freeCodeCamp(^)   Forum   Donate

Learn to code — free 3,000-hour curriculum

JANUARY 7, 2018 / #MACHINE LEARNING

An intuitive introduction to Generative Adversarial Networks (GANs)

freeCodeCamp(♨)   Forum   Donate

Learn to code — free 3,000-hour curriculum

Warm up

Let's say there's a very cool party going on in your neighborhood that
you really want to go to. But, there is a problem. To get into the party
you need a special ticket — that was long sold out.

Wait up! Isn't this a Generative Adversarial Networks article? Yes it is.
But bear with me for now, it is going to be worth it.

OK, since expectations are very high, the party organizers hired a
qualified security agency. Their primary goal is to not allow anyone to
crash the party. To do that, they placed a lot of guards at the venue's
entrance to check everyone's tickets for authenticity.

Since you don't have any martial artistic gifts, the only way to get
through is by fooling them with a **very convincing** fake ticket.

There is a big problem with this plan though — you never actually saw
how the ticket looks like.

Even if you design a ticket based on your creativity, it's almost
impossible to fool the guards at your first trial. Besides, you can't show
your face until you have a very decent replica of the party's pass.

To help solve the problem, you decide to call your friend Bob to do the
job for you.

Bob's mission is very simple. He will try to get into the party with your
fake pass. If he gets denied, he will come back to you with useful tips

freeCodeCamp(🔥)   Forum   Donate
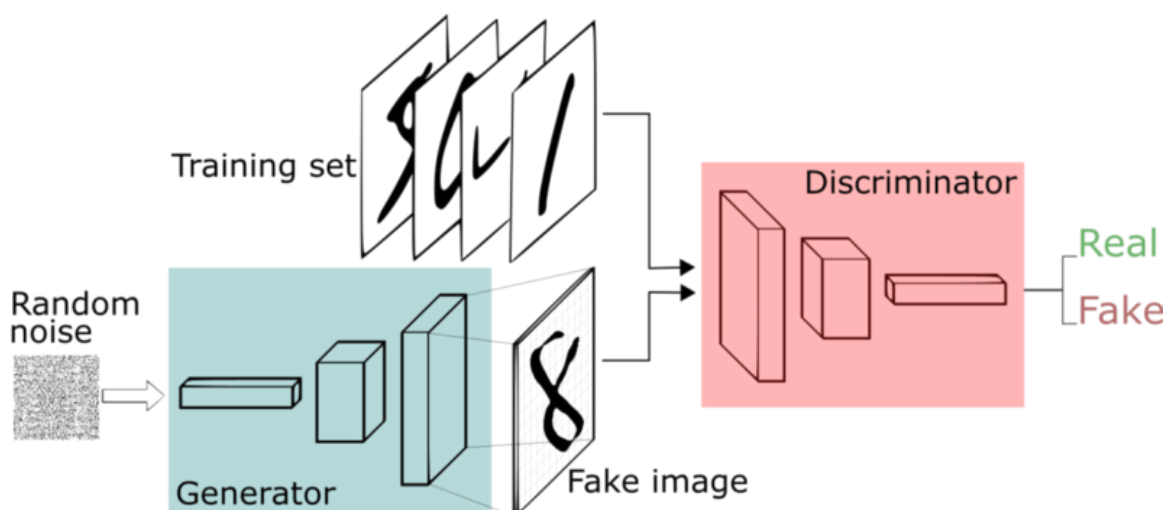
Learn to code — free 3,000-hour curriculum

hand it to Bob, who goes to try again. This process keeps repeating until you become able to design a perfect replica.

Putting aside the 'small holes' in this anecdote, this is pretty much how Generative Adversarial Networks (GANs) work.

Nowadays, most of the applications of GANs are in the field of computer vision. Some of the applications include training semi-supervised classifiers, and generating high resolution images from low resolution counterparts.

This piece provides an introduction to GANs with a hands-on approach to the problem of generating images. You can clone the notebook for this post here.

Generative Adversarial Networks


Generative Adversarial Network framework.

the discriminator) have different roles in this framework.

The generator tries to produce data that come from some probability distribution. That would be you trying to reproduce the party's tickets.

The discriminator acts like a judge. It gets to decide if the input comes from the generator or from the true training set. That would be the party's security comparing your fake ticket with the true ticket to find flaws in your design.

Learn to code — free 3,000-hour curriculum
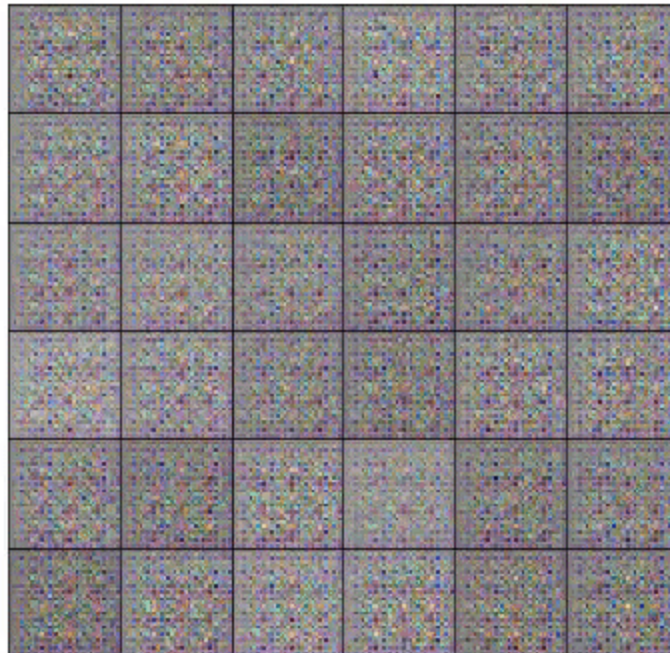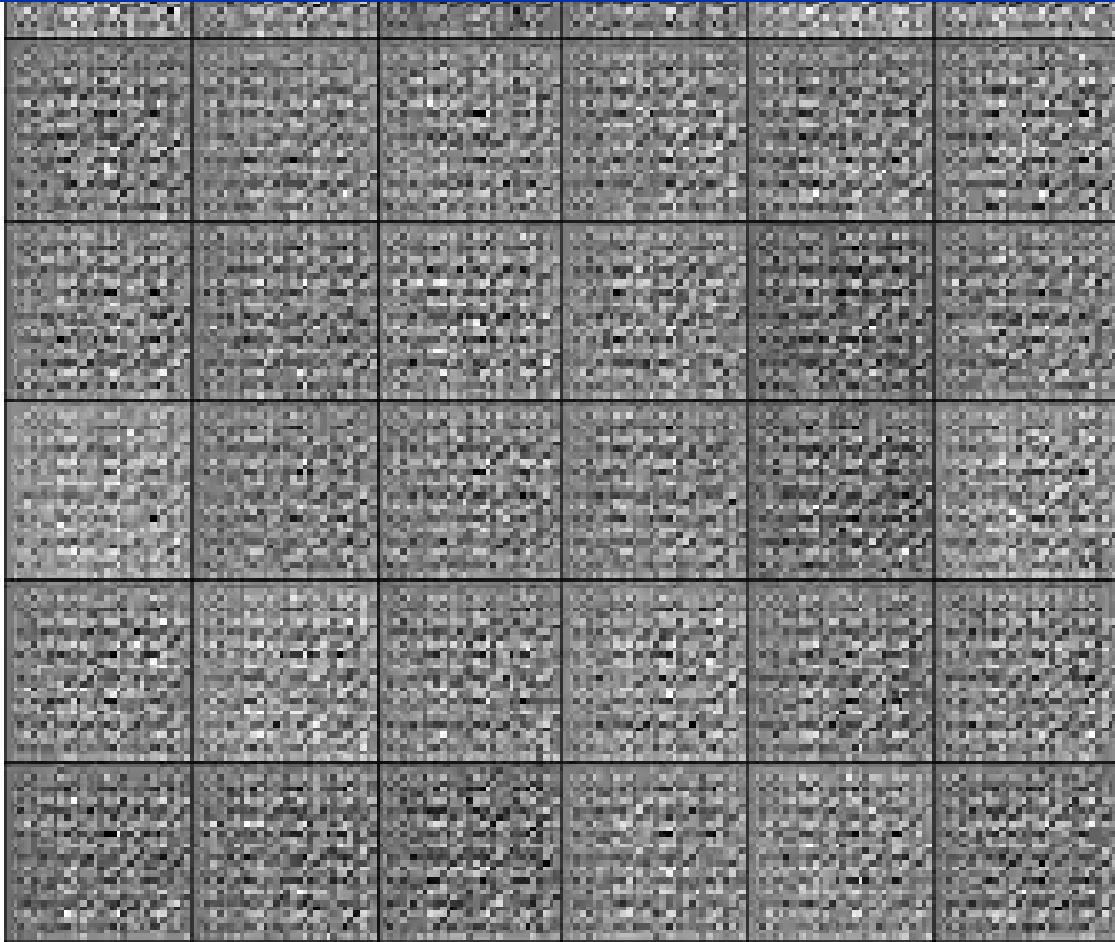
We used a 4 layer convolution network for (both discriminator and generator) with batch normalization. The model was trained to generate SVHNs and MNIST images. Above, are the SVHN's (left) and MNIST (right) generator samples during training.

In summary, the game follows with:

- The generator trying to maximize the probability of making the discriminator mistakes its inputs as real.

- And the discriminator guiding the generator to produce more realistic images.
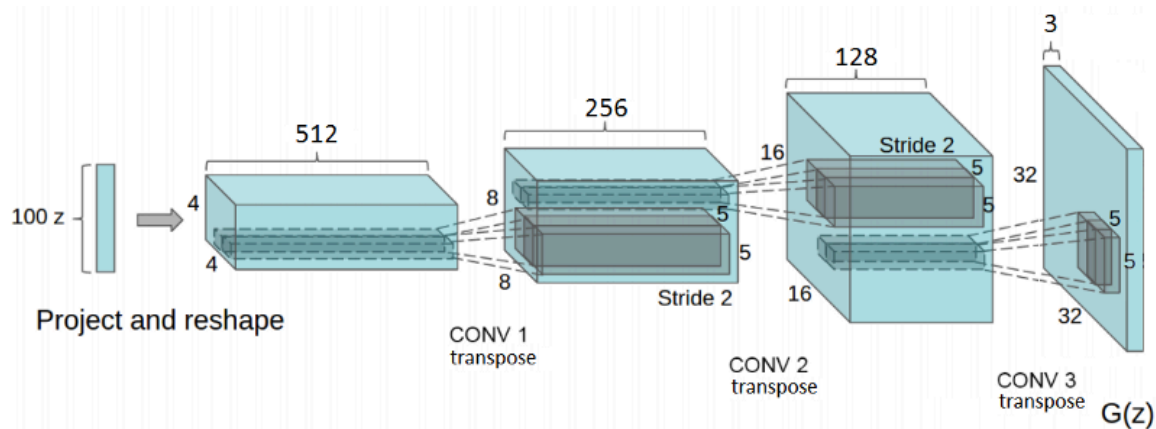
Adapted from the DCGAN paper. The Generator network implemented here. Note the non-existence of fully connected and pooling layers.

In the DCGAN paper, the authors describe the combination of some deep learning techniques as key for training GANs. These techniques include: (i) the all convolutional net and (ii) Batch Normalization (BN).

The first emphasizes *strided convolutions* (instead of pooling layers) for both: increasing and decreasing feature's spatial dimensions. And the second normalizes the feature vectors to have zero mean and unit variance in all layers. This helps to stabilize learning and to deal with poor weight initialization problems.

Without further ado, let's dive into the implementation details and talk more about GANs as we go. We present an implementation of a Deep Convolutional Generative Adversarial Network (DCGAN). Our implementation uses Tensorflow and follows some practices described in the DCGAN paper.

the output layer) and Rectified Linear unit (ReLU) activations.

It takes as an input a random vector *z* (drawn from a normal distribution). After reshaping *z* to have a 4D shape, we feed it to the generator that starts a series of upsampling layers.

Each upsampling layer represents a transpose convolution operation with strides 2. Transpose convolutions are similar to the regular convolutions.

Typically, regular convolutions go from wide and shallow layers to narrower and deeper ones. Transpose convolutions go the other way. They go from deep and narrow layers to wider and shallower.

The stride of a transpose convolution operation defines the size of the output layer. With "same" padding and stride of 2, the output features will have double the size of the input layer.

That happens because, every time we move one pixel in the input layer, we move the convolution kernel by two pixels on the output layer. In other words, each pixel in the input image is used to draw a square in the output image.

Learn to code — free 3,000-hour curriculum
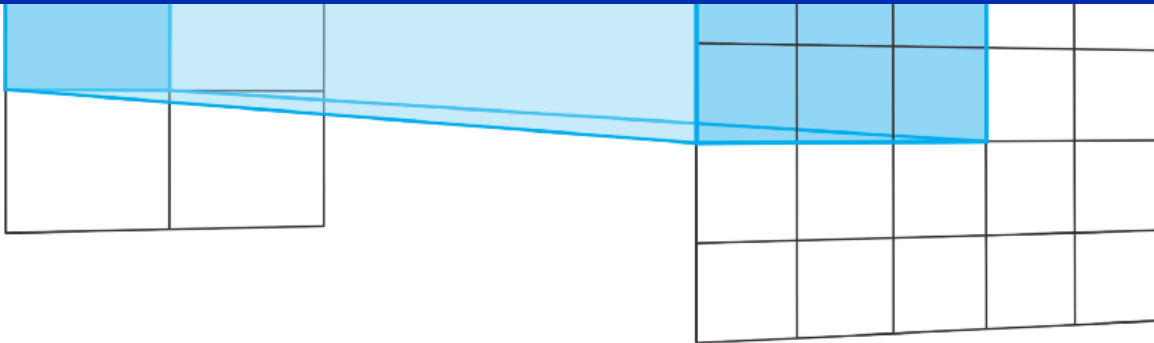
Transpose convolving a 3x3 kernel over a 2x2 input with stride 2 is equivalent to convolving a 3x3 kernel over a 5x5 input with stride 2. Using no padding "VALID", for both.

In short, the generator begins with this very deep but narrow input vector. After each transpose convolution, *z* becomes wider and shallower. All transpose convolutions use a *5x5* kernel's size with depths reducing from 512 all the way down to 3 — representing an RGB color image.

```
def transpose_conv2d(x, output_space):    return tf.layers.conv2d_t
```

The final layer outputs a *32x32x3* tensor — squashed between values of -1 and 1 through the Hyperbolic Tangent (*tanh*) function.

This final output shape is defined by the size of the training images. In this case, if training for SVHN, the generator produces *32x32x3* images. However, if training for MNIST, it would generate a *28x28* greyscale image.

Learn to code — free 3,000-hour curriculum

```
def generator(z, output_dim, reuse=False, alpha=0.2, training=True)
```
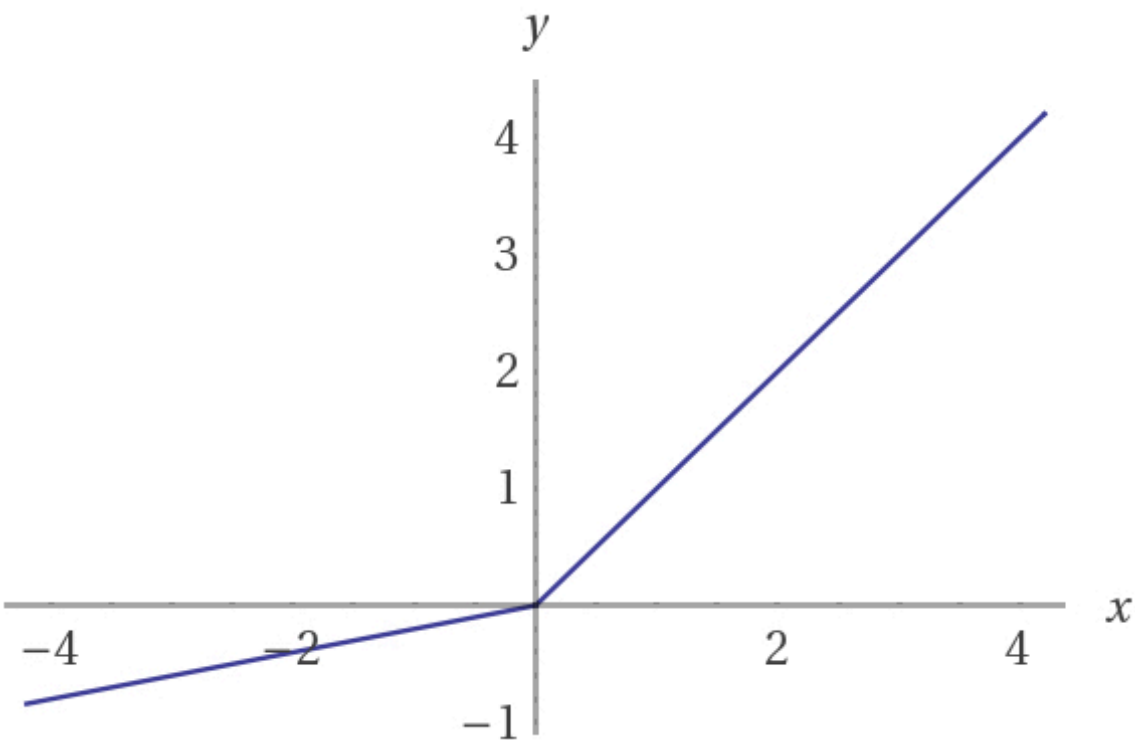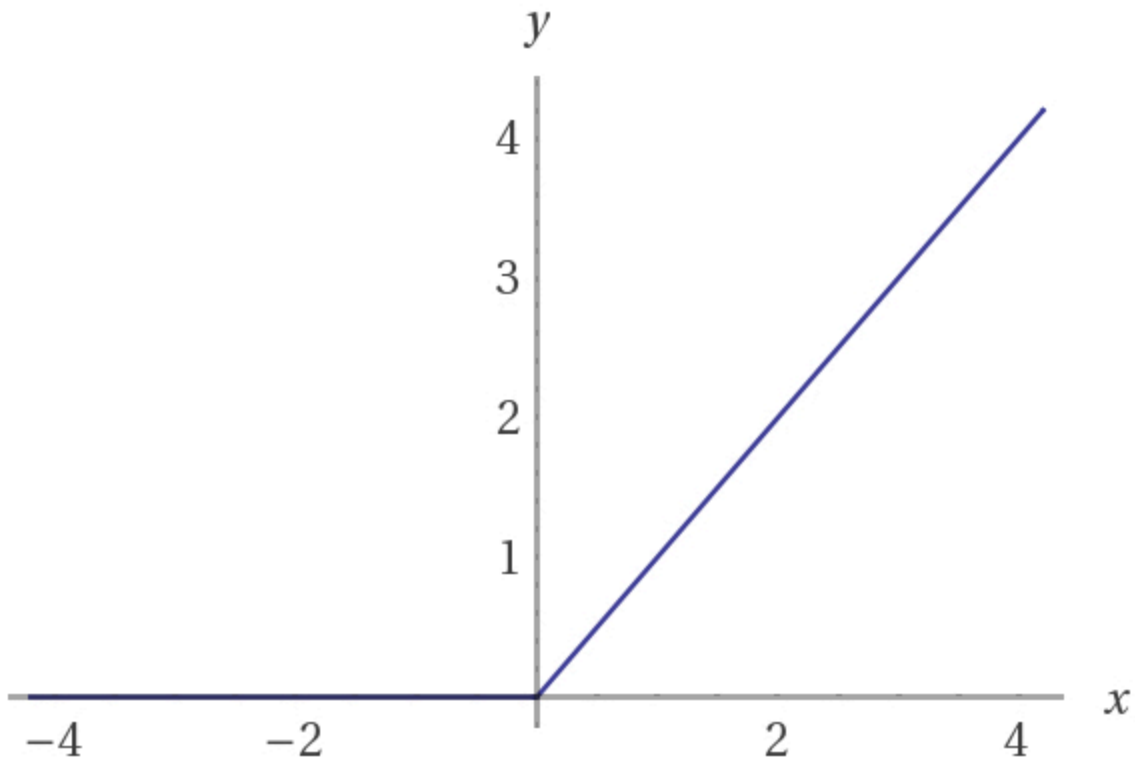
Discriminator

The discriminator is also a 4 layer CNN with BN (except its input layer) and leaky ReLU activations. Many activation functions will work fine with this basic GAN architecture. However, leaky ReLUs are very popular because they help the gradients flow easier through the architecture.

A regular ReLU function works by truncating negative values to 0. This has the effect of blocking the gradients to flow through the network. Instead of the function being zero, leaky ReLUs allow a small negative value to pass through. **That is, the function computes the greatest value between the features and a small factor.**

```
def lrelu(x, alpha=0.2):     # non-linear activation function     re
```

Leaky ReLUs represent an attempt to solve the *dying ReLU* problem. This situation occurs when the neurons get stuck in a state in which *ReLU* units always output 0s for all inputs. For these cases, the gradients are completely shut to flow back through the network.

Learn to code — free 3,000-hour curriculum

The discriminator starts by receives a 32x32x3 image tensor. Opposite to the generator, the discriminator performs a series of *strided 2* convolutions. Each, works by reducing the feature vector's spatial dimensions by half its size, also doubling the number of learned filters.

Finally, the discriminator needs to output probabilities. For that, we use the *Logistic Sigmoid* activation function on the final logits.

```
def discriminator(x, reuse=False, alpha=0.2, training=True):    """
```

Note that in this framework, the discriminator acts as a regular binary classifier. Half of the time it receives images from the training set and the other half from the generator.

Back to our adventure, to reproduce the party's ticket, the only source of information you had was the feedback from our friend Bob. In other words, the quality of the feedback Bob provided to you at each trial was essential to get the job done.

In the same way, every time the discriminator notices a difference between the real and fake images, it sends a signal to the generator. This signal is the gradient that flows backwards from the discriminator to the generator. By receiving it, the generator is able to adjust its parameters to get closer to the true data distribution.

Losses

Now, let's describe the trickiest part of this architecture — the losses. First, we know the discriminator receives images from both the training set and the generator.

We want the discriminator to be able to distinguish between real and fake images. Every time we run a mini-batch through the discriminator, we get logits. These are the unscaled values from the model.

However, we can divide the mini-batches that the discriminator receives in two types. The first, composed only with real images that come from the training set and the second, with only fake images — the ones created by the generator.

```
def model_loss(input_real, input_z, output_dim, alpha=0.2, smooth=0
```

Because both networks train at the same time, GANs also need two optimizers. Each one for minimizing the discriminator and generator's loss functions respectively.

We want the discriminator to output probabilities close to 1 for real images and near 0 for fake images. To do that, the discriminator needs two losses. Therefore, the total loss for the discriminator is the sum of these two partial losses. **One for maximizing the probabilities for the**

**Learn to code — free 3,000-hour curriculum**
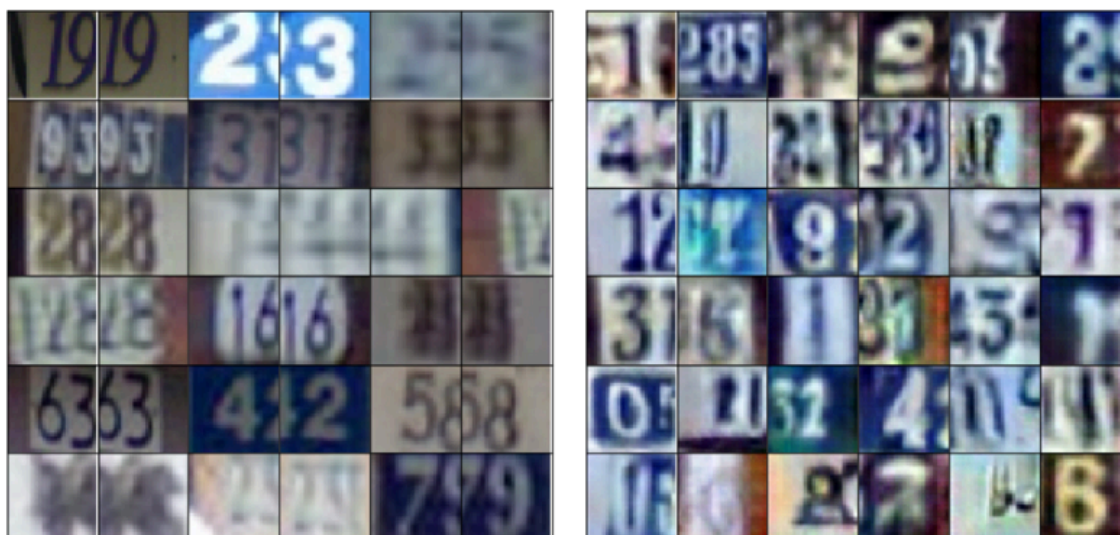
Comparing real (left) and generated (right) SVHN sample images. Although some images look blurred and some others are difficult to recognize, it's noticeable that the data distribution was captured by the model.

In the beginning of training two interesting situations occur. First, the generator does not know how to create images that resembles the ones from the training set. And second, discriminator does not know how to categorize the images it receives as real or fake.

As a result, the discriminator receives two very distinct types of batches. One, composed of true images from the training set and another containing very noisy signals. As training progresses, the generator starts to output images that look closer to the images from the training set. That happens, because the generator trains to learn the data distribution that composes the training set images.

At the same time, the discriminator starts to get real good at classifying samples as real or fake. As a consequence, the two types of

For the losses, we use vanilla cross-entropy with Adam as a good choice for the optimizer.
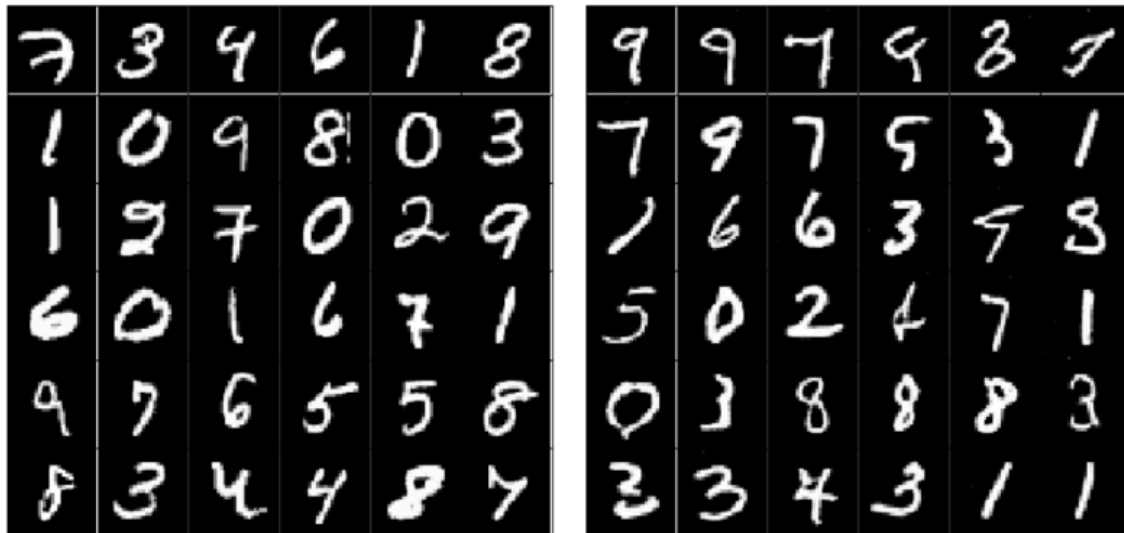
Comparing real (left) and generated (right) MNIST sample images. Because MNIST images have a simpler data structure, the model was able to produce more realistic samples when compared to the SVHNs.

Concluding

GANs are one of the hottest subjects in machine learning right now. These models have the potential of unlocking unsupervised learning methods that would expand ML to new horizons.

Since its creation, researches have been developing many techniques for training GANs. In Improved Techniques for Training GANs, the authors describe state-of-the-art techniques for both image generation and semi-supervised learning.

Learn to code — free 3,000-hour curriculum

Also, take a look at:

Dive head first into advanced GANs: exploring self-attention and spectral norm
Lately, Generative Models are drawing a lot of attention. Much of that comes from Generative Adversarial Networks…
medium.freecodecamp.org**Semi-supervised learning with Generative Adversarial Networks (GANs)**
If you ever heard or studied about deep learning, you probably heard about MNIST, SVHN, ImageNet, PascalVoc and others…
towardsdatascience.com

And if you need more, that is my deep learning blog.

Enjoy, and thanks for reading!

One clap, two clap, three clap, forty?
By clapping more or less, you can signal to us which stories really stand out.

👏 470   💬 1 🐦 ⓕ 🔖 ⋯

Credits to Sam Williams for this awesome "clap" gif! Check it out in his post.

freeCodeCamp(🔥)

Donate

Learn to code — <u>free 3,000-hour curriculum</u>

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can <u>make a tax-deductible donation here</u>.

Trending Guides

Date Formatting in JS	Java Iterator Hashmap	Cancel a Merge in Git
What is a Linked List?	Install Java in Ubuntu	Python Ternary Operator

freeCodeCamp(🔥)

Forum   Donate

Learn to code — free 3,000-hour curriculum

Pandas Count Rows	Python End Program	Python XOR Operator
Python Dict Has Key	Python List to String	Exit Function in Python
String to Array in Java	Python Import from File	Parse a String in Python
Python Merge Dictionaries	Copy a Directory in Linux	Reactive Programming Guide
Center Text Vertically CSS	What's a Greedy Algorithm?	Edit Commit Messages in Git

Mobile App

Our Charity

About   Alumni Network   Open Source   Shop   Support   Sponsors   Academic Honesty

Code of Conduct   Privacy Policy   Terms of Service   Copyright Policy