# Generative Adversarial Networks (GANs) & Restricted Boltzmann Machines (RBMs)

## Practice Session

Soobin Um

October 7, 2021

# Outline of today's session

1. **Restricted Boltzmann Machines (RBMs)**

   Recap

   Coding

2. **Generative Adversarial Networks (GANs)**

   Recap

   Coding

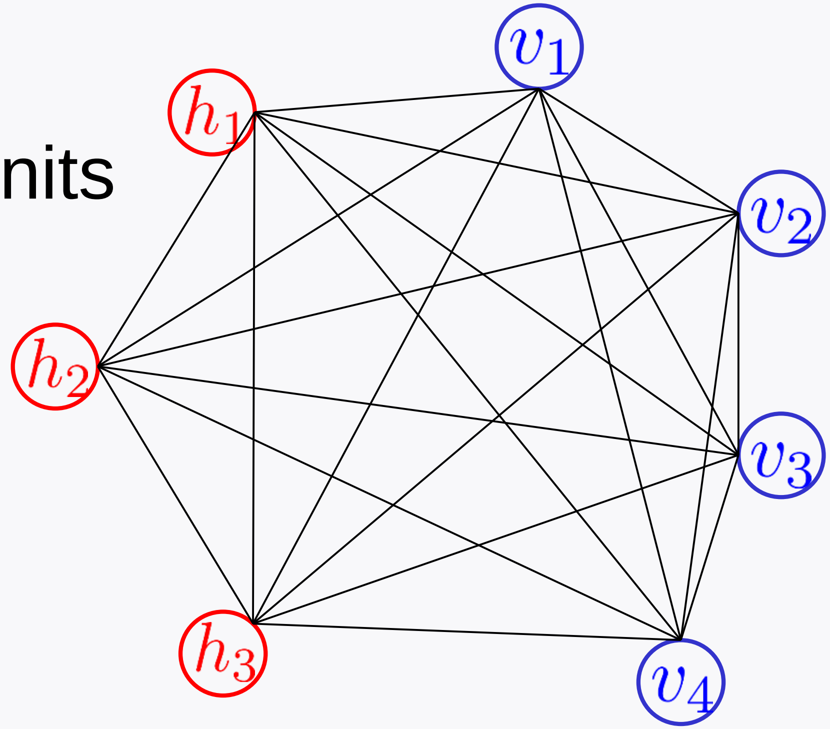
# Restricted Boltzmann Machines

# Recap

# Boltzmann Machine (BM)

hidden units

$h$




visible units

$v$

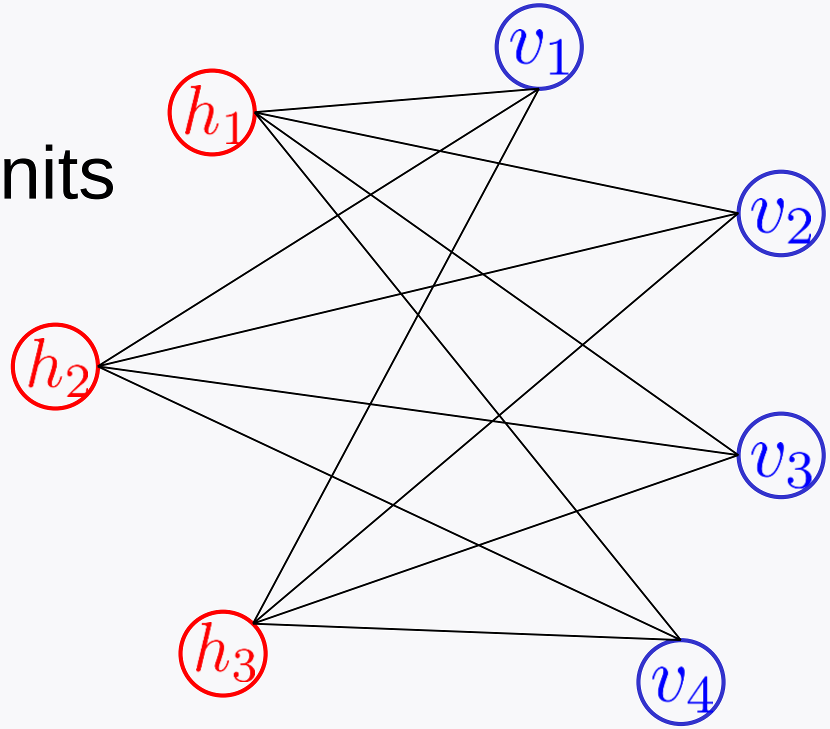
This captures arbitrary distribution between hidden and visible units:

$$\mathbb{P}(h, v)$$

# Restricted Boltzmann Machine (RBM)

hidden units

$h$



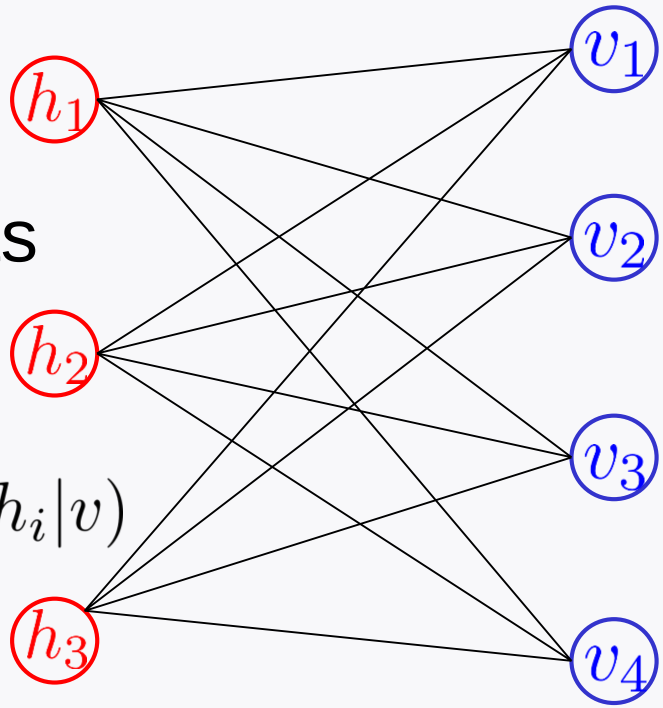

visible units

$v$

A simplified BM

No edge within hidden units as well as within visible units.

# How can RBM serve as a generative model?




hidden units

$h$

$$\mathbb{P}(h|v) = \prod_{i=1}^{d} \mathbb{P}(h_i|v)$$

visible units

$v$

$$\mathbb{P}(v|h) = \prod_{i=1}^{n} \mathbb{P}(v_i|h)$$

When only the visible units are available, can **generate hidden units** via

$$\mathbb{P}(h|v)$$

# Question

How to obtain such $\mathbb{P}(h|v)$ ?

To this end: Introduce a function that determines probabilities

**Energy**

$$\mathbb{P}(v,h) = \frac{e^{-E(v,h)}}{Z} \quad \text{where } Z = \sum_v \sum_h e^{-E(v,h)}$$

**Interpretation:**

Low energy → more probable

# Energy of visible units

Wish to find energy of $v$, say $F(v)$, such that

$$\mathbb{P}(v) = \sum_h \frac{e^{-E(v,h)}}{Z} = \frac{e^{-F(v)}}{Z}$$

$$F(v) = -\log\left(\sum_h e^{-E(v,h)}\right)$$

Called “**Free Energy**”.

(or, simply the energy of $v$ )

# How to parameterize energy

In RBM, we define $E(v, h)$ as:

$$E(v,h) := -b^T v - c^T h - h^T W v$$

$$\theta := (W, b, c)$$ parameters

# Parameterized conditional probabilities

$$\mathbb{P}(h|v) = \frac{e^{c^T h + h^T W v}}{\sum_h e^{c^T h + h^T W v}}$$

$$\mathbb{P}(v|h) = \frac{e^{b^T v + v^T W^T h}}{\sum_v e^{b^T v + v^T W^T h}}$$

Binary case:

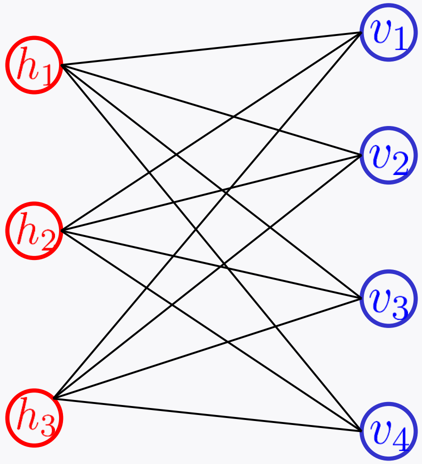
$$\mathbb{P}(h_i = 1|v) = \sigma(c_i + W_i v)$$

$$\mathbb{P}(v_i = 1|h) = \sigma(b_i + [W^T]_i h)$$

How to find good parameters $\theta$ ?

by training!

# Training procedure $\theta := (W, b, c)$




Given visible units with *m* examples:

$$\{v^{(i)}\}_{i=1}^{m}$$

**Step 1:** Sample $h^{(t),(i)} \sim \mathbb{P}(h|v^{(t),(i)}) \quad \forall i \in \{1, \dots, m\}$

**Step 2:** Sample $v^{(t),(i)} \sim \mathbb{P}(v|h^{(t),(i)}) \quad \forall i \in \{1, \dots, m\}$

**Step 3:** Compute a cost function: $J^{(t)}(\theta)$

**Step 4:** Update parameters via gradient descent:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \alpha^{(t)} \nabla J^{(t)}(\theta)$$

# Loss function $\ell\left(v^{(i)}, v^{(t),(i)}\right)$?

**Turns out:** The following loss is optimal in a certain sense:

$$\ell_{\mathsf{opt}}\left(v, \hat{v}\right) = F(v) - F(\hat{v})$$

↑ free energy

$$\text{where } F(v) = -\log\left(\sum_h e^{-E(v,h)}\right);$$

$$E(v,h) := -b^T v - c^T h - h^T W v.$$

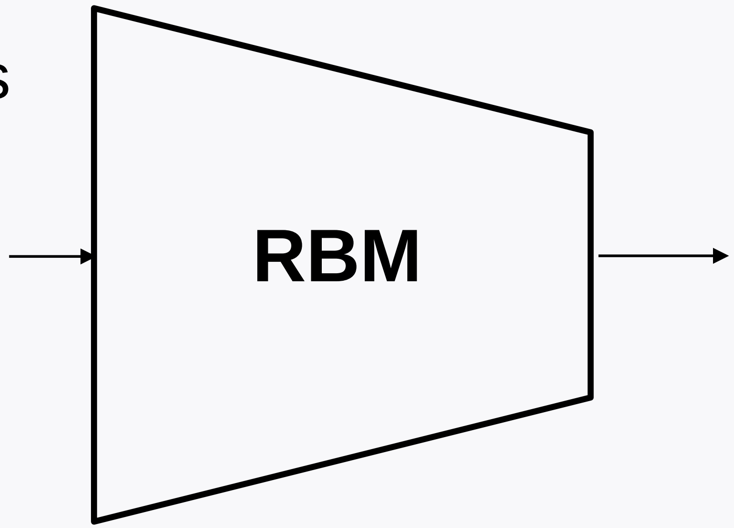
# Coding

# Task: Generative modeling

visible units

$v \in \mathbf{R}^{768}$

MNIST data




hidden units

$h \in \mathbf{R}^{d}$

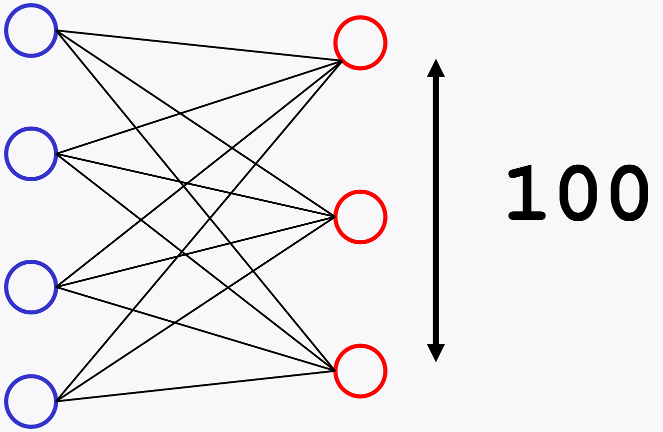
**Recall:**

When only the visible units are available, can **generate hidden units.**

# Code: Define a simple RBM model

```
from sklearn.neural_network import BernoulliRBM

rbm = BernoulliRBM(n_components=100, learning_rate=0.01)
```

Dim of hidden units



# Code: Training the RBM model

```
from sklearn.neural_network import BernoulliRBM

rbm = BernoulliRBM(n_components=100, learning_rate=0.01)

rbm.fit(X_train) ⟶ learn theta = (W,b,c)
```

**Step 1:** Sample $h^{(t),(i)} \sim \mathbb{P}(h|v^{(t),(i)}) \quad \forall i \in \{1, \dots, m\}$

**Step 2:** Sample $v^{(t),(i)} \sim \mathbb{P}(v|h^{(t),(i)}) \quad \forall i \in \{1, \dots, m\}$

**Step 3:** Compute a cost function: $J^{(t)}(\theta)$

**Step 4:** Update parameters via gradient descent:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \alpha^{(t)} \nabla J^{(t)}(\theta)$$

# Code: Compute $\mathbb{P}(h|v)$

```
from sklearn.neural_network import BernoulliRBM

rbm = BernoulliRBM(n_components=100, learning_rate=0.01)

rbm.fit(X_train) ⟶ learn theta = (W,b,c)

x_latent=rbm.transform(X_train)
    ↑
Compute the hidden layer probabilities:
```

$$\mathbb{P}(h_i = 1|v) = \frac{e^{c_i + W_i v}}{1 + e^{c_i + W_i v}}, \quad i \in \{1, \ldots, 100\}$$

$v$ ↓ **X_train**

$100$ ↓ **n_components**

# Code: Compute Free energy of $v$

```
from sklearn.neural_network import BernoulliRBM

rbm = BernoulliRBM(n_components=100, learning_rate=0.01)

rbm.fit(X_train) → learn theta = (W,b,c)

x_latent=rbm.transform(X_train)
```

↑

Compute the hidden layer probabilities: $\mathbb{P}(h_i = 1|v)$

`rbm.score_samples(X_train)` Compute Free energy w.r.t. X_train:

$$\propto F(v) = -\log\left(\sum_h e^{-E(v,h)}\right) \qquad E(v,h) := -b^T v - c^T h - h^T W v$$

↓ (v =) **X_train**

# Code: Gibbs Sampling

```
from sklearn.neural_network import BernoulliRBM

rbm = BernoulliRBM(n_components=100, learning_rate=0.01)

rbm.fit(X_train) ⟶ learn theta = (W,b,c)

x_latent=rbm.transform(X_train)
    ↑
Compute the hidden layer probabilities:
```

$\mathbb{P}(h_i = 1|v)$

```
rbm.score_samples(X_train) Compute Free energy of X_train:
```

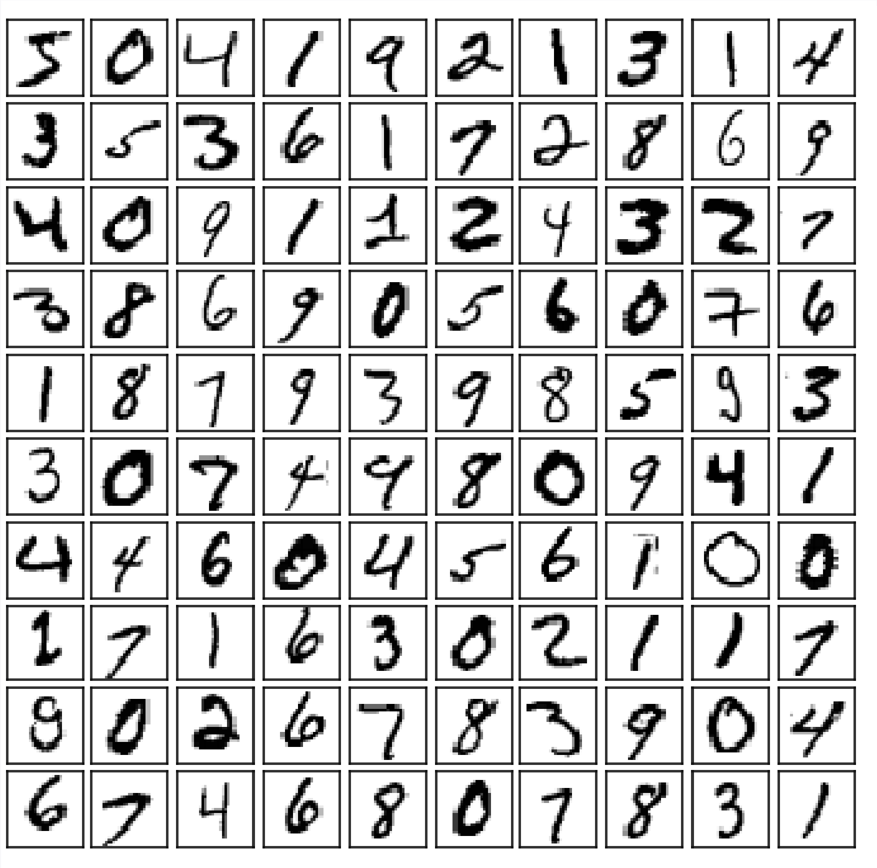
$F(v)$

```
X_hat = rbm.gibbs(X_train[:100])
```

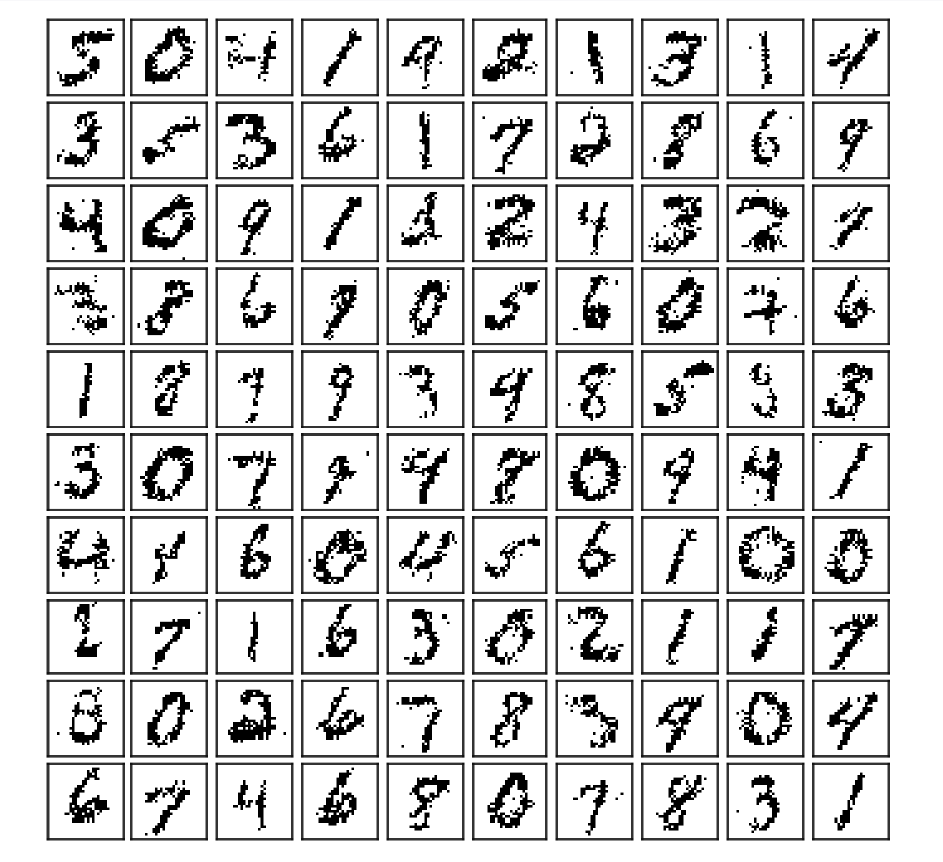
**X_train** ⟶ **h** ⟶ **X_hat** (sampled)

$\mathbb{P}(h|v)$ $\mathbb{P}(v|h)$

# Comparison: Original vs. sampling



X_train



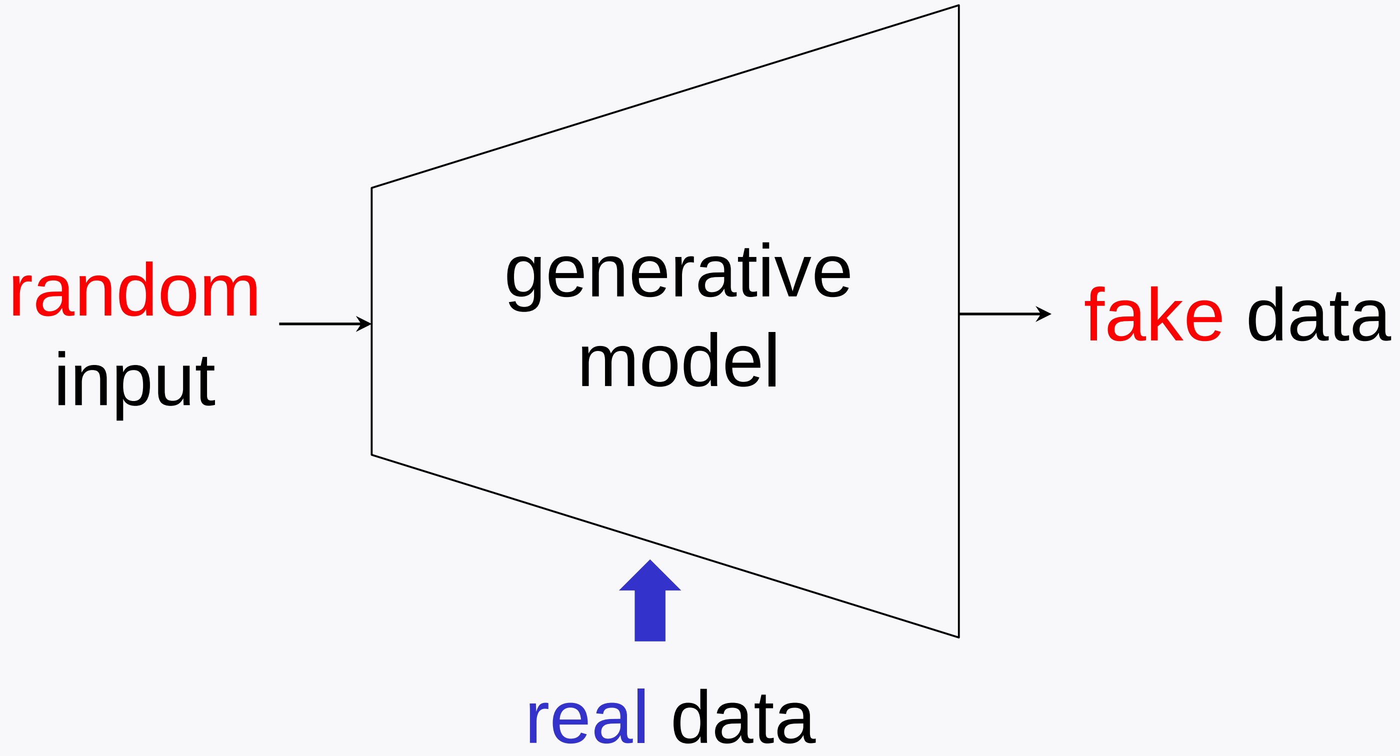
X_hat

# Generative Adversarial Networks

# Recap

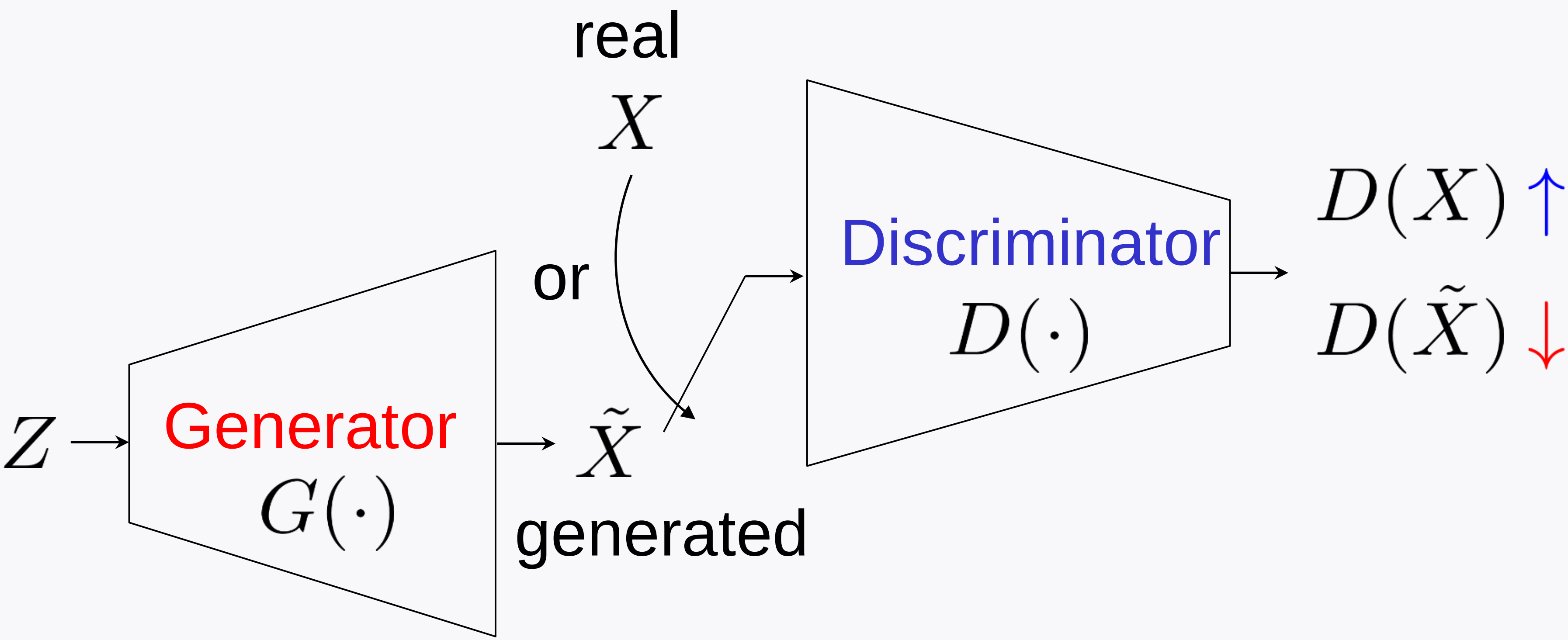
# A generative model

A model that generates fake data which has a similar distribution as that of real data.



# Generative Adversarial Networks

Goodfellow et al.
NeurIPS14

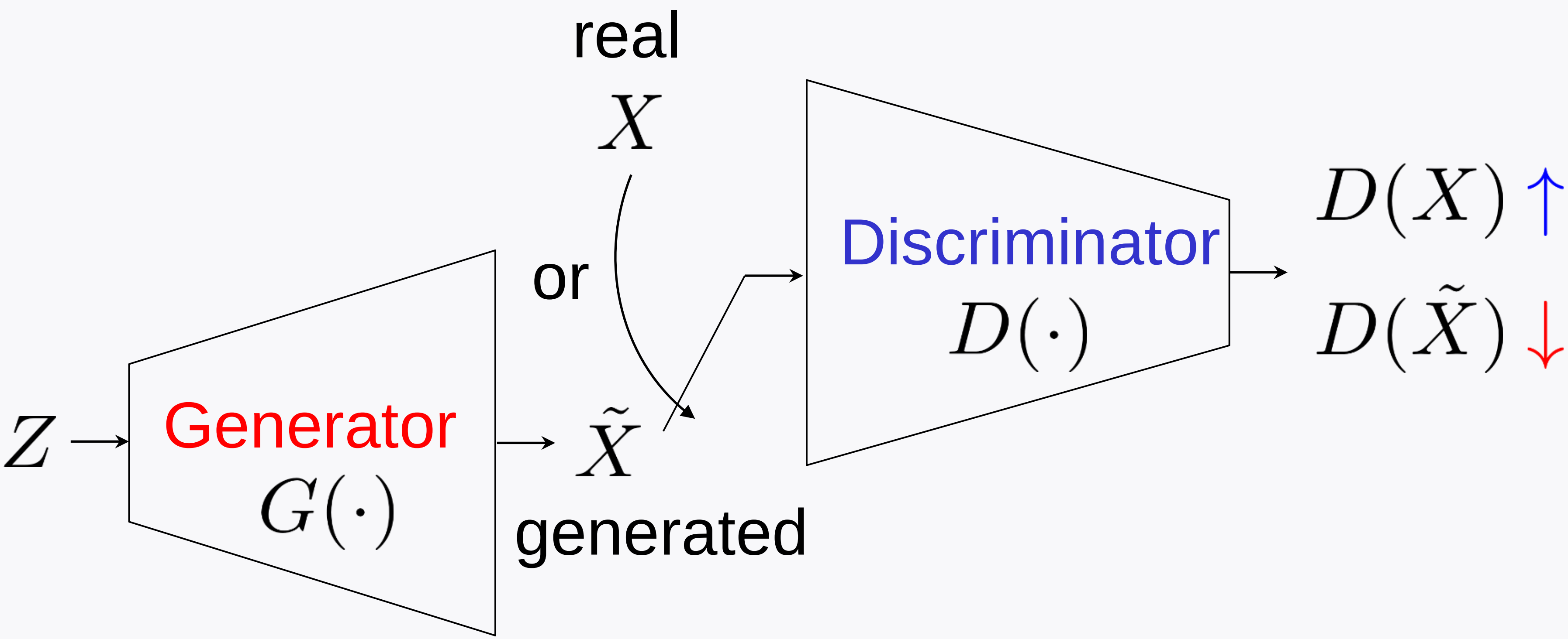



**Role:** Discriminate real from generated fake samples

Intend to yield a large $D(\cdot)$ if the input is real data;

a small $D(\cdot)$ for generated data.

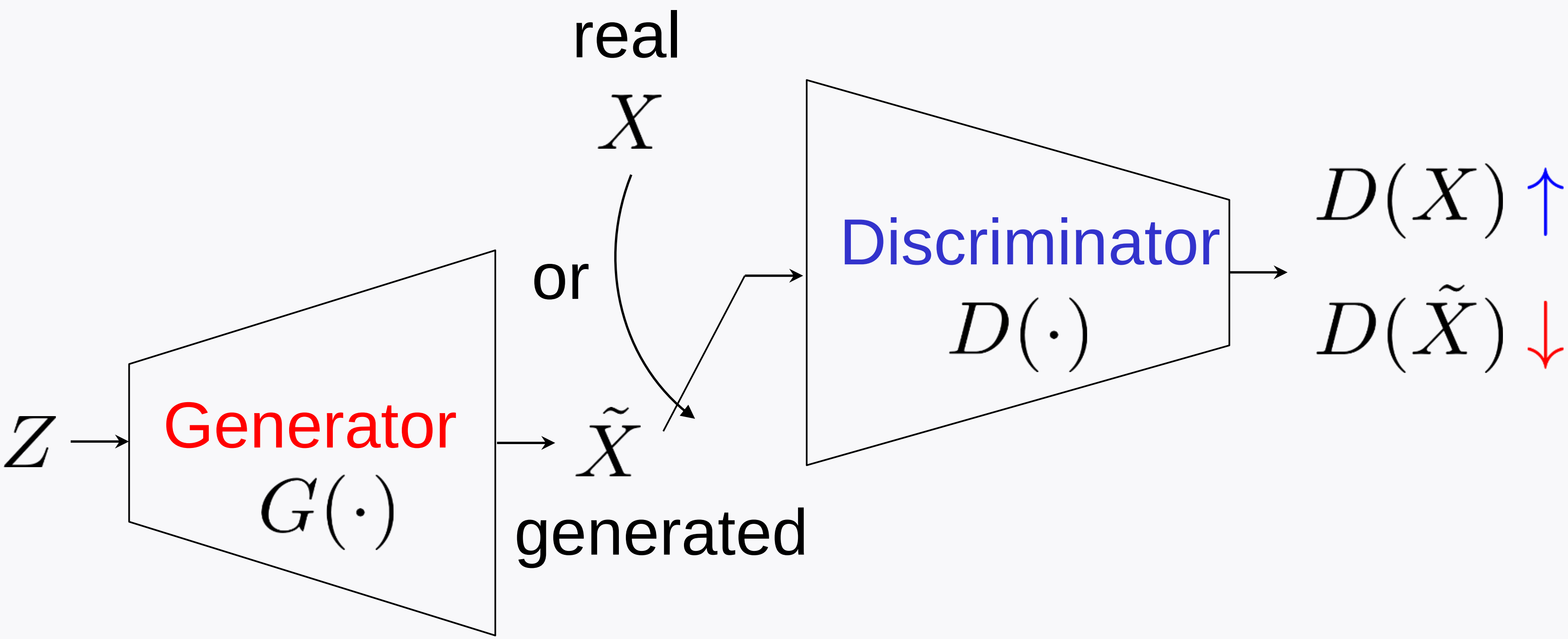
# A reasonable interpretation on $D(\cdot)$




Probability of the input being real:

$$D(\cdot\ \ ) = \mathbb{P}((\cdot) = \text{real})$$

# A reasonable interpretation on $D(\cdot)$

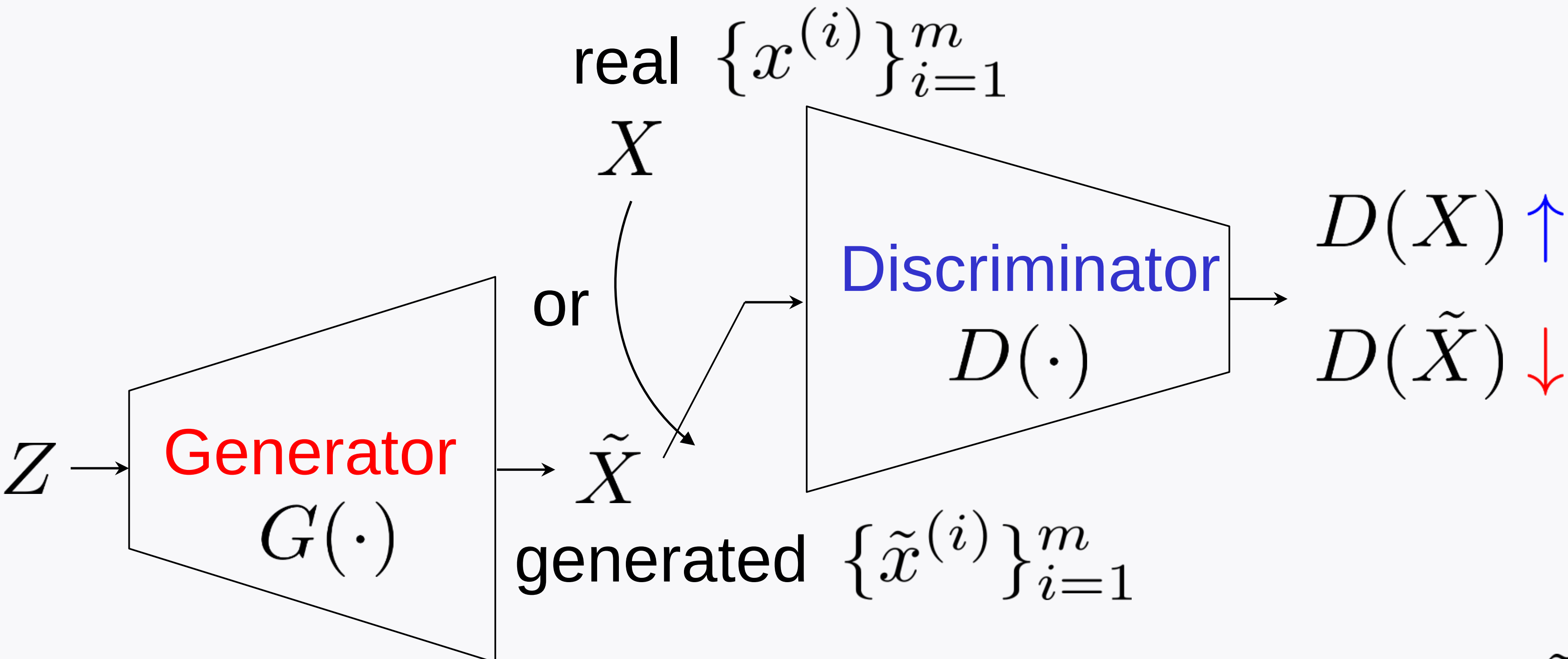



Probability of the input being real:

$$\uparrow \quad D(X) = \mathbb{P}(X = \text{real}) = 1$$

$$\downarrow \quad D(\tilde{X}) = \mathbb{P}(\tilde{X} = \text{real}) = 0$$

# Optimization? Log loss!

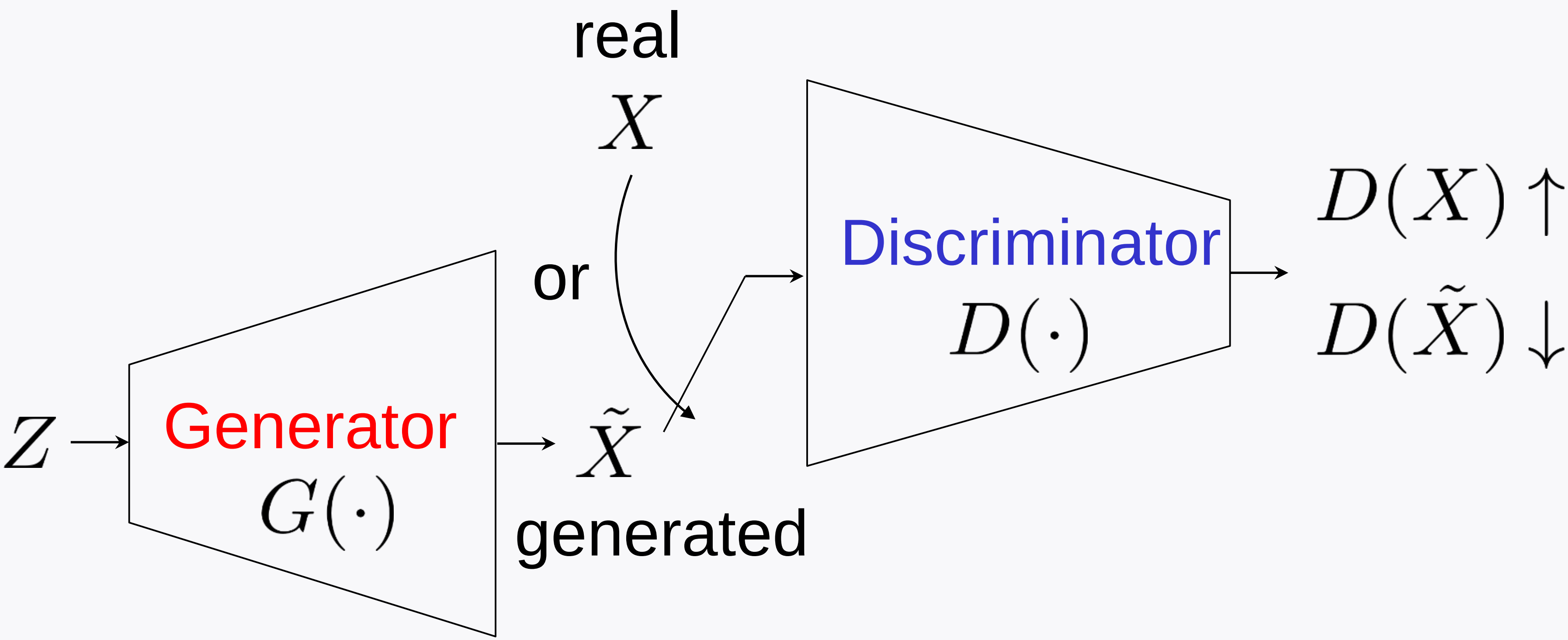



Discriminator wishes to maximize: $D(X)$ & $1 - D(\tilde{X})$

Goodfellow employed log loss:

$$\max_D \frac{1}{m}\sum_{i=1}^m \log D(x^{(i)}) + \frac{1}{m}\sum_{i=1}^m \log(1 - D(\tilde{x}^{(i)}))$$

# Optimization for GAN




Discriminator: $\max_{D} \frac{1}{m}\sum_{i=1}^{m} \log D(x^{(i)}) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(\tilde{x}^{(i)}))$

Generator: $\min_{G} \frac{1}{m}\sum_{i=1}^{m} \log D(x^{(i)}) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(\tilde{x}^{(i)}))$

$\tilde{x}^{(i)} = G(z^{(i)})$

# Neural net optimization

$$\min_{G(\cdot)\in\mathcal{N}} \max_{D(\cdot)\in\mathcal{N}} \frac{1}{m}\sum_{i=1}^{m} \log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))$$

Take function class as neural networks.

And then parameterize them:

$$G_{w}(\cdot) \quad D_{\theta}(\cdot)$$

# How to deal with min-max

$$\min_{w} \max_{\theta} \frac{1}{m} \sum_{i=1}^{m} \log D_\theta(x^{(i)}) + \log(1 - D_\theta(G_w(z^{(i)})))$$

$$:= J(w, \theta)$$

Find a stationary point such that

$$\nabla_w J(w^*, \theta^*) = 0, \ \nabla_\theta J(w^*, \theta^*) = 0$$

**Turns out:** Such point often yields a near optimal performance in reality.

**One practical method:**

Alternating gradient descent

# *k*:1 alternating gradient descent

1. Update Generator's weight:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_1 \nabla_w J(w^{(t)}, \theta^{(t \cdot k)})$$

2. Update Discriminator's weight *k* times:

```
for i=1:k
```

$$\theta^{(t \cdot k+i)} \leftarrow \theta^{(t \cdot k+i-1)} + \alpha_2 \nabla_\theta J(w^{(t+1)}, \theta^{(t \cdot k+i-1)})$$

3. Repeat the above.

**In practice:** Often use Batch version & Adam.

# A practical tip on Generator

Given Discriminator's parameter $\theta$ :

$$\min_{w} \frac{1}{m} \sum_{i=1}^{m} \underbrace{\log D_{\theta}(x^{(i)})}_{\text{irrelevant of } w} + \log(1 - D_{\theta}(G_{w}(z^{(i)})))$$

Suffice to consider:

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

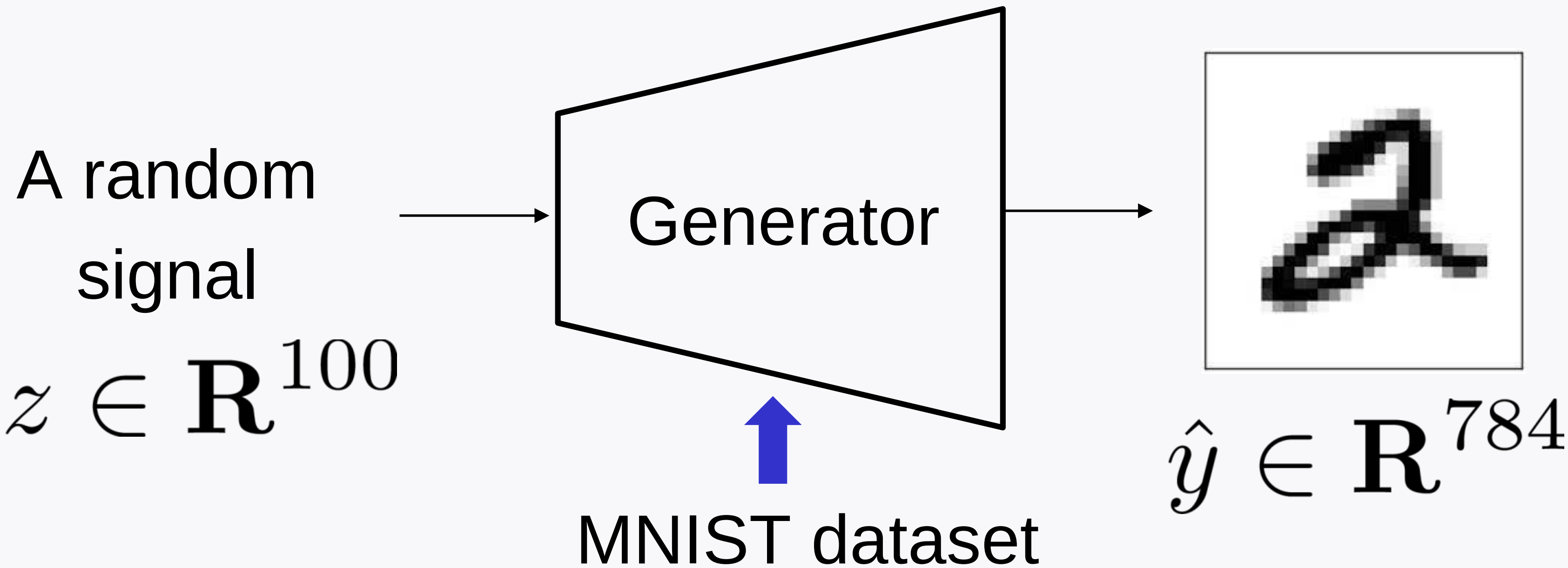
In practice, consider a *proxy*:

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

# Coding

# Task

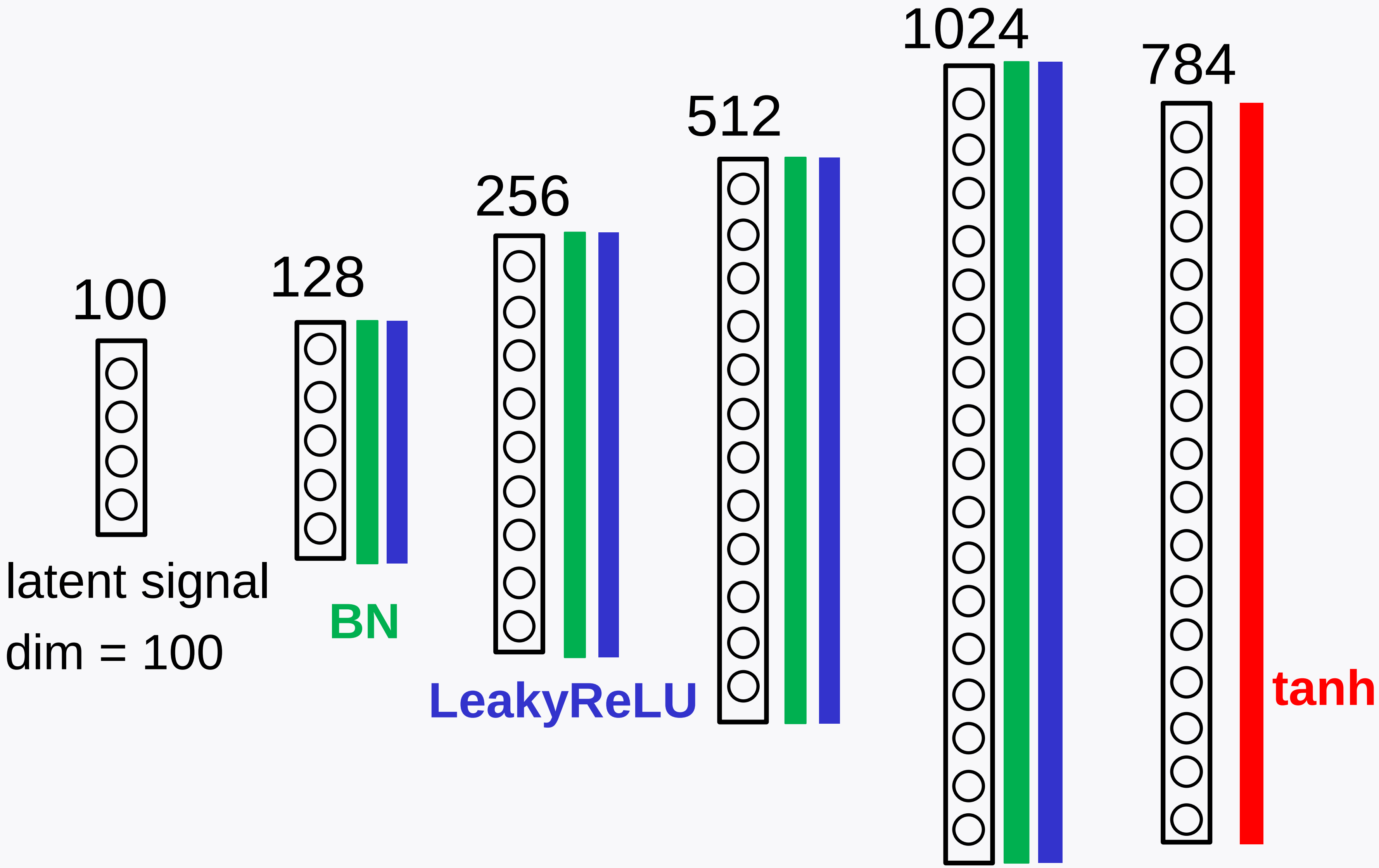
Generate MNIST-like images.



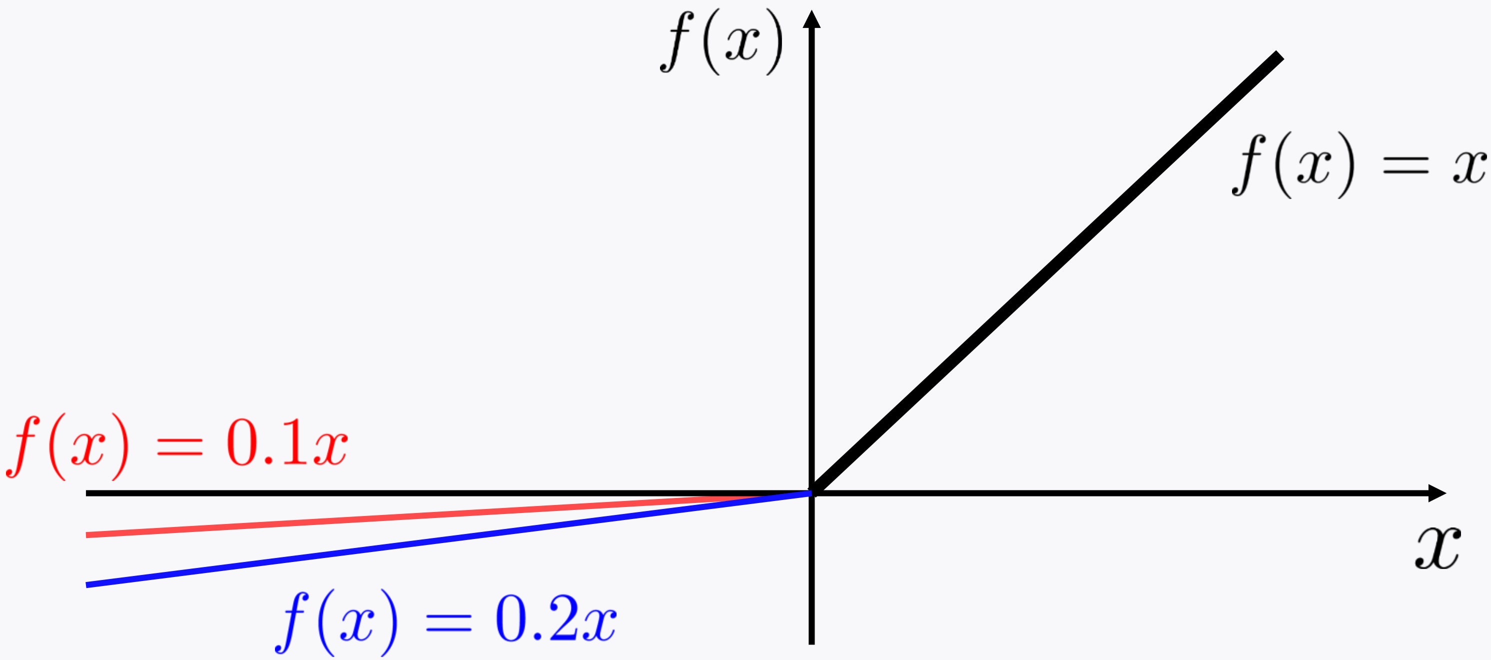
# Code: Data Normalization

```
from tensorflow.keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(-1,28*28)/ 127.5 - 1

x_test = x_test.reshape(-1,28*28)/ 127.5 - 1 # Normalize data in  [-1, 1]
```

# Model for Generator



# Leaky ReLU

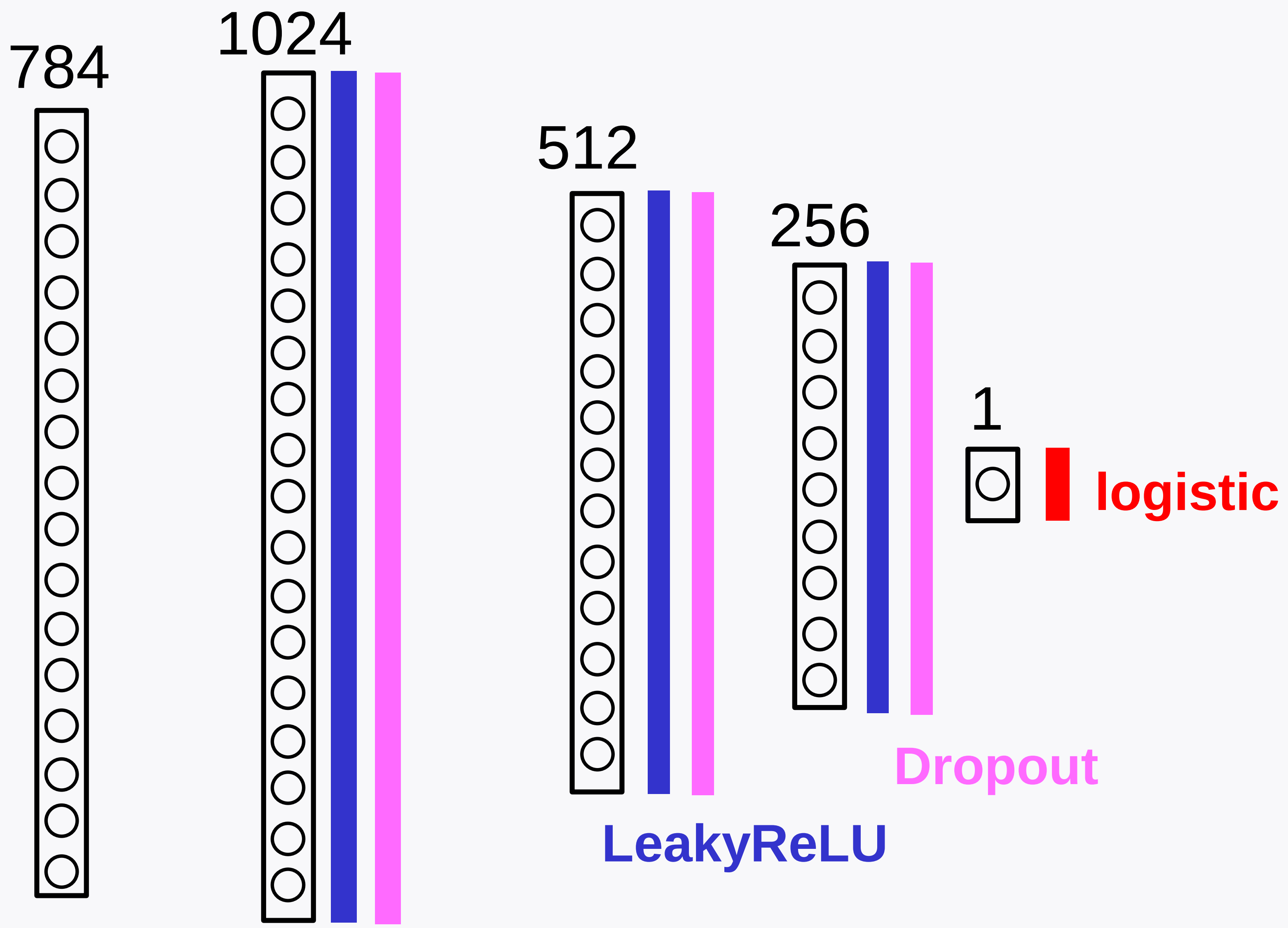



$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if} \quad x \geq 0; \\ \text{negative slope} \times x & \text{otherwise.} \end{cases}$$

# Code: Generator

```
from tensorflow.keras.layers import Dense, BatchNormalization, LeakyReLU
from tensorflow.keras.models import Sequential

generator = Sequential()
generator.add(Dense(128,input_dim=100))
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.2))
generator.add(Dense(256))
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.2))
generator.add(Dense(512))
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.2))
generator.add(Dense(1024))
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.2))
generator.add(28*28, activation='tanh')
```

# Model for Discriminator



# Code: Discriminator

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LeakyReLU, Dropout
discriminator = Sequential()
discriminator.add(Dense(1024 , input_shape=(784,)))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(512))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(256))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dense(1 , activation='sigmoid'))
```

Which loss function for training?

# Recall: Discriminator optimization

$$\max_{\theta} \frac{1}{m} \sum_{i=1}^{m} \log D_{\theta}(x^{(i)}) + \log(1 - D_{\theta}(G_w(z^{(i)})))$$

This reminds us: **Cross entropy (CE) loss!**

$$l_{\text{CE}}(y, \hat{y}) := -y \log \hat{y} - (1 - y) \log (1 - \hat{y})$$

**An equivalent form:**

$$\max_{\theta} \frac{1}{m} \sum_{i=1}^{m} -l_{\text{CE}}(\mathbf{1}, D_{\theta}(x^{(i)})) - l_{\text{CE}}(\mathbf{0}, D_{\theta}(G_w(z^{(i)})))$$

# Code: Discriminator

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LeakyReLU, Dropout
discriminator = Sequential()
discriminator.add(Dense(1024 , input_shape=(784,)))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(512))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dropout(0.3))
discriminator.add(Dense(256))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dense(1 , activation='sigmoid'))

discriminator.compile(loss='binary_crossentropy', optimizer='adam')
```

# Recall: Generator optimization (the *proxy*)

$$\min_{w} \frac{1}{m} \sum_{i=1}^{m} - \log \underline{D_{\theta}(G_{w}(z^{(i)}))}$$

Should examine discriminator outputs!

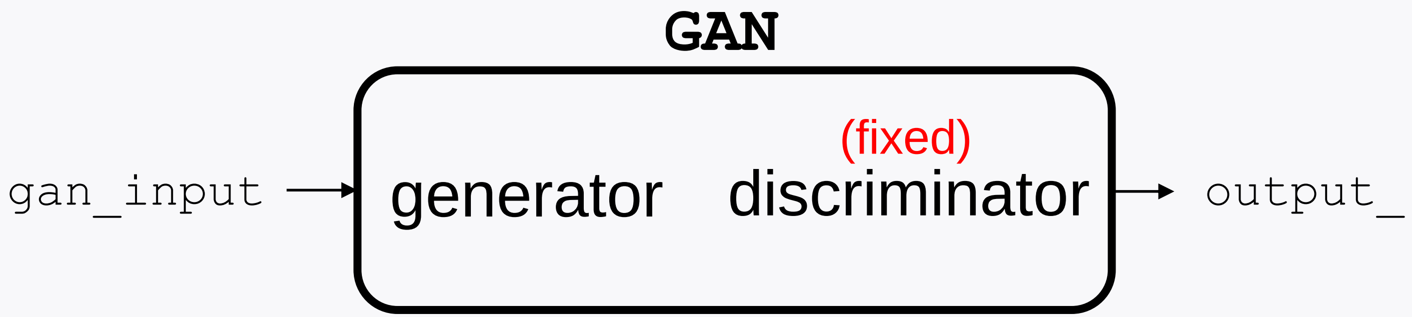
To implement this: Construct an integrated model only for training the generator

**An equivalent form using the CE loss:**

$$\min_{w} \frac{1}{m} \sum_{i=1}^{m} l_{\text{CE}}(\mathbf{1}, D_{\theta}(G_{w}(z^{(i)})))$$

# Code: Generator + Discriminator

```
from tensorflow.keras.models import Model

discriminator.trainable = False

gan_input = Input(shape=(100,))

 x = generator(inputs=gan_input)

output = discriminator(x)

GAN = Model(gan_input, output)

GAN.compile(loss='binary_crossentropy', optimizer='adam')
```



# Code: Alternating gradient descent (k=1)

Update discriminator weights $\max_{\theta} \frac{1}{m} \sum_{i=1}^{m} \log D_\theta(x^{(i)}) + \log(1 - D_\theta(G_w(z^{(i)})))$

```
noise = np.random.uniform(-1, 1, size=[BATCH_SIZE, 100])
generated_images = generator.predict(noise)  # fake image generation
x_dis = np.concatenate([real_images, generated_images])
y_dis = np.zeros(2 * BATCH_SIZE)
y_dis[:BATCH_SIZE] = 1
discriminator.train_on_batch(x_dis, y_dis)
```

Update generator weights $\min_{w} \frac{1}{m} \sum_{i=1}^{m} -\log D_\theta(G_w(z^{(i)}))$

```
noise = np.random.uniform(-1, 1, size=[BATCH_SIZE,100])
y_fake = np.ones(BATCH_SIZE)  # fake labels
GAN.train_on_batch(noise, y_fake)
```

# Generated images

Generator outputs