Neural Networks and Deep Learning Lecture 12

Wei WANG

cs5242@comp.nus.edu.sg

NUS National University of Singapore

School of Computing

Administrative

- Debriefing and verification session for quiz and assignment 2
  - 14:00-17:00 at COM1-0206. 14 April.
- Project report and code
  - One submission per group
  - With workload assignment included if there are 2 members in one group
- Final presentation
  - Optional

Recap

Attention modelling for Seq2seq

- Different attention weight calculation approaches

$e_{11} = a(s_0, h_1) = v^T \tanh(W_a s_0 + U_a h_1)$
$e_{12} = a(s_0, h_2) = v^T \tanh(W_a s_0 + U_a h_2)$
$e_{13} = a(s_0, h_3) = v^T \tanh(W_a s_0 + U_a h_3)$

$e_{11} = a(s_0, h_1) = v_1^T [s_0; y_0]$
$e_{12} = a(s_0, h_2) = v_2^T [s_0; y_0]$
$e_{13} = a(s_0, h_3) = v_3^T [s_0; y_0]$

$\alpha_{11} = \exp(e_{11}) / (\exp(e_{11})+\exp(e_{12})+\exp(e_{13}))$
$\alpha_{12} = \exp(e_{12}) / (\exp(e_{11})+\exp(e_{12})+\exp(e_{13}))$
$\alpha_{13} = \exp(e_{13}) / (\exp(e_{11})+\exp(e_{12})+\exp(e_{13}))$
$c_1 = \alpha_{11} h_1 + \alpha_{12} h_2 + \alpha_{13} h_3$

Neural Style Transfer

- Content loss
  - Lcontent = $||f(x) – f(c)||^2$
  - f() is the conv feature from some Conv layer of a pre-trained ConvNet
  - x is the images to be generated; c is the given/reference image
- Style loss
  - Lstyel = $||G(x) – G(s)||^2$
  - G(x) is the Gramian matrix computed as $G(x)=dot(f(x), f(x)^T)$
  - f(x) is the reshaped feature of some Conv layer of a pre-trained ConvNet
    - (num of channels, height x width)
- Total loss
  - Combine content loss and style loss (from multiple conv layers)

Generative Adversarial Network (GAN)

Intended learning outcome

Compare	Compare discriminative model and generative model
Understand	Understand GAN
Know	Know the difficulties of training GAN and applications of GAN

Machine learning model category

- With label or not
  - With label: Supervised learning
    - logistic regression, MLP, CNN, RNN for sentiment analysis/machine translation
  - Without label: Unsupervised learning
    - K-means, auto-encoder, RNN for language modelling, dimensionality reduction
- Probability modelling
  - Conditional probability: discriminative models
    - P(y | x) for classification (x is the sample feature, y is the sample label)
  - Joint probability: generative models
    - P(x, y) for generating samples

Discriminative VS generative models

- P(y|x)
  - For effective classification
- P(x, y)
  - Models the data distribution
  - Likelihood of a sample (pair) from the given distribution
    - P("Cat", ) P("Dot", )
    - P(Cat, 猫) P(Cat, 狗)

Example

Training data: (1,0), (1,0), (2,0), (2, 1)

`p(x,y)` is

	y=0	y=1
x=1	1/2	0
x=2	1/4	1/4

Bayes Rule $\longrightarrow$

`p(y|x)` is

	y=0	y=1
x=1	1	0
x=2	1/2	1/2

https://stackoverflow.com/questions/879432/what-is-the-difference-between-a-generative-and-discriminative-algorithm

"What I cannot create, I do not understand."

—Richard Feynman

More applications of generative models

- P(x)
  - E.g. if P(x) is a Gaussian distribution, then we can sample a set of points that follow Gaussian distribution
  - If P(x) models the image pixel distribution, then we can sample/generate images
- Face aging
- Anime face drawing
- Interactive image generation
- Domain/style transfer
- Deep photo style transfer

Approaches

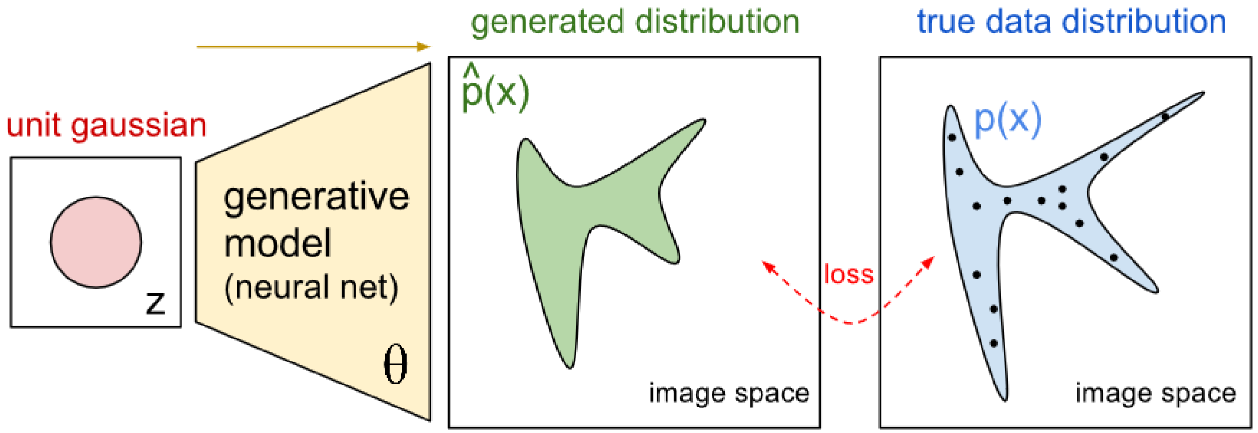
- Maximum likelihood over the training data
  - Explicit density model
  - Implicit density model
    - Variational Autoencoders (VAEs)
    - Autoregressive models, PixelRNN
    - Generative adversarial network (GAN)

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{x \sim p_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{\theta})$$

Ian Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. https://arxiv.org/abs/1701.00160

Generative adversarial network (GAN)

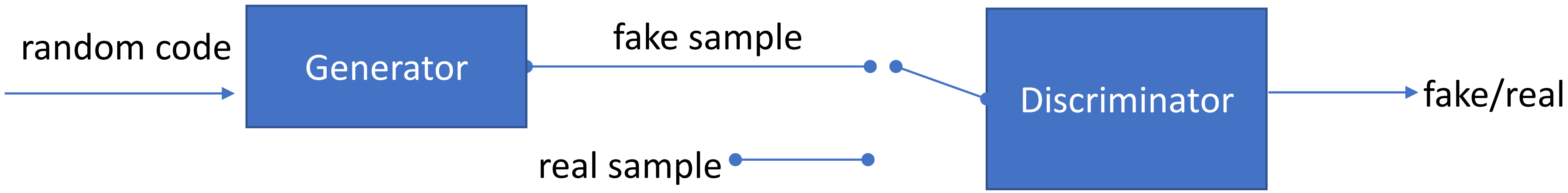
- Generate samples from the data
  - P_{data} is unknown → train a model to approximate P_{data}, call it P_{model}
  - Objective: make P_{model} similar as P_{data} ?



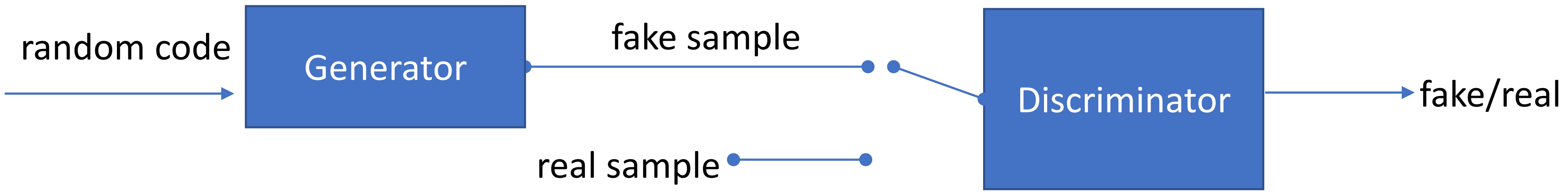
https://blog.openai.com/generative-models/

Generative adversarial network (GAN)

- Idea
  - Make the generated samples similar to samples from the training data
  - Similar?
    - Measure the difference of the generated samples with training samples?
    - Train a discriminator to distinguish training samples from generated samples
      - If the generated samples can fool the discriminator → good generator

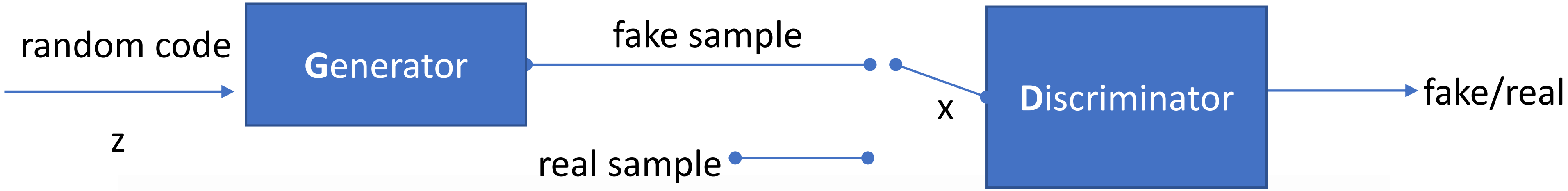


GAN



- Training procedure
  - Train the generator to fool the discriminator
  - Trian the discriminator to distinguish fake/real samples
  - Repeat the above two steps until converge
    - The generated samples too similar to real samples that the discriminator fails to work

GAN Math



Symbol	Meaning	Notes
p_z	Data distribution over noise input z	Usually, just uniform.
p_g	The generator's distribution over data x	
p_r	Data distribution over real sample x	

$$
\begin{aligned}
\min_G \max_D L(D,G) &= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))] \\
&= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1-D(x))]
\end{aligned}
$$

Fix D and train G

- Generator is implemented using a network (MLP or CNN)
  - Uniform input code z (different z increases the diversity of the output)
  - CNN can map small input into big output via deconvolution
    - Upsampling or transposed convolution
- To fool the discriminator == high probability of D(G(z))
  - Randomly sample z and generate G(z)
  - Train G's parameters to Minimize $_G$ $\mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$

Fix G and train D

- Discriminator is implemented using another network
  - The final layer uses sigmoid to generator D(x), i.e. probability for x being real
- For real samples, maximize the probability D(x), $x \sim P_r(x)$
- For fake samples, minimize the probability D(x), $x \sim P_g(x)$
  - == maximize the probability 1-D(x)
- The combined objective is to train D's parameters to maximize

$$\mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1 - D(x))]$$

Optimal solution

- Given x, find the optimal D that maximize the objective

$$\tilde{x} = D(x), A = p_r(x), B = p_g(x)$$

$$\begin{aligned} f(\tilde{x}) &= A log\tilde{x} + B log(1-\tilde{x}) \\ \frac{df(\tilde{x})}{d\tilde{x}} &= A\frac{1}{ln10}\frac{1}{\tilde{x}} - B\frac{1}{ln10}\frac{1}{1-\tilde{x}} \\ &= \frac{1}{ln10}\left(\frac{A}{\tilde{x}} - \frac{B}{1-\tilde{x}}\right) \\ &= \frac{1}{ln10}\frac{A-(A+B)\tilde{x}}{\tilde{x}(1-\tilde{x})} \end{aligned}$$

Thus, set $\frac{df(\tilde{x})}{d\tilde{x}} = 0$, we get the best value of the discriminator:

$D^*(x) = \tilde{x}^* = \frac{A}{A+B} = \frac{p_r(x)}{p_r(x)+p_g(x)} \in [0,1]$. When $p_g = p_r$, $D^*(x)$ = 1/2

Equations from: https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html

Implementation

- Generator
- Discriminator
- Alternative training
- The following code is from https://github.com/wayaai/GAN-Sandbox

Generator

```
rand_dim = 64  # dimension of generator's input tensor (gaussian noise)
# image dimensions
img_height = 28
img_width = 28
img_channels = 3
def generator_network(x):
    def add_common_layers(y):
        y = layers.advanced_activations.LeakyReLU()(y)
        y = layers.Dropout(0.25)(y)
        return y
    x = layers.Dense(1024)(x)
    x = add_common_layers(x)
    # input dimensions to the first de-conv layer in the generator
    height_dim = 7
    width_dim = 7
    x = layers.Dense(height_dim * width_dim * 128)(x)
    x = add_common_layers(x)
    x = layers.Reshape((height_dim, width_dim, -1))(x)
    x = layers.Conv2DTranspose(64, kernel_size, **conv_layer_keyword_args)(x)
    x = add_common_layers(x)
    # number of feature maps => number of image channels
    return layers.Conv2DTranspose(img_channels, 1, strides=2, padding='same', activation='tanh')(x)
```

Discriminator

```
def discriminator_network(x):
    def add_common_layers(y):
        y = layers.advanced_activations.LeakyReLU()(y)
        y = layers.Dropout(0.25)(y)
        return y
    x = layers.GaussianNoise(stddev=0.2)(x)
    x = layers.Conv2D(64, kernel_size, **conv_layer_keyword_args)(x)
    x = add_common_layers(x)
    x = layers.Conv2D(128, kernel_size, **conv_layer_keyword_args)(x)
    x = add_common_layers(x)
    x = layers.Flatten()(x)
    x = layers.Dense(1024)(x)
    x = add_common_layers(x)
    return layers.Dense(1, activation='sigmoid')(x)
```

Combine generator and discriminator

```
def adversarial_training(data_dir, generator_model_path, discriminator_model_path):
    generator_input_tensor = layers.Input(shape=(rand_dim, ))
    generated_image_tensor = generator_network(generator_input_tensor)

    generated_or_real_image_tensor = layers.Input(shape=(img_height, img_width, img_channels))
    discriminator_output = discriminator_network(generated_or_real_image_tensor)

    generator_model = models.Model(inputs=[generator_input_tensor],
                                   outputs=[generated_image_tensor],
                                   name='generator')
    discriminator_model = models.Model(inputs=[generated_or_real_image_tensor],
                                       outputs=[discriminator_output],
                                       name='discriminator')
    combined_output = discriminator_model(generator_model(generator_input_tensor))
    combined_model = models.Model(inputs=[generator_input_tensor],
                                  outputs=[combined_output], name='combined')

    adam = optimizers.Adam(lr=0.0002, beta_1=0.5, beta_2=0.999)
    generator_model.compile(optimizer=adam, loss='binary_crossentropy')
    discriminator_model.compile(optimizer=adam, loss='binary_crossentropy')
    discriminator_model.trainable = False
    combined_model.compile(optimizer=adam, loss='binary_crossentropy')
```

Alternative training

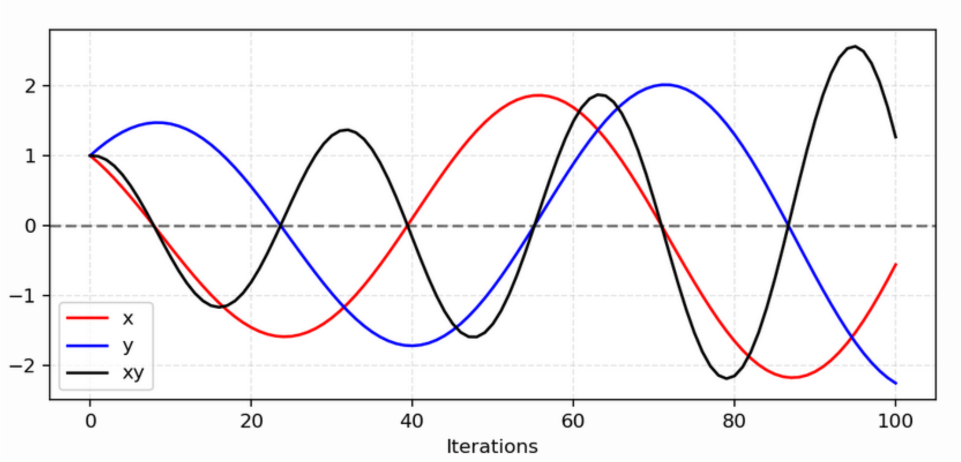
```
for i in range(nb_steps):
    # train the discriminator
    for _ in range(k_d):
        # sample a mini-batch of noise (generator input)
        z = np.random.normal(size=(batch_size, rand_dim))
        # sample a mini-batch of real images
        x = get_image_batch()
        # generate a batch of images with the current generator
        g_z = generator_model.predict(z)
        # update ϕ by taking an SGD step on mini-batch loss LD(ϕ)
        loss1 = discriminator_model.train_on_batch(x, np.random.uniform(low=0.7,
                                                                        high=1.2,
                                                                        size=batch_size))

        loss2 = discriminator_model.train_on_batch(g_z, np.random.uniform(low=0.0,
                                                                          high=0.3,
                                                                          size=batch_size))
    # train the generator
    for _ in range(k_g * 2):
        z = np.random.normal(size=(batch_size, rand_dim))
        # update θ by taking an SGD step on mini-batch loss LR(θ)
        loss=combined_model.train_on_batch(z, np.random.uniform(low=0.7,
                                                                high=1.2,
                                                                size=batch_size))
```

Difficulties of training GAN

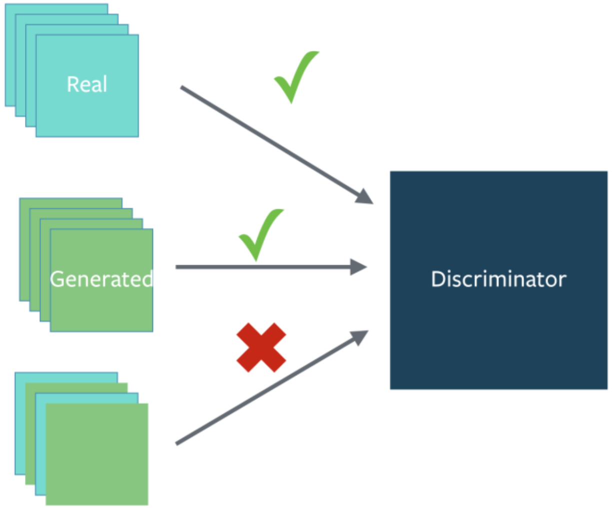
- **Vanishing gradient**
  - If the discriminator is perfect,
    - D(x) = 1 for real sample; D(x) = 0 for fake sample
  - Then the loss is zero → no gradient to update G(x)
- “If the discriminator behaves badly, the generator does not have accurate feedback and the loss function cannot represent the reality.
- If the discriminator does a great job, the gradient of the loss function drops down to close to zero and the learning becomes super slow or even jammed.” --- from https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html

Difficulties of training GAN

- **Mode Collapse**
  - The generator generates very similar samples
  - Low diversity
- **Difficult to optimize**
  - Not like traditional minimize or maximize problems
  - Min max → Nash equilibrium
  - Min f(x) = xy, f'(x) = y → x-= alpha * y
  - Min g(y) = -xy, g'(y) = -x → y += alpha *x



Improving GAN



- Normalize the inputs
  - Normalize the images between -1 and 1
  - Tanh as the last layer of the generator output
- Sample z from a gaussian distribution instead of uniform distribution
- BatchNorm
  - Construct different mini-batches for real and fake → bathnorm
  - For single instance, do instance normalization (standardization)
- Avoid Sparse Gradients: ReLU, MaxPool
- Use Dropouts in G in both train and test phase
- Use Soft and Noisy Labels
  - Real sample label: [0.7, 1.2]; fake sample label: [0, 0.3]
  - occasionally flip the labels when training the discriminator

Improving GAN

- DCGAN
  - Use batchnorm for most layers of D and G
    - except last layer of G and first layer of D
  - Avoid sparse gradient
  - Use Adam
- WGAN
  - Replace the KL divergence (~ loglikelihood) with Wasserstein distance
    - Wasserstein distance is also called earth mover distance

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))]$$

Improving GAN – semi-supervised training

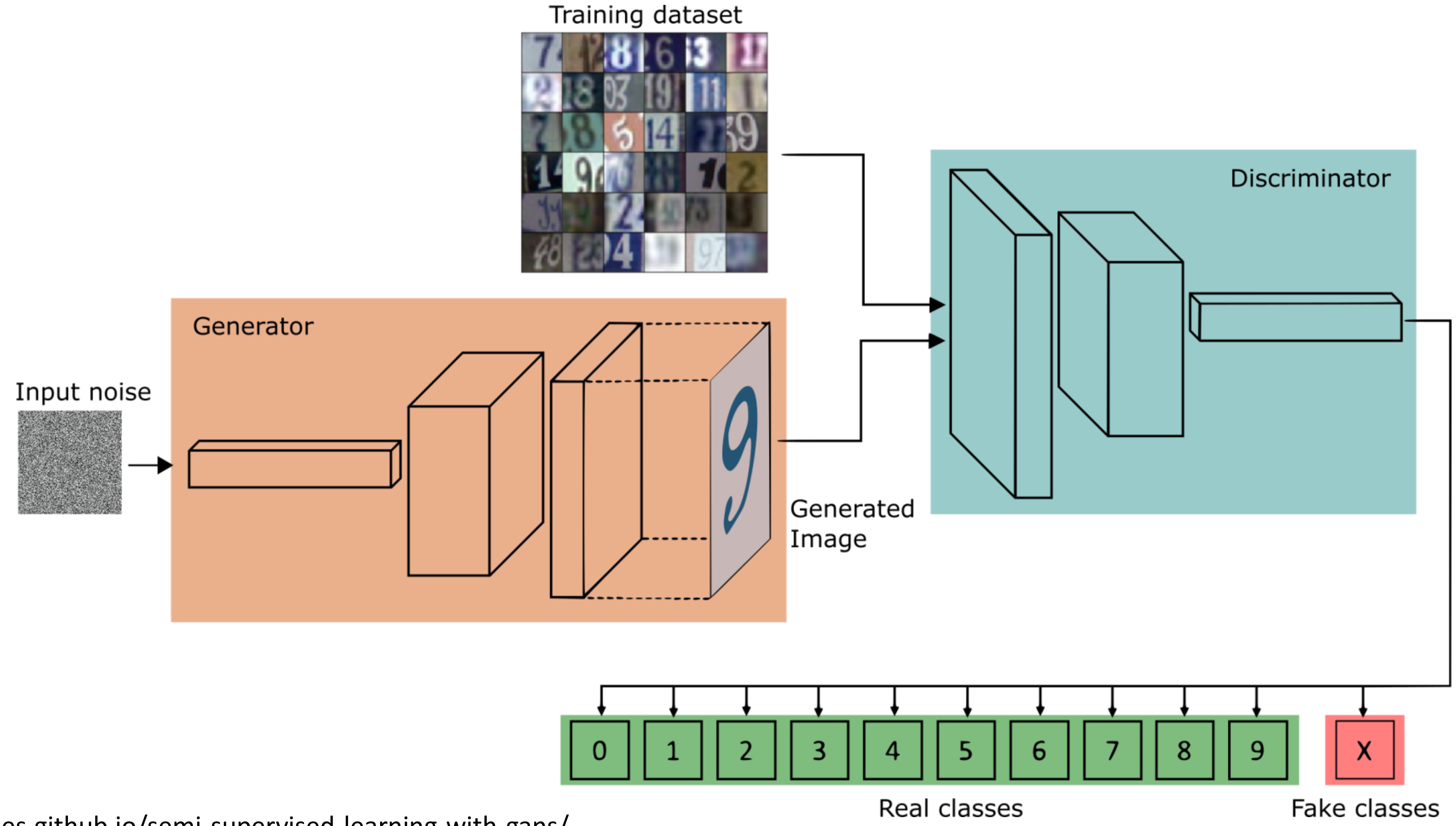


Image source: https://sthalles.github.io/semi-supervised-learning-with-gans/