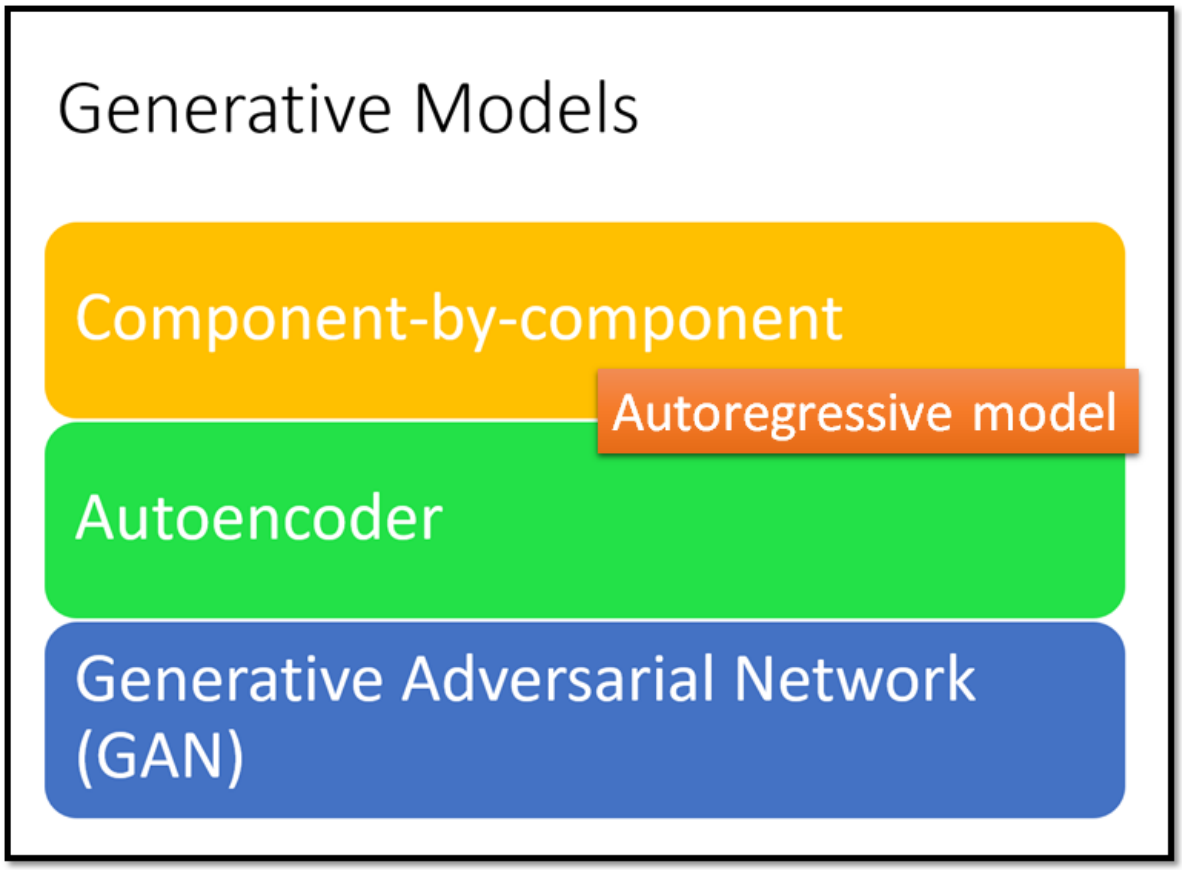
Flow-based Generative Model

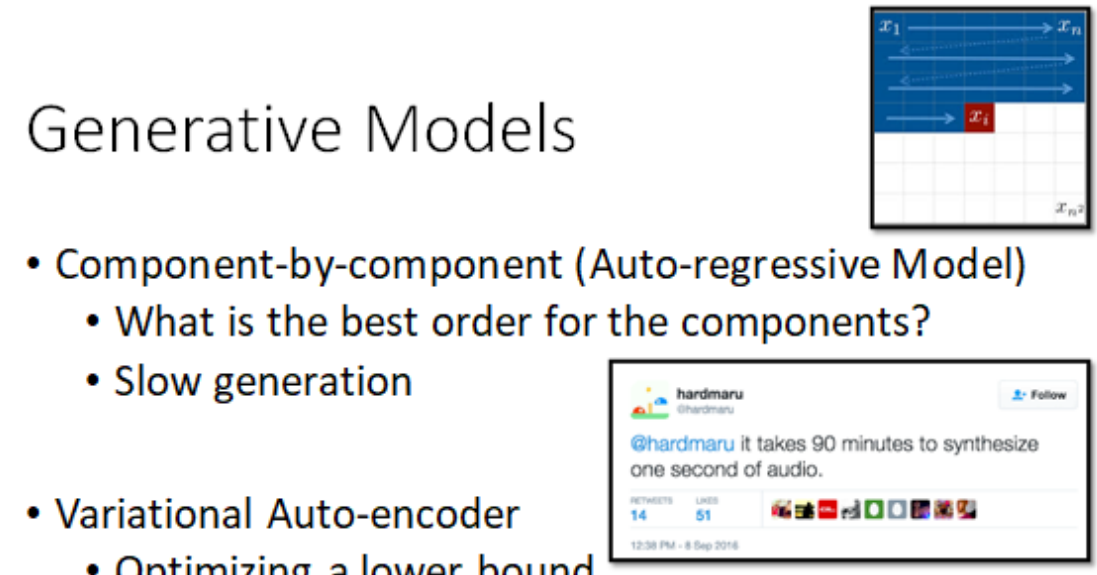
本节要讲一个新的生成模型，一个新的Generative 技术，它和GAN的目的相同，但是不如GAN有名，而且实际上也没有胜过GAN，但是这个方法有比较新颖的思想所以还是讲一讲。这个方法叫做Flow-based Generative Model，这里的Flow就是流的意思，后面会解释为什么说它是流。



> Link: https://youtu.be/YNUek8ioAJk
>
> Link: https://youtu.be/8zomhgKrsmQ

上图是从以前的课程中截取出来的，如果你有看过以前的课程（链接如上）你就能知道这里的三个方法的具体细节。我们之前说Generative Model 有三种，第一种Component-by-component，上课的时候我们以生成宝可梦图片为例子，以像素为component 一个像素一个像素的生成，这种方法更常见的名字是Autoregressive Model。我们也讲过Variational Autoencoder（VAE），和Generative Adversarial Network（GAN），上述三种是我们之前见过的生成模型，今天我们要介绍第四种Flow-based Generative Model 。

过去的Generative Models 的缺点



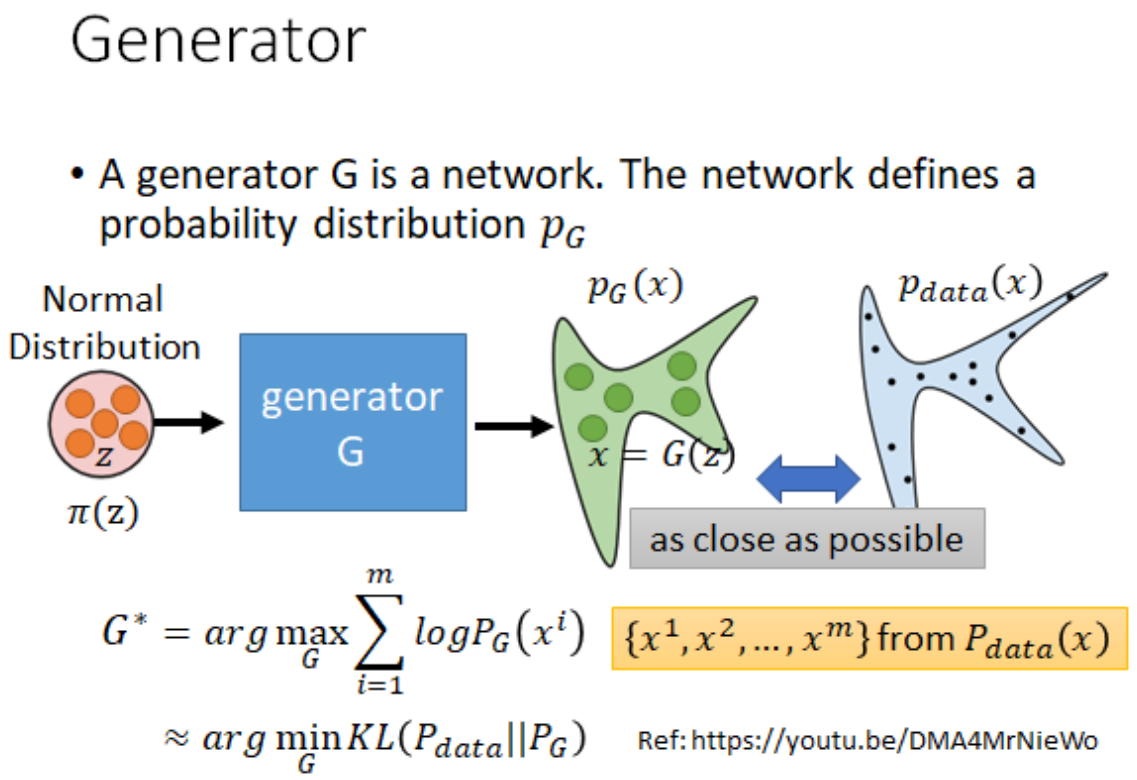
我们之前介绍的生成模型都有各自的问题，如上图所示。

Auto-regressive Model 是一个一个component 地生成的，所以component 的最佳生成顺序是难以求解的。比如说生成图像的例子中，我们就是一个一个pixel 地，从左上角开始一行一行地生成。当然，还有一些任务的生成目标就是有顺序的，比如说生成语音，后面的component 依赖前面的component，这样的目标比较适合用Auto-regressive Model，但是现在这个模型太慢了，如果你要生成一秒像是人类说的语音你需要做90分钟合成，这显然是不实用的。

Variational Auto-encoder 介绍这个模型的时候我们证明了，VAE optimize的对象是likelihood 的lower bound，它不是去maximize 我们要它maximize 的likelihood 而是去maximize likelihood 的lower bound，我们不知道这个lower bound 和真正想要maximize 的对象的差距到底有多大。Flow-base generative model 会解决这个问题，它并不会用approximation，并不是optimize lower bound 而是optimize probability 的本体，后面我们会介绍flow 是怎么做到这件事的。

Generative Adversarial Network 是现在做生成得到的目标质量最好的模型。但是我们都知道GAN是很难train的，因为你的Generative 和Discriminator 目标是不一致的，很容易train崩掉。

Generator 的内涵



我们抛开具体模型，先来看看Generator 的定义。Generator 是一个网络，该网络定义了一个可能性分布p_G 。怎么理解这句话呢，看上图，有一个Generator G 输入一个 z 输出一个 x ，如果现在是在做人脸生成， x 是一张人脸图片，可以看成一个高维向量其中每一个element 就是图片的一个pixel， z 我们通常假设它是从一个简单的空间中sample 出来的，比如说Normal Distribution，虽然 z 是从一个简单的分布中sample 出来的，但是通过 G 以后 x 可能会形成一个非常复杂的分布，这个Distribution 我们用 P_G 来表示。对我们而言我们会希望通过 G 得到的Distribution 能和real data 也就是真实人脸数据集的Distribution P_data 尽可能相同。

那怎么能让 P_G 和 P_data 尽可能相同呢，常见的做法是我们训练G的时候训练目标定为maximize likelihood ，讲的具体一点就是从人脸图片数据集中sample 出m笔data $\{x^1, x^2, \ldots, x^m\}$ ，然后你就是希望这m张图片是从 P_G 这个分布中sample 出来的概率越大越好。就是maximize 上图中左下角的公式，这就是我们熟知的maximize likelihood。如果你难以理解为什么要让产生这些data 的概率越大越好的话，老师提供了一个更直观的解释：maximize likelihood 等同于minimize P_G 和 P_data 的*K-L*变换。Ref: https://youtu.be/DMA4MrNieWo

总而言之就是让 P_G 和 P_data 这两个分布尽可能相同。

Flow 这个模型有什么厉害的地方呢？Generator 是个一个网络，所以 P_G 显然非常复杂，一般我们不知道怎么optimize objective function，而Flow 可以直接optimize objective function，可以直接maximize likelihood。

Flow 的数学基础

math warning

Math Background

Jacobian, Determinant, Change of Variable Theorem

这可能也是Flow-based Generative Model 不太出名的一个原因，其他生成模型都比较容易理解，而此模型用到了比较多的数学知识。你要知道上图中的三个概念：Jacobian、Determinant、Change of Variable Theorem 。

Jacobian Matrix

Jacobian Matrix

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$x = f(z) \quad z = f^{-1}(x)$$

$$J_f = \begin{bmatrix} \partial x_1/\partial z_1 & \partial x_1/\partial z_2 \\ \partial x_2/\partial z_1 & \partial x_2/\partial z_2 \end{bmatrix}$$

input

output

$$J_{f^{-1}} = \begin{bmatrix} \partial z_1/\partial x_1 & \partial z_1/\partial x_2 \\ \partial z_2/\partial x_1 & \partial z_2/\partial x_2 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} z_1 + z_2 \\ 2z_1 \end{bmatrix} = f\left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\right)$$

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

$$\begin{bmatrix} x_2/2 \\ x_1 - x_2/2 \end{bmatrix} = f^{-1}\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right)$$

$$J_f = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}$$

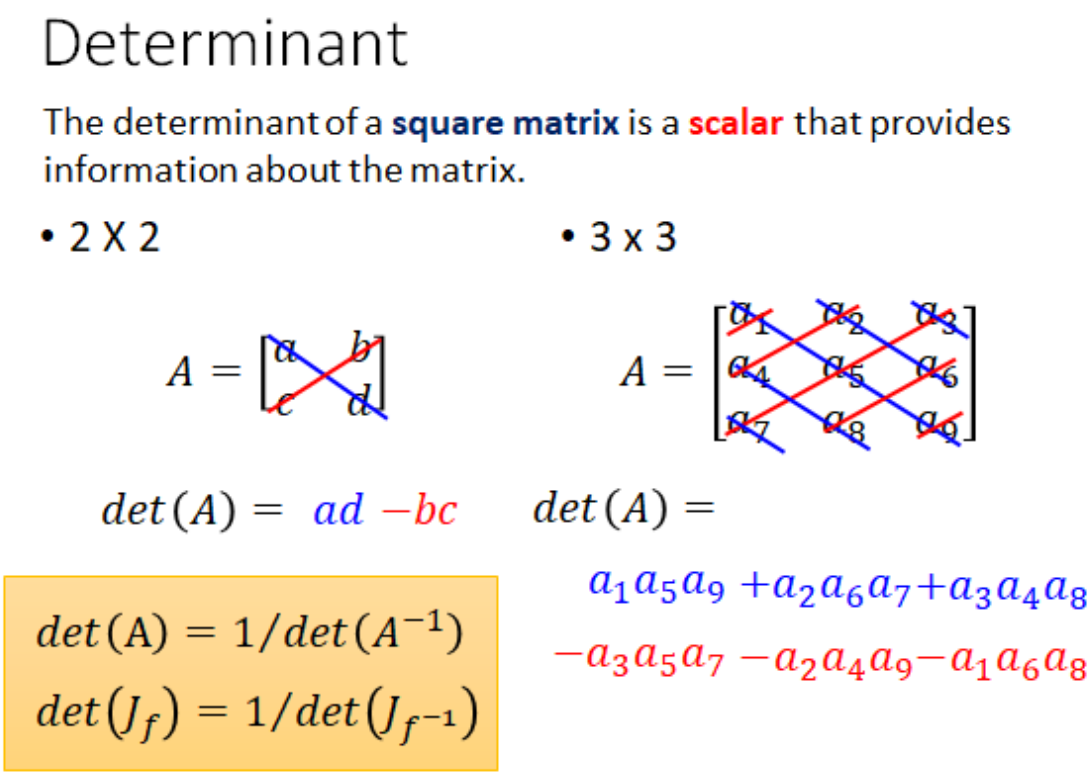
$$J_{f^{-1}} = \begin{bmatrix} 0 & 1/2 \\ 1 & -1/2 \end{bmatrix}$$

$$J_f J_{f^{-1}} = I$$

把 f 看作生成模型，z 就是输入，x 就是生成的输出。这里我们的栗子中输入输出的维度是相同的，实际做的时候往往是不同的。Jacobian Matrix 记为 J_f ，就定义为输出和输入向量中element 两两组合做偏微分组成的矩阵，如上图左下角所示，输出作为列，输入作为行， J_f 每个element 都是输出对输入做偏微分。 $J_{f^{-1}}$ 的定义同上只是变成了输入对输出的偏微分。

举一个栗子，如上图右侧，自己看一下就好，就不赘述了。需要强调一下的是 $J_f J_{f^{-1}}$ 得到单位矩阵，function 之间有inverse的关系，得到的Jacobian Matrix 也有inverse的关系。

Determinant



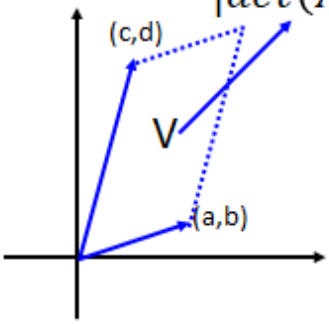
Determinant： 方阵的行列式是一个标量，它提供有关方阵的信息。你只要记得上图左下角行列式的性质，矩阵的det和矩阵逆的det互为倒数。

Determinant

- 2 X 2

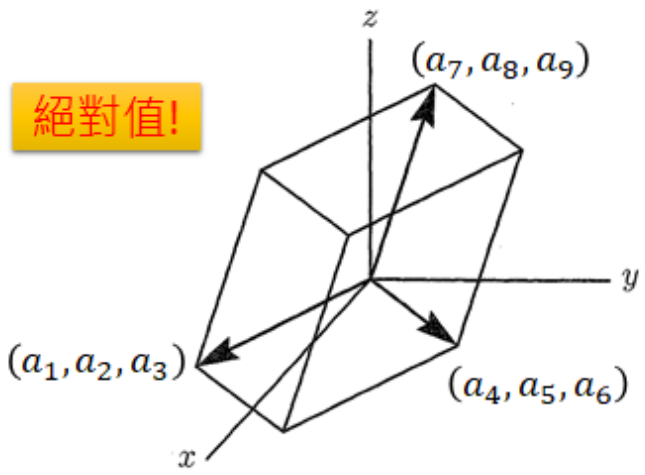
$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$|det(A)|$ 絕對值!



- 3 x 3

$$A = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix}$$

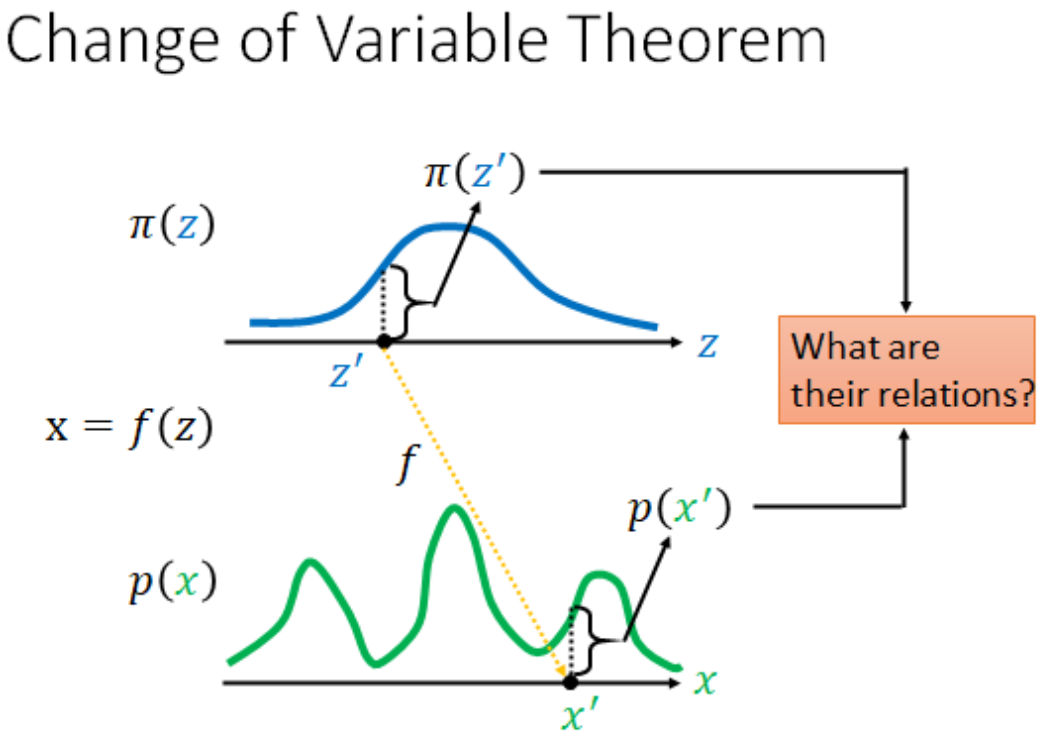


矩阵的行列式的绝对值可以表示行列式每一行表示的向量围成的空间的"面积"（3维空间中是体积，更高维空间中也是类似的概念）。或者你可以想象它是矩阵所表示的线性变换前后的“面积”放缩比例。

♡关于行列式的含义你可以参考3Blue1Brown的线性代数的本质05行列式 。

Change of Variable Theorem

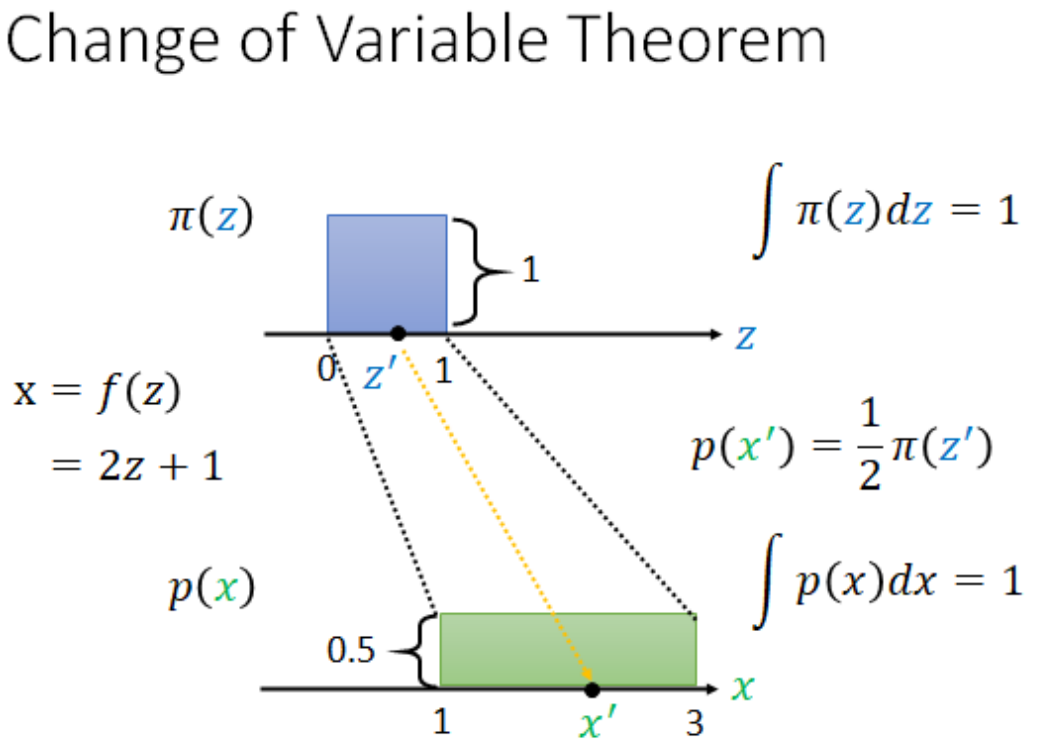
Determinant 就是为了解释Change of Variable Theorem



假设我们有一个分布 $\pi(z)$，z 带入 f 会得到 x，x 形成了分布 $p(x)$，我们现在想知道 $\pi(z)$ 和 $p(x)$ 之间的关系。如果我们可以写出两者之间的关系，就可以分析一个Generator 。

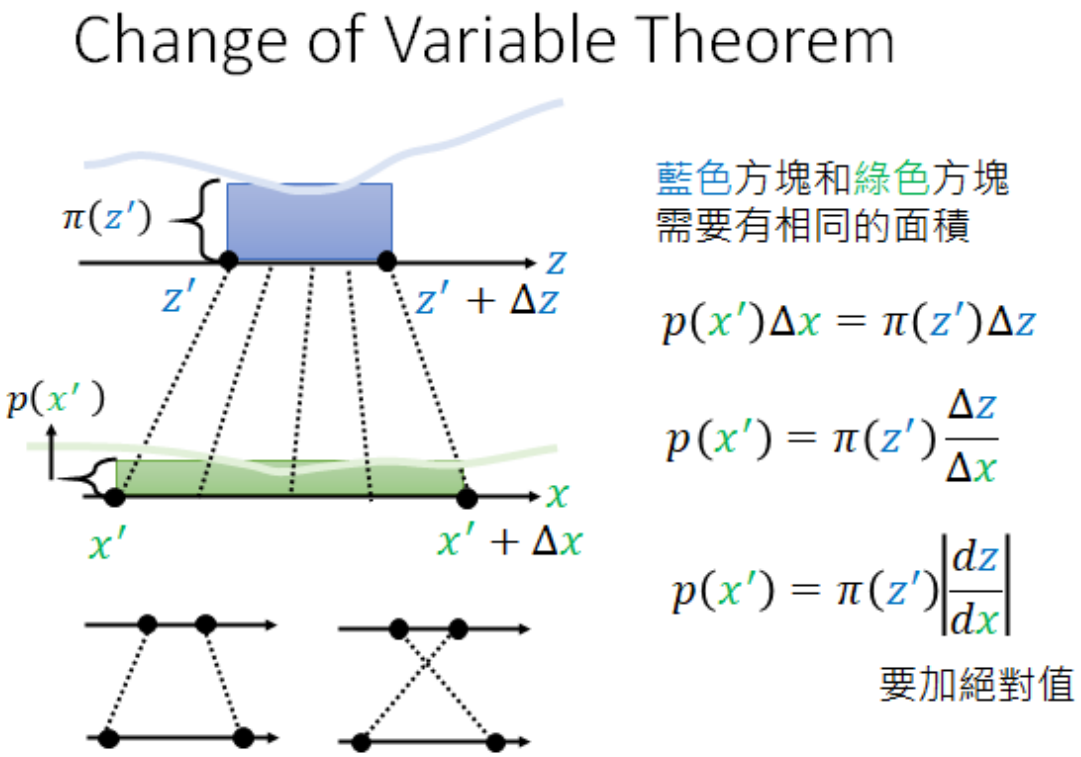
两者之间的关系能写出来吗？是可以的。我们现在要问的问题是：假设说分布 $\pi(z)$ 上有一个 z' 对应 $\pi(z')$，z' 通过 f 得到 x' 对应 $p(x')$，现在想知道 $\pi(z')$ 和 $p(x')$ 之间的关系是什么？

举一个简单的栗子来说明上述问题：



假设说两个分布如上图所示，我们知道probability density function 的积分等于1，所以两者的高度分别为1和$\frac{1}{2}$，假设 x 和 z 的关系是 $x = 2z + 1$，底就变成原来的两倍，高度就变成原来的二分之一。现在从 $\pi(z')$ 到 $p(x')$ 中间的关系很直觉就是二分之一的关系。

那我们再来看一个更general 的case：



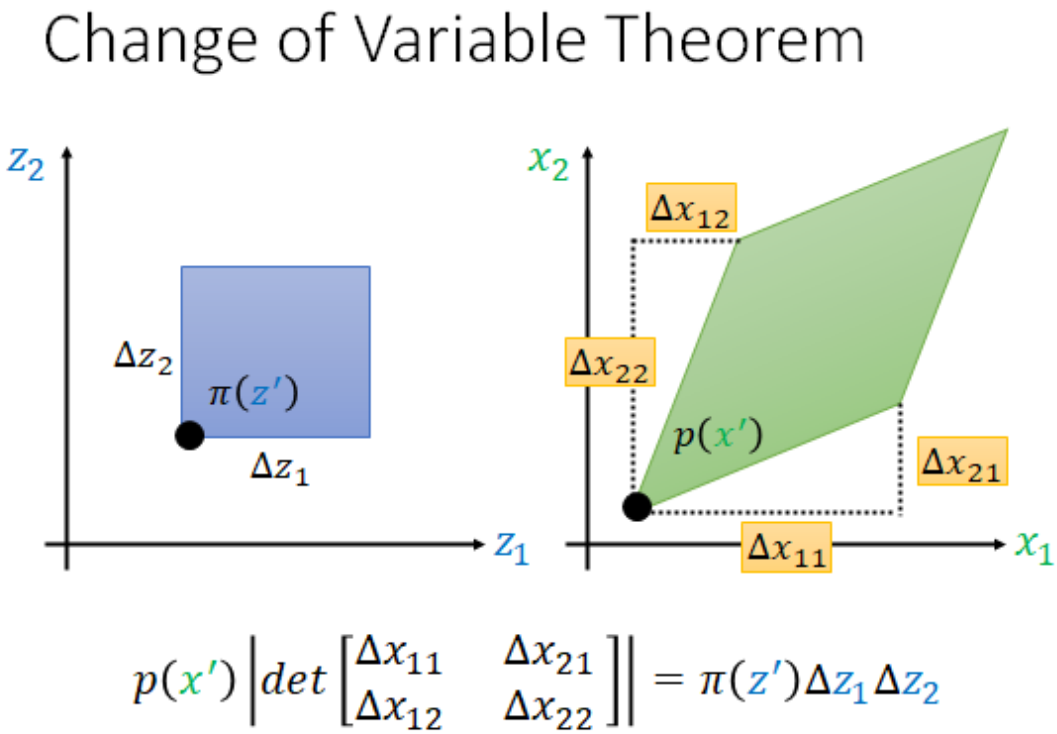
两个分布如上图浅蓝色和浅绿色线所示，现在我们不知道这两者的Distribution，假设我们知道 x 和 z 的关系即知道 f，那我们就可以求出 $\pi(z')$ 和 $p(x')$ 之间的关系，这就是Change of Variable Theorem 的概念。

我们把 z' 加一个 Δz 映射到 $x' + \Delta x$，此时上图中蓝色方块和绿色方块需要有同样的面积，所以有上图右侧所示公式。

$$p(x') = \pi(z')\left|\frac{dz}{dx}\right|$$

这个绝对值是因为有时会出现上图左下角的情况，通过 f 变换后，形成了交叉。

我们在举一个 x 和 z 都是二维的栗子：



我们在 z 的分布中取一个小小的矩形区域高是 Δz_2 宽是 Δz_1 ，它通过一个function f 投影到 x 的分布中变成一个菱形。我们用向量 $[\Delta x_{11}, \Delta x_{21}]$ 表示菱形的右下边，用向量$[\Delta x_{22}, \Delta x_{12}]$ 表示菱形的左上边。根据上面Determinant 一节中提到的知识点，我们可以用这两个向量组成的矩阵的determinant 表示该矩阵的面积。$\pi(z')$ 作为正方形为底的三维形体的高， $p(x')$ 作为以绿色菱形为底的三维形体的高，这两个形体的积分值相等，所以得到了如上图所示的公式。

另外再讲一下四个Δ值的意思：

Δx_{11}是z_1（维度上）改变的时候x_1（维度上）的变化量

Δx_{21}是z_1改变的时候x_2的变化量

Δx_{12}是z_2改变的时候x_1的变化量

Δx_{22}是z_2改变的时候x_2的变化量

整理公式得出结论

$$p(x')\left|det\begin{bmatrix}\Delta x_{11} & \Delta x_{21}\\ \Delta x_{12} & \Delta x_{22}\end{bmatrix}\right| = \pi(z')\Delta z_1 \Delta z_2 \qquad x = f(z)$$

$$p(x')\left|\frac{1}{\Delta z_1 \Delta z_2} det\begin{bmatrix}\Delta x_{11} & \Delta x_{21}\\ \Delta x_{12} & \Delta x_{22}\end{bmatrix}\right| = \pi(z')$$

$$p(x')\left|det\begin{bmatrix}\Delta x_{11}/\Delta z_1 & \Delta x_{21}/\Delta z_1\\ \Delta x_{12}/\Delta z_2 & \Delta x_{22}/\Delta z_2\end{bmatrix}\right| = \pi(z')$$

$$p(x')\left|det\begin{bmatrix}\partial x_1/\partial z_1 & \partial x_2/\partial z_1\\ \partial x_1/\partial z_2 & \partial x_2/\partial z_2\end{bmatrix}\right| = \pi(z')$$

$$p(x')\left|det\begin{bmatrix}\partial x_1/\partial z_1 & \partial x_1/\partial z_2\\ \partial x_2/\partial z_1 & \partial x_2/\partial z_2\end{bmatrix}\right| = \pi(z')$$

$$p(x')\left|det(J_f)\right| = \pi(z')$$

$$p(x') = \pi(z')\left|det(J_{f^{-1}})\right|$$

$$p(x') = \pi(z')\left|\frac{1}{det(J_f)}\right|$$

接下来就是做一波整理，相信大家都看得懂（看不懂就重学一下线代😂），就不赘述了。

最终的结论就是上图左下角的公式：

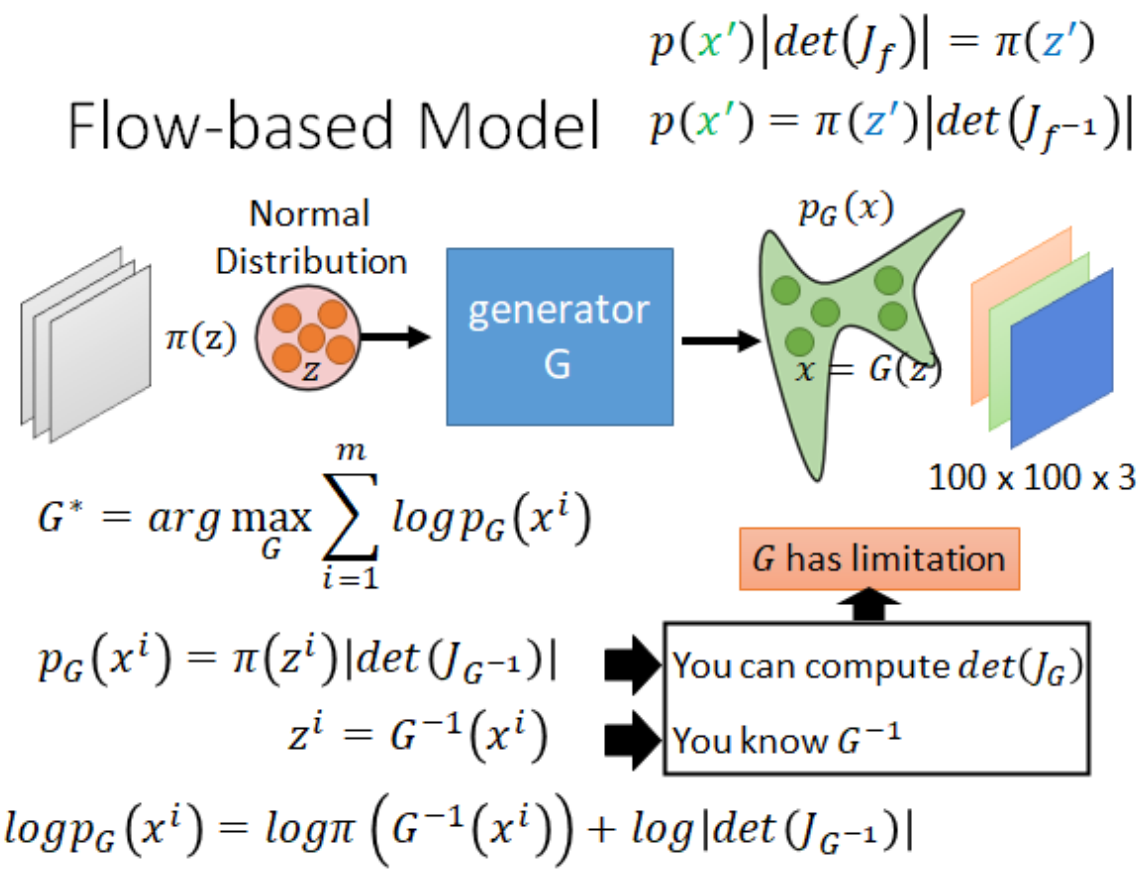
总而言之，$\pi(z')$ 乘上 $\left|det(J_{f^{-1}})\right|$ 就得到 $p(x')$，这就是 $\pi(z')$ 和 $p(x')$ 的关系。

如果你没看太懂数学推导，你就记住这个结论就好，$\pi(z')$ 和 $p(x')$ 之间就是差一个 $\left|det(J_{f^{-1}})\right|$。

end of math warning

Flow-based Model

接下来我们就正式进入Flow的部分，我们先回到Generation 上。



一个Generator 是怎么训练出来的呢，Generator 训练的目标就是要maximize $P_G(x^i)$ ，x^i 是从real data 中sample 出来的一笔data。那 $P_G(x^i)$ 长什么样子呢，有了前面的Change of Variable Theorem 的公式我们就可以把 $P_G(x^i)$ 写出来了：

$$P_G(x^i) = \pi(z^i)\,|det(J_{G^{-1}})|$$

而 z^i 就是把x^i 带入G inverse 以后的结果。

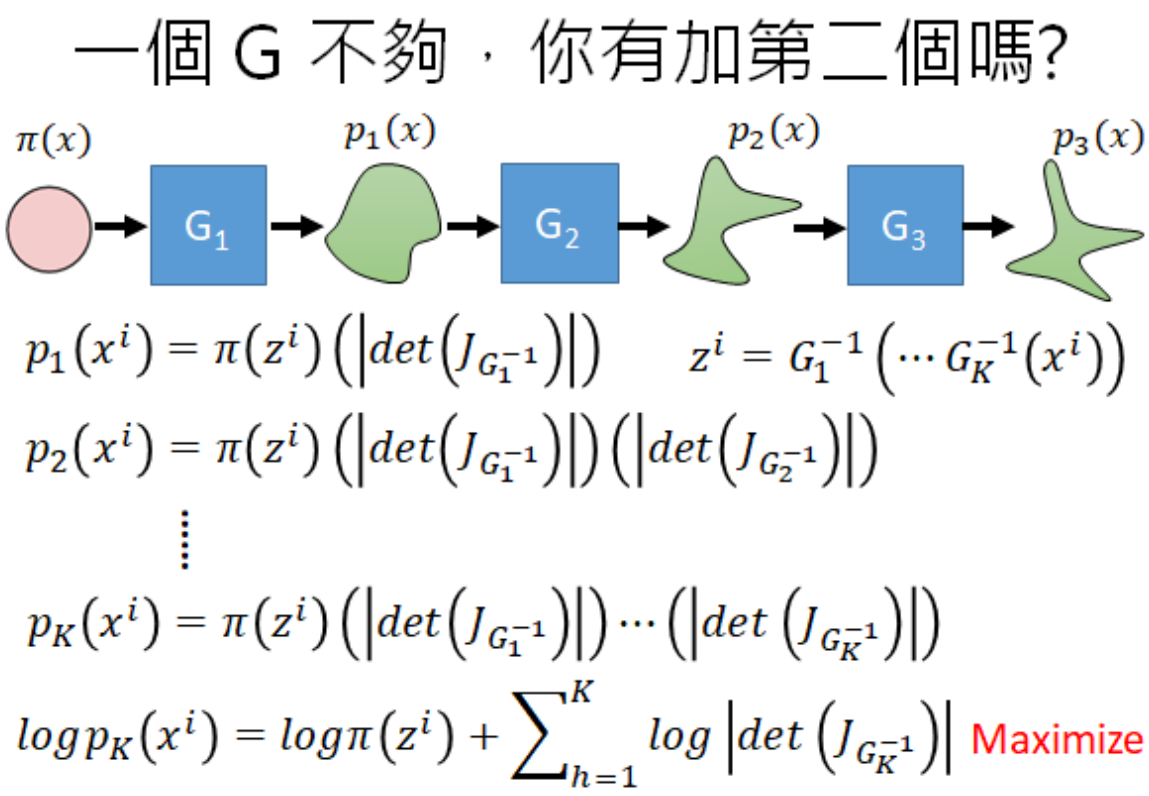
我们在给上式加一个log，把公式右侧的乘转为加。现在我们知道 $log(P_G(x^i))$ 长什么样子了，我们只要能maximize 这个式子我们就可以train Generator 就结束了，但是我们要想maximize 这个式子需要有一些前提：

- 要知道如何计算 $det(J_G)$
- 要知道 G^{-1}

要做Generator 的Jacobian 其实还比较好做，你只要知道如何计算$\partial x/\partial z$ ，但是问题是这个Jacobian Matrix 可能会很大，比如说 x 和 z 都有1000维，那你的Jacobian Matrix 就是一个1000*1000的矩阵，你要算它的det，计算量是非常大的，所以我们要好好设计Generator ，让它的Jacobian Matrix 的det好算一些。

另一个前提是：我们要把 z^i 变成 x^i ，就要知道G^{-1} 。我们同时也要让$det(J_{G^{-1}})$ 变得容易算。为了确保G是invertible，我们设计的Generator 的时候 z 和 x 的维度是相同的。 z 和 x 的维度相同不能保证G一定是invertible，但是如果不相同G一定不是invertible。

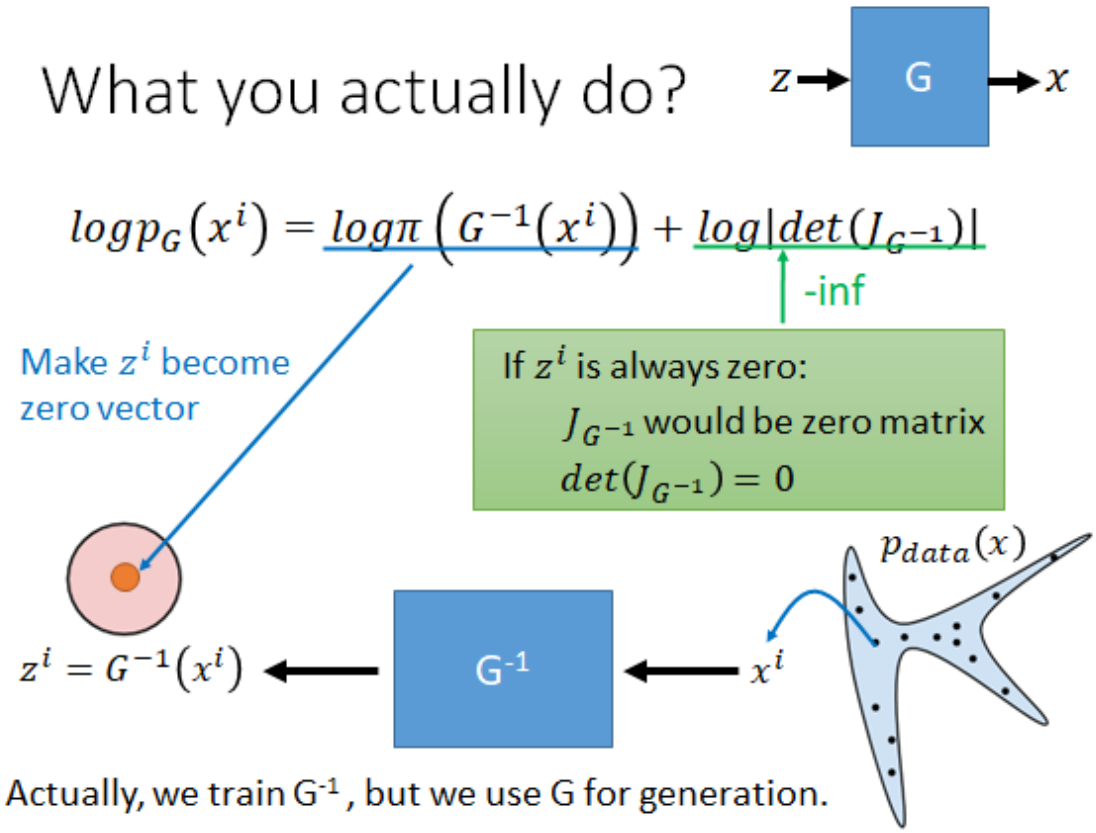
所以，到这里我们就能看出来，Flow-based Model 和GAN 不一样，GAN是低维生成高维，但是Flow这个技术中输入输出的维度必须相同，这一点是有点奇怪的。另外，为了保证G是可逆的，我们要限制G的架构，这样的做法必然会限制整个生成模型的能力。



既然一个G不够强，你就多加一些。最终maximize的目标就是上图最下面的公式。

$$G^* = arg\max_G \sum_i^m log(p_K(x^i))$$

实作上怎么做Flow-based Model



我们先考虑一个G，在公式中其实只有出现 G^{-1} ，所以实际上我们在训练的时候我们训练的是 G^{-1} ，在生成的时候把 G^{-1} 倒过来用 G 生成目标。在训练的时候，从real data 中sample 一些数据出来，输入 G^{-1} 中得到 z^i 。

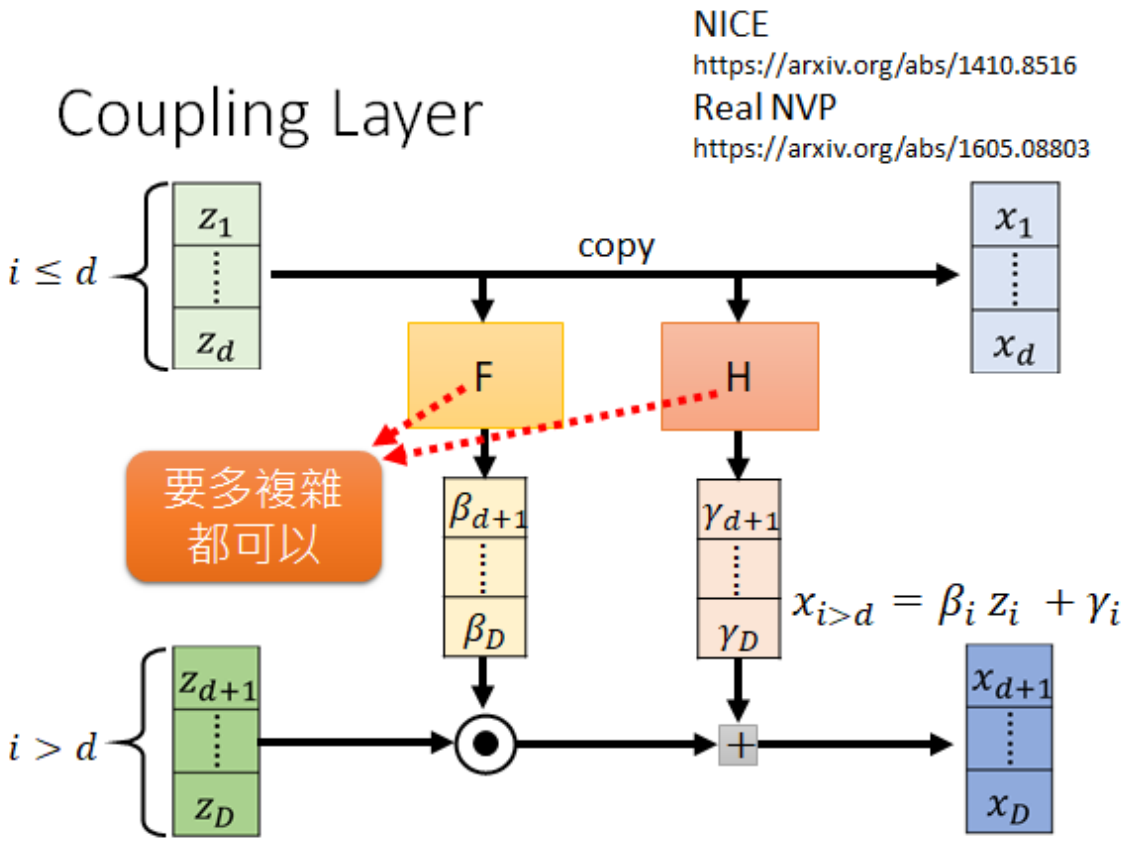
看一下上图中的公式，我们的目标是maximize 它，这个公式有两项，先看第一项。因为 π 是一个normal distribution，要这一项的值最大只要让 $G^{-1}(x^i)$ 是零向量就好了，但是显然这么做会出大问题，如果 z^i 变0，$J_{G^{-1}}$ 将是一个零矩阵，则 $det(J_{G^{-1}}) = 0$ 。此时第二项，就是负无穷。

为了maximize $log(P_G(x^i))$ ，如上图所示，一方面前一项要让 $G^{-1}(x^i)$ 向原点靠拢，一方面第二项又会对 G^{-1} 做一些限制，让它不会把所有的 z mapping 到零向量。

这就是我们在实作Flow-based Model 时所做的事情。

Coupling Layer

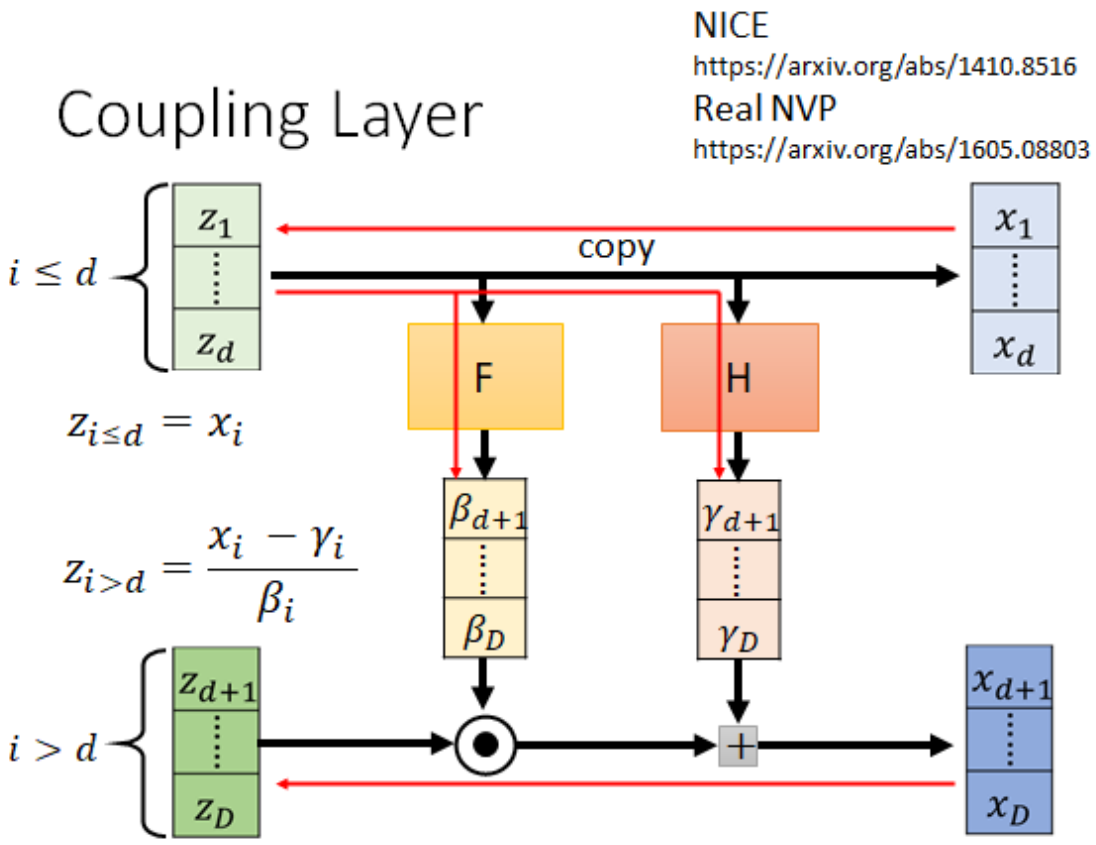
coupling layer 是一个实用的G，NICE 和Real NVP 这两个Flow-based Model 都用到了coupling layer ，接下来将介绍这个coupling layer 的具体做法。



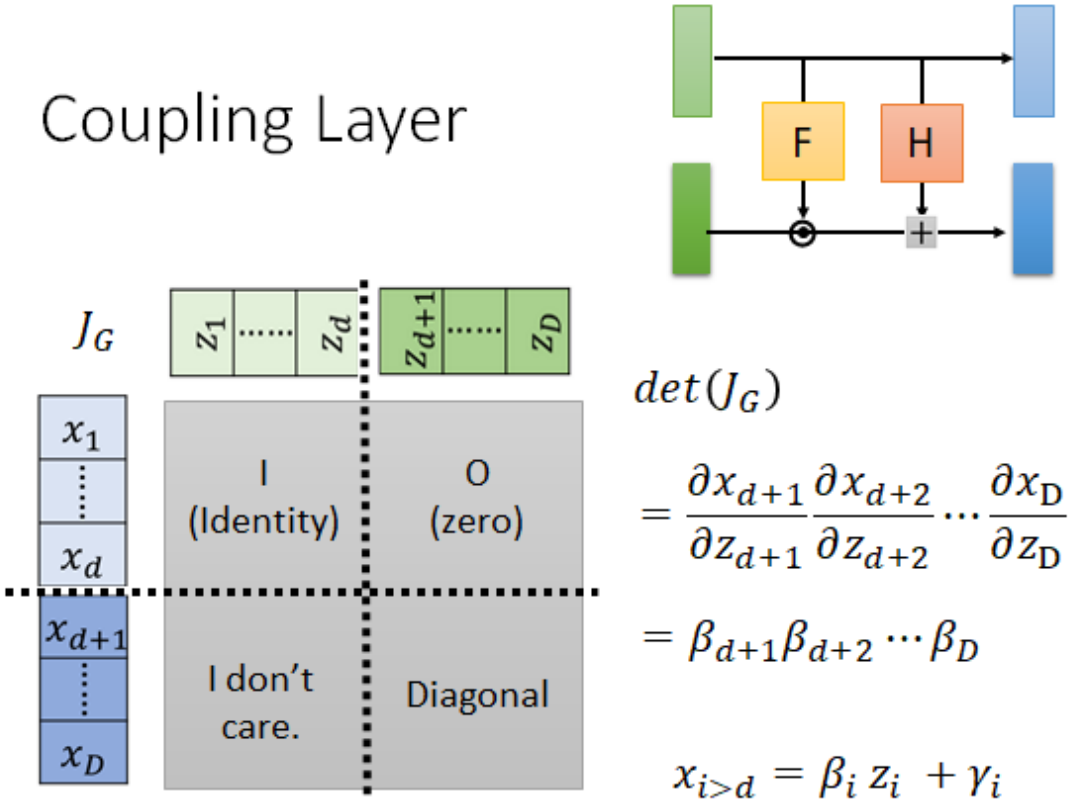
> NICE
>
> https://arxiv.org/abs/1410.8516
>
> Real NVP
>
> https://arxiv.org/abs/1605.08803

如上图所示，输入和输出都是D维向量，然后把输入输出的向量拆成两部分，前d维为一组，后D-d维为一组。对于上面的第一部分直接通过copy z得到x。对于第二部分，我们将第一部分通过两个变换F和H（这两个function 可以任意设计，多复杂都可以）得到向量β和γ，再将第二部分和β做内积再加上γ，就得到第二部分的x。

接下来要讲怎么取coupling layer 的inverse。就是说在不知道z，只有后面的x，这样的情况下怎么把z找出来呢?



对于第一部分直接把x copy过去就可以了，对于第二部分，我们就把刚才得到的z的第一部分通过F和H得到β和γ，然后将x的第二部分通过 $\frac{x_i-\gamma_i}{\beta_i}$ 就可以算出z，结束。



不只要能算G的inverse，还要算 J_G ，计算方法就如上图所示。

左上角显然是单位矩阵，因为x和z是copy的。

右上角是浅蓝色对应深绿色的部分，上图右上角可以明显看出两者没有关系，偏微分结果为0.

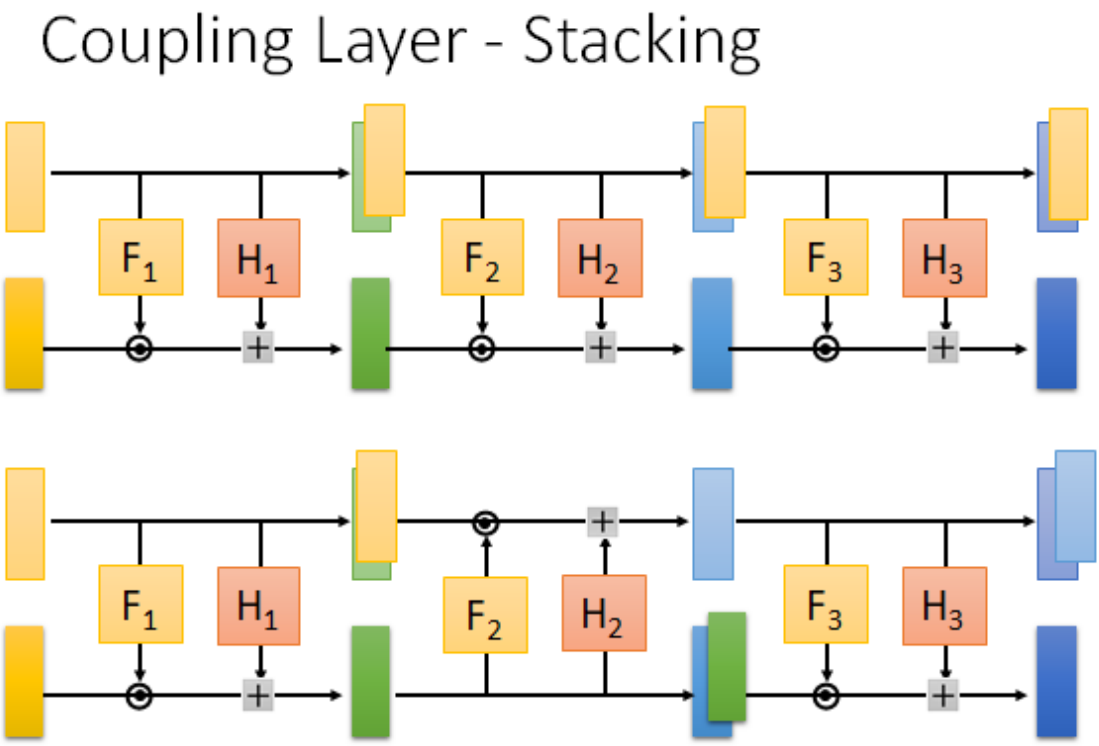
左下角我们并不关心了，因为现在矩阵的左上角是单位矩阵，右上角是零矩阵，整个矩阵的det就是右下角矩阵的det。

右下角矩阵是Diagonal，因为你想想看 z_{d+1} 只和 x_{d+1} 有关系，改行其他值都是0，同样的 z_D 只和 x_D 有关系，所以整个右下角矩阵就是一个对角矩阵。

所以这这个矩阵的det：

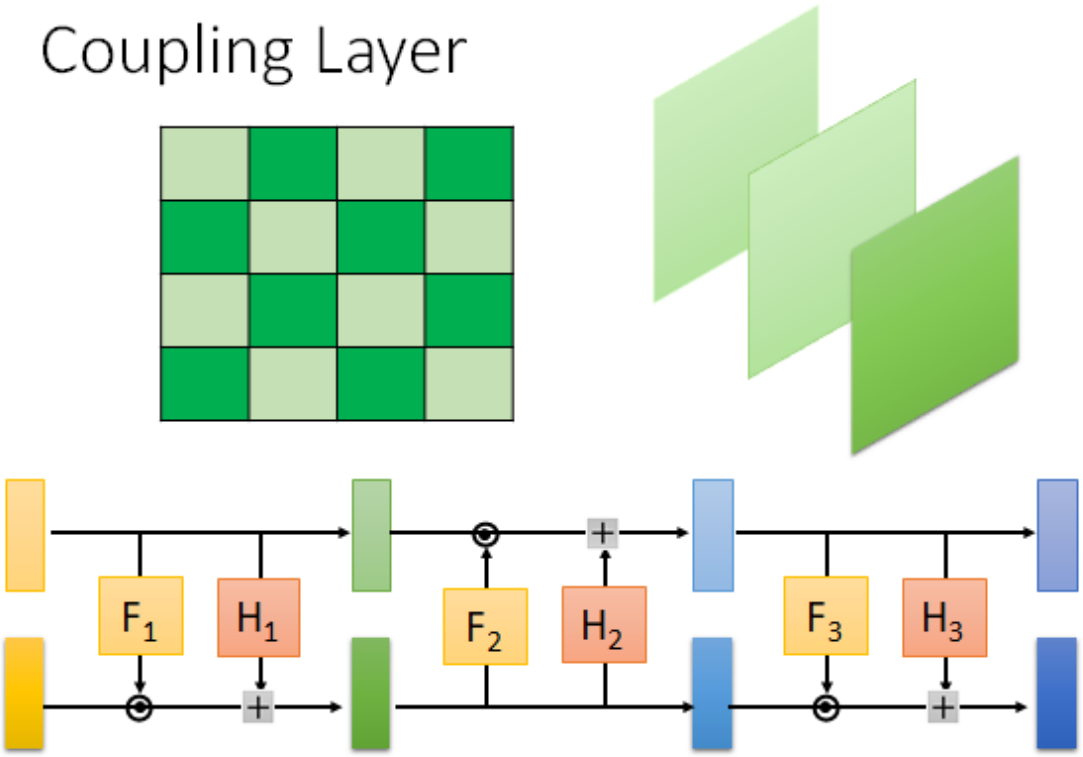
$$
\begin{aligned}
det(J_G) &= \frac{\partial x_{d+1}}{\partial z_{d+1}} \frac{\partial x_{d+2}}{\partial z_{d+2}} \cdots \frac{\partial x_D}{\partial z_D} \\
&= \beta_{d+1} \beta_{d+2} \cdots \beta_D
\end{aligned}
$$

stacking



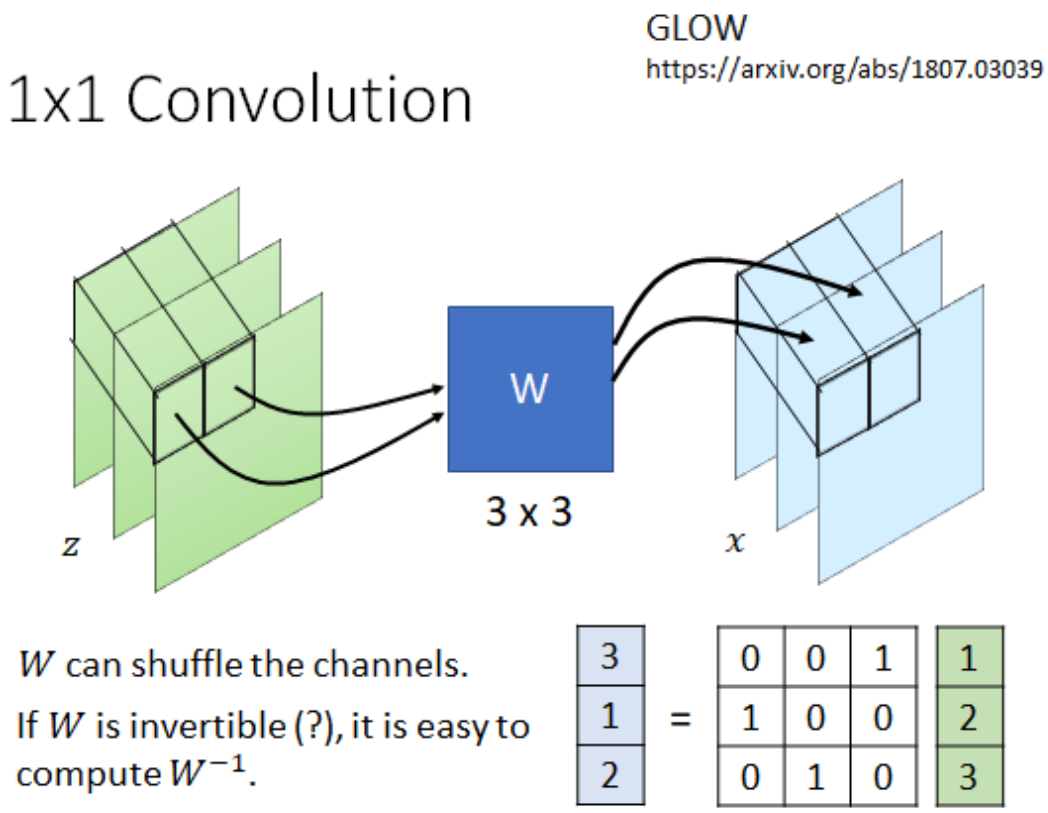
如果你是简单的把coupling layer 叠在一起，那你会发现最后生成的东西有一部分是noise，因为上面是这部分向量是直接一路copy到输出的。

所以我们做一些手脚，把其中一些coupling layer 做一下反转，如上图下侧所示。



我们再讲的更具体一点，如果说我们在做图像生成，有两种拆分向量的做法：一种是把横轴纵轴切分成多条，横轴纵轴的index之和是奇数就copy偶数就不copy做transform 。或者是，一张image 通常由rgb三个channel，你就其中某几个channel 做copy 某几个channel 做transform，每次copy 和transform的channel 可能是不一样，你有很多层coupling layer 嘛。当然，你也可以把这两种做法混在一起用，有时用第一种拆分方法，有时候用第二种拆分方法。

1x1 Convolution

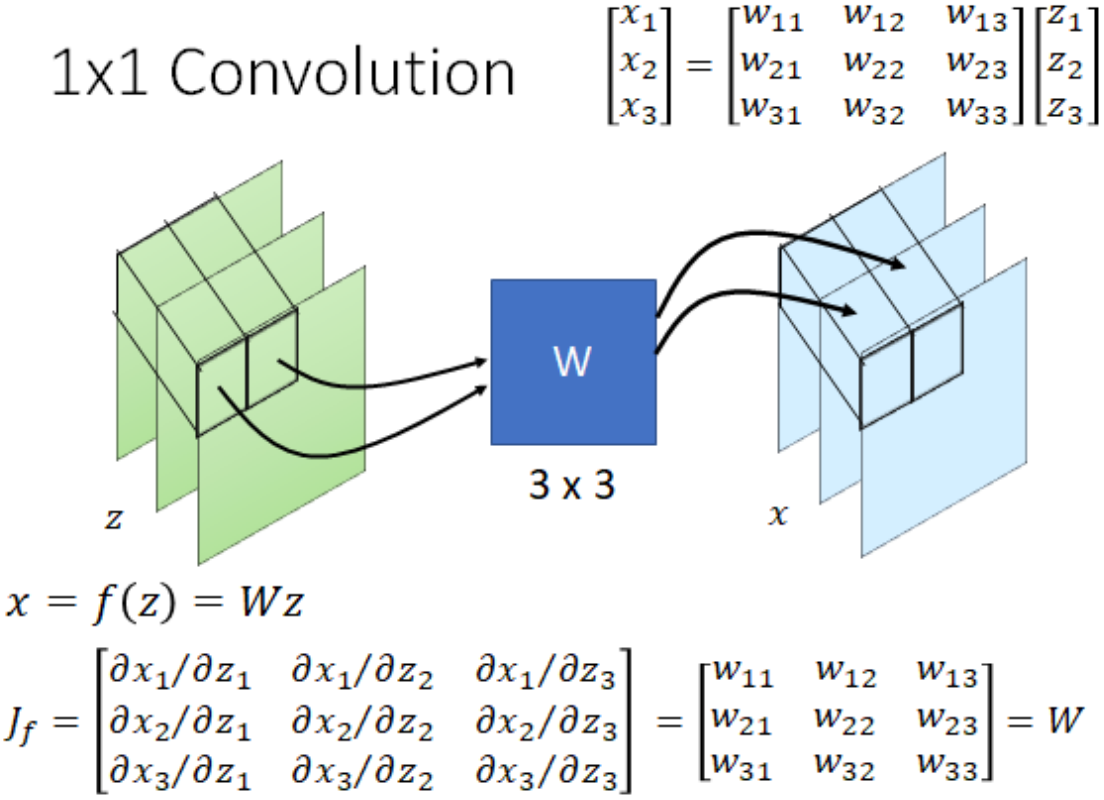


> GLOW
>
> https://arxiv.org/abs/1807.03039

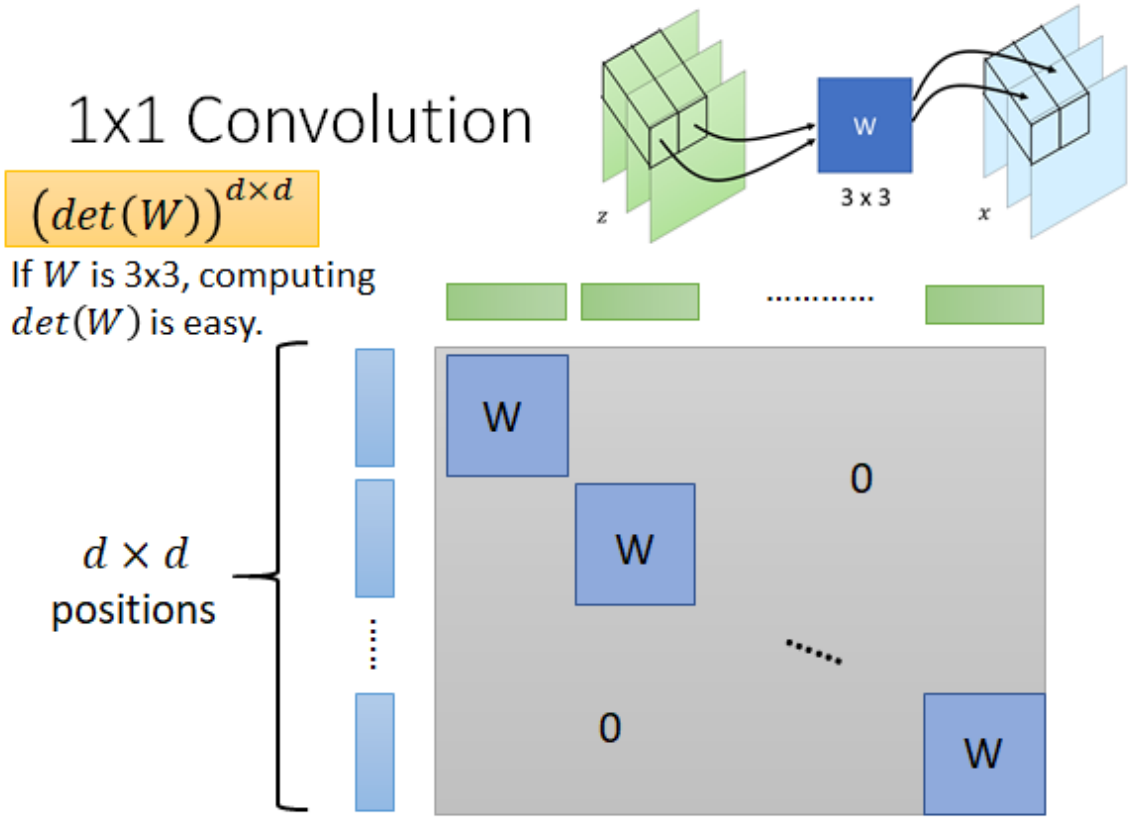
讲道理，Flow-based Generative Model 其实很早就被提出来了，上面说的coupling layer 做Generator 的方法中提到的NICE 和Real NVP 这两个Flow-based Model 都是14、15年左右被提出来了，算是很古老的东西了，为什么重提旧事呢，就是因为GLOW这个技术是去年（指2018年）被提出来了，这个技术的结果还是蛮惊人的。虽然，这个方法的结果还是没有打败GAN，但是你会惊叹"我靠，不用GAN也可以做到这种程度啊！"

GLOW除了Coupling Layer，还有个神奇的layer 叫做1x1 Convolution，举例来说我们现在要产生图片，我们就用一个3x3的W矩阵，将z一个像素一个像素地转换为x 。W是learn 出来的，它很可能能够做到的事情是shuffle channel，如上图所示的栗子，就是把z的channel 做了交换。

如果W是invertible ，那就很容易算出 W^{-1} ，但是W是learn 出来的，这样的话它一定是invertible 的吗，文献中作者在initial 的时候就用了一个invertible matrix，他也许是期待initial invertible matrix，最后的结果也能是invertible matrix。但是实际上invertible matrix 是比较容易出现的，你随机设置一个matrix 大概率就是invertible ，你想想看只有det是0才是非invertible，所以实作上也没太大问题，老师讲如果你看到有文献解释或者考虑这件事情了记得call他，也记得call我，方便我在这里做一些补充。



我们来举一个栗子，如上图所示，我们把一个像素 $[x_1, x_2, x_3]$ 乘上W，做一个 转换，求这个W的Jacobian Matrix 的公式如上图所示，你自己细算一下就会发现，J_f 就是W。



所以现在全部的z和x，做变换使用的Generator （也就是很dxd个W）的Jacobian Matrix 就是上图所示的对角矩阵，因为只有对角线上的元素对应的蓝色pixel 和绿色pixel 才是有关系的，其它地方的偏微分都是0。

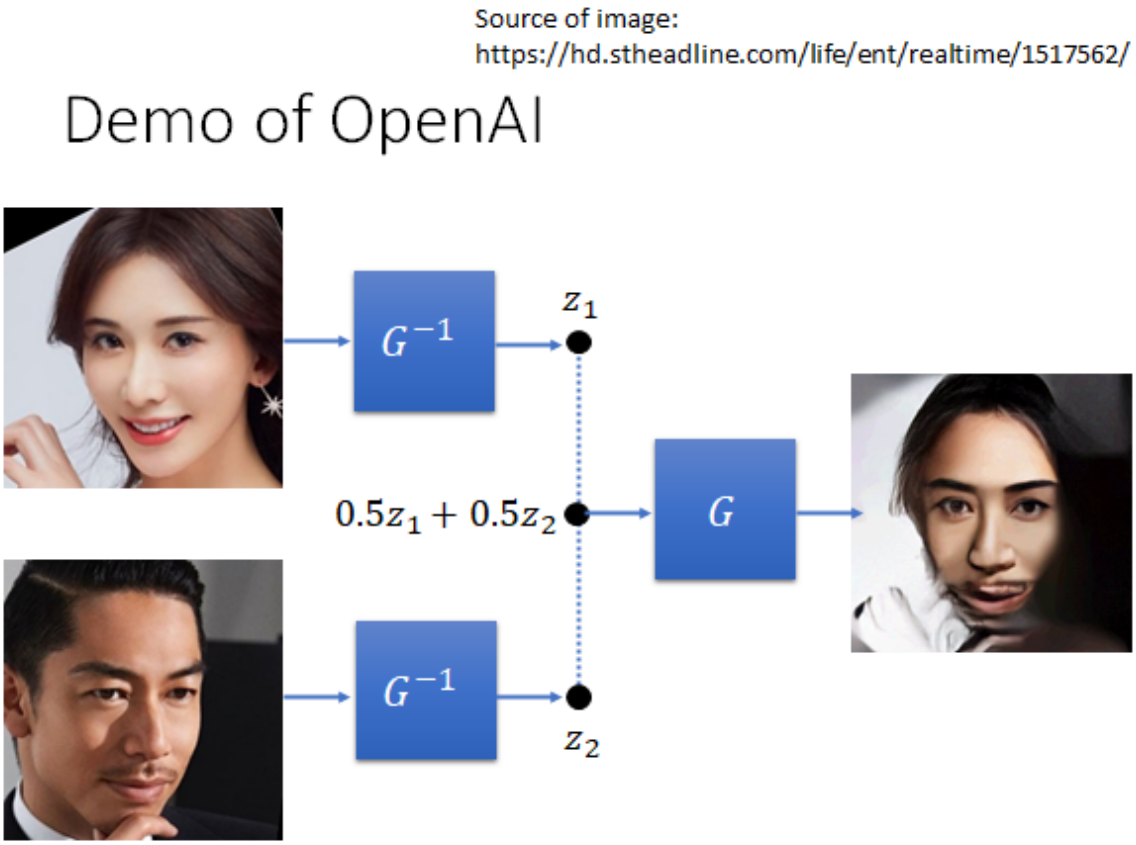
这个矩阵的det是什么呢，如果你线代够好的话，就知道这个值就是 $(det(W))^{d\times d}$ ，而W是3x3的，det很好算，所以大矩阵的det也就很好算出来了，结束。

Demo of OpenAI (GLOW)

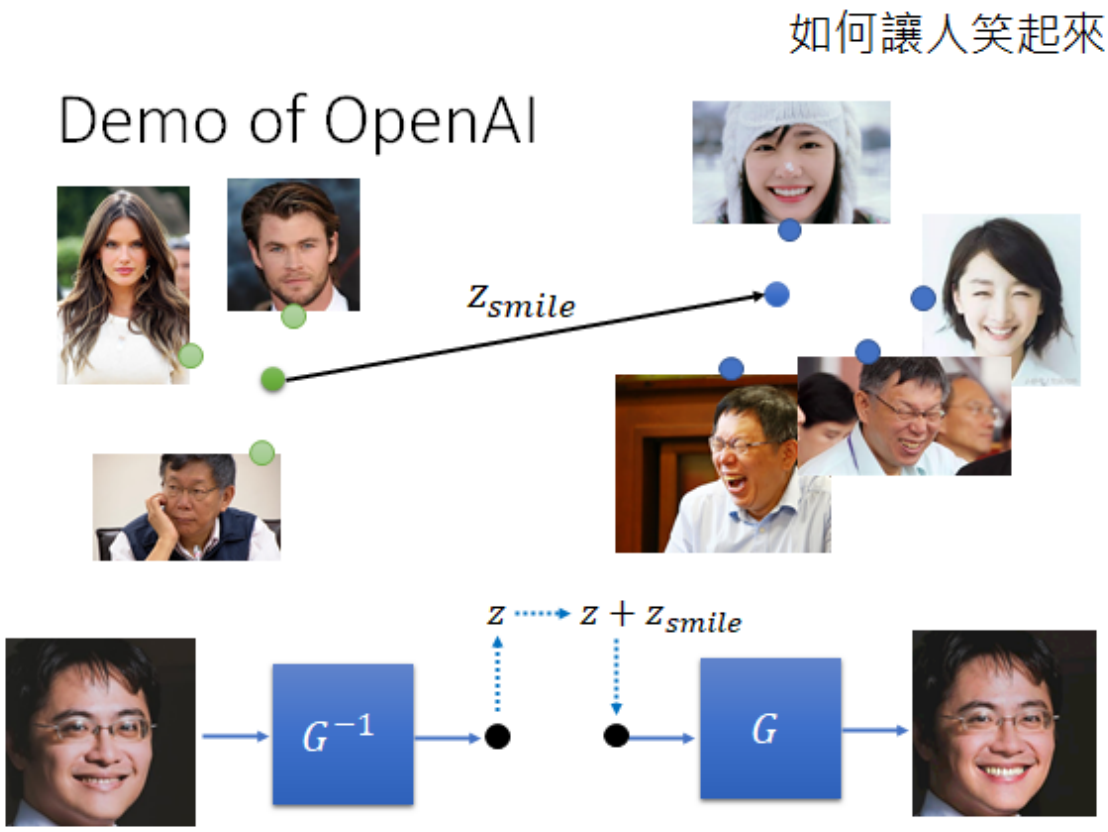
Flow-based Model 有一个很知名的结果就是GLOW这个Model，OpenAI有做一个GLOW Model demo的网站：

> https://openai.com/blog/glow/

你可以做人脸的结合：

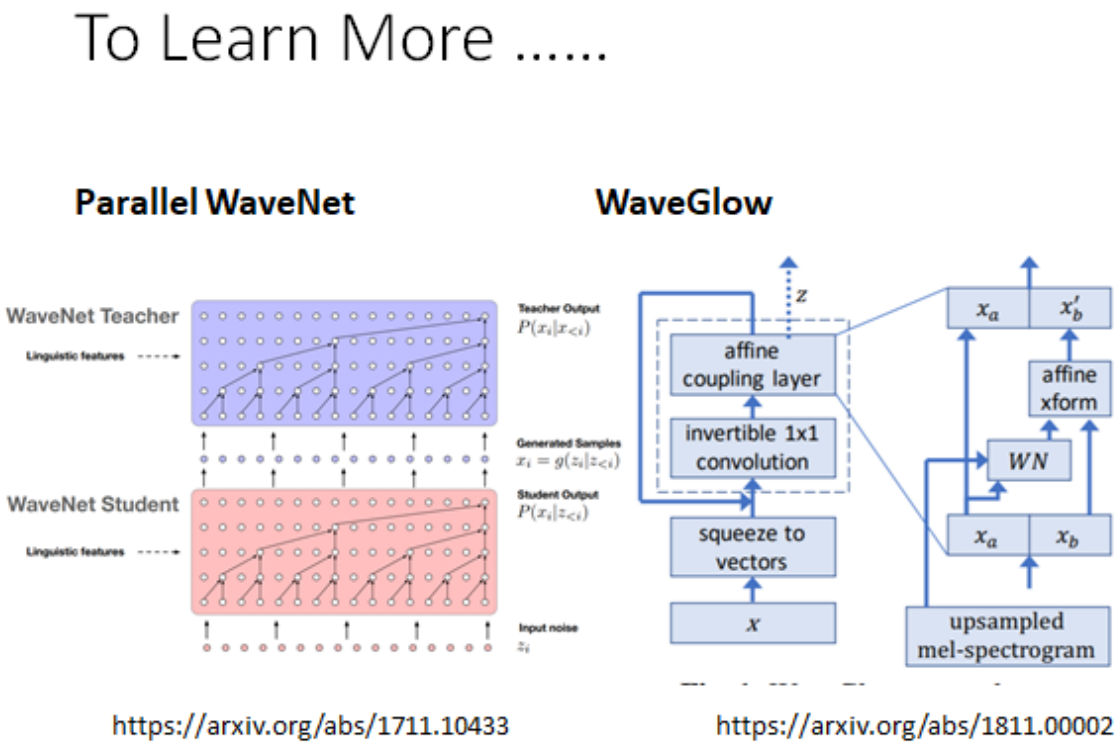


可以把脸做种种变形，比如说让人笑起来：



你需要先收集一堆笑的人脸、一堆不笑的人脸，把这两个集合的图片的z求出来分别取平均，然后相减的到 z_{smile} ， z_{smile} 就是从不笑到笑移动多少距离。然后，你只要往不笑的脸上加上一个 z_{smile} 通过G，就可以得到笑的脸了。

To Learn More ...



GLOW 现在做的最多的就是语音合成。在语音合成任务上，不知道为什么用GAN做的结果都不是很好，Auto-regressive Model 就是一个一个sample 产生出来，结果是很好，但是计算太慢不太实用，所以现在都在用GLOW ，就留给大家自学了。