Lecture 19: Generative Adversarial Networks

Roger Grosse

1 Introduction

Generative modeling is a type of machine learning where the aim is to model the distribution that a given set of data (e.g. images, audio) came from. Normally this is an unsupervised problem, in the sense that the models are trained on a large collection of data. For instance, recall that the MNIST dataset was obtained by scanning handwritten zip code digits from envelopes. So consider the distribution of all the digits people ever write in zip codes. The MNIST training examples can be viewed as samples from this distribution. If we fit a generative model to MNIST, we're trying to learn about the distribution from the training samples. Notice that this formulation doesn't use the labels, so it's an unsupervised learning problem.

Generative models are sometimes used for supervised learning, but we won't consider that here. See Gaussian discriminant analysis or naive Bayes in CSC411.

Figure 1(a) shows a random subset of the MNIST training examples, and Figure 1(b) shows some samples from a generative model (called a Deep Boltzmann Machine) trained on MNIST[1]; this was considered an impressive result back in 2009. The model's samples are visually hard to distinguish from the training examples, suggesting that the model has learned to match the distribution fairly closely. (We'll see later why this can be misleading.) But generative modeling has come a long way since then, and in fact has made astounding progress over the past 4 years. Figure 1(c) shows some samples from a Generative Adversarial Network (GAN) trained on the "dog" category from the CIFAR-10 object recognition dataset in 2015[2]; this was considered an impressive result at the time. Fast forward two years, and GANs are now able to produce convincing high-resolution images[3], as exemplified in Figure 1(d).

Why train a generative model?

- Most straightforwardly, we may want to generate realistic samples, e.g. for graphics applications. (Unfortunately, there are more nefarious uses as well, such as producing realistic fake videos of politicians.)
- We may wish to model the data distribution in order to tell which of several candidate outputs is more likely; e.g., see Lecture 7, which used speech recognition as a motivation for language modeling.
- We may want to train a generative model in order to learn useful high-level features which can be applied to other tasks. This got a

[1] Salakhutdinov and Hinton, 2009. Deep Boltzmann machines.

[2] Denton et al., 2015. Deep generative image models using a Laplacian pyramid of adversarial networks.

[3] Karras et al., 2017. Progressive growing of GANs for improved quality, stability, and variation.

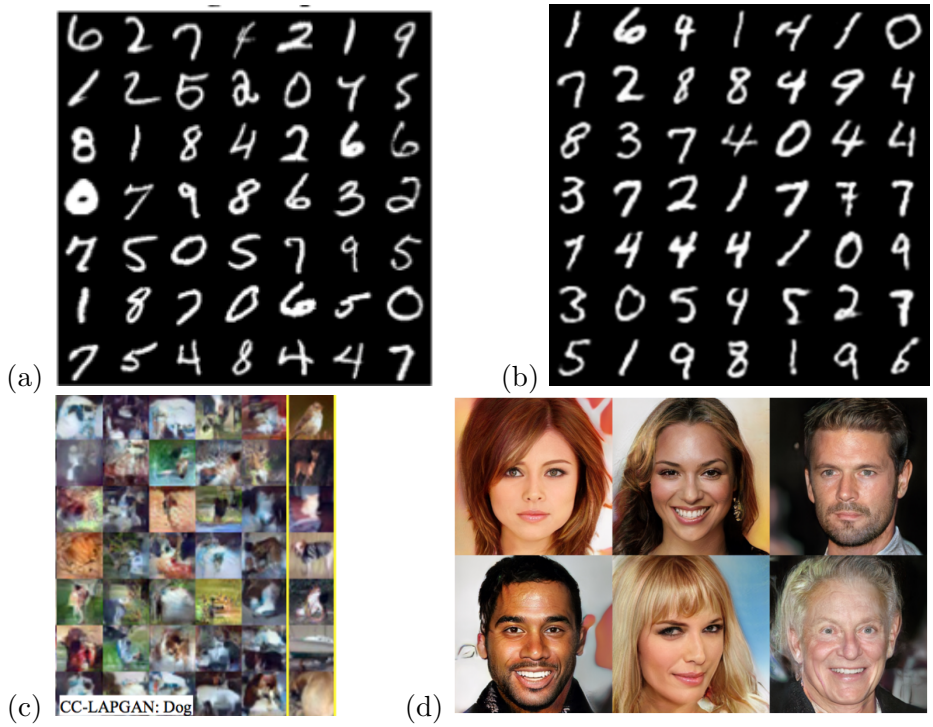


Figure 1: **(a)** Training images from the MNIST dataset. **(b)** Samples from a Deep Boltzmann Machine (Salakhutdinov and Hinton, 2009). **(c)** Samples from a GAN trained on the "dog" category of CIFAR-10 (Denton et al., 2015) **(d)** Samples from a GAN trained on images of celebrities (Karras et al., 2017).

lot of attention about 10 years ago, with the motivation that there's a lot more unlabeled data than labeled data, and having more data ought to let us learn better features. This motivation has declined in importance due to the availability of large labeled datasets such as ImageNet, and to the surprising success of supervised features at transferring to other tasks.

Last time, we saw very simple examples of learning distributions, i.e. fitting Gaussian and Bernoulli distributions using maximum likelihood. This lecture and the next one are about deep generative models, where we use neural nets to learn powerful generative models of complex datasets. There are four kinds of deep generative models in widespread use today:

- Generative adversarial networks (the topic of today's lecture)
- Reversible architectures (Lecture 20)
- Autoregressive models (Lectures 7, 15–17, and 20)
- Variational autoencoders (beyond the scope of this class)

You can learn about variational autoencoders in csc412.

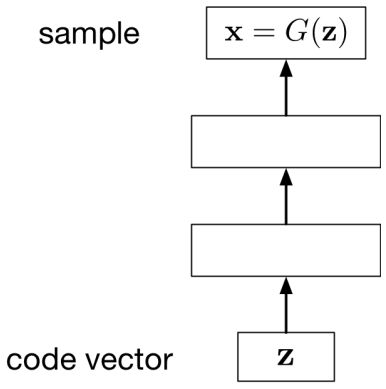
Three of these four kinds of generative models are typically trained with maximum likelihood. But **Generative Adversarial Networks (GANs)** are based on a very different idea: we'd like the model to produce samples which are indistinguishable from the real data, as judged by a discriminator network whose job it is to tell real from fake. GANs are probably the current state-of-the-art generative model, as judged by the quality of the samples.

2 Implicit Generative Models

GANs are a kind of **implicit generative model**, which means we train a neural net to produce samples; this implicitly defines a probability distribution, namely the distribution of samples that the network generates. But the model doesn't explicitly represent the distribution, in the sense that it can't answer other queries, such as the probability assigned to a particular observation.

When does it suffice to train an implicit generative model, and when would you like an explicit one?

The architecture of an implicit generative model, or **density network**, is as follows: we first sample a **code vector z** from a simple, fixed distribution such as a uniform distribution or a standard Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$. Note that this distribution is fixed, i.e. not learned. This code vector is then passed as input to a deterministic **generator network** G, which produces an output sample $\mathbf{x} = G(\mathbf{z})$. Schematically:



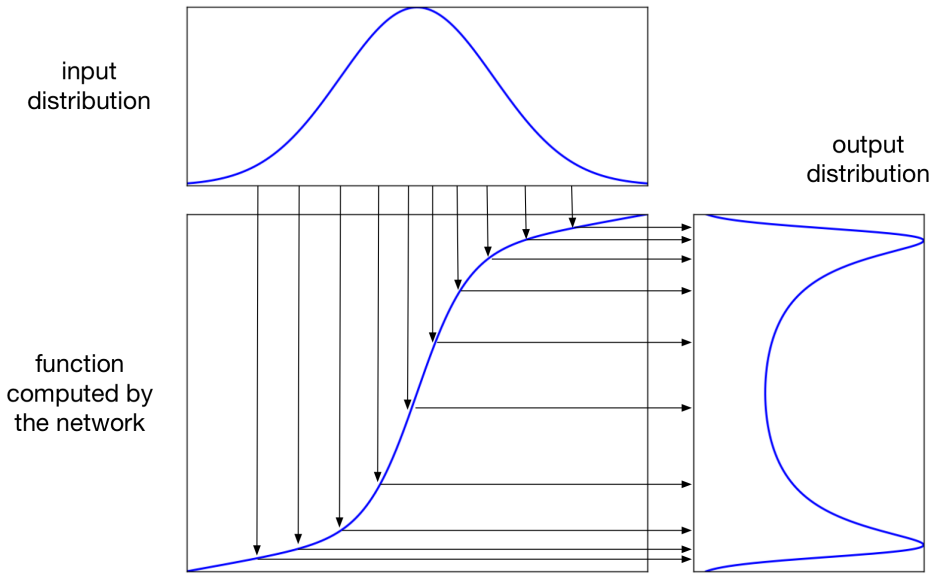
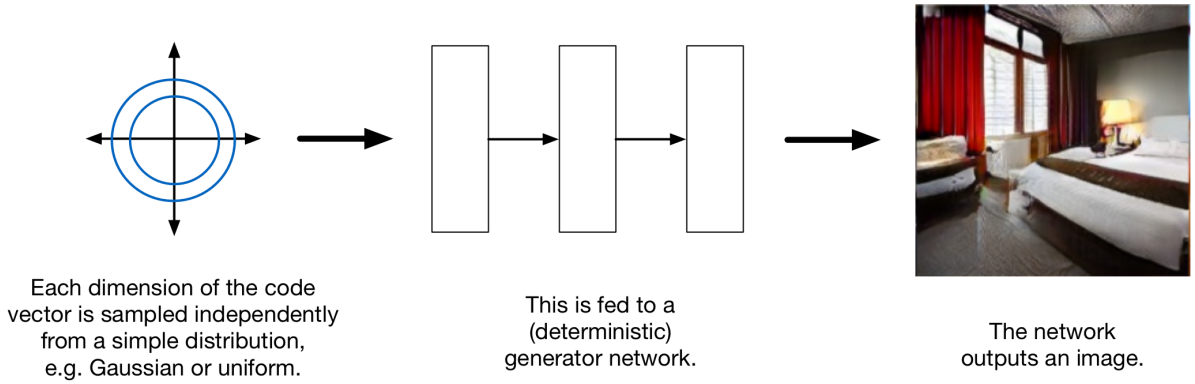


Figure 2: A 1-dimensional generator network.

Implicit generative models are pretty hard to think about, since the relationship between the network weights and the density is complicated. Figure 2 shows an example of a generator network which encodes a univariate distribution with two different modes. Try to understand why it produces the density shown.

When we train an implicit generative model of images, we're aiming to learn the following:



This probably seems preposterous at first; how can you encode something as complex as a probability distribution over images in terms of a deterministic mapping from a spherical Gaussian distribution? But amazingly, it works.

3 Generative Adversarial Networks

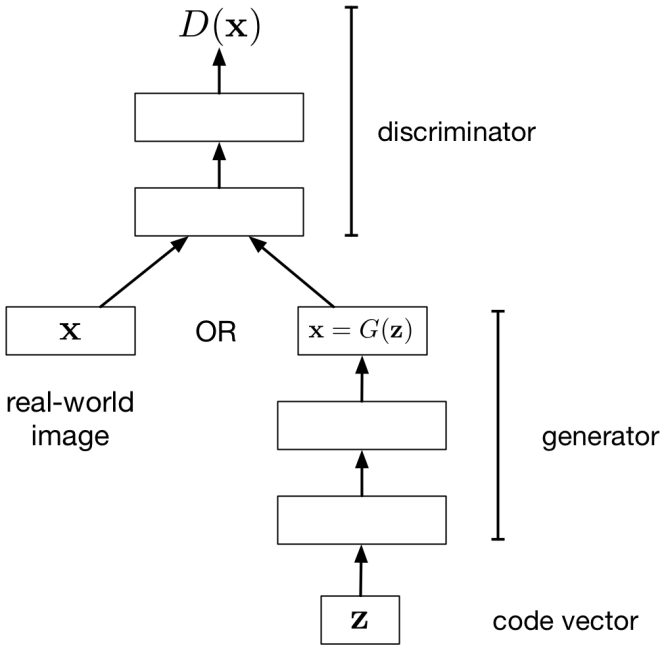
Recall that implicit generative models don't let us query the probability of an observation, so clearly we can't train them using maximum likeli-

hood. Generative adversarial networks use an elegant training criterion that doesn't require computing the likelihood. In particular, if the generator is doing a good job of modeling the data distribution, then the generated samples should be indistinguishable from the true data. So the idea behind GANs is to train a **discriminator network** whose job it is to classify whether an observation (e.g. an image) is from the training set or whether it was produced by the generator. The generator is evaluated based on the discriminator's inability to tell its samples from data.

To rephrase this, we simultaneously train two different networks:

- The generator network G, defined as in Section 2, which tries to generate realistic samples
- The discriminator network D, which is a binary classification network which tries to classify real vs. fake samples. It takes an input $\mathbf{x}$ and computes $D(\mathbf{x})$, the probability it assigns to $\mathbf{x}$ being real.

The two networks are trained competitively: the generator is trying to fool the discriminator, and the discriminator is trying not to be fooled by the generator. This is shown schematically as follows:



The discriminator is trained just like a logistic regression classifier. Its cost function is the cross-entropy for classifying real vs. fake:

$$\mathcal{J}_D = \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[-\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}}[-\log(1 - D(G(\mathbf{z})))] \tag{1}$$

The cost function is written as an expectation, but we estimate it using samples (training images or samples from the generator) in order to update the weights with SGD. This is known as Monte Carlo estimation.

Here, $\mathbf{x} \sim \mathcal{D}$ denotes sampling from the training set. If the discriminator has low cross entropy, that means it can easily distinguish real from fake; if it has high cross-entropy, that means it can't. Therefore, the most straightforward criterion for the generator is to *maximize* the discriminator's cross-entropy. This is equivalent to making the generator's cost function the negative cross-entropy:

$$\begin{aligned} \mathcal{J}_G &= -\mathcal{J}_D \\ &= \text{const} + \mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))] \end{aligned} \tag{2}$$

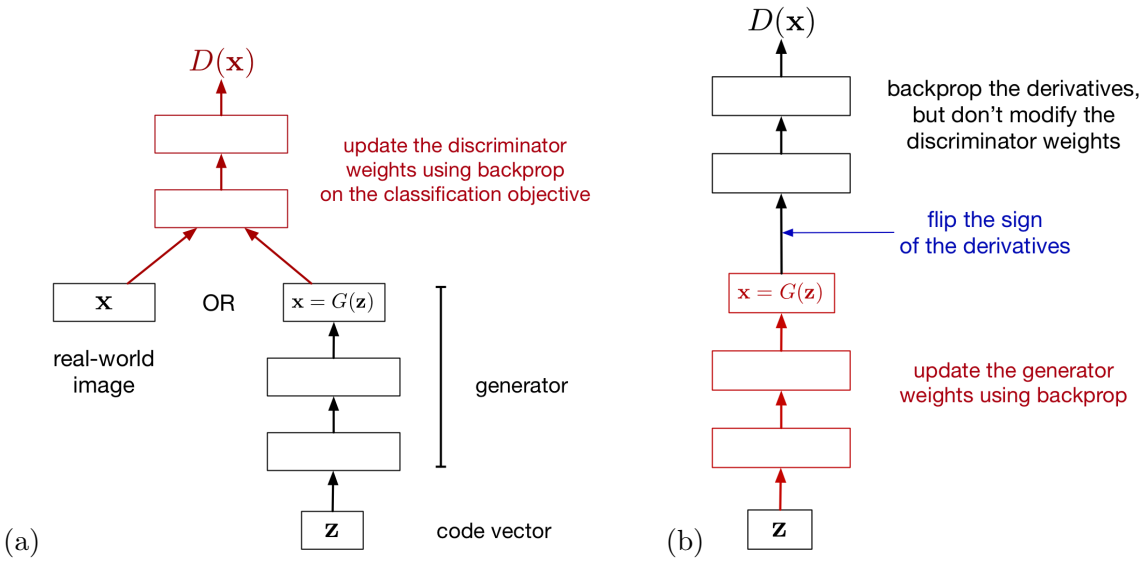


Figure 3: **(a)** Updating the discriminator. **(b)** Updating the generator.

Note that the generator has no control over the first term in Eqn. 1, which is why we simply write it as constant.

Consider the cost function from the perspective of the generator. Given a fixed generator, the discriminator will learn to minimize its cross-entropy. The generator knows this, so it wants to maximize the minimum cross-entropy achievable by any discriminator. Mathematically, it's trying to compute

$$\arg\max_G \min_D \mathcal{J}_D. \tag{3}$$

This this cost function involves a min inside a max, it's called the **minimax formulation**. It's an example of a perfectly competitive game, or **zero-sum game**, since the generator and discriminator have exactly opposing objectives.

The generator and discriminator are trained jointly, so they can adapt to each other. Both networks are trained using backprop on their cost functions. This is handled automatically by autodiff packages, but conceptually we can understand it as shown in Figure 3. In practice, we don't actually do separate updates for the generator and discriminator; rather, both networks are updated in a single backprop step. Figure 4 shows a cartoon example of a GAN being trained on a 1-dimensional toy dataset.

Unfortunately, not having a unified cost function for training both networks makes the training dynamics much more complicated compared with the optimization setting, as we assumed in the rest of this course. This means GAN training can be pretty finnicky.

3.1 A Better Cost Function

The minimax formulation is one way of training GANs, but it has a problem: saturation. Recall from Lecture 4 that if you use a logistic activation function and squared error loss to do classifiation, the cost function saturates when the predictions are very wrong. I.e., the cost function flattens out, resulting in small updates to the weights. We saw that this is problematic for optimization, since if the prediction is very wrong, we ought to make a large update. Our solution was to switch to the cross-entropy loss, which treats a prediction of 0.001 for the correct category as much worse than a prediction of 0.01. Hence, there would be a strong gradient signal

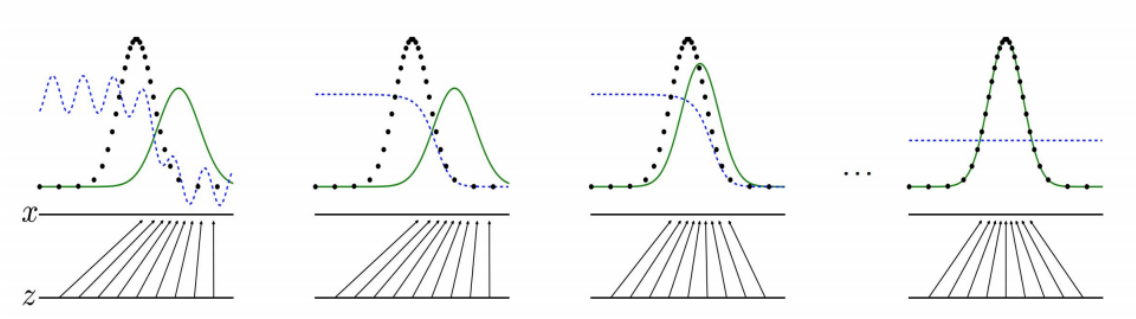


Figure 4: Cartoon of training a GAN to model a 1-dimensional distribution. **Black:** the data density. **Blue:** the discriminator function. **Green:** the generator distribution. **Arrows:** the generator function. First the discriminator is updated, then the generator, and so on. Figure from Goodfellow et al., 2014, "Generative adversarial nets".

pushing the network to assign probability 0.01 rather than 0.001, and then 0.1 rather than 0.01, and so on.

The same reasoning applies to GANs. Observe what happens if the discriminator is doing very well, or equivalently, the generator is doing very badly. This means $D(G(\mathbf{z}))$ is very close to 0, and hence $\mathcal{J}_G$ is close to 0 (the worst possible value). If we were to change the generator's weights just a little bit, then $\mathcal{J}_G$ would still be close to 0. This means we're in a plateau of the minimax cost function, i.e. the generator's gradient is close to 0, and it will hardly get updated.

We can apply a fix that's roughly analogous to when we switched from logistic-least-squares to logistic-cross-entropy in Lecture 4. In particular, we modify the generator's cost function to magnify small differences in $D(G(\mathbf{z}))$ when it is close to 0. Mathematically, we replace the generator cost from Eqn. 2 with the modifed cost:

$$\mathcal{J}_G = \mathbb{E}_{\mathbf{z}}[-\log D(G(\mathbf{z}))] \tag{4}$$

This cost function is really unhappy when the discriminator is able to confidently recognize one of its samples as fake, so the generator gets a strong gradient signal pushing it to make the discriminator less confident. Eventually, it should be able to produce samples which actually fool the discriminator. The relationship between the two generator costs is shown in Figure 5. The modified generator cost is typically much more effective than the minimax formulation, and is what's nearly always used in practice.

4 Style Transfer with CycleGAN

GANs by themselves are pretty impressive, judged by their ability to produce convincing samples. But we can also use GANs as components of a bigger architecture, which lets us do some pretty neat things. One of the most surprising recent examples is the **cycle consistent GAN**, or **CycleGAN**, an architecture for doing style transfer of images. Recall that the style transfer task is to take an image in one style (such as a photograph)

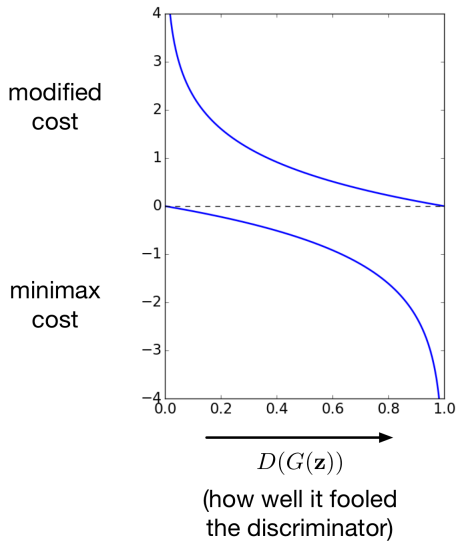


Figure 5: Comparison of the minimax generator cost to the modified one.

and transform it to be a different style (such as a van Gogh painting) while preserving the content of the image (e.g. objects and their locations).

It's unlikely that we have lots of pairs of images in both styles (e.g. a photograph and a van Gogh painting that matches it). So let's assume we have **unpaired data**, i.e. collections of unrelated images in the two styles.

We'd like to train two generators: one to go from Style A to Style B, and one to go from Style B to Style A. From how we stated the problem, we have two desiderata:

- we'd like the generator to produce outputs which are plausible images of the target style, and
- we'd like it to preserve the structure of the original image.

Think how many layers of abstraction we have here. The CycleGAN is composed of multiple neural nets, which are composed of layers, which are composed of units and connections, which compute simple arithmetic operations. Moving between layers of abstraction is part of being a good computer scientist.

We satisfy the first criterion using the GAN generator objective, which in this context is termed the **discriminator cost**. I.e., we train a discriminator network to distinguish between outputs of the first generator and training images from Style B, and then another discriminator to do the same for Style A. In order to satisfy the second criterion, we impose a **cycle consistency cost**, or **reconstruction cost**. Observe that if both generators preserve the structure, and you run both in sequence, you should get back the original image. The reconstruction cost penalizes the squared error in reconstructing the original image, i.e. $\|\mathbf{x} - G_2(G_1(\mathbf{x}))\|^2$. This architecture is shown in Figure 6.

You'll get a chance to implement this architecture for Assignment 4. You can find lots cool examples of style transfer here:

`https://github.com/junyanz/CycleGAN`

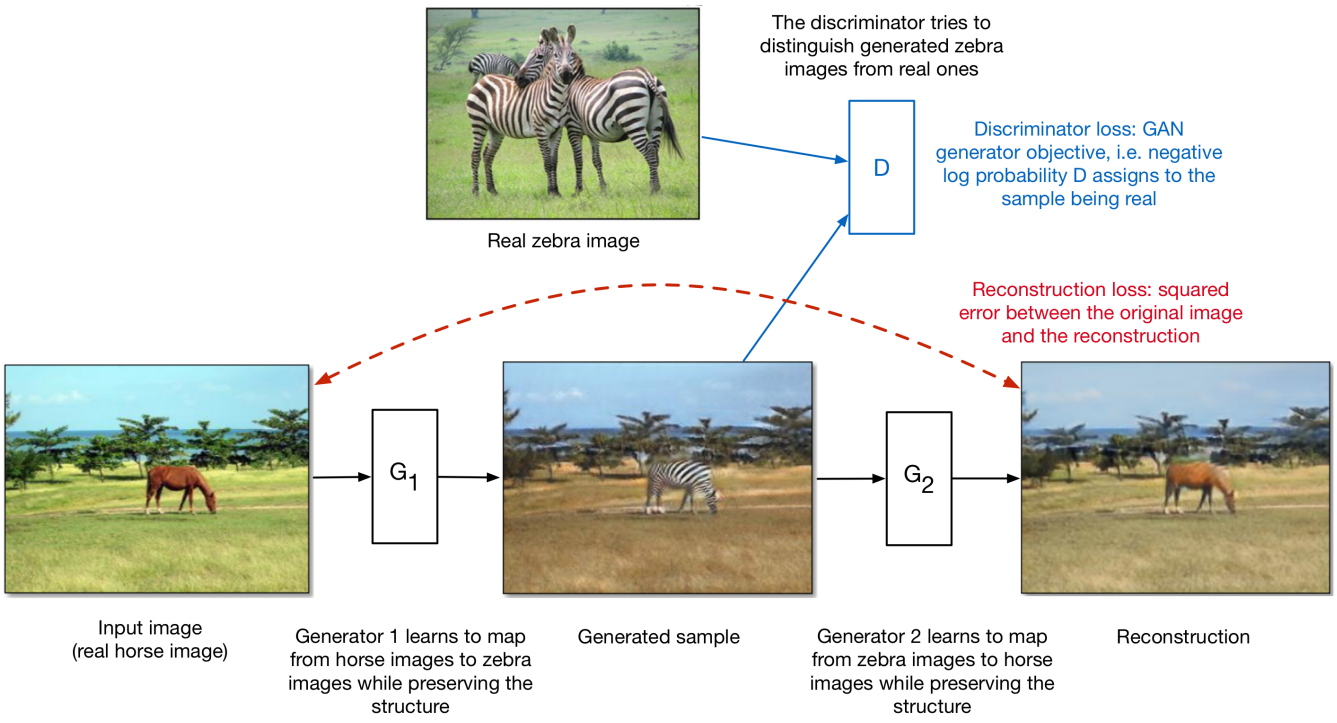


Figure 6: The CycleGAN architecture.