GAN

- LEARNING DISTRIBUTION
- GAN MODEL
- OBJECTIVE FUNCTIONS
- APPLICATION

Generative adversarial network

GAN is a deep neural network architecture comprise of two neural network , competing against the other (i.e. adversarial)

GAN are neural network that are trained in adversarial manner to generate data mimicking some distribution

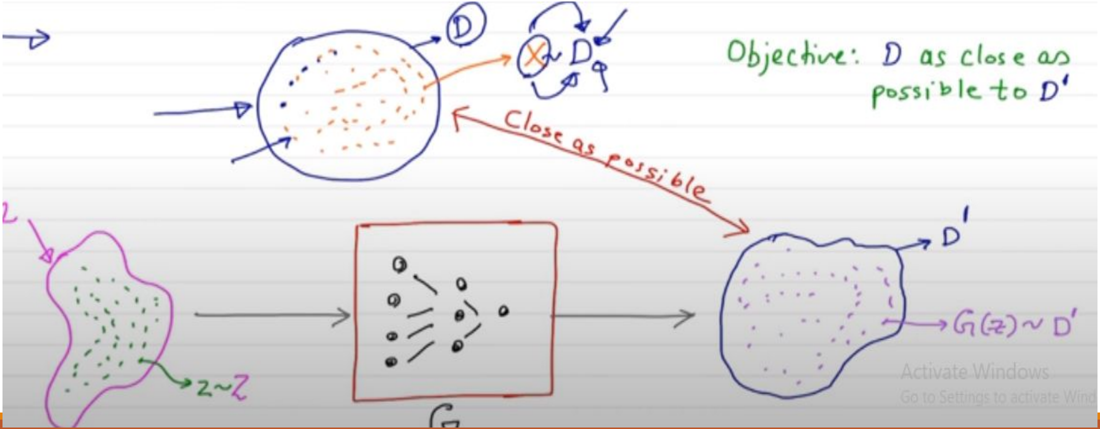
Two classes are

- Discriminator: discriminate between two different classes of data.
- Generative model : A generative model G to be trained on training data x sampled from some true distribution D is the one which given some standard random distribution

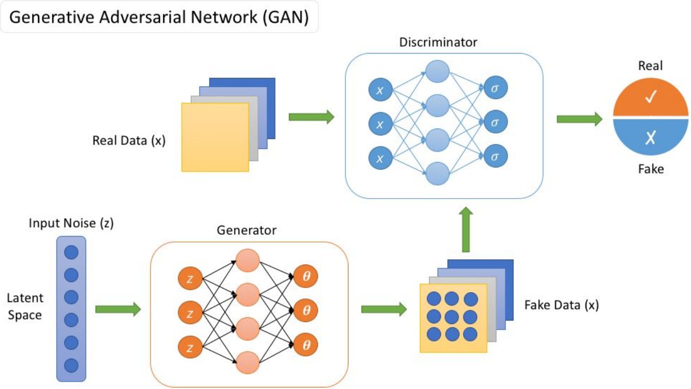
Distribution learning

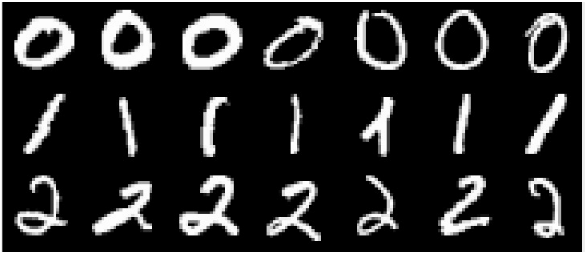
Z produces a distribution D` which is close to D according to some closeness matrix

G(z) ~ D`

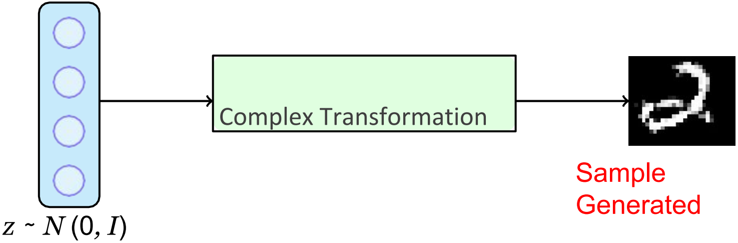


GAN MODEL


Generative Adversarial Network (GAN)
Discriminator
Real Data (x)
x
x
x
σ
σ
σ
Real
✓
X
Fake
Input Noise (z)
Latent Space
Generator
z
z
z
θ
θ
θ
Fake Data (x)

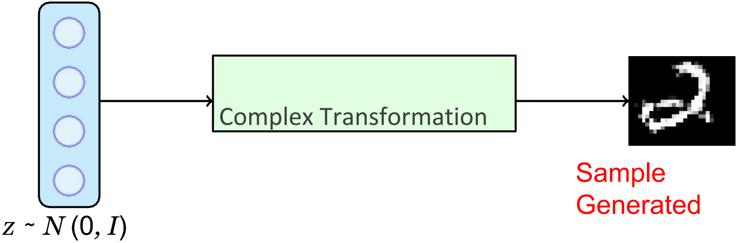


- As usual we are given some training data (say, MNIST images) which obviously comes from some underlying distribution
- Our goal is to generate more images from this distribution (*i.e.*, create images which look similar to the images from the training data)
- In other words, we want to sample from a complex high dimensional distribution which is intractable (recall RBMs, VAEs and AR models deal with this intractability in their own way)



Sample Generated

- GANs take a different approach to this problem where the idea is to sample from a simple tractable distribution (say, $z \sim N(0, I)$) and then learn a complex transformation from this to the training distribution

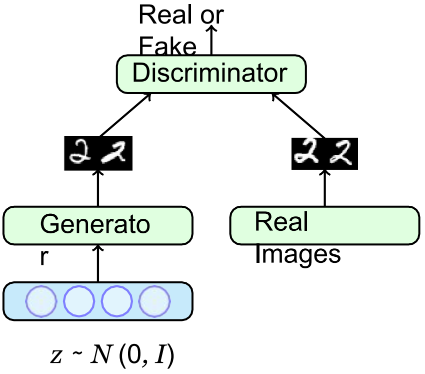


$z \sim N(0, I)$

Complex Transformation

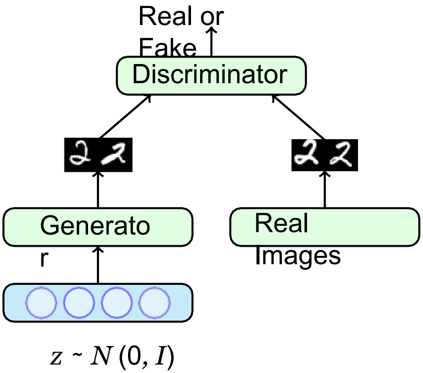
Sample Generated

- GANs take a different approach to this problem where the idea is to sample from a simple tractable distribution (say, $z \sim N(0, I)$) and then learn a complex transformation from this to the training distribution
- In other words, we will take a $z \sim N(0, I)$, learn to make a series of complex transformations on it so that the output looks as if it came from our training distribution



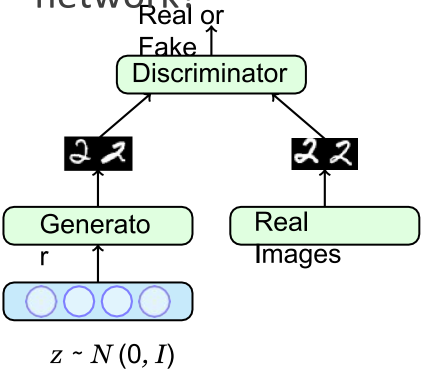
- What can we use for such a complex transformation? A Neural Network
- How do you train such a neural network? Using a two player game
- There are two players in the game: a generator and a discriminator
- The job of the generator is to produce images which look so natural that the discriminator thinks that the images came from the real data distribution
- The job of the discriminator is to get better and better at distinguishing between true images and generated (fake) images

GAN



- Let G_φ be the generator and D_θ be the discriminator (φ and θ are the parameters of G and D, respectively)
- We have a neural network based generator which takes as input a noise vector $z \sim N(0, I)$ and produces $G_\varphi(z) = X$
- We have a neural network based discriminator which could take as input a real X or a generated $X = G_\varphi(z)$ and classify the input as real/fake

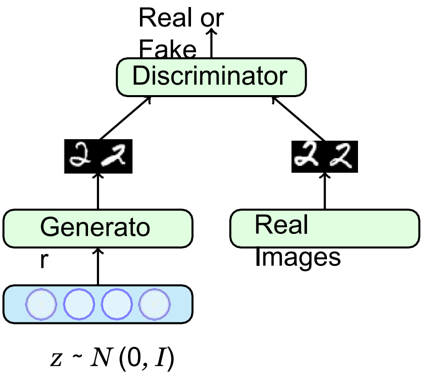
What should be the objective function of the overall network?



- Let’s look at the objective function of the generator first
- Given an image generated by the generator as $G_{\varphi}(z)$ the discriminator assigns a score $D_{\theta}(G_{\varphi}(z))$ to it
- This score will be between 0 and 1 and will tell us the probability of the image being real or fake

For a given z, the generator would want to maximize log $D_{\theta}(G_{\varphi}(z))$ (log likelihood) or minimize $\log(1 - D_{\theta}(G_{\varphi}(z)))$

Generator



- This is just for a single z and the generator would like to do this for all possible values of z,
- For example, if z was discrete and drawn from a uniform distribution (*i.e.*, $p(z) = \frac{1}{N} \forall z$) then the generator's objective function would be

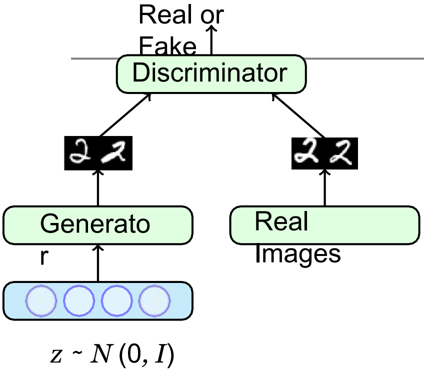
$$\min_{\varphi} \sum_{i=1}^{N} \frac{1}{N} \log(1 - D_\theta(G_\varphi(z)))$$

- However, in our case, z is continuous and not uniform ($z \sim N(0, I)$) so the equivalent objective function would be

$$\min_{\varphi} \quad p(z) \log(1 - D_\theta(G_\varphi(z)))$$

$$\min_{\varphi} E_{z \sim p(z)}[\log(1 - D_\theta(G_\varphi(z)))]$$

Discriminator


Real or Fake

Discriminator

Generator

Real Images

$z \sim N(0, I)$

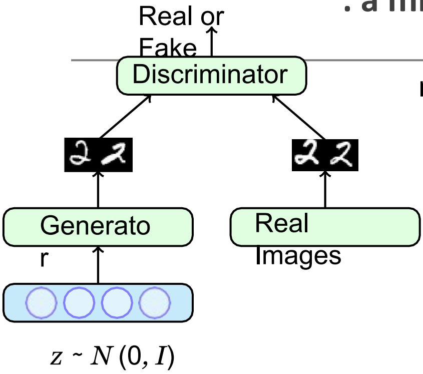
- The task of the discriminator is to assign a high score to real images and a low score to fake images

  And it should do this for all possible real images and all possible fake images

  In other words, it should try to maximize the following objective function

$$\max_{\theta} E_{x \sim p_{data}}[\log D_{\theta}(x)] + E_{z \sim p(z)}[\log(1 - D_{\theta}(G_{\varphi}(z)))]$$

objectives of the generator and discriminator : a minimax game

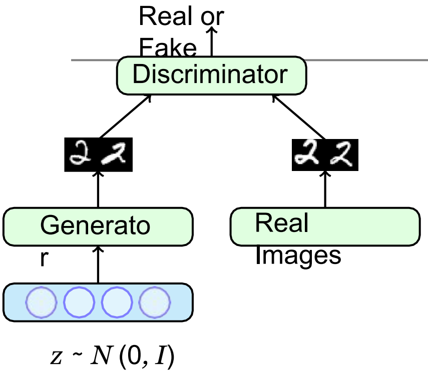


$$\min_{\varphi} \max_{\theta} \; [E_{x \sim p_{data}} \log D_\theta(x) + E_{z \sim p(z)} \log(1 - D_\theta(G_\varphi(z)))]$$

The first term in the objective is only w.r.t. the parameters of the discriminator (θ)

- The second term in the objective is w.r.t. the parameters of the generator (φ) as well as the discriminator (θ)
- The discriminator wants to maximize the second term whereas the generator wants to minimize it (hence it is a two-player game)

Training Process



Real or Fake

Discriminator

Generator

Real Images

$z \sim N(0, I)$

- **Step 1:** Gradient Ascent on Discriminator

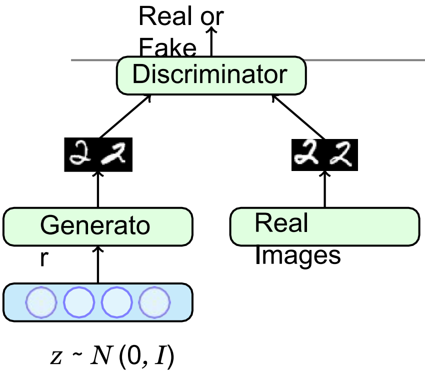
$$\max_{\theta} [\mathrm{E}_{x \sim pdata} \log D_{\theta}(x) + \mathrm{E}_{z \sim p(z)} \log(1 - D_{\theta}(G_{\varphi}(z)))]$$

- **Step 2:** Gradient Descent on Generator

$$\min_{\varphi} \mathrm{E}_{z \sim p(z)} \log(1 - D_{\theta}(G_{\varphi}(z)))$$

https://towardsdatascience.com/decoding-the-basic-math-in-gan-simplified-version-6fb6b079793

Training



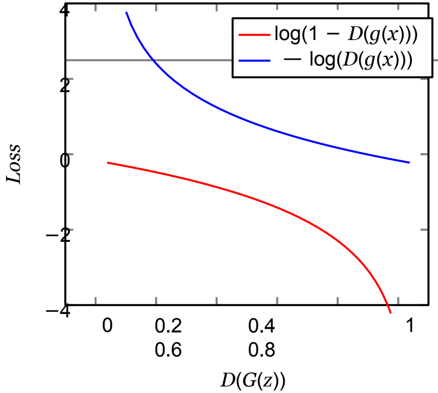
- **Step 1:** Gradient Ascent on Discriminator

$$\max_{\theta} [E_{x \sim pdata} \log D_{\theta}(x) + E_{z \sim p(z)} \log(1 - D_{\theta}(G_{\varphi}(z)))]$$

- **Step 2:** Gradient Descent on Generator

$$\min_{\varphi} E_{z \sim p(z)} \log(1 - D_{\theta}(G_{\varphi}(z)))$$

- In practice, the above generator objective does not work well and we use a slightly modified objective
- Let us see why



- When the sample is likely fake, we want to give a feedback to the generator (using gradients)
- However, in this region where $D(G(z))$ is close to 0, the curve of the loss function is very flat and the gradient would be close to 0
- Trick: Instead of minimizing the likelihood of the discriminator being correct, maximize the likelihood of the discriminator being wrong

In effect, the objective remains the same but the gradient signal becomes better

With that we are now ready to see the full algorithm for training GANs

```
procedure GAN TRAINING
end procedure
```

With that we are now ready to see the full algorithm for training GANs

```
procedure GAN Training
    for number of training iterations do
    end for
end procedure
```

With that we are now ready to see the full algorithm for training GANs

```
procedure GAN Training
    for number of training iterations do
        for k steps do
        end for
    end for
end procedure
```

With that we are now ready to see the full algorithm for training GANs

```
procedure GAN Training
    for number of training iterations
do
        for k steps do
            • Sample minibatch of $m$ noise samples $\{\mathbf{z}^{(1)}, .., \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$



        end for



    end for
end procedure
```

With that we are now ready to see the full algorithm for training GANs

```
procedure GAN Training
    for number of training iterations
do  for k steps do
        • Sample minibatch of $m$ noise samples $\{\mathbf{z}^{(1)}, .., \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$
        • Sample minibatch of $m$ examples $\{\mathbf{x}^{(1)}, .., \mathbf{x}^{(m)}\}$ from data generating distribution $p_{data}(\mathbf{x})$



    end for



    end for
end procedure
```

With that we are now ready to see the full algorithm for training GANs

```
procedure GAN Training
        for number of training iterations
do      for k steps do
                • Sample minibatch of $m$ noise samples $\{\mathbf{z}^{(1)}, .., \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$
                • Sample minibatch of $m$ examples $\{\mathbf{x}^{(1)}, .., \mathbf{x}^{(m)}\}$ from data generating distribution $p_{data}(\mathbf{x})$
                • Update the discriminator by ascending its stochastic gradient:
                        $\nabla_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D_\theta\left(x^{(i)}\right) + \log\left(1 - D_\theta\left(G_\varphi\left(z^{(i)}\right)\right)\right) \right]$
        end for
        end for
end procedure
```

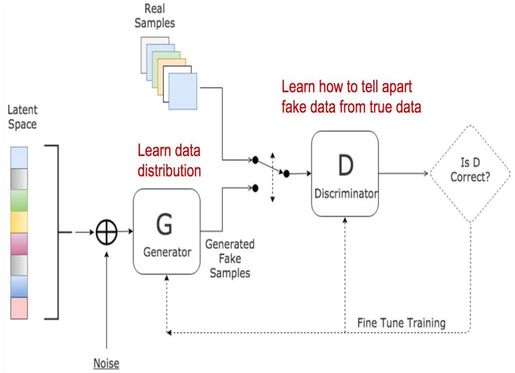
With that we are now ready to see the full algorithm for training GANs

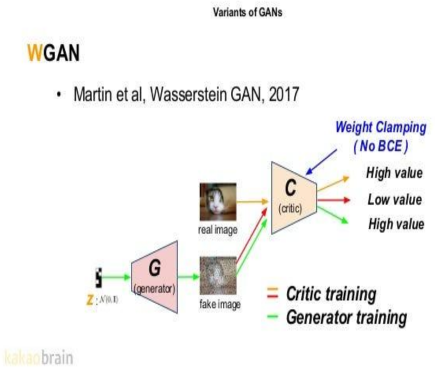
```
procedure GAN Training
    for number of training iterations
do
        for k steps do
            • Sample minibatch of $m$ noise samples $\{\mathbf{z}^{(1)}, .., \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$
            • Sample minibatch of $m$ examples $\{\mathbf{x}^{(1)}, .., \mathbf{x}^{(m)}\}$ from data generating distribution $p_{data}(\mathbf{x})$
            • Update the discriminator by ascending its stochastic gradient:

                $\nabla_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D_{\theta}\left(x^{(i)}\right) + \log\left(1 - D_{\theta}\left(G_{\varphi}\left(z^{(i)}\right)\right)\right) \right]$

        end for
        • Sample minibatch of $m$ noise samples $\{\mathbf{z}^{(1)}, .., \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$



    end for
end procedure
```

With that we are now ready to see the full algorithm for training GANs

```
procedure GAN Training
    for number of training iterations do
        for k steps do
            • Sample minibatch of $m$ noise samples $\{\mathbf{z}^{(1)}, .., \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$
            • Sample minibatch of $m$ examples $\{\mathbf{x}^{(1)}, .., \mathbf{x}^{(m)}\}$ from data generating distribution $p_{data}(\mathbf{x})$
            • Update the discriminator by ascending its stochastic gradient:
                  $\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \left[ \log D_\theta\left(x^{(i)}\right) + \log\left(1 - D_\theta\left(G_\varphi\left(z^{(i)}\right)\right)\right) \right]$
        end for
        • Sample minibatch of $m$ noise samples $\{\mathbf{z}^{(1)}, .., \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$
        • Update the generator by ascending its stochastic gradient
              $\nabla_\varphi \frac{1}{m} \sum_{i=1}^{m} \left[ \log D_\theta\left(G_\varphi\left(z^{(i)}\right)\right) \right]$
    end for
end procedure
```

GAN vs WGAN




Variants of GANs

WGAN

- Martin et al, Wasserstein GAN, 2017

WGAN Loss function

without the sigmoid function and outputs a scalar score rather than a probability. This score can be interpreted as how real the input images are.

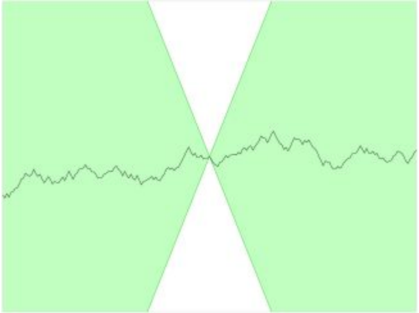
$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$$

We rename the discriminator to **critic** to reflect its new role.

Imagine the critic as a judge in a competition, deciding if something is real or fake. The Lipschitz constraint is like a rule that says the judge can't change their mind too drastically between similar things. Small difference between things (similar inputs): The judge can't make a huge jump in their decision (large gradient update). They need to adjust their score smoothly (smaller update).Large difference between things (dissimilar inputs): The judge can still change their mind significantly (update), but not too abruptly. This helps the judge learn in a more controlled way, ultimately leading to better decisions (better critic performance) and fairer competition (improved generated data).

The Lipschitz Constraint

It may surprise you that we are now allowing the critic to output any number in the range [–∞, ∞], rather than applying a sigmoid function to restrict the output to the usual [0, 1] range.



The Lipschitz Constraint

$$\frac{|D(x_1) - D(x_2)|}{|x_1 - x_2|} \leq 1$$

The critic is a function *D* that converts an image into a prediction. We say that this function is 1-Lipschitz if it satisfies the following inequality for any two input images, x_1 and x_2:

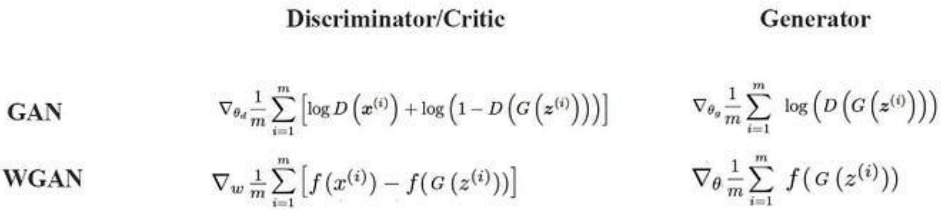
Here, $x_1 - x_2$ is the average pixel wise absolute difference between two images and D(x1)-D(x2) is the absolute difference between the critic predictions. Essentially, we require a limit on the rate at which the predictions of the critic can change between two images (i.e., the absolute value of the gradient must be at most 1 everywhere).

https://jonathan-hui.medium.com/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490

GAN vs WGAN

By training the critic function (f) to be Lipschitz continuous, the GAN aims to achieve smoother output changes and potentially improve training stability and critic performance, which can indirectly influence the overall loss function.

f has to be a 1-Lipschitz function.



	Discriminator/Critic	Generator
GAN	$\nabla_{\theta_d} \frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(x^{(i)}\right)+\log\left(1-D\left(G\left(z^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g} \frac{1}{m}\sum_{i=1}^{m} \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$
WGAN	$\nabla_w \frac{1}{m}\sum_{i=1}^{m}\left[f\left(x^{(i)}\right)-f\left(G\left(z^{(i)}\right)\right)\right]$	$\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f\left(G\left(z^{(i)}\right)\right)$

To enforce 1-Lipschitz function.

Weight Clipping

Gradient Penalty

Weight Clipping:

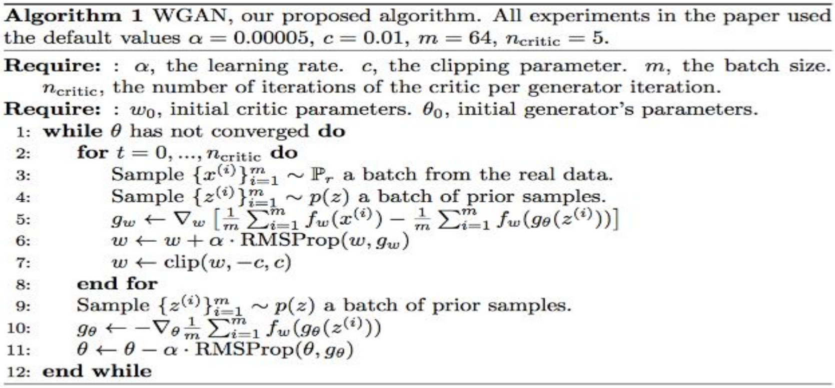
Idea: This technique limits the weights of the critic network within a specific range, typically [-c, c] for a constant c.
Mechanism: During training, after each gradient update, the weights of the critic are clipped to ensure they stay within the predefined range.
Pros: Simple to implement and computationally efficient.
Cons: Can lead to underfitting the critic, potentially hindering its ability to discriminate effectively between real and fake data. The choice of the clipping constant c can be crucial for achieving good performance.

Gradient Penalty:
Idea: Instead of directly enforcing weight constraints, the gradient penalty method penalizes the model when the Lipschitz constraint is violated.
Implementation: An additional term is added to the loss function that penalizes the norm of the gradient of the critic with respect to input samples. The penalty term encourages the model to have gradients with a norm close to 1.
Effect: By penalizing deviations from the 1-Lipschitz constraint, this method provides a more stable and effective way to enforce Lipschitz continuity.

the goal is to ensure that the norm (magnitude) of the gradient of the critic's output with respect to its inputs is close to 1. The Lipschitz continuity ensures that the critic's predictions don't change too rapidly in response to small changes in the input data.

When applying a gradient penalty, the norm of the gradient (del_inputs Critic_output||), where del represents the gradient) is penalized if it deviates from 1. The penalty term encourages the model to have gradients with a magnitude close to 1.

WGAN Algorithm with weight clipping



Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

```
Require: : $\alpha$, the learning rate. $c$, the clipping parameter. $m$, the batch size.
    $n_{\text{critic}}$, the number of iterations of the critic per generator iteration.
Require: : $w_0$, initial critic parameters. $\theta_0$, initial generator's parameters.
while $\theta$ has not converged do
    for $t = 0, ..., n_{\text{critic}}$ do
        Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
        Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
        $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
        $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
        $w \leftarrow \text{clip}(w, -c, c)$
    end for
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
    $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while
```

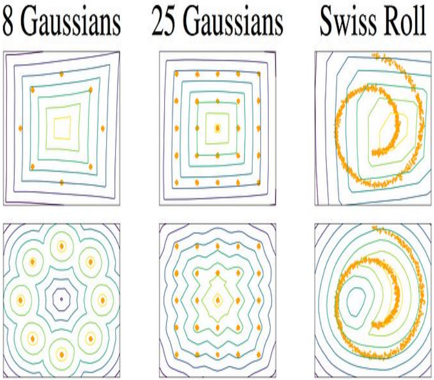
Issues

Quote from the research paper: Weight clipping is a clearly terrible way to enforce a Lipschitz constraint. If the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic till optimality. If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big, or batch normalization is not used (such as in RNNs) … and we stuck with weight clipping due to its simplicity and already good performance.

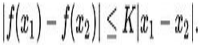
The model performance is very sensitive to this hyperparameter. In the diagram below, when batch normalization is off, the discriminator moves from diminishing gradients to exploding gradients when ***c*** *increases from 0.01 to 0.1.*

Issue with weight clipping

The weight clipping behaves as a weight regulation. It reduces the capacity of the model ***f*** and limits the capability to model complex functions.



To enforce Lipschitz constraint.

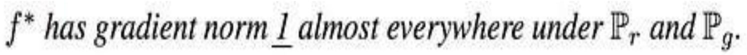


$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|.$$

WGAN-GP uses gradient penalty instead of the weight clipping to enforce the Lipschitz constraint.

Gradient penalty

A differentiable function ***f*** is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere.



f^* has gradient norm $\underline{1}$ almost everywhere under $\mathbb{P}_r$ and $\mathbb{P}_g$.

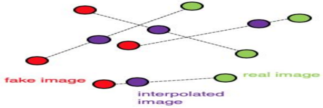
Loss function of WGAN

$$L = \underbrace{\underset{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}{\mathbb{E}}[D(\tilde{\boldsymbol{x}})] - \underset{\boldsymbol{x}\sim\mathbb{P}_r}{\mathbb{E}}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \underset{\hat{\boldsymbol{x}}\sim\mathbb{P}_{\hat{\boldsymbol{x}}}}{\mathbb{E}}\left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2\right]}_{\text{Our gradient penalty}}.$$

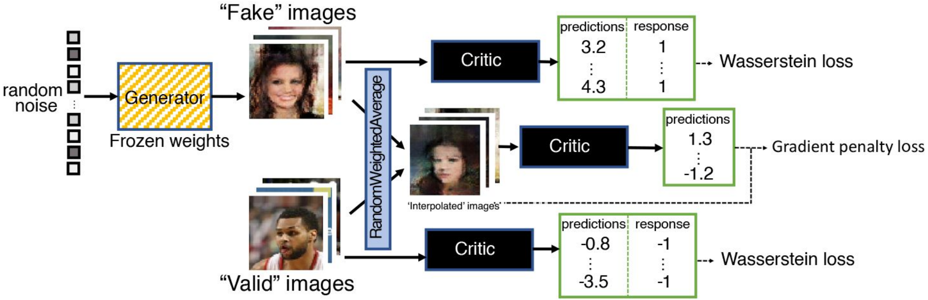
where $\hat{\boldsymbol{x}}$ *sampled from* $\tilde{\boldsymbol{x}}$ *and* $\boldsymbol{x}$ *with t uniformly sampled between 0 and 1*

$$\hat{\boldsymbol{x}} = t\,\tilde{\boldsymbol{x}} + (1-t)\,\boldsymbol{x} \text{ with } 0 \le t \le 1$$

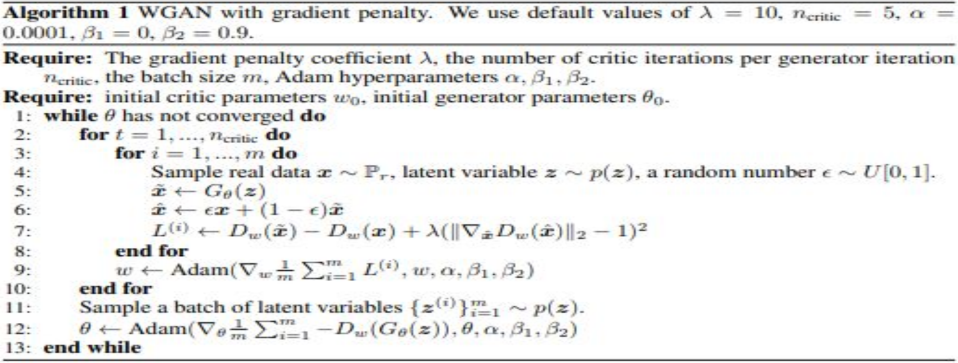
λ is set to 10. The point *x* used to calculate the gradient norm is any points sampled between the *Pg* and *Pr*.



Gradient Penalty Loss



WGAN : Gradient Penalty



Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.

Require: initial critic parameters w_0, initial generator parameters θ_0.

```
while $\theta$ has not converged do
    for $t = 1, ..., n_{\text{critic}}$ do
        for $i = 1, ..., m$ do
            Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
            $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
            $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
            $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
        end for
        $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
    end for
    Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
    $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while
```

Module 23.2: Generative Adversarial Networks - Architecture

- We will now look at one of the popular neural networks used for the generator and discriminator (Deep Convolutional GANs)

- We will now look at one of the popular neural networks used for the generator and discriminator (Deep Convolutional GANs)
- For discriminator, any CNN based classifier with 1 class (real) at the output can be used (e.g. VGG, ResNet, etc.)

- We will now look at one of the popular neural networks used for the generator and discriminator (Deep Convolutional GANs)
- For discriminator, any CNN based classifier with 1 class (real) at the output can be used (e.g. VGG, ResNet, etc.)

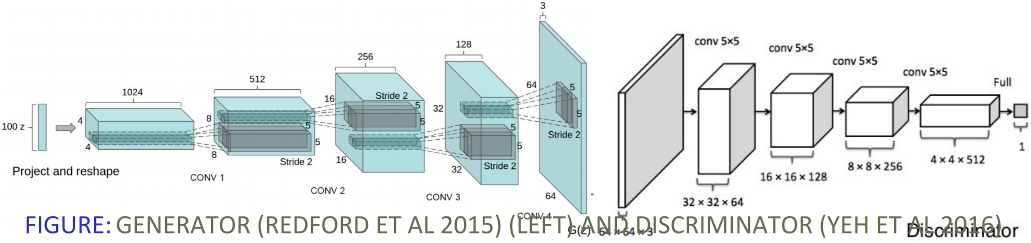


FIGURE: GENERATOR (REDFORD ET AL 2015) (LEFT) AND DISCRIMINATOR (YEH ET AL 2016) (RIGHT) USED IN DCGAN

Replacing Pooling with Strided Convolutions:

Traditionally, many convolutional neural networks (CNNs) used pooling layers like max pooling to downsample the input data, reducing its spatial resolution.
This approach discards information and relies on pre-defined rules for downsampling.
All Convolutional Net (ACN) Approach:

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).

The authors propose using strided convolutions instead of pooling layers.
Strided convolutions downsample the data by applying the convolution filter with a larger stride (e.g., stride of 2).
This allows the network to learn its own downsampling strategy through the weights and biases of the convolutional layers.
Benefits in GANs:

Generator: By using strided convolutions in the generator, the network can learn its own upsampling strategy to gradually increase the resolution of the generated data. This flexibility can potentially lead to more detailed and realistic outputs.
Discriminator: Similarly, using strided convolutions in the discriminator allows it to learn adaptive downsampling for analyzing data at different resolutions. This can potentially improve its ability to discriminate between real and fake data at various levels of detail.
Overall, replacing pooling layers with strided convolutions in this GAN architecture provides the network with more flexibility and control over its downsampling and upsampling processes, potentially leading to improved performance in both the generator and discriminator.

Here's an analogy:

Imagine you are analyzing images to identify objects. Traditionally, you might use a fixed grid to identify regions of interest.
The ACN approach is like letting you freely move the grid to focus on different areas of the image based on the information you are looking for. This can potentially lead to a more nuanced and effective analysis.
It's important to note that while the ACN approach offers potential benefits, it also introduces additional parameters and computational complexity compared to using pooling layers. The choice of architecture depends on various factors specific to the task and dataset at hand.

Batch Normalization which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance. This helps deal with training problems that arise due to poor initialization and helps gradient flow in deeper models. This proved critical to get deep generators to begin learning, preventing the generator from collapsing all samples to a single point which is a common failure mode observed in GANs. Directly applying batchnorm to all layers however, resulted in sample oscillation and model instability. This was avoided by not applying batchnorm to the generator output layer and the discriminator input layer

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.

Benefits of BatchNorm:

Stabilizes learning: BatchNorm normalizes the activations of each layer to have zero mean and unit variance. This helps alleviate issues arising from:
Poor initialization: Normalization reduces the sensitivity to the initial weights, making the network less prone to getting stuck in bad local minima.
Internal covariate shift: The distribution of activations can change throughout the network during training. BatchNorm helps address this by maintaining consistent activation statistics.

Improves gradient flow: By normalizing activations, BatchNorm prevents them from becoming too large or too small, which can hinder gradient propagation in deep networks (vanishing/exploding gradients).

Prevents generator collapse: In GANs, BatchNorm can prevent the generator from collapsing all outputs to a single point, a common failure mode.

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.

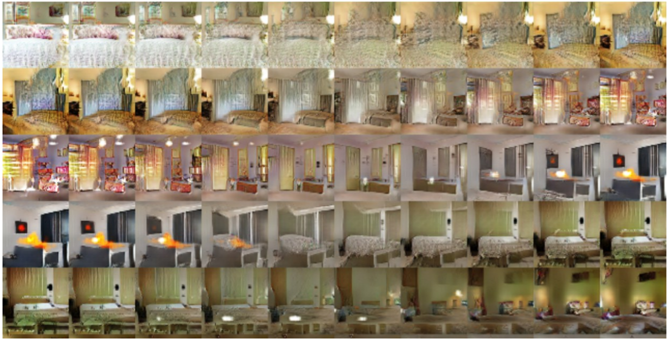
Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses tanh.

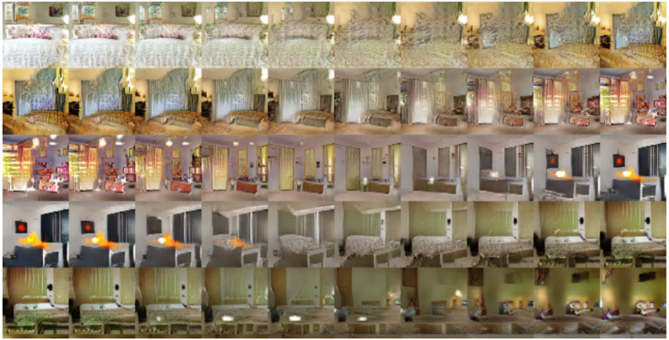
Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses tanh.
- Use LeakyReLU activation in the discriminator for all layers

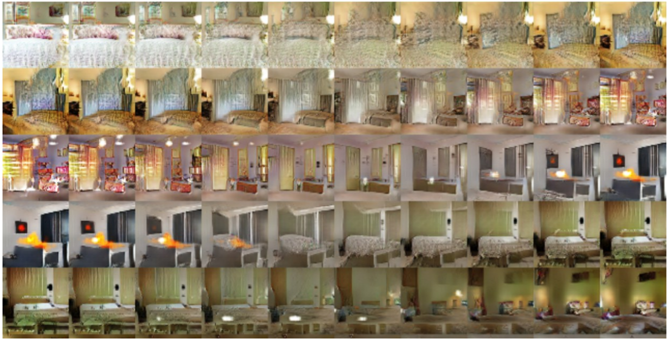
Module 23.4: Generative Adversarial Networks - Some Cool Stuff and Applications



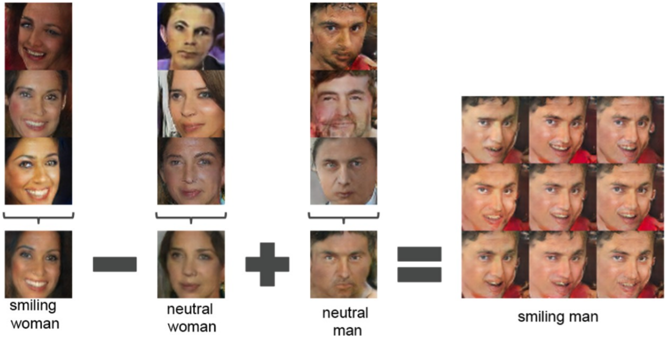
- In each row the first image was generated by the network by taking a vector z_1 as the input and the last images was generated by a vector z_2 as the input



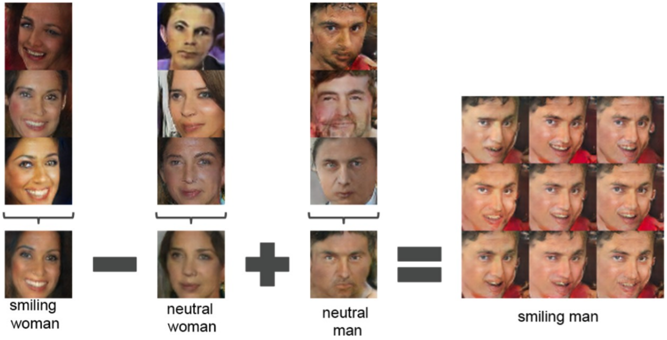
- In each row the first image was generated by the network by taking a vector z_1 as the input and the last images was generated by a vector z_2 as the input
- All intermediate images were generated by feeding z's which were obtained by interpolating z_1 and z_2 ($z = \lambda z_1 + (1 — \lambda)z_2$)



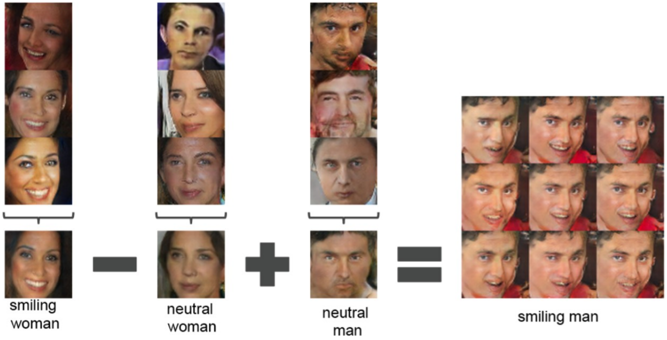
- In each row the first image was generated by the network by taking a vector z_1 as the input and the last images was generated by a vector z_2 as the input
- All intermediate images were generated by feeding z's which were obtained by interpolating z_1 and z_2 ($z = \lambda z_1 + (1 — \lambda)z_2$)
- As we transition from z_1 to z_2 in the input space there is a corresponding smooth transition in the image space also



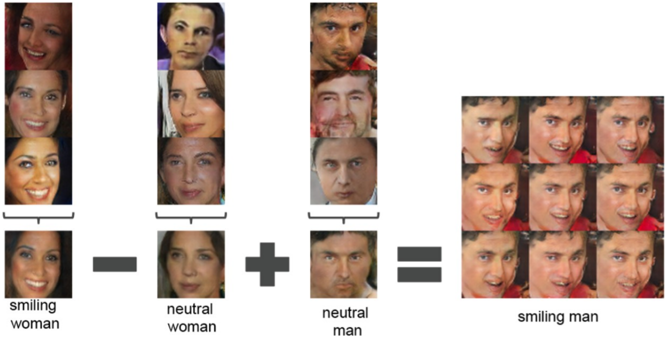
- The first 3 images in the first column were generated by feeding some
  z_{11}, z_{12}, z_{13} respectively as the input to the generator



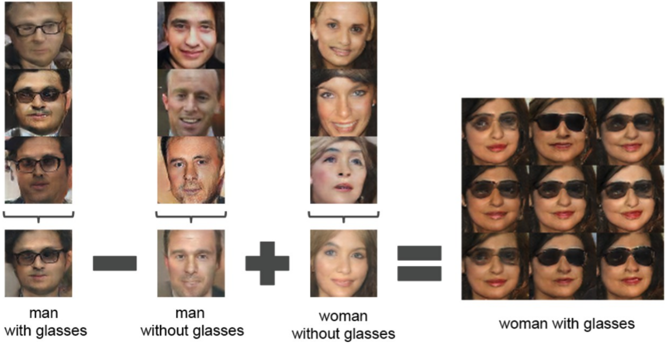
- The first 3 images in the first column were generated by feeding some z_{11}, z_{12}, z_{13} respectively as the input to the generator
- The fourth image was generated by taking an average of $z_1 = z_{11}, z_{12}, z_{13}$ and feeding it to the generator



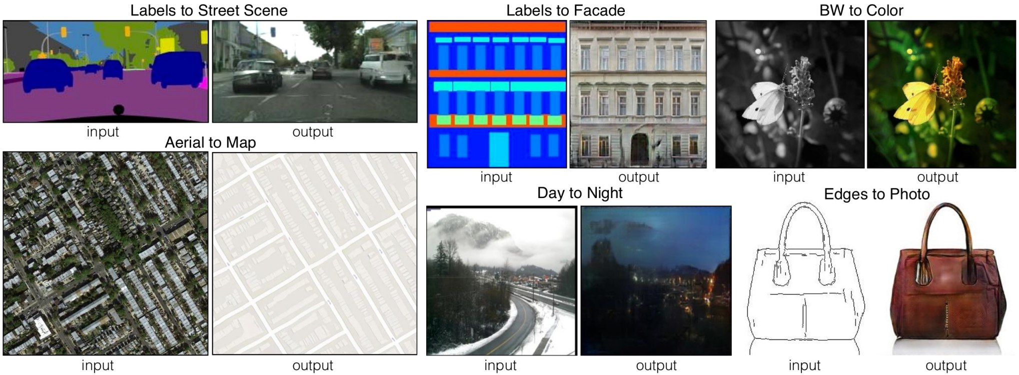
- The first 3 images in the first column were generated by feeding some z_{11}, z_{12}, z_{13} respectively as the input to the generator
- The fourth image was generated by taking an average of $z_1 = z_{11}, z_{12}, z_{13}$ and feeding it to the generator
- Similarly we obtain the average vectors z_2 and z_3 for the 2nd and 3rd columns



- The first 3 images in the first column were generated by feeding some z_{11}, z_{12}, z_{13} respectively as the input to the generator
- The fourth image was generated by taking an average of $z_1 = z_{11}, z_{12}, z_{13}$ and feeding it to the generator
- Similarly we obtain the average vectors z_2 and z_3 for the 2nd and 3rd columns
- If we do a simple vector arithmetic on these averaged vectors then we see the corresponding effect in the generated



- The first 3 images in the first column were generated by feeding some z_{11}, z_{12}, z_{13} respectively as the input to the generator
- The fourth image was generated by taking an average of $z_1 = z_{11}, z_{12}, z_{13}$ and feeding it to the generator
- Similarly we obtain the average vectors z_2 and z_3 for the 2nd and 3rd columns
- If we do a simple vector arithmetic on these averaged vectors then we see the corresponding effect in the generated images



Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros, Image-to-Image Translation with Conditional Adversarial Networks, CVPR, 2017.

Module 23.5: Bringing it all together (the deep generative summary)

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No

Table: Comparison of Generative Models

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes

Table: Comparison of Generative Models

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes
Compute P(X)	Intractable	Intractable	Tractable	No

Table: Comparison of Generative Models

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes
Compute P(X)	Intractable	Intractable	Tractable	No
Sampling	MCMC	Fast	Slow	Fast

Table: Comparison of Generative Models

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes
Compute P(X)	Intractable	Intractable	Tractable	No
Sampling	MCMC	Fast	Slow	Fast
Type of GM	Undirected	Directed	Directed	Directed

Table: Comparison of Generative Models

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes
Compute P(X)	Intractable	Intractable	Tractable	No
Sampling	MCMC	Fast	Slow	Fast
Type of GM	Undirected	Directed	Directed	Directed
Loss	KL-divergence	KL-divergence	KL-divergence	JSD

Table: Comparison of Generative Models

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes
Compute P(X)	Intractable	Intractable	Tractable	No
Sampling	MCMC	Fast	Slow	Fast
Type of GM	Undirected	Directed	Directed	Directed
Loss	KL-divergence	KL-divergence	KL-divergence	JSD
Assumptions	X independent given z	X independent given z	None	N.A.

Table: Comparison of Generative Models

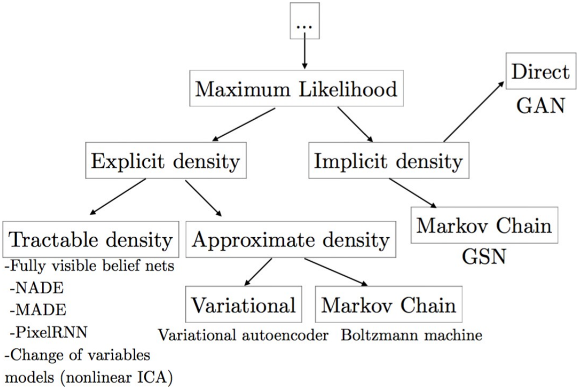
	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes
Compute P(X)	Intractable	Intractable	Tractable	No
Sampling	MCMC	Fast	Slow	Fast
Type of GM	Undirected	Directed	Directed	Directed
Loss	KL-divergence	KL-divergence	KL-divergence	JSD
Assumptions	X independent given z	X independent given z	None	N.A.
Samples	Bad	Ok	Good	Good (best)

Table: Comparison of Generative Models

	RBMs	VAEs	AR models	GANs
Abstraction	Yes	Yes	No	No
Generation	Yes	Yes	Yes	Yes
Compute P(X)	Intractable	Intractable	Tractable	No
Sampling	MCMC	Fast	Slow	Fast
Type of GM	Undirected	Directed	Directed	Directed
Loss	KL-divergence	KL-divergence	KL-divergence	JSD
Assumptions	X independent given z	X independent given z	None	N.A.
Samples	Bad	Ok	Good	Good (best)

Table: Comparison of Generative Models

Recent works look at combining these methods: e.g. Adversarial Autoencoders (Makhzani 2015), PixelVAE (Gulrajani 2016) and PixelGAN Autoencoders (Makhzani 2017)



...

Maximum Likelihood

Explicit density

Implicit density

Direct

GAN

Tractable density

-Fully visible belief nets

-NADE

-MADE

-PixelRNN

-Change of variables models (nonlinear ICA)

Approximate density

Markov Chain

GSN

Variational

Variational autoencoder

Markov Chain

Boltzmann machine

Source: Ian Goodfellow, NIPS 2016 Tutorial: Generative Adversarial Networks