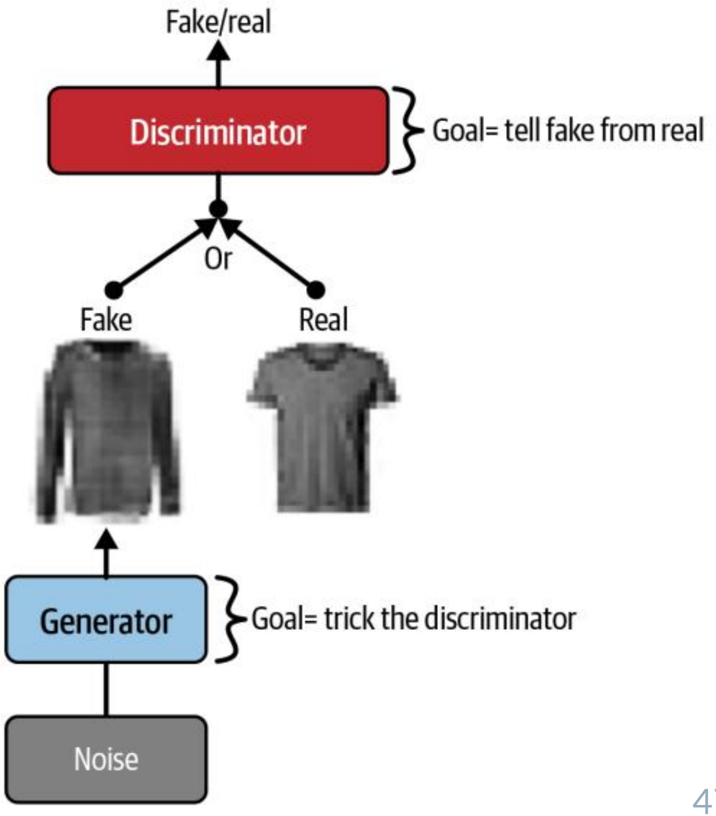
2.
Generative Adversarial Network

The Idea Behind GANs

- Idea: make neural networks compete against each other in the hope that this competition will push them to excel.

- A GAN is composed of two neural networks:
  - *Generator* learns to make fakes that look real.
  - *Discriminator* learns to distinguish real from fake.

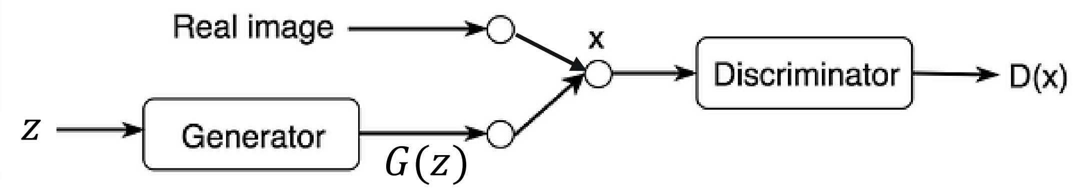


Generator and Discriminator

- *Generator*
  - Takes a random distribution as input (typically Gaussian) and outputs some data—typically, an image.
  - You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated.
  - The generator functions similar to a decoder in a variational autoencoder, and it can be used in the same way to generate new images: just feed it some Gaussian noise, and it outputs a brand-new image
- *Discriminator*
  - Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.

Training a GAN

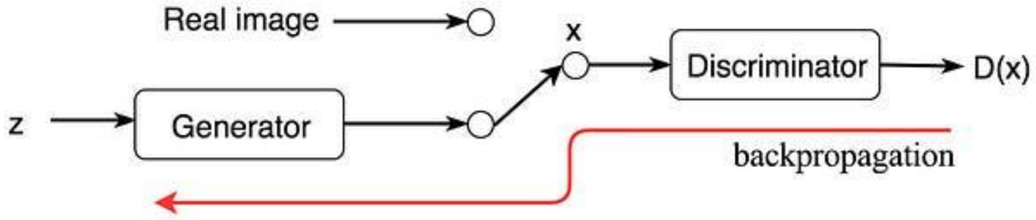
- Phase 1: training the discriminator.
  - A batch of real images is sampled from the training set and is completed with an equal number of fake images produced by the generator.
  - We use label 0 for fake images and 1 for real images, and the discriminator is trained on this labeled batch for one step, using the binary cross-entropy loss.
  - Backpropagation only optimizes the weights of the discriminator in this phase.



$$\max_D V(D) = \underbrace{\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})]}_{\text{recognize real images better}} + \underbrace{\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]}_{\text{recognize generated images better}}$$

Training a GAN

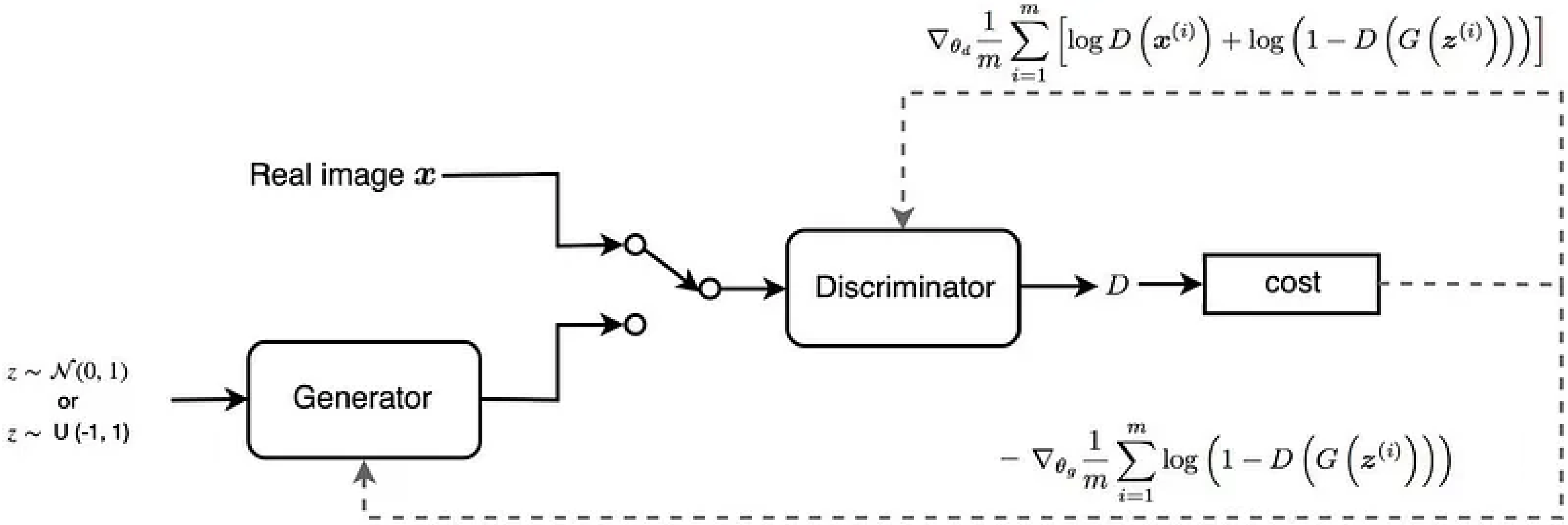
- Phase 2: training the generator.
  - Use the generator to produce another batch of fake images.
  - Do not add real images in the batch, and set all the labels to 1 (real): we want the generator to produce images that the discriminator will believe to be real!
  - The weights of the discriminator are frozen in this step, so backpropagation only affects the weights of the generator.



$$\min_{G} V(G) = \underline{\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]}$$

Optimize G that can fool the discriminator the most.

Training a GAN



$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Building a Simple GAN

```
codings_size = 30

Dense = tf.keras.layers.Dense
generator = tf.keras.Sequential([
    Dense(100, activation="relu", kernel_initializer="he_normal"),
    Dense(150, activation="relu", kernel_initializer="he_normal"),
    Dense(28 * 28, activation="sigmoid"),
    tf.keras.layers.Reshape([28, 28])
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    Dense(150, activation="relu", kernel_initializer="he_normal"),
    Dense(100, activation="relu", kernel_initializer="he_normal"),
    Dense(1, activation="sigmoid")
])
gan = tf.keras.Sequential([generator, discriminator])
```

Compiling the Models

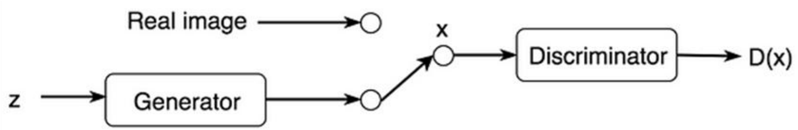
- The discriminator is a binary classifier, so we use the binary cross-entropy loss:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

- The `gan` model is also a binary classifier, so it can use the binary cross-entropy loss as well.
  - The generator will only be trained through the `gan` model, so we do not need to compile it at all.
  - The discriminator should not be trained during the second phase, so we make it non-trainable before compiling the `gan` model.

```
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

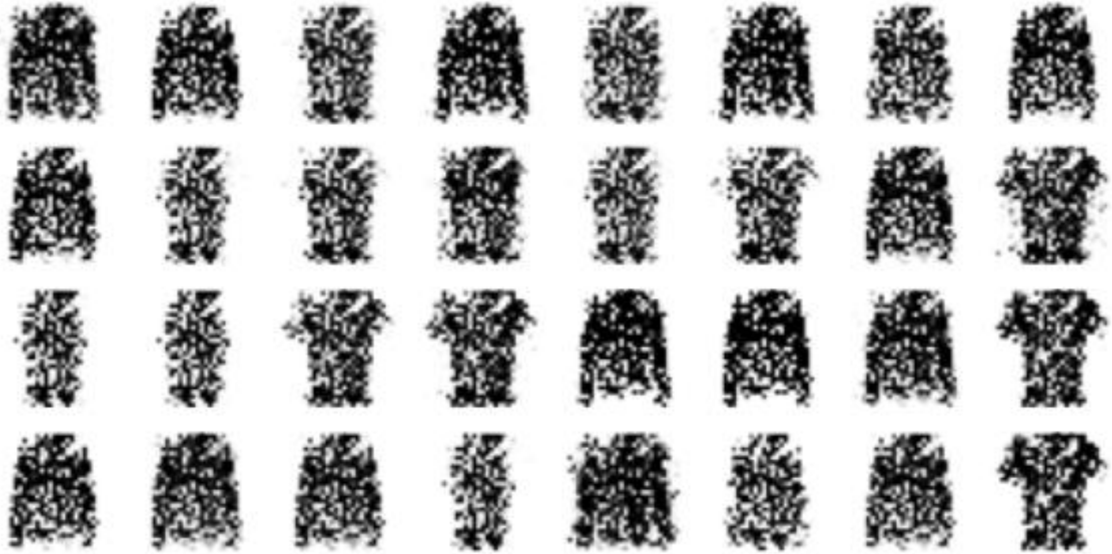
Fitting the Models



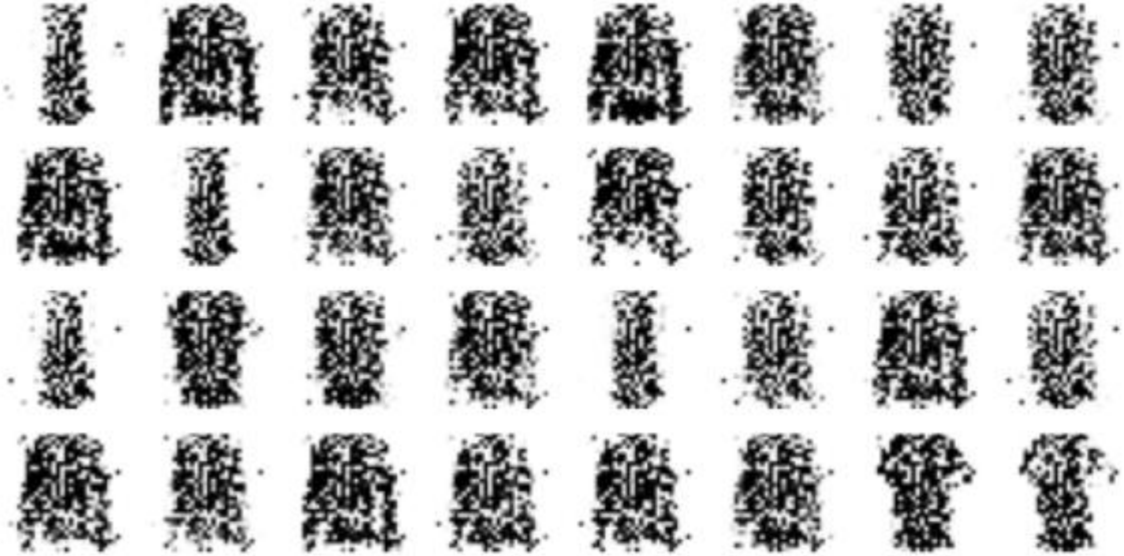

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.train_on_batch(X_fake_and_real, y1)
            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size, n_epochs=50)
```

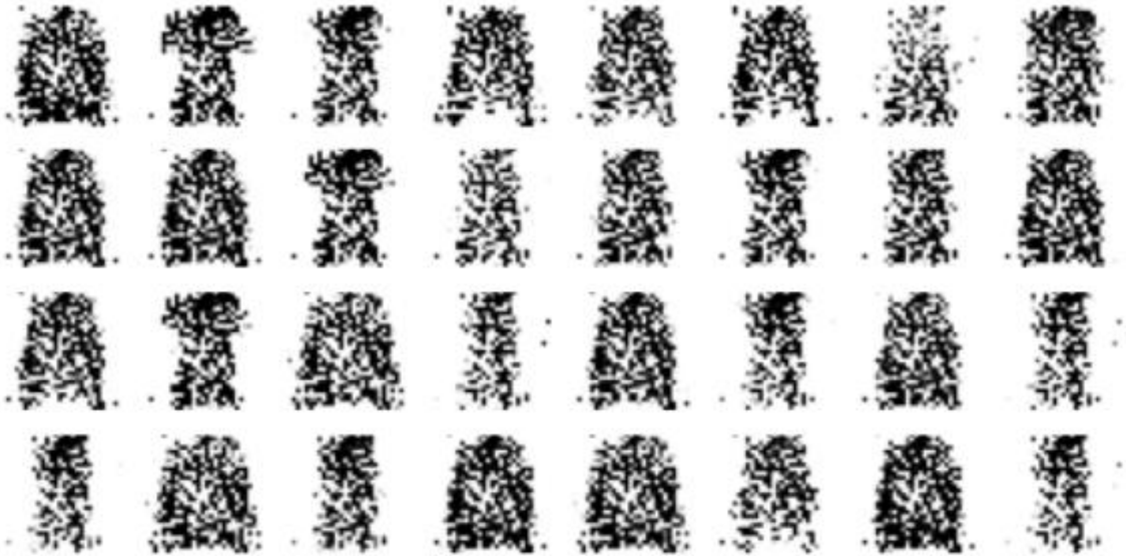
Epoch 1/50



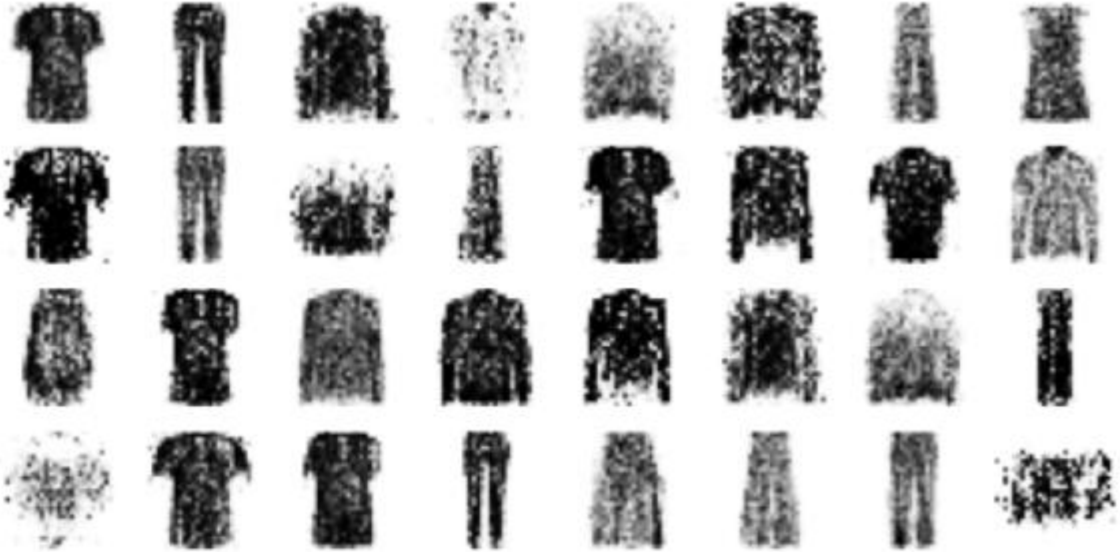
Epoch 2/50



Epoch 3/50



```
<<44 more epochs>>
Epoch 48/50
```




Epoch 49/50

Epoch 50/50

Generating New Images

- After training, you can randomly sample some codings from a Gaussian distribution, and feed them to the generator to produce new images:

```
codings = tf.random.normal(shape=[batch_size, codings_size])
generated_images = generator.predict(codings)
```

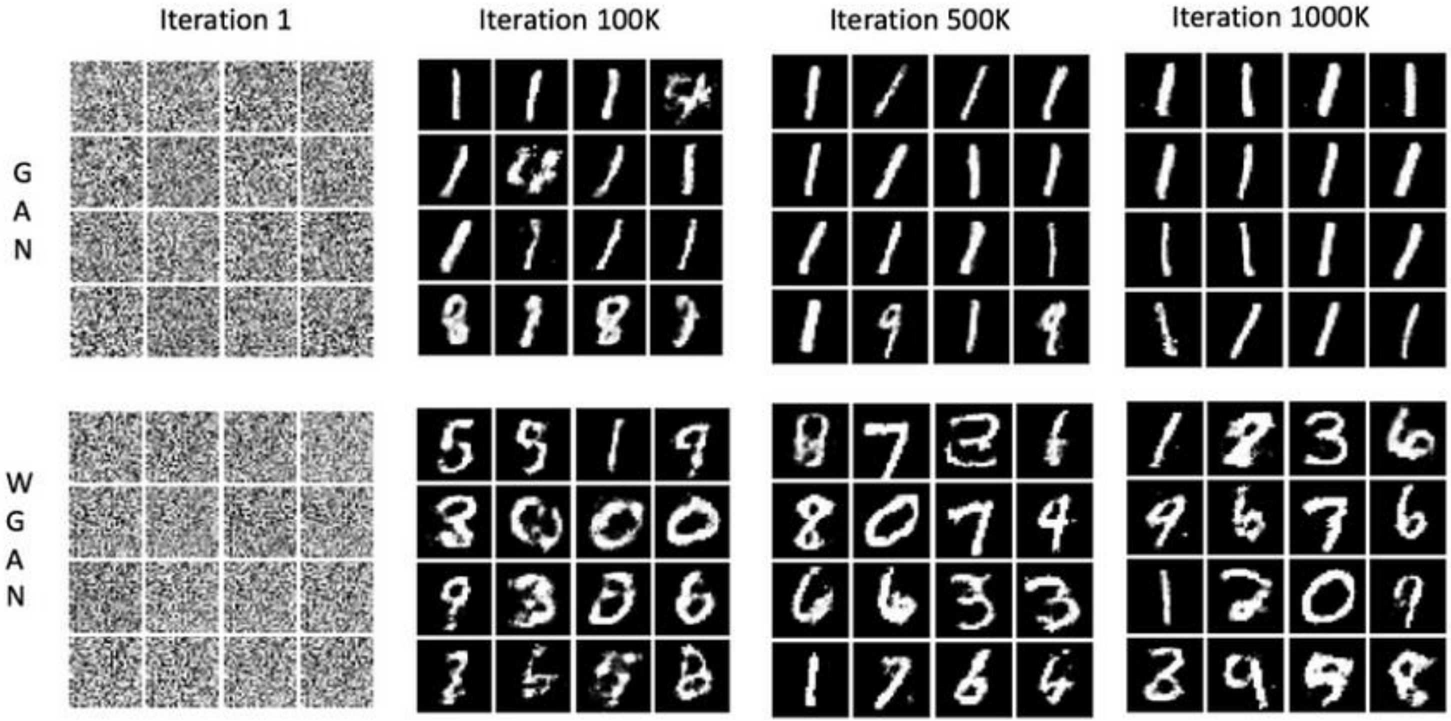
Challenges of Training GANs

- During training, the generator and the discriminator constantly try to outsmart each other, in a zero-sum game.
  - As training advances, the game may end up in a *Nash equilibrium*.
- A GAN can only reach a single Nash equilibrium: that's when the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake).
  - You just need to train the GAN for long enough, and it will eventually reach this equilibrium, giving you a perfect generator.
  - But nothing guarantees that the equilibrium will ever be reached!
- *Mode collapse*: when the generator's outputs gradually become less diverse.

Mode Collapse

- Suppose that the generator gets better at producing convincing shoes than any other class.
  - It will fool the discriminator more with shoes, and this will encourage it to produce more images of shoes, and gradually, it will forget how to produce anything else.
- The only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes.
- Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class.
  - It may then become good at shirts, forgetting about shoes, and the discriminator will follow.
- The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

Mode Collapse



Sensitivity to Hyperparameters

- Because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable.
  - Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities.
- Since many factors affect these complex dynamics, GANs are very sensitive to the hyperparameters: you may have to spend a lot of effort fine-tuning them.
- In fact, that's why we used RMSProp rather than the usual Nadam when compiling the models.
  - When we used Nadam, we ran into a severe mode collapse.

Suggested Solutions

- *Experience replay*: storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and training the discriminator using real images plus fake images drawn from this buffer.
  - This reduces the chances that the discriminator will overfit the latest generator's outputs.
- *Mini-batch discrimination*: measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole batch of fake images that lack diversity.
  - This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse.

Deep Convolutional GANs

- Guidelines for building stable deep convolutional GANs (DCGANs):
  - Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
  - Use batch normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
  - Remove fully connected hidden layers for deeper architectures.
  - Use ReLU activation in the generator for all layers except the output layer, which should use tanh.
  - Use leaky ReLU activation in the discriminator for all layers.
- These guidelines will work in many cases, but not always, so you may still need to experiment with different hyperparameters.
  - In fact, just changing the random seed and training the exact same model again will sometimes work.

Deep Convolutional GANs

```python
codings_size = 100

generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7 * 7 * 128),
    tf.keras.layers.Reshape([7, 7, 128]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same", activation="relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same", activation="tanh"),
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same", activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same", activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
gan = tf.keras.Sequential([generator, discriminator])
```

Compiling and Training the Model

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

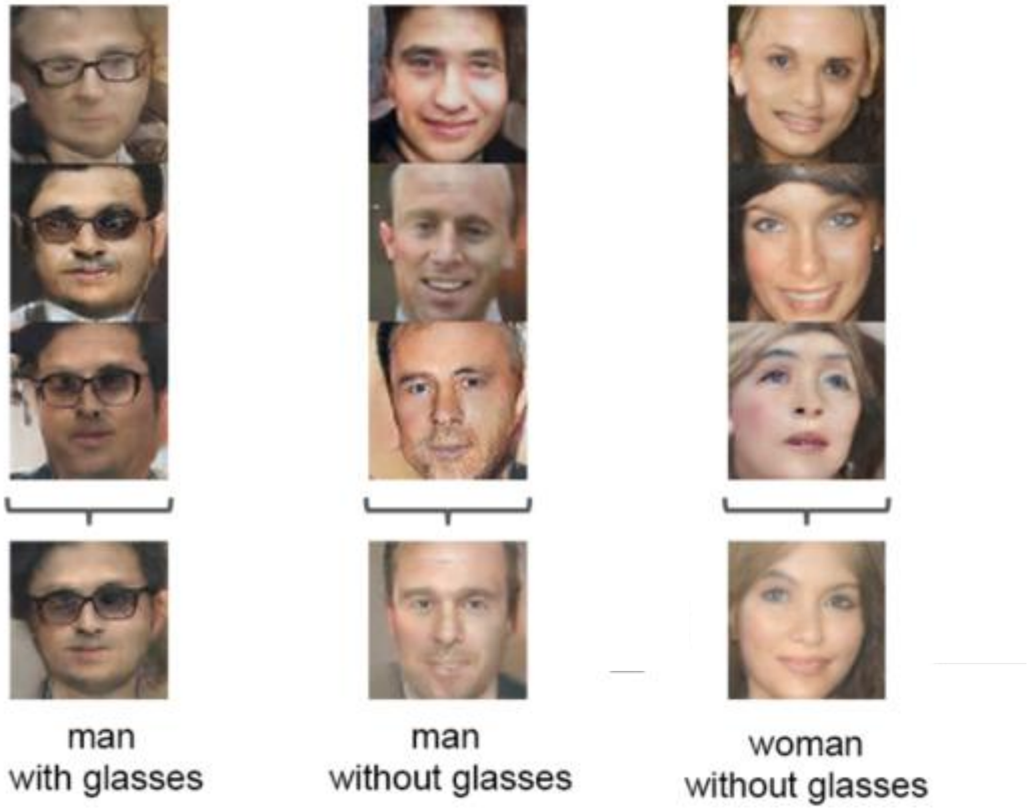
```
X_train_dcgan = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # reshape and rescale
```

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train_dcgan)
dataset = dataset.shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
train_gan(gan, dataset, batch_size, codings_size, n_epochs=50)
```

Generate Images

```
noise = tf.random.normal(shape=[batch_size, codings_size])
generated_images = generator.predict(noise)
```

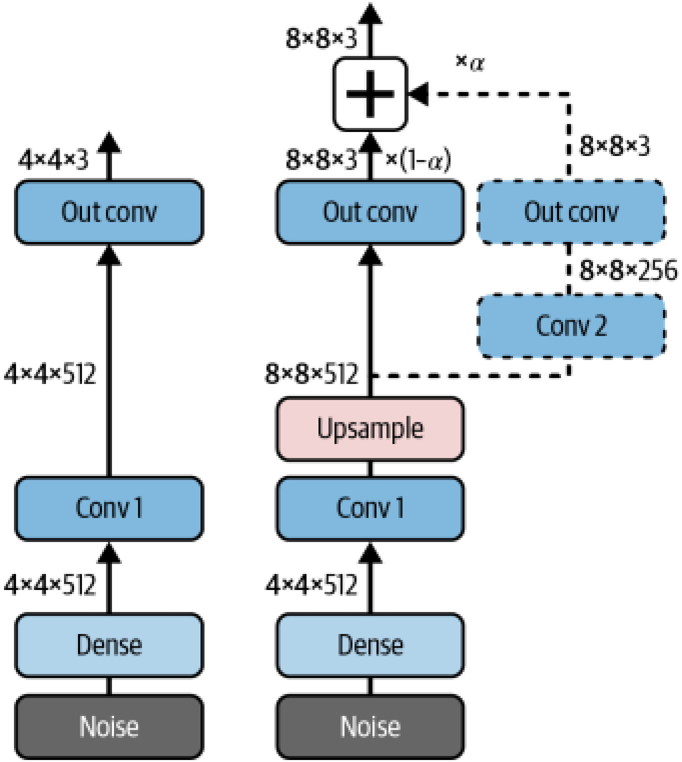
Vector Arithmetic for Visual Concepts



Progressive Growing of GANs

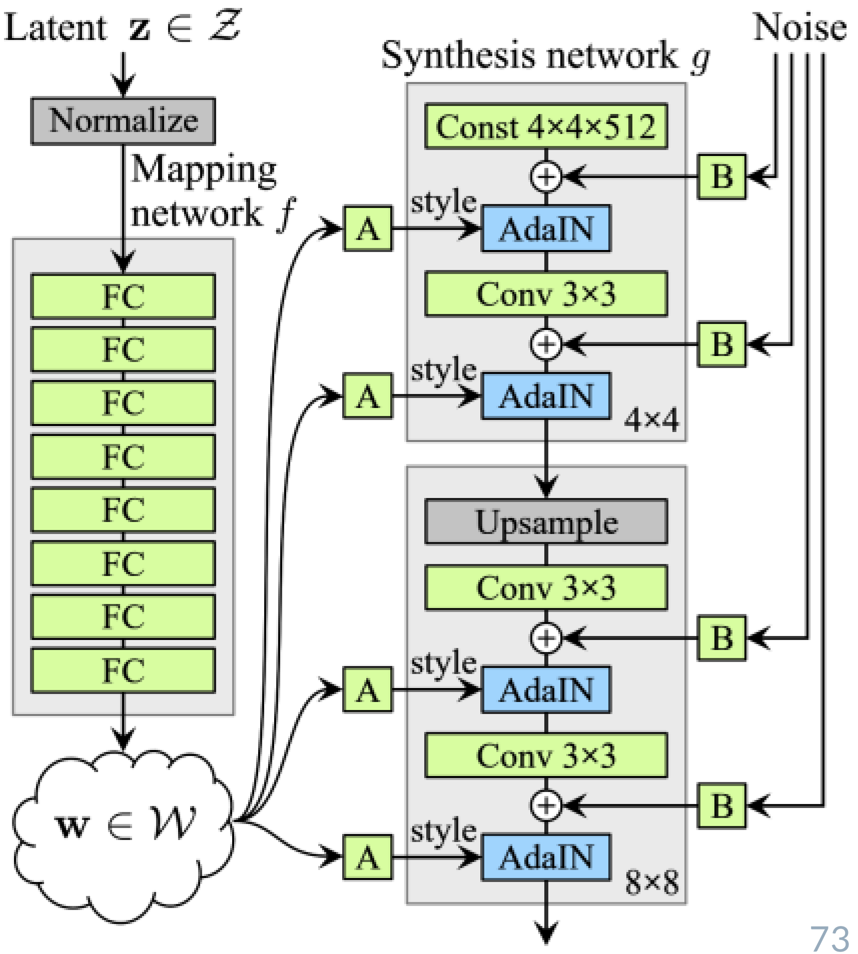
- Nvidia researchers suggested generating small images at beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images.

- This approach resembles greedy layer-wise training of stacked autoencoders.



StyleGAN

- Use *Style transfer* techniques in the generator to ensure that the generated images have the same local structure as the training images, at every scale, greatly improving the quality of the generated images.
- A StyleGAN generator is composed of two networks:
  - *Mapping network* maps the codings to multiple style vectors.
  - *Synthesis network* which is responsible for generating the images.



Style Transfer Using StyleGAN

Ukiyo-e is a genre of Japanese art that flourished from the 17th through 19th centuries.

Style Transfer Using StyleGAN

Style Transfer