# Table of contents

## What are Generative Adversarial Networks(GANs)?

Generative Adversarial Networks (GANs) were developed in 2014 someone called Ian Goodfellow and his teammates. GAN is a type of Generative modeling which generates a new set of data based on the training data.

In neural network language, what happens is that we have a pair of neural networks , the generator and the disciminator, the generator's job is to generate fake images , and the discremenator is trained to identify which images are real and which ones are fake. So the generator is trained to fool the discrimnator into classifying its images as real images.

Now we can break GAN into separate three parts:

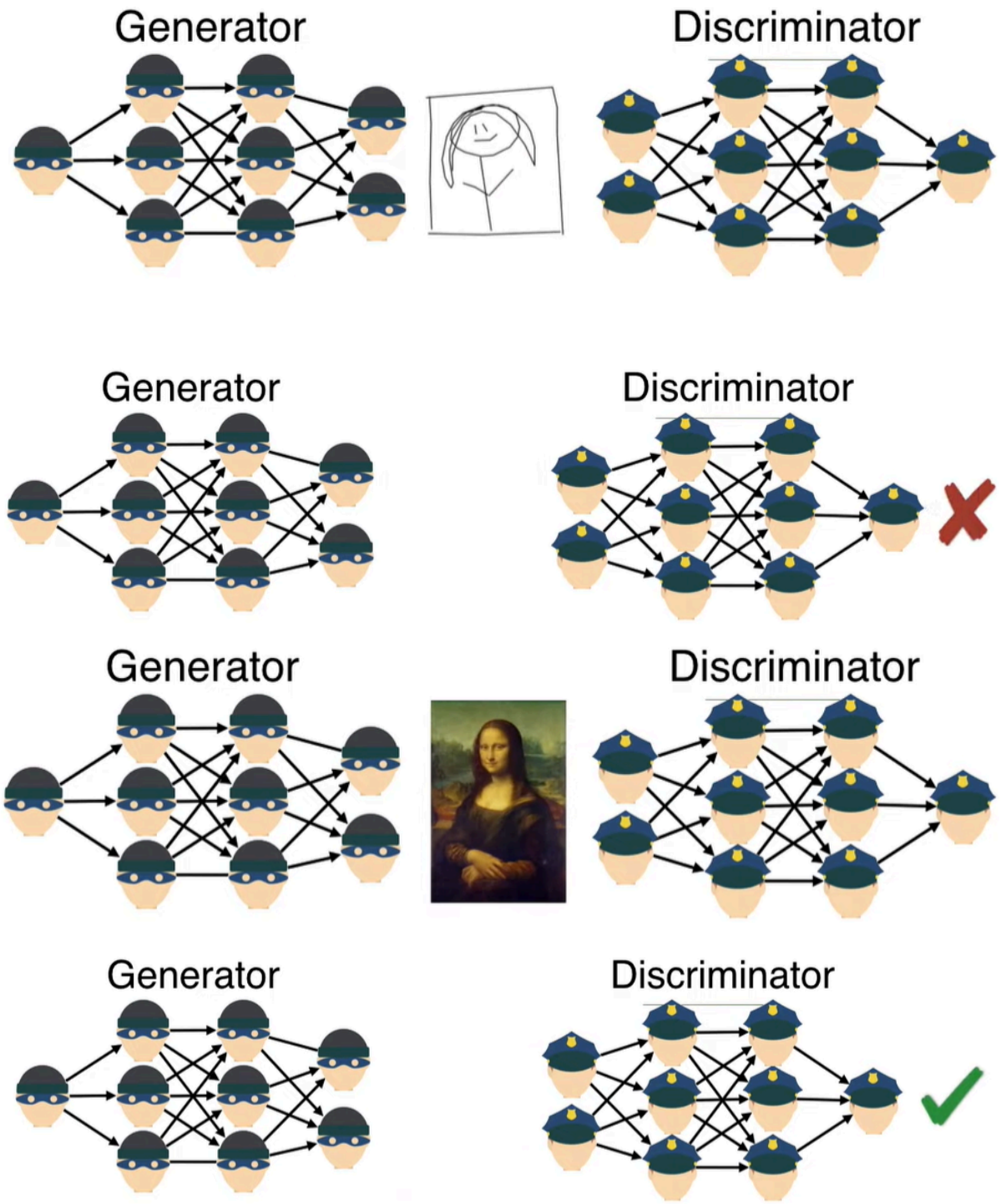
- Generative – To learn a generative model, that describes how data is generated in terms of a probabilistic model.so it explains how data is generated visually.
- Adversarial – The training of the model is done in an adversarial setting.
- Networks – use deep neural networks for training process.

We train both networks until we reach a point where the generator becomes more proficient at producing real samples, while the discimnator becomes more adept at differentiating between real and generated data. To put it in a nutshell, the process converges to a point where the generator is capable of generating high-quality samples that are difficult for the disciminator to distinguish from real data.

GANs are very good approach in many domains, such as text generation ,image synthesis and video generation. They've been used in tasks like

creating deepfakes, generating realistic images or even enhancing low-resolution images.

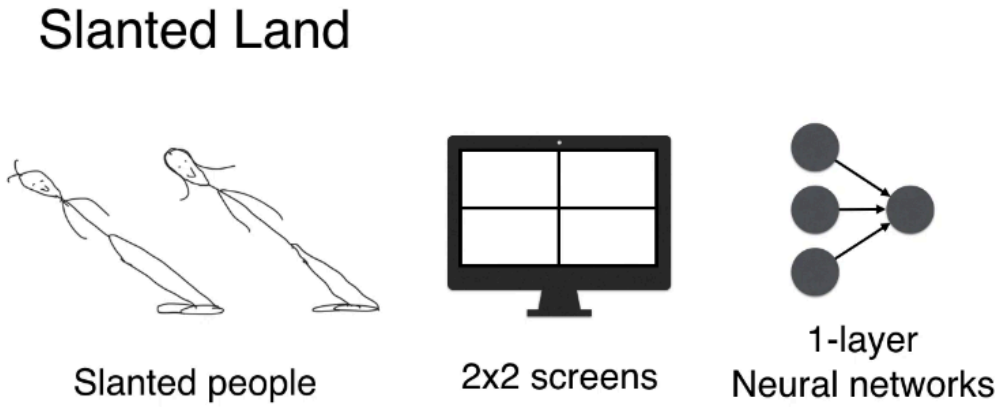
**Here's an illustrative example of how GANs work:**



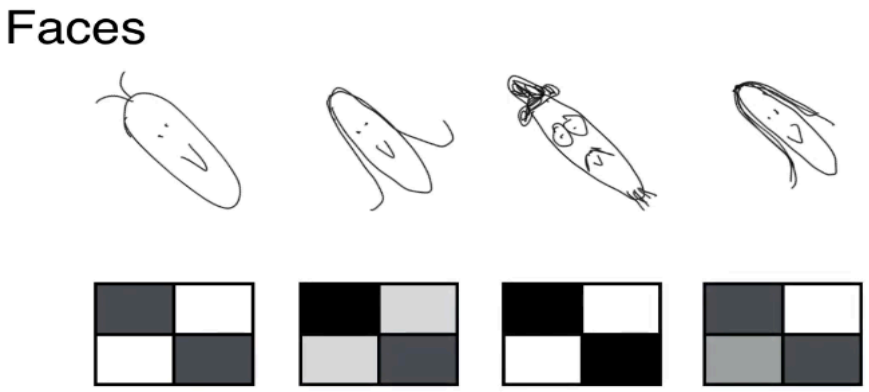
## How GANs work?

In order to understand how a pair of GANs work, let's take this very very simple example:

This is our setting. We live in a world called "Slanted Land", where everybody looks slightly elongated and walks at a 45 degree angle. This world has technology, but not as developed as ours. They have computer screens which display images, but the best resolution they've been able to workout is 2*2.
So they have black and white, two pixels by two pixels screens. Also, they have developed neural networks, but only very simple ones. As a matter of fact, they only know neural networks of one layer.

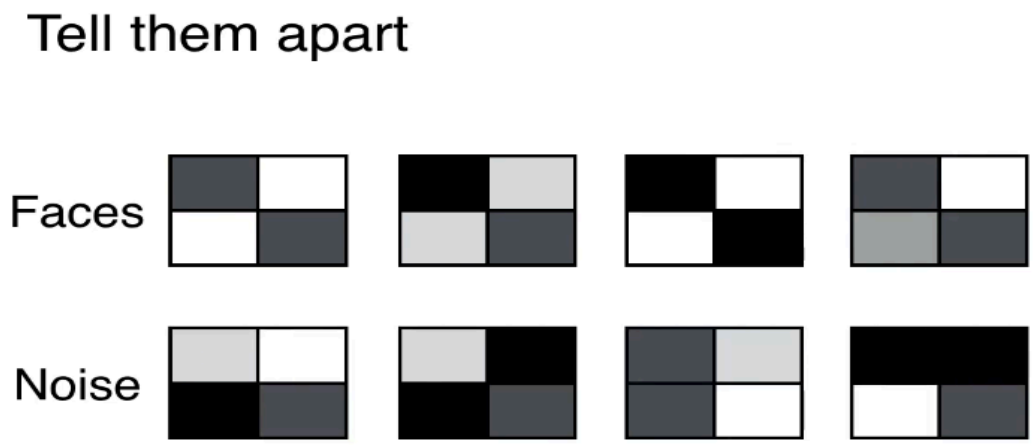



Here are four faces of people who live in slanted land. Notice that everybody is elongated and tilted 45 degrees to the left. Since the screens are only two pixels by two pixels. This is how the pictures of the people look

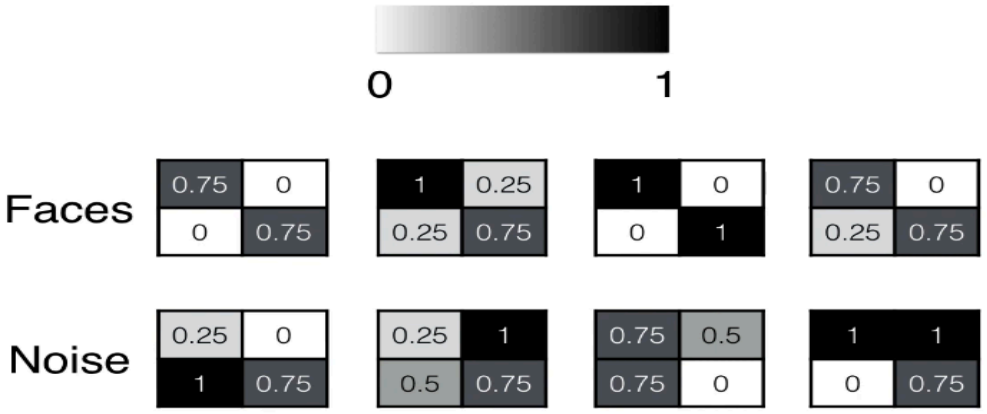


And this is how noisy images look. Notice that these are not faces since they don't look like a backlash, and these are mostly generated randomly.


So the goal of our networks is to be able to distinguish faces like these from noisy images, or non faces like these ones.




Let's attach some numbers here. We'll have a scale where a white pixel has values zero and a black pixel has a value of one. This way we can attach a value to every pixel in the two times two screens.

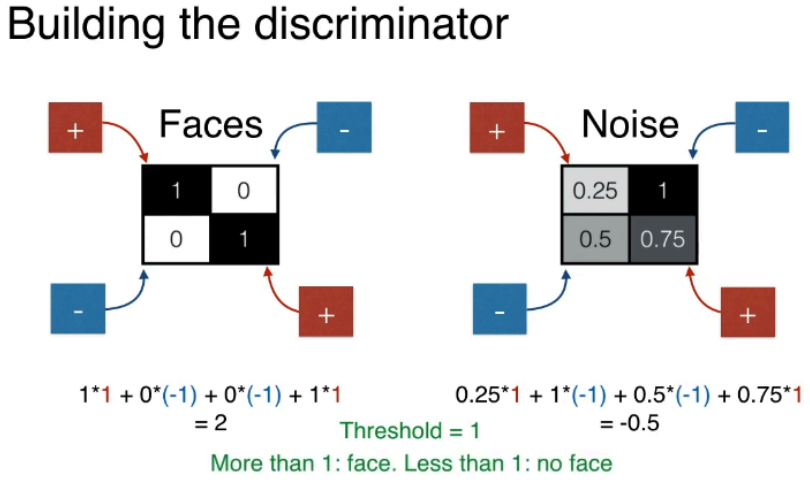



Okay, now we're ready to build our networks.

- **Build the Desciminator**

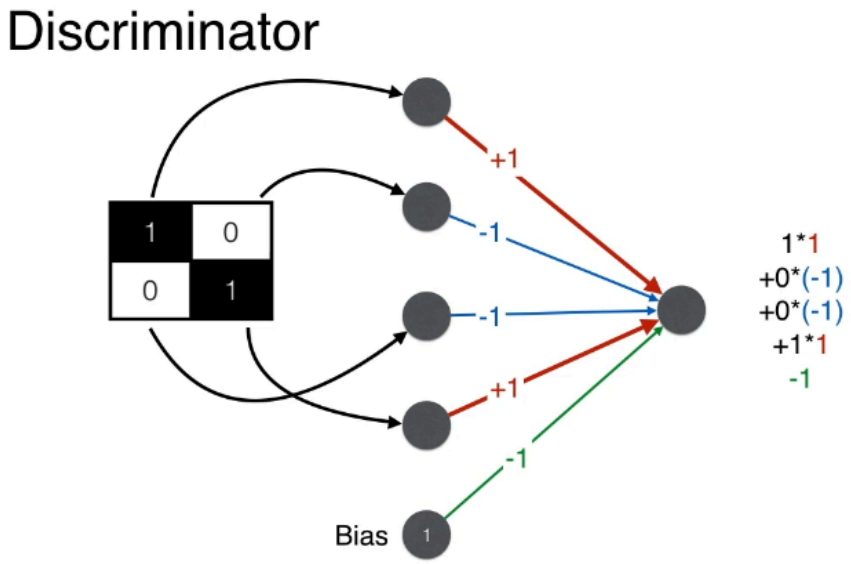
From the previous images we can notice that in the faces, the top left and the bottom right corners have large values because their pixels are dark, whereas the other two corners have small values because their pixels are light. On the other hand, in noisy images anything can happen.


Therefore, the way to tell faces apart is by **adding** the two values corresponding to the top left and bottom right corners, and **subtracting** the values corresponding to the other two corners. **In faces, this result will be very high, whereas in noisy images it will be low.**

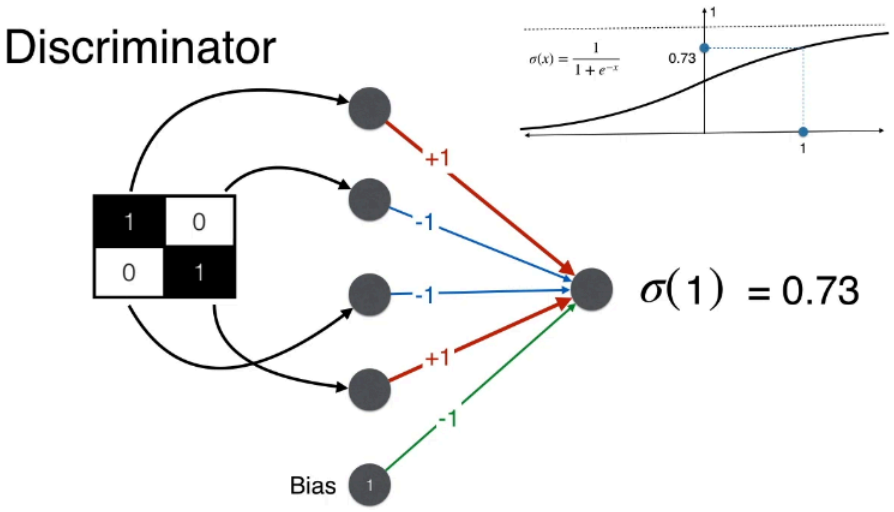


We can add a cutoff or a threshold of, say **one**, and say that any image that scores one or higher is a face, and any image with scores less than one is not a face or is a noisy image.

In neural network lingo, this is how things look ,The values of R4 pixels get multiplied by plus or minus one, depending on what diagonal they are on, and then we subtract the total value of one or the bias. We add these four numbers, and if the score is one or more, then the image is classified as a face. And if it's less than one, that is classified as not a face.

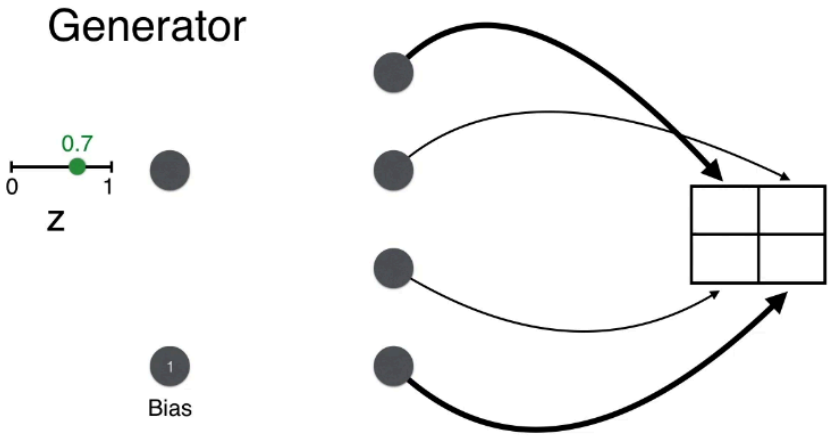



In this case, the image is a face because it gets a score of one. We can also output the probability that something is a face. By using the sigmoid function, we apply the sigmoid, which is this function that sends high numbers to numbers close to one, and low numbers to numbers close to zero, and we get sigmoid of one, which is 0.73. This discriminator network then assigns to this image a probability of 73% that it is a face. Since it is a high probability greater than 50%, we conclude that this discriminator thinks that the image is a face, which is correct.

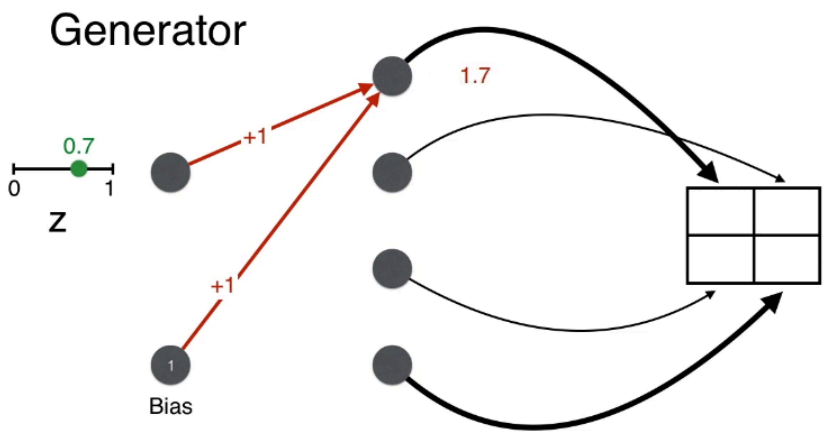



- **Build the Generator**

this is how the generator works. First we start by picking an input z which is a random number between 0 and 1. In this case, let's say will be 0.7.



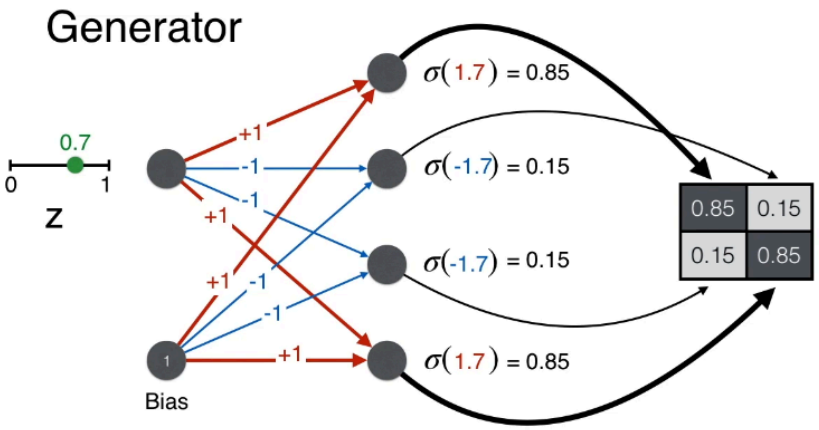

In general, the input would be a vector that comes from some fixed distribution. Now let's build a neural network. What we really want is to have some large values and some small ones. The large ones are represented by thick edges and the small ones by thin edges. Because remember that we want large values for the top left and bottom right corner, and small values for the top right and bottom left corner. So since the top output has to be large, we want these weights coming in to be large. So let's make them plus ones. Now what do we get for the output here. Well first we're going to get a score of plus one times 0.7 plus one which is 1.7.




Now let's look at the second node. It corresponds to the top right corner. So it has to be a small value. So let's put negative numbers here. What do we get for the score. We get minus one times 0.7 minus one which is -1.7. In the same way we want a small value here. So we put weights of minus one and we get -1.7. And for the last one we want a high value. So we put plus one and we get plus one times 0.7 plus one which is 1.7.

Now those are just the scores. We need to apply sigmoid to find the probabilities So we apply sigmoid and we get 0.85 0.15 0.15 and 0.85. Those are the values that will go in our pixels.

Notice that our image looks like a diagonal, which is how we define our faces. Notice that by the way we built it, this neural network will always generate large values for the top left and bottom right corners, and small values for the top right and bottom left corner. No matter what value z we input. Because remember z is between 0 and 1. Therefore, this neural network will always generate a face. That means it's a good generator.



- **Choosing Error Function**

An error or cost function is a way to tell the network how it's doing in order for it to improve. If the error is large, then the network is not doing well and it needs to reduce the error to improve the error function that we'll use to train these Gans is the **log loss**, which is the natural logarithm.

**So why we would use this error function for our networks training process?**
Let's first say that **we have a label of one**, and our neural network predicted is as 0.1. This is a bad prediction, and we should produce a large error because 0.1 is very far from one.
On the other hand, if the label is one and the prediction is 0.9, then that's a good prediction because the prediction is very close to the label. So this should produce a small error. How then can we find a formula for this error? Well, notice that the negative natural logarithm of 0.1 is 2.3, which is large, while the negative logarithm of 0.9 is 0.1.

$$\text{Error} = -\ln(\text{prediction})$$

| | | |
|---|---|---|
| Label: 1<br>Prediction: 0.1 | Error: large | $-\ln(0.1) = 2.3$ |
| Label: 1<br>Prediction: 0.9 | Error: small | $-\ln(0.9) = 0.1$ |

In the other extreme, when the label is zero, In this case, if we had a prediction of 0.1, it would be good because it's close to the label. Therefore, the error should be small. On the other hand, a prediction of 0.9 is terrible, so the error should be large.

The function that we need here is similar to the previous one, but with a slight difference It is the negative logarithm of one minus the prediction.

Error = -ln(1 - prediction)

| | | |
|---|---|---|
| Label: 0<br>Prediction: 0.1 | Error: small | $-\ln(0.9) = 0.1$ |
| Label: 0<br>Prediction: 0.9 | Error: large | $-\ln(0.1) = 2.3$ |

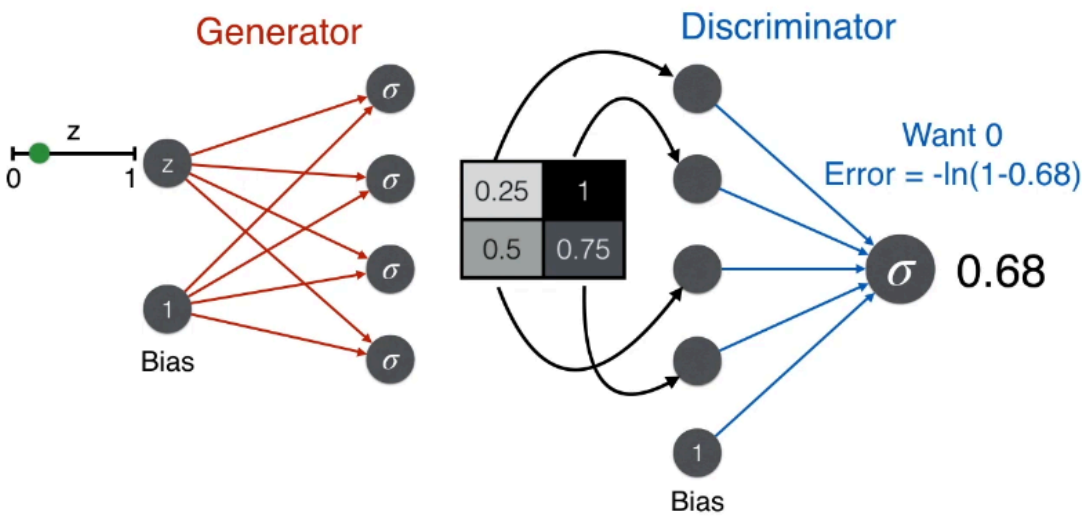
- **Training Process**

Now we're ready to train the generator and the discriminator, we've our pair of neural networks. And notice that the weights are not yet defined. So we start by defining them as random numbers. Now we select **z** as some random number between 0 and 1, which is going to serve as the input to the generator.
We do a forward pass on the generator to obtain some image, which is probably not a face. Since the weights are random, this is going to be our generated image.

Now we pass that generated image through the discriminator, so the discriminator can tell if it's fake or not. The discriminator outputs a probability.

So let's say that it's for example 0.68. Now pay close attention because this is where the rubber meets the road. This is where we define the correct error functions. First let's think what does the discriminator want to do here. In other words, if the discriminator was great, what should it output? Well, since the image is not a face, but it's a generated image from the generator, then the discriminator should be saying that it's fake. That means that the discriminator **should be outputting a zero**.

If we remember the error function, the way we measure an error when we want the neural network to output a zero is the negative logarithm of one minus the prediction. This is an error that will help us train the discriminator.




Also, we choosed the perfect error function for the generator , why? Because the generator wants this whole neural network the connection of the two to output a one. That means that the error function from the generator is negative logarithm of the prediction. So that is the error function that will help us train the weights of the generator.

**So what happens after many epochs?** We'll have updated weights for both networks , where the generator wants to generate the perfect Slanted faces , and the dicrementor trying its hardest not to be fooled.