 共享的参数越多, 过拟合几率越小, 共享的参数越少, 越趋近于单个任务学习分别学习。



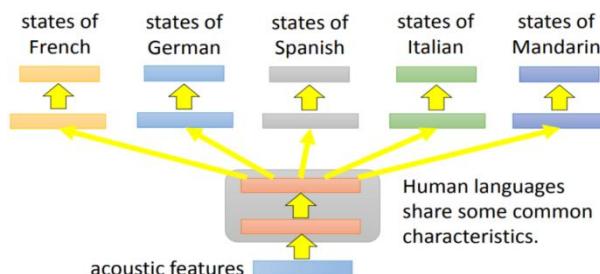
与硬参数共享所对应共享方法叫做软参数共享 (Soft parameter sharing) , 这也是现在多任务学习研究重点倾向的方法。其思想是: 从底层参数开始, 各个任务就相互独立, 当然也有一些变种, 比如底层共享一部分参数的同时, 也有相互独立的部分, 或者是底层不同, 中间互相共享, 顶层又是相互独立, 如下图所示:



总而言之, 多任务学习的共享模式就好比“切一块千层蛋糕”分给几个人, 切割的方法完全可以由自己决定, 唯一需要保证的是每个人都能分到蛋糕就行了。

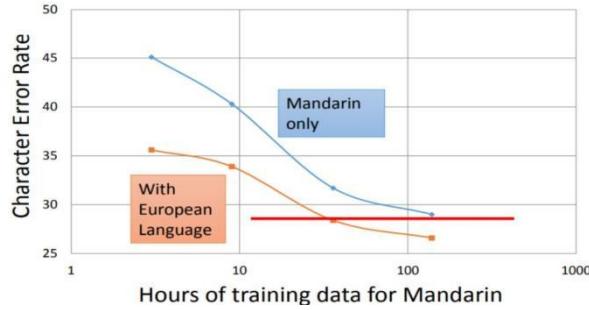
14.2.3 多任务学习的应用

多任务学习的一个比较成功的应用是: 多语音的语音辨识。假设我们有不同语言的语音数据, 我们可以训练一个能同时识别多个语言的模型, 因为虽然语言不同, 但声音的信号都差不多, 所以前面几层可以共用, 等到需要具体分语言时再拆分成多层即可。



你可能会觉得有点奇怪, 因为从语言学的角度来看, 中文和前面四个 (法语, 德语, 西

西班牙语，意大利语）的发音习惯有所不同，但是把欧洲语言和普通话一起做多任务学习，让欧洲语言去帮忙调整前几层的参数，可以看到训练出来的结果会比只有普通话作为训练数据要好，如下图所示：



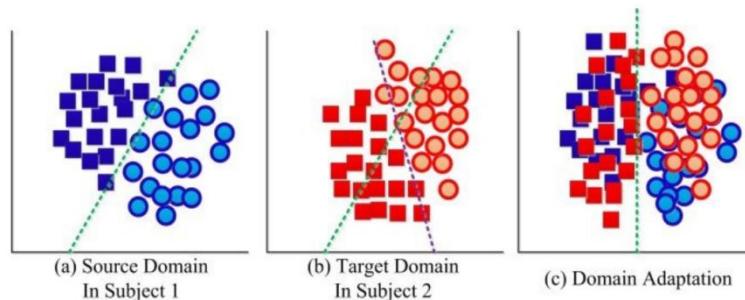
从图中横线的位置可以看出，多任务（语言）学习可以和“Mandarin only”达到几乎相同的错误率，但是多任务学习使用的训练数据明显更少，类似的思想也可以用在多语言翻译上，感兴趣的同学可以自行研究。

14.3 域对抗训练

第三种迁移学习的做法是域对抗训练，它是基于源数据有标签，目标数据没有标签的问题的一种做法。在本节中，我们将通过解读一篇对抗迁移学习领域中一篇很经典的论文：[Domain-Adversarial Training of Neural Networks \(DANN\)](#)，来对这一种方法进行讲解。

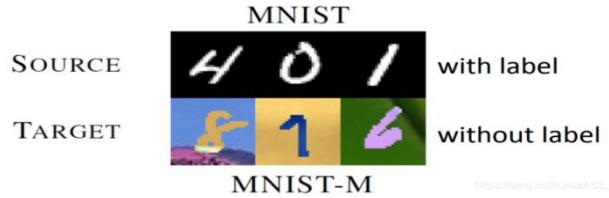
14.3.1 域对抗训练的提出背景

事实上，域适应（Domain Adaption）是迁移学习中一个重要的分支，目的是把具有不同分布的源域(Source Domain) 和目标域 (Target Domain) 中的数据，映射到同一个特征空间，寻找某一种度量准则，使其在这个空间上的“距离”尽可能近。然后，我们在源域（带标签）上训练好的分类器，就可以直接用于目标域数据的分类，如下图所示：

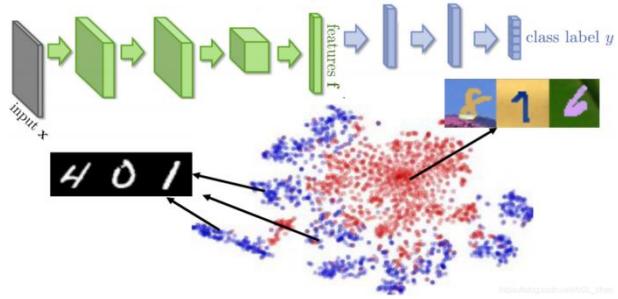


14.3.2 域对抗训练使用的原因

那么我们为什么需要用域对抗训练的技术呢？以一个具体的例子做说明，假设我们的源数据是 MNIST 数据集，它是有标签的；目标数据是 MNIST-M 数据集，它是没有标签的。这时可以把源数据看做训练数据，而目标数据看做测试数据。但是这里有一个问题是，这两者的特征不是很像，如下图所示：



我们将模型的前几层提取的特征显示出来，可以看到源数据和目标数据的特征是很不一样的（下图图中的红色和蓝色点有明显的分界线）。所以如果用源数据去学习一个模型后直接以目标数据作为输入进行预测，结果肯定是很差的。

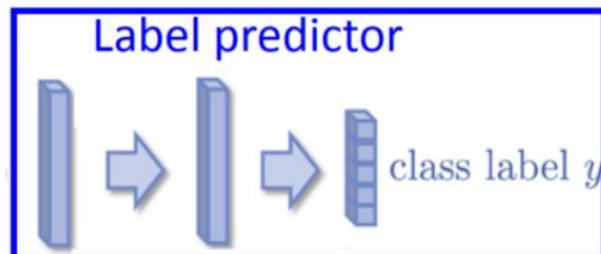


14.3.3 域对抗训练的组成

针对上述问题，我们需要利用一个特征提取器（Feature Extractor），可以将域的特性去除，这个方法就叫做域对抗训练（Domain-adversarial Training），我们希望特征提取器可以将不同域的数据“混在一起”。具体的做法是，在特征提取器之后接一个域分类器（Domain Classifier），将特征提取器得到的输出，输入域分类器中最终得到一个域标签 d ，如下图所示：

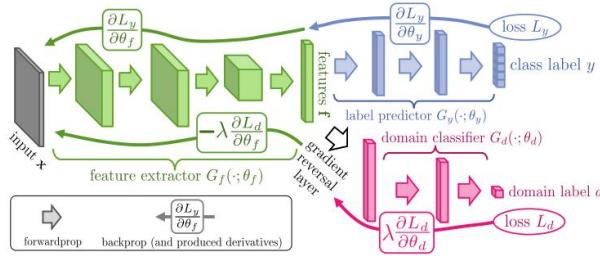


但是这里有一个漏洞：如果我们仅仅是像 GAN 一样将目标定为“欺骗”域分类器从而得到一个比较低的准确率的话，我们完全不需要做得太复杂，只需要让特征提取器不管遇到什么样的输入都输出一样的值就行了，但这样显然是不行的。因此，我们仍需要加一些限制，具体做法是加入一个标签预测器（Label Predictor），使特征提取器在“骗过”域分类器的基础上，保留原来的特性。



综上所述，域对抗训练包括：特征提取器，域分类器和标签预测器三个部分。事实上，我们可以将这三个部分组合起来，看成一个“各怀鬼胎”的大型神经网络，被称为：域对抗

迁移网络 (DANN)，其结构如下图所示：



其中，特征提取器和标签分类器构成了一个前馈神经网络。然后，在特征提取器后面，我们加上一个域判别器，中间通过一个梯度反转层 (gradient reversal layer, GRL) 连接。在训练的过程中，对来自源域的带标签数据，网络不断最小化标签预测器的损失 (loss)。对来自源域和目标域的全部数据，网络不断最小化域判别器的损失。

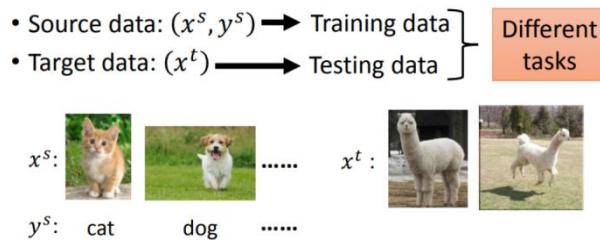
	MNIST	SYN NUMBERS	SVHN	SYN SIGNS	
SOURCE					
TARGET					
METHOD	SOURCE TARGET	MNIST MNIST-M	SYN NUMBERS SVHN	SVHN MNIST	SYN SIGNS GTSRB
SOURCE ONLY		.5749	.8665	.5919	.7400
SA (FERNANDO ET AL., 2013)		.6078 (7.9%)	.8672 (1.3%)	.6157 (5.9%)	.7635 (9.1%)
PROPOSED APPROACH		.8149 (57.9%)	.9048 (66.1%)	.7107 (29.3%)	.8866 (56.7%)
TRAIN ON TARGET		.9891	.9244	.9951	.9987

实验表明，使用域对抗训练与否对于一些特定问题而言，差距还是很大的，如上图所示，有研究对是否使用域对抗训练在不同数据集上进行了对比实验。可以看出，域对抗训练在源数据有标签，目标数据无标签的这类问题中是一个可行的办法。

14.4 零样本学习 (Zero-shot Learning)

在本章最后，我们将对零样本学习 (Zero-shot Learning) 进行介绍。我们在前面讲解自监督学习中的 GPT 模型曾经提到过零样本学习，事实上，零样本学习也是迁移学习中一个比较常见的技巧，它同样是适用于源数据有标签，目标数据无标签的情况。

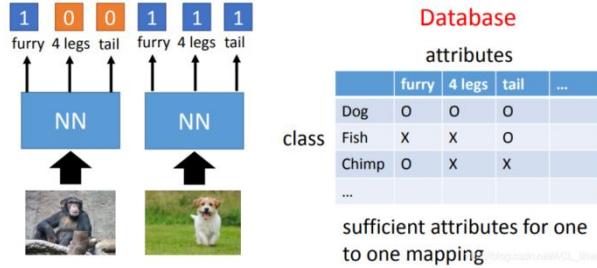
零样本学习与域对抗训练的不同之处在于：在域对抗训练中，源数据和目标数据可以分别被看成训练数据和测试数据；而在零样本学习中，源数据和目标数据的差异被定义得更加严格，因为它们可以是两种完全不同的任务，如下图所示：



这时可以有两个方法可以做，第一种是用每一类的属性分别表示该类别 (Representing each class by its attributes)，第二种方法是属性嵌入 (Attribute embedding)。我们将分别对这两种做法进行介绍。

14.4.1 用属性表示类别

用属性来分别表示类别的想法提出是很自然的：既然使用目标数据没办法直接预测图中的物体是什么，那就换一种角度，让神经网络去预测图中物体有哪些特征，根据这些特征，人工建立一个数据库，将拥有此特征的物体与类别对应起来，如下图所示：



在测试过程中，我们也只需要根据属性来查询数据库即可。

14.4.2 属性嵌入

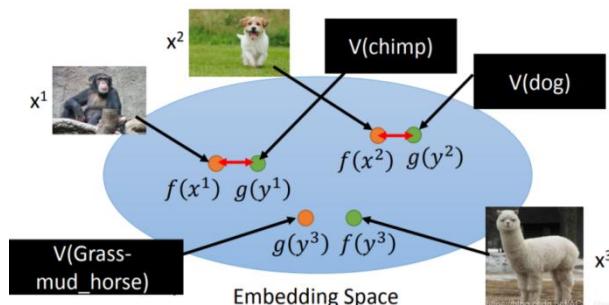
上述办法看起来是很自然的，但假如我们的属性维度比较大时，数据库的建立，查询会显得很复杂，这时，就有了第二种做法，叫做属性嵌入（Attribute Embedding），做法如下：

1. 把所有类别各自的描述（attribute，也可以理解为特征）先定义好。（这一步需要人工来定义，也可能能用机器自动去定义）
2. 把有标签的图像数据 x^i 及其描述 y^i 分别通过两个神经网络 f 和 g ，映射到嵌入空间（embedding space）上。要训练模型使得这两个输出越接近越好，优化问题如下：

$$f^*, g^* = \arg \min_{f, g} \sum_n \|f(x^n) - g(y^n)\|^2$$

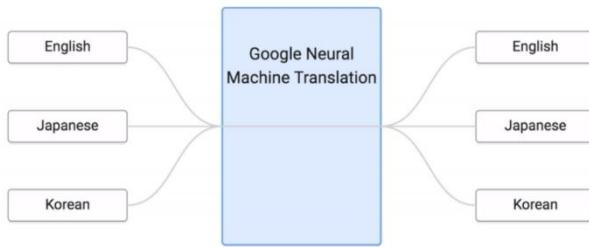
3. 对于没有标签的图像数据 x^j ，通过 f 映射到嵌入空间上，观察其与哪个描述的映射比较接近，即可得到最终结果。

回忆我们之前讲过词嵌入（Word Embedding）模型，属性嵌入也是可以与词嵌入相结合的，这种做法用在我们事先对于每一对数据都不知道它们的属性，如下图所示：



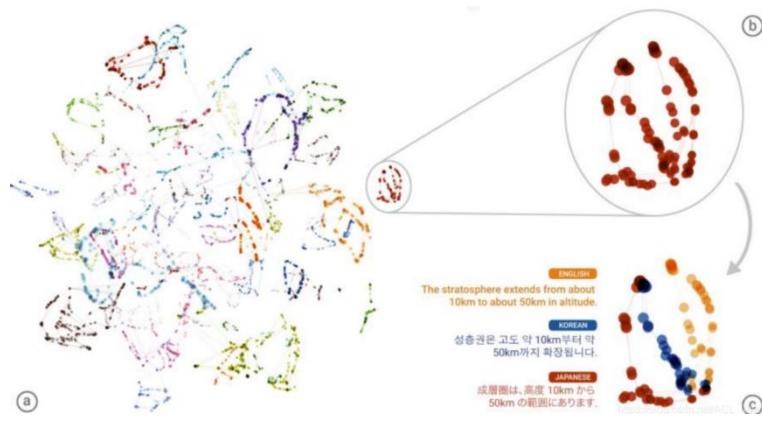
14.4.3 零样本学习的应用

对于零样本学习，一个常见的应用是（多语言）的机器翻译。如下图所示，我们的输入数据中包括英语，日语和韩语，输出也包括这三种语言。零样本学习的作用在于：即使我们的训练数据中没有日语转韩语的数据，但我们能通过零样本学习实现日转韩的翻译。



这个零样本学习的学习步骤大概可以理解为以下几步：

1. 输入的英语日语韩语到机器中。
2. 机器将数据投射到另一个空间上（转换为另一种机器自己看得懂的语言）。不同语言但是同一个意思的词在这个空间上的位置都差不多。
3. 最终输出翻译的时候把这个空间上的点转换为人类看得懂的语言。



当然，对于零样本学习，还有一些其他的应用，感兴趣的读者可以通过自行查询网络上的资料，论文等方式进行了解。总而言之，不管是零样本学习还是迁移学习，在实做中都是一个值得探究的领域。

15 强化学习 (Reinforcement Learning)

我们在本章需要介绍一个新的领域——强化学习。我们在之前在很多深度学习的例子中提到了“RL”，即强化学习（Reinforcement Learning），当时我们对这一领域的认知是：当遇到一个特别难训练的任务（比如 GAN）时，我们可以用“RL 硬做一发”。

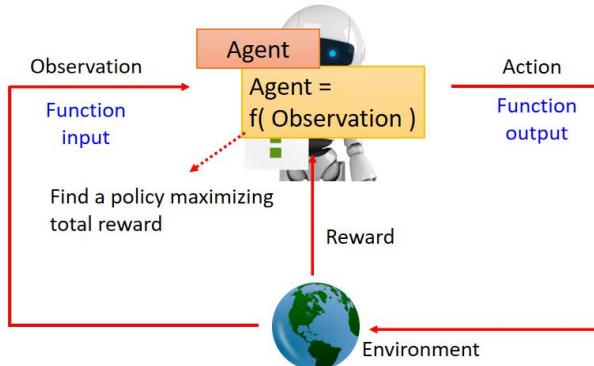
但事实上我们对强化学习这个领域的基本概念和相关算法并不了解，在本章中，我们将从强化学习的基本概念开始讲起，并对一些主流的强化学习算法进行介绍。

15.1 强化学习基本概念

15.1.1 强化学习的本质

强化学习的本质和机器学习（深度学习）是一样的，即寻找一个函数。只不过在强化学习中，我们需要寻找的函数叫智能体（Agent），他们和环境（Environment）会进行互动——即接受环境给与的观察（Observation），做出行动（Action）去影响环境，然后环境会给决策者一些奖励（Reward）从而来判断行动的好坏。

因此强化学习的目标是：找到一个用观察作为输入，智能体作为输出的函数，使得奖励最大化：



也许这样讲可能不是很容易理解，我们以一个电脑游戏（Space Invader）为例：人即决策者/智能体，向左或向右开火代表行动，游戏机就是环境，得到的分数就是奖励。游戏的画面发生变化时就代表了有了新的观察。在这个游戏中，我们想要让电脑学习出一个智能体，使得它在游戏中能获得尽可能高的分数。



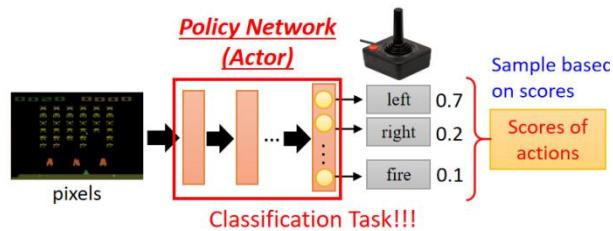
这样的任务也可以用在 AlphaGo 中，如上图所示，智能体是 AlphaGo，环境是 AlphaGo 的对手棋手，智能体的输入是棋盘上黑子跟白子的位置。一开始，棋盘上是空的，在每一回合智能体要决定下一步的落子。这个过程反复进行。在棋局中，智能体所采取的动作都无法

得到任何奖励，我们可以定义，如果赢了，就得到 1 分，如果输了就得到 -1 分，只有整场围棋结束，智能体才能够拿到奖励。智能体学习的目标是要最大化它可能得到的奖励。

15.1.2 强化学习的三个步骤

我们接下来将对强化学习的流程进行介绍，与机器学习一样，强化学习同样可以分为三个步骤：

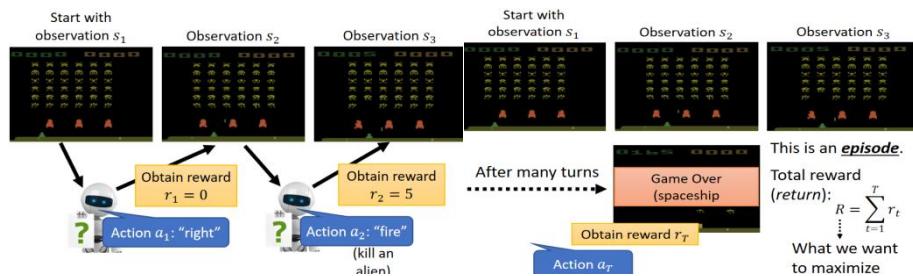
第一个步骤是定义一个含有未知参数的函数。我们知道，在强化学习中函数即为智能体，它是一个神经网络，通常被称为策略网络（Policy Network）。以 Space Invader 这款游戏为例，网络是一个非常复杂的函数，其输入是游戏画面上的像素，输出是每一个可以采取的动作的分数。



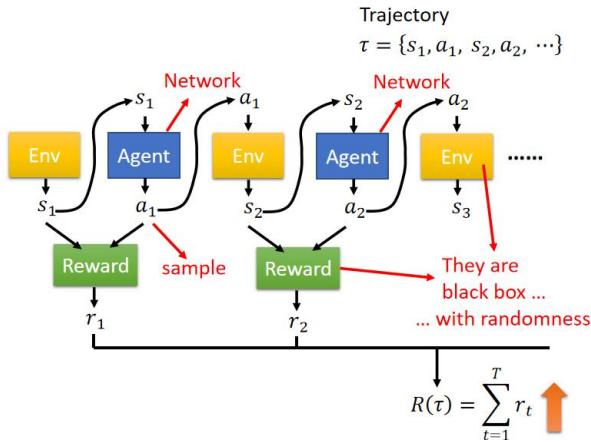
网络的架构是可以自己设计的，只要网络能够输入游戏的画面，输出动作即可。如果输入是一张图片，可以用卷积神经网络来处理。如果我们不要只看当前这一个时间点的游戏画面，而是要看整场游戏到目前为止发生的所有画面，可以考虑使用循环神经网络或 Transformer。

如上图所示，我们发现网络的输出是根据概率决定的，这一点和我们之前所讲的分类问题很像，那么为什么我们不采用分数最大的动作进行移动（输出）呢？因为在很多的游戏里面随机性是很重要的，比如玩石头、剪刀、布游戏，如果智能体总是出石头，很容易输。但如果有一些随机性，就比较不容易输。

第二步是定义强化学习的损失函数，在这里损失函数就相当于分数的获得机制。但在强化学习中，我们希望最大化我们的得分，因此我们需要将总得分取负值作为损失函数。需要注意的是，在“游戏”中的每一行为都有可能会获得奖励（Reward），而强化学习的目标函数是指整局“游戏”中获得的分数（Total Reward），一局游戏被称为一个 Episode。



第三步则是强化学习的优化问题，它所需要做的事是找到智能体的一组参数，如上图所示，强化学习的奖励是一个函数，它不仅需要考虑行动（Action），还需要考虑观察（Observation）：环境产生的观察 s_1 进入第一个智能体，并通过采样输出一个 a_1 ，再进入环境产生 s_2 ，如此反复循环，直到满足游戏结束的条件为止， s 和 a 所形成的系列被称为 Trajectory，用 τ 表示：



事实上，强化学习这三步中，最主要的难点就是如何求解优化问题，这是因为强化学习中的行动是通过采样产生的，具有很大的随机性，就算给定相同的环境，也可能产生不一样的行动。

15.1.3 强化学习 v.s GAN

强化学习的问题是如何找到一组网络参数来最大化回报。这跟生成对抗网络有异曲同工之妙。在训练生成器（generator）的时候，生成器与判别器（discriminator）会接在一起，我们希望调整生成器的参数，让判别器的输出越大越好。在强化学习里面，智能体就像是生成器，环境跟奖励就像是判别器，我们要调整生成器的参数，让判别器的输出越大越好。但在生成对抗网络里面判别器也是一个神经网络，可以用梯度下降来训练生成器，让判别器得到最大的输出。但是在强化学习的问题里面，奖励跟环境不是网络，不能用一般梯度下降的方法调整参数来得到最大的输出，所以这是强化学习跟一般机器学习不一样的地方。

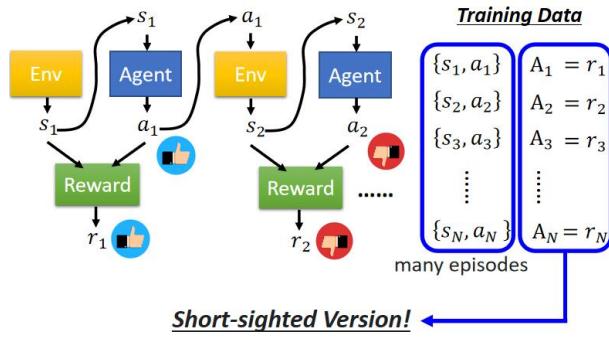
15.2 评价行为的标准

我们知道，强化学习需要训练一个“收益最大”的智能体，并且智能体和环境是成对的。事实上对于每个智能体的收益获得与否，以及收益的获得多少都是人为决定的。因此我们需要确定一个对行为（Action）的评分标准。

在强化学习的发展历史中，评价行为的标准版本经历了若干次的“更新换代”，我们将在接下来依次进行介绍。

15.2.1 版本 0

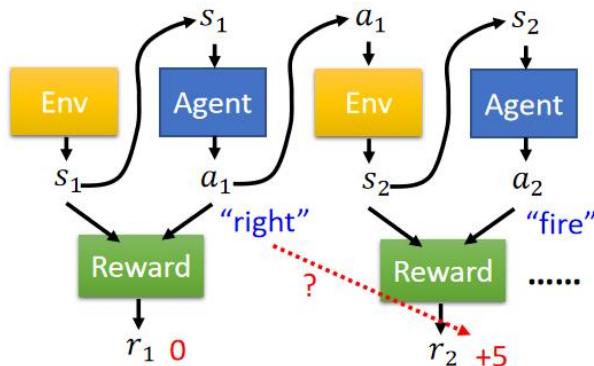
对于评价行为的标准的定义，我们先来看一个最简单的版本，在这个版本中，具体做法是：让一个随机的智能体去和环境做互动，并将每一次观察中执行的行为成对收集，作为训练数据，每一个行为对应一个奖励（Reward），我们将这个即时的奖励作为每一对状态-动作的分数 A。



如上图所示，智能体与环境的互动是多回合的，通常我们收集资料不会只把<智能体，环境>作为一个回合（Episode），而是运作多个回合从而期待收集到足够的资料。其评价方式定义如下：

假设在某一个状态 s_1 ，智能体执行动作 a_1 ，得到奖励 r_1 ，如果奖励是正的就说明该行为是好的，否则说明该行为是不好的，因此奖励可以当成 A，且 $A_1 = r_1$, $A_2 = r_2$...以此类推。

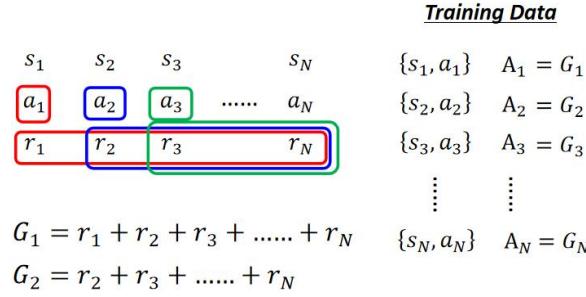
但这并不是一个好的版本，因为如果我们将奖励直接定义为 A 的话，会让智能体变得“鼠目寸光”，即它只关注了当前状态的收益，并没有纵观全局，而实际上每一个行为都不是独立的，它们会影响后续的发展。而且在跟环境做互动的时候，有一个问题叫做延迟奖励（delayed reward），即牺牲短期的利益以换取更长期的利益。比如在 Space Invader 的游戏里面，智能体要先左右移动一下进行瞄准，射击才会得到分数。而左右移动是没有任何奖励的，其得到的奖励是零。只有射击才会得到奖励，但是并不代表左右移动是不重要的，先需要左右移动进行瞄准，射击才会有效果，所以有时候我们会牺牲一些近期的奖励，而换取更长期的奖励。如果使用版本 0，左移和右移的奖励为 0，开火的奖励为正，智能体会觉得只有开火是对的，它会一直开火。



15.2.2 版本 1

我们已经知道了，对于某个时刻 t ，行为 a_t 的好坏不仅取决于当前的奖励 r_t ，而是取决于该时刻及其之后所有发生的事情。因此在改进的版本中，为了将延迟奖励机制考虑进去，采用了一种累积奖励（cumulated reward）的机制，即将每个行为的后续所有行为所获得的奖励也加入到该批次来得到数值，用 G_t 表示：

$$G_t = \sum_{i=t}^N G_i$$

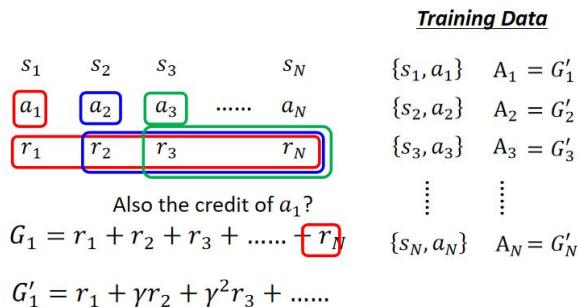


但是这一版本也有一些不合适的地方，假如我们的游戏周期 N 过长，那么将临近游戏结束时获得的收益 r_N “归功” 到智能体 a_1 处也并不合适，因为前面的智能体可能对太靠后时刻的收益获得并不会产生太多影响。

15.2.3 版本 2

对版本 1 中的累积奖励机制稍作改进就得到了版本 2：由于前面时刻的智能体对后面时刻的收益的影响并没有那么大，因此版本 2 中提出了一种“加权”的算法，具体做法是赋予一个折扣因子（discount factor） γ ，其值在 0-1 之间（通常设为 0.9, 0.99...），每经过一个时刻，其收益就会打一个折扣，如下所示：

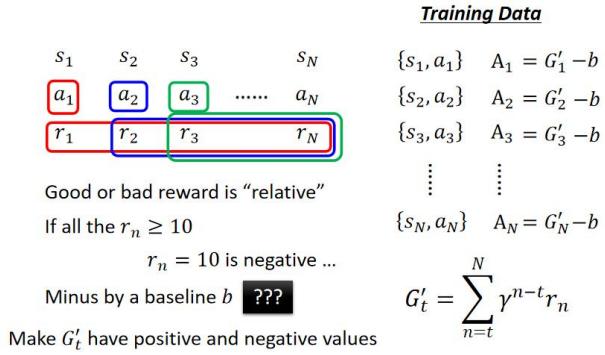
$$G_t' = \sum_{i=t}^N \gamma^{i-t} \cdot r_i$$



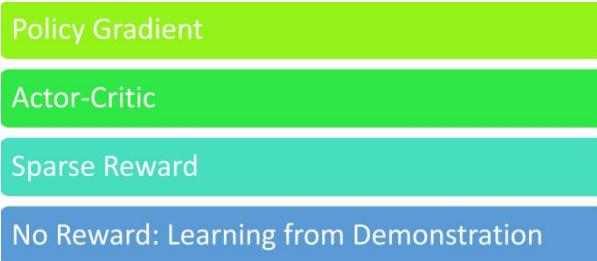
这样做好处是，当时间越来越长时，离采取的动作越远的奖励对当前采取的动作的影响会变得越来越小（接近于 0），这一操作对于一些“局末结算”的游戏也是可行的：智能体采取一系列动作，只要最后赢了，这一系列动作都是好的；如果最后输了，这一系列动作都是不好的。最早版本的 AlphaGo 采用这种方法训练网络，但它还有一些其它的方法，比如价值网络（value network）等等。

15.2.4 版本 3

当然上述版本仍然可以做进一步改进。考虑这样一个问题：如果所有的行动都能获得一个正的分数，只是大小不同，这样可能无法判断哪些行为是不好的（好的行为还是可以进行判断，只需要比较收益的大小即可），因此我们需要对奖励做一个标准化，最简单的做法是将所有获得的奖励统一减去一个基线（baseline） b ，经过这样的处理之后，奖励值就会出现有正有负的情况。



事实上，强化学习中的策略梯度算法（Policy Gradient）使用的就是上面的这种评价标准，当然评价标准还有继续改进的空间，而且除了策略梯度算法之外，强化学习还有一些其他的算法：行动者-批评者算法（Actor-Critic），稀疏奖励的处理方法以及模仿学习，如下图所示：



需要注意的是，在行动者-批评者算法中，对于评价行为的标准还提出了两个改进：版本 3.5 和版本 4，我们将在接下来对前两种算法分别进行介绍，至于后两种算法，感兴趣的读者可以自行阅读《Easy RL：强化学习教程》这本书。

15.3 策略梯度算法

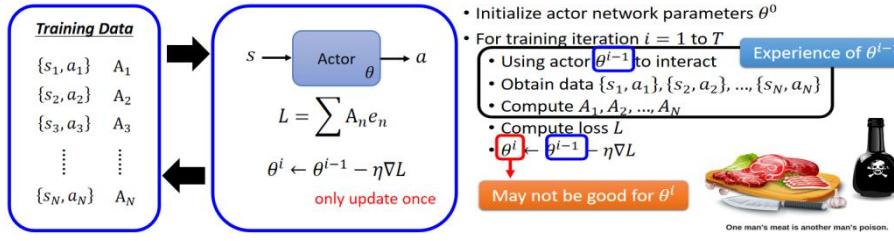
15.3.1 策略梯度的流程

策略梯度的流程是：首先要随机初始化智能体，给智能体一个随机初始化的参数 θ_0 ，接下来进入训练迭代阶段，假设要跑 T 个训练迭代。一开始智能体什么都不会，其采取的动作都是随机的，但它会越来越好。智能体去跟环境做互动，得到一大堆的<状态，动作>，并且我们需要用 A_1, A_2, \dots, A_N 决定行为的好坏。接下来利用与梯度下降类似的方法更新参数，具体算法流程如下：

- Initialize actor network parameters θ^0
- For training iteration $i = 1$ to T
 - Using actor θ^{i-1} to interact
 - Obtain data $\{s_1, a_1\}, \{s_2, a_2\}, \dots, \{s_N, a_N\}$
 - Compute A_1, A_2, \dots, A_N
 - Compute loss L
 - $\theta^i \leftarrow \theta^{i-1} - \eta \nabla L$

需要注意的是，在该算法中，数据的收集是在训练的循环中进行的，这一点与一般的训练任务是不同的（一般的训练任务收集数据都是在训练迭代之外）。这是因为同一个行为对于不同的智能体的好坏是不一样的，因此在某一时刻 i , θ_i 收集到的资料只能用于更新自己的

参数，不一定适合更新 θ_{i+1} 的参数，这就是为什么强化学习往往在训练过程中非常花费时间。

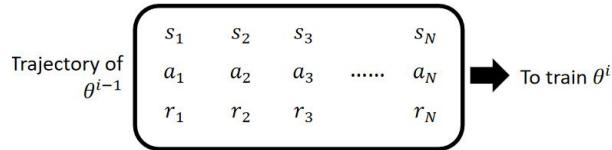


15.3.2 同策略 (On-policy) v. s. 异策略 (Off-policy)

对于强化学习难以训练的这一问题，有一种名叫异策略学习（Off-policy Learning）的方法可以解决。在讨论该问题之前，我们先来了解一下什么是异策略学习，以及与异策略学习相对应的同策略学习（On-policy Learning）。

同策略学习是指：需要更新的智能体与环境互动的智能体是一样的，与之对应的异策略学习就是指二者是不一样的。事实上我们之前讨论的都是同策略学习，而异策略学习的好处在于：搜集依次资料可以更新多次参数，而不是一笔资料只能更新一次参数。

- The **actor to train** and the **actor for interacting** is the same. → On-policy
- Can the **actor to train** and the **actor for interacting** be different? → Off-policy



In this way, we do not have to collection data after each update.

异策略学习的一个代表性的做法是 PPO，全称是 Proximal Policy Optimization，在这边我们不会对 PPO 的算法原理进行详细介绍，为了使读者更容易理解这个算法，我们通过举例说明：

假设你去问一个情圣如何追女生，情圣传授给你的技巧是大胆表白，因为凡是被他告白的女生无一例外都被他追到了，但是对你来说，假如你去用情圣告诉你的方法跟女生告白，你未必能追到女生，因此，前者代表着与环境互动的智能体，后者代表真实训练的智能体，后者显然会在前者的基础之上打一个折扣。

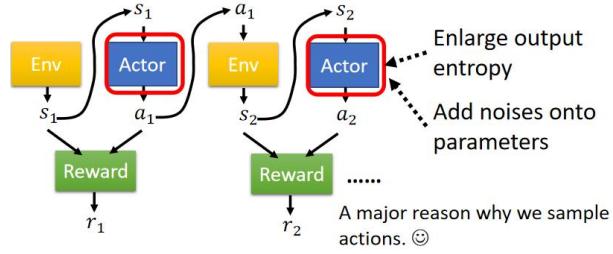


the actor to train

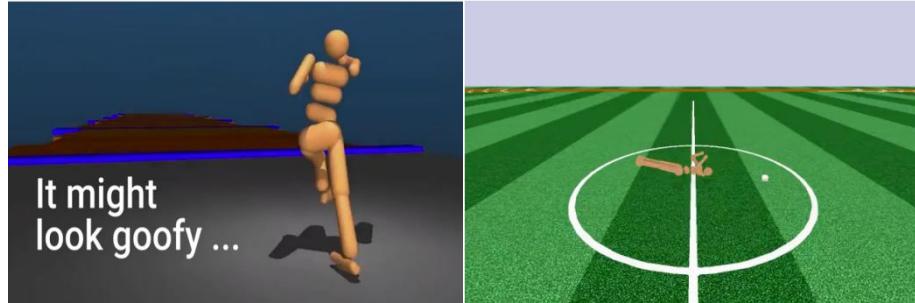
由此可见，智能体探索行为是有很大的随机性的，并且这个随机性是非常重要的。很多时候随机性不够会训练不起来或训练不出好的结果，因此探索（exploration）是强化学习训练的过程中一个非常重要的技巧：通过尽力遍历各种可能性，才能更加全面地考虑和做决策。

为了要让智能体的随机性大一点，甚至在训练的时候会刻意加大它的随机性。比如智能体的输出是一个分布，可以加大该分布的熵（entropy），让其在训练的时候，比较容易采样

到概率比较低的动作。或者会直接在这个智能体的参数上面加噪声，让它每一次采取的动作都不一样。



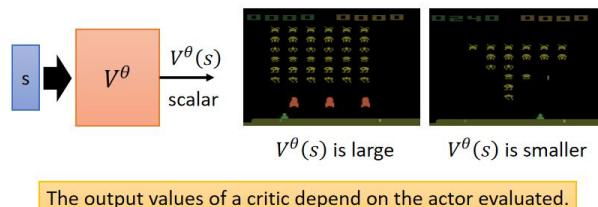
事实上，PPO 这个算法在强化学习中是很常见的，下图中就是 DeepMind(图左)和 OpenAI(图右)的实际应用：



15.4 行动者-批评者算法

15.4.1 什么是批评者 (Critic)

批评者是用来评价智能体/行动者 (Agent/Actor) 的好坏的，可以用一个价值函数 $V^\theta(s)$ 来表示：当使用参数为 θ 的行动者时，看到观察 s 后预估获得的折扣累积奖励。这个价值函数相当于“未卜先知”——看到 s 就能知道行动者会获得什么样的表现。

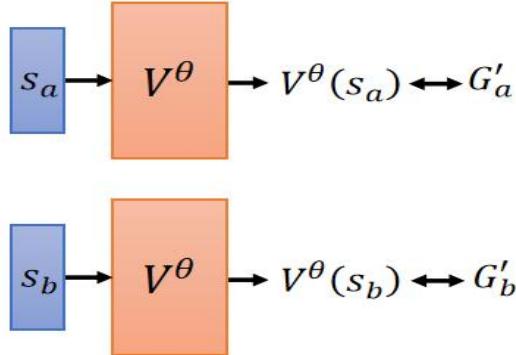


价值函数的输入是一个观察 s ，输出是一个标量，如上图所示，在 Space Invader 这款游戏中，直接预测接下来应该会得到很高的累积奖励，因为该游戏画面里面还有很多的外星人，假设智能体很厉害，接下来它就会得到很多的奖励；而当游戏快结束时，剩下的外星人不多了，可以得到的奖励就比较少。价值函数跟其观察的智能体是有关系的，同样的观测，不同的智能体得到的折扣累积奖励应该不同。

15.4.2 批评者是怎么被训练出来的？

批评者的训练有两种方法：蒙特卡洛 (Monte-Carlo, MC) 和时序差分 (Temporal-difference, TD)。智能体跟环境互动很多轮会得到一些游戏的记录。从这些游戏记录可知，看到游戏画

面 s_a , 累计奖励为 G'_a ; 看到游戏画面 s_b , 累积奖励为 G'_b 。如果使用蒙特卡洛方法, 只需要使批评者看到某个 s 后输出的值 $V^\theta(s)$ 与对应的 G' 越接近越好。

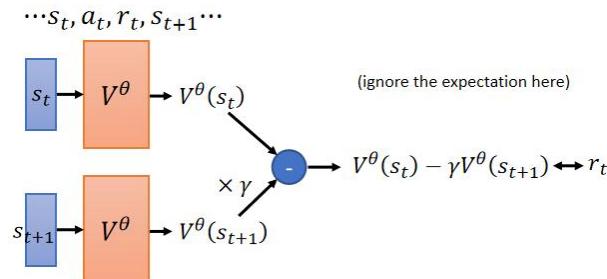


但这种做法有一个问题: 当游戏时间很长, 甚至有些游戏根本就没法结束, 一直进行下去时, 蒙特卡洛的方法就不适合这样的游戏, 因此就有了时序差分的训练方法, 它是可以在游戏还未结束就可以进行训练的。

对于时序差分的训练方法, 只要在 t 时刻看到观察 s_t , 行动者执行了行动 a_t 并获得了收益 r_t 时, 只需要看到下一时刻的观察 s_{t+1} 就能够对 $V^\theta(s)$ 进行训练了, 在时序差分中, $V^\theta(s_t)$ 和 $V^\theta(s_{t+1})$ 的关系如下:

$$V^\theta(s_t) = \gamma \cdot V^\theta(s_{t+1}) + r_t$$

因此, 假设有一笔数据为 $\{s_t, a_t, r_t, s_{t+1}\}$, 将 s_t 代入到价值函数中可以得到 $V^\theta(s_t)$, 将 s_{t+1} 代入到其对应的价值函数中可以得到 $V^\theta(s_{t+1})$, 尽管我们并不知道这两个函数的具体值, 但至少时序差分的训练方法可以让我们有一个较为明确的训练目标, 即让 $V^\theta(s_t) - \gamma \cdot V^\theta(s_{t+1})$ 和 r_t 越接近越好。



事实上, 同样的 θ 所得到的训练数据用蒙特卡洛跟时序差分计算出的价值很可能是不一样的, 以下是一个玩了 8 轮的游戏的示例, 我们发现二者算出来的价值函数是不同的。

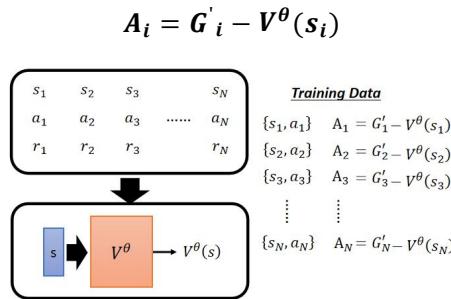
$s_a, r = 0, s_b, r = 0, \text{END}$	$V^\theta(s_b) = 3/4$
$s_b, r = 1, \text{END}$	
$s_b, r = 1, \text{END}$	
$s_b, r = 1, \text{END}$	$V^\theta(s_a) = ? \quad \text{0?} \quad \text{3/4?}$
$s_b, r = 1, \text{END}$	
$s_b, r = 1, \text{END}$	
$s_b, r = 1, \text{END}$	Monte-Carlo: $V^\theta(s_a) = 0$
$s_b, r = 1, \text{END}$	
$s_b, r = 0, \text{END}$	Temporal-difference:
(Assume $\gamma = 1$, and the actions are ignored here.)	$V^\theta(s_a) = V^\theta(s_b) + r$
	$3/4 \quad 3/4 \quad 0$

当然, 这并不代表这两种方法有对错之分, 其价值函数计算结果不同的原因只是在于: 它们背后的假设不同。对于蒙特卡罗方法, 两个相邻的观察 s_a 和 s_b 是有关联的, 因此蒙特卡

洛方法会在看到了 s_a 之后就让 s_b 的奖励为0；而在时序差分方法中，前后两个观察是没有关系的，即 s_b 的奖励不会受到 s_a 的影响，而 s_b 期望的奖励为 $3/4$ ，故而计算得到的 $V^\theta(s_a) = 3/4$.

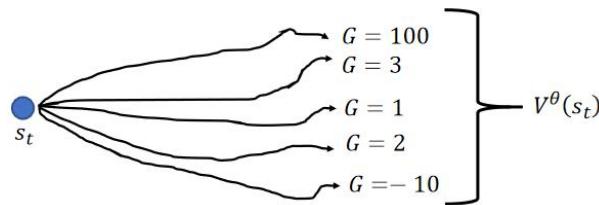
15.4.3 版本 3.5 和版本 4

在简单对批评者进行了一定了解之后，我们来介绍一下版本 3.5 的评价行为标准。我们在版本 3 中曾提及了标准化的技术，即对于每一个<状态，动作>对，将最终获得的分数减去一个基线 b 作为其获得的新分数。其中，基线 b 常见的设置方法是设置为 $V^\theta(s)$ ，即对于时刻 i ，我们有：



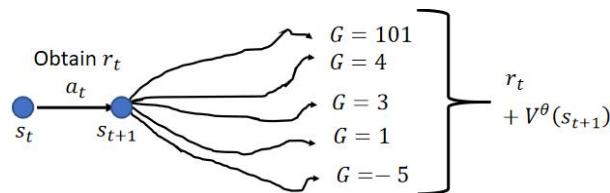
那么实际上，版本 3.5 与版本 3 的唯一区别在于，版本 3.5 是用价值函数来训练行动者的，而版本 3 中所用的算法是策略梯度算法。因此，其对于行为的好坏的判断方法仍然是观察 A_t 是否大于 0，即分数是否在平均值以上。

但是这个版本仍然存在问题，从上面的公式中可以看出：对于时刻 t ， A_t 代表着 $\{s_t, a_t\}$ 的好坏，然而游戏是具有一定的随机性的，换言之， a_t 之后的所有行为都是采样出来的，我们考察的东西是： a_t 是否是一个好的行为，不能让后面的特殊情况影响到对于当前行为的判断：



于是我们的改进就是版本 4，它也叫优势行动者-批评者（Advantage Actor-Critic）方法，具体做法是：当我们执行完 a_t 之后，得到一个即时奖励 r_t ，我们并不会直接将这个即时奖励作为该行为的奖励，取而代之的是，我们需要让“游戏继续进行下去”，到下一个观察 s_{t+1} 时，将所有可能产生的奖励值求一个平均值，即：

$$G'_t = r_t + V^\theta(s_{t+1})$$

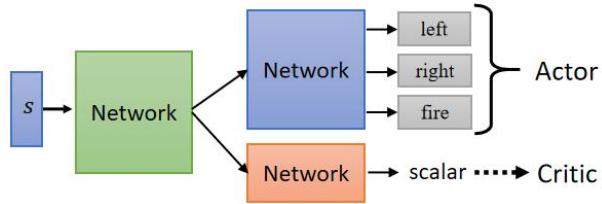


我们将上述公式代入到版本 3.5 的公式中，可以得到 A_t 的值，那么对于行为的判断就变

成了一个很自然的事情：当 G 大于 0，即 $r_t + V^\theta(s_{t+1}) > V^\theta(s_t)$ 时，则代表该行为是好的，也就是说，在优势行动者-批评者算法中， $A_t = r_t + V^\theta(s_{t+1}) - V^\theta(s_t)$ 。

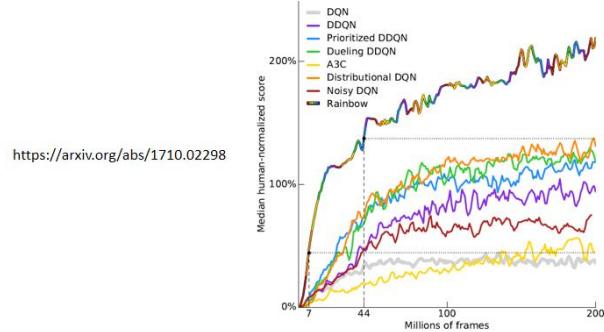
15.4.4 Actor-Critic 的训练技巧

Actor-Critic 有一个训练的技巧，事实上在强化学习中，Actor 和 Critic 都是一个网络，Actor 网络的输入是一个游戏画面，其输出是每一个动作的分数。Critic 的输入是游戏画面，输出是一个数值，代表接下来会得到的累积奖励。



如上图所示，有两个网络，它们的输入是一样的东西，因此二者应该有部分参数可以共用，尤其在输入非常复杂的情况下（比如游戏画面，前几层应该都是卷积神经网络）。所以 Actor 和 Critic 可以共用前面几个层，所以在实践的时候往往会这样设计 Actor-Critic。

当然，强化学习还可以直接由批评者决定要执行的动作，比如深度 Q 网络（Deep Q-Network, DQN）。DQN 有非常多的变形，有一篇非常知名的论文“Rainbow: Combining Improvements in Deep Reinforcement Learning”，把 DQN 的七种变形集合起来，因为有七种变形集合起来，所以这个方法称为彩虹（rainbow）。



15.5 本章小结

对于强化学习这个主题，我们在本书中并没有对其进行太深的介绍，其原因是这一领域内的学问相当深。事实上在网络上，大家可以搜索到 UCL David Silver 的强化学习课程进行学习，当然如果对于深度强化学习（Deep Reinforcement Learning）感兴趣的话，可以自行学习 UC Berkeley 的 CS285 (<https://rail.eecs.berkeley.edu/deeprlcourse/>)，当然李宏毅老师的强化学习课程也是一个不错的选择。

16 终生学习

本节我们对终生学习（Lifelong Learning）进行介绍，实际上，机器学习能够做终生学习这件事情，本质上还是基于人类对于人工智能的想象。终生学习的字面意思是活到老，学到老。对于一个训练好的神经网络模型来说，让这个网络学习新的任务的时候，如何避免遗忘之前的任务，这就是终生学习的内容。

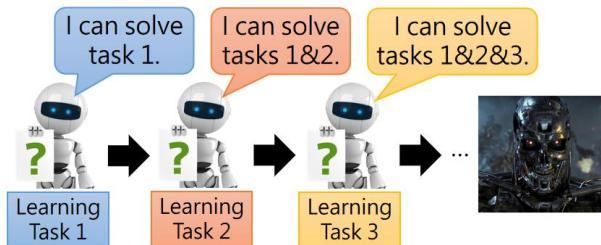


<https://world.edu/lifelong-learning-part-time-undergraduate-provision-crisis/>

16.1 终生学习基本介绍

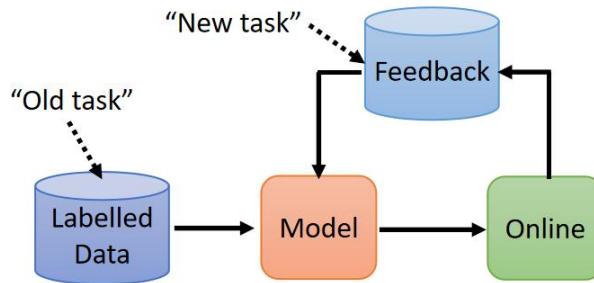
16.1.1 什么是终生学习？

终生学习的英文叫 Lifelong Learning，也有人将它翻译成 Continuous Learning, Increment Learning, Never Ending Learning。通常机器学习中，单个模型只解决单个或少数几个任务。对于新的任务，我们一般重新训练新的模型。而在终生学习中，我们可以在多个任务上使用同一个模型。



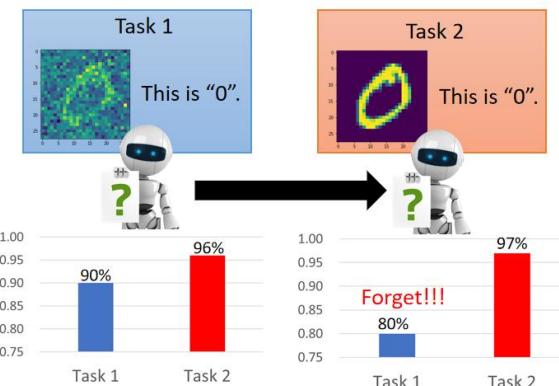
如上图所示，我们先训练机器做任务一比如语音识别，然后再教它做任务二比如图像识别，接着再教它做第三个任务比如翻译，这样一来它就一起学会了这三个任务。我们不断去教会它学习新的技能，这样一来等它学会成百上千个技能之后呢，它就会变得越来越厉害，最终可以到达人类无法企及的程度。

那有读者可能会疑惑，机器真的能达到这样远大的目标吗？如果目标难以实现，那终生学习的意义何在呢？事实上，以自监督学习为例，假设我们首先通过收集一些数据然后训练得到模型，模型上线之后它会收到来自使用者的反馈并且得到新的训练数据。这个时候我们希望能够形成一个循环，即模型上线之后我们得到新的数据，然后将新的数据用于更新我们的模型，模型更新之后又可以收到新的反馈和数据，对应地再次更新我们的模型，如此循环往复下去，最终我们的模型会越来越厉害。我们可以把过去的任务看成是旧的数据，把新的数据即来自于反馈的数据，这种情景也可以看作是终生学习的问题。



16.1.2 终生学习的难点

从上面的流程中，我们可能会认为终生学习是比较容易实现的，即只需要不断更新数据及其对应网络的参数即可。但事实上并没有那么容易。举一个简单的例子，如下图所示，这里有两个手写数字识别的任务，第一个任务的训练资料包含很多噪声，第二个没有噪声。我们训练完第一个任务的样本，第一个任务的测试准确率为 90%，在不看第二个任务样本的情况下，准确率已经可以达到 96%。将上述训练后的网络再经过第二个任务，第二个任务准确率达到 97%，但第一个任务准确率突然降低了，也就是加入任务二的训练资料之后，反而将第一个任务遗忘了。



但实际上我们把任务一和任务二的数据放在一起让这个网络同时去学的时候，会发现机器是能够同时学好这两个任务的。

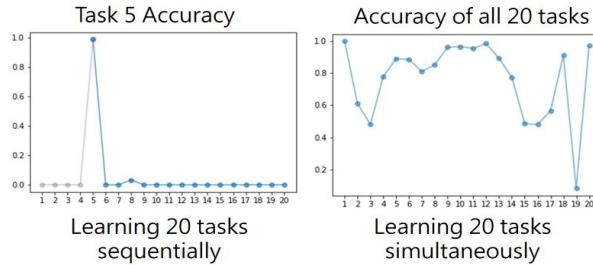


我们接下来再举一个自然语言处理方面的例子，假如我们在做一个 QA 的任务，使用的数据集是一个比较简单的 bAbi 数据集，里面包括 20 个 QA 任务，现在我们依次根据 20 个任务训练一个模型，看看最后的模型能不能把 20 个任务都学会。

Task 5: Three Argument Relations	Task 15: Basic Deduction
Mary gave the cake to Fred. Fred gave the cake to Bill. Jeff was given the milk by Bill. Who gave the cake to Fred? A: Mary Who did Fred give the cake to? A: Bill	Sheep are afraid of wolves. Cats are afraid of dogs. Mice are afraid of cats. Gertrude is a sheep. What is Gertrude afraid of? A:wolves

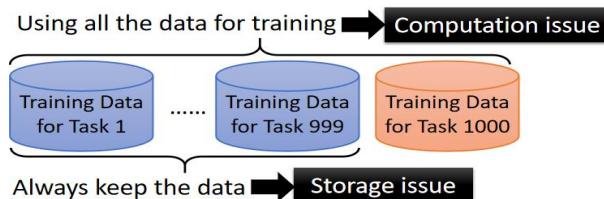
我们给出了对比训练的结果如下，下图左代表依次训练 20 个任务后，任务 5 的准确率；

下图右是同时训练 20 个任务，每个任务对应的准确率。对比两个图我们发现，机器其实是有能力学习 20 个任务的，只是这种将一个个任务依次训练的方法并不能很好地解决问题。



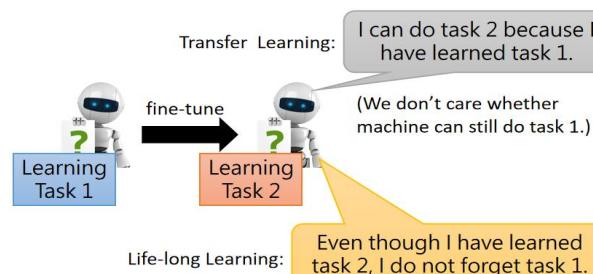
事实上，当模型依次学习多个任务的时候，它的学习和人类是很像的，经常会“学了后面忘了前面”，这种情况我们称之为灾难性遗忘（Catastrophic Forgetting），而与人类的“遗忘”相比，机器的这种遗忘的“灾难性”体现在模型的遗忘程度是很严重的。这就是终生学习的第一个难点。

也许读者会有这样的一个问题，刚才我们的例子有提到模型是能够同时学多个任务的，这种方式我们称作多任务学习（Multi-task Learning），既然有多任务学习的方法，那为什么还需要终生学习呢？事实上，当我们的任务不再仅仅是 20 个，而是成百上千个任务时，我们在做多任务学习时，需要将之前的任务数据放在一起进行训练，这样需要的时间是比较久的。就好比假设一个人要学一门新的课程，那他就必须要把之前上过的所有前置课程都学过一遍才有办法学习新的任务，这样其实是比较低效的学习方式。而且随着要学习的任务越来越多，所需要的时间也会越来越长。但如果我们能够解决终生学习的技术，那么其实就能够高效地学习多种新任务了。一般来说，由于需要学习越来越多的任务，模型参数需要一定的扩张。但我们希望模型参数扩张是有效率的，而不是来一个任务就扩张很多参数。这会导致计算和存储问题，这就是终生学习的第二个难点——模型扩张。



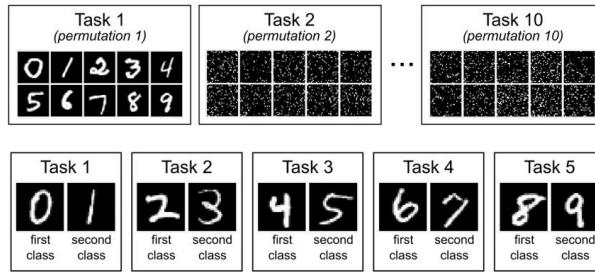
当然这种多任务学习也并非没有意义，我们通常把多任务学习的效果当成终生学习的上确界（Upper Bound），即我们在做终生学习的时候往往会先跑一个多任务学习的结果，来帮助我们参考终生学习的上线在哪里。

那有的人可能会想到一种让机器“触类旁通”的方法，也就是我们前文所介绍的迁移学习。但终生学习所能做到的事情比迁移学习更多。举例来说，假设有两个任务，迁移学习所做的事情是将任务 1 中的参数很好地运用到任务 2 中，它只关注了任务 2 的好坏；而终生学习希望做到的是两个任务都可以处理得很好，这就是终生学习的第三个难点。



16.2 终生学习的评估标准

在了解终生学习如何做得更好之前，我们先来看看怎么评判终生学习技术的一些标准。在做终生学习之前，需要有一系列任务让模型去学习，其实通常都是比较简单的任务，如下图所示，任务 1 就是常规的手写数字识别，任务 2 是把每一个数字用某一种特定的规则打乱的一种手写数字识别（这种称为 permutation），当然还有一些其它的处理方式，比如将数字进行旋转等。



<https://arxiv.org/pdf/1904.07734.pdf>

接下来对于具体的评估方式，我们可以按以下方式建表，横轴表示学习完某个任务后，在其他任务上的表现。纵轴表示某个任务在其他任务学习完后的表现。当 $i > j$ 时，说明学习完任务 i 之后，任务 j 没有被忘记，代表模型的记忆能力；反之当 $i < j$ 时，说明在学习完任务 i 后将模型迁移到还没有学习的任务 j 上，代表模型的迁移能力。

		Test on			
		Task 1	Task 2	Task T
Rand Init.		$R_{0,1}$	$R_{0,2}$		$R_{0,T}$
After Training	Task 1	$R_{1,1}$	$R_{1,2}$		$R_{1,T}$
	Task 2	$R_{2,1}$	$R_{2,2}$		$R_{2,T}$
	:				
	Task T-1	$R_{T-1,1}$	$R_{T-1,2}$		$R_{T-1,T}$
	Task T	$R_{T,1}$	$R_{T,2}$		$R_{T,T}$

评估主要包括三个指标：模型准确率，记忆能力和迁移能力。

16.2.1 模型准确率

模型准确率的计算方法是：当学习完任务列表中的最后一个任务 T 后，将模型在任务 1 到任务 T 上的准确率求和取平均，在上表中，代表最后 1 行的求和平均，对于第 i 个任务，其准确率公式计算如下：

$$\text{Accuracy}_i = \frac{1}{T} \sum_{j=1}^T R_{i,j}$$

16.2.2 记忆能力

模型的记忆能力衡量的是模型学习完最后一个任务 T 后，在之前任务上的表现。一般来说，模型会忘记之前的任务，所以这个指标一般为负数，一般我们将记忆能力也称为反向迁移（Backward Transfer），在计算第 i 个任务时，我们用学完最后一个任务 T 时在任务 i 上的

准确率 $R_{T,i}$ 和学完任务 i 时的准确率 $R_{i,i}$ 做一个差值，计算方法如下：

$$\text{Backward Transfer} = \frac{1}{T-1} \sum_{i=1}^{T-1} (R_{T,i} - R_{i,i})$$

		Test on			
		Task 1	Task 2	Task T
Rand Init.		$R_{0,1}$	$R_{0,2}$		$R_{0,T}$
After Training	Task 1	$R_{1,1}$	$R_{1,2}$		$R_{1,T}$
	Task 2	$R_{2,1}$	$R_{2,2}$		$R_{2,T}$
	:				
	Task T-1	$R_{T-1,1}$	$R_{T-1,2}$		$R_{T-1,T}$
	Task T	$R_{T,1}$	$R_{T,2}$		$R_{T,T}$

16.2.3 迁移能力

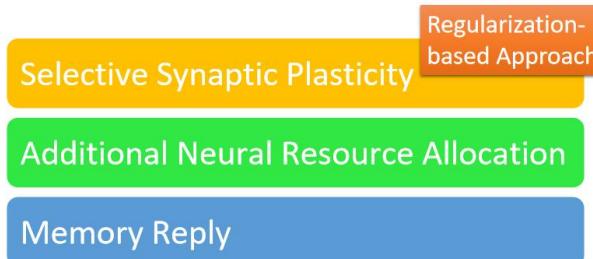
迁移能力衡量了模型在学习完一个任务后，在未学习的任务上的表现。一般和随机初始化的模型进行对比。迁移能力也称为前向迁移（Forward Transfer），计算公式如下：

$$\text{Forward Transfer} = \frac{1}{T-1} \sum_{i=2}^{T-1} (R_{i-1,i} - R_{0,i})$$

		Test on			
		Task 1	Task 2	Task T
Rand Init.		$R_{0,1}$	$R_{0,2}$		$R_{0,T}$
After Training	Task 1	$R_{1,1}$	$R_{1,2}$		$R_{1,T}$
	Task 2	$R_{2,1}$	$R_{2,2}$		$R_{2,T}$
	:				
	Task T-1	$R_{T-1,1}$	$R_{T-1,2}$		$R_{T-1,T}$
	Task T	$R_{T,1}$	$R_{T,2}$		$R_{T,T}$

16.3 终生学习的研究方向

在本章最后，我们来对终生学习的研究方向进行介绍。在终生学习中，最常见的研究方向有如下三种：

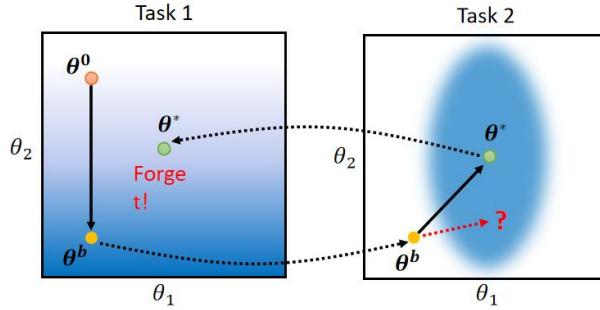


目前最成熟的解决方案为：选择性的突触可塑性（Selective Synaptic Plasticity），也称

为基于正则化的方法（Regularization based Approach），因此我们主要对这部分内容进行详细介绍。

16.3.1 选择性的突触可塑性

在介绍这种方法之前，我们首先来了解一下为什么机器会发生灾难性遗忘。假如现在有两个不同的任务：任务 1 和任务 2，为了简化考虑，我们假设训练的模型的参数只有两个，分别是 θ_1 和 θ_2 ，下图分别代表模型在不同任务上的损失函数，如果颜色越偏蓝色，代表损失越小，反之越大。



我们先让模型训练任务 1，做法是给一个随机化的初始参数 θ^0 ，用梯度下降的方法更新足够多次数的参数后得到 θ^b ；接下来训练任务 2，我们先将任务 1 中得到的 θ^b 迁移到任务 2 上，由于任务 2 的损失表面（Error Surface）和任务 1 是不同的，因此可能在任务 2 上还需要继续进行梯度下降，得到一个在任务 2 上表现更好的参数 θ^* 。然后我们将 θ^* 再次拿回到任务 1 上进行训练时，发现没有办法得到好的结果，这就是灾难性遗忘的产生原因。

那怎么解决这样的问题呢？其实对于一个任务而言，要实现较低的损失是可能有很多种不同的参数组合的，比如在任务 2 中可能所有椭圆内的参数都能有好的表现，而对于任务 1 则是偏下方的位置都能实现较低的损失。那么如果在训练完任务 1 之后训练任务 2 时，我们的参数不是像右上角移动形成 θ^* ，而是只往右移动，让最终得到的参数同时处于任务 1 和任务 2 较低的损失区域，这种情况下是有可能不会产生灾难性遗忘的，具体做法如下：

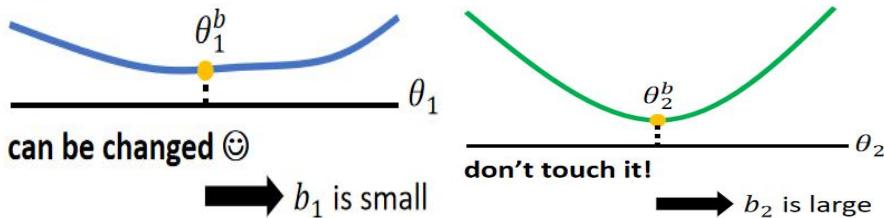
假设任务 1 学习到的参数是 θ^b ，在选择性的突触可塑性这一解法中会给每一个参数 θ_i^b 赋予一个参数 b_i ，这个系数代表着对应的参数对过去的任务到底重不重要，也称作“守卫（guard）”，因此在更新参数时，我们需要将损失函数改写成如下形式：

$$L'(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda \cdot \sum_i b_i (\theta_i - \theta_i^b)^2$$

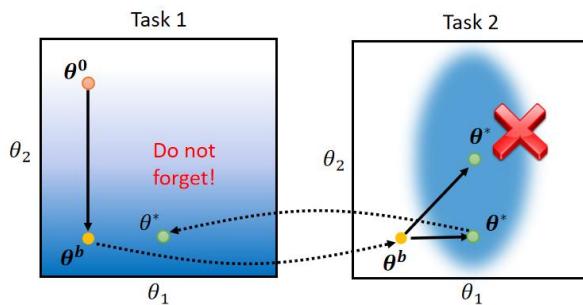
16.3.2 权重设置的解决

权重 b_i 的设置是人为决定的。其实有一种简单的控制变量的方法，就是移动或改变某个参数。

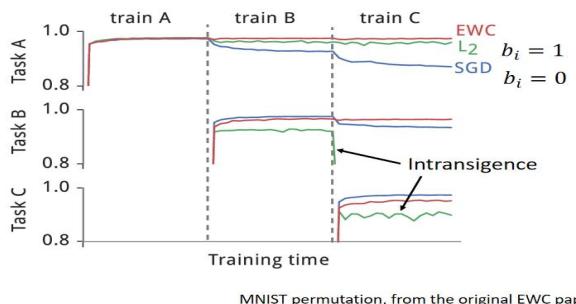
对于这个问题 EWC 给出了它的解决方案。它利用二阶导数来确定 b_i ，如果二阶导数比较小，参数位于一个平坦的盆地中，此时参数变化对任务的影响小， b_i 就比较小。反之，如果二阶导数比较大，参数位于一个狭小的盆地中，此时参数变化对任务的影响大， b_i 就比较大。



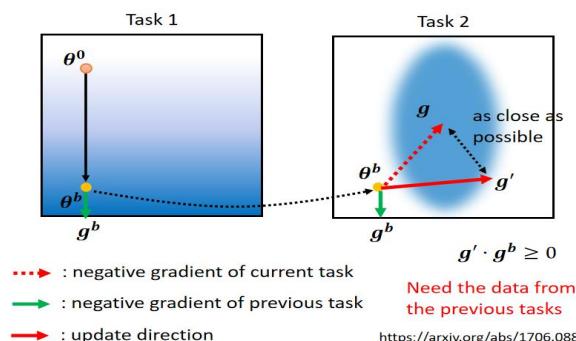
利用 EWC 后的效果如下图所示。假如模型有两个参数 θ_1 和 θ_2 ，如果我们沿水平方向更新参数，也就是 θ_1 的方向，则任务 1 上的模型效果依然是可观的。这表明，通过 EWC 是可以做到让模型记住以前学到的知识的。



接下来我们给出 EWC 原始文献中的一张图，在这篇论文中，我们分别训练三个任务：Task A, Task B, Task C，图中有三条线，蓝色的线代表一般的随机梯度下降的情况，即 $b_i = 0$ ，绿色的线是做了 L2 正则化的情形，即 b_i 始终为 1，这就会产生一种称为不妥协 (Intransigence) 的现象，而红色的线是 EWC 的结果，我们发现 EWC 在每一个任务中的正确率都是非常高的。



其实在基于正则的方法出现之前，还有一类方法，叫做 GEM (Gradient Episodic Memory)，它不是在参数上做限制，而是在梯度更新的方向上做限制，因此又被称为基于梯度的方法。原理如下图所示，它在计算当前任务梯度方向的同时，也会回去算历史任务对应的梯度方向，然后把两个梯度进行矢量求和，得出实际的梯度方向，这样持续更新就能尽可能接近一个不会陷入灾难性遗忘的最优解了。当然新的梯度方向需要是非负的，否则很难朝最优解方向优化。

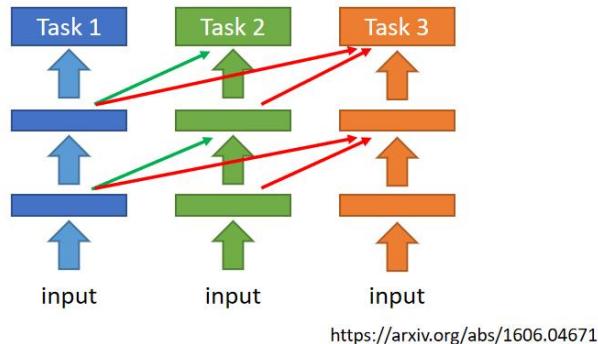


但是这类方法需要把过去的数据同时存储下来，其实这和终生学习的初衷有些违背了，因为终生学习本身就是希望不需要依赖过去的数据。但由于这里只是存储梯度这些少量的信息，像前面讲的选择性的突触可塑性方法也需要存储一些历史模型信息，所以在实际操作过程中也还能接受。

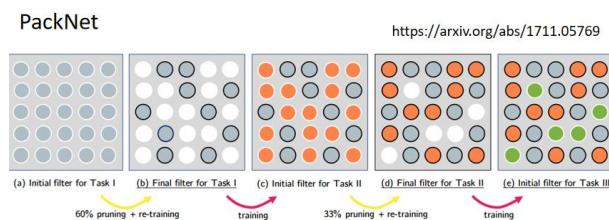
16.3.3 其它方法

我们在本节开头就说过，还有一些其它的方法解决终生学习问题。我们还可以用一种叫额外的神经资源分配（Additional Neural Resource Allocation）的方法。其思想是将之前学好的网络参数固定，对于新的任务，增加部分网络结构，以让新的网络能够同时在旧任务和新任务上表现得好。

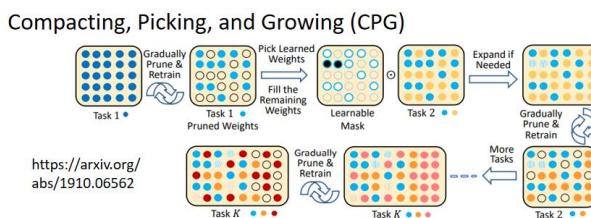
一个典型的做法是渐进式神经网络（Progressive Neural Network），以下图为例，在训练任务2时，将任务1网络的隐含层输出作为输入给下一个网络；训练任务3时，将任务1和2网络的隐含层输出均作为输入给下一个网络。随着任务增多，网络规模越来越大，但是避免任务1网络参数改变，同时还利用了之前任务的一些特征。



另一种想法是与渐进式神经网络相反，比如 PackNet，它是说一开始先设置一个大网络，然后对于每一个任务都用大网络中的一部分去解决，同时保证在旧任务和新任务上表现得好。



PackNet 和渐进式网络可以结合在一起使用，得到一种叫 CPG 的技术：



对于终生学习这项技术，还有一些其它的做法，感兴趣的读者可以自行探究，但其实最有效的方法就是 EWC 方法，对于大部分读者，只需要对 EWC 这个基于正则化的方法进行了解即可。

17 元学习 (Meta Learning)

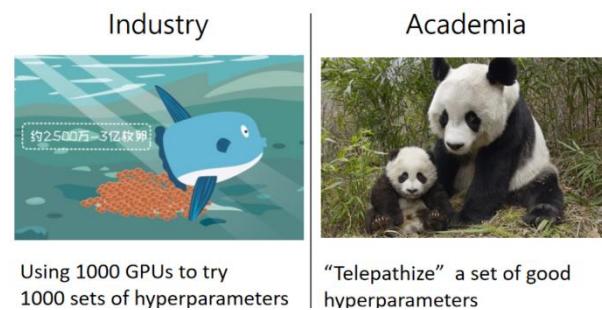
17.1 元学习的概念

17.1.1 元 (Meta) 的含义

我们在这一章节开始介绍元学习 (Meta Learning)。那什么是元学习呢？从字面意思来看，人们一般将 Meta 翻译成“元”的意思。事实上，当我们说一个东西是“Meta x”的时候，代表说这个东西是 x 的 x。所以元学习从字面的意思来看就是“学习”的“学习”，也就是学习如何学习。



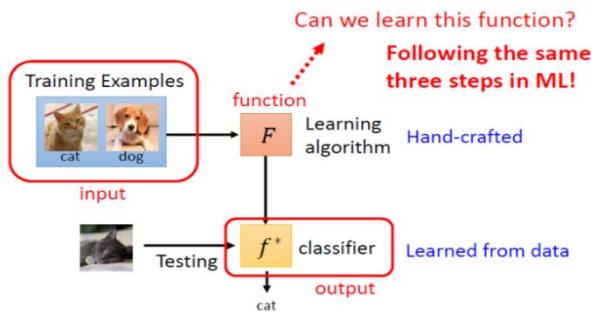
回忆我们之前做的绝大部分深度学习的任务，基本上都是通过调整超参数，或者是决定网络架构等方法从而获得更高的准确率。一直以来，我们希望找到更好地解决办法，通常在工业界的解决办法是十分“暴力”的——即用一大堆 GPU 来尝试各种不同的超参，模型等组合从而获得最优解。但这种“暴力解法”是存在一些问题的，首先，在学界，通常没有那么多的 GPU，其次，当你开了许多张 GPU 训练时，会耗费大量的时间。



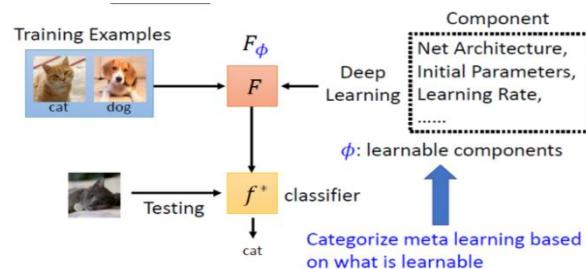
那么有没有更好的方法来解决这个问题呢？事实上，人们想到了一个办法，就是让机器“自己”使用“learning”的方法来调整参数。换言之，我们希望有一个方法可以让机器自己去调整这些超参数，学习一个最优的模型和网络架构，然后得到好的结果。这就是元学习其中一个可以帮助我们的事情。

17.1.2 元学习的步骤

在介绍元学习之前，我们在本书最后一次回忆一下机器学习，机器学习分为三个步骤，分别是：1. 找一个函数 2. 定义损失函数 3. 优化。与机器学习类似的是，元学习同样包含这三个步骤，但它与机器学习略微有一些区别。



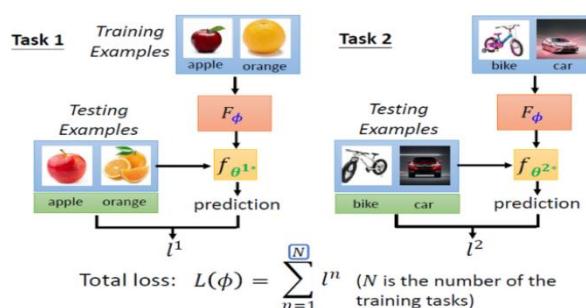
元学习的第一步同样是找寻一个函数，但与机器学习有所不同的是，元学习中寻找的函数 F ，其实是学习算法（Learning Algorithm）这个东西。换言之，要寻找的函数 F 的输入不是一张图片，而是一个数据集，它的输出是训练完的结果。以训练一个分类器为例，整个元学习的框架如下：我们需要找到一个函数 F ，使得分类器的损失函数 f 越小越好。那在学习算法中，我们通常会考虑要让机器自己决定网络的架构，让机器自己决定初始化的参数，让机器自己决定学习率等等。通常，我们将这些需要机器自己学习的东西统称为 ϕ ，那学习算法所代表的函数，一般我们就用 F_ϕ 进行表示。



第二步是要设置一个损失函数，在元学习中，我们的损失函数代表着我们的学习算法的好坏，通常我们将其记为 $L(\phi)$ （其值越小，代表我们的学习算法越好）。

- Define loss function for learning algorithm F_ϕ
- $$L(\phi) \downarrow \text{blue thumbs up} \quad L(\phi) \uparrow \text{red thumbs down}$$

那么我们该如何决定这个损失函数呢？假设现在有 n 个任务，对于第 i 个任务是分辨苹果和橘子，我们通过元学习的第一步得到了 F_ϕ ，从而得到了一个模型，记为 $f_{\phi^{i*}}$ ，接下来我们需要在测试集上对这个模型进行评估，并与正确结果进行比较，得到 l^i 。而 $L(\phi)$ 就是所有任务的 l^i 的求和。



在得到元学习的损失函数之后，元学习的第三步同样是做一个优化问题，使得 $L(\phi)$ 尽可能小。求解这个问题的方法有很多，我们当然也可以用机器学习中的梯度下降来做。事实上，

这一做法的前提是 $\frac{\partial L(\phi)}{\partial \phi}$ 是存在的，但我们的 ϕ 有时候是网络架构，或者是一些其他的不可导的东西，这个时候，我们只能采用强化学习或者是一些启发式算法（Evolutionary Algorithm）进行求解了。

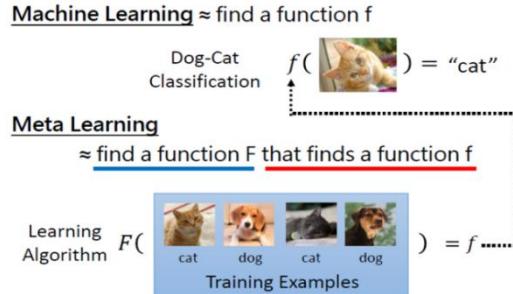
综上所述，我们通过以上三步找到了一个学习算法，这就是元学习的三部曲。需要注意的是，在元学习中，我们真正关心的是测试任务里面的测试资料，而训练任务是与测试任务无关的，训练任务的目标只是为了找到学习算法。

这个框架的厉害之处在于，你可能会觉得它与少样本学习（few-shot learning）差不多，因为我们同样只需要一小部分的样本。事实上，少样本学习本质上是期待机器达成“只用一点点资料”就能学会如何训练的目标，而元学习是“学习如何学习”这一件事，而我们想要达到小样本学习中的算法通常就是用元学习得到的学习算法。

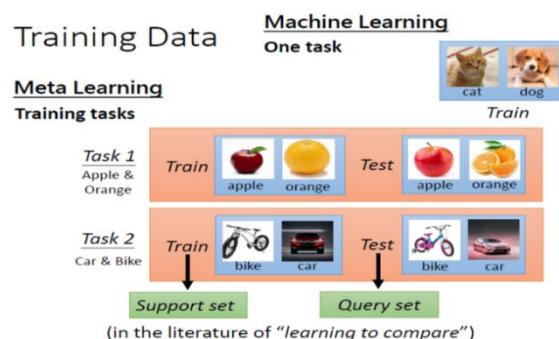
17.1.3 元学习与机器学习的比较

上一小节我们了解到，元学习与机器学习同样都可以被分为三步，而且这三步是非常相近的。那在本节，我们来简单地比较一下二者的异同。

首先，我们来比较一下机器学习和元学习的目标。在这里或许用英文能更好地理解二者的区别：Machine Learning \approx find a function f , Meta Learning \approx find a function F that finds a function f 。

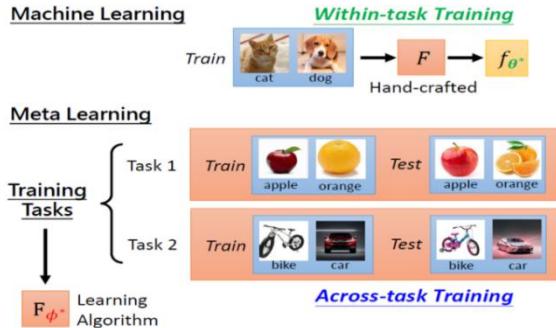


其次，我们来比较一下二者的训练资料，对于机器学习，我们只需要拿一笔数据集进行训练即可，这本质上来说是一个任务，而在元学习中，我们是拿训练任务（Training Task）作为训练资料进行训练的。这听起来可能有些拗口。因此，在学界中，一般将任务中的训练资料称为 Support Set，测试资料称为 Query Set。在每个训练任务中，均包含 Support Set 和 Query Set。

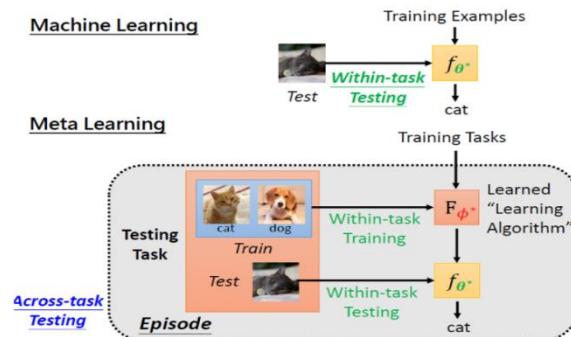


从上图可以看出，机器学习只有一个任务，且学习算法是 Hand-crafted 的，因此我们将

机器学习称为一个 Within-task Training，而元学习是多任务的，它的学习算法是在多个任务上训练得到的，我们将其称为 Across-task Training。



而在测试中，机器学习同样是直接使用训练得到的模型在任务中对测试数据进行测试，因此它的测试也是 Within-task Testing。而对于元学习的测试，它也是 Across-task Testing 的，它分为 2 步，第一步是在“测试任务”中，将任务的训练数据丢给“学习算法”，训练得到该任务的模型，这是 Within-task Training，而第二步需要将该任务的测试数据丢进模型，这是 Within-task Testing，如下图所示：Across-task Testing = Within-task Training + Within-task Testing。



二者在损失函数同样有区别，这其实也是因为机器学习是一个任务，而元学习是多个任务，公式对比如下：

Machine Learning $L(\theta) = \sum_{k=1}^K e_k$ <p style="text-align: right;">Sum over training examples in one task</p>	Meta Learning $L(\phi) = \sum_{n=1}^N l^n$ <p style="text-align: right;">Sum over testing examples in one task</p>
--	--

Episode

Sum over training tasks

当然，机器学习和元学习也有一些相似之处，事实上很多在机器学习那边学到的知识和基本概念都可以直接搬到元学习来，为了方便对比，列出相似之处的表格如下：

Problem & Solution	Machine Learning	Meta Learning
Overfitting	Collect more data	Collect more tasks
Augmentation	Data Augmentation	Task Augmentation
Hyperparameters	Model	Learning Algorithm

17.2 元学习的实例算法

前面我们已经讲完了元学习的基本概念，接下来我们就要讲一些元学习的实例算法。在这里我们会介绍两个算法，一个是 MAML，另一个是 Reptile。这两个算法都是在 2017 年提出来的，而且都是基于梯度下降的。

- Model-Agnostic Meta-Learning (MAML)

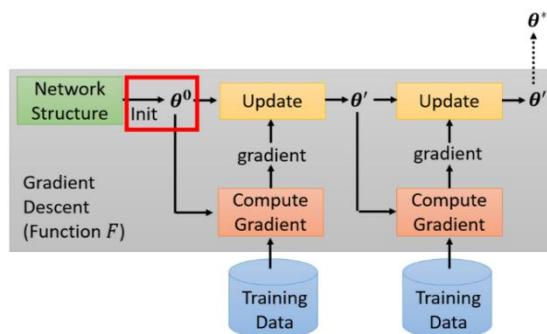


- Reptile

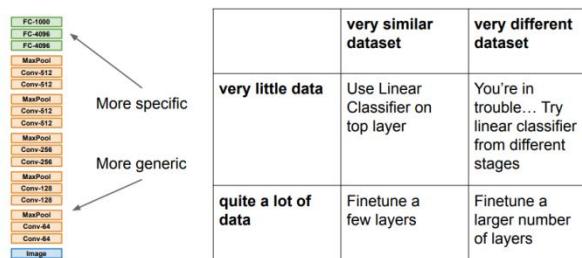
<https://arxiv.org/abs/1803.02999>

17.2.1 从迁移学习 (Transfer Learning) 到 MAML

我们知道，训练神经网络的第一步是初始化参数，即我们需要知道在这整个过程中哪些东西是可以训练的。首先初始化的参数 θ^0 是可以被训练的，而且一般它是随机初始化的。同时 θ^0 对结果往往有一定程度的影响，初始化参数的好坏所引起的结果有着天壤之别。那么我们能不能够透过一些训练的任务，来找出一个对训练特别有帮助的初始化参数呢？这个方法是 MAML，也就是 Model-Agnostic Meta-Learning。

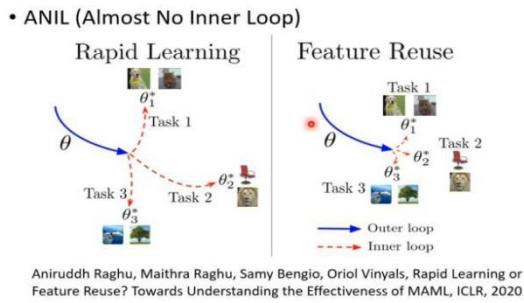


事实上，当前大多数深度学习框架都收录了不同的参数初始化方法，例如均匀分布、正太分布，或者用 Xavier_uniform, Kaiming_uniform, Xavier_normal, Kaiming_normal 等算法。除了用随机数，也可以用预训练的网络参数来初始化神经网络，也就是所谓迁移学习，或者更准确地说是微调的技术。



迁移学习（微调）的核心思想是：预训练的网络比随机初始化的网络有更强的学习能力，因此微调也算是一种元学习的算法。它和我们今天要介绍的 MAML 以及 Reptile 都是通过初始化网络参数，使神经网络获得更强的学习能力，从而在少量数据上训练后就能有很好的性能。

其实，MAML 和迁移学习有着异曲同工之妙，但其实 MAML 是更有优势的。我们首先提出两个假设：第一个假设是 MAML 找到的初始参数是一个很厉害的初始参数。它可以让我们的梯度下降这种学习算法快速找到每一个任务的参数。另外一个假设是这个初始化的参数它本来就和每一个任务上最终好的结果已经非常接近了，所以我们直接应用很少几次的梯度下降就可以轻易的找到好的结果。其中第二个是使得 MAML 有效的关键，它来源于 ICLR2020 的文章“Rapid Learning or Feature Reuse? Towards Understanding the Effectiveness of MAML”。



17.2.2 MAML 与预训练的区别

MAML 的思想是找一个足够好的初始化参数 ϕ ，这点与预训练（Pre-training）是类似的，但我们需要区分 MAML 和 Pre-training，而区分的标准在于，评判好的初始化参数的标准。

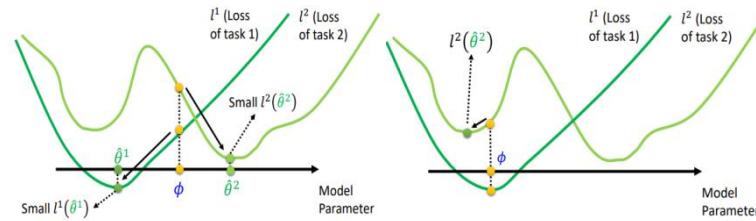
举一个现实的例子，MAML 就好比是这样一批人，他们认为虽然本科毕业读研究生需要三年时间，但研究生毕业之后的工资上限会更高，而 Pre-training 可以类比成另外一批人，他们认为本科毕业直接就业虽然上限没有研究生学历高，但三年工作经验+下限仍然能保证自己以后过得很好。

MAML	$\hat{\theta}^n$: model learned from task n
Loss Function:	$\hat{\theta}^n$ depends on ϕ
$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$	$l^n(\hat{\theta}^n)$: loss of task n on the testing set of task n
How to minimize $L(\phi)$? Gradient Descent	$\phi \leftarrow \phi - \eta \nabla_\phi L(\phi)$
Find ϕ achieving good performance after training	潜力

事实上，我们从损失函数的优化求解就能看出二者的区别，在 MAML 中，不同分任务对应一套不同的参数，它们都是基于网络参数 ϕ 通过梯度下降得到的，因为 ϕ 最终我们需要的初始化参数，它不能受制于某个分任务，而是要掌控全场。

Model Pre-training	Loss Function:
Widely used in transfer learning	$L(\phi) = \sum_{n=1}^N l^n(\phi)$
Find ϕ achieving good performance	現在表現如何

而对于普通的学习任务，我们通常会比较在意能否得到全局最优解。但在当前所有训练任务上的全局最优解，一定是好的初始化值么？反例如下：在下图中，横坐标代表网络参数，纵坐标代表损失函数。浅绿和墨绿两条曲线代表两个任务的损失函数随参数变化曲线。我们发现如果用模型预训练，可以帮助我们找到可以使两个任务损失函数之和最小的参数，但如果使用 MAML 时， \emptyset 会掌控全场，它不太在意训练集上的损失，而是着眼于最大化网络的学习能力。



也就是说，用这个全局最优解初始化网络参数，再在新任务上做微调，最终得到的模型可能不会收敛到新任务的最优解。这就是 MAML 优于微调的地方。

17.2.3 元学习算法的评价标准

既然元学习的目标是“学习如何学习”，那么如何证明元学习算法的有效性呢？显而易见，只需要证明用这种算法得到的网络模型学习能力很强就行了。具体到我们的 MAML 和 Reptile 只需要证明，用它们这些算法初始化之后的神经网络，在新的任务上训练，其收敛速率与准确率比从随机初始化的神经网络要高。

这里所谓“新任务”，一般是指难度比较大的任务，因此一般用少样本学习（few-shot learning）的任务来评估元学习算法的有效性。

- **N-ways K-shot** classification: In each training and test tasks, there are **N classes**, each has **K examples**.

<u>20 ways</u>	<u>1 shot</u>	<table border="1"> <tr><td>ମ</td><td>ପ</td><td>ଶ</td><td>ଦ</td><td>ର</td></tr> <tr><td>କ</td><td>ଟ</td><td>ସ୍ତ୍ରୀ</td><td>ବ୍ରାହ୍ମି</td><td></td></tr> <tr><td>ଫ</td><td>ଫ</td><td>ପ୍ରୀତି</td><td></td><td></td></tr> <tr><td>ୟ</td><td>ୟ</td><td>ଶୁଣୁ</td><td>ନୁହିଲା</td><td></td></tr> </table>	ମ	ପ	ଶ	ଦ	ର	କ	ଟ	ସ୍ତ୍ରୀ	ବ୍ରାହ୍ମି		ଫ	ଫ	ପ୍ରୀତି			ୟ	ୟ	ଶୁଣୁ	ନୁହିଲା			Testing set (Query set)
ମ	ପ	ଶ	ଦ	ର																				
କ	ଟ	ସ୍ତ୍ରୀ	ବ୍ରାହ୍ମି																					
ଫ	ଫ	ପ୍ରୀତି																						
ୟ	ୟ	ଶୁଣୁ	ନୁହିଲା																					
Each character represents a class		<table border="1"> <tr><td>ମ</td><td>ପ</td><td>ଶ</td><td>ଦ</td><td>ର</td></tr> <tr><td>କ</td><td>ଟ</td><td>ସ୍ତ୍ରୀ</td><td>ବ୍ରାହ୍ମି</td><td></td></tr> <tr><td>ଫ</td><td>ଫ</td><td>ପ୍ରୀତି</td><td></td><td></td></tr> <tr><td>ୟ</td><td>ୟ</td><td>ଶୁଣୁ</td><td>ନୁହିଲା</td><td></td></tr> </table>	ମ	ପ	ଶ	ଦ	ର	କ	ଟ	ସ୍ତ୍ରୀ	ବ୍ରାହ୍ମି		ଫ	ଫ	ପ୍ରୀତି			ୟ	ୟ	ଶୁଣୁ	ନୁହିଲା			Training set (Support set)
ମ	ପ	ଶ	ଦ	ର																				
କ	ଟ	ସ୍ତ୍ରୀ	ବ୍ରାହ୍ମି																					
ଫ	ଫ	ପ୍ରୀତି																						
ୟ	ୟ	ଶୁଣୁ	ନୁହିଲା																					

- Split your characters into training and testing characters
 - Sample N training characters, sample K examples from each sampled characters → one training task
 - Sample N testing characters, sample K examples from each sampled characters → one testing task

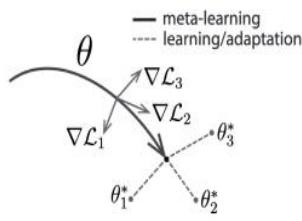
数据集 Omniglot: brendenlake/omniglot, 是一个类似 MNIST 的手写数据集, 如下图所示。该数据集包含 1623 类, 每类只有 20 个训练数据, 因此它属于少样本学习的范畴, 经常作为基准用来衡量元学习算法的性能。



这个数据集的用法如下：从其中采样 N 个类，每个类有 K 个训练两本，组成一个训练任务（task），称为 N -ways K -shot classification。然后再从剩下的类中，继续重复上一步的采样，构建第二个任务，最终构建了 m 个任务。把这 m 个任务分成训练任务和测试任务，在训练任务上训练元学习的算法，然后再用测试任务评估元学习得到的算法的学习能力。

17.2.4 MAML 算法和 Reptile 算法

上文中讲了那么多铺垫，终于可以正式开讲算法了。刚刚说了 MAML 关注的是，模型使用一份“适应性很强的”权重，它经过几次梯度下降就可以很好的适用于新的任务。那么我们训练的目标就变成了“如何找到这个权重”。而 MAML 作为其中一种实现方式，它先对一批中的每个任务都训练一遍，然后回到这个原始的位置，对这些任务的损失函数进行一个综合的判断，再选择一个适合所有任务的方向。



```
Algorithm 1 Model-Agnostic Meta-Learning
Require:  $p(\mathcal{T})$ : distribution over tasks
Require:  $\alpha, \beta$ : step size hyperparameters
1: randomly initialize  $\theta$ 
2: while not done do
3:   Sample batch of tasks  $\mathcal{T}_i \sim p(\mathcal{T})$ 
4:   for all  $\mathcal{T}_i$ , do
5:     Evaluate  $\nabla_{\theta} \mathcal{L}_{\mathcal{T}_i}(f_{\theta})$  with respect to  $K$  examples
6:     Compute adapted parameters with gradient descent:  $\theta_i^* = \theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{T}_i}(f_{\theta})$ 
7:   end for
8:   Update  $\theta \leftarrow \theta - \beta \nabla_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i^*})$ 
9: end while
```

在 MAML 原始的论文中，给出的算法伪代码如上图右所示，该算法实质上是 MAML 预训练阶段的算法，目的是得到模型 M_{Meta} ，该算法包括两个 Require，第一个 Require 指的是在 $D_{meta-train}$ 中任务的分布。结合我们在上一小节举的例子，这里即反复随机抽取任务： T ，形成一个由若干个（e.g., 1000 个） T 组成的 task 池，作为 MAML 的训练集。MAML 的目的，在于快速适应（fast adaptation），即通过对大量任务的学习，获得足够强的泛化能力，从而面对新的、从未见过的任务时，通过微调就可以快速拟合。不同任务之间只要存在一定的差异即可。再强调一下，MAML 的训练是基于任务的，而这里的每个任务就相当于普通深度学习模型训练过程中的一条训练数据。

第二个 Require 中，步长其实就是学习率，MAML 其实是一个 gradient by gradient 的算法，每次迭代包括两次参数更新的过程，所以有两个学习率可以调整。

接下来就是算法流程，第一步是随机初始化参数 θ ，而在第 2 步，算法进入了一个循环，可以理解为一轮迭代过程或一个 epoch。第三步可以理解成一个 Dataloader，即随机对若干个（e.g., 4 个）task 进行采样，形成一个 batch。而第 4 到 7 步是第一次梯度更新的过程，第 8 步是第二次梯度更新的过程。

我们接下来分析一下两次梯度更新的流程，第一次梯度更新其实是上述算法的第 5 步，它的做法是利用 batch 中的某一个任务中的 support set，计算每个参数的梯度。在 N -way K -shot 的设置下，这里的 support set 应该有 NK 个。作者在算法中写 with respect to K examples，默认对每一类下的 K 个样本做计算。实际上参与计算的总计有 NK 个样本。这里的损失函数计算方法，在回归问题中，就是 MSE；在分类问题中，就是交叉熵。第二次梯度更新是算法的第 8 步，这里的 loss 计算方法，大致与步骤 5 相同，但是不同点有两处。一处是我们不再是分别利用每个任务的 loss 更新梯度，而是像常见的模型训练过程一样，计算一个批次的 loss 总和，对梯度进行随机梯度下降。另一处是这里参与计算的样本，是任务中的 query set，在我们的例子中，即 $5\text{-way} * 15 = 75$ 个样本，目的是增强模型在任务上的泛化能力，避免过拟合。

support set。步骤 8 结束后，模型结束在该批次中的训练，开始回到步骤 3，继续采样下一个批次。

Algorithm : Reptile

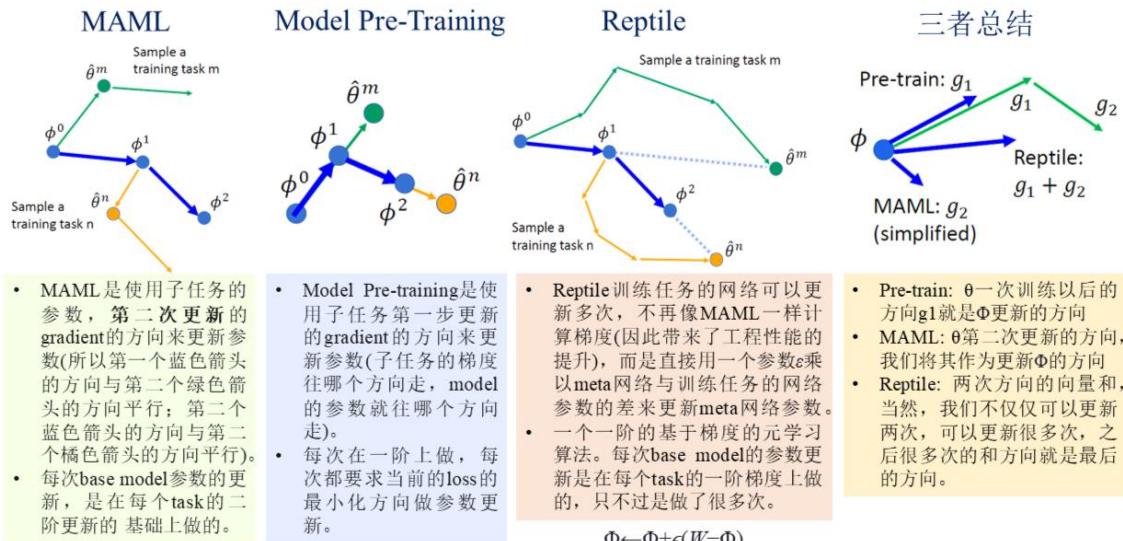
```

Initialize  $\theta$ , the vector of initial parameters
for iteration = 1, 2, ... do
    Sample task  $\tau$ 
    Compute  $\tilde{\theta} = U_{\tau}^k(\theta)$ , denoting  $k$  steps of SGD or Adam
    Update  $\theta = \theta + \epsilon(\tilde{\theta} - \theta)$ 
end for

```

Reptile 和 MAML 很像，它的算法如上图所示，Reptile 中，每更新一次 \emptyset ，需要采样一批任务（图中 batchsize=1），并在各个任务上施加多次梯度下降，得到各个任务对应的 $\hat{\theta}$ ，然后计算 $\hat{\theta}$ 与主任务的参数 \emptyset 的差向量，作为更新的方向。这样反复迭代，最终得到全局的初始化参数。

在本节最后，我们来简单对比一下 MAML，Pre-training 和 Reptile，同样，我们还是通过表格的形式进行说明，如下图所示：



到目前为止，我们看到的这些方法都是基于梯度下降再去做改进的，并且我们还是把训练跟测试分成两个阶段。事实上，现在也有人提出将训练和测试结合起来当作整个网络的输入，这种系列的方法叫做，learning to compare，它又叫做 metric-base 的方法，这一系列的做法就可以看作是训练和测试没有分界，一个网络直接把训练资料和测试资料都读进去，而直接输出测试资料的结果。

总之，元学习不是只能用非常简单的任务，今天在学界已经开始把元学习推向更复杂的任务，我们也一直希望未来元学习这个技术能够真的用在现实的应用上，可以走得越来越远。

18 网络压缩

我们在本书的最后介绍一个很有探索意义的研究方向——网络压缩（Network Compression）。随着大模型的发展，我们需要开始考虑模型的运算速度。网络压缩这项技术所考虑的问题是：能否将这些参数量比较大的模型进行简化，并且能和原模型的性能相差无几。举个例子，边缘设备（edge device）的内存和计算力有限，如果模型太过巨大，可能无法在这些设备上运行，因此需要比较小的模型。

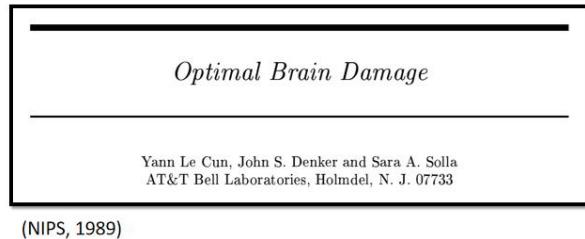


本章介绍五个以软件为导向的网络压缩技术，这些技术只是在软件上面对网络进行压缩，均不考虑硬件加速的部分。

18.1 网络剪枝

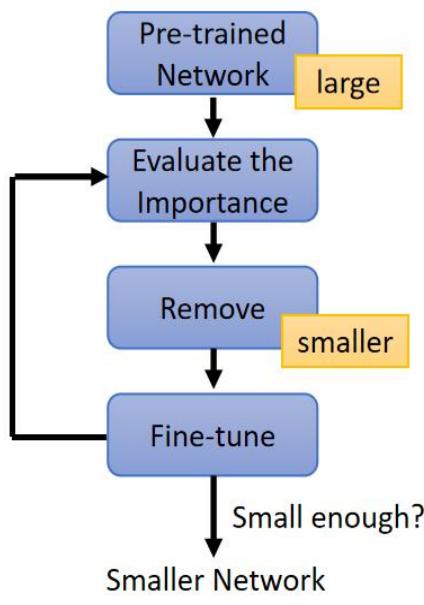
第一种技术叫网络剪枝（Network Pruning），网络剪枝顾名思义就是要把网络里面的一些参数剪掉。在神经网络中有很多参数，但其中有些参数对最终的输出结果贡献并不大，因此我们可以将这些没有太多贡献的参数去掉，这就是网络剪枝的技术。

事实上，网络剪枝不是太新的概念，早在上世纪 90 年代，Yann Le Cun 的论文“Optimal Brain Damage”实际上就是在讨论一种对“脑损伤”最小的剪枝方法。



18.1.1 网络剪枝的框架

网络剪枝的框架如下图所示，首先需要训练一个大的网络，然后评估出重要的权重或神经元，网络剪枝就是要对这些不太重要的权重或神经元进行移除，最后将处理后的模型进行微调，从而将移除的“损伤”拿回来。当然需要注意的是，移除的参数（权重/神经元）不能太多，否则很有可能无法修复。

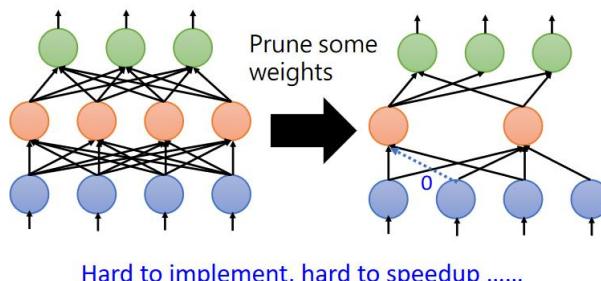


那么如何评估权重/神经元的重要性呢？对权重而言，我们可以通过计算其 L1, L2 值来判断重要程度；对于神经元而言，我们可以给出一定的数据集，然后查看在计算这些数据集的过程中神经元参数为 0 的次数，如果次数过多，则说明该神经元对数据的预测结果并没有起到什么作用，因此可以去除。

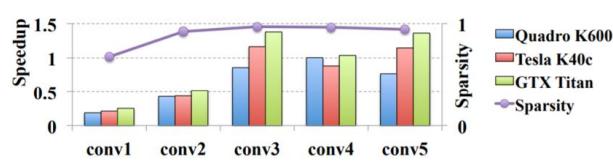
18.1.2 网络剪枝的两种形式

剪枝有两种形式，一种是以权重（weight）为单位，一种是以神经元（neuron）为单位，这两者有什么不同呢？

首先，以权重为单位的剪枝的做法是通过除去一些冗余的 weight，使网络结构变得简单一些，但这样就造成网络结构不规则（irregular），难以编程实现，同时难以用 GPU 加速。通常的做法是将冗余的 weight 置为 0，但这样做还是保留了参数（等于 0），不是真正去除掉。

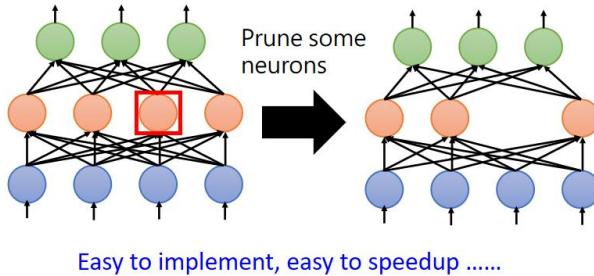


在这篇论文中有个关于参数剪枝多少与训练速度提升关系的实验证，其中紫色线表示参数去掉的量。可以发现，虽然参数去掉了将近 95%，但是速度依然没有提升。



<https://arxiv.org/pdf/1608.03665.pdf>

第二种方法是以神经元为单位的剪枝，如下图所示，删减神经元之后网络结构能够保持一定的规则，实现起来方便，而且也能起到一定的加速作用。



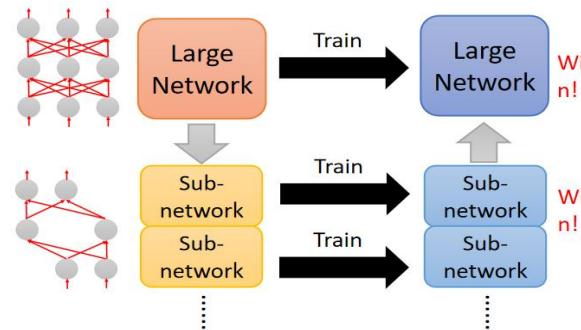
通过上述可知，实际上做权重剪枝是很麻烦的，通常来讲，神经元剪枝是更容易实现的，也更容易起到加速作用。

18.1.3 为什么需要剪枝

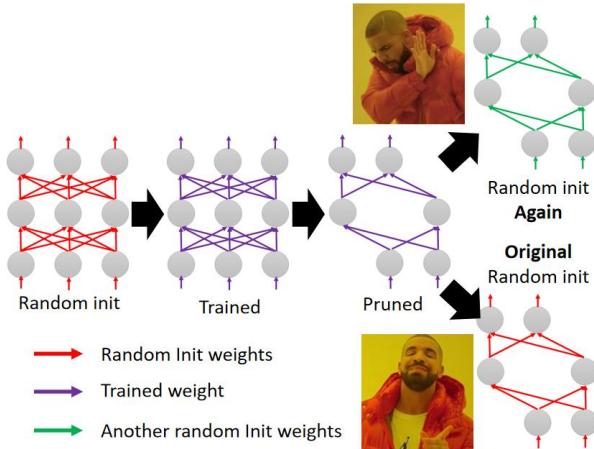
那么可能有读者会问，剪枝的做法是先训练一个大的网络，再将它变小，并且小的网络和大的网络准确率相差不是很大，那我们为什么不直接训练一个小的神经网络呢？

一个普遍的解释是大的网络比较好训练。如果直接训练一个小的网络，往往没有办法得到跟大的网络一样的正确率。因为模型越大，越容易在数据集上找到一个局部最优解，而小模型比较难训练，有时甚至无法收敛。（即只要网络结构够大，梯度下降就可以找到全局最小区点）

另外一个解释是基于 2018 年的一个发表的大乐透假设(Lottery Ticket Hypothesis)，它指的是每次训练网络的结果不一定会一样。我们抽到一组好的初始的参数，就会得到好的结果；抽到一组坏初始的参数，就会得到坏的结果。这就和彩票抽奖一样，但随着我们增加“抽奖”次数，中奖的概率也会大大地增加。那么对于一个大的神经网络而言，我们可以将它看成很多小的子网络（sub-network）的组合，我们并不一定需要每个子网络都能训练成功，但是随着子网络个数的增多，我们只需要找到其中一个成功的子网络就可以保证整个网络训练成功。

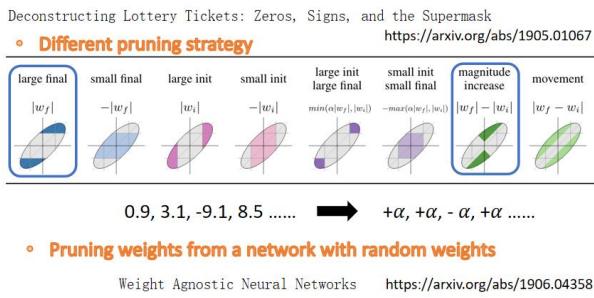


大乐透假设其实是在一个实验上证实的。如下图所示，我们首先初始化网络结构，随机初始化参数并对网络进行训练，将训练完的网络做剪枝。如果这时候我们还是采用这个压缩的网络结构但是随机初始化参数（绿色部分），发现网络训练不起来。但是如果我们将这个压缩的网络结构，但是参数是从大网络对应参数位置拿过来训练（红色部分），就能训练起来。说明留下来的这组参数组成的网络是可以训练起来的子网络。



关于大乐透假说，研究人士进行了一些后续研究，通过充分的实验得到了一些结论：

1. 如果训练前和训练后参数的差距越大，将其进行剪枝后得到的结果越有效。
2. 小的子网络只要我们不改变参数的正负号（所有 > 0 的参数用 $+ \alpha$ 表示，所有 < 0 的参数用 $-\alpha$ 表示），就可以训练起来。
3. 对于一个初始的大网络（参数随机初始化），有可能不训练就已经有一个子网络可以有一个比较好的效果。



此外还有一个很有意思的事情，即大乐透假说不一定是正确的，因为很快就有人“打脸”了这一说法：如下图所示，我们用剪枝完的小网络随机初始化参数再训练，只要多训练几轮，就可以比不随机初始化训练小网络的效果要好。

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

<https://arxiv.org/abs/1810.05270>

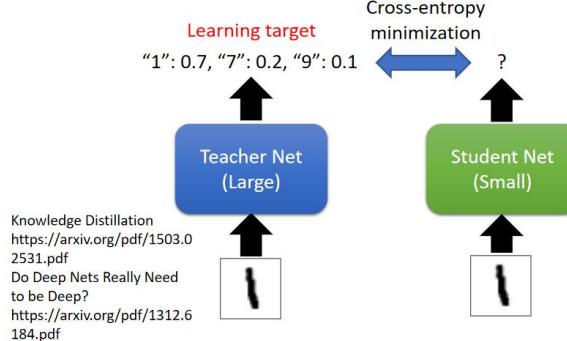
当然这篇文章的作者也给出了一些对大乐透假说的回应，大乐透假说出现的前提是学习率需要设置得很小，或者在做权重剪枝的时候才有可能出现大乐透假说现象。

18.2 其他技术

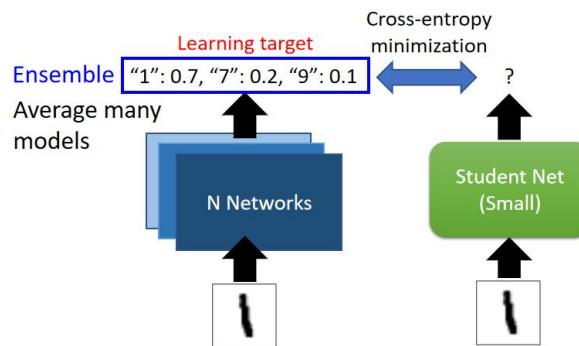
18.2.1 知识蒸馏

我们首先来介绍一种能让网络变小的方法——知识蒸馏（Knowledge Distillation）。先训练一个大的网络，这个大的网络在知识蒸馏里面称为教师网络（teacher network），这个教师网络需要训练其特定的任务，比如手写数字识别，如图所示，训练好的大网络收敛后，预测

“1”的结果可能“1”的概率是0.7，“7”的概率是0.2，“9”的概率是0.1。我们再定义一个小网络，称为学生网络（Student Net），让它学习教师网络的训练结果。为什么这么做呢？因为我们之前说过，如果让小的网络得到跟大网络一样的效果往往比较难训练，所以我们直接按照大网络的训练精度训练小网络，会更容易训练一些。



在图中给出的论文中有介绍说，利用这种方式训练出的小网络，可能就算不给它“7”数据集样本，也有可能有一定的概率预测出“7”这个数字。因为“1”跟“7”很相近。其实教师网络不一定是一个巨大的网络，也有可能是将多个网络集成得到的。但是多个网络组合的模型往往比较复杂，在实际应用中，我们可以训练一个学生网络，让结果逼近集成网络训练的结果，使得模型准确度差不多的情况下，复杂度大大减少。



在知识蒸馏中还有一个小技巧，在softmax函数基础上对每个输出结果加一个超参数T（Temperature），这样会对最后的预测结果进行一个平滑处理，让学生网络更好训练一些。

$$y'_i = \frac{\exp(y_i)}{\sum_j \exp(y_j)} \quad T = 100 \quad \Rightarrow \quad y'_i = \frac{\exp(y_i/T)}{\sum_j \exp(y_j/T)}$$

Below the equation, there are two bar charts. The first chart, labeled with $T = 100$, shows a sharp peak for the digit '1' (approx. 0.7) and very low values for '7' and '9'. The second chart shows a more uniform distribution where the peaks for all three digits ('1', '7', '9') are lower and closer in value, indicating a smoother (less confident) prediction due to the temperature scaling.

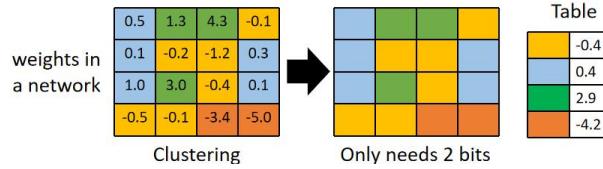
18.2.2 参数量化

接下来介绍下一个技巧：参数量化（parameter quantization），也可以称为参数压缩。这种方法主要是对权重在存储量上减少的一类方法，具体来说有如下几种方式：

最直观的一种方法是使用更少的字节来存储数值（参数），例如一般默认是32位，那我们可以用16或者8位来存数据。

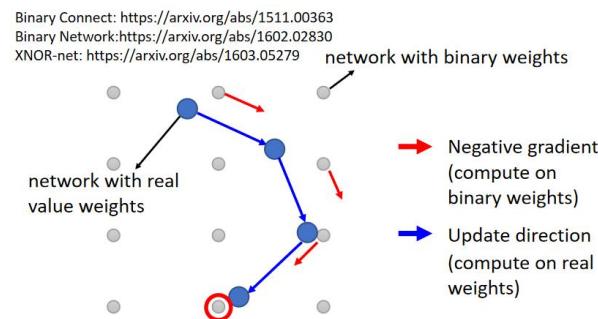
第二种方法是权重聚类（Weight Clustering），如下图所示，最左边表示网络中正常权重

矩阵，之后我们对这个权重参数做聚类，比如将 16 个参数做聚类，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 个字节，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。



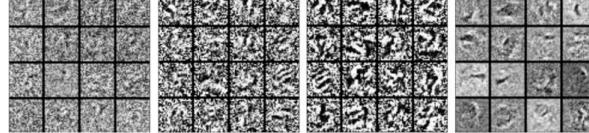
第三种方法是哈夫曼编码（Huffman Coding），具体做法是将常出现的东西用比较少的字节描述，不常出现的东西用比较多的字节描述，这样平均起来存储的数据量将大大减少，这也是一种有效减少数据量的方法。

权重可以压缩到什么程度呢？这里有一种方法叫二进制权重（Binary Weights），也就是网络中的权重要么是 +1，要么是 -1，这样每个权重只需要 1 个字节就可以存下来。



那这样训练出的网络效果会不会不太好？这里有一篇文章介绍了该方法用于 3 种数据集的图像分类问题中，结果发现 BinaryConnect 的方法识别错误率更小，原文给出的解释是这种方法会在一定程度上减少过拟合情况的发生，当然这种方法还是不如 Dropout 正则化。

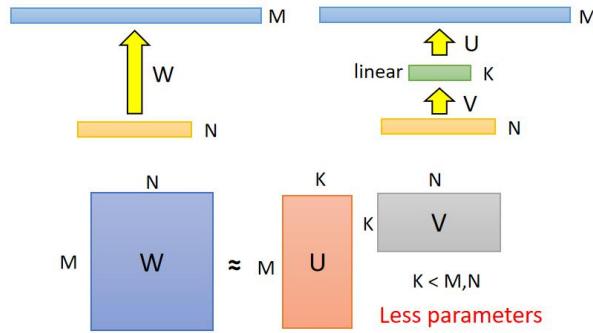
Method	MNIST	CIFAR-10	SVHN
No regularizer	$1.30 \pm 0.04\%$	10.64%	2.44%
BinaryConnect (det.)	$1.29 \pm 0.08\%$	9.90%	2.30%
BinaryConnect (stoch.)	$1.18 \pm 0.04\%$	8.27%	2.15%
50% Dropout	$1.01 \pm 0.04\%$		



<https://arxiv.org/abs/1511.00363>

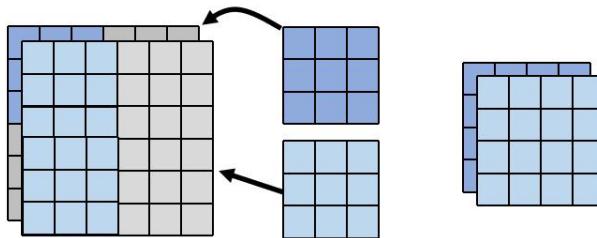
18.2.3 架构设计

接下来我们来介绍一下关于卷积神经网络的减少参数量的结构化设计。在架构设计中，第一种方法是低秩近似（Low-rank Approximation），如下图所示，左边是一个普通的全连接层，其权重矩阵的大小为 $M \times N$ ，低秩近似的原理就是在两个全连接层之间再插入一层 K ，新插入一层后的参数数量为 $N \times K + K \times M = K \times (M + N) < M \times N$ ，这是因为 K 远小于 M 和 N 。

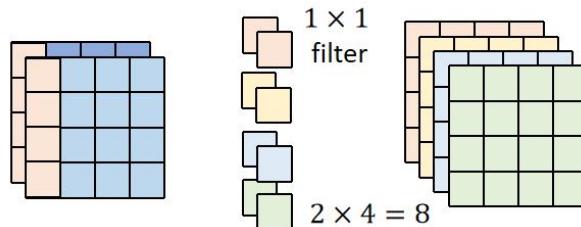


第二种做法是深度可分离卷积（Depthwise Separable Convolution），深度可分离卷积分步为两步：Depthwise Convolution（DW 卷积）和 Pointwise Convolution（PW 卷积）。

DW 卷积在做卷积的时候与传统的对图像做卷积有很大的不同。图片有几个通道就对应有几个滤波器，每个滤波器只管一个通道，因此通道之间的联系无法体现出来。



为了解决无法学习输入图像通道与通道之间联系的问题，将 DW 卷积的输出结果用的滤波器做卷积，以 4 个滤波器为例，效果如下：



结合上述思想，深度可分离卷积相当于将 CNN 中间多加了一层，这样就可以减少整体网络的参数量。关于网络结构设计方面还有一些文献参考，感兴趣可以看一下相关论文，这里就不多介绍。

- 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>
- GhostNet
 - <https://arxiv.org/abs/1911.11907>

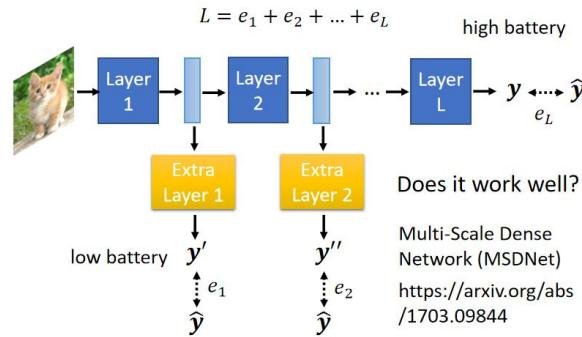
18.2.4 动态计算

最后一种技术叫动态计算（Dynamic Computation），它就是让神经网络可以自适应调整计算量，比如让神经网络自适应不同算力的设备，或者同一设备不同电量时对算力的分配。

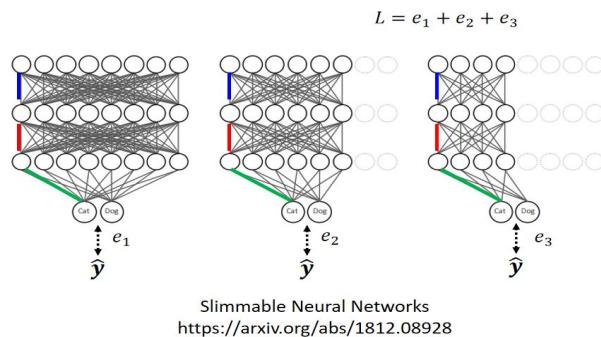
那么为什么不在一个设备上放好多个模型呢？因为需要占更多的空间。



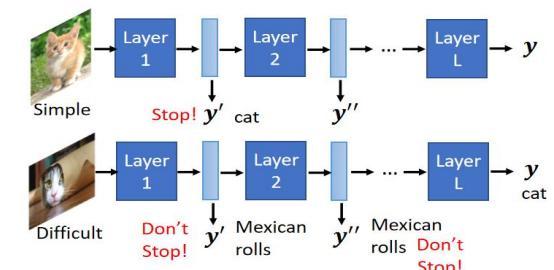
如何自适应调整网络的计算量呢，一种方式是调整网络的动态深度，举例来说，我们让每层之间都接一个输出层，并决定不同算力应该从哪个输出层输出出来。这样的网络训练过程也比较简单，一种很直接的方式就是把所有输出层的损失函数统统加起来，让其最小。这种方式效果到底如何，可以参考下面的文献（MSDNet）中给出的结果。



另外一种方式是调整网络的动态宽度，也就是对某一个网络，根据不同算力动态决定其宽度，训练方法与上述动态深度方法类似，所有输出的损失函数都加起来，让其最小。



上述两种方法都是人为决定根据设备不同的算力（比如电量）动态调整网络深度和宽度，但实际上来说，对于不同难度的训练样本可能需要的层数也不一样。具体来说，对于简单的样本可能几层就可以得到正确的结果，但是对于复杂的样本可能需要多训练几层才能得到正确的结果，关于通过样本自适应调整网络结构的研究可以参考一下下面的一些文献。



- SkipNet: Learning Dynamic Routing in Convolutional Networks
- Runtime Neural Pruning
- BlockDrop: Dynamic Inference Paths in Residual Networks

以上就是网络压缩的五个技术。前面四个技术都是让网络可以变小，这四个技术并不是互斥的。其实在做网络压缩的时候，可以既用网络架构，也做知识蒸馏，还可以在做完知识蒸馏以后，再去做网络剪枝。还可以在做完网络剪枝以后，再去做参数量化。如果想要把网络压缩到很小，这些方法都是可以一起被使用的。随着大模型的发展，网络压缩和量化变得越来越重要，这也是一个很值得研究的方向。

总结

本书从深度学习的基本概念开始，涵盖了几乎所有热门的研究方向。在书中，作者广泛参考了李宏毅的机器学习课程、cs231n 等优质人工智能课程的内容，为读者提供了一份深度学习领域的全面介绍。

书中首先介绍了深度学习的基本概念和发展历程，为初学者提供了对该领域的整体认识。接着，作者深入探讨了各种深度学习的基础理论，包括神经网络结构、激活函数、损失函数等重要概念，期待为读者打下了坚实的理论基础。

与此同时，书中还涉及了一些最新的研究进展和趋势，使读者能够了解深度学习领域的最新动态。通过对优质课程的参考，本书不仅注重理论知识的传授，更加注重实际应用和案例分析，帮助初学者更好地理解和运用深度学习技术。

总体而言，这本书以偏科普性质的方式，系统性地介绍了深度学习的方方面面，适合初学者入门。通过对优秀课程的借鉴，本书在理论和实践结合上下足功夫，为读者提供了一本全面而深入的深度学习入门指南。当然，如果有读者有志于在某一领域内进行深入探究的话，作者也推荐了一些论文和一些优质公开课程（比如 cs224n, cs224w, cs285 等），感兴趣的读者可以自行学习。此外，鉴于本书作者的水平有限，对某些地方的介绍可能有所不足，如果出现了一些问题敬请指正！