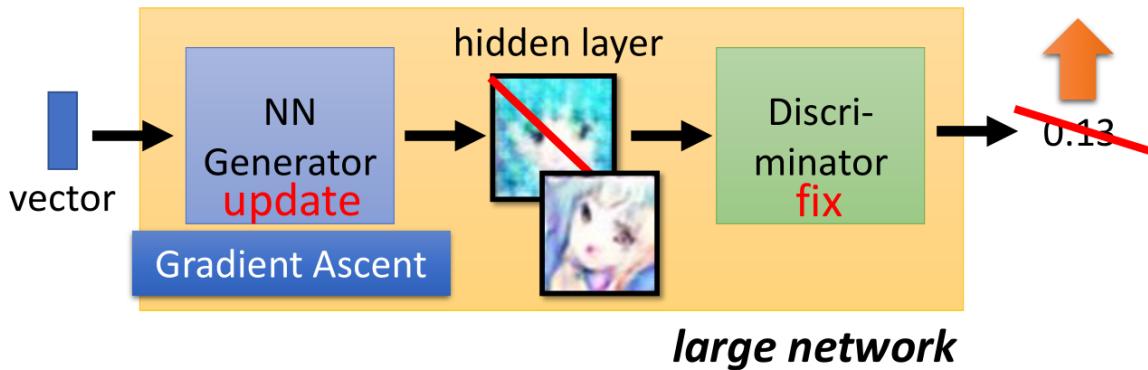


Step 2: Fix discriminator D, and update generator G

Generator learns to “fool” the discriminator



Algorithm Initialize θ_d for D and θ_g for G

- In each training iteration:

Learning
D

- Sample m examples $\{x^1, x^2, \dots, x^m\}$ from database
- Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from a distribution
- Obtaining generated data $\{\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^m\}$, $\tilde{x}^i = G(z^i)$
- Update discriminator parameters θ_d to maximize
 - $\tilde{V} = \frac{1}{m} \sum_{i=1}^m \log D(x^i) + \frac{1}{m} \sum_{i=1}^m \log (1 - D(\tilde{x}^i))$
 - $\theta_d \leftarrow \theta_d + \eta \nabla \tilde{V}(\theta_d)$

Learning
G

- Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from a distribution
- Update generator parameters θ_g to maximize
 - $\tilde{V} = \frac{1}{m} \sum_{i=1}^m \log (D(G(z^i)))$
 - $\theta_g \leftarrow \theta_g + \eta \nabla \tilde{V}(\theta_g)$

GAN as structured learning

Structured Learning

Machine learning is to find a function f

$$f : X \rightarrow Y$$

Regression: output a scalar

Classification: output a “class” (one-hot vector)

<table border="1"><tr><td>1</td><td>0</td><td>0</td></tr></table>	1	0	0	<table border="1"><tr><td>0</td><td>1</td><td>0</td></tr></table>	0	1	0	<table border="1"><tr><td>0</td><td>0</td><td>1</td></tr></table>	0	0	1
1	0	0									
0	1	0									
0	0	1									
Class 1	Class 2	Class 3									

Structured Learning/Prediction: output a sequence, a matrix, a graph, a tree

Output is composed of components with dependency

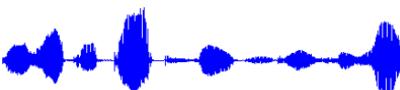
Output Sequence $f : X \rightarrow Y$

Machine Translation

X : “機器學習及其深層與
結構化”
(sentence of language 1)

Y : “Machine learning and
having it deep and structured”
(sentence of language 2)

Speech Recognition

X : 
(speech)

Y : 感謝大家來上課
(transcription)

Chat-bot

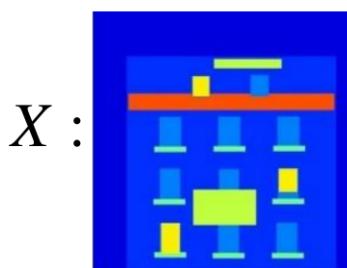
X : “How are you?”
(what a user says)

Y : “I'm fine.”
(response of machine)

Output Matrix

$$f : X \rightarrow Y$$

Image to Image



$Y :$



Colorization:



Ref: <https://arxiv.org/pdf/1611.07004v1.pdf>

Text to Image

$X :$ “this white and yellow flower have thin white petals and a round yellow stamen”

$Y :$



ref: <https://arxiv.org/pdf/1605.05396.pdf>

Why Structured Learning Challenging?

One-shot/Zero-shot Learning

- In classification, each class has some examples.
- In structured learning,
 - If you consider each possible output as a “class”
 - Since the output space is huge, most “classes” do not have any training data.
 - Machine has to create new stuff during testing.
 - Need more intelligence

Machine has to learn to do **planning**

- Machine generates objects component-by-component, but it should have a big picture in its mind.
- Because the output components have dependency, they should be considered globally.
- 在 Structured Learning 里面, 真正重要的是component和component之间的关系

Structured Learning Approach

综上, Structured Learning 有趣而富有挑战性的问题。而GAN 他其实是一个 Structured Learning 的 solution。

Structured Learning有两种方法, Bottom Up和Top Down, 前者是每个component分开产生, 缺点是容易失去大局观, 后者是产生一个完整的object后再从整体来看这个object好不好, 缺点是这个方法很难做 generation

Generator 可以视为是一个 Bottom Up 的方法, Discriminator 可以视为是一个 Top Down 的方法, 把这两个方法结合起来就是 Generative Adversarial Network, 就是 GAN。

Generator

Learn to generate
the object at the
component level



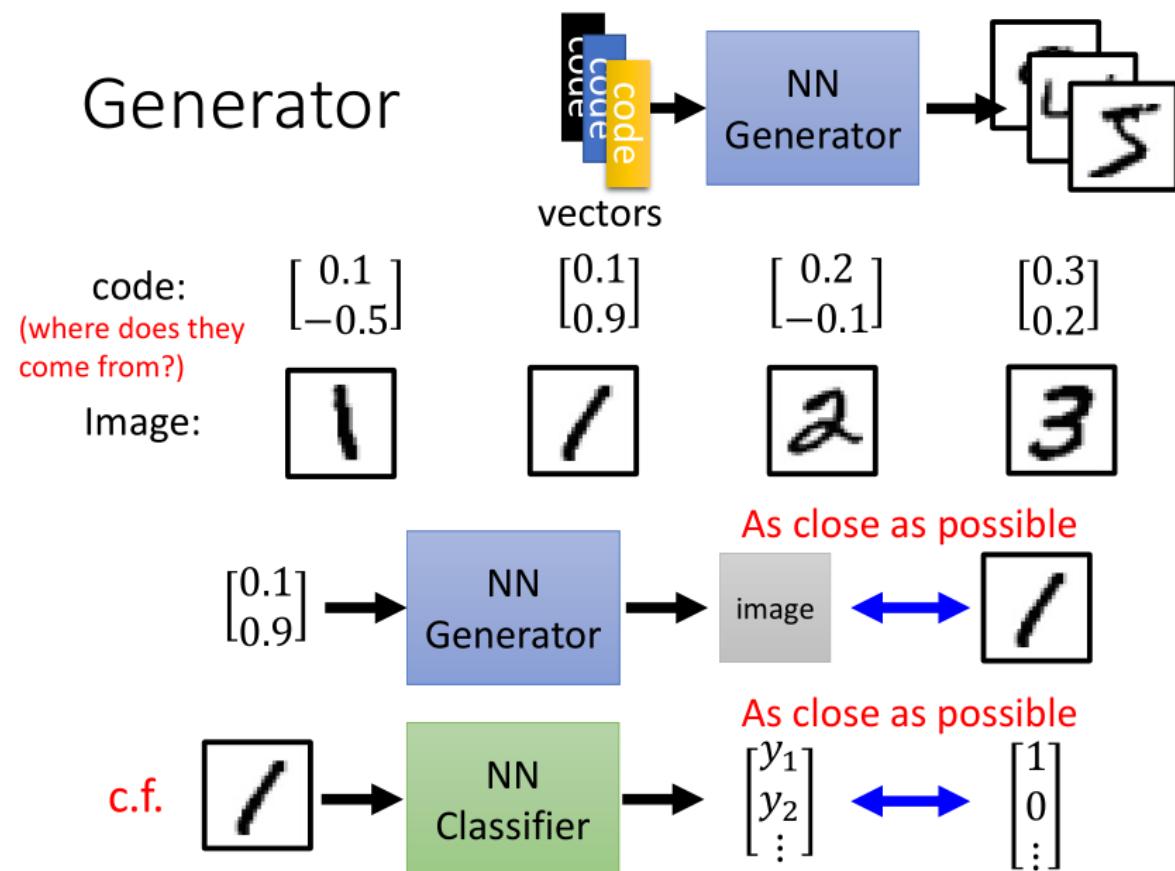
Discriminator

Evaluating the
whole object, and
find the best one



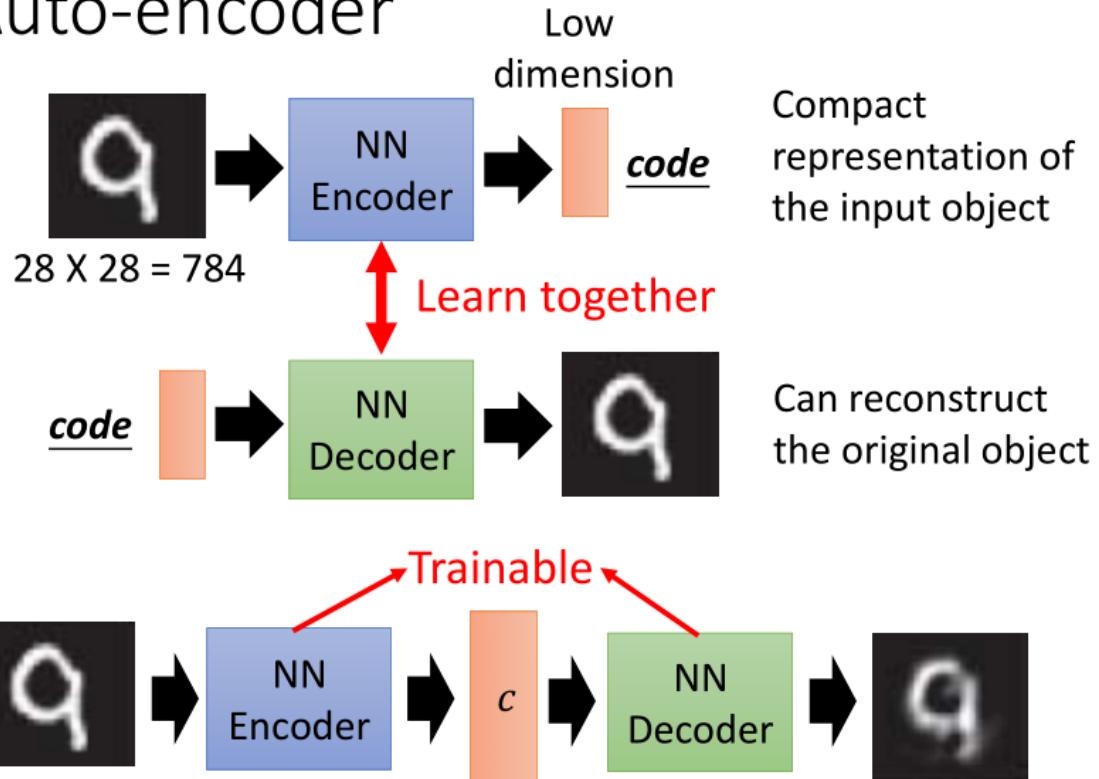
Can Generator learn by itself?

事实上Generator是可以自己学的。在传统的Supervised Learning里面，给network input跟output的pair，然后train下去就可以得到结果。搜集一堆图片，并给每张图片assign一个vector即可。输入是一个vector，输出是图片（一个很长的向量）。因此NN Generator和NN Classifier其实可以用同样的方式来train（两者输入输出都是向量）。

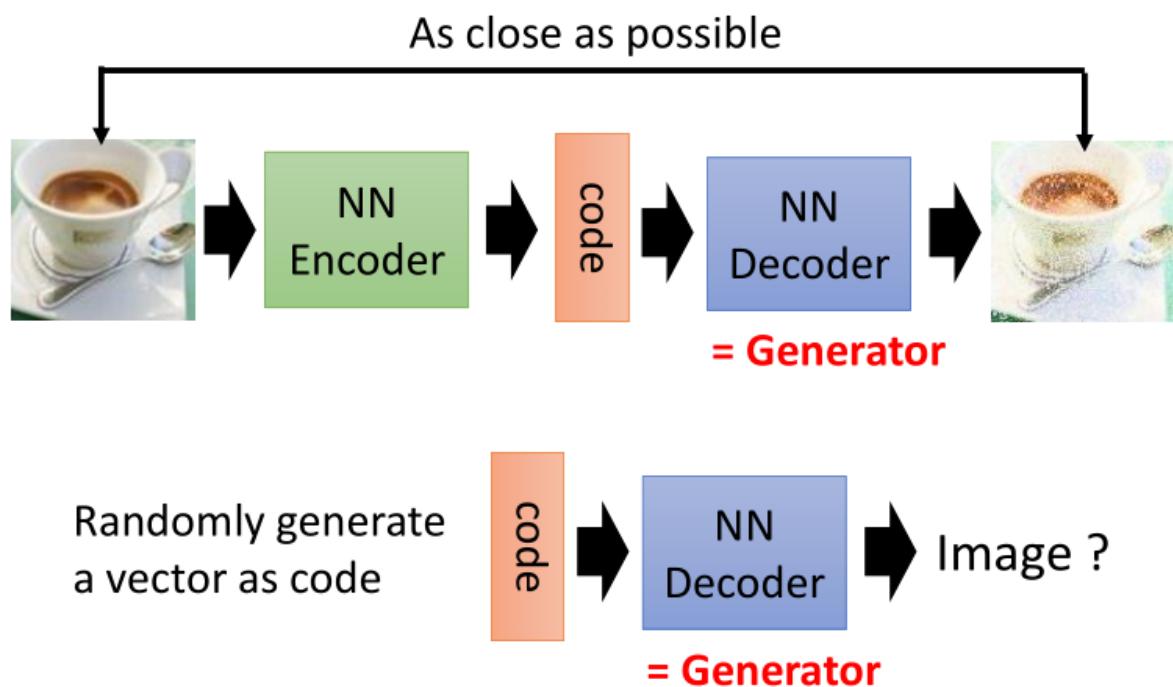


问题在于如何产生input vector, Encoder in auto-encoder provides the code

Auto-encoder

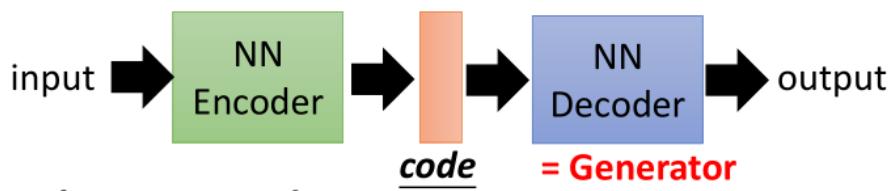


Decoder 其实就是我们要的 generator, 随便丢一些东西就会 output 你想要的 object

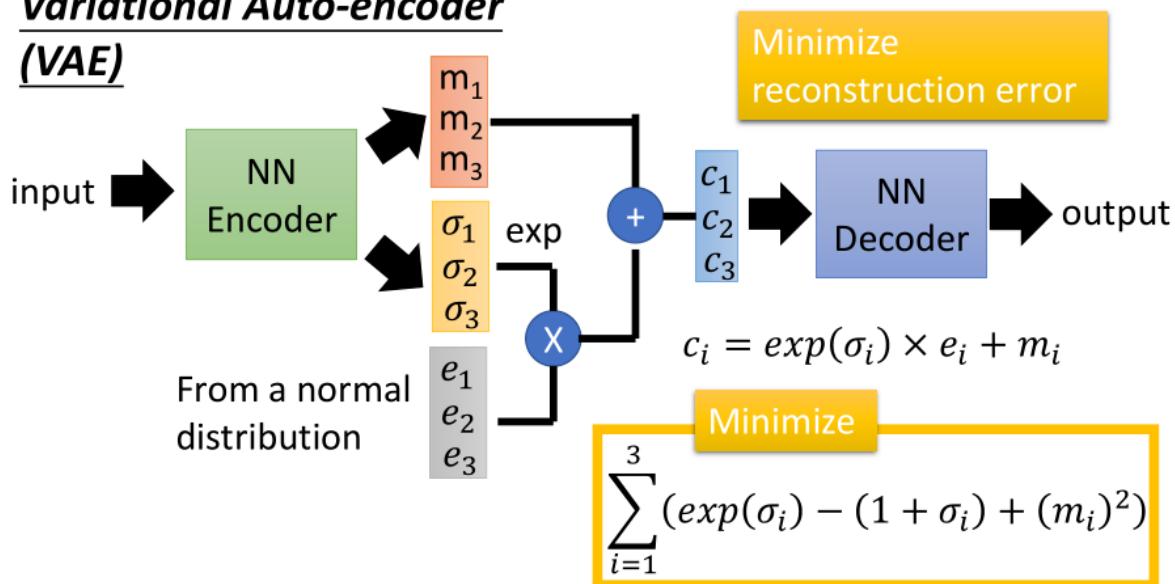


也可以用VAE把decoder train 的更加稳定

Auto-encoder



Variational Auto-encoder (VAE)



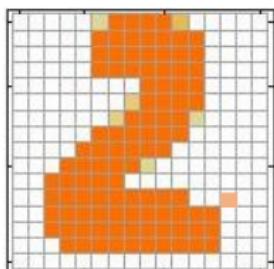
至于code的维度，是需要自己决定的，train 不起来，增加 dimension 会有效，但是增加 dimension 以后未必会得到你要的东西，因为 train 的是一个 Auto-encoder，训练的目标是要让 input 跟 output 越接近越好，要达到这个目标其实非常简单，把中间那个 code 开得跟 input 一样大，就不会有任何 loss，因为 machine 只要学着一直 copy 就好了，但这个并不是我们要的结果。虽然说 input 的 vector 开得越大 loss 可以压得越低，但 loss 压得越低并不代表 performance 会越好。

What do we miss?

It will be fine if the generator can truly copy the target image.

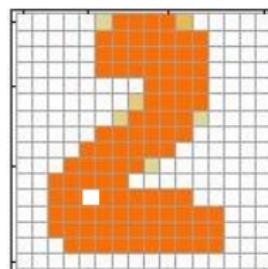
What if the generator makes some mistakes

- Some mistakes are serious, while some are fine.



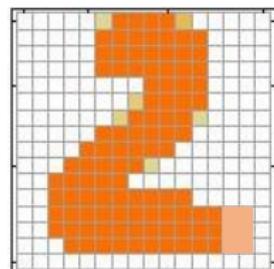
1 pixel error

我覺得不行



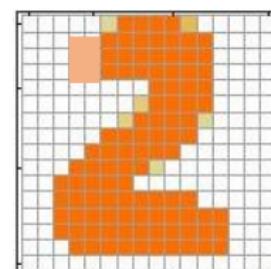
1 pixel error

我覺得不行



6 pixel errors

我覺得其實
可以



6 pixel errors

我覺得其實
可以

不能够单纯的去让output跟目标越像越好

在 Structured Learning 里面 component 和 component 之间的关系是非常重要的，但一个 network 的架构其实没有那么容易让我们把 component 和 component 之间的关系放进去。

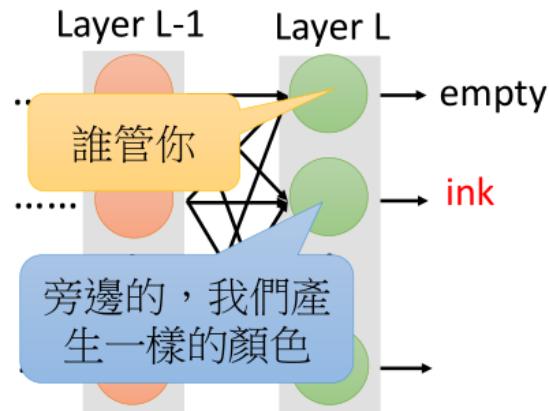
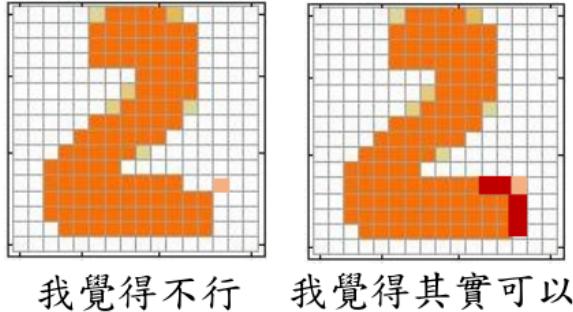
Although highly correlated, they cannot influence each other.

Need deep structure to catch the relation between components.

根据经验，如果有同样的 network，一个用 GAN train，一个用 Auto-encoder train。往往就是 GAN 的那个可以产生图片，Auto-encoder 那个需要更大的 network 才能够产生跟 GAN 接近的结果。因为要把 correlation 考虑进去会需要比较深的 network。

What do we miss?

Each neural in output layer corresponds to a pixel.



The relation between the components are critical.

Although highly correlated, they cannot influence each other.

Need deep structure to catch the relation between components.

Can Discriminator generate?

可以，但很卡

Discriminator也被称为Evaluation function, Potential Function, Energy Function ...

Discriminator相较于Generator来说，考虑component和component之间的correlation比较容易，检查correlation对不对是比较容易的。因为是产生完一张完整的 image 以后再把这张 image 丢给discriminator。

如何用Discriminator生成？

Suppose we already have a good discriminator
 $D(x)$...

Inference

- Generate object \tilde{x} that

$$\tilde{x} = \arg \max_{x \in X} D(x)$$

Enumerate all possible x !!!

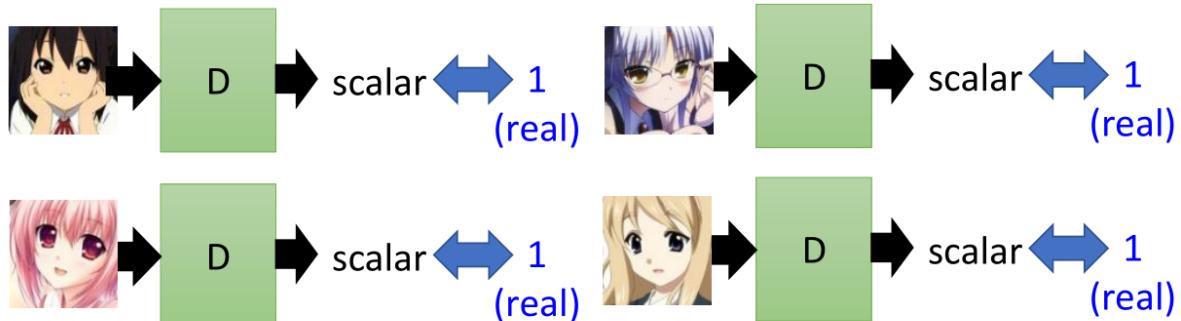
It is feasible ???

How to learn the discriminator?

Training

如何训练Discriminator?

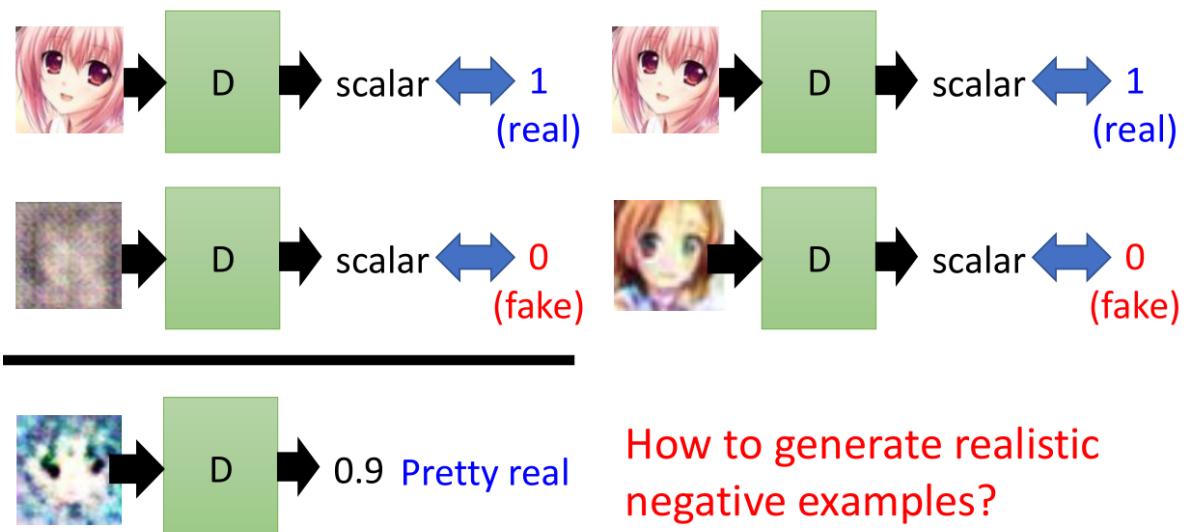
- I have some real images



Discriminator only learns to output “1” (real).

Discriminator training needs some negative examples.

- Negative examples are critical.



从哪里找 negative example 非常关键。如果找出来的 negative example 非常的差，你就跟机器说人画的就是好的，就是要给高分，然后随机产生一大堆的 noise，这些 noise 就是要给它低分。对机器来说当然可以分辨这两种图片的差别，但之后给它左下角这种图片，也许画得很差，但是它觉得这个还是比 noise 好很多，也会给它高分，这个不是我们要的。

假设可以产生非常真实的 negative example，但还是有些错，比如说两只眼睛的颜色不一样，你可以产生非常好的 negative example，这样 discriminator 才能够真的学会鉴别好的 image 跟坏的 image。

现在问题就是怎么产生这些非常好的 negative example。要产生这些好的 negative example也需要一个很好的 process 去产生这些 negative example，现在就是不知道怎么产生 image 才要 train model，这样就变一个鸡生蛋，蛋生鸡的问题。要有好的 negative example 才能够训练 discriminator，要有好的 discriminator 才能够帮我们找出好的 negative example。

实际上可以迭代训练

General Algorithm



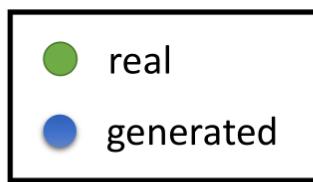
- Given a set of **positive examples**, randomly generate a set of **negative examples**.
- In each iteration
 - Learn a discriminator D that can discriminate positive and negative examples.



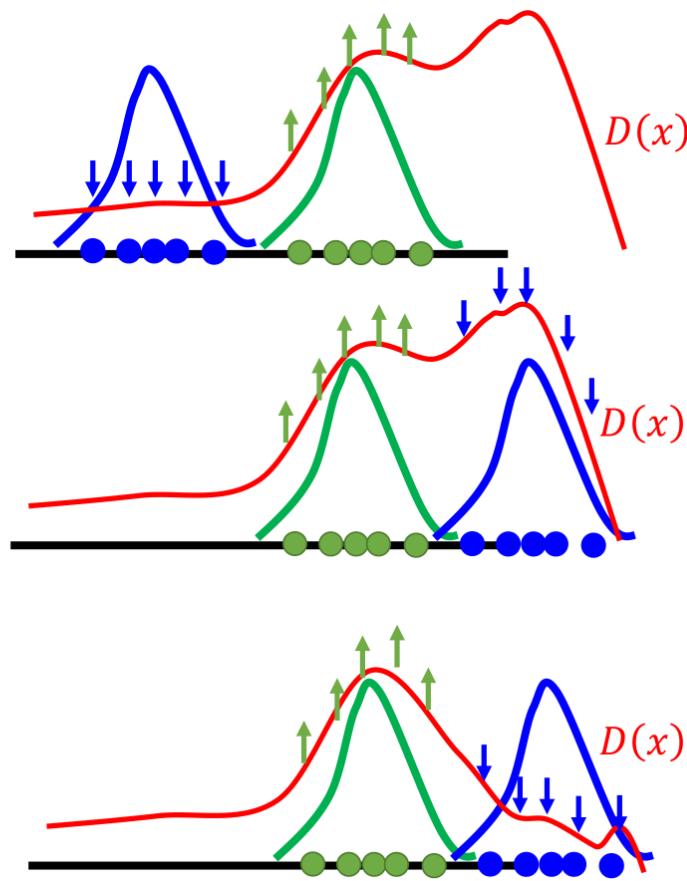
- Generate negative examples by discriminator D

$$\tilde{x} = \arg \max_{x \in X} D(x)$$

Discriminator - Training



In the end



Structured Learning and Graphical Model

有人真的拿 discriminator 做生成吗?有的，其实有满坑满谷的work都是拿 discriminator 来做生成。

假设你熟悉整个 Graphical Model 的 work 的话，仔细想一下，刚才讲的那个 train discriminator 的 process 其实就是 general 的 Graphical Model 的 training。只是在不同的 method 里面讲法会略有不同而已。

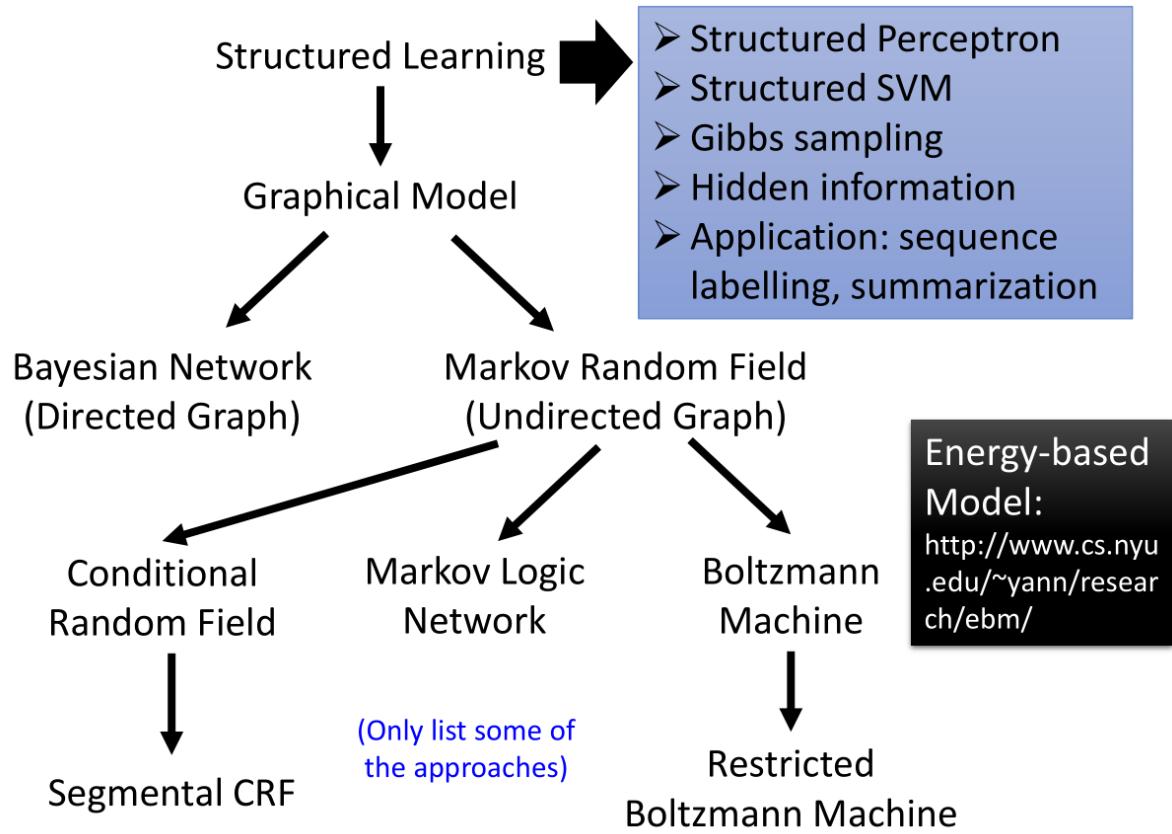
Graphical Model 其实就是 Structured Learning 的一种，Graphical Model 里面又分成很多类，比如说有 Bayesian Network、Markov Random Field 等等。

Structured Learning 的技术里面，其中非常具有代表性的东西就是 Graphical Model

在 Markov Random Field、Bayesian Network 里面定一个 graph，这个 graph 上面有一个 Potential Function，这个东西就是你的 discriminator。你输入你的 observation，那个 graph 会告诉你这组 data 产生出来的机率有多少，那个机率就是 discriminator assign 的分数。

Graphical Model 里面的那个 graph、你的 Potential Function、你的 Markov Random Field、你的 Bayesian Network 其实就是 discriminator。

再回想一下当你去 train Structured SVM、train Graphical Model 的时候，train 种种和 Structured Learning 有关的技术的时候是不是 iterative 的去 train 的。你做的事情是不是用 positive 用 negative example 训练出你的 model，接下来用 model sample 出 negative example 再去 update model。就跟我刚才讲的 training discriminator 的流程其实是一样的，只是把同样的事情换个名词来讲，让你觉得不太一样而已。但你仔细想想，它们就是同一回事。



Generator v.s. Discriminator

Generator

- Pros
 - Easy to generate even with deep model
- Cons
 - Imitate the appearance
 - Hard to learn the correlation between components

Discriminator

- Pros
 - Considering the big picture
- Cons
 - Generation is not always feasible (不知道如何解argmax)
 - Especially when your model is deep
 - How to do negative sampling?

Generator + Discriminator

General Algorithm



- Given a set of **positive examples**, randomly generate a set of **negative examples**.
- In each iteration
 - Learn a discriminator D that can discriminate positive and negative examples.



- Generate negative examples by discriminator D

$$\boxed{G \rightarrow \tilde{x}} = \boxed{\tilde{x} = \arg \max_{x \in X} D(x)}$$

GAN不一样的就是我们有了 generator，generator 就是取代了这个 arg max 的 problem。

本来要一个 algorithm 来解这个 arg max 的 problem，往往我们都不知道要怎么解。但现在用 generator 来产生 negative example，generator 它就可以产生出 \tilde{x} ，即可以让 discriminator 给它高分的 image。

所以可以想成 generator 在学怎么解 arg max 这个 problem。学习的过程中就是在学怎么产生 discriminator 会高分的那些 image。

过去是解这样一个 optimization 的 problem，现在不一样，是用一个 intelligent 的方法，用一个 network 来解这个 arg max 的 problem。

Benefit of GAN

From Discriminator's point of view

- Using generator to generate negative samples

$$\boxed{G \rightarrow \tilde{x}} = \boxed{\tilde{x} = \arg \max_{x \in X} D(x)}$$

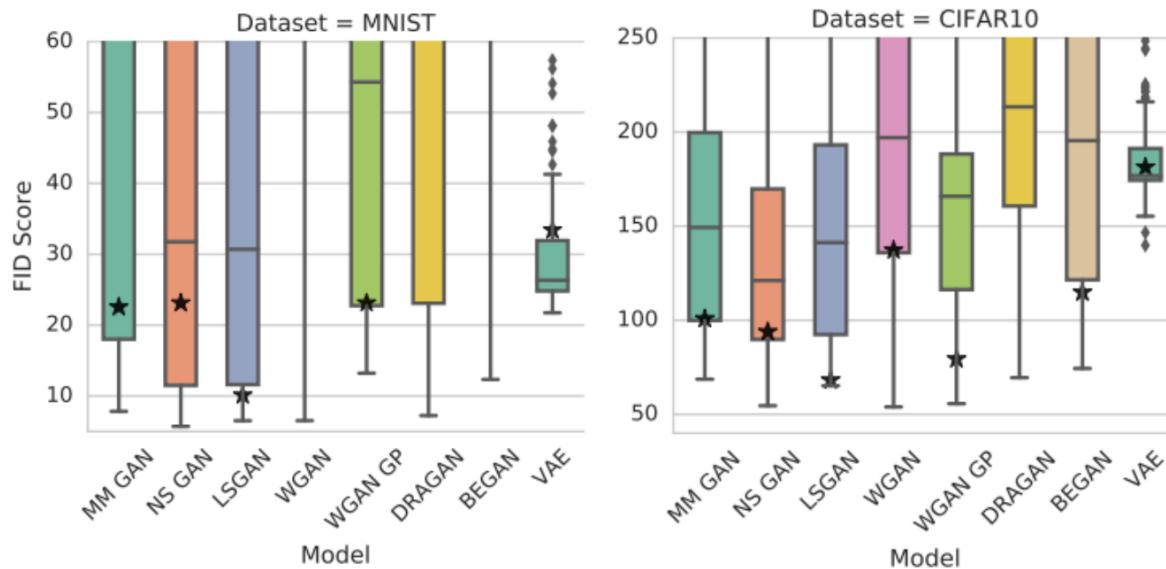
efficient

From Generator's point of view

- Still generate the object component-by-component
- But it is learned from the discriminator with global view.

从 discriminator 的角度来看，过去不知道怎么解 arg max 的 problem，现在用 generator 来解 arg max 的 problem，显然是比这个方法更加有效，而且更容易一般化的。

对 generator 来说，它在产生 object 的时候仍然是 component by component 的产生，但是得到的 feedback 不再是 L1 L2 的 loss，不再是 pixel by pixel 的去算两张图片的相似度。它的 loss 将是来自于一个有大局观的 discriminator。希望通过 discriminator 带领，generator 可以学会产生有大局观的结果。



FID [[Martin Heusel, et al., NIPS, 2017](#)]: Smaller is better

这是来自于 Google 的一篇 paper，这篇 paper 主要的内容是想要比较各种不同 GAN 的技术。它得到的结论是所有不同的 GAN 其实 performance 都差不多。

这个纵轴是 FID Score，FID Score 越小代表产生出来的图片越像真实的图片

对于不同的 GAN 它们都试了各种不同的参数，GAN 在 training 的时候是非常的 sensitive 的，往往可能只有特定某一组参数才 train 得起来，它会发现 GAN 用不同的参数它的 performance 有一个非常巨大的 range。

如果比较 VAE 跟这些 GAN 的话可以发现，VAE 倒是明显的跟 GAN 有非常大的差别。首先 VAE 比较稳，给不同的参数，VAE 的分数非常的集中，虽然比较稳，但它比较难做到最好，所以比较每一个 model 可以产生的最好的结果的话，VAE 相较于 GAN 还是输了一截的。

Conditional Generation by GAN

Text-to-Image

Traditional supervised approach

A blurry image

Text-to-Image

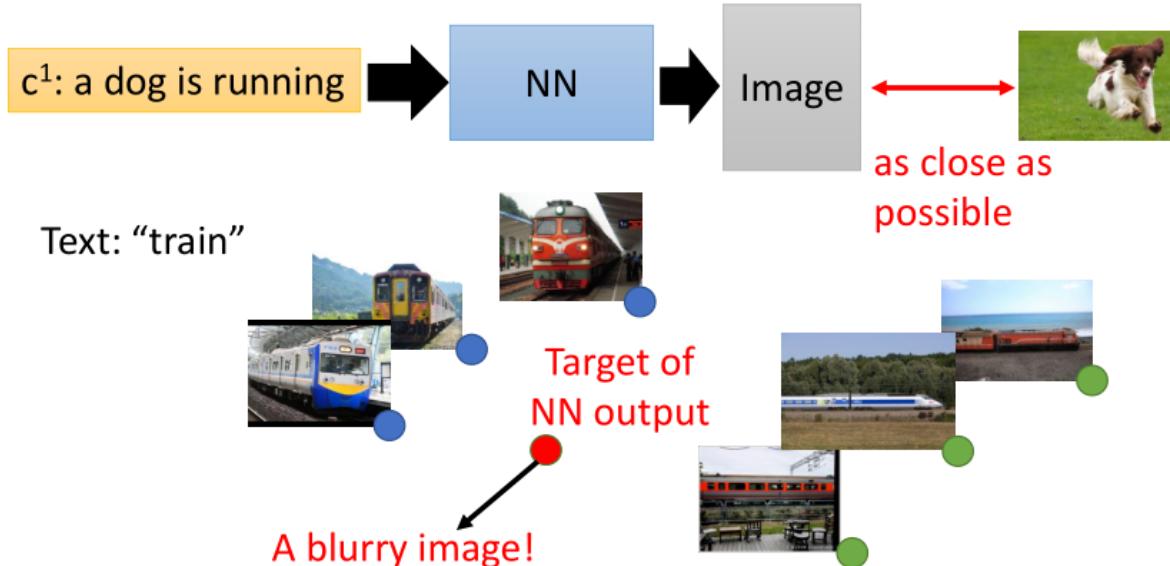
a dog is running



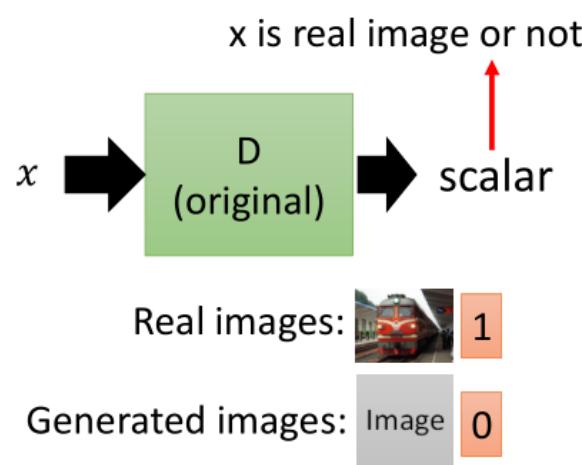
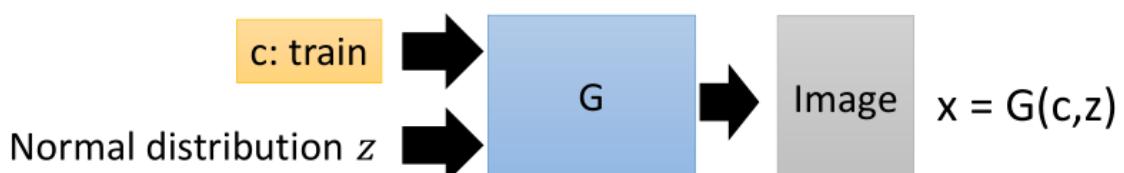
a bird is flying



- Traditional supervised approach



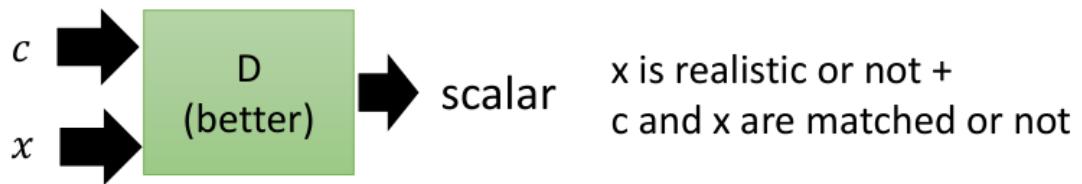
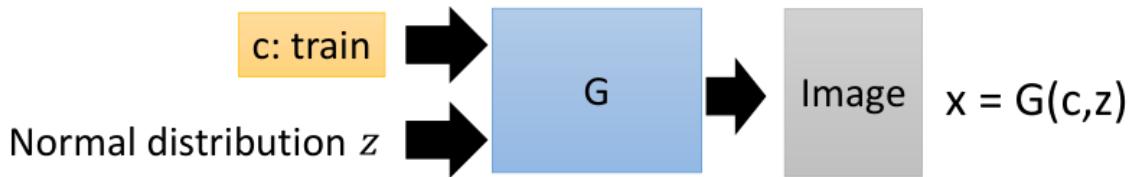
Conditional GAN



Generator will learn to generate realistic images

But completely ignore the input conditions.





True text-image pairs: (train , ) 1

(cat , ) 0 (train , ) 0

Algorithm

- In each training iteration
- Learning D
 - Sample m positive examples $\{(c^1, x^1), (c^2, x^2), \dots, (c^m, x^m)\}$ from database
 - Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from a distribution
 - Obtaining generated data $\{\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^m\}, \tilde{x}^i = G(c^i, z^i)$
 - Sample m objects $\{\hat{x}^1, \hat{x}^2, \dots, \hat{x}^m\}$ from database
 - Update discriminator parameters θ_d to maximize

$$\begin{aligned}\tilde{V} = & \frac{1}{m} \sum_{i=1}^m \log D(c^i, x^i) + \\ & \frac{1}{m} \sum_{i=1}^m \log(1 - D(c^i, \tilde{x}^i)) + \\ & \frac{1}{m} \sum_{i=1}^m \log(1 - D(c^i, \hat{x}^i))\end{aligned}$$

$$\theta_d \leftarrow \theta_d + \eta \nabla \tilde{V}(\theta_d)$$

- Learning G
 - Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from a distribution
 - Sample m conditions $\{c^1, c^2, \dots, c^m\}$ from a database
 - Update generator parameters θ_g to maximize

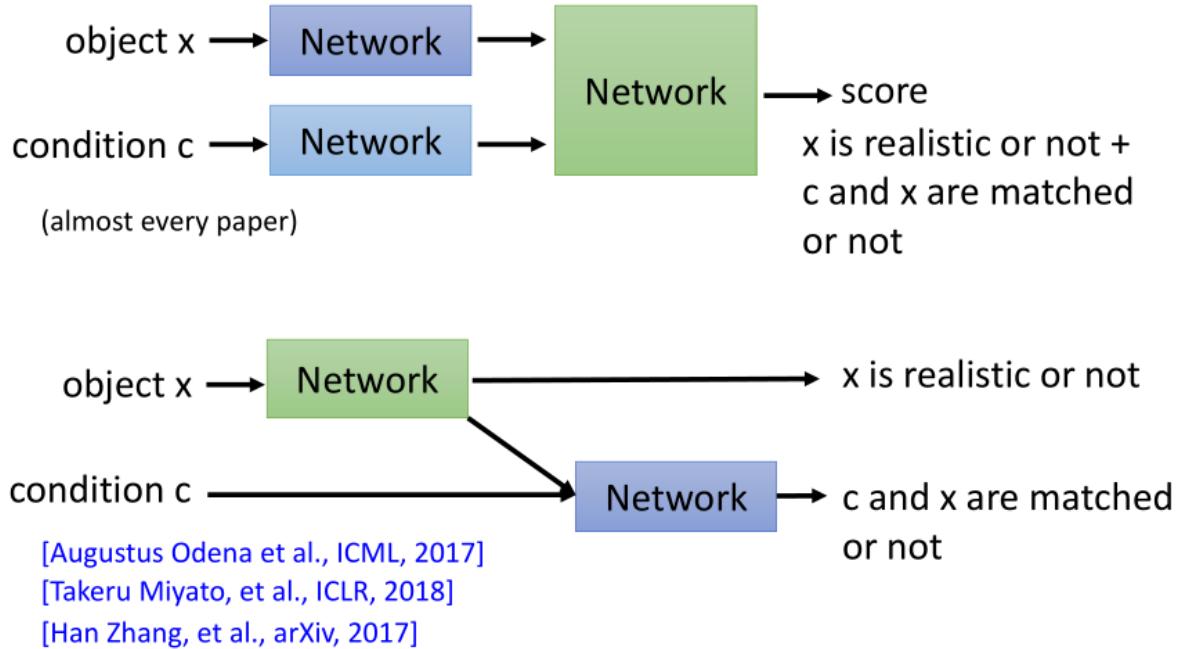
$$\begin{aligned}\tilde{V} = & \frac{1}{m} \sum_{i=1}^m \log(D(G(c^i, z^i))) \\ \theta_g \leftarrow & \theta_g - \eta \nabla \tilde{V}(\theta_g)\end{aligned}$$

Discriminator

有两种常见架构。

下面的一个架构拆开两个evaluation可能是更合理的，因为给一个清晰的图片低分可能会让Network confused，可能会觉得这个图片不够清晰。因为有两种错误，机器并不知道是哪种情况的错误，它需要自己分辨。

分开两个case，可以让机器有针对性的使某个case的值变化，当x清晰时，只改变match score即可。



Stack GAN

先产生比较小的图，再产生比较大的图

Stack GAN

Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, Dimitris Metaxas, "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks", ICCV, 2017

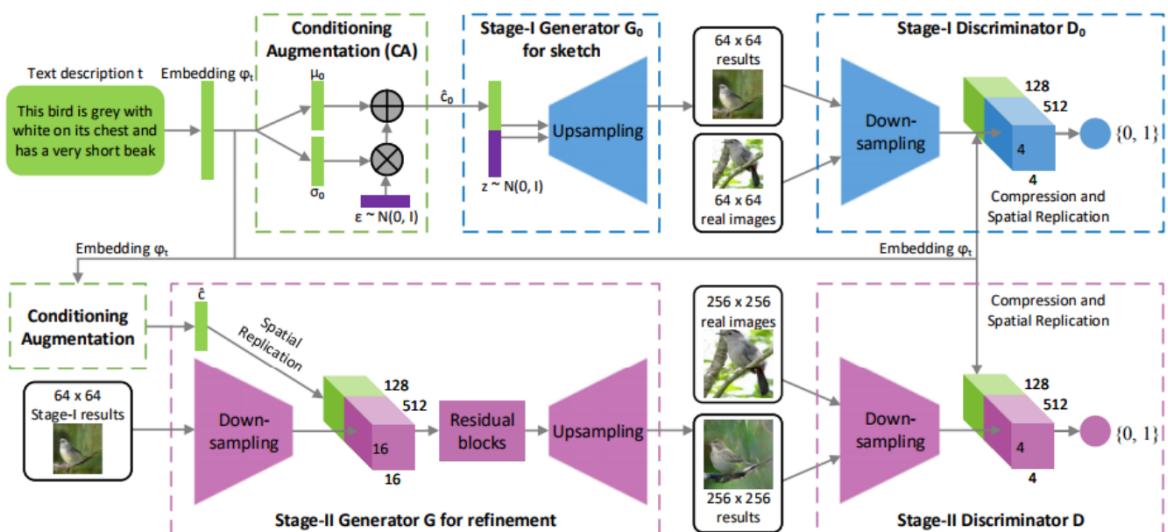
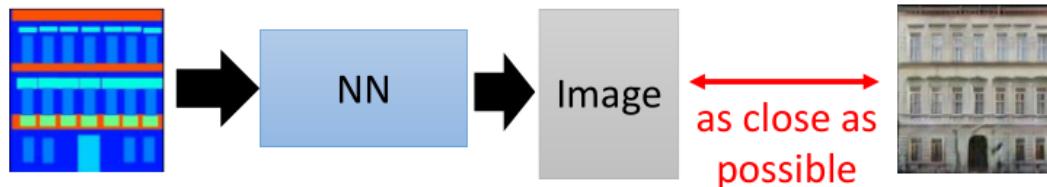


Image-to-image

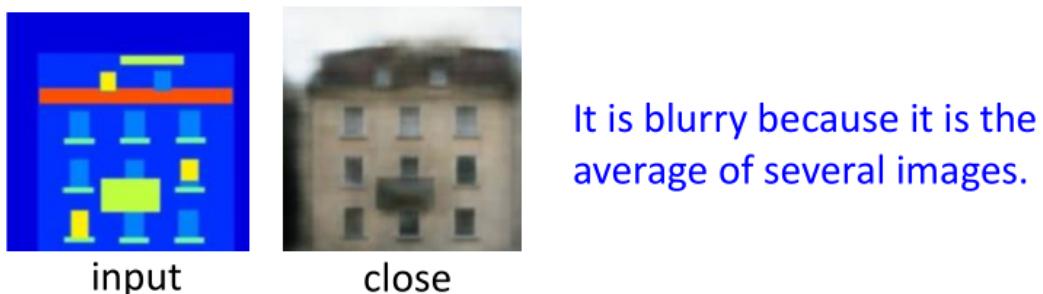
Traditional supervised approach

同一个image可以对应到不同的房子，因此会产生平均的结果，图片会是比较模糊的。

- Traditional supervised approach



Testing:

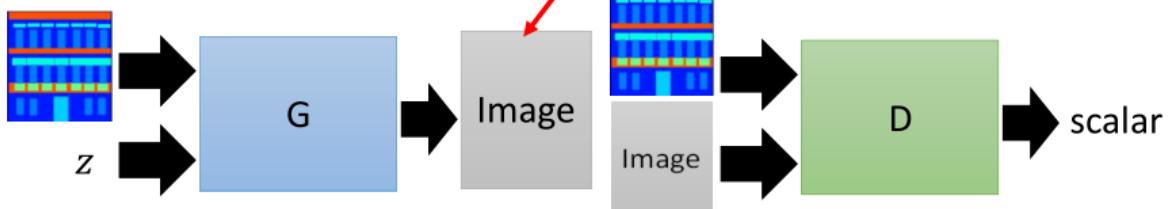


Conditional GAN

GAN有时也会生成奇怪的东西，如左上角。可以加一个constraint，希望Generator Output与原目标越接近越好，考虑这种情况下效果会更好。

Image-to-image

- Experimental results



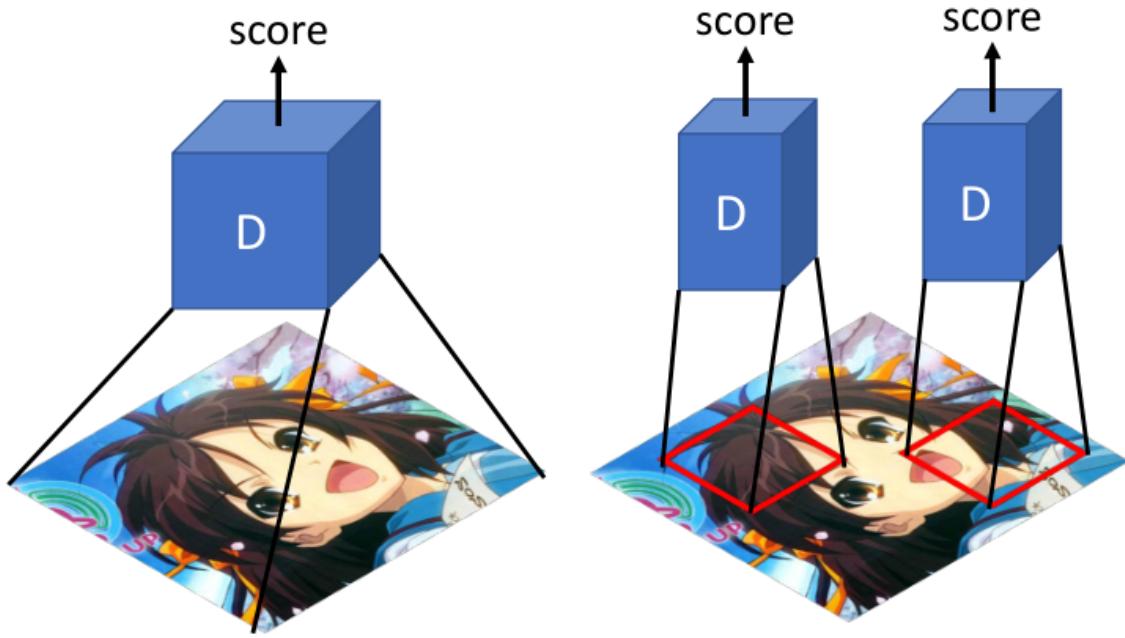
Testing:



Patch GAN

如果image很大，Discriminator参数量太多很容易overfitting或train的时间非常长，因此在image-to-image论文中，每次Discriminator不是检查一整张图片，而是每次检查一小块图片，这样叫Patch GAN。patch的size需要调整。

<https://arxiv.org/pdf/1611.07004.pdf>



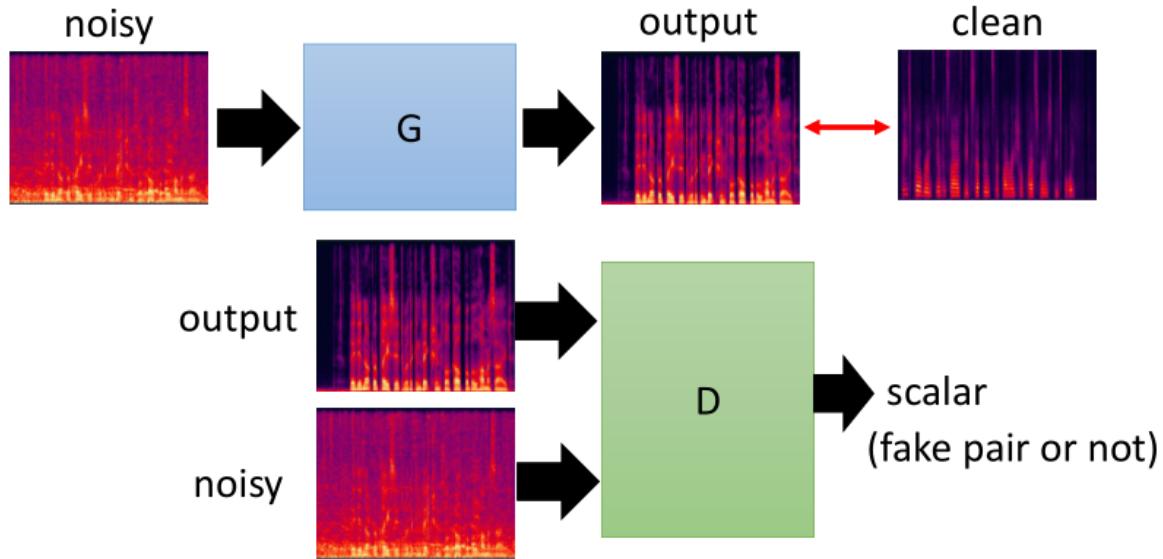
Speech Enhancement

Typical deep learning approach

找很多声音，然后把这些声音加上一些杂讯，接下来，你就 train 一个 generator，input 一段有杂讯的声音，希望 output 就是没有杂讯的声音。

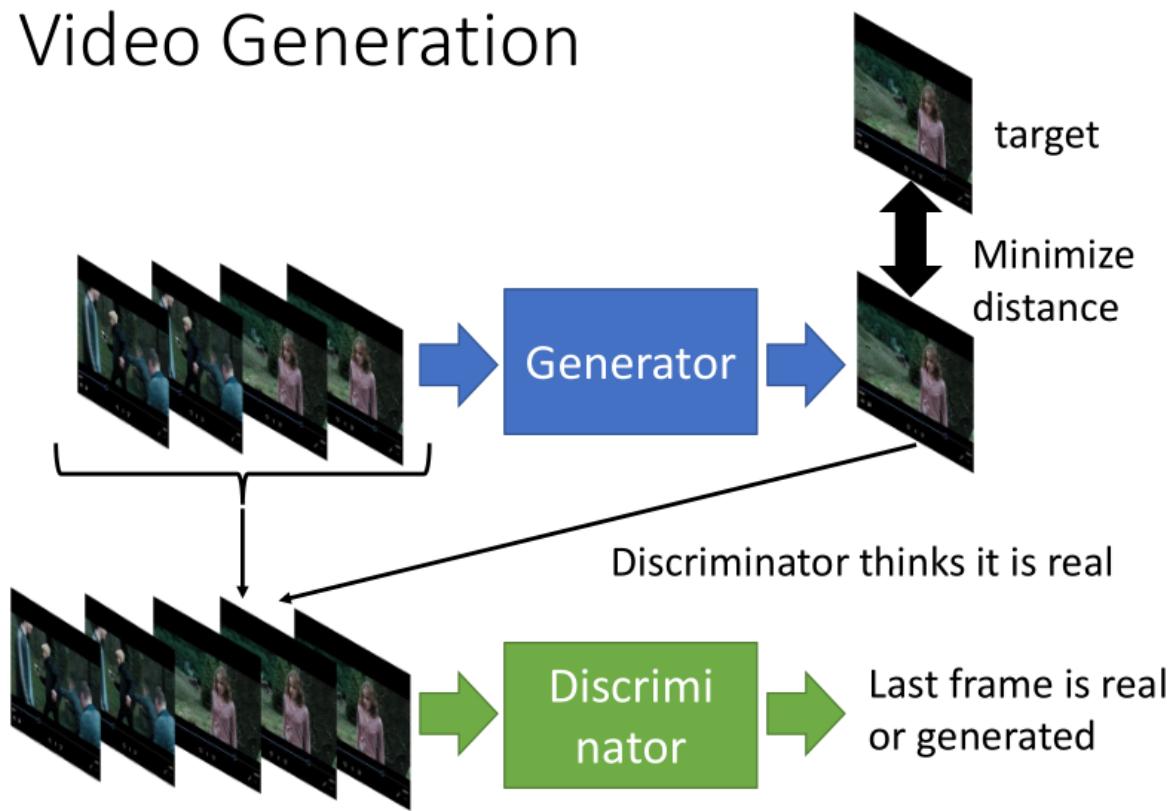
一段声音讯号可以用 spectrum 来表示，它看起来就像是一个 image 一样，所以这个 generator 常常也会直接套用那些 image 上常用的架构。

Conditional GAN



在 conditional GAN 里面，discriminator 要同时看generator 的 input 跟output，然后给它一个分数。这个分数决定现在 output 的这一段声音讯号是不是 clean 的，同时 output 跟input 是不是 match 的。

Video Generation



Unsupervised Conditional Generation by GAN

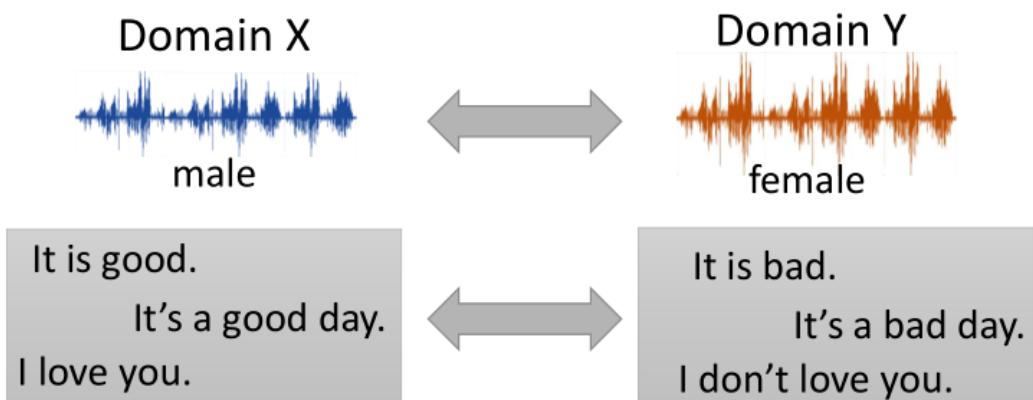
Unsupervised Conditional Generation

如果是 supervised conditional generation，你需要 label 告诉机器什么样子的 input，应该有什么样的 output。

所以今天我们要讨论的问题就是，有没有办法做到 unsupervised conditional generation。只有两堆 data，machine 自己学到怎么从其中一堆转到另外一堆。这样的技术，有很多的应用，不是只能够用在影像上。



Transform an object from one domain to another
without paired data (e.g. style transfer)



我 surveyed 一下文献，我认为大致上可分为两大类的作法

Approach 1: Direct Transformation

第一大类的做法是直接转。直接 learn 一个 generator，input x domain 的东西，想办法转成 y domain 的东西。在经验上，如果你今天要用这种 direct transformation 的方法，你的 input output 没有办法真的差太多。如果是影像的话，它通常能够改的是颜色、质地。所以如果是画风转换，这个是比较有可能用第一个方法来实践。

那今天假设你要转的 input 跟 output 差距很大，它们不是只有在颜色、纹理上面的转换的话，那你就需要用到第二个方法。

Approach 2: Projection to Common Space

第二个方法是如果今天你的 input 跟 output，差距很大。比如说你要把真人转成动画人物。真人跟动画人物就是不像，它不是你改改颜色，或改改纹理。就可以从真人转成动画人物的。

你先 learn 一个 encoder 比如说第一个 encoder 做的事情就是吃一张人脸的图，然后它把人脸的特征抽出来，比如说男，戴眼镜，接下来你输入到一个 decoder，这个 decoder 它画出来的就是动画的人物，它根据你 input 的人脸特征比如说是男的，有戴眼镜的，去产生一个对应的角色。如果你 input output 真的差很多的时候，你就可以做这件事。

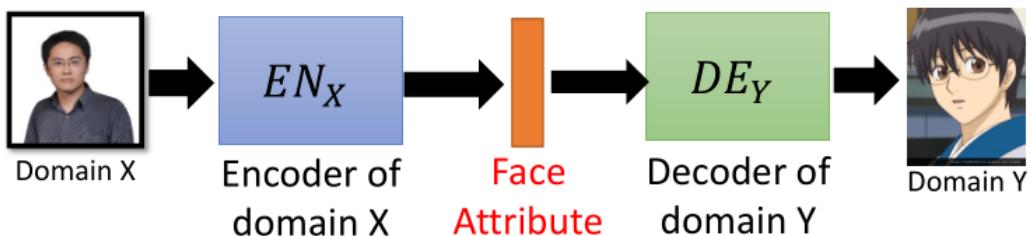
Unsupervised Conditional Generation

- Approach 1: Direct Transformation



For texture or color change

- Approach 2: Projection to Common Space



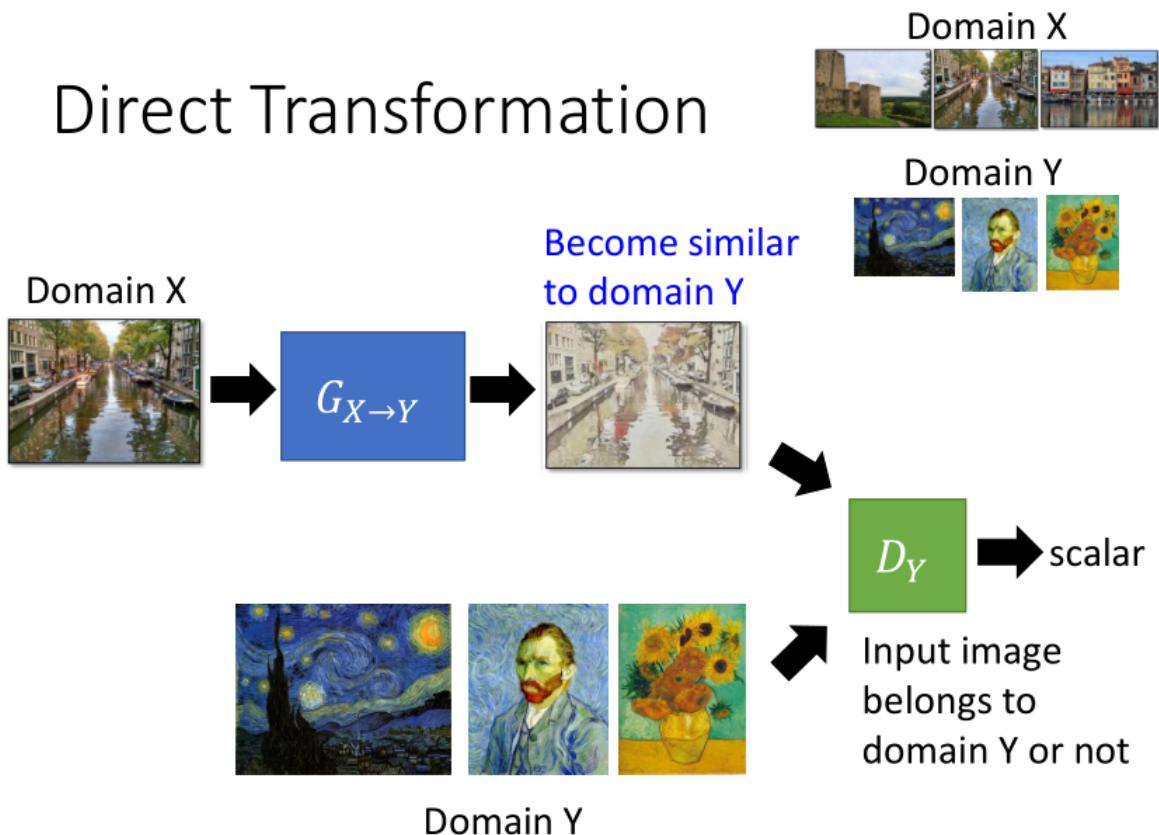
Larger change, only keep the semantics

Direct Transformation

第一个做法是说，我们要 learn 一个 generator。这个 generator input x domain 的东西，要转成 y domain 的东西。那我们现在 x domain 的东西有一堆，y domain 的东西有一堆，但是合起来的 pair 没有，我们没有它们中间的 link。那 generator 怎么知道给一个 x domain 的东西，要 output 什么样 y domain 的东西呢？用 supervised learning 当然没有问题，但现在是 unsupervised generator 怎么知道如何产生 y domain 的东西呢？

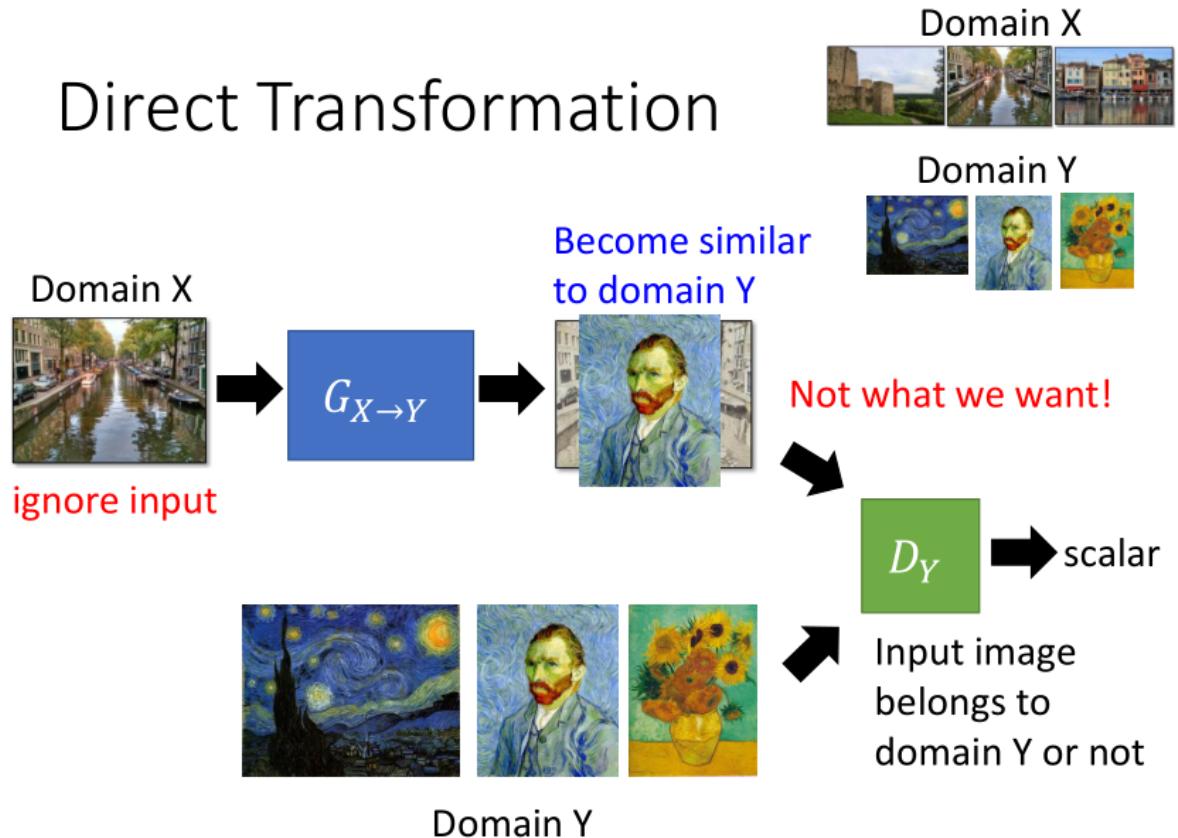
这个时候你就需要一个 y domain 的 discriminator。这个 discriminator 做的事情是，它可以鉴别说这张 image 是 x domain 的 image，还是 y domain 的 image。

Direct Transformation

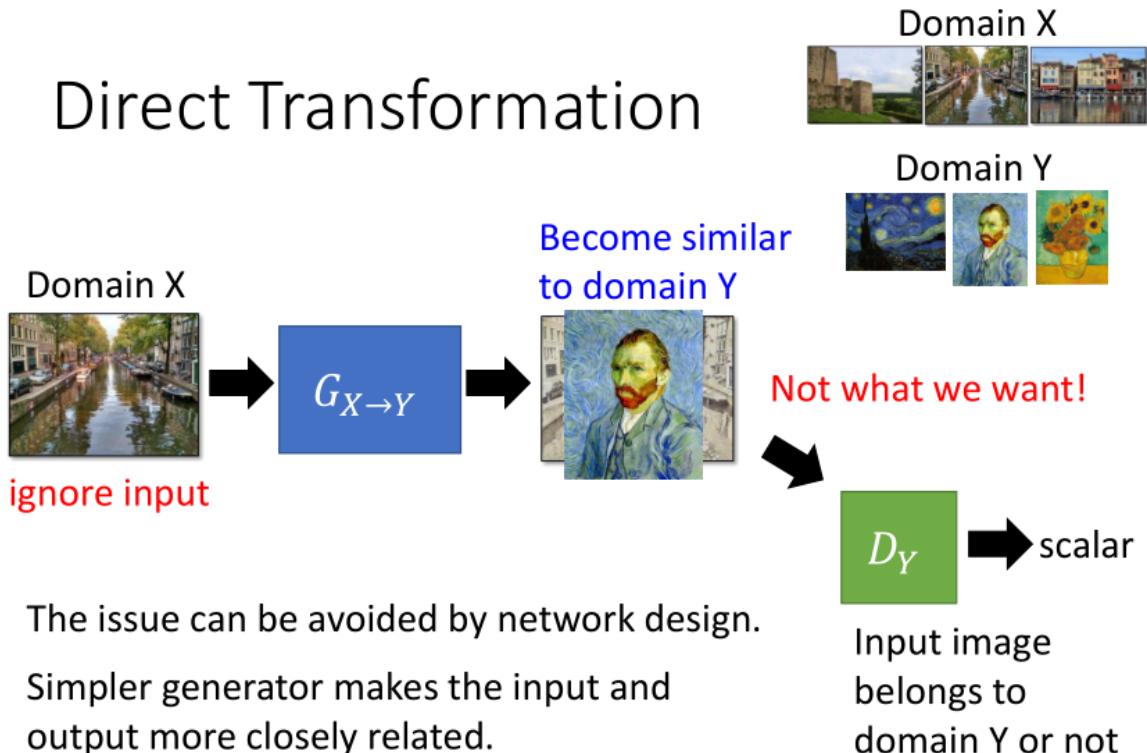


接下来 generator 要做的事情就是想办法去骗过 discriminator。如果 generator 可以产生一张 image 去骗过 discriminator，那 generator 产生的 image，就会像是一张 y domain 的 image。如果 y domain 现在是梵谷的画作，generator 产生的 output 就会像是梵谷的画作，因为 discriminator 知道梵谷的画作，长得是什么样子。

但是现在的问题是，generator 可以产生像是梵谷画作的东西，但完全可以产生一个跟 input 无关的东西。举例来说，它可能就学到画这张自画像就可以骗过 discriminator，因为这张自画像，确实很像是梵谷画的。但是这张自画像跟输入的图片完全没有任何半毛钱的关系，这个就不是我们要的。



所以我们希望 generator 不只要骗过 discriminator，generator 的输入和输出是必须有一定程度的关系的。这件事在文献上有不同的做法，我们等一下会讲 Cycle GAN，这是最知名的方法。



The issue can be avoided by network design.

Simpler generator makes the input and output more closely related.

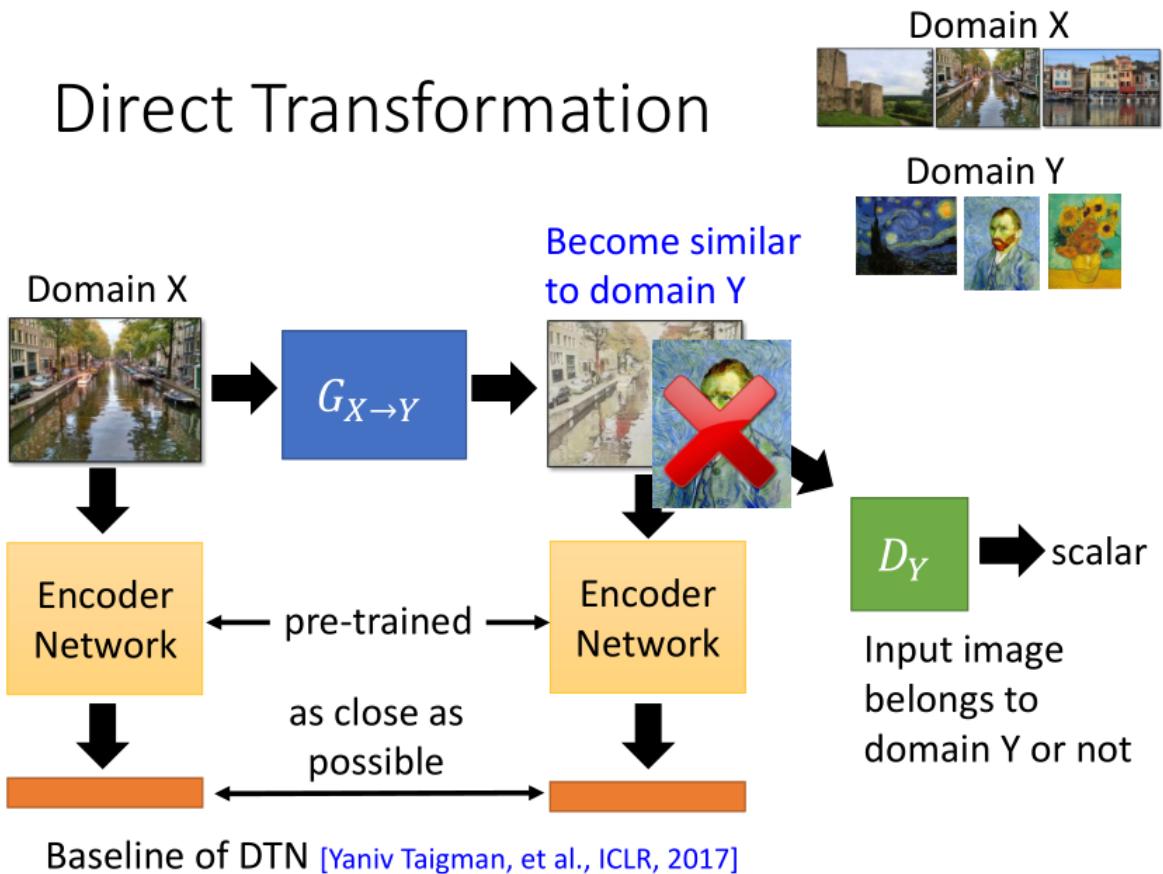
[Tomer Galanti, et al. ICLR, 2018]

第一个方法就是不要管它，这是最简单的做法，无视这个问题直接做下去，事实上有时也做得起来。无视这个问题，直接 learn 一个 generator，一个 discriminator。为什么这样子有机会可以 work 呢？因为 generator 的 input 跟 output 其实不会差太多。假设你的 generator 没有很深，那总不会 input 一张图片，然后 output 一个梵谷的自画像，这未免差太多了。

所以今天其实 generator 如果你没有特别要求它的话，它喜欢 input 就跟 output 差不多。它不太想要改太多，它希望改一点点就骗过 discriminator 就好。所以今天你直接 learn 一个这样的 generator，这样的 discriminator，不加额外的 constrain，其实也是会 work 的，你可以自己试试看。

在这个文献里面说如果今天 generator 比较 shallow，所以它 input 跟 output 会特别像，那这个时候，你就不需要做额外的 constrain，就可以把这个 generator learn 起来。那如果你 generator 很 deep，有很多层，那它就真的可以让 input output 非常不一样，这个时候，你就需要做一些额外的处理，免得让 input 跟 output 变成完全不一样的 image。

Direct Transformation

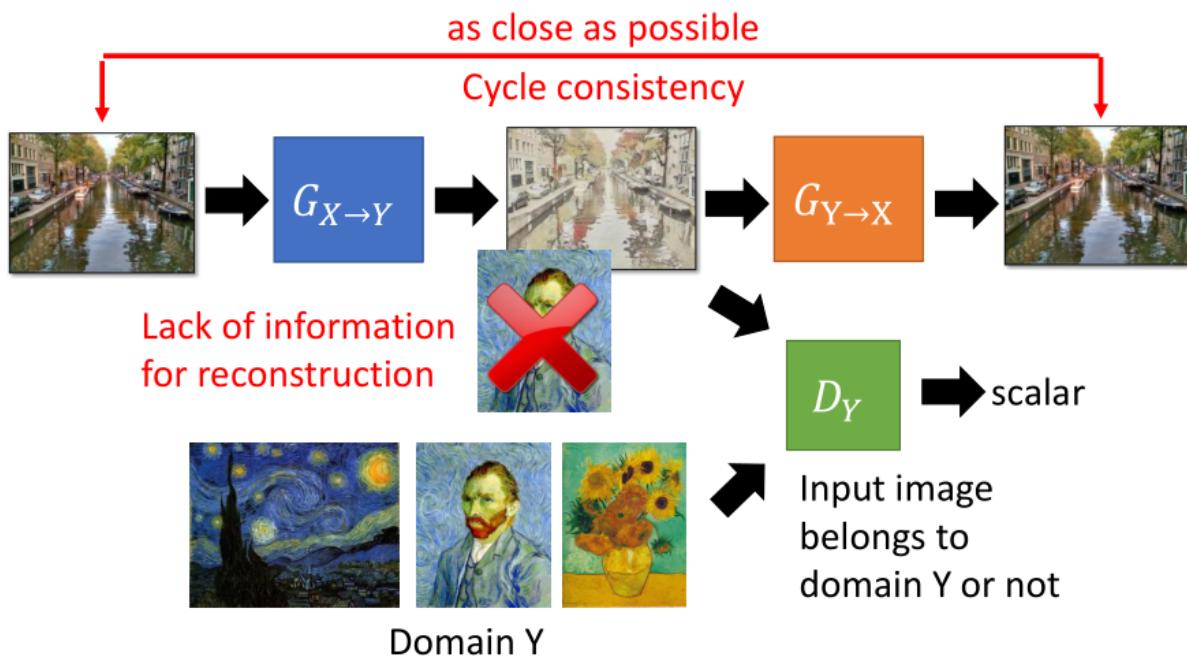


第二个方法是拿一个 pre-trained 好的 network，比如说 VGG 之类的。把这个 generator 的 input 跟 output 通通都丢给这个 pre trained 好的 network，然后 output 一个 embedding。接下来你在 train 的时候，generator 一方面会想要骗过 discriminator，让它 output 的 image 看起来像是梵谷的画作

但是同时，generator 会希望这个 pre-trained 的 model，它们 embedding 的 output 不要差太多。那这样的好处就是，因为这两个 vector 没有差太多代表说 generator 的 input 跟 output 就不会差太多。

[Jun-Yan Zhu, et al., ICCV, 2017]

Direct Transformation



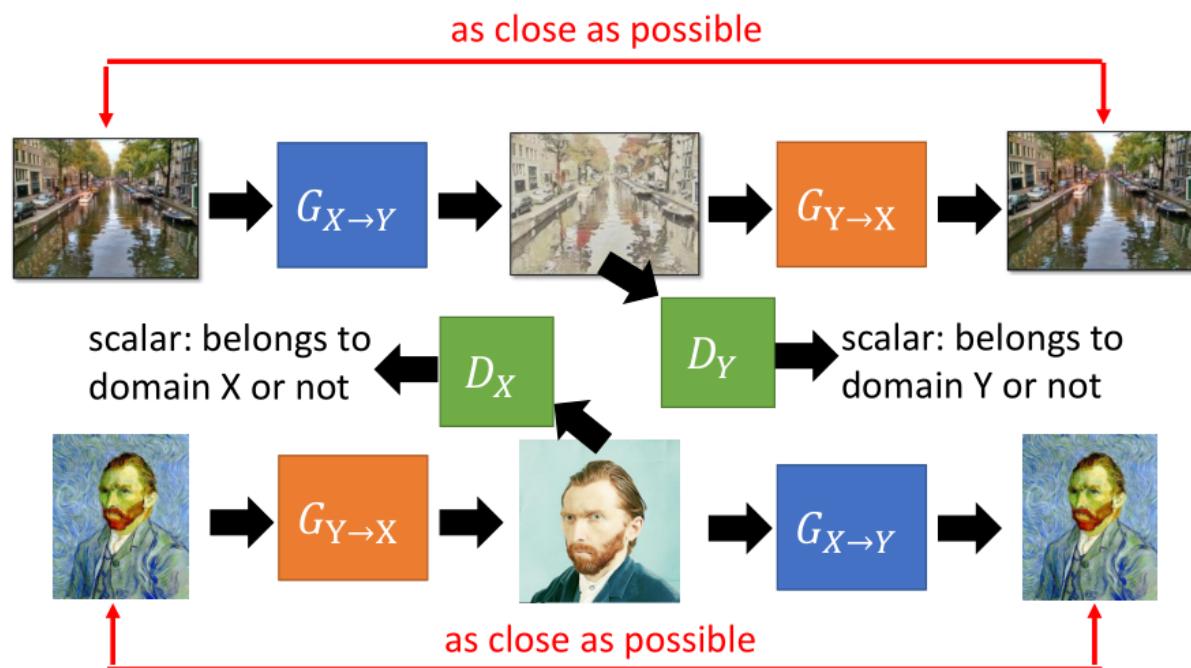
Cycle GAN

第三个做法就是大家所熟知的 Cycle GAN。在 Cycle GAN 里面，你要 train 一个 x domain 和 y domain 的 generator。它的目的是，给它一张 y domain 的图，input 一张风景画，第一个 generator 把它转成 y domain 的图，第二个 generator 要把 y domain 的图，还原回来一模一样的图。

因为现在除了要骗过 discriminator 以外，generator 要让 input 跟 output 越像越好，为了要让 input 跟 output 越像越好，你就不能在中间产生一个完全无关的图。如果你在这边产生一个梵谷的自画像，第二个 generator 就无法从梵谷的自画像还原成原来的风景画，因为它已经完全不知道原来的输入是什么了。所以这张图片，必须要保留有原来输入的资讯，那这样第二个 generator 才可以根据这张图片转回原来的 image。

这个就是 Cycle GAN，那这样 input 跟 output 越接近越好，input 一张 image 转换以后要能够转得回来，两次转换要转得回来这件事情就叫做 Cycle consistency。

Direct Transformation



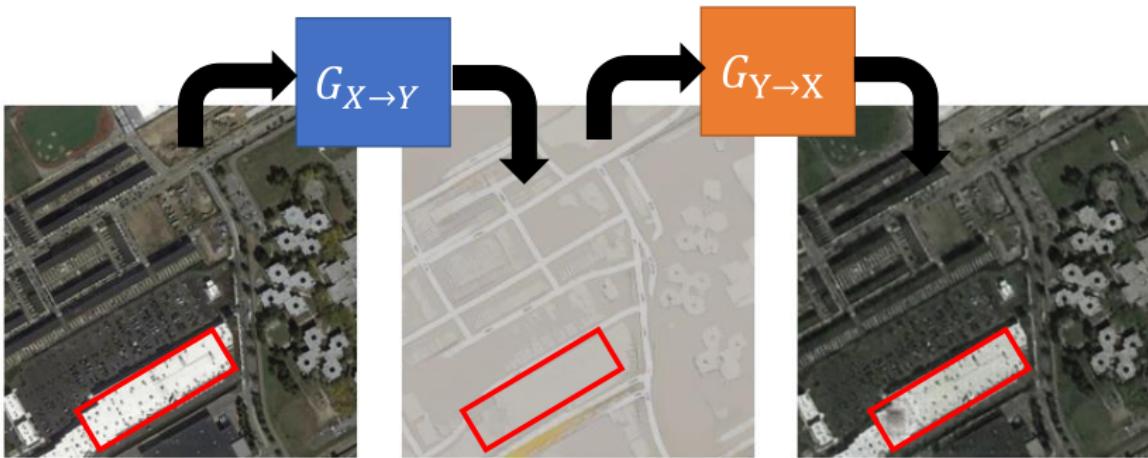
Cycle GAN 可以是双向的，本来有 x domain 转 y domain，y domain 转 x domain，再 train 另外一个 task 把 y domain 的图丢进来，然后把它转成 x domain 的图，同时你要有一个 discriminator 确保这个 generator 它 output 的图像是 x domain 的。接下来再把 x domain 的图转回原来 y domain 的图，一样希望 input 跟 output 越接近越好。这样就可以同时去 train，两个 generator 和两个 discriminator。

Issue of Cycle Consistency

其实 Cycle GAN，现在还是有一些问题是没有解决的。一个问题就是，NIPS 有一篇 paper 叫做 Cycle GAN: a master of stenography。stenography 是隐写术，就是说 Cycle GAN 会把 input 的东西藏起来，然后在 output 的时候，再呈现出来。

• CycleGAN: a Master of Steganography (隱寫術)

[Casey Chu, et al., NIPS workshop, 2017]

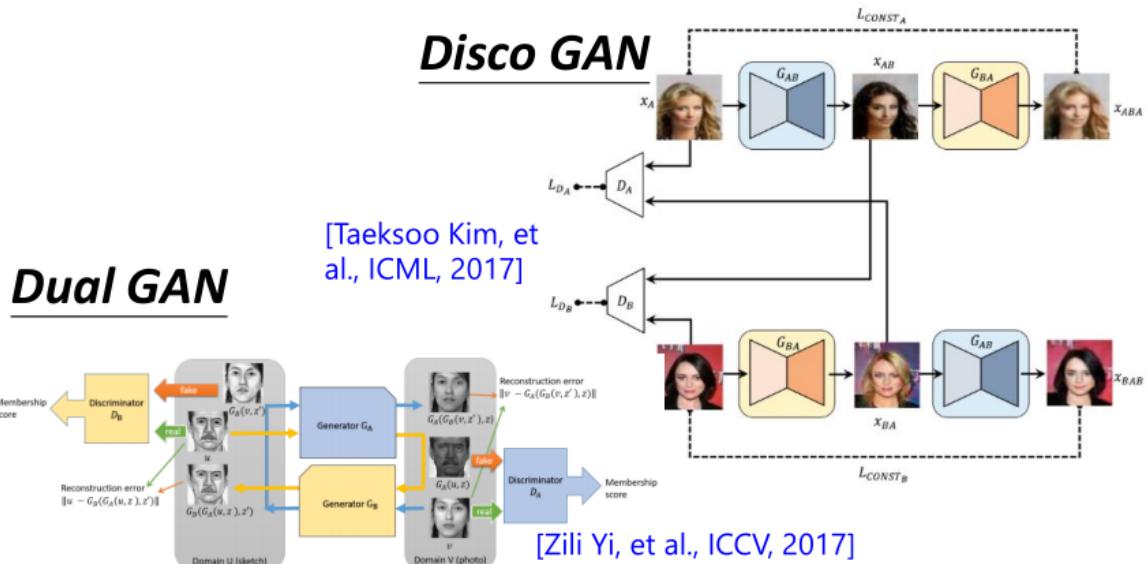


The information is hidden.

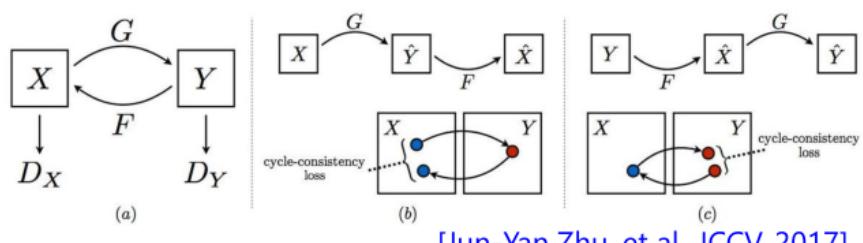
对第二个 generator 来说，如果你只看到一张图，屋顶上是没有黑点的，你是怎么知道上面应该要产生黑点的。

所以有一个可能是，Cycle GAN虽然有 Cycle consistency 的 loss 强迫你 input 跟 output 要越像越好，但是 generator 有很强的能力把资讯藏在人看不出来的地方，也就是说要如何 reconstruct 这张 image 的资讯可能是藏在这张 image 里面，让你看不出来这样，也许这个屋顶上仍然是有黑点的，只是你看不出来而已。那如果是这样子的情况，那就失去 Cycle consistency 的意义了。因为 Cycle consistency 的意义就是，第一个 generator output 跟 input 不要差太多，但如果今天 generator 很擅长藏资讯，然后再自己解回来那这个 output 就有可能跟这个 input 差距很大了。

那这个就是一个尚待研究的问题，也就是 Cycle consistency 不一定有用，machine 可能会自己学到一些方法去避开 Cycle consistency 带给你的 constrain。



Cycle GAN



在文献上除了 Cycle GAN 以外，可能还看到其它的 GAN 比如说 Dual GAN、Disco GAN。这 3 个东西没有什么不同，这些方法其实就跟 Cycle GAN 是一样的。

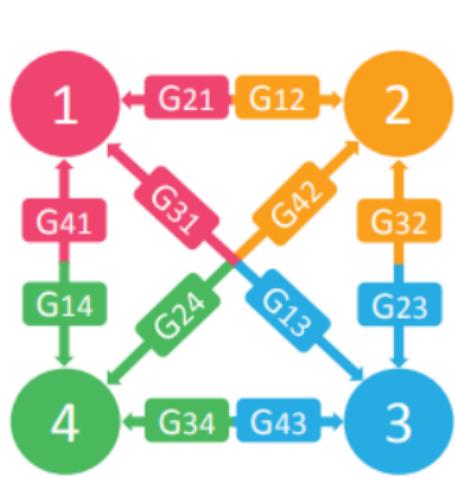
StarGAN

StarGAN

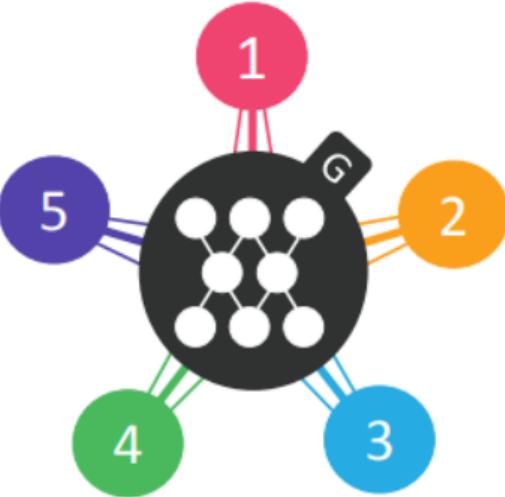
For multiple domains,
considering starGAN

[Yunjey Choi, arXiv, 2017]

(a) Cross-domain models

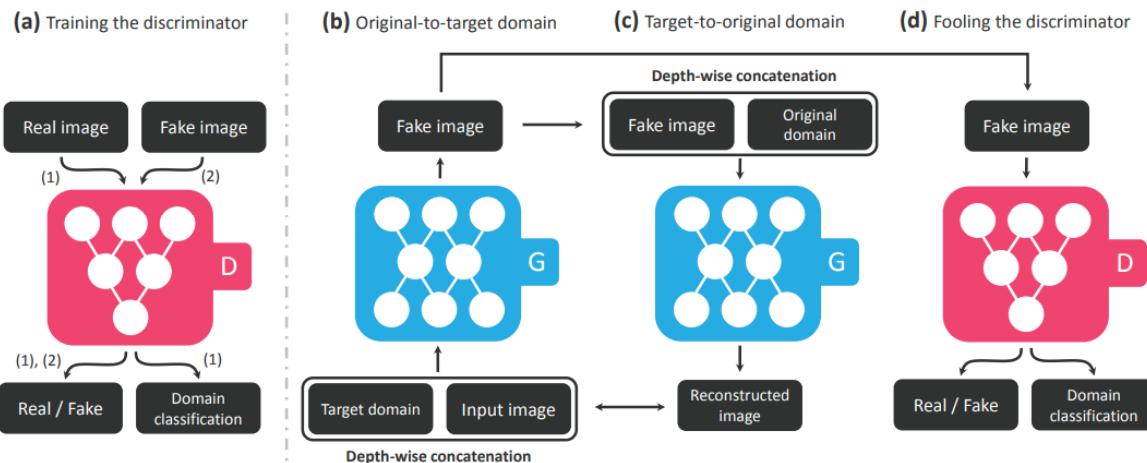


(b) StarGAN



有时候你会有一个需求是你要用多个 domain 互转。star GAN 它做的事情是，它只 learn 了一个 generator，但就可以在多个 domain 间互转。

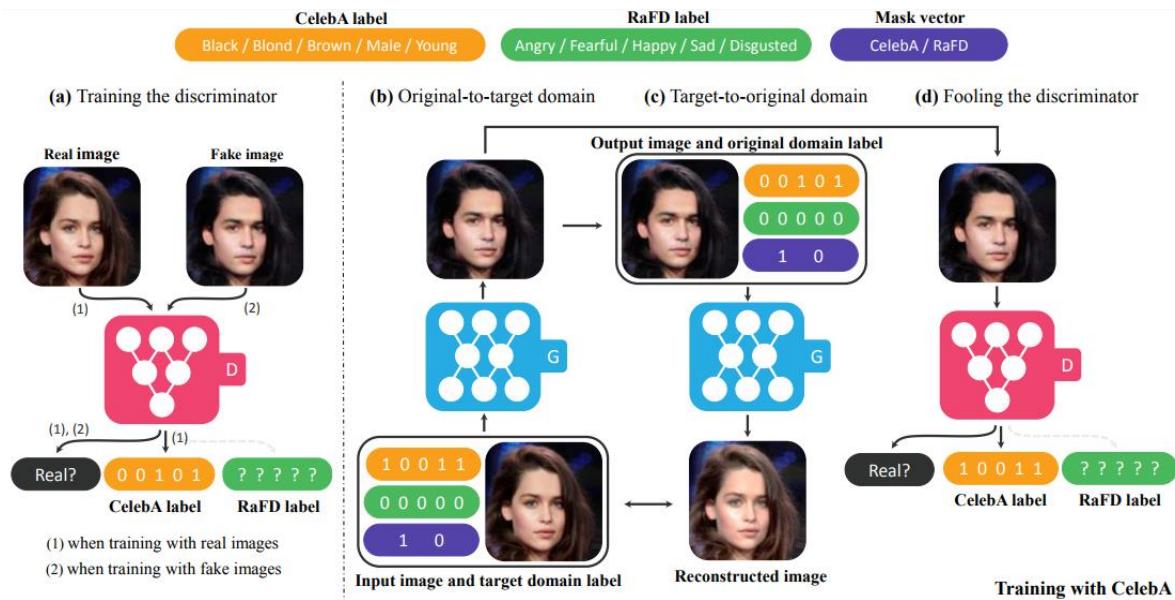
那在 star GAN 里面是怎么样呢？在 star GAN 里面你要 learn 一个 discriminator，这个 discriminator 它会做两件事：首先给它一张 image，它要鉴别说这张 image 是 real 还是 fake 的；再来它要去鉴别这一张 image 来自于哪一个 domain。



在 star GAN 里面，只需要 learn 一个 generator 就好，这个 generator 它的 input 是一张图片跟你目标的 domain。然后它根据这个 image 和目标的 domain，就把新的 image 生成出来。

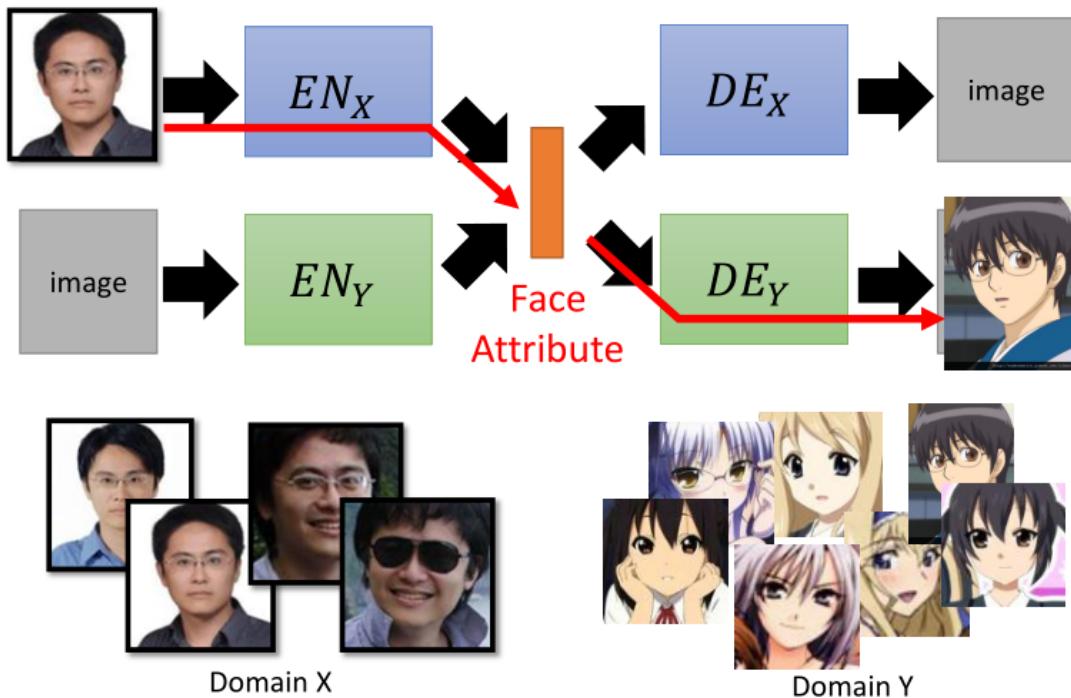
接下来再把这个同样的 image，丢给同一个 generator，把这一张被 generated 出来的 image，丢给同一个 generator，然后再告诉它原来 input 的 image 是哪一个 domain，然后再用这个 generator 合回另外一张图片，希望 input 跟 output 越接近越好。那这个东西就是 Cycle consistency 的 loss。

那这个 discriminator 做的事情就是要确认说这张转出来的 image 到底对不对。那要确认两件事，第一件事是，这张转出来的 image 看起来有没有真实；再来就是，这张转出来的 image 是不是我们要的 target domain。然后 generator 就要去想办法骗过 discriminator。



Projection to Common Space

Target



第二个做法是要把 input 的 object 用 encoder project 到一个 latent 的 space，再用 decoder 把它转换回来。

那假设有两个 domain 一个是人的 domain，一个是动画人物的 domain。想要在这两个 domain 间做互相转换的话，就用 x domain 的 encoder 抽取人的特征，用 y domain 的 encoder 抽取动画人物的特征，它们参数可能是不一样的，因为毕竟人脸和动画人物的脸还是有一些差别，所以这两个 network 不见得是一样的 network。

x domain 的 encoder 一张image，它会抽出它的 attribute。

所谓的 attribute 就是一个 latent 的 vector, input 一张 image, encoder 的 output 就是一个 latent 的 vector。

接下来把这个 latent 的 vector, 丢到 decoder 里面, 如果丢到 x domain 的 decoder, 它产生出来的是真实人物的人脸; 如果是丢到 y domain 的 decoder, 它产生出来就是二次元人物的人脸。

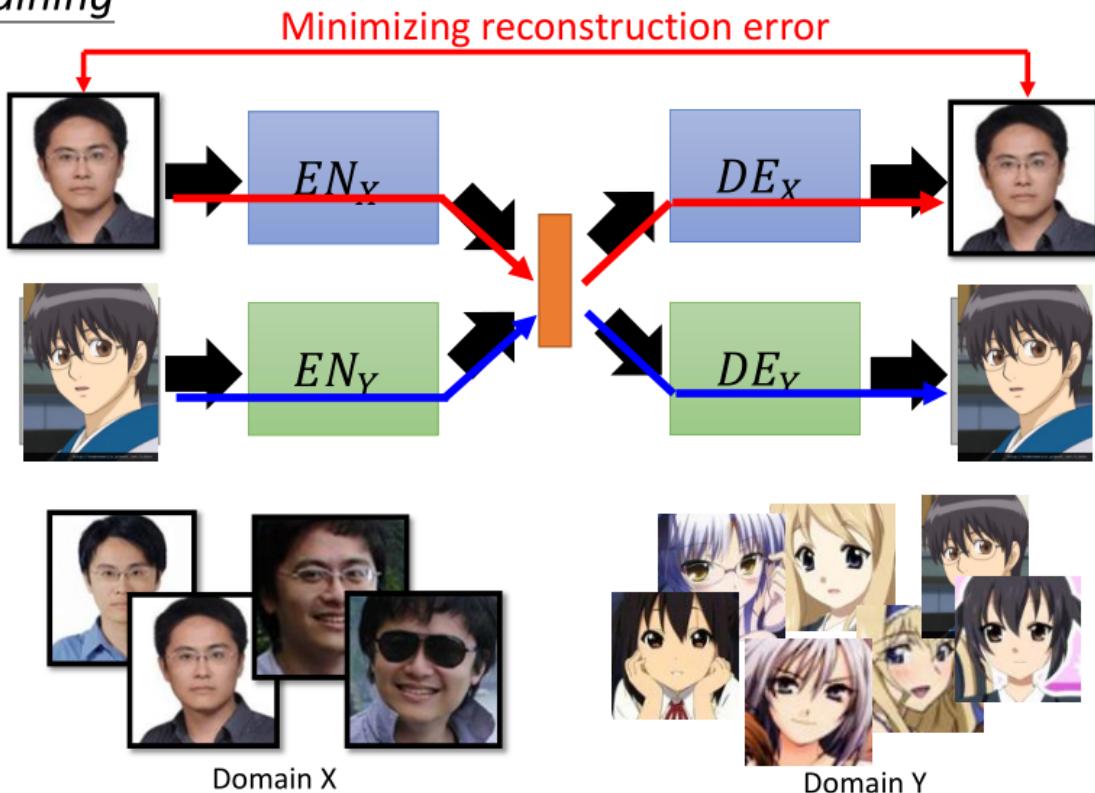
我们希望最后可以达到的结果是你给它一张真人的人脸, 通过 x domain 的 encoder 抽出 latent 的 representation。这个 latent 的 representation 是一个 vector, 我们期待说这个 vector 的每一个 dimension 就代表了 input 的这张图片的某种特征, 有没有戴眼镜, 是什么性别, 等等。

那接下来你用 y domain 的 decoder, 吃这个 vector, 根据这个 vector 里面所表示的人脸的特征合出一张 y domain 的图, 我们希望做到这一件事。

但是实际上如果我们今天有 x domain 跟 y domain 之间的对应关系, 要做到这件事非常容易, 因为就是一个 supervised learning 的问题。

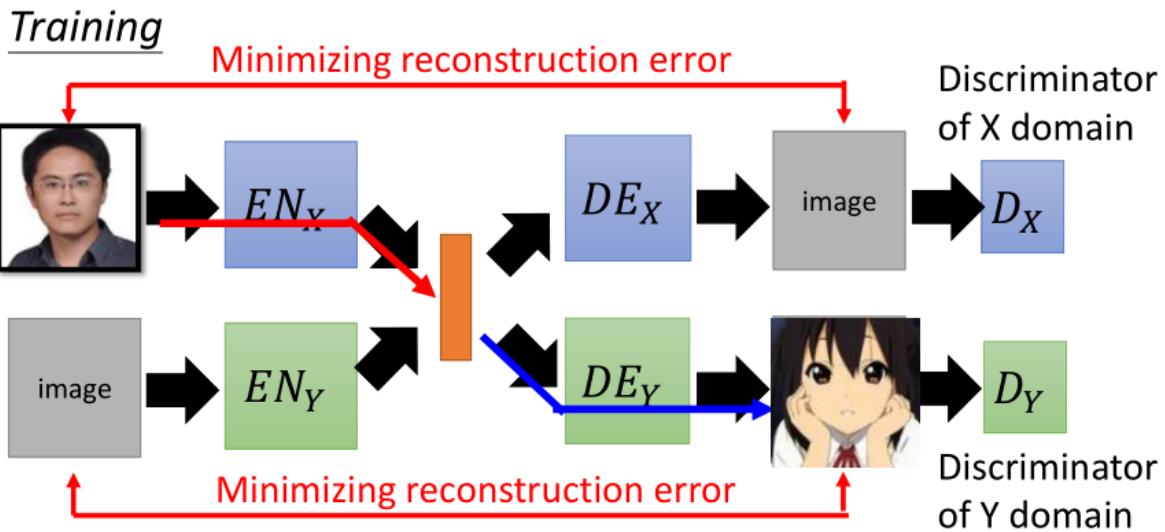
但是现在我们是一个 unsupervised learning 的问题, 只有 x domain 的 image, 跟 y domain 的 image, 它们是分开的, 那怎么 train 这些 encoder 跟这些 decoder 呢?

Training



可以组成两个 auto encoder, x domain 的 encoder 跟 x domain 的 decoder 组成一个 auto encoder, input 一张 x domain 的图让它 reconstruct 回原来 x domain 的图。y domain 的 encoder 跟 y domain 的 decoder 组成一个 auto encoder, input 一个 y domain 的图, reconstruct 回原来 y domain 的图。我们知道这两个 auto encoder 在 train 的时候它们都是要 minimize reconstruction error。

用这样的方法, 你确实可以得到 2 个 encoder, 2 个 decoder, 但是这样会造成的问题是这两个 encoder 之间是没有任何关联的。



Because we train two auto-encoders separately ...

The images with the same attribute may not project to the same position in the latent space.

还可以多做一件事情是，可以把 discriminator 加进来。你可以 train 一个 x domain 的 discriminator，强迫 decoder 的 output 看起来像是 x domain 的图。因为我们知道，假设如果只 learn auto encoder，只去 minimize reconstruction error，decoder output 的 image 会很模糊。有一个 x domain 的 discriminator 吃这一张 image，然后鉴别它是不是 x domain 的图，有一个 y domain 的 discriminator，它吃一张 image 鉴别它是不是 y domain 的图。这样你会强迫你的 x domain 的 decoder 跟 y domain 的 decoder 它们 output 的 image 都比较 realistic。

encoder decoder discriminator，它们 3 个合起来其实就是一个 VAE GAN，可以看做是用 GAN 强化 VAE，也可以看做 VAE 来强化 GAN。另外 3 个合起来其实就是另外一个 VAE GAN。

但是因为这两个 VAE GAN 它们的 training 是完全分开，各自独立的。所以实际上 train 完以后，会发现它们的 latent space 可能意思是不一样的。

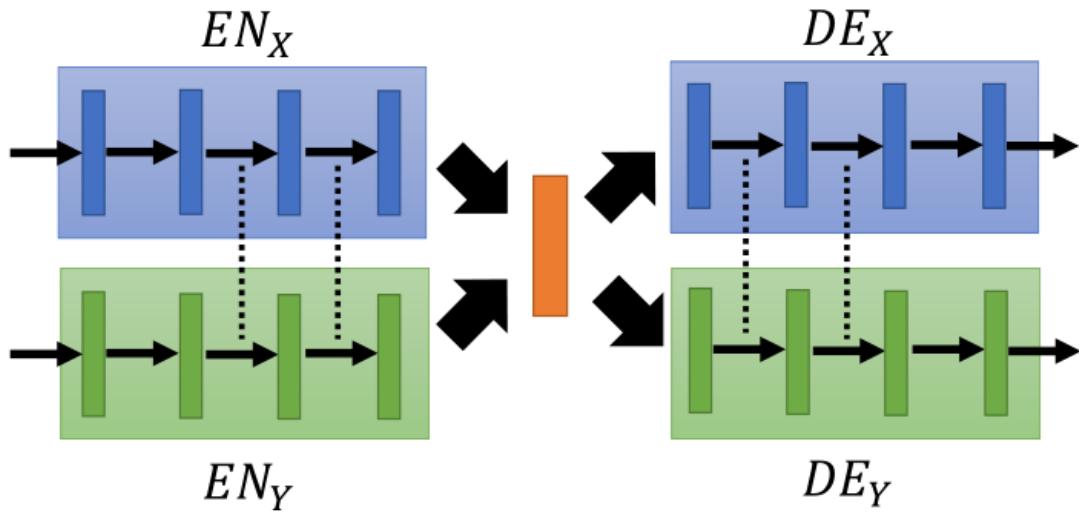
也就是说你今天丢这张人脸进去变成一个 vector，把这个 vector 丢到这张图片里面，搞不好它产生的就是一个截然不同的图片。

因为今天这两组 auto encoder 是分开 train 的，也许上面这组 auto encoder 是用这个 latent vector 的第一维代表性别，第二维代表有没有戴眼镜；下面这个是用第三维代表性别，第四维有没有戴眼镜。

如果是这样子的话，就做不起来，也就是说今天 x 这一组 encoder 跟 decoder，还有 y 这一组 encoder 跟 decoder，它们用的 language 是不一样的，它们说的语言是不一样的。

所以 x domain 的 encoder 吐出一个东西，要叫 y domain 的 decoder 吃下去，它 output 并不会跟 x domain encoder 的 input 有任何的关联性。

怎么解决这个问题？在文献上，有各式各样的解法。



Sharing the parameters of encoders and decoders

Couple GAN [[Ming-Yu Liu, et al., NIPS, 2016](#)]

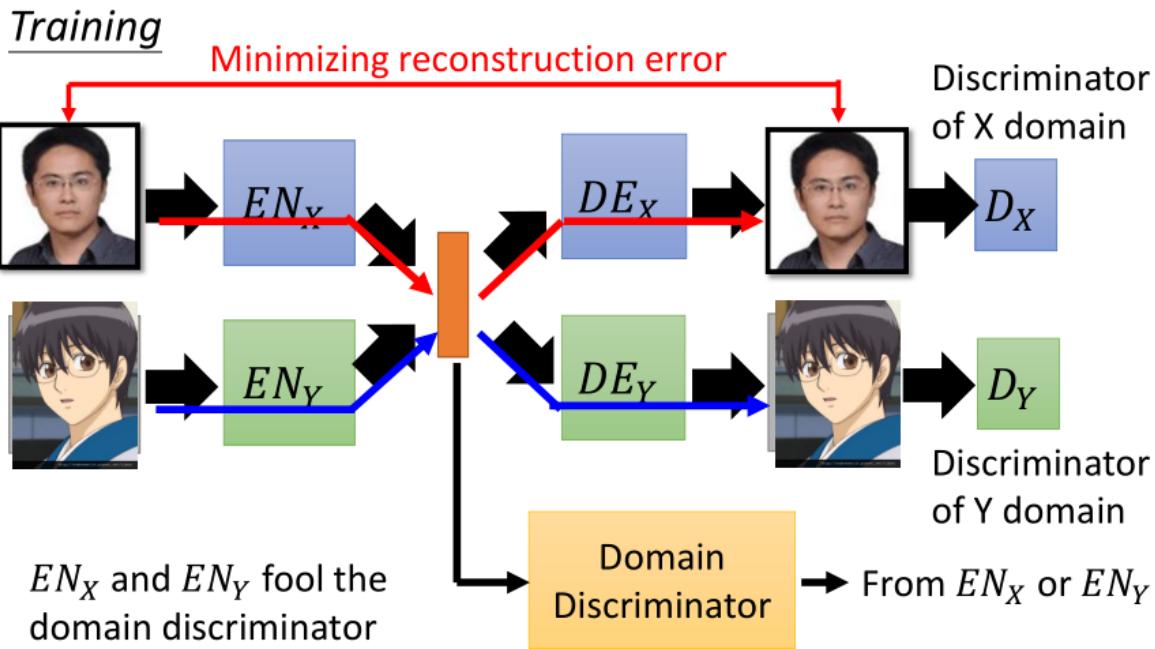
UNIT [[Ming-Yu Liu, et al., NIPS, 2017](#)]

一个常见的解法是让不同 domain 的 encoder 跟 decoder，它们的参数被 tie 在一起。

我们知道 encoder 有好几个 hidden layer, x domain encoder 有好几个 hidden layer, y domain encoder 也有好几个 hidden layer。你希望它们最后的几个 hidden layer 参数是共用的，它们**共用同一组参数**。可能前面几个 layer 是不一样的，但最后的几个 layer，必须是共用的，或者两个不同 domain 的 decoder，它们前面几个 layer 是共用的，后面几个 layer 是不一样。

那这样的好处是什么？因为它们最后几个 hidden layer 是共用的，也许因为透过最后几个 hidden layer 是共用这件事会让这两个 encoder 把 image 压到同样的 latent space 的时候，它们的 latent space 是同一个 latent space，它们的 latent space 会用同样的 dimension 来表示同样的人脸特征。这样的技术，被用在 couple GAN 跟 UNIT 里面。

像这种 share 参数的 task，它最极端的状况就是，这两个 encoder 共用同一组参数，就是同一个 encoder。只是在使用的时候吃一个 flag，代表现在要 encoder 的 image 是来自于 x domain 还是来自于 y domain。



The domain discriminator forces the output of EN_X and EN_Y have the same distribution. [Guillaume Lample, et al., NIPS, 2017]

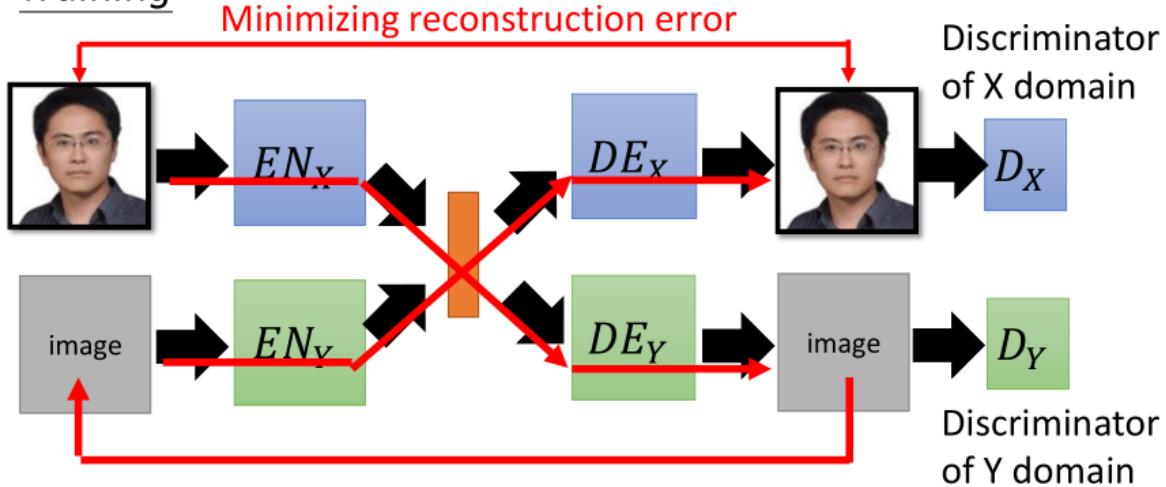
还有满坑满谷的招式。比如说加一个 **domain 的 discriminator**。这个概念跟domain adversarial training 是一样的，其实是一样的。

它的概念是：原来 x domain 跟 y domain 都是自己搞自己的东西，但我们现在再加一个 domain discriminator，这个 domain discriminator 要做的事情是给它这个 latent 的 vector，它去判断说这个 vector 是来自于 x domain 的 image，还是来自于 y domain 的 image。然后 x domain encoder 跟 y domain encoder，它们的工作就是想要去骗过这个 domain 的 discriminator，让 domain discriminator 没办法凭借这个 vector 就判断它是来自于 x domain 还是来自于 y domain。如果今天 domain 的 discriminator 无法判断这个 vector 是来自于 x domain 和 y domain，意味着，两个 domain 的 image 变成 code 的时候，它们的 distribution 都是一样的。因为它们的 distribution 是一样的，也许我们就可以期待同样的维度就代表了同样的意思。举例来说假设真人的照片男女比例是 1:1，动画人物的照片，男女比例也是 1:1。因为男女的比例都是 1:1，最后如果你要让两个 domain 的 feature，它的 distribution 一样，那你就要用同一个维度来存这个男女比例是 1:1 的 feature，如果是性别都用第一维来存，这样它们的 distribution 才会变得一样。

所以假设你今天的这两个 domain 它们 attribute 的 distribution 是一样的

比如说，男女的比例是一样的，有戴眼镜跟没戴眼镜的比例是一样的，长发短发比例是一样的。那你也期待说，通过 domain discriminator 强迫这两个 domain 的 embedding latent feature 是一样的时候，那它们就会用同样的 dimension 来表示同样的事情，来表示同样的 characteristic。

Training



Cycle Consistency:

Used in ComboGAN [Asha Anoosheh, et al., arXiv, 017]

还有其它的招数。举例来说，你也可以用 **Cycle consistency**

那如果把这个技术来跟 Cycle GAN 来做比较的话，Cycle GAN 就是有两个 transformation 的 network，这个跟 Cycle GAN 的 training 其实就是一模一样的。

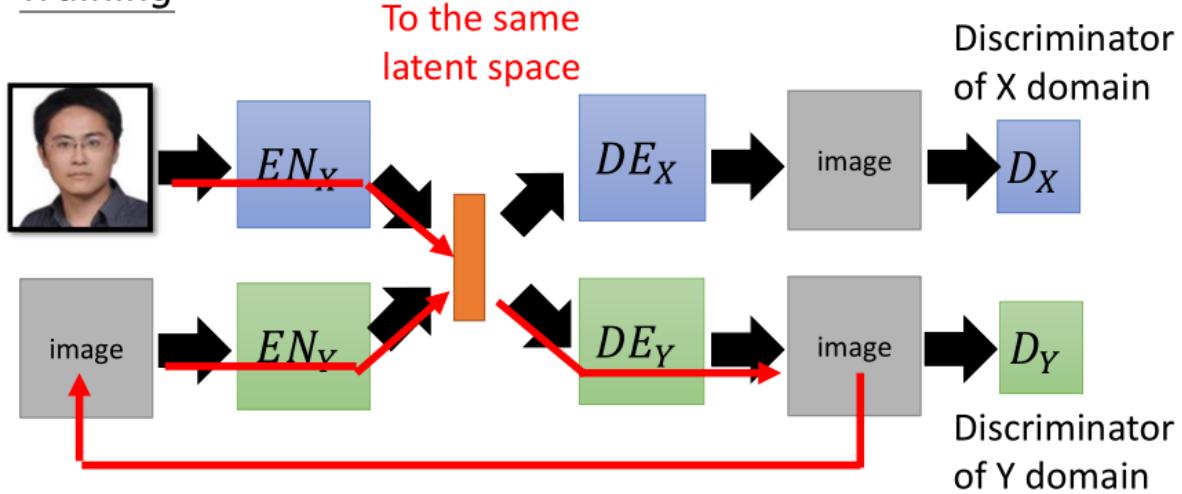
x domain 的 encoder 加 y domain 的 decoder，它们合起来就是从 x domain 转到 y domain，然后有一个 discriminator 确定这个 image 看起来像不像是 y domain 的 image。

接下来，再把这张 image，通过 y domain 的 encoder 跟 x domain 的 decoder，转回原来的 image，希望 input 的 image 跟 output 的 image 越接近越好。

只是原来在 Cycle GAN 里面，我们说从 x domain 到 y domain generator 就是一个 network，我们没有把它特别切成 encoder 跟 decoder，在这边，我们会把它切成把 x domain 到 y domain 的 network 切成一个 x domain 的 encoder 和一个 y domain 的 decoder。从 y domain 到 x domain 的 network，我们说它有一个 y domain 的 encoder，一个 x domain 的 decoder。network 的架构不太一样，然后中间的那个 latent space 是 shared。但是实际上它们是 training 的 criteria，其实是一样的。

这样的技术，就用在 Combo GAN 里面。

Training



Semantic Consistency:

Used in DTN [[Yaniv Taigman, et al., ICLR, 2017](#)] and
XGAN [[Amélie Royer, et al., arXiv, 2017](#)]

还有一个叫做 **semantic consistency**

你把一张图片丢进来，然后把它变成 code，然后接下来，把这个 code 用 y domain 的 decoder 把它转回来。再把 y domain 的 image 丢到 y domain 的 encoder，希望通过 x domain encoder 的 encode 跟 y domain encoder 的 encode，它们的 code 要越接近越好。

那这样的好处是说，我们本来在做 Cycle consistency 的时候，你算的是两个 image 之间的 similarity。那如果是 image 和 image 之间的 similarity，通常算的是 pixel wise 的 similarity，不会考虑 semantic，而是看它们表象上像不像。如果是在这个 latent 的 space 上面考虑的话，你就是在算它们的 semantic 像不像，算它们的 latent 的 code 像不像。

这个技术被用在 XGAN 里面。

Voice Conversion

也可以做 voice conversion，就是把 A 的声音，转成 B 的声音。

过去的 voice conversion，就是要收集两个人的声音，假如你要把 A 的声音，转成 B 的声音，你就要把 A 找来念 50 句话，B 找来也念 50 句话，让它们念一样的句子，接下来 learn 一个 model，比如说 sequence to sequence model 或是其它，吃一个 A 的声音，然后转成 B 的声音就结束了，这就是一个 supervised learning problem。

若用 GAN 的技术，就我们用今天学到的那些技术在两堆声音间互转。只需要收集 speaker A 的声音，再收集 speaker B 的声音，它们两个甚至可以说的就是不同的语言，用我们刚才讲的 Cycle consistency，把 A 的声音转成 B 的声音。

为什么不做 TTS(Text To Speech)呢？voice conversion 可以保留原来说话人的情绪语调。

Theory behind GAN

我们已经讲了 GAN 的直观的想法，今天要来讲 GAN 背后的理论。

要讲的是 2014 年 Ian Goodfellow 在 propose GAN 的时候它的讲法，等一下可以看看跟我们讲的 GAN 的直观的想法里面有没有矛盾的地方。其实是有一些地方还颇矛盾的，至今仍然没有好的 solution、好的手法可以解决。

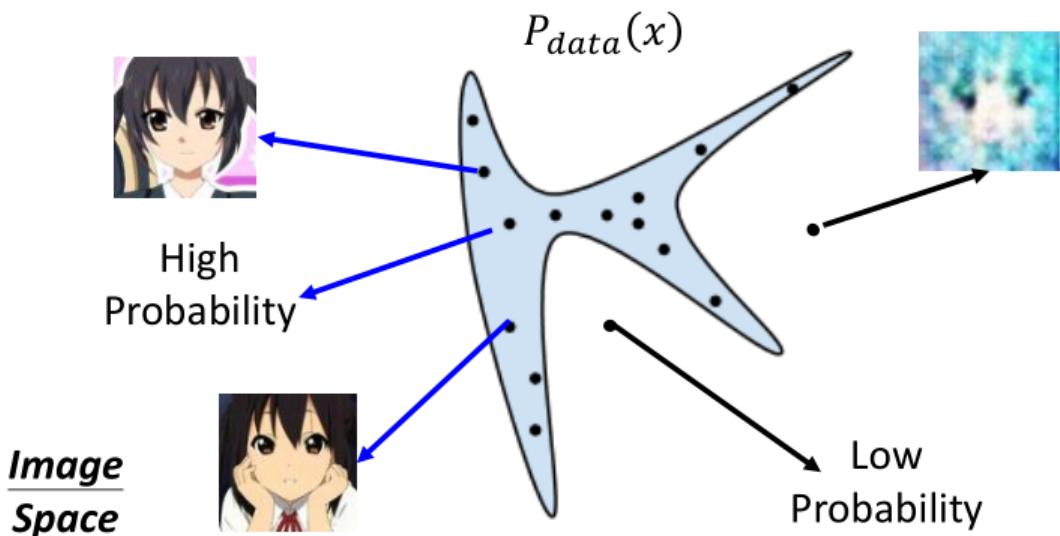
GAN 要做的就是根据很多 example 自己去进行生成。所谓的生成到底是什么样的问题？假设要生成的东西是 image，用 x 来代表一张 image，每一个 image 都是 high dimensional 高维空间中的一个点。假设产生 64×64 的 image，它是 64×64 维空间中的一个点。

这边为了画图方便假设每一个 x 就是二维空间中的一个点，虽然实际上它是高维空间中的一个点。

Generation

x : an image (a high-dimensional vector)

- We want to find data distribution $P_{data}(x)$



现在要产生的东西比如说要产生 image，它其实有一个固定的 distribution 写成 $P_{data}(x)$ 。在这整个 image 的 space 里面，在这整个 image 所构成的高维空间中只有非常少的部分、一小部分 sample 出来的 image 看起来像是人脸，在多数的空间中 sample 出来的 image 都不像是人脸。

假设生成的 x 是人脸的话，它有一个固定的 distribution，这个 distribution 在蓝色的这个区域，它的机率是高的，在蓝色的区域以外，它的机率是低的。

我们要机器找出这个 distribution，而这个 distribution 到底长什么样子，实际上是不知道的。可以搜集很多的 data 知道 x 可能在某些地方分布比较高，但是要我们把这个式子找出来我们是不知道要怎么做的。

现在 GAN 做的是一个 generative model 做的事情，就是要找出这个 distribution。

Maximum Likelihood Estimation

在有 GAN 之前怎么做 generative？我们是用 Maximum Likelihood Estimation。

现在有一个 data 的 distribution 它是 $P_{data}(x)$ ，这个 distribution 它的 formulation 长什么样子我们是不知道的。我们可以从这个 distribution sample 它，假设做二次元人物的生成，就是从 database 里面 sample 出 image。

接下来我们要自己去找一个 distribution，这个 distribution 写成 $P_G(x; \theta)$ 。这个 distribution 是由一组参数 θ 所操控的。由 θ 所操控的意思是这个 distribution 假设它是一个 Gaussian Mixture Model，这个 θ 指的就是 Gaussian 的 mean 跟 variance。我们要去调整 Gaussian 的 mean 跟 variance，使得我们得到的这个 distribution $P_G(x; \theta)$ 跟真实的 distribution $P_{data}(x)$ 越接近越好。

虽然我们不知道 $P_{data}(x)$ 长什么样子，我们只能够从 $P_{data}(x)$ 里面去 sample，但是我们希望 $P_G(x; \theta)$ 可以找一个 θ 让 $P_G(x; \theta)$ 跟 $P_{data}(x)$ 越接近越好。

怎么做？

首先可以从 $P_{data}(x)$ sample 一些东西出来

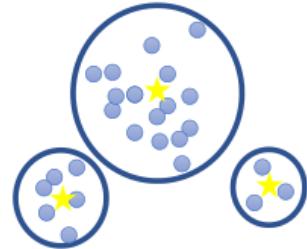
- Given a data distribution $P_{data}(x)$ (We can sample from it.)
- We have a distribution $P_G(x; \theta)$ parameterized by θ
 - We want to find θ such that $P_G(x; \theta)$ close to $P_{data}(x)$
 - E.g. $P_G(x; \theta)$ is a Gaussian Mixture Model, θ are means and variances of the Gaussians

Sample $\{x^1, x^2, \dots, x^m\}$ from $P_{data}(x)$

We can compute $P_G(x^i; \theta)$

Likelihood of generating the samples

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



Find θ^* maximizing the likelihood

接下来对每一个 sample 出来的 x^i ，我们都可以计算它的 likelihood

假设给定一组参数 θ ，我们就知道 P_G 这个 probability 的 distribution 长什么样子，我们就可以计算从这个 distribution 里面sample 出某一个 x^i 的机率，可以计算这个 likelihood。

接下来要做的就是，我们要找出一个 θ ，使得 P_G 跟 $P_{data}(x)$ 越接近越好。

我们希望这些从 $P_{data}(x)$ 里面 sample 出来的 example，如果是用 P_G 这个 distribution 来产生的话，它的 likelihood 越大越好。把所有的机率乘起来就得到 total 的 likelihood，我们希望 total likelihood 越大越好。

就是要去找一个 θ^* ，找一组最佳的参数，它可以去 maximize L 这个值。

Minimize KL Divergence

$$\begin{aligned}
\theta^* &= \arg \max_{\theta} \prod_{i=1}^m P_G(x^i; \theta) = \arg \max_{\theta} \log \prod_{i=1}^m P_G(x^i; \theta) \\
&= \arg \max_{\theta} \sum_{i=1}^m \log P_G(x^i; \theta) \quad \{x^1, x^2, \dots, x^m\} \text{ from } P_{data}(x) \\
&\approx \arg \max_{\theta} E_{x \sim P_{data}} [\log P_G(x; \theta)] \\
&= \arg \max_{\theta} \int_x P_{data}(x) \log P_G(x; \theta) dx - \int_x P_{data}(x) \log P_{data}(x) dx \\
&= \arg \min_{\theta} KL(P_{data} || P_G) \quad \text{How to define a general } P_G?
\end{aligned}$$

Maximum Likelihood 的另外一个解释是：Maximum Likelihood Estimation = Minimize KL Divergence。

这个式子可以稍微改变一下，取一个 log，把 log 放进去，变成 summation over 第一笔 example 到第 m 笔 example。

Suppose we sample N of these $x \sim P_{data}$. Then, the Law of Large Number $\bar{X}_n = \frac{1}{n}(X_1 + \dots + X_n)$ says that as N goes to infinity:

$$\frac{1}{N} \sum_i^N \log P_G(x^i | \theta) = \mathbb{E}_{x \sim P_{data}} [\log P(x | \theta)]$$

这件事情其实就是在 approximate 从 $P_{data}(x)$ 这个 distribution 里面 sample x 出来，maximize $\log P_G(x)$ 的 expected value，expectation distribution 是你要 sample 的 data

接下来可以把 expectation 这一项展开，就是一个积分

加一项看起来没有什么用的东西，在后面加这么一项，里面只有 $P_{data}(x)$ ，跟 P_G 是完全没有任何关系的。所以加这一项根本不会影响你找出来的最大的 x。

那为什么要加这一项？目的是为了告诉你 Maximum Likelihood 它就是 KL Divergence。把式子做一下整理，这个式子它就是 $P_{data}(x)$ 跟 P_G 的 KL Divergence。

所以找一个 θ 去 maximize likelihood，等同于找一个 θ 去 minimize $P_{data}(x)$ 跟 P_G 的 KL Divergence。

在机器学习里面讲的所谓 Maximum Likelihood，我们要找一个 Generative Model 去 Maximum Likelihood，Maximum Likelihood 这件事情就等同于 minimize 你的 Generative Model 所定义的 distribution P_G 跟现在的 data $P_{data}(x)$ 之间的 KL Divergence。

Generative Adversarial Network

接下来会遇到的问题是：假设我们的 P_G 只是一个 Gaussian Mixture Model 显然有非常多的限制，我们希望 P_G 是一个 general 的 distribution。但假设把 P_G 换成比 Gaussian 更复杂的东西，会遇到的问题就是算不出你的 likelihood，算不出 $P_G(x; \theta)$ 这一项。

它可能是一个 Neural Network，你就没有办法计算它的 likelihood，所以就有了一些新的想法。

让 machine 自动的生成东西比如说做 image generation 从来都不是新的题目，你可能看最近用 GAN 做了很多 image generation 的 task，好像 image generation 是这几年才有的东西。其实不是，image generation 在八零年代就有人做过了。

那个时候的作法是用 Gaussian Mixture Model，搜集很多很多的 image，每一个 image 就是高维空间中中间一个 data point，就可以 learn 一个 Gaussian Mixture Model 去 maximize 产生那些 image likelihood。

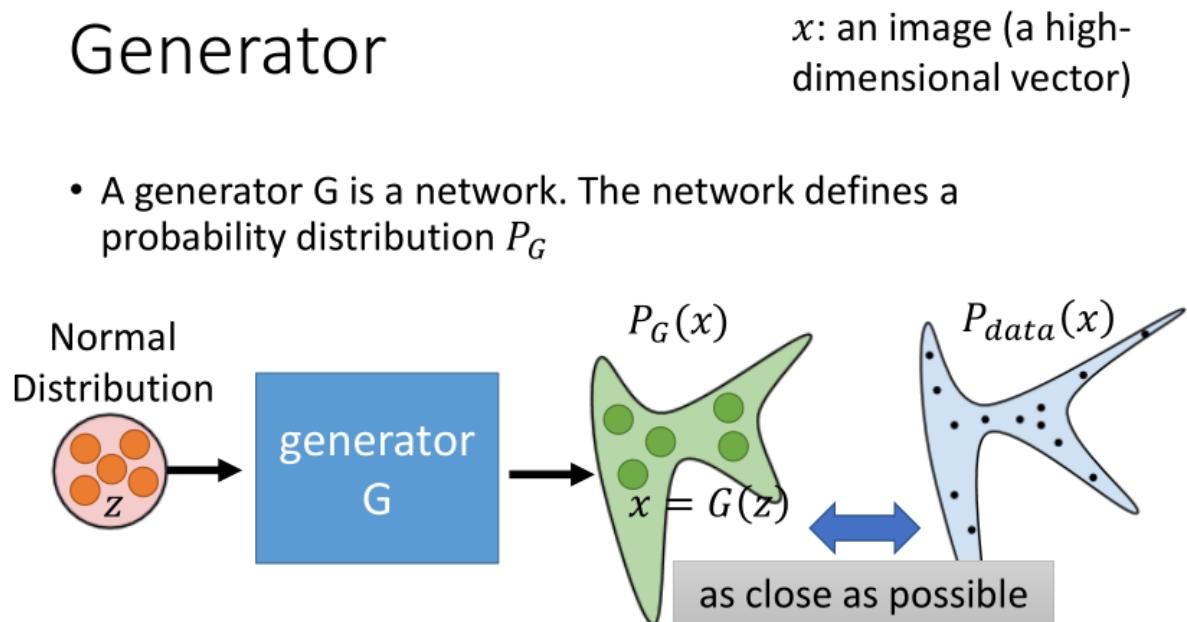
但如果你看古圣先贤留下来的文献的话，就会发现如果用 Gaussian Mixture Model 产生出来的 image，非常非常的糊。

这个可能原因是因为 image 它是高维空间中一个 manifold。image 其实是高维空间中一个低维的 manifold。

所以如果用 Gaussian Mixture Model，它其实就不是一个 manifold。用 Gaussian Mixture Model 不管怎么调 mean 跟 variance，它就不像是你的 target distribution，所以怎么做都是做不好。

所以需要用更 generalize 的方式来 learn generation 这件事情。

Generator



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

Divergence between distributions P_G and P_{data}
How to compute the divergence?

在 Generative Adversarial Network 里面，generator 就是一个 network。

我们都知道 network 就是一个东西然后 output 一个东西。举例来说，input 从某一个 distribution sample 出来的 noise z ，input 一个随机的 vector z ，然后它就会 output 一个 x 。

如果 generator G 看作是一个 function 的话，这个 x 就是 $G(z)$ 。如果是做 image generation 的话，那你的 x 就是一个 image。

我们说这个 z 是从某一个 prior distribution，比如说是从一个 normal distribution sample 出来的，sample 出来的 z 通通通过 G 得到另外一大堆 sample，把这些 sample 通通集合起来得到的就会是另外一个 distribution。

虽然 input 是一个 normal distribution 是一个单纯的 Gaussian Distribution，但是通过 generator 以后，因为这个 generator 是一个 network，它可以把这个 z 通过一个非常复杂的转换把它变成 x ，所以把通过 generator 产生的 x 通通集合起来，它可以是一个非常复杂的 distribution。而这个 distribution 就是我们所谓的 P_G 。

有人可能会问这个 Prior Distribution 应该要设成什么样子。文献上有人会用 Normal Distribution，有人会用 Uniform Distribution。我觉得这边其实 Prior Distribution 用哪种 distribution 也许影响并没有那么大。

因为 generator 它是一个 network。一个 hidden layer 的 network 它就可以 approximate 任何 function，更何况是有多个 hidden layer 的 network，它可以 approximate 非常复杂的 function。

所以就算是 input distribution 是一个非常简单的 distribution，通过了这个 network 以后，它也可以把这个简单的 distribution 凹成各式各样不同的形状，所以不用担心这个 input 是一个 normal distribution 会对 output 来说有很大的限制。

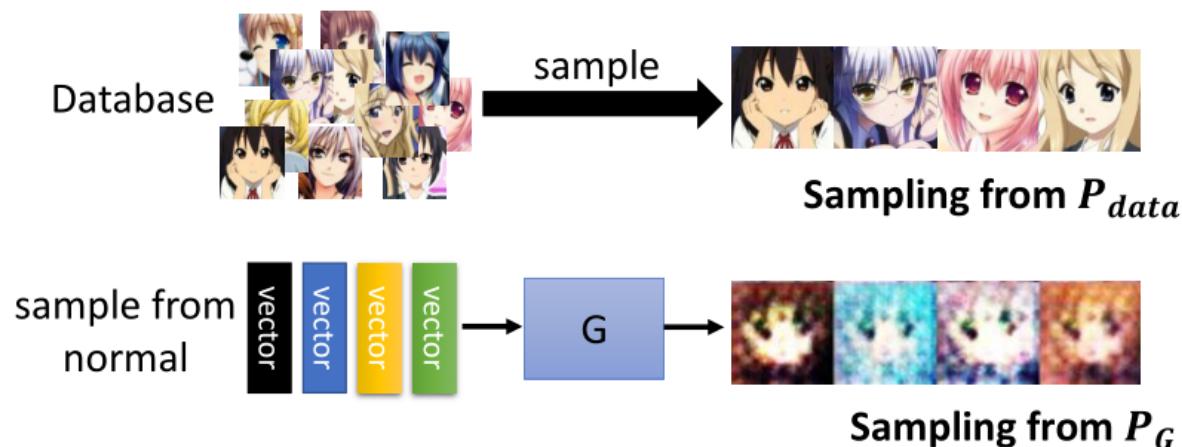
接下来目标是希望根据这个 generator 所定义出来的 distribution P_G 跟我们的 data 的 distribution $P_{data}(x)$ 越接近越好。

Discriminator

Discriminator

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

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



如果要写一个 Optimization Formulation 的话，这个 formulation 看起来是这个样子。

我们要找一个 generator G，这个 generator 可以让它所定义出来的 distribution P_G 跟我们的 data $P_{data}(x)$ 之间的某种 divergence 越小越好。

举例来说如果是 Maximum Likelihood 的话它就是要 minimize KL Divergence。在 GAN 里面 minimize 的不是 KL Divergence 而是其它的 Divergence。这边写一个 Div 就代表反正它是某一种 Divergence。

假设能够计算这个 Divergence，要找一个 G 去 minimize 这个 Divergence，那就用 Gradient Descent 就可以做了。但问题是要是怎么计算出这个 Divergence？

$P_{data}(x)$ 的 formulation 我们是不知道的。它并不是什么 Gaussian Distribution。 P_G 的 formulation 我们也是不知道的。

假设 P_G 跟 $P_{data}(x)$ 它的 formulation 我们是知道的，我们代进 Divergence 的 formulation 里面就可以算出它的 Divergence 是多少，就可以用 Gradient Descent 去 minimize 它的 Divergence。

问题就是 P_G 跟 $P_{data}(x)$ 它的 formulation 我们是不知道的，我们根本就不知道要怎么去计算它的 Divergence。所以根本不知道要怎么找一个 G 去 minimize 它的 Divergence。

这个就是 GAN 神奇的地方。在进入比较多的数学式之前我们先很直观的讲一下，GAN 到底怎么做到 minimize Divergence 这件事情。

这边的前提是我们不知道 P_G 跟 $P_{data}(x)$ 的 distribution 长什么样子，但是我们可以从这两个 distribution 里面 sample data 出来

从 $P_{data}(x)$ 去 sample distribution 出来就是把你的 database 拿出来
然后从里面 sample 很多 image 出来。

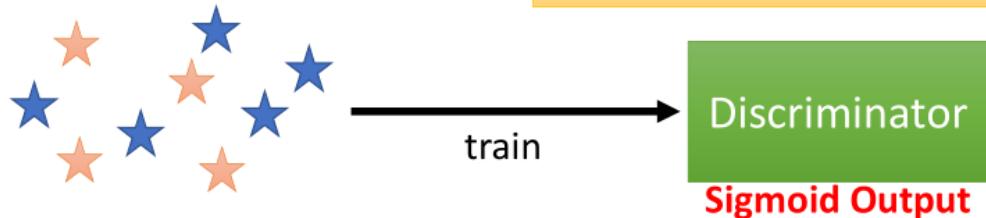
从 P_G 里面做 sample 其实就是 random sample 一个 vector，把这个 vector 丢到 generator 里面产生一张 image，这个就是从 P_G 里面做 sample。

我们可以从 P_G 和 $P_{data}(x)$ 做 sample，根据这个 sample 我们要怎么知道这两个 distribution 的 Divergence 呢？

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

- ★ : data sampled from P_{data}
- ☆ : data sampled from P_G

Using the example objective function is exactly the same as training a binary classifier.



Example Objective Function for D

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

↑
(G is fixed)

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

[Goodfellow, et al., NIPS, 2014]

The maximum objective value is related to JS divergence.

GAN 神奇的地方就是通过 discriminator，我们可以量这两个 distribution 间的 Divergence。假设蓝色的星星是从 $P_{data}(x)$ 里面 sample 出来的东西，红色的星星是从 P_G sample 出来的东西。根据这些 data 我们去训练一个 discriminator，上周我们已经讲过训练 discriminator 意思就是给 $P_{data}(x)$ 的分数越大越好，给 P_G 的分数越小越好。这个训练的结果就会告诉我们 $P_{data}(x)$ 跟 P_G 它们之间的 Divergence 有多大。

我们怎么训练 discriminator 呢，我们会写一个 Objective Function D，这个 Objective Function 它跟两项有关，一个是跟 generator 有关，一个是跟 discriminator 有关。

在 train discriminator 的时候我们会 fix 住 generator，所以 G 这一项是 fix 住的，公式的意思是 x 是从 $P_{data}(x)$ 里面 sample 出来的，我们希望 $\log D(x)$ 越大越好，也就是我们希望 discriminator 的 output，假设 x 是从 $P_{data}(x)$ 里面 sample 出来的，我们就希望 $D(x)$ 越大越好。

反之假设 x 是从 generator sample 出来的，是从 P_G 里面 sample 出来的，那我们要 maximize $\log(1 - D(x))$ ，就是要 maximize $1 - D(x)$ ，也就是要 minimize $D(x)$ 。

在训练的时候就是要找一个 D，它可以 maximize 这个 Objective Function。

如果你之前 Machine Learning 有学通的话，下面这个 optimization 的式子跟 train 一个 Binary Classifier 的式子，其实是完全一模一样的。

假设今天要 train 一个 Logistic Regression 的 model, Logistic Regression Model 是一个 Binary Classifier。然后就把 $P_{data}(x)$ 当作是 class 1, 把 P_G 当作是 class 2, 然后 train 一个 Logistic Regression Model。

你会发现你的 Objective Function 其实就是这个式子。所以这个 discriminator 在做的事情跟一个 Binary Classifier 在做的事情其实是一模一样的。

假设蓝色的点是 class 1, 红色的点是 class 2。discriminator 就是一个 Binary Classifier。然后这个 Binary Classifier 它是在 minimize Cross Entropy, 你其实就是在解这个 optimization 的 problem。这边神奇的地方是当我们解完这个 optimization 的 problem 的时候, 你最后会得到一个最小的 loss, 或者是得到最大的 objective value.

我们今天这边不是 minimize loss, 而是 maximize 一个 Objective Function。这个 V 是我们的 Objective Value, 我们要调 D 去 maximize 这个 Objective Value。然后这边神奇的地方是, 这个 maximize Objective Value

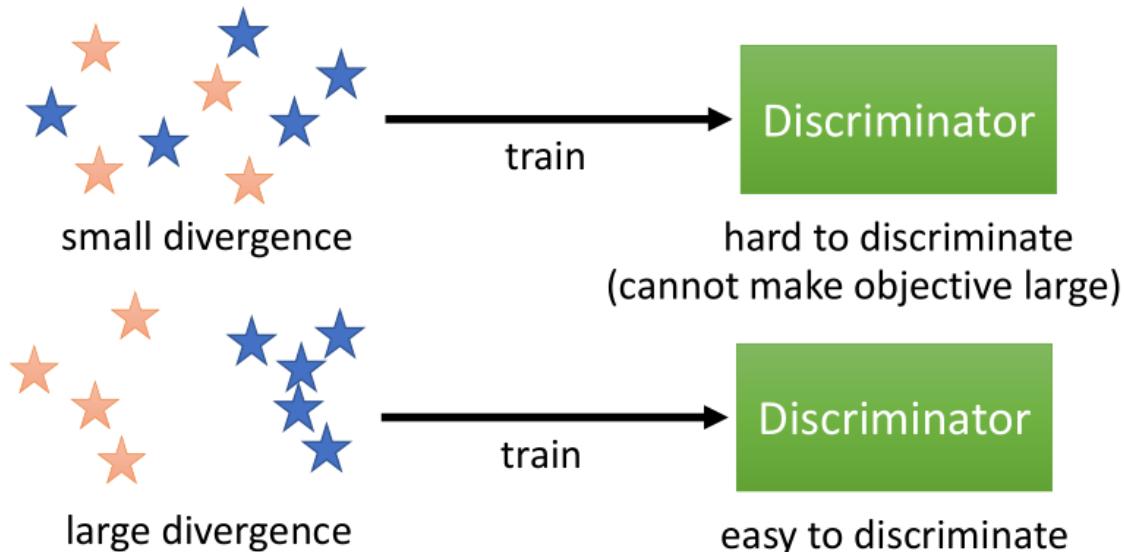
就是把这个 D train 到最好, 给了这些 data, 把这个 D train 到最好, 找出最大的 D 可以达到的 Objective Value。这个 value 其实会跟 JS Divergence 是有非常密切关系, 你说这个结果它其实就是 JS Divergence。

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

- ★ : data sampled from P_{data}
- ★ : data sampled from P_G

Training:

$$D^* = \arg \max_D V(D, G)$$



直观的解释: 你想想看, 假设现在 sample 出来的 data 它们靠得很近, 这个蓝色的这些星星跟红色的星星如果把它们视成两个类别的话, 它们靠得很近。对一个 Binary Classifier 来说, 它很难区别红色的星星跟蓝色的星星的不同, 因为对一个 Binary Classifier 也就是 discriminator 来说, 它很难区别这两个类别的不同, 所以直接 train 下去, loss 就没有办法压低。反过来说

在 training data 上的 loss 压不下去, 就是我们刚才看到的 Objective Value 没办法把它拉得很高, 没有办法找到一个 D 它让 V 的值变得很大。这个时候意味这两堆 data 它们是非常接近的, 它们的 Divergence 是小的。

所以如果对一个 discriminator 来说, 很难分别这两种 data 之间的不同, 它很难达到很大的 Objective Value, 那意味着这两堆 data 的 Divergence 是小的。所以最后你可以达到最好的 Objective Value, 跟 Divergence 是会有非常紧密的关系的。

这是一样的例子，假设蓝色的星星跟红色的星星它们距离很远，它们有很大的 Divergence，对 discriminator 来说它就可以轻易地分辨这两堆 data 的不同，也就是说它可以轻易的让你的 Objective Value，V 的这个 value 变得很大。当 V 的 value 可以变得很大的时候，意味着从 $P_{data}(x)$ 里面 sample 出来的东西和从 P_G generate 出来的东西，它们的 Divergence 是大的，所以 discriminator 就可以轻易地分辨它的不同，discriminator 就可以轻易的 maximize Objective Value。

Math

接下来就是实际证明为什么 Objective Value 跟 Divergence 是有关系的

$$\max_D V(G, D)$$

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

- Given G, what is the optimal D* maximizing

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

Assume that D(x) can be any function

- Given x, the optimal D* maximizing

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

转换成积分的形式，假设 D(x) 它可以是任何的 function (实际上不见得是成立的，因为假设 D(x) 是一个 network 除非它的 neural 无穷多，不然它也没有办法变成任何的 function)。

对 x 做积分中括号里面的式子，代各个不同的 x 再把它通通加起来

这就是积分在做的事情。假设 D(x) 可以是任意的 function 的话，这个时候 maximize 等同于把某一个 x 拿出来，然后要找一个 D 它可以让这个式子越大越好，所有不同的 x 通通都分开来算，因为所有的 x 都是没有任何关系的，因为不管是哪一个 x 你都可以 assign 给它一个不同的 D(x)。所以积分里面的每一项都分开来算，你就可以分开为它找一个最好的 D(x)。

$$\max_D V(G, D)$$

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

- Given x , the optimal D^* maximizing

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

- Find D^* maximizing: $f(D) = a \log(D) + b \log(1 - D)$

$$\frac{df(D)}{dD} = a \times \frac{1}{D} + b \times \frac{1}{1-D} \times (-1) = 0$$

$$a \times \frac{1}{D^*} = b \times \frac{1}{1 - D^*} \quad a \times (1 - D^*) = b \times D^* \\ a - aD^* = bD^* \quad a = (a + b)D^*$$

$$D^* = \frac{a}{a + b} \quad \rightarrow \quad D^*(x) = \frac{P_{data}(x)}{P_{data}(x) + P_G(x)} < 1$$

$P_{data}(x)$ 是固定的, P_G 也是固定的。唯一要做的事情就是找一个 $D(x)$ 的值让这个式子算起来最大。

求一下微分, 找出它的 Critical Point, 就是微分是 0 的地方, 求一下 $D^*(x)$ 是多少即可。

$$\max_D V(G, D)$$

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

$$\begin{aligned} \max_D V(G, D) &= V(G, D^*) & D^*(x) &= \frac{P_{data}(x)}{P_{data}(x) + P_G(x)} \\ &= E_{x \sim P_{data}} \left[\log \frac{P_{data}(x)}{P_{data}(x) + P_G(x)} \right] & &+ E_{x \sim P_G} \left[\log \frac{P_G(x)}{P_{data}(x) + P_G(x)} \right] \\ &= \int_x P_{data}(x) \log \frac{\frac{P_{data}(x)}{2}}{\frac{P_{data}(x) + P_G(x)}{2}} dx & &+ \int_x P_G(x) \log \frac{\frac{P_G(x)}{2}}{\frac{P_{data}(x) + P_G(x)}{2}} dx \\ &\quad + 2 \log \frac{1}{2} - 2 \log 2 & & \end{aligned}$$

接下来要做的事情就是把 D^* 代到这个式子里面, 看看 Objective Function 长什么样子。

为了要把整理成看起来像是 JS Divergence, 就把分子跟分母都同除 2, 把 $1/2$ 这一项把它提出来变成 $-2 \log 2$.

$$\begin{aligned} \text{JSD}(P \parallel Q) &= \frac{1}{2}D(P \parallel M) + \frac{1}{2}D(Q \parallel M) \\ \max_D V(G, D) & M = \frac{1}{2}(P + Q) \end{aligned}$$

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

后面这两项合起来就叫做 JS Divergence，如果 $P_{data}(x)$ 跟 P_G 它们距离的越远这两项合起来就越大，反之它们合起来就越小。

假设 learn 一个 discriminator，写出了某一个 Objective Function，去 maximize 那个 Objective Function 后得到的结果，maximize 的那个 Objective Function，maximize 的那个 value，其实就是 $P_{data}(x)$ 跟 P_G 的 JS Divergence

当我们 train 一个 discriminator 的时候，我们想做的事情就是去 evaluate

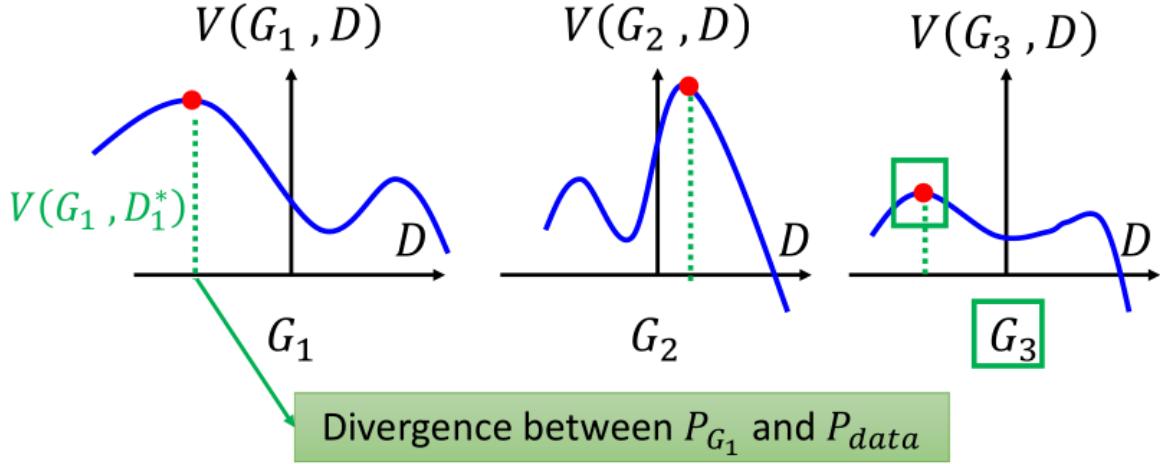
$P_{data}(x)$ 跟 P_G 这两个 distribution 之间的 JS Divergence。如果定的 Objective Function 是跟前面的式子一样的话，你就是在量 JS Divergence

如果把那个 Objective Function 写的不一样，你就可以量其它的各种不同的 Divergence。

$$G^* = \arg \min_G \max_D V(G, D)$$

$$D^* = \arg \max_D V(D, G)$$

The maximum objective value is related to JS divergence.



现在整个问题变成这个样子

本来要找一个 $G^* = \arg \min_G \text{Div}(P_G, P_{data})$, 但这个式子没有办法算。

于是我们写出一个 Objective Function $V(D, G)$, 找一个 D^* 去 maximize Objective Function, 它就是 P_G 和 $P_{data}(x)$ 之间的 Divergence

所以我们可以把 Divergence 这一项用 max 这一项把它替换掉, 变成上图第一个式子。

所以我们要找一个 generator, generate 出来的东西跟你的 data 越接近越好, 实际上要解这样一个 min max 的 optimization problem,

它实际上做的事情像是这个例子所讲的这样

假设世界上只有三个 generator, 要选一个 generator 去 minimize 这个 Objective Function, 但是可以选的 generator 总共只有三个, 一个是 G_1 一个是 G_2 一个是 G_3 . 假设选了 G_1 这个 generator 的话那 $V(G_1, D)$ 就是图中这个样子, 假设这个横坐标在改变的时候, 代表选择了不同的 discriminator。

接下来的问题是我们在给定一个 generator 的时候, 我们要找一个 discriminator 它可以让 $V(G, D)$ 最大。接下来要找一个 G 去 minimize 最大的那个 discriminator 可以找到的 value. 找一个 G 它可以 minimize $V(G, D)$, 用最大的 D 可以达到的 value。

现在要解这个 optimization problem, 哪一个 G 才是我们的 solution 呢? 正确答案是 G_3 。现在找出来的 G^* 就是 G_3 。

当我们给定一个 G_1 的时候, 这边这个 D_1^* 的这个高度其实就代表了 G_1 的 generator 它所 generate 出来的 distribution 跟 $P_{data}(x)$ 之间的距离。

所以 G_1 G_2 所定义的 distribution 跟 data 之间的 Divergence 比较大, 今天要 minimize Divergence 所以会选择 G_3 当作是最好的结果。

Algorithm

$$G^* = \arg \min_G \max_D V(G, D)$$

$$D^* = \arg \max_D V(D, G)$$

The maximum objective value is related to JS divergence.

- Initialize generator and discriminator
- In each training iteration:
 - Step 1:** Fix generator G, and update discriminator D
 - Step 2:** Fix discriminator D, and update generator G

接下来就是要想办法解这个 min max 的 problem。GAN 的这个算法就是在解这个 min max problem。解这个 min max problem 的目的就是要 minimize generator 跟你的 data 之间的 JS Divergence。

为什么这个 algorithm 是在解这一个 optimization problem?

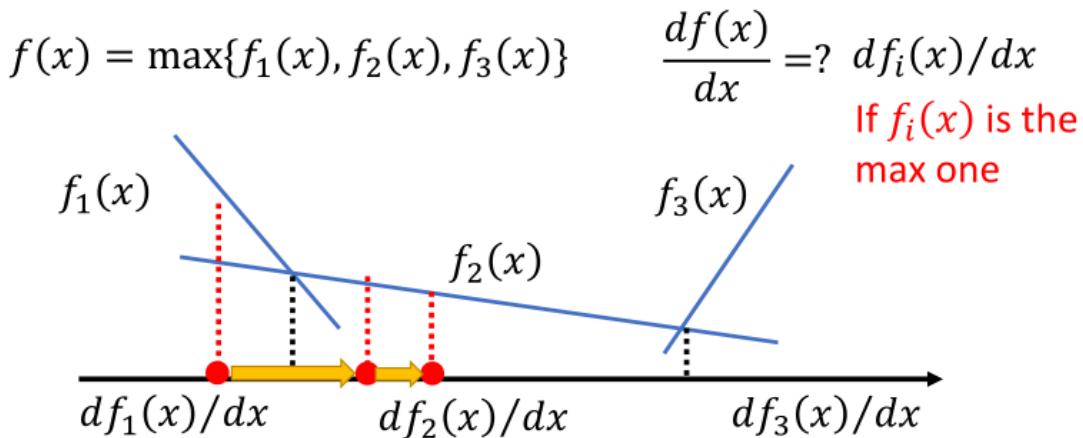
Algorithm

$$G^* = \arg \min_G \max_D V(G, D)$$

$$L(G)$$

- To find the best G minimizing the loss function $L(G)$,

$$\theta_G \leftarrow \theta_G - \eta \partial L(G) / \partial \theta_G \quad \theta_G \text{ defines } G$$



假设要解这个 optimization problem 的话用 $L(G)$ 来取代 $\max V(G, D)$ ，它其实跟 D 是没有关系的，given 一个 G 就会找到最好的 D 让 $V(G, D)$ 的值越大越好，假设最大的值就是 $L(G)$ 。

现在整个问题就变成要找一个最好的 generator G，它可以 minimize $L(G)$ 。

它就跟 train 一般的 network 是一样的，就是用 Gradient Descent 来解它。

$L(G)$ 式子里面有 max 的，有 max 可以微分吗？

我们之前有学到一个 Maxout Network，Maxout Network 里面也有 max operation，但它显然是有办法用 Gradient Descent 解。

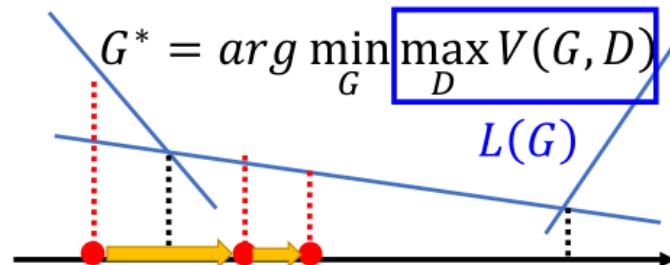
到底实际上是怎么做的呢

如果现在要把 $f(x)$ 对 x 做微分的话，这件事情等同于看看现在的 x 可以让哪一个 function f_1, f_2, f_3 最大，拿最大的那个出来算微分，就是 x 对 $f(x)$ 的微分。

假如你的这个 function 里面有一个 max operation，实际上在算微分的时候，你只是看现在在 f_1, f_2, f_3 里面哪一个最大，就把最大的那个人拿出来算微分。你就可以用 Gradient Descent 去 optimize 这个 $f(x)$ 。

总之就算是 Objective Function 里面有 max operation，你一样是可以对它做微分的。

Algorithm



- Given G_0
- Find D_0^* maximizing $V(G_0, D)$ **Using Gradient Ascent**

$V(G_0, D_0^*)$ is the JS divergence between $P_{data}(x)$ and $P_{G_0}(x)$

- $\theta_G \leftarrow \theta_G - \eta \partial V(G, D_0^*) / \partial \theta_G \rightarrow$ Obtain G_1 **Decrease JS divergence(?)**
- Find D_1^* maximizing $V(G_1, D)$

$V(G_1, D_1^*)$ is the JS divergence between $P_{data}(x)$ and $P_{G_1}(x)$

- $\theta_G \leftarrow \theta_G - \eta \partial V(G, D_1^*) / \partial \theta_G \rightarrow$ Obtain G_2 **Decrease JS divergence(?)**
-

所以就回到现在要解的这个 optimization problem

一开始有一个初始的 G_0 ，接下来要算 G_0 对 $L(G)$ 的 gradient，但是在算 G_0 对 $L(G)$ 的 gradient 之前，因为 $L(G)$ 里面有 max，所以不知道 $L(G)$ 长什么样子，要把 max D 找出来。

所以假设在 given G_0 前提之下， D_0^* 可以让 $V(G_0, D)$ 最大，如果这个 D 代 D_0^* 的话，就可以得到 $L(G)$ 。可以用 Gradient Ascent 就可以找出这个 D 。

找到 D 可以 maximize 这个 Objective Function 以后，就是 $L(G)$ ，把 θ_G 对这一项算 gradient，就可以 update 参数，就得到新的 generator G_1 。

有新的 generator G_1 以后，就要重新找一下最好的 D ，可以让这个 $V(G_1, D)$ 最大的那个 D 假设是 D_1^* ，接下来就有一个新的 Objective Function，重新计算 gradient 再 update generator，得到 G_2

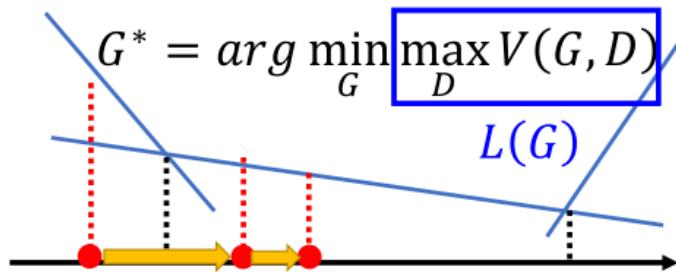
这个 operation 就是一个 G_0 ，找一个可以让 $V(G_0, D)$ 最大的 D_0^* ，就得到 V 的 function。然后让它对 G 做微分，再重新去找一个新的 D ，再重新对 Objective Function 做微分。就会发现这整个 process 其实跟 GAN 是一模一样的。

你可以把它想成现在在找 D_0^* 去 maximize 这个 Objective Function 的 process，其实就是在量 $P_{data}(x)$ 跟 P_{G_0} 的 JS Divergence。

找到一个 D_1^* 它可以让这个 Objective Function 的值变 maximum, 其实就是在计算 $P_{data}(x)$ 跟 P_{G_1} 的 JS Divergence。

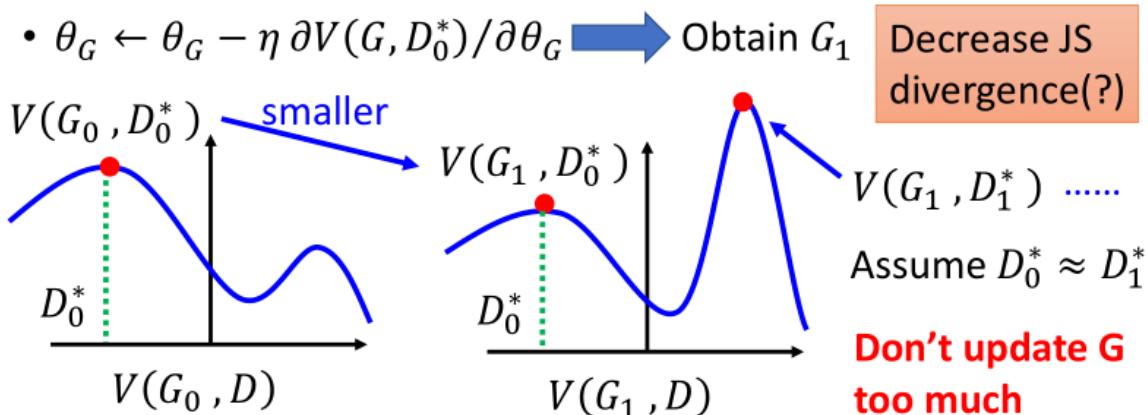
我们求gradient的一项就是你的 JS Divergence, 你要 update generator 去 minimize JS Divergence, 这个时候你其实就是在减少你的JS Divergence, 就是在达成你的目标。

Algorithm



- Given G_0
- Find D_0^* maximizing $V(G_0, D)$

$V(G_0, D_0^*)$ is the JS divergence between $P_{data}(x)$ and $P_{G_0}(x)$



但是这边打了一个问号，因为这件事情未必等同于真的在 minimize JS Divergence。

为什么这么说，因为假设给你一个generator G_0 , 那你的 $V(G_0, D)$ 假设它长这个样子，找到一个 D_0^* , 这个 D_0^* 的值，就是 G_0 跟你的 data 之间的JS Divergence；但是当你 update 你的 G_0 ，变成 G_1 的时候，这个时候 $V(G_1, D)$ 它的 function 可能就会变了。本来 $V(G_0, D)$ 是这个样子， $V(G_0, D_0^*)$ 就是 G_0 跟你的 data 的JS Divergence，今天你 update 你的 G_0 变成 G_1 ，这个时候整个 function 就变了，这个时候因为 G_0^* 仍然是固定的，所以 $V(G_1, D_0^*)$ 它就不是在 evaluate JS Divergence。我们说 evaluate JS Divergence 的 D 是 $V(G, D)$ 这个值里面最大的那个，所以当你的 G 变了，你的这个 function 就变了，当你的 function 变的时候同样的 D 就不是在 evaluate 你的JS Divergence。如果在这个例子里面，JS Divergence 会变大。

但是为什么我们又说这一项可以看作是在减少JS Divergence 呢？这边作的前提假设就是这两个式子可能是非常像的，假设只 update 一点点的 G 从 G_0 变到 G_1 ， G 的参数只动了一点点，那这两个 function 它们的长相可能是比较像的。因为它们的长相还是比较像的，所以一样用 D_0^* 你仍然是在量JS Divergence，这边本来值很小，突然变很高的情形可能是不会发生的。因为 G_0 跟 G_1 是很像的所以这两个 function 应该是比较接近。所以你可以只同样用固定的 D_0^* ，就可以 evaluate G_0 跟 G_1 的JS Divergence。

所以在 train GAN 的时候，它的 tip 就是因为你有这个假设，就是 G_0 跟 G_1 应该是比较像的，所以在 train generator 的时候，你就不能够一次 update 太多。但是在 train discriminator 的时候，理论上应该把它 train 到底，应该把它 update 多次一点，因为你必须要找到 maximum 的值你才是在量JS Divergence，所以 train discriminator 的时候，你其实会需要比较多的 iteration 把它 train 到底。但是 generator 的话，你应该只要跑比较少的 iteration，免得投影片上讲的假设是不成立的。

In practice ...

接下来讲一下实际上在做 GAN 的时候其实是怎么做的。

我们的 Objective Function 里面要对 x 取 expectation，但是在实际上没有办法真的算 expectation，所以都是用 sample 来代替 expectation。

实际上在做的时候，我们就是在 maximize 图中这个式子，而不是真的去 maximize 它的 expectation。

这个式子就等同于是在 train 一个 Binary Classifier

所以在实作 GAN 的时候，你完全不需要用原来不知道的东西，你在 train discriminator 的时候，你就是在 train 一个 Binary Classifier。

实际在做的时候 discriminator 是一个 Binary Classifier，这个 Binary Classifier 它是一个 Logistic Regression，它的 output 有接一个 sigmoid，所以它 output 的值是介于 0 到 1 之间的。

然后从 $P_{data}(x)$ 里面 sample m 笔 data 出来，这 m 笔 data 就当作是 positive example 或是 class 1 的 example；然后从 P_G 里面再 sample 另外 m 笔 data 出来，这 m 笔 data 就当作是 negative example，就当作是 class 2 的 example。接下来就 train 你的 Binary Classifier，train 一个 criterion 来 minimize Cross Entropy，minimize Cross Entropy 的式子写出来，它会等同于上面 maximize 这个 Objective Function。

Algorithm

Algorithm		Initialize θ_d for D and θ_g for G
Learning D	• In each training iteration:	Can only find lower bound of $\max_D V(G, D)$
Repeat k times	<ul style="list-style-type: none">Sample m examples $\{x^1, x^2, \dots, x^m\}$ from data distribution $P_{data}(x)$Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from the prior $P_{prior}(z)$Obtaining generated data $\{\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^m\}$, $\tilde{x}^i = G(z^i)$Update discriminator parameters θ_d to maximize<ul style="list-style-type: none">$\tilde{V} = \frac{1}{m} \sum_{i=1}^m \log D(x^i) + \frac{1}{m} \sum_{i=1}^m \log (1 - D(\tilde{x}^i))$$\theta_d \leftarrow \theta_d + \eta \nabla \tilde{V}(\theta_d)$Sample another m noise samples $\{z^1, z^2, \dots, z^m\}$ from the prior $P_{prior}(z)$	
Learning G Only Once	<ul style="list-style-type: none">Update generator parameters θ_g to minimize<ul style="list-style-type: none">$\tilde{V} = \frac{1}{m} \sum_{i=1}^m \log D(\tilde{x}^i) + \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^i)))$$\theta_g \leftarrow \theta_g - \eta \nabla \tilde{V}(\theta_g)$	

最后就再重新复习一次 GAN 的 algorithm

我们之前有讲过我们 train discriminator 的目的是什么，是为了要 evaluate JS Divergence，而当它可以让你的 V 的值最大的时候，那个 discriminator 才是在 evaluate JS divergence。

所以你一定要 train 很多次，train 到收敛为止，它才能让 V 的值最大，但在实作上你没有办法真的 train 很多次，train 到收敛为止。但是你会说，我今天 train d 的时候，我要反复 k 次，这个参数要 update k 次，而不是像投影片上面只写 update 一次而已，你可能会 update 三次或五次才停止。

这个步骤是在解这个问题，找一个 D 它可以 maximize $V(G, D)$

但是其实你没有办法真的找到一个最好的 D 去 maximize $V(G, D)$ ，你能够找的其实只是一个 lower bound 而已。因为这边通常在实作的时候你没有办法真的 train 到收敛，你没有办法真的一直 train，train 到说可以让 $V(G, D)$ 变的最大，通常就是 train 几步然后就停下来。就算我们退一万步说这边可以一直 train，train 到收敛，你其实也未必真的能够 maximize 这个 Objective Function，因为在 train 的时候，D 的 capacity 并不是无穷大的，你会卡在一个 Local Maximum 然后就结束了，你并不真的可以 maximize 这个式子。再退一万步说假设没有 Local Maximum 的问题，你可以直接解这个问题，你的 D 它的 capacity 也是有限，记得我们说过如果要量 JS Divergence，一个假设是 D 可以是任何 function，事实上 D 是一个 network，所以它也不是任何 function，所以你没有办法真的 maximize $V(G, D)$ ，你能够找到的只是一个 lower bound 而已。但我们就假设你可以 maximize 这一项就是了。

接下来要 train generator，我们说 train discriminator 是为了量 JS Divergence，train generator 的时候是为了要 minimize JS Divergence。

为了要减少 JS Divergence，下面这个式子里面你会发现第一项跟 generator 是没有关系的，因为第一项只跟 discriminator 有关，它跟 generator 没有关系，所以要 train generator 去 minimize 这个式子的时候，第一项是可以不用考虑它的，所以把第一项拿掉只去 minimize 第二项式子。这个第二个步骤就是在 train generator，刚才有讲过 generator 不能够 train 太多，因为一旦 train 太多的话，discriminator 就没有办法 evaluate JS Divergence。所以 generator 不能 train 太多，你只能少量的 update 它的参数而已，所以通常 generator update 一次就好。

你可以 update discriminator 很多次，但是 generator update 一次就好。你 update 太多，量出来 JS Divergence 就不对了。所以这边就不能够 update 太多。

Objective Function for Generator in Real Implementation

Objective Function for Generator in Real Implementation

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

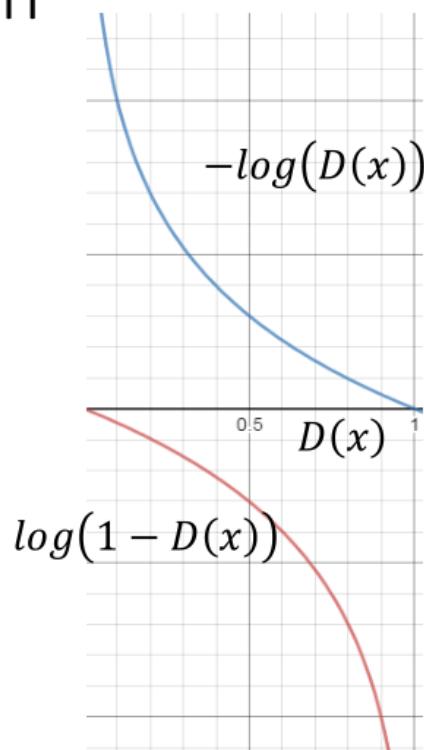
Slow at the beginning

Minimax GAN (MMGAN)

$$V = E_{x \sim P_G} [-\log(D(x))]$$

Real implementation:
label x from P_G as positive

Non-saturating GAN (NSGAN)



到目前为止讲说 train generator 的时候，你要去 minimize 的式子长上面这个样子。

但在 Ian Goodfellow 原始的 paper 里面，从有 GAN 以来，它就不是在 minimize 这个式子，paper 加了一小段，说这个式子 $\log(1 - D(x))$ 它长的是右边这个样子，而我们一开始在做 training 的时候 $D(x)$ 的值通常是很小的，因为 discriminator 会知道 generator 产生出来的 image 它是 fake 的，所以它会给它很小的值，所以一开始 $D(x)$ 的值会落在微分很小的地方，所以在 training 的时候，会造成你在 training 的一些问题，所以说我们把它改成这个样子。

没有为什么，它们的趋势是一样的，但是它们在同一个位置的斜率就变得不一样。在一开始 $D(x)$ 还很小的时候，算出来的微分会比较大，所以 Ian Goodfellow 觉得这样子 training 是比较容易的。

其实你再从另外一个实作的角度来看，如果你是要 minimize 上边这个式子，你会发现你需要改 code 有点麻烦。如果你是 minimize 下边这个式子你可以不用改 code。如果你是要 minimize 下面这个式子的时候，其实只是把 Binary Classifier 的 label 换过来，本来是说从 data sample 出来的是 class 1，从 generator sample 出来的是 class 2，把它 label 换过来，把 generator sample 出来的改标成 label 1，然后用同样的 code 跑下去就可以了。我认为 Ian Goodfellow 只是懒得改 code 而已，所以就胡乱编一个理由应该要用下面这个式子。（大雾

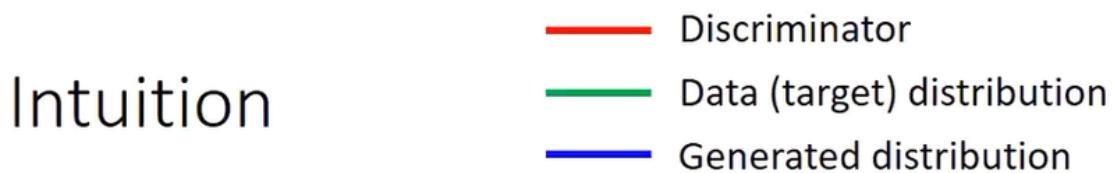
但实际上后来有人试了比较这两种不同的方法，发现都可以 train 得起来，performance 也是差不多的，不知道为什么 Ian Goodfellow 一开始就选了这个。

后来 Ian Goodfellow 还写了另外一篇文章，把上面这个叫做 Minimax GAN 就是 MMGAN，把下面这个叫做 Non-saturating GAN 就是 NSGAN。

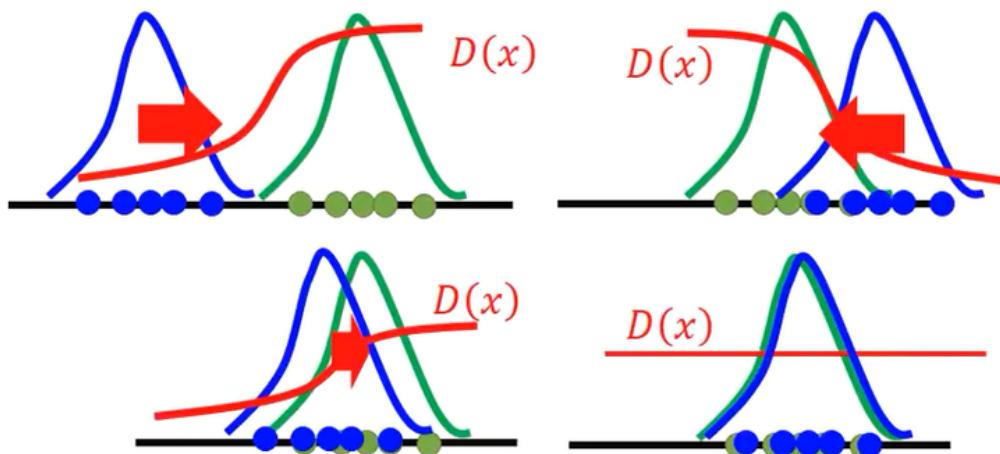
Intuition

现在讲一些比较直观的东西

所以按照 Ian Goodfellow 的讲法今天这个 generator 和 discriminator 它们之间的关系是什么样呢？



- Discriminator leads the generator



<https://www.youtube.com/watch?v=ebMei6bYeWw>

绿色的点是你的目标，蓝色的点是 generator 产生出来的东西

背景的颜色是 discriminator 的值，discriminator 会 assign 给每一个 space 上的 x 一个值，背景的这个颜色是 discriminator 的值。

你就会发现这个 discriminator 就把 P_G 产生出来蓝色的点赶来赶去，直到最后蓝色的点跟绿色的点重合在一起的时候，discriminator 就会坏掉，因为完全没有办法分辨 generator 跟 discriminator 之间的差别。

Question

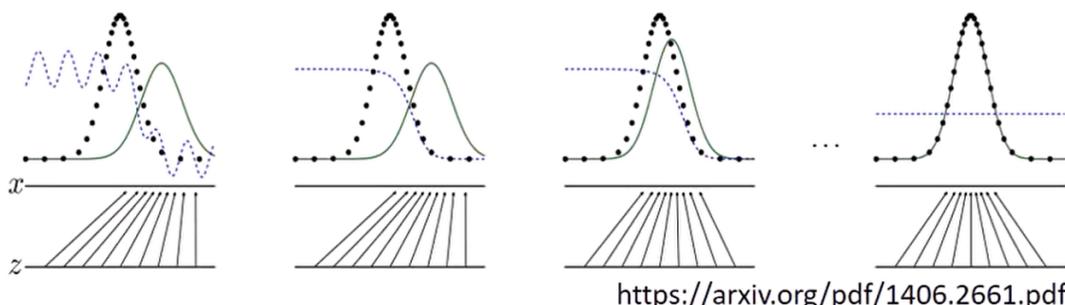
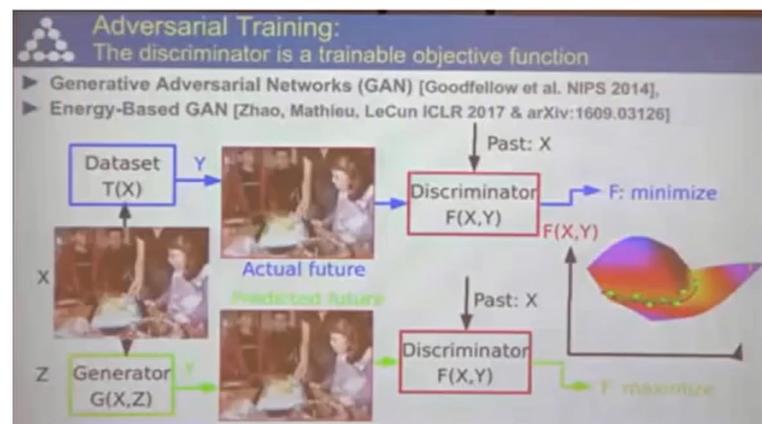
会不会出现data imbalance ?

一般在做的时候，在 train 一个 classifier 的时候其实会害怕 data imbalance 的问题，今天在这个 task 里面，data 是自己 sample 出来的，我们不会给自己制造 data imbalance 的问题，所以两种 task 会 sample 一样的数目，假设从 generator 里面 generate 256 笔 data，那你今天从你的 sample 的 database 里面你也会 sample 256 笔 data。

Yann LeCun's talk

Question?

Can we consider
discriminator as
evaluation function?



<https://arxiv.org/pdf/1406.2661.pdf>

你不觉得今天讲的跟上周讲的是有点矛盾的吗

如果按照 Ian Goodfellow 的讲法，最后 discriminator train 到后来它就会烂掉变成一个水平线，但我们说 discriminator 其实就是 evaluation function，也就是说 discriminator 的值代表它想要判断这个 object，generate 出来的东西它到底是好还是不好。

如果 discriminator 是一条水平线，它就不是一个 evaluation function，对它来说所有的东西都是一样好，或者是一样坏。

右上角是 Yann LeCun 画的图，这个图就是 discriminator 的图，绿色的点就是 real data 分布，你发现他在画的时候，在他的想象里面 discriminator 并没有烂掉变成一个水平线，而是有 data 分布的地方它会得到比较小的值，而没有 data 分布的地方它会得到比较大的值。跟之前讲的是相反的，不过意思完全是一样的。

跟 Ian Goodfellow 讲的是有一些矛盾的，这个就是神奇的地方，因为这个都是尚待发展中的理论，所以有很多的问题是未知的。

以前在 train Deep Learning 的时候，我们都要用 Restricted Boltzmann Machine，过去我们都相信没有 Restricted Boltzmann Machine 是 train 不起来的，但现在根本就用不上这个技术。

所以这个变化是非常快的，也许明年再来讲同样东西的时候，就会有截然不同的讲法也说不定。

你如果问我到底是哪一种的话，假设你硬要我给你一个答案，告诉你到底应该是 Ian Goodfellow 讲得比较对，还是 Yann LeCun 讲得比较对。我的感觉是首先可以从实验上来看看，如果你真的 train 完你的 GAN，然后去 evaluate 一下 discriminator，它的感觉好像是介于这两个 case 中间，它绝对不是烂掉，绝对不是变成一个完全烂掉的 discriminator。

你自己回去做做看，几时 train 出这样的结果，虽然是这种简单的例子你也 train 不出这个结果的，就算是一维的例子也都做不出这个结果。所以不太像是 Ian Goodfellow 讲的这样。但是 discriminator 也不完全反映了 data distribution，感觉是介于这两个 case 之间。

这些观点到底对我们了解 GAN 有什么帮助？

也许 GAN 的 algorithm 就是一样，那算法就是那个样子，就是 train generator、train discriminator、iterative train，也许它的 algorithm 是不会随着你的观点不同。

但是你用不同的观点来看待 GAN，你其实再设计 algorithm 的时候，中间会有些微妙的差别，也许这些微妙的差别导致最后 training 的结果会是很不一样的。

我觉得也许 Yann LeCun 的这个讲法，之前讲的 discriminator 是在 evaluate 一个 object 的好还是不好，它是在反映了 data distribution 这件事也许更接近现实。

为什么会这么说？

首先，你在文献上会看到很多人会把 discriminator 当作 classifier 来用，所以先 train 好一个 GAN，然后把 discriminator 拿来做其它事情。假设 discriminator train 到最后，按照 Ian Goodfellow 猜想会烂掉的话，拿它来当作 pre-training 根本就没有意义，但很多人会拿它当作 pre-training，也显示它是有用的，所以它不太可能真的 train 到后来就坏掉。这个是第一个 evidence。

另外一个 evidence 是你想想看你在 train GAN 的时候，你并不是每一次都重新 train discriminator，而是会拿前一个 iteration 的 discriminator，当作下一个 iteration 的 initialize 的参数。如果你的 discriminator 是想要衡量两个 data distribution 的 Divergence 的话，你其实没有必要把前一个 iteration 的东西拿来用，因为 generator 已经变了，保留前一个 iteration 的东西有什么意义呢？这样感觉是不太合理的。也有人可能会说因为 generator update 的参数，update 的量是比较小的，所以也许把前一个 time step 得到的 generator，当作下一个 time step 的 initialization，可以加快 discriminator 训练的速度，也说不定，这个理由感觉也是成立的。

不过在文献上我看到有人在 train GAN 的时候它有一招，每次 train 的时候它不只拿现在的 generator 去 sample data，它也会拿过去的 generator 也 sample data，然后把这些各个不同 generator sample 的 data 通通集合起来，再去 train discriminator，可以得到的 performance 会是比较好的。

如果 discriminator 是在 evaluate 现在的 generator，跟 data distribution 的差异的话，好像做这件事情也没有太大的意义，因为现在量 generator 跟 data 之间的差异，拿过去 generator 产生的东西有什么用？没什么用。但是在实作上发现拿过去 generator 产生的东西，再去训练 discriminator 是可以得到比较好的成果。所以这样看起来，也许这是另外一个 support 支持也许 discriminator 在做的事情，并不见得是在 evaluate 两个 distribution 之间的 Divergence。

不过至少 Ian Goodfellow 一开始是这么说的，所以我们把 GAN 最开始的理论告诉大家。

fGAN: General Framework of GAN

我们定某种 objective function，就是在量 js divergences。那我们能不能够量其他的 divergence 呢？

fGAN 就是要告诉我们怎么量其他的 divergences。

这个东西有点用不上，原因就是，fGAN 可以让你用不同的 f divergences 来量你 generated 的 example 跟 real example 的差距。但是用不同的 x divergences 的结果是差不多的，所以这一招好像没什么特别有用的地方。

但是我们还是跟大家介绍一下，因为这个在数学上，感觉非常的厉害，但是在实作上，好像没什么特别的不同。

fGAN 想要告诉我们的是，其实不只是用 js divergence，任何的f-divergence都可以放到 GAN 的架构里面去。

f-divergence

f-divergence P and Q are two distributions. $p(x)$ and $q(x)$ are the probability of sampling x .

$$D_f(P||Q) = \int_x q(x)f\left(\frac{p(x)}{q(x)}\right)dx \quad \begin{array}{l} f \text{ is convex} \\ f(1) = 0 \end{array} \quad D_f(P||Q) \text{ evaluates the difference of P and Q}$$

If $p(x) = q(x)$ for all x

smallest

$$D_f(P||Q) = \int_x q(x)f\left(\frac{p(x)}{q(x)}\right)dx = 0$$

$$\begin{array}{c} = 1 \\ \boxed{\frac{p(x)}{q(x)}} \\ = 0 \end{array}$$

$$D_f(P||Q) = \int_x q(x)f\left(\frac{p(x)}{q(x)}\right)dx$$

Because f is convex

$$\geq f\left(\int_x q(x)\frac{p(x)}{q(x)}dx\right)$$

$$= f(1) = 0$$

If P and Q are the same distributions,
 $D_f(P||Q)$ has the smallest value, which is 0

f-divergence $D_f(P||Q) = \int_x q(x)f\left(\frac{p(x)}{q(x)}\right)dx$

有两个条件， f is convex and $f(1) = 0$

f-divergence

$$D_f(P||Q) = \int_x q(x)f\left(\frac{p(x)}{q(x)}\right)dx \quad \begin{array}{l} f \text{ is convex} \\ f(1) = 0 \end{array}$$

$$f(x) = x \log x$$

$$D_f(P||Q) = \int_x q(x) \frac{p(x)}{q(x)} \log\left(\frac{p(x)}{q(x)}\right) dx = \int_x p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

$$f(x) = -\log x$$

$$D_f(P||Q) = \int_x q(x) \left(-\log\left(\frac{p(x)}{q(x)}\right)\right) dx = \int_x q(x) \log\left(\frac{q(x)}{p(x)}\right) dx$$

$$f(x) = (x - 1)^2$$

$$D_f(P||Q) = \int_x q(x) \left(\frac{p(x)}{q(x)} - 1\right)^2 dx = \int_x \frac{(p(x) - q(x))^2}{q(x)} dx$$

假设 f 带不同的式子，你就得到各式各样的 f -divergence 的 measure。

Fenchel Conjugate

Fenchel Conjugate

$$D_f(P||Q) = \int_x q(x)f\left(\frac{p(x)}{q(x)}\right)dx$$

f is convex, $f(1) = 0$

- Every convex function f has a conjugate function f^*

$$f^*(t) = \max_{x \in \text{dom}(f)} \{xt - f(x)\}$$

$$f^*(\mathbf{t}_1) = \max_{x \in \text{dom}(f)} \{x\mathbf{t}_1 - f(x)\}$$

$$x_1 \mathbf{t}_1 - f(x_1) \bullet f^*(\mathbf{t}_1) \quad f^*(\mathbf{t}_2) = \max_{x \in \text{dom}(f)} \{x\mathbf{t}_2 - f(x)\}$$

$$x_2 \mathbf{t}_1 - f(x_2)$$

$$x_3 \mathbf{t}_2 - f(x_3) \bullet f^*(\mathbf{t}_2)$$

$$x_3 \mathbf{t}_1 - f(x_3)$$

$$x_2 \mathbf{t}_2 - f(x_2)$$

$$x_1 \mathbf{t}_2 - f(x_1)$$



要知道 $f^*(t)$ 长什么样子，就把 t 的每一点，通通这个方法去算。

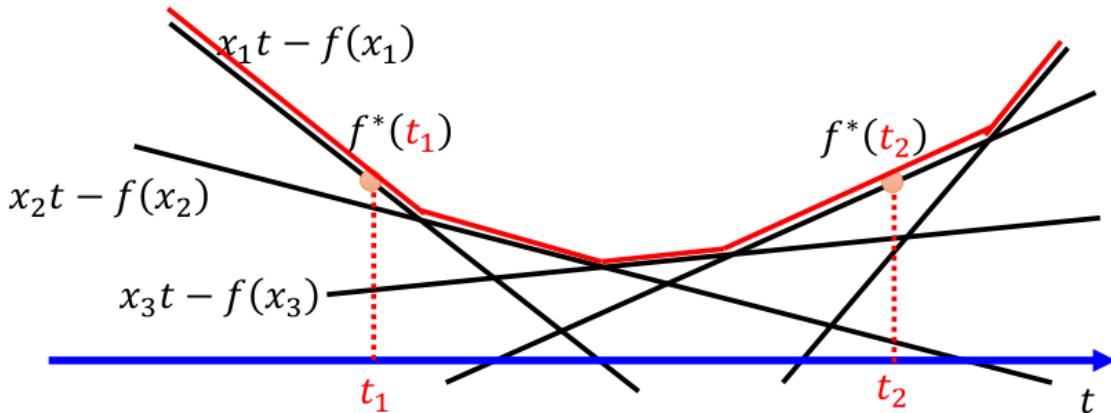
Fenchel Conjugate

$$D_f(P||Q) = \int_x q(x) f\left(\frac{p(x)}{q(x)}\right) dx$$

f is convex, f(1) = 0

- Every convex function f has a conjugate function f^*

$$f^*(t) = \max_{x \in \text{dom}(f)} \{xt - f(x)\}$$

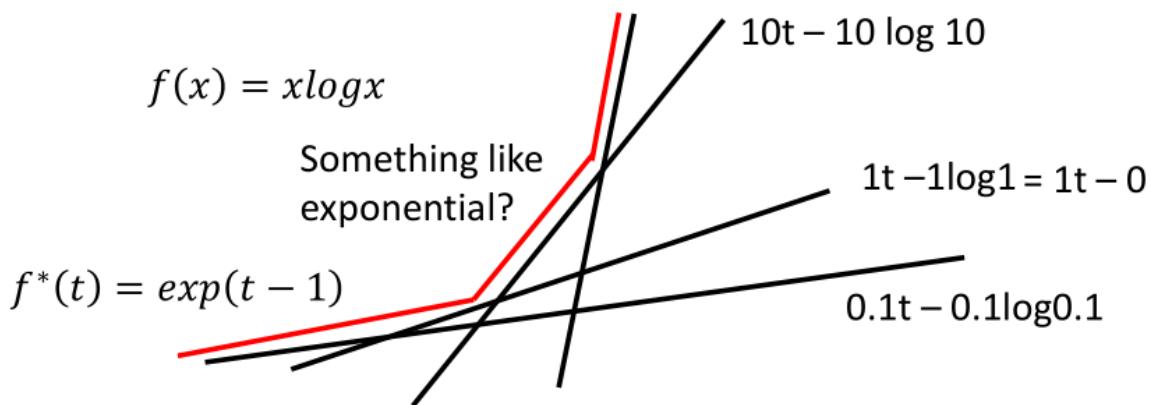


也可以用这个方法把 $f^*(t)$ 画出来。就是把所有不同的 x 所造成的直线，通通画出来，然后再取它们的 upper bound。

所以今天你会发现 $f^*(t)$ 一定是 convex 的，如果很多条直线，随便乱画，不管你怎么随便画，最后你只要找的是 upper bound，得到的 function 都是 convex 的。

- Every convex function f has a conjugate function f^*

$$f^*(t) = \max_{x \in \text{dom}(f)} \{xt - f(x)\}$$



你从 0.1 带进去，得到一条线，0.1001 带进去，也是一条线，0.1002 带进去，也是一条线，通通带进去，你得到无穷无尽的线。把这些所有的线 upper bound 都找出来，就是红色这一条线，会发现说这条红色的线，它看起来像是 exponential。这一条红色的线，它是 $\exp(t-1)$ 。所以 $f(x) = x \log x$ 的 conjugate，就是 $\exp(t-1)$ 。

Proof

- Every convex function f has a conjugate function f^*

- $(f^*)^* = f$

$$f^*(t) = \max_{x \in \text{dom}(f)} \{xt - f(x)\}$$

$$f(x) = x \log x \longleftrightarrow f^*(t) = \exp(t - 1)$$

$$f^*(t) = \max_{x \in \text{dom}(f)} \{xt - x \log x\}$$

$$g(x) = xt - x \log x \quad \text{Given } t, \text{ find } x \text{ maximizing } g(x)$$

$$t - \log x - 1 = 0 \quad x = \exp(t - 1)$$

$$f^*(t) = \exp(t - 1) \times t - \exp(t - 1) \times (t - 1) = \exp(t - 1)$$

Connection with GAN

$$f^*(t) = \max_{x \in \text{dom}(f)} \{xt - f(x)\} \longleftrightarrow f(x) = \max_{t \in \text{dom}(f^*)} \{xt - f^*(t)\}$$

$$\begin{aligned} D_f(P||Q) &= \int_x q(x) f\left(\frac{p(x)}{q(x)}\right) dx \quad \frac{p(x)}{q(x)} \quad \frac{p(x)}{q(x)} \\ &= \int_x q(x) \left(\max_{t \in \text{dom}(f^*)} \left\{ \frac{p(x)}{q(x)} t - f^*(t) \right\} \right) dx \end{aligned}$$

$$\approx \max_D \int_x p(x) D(x) dx - \int_x q(x) f^*(D(x)) dx$$

D is a function
whose input is x ,
and output is t

$$D_f(P||Q) \geq \int_x q(x) \left(\frac{p(x)}{q(x)} D(x) - f^*(D(x)) \right) dx$$

$$= \int_x p(x) D(x) dx - \int_x q(x) f^*(D(x)) dx$$

我们 learn 一个 D , 它就是 input 一个 x , 它 output 的这个 scalar, 就是这边这个 t , 所以我们把这个 t 用 $D(x)$ 取代掉。

所以我们希望可以 learn 出一个 function, 这个 discriminator 帮我们解这个 max 的 problem, input 一个 x , 它告诉我说, 你现在 input 这个 x 后, 到底哪一个 t , 可以让这个值最大。 D 就是要做这件事。

但是因为假设 D 的 capacity 是有限的, 那你今天把这个 t 换成 $D(x)$, 就会变成是 f -divergence 的一个 lower bound。

所以我们找一个 D , 它可以去 maximize 这一项, 它就可以去逼近 f -divergence。

$$\begin{aligned}
D_f(P||Q) &\approx \max_D \int_x p(x)D(x)dx - \int_x q(x)f^*(D(x))dx \\
&= \max_D \{E_{x \sim P}[D(x)] - E_{x \sim Q}[f^*(D(x))]\} \\
&\quad \text{Samples from P} \qquad \text{Samples from Q} \\
D_f(P_{data}||P_G) &= \max_D \{E_{x \sim P_{data}}[D(x)] - E_{x \sim P_G}[f^*(D(x))]\} \\
G^* &= \arg \min_G D_f(P_{data}||P_G) \qquad \text{Original GAN has different V(G,D)} \\
&= \arg \min_G \max_D \{E_{x \sim P_{data}}[D(x)] - E_{x \sim P_G}[f^*(D(x))]\} \\
&= \arg \min_G \max_D V(G, D) \quad \text{familiar? 😊}
\end{aligned}$$

变成期望的形式，把 p 改成 p data，把 Q 改成 PG。

所以，p data 跟 PG 之间的 f-divergence，就可以写成这个式子。f-divergence 是什么，就会影响到这个 f^* 是什么。

所以今天假如你的 f-divergence 是 KL divergence，那你就看 KL-divergence f^* 是什么。KL divergence f 是 $x \log x$ ，它的 f^* 是 $\exp(t-1)$ ，所以这个 f^* 就带 $\exp(t-1)$ 。

这个式子跟 GAN 看起来的式子，看起来很像。

想想看我们今天在 train 一个 generator 的时候，我们要做的事情，就是去 minimize 某一个 divergence。而这个 divergence，我们就可以把它写成这个式子。随着你要用什么 divergence，你这 f^* 就换不同的式子，你就是在量不同的 divergence。

而这个东西就是我们说在 train GAN 的时候，你要用 discriminator 去 maximize 你的 generator 要去 minimize 的 objective function V of (G,D)。只是 V (G,D) 的定义不同，就是在量不同的 divergence。

$$D_f(P_{data} || P_G) = \max_D \{ E_{x \sim P_{data}} [D(x)] - E_{x \sim P_G} [f^*(D(x))] \}$$

Name	$D_f(P Q)$	Generator $f(u)$
Total variation	$\frac{1}{2} \int p(x) - q(x) dx$	$\frac{1}{2} u - 1 $
Kullback-Leibler	$\int p(x) \log \frac{p(x)}{q(x)} dx$	$u \log u$
Reverse Kullback-Leibler	$\int q(x) \log \frac{q(x)}{p(x)} dx$	$-\log u$
Pearson χ^2	$\int \frac{(q(x)-p(x))^2}{p(x)} dx$	$(u - 1)^2$
Neyman χ^2	$\int \frac{(p(x)-q(x))^2}{q(x)} dx$	$\frac{(1-u)^2}{u}$
Squared Hellinger	$\int (\sqrt{p(x)} - \sqrt{q(x)})^2 dx$	$(\sqrt{u} - 1)^2$
Jeffrey	$\int (p(x) - q(x)) \log \left(\frac{p(x)}{q(x)} \right) dx$	$(u - 1) \log u$
Jensen-Shannon	$\frac{1}{2} \int p(x) \log \frac{2p(x)}{p(x)+q(x)} + q(x) \log \frac{2q(x)}{p(x)+q(x)} dx$	$-(u + 1) \log \frac{1+u}{2} + u \log u$
Jensen-Shannon-weighted	$\int p(x) \pi \log \frac{p(x)}{\pi p(x)+(1-\pi)q(x)} + (1 - \pi)q(x) \log \frac{q(x)}{\pi p(x)+(1-\pi)q(x)} dx$	$\pi u \log u - (1 - \pi + \pi u) \log(1 - \pi + \pi u)$
GAN	$\int p(x) \log \frac{2p(x)}{p(x)+q(x)} + q(x) \log \frac{2q(x)}{p(x)+q(x)} dx - \log(4)$	$u \log u - (u + 1) \log(u + 1)$

Name	Conjugate $f^*(t)$
Total variation	t
Kullback-Leibler (KL)	$\exp(t - 1)$
Reverse KL	$-1 - \log(-t)$
Pearson χ^2	$\frac{1}{4}t^2 + t$
Neyman χ^2	$2 - 2\sqrt{1-t}$
Squared Hellinger	$\frac{t}{1-t}$
Jeffrey	$W(e^{1-t}) + \frac{1}{W(e^{1-t})} + t - 2$
Jensen-Shannon	$-\log(2 - \exp(t))$
Jensen-Shannon-weighted	$(1 - \pi) \log \frac{1-\pi}{1-\pi e^{t/\pi}}$
GAN	$-\log(1 - \exp(t))$

Using the f-divergence you like 😊

<https://arxiv.org/pdf/1606.00709.pdf>

这边就是从 paper 上面的图，它就告诉你说各种不同的 divergence 的 objective function。

那可以 optimize 不同的 divergence，到底有什么厉害的地方呢？

Mode Collapse

也许这一招可以解决一个长期以来困扰大家的问题是

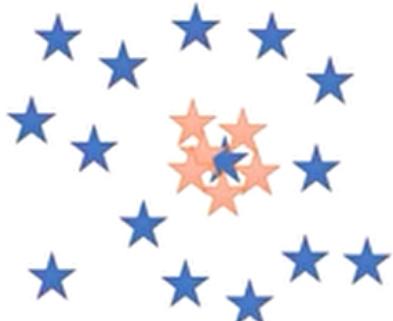
当你 train GAN 的时候你会遇到一个现象叫做，Mode Collapse

Mode Collapse 的意思是说你的 real data 的 distribution 是比较大的，但是你 generate 出来的 example，它的 distribution 非常的小。

Mode Collapse

Training with too many iterations

★ : real data
☆ : generated data



举例来说，你在做二次元人物生成的时候，如果你 update 的 iteration 太多，你得到的结果可能会某一张特定的人脸开始蔓延，变得到处都是这样，但它这些人脸，其实是略有不同的，有的比较偏黄，有的比较偏红，但是他们都是看起来就像是同一张人脸。也就是说你今天产生出来的 distribution 它会越来越小，而最后会发现同一张人脸不断的反复出现，这个 case，叫做 Model collapse。

Mode Dropping

那有另外一个 case 比 mode collapse 稍微轻微一点叫做 Mode dropping。

意思是说你的 distribution 其实有很多个 mode，假设你 real distribution 是两群，但是你的 generator 只会产生同一群而已，他没有办法产生两群不同的 data。

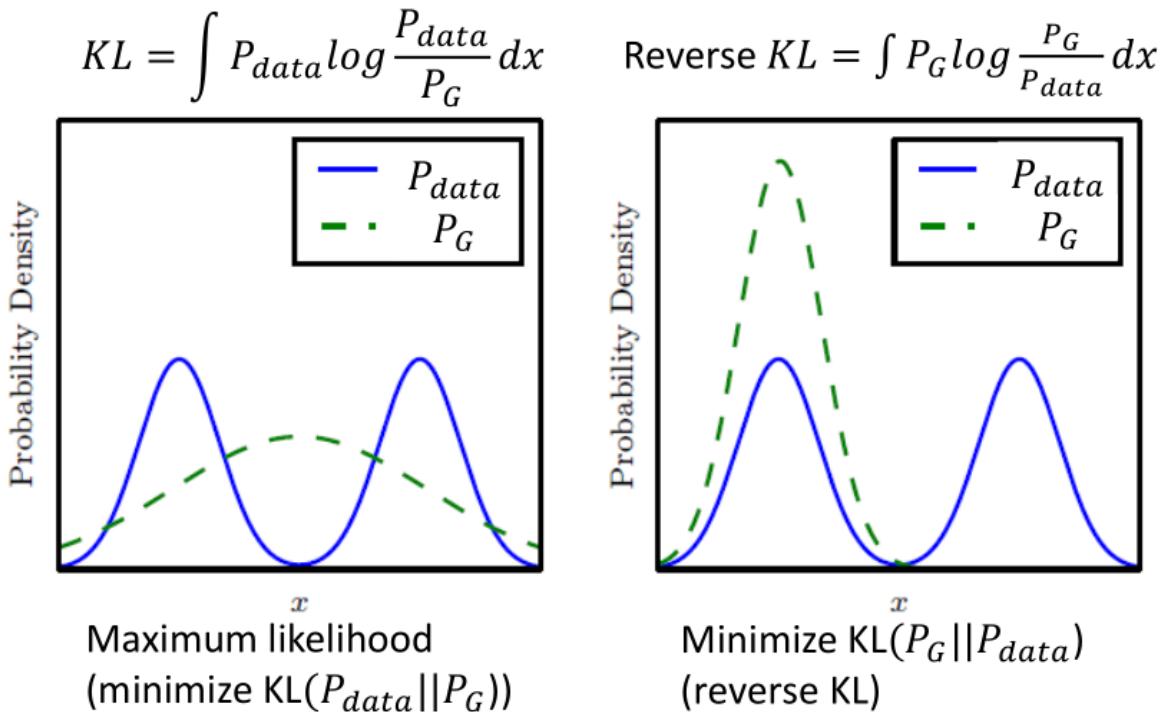
举例来说，你可能 train 一个人脸产生的系统

你在 update 一次 generator 参数以后，产生的 image，他没有产生黄皮肤的人，他只有产生肤色比较偏白的人；但是你 update 一次，它就变成产生黄皮肤的人，就没产生白皮肤的人；再 update 一次，它就变成产生黑皮肤的人。他每次都只能产生某一种肤色的人。

那为什么会发生这种现象呢？

一个远古的猜测是也许是因为我们 divergence 选得不好。

Flaw in Optimization?



如果今天你的 data 的 distribution 是蓝色的分布，你的 generator 的 distribution，它只能有一个 mixture，它是绿色的虚线分布。

如果你选不同的 divergence，你最后 optimize 的结果，最后选出来可以 minimize divergence 的那个 generator distribution，会是不一样的。

假设你用 maximum likelihood 的方法，去 minimize KL divergence，那你的 generator 最后认为最好的那个 distribution 长左边这个样子。

假设你的 generator distribution 长的是这个样子，你从它里面去 sample data，你 sample 在 mixture-mixture 之间，结果反而会是差的。

所以这个可以解释为什么，过去没有 GAN 的时候，我们是在 minimize KL divergence，我们是在 maximize likelihood，我们产生的图片会那么模糊。

也许就是因为我们产生的 distribution 是这个样子的，我们在 sample 的时候其实并不是真的在 data density 很高的地方 sample，而是会 sample 到 data density 很低的地方，所以这地方就对应到模糊的图片。

那有人就说，如果你觉得是 KL divergence 所造成的，那如果换别的 divergence，比如说你换 reverse KL divergence。

那你就会发现说，对 generator 来说最好的 distribution 是完全跟某个 mode 一模一样，就因为如果你看这个 reverse KL divergence 的式子，

你就会发现说，对它来说，如果他产生出来的 data 是蓝色 distribution 没有涵盖它的 penalty 比较大，所以如果你今天选择的是 reverse KL divergence，那你的那个 generator，它就会选择集中在某一个 mode 就好，而不是分散在不同的 mode。

而我们传统的 GAN 的那个 js divergence，它比较接近 reverse KL divergence，这也许解释了为什么你 train GAN 的时候，会有 mode collapse 或者是 mode dropping 的情形。

因为对你的 generator 来说，产生这种 mode collapse 或 mode dropping 的情形其实反而是比较 optimal 的。

所以今天 fGAN 厉害的地方就是，如果你觉得是 js divergence 的问题，你可以换 KL divergence。但结果就是，换不同的 divergence，mode dropping 的 case 状况还是一样，所以看起来不是 mode dropping 或 mode collapse 的问题，并不完全是选择不同的 divergence 所造成的。

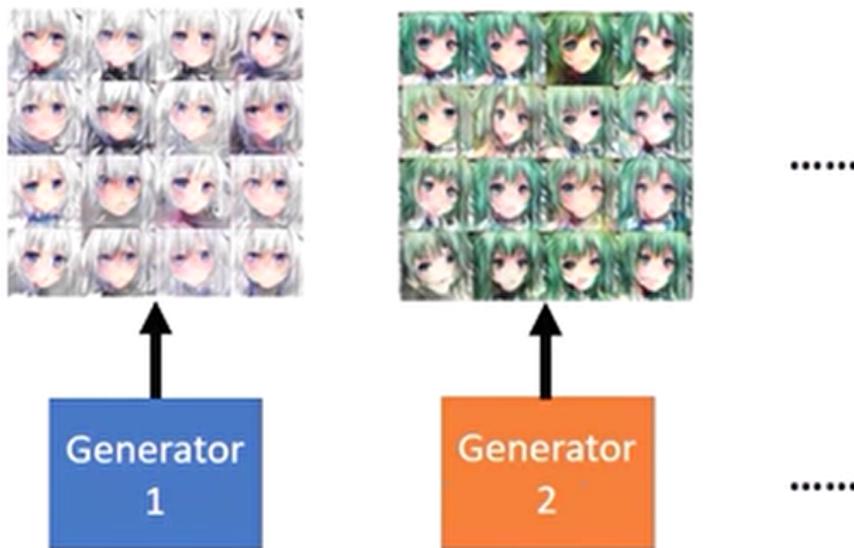
那你可能会问说，那我要怎么解决 mode collapse 的问题呢？

你很可能会遇到 mode collapse 的问题，你的 generator 可能会产生出来的图通通都是一样的。

那要怎么避免这个情形呢？就是做 Ensemble

Outlook: Ensemble

Train a set of generators: $\{G_1, G_2, \dots, G_N\}$
To generate an image
Random pick a generator G_i
Use G_i to generate the image



什么意思呢？今天要你产生25张图片，你就 train 25个 generator

然后你的每一个 generator 也许它都 mode collapse，但是对使用者来说，使用者并不知道你有很多个 generator，那所以你产生出来的结果，看起来就会 diverse。这是一个我觉得最有效可以避免 mode collapse 的方法。

Tips for Improving GAN

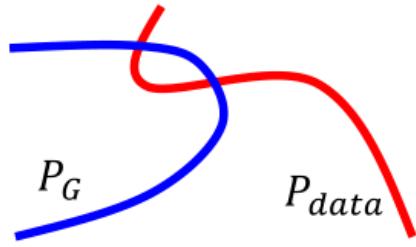
JS divergence is not suitable

In most cases, P_G and P_{data} are not overlapped.

1. The nature of data

Both P_{data} and P_G are low-dim manifold in high-dim space.

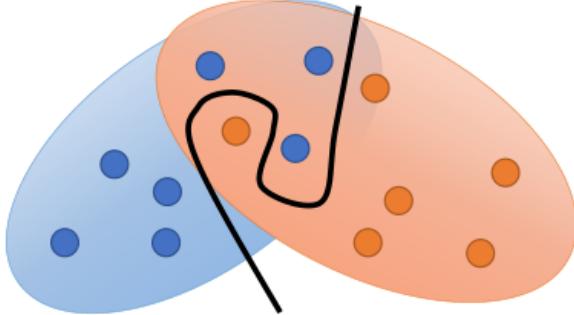
The overlap can be ignored.



2. Sampling

Even though P_{data} and P_G have overlap.

If you do not have enough sampling



最原始的 GAN，他量的是 generated data 跟 real data 之间的 JS divergence。但是用 JS divergence 来衡量的时候，其实有一个非常严重的问题。

你的 generator 产生出来的 data distribution，跟你的 real data 的 distribution，往往是没有任何重叠的。

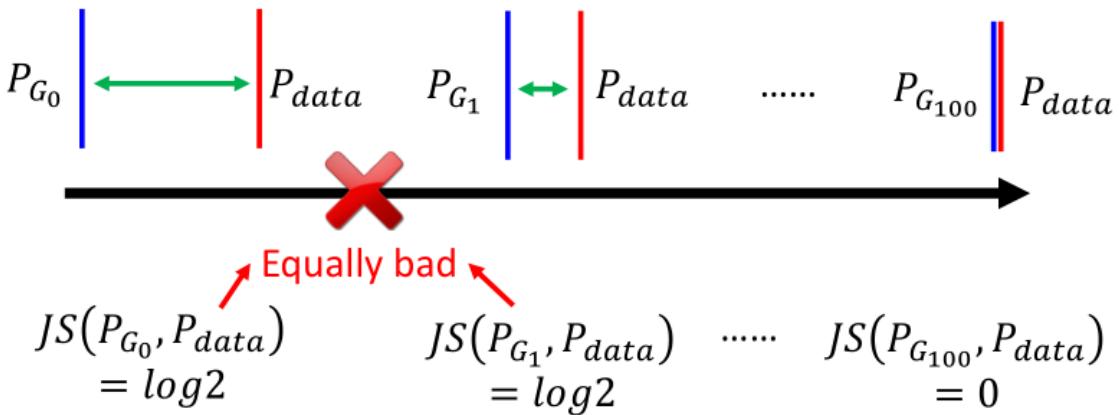
为什么 generate 出来的 data，跟 real 的 data，往往是没有任何重叠的呢？

一个理由是，data 本质上的问题。因为我们通常相信 image 实际上在高维空间中的分布，其实是低维的一个 manifold。在一个高维空间中的两个低维的 manifold，它们的 overlap 的地方几乎是完全可以忽略的，你有两条曲线，在一个二维的平面上，他们中间重叠的地方几乎是可以忽略的。

从另外一个角度，我们实际上在衡量 P_G 跟 P_{data} 的 divergence 的时候，我们是先做 sample，我们从两个 data distribution 里面做一些 sample 得到两堆 data，再用 discriminator 去量他们之间的 divergence。

那所以我们现在就算你的 P_G 跟 P_{data} 这两个 distribution 是有 overlap 的，但是你是先从这两个 distribution 里面做一些 sample，而且 sample 的时候，你也不会 sample 太多。也就是从红色 distribution sample 一些点，从蓝色 distribution 再 sample 一些点。这两堆点，它们的 overlap 几乎是不会出现的，除非你 sample 真的很多，不然这两堆点其实完全就可以视为是两个没有任何交集的 distribution。所以就算本质上 P_{data} 跟 P_G 有 overlap，但你在量 divergence 的时候，你是 sample 少量的 data 出来才量 divergence，那在你 sample 出来的少量 data 里面， P_G 跟 P_{data} ，看起来就是没有重合的。

What is the problem of JS divergence?



JS divergence is $\log 2$ if two distributions do not overlap.

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

→ Same objective value is obtained. → Same divergence

当 PG 跟 Pdata 没有重合的时候，你用 JS divergence 来衡量 PG 跟 Pdata 之间的距离，会对你 training 的时候，造成很大的障碍。

因为 JS divergence 它的特性是：如果两个 distribution 没有任何的重合，算出来就是 $\log 2$ ，不管这两个 distribution 实际上是不是有接近，只要没有重合，没有 overlap，算出来就是 $\log 2$ 。

所以假设你的 Pdata 是红色这一条线，虽然实际上 G1 其实是比 G0 好的，因为 G1 产生出来的 data，其实相较于 G0 更接近 real data distribution。但从 JS divergence 看起来，G1 和 G0 是一样差的，除非说现在你的 G100 跟 Pdata 完全重合，这时候 JS divergence，算出来才会是 0。

只要没有重合，他们就算是非常的靠近，你算出来也是 $\log 2$ 。所以这样子会对你的 training 造成问题。

因为我们知道说我们实际上 training 的时候，generator 要做的事情就是想要去 minimize 你的 divergence。

你用 discriminator 量出 divergence，量出 JS divergence，或其他 divergence 以后，generator 要做的事情是 minimize 你的 divergence。那对 generator 来说，PG0 跟 PG1 他们其实是一样差的。所以对 generator 来说，他根本就不会把 PG0 update 成 PG1。所以你没有办法把 PG0 update 到 PG1，你最后也没有办法 update 到 PG100，因为在 PG0 的地方就卡住了，他没有办法 update 到 PG1。

所以你今天是用 JS divergence 来衡量两个 distribution，而恰好这两个 distribution 又没有太多重叠，他们重叠几乎可以无视的时候，你会发现，你 train 起来是有问题的。

从另外一个直觉的方向来说，为什么今天只要两个 distribution 没有重合，他们算出来的 loss，他们量出来的 divergence 就会一样。

因为你想想看，我们今天实际上在量 JS divergence 的时候，我们做的事情是什么？

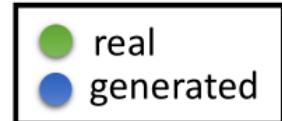
我们有两群 data，把它视为是两个 class，learn 一个 discriminator，你用 minimize cross entropy 当成你的 loss function，去分别出这两组 data 之间的差异。但假设你 learn 的是一个 binary 的 classifier，其实只要这两堆 data，没有重合，它的 loss 就是一样的。

因为假设这两堆 data 没有重合，binary 的 classifier，假如它 capacity 是无穷大，它就可以分辨这两堆 data。在这两堆 data 都可以分辨的前提下，你算出来的 loss，其实会是一样大或者是一样小的。在 train binary classifier 的时候，你 train 到最后得到的那个 loss，或是 objective value 其实就是你的 JS divergence。

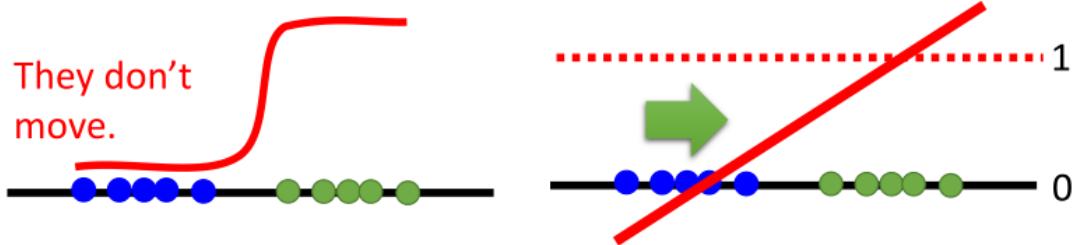
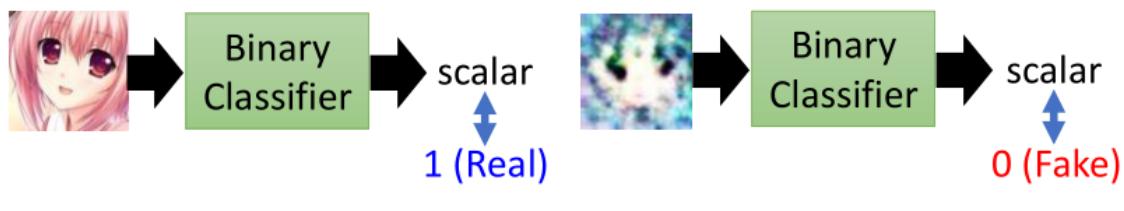
今天如果你的 binary 的 classifier，在G1这个case和G2这个case，它都可以完全把两堆 data 分开。它算出来的 objective 都是一样大，它算出来的 loss 都是一样小的。那意味着，你量出来的 divergence，就是这样。

Least Square GAN (LSGAN)

Least Square GAN (LSGAN)



- Replace sigmoid with linear (replace classification with regression)



在原始的 GAN 里面，当你 train 的是一个 binary classifier 的时候，你会发现，你是比较难 train 的。

用另外一个直观的方法来说明

这个 binary classifier 会给蓝色的点 0 分，绿色的点 1 分。我们知道我们的 binary classifier 它的 output 是 sigmoid function，所以它在接近 1 这边特别平，它在接近 0 这边特别平。

那你 train 好这个 classifier 以后，本来我们期待，train 一个 generator，这个 generator 会带领这些蓝色的点，顺着这个红色的线的 gradient，就 generator 会顺着 discriminator 给我们的 gradient，去改变它的 generated distribution。

所以我们本来是期待 generator，会顺着这个红色线的 gradient，把蓝色的点往右移。但实际上你会发现，这些蓝色的点是不动的，因为在这蓝色的点附近的 gradient 都是 0。

如果你今天是 train 一个binary 的 classifier，它的 output 有一个 sigmoid function 的话，他在蓝色的点附近，它是非常平的，你会发现说他的微分几乎都是 0，你根本就 train 不动它。

所以你真的直接 train GAN，然后 train 一个 binary classifier 的话，你很容易遇到这样子的状况。

过去的一个解法是说，不要把那个 binary classifier train 的太好。

因为如果你 train 的太好的话，它把这些蓝色的点，都给他 0，这边就会变得很平，绿色点都给它 1，就会变得很平。不要让它 train 的太好，不要 update 太多次，让它在这边仍然保有一些斜率。

那这样的问题就是，什么叫做不要 train 的太好，你就会很痛苦，你搞不清楚什么叫做不要 train 的太好，你不能够在 train discriminator 的时候太小力，太小力没办法分别 real 跟 fake data；太大力也不行，太大力的话你就会陷入这个状况，你会陷入这个微分是 0，没有办法 train 的状况。

但是什么叫做不要太大力，不要太小力，你就会很难控制。

那在早年还没有我们刚才讲的种种 tip 的时候，GAN 其实不太容易 train 起来，所以你 train 的时候通常就是，你一边 update discriminator，然后你就一边吃饭，然后你就看他 output 的结果，每 10 个 iteration 就 output 一次结果，我要看它好不好，如果发现结果不好的话，就重做这样子。

所以后来就有一个方法，叫做 Least Square GAN (LSGAN)，那 LSGAN 做的事情，就是把 sigmoid 换成 linear。

这样子你就不会有这种在某些地方特别平坦的情形，因为你现在的 output 是 linear 的。

那我们本来是一个 classification problem，现在把 output 换成了 linear 以后呢，它就变成一个 regression 的 problem。

这 regression problem 是说如果是 positive 的 example，我们就让它的值越接近 1 越好，如果是 negative example，我们就让它的值越接近 0 越好。

但其实跟原来 train binary classifier 是非常像的，只是我们把 sigmoid 拆掉，把它变成 linear。

Wasserstein GAN (WGAN)

那今天很多人都会用的一个技术，叫做 WGAN。

WGAN 是什么呢？在 WGAN 里面我们做的事情是我们换了另外一种 evaluation 的 measure 来衡量 Pdata 跟 PG。

我们之前说在原来的 GAN 里面要衡量 Pdata 跟 PG 的差异，用的是 JS divergence。

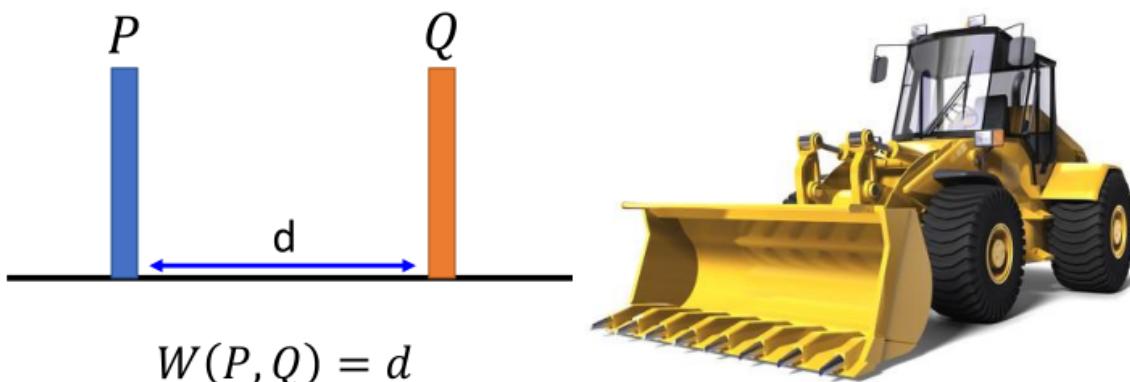
在我们讲 fGAN 的时候我们说，你不一定要用 JS divergence，你其实可以用任何其他的 f divergence，在 WGAN 里面用的是 Earth Mover's Distance 或叫 Wasserstein Distance 来衡量两个 distribution 的差异。它其实不是 f divergence 的一种，所以在 fGAN 那个 table 里面，其实是没有 WGAN 的。

所以这边是另外不一样的方法。同样的地方是，就是你换了一个 divergence 来衡量你的 generated data 和 real data 之间的差异。

Earth Mover's Distance

Considering one distribution P as a pile of earth,
and another distribution Q as the target

The average distance the earth mover has to move
the earth.



那我们先来介绍一下，什么是 Earth Mover's Distance

Earth Mover's Distance 的意思是这样，假设你有两堆 data，这两个 distribution 叫做 P 和 Q，Earth Mover's Distance 的意思是说，你就想象成你是在开一台推土机，那你的土从 P 的地方铲到 Q 的地方。

P 的地方是一堆土，Q 的地方是你准备要把土移过去的位置。然后你看推土机把 P 的土铲到 Q 那边，所走的平均的距离，就叫做 Earth Mover's Distance，就叫做 Wasserstein Distance。

那这个 Wasserstein Distance 怎么定义呢？

如果是在这个非常简单的 case，我们假设 P 的 distribution 就集中在一维空间中的某一个点，Q 的 distribution，也集中在一维空间中的某一个点。

如果你要开一台推土机把 P 的土挪到 Q 的地方去，那假设 P 跟 Q 它们之间的距离是 d ，那你的 Wasserstein Distance，P 这个 distribution 跟 Q distribution 的 Wasserstein Distance 就等于 d 。

但是实际上你可能会遇到一个更复杂的状况



假设你 P distribution 是长这个样子

假设你 Q distribution 是长这个样子

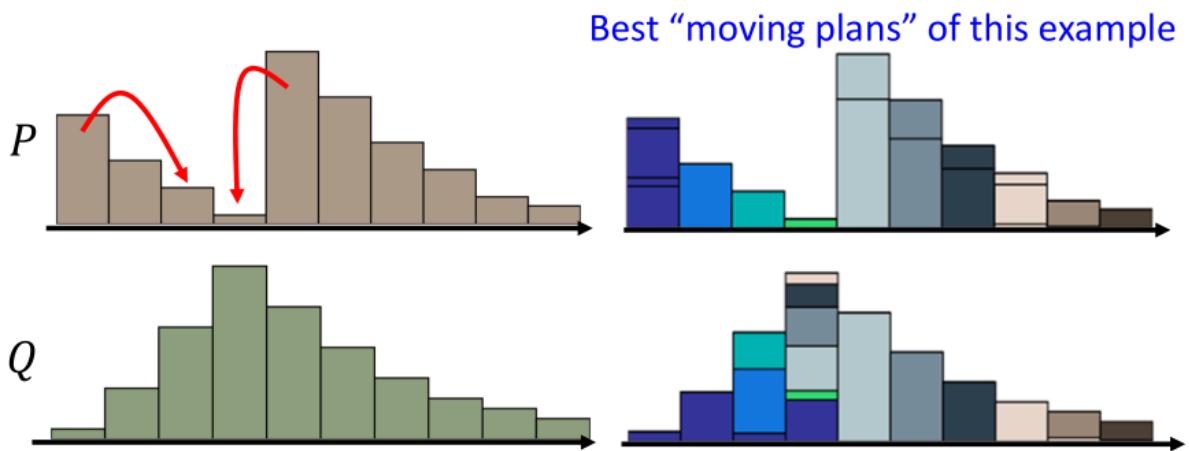
那如果你今天要衡量这两个 distribution 之间的

Earth Mover's Distance，假设你要衡量他们之间的 Wasserstein Distance，怎么办呢？

你会发现当你要把 P 的土铲到 Q 的位置的时候，其实有很多组不同的铲法。推土机走的平均距离是不一样的，这样就会变成说同样的两个 distribution 推土机走的距离不一样，你不知道哪个才是 Wasserstein Distance。

我们说你把某一堆土，铲到你目标的位置去，平均所走的距离就是，Wasserstein Distance。

但现在的问题就是，铲土的方法有很多种，到底哪一个才是 Wasserstein Distance 呢？



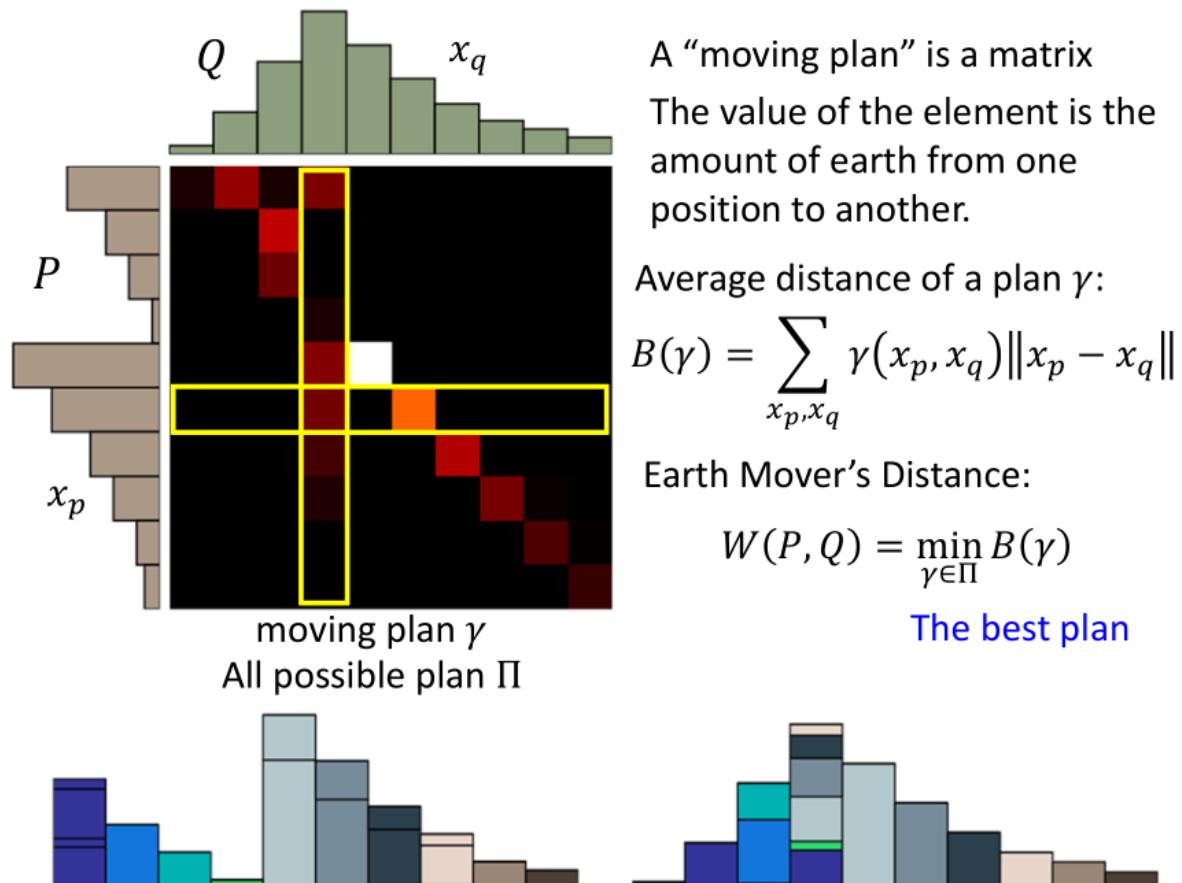
There many possible “moving plans”.

Using the “moving plan” with the smallest average distance to define the earth mover’s distance.

所以今天 Wasserstein Distance 实际上的定义是，穷举所有可能铲土的方法。每种铲土的方法，我们就叫它一个 moving plan，叫它一个铲土的计划。

穷举出所有铲土的计划，有的可能是比较有效的，有的可能是舍近求远的。每一个铲土的计划，推土机平均要走的距离通通都算出来，看哪一个距离最小，就是 Wasserstein Distance。

那今天在这个例子里面，其实最好的铲土的方法，是像这个图上所示这个样子。这样你用这一个 moving plan 来挪土的时候，你的推土机平均走的距离是最短的，这个平均走的距离就是 Wasserstein Distance。



这边是一个更正式的定义，假设你要把这个 P 的图挪到 Q 这边，那首先你要定一个 moving plan。那什么是一个 moving plan 呢？

moving plan 其实你要表现它的话，你可以把它化做是一个 matrix。

今天这个矩阵，就是某一个 moving plan，我们把它叫做 γ

那在这个矩阵上的每一个 element，就代表说，我们要从纵坐标的这个位置挪多少土到横坐标的这个位置。这边的值越亮，就代表说，我们挪的土越多。

实际上你会发现你把 column\row 这些值合起来就会变成 bar 的高度

接下来的事情是，假设给你一个 moving plan 叫做 γ ，你会不会算用这个 moving plan，挪土的时候要走多少距离呢？

$$B(\gamma) = \sum_{x_p, x_q} \gamma(x_p, x_q) \|x_p - x_q\|$$

Wasserstein Distance 或 Earth mover's distance 就是穷举所有可能的 γ ，

看哪一个 γ 算出来的距离最小，这个最小的距离就是 Wasserstein Distance。

Wasserstein Distance，它是一个很神奇的 distance。今天一般的 distance 就是直接套一个公式运算出来你就得到结果，但 Wasserstein Distance，你要算它的话你要解一个 optimization problem，很麻烦。

所以今天给你两个 distribution，要算 Wasserstein Distance 是很麻烦的，因为你要解一个 optimization problem，才算得出 Wasserstein Distance。

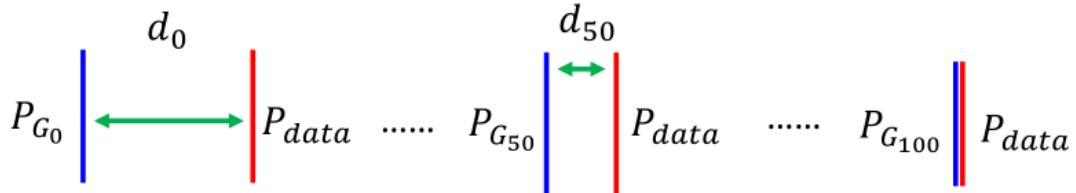
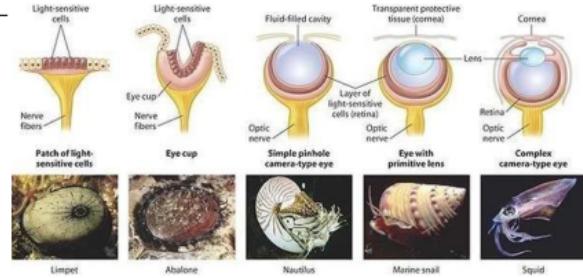
Why Earth Mover's Distance?

Why Earth Mover's Distance?

$$D_f(P_{data} || P_G)$$



$$W(P_{data}, P_G)$$



$$\begin{aligned} JS(P_{G_0}, P_{data}) \\ = \log 2 \end{aligned}$$

$$\begin{aligned} JS(P_{G_{50}}, P_{data}) \\ = \log 2 \end{aligned}$$

$$\begin{aligned} JS(P_{G_{100}}, P_{data}) \\ = 0 \end{aligned}$$

$$\begin{aligned} W(P_{G_0}, P_{data}) \\ = d_0 \end{aligned}$$

$$\begin{aligned} W(P_{G_{50}}, P_{data}) \\ = d_{50} \end{aligned}$$

$$\begin{aligned} W(P_{G_{100}}, P_{data}) \\ = 0 \end{aligned}$$

用 Wasserstein Distance 来衡量两个 distribution 的距离有什么样的好处？

假设你今天是用 JS divergence，这一个 G0 跟 data 的距离，G50 跟 data 之间的距离对 JS divergence 来说，根本就是一样的。

除非你今天可以把 G0 一步跳到 G100，然后让 G100 正好跟 Pdata 重叠，不然 machine 在 update 你的 generator 参数的时候，它根本没有办法从 G0 update 到 G50。因为在这个 case，JS divergence 其实是一样大。

那这个其实就让我想到一个演化上的例子，我们知道说人眼是非常复杂的器官，有人就会想说，凭借着天择的力量，不断的突变，到底怎么可能让生物突然产生人眼呢？那也许天择的假说并不是正确的，但是实际上今天生物是怎么从完全没有眼睛，变到有眼睛呢？并不是一步就产生眼睛，而是通过不断微小的突变的累积，才产生眼睛这么复杂的器官。比如说一开始，生物只是在皮肤上面，产生一些感光的细胞，那通过突变，某些细胞具有感光的能力，也许是做得到的，接下来呢，感光细胞所在的那个皮肤，就凹陷下去，凹陷的好处是，光线从不同方向进来，就不同的感光细胞会受到刺激，那生物就可以判断光线进来的方向。接下来因为有凹洞的关系所就会容易堆灰尘，就在里面放了一些液体，然后免得灰尘跑进去，然后再用一个盖子把它盖起来，最后就变成眼睛这个器官。但是你要直接从皮肤就突然突变，变异产生出眼睛是不可能的，所以就像人，没有办法一下子就长出翅膀变成一个鸟人一样。天择只能做小小的变异，而每一个变异都必须是有好处的，那才能够把这些变异累积起来，最后才能够产生巨大的变异。所以从产生感光细胞，到皮肤凹陷下去，到产生体液把盖子盖起来等等，每一个小小步骤对生物的生存来说都是有利的。所以演化才会由左往右走，生物才会产生眼睛。那如果要产生翅膀可能就比较困难，因为假设你一开始产生很小的翅膀，没有办法飞的话，那就没有占到什么优势。

那对这个 generator 来说也是一样的，它如果说 G50 并没有比 G0 好，你就没有办法从 G0，变到 G50，然后慢慢累积变化变到 G100。

但是如果你用 Wasserstein Distance 就不一样了，因为对 Wasserstein Distance 来说，d50 是比 d0 还要小的，所以对 generator 来说，它就可以 update 参数，把 distribution 从这个地方挪到这个地方，直到最后你 generator 的 output 可以和 data 真正的重合。

WGAN

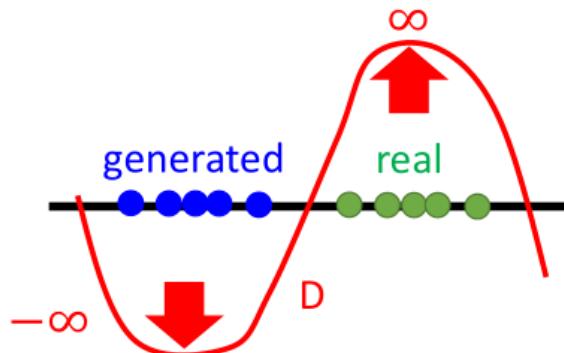
Evaluate wasserstein distance between P_{data} and P_G

$$V(G, D) = \max_{D \in 1-\text{Lipschitz}} \left\{ E_{x \sim P_{data}} [D(x)] - E_{x \sim P_G} [D(x)] \right\}$$

D has to be smooth enough.

Without the constraint, the training of D will not converge.

Keeping the D smooth forces $D(x)$ become ∞ and $-\infty$



我们现在要量 P_G 和 P_{data} 之间的 Wasserstein Distance，我们要怎么去改 discriminator，让他可以衡量 P_G 和 P_{data} 的 Wasserstein Distance 呢？

这边就是直接告诉大家结果，这个推论的过程其实是非常复杂的，这个证明过程其实很复杂，所以我们就直接告诉大家结果，怎么样设计一个 discriminator，它 train 完以后 objective function 的值，就是 Wasserstein Distance。

x 是从 P_{data} 里面 sample 出来的，让它的 discriminator 的 output 越大越好，如果 x 是从 P_G 里面 sample 出来的，让它的 discriminator 的 output 越小越好。

你还要有一个 constrain，discriminator 必须要是一个 1-Lipschitz function

所谓的 1-Lipschitz function 意思是说这个 discriminator，他是很 smooth 的。

为什么这个 1-Lipschitz function 是必要的呢？

你可以说根据证明就是要这么做，算出来才是 Wasserstein Distance，但是你也可以非常的直观地了解这件事。

如果我们不考虑这个 constrain，我们只说要让这些绿色 data 带到 discriminator 里面分数越大越好，这些蓝色 data 带到 discriminator 里面分数越小越好。

那你 train 的时候 discriminator 就会知道说，这边的分数要让他一直拉高一直拉高，这边的分数要让他一直压低一直压低。如果你的这两堆 data 是没有 overlap 的，我们讲过 real data 跟 generated data 很有可能是没有 overlap 的。如果这两堆 data 是没有 overlap 的，今天如果只是 discriminator 一味的要让这些 data 值越来越高，这边 data 值越来越小，它就崩溃了，因为这个 training 永远不会收敛，这个值可以越来越大直到无限大，这个值可以越来越小直到无限小，你的 training 永远不会停止。

所以你必须要有一个额外的限制，你今天的 discriminator，必须要是够平滑的，这样就可以强迫你在 learn 这个 discriminator 的时候，不会 learn 到说这边一直上升，这边一直下降永远不会停下来，那最终还是会停下来的。

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 $w < -c$, $w = -c$

WGAN

Evaluate wasserstein distance between P_{data} and P_G

$$V(G, D) = \max_{D \in \underline{1-Lipschitz}} \{E_{x \sim P_{data}}[D(x)] - E_{x \sim P_G}[D(x)]\}$$

D has to be smooth enough. How to fulfill this constraint?

Lipschitz Function

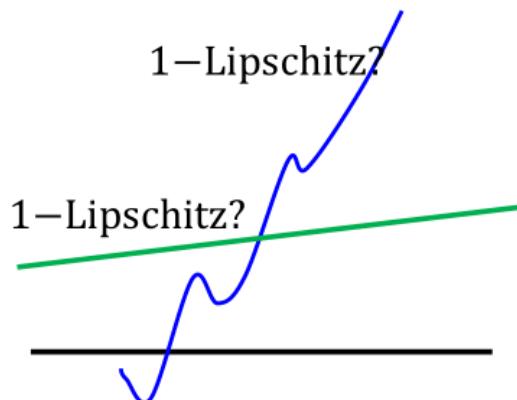
$$\|f(x_1) - f(x_2)\| \leq K \|x_1 - x_2\|$$

Output
change

Input
change

K=1 for "1 – Lipschitz"

Do not change fast



所以这个 Lipschitz function 它的意思到底是什么？

他的意思是说，当你 input 有一个变化的时候，output 的变化不能太大，能够让 input 的差距乘上 K 倍，大于等于 output 的差距。也就是说你 output 的差距不能够太大，不能够比 input 的差距大很多。

当你把 K 设为 1 的时候是 1-Lipschitz function，意味着说，你 output 的变化总是比 input 的变化要小的。

那像蓝色的 function 它变化这么剧烈，它变化这么剧烈，所以那就不是 1-Lipschitz function。那像绿色这个 function，他很平滑，它的变化很小，它在每一个地方，output 的变化都小于 input 的变化，那它就是一个 1-Lipschitz function。

怎么解这个 optimization problem? 如果我们把这个给 discriminator 的 constrain 拿掉，你就用 gradient ascent 去 maximize 它就好了。用 gradient ascent 你就可以 maximize 大括号里面的这个式子。

但现在问题是你的 discriminator 是有 constrain 的，我们一般在做 gradient decent 的时候，我们并不会给我们的参数 constrain，你会发现说如果你要给参数 constrain 的话，在 learning 的时候，还蛮困难的，你会不太清楚应该要怎么做。

所以你今天要给 discriminator constrain 是蛮困难，但实际上到底是怎么做的呢？

在最原始的 WGAN 里面，他的作法就是 weight clipping。

我们用 gradient ascent 去 train 你的 model，去 train 你的 discriminator，但是 train 完之后，如果你发现你的 weight，大过某一个你事先设好的常数 c ，就把它设为 c ，如果小于 $-c$ 就把它设为 $-c$ ，结束。

那他希望说通过这个 weight clipping 的技术，可以让你 learn 出来的 discriminator，它是比较平滑的，因为你限制着它 weight 的大小，所以可以让这个 discriminator 它在 output 的时候，没有办法产生很剧烈的变化，这个 discriminator 可以是比较平滑的。

加了这个限制就可以让他变成 1-Lipschitz function 吗？答案就是不行，因为一开始也不知道要怎么解这个问题，所以就胡乱想一招，能动再说，那我觉得有时候做研究就是这样子嘛，不需要一次解决所有的问题。

在 WGAN 的第一篇原始 paper 里面，他就 propose 说如果 D 是 1-Lipschitz function，那我们就可以量 Wasserstein Distance，但他不知道要怎么真的 optimize 这个 problem，没关系先胡乱提一个挡着先，先 propose，先把 paper publish 出去，再慢慢想这样。

这个是 WGAN 最原始的版本，用的是 weight clipping。那当然它的 performance 不见得是最好的，因为你用这个方法他并没有真的让 D 限制在 1-Lipschitz function，它就只是希望通过这个限制，可以让你的 D 是比较 smooth 的。

Improved WGAN (WGAN-GP)

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

A differentiable function is 1-Lipschitz if and only if it has gradients with norm less than or equal to 1 everywhere.

$$D \in 1 - \text{Lipschitz} \iff \|\nabla_x D(x)\| \leq 1 \text{ for all } x$$

$$V(G, D) \approx \max_D \{E_{x \sim P_{\text{data}}}[D(x)] - E_{x \sim P_G}[D(x)] - \lambda \int_x \max(0, \|\nabla_x D(x)\| - 1) dx\}$$

Prefer $\|\nabla_x D(x)\| \leq 1$ for all x



$$- \lambda E_{x \sim P_{\text{penalty}}} [\max(0, \|\nabla_x D(x)\| - 1)]$$

Prefer $\|\nabla_x D(x)\| \leq 1$ for x sampling from $x \sim P_{\text{penalty}}$

后来就有一个新的招数，不是用 weight clipping，它是用 gradient 的 penalty，那这个技术叫做 improved WGAN 或者是又叫做 WGAN GP。

那 WGAN GP 这边想要讲的是什么呢？一个 discriminator 它是 1-Lipschitz function 等价于，如果你对所有可能的 input x ，都拿去对 discriminator 求他的 gradient 的话，这 gradient 的 norm 总是会小于等于 1 的，这两件事情是等价的。

你不知道怎么限制你的 discriminator，是 1-Lipschitz function，你能不能限制你的 discriminator 对所有的 input x ，去算他的 gradient 的时候，它的 norm，都要小于等于 1 呢？这件事显然是有办法 approximate 的。

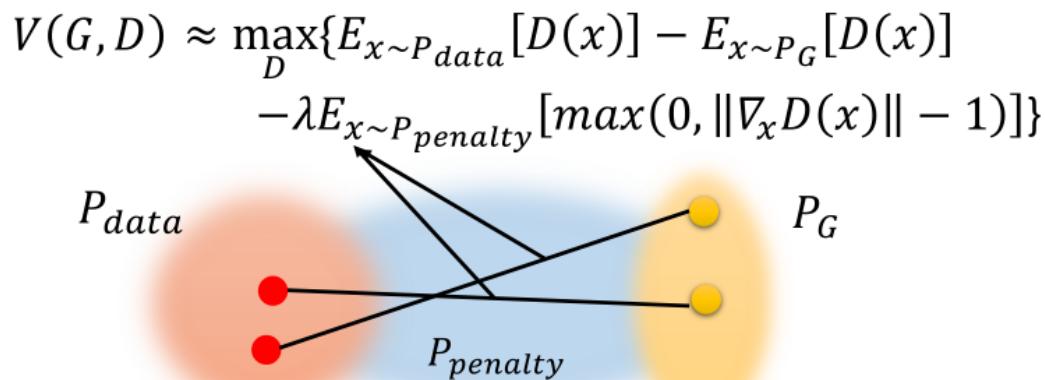
要怎么 approximate 呢？这个 approximate 方法就是说在原来的这项后面，再加一个 penalize 的项，这一项的作用有点像是 regularization，这一项的作用是说，它对所有的 x 做积分，然后取一个 max，也就是说如果这个 gradient norm 小于 1 的话，那就没有 penalty，如果 gradient norm > 1，这一项就会有值，就会有 penalty。

所以今天在 train 这个 discriminator 的时候，今天在 training 的时候会尽量希望这个 discriminator 它的 gradient norm，小于等于 1。

但实际上这么做会有一个问题，因为你不可能对所有的 x 都做积分。我们说一个 function 是 Lipschitz function，它的 if and only if 的条件是对所有的 x 这件事情都要满足。但是你无法真的去 check 说，不管你是在 train 还是在 check 的时候，你都无法做到说 sample 所有的 x ，让他们通通满足这个条件。

x 代表是所有可能的 image，那个 space 这么大，你根本无法 sample 所有的 x ，保证这件事情成立。所以怎么办？

这边做的另外一个 approximation 是说，假设事先定好的 distribution 叫做 P penalty。这个 x 是从 P penalty 那个 distribution sample 出来的，我们只保证说在 P penalty 那个 distribution 里面的 x ，它的 gradient norm 小于等于 1。



“Given that enforcing the Lipschitz constraint everywhere is intractable, enforcing it **only along these straight lines** seems sufficient and experimentally results in good performance.”

Only give gradient constraint to the region between P_{data} and P_G because they influence how P_G moves to P_{data}

这个 P penalty 长什么样子呢？

在 WGAN GP 里面，从 P_{data} 里面 sample 一个点出来，从 P_G 里面 sample 一个点出来，把这两个点相连，然后在这两个点所连成的直线间，做一个 random 的 sample，sample 出来的 x 就当作是从 P penalty sample 出来的。

这个红色的点可以是 P_{data} 里面 sample 出来的任何点，这个黄色的点可以是 P_G 里面 sample 出来的任何点，从这两个点连起来，从这个连线中间去 sample，就是 P penalty。

所以 P penalty 的分布大概就是在 P_G 和 P_{data} 中间，就是蓝色的这个范围。

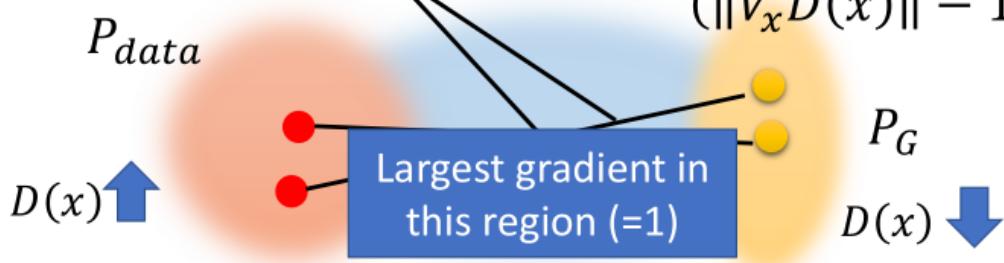
为什么会是这样子呢？为什么我们本来应该对，整个 space 整个 image 的 space 所有的 x 通通去给它 penalty，但为什么只在蓝色的部分给 penalty 是可以的呢？

在原始的 improved WGAN paper 它是这样写的，给每个地方都给它 gradient penalty 是不可能的，就是说实验做起来，这样就是好的这样子。实验做起来，这样看起来是 ok 的。

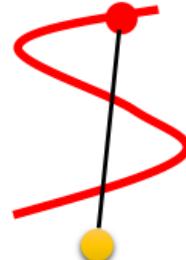
但是你从直觉上也可以了解说这么做是 make sense 的，因为我们今天在 train GAN 的时候，我们不是要 update 那个 generator，然后让 generator 顺着 discriminator 给我们的 gradient 的方向，挪到 P data 的位置去吗。也就是说，我们要让 generator 的这些点慢慢往左移，往左移，在这个例子里面 generator 的点，要慢慢往左移，挪到 P data 的位置去。那所以 generator 在挪动它的位置的时候，在 update 参数的时候，它看的就是 discriminator 的 gradient，所以应该只有在 generator output 的 distribution，跟 real data 的 distribution，中间的连线这个区域，才会真的影响你最后的结果。因为今天这个 PG 是看着这个地方的 gradient，

这个地方的斜率，去 update 它的参数的，所以只有 PG 和 P data 之间的区域你需要去考虑你的 discriminator 的 shape 长什么样子，其他这些地方，反正你的 generator 也走不到，那你就不需要去考虑 discriminator 的 shape 长什么样子。所以我觉得在 PG 和 Pdata 中间做 sample 也是有道理的，也算是 make sense 的。

$$V(G, D) \approx \max_D \{ E_{x \sim P_{data}} [D(x)] - E_{x \sim P_G} [D(x)] \\ - \lambda E_{x \sim P_{penalty}} [\max(0, \|\nabla_x D(x)\| - 1)] \}$$



“Simply penalizing overly large gradients also works in theory, but experimentally we found that this approach converged faster and to better optima.”



接下来要再做另外一个 approximation。

本来我们是希望这个 gradient norm 如果大过 1 给它 penalty，小于 1 不用 penalty。但实际上在 WGAN 的 implementation 里面，我们实际上 training 的时候，我们是希望 gradient 越接近 1 越好，本来理论上我们只需要 gradient < 1，大过 1 给他惩罚，小于 1 没有关系，但实作的时候说，gradient norm 必须离 1 越接近越好。gradient norm > 1 有惩罚，< 1 也有惩罚。为什么会这样呢？在 paper 里面说，实验上这么做的 performance 是比较好的。

当然这个 improved WGAN 也不会是最终的 solution，实际上你很直觉的会觉得，它是有一些问题的。举例来说我这边举一个例子，假设红色的曲线是你的 data，你在 data 上 sample 一个点是红色的，你在黄色的是你的 distribution，这边 sample 一个点，你说把他们两个连起来，然后给这边的这些线 constrain，你不觉得其实是不 make sense 的嘛。

因为如果我们今天照理说，我们只考虑黄色的点，要如何挪到红色的点，所以照理说，我们应该在红色的这个地方，sample 一个点跟黄色是最近的，然后只 penalize 这个地方跟黄色的点之间的 gradient，这个才 make sense 嘛，因为到时候黄色的点，其实它要挪动的话，它也是走到最近的地方，它不会跨过这些已经有红色点的地方跑到这里来。这个是有点奇怪的，我认为他会走这个方向（最近的点），而不是走这样的方向（连线）。所以你 gradient penalty penalize 在（连线）这个地方，是有点奇怪的。

那其实 improved WGAN 后面还有很多其他的变形，大家可以自己找一下

其实像今年的 ICLR 2018，就有一个 improved WGAN 的变形，叫做 improved 的 improved WGAN 这样子，那 improved 的 improved WGAN 他一个很重要的不同是说，它的 gradient penalty 不是只放在 Pdata 跟 PG 之间，他觉得要放在这个红色的区块。

Spectrum Norm

Spectrum Norm

Spectral Normalization → Keep gradient norm smaller than 1 everywhere [Miyato, et al., ICLR, 2018]



刚才 WGAN 什么都是一个一堆 approximation 嘛，spectrum norm 是这样，他 propose 了一个方法，这个方法真的可以限制你的 discriminator 在每一个位置的 gradient norm 都是小于 1 的，本来 WGAN GP 它只是 penalize 某一个区域的 gradient norm < 1，但是 spectrum norm 这个方法可以让你的 discriminator learn 完以后，它在每一个位置的 gradient norm 都是小于 1。

这个也是 ICLR 2018 的 paper，那细节我们就不提。

Algorithm of WGAN

我们看一下怎么从 GAN 改成 WGAN

Algorithm of WGAN

Learning D

Repeat k times

Learning G

Only Once

- In each training iteration:
No sigmoid for the output of D
 - Sample m examples $\{x^1, x^2, \dots, x^m\}$ from data distribution $P_{data}(x)$
 - Sample m noise samples $\{z^1, z^2, \dots, z^m\}$ from the prior $P_{prior}(z)$
 - Obtaining generated data $\{\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^m\}$, $\tilde{x}^i = G(z^i)$
 - Update discriminator parameters θ_d to maximize
 - $\tilde{V} = \frac{1}{m} \sum_{i=1}^m D(x^i) - \frac{1}{m} \sum_{i=1}^m D(\tilde{x}^i)$
 - $\theta_d \leftarrow \theta_d + \eta \nabla \tilde{V}(\theta_d)$
 - Sample another m noise s Weight clipping / Gradient Penalty ... } from the prior $P_{prior}(z)$
- Update generator parameters θ_g to minimize
 - $\tilde{V} = \frac{1}{m} \sum_{i=1}^m \log D(x^i) - \frac{1}{m} \sum_{i=1}^m D(G(z^i))$
 - $\theta_g \leftarrow \theta_g - \eta \nabla \tilde{V}(\theta_g)$

那这边要注意的地方是，在原来的 GAN 里面你的 discriminator 有 sigmoid，有那个 sigmoid 你算出来才会是 JS divergence。

但是在 WGAN 里面，你要把 sigmoid 拆掉，让它的 output 是 linear 的，算出来才会是 Wasserstein Distance。

接下来你在 update 你的 discriminator，在 train 你的 discriminator 的时候呢，要注意一下就是你要加上 weight clipping，或者是加上 gradient penalty，不然这个 training 可能是不会收敛的。

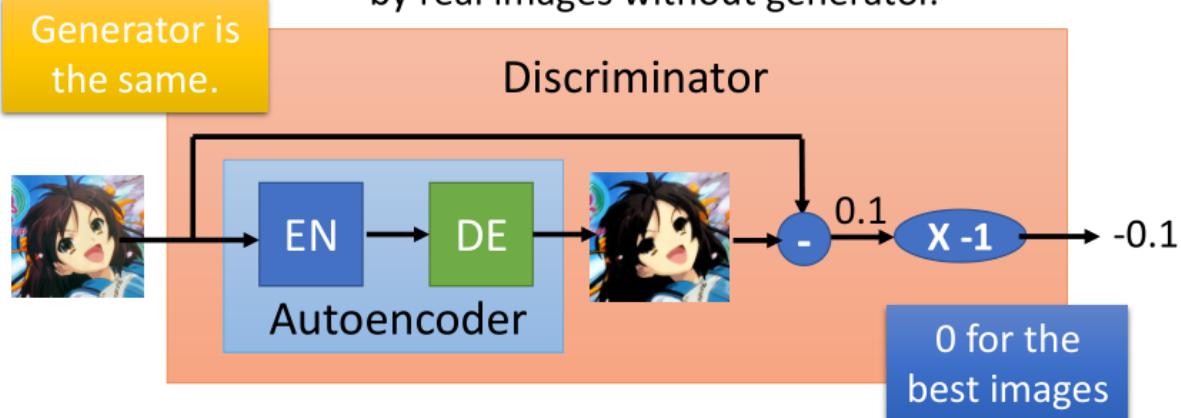
所以你总共只要改 4 个地方，改 objective function、把 sigmoid 拆掉、把 weight clipping 加进去、改一下 generator update 的 objective function，就结束了。

Energy-based GAN (EBGAN)

EBGAN 还有另外一个变形叫做BEGAN，另外一个变形我们不讲。

EBGAN 是什么，EBGAN 他唯一跟一般的 GAN 不同的地方是，它改了 discriminator 的 network 架构。

- Using an autoencoder as discriminator D
 - Using the negative reconstruction error of auto-encoder to determine the goodness
 - **Benefit:** The auto-encoder can be pre-train by real images without generator.



本来 discriminator 是一个 binary 的 classifier，它现在把它改成 auto encoder。

所以 Energy based GAN 的意思就是说，你的 discriminator 是这样，input 一张 image，有一个 encoder，把它变成 code，然后有一个 decoder 把它解回来，接下来你算那个 auto encoder 的 reconstruction error，把 reconstruction error 乘一个负号，就变成你的 discriminator 的 output。

也就是说这个 energy based GAN 它的假设就是，假设某一张 image 它可以被 reconstruction 的越好，它的 reconstruction error 越低，代表它是一个 high quality 的 image，如果它很难被 reconstruct，它的 reconstruction error 很大，代表它是一个 low quality 的 image。

那这种 EBGAN 他到底有什么样的好处呢？

我觉得他最大的好处就是，你可以 pre-train 你的 discriminator。

auto encoder 在 train 的时候，不需要 negative example，你在 train 你的 discriminator 的时候，它是一个 binary classifier，你需要 negative example，这个东西无法 pre trained。你没有办法只拿 positive example 去 train 一个 binary classifier。

所以这会造成的问题是一开始你的 generator 很弱，所以它 sample 出来的 negative example 也很弱，用很弱的 negative example 你 learn 出来就是一个很弱的 discriminator，那 discriminator 必须要等 generator 慢慢变强以后，你要 train 很久，才会让 discriminator 变得比较厉害。

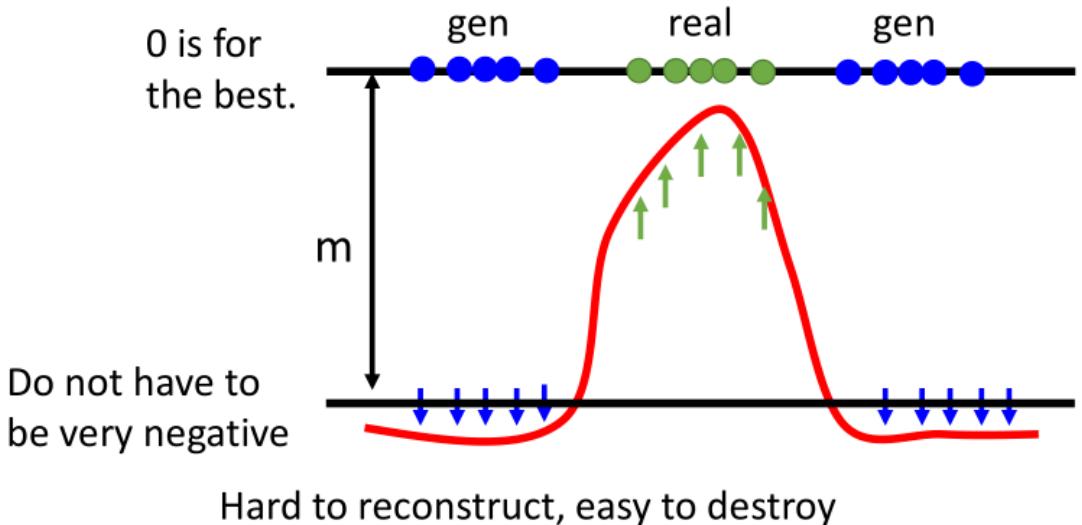
但是 energy base GAN 就不一样，discriminator 是一个 auto encoder，auto encoder 是可以 pre trained，auto encoder 不需要 negative example，你只要给它 positive example，让它去 minimize reconstruction error 就好了。

所以你真的要用 energy based GAN 的时候，你要先 pre-train 好你的 discriminator，先拿你手上的那些 real 的 image，去把你的 auto encoder 先 train 好，所以你一开始的 discriminator，会很强，所以因为你的 discriminator 一开始就很强，所以你的 generator 一开始就可以 generate 很好的 image。

所以如果你今天是用 energy base GAN，你会发现说你前面几个 epoch，你就可以还蛮清楚的 image。那这个就是 energy base GAN 一个厉害的地方。

EBGAN

Auto-encoder based discriminator
only gives limited region large value.



那 energy based GAN 实际上在 train 的时候，还有一个细节你是要注意的，就是今天在 train energy based GAN 的时候，你要让 real example 它的 reconstruction error 越小越好。

但是要注意，你并不是要让 generated example 的 reconstruction error 越大越好，为什么？

因为建设是比较难的，破坏是比较容易的。reconstruction error 要让它变小很难，因为，你必须要 input 一张 image 把它变成 code，再 output 同样一张 image，这件事很难，但是如果你要让 input 跟 output 非常不像，这件事太简单了，input 一张 image，你要让它 reconstruction error 很大，不就 output 一个 noise 就很大了吗？

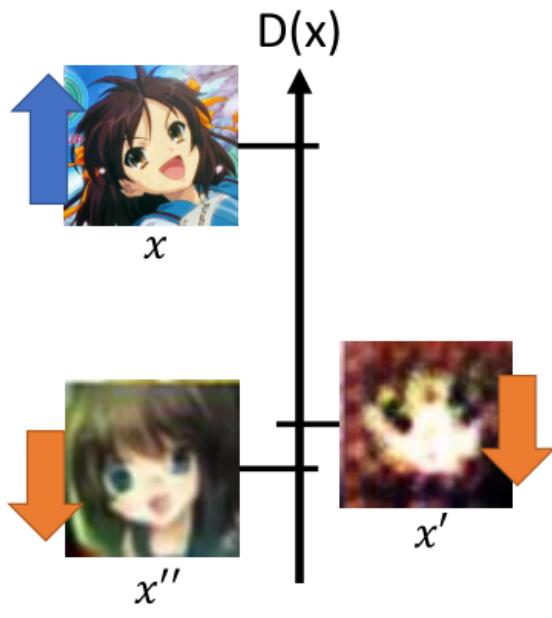
所以如果你今天太专注于说要 maximize 这些 generated image 的 reconstruction error，那你的 discriminator，到时候就学到说看到什么 image 都 output 那个 noise，都 output noise，故意把它压低，这个时候你的 discriminator 的 loss 可以把它变得很小，但这个不是我们要的。

所以实际上在做的时候，你会设一个 margin 说，今天 generator 的 reconstruction loss 只要小于某一个 threshold 就好，当然 threshold 这个 margin 是你要手调的。

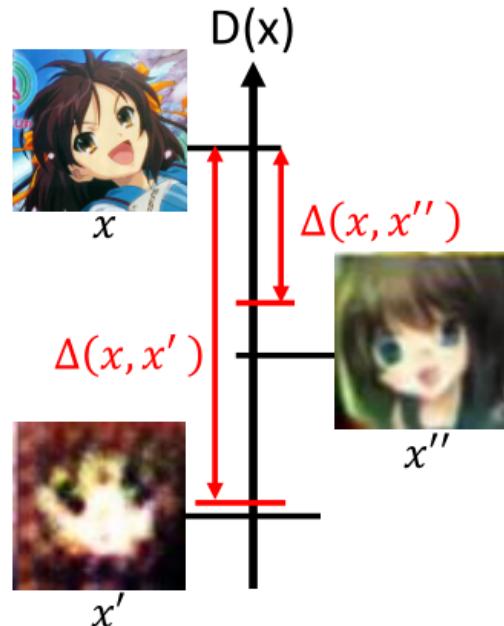
这个 margin 意思是说 generator loss 只要小于 margin 就好，不用再小，小于 margin 就好，不用让它再更小。

Loss-sensitive GAN (LSGAN)

WGAN



LSGAN



其实还有另外一个东西也是有用到 margin 的概念，叫做Loss-Sensitive GAN。它也是LSGAN 这样，我们有一个 least square GAN，这边还有一个 Loss-Sensitive GAN。

那 Loss-Sensitive GAN 它也有用到 margin 的概念。我们之前在做 WGAN 的时候是说，如果是 positive example，就让他的值越大越好，negative example，就让他的值越小越好。

但是假设你有些 image 其实已经很 realistic，你让它的值越小越好，其实也不 make sense 对不对，所以今天在 LSGAN 里面它的概念就是，他加了一个叫做 margin 的东西。

就是你需要先有一个方法，去 evaluate 说你现在产生出来的 image 有多好，可能是把你产生出来的 image 呢，如果今天这个 x double prime 跟 x 已经很像了，那它们的 margin 就小一点，如果 x prime 跟 x 很不像，它们 margin 就大一点，所以你会希望 x prime 的分数被压得很低， x double prime 的分数只要压低过 margin 就好，不需要压得太低。

Feature Extraction by GAN

讲一下用 GAN 做 Feature Extraction 有关的事情，我想先跟大家讲的是 InfoGAN。

我们知道 GAN 会 random input 一个 vector，然后 output 一个你要的 object。我们通常期待 input 的那个 vector 它的每一个 dimension 代表了某种 specific 的 characteristic，你改了 input 的某个 dimension，output 就会有一个对应的变化，然后你可以知道每一个 dimension 它做的事情是什么。

但是实际上未必有那么容易，如果真的 train 了一个 GAN 你会发现，input 的 dimension 跟 output 的关系，观察不到什么关系。

InfoGAN

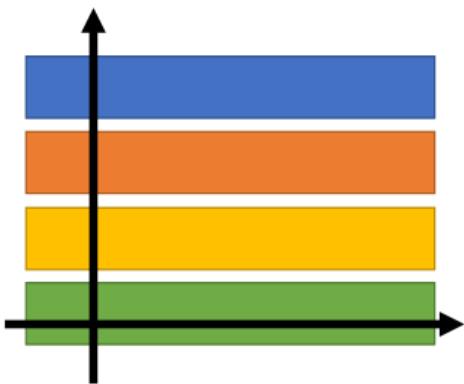
(The colors represents the characteristics.)

Regular
GAN

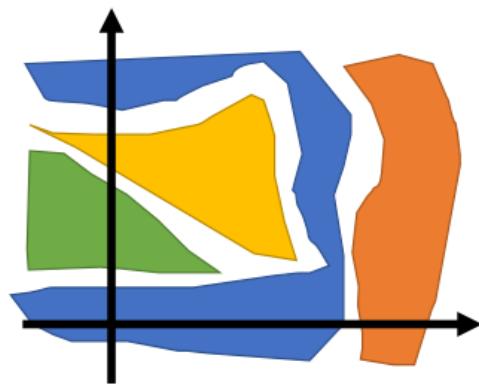
7	7	7	7	7	7	7	7	7	7	7
0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	7	7	7	7	7	7
9	9	9	9	9	9	9	9	9	9	9
8	8	8	5	8	8	5	5	5	8	8

Modifying a specific dimension,
no clear meaning

What we expect



Actually ...



这边这是一个文献上的例子，假设 train 了一个 GAN，这个 GAN 做的事情，是手写数字的生成，你会发现你改了 input 的某一个维度，对 output 来说，横轴代表改变了 input 的某一个维度，output 的变化是不太出规律的。比如说这边的 7，突然中间写了一横也不知道是什么意思，搞不清楚说，改变了某一维度到底对 output 的结果，起了什么样的作用。

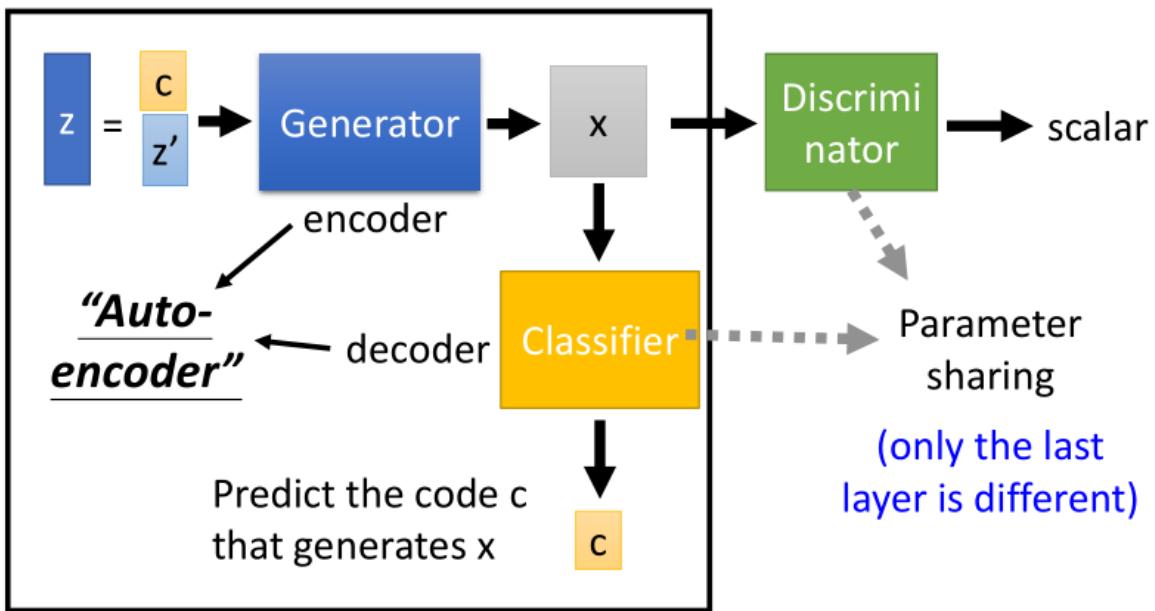
为什么会这样呢，现在这个投影片上这个二维平面，代表 generator input 的 random vector 的 space，假设 input 的 vector 只有两维，我们通常期待在这个 latent 的 space 上面，不同的 characteristic 的 object 它的分布是有某种规律性的，我们这边用不同的颜色来代表，假设你在这个区块，你使用这个区块的 vector 当作 generator 的 input，它 output 会有蓝色的特征，这个区块会有橙色的特征，这个区块会有黄色的特征，这个区块会有绿色的特征。本来的假设是这些不同的特征，他们在 latent space 上的分布是有某种规律性的，但是实际上也许它的分布是非常不规则的。

我们本来期待如果改变了 input vector 的某一个维度，它就会从绿色变到黄色再变到橙色再变到蓝色，它有一个固定的变化，但是实际上也许它的分布长的这个样子，也许 latent space 跟你要生成的那个 object 之间的关系，是非常复杂的。所以当你改变某一个维度的时候，你从蓝色变到绿色再变到黄色又再变回蓝色，你就觉得说不知道在干嘛。

InfoGAN

所以 InfoGAN 就是想要解决这个问题。

在 InfoGAN 里面你会把 input 的 vector 分成两个部分，比如说假设 input vector 是二十维，就说前十维把它叫作 c ，后十维我们把它叫作 z' 。



在 InfoGAN 里面你会 train 一个 classifier，这个 classifier 工作是、看 generator 的 output，然后决定根据这个 generator 这个 output 去预测现在 generator input 的 c 是什么。

所以这个 generator 吃这个 vector，产生了 x ，classifier 要能够从 x 里面反推原来 generator 输入的 c 是什么样的东西。

在这个 InfoGAN 里面，你可以把 classifier 视为一个 decoder，这个 generator 视为一个 encoder。这个 generator 跟 classifier 合起来，可以把它看作是一个 Autoencoder。它跟传统的 Autoencoder 做的事情是正好相反的，所以这边加一个双引号，因为我们知道传统的 Autoencoder 做的事情是给一张图片，他把它变成一个 code，再把 code 解回原来的图片，但是在 InfoGAN 里面这个 generator 和 classifier 所组成的 Autoencoder 做的事情，跟我们所熟悉的 Autoencoder 做的事情，是正好相反的。

在 InfoGAN 里面，generator 是一个 code 产生一张 image，然后 classifier 要根据这个 image 决定那个原来的 code 是什么样的东西。

当然如果只有 train generator 跟 classifier 是不够的，这个 discriminator 一定要存在，为什么 discriminator 一定要存在，假设没有 discriminator 的话，对 generator 来说，因为 generator 想要帮助 classifier，让 classifier 能够成功的预测， x 是从什么样的 c 弄出来的，如果没有 discriminator 的话，对 generator 来说，最容易让 classifier 猜出 c 的方式就是直接把 c 贴在这个图片上，然后 classifier 只要知道他去读这个图片中的数值，就知道 c 是什么，那这样就完全没有意义，所以这边一定要有一个 discriminator。discriminator 会检查这张 image 看起来像不像一个 real image。如果 generator 为了要让 classifier 猜出 c 是什么，而刻意地把 c 原本的数值，我们期待是 generator 根据 c 所代表的信息，去产生对应的 x ，但 generator 它可能就直接把 c 原封不动贴到这个图片上，但是如果只是把 c 原封不动贴到这个图片上，discriminator 就会发现这件事情不对，发现这看起来不像是真的图片，所以 generator 并不能够直接把 c 放在图片里面，透露给 classifier。

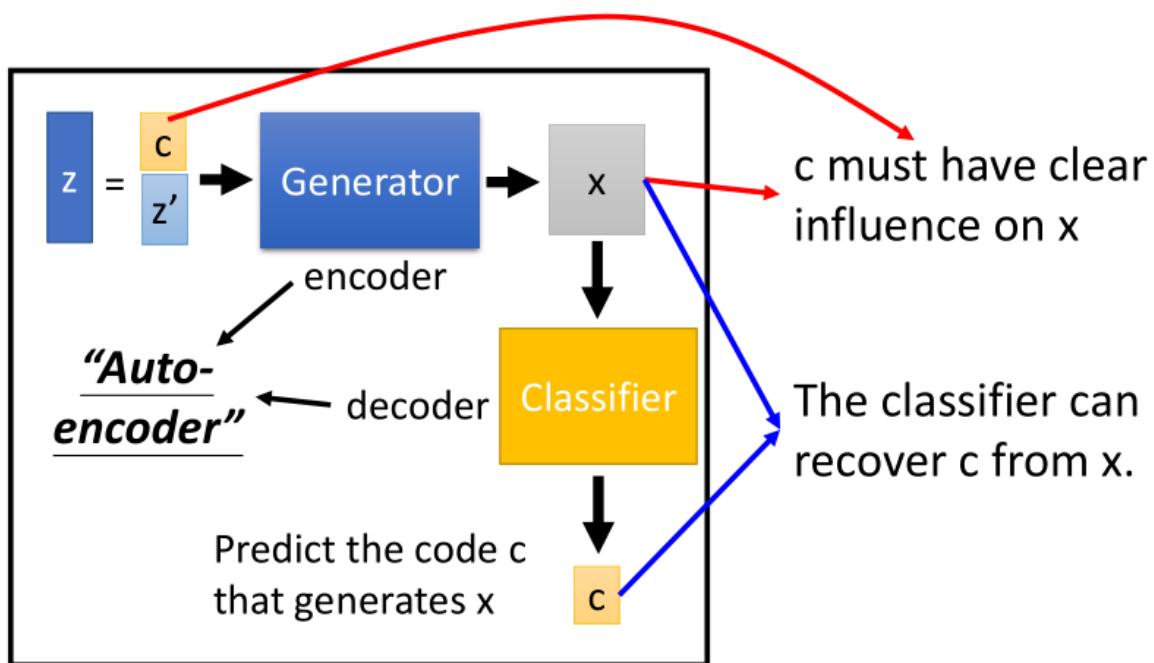
InfoGAN 在实作上 discriminator 跟 classifier 往往会 share 参数，因为他们都是吃同样的 image 当作 input，不过他们 output 的地方不太一样，一个是 output scalar，一个是 output 一个 code、vector，不过通常你可以让他们的一些参数是 share 的。

加上这个 classifier 会有什么好处，我们说我们刚才想要解的问题就是，input feature 它对 output 的影响不明确这件事，InfoGAN 怎么解决 input feature 对 output 影响不明确这件事呢？

InfoGAN 的想法是这个样子：为了让 classifier 可以成功地从 image x 里面知道原来的 input c 是什么，generator 要做的事情就是，他必须要让 c 的每一个维度，对 output 的 x 都有一个明确的影响，如果 generator 可以学到 c 的每一个维度对 output 的 x 都有一个非常明确的影响，那 classifier 就可以轻易地根据 output 的 image 反推出原来的 c 是什么。如果 generator 没有学到让 c 对 output 有明确影响，就像刚看到那个例子，改了某一个 dimension 对 output 影响是很奇怪的，classifier 就会无法从 x 反推原来的 c 是什么。

在原来的 InfoGAN 里面他把 input z 分成两块，一块是 c 一块是 z' ，这个 c 他代表了某些特征，也就是 c 的每一个维度代表图片某些特征，他对图片是会有非常明确影响，如果你是做手写数字生成，那 c 的某一个维度可能就代表了那个数字笔画有多粗，那另外一个维度可能代表写的数字的角度是什么。

其实在 generator input 里面还有一个 z' ，在原始的 InfoGAN 里面他还加一个 z' ， z' 代表的是纯粹随机的东西，代表的是那些无法解释的东西。



那有人可能会问这个 c 跟 z' 到底是怎么分的，我们怎么知道前十维这个 feature 是应该对 output 有影响的，后十维这个 feature 他是属于 z' ，对 output 的影响是随机的呢？

你不知道，但是这边的道理是这个 c 并不是因为它代表了某些特征，而被归类为 c ，而是因为他被归类为 c 所以他会代表某些特征。

并不是因为他代表某些特征所以我们把他设为 c ，而是因为他被设为 c 以后根据 InfoGAN 的 training，使得他必须具备某种特征，希望大家听得懂我的意思。

(a) Varying c_1 on InfoGAN (Digit type)

(b) Varying c_1 on regular GAN (No clear meaning)

(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation)

(d) Varying c_3 from -2 to 2 on InfoGAN (Width)

这个是文献上的结果，第一张图是 learn 了 InfoGAN 以后，他改了 c 的第一维，然后发现什么事，发现 c 的第一维代表了 digit，这个很神奇，改了 c 的第一维以后，更动他的数值就从 0 跑到 9。这个 b 是原来的结果，他有做普通的 GAN，output 结果是很奇怪的。改第二维的话你产生的数字的角度就变了，改第三维的话你产生的数字就从笔划很细变到笔划很粗，这个就是 InfoGAN。

VAE-GAN

另外一个跟大家介绍的叫作 VAE-GAN，VAE-GAN 是什么，VAE-GAN 可以看作是用 GAN 来强化 VAE，也可以看作是用 VAE 来强化 GAN。

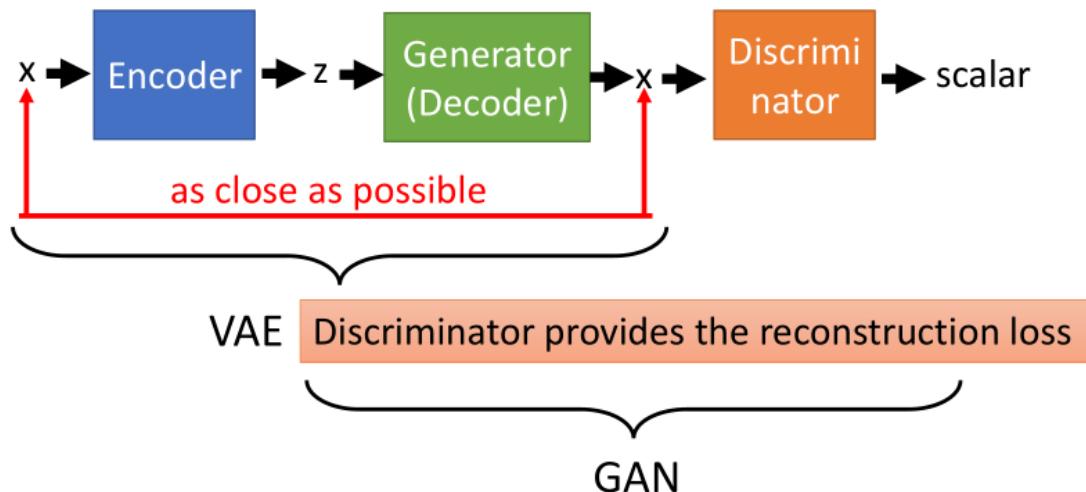
VAE 在 ML 有讲过的就是 Autoencoder 的变形，这个 Variational Autoencoder，Autoencoder 大家都很熟，就是有一个 encoder，有一个 decoder，encoder input x output 是一个 z ，decoder 吃那个 z output 原来的 x ，你要让 input 跟 output 越近越好。

这个是 Variational Autoencoder，如果是 Variational Autoencoder 你还会给 z 一个 constrain，希望 z 的分布像是一个 Normal Distribution，只是在这边图上没有把它画出来。

Anders Boesen, Lindbo Larsen, Søren Kaae
Sønderby, Hugo Larochelle, Ole Winther, "Autoencoding
beyond pixels using a learned similarity metric", ICML. 2016

VAE-GAN

- Minimize reconstruction error
- Minimize reconstruction error
- Discriminate real, generated and reconstructed images
- z close to normal
- Cheat discriminator



那 VAE-GAN 的意思是在原来的 encoder decoder 之外 再加一个 discriminator。这个 discriminator 工作就是 check 这个 decoder 的 output x 看起来像不像是真的。

如果看前面的 encoder 跟 decoder 他们合起来是一个 Autoencoder，如果看后面的这个 decoder 跟 discriminator，在这边 decoder 他扮演的角色其实是 generator，我们看这个 generator 跟 discriminator 他们合起来是一个 GAN。

在 train VAE-GAN 的时候，一方面 encoder decoder 要让这个 Reconstruction Error 越小越好，但是同时 decoder 也就是这个 generator 要做到另外一件事，他会希望他 output 的 image 越 realistic 越好。如果从 VAE 的角度来看，原来我们在 train VAE 的时候，是希望 input 跟 output 越接近越好，但是对 image 来说，如果单纯只是让 input 跟 output 越接近越好，VAE 的 output 不见得会变得 realistic，他通常产生的东西就是很模糊的，如果你实际做过 VAE 生成的话，因为根本不知道怎么算 input 跟 output 的 loss，如果 loss 是用 L1 L2 norm，那 machine 学到的东西就会很模糊，那怎么办，就加一个 discriminator，你就会迫使 Autoencoder 在生成 image 的时候，不是只是 minimize Reconstruction Error，同时还要产

生比较 realistic image, 让 discriminator 觉得是 realistic, 所以从 VAE 的角度来看, 加上 discriminator 可以让他的 output 更加 realistic。

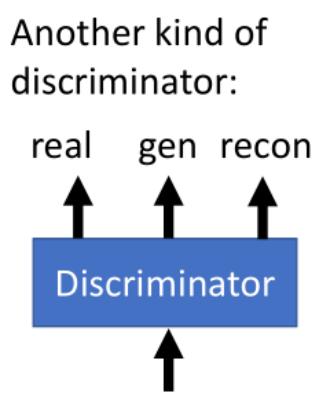
如果从 GAN 的角度来看, 前面这边 generator 加 discriminator 合起来, 是一个 GAN。然后在前面放一个 encoder, 从 GAN 的角度来看, 原来在 train GAN 的时候, 你是随机 input 一个 vector, 你希望那个 vector 最后可以变成一个 image, 对 generator 来说他从来没有看过真正的 image 长什么样子, 他要花很多力气, 你需要花很多的时间去调参数, 才能够让 generator 真的学会产生真正的 image, 知道 image 长什么样子。但是如果加上 Autoencoder 的架构, 在学的时候 generator 不是只要骗过 discriminator, 他同时要 minimize Reconstruction Error, generator 在学的时候他不是只要骗过 discriminator, 他还有一个目标, 他知道真正的 image 长什么样子, 他想要产生一张看起来像是 encoder input 的 image, 他在学习的时候有一个目标不是只看 discriminator 的 feedback, 不是只看 discriminator 传来那边的 gradient, 所以 VAE-GAN 学起来会比较稳一点。

在 VAE-GAN 里面, encoder 要做的事情就是要 minimize Reconstruction Error, 同时希望 z 它的分布接近 Normal Distribution, 对 generator 来说他也是要 minimize Reconstruction Error, 同时他想要骗过 discriminator, 对 discriminator 来说他就要分辨一张 image 是真正的 image 还是生成出来的 image, 跟一般的 discriminator 是一样的。

Algorithm

假如你对 VAE-GAN 有兴趣的话这边也是列一下 algorithm, 这 algorithm 是这样, 有三个东西, 一个 encoder 一个 decoder 一个 discriminator, 他们都是 network, 所以先 initialize 他们的参数。

Algorithm



- Initialize En, De, Dis
- In each iteration:
 - Sample M images x^1, x^2, \dots, x^M from database
 - Generate M codes $\tilde{z}^1, \tilde{z}^2, \dots, \tilde{z}^M$ from encoder
 - $\tilde{z}^i = En(x^i)$
 - Generate M images $\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^M$ from decoder
 - $\tilde{x}^i = De(\tilde{z}^i)$
 - Sample M codes z^1, z^2, \dots, z^M from prior $P(z)$
 - Generate M images $\hat{x}^1, \hat{x}^2, \dots, \hat{x}^M$ from decoder
 - $\hat{x}^i = De(z^i)$
 - Update En to decrease $\|\tilde{x}^i - x^i\|$, decrease $KL(P(\tilde{z}^i | x^i) || P(z))$
 - Update De to decrease $\|\tilde{x}^i - x^i\|$, increase $Dis(\tilde{x}^i)$ and $Dis(\hat{x}^i)$
 - Update Dis to increase $Dis(x^i)$, decrease $Dis(\tilde{x}^i)$ and $Dis(\hat{x}^i)$

这 algorithm 是这样说的, 我们要先 sample M 个 real image, 接下来再产生这 M 个 image 的 code, 把这个 code 写作 \tilde{z} , 把 x 丢到 encoder 里面产生 \tilde{z} , 他们是真正的 image 的 code, 接下来再用 decoder 去产生 image, 你把真正的 image 的 code z , 把 z 丢到 decoder 里面, decoder 就会产生 reconstructed image, 就边写作 \tilde{x} , \tilde{x} 是 reconstructed image。

接下来 sample M 个 z , 这个现在 z 不是从某一张 image 生成的, 这边这个 z 是从一个 Normal Distribution sample 出来的。

用这些从 Normal Distribution sample 出来的 z , 再丢到 encoder 里面再产生 image 这边叫做 \hat{x} 。

现在总共有三种 image，一种是真的从 database 里面 sample 出来的 image，一个是从 database sample 出来的 image 做 encode，变成 $z \tilde{}$ 以后再用 decoder 再还原出来，叫做 $x \tilde{}$ ，还有一个是 generator 自己生成的 image，他不是看 database 里面任何一张 image 生成的，他是自己根据一个 Normal Distribution sample 所生成出来的 image，这边写成 $x \hat{}$ 。

再来在 training 的时候，你先 train encoder，encoder 目标是什么，他要 minimize Autoencoder Reconstruction Error，所以要让真正的 image x_i 跟 reconstruct 出来的 image $x \tilde{}$ 越接近越好，encoder 目的是什么，他希望原来 input 的 image 跟 reconstructed image， x 跟 $x \tilde{}$ 越接近越好，这第一件他要做的事，第二件他要做的事情是他希望这个 x 产生出来的 $z \tilde{}$ 跟 Normal Distribution 越接近越好，这是本来 VAE 要做的事情。

接下来 decoder 要做的事情是他同时要 minimize Reconstruction Error，他有另外一个工作是他希望他产生出来的东西，可以骗过 discriminator，他希望他产生出来的东西，discriminator 会给他高的分数。

现在 decoder 其实会产生两种东西，一种是 $x \tilde{}$ ，是 reconstructed image，通常 reconstructed image 就是会看起来整个结构比较好，但是比较模糊，这个 $x \tilde{}$ 产生一个 reconstructed image，这个 reconstructed image 就到 discriminator 里面，分数要越大越好。那把你 $x \hat{}$ ，就是 machine 自己生出来的 image，丢到 discriminator 里面，希望值越大越好。

最后轮到 discriminator，discriminator 要做的事情是如果是一个 real image 给他高分，如果是 faked image，faked image 有两种，一种是 reconstruct 出来的，一种是自己生成出来的，都要给他低分。这是 VAE-GAN 的作法。

我们之前看到 discriminator 都是一个 Binary Classifier，他就是要鉴别一张 image 是 real 还是 fake，其实还有另外一个做法是，discriminator 其实是一个三个 class 的 classifier，给他一张 image 他要鉴别他是 real 还是 generated 还是 reconstructed。因为 generated image 跟 reconstructed image 他们本质上看起来颇不像的，在右边的 algorithm 里面，是把 generated 跟 reconstructed 视为是同一个 class，就是 fake 的 class，都当作 fake 的 image。

但是这个做法是把 generate 出来的 image 跟 reconstruct 出来的 image，视为两种不同的 image，discriminator 必须去学着鉴别这两种的差异。generator 在学的时候，有可能产生 generated image，他也有可能产生 reconstructed image，他都要试着让这两种 image discriminator 都误判，认为他是 real 的 image，这个是 VAE-GAN，VAE-GAN 是去修改了 Autoencoder。

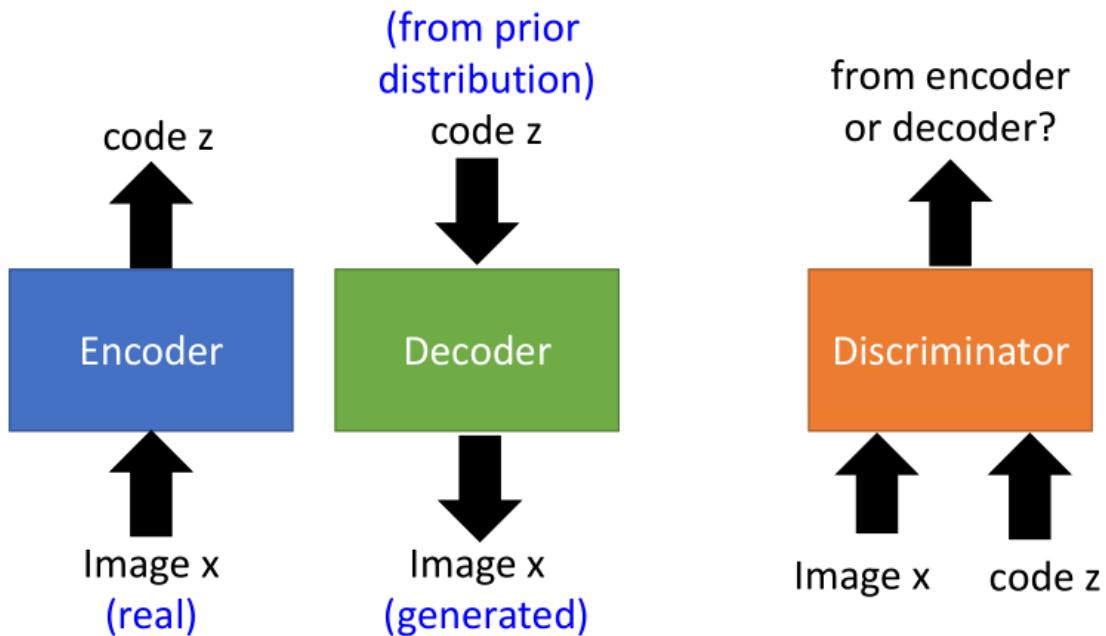
BiGAN

其实 BiGAN 还有另外一个技术，跟他非常地相近，其实不只是相近根本是一模一样，叫做 ALI，BiGAN 跟 ALI 如果没记错的话是同时发表在 ICLR 2017 上面，有什么差别？就是没有任何差别，不同的两群人居然想出了一模一样的方法，而且我发现 BiGAN 的 citation 比较高，我想原因就是因为他有 GAN，然后 ALI 他没有用到 GAN 这个字眼，citation 就少一点。

还有另外一个技术叫做 BiGAN，BiGAN 他也是修改了 Autoencoder。

我们知道在 Autoencoder 里面，有一个 encoder，有一个 decoder，在 Autoencoder 里面是把 encoder 的 output 丢给 decoder 去做 reconstruction。

但是在 BiGAN 里面不是，在 BiGAN 里面就有一个 encoder 有一个 decoder，但是他们的 input output 不是接在一起的。



encoder 吃一张 image 他就变成一个 code, decoder 是从一个 Normal Distribution 里面 sample 一个 z 出来丢进去, 他就产生一张 image。

但是我们并不会把 encoder 的输出丢给 decoder, 并不会把 decoder 的输出丢给 encoder, 这两个是分开的。

有一个 encoder 有一个 decoder, 这两个是分开的那他们怎么学呢, 在 Autoencoder 里面可以学是因为收集了一大堆 image 要让 Autoencoder 的 input 等于 Autoencoder output, 现在 encoder 跟 decoder 各自都只有一边, encoder 只有 input 他不知道 output target 是什么, decoder 他只有 input, 他不知道 output 的 image 应该长什么样子, 怎么学这个 encoder 跟 decoder.

这边的做法是再加一个 discriminator, 这个 discriminator 他是吃 encoder 的 input 加 output, 他吃 decoder 的 input 加 output, 他同时吃一个 code z 跟一个 image x , 一起吃进去, 然后它要做的事情是鉴别 x 跟 z 的 pair 他们是从 encoder 来的还是从 decoder 来的, 所以它要鉴别一个 pair 他是从 encoder 来的还是从 decoder 来的。

我们先讲一下 BiGAN 的 algorithm 然后再告诉你为什么, BiGAN 这样做到底是什么样的道理。

Algorithm

Initialize encoder En, decoder De, discriminator Dis

In each iteration:

- Sample M images x^1, x^2, \dots, x^M from database
- Generate M codes $\tilde{z}^1, \tilde{z}^2, \dots, \tilde{z}^M$ from encoder
 - $\tilde{z}^i = En(x^i)$
- Sample M codes z^1, z^2, \dots, z^M from prior $P(z)$
- Generate M codes $\tilde{x}^1, \tilde{x}^2, \dots, \tilde{x}^M$ from decoder
 - $\tilde{x}^i = De(z^i)$
- Update Dis to increase $Dis(x^i, \tilde{z}^i)$, decrease $Dis(\tilde{x}^i, z^i)$
- Update En and De to decrease $Dis(x^i, \tilde{z}^i)$, increase $Dis(\tilde{x}^i, z^i)$

现在有一个 encoder 有一个 decoder 有一个 discriminator，这个跟刚才讲 VAE-GAN 虽然一致，不过这边 BiGAN 的运作方式跟 VAE-GAN 是非常不一样。

每一个 iteration 里面，你会先从 database 里面 sample 出 M 张真的 image，然后把这些真的 image 丢到 encoder 里面，encoder 会 output code 就得到了 M 组 code，得到了 M 个 z tilde，这个是用 encoder 生出来的东西。

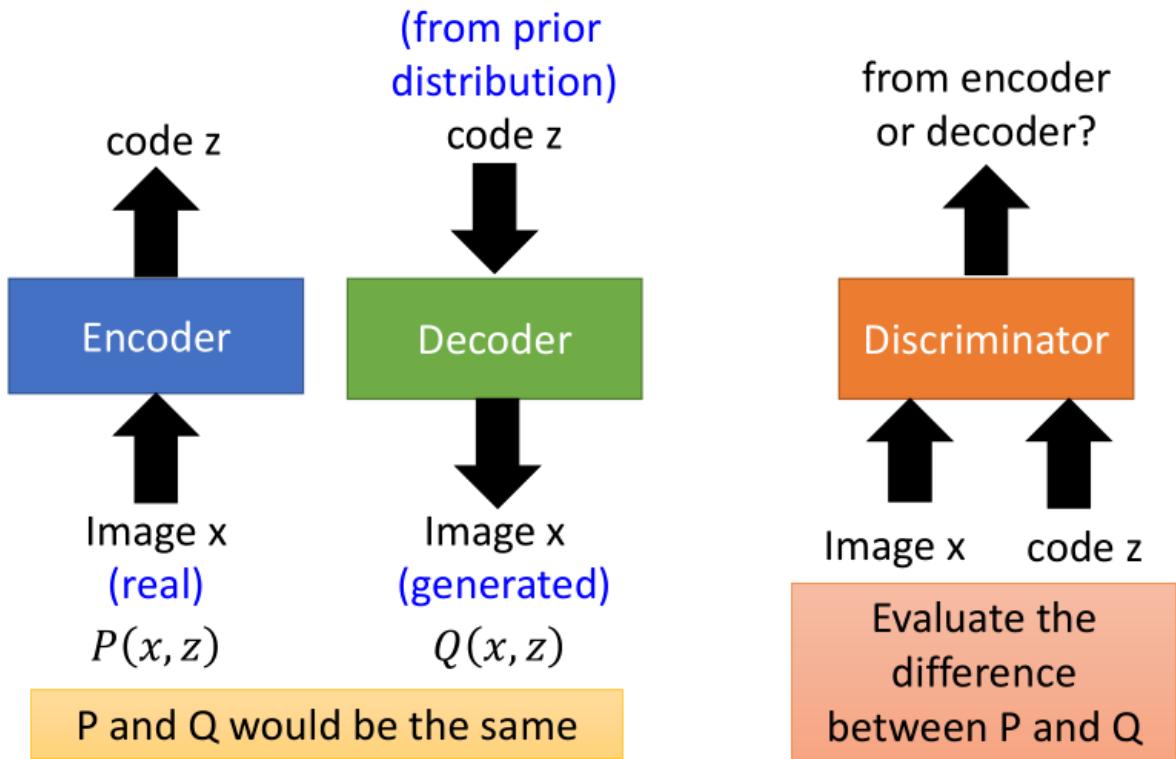
接下来用 decoder 生东西，sample M 个 code，这个从一个 Normal Distribution sample 出来，把这些 code 丢到 decoder 里面，decoder 就产生他自己生成的 image x tilde，所以这边没有 tilde 的东西都是真的，有 tilde 的东西都是生成的。

这边有 M 个 real image 生成出 M 个 code，这边有 M 个 code 生成出 M 个 image。

接下来要 learn 一个 discriminator，discriminator 工作是给他 encoder 的 input 跟 output 给他高分，给它 decoder 的 input 跟 output 给它低分，如果这个 pair 是 encoder 的 input 跟 output，给他高分，如果这个 pair 是 decoder 的 input 跟 output，就给他低分。

有人会问为什么是 encoder 会给高分，decoder 会给低分，其实反过来讲你也会问同样的问题，不管是你要让 encoder 高分 decoder 低分，还是 encoder 低分 decoder 高分，是一样的，意思是完全一模一样的，learn 出来结果也会是一样的，它并没有什么差别，只是选其中一个方法来做就是了。

encoder 跟 decoder 要做的事情就是去骗过 discriminator。如果 discriminator 要让 encoder 的 input output 高分，decoder 的 input output 低分，encoder decoder 他们就要连手起来，让 encoder 的 input output 让 discriminator 给它低分，让 decoder 的 input output，discriminator 给他高分。所以 discriminator 要做什么事，encoder 跟 decoder 就要连手起来，去骗过 discriminator 就对了，到底要让 encoder 高分还是 decoder 高分，是无关紧要的。这个是 BiGAN 的 algorithm。



Optimal encoder and decoder:
 $\text{En}(x') = z' \rightarrow \text{De}(z') = x' \quad \text{For all } x'$
 $\text{De}(z'') = x'' \rightarrow \text{En}(x'') = z'' \quad \text{For all } z''$

BiGAN 这么做到底是什么道理，我们知道 GAN 做的事情，这个 discriminator 做的事情就是在 evaluate 两组 sample 出来的 data，到底他们接不接近。

我们讲过从 real database 里面 sample 一堆 image 出来，用 generator sample 一堆 image 出来，一个 discriminator 做的事情其实就是在量这两堆 image 的某种 divergence 到底接不接近。

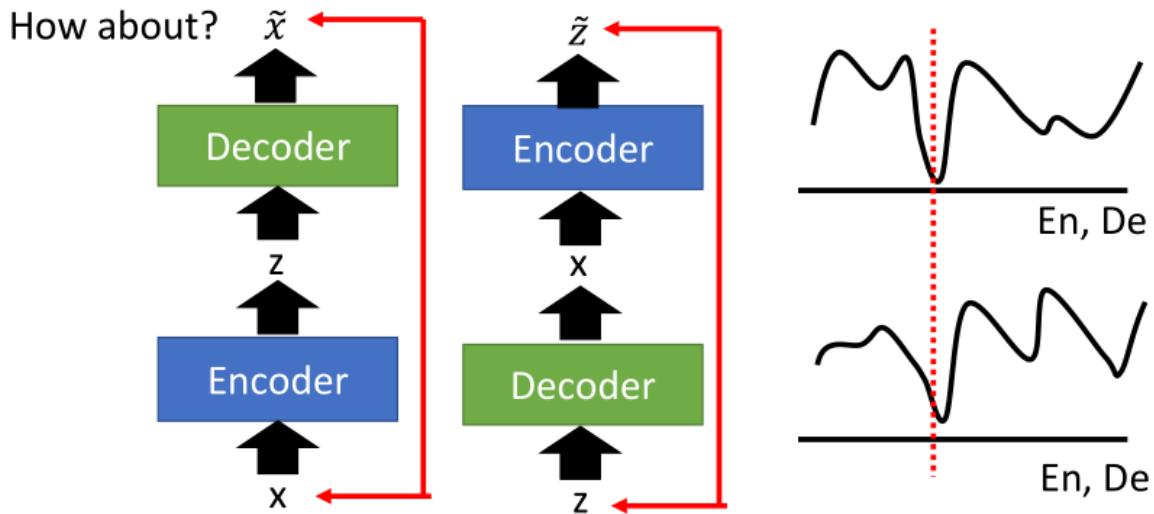
这个道理是一样的，可以把 encoder 的 input output 合起来，当作是一个 Joint Distribution，encoder input 跟 output 合起来有一个 Joint Distribution 写成 $P(x, z)$ ，decoder input 跟 output 合起来也是另外一个 Joint Distribution $Q(x, z)$ 。discriminator 要做的事情就是去衡量这两个 distribution 之间的差异，然后希望透过 discriminator 的引导让这两个 distribution 之间越近越好。

在原来的 GAN 里面，我们希望 generator 生成出来的 data distribution，跟 P data 越接近越好，这边的道理是完全一模一样的，discriminator 希望 encoder input output 所组成的 Joint Probability，跟 decoder input output 所组成的 Joint Probability，这两个 Data Distribution 越接近越好，所以 eventually 在理想的状况下，应该会学到 P 这个 distribution，也就是 encoder 的 input 跟 output 所组成的 distribution，跟 Q 这个 distribution，这两个 distribution，他们是一模一样。

如果最后他们 learn 到一模一样的时候，会发生什么事情？你可以轻易的知道如果 P 跟 Q 的 distribution，是一模一样的，你把一个 image x' 丢到 encoder 里面让它给你一个 code z' ，再把 z' 丢到 decoder 里面让它给你一个 image x' 。 x' 会等于原来的 input x' ，你把 x' 丢进去它会产生 z' ，你把 z' 丢到 decoder 里面，它会产生原来的 x' 。你把 z'' 丢到 decoder 里面让它产生 x'' ，你就把 x'' 丢到 encoder 里面，那它就会产生 z'' 。

所以 encoder 的 input 产生一个 output，再把 output 丢到 decoder 里面会产生原来 encoder 的 input，decoder 给它一个 input 它产生一个 output，再把它的 output 丢到 encoder 里面，它会产生一模一样的 input，虽然说实际上在 training 的时候，encoder 跟 decoder 并没有接在一起，但是透过 discriminator 会让 encoder decoder 最终在理想上达成这个特性。

Optimal encoder and decoder:	$En(x') = z'$	$De(z') = x'$	For all x'
	$De(z'') = x''$	$En(x'') = z''$	For all z''



所以有人会问这样 encoder 跟 decoder 做的事情是不是就好像是 learn 了一个 Autoencoder，这个 Autoencoder input 一张 image 它变成一个 code，再把 code 用 decoder 解回原来一样的 image。再 learn 一个反向的 Autoencoder，所谓的反向的 Autoencoder 的意思是，decoder 吃一个 code 它产生一张 image，再从这个 image 还原回原来的 code。

假设在理想状况下，BiGAN 它可以 learn 到 optimal 的结果，确实会跟同时 learn 这样子一个 encoder 跟 Autoencoder 得到的结果是一样的。

那有人就会问为什么不 learn 这样子一个 encoder 跟一个 inverse Autoencoder 就好了呢，为什么还要引入 GAN，这样听起来感觉上是画蛇添足。

我觉得如果用 BiGAN learn 的话，得到的结果还是会不太一样，这边想要表达的意思是，learn 一个 BiGAN，跟 learn 一个下面这个 Autoencoder，他们的 optimal solution 是一样的，但它们的 Error Surface 是不一样的，如果这两个 model 都 train 到 optimal 的 case，得到的结果会是一样的，但是实际上不可能 train 到 optimal 的 case。BiGAN 无法真的 learn 到 P 跟 Q 的 distribution 一模一样，Autoencoder 无法 learn 到 input 跟 output 真的一模一样，这件事情是不可能发生的，所以不会真的收敛到 optimal solution。

但不是收敛到 optimal solution 的状况下，这两种方法 learn 出来的结果就会不一样，到底有什么不一样，这边没有把文献上的图片列出来，如果你看一下文献上的图片的话，一般的 Autoencoder learn 完以后，input 一张 image 它就是 reconstruct 另外一张 image，跟原来的 input 很像，然后比较模糊，这个大家应该都知道 Autoencoder 就是这么回事。

但是如果用 BiGAN 的话，其实也是 learn 出来了一个 Autoencoder，learn 了一个 encoder 一个 decoder，他们合起来就是一个 Autoencoder。

但是当你把一张 image 丢到这个 encoder，再从 decoder 输出出来的时候，其实你可能会得到的 output 跟 input 是非常不像的。它会比较清晰，但是非常不像，比如说你把一只鸟丢进去，它 output 还是会是一只鸟，但是是另外一只鸟。这个就是 BiGAN 的特性，你可以去看一下它的 paper，如果跟 Autoencoder 比起来，他们的最佳的 solution 是一样的，但是实际上 learn 出来的结果会发现这两种 Autoencoder，就是用这种 minimize Reconstruction Error 方法 learn 了一个 Autoencoder，还是用 BiGAN learn 的 Autoencoder，他们的特性其实是非常不一样。

BiGAN 的 Autoencoder 它比较能够抓到语意上的信息，就像刚才说的你 input 一只鸟，它知道是一只鸟，它 reconstruct 出来的结果，decoder output 也是一只鸟，但是不是同一只鸟，这就是一个还满神奇的结果。

Triple GAN

Triple GAN 里面有三个东西，一个 discriminator 一个 generator，一个 classifier。如果先不要管 classifier 的话，Triple GAN 本身就是一个 Conditional GAN。

Conditional GAN 就是 input 一个东西，output 一个东西，比如说 input 一个文字，然后就 output 一张图片，generator 就是吃一个 condition，这边 condition 写成 Y，然后产生一个 X，它把 X 跟 Y 的 pair 丢到 discriminator 里面，discriminator 要分辨出 generator 产生出来的东西是 fake 的，real database sample 出来的东西就是 true，所以 generator 跟 discriminator 合起来就是一个 Conditional GAN。

这边再加一个 classifier 是什么意思，这边再加一个 classifier 意思是 Triple GAN 是一个 Semi-supervised Learning 的做法。

假设有少量的 labeled data，但是大量的 unlabeled data，也就是说你有少量的 X 跟 Y 的 pair，有大量的 X 跟 Y 他们是没有被 pair 在一起。

所以 Triple GAN 它主要的目标，是想要去学好一个 classifier，这 classifier 可以 input X，然后就 output Y，你可以用 labeled data 去训练 classifier，你可以从有 label 的 data 的 set 里面，去 sample X Y 的 pair，去 train classifier，但是同时也可以根据 generator，generator 会吃一个 Y 产生一个 X，把 generator 产生的 X Y 的 pair，也丢给这个 classifier 去学。它的用意就是增加 training data，本来有 labeled 的 X Y 的 pair 很少，但是有一大堆的 X 跟 Y 是没有 pair 的，所以用 generator 去给他吃一些 Y 让它产生 X，得到更多 X Y 的 pair 去 train classifier。

这个 classifier 它吃 X，然后去产生 Y。

discriminator 会去鉴别这 classifier input 跟 output 之间的关系，看起来跟真正 X Y 的 pair 有没有像。

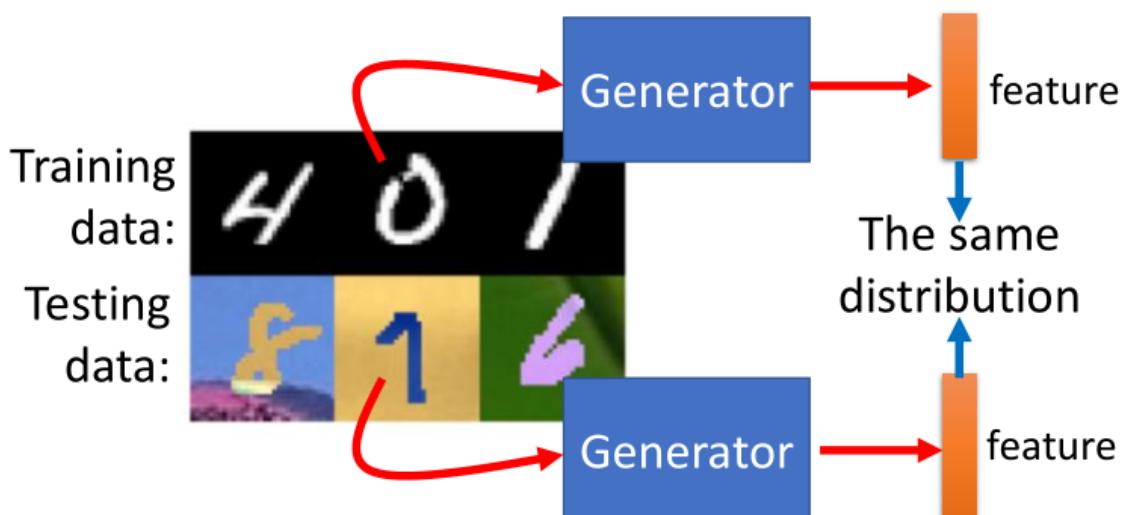
所以 Triple GAN 是一个 Semi-supervised Learning 的做法。

这边就不特别再仔细地说它，只是告诉大家有 Triple GAN 这个东西，有 BiGAN 就要有 Triple GAN。

Domain-adversarial training

在讲 Unsupervised Conditional Generation 的时候，我们用上了这个技术。这个技术在 ML 有讲过，所以这边就只是再复习一下。

- Training and testing data are in different domains



Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand,
Domain-Adversarial Training of Neural Networks, JMLR, 2016

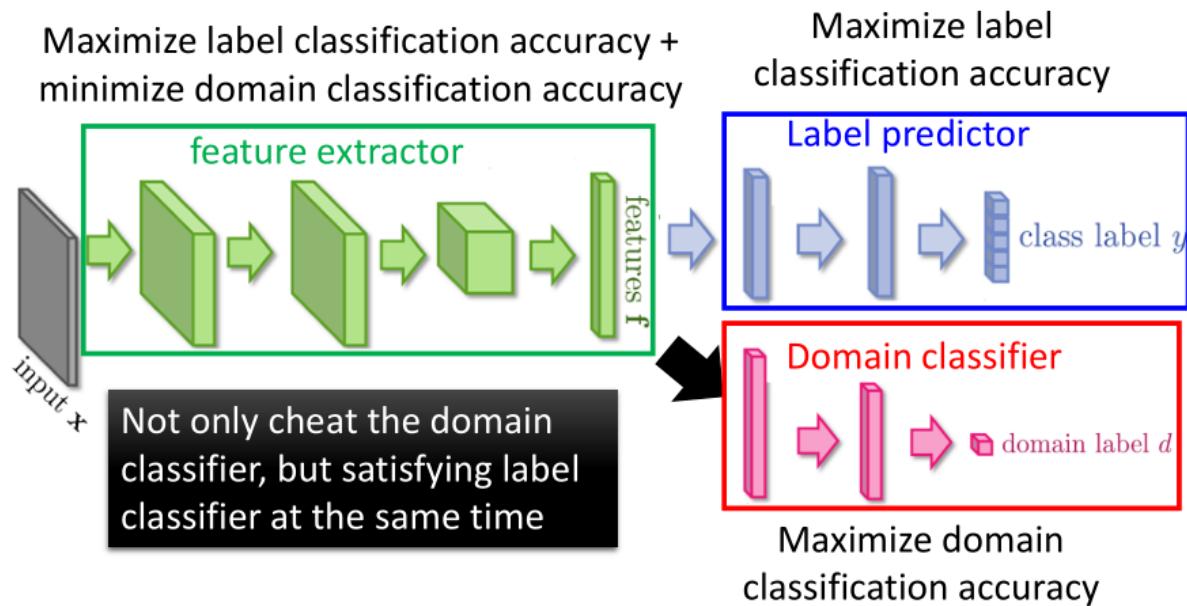
这个 Domain-adversarial training 就是要 learn 一个 generator，这个 generator 工作就是抽 feature。

假设要做影像的分类，这个 generator 工作就是吃一张图片 output 一个 feature。

在做 Machine Learning 的时候，很害怕遇到一个问题是，training data 跟 testing data 不 match，假设 training data 是黑白的 MNIST，testing data 是彩色的图片，是彩色的 MNIST，你可能会以为你在这个 training data 上 train 起来，apply 到这个 testing data 上，搞不好也 work。因为 machine 搞不好可以学到反正 digit 就是跟颜色无关，考虑形状就好了，所以他在黑白图片上 learn 的东西也可以 apply 到彩色图片。

但是事实上事与愿违，machine 就是很笨，实际上 train 下去，train 在黑白图片上，apply 彩色图片上，虽然你觉得 machine 只要学到把彩色图片自己在某个 layer 转成黑白的，应该就可以得到正确结果，但是实际上不是，它很笨，它就是会答错，怎么办？

我们希望有一个好的 generator，这个 generator 做的事情是 training set 跟 testing set 的 data 不 match 没有关系，透过 generator 帮你抽出 feature。在 training set 跟 testing set 虽然他们不 match，他们的 domain 不一样，但是透过 generator 抽出来的 feature，他们有同样的 distribution，他们是 match 的，这个就是 Domain-adversarial training。



This is a big network, but different parts have different goals.

怎么做呢，这个图在 Machine Learning 有看过了，就 learn 一个 generator，其实也就是 feature extractor，它吃一张 image 它会 output 一个 feature。有一个 Domain Classifier 其实就是 discriminator，这个 discriminator 是要判断现在这个 feature 来自于哪个 domain，假设有两个 domain，domain x 跟 domain y，你要 train 在 domain x 上面，apply 在 domain y 上面。然后这个时候 Domain Classifier 要做的事情是分辨这个 feature 来自于 domain x 还是 domain y，在这边同时你又要有另外一个 classifier，这个 classifier 工作是根据这个 feature 判断，假设现在是数字的分类，要根据这个 feature 判断它属于哪个 class，它属于哪个数字，这三个东西是一起 learn 的，但是实际上在真正 implement 的时候不一定一起 learn。

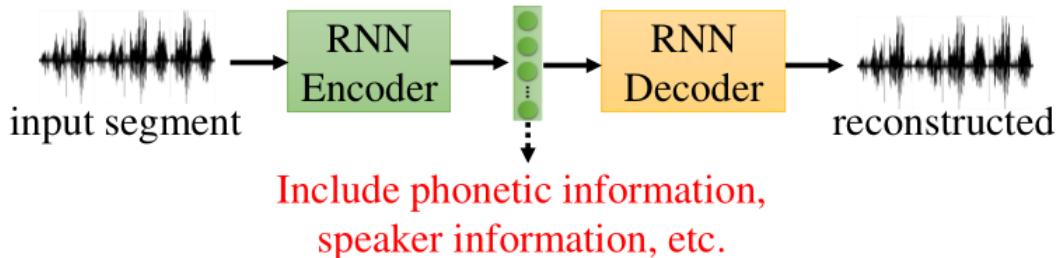
在原始 Domain-adversarial training 的 paper 里面，它就是一起 learn 的，这三个 network 就是一起 learn，只是这个 Domain Classifier 它的 gradient 在 back propagation 的时候在进入 Feature Extractor 之前，会乘一个负号，但是实际上真的在 implement 的时候你不一定要同时一起 train，你可以 iterative train，就像 GAN 一样。

在 GAN 里面也不是同时 train generator 跟 discriminator，你是 iterative 的去 train，有人可能会问能不能够同时 train generator 跟 discriminator，其实是可以的。如果你去看 f-GAN 那篇 paper 的话，它其实就 propose 一个方法，它的 generator 跟 discriminator 是 simultaneously train 的，就跟原始的 Domain-adversarial training 的方法是一样，有同学试过类似的做法，但发现同时 train 比较不稳，如果是 iterative train 其实比较稳，如果先 train Domain Classifier，再 train Feature Extractor，先 train

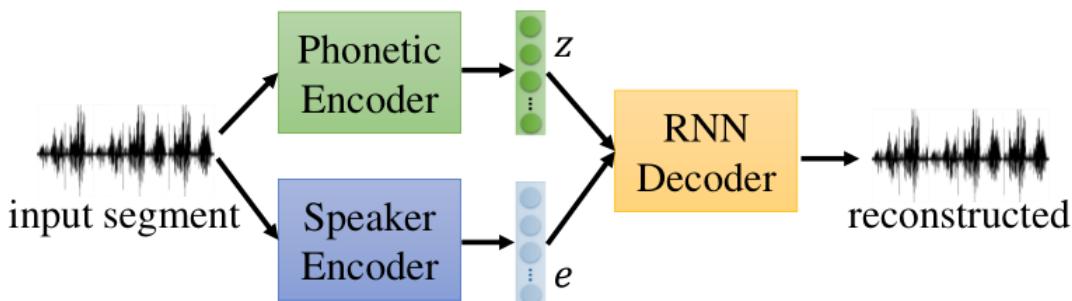
discriminator 再 train generator, iterative 的去 train, 它的结果会是比较稳的, 这个是 Domain-adversarial training。

Feature Disentangle

Original Seq2seq Auto-encoder



Feature Disentangle



用类似这样的技术可以做一件事情，这件事情叫做 Feature Disentangle，Feature Disentangle 是什么意思，用语音来做一下举例，在别的 domain 上比如说 image processing，或者是 video processing，这样的技术也是用得非常多。

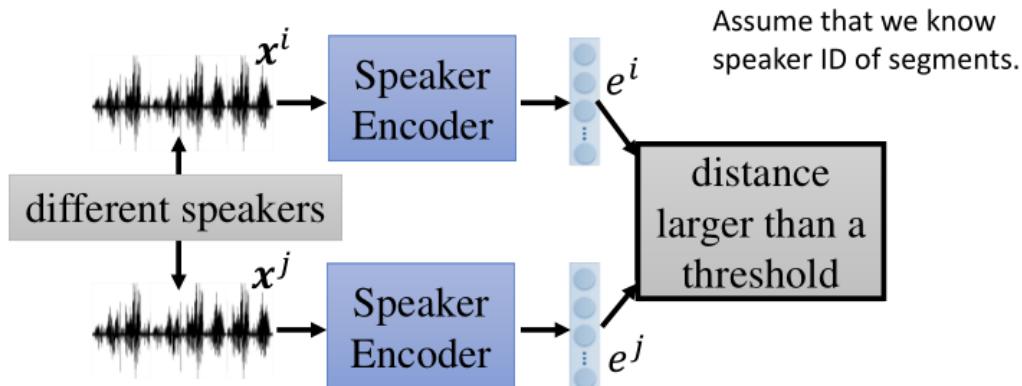
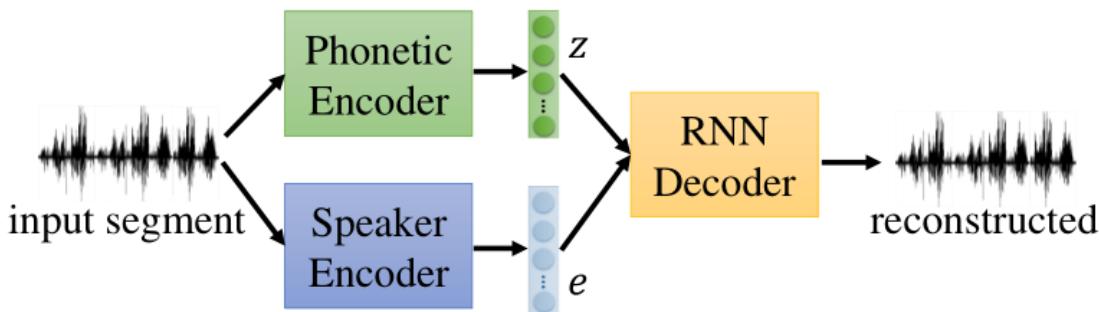
用语音来做例子，假设 learn 一个语音的 Autoencoder，learn 一个 sequence to sequence 的 Autoencoder，learn 一个 Autoencoder 它 input 是一段声音讯号，把这段声音讯号压成 code，再把这段 code 透过 decoder 解回原来的声音讯号，你希望 input 跟 output 越接近越好，就要 learn 这样一个 sequence to sequence Autoencoder，它中间你的 encoder 会抽出一个 latent representation，现在你的期待是 latent representation 可以代表发音的信息，但是你发现你实际 train 这样 sequence to sequence Autoencoder 的时候，你抽出来未必能让中间的 latent representation 代表发音的信息。

为什么？因为中间的 latent representation 它可能包含了很多各式各样不同的信息，因为 input 一段声音讯号，这段声音讯号里面不是只有发音的信息，它还有语者的信息，还有环境的信息，对 decoder 来说，这个 feature 里面一定必须要同时包含各种的信息，包含发音的信息，包含语者的信息，包含环境的信息，这个 decoder 根据所有的信息合起来，才可以还原出原来的声音。

我们希望做的事情是，知道在这个 vector 里面，到底哪些维度代表了发音的信息，那些维度代表了语者的信息或者是其他的信息，这边就需要用到一个叫做 Feature Disentangle 的技术，这种技术就有很多的用处。

因为你可以想象，假设你可以 learn 一个 encoder，它的 output 你知道那些维是跟发音有关的，那些维是跟语者有关的，你可以只把发音有关的部分，丢到语音识别系统里面去做语音识别，把有关语者的信息，丢到声纹比对的系统里面去，然后它就会知道现在是不是某个人说的。所以像这种 Feature Disentangle 技术有很多的应用。

Feature Disentangle



怎么做到 Feature Disentangle 这件事。

现在假设要 learn 两个 encoder，一个 encoder 它的 output 就是发音的信息，另外一个 encoder 它的 output 就是语者的信息，然后 decoder 吃发音的信息加语者的信息合起来，还原出原来的声音讯号。

接下来就可以把抽发音信息的 encoder 拔出来，把它的 output 去接语音识别系统，因为在做语音识别的时候，常会遇到的问题是两个不同的人说同一句话，它听起来不太一样，在声音讯号上不太一样，如果这个 encoder 可以把语者的 variation、语者所造成的差异 remove 掉。对语音识别系统来说辨识就会比较容易，对声纹比对也是一样，同一个人说不同的句子，他的声音讯号也是不一样，如果可以把这种发音的信息、content 的信息、跟文字有关的信息，把它滤掉，只抽出语者的特征的话，对后面声纹比对的系统也是非常有用。

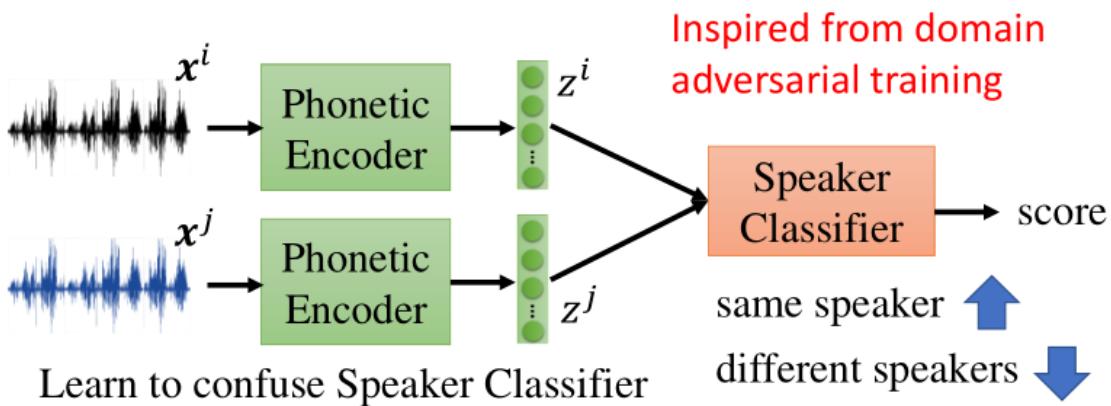
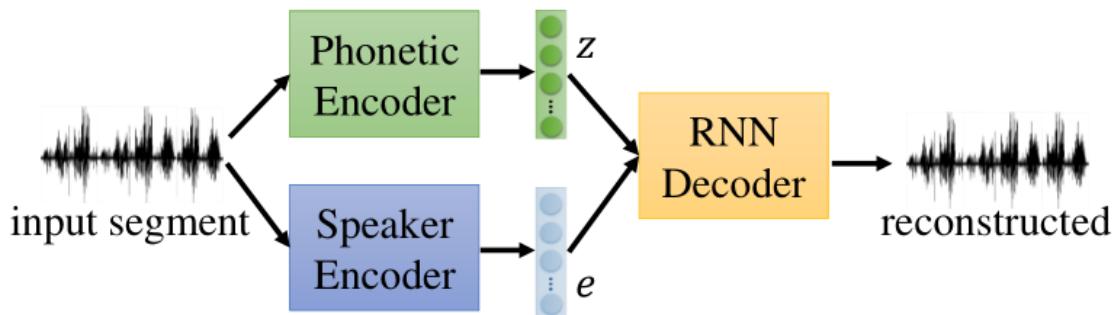
这件事怎么做，怎么让机器自动学到这个 encoder，如果这三个东西 joint learn，当然没有办法保证 Phonetic Encoder 的 output 一定要是发音的信息，Speaker Encoder 的 output 一定要是语者的信息。

于是就需要加一些额外的 constrain，比如说对语者的地方，你可能可以假设现在 input 一段声音讯号在训练的时候，我们知道那些声音讯号是同一个人说的。这个假设其实也还满容易达成的，因为可以假设同一句话就是同一个人说的，同一句话把它切成很多个小块，每一个小块就是同一个人说的。

所以对 Speaker Encoder 来说，给它同一个人说的声音讯号，虽然他们的声音讯号可能不太一样，但是 output 的 vector、output 的 embedding 要越接近越好。

同时假设 input 的两段声音讯号是不同人说的，那 output 的 embedding 就不可以太像，他们要有一些区别。

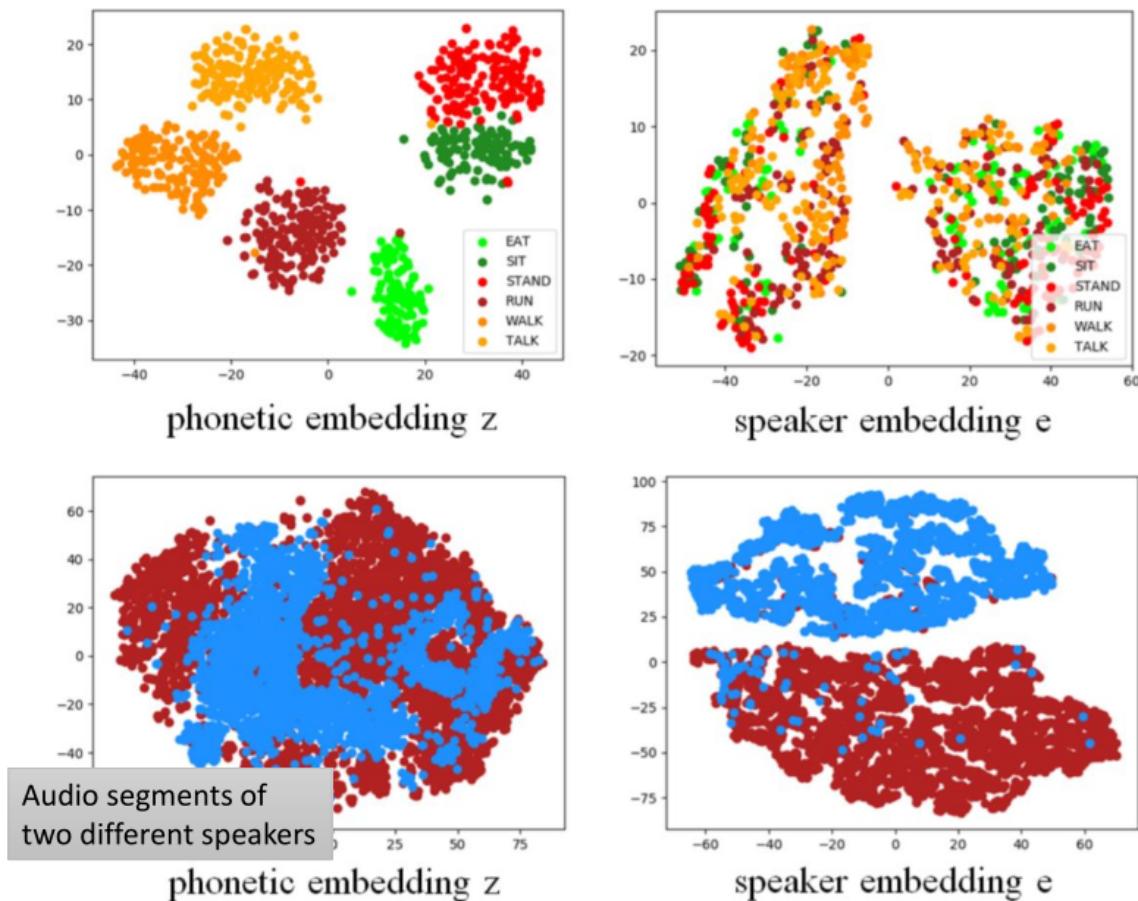
就算是这样做，你只能让 Speaker Encoder 的 output 考虑语者的信息，没有办法保证 Phonetic Encoder output 一定是发音的信息，因为也许语者的信息也会被藏在绿色的 vector 里，所以怎么办？



这边就可以用到 Domain Adversarial Training 的概念，再另外去 train 一个 Speaker Classifier。

Speaker Classifier 作用是给它两个 vector，它去判断这两个 vector 到底是同一个人说的还是不同的人说的，Phonetic Encoder 要做的事情就是去想办法骗过 Speaker Classifier，Speaker Classifier 要尽力去判断给他两个 vector 到底是同一个人说的还是不同人说的，Phonetic Encoder 要想尽办法去骗过 classifier，这个其实就是一个 GAN，后面就是 discriminator，前面就是 generator。

如果 Phonetic Encoder 可以骗过 Speaker Classifier，Speaker Classifier 完全无法从这些 vector 判断到底是不是同一个人说的，那就意味着 Phonetic Encoder 它可以滤掉所有跟语者有关的信息，只保留和语者无关的信息，这个就是 Feature Disentangle 的技术。



这边就是一些真正的实验结果，搜集很多有声书给机器去学，左边是 Phonetic Encoder 的 output，右边是 Speaker Encoder 的 output。

上面两个图每一个点就代表一段声音讯号，这边不同颜色的点代表声音讯号背后对应的词汇是不一样的，但他们都是不同的人讲的。如果看 Phonetic Embedding 的 output 就会发现，同样的词汇它是被聚集在一起的。虽然他们是不同人讲的，但是 Phonetic Encoder 知道它会把语者的信息滤掉，知道不同人讲的声音讯号不太一样，但是这些都是同一个词汇。

Speaker Encoder output 很明显就分成两群，不同的词汇发音虽然不太一样，但是因为现在 Speaker Encoder 已经把发音的信息都滤掉只保留语者的信息，就会发现不同的词汇都是混在一起的。

下面是两个不同颜色的点代表两个不同的 speaker，两个不同的语者他们所发出来的声音讯号。

如果看 Phonetic Embedding，看发音上面的信息，两个不同的人他们很有可能会说差不多的内容，所以这两个 embedding 重迭在一起。

如果看 Speaker Encoding 就会发现这两个人的声音，是很明显的分成两群的，这个就是 Feature Disentangle。

这边是举语音做例子，但是它也可以用在影像等等其他 application 上

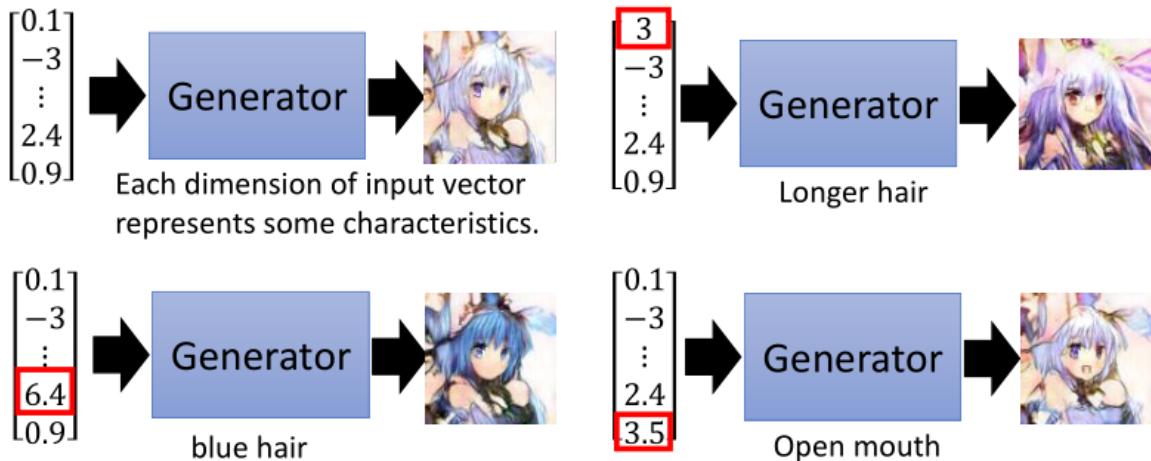
Intelligent Photo Editing by GAN

NVIDIA 自动修图是怎么做的？

我们知道假设 train 一个人脸的 generator，会 input 一个 random vector 然后就 output 一个人脸。

Modifying Input Code

在一开始讲 GAN 的时候跟大家说过 input vector 的每一个 dimension 其实可能对应了某一种类的特征，只是问题是当我们并不知道每一个 dimension 对应的特征是什么，现在要讲的是怎么去反推出现在 input 的这个 vector 每一个 dimension 它对应的特征是什么。



- The input code determines the generator output.
- Understand the meaning of each dimension to control the output.

现在的问题是这个样子，你其实可以收集到大量的 image，你可以收集到这些 image 的 label，label 说这张 image 里面的人，是金头发的、是男的，是年轻的等等，你可以得到 image，你也可以得到 image 的特征，你也可以得到 image 的 label，但现在的问题是会搞不清楚这张 image 它到底应该是由什么 vector 所生成的。

Connecting Code and Attribute

假设你可以知道生成这张 image 的 vector 长什么样子，你就可以知道 vector 跟 label 之间的关系。

因为你有 image 跟它特征的 label，假设可以知道某一张 image 可以用什么样的 random vector 丢到 generator 就可以产生这张 image，你就可以把这个 vector 跟 label 的特征 link 起来。

现在的问题就是给你一张 image，你其实并不知道什么样的 random vector 可以产生这张 image。

所以这边要做的第一件事情是，假设已经 train 好一个 generator，这个 generator 给一个 vector z 它可以产生一个 image x 。

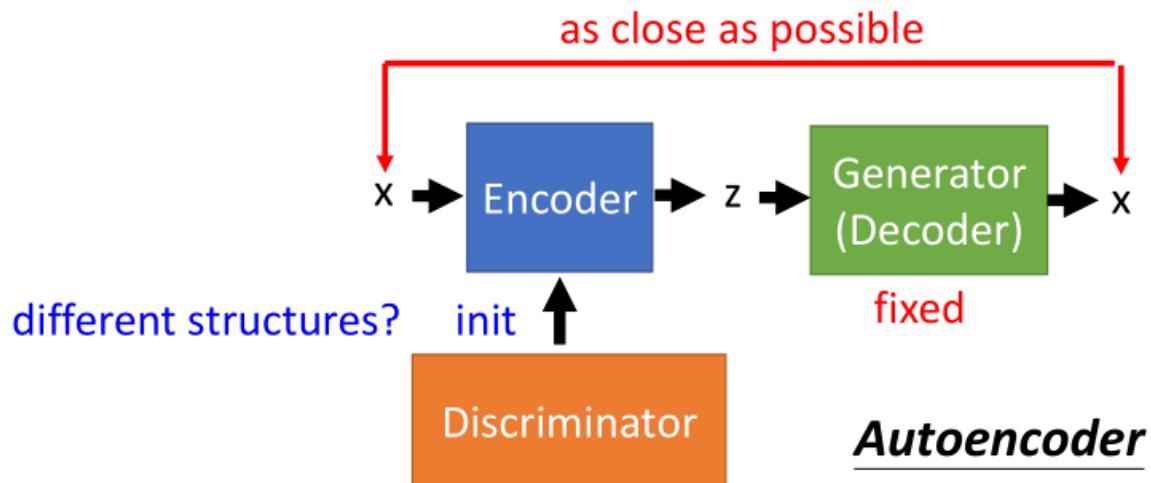
这边要做的事情是去做一个逆向的工程，去反推说如果给你一张现成的 image，什么样的 z 可以生成这张现成的 image。怎么做呢？

GAN + Autoencoder

这边的做法是再 learn 另外一个 encoder，再 learn 一个 encoder，这个 encoder 跟这个 generator 合起来就是一个 Autoencoder。在 train 这个 Autoencoder 的时候 input 一张 image x ，它把这个 x 压成一个 vector z ，希望把 z 丢到 generator 以后它 output 的是原来那张 image。

在 train 的过程中，generator 的参数是固定不动的，generator 是事先已经训练好的就放在那边，我们要做一个逆向工程猜出，假设 generator 产生某一张 image x 的时候，应该 input 什么样的 z ，要作一个反向的工程。这个怎么做？

- We have a generator (input z, output x)
- However, given x, how can we find z?
 - Learn an encoder (input x, output z)



就是 learn 一个 encoder，然后在 train 的时候给 encoder 一张 image，它把这个 image 变成一个 code z ，再把 z 丢到 generator 里面让它产生一张 image x ，希望 input 跟 output 的 image 越接近越好。

在 train 的时候要记得这个 generator 是不动的，因为我们是要对 generator 做逆向的工程，我们是要反推它用什么样的 z 可以产生什么样的 x ，所以这个 generator 是不动的，我们只 train encoder 就是了。

在实作上，这个 encoder 因为它跟 discriminator 很像，所以可以拿 discriminator 的参数来初始化 encoder 的参数，这是一个实验的细节。

接下来假设做了刚才那件事以后就得到一个 encoder，encoder 做的事情就是给一张 image 它会告诉你这个 image 可以用什么样的 vector 来生成。

Attribute Representation

现在你就把 database 里面的 image 都倒出来，然后反推出他们的 vector，就是这个 vector 可以生成这张图。



$$z_{long} = \frac{1}{N_1} \sum_{x \in long} En(x) - \frac{1}{N_2} \sum_{x' \notin long} En(x')$$

Short
Hair

$$x \rightarrow En(x) + z_{long} = z' \rightarrow Gen(z')$$

Long
Hair

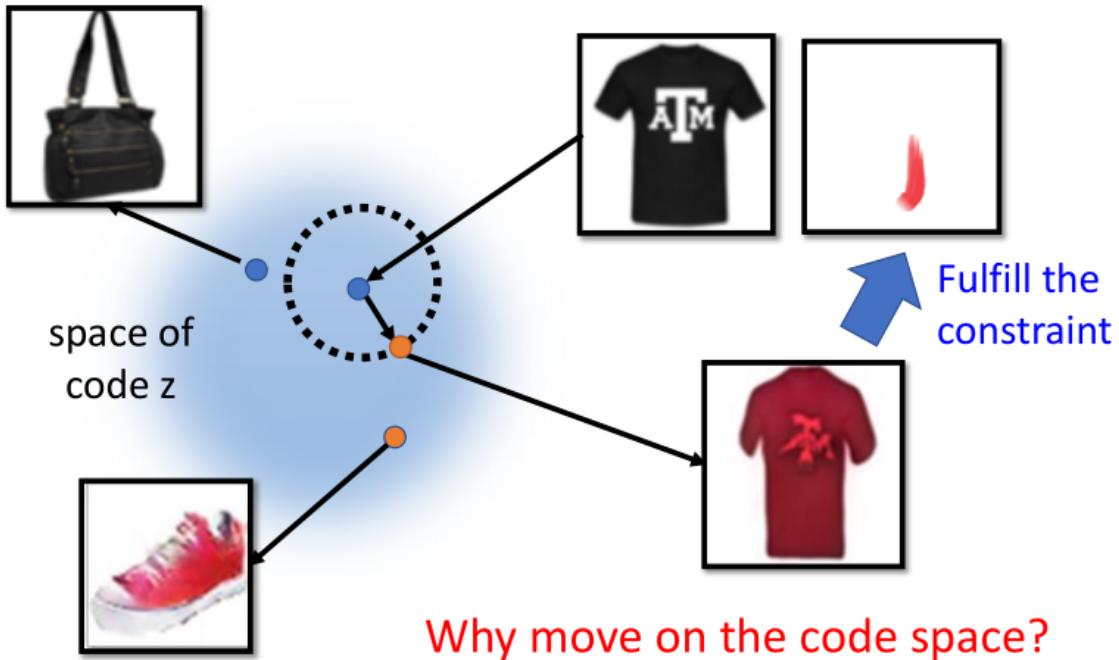
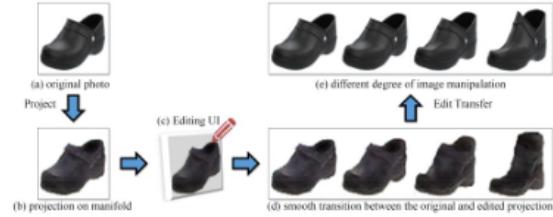
然后我们知道这些 image 他们的特征，这些是短发的人脸，这些是长发的人脸，把短发的人脸它的 code 推出来，再平均就得到一个短发的人脸的代表，把这个长发的人脸的 code 都平均就得到长发人脸的代表。再把他们相减就知道在这个 code space 上面做什么样的变化，就可以把短发的脸变成长发的脸，你把短发的脸加上这个向量 z_{long} 它就会变成长发。

z_{long} 是怎么来的？你就把长发的 image x 它的 code 都找出来，把 x 丢到 encoder 里面，把它 code 都找出来，然后变平均得到这个 z ，把这个不是长发的，短发的 code 都找出来平均，得到这个点，这两个点相减，就得到 z_{long} 这个向量。

接下来在生成 image 的时候，给你一张短发，你怎么把它变长发呢？，给你一张短发 image x ，你把 x 这张 image 丢到 encoder 里面得到它的 code，再加上 z_{long} 得到新的 vector z' ，再把 z' 丢到 generator 里面就可以产生一张长发的图。

Another Idea

Basic Idea



有另外一个版本的智能的 Photoshop。

这个做法是这样，首先 train 一个 GAN，train 一个 generator，这个 generator train 好以后，这个 generator 你从它的 latent space 随便 sample 一个点，假设 train 的时候是用商品的图来 train，那你在 latent space 上面、在 z 的 space 上面，随便 sample 一个 vector，丢到 generator 里面它就 output 一个商品。你拿不同位子做 sample 会 output 出不同商品，那接下来刚才看到智能的 Photoshop，给一张图片，然后在这个图片上面稍微做一点修改，结果就会产生一个新的商品，这件事情是怎么做的？

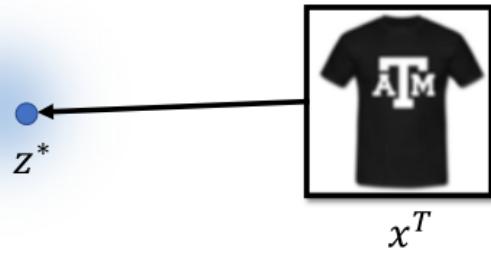
这个做法大概是这个样子，先把这张图片反推出它在 code space 上面的哪一个位子，然后接下来在 code space 上面做一下小小的移动，希望产生一张新的图片，这张新的图片一方面跟原来的图片够像，一方面它跟原来的图片够像，新的图片跟原来的图片够像，但同时又要满足使用者给的指示，比如使用者说这个地方是红色的，所以产生出来的图片在这个地方是红色的，但它仍然是一件 T-shirt。

假设 GAN train 的够好的话，只要在 code space 上做 sample，你在这 code space 上做一些移动，你的 output 仍然会是一个商品，只是有不同的特征。

所以你已经推出这张 image 对应的 code 就在这个地方，你把它小小的移动一下，就可以产生一张新的图，然后这张新的图要符合使用者给你的 constrain，接下来实际上怎么做的呢？

实际上会遇到的第一个问题就是要给一张 image，你要反推它原来的 code 长什么样子，怎么做到这件事？

Back to z



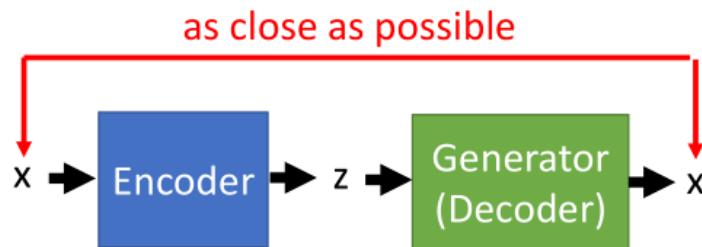
• Method 1

$$z^* = \arg \min_z L(G(z), x^T)$$
 Gradient Descent Difference between $G(z)$ and x^T

- Pixel-wise
- By another network

• Method 2

• Method 3



Using the results from method 2 as the initialization of method 1

有很多不同的做法，举例来说一个可行的做法是你把它当作是一个 optimization 的 problem 来解。你就在这个 code space 上面想要找到一个 vector z^* , z 可以产生所有的 image X^T , 所以要解的是这样一个 optimization problem, 要找一个 z^* 。把这个 z^* 丢到 generator 以后产生一张 image。

这个 $G(z)$ 代表一张产生出来的 image, 产生出来的 image 要跟原来的图片 X^T 越接近越好。 L 是一个 Loss Function, 它代表的是要衡量这个 $G(z)$ 这张图片跟 X^T 之间的差距。至于怎么衡量他们之间的差距有很多不同的方法, 比如说你用 Pixel-wise 的方法, 直接衡量 $G(z)$ 这张图片跟 X^T 的 L1 或 L2 的 loss, 也有人会说它是用 Perception Loss, 所谓 Perception Loss 是拿一个 pretrain 好的 classifier 出来, 这个 pretrain 好的 classifier 就吃这张图片得到一个 embedding, 再吃 X^T 得到一个 embedding, 希望 $G(z)$ 根据 pretrain 的 classifier (比如 VGG) 得到 embedding, 跟 X^T 得到的 embedding, 越接近越好。找一个 z^* , 这个 z^* 丢到 generator 以后, 它跟你目标的图片 X^T 越接近越好, 就得到了 z^* , 这是一个方法, 解这个问题可以用 Gradient Descent 来解。

第二个方法就是我们刚才在讲前一个 demo 的时候用的方法。

learn 一个 encoder, 这个 encoder 要把一张 image 变成一个 code z , 这个 z 丢到 generator 要产生回原来的 image, 这是个 Autoencoder, 你希望 input 跟 output 越接近越好。

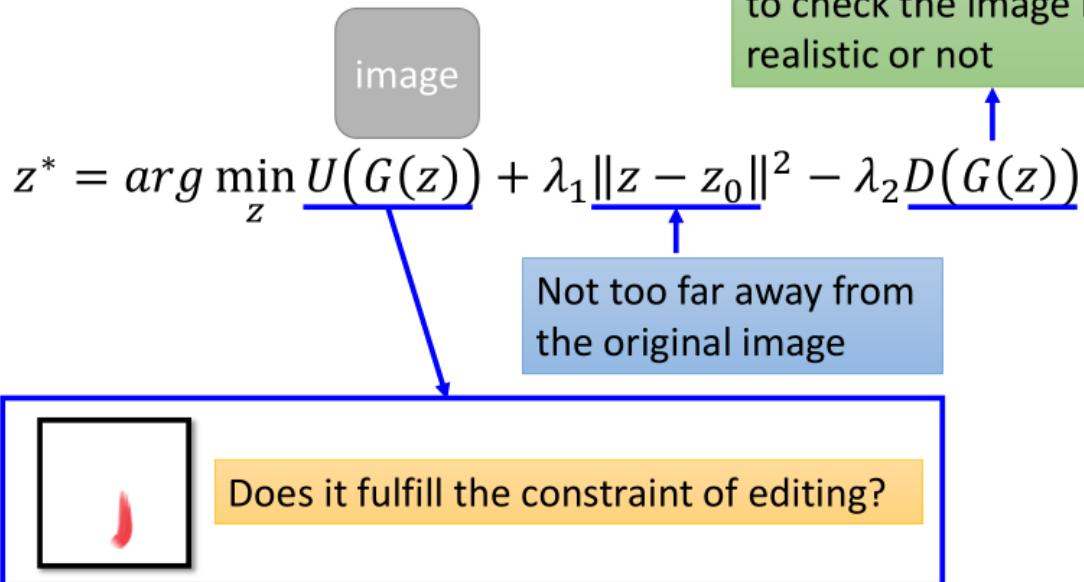
还有一个方法, 就是把第一个方法跟第二个方法做结合, 怎么做结合?

因为第一个方法要用 Gradient Descent, Gradient Descent 可能会遇到一个问题就是 Local Minimum 的问题, 所以在不同的地方做 initialization, 给 z 不同的 initialization 找出来得结果是不一样的, 你先用方法 2 得到一个 z , 用方法 2 得到的 z 当作方法 1 的 initialization, 再去 fine tune 你的结果, 可能得到的结果会是最好的。

Editing Photos



- z_0 is the code of the input image



总之有不同方法可以从 x 反推 z , 你可以从 x 反推 z 以后, 接下来要解另外一个 optimization problem, 这个 optimization problem 是要找一个 z , 这个 z 可以做到什么事情?

这个 z 一方面, 你把 z 丢到 generator 产生一张 image 以后, 这个 image 要符合人给的 constrain, 举例来说是这个地方要是红色的等等。

U 代表有没有符合 constrain, 那至于什么样叫做符合 constrain 这个就要自己去定义, 写智能 Photoshop 的 developer 要自己去定义。

你用 $G(z)$ 产生一张 image, 接下来用 U 这个 function 去算这张 image 有没有符合人定的 constrain。这是第一个要 minimize 的东西。

第二个要 minimize 的东西是你希望新找出来的 z , 跟原来的 z , 假设原来是一只鞋子, 原来这只鞋子, 你反推出它的 z 就是 z_0 , 你希望做一下修改以后, 新的 z 跟原来的 z_0 越接近越好。因为你不希望本来是一张鞋子, 然后你画一笔希望变红色的鞋子, 但它变成一件衣服, 不希望这个样子, 你希望它仍然是只鞋子, 所以希望新的 vector z 跟旧的 z_0 他们越接近越好。

最后还可以多加一个 constrain, 这个 constrain 是来自于 discriminator, discriminator 会看你把 z 丢到 generator 里面再产生的 image 丢到 D 里面, 把 generator output 再丢到 discriminator 里面, discriminator 去 check 这个结果是好还是不好。

你要找一个 z 同时满足这三个条件, 你要找一个 z 它产生的 image 符合使用者给的 constrain, 你要找一个 z 它跟原来的 z 不要差太多, 因为你希望 generate 出来的东西跟原来的东西仍然是同类型的, 希望找一个 z 它可以骗过 discriminator, discriminator 觉得你产生的结果是好的, 就解这样一个 optimization problem, 可以用 Gradient Descent 来解, 就找到一个 z^* , 这个就可以做到刚才讲的智能的 Photoshop, 就是这个做出来的。

Image super resolution

GAN 在影像上还有很多其他的应用，比如说它可以做 Super-resolution。

你完全可以想象怎么做 Super-resolution，它就是一个 Conditional GAN 的 problem，input 模糊的图 output 就是清晰的图。

input 是模糊的图，output 是清晰的图就结束了，要 train 的时候要搜集很多模糊的图跟清晰的图的 pair，要搜集这种 pair 很简单，就把清晰的图故意弄模糊就行了，实作就是这么做，清晰的图弄模糊比较容易，模糊弄清晰比较难。

- Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi, “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network”, CVPR, 2016



Figure 2: From left to right: bicubic interpolation, deep residual network optimized for MSE, deep residual generative adversarial network optimized for a loss more sensitive to human perception, original HR image. Corresponding PSNR and SSIM are shown in brackets. [4× upscaling]

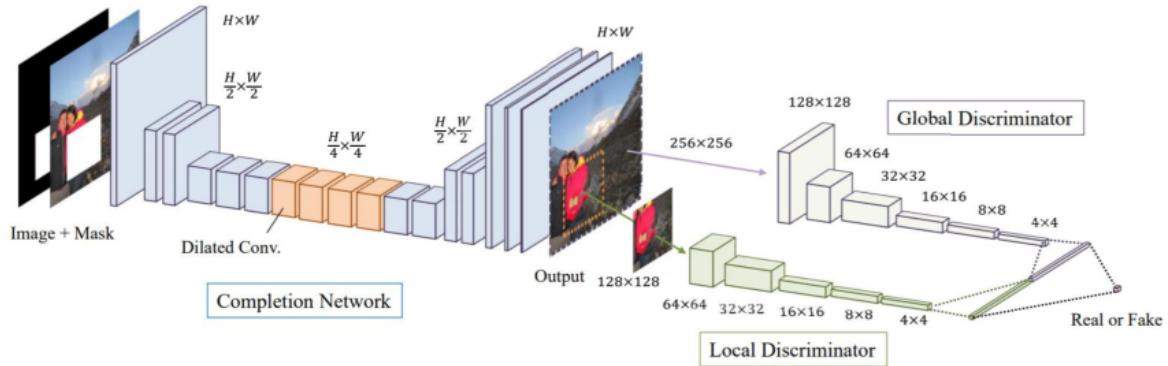
这个是文献上的结果，这个是还满知名的图，如果你有看过 GAN 的介绍，通常都会引用这组图，最左边这个是传统的、不是用 network 的方法得到的结果，产生出来的图是比较模糊的。第二个是用 network 的方法产生出来的图。最右边是原图。第三个是用 GAN 产生的出来的图，你会发现如果用 network 虽然比较清楚，但是在一些细节的地方，比如说衣领的地方，这个头饰的地方还是有些模糊的。但是如果看这个 GAN 的结果的话，在衣领和头饰的地方，花纹都是满清楚的。

有趣的地方是衣领的花纹虽然清楚，但衣领的花纹跟原图的花纹其实不一样，头饰的花纹跟原图的花纹是不一样，机器自己创造出清晰的花纹，反正能骗过 discriminator 就好，未必要跟原来的花纹是一样的，这是 image 的 Super-resolution。

Image Completion

Image Completion

<http://hi.cs.waseda.ac.jp/~iizuka/projects/completion/en/>



现在还会做的一个事情是 Image Completion, Image Completion 就是给一张图片, 然后它某个地方挖空, 机器自己把挖空的地方补上去, 这个怎么做?

这个就是 Conditional GAN, 就是给机器一张有挖空的图, 它 output 一张填进去的图就结束了, 怎么产生这样的 training data? 它非常容易产生, 就找一堆图片, 中间故意挖空就得到这种 training data pair, 然后就结束了。

Improving Sequence Generation by GAN

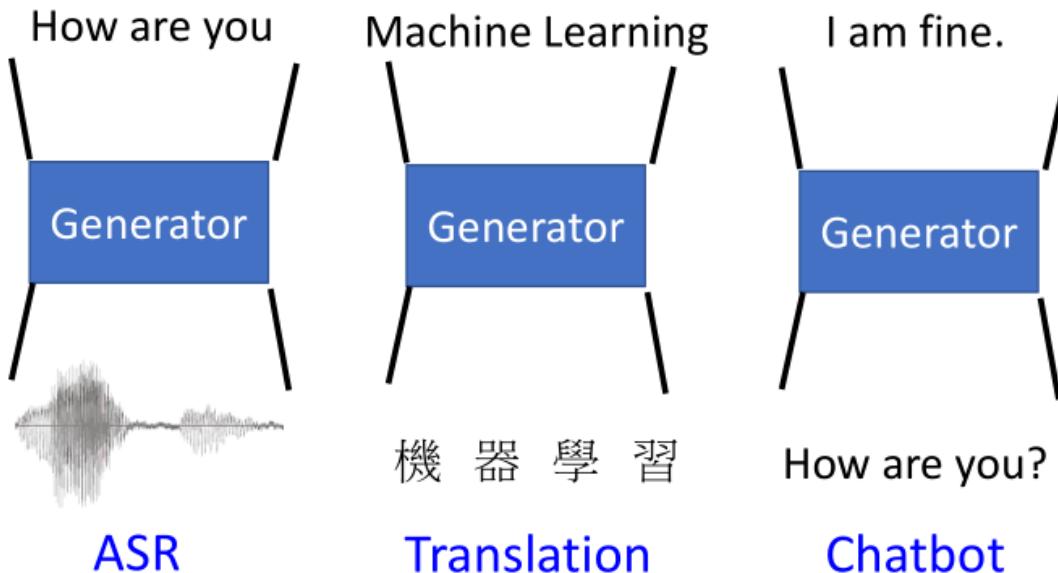
我们今天要讲的是用 GAN, 来 improve sequence generation。

那 sequence generation 的 task 它有非常多的应用, 我们先讲怎么用 GAN 来 improve conditional sequence generation。

接下来我们会讲, 我们还可以做到 Unsupervised conditional sequence generation。

Conditional Sequence Generation

只要是要产生一个 sequence 的 task, 都是 conditional sequence generation。



The generator is a typical seq2seq model.

With GAN, you can train seq2seq model in another way.

举例来说语音识别可以看作是一个 conditional sequence generation 的 task。你需要的是一个 generator, input 是声音讯号, output 就是语音识别的结果就是这一段声音讯号所对应到的文字。或者假设你要做翻译, 你要做 translation 的话, 你的 input 是中文, output 就是翻译过的结果, 是一串 word sequence。或者是说 chatbot 也是一个 conditional sequence generation 的 task, 它的 input 是一个句子, output 是另外一个 sequence。

那我们之前有讲过, 其实这些 task, 语音识别, 翻译或 chatbot, 都是用 sequence to sequence 的 model 来解它的, 所以实际上这边这个图上所画的 generator, 它们都是 sequence to sequence 的 model。

只是今天要讲的是用一个不一样的方法, 用 GAN 的技术来 train 一个 seq2seq model。

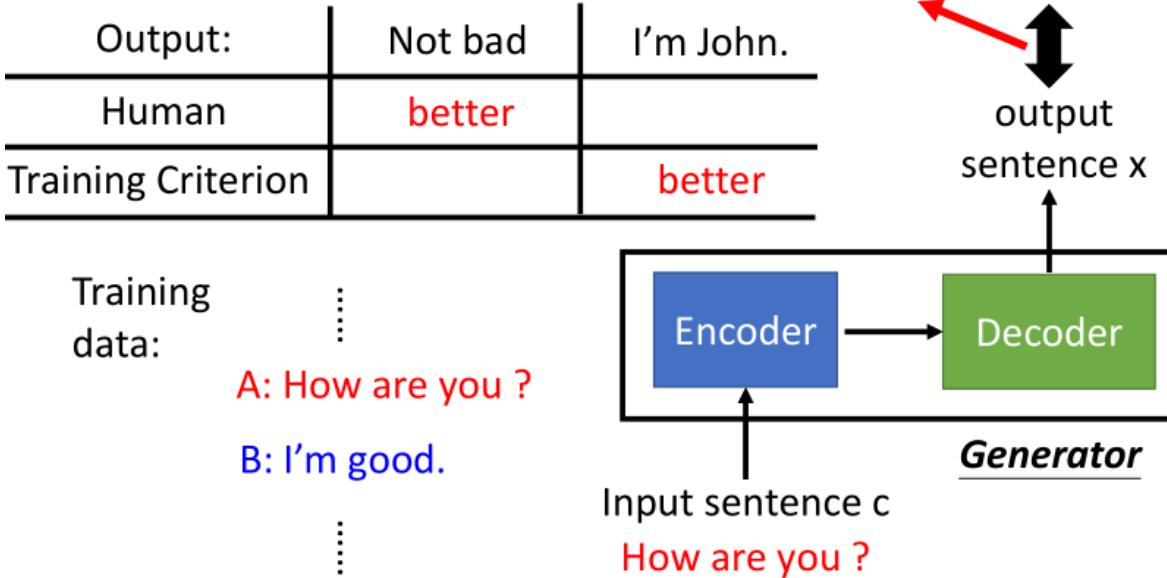
那为什么我们会要用到 GAN 的技术或其他的技术来 train seq2seq model 呢。我们先来看看我们 train seq2seq model 的方法有什么不足的地方, 假设我们就 train 了一个 chatbot, 一个 chatbot 它是一个 seq2seq model, 它里面有一个 encoder, 有一个 decoder, 这个 seq2seq model 就是我们的 generator, 那这个 encoder 会吃一个 input 的句子, 这边用 c 来表示, 那它会 output 另外一个句子 x , encoder 吃一个句子, 之于 decoder 会 output 一个句子 x 。

那我们知道说要 train 这样子的 chatbot, 你需要收集一些 training data。所谓的 training data 就是人的对话, 所以你今天告诉 chatbot 说, 在这个 training data 里面, A 说 How are you 的时候, B 的响应是 I'm good, 所以 chatbot 必须学到, 当 input 的句子是 How are you 的时候, 它 output 这个 I'm good 的 likelihood 应该越大越好。

意思就是说, 今天假设正确答案是 I'm good, 那你在用 decoder 产生句子的时候, 第一个 time step 产生 I'm 的机率要越大越好, 那在第二个 time step 产生 good 的机率要越大越好。

那这么做显然有一个非常大的问题, 就是我们看两个可能的 output, 假设今天有一个 chatbot, 它 input How are you 的时候, 它 output 是 Not bad, 有另外一个 chatbot, 它 input How are you 的时候, 它 output 是 I'm John。

- Chat-bot as example



如果从人的观点来看，Not bad 是一个比较合理的 answer，I'm John 是一个比较奇怪的 answer。但是如果从我们 training 的 criteria 来看，从我们在 train 这个 chatbot 的时候，希望 chatbot 要 maximize 的 object 希望 chatbot 学到的结果来看，事实上 I'm John 是一个比较好的结果，为什么呢？因为 I'm John 至少第一个 word 的还是对的，那如果是 Not bad，你两个 word 都是错的，所以从这个 training 的 criteria 来看是这样子的，假设你 train 的时候是 maximum likelihood。

其实 maximum likelihood 就是 minimize 每一个 time step 的 cross entropy，这两个其实是 equivalent 的东西，maximum likelihood 就是 minimize cross entropy。这是一个真正的例子，某人去面试某一个大家都知道的，全球性的科技公司，被问了这个问题，人家问他说，train 这个 classifier 的时候，有时候我们会说我们是 maximum likelihood，有时候我们会说我们是在 minimize cross entropy，这两者有什么不同呢？如果你答这两个东西有点像，但他们中间有微妙的不同，你就错了。这个时候你就是要说，他们两个就是一模一样的东西，maximum likelihood 跟 minimize cross entropy，是一模一样的东西。

Improving Supervised Seq-to-seq Model

RL (human feedback)

我们先讲一下怎么去 improve 这个 seq2seq 的 model。

我们会先讲，怎么用 reinforcement learning 来 improve conditional generation。然后接下来我们才会讲说，怎么用 GAN 来 improve conditional generation。

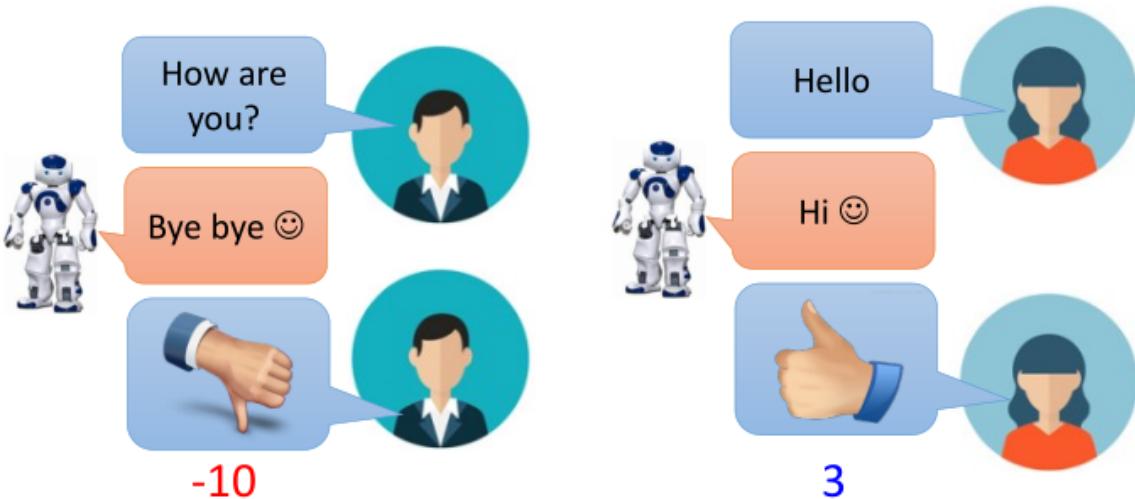
之所以要讲 RL，是因为等一下你会发现，用 GAN 来 improve conditional generation 这件事情，其实跟 RL 是非常像的。你甚至可以说使用 RL，来 improve seq2seq 的 chatbot，可以看作是 GAN 的一个 special case。

假设我们今天要 train 一个 seq to seq 的 model，你不想要用 train maximum likelihood 的方法，来 train seq to seq model，因为我们刚才讲用 maximum likelihood 的方法有很明显的问题。

我们都用 chatbot 来做例子，其实我们讨论的技术，不是只限于 chatbot 而已，任何 seq to seq model，都可以用到等一下讨论的技术。不过我们等一下举例的时候，我们都假设我们是要做 chatbot 就是了。

那今天假设你要 train 一个 chatbot，你不要 maximum likelihood 的方法，你想要 Reinforcement learning 的方法，那你会怎么做呢？

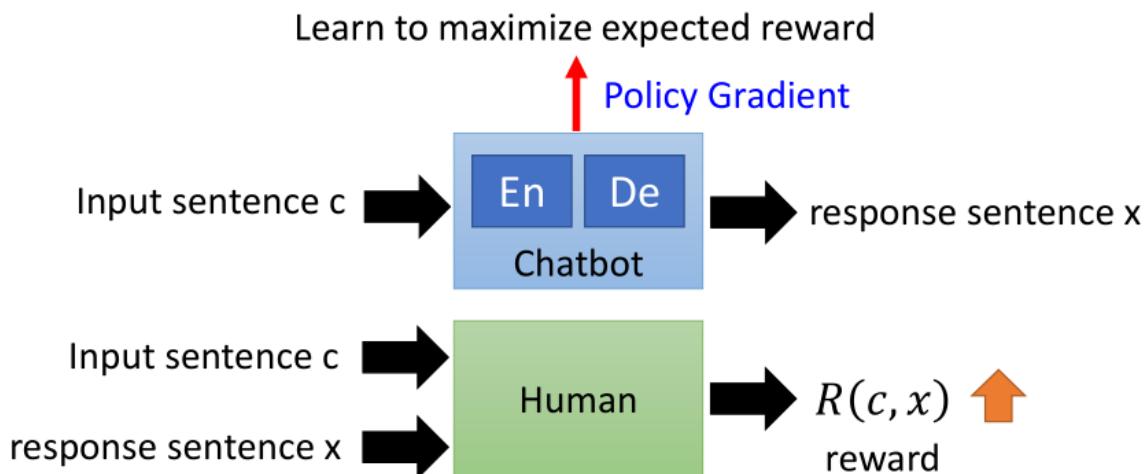
Machine obtains feedback from user



Chat-bot learns to maximize the expected reward

你的做法可能是这样，你就让这个 chatbot 去胡乱在线上跟人讲话，就有一个人说 How are you，chatbot 就回答 bye-bye，人就会给 chatbot 一个很糟的评价，chatbot 就知道说这样做是不好的；再下一次他跟人对话的时候，人说 Hello，chatbot 说 Hi，人就觉得说它的回答是对的，就给它一个 positive 的评价，chatbot 就知道说它做的事情是好的。那 chatbot 在跟人互动的过程中呢，他会得到 reward。把这个问题想的单纯一点，就是人说一个句子，然后 chatbot 就做一个响应，人就会给 chatbot 一个分数，chatbot 要做的事情，就是希望透过互动的过程，它去学习怎么 maximize 它可以得到的分数，

Maximizing Expected Reward



[Li, et al., EMNLP, 2016]

我们现在的问题是，有一个 chatbot，它 input 一个 sentence c ，要 output 一个 response x ，它就是一个 seq to seq model，接下来有一个人，人其实也可以看作是一个 function，人这个 function 做的事情就是，input 一个 sentence c ，还有 input 一个 response x ，然后给一个评价，给一个 reward，这个 reward 我们就写成 $R(c, x)$ ，但如果你熟悉 conditional generation 的话，你会发现这个图，跟用 GAN 做 conditional generation，其实是非常像的。唯一的不同是，如果用 GAN 做 conditional generation 的话，这个绿色的方块，它是一个 discriminator。切记 discriminator 它不要只吃 generator 的 output，它要同时吃 generator 的 input 跟 output，才能给与评价。今天人也是一样，人来取代那个

discriminator, 人就不用 train, 或者是说你可以说人已经 train 好了, 人有一个脑, 然后在数十年的成长历程中其实已经 train 好了, 所以你不用再 train。给一个 input sentence c , 给一个 response x , 然后你可以给一个评价。

我们接下来要做的事情, chatbot 要做的事情就是, 它调整这个 seq to seq model 里面内部的参数, 希望去 maximize 人会给它的评价, 这边写成 $R(c, x)$, 这件事情怎么做呢? 我们要用的技术就是 policy gradient。policy gradient 我们其实在 machine learning 的最后几堂课其实是有说过的。

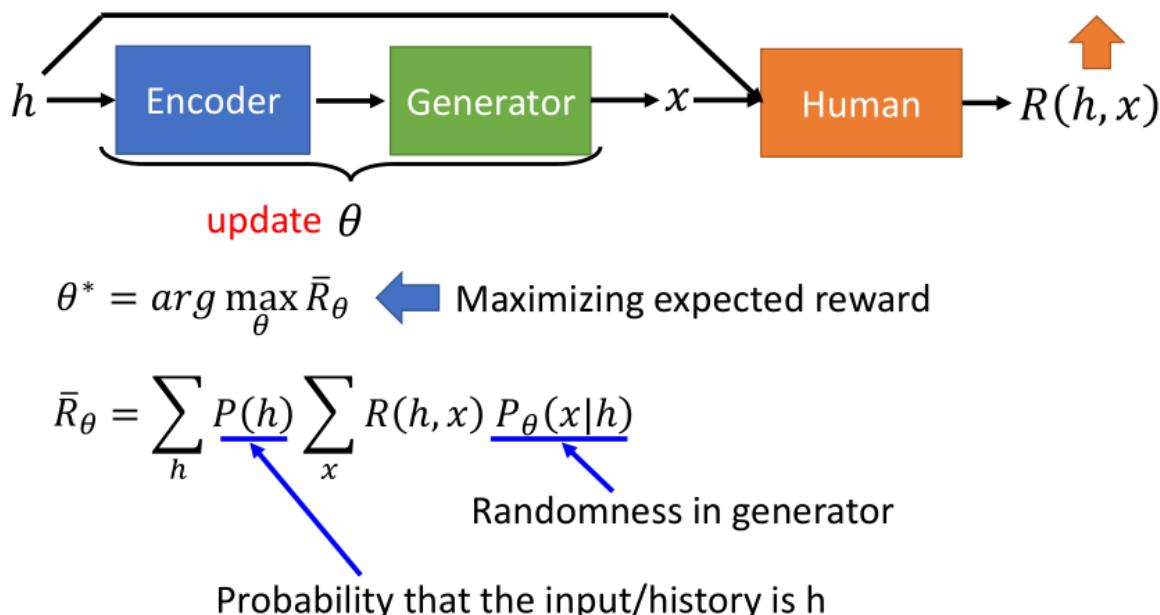
(文章中的 c , 与slides中的 h 是同一个东西)

我们这边以 chatbot 做例子, 来很快地复习一下, policy gradient 是怎么做的。

那我们有一个 seq to seq model, 它的 input 是 c output 是 x 。接下来我们有另外一个 function, 这个 function 是人, 人吃 c 跟 x , 然后 output 一个 R 。

那我们现在要做的事情是什么呢? 我们要去调 encoder 跟 generator 的参数, 这个 encoder 跟 generator 合起来是一个 seq to seq model, 他们合起来的参数, 我们叫做 θ 。我们希望调这个 θ 去 maximize human 这个 function 的 output。

那怎么做呢, 我们先来计算给定某一组参数 θ 的时候, 这个时候这个 chatbot, 会得到的期望的 reward 有多大。假设这个 θ 是固定的, 然后计算一下这个 seq to seq model, 它会得到的期望的 reward 是有多大。



怎么算呢? 首先我们先 summation over 所有可能的 input c , 然后乘上每一个 c 出现的机率, 因为 c 可能有各种不同的 output, 比如说人可能说 How are you, 人可能说 Good morning, 人可能说 Good evening, 你有各种各样的 input, 你有各种各样的 c , 但是每一个 input 出现的机率, 可能是不太一样的, 比如说 How are you 相较于其他的句子, 也许它出现的机率是特别大的, 因为人特别常对 chatbot 说这个句子。接下来, summation over 所有可能的回应 x 。当你有一个 c 的时候, 当你有一个 input c , 再加上假设这个 chatbot 的参数 θ 我们已经知道的时候, 接下来你就可以算出一个机率, 这个机率是在 given c 这组参数的情况下, chatbot 会回答某一个答复 x 的机率有多少。给一个 input c , 为什么 output 会是一个机率呢?

你想想看, 我们今天在 train seq to seq model 的时候, 每一个 time step 我们不是其实要做一个 sampling 嘛, 我们 train 一个 seq to seq model 的时候, 每一次给同样的 input, 它的 output, 不见得是一样的, 假设你在做 sampling 的时候, 我们的 decoder 的 output 是一个 distribution, 你要把这个 distribution 变成一个 token 的时候, 如果你是采取 sampling 的方式, 那你 chatbot 的每一次 output 都会是不一样的。所以今天给一个 c , 每一次 output 的 x , 其实是不一样的, 所以给一个 c , 我们其实得到的是一个 x 的机率。

假设你不是用 sampling 的方式，你是用 argmax 的方式呢？其实也可以，如果是用 argmax 的方式，给一个 c ，那你一定会得到一模一样的 x ，但我们可以这么说，那个 x 出现的机率就是 1，其他的 response 出现的机率都是 0，其他的 x 出现机率都是 0。

总之给你一个 c ，在参数 x, θ 知道的情况下，你可以把 chatbot 可能的 output 看作是一个 distribution，写成 $P_\theta(x | c)$ 。当给一个 c ，chatbot 产生一个 x 的时候，接下来人就会给一个 reward $R(c, x)$ 。

这一整项 summation over 所有的 c ，summation over 所有的 x ，这边乘上 c 的机率，这边乘上 x 出现的机率，再 weighted by 这个 reward，其实就是 reward 的期望值。

接下来我们要做的事情就是，我们要调这个 θ ，要调这个 chatbot 的参数 θ ，让 reward 的期望值，越大越好，那这件事情怎么做呢？

$$\theta^* = \arg \max_{\theta} \bar{R}_{\theta} \quad \leftarrow \text{Maximizing expected reward}$$

$$\begin{aligned} \bar{R}_{\theta} &= \sum_h P(h) \sum_x R(h, x) P_{\theta}(x|h) = E_{h \sim P(h)} \left[E_{x \sim P_{\theta}(x|h)} [R(h, x)] \right] \\ &= E_{h \sim P(h), x \sim P_{\theta}(x|h)} [R(h, x)] \approx \frac{1}{N} \sum_{i=1}^N R(h^i, x^i) \end{aligned}$$

Sample: $(h^1, x^1), (h^2, x^2), \dots, (h^N, x^N)$

**Where
is θ ?**

我们把这个 reward 的期望值稍微做一下整理，就是我们从 P of c 里面 sample 出一个 c 来，我们从这个机率里面 sample 出一个 x 来，然后取 $R(c, x)$ 的期望值。

然后接下来的问题就是，这个期望值要怎么算？

你要算这个期望值，theoretical 做法要 summation over 所有的 c ，summation over 所有的 x ，但是在实作上，你根本无法穷举所有 input，你根本无法穷举所有可能 output。

所以实作上就是做 sampling，假设这两个 distribution 我们知道。 $P(c)$ ，人会说什么句子，你就从你的 database 里面 sample 看看，从 database 的句子里面 sample，就知道人常输入什么句子，那 $P_{\theta}(x | c)$ ，你只要知道参数，他就是给定的。

所以我们根据这两个机率，去做一些 sample，我们去 sample 大 N 笔的 c 跟 x 的 pair，比如说上百笔的 c 跟 x 的 pair。

所以本来这边应该是要取一个期望值，但实际上我们并没有办法真的去取期望值，我们真正做法是，做一下 sample，sample 出大 N 笔 data，这大 N 笔 data，每一笔都去算它的 reward，把这大 N 笔 data 的 reward 全部平均起来，我们用 $\sum_{i=1}^N R(c^i, x^i)$ 来 approximate 期望值， $\frac{1}{N} \sum_{i=1}^N R(c^i, x^i)$ 就是期望的 reward 的 approximation。

那我们现在要对 θ ，我们要对 θ 做 optimization，我们要找一个 θ 让 \bar{R}_{θ} 这一项越大越好，那意味着说我们要拿 θ 去对 \bar{R}_{θ} 算它的 gradient。

但是问题是在 $\frac{1}{N} \sum_{i=1}^N R(c^i, x^i)$ 里面，我们说 \bar{R}_{θ} 就等于这项，这项里面没有 θ 啊，没有 θ 你根本没有办法对 θ 算 gradient，不知不觉间，它就不见了，它到哪里去了呢？

它被藏到 sampling 的这个 process 里面去了，当你改变 θ 的时候，你会改变 sample 到的东西，但在这式子里面， θ 就不见了，你根本就不知道要怎么对这个式子算 θ 的 gradient，所以怎么办呢？

Policy Gradient

实作上的方法是这个样子的，这一项如果把它 approximate 成 $\frac{1}{N} \sum_{i=1}^N R(c^i, x^i)$ 的话，就会没有办法算 gradient 了，所以怎么办？

先把对 \bar{R}_θ 算 gradient，再做 approximation，这一项算 gradient 是怎么样呢？只有 $P_\theta(x | c)$ 跟 θ 是有关的，所以你对 \bar{R}_θ 取 gradient 的时候，那你只需要把 gradient 放到 P_θ 的前面就好了。

接下来，唯一的 trick 是对这一个式子，分子和分母都同乘 $P_\theta(x | c)$ ，分子分母同乘一样的东西，当然对结果是没有任何影响的。

那我们知道右上角的式子，微分告诉我们反正就是这个样子。

所以今天这个式子，其实蓝框里面的这两项是一样的。

Policy Gradient

$$\frac{d \log(f(x))}{dx} = \frac{1}{f(x)} \frac{df(x)}{dx}$$

$$\begin{aligned}\bar{R}_\theta &= \sum_h P(h) \sum_x R(h, x) P_\theta(x|h) \approx \frac{1}{N} \sum_{i=1}^N R(h^i, x^i) \\ \nabla \bar{R}_\theta &= \sum_h P(h) \sum_x R(h, x) \nabla P_\theta(x|h) \approx \frac{1}{N} \sum_{i=1}^N R(h^i, x^i) \nabla \log P_\theta(x^i|h^i) \\ &= \sum_h P(h) \sum_x R(h, x) P_\theta(x|h) \boxed{\frac{\nabla P_\theta(x|h)}{P_\theta(x|h)}} \\ &= \sum_h P(h) \sum_x R(h, x) P_\theta(x|h) \boxed{\nabla \log P_\theta(x|h)} \\ &= E_{h \sim P(h), x \sim P_\theta(x|h)} [R(h, x) \nabla \log P_\theta(x|h)]\end{aligned}$$

Sampling



那接下来呢，变成期望值的形式。

所以这一项，当你要对 \bar{R}_θ 做 gradient 的时候，你要去 approximate $\nabla \bar{R}_\theta$ 的话，你是怎么算的呢？

这一项就是，把 summation 换做 sampling，你就 sample 大 N 项，每一项都去算 $R(c^i, x^i) \nabla \log P_\theta(x^i | c^i)$ ，把它们平均起来就是 expectation 的 approximation。

所以我们实际上是怎么做的呢？你 update 的方法是，原来你的参数叫做 θ^{old} ，然后你用 gradient ascent 去 update 它，加上某一个 gradient 的项，你得到新的 model θ^{new} ，gradient 这一项怎么算？gradient 这一项算法就是，去 sample N 个 pair 的 c^i 跟 x^i 出来，然后计算 $R(c^i, x^i) \nabla \log P_\theta(x^i | c^i)$ ，就结束了。

其实这一项它是非常的直觉的，怎么说它非常的直觉呢？这个 gradient 所代表的意思是说，假设今天 given c_i, x_i ，也就是说有人对 machine 说了 c_i 这个句子，machine 回答 x_i 这个句子，然后人给的 reward 是 positive 的，那我们就要增加 given c_i 的时候， x_i 出现的机率。反之如果 R of (c_i, x_i) 是 negative 的，当人对 chatbot 说 c_i ，chatbot 回答 x_i ，然后得到负面的评价的时候，这个时候我们就应该调整参数 θ ，让 given c_i ，回答 x_i 的这个机率呢，越小越好。

Gradient Ascent

$$\theta^{new} \leftarrow \theta^{old} + \eta \nabla \bar{R}_{\theta^{old}}$$

$$\nabla \bar{R}_{\theta} \approx \frac{1}{N} \sum_{i=1}^N R(h^i, x^i) \nabla \log P_{\theta}(x^i | h^i)$$

$R(h^i, x^i)$ is positive

→ After updating θ , $P_{\theta}(x^i | h^i)$ will increase

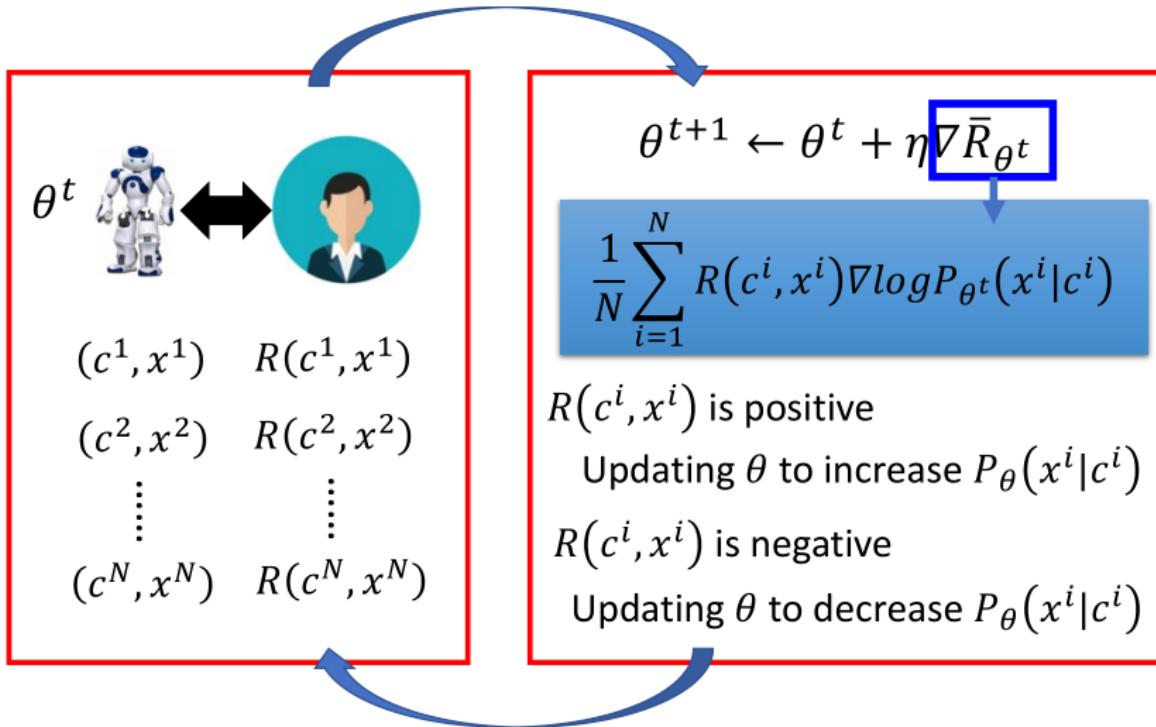
$R(h^i, x^i)$ is negative

→ After updating θ , $P_{\theta}(x^i | h^i)$ will decrease

所以实作上的时候，如果你要用 policy gradient 这个技术，来 implement 一个 chatbot，让它在 reinforcement learning 的情境中，可以去学习怎么和人对话的话，实际上你是怎么做的呢？

Implementation

实际上你的做法是这个样子，你有一个 chatbot 它的参数叫做 $\theta(t)$ ，然后你把你的 chatbot 拿去跟人对话，然后他们就讲了很多，这个是一个 sampling 的 process。



你先用 chatbot 跟人对话，做一个 sampling 的 process，在这个 sampling 的 process 里面，当人说 c_1 chatbot 回答 x_1 的时候，会得到 reward R of (c_1, x_1) ，当输入 c_2 回答 x_2 的时候，会得到 reward R of (c_2, x_2) ，那你会 sample 出 N 笔 data，每一笔 data 都会得到一个 reward， N 笔 data N 个 reward。

接下来你做的事情是这样，你有一个参数 $\theta(t)$ ，你要 update 这个参数，让它变成 $\theta(t+1)$ 。那怎么 update 呢？你要把它加上对这个 \bar{R}_{θ} 的 gradient，那这个 \bar{R}_{θ} 的 gradient 这一项到底怎么算呢？这一项式子就列在这边，那这个式子的直观解释我们刚才讲过说，如果 R of (c_i, x_i) 是正的，那就增加这一项的机率，如果 R of (c_i, x_i) 是负的，就减少这一项的机率。

但是你要注意，每次你 update 完参数以后，你要从头回去，再去 sample data，因为这个 \bar{R}_θ 它是在 given 参数是 θ 的情况下，所算出来的结果，一但 update 你的参数，从 $\theta(t)$ 变成 $\theta(t+1)$ ，gradient 这一项就不对了，你本来参数 $\theta(t)$ ，一但你 update 变成 $\theta(t+1)$ 以后，你就要回过头去再重新收集参数。

所以这跟一般的 gradient decent 非常不同，因为一般 gradient decent，你就算 gradient，然后就可以 update 参数，然后就可以马上再算下一次 gradient，再 update 参数。

但是如果你 apply reinforcement learning 的时候，你的做法是，每次你 update 完参数以后，你就要去跟使用者再互动，然后才能再次 update 参数，所以每次 update 参数的时间呢，需要的 effort 是非常大的。每 update 一次参数，你就要跟使用者互动 N 次，才能 update 下一次参数，所以在 policy gradient 里面，update 参数这件事情，是非常宝贵的，就这一步是非常宝贵的，绝对不能够走错这样子，你一走错，你就要要你要重新再去跟人互动，才能够走回来，那你也有可能甚至就走不回来。所以之后会讲到一些新的技术，来让这一步做得更好，不过这是我们之后才要再讲的东西。

Comparison

	Maximum Likelihood	Reinforcement Learning
Objective Function	$\frac{1}{N} \sum_{i=1}^N \log P_\theta(\hat{x}^i c^i)$	$\frac{1}{N} \sum_{i=1}^N R(c^i, x^i) \log P_\theta(x^i c^i)$
Gradient	$\frac{1}{N} \sum_{i=1}^N \nabla \log P_\theta(\hat{x}^i c^i)$	$\frac{1}{N} \sum_{i=1}^N R(c^i, x^i) \nabla \log P_\theta(x^i c^i)$
Training Data	$\{(c^1, \hat{x}^1), \dots, (c^N, \hat{x}^N)\}$ $R(c^i, \hat{x}^i) = 1$	$\{(c^1, x^1), \dots, (c^N, x^N)\}$ obtained from interaction weighted by $R(c^i, x^i)$

那这边是把 reinforcement learning 跟 maximum likelihood 呢，做一下比较，在做 maximum likelihood 的时候，你有一堆 training data，这些 training data 告诉我们说，今天假设人说 c_1 ，chatbot 最正确的回答是 x_1 hat，我们就会有 labeled 的 data 嘛，就你有 input c_1 output x_1 hat，，input c_N 正确答案就是 x_N hat，这是 training data 告诉我们的，在 training 的时候，你就是 maximize 你的 likelihood，怎么样 maximize 你的 likelihood 呢？

你希望 input c_i 的时候，output x_i hat 的机率越大越好，input 某个 condition，input 某个 input 的时候，input 某个输入的句子的时候，你希望正确的答案出现的机率越大越好，那算 gradient 的时候很简单，你就把这个 $\log P_\theta$ 前面呢，加上一个 gradient，你就算 gradient 了，这个是 maximum likelihood。

那我们来看一下 reinforcement learning，在做 reinforcement learning 的时候呢，你也会得到一堆 c 跟 x 的 pair，但这些 c 跟 x 的 pair，它并不是正确的答案，这些 x 并不是人去标的答案，这些 x 是机器自己产生的，就人输入 c_1 到 c_N ，机器自己产生了 x_1 到 x_N ，所以有些答案是对的，有些答案有可能是错的。

接下来呢我们说，我们在做 reinforcement learning 的时候，我们是怎么计算 gradient 的呢？我们是用这样的式子来计算 gradient，所以我们实际上的作法呢，我们这个式子的意思就是把这个 gradient $\log P_\theta$ 前面乘上 $R(c, x)$ 。

就如果你比较这两个式子的话，你会发现说他们唯一的差别是，在做 reinforcement learning 的时候，你在算 gradient 的时候，每一个 x 跟 c 的 pair 前面都乘上 $R(c, x)$ ，如果你觉得这个 gradient 算起来不太直观，那没关系，我们根据这个 gradient，反推 objective function。我们反推说什么样的 objective function，在取 gradient 的时候，会变成下面这个式子。那如果你反推了以后，你就会知道说，什么样的 objective function，取 gradient 以后会变成下面这个式子呢？你的 objective function 就是，summation over 你 sample 到的 data，每一笔 sample 到的 data，你都乘上 $R(c, x)$ ，然后你去计算每一笔 sample 到的 data 的 log 的 likelihood，你去计算每一笔 sample 到的 data 的 $\log P_\theta$ ，再把它乘上 $R(c, x)$ ，就是你的 objective function。

把这个 objective function，做 gradient 以后，你就会得到这个式子。我们在做 reinforcement learning 的时候，我们每一个 iteration，其实是在 maximize 这样一个 objective function，那如果你把这两个式子做比较的话，那就非常清楚了。右边这个 reinforcement learning 的 case，可以想成是每一笔 training data 都是有 weight，而在 maximum likelihood 的 case 里面，每一笔 training data 的 weight 都是一样的，每一笔 training data 的 weight 都是 1，在 reinforcement learning 里面，每一笔 training data 都有不同的 weight，这一个 weight 就是那一笔 training data 得到的 reward。

也就是说今天输入一个 c_i ，机器回答一个 x_i ，如果今天机器的回答正好是好的，这个 x_i 是一个正确的回答，那我们在 training 的时候就给那笔 data 比較大的 weight。如果今天 x_i 是一个不好的回答，代表这笔 training data 是错的，我们 even 会给它一个 negative 的 weight，这个就是 maximum likelihood，和 reinforcement learning 的比較。

reward理论上并没有特别的限制，你用 policy gradient，都可以去 maximize objective function，但是在实作上，会有限制，我们刚才不是讲到说，如果 R 是正的，你就要让机率越大越好，那你会不会遇到一个问题就是，假设 R 永远都是正的，今天这个 task R 就是正的，你做的最差，也只是得到的分数比较小而已，它永远都是正的，那今天不管你采取什么样的行为，machine 都会说我要让机率上升，听来有点怪怪的。但是在理论上这样未必会有问题。

为什么说理论上这样未必会有问题呢？你想想看，你要 maximize 的这一项，是一个机率，它的和是 1，所以今天就算是所有不同的 x_i ，他前面乘的 R 是正的，他终究是有大有小的，你不可能让所有的机率都上升，因为机率的和是 1，你不可能让所有机率都上升，所以变成说，如果 weight 比較大的，就比較 positive 的，就上升比较多，如果 weight 比較小的，比較 negative 的，它就可能反而是会減少的，就算是正的，但如果值比較小，它可能也是会減小，因为 constrain 就是它的和要是 1。

但是你今天在实作上并没有那么容易，因为在实作上会遇到的问题是，你不可能 sample 到所有的 x ，所以到时候就会变成说，假设一笔 data 你没有 sample 到，其他人只要有 sample 到都是 positive 的 reward。没 sample 到的，反而就会机率下降，而 sample 到的都会机率上升。这个反而是我们要的。所以其实今天在设计那个 reward 的时候，你其实会希望那个 reward 是有正有负的，你 train 起来会比較容易，那假设你的 task reward 都是正的，实际上你会做的一件事情是，把 reward 通通都减掉一个 threshold，让它变成是有正有负，这样你 train 起来会容易很多。

这个是讲了 maximum likelihood，跟 reinforcement learning 的比較。

Alpha GO style training

但是你知道实作上要做什么 reinforcement learning 根本就是不太可能的，有一个人写一篇网络文章说，当有人问他说某一个 task 用 reinforcement learning 好不好的时候，他的回答都是不好，多数的时候他都是对的。

要做 reinforcement learning 一个最大的问题就是，机器必须要跟人真的互动很多次，才能够学得起来。

你不要看今天 google 或者是 Deep mind 或者是 OpenAI 他们在玩那些什么 3D 游戏都玩得很好这样，那个 machine 跟环境互动的次数都可能是上千万次，或者是上亿次，那么多互动的次数，除了在电玩这种 simulated 的 task 以外，在真实的情境，几乎是不可能发生。

所以如果你要用 reinforcement learning 去 train 一个 chatbot，几乎是不可能的。

因为在现实的情境中，人没有办法花那么多力气，去跟 chatbot 做互动，所以来就有人想了一个 Alpha Go style training。也就是说我们 learn 两个 chatbot，让它们去互讲，例如有一个 bot 说 How are you。另外一个说 see you，然后它再说 see you，它说 see you，然后陷入一个无穷循环永远都跳不出来。它们有时候可能也会说出比较正确的句子，因为我们知道说机器在回应的时候其实是有随机性的。所以问它同一个句子，每次的回答不见得是一样的。

接下来你再去定一个 evaluation 的 function，因为你还是不可能说让两个 chatbot 互相对话，然后产生一千万则对话以后，人再去一千万则对话每一个去给它 feedback 说，讲得好还是不好，你可能会设计一个 evaluation function，这个就是人订一个 evaluation function，给一则对话，然后看说这则对话好不好，但是这种 evaluation function 是人订的，你其实没有办法真的定出太复杂的 function，就只能定义一些很简单的。就是，比如说陷入无穷循环，就是得到负的 reward，说出 I don't know，就是得到负的 reward，你根本没有办法真的订出太复杂的 evaluation function，所以用这种方法还是有极限的。

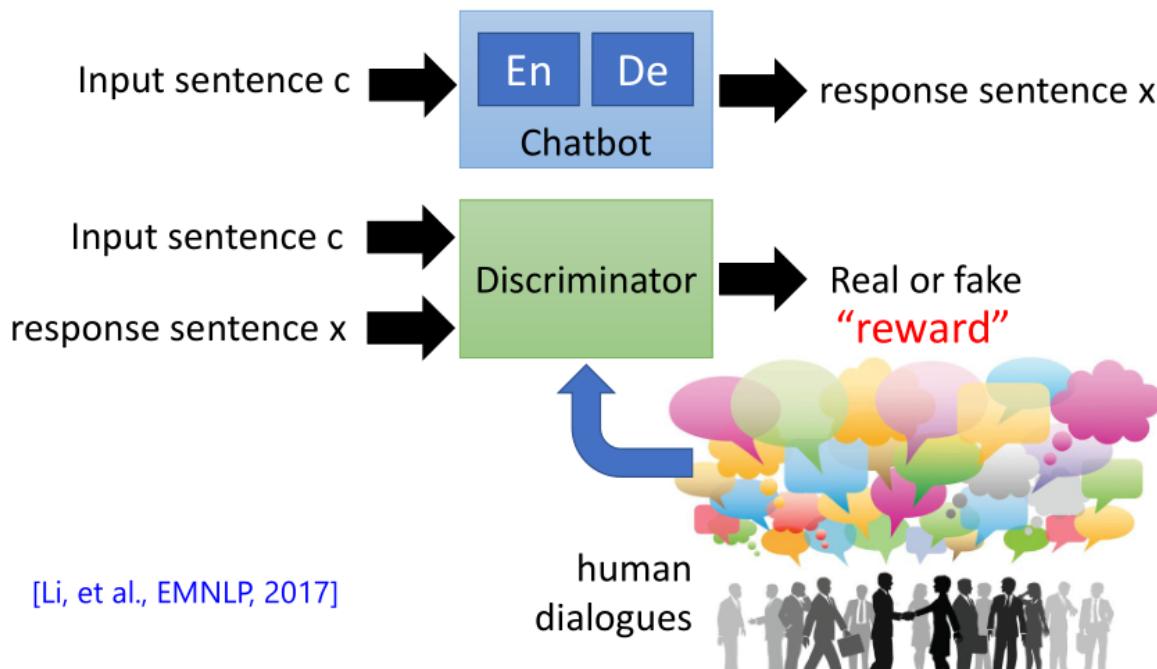
所以接下来要解这个问题，你可以引入 GAN 的概念。

GAN (discriminator feedback)

GAN 和 RL 有什么不同呢？在 RL 里面，你是人给 feedback，在 GAN 里面，你变成是 discriminator 来给 feedback。

我们一样有一个 chatbot，一样吃一个句子，output 另外一个句子，现在有一个 discriminator，这个 discriminator，其实就是取代了人的角色，它吃 chatbot 的 input 跟 output，然后吐出一个分数。

Conditional GAN



那这个跟 typical 的 conditional GAN 就是一样的，我们知道就算是别的 task，什么 image 的生成，你做的事情也是一样的。你就是有一个 discriminator，它吃你的 generator 的 input 跟 output，接下来给你一个评价，那在 chatbot 里面也是一样，你有一个 discriminator，它吃 chatbot input 的 sentence，跟 output sentence，然后给予一个评价。那这个 discriminator 呢，你要给它大量人类的对话，让它知道说真正的人类的对话，真正的当这个chatbot 换成一个人的时候，它的 c 跟 x 长什么样子，那这个 discriminator 就会学着鉴别这个 c 跟 x 的 pair，是来自于人类，还是来自于 chatbot。然后 discriminator 会把他学到的东西，feedback 给 chatbot，或者是说 chatbot 要想办法骗过这个 discriminator。那这跟 conditional GAN 就是一模一样的事情了。

Algorithm

那这个 algorithm 是什么样子呢？

其实这个 discriminator 的 output，就可以想成是人在给 reward，你要把这个 discriminator，想成是一个人，只是这个 discriminator 和人不一样的地方是，它不是完美的，所以要去更新它自己的参数。整个 algorithm 其实就跟传统的 GAN是一样的，传统 conditional GAN 是一样的。

Algorithm

Training data:

Pairs of conditional input c and response x

- Initialize generator G (chatbot) and discriminator D

- In each iteration:

- Sample input c and response x from training set
- Sample input c' from training set, and generate response \tilde{x} by $G(c')$
- Update D to increase $D(c, x)$ and decrease $D(c', \tilde{x})$

- Update generator G (chatbot) such that



你有 training data，这些 training data，就是一大堆的正确的 c 跟 x 的 pair。

然后你一开始你就 initialize 一个 G ，其实你的 G 就是你的 generator 你的 chatbot，然后 initialize 你的 discriminator D 。

在每一个 training 的 iteration 里面，你从你的 training data 里面，sample 出正确的 c 跟 x 的 pair。

你从你的 training data 里面 sample 出一个 c prime，然后把这个 c prime 丢到你的 generator 也就是 chatbot 里面，让它回一个句子 x tilde，那这个 c prime, x tilde 就是一个 native 的一个 example。

接下来discriminator 要学着说，看到正确的 c 跟 x ，给它比较高的分数，看到错误的 c prime 跟 x tilde，给它比较低的分数。

至于怎么 train 这个 discriminator，你可以用传统的方法，量 js divergence 的方法，你完全也可以套用 WGAN，都是没有问题的。

那接下来的问题是说，我们知道在 GAN 里面你 train discriminator 以后，接下来你就要 train 你的 chatbot，也就是 generator。

那 train generator 他的目标是什么呢？你要 train 你的 generator，这个 generator 的目标就是要去 update 参数，然后你 generator 产生出来的 c 跟 x 的 pair，能让 discriminator 的 output 越大越好，那这个就是 generator 要做的事情。

这边要做的事情，跟我们之前看到的 conditional GAN，其实是一模一样的，我们说 generator 要做的事情，其实就是要去做骗过 discriminator。

但是这边我们会遇到一个问题。什么样的问题呢？如果你仔细想一想你的 chatbot 的 network 的架构的话，我们的 chatbot 的 network 的架构它是一个 seq to seq 的 model，它是一个 RNN 的 generator。

我们看 chatbot 在 generate 一个 sequence 的时候，它 generate sequence 的 process 是这样子的。一开始你给它一个 condition，这个 condition 可能是从 attention based model 来的，给它一个 condition，然后它 output 一个 distribution，那根据这个 distribution 它会去做一个 sample，就 sample 出一个 token，sample 出一个 word，然后接下来你会把这个 sample 出来的 word，当作下一个 time step 的 input，再产生新的 distribution，再做 sample，再当做下一个 time step 的 input，再产生 distribution。

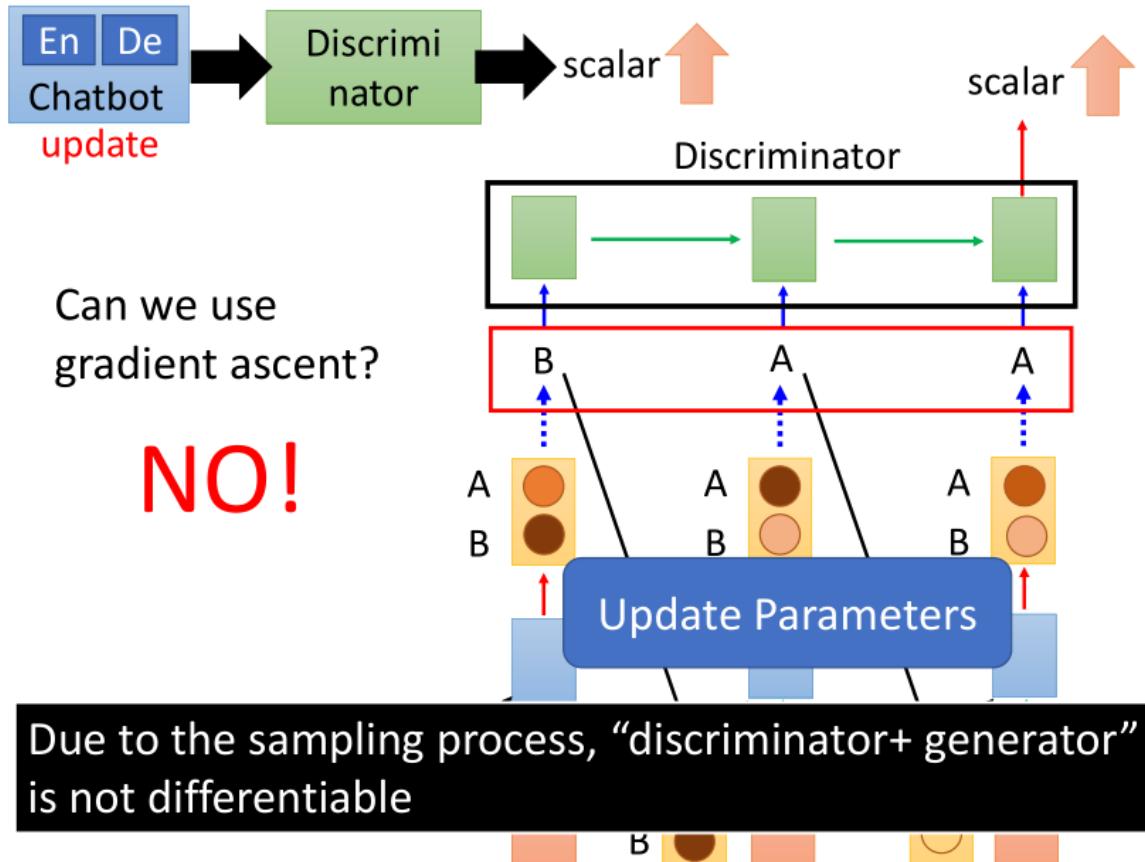
然后我们说我们要把 generator 的 output，丢给 discriminator，你对这个 discriminator 的架构，你也是自己设计，反正只要可以吃两个 sequence。注意一下这个 discriminator，前一页的图，只有画说它吃 chatbot 的 output，但它不能只吃 chatbot 的 output，它是同时吃 chatbot 的 input 和 output。

在做 conditional GAN 的时候，你的 discriminator 要同时吃你的 generator 的 input 和 output。所以其实这个 discriminator，是同时吃了这个 chatbot 的 input 跟 output，就是两个 word sequence。

那至于这个 discriminator network 架构要长什么样子，这个就是看你高兴。你可以说你就 learn 一个 RNN，然后你把 chatbot input 跟 output 把它接起来，变成一个很长的 sequence。然后 discriminator 把这个很长的 sequence 就读过，然后就吐出一个数值，这样也是可以的。有人说我可以用 CNN，反正只要吃两个 sequence，可以吐出一个分数，怎么样都是可以的。

那反正 discriminator 就吃一个 word sequence，接下来他吐出一个分数。当我们知道说假设我们今天要，train generator 去骗过 discriminator，我们要做的事情是，update generator 的参数，update 这个 chatbot seq to seq model 的参数，让 discriminator 的 output 的 scalar 越大越好，这件事情你仔细想一下，你有办法做吗？你想说这个很简单啊，就是把 generator 跟 discriminator 串起来就变成一个巨大的 network，然后我们要做的事情就是，调这个巨大的 network 的前面几个 layer 让，这个 network 最后的 output 越大越好。

但是你会遇到的问题是，你发现这个 network 其实是没有办法微分的，为什么它没有办法微分？这整个 network 里面有一个 sampling 的 process，这跟我们之前在讲 image 的时候，是不一样的。



我觉得这个其实是要用 GAN 来做 natural language processing，跟你用 GAN 来做 image processing 的时候，一个非常不一样的地方。在 image 里面，当你用 GAN 来产生一张影像的时候，你可以直接把产生的影像，丢到 discriminator 里面，所以你可以把 generator 跟 discriminator 合起来，看作是一个巨大的 network。

但是今天在做文字的生成的时候，你生成出一个 sentence，这个 sentence 是一串 sequence，是一串 token，你把这串 token 丢到 discriminator 里面，你要得到这个 token 的时候，这中间有一个 sampling 的 process。

当一整个 network 里面有一个 sampling 的 process 的时候，它是没有办法微分的。一个简单的解释是，你想看所谓的微分的意思是什么？微分的意思是你把某一个参数小小的变化一下，看它对最后的 output 的影响有多大。这两个相除，就是微分。那今天假设一个 network 里面有 sampling 的 process，你把里面的参数做一下小小的变化，对 output 的影响是不确定的，因为中间有个 sampling 的 process，所以你每次得到的 output 是不一样的。你今天对你整个 network 做一个小小的变化的时候，它对 output 的影响是不确定的，所以你根本就没有办法算微分出来。

另外一个更简单的解释就是，你回去用TensorFlow 或 PyTorch implement 一下，看看如果 network 里面有一个 sampling 的 process，你跑不跑得动这样子，你应该是会得到一个 error，应该是跑不动的，结果就是这样。

反正无论如何，今天你把这个 seq to seq model，跟你的 discriminator 接起来的时候，你是没有办法微分的。所以接下来真正的难点就是，怎么解这个问题。

Three Categories of Solutions

那我在文献上看到，大概有三类的解法，一个是 Gumbel-softmax，一个是给 discriminator continuous input，另外一个方法就是做 reinforcement learning。

Gumbel-softmax

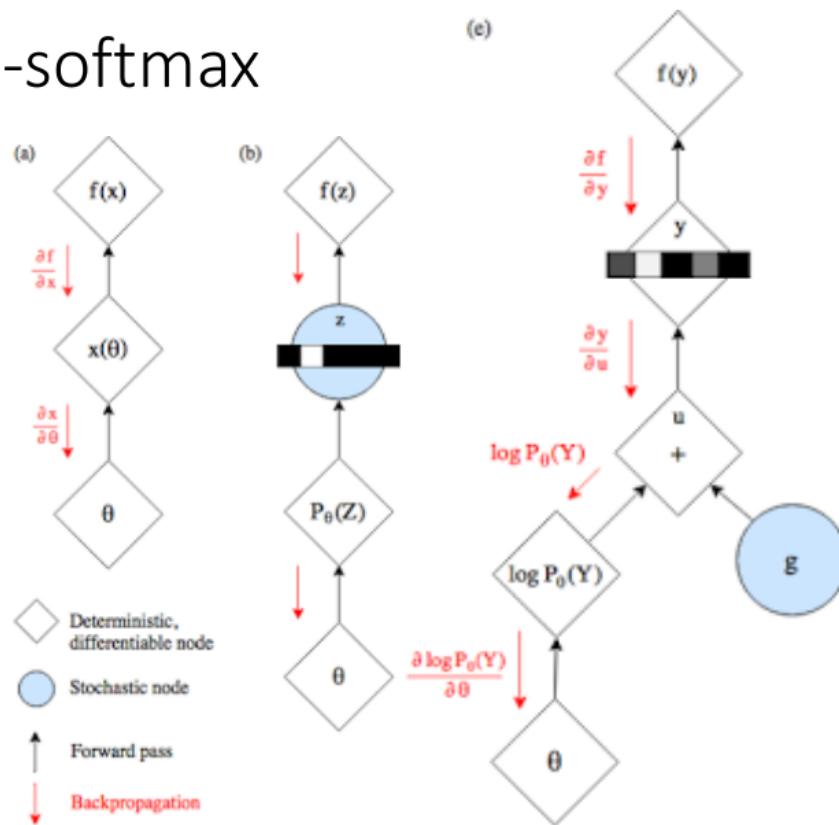
Gumbel-softmax 我们就不解释，那它其实 implement 也是蛮简单的，但是我发现用在 GAN 上目前没有那么多，所以我们就不解释。总之 Gumbel-softmax 就是想了一个 trick，让本来不能微分的东西，somehow 变成可以微分，如果你有兴趣的话，你再自己研究 Gumbel-softmax 是怎么做的。

Gumbel-softmax

<https://gabrielhuang.gitbooks.io/machine-learning/reparametrization-trick.html>

<https://casmls.github.io/general/2017/02/01/GumbelSoftmax.html>

<http://blog.evjang.com/2016/11/tutorial-categorical-variational.html>

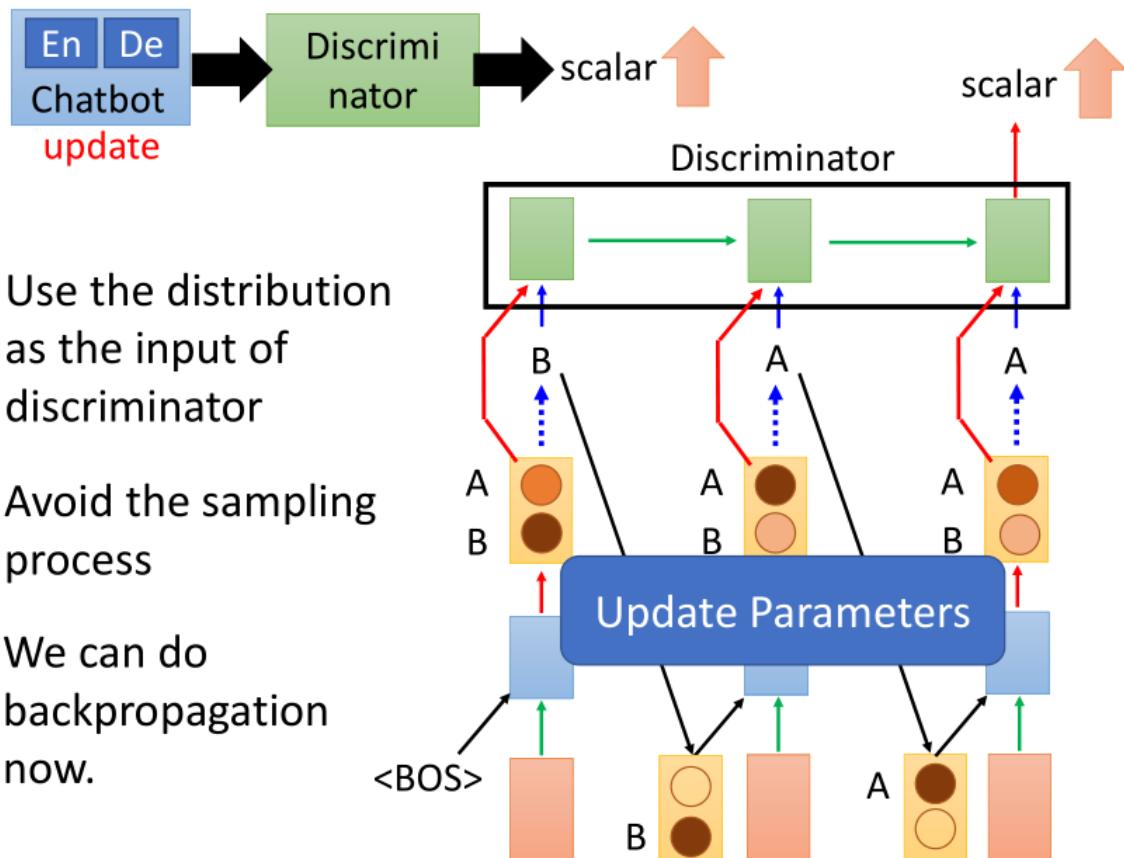


Continuous Input for Discriminator

那另外一个很简单的方法就是，给 discriminator continuous 的 input。

你说今天如果问题是在这一个 sampling 的 process，那我们何不就避开 sampling process 呢。discriminator 不是吃 word sequence，不是吃 discrete token，来得到分数，而是吃 word distribution，来得到分数。

那今天如果我们把这一个 seq to seq model，跟这个 discriminator 串在一起，你就会发现说它变成一个是可以微分的 network 了，因为现在没有那一个 sampling process 了，问题就解决了。



但是实际上问题并没有这么简单，仔细想想看当你今天给你的 discriminator一个 continuous input 的时候，你会发生什么样的问题。

你会发生的问题是这样，Discriminator 会看 real data 跟 fake data，然后去给它一笔新的 data 的时候，它会决定它是 real 还是 fake 的。

当你今天给 discriminator word distribution 的时候，你会发现说，real data 跟 fake data 它在本质上就是不一样的。

Real sentence

Generated

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Discriminator can immediately find the difference.

Can never be 1-of-N

0.9	0.1	0.1	0	0
0.1	0.9	0.1	0	0
0	0	0.7	0.1	0
0	0	0.1	0.8	0.1
0	0	0	0.1	0.9

WGAN is helpful

因为对 real data 来说，它是 discrete token，或者是说每一个 discrete token，我们其实是用一个 1 one-hot 的 vector 来表示它。对一个 discrete token，我们是用 1 one-hot vector 来表示它。

而对 generator 来说，它每次只会 output 一个 word distribution，它每次 output 的都是一个 distribution。

所以对 discriminator 来说，要分辨今天的 input 是 real 还是 fake 的，太容易了，他完全不需要管这个句子的语义，它完全不管句子的语义，它只要一看说，是不是 one-hot，就知道说它是 real 还是 fake 的。

所以如果你直接用这个方法，来 train GAN 的话，你会发现会遇到什么问题呢？你会发现，generator 很快就会发现说 discriminator，判断一笔 data 是 real 还是 fake 的准则，是看说今天你的每一个 output，是不是 one-hot 的，所以 generator 唯一会学到的事情就是，迅速的变成 one-hot，它会想办法赶快把某一个，随便选一个 element 谁都好，也不要在意语意了，因为就算你考虑语意，也很快会被 discriminator 发现，因为 discriminator 就是要看说是不是 one-hot。

所以今天随便选一个 element，想办法赶快把它的值变到 1，其他都赶快压成 0，然后产生的句子完全不 make sense，然后就结束了。

你会发现所以今天直接让 discriminator，吃 continuous input 是不够的，是没有办法真的解决这个问题。

那其实还有一个解法是，也许用一般的 GAN，train 不起来，但是你可以试试看用 WGAN。为什么在这个 case 用 WGAN，是有希望的呢？

因为 WGAN 在 train 的时候，你会给你的 model 一个 constrain，你要去 constrain 你的 discriminator 一定要是 1-Lipschitz function。因为你有这个 constrain，所以你的 discriminator 它的手脚会被绑住，所以它就没有办法马上分别出 real sentence，跟 generated sentence 的差别。它的视线是比较模糊的，它是比较看不清楚的。因为它有一个 1-Lipschitz function constrain，所以它是比较 fuzzy 的，所以它就没有办法马上分别这两者的差别。

所以今天假设你要做 conditional generation 的时候呢，如果你是要做这种 sequence generation，然后你要用的方法是让 discriminator 吃 continuous input，WGAN 是一个可以的选择。

如果你没有用 WGAN 的话，应该是很难把它做起的，因为 generator 其实学不到语意相关的东西，它只学到说，output 必须要像是 one-hot，才能够骗过 discriminator。

所以这个是第二个 solution，给它 continuous input。

Reinforcement Learning

第三个 solution 呢，就是套用 RL。

我们刚才已经讲过说，假设这个 discriminator，换成一个人的话，你知道怎么去调你 chatbot 的参数，去 maximize 人会给予 chatbot 的 reward。

那今天把人换成 discriminator，solution 其实是一模一样的。怎么解这个问题呢？

也就是说现在呢，discriminator 就是一个 human，我们说人其实就是一个 function 嘛，然后看 chatbot 的 input output 给予分数，所以 discriminator 就是我们的人，它的 output，它的 output 那个 scalar，discriminator output 的那个数值，就是 reward，然后今天你的 chatbot 要去调它的参数，去 maximize discriminator 的 output，也就是说本来人的 output 是 $R(c, x)$ ，那我们只是把它换成 discriminator 的 output $D(c, x)$ ，就结束了。



Consider the output of discriminator as **reward**

- Update generator to increase discriminator = to get maximum reward
- Using the formulation of policy gradient, replace reward $R(c, x)$ with discriminator output $D(c, x)$

Different from typical RL

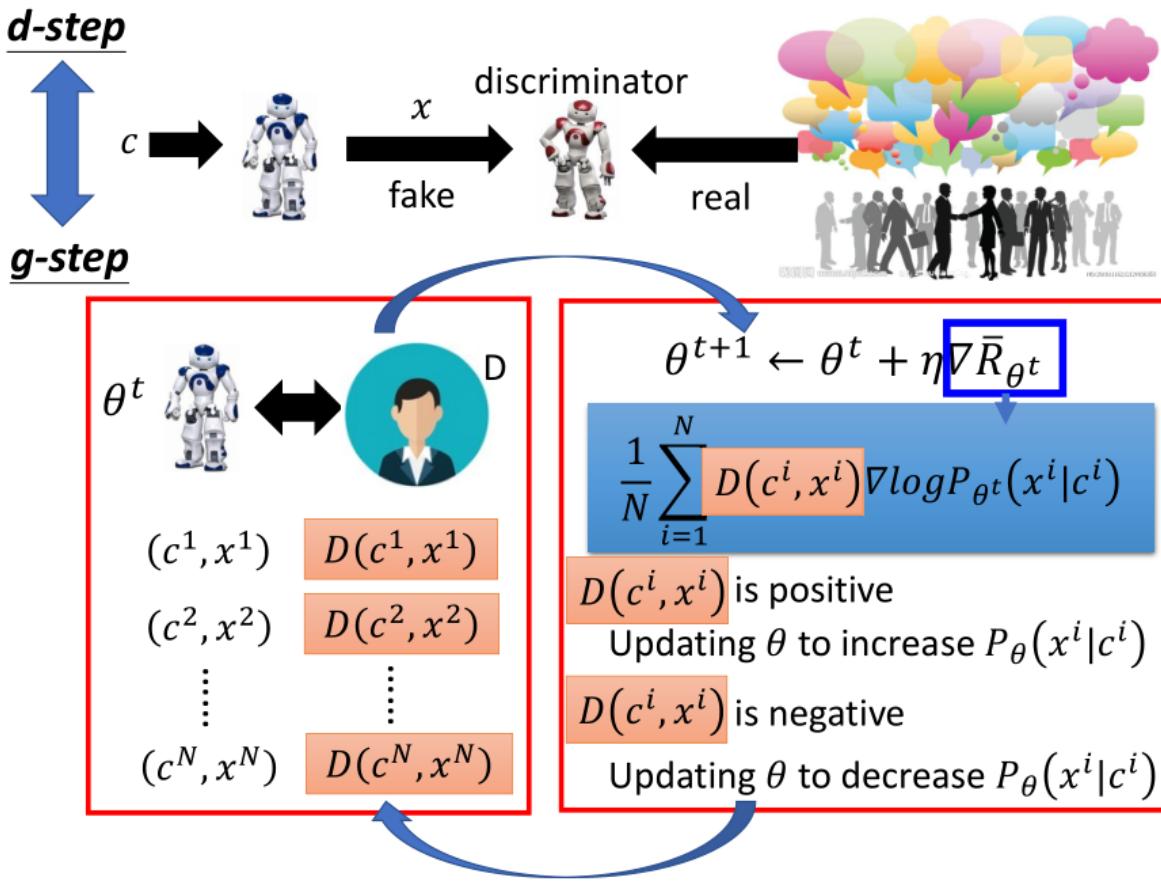
- The discriminator would update

接下来怎么 maximize $D(c, x)$ ，你在 RL 怎么做，在这边就怎么做。

所以呢，我们说在这个 RL 里面是怎么做的呢？你让 θ 去跟人互动，然后得到很多 reward，接下来套右边这个式子，你就可以去 train 你的 model。

现在我们唯一做的事情，是把人呢，换成另外一个机器，就是换成 discriminator，本来是人给 reward，现在换成 discriminator 给 reward。

我们唯一做的事情，就是把 R 换成 D，所以右边也是一样，把 R 换成 D。



当然这样跟人互动还是不一样，因为人跟机器互动很花时间嘛，那如果是 discriminator，它要跟 generator 互动多少次，反正都是机器，你就可以让它们真的互动非常多次。

但是这边只完成了 GAN 的其中一个 step 而已，我们知道说在 GAN 的每一个 iteration 里面，你要 train generator，你要 train discriminator 再 train generator，再 train discriminator，再 train generator。

今天这个 RL 的 step 只是 train 了 generator 而已，接下来你还要 train discriminator，怎么 train discriminator 呢？

你就给 discriminator 很多人真正的对话，你给 discriminator 很多，现在你的这个 generator 产生出来的对话，你给 discriminator 很多 generator 产生出来的对话，给很多人的对话，然后 discriminator 就会去学着分辨说这个对话是 real 的，是真正人讲的，还是 generator 产生的。

那你就可以学出一个 discriminator，那你学完 discriminator 以后，因为你的 discriminator 不一样了，这边给的分数当然也不一样了，你 train 好 discriminator 以后，再回头去 train generator，再回头去 train discriminator，这两个 step 就反复地进行。这个就是用 GAN 来 train seq to seq model 的方法。

那其实还有很多的 tip，那这边也稍跟大家讲一下，那如果我们看这个式子的话，你会发现有一个问题，什么样的问题呢？

Reward for Every Generation Step

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{i=1}^N D(c^i, x^i) \nabla \log P_\theta(x^i | c^i)$$

$c^i = \text{"What is your name?"}$ $D(c^i, x^i)$ is negative
 $x^i = \text{"I don't know"}$ Update θ to decrease $\log P_\theta(x^i | c^i)$

$$\log P_\theta(x^i | c^i) = \log P(x_1^i | c^i) + \log P(x_2^i | c^i, x_1^i) + \log P(x_3^i | c^i, x_{1:2}^i)$$

$$P(\text{"I"} | c^i) \downarrow ? \downarrow \downarrow$$

$c^i = \text{"What is your name?"}$ $D(c^i, x^i)$ is positive
 $x^i = \text{"I am John"}$ Update θ to increase $\log P_\theta(x^i | c^i)$

$$\log P_\theta(x^i | c^i) = \log P(x_1^i | c^i) + \log P(x_2^i | c^i, x_1^i) + \log P(x_3^i | c^i, x_{1:2}^i)$$

$$P(\text{"I"} | c^i) \uparrow \uparrow \uparrow$$

这个式子跟刚才那个 RL 看到的式子是一样的，我们只是把 R 换成了 D。今天假设 c_i 是 what is your name，然后 x_i 是 I don't know，这可能不是一个很好的回答，所以你得到的 discriminator 给它的分数是负的。当 discriminator 给它的分数是负的时候，我们希望调整我们的参数 θ ，让 $\log P_\theta(x^i | c^i)$ 的值变小，那我们再想想看， $P_\theta(x^i | c^i)$ ，到底是什么样的东西呢？它其实是一大堆 term 的连乘。

也就是说，我们今天实际上在做 generation 的时候，我们每次只会 generate 一个 word 而已。我们假设 I don't know 这边有三个 word，第一个 word 是 x_1 ，第二个 word 是 x_2 ，第三个 word 是 x_3 。那你说让这个机率下降，你希望他们每一项都下降。

但是我们看看 P of (c_i , given x_1) 是什么 is what is your name 的时候，产生 I 的机率，那如果输入 what is your name? 一个好的答案其实可能是比如说 I am John。所以今天问 What is your name 的时候，你其实回答 I 当作句子的开头是好的，但是你在 training 的时候，你却告诉 chatbot 说，看到 What is your name 的时候，回答 I 这个机率，应该是下降的。

看到 What is your name? 你已经产生 I，产生 don't 的机率要下降，这项是合理的，产生 I don't 再产生 know 的机率要下降是合理的，但是 given What is your name? 产生 I 的机率要下降，其实是不合理的。

那这个 training 不是有问题吗？理论上这个 training 不会有问题，因为今天你的 output，其实是一个 sampling 的 process，所以今天在另外一个 case，当你输入 What is your name 的时候，机器的回答可能是 I am John，这个时候机器就会得到一个 positive 的 reward，也就是 discriminator 会给机器一个 positive 的评价。这个时候 model 要做的事情就是 update 它的参数，去 increase $\log P_\theta(x^i | c^i)$ ，那 $P_\theta(x^i | c^i)$ ，是这三个项的相乘，而第一项是 $P(I | c^i)$ ，我们会希望它越大越好，当你输入 What is your name? sample 到 I don't know 的时候， $P(I | c^i)$ 要减小，当你 sample 到 I am John 的时候，你希望这个机率上升，那如果你今天 sample 的次数够多，这两项就会抵消，那就没事了。

但问题就是在实作上，你永远 sample 不到够多的次数，所以在实作上这个方法是会造成一些问题的，所以怎么办呢？

Reward for Every Generation Step

$h^i = \text{"What is your name?"}$ $x^i = \text{"I don't know"}$

$$\log P_{\theta}(x^i|h^i) = \log P(x_1^i|c^i) + \log P(x_2^i|c^i, x_1^i) + \log P(x_3^i|c^i, x_{1:2}^i)$$

$$\nabla \bar{R}_{\theta} \approx \frac{1}{N} \sum_{i=1}^N D(c^i, x^i) \nabla \log P_{\theta}(x^i | c^i)$$

$$\nabla \bar{R}_{\theta} \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T (Q(c^i, x_{1:t}^i) - b) \nabla \log P_{\theta}(x_t^i | c^i, x_{1:t-1}^i)$$

Method 1. Monte Carlo (MC) Search [Yu, et al., AAAI, 2017]

Method 2. Discriminator For Partially Decoded Sequences

[Li, et al., EMNLP, 2017]

今天的 solution 是这个样子，我们今天希望当输入 What is your name? sample 到 I don't know 的时候，machine 可以自动知道说，在这三个机率里面，虽然 I don't know 整体而言是不好的，但是造成 I don't know 不好的原因，并不是因为在开头 sample 到了 I，在开头 sample 到 I，是没有问题的，是因为之后你产生了 don't 跟 know，所以才做得不好。所以希望机器可以自动学到说，今天这个句子不好，到底是哪里不好，是因为产生这两个 word 不好，而不是产生第一个 word 不好。

那所以你今天会改写你的式子，现在你给每一个 generation step，都不同的分数，今天在给定 condition c_i ，已经产生前 $t-1$ 个 word 的情况下，产生的 word x_t ，它到底有多好或多不好。

我们换另外一个 measure 叫做 Q ，来取代 D ，这个 Q 它是对每一个 time step 做 evaluation，它对这边每一次 generation 的 time step 做 evaluation，而不是对整个句子做 evaluation。

这件事情要怎么做呢？你如果想知道的话，你就自己查一下文献，那有不同的作法，这其实是一个还可以尚待研究中的问题。

一个作法就是做 Monte Carlo，跟 Alpha Go 的方法非常像，你就想成是在做 Alpha Go，你去 sample 接下来会发生到的状况，然后去估测每一个 generation，每一个 generation 就像是在棋盘上下一个子一样，可以估测每一个 generation 在棋盘上落一个子的胜率。那这个方法最大的问题就是，它需要的运算量太大，所以在实作上你会很难做。

那有另外一个运算量比较小的方法，这个方法它的缩写叫做 REGS，不过这个方法，在文献上看到的结果就是它不如 Monte Carlo，我自己也有实作过，觉得它确实不如 Monte Carlo。但 Monte Carlo 的问题是，它的运算量太大了，所以这个仍然是一个目前可以研究的问题。

那还有另外一个技术可以 improve 你的 training，这个方法，叫做 RankGAN。

Tips: RankGAN

Kevin Lin, Dianqi Li, Xiaodong He, Zhengyou Zhang, Ming-Ting Sun, "Adversarial Ranking for Language Generation", NIPS 2017

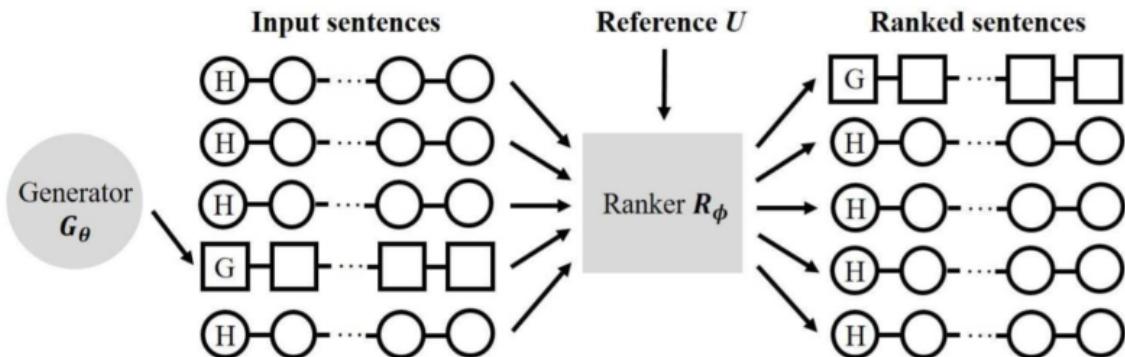


Image caption generation:

Method	BLEU-2	BLEU-3	BLEU-4	Method	Human score
MLE	0.781	0.624	0.589	SeqGAN	3.44
SeqGAN	0.815	0.636	0.587	RankGAN	4.61
RankGAN	0.845	0.668	0.614	Human-written	6.42

那这边是讲一些我们自己的Experimental Results, 今天到底把 maximum likelihood, 换到 GAN 的时候, 有什么样的不同呢?

事实上如果你有 train 过 chatbot 的话, 你会知道说, 今天 train 完以后, chatbot 非常喜欢回答一些没有很长, 然后非常 general 的句子, 通常它的回答就是 I'm sorry, 就是 I don't know, 这样讲来讲去都是那几句。我们用一个 benchmark corpus 叫 Open subtitle 来 train 一个 end to end 的 chatbot 的时候, 其实有 1/10 的句子, 它都会回答 I don't know 或是 I'm sorry, 这听起来其实是没有非常 make sense。

那如果你要解这个问题, 我觉得 GAN 就可以派上用场, 为什么今天会回答 I'm sorry 或 I don't know 呢? 我的猜测是, 这些 I'm sorry 或 I don't know 这些句子, 对应到影像上, 就是那些模糊的影像。

我们有讲过说, 为什么我们今天在做影像生成的时候要用 GAN, 而不是传统的 supervised learning 的方法, 是因为, 今天在做影像的生成的时候, 你可能同样的 condition, 你有好多不同的对应的 image, 比如说火车有很多不同的样子, 那机器在学习的时候, 它是会产生所有火车的平均, 然后看起来是一个模糊的东西。

那今天对一般的 training 来说, 假设你没有用 GAN 去 train 一个 chatbot 来说, 也是一样的, 因为输入同一个句子, 在你的 training data 里面, 有好多个不同的答案, 对 machine 来说他学习的结果就是希望去, 同时 maximize 所有不同答案的 likelihood。但是同时 maximize 所有答案的 likelihood 的结果, 就是产生一些奇怪的句子。那我认为这就是导致为什么 machine, 今天用 end to end 的方法, 用 maximum likelihood 的方法, train 完一个 chatbot 以后它特别喜欢说 I'm sorry, 或者是 I don't know。

那用 GAN 的话, 一个非常明显你可以得到的结果是, 用 GAN 来 train 你的 chatbot 以后, 他比较喜欢讲长的句子, 那它讲的句子会比较有内容, 就这件事情算是蛮明显的。

Input	We've got to look for another route.
MLE	I'm sorry.
GAN	You're not going to be here for a while.
Input	You can save him by talking.
MLE	I don't know.
GAN	You know what's going on in there, you know what I mean?

- MLE frequently generates “I'm sorry”, “I don't know”, etc. (corresponding to fuzzy images?)
 - GAN generates longer and more complex responses (however, no strong evidence shows that they are better)
-

Find more comparison in the survey papers.

[Lu, et al., arXiv, 2018][Zhu, et al., arXiv, 2018]

那一个比较不明显的地方是我们其实不确定说，产生比较长的句子以后，是不是一定就是比较好的对话。

但是蛮明显可以观察到说，当你把原来 MLE 换成 GAN 的时候，它会产生比较长的句子。

More Applications

- Supervised machine translation [Wu, et al., arXiv 2017][Yang, et al., arXiv 2017]
- Supervised abstractive summarization [Liu, et al., AAAI 2018]
- Image/video caption generation [Rakshith Shetty, et al., ICCV 2017][Liang, et al., arXiv 2017]

If you are using seq2seq models,
consider to improve them by GAN.

那其实各种不同的 seq to seq model 都可以用上 GAN 的技术，如果你今天在 train seq to seq model 的时候，你其实可以考虑加上 GAN，看看 train 的会不会比较好。

Unsupervised Conditional Sequence Generation

刚才讲个 conditional sequence generation，那还是 supervised 的，你要有 seq to seq model 的 input 跟 output。接下来要讲 Unsupervised conditional sequence generation。

Text Style Transfer

那我们先讲 Text style transformation，那我们今天已经看过满坑满谷的例子是做image style transformation。

那其实在文字上，你也可以做 style 的 transformation，什么叫做文字的 style 呢？我们可以把正面的句子算做是一种 style，负面的句子算做是另一种 style，接下来你只要 apply cycle GAN 的技术，把两种不同 style 的句子，当作两个 domain，你就可以用 unsupervised 的方法。

你并不需要两个 domain 的文字句子的 pair，你并不需要知道说这个 positive 的句子应该对应到哪一个 negative 的句子，你不需要这个信息，你只需要两堆句子，一堆 positive，一堆 negative，你就可以直接 train 一个 style transformation。

那我们知道说其实要做这种你要知道，一个句子是不是 positive 的，其实还蛮容易的，因为我们在 ML 的作业 5 里面，你就会 train 一个 RNN，那你就把你 train 过那个 RNN 拿出来，然后给他一堆句子，然后如果很 positive，就放一堆，很 negative 就放一堆，你就自动有 positive 跟 negative 的句子了。那这个技术怎么做呢？

我们不需要多讲，image style transformation 换成 text style transfer，唯一做的事情就是影像换成本字。

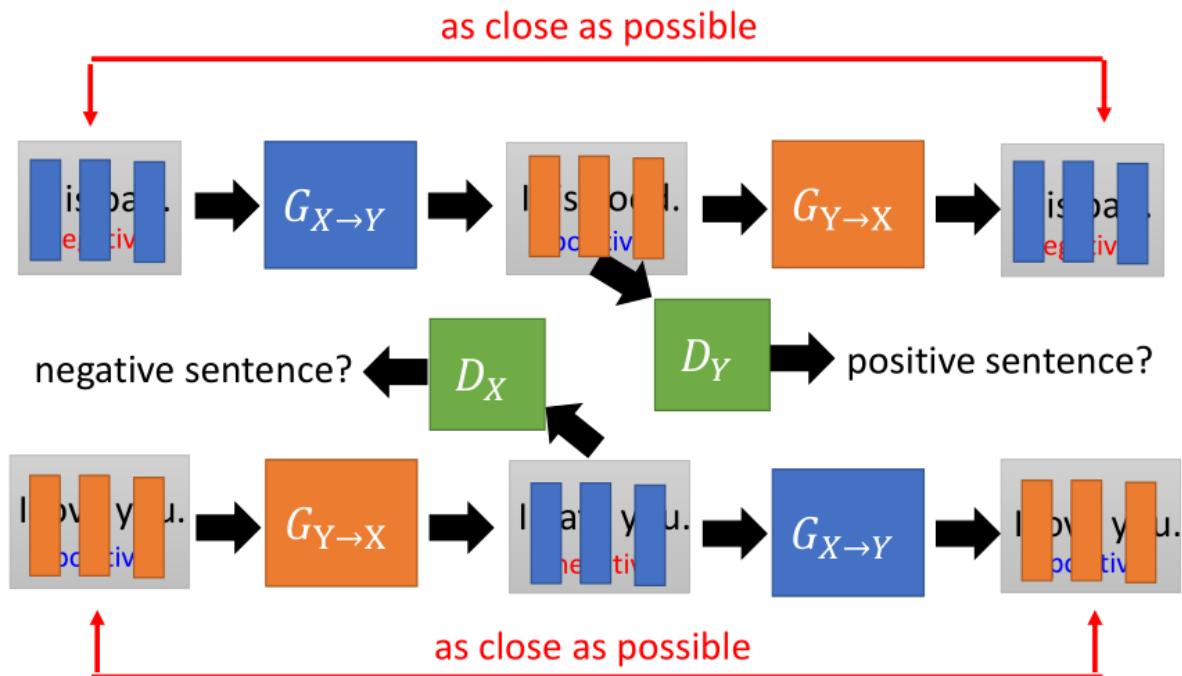
所以我们就把 positive 的句子算是一个 domain，negative 的句子算是另外一个 domain，用 cycle GAN 的方法 train 下去就结束了。

那你这边可能会遇到一个问题是，我们刚才有讲到说，如果今天你的 generator 的 output，是 discrete 的，你没有办法直接做 training，假设你今天你的 generator output 是一个句子，句子是一个 discrete 的东西，你用一个 sampling 的 process，你才能够产生那个句子，当你把这两个 generator，跟这个 discriminator 全部串在一起的时候，你没办法一起 train，那怎么办呢？

Discrete?

Direct Transformation

Word embedding
[Lee, et al., ICASSP, 2018]



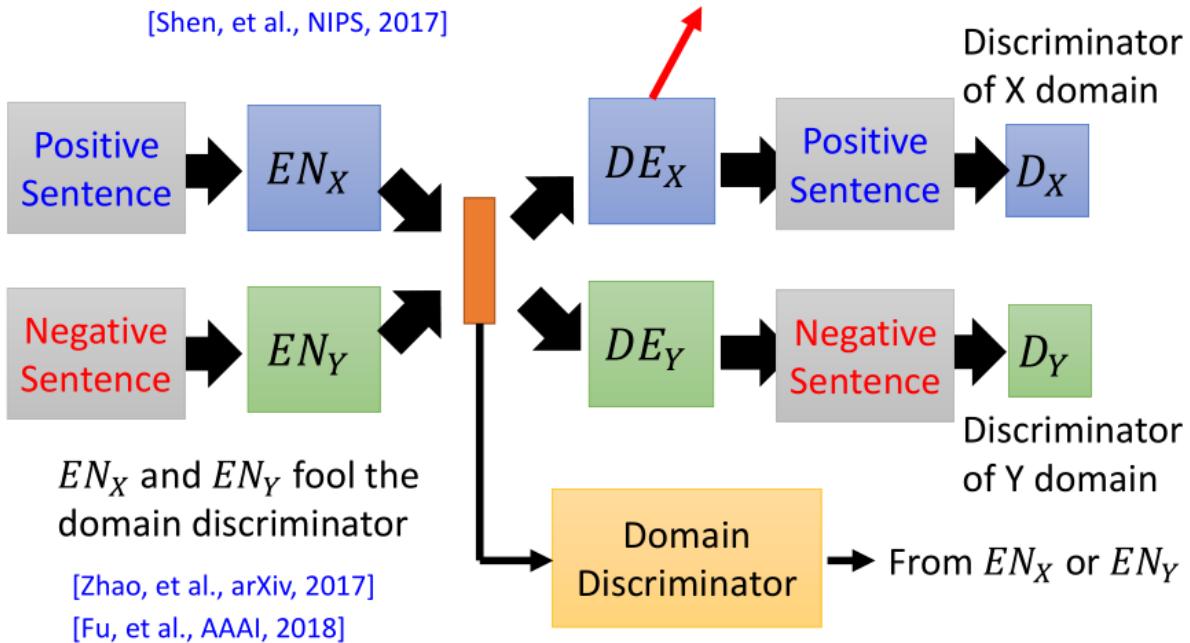
有很多不同的解法，我们刚才就说说有三个解法，一个是用 Gumbel-softmax，一个是给 discriminator continuous 的东西，第三个是用 RL，那就看你爱用哪一种。

在我们的实验里面，我们是用 continuous 的东西，怎么做呢？其实就是在每一个 word，用它的 word embedding 来取代，你把每一个 word，用它的 word embedding 来取代以后。每一个句子，就是一个 vector 的 sequence，那 word embedding 它并不是 one-hot，它是 continuous 的东西，现在你的 generator，是 output continuous 的东西，这个 discriminator 跟这个 generator，就可以吃这个 continuous 的东西，当作 input，所以你只要把 word 换成 word embedding，你就可以解这个 discrete 的问题。

那我们上次讲到说这种 unsupervised 的 transformation 有两个做法，一个就是 cycle GAN 系列的做法，那我们刚才看到哪个 Text style transfer，是用 cycle GAN 系列的做法。那也可以有另外一个系列的做法 Projection to Common Space，就是你把不同 domain 的东西，都 project 到同一个 space，然后再用不同 domain 的 decoder，把它解回来。

Decoder hidden layer as discriminator input

[Shen, et al., NIPS, 2017]



Text style transfer 也可以用这样子的做法。你唯一做的事情，就只是把本来你的 x domain 跟 y domain，可能是真人的头像，跟二次元人物的头像，把他们换成正面的句子，跟负面的句子。

当然我们有说，今天如果是产生文字的时候，你会遇到一些特别的问题就是因为，文字是 discrete 的，所以今天这个 discriminator 没有办法吃 discrete 的 input，如果它吃 discrete 的 input 的话，它会没有办法跟 decoder jointly trained，所以怎么解呢？

在文献上我们看过的一个作法是，当然你可以用 RL, Gumbel-softmax 等不同的解法，但我在文献上看到 MIT CSAIL lab 做的一个有趣的解法是，有人说这 discriminator 不要吃 decoder output 的 word，它吃 decoder 的 hidden state，就 decoder 也是一个 RNN 嘛，那 RNN 每一个 time step 就会有一个 hidden vector，这个 decoder 不吃最终的 output，它吃 hidden vector，hidden vector 是 continuous 的，所以就没有那个 discrete 的问题，这是一个解法。

然后我们说这个今天你要让这两个不同的 encoder，可以把不同 domain 的东西 project 到同一个 space，你需要下一些 constrain，我们讲了很多各式各样不同的 constrain，那我发现说那些各式各样不同的 constrain，还没有被 apply 到文字的领域，所以这是一个未来可以做的事情。

我现在看到唯一做的技术只有说有人 train 了一个 classifier，那这个 classifier，就吃这两个 encoder 的 output，那这两个 encoder 要尽量去骗过这个 classifier，这个 classifier 要从这个 vector，判断说这个 vector 是来自于哪一个 domain，我把文献放在这边给大家参考。

Unsupervised Abstractive Summarization

那接下来我要讲的是说，用 GAN 的技术来做，Unsupervised Abstractive summarization。

那怎么 train 一个 summarizer 呢？怎么 train 一个 network 它可以帮你做摘要呢？那所谓做摘要的意思是说，假设你收集到一些文章，那你有没有时间看，你就把那些文章直接丢给 network，希望它读完这篇文章以后，自动地帮你生成出摘要。

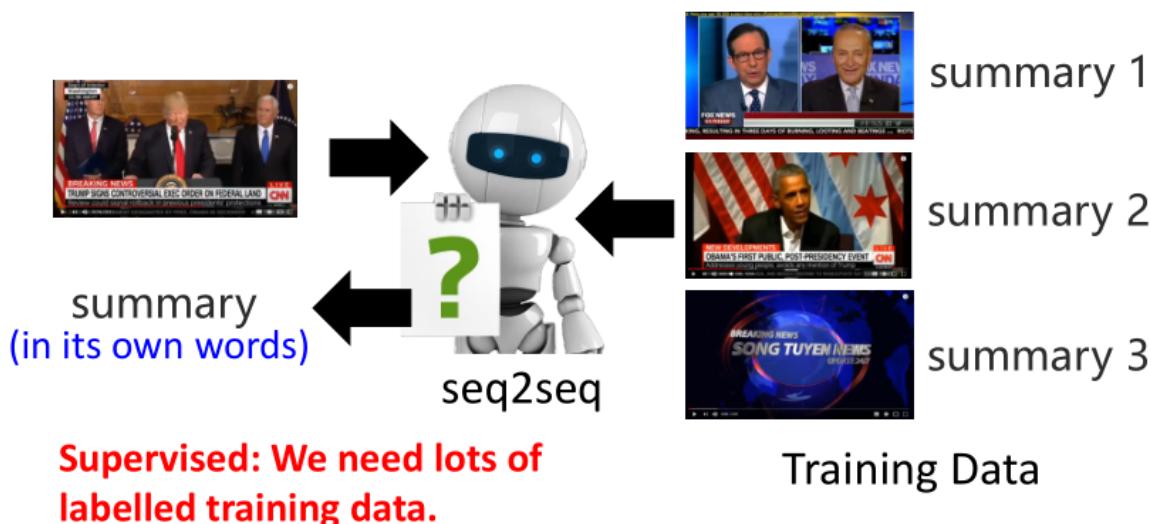
当然做摘要这件事，从来不是一个新的问题，因为这个显然是一个非常有应用价值的东西，所以他从来不是一个新的问题，五六十年前就开始有人在做了，只是在过去的时候，machine learning 的技术还没有那么强，所以过去你要让机器学习做摘要的时候，通常机器学做的事情是 extracted summarization，这边 title 写的是 abstractive summarization，还有另外一种作摘要的方法叫做 extracted summarization，extracted summarization 的意思就是说，给机器一篇文章，那每一篇文章机器做的事情就是判断这篇文章的这个句子，是重要的还是不重要的，接下来他把所有判断为重要的句子接起来，就变成一则摘要了。

那你可能会说用这样的方法，可以产生好的摘要吗？那这种方法虽然很简单，你就是 learn 一个 binary classifier，决定一个句子是重要的还是不重要的，但是你没有办法用这个方法，产生真的非常好的摘要。

为什么呢？你要用自己的话，来写摘要，你不能够把课文里面的句子就直接抄出来，当作摘要，你要自己 understanding 这个课文以后，看懂这个课文以后，用自己的话，来写出摘要。那过去 extracted summarization，做不到这件事，但是今天多数我们都可以做 abstractive summarization。

Abstractive Summarization

- Now machine can do **abstractive summary** by seq2seq (write summaries in its own words)

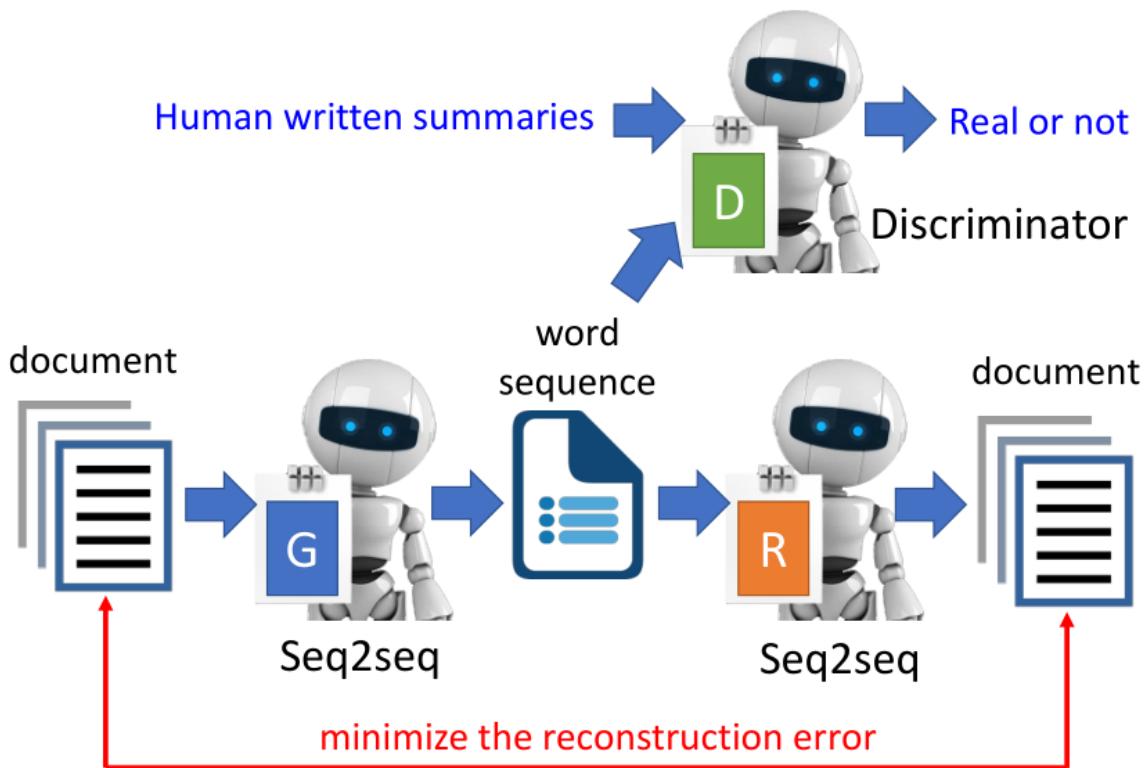


**Supervised: We need lots of
labelled training data.**

Training Data

怎么做？learn 一个 seq2seq model，收集一大堆的文章，每一篇文章都有人标的摘要，然后 seq2seq model 硬 train 下去，train 下去就结束了，给它一个新的文章，它就会产生一个摘要，而且这个摘要是机器用自己的话说出来的，不见得是文章里面现有的句子。但是这整套技术最大的问题就是，你要 train 这个 seq2seq model，你显然需要非常大量的数据。到底要多少数据才够呢？很多同学会想要自己 train 一个 summarizer，然后他去网络上收集比如说 10 万篇文章，10 万篇文章它通通有标注摘要，他觉得已经很多了，train 下去结果整个坏掉。为什么呢？你要 train 一个 abstractive summarization 系统，通常至少要一百万个 examples 才做得起来，没有一百万个 examples，机器可能连产生符合文法的句子都做不到。如果有上百万个 examples，对机器来说，要产生合文法的句子，其实不是一个问题。但是这个 abstractive summarization 最大的问题就是，要收集大量的资料，才有办法去训练。

Unsupervised Abstractive Summarization



所以怎么办呢？我们就想要提出一些新的方法，我们其实可以把文章视为是一种 domain，把摘要视为是另外一种 domain。现在如果我们有了 GAN 的技术，我们可以在两个 domain 间直接用 unsupervised 的方法互转，我们并不需要两个 domain 间的东西的 pair。所以今天假设我们把文章视为一个 domain，摘要视为另外一个 domain，我们不需要文章和摘要的 pair，只要收集一大堆文章，收集一大堆摘要当作范例告诉机器说，摘要到底长什么样子，这些摘要不需要是这些文章的摘要，只要收集两堆 data，机器就可以自动在两个 domain 间互转，你就可以自动地学会怎么做摘要这件事。而这个 process 是 unsupervised 的，你并不需要标注这些文章的摘要，你只需要提供机器一些摘要，作为范例就可以了。那这个技术怎么做的呢？

这个技术就跟 cycle GAN 是非常像的，我们 learn 一个 generator，这个 generator 是一个 seq2seq model。这个 seq2seq model 吃一篇文章，然后 output 一个比较短的 word sequence，但是假设只有这个 generator，你没办法 train，因为 generator 根本不知道说，output 什么样的 word sequence，才能当作 input 的文章的摘要。所以接下来，你就要 learn 一个 discriminator，这个 discriminator 的工作是什么呢？这个 discriminator 的工作就是，他看过很多人写的摘要，这些摘要不需要是这些文章的摘要，，他知道人写的摘要是什么样子，接下来他就可以给这个 generator feedback，让 generator output 出来呢 word sequence，看起来像是摘要一样。

就跟我们之前讲说什么风景画转梵高画一样，你需要一个 discriminator，看说一张图是不是梵高的图，把这个信息 feedback 给 generator，generator 就可以产生看起来像是梵高的画作。那这边其实一样，你只需要一个 generator，一个 discriminator，discriminator 给这个 generator feedback，就可以希望它 output 出来的句子，看起来像是 summary。

但是在讲 cycle GAN 的时候我们有讲过说，光是这样的架构是不够的。因为 generator 可能会学到产生看起来像是 summary 的句子，就人写的 summary 可能有某些特征，比如说它都是比较简短的，也许 generator 可以学到产生一个简短的句子，但是跟输入是完全没有关系的。那怎么解这个问题呢？就跟 cycle GAN 一样，你要加一个 reconstructor，在做 cycle GAN 的时候我们说，我们把 x domain 的东西转到 y domain，接下来要 learn 一个 generator，把 y domain 的东西转回来，这样我们就可以迫使，x domain 跟 y domain 的东西，是长得比较像的。我们希望 generator output，跟 input 是有关系的，所以在做 unsupervised abstractive summarization 的时候，我们这边用的概念，跟 cycle GAN 其实是一模一样的。你 learn 另外一个 generator，我们这边称为 reconstructor，他的工作是，吃第一个 generator output 的 word sequence，把这个 word sequence，转回原来的 document，那你在 train 的时候你就希

望，原来输入的文章被缩短以后要能被扩写回原来的 document。这个跟 cycle GAN 用的概念是一模一样。

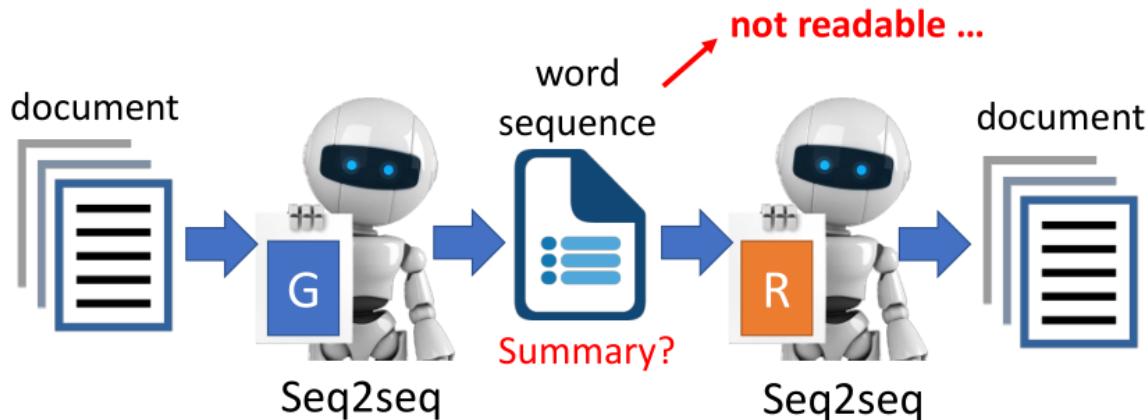
Unsupervised Abstractive Summarization

Only need a lot
of documents to
train the model



This is a seq2seq2seq auto-encoder.

Using a sequence of words as latent representation.



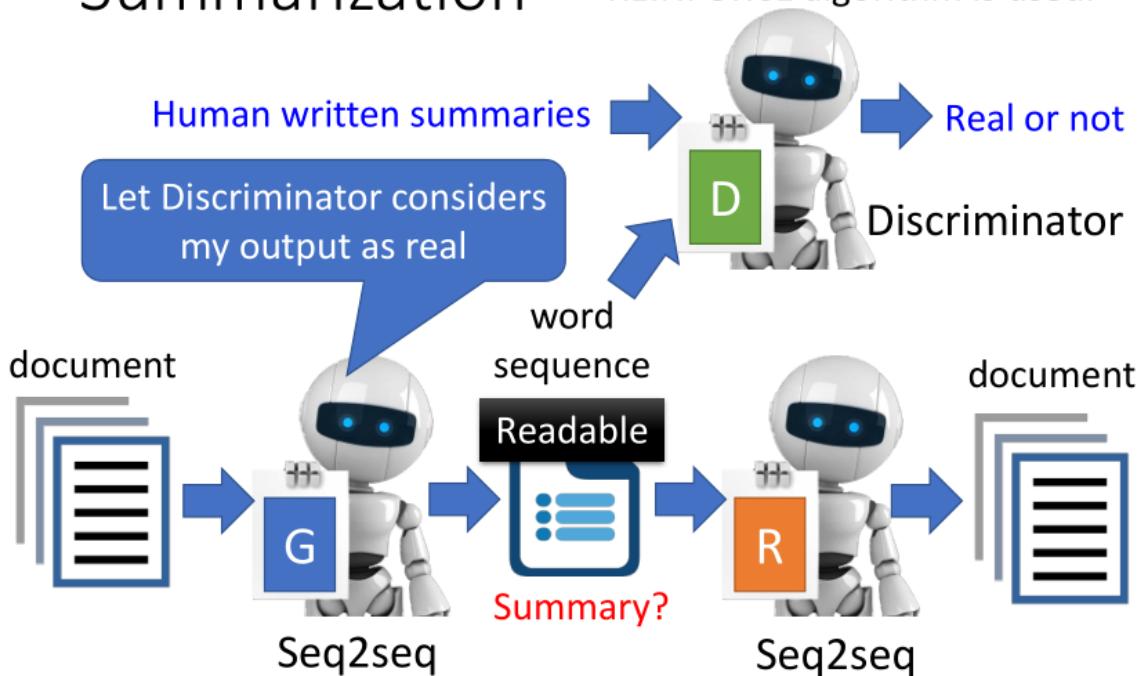
那你其实可以用另外一个方法来理解这个 model，你说我有一个 generator，这个 generator 把文章变成简短的句子，那你有另外一个 reconstructor 它把简短的句子变回原来的文章。如果这个 reconstructor 可以把简短的句子，变回原来的文章，代表说这个句子，有原来的文章里面重要的信息，因为这个句子有原来的文章里面重要的信息，所以你就可以把它当作一个摘要。在 training 的时候，这个 training 的 process 是 unsupervised，因为你只需要文章就好，你只需要输入和输出的文章越接近越好，所以并不需要给机器摘要，你只需要提供给机器文章就好。那这个整个 model，这个 generator 跟 reconstructor 合起来，可以看作是一个 seq2seq2seq auto-encoder，你就一般你 train auto-encoder 就 input 一个东西，把它变成一个 vector，把这个 vector 变回原来的 object，比如说是个 image 等等，那现在是 input 一个 sequence，把它变成一个短的 sequence，再把它解回原来长的 sequence，这样是一个 seq2seq2seq auto-encoder。

那一般的 auto-encoder 都是用一个 latent vector 来表示你的信息，那我们现在不是用一个人看不懂的 vector 来表示信息，我们是用一个句子来表示信息，这个东西希望是人可以读的。

但是这边会遇到的问题是，假设你只 train 这个 generator 跟这个 reconstructor，你产生出来的 word sequence 可能是人没有办法读的，他可能是人根本就没办法看懂的，因为机器可能会自己发明奇怪的暗语，因为 generator 跟 reconstructor，他们都是 machine，所以他们可以发明奇怪的暗语，反正只要他们彼此之间看得懂就好，那人看不懂没有关系，比如说台湾大学，它可能就缩写成湾学，而不是台大，反正只要 reconstructor 可以把湾学解回台湾大学其实就结束了。

Unsupervised Abstractive Summarization

REINFORCE algorithm is used.



所以为了希望 generator 产生出来的句子是人看得懂的，所以我们要加一个 discriminator，这个 discriminator 就可以强迫说，generator 产生的句子，一方面要是一个 summary 可以对 reconstructor 解回原来的文章，同时 generator output 的这个句子，也要是 discriminator 可以看得懂的，觉得像是人类写的 summary。那这个就是 unsupervised abstractive summarization 的架构。

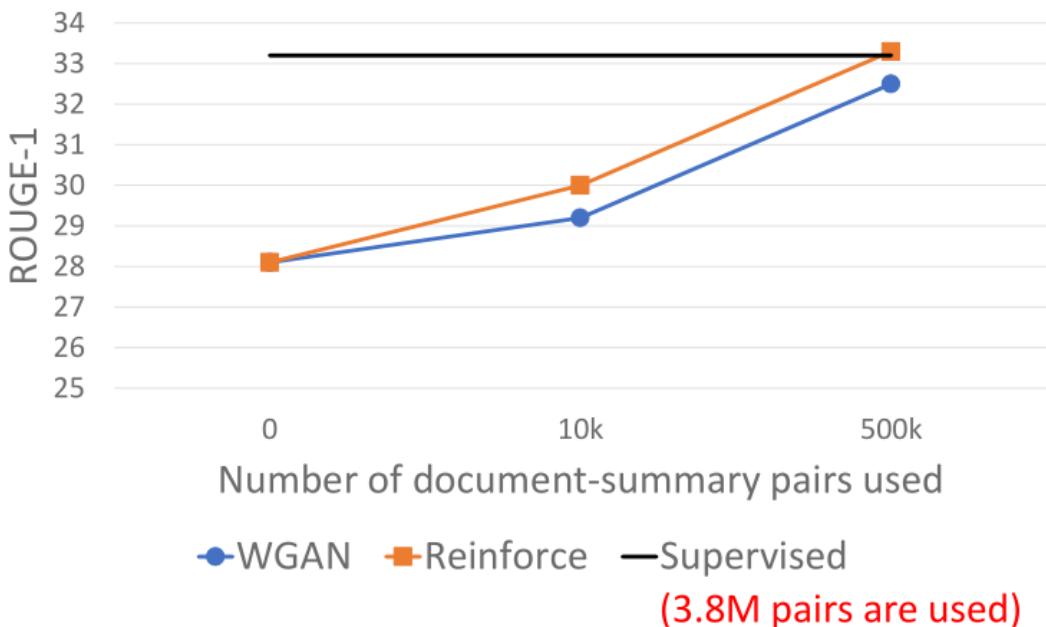
这边可以跟大家讲一下就是说，在 training 的时候，因为这边 output 是 discrete 的嘛，所以你当然是需要有一些方法来处理这种 discrete output，那我们用的就是 reinforced algorithm。

那有人可能会想说用 unsupervised learning 有什么好处，因为你用 unsupervised learning，永远赢不过 supervised learning，supervised learning 就是，unsupervised learning 的 upper bound，unsupervised learning 的意义何在。

那所以我们用这个实验来说明一下 unsupervised learning 的意义。

Semi-supervised Learning

Using
matched data



那这边这个纵轴是 ROUGE 的分数，总之就是用来衡量摘要的一个方法，值越大，代表我们产生的摘要越好。黑色的线是 supervised learning 的方法，今天在做 supervised learning 的时候，需要 380 万笔 training example，380 万篇文章跟它的摘要，你才能够 train 出一个好的 summarization 的系统，是黑色的这一条线。那这边我们用了不同的方法来做这个，来 train 这个 GAN，我们有用 WGAN 的方法，有用 reinforcement learning 的方法，分别是蓝线跟橙线。得到的结果其实是差不多的，WGAN 差一点，用 reinforcement learning 的结果是比较好的，那今天如果在完全没有 label 情况下，得到的结果是这个样子。那当然跟 supervised 的方法，还是差了一截。

但是今天你可以用少量的 summary，再去 fine tune unsupervised learning 的 model，就是你先用 unsupervised learning 的方法把你的 model 练得很强，再用少量的 label data 去 fine tune，那它的进步就会很快。

举例来说，我们这边只用 50 万笔的 data，得到的结果就已经跟 supervised learning 的结果一样了，所以这边你只需要原来的 1/6 或者更少的 data，其实就可以跟用全部的 data 得到一样好的结果。

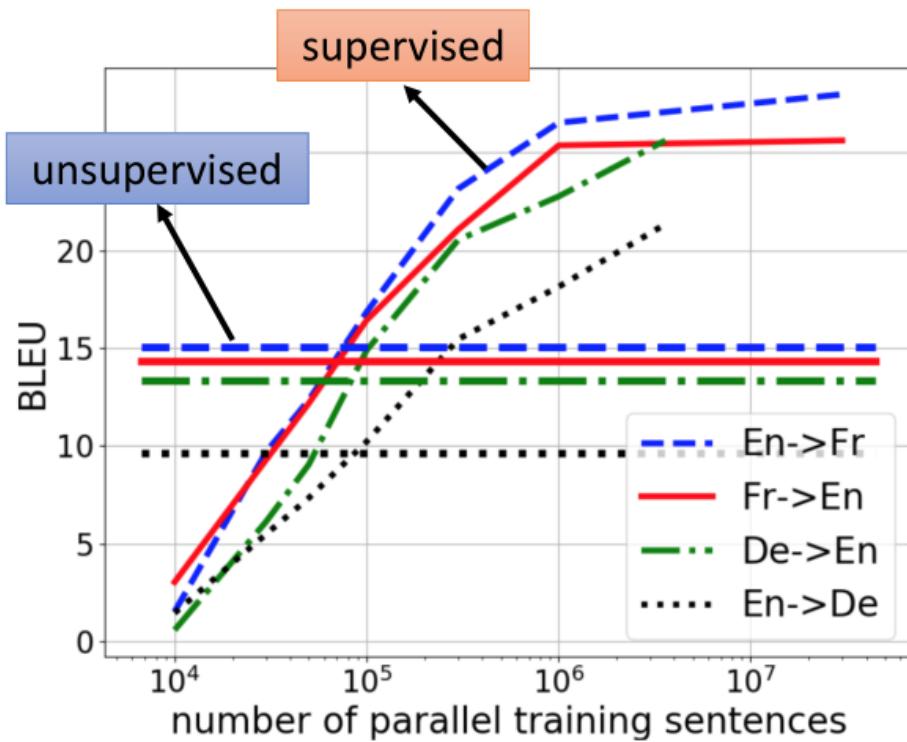
所以 unsupervised learning 带给我们的好处就是，你只需要比较少的 label data，就可以跟过去大量 label data 的时候得到的结果也许是一样好的。那这就是 unsupervised learning 的妙用。

Unsupervised Machine Translation

这边举最后一个例子是 unsupervised machine translation，我们今天可以把不同的语言视为是不同的 domain，就假设你要英文转法文，你就要把英文视为一个 domain，法文视为另外一个 domain，然后就可以用 unsupervised learning 的方法把英文转成法文，法文转成英文，做到翻译就结束了，所以你就去做 unsupervised 的翻译。

那这个方法听起来还蛮匪夷所思的，真的能够做得到吗？其实 facebook 在 ICLR2018 就发了两篇这种 paper，看起来还真的是可以的。

细节我们就不讲，细节你可以想象就很像那个 cycle GAN 这样，只是前面我们有说拿两种不同 image 当作两个不同的 domain，两种不同的语音当作两个不同的 domain，现在只是把两种语言当作两个不同的 domain，然后让机器去学两种语言间的对应，硬做看看做不做的起来。



Unsupervised learning with 10M sentences = **Supervised learning with 100K sentence pairs**

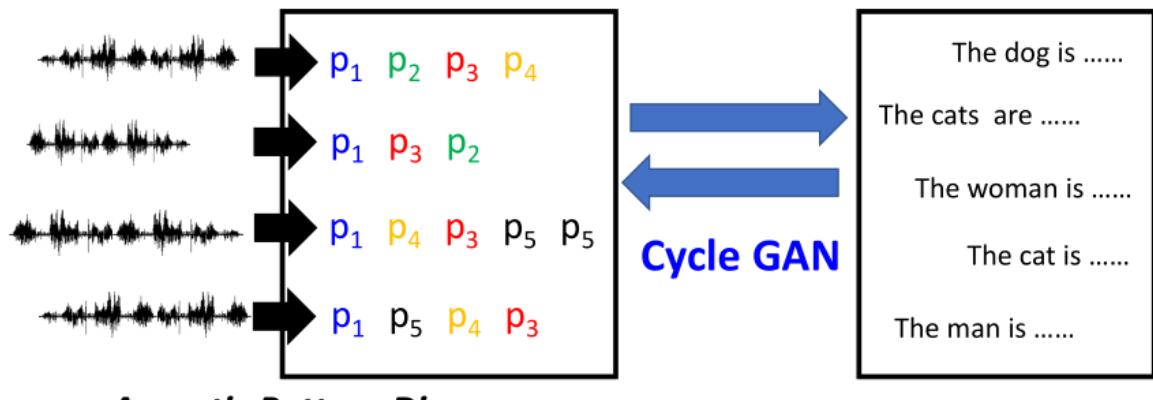
这个是文献上的结果，这个虚线代表 supervised learning 的方法，纵轴是 BLEU score，是拿来衡量摘要好坏的方法，BLEU 越高，代表摘要做得越好，横轴是训练资料的量，从 10^4 一直到 10^7 。

如果 supervised learning 的方法，这边是不同语言的翻译，英文转法文，法文转英文，德文转英文，英文转德文，四条线代表四种不同语言的 pair，语言组合间的翻译。

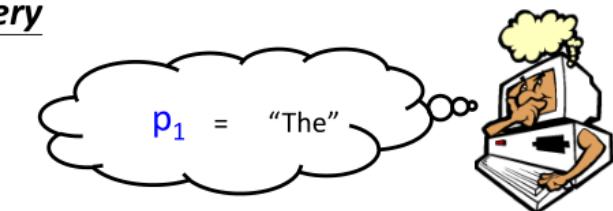
那你发现训练资料越多，当然结果就越好，这个没有什么特别稀奇的，横线是这个横线是什么？横线用 10^7 的 data 去 train 的 unsupervised learning 的方法，但是你并不需要两个语言间的 pair。做 supervised learning 的时候，你需要两个语言间的 pair。但做 unsupervised learning 的时候，就是两堆句子，不需要他们之间的 pair，得到的结果，只要 unsupervised learning 的方法有 10 million 的 sentences，你的 performance 就可以跟 supervised learning 的方法，只用 10 万笔 data，是一样的好的。

所以假设你手上没有 10 万笔 data pair，unsupervised 方法其实还可以赢过 supervised learning 的方法，这个结果是我觉得还颇惊人的。

Unsupervised Speech Recognition



**Can we achieve
unsupervised speech
recognition?**



[Liu, et al., arXiv, 2018] [Chen, et al., arXiv, 2018]

既然两种不同的语言可以做，那语音跟文字间可不可以做呢？把语音视为是一个 domain，把文字视为是另外一个 domain，然后你就可以 apply 类似 GAN 的技术，在这两个 domain 间，互转，这样看看机器能不能够学得起来。如果假设今天机器可以学会说，给它一堆语音给它一堆文字，它就可以自动学会怎么把声音转成文字的话，你就可以做 unsupervised 的语音识别了。未来机器可能在日常生活中，听人讲话，然后它自己再去网络上，看一下人写的文章，就自动学会，语音识别了。

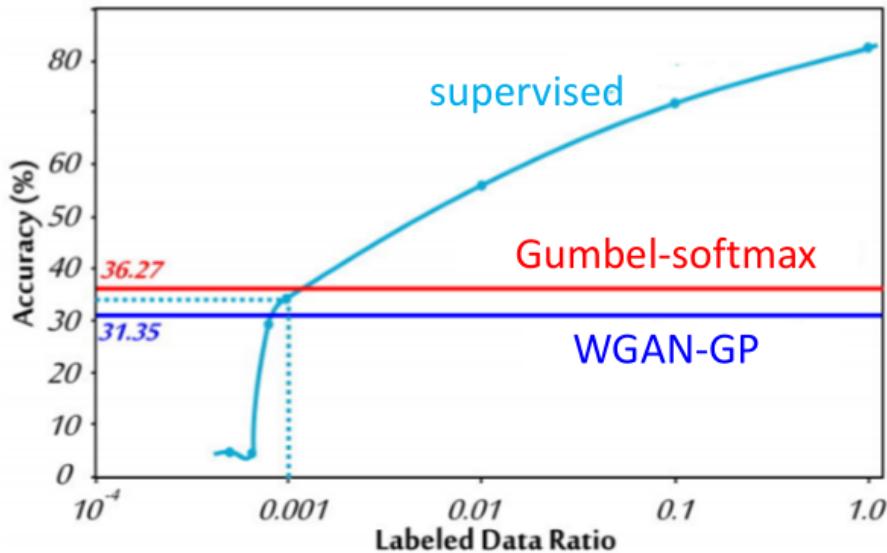
有人可能会想说，这个听起来也是还蛮匪夷所思的，这个东西到底能不能够做到呢？我觉得是有可能的，如果翻译可以做到，这件事情也是有机会的，unsupervised 语音识别也是有机会的。

这边举一个非常简单的例子，假如说所有声音讯号的开头，都是某个样子，比如说都有 P1 这个 pattern，我们用 P 代表一个 pattern，就 P1 这个 pattern，那机器在自己去读文章以后发现说，所有的文章都是 The 开头，它就可以自动 mapping 到说 P1 这种声音讯号，这种声音讯号的 pattern，就是 The 这样，那这是一个过度简化的例子。

Unsupervised Speech Recognition

- Phoneme recognition

Audio: TIMIT
Text: WMT



实际上做不做得起来呢？这个是实际上得到的结果，我们用的声音讯号来自于 TIMIT 这个 corpus，用的文字来自于 WMT 这个 corpus。

那这两个 corpus 是没有对应关系的，一堆语音讲自己的，文字讲自己的，两堆不相关的东西，用类似 cycle GAN 的技术，看能不能够把声音讯号硬是转成文字。

这是一个实验的结果，纵轴是辨识的正确率，那其实是 Phoneme recognition，不是辨识出文字，你是辨识出音标而已，辨识出文字还是比较难，直接辨识出音标而已。

那这个横轴代表说训练资料的量，如果是 supervised learning 的方法，当然训练数据的量越多，performance 越好。这两个横线就是用 unsupervised 的方法硬做得到的结果，那硬做其实有得到 36% 的正确率，你会想 36% 的正确率，这么低，这个 output 结果应该人看不懂吧，是的人看不懂，但是它是远比 random 好的，所以就算是在完全 unsupervised 的情况下，只给机器一堆文字，一堆语音，它还是有学到东西的。

Concluding Remarks

Conditional Sequence Generation

- RL (human feedback)
- GAN (discriminator feedback)

Unsupervised Conditional Sequence Generation

- Text Style Transfer
- Unsupervised Abstractive Summarization
- Unsupervised Translation

GAN Evaluation

这个投影片就是 GAN 的最后要跟大家讲的东西，就是怎么做 Evaluation。

Evaluation 是要做什么？我们要讲的是，怎么 evaluate 用 GAN 产生的 object 的好坏。怎么知道你的 image 是好还是不好。我觉得最准的方法就是人来看，但是在人来看往往不是很客观，如果你在看文献上的话，很多 paper 只是秀几张它产生的图，然后加一个 comment 说你看到我今天产生的图，我觉得这应该是我在文献上看过最清楚的图，然后就结束了，你也不知道是真的还是假的。

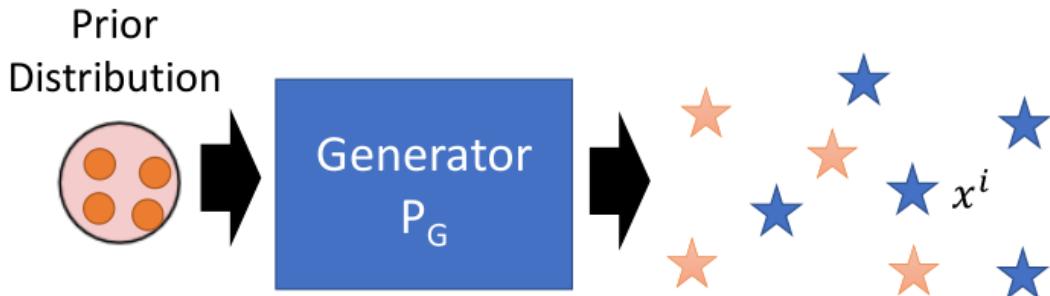
今天要探讨的就是有没有哪一些比较客观的方法，来衡量产生出来的 object 到底是好还是不好。

Likelihood

在传统上怎么衡量一个 generator？传统衡量 generator 的方法是算 generator 产生 data 的 likelihood，也就是说 learn 了一个 generator 以后，接下来给这 generator 一些 real data，假设做 image 生成，已经有一个 image 的生成的generator，接下来拿一堆 image 出来，这些 image 是在 train generator 的时候 generator 没有看过的 image，然后去计算 generator 产生这些 image 的机率，这个东西叫做 likelihood。

Likelihood

- ★ : real data (not observed during training)
- ★ : generated data



$$\text{Log Likelihood: } L = \frac{1}{N} \sum_i \log P_G(x^i)$$

We cannot compute $P_G(x^i)$. We can only sample from P_G .

其实是你的 testing data 的 image 的 likelihood 通通算出来做平均，就得到一个 likelihood，这个 likelihood 就代表了 generator 的好坏，因为假设 generator 它有很高的机率产生这些 real data，就代表这个 generator 可能是一个比较好的 generator。

但是如果是 GAN 的话，假设你的 generator 是一个 network 用 GAN train 出来的话，会遇到一个问题就是没有办法计算 $P_G(x^i)$ ，为什么？

因为 train 完一个 generator 以后，它是一个 network，这个 network 你可以丢一些 vector 进去，让它产生一些 data，但是你无法算出它产生某一笔特定 data 的机率。

它可以产生东西，但你说指定你要产生这张图片的时候，它根本不可能产生你指定出来的图片，所以根本算不出它产生某一张指定图片的机率是多少。所以如果是一个 network 所构成的 generator，要算它的 likelihood 是有困难。

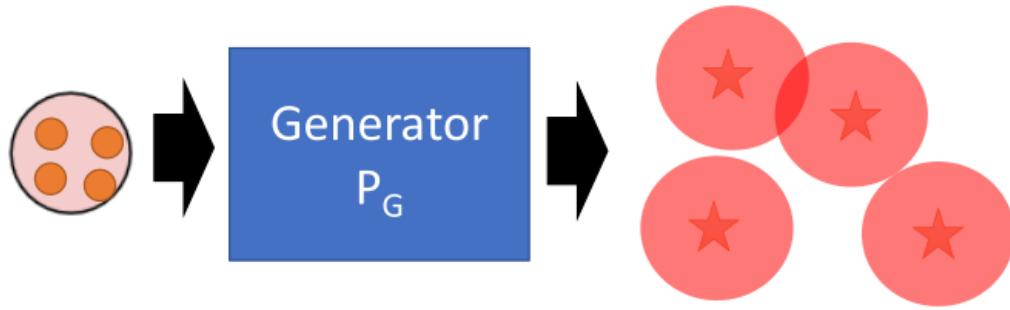
假设这个 generator 不是 network 所构成的，举例来说这个 generator 就是一个 Gaussian Distribution，或是这个 generator 是一个 Gaussian Mixture Model，给它一个 x ，Gaussian Mixture Model 可以推出它产生这个 x 的机率，但是因为那是 Gaussian Mixture Model，它是个比较简单的 model。如果 generator 不是一个简单的 model，是一个复杂的 network，你求不出它产生某一笔 data 的机。

但是我们又不希望 generator 就只是 Gaussian Mixture Model，我们希望我们的 generator 是一个比较复杂的模型。所以遇到的困难就是如果是一个复杂的模型，我们就不知道怎么去计算 likelihood，不知道怎么计算这个复杂的模型，产生某一笔 data 的机率。

Kernel Density Estimation

怎么办？在文献上一个 solution 叫做，Kernel Density Estimation。

Estimate the distribution of $P_G(x)$ from sampling



Each sample is the mean of a Gaussian with the same covariance.

Now we have an approximation of P_G , so we can compute $P_G(x^i)$ for each real data x^i
Then we can compute the likelihood.

也就是把你的 generator 拿出来，让你的 generator 产生很多很多的 data，接下来再用一个 Gaussian Distribution 去逼近你产生的 data。什么意思？

假设有一个 generator 你让它产生一大堆的 vector 出来，假设做 Image Generation 的话，产生出来的 image 就是 high dimensional 的 vector，你用你的 generator 产生一堆 vector 出来，接下来把这些 vector 当作 Gaussian Mixture Model 的 mean，然后每一个 mean 它有一个固定的 variance，然后再把这些 Gaussian 通通都叠在一起，就得到了一个 Gaussian Mixture Model。有了这个 Gaussian Mixture Model 以后，你就可以去计算这个 Gaussian Mixture Model 产生那些 real data 的机率，就可以估测出这个 generator 它产生出那些 real data 的 likelihood 是多少。

我们现在要做的事情是，我们先让 generator 先生一大堆的 data，然后再用 Gaussian 去 fit generator 的 output，到底要几个 Gaussian？32 个吗？64 个吗？还是一个点一个？问题是我不知道，所以这就是一个难题。

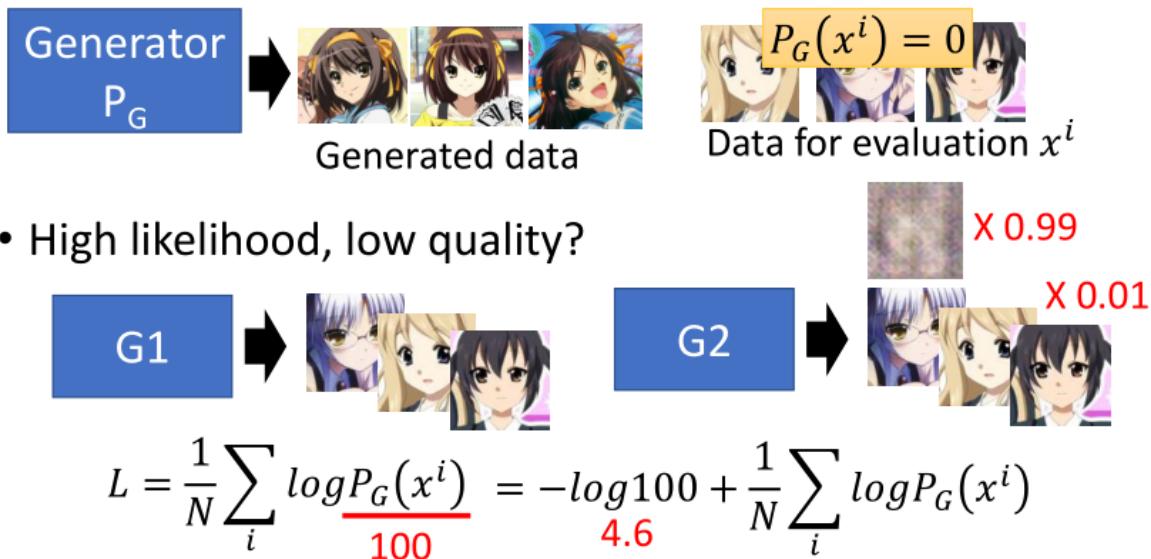
而且另外一个难题是你不知道 generator 应该要 sample 多少的点，才估的准它的 distribution，要 sample 600 个点还是 60,000 个点你不知道。所以这招在实作上也是有问题的。在文献上你会看到有人做这招，就会出现一些怪怪的结果，举例来说，你可能会发现你的 model 算出来的 likelihood 比 real data 还要大。总之这个方法也是怪怪的，因为里面问题太多了，你不知道要 sample 几个点，然后你不知道要怎么估测 Gaussian 的 Mixture，有太多的问题在里面了。

Likelihood v.s. Quality

还有接下来还有更糟的问题，我们就算退一步讲说你真的想出了一个方法，可以计算 likelihood，likelihood 本身也未必代表 generator 的 quality。

- Low likelihood, high quality?

Considering a model generating good images (small variance)



为什么这么说？因为有可能第一个 likelihood 确有高的 quality，举例来说有一个 generator，它很厉害，它产生出来的图都非常的清晰。

所谓 likelihood 的意思是计算这个 generator，产生某张图片的机率，也许这个 generator 虽然它产生的图很清晰，但它产生出来都是凉宫春日的头像而已。如果是其它人物的头像，它从来不会生成，但是 testing data 就是其它人物的头像。所以如果是用 likelihood 的话，likelihood 很小，因为它从来不会产生这些图，所以 likelihood 很小。但是又不能说它做得不好，它其实做得很好，它产生的图是 high quality 的，只是算出来 likelihood 很小。所以 likelihood 并不代表 quality，它们俩者是不等价。

反过来说，高的 likelihood 也并不代表你产生的图就一定很好，你有一个 model 它 likelihood 很高，它仍然有可能产生的图很糟，怎么说？

这边举一个例子，里面有一个 generator 1，generator 1 很厉害，它的 likelihood 很大，假设我们不知道怎么回事，somehow 想了一个方法可以估测 likelihood，虽然之前我们在前期的投影片已经告诉你，估测 likelihood 也是很麻烦，不知道怎么做，现在 somehow 想了一个方法可以估测 likelihood。现在有个很强的 generator，它的 likelihood 是大 L，它产生这些图片的机率很高，现在有另外一个 generator，generator 2 它有 99% 的机率产生 random noise，它有 1% 的机率，它做的事情跟 generator 1 一样。如果我们今天计算 generator 2 的 likelihood，generator 2 它产生每一张图片的机率是 generator 1 的 1/100。假设 generator 1 产生某张图片 x_i 的机率是 $P_G(x_i)$ ，generator 2 产生那张图片的机率，就是 $P_G(x_i) * 100$ ，因为 generator 2 有两个 mode，它有 99% 的机率会整个坏掉，但它有 1% 的机率会跟 generator 1 一样，所以 generator 1 产生某张图片的机率如果是 P_G ，那 generator 产生某张图片的机率就是 $P_G / 100$ 。

现在问题来了，假设把这个 likelihood 每一项都除以 100，你会发现你算出来的值，也差不了多少，因为除一百这项把它拿出来，就是 $-\log(100)$ ，才减 4.65 而已，如果看文献 likelihood 算出来都几百，差了 4 你可能会觉得没什么差别。

但是如果看实际上的 generator 2 跟 generator 1 比的话，generator 1 你会觉得它应该是比 generator 2 好一百倍的，只是你看不出来而已，数字上看不出来。

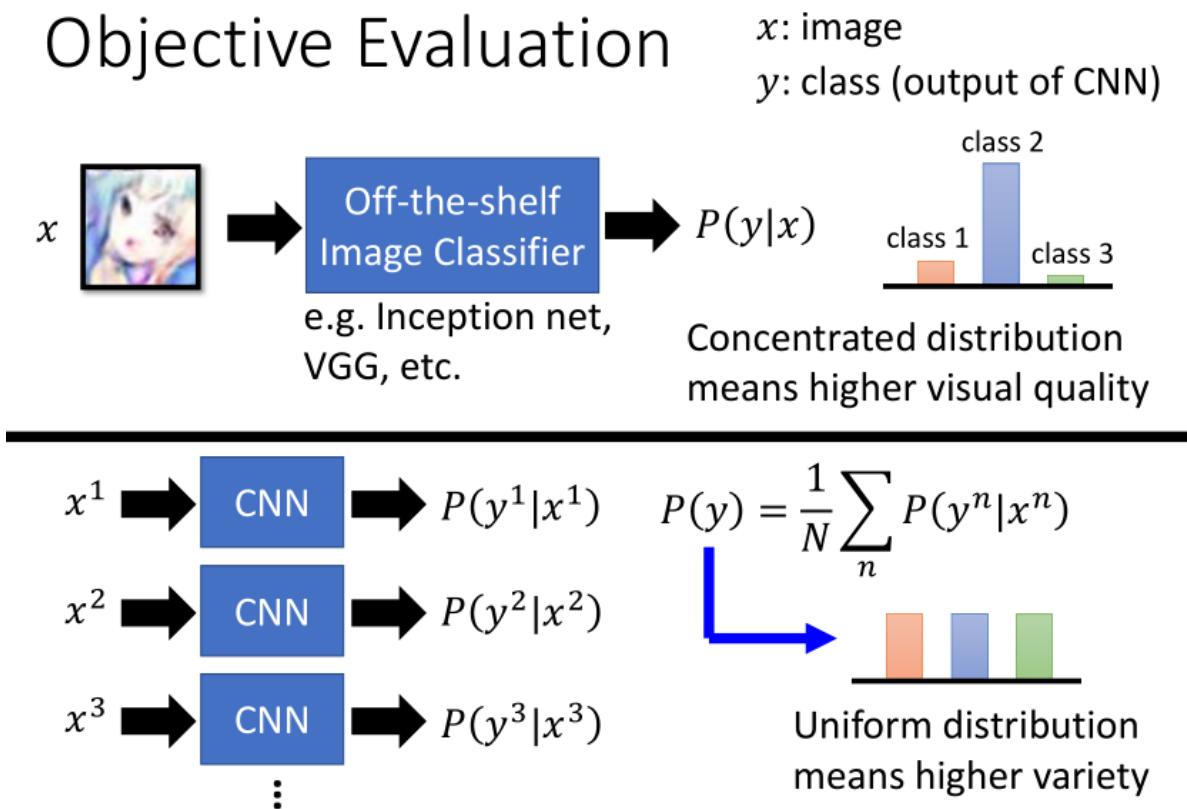
所以 likelihood 跟 generator 真正的能力其实也未必是有关系的。

Objective Evaluation

今天这个文献上你常常看到一种 Evaluation 的方法，常常看到一种客观的 Evaluation 的方法，是拿一个已经 train 好的 classifier 来评价现在，产生出来的 object。

[Tim Salimans, et al., NIPS, 2016]

Objective Evaluation



假设要产生的 object 是影像的话，你就拿一个影像的 classifier 来判断这个 object 的好坏，就好像我们是拿一个人脸的辨识系统，来看你产生的图片，这个人脸辨识系统能不能够辨识的出来，如果可以就代表你产生出来的是还可以的，如果不可以就代表你产生出来的真的很弱。

今天这个道理是一样的，假设你要分辨机器产生出来的一张影像好还是不好，你就拿一个 Image Classifier 出来，这 Image Classifier 通常是已经事先 train 好的，举例来说它是个 VGG，它是个 Inception Net。

把这个 Image Classifier 丢一张机器产生的 image 给它，它会产生一个 class 的 distribution，它给每一个 class 一个机率，如果产生的机率越集中，代表产生的图片的质量越高。因为这个 classifier 它可以轻易的判断，现在这个图片它是什么样的东西。所以它给某一个 class 机率特别高，代表产生的图片是这个 model 看得懂的。

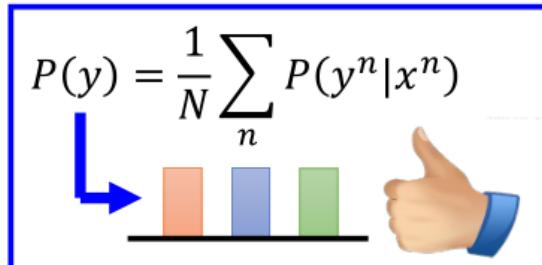
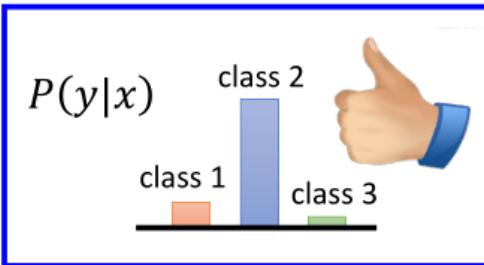
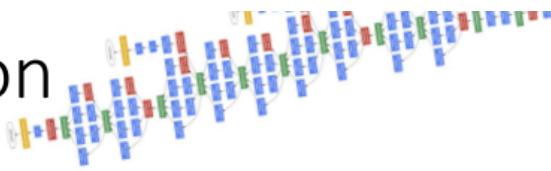
但这个只是一个衡量的方向而已，你同时还要衡量另外一件事情，因为我们知道在 train GAN 会遇到一个问题就是，**Mode collapse** 的问题，你的机器可能可以产生某张很清晰的图，但它就只能产生那张图而已，这个不是我们要的。

所以在 evaluate GAN 的时候还要从另外一个方向，还要从 diverse 的方向去衡量它，什么叫从 diverse 的方向去衡量它呢？你让你的机器产生一把，这边举例就产生三张，把这三张图通通丢到 CNN 里面，让它产生三个 distribution，接下来把这三个 distribution 平均起来，如果平均后的 distribution 很 uniform 的话，这个 distribution 平均完以后，它仍然很平均的话，那就意味着每一种不同的 class 都有被产生到，代表产生的 output 是比较 diverse。如果平均完发现某一个 class 分数特别高，就代表它的 output，你的 model 倾向于产生某个 class 的东西，就代表它产生的 output 不够 diverse。

所以我们可以从两个不同的方向，用某一个事先 train 好的 Image Classifier 来衡量 image，可以只给它一张图，然后看产生的图清不清楚，接下来给它一把图，看看是不是各种不同的 class，都有产生到。

Inception Score

Objective Evaluation



Inception Score

$$\begin{aligned} &= \sum_x \sum_y P(y|x) \log P(y|x) && \text{Negative entropy of } P(y|x) \\ &\quad - \sum_y P(y) \log P(y) && \text{Entropy of } P(y) \end{aligned}$$

有了这些原则以后，就可以定出一个 Score，现在一个常用的 Score，叫做 Inception Score。

那至于为什么叫做 Inception Score，当然是因为它用 Inception Net 去 evaluate，所以叫做 Inception Score。

我们之前有讲怎样的 generator 叫做好，好的 generator 它产生的单一的图片，丢到 Inception Net 里面，某一个 class 的分数越大越好，它是非常的 sharp。把所有的 output 都掉到 classifier 里面，产生一堆 distribution，把所有 distribution 做平均，它是越平滑越好。

根据这两者就定一个 Inception Score，把这两件事考虑进去，在 Inception Score 里面第一项要考虑的是，summation over 所有产生出来的 x ，每一个 x 丢到 classifier 去算它的 distribution，然后就计算 Negative entropy。Negative entropy 就是拿来衡量这个 distribution 够不够 sharp，每一张 image 它 output 的 distribution 越 sharp 的话，就代表产生的图越好。同时要衡量另外一项，另外一项就是把所有的 distribution 平均起来，如果平均的结果它的 entropy 越大也代表越好。同时衡量这两项，把这两项加起来，就是 Inception Score。

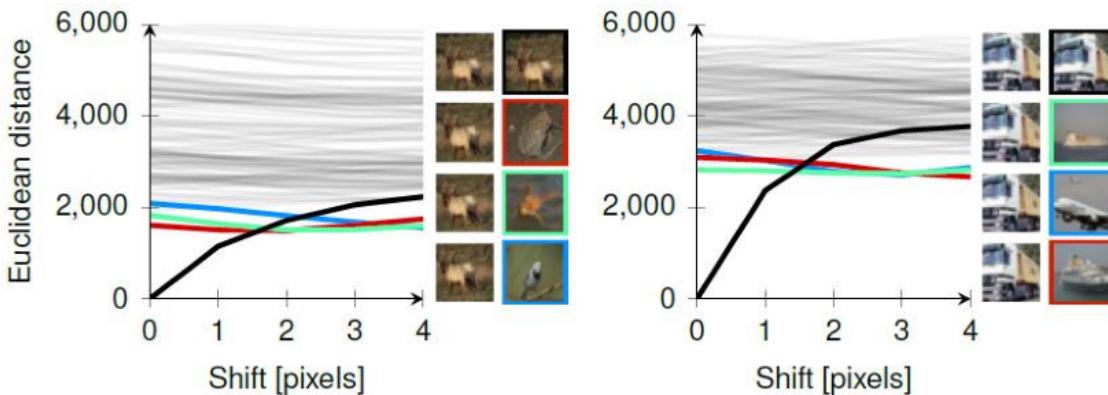
其实还有其它衡量的方法，但一个客观的方法就是拿一个现成的 model 来衡量你的 generator。

We don't want memory GAN.

还有另外一个 train GAN 要注意的问题，有时候就算 train 出来的结果非常的清晰，也并不代表你的结果是好的，为什么？因为有可能 generator 只是硬记了 training data 里面的，某几张 image 而已。这不是我们要的，因为假设 generator 要硬记 image 的话，那直接从 database sample 一张图不是更好吗？干嘛还要 train 一个 generator。

所以我们希望 generator 它是有创造力的，它产生出来的东西不要是 database 里面本来就已经现成的东西。但是怎么知道现在 GAN 产生出来的东西，是不是 database 已经现存的东西呢？这是另外一个 issue，因为没有办法把 database 里面每张图片都一个一个去看过。database 里面图片有上万张，根本没办法一张一张看过，所以根本不知道 generator 产生出来的东西是不是 database 里面的。

- Using k-nearest neighbor to check whether the generator generates new objects



GAN 产生一张图片的时候，就把这张图片拿去跟 database 里面每张图片都算 L1 或 L2 的相似度，但光算 L1 或 L2 的相似度是不够的，为什么？以下是文献上举的一个例子，这个例子是想要告诉大家，光算相似度，尤其是只算那种 pixel level 的相似度，是非常不够的。为什么这么说？这个例子是这样，假设有只羊的图，这个羊的图跟谁最像，当然是跟自己最像，跟 database 里面一模一样的那张图最像。黑色这条线代表的是羊这张图片，羊这张图片跟自己的距离当然是 0，跟其它图片的距离是比较大的，这边每一条横线就代表一张图片。把羊那张图的 pixel 都往左边移一格，还是跟自己最像。但是如果往左边移两格，会发现最像的图片，就变成红色这张，移三格就变绿色这张，移四格就变蓝色的这张。假设 generator 学到怪怪的东西就是，把所有的 pixel 都往左移两格，这个时候就算它 copy 了 database 你也看不出来。因为检测的方法检测不出这个 case。右边也是一样，把卡车的图片往左移一个 pixel 跟自己最像，移三个 pixel 就变跟飞机最像，移四个 pixel 就变跟船最像。

因为很难算两张图片的相似度，所以 GAN 产生一个图片的时候，你很难知道它是不是 copy 了 database 里面的 specific 的某一张图片，这个也都是尚待解决的问题。

所以有时候 GAN 产生出来结果很好，也不用太得意，因为它搞不好只是 copy 某一张图片而已。

Mode Dropping

Mode Collapse 的意思是说你的 real data 的 distribution 是比较大的，但是你 generate 出来的 example，它的 distribution 非常的小。

Mode dropping 意思是说你的 distribution 其实有很多个 mode，假设你 real distribution 是两群，但是你的 generator 只会产生同一群而已，他没有办法产生两群不同的 data。

假设 GAN 产生出来的是人脸的话，它产生人脸的多样性不够。

怎么检测它产生出来的东西它的多样性够不够，假设 train 了一个 DCGAN，DCGAN 是 Deep Convolutional GAN 的缩写，它的 training 的方法跟 Ian Goodfellow 一开始提出来的办法是一样的，只是在 DCGAN 里面那个作者爆搜了各种不同的参数，然后告诉你怎么样 train GAN 的时候结果会比较好，有不同 network 的架构，不同的 Activation Function，有没有加 batch，各种方法都爆搜一遍，然后告诉你怎么样做比较好。

怎么知道 DCGAN，train 一个产生人脸的 DCGAN，它产生的人脸的多样性是够的呢？一个检测方法是从 DCGAN 里面 sample 一堆 image，叫 DCGAN 产生一堆 image，然后确认产生出来的 image 里面有没有非常像的，有没有人会觉得是同一个人。

怎么知道是不是同一个人，结果来自于 ICLR 2018 叫 "Do GANs learn the distribution?"，里面的做法是让机器产生一堆的图片，接下来先用 classifier 决定有没有两张图片看起来很像，再把长的很像的图片拿给人看，问人说：你觉得这两个是不是同一个人，如果是，就代表 DCGAN 产生重复的图了，虽然产生图片每张都略有不同，但人可以看出这个看起来算不算是同一个人。

一些被人判断是感觉是同一个人的图片，DCGAN 会产生很像的图片，把这个图片拿去 database，里面找一张最像的图，会发现最像的图跟这个图没有完全一样，代表 DCGAN 没有真的硬背 training data 里面的图。不知道为什么它会产生很像的图，但这个图并不是从 database 里面背出来的。

它要衡量 DCGAN 到底可以产生多少不一样的 image，它发现如果 sample 四百张 image 的时候，有大于 50% 的机率，可以从四百张 image 里面，找到两张人觉得是一样的人脸，借由这个机率就可以反推到底整个 database 里面，整个 DCGAN 可以产生的人脸里面，有多少不同的人脸。

详细反推的细节，你再 check 一下 paper，有一个 database 有面有 M 张 image，M 到底应该多大才会让你 sample 四百张 image 的时候，有大于 50% 的机率 sample 到重复的。总之反推出 DCGAN 它可以产生各种不同的 image，其实只有 0.16 个 million 而已，只有十六万张图而已。

有另外一个做法叫 ALI，它比较强，反推出可以产生一百万张各种不同的人脸。ALI 看起来可以产生的脸多样性是比较高的。

但是不论是哪些方法都觉得它们产生的人脸的多样性，跟真实的人脸比起来，还是有一定程度的差距。感觉 GAN 没办法真的产生人脸的 distribution，这些都是尚待研究的问题。

GAN 的一个 issue 就是它产生出来的，distribution 不够大，它产生出来的 distribution 太 narrow，有一些 solution，比如说有一个方法，现在比较少人用，因为它 implement 起来很复杂，运算量很大，叫做 Unroll GAN。

Mini-batch Discrimination

有另外一个方法叫做 Mini-batch Discrimination，一般在 train discriminator 的时候，discriminator 只看一张 image，决定它是好的还是不好。

Mini-batch Discriminator 是让 discriminator 看一把 image，决定它是好的还是不好，看一把 image 跟看一张 image 有什么不同？看一把 image 的时候不仅要 check 每一张 image 是不是好的，还要 check 这些 image 它们看起来像不像。

discriminator 会从 database 里面 sample 一把 image 出来，会让 generator sample 一把 image 出来，如果 generator 每次 sample 都是一样的 image，发生 Mode collapse 的情形，discriminator 就会抓到这件事，因为在 training data 里面每张图都差很多，如果 generator 产生的图都很像，discriminator 因为它不是只看一张图，它是看一把图，它就会抓到这把图看起来不像是 realistic。

还有另外一个也是看一把图的方法，叫做 OTGAN，Optimal Transport GAN。

Transfer Learning

Transfer Learning

迁移学习，主要介绍共享 layer 的方法以及属性降维对比的方法

迁移学习，transfer learning，旨在利用一些不直接相关的数据对完成目标任务做出贡献

以猫狗识别为例，解释“不直接相关”的含义：

- input domain 是类似的，但 task 是无关的

比如输入都是动物的图像，但这些data是属于另一组有关大象和老虎识别的task

- input domain是不同的，但task是一样的

比如task同样是做猫狗识别，但输入的是卡通图

Dog/Cat
Classifier



cat



dog

Data not directly related to the task considered



elephant



tiger



dog

cat

Similar domain, different tasks

Different domains, same task

迁移学习问的问题是：我们能不能再有一些不相关data的情况下，然后帮助我们现在要做的task。

为什么要考虑迁移学习这样的task呢？

举例来说：在speech recognition里面(台语的语音辨识)，台语的数据是很少的(但是语音的数据是很好收集的，中文，英文等)。那我们能不能用其他语音的数据来做台语这件事情。

或者在image recognition里面有一些是medical images，你想要让机器自动诊断说，有没有 tumor 之类的，这种medical image其实是很少的，但是image data是很不缺的。

或者是在文件的分析上，你现在要分析的文件是某个很 specific 的 domain，比如说你想要分析的是，某种特别的法律的文件，那这种法律的文件或许 data 很少，但是假设你可以从网络上 collect 一大堆的数据，那这些 data，有可能是有帮助的。

用不相干的数据来做domain其他的data，来帮助现在的task，是有可能的。事实上，我们在日常生活中经常会使用迁移学习，比如我们会把漫画家的生活自动迁移类比到研究生的生活。

迁移学习有很多的方法，它是很多方法的集合。下面你有可能会看到我说的terminology可能跟其他的有点不一样，不同的文献用的词汇其实是不一样的，有些人说是迁移学习，有些人说不是迁移学习，所以这个地方比较混乱，你只需要知道那个方法是什么就好了。

我们现在有一个我们想要做的task，有一些跟这个task有关的数据叫做**target data**，有一些跟这个task无关的数据，这个data叫做**source data**。这个target data有可能是有label的，也有可能是没有label的，这个source data有可能是有label的，也有可能是没有label的，所以现在我们就有四种可能，所以之后我们会分这四种来讨论。

Case 1

这里target data和source data都是带有标签的：

- target data: (x^t, y^t) , 作为有效数据, 通常量是很少的
如果target data量非常少, 则被称为**one-shot learning**
- source data: (x^s, y^s) , 作为不直接相关数据, 通常量是很多的

Model Fine-tuning

One-shot learning: only a few examples in target domain

- Task description
 - Source data: (x^s, y^s) ← A large amount
 - Target data: (x^t, y^t) ← Very little
- Example: (supervised) speaker adaption
 - Source data: audio data and transcriptions from many speakers
 - Target data: audio data and its transcriptions of specific user
- Idea: training a model by source data, then fine-tune the model by target data
 - Challenge: only limited target data, so be careful about overfitting

Model Fine-tuning

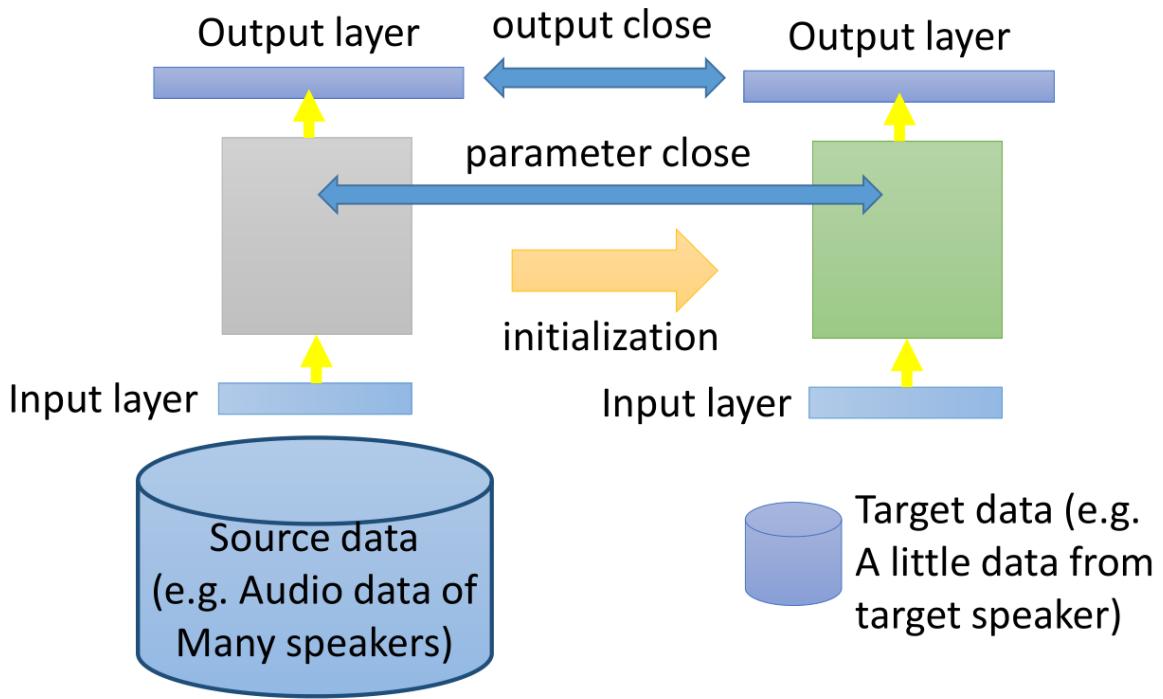
模型微调的基本思想：用source data去训练一个model，再用target data对model进行微调(fine tune)

所谓“微调”，类似于pre-training，就是把用source data训练出的model参数当做是参数的初始值，再用target data继续训练下去即可，但当target data非常少时，可能会遇到的challenge是，你在source data train出一个好的model，然后在target data上做train，可能就坏掉了。

所以训练的时候要小心，有许多技巧值得注意

Conservation Training

如果现在有大量的source data，比如在语音识别中有大量不同人的声音数据，可以拿它去训练一个语音识别的神经网络，而现在你拥有的target data，即特定某个人的语音数据，可能只有十几条左右，如果直接拿这些数据去再训练，肯定得不到好的结果。



此时我们就需要在训练的时候加一些限制，让用target data训练前后的model不要相差太多：

- 我们可以让新旧两个model在看到同一笔data的时候，output越接近越好
- 或者让新旧两个model的L2 norm越小越好，参数尽可能接近
- 总之让两个model不要相差太多，防止由于target data的训练导致过拟合

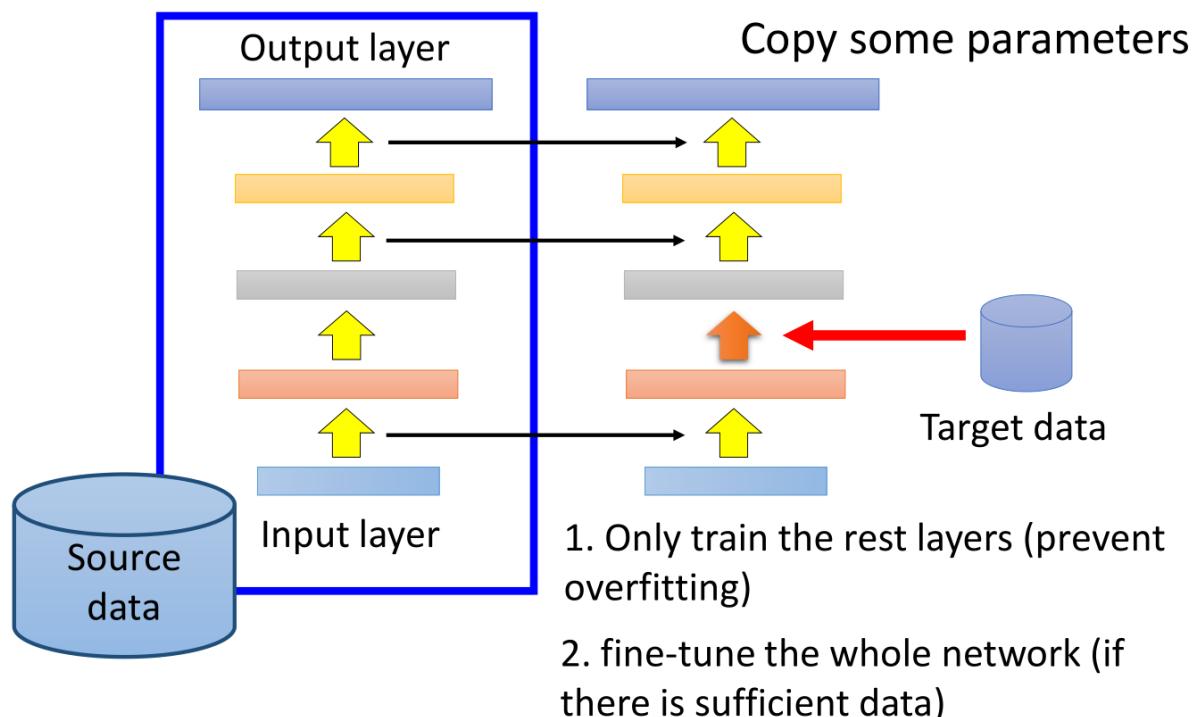
注：这里的限制就类似于regularization

Layer Transfer

现在我们已经有一个用source data训练好的model，此时把该model的某几个layer拿出来复制到同样大小的新model里，接下来只用target data去训练余下的没有被复制到的layer

这样做的好处是target data只需要考虑model中非常少的参数，这样就可以避免过拟合。

如果target data足够多，fine-tune 整个model也是可以的。



Layer Transfer是个非常常见的技巧，接下来要面对的问题是，哪些layer应该被transfer，哪些layer不应该去transfer呢？

有趣的是在不同的task上面需要被transfer的layer往往是不一样的：

- 在语音识别中，往往迁移的是最后几层layer，再重新训练与输入端相邻的那几层

由于口腔结构不同，同样的发音方式得到的发音是不一样的，NN的前几层会从声音信号里提取出发音方式，再用后几层判断对应的词汇，从这个角度看，NN的后几层是跟特定的人没有关系的，因此可做迁移

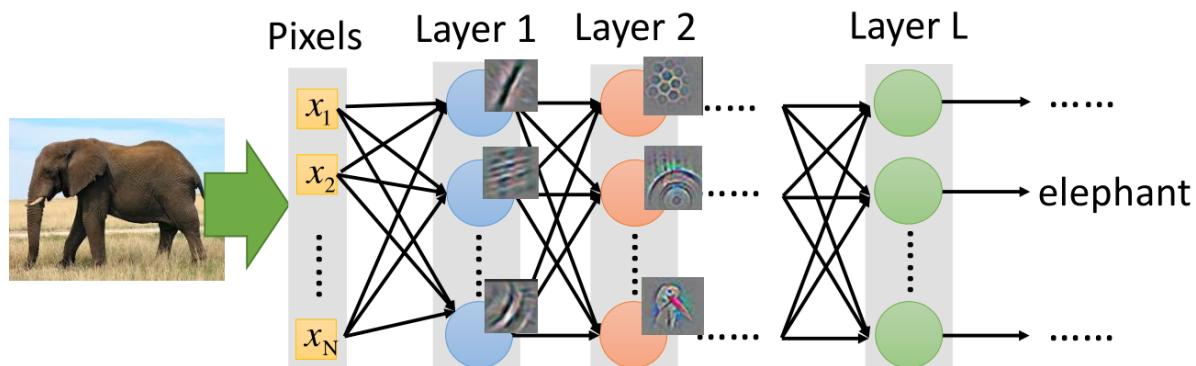
- 在图像处理中，往往迁移的是前面几层layer，再重新训练后面的layer

CNN在前几层通常是做最简单的识别，比如识别是否有直线斜线、是否有简单的几何图形等，这些layer的功能是可以被迁移到其它task上通用的

- case by case，运用之妙，存乎一心

- **Which layer can be transferred (copied)?**

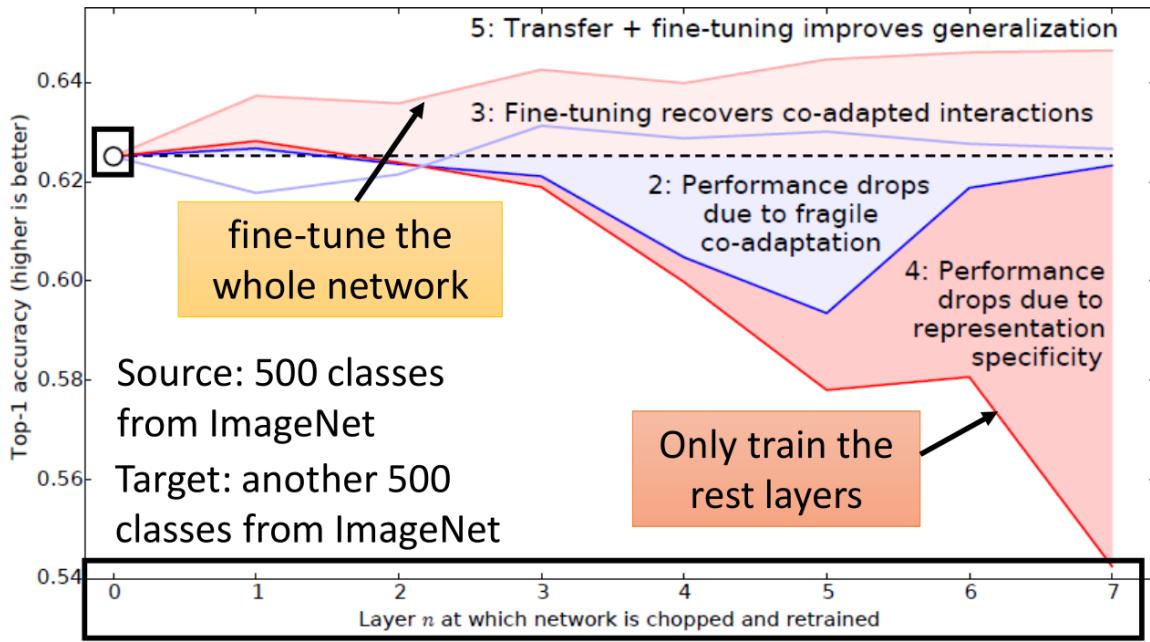
- Speech: usually copy the last few layers
- Image: usually copy the first few layers



Demo

这边是 image 在 layer transfer 上的实验，这个实验做在 ImageNet 上，把 ImageNet 的 corpus，一百二十万张 image 分成 source 跟 target。这个分法是按照 class 来分的，我们知道 ImageNet 的 image 一个 typical 的 setup 是有一千个 class，把其中五百个 class 归为 source data，把另外五百个 class 归为 target data。

横轴是我们在做 transfer learning 的时候，copy 了几个 layer。copy 0 个 layer 就代表完全没有做 transfer learning。这是一个 baseline，就直接在 target data 上面 train 下去。纵轴是 top-1 accuracy，所以是越高越好。没有做 transfer learning 是白色这个点



Jason Yosinski, Jeff Clune, Yoshua Bengio, Hod Lipson, "How transferable are features in deep neural networks?", NIPS, 2014

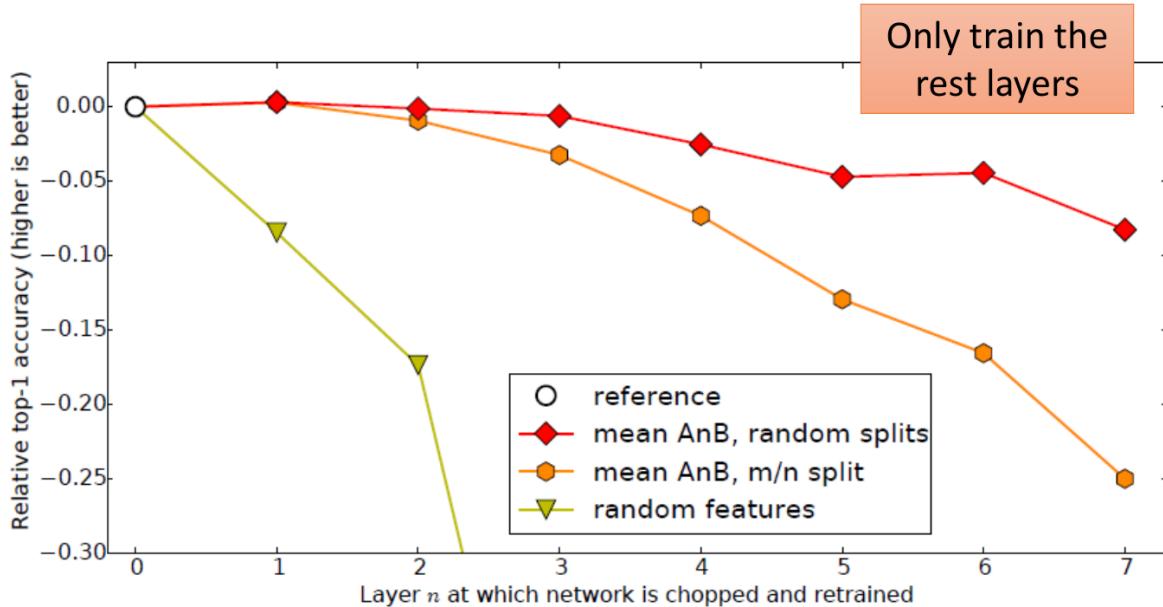
只有 copy 第一个 layer 的时候，performance 稍微有点进步，copy 前面两个 layer，performance 几乎是持平的，但是 copy 的 layer 太多，结果是会坏掉。

这个实验显示说在不同的 data 上面，train 出来的 neural network，前面几个 layer 是可以共享的，后面几个可能是没有办法共享的。如果 copy 完以后，还有 fine-tune 整个 model 的话，把第一个 layer，在 source domain 上 train 一个 model，然后把第一个 layer copy 过去以后，再用 target domain fine-tune 整个 model，包括前面 copy 过的 layer 的话，那得到 performance 是橙色这条线，在所有的 case 上面都是有进步的。

其实这个结果很 surprised，不要忘了，这可是 ImageNet 的 corpus，一般在做 transfer learning 的时候，都是假设 target domain 的 data 非常少，这边 target domain 可是有六十万张，这 target domain 的 data 是非常多的。但是就算在这个情况下，再多加了另外六十张 image 做 transfer learning，其实还是有帮助的。

这两条蓝色的线跟 transfer learning 没有关系，不过是这篇paper里面发现一个有趣的现象。他想要做一个对照组，在 target domain 上面 learn 一个 model，把前几个 layer copy 过来，再用一次 target domain 的 data train 剩下几个 layer。前面几个 layer 就 fix 住，只 train 后面几个 layer，直觉上这样做应该跟直接 train 整个 model 没有太大差别，先 train 好一个 model，fix 前面几个 layer，接下来只 train 后面几个 layer，结果有些时候是会坏掉的。他的理由是 training 的时候，前面的 layer 跟后面的 layer 他们其实是要互相搭配，所以如果只 copy 前面的 layer，然后只 train 后面的 layer，后面的 layer 就没有办法跟前面的 layer 互相搭配，结果有点差。如果可以 fine-tune 整个 model 的话，performance 就跟没有 transfer learning 是一样的。这是另一个有趣的发现，作者自己对这件事情是很 surprised。

这是另外一个实验结果，红色这条线是前一页看到的红色的这条线。这边假设 source 跟 target 是比较没有关系的，把 ImageNet 的 corpus 分成 source data 跟 target data 的时候把自然界的东西，通通当作 source，target 通通是人造的东西，桌子、椅子等等。这样 transfer learning 会有什么样的影响？



Jason Yosinski, Jeff Clune, Yoshua Bengio, Hod Lipson, "How transferable are features in deep neural networks?", NIPS, 2014

如果 source 跟 target 的 data 是差很多的，在做 transfer learning 的时候，performance 会掉的比较多，前面几个 layer 影响还是比较小的，如果只 copy 前面几个 layer，仍然跟没有 copy 是持平的，这意味着，就算是 source domain 跟 target domain 是非常不一样的，一边是自然的东西，一边是人造的东西，在 neural network 第一个 layer，他们仍然做的事情很有可能是一样的。黄色的这条线，烂掉的这条线是假设前面几个 layer 的参数是 random 的。

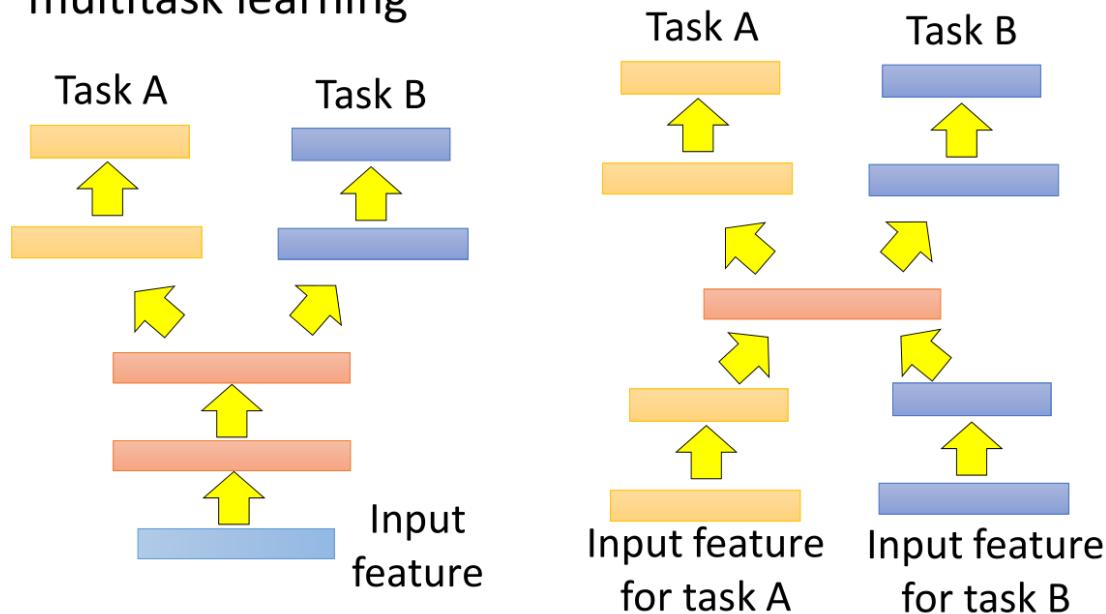
Multitask Learning

fine-tune仅考虑在target data上的表现，而多任务学习，则是同时考虑model在source data和target data上的表现

如果两个task的输入特征类似，则可以用同一个神经网络的前几层layer做相同的工作，到后几层再分方向到不同的task上，这样做的好处是前几层得到的数据比较多，可以被训练得更充分。这样做的前提是：这两个task有共通性，可以共用前面几个layer。

有时候task A和task B的输入输出都不相同，两个不同task的input都用不同的neural network把它 transform 到同一个domain上去，中间可能有某几个 layer 是 share 的，也可以达到类似的效果

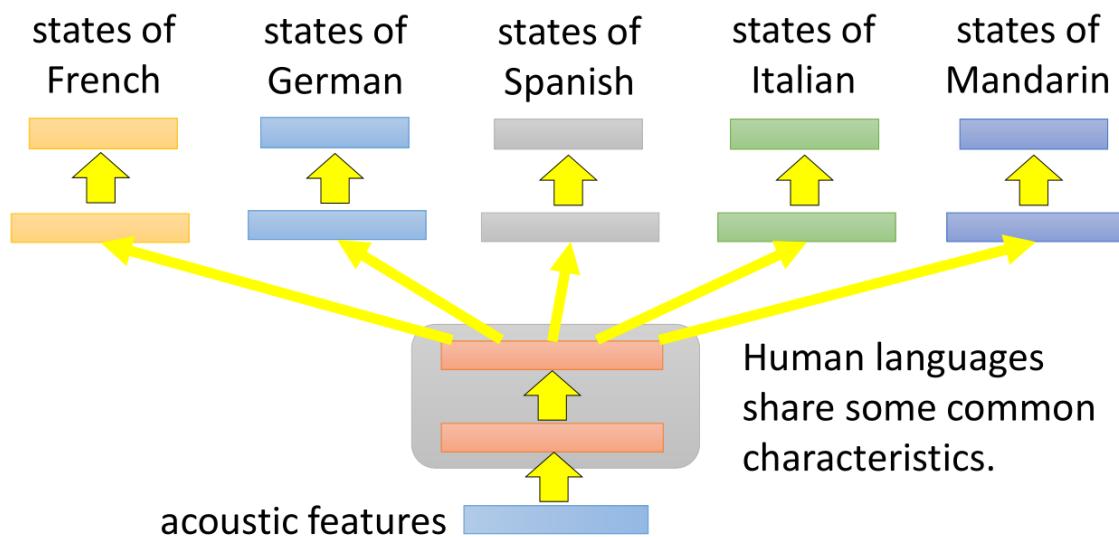
- The multi-layer structure makes NN suitable for multitask learning



以上方法要求不同的task之间要有一定的共性，这样才有共用一部分layer的可能性

Multilingual Speech Recognition

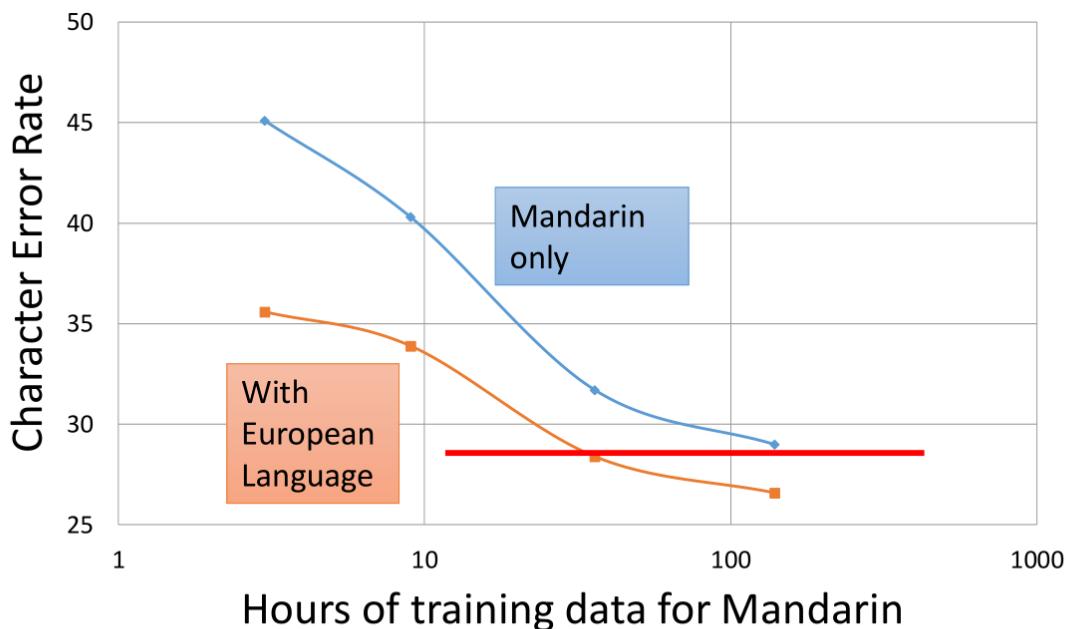
多任务学习一个很成功的例子就是多语言的语音辨识，假设你现在手上有一大堆不同语言的数据（法文，中文，英文等），那你在train你的model的时候，同时可以辨识这五种不同的语言。这个model前面几个layer他们会共用参数，后面几个layer每一个语言可能会有自己的参数，这样做是合理的。虽然是不同的语言，但是都是人类所说的，所以前面几个layer它们可能是share同样的特征，共用同样的参数。



Similar idea in translation: Daxiang Dong, Hua Wu, Wei He, Dianhai Yu and Haifeng Wang, "Multi-task learning for multiple language translation.", ACL 2015

在translation上，你也可以做同样的事情，假设你今天要做中翻英，也要做中翻日，你也把这两个model一起train。在一起train的时候无论是中翻英还是中翻日，你都要把中文的数据先做process，那一部分神经网络就可以是两种不同语言的数据共同使用。

在过去收集了十几种语言，把它们两两之间互相做transfer，做了一个很大的 $N \times N$ 的 table，每一个 case 都有进步。所以目前发现大部分case，不同人类的语言就算你觉得它们不是非常像，但是它们之间都是可以transfer的。



Huang, Jui-Ting, et al. "Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers." /CASSP, 2013

上图为从欧洲语言去transfer中文，横轴是中文的data，纵轴是character error rate。假设你一开始用中文train一个model，data很少，error rate很大，随着data越来越多，error rate就可以压到30以下。但是今天如果你有一大堆的欧洲语言，你把这些欧洲语言跟中文一起去做multi-task training，用这个欧洲语言的data来帮助中文model前面几层让它train的更好。你会发现：在中文data很少的情况下，你有做迁移学习，你就可以得到比较好的性能。随着中文data越多的时候，中文本身performance越好，就算是中文有一百小时的data，借用一些从欧洲语言来的knowledge，对这个辨识也是有微幅帮助的。

所以这边的好处是：假设你做多任务学习的时候，你会发现你有100多个小时跟50小时对比，如果你有做迁移学习的话，你只需要1/2的数据就可以跟有两倍的数据做的一样好。

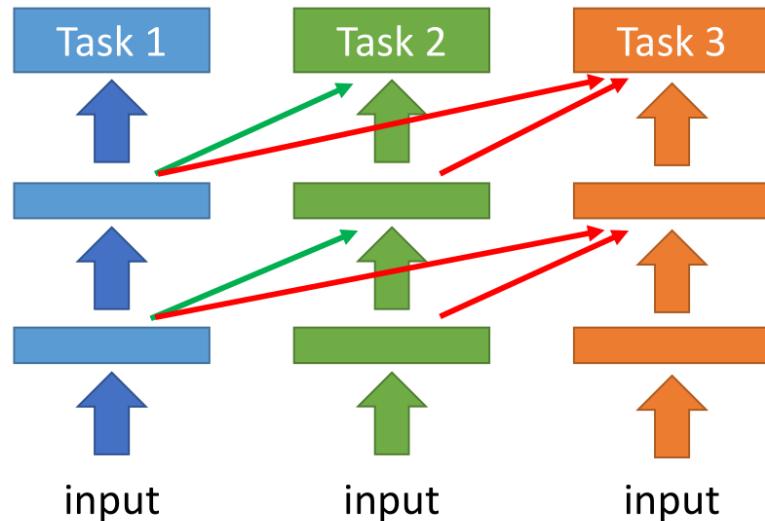
常常有人会担心说：迁移学习会不会有负面的效应，这是有可能的，如果两个task不像的话，你的transfer就是negative的。但是有人说：总是思考两个task到底之间能不能transfer，这样很浪费时间。所以有人 propose 了 progressive neural networks

Progressive Neural Network

如果两个task完全不相关，硬是把它们拿来一起训练反而会起到负面效果

而在Progressive Neural Network中，每个task对应model的hidden layer的输出都会被接到后续model的hidden layer的输入上，这样做好处是：

- task 2的数据并不会影响到task 1的model，因此task 1一定不会比原来更差
- task 2虽然可以借用task 1的参数，但可以将之直接设为0，最糟的情况下就等于没有这些参数，也不会对本身的performance产生影响
- task 3也做一样的事情，同时从task 1和task 2的hidden layer中得到信息



Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, Raia Hadsell, "Progressive Neural Networks", arXiv preprint 2016

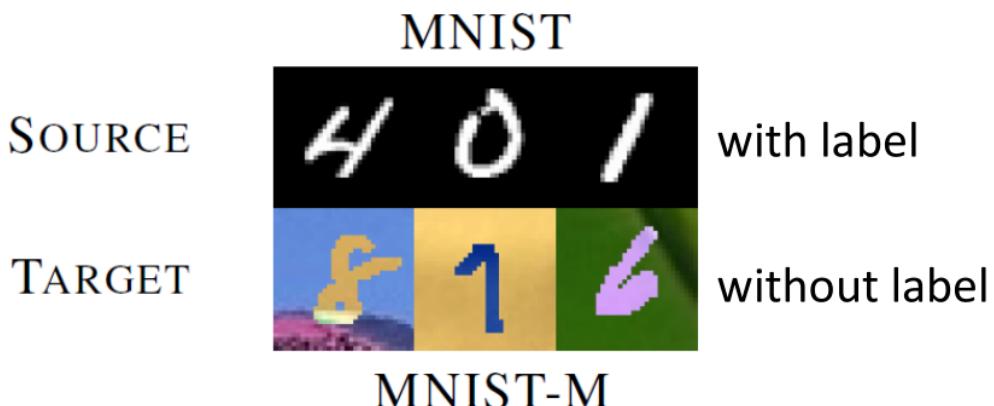
Case 2

这里target data不带标签，而source data带标签：

- target data: (x^t)
- source data: (x^s, y^s)

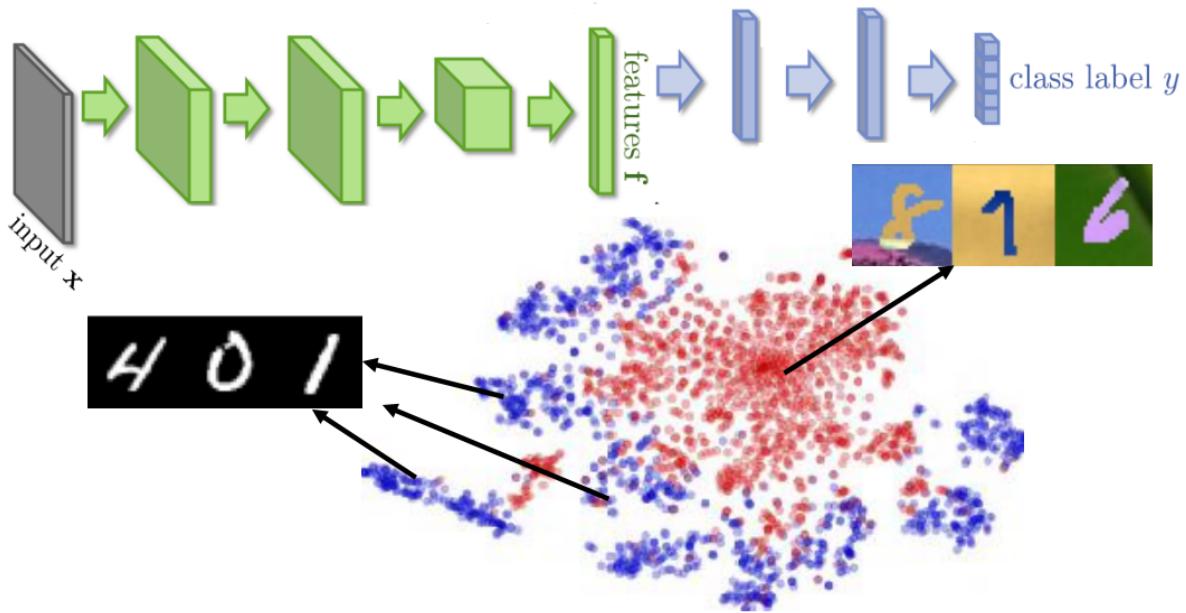
举例来说：我们可以说：source data是MNIST image，target data是MNIST-M image(MNIST image加上一些奇怪的背景)。MNIST是有label的，MNIST-M是没有label的，在这种情况下我们通常是把source data视作training data，target data视作testing data。产生的问题是：training data跟testing data是非常mismatch的。

- Source data: $(x^s, y^s) \rightarrow$ Training data
 - Target data: $(x^t) \rightarrow$ Testing data
- } Same task, mismatch



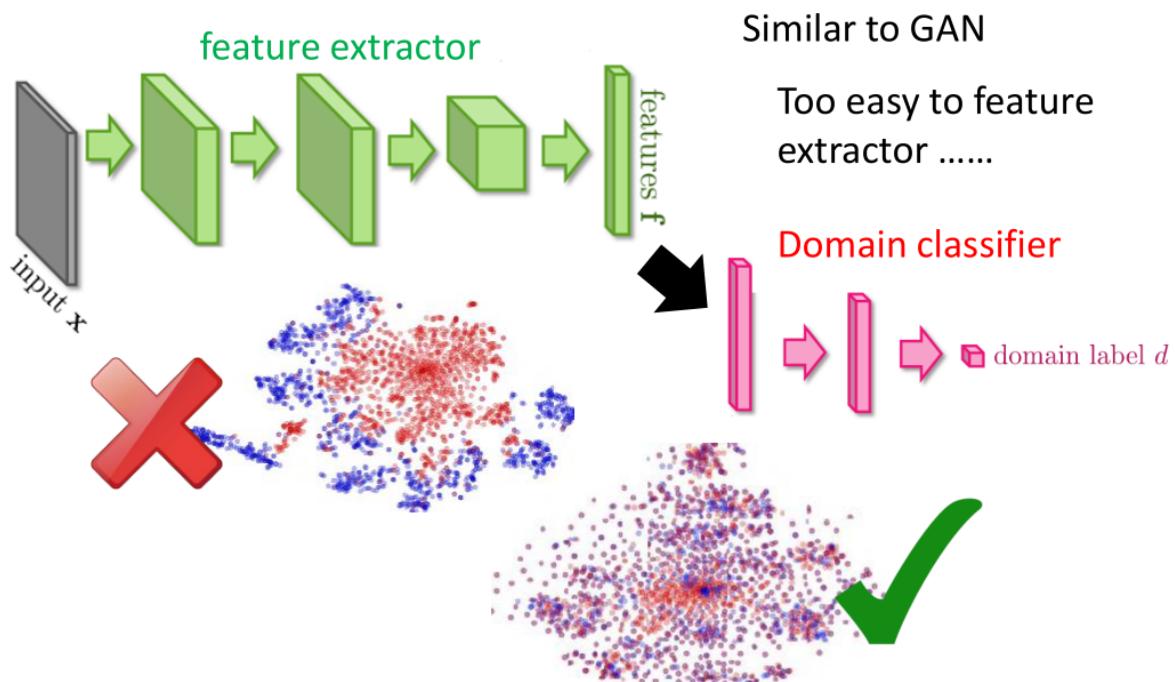
这个时候一般会把source data当做训练集，而target data当做测试集，如果不管训练集和测试集之间的差异，直接训练一个普通的model，得到的结果准确率会相当低。

实际上，神经网络的前几层可以被看作是在抽取feature，后几层则是在做classification，如果把用MNIST训练好的model所提取出的feature做t-SNE降维后的可视化，可以发现MNIST的数据特征明显分为紫色的十团，分别代表10个数字，而作为测试集的数据却是挤成一团的红色点，因此它们的特征提取方式根本不匹配。



Domain-adversarial Training

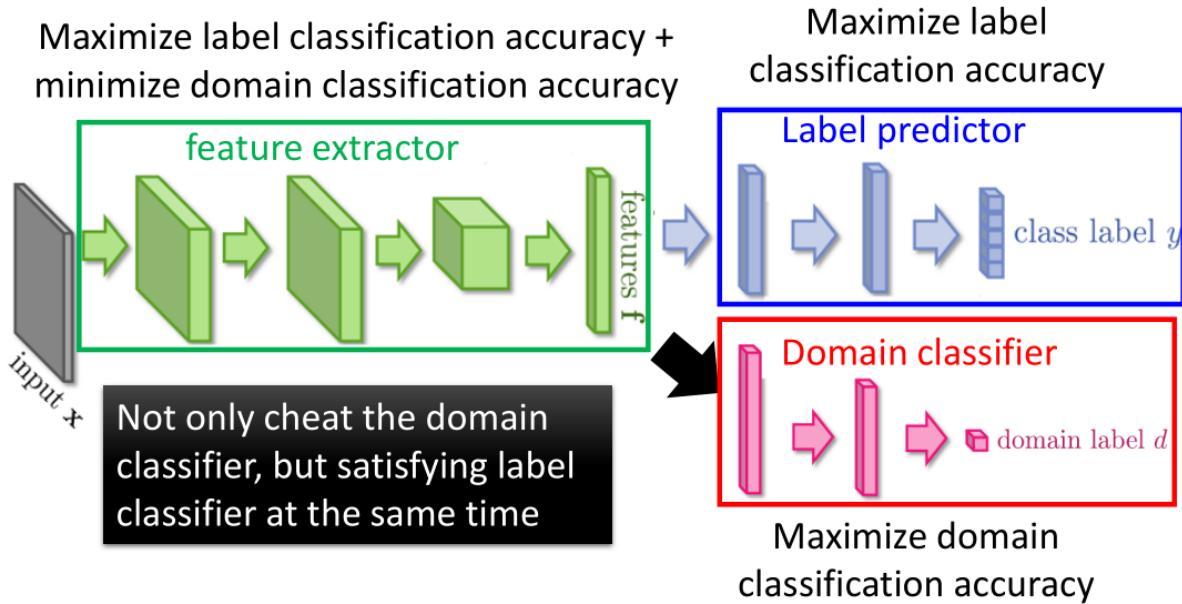
所以该怎么办呢？希望做的事情是：前面的feature extract 它可以把domain的特性去除掉，这一招较做 Domain-adversarial training。也就是feature extract output不应该是红色跟蓝色的点分成两群，而是不同的domain应该混在一起。



那如何learn这样的feature extract呢？这边的做法是在后面接一下domain classifier。把feature extract output丢给domain classifier，domain classifier它是一个classification task，它要做的事情就是：根据feature extract给它的feature，判断这个feature来自于哪个domain，在这个task里面，要分辨这些feature是来自MNIST还是来自与MNIST-M。

有一个generator的output，然后又有discriminator，让它的架构非常像GAN。但是跟GAN不一样的事情是：之前在GAN那个task里面，你的generator要做的事情是产生一个image，然后骗过discriminator，这件事很难。

但是在这个Domain-adversarial training里面，要骗过domain classifier太简单了。有一个solution是：不管看到什么东西，output都是0，这样就骗过了classifier。



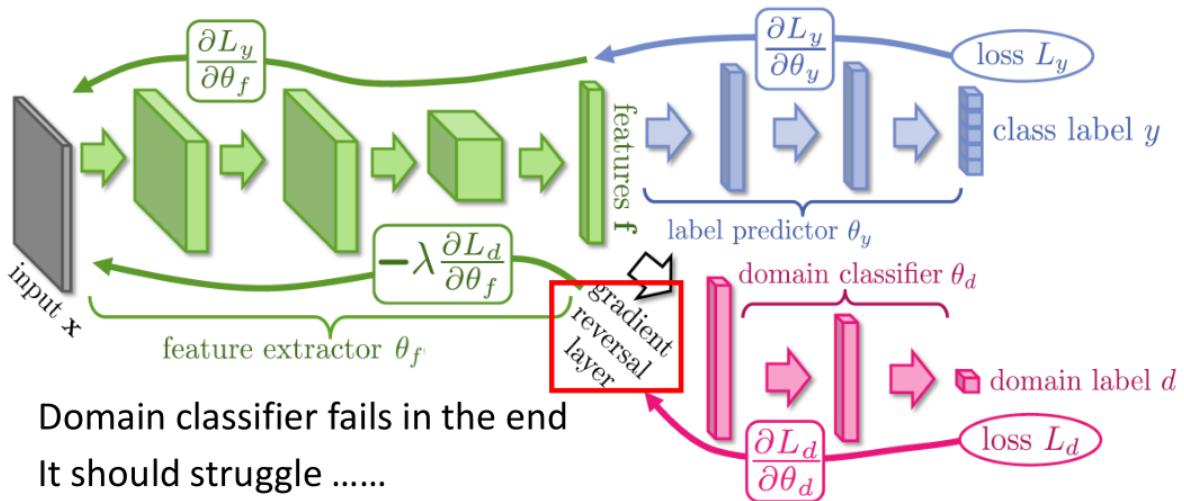
This is a big network, but different parts have different goals.

所以你要在feature extract增加它任务的难度，所以feature extract它output feature不仅要骗过domain classifier还要同时让label predictor做好。这个label predictor它就吃feature extract output，然后它的output就是10个class。

所以今天你的feature extract 不只要骗过domain classifier，还要满足label predictor的需求。抽出的feature不仅要把domain的特性消掉，同时还要保留原来feature的特性。

那我们把这三个neural放在一起的话。实际上就是一个大型的neural network，是一个各怀鬼胎的neural network(一般的neural network整个参数想要做的事情都是一样的，要minimize loss)，在这个neural network里面参数的目标是不同的。label predictor做的事情是把class分类做的正确率越高越好，domain classifier做的事情是想正确predict image是属于哪个domain。feature extractor想要做的事情是：要同时improve label predictor，同时想要minimize domain classifier accuracy，所以feature extractor 其实是在做陷害队友这件事的。

feature extractor 怎样陷害队友呢(domain classifier)？这件事情是很容易的，你只要加一个gradient reversal layer就行了。也就是你在做back-propagation(Domain classifier 计算 back propagation 有forward 跟 backward 两个 path)，在做backward task的时候你的domain classifier传给feature extractor什么样的value，feature extractor就把它乘上一个负号。也就是domain classifier 告诉你说某个value要上升，它就故意下降。

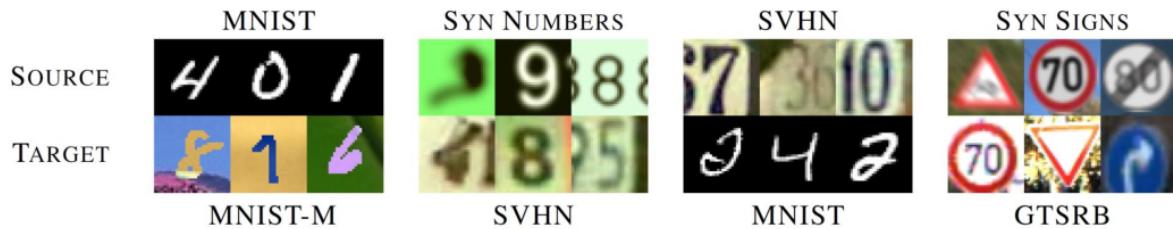


Yaroslav Ganin, Victor Lempitsky, Unsupervised Domain Adaptation by Backpropagation, ICML, 2015

Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, Domain-Adversarial Training of Neural Networks, JMLR, 2016

domain classifier因为看不到真正的image，所以它最后一定fail掉。因为它所能看到的东西都是feature extractor告诉它的，所以它最后一定会无法分辨feature extractor所抽出来的feature是来自哪个domain。

这个 model 原理讲起来很简单，但可能实际上的 training 可能跟 GAN 一样是没有那么好 train 的，问题就是domain classifier一定要奋力的挣扎，因为它要努力去判断现在的feature是来自哪个domain。如果 domain classifier 他比较弱、懒惰，他一下就放弃不想做了，就没有办法把前面的 feature extractor 逼到极限，就没有办法让 feature extractor 真的把 domain information remove 掉。如果 domain classifier 很 weak，他一开始就不想做了，他 output 永远都是 0 的话，那 feature extractor 胡乱弄什么 feature 都可以骗过 classifier 的话，那就达不到把 domain 特性 remove 掉的效果，这个 task 一定要让 domain classifier 奋力挣扎然后才死掉，这样才能把 feature extractor 的潜能逼到极限。



METHOD	SOURCE	MNIST	SYN NUMBERS	SVHN	SYN SIGNS
	TARGET	MNIST-M	SVHN	MNIST	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

Yaroslav Ganin, Victor Lempitsky, Unsupervised Domain Adaptation by Backpropagation, ICML, 2015

Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, Domain-Adversarial Training of Neural Networks, JMLR, 2016

这是 paper 一些实验的结果，做不同 domain 的 transfer。如果看实验结果的话，纵轴代表用不同的方法，这边有一个 source only 的方法，直接在 source domain 上 train 一个 model，然后 test 在 target domain 上，如果只用 source only 的话，Performance 是比较差的。这边比较另一个 transfer learning 的方法，大家可以自己去看参考文献。这篇paper proposed 的方法是刚刚讲的 domain-adversarial

training.

直接拿 target domain 的 data 去做 training，会得到 performance 是最下面这个 row，这其实是 performance 的 upper bound。用 source data 跟 target data train 出来的结果是天差地远的，这中间有一个很大的 gap。

如果用 domain-adversarial training 可以在不同的 case 上面都有很好的 improvement。

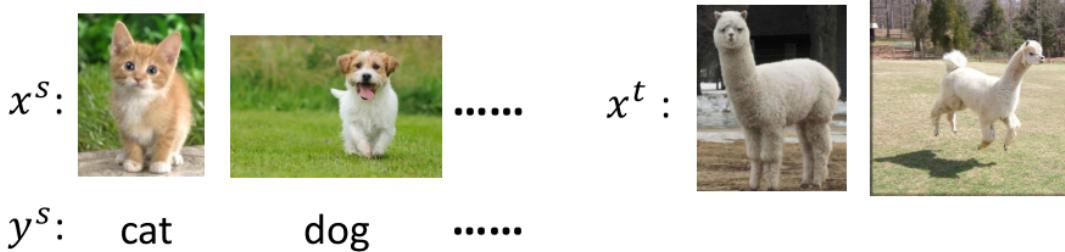
Zero-shot Learning

在zero-shot-learning里面跟刚才讲的task是一样的，source data有label，target data没有label。

在刚才task里面可以把source data当做training data，把target data当做testing data，但是实际上在 zero-shot learning里面，它的define又更加严格一点。它的define是：今天在source data和target data里面，它的task是不一样的。

比如说在影像上面(你可能要分辨猫跟狗)，你的source data可能有猫的class，也有狗的class。但是你的 target data里面image是草泥马，在source data里面是从来没有出现过草泥马的，如果machine看到草泥马，就未免有点强人所难了。但是这个task在语音上很早就有solution了，其实语音是常常会遇到zero-shot learning的问题。

- Source data: $(x^s, y^s) \rightarrow$ Training data
 - Target data: $(x^t) \rightarrow$ Testing data
- } Different tasks



In speech recognition, we can not have all possible words in the source (training) data.

How we solve this problem in speech recognition?

假如我们把不同的word都当做一个class的话，那本来在training的时候跟testing的时候就有可能看到不同的词汇。你的testing data本来就有一些词汇是在training的时候是没有看过的。

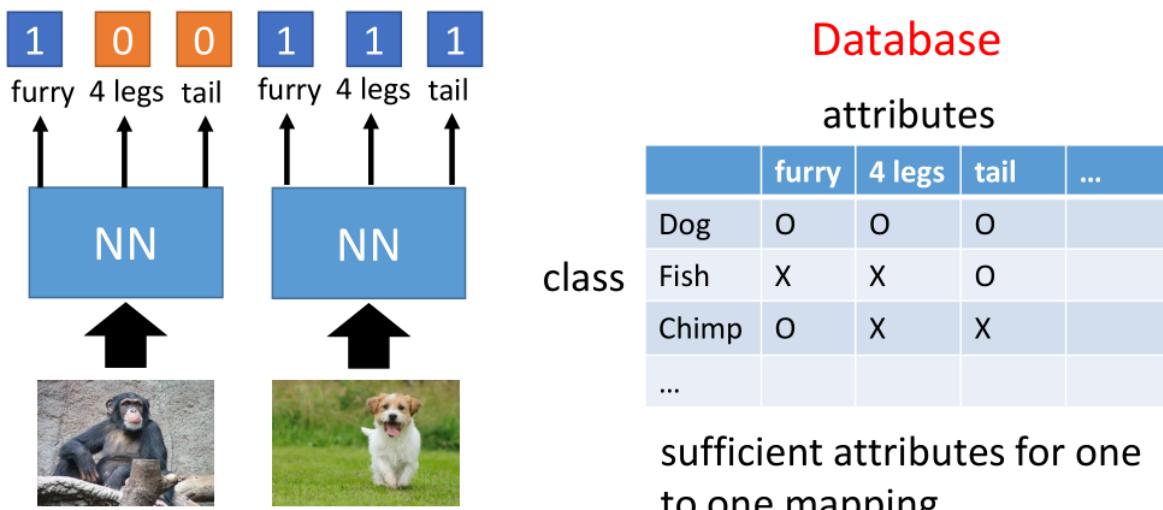
那在语音上我们如何来解决这个问题呢？不要直接去辨识一段声音是属于哪一个word，我们辨识的是一段声音是属于哪一个phoneme。然后我们在做一个phoneme跟table对应关系的表，这个东西也就是 lexicon(词典)。在辨识的时候只要辨识出phoneme就好，再去查表说：这个phoneme对应到哪一个 word。

这样就算有一些word是没有在training data里面的，它只要在lexicon里面出现过，你的model可以正确辨识出声音是属于哪一个phoneme的话，你就可以处理这个问题。

Attribute embedding

- Representing each class by its attributes

Training

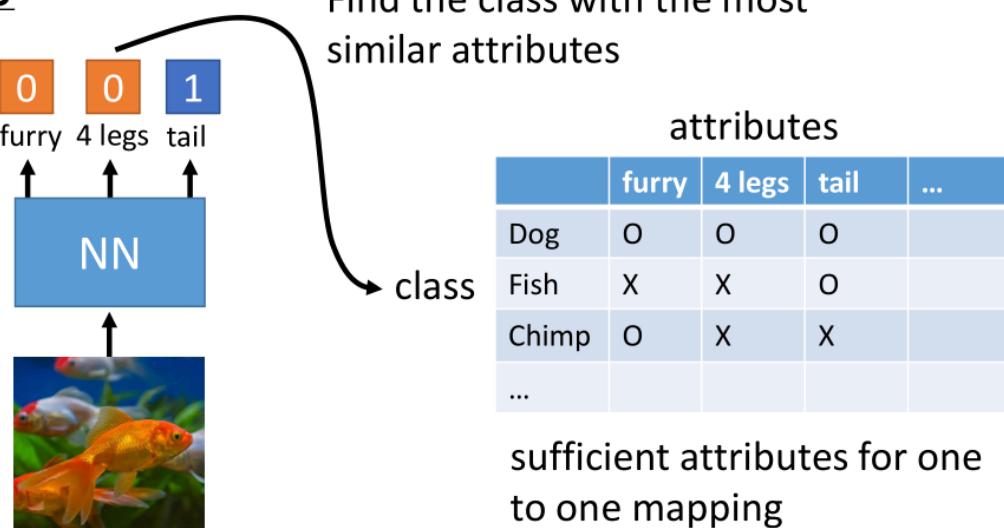


在影像上我们可以把每一个class用它的attribute来表示，也就是说：你有一个database，这个database里面会有所以不同可能的object跟它的特性。假设你要辨识的是动物，但是你training data跟testing data他们的动物是不一样的。但是你有一个database，这个database告诉你说：每一种动物它是有什么样的特性。比如狗就是毛茸茸，四只脚，有尾巴；鱼是有尾巴但不是毛茸茸，没有脚。

这个attribute要更丰富，每一个class都要有不一样的attribute(如果两个class有相同的attribute的话，方法会fail掉)。那在training的时候，我们不直接辨识每一张image是属于哪一个class，而是去辨识：每一张image里面它具备什么样的attribute。所以你的neural network target就是说：看到猩猩的图，就要说：这是一个毛茸茸的动物，没有四只脚，没有尾巴。看到狗的图就要说：这是毛茸茸的动物，有四只脚，有尾巴。

- Representing each class by its attributes

Testing



那在testing的时候，就算今天来了你从来没有见过的image，也是没有关系的。你今天neural network target也不是input image判断它是哪一种动物，而是input这一张image判断具有什么样的attribute。所以input你从来没有见过的动物，你只要把它的attribute找出来，然后你就查表看说：在database里面哪一种动物它的attribute跟你现在model output最接近。有时可能没有一摸一样的也是没有关系的，看谁

最接近，那个动物就是你要找的。

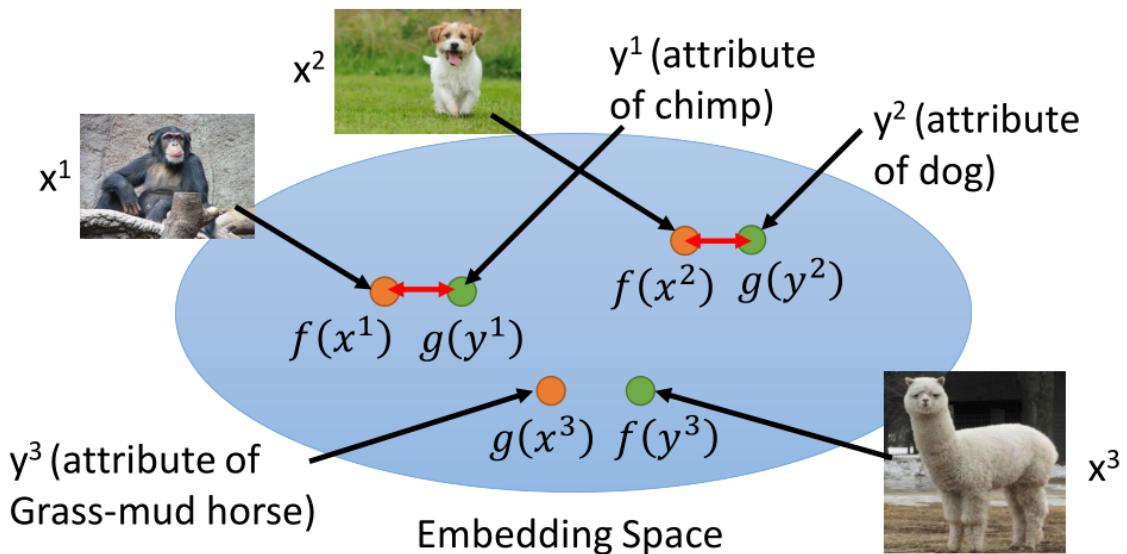
Zero-shot Learning

- Attribute embedding

$f(*)$ and $g(*)$ can be NN.

Training target:

$f(x^n)$ and $g(y^n)$ as close as possible



那有时候你的attribute可能非常的复杂(attribute dimension非常大)，你可以做attribute embedding。也就是说现在有一个embedding space，把training data每一个image都通过一个transform，变成一个embedding space上的一个点。然后把所有的attribute也都变成embedding space上的一个点，这个 $g(*)$ 跟 $f(*)$ 都可以是neural network，那training的时候希望 f 跟 g 越接近越好。那在testing的时候如果有一张没有看过的image，你就可以说这张image attribute embedding以后跟哪个attribute最像，那你就可以知道它是什么样的image。

image跟attribute都可以描述为vector，要做的事情就是把attribute跟image都投影到同一个空间里面。也就是说：你可以想象成是对image的vector，也就是图中的x，跟attribute的vector，也就是图中的y都做降维，然后都降到同一个dimension。所以你把x通过一个function f都变成embedding space上的vector，把y通过另外一个function g也都变成embedding space上的vector。

但是咋样找这个f跟g呢？你可以说f跟g就是neural network。input一张image它变成一个vector，或者input attribute 变成一个vector。training target是你希望说：假设我们已经知道 y^1 是 x^1 的attribute， y^2 是 x^2 的attribute，那你就希望说找到一个f跟g，它可以让 x^1 跟 y^1 投影到embedding space以后越接近越好， x^2 跟 y^2 投影到embedding space以后越接近越好。

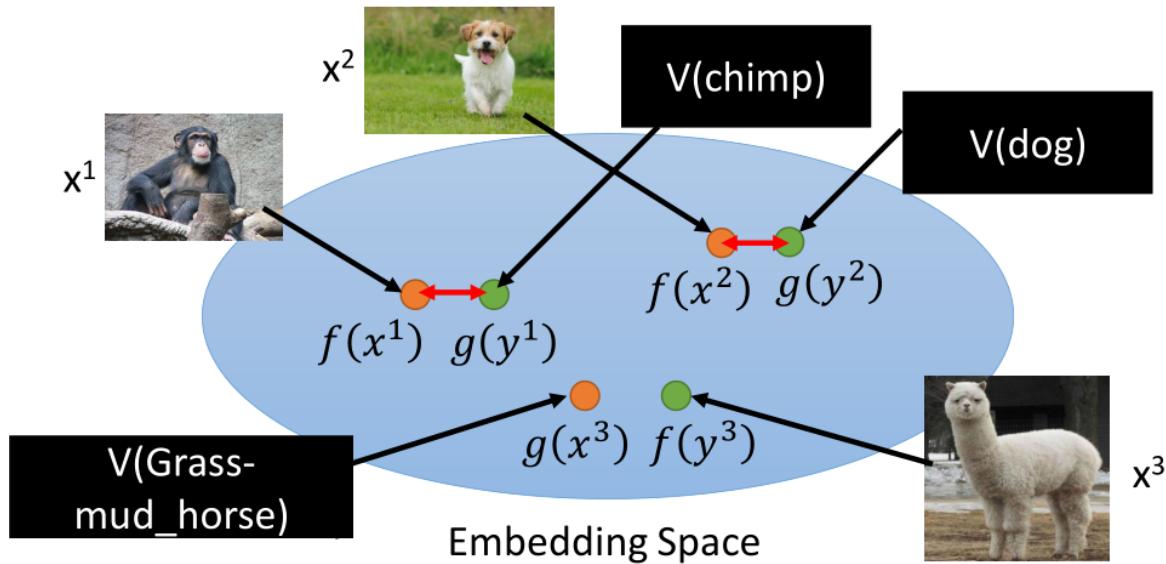
那现在把f跟g找出来了，那现在假如有一张你从来没见过的image x^3 在你的testing data里面，它也可以通过这个f变成embedding space上面的一个vector，接下来你就可以说这个embedding vector它跟 y^3 最接近，那 y^3 就是它的attribute，再来确定是哪个动物。

Attribute embedding + word embedding

Zero-shot Learning

What if we don't have database

- Attribute embedding + word embedding



又是你会遇到一个问题，如果我没有database呢？我根本不知道每一个动物的attribute是什么，怎么办呢？

可以借用 word vector，word vector 的每一个 dimension 就代表了现在这个 word 的某种 attribute。所以不一定需要有个 database 去告诉你每一个动物的 attribute 是什么。假设有一组 word vector，这组 word vector

里面你知道每个动物他对应的 word 的 word vector，这 word vector 你可以拿一个很大的 corpus，比如说 Wikipedia train 出来，就可以把 attribute 直接换成 word vector，所以把 attribute 通通换成那个 word 的 word vector，再做跟刚才一样的 embedding，就结束了。

$$f^*, g^* = \arg \min_{f,g} \sum_n \|f(x^n) - g(y^n)\|_2 \quad \text{Problem?}$$

$$f^*, g^* = \arg \min_{f,g} \sum_n \max \left(0, k - f(x^n) \cdot g(y^n) \right. \\ \left. + \max_{m \neq n} f(x^n) \cdot g(y^m) \right)$$

↑
Margin you defined

$$\text{Zero loss: } k - f(x^n) \cdot g(y^n) + \max_{m \neq n} f(x^n) \cdot g(y^m) < 0$$

$$\frac{f(x^n) \cdot g(y^n)}{\max_{m \neq n} f(x^n) \cdot g(y^m)} > k$$

$f(x^n)$ and $g(y^n)$ as close $f(x^n)$ and $g(y^m)$ not as close

这个loss function存在些问题，它会让model把所有不同的x和y都投影到同一个点上：

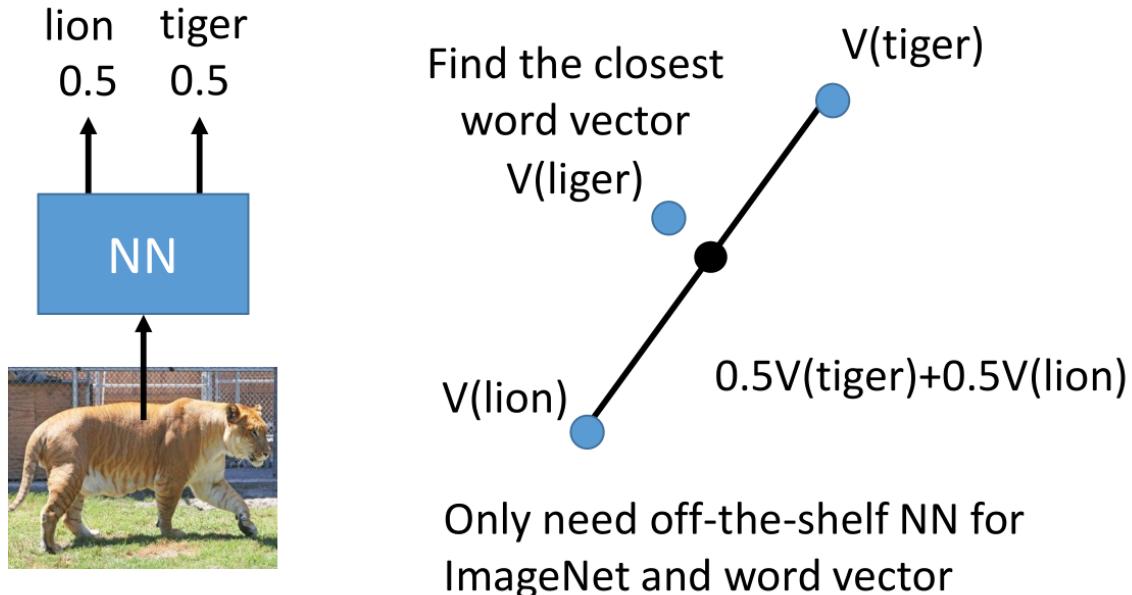
$$f^*, g^* = \arg \min_{f,g} \sum_n \|f(x^n) - g(y^n)\|_2$$

类似用t-SNE的思想，我们既要考虑同一对 x^n 和 y^n 距离要接近，又要考虑不属于同一对的 x^n 与 y^m 距离要拉大(这是前面的式子没有考虑到的)，于是有：

$$f^*, g^* = \arg \min_{f,g} \sum_n \max(0, k - f(x^n) \cdot g(y^n) + \max_{m \neq n} f(x^n) \cdot g(y^m))$$

0loss的情况是： x^n 跟 y^n 之间的inner product大过所有其它的 y^m 跟 x^n 之间的inner product，而且要大过一个margin k

Convex Combination of Semantic Embedding



还有另外一个简单的Zero-Shot learning的方法叫做convex combination of semantic embedding。这个方法是说：在这边不需要做任何 training，就可以做 transfer learning。假设有一个 off-the-shelf 识别系统，跟一个 off-the-shelf 的 word vector。这两个可能不是自己 train，或网络上载下来的。

我把一张图丢到neural network里面去，它的output没有办法决定是哪一个class，但它觉得有0.5的机率是lion，有0.5的机率是tiger。接下来去找lion跟tiger的word vector，然后把 lion 跟 tiger 的 word vector 用 1:1 的比例混合，0.5 tiger 的 vector 加 0.5 lion 的 vector，得到另外一个新的 vector。再看哪个word的vector跟这个混合之后的结果最接近。假设是liger最接近，那这个东西就是liger。

Example of Zero-shot Learning

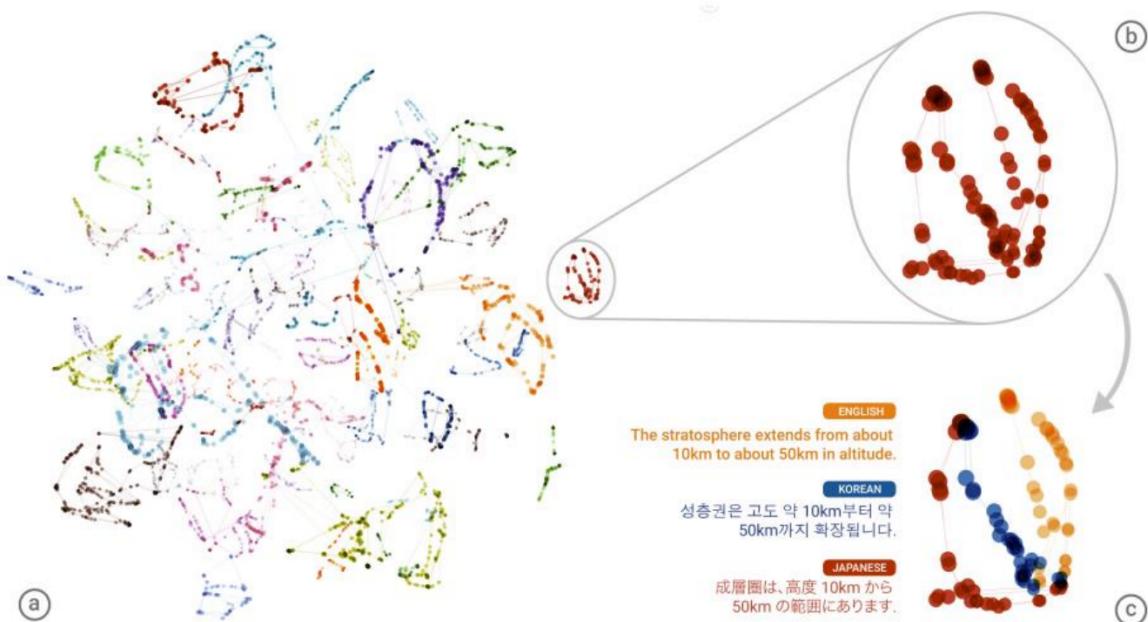
在training的时候，machine看过如何把英文翻译成韩文，知道咋样把韩文翻译为英文，知道咋样把英文翻译为日文，知道咋样把日文翻译为英文。但是它从来没有看过日文翻译韩文的数据，但是可以翻，但是它从来没有看过韩文翻译日文的数据，但是可以翻。

为什么zero-shot在这个task上是可行的呢？如果你今天用同一个model做了不同语言之间的translation以后，machine可以学到的事情是：对不同语言的input句子都可以project到同一个space上面。

在training的时候，machine看过如何把英文翻译成韩文，知道咋样把韩文翻译为英文，知道咋样把英文翻译为日文，知道咋样把日文翻译为英文。但是它从来没有看过日文翻译韩文的数据，但是可以翻，但是它从来没有看过韩文翻译日文的数据，但是可以翻。

为什么zero-shot在这个task上是可行的呢？如果你今天用同一个model做了不同语言之间的translation以后，machine可以学到的事情是：对不同语言的input句子都可以project到同一个space上面。

我们现在根据我们learn好的translation，那个translation有一个encoder，它会把你input的句子变成vector，decoder根据这个vector解回一个句子，就是翻译的结果。那今天我们把不同语言都丢到这个encoder里面让它变成vector的话，那这些不同语言的不同句子在这个space上面有什么不一样的关系呢？



它发现说今天有日文、英文、韩文这三个句子，这三个句子讲的是同一件事情，通过encoder embedding以后再space上面其实是差不多的位置。在左边这个图上面不同的颜色代表说：不同语言的用一个意思的句子。所以你这样说：machine发明了一个新语言也是可以接受的，如果你把这个embedding space当做一个新的语言的话。machine做的是：发现可一个sequence language，每一种不同的语言都先要先转成它知道的sequence language，在用这个sequence language转为另外一种语言。

所以今天就算是某一个翻译task，你的input语言和output语言machine没有看过，它也可以通过过这种自己学出来的sequence language来做translation。

Case 3 & 4

刚刚讲的状况都是 source data 有 label 的状况，有时候会遇到 source data 没有 label 的状况。

target data 有 label，source data 没有 label，这种是 Self-taught learning

target data 没有 label，source data 也没有 label，这种是 Self-taught clustering

有一个要强调的是 Self-taught learning 跟 source data 是 unlabeled data，target data 是 labeled data

这也是一种 semi-supervised learning。这种 semi-supervised learning 跟一般 semi-supervised learning 有一些不一样，一般 semi-supervised learning 会假设那些 unlabeled data 至少还是跟 labeled data 是比较有关系的。但在 Self-taught learning 里面，那些 unlabeled data、那些 source data，跟 target data 关系比较远。

其实 Self-taught learning 概念很简单，假设 source data 够多，虽然它是 unlabeled，可以去 learn 一个 feature extractor，在原始的 Self-taught learning paper 里面，他的 feature extractor 是 sparse coding。因为这 paper 比较旧，大概十年前，现在也不见得要用 sparse coding 也可以 learn，比如说 auto-encoder。

总之，有大量的 data，他们没有 label，可以做的是用这些 data learn 一个好的 feature extractor，learn 一个好的 representation。用这个 feature extractor 在 target data 上面去抽 feature。

在 Self-taught learning 原始的 paper 里面，其实做了很多 task，这些 task 都显示 Self-taught learning 是可以得到显著 improvement 的。

Learning to extract better representation from the source data (unsupervised approach)

Extracting better representation for target data

Concluding Remarks

		Source Data (not directly related to the task)	
		labelled	unlabeled
Target Data	labelled	Fine-tuning Multitask Learning	Self-taught learning Rajat Raina , Alexis Battle , Honglak Lee , Benjamin Packer , Andrew Y. Ng, Self-taught learning: transfer learning from unlabeled data, ICML, 2007
	unlabeled	Domain-adversarial training Zero-shot learning	Different from semi-supervised learning Self-taught Clustering Wenyuan Dai, Qiang Yang, Gui-Rong Xue, Yong Yu, "Self-taught clustering", ICML 2008

##

Meta Learning

Meta Learning

Meta learning 总体来说就是让机器学习如何学习。

Introduction

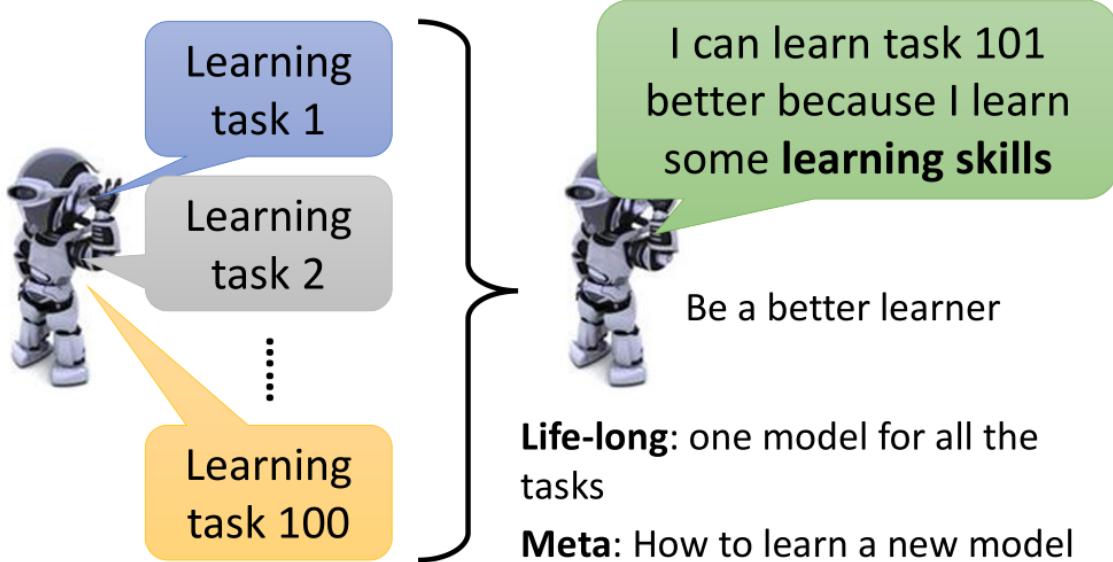
Task 1: speech recognition

Task 2: image recognition

:

Task 100: text classification

- Meta learning = Learn to learn



如上图，我们希望机器在学过一些任务以后，它学会如何去学习更有效率，也就是说它会成为一个更优秀的学习者，因为它学会了**学习的技巧**。举例来说，我们教机器学会了语音辨识、图像识别等模型以后，它就可以在文本分类任务上做的更快更好，虽然说语音辨识、图像识别和文本分类没什么直接的关系，但是我们希望机器从先前的任务学习中学会了学习的技巧。

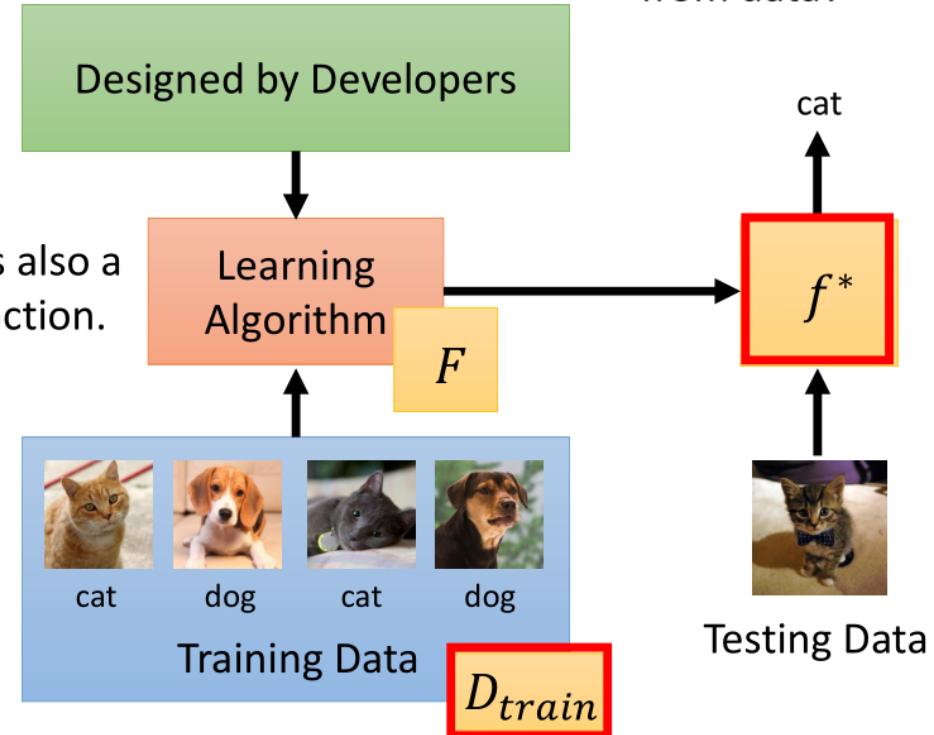
讲到这里，你可能会觉得Meta Learning 和 Life-Long Learning 有点像，确实很像，但是 Life-Long Learning 的着眼点是用同一个模型apply 到不同的任务上，让一个模型可以不断地学会新的任务，而 Meta Learning 中不同的任务有不同的模型，我们的着眼点是机器可以从过去的学习中学到学习的方法，让它在以后的学习中更快更好。

我们先来看一下传统的ML 的做法：

$$f^* = F(D_{train})$$

Meta Learning

Can machine find F from data?



我们过去学过的ML，通常来说就是定义一个学习算法，然后用训练数据train，吐出一组参数（或者说一个参数已定的函数式），也就是得到了模型，这个模型可以告诉我们测试数据应该对应的结果。比如我们做猫狗分类，train完以后，给模型一个猫的照片，它就会告诉我们这是一只猫。

我们把学习算法记为 F ，这个学习算法吃training data 然后吐出目标模型 f^* ，形式化记作：

$$f^* = F(D_{train})$$

Meta Learning 就是要让机器自动的找一个可以吃training data 吐出函数 f^* 的函数 F 。

总结一下：

Machine Learning ≈ 根據資料找一個函數 f 的能力



Meta Learning

≈ 根據資料找一個找一個函數 f 的函數 F 的能力



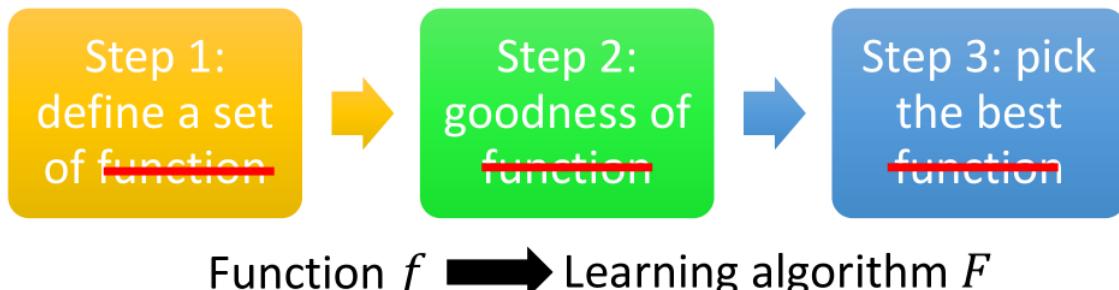
Machine Learning 和Meta Learning 都是让机器找一个function , 只不过要找的function 是不一样的。

我们知道Machine Learning 一共分三步 (如下图) , Meta Learning 也是一样的, 你只要把**Function f** 换成**学习的算法 F** 这就是Meta Learning 的步骤:

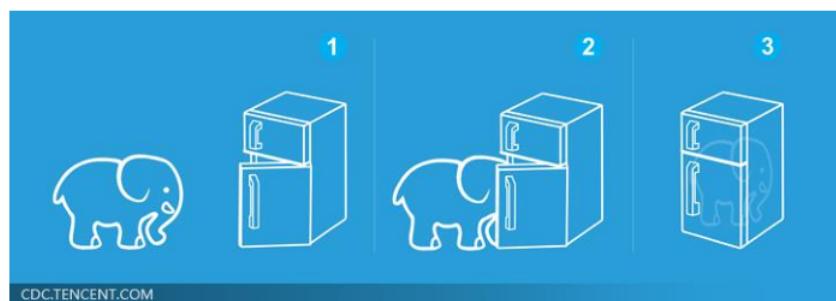
1. 我们先定义一组Learning 的Algorithm 我们不知道哪一个算法是比较好的,
2. 然后定义一个Learning Algorithm 的Loss , 它会告诉你某个算法的好坏,
3. 最后, 去train 一发找出哪个Learning Algorithm比较好。

所以接下来我们将分三部分来讲Meta Learning 的具体过程。

Machine Learning is Simple Meta



就好像把大象放进冰箱



Three Step of Meta-Learning

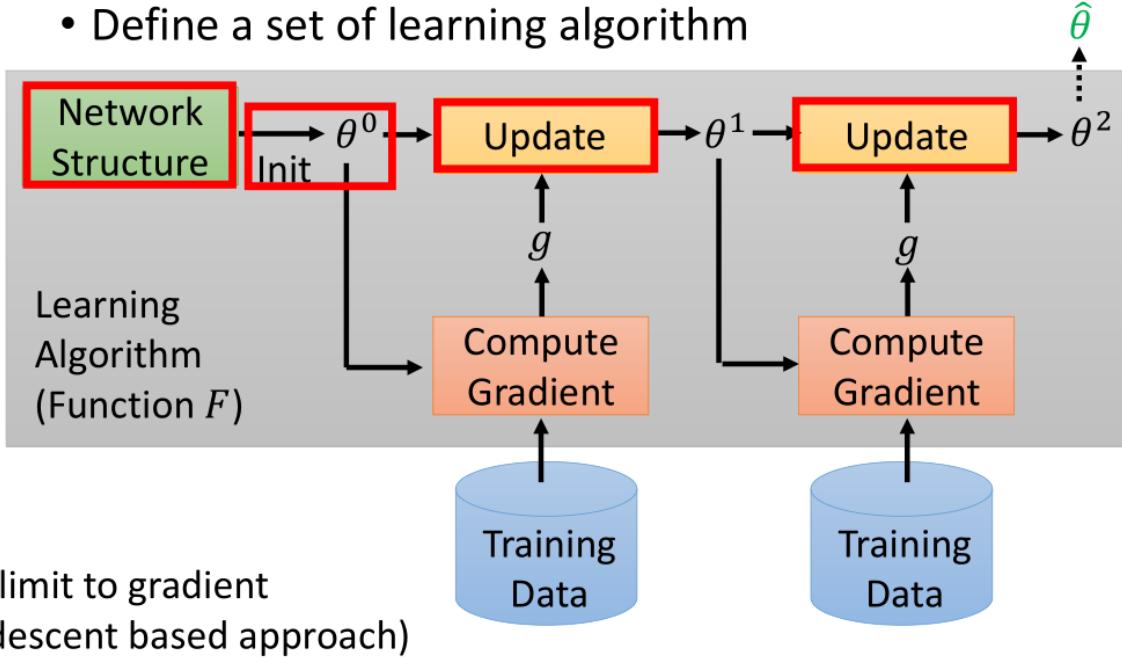
Define a set of learning algorithm

什么是一组learning algorithm 呢?

Meta Learning

Different decisions in the red boxes lead to different algorithms.
What happens in the red boxes is decided by humans until now.

- Define a set of learning algorithm



如上图所示，灰色框中的，包括网络（模型）架构，初始化参数的方法，更新参数的方法，学习率等要素构成的整个process，可以被称为一个learning algorithm。在训练的过程中有很多要素（图中的红色方框）都是人设计的，当我们选择不同的设计的时候就相当于得到了不同的learning algorithm。现在，我们考虑能不能让机器自己学出某一环节，或者全部process的设计。比如说，我们用不同的初始化方法得到不同的初始化参数以后，保持训练方法其他部分的相同，且用相同的数据来训练模型，最后都会得到不同的模型，那我们就考虑能不能让机器自己学会初始化参数，直接得到最好的一组初始化参数，用于训练。

我们就希望通过Meta Learning 学习到初始化参数这件事，好，现在我们有了一组learning algorithm，其中各个算法只有初始化参数的方法未知，是希望机器通过学习得出来的。

那现在我们怎么衡量一个learning algorithm 的好坏呢？

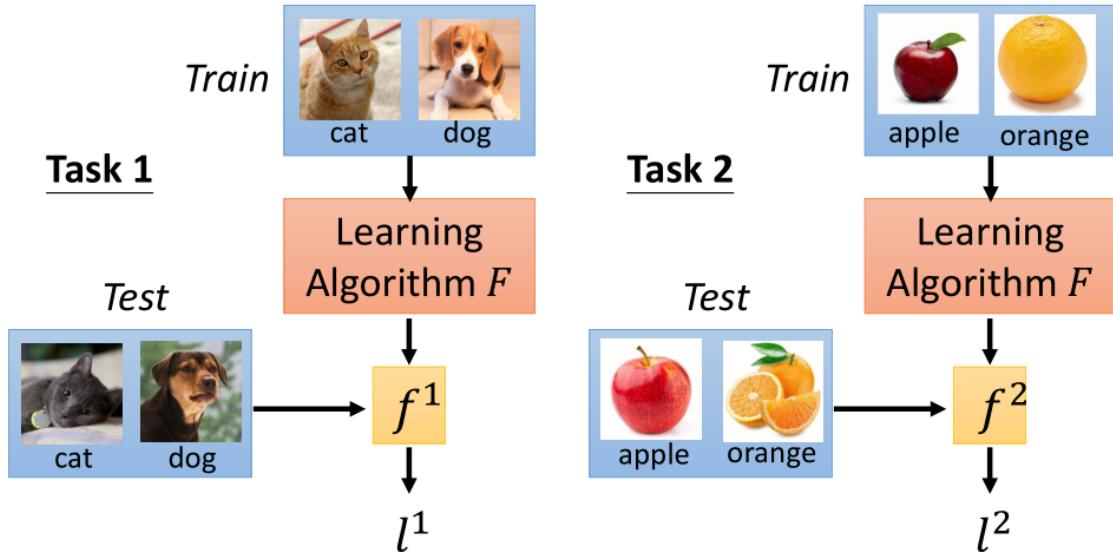
Define the goodness of a function F

Meta Learning

$$L(F) = \sum_{n=1}^N l^n$$

N tasks
Testing loss for task n
after training

- Defining the goodness of a function F



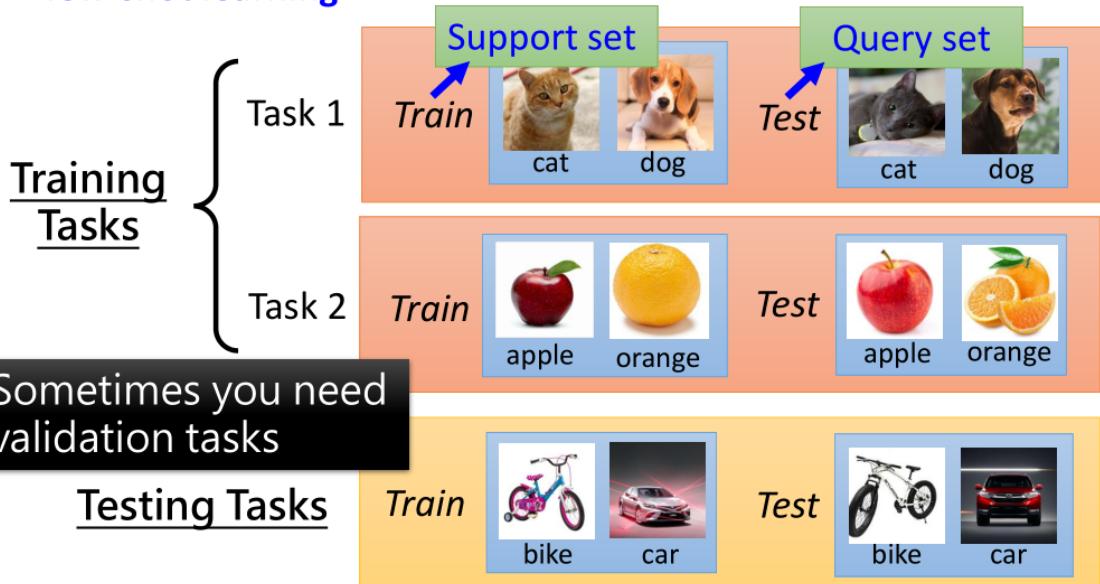
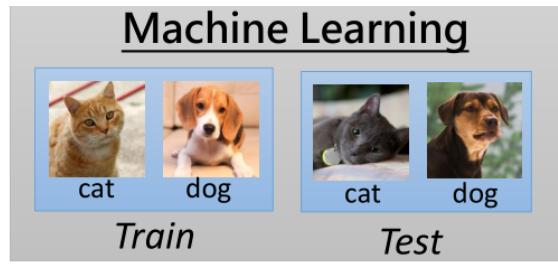
我们需要很多个task，每个task都有training set 和testing set，然后就把learning algorithm 应用到每个task上，用training set 训练，用testing set 测试，得到每一个task 的loss l^i ，对于一个learning algorithm F 的整体loss 就可以用每个task 的loss 进行求和。

$$L(F) = \sum_{n=1}^N l^n$$

从这里我们能看出，meta learning 和传统的machine learning 在训练资料上是有些不同的：

Meta Learning

Widely considered in
few-shot learning



做meta learning 的话你可能需要准备成百上千个task，每个task 都有自己的training set 和testing set 。这里为了区分，我们把meta learning的训练集叫做Training Tasks，测试集叫做Testing Tasks，其中中每个task 的训练集叫做Support set ，测试集叫做 Query set 。

Widely considered in few-shot learning，常常和few-shot learning搭配使用

讲到这里你可能觉得比较抽象，后面会讲到实际的例子，你可能就理解了meta learning 的实际运作方法。Meta learning 有很多方法，加下来会讲几个比较常见的算法，本节课会讲到一个最有名的叫做 MAML，以及MAML 的变形叫做Reptile 。

Find the best function F^*

定好了loss function 以后我们就要找一个最好的 F^* ，这个 F^* 可以使所有的training tasks 的loss 之和最小，形式化的写作下图下面的公式（具体计算方法后面再讲）：

Defining the goodness of a function F

$$L(F) = \sum_{n=1}^N l^n$$

Find the best function F^*

$$F^* = \arg \min_F L(F)$$

Testing:
Task New

Train



Learning
Algorithm F^*

Test



bike car

f^*

l

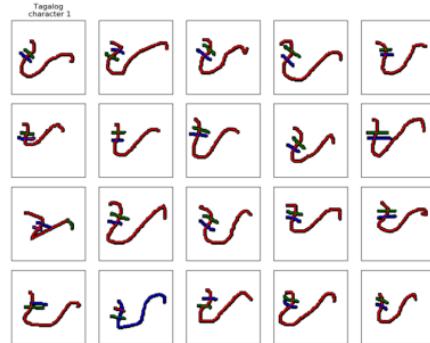
现在我们就有了meta learning 的algorithm，我们可以用testing tasks 测试这个 F^* 。把测试任务的训练集放丢入Learning Algorithm F^* ，就会得到一个 f^* ，再用测试任务的测试集去计算这个 f^* 的loss，这个loss 就是整个meta learning algorithm 的loss，来衡量这个方法的好坏。

Omniglot Corpus

Omniglot

<https://github.com/brendenlake/omniglot>

- 1623 characters
- Each has 20 examples



这是一个corpus，这里面有一大堆奇怪的符号，总共有1623个不同的符号，每个符号有20个不同的范例。上图下侧就是那些符号，右上角是一个符号的20个范例。

Few-shot Classification

Omniglot – Few-shot Classification

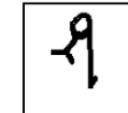
Demo of Reptile:
<https://openai.com/blog/reptile/>

- **N-ways K-shot** classification: In each training and test tasks, there are **N classes**, each has **K examples**.

20 ways
1 shot

Each character represents a class

𠂇	𠂊	𠂔	𠂎	𠂉
𠂆	𠂈	𠂋	𠂌	𠂅
𠂁	𠂄	𠂃	𠂆	𠂇
𠂇	𠂊	𠂔	𠂎	𠂉



Testing set
(Query set)

Training set
(Support set)

- 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

N-ways K-shot classification 的意思是，分N个类别，每个类别有K个样例。

所以，20 ways 1shot 就是说分20类，每类只有1个样例。这个任务的数据集就例如上图中间的20张 support set 和1张query set 。

- 把符号集分为训练符号集和测试符号集
 - 从训练符号集中随机抽N个符号，从这N个符号的范例中各随机抽K个样本，这就组成了一个训练任务training task。
 - 从测试符号集中随机抽N个符号，从这N个符号的范例中各随机抽K个样本，这就组成了一个测试任务testing task。

Techniques Today

这两个大概是最近（2019年）比较火的吧，Reptile 可以参考一下openai的这篇文章。

Reptile: A Scalable Meta-Learning Algorithm ([openai.com](https://openai.com/blog/reptile/))

MAML

- Chelsea Finn, Pieter Abbeel, and Sergey Levine, “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks”, ICML, 2017

Reptile

- Alex Nichol, Joshua Achiam, John Schulman, On First-Order Meta-Learning Algorithms, arXiv, 2018

MAML

MAML要做的就是学一个初始化的参数。过去你在做初始化参数的时候你可能要从一个distribution中sample出来，现在我们希望通过学习，机器可以自己给出一组最好的初始化参数。

MAML

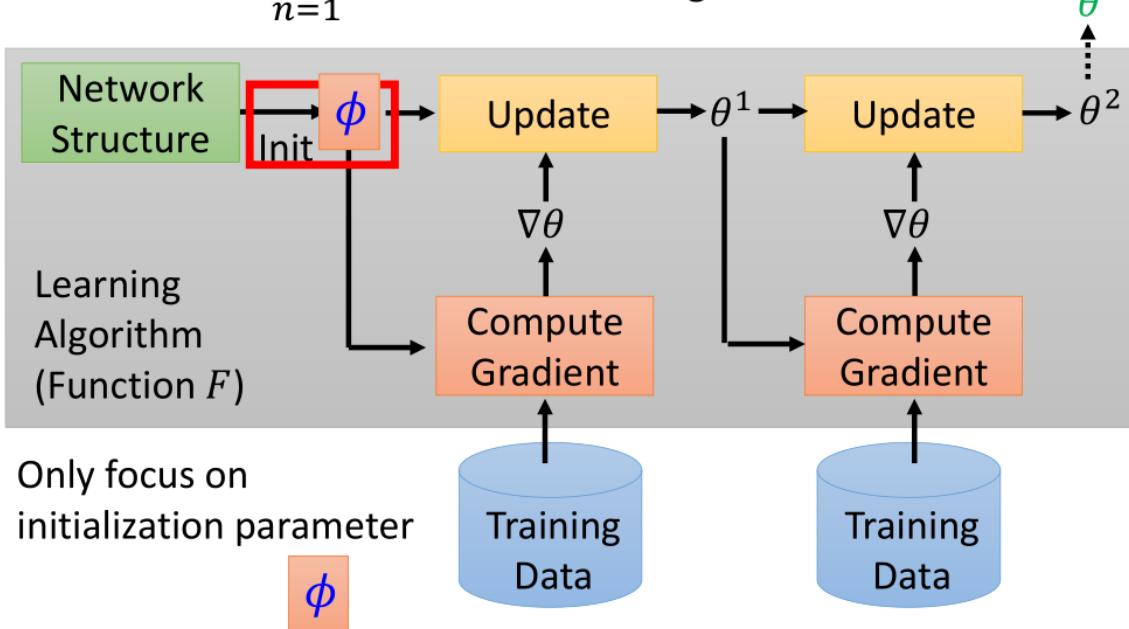
Loss Function:

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

$\hat{\theta}^n$: model learned from task n

$\hat{\theta}^n$ depends on ϕ

$l^n(\hat{\theta}^n)$: loss of task n on the testing set of task n



做法就如上图所示，我们先拿一组初始化参数 ϕ 去各个training task 做训练，在第 n 个task 上得到的最终参数记作 $\hat{\theta}^n$ ，而 $l^n(\hat{\theta}^n)$ 代表第 n 个task 在其testing set 上的loss，此时整个MAML 算法的Loss 记作：

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

这里提示一下，MAML就是属于需要所有任务的网络架构相同的meta learning algorithm，因为其中所有的function 要共用相同的初始化参数 ϕ 。

那怎么minimize $L(\phi)$ 呢？

答案就是Gradient Descent，你只要能得到 $L(\phi)$ 对 ϕ 的梯度，那就可以更新 ϕ 了，结束。

$$\phi = \phi - \eta \nabla_\phi L(\phi)$$

这里我们先假装已经会算这个梯度了，把这个梯度更新参数的思路理解就好，我们先来看一下MAML 和 Model Pre-training 在Loss Function 上的区别。

MAML v.s. Model Pre-training

MAML

Loss Function:

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

$\hat{\theta}^n$: model learned from task n

$\hat{\theta}^n$ depends on ϕ

$l^n(\hat{\theta}^n)$: loss of task n on the testing set of task n

How to minimize $L(\phi)$? Gradient Descent

$$\phi \leftarrow \phi - \eta \nabla_\phi L(\phi)$$

Model Pre-training

Widely used in transfer learning

Loss Function:

$$L(\phi) = \sum_{n=1}^N l^n(\phi)$$

通过上图我们仔细对比两个损失函数，可以看出，MAML是根据训练完的参数 $\hat{\theta}^n$ 计算损失，计算的是训练好的参数在各个task 的训练集上的损失；而预训练模型的损失函数则是根据当前初始化的参数计算损失，计算的是当前参数在要应用pre-training 的task 上的损失。

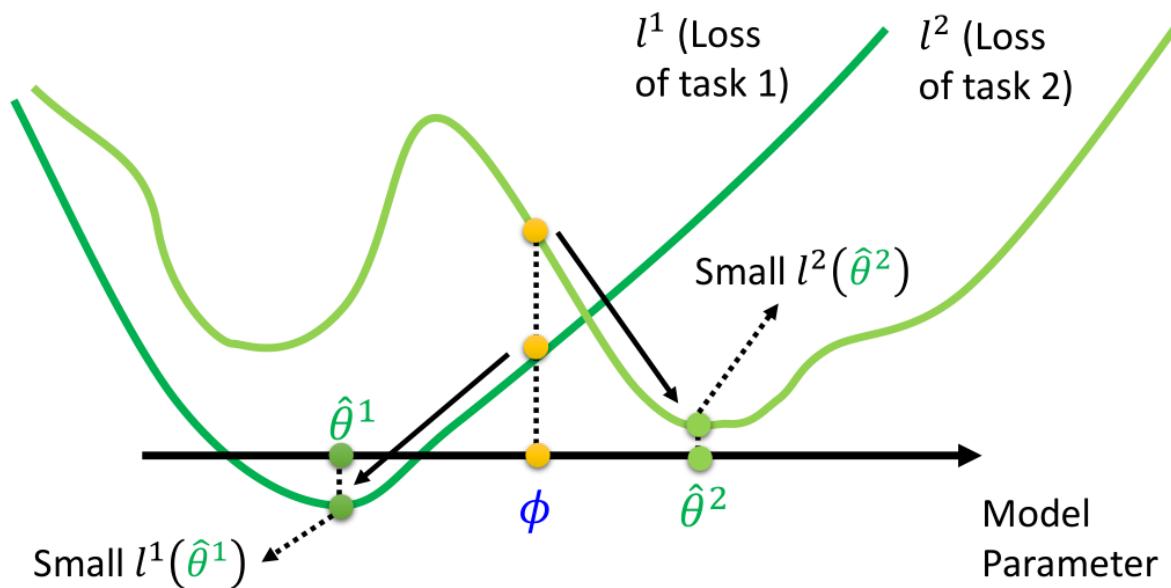
再举一个形象一点的例子：

MAML

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

我們不在意 ϕ 在 training task 上表現如何

我們在意用 ϕ 訓練出來的 $\hat{\theta}^n$ 表現如何



(横轴是模型参数，简化为一个参数，纵轴是loss)

如上图说的，我们在意的是这个初始化参数经过各个task 训练以后的参数在对应任务上的表现，也就是说如果初始参数 ϕ 在中间位置（如上图），可能这个位置根据当前参数计算的总体loss 不是最好的，但是在各个任务上经过训练以后 $\hat{\theta}$ 都能得到较低的loss (如 $\hat{\theta}^1$ 、 $\hat{\theta}^2$)，那这个初始参数 ϕ 就是好的，其 loss 就是小的。

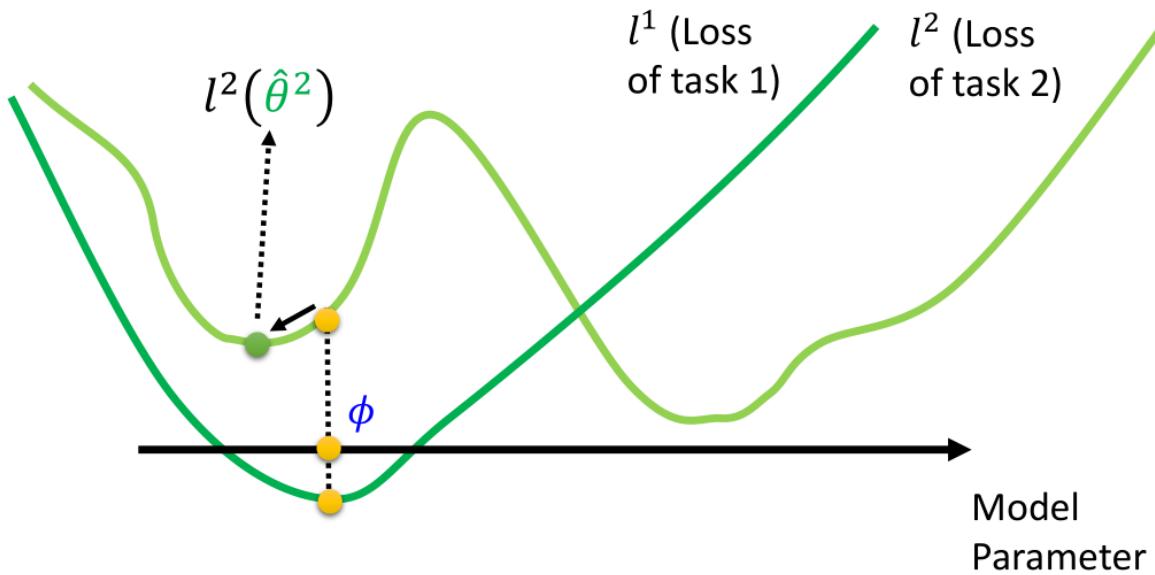
反之，在Model pre-training 上：

Model Pre-training

$$L(\phi) = \sum_{n=1}^N l^n(\phi)$$

找尋在所有 task 都最好的 ϕ

並不保證拿 ϕ 去訓練以後會得到好的 $\hat{\theta}^n$



我们希望直接通过这个 ϕ 计算出来的各个task 的loss 是最小的，所以它的取值点就可能会是上图的样子。此时在task 2上初始参数可能不能够被更新到global minima，会卡在local minima 的点 $\hat{\theta}^2$ 。

综上所述：

MAML

Loss Function:

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

$\hat{\theta}^n$: model learned from task n

$\hat{\theta}^n$ depends on ϕ

$l^n(\hat{\theta}^n)$: 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

潛力

Model Pre-training

Widely used in transfer learning

Loss Function:

$$L(\phi) = \sum_{n=1}^N l^n(\phi)$$

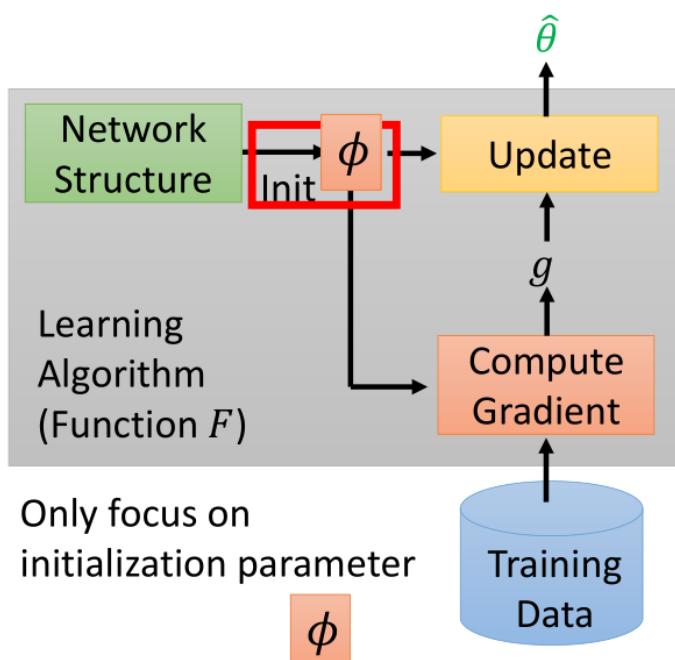
Find ϕ achieving good performance

現在表現如何

MAML 是要找一个 ϕ 在训练以后具有好的表现，注重参数的潜力，Model Pre-training 是要找一个 ϕ 在训练任务上得到好的结果，注重现在的表现。

One-Step Training

- MAML
- Fast ... Fast ... Fast ...
 - Good to truly train a model with one step. ☺
 - When using the algorithm, still update many times.
 - Few-shot learning has limited data.



$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

$$\phi \leftarrow \phi - \eta \nabla_\phi L(\phi)$$

Considering one-step training:

$$\hat{\theta} = \phi - \varepsilon \nabla_\phi l(\phi)$$

在MAML 中我们假设只update 参数一次。所以在训练阶段，你只做one-step training，参数更新公式就变成

$$\hat{\theta} = \phi - \varepsilon \nabla_{\phi} l(\phi)$$

只更新一次是有些理由的：

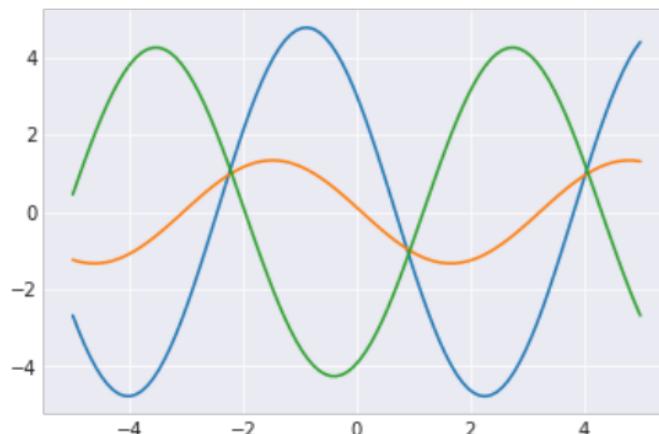
- 为了速度，多次计算梯度更新参数是比较耗时的，而且MAML 要train 很多个task
- 把只更新一次就得到好的performance作为目标，这种情况下初始化参数是好的参数
- 实际上你可以在training 的时候update 一次，在测试的时候，解testing task 的时候多update 几次，结果可能就会更好
- 如果是few-shot learning 的task，由于data 很少，update 很多次很容易overfitting

Toy Example

Each task:

- Given a target sine function $y = a \sin(x + b)$
- Sample K points from the target function
- Use the samples to estimate the target function

Sample a and b to
form a task

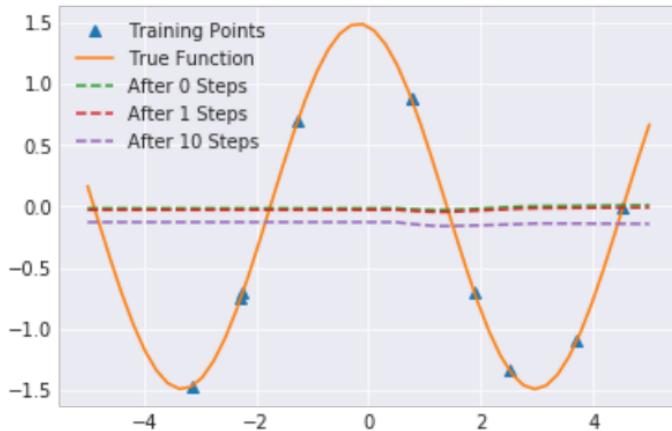


- 要拟合的目标函数是 $y = a \sin(x + b)$
- 对每个函数，也就是每个task，sample k个点
- 通过这些点做拟合

我们只要sample 不同的a, b就可以得到不同的目标函数。

来看看对比结果：

Toy Example

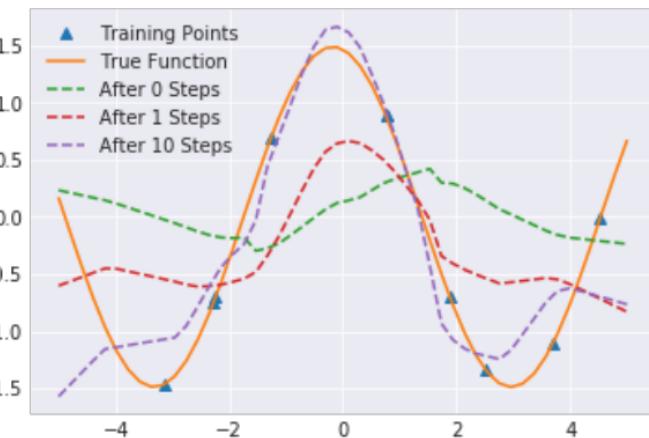


Model Pre-training

Source of images

<https://towardsdatascience.com/paper-repro-deep-metalearning-using-maml-and-reptile-fd1df1cc81b0>

MAML



可以看到，预训练模型想要让参数在所有的training task上都做好，也就是一大堆sin函数，多个task叠加起来，导致预训练模型的参数最后拟合得到的是一条直线。

MAML的结果直接用上去是绿色的线，在测试task 上training step 增加的过程中有明显的拟合效果提升。

Omniglot & Mini-ImageNet

在MAML 的原始论文中把这个技术用于Omniglot & Mini-ImageNet

	5-way Accuracy		20-way Accuracy	
	1-shot	5-shot	1-shot	5-shot
Omniglot (Lake et al., 2011)				
MANN, no conv (Santoro et al., 2016)	82.8%	94.9%	–	–
MAML, no conv (ours)	$89.7 \pm 1.1\%$	$97.5 \pm 0.6\%$	–	–
Siamese nets (Koch, 2015)	97.3%	98.4%	88.2%	97.0%
matching nets (Vinyals et al., 2016)	98.1%	98.9%	93.8%	98.5%
neural statistician (Edwards & Storkey, 2017)	98.1%	99.5%	93.2%	98.1%
memory mod. (Kaiser et al., 2017)	98.4%	99.6%	95.0%	98.6%
MAML (ours)	$98.7 \pm 0.4\%$	$99.9 \pm 0.1\%$	$95.8 \pm 0.3\%$	$98.9 \pm 0.2\%$

	5-way Accuracy	
	1-shot	5-shot
MiniImagenet (Ravi & Larochelle, 2017)		
fine-tuning baseline	$28.86 \pm 0.54\%$	$49.79 \pm 0.79\%$
nearest neighbor baseline	$41.08 \pm 0.70\%$	$51.04 \pm 0.65\%$
matching nets (Vinyals et al., 2016)	$43.56 \pm 0.84\%$	$55.31 \pm 0.73\%$
meta-learner LSTM (Ravi & Larochelle, 2017)	$43.44 \pm 0.77\%$	$60.60 \pm 0.71\%$
MAML, first order approx. (ours)	$48.07 \pm 1.75\%$	$63.15 \pm 0.91\%$
MAML (ours)	$48.70 \pm 1.84\%$	$63.11 \pm 0.92\%$

<https://arxiv.org/abs/1703.03400>

我们看上图下侧，MAML, first order approx 和 MAML 的结果很相似，那first order approx 是怎么做的呢？解释这个东西需要一点点数学：

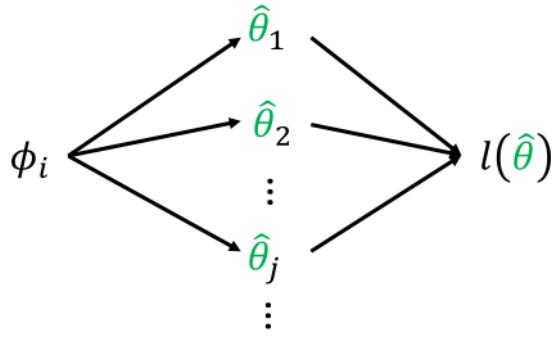
first-order approximation

$$\begin{aligned}\phi &\leftarrow \phi - \eta \nabla_{\phi} L(\phi) \\ L(\phi) &= \sum_{n=1}^N l^n(\hat{\theta}^n) \\ \hat{\theta} &= \phi - \varepsilon \nabla_{\phi} l(\phi)\end{aligned}$$

$$\nabla_{\phi} L(\phi) = \nabla_{\phi} \sum_{n=1}^N l^n(\hat{\theta}^n) = \sum_{n=1}^N \nabla_{\phi} l^n(\hat{\theta}^n)$$

$$\frac{\partial l(\hat{\theta})}{\partial \phi_i} = \sum_j \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j} \frac{\partial \hat{\theta}_j}{\partial \phi_i}$$

$$\nabla_{\phi} l(\hat{\theta}) = \begin{bmatrix} \partial l(\hat{\theta}) / \partial \phi_1 \\ \partial l(\hat{\theta}) / \partial \phi_2 \\ \vdots \\ \boxed{\partial l(\hat{\theta}) / \partial \phi_i} \\ \vdots \end{bmatrix}$$



MAML 的参数更新方法如上图左上角灰色方框所示，我们来具体看看这个 $\nabla_{\phi} L(\phi)$ 怎么算，把灰框第二条公式带入，如黄色框所示。其中 $\nabla_{\phi} l^n(\hat{\theta}^n)$ 就是左下角所示，它就是loss 对初始参数集 ϕ 的每个分量的偏微分。也就是说 ϕ_i 的变化会通过 $\hat{\theta}$ 中的每个参数 $\hat{\theta}_i$ ，影响到最终训练出来的 $\hat{\theta}$ ，所以根据chain rule 你就可以把左下角的每个偏微分写成上图中间的公式。

$$\frac{\partial l(\hat{\theta})}{\partial \phi_i} = \sum_j \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j} \frac{\partial \hat{\theta}_j}{\partial \phi_i}$$

上式中前面的项 $\frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j}$ 是容易得到的，具体的计算公式取决于你的model 的loss function，比如cross entropy 或者regression，结果的数值却决于你的训练数据的测试集。

后面的项 $\frac{\partial \hat{\theta}_j}{\partial \phi_i}$ 是需要我们算一下。可以分成两个情况来考虑：

$$\phi \leftarrow \phi - \eta \nabla_{\phi} L(\phi)$$

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

$$\hat{\theta} = \phi - \varepsilon \nabla_{\phi} l(\phi)$$

$$\nabla_{\phi} L(\phi) = \nabla_{\phi} \sum_{n=1}^N l^n(\hat{\theta}^n) = \sum_{n=1}^N \nabla_{\phi} l^n(\hat{\theta}^n)$$

$$\frac{\partial l(\hat{\theta})}{\partial \phi_i} = \sum_j \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j} \frac{\partial \hat{\theta}_j}{\partial \phi_i} \approx \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_i}$$

$$\hat{\theta}_j = \phi_j - \varepsilon \frac{\partial l(\phi)}{\partial \phi_j}$$

$$\nabla_{\phi} l(\hat{\theta}) = \begin{bmatrix} \frac{\partial l(\hat{\theta})}{\partial \phi_1} \\ \frac{\partial l(\hat{\theta})}{\partial \phi_2} \\ \vdots \\ \boxed{\frac{\partial l(\hat{\theta})}{\partial \phi_i}} \\ \vdots \end{bmatrix}$$

$i \neq j:$	$\frac{\partial \hat{\theta}_j}{\partial \phi_i} = -\varepsilon \frac{\partial l(\phi)}{\partial \phi_i \partial \phi_j} \approx 0$
$i = j:$	$\frac{\partial \hat{\theta}_j}{\partial \phi_i} = 1 - \varepsilon \frac{\partial l(\phi)}{\partial \phi_i \partial \phi_j} \approx 1$

根据灰色框中第三个式子，我们知道 $\hat{\theta}_j$ 可以用下式代替：

$$\hat{\theta}_j = \phi_j - \varepsilon \frac{\partial l(\phi)}{\partial \phi_j}$$

此时，对于 $\frac{\partial \hat{\theta}_j}{\partial \phi_i}$ 这一项，分为 $i=j$ 和 $i \neq j$ 两种情况考虑，如上图所示。在 MAML 的论文中，作者提出一个想法，不计算二次微分这一项。如果不计算二次微分，式子就变得非常简单，我们只需要考虑 $i=j$ 的情况， $i \neq j$ 时偏微分的答案总是0。

此时， $\frac{\partial l(\hat{\theta})}{\partial \phi_i}$ 就等于 $\frac{\partial l(\hat{\theta})}{\partial \theta_i}$ 。这样后一项也解决了，那就可以算出上图左下角 $\nabla_{\phi} l(\hat{\theta})$ ，就可以算出上图黄色框 $\nabla_{\phi} L(\phi)$ ，就可以根据灰色框第一条公式更新 ϕ ，结束。

$$\phi \leftarrow \phi - \eta \nabla_{\phi} L(\phi)$$

$$L(\phi) = \sum_{n=1}^N l^n(\hat{\theta}^n)$$

$$\hat{\theta} = \phi - \varepsilon \nabla_{\phi} l(\phi)$$

$$\nabla_{\phi} L(\phi) = \nabla_{\phi} \sum_{n=1}^N l^n(\hat{\theta}^n) = \sum_{n=1}^N \nabla_{\phi} l^n(\hat{\theta}^n)$$

$$\frac{\partial l(\hat{\theta})}{\partial \phi_i} = \sum_j \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j} \frac{\partial \hat{\theta}_j}{\partial \phi_i} \approx \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_i}$$

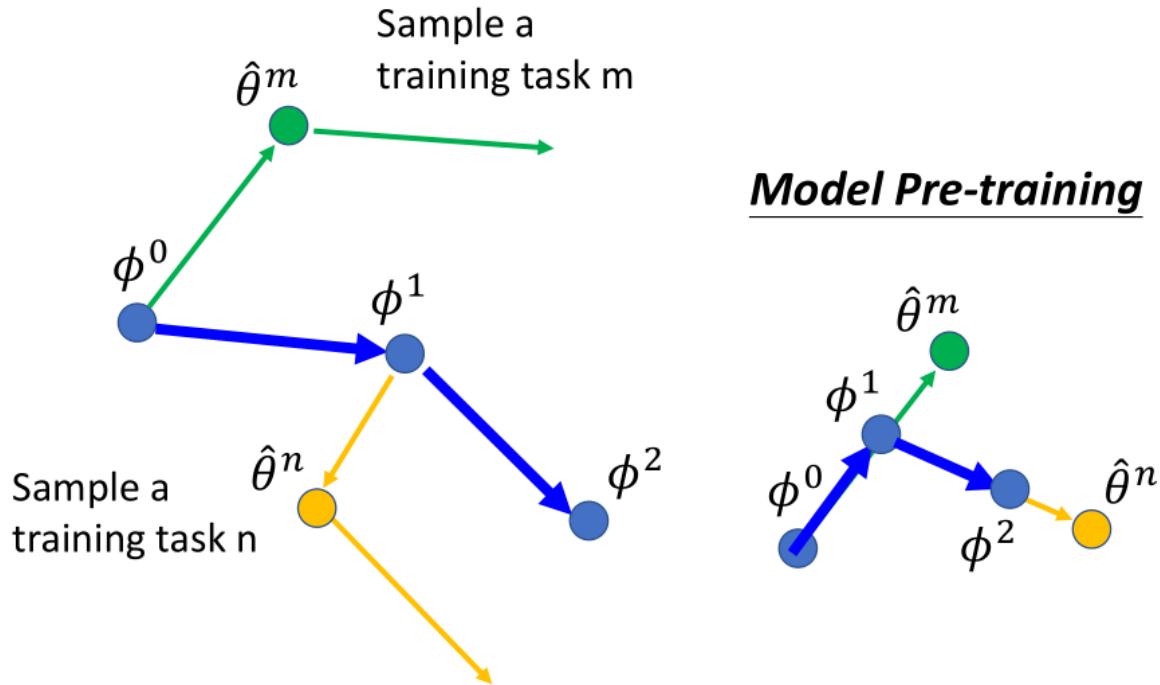
products, which is supported by standard deep learning libraries such as TensorFlow (Abadi et al., 2016). In our experiments, we also include a comparison to dropping this backward pass and using a first-order approximation, which we discuss in Section 5.2.

$$\nabla_{\phi} l(\hat{\theta}) = \begin{bmatrix} \partial l(\hat{\theta}) / \partial \phi_1 \\ \partial l(\hat{\theta}) / \partial \phi_2 \\ \vdots \\ \partial l(\hat{\theta}) / \partial \phi_i \\ \vdots \end{bmatrix} = \begin{bmatrix} \partial l(\hat{\theta}) / \partial \hat{\theta}_1 \\ \partial l(\hat{\theta}) / \partial \hat{\theta}_2 \\ \vdots \\ \partial l(\hat{\theta}) / \partial \hat{\theta}_i \\ \vdots \end{bmatrix} = \nabla_{\hat{\theta}} l(\hat{\theta})$$

在原始paper 中作者把，去掉二次微分这件事，称作using a first-order approximation。

当我们把二次微分去掉以后，上图左下角的 $\nabla_{\phi} l(\hat{\theta})$ 就变成 $\nabla_{\hat{\theta}} l(\hat{\theta})$ ，所以我们就再用 $\hat{\theta}$ 直接对 $\hat{\theta}$ 做偏微分，就变得简单很多。

Real Implementation

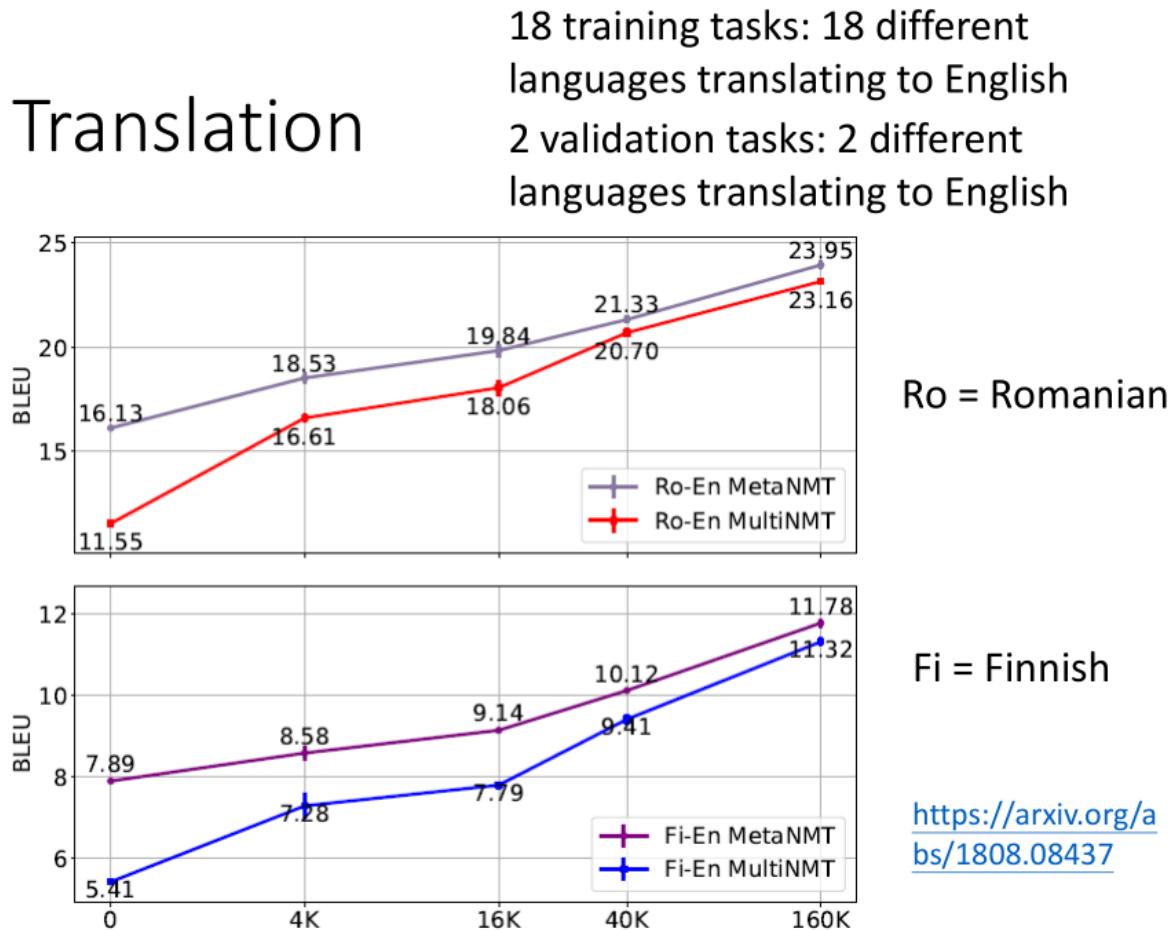


实际上，我们在MAML 中每次训练的时候会拿一个task batch 做。如上图，当我们初始化好参数 ϕ^0 我们就开始进行训练，sample出task m，完成task m训练以后，根据一次update 得到 $\hat{\theta}^m$ ，我们再计算一下 $\hat{\theta}^m$ 对它的loss function 的偏微分，也就是说我们虽然只需要update 一次参数就可以得到最好的参数，但现在我们update 两次参数， ϕ 的更新方向就和第二次更新参数的方向相同，可能大小不一样，毕竟它们的learning rate 不一样。

刚才我们讲了在精神上MAML 和Model Pre-training 的不同，现在我们来看看这两者在实际运作上的不同。如上图，预训练的参数更新完全和每个task 的gradient 的方向相同。

Translation

这里有一个把MAML 应用到机器翻译的例子：



18个不同的task：18种不同语言翻译成英文

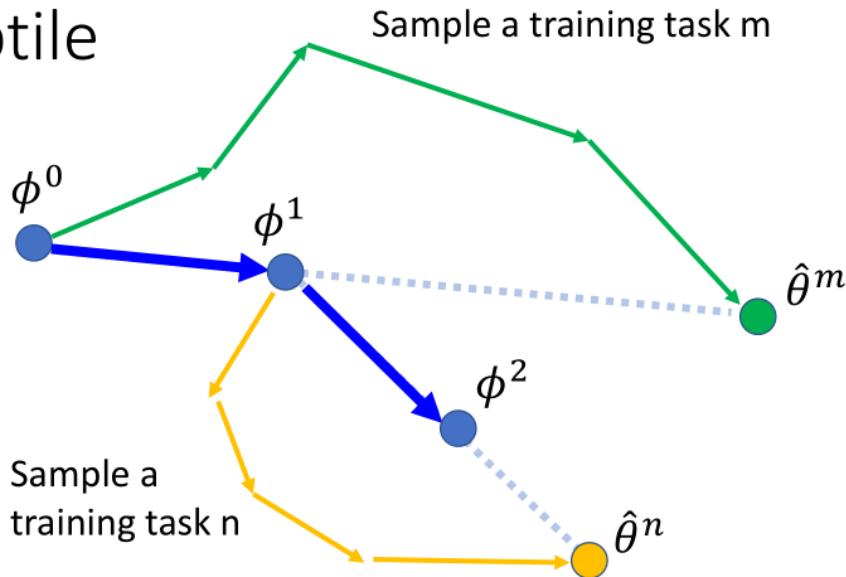
2个验证task：2种不同语言翻译成英文

Ro 是validation tasks 中的任务，Fi 即没有出现在training tasks 也没出现在validation tasks，是test的结果

横轴是每个task 中的训练资料量。MetaNMT 是MAML 的结果，MultiNMT 是 Model Pre-training 的结果，我们可以看到在所有case上面，前者都好过后者，尤其是在训练资料量少的情况下，MAML 更能发挥优势。

Reptile

Reptile



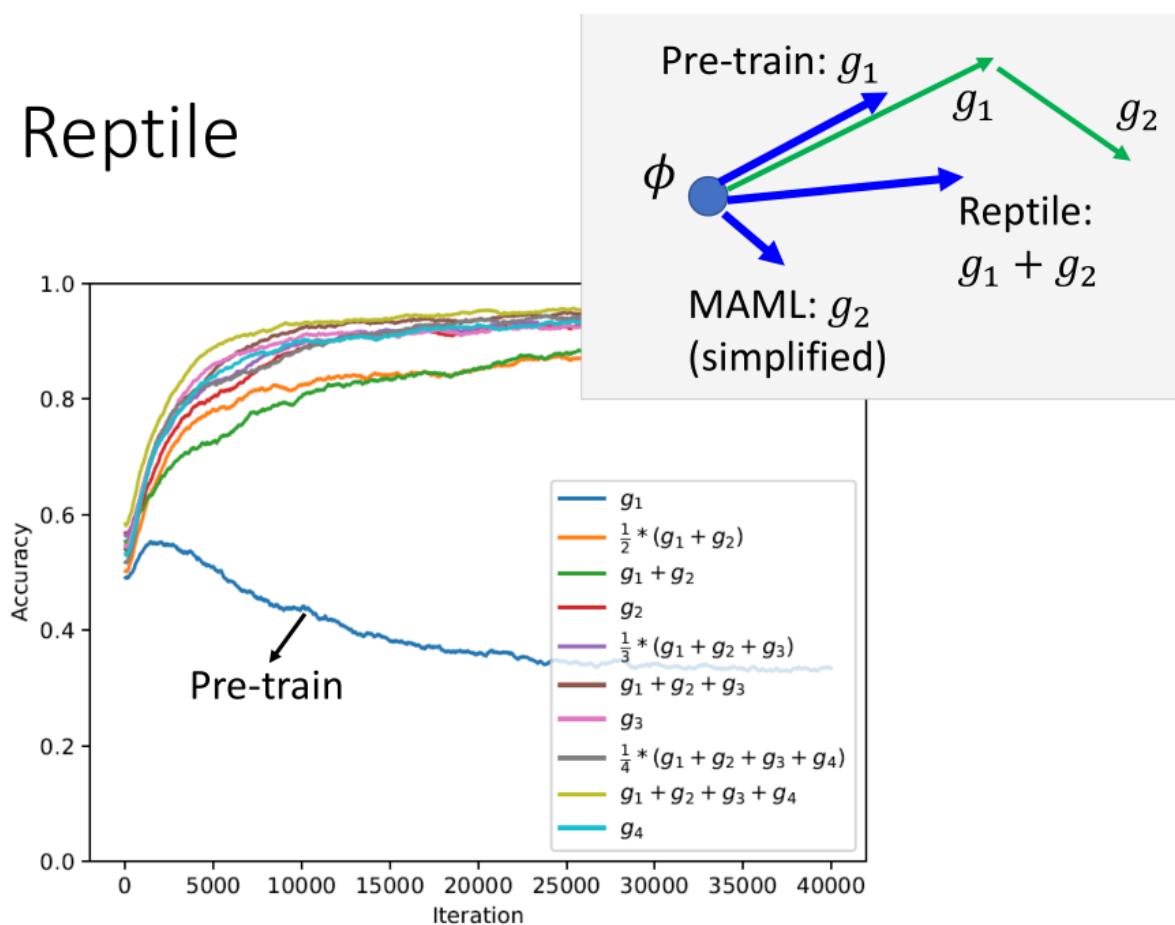
You might be thinking “isn’t this the same as training on the expected loss $\mathbb{E}_\tau [L_\tau]$?” and then checking if the date is April 1st. Indeed, if the partial minimization consists of a single gradient step, then this algorithm corresponds to minimizing the expected loss:

(this sentence is removed in the updated version)

做法就是初始化参数 ϕ_0 以后，通过在task m上训练更新参数，可以多更新几次，然后根据最后的 $\hat{\theta}^m$ 更新 ϕ_0 ，同样的继续，训练在task n以后，多更新几次参数，得到 $\hat{\theta}^n$ ，据此更新 ϕ_1 ，如此往复。

你可能会说，这不是和预训练很像吗，都是根据参数的更新来更新初始参数，希望最后的参数在所有的任务上都能得到很好的表现。作者自己也说，如上图下侧。

Reptile

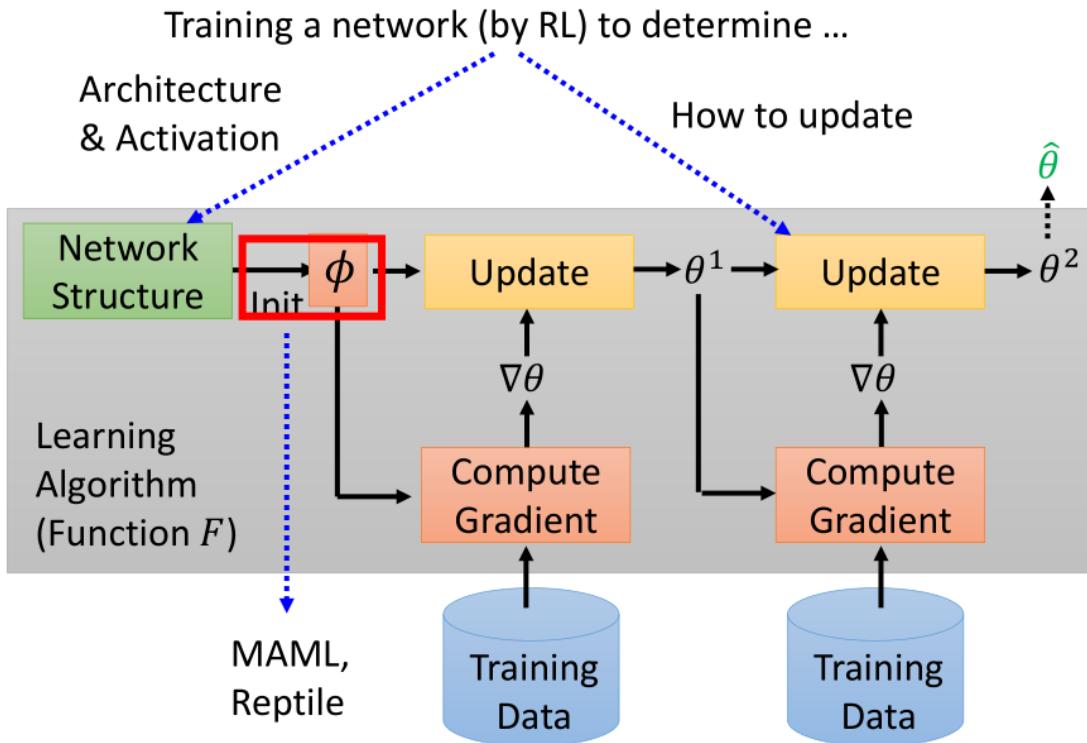


通过上图来对比三者在更新参数 ϕ 的不同，似乎Reptile 在综合两者。但是Reptile 并没有限制你只能走两步，所以如果你多更新几次参数多走几步，或许Reptile 可以考虑到另外两者没有考虑到的东西。

上图中，蓝色的特别惨的线是pre-training，所以说和预训练比起来meta learning的效果要好一些。

More...

上面所有的讨论都是在初始化参数这件事上，让机器自己学习，那有没有其他部分可以让机器学习呢，当然有很多。



比如说，学习网络结构和激活函数、学习如何更新参数.....

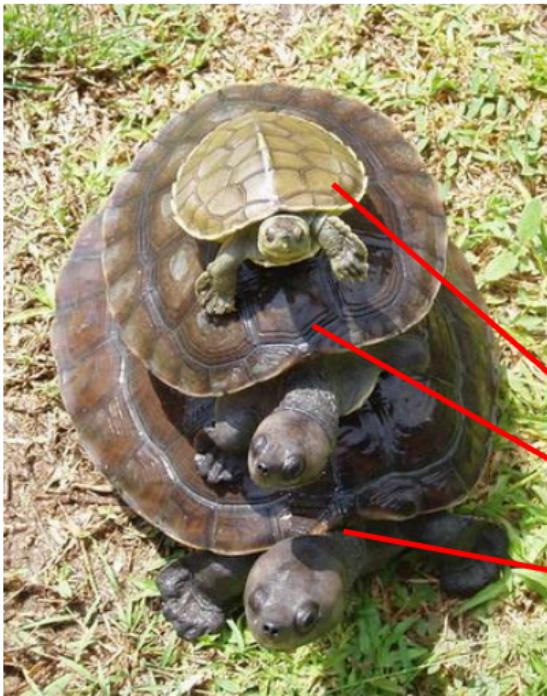
Automatically Determining Hyperparameters AutoML

Think about it...

我们使用MAML 或Reptile 来寻找最好的初始化参数，但是这个算法本身也需要初始化参数，那我们是不是也要训练一个模型找到这个模型的初始化参数.....

就好像说神话中说世界在乌龟背上，那这只乌龟应该在另一只乌龟背上.....

Turtles all the way down ?



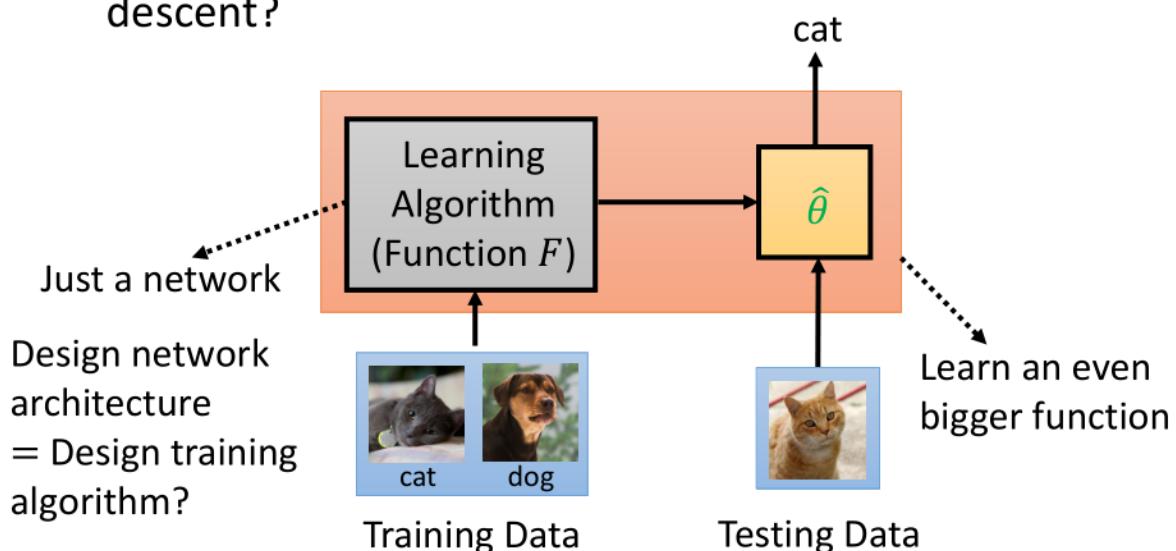
- We learn the initialization parameter ϕ by gradient descent
 - What is the initialization parameter ϕ^0 for initialization parameter ϕ ?
- Learn
- Learn to learn
- Learn to learn to learn

Crazy Idea?

传统的机器学习和深度学习的算法基本上都是gradient descent，你能不能做一个更厉害的算法，只要我们给他所有的training data 它就可以返回给我们需要model，它是不是梯度下降train 出来的不重要，它只要能给我一个能完成这个任务的model 就好。

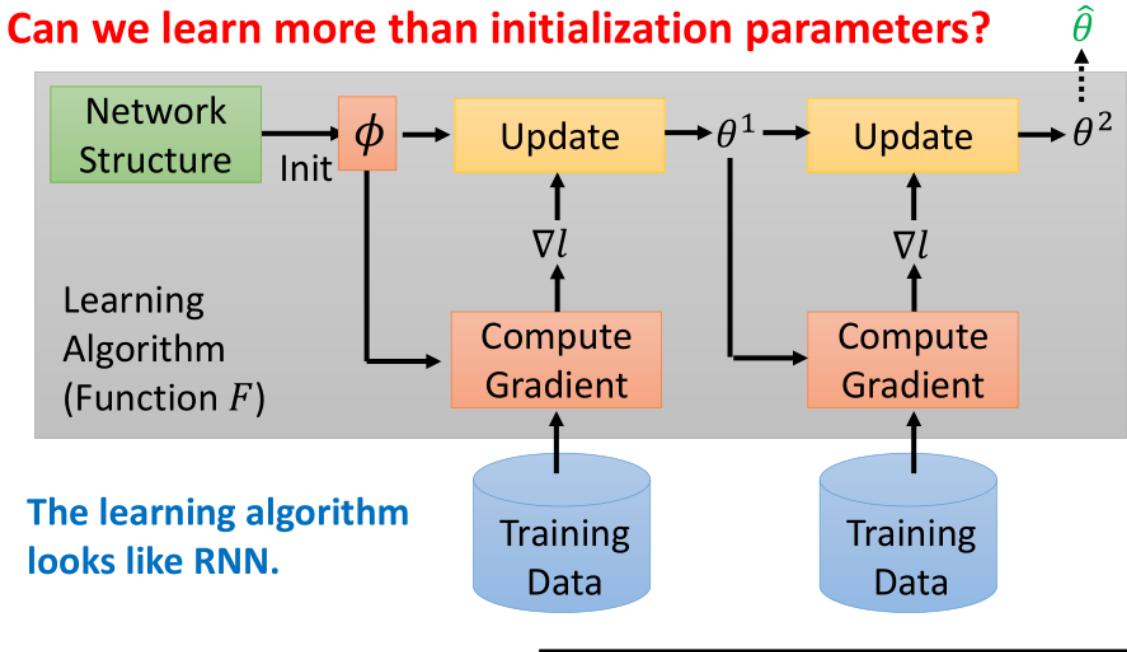
或者，反之我们最后都要应用到测试集上，那我们干脆就搞一个大黑箱，把training set 和testing set 全部丢给他，它直接返回testing data 的结果，连测试都帮你做好。这些想法能不能做到，留到下一节讲。

- How about learning algorithm beyond gradient descent?



Gradient Descent as LSTM

上节课讲了MAML 和Reptile，我们说Meta Learning就是要让机器自己learn出一个learning的 algorithm。今天我们要讲怎么把我们熟悉的learning algorithm：Gradient Descent，当作一个LSTM来看待，你直接把这个LSTM train下去，你就train出了Gradient Descent这样的Algorithm。（也就是说我现在要把学习算法，即参数的更新算法当作未知数，用Meta Learning训练出来）



OPTIMIZATION AS A MODEL FOR FEW-SHOT LEARNING

Sachin Ravi* and Hugo Larochelle

Twitter, Cambridge, USA

{sachinr, hugo}@twitter.com

Learning to learn by gradient descent by gradient descent

Marcin Andrychowicz¹, Misha Denil¹, Sergio Gómez Colmenarejo¹, Matthew W. Hoffman¹,

David Pfau¹, Tom Schaul¹, Brendan Shillingford^{1,2}, Nando de Freitas^{1,2,3}

¹Google DeepMind ²University of Oxford ³Canadian Institute for Advanced Research

上周我们讲的MAML 和Reptile都是在Initial Parameters 上做文章，用Meta Learning训练出一组好的初始化参数，现在我们希望能更进一步，通过Meta Learning训练出一个好的参数update 算法，上图黄色方块。

我们可以把整个Meta Learning 的算法看作RNN，它和RNN 有点像的，同样都是每次吃一个batch 的 data，RNN 中的memory 可以类比到Meta Learning 中的参数 θ 。

把这个Meta Learning 的算法看作RNN 的思想主要出自两篇paper：

[Optimization as a Model for Few-Shot Learning | OpenReview](#)

Sachin Ravi, Hugo Larochelle

[\[1606.04474\] Learning to learn by gradient descent by gradient descent \(arxiv.org\)](#)

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, Nando de Freitas

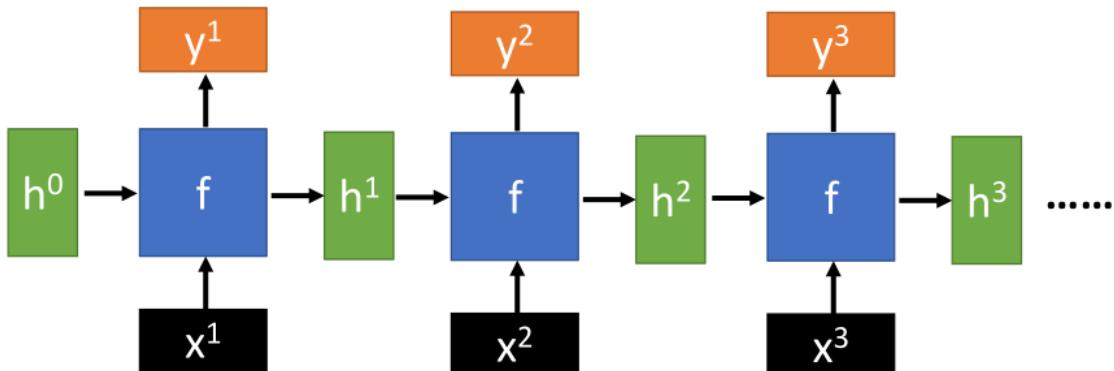
第二篇文章的题目非常有趣，也说明了此篇文章的中心：让机器学习用梯度下降学习这件事，使用的方法就是梯度下降。

Review: RNN

从与之前略微不同的角度快速回顾一下RNN。

- Given function $f: h', y = f(h, x)$

h and h' are vectors with the same dimension



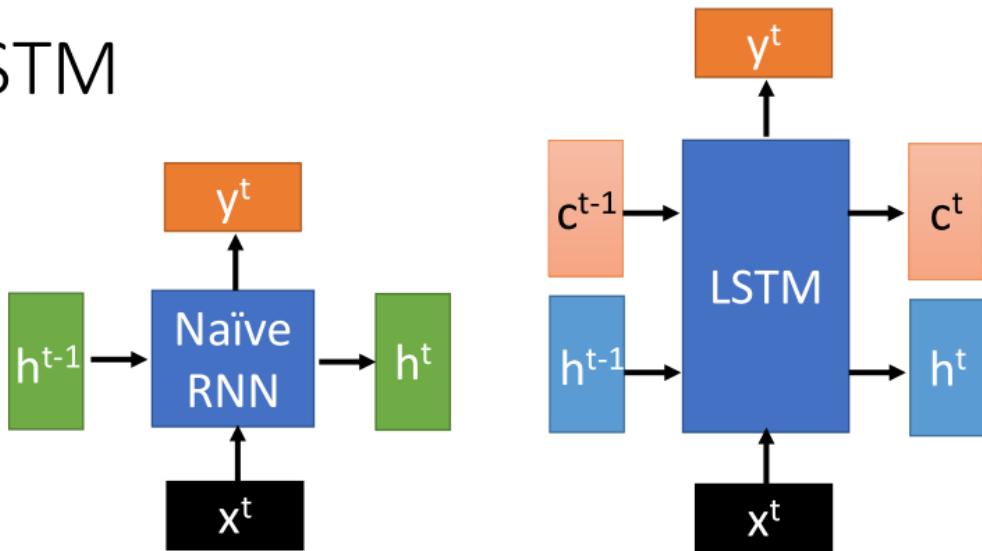
No matter how long the input/output sequence is,
we only need one function f

RNN就是一个function f , 这个函数吃 h, x 吐出 h', y , 每个step 会有一个 x (训练样本数据) 作为input, 还有一个初始的memory 的值 h_0 作为input, 这个初始参数有时候是人手动设置的, 有时候是可以让模型learn 出来的, 然后输出一个 y 和一个 h^1 。到下一个step, 它吃上一个step 得到的 h^1 和新的 x , 也是同样的输出。需要注意的是, h 的维度都是一致的, 这样同一个 f 才能吃前一个step 得到 h 。这个过程不断重复, 就是RNN。

所以, 无论多长的input/output sequence 我们只需要一个函数 f 就可以运算, 无论你的输入再怎么多, 模型的参数量不会变化, 这就是RNN 厉害的地方, 所以它特别擅长处理input 是一个sequence 的状态。(比如说自然语言处理中input 是一个长句子, 用word vector 组成的很长的sequence)

我们如今用的一般都是RNN 的变形LSTM, 而且我们现在说使用RNN 基本上就是在指使用LSTM 的技术。那LSTM 相比于RNN 有什么特别的地方呢。

LSTM



c change slowly $\rightarrow c^t$ is c^{t-1} added by something

h change faster $\rightarrow h^t$ and h^{t-1} can be very different

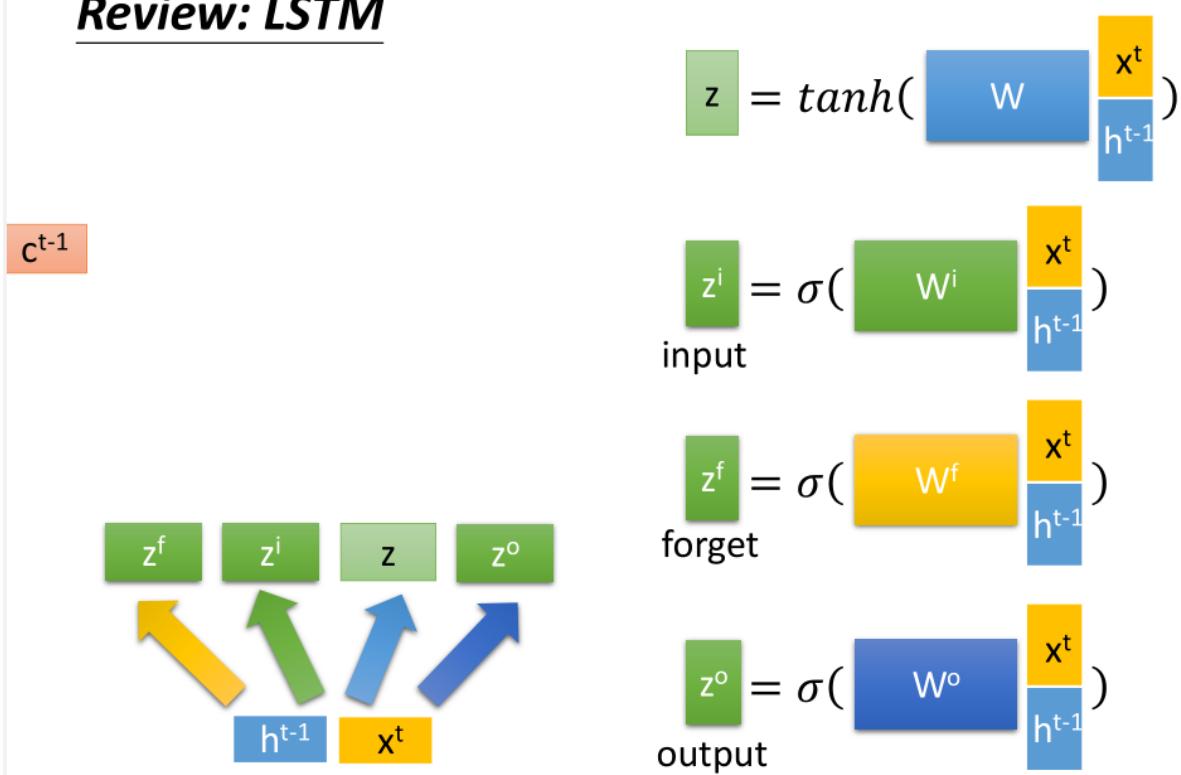
如上图, LSTM (右) 相比于RNN , 把input 的h 拆解成两部分, 一部分仍然叫做 h , 一部分我们叫做 c 。为什么要这样分呢, 你可以想象是因为 c 和 h 扮演了不同的角色。

- c 变化较慢, 通常就是把某个向量加到上一个 c^{t-1} 上就得到了新的 c^t , 这个 c^t 就是LSTM 中 memory cell 存储的值, 由于这个值变化很慢, 所以LSTM 可以记住时间比较久的数据
- h 变化较快, h^{t-1} 和 h^t 的变化是很大的

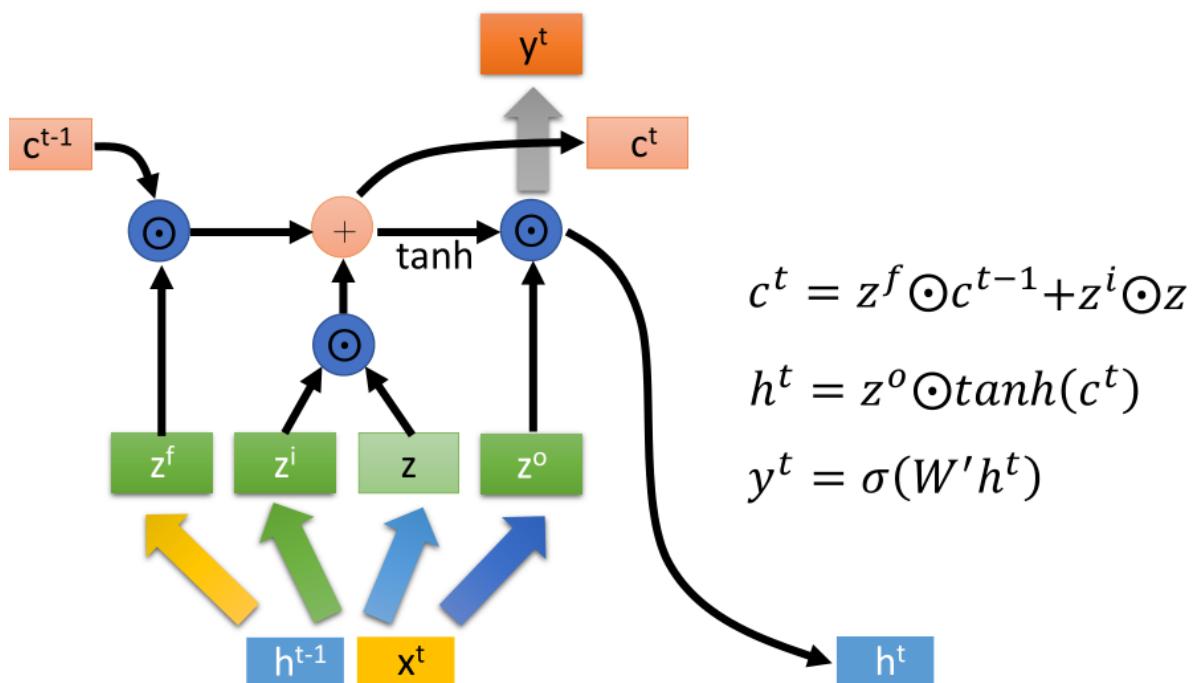
Review: LSTM

我们接下来看看LSTM 的做法和结构:

Review: LSTM



c^{t-1} 是 memory 记忆单元，把 x 和 h 拼在一起乘上一个权重矩阵 W ，再通过一个 \tanh 函数得到 input z ， z 是一个向量。同样的 x 和 h 拼接后乘上对应的权重矩阵得到对应向量 input gate z^i ，forget gate z^f ，output gate z^o ，接下来：



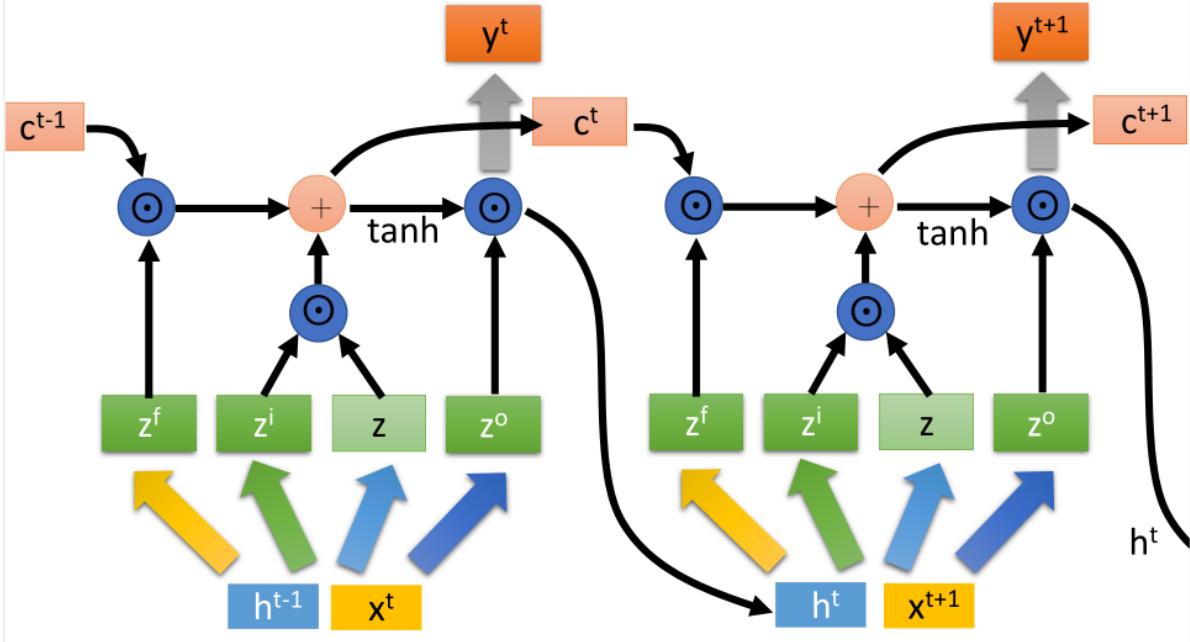
$z^f \cdot c^{t-1}$ 决定是否保留上个 memory， $z^i \cdot z$ 决定是否把现在的 input 存到 memory；

通过 $z^o \cdot \tanh(c^t)$ 得到新的 h^t ；

W' 乘上新的 h^t ，再通过一个 sigmoid function 得到当前 step 的 output y^t ；

重复上述步骤，就是 LSTM 的运作方式：

Review: LSTM



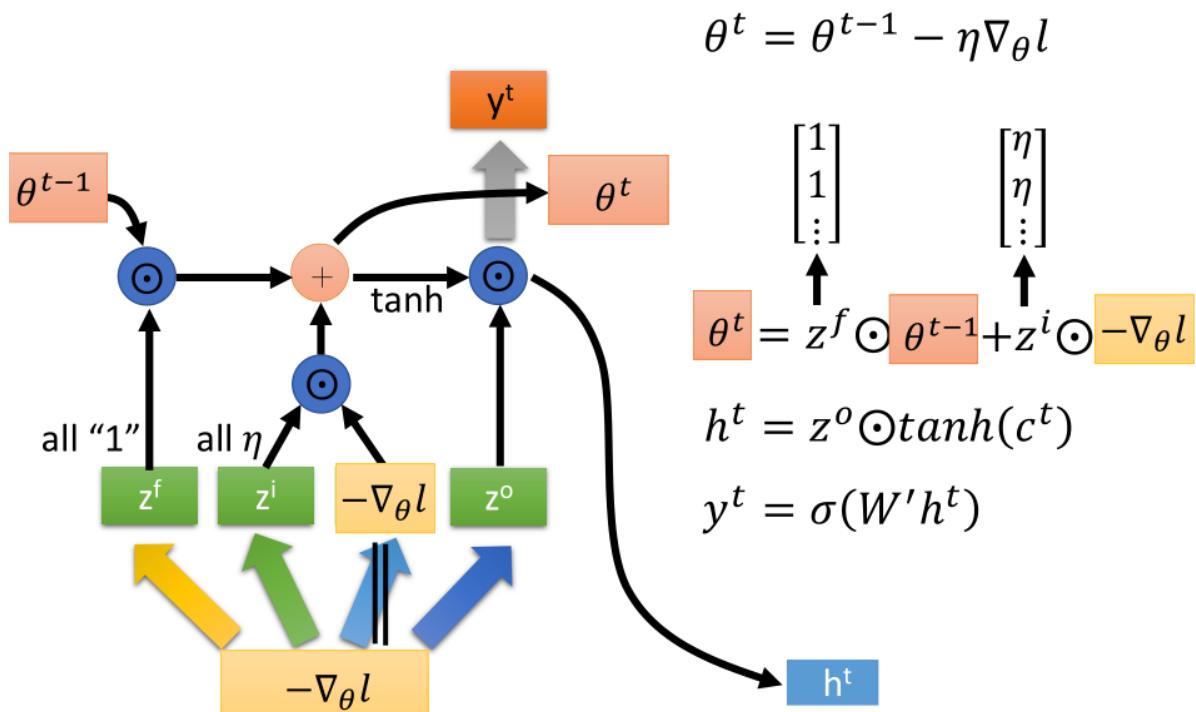
好，讲了这么多，它和Gradient Descent 到底有什么样的关系呢？

Similar to gradient descent based algorithm

我们把梯度下降参数 θ 更新公式和LSTM 的memory c 更新公式都列出来，如下：

$$\begin{aligned}\theta^t &= \theta^{t-1} - \eta \nabla_{\theta} l \\ c^t &= z^f \odot c^{t-1} + z^i \odot z \\ h^t &= z^o \odot \tanh(c^t) \\ y^t &= \sigma(W'h^t)\end{aligned}$$

我们知道在gradient descent 中我们在每个step 中，把旧的参数减去，learning rate 乘梯度，作为更新后的新参数，此式和LSTM 中memory 单元 c 有些相似，我们就把 c 替换成 θ ：



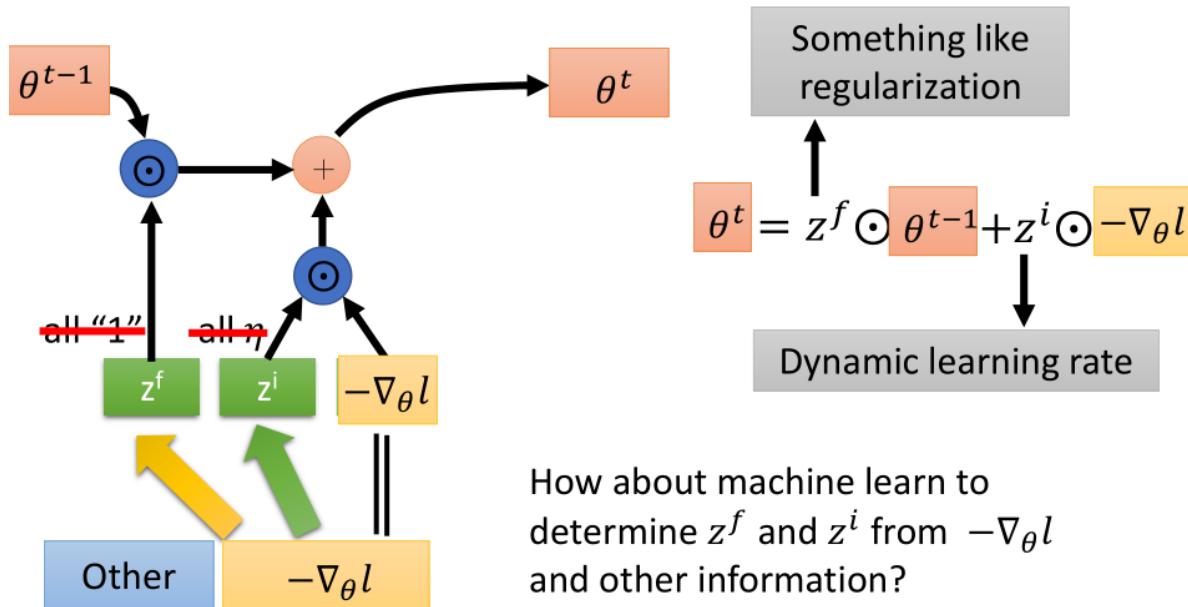
接下来我们再做一些变换。输入 h^{t-1} 来自上一个step, x^t 来自外界输入, 我们就把 $h^{t-1} x^t$ 换成 $-\nabla_\theta l$ 。然后我们假设从input到 z 的公式中乘的matrix是单位矩阵, 所以 z 就等于 $-\nabla_\theta l$ 。再然后, 我们把 z^f 定为全1的列向量, z^i 定位全为learning rate的列向量, 此时LSTM的memory c 的更新公式变得和Gradient Descent一样。

所以你可以说Gradient Descent就是LSTM的简化版, LSTM中input gate和forget gate是通过机器学出来的, 而在梯度下降中input gate和forget gate都是人设的, input gate永远都是学习率, forget gate永远都是不可以忘记。

现在, 我们考虑能不能让机器自己学习gradient descent中的input gate和forget gate呢?

另外, input的部分刚才假设只有gradient的值, 实作上可以拿更多其他的数据作为input, 比如常见的做法, 可以把 $c^{t-1}(\theta^{t-1})$ 在这个step算出来的loss作为输入来control这个LSTM的input gate和forget gate的值。

$$\theta^t = \theta^{t-1} - \eta \nabla_\theta l$$



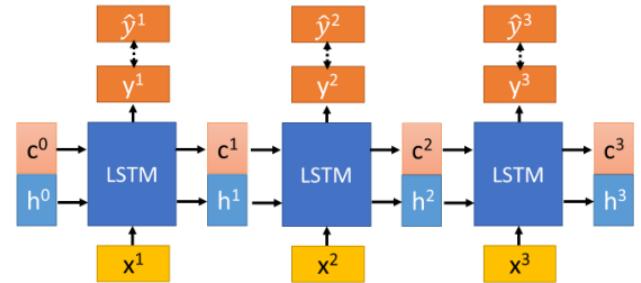
如果们可以让机器自动的学input gate和forget gate的值意味着什么? 意味着我们可以拥有动态的learning rate, 每一个dimension在每一个step的learning rate都是不一样的而不是一个不变的值。

而 z^f 就像一个正则项, 它做的事情是把前一个step算出来的参数缩小。我们以前做的L2 regularization又叫做Weight Decay, 为什么叫Weight Decay, 因为如果你把update的式子拿出来看, 每个step都会把原来的参数稍微变小, 现在这个 z^f 就扮演了像是Weight Decay的角色。但是我们现在不是直接告诉机器要做多少Weight Decay, 而是要让机器学出来, 它应该做多少Weight Decay。

LSTM for Gradient Descent

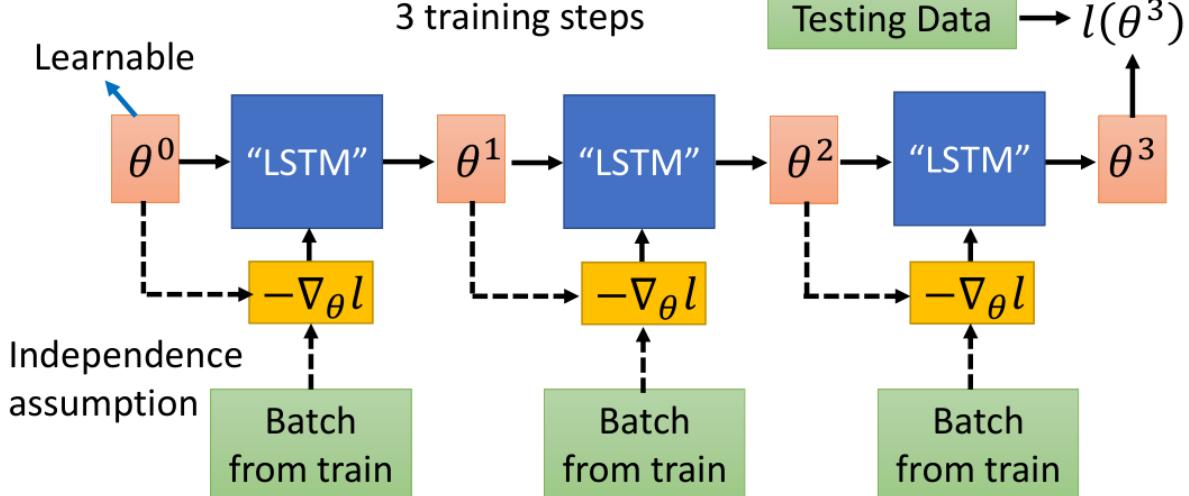
我们来看看一般的LSTM和for Gradient Descent的LSTM:

Typical LSTM



LSTM for Gradient Descent

$$\theta^t = z^f \odot \theta^{t-1} + z^i \odot -\nabla_{\theta} l$$



Typical LSTM 就是input x , output c 和 h , 每个step 会output 一个 y , 希望 y 和label 越接近越好。

Gradient Descent 的LSTM是这样：我们先sample 一个初始参数 θ ，然后sample 一个batch 的data，根据这一组data 算出一个gradient $\nabla_{\theta} l$ ，把负的gradient input 到LSTM 中进行训练，这个LSTM 的参数过去是人设死的，我们现在让参数在Meta Learning 的架构下被learn 出来。上述的这个update 参数的公式就是：

$$\theta^t = z^f \cdot \theta^{t-1} + z^i \cdot -\nabla_{\theta} l$$

z^f z^i 以前是人设死的，现在LSTM 可以自动把它学出来。

现在就可以output 新的参数 θ^1 ，接着就是做一样的事情：再sample 一组数据，算出梯度作为新的input，放到LSTM 中就得到output θ^2 ，以此类推，不断重复这个步骤。最后得到一组参数 θ^3 （这里假设只update 3次，实际上要update 更多次），拿这组参数去应用到Testing data 上算一下loss： $l(\theta^3)$ ，这个loss 就是我们要minimize 的目标，然后你就要用gradient descent 调LSTM 的参数，去minimize 最后的loss 。

这里有一些需要注意的地方。在一般的LSTM 中 c 和 x 是独立的，LSTM 的memory 存储的值不会影响到下一次的输入，但是Gradient Descent LSTM 中参数 θ 会影响到下一个step 中算出的gradient 的值，如上图虚线所示。

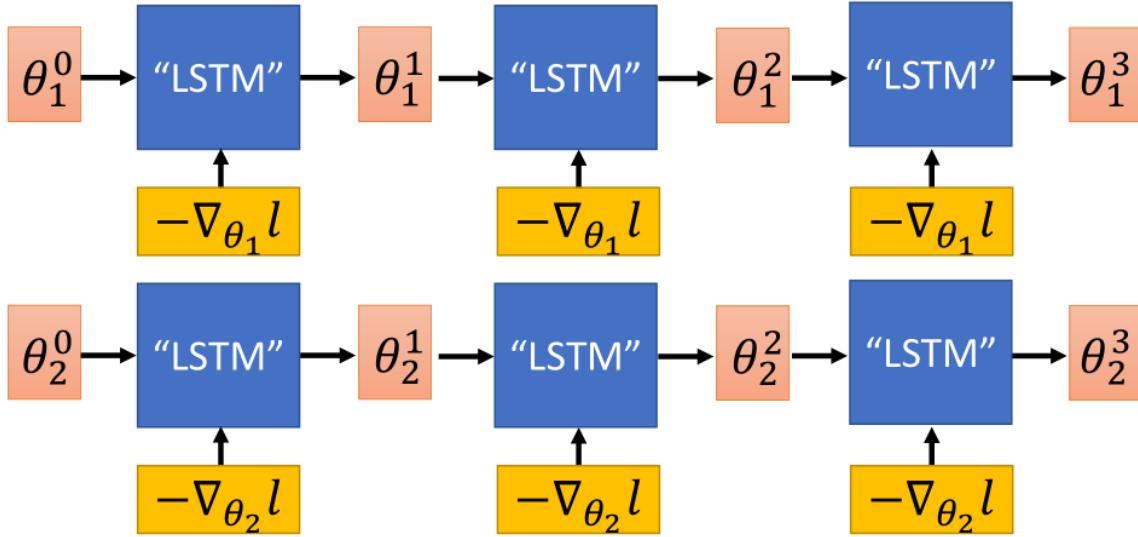
所以说在Gradient Descent LSTM 中现在的参数会影响到未来看到的梯度。所以当你做back propagation 的时候，理论上你的error signal 除了走实线的一条路，它还可以走 θ 到 $-\nabla_{\theta} l$ 虚线这一条路，可以通过gradient 这条路更新参数。但是这样做会很麻烦，和一般的LSTM 不太一样了，一般的LSTM c 和 x 是没有关系的，现在这里确实有关系，为了让它和一般的LSTM 更像，为了少改一些code，我们就假设没有虚线那条路，结束。现在的文献上其实也是这么做的。

另外，在LSTM input 的地方memory 中的初始值 θ_0 可以通过训练直接被learn 出来，所以在LSTM中也可以做到和MAML相同的事，可以把初始的参数跟着LSTM一起学出来。

Real Implementation

LSTM 的memory 就是要训练的network 的参数，这些参数动辄就是十万百万级别的，难道要开十万百万个cell 吗？平常我们开上千个cell 就会train 很久，所以这样是train不起来的。在实际的实现上，我们做了一个非常大的简化：我们所learn 的LSTM 只有一个cell 而已，它只处理一个参数，所有的参数都共用一个LSTM。所以就算你有百万个参数，都是使用这同一个LSTM 来处理。

The LSTM used only has one cell. Share across all parameters



- Reasonable model size
- In typical gradient descent, all the parameters use the same update rule
- Training and testing model architectures can be different.

也就是说如上图所示，现在你learn 好一个LSTM以后，它是直接被用在所有的参数上，虽然这个LSTM 一次只处理一个参数，但是同样的LSTM 被用在所有的参数上。 θ^1 使用的LSTM 和 θ^2 使用的LSTM 是同一个处理方式也相同。那你可能会说， θ^1 和 θ^2 用的处理方式一样，会不会算出同样的值呢？会不，因为他们的初始参数是不同的，而且他们的gradient 也是不一样的。在初始参数和算出来的gradient 不同的情况下，就算你用的LSTM的参数是一样的，就是说你update 参数的规则是一样的，最终算出来的也是不一样的 θ^3 。

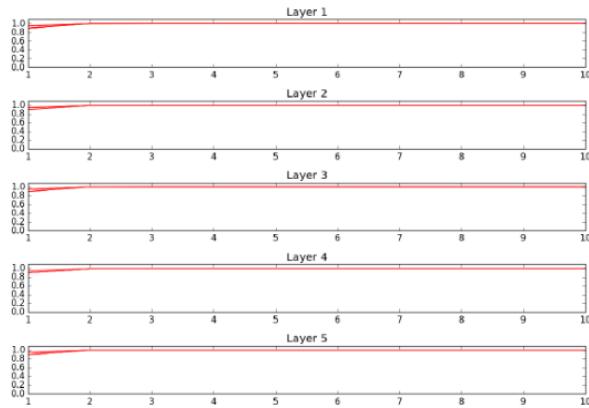
这就是实作上真正implement LSTM Gradient Descent 的方法。

这么做有什么好处：

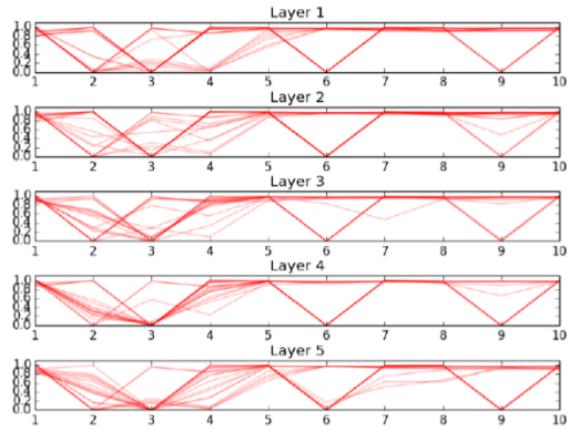
- 在模型规模上问题上比较容易实现
- 在经典的gradient descent 中，所有的参数也都是使用相同的规则，所以这里使用相同的LSTM，就是使用相同的更新规则是合理的
- 训练和测试的模型架构可以是不一样的，而之前讲的MAML 需要保证训练任务和测试任务使用的model architecture 相同

Experimental Results

$$\theta^t = z^f \odot \theta^{t-1} + z^i \odot -\nabla_{\theta} l$$



(a) Forget gate values for 1-shot meta-learner



(b) Input gate values for 1-shot meta-learner

我们来看一个文献上的实验结果，这是做在few-shot learning 的task上。横轴是update 的次数，每次train 会update 10次，左侧是forget gate z^f 的变化，不同的红线就是不同的task 中forget gate 的变化，可以看出 z^f 的值多数时候都保持在1附近，也就是说LSTM 有learn到 θ^{t-1} 是很重要的东西，没事就不要忘掉，只做一个小小的weight decay，这和我们做regularization 时候的思想相同，只做一个小小的weight decay 防止overfitting。

右侧是input gate z^i 的变化，红线是不同的task，可以看出它的变化有点复杂，但是至少我们知道，它不是一成不变的固定值，它是有学到一些东西的，是动态变化的，放到经典梯度下降中来说就是learning rate 是动态变化的。

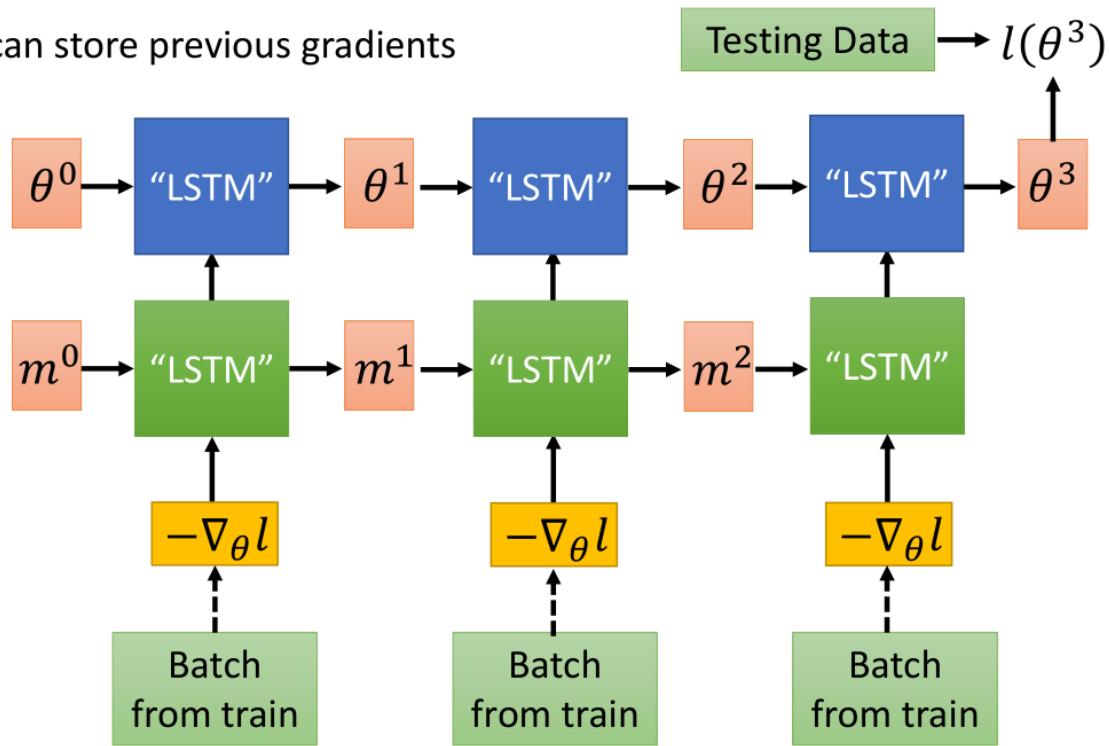
LSTM for Gradient Descent (v2)

只有刚才的架构还不够，我们还可以更进一步。想想看，过去我们在用经典梯度下降更新参数的时候我们不仅会考虑当前step 的梯度，我们还会考虑过去的梯度，比如RMSProp、Momentum 等。

在刚才的架构中，我们没有让机器去记住过去的gradient，所以我们可以做更进一步的延伸。我们在过去的架构上再加一层LSTM，如下图所示：

3 training steps

m can store previous gradients



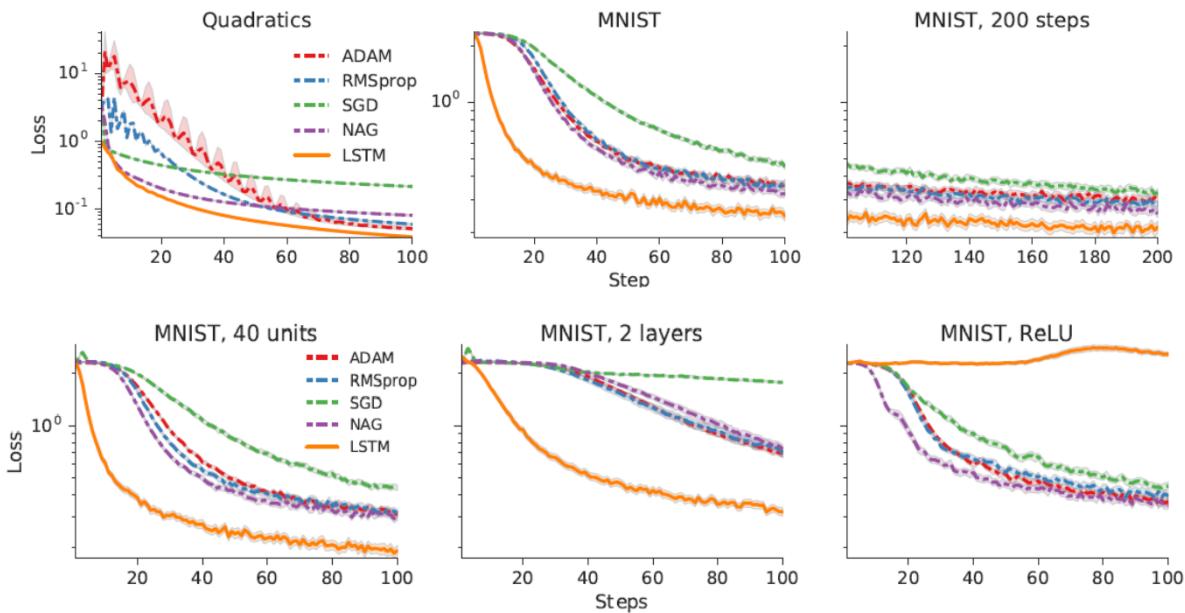
蓝色的一层LSTM 是原先的算learning rate、做weight decay 的LSTM，我们再加入一层LSTM，让算出来的gradient $-\nabla_{\theta} l$ 先通过这个LSTM，把这个LSTM 吐出来的东西input 到原先的LSTM 中，我们希望绿色的这一层能做到记住以前算过的gradient 这件事。这样，可能就可以做到Momentum 可以做的的事情。

上述的这个方法，是老师自己想象的，在learning to learn by gradient descent by gradient descent 这篇paper 中上图中蓝色的LSTM 使用的是一般的梯度下降算法，而在另一篇paper 中只有上面没有下面，而老师觉得这样结合起来才是合理的能考虑过去的gradient 的gradient descent 算法的完全体。

Experimental Result 2

learning to learn by gradient descent by gradient descent 这篇paper 的实验结果。

Experimental Results



<https://arxiv.org/abs/1606.04474>

第一个实验图，是做在toy example 上，它可以制造一大堆训练任务，然后测试在测试任务上，然后发现，LSTM 来当作gradient descent 的方法要好过人设计的梯度下降方法。

其他图中这个实验是训练任务测试任务都是MNIST。虽然训练和测试任务都是相同的dataset也是相同的，但是train 和test 的时候network 的架构是不一样的。在train 的时候network 是只有一层，只有20个neuron。

第四张图是上述改变network 架构后在testing 的结果，testing 的时候network 只有一层该层40个neuron。从图上看还是做的起来，而且比一般的gradient descent 方法要好很多。

第五张图是上述改变network 架构后在testing 的结果，testing 的时候network 有两层。从图上看还是做的起来，而且比一般的gradient descent 方法要好很多。

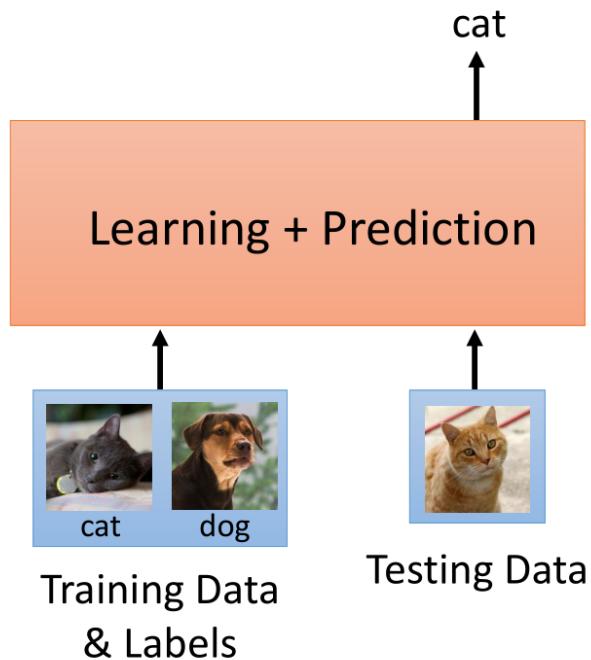
第六张图是上述改变network 激活函数后在testing 的结果，training 的时候激活函数是sigmoid 而testing 的时候改成ReLU。从图上看做不起来，崩掉了，training 和testing 的network 的激活函数不一样时候，LSTM 没办法跨model 应用。

Metric-based

加下来我们就要实践我们之前提到的疯狂的想法：直接学一个function，输入训练数据和对应的标签，以及测试数据，直接输出测试数据的预测结果。也就是说这个模型把训练和预测一起做了。

虽然这个想法听起很crazy，但是实际上现实生活中有在使用这样的技术，举例来说：手机的人脸验证

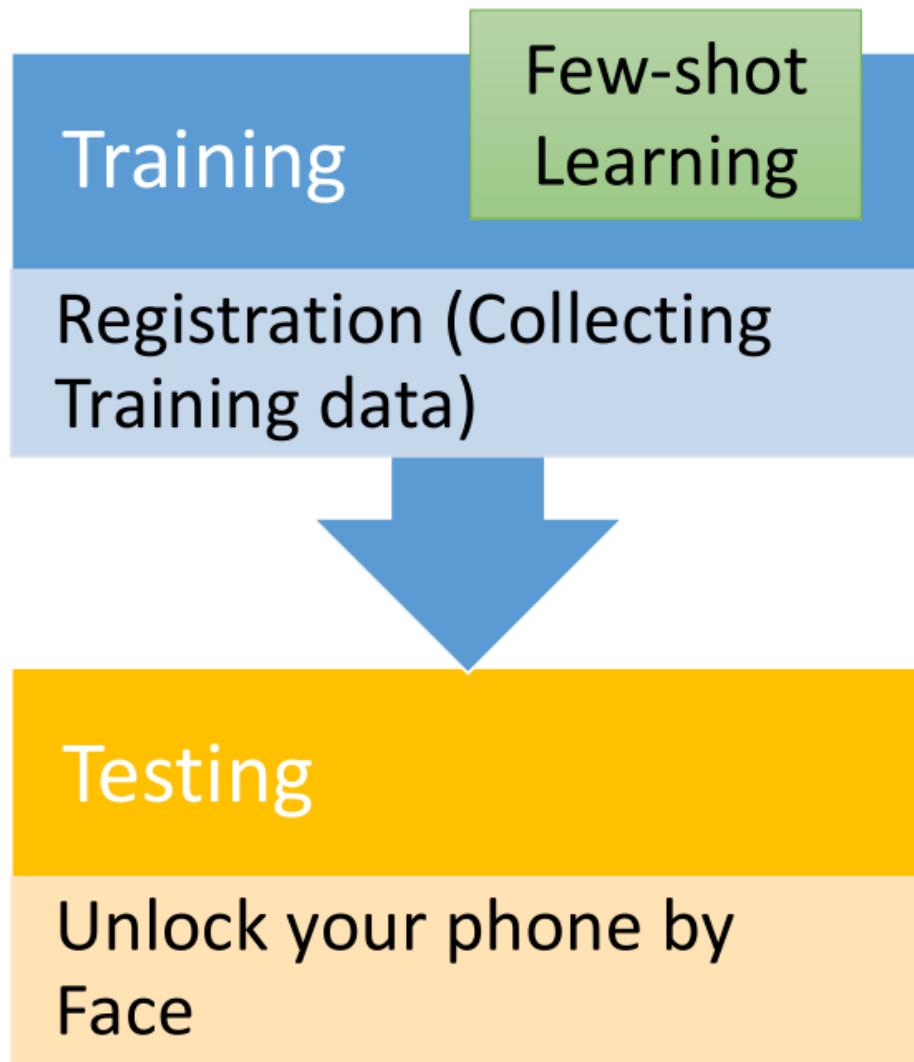
- Input:
 - Training data and their labels
 - Testing data
- Output:
 - Predicted label of testing data



我们在使用手机人脸解锁的时候需要录制人脸信息，这个过程中我们转动头部，就是手机在收集资料，收集到的资料就是作为few-shot learning 的训练资料。另外，语音解锁Speaker Verification 也是一样的技术，只要换一下输入资料和network 的架构。

Face Verification

In each task:



这里需要注意Face Verification 和Face Recognition 是不一样的，前者是说给你一张人脸，判定是否是指定的人脸，比如人脸验证来解锁设备；后者是辨别一个人脸是人脸集合里面谁，比如公司人脸签到打卡。

下面我们就以Face Verification 为例，讲一下Metric-based Meta Learning

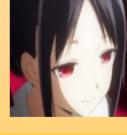
Meta Learning

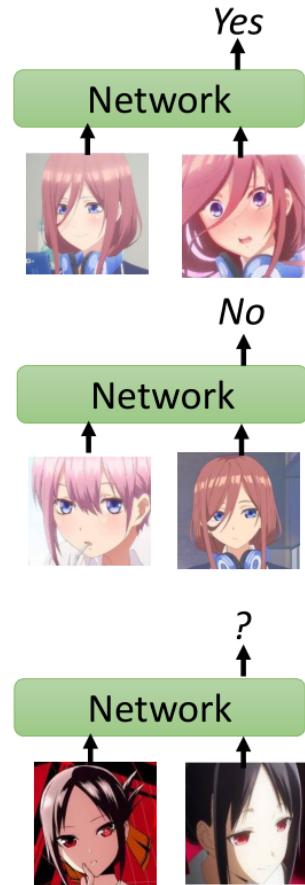
Same approach for Speaker Verification

Training Tasks

Train		Test		Yes
Train		Test		No
Train		Test		No

Testing Tasks

Train		Test		Yes or No
-------	---	------	---	-----------------



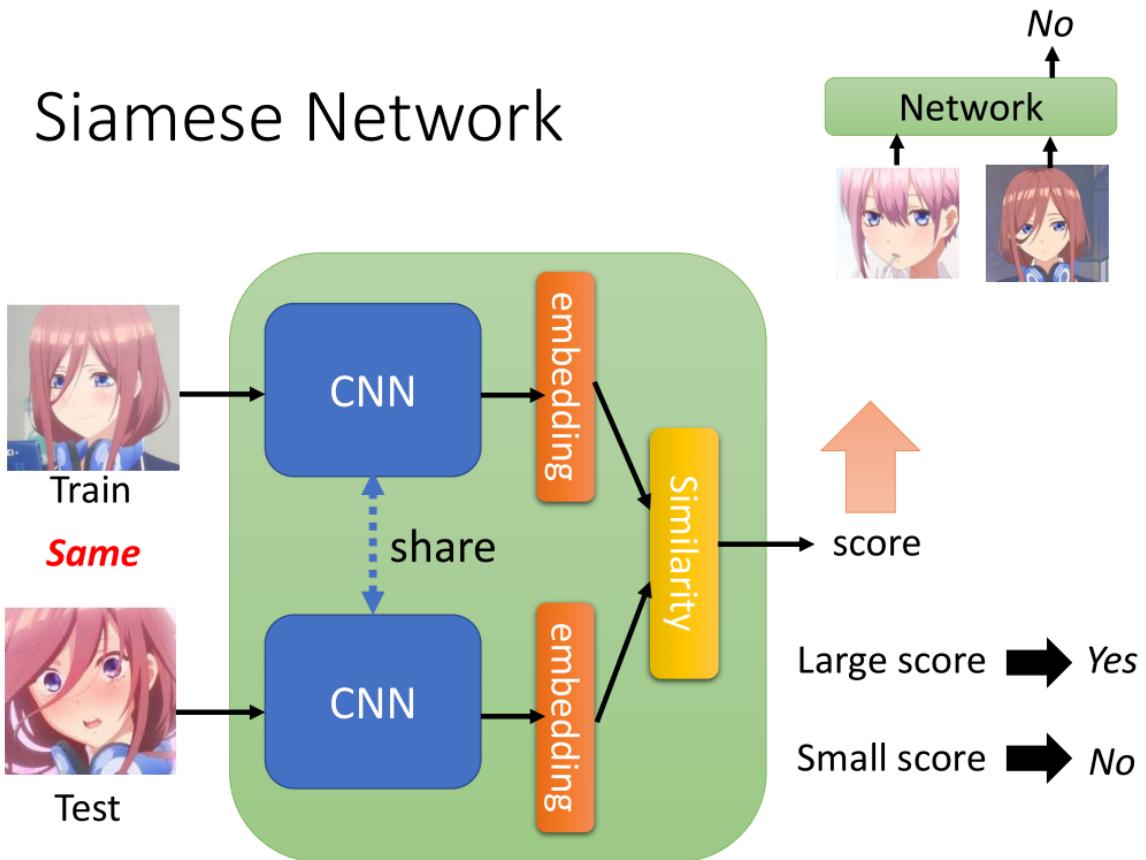
训练任务集中的任务都是人脸辨识数据，每个任务的测试集就是某个人的面部数据，测试集就是按标准（如手机录制人脸）收集的人脸数据，如果这个人和训练集相同就打一个Yes 标签，否则就打一个No 标签。测试任务和训练任务类似。总的来说，network 就是吃训练的人脸和测试的人脸，它会告诉你Yes or No。

测试任务要与训练任务有点不同，测试的脸应该没有出现在测试任务中。

Siamese Network

实际上是怎么做的呢，使用的技术是Siamese Network（孪生网络）。

Siamese Network



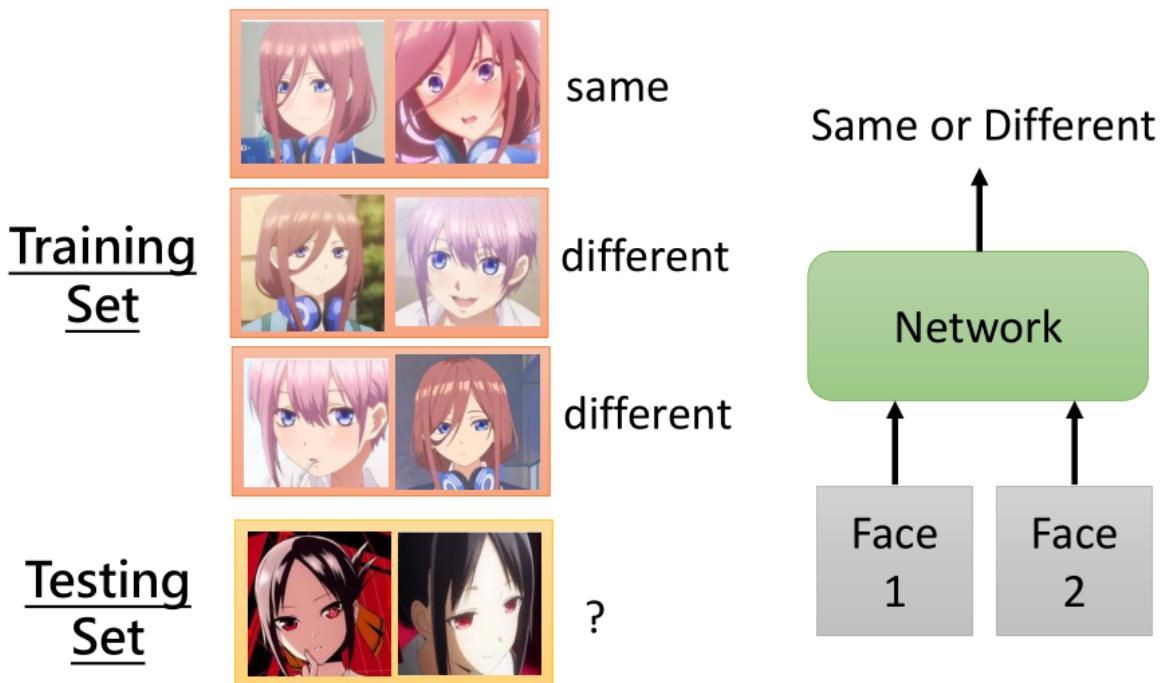
Siamese Network 的结构如上图所示，两个网络往往是共享参数的，根据需要有时候也可以不共享，假如说你现在觉得Training data 和Testing data 在形态上有比较大的区别，那你就应该不共享两个网络的参数。

从两个CNN 中抽出两个embedding , 然后计算这两个embedding 的相似度，比如说计算conference similarity 或者Euclidean Distance , 你得到一个数值score , 这个数值大就代表Network 的输出是Yes , 如果数值小就代表输出是No 。

Intuitive Explanation

接下来从直觉上来解释一下孪生网络。

Binary classification problem: “Are they the same?”

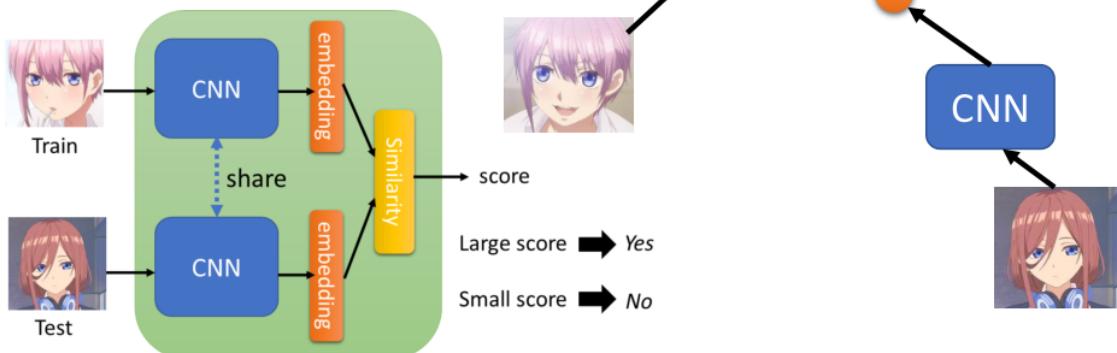


如上图所示，你可以把Siamese Network 看成一个二分类器，他就是吃进去两张人脸比较一下相似度，然后告诉我们Yes or No。这样解释会比从Meta Learning 的角度来解释更容易理解。

Siamese Network - Intuitive Explanation

Learning embedding
for faces

e.g. learn to ignore the
background



如上图所示，Siamese Network 做的事情就是把人脸投影到一个空间上，在这个空间上只要是同一个人的脸，不管机器看到的是他的哪一侧脸，都能被投影到这个空间的同一个位置上。同一个人距离越近越好，不同的人距离越远越好。

这种图片降维的方法，这和Auto-Encoder有什么区别呢，他比Auto-Encoder 好在哪？

你想你在做Auto-Encoder 的时候network不知道你要解的任务是什么，它会尽可能记住图片中所有的信息，但是它不知道什么样的信息是重要的什么样的信息是不重要的。

例子里面上图右侧，如果用Auto-Encoder 它可能会认为一花（左下）和三玖（右上）是比较接近的，因为他们的背景相似。在Siamese Network 中，因为你要求network 把一花（左下）和三玖（右上）拉远，把三玖（右上）和三玖（右下）拉近，它可能会学会更加注意头发颜色的信息，要忽略背景的信息。

To learn more...

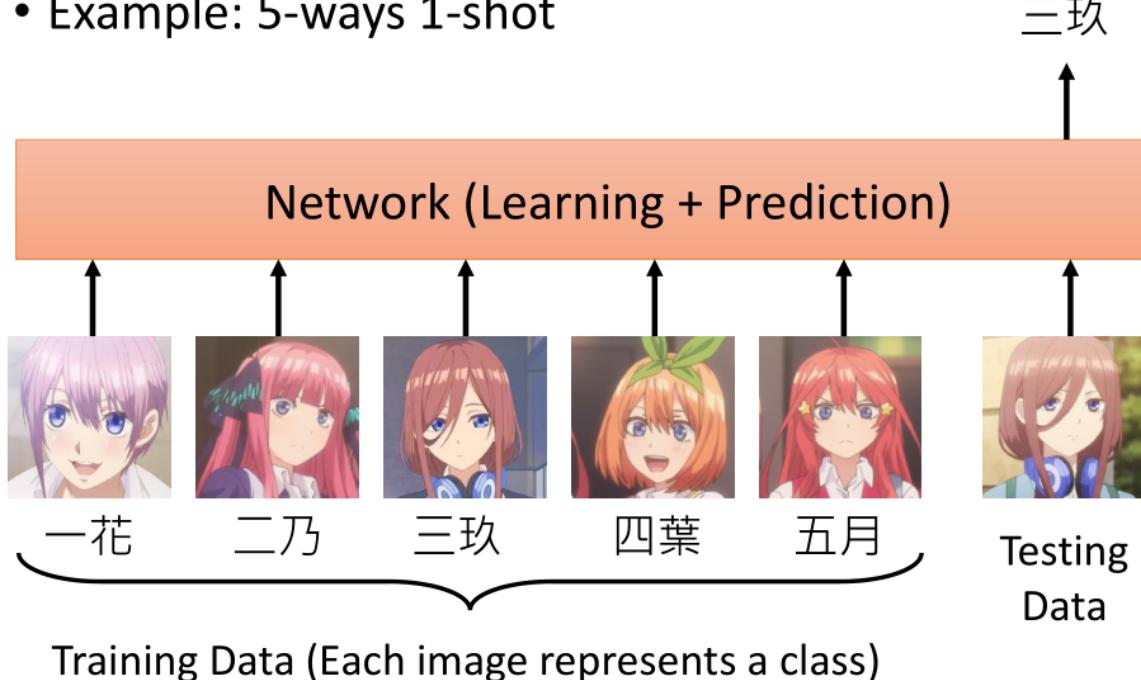
计算两个embedding的相近度

- What kind of distance should we use?
 - SphereFace: Deep Hypersphere Embedding for Face Recognition
 - Additive Margin Softmax for Face Verification
 - ArcFace: Additive Angular Margin Loss for Deep Face Recognition
- Triplet loss (三元是指：从训练集中选取一个样本作为Anchor，然后再随机选取一个与Anchor属于同一类别的样本作为Positive，最后再从其他类别随机选取一个作为Negative)
 - Deep Metric Learning using Triplet Network
 - FaceNet: A Unified Embedding for Face Recognition and Clustering

N-way Few/One-shot Learning

刚才的例子中，训练资料都只有一张，机器只要回答Yes or No。那现在如果是一个分类的问题呢？现在我们打算把同样的概念用在5-way 1-shot 的任务上该怎么办呢？

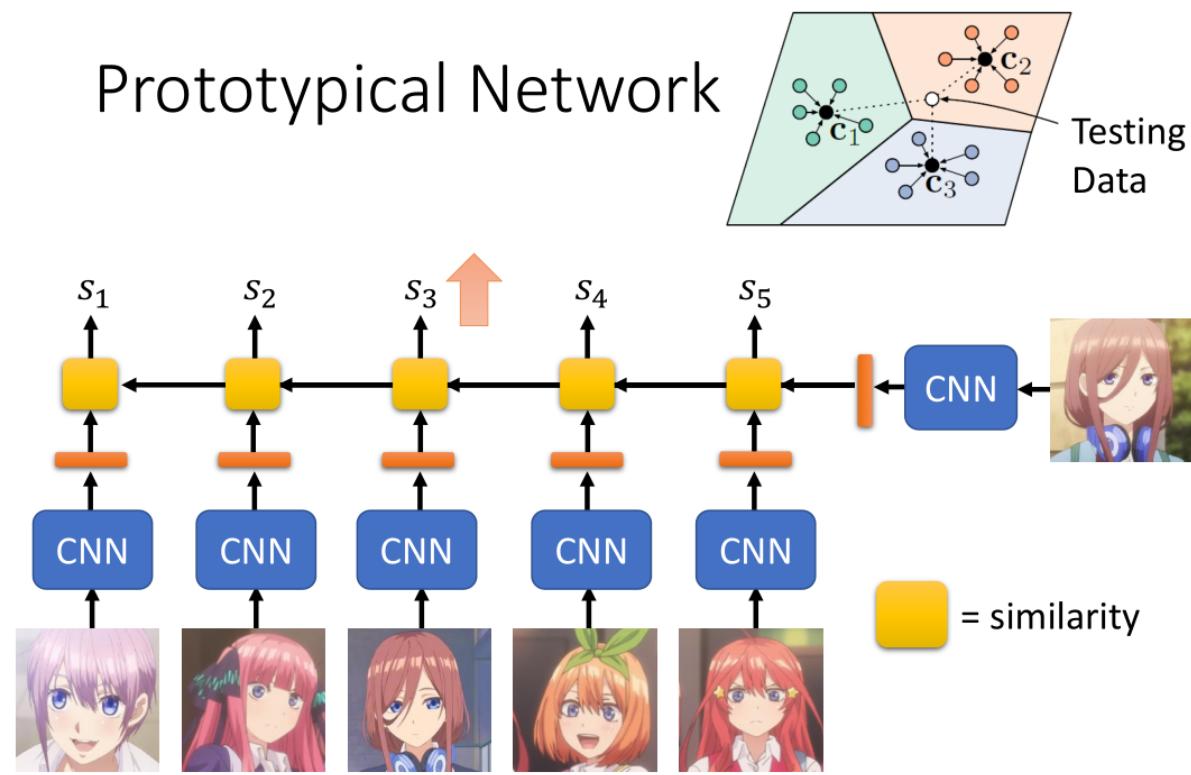
• Example: 5-ways 1-shot



5-way 1-shot 就是说5个类别，每个类别中只有1个样本。就比如说上图，《五等分花嫁》中的五姐妹，要训一个模型分辨一个人脸是其中的谁，而训练资料是每个人只有一个样本。我们期待做到的事情是，Network 就把这五张带标签的训练图片外加一张测试图片都吃进去，然后模型就会告诉我们测试图片的分辨结果。

Prototypical Network

那模型的架构要怎么设计呢，这是一个经典的做法：



<https://arxiv.org/abs/1703.05175>

这个方法和Siamese Network 非常相似，只不过从input 一张training data 扩展到input 多张training data。

如上图所示，把每张图片丢到同一个CNN 中算出一个embedding 用橙色条表示，然后把测试图片的embedding 和所有训练图片的embedding 分别算出相似度 s_i 。黄色的方块表示计算相似度。

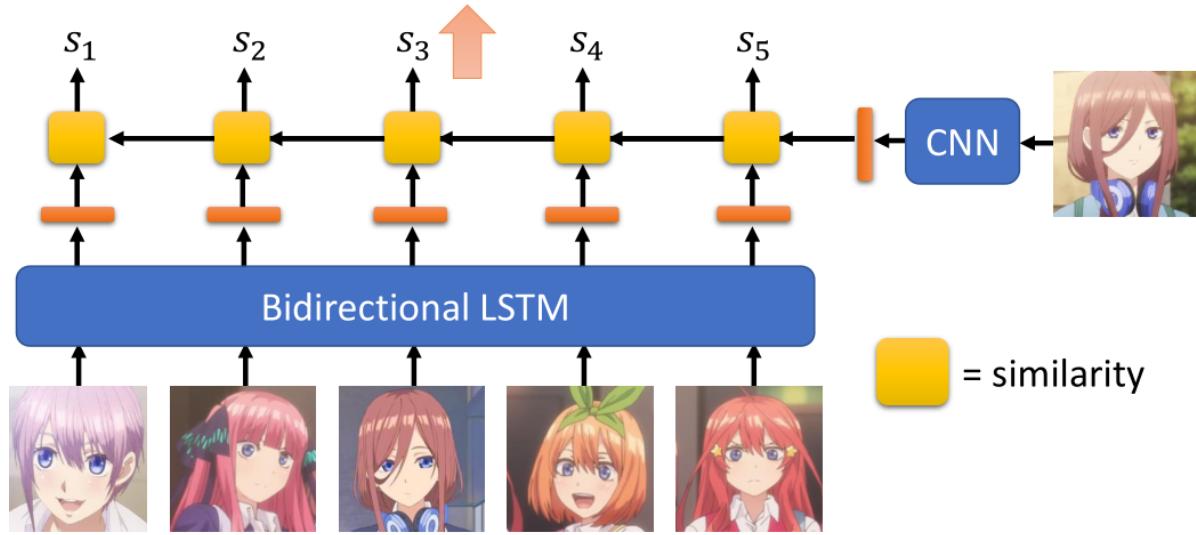
接下来，取一个softmax，这样就可以和正确的标签做cross entropy，去minimize cross entropy，这就和一般的分类问题的loss function相同的，就可以根据这个loss 做一次gradient descent，因为是1-shot 所以只能做一次参数更新。

那如果是few-shot 呢，怎么用Prototypical Network 解决呢。如右上角，我们把每个类别的几个图片用CNN 抽出的embedding 做average 来代表这个类别就好了。进来一个Testing Data 我们就看它和哪个 class 的average 值更接近，就算作哪一个class。

Matching Network

Matching Network 和Prototypical Network 最不同的地方是，Matching Network 认为也许Training data 中的图片互相之间也是有关系的，所以用Bidirectional LSTM 处理Training data，把Training data 通过一个Bidirectional LSTM 也会得到对应的embedding，然后的做法就和Prototypical Network 是一样的。

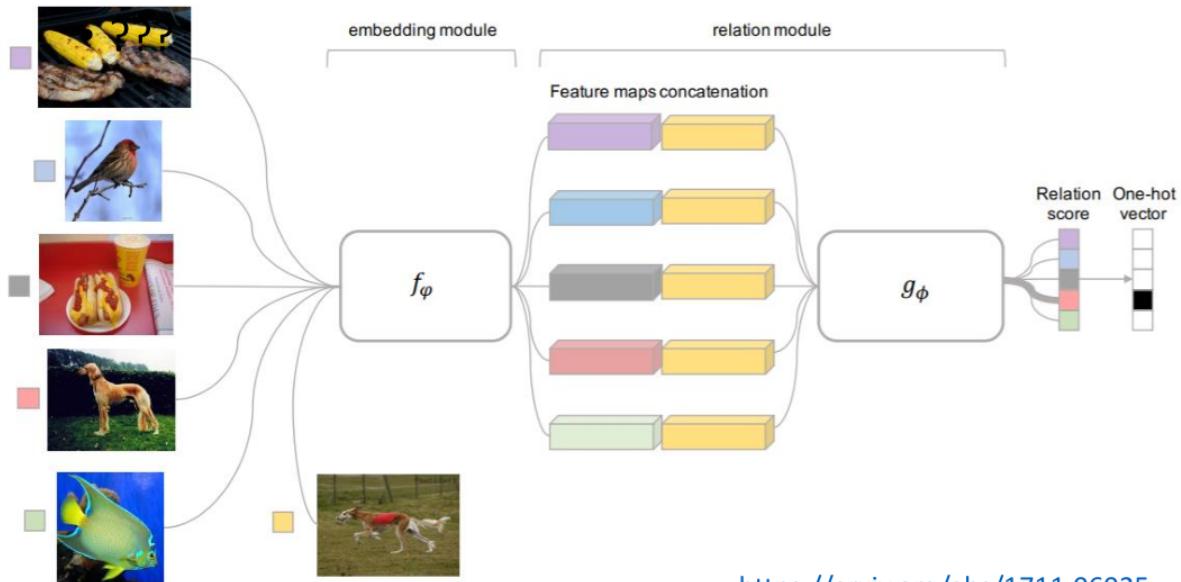
Considering the relationship among the training examples



<https://arxiv.org/abs/1606.04080>

事实上是Matching Network 先被提出来的，然后人们觉得这个方法有点问题，问题出在Bidirectional LSTM 上，就是说如果输入Training data 的顺序发生变化，那得到的embedding 就变了，整个network 的辨识结果就可能发生变化，这是不合理的。

Relation Network



<https://arxiv.org/abs/1711.06025>

这个方法和上面讲过的很相似，只是说我们之前通过人定的相似度计算方法计算每一类图片和测试图片的相似度，而Relation Network 是希望用另外的模型 g_ϕ 来计算相似度。

具体做法就是先通过一个 f_ϕ 计算每个类别的以及测试数据的embedding，然后把测试数据的embedding 接在所有类别embedding 后面丢入 g_ϕ 计算相似度分数。

Few-shot learning for Imaginary Data

我们在做Few-Shot Learning 的时候的难点就是训练数据量太少了，那能不能让机器自己生成一些数据提供给训练使用呢。这就是Few-shot learning for Imaginary Data 的思想。

Learn 一个Generator G ，怎么Learn 出这个Generator 我们先不管，你给Generator 一个图片，他就会生成更多图片，比如说你给他三玖面无表情的样子，他就会YY出三玖卖萌的样子、害羞的样子、生气的样子等等。然后把生成的图片丢到Network 中做训练，结束。

实际上，真正做训练的时候Generator 和Network 是一起training的，这就是Few-shot learning for Imaginary Data 的意思。

Meta Learning-Train+Test as RNN

我们在讲Siamese Network 的时候说，你可以把Siamese Network 或其他Metric-based 的方法想成是Meta Learning，但其实你是可以从其他更容易理解的角度来考虑这些方法。总的来说，我们就是要找一个function，这个function 可以做的到就是吃训练数据和测试数据，然后就可以吐出测试数据的预测结果。我们实际上用的Siamese Network 或者Prototypical Network、Matching Network 等等的方法多可以看作我们为了实现这个目的做模型架构的变形。

现在我们想问问，有没有可能直接用常规的network 做出这件事？有的。

用LSTM 把训练数据和测试数据吃进去，在最后输出测试数据的判别结果。训练图片通过一个CNN 得到一个embedding，这个embedding 和这个图片的label (one-hot vector) 做concatenate (拼接) 丢入LSTM 中，Testing data 我们不知道label 怎么办，我们就用0 vector 来表示，然后同样丢入LSTM，得到output 结束。这个方法用常规的LSTM 是train 不起来的，我们需要修改LSTM 的架构，有两个方法，具体方法我们就不展开讲了，放出参考链接：

One-shot Learning with Memory-Augmented Neural Networks

<https://arxiv.org/abs/1605.06065>

A Simple Neural Attentive Meta-Learner

<https://arxiv.org/abs/1707.03141>

SNAIL和我们上面刚说过想法的是一样的，输入一堆训练数据给RNN 然后给他一个测试数据它输出预测结果，唯一不同的东西就是，它不是一个单纯的RNN，它里面有在做回顾这件事，它在input 第二笔数据的时候会回去看第一笔数据，在input 第三笔数据的时候会回去看第一第二笔数据...在input 测试数据的时候会回去看所有输入的训练数据。

所以你会发现这件事是不是和prototypical network 和matching network 很相似呢，matching network 就是计算input 的图片和过去看过的图片的相似度，看谁最像，就拿那张最像的图片的label 当作network 的输出。SNAIL 的回顾过去看过的数据的做法就和matching network 的计算相似度的做法很像。

所以说，你虽然想用更通用的方法做到一个模型直接给出测试数据预测结果这件事，然后你发现你要改network 的架构，改完起了个名字叫SNAIL 但是他的思想变得和原本专门为这做到这件事设计的特殊的方法如matching network 几乎一样了，有点殊途同归的意思。

Life-long Learning

Life-long Learning

开始之前的说明，如果读者是学过transfer learning 的话，学这一节可能会轻松很多，LLL的思想在我看来是和transfer learning是很相似的。

可以直观的翻译成终身学习，我们人类在学习过程中是一直在用同一个大脑在学习，但是我们之前讲的所有机器学习的方法都是为了解决一个专门的问题设计一个模型架构然后去学习的。所以，传统的机器学习的情景和人类的学习是很不一样的，现在我们就要考虑为什么不能用同一个模型学会所有的任务。

也有人把Life Long Learning 称为Continuous Learning, Never Ending Learning, Incremental Learning，在不同的文献中可能有不同的叫法，我们只要知道这些方法都是再指终生学习就可。

我想大多数人在学习机器学习之前的是这样认为的，我们教机器学学会任务1，再教会它任务2，我们就不断地教它各种任务，学到最后它就成了天网。但是实际上我们都知道，现在的机器学习是分开任务来学的，就算是这样很多任务还是得不到很好的结果。所以机器学习现在还是很初级的阶段，在很多任务上都无法胜任。

我们今天分三个部分来叙述**Life-Long Learning**：

- **Knowledge Retention 知识保留**
 - but NOT Intransigence 但不固执，不会拒绝学习新的东西
- **Knowledge Transfer 知识潜移**
- **Model Expansion 模型扩展**
 - but Parameter Efficiency 但参数高效

Knowledge Retention

知识保留，但不顽固

知识保留但不顽固的精神是：我们希望模型在做完一个任务的学习之后，在学新的知识的时候，能够保留对原来任务能力，但是这种能力的保留又不能太过顽固以至于不能学会新的任务。

Example - Image

我们举一个例子看看机器的脑洞有多大。这里是影像辨识的例子，来看看在影像辨识任务中是否需要终身学习。

我们有两个任务，都是在做手写数字辨识，但是两个的corpus 是不同的（corpus1 图片上存在一些噪声）。network 的架构是三层，每层都是50个neuron，然后让机器先学任务1，学完第一个任务以后在两个corpus 上进行测试，得到的结果task1: 90%；task2: 96% (task2的结果更好一点其实是很直觉的，因为corpus2上没有noise，这可以理解为transfer learning)。然后我们在把这个模型用corpus2 进行一波训练，再在两个corpus上进行测试得到的结果task1: 80%；task2: 97%，发现第一个任务有被遗忘的现象发生。

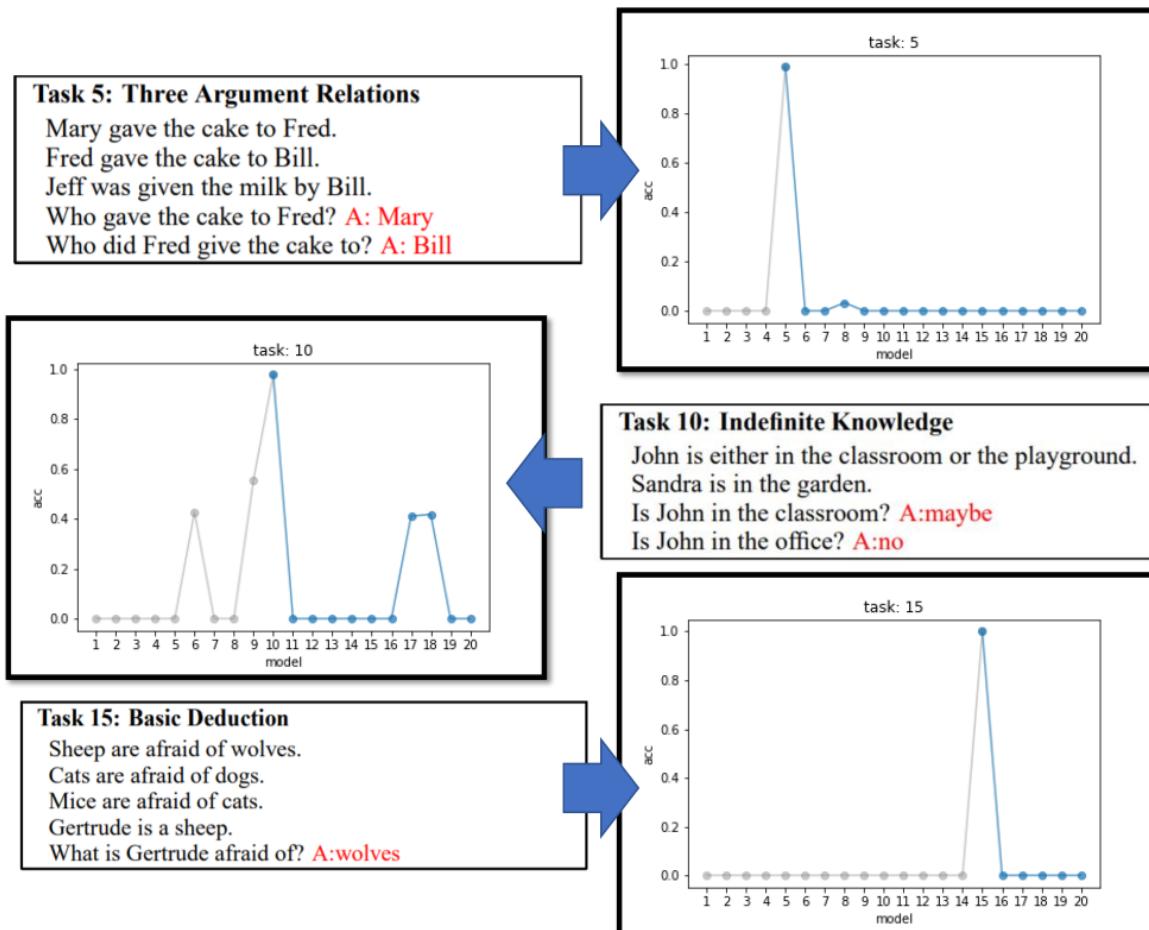
这时候你可能会说，这个模型的架构太小了，他只有三层每层只有50个neuron，会发生遗忘的现象搞不好是因为它脑容量有限。但是我们实践过发现并不是模型架构太小。我们把两个corpus 混到一起用同样的模型架构train 一发，得到的结果task1: 89%；task2: 98%

所以说，明明这个模型的架构可以把两个任务都学的很好，为什么先学一个在学另一个的话会忘掉第一个任务学到的东西呢。

Example - Question Answering

问答系统要做的事情是训练一个Deep Network，给这个模型看很多的文章和问题，然后你问它一个问题，他就会告诉你答案。具体怎么输入文章和问题，怎么给你答案，怎么设计网络，不展开。

对于QA系统已经被玩烂的corpus 是bAbi 这个数据集，这里面有20种不同的题型，比如问where、what等。可以分别用20个模型解题，也可以用1个模型同时解20个题型。我们训练一个模型从第一个题型开始学习，依次学完20种题型，每次学习完成以后我们都用题型五做一次测试，也就是以题型五作为baseline，结果如下：

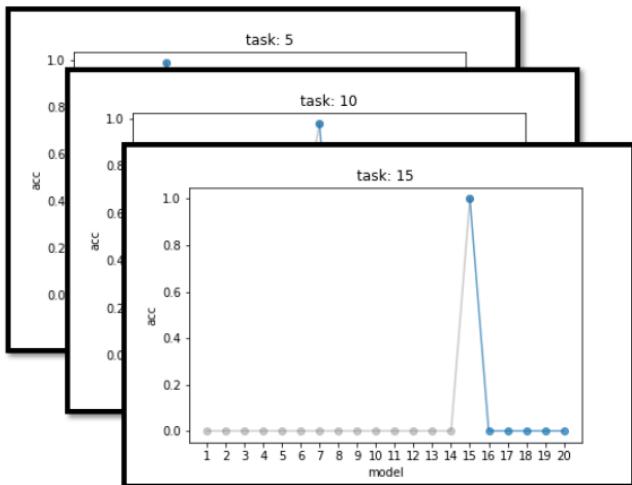


我们可以看到只有在学完题型五的时候，再问机器题型五的问题，它可以给出很好的答案，但是在学完题型六以后它马上把题型五忘的一干二净了。这个现象在以其他的题型作为baseline 的时候同样出现了。

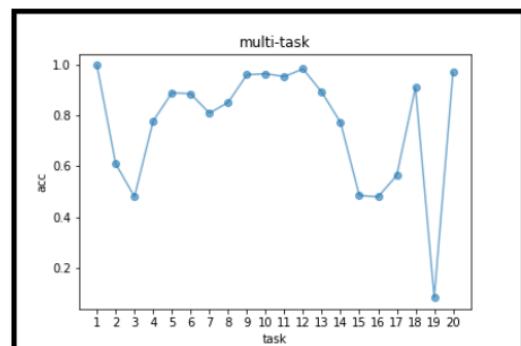
有趣的是，在题型10作为baseline 的时候可能是由于题型6、9、17、18和题型10比较相似，所以在做完这些题型的QA任务的时候在题型10上也能得到比较好的结果。

那你又会问了，是不是因为网络的架构不够大，机器的脑容量太小以至于学不起来。其实不是，当我们同时train这20种题型得到的结果是还不错的。

Sequentially train the 20 tasks



Jointly training the 20 tasks



是不為也

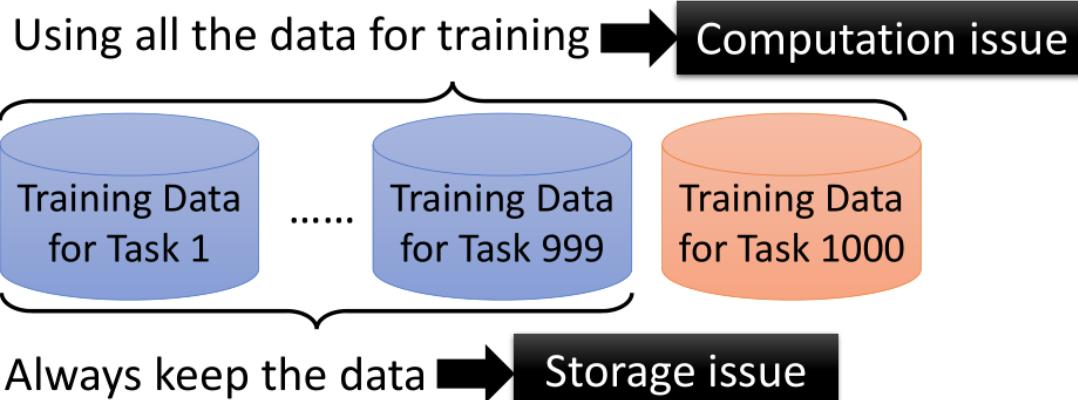
非不能也

Catastrophic Forgetting

所以机器的遗忘是和人类很不一样的，他不是因为脑容量不够而忘记的，不知道为什么它在学过一些新的任务以后就会较大程度的遗忘以前学到的东西，这个状况我们叫做Catastrophic Forgetting（灾难性遗忘）。之所以加个形容词是因为这种遗忘是不可接受，只要学新的东西旧的东西就都出来了。

你可能会说这个灾难性遗忘的问题你上面不是已经有了一个很好的解决方法了吗，你只要把多个任务的corpus 放在一起train 就好了啊。

- Multi-task training can solve the problem!



- Multi-task training can be considered as the upper bound of LLL.

但是，长远来说这一招是行不通的，因为我们很难一直维护所有使用过的训练数据；而且就算我们很好的保留了所有数据，在计算上也有问题，我们每次学新任务的时候就要重新训练所有的任务，这样的代价是不可接受的。

另外，**多任务同时train** 这个方法其实可以作为LLL的上界。

我们期待的是，不做Multi-task training的情况下，让机器不要忘记过去学过的东西。

那这个问题有什么样的解法呢，接下来就来介绍一个经典解法。

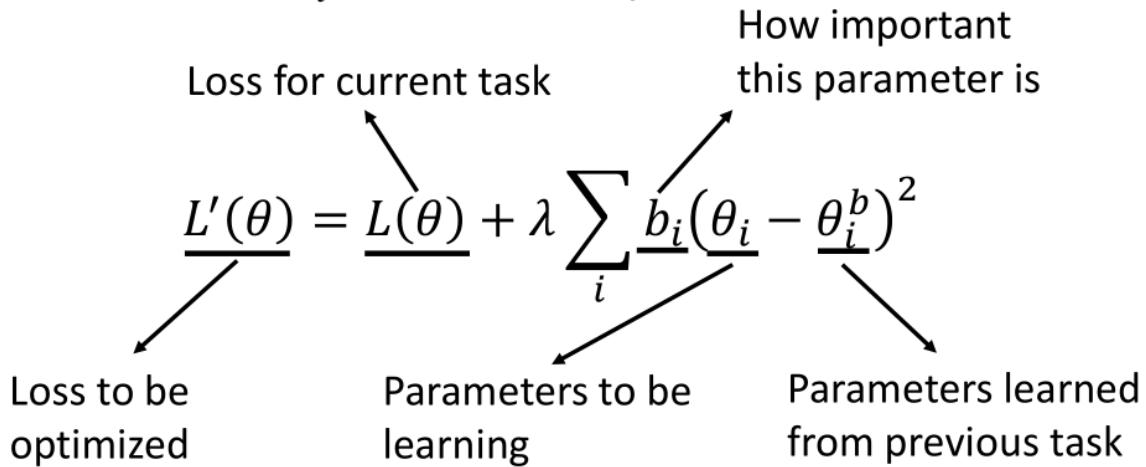
Elastic Weight Consolidation (EWC)

基本精神：网络中的部分参数对先前任务是比较有用的，我们在学新的任务的时候只改变不重要的参数。

Basic Idea: Some parameters in the model are important to the previous tasks. Only change the unimportant parameters.

θ^b is the model learned from the previous tasks.

Each parameter θ_i^b has a “guard” b_i



如上图所示， θ^b 是模型从先前的任务中学出来的参数。

每个参数 θ_i^b 都有一个守卫 b_i ，这个守卫就会告诉我们这个参数有多重要，我们有多么不能更改这个参数。

我们在做EWC 的时候 (train 新的任务的时候) 需要在原先的损失函数上加上一个regularization，如上图所示，我们通过平方差的方式衡量新的参数 θ_i 和旧的参数 θ_i^b 的差距，然后乘上守卫，把所有参数加起来。

我们学习新的任务时，不止希望把新的任务做好，也希望新的参数和旧的参数差别不要太大，这种限制对每个参数是不同的：当这个守卫 b_i 等于零的时候就是说参数 θ_i 是没有约束的，可以根据当前任务随意更改，当守卫 b_i 趋近于无穷大的时候，说明这个参数 θ_i 对先前的任务是非常重要的，希望模型不要变动这个参数。

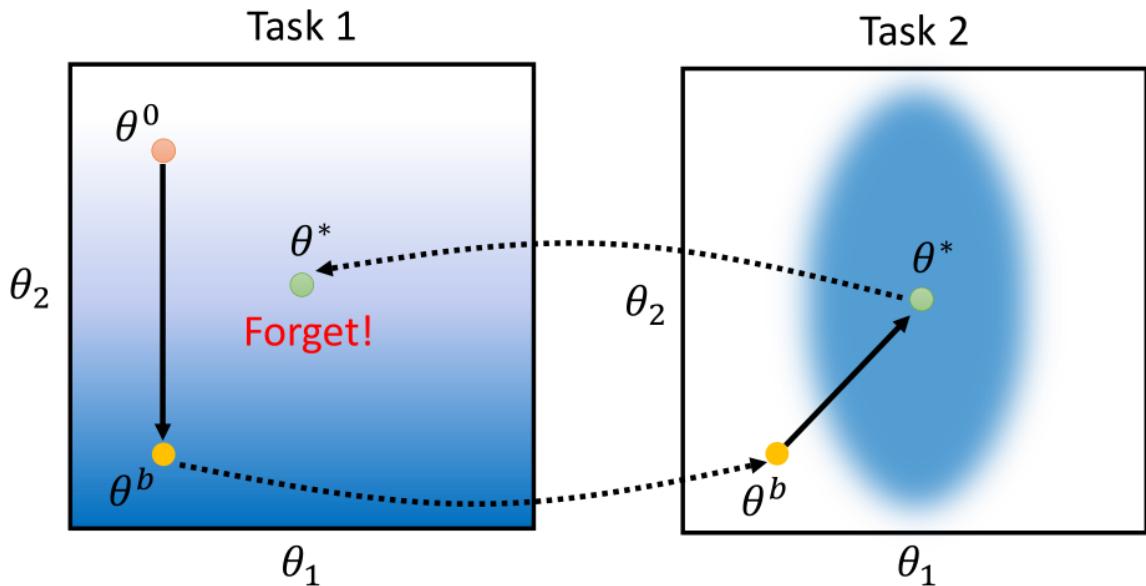
One kind of regularization. θ_i should be close to θ^b in certain directions.

$$L'(\theta) = L(\theta) + \lambda \sum_i b_i (\theta_i - \theta_i^b)^2$$

If $b_i = 0$, there is no constraint on θ_i

If $b_i = \infty$, θ_i would always be equal to θ_i^b

所以现在问题是， b_i 如何决定。这个问题我们下面来讲，先来通过一个简单的例子再理解一下EWC的思想：

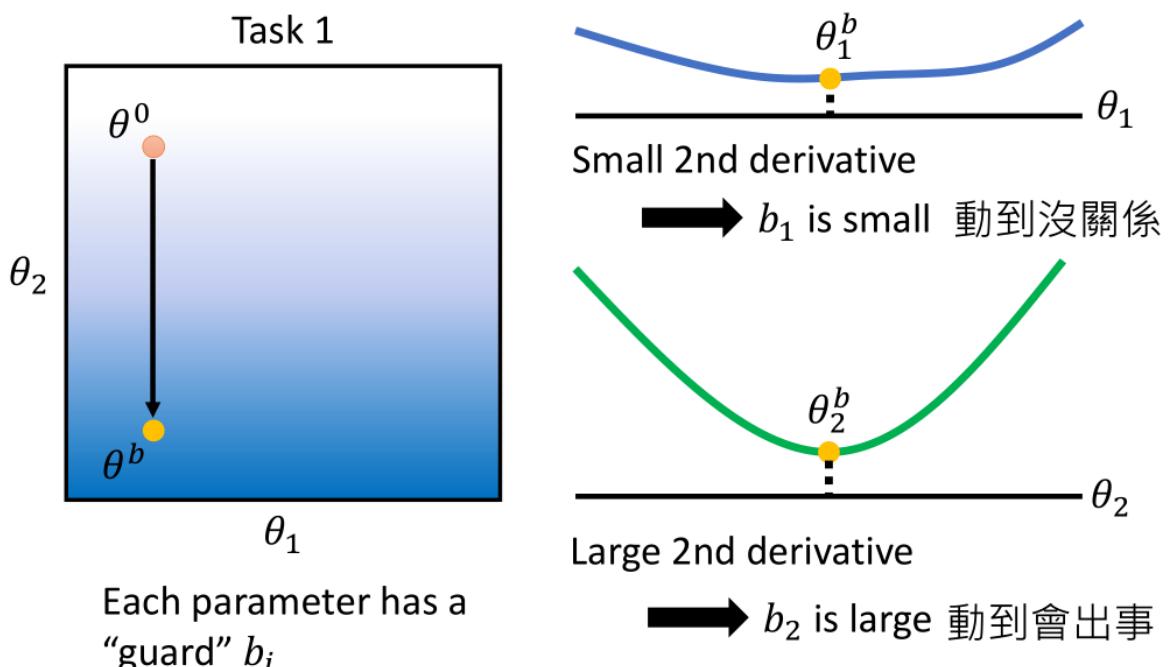


The error surfaces of tasks 1 & 2.

(darker = smaller loss)

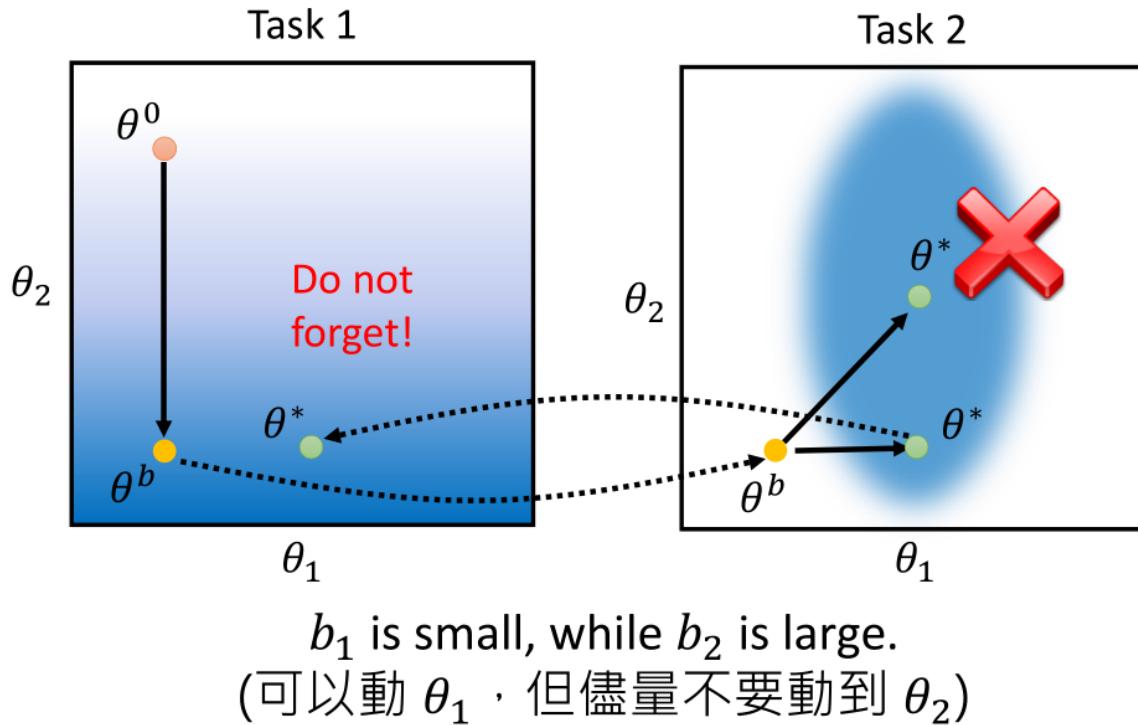
上图是这样的，假设我们的模型只有两个参数，这两个图是两个task 的error surface，颜色越深loss 越大。假如说我们让机器学task1的时候我们的参数从 θ^0 移动到 θ^b ，然后我们又让机器学task2，在这学这个任务的时候我们没有加任何约束，它学完之后参数移动到了 θ^* ，这时候模型参数在task1的error surface 上就是一个不太好的点。这就直观的解释了为什么会出现Catastrophic Forgetting。

当我们使用EWC 对模型的参数的变化做一个限制，就如上面说的，我们给每个参数加一个守卫 b_i ，这个 b_i 是这么来的呢？



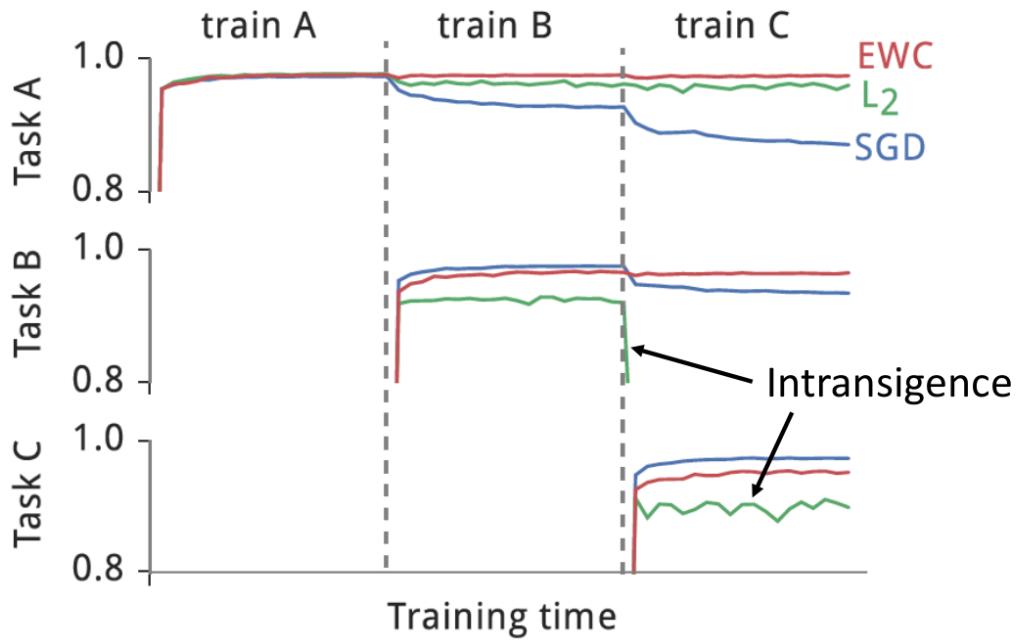
不同文章有不同的做法，这里有一个简单的做法就是算这个参数的二次微分（loss对 θ 的二次微分体现参数loss变化的剧烈程度，二次微分值越大，原函数图像在该点变化越剧烈），如上图所示。我们可以看出， θ_1^b 在二次微分曲线的平滑段其变化不会造成原函数图像的剧烈变化，我们要给它一个大的守卫 b_1 ，反之 θ_2^b 则在谷底其变化会造成二次微分值的增大，导致原函数的变化更剧烈，我们要给它一个大的守卫 b_2 。也就是说， θ_1^b 可以动， θ_2^b 尽量别动。

有了上述的constraint，我们就能让模型参数尽量不要在 θ_2 方向上移动，可以在 θ_1 上移动，得到的效果可能就会是这样的：



Experiment

我们来看看EWC的原始paper中的实验结果：



MNIST permutation, from the original EWC paper

三个task其实就是对MNIST 数据集做不同的变换后做辨识任务。每行是模型对该行的task准确率的变化，从第一行可以看出，当我们用EWC的方法做完三个任务学习以后仍然能维持比较好的准确率。值得注意的是，在下面两行中，L2的方法在学习新的任务的时候发生了Intransigence（顽固）的现象，就是模型顽固的记住了以前的任务，过于保守，而无法学习新的任务。

Variant

有很多EWC 的变体，给几个参考：

- Elastic Weight Consolidation (EWC)
 - <http://www.citeulike.org/group/15400/article/14311063>
- Synaptic Intelligence (SI)
 - <https://arxiv.org/abs/1703.04200>
- Memory Aware Synapses (MAS)
 - Special part: Do not need labelled data
 - <https://arxiv.org/abs/1711.09601>

Generating Data

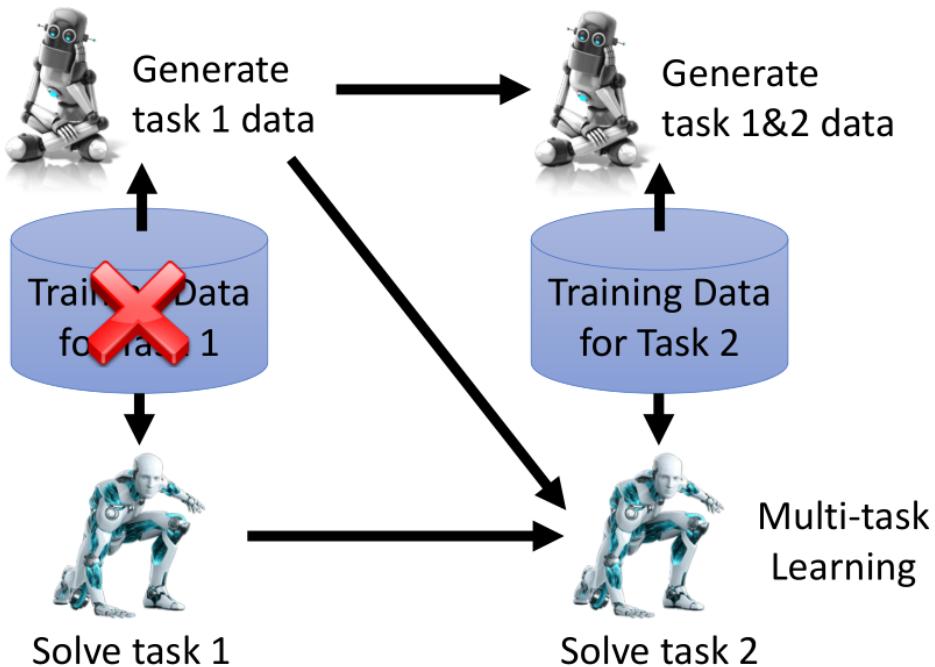
上面我们说Multi-task Learning 虽然好用，但是由于存储和计算的限制我们不能这么做，所以采取了EWC 等其他方法，而Multi-task Learning 可以考虑为Life-Long Learning 的upper bound。反过来我们不禁在想，虽然说要存储所有过去的资料很难，但是Multi-task Learning 确实那么好用，那我们能不能Learning 一个model，这个model 可以产生过去的资料，所以我们只要存一个model 而不用存所有训练数据，这样我们就做Multi-task 的learning。（这里暂时忽略算力限制，只讨论数据生成问题）

Generating Data

<https://arxiv.org/abs/1705.08690>

<https://arxiv.org/abs/1711.10563>

- Conducting multi-task learning by generating pseudo-data using generative model



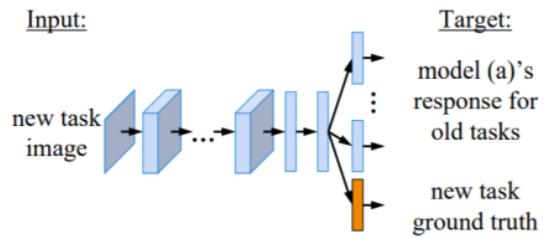
这个过程是这样的，我们先用training data 1 训练得到解决task 1 的model，同时用这些数据生成train 一个能生成这些数据的generator，存储这个generator 而不是存储training data；当来了新的任务，我们就用这个generator 生成task 1的training data 和task2 的training data 混在一起，用Multi-task Learning 的方法train 出能同时解决task1 和task2 的model，同时我们用混在一起的数据集train 出一个新的generator，这个generator 能生成这个混合数据集；以此类推。这样我们就可以做Mutli-task Learning，而不用存储大量数据。但是这个方法在实际中到底能不能做起来，还尚待研究，一个原因是实际上生成数据是没有那么容易的，比如说生成贴合实际的高清的影像对于机器来说就很难，所以这个方法是否做的起来还是一个尚待研究的问题。

Adding New Classes

在刚才的讨论中，我们都是假设解不同的任务用的是相同的网络架构，但是如果现在我们的task 是不同，需要我们更改网络架构的话要怎么办呢？比如说，两个分类任务的class数量不同，我们就要修改network 的output layer 。这里就列一些参考给大家：

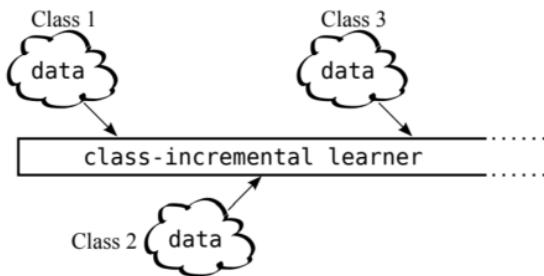
Learning without forgetting (LwF)

- <https://arxiv.org/abs/1606.09282>



iCaRL: Incremental Classifier and Representation Learning

- <https://arxiv.org/abs/1611.07725>



Knowledge Transfer

我们不仅希望机器可以记住以前学的knowledge，我们还希望机器在学习新的knowledge 的时候能把以前学的知识做transfer。

Train a model for each task?

- Knowledge cannot transfer across different tasks
- Eventually we cannot store all the models ...

我们之前都是每个任务都训练一个单独的模型，这种方式会损失一个很重要的信息，就是解决不同问题之间的通用知识。形象点来说，比如你先学过线性代数和概率论，那你在学机器学习的时候就会应用先前学过的知识，学起来就会很顺利。我们希望机器可以学完某些task后，可以在之后的task学习中更加顺利，希望机器能够把不同任务之间的知识进行迁移，让以前学过的知识可以应用到解决新的任务上面。

Life-Long v.s. Transfer

讲了这么多，你可能会说，这不就是在做transfer Learning 吗？

Transfer Learning 的精神是应用先前任务的模型到新的任务上，让模型可以解决或者说更好的解决新的任务，而不在乎此时模型是否还能解决先前的任务；

但是LLL 就比Transfer Learning 更进一步，它会考虑到模型在学会新的任务的同时，还不能忘记以前的任务的解法。

Evaluation

讲到这里，我们来说一下如何衡量LLL的好坏。其实，有很多不同的的衡量方法，这里简介一种。

Evaluation

$R_{i,j}$: after training task i, performance on task j

If $i > j$,

After training task i, does task j be forgot

If $i < j$,

Can we transfer the skill of task i to task j

		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	$R_{T,1}$	$R_{T,2}$		$R_{T,T}$

$$\text{Accuracy} = \frac{1}{T} \sum_{i=1}^T R_{T,i}$$

$$\text{Backward Transfer} = \frac{1}{T-1} \sum_{i=1}^{T-1} R_{T,i} - R_{i,i}$$

(It is usually negative.)

这里每一行是一个模型在不同任务上的测试结果，每一列是用一个任务对一个模型在做完某些任务的训练以后进行测试的结果。

$R_{i,j}$: 在训练完task i 后，模型在task j 上的performance 。

如果 $i > j$: 在学完task i 以后，模型在先前的task j 上的performance。

如果 $i < j$: 在学完task i 以后，模型在没学过的task j 上的performance，来说明前面学完的 i 个task 能不能transfer 到 task j 上。

$$\text{Accuracy} = \frac{1}{T} \sum_{i=1}^T R_{T,i}$$

$$\text{Backward Transfer} = \frac{1}{T-1} \sum_{i=1}^{T-1} R_{T,i} - R_{i,i}, \quad (\text{It is usually negative.})$$

$$\text{Forward Transfer} = \frac{1}{T-1} \sum_{i=2}^T R_{i-1,i} - R_{0,i}$$

Accuracy 是指说机器在学玩所有T 个task 以后，在所有任务上的平均准确率，所以如上图红框，就把最后一行加起来取平均就是现在这个LLL model 的Accuracy，形式化公式如上图所示。

Backward Transfer 是指机器有多会做Knowledge Retention (知识保留)，有多不会遗忘过去学过的任务。做法是针对每一个task 的测试集（每列），计算模型学完T 个task 以后的performance 减去模型刚学完对应该测试集的时候的performance，求和取平均，形式化公式如上图所示。

Backward Transfer 的思想就是把机器学到最后的表现减去机器刚学完那个任务还记忆犹新的表现，得到的差值通常都是负的，因为机器总是会遗忘的，它学到最后往往就一定程度的忘记以前学的任务，如果你做出来是正的，说明机器在学过新的知识以后对以前的任务有了触类旁通的效果，那就很强。

Evaluation

$R_{i,j}$: after training task i, performance on task j

If $i > j$,

After training task i, does task j be forgot

If $i < j$,

Can we transfer the skill of task i to task j

		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	$R_{T,1}$	$R_{T,2}$		$R_{T,T}$

$$\text{Accuracy} = \frac{1}{T} \sum_{i=1}^T R_{T,i}$$

$$\text{Backward Transfer} = \frac{1}{T-1} \sum_{i=1}^{T-1} R_{T,i} - R_{i,i}$$

$$\text{Forward Transfer} = \frac{1}{T-1} \sum_{i=2}^T R_{i-1,i} - R_{0,i}$$

Forward Transfer 是指机器有多会做Knowledge Transfer (知识迁移) , 有多会把过去学到的知识应用到新的任务上。做法是对每个task 的测试集，计算模型学过task i 以后对task i+1 的performance 减去随机初始的模型在task i+1 的performance , 求和取平均。

Gradient Episodic Memory (GEM)

上述的Backward Transfer 让这个值是正的就说明, model 不仅没有遗忘过学过的知识, 还在学了新的知识以后对以前的任务触类旁通, 这件事是有研究的, 比如GEM 。

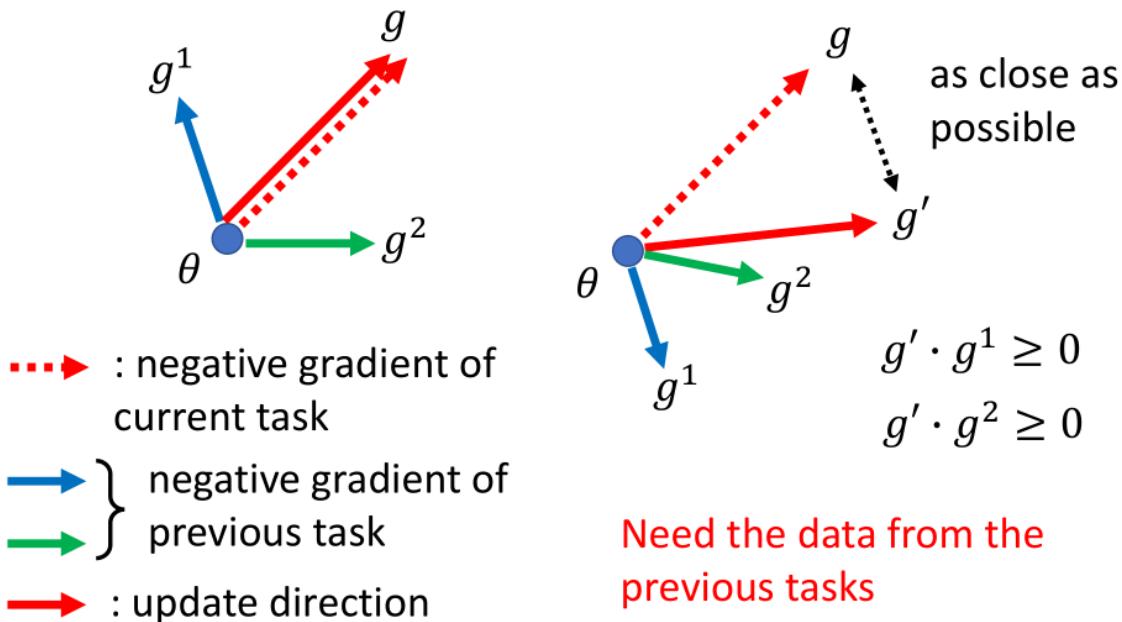
GEM: <https://arxiv.org/abs/1706.08840>

A-GEM: <https://arxiv.org/abs/1812.00420>

GEM 想做到的事情是, 在新的task 上训练出来的gradient 在更新的参数的时候, 要考虑一下过去的gradient , 使得参数更新的方向至少不能是以前梯度的方向 (更新参数是要向梯度的反方向更新) 。

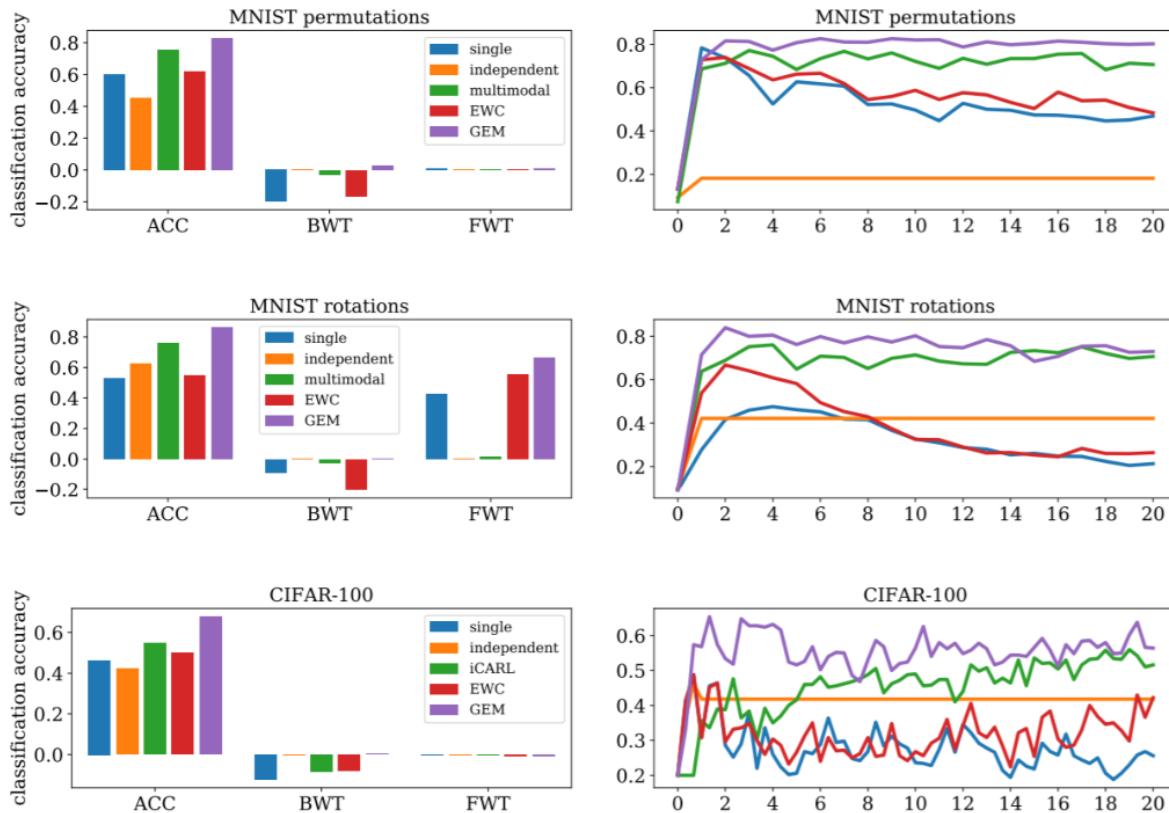
需要注意的是, 这个方法需要我们保留少量的过去的数据, 以便在train 新的task 的时候 (每次更新参数的时候) 可以计算出以前的梯度。

- Constraint the gradient to improve the previous tasks



形象点来说，以上图为例，左边，如果现在新的任务学出来的梯度是 g ，那更新的时候不会对以前的梯度 g^1, g^2 造成反向的影响；右边，如果现在新的情况是这样的，那梯度在更新的时候会影响到 g^1, g 和 g^2 的内积是负的，意味着梯度 g 会把参数拉向 g^1 的反方向，因此会损害model 在task 1上的performance。所以我们取一个尽可能接近 g 的 g' ，使得 g' 和两个过去任务数据算出来的梯度的内积都大于零。这样的话就不会损害到以前task 的performance，搞不好还能让过去的task 的loss 变得更小。

我们来看看GEM的效果：



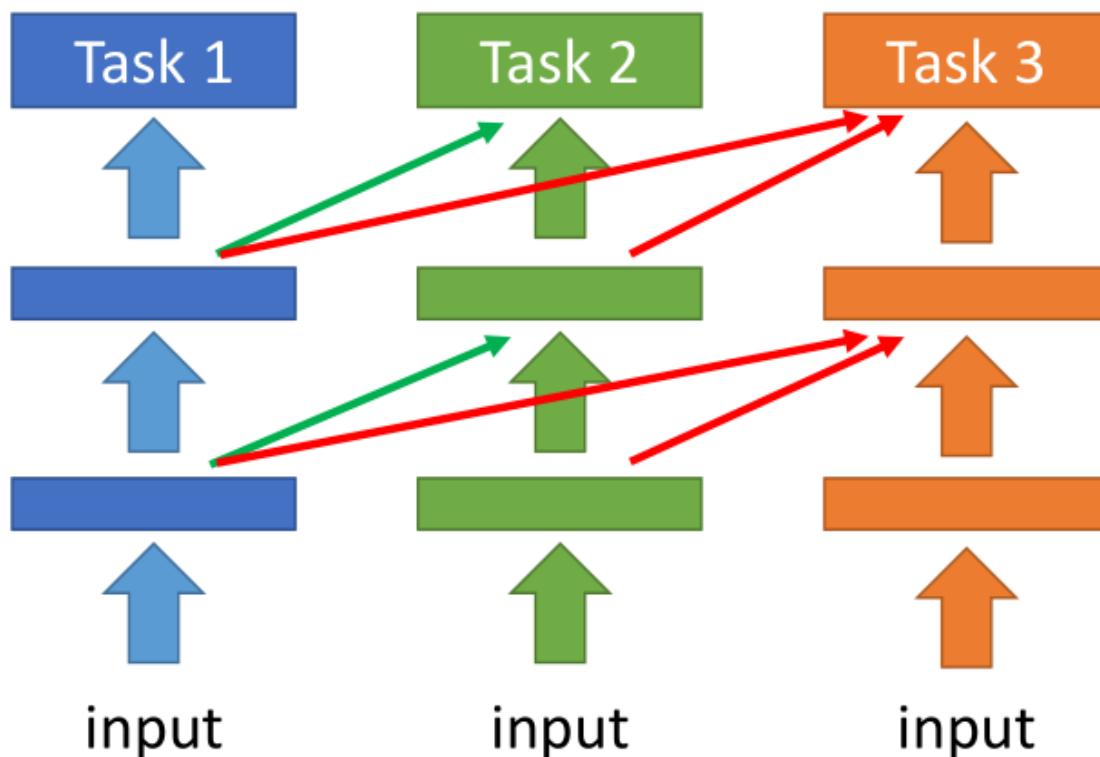
Model Expansion

but parameter efficiency

上面讲的内容，我们都假设模型是足够大的，也就是说模型的参数够多，它是有能力把所有任务都做好，只不过因为某些原因它没有做到罢了。但是如果现在我们的模型已经学了很多任务了，所有参数都被充分利用了，他已经没有能力学新的任务了，那我们就要给模型进行扩张。同时，我们还要保证扩张不是任意的，而是有效率的扩张，如果每次学新的任务，模型都要进行一次扩张，那这样的话model会扩张的太快导致你最终就会无法存下你的模型，而且臃肿的模型中大概率很多参数都是没有用的。

这个问题在2018年老师讲课的时候还没有很多文献可以参考，存在的模型也都做的不是特别好。

Progressive Neural Networks



<https://arxiv.org/abs/1606.04671>

这个方法是这样的，我们在学task 1的时候就正常train，在学task 2的时候就搞一个新的network，这个网路不仅会吃训练集数据，而且会把训练集数据input 到task 1的network中得到的每层输出吃进去，这时候是fix 住task 1 network，而调整task 2 network。同理，当学task 3的时候，搞一个新的network，这个网络不仅吃训练集数据，而且会把训练集数据丢入task 1 network 和 task 2 network，将其每层输出吃进去，也是fix 住前两个network 只改动第三个network。

这是一个早期的想法，2016年就出现了，但是这个方法终究还是不太能学很多任务。

Expert Gate

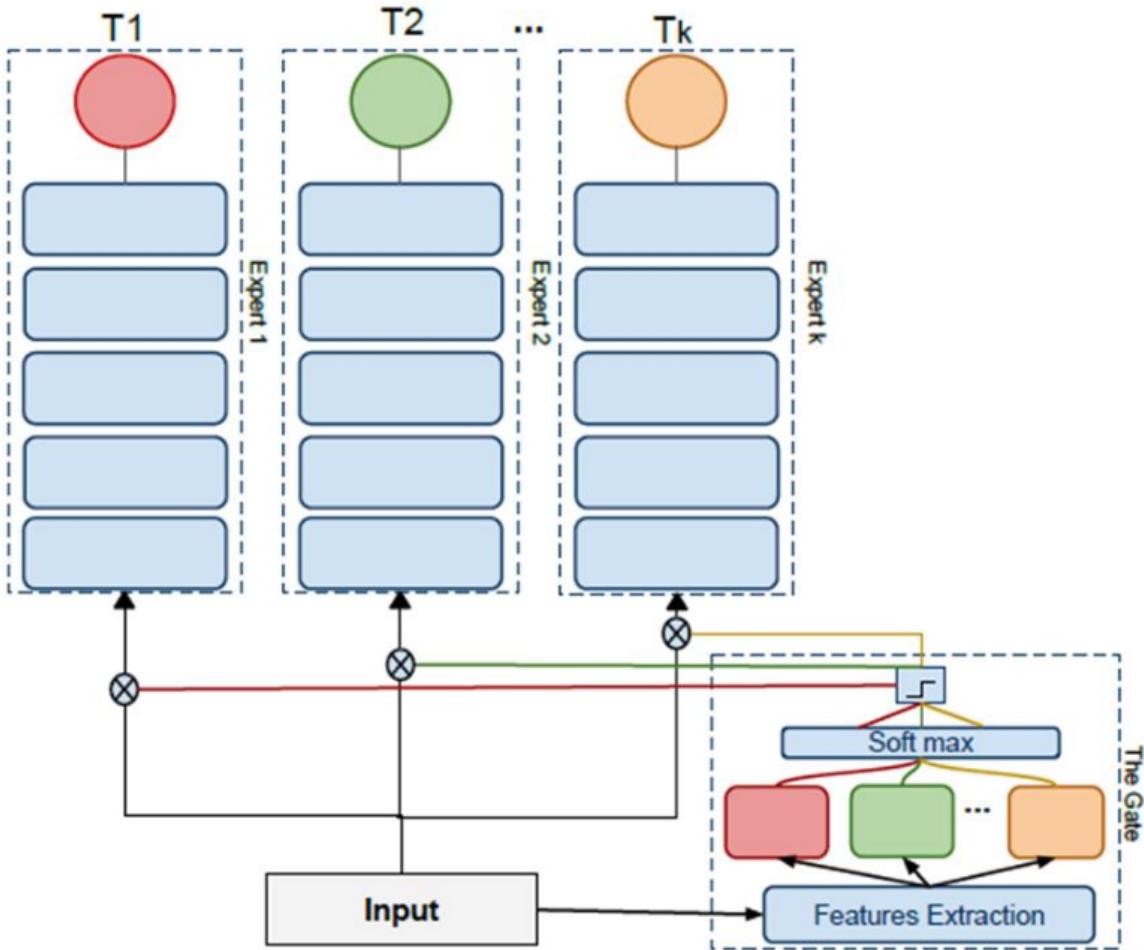
<https://arxiv.org/abs/1611.06194>

Aljundi, R., Chakravarty, P., Tuytelaars, T.: Expert gate: Lifelong learning with a network of experts. In: CVPR (2017)

思想是这样的：每一个task 训练一个network。

但是train 了另一个network，这个network 会判断新的任务和原先的那个任务最相似，加入现在新的任务和T1 最相似，那他就把network 1最为新任务的初始化network，希望以此做到知识迁移。

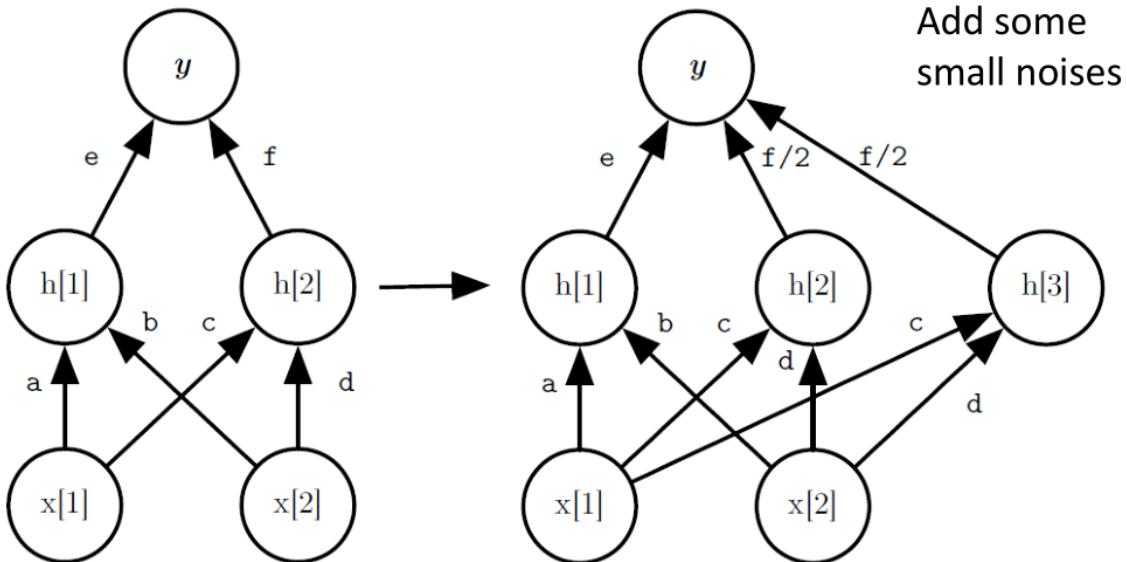
但是这个方法还是每一个任务都会有一个新的network，所以还是不太好。



Net2Net

如果我们在增加network 参数的时候直接增加神经元进去，可能会破坏这个模型原来做的准确率，那我们怎么增加参数才能保证不会损害模型在原来任务上的准确率呢？Net2Net 是一个解决方法：

Net2Net



Expand the network only when the training accuracy of the current task is not good enough. <https://arxiv.org/abs/1811.07017>

Net2Net的具体做法是这样的，如上图所示，当我要在中间增加一个neuron时，我们把f变为f/2，这样的话同样的输入在新旧两个模型中得到的输出就还是相同的，同时我们也增加了模型的参数。但是这样做出现一个问题，就是h[2] h[3]两个神经元将会在后面更新参数的时候完全一样，这样的话就相当于没有扩张模型，所以我们要在这些参数上加上一个小小的noise，让他们看起来还是有小小的不同，以便更新参数。

图中引用的文章就用了Net2Net，需要注意，不是来一个任务就扩张一次模型，而是当模型在新的任务的training data 上得不到好的Accuracy 的时候才用Net2Net 扩张模型。

Curriculum Learning

模型的效果是非常受任务训练顺序影响的。也就是说，会不会发生遗忘，能不能做到知识迁移，和训练任务的先后顺序是有很大关系的。假如说LLL在未来变得非常热门，那怎么安排机器学习的任务顺序可能会是一个需要讨论的热点问题，这个问题叫做Curriculum Learning。

<http://taskonomy.stanford.edu/#abstract> CVPR2018 的best paper

文章目的是找出任务间的先后次序，比如说先做3D-Edges 和 Normals 对 Point Matching 和 Reshading 就很有帮助。

Reinforcement Learning

Deep Reinforcement Learning

Scenario of Reinforcement Learning

在Reinforcement Learning里面会有一个Agent跟一个Environment。

这个Agent会有Observation看到世界种种变化，这个Observation又叫做State，这个State指的是环境的状态，也就是你的machine所看到的东西。我们的state能够观察到一部分的情况，机器没有办法看到环境所有的状态，这个state其实就是Observation。

machine会做一些事情，它做的事情叫做Action，Action会影响环境，会跟环境产生一些互动。因为它对环境造成的一些影响，它会得到Reward，这个Reward告诉它，它的影响是好的还是不好的。

举个例子，比如机器看到一杯水，然后它就take一个action，这个action把水打翻了，Environment就会得到一个negative的reward，告诉它不要这样做，它就得到一个负向的reward。在Reinforcement Learning，这些动作都是连续的，因为水被打翻了，接下来它看到的就是水被打翻的状态，它会take另外一个action，决定把它擦干净，Environment觉得它做得很对，就给它一个正向的reward。机器生来的目标就是要去学习采取哪些action，可以maximize reward

接着，以alpha go为例子，一开始machine的Observation是棋盘，棋盘可以用一个19*19的矩阵来描述，接下来，它要take一个action，这个action就是落子的位置。落子在不同的位置就会引起对手的不同反应，对手下一个子，Agent的Observation就变了。Agent看到另外一个Observation后，就要决定它的action，再take一个action，落子在另外一个位置。用机器下围棋就是这么个回事。在围棋这个case里面，还是一个蛮难的Reinforcement Learning，在多数的时候，你得到的reward都是0，落子下去通常什么事情也没发生这样子。只有在你赢了，得到reward是1，如果输了，得到reward是-1。Reinforcement Learning困难的地方就是有时候你的reward是sparse的，即在只有少数的action有reward的情况下挖掘正确的action。

对于machine来说，它要怎么学习下围棋呢，就是找一某个对手一直下下，有时候输有时候赢，它就是调整Observation和action之间的关系，调整model让它得到的reward可以被maximize。

Agent learns to take actions maximizing expected reward.

Supervised v.s. Reinforcement

- Supervised: Learning from teacher



Next move:
“5-5”



Next move:
“3-3”

- Reinforcement Learning Learning from experience

First move → many moves → Win!

(Two agents play with each other.)

Alpha Go is supervised learning + reinforcement learning.

我们可以比较下下围棋采用Supervised 和Reinforcement 有什么区别。如果是Supervised 你就是告诉机器说看到什么样的盘势就落在指定的位置。

Supervised不足的地方就是：具体盘势下落在哪个地方是最好的，其实人也不知道，因此不太容易做 Supervised。机器可以看着棋谱学，但棋谱上面的这个应对不见得是最 optimal的，所以用 Supervised learning 可以学出一个会下围棋的 Agent，但它可能不是真正最厉害的 Agent。

如果是Reinforcement 呢，就是让机器找一个对手不断下下，赢了就获得正的reward，没有人告诉它之前哪几步下法是好的，它要自己去试，去学习。Reinforcement 是从过去的经验去学习，没有老师告诉它什么是好的，什么是不好的，machine要自己想办法知道。

其实在做Reinforcement 这个task里面，machine需要大量的training，可以两个machine互相下。alpha Go 是先做Supervised Learning，做得不错再继续做Reinforcement Learning。

Learning a chat-bot

Reinforcement Learning 就是让机器去跟人讲话，讲讲人就生气了，machine就知道一句话可能讲得不太好。不过没人告诉它哪一句话讲得不好，它要自己去发掘这件事情。

这个想法听起来很crazy，但是真正有chat-bot是这样做的，这个怎么做呢？因为你要让machine不断跟人讲话，看到人生气后进行调整，去学怎么跟人对话，这个过程比较漫长，可能得好几百万次对话之后才能学会。这个不太现实，那么怎么办呢，就用Alpha Go的方式，Learning 两个agent，然后让它们互讲的方式。

两个chat-bot互相对话，对话之后有人要告诉它们它们讲得好还是不好。

在围棋里比较简单，输赢是比较明确的，对话的话就比较麻烦，你可以让两个machine进行无数轮互相对话，问题是不知道它们这聊天聊得好还是不好，这是一个待解决问题。

现有的方式是制定几条规则，如果讲得好就给它positive reward，讲得不好就给它negative reward，好不好由人主观决定，然后machine就从它的reward中去学说它要怎么讲才是好。后续可能会有人用GAN的方式去学chat-bot。通过discriminator判断是否像人对话，两个agent就会想骗过discriminator，即用discriminator自动learn出给reward的方式。

Reinforcement Learning 有很多应用，尤其是人也不知道怎么做的场景非常适合。

Interactive retrieval

让machine学会做Interactive retrieval，意思就是说有一个搜寻系统，能够跟user进行信息确认的方式，从而搜寻到user所需要的信息。直接返回user所需信息，它会得到一个positive reward，然后每问一个问题，都会得到一个negative reward。

More applications

Reinforcement Learning 还有很多应用，比如开个直升机，开个无人车呀，据说最近 DeepMind 用 Reinforcement Learning 的方法来帮 Google 的 server 节电，也有文本生成等。

现在Reinforcement Learning最常用的场景是电玩。现在有现成的environment，比如Gym，Universe。

让machine 用Reinforcement Learning来玩游戏，跟人一样，它看到的东西就是一幅画面，就是pixel，然后看到画面，它要做什么事情它自己决定，并不是写程序告诉它说你看到这个东西要做什么。需要它自己去学出来。

Playing Video Game

- Space invader

游戏的终止条件是所有的外星人被消灭或者你的太空飞船被摧毁。

这个游戏里面，你可以take的actions有三个，可以左右移动跟开火。

machine会看到一个observation，这个observation就是一幕画面。一开始machine看到一个observation s_1 ，这个 s_1 其实就是一个matrix，因为它有颜色，所以是一个三维的pixel。machine看到这个画面以后，就要决定它take什么action，现在只有三个action可以选择。比如它take 往右移。每次machine take一个action以后，它会得到一个reward，这个reward就是左上角的分数。往右移不会得到任何的reward，所以得到的reward $r_1 = 0$ ，machine 的action会影响环境，所以machine看到的observation就不一样了。现在observation为 s_2 ，machine自己往右移了，同时外星人也有点变化了，这个跟machine的action是没有关系的，有时候环境会有一些随机变化，跟machine无关。machine看到 s_2 之后就要决定它要take哪个action，假设它决定要射击并成功的杀了一只外星人，就会得到一个reward，杀不同的外星人，得到的分数是不一样的。假设杀了一只5分的外星人，这个observation就变了，少了一只外星人。

这个过程会一直进行下去，直到某一天在第 T 个回合的时候，machine take action a_T ，然后他得到的 reward r_T 进入了另外一个 state，这个 state 是个 terminal state，它会让游戏结束。可能这个machine往左移，不小心碰到alien的子弹，就死了，游戏就结束了。从这个游戏的开始到结束，就是一个**episode**，machine要做的事情就是不断的玩这个游戏，学习怎么在一个episode里面怎么去maximize reward，在死之前杀最多的外星人同时要闪避子弹，让自己不会被杀死。

Difficulties of Reinforcement Learning

那么Reinforcement Learning的难点在哪里呢？它有两个难点

- Reward delay

第一个难点是，reward出现往往会有存在delay，比如在space invader里面只有开火才会得到reward，但是如果machine只知道开火以后就会得到reward，最后learn出来的结果就是它只会乱开火。对它来说，往左往右移没有任何reward。事实上，往左往右这些moving，它对开火是否能够得到reward是有关键影响的。虽然这些往左往右的action，本身没有办法让你得到任何reward，但它帮助你在未来得到reward，就像规划未来一样，machine需要有这种远见，要有这种vision，才能玩好。在下围棋里面，有时候也是一样的，短期的牺牲可以换来最好的结果。

- Agent's actions affect the subsequent data it receives

Agent采取行动后会影响之后它所看到的东西，所以Agent要学会去探索这个世界。比如说在这个space invader里面，Agent只知道往左往右移，它不知道开火会得到reward，也不会试着击杀最上面的外星人，就不会知道击杀这个东西可以得到很高的reward，所以要让machine去explore它没有做过的行为，这个行为可能会有好的结果也会有坏的结果。但是探索没有做过的行为在Reinforcement Learning里面也是一种重要的行为。

Outline

Reinforcement Learning 其实有一个 typical 的讲法，要先讲 Markov Decision Process，在 Reinforcement Learning 里面很红的一个方法叫 Deep Q Network，今天也不讲 Deep Q Network，现在最强的方法叫 A3C，所以我想说不如直接来讲 A3C，直接来讲最新的东西。

Approach

Reinforcement Learning 的方法分成两大块，一个是Policy-based的方法，另一个是Valued-based的方法。先有Valued-based的方法，再有Policy-based的方法，所以一般教科书都是讲 Value-based 的方法比较多。

在Policy-based的方法里面，会learn一个负责做事的Actor，在Valued-based的方法会learn一个不做事的Critic。我们要把Actor和Critic加起来叫做Actor+Critic的方法。

现在最强的方法就是Asynchronous Advantage Actor-Critic(A3C)。Alpha Go是各种方法大杂烩，有Policy-based的方法，有Valued-based的方法，有model-based的方法。下面是一些学习deep Reinforcement Learning的资料

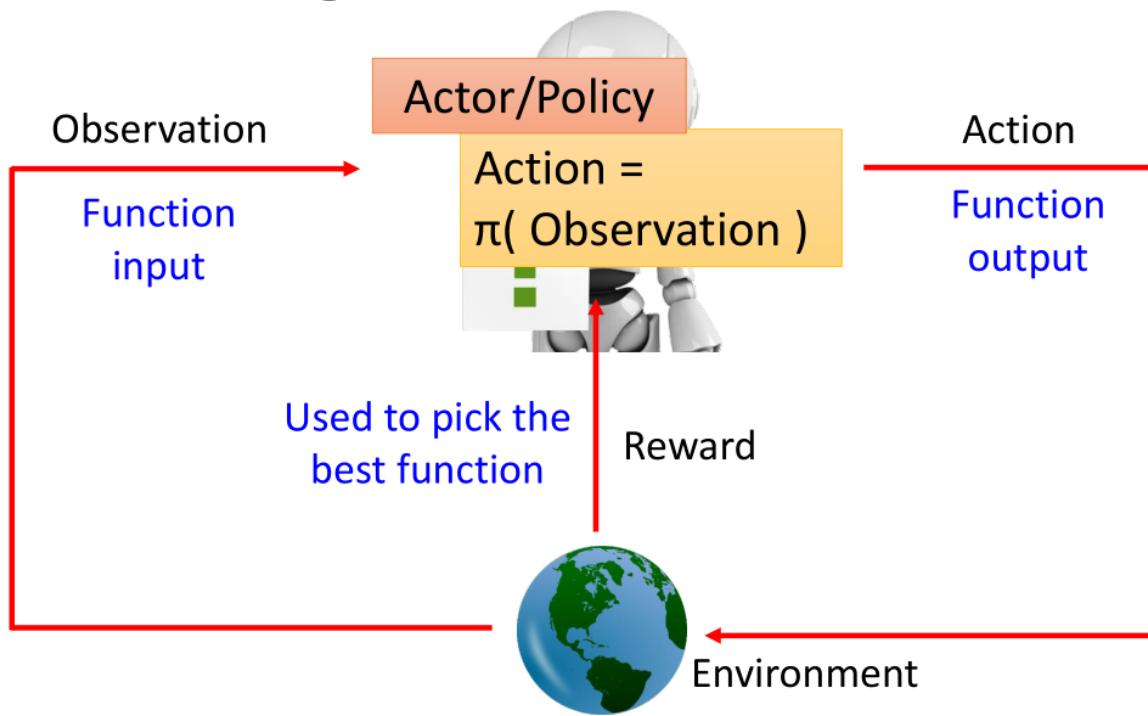
Reference

- Textbook: Reinforcement Learning: An Introduction
<https://webdocs.cs.ualberta.ca/~sutton/book/the-book.html>
- Lectures of David Silver
<http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html> (10 lectures, 1:30 each)
http://videolectures.net/rldm2015_silver_reinforcement_learning/ (Deep Reinforcement Learning)
- Lectures of John Schulman
https://youtu.be/aUrX-rP_ss4

Policy-based Approach

先来看看怎么学一个Actor，所谓的Actor是什么呢？我们之前讲过，Machine Learning 就是找一个 Function，Reinforcement Learning也是Machine Learning 的一种，所以要做的事情也是找Function。Actor就是一个Function π ，这个Function的input就是Machine看到的observation，它的output就是Machine要采取的Action。我们要通过reward来帮我们找这个best Function。

Machine Learning ≈ Looking for a Function



找个这个Function有三个步骤：

Neural Network as Actor

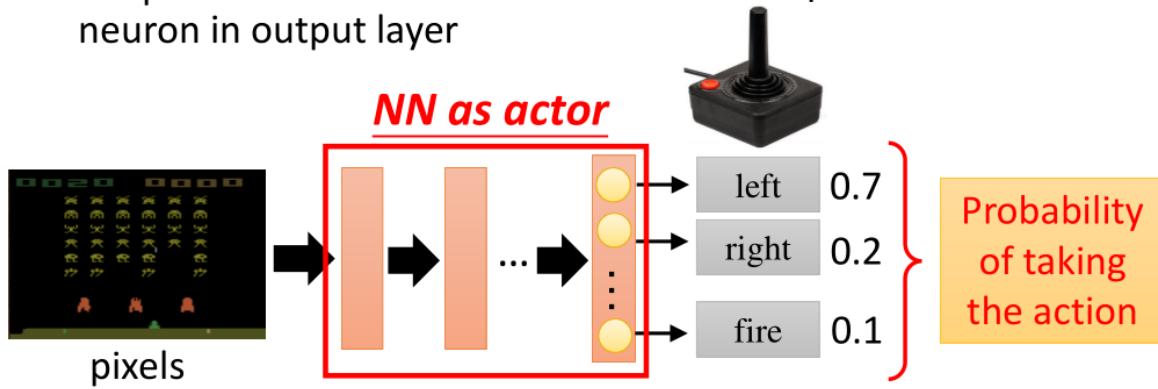
第一个步骤就是决定你的Function长什么样子，假设你的Function是一个Neural Network，就是一个deep reinforcement learning。

如果Neural Network作为一个Actor，这个Neural Network的输入就是observation，可以通过一个vector或者一个matrix 来描述。output就是你现在可以采取的action。

举个例子，Neural Network作为一个Actor，input是一张image，output就是你现在有几个可以采取的action，output就有几个dimension。假设我们在玩Space invader，output就是可能采取的action左移、右移和开火，这样output就有三个dimension分别代表了左移、右移和开火。

这个Neural Network怎么决定这个Actor要采取哪个action呢？通常做法是这样，把 image 丢到 Neural Network 里面去，他就会告诉你每一个 output 的 dimension 也就是每一个 action 所对应的分数。你可以采取分数最高的 action，比如说 left 分数最高，假设已经找好这个 Actor，machine 看到这个画面他可能就采取 left。

- Input of neural network: the observation of machine represented as a vector or a matrix
- Output neural network : each action corresponds to a neuron in output layer



What is the benefit of using network instead of lookup table?

generalization

但是做 Policy Gradient 的时候，通常会假设Policy 是 stochastic，所谓的 stochastic 的意思是你的 Policy 的 output 其实是个机率，如果你的分数是 0.7、0.2 跟 0.1，有 70% 的机率会 left，有 20% 的机率会 right，10% 的机率会 fire，看到同样画面的时候，根据机率，同一个 Actor 会采取不同的 action。这种 stochastic 的做法其实很多时候是会有好处的，比如说要玩猜拳，如果 Actor 是 deterministic，可能就只会一直输，所以有时候会需要 stochastic 这种 Policy。在底下的 lecture 里面都假设 Actor 是 stochastic 的。

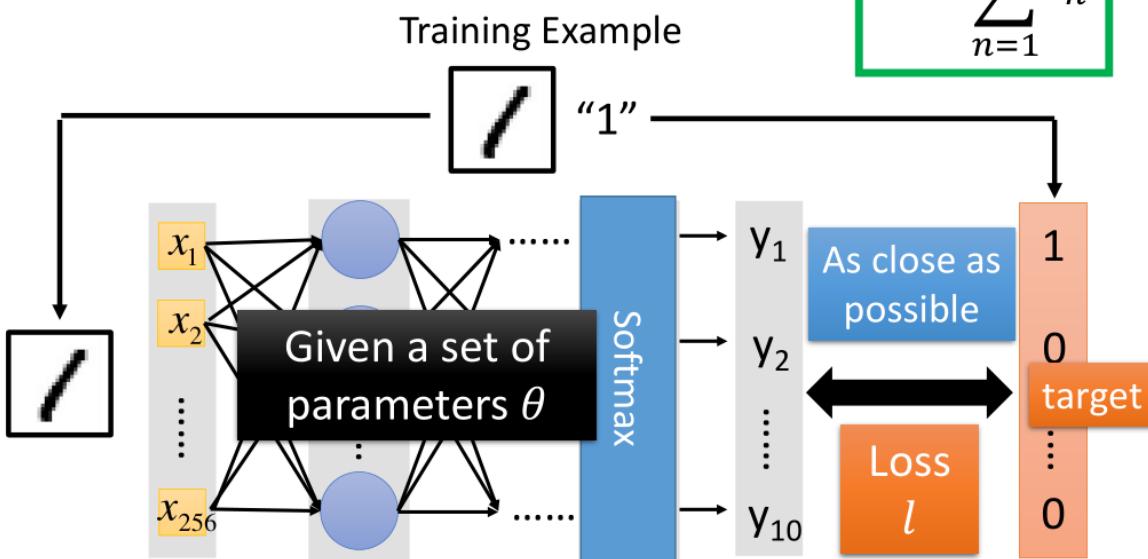
用 Neural Network 来当 Actor 有什么好处？传统的作法是直接存一个 table，这个 table 告诉我看到这个 observation 就采取这个 action，看到另外一个 observation 就采取另外一个 action。但这种作法要玩电玩是不行的，因为电玩的 input 是 pixel，要穷举所有可能 pixel 是没有办法做到的，所以一定要用 Neural Network 才能够让 machine 把电玩玩好。用 Neural Network 的好处就是 Neural Network 可以举一反三，就算有些画面完全没有看过，因为 Neural Network 的特性，input 一个东西总是会有 output，就算是他没有看过的东西，他也有可能得到一个合理的结果，用 Neural Network 的好处是他比较 generalize。

Goodness of Actor

第二步骤就是，我们要决定一个Actor的好坏。在Supervised learning中，我们是怎样决定一个Function的好坏呢？假设给一个 Neural Network，参数假设已经知道就是 θ ，有一堆 training example，假设在做 image classification，就把 image 丢进去看 output 跟 target 像不像，如果越像的话这个Function就会越好，定义一个东西叫做 Loss，算每一个 example 的 Loss，合起来就是 Total Loss。需要找一个参数去 minimize 这个 Total Loss。

Goodness of Actor

- Review: Supervised learning



在Reinforcement Learning里面，一个Actor的好坏的定义是非常类似的。假设我们现在有一个Actor，这个Actor就是一个Neural Network。

Neural Network的参数是 θ ，即一个Actor可以表示为 $\pi_\theta(s)$ ，它的input就是Machine看到的observation。

那怎么知道一个Actor表现得好还是不好呢？我们让这个Actor实际的去玩一个游戏，玩完游戏得到的total reward为 $R_\theta = \sum_{t=1}^T r_t$ ，把每个时间得到的reward合起来，这就是一个episode里面，你得到的total reward。

这个total reward是我们需要去maximize的对象。我们不需要去maximize 每个step的reward，我们是要maximize 整个游戏玩完之后的total reward。

假设我们拿同一个Actor，每次玩的时候， R_θ 其实都会不一样的。因为两个原因，首先 Actor 如果是stochastic，看到同样的场景也会采取不同的 action。所以就算是同一个Actor，同一组参数，每次玩的时候你得到的 R_θ 也会不一样的。游戏本身也有随机性，就算你采取同一个Action，你看到的observation每次也可能都不一样。所以 R_θ 是一个Random Variable。我们做的事情，不是去maximize某一次玩游戏时的 R_θ ，而是去maximize R_θ 的期望值。这个期望值就衡量了某一个Actor的好坏，好的Actor期望值就应该要比较大。

An episode is considered as a trajectory τ

- $\tau = \{s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_T, a_T, r_T\}$
- $R(\tau) = \sum_{t=1}^T r_t$
- If you use an actor to play the game, each τ has a probability to be sampled
 - The probability depends on actor parameter θ : $P(\tau|\theta)$

$$\bar{R}_\theta = \sum_{\tau} R(\tau) P(\tau|\theta) \approx \frac{1}{N} \sum_{n=1}^N R(\tau^n)$$

Sum over all possible trajectory

Use π_θ to play the game N times, obtain $\{\tau^1, \tau^2, \dots, \tau^N\}$

Sampling τ from $P(\tau|\theta)$ N times

那么怎么计算呢，我们假设一场游戏就是一个trajectory τ

$$\tau = \{s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_T, a_T, r_T\}$$

τ 包含了state(observation)，看到这个 observation 以后take的Action，得到的Reward，是一个 sequence。

$$R(\tau) = \sum_{n=1}^N r_n$$

$R(\tau)$ 代表在这个episode里面，最后得到的总reward。

当我们用某一个Actor去玩这个游戏的时候，每个 τ 都会有出现的机率， τ 代表从游戏开始到结束过程，这个过程有千百万种。当你选择这个Actor的时候，你可能只会看到某一些过程，某些过程特别容易出现，某些过程比较不容易出现。每个游戏出现的过程，可以用一个机率 $P(\tau|\theta)$ 来表示它，就是说参数是 θ 时 τ 这个过程出现的机率。

那么 R_θ 的期望值为

$$\bar{R}_\theta = \sum_{\tau} R(\tau) P(\tau|\theta)$$

实际上要穷举所有的 τ 是不可能的，那么要怎么做？让Actor去玩N场这个游戏，获得N个过程 $\tau^1, \tau^2, \dots, \tau^N$ ，玩N场就好像从 $P(\tau|\theta)$ 去 Sample N个 τ 。假设某个 τ 它的机率特别大，就特别容易被 sample出来。让Actor去玩N场，相当于从 $P(\tau|\theta)$ 概率场抽取N个过程，可以通过N个Reward的均值进行近似，如下表达

$$\bar{R}_\theta = \sum_{\tau} R(\tau) P(\tau|\theta) \approx \frac{1}{N} R(\tau^n)$$

Pick the best function

怎么选择最好的function，其实就是用我们的Gradient Ascent。我们已经找到目标了，就是最大化这个 \bar{R}_θ

$$\theta^* = \arg \max_{\theta} \bar{R}_\theta \quad \bar{R}_\theta = \sum_{\tau} R(\tau) P(\tau|\theta)$$

就可以用Gradient Ascent进行最大化，过程为：

$$\begin{aligned}
&\text{start with } \theta^0 \\
\theta^1 &\leftarrow \theta^0 + \eta \nabla \bar{R}_{\theta^0} \\
\theta^2 &\leftarrow \theta^1 + \eta \nabla \bar{R}_{\theta^1} \\
&\dots
\end{aligned}$$

参数 $\theta = w_1, w_2, \dots, b_1, \dots$, 那么 $\nabla \bar{R}_\theta$ 就是 \bar{R}_θ 对每个参数的偏微分, 如下

$$\nabla \bar{R}_\theta = \begin{bmatrix} \partial \bar{R}_\theta / \partial w_1 \\ \partial \bar{R}_\theta / \partial w_2 \\ \vdots \\ \bar{R}_\theta / \partial b_1 \\ \vdots \end{bmatrix}$$

实际的计算中

$\bar{R}_\theta = \sum_\tau R(\tau) P(\tau|\theta)$ 中, 只有 $P(\tau|\theta)$ 跟 θ 有关系, 所以只需要对 $P(\tau|\theta)$ 做 Gradient, 即

$$\nabla \bar{R}_\theta = \sum_\tau R(\tau) \nabla P(\tau|\theta)$$

所以 $R(\tau)$ 就算不可微也没有关系, 或者是不知道它的 function 也可以, 我们只要知道把 τ 放进去得到值就可以。

接下来, 为了让 $P(\tau|\theta)$ 出现

$$\nabla \bar{R}_\theta = \sum_\tau R(\tau) \nabla P(\tau|\theta) = \sum_\tau R(\tau) P(\tau|\theta) \frac{\nabla P(\tau|\theta)}{P(\tau|\theta)}$$

由于

$$\frac{d \log(f(x))}{dx} = \frac{1}{f(x)} \frac{df(x)}{dx}$$

所以

$$\nabla \bar{R}_\theta = \sum_\tau R(\tau) P(\tau|\theta) \frac{\nabla P(\tau|\theta)}{P(\tau|\theta)} = \sum_\tau R(\tau) P(\tau|\theta) \nabla \log P(\tau|\theta)$$

从而可以通过抽样的方式去近似, 即

$$\nabla \bar{R}_\theta = \sum_\tau R(\tau) P(\tau|\theta) \nabla \log P(\tau|\theta) \approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log P(\tau^n|\theta)$$

即拿 θ 去玩 N 次游戏, 得到 $\tau^1, \tau^2, \dots, \tau^N$, 算出每次的 $R(\tau)$ 。

Policy Gradient

$$\bar{R}_\theta = \sum_{\tau} R(\tau)P(\tau|\theta) \quad \nabla \bar{R}_\theta = ?$$

$$\nabla \bar{R}_\theta = \sum_{\tau} R(\tau)\nabla P(\tau|\theta) = \sum_{\tau} R(\tau)P(\tau|\theta) \frac{\nabla P(\tau|\theta)}{P(\tau|\theta)}$$

$R(\tau)$ do not have to be differentiable
It can even be a black box.

$$= \boxed{\sum_{\tau}} R(\tau) \boxed{P(\tau|\theta)} \nabla \log P(\tau|\theta) \quad \boxed{\frac{d \log(f(x))}{dx} = \frac{1}{f(x)} \frac{df(x)}{dx}}$$

$$\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log P(\tau^n|\theta)$$

Use π_θ to play the game N times,
Obtain $\{\tau^1, \tau^2, \dots, \tau^N\}$

接下来的问题是怎么计算 $\nabla \log P(\tau^n|\theta)$, 因为

$$\begin{aligned} P(\tau|\theta) &= p(s_1)p(a_1|s_1, \theta)p(r_1, s_2|s_1, a_1)p(a_2|s_2, \theta)p(r_2, s_3|s_2, a_2)\dots \\ &= p(s_1) \prod_{t=1}^T p(a_t|s_t, \theta)p(r_t, s_{t+1}|s_t, a_t) \end{aligned}$$

其中 $p(s_1)$ 是初始状态出现的机率, 接下来根据 θ 会有某个概率在 s_1 状态下采取 Action a_1 , 然后根据 a_1, s_1 会得到某个 reward r_1 , 并跳到另一个 state s_2 , 以此类推。其中 $p(s_1)$ 和 $p(r_t, s_{t+1}|s_t, a_t)$ 跟 Actor 是无关的, 只有 $p(a_t|s_t, \theta)$ 跟 Actor π_θ 有关系。

Goodness of Actor

We define \bar{R}_θ as the expected value of R_θ

- $\tau = \{s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_T, a_T, r_T\}$

$$P(\tau|\theta) = p(s_1)p(a_1|s_1, \theta)p(r_1, s_2|s_1, a_1)p(a_2|s_2, \theta)p(r_2, s_3|s_2, a_2)\dots$$

$$= p(s_1) \prod_{t=1}^T p(a_t|s_t, \theta)p(r_t, s_{t+1}|s_t, a_t)$$

not related to your actor
 Control by your actor π_θ

$p(a_t = "fire" | s_t, \theta) = 0.7$

$s_t \rightarrow$

Actor π_θ	left $\rightarrow 0.1$
	right $\rightarrow 0.2$
	fire $\rightarrow 0.7$

Policy Gradient

$$\nabla \log P(\tau|\theta) = ?$$

- $\tau = \{s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_T, a_T, r_T\}$

$$\begin{aligned}
 P(\tau|\theta) &= p(s_1) \prod_{t=1}^T p(a_t|s_t, \theta) p(r_t, s_{t+1}|s_t, a_t) \\
 \log P(\tau|\theta) &= \log p(s_1) + \sum_{t=1}^T \log p(a_t|s_t, \theta) + \log p(r_t, s_{t+1}|s_t, a_t) \\
 \nabla \log P(\tau|\theta) &= \sum_{t=1}^T \nabla \log p(a_t|s_t, \theta)
 \end{aligned}$$

Ignore the terms
not related to θ

通过取 \log , 连乘转为连加, 即

$$\log P(\tau|\theta) = \log p(s_1) + \sum_{t=1}^T \log p(a_t|s_t, \theta) + \log p(r_t, s_{t+1}|s_t, a_t)$$

然后对 θ 取Gradient, 删去无关项, 得到

$$\nabla \log P(\tau|\theta) = \sum_{t=1}^T \nabla \log p(a_t|s_t, \theta)$$

则

$$\begin{aligned}
 \nabla \bar{R}_\theta &\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log P(\tau^n|\theta) = \frac{1}{N} \sum_{n=1}^N R(\tau^n) \sum_{t=1}^{T_n} \nabla \log p(a_t^n|s_t^n, \theta) \\
 &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p(a_t^n|s_t^n, \theta)
 \end{aligned}$$

这个式子就告诉我们, 当我们在某一次 τ^n 游戏中, 在 s_t^n 状态下采取 a_t^n 得到 $R(\tau^n)$ 是正的, 我们就希望 θ 能够使 $p(a_t^n|s_t^n)$ 的概率越大越好。反之, 如果 $R(\tau^n)$ 是负的, 就要调整 θ 参数, 能够使 $p(a_t^n|s_t^n)$ 的机率变小。

Policy Gradient

$$\theta^{new} \leftarrow \theta^{old} + \eta \nabla \bar{R}_{\theta^{old}}$$

$$\begin{aligned} & \nabla \log P(\tau | \theta) \\ &= \sum_{t=1}^T \nabla \log p(a_t | s_t, \theta) \end{aligned}$$

$$\begin{aligned} \nabla \bar{R}_\theta &\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log P(\tau^n | \theta) = \frac{1}{N} \sum_{n=1}^N R(\tau^n) \sum_{t=1}^{T_n} \nabla \log p(a_t^n | s_t^n, \theta) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p(a_t^n | s_t^n, \theta) \end{aligned}$$

What if we replace
 $R(\tau^n)$ with r_t^n

If in τ^n machine takes a_t^n when seeing s_t^n in

- $R(\tau^n)$ is positive  Tuning θ to increase $p(a_t^n | s_t^n)$
- $R(\tau^n)$ is negative  Tuning θ to decrease $p(a_t^n | s_t^n)$

It is very important to consider the cumulative reward $R(\tau^n)$ of the whole trajectory τ^n instead of immediate reward r_t^n

注意，某个时间点的 $p(a_t^n | s_t^n, \theta)$ 是乘上这次游戏的所有 reward $R(\tau^n)$ 而不是这个时间点的 reward。假设我们只考虑这个时间点的 reward，那么就是说只有 fire 才能得到 reward，其他的 action 你得到的 reward 都是 0。Machine 就只会增加 fire 的机率，不会增加 left 或者 right 的机率。最后 Learn 出来的 Agent 它就只会 fire。

接着还有一个问题，为什么要取 log 呢？

$$\nabla \log p(a_t^n | s_t^n, \theta) = \frac{\nabla p(a_t^n | s_t^n, \theta)}{p(a_t^n | s_t^n, \theta)}$$

那么为什么要除以 $p(a_t^n | s_t^n, \theta)$ 呢？

假设现在让 machine 去玩 N 次游戏，那某一个 state 在第 13 次、第 15 次、第 17 次、第 33 次的游戏， $\tau^{13}, \tau^{15}, \tau^{17}, \tau^{33}$ 里面看到了同一个 observation。因为 Actor 其实是 stochastic，所以它有个机率，所以看到同样的 s，不见得采取同样 action，所以假设在第 13 个 trajectory，它采取 action a，在第 17 个它采取 b，在 15 个采取 b，在 33 也采取 b，最后 τ^{13} 的这个 trajectory 得到的 reward 比较大是 2，另外三次得到的 reward 比较小。

但实际上在做 update 的时候，它会偏好那些出现次数比较多的 action，就算那些 action 并没有真的比较好。

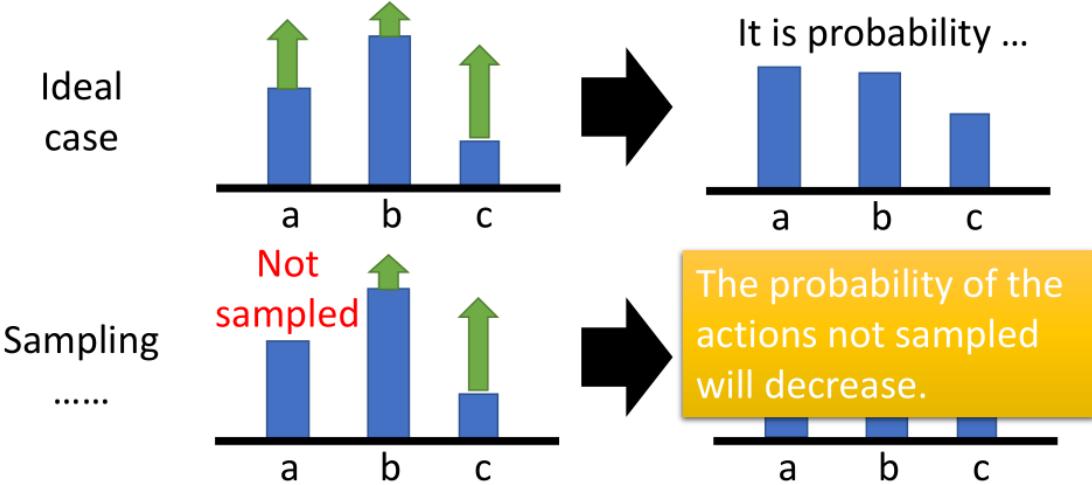
因为是 summation over 所有 sample 到的结果，如果 take action b 这件事情，出现的次数比较多，就算它得到的 reward 没有比较大，machine 把这件事情的机率调高，也可以增加最后这一项的结果，虽然这个 action a 感觉比较好，但是因为它很罕见，所以调高这个 action 的机率，最后也不会对你要 maximize 的对象 Objective 的影响也是比较小的，machine 就会变成不想要 maximize action a 出现的机率，转而 maximize action b 出现的机率。这就是为什么这边需要除掉一个机率，除掉这个机率的好处就是做一个 normalization，如果有某一个 action 它本来出现的机率就比较高，它除掉的值就比较大，让它除掉一个比较大的值，machine 最后在 optimize 的时候，就不会偏好那些机率出现比较高的 action。

Add a Baseline

It is possible that $R(\tau^n)$ is always positive.

$$\theta^{new} \leftarrow \theta^{old} + \eta \nabla \bar{R}_{\theta^{old}}$$

$$\nabla \bar{R}_{\theta} \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p(a_t^n | s_t^n, \theta)$$



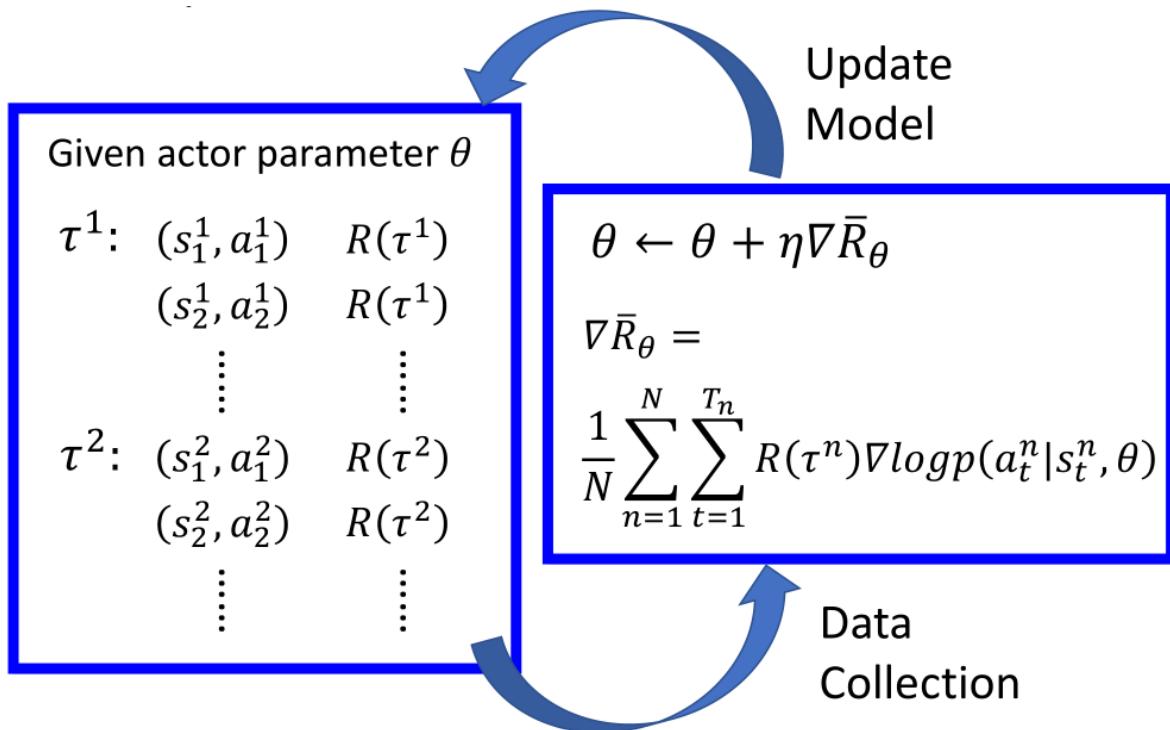
还有另外一个问题，假设 $R(\tau^n)$ 总是正的，那么会出现什么事情呢？在理想的状态下，这件事情不会构成任何问题。假设有三个action, a, b, c 采取的结果得到的reward都是正的，这个正有大有小，假设 a 和 c 的 $R(\tau^n)$ 比较大， b 的 $R(\tau^n)$ 比较小，经过update之后，你还是会让 b 出现的机率变小， a, c 出现的机率变大，因为会做normalization。但是实做的时候，我们做的事情是sampling，所以有可能只sample b 和 c ，这样 b, c 机率都会增加， a 没有sample到，机率就自动减少，这样就会有问题了。

这样，我们就希望 $R(\tau^n)$ 有正有负这样，可以通过将 $R(\tau^n) - b$ 来避免， b 需要自己设计。如下

$$\nabla \bar{R}_{\theta} \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p(a_t^n | s_t^n, \theta)$$

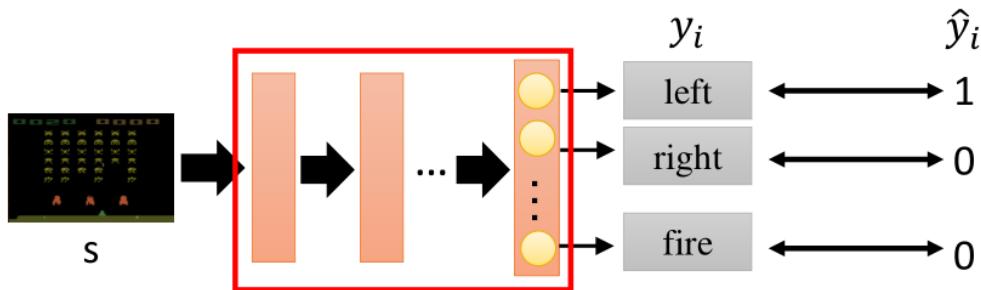
这样 $R(\tau^n)$ 超过 b 的时候就把机率增加，小于 b 的时候就把机率降低，不会造成没被sample到的action机率会减小。

Policy Gradient



可以把训练过程看成多个分类网络的训练过程，优化目标一致。实作上也一样。

Considered as Classification Problem



$$\text{Minimize: } - \sum_{i=1}^3 \hat{y}_i \log y_i$$

y_i	\hat{y}_i
left	1
right	0
fire	0

$$\text{Maximize: } \log y_i =$$

$$\log P(\text{"left"}|s)$$

$$\theta \leftarrow \theta + \eta \nabla \log P(\text{"left"}|s)$$

Policy Gradient

Given actor parameter θ

$$\tau^1: (s_1^1, a_1^1) \quad R(\tau^1)$$

$$(s_2^1, a_2^1) \quad R(\tau^1)$$

\vdots

$$\tau^2: (s_1^2, a_1^2) \quad R(\tau^2)$$

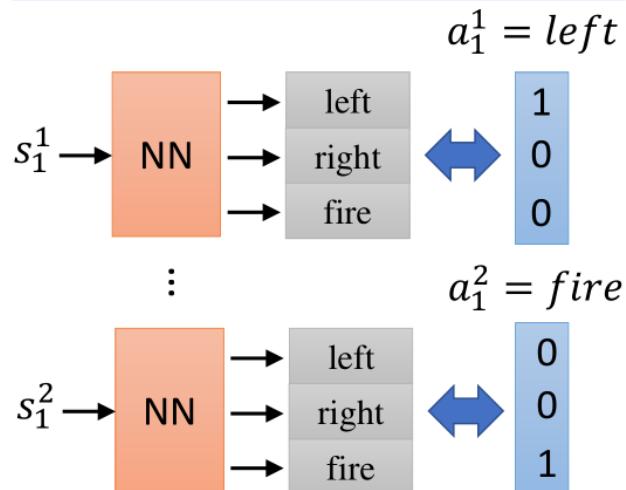
$$(s_2^2, a_2^2) \quad R(\tau^2)$$

\vdots

$$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta$$

$$\nabla \bar{R}_\theta =$$

$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \text{[Blue Box]} \nabla \log p(a_t^n | s_t^n, \theta)$$



Policy Gradient

Given actor parameter θ

$$\tau^1: (s_1^1, a_1^1) \quad R(\tau^1) \boxed{2}$$

$$(s_2^1, a_2^1) \quad R(\tau^1) \boxed{2}$$

\vdots

$$\tau^2: (s_1^2, a_1^2) \quad R(\tau^2) \boxed{1}$$

$$(s_2^2, a_2^2) \quad R(\tau^2) \boxed{1}$$

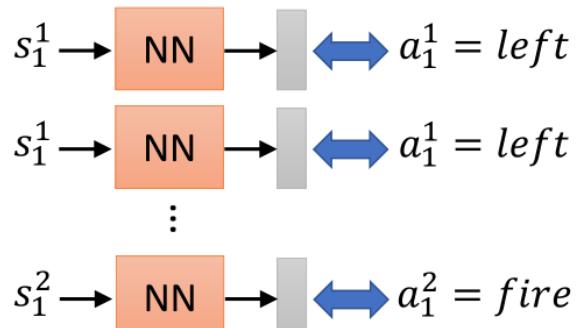
\vdots

$$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta$$

$$\nabla \bar{R}_\theta =$$

$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p(a_t^n | s_t^n, \theta)$$

Each training data is weighted by $R(\tau^n)$



Value-based Approach

Critic

Critic就是Learn一个Neural Network，这个Neural Network不做事。

A critic does not determine the action. Given an actor π , it evaluates the how good the actor is.

An actor can be found from a critic. e.g. Q-learning。其实也可以从 Critic 得到一个 Actor，这就是Q-learning。

Critic就是learn一个function，这个function可以告诉你说现在看到某一个observation的时候，这个observation有多好这样。

这个 Critic 其实有很多种，我们今天介绍 state value function

State value function $V^\pi(s)$

When using actor π , the cumulated reward expects to be obtained after seeing observation (state) s .

How to estimate $V^\pi(s)$

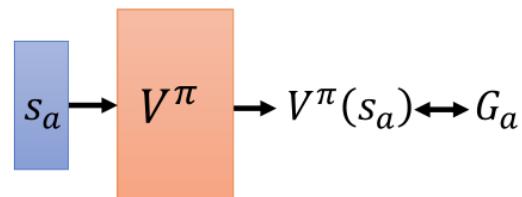
Monte-Carlo based approach

类似回归问题，训练时需要cumulated reward

- Monte-Carlo based approach
 - The critic watches π playing the game

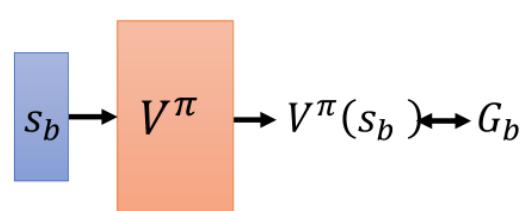
After seeing s_a ,

Until the end of the episode,
the cumulated reward is G_a



After seeing s_b ,

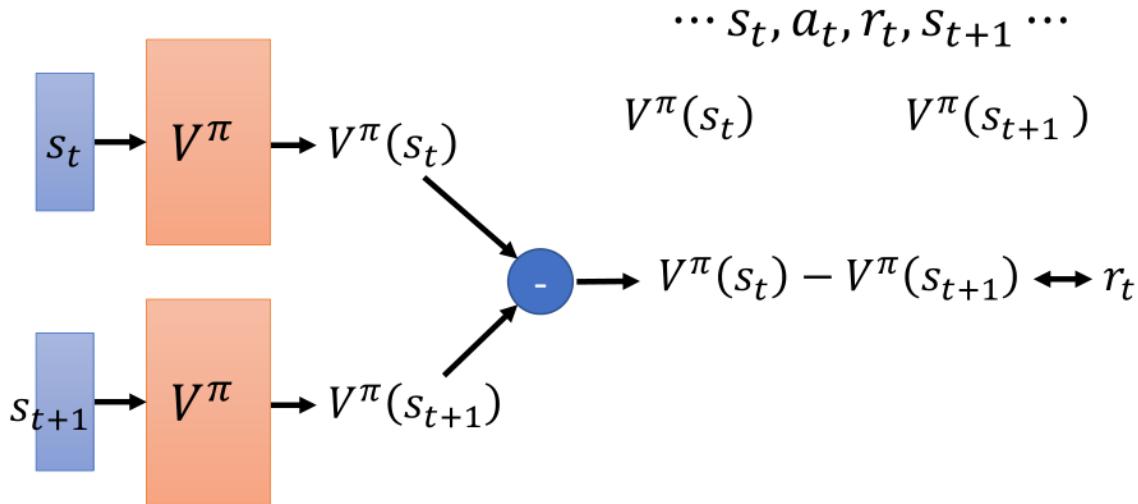
Until the end of the episode,
the cumulated reward is G_b



Temporal-difference approach

同样类似回归问题，训练时只需要让 s_{t+1} 和 s_t 中间差的 reward 接近 $r(t)$ ，输出仍然是游戏结束时的 reward

Temporal-difference approach

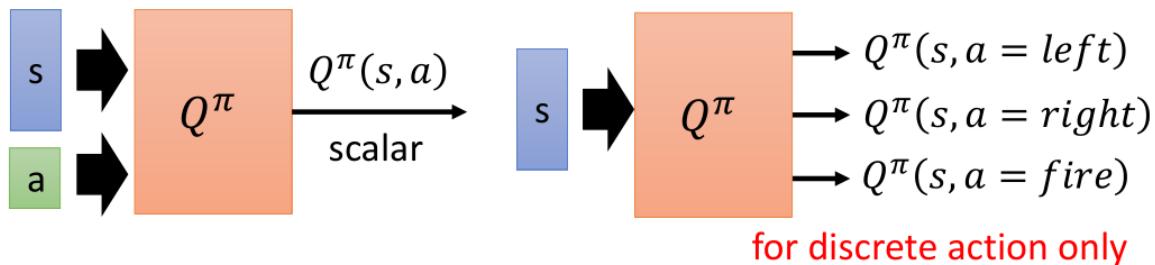


Some applications have very long episodes, so that delaying all learning until an episode's end is too slow.

State-action value function $Q^{\pi}(s, a)$

另外一种 critic, 它可以拿来决定 action, 这种 critic 我们叫做 Q function。它的 input 就是一个 state, 一个 action, output 是在这个 state 采取了 action a 的话, 到游戏结束的时候, 会得到多少 accumulated reward

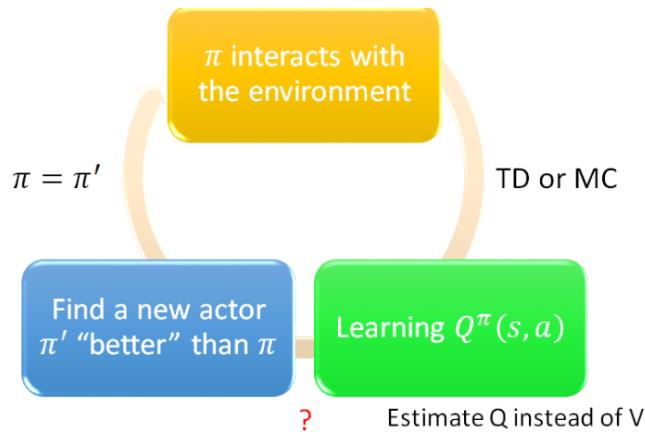
- State-action value function $Q^{\pi}(s, a)$
 - When using actor π , the *cumulated* reward expects to be obtained after seeing observation s and taking a



有时候我们会改写这个 Q function, 假设你的 a 是可以穷举的, 你只要输入一个 state s , 你就可以知道说, 所有action的情况下, 输出分数是多少。它的妙用是这个样子, 你可以用 Q function 找出一个比较好的 actor。这一招就叫做 Q learning。

DQN (Deep Q-Learning)

Q-Learning



- Given $Q^\pi(s, a)$, find a new actor π' “better” than π
 - “Better”: $V^{\pi'}(s) \geq V^\pi(s)$, for all state s

$$\pi'(s) = \arg \max_a Q^\pi(s, a)$$

- π' does not have extra parameters. It depends on Q
- Not suitable for continuous action a

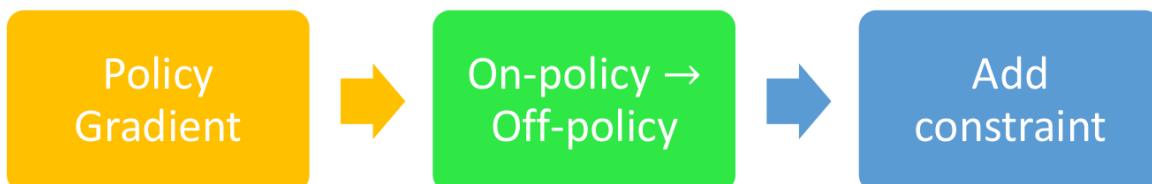
Actor-Critic

Inverse Reinforcement Learning

用 inverse reinforcement learning 的方法去推出 reward function, 再用 reinforcement learning 的方法去找出最好的 actor

Proximal Policy Optimization (PPO)

我们要讲一个 policy gradient 的进阶版叫做 Proximal Policy Optimization (PPO), 这个技术是 default reinforcement learning algorithm at OpenAI, 所以今天假设你要 implement reinforcement learning, 也许这是一个第一个你可以尝试的方法。

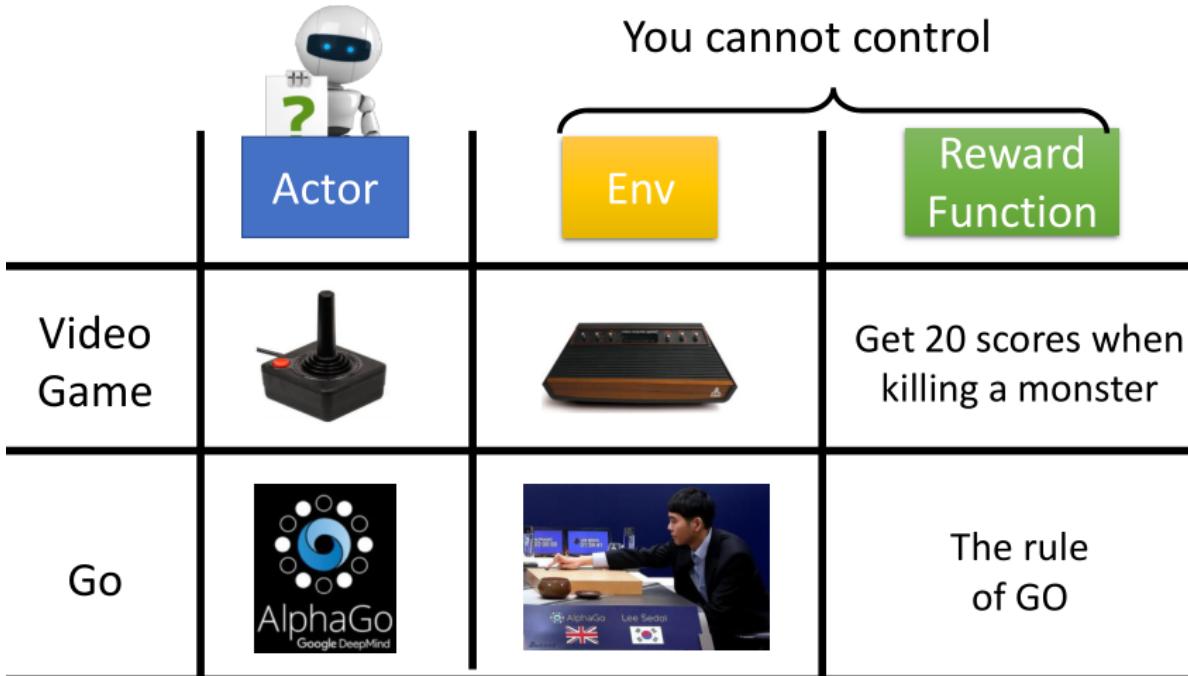


Policy Gradient (Review)

那我们就来先复习一下 policy gradient, PPO 是 policy gradient 一个变形, 所以我们先讲 policy gradient。

Basic Components

在 reinforcement learning 里面呢有 3 个 components，一个 actor，一个 environment，一个 reward function。



让机器玩 video game，那这个时候你 actor 做的事情，就是去操控，游戏的游戏杆，，比如说向左向右，开火，等等。你的 environment 就是游戏的主机，负责控制游戏的画面，负责控制说，怪物要怎么移动，你现在要看到什么画面，等等。所谓的 reward function，就是决定，当你做什么事情，发生什么状况的时候，你可以得到多少分数，比如说杀一只怪兽，得到20 分等等。

那同样的概念，用在围棋上也是一样，actor 就是 alpha Go，它要决定，下哪一个位置，那你的 environment 呢，就是对手，你的 reward function 就是按照围棋的规则，赢就是得一分，输就是负一分等等。

那在 reinforcement 里面，你要记得说 environment 跟 reward function，不是你可以控制的，environment 跟 reward function 是在开始学习之前，就已经事先给定的。

你唯一能做的事情，是调整你的 actor，调整你 actor 里面的 policy，使得它可以得到最大的 reward，你可以调的只有 actor。environment 跟 reward function 是事先给定，你是不能够去动它的。

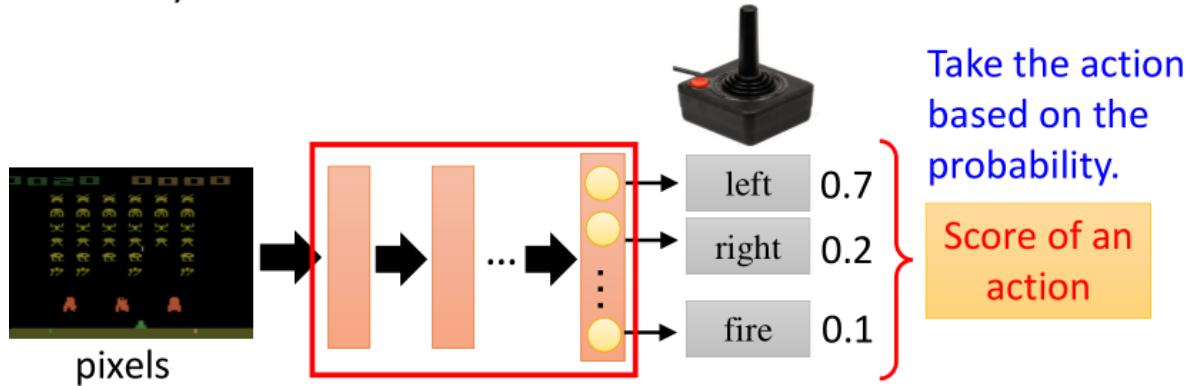
那这个 actor 里面，会有一个 policy，这个 policy 决定了 actor 的行为。那所谓的 policy 呢，就是给一个外界的输入，然后它会输出 actor 现在应该要执行的行为。

Policy of Actor

那今天假设你是用 deep learning 的技术来做 reinforcement learning 的话，那你的 policy，policy 我们一般写成 π ，policy 就是一个 network，那我们知道说，network 里面，就有一堆参数，我们用 θ 来代表 π 的参数。

你的 policy 它是一个 network，这个 network 的 input 它就是现在 machine 看到的东西，如果让 machine 打电玩的话，那 machine 看到的东西，就是游戏的画面，当然让 machine 看到什么东西，会影响你现在 training，到底好不好 train。举例来说，在玩游戏的时候，也许你觉得游戏的画面，前后是相关的，也许你觉得说，你应该让你的 policy，看从游戏初始，到现在这个时间点，所有画面的总和，你可能会觉得你要用到 RNN 来处理它，不过这样子，你会比较难处理就是了。那要让，你的 machine 你的 policy 看到什么样的画面，这个是你自己决定的。

- Policy π is a network with parameter θ
 - Input: the observation of machine represented as a vector or a matrix
 - Output: each action corresponds to a neuron in output layer



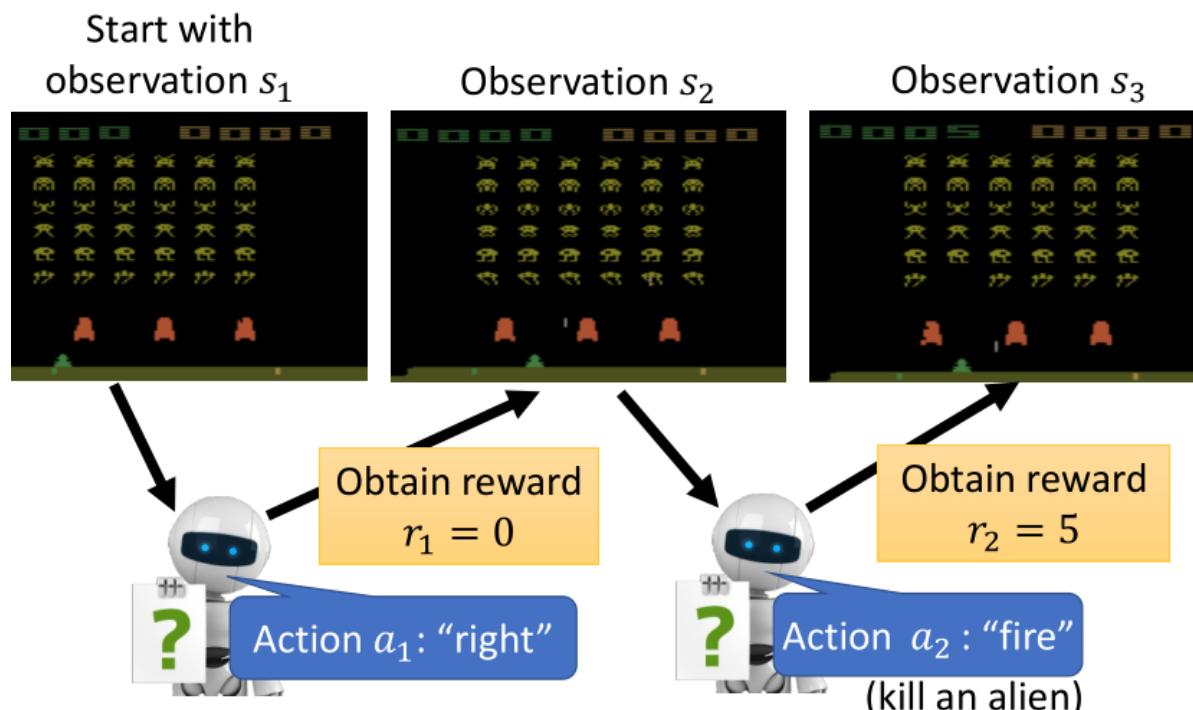
那在 output 的部分，输出的就是今天机器要采取什么样的行为，这边这个是具体的例子，你的 policy 就是一个 network，input 就是游戏的画面，那它通常就是由 pixels 所组成的，那 output 就是看看说现在有那些选项是你可以去执行的，那你的 output layer 就有几个 neurons，假设你现在可以做的行为就是有 3 个，那你的 output layer 就是有 3 个 neurons，每个 neuron 对应到一个可以采取的行。

那 input 一个东西以后呢，你的 network 就会给每一个可以采取的行为一个分数，接下来你把这个分数当作是机率，那你的 actor 就是看这个机率的分布，根据这个机率的分布，决定它要采取的行为，比如说 70% 会走 left，20% 走 right，10% 开火，等等，那这个机率分布不同，你的 actor 采取的行为就会不一样。

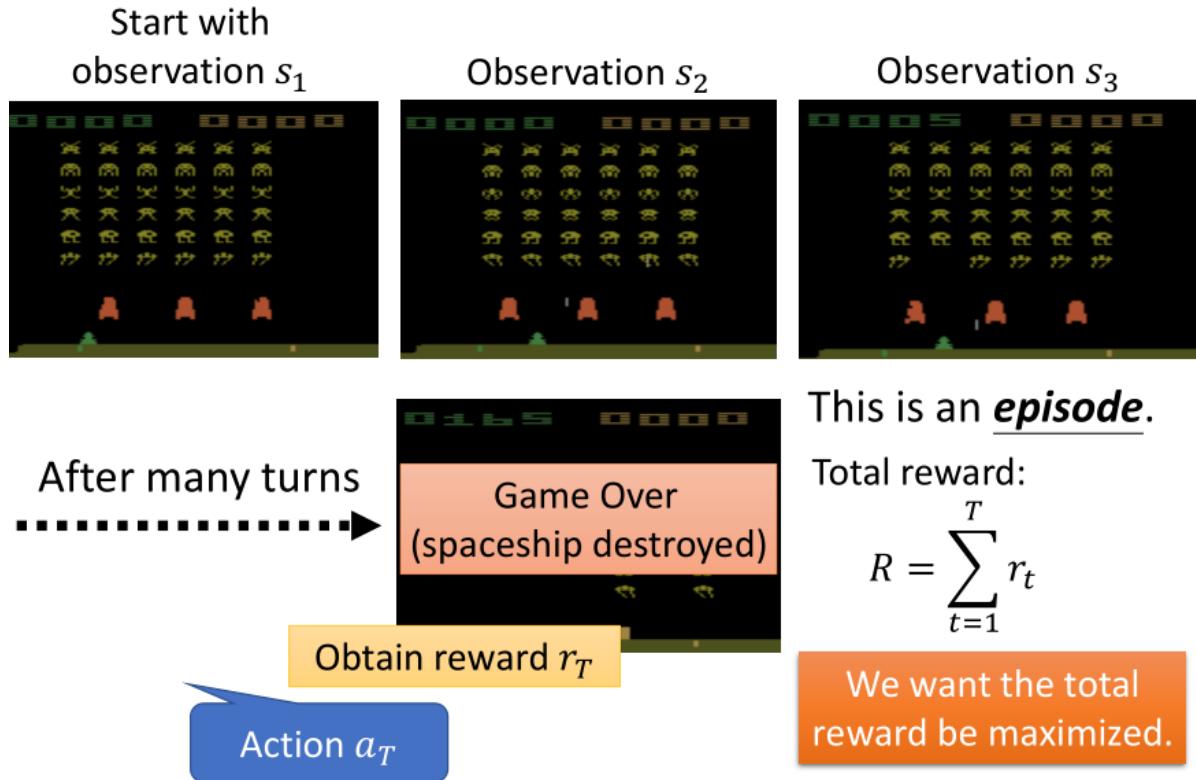
那这是 policy 的部分，它就是一个 network。

Example: Playing Video Game

接下来用一个例子，具体的很快地说一下说，今天你的 actor 是怎么样跟环境互动的。



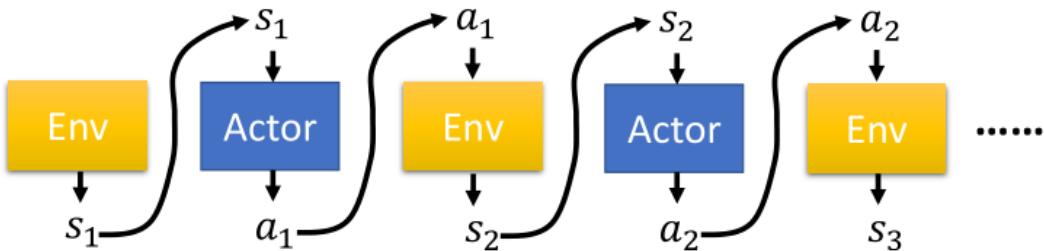
首先你的 actor 会看到一个游戏画面，这个游戏画面，我们就用 s_1 来表示它，它代表游戏初始的画面。接下来你的 actor 看到这个游戏的初始画面以后，根据它内部的 network，根据它内部的 policy，它就会决定一个 action，那假设它现在决定的 action 是向右，那它决定完 action 以后，它就会得到一个 reward，代表它采取这个 action 以后，它会得到多少的分数，那这边我们把一开始的初始画面，写作 s_1 ，我们把第一次执行的动作叫做 a_1 ，我们把第一次执行动作完以后得到的 reward，叫做 r_1 ，那不同的文献，其实有不同的定义，有人会觉得说，这边应该要叫做 r_2 ，这个都可以，你自己看得懂就好。



那接下来就看到新的游戏画面，你的，actor 决定一个的行为以后，，就会看到一个新的游戏画面，这边是 s_2 ，然后把这个 s_2 输入给 actor，这个 actor 决定要开火，然后它可能杀了一只怪，就得到五分，然后这个 process 就反复的持续下去，直到今天走到某一个 time step，执行某一个 action，得到 reward 之后，这个 environment 决定这个游戏结束了，比如说，如果在这个游戏里面，你是控制绿色的船去杀怪，如果你被杀死的话，游戏就结束，或是你把所有的怪都清空，游戏就结束了。那一场游戏，叫做一个 episode，把这个游戏里面，所有得到的 reward，通通总合起来，就是 Total reward，那这边用大 R 来表示它，那今天这个 actor 它存在的目的，就是想办法去 maximize 它可以得到的 reward。

Actor, Environment, Reward

那这边是用图像化的方式，来再跟大家说明一下，你的 environment，actor，还有 reward 之间的关系。



Trajectory $\tau = \{s_1, a_1, s_2, a_2, \dots, s_T, a_T\}$

$p_\theta(\tau)$

$$= p(s_1)p_\theta(a_1|s_1)p(s_2|s_1, a_1)p_\theta(a_2|s_2)p(s_3|s_2, a_2) \dots \\ = p(s_1) \prod_{t=1}^T p_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)$$

首先，environment 其实它本身也是一个 function，连那个游戏的主机，你也可以把它看作是一个 function，虽然它里面不见得是 neural network，可能是 rule-based 的规则，但你可以把它看作是一个 function。

那这个 function，一开始就先吐出一个 state，也就是游戏的画面，接下来你的 actor 看到这个游戏画面 s_1 以后，它吐出 a_1 ，接下来 environment 把这个 a_1 当作它的输入，然后它再吐出 s_2 ，吐出新的游戏画面，actor 看到新的游戏画面，又再决定新的行为 a_2 ，然后 environment 再看到 a_2 ，再吐出 s_3 ，那这个 process 就一直下去，直到 environment 觉得说应该要停止为止。

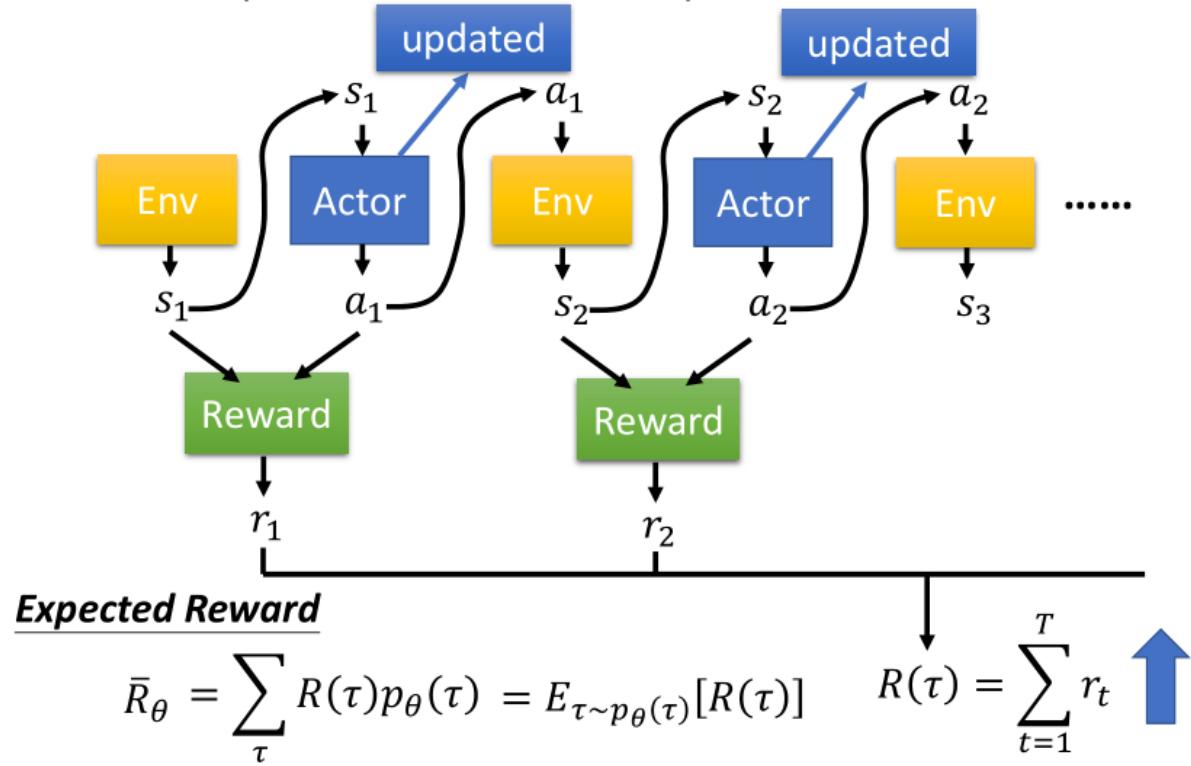
在一场游戏里面，我们把 environment 输出的 s ，跟 actor 输出的行为 a ，把这个 s 跟 a 全部串起来，叫做一个 Trajectory，每一个 trajectory，你可以计算它发生的机率，假设现在 actor 的参数已经被给定了话，就是 θ ，根据这个 θ ，你其实可以计算某一个 trajectory 发生的机率，你可以计算某一个回合，某一个 episode 里面，发生这样子状况的机率。

假设你 actor 的参数就是 θ 的情况下，某一个 trajectory τ ，它的机率就是这样算的，你先算说 environment 输出 s_1 的机率，再计算根据 s_1 执行 a_1 的机率，这个机率是由你 policy 里面的那个 network 参数 θ 所决定的，它是一个机率，因为我们之前有讲过说，你的 policy 的 network，它的 output 它其实是一个 distribution，那你的 actor 是根据这个 distribution 去做 sample，决定现在实际上要采取的 action 是哪一个。

接下来你这个 environment，根据，这边图中是说根据 a_1 产生 s_2 ，那其实它是根据， a_1 跟 s_1 产生 s_2 ，因为 s_2 跟 s_1 还是有关系的，下一个游戏画面，跟前一个游戏画面，通常还是有关系的，至少要是连续的，所以这边是给定前一个游戏画面 s_1 ，跟你现在 actor 采取的行为 a_1 ，然后会产生 s_2 ，这件事情它可能是机率，也可能不是机率，这个是就取决于那个 environment，就是那个主机它内部设定是怎么样，看今天这个主机在决定，要输出什么样的游戏画面的时候，有没有机率。如果没有机率的话，那这个游戏的每次的行为都一样，你只要找到一条 path，就可以过关了，这样感觉是蛮无聊的。所以游戏里面，通常是还是有一些机率的，你做同样的行为，给同样的给前一个画面，下次产生的画面其实不见得是一样的，Process 就反复继续下去，你就可以计算说，一个 trajectory s_1, a_1, s_2, a_2 它出现的机率有多大。

那这个机率，取决于两件事，一部分是 environment 本身的行为，environment 的 function，它内部的参数或内部的规则长什么样子，那这个部分，就这一项 $p(s_{t+1} | s_t, a_t)$ ，代表的是 environment，这个 environment 这一项通常你是无法控制它的，因为那个是人家写好的，你不能控制它，你能控制的是 $p_\theta(a_t | s_t)$ ，你就 given 一个 s_t ，你的 actor 要采取什么样的行为 a_t 这件事，会取决于你 actor 的参数，

你的 passed 参数 θ , 所以这部分是 actor 可以自己控制的。随着 actor 的行为不同, 每个同样的 trajectory, 它就会有不同的出现的机率。



我们说在 reinforcement learning 里面, 除了 environment 跟 actor 以外呢, 还有第三个角色, 叫做 reward function。Reward function 做的事情就是, 根据在某一个 state 采取的某一个 action, 决定说现在在这个行为, 可以得到多少的分数, 它是一个 function, 给它 s_1, a_1 , 它告诉你得到 r_1 , 给它 s_2, a_2 , 它告诉你得到 r_2 , 我们把所有的小 r 都加起来, 我们就得到了大 R , 我们这边写做大 $R(\tau)$, 代表说是, 某一个 trajectory τ , 在某一场游戏里面, 某一个 episode 里面, 我们会得到的大 R 。

那今天我们要做的事情就是调整 actor 内部的参数 θ , 使得 R 的值越大越好, 但是实际上 reward, 它并不仅仅是一个 scalar, reward 它其实是一个 random variable, 这个大 R 其实是一个 random variable。

为什么呢? 因为你的 actor 本身, 在给定同样的 state 会做什么样的行为, 这件事情是有随机性的, 你的 environment, 在给定同样的 action 要产生什么样的 observation, 本身也是有随机性的。所以这个大 R 其实是一个 random variable, 你能够计算的是它的期望值。你能够计算的是, 在给定某一组参数 θ 的情况下, 我们会得到的这个大 R 的期望值是多少, 那这个期望值是怎么算的呢? 这期望值的算法就是, 穷举所有可能的 trajectory, 穷举所有可能的 trajectory τ , 每一个 trajectory τ , 它都有一个机率, 比如说今天你的 θ 是一个很强的 model, 它都不会死, 那如果今天有一个 episode 是很快就死掉了, 它的机率就很小, 如果有一个 episode 是都一直没有死, 那它的机率就很大, 那根据你的 θ , 你可以算出某一个 trajectory τ 出现的机率, 接下来你计算这个 τ 的 total reward 是多少, 把 total reward weighted by 这个 τ 出现的机率, summation over 所有的 τ , 显然就是 given 某一个参数你会得到的期望值, 或你会写成这样, 从 $p(\theta)$ of τ 这个 distribution, sample 一个 trajectory τ , 然后计算 R of τ 的期望值, 就是你的 expected reward。

Policy Gradient

那我们要做的事情, 就是 maximize expected reward, 怎么 maximize expected reward 呢? 我们用的就是 gradient ascent。因为我们要让它越大越好, 所以是 gradient ascent, 所以跟 gradient decent 唯一不同的地方就只是, 本来在 update 参数的时候, 要减, 现在变成加。

$$\text{Policy Gradient} \quad \bar{R}_\theta = \sum_\tau R(\tau)p_\theta(\tau) \quad \nabla \bar{R}_\theta = ?$$

$$\nabla \bar{R}_\theta = \sum_\tau R(\tau) \nabla p_\theta(\tau) = \sum_\tau R(\tau)p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)}$$

$R(\tau)$ do not have to be differentiable

It can even be a black box.

$$= \boxed{\sum_\tau} R(\tau) \boxed{p_\theta(\tau)} \nabla \log p_\theta(\tau)$$

$$\begin{aligned} \nabla f(x) &= \\ f(x) \nabla \log f(x) & \end{aligned}$$

$$\begin{aligned} &= E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_\theta(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned}$$

然后这 gradient ascent 你就必须计算， R bar 这个 expected reward，它的 gradient， R bar 的 gradient 怎么计算呢？这跟 GAN 做 sequence generation 的式子，其实是一模一样的。

R bar 我们取一个 gradient，这里面只有 $p(\theta)$ ，是跟 θ 有关，所以 gradient 就放在 $p(\theta)$ 这个地方。

R 这个 reward function，不需要是 differentiable，我们也可以解接下来的问题，举例来说，如果是在 GAN 里面，你的这个 R 其实是一个 discriminator，它就算是没有办法微分也无所谓，你还是可以做接下来的运算。

接下来要做的事情，分子分母，上下同乘 $p_\theta(\tau)$ ，后面这一项其实就是这个 $\log p_\theta(\tau)$ ，取 gradient。

或者是你其实之后就可以直接背一个公式，就某一个 function f of x ，你对它做 gradient 的话，就等于 f of x 乘上 gradient $\log f$ of x 。

所以今天这边有一个 gradient $p_\theta(\tau)$ ，带进这个公式里面呢，这边应该变成 $p_\theta(\tau)$ 乘上 gradient $\log p_\theta(\tau)$ 。

然后接下来呢，这边又 summation over τ ，然后又有把这个 R 跟这个 \log 这两项，weighted by $p_\theta(\tau)$ ，那既然有 weighted by $p_\theta(\tau)$ ，它们就可以被写成这个 expected 的形式，也就是你从 $p_\theta(\tau)$ 这个 distribution 里面 sample τ 出来，去计算 R of τ 乘上 gradient $\log p_\theta(\tau)$ ，然后把它对所有可能的 τ 做 summation，就是这个 expected value。

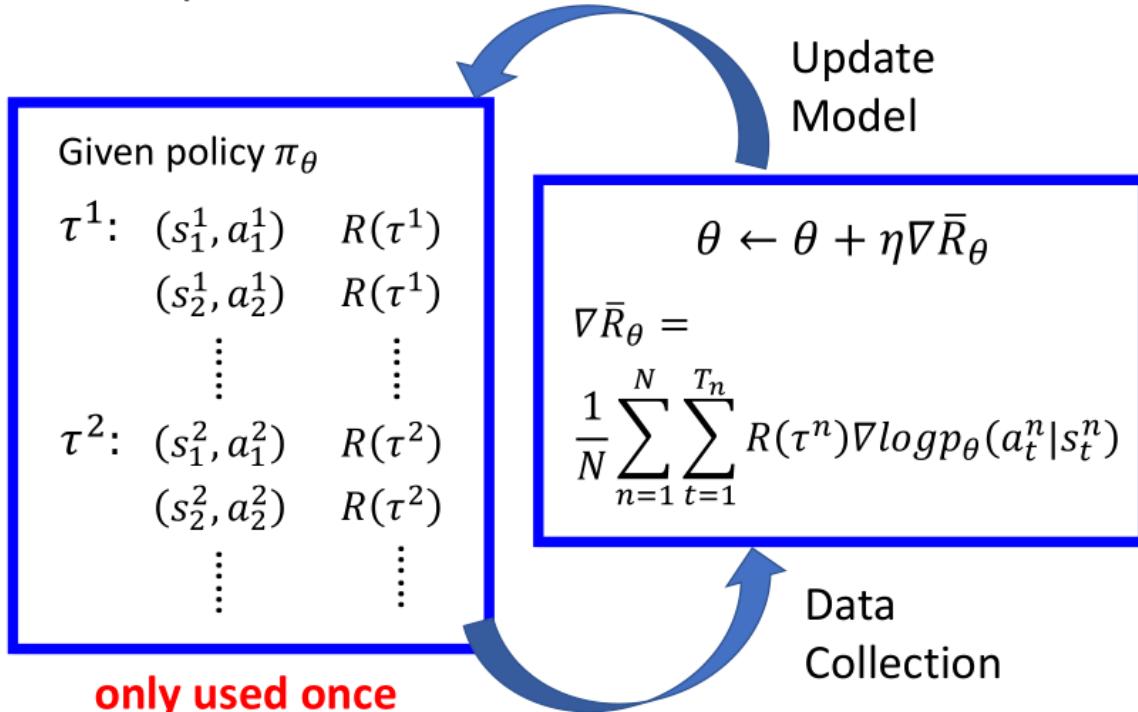
这个 expected value 实际上你没有办法算，所以你是用 sample 的方式，来 sample 一大堆的 τ ，你 sample 大 N 笔 τ ，然后每一笔呢，你都去计算它的这些 value，然后把它全部加起来，最后你就得到你的 gradient。你就可以去 update 你的参数，你就可以去 update 你的 agent。

那这边呢，我们跳了一大步，这边这个 $p(\theta)$ of τ ，我们前面有讲过 $p(\theta)$ of τ 是可以算的，那 $p(\theta)$ of τ 里面有两项，一项是来自于 environment，一项是来自于你的 agent，来自 environment 那一项，其实你根本就不能算它，你对它做 gradient 是没有用的，因为它跟 θ 是完全没有任何关系的，所以你不需要对它做 gradient。你真正做 gradient 的，只有 $\log p(\theta)$ of at given st 而已。

这个部分，其实你可以非常直观的来理解它，也就是在你 sample 到的 data 里面，你 sample 到，在某一个 state s_t 要执行某一个 action a_t 。就是这个 s_t 跟 a_t ，它是在整个 trajectory τ 的里面的某一个 state and action 的 pair，假设你在 s_t 执行 a_t ，最后发现 τ 的 reward 是正的，那你要增加这一项的机率，你就要增加在 s_t 执行 a_t 的机率，反之，在 s_t 执行 a_t 会导致整个 trajectory 的 reward 变成负的，你就减少这一项的机率，那这个概念就是怎么简单。

$$\nabla \bar{R}_\theta = E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)]$$

Policy Gradient



这个怎么实作呢？你用 gradient ascent 的方法，来 update 你的参数，所以你原来有一个参数 θ ，你把你的 θ 加上你的 gradient 这一项，那当然前面要有个 learning rate，learning rate 其实也是要调的，你要用 adam、rmsprop 等等，还是要调一下。那这 gradient 这一项怎么来呢？gradient 这一项，就套下面这个公式，把它算出来，那在实际上做的时候，要套下面这个公式，首先你要先收集一大堆的 s 跟 a 的 pair，你还要知道这些 s 跟 a ，如果实际上在跟环境互动的时候，你会得到多少的 reward，所以这些数据，你要去收集起来，这些资料怎么收集呢？你就要拿你的 agent，它的参数是 θ ，去跟环境做互动，也就是你拿你现在已经 train 好的那个 agent，先去跟环境玩一下，先去跟那个游戏互动一下，那互动完以后，你就会得到一大堆游戏的纪录，你会记录说，今天先玩了第一场，在第一场游戏里面，我们在 state s_1 ，采取 action a_1 ，在 state s_2 ，采取 action a_2 。那要记得说其实今天玩游戏的时候，是有随机性的，所以你的 agent 本身是有随机性的，所以在同样 state s_1 ，不是每次都会采取 a_1 ，所以你要记录下来，在 state s_1 ，采取 a_1 ，在 state s_2 ，采取 a_2 ，整场游戏结束以后，得到的分数，是 R of $\tau(1)$ ，那你会 sample 到另外一笔 data，也就是另外一场游戏，在另外一场游戏里面，你在第一个 state 采取这个 action，在第二个 state 采取这个 action，在第二个游戏画面采取这个 action，你得到的 reward 是 R of $\tau(2)$ ，你有了这些东西以后，你就去把这边你 sample 到的东西，带到这个 gradient 的式子里面，把 gradient 算出来。

也就是说你会做的事情是，把这边的每一个 s 跟 a 的 pair，拿进来，算一下它的 log probability，你计算一下，在某一个 state，采取某一个 action 的 log probability，然后对它取 gradient，然后这个 gradient 前面会乘一个 weight，这个 weight 就是这场游戏的 reward。

你有了这些以后，你就会去 update 你的 model，你 update 完你的 model 以后，你回过头来要重新再去收集你的 data，再 update model...

那这边要注意一下，一般 policy gradient，你 sample 的 data 就只会用一次，你把这些 data sample 起来，然后拿去 update 参数，这些 data 就丢掉了，再重新 sample data，才能够再重新去 update 参数。等一下我们会解决这个问题。

Implementation

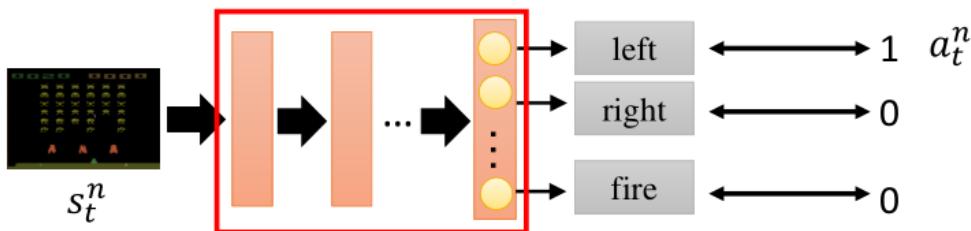
$$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta$$

Implementation

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n)$$

Consider as classification problem

s_t^n a_t^n $R(\tau^n)$



$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \log p_\theta(a_t^n | s_t^n) \xrightarrow{\text{TF, pyTorch ...}} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \nabla \log p_\theta(a_t^n | s_t^n)$$

$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \underline{R(\tau^n)} \log p_\theta(a_t^n | s_t^n) \xrightarrow{} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \underline{R(\tau^n)} \nabla \log p_\theta(a_t^n | s_t^n)$$

那接下来的就是实作的时候你会遇到的实作的一些细节，这个东西到底实际上在用这个 deep learning 的 framework implement 的时候，它是怎么实作的呢，其实你的实作方法是这个样子，你要把它想成你就是在做一个分类的问题，所以那要怎么做 classification，当然要收集一堆 training data，你要有 input 跟 output 的 pair，那今天在 reinforcement learning 里面，在实作的时候，你就把 state 当作是 classifier 的 input，你就当作你是要做 image classification 的 problem，只是现在的 class 不是说 image 里面有什么 objects，现在的 class 是说，看到这张 image 我们要采取什么样的行为，每一个行为就叫做一个 class，比如说第一个 class 叫做向左，第二个 class 叫做向右，第三个 class 叫做开火。

那这些训练的资料是从哪里来的呢？我们说你要做分类的问题，你要有 classified 的 input，跟它正确的 output，这些训练数据，就是从 sampling 的 process 来的，假设在 sampling 的 process 里面，在某一个 state，你 sample 到你要采取 action a，你就把这个 action a 当作是你的 ground truth，你在这个 state，你 sample 到要向左，本来向左这件事机率不一定是最高，因为你是 sample，它不一定机率最高，假设你 sample 到向左，那接下来在 training 的时候，你叫告诉 machine 说，调整 network 的参数，如果看到这个 state，你就向左。

在一般的 classification 的 problem 里面，其实你在 implement classification 的时候，你的 objective function，都会写成 minimize cross entropy，那其实 minimize cross entropy 就是 maximize log likelihood，所以你今天在做 classification 的时候，你的 objective function，你要去 maximize 或是 minimize 的对象，因为我们现在是 maximize likelihood，所以其实是 maximize，你要 maximize 的对象，其实就长这样子，像这种 lost function，你在 TensorFlow 里面，你 even 不用手刻，它都会有现成的 function 就是了，你就 call 个 function，它就会自动帮你算这样子的东西。

然后接下来呢，你就 apply 计算 gradient 这件事，那你就可以把 gradient 计算出来，这是一般的分类问题。

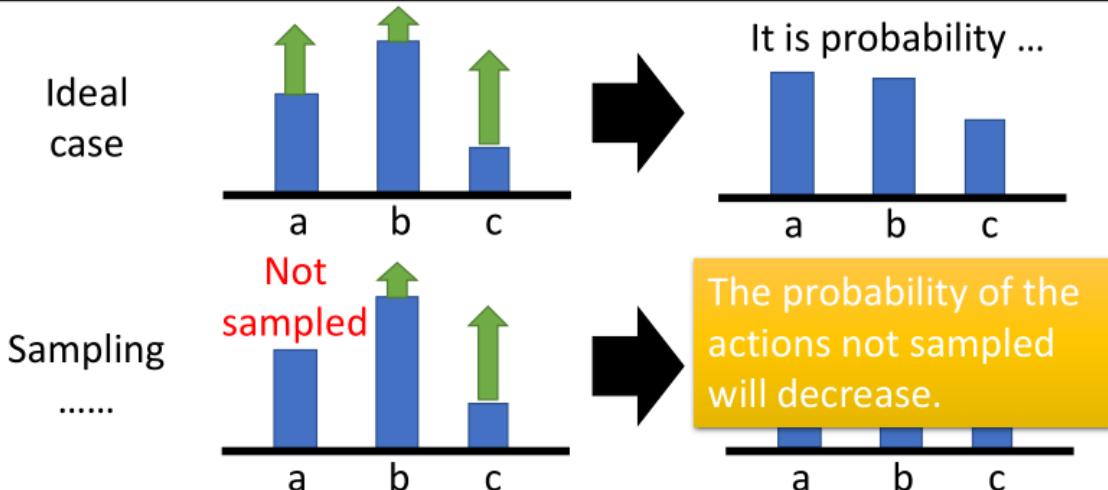
那如果今天是 RL 的话，唯一不同的地方只是，你要记得在你原来的 loss 前面，乘上一个 weight，这个 weight 是什么？这个 weight 是，今天在这个 state，采取这个 action 的时候，你会得到的 reward，这个 reward 不是当时得到的 reward，而是整场游戏的时候得到的 reward，它并不是在 state s 采取 action a 的时候得到的 reward，而是说，今天在 state s 采取 action a 的这整场游戏里面，你最后得到的 total reward 这个大 R。你要把你的每一笔 training data，都 weighted by 这个大 R。然后接下来，你就交给 TensorFlow 或 PyTorch 去帮你算 gradient，然后就结束了。跟一般 classification 其实也没太大的差别。

Tip 1: Add a Baseline

这边有一些通常实作的时候，你也许用得上的 tip，一个就是你要 add 一个东西叫做 baseline，所谓的 add baseline 是什么意思呢？今天我们会遇到一个状况是，我们说这个式子，它直觉上的含意就是，假设 given state s 采取 action a，会给你整场游戏正面的 reward，那你就增加它的机率，如果说今天在 state s 执行 action a，整场游戏得到负的 reward，你就要减少这一项的机率。但是我们今天很容易遇到一个问题，很多游戏里面，它的 reward 总是正的，就是说最低都是 0。这个 R 总是正的，所以假设你直接套用这个式子，你会发现说在 training 的时候，你告诉 model 说，今天不管是什 action，你都应该要把它的机率提升，这样听起来好像有点怪怪的。在理想上，这么做并不一定会有问题，因为今天虽然说 R 总是正的，但它正的量总是有大有小，你采取某些 action 可能是得到 0 分，采取某些 action 可能是得到 20 分。

$$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta \quad \boxed{\text{It is possible that } R(\tau^n) \text{ is always positive.}}$$

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p_\theta(a_t^n | s_t^n) \quad b \approx E[R(\tau)]$$



假设在某一个 state 有 3 个 action a/b/c，可以执行，根据这个式子，你要把这 3 项的 log probability 都拉高，但是它们前面 weight 的这个 R，是不一样的，那么前面 weight 的这个 R 是有大有小的，weight 小的，它上升的就少，weight 多的，它上升的就大一点。那因为今天这个 log probability，它是一个机率，所以，这三项的和，要是 1，所以上升的少的，在做完 normalize 以后，它其实也是下降的，上升的多的，才会上升，那这个是一个理想上的状况。但是实际上，你千万不要忘了，我们是在做 sampling，本来这边应该是一个 expectation，summation over 所有可能的 s 跟 a 的 pair，但是实际上你真正在学的时候，当然不可能是这么做的，你只是 sample 了少量的 s 跟 a 的 pair 而已。

所以我们今天做的是 sampling，有一些 action 你可能从来都没有 sample 到，在某一个 state，虽然可以执行的 action 有 a/b/c 3 个，但你可能没有 sample 到 action a，但现在所有 action 的 reward 都是正的，今天它的每一项的机率都应该是上升的，但现在你会遇到的问题是，因为 a 没有被 sample 到，其它人的机率如果都要上升，那 a 的机率就下降，所以，a 可能不是一个好的 action，它只是没被 sample 到，也就是运气不好没有被 sample 到，但是只是因为它没被 sample 到，它的机率就会下降，那这个显然是有问题的。要解决这个问题要怎么办呢？

你会希望你的 reward 不要总是正的。为了解决你的 reward 不要总是正的这个问题，你可以做的一个非常简单的改变就是，把你的 reward 减掉一项叫做 b ，这项 b 叫做 baseline，你减掉这项 b 以后，就可以让 $R - b$ 有正有负，所以今天如果你得到的 reward 这个 R of $\tau(n)$ ，这个 total reward 大于 b 的话，就让他的机率上升，如果这个 total reward 小于 b ，你就要让这个 state 采取这个 action 的分数下降。

那这个 b 怎么设呢？你就随便设，你就自己想个方法来设，那一个最最简单的做法就是，你把 $\tau(n)$ 的值，取 expectation，算一下 $\tau(n)$ 的平均值，你就可以把它当作 b 来用，这是其中一种做法。

所以在实作上，你就是在 implement/training 的时候，你会不断的把 R of τ 的分数，把它不断的记录下来，你会不断的去计算 R of τ 的平均值，然后你会把你的这个平均值，当作你的 b 来用，这样就可以让你在 training 的时候，这个 gradient log probability 乘上前面这一项，是有正有负的，这个是第一个 tip。

Tip 2: Assign Suitable Credit

第二个 tip 是在 machine learning 那一门课没有讲过的 tip。这个 tip 是这样子，今天你应该要给每一个 action，合适的 credit。

如果我们看今天下面这个式子的话，我们原来会做的事情是，今天在某一个 state，假设，你执行了某一个 action a ，它得到的 reward，它前面乘上的这一项，就是 $(R \text{ of } \tau) - b$ ，今天只要在同一个 episode 里面，在同一场游戏里面，所有的 state 跟 a 的 pair，它都会 weighted by 同样的 reward/term，这件事情显然是不公平的。因为在同一场游戏里面，也许有些 action 是好的，也许有些 action 是不好的，那假设最终的结果，整场游戏的结果是好的，并不代表这个游戏里面每一个行为都是对的，若是整场游戏结果不好，但不代表游戏里的所有行为都是错的。所以我们其实希望，可以给每一个不同的 action，前面都乘上不同的 weight，那这个每一个 action 的不同 weight，它真正的反应了每一个 action，它到底是好还是不好。

$$\begin{array}{cccccc}
 \times 3 & \times -2 & \times -2 & \times -7 & \times -2 & \times -2 \\
 (s_a, a_1) & (s_b, a_2) & (s_c, a_3) & (s_a, a_2) & (s_b, a_2) & (s_c, a_3) \\
 +5 & +0 & -2 & -5 & +0 & -2 \\
 R = +3 & & & & R = -7 &
 \end{array}$$

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (\cancel{R(\tau^n)} - b) \nabla \log p_\theta(a_t^n | s_t^n)$$

$\sum_{t'=t}^{T_n} r_{t'}^n$

假设现在这个游戏都很短，只会有 3-4 个互动，在 s_a 这个 state 执行 a_1 这件事，得到 5 分，在 s_b 这个 state 执行 a_2 这件事，得到 0 分，在 s_c 这个 state 执行 a_3 这件事，得到 -2 分，整场游戏下来，你得到 +3 分。那今天你得到 +3 分，代表在 state s_b 执行 action a_2 是好的吗？并不见得。因为这个正的分数，主要是来自于一开始的时候 state s_a 执行了 a_1 ，也许跟在 state s_b 执行 a_2 是没有关系的，也许在 state s_b 执行 a_2 反而是不好的，因为它导致你接下来会进入 state s_c 执行 a_3 被扣分。

所以今天整场游戏得到的结果是好的，并不代表每一个行为都是对的，如果按照我们刚才的讲法，今天整场游戏得到的分数是 3 分，那到时候在 training 的时候，每一个 state 跟 action 的 pair，都会被乘上 +3。

在理想的状况下，这个问题，如果你 sample 够多，就可以被解决，为什么？因为假设你今天 sample 够多，在 state sb 执行 a2 的这件事情，被 sample 到很多次，就某一场游戏，在 state sb 执行 a2，你会得到 +3 分，但在另外一场游戏，在 state sb 执行 a2，你却得到了 -7 分，为什么会得到 -7 分呢？因为在 state sb 执行 a2 之前，你在 state sa 执行 a2 得到 -5 分，那这 -5 分可能也不是，中间这一项的错，这 -5 分这件事可能也不是在 sb 执行 a2 的错，这两件事情，可能是没有关系的，因为它先发生了，这件事才发生，所以他们是没关系的。在 state sb 执行 a2，它可能造成问题只有，会在接下来 -2 分，而跟前面的 -5 分没有关系的，但是假设我们今天 sample 到这项的次数够多，把所有有发生这件事情的情况的分数通通都集合起来，那可能不是一个问题。

但现在的问题就是，我们 sample 的次数，是不够多的，那在 sample 的次数，不够多的情况下，你就需要想办法，给每一个 state 跟 action pair 合理的 credit，你要让大家知道它实际上对这些分数的贡献到底有多大，那怎么给它一个合理的 contribution 呢？

一个做法是，我们今天在计算这个 pair，它真正的 reward 的时候，不把整场游戏得到的 reward 全部加起来，我们只计算从这一个 action 执行以后，所得到的 reward。因为这场游戏在执行这个 action 之前发生的事情，是跟执行这个 action 是没有关系的，前面的事情都已经发生了，那跟执行这个 action 是没有关系的。所以在执行这个 action 之前，得到多少 reward 都不能算是这个 action 的功劳。跟这个 action 有关的东西，只有在执行这个 action 以后发生的所有的 reward，把它总合起来，才是这个 action 它真正的 contribution，才比较可能是这个 action 它真正的 contribution。

所以在这个例子里面，在 state sb，执行 a2 这件事情，也许它真正会导致你得到的分数，应该是 -2 分而不是 +3 分，因为前面的 +5 分，并不是执行 a2 的功劳，实际上执行 a2 以后，到游戏结束前，你只有被扣 2 分而已，所以它应该是 -2。

那一样的道理，今天执行 a2 实际上不应该是扣 7 分，因为前面扣 5 分，跟在 sb 这个 state 执行 a2 是没有关系的。所以也许在 sb 这个 state 执行 a2，你真正会导致的结果只有扣两分而已。

那如果要把它写成式子的话是什么样子呢？你本来前面的 weight，是 R of τ ，是整场游戏的 reward 的总和，那现在改一下，怎么改呢？改成从某个时间 t 开始，假设这个 action 是在 t 这个时间点所执行的，从 t 这个时间点，一直到游戏结束，所有 reward R 的总和，才真的代表这个 action，是好的，还是不好的。

Advantage Function $A^\theta(s_t, a_t)$

How good it is if we take a_t other than other actions at s_t .
Estimated by “critic” (later)

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p_\theta(a_t^n | s_t^n)$$

Can be state-dependent

\downarrow

$$\sum_{t'=t}^{T_n} r_{t'}^n \rightarrow \sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$$

Add discount factor $\gamma < 1$

接下来再更进一步，我们会把比较未来的 reward，做一个 discount，为什么我要把比较未来的 reward 做一个 discount 呢？因为今天虽然我们说，在某一个时间点，执行某一个 action，会影响接下来所有的结果，有可能在某一个时间点执行的 action，接下来得到的 reward 都是这个 action 的功劳。但是在比较真实的情况下，如果时间拖得越长，影响力就越小，就是今天我在第二个时间点执行某一个 action，我在第三个时间点得到 reward，那可能是再第二个时间点执行某个 action 的功劳，但是在 100 个 time step 之后，又得到 reward，那可能就不是在第二个时间点执行某一个 action 得到的功劳。

所以我们实际上在做的时候，你会在你的 R 前面，乘上一个 term 叫做 gamma，那 gamma 它是小于 1 的，它会设个 0.9 或 0.99，那如果你今天的 R，它是越之后的 time stamp，它前面就乘上越多次的 gamma，就代表现在在某一个 state s_t ，执行某一个 action a_t 的时候，真正它的 credit，其实是它之后，在执行这个 action 之后，所有 reward 的总和，而且你还要乘上 gamma。

实际上你 implement 就是这么 implement 的，那这个 b 呢，b 这个我们之后再讲，它可以是 state-dependent 的，事实上 b 它通常是一个 network estimate 出来的，这还蛮复杂，它是一个 network 的 output，这个我们之后再讲。

那把这个 R 减掉 b 这一项，这一项我们可以把它合起来，我们统称为 advantage function，我们这边用 A 来代表 advantage function，那这个 advantage function，它是 dependent on s and a ，我们就是要计算的是，在某一个 state s 采取某一个 action a 的时候，你的 advantage function 有多大，然后这个 advantage function 它的上标是 θ ， θ 是什么意思呢？因为你实际上在算这个 summation 的时候，你会需要有一个 interaction 的结果嘛，对不对，你会需要有一个 model 去跟环境做 interaction，你才知道你接下来得到的 reward 会有多少，而这个 θ 就是代表说，现在是用 θ 这个 model，跟环境去做 interaction，然后你才计算出这一项，从时间 t 开始到游戏结束为止，所有 R 的 summation，把这一项减掉 b，然后这个就叫 advantage function。

它的意义就是，现在假设，我们在某一个 state s_t ，执行某一个 action a_t ，相较于其他可能的 action，它有多好，它真正在意的不是一个绝对的好，而是说在同样的 state 的时候，是采取某一个 action a_t ，相较于其它的 action，它有多好，它是相对的好，不是绝对好，因为今天会减掉一个 b，减掉一个 baseline，所以这个东西是相对的好，不是绝对的好，那这个 A 我们之后再讲，它通常可以是由一个 network estimate 出来的，那这个 network 叫做 critic，我们讲到 Actor-Critic 的方法的时候，再讲这件事情。

From on-policy to off-policy

Using the experience more than once

On-policy v.s. Off-policy

那在讲 PPO 之前呢，我们要讲 on-policy and off-policy，这两种 training 方法的区别，那什么是 on-policy 什么是 off-policy 呢？

我们知道在 reinforcement learning 里面，我们要 learn 的就是一个 agent，那如果我们今天拿去跟环境互动的那个 agent，跟我们要 learn 的 agent 是同一个的话，这个叫做 on-policy，如果我们今天要 learn 的 agent，跟和环境互动的 agent 不是同一个的话，那这个叫做 off-policy，比较拟人化的讲法就是，如果今天要学习的那个 agent，它是一边跟环境互动，一边做学习，这个叫 on-policy，果它是在旁边看别人玩，透过看别人玩，来学习的话，这个叫做 off-policy。

On-policy: The agent learned and the agent interacting with the environment is the same.

Off-policy: The agent learned and the agent interacting with the environment is different.



阿光下棋



佐為下棋、阿光在旁邊看

为什么我们会想要考虑 off-policy 这样的选项呢？让我们来想想看我们已经讲过的 policy gradient，其实我们之前讲的 policy gradient，它是 on-policy 还是 off-policy 的做法呢？它是 on-policy 的做法。为什么？我们之前讲说，在做 policy gradient 的时候呢，我们会需要有一个 agent，我们会需要有一个 policy，我们会需要有一个 actor，这个 actor 先去跟环境互动，去搜集资料，搜集很多的 τ ，那根据它搜集到的资料，会按照这个 policy gradient 的式子，去 update 你 policy 的参数，这个就是我们之前讲过的 policy gradient，所以它是一个 on-policy 的 algorithm，你拿去跟环境做互动的那个 policy，跟你要 learn 的 policy 是同一个。

那今天的问题是，我们之前有讲过说，因为在这个 update 的式子里面，其中有一项，你的这个 expectation，应该是对你现在的 policy θ ，所 sample 出来的 trajectory τ ，做 expectation，所以当你今天 update 参数以后，一旦你 update 了参数，从 θ 变成 θ' ，那这一个机率，就不对了，之前 sample 出来的 data，就变的不能用了。

所以我们之前就有讲过说，policy gradient，是一个会花很多时间来 sample data 的algorithm，你会发现大多数时间都在 sample data。你的 agent 去跟环境做互动以后，接下来就要 update 参数，你只能做一次 gradient decent，你只能 update 参数一次，接下来你就要重新再去 collect data，然后才能再次 update 参数，这显然是非常花时间的。

On-policy → Off-policy

$$\nabla \bar{R}_\theta = E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)]$$

- Use π_θ to collect data. When θ is updated, we have to sample training data again.
- Goal: Using the sample from $\pi_{\theta'}$ to train θ . θ' is fixed, so we can re-use the sample data.

所以我们现在想要从 on-policy 变成 off-policy 的好处就是，我们希望说现在我们可以用另外一个 policy，另外一個 actor θ' 去跟环境做互动，用 θ' collect 到的 data 去训练 θ ，假设我们可以用 θ' collect 到的 data 去训练 θ ，意味着说，我们可以把 θ' collect 到的 data 用非常多次，你在做 gradient ascent 的时候，我们可以执行那个 gradient ascent 好几次，我们可以 update 参数好几次，都只要用同一笔 data

就好了。

因为假设现在 θ 有能力从另外一个 actor θ' , 它所 sample 出来的 data 来学习的话, 那 θ' 就只要 sample 一次, 也许 sample 多一点的 data, 让 θ 去 update 很多次, 这样就会比较有效率。

Importance Sampling

所以怎么做呢? 这边就需要介绍一个 important sampling 的概念, 那这个 important sampling 的概念不是只能用在 RL 上, 它是一个 general 的想法, 可以用在其他很多地方。

Importance Sampling

$$E_{x \sim p}[f(x)] \approx \frac{1}{N} \sum_{i=1}^N f(x^i)$$

x^i is sampled from $p(x)$

We only have x^i sampled from $q(x)$

$$= \int f(x)p(x)dx = \int f(x) \frac{p(x)}{q(x)} q(x)dx = E_{x \sim q}[f(x) \frac{p(x)}{q(x)}]$$

Importance weight

我们先介绍这个 general 的想法, 那假设现在你有一个 function $f(x)$, 那你要计算, 从 p 这个 distribution, sample x , 再把 x 带到, f 里面, 得到 $f(x)$, 你要计算这个 $f(x)$ 的期望值, 那怎么做呢? 假设你今天没有办法对 p 这个 distribution 做积分的话, 那你可以从 p 这个 distribution 去 sample 一些 data x^i , 那这个 $f(x)$ 的期望值, 就等同是你 sample 到的 x^i , 把 x^i 带到 $f(x)$ 里面, 然后取它的平均值, 就可以拿来近似这个期望值。

假设你知道怎么从 p 这个 distribution 做 sample 的话, 你要算这个期望值, 你只需要从 p 这个 distribution, 做 sample 就好了。

但是我们现在有另外一个问题, 那等一下我们会更清楚知道说为什么会有这样的问题。

我们现在的问题是这样, 我们没有办法从 p 这个 distribution 里面 sample data。

假设我们不能从 p sample data, 我们只能从另外一个 distribution q 去 sample data, q 这个 distribution 可以是任何 distribution, 不管它是什么样的 distribution, 在多数情况下, 等一下讨论的情况都成立, 我们不能够从 p 去 sample data, 但我们可以从 q 去 sample x^i , 但我们从 q 去 sample x^i , 我们不能直接套这个式子, 因为这边是假设你的 x^i 都是从 p sample 出来的, 你才能够套这个式子, 从 q sample 出来的 x^i 套这个式子, 你也不会等于左边这项期望值。

所以怎么办? 做一个修正, 这个修正是这样子的, 期望值这一项, 其实就是积分, 然后我们现在上下都同乘 $q(x)$, 我们可以把这一个式子, 写成对 q 里面所 sample 出来的 x 取期望值, 我们从 q 里面, sample x , 然后再去计算 $f(x)$ 乘上 $p(x)$ 除以 $q(x)$, 再去取期望值。

左边这一项, 会等于右边这一项。要算左边这一项, 你要从 p 这个 distribution sample x , 但是要算右边这一项, 你不是从 p 这个 distribution sample x , 你是从 q 这个 distribution sample x , 你从 q 这个 distribution sample x , sample 出来之后再带入, 接下来你就可以计算左边这项你想要算的期望值, 所以就算是我们不能从 p 里面去 sample data, 你想要计算这一项的期望值, 也是没有问题的, 你只要能够从 q 里面去 sample data, 可以带这个式子, 你就一样可以计算, 从 p 这个 distribution sample x , 带入 f 以后所算出来的期望值。

那这个两个式子唯一不同的地方是说, 这边是从 p 做 sample, 这边是从 q 做 sample, 因为他是从 q 里做 sample, 所以 sample 出来的每一笔 data, 你需要乘上一个 weight, 修正这两个 distribution 的差异。而这个 weight 就是 $p(x)$ 的值除以 $q(x)$ 的值, 所以 $q(x)$ 它是任何 distribution 都可以, 这边唯一的限制就是, 你不能够说 q 的机率是 0 的时候, p 的机率不为 0, 不然这样会没有定义。假设 q 的机率是 0 的时候, p 的机率也都是 0 的话, 那这样 p 除以 q 是有定义的, 所以这个时候你就可以, apply important sampling 这个技巧。所以你就可以本来是从 p 做 sample, 换成从 q 做 sample。

Issue of Importance Sampling

这个跟我们刚才讲的从 on-policy 变成 off-policy, 有什么关系呢? 在继续讲之前, 我们来看一下 important sampling 的 issue。

$$E_{x \sim p}[f(x)] = E_{x \sim q}[f(x) \frac{p(x)}{q(x)}]$$

$$\text{Var}_{x \sim p}[f(x)] \quad \text{Var}_{x \sim q}[f(x) \frac{p(x)}{q(x)}] \quad \boxed{\begin{aligned} & \text{VAR}[X] \\ &= E[X^2] - (E[X])^2 \end{aligned}}$$

$$\text{Var}_{x \sim p}[f(x)] = E_{x \sim p}[f(x)^2] - (E_{x \sim p}[f(x)])^2$$

$$\text{Var}_{x \sim q}[f(x) \frac{p(x)}{q(x)}] = E_{x \sim q} \left[\left(f(x) \frac{p(x)}{q(x)} \right)^2 \right] - \left(E_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right] \right)^2$$

$$= E_{x \sim p} \left[f(x)^2 \frac{p(x)}{q(x)} \right] - (E_{x \sim p}[f(x)])^2$$

虽然理论上你可以把 p 换成任何的 q , 但是在实作上, 并没有那么容易, 实作上 p q 还是不能够差太多, 如果差太多的话, 会有一些问题, 什么样的问题呢? 虽然我们知道说, 左边这个式子, 等于右边这个式子, 但你想看, 如果今天左边这个是 $f(x)$, 它的期望值 distribution 是 p , 这边是 $f(x)$ 乘以 p 除以 q 的期望值, 它的 distribution 是 q , 我们现在如果不是算期望值, 而是算 various 的话, 这两个 various 会一样吗? 不一样的。两个 random variable 它的 mean 一样, 并不代表它的 various 一样。

所以可以实际算一下, $f(x)$ 这个 random variable, 跟 $f(x)$ 乘以 $p(x)$ 除以 $q(x)$, 这个 random variable, 他们的这个 various 是不一样的, 这一项的 various, 就套一下公式。

其实可以做一些整理的, 这边有一个 $f(x)$ 的平方, 然后有一个 $p(x)$ 的平方, 有一个 $q(x)$ 的平方, 但是前面呢, 是对 q 取 expectation, 所以 q 的 distribution 取 expectation, 所以如果你要算积分的话, 你就会把这个 q 呢, 乘到前面去, 然后 q 就可以消掉了, 然后你可以把这个 p 拆成两项, 然后就会变成是对 p 呢, 取期望值。这个是左边这一项。

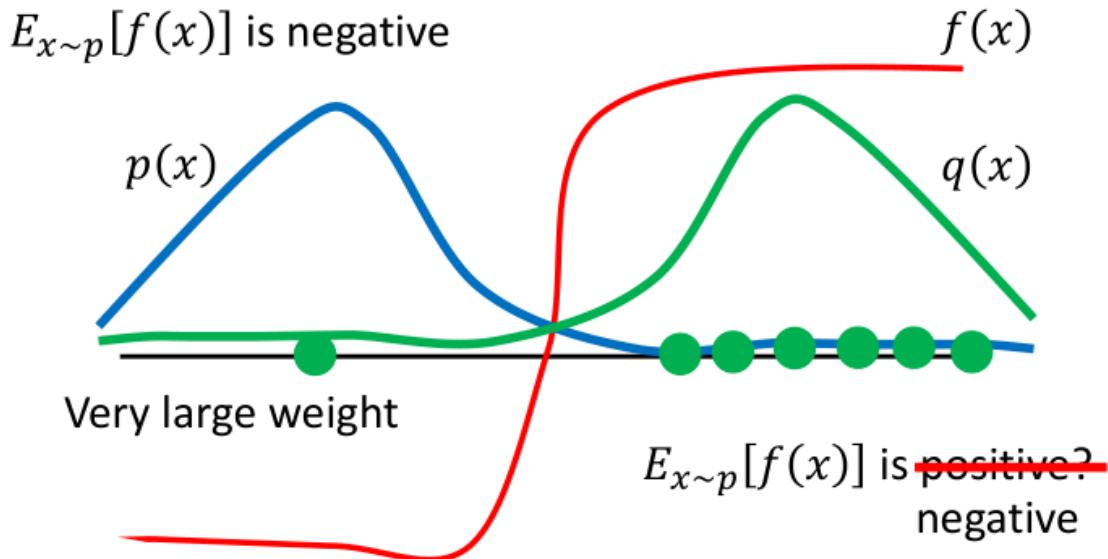
那右边这一项, 其实就写在最前面的公式。

他们 various 的差别在第一项, 第一项这边多乘了 p 除以 q , 如果 p 除以 q 差距很大的话, 这个时候, $\text{Var}_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right]$ 就会很大, 所以虽然理论上 expectation 一样, 也就是说, 你只要对 p 这个 distribution sample 够多次, q 这个 distribution sample 够多次, 你得到的结果会是一样的。

但是假设你 sample 的次数不够多, 因为它们的 various 差距是很大的, 所以你就有可能得到非常大的差别。

这边就是举一个具体的例子告诉你说, 当 p q 差距很大的时候, 会发生什么样的问题。

$$E_{x \sim p}[f(x)] = E_{x \sim q}[f(x) \frac{p(x)}{q(x)}]$$



假设这个是 p 的 distribution, 这个是 q 的 distribution, 这个是 $f(x)$, 那如果我们要计算 $f(x)$ 的期望值, 它的 distribution 是从 p 这个 distribution 做 sample, 那显然这一项是负的。因为 $f(x)$ 在这个区域, 这个区域 $p(x)$ 的机率很高, 所以要 sample 的话, 都会 sample 到这个地方, 而 $f(x)$ 在这个区域是负的, 所以理论上这一项算出来会是负的。

接下来我们改成从 q 这边做 sample, 那因为 q 在右边这边的机率比较高, 所以如果你 sample 的点不够的话, 那你可能都只 sample 到右侧, 如果你都只 sample 到右侧的话, 你会发现说, 如果只 sample 到右侧的话, 算起来右边这一项, 你 sample 到这些点, 都是正的, 所以你去计算 $f(x) \frac{p(x)}{q(x)}$, 都是正的。

那为什么会这样, 那是因为你 sample 的次数不够多。假设你 sample 次数很少, 你只能 sample 到右边这边, 左边这边虽然机率很低, 但也不是没有可能被 sample 到, 假设你今天好不容易 sample 到左边的点, 因为左边的点 pq 是差很多的, p 很大, q 很小, 这个负的就会被乘上一个非常巨大的 weight, 就可以平衡掉刚才那边, 一直 sample 到 positive 的 value 的情况。eventually, 你就可以算出这一项的期望值, 终究还是负的。

但问题就是, 这个前提是你要 sample 够多次, 这件事情才会发生。如果 sample 不够, 左式跟右式就有可能有很大的差距, 所以这是 importance sampling 的问题。

On-policy → Off-policy

On-policy → Off-policy

$$\nabla \bar{R}_\theta = E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)]$$

- Use π_θ to collect data. When θ is updated, we have to sample training data again.
- Goal: Using the sample from $\pi_{\theta'}$ to train θ . θ' is fixed, so we can re-use the sample data.

$$\nabla \bar{R}_\theta = E_{\tau \sim p_{\theta'}(\tau)} \left[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)} R(\tau) \nabla \log p_\theta(\tau) \right]$$

- Sample the data from θ' .
- Use the data to train θ many times.

Importance Sampling

$$E_{x \sim p}[f(x)] = E_{x \sim q}[f(x) \frac{p(x)}{q(x)}]$$

现在要做的事情就是，把 importance sampling 这件事，用在 off-policy 的 case。

我要把 on-policy training 的 algorithm，改成 off-policy training 的 algorithm。

那怎么改呢？之前我们是看，我们是拿 θ 这个 policy，去跟环境做互动，sample 出 trajectory τ ，然后计算中括号里面这一项，现在我们不根据 θ ，我们不用 θ 去跟环境做互动，我们假设有另外一个 policy，另外一个 policy 它的参数 θ' ，它就是另外一个 actor，它的工作是他要做 demonstration，它要去示范给你看，这个 θ' 它的工作是要去示范给 θ 看，它去跟环境做互动，告诉 θ 说，它跟环境做互动会发生什么事，然后，借此来训练 θ ，我们要训练的是 θ 这个 model， θ' 只是负责做 demo 负责跟环境做互动，我们现在的 τ ，它是从 θ' sample 出来的，不是从 θ sample 出来的，但我们本来要求的式子是这样，但是我们实际上做的时候，是拿 θ' 去跟环境做互动，所以 sample 出来的 τ ，是从 θ' sample 出来的，这两个 distribution 不一样。

但没有关系，我们之前讲过说，假设你本来是从 p 做 sample，但你发现你不能够从 p 做 sample，所以现在我们说我们不拿 θ 去跟环境做互动，所以不能跟 p 做 sample，你永远可以把 p 换成另外一个 q ，然后在后面这边补上一个 importance weight。

所以现在的状况就是一样，把 θ 换成 θ' 以后，要在中括号里面补上一个 importance weight，这个 importance weight 就是某一个 trajectory τ ，它用 θ 算出来的机率，除以这个 trajectory τ ，用 θ' 算出来的机率。

这一项是很重要的，因为，今天你要 learn 的是 actor θ and θ' 是不太一样的， θ' 会遇到的状况，会见到的情形，跟 θ 见到的情形，不见得是一样的，所以中间要做一个修正的项。

所以我们做了一下修正，现在的 data 不是从 θ' sample 出来的，是从 θ sample 出来的，那我们从 θ 换成 θ' 有什么好处呢？我们刚才就讲过说，因为现在跟环境做互动是 θ' 而不是 θ ，所以你今天 sample 出来的东西，跟 θ 本身是没有关系的，所以你就可以让 θ' 做互动 sample 一大堆的 data 以后， θ 可以 update 参数很多次，然后一直到 θ 可能 train 到一定的程度，update 很多次以后， θ' 再重新去做 sample，这就是 on-policy 换成 off-policy 的妙用。

On-policy → Off-policy

Gradient for update

$$\nabla f(x) = f(x) \nabla \log f(x)$$

$$= E_{(s_t, a_t) \sim \pi_\theta} [A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n)]$$

$A^{\theta'}(s_t, a_t)$ This term is from sampled data.

$$= E_{(s_t, a_t) \sim \pi_\theta} \left[\frac{P_\theta(s_t, a_t)}{P_{\theta'}(s_t, a_t)} A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right]$$

$$= E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} \frac{p_\theta(s_t)}{p_{\theta'}(s_t)} A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right]$$

$$J^{\theta'}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right] \text{ When to stop?}$$

那我们其实讲过，实际上我们在做 policy gradient 的时候，我们并不是给一整个 trajectory τ 都一样的分数，而是每一个 state/action 的 pair，我们会分开来计算，所以我们上周其实有讲过说，我们实际上 update 我们的 gradient 的时候，我们的式子是长这样子的。

我们用 θ 这个 actor 去 sample 出 state 跟 action 的 pair，我们会计算这个 state 跟 action pair 它的 advantage，就是它有多好，这一项就是那个 accumulated 的 reward 减掉 bias，这一项是估测出来的，它要估测的是，现在在 state s_t 采取 action a_t ，它是好的，还是不好的。那接下来后面会乘上这个 $\nabla \log p_\theta(a_t^n | s_t^n)$ ，也就是说如果这一项是正的，就要增加机率，这一项是负的，就要减少机率，那我们现在用了 importance sampling 的技术把 on-policy 变成 off-policy，就从 θ 变成 θ' 。

所以现在 s_t at a_t 它不是 θ 跟环境互动以后所 sample 到的 data，它是 θ' ，另外一个 actor 跟环境互动以后，所 sample 到的 data，但是拿来训练我们要调整参数的那个 model θ ，但是我们有讲过说，因为 θ' 跟 θ 是不同的 model，所以你要做一个修正的项，那这项修正的项，就是用 importance sampling 的技术，把 s_t at a_t 用 θ sample 出来的机率，除掉 s_t at a_t 用 θ' sample 出来的机率。

那这边其实有一件事情我们需要稍微注意一下，这边 A 有一个上标 θ 代表说，这个是 actor θ 跟环境互动的时候，所计算出来的 A ，但是实际上我们今天从 θ 换到 θ' 的时候，这一项，你其实应该改成 θ' ，而不是 θ 。为什么？ A 这一项是想要估测说现在在某一个 state，采取某一个 action 接下来，会得到 accumulated reward 的值减掉 baseline。在这个 state s_t ，采取这个 action a_t ，接下来会得到的 reward 的总和，再减掉 baseline，就是这一项。

之前是 θ 在跟环境做互动，所以你观察到的是 θ 可以得到的 reward，但现在不是 θ 跟环境做互动，现在是 θ' 在跟环境做互动，所以你得到的这个 advantage，其实是根据 θ' 所 estimate 出来的 advantage，但我们现在先不要管那么多，我们就假设这两项可能是差不多的。

那接下来， s_t at 的机率，你可以拆解成 s_t 的机率乘上 at given s_t 的机率。

接下来这边需要做一件事情是，我们假设当你的 model 是 θ 的时候，你看到 s_t 的机率，跟你的 model 是 θ' 的时候，你看到 s_t 的机率，是差不多的，你把它删掉，因为它们是一样的。

为什么可以假设它是差不多的，当然你可以找一些理由，举例来说，会看到什么 state，往往跟你会采取什么样的 action 是没有太大的关系的，也许不同的 θ 对 s_t 是没有影响的。

但是有一个更直觉的理由就是，这一项到时候真的要你算，你会算吗？你不觉得这项你不太能算吗？因为想想看这项要怎么算，这一项你还要说，我有一个参数 θ ，然后拿 θ 去跟环境做互动，算 s_t 出现的机率，这个你根本很难算，尤其是你如果 input 是 image 的话，同样的 s_t 根本就不会出现第二次。所以你根本没有办法估这一项，干脆就无视这个问题。这一项其实不太好算，所以你就说服自己，其实这一项不太会有影响，我们只管前面这个部分就好了。

但是 given s_t ，接下来产生 a_t 这个机率，你是会算的，这个很好算，你手上有 θ 这个参数，它就是个 network，你就把 s_t 带进去， s_t 就是游戏画面，你把游戏画面带进去，它就会告诉你某一个 state 的 a_t 机率是多少。我们其实有个 policy 的 network，把 s_t 带进去，它会告诉我们每一个 a_t 的机率是多少，所以这一项你只要知道 θ 的参数，知道 θ' 的参数，这个就可以算。

这一项是 gradient，其实我们可以从 gradient 去反推原来的 objective function，怎么从 gradient 去反推原来的 objective function 呢？这边有一个公式，我们就背下来， $f(x)$ 的 gradient，等于 $f(x)$ 乘上 $\log f(x)$ 的 gradient。把公式带入到 gradient 的项，还原原来没有取 gradient 的样子。那所以现在我们得到一个新的 objective function。

所以实际上，当我们 apply importance sampling 的时候，我们要去 optimize 的那一个 objective function 长什么样子呢，我们要去 optimize 的那一个 objective function 就长这样子，

$$J^{\theta'}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right]$$

这个括号里面那个 θ 代表我们要去 optimize 的那个参数，我们拿 θ' 去做 demonstration。

现在真正在跟环境互动的是 θ' ，sample 出 s_t 和 a_t 以后，那你要去计算 s_t 跟 a_t 的 advantage，然后你再去把它乘上 $\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)}$ ，这两项都是好算的，advantage 是可以从 sample 的结果里面去估测出来的，所以这一整项，你是可以算的。

那我们实际上在 update 参数的时候，就是按照上面这个式子 update 参数。现在我们做的事情，我们可以把 on-policy 换成 off-policy，但是我们会遇到的问题是，我们在前面讲 importance sampling 的时候，我们说 importance sampling 有一个 issue，这个 issue 是什么呢？其实你的 $p(\theta)$ 跟 $p(\theta')$ 不能差太多。差太多的话，importance sampling 结果就会不好。

所以怎么避免它差太多呢？这个就是 PPO 在做的事情。

Add Constraint

稳扎稳打，步步为营

PPO / TRPO

PPO 你虽然你看它原始的 paper 或你看 PPO 的前身 TRPO 原始的 paper 的话，它里面写了很多的数学式，但它实际上做的事情怎么样呢？

它实际上做的事情就是这样：

我们原来在 off-policy 的方法里面说，我们要 optimize 的是这个 objective function，但是我们又说这个 objective function 又牵涉到 importance sampling，在做 importance sampling 的时候， $p(\theta)$ 不能跟 $p(\theta')$ 差太多，你做 demonstration 的 model 不能够跟真正的 model 差太多，差太多的话 importance sampling 的结果就会不好。

我们在 training 的时候，多加一个 constrain。这个 constrain 是什么？这个 constrain 是 θ 跟 θ' ，这两个 model 它们 output 的 action 的 KL divergences。

就是简单来说，这一项的意思就是要衡量说 θ 跟 θ' 有多像，然后我们希望，在 training 的过程中，我们 learn 出来的 θ 跟 θ' 越像越好，因为 θ 如果跟 θ' 不像的话，最后你做出来的结果，就会不好。

所以在 PPO 里面呢，有两个式子，一方面就是 optimize 你要得到的你本来要 optimize 的东西。但是再加一个 constrain，这个 constrain 就好像那个 regularization 的 term 一样，就好像我们在做 machine learning 的时候不是有 L1/L2 的 regularization，这一项也很像 regularization，这样 regularization 做的事情就是希望最后 learn 出来的 θ ，不要跟 θ' 太不一样。

PPO / TRPO

θ cannot be very different from θ'

Constraint on behavior not parameters

Proximal Policy Optimization (PPO)

$$\nabla f(x) = f(x)\nabla \log f(x)$$

$$J_{PPO}^{\theta'}(\theta) = J^{\theta'}(\theta) - \beta KL(\theta, \theta')$$

$$J^{\theta'}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_{\theta}(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right]$$

TRPO (Trust Region Policy Optimization)

$$J_{TRPO}^{\theta'}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_{\theta}(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right]$$

$$KL(\theta, \theta') < \delta$$

那 PPO 有一个前身叫做 TRPO，TRPO 写的式子是这个样子的，它唯一不一样的地方是说，这一个 constrain 摆的位置不一样，PPO 是直接把 constrain 放到你要 optimize 的那个式子里面，然后接下来你就可以用 gradient ascent 的方法去 maximize 这个式子，但是如果是在 TRPO 的话，它是把 KL diversions 当作 constrain，他希望 θ 跟 θ' 的 KL diversions，小于一个 δ 。

那你知道如果你是用 gradient based optimization 的时候，有 constrain 是很难处理的。那个是很难处理的，就是因为它是把这一个 KL diversions constrain 当做一个额外的 constrain，没有放 objective 里面，所以它很难算，所以如果你不想搬石头砸自己的脚的话，你就用 PPO 不要用 TRPO。

看文献上的结果是，PPO 跟 TRPO 可能 performance 差不多，但是 PPO 在实作上，比 TRPO 容易的多。

那这边要注意一下，所谓的 KL diversions，到底指的是什么？这边我是直接把 KL diversions 当做一个 function，它吃的 input 是 θ 跟 θ' ，但我的意思并不是说把 θ 和 θ' 当做 distribution，算这两个 distribution 之间的距离，今天这个所谓的 θ 跟 θ' 的距离，并不是参数上的距离，而是它们 behavior 上的距离，我不知道大家可不可以了解这中间的差异，就是假设你现在有一个 model，有一个 actor 它的参数是 θ ，你有另外一个 actor 它的参数是 θ' ，所谓参数上的距离就是你算这两组参数有多像，今天所讲的不是参数上的距离，今天所讲的是它们行为上的距离，就是你先带进去一个 state s ，它会对这个 action 的 space output 一个 distribution，假设你有 3 个 actions，3 个可能 actions 就 output 3 个值。

那我们今天所指的 distance 是 behavior distance，也就是说，给同样的 state 的时候，他们 output 的 action 之间的差距，这两个 actions 的 distribution 他们都是一个机率分布，所以就可以计算这两个机率分布的 KL diversions。

把不同的 state 它们 output 的这两个 distribution 的 KL diversions，平均起来，才是我这边所指的这两个 actor 间的 KL diversions。

那你可能说那怎么不直接算这个 θ 或 θ' 之间的距离，甚至不要用 KL diversions 算，L1 跟 L2 的 norm 也可以保证， θ 跟 θ' 很接近。

在做 reinforcement learning 的时候，之所以我们考虑的不是参数上的距离，而是 action 上的距离，是因为很有可能对 actor 来说，参数的变化跟 action 的变化，不一定是完全一致的，就有时候你参数小小变了一下，它可能 output 的行为就差很多，或是参数变很多，但 output 的行为可能没什么改变。所以我们真正在意的是这个 actor 它的行为上的差距，而不是它们参数上的差距。

所以这里要注意一下，在做 PPO 的时候，所谓的 KL diversions 并不是参数的距离，而是 action 的距离。

PPO algorithm

PPO algorithm

- Initial policy parameters θ^0
- In each iteration
 - Using θ^k to interact with the environment to collect $\{s_t, a_t\}$ and compute advantage $A^{\theta^k}(s_t, a_t)$
 - Find θ optimizing $J_{PPO}(\theta)$

$$J^{\theta^k}(\theta) \approx \sum_{(s_t, a_t)} \frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)} A^{\theta^k}(s_t, a_t)$$

$$J_{PPO}^{\theta^k}(\theta) = J^{\theta^k}(\theta) - \beta KL(\theta, \theta^k)$$

Update parameters several times

- If $KL(\theta, \theta^k) > KL_{max}$, increase β
- If $KL(\theta, \theta^k) < KL_{min}$, decrease β

Adaptive KL Penalty

我们来看一下 PPO 的 algorithm，它就是这样，initial 一个 policy 的参数 θ^0 ，然后在每一个 iteration 里面呢，你要用你在前一个 training 的 iteration，得到的 actor 的参数 θ^k ，去跟环境做互动，sample 到一大堆 state/action 的 pair，然后你根据 θ^k 互动的结果，你也要估测一下，st 跟 at 这个 state/action pair 它的 advantage，然后接下来，你就 apply PPO 的 optimization 的 formulation。

但是跟原来的 policy gradient 不一样，原来的 policy gradient 你只能 update 一次参数，update 完以后，你就要重新 sample data。但是现在不用，你拿 θ^k 去跟环境做互动，sample 到这组 data 以后，你就努力去 train θ ，你可以让 θ update 很多次，想办法去 maximize 你的 objective function，你让 θ update 很多次，这边 θ update 很多次没有关系，因为我们已经有做 importance sampling，所以这些 experience，这些 state/action 的 pair 是从 θ^k sample 出来的是没有关系的， θ 可以 update 很多次，它跟 θ^k 变得不太一样也没有关系，你还是可以照样训练 θ ，那其实就说完了。

在 PPO 的 paper 里面，这边还有一个 adaptive 的 KL diversions，因为这边会遇到一个问题就是，这个 β 要设多少，它就跟那个 regularization 一样，regularization 前面也要乘一个 weight，所以这个 KL diversions 前面也要乘一个 weight。但是 β 要设多少呢？所以有个动态调整 β 的方法。

这个调整方法也是蛮直观的，在这个直观的方法里面呢，你先设一个 KL diversions，你可以接受的最大值，然后假设你发现说，你 optimize 完这个式子以后，KL diversions 的项太大，那就代表说后面这个 penalize 的 term 没有发挥作用，那就把 β 调大，那另外你定一个 KL diversions 的最小值，而且发现 optimize 完上面这个式子以后，你得到 KL diversions 比最小值还要小，那代表后面这一项它的效果太强了，怕它都只弄后面这一项，那 θ 跟 θ^k 都一样，这不是你要的，所以你这个时候你就要减少 β ，所以这个 β 是可以动态调整的，这个叫做 adaptive 的 KL penalty。

PPO2 algorithm

如果你觉得这个很复杂，有一个 PPO2。

PPO2 它的式子我们就写在这边，要去 maximize 的 objective function 写成这样，它的式子里面就没有什么 KL 了。

这个式子看起来有点复杂，但实际 implement 就很简单。

我们来实际看一下说这个式子到底是什么意思，这边是 summation over state/action 的 pair，min 这个 operator 做的事情是，第一项跟第二项里面选比较小的那个，第一项比较单纯，第二项比较复杂，第二项前面有个 clip function，clip 这个 function 是什么意思呢？clip 这个 function 的意思是说，在括号里面有 3 项，如果第一项小于第二项的话，那就 output $1-\epsilon$ ，第一项如果大于第三项的话，那就 output $1+\epsilon$ ，那 ϵ 是一个 hyper parameter，你要 tune 的，比如说你就设 0.1、0.2。也就是说，假设这边设 0.2 的话，就是说这个值如果算出来小于 0.8，那就当作 0.8，这个值如果算出来大于 1.2，那就当作 1.2。

这个式子到底是什么意思呢？我们先来解释一下，我们先来看第二项这个算出来到底是什么的东西。

第二项这项算出来的意思是这样，假设这个横轴是 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ ，纵轴是 clip 这个 function 它实际的输出，那我们刚才讲过说，如果 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ 大于 $1+\epsilon$ ，它输出就是 $1+\epsilon$ ，如果小于 $1-\epsilon$ 它输出就是 $1-\epsilon$ ，如果介于 $1+\epsilon$ 跟 $1-\epsilon$ 之间，就是输入等于输出。 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ 跟 clip function 输出的关系，是这样的一个关系。

PPO algorithm

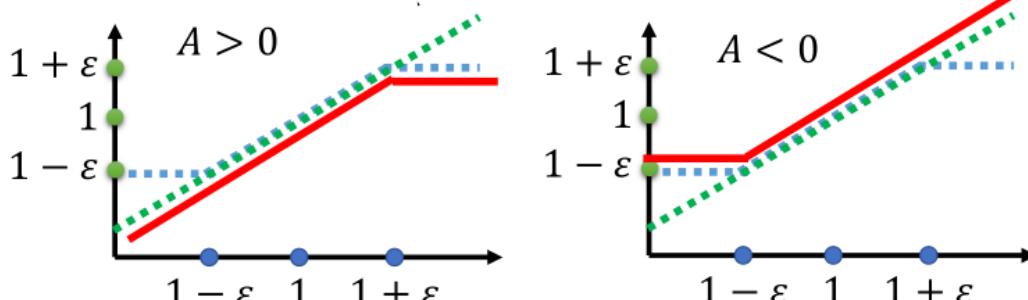
$$J_{PPO}^{\theta^k}(\theta) = J^{\theta^k}(\theta) - \beta KL(\theta, \theta^k)$$

$$J^{\theta^k}(\theta) \approx \sum_{(s_t, a_t)} \frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)} A^{\theta^k}(s_t, a_t)$$

PPO2 algorithm

$$J_{PPO2}^{\theta^k}(\theta) \approx \sum_{(s_t, a_t)} \min \left(\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)} A^{\theta^k}(s_t, a_t), \right.$$

$$\left. \text{clip} \left(\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon \right) A^{\theta^k}(s_t, a_t) \right)$$



接下来，我们就加入前面这一项，来看看前面这一项，到底在做什么？前面这一项呢，其实就是绿色的这一条线，就是绿色的这一条线。这两项里面，第一项跟第二项，也就是绿色的线，跟蓝色的线中间，我们要取一个最小的。假设今天前面乘上的这个 term A，它是大于 0 的话，取最小的结果，就是红色的这一条线。反之，如果 A 小于 0 的话，取最小的以后，就得到红色的这一条线，这一个结果，其实非常的直观，这一个式子虽然看起来有点复杂，implement 起来是蛮简单的，想法也非常的直观。

因为这个式子想要做的事情，就是希望 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ ，也就是你拿来做 demonstration 的那个 model，跟你实际上 learn 的 model，最后在 optimize 以后，不要差距太大。那这个式子是怎么让它做到不要差距太大的呢？

复习一下这横轴的意思，就是 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ ，如果今天 A 大于 0，也就是某一个 state/action 的 pair 是好的，那我们想要做的事情，当然是希望增加这个 state/action pair 的机率。也就是说，我们想要让 p_θ 越大越好，没有问题，但是，它跟这个 θ^k 的比值，不可以超过 $1 + \text{epsilon}$ 。红色的线就是我们的 objective function，我们希望我们的 objective 越大越好，比值只要大过 $1 + \text{epsilon}$ ，就没有 benefit 了，所以今天在 train 的时候， p_θ 只会被 train 到，比 p_{θ^k} 它们相除大 $1 + \text{epsilon}$ ，它就会停止。

那假设今天不幸的是， p_θ 比 p_{θ^k} 还要小，假设这个 advantage 是正的，我们当然希望 p_θ 越大越好，假设这个 action 是好的，我们当然希望这个 action 被采取的机率，越大越好，所以假设 p_θ 还比 p_{θ^k} 小，那就尽量把它挪大，但只要大到 $1 + \text{epsilon}$ 就好。

那负的时候也是一样，如果今天，某一个 state/action pair 是不好的，我们当然希望 p_θ 把它减小，假设今天 p_θ 比 p_{θ^k} 还大那你就要赶快尽量把它压小，那压到什么样就停止呢？，压到 p_θ 除以 p_{θ^k} 是 $1 - \text{epsilon}$ 的时候，就停了，就算了，就不要再压得更小。

那这样的好处就是，你不会让 p_θ 跟 p_{θ^k} 差距太大，那要 implement 这个东西，其实对你来说可能不是太困难的事情。

Experimental Results

<https://arxiv.org/abs/1707.06347>

Experimental Results

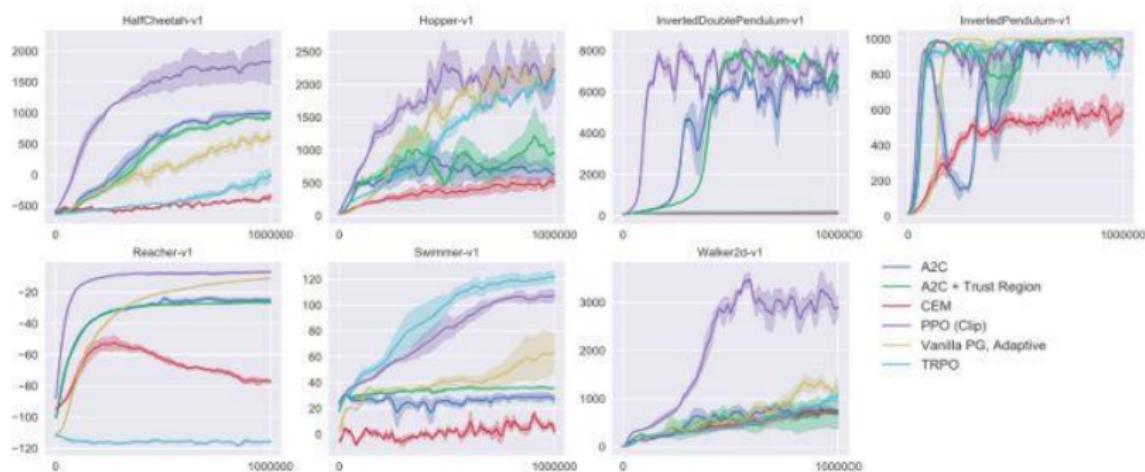


Figure 3: Comparison of several algorithms on several MuJoCo environments, training for one million timesteps.

那最后这页投影片呢，只是想要 show 一下，在文献上，PPO 跟其它方法的比较，有 Actor-Critic 的方法，这边有 A2C+TrustRegion，他们都是 Actor-Critic based 的方法，然后这边有 PPO，PPO 是紫色线的方法，然后还有 TRPO。

PPO 就是紫色的线，那你会发现在多数的 task 里面，这边每张图就是某一个 RL 的任务，你会发现说在多数的 cases 里面，PPO 都是不错的，不是最好的，就是第二好的。

Q-Learning

Introduction of Q-Learning

Critic

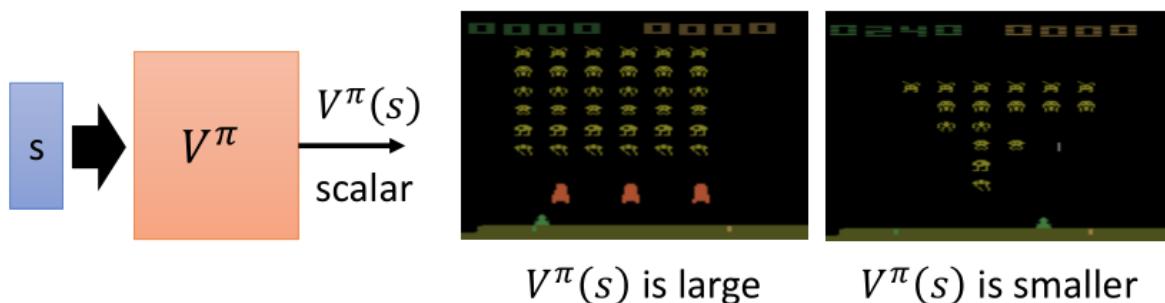
Q-learning 这种方法，它是 value-based 的方法，在 value based 的方法里面，我们并不是直接 learn policy，我们要 learn 的是一个 critic，critic 并不直接采取行为，它想要做的事情是，评价现在的行为有多好或者是有多不好。

这边说的是 critic 并不直接采取行为，它是说我们假设有一个 actor π ，那 critic 的工作就是来评价这个 actor π 它做得有多好，或者是有多不好。

Critic

The output values of a critic depend on the actor evaluated.

- A critic does not directly determine the action.
- Given an actor π , it evaluates how good the actor is
- State value function $V^\pi(s)$
 - When using actor π , the *cumulated* reward expects to be obtained after visiting state s



举例来说，有一种 function 叫做 state value 的 function。这个 state value function 的意思就是说，假设现在的 actor 叫做 π ，拿这个 π 跟环境去做互动。拿 π 去跟环境做互动的时候，现在假设 π 这个 actor，它看到了某一个 state s ，那如果在玩游戏的话，state s 是某一个画面，看到某一个画面，某一个 state s 的时候，接下来一直玩到游戏结束，累积的 reward 的期望值有多大，accumulated reward 的 expectation 有多大。

所以 V^π 它是一个 function，这个 function 它是吃一个 state，当作 input，然后它会 output 一个 scalar，这个 scalar 代表说，现在 π 这个 actor 它看到 state s 的时候，接下来预期到游戏结束的时候，它可以得到多大的 value。

假设你是玩 space invader 的话，也许这个 state 这个 s ，这一个游戏画面，你的 $V^\pi(s)$ 会很大，因为接下来还有很多的怪可以杀，所以你会得到很大的分数，一直到游戏结束的时候，你仍然有很多的分数可以吃。

那在这个 case，也许你得到的 $V^\pi(s)$ ，就很小，因为一方面，剩下的怪也不多了，那再来就是现在因为那个防护罩，这个红色的东西防护罩已经消失了，所以可能很快就会死掉，所以接下来得到预期的 reward，就不会太大。

那这边需要强调的一个点是说，当你在讲这个 critic 的时候，你一定要注意，critic 都是绑一个 actor 的，就 critic 它并没有办法去凭空去 evaluate 一个 state 的好坏，而是它所 evaluate 的东西是在 given 某一个 state 的时候，假设我接下来互动的 actor 是 π ，那我会得到多少 reward，因为就算是给同样的 state，你接下来的 π 不一样，你得到的 reward 也是不一样的，举例来说，在这个 case，虽然假设是一个正常的 π ，它可以杀很多怪，那假设它是一个很弱的 π ，它就站在原地不动，然后马上就被射死了，那你得到的 V 还是很小，所以今天这个 critic output 值有多大，其实是取决于两件事，一个是 state，另外一个其实是 actor。所以今天你的 critic 其实都要绑一个 actor，它是在衡量某一个 actor 的好坏，而不是 generally 衡量一个 state 的好坏，这边有强调一下，你这个 critic output 是跟 actor 有关的。

你的 state value 其实是 depend on 你的 actor，当你的 actor 变的时候，你的 state value function 的 output，其实也是会跟着改变的。

How to estimate $V^\pi(s)$

再来问题就是，怎么衡量这一个 state value function 呢？怎么衡量这一个 $V^\pi(s)$ 呢？有两种不同的作法，那等一下会讲说，像这种 critic，它是怎么演变成可以真的拿来采取 action。我们现在要先问的是怎么 estimate 这些 critic。

Monte-Carlo (MC) based approach

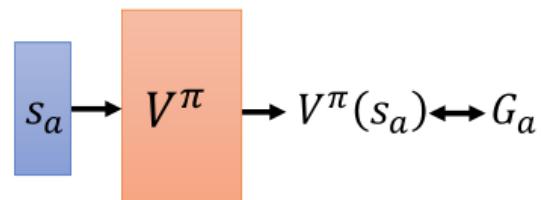
那怎么 estimate $V^\pi(s)$ 呢，有两个方向，一个是用 Monte-Carlo MC based 的方法，如果是 MC based 的方法，它非常的直觉，它怎么直觉呢，它就是说，你就让 actor 去跟环境做互动，你要量 actor 好不好，你就让 actor 去跟环境做互动，给 critic 看，然后，接下来 critic 就统计说，这个 actor 如果看到 state s_a ，它接下来 accumulated reward，会有多大，如果它看到 state s_b ，它接下来 accumulated reward，会有多大。但是实际上，你当然不可能把所有的 state 通通都扫过，不要忘了如果你是玩 Atari 游戏的话，你的 state 可是 image，你可是没有办法把所有的 state 通通扫过。

- **Monte-Carlo (MC) based approach**

- The critic watches π playing the game

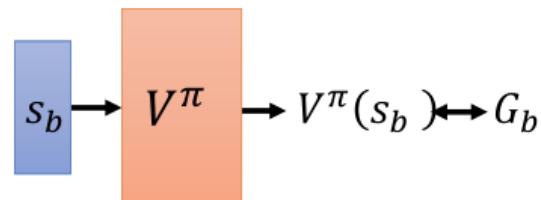
After seeing s_a ,

Until the end of the episode,
the cumulated reward is G_a



After seeing s_b ,

Until the end of the episode,
the cumulated reward is G_b



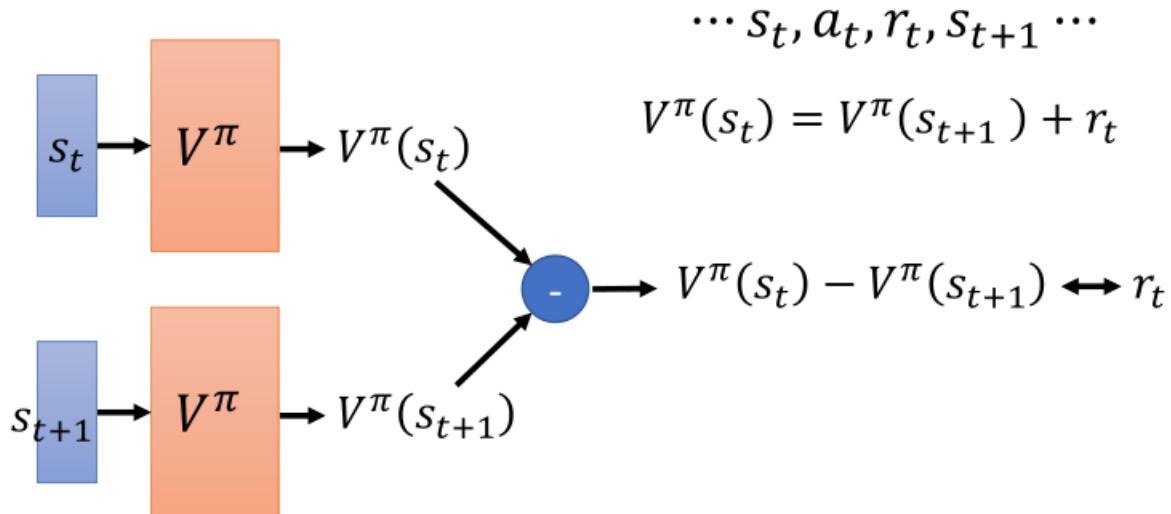
所以实际上我们的 $V^\pi(s)$ ，它是一个 network，对一个 network 来说，就算是 input state 是从来都没有看过的，它也可以想办法估测一个 value 的值。

怎么训练这个 network 呢？因为我们现在已经知道说，如果在 state s_a ，接下来的 accumulated reward 就是 G_a ，也就是说，今天对这 value function 来说，如果 input 是 state s_a ，正确的 output 应该是 G_a ，如果 input state s_b ，正确的 output 应该是 value G_b ，所以在 training 的时候，其实它就是一个 regression 的 problem，你的 network 的 output 就是一个 value，你希望在 input s_a 的时候，output value 跟 G_a 越近越好，input s_b 的时候，output value 跟 G_b 越近越好，接下来把 network train 下去，就结束了。这是第一个方法，这是 MC based 的方法。

Temporal-difference (TD) approach

那还有第二个方法是 Temporal-difference 的方法，这个是 TD based 的方法，那 TD based 的方法是什么意思呢？在刚才那个 MC based 的方法，每次我们都要算 accumulated reward，也就是从某一个 state S_a ，一直玩游戏玩到游戏结束的时候，你会得到的所有 reward 的总和，我在前一个投影片里面，把它写成 G_a 或 G_b ，所以今天你要 apply MC based 的 approach，你必须至少把这个游戏玩到结束，你才能够估测 MC based 的 approach。但是有些游戏非常的长，你要玩到游戏结束才能够 update network，你可能根本收集不到太多的数据，花的时间太长了。所以怎么办？

Temporal-difference (TD) approach



Some applications have very long episodes, so that delaying all learning until an episode's end is too slow.

有另外一种 TD based 的方法，TD based 的方法，不需要把游戏玩到底，只要在游戏的某一个情况，某一个 state s_t 的时候，采取 action a_t ，得到 reward r_t ，跳到 state $s(t+1)$ ，就可以 apply TD 的方法，怎么 apply TD 的方法呢？基于以下这个式子，这个式子是说，我们知道说，假设我们现在用的是某一个 policy π ，在 state s_t ，以后在 state s_t ，它会采取 action a_t ，给我们 reward r_t ，接下来进入 $s(t+1)$ ，那就告诉我们说，state $s(t+1)$ 的 value，跟 state s_t 的 value，它们的中间差了一项 r_t ，因为你把 $s(t+1)$ 得到的 value，加上这边得到的 reward r_t ，就会等于 s_t 得到的 value。

有了这个式子以后，你在 training 的时候，你要做的事情并不是真的直接去估测 V ，而是希望你得到的结果，你得到的这个 V ，可以满足这个式子。

也就是说你 training 的时候，会是这样 train 的：你把 s_t 丢到 network 里面，会得到 V of s_t ，你把 $s(t+1)$ 丢到你的 value network 里面，会得到 V of $s(t+1)$ 。 V of s_t 减 V of $s(t+1)$ ，它得到的值应该是 r_t 。然后按照这样的 loss，希望它们两个相减跟 r_t 越接近越好的 loss train 下去，update V 的参数，你就可以把 V function learn 出来。

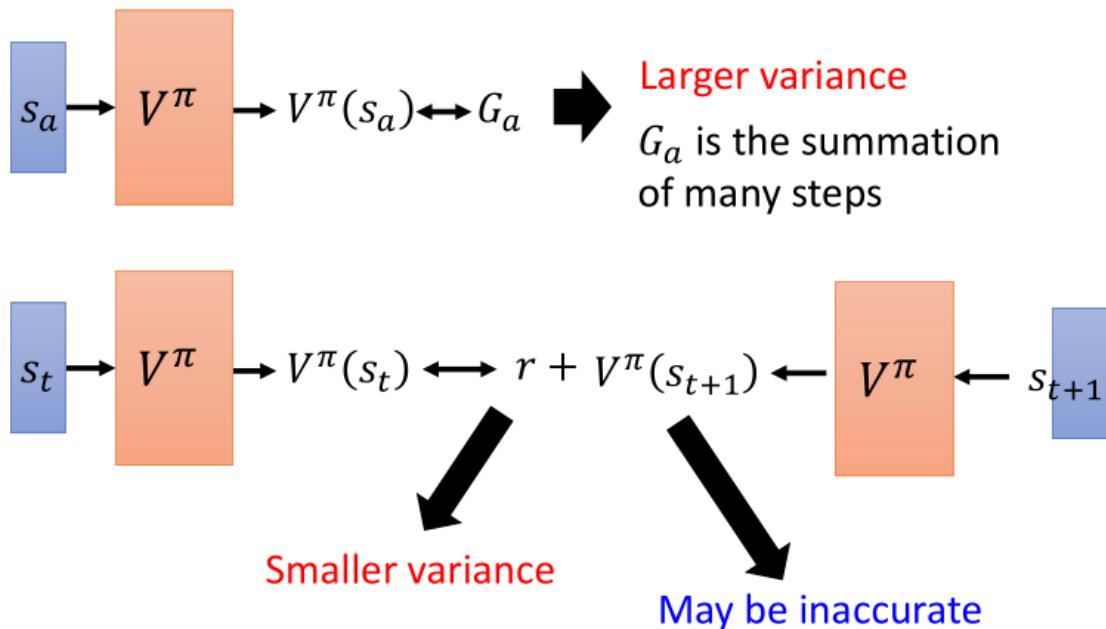
MC v.s. TD

这边是比较一下 MC 跟 TD 之间的差别，那 MC 跟 TD 它们有什么样的差别呢？

MC 它最大的问题就是它的 variance 很大。因为今天我们在玩游戏的时候，它本身是有随机性的，所以 G_a 本身你可以想成它其实是一个 random 的 variable，因为你每次同样走到 s_a 的时候，最后你得到的 G_a ，其实是不一样的。你看到同样的 state s_a ，最后玩到游戏结束的时候，因为游戏本身是有随机性的，你的玩游戏的 model 本身搞不好也有随机性，所以你每次得到的 G_a 是不一样的。那每一次得到 G_a 的差别，其实会很大。为什么它会很大呢？假设你每一个 step 都会得到一个 reward， G_a 是从 state s_a 开始，一直玩到游戏结束，每一个 time step reward 的和。

$$Var[kX] = k^2 Var[X]$$

MC v.s. TD



那举例来说，我在右上角就列一个式子是说，假设本来只有 X ，它的 variance 是 var of X ，但是你把某一个 variable 乘上 K 倍的时候，它的 variance 就会变成原来的 K 平方。所以 G_a 的 variance 相较于某一个 state，你会得到的 reward variance 是比较大的。

如果说用 TD 的话呢？用 TD 的话，你是要去 minimize 这样的一个式子，在这中间会有随机性的是 r ，因为你在 s_t 就算你采取同一个 action，你得到的 reward 也不见得是一样的，所以 r 其实也是一个 random variable，但这个 random variable 它的 variance，会比 G_a 要小，因为 G_a 是很多 r 合起来，这边只是某一个 r 而已， G_a 的 variance 会比较大， r 的 variance 会比较小。但是这边你会遇到的一个问题是 V 不见得估的准，假设你的这个 V 估的是不准的，那你 apply 这个式子 learn 出来的结果，其实也会是不准的。

所以今天 MC 跟 TD，它们是各有优劣，那等一下其实会讲一个 MC 跟 TD 综合的版本。今天其实 TD 的方法是比较常见的，MC 的方法其实是比较少用的。

MC v.s. TD

[Sutton, v2,
Example 6.4]

- The critic has the following 8 episodes

$s_a, r = 0, s_b, r = 0, \text{END}$	
$s_b, r = 1, \text{END}$	$V^\pi(s_b) = 3/4$
$s_b, r = 1, \text{END}$	
$s_b, r = 1, \text{END}$	$V^\pi(s_a) = ? \quad 0? \quad 3/4?$
$s_b, r = 1, \text{END}$	
$s_b, r = 1, \text{END}$	Monte-Carlo: $V^\pi(s_a) = 0$
$s_b, r = 1, \text{END}$	
$s_b, r = 0, \text{END}$	Temporal-difference: $V^\pi(s_a) = V^\pi(s_b) + r$ $3/4 \quad 3/4 \quad 0$

那这张图是想要讲一下，TD 跟 MC 的差异，这个图想要说的是什么呢？

这个图想要说的是，假设我们现在有某一个 critic，它去观察某一个 policy π_i ，跟环境互动8个 episode 的结果，有一个 actor π_i 它去跟环境互动了 8 次，得到了 8 次玩游戏的结果是这个样子。

接下来我们要这个 critic 去估测 state 的 value，那如果我们看 s_b 这个 state 它的 value 是多少， s_b 这个 state 在 8场游戏里面都有经历过，然后在这 8 场游戏里面，其中有 6 场得到 reward 1，再有两场得到 reward 0。所以如果你是要算期望值的话，看到 state s_b 以后，一直到游戏结束的时候，得到的 accumulated reward 期望值是 $3/4$ ，非常直觉。但是，不直觉的地方是说， s_a 期望的 reward 到底应该是多少呢？

这边其实有两个可能的答案，一个是 0，一个是 $3/4$ ，为什么有两个可能的答案呢？这取决于你用 MC 还是 TD。

假如你用 MC 的话，你用 MC 的话，你会发现说，这个 s_a 就出现一次，它就出现一次，看到 s_a 这个 state，接下来 accumulated reward 就是 0，所以今天 s_a 它的 expected reward 就是 0。

但是如果你今天去看 TD 的话，TD 在计算的时候，它是要 update 下面这个式子。下面这个式子想要说的事情是，因为我们在 state s_a 得到 reward $r=0$ 以后，跳到 state s_b ，所以 state s_a 的 reward，会等于 state s_b 的 reward，加上在 state s_a 它跳到 state s_b 的时候可能得到的 reward r 。而这个可能得到的 reward r 的值是多少？它的值是 0，而 s_b expected reward 是多少呢？它的 reward 是 $3/4$ 。那 s_a 的 reward 应该是 $3/4$ 。

有趣的地方是用 MC 跟TD 你估出来的结果，其实很有可能是不一样的，就算今天你的 critic observed 到一样的 training data，它最后估出来的结果，也不见得会是一样。那为什么会这样呢？你可能问，那一个比较对呢？其实都对，因为今天在 s_a 这边，今天在第一个 trajectory， s_a 它得到 reward 0 以后，再跳到 s_b 也得到 reward 0。

这边有两个可能，一个可能是，只要有看到 s_a 以后， s_b 就会拿不到 reward，有可能 s_a 其实影响了 s_b ，如果是用 MC 的算法的话，它就会考虑这件事，它会把 s_a 影响 s_b 这件事，考虑进去，所以看到 s_a 以后，接下来 s_b 就得不到 reward，所以看到 s_a 以后，期望的 reward 是 0。

但是今天看到 sa 以后， sb 的 reward 是 0 这件事有可能只是一个巧合，就并不是 sa 所造成，而是因为说， sb 有时候就是会得到 reward 0，这只是单纯运气的问题，其实平常 sb 它会得到 reward 期望值是 $3/4$ ，跟 sa 是完全没有关系的。所以 sa 假设之后会跳到 sb ，那其实得到的 reward 按照 TD 来算，应该是 $3/4$ 。

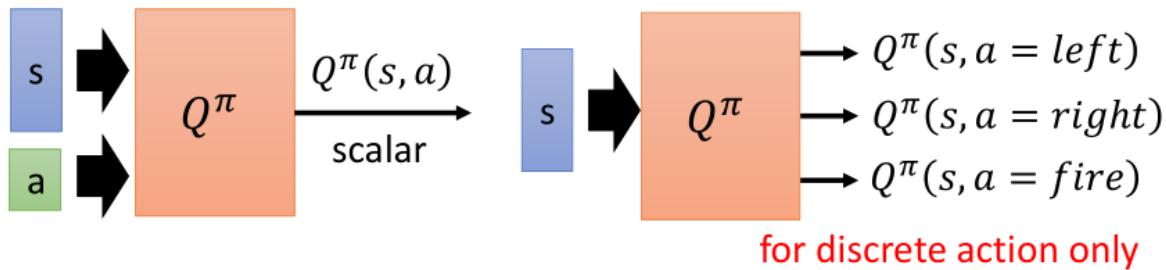
所以不同的方法，它考虑了不同的假设，最后你其实是会得到不同的运算结果的。

Another Critic

那接下来我们要讲的是另外一种 critic，这种 critic 叫做 Q function，它又叫做 state-action value function。

那我们刚才看到的那一个 state function，它的 input，就是一个 state，它是根据 state 去计算出看到这个 state 以后的 expected accumulated reward 是多少。

- State-action value function $Q^\pi(s, a)$
- When using actor π , the *cumulated* reward expects to be obtained after taking a at state s



那这个 state-action value function 它的 input 不是 state，它是一个 state 跟 action 的 pair，它的意思是说，在某一个 state，采取某一个 action，接下来假设我们都使用 actor π ，得到的 accumulated reward 它的期望值有多大。

在讲这个 Q-function 的时候，有一个会需要非常注意的问题是，今天这个 actor π ，在看到 state s 的时候，它采取的 action，不一定是 a 。Q function 是假设在 state s ，强制采取 action a ，不管你现在考虑的这个 actor π ，它会不会采取 action a 不重要，在 state s ，强制采取 action a ，接下来，都用 actor π 继续玩下去，就只有在 state s ，我们才强制一定要采取 action a ，接下来就进入自动模式，让 actor π 继续玩下去，得到的 expected reward，才是 $Q(s, a)$ 。

Q function 有两种写法，一种写法是你 input 就是 state 跟 action，那 output 就是一个 scalar，就跟 value function 是一样。

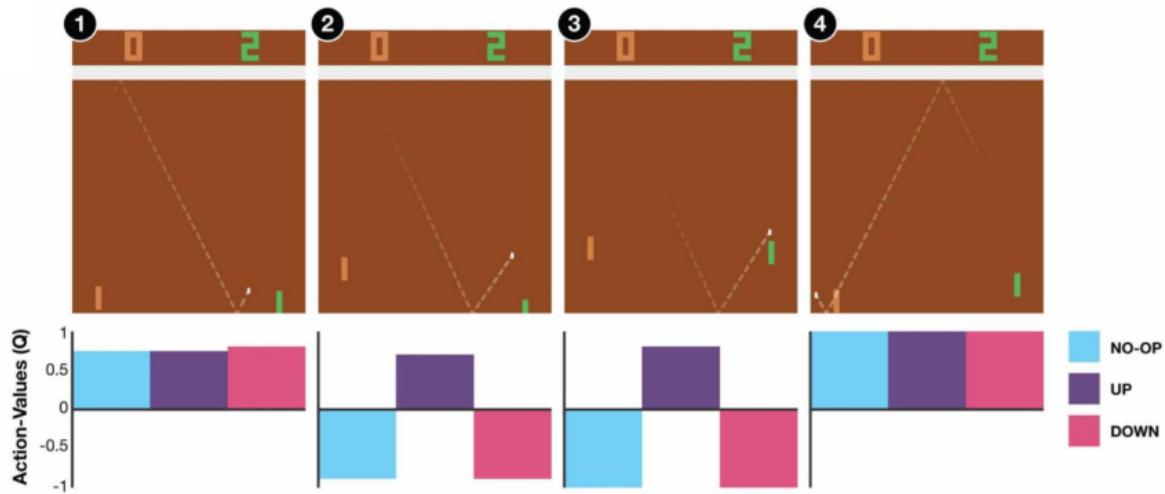
那还有另外一种写法，也许是常见的写法是这样，你 input 一个 state s ，接下来你会 output 好几个 value，假设你 action 是 discrete 的，你 action 就只有 3 个可能，往左往右或是开火，那今天你的这个 Q function output 的 3 个 values，就分别代表假设， a 是向左的时候的 Q value， a 是向右的时候的 Q value，还有 a 是开火的时候的 Q value。那你要注意的事情是，像这样的 function 只有 discrete action 才能够使用。如果你的 action 是无法穷举的，你只能用左边这个式子，不能够用右边这个式子。

State-action value function

这个是文献上的结果，一个碰的游戏你去 estimate Q function 的话，看到的结果可能会像是这个样子，这是什么意思呢？它说假设上面这个画面就是 state，我们有 3 个 actions，原地不动，向上，向下。那假设是在第一幅图这个 state，最后到游戏结束的时候，得到的 expected reward，其实都差不了多少，因为球在这个地方，就算是你向下，接下来你其实应该还来的急救，所以今天不管是采取哪一个 action，就差不了太多。但假设现在这个球，这个乒乓球它已经反弹到很接近边缘的地方，这个时候你

采取向上，你才能得到 positive 的 reward，才接的到球，如果你是站在原地不动或向下的话，接下来你都会 miss 掉这个球，你得到的 reward 就会是负的。这个 case 也是一样，球很近了，所以就要向上。接下来，球被反弹回去，这时候采取那个 action，就都没有差了。

大家应该都知道说，deep reinforcement learning 最早受到大家重视的一篇paper 就是 deep mind 发表在 Nature 上的那个 paper，就是用 DQN 玩 Atari 可以痛扁人类，这个是 state-action value 的一个例子，是那篇 paper 上截下来的。



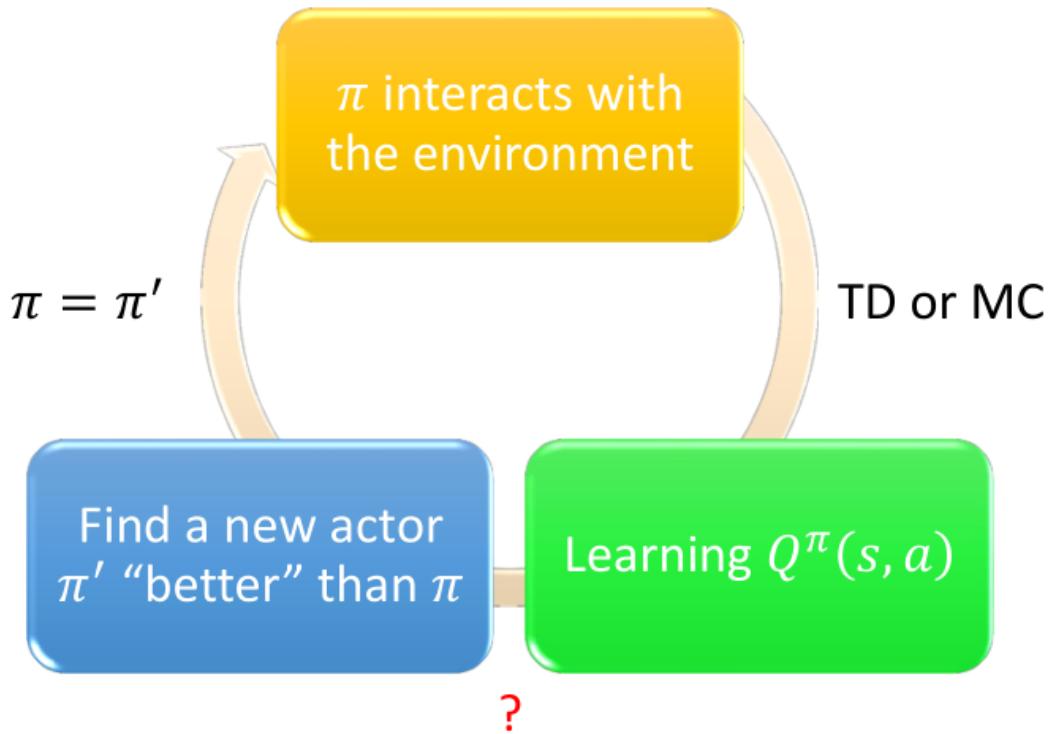
<https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassabis15NatureControlDeepRL.pdf>

Another Way to use Critic: Q-Learning

接下来要讲的是说，虽然表面上我们 learn 一个 Q function，它只能拿来评估某一个 actor π 的好坏，但是实际上只要有了这个 Q function，我们就可以做 reinforcement learning。其实有这个 Q function，我们就可以决定要采取哪一个 action。

它的大原则是这样，假设你有一个初始的 actor，也许一开始很烂，随机的也没有关系，初始的 actor 叫做 π ，那这个 π 跟环境互动，会 collect data。

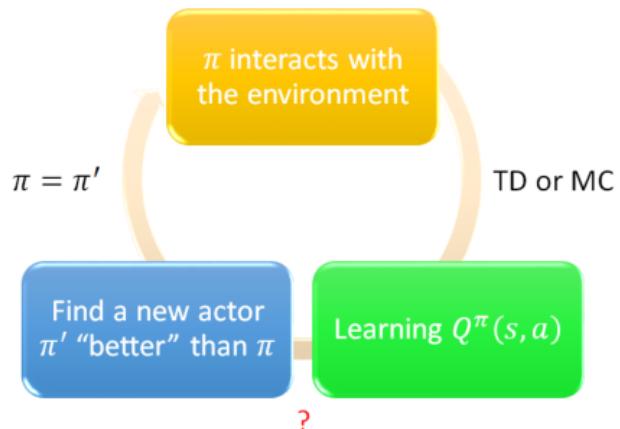
接下来你去衡量一下 π 这个 actor，它在某一个 state 强制采取某一个 action，接下来用 π 这个 policy 会得到的 expected reward，那你可以用 TD 也可以用 MC 都是可以的。



你 learn 出一个 Q function 以后，一个神奇的地方就是，只要 learn 得出某一个 policy π 的 Q function，就保证你可以找到一个新的 policy，这个 policy 就做 π' ，这一个 policy π' ，它一定会比原来的 policy π 还要好。

那等一下会定义什么叫做好，所以这边神奇的地方是，假设你只要有一个 Q function，你有某一个 policy π ，你根据那个 policy π learn 出 policy π 的 Q function，接下来保证你可以找到一个新的 policy 叫做 π' ，它一定会比 π 还要好，你今天找到一个新的 π' ，一定会比 π 还要好以后，你把原来的 π 用 π' 取代掉，再去找它的 Q。得到新的 Q 以后，再去找一个更好的 policy，然后这个循环一直下去，你的 policy 就会越来越好。

Q-Learning



- Given $Q^\pi(s, a)$, find a new actor π' "better" than π
 - "Better": $V^{\pi'}(s) \geq V^\pi(s)$, for all state s

$$\pi'(s) = \arg \max_a Q^\pi(s, a)$$

- π' does not have extra parameters. It depends on Q
- Not suitable for continuous action a (solve it later)

首先要定义的是，什么叫做比较好？我们说 π' 一定会比 π 还要好，什么叫做好呢？这边所谓好的意思是说，对所有可能的 state s 而言，对同一个 state s ， π 的 value function 一定会小于 π' 的 value function，也就是说我们走到同一个 state s 的时候，如果拿 π 继续跟环境互动下去，我们得到的 reward 一定会小于用 π' 跟环境互动下去得到的 reward，所以今天不管在哪一个 state，你用 π' 去做 interaction，你得到的 expected reward 一定会比较大，所以 π' 是比 π 还要好的一个 policy。

那有了这个 Q 以后，怎么找这个 π' 呢？这边的构想非常的简单，事实上这个 π' 是什么？如果根据以下的这个式子去决定你的 action 的步骤叫做 π' 的话，那这个 π' 一定会比 π 还要好。

这个意思是说，假设你已经 learn 出 π 的 Q function，今天在某一个 state s ，你把所有可能的 action a ，都——带入这个 Q function，看看说哪一个 a ，可以让 Q function 的 value 最大，那这一个 action，就是 π' 会采取的 action。

那这边要注意一下，我们刚才有讲过 Q function 的定义，given 这个 state s ，你的 policy π ，并不一定会采取 action a 。今天是 given 某一个 state s ，强制采取 action a ，用 π 继续互动下去，得到的 expected reward，才是这个 Q function 的定义。所以我们强调，在 state s 里面，不一定会采取 action a 。

今天假设我们用这一个 π' ，它在 state s 采取 action a ，跟 π 所谓采取 action，是不一定会一样的，然后 π' 所采取的 action，会让它得到比较大的 reward。

所以实际上，根本就没有所谓一个 policy 叫做 π' ，这个 π' 其实就是用 Q function 推出来的，所以并没有另外一个 network 决定 π' 怎么 interaction。我们只要 Q 就好，有 Q 就可以找出 π' 。

但是这边有另外一个问题是等一下会解决的就是，在这边要解一个 Arg Max 的 problem，所以 a 如果是 continuous 的就会有问题，如果是 discrete 的， a 只有 3 个选项，一个一个带进去，看谁的 Q 最大，没有问题。但如果是 continuous 要解 Arg Max problem，你就会有问题，但这个是之后才会解决的。

Proof

为什么用 Q function，所决定出来的 π' ，一定会比 π 还要好？

假设我们有一个 policy $\pi'(s) = \arg \max_a Q^\pi(s, a)$ ，它是由 Q^π 决定的。我们要证：对所有的 state s 而言， $V^{\pi'}(s) \geq V^\pi(s)$ 。

假设你在 state s 这个地方，你 follow π 这个 actor，它会采取的 action 也就是 $\pi(s)$ ，那 $V^\pi(s) = Q^\pi(s, \pi(s))$ 。In general 而言， Q^π 不见得等于 V^π ，是因为 action 不见得是 $\pi(s)$ ，但这个 action 如果是 $\pi(s)$ 的话， Q^π 是等于 V^π 的。

因为这边是某一个 action，这边是所有 action 里面可以让 Q 最大的那个 action，所以 $Q^\pi(s, \pi(s)) \leq \max_a Q^\pi(s, a)$ 。

a 就是 $\pi'(s)$ ，所以今天这个式子可以写成 $Q^\pi(s, \pi(s)) \leq \max_a Q^\pi(s, a) = Q^\pi(s, \pi'(s))$

因此我们知道 $V^\pi(s) \leq Q^\pi(s, \pi'(s))$ ，也就是说你在某一个 state，如果你按照 policy π ，一直做下去，你得到的 reward 一定会小于等于你在现在这个 state s ，你故意不按照 π 所给你指示的方向，你故意按照 π' 的方向走一步。

但之后，只有第一步是按照 π' 的方向走，只有在 state s 这个地方，你才按照 π' 的指示走，但接下来你就按照 π 的指示走。

虽然只有一步之差，但是我们可以按照上面这个式子知道说，这个时候你得到的 reward，只有一步之差，你得到的 reward 一定会比完全 follow π ，得到的 reward 还要大。

那接下来，eventually，想要证的东西就是，这一个 $Q^\pi(s, \pi'(s))$ ，会小于等于 $V^{\pi'}(s)$ ，也就是说，只有一步之差，你会得到比较大的 reward，但假设每步都是不一样的，每步通通都是 follow π' 而不是 π 的话，那你得到的 reward 一定会更大。直觉上想起来是这样子的。

如果你要用数学式把它写出来的话，略嫌麻烦，但也没有很难，只是比较繁琐而已。

你可以这样写 $Q^\pi(s, \pi'(s)) = E[r_{t+1} + V^\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)]$ ，它的意思就是说，我们在 state s_t ，我们会采取 action a_t ，接下来我们会得到 reward $r(t+1)$ ，然后跳到 state $s(t+1)$ 。

这边写得不太好，这边应该写成 r_t ，跟之前的 notation 比较一致，但这边写成了 $r(t+1)$ ，其实这都是可以的，在文献上有时候有人会说，在 state s_t 采取 action a_t 得到 reward $r(t+1)$ ，有人会写成 r_t ，但意思其实都是一样的。

$V^\pi(s_{t+1})$ 是 state $s(t+1)$ ，根据 π 这个 actor 所估出来的 value，上面这个式子，等于下面这个式子。

要取一个期望值，因为在同样的 state 采取同样的 action，你得到的 reward 还有会跳到 state 不见得是一样，所以这边需要取一个期望值。

因为 $V^\pi(s) \leq Q^\pi(s, \pi'(s))$ ，带入，可以得到

$$\begin{aligned} & E[r_{t+1} + V^\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)] \\ & \leq E[r_{t+1} + Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \end{aligned}$$

也就是说，现在你一直 follow π ，跟某一步 follow π' ，接下来都 follow π ，比起来，某一步 follow π' 得到的 reward 是比较大的。

就可以写成下面这个式子，因为 Q^π 这个东西可以写成 $r(t+2) + s(t+2)$ 的 value。

$$\begin{aligned} & E[r_{t+1} + Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \\ & = E[r_{t+1} + r_{t+2} + V^\pi(s_{t+2}) | \dots] \\ & \leq E[r_{t+1} + r_{t+2} + Q^\pi(s_{t+2}, \pi'(s_{t+2})) | \dots] \dots \leq V^{\pi'}(s) \end{aligned}$$

你再把 $V^\pi(s) \leq Q^\pi(s, \pi'(s))$ 带进去，然后一直算，算到 episode 结束，那你就知道 $V^\pi(s) \leq V^{\pi'}(s)$ 。

假设你没有办法 follow 的话，总是想要告诉你的事情是说，你可以 estimate 某一个 policy 的 Q function，接下来你就一定可以找到另外一个 policy 叫做 π' ，它一定比原来的 policy 还要更好。

$$\begin{aligned} \pi'(s) &= \arg \max_a Q^\pi(s, a) \\ & V^{\pi'}(s) \geq V^\pi(s), \text{ for all state } s \end{aligned}$$

$$\begin{aligned} V^\pi(s) &= Q^\pi(s, \pi(s)) \\ & \leq \max_a Q^\pi(s, a) = Q^\pi(s, \pi'(s)) \end{aligned}$$

$$\begin{aligned} V^\pi(s) &\leq Q^\pi(s, \pi'(s)) \\ &= E[r_{t+1} + V^\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)] \\ &\leq E[r_{t+1} + Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \\ &= E[r_{t+1} + r_{t+2} + V^\pi(s_{t+2}) | \dots] \\ &\leq E[r_{t+1} + r_{t+2} + Q^\pi(s_{t+2}, \pi'(s_{t+2})) | \dots] \dots \leq V^{\pi'}(s) \end{aligned}$$

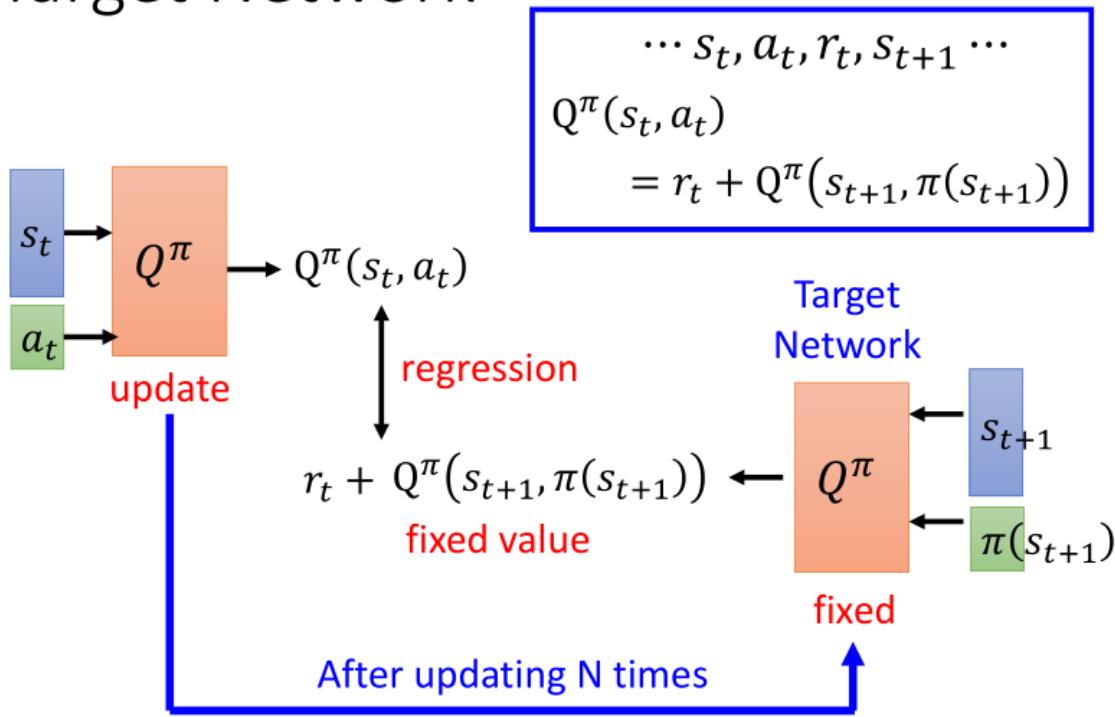
Target Network

我们讲一下接下来在 Q learning 里面, typically 你一定会用到的 tip。

第一个, 你会用一个东西叫做 target network, 什么意思呢?

我们在 learn Q function 的时候, 你也会用到 TD 的概念, 那怎么用 TD 的概念呢?

Target Network



就是说你现在收集到一个 data, 是说在 state s_t , 你采取 action a_t 以后, 你得到 reward r_t , 然后跳到 state $s(t+1)$ 。

然后今天根据这个 Q function 你会知道说, $Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$, 它们中间差了一项就是 r_t , 所以你在 learn 的时候, 你会说我们有 Q function, input s_t, a_t 得到的 value, 跟 input $s(t+1), \pi(s_{t+1})$ 得到的 value 中间, 我们希望它差了一个 r_t , 这跟 TD 的概念是一样的。

但是实际上在 learn 的时候, 这样 in general 而言这样的一个 function 并不好 learn。因为假设你说这是一个 regression 的 problem, $Q^\pi(s_t, a_t)$ 是你 network 的 output, $r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 是你的 target, 你会发现你的 target 是会动的。当然你要 implement 这样的 training 其实也没有问题。

就是你在做 back propagation 的时候, 这两个 model 的参数都要被 update。它们是同一个 model, 所以两个 update 的结果会加在一起。

但是实际上在做的时候, 你的 training 会变得不太稳定, 因为假设你把这个当作你 model 的 output, 这个当作 target 的话, 你会变成说你要去 fit 的 target, 它是一直在变的。这种一直在变的 target 的 training 其实是不太好 train 的。

所以实际上怎么做呢? 实际上你会把其中一个 Q, 通常是选择下面这个 Q, 把它固定住, 也就是说你在 training 的时候, 你并不 update 这个 Q 的参数, 你只 update 左边这个 Q 的参数, 而右边这个 Q 的参数, 它会被固定住, 我们叫它 target network。它负责产生 target, 所以叫做 target network。因为 target network 是固定的, 所以你现在得到的 target, 也就是 $r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 的值也会是固定的。

那我们只调左边这个 network 的参数，那假设因为 target network 是固定的，我们只调左边 network 的参数，它就变成是一个 regression 的 problem。我们希望我们 model 的 output，它的值跟你的目标越接近越好，你会 minimize 它的 mean square error，那你会 minimize 它们 L2 的 distance，那这个东西就是 regression。

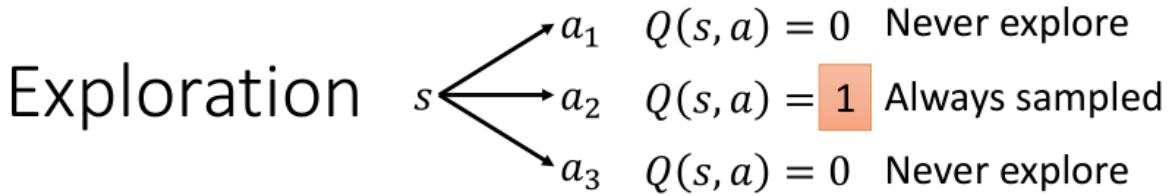
在实现上呢，你会把这个 Q update 好几次以后，再去把这个 target network 用 update 过的 Q，去把它替换掉。你在 train 的时候，先 update 它好几次，然后再把它替换掉，但它们两个不要一起动，它们两个一起动的话，你的结果会很容易坏掉。

一开始这两个 network 是一样的，然后接下来在 train 的时候，你在做 gradient decent 的时候，只调左边这个 network 的参数，那你可能 update 100 次以后，才把这个参数，复制到右边去，把它盖过去。把它盖过去以后，你这个 target 的 value，就变了，就好像说你今天本来在做一个 regression 的 problem，那你 train 把这个 regression problem 的 loss 压下去以后，接下来你把这边的参数把它 copy 过去以后，你的 target 就变掉了，你 output 的 target 就变掉了，那你接下来就要重新再 train。

loss会不会变成 0？因为首先它们的 input 是不一样，同样的 function，这边的 input 是 s_t 跟 a_t ，这边 input 是 s_{t+1} 跟 action $\pi(s_{t+1})$ ，因为 input 不一样，所以它 output 的值会不一样，今天再加上 r_t ，所以它们的值就会更不一样，但是你希望说你会把这两项的值把它拉近。

Exploration

第二个会用到的 tip 是 Exploration，我们刚才讲说，当我们使用 Q function 的时候，我们的 policy 完全 depend on 那个 Q function，看说 given 某一个 state，你就穷举所有的 a ，看哪个 a 可以让 Q value 最大，它就是你采取的 policy，它就是采取的 action。



- The policy is based on Q-function

$$a = \arg \max_a Q(s, a)$$

This is not a good way
for data collection.

Epsilon Greedy ε would decay during learning

$$a = \begin{cases} \arg \max_a Q(s, a), & \text{with probability } 1 - \varepsilon \\ \text{random}, & \text{otherwise} \end{cases}$$

Boltzmann Exploration

$$P(a|s) = \frac{\exp(Q(s, a))}{\sum_a \exp(Q(s, a))}$$

那其实这个跟 policy gradient 不一样，在做 policy gradient 的时候，我们的 output 其实是 stochastic 的，我们 output 一个 action 的 distribution，根据这个 action 的 distribution 去做 sample，所以在 policy gradient 里面，你每次采取的 action 是不一样的，是有随机性的。

那像这种 Q function，如果你采取的 action 总是固定的，会有什么问题呢？你会遇到的问题就是，这不是一个好的收集 data 的方式，为什么这不是一个好的收集 data 的方式呢？因为假设我们今天你要估测在某一个 state 采取某一个 action 会得到的 Q value，你一定要在那一个 state，采取过那个 action，你才估得出它的 value。

如果你没有在那个 state 采取过那个 action，你其实估不出那个 value 的。当然如果是用 deep 的 network，就你的 Q function 其实是一个 network。这种情形可能会比较没有那么严重，但是 in general 而言，假设你 Q function 是一个 table，没有看过的 state-action pair 就是估不出值来，当然 network 也是会有一样的问题，只是没有那么严重，但也会有一样的问题。

所以今天假设你在某一个 state，action a1, a2, a3 你都没有采取过，那你估出来的 $(s, a1)$ $(s, a2)$ $(s, a3)$ 的 Q value，可能就都是一样的，就都是一个初始值，比如说 0。

但是今天假设你在 state s，你 sample 过某一个 action a2 了，那 sample 到某一个 action a2，它得到的值是 positive 的 reward，那现在 $Q(s, a2)$ ，就会比其它的 action 都要好。那我们说今天在采取 action 的时候，就看谁的 Q value 最大，就采取谁。所以之后你永远都只会 sample 到 a2，其它的 action 就再也不会被做了，所以今天就会有问题。

就好像说你进去一个餐厅吃饭，餐厅都有一个菜单，那其实你都很难选，你今天点了某一个东西以后，假说点了某一样东西，比如说椒麻鸡，你觉得还可以，接下来你每次去，就都会点椒麻鸡，再也不会点别的东西了，那你就知道说别的东西是不是会比椒麻鸡好吃，这个是一样的问题。

那如果你今天没有好的 exploration 的话，你在 training 的时候就会遇到这种问题。

举一个实际的例子，假设你今天是用 Q learning 来玩比如说 slither.io，在玩 slither.io 你会有一个蛇，然后它在环境里面就走来走去，然后就吃到星星，它就加分。

那今天假设这个游戏一开始，它采取往上走，然后就吃到那个星星，它就得到分数，它就知道说往上走是 positive，接下来它就再也不会采取往上走以外的 action 了。所以接下来就会变成每次游戏一开始，它就往上冲，然后就死掉，再也做不了别的事。

所以今天需要有 exploration 的机制，需要让 machine 知道说，虽然 a2，根据之前 sample 的结果，好像是不错的，但你至少偶尔也试一下 a1 跟 a3，搞不好它们更好也说不定。

有两个方法解这个问题，一个是 Epsilon Greedy，Epsilon Greedy 的意思是说，我们有，1-epsilon 的机率，通常 epsilon 就设一个很小的值，1-epsilon 可能是 90%，也就是 90% 的机率完全按照 Q function 来决定 action，但是你有 10% 的机率是随机的。

通常在实作上 epsilon 会随着时间递减。

也就是在最开始的时候，因为还不知道那个 action 是比较好的，所以你会花比较大的力气在做 exploration。

那接下来随着 training 的次数越来越多，已经比较确定说哪一个 Q 是比较好的，你就会减少你的 exploration，你会把 epsilon 的值变小，主要根据 Q function 来决定你的 action，比较少做 random，这是 Epsilon Greedy。

那还有另外一个方法叫做 Boltzmann Exploration，这个方法就比较像是 policy gradient，在 policy gradient 里面我们说 network 的 output 是一个 probability distribution，再根据 probability distribution 去做 sample。

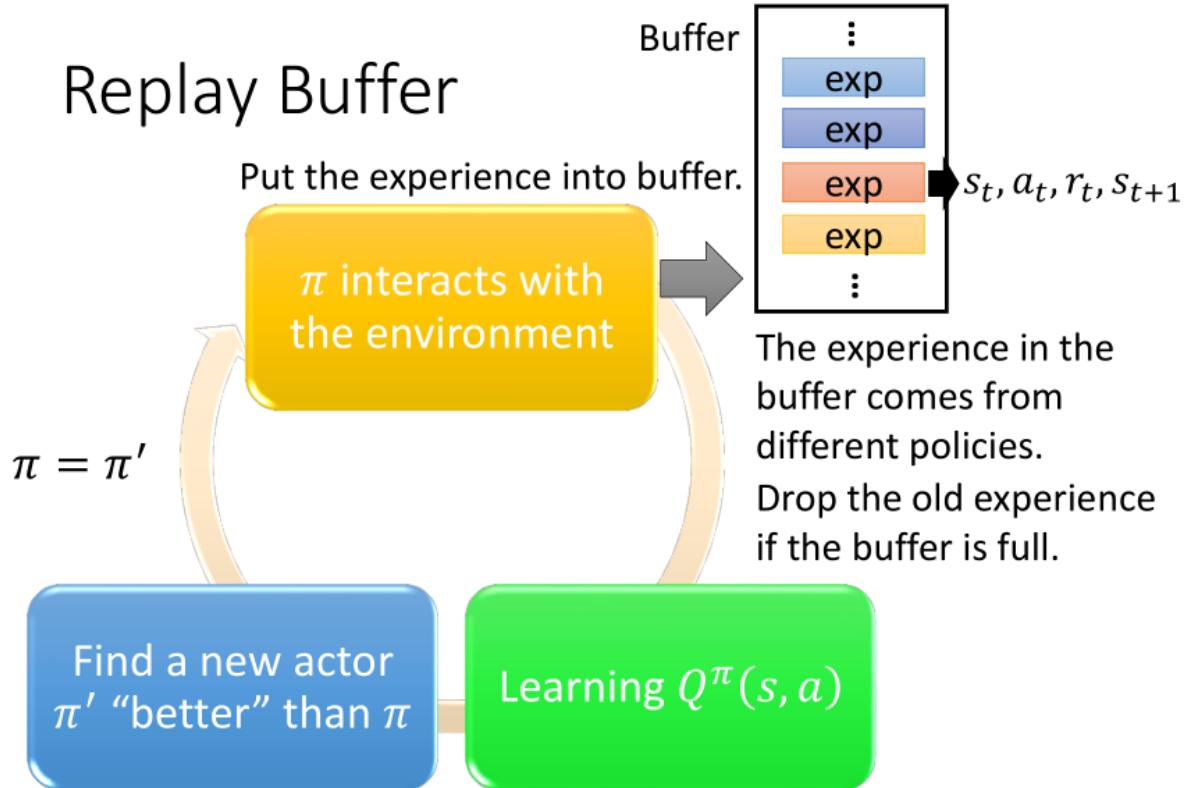
那其实你也可以根据 Q value 去定一个 probability distribution，你可以说，假设某一个 action，它的 Q value 越大，代表它越好，那我们采取这个 action 的机率就越高，但是某一个 action 它的 Q value 小，不代表我们不能 try try 看它好不好用，所以我们有时候也要 try try 那些 Q value 比较差的 action。

那怎么做呢？因为 Q value 它是有正有负的，所以你要把它弄成一个机率，你可能就先取 exponential，然后再做 normalize，然后把 $Q(s, a)$ exponential，再做 normalize 以后的这个机率，就当作是你在决定 action 的时候 sample 的机率。

其实在实作上，你那个 Q 是一个 network，所以你有点难知道说，今天在一开始的时候 network 的 output，到底会长怎么样子。但是其实你可以猜测说，假设你一开始没有任何的 training data，你的参数是随机的，那 given 某一个 state s ，你的不同的 a output 的值，可能就是差不多的。所以一开始 $Q(s, a)$ 应该会倾向于 uniform，也就是在一开始的时候，你这个 probability distribution 算出来，它可能是比较 uniform 的。

Replay Buffer

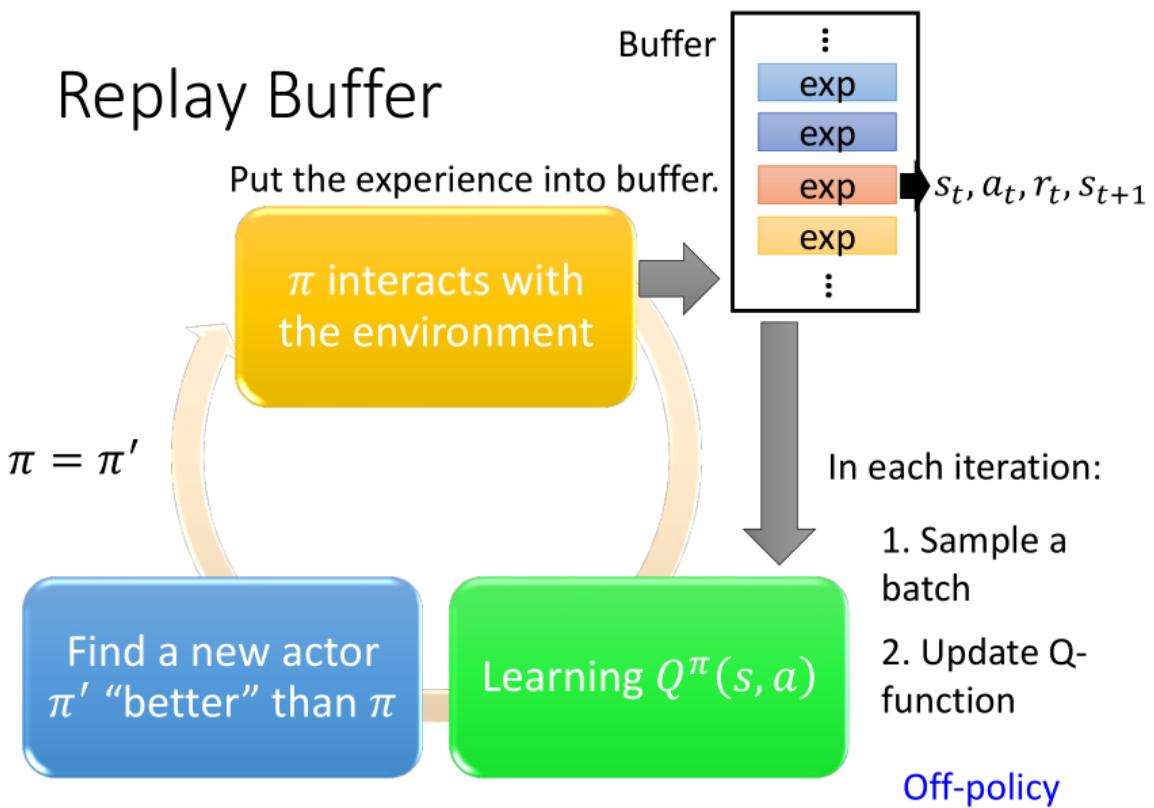
那还有第三个你会用的 tip，这个 tip 叫做 replay buffer。



replay buffer 的意思是说，现在我们会有某一个 policy π 去跟环境做互动，然后它会去收集 data，我们会把所有的 data 放到一个 buffer 里面，那 buffer 里面就排了很多 data，那你 buffer 设比如说 5 万，这样它里面可以存 5 万笔数据，每一笔数据是什么？每一笔数据就是记得说，我们之前在某一个 state s_t ，采取某一个 action a_t ，接下来我们得到的 reward r_t ，然后接下来跳到 state s_{t+1} ，某一笔数据，就是这样。那你用 π 去跟环境互动很多次，把所有收集到的数据通通都放到这个 replay buffer 里面。

这边要注意的事情是，这个 replay buffer 它里面的 experience，可能是来自于不同的 policy，就你每次拿 π 去跟环境互动的时候，你可能只互动 10,000 次，然后接下来你就更新你的 π 了。但是你的这个 buffer 里面可以放 5 万笔数据。所以那 5 万笔数据，它们可能是来自于不同的 policy。那这个 buffer 只有在它装满的时候，才会把旧的资料丢掉，所以这个 buffer 里面它其实装了很多不同的 policy，所计算出来的不同的 policy 的 experiences。

Replay Buffer



接下来你有了这个 buffer 以后，你做的事情，你是怎么 train 这个 Q 的 model 呢？你是怎么估 Q 的 function？

你的做法是这样，你会 iterative 去 train 这个 Q function，在每一个 iteration 里面，你从这个 buffer 里面，随机挑一个 batch 出来。

就跟一般的 network training 一样，你从那个 training data set 里面，去挑一个 batch 出来，你去 sample 一个 batch 出来，里面有一把的 experiences。根据这把 experiences 去 update 你的 Q function。就跟我们刚才讲那个 TD learning 要有一个 target network 是一样的。你去 sample 一个 batch 的 data，sample 一堆 experiences，然后再去 update 你的 Q function。

这边其实有一个东西你可以稍微想一下，你会发现说，实际上当我们这么做的时候，它变成了一个 off policy 的做法。

因为本来我们的 Q 是要观察， π 这个 action 它的 value，但实际上存在你的 replay buffer 里面的这些 experiences，不是通通来自于 π 。有些是过去其它的 π ，所遗留下来的 experience。

因为你不会拿某一个 π 就把整个 buffer 装满，然后拿去测 Q function，这个 π 只是 sample 一些 data，塞到那个 buffer 里面去，然后接下来就让 Q 去 train，所以 Q 在 sample 的时候，它会 sample 到过去的一些数据，但是这么做到底有什么好处呢？

这么做有两个好处，第一个好处，其实在做 reinforcement learning 的时候，往往最花时间的 step，是在跟环境做互动，train network 反而是比较快的，因为你用 GPU train 其实很快，真正花时间的往往是在跟环境做互动。

今天用 replay buffer，你可以减少跟环境做互动的次数，因为今天你在做 training 的时候，你的 experience 不需要通通来自于某一个 policy，一些过去的 policy 它所得到的 experience，可以放在 buffer 里面被使用很多次，被反复的再利用。这样让你的 sample 到 experience 的利用是比较 efficient。

还有另外一个理由是，你记不记得我们说在 train network 的时候，其实我们希望一个 batch 里面的 data，越 diverse 越好。

如果你的 batch 里面的 data 通通都是同样性质的，你 train 下去，其实是容易坏掉的。不知道大家有没有这样子的经验，如果你 batch 里面都是一样的 data，你 train 的时候，performance 会比较差。我们希望 batch data 越 diverse 越好。

那如果你的这个 buffer 里面的那些 experience，它通通来自于不同的 policy 的话，那你得到的结果，你 sample 到的一个 batch 里面的 data，会是比较 diverse 的。

但是接下来你会问的一个问题是，我们明明确是要观察 π 的 value，我们要量的明明是 π 的 value 啊，里面混杂了一些不是 π 的 experience，到底有没有关系？

这一件事情其实是没有关系的，这并不是因为过去的 π 跟现在的 π 很像，就算过去的 π 没有很像，其实也是没有关系的。这个留给大家回去想一下，为什么会这个样子。主要的原因是，我们并不是去 sample 一个 trajectory，我们只 sample 了一笔 experience，所以跟我们是不是 off policy 这件事是没有关系的。就算是 off-policy，就算是这些 experience 不是来自于 π ，我们其实还是可以拿这些 experience 来估测 $Q^\pi(s, a)$ 。这件事有点难解释，不过你就记得说，replay buffer 这招其实是在理论上也是没有问题的。

Typical Q-Learning Algorithm

这个就是 typical 的一般的正常的 Q learning 演算法。

Initialize Q-function Q , target Q-function $\hat{Q} = Q$

In each episode

- For each time step t
 - Given state s_t , take action a_t based on Q (epsilon greedy)
 - Obtain reward r_t , and reach new state s_{t+1}
 - Store (s_t, a_t, r_t, s_{t+1}) into buffer
 - Sample (s_i, a_i, r_i, s_{i+1}) from buffer (usually a batch)
 - Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
 - Update the parameters of Q to make $Q(s_i, a_i)$ close to y (regression)
 - Every C steps reset $\hat{Q} = Q$

我们说我们需要一个 target network，先开始 initialize 的时候，你 initialize 2 个 network，一个是 Q ，一个是 Q hat，那其实 Q hat 就等于 Q ，一开始这个 target Q-network，跟你原来的 Q network 是一样的。

那在每一个 episode，你拿你的 agent，你拿你的 actor 去跟环境做互动，那在每一次互动的过程中，你都会得到一个 state s_t ，一个游戏的画面，那你会采取某一个 action a_t 。那怎么知道采取那一个 action a_t 呢？你就根据你现在的 Q-function，但是记得你要有 exploration 的机制，比如说你用 Boltzmann exploration 或是 Epsilon Greedy 的 exploration。

那接下来你得到 reward r_t ，然后跳到 state $s(t+1)$ ，所以现在 collect 到一笔 data，这笔 data 是 s_t, a_t, r_t, s_{t+1} 。把这笔 data 就塞到你的 buffer 里面去，那如果 buffer 满的话，你就再把一些旧的数据再把它丢掉。

那接下来你就从你的 buffer 里面去 sample data, 那你 sample 到的是 s_i, a_i, r_i, s_{i+1} 这笔 data 跟你刚放进去的, 不见得是同一笔, 抽到一个旧的也是有可能的。

那这边另外要注意的是, 其实你 sample 出来不是一笔 data, 你 sample 出来的是一个 batch 的 data, sample 一把 experiences 出来。

你 sample 这一把 experience 以后, 接下来你要做的事情就是, 计算你的 target, 根据你 sample 出这么一笔 data 去算你的 target, 你的 target 是什么呢? target 记得要用 target network, 也就是 $Q \hat{}$ 来算, 我们用 $Q \hat{}$ 来代表 target network。

target 是多少呢? target 就是 r_i 加上, $Q \hat{}$ of (s_{i+1}, a) 。现在哪一个 a , 可以让 $Q \hat{}$ 的值最大, 你就选那个 a 。因为我们在 state s_{i+1} , 会采取的 action a , 其实就是那个可以让 Q value 的值最大的那个 a 。

接下来我们要 update Q 的值, 那就把它当作一个 regression 的 problem, 希望 Q of (s_i, a_i) 跟 target 越接近越好, 然后今天假设这个 update, 已经 update 了某一个数目的次, 比如说 c 次, 你就设一个 $c = 100$, 那你就把 $Q \hat{}$ 设成 Q , 就这样。

Tips of Q-Learning

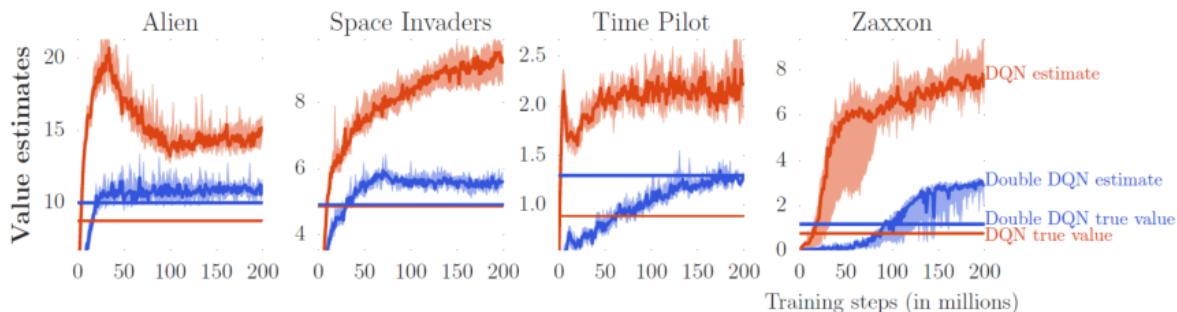
Double DQN

接下来我们要讲的是 train Q learning 的一些 tip。

第一个要介绍的 tip, 叫做 double DQN。那为什么要 double DQN 呢? 因为在实作上, 你会发现说, Q value 往往是被高估的。

那下面这几张图是来自于 double DQN 的原始 paper, 它想要显示的结果就是, Q value 往往是被高估的。

- Q value is usually over-estimated



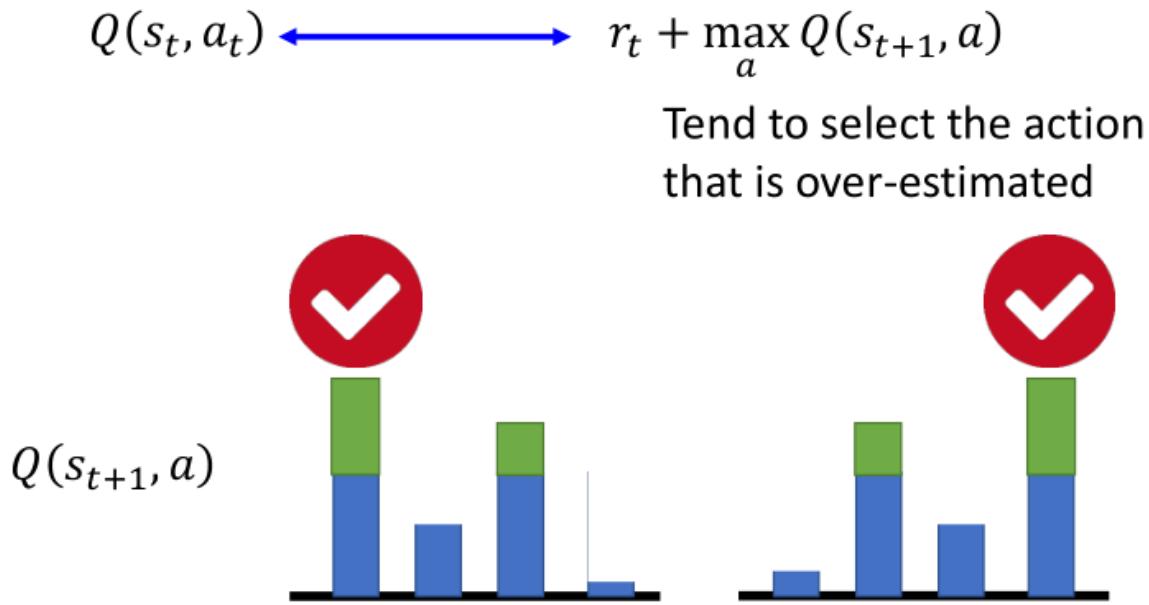
这边就是有 4 个不同的小游戏, 那横轴是 training 的时间, 然后红色这个锯齿状一直在变的线就是, 对不同的 state estimate 出来的平均 Q value, 就有很多不同的 state, 每个 state 你都 sample 一下, 然后算它们的 Q value, 把它们平均起来, 这是红色这一条线, 它在 training 的过程中会改变, 但它是不断上升的。为什么它不断上升, 很直觉, 不要忘了 Q function 是 depend on 你的 policy 的, 你今天在 learn 的过程中你的 policy 越来越强, 所以你得到 Q 的 value 会越来越大。在同一个 state, 你得到 expected reward 会越来越大, 所以 general 而言, 这个值都是上升的。

但是它说, 这是 Q network 估测出来的值, 接下来你真的去算它。怎么真的去算? 你有那个 policy, 然后真的去玩那个游戏, 你就可以估说, 你就可以真的去算说, 就玩很多次, 玩个 1 百万次, 然后就去真的估说, 在某一个 state, 你会得到的 Q value, 到底有多少。你会得到说在某一个 state, 采取某一个 action, 你接下来会得到 accumulated reward 的总和是多少。那你会发现说, 估测出来的值是远比实际的值大, 在每一个游戏都是这样, 都大很多。

double DQN 可以让估测的值跟实际的值是比较接近的。蓝色的锯齿状的线是 double DQN 的 Q network 所估测出来的 Q value，蓝色的是真正的 Q value，你会发现他们是比较接近的。

还有另外一个有趣可以观察的点就是说，用 double DQN 得出来真正的 accumulated reward，在这 3 个 case，都是比原来的 DQN 高的，代表 double DQN learn 出来那个 policy 比较强，所以它实际上得到的 reward 是比较大的。虽然说看那个 Q network 的话，一般的 DQN 的 Q network 虚张声势，高估了自己会得到的 reward，但实际上它得到的 reward 是比较低的。

Q value is usually over estimate



那接下来要讲的第一个问题就是，为什么 Q value 总是被高估了呢？这个是有道理的，因为我们实际上在做的时候，我们是要让左边这个式子，跟右边我们这个 target，越接近越好，那你会发现说，target 的值，很容易一不小心就被设得太高。

为什么 target 的值很容易一不小心就被设得太高呢？因为你在算这个 target 的时候，我们实际上在做的事情是说，看哪一个 a 它可以得到最大的 Q value，就把它加上去，就变成我们的 target。所以今天假设有某一个 action，它得到的值是被高估的。

举例来说，我们现在有 4 个 actions，那本来其实它们得到的值都是差不多的，他们得到的 reward 都是差不多的，但是在 estimate 的时候，那毕竟是个 network，所以 estimate 的时候是有误差的。

所以假设今天是第一个 action，它被高估了，假设绿色的东西代表是被高估的量，它被高估了，那这个 target 就会选这个高估的 Q value，来加上 r_t ，来当作你的 target。所以你总是会选那个 Q value 被高估的，你总是会选那个 reward 被高估的 action 当作这个 max 的结果，去加上 r_t 当作你的 target，所以你的 target 总是太大。

- Q value is usually over estimate

$$Q(s_t, a_t) \longleftrightarrow r_t + \max_a Q(s_{t+1}, a)$$

- Double DQN: two functions Q and Q' Target Network

$$Q(s_t, a_t) \longleftrightarrow r_t + Q'\left(s_{t+1}, \arg \max_a Q(s_{t+1}, a)\right)$$

If Q over-estimate a, so it is selected. Q' would give it proper value.

How about Q' overestimate? The action will not be selected by Q.

Hado V. Hasselt, "Double Q-learning", NIPS 2010

Hado van Hasselt, Arthur Guez, David Silver, "Deep Reinforcement Learning with Double Q-learning", AAAI 2016

那怎么解决这 target 总是太大的问题呢? 那 double DQN 它的设计是这个样子的。在 double DQN 里面, 选 action 的 Q function, 跟算 value 的 Q function, 不是同一个。

今天在原来的 DQN 里面, 你穷举所有的 a, 把每一个 a 都带进去, 看哪一个 a 可以给你的 Q value 最高, 那你就把那个 Q value 加上 r_t 。

但是在 double DQN 里面, 你有两个 Q network, 第一个 Q network 决定那一个 action 的 Q value 最大, 你用第一个 Q network 去带入所有的 a, 去看看哪一个 Q value 最大, 然后你决定你的 action 以后, 实际上你的 Q value 是用 Q' 所算出来的。这样子有什么好处呢? 为什么这样就可以避免 over estimate 的问题呢?

因为今天假设我们有两个 Q function, 假设第一个 Q function 它高估了它现在选出来的 action a, 那没关系, 只要第二个 Q function Q' , 它没有高估这个 action a 的值, 那你算出来的, 就还是正常的值。那今天假设反过来是 Q' 高估了某一个 action 的值, 那也没差, 因为反正只要前面这个 Q 不要选那个 action 出来, 就没事了。这个就跟行政跟立法是分立的概念是一样的。Q 负责提案, 它负责选 a, Q' 负责执行, 它负责算出 Q value 的值。所以今天就算是前面这个 Q, 做了不好的提案, 它选的 a 是被高估的, 只要后面 Q' 不要高估这个值就好了, 那就算 Q' 会高估某个 a 的值, 只要前面这个 Q 不提案那个 a, 算出来的值就不会被高估了, 所以这个就是 double DQN 神奇的地方。

然后你可能会说, 哪来两个 Q 跟 Q' 呢? 哪来两个 network 呢? 其实在实作上, 你确实是有两个 Q value 的, 因为一个就是你真正在 update 的 Q, 另外一个就是 target 的 Q network。就是你其实有两个 Q network, 一个是 target 的 Q network, 一个是你真正在 update 的 Q network。所以在 double DQN 里面, 你的实作方法会是, 你拿真正的 Q network, 你会 update 参数的那个 Q network, 去选 action, 然后你拿 target 的 network, 那个固定住不动的 network, 去算 value。那 double DQN 相较于原来的 DQN 的改动是最少的, 它几乎没有增加任何的运算量, 看连新的 network 都不用, 因为你原来就有两个 network 了。

你唯一要做的事情只有, 本来你在找最大的 a 的时候, 你在决定这个 a 要放哪一个的时候, 你是用 Q' 来算, 你是用 freeze 的那个 network 来算, 你是用 target network 来算, 现在改成用另外一个是 update 的 Q network 来算, 这个应该是改一行 code 就可以解决了, 所以这个就是轻易的就可以 implement。

Dueling DQN

那第二个 tip，叫做 dueling 的 DQN。dueling DQN 是什么呢？

其实 dueling DQN 也蛮好做的，相较于原来的 DQN，它唯一的差别是改了 network 的架构。等一下你听了如果觉得，有点没有办法跟上的话，你就要记住一件事，dueling DQN 它唯一做的事情，是改 network 的架构。

我们说 Q network 就是 input state，output 就是每一个 action 的 Q value。dueling DQN 唯一做的事情，是改了 network 的架构，其它的演算法，你都不要去动它。

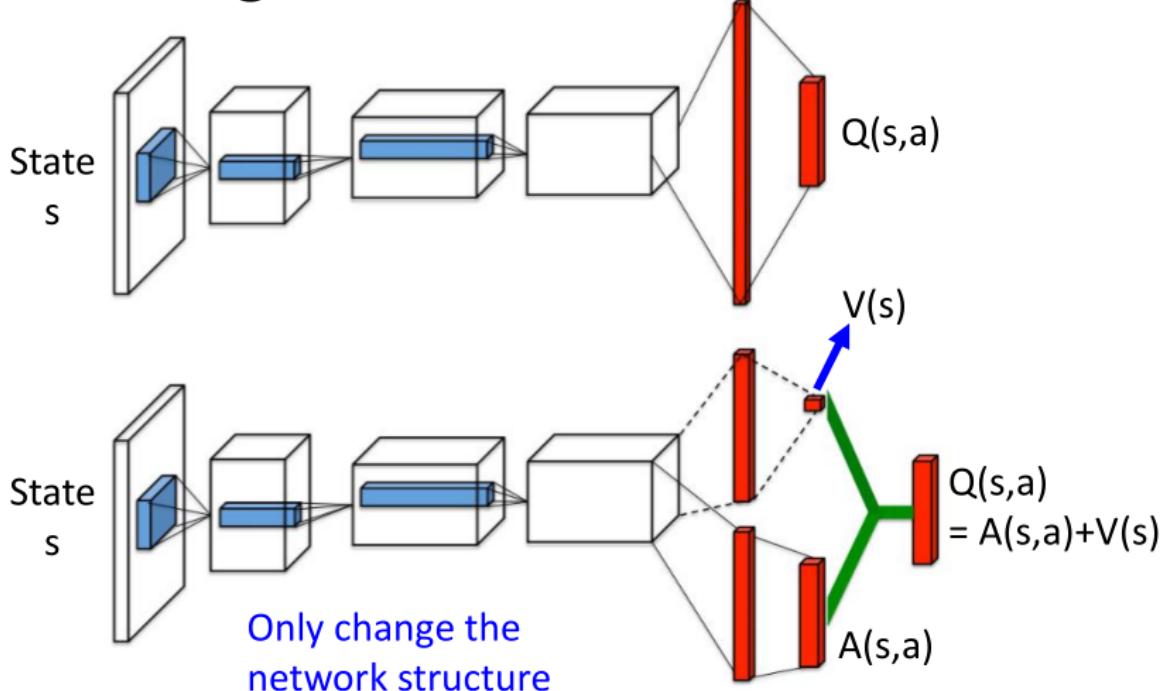
那 dueling DQN 它是怎么改了 network 的架构呢？它是这样说的，本来的 DQN 就是直接 output Q value 的值，现在这个 dueling 的 DQN 就是下面这个 network 的架构，它不直接 output Q value 的值，它是怎么做的？

它在做的时候，它分成两条 path 去运算，第一个 path，它算出一个 scalar，那这个 scalar 因为它跟 input s 是有关系，所以叫做 $V(s)$ 。

那下面这个，它会 output 另外一个 vector 叫做 $A(s, a)$ ，它的 dimension 与 action 的数目相同。那下面这个 vector，它是每一个 action 都有一个 value，然后你再把这两个东西加起来，就得到你的 Q value。

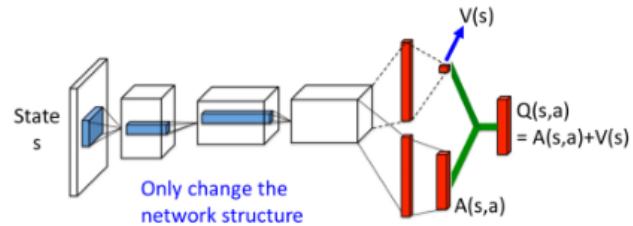
Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning", arXiv preprint, 2015

Dueling DQN



这么改有什么好？

Dueling DQN



state

Q(s,a)	action	3	3 4	3	1
		1	-1 0	6	1
		2	-2 -1	3	1
V(s)	Average of column	2	0 1	4	1
+ A(s,a)	sum of column = 0	1	3	-1	0
		-1	-1	2	0
		0	-2	-1	0

那我们假设说，原来的 $Q(s, a)$ ，它其实就是一个 table。我们假设 state 是 discrete 的，那实际上 state 不是 discrete 的，那为了说明方便，我们假设就是只有 4 个不同的 state，只有 3 个不同的 action，所以 $Q(s, a)$ 你可以看作是一个 table。

那我们说 $Q(s, a)$ 等于 $V(s)$ 加上 $A(s, a)$ ，那 $V(s)$ 是对不同的 state 它都有一个值， $A(s, a)$ 它是对不同的 state，不同的 action，都有一个值，那你把这个 V 的值加到 A 的每一个 column，就会得到 Q 的值。你把 V 加上 A ，就得到 Q 。

那今天假设说，你在 train network 的时候，你现在的 target 是希望，这一个值变成 4，这一个值变成 0，但是你实际上能更动的，并不是 Q 的值，你的 network 更动的是 V 跟 A 的值，根据 network 的参数， V 跟 A 的值 output 以后，就直接把它们加起来，所以其实不是更动 Q 的值。

然后在 learn network 的时候，假设你希望这边的值，这个 3 增加 1 变成 4，这个 -1 增加 1 变成 0，最后你在 train network 的时候，network 可能会选择说，我们就不要动这个 A 的值，就动 V 的值，把 V 的值，从 0 变成 1。那你把 0 变成 1 有什么好处呢？这个时候你会发现说，本来你只想动这两个东西的值，那你会发现说，这个第三个值也动了。所以有可能说你在某一个 state，你明明只 sample 到这 2 个 action，你没 sample 到第三个 action，但是你其实也可以更动到第三个 action 的 Q value。

那这样的好处就是，你就变成你不需要把所有的 state action pair 都 sample 过，你可以用比较 efficient 的方式，去 estimate Q value 出来。

因为有时候你 update 的时候，不一定是 update 下面这个 table，而是只 update 了 $V(s)$ ，但 update $V(s)$ 的时候，只要一改，所有的值就会跟着改，这是一个比较有效率的方法，去使用你的 data。

这个是 Dueling DQN 可以带给我们的的好处，那可是接下来有人就会问说，真的会这么容易吗？

会不会最后 learn 出来的结果是，反正 machine 就学到说我们也不要管什么 V 了， V 就永远都是 0，然后反正 A 就等于 Q ，那你就没有得到任何 Dueling DQN 可以带给你的好处，就变成跟原来的 DQN 一模一样。

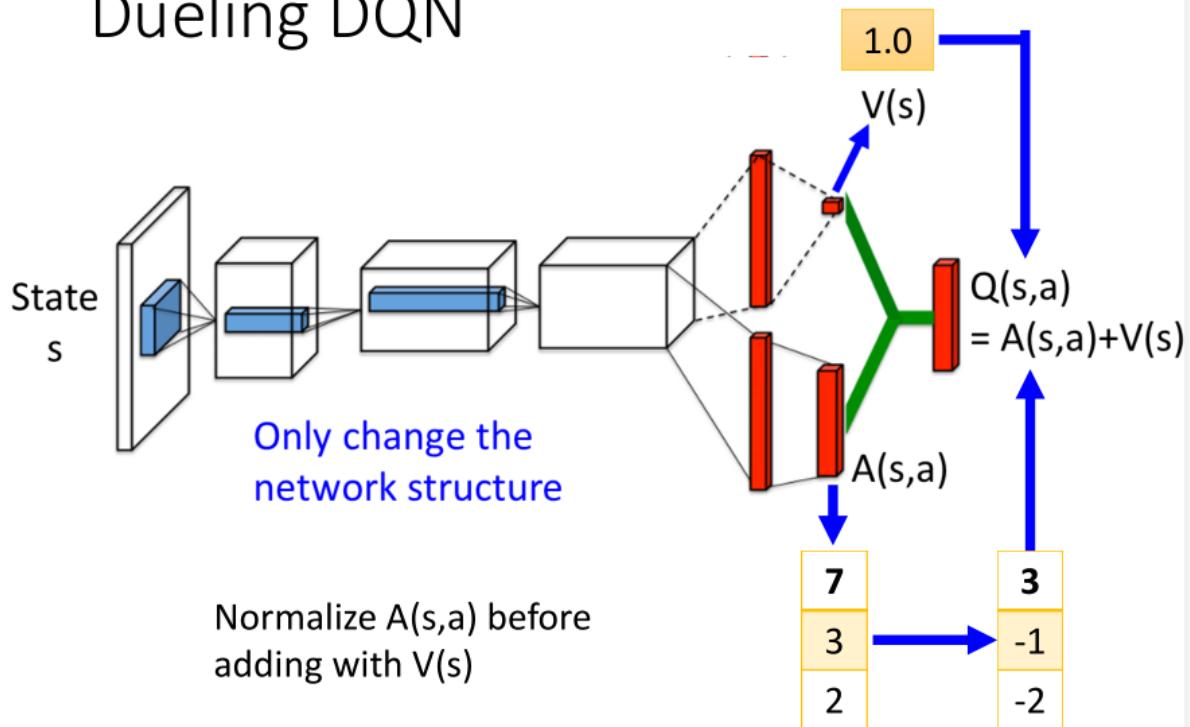
所以为了避免这个问题，实际上你会对下面这个 A 下一些 constrain。

你要给 A 一些 constrain，让 update A 其实比较麻烦，让 network 倾向于会想要去用 V 来解问题。举例来说，你可以看原始的文献，它有不同的 constrain。那个最直觉的 constrain 是，你必须要让这个 A 的每一个 column 的和都是 0，每一个 column 的值的和都是 0，所以看我这边举的例子，我的 column 的和都是 0。

那如果这边 column 的和都是 0，这边这个 V 的值，你就可以想成是上面 Q 的每一个 column 的平均值，这个平均值，加上这些值，才会变成是 Q 的 value。

所以今天假设你发现说你在 update 参数的时候，你是要让整个 row 一起被 update，你就不会想要 update A 这个 matrix，因为 A 这个 matrix 的每一个 column 的和都要是 0，所以你没有办法说，让这边的值，通通都 +1，这件事是做不到的，因为它的 constrain 就是你的和永远都是要 0，所以不可以都 +1，这时候就会强迫 network 去 update V 的值。这样你可以用比较有效率的方法，去使用你的 data。

Dueling DQN



那实作上怎么做呢？所以实作上我们刚才说，你要给这个 A 一个 constrain。

那所以在实际 implement 的时候，你会这样 implement。

假设你有 3 个 actions，然后在这边 network 的 output 的 vector 是 7 3 2，你在把这个 A 跟这个 B 加起来之前，先加一个 normalization，就好像做那个 layer normalization 一样，加一个 normalization。

这个 normalization 做的事情，就是把 $7+3+2$ 加起来等于 12， $12/3 = 4$ ，然后把这边通通减掉 4，变成 3, -1, 2，再把 3, -1, 2 加上 1.0，得到最后的 Q value。

这个 normalization 的这个 step，就是 network 的其中一部分，在 train 的时候，你从这边也是一路 back propagate 回来的。只是 normalization 这一个地方，是没有参数的，它就是一个 normalization 的 operation，它可以放到 network 里面，跟 network 的其他部分 jointly trained，这样 A 就会有比较大的 constrain，这样 network 就会给它一些 penalty，倾向于去 update V 的值，这个是 Dueling DQN。

Prioritized Reply

那其实还有很多技巧可以用，这边我们就比较快的带过去。

有一个技巧叫做 Prioritized Replay。Prioritized Replay 是什么意思呢？

我们原来在 sample data 去 train 你的 Q-network 的时候，你是 uniform 地从 experience buffer，从 buffer 里面去 sample data。那这样不见得是最好的，因为也许有一些 data 比较重要呢，比如你做不好的那些 data。

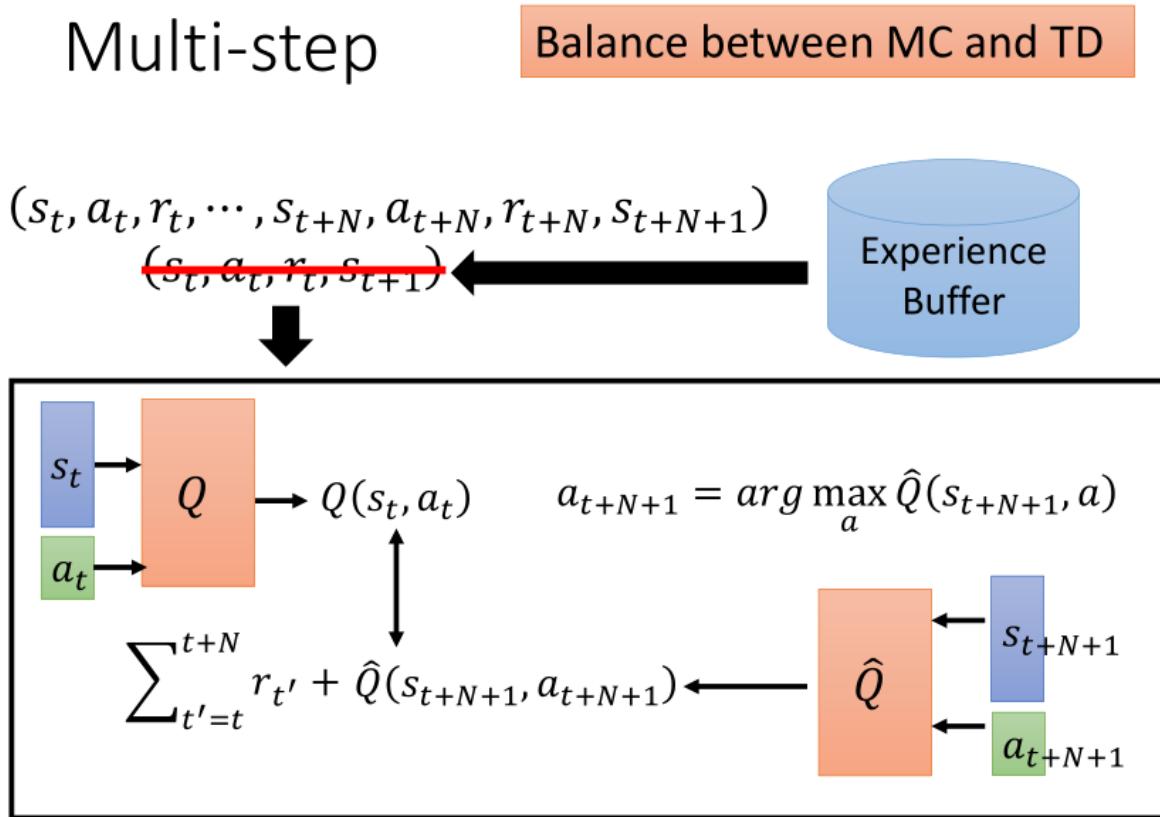
就假设有一些 data，你之前有 sample 过，你发现说那一笔 data 的 TD error 特别大，所谓 TD error 就是你的 network 的 output 跟 target 之间的差距。那这些 data 代表说你在 train network 的时候，你是比较 train 不好的，那既然比较 train 不好，那你就应该给它比较大的机率被 sample 到，所以这样在 training 的时候，才会考虑那些 train 不好的 training data 多一点。这个非常的直觉。

那详细的式子呢，你再去看一下 paper。

实际上在做 prioritized replay 的时候，你还不只会更改 sampling 的 process，你还会因为更改了 sampling 的 process，你会更改 update 参数的方法，所以 prioritized replay 其实不只是改变了，sample data 的 distribution 这么简单，你也会改 training process。

Multi-step

那另外一个可以做的方法是，你可以 balance MC 跟 TD，我们刚才讲说 MC 跟 TD 的方法，他们各自有各自的优劣。



我们怎么在 MC 跟 TD 里面取得一个平衡呢？那我们的做法是这样，在 TD 里面，你只需要存，在某一个 state s_t ，采取某一个 action a_t ，得到 reward r_t ，还有接下来跳到哪一个 state $s(t+1)$ ，但是我们现在可以不要只存一个 step 的 data，我们存 N 个 step 的 data，我们记录在 s_t 采取 a_t ，得到 r_t ，会跳到什么样 s_t ，一直纪录到在第 N 个 step 以后，在 $s(t+N)$ 采取 $a(t+N)$ 得到 reward $r(t+N)$ ，跳到 $s(t+N+1)$ 的这个经验，通通把它存下来。

实际上你今天在做 update 的时候，在做你 Q network learning 的时候，你的 learning 的方法会是这样，你要让 $Q(s_t, a_t)$ ，跟你的 target value 越接近越好。而你的 target value 是什么？你的 target value 是会把从时间 t，一直到 $t+N$ 的 N 个 reward 通通都加起来。然后你现在 Q hat 所计算的，不是 $s(t+1)$ ，而是 $s(t+N+1)$ ，你会把 N 个 step 以后的 state 丢进来，去计算 N 个 step 以后，你会得到的 reward，再加上 multi-step 的 reward，然后希望你的 target value，跟这个 multi-step reward 越接近越好。

那你会发现说这个方法，它就是 MC 跟 TD 的结合，因为它有 MC 的好处跟坏处，也有 TD 的好处跟坏处。

那如果看它的这个好处的话，因为我们现在 sample 了比较多的 step。之前是只 sample 了一个 step，所以某一个 step 得到的 data 是 real 的，接下来都是 Q value 估测出来的，现在 sample 比较多 step，sample N 个 step，才估测 value，所以估测的部分所造成的影响就会比较轻微。当然它的坏处就跟 MC 的坏处一样，因为你的 r 比较多项，你把大 N 项的 r 加起来，你的 variance 就会比较大。但是你可以去调这个 N 的值，去在 variance 跟不精确的 Q 之间取得一个平衡，那这个就是一个 hyper parameter，你要调这个大 N 到底是多少。你是要多 sample 三步，还是多 sample 五步，这个就跟 network structure 是一样，是一个你需要自己调一下的值。

Noisy Net

那还有其他的技术，有一个技术是要 improve 这个 exploration 这件事，我们之前讲的 Epsilon Greedy 这样的 exploration，它是在 action 的 space 上面加 noise。

但是有另外一个更好的方法叫做 Noisy Net，它是在参数的 space 上面加 noise。Noisy Net 的意思是说，每一次在一个 episode 开始的时候，在你要跟环境互动的时候，你就把你的 Q function 拿出来，那 Q function 里面其实就是一个 network，你把那个 network 拿出来，在 network 的每一个参数上面，加上一个 Gaussian noise，那你就把原来的 Q function，变成 Q tilde，因为 Q hat 已经用过，Q hat 是那个 target network，我们用 Q tilde 来代表一个 Noisy Q function。

Noisy Net

<https://arxiv.org/abs/1706.01905>

<https://arxiv.org/abs/1706.10295>

- Noise on Action (Epsilon Greedy)

$$a = \begin{cases} \arg \max_a Q(s, a), & \text{with probability } 1 - \varepsilon \\ \text{random}, & \text{otherwise} \end{cases}$$

- Noise on Parameters

Inject noise into the parameters of Q-function **at the beginning of each episode**

$$a = \arg \max_a \tilde{Q}(s, a) \quad Q(s, a) \xrightarrow{\text{Add noise}} \tilde{Q}(s, a)$$

The noise would **NOT** change in an episode.

那我们把每一个参数都可能都加上一个 Gaussian noise，你就得到一个新的 network 叫做 Q tilde。

那这边要注意的事情是，我们每次在 sample noise 的时候，要注意在每一个 episode 开始的时候，我们才 sample network。每个 episode 开始的时候，开始跟环境互动之前，我们就 sample network，接下来你就会用这个固定住的 noisy network，去玩这个游戏直到游戏结束，你才重新再去 sample 新的 noise。

那这个方法神奇的地方就是，OpenAI 跟 Deep Mind 又在同时间 propose 一模一样的方法，通通都 publish 在 ICLR 2018，两篇 paper 的方法就是一样的，不一样的地方是，他们用不同的方法，去加 noise。我记得那个 OpenAI 加的方法好像比较简单，他就直接加一个 Gaussian noise 就结束了，就你把每一个参数，每一个 weight，都加一个 Gaussian noise 就结束了。然后 Deep Mind 他们做比较复杂，他们的 noise 是由一组参数控制的，也就是说 network 可以自己决定说，它那个 noise 要加多大。

但是概念就是一样的，总之你就是把你的 Q function 里面的那个 network 加上一些 noise，把它变得跟原来的 Q function 不一样，然后拿去跟环境做互动。那两篇 paper 里面都有强调说，参数虽然会加 noise，但在同一个 episode 里面，你的参数就是固定的，你是在换 episode，玩第二场新的游戏的时候，你才会重新 sample noise，在同一场游戏里面，就是同一个 noisy Q network，在玩那一场游戏，这件事非常重要。

为什么这件事非常重要呢？因为这是 Noisy Net 跟原来的 Epsilon Greedy 或是其他在 action 做 sample 方法本质上的差异。

Noise on Action

- Given the same state, the agent may takes different actions.
- No real policy works in this way

隨機亂試

Noise on Parameters

- Given the same (similar) state, the agent takes the same action.
 - → State-dependent Exploration
- Explore in a *consistent* way

有系統地試

有什么样本质上的差异呢？在原来 sample 的方法，比如说 Epsilon Greedy 里面，就算是给同样的 state，你的 agent 采取的 action，也不一定是一样的。因为你是用 sample 决定的，given 同一个 state，你如果 sample 到说，要根据 Q function 的 network，你会得到一个 action，你 sample 到 random，你会采取另外一个 action。

所以 given 同样的 state，如果你今天是用 Epsilon Greedy 的方法，它得到的 action，是不一样的。但是你想看，实际上你的 policy，并不是这样运作的，在一个真实世界的 policy，给同样的 state，他应该会有同样的响应，而不是给同样的 state，它其实有时候吃 Q function，然后有时候又是随机的，所以这是一个比较奇怪的，不正常的 action，是在真实的情况下不会出现的 action。

但是如果你是在 Q function 上面去加 noise 的话，就不会有这个情形，在 Q function 的 network 的参数上加 noise，那在整个互动的过程中，在同一个 episode 里面，它的 network 的参数总是固定的。所以看到同样的 state，或是相似的 state，就会采取同样的 action，那这个是比较正常的。

那在 paper 里面有说，这个叫做 state dependent exploration，也就是说你虽然会做 explore 这件事，但是你的 explore 是跟 state 有关系的，看到同样的 state，你就会采取同样的 exploration 的方式。也就是说你在 explore 你的环境的时候，你是用一个比较 consistent 的方式，去测试这个环境，也就是上面你是 noisy 的 action，你只是随机乱试，但是如果你是在参数下加 noise，那在同一个 episode 里面，你的参数是固定的，那你就是有系统地在尝试。每次会试说，在某一个 state，我都向左试试看，然后再下一次在玩这个同样游戏的时候，看到同样的 state，你就说我再向右试试看，你是有系统地在 explore 这个环境。

Distributional Q-function

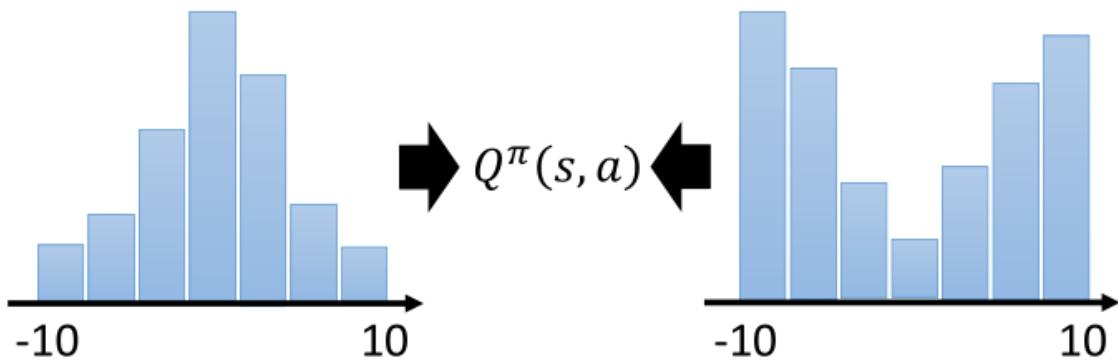
Distributional Q-function，我们就讲大的概念。

Distributional Q-function 我觉得还蛮有道理的，但是它没有红起来，你就发现说没有太多人真的在实作的时候用这个技术，可能一个原因就是，是因为他不好实作。

我们说 Q function 是 accumulated reward 的期望值。

State-action value function $Q^\pi(s, a)$

- When using actor π , the cumulated reward expects to be obtained after seeing observation s and taking a



Different distributions can have the same values.

所以我们算出来的这个 Q value 其实是一个期望值，也就是说实际上我在某一个 state 采取某一个 action 的时候，因为环境是有随机性，在某一个 state 采取某一个 action 的时候，实际上我们把所有的 reward 玩到游戏结束，的时候所有的 reward，进行一个统计，你其实得到的是一个 distribution。

也许在 reward 得到 0 的机率很高，在 -10 的机率比较低，在 +10 的机率比较低，它是一个 distribution。

那这个 Q value 代表的值是说，我们对这一个 distribution 算它的 mean，才是这个 Q value，我们算出来是 expected accumulated reward。真正的 accumulated reward 是一个 distribution，对它取 expectation，对它取 mean，你得到了 Q value。

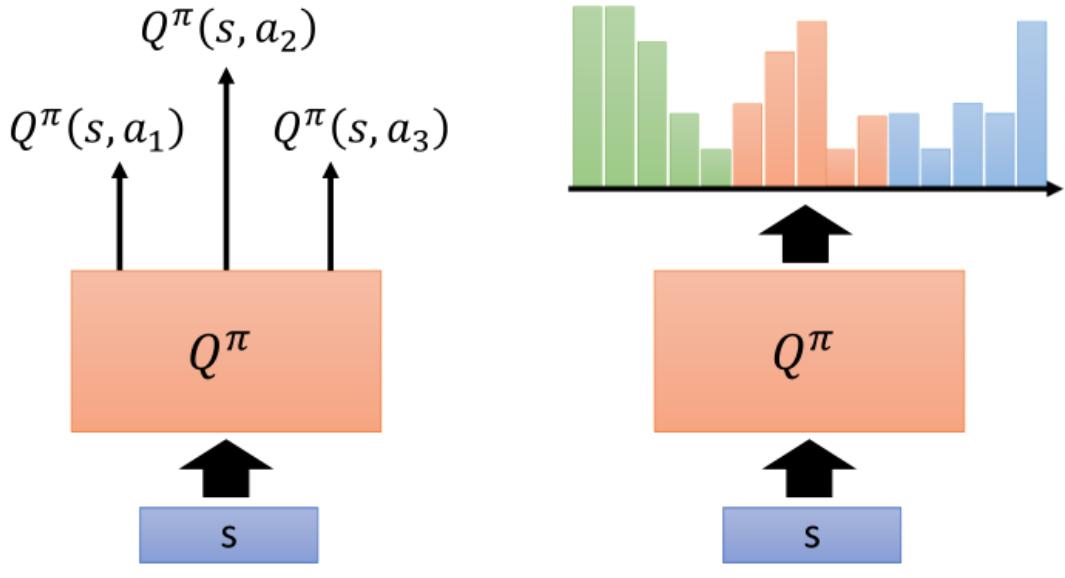
但是有趣的地方是，不同的 distribution，他们其实可以有同样的 mean，也许真正的 distribution 是这个样子，它算出来的 mean 跟这个 distribution 算出来的 mean，其实是一样的，但它们背后所代表的 distribution 是不一样的。

所以今天假设我们只用一个 expected 的 Q value，来代表整个 reward 的话。其实可能是有一些 information 是 loss 的，你没有办法 model reward 的 distribution。

所以今天 Distributional Q function 它想要做的事情是，model distribution。所以怎么做？

在原来的 Q function 里面，假设你只能够采取 $a_1, a_2, a_3, 3$ 个 actions，那你就是 input 一个 state，output 3 个 values，3个 values 分别代表 3 个 actions 的 Q value。但是这个 Q value 是一个 distribution 的期望值。

所以今天 Distributional Q function，它的 ideas 就是何不直接 output 那个 distribution。但是要直接 output 一个 distribution 也不知道怎么做，实际上的做法是说，假设 distribution 的值就分布在某一个 range 里面，比如说 -10 到 10，那把 -10 到 10 中间，拆成一个一个的 bin，拆成一个一个的直方图。



A network with 3 outputs

A network with 15 outputs
(each action has 5 bins)

举例来说，在这个例子里面，对我们把 reward 的 space 就拆成 5 个 bin。详细一点的作法就是，假设 reward 可以拆成 5 个 bin 的话，今天你的 Q function 的 output，是要预测你在某一个 state，采取某一个 action，你得到的 reward，落在某一个 bin 里面的机率。所以其实这边的机率的和，这些绿色的 bar 的和应该是 1，它的高度代表说，在某一个 state，采取某一个 action 的时候，它落在某一个 bin 的机率。这边绿色的代表 action 1，红色的代表 action 2，蓝色的代表 action 3。

所以今天你就可以真的用 Q function 去 estimate a_1 的 distribution, a_2 的 distribution, a_3 的 distribution。

那实际上在做 testing 的时候，我们还是要选某一个 action，去执行。那选哪一个 action 呢？实际上在做的时候，它还是选这个 mean 最大的那个 action 去执行。但是假设我们今天可以 model distribution 的话，除了选 mean 最大的以外，也许在未来你可以有更多其他的运用。

举例来说，你可以考虑它的 distribution 长什么样子，若 distribution variance 很大，代表说采取这个 action，虽然 mean 可能平均而言很不错，但也许风险很高。你可以 train 一个 network 它是可以规避风险的，就在 2 个 action mean 都差不多的情况下，也许他可以选一个风险比较小的 action 来执行。这是 Distributional Q function 的好处。

那细节，怎么 train 这样的 Q network，我们就不讲，你只要记得说反正 Q network 有办法 output 一个 distribution 就对了。我们可以不只是估测得到的期望 reward mean 的值，我们其实是可以估测一个 distribution 的。

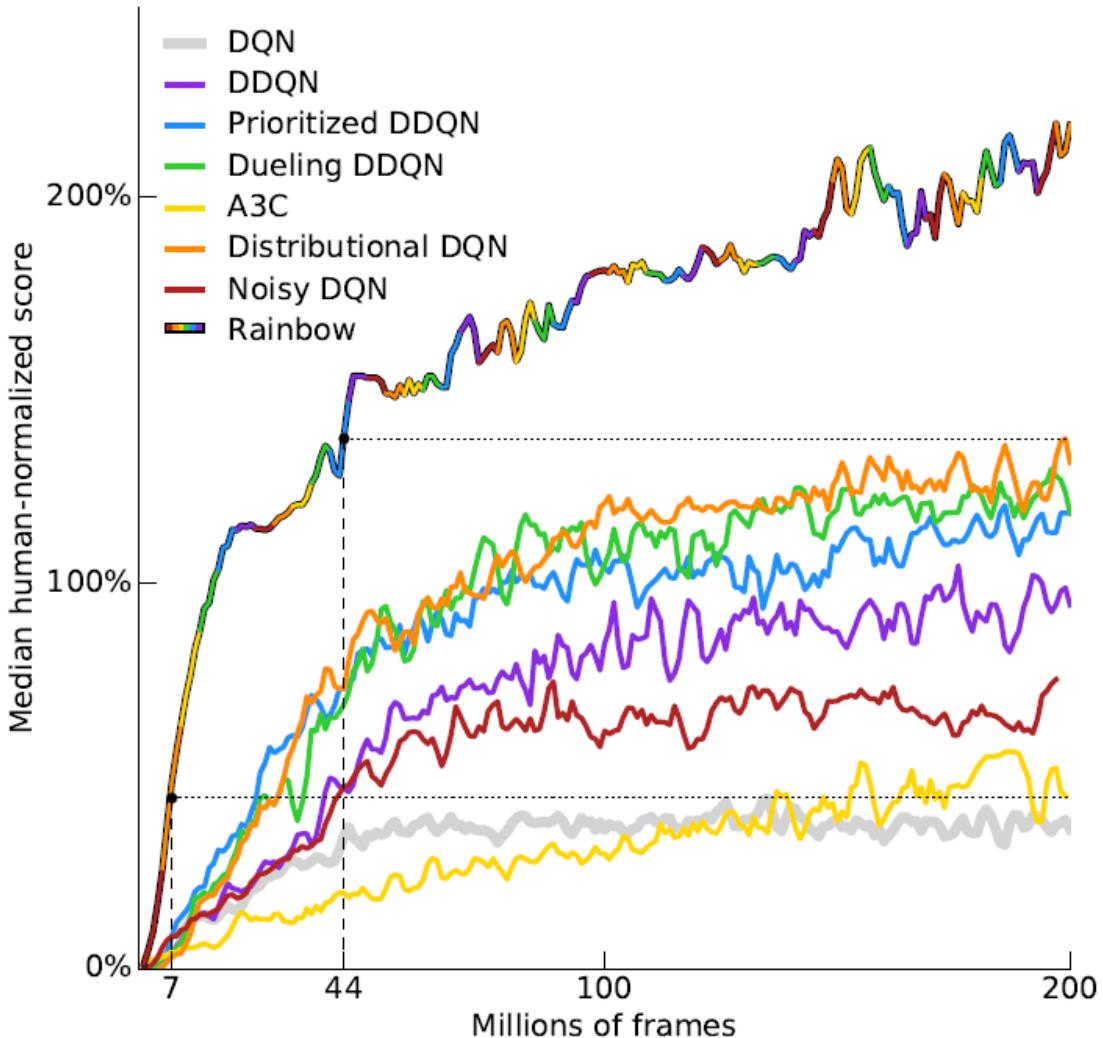
Rainbow

那最后跟大家讲的是一个叫做 rainbow 的技术，这个 rainbow 它的技术是什么呢？

rainbow 这个技术就是，把刚才所有的方法都综合起来就变成 rainbow。

因为刚才每一个方法，就是有一种自己的颜色，把所有的颜色通通都合起来，就变成 rainbow，我仔细算一下，不是才 6 种方法而已吗？为什么你会变成是 7 色的，也许它把原来的 DQN 也算是一种方法。

那我们来看看这些不同的方法。



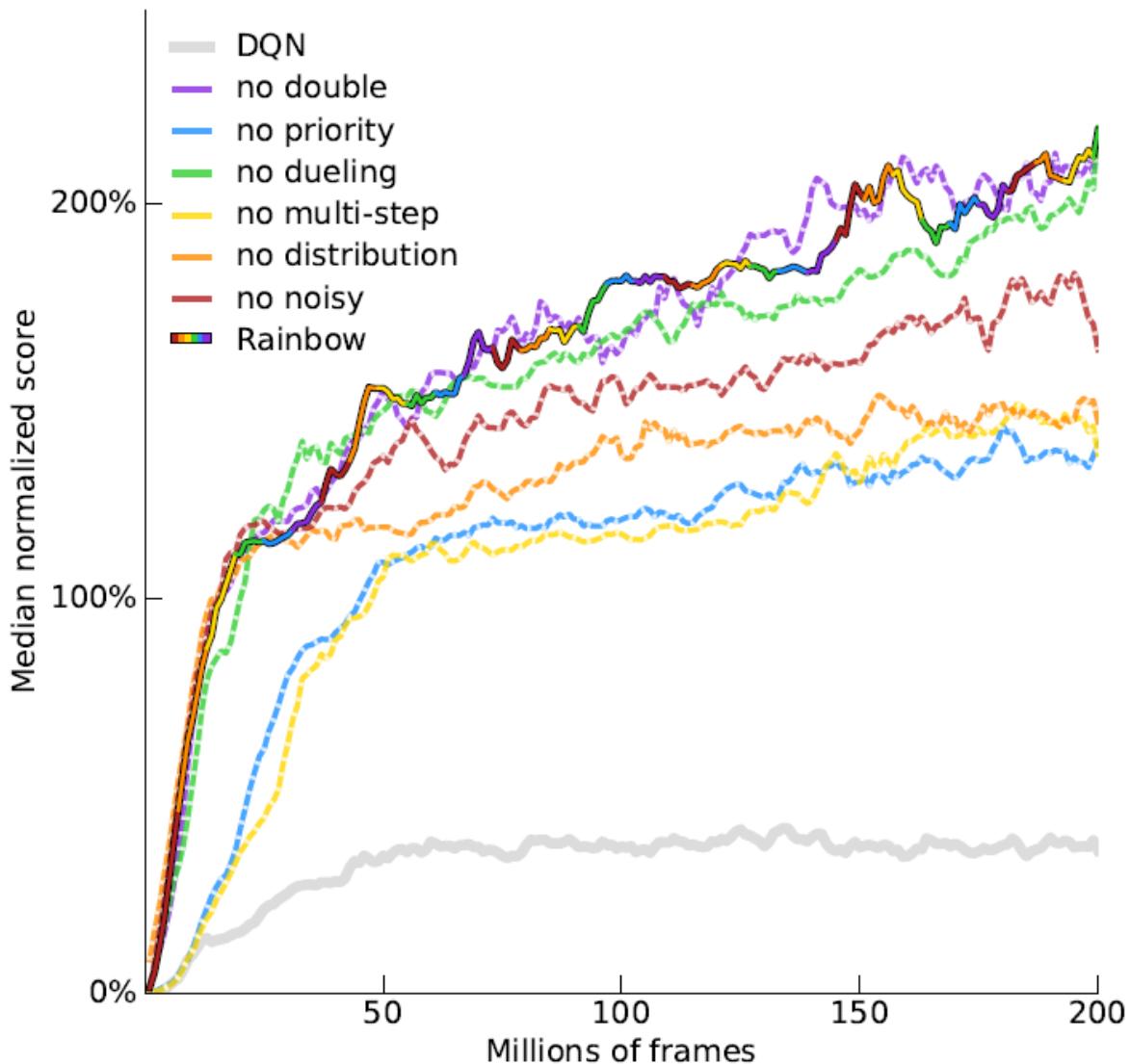
这个横轴是你 training process，纵轴是玩了 10 几个 ATARI 小游戏的平均的分数的和，但它取的是 median 的分数，为什么是取 median 不是直接取平均呢？因为它说每一个小游戏的分数，其实差很多，如果你取平均的话，到时候某几个游戏就 dominate 你的结果，所以它取 median 的值。

那这个如果你是一般的 DQN，就是灰色这一条线，没有很强。

那如果是你换 noisy DQN，就强很多，然后如果这边每一个单一颜色的线是代表说只用某一个方法，那紫色这一条线是 DDQN double DQN，DDQN 还蛮有效的，你换 DDQN 就从灰色这条线跳成紫色这一条线，然后 Prioritized DQN，Dueling DQN，还有 Distributional DQN 都蛮强的，它们都差不多很强的。

这边有个 A3C，A3C 其实是 Actor-Critic 的方法。那单纯的 A3C 看起来是比 DQN 强的，这边没有 Multi step 的方法，我猜是因为 A3C 本身内部就有做 Multi step 的方法，所以他可能觉得说有 implement A3C就算是有 implement，Multi step 的方法，所以可以把这个 A3C 的结果想成是 Multi step 的方法。

最后其实这些方法他们本身之间是没有冲突的，所以全部都用上去，就变成七彩的一个方法，就叫做 rainbow，然后它很高这样。



这是下一张图，这张图要说的是什么呢？这张图要说的事情是说，在 rainbow 这个方法里面，如果我们每次拿掉其中一个技术，到底差多少。因为现在是把所有的方法通通倒在一起，发现说进步很多，但会不会有些方法其实是没用的。所以看看说，哪些方法特别有用，哪些方法特别没用，所以这边的虚线就是，拿掉某一种方法以后的结果，那你发现说，拿掉 Multi time step 掉很多，然后拿掉 Prioritized replay，也马上就掉下来，拿掉这个 distribution，它也掉下来。

那这边有一个有趣的地方是说，在开始的时候，distribution 训练的方法跟其他方法速度差不多，但是如果你拿掉 distribution 的时候，你的训练不会变慢，但是你最后 performance，最后会收敛在比较差的地方。

拿掉 Noisy Net，performance 也是差一点，拿掉 Dueling 也是差一点，那发现拿掉 Double，没什么用这样子，拿掉 Double 没什么差，所以看来全部倒再一起的时候，Double 是比较没有影响的。

那其实在 paper 里面有给一个 make sense 的解释说，其实当你有用 Distributional DQN 的时候，本质上就不会 over estimate 你的 reward。

因为我们之所以用 Double 是因为，害怕会 over estimate reward，那在 paper 里面有讲说，如果有做 Distributional DQN，就比较不会有 over estimate 的结果。

事实上他有真的算了一下发现说，它其实多数的状况，是 under estimate reward 的，所以会变成 Double DQN 没有用。

那为什么做 Distributional DQN，不会 over estimate reward，反而会 under estimate reward 呢？可能是说，现在这个 distributional DQN，我们不是说它 output 的是一个 distribution 的 range 吗？所以你 output 的那个 range 啊，不可能是无限宽的，你一定是设一个 range，比如说我最大 output range 就是从 -10 到 10，那假设今天得到的 reward 超过 10 怎么办？是 100 怎么办，就当作没看到这件事，所以

会变成说，reward 很极端的值，很大的值，其实是会被丢掉的，所以变成说你今天用 Distributional DQN 的时候，你不会有 over estimate 的现象，反而有 under estimate 的倾向。

Q-Learning for Continuous Actions

那其实跟 policy gradient based 方法比起来，Q learning 其实是比较稳的，policy gradient 其实是没有太多游戏是玩得起来的。policy gradient 比较不稳，尤其在没有 PPO 之前，你很难用 policy gradient 做什么事情，Q learning 相对而言是比较稳的，可以看最早 Deep reinforcement learning 受到大家注意，最早 deep mind 的 paper 拿 deep reinforcement learning 来玩 Atari 的游戏，用的就是 Q-learning。

那我觉得 Q-learning 比较容易，比较好 train 的一个理由是，我们说在 Q-learning 里面，你只要能够 estimate 出 Q-function，就保证你一定可以找到一个比较好的 policy，也就是你只要能够 estimate 出 Q-function，就保证你可以 improve 你的 policy，而 estimate Q function 这件事情，是比较容易的。

为什么？因为它就是一个 regression 的 problem，在这个 regression 的 problem 里面，你可以轻易地知道，你现在的 model learn 的是不是越来越好，你只要看那个 regression 的 loss 有没有下降，你就知道说你的 model learn 的好不好。

所以 estimate Q function 相较于 learn 一个 policy，是比较容易的，你只要 estimate Q function，就可以保证你现在一定会得到比较好的 policy，所以一般而言 Q learning 是比较容易操作。

那 Q learning 有什么问题呢？它一个最大的问题是，它不太容易处理 continuous action。

Continuous Actions

很多时候你的 action 是 continuous 的，什么时候你的 action 会是 continuous 的呢？你的 agent 只需要决定，比如说上下左右，这种 action 是 discrete 的，那很多时候你的 action 是 continuous 的，举例来说假设你的 agent 要做的事情是开自驾车，它要决定说它方向盘要左转几度，右转几度，这是 continuous 的。假设你的 agent 是一个机器人，它的每一个 action 对应到的就是它的，假设它身上有 50 个关节，它的每一个 action 就对应到它身上的这 50 个关节的角度，而那些角度，也是 continuous 的。

所以很多时候你的 action，并不是一个 discrete 的东西。它是一个 vector，这个 vector 里面，它的每一个 dimension 都有一个对应的 value，都是 real number，它是 continuous 的。

假设你的 action 是 continuous 的时候，做 Q learning 就会有困难，为什么呢？

因为我们说在做 Q-learning 里面，很重要的一步是，你要能够解这个 optimization 的 problem。你 estimate 出 Q function， $Q(s, a)$ 以后，必须要找到一个 a ，它可以让 $Q(s, a)$ 最大，假设 a 是 discrete 的，那 a 的可能性都是有限的。举例来说 Atari 的小游戏里面， a 就是上下左右跟开火。它是有限的，你可以把每一个可能的 action 都带到 Q 里面，算它的 Q value。

但是假如 a 是 continuous 的，会很麻烦，你无法穷举所有可能 continuous action 试试看那一个 continuous action 可以让 Q 的 value 最大。所以怎么办呢？就有各种不同的 solution。

Action a is a *continuous vector*

$$a = \arg \max_a Q(s, a)$$

Solution 1

Sample a set of actions: $\{a_1, a_2, \dots, a_N\}$

See which action can obtain the largest Q value

Solution 2

Using gradient ascent to solve the optimization problem.

Solution 1

第一个 solution 是，假设你不知道怎么解这个问题，因为 a 是很多的， a 是没有办法穷举的，怎么办？用 sample。

sample 出 N 个可能的 a ，一个一个带到 Q function 里面，那看谁最快。这个方法其实也不会太不 efficient，因为其实你真的在运算的时候，你会用 GPU，所以你一次会把 N 个 continuous action，都丢到 Q function 里面，一次得到 N 个 Q value，然后看谁最大，那当然这个不是一个非常精确的做法，因为你真的没有办法做太多的 sample，所以你 estimate 出来的 Q value，你最后决定的 action，可能不是非常的精确。

Solution 2

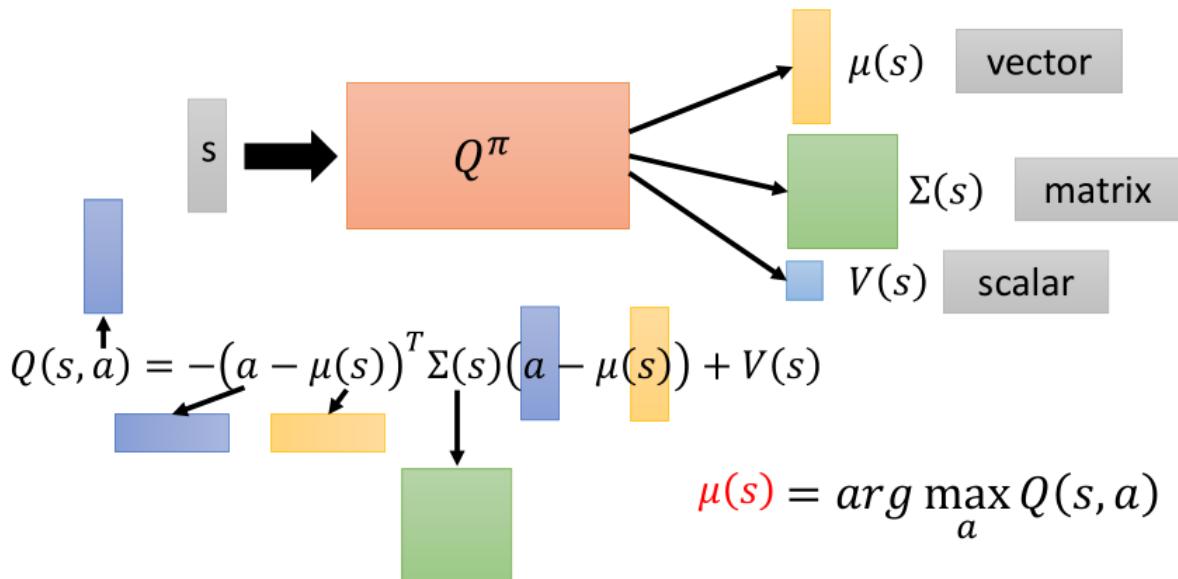
第二个 solution 是今天既然我们要解的是一个 optimization 的 problem，你会不会解这种 optimization 的 problem 呢？你其实是会的，因为你其实可以用 gradient decent 的方法，来解这个 optimization 的 problem，我们现在其实是要 maximize 我们的 objective function，所以是 gradient ascent，我的意思是一样的。你就把 a 当作是你的 parameter，然后你要找一组 a 去 maximize 你的 Q function，那你就用 gradient ascent 去 update a 的 value，最后看看你能不能找到一个 a ，去 maximize 你的 Q function，也就是你的 objective function。当然这样子你会遇到的问题就是 global maximum，不见得能够真的找到最 optimal 的结果。而且这个运算量显然很大，因为你要 iterative 的去 update 你的 a ，我们 train 一个 network 就很花时间了，今天如果你是用 gradient ascent 的方法来处理这个 continuous 的 problem，等于是你每次要决定要 take 哪一个 action 的时候，你都还要做一次 train network 的 process，这个显然运算量是很大的。

Solution 3

第三个 solution 是，特别 design 一个 network 的架构，特别 design 你的 Q function，使得解那个 arg max 的 problem，变得非常容易。也就是这边的 Q function 不是一个 general 的 Q function，特别设计一下它的样子，让你要找哪一个 a 可以让这个 Q function 最大的时候，非常容易。

那这边是一个例子，这边有我们的 Q function，然后这个 Q function 它的作法是这样，input 你的 state s ，通常它就是一个 image，它可以用一个向量，或是一个 matrix 来表示。

Solution 3 Design a network to make the optimization easy.



input 这个 s , 这个 Q function 会 output 3 个东西, 它会 output $\mu(s)$, 这是一个 vector, 它会 output $\Sigma(s)$, 是一个 matrix, 它会 output $V(s)$, 是一个 scalar。output 这 3 个东西以后, 我们知道 Q function 其实是吃一个 s 跟 a , 然后决定一个 value。

Q function 意思是说在某一个 state, take 某一个 action 的时候, 你 expected 的 reward 有多大, 到目前为止这个 Q function 只吃 s , 它还没有吃 a 进来。 a 在那里呢? 当这个 Q function 吐出 μ , Σ 跟 V 的时候, 我们才把 s 引入, 用 a 跟这 3 个东西互相作用一下, 你才算出最终的 Q value。

a 怎么和这 3 个东西互相作用呢? 它的作用方法就写在下面, 所以实际上 $Q(s, a)$, 你的 Q function 的运作方式是, 先 input s , 让你得到 μ , Σ 跟 V , 然后再 input a , 然后接下来的计算方法是把 a 跟 μ 相减。

注意一下 a 现在是 continuous 的 action, 所以它也是一个 vector, 假设你现在是要操作机器人的话, 这个 vector 的每一个 dimension, 可能就对应到机器人的某一个关节, 它的数值, 就是那关节的角度, 所以 a 是一个 vector。

把 a 的这个 vector, 减掉 μ 的这个 vector, 取 transpose, 所以它是一个横的 vector, Σ 是一个 matrix, 然后 a 减掉 $\mu(s)$, 这两个都是 vector, 减掉以后还是一个竖的 vector。

然后接下来你把这一个 vector, 乘上这个 matrix, 再乘上这个 vector, 你得到的是什么? 你得到是一个 scalar。

把这个 scalar 再加上 $V(s)$, 得到另外一个 scalar, 这一个数值就是你的 $Q(s, a)$, 就是你的 Q value。

假设我们的 $Q(s, a)$ 定义成这个样子, 我们要怎么找到一个 a , 去 maximize 这个 Q value 呢?

其实这个 solution 非常简单, 因为我们把 formulation 写成这样, 那什么样的 a , 可以让这一个 Q function 最终的值最大呢? 因为 a 减 μ 乘上 Σ , 再乘上 a 减 μ 这一项一定是正的, 然后前面乘上一个负号, 所以第一项这个值越小, 你最终的这个 Q value 就越大。

因为我们是把 V 减掉第一项, 所以第一项, 假设不要看这个负号的话, 第一项的值越小, 最后的 Q value 就越大。

怎么让第一项的值最小呢? 你直接把 a 带 μ , 让它变成 0, 就会让第一项的值最小。

这个东西, 就像是那个 Gaussian distribution, 所以 μ 就是 Gaussian 的 mean, Σ 就是 Gaussian 的 variance, 但是 variance 是一个 positive definite 的 matrix。所以其实怎么样让这个 Σ , 一定是 positive definite 的 matrix 呢? 其实在 Q^π 里面, 它不是直接 output Σ , 就如果直接 output 一个 Σ , 它可能不见得是 positive definite 的 matrix。它其实是 output 一个 matrix, 然后再把那个 matrix 跟另外一个 matrix, 做 transpose 相乘, 然后可以确保它是 positive definite 的。

这边要强调的点就是说，实际上它不是直接output一个 matrix，你去那个 paper 里面 check 一下它的 trick，它可以保证说 sigma 是 positive definite 的。

所以今天前面这一项，因为 sigma 是 positive definite，所以它一定是正的。

所以现在怎么让它值最小呢？你就把 a 带 mu(s)。

把你 a 带 mu(s) 以后呢，你可以让 Q 的值最大，所以这个 problem 就解了。

所以今天假设要你 arg max 这个东西，虽然 in general 而言，若 Q 是一个 general function，你很难算，但是我们这边 design 了 Q 这个 function，所以 a 只要设 mu(s)，我们就得到 maximum 的 value，你在解这个 arg max 的 problem 的时候，就变得非常的容易。

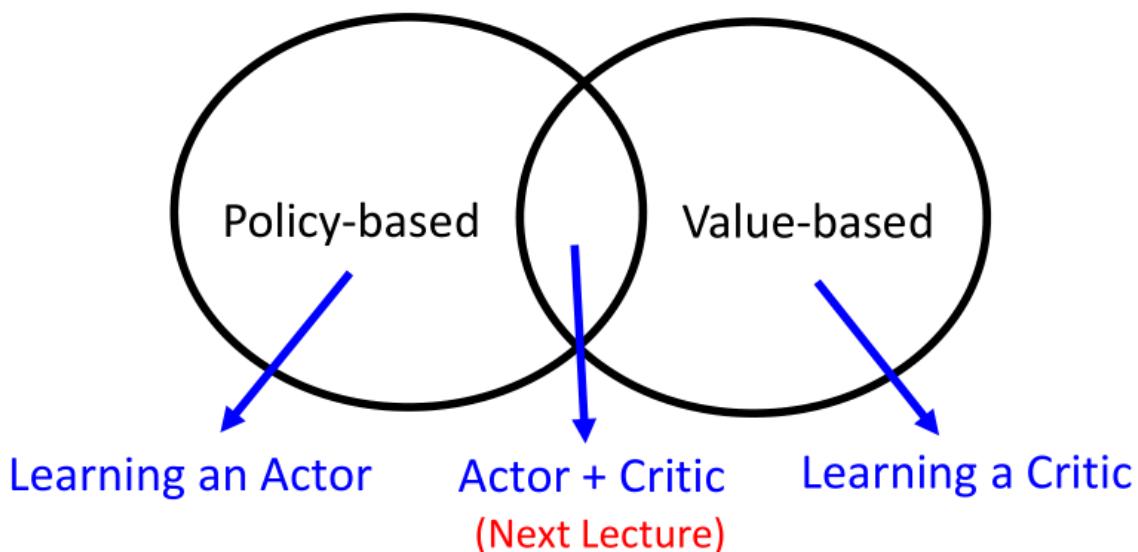
所以其实 Q learning 也不是不能够用在 continuous case。

是可以用的，只是就是有一些局限，就是你的 function 就是不能够随便乱设，它必须有一些限制。

Solution 4

第 4 招就是不要用 Q-learning，用 Q learning 处理 continuous 的 action 还是比较麻烦。

Solution 4 Don't use Q-learning



那到目前为止，我们讲了 policy based 的方法，我们讲了 PPO，讲了 value based 的方法，也就是 Q learning，但是这两者其实是可以结合在一起的，也就是 Actor-Critic 的方法。

Actor-Critic

在 Actor-Critic 里面，最知名的方法就是 A3C，Asynchronous Advantage Actor-Critic。如果去掉前面这个 Asynchronous，只有 Advantage Actor-Critic，就叫做 A2C。

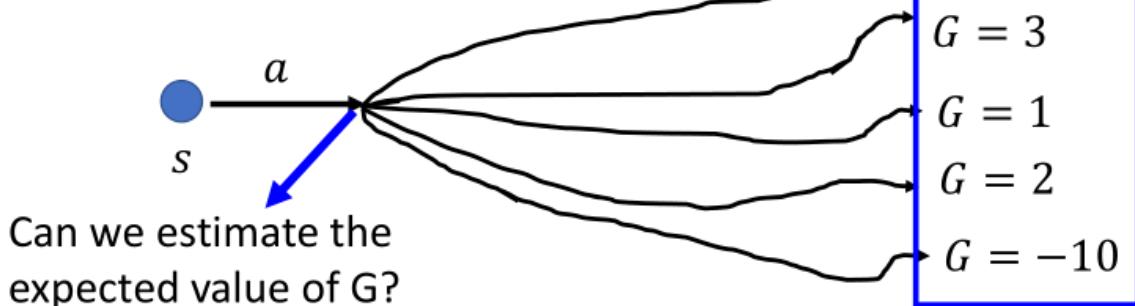
Review – Policy Gradient

那我们很快复习一下 policy gradient，在 policy gradient 里面，我们是怎么说的呢？在 policy gradient 里面我们说我们在 update policy 的参数 θ 的时候，我们是用了以下这个式子，来算出我们的 gradient。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \left(\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b \right) \nabla \log p_\theta(a_t^n | s_t^n)$$

baseline
 G_t^n : obtained via interaction
Very unstable

With sufficient samples,
approximate the expectation of G.



那我们说这个式子其实是还蛮直觉的，这个式子在说什么呢？我们先让 agent 去跟环境互动一下，然后我们知道我们在某一个 state s ，采取了某一个 action a ，那我们可以计算出在某一个 state s ，采取了某一个 action a 的机率。接下来，我们去计算说，从这一个 state 采取这个 action a 之后，accumulated reward 有多大。从这个时间点开始，在某一个 state s ，采取了某一个 action a 之后，到游戏结束，互动结束为止，我们到底 collect 了多少的 reward。

那我们把这些 reward，从时间 t 到时间 T 的 reward 通通加起来。

有时候我们会在前面，乘一个 discount factor，因为我们之前也有讲过说，离现在这个时间点比较久远的 action，它可能是跟现在这个 action 比较没有关系的，所以我们会给它乘一个 discount 的 factor，可能设 0.9 或 0.99。

那我们接下来还说，我们会减掉一个 baseline b ，减掉这个值 b 的目的，是希望括号这里面这一项，是有正有负的。那如果括号里面这一项是正的，那我们就要增加在这个 state 采取这个 action 的机率，如果括号里面是负的，我们就要减少在这个 state 采取这个 action 的机率。

那我们把这个 accumulated reward，从这个时间点采取 action a ，一直到游戏结束为止会得到的 reward，用 G 来表示它。但是问题是 G 这个值啊，它其实是非常的 unstable 的。

为什么说 G 这个值是非常的 unstable 的呢？因为这个互动的 process，其实本身是有随机性的，所以我们在某一个 state s ，采取某一个 action a ，然后计算 accumulated reward。每次算出来的结果，都是不一样的，所以 G 其实是一个 random variable，给同样的 state s ，给同样的 action a ， G 它可能有一个固定的 distribution。但我们可以采取 sample 的方式，我们在某一个 state s ，采取某一个 action a ，然后玩到底，我们看看说我们会得到多少的 reward，我们把这个东西当作 G 。

把 G 想成是一个 random variable 的话，我们实际上做的事情是，对这个 G 做一些 sample，然后拿这些 sample 的结果，去 update 我们的参数。

但实际上在某一个 state s 采取某一个 action a ，接下来会发生什么事，它本身是有随机性的，虽然说有个固定的 distribution，但它本身是有随机性的。而这个 random variable，它的 variance，可能会非常的大。你在同一个 state 采取同一个 action，你最后得到的结果，可能会是天差地远的。

那今天假设我们在每次 update 参数之前，我们都可以 sample 足够的次数，那其实没有什么问题。

但问题就是，我们每次做 policy gradient，每次 update 参数之前都要做一些 sample，这个 sample 的次数，其实是不可能太多的，我们只能够做非常少量的 sample。那如果你今天正好 sample 到差的结果，比如说你正好 sample 到 $G = 100$ ，正好 sample 到 $G = -10$ ，那显然你的结果会是很差的。

所以接下来我们要问的问题是，能不能让这整个 training process，变得比较 stable 一点，我们能不能够直接估测， G 这个 random variable 的期望值。

我们在 state s 采取 action a 的时候，我们直接想办法用一个 network 去估测在 state s 采取 action a 的时候，你的 G 的期望值。

如果这件事情是可行的，那之后 training 的时候，就用期望值来代替 sample 的值，那这样会让 training 变得比较 stable。

Review – Q-Learning

那怎么拿期望值代替 sample 的值呢？这边就需要引入 value based 的方法。

value based 的方法我们介绍的就是 Q learning。在讲 Q learning 的时候我们说，有两种 functions，有两种 critics，第一种 critic 我们写作 V ，它的意思是说，假设我们现在 actor 是 π ，那我们拿 π 去跟环境做互动，当今天我们看到 state s 的时候，接下来 accumulated reward 的期望值有多少。

- State value function $V^\pi(s)$
 - When using actor π , the *cumulated reward expects to be obtained after visiting state s*
 - State-action value function $Q^\pi(s, a)$
 - When using actor π , the *cumulated reward expects to be obtained after taking a at state s*
-
- Estimated by TD or MC

还有另外一个 critic，叫做 Q ， Q 是吃 s 跟 a 当作 input，它的意思是说，在 state s 采取 action a ，接下来都用 actor π 来跟环境进行互动，那 accumulated reward 的期望值，是多少。

V input s , output 一个 scalar, Q input s , 然后它会给每一个 a 呢，都 assign 一个 Q value。

那 estimate 的时候，你可以用 TD 也可以用 MC。TD 会比较稳，用 MC 比较精确。

Actor-Critic

那接下来我们要做的事情其实就是， G 这个 random variable，它的期望值到底是什么呢？其实 G 的 random variable 的期望值，正好就是 Q 这样子。

因为这个就是 Q 的定义， Q 的定义就是，在某一个 state s ，采取某一个 action a ，假设我们现在的 policy，就是 π 的情况下，accumulated reward 的期望值有多大，而这个东西就是 G 的期望值。 Q function 的定义，其实就是 accumulated reward 的期望值，就是 G 的期望值。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \left(\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b \right) \nabla \log p_\theta(a_t^n | s_t^n)$$

G_t^n : obtained via interaction

$$E[G_t^n] = Q^{\pi_\theta}(s_t^n, a_t^n)$$

所以我们现在要做的事情就是，假设我们把式子中的G用期望值来代表的话，然后把Q function套在这里，就结束了，那我们就可以Actor跟Critic这两个方法，把它结合起来。

这个其实很直觉。通常一个常见的做法是，就用value function，来表示baseline。所谓value function的意思就是说，假设现在policy是 π ，在某一个state s，一直interact到游戏结束，那你expected的reward有多大。V没有involve action，然后Q有involve action。那其实V它是Q的期望值，所以你今天把Q，减掉V，你的括号里面这一项，就会是有正有负的。

所以我们现在很直觉的，我们就把原来在policy gradient里面，括号这一项，换成了Q function的value，减掉V function的value，就结束了。

Advantage Actor-Critic

那接下来呢，其实你可以就单纯的这么实作，但是如果你这么实作的话，他有一个缺点是，你要estimate两个networks，而不是一个network，你要estimate Q这个network，你也要estimate V这个network，你现在就有两倍的风险，你有estimate估测不准的风险就变成两倍，所以我们何不只估测一个network就好了呢？

$$Q^\pi(s_t^n, a_t^n) - V^\pi(s_t^n)$$

Estimate two networks? We can only estimate one.



$$r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)$$

Only estimate state value
A little bit variance

$$Q^\pi(s_t^n, a_t^n) = E[r_t^n + V^\pi(s_{t+1}^n)]$$

$$Q^\pi(s_t^n, a_t^n) = r_t^n + V^\pi(s_{t+1}^n)$$

事实上在这个 Actor-Critic 方法里面，你可以只估测 V 这个 network。你可以把 Q 的值，用 V 的值来表示。什么意思呢？

现在其实 $Q(s, a)$ 呢，它可以写成 $r + V(s)$ 的期望值。当然这个 r 这个本身，它是一个 random variable，就是你今天在 state s ，你采取了 action a ，接下来你会得到什么样的 reward，其实是不确定的，这中间其实是有随机性的。所以小 r 呢，它其实是一个 random variable，所以要把右边这个式子，取期望值它才会等于 Q function。但是，我们现在把期望值这件事情去掉，就当作左式等于右式，就当作 Q function 等于 r 加上 state value function。

然后接下来我们就可以把这个 Q function，用 $r + V$ 取代掉。变成 $r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)$

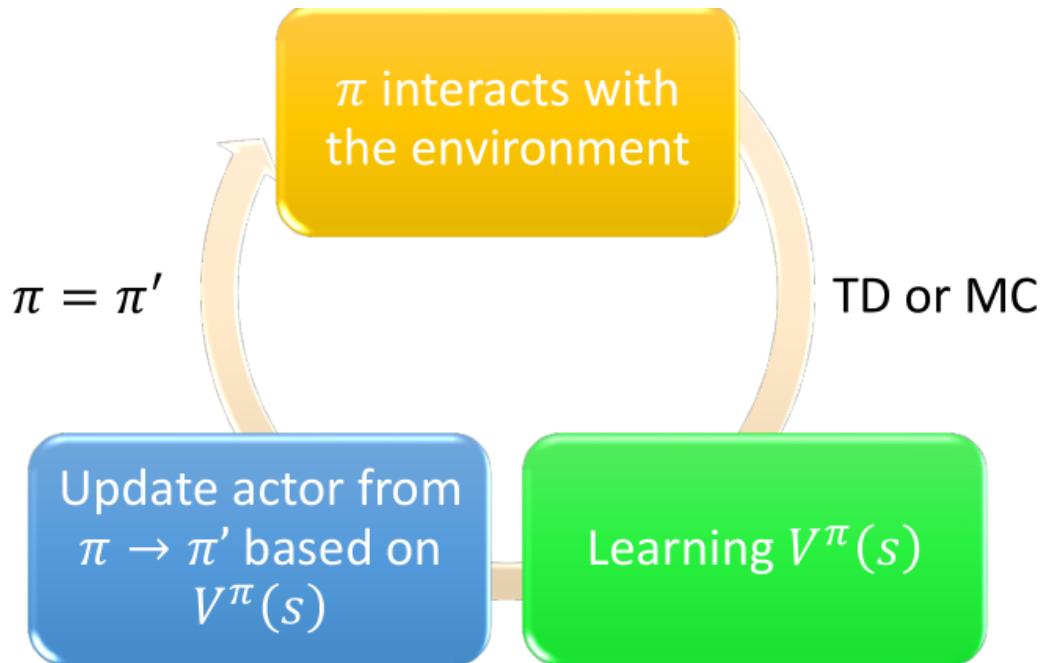
如果大家可以接受这个想法，因为这个其实也是很直觉。

因为我们说 Q function 的意思就是在 state s 采取 action a 的时候，接下来会得到 reward 的期望值，那接下来会得到 reward 的期望值怎么算呢？我们现在在 state s_t ，然后我们采取 action a_t ，然后我们想要知道说，接下来会得到多少 reward，那接下来会发生什么事呢？接下来你会得到 reward r_t ，然后跳到 state $s(t+1)$ ，那在 state s 采取 action a 得到的 reward，其实就是等于接下来得到 reward r_t ，加上从 state $s(t+1)$ 开始，得到接下来所有 reward 的总和。

而从 state $s(t+1)$ 开始，得到接下来所有 reward 的总和，就是 $V^\pi(s_{t+1}^n)$ ，那在 state s_t 采取 action a_t 以后得到的 reward r_t ，就写在这个地方，所以这两项加起来，会等于 Q function。那为什么前面要取期望值呢？因为你在 s_t 采取 action a_t 会得到什么样的 reward，跳到什么样的 state 这件事情，本身是有随机性的，不见得是你的 model 可以控制的，为了要把这随机性考虑进去，前面你必须加上期望值。

但是我们现在把这个期望值拿掉就说他们两个是相等的，把 Q 替换掉。

这样的好处就是，你不需要再 estimate Q 了，你只需要 estimate V 就够了，你只要 estimate 一个 network 就够了。但这样的坏处是什么呢？这样你引入了一个随机的东西， r 现在，它是有随机性的，它是一个 random variable。但是这个 random variable，相较于刚才的 G ，accumulated reward 可能还好，因为它是某一个 step 会得到的 reward，而 G 是所有未来会得到的 reward 的总和， G variance 比较大， r 虽然也有一些 variance，但它的 variance 会比 G 还要小，所以把原来 variance 比较大的 G ，换成现在只有 variance 比较小的 r 这件事情也是合理的。



$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)) \nabla \log p_\theta(a_t^n | s_t^n)$$

那如果你不相信的话，如果你觉得说什么期望值拿掉不相信的话，那我就告诉你原始的 A3C paper，它试了各式各样的方法，最后做出来就是这个最好。当然你可能说，搞不好 estimate Q 跟 V 也都 estimate 很好。那我给你的答案就是做实验的时候，最后结果就是这个最好。所以来大家都用这个。

所以那这整个流程就是这样。

前面这个式子叫做 advantage function，所以这整个方法就叫 Advantage Actor-Critic。

整个流程是这样子的，我们现在先有一个 π ，有个初始的 actor 去跟环境做互动，先收集资料，在每一个 policy gradient 收集资料以后，你就要拿去 update 你的 policy。但是在 actor critic 方法里面，你不是直接拿你的那些数据，去 update 你的 policy。你先拿这些资料去 estimate 出 value function。

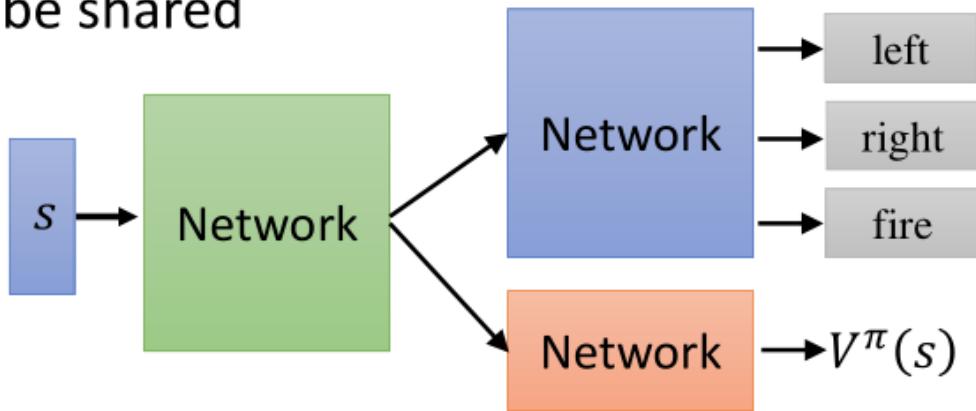
假设你是用别的方法，你有时候可能也需要 estimate Q function，那我们这边是 Advantage Actor-Critic，我们只需要 value function 就好，我们不需要 Q function。你可以用 TD，也可以用 MC，你 estimate 出 value function 以后，接下来，你再 based on value function，套用下面这个式子去 update 你的 π ，然后你有了新的 π 以后，再去跟环境互动，再收集新的资料，去 estimate 你的 value function，然后再用新的 value function，去 update 你的 policy，去 update 你的 actor。

整个 actor-critic 的 algorithm，就是这么运作的。

Tips

implement Actor-Critic 的时候，有两个几乎一定会用的 tip。

The parameters of actor $\pi(s)$ and critic $V^\pi(s)$ can be shared



Use output entropy as regularization for $\pi(s)$

- Larger entropy is preferred \rightarrow exploration

第一个 tip 是，我们现在说，我们其实要 estimate 的 network 有两个，我们只要 estimate V function，而另外一个需要 estimate 的 network，是 policy 的 network，也就是你的 actor。那这两个 network，那个 V 那个 network 它是 input 一个 state，output 一个 scalar。然后 actor 这个 network，它是 input 一个 state，output 就是一个 action 的 distribution。假设你的 action 是 discrete 不是 continuous 的话。如果是 continuous 的话，它也是一样，如果是 continuous 的话，就只是 output 一个 continuous 的 vector。这边是举 discrete 的例子，但是 continuous 的 case，其实也是一样的。

input 一个 state，然后它要决定你现在要 take 那一个 action，那这两个 network，这个 actor 跟你的 critic，跟你的 value function，它们的 input 都是 s ，所以它们前面几个 layer，其实是可以 share 的。尤其是假设你今天是玩 ATARI 游戏，或者是你玩的是那种什么 3D 游戏，那 input 都是 image，那 input 那个 image 都非常复杂，通常你前面都会用一些 CNN 来处理，把那些 image 抽成 high level 的 information。把那个 pixel level 到 high level information 这件事情，其实对 actor 跟 critic 来说可能是可以共享的。

所以通常你会让这个 actor 跟 critic 的前面几个 layer 是 shared，你会让 actor 跟 critic 的前面几个 layer 共享同一组参数，那这一组参数可能是 CNN，先把 input 的 pixel，变成比较 high level 的信息，然后再给 actor 去决定说它要采取什么样的行为，给这个 critic，给 value function，去计算 expected 的 return，也就是 expected reward。

那另外一件事情是，我们一样需要 exploration 的机制。那我们之前在讲 Q learning 的时候呢，我们有讲过 exploration 这件事是很重要的。

那今天在做 Actor-Critic 的时候呢，有一个常见的 exploration 的方法是你会对你的 π 的 output 的这个 distribution，下一个 constrain。

这个 constrain 是希望这个 distribution 的 entropy 不要太小，希望这个 distribution 的 entropy 可以大一点，也就是希望不同的 action，它的被采用的机率平均一点，这样在 testing 的时候，才会多尝试各种不同的 action，才会把这个环境探索的比较好，explore 的比较好，才会得到比较好的结果，这个是 advantage 的 Actor-Critic。

Asynchronous Advantage Actor-Critic (A3C)

那接下来什么东西是 Asynchronous Advantage Actor-Critic 呢？因为 reinforcement learning 它的一个问题，就是它很慢，那怎么增加训练的速度呢？

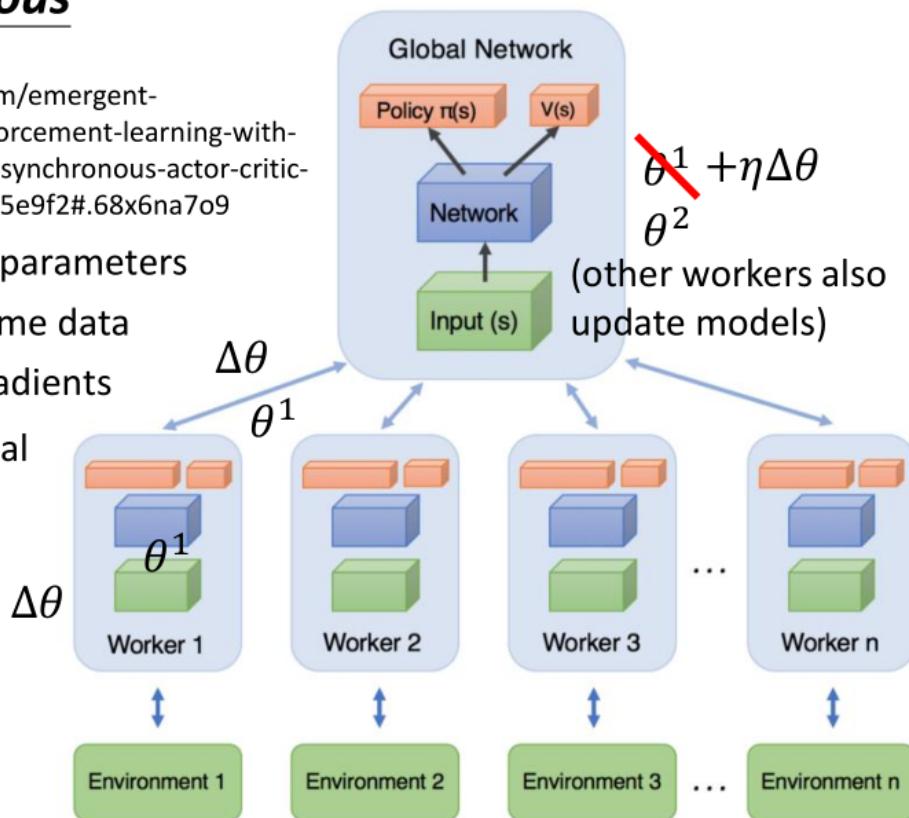
A3C 这个方法的精神，同时开很多个 worker，那每一个 worker 其实就是一个分身，那最后这些分身会把所有的经验，通通集合在一起。

Asynchronous

Source of image:

<https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2#.68x6na7o9>

1. Copy global parameters
2. Sampling some data
3. Compute gradients
4. Update global models



这个 A3C 是怎么运作的呢？首先，当然这个你可能自己实作的时候，你如果没有很多个 CPU，你可能也是不好做。

那 A3C 是这样子，一开始有一个 global network，那我们刚才有讲过说，其实 policy network 跟 value network 是 tie 在一起的，他们的前几个 layer 会被 tie 一起。我们有一个 global network，它们有包含 policy 的部分，有包含 value 的部分，假设他的参数就是 θ_1 。你会开很多个 worker，那每一个 worker 就用一张 CPU 去跑，比如你就开 8 个 worker 那你至少 8 张 CPU。那第一个 worker 呢，就去跟 global network 进去把它的参数 copy 过来，每一个 worker 要工作前就把他的参数 copy 过来，接下来你就去跟环境做互动，那每一个 actor 去跟环境做互动的时候，为了要 collect 到比较 diverse 的 data，所以举例来说如果是走迷宫的话，可能每一个 actor 它出生的位置起始的位置都会不一样，这样他们才能够收集到比较多样性的 data。

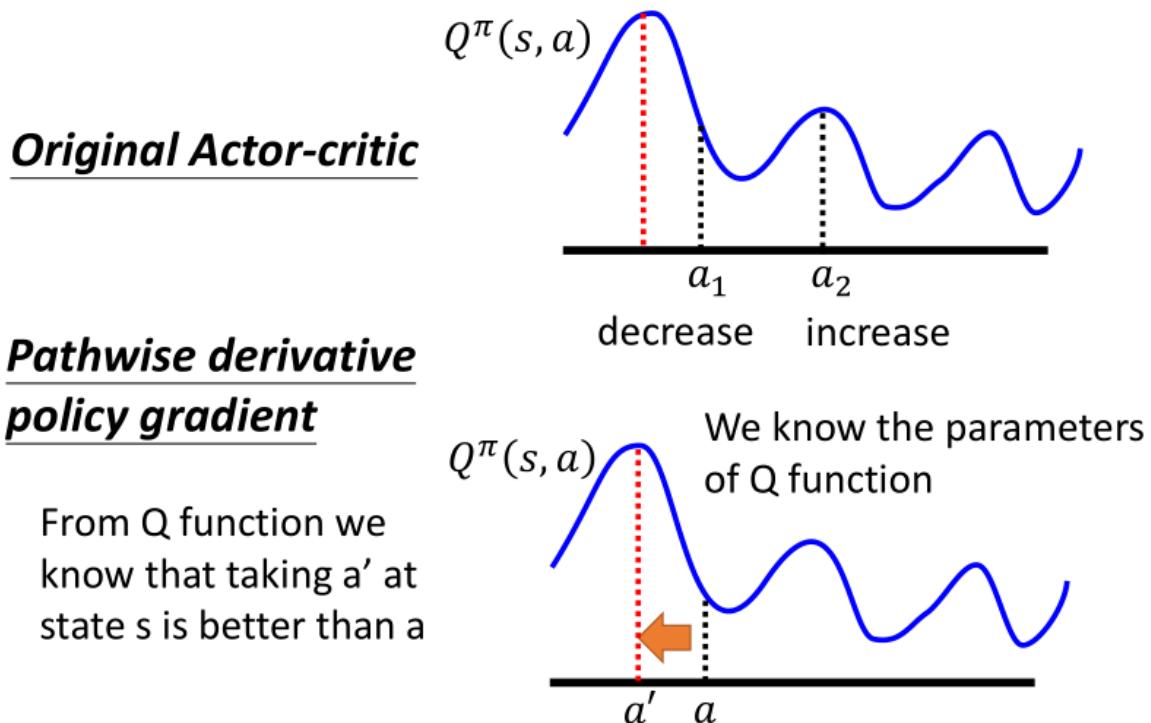
每一个 actor 就自己跟环境做互动，互动完之后，你就会计算出 gradient，那计算出 gradient 以后，你要拿 gradient 去 update global network 的参数。（图中应该是倒三角形）。这个 worker，它算出 gradient 以后，就把 gradient 传回给中央的控制中心，然后中央的控制中心，就会拿这个 gradient，去 update 原来的参数。但是要注意一下，所有的 actor，都是平行跑的，就每一个 actor 就是各做各的，互相之间就不要管彼此，就是各做各的，所以每个人都是去要了一个参数以后，做完它就把它的参数传回去，做完就把参数传回去，所以，当今天第一个 worker 做完，想要把参数传回去的时候，本来它要的参数是 θ_1 ，等它要把 gradient 传回去的时候，可能别人已经把原来的参数覆盖掉，变成 θ_2 了，但是没有关系，就不要在意这种细节，它一样会把这个 gradient 就覆盖过去就是了，这个 Asynchronous actor-critic 就是这么做的。

Pathwise Derivative Policy Gradient

那在讲 A3C 之后，我们要讲另外一个方法叫做，Pathwise Derivative Policy Gradient。

Another Way to use Critic

这个方法很神奇，它可以想成是 Q learning 解 continuous action 的一种特别的方法。它也可以想成是一种特别的 Actor-Critic 的方法。

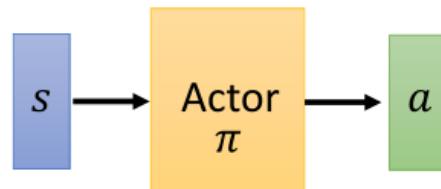


一般的这个 Actor-Critic 里面那个 critic，就是 input state 或 input state 跟 action 的 pair，然后给你一个 value，然后就结束了，所以对 actor 来说它只知道说现在，它做的这个行为，到底是好还是不好，但是，如果是 Pathwise derivative policy gradient 里面，这个 critic 会直接告诉 actor 说，采取什么样的 action，才是好的。critic 会直接引导 actor 做什么样的 action，才是可以得到比较大的 value 的。

那如果今天从这个 Q learning 的观点来看，我们之前说，Q learning 的一个问题是你没有办法在用 Q learning 的时候，考虑 continuous vector，其实也不是完全没办法，就是比较麻烦，比较没有 general solution。

Action a is a *continuous vector*

$$a = \arg \max_a Q(s, a)$$



Actor as the solver of this optimization problem

那今天我们其实可以说，我们怎么解这个 optimization problem 呢？我们用一个 actor 来解这个 optimization 的 problem。所以我们本来在 Q learning 里面，如果是一个 continuous action，我们要解这个 optimization problem，现在这个 optimization problem 由 actor 来解，我们假设 actor 就是一个 solver，这个 solver 它的工作就是，给你 state s ，然后它就去解解解告诉我们说，那一个 action，可以给我们最大的 Q value，这是从另外一个观点来看，Pathwise derivative policy gradient 这件事情。

那这个说法，你有没有觉得非常的熟悉呢？我们在讲 GAN 的时候，不是也讲过一个说法，我们说，我们 learn 一个 discriminator，它是要 evaluate 东西好不好，discriminator 要自己生东西，非常的困难，那怎么办？因为要解一个 Arg Max 的 problem，非常的困难，所以用 generator 来生，所以今天的概念其实是一样的。Q 就是那个 discriminator，要根据这个 discriminator 决定 action 非常困难，怎么办？另外 learn 一个 network，来解这个 optimization problem，这个东西就是 actor。所以今天是从两个不同的观点，其实是同一件事。从两个不同的观点来看，一个观点是说，原来的 Q learning 我们可以加以改

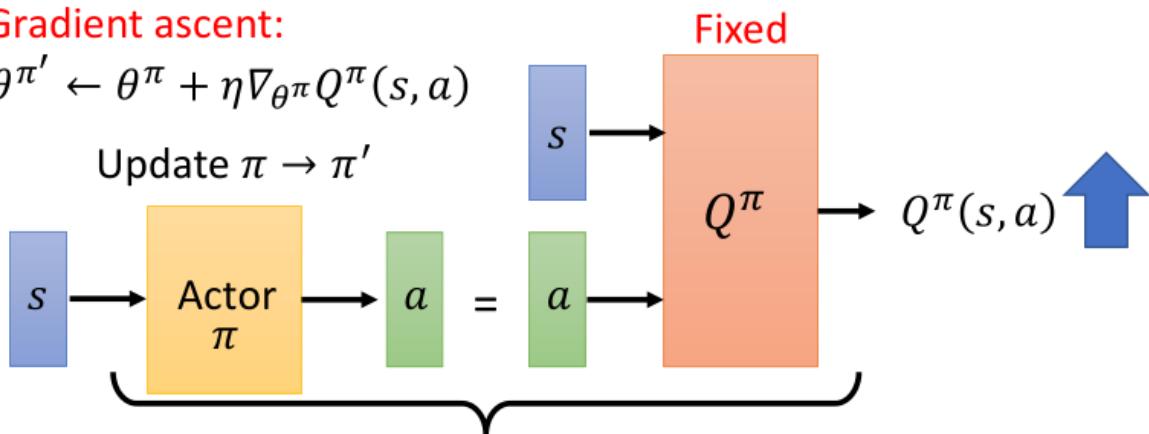
进，怎么改进呢？我们 learn 一个 actor 来决定 action，以解决 Arg Max 不好解的问题。或换句话说，或是另外一个观点是，原来的 actor-critic 的问题是，critic 并没有给 actor 足够的信息，它只告诉它好或不好，没有告诉它说什么样叫好，那现在有新的方法可以直接告诉 actor 说，什么样叫做好。

$$\pi'(s) = \arg \max_a Q^\pi(s, a) \quad \text{a is the output of an actor}$$

Gradient ascent:

$$\theta^{\pi'} \leftarrow \theta^\pi + \eta \nabla_{\theta^\pi} Q^\pi(s, a)$$

Update $\pi \rightarrow \pi'$



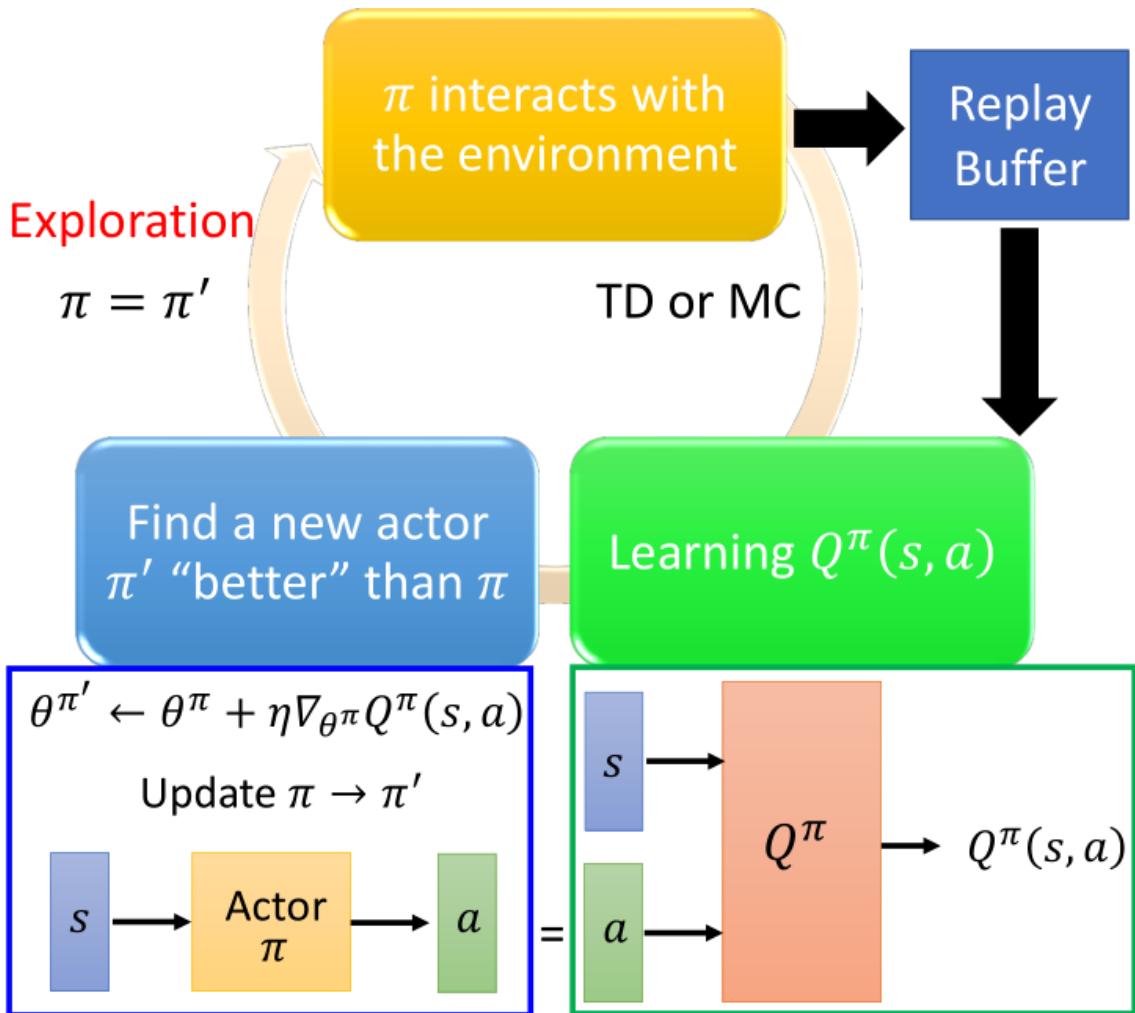
This is a large network

那我们就实际讲一下它的 algorithm，那其实蛮直觉的。

就假设我们 learn 了一个 Q function，假设我们 learn 了一个 Q function，Q function 就是 input s 跟 a，output 就是 $Q(s, a)$ 。

那接下来呢，我们要 learn 一个 actor，这个 actor 的工作是什么，这个 actor 的工作就是，解这个 Arg Max 的 problem，这个 actor 的工作，就是 input 一个 state s，希望可以 output 一个 action a，这个 action a 被丢到 Q function 以后，它可以让 $Q(s, a)$ 的值，越大越好，那实际上在 train 的时候，你其实就是把 Q 跟 actor 接起来，变成一个比较大的 network，Q 是一个 network，input s 跟 a，output 一个 value。那 actor 它在 training 的时候，它要做的事情就是 input s，output a，把 a 丢到 Q 里面，希望 output 的值越大越好。在 train 的时候会把 Q 跟 actor 直接接起来，当作是一个大的 network，然后你会 fix 住 Q 的参数，只去调 actor 的参数，就用 gradient ascent 的方法，去 maximize Q 的 output。

这个东西你有没有觉得很熟悉呢？这就是 conditional GAN，Q 就是 discriminator，但在 reinforcement learning 就是 critic，actor 在 GAN 里面它就是 generator，其实就是同一件事情。



那我们来看一下这个，Pathwise derivative policy gradient 的演算法，一开始你会有一个 actor π ，它去跟环境互动，然后，你可能会要它去 estimate Q value，estimate 完 Q value 以后，你就把 Q value 固定，只去update 那个 actor。

假设这个 Q 估得是很准的，它真的知道说，今天在某一个 state 采取什么样的 action，会真的得到很大的 value，

actor learning 的方向，就是希望 actor 在 given s 的时候 output，采取了 a ，可以让最后 Q function 算出来的 value 越大越好。

你用这个 criteria，去 update 你的 actor π ，然后接下来有新的 π 再去跟环境做互动，然后再 estimate Q，然后再得到新的 π ，去 maximize Q 的 output。

那其实本来在 Q learning 里面，你用得上的技巧，在这边也几乎都用得上，比如说 replay buffer，exploration 等等，这些都用得上。

Q-Learning Algorithm

这个是原来 Q learning 的 algorithm，你有一个 Q function，那你会有另外一个 target 的 Q function，叫做 Q hat。

Initialize Q-function Q , target Q-function $\hat{Q} = Q$

In each episode

- For each time step t
 - Given state s_t , take action a_t based on Q (exploration)
 - Obtain reward r_t , and reach new state s_{t+1}
 - Store (s_t, a_t, r_t, s_{t+1}) into buffer
 - Sample (s_i, a_i, r_i, s_{i+1}) from buffer (usually a batch)
 - Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
 - Update the parameters of Q to make $Q(s_i, a_i)$ close to y (regression)
- Every C steps reset $\hat{Q} = Q$

在每一个 episode 里面，在每一个 episode 的每一个 time step 里面，你会看到一个 state s_t ，你会 take 某一个 action a_t ，那至于 take 哪一个 action，是由 Q function 所决定的。因为解一个 Arg Max 的 problem，如果是 discrete 的话没有问题，你就看说哪一个 a 可以让 Q 的 value 最大，就 take 那一个 action。那你需要加一些 exploration，这样 performance 才会好，你会得到 reward r_t ，跳到新的 state s_{t+1} ，你会把 s_t, a_t, r_t, s_{t+1} 塞到你的 buffer 里面去。你会从你的 buffer 里面 sample 一个 batch 的 data，这个 batch data 里面，可能某一笔是 s_i, a_i, r_i, s_{i+1} 。接下来你会算一个 target，这个 target 叫做 y ， y 是 r_i 加上你拿你的 target Q function 过来，拿你的 Q function 过来，去计算 target 的 Q function，input 那一个 a 的时候，它的 value 会最大，你把这个 target Q function 算出来的 Q value 跟 r 加起来，你就得到你的 target y ，然后接下来你怎么 learn 你的 Q 呢？你就希望你的 Q function，在带 s_i 跟 a_i 的时候，跟 y 越接近越好，这是一个 regression 的 problem。最后，每 t 个 step，你要把 Q hat 用 Q 替代掉。

Pathwise Derivative Policy Gradient

接下来我们把它改成，Pathwise Derivative Policy Gradient。

这边就是只要做四个改变就好。

Initialize Q-function Q , target Q-function $\hat{Q} = Q$, actor π ,
target actor $\hat{\pi} = \pi$

In each episode

- For each time step t

- Given state s_t , take action a_t based on ~~Q~~ π (exploration)
 - Obtain reward r_t , and reach new state s_{t+1}
 - Store (s_t, a_t, r_t, s_{t+1}) into buffer
 - Sample (s_i, a_i, r_i, s_{i+1}) from buffer (usually a batch)
- Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a) \hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}))$
 - Update the parameters of Q to make $Q(s_i, a_i)$ close to y (regression)
- Update the parameters of π to maximize $Q(s_i, \pi(s_i))$
- Every C steps reset $\hat{Q} = Q$
- Every C steps reset $\hat{\pi} = \pi$

第一个改变是，你要把 Q 换成 π 。本来是用 Q 来决定在 state s_t , 产生那一个 action a_t , 现在是直接用 π , 我们不用再解 Arg Max 的 problem 了, 我们直接 learn 了一个 actor, 这个 actor input s_t , 就会告诉我们应该采取哪一个 a_t 。所以本来 input s_t , 采取哪一个 a_t , 是 Q 决定的, 在 Pathwise Derivative Policy Gradient 里面, 我们会直接用 π 来决定, 这是第一个改变。

第二个改变是, 本来这个地方是要计算在 $s(t+1)$, 根据你的 policy, 采取某一个 action a , 会得到多少的 Q value, 那你会采取的 action a , 就是看说哪一个 action a 可以让 Q hat 最大, 你就会采取那个 action a 。这就是你为什么把式子写成这样。

那现在因为我们其实不好解这个 Arg Max 的 problem, 所以 Arg Max problem, 其实现在就是由 policy π 来解了, 所以我们就直接把 $s(t+1)$, 带到 policy π 里面。那你就会知道说, 现在 given $s(t+1)$, 哪一个 action 会给我们最大的 Q value, 那你在这边就会 take 那一个 action。

这边还有另外一件事情要讲一下, 我们原来在 Q function 里面, 我们说, 有两个 Q network, 一个是真正的 Q network, 另外一个是 target Q network, 实际上你在 implement 这个 algorithm 的时候, 你也会有两个 actor, 你会有一个真正要 learn 的 actor π , 你会有一个 target actor $\hat{\pi}$, 这个原理就跟, 为什么要有一个 target Q network 一样, 我们在算 target value 的时候, 我们并不希望它一直的变动, 所以我们会有一个 target 的 actor, 跟一个 target 的 Q function, 那它们平常的参数, 就是固定住的, 这样可以让你的这个 target, 它的 value 不会一直的变化。

所以本来到底是要用哪一个 action a , 你会看说哪一个 action a , 可以让 Q hat 最大。但是现在, 因为哪一个 action a 可以让 Q hat 最大这件事情, 已经被直接用那个 policy 取代掉了, 所以我们要知道哪一个 action a 可以让 Q hat 最大, 就直接把那个 state 带到 $\hat{\pi}$ 里面, 看它得到哪一个 a , 就用那个 a 。那个 a 就是会让 Q hat of (s, a) 的值最大的那个 a 。

其实跟原来的这个 Q learning 也是没什么不同, 只是原来 Max a 的地方, 通通都用 policy 取代掉就是了。

第三个不同就是, 之前只要 learn Q , 现在你多 learn 一个 π , 那 learn π 的时候的方向是什么呢? learn π 的目的, 就是为了 Maximize Q function, 希望你得到的这个 actor, 它可以让你的 Q function output 越来越好, 这个跟 learn GAN 里面的 generator 的概念, 其实是一样的。

第四个 step，就跟原来的 Q function 一样，你要把 target 的 Q network 取代掉，你现在也要把 target policy 取代掉。

Connection with GAN

那其实确实 GAN 跟 Actor-Critic 的方法是非常类似的。

那我们这边就不细讲，你可以去找到一篇 paper 叫 Connecting Generative Adversarial Network and Actor-Critic Method。

Method	GANs	AC
Freezing learning	yes	yes
Label smoothing	yes	no
Historical averaging	yes	no
Minibatch discrimination	yes	no
Batch normalization	yes	yes
Target networks	n/a	yes
Replay buffers	no	yes
Entropy regularization	no	yes
Compatibility	no	yes

David Pfau, Oriol Vinyals, "Connecting Generative Adversarial Networks and Actor-Critic Methods", arXiv preprint, 2016

那知道 GAN 跟 Actor-Critic 非常像有什么帮助呢？一个很大的帮助就是 GAN 跟 Actor-Critic 都是以难 train 而闻名的。所以在文献上就会收集 develop 的各式各样的方法，告诉你说怎么样可以把 GAN train 起来，怎么样可以把 Actor-Critic train 起来，但是因为做 GAN 跟 Actor-Critic 的其实是两群人，所以这篇 paper 里面就列出说在 GAN 上面，有哪些技术是有人做过的，在 Actor-Critic 上面，有哪些技术是有人做过的。

但是也许在 GAN 上面有试过的技术，你可以试着 apply 在 Actor-Critic 上，在 Actor-Critic 上面做过的技术，你可以试着 apply 在 GAN 上面，看看 work 不 work。

这个就是 Actor-Critic 和 GAN 之间的关系，可以带给我们的一个好处，那这个其实就是 Actor-Critic。

Sparse Reward

我们稍微讲一下 sparse reward problem。

sparse reward 是什么意思呢？就是实际上当我们在用 reinforcement learning learn agent 的时候，多数的时候 agent 都是没有办法得到 reward 的。

在没有办法得到 reward 的情况下，对 agent 来说它的训练是非常困难的。假设你今天要训练一个机器手臂，然后桌上有一个螺丝钉跟螺丝起子，那你要训练他用螺丝起子把螺丝钉栓进去，那这个很难，为什么？因为你知道一开始你的 agent，它是什么都不知道的，它唯一能够做不同的 action 的原因，是因为 exploration。举例来说你在做 Q learning 的时候，你会有一些随机性，让它去采取一些过去没有采取过的 action，那你要随机到说它把螺丝起子捡起来，再把螺丝栓进去，然后就会得到 reward 1，这件事情是永远不可能发生的。所以你会发现，不管今天你的 actor 它做了什么事情，它得到 reward 永远都是 0，对它来说不管采取什么样的 action，都是一样糟或者是一样的好，所以它最后什么都不会学到。

所以今天如果你环境中的 reward 非常的 sparse，那这个 reinforcement learning 的问题，就会变得非常的困难。对人类来说，人类很厉害，人类可以在非常 sparse 的 reward 上面去学习，就我们的人生通常多数的时候我们就只是活在那里，都没有得到什么 reward 或者是 penalty，但是人还是可以采取各种各样的行为。所以，一个真正厉害的人工智能，它应该能够在 sparse reward 的情况下，也学到要怎么跟这个环境互动。

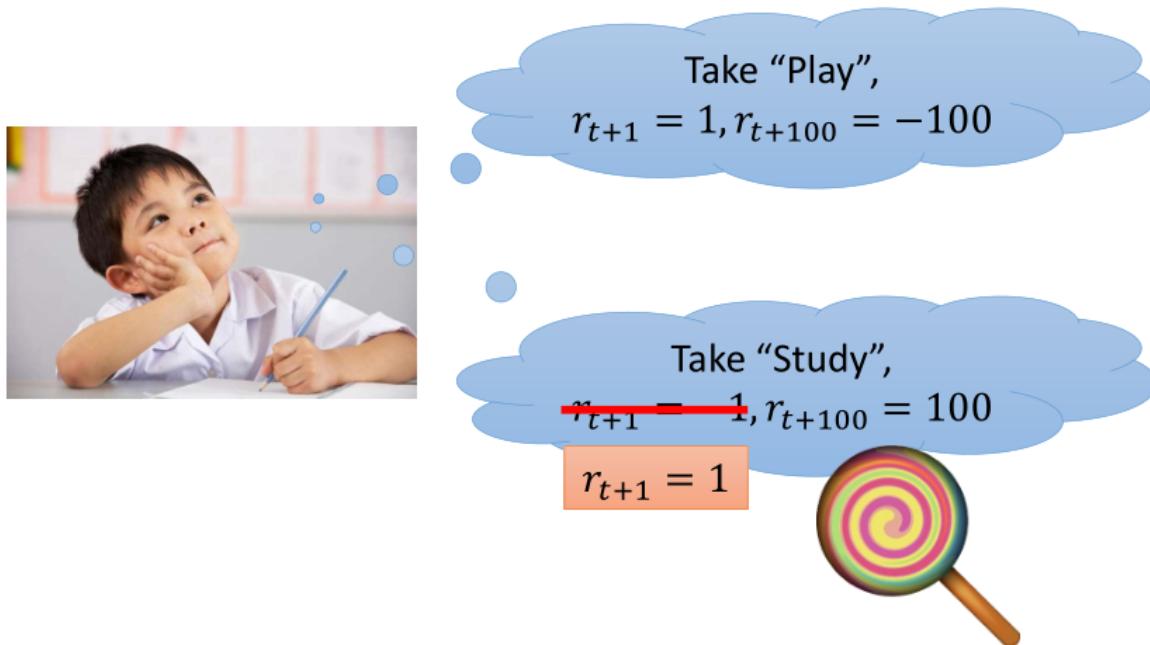
所以，接下来我想要跟大家很快的，非常简单的介绍，就是一些 handle sparse reward 的方法。

Reward Shaping

那怎么解决 sparse reward 的这件事情呢？我们会讲三个方向。

第一个方向叫做 reward shaping。reward shaping 是什么意思呢？

reward shaping 的意思是说，环境有一个固定的 reward，它是真正的 reward，但是我们为了引导 machine，为了引导 agent，让它学出来的结果是我们要的样子，developer 就是我们人类，刻意的去设计了一些 reward，来引导我们的 agent。



举例来说，如果是把小孩当作一个 agent 的话，那一个小孩，他可以 take 两个 actions，一个 action 是他可以出去玩，那他出去玩的话，在下一秒钟他会得到 reward 1，但是他可能在月考的时候，成绩会很差，所以，在 100 个小时之后呢，他会得到 reward -100。他也可以决定他要念书，然后在下一个时间，因为他没有出去玩，所以他觉得很不爽，所以他得到 reward -1。但是在 100 个小时后，他可以得到 reward 100。对一个小孩来说，他可能就会想要 take play，而不是 take study。因为今天我们虽然说，我们计算的是 accumulated reward，但是也许对小孩来说，他的 discount factor 很大这样。所他就不太在意未来的 reward，而且也许因为他是一个小孩，他还没有很多 experience，所以，他的 Q function estimate 是非常不精准的，所以要他去 estimate 很遥远以后，会得到的 accumulated reward，他其实是预测不出来的。

所以怎么办呢？这时候大人就要引导他，怎么引导呢？就骗他说，如果你坐下来念书我就给你吃一个棒棒糖。

所以对他来说，下一个时间点会得到的 reward 就变成是 positive 的，所以他觉得说，也许 take 这个 study 是比 play 好的，虽然实际上这并不是真正的 reward，而是其他人去骗他的 reward，告诉他说你采取这个 action 是好的，所以我给你一个 reward，虽然这个不是环境真正的 reward。

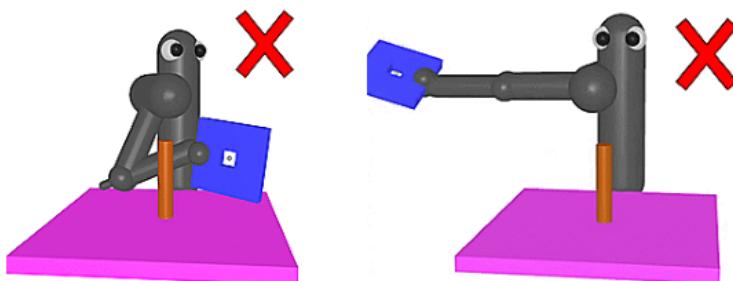
reward shaping 的概念是一样的，简单来说，就是你自己想办法 design 一些 reward，他不是环境真正的 reward，在玩 ATARI 游戏里面，真的 reward 是那个游戏的主机给你的 reward。但是你自己去设一些 reward，好引导你的 machine，做你想要它做的事情。

Reward Shaping

VizDoom

<https://openreview.net/forum?id=Hk3mPK5gg>

Parameters	Description	FlatMap	CIGTrack1
living	Penalize agent who just lives	-0.008 / action	
health_loss	Penalize health decrement	-0.05 / unit	
ammo_loss	Penalize ammunition decrement	-0.04 / unit	
health_pickup	Reward for medkit pickup	0.04 / unit	
ammo_pickup	Reward for ammunition pickup	0.15 / unit	
dist_penalty	Penalize the agent when it stays	-0.03 / action	
dist_reward	Reward the agent when it moves	9e-5 / unit distance	



Get reward,
when closer
Need domain
knowledge

<https://openreview.net/pdf?id=Hk3mPK5gg>

Curiosity

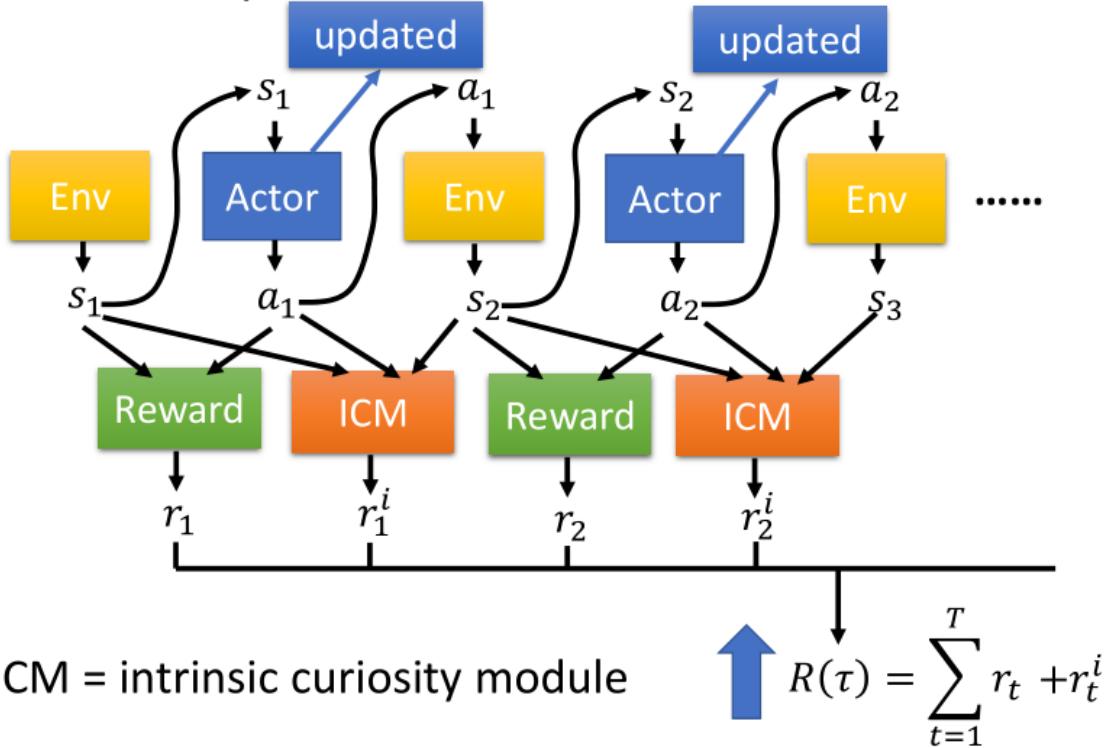
接下来介绍各种你可以自己加进去，In general 看起来是有用的 reward，举例来说，一个技术是，给 machine 加上 curiosity，给它加上好奇心，所以叫 curiosity driven 的 reward。

那这个是我们之前讲 Actor-Critic 的时候看过的图，我们有一个 reward function，它给你某一个 state，给你某一个 action，它就会判断说，在这个 state 采取这个 action 得到多少的 reward。

那我们当然是希望 total reward 越大越好，那在 curiosity driven 的这种技术里面，你会加上一个新的 reward function，这个新的 reward function 叫做 ICM，Intrinsic curiosity module，它就是要给机器加上好奇心。

这个 ICM，它会吃 3 个东西，它会吃 state s_1 ，它会吃 action a_1 跟 state s_2 ，根据 s_1, a_1, a_2 ，它会 output 另外一个 reward，我们这边叫做 $r(i)$ ，那你最后你的 total reward，对 machine 来说，total reward 并不是只有 r 而已，还有 $r(i)$ ，它不是只有把所有的 r 都加起来，它把所有 $r(i)$ 加起来当作 total reward，所以，它在跟环境互动的时候，它不是只希望 r 越大越好，它还同时希望 $r(i)$ 越大越好，它希望从 ICM 的 module 里面，得到的 reward 越大越好。

Curiosity



那这个 ICM 呢，它就代表了一种 curiosity。

那怎么设计这个 ICM 让它变成一种，让它有类似这种好奇心的功能呢？

Intrinsic Curiosity Module

这个是最原始的设计。这个设计是这样，我们说 curiosity module 就是 input 3 个东西。input 现在的 state, input 在这个 state 采取的 action, 然后接下来 input 下一个 state $s(t+1)$, 然后接下来会 output 一个 reward, $r(i)$, 那这个 $r(i)$ 怎么算出来的呢？

在 ICM 里面，你有一个 network，这个 network 会 take $a(t)$ 跟 $s(t)$ ，然后去 output $s(t+1)$ hat，也就是这个 network 做的事情，是根据 $a(t)$ 跟 $s(t)$ ，去 predict 接下来我们会看到的 $s(t+1)$ hat。

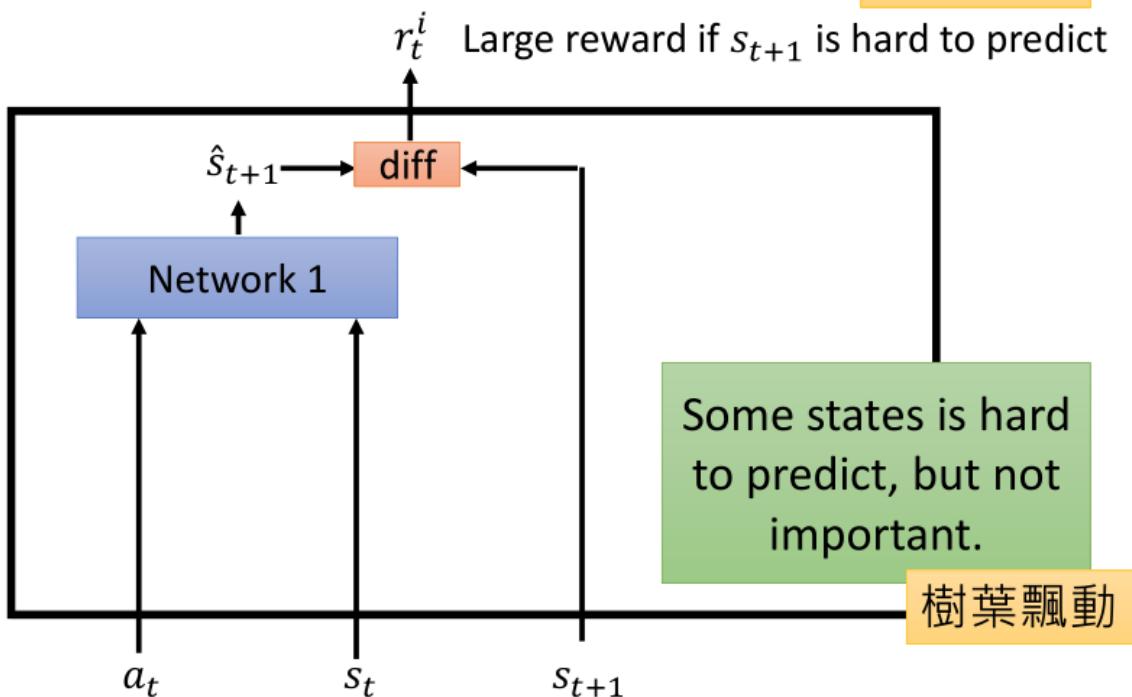
你会根据现在的 state，跟在这个 state 采取的 action，我们有另外一个 network 去预测，接下来会发生什么事。

接下来再看说，machine 自己的预测，这个 network 自己的预测，跟真实的情况像不像，越不像，那越不像那得到的 reward 就越大。

所以今天这个 reward 呢，它的意思是说，如果今天未来的 state，越难被预测的话，那得到的 reward 就越大。这就是鼓励 machine 去冒险，现在采取这个 action，未来会发生什么事，越没有办法预测的话，那这个 action 的 reward 就大。

Intrinsic Curiosity Module

鼓勵冒險



所以，machine 如果有这样子的 ICM，它就会倾向于采取一些风险比较大的 action。它想要去探索未知的世界，想要去看看说，假设某一个 state，是它没有办法预测，假设它没有办法预测未来会发生什么事，它会特别去想要采取那种 state，可以增加 machine exploration 的能力。

那这边这个 network 1，其实是另外 train 出来的。在 training 的时候，你会给它 $a_t, s_t, s(t+1)$ ，然后让这个 network 1 去学说 given a_t, s_t ，怎么 predict $s(t+1)$ hat。

apply 到 agent 互动的时候，这个 ICM module，其实要把它 fix 住。

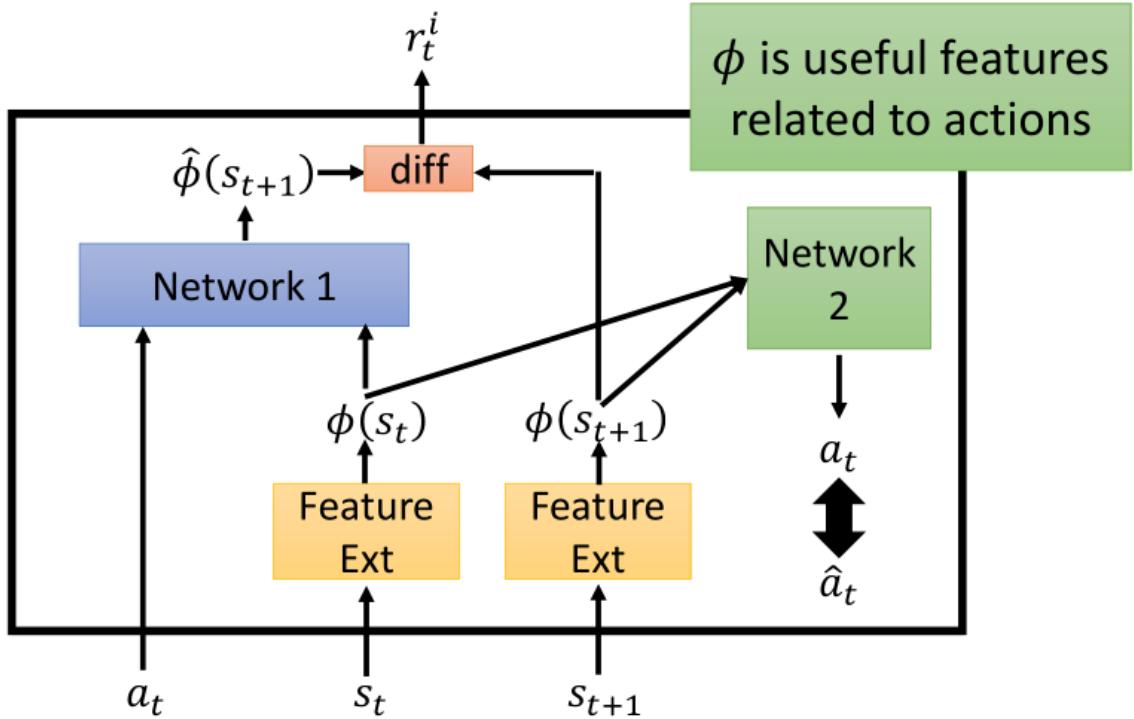
其实，这一整个想法里面，是有一个问题的，这个问题是什么呢？这个问题是，某一些 state，它很难被预测，并不代表它就是好的，它就应该要去被尝试的。

所以，今天光是告诉 machine，鼓励 machine 去冒险是不够的，因为如果光是只有这个 network 的架构，machine 只知道说什么东西它无法预测，如果在某一个 state 采取某一个 action，它无法预测接下来结果，它就会采取那个 action，但并不代表这样的结果一定是好的。

举例来说，可能在某个游戏里面，背景会有树叶飘动，那也许树叶飘动这件事情，是很难被预测的，对 machine 来说它在某一个 state 什么都不做，看着树叶飘动，然后，发现这个树叶飘动是没有办法预测的，接下来它就会一直站在那边，看树叶飘动。

所以说，光是有好奇心是不够的，还要让它知道说，什么事情是真正重要的。那怎么让 machine 真的知道说什么事情是真正重要的，而不是让它只是一直看树叶飘动呢？

Intrinsic Curiosity Module



你要加上另外一个 module，我们要 learn 一个 feature 的 extractor，这个黄色的格子代表 feature extractor，它是 input 一个 state，然后 output 一个 feature vector，代表这个 state。

那我们现在期待的是，这个 feature extractor 可以做的事情是把 state 里面没有意义的东西把它滤掉，比如说风吹草动，白云的飘动，树叶的飘动这种，没有意义的东西直接把它滤掉。假设这个 feature extractor，真的可以把无关紧要的东西，滤掉以后，那我们的 network 1 实际上做的事情是，给它一个 actor，给他一个 state s_1 的 feature representation，让它预测，state $s(t+1)$ 的 feature representation，然接下来我们再看说，这个预测的结果，跟真正的 state $s(t+1)$ 的 feature representation 像不像。越不像，reward 就越大。

接下来的问题就是，怎么 learn 这个 feature extractor 呢？让这个 feature extractor 它可以把无关紧要的事情滤掉呢？这边的 learn 法就是，learn 另外一个 network 2，这个 network 2 它是吃这两个 vector 当做 input，然后接下来它要 predict action a ，然后它希望这个 action a ，跟真正的 action a 越接近越好（这里这个 a 跟 \hat{a} 应该要反过来，预测出来的东西我们用 \hat{a} 来表示，真正的东西没有 \hat{a} ，这样感觉比较对）。

所以这个 network 2，它会 output 一个 action。根据 state s_t 的 feature 跟 state $s(t+1)$ 的 feature，output 从 state s_t ，跳到 state $s(t+1)$ ，要采取哪一个 action，才能够做到。希望这个 action 跟真正的 action，越接近越好。

那加上这个 network 的好处就是，因为这两个东西要拿去预测 action，所以，今天我们抽出来的 feature，就会变成是跟 action，跟预测 action 这件事情是有关的。

所以，假设是一些无聊的东西，是跟 machine 本身采取的 action 无关的东西，风吹草动或是白云飘过去，是 machine 自己要采取的 action 无关的东西，那就会被滤掉，就不会被放在抽出来的 vector representation 里面。

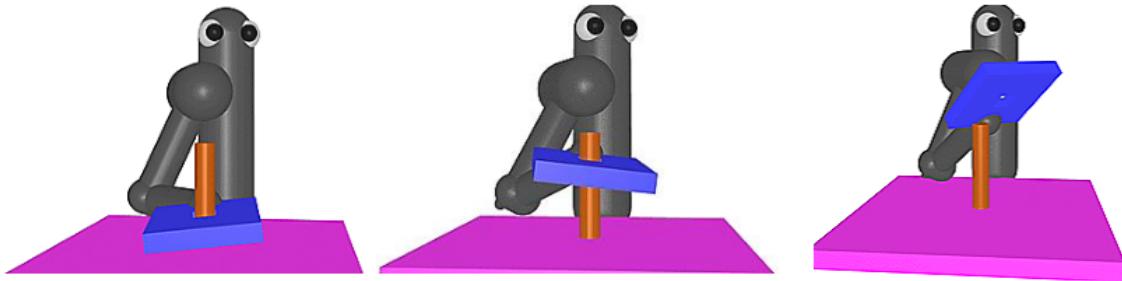
Curriculum Learning

Curriculum learning 不是 reinforcement learning 所独有的概念，那其实在很多 machine learning，尤其是 deep learning 里面，你都会用到 Curriculum learning 的概念。

- Starting from simple training examples, and then becoming harder and harder.

VizDoom

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7
Speed	0.2	0.2	0.4	0.4	0.6	0.8	0.8	1.0
Health	40	40	40	60	60	60	80	100



所谓 Curriculum learning 的意思是说，你为机器的学习做规划，你给他喂 training data 的时候，是有顺序的，那通常都是由简单到难。就好比说假设你今天要交一个小朋友作微积分，他做错就打他一巴掌，可是他永远都不会做对，太难了，你要先教他乘法，然后才教他微积分，打死他，他都学不起来这样，所以很。所以 Curriculum learning 的意思就是在教机器的时候，从简单的题目，教到难的题目，那如果不是 reinforcement learning，一般在 train deep network 的时候，你有时候也会这么做。举例来说，在 train RNN 的时候，已经有很多的文献，都 report 说，你给机器先看短的 sequence，再慢慢给它长的 sequence，通常可以学得比较好。

那用在 reinforcement learning 里面，你就是要帮机器规划一下它的课程，从最简单的到最难的。举例来说，Facebook 那个 VizDoom 的 agent 据说蛮强的，他们在参加机器的 VizDoom 比赛是得第一名的，他们是有为机器规划课程的，先从课程 0 一直上到课程 7。在这个课程里面，那些怪有不同的 speed 跟 health，怪物的速度跟血量是不一样的。所以，在越进阶的课程里面，怪物的速度越快，然后他的血量越多。在 paper 里面也有讲说，如果直接上课程 7，machine 是学不起来的，你就是要从课程 0 一路玩上去，这样 machine 才学得起来。

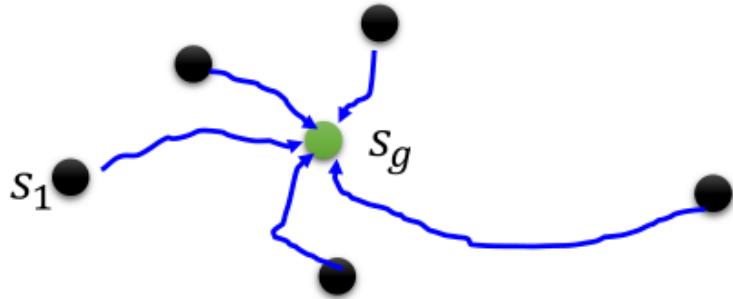
所以，再拿刚才的把蓝色的板子放到柱子上的实验。怎么让机器一直从简单学到难呢？也许一开始你让机器初始的时候，它的板子就已经在柱子上了，这个时候，你要做的事情只有，这个时候，机器要做的事情只有把蓝色的板子压下去，就结束了，这比较简单，它应该很快就学的会。它只有往上跟往下这两个选择，往下就得到 reward 就结束了。

这边是把板子挪高一点。假设它现在学的到，只要板子接近柱子，它就可以把这个板子压下去的话。接下来，你再让它学更 general 的 case，先让一开始，板子离柱子远一点，然后，板子放到柱子上面的时候，它就会知道把板子压下去，这个就是 Curriculum Learning 的概念。

Reverse Curriculum Generation

当然 Curriculum learning 这边有点 ad hoc，就是你需要人当作老师去为机器设计它的课程。

那有一个比较 general 的方法叫做，Reverse Curriculum Generation，你可以用一个比较通用的方法，来帮机器设计课程。这个比较通用的方法是怎么样呢？假设你现在一开始有一个 state sg，这是你的 gold state，也就是最后最理想的结果，如果是拿刚才那个板子和柱子的例子的话，就把板子放到柱子里面，这样子叫做 gold state。你就已经完成了，或者你让机器去抓东西，你训练一个机器手臂抓东西，抓到东西以后叫做 gold state。



Given a goal state s_g .

Sample some states s_1 “close” to s_g

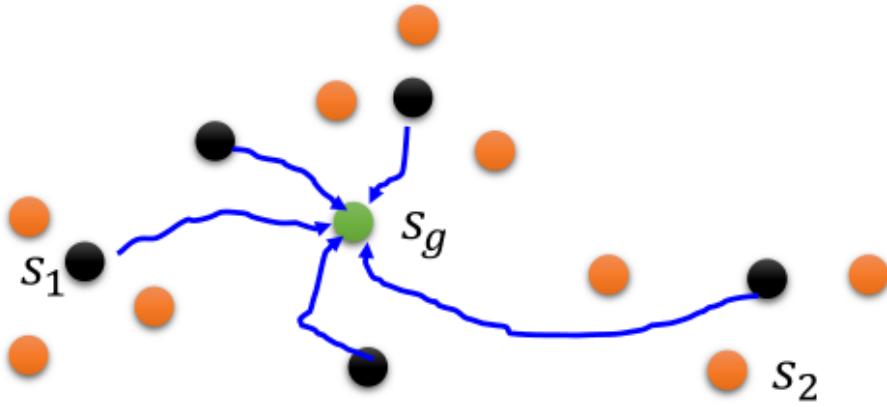
Start from states s_1 , each trajectory has reward $R(s_1)$

那接下来你根据你的 gold state, 去找其他的 state, 这些其他的 state, 跟 gold state 是比较接近的。

举例来说, 假装这些跟 gold state 很近的 state 我们叫做 s_1 , 你的机械手臂还没有抓到东西, 但是, 它离 gold state 很近, 那这个叫做 s_1 。

至于什么叫做近, 这个就麻烦, 就是 case dependent, 你要根据你的 task, 来 design 说怎么从 s_g sample 出 s_1 , 如果是机械手臂的例子, 可能就比较好想, 其他例子可能就比较难想。

接下来呢, 你再从这些 state 1 开始做互动, 看它能不能够达到 gold state s_g 。那每一个 state, 你跟环境做互动的时候, 你都会得到一个 reward R 。接下来, 我们把 reward 特别极端的 case 去掉。reward 特别极端的 case 的意思就是说, 那些 case 它太简单, 或者是太难, 就 reward 如果很大, 代表说这个 case 太简单了, 就不用学了, 因为机器已经会了, 它可以得到很大的 reward。那 reward 如果太小代表这个 case 太难了, 依照机器现在的能力这个课程太难了, 它学不会, 所以就不要学这个, 所以只找一些 reward 适中的 case。那当然什么叫做适中, 这个就是你要调的参数。



Delete s_1 whose reward is too large (already learned) or too small (too difficult at this moment)

Sample s_2 from s_1 , start from s_2

找一些 reward 适中的 case, 接下来, 再根据这些 reward 适中的 case, 再去 sample 出更多的 state, 更多的 state, 就假设你一开始, 你的东西在这里, 你机械手臂在这边, 可以抓的到以后, 接下来, 就再离远一点, 看看能不能够抓得到, 又抓的到以后, 再离远一点, 看看能不能够抓得到。

因为它说从 gold state 去反推, 就是说你原来的目标是长这个样子, 我们从我们的目标去反推, 所以这个叫做 reverse。

这个方法很直觉, 但是, 它是一个有用的方法就是了, 特别叫做 Reverse Curriculum learning。

Hierarchical Reinforcement learning

那刚才讲的是 Curriculum learning, 就是你要为机器规划它学习的顺序。

那最后一个要跟大家讲的 tip, 叫做 Hierarchical Reinforcement learning, 有阶层式的 reinforcement learning。

所谓阶层式的 Reinforcement learning 是说, 我们有好几个 agent, 然后, 有一些 agent 负责比较 high level 的东西, 它负责订目标, 然后它订完目标以后, 再分配给其他的 agent, 去把它执行完成。

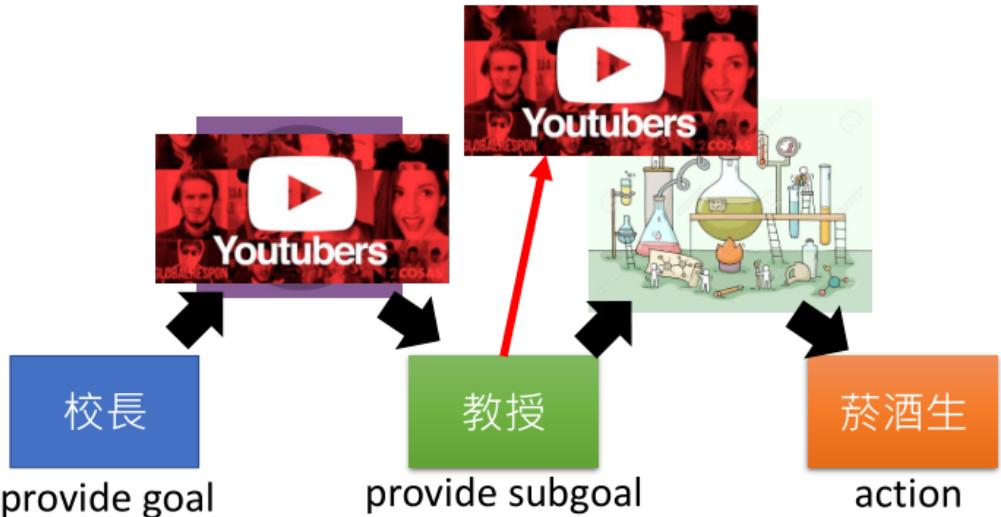
那这样的想法其实也是很合理的, 因为我们知道说, 我们人在一生之中, 我们并不是时时刻刻都在做决定。

举例来说, 假设你想要写一篇 paper, 那你会先想说我要写一篇 paper 的时候, 我要做那些 process, 就是说我先想个梗这样子。然后想完梗以后, 你还要跑个实验, 跑完实验以后, 你还要写, 写完以后呢, 你还要去发表这样子, 那每一个动作下面又还会再细分。比如说, 怎么跑实验呢? 你要先 collect data, collect 完 data 以后, 你要再 label, 你要弄一个 network, 然后又 train 不起来, 要 train 很多次, 然后重新 design network 架构好几次, 最后才把 network train 起来。

所以, 我们要完成一个很大的 task 的时候, 我们并不是从非常底层的那些 action 开始想起, 我们其实是有有个 plan, 我们先想说, 如果要完成这个最大的任务, 那接下来要拆解成哪些小任务, 每一个小任务要再怎么拆解成, 小小的任务, 这个是我们人类做事情的方法。

举例来说, 叫你直接写一本书可能很困难, 但叫你先把一本书拆成好几个章节, 每个章节拆成好几段, 每一段又拆成好几个句子, 每一个句子又拆成好几个词汇, 这样你可能就比较写得出来。这个就是阶层式的 Reinforcement learning 的概念。

Hierarchical RL



- If lower agent cannot achieve the goal, the upper agent would get penalty.
- If an agent get to the wrong goal, assume the original goal is the wrong one.

<https://arxiv.org/abs/1805.08180>

这边是随便举一个好像可能不恰当的例子，就是假设校长跟教授跟研究生通通都是 agent。那今天假设我们的 reward 就是，只要进入百大就可以得到 reward 这样，假设进入百大的话，校长就要提出愿景，告诉其他的 agent 说，现在你要达到什么样的目标，那校长的愿景可能就是说，教授每年都要发三篇期刊。然后接下来，这些 agent 都是有阶层式的，所以上面的 agent，他的 action 他所提出的动作，他不真的做事，他的动作就是提出愿景这样，那他把他的愿景传给下一层的 agent。

下一层的 agent 就把这个愿景吃下去，如果他下面还有其他人的话，它就会提出新的愿景，比如说，校长要教授发期刊，但是其实教授自己也是不做实验的，所以，教授也只能叫下面的苦命研究生做实验，所以教授就提出愿景，就做出实验的规划，然后研究生才是真的去执行这个实验的人，然后，真的把实验做出来，最后大家就可以得到 reward。

这个例子其实有点差。因为真实的情况是，校长其实是不会管这些事情的，校长并不会管教授有没有发期刊，而且发期刊跟进入百大其实关系也不大，而且更退一步说好了，我们现在是没有校长的。所以，现在显然这个就不是指台大，所以，这是一个虚构的故事，我随便乱编的，没有很恰当。

那现在是这样子的，在 learn 的时候，其实每一个 agent 都会 learn。他们的整体的目标，就是要达成，就是要达到最后的 reward。那前面的这些 agent，他提出来的 actions，就是愿景。你如果是玩游戏的话，他提出来的就是，我现在想要产生这样的游戏画面，然后，下面的能不能够做到这件事情，上面的人就是提出愿景。但是，假设他提出来的愿景，是下面的 agent 达不到的，那就会被讨厌，举例来说，教授对研究生，都一直逼迫研究生做一些很困难的实验，研究生都做不出来的话，研究生就会跑掉，所以他就会得到一个 penalty。

如果今天下层的 agent，他没有办法达到上层 agent 所提出来的 goal 的话，上层的 agent 就会被讨厌，它就会得到一个 negative reward，所以他要避免提出那些愿景是，底下的 agent 所做不到的。那每一个 agent 他都是吃，上层的 agent 所提出来的愿景，当作输入，然后决定他自己要产生什么输出，决定他自己要产生什么输出。但是你知道说，就算你看到，上面的的愿景说，叫你做这一件事情，你最后也不见得，做得到这一件事情。

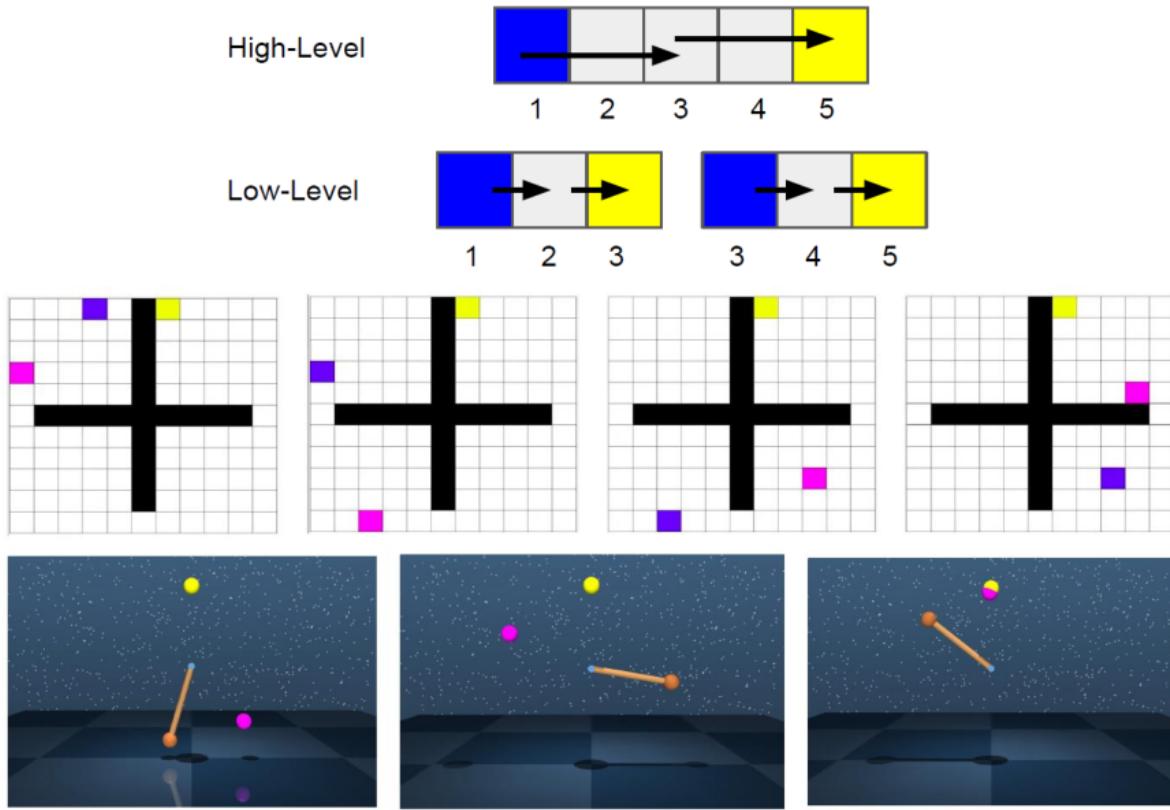
假设，本来教授目标是要写期刊，但是不知道怎么回事，他就要变成一个 YouTuber。

这个 paper 里面的 solution，我觉得非常有趣，给大家做一个参考，这其实本来的目标是要写期刊，但却变成 YouTuber，那怎么办呢？把原来的愿景改成变成 YouTuber，就结束了。在 paper 里面就是这么做的，为什么这么做呢？因为虽然本来的愿景是要写期刊，但是后来变成 YouTuber。

难道这些动作都浪费了吗？不是，这些动作是没有被浪费的。

我们就假设说，本来的愿景，其实就是要成为 YouTuber，那你就知道说，成为 YouTuber 要怎做了。

这个细节我们就不讲了，你自己去研究一下 paper，这个是阶层式 RL，可以做得起来的 tip。



那这个是真实的例子，给大家参考一下，实际上呢，这里面就做了一些比较简单的游戏，这个是走迷宫，蓝色是 agent，蓝色的 agent 要走走走，走到黄色的目标。

这边也是，这个单摆要碰到黄色的球。那愿景是什么呢？在这个 task 里面，它只有两个 agent，只有下面的一个，最底层的 agent 负责执行，决定说要怎么走，还有一个上层的 agent，负责提出愿景。虽然实际上你 general 而言可以用很多层，但是 paper 我看那个实验，只有两层。

那今天这个例子是说，粉红色的这个点，代表的就是愿景，上面这个 agent，它告诉蓝色的这个 agent 说，你现在的第一个目标是先走到这个地方。

蓝色的 agent 走到以后，再说你的新的目标是走到这里，蓝色的 agent 再走到以后，新的目标在这里，接下来又跑到这边，然后，最后希望蓝色的 agent 就可以走到黄色的这个位置。

这边也是一样，就是，粉红色的这个点，代表的是目标，代表的是上层的 agent 所提出来的愿景。所以，这个 agent 先摆到这边，接下来，新的愿景又跑到这边，所以它又摆到这里，然后，新的愿景又跑到上面，然后又摆到上面，最后就走到黄色的位置了。

这个就是 hierarchical 的 Reinforcement Learning。

Imitation Learning

Imitation learning 就更进一步讨论的问题是，假设我们今天连 reward 都没有，那要怎么办才好呢？

Introduction

这个 Imitation learning 又叫做 learning by demonstration, 或者叫做 apprenticeship learning。apprenticeship 是学徒的意思。

Imitation Learning

- Also known as learning by demonstration, apprenticeship learning

An expert demonstrates how to solve the task

- Machine can also interact with the environment, but cannot explicitly obtain reward.
- It is hard to define reward in some tasks.
- Hand-crafted rewards can lead to uncontrolled behavior

Two approaches:

- Behavior Cloning
- Inverse Reinforcement Learning (inverse optimal control)

那在这 Imitation learning 里面，你有一些 expert 的 demonstration, machine 也可以跟环境互动，但它没有办法从环境里面得到任何的 reward，他只能够看着 expert 的 demonstration，来学习什么是好，什么是不好。

那你说为什么有时候，我们没有办法从环境得到 reward。其实，多数的情况，我们都没有办法，真的从环境里面得到非常明确的 reward。

如果今天是棋类游戏，或者是电玩，你有非常明确的 reward，但是其实多数的任务，都是没有 reward 的。举例来说，虽然说自驾车，我们都知道撞死人不好，但是，撞死人应该扣多少分数，这个你没有办法订出来，撞死人的分数，跟撞死一个动物的分数显然是不一样的，但你也不知道要怎么订，这个问题很难，你根本不知道要怎么订 reward。

或是 chat bot 也是一样，今天机器跟人聊天，聊得怎么样算是好，聊得怎么样算是不好，你也无法决定，所以很多 task，你是根本就没有办法订出 reward 的。

虽然没有办法订出 reward，但是收集 expert 的 demonstration 是可能可以做到的，举例来说，在自驾车里面，虽然，你没有办法订出自驾车的 reward，但收集很多人类开车的纪录，这件事情是可行的。

在 chat bot 里面，你可能没有办法收集到太多，你可能没有办法真的定义什么叫做好的对话，什么叫做不好的对话，但是，收集很多人的对话当作范例，这一件事情，也是可行的。

所以，今天 Imitation learning，其实他的实用性非常高，假设，你今天有一个状况是，你不知道该怎么定义 reward，但是你可以收集到 expert 的 demonstration，你可以收集到一些范例的话，你可以收集到一些很厉害的 agent，比如说人跟环境实际上的互动的话，那你就可以考虑 Imitation learning 这个技术。

那在 Imitation learning 里面，我们介绍两个方法，第一个叫做 Behavior Cloning，第二个叫做 Inverse Reinforcement Learning，或者又叫做 Inverse Optimal Control。

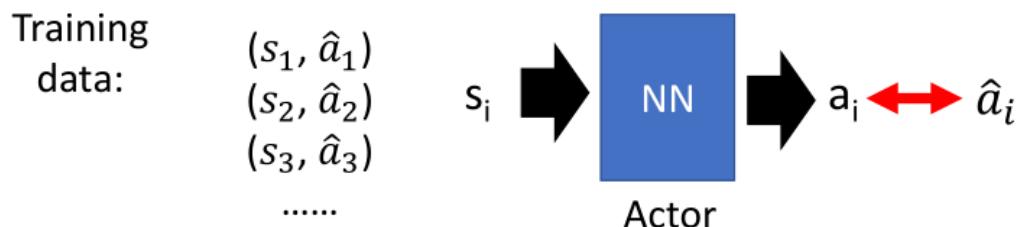
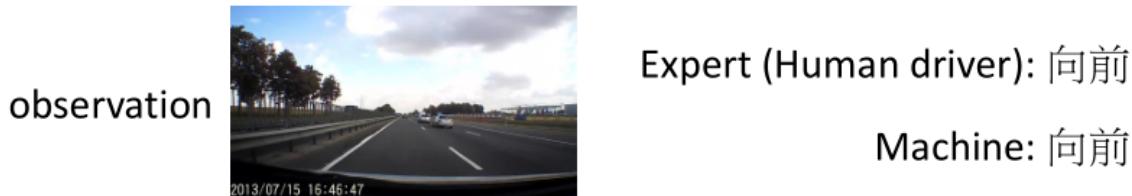
Behavior Cloning

我们先来讲 Behavior Cloning，其实 Behavior Cloning，跟 Supervised learning 是一模一样的，举例来说，我们以自驾车为例。

Behavior Cloning

Yes, this is
supervised learning.

- Self-driving cars as example



今天，你可以收集到人开自驾车的所有数据，比如说，人类的驾驶跟收集人的行车记录器，看到这样子的 observation 的时候，人会决定向前，机器就采取跟人一样的行为，也采取向前，也踩个油门就结束了，这个就叫做 Behavior Cloning。expert 做什么，机器就做一模一样的事。

那怎么让机器学会跟 expert 一模一样的行为呢？就把它当作一个 Supervised learning 的问题，你去收集很多自驾车，你去收集很多行车纪录器，然后再收集人在那个情境下会采取什么样的行为，你知道说人在 state s_1 会采取 action a_1 ，人在 state s_2 会采取 action a_2 ，人在 state s_3 会采取 action a_3 。

接下来，你就 learn 一个 network，这个 network 就是你的 actor，他 input s_i 的时候，你就希望他的 output 是 a_i ，就这样结束了。他就是一个非常单纯的 Supervised learning 的 problem。

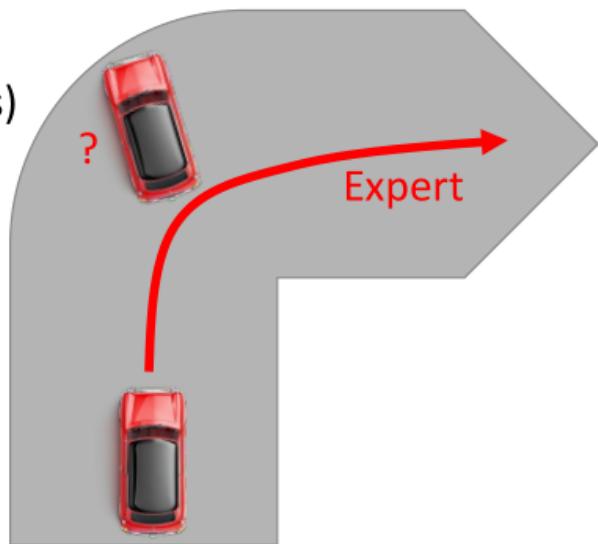
Problem

Problem

Expert only samples
limited observation (states)

Let the expert in the
states seem by
machine

Dataset Aggregation



Behavior Cloning 虽然非常简单，但是他的问题是，今天如果你只收集 expert 的资料，你可能看过的 observation 会是非常 limited，举例来说，假设你要 learn 一部自驾车，自驾车就是要过这个弯道。那如果是 expert 的话，你找人来，不管找多少人来，他就是把车，顺着这个红线就开过去了。

但是，今天假设你的 agent 很笨，他今天开着开着，不知道怎么回事，就开到撞墙了，他永远不知道撞墙这种状况要怎么处理。为什么？因为 training data 里面从来没有撞过墙，所以他根本就不知道撞墙这一种 case，要怎么处理。

或是打电玩也是一样，让机器让人去玩 Mario，那可能 expert 非常强，他从来不会跳不上水管，所以，机器根本不知道跳不上水管时要怎么处理，人从来不会跳不上水管，但是机器今天如果跳不上水管时，就不知道要怎么处理。

Dataset Aggregation

所以，今天光是做 Behavior Cloning 是不够的，只观察 expert 的行为是不够的，需要一个招数，这个招数叫作 Data aggregation。

我们会希望收集更多样性的 data，而不是只有收集 expert 所看到的 observation，我们会希望能够收集 expert 在各种极端的情况下，他会采取什么样的行为。

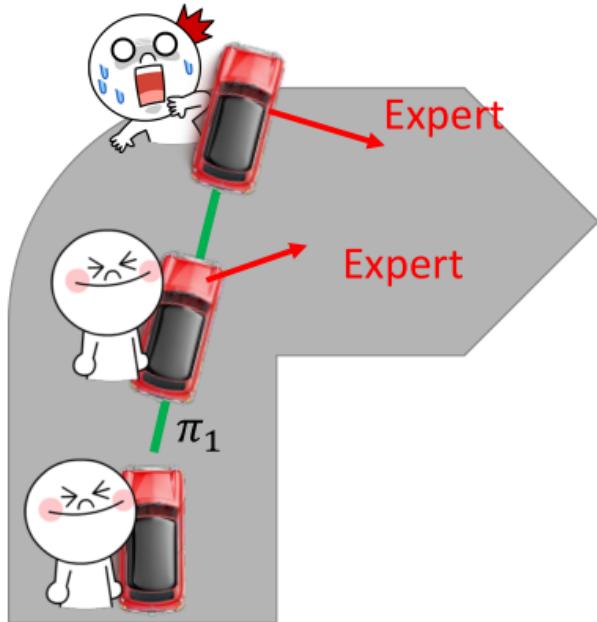
Dataset Aggregation

Get actor π_1 by behavior cloning

Using π_1 to interact with the environment

Ask the expert to label the observation of π_1

Using new data to train π_2



如果以自驾车为例的话，那就是这样，假设一开始，你的 actor 叫作 π_1 。

然后接下来，你让 π_1 ，真的去开这个车，车上坐了一个 expert，这个 expert 会不断的告诉，如果今天在这个情境里面，我会怎么样开。所以，今天 π_1 ，machine 自己开自己的，但是 expert 会不断地表示他的想法，比如说，在这个时候，expert 可能说，那就往前走，这个时候，expert 可能就会说往右转。

但是， π_1 是不管 expert 的指令的，所以，他会继续去撞墙。expert 虽然说要一直往右转，但是不管他怎么下指令都是没有用的， π_1 会自己做自己的事情。

因为我们要做的纪录的是说，今天 expert，在 π_1 看到这种 observation 的情况下，他会做什么样的反应。

那这个方法显然是有一些问题的，因为每次你开一次自驾车，都会牺牲一个人。

那你用这个方法，你牺牲一个 expert 以后，你就会得到说，人类在这样子的 state 下，在快要撞墙的时候，会采取什么样的反应，再把这个 data 拿去 train 新的 π_2 。这个 process 就反复继续下去，这个方法就叫做 Data aggregation。

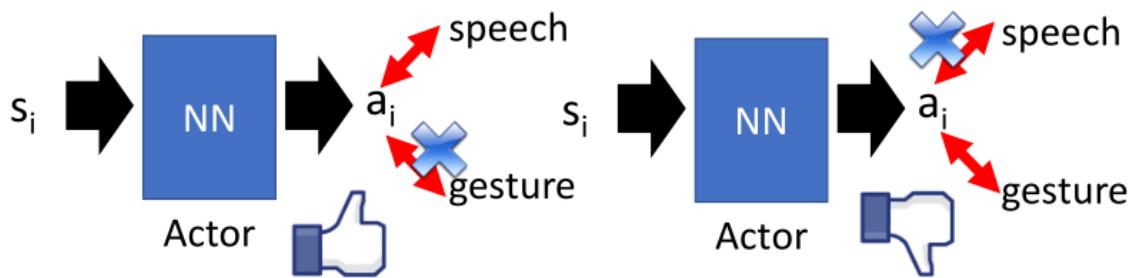
The agent will copy every behavior, even irrelevant actions.

那 Behavior Cloning 这件事情，会有什么样的 issue？还有一个 issue 是说，今天机器会完全 copy expert 的行为。不管今天 expert 的行为，有没有道理，就算没有道理，没有什么用的，这是 expert 本身的习惯，机器也会硬把它记下来。

机器就是你教他什么，他就硬学起来，不管那个东西到底是不是值得的学的。

那如果今天机器确实可以记住，所有 expert 的行为，那也许还好，为什么呢？因为如果 expert 这么做，有些行为是多余的，但是没有问题，在机器假设他的行为，可以完全仿造 expert 行为，那也就算了，那他是跟 expert 一样的好，只是做一些多余的事。

- Major problem: if machine has limited capacity, it may choose the wrong behavior to copy.



- Some behavior must copy, but some can be ignored.
 - Supervised learning takes all errors equally

但是问题就是，他毕竟是一个 machine，他是一个 network，network 的 capacity 是有限的，我们知道说，今天就算给 network training data，他在 training data 上得到的正确率，往往也不是 100，他有些事情，他是学不起来。这个时候，什么该学，什么不该学，就变得很重要。

举例来说，在学习中文的时候，你看到你的老师，他有语音，他也有行为，他也有知识，但是今天其实只有语音部分是重要的，知识的部分是不重要的，也许 machine 他只能够学一件事，也许他就只学到了语音，那没有问题。如果他今天只学到了手势，那这样子就有问题了。

所以，今天让机器学习什么东西是需要 copy，什么东西是不需要copy，这件事情是重要的，而单纯的 Behavior Cloning，其实就没有把这件事情学进来，因为机器唯一做的事情只是复制 expert 所有的行为而已，他并不知道哪些行为是重要，是对接下来有影响的，哪些行为是不重要的，接下来是没有影响的。

Mismatch

那 Behavior Cloning 还有什么样的问题呢？在做 Behavior Cloning 的时候，这个你的 training data 跟 testing data，其实是 mismatch 的，我们刚才其实是有讲到这个样子的 issue，那我们可以用这个 Data aggregation 的方法，来稍微解决这个问题。

那这样子的问题到底是什么样的意思呢？这样的问题是，我们在 training 跟 testing 的时候，我们的 data distribution 其实是不一样，因为我们知道在 Reinforcement learning 里面，有一个特色是你的 action 会影响到接下来所看到的 state，我们是先有 state s1，然后再看到 action a1，action a1 其实会决定接下来你看到什么样的 state s2。

所以在 Reinforcement learning 里面，一个很重要的特征就是你采取的 action 会影响你接下来所看到的 state。

Mismatch



- In supervised learning, we expect training and testing data have the same distribution.
- In behavior cloning:
 - Training: $(s, a) \sim \hat{\pi}$ (expert)
 - Action a taken by actor influences the distribution of s
 - Testing: $(s', a') \sim \pi^*$ (actor cloning expert)
 - If $\hat{\pi} = \pi^*$, (s, a) and (s', a') from the same distribution
 - If $\hat{\pi}$ and π^* have difference, the distribution of s and s' can be very different.

那今天如果我们做了 Behavior Cloning 的话，做 Behavior Cloning 的时候，我们只能够观察到 expert 的一堆 state 跟 action 的 pair。

然后，我们今天希望说我们可以 learn 一个 policy，假设叫做 π^* 好了，我们希望这一个 π^* 跟 $\hat{\pi}$ 越接近越好，如果 π^* 确实可以跟 $\hat{\pi}$ 一模一样的话，那这个时候，你 training 的时候看到的 state，跟 testing 的时候所看到的 state 会是一样。

因为虽然 action 会影响我们看到的 state，假设两个 policy 都一模一样，在同一个 state 都会采取同样的 action，那你接下来所看到的 state 都会是一样。但是问题就是，你很难让你的 learn 出来的 π ，跟 expert 的 π 一模一样，expert 是一个人，network 要跟人一模一样感觉很难。

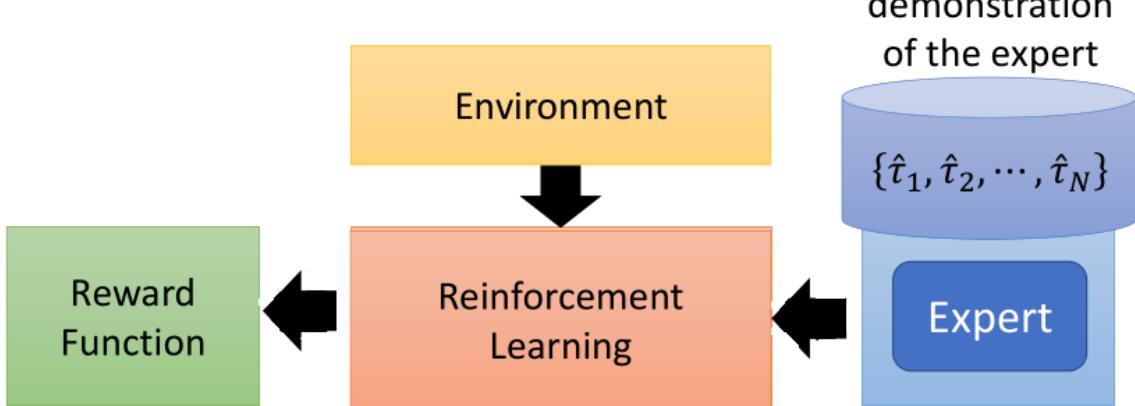
今天你的 π^* 如果跟 $\hat{\pi}$ 有一点误差，这个误差也许在一般 Supervised learning problem 里面，每一个 example 都是 independent 的，也许还好。但是，今天假设 Reinforcement learning 的 problem，你可能在某个地方，也许你的 machine 没有办法完全复制 expert 的行为，它只差了一点点，也许最后得到的结果，就会差很多这样。

所以，今天这个 Behavior Cloning 的方法，并不能够完全解决 Imitation learning 这件事情。

Inverse Reinforcement Learning (IRL)

所以接下来，就有另外一个比较好的做法，叫做 Inverse Reinforcement Learning。

为什么叫 Inverse Reinforcement Learning？因为原来的 Reinforcement Learning 里面，也就是有一个环境，跟你互动的环境，然后你有一个 reward function，然后根据环境跟 reward function，透过 Reinforcement Learning 这个技术，你会找到一个 actor，你会 learn 出一个 optimal actor。



- Using the reward function to find the *optimal actor*.
- Modeling reward can be easier. Simple reward function can lead to complex policy.

但是 Inverse Reinforce Learning 刚好是相反的，你今天没有 reward function，你只有一堆 expert 的 demonstration，但是你还是有环境的，IRL 的做法是说，假设我们现在有一堆 expert 的 demonstration，我们用这个 \hat{t} 来，代表 expert 的 demonstration。

如果今天是在玩电玩的话，每一个 \hat{t} 就是一个很会玩电玩的人，他玩一场游戏的纪录，如果是自驾车的话，就是人开自驾车的纪录，如果是用人开车的纪录，这一边就是 expert 的 demonstration，每一个 \hat{t} 是一个 trajectory，把所有 trajectory expert demonstration 收集起来，然后使用 Inverse Reinforcement Learning 这个技术。

使用 Inverse Reinforcement Learning 技术的时候，机器是可以跟环境互动的，但是他得不到 reward，他的 reward 必须要从 expert 那边推论出来。

现在有了环境，有了 expert demonstration 以后，去反推出 reward function 长什么样子。

之前 Reinforcement learning 是由 reward function，反推出什么样的 actor 是最好的。

Inverse Reinforcement Learning 是反过来，我们有 expert 的 demonstration，我们相信他是不错的，然后去反推，expert 既然做这样的行为，那实际的 reward function 到底长什么样子。我就反推说，expert 是因为什么样的 reward function，才会采取这些行为。你今天有了 reward function 以后，接下来，你就可套用一般的，Reinforcement learning 的方法，去找出 optimal actor，所以 Inverse Reinforcement Learning 里面是先找出 reward function。找出 reward function 以后，再去实际上用 Reinforcement Learning，找出 optimal actor。

有人可能就会问说，把 Reinforcement Learning，把这个 reward function learn 出来，到底相较于原来的 Reinforcement Learning 有什么样好处？

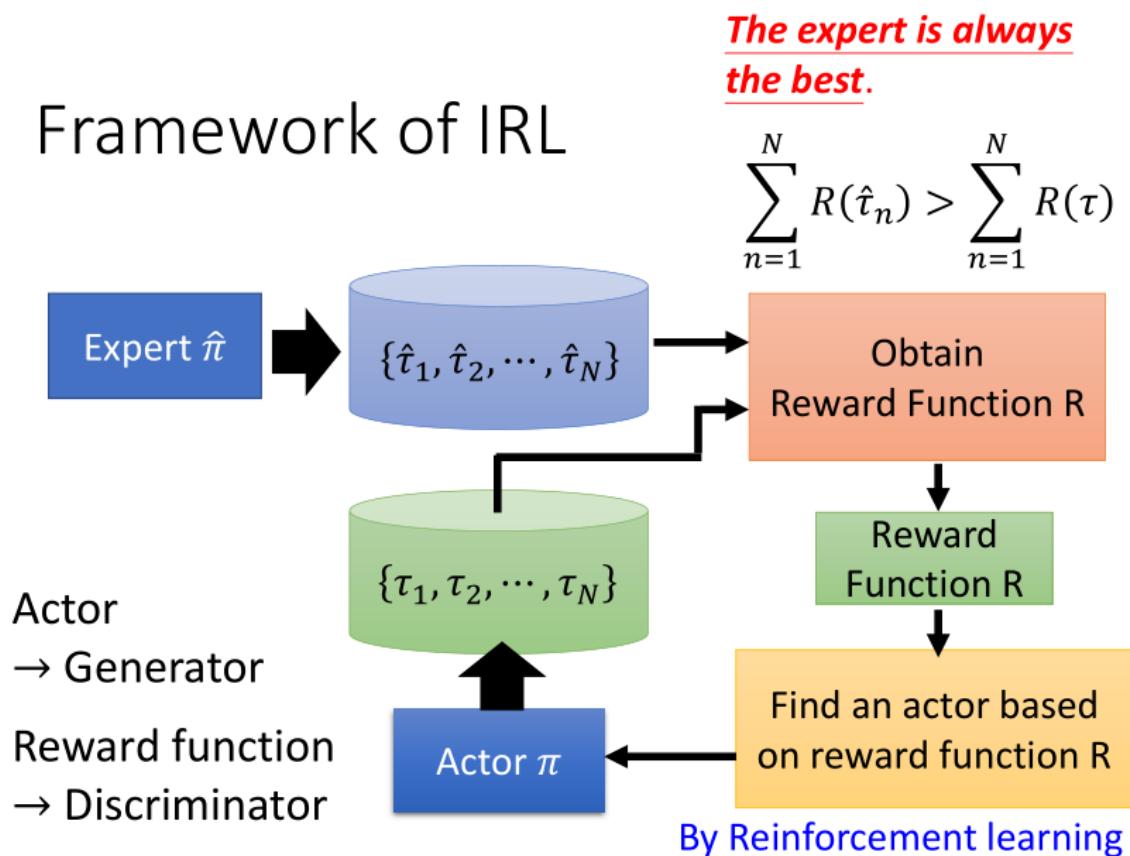
一个可能的好处是，也许 reward function 是比较简单的。虽然这个 actor，这个 expert 他的行为非常复杂，也许简单的 reward function，就可以导致非常复杂的行为。一个例子就是，也许人类本身的 reward function 就只有活着这样，每多活一秒，你就加一分，但是，人类有非常复杂的行为，但是这些复杂的行为，都只是围绕着，要从这个 reward function 里面得到分数而已。有时候很简单的 reward function，也许可以推导出非常复杂的行为。

Framework of IRL

那 Inverse Reinforcement Learning，实际上是怎么做的呢？首先，我们有一个 expert，我们叫做 $\hat{\pi}$ ，这个 expert 去跟环境互动，给我们很多 $\hat{\tau}_1$ 到 $\hat{\tau}_n$ ，如果是玩游戏的话，就让某一个电玩高手，去玩 n 场游戏，把 n 场游戏的 state 跟 action 的 sequence，通通都记录下来。

接下来，你有一个 actor，一开始 actor 很烂，他叫做 π ，这个 actor 他也去跟环境互动，他也去玩了 n 场游戏，他也有 n 场游戏的纪录。

接下来，我们要反推出 reward function。



怎么推出 reward function 呢？这一边的原则就是，expert 永远是最棒的，是先射箭，再画靶的概念。expert 他去玩一玩游戏，得到这一些游戏的纪录，你的 actor 也去玩一玩游戏，得到这些游戏的纪录。接下来，你要定一个 reward function，这个 reward function 的原则就是，expert 得到的分数，要比 actor 得到的分数高。

先射箭，再画靶。所以我们今天就 learn 出一个 reward function，你要用什么样的方法都可以，你就找出一个 reward function R，这个 reward function 会使 expert 所得到的 reward，大过于 actor 所得到的 reward。

你有 reward function 就可以套用一般，Reinforcement Learning 的方法，去 learn 一个 actor，这个 actor 会对这一个 reward function，去 maximize 他的 reward，他也会采取一大堆的 action。

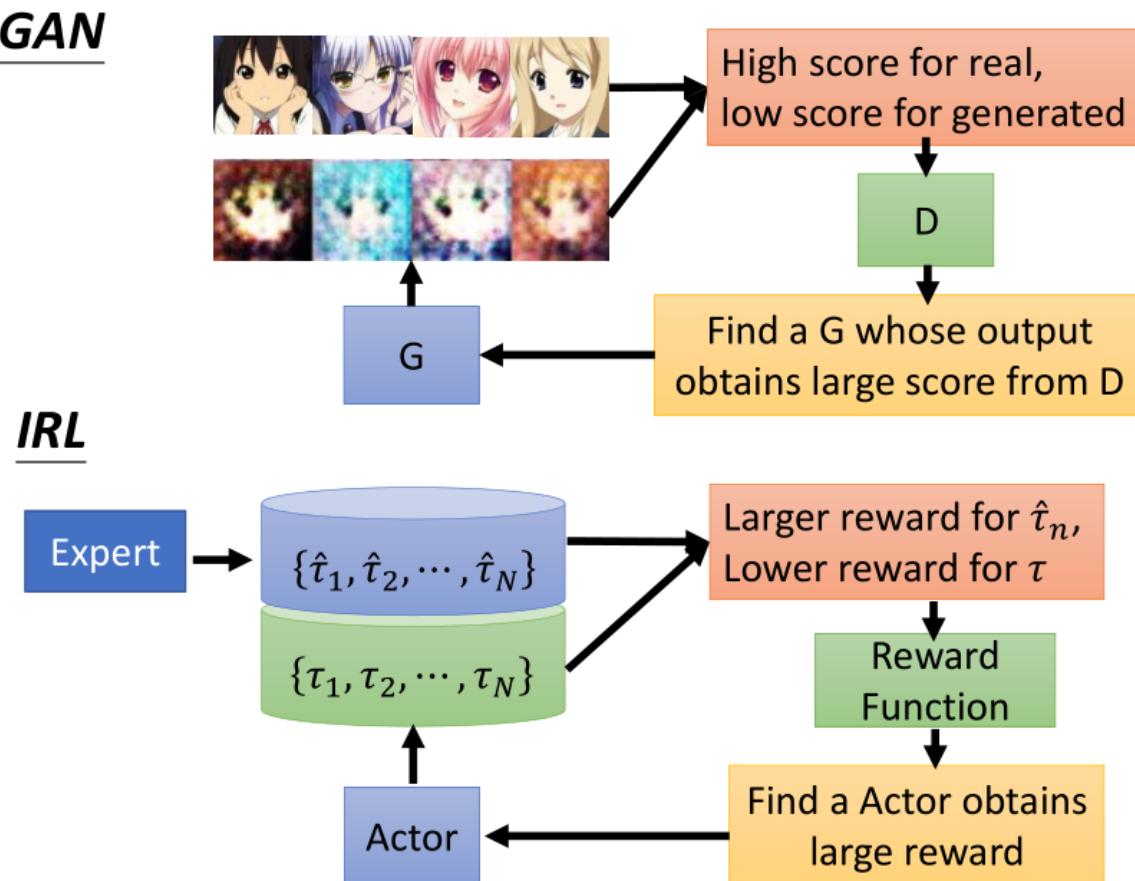
但是，今天这个 actor，他虽然可以 maximize 这个 reward function，采取一大堆的行为，得到一大堆游戏的纪录，但接下来，我们就改 reward function，这个 actor 已经可以在这个 reward function 得到高分，但是他得到高分以后，我们就改 reward function，仍然让 expert 比我们的 actor，可以得到更高的分数。

这个就是 Inverse Reinforcement learning，你有新的 reward function 以后，根据这个新的 reward function，你就可以得到新的 actor，新的 actor 再去跟环境做一下互动，他跟环境做互动以后，你又会重新定义你的 reward function，让 expert 得到 reward 大过 actor 得到的 reward。

这边其实就没有讲演算法的细节，那你至于说要，怎么让他大于他，其实你在 learning 的时候，你可以很简单地做一件事。我们的 reward function 也许就是 neural network，这个 neural network 它就是吃一个 τ ，然后，output 就是这个 τ 应该要给他多少的分数，或者是说，你假设觉得 input 整个 τ 太难了，因为 τ 是 s 跟 a 一个很长的 sequence，也许就说，他就是 input s 跟 a，他是一个 s 跟 a 的 pair，然后 output 一个 real number，把整个 sequence，整个 τ ，会得到的 real number 都加起来，就得到 total R，在 training 的时候，你就说，今天这组数字，我们希望他 output 的 R 越大越好，今天这个，我们就希望他 R 的值，越小越好。

你有没有觉得这个东西，其实看起来还颇熟悉呢？其实你只要把他换个名字说，actor 就是 generator，然后说 reward function 就是 discriminator。

其实他就是 GAN，他就是 GAN，所以你说，他会不会收敛这个问题，就等于是问说 GAN 会不会收敛，你应该知道说也是很麻烦，不见得会收敛，但是，除非你对 R 下一个非常严格的限制，如果你的 R 是一个 general 的 network 的话，你就会有很大的麻烦就是了。



那怎么说他像是一个 GAN？我们来跟 GAN 比较一下。

GAN 里面，你有一堆很好的图，然后你有一个 generator，一开始他根本不知道要产生什么样的图，他就乱画，然后你有一个 discriminator，discriminator 的工作就是，expert 画的图就是高分，generator 画的图就是低分，你有 discriminator 以后，generator 会想办法去骗过 discriminator，generator 会希望他产生的图，discriminator 也会给他高分。这整个 process 跟 Inverse Reinforcement Learning，是一模一样的，我们只是把同样的东西换个名子而已。

今天这些人画的图，在这边就是 expert 的 demonstration，你的 generator 就是 actor，今天 generator 画很多图，但是 actor 会去跟环境互动，产生很多 trajectory。这些 trajectory 跟环境互动的记录，游戏的纪录其实就等于是 GAN 里面的这些图。

然后，你 learn 一个 reward function，这个 reward function 其实就是 discriminator，这个 reward function 要给 expert 的 demonstration 高分，给 actor 互动的结果低分。然后接下来，actor 会想办法，从这个已经 learn 出来的 reward function 里面得到高分，然后接下来 iterative 的去循环，跟 GAN 其实是一模一样的。我们只是换个说法来讲同样的事情而已。

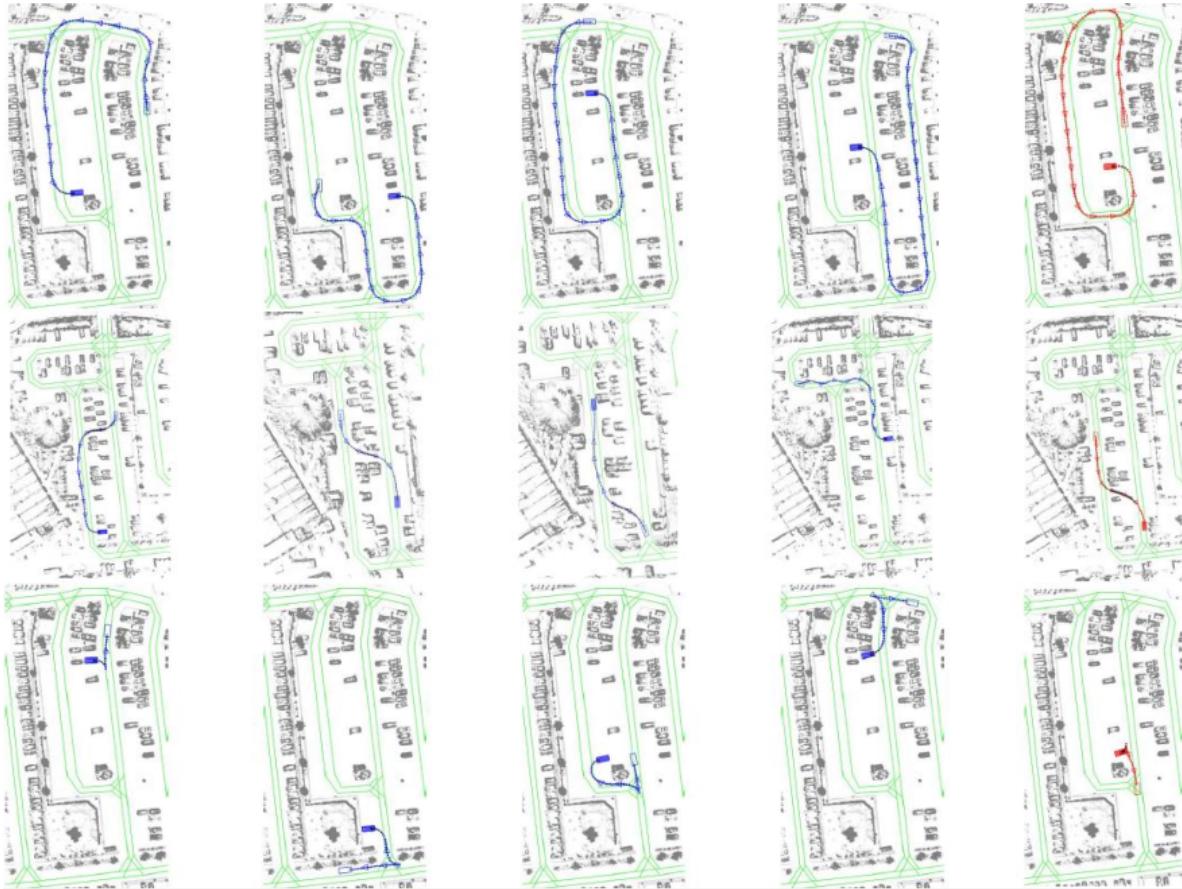
Parking Lot Navigation

那这个 IRL 其实有很多的 application，举例来说，当然可以用开来自驾车，然后，有人用这个技术来学开自驾车的不同风格。

Reward function:

- Forward vs. reverse driving
- Amount of switching between forward and reverse
- Lane keeping
- On-road vs. off-road
- Curvature of paths

每个人在开车的时候，其实你会有不同风格，举例来说，能不能够压到线，能不能够倒退，要不要遵守交通规则等等，每个人的风格是不同的。用 Inverse Reinforcement Learning，又可以让自驾车学会各种不同的开车风格。



这个是文献上真实的例子，在这个例子里面，Inverse Reinforcement Learning 有一个有趣的地方，通常你不需要太多的 training data，因为 training data 往往都是个位数，因为 Inverse Reinforcement Learning 只是一种 demonstration，他只是一种范例。今天机器他仍然实际上可以去跟环境互动，非常的多次，所以在Inverse Reinforcement Learning 的文献，往往可以看到说，只用几笔 data 就训练出一些有趣的结果。

比如说，在这个例子里面，然后就是给机器只看一个 row，的四个 demonstration，然后让他去学怎么样开车，怎么样开车。

今天给机器看不同的 demonstration，最后他学出来开车的风格，就会不太一样。举例来说，这个是不守规的矩开车方式，因为他会开到道路之外，这边，他会穿过其他的车，然后从这边开进去，所以机器就会学到说，不一定要走在道路上，他可以走非道路的地方。

或是这个例子，机器是可以倒退的，他可以倒退一下，他也会学会说，他可以倒退。

Robot

那这种技术，也可以拿来训练机器人，你可以让机器人，做一些你想要他做的动作。过去如果你要训练机器人，做你想要他做的动作，其实是比较麻烦的，怎么麻烦，过去如果你要操控机器的手臂，你要花很多力气去写那 program，才让机器做一件很简单的事。

那今天假设你有 Imitation Learning 的技术，那也许你可以做的事情是，让人做一下示范，然后机器就跟着人的示范来进行学习。

Third Person Imitation Learning

其实还有很多相关的研究。举例来说，你在教机械手臂的时候，要注意就是，也许机器看到的视野，跟人看到的视野，其实是不太一样的。

在刚才那个例子里面，我们人跟机器的动作是一样的，但是在未来的世界里面，也许机器是看着人的行为学的。假设你要让机器学会打高尔夫球，在刚才的例子里面就是，人拉着机器人手臂去打高尔夫球，但是在未来有没有可能，机器就是看着人打高尔夫球，他自己就学会打高尔夫球了呢？

但这个时候，要注意的事情是，机器的视野，跟他真正去采取这个行为的时候的视野，是不一样的，机器必须了解到，当他是作为第三人称的时候，当他是第三人的视角的时候，看到另外一个人在打高尔夫球，跟他实际上自己去打高尔夫球的时候，看到的视野显然是不一样的，但他怎么把他是第三人的时候，所观察到的经验，把它 generalize 到他是第一人称视角的时候，第一人称视角的时候，所采取的行为，这就需要用到 Third Person Imitation Learning 的技术。

那这个怎么做呢？细节其实我们就不细讲，他的技术，其实也是不只是用到 Imitation Learning，他用到了 Domain-Adversarial Training，这也是一个 GAN 的技术，那我们希望今天有一个 extractor，有两个不同 domain 的 image，通过这个 extractor 以后，没有办法分辨出他来自哪一个 domain。

Imitation Learning 用的技术其实也是一样的，希望 learn 一个 Feature Extractor，当机器在第三人称的时候，跟他在第一人称的时候，看到的视野其实是一样的，就是把最重要的东西抽出来就好了。

Recap: Sentence Generation & Chat-bot

其实我们在讲 Sequence GAN 的时候，我们有讲过 Sentence Generation 跟 Chat-bot，那其实 Sentence Generation 或 Chat-bot 这件事情，也可以想成是 Imitation Learning。机器在 imitate 人写的句子。

Sentence Generation

Expert trajectory:

床前明月光

(s_1, a_1) : (“<BOS>”, “床”)

(s_2, a_2) : (“床”, “前”)

(s_3, a_3) : (“床前”, “明”)

⋮

⋮

Chat-bot

Expert trajectory:

input: how are you

Output: I am fine

(s_1, a_1) : (“input, <BOS>”, “I”)

(s_2, a_2) : (“input, I”, “am”)

(s_3, a_3) : (“input, I am”, “fine”)

⋮

⋮

Maximum likelihood is behavior cloning. Now we have better approach like SeqGAN.

你可以把写句子这件事情，你在写句子的时候，你写下去的每一个 word，你都当成是一个 action，所有的 word 合起来就是一个 episode。

举例来说，sentence generation 里面，你会给机器看很多人类写的文字，那这个人类写的文字，你要让机器学会写诗，那你就需要给他看唐诗 300 首，这个人类写的文字，其实就是这个 expert 的 demonstration，每一个词汇，其实就是一个 action。

你让机器做 Sentence Generation 的时候，其实就是在 imitate expert 的 trajectory，或是如果 Chat-bot 也是一样，在 Chat-bot 里面你会收集到很多人互动对话的纪录，那一些就是 expert 的 demonstration。

如果我们今天单纯用 Maximum likelihood 这个技术，来 maximize 会得到 likelihood，这个其实就是 behavior cloning，对不对？用我们今天做 behavior cloning，就是看到一个 state，接下来预测，我们会得到，看到一个 state，然后有一个 Ground truth 告诉机器说，什么样的 action 是最好的，在做 likelihood 的时候也是一样，Given sentence 已经产生的部分，接下来 machine 要 predict 说，接下来要写哪一个 word 才是最好的。

所以，Maximum likelihood 对应到 Imitation Learning 里面，就是 behavior cloning。

那我们说光 Maximum likelihood 是不够的，我们想要用 Sequence GAN，其实 Sequence GAN 就是对应到 Inverse Reinforcement Learning，我们刚才已经有讲过说，其实 Inverse Reinforcement Learning，就是一种 GAN 的技术。

你把 Inverse Reinforcement Learning 的技术，放在 Sentence generation，放到 Chat-bot 里面，其实就是 Sequence GAN 跟他的种种的变形。

Structured Learning

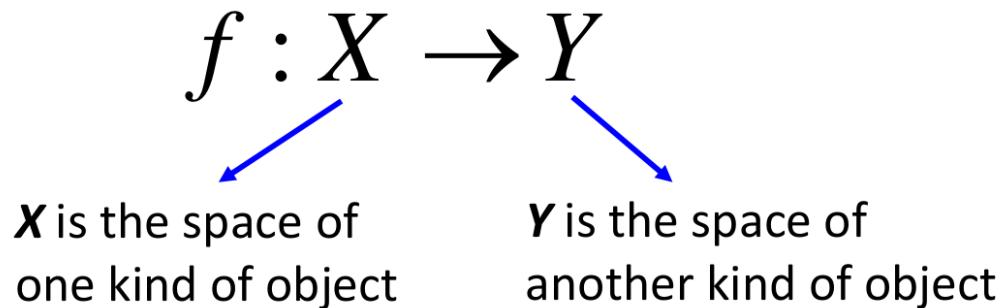
Structured Learning

什么是 Structured Learning 呢？到目前为止，我们考虑的 input 都是一个 vector，output 也是一个 vector，不管是 SVM 还是 Deep Learning 的时候，我们的 input，output 都是 vector 而已。但是实际上我们要真正面对的问题往往比这个更困难，我们可能需要 input 或者 output 是一个 sequence，我们可能希望 output 是一个 list，是一个 tree，是一个 bounding box 等等。比如 recommendation 里面你希望 output 是一个 list，

而不是一个个element。

当然，大原则上我们知道怎么做，我们就是要找一个function，它的input就是我们要的object，它的output就是另外一种object，只是我们不知道要怎么做。比如说，我们目前学过的deep learning的Neural Network的架构，你可能不知道怎样Network的input才是一个tree structure，output是另外一个tree structure。

- We need a more powerful function f
 - Input and output are both objects with structures
 - Object: sequence, list, tree, bounding box ...



In the previous lectures, the input and output are both vectors.

特点：

- 输入输出都是一种带有结构的对象
- 对象：sequence, list, tree, bounding box

Example Application

Structured Learning 的应用比比皆是

- Speech recognition(语音辨识)
input 是一个signal sequence, output是另一个text sequence
- Translation(翻译)
input 是一种语言的sequence, output是另外一种语言的sequence
- Syntactic Paring(文法解析)
input 是一个sentence, output 是一个文法解析树
- Object Detection(目标检测)
或者你要做Object detection, input 是一张image, output是一个bounding box。你会用这个bounding box把这个object给框出来。
- Summarization
或者你要做一个Summarization, input是一个大的document, output是一个summary。input 和 output都是一个sequence。
- Retrieval
或者你要做一个Retrieval, input是搜寻的关键词, output是搜寻的结果, 是一个webpage的list。

Unified Framework

那么Structured到底要怎么做呢？虽然这个Structured听起来很困难，但是实际上它有一个Unified Framework，统一的框架。

在Training的时候，就是找到function，记为 F ，这个大写 F 的input是 X 跟 Y ，它的output是一个real number。这个大写的 F 它所要做的事情就是衡量出输入 x ，输出 y 都是structure的时候， x 和 y 有多匹配。越匹配， R 值越大。

Training

- Find a function F

$$F: X \times Y \rightarrow \mathbb{R}$$

- $F(x, y)$: evaluate how compatible the objects x and y is

Inference (Testing)

- Given an object x

$$\tilde{y} = \arg \max_{y \in Y} F(x, y)$$

$$f: X \rightarrow Y \rightarrow f(x) = \tilde{y} = \arg \max_{y \in Y} F(x, y)$$

那testing的时候，给定一个新的 x ，我们去穷举所有的可能的 y ，——带进大写的 F function，看哪一个 y 可以让 F 函数值最大，此时的 \tilde{y} 就是最后的结果，model的output。

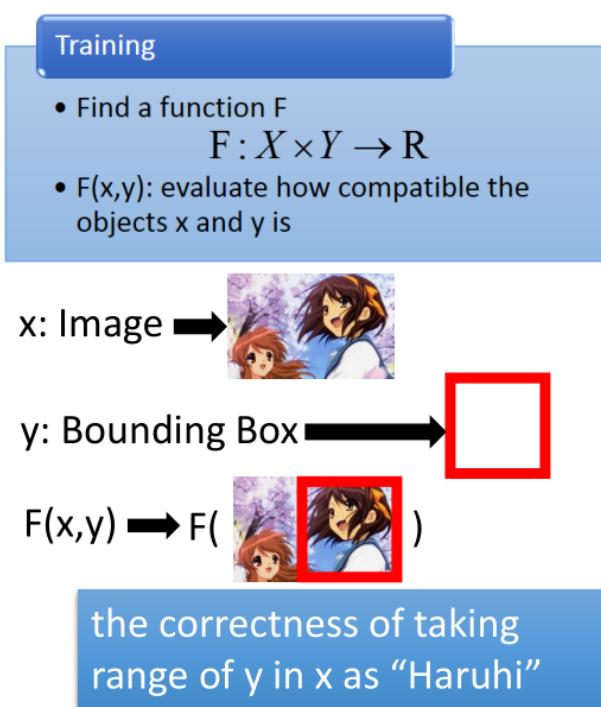
之前我们所要做的事情，是找一个小写的 $f: X \rightarrow Y$ ，可以想象成现在小写的 $f(x) = \tilde{y} = \arg \max_{y \in Y} F(x, y)$ ，这样讲可能比较抽象，我们来举个实际的例子。

Object Detection

用一个方框标识出一张图片中的要找的object，在我们的task中input是一张image，output是一个Bounding Box。举例来说，我们的目标是要检测出Haruhi。input是一张image，output就是Haruhi所在的位置。可以用于侦测人脸，无人驾驶等等。

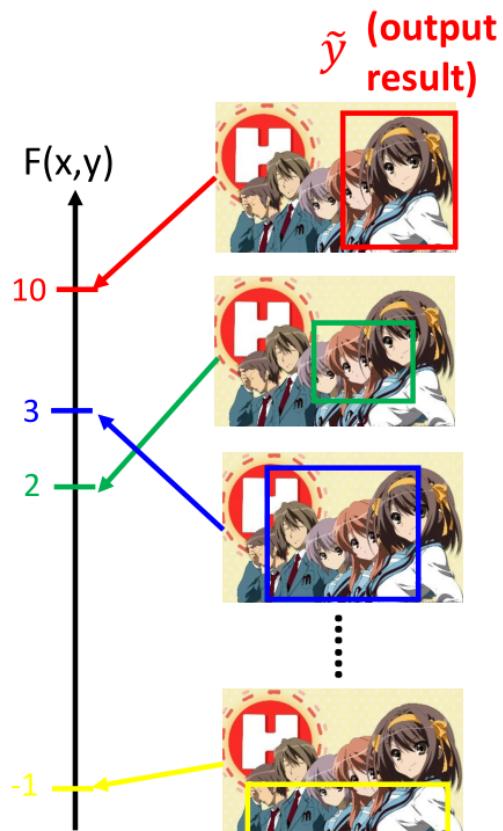
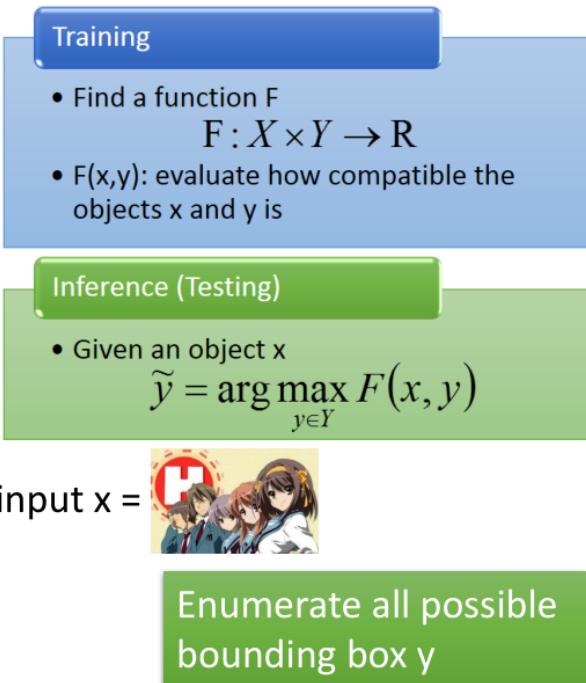
在做object detection的时候，也可以用Deep Learning。事实上，Deep Learning 和 Structured Learning 是有关系的，这个是我个人的想法，GAN就是 $F(X, Y)$ ，具体的后续再讲。

那么Object Detection是怎么做的呢？input就是一张image，output就是一个Bounding Box， $F(x, y)$ 就是这张image配上这个红色的bounding box，它们有多匹配。如果是按照Object Detection的例子，就是它有多正确，真的吧Haruhi给框出来。所以你会期待，给这一张图，如果框得很对，那么它的分数就会很高。如下图右侧所示。



接下来，testing的时候，给一张image，这个 x 是从来没有看过的东西。你穷举所有可能的bounding box，画在各种不同的地方，然后看说哪一个bounding box得到的分数最高。红色的最高，所以红色的就是你的model output。

Unified Framework – Object Detection



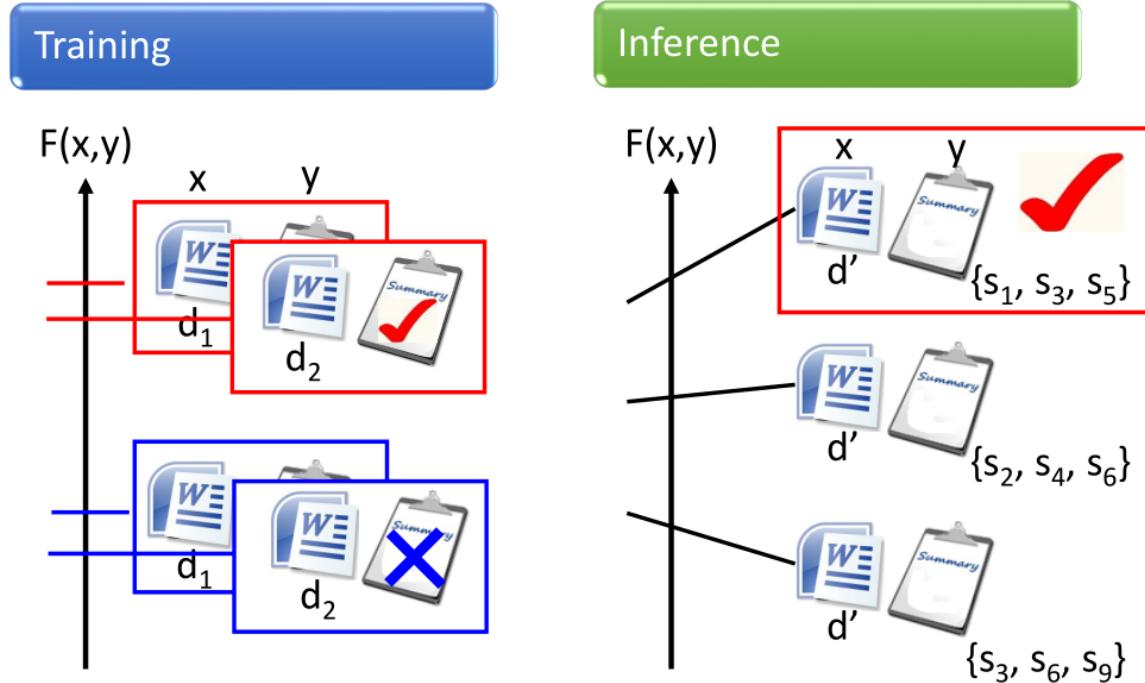
在别的task上其实也是差不多的，比如

Summarization

input一个长document，里面有很多句子。output是一个summary，summary可以从document上取几个句子出来。

那么我们training的时候，你的这个 $F(x,y)$ ，当document和summary配成一对的时候， F 的值就很大，如果document和不正确的summary配成一对的时候， F 的值就很小，对每一个training data都这么做。

testing的时候，就穷举所有可能的summary，看哪个summary配上的值最大，它就是model的output。



Retrieval

input 是一个查询集（查询关键字），output是一个webpages的list

Training的时候，我们要知道input一个query时，output是哪一些list，才是perfect。以及那些output是不对的，分数就会很低。

Testing的时候，根据input，穷举所有的可能，看看哪个list的分数最高，就是model的输出。

Statistics

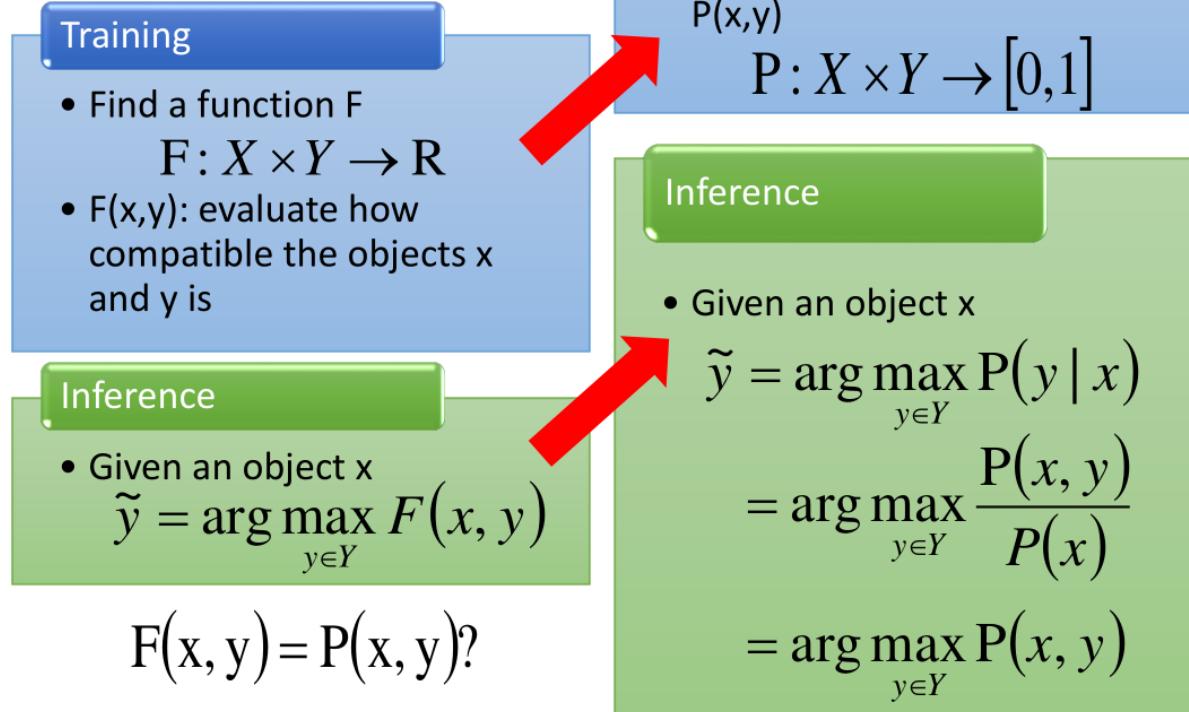
这个Unified Framework或许你听得觉得很怪这样，第一次听到，搞什么东西呀。

那么我换一个说法，我们在Training的时候要estimate x 和 y 的联合概率 $P(x,y)$ ，即 x 和 y 一起出现的机率，这样，input就是 X 和 Y ，output就是一个介于0到1之间的值。

那我在做testing的时候，给我一个object x ，我去计算所有的 $p(y|x)$ ，经过条件概率的推导，哪一个 $p(x,y)$ 的机率最高， \hat{y} 就是model的输出。

Statistics

Unified Framework



graphical model也是一种structured learning，就是把 $F(x, y)$ 换成机率

用机率表达的方式

- 缺点
 - 机率解释性有限，比如搜寻，我们说查询值和结果共同出现的机率就很怪
 - 机率值限定在[0,1]范围， X 和 Y 都是很大的space，要变成机率可能很多时间都用来做 normalization，不一定有必要
- 优点
 - 具有现象意义，机率比较容易描述现象

Energy-based model 也是structured learning

Three Problems

要做这个Framework要解三个问题

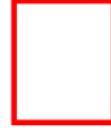
Problem 1: Evaluation

第一个问题是，在不同的问题中， $F(x,y)$ 到底应该是什么样的。

- **Evaluation:** What does $F(x, y)$ look like?
 - How $F(x, y)$ compute the “compatibility” of objects x and y

Object Detection:

$F(x = \text{girl's face}, y = \text{red square})$



Summarization:

$F(x = \text{document icon}, y = \text{summary icon})$



(a long document)

(a short paragraph)

Retrieval:

$F(x = \text{"Obama"}, y = \text{Search Result})$



(Search Result)

Problem 2: Inference

再来就是那个荒唐的Inference，怎么解“arg max”这个问题。这个Y可是很大的，比如说你要做Object Detection，这个Y是所有可能的bounding box。这件事情做得到吗？

- **Inference:** How to solve the “arg max” problem

$$y = \arg \max_{y \in Y} F(x, y)$$

The space Y can be extremely large!

Object Detection: $Y = \text{All possible bounding box}$ (maybe tractable)

Summarization: $Y = \text{All combination of sentence set in a document ...}$

Retrieval: $Y = \text{All possible webpage ranking}$

Problem 3: Training

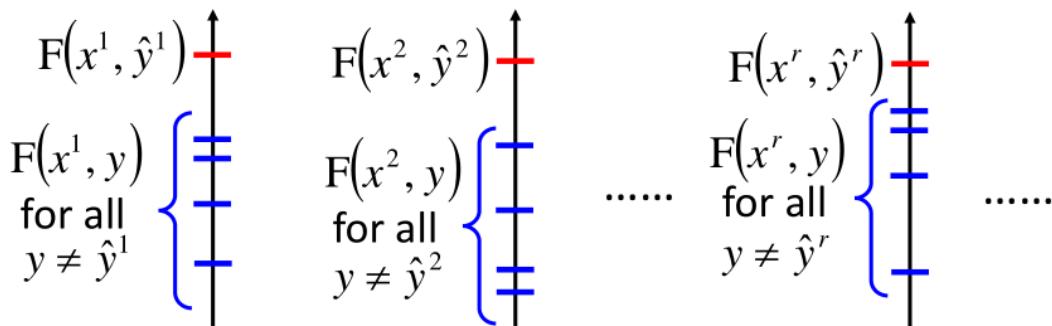
第三个问题是Training，给定training data，如何找到 $F(x, y)$ 。Training Principle是正确的 $F(x, \hat{y})$ 能大于其他的情况，这个Training应该是可以完成的。

- **Training:** Given training data, how to find $F(x, y)$

Principle

Training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^r, \hat{y}^r), \dots\}$

We should find $F(x, y)$ such that



只要你解出这三个问题，你就可以做Structured Learning。

Problem 1: Evaluation

- What does $F(x, y)$ look like?



Problem 2: Inference

- How to solve the “arg max” problem

$$y = \arg \max_{y \in Y} F(x, y)$$



Problem 3: Training

- Given training data, how to find $F(x, y)$



这三个问题可以跟HMM的三个问题联系到一起，也可以跟DNN联系到一起。

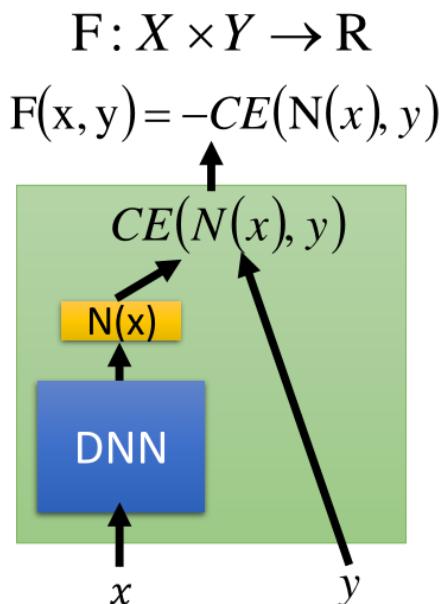
Link to DNN?

怎么说呢，比如说我们现在要做手写数字辨识，input一个image，把它分成10类，先把x扔进一个DNN，得到一个 $N(x)$ ，接下来我再input y ， y 是一个vector，把这个 y 和 $N(x)$ 算cross entropy， $-CE(N(x), y)$ 就是 $F(x, y)$ 。

Link to DNN?

The same as what we have learned.

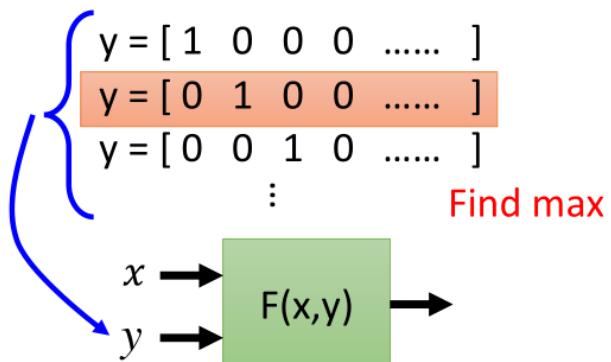
Training



Inference

$$\tilde{y} = \arg \max_{y \in Y} F(x, y)$$

In handwriting digit classification, there are only 10 possible y .



接下来，在testing的时候，就是说，我穷所有可能的辨识结果，也就是说10个 y ，每个都带进去这个Function里面，看哪个辨识结果能够让 $F(x, y)$ 最大，它就是我的辨识结果。这个跟我们之前讲的知识是一模一样的。

Structured Linear Model

Solution

假如Problem 1中的 $F(x, y)$ 有一种特殊的形式，那么Problem 3就不是个问题。所以我们就要先来讲special form应该长什么样子。

Problem 1

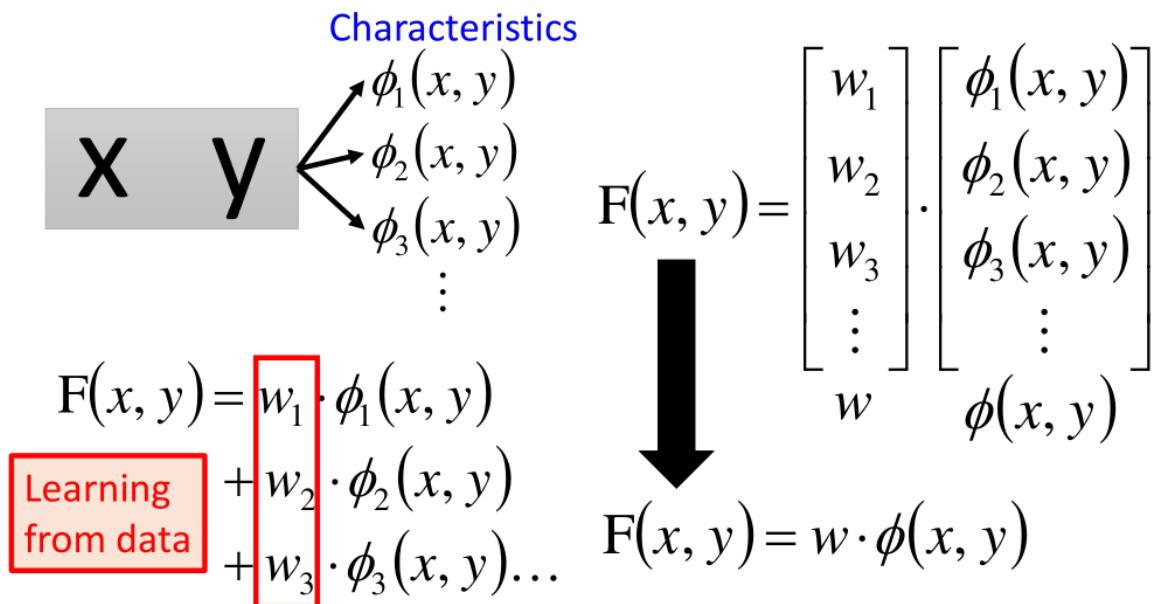
Evaluation: What does $F(x, y)$ look like?

special form必须是Linear，也就是说一个 (x, y) 的pair，首先我用一组特征来描述 (x, y) 的pair，其中 ϕ_i 代表一种特征，也就说 (x, y) 具有特征 ϕ_1 是 $\phi_1(x, y)$ 这个值，具有特征 ϕ_2 是 $\phi_2(x, y)$ 这个值，等等。然后 $F(x, y)$ 它长得什么样子呢？

$$F(x, y) = w_1 \phi_1(x, y) + w_2 \phi_2(x, y) + w_3 \phi_3(x, y) + \dots$$

向量形式可以写为 $F(x, y) = \mathbf{w} \cdot \phi(x, y)$

- **Evaluation:** What does $F(x,y)$ look like?



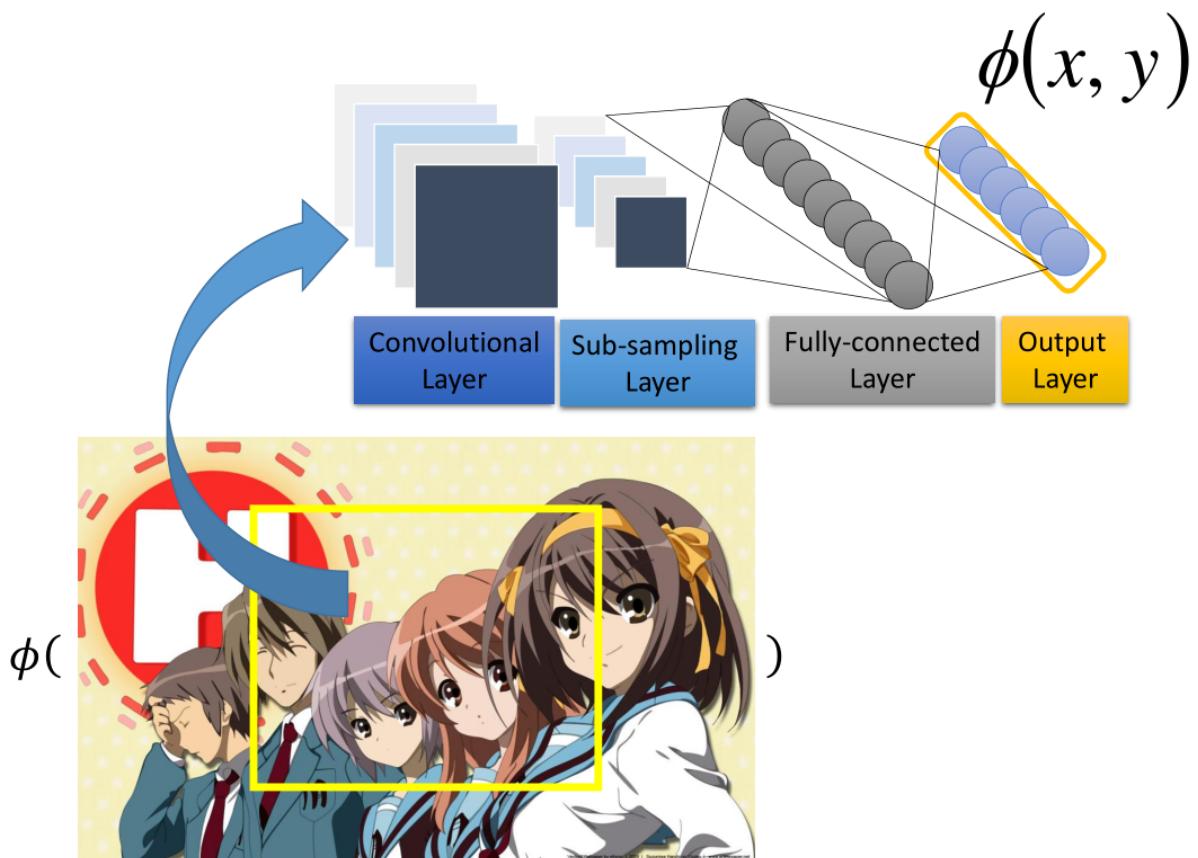
Object Detection

举个object detection的例子，框出Harihu， ϕ 函数可能为红色的pixel在框框里出现的百分比为一个维度，绿色的pixel在框框里出现的百分比为一个维度，蓝色的是一个维度，或者是红色在框框外的百分比是一个维度，等等，或者是框框的大小是一个维度。

现在image中比较state-of-the-art 可能是用visual word，visual word就是图片上的小方框片，每一个方片代表一种pattern，不同颜色代表不同的pattern，就像文章词汇一样。你就可以说在这个框框里面，编号为多少的visual word出现多少个就是一个维度的feature。

这些feature要由人找出来的吗？还是我们直接用一个model来抽呢， $F(x,y)$ 是一个linear function，它的能力有限，没有办法做太厉害的事情。如果你想让它最后performance好的话，那么就需要抽出很好的feature。用人工抽取的话，不见得能找出好的feature。

所以如果是在object detection 这个task上面，state-of-the-art 方法，比如你去train一个CNN，你可以把image丢进CNN，然后output一个vector，这个vector能够很好的代表feature信息。现在google在做object detection 的时候其实是用deep network 加上 structured learning 的方法做的，抽feature是用deep learning的方式来做的，具体如下图

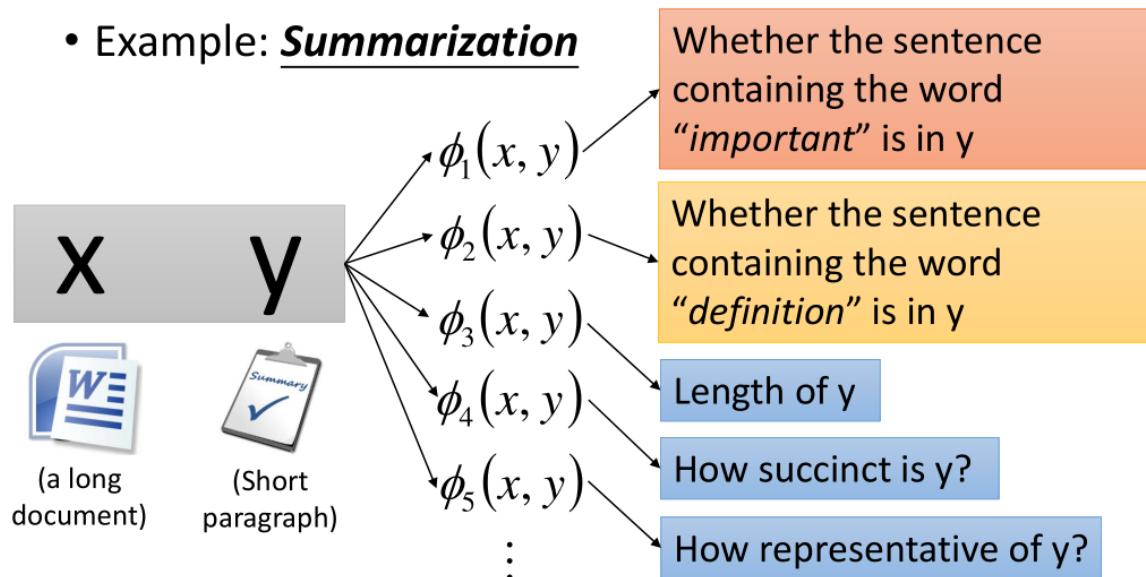


Summarization

你的 x 是一个document, y 是一个paragraph。你可以定一些feature, 比如说 $\phi_1(x, y)$ 表示 y 里面包含“important”这个单词则为1, 反之为0, 包含的话 y 可能权重会比较大, 可能是一个合理的summarization, 或者是 $\phi_2(x, y)$, y 里面有没有包含“definition”这个单词, 或者是 $\phi_3(x, y)$, y 的长度, 或者你可以定义一个evaluation说 y 的精简程度等等, 也可以想办法用deep learning找比较有意义的表示。具体如下图

- Evaluation: What does $F(x, y)$ look like?

- Example: Summarization

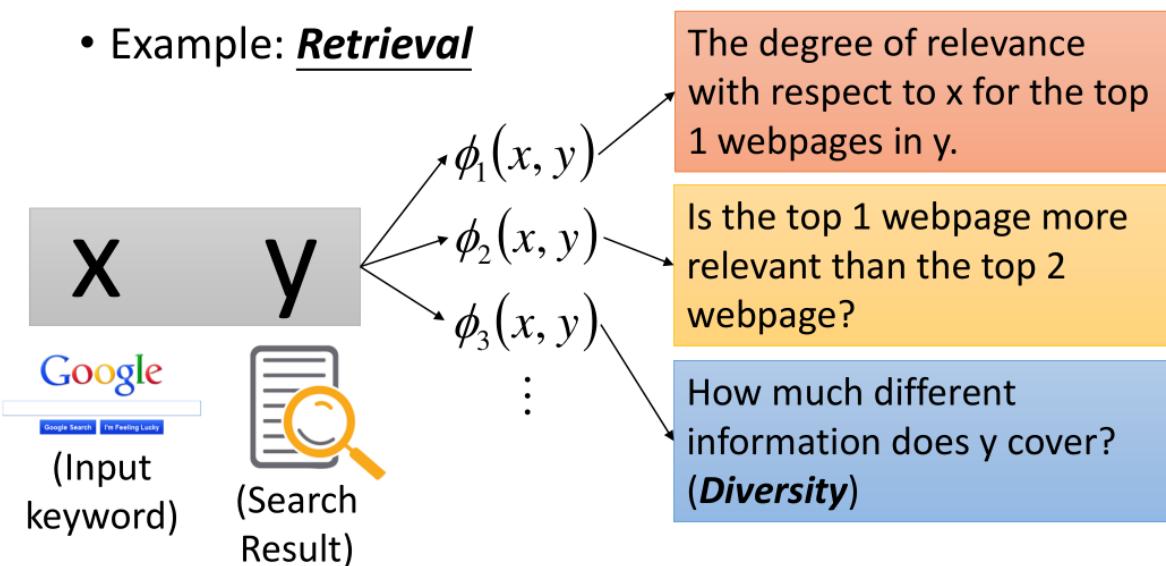


Retrieval

那比如说是Retrieval，其实也是一样啦。 x 是keyword， y 是搜寻的结果。比如 $\phi_1(x, y)$ 表示 y 第一笔搜寻结果跟 x 的相关度，或者 $\phi_2(x, y)$ 表示 y 的第一笔搜寻结果有没有比第二笔高等等，或者 y 的Diversity的程度是多少，看看我们的搜寻结果是否包含足够的信息。具体如下图

- Evaluation: What does $F(x, y)$ look like?

- Example: Retrieval



Problem 2

如果第一个问题定义好了以后，那第二个问题怎么办呢。 $F(x, y) = w \cdot \phi(x, y)$ 但是我们一样需要去穷举所有的 y ， $y = \arg \max_{y \in Y} w \cdot \phi(x, y)$ 来看哪个 y 可以让 $F(x, y)$ 值最大。

这个怎么办呢？假设这个问题已经被解决了

Problem 3

假装第二个问题已经被解决的情况下，我们就进入第三个问题。

有一堆的Training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^r, \hat{y}^r), \dots\}$ ，我希望找到一个function $F(x, y)$ ，其实是希望找到一个 w ，怎么找到这个 w 使得以下条件被满足：

对所有的training data而言，希望正确的 $w \cdot \phi(x^r, \hat{y}^r)$ 应该大过于其他的任何 $w \cdot \phi(x^r, y)$ 。

- Training: Given training data, how to learn $F(x, y)$
 - $F(x, y) = w \cdot \phi(x, y)$, so what we have to learn is w

Training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^r, \hat{y}^r), \dots\}$

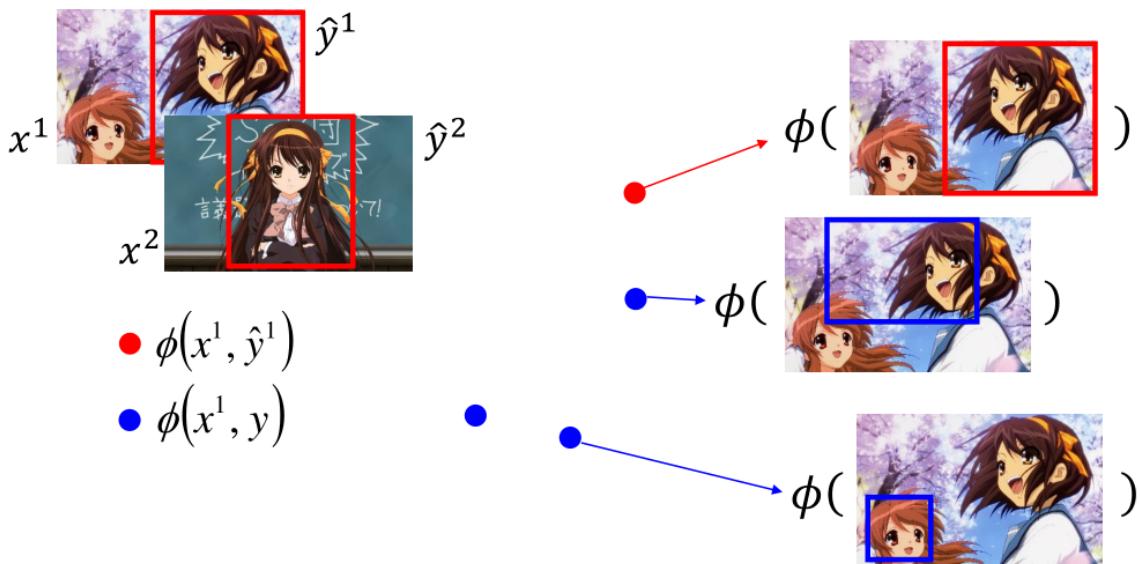
We should find w such that

$\forall r$ (All training examples)

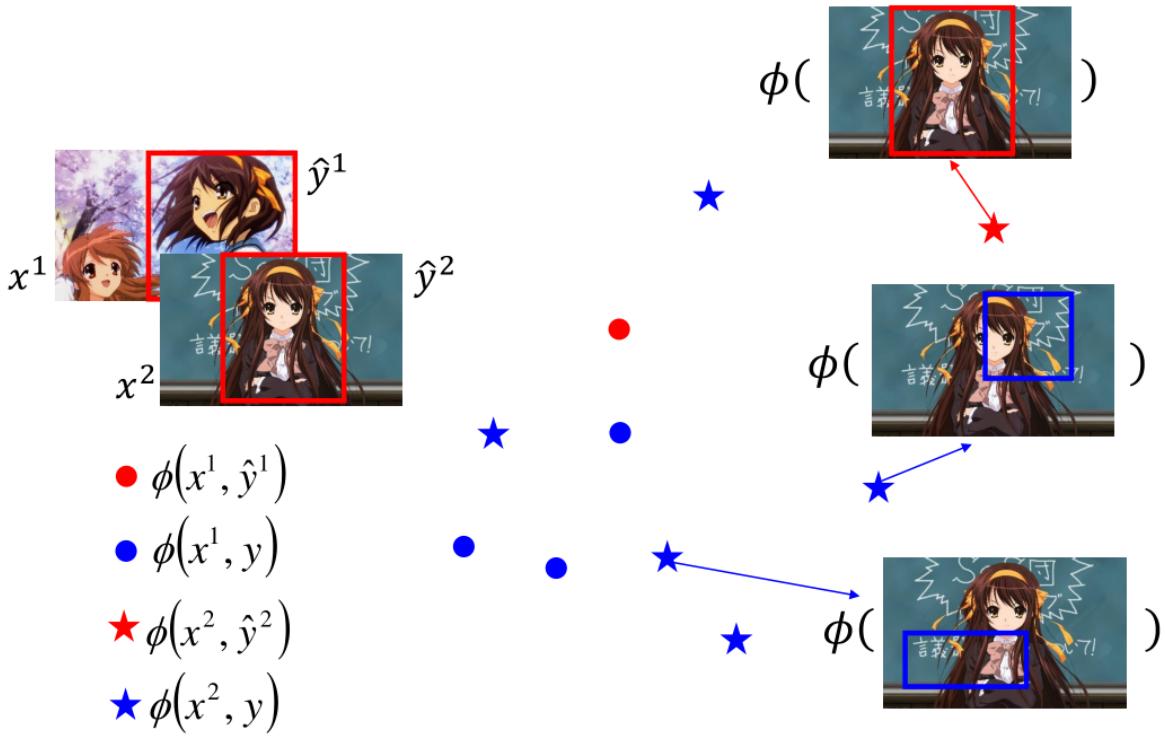
$\forall y \in Y - \{\hat{y}^r\}$ (All incorrect label for r -th example)

$$w \cdot \phi(x^r, \hat{y}^r) > w \cdot \phi(x^r, y)$$

用比较具体的例子来说明，假设我现在要做的object detection，我们收集了一张image x^1 ，然后呢，知道 x^1 所对应的 \hat{y}^1 ，我们又收集了另外一张图片，对应的框框也标出。对于第一张图，我们假设 (x^1, \hat{y}^1) 所形成的feature是红色 $\phi(x^1, \hat{y}^1)$ 这个点，其他的 y 跟 x^1 所形成的是蓝色的点。红色的点只有一个，蓝色的点有好多好多。

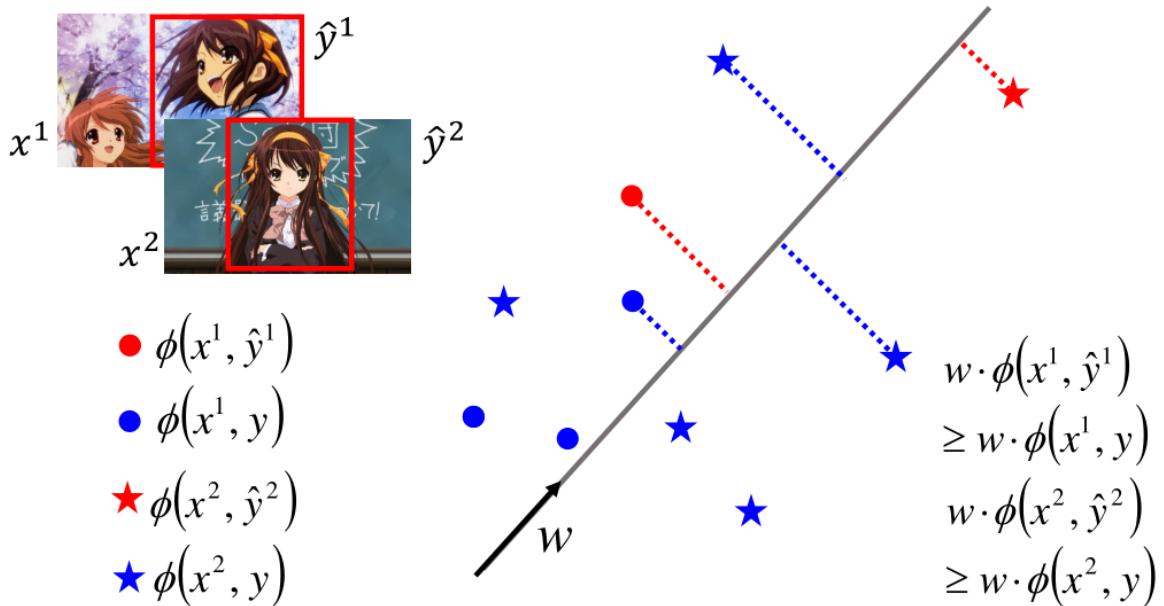


假设 (x^2, \hat{y}^2) 所形成的feature是红色的星星， x^2 与其他的 y 所形成的是蓝色的星星。可以想象，红色的星星只有一个，蓝色的星星有无数个。把它们画在图上，假设它们是如下图所示位置



我们所要达到的任务是，希望找到一个 w ，那这个 w 可以做到什么事呢？我们把这上面的每个点，红色的星星，红色的圈圈，成千上万的蓝色圈圈和蓝色星星通通拿去和 w 做inner cdot后，我得到的结果是红色星星所得到的大过于所有蓝色星星，红色的圈圈大过于所有红色的圈圈所得到的值。

不同形状之间我们就不比较。圈圈自己跟圈圈比，星星自己跟星星比。做的事情就是这样子，也就是说我希望正确的答案结果大于错误的答案结果，即
 $w \cdot \phi(x^1, \hat{y}^1) \geq w \cdot \phi(x^1, y^1)$, $w \cdot \phi(x^2, \hat{y}^2) \geq w \cdot \phi(x^2, y^2)$ 。



你可能会觉得这个问题会不会很难，蓝色的点有成千上万，我们有办法找到这样的 w 吗？这个问题没有我们想象中的那么难，以下我们提供一个演算法。

Algorithm

输入：训练数据 $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^r, \hat{y}^r), \dots\}$

输出：权重向量 w

假设我刚才说的那个要让红色的大于蓝色的vector，只要它存在，用这个演算法可以找到答案。

这个演算法是长什么样子呢？这个演算法的input就是我们的training data，output就是要找到一个vector w ，这个vector w 要满足我们之前所说的特性。

一开始，我们先initialize $w = 0$ ，然后开始跑一个循环，这个循环里面，每次我们都取出一笔training data (x^r, \hat{y}^r) ，然后我们去找一个 \tilde{y}^r ，它可以使得 $w \cdot (x^r, y)$ 的值最大，那么这个事情要怎么做呢？

这个问题其实就是Problem 2，我们刚刚假设这个问题已经解决了的，如果找出来的 \tilde{y}^r 不是正确答案，即 $\tilde{y}^r \neq \hat{y}^r$ ，代表这个 w 不是我要的，就要把这个 w 改一下。

怎么改呢？把 $\phi(x^r, \hat{y}^r)$ 计算出来，把 $\phi(x^r, \tilde{y}^r)$ 也计算出来，两者相减在加到 w 上，update w 。

有新的 w 后，再去取一个新的example，然后重新算一次max，如果算出来不对再update，步骤一直下去，如果我们要找的 w 是存在的，那么最终就会停止。

- **Input:** training data set $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^r, \hat{y}^r), \dots\}$
- **Output:** weight vector w
- **Algorithm:** Initialize $w = 0$
 - do
 - For each pair of training example (x^r, \hat{y}^r)
 - Find the label \tilde{y}^r maximizing $w \cdot \phi(x^r, y)$
$$\tilde{y}^r = \arg \max_{y \in Y} w \cdot \phi(x^r, y)$$
 (question 2)
 - If $\tilde{y}^r \neq \hat{y}^r$, update w
$$w \rightarrow w + \phi(x^r, \hat{y}^r) - \phi(x^r, \tilde{y}^r)$$
 - until w is not updated \rightarrow We are done!

这个算法有没有觉得很熟悉呢？这就是perceptron algorithm。perceptron 做的是二元分类，其实也是structured learning 的一个特例，它们的证明几乎是一样的。

举个例子来说明一下，刚才那个演算法是怎么运作的。

我们的目标是要找到一个 w ，它可以让红色星星大过蓝色星星，红色圈圈大过蓝色圈圈，假设这个 w 是存在的。首先我们假设 $w = 0$ ，然后我们随便pick 一个example (x^1, \hat{y}^1) ，根据手上的data 和 w 去看哪一个 \tilde{y}^1 使得 $w \cdot \phi(x^1, y)$ 的值最大。

现在 $w = 0$ ，不管是谁，所算出来的值都为0，所以结果值都是一样的。那么没关系，我们随机选一个 y 当做 \tilde{y}^1 就可以。我们假设选了下图的点作为 \tilde{y}^1 ，选出来的 $\tilde{y}^1 \neq \hat{y}^1$ ，对 w 进行调整，把 $\phi(x^1, \hat{y}^1)$ 值减掉 $\phi(x^1, \tilde{y}^1)$ 的值再和 w 加起来，更新 w

$$w \rightarrow w + \phi(x^1, \hat{y}^1) - \phi(x^1, \tilde{y}^1)$$

Algorithm - Example

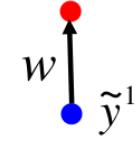
Initialize $w = 0$

pick (x^1, \hat{y}^1)

$$\tilde{y}^1 = \arg \max_{y \in Y} w \cdot \phi(x^1, y)$$

If $\tilde{y}^1 \neq \hat{y}^1$, update w

$$w \rightarrow w + \phi(x^1, \hat{y}^1) - \phi(x^1, \tilde{y}^1)$$



Because $w=0$ at this time, $\phi(x^1, y)$ always 0

Random pick one point as \tilde{y}^r

我们就可以获取到第一个 w , 第二步呢, 我们就在选一个example (x^2, \hat{y}^2) , 穷举所有可能的 y , 计算 $w \cdot \phi(x^2, y)$, 找出值最大时对应的 y , 假设选出下图的 \tilde{y}^2 , 发现不等于 \hat{y}^2 , 按照公式 $w \rightarrow w + \phi(x^2, \hat{y}^2) - \phi(x^2, \tilde{y}^2)$ 更新 w , 得到一个新的 w 。

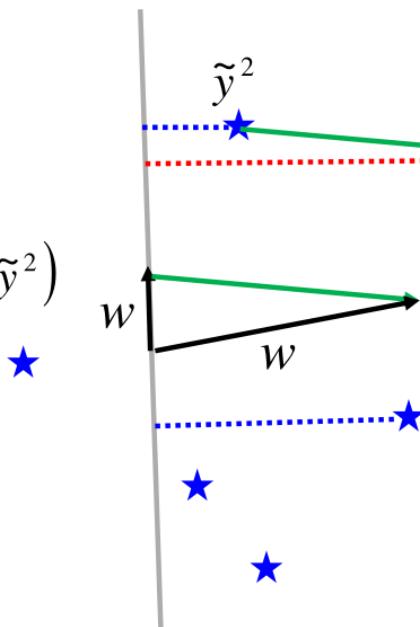
Algorithm - Example

pick (x^2, \hat{y}^2)

$$\tilde{y}^2 = \arg \max_{y \in Y} w \cdot \phi(x^2, y)$$

If $\tilde{y}^2 \neq \hat{y}^2$, update w

$$w \rightarrow w + \phi(x^2, \hat{y}^2) - \phi(x^2, \tilde{y}^2)$$



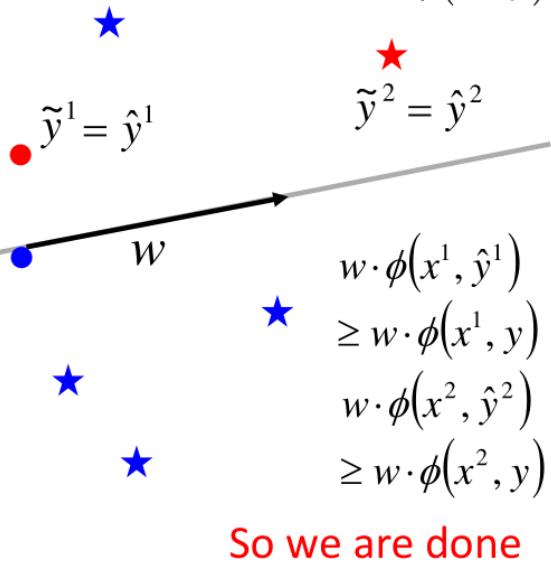
然后再取出 (x^1, \hat{y}^1) , 得到 $\tilde{y}^1 = \hat{y}^1$, 对于第一笔就不用更新。再测试第二笔data, 发现 $\tilde{y}^1 = \hat{y}^2$, w 也不用更新, 等等。看过所有data后, 发现 w 不再更新, 就停止整个training。所找出的 w 可以让 $\tilde{y}^r = \hat{y}^r$ 。

Algorithm - Example

pick (x^1, \hat{y}^1) again

$$\tilde{y}^1 = \arg \max_{y \in Y} w \cdot \phi(x^1, y)$$

$\tilde{y}^1 = \hat{y}^1 \rightarrow$ do not update w



下一节会证明这个演算法的收敛性，即演算法会结束。

Structured SVM

结构化学习要解决的问题，即需要找到一个强有力的函数 f

$$f : X \rightarrow Y$$

- 1. 输入和输出都是结构化的对象；
- 2. 对象可以为：sequence(序列), list(列表), tree(树结构), bounding box(包围框)，等等

其中， X 是一种对象的空间表示， Y 是另一种对象的空间表示。

这些问题有一个Unified Framework，只有两步

- 第一步：训练
 - 寻找一个函数 F , input是x和y, output是一个real number
- $F : X \times Y \rightarrow \mathbb{R}$
- $F(x, y)$: 用来评估对象x和y的兼容性 or 合理性
- 第二步：推理 or 测试
 - 即给定任意一个x, 穷举所有的y, 将 (x, y) 带入F, 找出最适当的y作为系统的输出。

$$\tilde{y} = \arg \max_{y \in Y} F(x, y)$$

虽然这个架构看起来很简单，但是想要使用的话要回答三个问题

- Q1: 评估
 - **What** does $F(x, y)$ look like?
- Q2: 推理

- **How** to solve the “arg max” problem, y 的可能性很多, 穷举是一件很困难的事, 需要找到某些方法解optimization的问题

$$\tilde{y} = \arg \max_{y \in Y} F(x, y)$$

- Q3: 训练

- 给定训练数据, 如何求解 $F(x, y)$?

Example Task: Object Detection

有比找框框更复杂的问题, 比如画出物体轮廓, 找出人的动作, 甚至不只是image processing的问题, 这些问题都可以套用接下来的解法。



Keep in mind that what you will learn today can be applied to other tasks.

- Q1: Evaluation

- 假设 $F(x, y)$ 是线性的, $F(x, y) = w \cdot \phi(x, y)$, ϕ 是人为定义的规则, w 是在Q3中利用训练数据来学习到的参数。
- 开放问题: 如果 $F(x, y)$ 不是线性, 该如何处理? F 是线性的话会很weak, 依赖于复杂的抽取特征的方式 ϕ , 我们希望机器做更复杂的事, 减少人类的接入。如果是非线性的话等下的讨论就不成立了, 因此目前的讨论多数是基于线性的 F 。

- Q2: Inference

$$\tilde{y} = \arg \max_{y \in Y} w \cdot \phi(x, y)$$

即给定一张图片 x , 穷举出所有可能的标记框 y , 对每一对 (x, y) , 用 $w \cdot \phi$ 计算出一对分数最大的 (x, y) , 我们就把对应的 y 作为输出。

算法的选择取决于task, 也取决于 $\phi(x, y)$

- 对于 Object Detection 可以选择的解决方法有
 - Branch & Bound algorithm(分支定界法)
 - Selective Search(选择性搜索)
- Sequence Labeling
 - Viterbi Algorithm(维特比译码算法)
- Genetic Algorithm(基因演算)
- 开放问题: What happens if the inference is non exact? 对结果影响会有多大呢? 这件事目前还没有太多讨论。

- Q3: Training

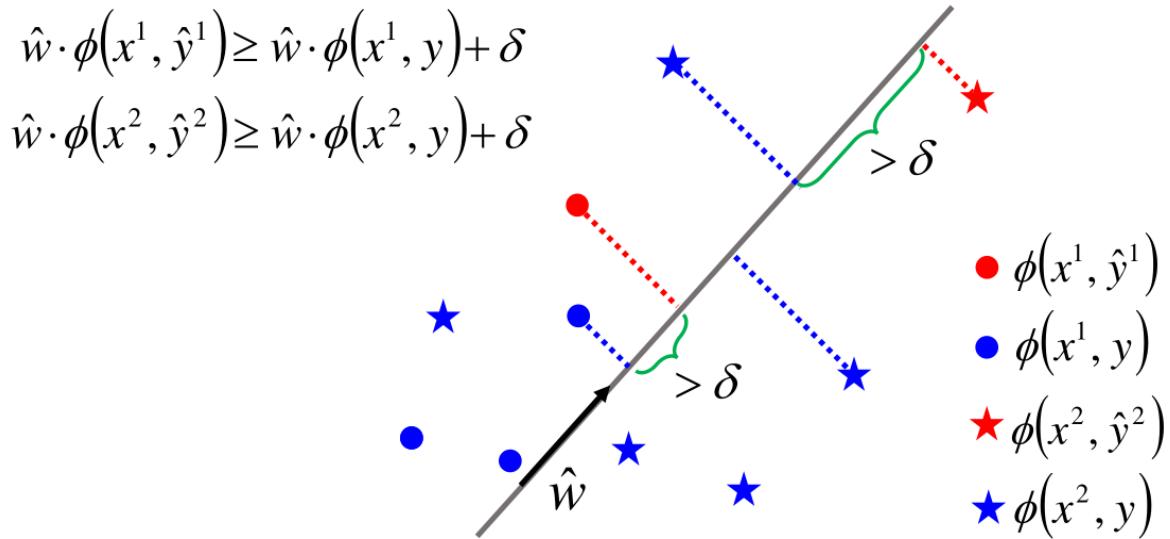
- Principle

对所有的training data $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2) \dots, (x^N, \hat{y}^N)\}$ 而言，希望正确的 $F(x^r, \hat{y}^r)$ 应该大于其他的任何 $F(x^r, y)$ 。

假定我们已经解决了Q1和Q2，只关注Q3如何处理：找到最佳的 $F(x, y)$ 。

Assumption: Separable

- There exists a weight vector \hat{w}



Separable: 存在一个权值向量 \hat{w} , 使得:

$$\begin{aligned}\hat{w} \cdot \phi(x^1, \hat{y}^1) &\geq \hat{w} \cdot \phi(x^1, y) + \delta \\ \hat{w} \cdot \phi(x^2, \hat{y}^2) &\geq \hat{w} \cdot \phi(x^2, y) + \delta\end{aligned}$$

红色代表正确的特征点(feature point)，蓝色代表错误的特征点(feature point)，可分性可以理解为，我们需要找到一个权值向量，其与 $\phi(x, y)$ 做内积(inner product)，能够让正确的point比蓝色的point的值均大于一个 δ 。

如果可以找到的话，就可以用以下的演算法找出 w

Structured Perceptron

- **Input:** training data set $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$
- **Output:** weight vector w
- **Algorithm:** Initialize $w = 0$
 - do
 - For each pair of training example (x^n, \hat{y}^n)
 - Find the label \tilde{y}^n maximizing $w \cdot \phi(x^n, y)$
$$\tilde{y}^n = \arg \max_{y \in Y} w \cdot \phi(x^n, y) \quad (\text{problem 2})$$
 - If $\tilde{y}^n \neq \hat{y}^n$, update w
- until w is not updated \rightarrow We are done!

输入：训练数据集

$$\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$$

输出：可以让data point separate 的 weight vector w

算法：首先我们假设 $w = 0$, 然后我们随便pick 一个example (x^1, \hat{y}^1) , 根据手上的data 和 w 去看 哪一个 \hat{y}^1 使得 $w \cdot \phi(x^1, y)$ 的值最大。假设选出来的 $\tilde{y}^1 \neq \hat{y}^1$, 对 w 进行调整, 把 $\phi(x^r, \hat{y}^r)$ 值减掉 $\phi(x^r, \tilde{y}^r)$ 的值再和 w 加起来, 更新 w 。不断进行iteration, 当对于所有data来说, 找到的 \tilde{y}^n 与 \hat{y}^n 都相等, w 不再更新, 就停止整个training。所找出的 w 可以让 $\tilde{y}^r = \hat{y}^r$ 。

问题是这个演算法要花多久的时间才可以收敛, 是否可以轻易的找到一个vector把蓝色的点和红色的点分开?

结论：在可分情形下, 我们最多只需更新 $(R/\delta)^2$ 次就可以找到 \hat{w} 。其中, δ 为margin(使得误分的点和正确的点能够线性分离), R 为 $\phi(x, y)$ 与 $\phi(x, y')$ 的最大距离, 与 y 的space无关, 因此蓝色的点非常多也不会影响我们update的次数。

Proof of Termination

一旦有错误产生, w 将会被更新

$$w^0 = 0 \rightarrow w^1 \rightarrow w^2 \rightarrow \dots \rightarrow w^k \rightarrow w^{k+1} \rightarrow \dots$$

$$w^k = w^{k-1} + \phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n))$$

注意：此处我们仅考虑可分情形

假定存在一个权值向量 \hat{w} 使得对于 $\forall n$ (所有的样本) 、 $\forall y \in Y - \{\hat{y}^n\}$ (对于一个样本的所有不正确的标记)

$$\hat{w} \cdot \phi(x^n, \hat{y}^n) \geq \hat{w} \cdot \phi(x^n, y) + \delta$$

不失一般性, 假设 $\|\hat{w}\| = 1$

证明：随着 k 的增加 \hat{w} 与 w^k 之间的角度 ρ_k 将会变小, $\cos \rho_k$ 会越来越大

$$\cos \rho_k = \frac{\hat{w}}{\|\hat{w}\|} \cdot \frac{w^k}{\|w^k\|}$$

$$\begin{aligned}\hat{w} \cdot w^k &= \hat{w} \cdot (w^{k-1} + \phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)) \\ &= \hat{w} \cdot w^{k-1} + \hat{w} \cdot \phi(x^n, \hat{y}^n) - \hat{w} \cdot \phi(x^n, \tilde{y}^n)\end{aligned}$$

在可分情形下，有

$$[\hat{w} \cdot \phi(x^n, \hat{y}^n) - \hat{w} \cdot \phi(x^n, \tilde{y}^n)] \geq \delta$$

所以得到

$$\hat{w} \cdot w^k \geq \hat{w} \cdot w^{k-1} + \delta$$

可得：

$$\begin{aligned}\hat{w} \cdot w^1 &\geq \hat{w} \cdot w^0 + \delta \quad \text{and} \quad w^0 = 0 \Rightarrow \hat{w} \cdot w^1 \geq \delta \\ \hat{w} \cdot w^2 &\geq \hat{w} \cdot w^1 + \delta \quad \text{and} \quad \hat{w} \cdot w^1 \geq \delta \Rightarrow \hat{w} \cdot w^2 \geq 2\delta \\ &\dots \\ \hat{w} \cdot w^k &\geq k\delta\end{aligned}$$

分子项不断增加

考虑分母 $\|w^k\|$, w^k 的长度

$$w^k = w^{k-1} + \phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)$$

则：

$$\begin{aligned}\|w^k\|^2 &= \|w^{k-1} + \phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)\|^2 \\ &= \|w^{k-1}\|^2 + \|\phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)\|^2 + 2w^{k-1} \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n))\end{aligned}$$

其中，

$$\begin{aligned}\|\phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)\|^2 &> 0 \\ 2w^{k-1} \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)) &< 0\end{aligned}$$

由于 w 是错误的，和此时找出的 \tilde{y}^n 内积要大于与正确 \hat{y}^n 的内积，因此第二个式子是小于零。

我们假设任意两个特征向量之间的距离 $\|\phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)\|^2$ 小于 R ，则有

$$\|w^k\|^2 \leq \|w^{k-1}\|^2 + R^2$$

于是

$$\begin{aligned}\|w^1\|^2 &\leq \|w^0\|^2 + R^2 = R^2 \\ \|w^2\|^2 &\leq \|w^1\|^2 + R^2 \leq 2R^2 \\ &\dots \\ \|w^k\|^2 &\leq kR^2\end{aligned}$$

综上可以得到

$$\hat{w} \cdot w^k \geq k\delta \quad \|w^k\|^2 \leq kR^2$$

则

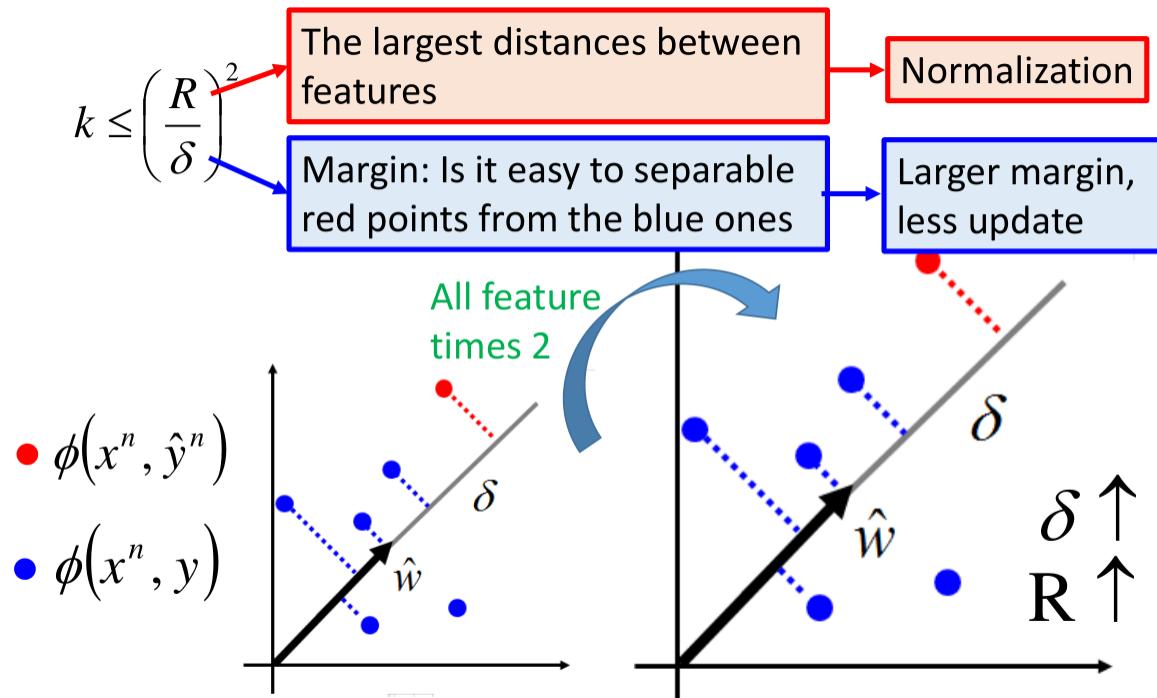
$$\cos \rho_k = \frac{\hat{w}}{\|\hat{w}\|} \cdot \frac{w^k}{\|w^k\|} \geq \frac{k\delta}{\sqrt{kR^2}} = \sqrt{k} \frac{\delta}{R} \leq 1$$

因此随着 k 的增加， $\cos \rho_k$ 的 lower bound 也在增加，并且 $\cos \rho_k \leq 1$

即得到

$$k \leq \left(\frac{R}{\delta}\right)^2.$$

How to make training fast?



单纯把feature $\times 2$, 随着 δ 的增大, R 也会增大, 因此training不会变快

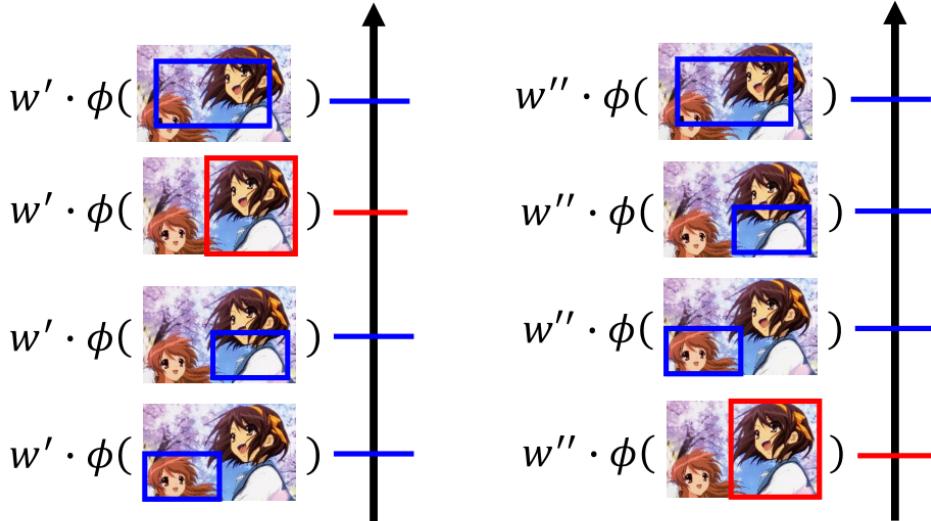
Non-separable Case

虽然可能没有任何一个vector可以让正确和错误答案完全分开, 但是还是可以鉴别出vector的好坏。比如下图左就比右要好。

Non-separable Case

Undoubtedly, w' is better than w'' .

- When the data is non-separable, some weights are still better than the others.



Defining Cost Function

定义一个成本函数C来评估w的效果有多差，然后选择w，从而最小化成本函数C。

第n笔data的Cost为，在此w下，与 x^n 最匹配的 y 的分数减去真实的 \hat{y} 的分数

$$C^n = \max_y [w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

$$C = \sum_{n=1}^N C^n$$

What is the minimum value?

$$C^n \geq 0$$

Other alternatives?

Problem 2中已经计算出了第一名的值是多少，因此用第一名的值减去 \hat{y} 最方便，其他的方案，比如用前三名的值，需要算出前三名的结果才可以

(Stochastic) Gradient Descent

Find w minimizing the cost C

$$C = \sum_{n=1}^N C^n$$

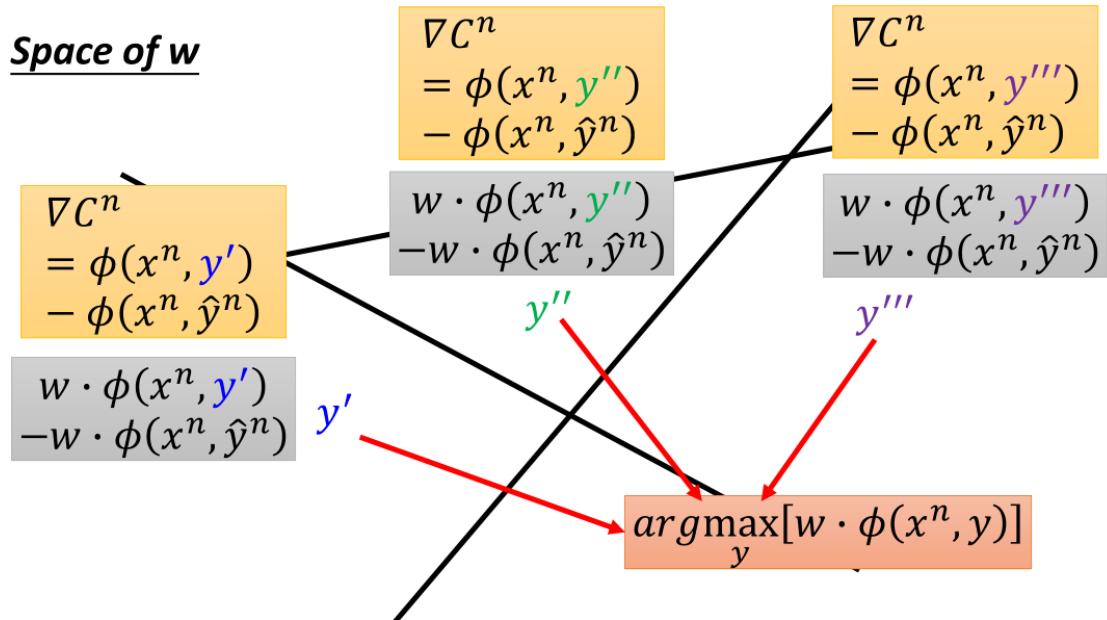
$$C^n = \max_y [w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

我们只需要算出 C^n 的梯度，就可以利用梯度下降法，但是式子中有 \max ，如何求梯度？

$$C^n = \max_y [w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

How to compute ∇C^n ?

When w is different,
the y can be different.



当 w 不同时，得到的 $y = \arg \max_y [w \cdot \phi(x^n, y)]$ 也会改变；假设 w 的 space 被 $y = \arg \max_y [w \cdot \phi(x^n, y)]$ 切割成好几块，得到的 $y = \arg \max_y [w \cdot \phi(x^n, y)]$ 分别等于 y', y'', y''' ，在边界的地方没有办法微分，但是在每一个 region 里面都是可以微分的。得到的梯度如图中黄色方框中。

利用(Stochastic) Gradient Descent求解

For $t = 1$ to T : ← Update the parameters T times

Randomly pick a training data $\{x^n, \hat{y}^n\}$ ← stochastic

$\tilde{y}^n = \arg \max_y [w \cdot \phi(x^n, y)]$ ← Locate the region

$\nabla C^n = \phi(x^n, \tilde{y}^n) - \phi(x^n, \hat{y}^n)$ ← simple

$$w \rightarrow w - \eta \nabla C^n$$

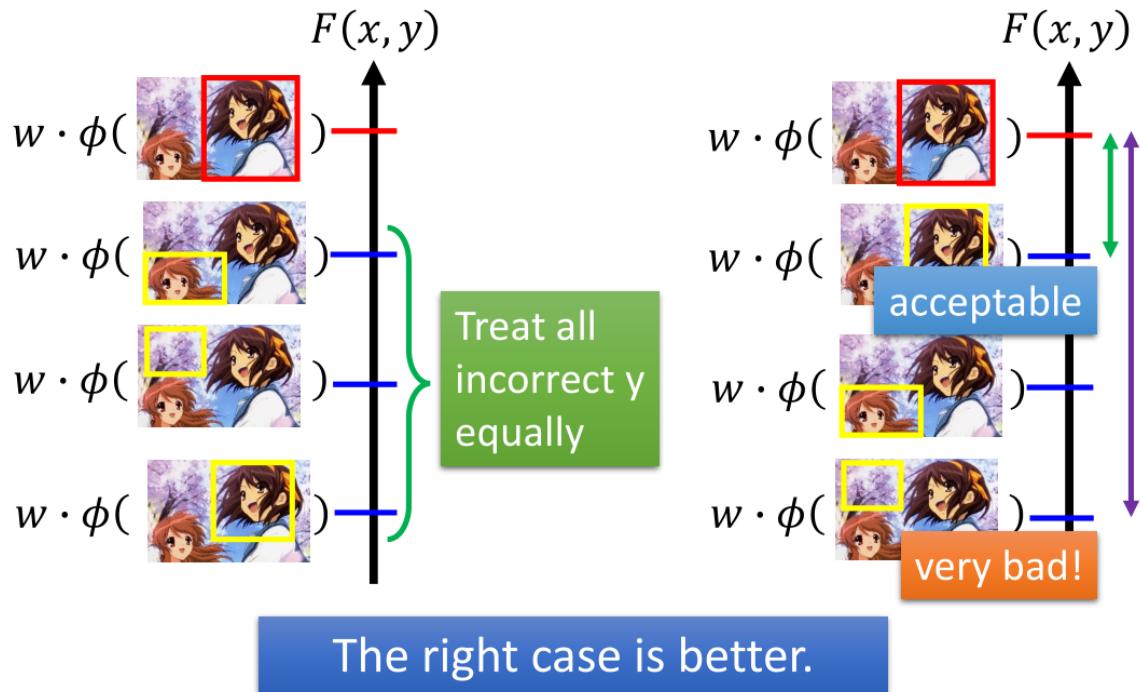
$$= w - \eta [\phi(x^n, \tilde{y}^n) - \phi(x^n, \hat{y}^n)]$$

If we set $\eta = 1$, then we are doing structured perceptron.

当学习率设为1时，就转换为structured perceptron。

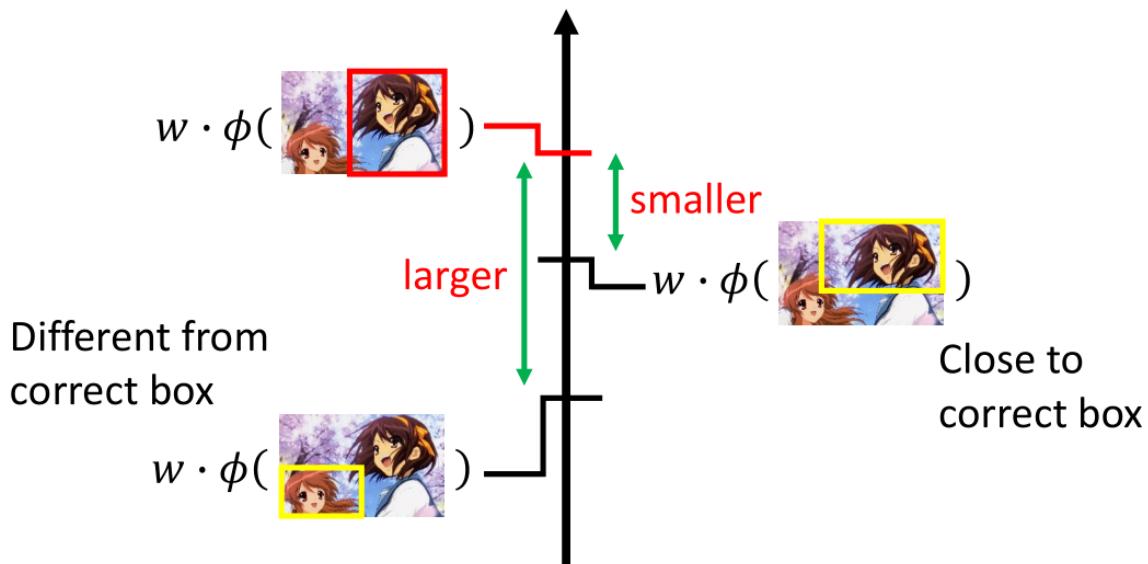
Considering Errors

在刚才，所有错误是视为一样的，然而不同的错误之间是存在差异的，错误可以分为不同的等级，我们在训练时需要考虑进去。比如框在樱花树上分数会特别低，框在凉宫春日脸上，分数会比较高，接近正确的分数也是可以的。如果有一个w只知道把正确的摆在第一位；相反另一个w，可以按照方框好坏来排序，那learn到的结果是比较安全的，因为分数比较高的和第一名差距没有很大。



Defining Error Function

错误的结果和正确的结果越像，那么分数的差距比较小；相反，差距就比较大。问题是如何衡量这种差异呢？

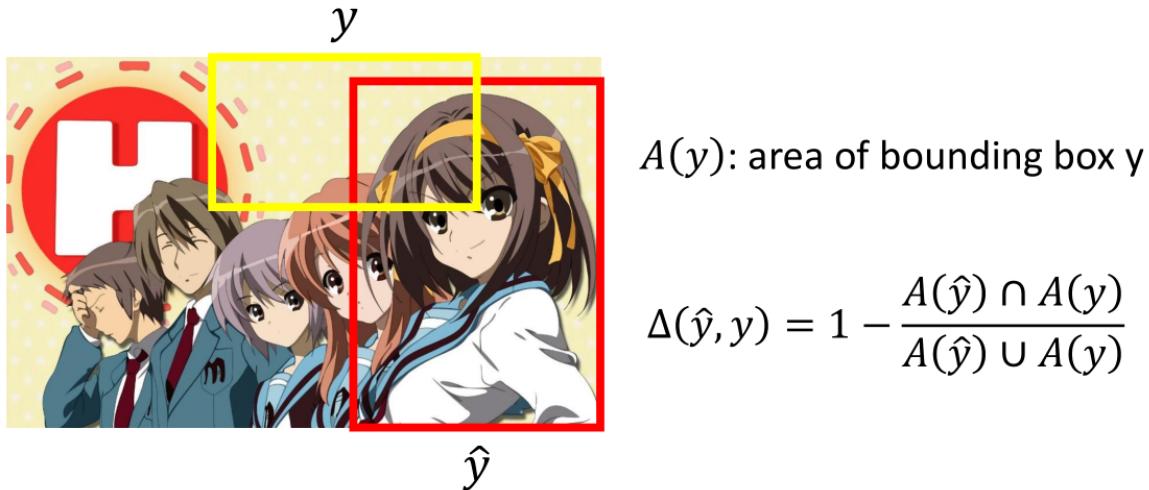


How to measure the difference

\hat{y} (正确的标记)与某一个 y 之间的差距定义为 $\Delta(\hat{y}, y) (> 0)$ ，如果和真实结果相同 $\Delta = 0$ ，具体形式根据任务不同而不同。

在下面的讨论中我们定义为

- $\Delta(\hat{y}, y)$: difference between \hat{y} and y (> 0)



Another Cost Function

修改Cost Function，本来的Cost是取分数最高的 y 的分数减去 \hat{y} 得到的分数；

我们会把 y 的分数加上 Δ ，这样可以使得当存在与 x^n 最匹配的 y 分数大，margin也大的项时，Cost会很大，当分数大， Δ 小，我们认为他是真正的比较好的。

当 Δ 很大时，我们希望他的分数很小；当 Δ 很小时，即使它的分数高也没有关系。margin越大，也就说明和真实之间的差距越大，损失也就越大，当然你可以定其他的差距式子，定的好不好可能会影响损失函数的结果。

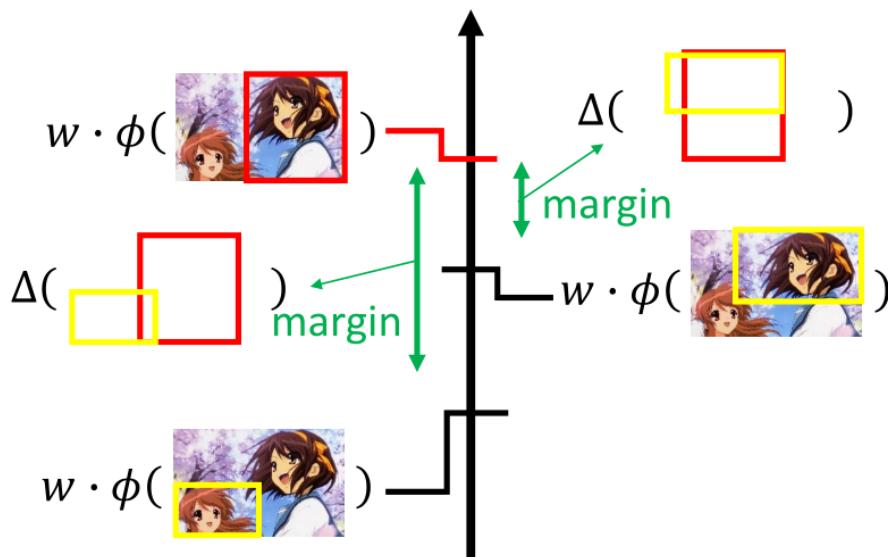
什么时候Cost最小？当真实值比最大的 $y+margin$ 的值还要大时，Cost最小。

$$C^n = \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

Curved arrow pointing from the first equation to the second:

$$C^n = \max_y [w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

$$C^n = \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$



In each iteration, pick a training data $\{x^n, \hat{y}^n\}$

$$\frac{\tilde{y}^n}{\bar{y}^n} = \underset{y}{\operatorname{argmax}} [w \cdot \phi(x^n, y)] \underset{y}{\operatorname{argmax}} [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)]$$

Oh no! Problem 2.1

$$\nabla C^n(w) = \phi(x^n, \cancel{\tilde{y}^n}) - \phi(x^n, \hat{y}^n)$$

$$w \rightarrow w - \eta [\phi(x^n, \cancel{\tilde{y}^n}) - \phi(x^n, \hat{y}^n)]$$

Another Viewpoint

我们也可以从另外一个角度来分析，最小化新的目标函数，其实就是最小化训练集里的损失上界，我们想最小化我们的最大y和真实y之间的差距本来是这样的，假设我们的output是 \tilde{y} ，希望minimize C' 。

但是这个很难，因为 Δ 可能是任何的函数，比如阶梯状函数，就不好微分了，梯度下降法就不好做了，比如语音识别，就算 w 有改变，但是 Δ 不一定就有改变，可能要到某个点上才可能会出现变化。所以我们就最小化它的上界，或许没办法让他变小，至少不会变大。

Another Viewpoint

$$\tilde{y}^n = \underset{y}{\operatorname{argmax}} w \cdot \phi(x^n, y)$$

- Minimizing the new cost function is minimizing the upper bound of the errors on training set

$$C' = \sum_{n=1}^N \Delta(\hat{y}^n, \tilde{y}^n) \leq C = \sum_{n=1}^N C^n \quad \text{upper bound}$$

We want to find w minimizing C' (errors)

It is hard!

Because y can be any kind of objects, $\Delta(\cdot, \cdot)$ can be any function

C serves as the surrogate of C'

Proof that $\Delta(\hat{y}^n, \tilde{y}^n) \leq C^n$

那接下来就是证明上面的式子为什么最小化新的代价函数，就是在最小化训练集上误差的上界：

$$C' = \sum_{n=1}^N \Delta(\hat{y}^n, \tilde{y}^n) \leq C = \sum_{n=1}^N C^n$$

只需要证明：

$$\Delta(\hat{y}^n, \tilde{y}^n) \leq C^n$$

$$C^n = \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

Proof that $\Delta(\hat{y}^n, \tilde{y}^n) \leq C^n$

$$\begin{aligned}\Delta(\hat{y}^n, \tilde{y}^n) &\leq \Delta(\hat{y}^n, \tilde{y}^n) + [w \cdot \phi(x^n, \tilde{y}^n) - w \cdot \phi(x^n, \hat{y}^n)] \\ &\quad \tilde{y}^n = \arg \max_y w \cdot \phi(x^n, y) \geq 0 \\ &= [\Delta(\hat{y}^n, \tilde{y}^n) + w \cdot \varphi(x^n, \tilde{y}^n)] - w \cdot \varphi(x^n, \hat{y}^n) \\ &\leq \max_y [\Delta(\hat{y}^n, y) + w \cdot \varphi(x^n, y)] - w \cdot \varphi(x^n, \hat{y}^n) \\ &= C^n\end{aligned}$$

More Cost Functions

也可以满足下式

$$\Delta(\hat{y}^n, \tilde{y}^n) \leq C^n$$

- Margin Rescaling(间隔调整)

$$C^n = \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

- Slack Variable Rescaling(松弛变量调整)

$$C^n = \max_y \Delta(\hat{y}^n, y) [1 + w \cdot \phi(x^n, y) - w \cdot \phi(x^n, \hat{y}^n)]$$

Regularization

训练数据和测试数据可以有不同的分布；

如果 w 与 0 比较接近，那么我们就可以最小化误差匹配的影响；

即在原来的基础上，加上一个正则项 $\frac{1}{2} \|w\|^2$ ， λ 为权衡参数；

$$C = \sum_{n=1}^N C^n \Rightarrow C = \lambda \sum_{n=1}^N C^n + \frac{1}{2} \|w\|^2$$

Training data and testing data can have different distribution.

w close to zero can minimize the influence of mismatch.

Keep the incorrect answer from a margin depending on errors

$$C = \sum_{n=1}^N C^n \longrightarrow$$

$$\begin{aligned} C^n \\ &= \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)] \\ &\quad - w \cdot \phi(x^n, \hat{y}^n) \end{aligned}$$

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N C^n$$

Regularization:
Find the w close to zero

每次迭代，选择一个训练数据 $\{x^n, \hat{y}^n\}$

$$C = \sum_{n=1}^N C^n \longrightarrow C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N C^n$$

In each iteration, pick a training data $\{x^n, \hat{y}^n\}$

$$\bar{y}^n = \operatorname{argmax}_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)]$$

$$\nabla C^n = \phi(x^n, \bar{y}^n) - \phi(x^n, \hat{y}^n) + w$$

$$w \rightarrow w - \eta [\phi(x^n, \bar{y}^n) - \phi(x^n, \hat{y}^n)] - \eta w$$

$$= (1 - \eta)w - \eta [\phi(x^n, \bar{y}^n) - \phi(x^n, \hat{y}^n)]$$

Weight decay as in DNN

得到的结果类似于DNN中的weight decay

Structured SVM

Find w minimizing C

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N C^n$$

$$C^n = \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

$$C^n + w \cdot \phi(x^n, \hat{y}^n) = \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)]$$

Are they equivalent?

We want to minimize C

For $\forall y$:

$$C^n + w \cdot \phi(x^n, \hat{y}^n) \geq \Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)$$

$$w \cdot \phi(x^n, \hat{y}^n) - w \cdot \phi(x^n, y) \geq \Delta(\hat{y}^n, y) - C^n$$

注意：第二个蓝色箭头并不完全等价，当最小化 C^n 时等价。

一般我们将 C^n 用 ε^n 代替之，表示松弛变量，此时条件变成了 Find $w, \varepsilon^1, \dots, \varepsilon^N$ minimizing C

Find w minimizing C

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N C^n$$

$$C^n = \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)] - w \cdot \phi(x^n, \hat{y}^n)$$

Find $w, \varepsilon^1, \dots, \varepsilon^N$ minimizing C

III

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

For $\forall n$:

Slack variable

For $\forall y$:

$$w \cdot \phi(x^n, \hat{y}^n) - w \cdot \phi(x^n, y) \geq \Delta(\hat{y}^n, y) - \varepsilon^n$$

单独讨论 $y = \hat{y}^n$ 时的情况，得到新的表达式

Find $w, \varepsilon^1, \dots, \varepsilon^N$ minimizing C

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

For $\forall n$:

For $\forall y$:

$$w \cdot \phi(x^n, \hat{y}^n) - w \cdot \phi(x^n, y) \geq \Delta(\hat{y}^n, y) - \varepsilon^n$$

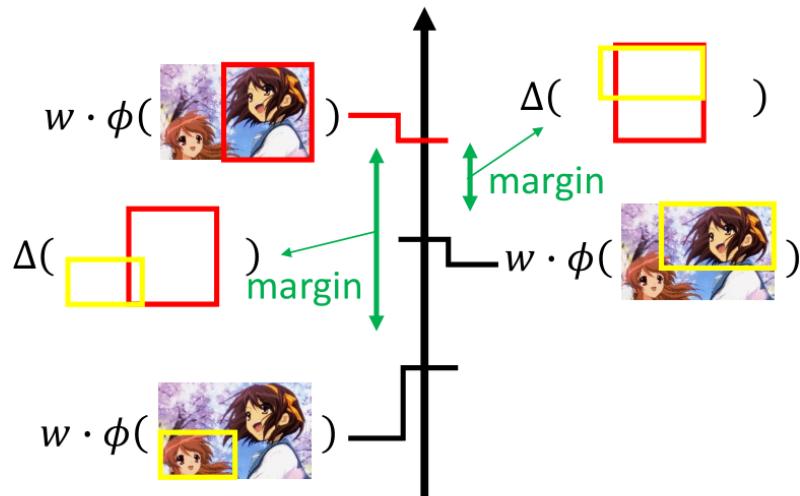
For $\forall y \neq \hat{y}^n$:

$$w \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, y)) \geq \Delta(\hat{y}^n, y) - \varepsilon^n, \quad \varepsilon^n \geq 0$$

If $y = \hat{y}^n$: $w \cdot \phi(x^n, \hat{y}^n) - w \cdot \phi(x^n, \hat{y}^n) \geq \Delta(\hat{y}^n, \hat{y}^n) - \varepsilon^n$
 $= 0 - 0 \rightarrow \varepsilon^n \geq 0$

Intuition

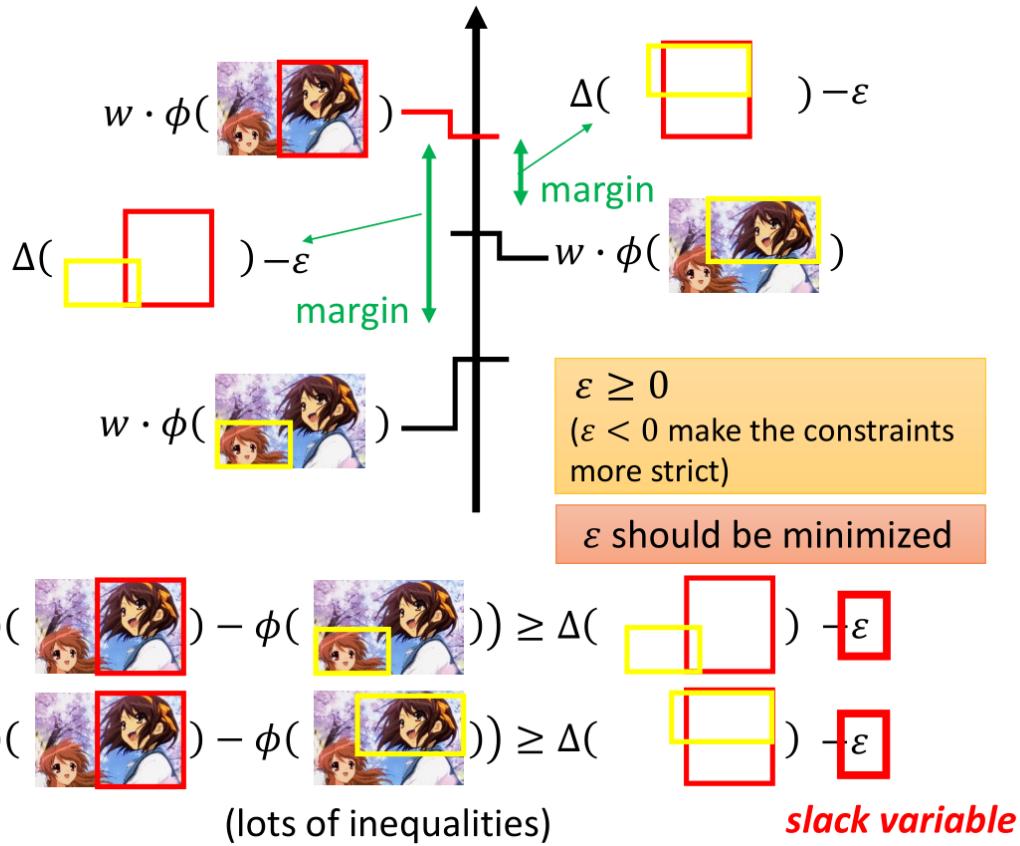
我们希望分数差大于margin



It is possible that no w can achieve this.

$$\left. \begin{aligned} w \cdot (\phi(\text{[red box]}) - \phi(\text{[yellow box]})) &\geq \Delta(\text{[red box]}) \\ w \cdot (\phi(\text{[red box]}) - \phi(\text{[yellow box]})) &\geq \Delta(\text{[yellow box]}) \end{aligned} \right\} \text{lots of inequalities} \quad \text{margin} \quad \forall y \neq \hat{y}$$

我们可能找不到一个 w 满足以上所有的不等式都成立。



因此将margin减去一个 ε (为了放宽限制, 但限制不应过宽, 否则会失去意义, ε 越小越好, 且要大于等于0)

假设, 我们现在有两个训练数据: (x^1, \hat{y}^1) 和 (x^2, \hat{y}^2)

对于 x^1 而言, 我们希望正确的分数减去错误的分数大于它们之间的 Δ 减去 ε^1 , 同时满足 $\varepsilon^1 \geq 0$

对于 x^2 而言, 同理, 我们希望正确的分数减去错误的分数, 要求大于它们之间的 Δ 减去 ε^2 , 同时满足: $\varepsilon^2 \geq 0$

在满足以上这些不等式的前提之下, 我们希望 $\lambda \sum_{n=1}^2 \varepsilon^n$ 是最小的, 同时加上对应的正则项也满足最小化。

Training data: \hat{y}^1 \hat{y}^2

Minimize $\frac{1}{2}\|w\|^2 + \lambda \sum_{n=1}^2 \varepsilon^n$

For x^1

$$w \cdot (\phi(\text{[Image]}) - \phi(\text{[Image]})) \geq \Delta(\text{[Image]} - \varepsilon^1) \quad \left. \begin{array}{l} \text{[Image]} \\ \text{[Image]} \end{array} \right\} \forall y \neq \hat{y}^1$$

$$w \cdot (\phi(\text{[Image]}) - \phi(\text{[Image]})) \geq \Delta(\text{[Image]} - \varepsilon^1) \quad \left. \begin{array}{l} \text{[Image]} \\ \text{[Image]} \end{array} \right\} \forall y \neq \hat{y}^1$$

(lots of inequalities) $\varepsilon^1 \geq 0$

For x^2

$$w \cdot (\phi(\text{[Image]}) - \phi(\text{[Image]})) \geq \Delta(\text{[Image]} - \varepsilon^2) \quad \left. \begin{array}{l} \text{[Image]} \\ \text{[Image]} \end{array} \right\} \forall y \neq \hat{y}^2$$

$$w \cdot (\phi(\text{[Image]}) - \phi(\text{[Image]})) \geq \Delta(\text{[Image]} - \varepsilon^2) \quad \left. \begin{array}{l} \text{[Image]} \\ \text{[Image]} \end{array} \right\} \forall y \neq \hat{y}^2$$

(lots of inequalities) $\varepsilon^2 \geq 0$

我们的目标是，求得 $w, \varepsilon^1, \dots, \varepsilon^N$ ，最小化 C

$$C = \frac{1}{2}\|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

同时，要满足：

对所有的训练样本的所有不是正确答案的标记， $w \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, y)) \geq \Delta(\hat{y}^n, y) - \varepsilon^n, \varepsilon^n \geq 0$

Find $w, \varepsilon^1, \dots, \varepsilon^N$ minimizing C

$$C = \frac{1}{2}\|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

For $\forall n$:

For $\forall y \neq \hat{y}^n$:

$$w \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, y)) \geq \Delta(\hat{y}^n, y) - \varepsilon^n, \varepsilon^n \geq 0$$

Solve it by the solver in SVM package

Quadratic Programming (QP) Problem

Too many constraints

可以利用**SVM包**中的solver来解决以上的问题；是一个二次规划(Quadratic Programming **QP**)的问题；但是约束条件过多，需要通过切割平面算法(**Cutting Plane Algorithm**)解决受限的问题。

Cutting Plane Algorithm for Structured SVM

在 w 和 ε^i 组成的参数空间中，颜色表示C的值，在没有限制的情况下， w 和 ε 越小越好，在有限制的情况下，只有内嵌的多边形区域内是符合约束条件的，因此需要在该区域内寻找最小值，即

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

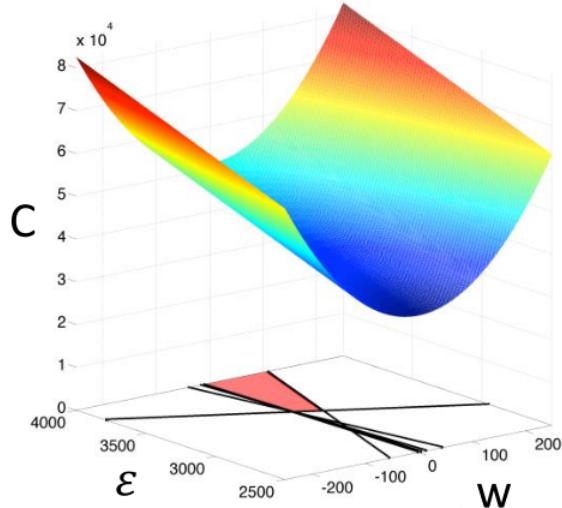
Find $w, \varepsilon^1, \dots, \varepsilon^N$ minimizing C

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

For $\forall n$:

For $\forall y \neq \hat{y}^n$:

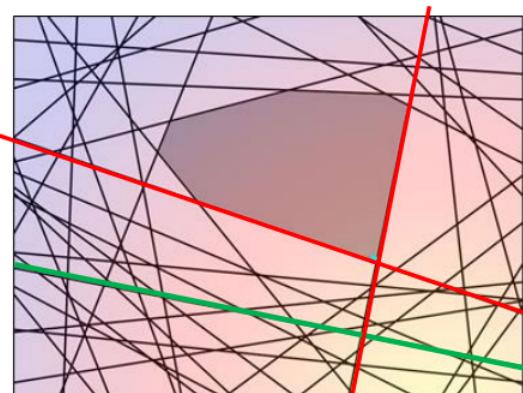
$$w \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, y)) \geq \Delta(\hat{y}^n, y) - \varepsilon^n, \quad \varepsilon^n \geq 0$$



Source of image:
http://abnerguzman.com/publications/gkb_aistats13.pdf

Cutting Plane Algorithm

Although there are lots of constraints, most of them do not influence the solution.



Parameter space
 $(w, \varepsilon^1, \dots, \varepsilon^N)$

Red lines: determine the solution
 Green line: Remove this constraint will not influence the solution

$y \in \mathbb{A}^n$

For $\forall r, \forall y, y \neq \hat{y}^n$:

- $w \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, y)) \geq \Delta(\hat{y}^n, y) - \varepsilon^n$
- $\varepsilon^n \geq 0$

\mathbb{A}^n : a very small set of $y \rightarrow$ working set

虽然有很多约束条件，但它们中的大多数的约束都是冗元，并不影响问题的解决；

原本是穷举 $y \neq \hat{y}^n$ ，而现在我们需要移除那些不起作用的线条，保留有用的线条，这些有影响的线条集可以理解为Working Set，用 \mathbb{A}^n 表示。

Elements in working set \mathbb{A}^n is selected iteratively

- Elements in **working set \mathbb{A}^n** is selected iteratively

Initialize $\mathbb{A}^1 \dots \mathbb{A}^N$

Find $w, \varepsilon^1 \dots \varepsilon^N$ minimizing C

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

Solve a QP problem

For $\forall r$:

For $\forall y \in \mathbb{A}^n, y \neq \hat{y}^n$:

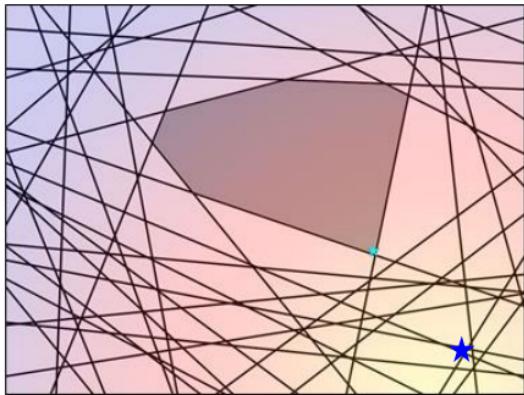
$$w \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, y)) \geq \Delta(\hat{y}^n, y) - \varepsilon^n \quad \varepsilon^n \geq 0$$

obtain solution w

Repeatedly

Add elements into $\mathbb{A}^1 \dots \mathbb{A}^N$

Strategies of adding elements into working set \mathbb{A}^n

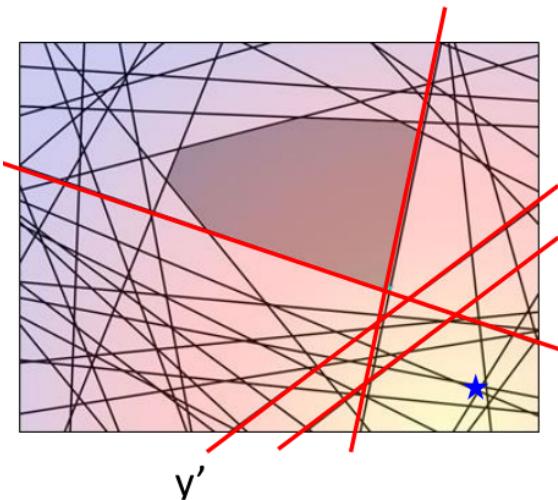


Initialize $\mathbb{A}^n = \text{null}$

No constraint at all

Solving QP

The solution w is
the blue point.



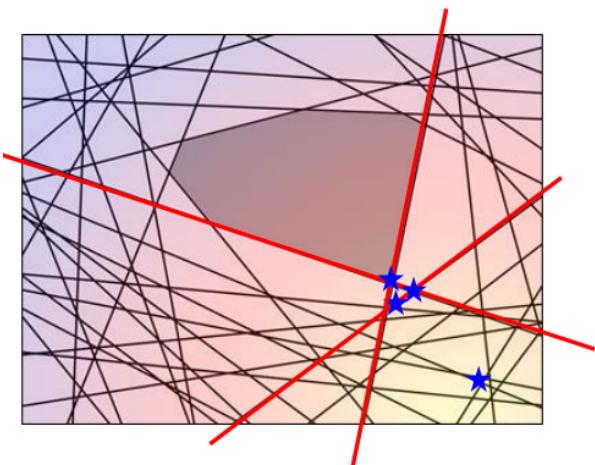
There are lots of constraints
is violated

Find **the most violated one**

Suppose it is the constraint
from y'

Extent the working set

$$\mathbb{A}^n = \mathbb{A}^n \cup \{y'\}$$



假设 \mathbb{A}^n 初始值为空集合null，即没有任何约束限制，求解QP的结果就是对应的蓝点，但是不能满足条件的线条有很多很多，我们现在只找出没有满足的最“严重的”那一个即可。那么我们就把 $\mathbb{A}^n = \mathbb{A}^n \cup \{y'\}$

根据新获得的Working Set中唯一的成员 y' ，找寻新的最小值，进而得到新的 w ，尽管得到新的 w 和最小值，但依旧存在不满足条件的约束，需要继续把最难搞定的限制添加到有效集中，再求解一次。得到新的 w ，直到所有难搞的线条均添加到Working Set之中，最终Working Set中有三个线条，根据这些线条确定求解区间内的point，最终得到问题的解。

Find the most violated one

Given w' and ε' from working sets at hand, which constraint is the most violated one?

Constraint: $w \cdot (\phi(x, \hat{y}) - \phi(x, y)) \geq \Delta(\hat{y}, y) - \varepsilon$

Violate a Constraint:

$$w' \cdot (\phi(x, \hat{y}) - \phi(x, y)) < \Delta(\hat{y}, y) - \varepsilon'$$

Degree of Violation

$$\begin{aligned} \Delta(\hat{y}, y) - \varepsilon' - w' \cdot (\phi(x, \hat{y}) - \phi(x, y)) \\ \xrightarrow{\text{blue arrow}} \Delta(\hat{y}, y) + w' \cdot \phi(x, y) \end{aligned}$$

The most violated one:

$$\arg \max_y [\Delta(\hat{y}, y) + w \cdot \phi(x, y)]$$

Cutting Plane Algorithm

- 给定训练数据集

$$\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$$

Working Set初始设定为

$$\mathbb{A}^1 \leftarrow \text{null}, \mathbb{A}^2 \leftarrow \text{null}, \dots, \mathbb{A}^N \leftarrow \text{null}$$

- 重复以下过程
 - 在初始的Working Set中求解一个QP问题的解，只需求解出w即可。
 - 针对求解出的w，要求对每一个训练数据 (x^n, \hat{y}^n) ，寻找最violated的限制，同时更新Working Set
- 直到Working Set中的元素不再发生变化，迭代终止，即得到要求解的w。

Given training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$

Working Set $\mathbb{A}^1 \leftarrow \text{null}, \mathbb{A}^2 \leftarrow \text{null}, \dots, \mathbb{A}^N \leftarrow \text{null}$

Repeat

$w \leftarrow$ Solve a **QP** with Working Set $\mathbb{A}^1, \mathbb{A}^2, \dots, \mathbb{A}^N$

QP: Find $w, \varepsilon^1 \dots \varepsilon^N$ minimizing $\frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$

For $\forall n$:

For $\forall y \in \mathbb{A}^n$:

$$w \cdot (\phi(x^n, \hat{y}^n) - \phi(x^n, y)) \geq \Delta(\hat{y}^n, y) - \varepsilon^n, \varepsilon^n \geq 0$$

Given training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$

Working Set $\mathbb{A}^1 \leftarrow null, \mathbb{A}^2 \leftarrow null, \dots, \mathbb{A}^N \leftarrow null$

Repeat

$w \leftarrow$ Solve a **QP** with Working Set $\mathbb{A}^1, \mathbb{A}^2, \dots, \mathbb{A}^N$

For each training data (x^n, \hat{y}^n) :

$$\bar{y}^n = \arg \max_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)]$$

find the most violated constraints

Update working set $\mathbb{A}^n \leftarrow \mathbb{A}^n \cup \{\bar{y}^n\}$

Until $\mathbb{A}^1, \mathbb{A}^2, \dots, \mathbb{A}^N$ doesn't change any more

Return w

Multi-class and binary SVM

Multi-class SVM

Multi-class SVM

$$F(x, y) = w \cdot \phi(x, y)$$

- Problem 1: Evaluation

- If there are K classes, then we have K weight vectors $\{w^1, w^2, \dots, w^K\}$

$$y \in \{1, 2, \dots, k, \dots, K\}$$

$$F(x, y) = w^y \cdot \vec{x}$$

\vec{x} : vector

representation of x

$$w = \begin{bmatrix} w^1 \\ w^2 \\ \vdots \\ w^k \\ \vdots \\ w^K \end{bmatrix} \quad \phi(x, y) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \vec{x} \\ \vdots \\ 0 \end{bmatrix}$$

- Problem 2: Inference

$$F(x, y) = w^y \cdot \vec{x}$$

$$\hat{y} = \arg \max_{y \in \{1, 2, \dots, k, \dots, K\}} F(x, y)$$

$$= \arg \max_{y \in \{1, 2, \dots, k, \dots, K\}} w^y \cdot \vec{x}$$

The number of classes are usually small, so we can just enumerate them.

Multi-class SVM

$$y \in \{\text{dog, cat, bus, car}\}$$

$$\Delta(\hat{y}^n = \text{dog}, y = \text{cat}) = 1$$

$$\Delta(\hat{y}^n = \text{dog}, y = \text{bus}) = 100$$

(defined as your wish)

- Problem 3: Training

Find $w, \varepsilon^1, \dots, \varepsilon^N$ minimizing C

$$C = \frac{1}{2} \|w\|^2 + \lambda \sum_{n=1}^N \varepsilon^n$$

For $\forall n$:

For $\forall y \neq \hat{y}^n$:

There are only $N(K-1)$ constraints.

$$(w^{\hat{y}^n} - w^y) \cdot \vec{x} \geq \underline{\Delta(\hat{y}^n, y)} - \varepsilon^n, \quad \varepsilon^n \geq 0$$

$$w \cdot \phi(x^n, \hat{y}^n) = w^{\hat{y}^n} \cdot \vec{x}$$

$$w \cdot \phi(x^n, y) = w^y \cdot \vec{x}$$

Some types of misclassifications may be worse than others.

Binary SVM

- Set $K = 2 \quad y \in \{1,2\}$

For $\forall y \neq \hat{y}^n:$ =1

$$(w^{\hat{y}^n} - w^y) \cdot \vec{x} \geq \underline{\Delta(\hat{y}^n, y)} - \varepsilon^n, \varepsilon^n \geq 0$$

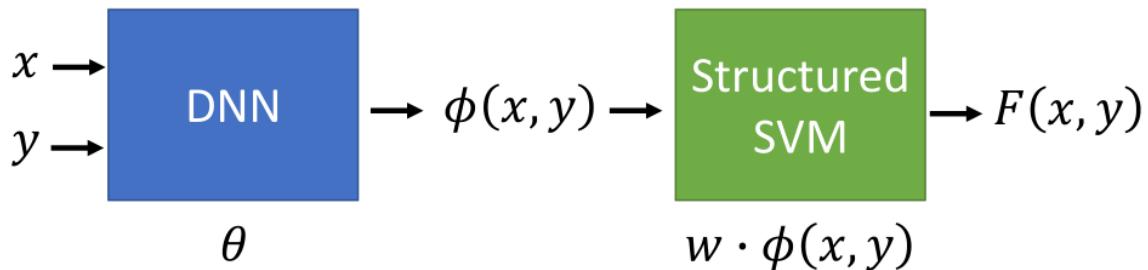
If $y=1:$ $(w^1 - w^2) \cdot \vec{x} \geq 1 - \varepsilon^n \quad \Rightarrow \quad w \cdot \vec{x} \geq 1 - \varepsilon^n$

If $y=2:$ $(w^2 - w^1) \cdot \vec{x} \geq 1 - \varepsilon^n \quad \Rightarrow \quad -w \cdot \vec{x} \geq 1 - \varepsilon^n$

Beyond Structured SVM

结构化SVM是线性结构的，如果想要结构化SVM的表现更好，我们需要定义一个较好的特征，但是人为设定特征往往十分困难，一个较好的方法是利用DNN生成特征，先用一个DNN，最后训练的结果往往十分有效。

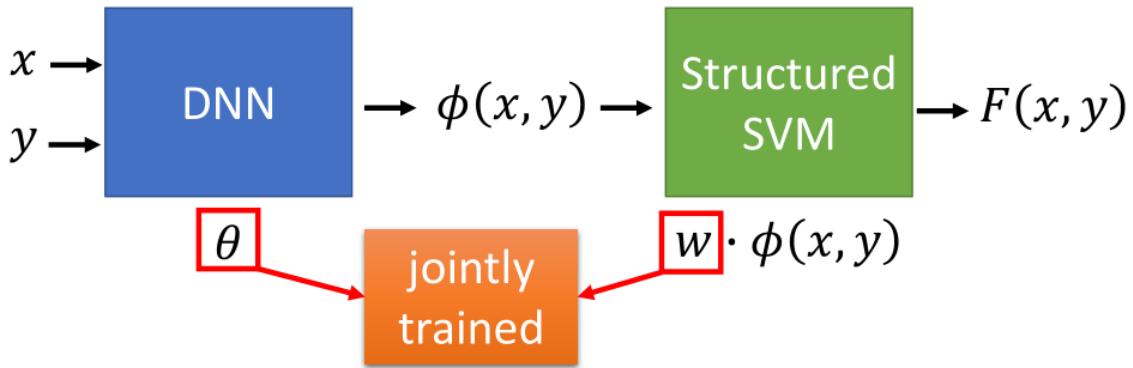
- Involving DNN when generating $\phi(x, y)$



Ref: Hao Tang, Chao-hong Meng, Lin-shan Lee, "An initial attempt for phoneme recognition using Structured Support Vector Machine (SVM)," ICASSP, 2010
 Shi-Xiong Zhang, Gales, M.J.F., "Structured SVMs for Automatic Speech Recognition," in Audio, Speech, and Language Processing, IEEE Transactions on, vol.21, no.3, pp.544-555, March 2013

将DNN与结构化SVM一起训练，同时更新DNN与结构化SVM中的参数。

- Jointly training structured SVM and DNN

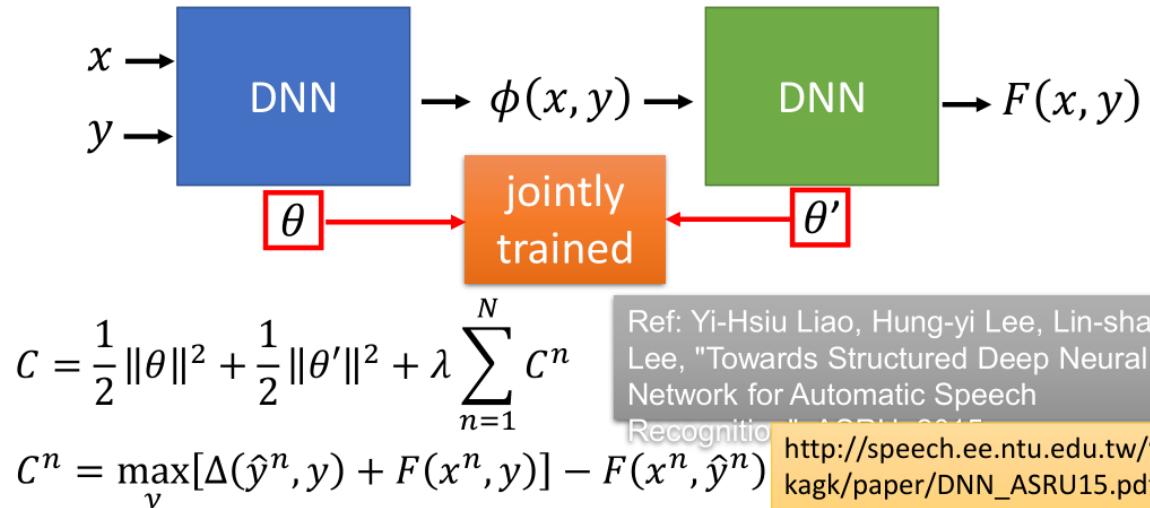


Ref: Shi-Xiong Zhang, Chaojun Liu, Kaisheng Yao, and Yifan Gong, "DEEP NEURAL SUPPORT VECTOR MACHINES FOR SPEECH RECOGNITION", Interspeech 2015

用一个DNN代替结构化SVM，即将 x 和 y 作为输入， $F(x, y)$ (为一个标量)作为输出。

- Replacing Structured SVM with DNN

A DNN with x and y as input and $F(x, y)$ (a scalar) as output



Sequence Labeling Problem

Sequence Labeling

$$f : X \rightarrow Y$$

序列标注的问题可以理解为：机器学习所要寻找的目标函数的输入是一个序列，输出也为一个序列，并且假设输入输出的序列长度相同，即输入可以写成序列向量的形式，输出也为序列向量。该任务可以利用循环神经网络来解决，但本章节我们可以基于结构化学习的其它方法进行解决(两步骤，三问题)。

Example Task

词性标记(POS tagging)

- 标记一个句子中每一个词的词性(名词、动词等等);
- 输入一个句子(比如, John saw the saw), 系统将会标记John为专有名词, saw为动词, the为限定词, saw为名词;
- 其在自然语言处理(NLP)中, 是非常典型且重要的任务, 也是许多文字理解的基石, 用于后续句法分析和词义消歧。

如果不考虑序列, 问题就无法解决(POS tagging仅仅依靠查表的方式是不够的, 比如Hash Table, 你需要知道一整个序列的信息, 才能有可能把每个词汇的词性找出)

- John saw the saw.
 - 第一个"saw"更有可能是动词V, 而不是名词N;
 - 然而, 第二个"saw"是名词N, 因为名词N更可能跟在限定词后面。

Hidden Markov Model (HMM)

How to generate a sentence?

Step 1

- 生成POS序列
- 基于语法(根据脑中内建的的语法)

假设你大脑中一个马尔科夫链, 开始说一句话时, 放在句首的词性有50%的可能性为冠词, 40%的可能性为专有名词, 10%的可能性为动词, 然后进行随机采样, 再从专有名词开始, 有80%的可能性后面为动词, 动词后面有25%的可能性为冠词, 冠词后面有95%的可能性为名词, 名词后面有10%的可能性句子就结束了。

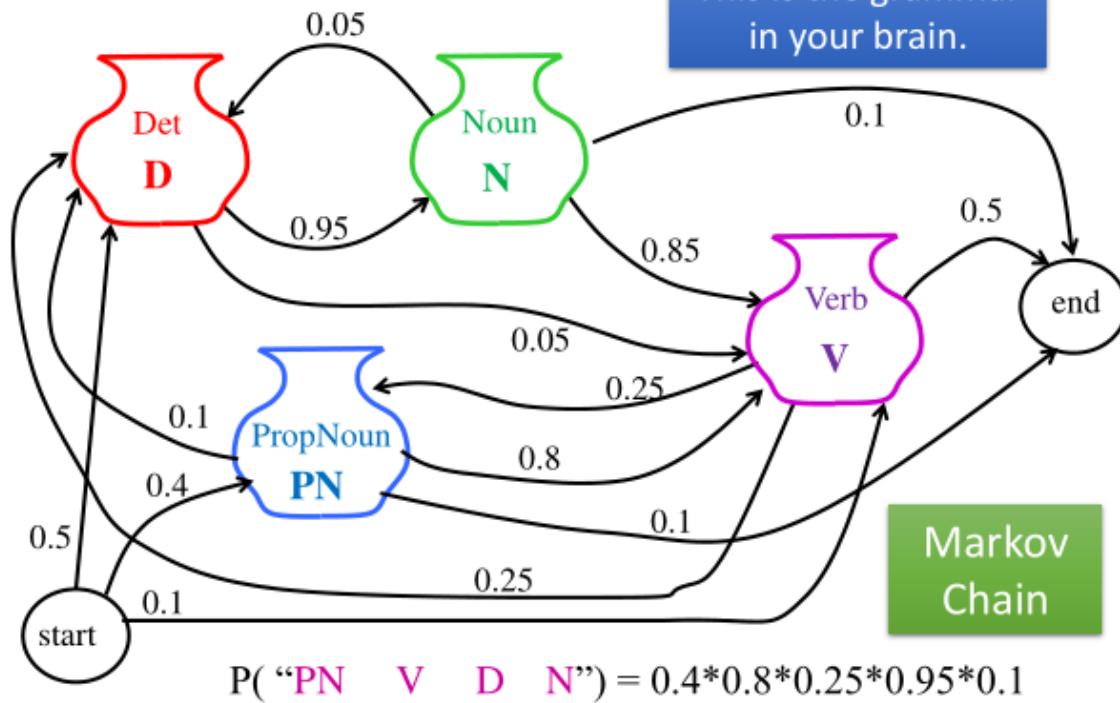
Step 2

- 根据词序生成一个句子
- 基于词典

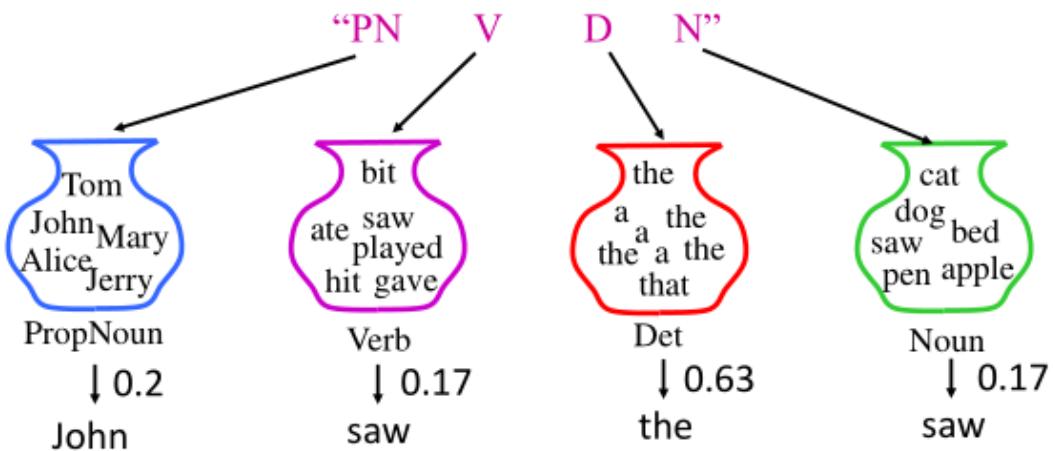
根据词性找到词典中中对应的词汇, 从不同的词性集合中采样出不同词汇所出现的机率。 HMM可以描述为利用POS标记序列得到对应句子的机率, 即

$$P(x, y) = P(y)P(x|y)$$

HMM – Step 1



HMM – Step 2



$$P(\text{"John saw the saw"} | \text{"PN V D N"}) \\ = 0.2 * 0.17 * 0.63 * 0.17$$

x : John saw the saw.

Y : PN V D N

对应于：

$$x = x_1, x_2 \dots x_L$$

$$y = y_1, y_2 \dots y_L$$

其中，

$$P(x, y) = P(y)P(x|y)$$

- Step1(Transition probability)

$$P(y) = P(y_1|start) \times \prod_{l=1}^{L-1} P(y_{l+1}|y_l) \times P(end|y_L)$$

- Step2(Emission probability)

$$P(x|y) = \prod_{l=1}^L P(x_l|y_l)$$

Estimating the probabilities

- 我们如何知道 $P(V|PN)$, $P(saw|V)$?

- 从训练数据中得到

$$P(x, y) = P(y_1|start) \prod_{l=1}^{L-1} P(y_{l+1}|y_l) P(end|y_L) \prod_{l=1}^L P(x_l|y_l)$$

其中, 计算 $y_l = s$, 下一个标记为 s' 的机率, 就等价于现在训练集里面 s 出现的次数除去 s 后面跟 s' 的次数;

$$\frac{P(y_{l+1} = s'|y_l = s)}{(s \text{ and } s' \text{ are tags })} = \frac{\text{count}(s \rightarrow s')}{\text{count}(s)}$$

计算某一个标记为 s 所产生的词为 t 的机率, 就等价于 s 在整个词汇中出现的次数除去某个词标记为 t 的次数。

$$\frac{P(x_l = t|y_l = s)}{(s \text{ is tag, and } t \text{ is word })} = \frac{\text{count}(s \rightarrow t)}{\text{count}(s)}$$

How to do POS Tagging?

We can compute $P(x, y)$

给定 x (Observed), 发现 y (Hidden), 即如何计算 $P(x, y)$ 的问题

given x , find y

$$\begin{aligned} y &= \arg \max_{y \in Y} P(y|x) \\ &= \arg \max_{y \in Y} \frac{P(x, y)}{P(x)} \\ &= \arg \max_{y \in Y} P(x, y) \end{aligned}$$

Viterbi Algorithm

$$\tilde{y} = \arg \max_{y \in Y} P(x, y)$$

- 穷举所有可能的 y
 - 假设有 $|S|$ 个标记, 序列 y 的长度为 L ;
 - 有可能的 y 即 $|s|^L$ (空间极为庞大)。
- 利用维特比算法解决此类问题
 - 复杂度为:

$$O(L|S|^2)$$

HMM - Summary

Evaluation

$$F(x, y) = P(x, y) = P(y)P(x|y)$$

该评估函数可以理解为x与y的联合概率。

Inference

$$\tilde{y} = \arg \max_{y \in \mathbb{Y}} P(x, y)$$

给定一个x，求出最大的y，使得我们定义函数的值达到最大(即维特比算法)。

Training

从训练数据集中得到 $P(y)$ 与 $P(x|y)$

该过程就是计算机率的问题或是统计语料库中词频的问题。

HMM - Drawbacks

- 在推理过程

$$\tilde{y} = \arg \max_{y \in \mathbb{Y}} P(x, y)$$

把求解最大的y作为我们的输出值。

- 为了得到正确的结果，我们需要让

$$(x, \hat{y}) : P(x, \hat{y}) > P(x, y)$$

但是HMM可能无法处理这件事情，它不能保证错误的y带进去得到的 $P(x,y)$ 一定是小的。

- Inference:

$$\tilde{y} = \arg \max_{y \in \mathbb{Y}} P(x, y)$$

- To obtain correct results ...

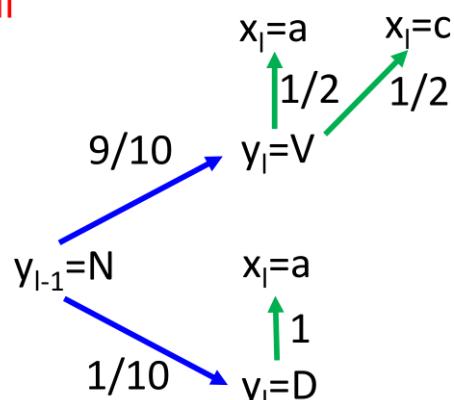
(x, \hat{y}) : $P(x, \hat{y}) > \underline{P(x, y)}$ Can HMM guarantee that?
not necessarily small

Transition probability:

$$P(V|N)=9/10 \quad P(D|N)=1/10 \quad \dots$$

Emission probability:

$$P(a|V)=1/2 \quad P(a|D)=1 \quad \dots$$



假设我们知道在 $l-1$ 时刻词性标记为N，即 $y_{l-1} = N$ ，在 l 时刻我们看到的单词为a，现在需要求出 $y_l = ?$

根据计算可以得到V的机率是0.45，D的机率是0.1。但是如果测试数据中有9个 $N \rightarrow V \rightarrow c$, 9个 $P \rightarrow V \rightarrow a$, 1个 $N \rightarrow D \rightarrow a$, 里面有存有和训练数据一样的数据，因此D更合理。

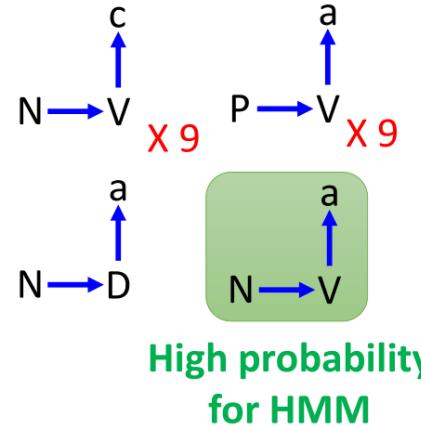
- The (x, y) never seen in the training data can have large probability $P(x, y)$.

- Benefit:

- When there is only little training data

➤ More complex model can deal with this problem

➤ However, CRF can deal with this problem based on the same model



通常情况下，隐马尔可夫模型是判断**未知数据**出现的**最大可能性**，即 (x, y) 在训练数据中从未出现过，但也可能有较大的概率 $P(x, y)$ ；

当训练数据很少的时候，使用隐马尔可夫模型，其性能表现是可行的，但当训练集很大时，性能表现较差；

隐马尔可夫模型会产生**未卜先知**的情况，是因为转移概率和发散概率，在训练时是分开建模的，两者是相互独立的，我们也可以用一个更复杂的模型来模拟两个序列之间的可能性，但要避免过拟合。

条件随机场的模型和隐马尔可夫模型是一样的，同时可以克服隐马尔可夫模型的缺点。

Conditional Random Field (CRF)

$$P(x, y) \propto \exp(w \cdot \phi(x, y))$$

条件随机场模型描述的也是 $P(x, y)$ 的问题，但与HMM表示形式很不一样(本质上是在训练阶段不同)，其机率正比于 $\exp(w \cdot \phi(x, y))$ 。

- $\phi(x, y)$ 为一个特征向量；
- w 是一个权重向量，可以从训练数据中学习得到；
- $\exp(w \cdot \phi(x, y))$ 总是正的，可能大于1。

$$P(x, y) = \frac{\exp(w \cdot \phi(x, y))}{R}$$

$$P(y|x) = \frac{P(x, y)}{\sum_{y'} P(x, y')} = \frac{\exp(w \cdot \phi(x, y))}{\sum_{y' \in \mathbb{Y}} \exp(w \cdot \phi(x, y'))} = \frac{\exp(w \cdot \phi(x, y))}{Z(x)}$$

其中 $\sum_{y' \in \mathbb{Y}} \exp(w \cdot \phi(x, y'))$ 仅与 x 有关，与 y 无关

$P(x, y)$ for CRF

- HMM

$$P(x, y) = P(y_1 | start) \prod_{l=1}^{L-1} P(y_{l+1} | y_l) P(end | y_L) \prod_{l=1}^L P(x_l | y_l)$$

取对数

$$\begin{aligned} & \log P(x, y) \\ &= \log P(y_1 | \text{start}) + \sum_{l=1}^{L-1} \log P(y_{l+1} | y_l) + \log P(\text{end} | y_L) \\ & \quad + \sum_{l=1}^L \log P(x_l | y_l) \end{aligned}$$

其中,

$$\sum_{l=1}^L \log P(x_l | y_l) = \sum_{s,t} \log P(t|s) \times N_{s,t}(x, y)$$

- $\sum_{s,t}$ 穷举所有可能的标记s和所有可能的单词t;
- $\log P(t|s)$ 表示给定标记s的得到单词t的概率取对数
- $N_{s,t}(x, y)$ 表示为单词t被标记成s的事情, 在(x, y)对中总共出现的次数。

Example

$x: \text{The dog ate the homework.}$ $y: \begin{array}{ccccc} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \text{D} & \text{N} & \text{V} & \text{D} & \text{N} \end{array}$		$N_{D,\text{the}}(x, y) = 2$ $N_{N,\text{dog}}(x, y) = 1$ $N_{V,\text{ate}}(x, y) = 1$ $N_{N,\text{homework}}(x, y) = 1$ $N_{s,t}(x, y) = 0$ (for any other s and t)
---	--	---

$$\begin{aligned} & \sum_{l=1}^L \log P(x_l | y_l) \\ &= \cancel{\log P(\text{the}|D)} + \log P(\text{dog}|N) + \log P(\text{ate}|V) \\ & \quad + \cancel{\log P(\text{the}|D)} + \log P(\text{homework}|N) \\ &= \cancel{\log P(\text{the}|D)} \times 2 + \log P(\text{dog}|N) \times 1 + \log P(\text{ate}|V) \times 1 \\ & \quad + \log P(\text{homework}|N) \times 1 \\ &= \sum_{s,t} \log P(t|s) \times N_{s,t}(x, y) \end{aligned}$$

每个单词都已经标记成对应的词性, 我们分别计算出 D, N, V 在(x, y)对中出现的次数

然后计算所有的机率相乘的结果 $\sum_{l=1}^L \log P(x_l | y_l)$, 如上图, 整理之后的结果为
 $\sum_{s,t} \log P(t|s) \times N_{s,t}(x, y)$

$$\begin{aligned}
logP(x, y) &= logP(y_1|start) + \sum_{l=1}^{L-1} logP(y_{l+1}|y_l) + logP(end|y_L) \\
&\quad + \sum_{l=1}^L logP(x_l|y_l) \\
logP(y_1|start) &= \sum_s logP(s|start) \times N_{start,s}(x, y) \\
\sum_{l=1}^{L-1} logP(y_{l+1}|y_l) &= \sum_{s,s'} logP(s'|s) \times N_{s,s'}(x, y) \\
logP(end|y_L) &= \sum_s logP(end|s) \times N_{s,end}(x, y)
\end{aligned}$$

分析 $logP(x, y)$ 的其他项

其中，黄色表示对所有词性s放在句首的机率取对数，再乘上在 (x, y) 对中，s放在句首所出现的次数；

绿色表示计算s后面跟s'在 (x, y) 里面所出现的次数，再乘上s后面跟s'的机率取对数；

紫色同理，最后一项表示两项相乘的形式。

则有 $logP(x, y)$

P(x, y) for CRF

$$\begin{aligned}
logP(x, y) &= \sum_{s,t} logP(t|s) \times N_{s,t}(x, y) \\
&\quad + \sum_s logP(s|start) \times N_{start,s}(x, y) \\
&\quad + \sum_{s,s'} logP(s'|s) \times N_{s,s'}(x, y) \\
&\quad + \sum_s logP(end|s) \times N_{s,end}(x, y) \\
&= \begin{bmatrix} \vdots \\ logP(t|s) \\ \vdots \\ logP(s|start) \\ \vdots \\ logP(s'|s) \\ \vdots \\ logP(end|s) \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} \vdots \\ N_{s,t}(x, y) \\ \vdots \\ N_{start,s}(x, y) \\ \vdots \\ N_{s,s'}(x, y) \\ \vdots \\ N_{s,end}(x, y) \\ \vdots \end{bmatrix} \\
&= w \cdot \phi(x, y) \\
P(x, y) &= exp(w \cdot \phi(x, y))
\end{aligned}$$

等价于两个向量做内积，进而可以用 $logP(x, y) = w \cdot \phi(x, y)$ 表示，第二个向量每一个element是依赖于 (x, y) 的，因此可以写成 $\phi(x, y)$

由此可知， $P(x, y) = exp(w \cdot \phi(x, y))$ ，其中每一个w，都对应着HMM模型中的某一个机率取对数。

因此对于每一个 w , 取exponential就可以变为机率。但是我们在训练时对 w 没有任何限制, 得到 w 大于0时, 机率会大于一。

因此需要把 $P(x, y)$ 表达式变化为 $P(x, y) \propto \exp(w \cdot \phi(x, y))$

$$P(x, y) \propto \exp(w \cdot \phi(x, y))$$

$$\phi(x, y) = \begin{bmatrix} \vdots \\ N_{s,t}(x, y) \\ \vdots \\ \vdots \\ N_{start,s}(x, y) \\ \vdots \\ \vdots \\ N_{s,s'}(x, y) \\ \vdots \\ \vdots \\ N_{s,end}(x, y) \\ \vdots \end{bmatrix}$$

$$w = \begin{bmatrix} \vdots \\ w_{s,t} \\ \vdots \\ w_{start,s} \\ \vdots \\ \vdots \\ w_{s,s'} \\ \vdots \\ \vdots \\ w_{s,end} \\ \vdots \end{bmatrix}$$

However, we do not give w any constraints during training

- $\rightarrow \log P(x_i = t | y_i = s)$
- $P(x_i = t | y_i = s) = e^{w_{s,t}}$ means
- $\rightarrow \log P(s | start)$
- $P(s | start) = e^{w_{start,s}}$ means
- $\rightarrow \log P(y_i = s' | y_{i-1} = s)$
- $P(y_i = s' | y_{i-1} = s) = e^{w_{s,s'}}$ means
- $\rightarrow \log P(end | s)$
-

Feature Vector

$\phi(x, y)$ 的形式是什么样的? $\phi(x, y)$ 分为两部分

Part 1: relations between tags and words

Feature Vector

- What does $\phi(x, y)$ look like?

x: The dog ate the homework.
 ↓ ↓ ↓ ↓ ↓
 y: D N V D N

- $\phi(x, y)$ has two parts

- Part 1: relations between tags and words

- Part 2: relations between tags

If there are $|S|$ possible tags,
 $|L|$ possible words

Part 1 has $|S| \times |L|$ dimensions

Part 1	Value
D, the	2
D, dog	0
D, ate	0
D, homework	0
.....
N, the	0
N, dog	1
N, ate	0
N, homework	1
.....
V, the	0
V, dog	0
V, ate	1
V, homework	0
.....

如果有 $|S|$ 个可能的标记, $|L|$ 个可能的单词, Part 1的维度为 $|S| \times |L|$, value表示在(标记, 单词)对中出现的次数, 所以这是一个维度很大的稀疏vector;

Feature Vector

- What does $\phi(x, y)$ look like?

X: The dog ate the homework.
 ↓ ↓ ↓ ↓ ↓
 y: D N V D N

- $\phi(x, y)$ has two parts

- Part 1: relations between tags and words

- Part 2: relations between tags

$N_{s,s'}(x, y)$: Number of tags s and s' consecutively in (x, y)

Part 2	Value
$N_{D,D}(x, y) \rightarrow D, D$	0
$N_{D,N}(x, y) \rightarrow D, N$	2
D, V	0
.....
N, D	0
N, N	0
N, V	1
.....
V, D	1
V, N	0
V, V	0
.....
Start, D	1
Start, N	0
.....
End, D	0
End, N	1

定义 $N_{s,s'}(x, y)$: 为标记 s 和 s' 在 (x, y) 对中连续出现的次数，如果有 $|S|$ 个可能的标记，这部分向量维度为 $|S| \times |S| + 2|S|$ (s 之间、start、end)。

CRF 中可以自己定义 $\phi(x, y)$

CRF – Training Criterion

给定训练数据：

$$\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$$

找到一个权重向量 w^* 去最大化目标函数 $O(w)$ ；

其中， w^* 与目标函数定义如下：

$$w^* = \arg \max_w O(w)$$

$$O(w) = \sum_{n=1}^N \log P(\hat{y}^n | x^n)$$

表示为我们要寻找一个 w ，使得最大化给定的 x_n 所产生 \hat{y}^n 正确标记的机率，再取对数进行累加，此处可以联想到交叉熵也是最大化正确维度的机率再取对数，只不过此时是针对整个序列而言的。

对 $\log P(y|x)$ 做相应的转换

$$P(y|x) = \frac{P(x,y)}{\sum_{y'} P(x,y')}$$

$$\log P(\hat{y}^n | x^n) = \log P(x^n, \hat{y}^n) - \log \sum_{y'} P(x^n, y')$$

CRF – Training Criterion

- Given training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \dots, (x^N, \hat{y}^N)\}$
- Find the weight vector w^* maximizing objective function $O(w)$:

$$w^* = \arg \max_w O(w) \quad O(w) = \sum_{n=1}^N \log P(\hat{y}^n | x^n)$$

$$\log P(\hat{y}^n | x^n) = \log \underline{P(x^n, \hat{y}^n)} - \log \sum_{y'} \overline{P(x^n, y')}$$

Maximize what we observe

Minimize what we don't observe

根据CRF的定义可知，可以分解为两项再分别取对数，即最大化观测到的机率，最小化没有观测到的机率。

Gradient Ascent

梯度下降：找到一组参数 θ ，最小化成本函数 $C(\theta)$ ，即梯度的反方向

$$\theta \rightarrow \theta - \eta \nabla C(\theta)$$

梯度上升：找到一组参数 θ ，最大化成本函数 $O(\theta)$ ，即梯度的同方向

$$\theta \rightarrow \theta + \eta \nabla O(\theta)$$

Gradient descent

Find a set of parameters θ minimizing cost function $C(\theta)$

$$\theta \rightarrow \theta - \eta \nabla C(\theta)$$

Opposite direction of the gradient

Gradient Ascent

Find a set of parameters θ maximizing objective function $O(\theta)$

$$\theta \rightarrow \theta + \eta \nabla O(\theta)$$

The same direction of the gradient

CRF - Training

$$O(w) = \sum_{n=1}^N \log P(\hat{y}^n | x^n) = \sum_{n=1}^N O^n(w)$$

Compute $\nabla O^n(w) = \begin{bmatrix} \vdots \\ \partial O^n(w)/\partial w_{s,t} \\ \vdots \\ \partial O^n(w)/\partial w_{s,s'} \\ \vdots \end{bmatrix}$

Let me show $\frac{\partial O^n(w)}{\partial w_{s,t}}$
 $\frac{\partial O^n(w)}{\partial w_{s,s'}}$ very similar

求偏导

$$\begin{aligned} O^n(w) &= \log \frac{\exp(w \cdot \phi(x^n, \hat{y}^n))}{Z(x^n)} \quad Z(x^n) = \sum_{y'} \exp(w \cdot \phi(x^n, y')) \\ &= \underline{w \cdot \phi(x^n, \hat{y}^n)} - \log \underline{Z(x^n)} \end{aligned}$$

$$\frac{\partial O^n(w)}{\partial w_{s,t}} = \underline{N_{s,t}(x^n, \hat{y}^n)}$$

↓

The number of word t labeled
as s in (x^n, \hat{y}^n)

The value of the dimension in
 $\phi(x^n, \hat{y}^n)$ corresponding to $w_{s,t}$.

$$\begin{aligned} &w \cdot \phi(x^n, \hat{y}^n) \\ &= \sum_{s,t} w_{s,t} \cdot N_{s,t}(x^n, \hat{y}^n) \\ &\quad + \sum_{s,s'} w_{s,s'} \cdot N_{s,s'}(x^n, \hat{y}^n) \end{aligned}$$

$$O^n(w) = \log \frac{\exp(w \cdot \phi(x^n, \hat{y}^n))}{Z(x^n)} \quad Z(x^n) = \sum_{y'} \exp(w \cdot \phi(x^n, y'))$$

$$= \underline{w \cdot \phi(x^n, \hat{y}^n)} - \underline{\log Z(x^n)}$$

$$\frac{\partial O^n(w)}{\partial w_{s,t}} = \underline{N_{s,t}(x^n, \hat{y}^n)} - \frac{1}{\underline{Z(x^n)}} \underline{\frac{\partial Z(x^n)}{\partial w_{s,t}}}$$

$$= \sum_{y'} \underline{\frac{\exp(w \cdot \phi(x^n, y'))}{Z(x^n)}} N_{s,t}(x^n, y') \cancel{=} \sum_{y'} \underline{P(y'|x^n)} N_{s,t}(x^n, y')$$

$$\underline{\frac{\partial Z(x^n)}{\partial w_{s,t}}} = \sum_{y'} \exp(w \cdot \phi(x^n, y')) N_{s,t}(x^n, y')$$

$$P(y'|x^n) = \frac{\exp(w \cdot \phi(x^n, y'))}{Z(x^n)}$$

CRF - Training

$$w_{s,t} \rightarrow w_{s,t} + \eta \frac{\partial O(w)}{\partial w_{s,t}}$$

After some math

Can be computed by Viterbi algorithm as well

$$\frac{\partial O^n(w)}{\partial w_{s,t}} = \underline{N_{s,t}(x^n, \hat{y}^n)} - \sum_{y'} \underline{P(y'|x^n)} \underline{N_{s,t}(x^n, y')}$$

If word t is labeled by tag s in training examples (x^n, \hat{y}^n) , then increase $w_{s,t}$

If word t is labeled by tag s in (x^n, y') which not in training examples, then decrease $w_{s,t}$

偏导求解得到两项：

$$\frac{\partial O^n(w)}{\partial w_{s,t}} = N_{s,t}(x^n, \hat{y}^n) - \sum_{y'} P(y'|x^n) N_{s,t}(x^n, y')$$

- 第一项为单词t被标记为s, 在 (x^n, \hat{y}^n) 中出现的次数;
- 第二项为累加所有可能的y, 每一项为 单词t被标记成s在 x_n 与任意y的pair里面出现的次数乘上给定 x_n 下产生这个y的机率。
- 实际意义解释
 - 第一项说明：如果(s, t)在训练数据集正确出现的次数越多，对应的w的值就会越大，即如果单词t在训练数据对集 (x^n, \hat{y}^n) 中被标记成s，则会增加 $w_{s,t}$ ；

- 第二项说明：如果 (s, t) 在训练数据集任意的 y 与 x 配对之后出现的次数依然越多，那么我们应该将其权值进行减小(可以通过Viterbi算法计算)，即如果任意一个单词 t 在任意一个训练数据对集 (x^n, y') 中被标记成 s 的话，我们要减小 $w_{s,t}$ 。

对所有的权值向量来说，更新过程是：正确的 \hat{y}^n 所形成的的向量减去任意一个 y' 形成的的向量乘上 y' 的机率。

$$\nabla O(w) = \phi(x^n, \hat{y}^n) - \sum_{y'} P(y'|x^n) \phi(x^n, y')$$

Stochastic Gradient Ascent

Random pick a data (x^n, \hat{y}^n)

$$w \rightarrow w + \eta \left(\phi(x^n, \hat{y}^n) - \sum_{y'} P(y'|x^n) \phi(x^n, y') \right)$$

CRF – Inference

$$y = \arg \max_{y \in Y} P(y|x) = \arg \max_{y \in Y} P(x, y)$$

$$= \arg \max_{y \in Y} w \cdot \phi(x, y) \quad \text{Done by Viterbi as well}$$

$$P(x, y) \propto \exp(w \cdot \phi(x, y))$$

等同于找一个 y ，使得 $w \cdot \phi(x, y)$ 机率最大，因为由 $P(x, y) \propto \exp(w \cdot \phi(x, y))$ 可知。

CRF v.s. HMM

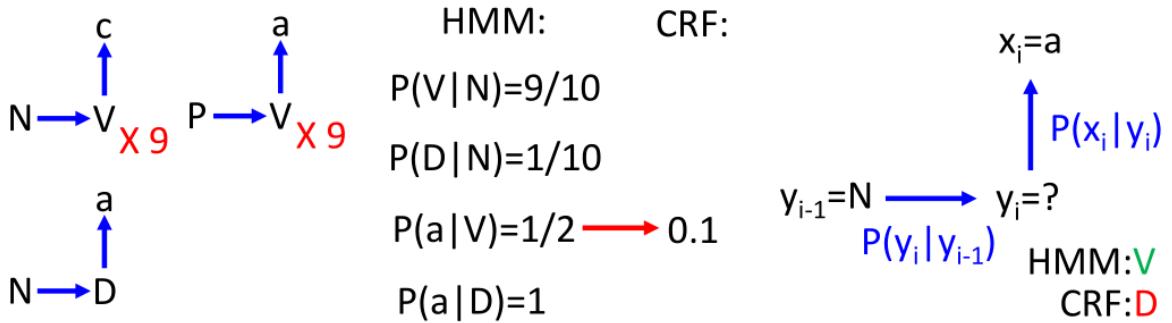
CRF增加 $P(x, \hat{y})$ ，减少 $P(x, y')$ (HMM做不到这一点)

- 如果要得到正确的答案，我们希望

$$(x, \hat{y}) : P(x, \hat{y}) > P(x, y)$$

条件随机场更有可能得到正确的结果。

CRF可能会想办法调整参数，把 v 产生 a 的机率变小，让正确的机率变大，错误的变小。



Synthetic Data

输入输出分别为：

$$x_i \in \{a - z\}, y_i \in \{A - E\}$$

从混合顺序隐马尔科夫模型生成数据

- 转移概率

$$\alpha P(y_i|y_{i-1}) + (1 - \alpha)P(y_i|y_{i-1}, y_{i-2})$$

α 取1时，变为一般的隐马尔科夫模型，其值可以任意地进行调整。

- 发散概率

$$\alpha P(x_i|y_i) + (1 - \alpha)P(x_i|y_i, x_{i-1})$$

如果 α 取1时，变为一般的隐马尔科夫模型。

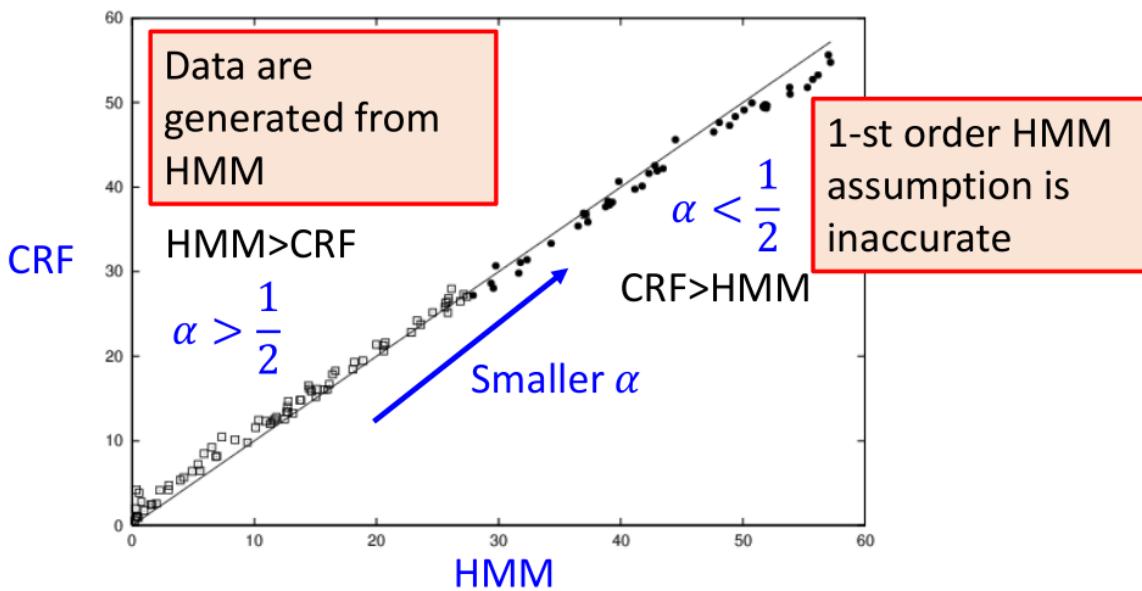
Comparing HMM and CRF

- $x_i \in \{a - z\}, y_i \in \{A - E\}$
- Generating data from a mixed-order HMM
 - Transition probability:
 - $\alpha P(y_i|y_{i-1}) + (1 - \alpha)P(y_i|y_{i-1}, y_{i-2})$
 - Emission probability:
 - $\alpha P(x_i|y_i) + (1 - \alpha)P(x_i|y_i, x_{i-1})$
- Comparing HMM and CRF
 - All the approaches only consider 1-st order information
 - Only considering the relation of y_{i-1} and y_i
 - In general, all the approaches have worse performance with smaller α

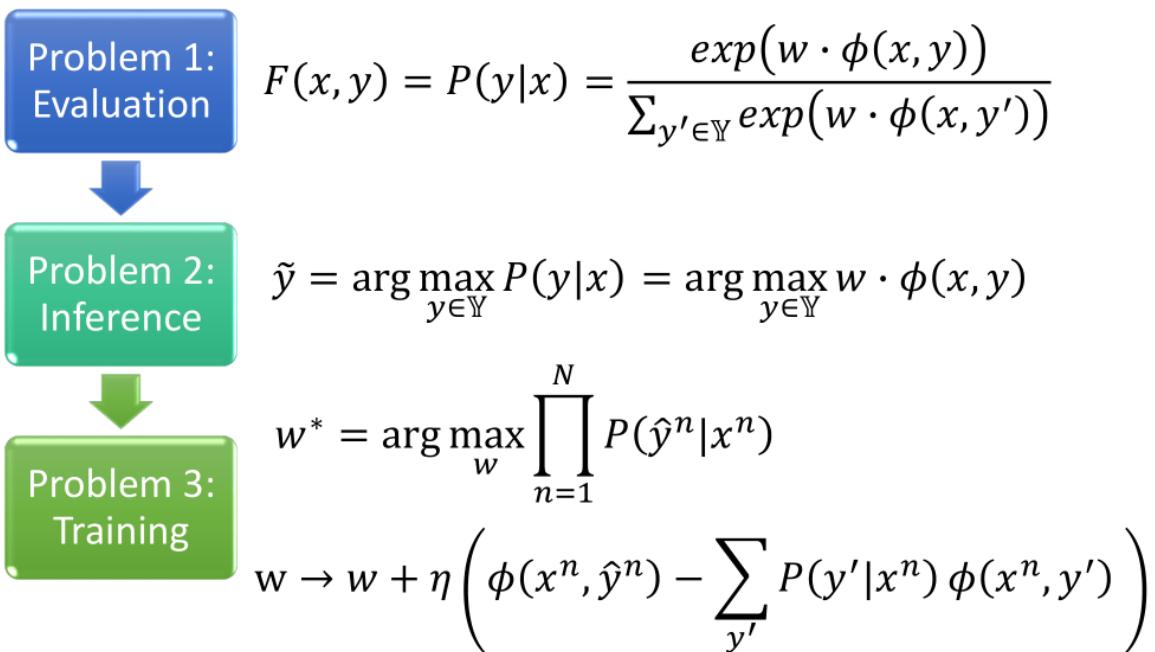
Ref: John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira, “Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data”, ICML, 2001

α 从左下方到右上方不断减小，每一个圈圈表示不同的 α 所得到的结果，对每一个点都做一个隐马尔科夫模型与条件随机场的实验，横轴代表隐马尔科夫模型犯错的百分比，纵轴表示条件随机场犯错的百分比。

当模型与假设不合的时候，CRF比HMM得到了更好的结果



CRF - Summary



w^* = arg max的式子，可以写成对数相加形式。

Structured Perceptron

x, y 假设都为序列，可以用**条件随机场模型**来定义 $\phi(x, y)$ ；

Problem 2 利用**维特比算法**求解即可；

训练时，对所有的训练数据n，以及对所有的 $y \neq \hat{y}^n$ ，我们希望：

$$w \cdot \phi(x^n, \hat{y}^n) > w \cdot \phi(x^n, y)$$

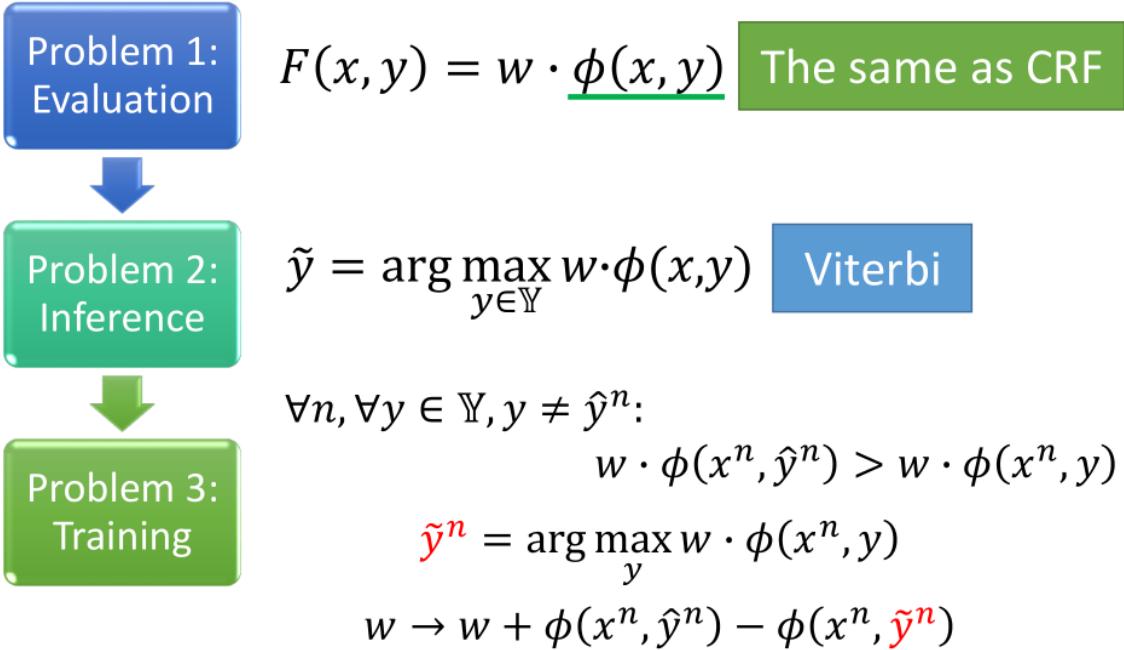
在每个iteration里面，我们会根据目前的w，找到一个 \tilde{y}^n ，使得：

$$\tilde{y}^n = \arg \max_y w \cdot \phi(x^n, y)$$

然后，更新w

$$w \rightarrow w + \phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)$$

即正确的 \hat{y}^n 减去其他的 \tilde{y}^n 所形成的向量。



Structured Perceptron v.s. CRF

Structured Perceptron

$$\begin{aligned}\tilde{y}^n &= \arg \max_y w \cdot \phi(x^n, y) \\ w &\rightarrow w + \phi(x^n, \hat{y}^n) - \phi(x^n, \tilde{y}^n)\end{aligned}$$

只减去机率最大的y的特征向量

CRF

$$w \rightarrow w + \eta \left(\frac{\phi(x^n, \hat{y}^n)}{\sum_{y'} P(y'|x^n) \phi(x^n, y')} \right)$$

减去了所有的 y' 所形成的特征向量与对应的机率做weighted sum

• Structured Perceptron

$$\tilde{y}^n = \arg \max_y w \cdot \phi(x^n, y)$$

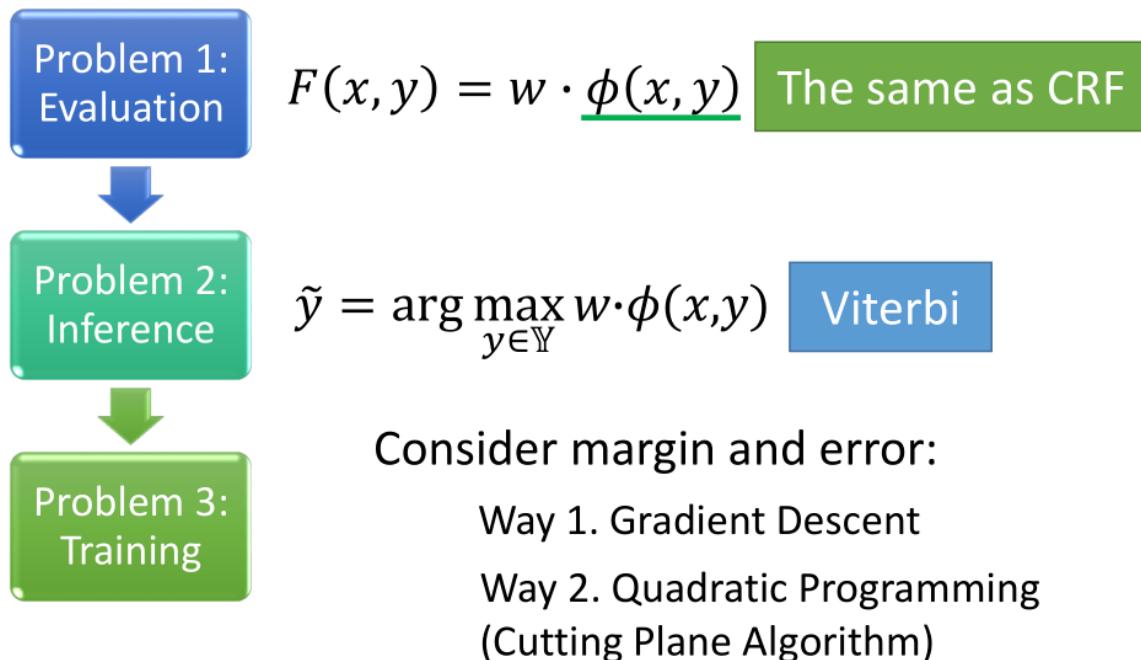
$$w \rightarrow w + \underbrace{\phi(x^n, \hat{y}^n)}_{\text{Hard}} - \underbrace{\phi(x^n, \tilde{y}^n)}_{\text{Soft}}$$

• CRF

$$w \rightarrow w + \eta \left(\underbrace{\phi(x^n, \hat{y}^n)}_{\text{Hard}} - \sum_{y'} P(y' | x^n) \phi(x^n, y') \right)$$

Structured SVM

目标函数需要考虑到间隔和误差，训练时可以采用梯度下降法，也可以作为QP问题，因为限制条件过多，所以采用切割平面算法解。



Error Function

Error function

$$\Delta(\hat{y}^n, y)$$

- Δ 用来计算 y 与 \hat{y}^n 之间的差异性；
- 结构化支持向量机的成本函数就是 Δ 的上界，最小化Cost Function就是在最小化上界；

- 理论上讲， Δ 可以为任何适当的函数；但是，我们必须要考虑到，我们需要穷举所有的 y ，看哪—一个使得 Δ 加上 $w \cdot \phi$ 最大化（这是Problem 2.1，相比Problem 2是一个不一样的问题）

在下图示例情况下，把 Δ 定义成错误率，Problem 2.1可以通过维特比算法求解。

- Error function: $\Delta(\hat{y}^n, y)$
 - $\Delta(\hat{y}^n, y)$: Difference between y and \hat{y}^n
 - Cost function of structured SVM is the upper bound of $\Delta(\hat{y}^n, y)$
 - Theoretically, $\Delta(y, \hat{y}^n)$ can be any function you like
 - However, you need to solve **Problem 2.1**
 - $\bar{y}^n = \operatorname{argmax}_y [\Delta(\hat{y}^n, y) + w \cdot \phi(x^n, y)]$

Example

$$\Delta(\hat{y}, y) = 3/10$$

\hat{y} :	A	T	T	C	G	G	G	G	A	T
y :	A	T	T	A	G	G	A	G	A	A

In this case, problem 2.1 can be solved by Viterbi Algorithm

Performance of Different Approaches

POS

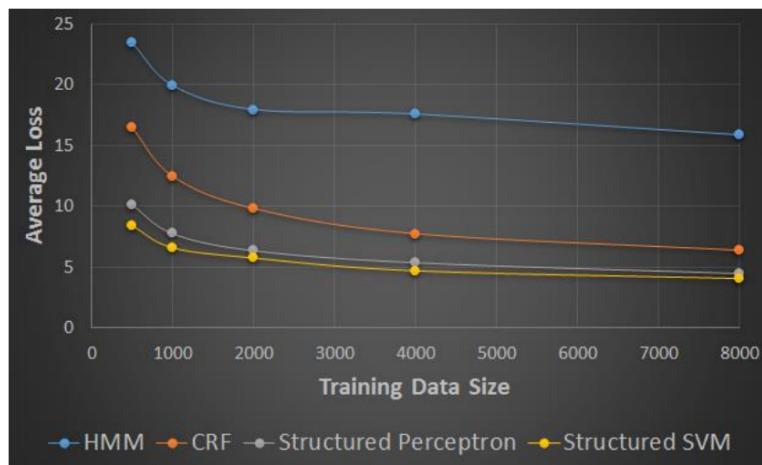
- HMM performance表现最差，但是当训练数据更少的情况下，可能隐马尔科夫模型表现会相比其他方法好一些；CRF赢过HMM，
- 条件随机场与结构化感知机谁比较强文献上是没有定论的，条件随机场是soft的，而结构化感知机是hard的。但是结构化感知机只需要求解Problem 2就可以了，而条件随机场需要summation over所有的 y' ，这件事不一定知道怎么解，如果不知道怎么解的话，推荐用结构化感知机。
- 结构化支持向量机整体表现最好；

命名实体识别（把tag换成公司名/地名/人名等）

- 结构化支持向量机模型表现最好；
- 隐马尔科夫模型表现最差。

POS Tagging

Ref: Nguyen, Nam, and Yunsong Guo.
"Comparisons of sequence labeling
algorithms and extensions." *ICML*, 2007.



Name Entity Recognition

Method	HMM	CRF	Perceptron	SVM
Error	9.36	5.17	5.94	5.08

Ref: Tsochantaridis, Ioannis, et al. "Large margin methods for structured and interdependent output variables." *Journal of Machine Learning Research*. 2005.

RNN v.s. Structured Learning

RNN, LSTM

- 单方向的循环神经网络或长短时记忆网络并没有考虑到全部的序列，换言之，只考虑时间 t_1 至当前时间 t_k 的情形，对 t_{k+1} 的情形没有考虑。
- 有足够多的data，或许可以learn到标签之间的依赖关系
- Cost和Error并不见得总是相关的
- 可以叠加很多层（利用Deep的特性）

HMM, CRF, Structured Perceptron/SVM

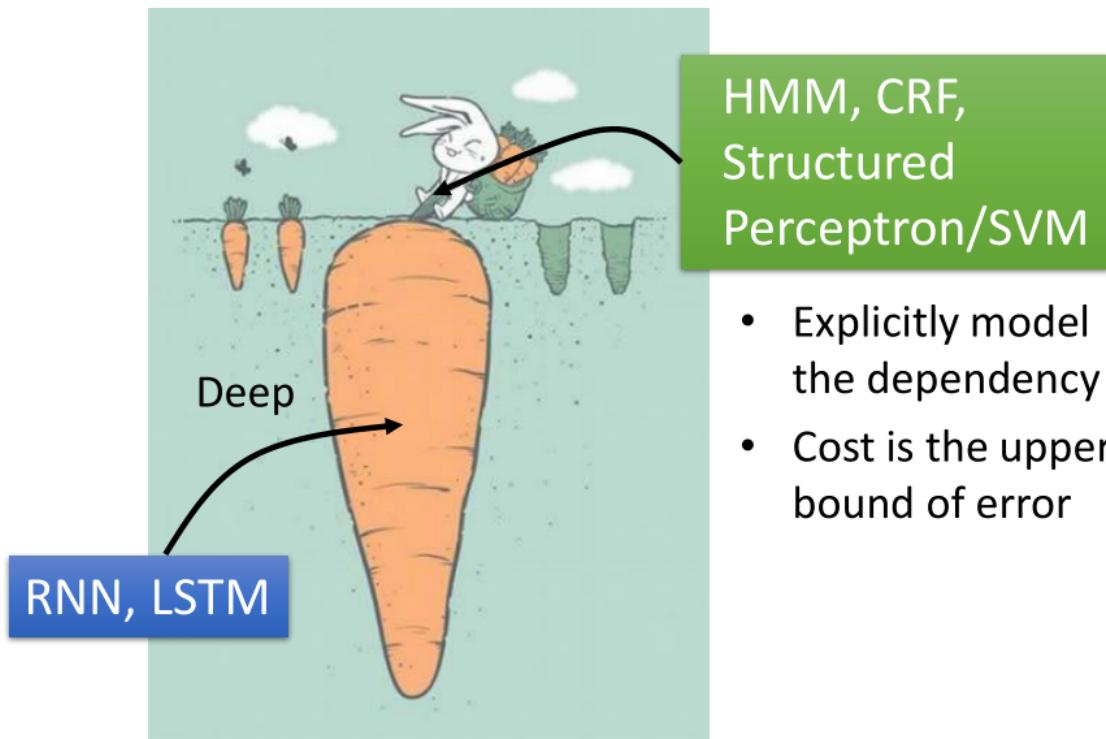
- 在输出结果之前，做的都是利用维特比算法穷举所有的序列，观测最大的序列，在计算过程中考虑到的是整个序列；（开放问题：如果利用双向的循环神经网络与其相比，结果如何？）胜的可能比较牵强。
- 我们可以直接把限制加到维特比算法中，可以只穷举符合限制的sequence，可以把标签之间的依赖关系明确的描述在model中；
- 结构化支持向量机的Cost Function就是Error的上界，当Cost Function不断减小的时候，Error很有可能会随之降低。
- 其实也可以是deep，但是它们要想拿来做deep learning 是比较困难的。在我们讲的内容里面它们都是linear，因为他们定义的evaluation函数是线性的。如果不是线性的话也会很麻烦，因为只有是线性的我们才能套用这些方法来做inference和training。

- RNN, LSTM
 - Unidirectional RNN does not consider the whole sequence
 - Cost and error not always related
 - Deep 
- HMM, CRF, Structured Perceptron/SVM
 - Using Viterbi, so consider the whole sequence 
 - How about Bidirectional RNN?
 - Can explicitly consider the label dependency 
 - Cost is the upper bound of error 



最后总结来看，RNN/LSTM在deep这件事的表现其实会比较好，同时在SOTA上，RNN是不可或缺的，如果只是线性的模型，function space就这么大，就算可以直接最小化一个错误的上界，但是这样没什么，因为所有的结果都是坏的，所以相比之下，deep learning占到很大的优势。

Integrated Together



- 底层（埋在土里的萝卜）
 - 循环神经网络与长短时记忆网络
- 叶子
 - 隐马尔可夫模型，条件随机场与结构化感知机/支持向量机，有很多优点。

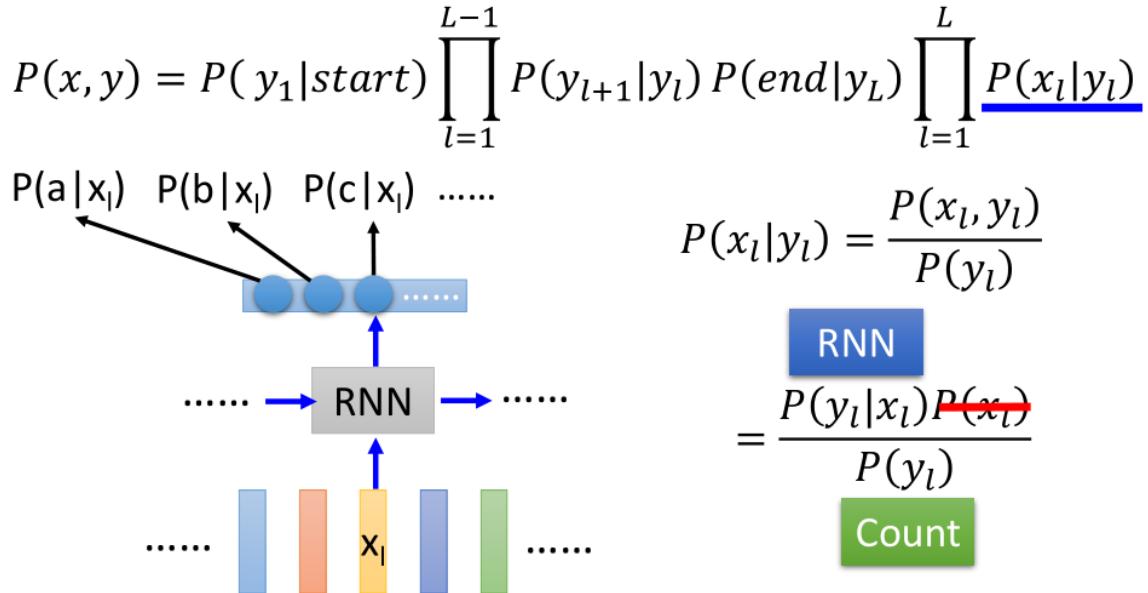
语音识别

- 卷积神经网络/循环神经网络或长短时记忆网络/深度神经网络 + 隐马尔可夫模型

$$P(x, y) = P(y_1 | start) \prod_{l=1}^{L-1} P(y_{l+1} | y_l) P(end | y_L) \prod_{l=1}^L P(x_l | y_l)$$

- 根据隐马尔科夫模型中，发散概率可以由神经网络得到，用循环神经网络的输出结果经过变换代替发散概率；不需要考虑 $P(x_l)$ 的机率；
- 双向循环神经网络/长短时记忆网络 + 条件随机场/结构化支持向量机。

• Speech Recognition: CNN/LSTM/DNN + HMM



其实加上HMM在语音辨识里很有帮助，就算是用RNN，但是在辨识的时候，常常会遇到问题，假设我们是一个frame，用RNN来问这个frame属于哪个form，往往会产生奇怪的结果，比如说一个frame往往是蔓延好多个frame，比如理论是看到第一个frame是A，第二个frame是A，第三个是A，第四个是A，然后BBB，但是如果用RNN做的时候，RNN每个产生的label都是独立的，所以可能会若无其事的改成B，然后又是A，RNN很容易出现这个现象。HMM则可以把这种情况修复。因为RNN在训练的时候是每个frame分来考虑的，因此不同地方犯的错误对结果的影响相同，结果就会不好，如果想要不同，加上结构化学习的概念才可以做到。所以加上结构化学习的概念会很有帮助。

Semantic Tagging

- 从RNN的输出结果中，抽出新的特征再计算； $w \cdot \phi(x, y)$ 作为评估函数

- 训练阶段

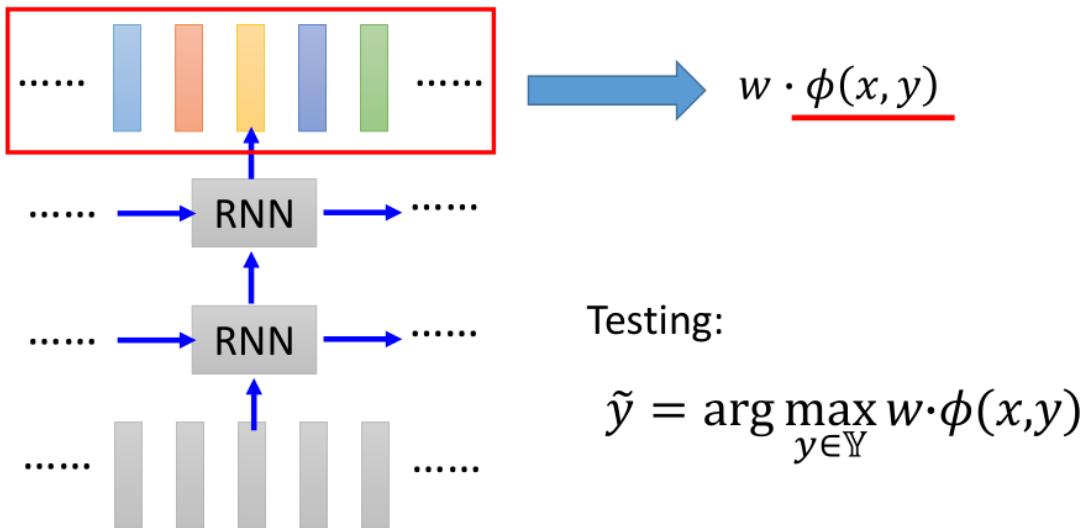
利用梯度下降法让 w 和循环神经网络中的所有参数一起训练；

- 测试阶段

$$\tilde{y} = \arg \max_{y \in \mathbb{Y}} w \cdot \phi(x, y)$$

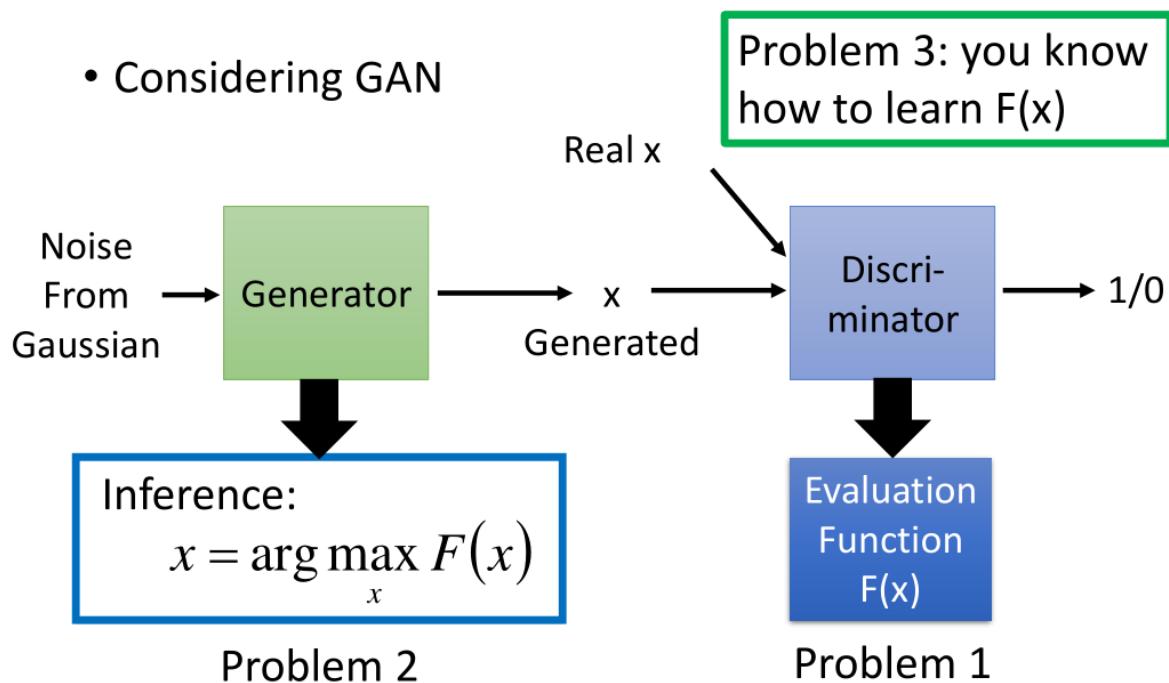
找一个 y ，使得 $w \cdot \phi(x, y)$ 的结果最大化，但此时的 x 不是input x ，而是来自于循环神经网络的输出结果。

- Semantic Tagging: Bi-directional LSTM + CRF/Structured SVM



Is Structure learning practical?

structured learning需要解三个问题，其中problem 2往往很困难，因为要穷举所有的 y 让其最大，解一个optimization的问题，大部分状况都没有好的解决办法。所有有人说structured learning应用并不广泛，但是未来未必是这样的。



其实GAN就是一种structured learning。可以把discriminator看做是evaluation function (也就是problem 1) 我们要解一个inference的问题 (problem 2)，我们要穷举我们未知的东西，看哪个可以让我们的evaluation function最大。这步往往比较困难，因为 x 的可能性太多了。但这个东西可以就是generator，我们可以想成generator就是给出一个noise，输出一个object，它输出的这个object，就是让discriminator分辨不出的object，如果discriminator就是evaluation function的话，那output的值就是让evaluation function的值很大的那个对应值。所以这个generator就是在解problem 2，其实generator的输出就是argmax的输出，可以把generator当做在解inference这个问题。Problem 3的solution就是train GAN的方法。

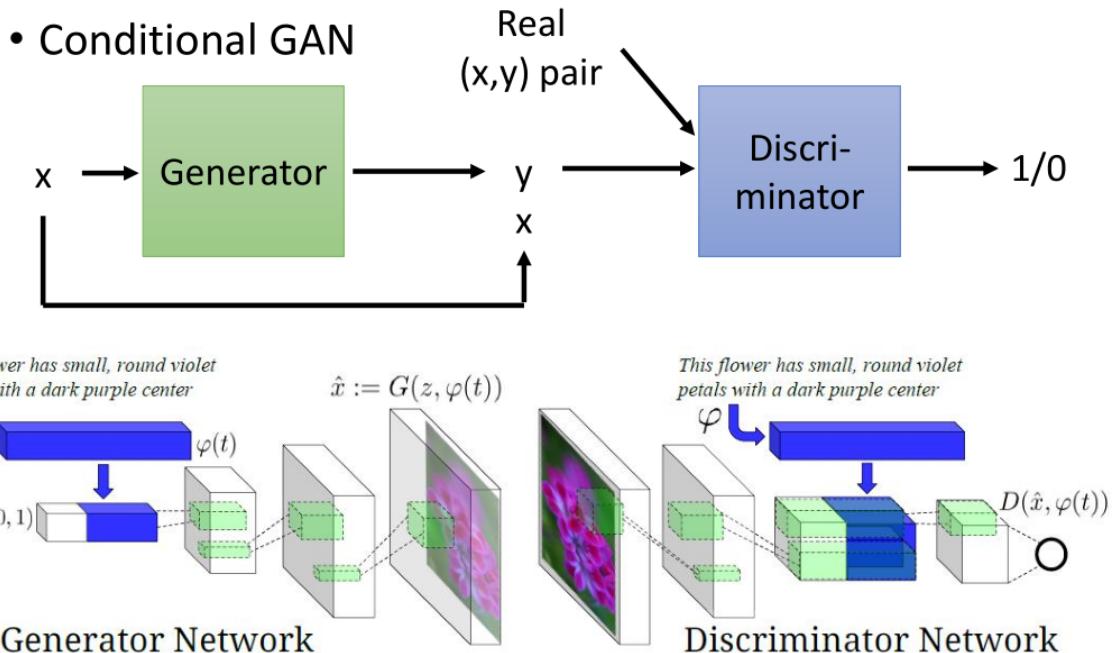
在 structured SVM 的 training 里面，我们每次找出最 competitive 的那些 example，然后我们希望正确的 example 的 evaluation function 的分数大过 competitive 的 example，然后 update 我们的 model，然后再重新选 competitive 的 example，然后再让正确的，大过 competitive，就这样 iterative 做去。

GAN 的 training 是我们有正确的 example，它应该要让 evaluation function 的值比 Discriminator 的值大，然后我们每次用这个 Generator，Generate 出最competitive 的那个 x ，也就是可以让 Discriminator 的值最大的那个 x ，然后再去 train Discriminator。Discriminator 要分辨正确的跟 Generated 的。也就是 Discriminator 要给 real 的 example 比较大的值，给那些 most competitive 的 x 比较小的值，然后这个 process 就不断的 iterative 的进行下去，你会 update 你的 Discriminator，然后 update 你的 Generator。

其实这个跟 Structured SVM 的 training 是有异曲同工之妙的。

我们在讲 structured SVM 的时候都是有一个 input/output，有一个 x 有一个 y ；GAN 只有 x ，听起来好像不太像，那我们就另外讲一个像的。

GAN 也可以是 conditional 的 GAN，example 都是 x, y 的 pair，现在的任务是，given x 找出最有可能的 y 。



比如语音辨识， x 是声音讯号， y 是辨识出来的文字，如果是用 conditional 的概念，generator 输入一个 x ，就会 output 一个 y ，discriminator 是去检查 y 的 pair 是不是对的，如果给他一个真正的 x, y 的 pair，会得到一个比较高的分数，给一个 generator 输出的一个 y 配上输入的 x 所产生的一个假的 pair，就会给他一个比较低的分数。

训练的过程就和原来的 GAN 就是一样的，这个已经成功运用在文字产生图片这个 task 上面。这个 task 的 input 就是一句话，output 就是一张图，generator 做的事就是输入一句话，然后产生一张图片，而 discriminator 要做的事就是给他一张图片和一句话，要他判断这个 x, y 的 pair 是不是真的，如果把 discriminator 换成 evaluation function，把 generator 换成 inference 的 problem，其实 conditional GAN 和 structured learning 就是可以类比，或者说 GAN 就是训练 structured learning model 的一种方法。

很多人都有类似的想法，比如 GAN 可以跟 energy based model 做 connection，可以视为 train energy based model 的一种方法。所谓 energy based model，它就是 structured learning 的另外一种称呼。

Generator 视做是在做 inference 这件事情，是在解 arg max 这个问题，听起来感觉很荒谬。但是也有人觉得一个 neural network，它有可能就是在解 arg max 这个 problem，这里也给出一些 Reference。

所以也许 deep and structured 就是未来一个研究的重点的方向。

Sounds crazy?
People do think in this way ...

Deep and Structured
will be the future.

- Connect Energy-based model with GAN:
 - A Connection Between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models
 - Deep Directed Generative Models with Energy-Based Probability Estimation
 - ENERGY-BASED GENERATIVE ADVERSARIAL NETWORKS
- Deep learning model for inference
 - Deep Unfolding: Model-Based Inspiration of Novel Deep Architectures
 - Conditional Random Fields as Recurrent Neural Networks

Concluding Remarks

Concluding Remarks

	Problem 1	Problem 2	Problem 3
HMM	$F(x, y) = P(x, y)$	Viterbi	Just count
CRF	$F(x, y) = P(y x)$	Viterbi	Maximize $P(\hat{y} x)$
Structured Perceptron	$F(x, y) = w \cdot \phi(x, y)$ (not a probability)	Viterbi	$F(x, \hat{y}) > F(x, y')$
Structured SVM	$F(x, y) = w \cdot \phi(x, y)$ (not a probability)	Viterbi	$F(x, \hat{y}) > F(x, y')$ with margins
Semi-Markov	$F(x, y)$ for x and y with different lengths	Modified Viterbi	Can be the same as CRF, structured perceptron or SVM

The above approaches can combine with deep learning to have better performance.

- 隐马尔可夫模型，条件随机场，结构化感知机或支持向量机都是求解三个问题；
- 三个方法定义Evaluation Function的方式有所差异；结构化感知机或支持向量机跟机率都没有关系；

- 以上这些方法都可以加上深度学习让它们的性能表现地更好。

Flow-based Generative Model

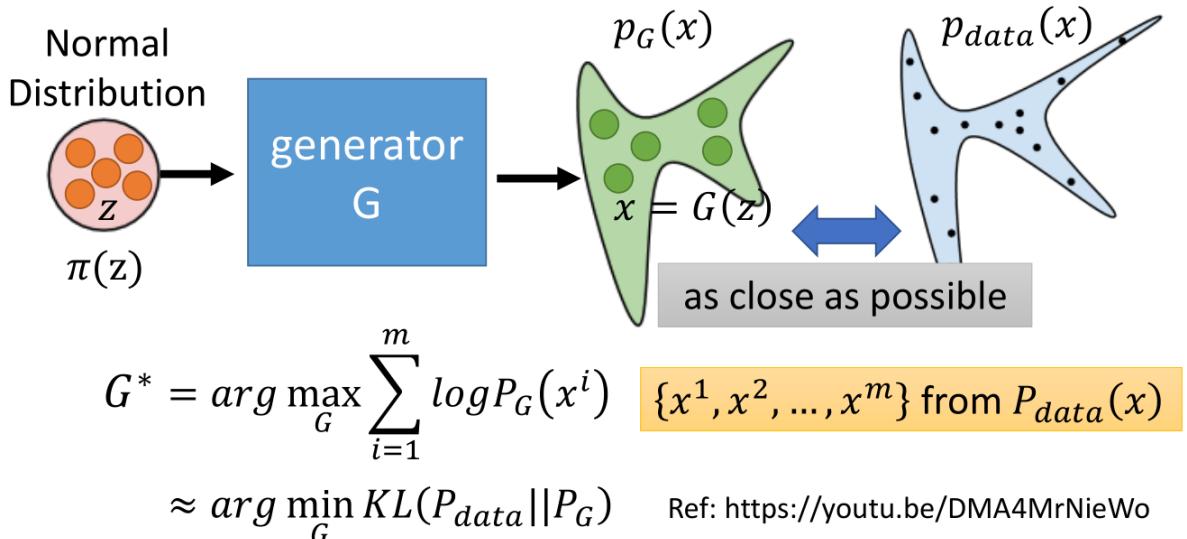
Flow-based Generative Model

Generative Models

- Component-by-component (Auto-regressive Model)
 - What is the best order for the components?
 - Slow generation
- Variational Auto-encoder
 - Optimizing a lower bound (of likelihood)
- Generative Adversarial Network
 - Unstable training

Generator

A generator G is a network. The network defines a probability distribution p_G



Flow-based model directly optimizes the objective function.

Math Background

Jacobian Matrix

$$\begin{aligned}
z &= \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} & x &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\
x = f(z) && z = f^{-1}(x) & \\
J_f &= \overbrace{\begin{bmatrix} \frac{\partial x_1}{\partial z_1} & \frac{\partial x_1}{\partial z_2} \\ \frac{\partial x_2}{\partial z_1} & \frac{\partial x_2}{\partial z_2} \end{bmatrix}}^{\text{input}} \text{ output} \\
J_{f^{-1}} &= \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} \\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} \end{bmatrix} \\
J_f J_{f^{-1}} &= I
\end{aligned}$$

Demo

$$\begin{aligned}
\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} &= \begin{bmatrix} z_1 + z_2 \\ 2z_1 \end{bmatrix} = f \left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \right) \\
\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} &= \begin{bmatrix} x_2/2 \\ x_1 - x_2/2 \end{bmatrix} = f^{-1} \left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right) \\
J_f &= \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix} \\
J_{f^{-1}} &= \begin{bmatrix} 0 & 1/2 \\ 1 & -1/2 \end{bmatrix}
\end{aligned}$$

Determinant

The determinant of a **square matrix** is a **scalar** that provides information about the matrix.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix}$$

$$\det(A) = ad - bc \quad \det(A) =$$

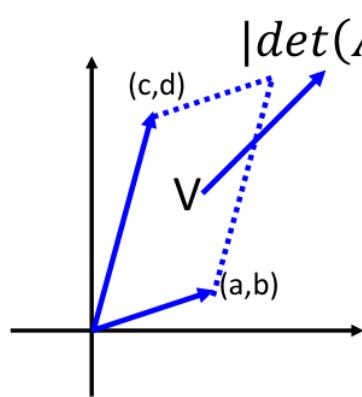
$$\begin{aligned}
\det(A) &= 1/\det(A^{-1}) \\
\det(J_f) &= 1/\det(J_{f^{-1}})
\end{aligned}$$

$$\begin{aligned}
&a_1 a_5 a_9 + a_2 a_6 a_7 + a_3 a_4 a_8 \\
&- a_3 a_5 a_7 - a_2 a_4 a_9 - a_1 a_6 a_8
\end{aligned}$$

高维空间中的体积的概念

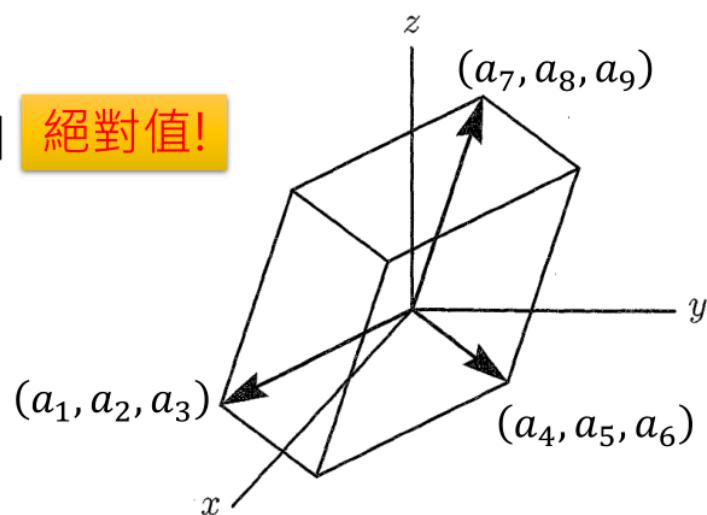
• 2×2

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

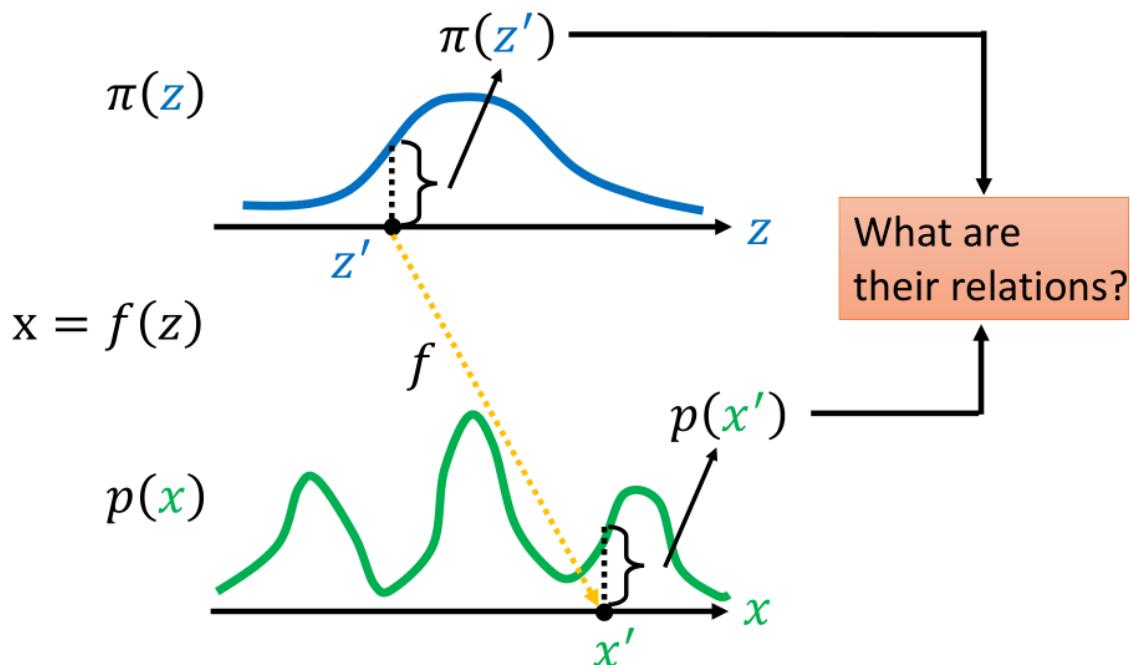


• 3×3

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix}$$



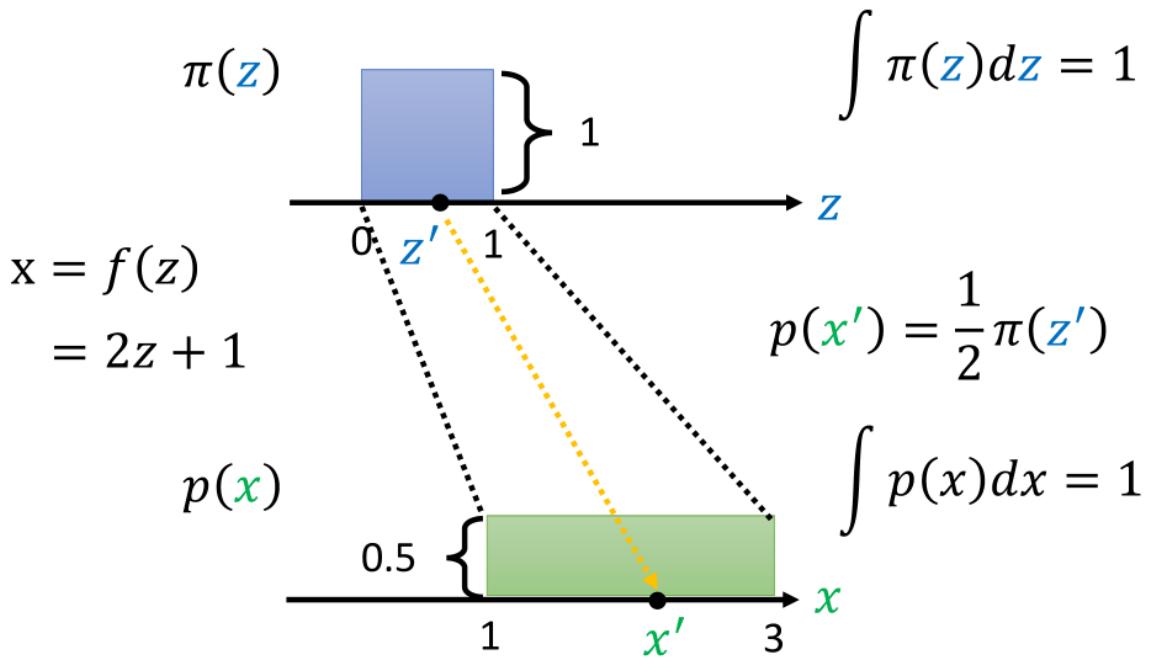
Change of Variable Theorem



如上图所示，给定两组数据 z 和 x ，其中 z 服从已知的简单先验分布 $\pi(z)$ （通常是高斯分布）， x 服从复杂的分布 $p(x)$ （即训练数据代表的分布），现在我们想要找到一个变换函数 f ，它能建立一种 z 到 x 的映射，使得每对于 $\pi(z)$ 中的一个采样点，都能在 $p(x)$ 中有一个（新）样本点与之对应。

如果这个变换函数能找到的话，那么我们就实现了一个生成模型的构造。因为， $p(x)$ 中的每一个样本点都代表一张具体的图片，如果我们希望机器画出新图片的话，只需要从 $\pi(z)$ 中随机采样一个点，然后通过映射，得到新样本点，也就是对应的生成的具体图片。

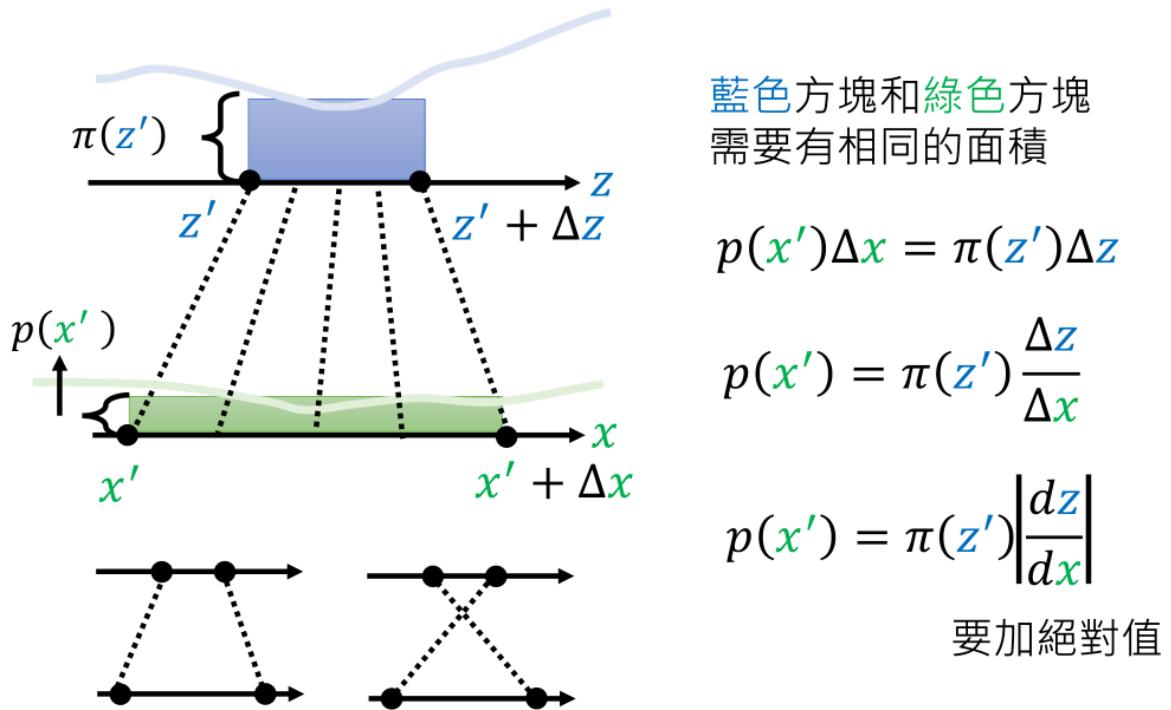
接下来的关键在于，这个变换函数如何找呢？我们先来看一个最简单的例子。



如上图所示，假设 z 和 x 都是一维分布，其中 z 满足简单的均匀分布： $\pi(z) = 1(z \in [0, 1])$ ， x 也满足简单均匀分布： $p(x) = 0.5(x \in [1, 3])$ 。

那么构建 z 与 x 之间的变换关系只需要构造一个线性函数即可： $x = f(z) = 2z + 1$ 。

下面再考虑非均匀分布的更复杂的情况



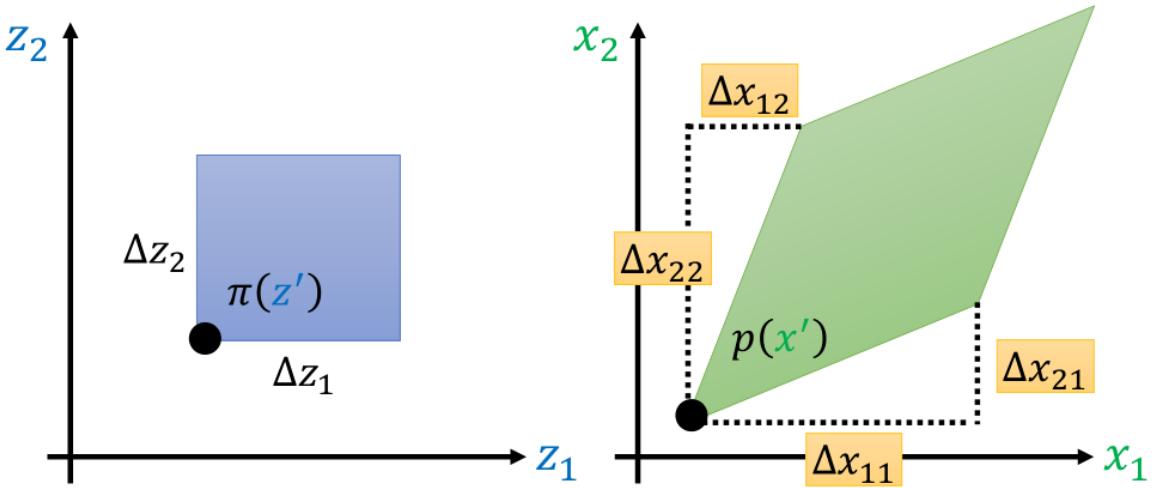
如上图所示， $\pi(z)$ 与 $p(x)$ 都是较为复杂的分布，为了实现二者的转化，我们可以考虑在很短的间隔上将二者视为简单均匀分布，然后应用前边方法计算小段上的，最后将每个小段变换累加起来（每个小段实际对应一个采样样本）就得到最终的完整变换式 f 。

如上图所示，假设在 $[z', z' + \Delta z]$ 上 $\pi(z)$ 近似服从均匀分布，在 $[x', x' + \Delta x]$ 上 $p(x)$ 也近似服从均匀分布，于是有 $p(x') \Delta x = \pi(z') \Delta z$ （因为变换前后的面积/即采样概率是一致的），当 Δx 与 Δz 极小时，有：

$$p(x') = \pi(z') \left| \frac{dz}{dx} \right|$$

又考虑到 $\frac{dz}{dx}$ 有可能是负值，而 $p(x')$ 、 $\pi(z')$ 都为非负，所以的实际关系需要加上绝对值

进一步地做推广，我们考虑 z 与 x 都是二维分布的情形。



$$p(x') \left| \det \begin{bmatrix} \Delta x_{11} & \Delta x_{21} \\ \Delta x_{12} & \Delta x_{22} \end{bmatrix} \right| = \pi(z') \Delta z_1 \Delta z_2$$

如上图所示， z 与 x 都是二维分布，左图中浅蓝色区域表示初始点在 z_1 方向上移动 Δz_1 ，在 z_2 方向上移动 Δz_2 所形成的区域，这一区域通过映射，形成 x 域上的浅绿色菱形区域。其中，二维分布 $\pi(z)$ 与 $p(x)$ 均服从简单均匀分布，其高度在图中未画出（垂直纸面向外）。 Δx_{11} 代表 z_1 改变时， x_1 改变量； Δx_{21} 是 z_1 改变时， x_2 改变量。 Δx_{12} 代表 z_2 改变时， x_1 改变量； Δx_{22} 是 z_2 改变时， x_2 改变量。

因为蓝色区域与绿色区域具有相同的体积，所以有：

$$p(x') \left| \det \begin{bmatrix} \Delta x_{11} & \Delta x_{21} \\ \Delta x_{12} & \Delta x_{22} \end{bmatrix} \right| = \pi(z') \Delta z_1 \Delta z_2 \quad x = f(z)$$

其中 $\det \begin{bmatrix} \Delta x_{11} & \Delta x_{21} \\ \Delta x_{12} & \Delta x_{22} \end{bmatrix}$ 代表行列式计算，它的计算结果等于上图中浅绿色区域的面积（行列式的定义）。下面我们将移 $\Delta z_1 \Delta z_2$ 至左侧，得到：

$$p(x') \left| \frac{1}{\Delta z_1 \Delta z_2} \det \begin{bmatrix} \Delta x_{11} & \Delta x_{21} \\ \Delta x_{12} & \Delta x_{22} \end{bmatrix} \right| = \pi(z')$$

即可得到

$$p(x') \left| \det \begin{bmatrix} \Delta x_{11}/\Delta z_1 & \Delta x_{21}/\Delta z_1 \\ \Delta x_{12}/\Delta z_2 & \Delta x_{22}/\Delta z_2 \end{bmatrix} \right| = \pi(z')$$

当变化很小时

$$p(x') \left| \det \begin{bmatrix} \partial x_1 / \partial z_1 & \partial x_2 / \partial z_1 \\ \partial x_1 / \partial z_2 & \partial x_2 / \partial z_2 \end{bmatrix} \right| = \pi(z')$$

做转置，转置不会改变行列式

$$p(x') \left| \det \begin{bmatrix} \partial x_1 / \partial z_1 & \partial x_1 / \partial z_2 \\ \partial x_2 / \partial z_1 & \partial x_2 / \partial z_2 \end{bmatrix} \right| = \pi(z')$$

就得到

$$p(x') |\det(J_f)| = \pi(z')$$

根据雅各比行列式的逆运算，得到

$$p(x') = \pi(z') \left| \frac{1}{\det(J_f)} \right| = \pi(z') \left| \det(J_{f^{-1}}) \right|$$

至此，我们得到了一个比较重要的结论：如果 z 与 x 分别满足两种分布，并且 z 通过函数 f 能够转变为 x ，那么 z 与 x 中的任意一组对应采样点 z' 与 x' 之间的关系为：

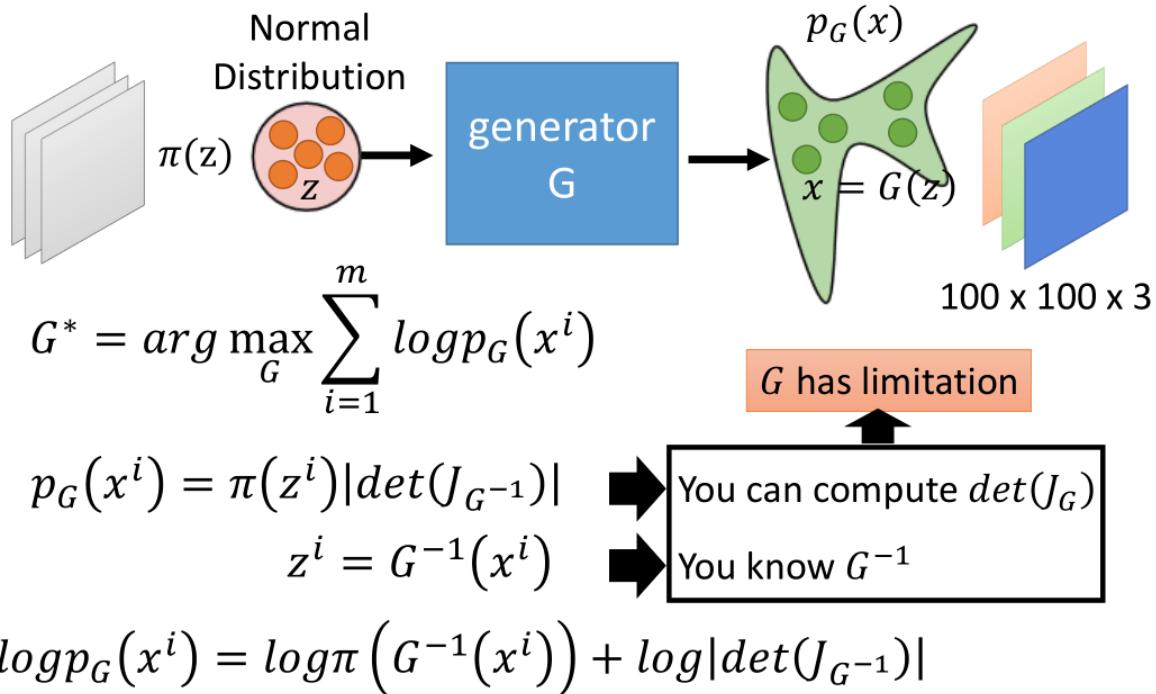
$$\begin{cases} \pi(z') = p(x') |\det(J_f)| \\ p(x') = \pi(z') |\det(J_{f^{-1}})| \end{cases}$$

Formal Explanation

那么基于这一结论，再带回到生成模型要解决的问题当中，我们就得到了 Flow-based Model (流模型) 的初步建模思维。

$$p(\textcolor{red}{x}') |\det(J_f)| = \pi(\textcolor{blue}{z}')$$

Flow-based Model $p(\textcolor{red}{x}') = \pi(\textcolor{blue}{z}') |\det(J_{f^{-1}})|$



上图所示，为了实现 $z \sim \pi(z)$ 到 $x = G(z) \sim p_G(x)$ 间的转化，待求解的生成器 G 的表达式为：

$$G^* = \arg \max_G \sum_{i=1}^m \log p_G(x^i)$$

基于前面推导，我们有 $p_c(x)$ 中的样本点与 $\pi(z)$ 中的样本点间的关系为：

$$p_G(x^i) = \pi(z^i) |\det(J_{G^{-1}})|$$

其中 $z^i = G^{-1}(x^i)$

所以，如果 G^* 的目标式能够通过上述关系式求解出来，那么我们就实现了一个完整的生成模型的求解。

Flow-based Model 就是基于这一思维进行理论推导和模型构建，下面详细解释 Flow-based Model 的求解过程。

将上述式子取 \log ，得到

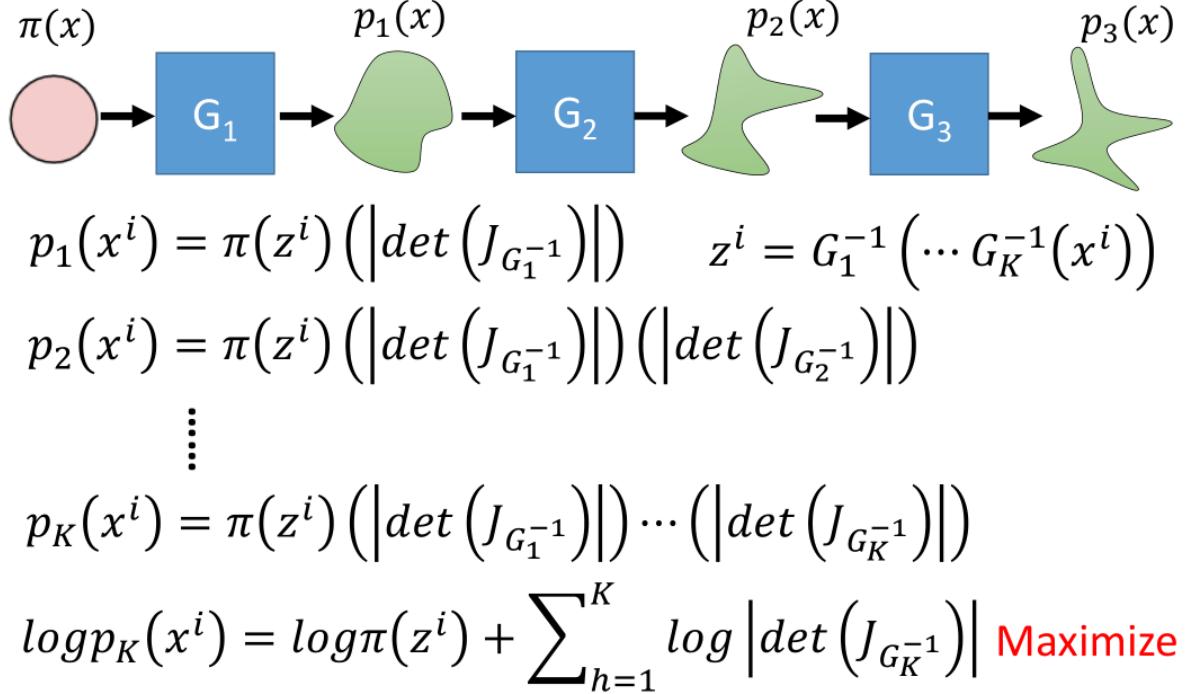
$$\log p_G(x^i) = \log \pi(G^{-1}(x^i)) + \log |\det(J_{G^{-1}})|$$

现在，如果想直接maximize求解这个式子有两方面的困难。

第一个困难是 $\det(J_{G^{-1}})$ 是不好计算的——由于 G^{-1} 的Jacobian矩阵一般维度不低（譬如256*256矩阵），其行列式的计算量是异常巨大的，所以在实际计算中，我们必须对 G^{-1} 的Jacobian行列式做一定优化，使其能够在计算上变得简洁高效。

第二个困难是，表达式中出现了 G^{-1} ，这意味着我们要知道 G^{-1} 长什么样子，而我们的目标是求 G ，所以这需要巧妙地设计 G 的结构使得 G^{-1} 也是好计算的，同时要求 z 和 x 的dimension是一样的才能使 G^{-1} 存在。

这些要求使得 G 有很多限制。由于单个 G 受到了较多的约束，所以可能表征能力有限，因此可以进行多层扩展，其对应的关系式只要进行递推便可。

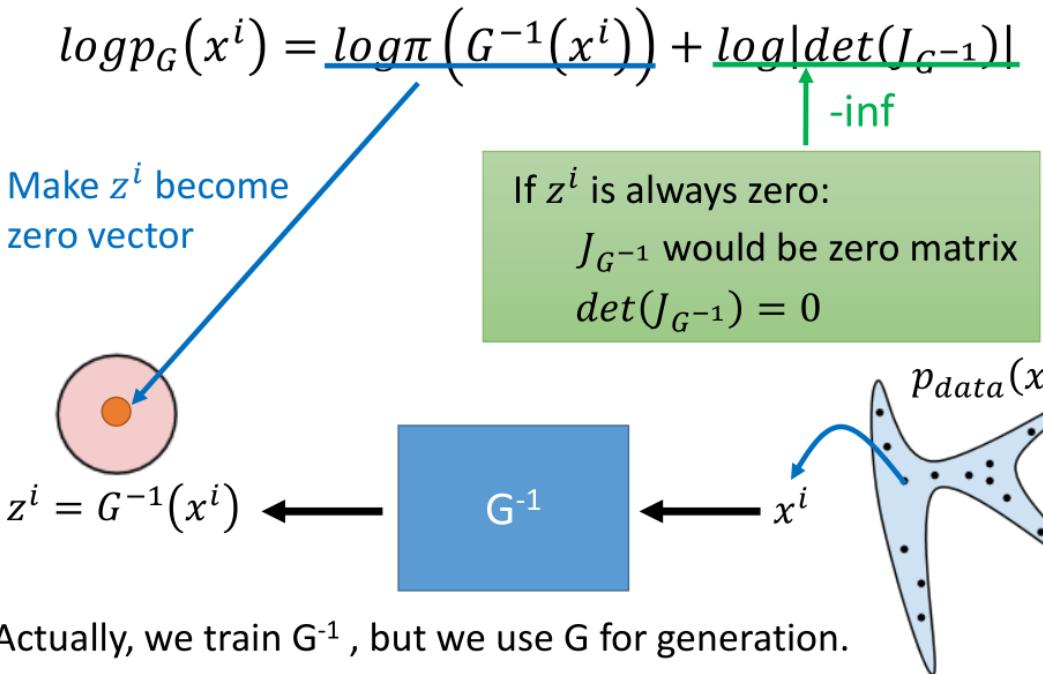
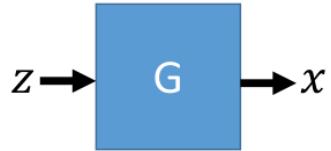


What you actually do?

下面我们来逐步设计 G 的结构，首先从最基本的架构开始构思。考虑到 G^{-1} 必须是存在的且能被算出，这意味着 G 的输入和输出的维度必须是一致的并且 G 的行列式不能为0。

然后，既然 G^{-1} 可以计算出来，而 $\log p_G(x^i)$ 的目标表达式只与 G^{-1} 有关，所以在实际训练中我们可以训练 G^{-1} 对应的网络，然后想办法算出 G 来，并且在测试时改用 G 做图像生成。

What you actually do?



如上图所示，在训练时我们从真实分布 $p_{data}(x)$ 中采样出 x^i ，然后去训练 G^{-1} ，使得通过 G^{-1} 生成的满足特定 $z^i = G^{-1}(x^i)$ 的先验分布，maximize 上面的 objective function，这里一般需要保证 x^i 和 z^i 具有相同的尺寸；接下来在测试时，我们从 z 中采样出一个点 z^j ，然后通过 G 生成的样本 $x^j = G(z^j)$ 就是新的生成图像。

由于 z^i 是符合 Normal Distribution，等于 0 时 $\pi(z^i)$ 最大，因此第一项让 z^i 趋向于 0，第二项又让 z^i 远离 0。

Coupling Layer

接下来开始具体考虑 G 的内部设计，为了让 G^{-1} 可以计算并且 G 的 Jacobian 行列式也易于计算，Flow-based Model 采用了一种称为耦合层（Coupling Layer）的设计来实现。其被应用在 NICE 和 Real NVP 这两篇论文当中。

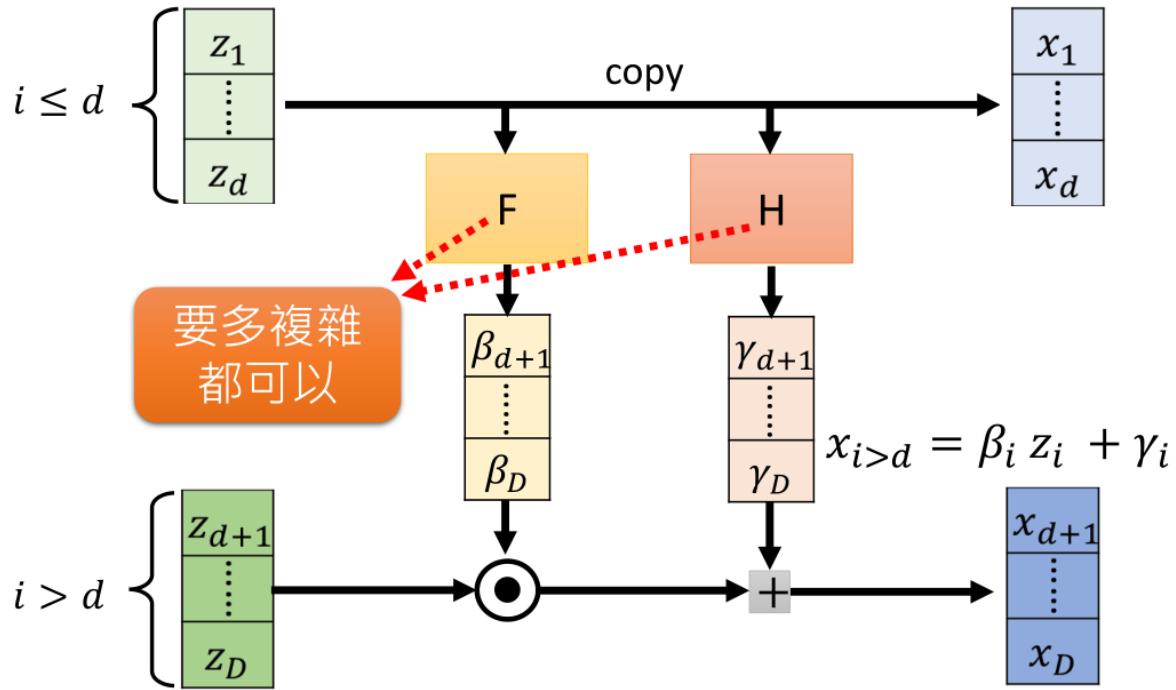
整个流程可以表述为

先将输入 z 拆分成两个部分（可以是按 channel 进行拆分，也可以还是按照 pixel 的 location 进行拆分），对于上面的部分 z_1, \dots, z_d 直接 copy 得到对应的 output x_1, \dots, x_d ，而对于下面的分支则有如下的变换（公式中符号代表 element-wise 相乘）

$$(z_{d+1}, \dots, z_D) \odot F(z_1, \dots, z_d) + H(z_1, \dots, z_d) = x_{d+1}, \dots, x_D,$$

可以简化为： $(z_{d+1}, \dots, z_D) \odot (\beta_1, \dots, \beta_d) + (\gamma_1, \dots, \gamma_d) = x_{d+1}, \dots, x_D$ 或 $\beta_i z_i + \gamma_i = x_{i>d}$ 。

之所以采用以上设计结构的原因在于上述的结构容易进行逆运算。

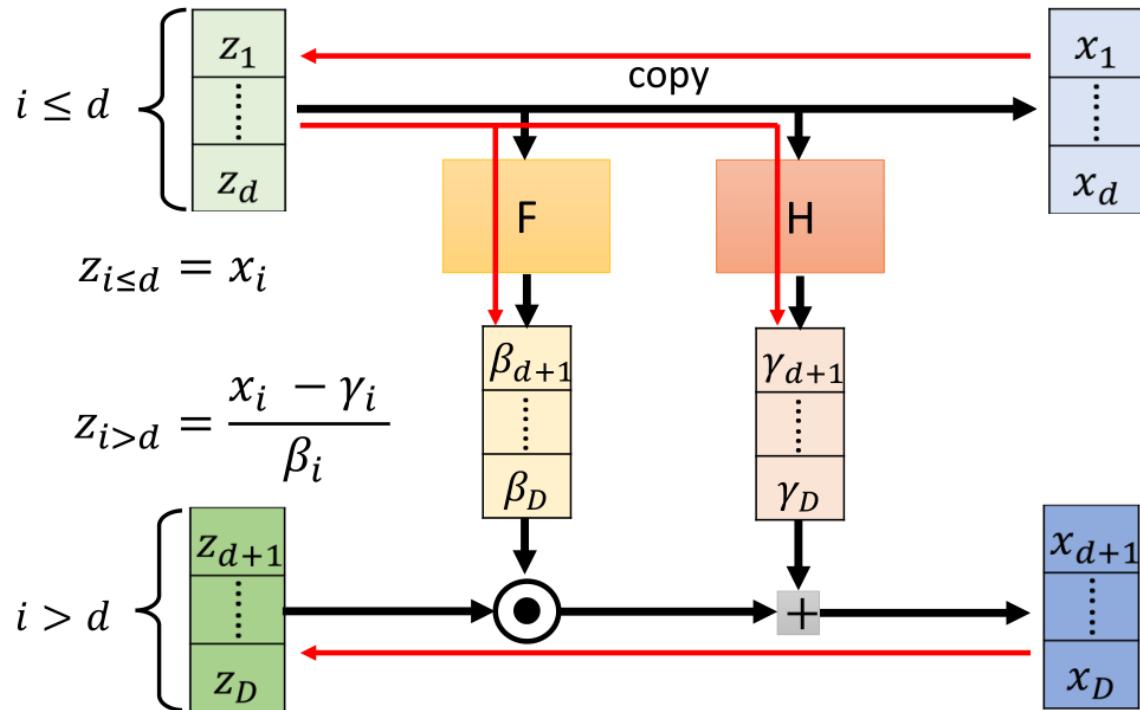


Inverse

z, x 都被分成两部分, 前 d 维直接copy, 因此求逆也是直接copy。

后 $D - d$ 维使用两个函数进行仿射变化, 可以将 $x_{i>d}; z_{i>d}$ 之间看作线性关系, 因此求逆时直接进行反操作即可。

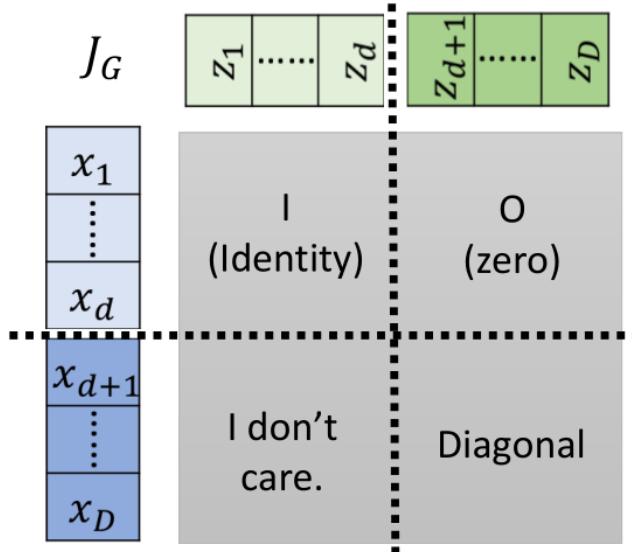
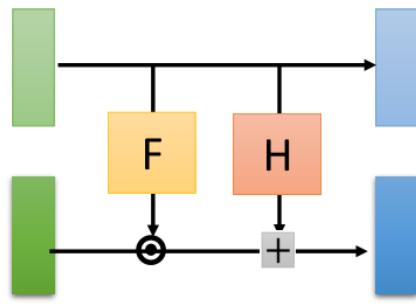
F, H 可以是任意复杂的函数, 我们并不需要求他的逆。在逆向过程中, 容易得到 $z_{i \leq d} = x_i$ 和 $z_{i>d} = \frac{x_i - \gamma_i}{\beta_i}$ 。



解决完 G^{-1} 部分, 还需要求解生成器对应的雅可比矩阵 J_G 的行列式

Jacobian

Coupling Layer



$$\det(J_G)$$

$$= \frac{\partial x_{d+1}}{\partial z_{d+1}} \frac{\partial x_{d+2}}{\partial z_{d+2}} \dots \frac{\partial x_D}{\partial z_D}$$

$$= \beta_{d+1} \beta_{d+2} \dots \beta_D$$

$$x_{i>d} = \beta_i z_i + \gamma_i$$

我们可以将生成器对应的雅克比矩阵分为以上的四个子块，左上角由于是直接copy的，所以对应的部分应该是一个单位矩阵，右上角中由于 x_1, \dots, x_d 与 z_{d+1}, \dots, z_D 没有任何关系，所以是一个零矩阵，而左下角 We don't care，因为行列式的右上角为0，所以只需要求解主对角线上的值即可。

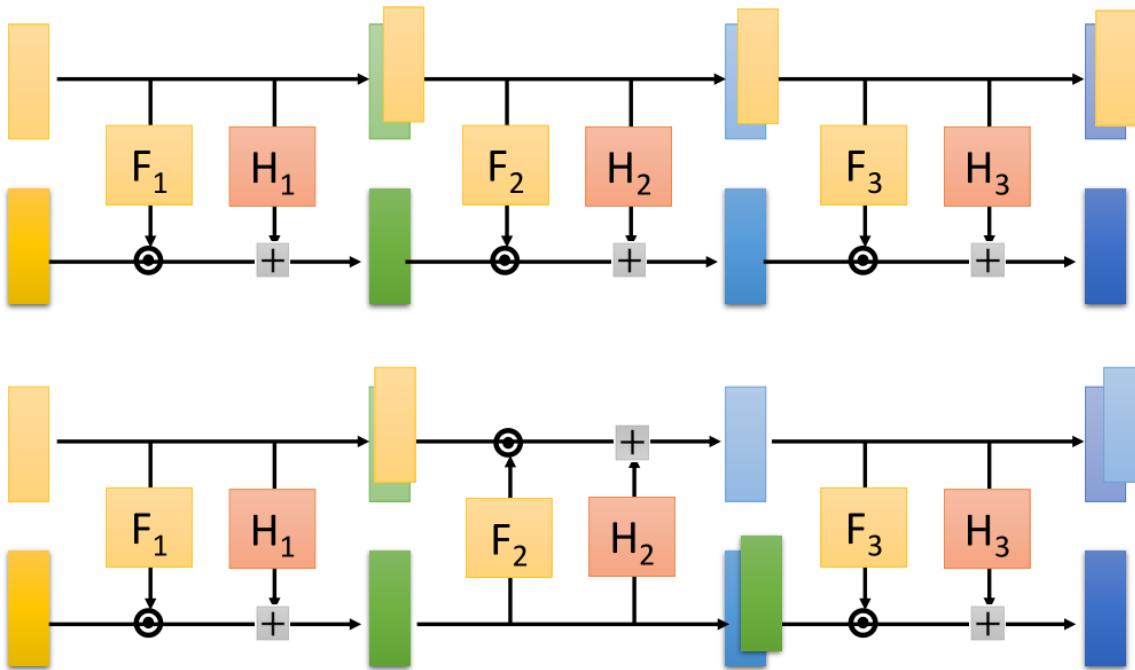
右下角由于 $x_i = \beta_i z_i, i > d$ ，是一对一的关系，因此是对角阵，对角线上的元素分别为 $\beta_{d+1}, \dots, \beta_D$ 。

所以上述的 $|\det(J_G)| = |\beta_{d+1} \beta_{d+2} \dots \beta_D|$ 。

那么这么一来coupling layer的设计把 G^{-1} 和 $\det(J_G)$ 这个问题都解决了

Stacking

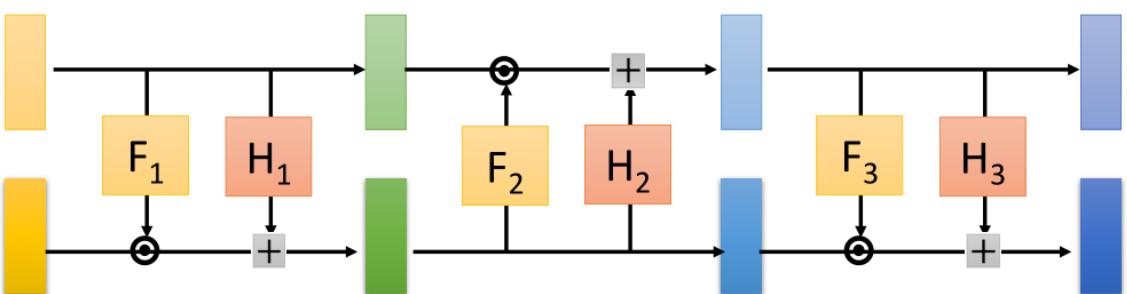
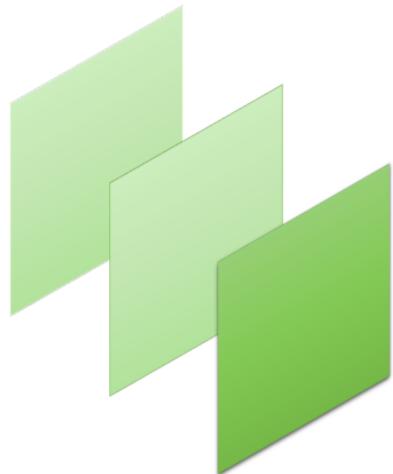
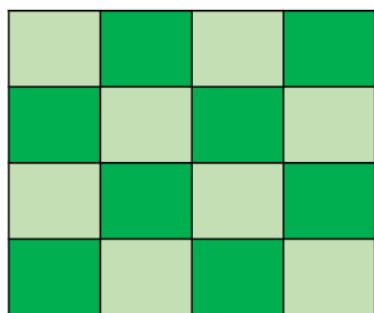
我们将多个耦合层堆叠在一起，从而形成一个更完整的生成器。但是这样会有一个新问题，就是最终生成数据的前 d 维与初始数据的前 d 维是一致的，这会导致生成数据中总有一片区域看起来像是固定的图样（实际上它代表着来自初始高斯噪音的一个部分），我们可以通过将复制模块（copy）与仿射模块（affine）交换顺序的方式去解决这一问题。



如上图所示，通过将某些耦合层的copy与affine模块进行位置上的互换，使得每一部分数据都能走向 copy->affine->copy->affine 的交替变换通道，这样最终的生成图像就不会包含完全copy自初始图像的部分。值得说明的是，在图像生成当中，这种copy与affine模块互换的方式有很多种，下面举两个例子来说明：

像素维度划分/通道维度划分

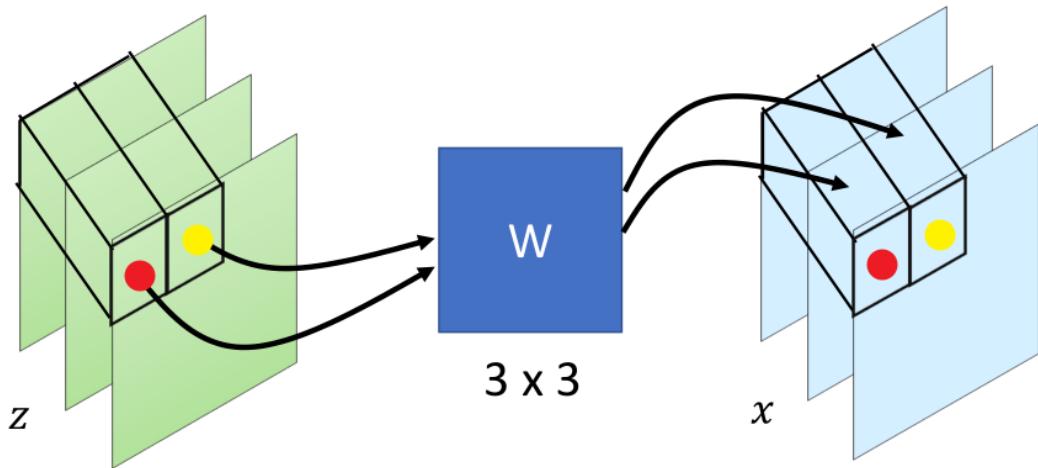
Coupling Layer



上图展示了两种按照不同的数据划分方式做 copy 与 affine 的交替变换。左图代表的是在像素维度上做划分，即将横纵坐标之和为偶数的划分为一类，和为奇数的划分为另外一类，然后两类分别交替做 copy 和 affine 变换（两两交替）；右图代表的是在通道维度上做划分，通常图像会有三通道，那么在每一次耦合变换中按顺序选择一个通道做 copy，其他通道做 affine（三个轮换交替），从而最终变换出我们需要的生成图形出来。

1×1 convolution layer

更进一步地，如何进行 copy 和 affine 的变换能够让生成模型学习地更好，这是一个可以由机器来学习的部分，所以我们引入 W 矩阵，帮我们决定按什么样的顺序做 copy 和 affine 变换，这种方法叫做 1×1 convolution（被用于知名的GLOW当中）。



W can shuffle the channels.

If W is invertible (?), it is easy to compute W^{-1} .

$$\begin{array}{c|c} \begin{matrix} 3 \\ 1 \\ 2 \end{matrix} & = & \begin{matrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix} & \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} \end{array}$$

W can shuffle the channels. 所以copy时可以只copy第一个channel，反正 1×1 convolution会在适当的时候对channel进行对调。

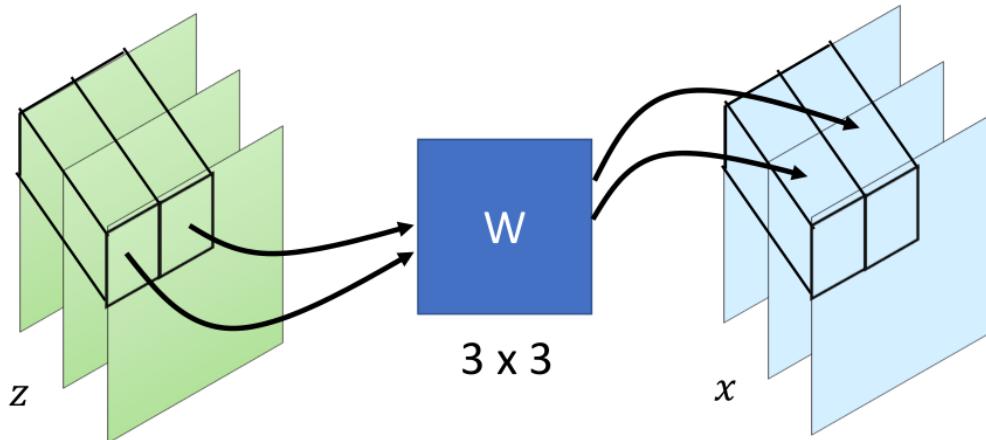
1×1 convolution 只需要让机器决定在每次仿射计算前对图片哪些区域实行像素对调，而保持 copy 和 affine 模块的顺序不变，这实际上和对调 copy 和 affine 模块顺序产生的效果是一致的。

比如右侧三个通道分别是1, 2, 3, 经过一个 W 矩阵就会变成3, 1, 2, 相当于对通道调换了顺序。而 coupling layer 不要动，只copy某几个channel。至于channel如何交换，需要机器学出来。

W 也在 G 里面，也需要invertible。

1x1 Convolution

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$



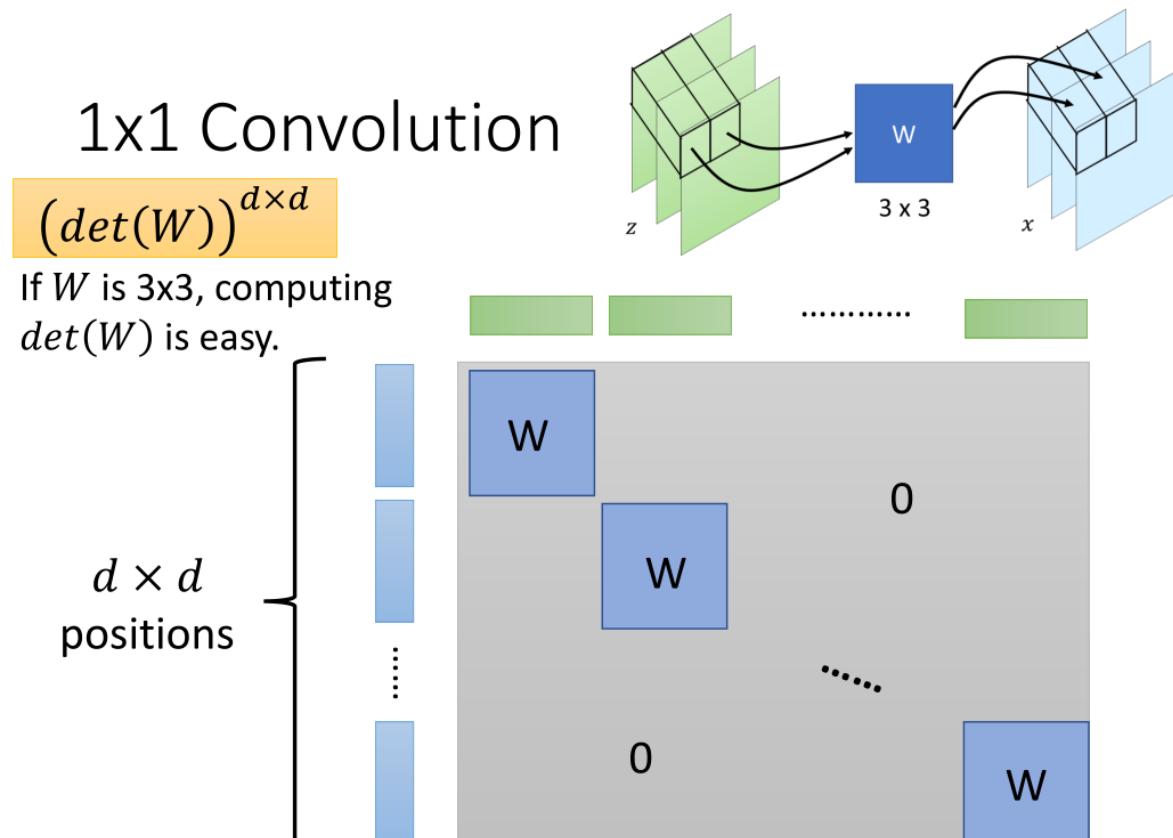
$$x = f(z) = Wz$$

$$J_f = \begin{bmatrix} \partial x_1 / \partial z_1 & \partial x_1 / \partial z_2 & \partial x_1 / \partial z_3 \\ \partial x_2 / \partial z_1 & \partial x_2 / \partial z_2 & \partial x_2 / \partial z_3 \\ \partial x_3 / \partial z_1 & \partial x_3 / \partial z_2 & \partial x_3 / \partial z_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = W$$

下面我们看一下，将W引入flow模型之后，对于原始的Jacobian行列式的计算是否会有影响。

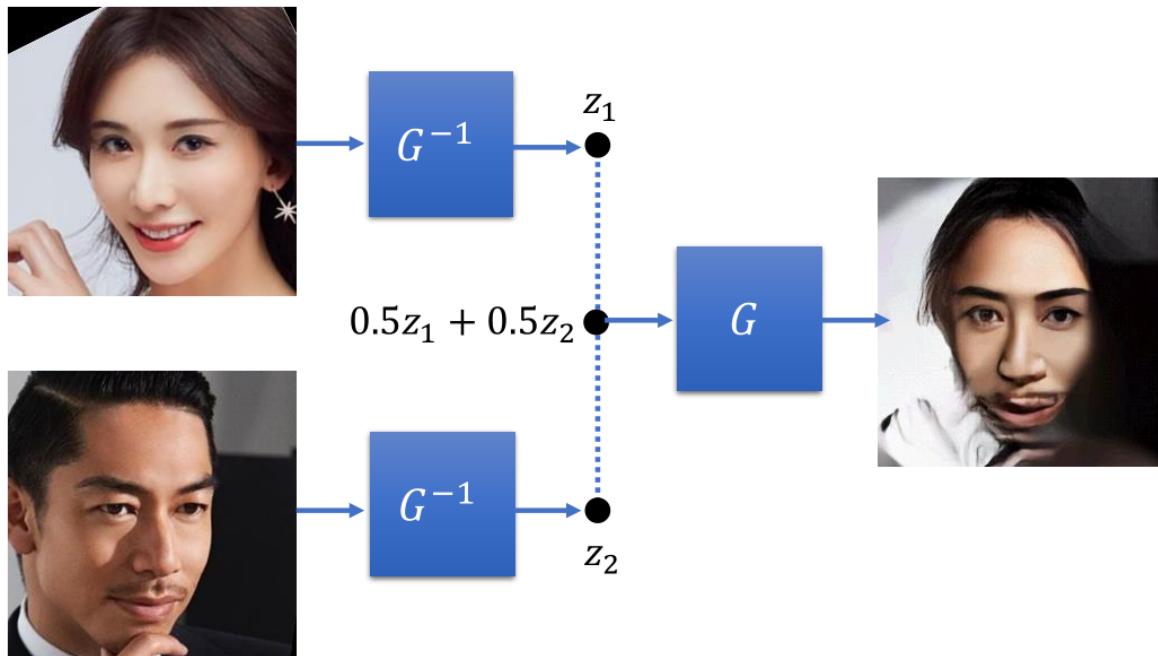
对于每一个3*3维划分上的仿射操作来说，由 $x = f(z) = Wz$ ，可以得到Jacobian行列式的计算结果就是 W

代入到整个含有 $d \times d$ 个3*3维的仿射变换矩阵当中，只有对应位置相同的时候才会有权值，得到最终的Jacobian行列式的计算结果就为 $(\det(W))^{d \times d}$

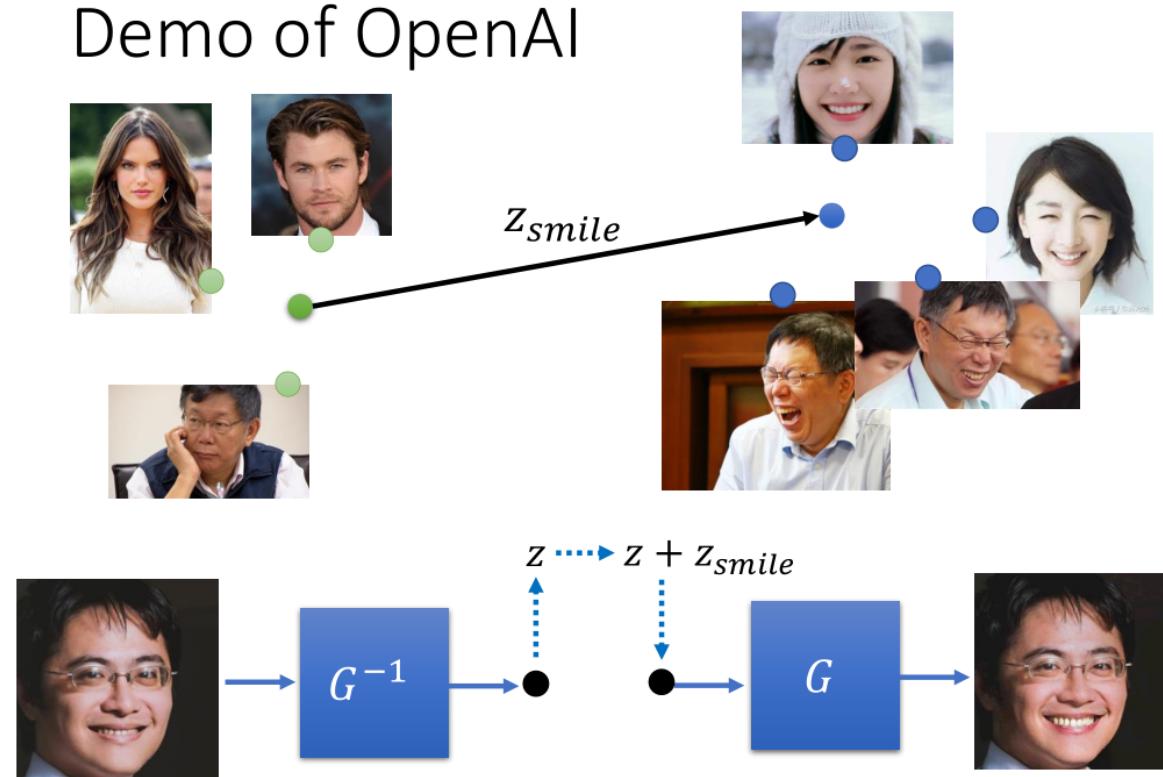


因此，引入 1×1 convolution后的 G 的Jacobian行列式计算依然非常简单，所以引入 1×1 convolution是可取的，这也是GLOW这篇Paper最有突破和创意的地方。

Demo of OpenAI



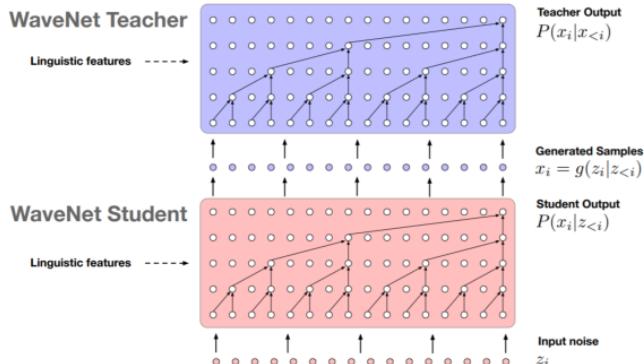
Demo of OpenAI



To Learn More

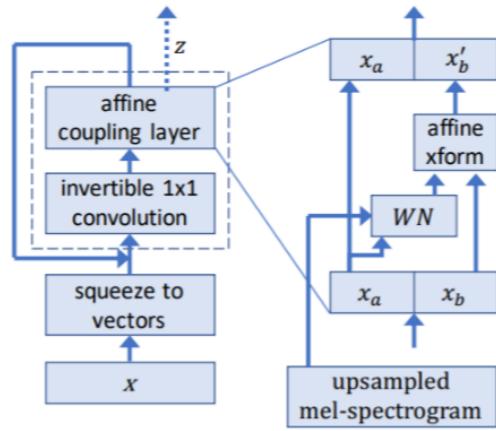
Flow-based Model可以用于语音合成

Parallel WaveNet



<https://arxiv.org/abs/1711.10433>

WaveGlow



<https://arxiv.org/abs/1811.00002>

综上，关于 Flow-based Model 的理论讲解和架构分析就全部结束了，它通过巧妙地构造仿射变换的方式实现不同分布间的拟合，并实现了可逆计算和简化雅各比行列式计算的功能和优点，最终我们可以通过堆叠多个这样的耦合层去拟合更复杂的分布变化，从而达到生成模型需要的效果。