Generative Adversarial Networks (GANs)

An Illustrative Tutorial

Intro

Recently GANs have garnered a lot of attention in the machine learning community. This tutorial provides a (hopefully) simple, intuitive, and illuminating look at how GANs work "under the hood" and how to implement them using Keras, a Python deep-learning library that runs on TensorFlow.

GANs at a Glance…

What ARE GANs even? *A bunch of math equations? A machine learning model? A single line of code I can call from scikit-learn to finally win a Kaggle competition?*

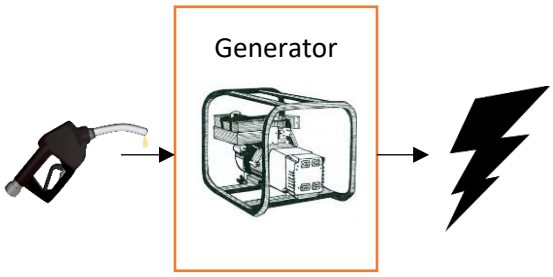
GANs are nothing more than a "framework for estimating generative models" (taken straight from the first sentence of the Goodfellow et. al. paper). That's probably not a very satisfying answer, it's a very high-level definition that barely describes GANs. But in a nutshell that's all GANs are. Think of them as a blueprint or a process that you can use to estimate a generator. Much like there are many ways to prepare for an exam (pull an all-nighter, pace yourself and study a little each day, or cheat), there are many ways to estimate a generator. GANs are just one of those "ways".

ASIDE

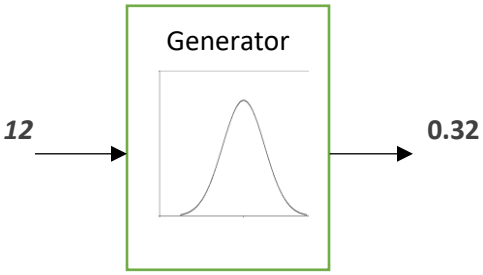
Hold up, you lost me at "generator"… like a machine that produces electricity..?

Sort of, a generator is any "thing" that take an input and produces (or generates) an output. In the machine learning context we're usually referring to some sort of probability distribution. When you give it a real number in its domain, it outputs a probability. Sometimes generators are called generative models too.

Not what we're talking about:



What we're talking about:



Let's go through the acronym one word at a time to get a clear understanding of its component parts.

Generative

Let's consider standard classification models. Generally, these fall into one of two categories: discriminative or generative. Generative models (or generators) attempt to "learn" *how* the data in question is created (or generated). The idea is that if the model can learn how the data is generated then it can use that information for classification. On the other hand, discriminative models (or discriminators) focus simply on learning the best way to categorize the data in question, paying no attention to the generation process.

In probabilistic terms, generative models are concerned with P(X, Y) – "the probability of X and Y", while discriminative models focus on P(Y | X) – "the probability of Y given X". Here Y is a random variable that defines a class and X is a feature vector.

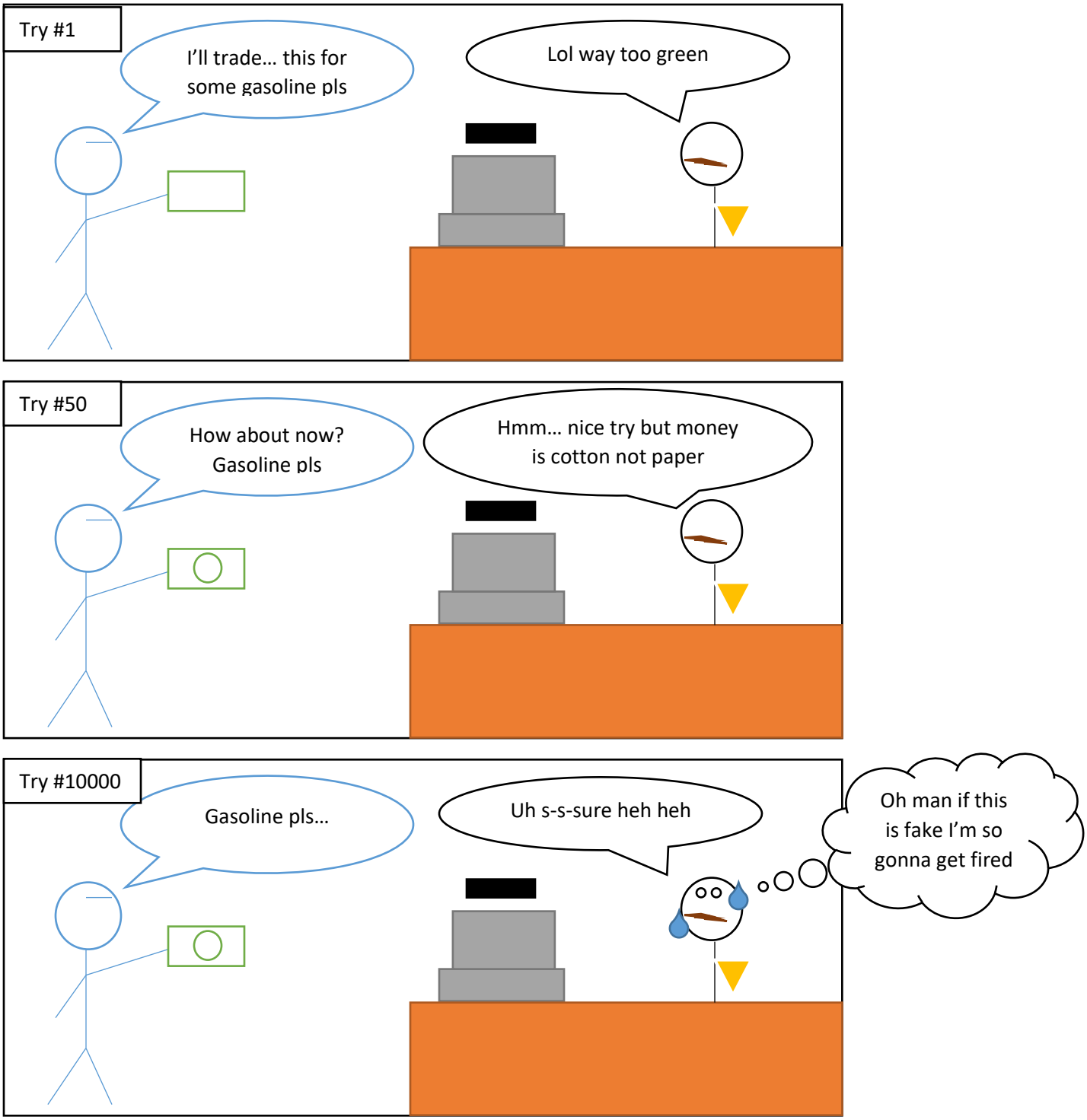
At the end of the day, the goal of GANs is to estimate a data generating distribution via a generative model. This isn't a completely new and exciting idea by itself though. There already exist many generative models such as Naïve Bayes, Latent Dirichlet Allocation, Hidden Markov Models, etc. The interesting part is *how* the generator is estimated. And that brings us to the next word.

Adversarial

The GANs framework not only consists of a generator but also a discriminator. Recall that discriminative models are usually used for classification purposes – they allow one to map an input (usually a vector) to a class or label from a set of possible labels. In the context of GANs, the generator iteratively learns a distribution by creating "fake" samples from noise and then showing them to the discriminator. The discriminator (which knows the true distribution the generator is trying to fake) sees the "fake" and has to decide whether or not it came from the true distribution. The goal of the generator is to create such good "fakes" that the discriminator can't dependably determine whether or not what it's presented with is real or fake.

Notice that in this scenario not only is the generator learning from the discriminator's feedback, but the discriminator is learning what the "fake" samples produced from the generator are too! In other words, the discriminator is trained in such a way to maximize the probability that it classifies input as real (from the true distribution) or fake (from the generator). The metaphorical tug-of-war between the generator and discriminator naturally forces the generator to incrementally change its output to match the true distribution.

Kinda hard to visualize huh? The go-to analogy is to think of the generator as a team of counterfeiters and the discriminator as the police. Imagine this scenario: the generator counterfeits a fake $100 bill and tries to use it at the local gas station. Unbeknownst to them, the cashier is an undercover cop. The first time the generator tries to use its fake money, the undercover cop laughs and lets the generator know that their $100 bill is way too green to be real. Taking this information, the generator goes back to their secret counterfeiting lab and creates a revised version of the fake $100 bill. This process continues until the undercover cop can't determine whether or not the counterfeiter's money is real or fake!



Maybe a better analogy is how you change your behavior and appearance in order to trick society into thinking that you're a functional member like everyone else. So when the McDonald's employees kick

you out for ordering 100 apple pies with big mac sauce and making a scene when they charge you extra, you learn that you're not putting on a convincing enough act and resign yourself to only 2 plain apple pies in the future. In the same way, the generator learns and alters its parameters accordingly.

(Just trying to inject some humor into this soul-sucking project. Not a personal attack or anything.)

Networks

The networks portion of the GANs acronym comes from its natural application in multilayer perceptron models e.g. neural networks.

Putting it all together

To sum up, the GANs framework simultaneously trains a generator to trick its adversary, a discriminator, into misclassifying its output as coming from the true distribution (which the generator doesn't know) while training the discriminator to accurately classify input as coming from the true distribution or from the generator.

Taking a Closer Look

Now that we know what GANs are at an intuitive level, let's look at the details that we glossed over in the previous section.

The Generator

- *What it is*: a generative model (e.g. a neural network)
- *Inputs:* noise, represented as **z**, from a predetermined prior distribution of any dimension
- *Parameters*: θ_g – a set of parameters specific to the generator's model class (e.g. neural network) that define how inputs are mapped to outputs
- *Outputs:* G(**z**) –a mapping of **z**, the noise, *to the data space* (i.e. if the true data distribution has 2 dimensions, the output of the generator should have 2 dimensions as well)

Usually the generator is a multilayer perceptron model, like a neural network. Its input is just noise, or to be more specific, a collection of noise variables. We can basically just sample from a given probability distribution and feed them into the generator. The generator then maps these noise inputs to the data space.

The data space in this context is the space that the true distribution that we ultimately want to estimate lies in. For example, if we want to approximate a univariate Gaussian distribution (i.e. the normal distribution sometimes referred to as a "bell curve") then we have to use a generator that outputs a single real number.

The Discriminator

- *What it is*: a discriminative model (e.g. a decision tree)
- *Inputs*: samples (individually represented as **x**) from the true data distribution and output from the generator
- *Parameters*: θ_d – a set of parameters specific to discriminator's model class that define how inputs are mapped to outputs

- *Output*: D(**x**) – the probability that **x** came from the true data distribution rather than the generator. Labels are assigned to **x** based on D(**x**).

Usually the discriminator is a multilayer perceptron model, just like the generator. The main difference here is that the discriminator's inputs are samples from both from the true data distribution and the generator's estimated distribution. The model outputs a probability that the input it was given comes from the true data distribution. If it's too low, the input is classified as "fake", otherwise it's classified as "true".

Simultaneous Training

The generator and the discriminator are trained simultaneously but have adversarial objective functions. The discriminator is trained so that it maximizes the probability that it classifies input correctly, as either real or fake. While the generator is trained to maximize the probability that the discriminator wrongly classifies its output. Intuitively, if the generator can approximate the true data distribution exactly, it will fool the discriminator as well as it possibly can. Specifically, the generator minimizes $\log\Big(1 - D\big(G(\boldsymbol{z})\big)\Big)$. Equivalently, the generator maximizes $\log\Big(D\big(G(\boldsymbol{z})\big)\Big)$.

Reformulating the Discriminator and Generator's Objectives

Putting these adversarial objectives together results in a single shared objective that dictates how the generator and discriminator learn. The shared objective is referred to as a two-player "minimax game". This is just a fancy way of saying minimizing the possible loss for a worst case scenario. In the GANs context, the players are the discriminator and the generator. Both "make moves", where "moves" describe the learning process, or equivalently, updates to their parameters, based on the shared objective that describes the value they receive from making a move:

$$V(D,G) = E\big[\log\big(D(\boldsymbol{x})\big)\big|\boldsymbol{x}\sim p_{data}(\boldsymbol{x})\big] + E\big[\log\Big(1 - D\big(G(\boldsymbol{z})\big)\Big) \mid \boldsymbol{z}\sim p_z(\boldsymbol{z})\big]$$

Let's break this down so it's more easily digestible. Let's focus on the first term. It's is a conditional expectation and can be summarized as the expected probability the discriminator classifies a sample from the true data distribution correctly. This is pretty intuitive, the discriminator's whole job is to correctly classify its input as true or fake.

The second term has a very similar interpretation. It's again a conditional expectation. This time it refers to the expected probability the discriminator classifies a "fake" sample as such. Recall that "fake" samples are just the output of the generator.

In sum, V(D,G) is essentially just a measure of how well the discriminator classifies what its given. This is a rather elegant formulation. The discriminator intuitively wants to maximize V(G,D) while the generator wants to minimize it. Now it should be clearer why these are called *adversarial* networks.

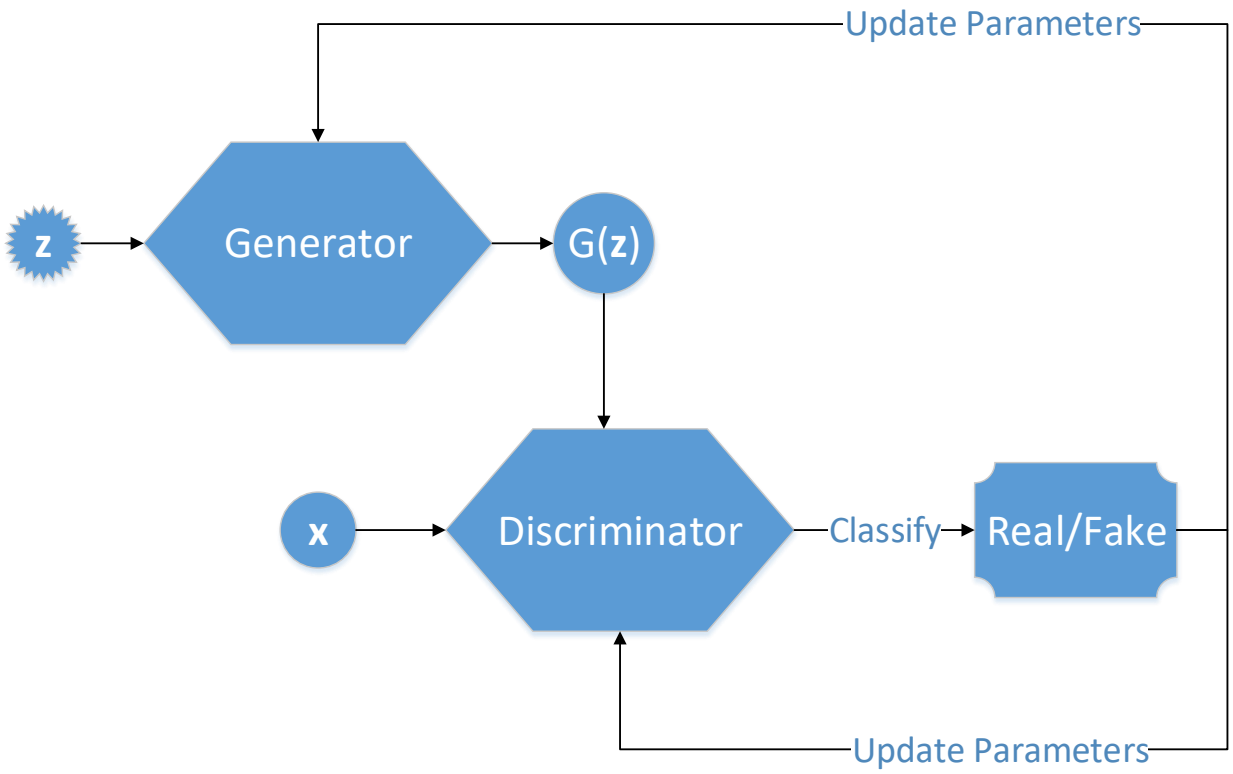
Let's formalize what we've just discussed and fold in the optimization layer that describes exactly *how* the discriminator and generator update their parameters. The shared objective becomes:

$$\min_G \max_D V(D,G)$$

Notice the inner expression is maximizing the value function V(D,G) by changing D. So, the parameters of the discriminator are updated in such a way that it's better at classifying, given the current parameters of G. Once those parameters are updated, the inner maximization expression is finished. Now we move on to the outer minimization expression. The parameters of the generator are updated so that the value function V(G,D) is as small as possible. But remember that the discriminator just learned how to better classify real and fake input. This creates a sort of signal that the generator picks up on, and learns. This process iterates until the generator converges to the data distribution (or any other stopping criteria like a specific number of training iterations have passed).

Diagrammatic View

Hopefully you didn't get lost in the details. To solidify and recap everything, let's take a look at a diagram which illustrates exactly what happens each iteration of training a GANs model.



1. Noise (**z**) is sampled from a prior distribution (e.g. a uniform distribution)
2. The noise is input to the generator which produces G(**z**)
3. The discriminator receives its input and calculates D(input), the probability its input comes from the true data distribution
    - Input is either a sample from the true data distribution (**x**) or the generator's output (G(**z**))
4. Using D(input), the discriminator classifies its input as either real (a sample from the true data distribution) or fake (the generator's output).

5. Based on the correctness of the discriminator's classification, the discriminator and generator both update their parameters with different objectives in mind
    - The discriminator tweaks its parameters to better classify its input correctly
    - The generator tweaks its parameters to maximize the probability it fools the discriminator
6. Repeat from step 1

Putting it into Practice

Now that we know all the nitty gritty details about GANs, let's code one up ourselves! To do so we'll be using the Python programming language, along with Keras, a deep learning toolkit that runs on TensorFlow (or theano, but we'll be using the tensorflow backend).

Data

The dataset we'll be working with is pretty small. It's a set of flag emoji images in black and white. The motivation behind using this dataset is because I just think it would be cool for a GAN to produce a flag all on its own (maybe the flag of the nation our robot overlords will found once AI surpasses human intelligence?). Further, it's easier to see how well the generator is performing when we can see the images it produces from noise. And finally, a smaller data set makes the implementation of GANs computationally tractable on most computers.

Tools for the job

We'll be using the following Python packages in our code so if you're following along make sure you've got everything listed below:

- Keras (https://keras.io/) for high-level deep-learning tools
- TensorFlow (https://www.tensorflow.org/) for the Keras backend
- Matplotlib (http://matplotlib.org/) for making graphs
- NumPy (http://numpy.org/) for data structures like vectors and tensors
- SciPy (http://scipy.org/) for reading in images
- Python Imaging Library (PIL) (http://www.pythonware.com/products/pil/) for reading in images backend
- Scikit-image (http://scikit-image.org/) for more image manipulation
- tqdm (https://github.com/tqdm) for a convenient progress bar for loops

Preprocessing

First thing's first, let's import the tools we need to set up the experiment:

```
import numpy as np
import matplotlib.pyplot as plt
import os
import scipy.misc
import random
from tqdm import tqdm
```

Now let's read in our data, split it into training and testing sets, then normalize it between 0.0 and 1.0:

```
w,h = 64,64
datadir = os.getcwd()+'/data/'
print(datadir)
X = []
for f in os.listdir(datadir):
    img = scipy.misc.imread(datadir+f, mode='RGBA')
    img = rgb2gray(rgba2rgb(img))
    img = scipy.misc.imresize(img, (w, h, 1))
    X.append(img)
X = np.asarray(X)
```

(for the sake of the tutorial we've omitted the splitting of the training and testing set code, since we're more interested in the quality of the generator's output rather than any sort of classification accuracy measure)

```
# normalize the sets between 0 and 1
X_train = X_train.astype('float32') / 255.
X_test = X_test.astype('float32') / 255.
```

```
# reshape sets
X_train = np.reshape(X_train, (len(X_train), w, h, 1))
X_test = np.reshape(X_test, (len(X_test), w, h, 1))
```

Finally, let's import the tools we need for implementing the GAN itself, all courtesy of Keras:

```
from keras.layers import Input, Dropout, Dense, Conv2D, MaxPooling2D, Flatten, UpSampling2D, \
BatchNormalization, Reshape, Activation
from keras.initializers import RandomUniform
from keras.optimizers import Adam
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Model
from keras import backend as K
from keras import metrics
```

There are a lot of parts as you can see! Some of them aren't used in the final product of the code, but if you're planning on altering the code yourself, this set of tools gives you a lot of options already.

Implementing the GAN

Now we're finally ready to implement the generative adversarial network!

Generator

The first thing we want to do is create the generator:

```
# build the generator
num_noise = 100
gen_input = Input(shape=(num_noise,))
layer = Dense(num_noise*w//4*h//4, kernel_initializer='glorot_normal')(gen_input)
layer = BatchNormalization()(layer)
layer = Activation('relu')(layer)
layer = Reshape( (w//4,h//4,num_noise) )(layer)
layer = UpSampling2D(size=(4,4))(layer)
layer = Dropout(0.5)(layer)
layer = Conv2D(64, (3, 3), padding='same', kernel_initializer='glorot_uniform')(layer)
layer = BatchNormalization()(layer)
layer = Activation('relu')(layer)
layer = Dropout(0.5)(layer)
layer = Conv2D(32, (3, 3), padding='same', kernel_initializer='glorot_uniform')(layer)
layer = BatchNormalization()(layer)
layer = Activation('relu')(layer)
layer = Dropout(0.5)(layer)
layer = Conv2D(1, (1, 1), padding='same', kernel_initializer='glorot_uniform')(layer)
gen_val = Activation('sigmoid')(layer) # final output is a value between 0 and 1

gen_opt = Adam(lr=1e-4)
generator = Model(gen_input, gen_val)
generator.compile(loss='binary_crossentropy', optimizer=gen_opt)
generator.summary()
```

There's a lot going on here so let's break it down.

First, we define the shape of the noise that the generator receives as its input. The variable "num_noise" refers to the length of the noise vector, **z** that we want. Here we define it as 100.

> ASIDE
>
> You might think to yourself (as I did), "let's just create our noise samples to have the same shape as the images and skip the reshaping step to save some time and computation." DON'T DO THIS! It will dramatically impact the quality of your results and you'll spend 2 days pulling your hair out trying to tune hyperparameters to no avail.
>
> Moral of the story: don't take shortcuts.

Next we create a dense layer of perceptrons, a typical step in any multilayer perceptron model. The kernel initializer follows the Glorot Normal distribution. It might sound fancy but really all this does is set the initial values of the weights in this layer to follow a normal distribution whose parameters are subject to the number of arcs going into this layer and the number going out of this layer.

We add batch normalization to this layer to speed up learning. Batch normalization just normalizes the weights of the layer to be in the range (0.0,1.0). Following that, we set the activation to "relu" or rectified linear unit, which is a linear activation function starting at 0. In order for the generator to produce grayscale images, we have to reshape the noise input accordingly from one dimension to two. So we do that with the "Reshape" line and then up-sample the result to fill in the gaps created by the reshaping. Finally, we add a constraint to this layer stating that only 50% of the perceptrons can be active at once (this is the line that contains "Dropout").

Next, we add a convolutional layer to the network. This is another standard practice when working with images (if you're unfamiliar with convolution, or convolutional neural networks read up on it here: https://en.wikipedia.org/wiki/Convolutional_neural_network). This layer has 64 filters, and the convolutional window has dimensions 5 by 5. Then we repeat the same steps as above for the convolutional layer (implement batch normalization, dropout, etc.).

And repeat a couple more times, slightly changing the parameters. Pay careful attention that the output of the generator has the correct dimensions for your application! Here we want our dimensions to be (64, 64) (or equivalently, (64, 64, 1)). Finally, we're left with our generative model! We use Adam for optimizing it. Let's take a look at the final product:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_35 (InputLayer)        (None, 100)               0
_________________________________________________________________
dense_34 (Dense)             (None, 25600)             2585600
_________________________________________________________________
batch_normalization_40 (Batc (None, 25600)             102400
_________________________________________________________________
activation_51 (Activation)   (None, 25600)             0
_________________________________________________________________
reshape_18 (Reshape)         (None, 16, 16, 100)       0
_________________________________________________________________
up_sampling2d_16 (UpSampling (None, 64, 64, 100)       0
_________________________________________________________________
dropout_58 (Dropout)         (None, 64, 64, 100)       0
_________________________________________________________________
conv2d_50 (Conv2D)           (None, 64, 64, 64)        57664
_________________________________________________________________
batch_normalization_41 (Batc (None, 64, 64, 64)        256
_________________________________________________________________
activation_52 (Activation)   (None, 64, 64, 64)        0
_________________________________________________________________
dropout_59 (Dropout)         (None, 64, 64, 64)        0
_________________________________________________________________
conv2d_51 (Conv2D)           (None, 64, 64, 32)        18464
_________________________________________________________________
batch_normalization_42 (Batc (None, 64, 64, 32)        128
_________________________________________________________________
activation_53 (Activation)   (None, 64, 64, 32)        0
_________________________________________________________________
dropout_60 (Dropout)         (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_52 (Conv2D)           (None, 64, 64, 1)         33
_________________________________________________________________
activation_54 (Activation)   (None, 64, 64, 1)         0
=================================================================
Total params: 2,764,545.0
Trainable params: 2,713,153.0
Non-trainable params: 51,392.0
_________________________________________________________________
```

In total we have ~2.5 MILLION parameters to learn. This is a large network, clearly. If you didn't install the GPU version of tensorflow you might have trouble running this in a feasible amount of time, unfortunately.

Discriminator

Let's move on to the discriminator now:

```
# build the discriminator
dis_input = Input(shape=(w,h,1))
layer = Conv2D(256, (5,5), padding='same', activation='relu')(dis_input)
layer = LeakyReLU(0.2)(layer)
layer = Dropout(0.25)(layer)
layer = Conv2D(128, (5,5), padding='same', activation='relu')(layer)
layer = LeakyReLU(0.2)(layer)
layer = Dropout(0.25)(layer)
layer = Flatten()(layer)
layer = Dense(32)(layer)
layer = LeakyReLU(0.2)(layer)
layer = Dropout(0.25)(layer)
dis_val = Dense(2, activation='softmax')(layer)

dis_opt = Adam(lr=1e-4)
discriminator = Model(dis_input,dis_val)
discriminator.compile(loss='categorical_crossentropy',optimizer=dis_opt)
discriminator.summary()
```

We follow similar steps when creating the discriminative model. The input has the same dimensions as the generator. Our first layer isn't a densely connected layer though, we jump straight to convolution with a large kernel size. The activation function is slightly different, instead of "relu" we use "Leaky ReLU", which allows the activation to take on very small values when its input is less than 0 (whereas relu's activation is always 0 when its input is less than 0). This is done to avoid "dying" perceptrons. In some cases during training the a perceptron will encounter a value such that it will never activate for any other data point again, this leaky relu activation function helps remedy that[1]. We add the same constraint to limit the number of nodes that can be activated at any given time.

At the end of the network we flatten everything so that it's one-dimensional and finally coerce the network to output only two values. These will essentially be the two terms that make up the valuation function we previously discussed.

Let's take a look:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_36 (InputLayer)        (None, 64, 64, 1)         0
_________________________________________________________________
conv2d_53 (Conv2D)           (None, 64, 64, 128)       3328
_________________________________________________________________
leaky_re_lu_25 (LeakyReLU)   (None, 64, 64, 128)       0
_________________________________________________________________
dropout_61 (Dropout)         (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_54 (Conv2D)           (None, 64, 64, 64)        204864
_________________________________________________________________
leaky_re_lu_26 (LeakyReLU)   (None, 64, 64, 64)        0
_________________________________________________________________
dropout_62 (Dropout)         (None, 64, 64, 64)        0
_________________________________________________________________
flatten_9 (Flatten)          (None, 262144)            0
_________________________________________________________________
dense_35 (Dense)             (None, 32)                8388640
_________________________________________________________________
```

[1] http://cs231n.github.io/neural-networks-1/

```
leaky_re_lu_27 (LeakyReLU)   (None, 32)                0
_________________________________________________________________
dropout_63 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_36 (Dense)             (None, 2)                 66
=================================================================
Total params: 8,596,898.0
Trainable params: 8,596,898.0
Non-trainable params: 0.0
_________________________________________________________________
```

Here we have another boat-load of parameters, ~8.5 million!

GAN

Finally we can build our generative adversarial network by stacking the generator and discriminator. Conveniently, they will retain their learned weights throughout the training process. One small caveat is that we don't want back-propagation to occur for the discriminator during the learning stage. Why? Because of the optimization step for GANs, recall that first the discriminator updates its parameters to maximize its probability of correctly classifying its input and then the generator updates its own parameters to maximize the probability of fooling the discriminator. If we allowed the discriminator to learn while the generator is changing its parameters to fool the discriminator, the generator would effectively be "blocked" or "outsmarted" by the discriminator since it knows exactly what the generator is learning.

```
# define a function to freeze the weights on the discriminator
# (we don't want backpropogation to occur for the discriminator when training the GAN)
def toggle_trainable(network, flag):
    network.trainable = flag
    for layer in network.layers:
        layer.trainable = flag
```

```
toggle_trainable(discriminator, False)
```

```
# build the GAN!
GAN_input = Input(shape=(w,h,1))
layer = generator(GAN_input)
GAN_val = discriminator(layer)
GAN = Model(GAN_input, GAN_val)
GAN.compile(loss='categorical_crossentropy', optimizer=gen_opt)
GAN.summary()
```

That was easy, huh? Keras conveniently allows us to just concatenate the discriminator onto the generator and wah-lah, an adversarial network!

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_37 (InputLayer)        (None, 100)               0
_________________________________________________________________
model_28 (Model)             (None, 64, 64, 1)         2764545
_________________________________________________________________
model_29 (Model)             (None, 2)                 8596898
=================================================================
Total params: 11,361,443.0
Trainable params: 2,713,153.0
```

```
Non-trainable params: 8,648,290.0
_________________________________________________________________
```

In total we have ~11 million parameters. Here the non-trainable parameters is inflated because we just toggled the discriminator's learning ability to "off".

Training the Generative Adversarial Network

Let's define a function that will train the GAN on a per-batch basis.

```
def train_GAN(num_epochs=1000, batch_size=32):

    for e in tqdm(range(num_epochs)):

        # sample from noise
        noise = np.random.uniform(0.0, 1.0, size=(batch_size,num_noise))

        # run noise through the generator to get G(z)
        gen_imgs = generator.predict(noise)

        # sample from true data distribution
        np.random.shuffle(X_train)
        true_img_batch = X_train[:batch_size,:,:,:]

        # combine G(z) and true sample
        this_epoch_X = np.concatenate((true_img_batch, gen_imgs))

        # create true class response var
        y = np.zeros((2*batch_size,2))
        y[:batch_size,1] = 1
        y[batch_size:,0] = 1

        # train the discriminator
        toggle_trainable(discriminator,True)
        dis_loss = discriminator.train_on_batch(this_epoch_X, y)
        loss_dict['d'].append(dis_loss)

        # create GAN response var
        noise_tr = np.random.uniform(0.0, 1.0, size=(batch_size,num_noise))
        y2 = np.zeros((batch_size,2))
        y2[:,1] = 1

        # train GAN
        toggle_trainable(discriminator,False)
        gan_loss = GAN.train_on_batch(noise_tr, y2)
        loss_dict['g'].append(gan_loss)
```

Our function takes two arguments: the number of epochs to train the GAN for (basically just the number of learning iterations) and the size of batch to train on. This batch size defines how many samples the GAN trains on before altering its parameters. Earlier in the post we spoke in terms of singular samples, this is just a tiny added layer of complexity we're adding in order to speed up the learning process.

On to the algorithm itself, we perform the following steps for each epoch (iteration):

1. Sample from the uniform distribution, batch_size times. This is **z** described above. The length of **z** is determined by the num_noise variable.
2. Use **z** as input to the generator. In other words "predict" using the generative model. This produces G(**z**).
3. Randomly sample from the training set batch_size times. This is **x**.
4. Combine G(**z**) and **x** into one collection of samples.
5. Create a response array of size (batch_size*2, 2) where the first column equals one if the corresponding row is a sample from the true data distribution (i.e. **x**) and zero otherwise. The

second column follows a similar structure where it equals one if the corresponding row is NOT a sample from the true data distribution (i.e. output from the generator, G(**z**)) and zero otherwise.
    - This is the data that the discriminator will train on. Recall the discriminator outputs 2 values per input. These can be thought of as the probability the input is either **x** or G(**z**). The discriminator classifies the input corresponding to whichever output value is largest.
6. Toggle the discriminator's ability to learn to "on" and train it.
    - This step corresponds to the inner maximization expression in the value function V(D,G) of the minimax game the discriminator and generator play.
7. Resample more noise following the same steps in (1) and (2)
8. Create a response array of size (batch_size, 2). Here, the second column always equals one essentially telling the GAN to consider all input samples as "true" samples. The first column always equals zero.
    - This makes the generator portion of the GAN to learn how to map the noise in such a way that the discriminator will label it as "true".
9. Toggle the discriminator's ability to learn to "off"
    - This allows the generator to learn how best to fool the discriminator without the discriminator learning exactly what the generator is doing.
10. Train the GAN
11. Repeat from (1), num_epochs times.

Now let's run the code for 100 epochs and take a look at how the generator is performing:

```
# train GAN for 100 epochs
train_GAN(num_epochs=100, batch_size=32)
```

```
100%|██████████████████████████████████████████████████████████████████████████████| 100/100 [00:31<00:00,  3.48it/s]
```

```
# view the quality of the generated "fake" images after 100 iterations of training
plt.figure(figsize=(20,2))
noise = np.random.uniform(0.0, 1.0, size=(10,num_noise))
gen_imgs = generator.predict(noise)
for i in range(1,10):
    ax = plt.subplot(1, 10, i)
    plt.imshow(gen_imgs[i].reshape(w,h))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Not bad results after only 100 epochs, the generator is producing circular-looking, yet noisy, images. More training means better results, so let's train for 200 more epochs.

```
# train GAN for 200 more epochs
K.set_value(gen_opt.lr,1e-4)
K.set_value(dis_opt.lr,1e-5)
train_GAN(num_epochs=200, batch_size=16)
```

```
100%|██████████| 200/200 [00:38<00:00,  5.17it/s]
```

```
# view the quality of the generated "fake" images after 200 more iterations of training
plt.figure(figsize=(20,2))
noise = np.random.uniform(0.0, 1.0, size=(10,num_noise))
gen_imgs = generator.predict(noise)
for i in range(1,10):
    ax = plt.subplot(1, 10, i)
    plt.imshow(gen_imgs[i].reshape(w,h))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

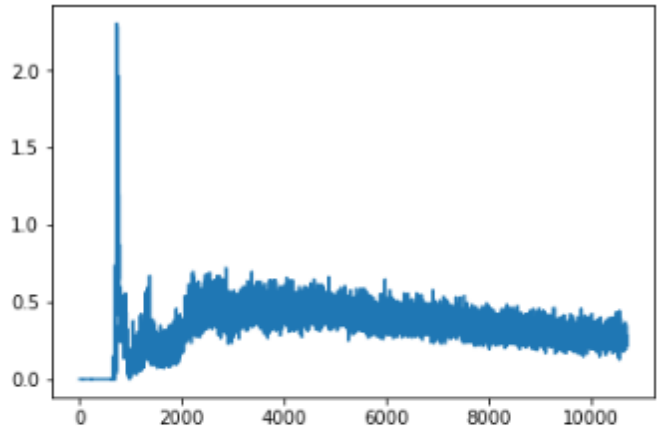
We're getting somewhere! The generated images are getting more and more circular. Notice how we change the learning rate for the generator and the discriminator. This is to avoid saturation of the model. In other words, it's pretty easy for the discriminator to discern what's from the generator and what's not at this point early in the training process. To remedy the discriminator being "too smart" for the generator (and blocking it from learning), we reduce its learning rate to less than the generator's. This allows the generator to "catch up" to the discriminator by learning at a faster rate.

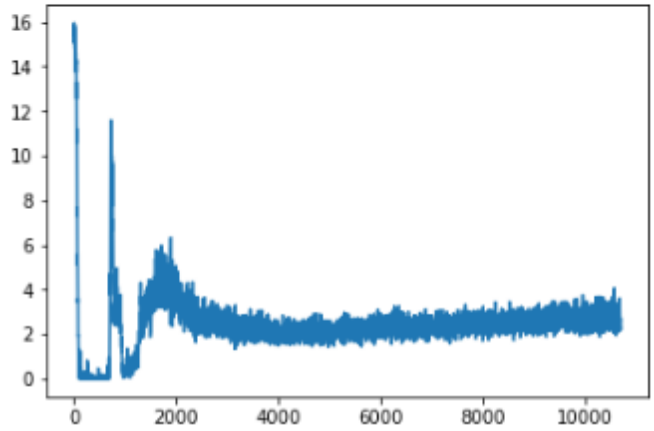
Usually it'll take thousands of epochs to produce good looking results. Take a look at the generated images after 1000 more epochs:

Another 10,000 more epochs:

And we have some solid(ish) looking flag emojis!

Let's take a look at the GAN's path to producing semi-believable flags by plotting the loss for the adversarial net as a whole and the discriminator





The top plot is the discriminator's loss and the bottom is the adversarial net (i.e. the stacked generator and discriminator model we created previously). As you can see the first 2000 or so epochs were rough while the remaining iterations saw a stabilization in the loss. If we were to run this for another 50,000 epochs, the loss for both would likely converge at which point the discriminator has a ½ chance of correctly discerning between generator output and a sample from the true data distribution. Fascinating stuff…

Takeaways

1. GANs are fragile things. There needs to be a delicate balance between the discriminator and generators learning capabilities and very, very careful hyperparameter tuning to ensure the model is fits "correctly" (i.e. doesn't oversaturate).
2. Sort of an extension of (1) but network setup/structure plays a very important role in both computational tractability (can't tell you how many times I ran into the error "Failed to allocate XX.XX GB of memory") and convergence.
3. Keras is awesome. At first I tried to approach this tutorial using tensorflow but the overhead involved in learning its lower-level interface quickly proved to be more trouble than it was worth.
4. Blogs hold all the information you could ever need to implement a deep-learning algorithm (seriously questioning my Master's degree and all the debt that is coming with it).

Resources and References

Shoutout to the following papers and blogs for turning this 100 hour project into a 40 hour project:

http://www.kdnuggets.com/2016/07/mnist-generative-adversarial-model-keras.html

https://github.com/bstriner/keras-adversarial

https://oshearesearch.com/index.php/2016/07/01/mnist-generative-adversarial-model-in-keras/

https://arxiv.org/pdf/1406.2661.pdf

https://medium.com/towards-data-science/gan-by-example-using-keras-on-tensorflow-backend-1a6d515a60d0

https://arxiv.org/pdf/1611.02163.pdf