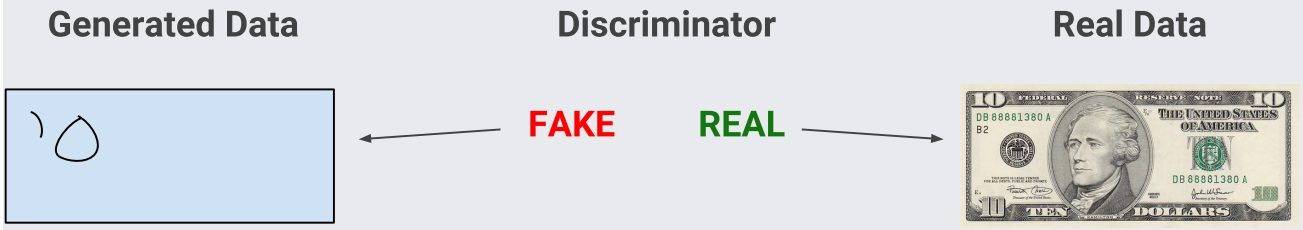
Overview of GAN Structure

A generative adversarial network (GAN) has two parts:

- The **generator** learns to generate plausible data. The generated instances become negative training examples for the discriminator.
- The **discriminator** learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

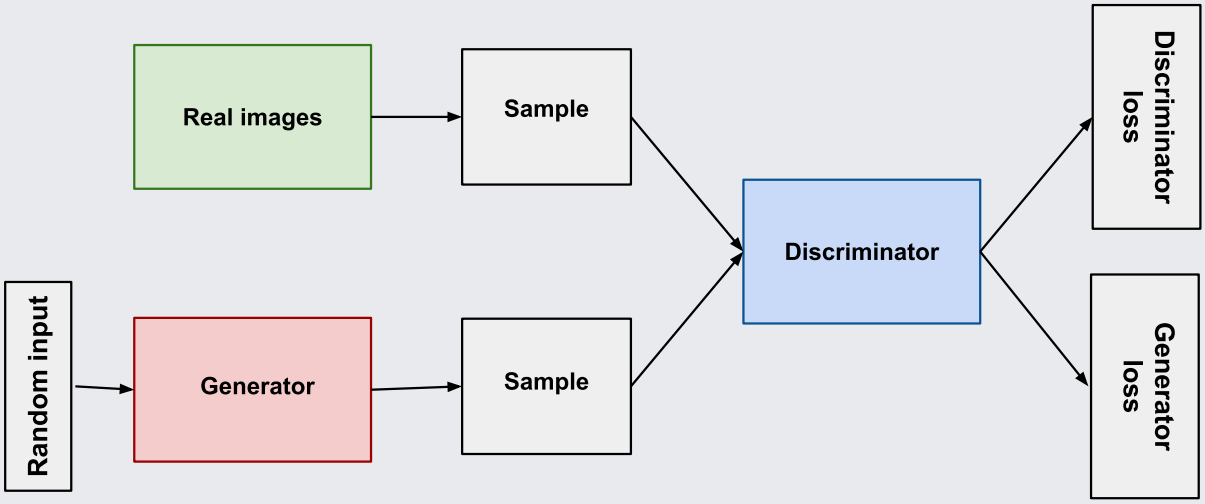
When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake:



As training progresses, the generator gets closer to producing output that can fool the discriminator:

Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

Here's a picture of the whole system:



Both the generator and the discriminator are neural networks. The generator output is connected directly to the discriminator input. Through backpropagation (/machine-learning/glossary#backpropagation), the discriminator's classification provides a signal that the generator uses to update its weights.

Let's explain the pieces of this system in greater detail.

Previous
← Generative Models (/machine-learning/gan/generative)

Next
Discriminator (/machine-learning/gan/discriminator) →

The Discriminator

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying.

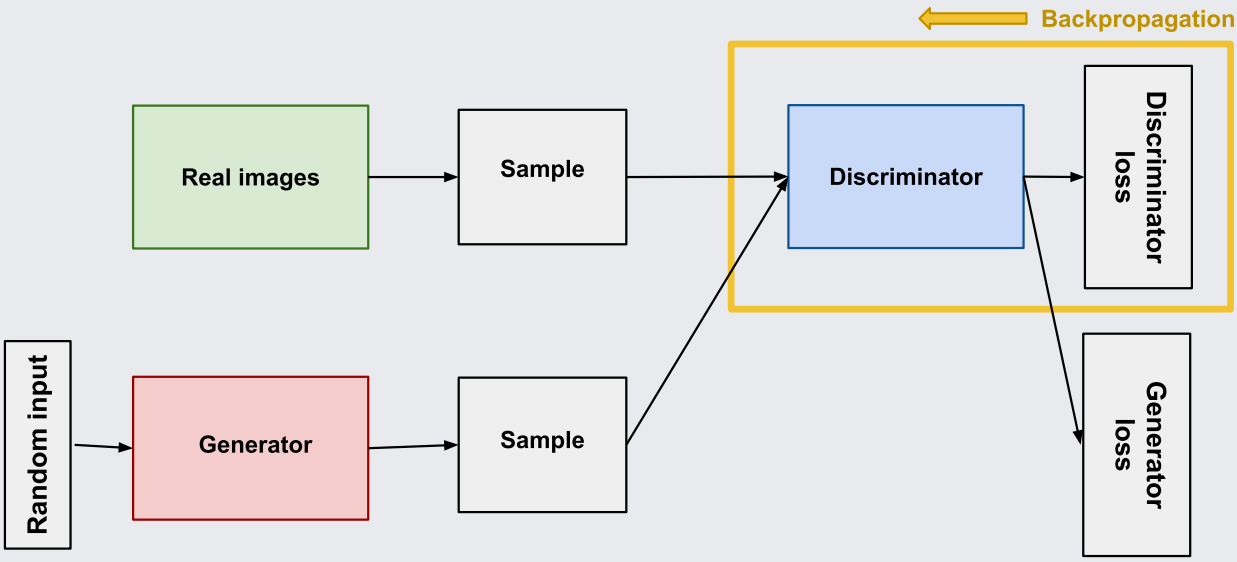


Figure 1: Backpropagation in discriminator training.

Discriminator Training Data

The discriminator's training data comes from two sources:

- **Real data** instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.
- **Fake data** instances created by the generator. The discriminator uses these instances as negative examples during training.

In Figure 1, the two "Sample" boxes represent these two data sources feeding into the discriminator. During discriminator training the generator does not train. Its weights remain constant while it produces examples for the discriminator to train on.

Training the Discriminator

The discriminator connects to two loss (/machine-learning/glossary#loss) functions. During discriminator training, the discriminator ignores the generator loss and just uses the

discriminator loss. We use the generator loss during generator training, as described in the next section (/machine-learning/gan/generator).

During discriminator training:

1. The discriminator classifies both real data and fake data from the generator.
2. The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.
3. The discriminator updates its weights through backpropagation (https://developers.google.com/machine-learning/glossary/#b) from the discriminator loss through the discriminator network.

In the next section we'll see why the generator loss connects to the discriminator.

Previous
← Overview of GAN Structure (/machine-learning/gan/gan_structure)

Next
Generator (/machine-learning/gan/generator) →

Last updated 2022-07-18 UTC.

The Generator

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes:

- random input
- generator network, which transforms the random input into a data instance
- discriminator network, which classifies the generated data
- discriminator output
- generator loss, which penalizes the generator for failing to fool the discriminator

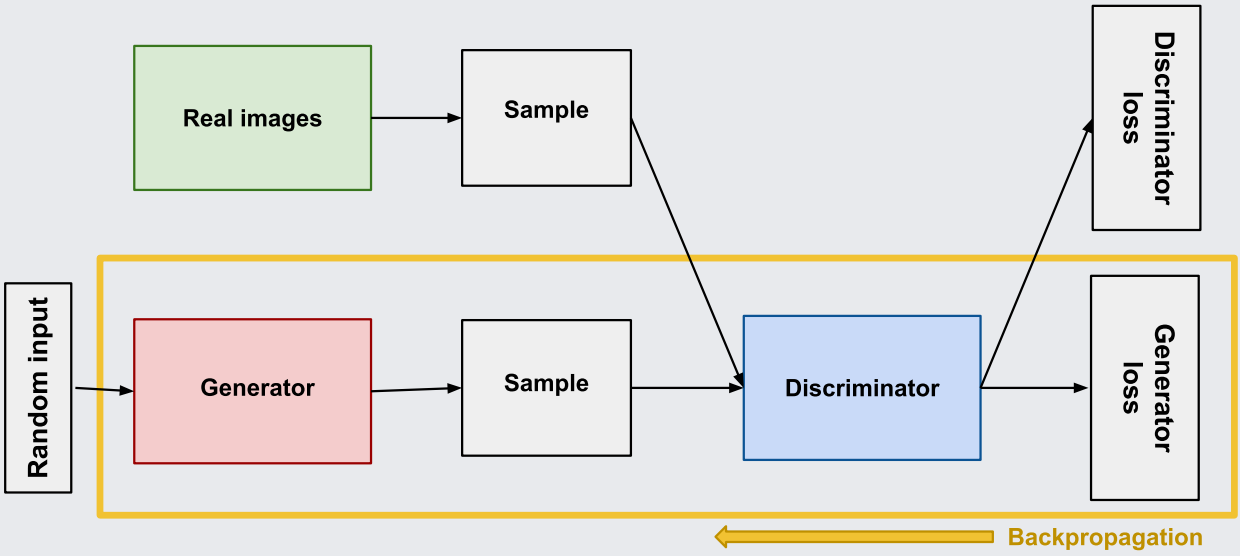


Figure 1: Backpropagation in generator training.

Random Input

Neural networks need some form of input. Normally we input data that we want to do something with, like an instance that we want to classify or make a prediction about. But what do we use as input for a network that outputs entirely new data instances?

In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output. By introducing noise, we can get the GAN to produce a wide variety of data, sampling from different places in the target distribution.

Experiments suggest that the distribution of the noise doesn't matter much, so we can choose something that's easy to sample from, like a uniform distribution. For convenience the space from which the noise is sampled is usually of smaller dimension than the dimensionality of the output space.

Note: Some GANs use non-random input to shape the output. See GAN Variations (/machine-learning/gan/applications).

Using the Discriminator to Train the Generator

To train a neural net, we alter the net's weights to reduce the error or loss of its output. In our GAN, however, the generator is not directly connected to the loss that we're trying to affect. The generator feeds into the discriminator net, and the *discriminator* produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

This extra chunk of network must be included in backpropagation. Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output — how the output would change if you changed the weight. But the impact of a generator weight depends on the impact of the discriminator weights it feeds into. So backpropagation starts at the output and flows back through the discriminator into the generator.

At the same time, we don't want the discriminator to change during generator training. Trying to hit a moving target would make a hard problem even harder for the generator.

So we train the generator with the following procedure:

1. Sample random noise.
2. Produce generator output from sampled random noise.
3. Get discriminator "Real" or "Fake" classification for generator output.
4. Calculate loss from discriminator classification.
5. Backpropagate through both the discriminator and generator to obtain gradients.
6. Use gradients to change only the generator weights.

This is one iteration of generator training. In the next section we'll see how to juggle the training of both the generator and the discriminator.

Previous
← Discriminator (/machine-learning/gan/discriminator)

Next
GAN Training (/machine-learning/gan/training) →

GAN Training

Because a GAN contains two separately trained networks, its training algorithm must address two complications:

- GANs must juggle two different kinds of training (generator and discriminator).
- GAN convergence is hard to identify.

Alternating Training

The generator and the discriminator have different training processes. So how do we train the GAN as a whole?

GAN training proceeds in alternating periods:

1. The discriminator trains for one or more epochs.
2. The generator trains for one or more epochs.
3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks.

We keep the generator constant during the discriminator training phase. As discriminator training tries to figure out how to distinguish real data from fake, it has to learn how to recognize the generator's flaws. That's a different problem for a thoroughly trained generator than it is for an untrained generator that produces random output.

Similarly, we keep the discriminator constant during the generator training phase. Otherwise the generator would be trying to hit a moving target and might never converge.

It's this back and forth that allows GANs to tackle otherwise intractable generative problems. We get a toehold in the difficult generative problem by starting with a much simpler classification problem. Conversely, if you can't train a classifier to tell the difference between real and generated data even for the initial random generator output, you can't get the GAN training started.

Convergence

As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. If the generator

succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator flips a coin to make its prediction.

This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse.

For a GAN, convergence is often a fleeting, rather than stable, state.

Previous
← Generator (/machine-learning/gan/generator)

Next
Loss Functions (/machine-learning/gan/loss) →

Loss Functions

GANs try to replicate a probability distribution. They should therefore use loss functions that reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data.

How do you capture the difference between two distributions in GAN loss functions? This question is an area of active research, and many approaches have been proposed. We'll address two common GAN loss functions here, both of which are implemented in TF-GAN:

- **minimax loss**: The loss function used in the paper that introduced GANs (https://arxiv.org/abs/1406.2661).
- **Wasserstein loss**: The default loss function for TF-GAN Estimators. First described in a 2017 paper (https://arxiv.org/abs/1701.07875).

TF-GAN implements many other loss functions as well.

One Loss Function or Two?

A GAN can have two loss functions: one for generator training and one for discriminator training. How can two loss functions work together to reflect a distance measure between probability distributions?

In the loss schemes we'll look at here, the generator and discriminator losses derive from a single measure of distance between probability distributions. In both of these schemes, however, the generator can only affect one term in the distance measure: the term that reflects the distribution of the fake data. So during generator training we drop the other term, which reflects the distribution of the real data.

The generator and discriminator losses look different in the end, even though they derive from a single formula.

Minimax Loss

In the paper that introduced GANs, the generator tries to minimize the following function while the discriminator tries to maximize it:

$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

In this function:

- `D(x)` is the discriminator's estimate of the probability that real data instance x is real.
- E_x is the expected value over all real data instances.
- `G(z)` is the generator's output when given noise z.
- `D(G(z))` is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
- The formula derives from the cross-entropy (/machine-learning/glossary#cross-entropy) between the real and generated distributions.

The generator can't directly affect the `log(D(x))` term in the function, so, for the generator, minimizing the loss is equivalent to minimizing `log(1 - D(G(z)))`.

In TF-GAN, see minimax_discriminator_loss and minimax_generator_loss (https://github.com/tensorflow/gan/blob/master/tensorflow_gan/python/losses/losses_impl.py) for an implementation of this loss function.

Modified Minimax Loss

The original GAN paper notes that the above minimax loss function can cause the GAN to get stuck in the early stages of GAN training when the discriminator's job is very easy. The paper therefore suggests modifying the generator loss so that the generator tries to maximize `log D(G(z))`.

In TF-GAN, see modified_generator_loss (https://github.com/tensorflow/tensorflow/blob/2007e1ba474030fcce840b0b8a599558e7d5998f/tensorflow/contrib/gan/python/losses/python/losses_impl.py) for an implementation of this modification.

Wasserstein Loss

By default, TF-GAN uses Wasserstein loss (https://arxiv.org/abs/1701.07875).

This loss function depends on a modification of the GAN scheme (called "Wasserstein GAN" or "WGAN") in which the discriminator does not actually classify instances. For each instance it outputs a number. This number does not have to be less than one or greater

than 0, so we can't use 0.5 as a threshold to decide whether an instance is real or fake. Discriminator training just tries to make the output bigger for real instances than for fake instances.

Because it can't really discriminate between real and fake, the WGAN discriminator is actually called a "critic" instead of a "discriminator". This distinction has theoretical importance, but for practical purposes we can treat it as an acknowledgement that the inputs to the loss functions don't have to be probabilities.

The loss functions themselves are deceptively simple:

Critic Loss: `D(x) - D(G(z))`

The discriminator tries to maximize this function. In other words, it tries to maximize the difference between its output on real instances and its output on fake instances.

Generator Loss: `D(G(z))`

The generator tries to maximize this function. In other words, It tries to maximize the discriminator's output for its fake instances.

In these functions:

- `D(x)` is the critic's output for a real instance.
- `G(z)` is the generator's output when given noise z.
- `D(G(z))` is the critic's output for a fake instance.
- The output of critic D does *not* have to be between 1 and 0.
- The formulas derive from the earth mover distance (https://wikipedia.org/wiki/Earth_mover%27s_distance) between the real and generated distributions.

In TF-GAN, see wasserstein_generator_loss and wasserstein_discriminator_loss (https://github.com/tensorflow/gan/blob/master/tensorflow_gan/python/losses/losses_impl.py) for implementations.

Requirements

The theoretical justification for the Wasserstein GAN (or WGAN) requires that the weights throughout the GAN be clipped so that they remain within a constrained range.

Benefits

Wasserstein GANs are less vulnerable to getting stuck than minimax-based GANs, and avoid problems with vanishing gradients. The earth mover distance also has the advantage of being a true metric: a measure of distance in a space of probability distributions. Cross-entropy is not a metric in this sense.

Previous
← GAN Training (/machine-learning/gan/training)

Next
Check Your Understanding (/machine-learning/gan/check) →

Check Your Understanding: GAN Anatomy

True or false: the discriminator network and generator network influence each other solely through the data produced by the generator and the labels produced by the discriminator. When it comes to backpropagation, they are separate networks.

True

False ✓

Correct: during generator training, gradients propagate through the discriminator network to the generator network (although the discriminator does not update its weights during generator training). So the weights in the discriminator network influence the updates to the generator network.

Correct answer.

True or false: a typical GAN trains the generator and the discriminator simultaneously.

False ✓

Correct. A typical GAN alternates between training the discriminator and training the generator.

Correct answer.

True

True or false: a GAN always uses the same loss function for both discriminator and generator training.

False ✓

Correct. While it's possible for a GAN to use the same loss for both generator and discriminator training (or the same loss differing only in sign), it's not required. In fact it's more common to use different losses for the discriminator and the generator.

Correct answer.

True

Previous
Loss Functions (/machine-learning/gan/loss)

Next
Common Problems (/machine-learning/gan/problems)

Last updated 2022-07-18 UTC.