Deep Learning

Deep Learning

Lecture: Recap

Ted Scully

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Deep Learning

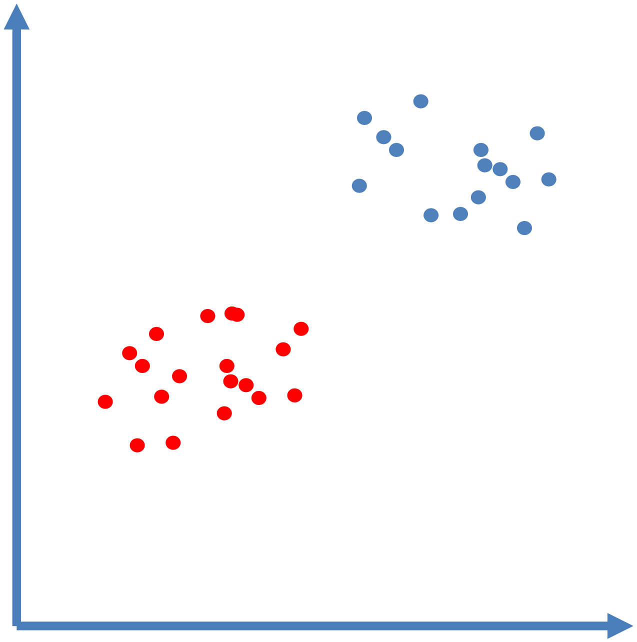
Deep Learning

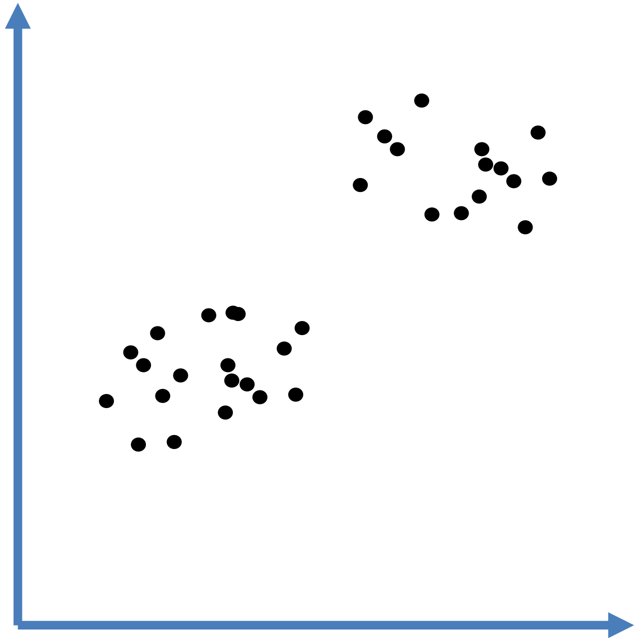
Lecture: Generative Adversarial Networks

Ted Scully

Supervised v Unsupervised ML

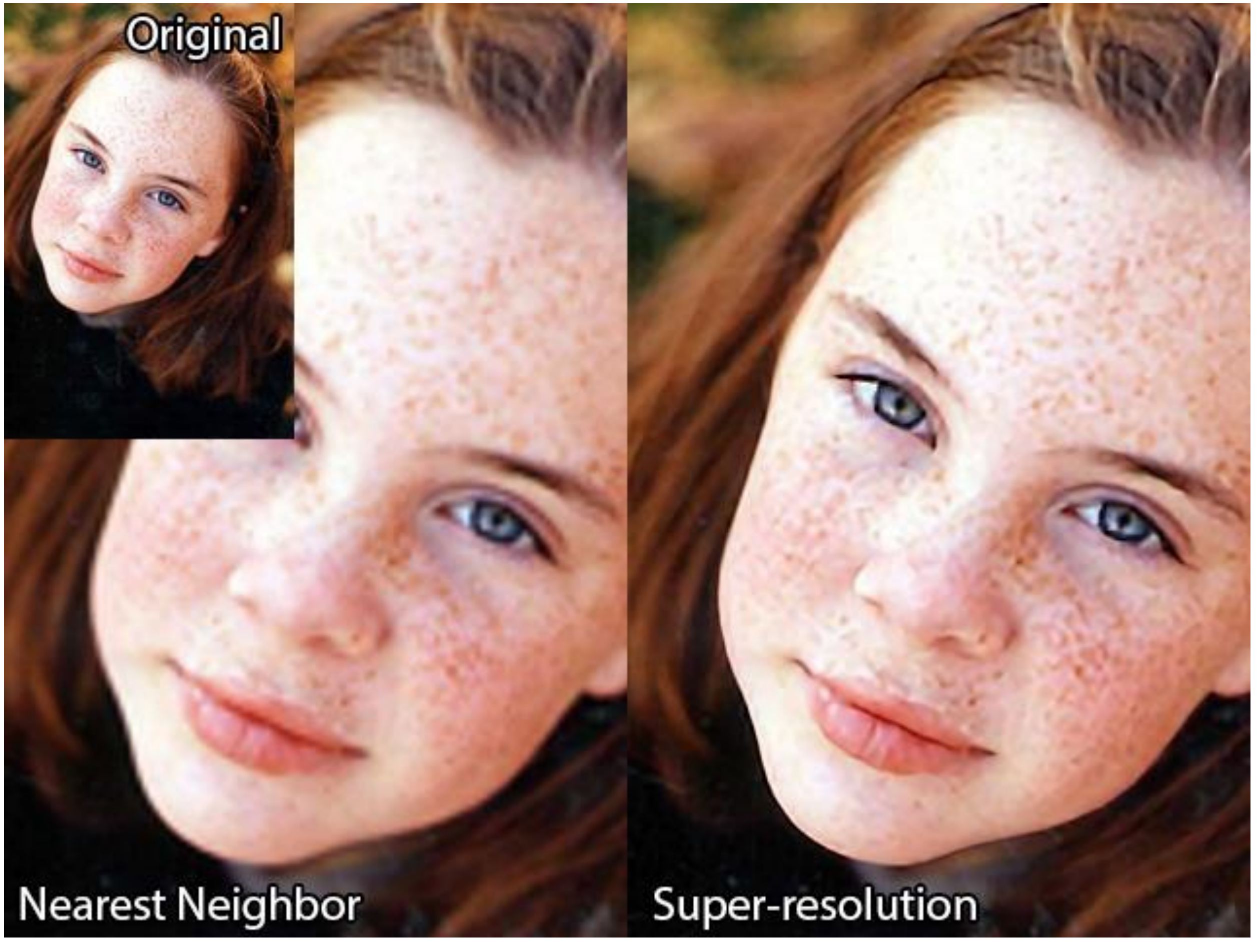
- At this stage you should be comfortable with the distinction between **supervised** and **unsupervised** machine learning.
- Data in supervised machine learning is labelled (feature vectors + labels).
- Data in unsupervised machine learning is not labelled (feature vectors)
- GANs are unsupervised learned (but use a supervised component)





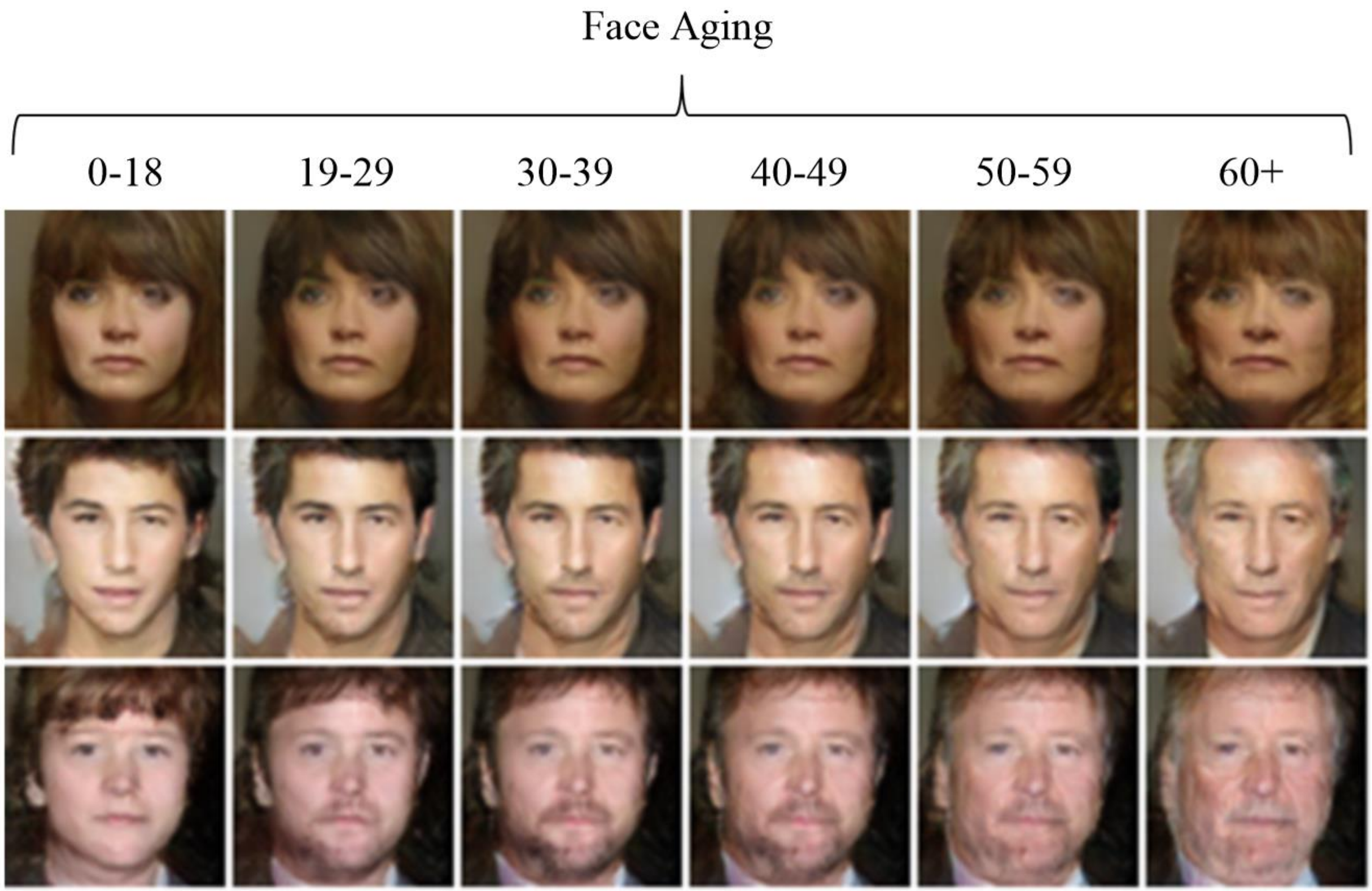
Rationale for Study of Generative Modelling

- Generative modelling can be used for a broad range of problems.
- Much of the high impact applications for GANs are in **image to image translation**.
- For example it has been applied to image super-resolution: In this task, the goal is to take a low resolution image and synthesize a high-resolution equivalent.



https://www.cs.toronto.edu/~duvenaud/courses/csc2541/slides/gan-applications.pdf

- Taking as input an original image and producing an aged version of the image.



https://arxiv.org/pdf/1702.01983.pdf

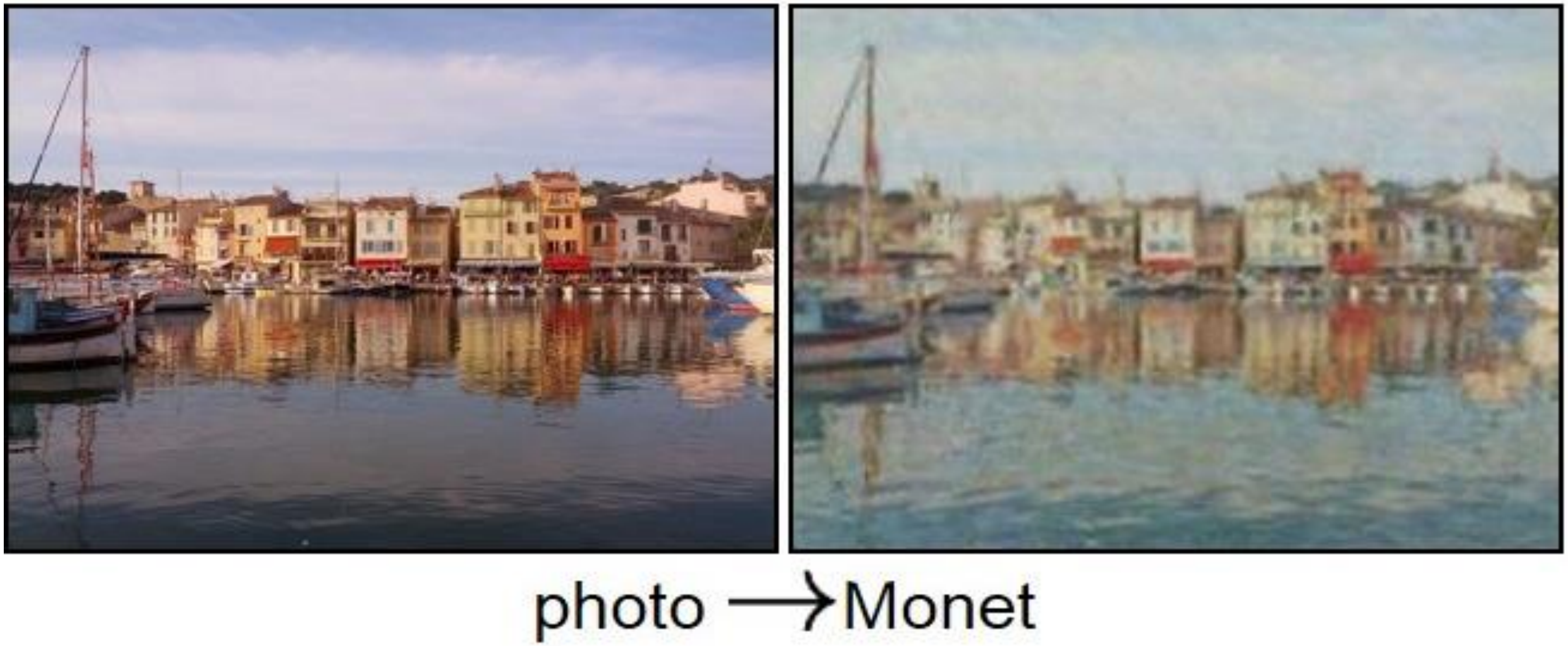
Input Ground truth Output

Input Ground truth Output

https://arxiv.org/pdf/1611.07004.pdf

summer $\longrightarrow$ winter

[Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks - 2017]



[Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks - 2017]

However, GANs despite the high profile image to image applications, GANs have also been applied to more practical tasks as well.

- Adversarial Machine Learning
- Anomaly Detection
- Data Augmentation

GANs

- Learning a generative adversarial network takes the form of a **zero sum game**. In game theory a zero sum game occurs when the gain of one player(s) is equivalent (is balanced out) by the loss of the other player(s).

- A GAN is typically composed of two different models that play a zero sum game against each other.

- One of the models is called the **generator**, the objective of the generator is to generate samples that resemble those that are in the original training distribution.

- The other model is referred to as the **discriminator**. It is used during the training process. The objective of the discriminator is to take a sample and determine if that sample is real or fake.

- The **discriminator** learns using traditional **supervised learning** techniques, dividing inputs into two classes (real or fake).

- The **generator** in contrast is trained to **fool the discriminator**.

Counterfeiter Analogy

- A common analogy used to explain the interaction between a discriminator and a generator is that of the relationship between a **counterfeiter** and the **police**.

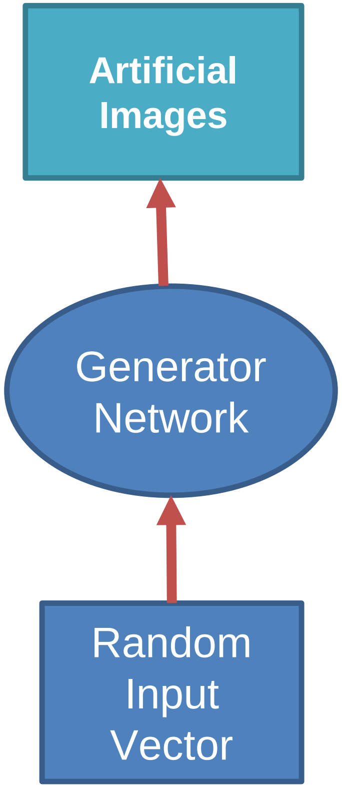
- The <u>generator</u> is the <u>counterfeiter</u>, attempting to generate fake money
- The <u>discriminator</u> is the <u>police</u>, trying to differentiate between legitimate and counterfeit money.

- When the game starts the counterfeiter may make **poor counterfeit notes** and the police will learn to differentiate easily between them.

- However, the counterfeiter will learn to produce **better quality counterfeit notes**. Ultimately the goal of the counterfeiter is to learn to make money that is indistinguishable from genuine money.

- In other words the goal of the generator network is <u>**learn to generate samples from the same distribution as the training data**</u>.

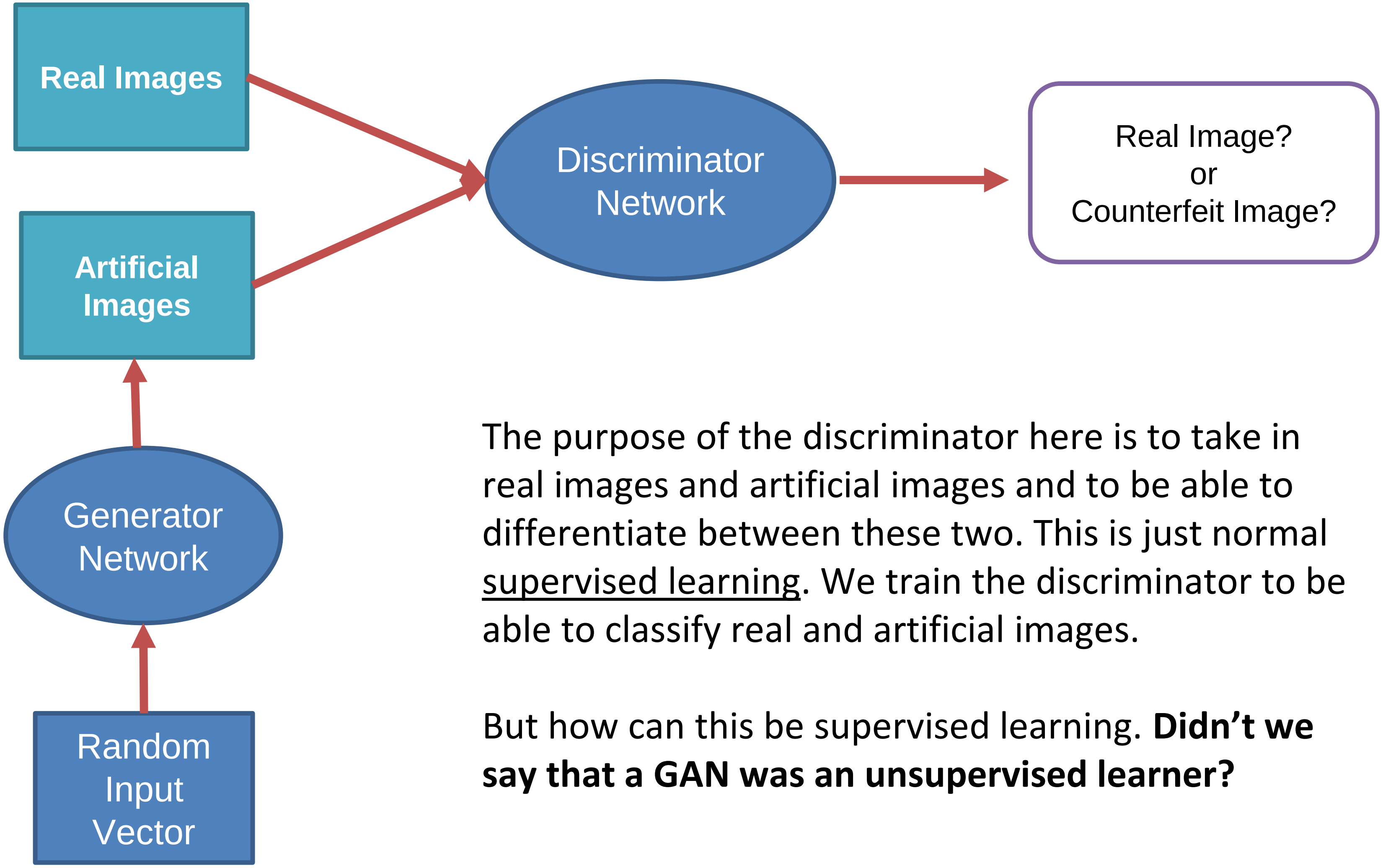
GAN Structure - Generator



- As we mentioned, in a GAN we are actually training two separate networks. We are going to look at GANs in the context of **generating images**.

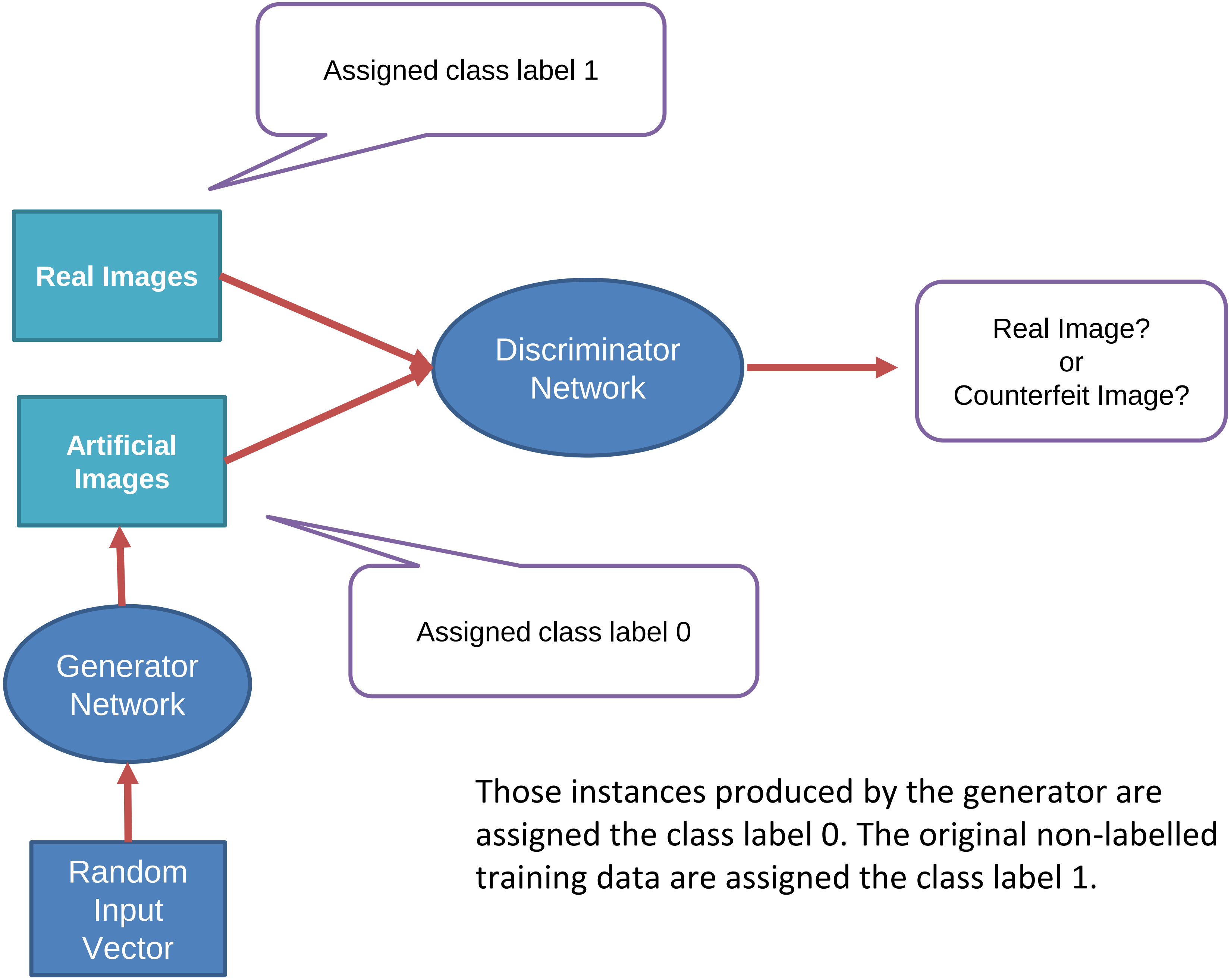
- In the image below we show the high level depiction of the first network, **the generator**. The generator is typically a deep learning neural network or in the case of images a convolutional network.

GAN Structure - Discriminator



The purpose of the discriminator here is to take in real images and artificial images and to be able to differentiate between these two. This is just normal <u>supervised learning</u>. We train the discriminator to be able to classify real and artificial images.

But how can this be supervised learning. **Didn't we say that a GAN was an unsupervised learner?**



Those instances produced by the generator are assigned the class label 0. The original non-labelled training data are assigned the class label 1.

GAN Training Algorithm (High Level)

- For number of training iterations repeat:
  - Generate a mini-batch sample of m artificial images using the generator.
  - Select a mini-batch of m training images.
  - Train the discriminator using these two mini-batches

  - Generate a new batch of artificial images using the generator.
  - Push the output of the generator through the discriminator
  - Update the generator using stochastic gradient descent

The slide depicts the high level steps for training a GAN. You will likely have a number of **questions** after you have after reading this description. How exactly do we generate artificial images using the generator. What is the objective cost function being used? We will look at this a little more formally over the next few slides.

Formal Description of GAN

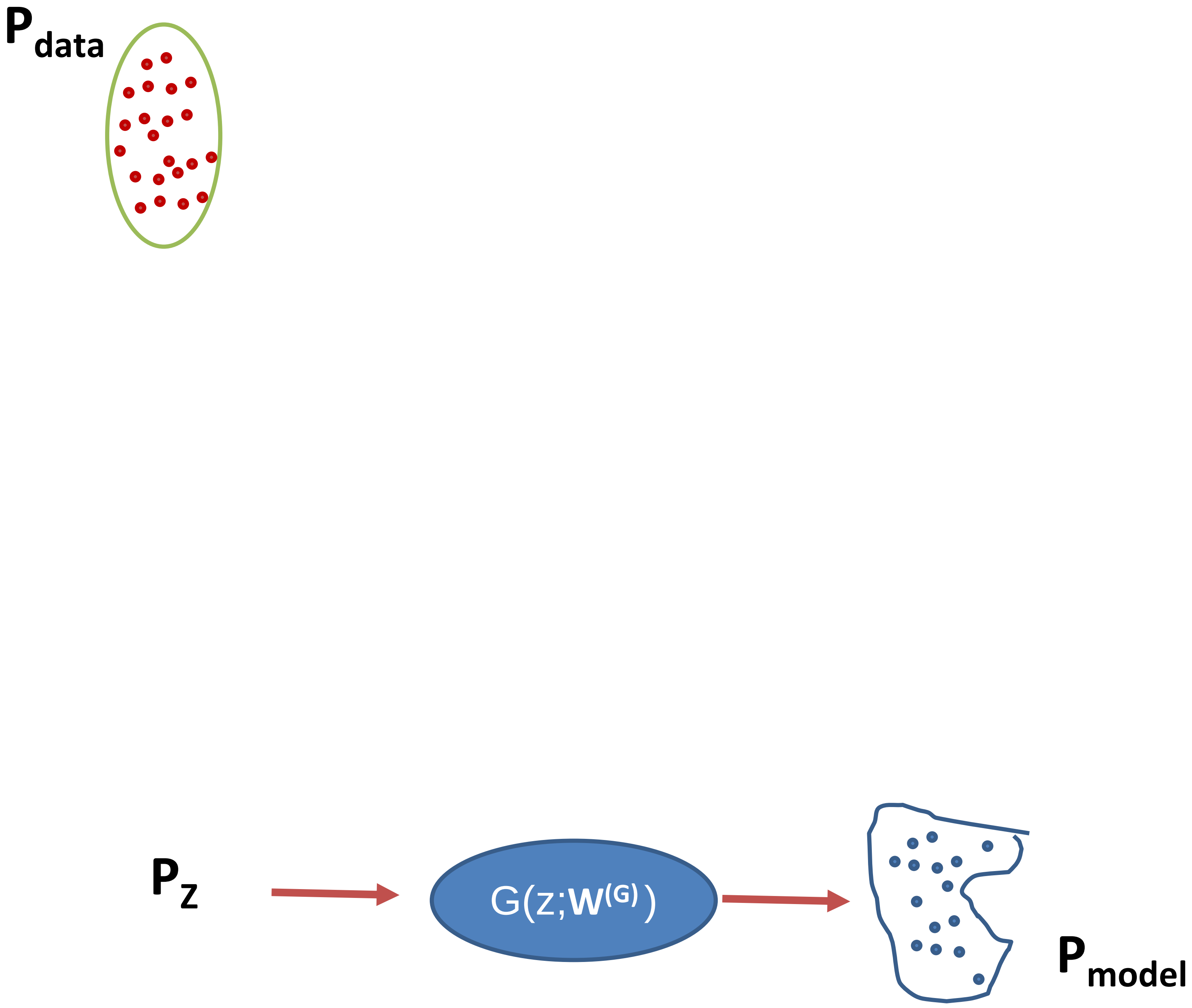
We refer to our training data as **X** that consists of individual data instances x ($x \in X$). We refer to the distribution of X as $\mathbf{P_{data}}$

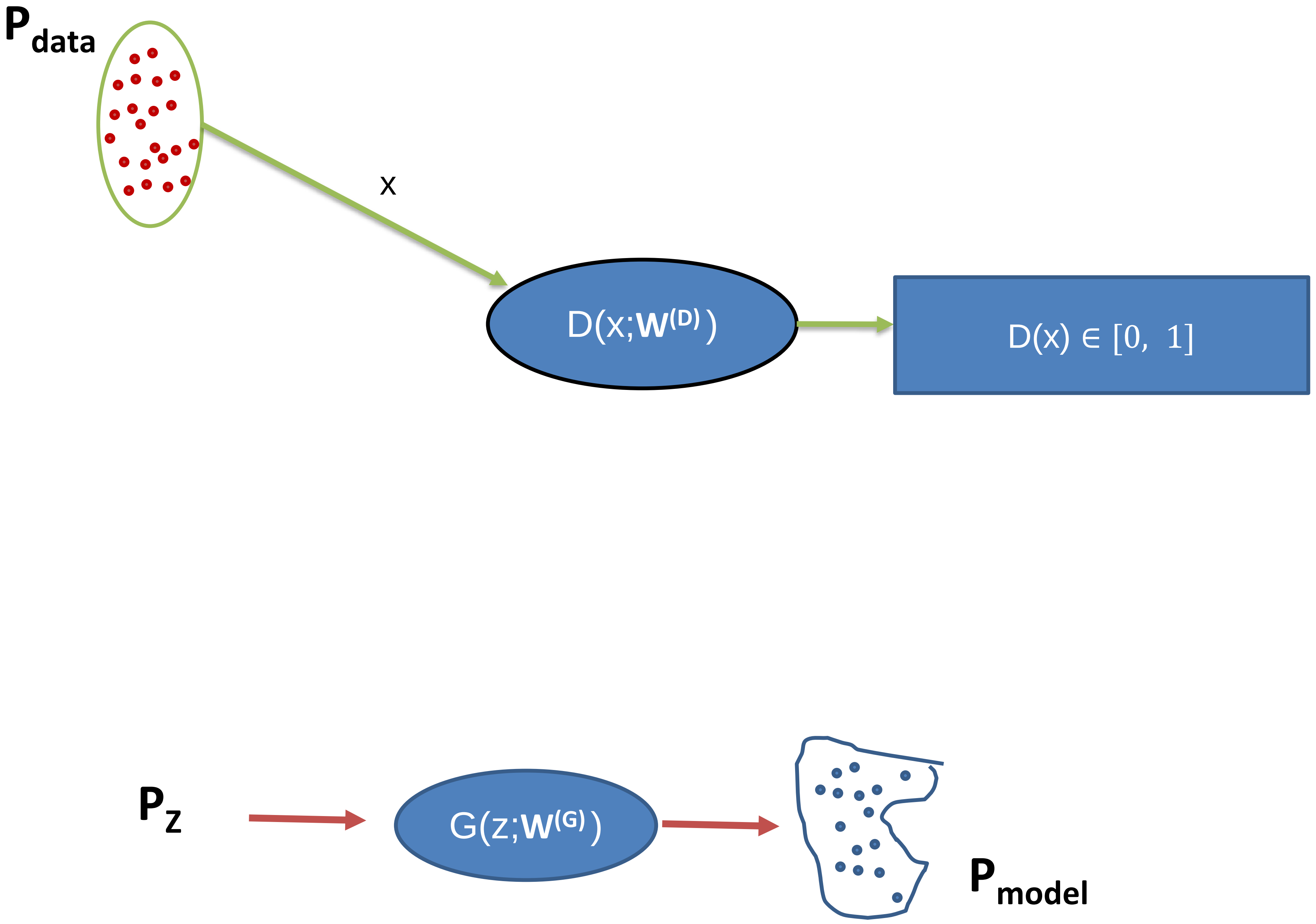
We refer to the discriminator as the function **D** and it's parameters as $\mathbf{W^{(D)}}$

The generator is defined by a function **G** that takes a random vector **z** as input and uses $\mathbf{W^{(G)}}$ as parameters. The random vector z is sampled randomly from a distribution we refer to as $\mathbf{P_Z}$.

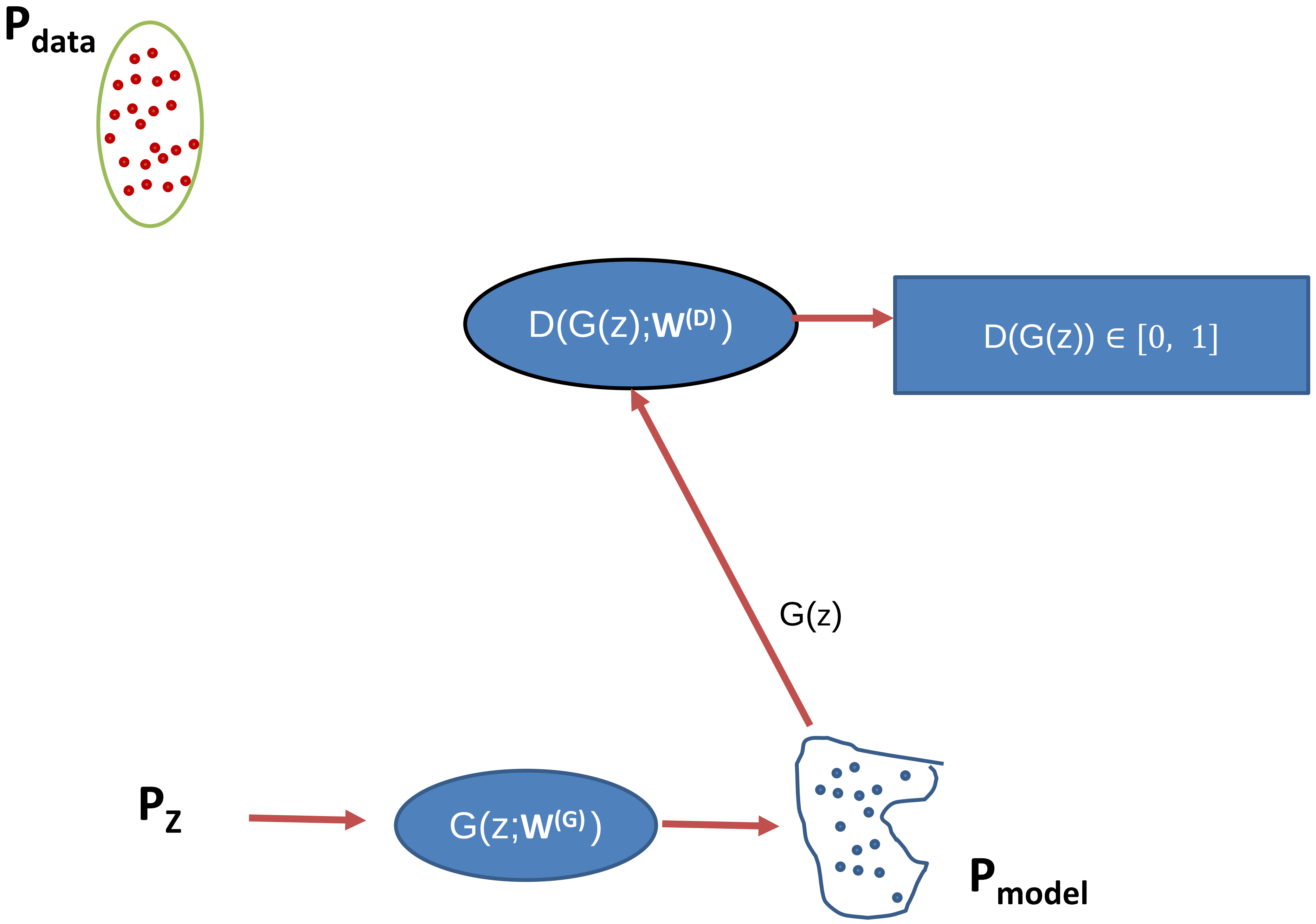
The generator function produces as output **G(z)**. The objective of **G** is to learn appropriate parameters for $\mathbf{W^{(G)}}$ so that the distribution of generated data points (which we will refer to as $\mathbf{P_{model}}$) closely approximates $\mathbf{P_{data}}$.

Notation use is based on notation used by Goodfellow in https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf


P_{data}
P_Z
$G(z;W^{(G)})$
P_{model}

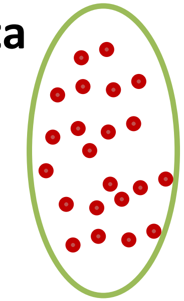


P_{data}
x
$D(x;W^{(D)})$
$D(x) \in [0, 1]$
P_Z
$G(z;W^{(G)})$
P_{model}



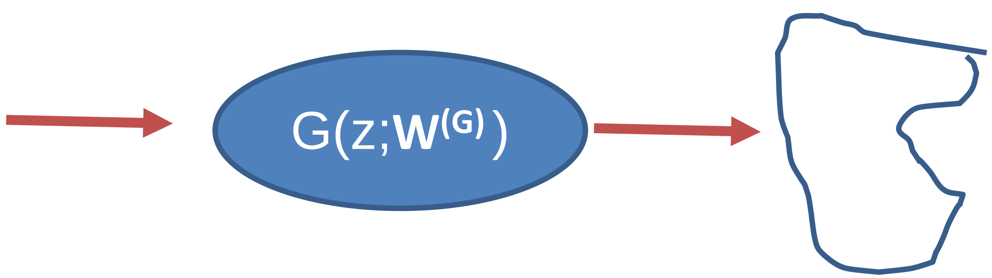
P_{data}
$D(G(z); W^{(D)})$
$D(G(z)) \in [0, 1]$
$G(z)$
P_Z
$G(z; W^{(G)})$
P_{model}

P_{data}



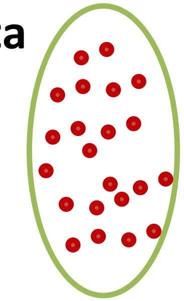
The high level objective of a GAN is that as we train over time the distribution represented by **P_{model}** will begin to approximate the distribution of **P_{data}**.

P_Z



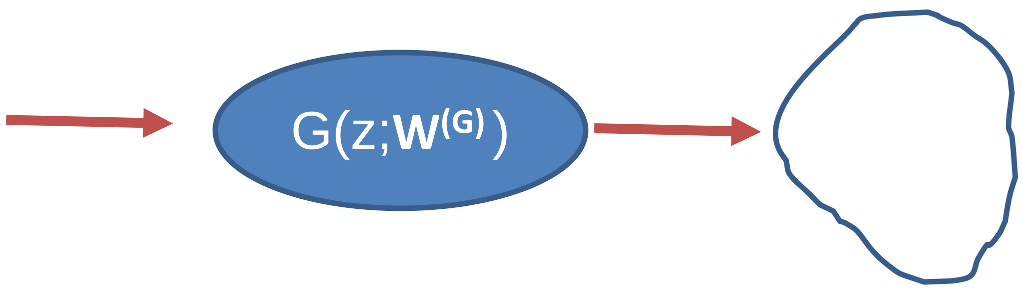
P_{model}

P_{data}

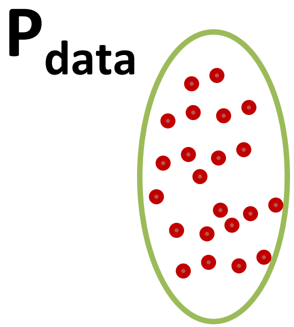


The high level objective of a GAN is that as we train over time the distribution represented by $\mathbf{P_{model}}$ will begin to approximate the distribution of $\mathbf{P_{data}}$.

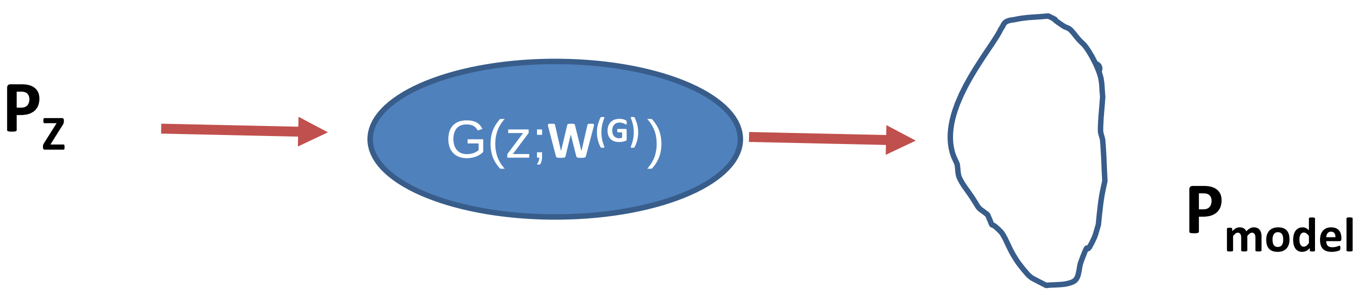
P_Z



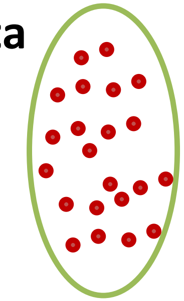
P_{model}



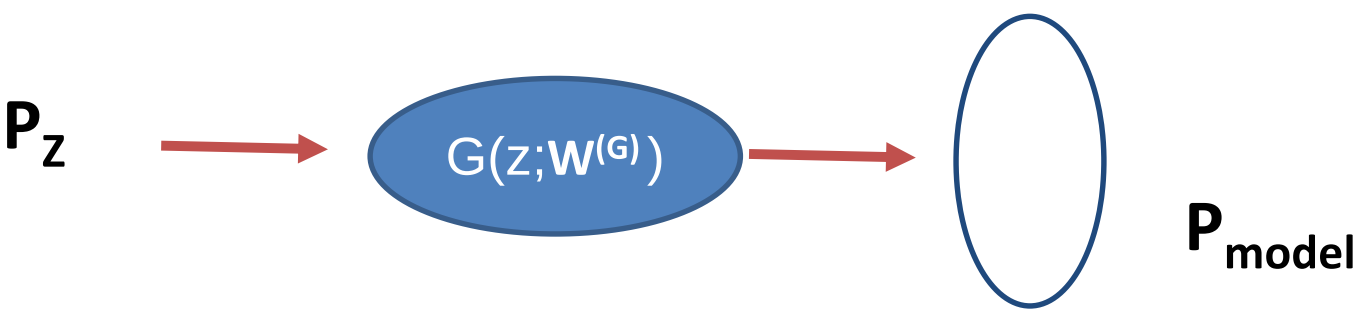
The high level objective of a GAN is that as we train over time the distribution represented by **P_{model}** will begin to approximate the distribution of **P_{data}**.



P_{data}



The high level objective of a GAN is that as we train over time the distribution represented by $\mathbf{P_{model}}$ will begin to approximate the distribution of $\mathbf{P_{data}}$.



Loss Function for Discriminator

The loss function that is used for the discriminator is the minimize the average binary cross entropy loss. In the notation below we represent this for just one instance:

$$\frac{1}{m}\sum_{i=1}^{m}[-\log\left(D\left(x^{i}\right) \right) - \log(1 - D(G\left(z^{i}\right))]$$

Where:

- m is the total number of instances generated from the generator and the total number of instances take from the training set (it is the batch size)

- $D\left(x^{i}\right)$ is the value predicted by the discriminator when a real image x^{i} is inputted into the discriminator.

- $D(G\left(z^{i}\right))$ is the value predicted by the discriminator when an artificial image (generator by the generator) $G\left(z^{i}\right)$ is inputted into the discriminator.

Loss Function for Discriminator

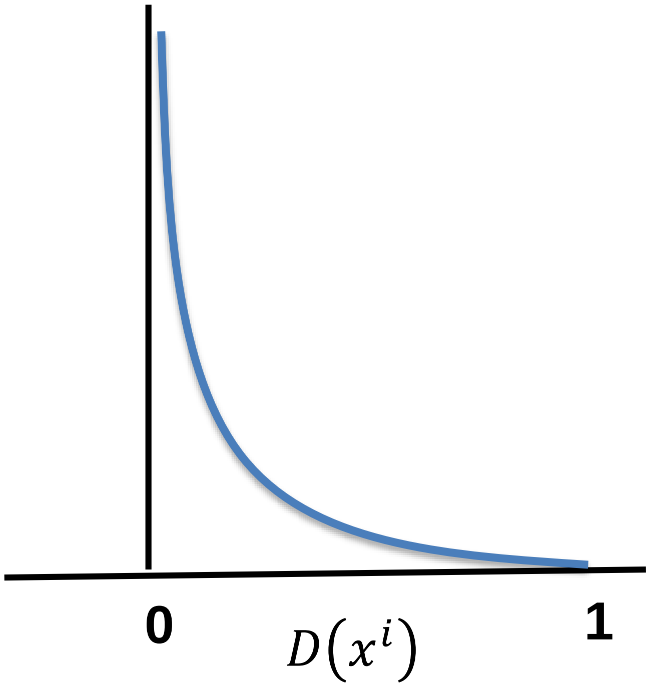
$$\frac{1}{m}\sum_{i=1}^{m}[-\underline{\log(\,D(x^i)\,)} - \log(1 - D(G(z^i)))]$$

So let's consider that the input is from original training data. We know the true label of all these image are **1**.

The loss for when the input is from the original training data is just:

$$\log(D(x^i))$$

Notice the closer $D(x^i)$ gets to 1 then the smaller the value of $\log(D(x^i))$



Loss Function for Discriminator

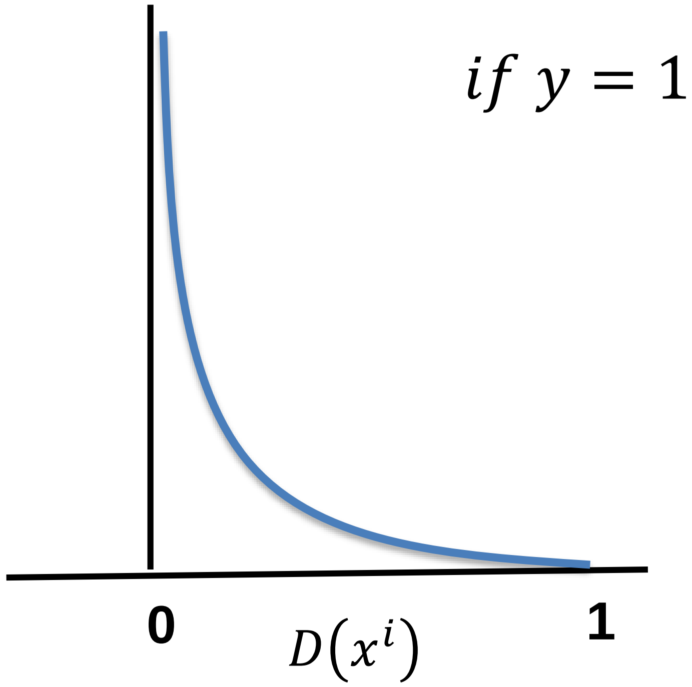
$$\frac{1}{m}\sum_{i=1}^{m}\left[\underline{-\log\left(D\left(x^{i}\right) \right)} - \log(1 - D(G(z^{i})))\right]$$

<u>Good Scenario</u>:

$D(x^i)$ is close to 1. Notice the loss is very small.

<u>Bad Scenario</u>:

$D(x^i)$ is close to 0. Notice the loss is very small.



Loss Function for Discriminator

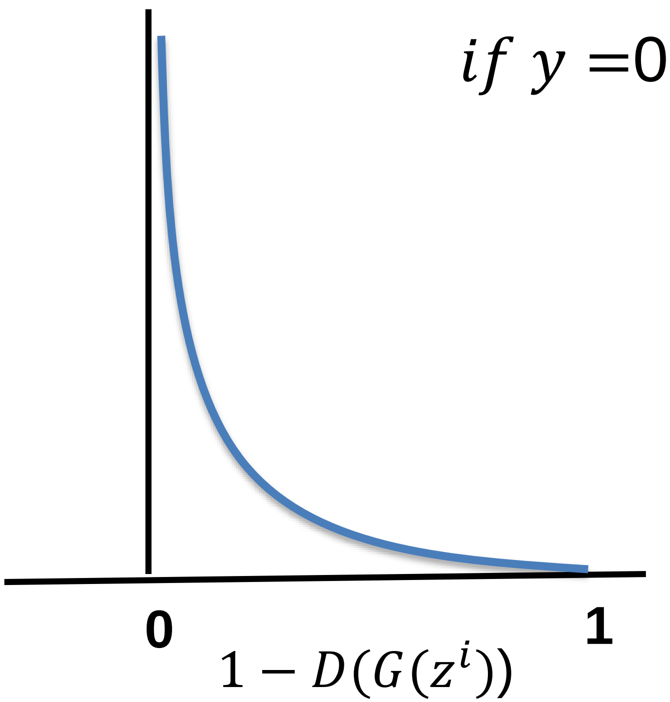
$$\frac{1}{m}\sum_{i=1}^{m}[-\log(\,D(x^{i})\,) - \log(1 - D(G(z^{i}))]$$

So let's consider that the input is from the generator , which has a true label of **0**.

The loss is: $\log\left(1 - D\left(G(z^{i})\right)\right)$

Notice the closer $D\left(G(z^{i})\right)$ gets to 0 the closer the value of $\left(1 - D\left(G(z^{i})\right)\right)$ gets to 1 and the smaller the resulting loss

$if\ y$ =0



Loss Function for Discriminator

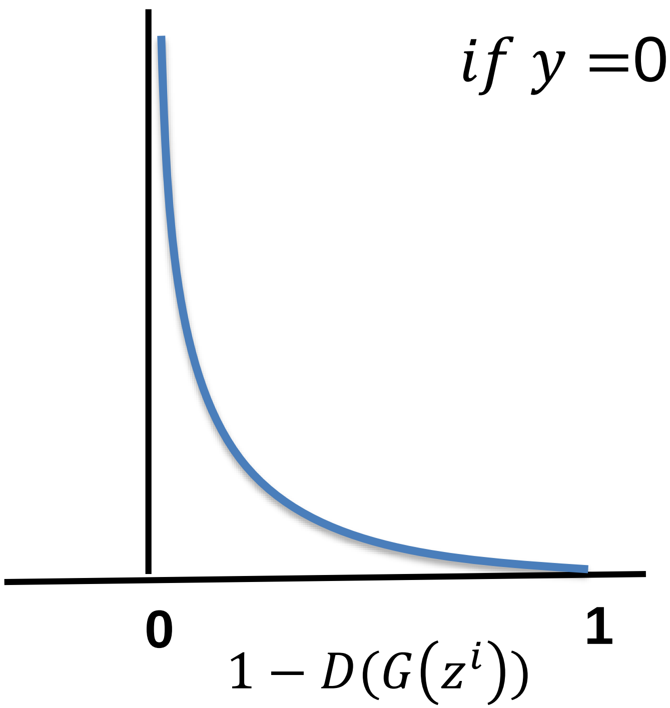
$$\frac{1}{m}\sum_{i=1}^{m}[-\log(D(x^i)) \underline{- \log(1 - D(G(z^i)}))]$$

<u>Good Scenario</u>

$D(G(z^i)$ is close to 0. Which means that 1-$D(G(z^i)$ is close to 1, which as we see means a low loss.

<u>Bad Scenario</u>:

$D(G(z^i)$ is close to 1. Which means that 1-$D(G(z^i)$ is close to 0, which results in a high loss.



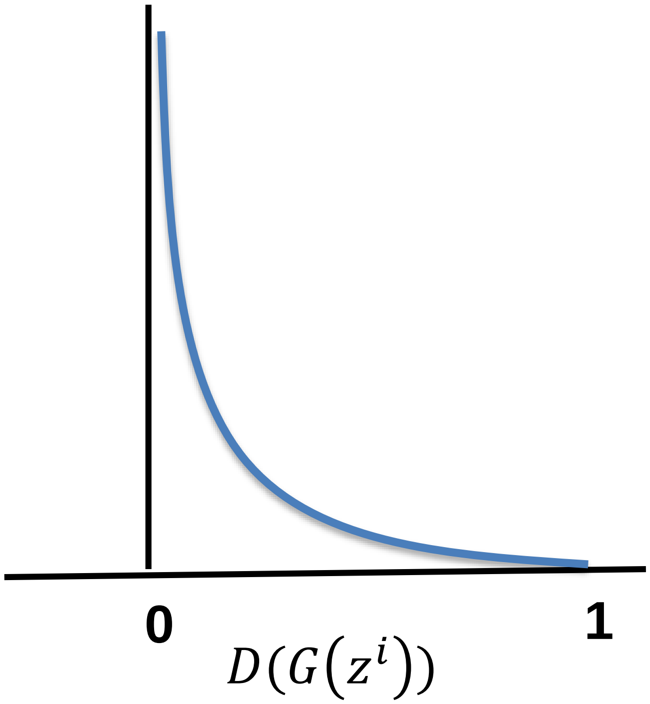
Loss Function for the Generator

The generator attempts to minimize the following loss function (in other words it wants to):

$$\frac{1}{m}\sum_{i=1}^{m} -\log(D(G(z^i)))$$

Remember the generator wants to produce output that will fool the discriminator.

In other words it wants the output of the **<u>discriminator to be as close as possible to 1</u>**.



Loss Function for the Generator

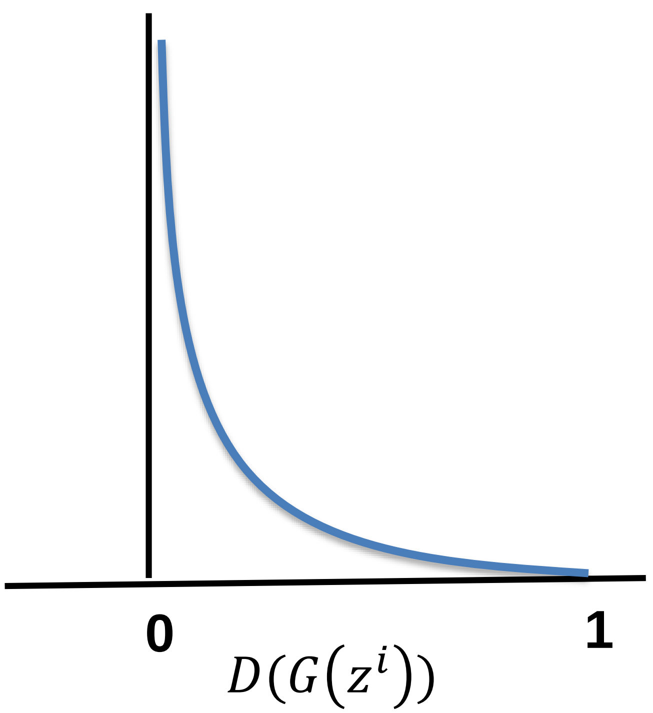
The generator attempts to minimize the following loss function (in other words it wants to):

$$\frac{1}{m}\sum_{i=1}^{m} \underline{-\log(D(G(z^i))}$$

<u>Good Scenario (from the generators perspective):</u>

If $\mathbf{D}(\mathbf{G}(z^i))$ **is close to 1**.

This means the discriminator believes with a high confidence that z^i is real. Therefore, $\log(D(G(z^i))$ is close to 0. (*low loss*)



Loss Function for the Generator

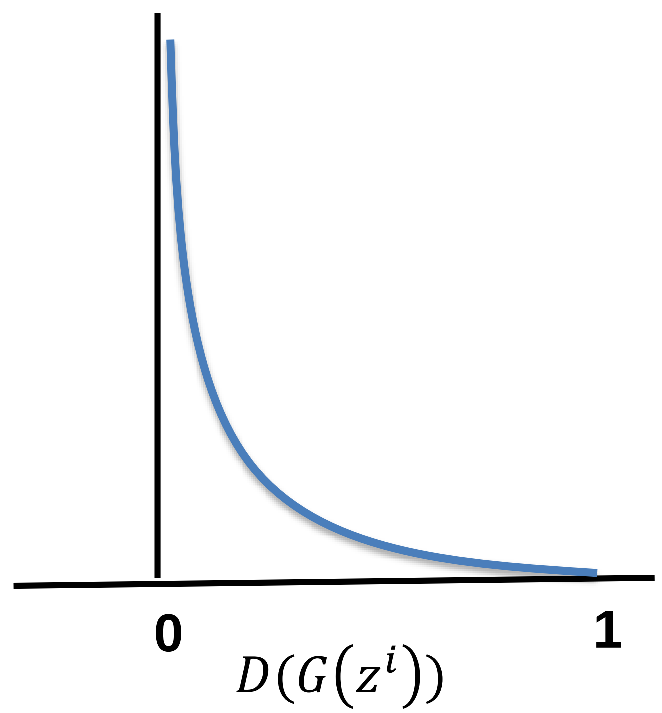
The generator attempts to minimize the following loss function (in other words it wants to):

$$\frac{1}{m}\sum_{i=1}^{m} -\log(D(G(z^i)))$$

Bad Scenario (from the generators perspective):

If $\mathbf{D}(\mathbf{G}(z^i))$ is close to 0.

This means the discriminator believes with a high confidence that z^i is not a real instance. Therefore, $\log(D(G(z^i)))$ is close to very high (*high loss*).



GAN Training Algorithm (High Level)

- For a number of training iterations repeat:
  - Sample a mini-batch of m noise samples $\{z^1, z^2, .. z^m\}$ from P_z
  - Sample a mini-batch of m training images $\{x^1, x^2, .. x^m\}$ from P_{data}.
  - Update the discriminator using stochastic gradient descent using the following loss function:

$$\frac{1}{m}\sum_{i}^{m} -\log\left(D(x^i)\right) - log(1 - D(G(zi)))$$

  - Sample a mini-batch of m noise samples $\{z^1, z^2, .. z^m\}$ from P_z
  - Update the generator with stochastic gradient ascent using the following loss function

$$\frac{1}{m}\sum_{i}^{m} -log(D(G(zi)))$$

GAN Training Algorithm (High Level)

- For a number of training iterations repeat:
  - Sample a min
  - Sample a min
  - Update the di
    following loss function:

The discriminator is trying to ensure that it predicts data items coming from the generator as 0 and training data items as 1. Maximizing this loss function enables that.

$$\frac{1}{m}\sum_{i}^{m} -\log\left(D(x^{i})\right) - log(1 - D(G(zi)))$$

  - Sample a mini-batch of m noise samples $\{z^1, z^2, .. z^m\}$ from P_z
  - Update the generator with stochastic gradient ascent using the following loss function

$$\frac{1}{m}\sum_{i}^{m} -log(D(G(zi)))$$

GAN Training Algorithm (High Level)

Notice that the generator model is only concerned with the discriminator's performance on fake examples.

Remember the generator wants the output of the discriminator to be as close to 1 as possible for the artificial data items it produces. The closer $D(G(zi))$ gets to 1 the lower the resulting log value.

om P_z

om P_{data}.

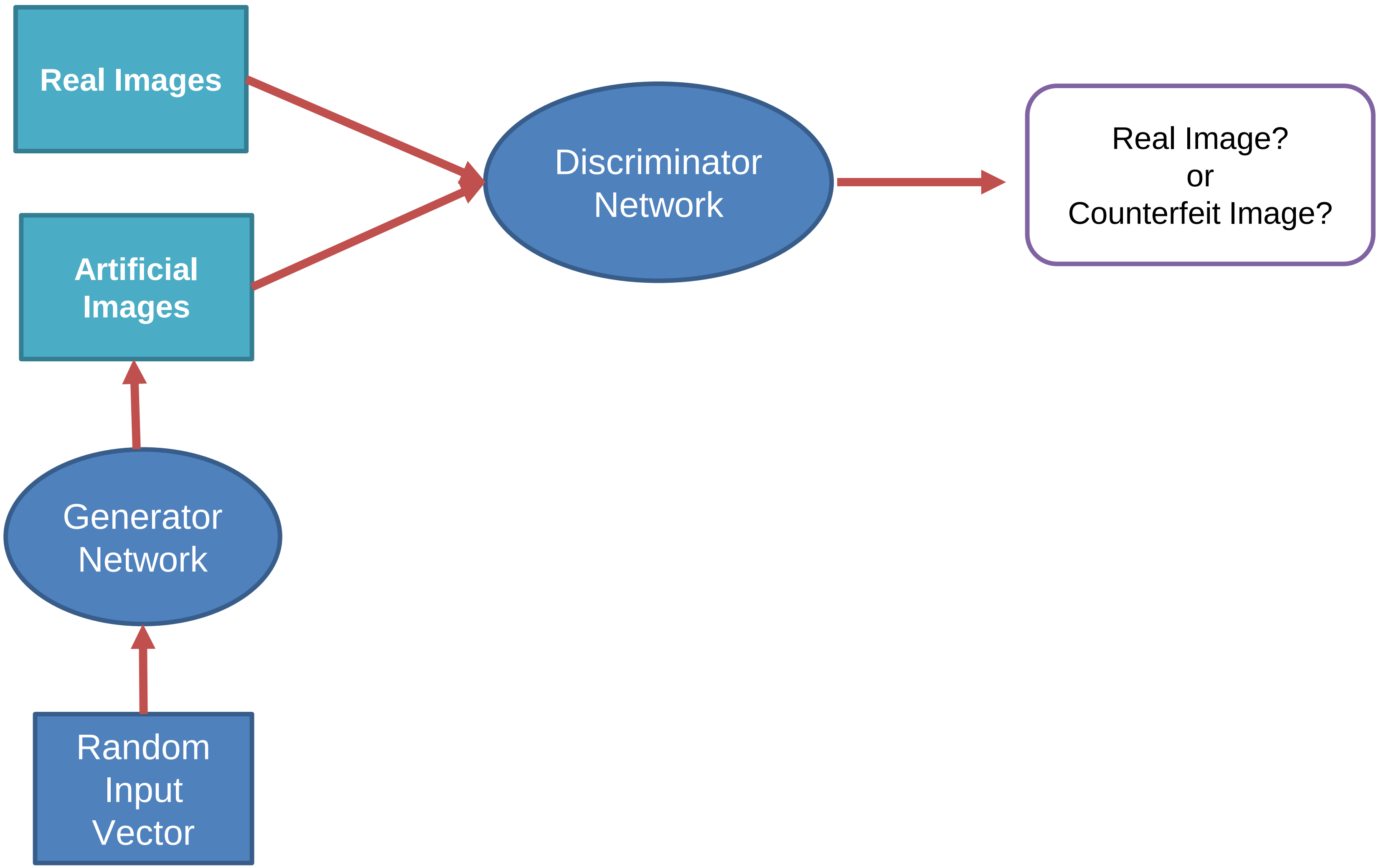
nt using the

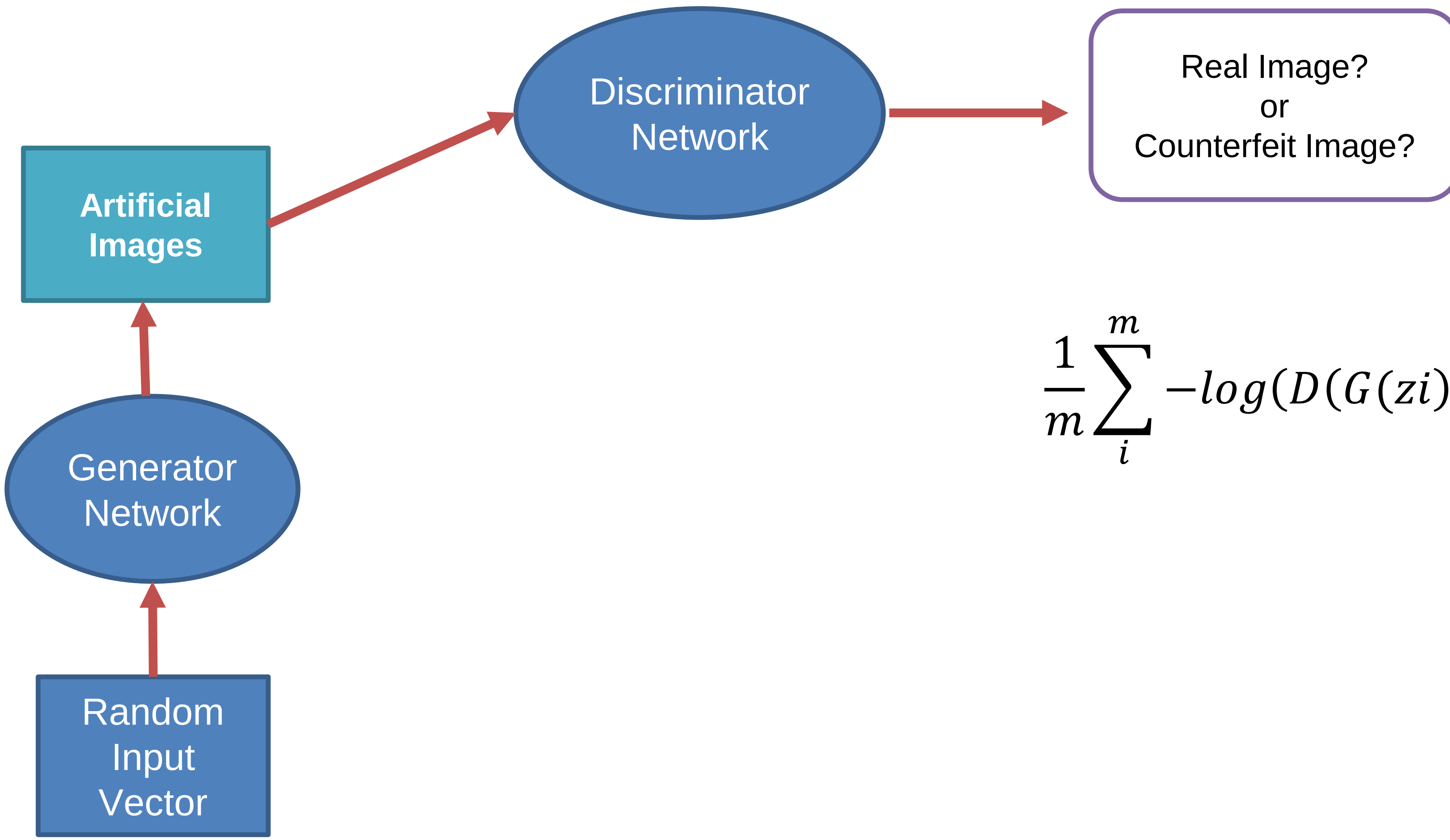
$$\frac{1}{m}\sum_{i}^{m} -\log\left(D\left(x^{i}\right)\right) - log\left(1 - D\left(G\left(zi\right)\right)\right)$$

- Sample a mini- of m noise samples $\{z^1, z^2, .. z^m\}$ from P_z
- Update the gener r with stochastic gradient ascent using the following loss function

$$\frac{1}{m}\sum_{i}^{m} -log\left(D\left(G\left(zi\right)\right)\right)$$

$$\frac{1}{m}\sum_{i}^{m} -\log\big(D(x^i)\big) - log(1 - D(G(zi)))$$





$$\frac{1}{m}\sum_{i}^{m} -log(D(G(zi)))$$

Building a Deep Convolutional GAN

Over the next few slides we will look at implementing a Deep Convolutional Generative Adversarial Network for generating MNIST images.

This is very much the "Hello World" of GANs but it is a excellent way of solidifying your understanding of operational aspect of a GAN model.

The DCGAN was first published by **Alec Radford** in 2015 in paper entitled "**Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks**". The full text is available here.

There are many different variants and architectures. The implementation that follows is based on Radford's original paper (here) and is based on the implementations provided here and here)

Step 1: Building the Discriminator Model

- Our initial step is to build the discriminator model.
- The discriminator model takes a sample image from our dataset (shape (28,28,1)) as input and outputs a binary classification prediction (either the input image is real of fake).
- The following is the architecture of the discriminator model.

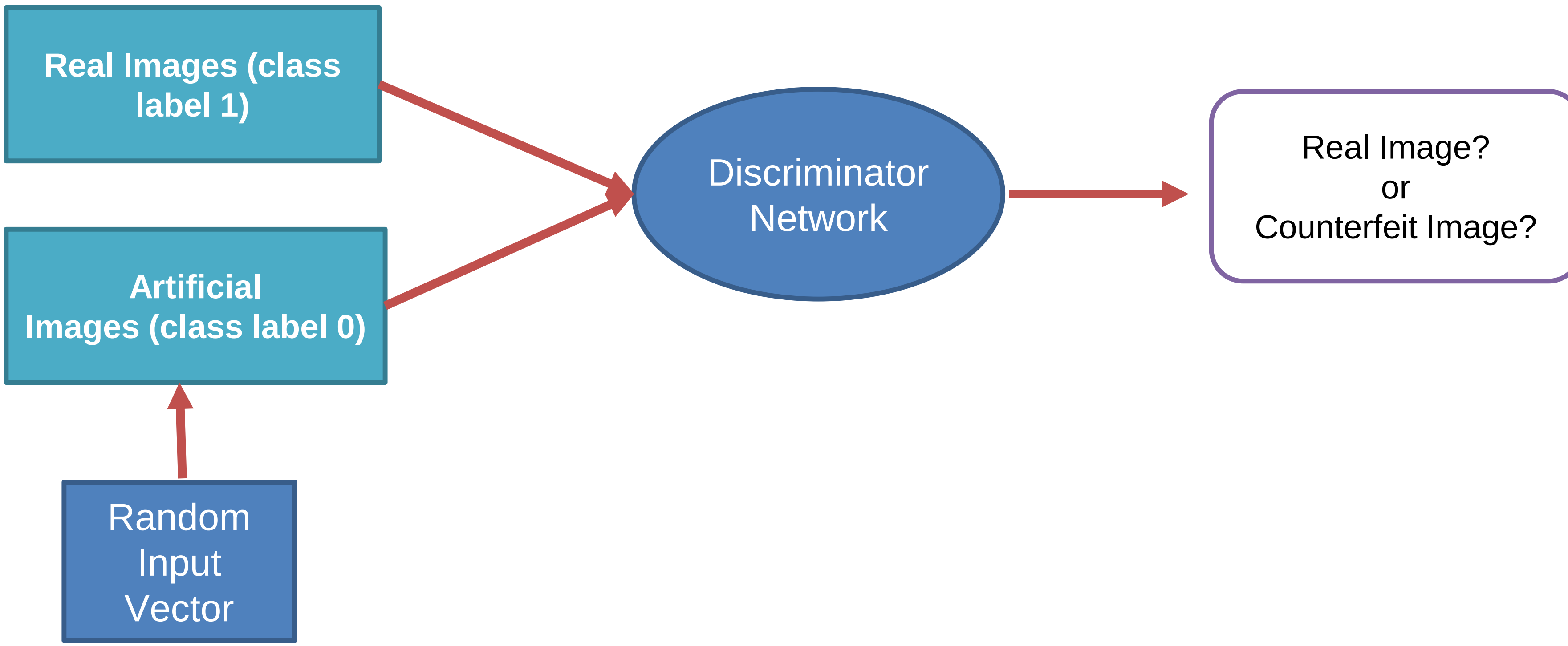
Layer (type)	Output Shape	Param #
conv2d_29 (Conv2D)	(None, 14, 14, 64)	1664
leaky_re_lu_47 (LeakyReLU)	(None, 14, 14, 64)	0
dropout_20 (Dropout)	(None, 14, 14, 64)	0
conv2d_30 (Conv2D)	(None, 7, 7, 128)	204928
leaky_re_lu_48 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_21 (Dropout)	(None, 7, 7, 128)	0
flatten_10 (Flatten)	(None, 6272)	0
dense_19 (Dense)	(None, 1)	6273

Total params: 212,865
Trainable params: 212,865
Non-trainable params: 0

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from matplotlib import pyplot



def discriminator(shape):

    model = tf.keras.models.Sequential()

    model.add(layers.Conv2D(64, (5,5), strides=(2, 2), padding='same', input_shape=shape))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5,5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1, activation='sigmoid'))

    # compile model
    opt = tf.keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
    return model
```

```
import numpy as n
import tensorflow a
from tensorflow.ke
from matplotlib imp


def discriminator(

        model

        model.ad
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))

        model.add(layers.Conv2D(128, (5,5), strides=(2, 2), padding='same'))
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))

        model.add(layers.Flatten())
        model.add(layers.Dense(1, activation='sigmoid'))

        # compile model
        opt = tf.keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
        model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
        return model
```

The selection of the optimization parameter values are based on previous published work. Training GANs is a challenging task and parameter values such as those adopted below have proved relatively stable for ensuring a stable training process. These are empirically derived and are in no means optimal for all problems. In reality they have worked well for a subset of a problem. There is no underpinning hard theoretical justification for the adoption of these values.



So as a reminder we go back to the general architecture of a GAN. We will initially just implement a basic variant without the generator. Then in step 2 we will add the generator.

Step 1: Building the Discriminator Model

- As we know the purpose of the discriminator is to differentiate between artificial and real images.

- We could now start training the discriminator using real examples of MNIST (which we will associate with class label 1) images and randomly generated samples (which we will associate with class label 0).

- The code below will allow us to load real images from the MNIST dataset.
- Train data by default is (60000, 28, 28) but we reshape it to be (60000, 28, 28, 1)

```
def load_real_data():

  (trainX, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
   # MNIST images are 2D, here we add an extra dimension
  X = np.expand_dims(trainX, axis=-1)

  X = X.astype('float32')
  # normalize data
  X = X / 255.0

  return X
```

Step 1: Building the Discriminator Model

- The function below, get_real_samples() takes in the training dataset as an argument and returns a **random batch of images** along with the class label for each image (the class label for each real image is set to one)

```
def get_real_samples(dataset, n_samples):

    # randomly generate indices to extract from training set
    indicies = np.random.randint(0, dataset.shape[0], n_samples)

    # extract batch of slected images
    X = dataset[indicies]

    # set class label to 1
    y = np.ones((n_samples, 1))

    return X, y
```

Step 1: Building the Discriminator Model

- So now that we have real examples, let's just generate some fake examples so we can validate that our discriminator is working properly.
- Please note we will not use this code in our final GAN. This is just used to initially test our discriminator to illustrate it can differentiate between fake and real images.

```
def generate_fake_samples(n_samples):

    X = np.random.rand(28 * 28 * n_samples)

    # reshape into a batch of grayscale images
    X = X.reshape((n_samples, 28, 28, 1))

    # generate class labels for fake images
    y = np.zeros((n_samples, 1))
    return X, y
```

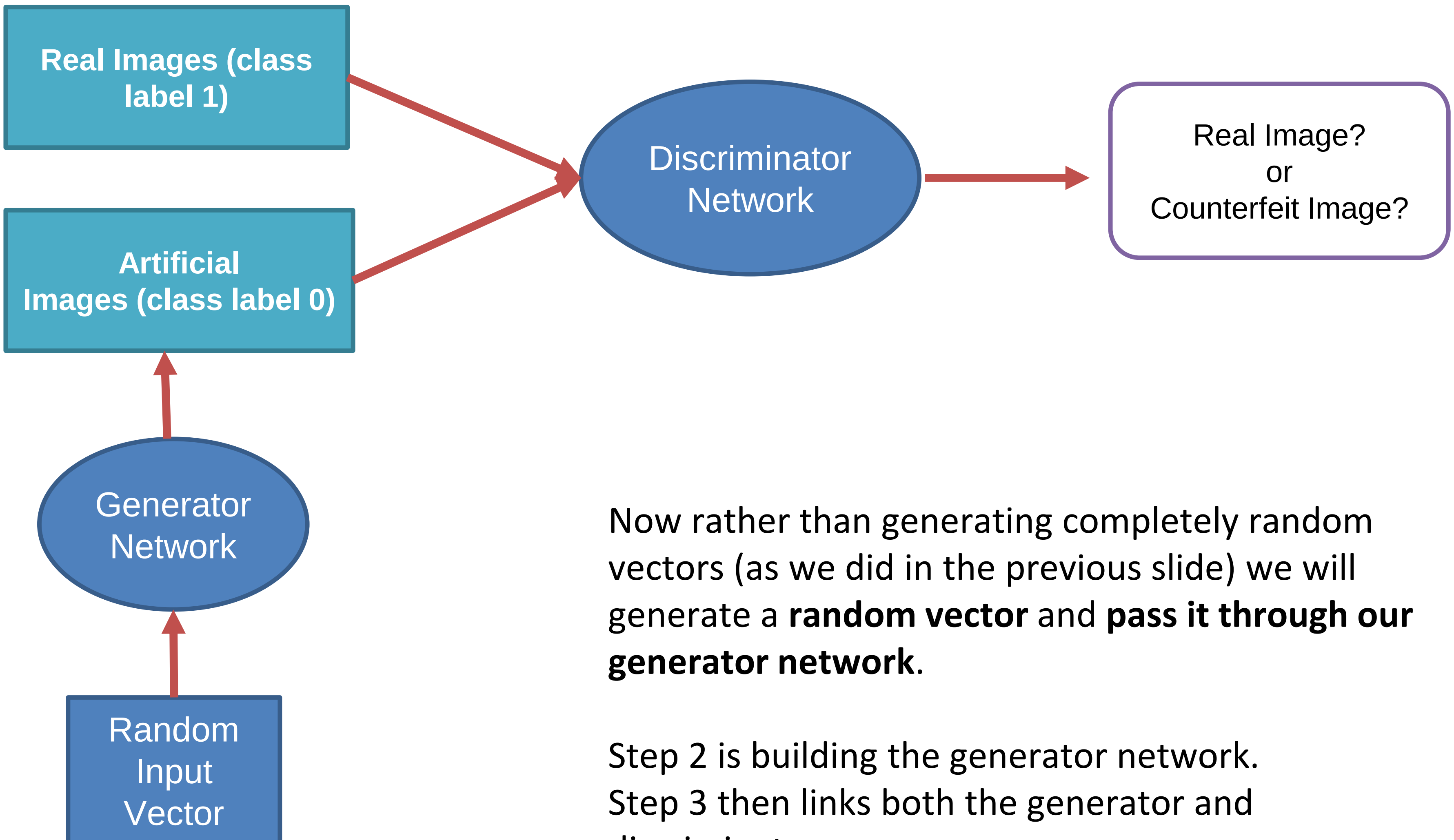
Step 1: Building the Discriminator Model

```
def train_discriminator(model, dataset, n_iter=100, n_batch=256):


  for i in range(n_iter):

      X_real, y_real = fetch_real_samples(dataset, n_batch)
      _, real_acc = model.train_on_batch(X_real, y_real)

      X_fake, y_fake = generate_fake_samples(n_batch)
      _, fake_acc = model.train_on_batch(X_fake, y_fake)

      print("Epoch ",i+1,": Real Accuracy ", real_acc, " Fake Accuracy ",fake_acc)


shape=(28,28,1)
disc_model = discriminator(shape)

# load image data
dataset = load_real_data()

# train model
train_discriminator(disc_model, dataset)
```

Step 1: Building the Discriminator Model

- We can see from the output that the discriminator easily learns to differentiate between real images and fake images.
- The full code for step 1 code can be found here.
- Next we move on to step 2 where we introduce our generator model.

```
Batch  1 : Real Accuracy  0.28125  Fake Accuracy  0.0
Batch  2 : Real Accuracy  0.75  Fake Accuracy  0.609375
Batch  3 : Real Accuracy  0.828125  Fake Accuracy  1.0
Batch  4 : Real Accuracy  0.734375  Fake Accuracy  1.0
Batch  5 : Real Accuracy  0.8125  Fake Accuracy  1.0
Batch  6 : Real Accuracy  0.7890625  Fake Accuracy  1.0
Batch  7 : Real Accuracy  0.8125  Fake Accuracy  1.0
Batch  8 : Real Accuracy  0.8984375  Fake Accuracy  1.0
Batch  9 : Real Accuracy  0.9375  Fake Accuracy  1.0
Batch  10 : Real Accuracy  0.90625  Fake Accuracy  1.0
Batch  11 : Real Accuracy  0.953125  Fake Accuracy  1.0
Batch  12 : Real Accuracy  0.953125  Fake Accuracy  1.0
Batch  13 : Real Accuracy  0.9609375  Fake Accuracy  1.0
Batch  14 : Real Accuracy  0.984375  Fake Accuracy  1.0
Batch  15 : Real Accuracy  0.984375  Fake Accuracy  1.0
Batch  16 : Real Accuracy  0.9921875  Fake Accuracy  1.0
Batch  17 : Real Accuracy  1.0  Fake Accuracy  1.0
Batch  18 : Real Accuracy  1.0  Fake Accuracy  1.0
```



Now rather than generating completely random vectors (as we did in the previous slide) we will generate a **random vector** and **pass it through our generator network**.

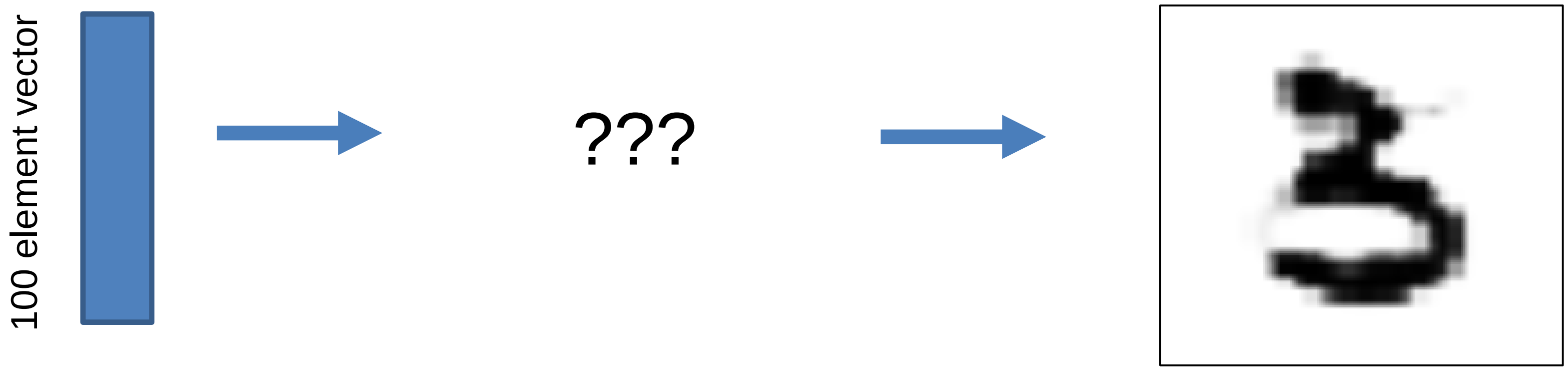
Step 2 is building the generator network.
Step 3 then links both the generator and discriminator

Step 2: Building the Generator Model

- The objective of the generator model (as previously described) is to take in a **randomly generated vector** and generate **new (artificial) plausible images** of handwritten digits.

- Therefore, the <u>input</u> in this case is a 100 element vector of Gaussian random numbers.

- The <u>output</u> is a two-dimensional square grayscale image of 28 × 28 pixels

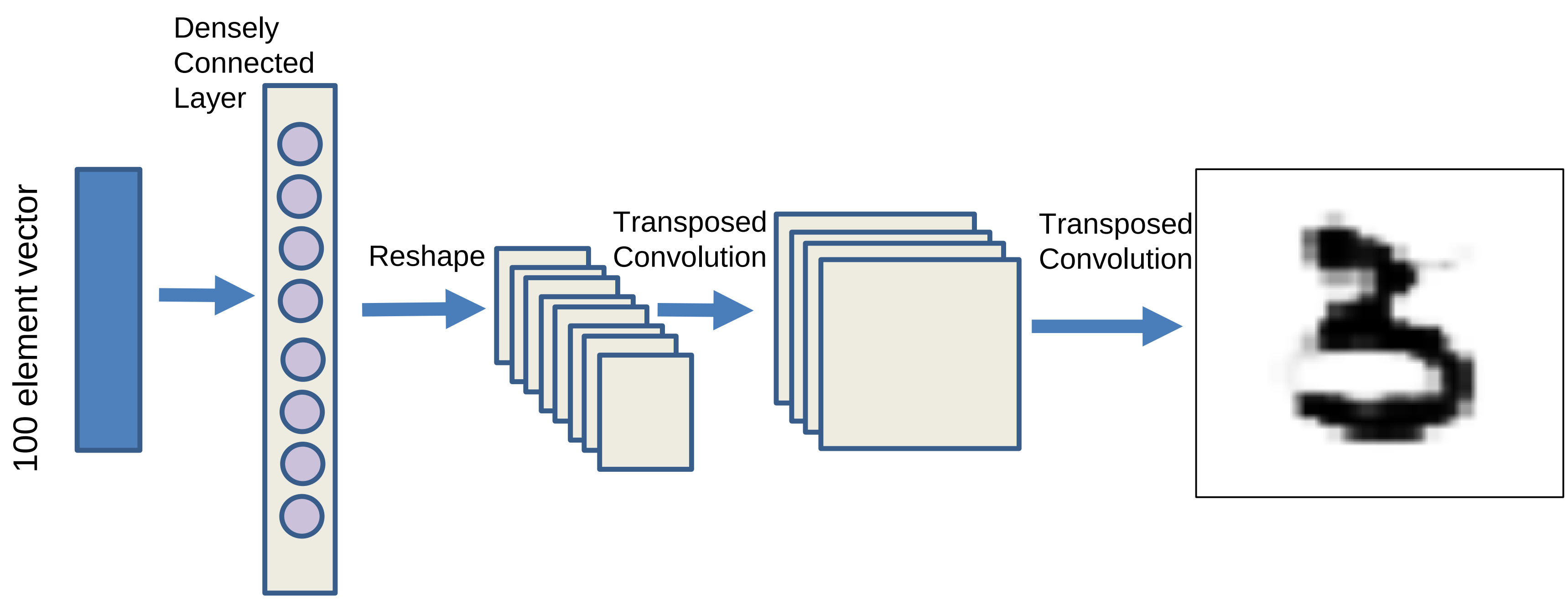


Step 2: Building the Generator Model

- The most common architecture for implementing the **generator** model involves **two main elements**.

- The first is to pass the random vector through a **densely connected network**.

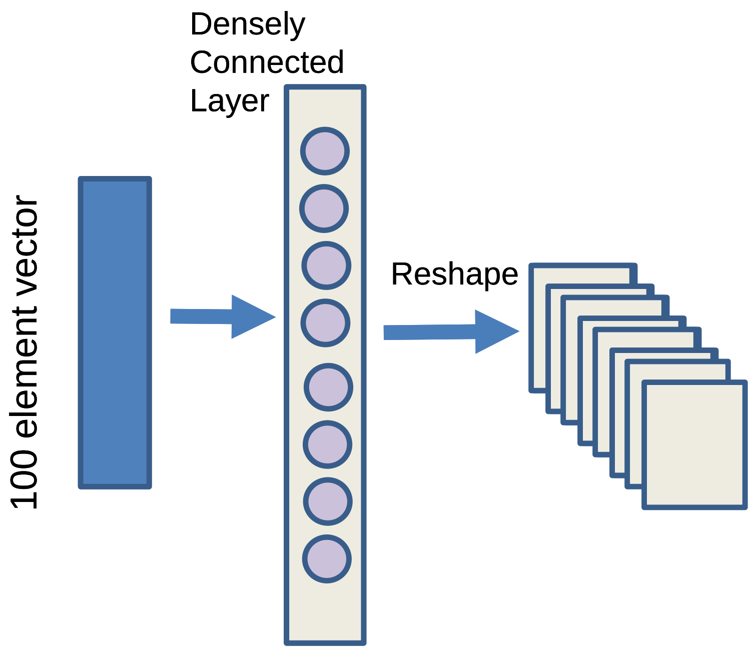
- The second element involves **up-sampling** the low resolution image to a higher resolution image. This process is referred to as a transposed convolution.
- (You might notice that this architecture is like a reverse architecture for a traditional convolutional neural network)



Step 2: Building the Generator Model

- In the first section of our generator we pass our 100 element vector into a densely connected layer. This produces a set of activations which we then reshape.
- In the code we use a densely connected layer with a **12544** (7*7*256) neurons. We then reshape the output activations into a **7*7*256 data structure**.

- You can view the resulting data structure as a **collection of feature maps** (even through no filters have been applied yet) containing multiple low resolution images.



Up-sampling

- You will remember that convolutional neural networks tend to **down-sample** the spatial size of feature maps. For example, **max pooling** will typically half the size of a feature map. Likewise a **convolution operation** without padding will have the same impact.

- The generator model requires the **inverse** of this. It needs to take coarse features and produce a spatially larger more dense set of features. This is referred to as the process of up-sampling.

- A very basic method of up-sampling is called **un-pooling** as shown below.
- You will notice that this there is no learning in the process.

1	2
3	4

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Transposed Convolutions for Up-sampling

- A widely used mechanism of up-sampling used in Generative Adversarial Networks is referred to as **transposed convolutions** (also sometimes referred to as a deconvolutional operation).

Now let's consider a transpose convolution. We look at a **4*4 feature map** and we want to **up-sample** it to a **6*6 image**.
In this example we perform a **3*3 transpose convolution** with stride of 1.

Original

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

1	2	1
1	2	1
1	2	1

Up-sampled Image

Transposed Convolutions for Up-sampling

Now let's consider a transpose convolution. We look at a **4*4 feature map** and we want to **up-sample** it to a **6*6 image**.

In this example we perform a **3*3 transpose convolution** with stride of 1.

Original

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

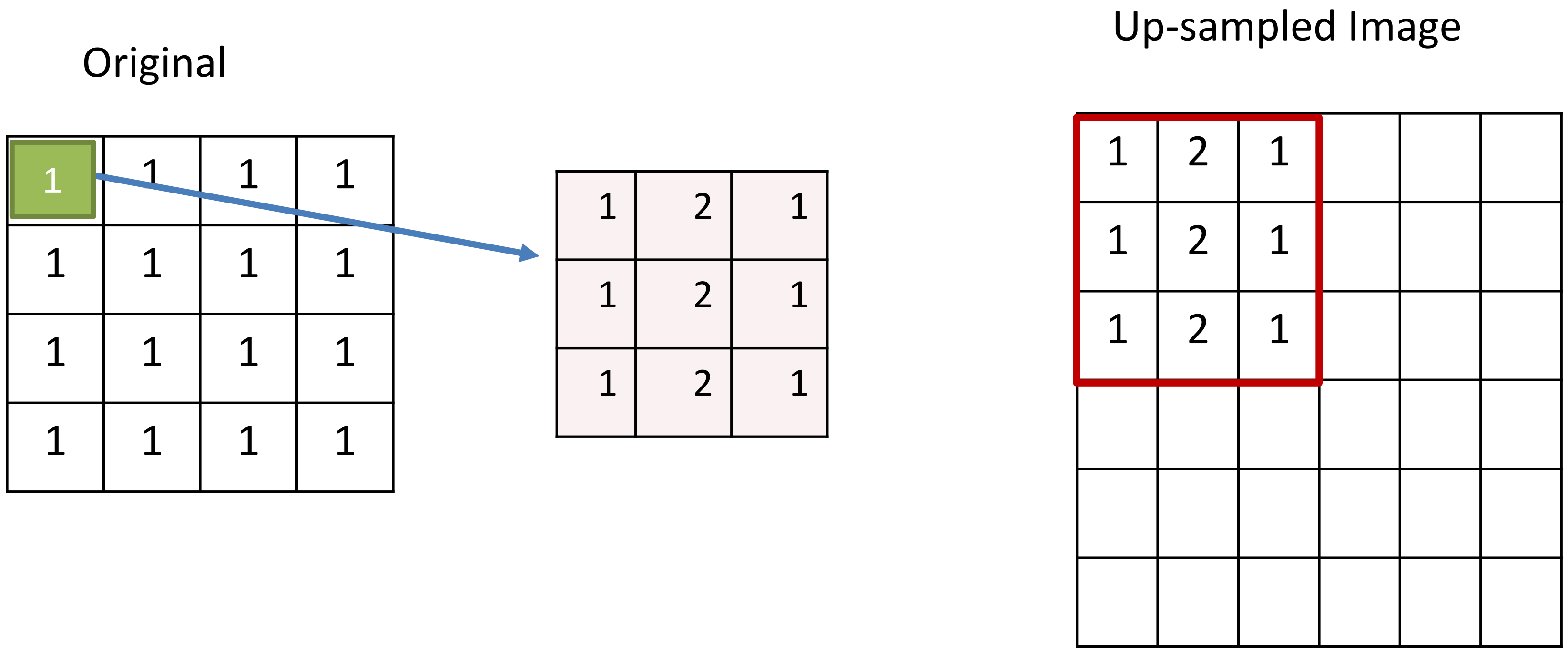
1	2	1
1	2	1
1	2	1

Up-sampled Image

We take the **first value** of the incoming matrix 1 (highlighted in green below).

We then **multiply** it by the each separate **values in the filter**. The resulting 3*3 matrix is then stored as the upper left hand portion of the image (highlighted in red).

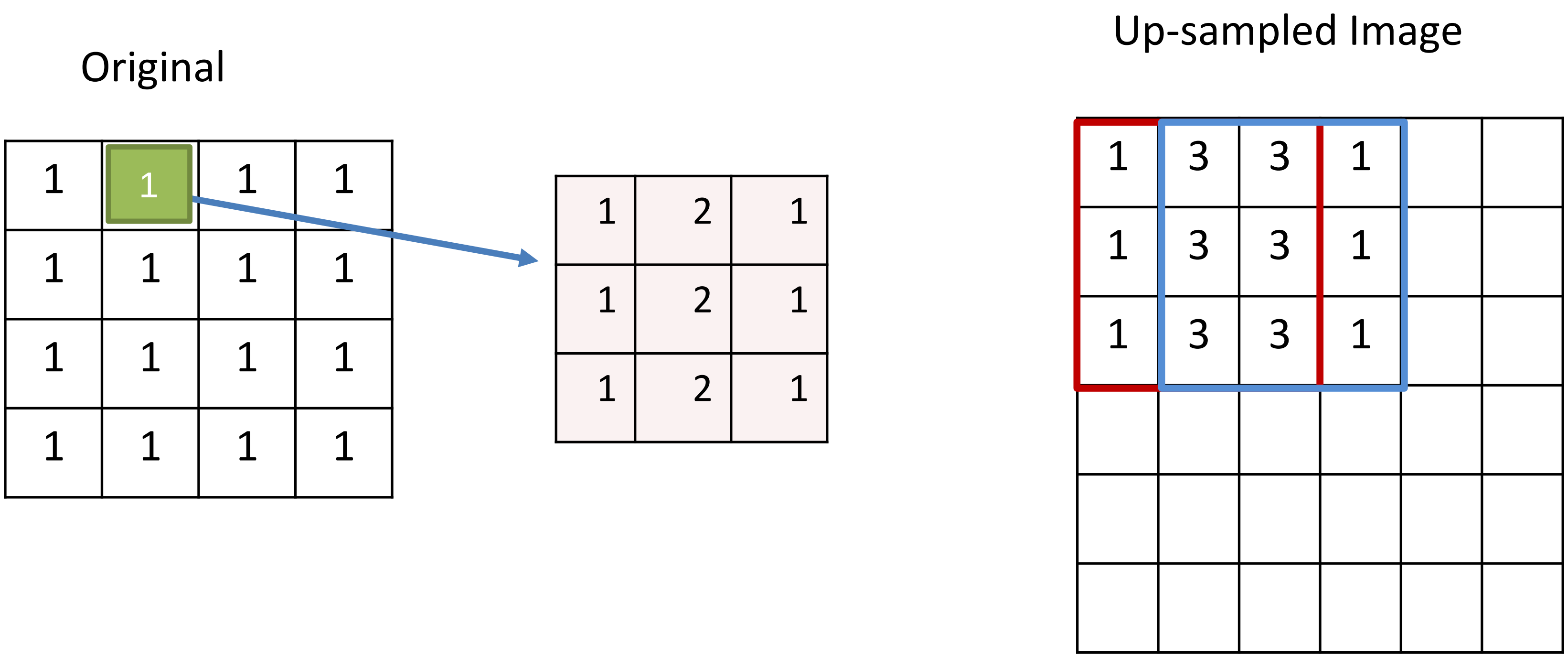
We then repeat this process for each value in the original matrix.



We take the second value of the incoming matrix 1 (highlighted in green below).

We then multiply it by the each separate value in the filter. The resulting 3*3 matrix is then stored as the upper left hand portion of the image (one digit to the right of the last position). This is highlighted in blue below.

Where **overlapping** values occur they are **added together**.



We then repeat this process for each value in the original matrix.

Original

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

1	2	1
1	2	1
1	2	1

Up-sampled Image

1	3	4	3	1	
1	3	4	3	1	
1	3	4	3	1	

We then repeat this process for each value in the original matrix.

Original

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

1	2	1
1	2	1
1	2	1

Up-sampled Image

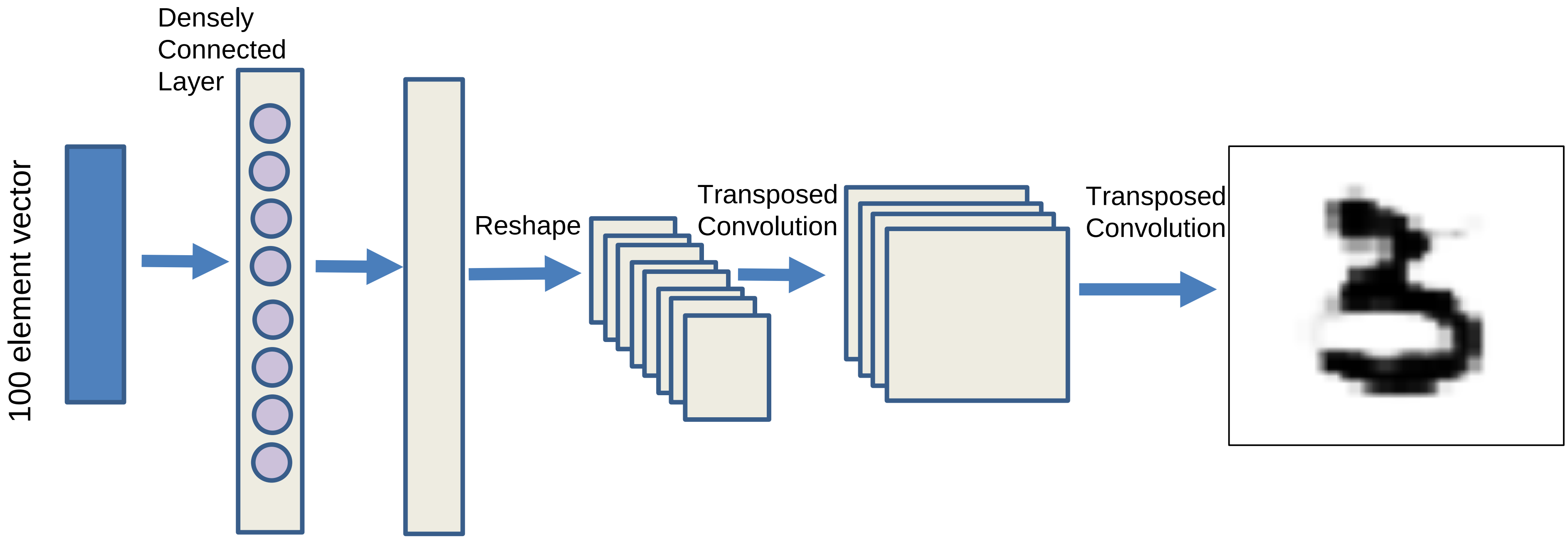
1	2	4	4	3	1
1	2	4	4	3	1
1	2	4	4	3	1

tf.keras.layers.Conv2DTranspose

- TensorFlows tf.keras provides a Conv2DTranspose implementation with the following main arguments.

- **filters**: Integer value that specifies the number of output filters in the convolution.
- **kernel_size**
- **strides**
- **padding**: one of "valid" or "same"

- Notice as we get deeper into our network we decrease the number of filters until in the final layer we just use a single filter.



```
def generator(noise_dim):

    model = tf.keras.models.Sequential()

    # densely connected layer
    model.add(layers.Dense(7*7*256, input_shape= noise_dim))
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same'))
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', activation='sigmoid'))
    return model
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 12544)             12556544
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 12544)             0
_________________________________________________________________
reshape (Reshape)            (None, 7, 7, 256)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 7, 7, 128)         819328
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 64)        204864
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 1)         1601
=================================================================
Total params: 13,582,337
Trainable params: 13,582,337
Non-trainable params: 0
```

Step 2: Building the Generator Model

- Push randomly generated data through our generator model and visualize it.
- In our example we will set the **noise_dim to 100** and the **batch_size to 256**. Therefore, in the first line we just generate **25,600** numbers randomly from a normal distribution and we next reshape this into an **256** rows, where each row is 100 elements in length.

```
def generate_fake_samples(g_model, noise_dim, n_samples):

    # generate random data points of size noise_dim
    x = np.random.randn(n_samples * noise_dim)

    # reshape into a batch of inputs for the network
    x_data = x.reshape(n_samples, noise_dim)

    # push data through the generator model
    x = g_model.predict(x_data)

    # set class labels to 0 for artificial data
    y = np.zeros((n_samples, 1))

    return x, y
```

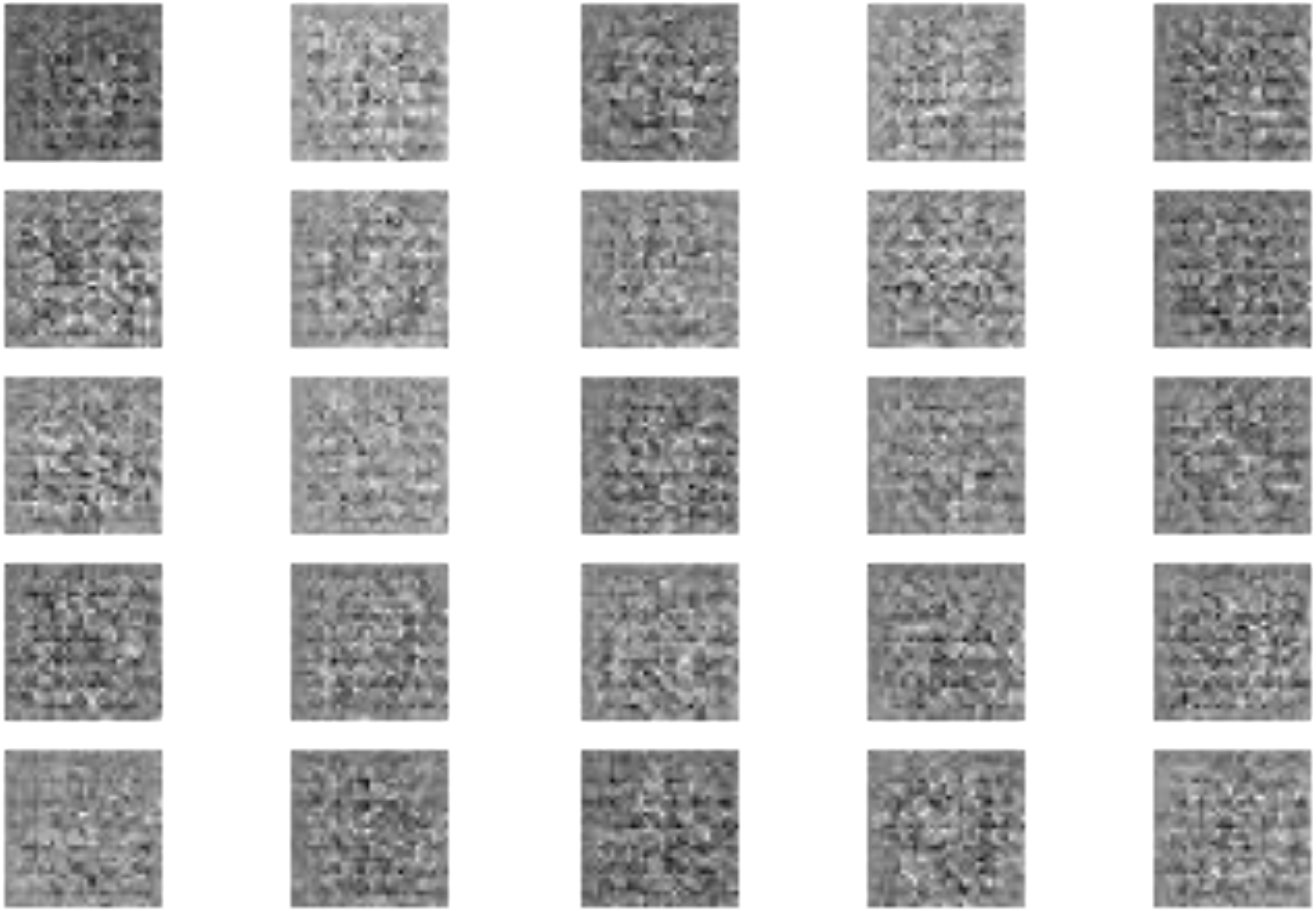
Step 2: Building the Generator Model

- So now we put the two stages together.
- First we create our generator model.
- Next we generate fake example images from the generator using our randomly generated data (the shape of x below is (25, 28, 28, 1)).
- Full code for Step 2 available here.

```
noise_dim = 100

# generate samples
n_samples = 25

# define the generator model
model = generator(noise_dim)

x,y = generate_fake_samples(model, noise_dim, n_samples)
```

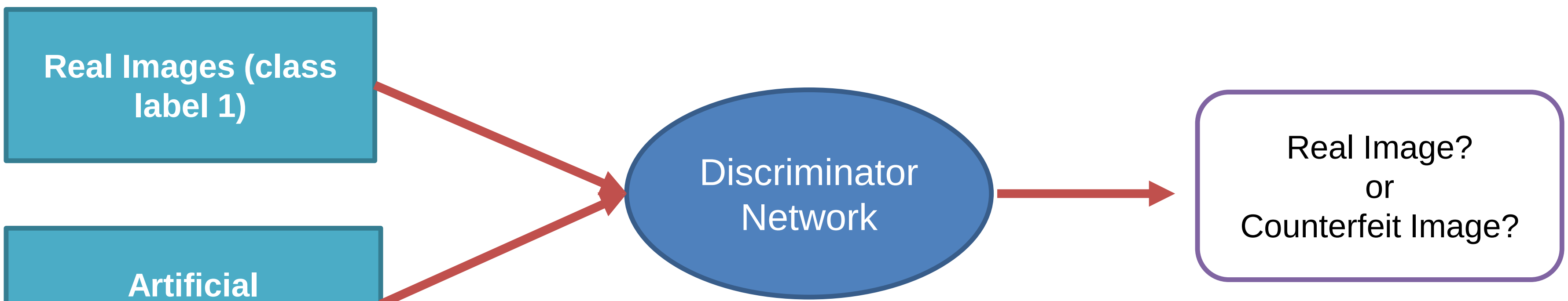
Step 2: Building the Generator Model

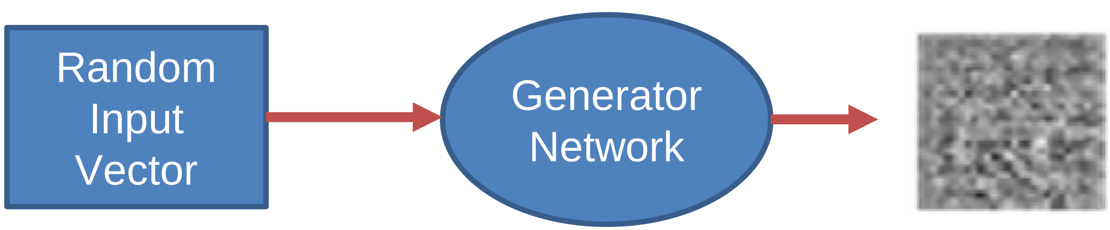
- When we visualize the data it is as follows:



Step 3 – Create the GAN Model

At this stage we have built a **separate discriminator** network and a **separate generator** network.



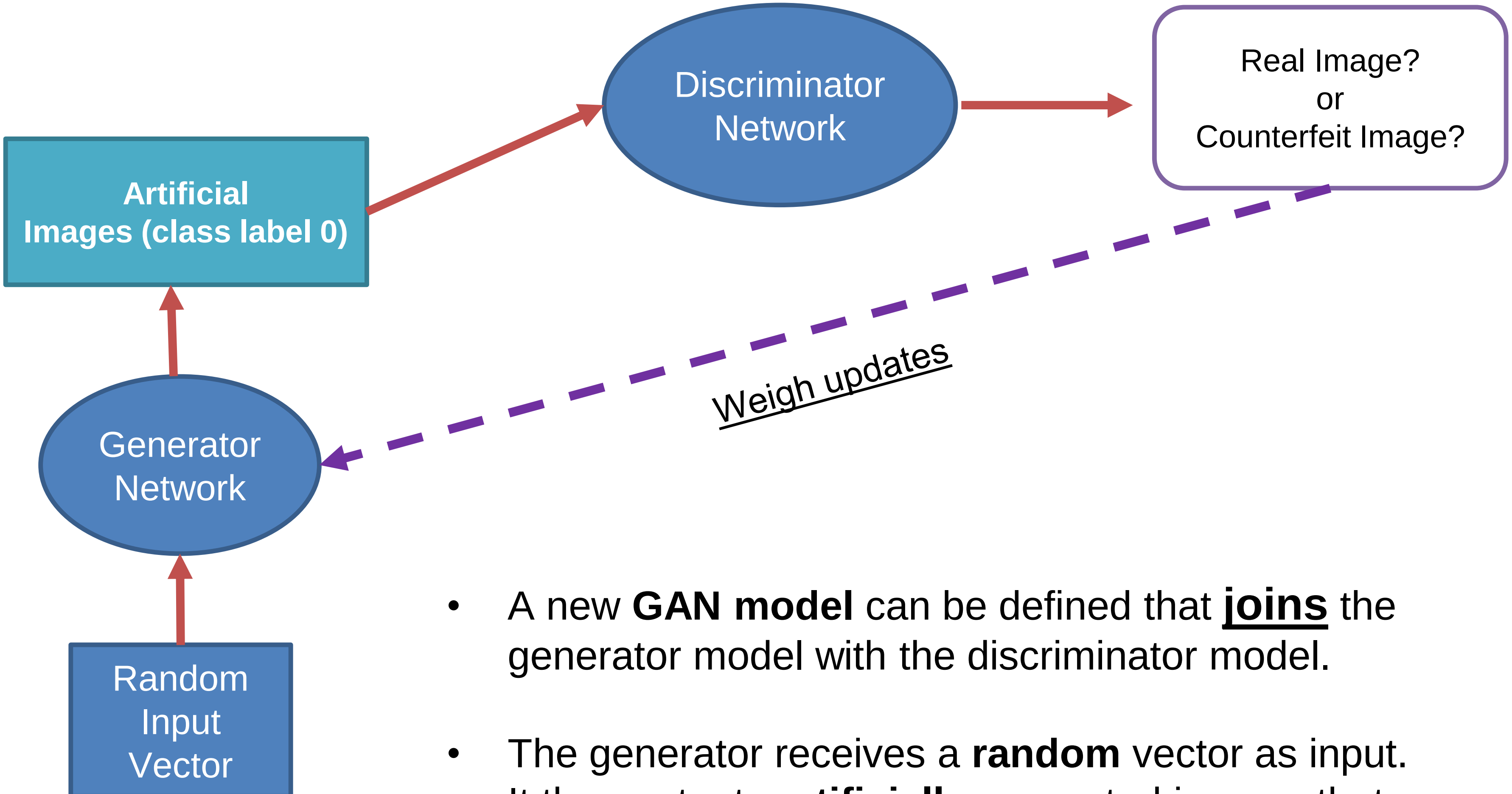


The **discriminator** is built only to differentiate between real and artificial images, so it can be **<u>trained</u>** in a **standalone** fashion.

Step 3 – Create the GAN Model

Training the Generator:

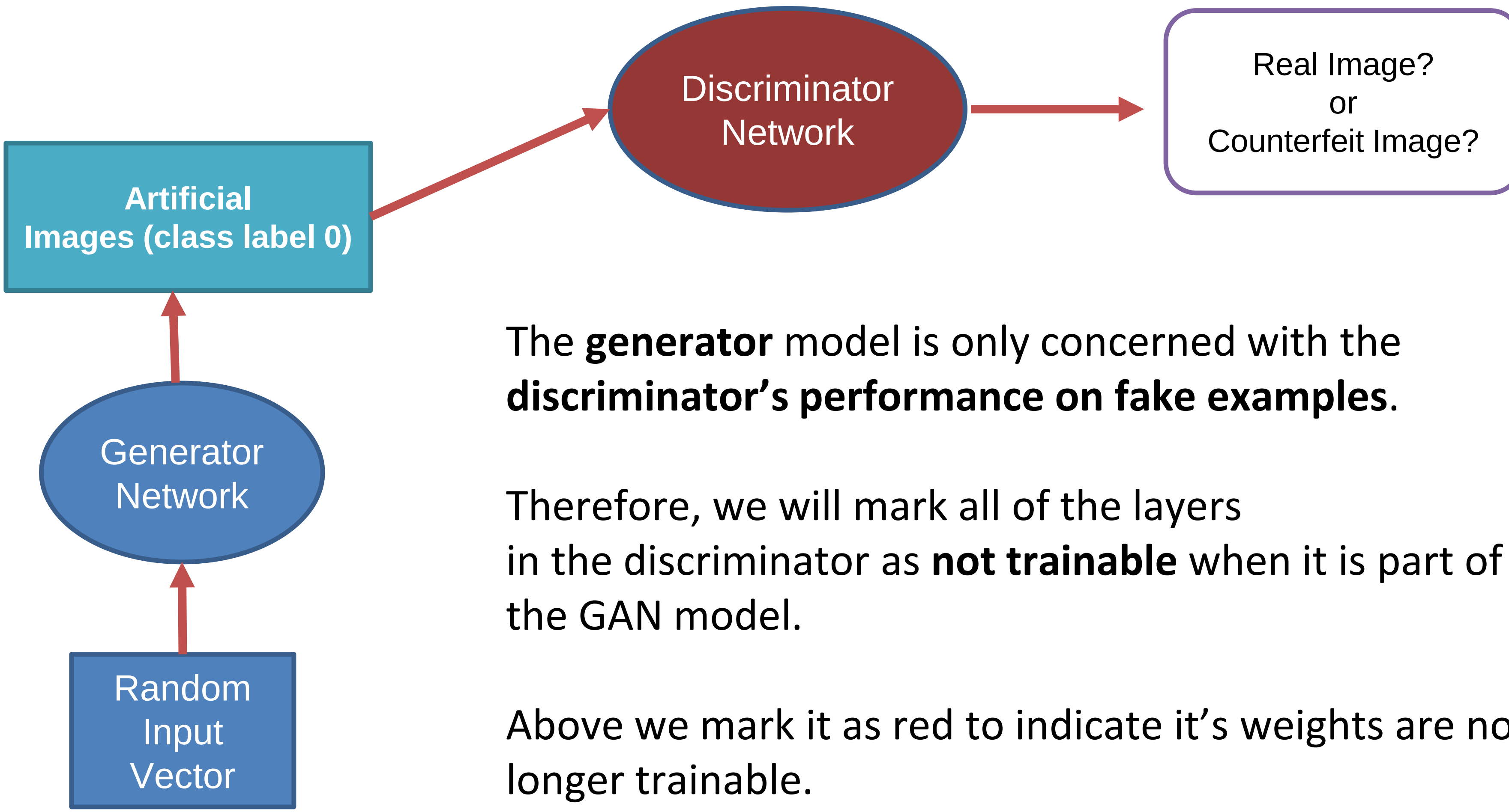
- In order to **train the generator** we must create a GAN model that connects the generator and discriminator.
- Remember we pass our **randomized vector** to the **Generator** model.
- It then generates **artificial samples** that are forwarded to the **discriminator** model.
- The **discriminator** model classifies the samples.
- We then use the output of the GAN model to **update the weights** for the **generator** model.



- A new **GAN model** can be defined that **joins** the generator model with the discriminator model.

- The generator receives a **random** vector as input.
- It then outputs **artificially** generated images that are then fed into the discriminator model.
- The discriminator will output a predicted **probability**.
- Finally based the **weights** of the generator model are updated (not the discriminator)

Step 3 – Create the GAN Model

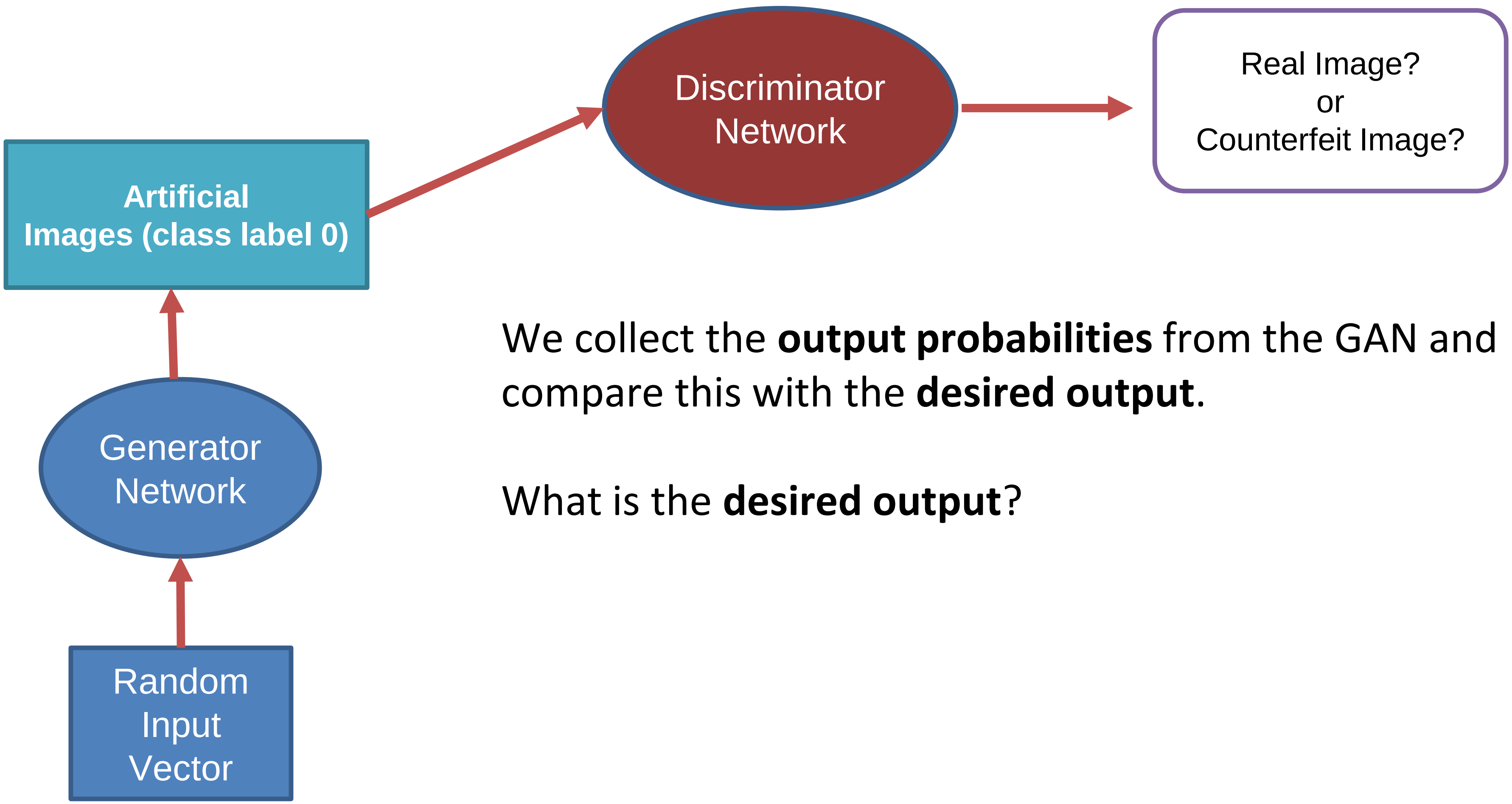


The **generator** model is only concerned with the **discriminator's performance on fake examples**.

Therefore, we will mark all of the layers in the discriminator as **not trainable** when it is part of the GAN model.

Above we mark it as red to indicate it's weights are no longer trainable.

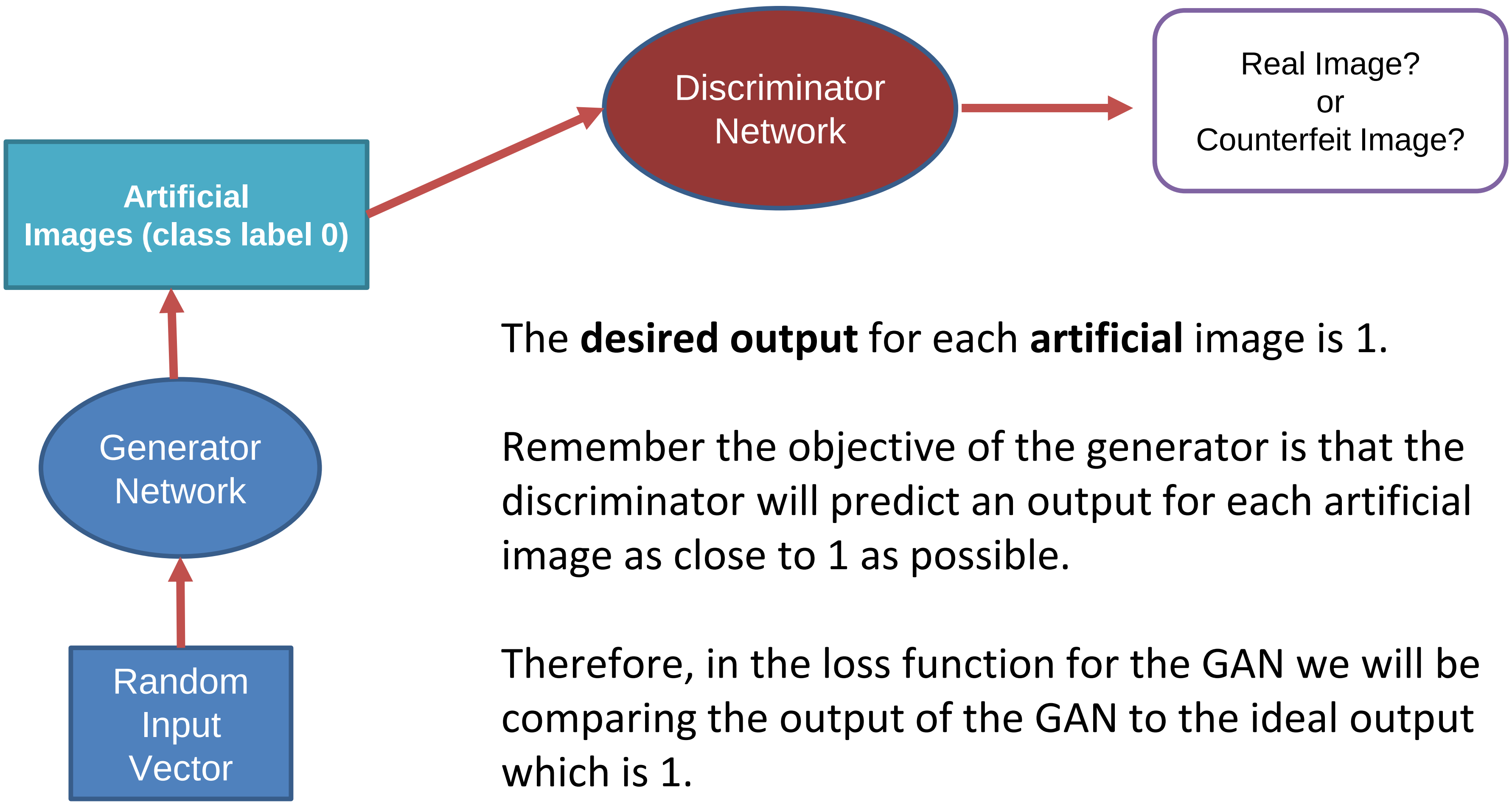
Step 3 – Create the GAN Model



We collect the **output probabilities** from the GAN and compare this with the **desired output**.

What is the **desired output**?

Step 3 – Create the GAN Model



The **desired output** for each **artificial** image is 1.

Remember the objective of the generator is that the discriminator will predict an output for each artificial image as close to 1 as possible.

Therefore, in the loss function for the GAN we will be comparing the output of the GAN to the ideal output which is 1.

```python
def gan(g_model, d_model):

        # disable weight update on GAN model
        d_model.trainable = False

        # connect the generator with the discriminator
        model = tf.keras.models.Sequential()
        model.add(g_model)
        model.add(d_model)

        # compile model
        opt = tf.keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
        model.compile(loss='binary_crossentropy', optimizer=opt)

        return model
```

Notice that we set the weights of the discriminator model to **not trainable** before we run it through the GAN. It is important to understand that this only impacts the discriminator when used as part of the GAN model (not the original discriminator model),

We have already compiled the discriminator model and it's weights were set to trainable. However, when using the GAN we don't want these weights to be trainable.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_10 (Sequential)   (None, 28, 28, 1)         2343681

_________________________________________________________________
sequential_9 (Sequential)    (None, 1)                 212865
=================================================================
Total params: 2,556,546
Trainable params: 2,318,209
Non-trainable params: 238,337

_________________________________________________________________
None
```

```python
def  generate_random_vectors(noise_dim, batch_size):

  # generate random numbers from a normal distribution
  x = np.random.randn(batch_size * noise_dim)

  # reshape into a batch of randomized vectors
  # (each vector will be inputted to the generator and will output an image)
  x_data = x.reshape(batch_size, noise_dim)

  return x_data
```

Before we go on to train our model we have one more helper method. In this method we just generate the randomized vectors which will be inputted into the GAN model.

In our example we will set the **noise_dim to 100** and the **batch_size to 256**. Therefore, in the first line we just generate **25,600** numbers randomly from a normal distribution and we next reshape this into an **256** rows, where each row is 100 elements in length.

```
noise_dim = 100
shape=(28,28,1)

disc_model = discriminator(shape)
generator_model = generator(noise_dim)
gan_model = gan(generator_model, disc_model)

# load image data
dataset = load_real_data()

# train model
epochs=50
batch_size=256

train(generator_model, disc_model, gan_model, dataset, noise_dim, epochs, batch_size)
```

At this point we have our three main models, the generator, the discriminator and the GAN model. The code above creates each of the required models, loads the real data and call the training process.

In the next step we will look in more detail at the training process.

Step 4 – Training the GAN Model

- For a number of training iterations repeat:

  - Sample a mini-batch of m noise samples $\{z^1, z^2, .. z^m\}$ from P_z
  - Sample a mini-batch of m training images $\{x^1, x^2, .. x^m\}$ from P_{data}.
  - Update the discriminator using stochastic gradient descent using the following loss function:

$$\frac{1}{m}\sum_{i}^{m} -\log\left(D(x^i)\right) - log(1 - D(G(zi)))$$

  - Sample a mini-batch of m noise samples $\{z^1, z^2, .. z^m\}$ from P_z
  - Update the generator with stochastic gradient ascent using the following loss function

$$\frac{1}{m}\sum_{i}^{m} -log(D(G(zi)))$$

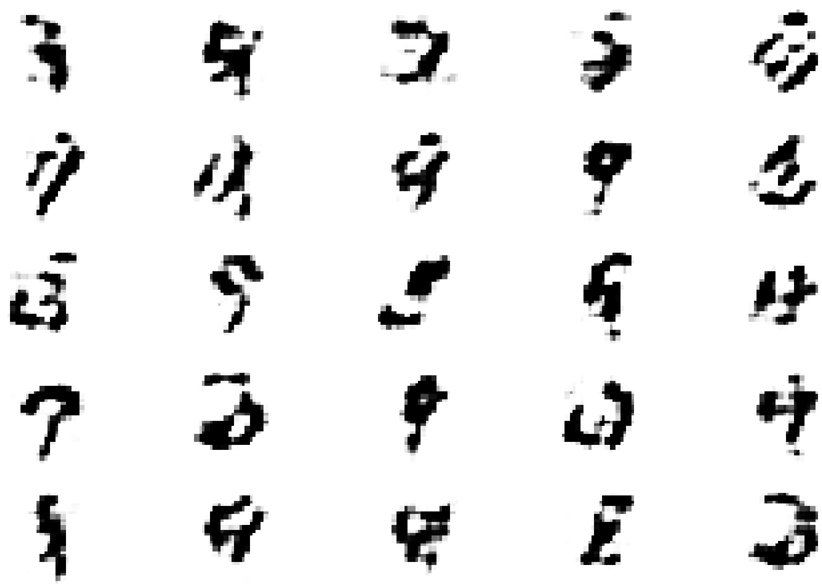
```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, noise_dim, epochs, batch_size):

    dLoss = []
    gLoss = []
    inx =[]
    counter = 0
    batches = int(dataset.shape[0] / batch_size)

    # for each epoch
    for i in range(epochs):

        # for each batch
        for j in range(batches):

            # obtain real image data
            X_real, y_real = get_real_samples(dataset,  batch_size)

            # generate artificial data
            X_fake, y_fake = generate_fake_samples(g_model, noise_dim,  batch_size)

            # combines both real and artificial data into a single data structure
            X, y = np.vstack((X_real, X_fake)), np.vstack((y_real, y_fake))

            # train discriminator for current batch and capture it's current loss
            d_loss, _ = d_model.train_on_batch(X, y)
```

```
# continued from previous slide

        # generate random vectors of noise (this becomes input the the GAN model)
        X_gan = generate_random_vectors(noise_dim, batch_size)

        # assign all GAN output data vectors a class label of 1
        y_gan = np.ones((batch_size, 1))

        # train the generator
        gan_loss = gan_model.train_on_batch(X_gan, y_gan)

        dLoss.append(d_loss)
        gLoss.append(gan_loss)

        inx.append(counter)
        counter += 1
```

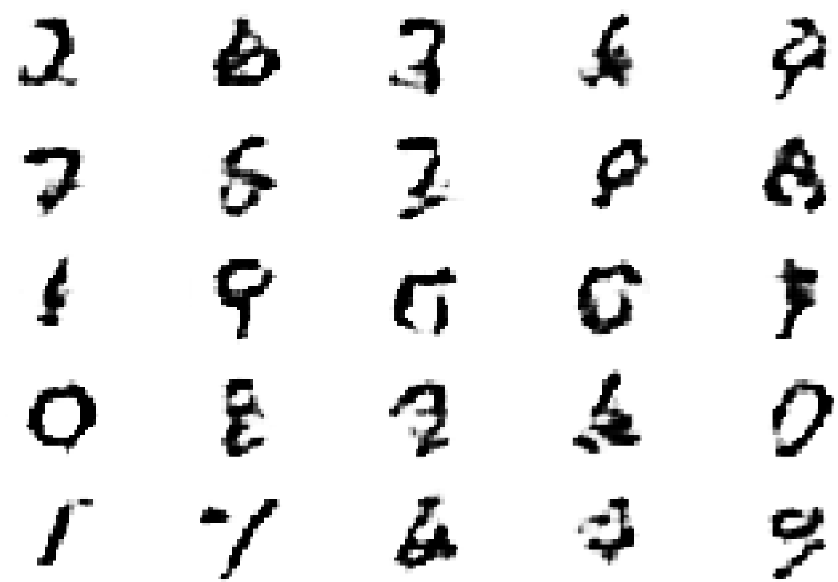
The full code for this example is available as a Google Colab notebook here.

Results (2 Epochs)

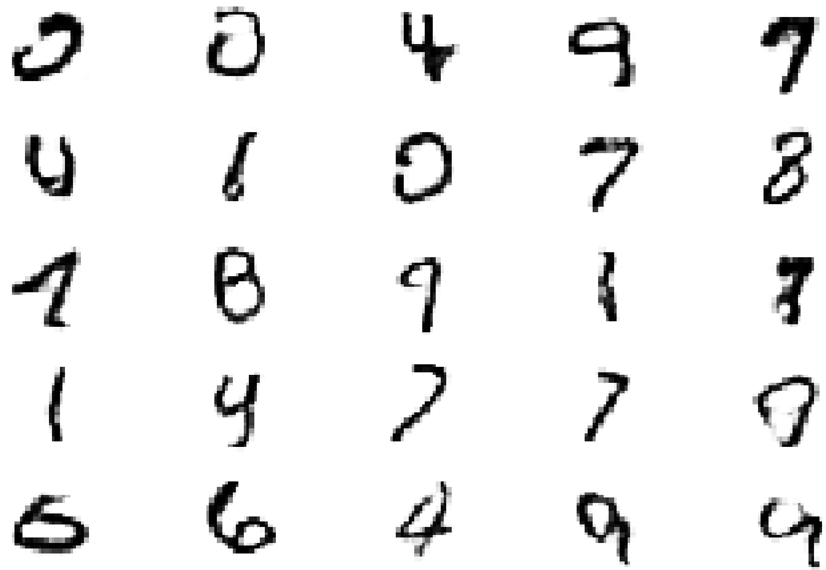
Results (5 Epochs)



Results (10 Epochs)



Results (50 Epochs)



Monitoring Loss

- You will notice in the training code that we store the discriminator and generator loss values as we continue to train the GAN.

- It is particular important to monitor **discriminator loss value**. If the discriminator loss **falls significantly** it is a strong indication that that generator is no longer producing competitive images and the discriminator is having little trouble in differentiating between real and artificial images.