DCGANs in Keras - Part 1

This Tutorial

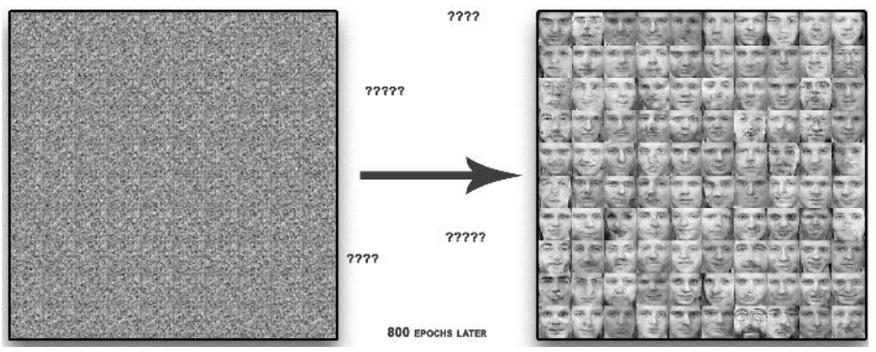
This tutorial is a walkthrough of creating a very basic DCGAN in Keras for image generation. I am using TensorFlow as my backend. The assumption going into this is that you already have Keras and TensorFlow installed and working. If you don't have that setup, do that first before proceeding. This part is also going to skip the fundamentals of what a GAN is, so if you would like to get acquainted with that, check out this article, Getting Started with GANs. It is short, but links to a lot of good resources that really helped me to get a grasp on how a GAN works and why it works.

In this part, we are going to talk about the generator portion of the DCGAN and how one might set up the model in Keras.

The Generator

The generator is the part of a GAN that is responsible for generating the new data that is to be compared to the original dataset by the discriminator. The most basic GANs tend to setup their generators to take in some sort of input noise, usually a vector of 100 random numbers. The generator then uses this random data and manipulates it into the format of whatever target data is in the dataset. The goal of the generator is to ultimately learn how to

better manipulate that random noise data into something that resembles the target dataset, thus fooling the discriminator in the process. Now with that basic description out of the way, let's dive into a basic generator architecture I wrote that is relatively shallow but is a starting point for understanding how the generator does what it does.



The Sequential Model

This generator is constructed using Keras' Sequential Model. The model is just what its name suggests, sequential. A sequence of layers are added to this model, strung one after another. It is the simple model that Keras offers (its Model API can be used for more complex networks, which will be seen later). Keras' documentation has a very thorough guide that can be found here. It's pretty decent for getting acquainted with the Sequential Model, something we'll be doing step-by-step, however, if you wish to dive right in and see it in action in a couple instances, that guide is definitely a good starting place. To use this model is fairly simple, just import the needed portions and create a model variable:

```
from keras.models import Sequential

model = Sequential()
```

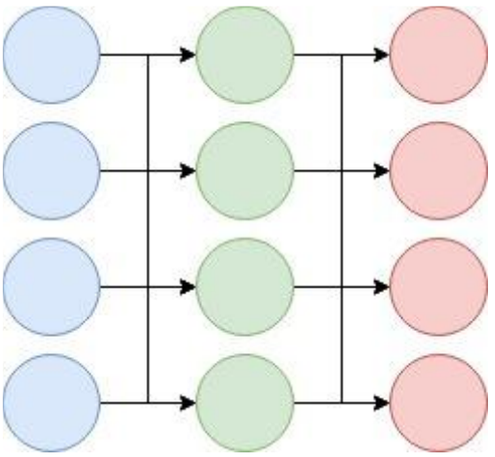
Layers

Next, we will discuss some of the layers we will be using to construct the generator. All layers discussed can be found in keras.layers. For a more extensive overview of some general proprties of layers in Keras, the documentation provides a quick read here.

Dense Layer

The dense layer is arguably one of the simplest layers to work with in Keras. It is just a densely connected neural network layer. The least amount of parameters it needs to

work is the amount of nodes in the layer, though if it is the first layer in the network, it also needs an input dimension. An example of this in action, slightly modified from the documentation:



```
# Input size specified
model.add(Dense(32, input_shape=(16,))) #
[batch_size X 16] -> [batch_size X 32]

# Just number of nodes
model.add(Dense(32))                    #
[batch_size X 32] -> [batch_size X 32]
```

It's useful to note, by not specifying a second size parameter in the input_shape, it can be any size. This is usually where your batch size determines the size of the data flowing through.

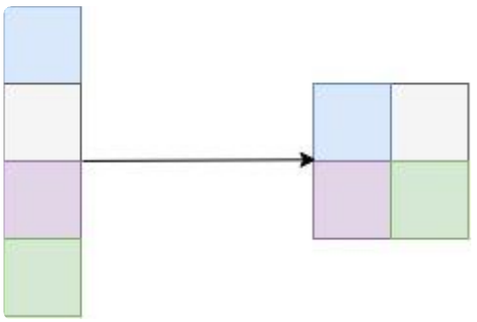
Batch Normalization Layer

The batch normalization layer basically normalizes the layer directly before it, helping to prevent overfitting the sample data. It has a lot of parameters that can be customized, but I find in most instances I've used it, I've just used the default version simply by adding to my model:

```
model.add(BatchNormalization())
```

Reshape Layer

The reshape layer takes data from the previous layer (or input) and transforms it into a specified size. This is particularly useful in this instance because it allows our model to transform a single column of noise and apply a transformation that puts it into a form that can be upscaled and transformed into an image, a 3 dimensional entity if you count width, height, and color channels.

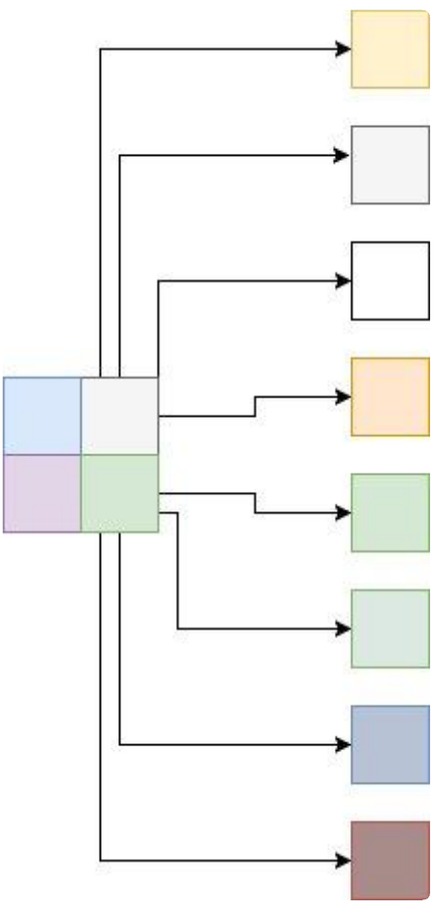


```
model.add(Reshape((5, 3), input_shape=(15,))) #
[batch_size X 15] -> [batch_size X 5 X 3]
```

Conv2D Layer

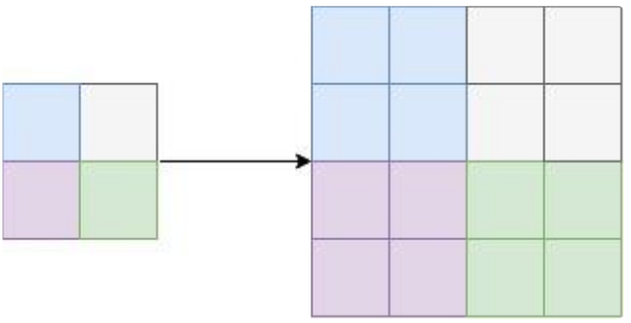
The convolution 2D layer is basically a layer that applies a series of filters to an image or array of pixels/data values. The first argument is always the number of filters. A kernel_size can be specified as a number or tuple, specifying the window of pixels to be filtered. The stride argument is similar, but specifies the strides that should be made along the width and height of the image.

```
model.add(Reshape((6, 6, 2), input_shape=(72,))) #
[batch_size X 72] -> [batch_size X 6 X 6 X 2]
model.add(Convolution2D(24, kernel_size=(6,6)))  #
[batch_size X 6 X 6 X 2] -> [batch_size X 1 X 1 X
24]
```



UpSampling2D Layer

The upsampling 2D layer is a type of convolutional layer that scales up data by a factor of 2 for both the rows and columns. It has a very simple usage:



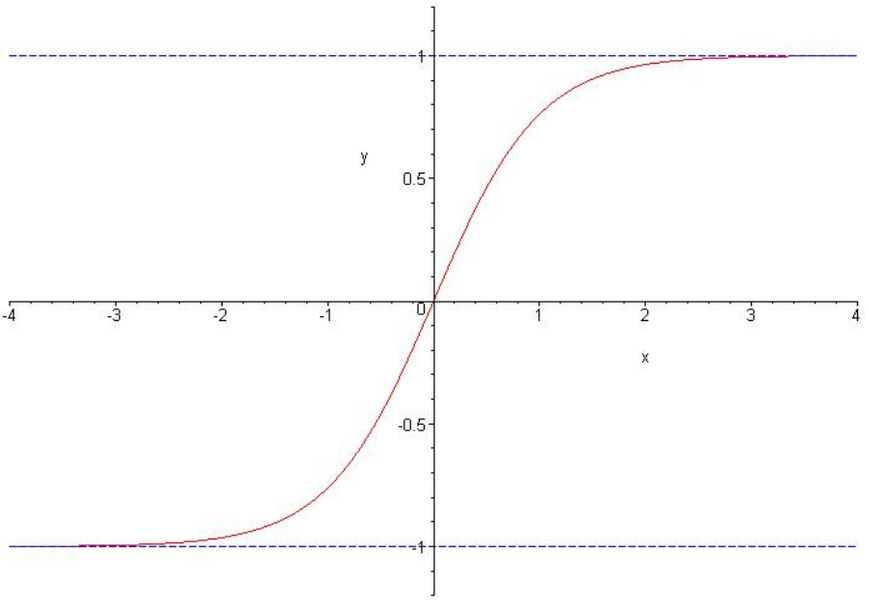
```
model.add(Reshape((6, 6, 2), input_shape=(72,)))
# [batch_size X 72] -> [batch_size X 6 X 6 X 2]
model.add(UpSampling2D())
# [batch_size X 6 X 6 X 2] -> [batch_size X 12 X
12 X 2]
```

Activations

Activations are basically the function that is used to calculate the output given an input. By default, all layers use a linear activation function of f(x) = x. Activations can be specified for a layer by using the activation parameter, or for more complex activations, by simply adding it as a layer. Below, we address some of the useful activation functions for the generator of a GAN.

Tanh Activation

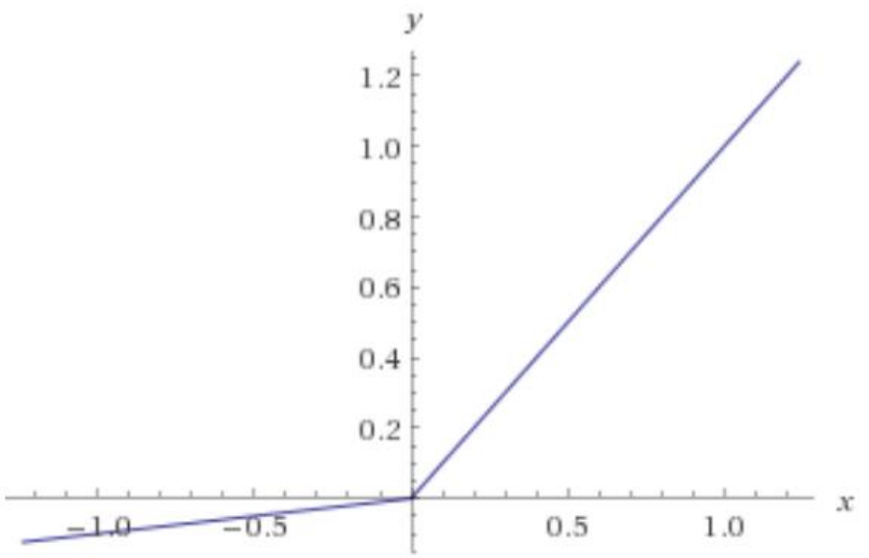
Tanh is pretty self-explanatory. It is the hyperbolic tangent function (graphed below), and basically ends up scaling all values to [-1, 1]. This is useful in this situation in comparison to the more classically used sigmoid function, because it provides a more balanced distribution instead of mapping values to [0, 1]. This is a simple activation function and can just be specified like so:




```
model.add(Dense(32, input_shape=(16,),
activation='tanh'))
```

LeakyReLU Activation

LeakyReLU is one of Keras' "advanced" activation functions. It is graphed below and follows the form f(x) = x for x >= 0, and f(x) = alpha * x for x < 0, where alpha is a specified "leaky" value. Because it is an advanced activation function, it is usually used after the layer it is for like this:




```
from keras.layers.advanced_activations import
LeakyReLU
model.add(Dense(32, input_shape=(16,)))
model.add(LeakyReLU(0.2)) # Substitutes the linear
activation for LeakyReLU
```

Putting It All Together

So taking everything we've mentioned above, I've sort of compiled it all into a central model of what a generator might look like below. I have some initial configuration

parameters at the top of the code snippet, but by and large, everything you see should look familiar if you've read and understood the above descriptions.

```
# Some configuration parameters:

divisor = 4
noise_size = 100
output_width = 32
output_height = 32
color_channels = 3

start_width = output_width // divisor
start_height = output_height // divisor

# End config params


model = Sequential()

# Sequence 1
model.add(Dense(256 * start_width * start_height,
input_dim=noise_size))
model.add(LeakyReLU(0.2))
model.add(BatchNormalization())
model.add(Reshape((start_width, start_height,
256)))

# Sequence 2
model.add(UpSampling2D())
model.add(Conv2D(128, (5,5), padding='same'))
model.add(LeakyReLU(0.2))
model.add(BatchNormalization())

# Sequence 3
model.add(UpSampling2D())
model.add(Conv2D(color_channels, (5,5),
padding='same', activation='tanh'))
```

So there we have it. The above is what a generator component of a GAN would look like, or at least my simplified example. If you have questions, let me know below. And if you have any corrections, or tips for improving (I'm still learning too), definitely let me know about those as well.

For a full viewing of what we are aiming to build step-by-step, check out my github repo. And for the next section, check out DCGANs in Keras - Part 2.