Creative Machines: Generating 19th century artwork using Deep Convolutional Generative Adversarial Networks

Kenan Karavoussanos
School of Computer Science and
Applied Mathematics
University of the Witwatersrand, Johannesburg

***Abstract*—The abstract goes here.**

I. Introduction

Generative Adversarial Networks(GANs) have seen a significant amount of research since the release of the seminal paper by Goodfellow et al. [2014]. The basic idea of a GAN is to pit two neural networks against one another in what is known as a minimax game. The *Generator* takes in random input $\boldsymbol{z}$ and constructs an output $G(\boldsymbol{z})$. This is fed into the *Discriminator* which tries to determine whether the input is real or fake. The game is played by the Generator continually trying to fool the Discriminator into thinking all its outputs are real, while the Discriminator continually tries to learn to discriminate between $G(z)$ and $x \in$ the dataset.

GANs have been used extensively to generate realistic images including faces Gauthier [2014] and artwork Tan et al. [2017]. The focus of this paper is to use a Deep Convolutional Generative Adversarial Network(DCGAN) to generate artworks from four similar styles of the late 19th century. Specifically, impressionism, pointillism, fauvism and expressionism, see Fig. 1.

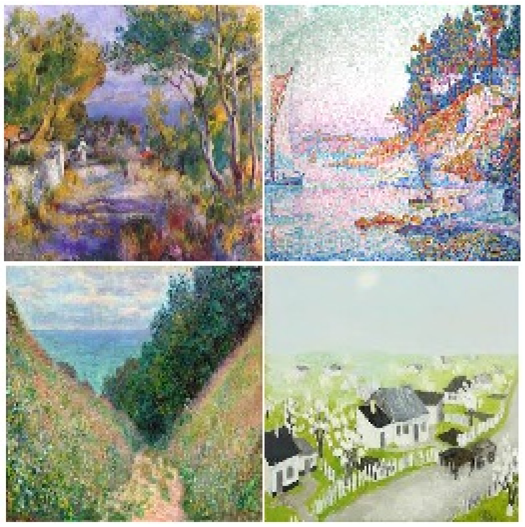


Fig. 1. Image samples by style: From top left; Impressionism, Pointillism, Expressionism and Fauvism.

II. Dataset

A. Collection

The images were collected using google image search and a bulk images downloader. Specifically, we searched the site www.wikiart.org for the style keyword e.g impressionism, pointillism etc. Once all the images were downloaded, a small script was run to remove all images that were not of the JPEG file type. The JPEG file type is preferable to PNG because JPEG has 3 colour channels as opposed to 4 in PNG. This means the dimensionality of our datapoints is reduced meaning less data is required.

B. Preprocessing

All the JPEG images were resized to 128 x 128 pixels using the PIL image library with Antialiasing to downscale the image. If the image was not square, it was then padded to fit the 128 x 128 size. These images were then transposed to double the size of our dataset. After this process, the image pixel values were scaled to the range [-1,1] so the Generator can make use of the tanh activation function at its output. This has been shown to improve color saturation of generated images Radford et al. [2015]. Once scaled, the images were flattened, this improves performance of the next step, which is object serialization. The python pickle package was used to serialize an array of 512 images. In total we have 11 of such batches, thus our data set size is 5632 images.

III. Objective Function

The network can be described as a two-player zero sum minimax game and is expressed by the following objective function:

$$\min_G \max_D V(D,G) = \mathbf{E}_{\boldsymbol{x}\sim p_{dataset}(\boldsymbol{x})}[logD(\boldsymbol{x})] + \mathbf{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[log(1-D(G(\boldsymbol{z})))] \quad (1)$$

Where, $\boldsymbol{x}$ is an image from our dataset, $\boldsymbol{z}$ is a random variable distributed as $p_{\boldsymbol{z}}(\boldsymbol{z})$. $G(\boldsymbol{z})$ is a fake image. V the value function is some loss function that the Generator seeks to minimize and the Discriminator seeks to maximize. The optimal point for this minimax game is known as *Nash Equilibrium*, this is a state where any change in the strategy of either player will not produce an advantage given that the other player's strategy remains the same. We will see in the next section why reaching this equilibrium point is desirable for our GAN.

IV. Training a Generative Adversarial Network

In this section we provide the main training algorithm of the Generative Adversarial Network as well as describe the potential issues one may face while training a GAN.

A. Training Algorithm

```
Input: $D, G, V, Dataset, Epochs, batchSize, \epsilon$
Output: $D, G$
$numBatches \leftarrow floor(length(Dataset)/batchSize)$
$Count \leftarrow 0$
$loss_{previous} \leftarrow V(D, G)$
$loss_{current} \leftarrow 0$
while $Count < Epochs$ do
    for $(i \leftarrow 0; i < numBatches; i \leftarrow i + 1)$ do
        $batch \leftarrow getBatch(Dataset, i, batchSize)$
        $dLoss \leftarrow 0$
        $gLoss \leftarrow 0$
        for $image\ in\ batch$ do
            $Let\ \boldsymbol{z} \sim \mathcal{N}(\mu, \sigma)$
            $generatedImage \leftarrow G(\boldsymbol{z})$
            $Dreal \leftarrow D(image)$
            $Dfake \leftarrow D(generatedImage)$
            $dLoss \leftarrow dLoss + discLoss(Dfake, Dreal)$
            $gLoss \leftarrow gLoss + genLoss(dLoss)$
        end
        $loss_{previous} \leftarrow loss_{current}$
        $loss_{current} \leftarrow dLoss + gLoss$
        if $||loss_{previous} - loss_{current}|| < \epsilon$ then
            return $D, G$
        end
        $G \leftarrow updateModel(G, gLoss)$
        $D \leftarrow updateModel(D, dLoss)$
    end
    $Count \leftarrow Count + 1$
end
```

Algorithm 1: GAN Training

B. Common GAN training issues.

1) Modal Collapse: The distribution of the dataset may consist of multiple modes. In a situation where the generator sees success when generating data similar to these modes, unless the discriminator can find a strategy where it can distinguish the generator output and data from this mode, the generator will stick to the strategy of generating data similar to the mode. This situation is referred to as Modal Collapse. This is can be seen when the generator loss decreases drastically and the discriminator loss increases drastically. Modal collapse is a common occurrence during GAN training and the two prevention strategies used in this work are:

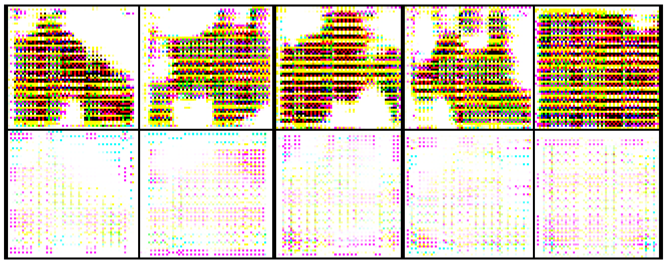


Fig. 2. Modal collapse. Top row: 5 samples generated from 5 random seeds. Bottom row: 5 samples generated using the same random seeds.

- Penalizing overconfidence in the Generator.
- Adding randomness ,e.g noise or dropout, to the discriminator network.

Figure 2 shows an example of modal collapse. The top row were images generated from 5 random seeds. The bottom row shows images generated using the same 5 random seeds, 10 epochs later. Notice the variation in the top row and the lack thereof in the bottom row.

A special case of Modal Collapse is when an initial modal collapse occurs, the discriminator learns a strategy to discriminate these outputs, then the generator will collapse onto another mode. This repeats and the generator oscillates from mode to mode and the discriminator cannot find a strategy to stop this oscillation.

2) Non-convergence: The GAN objective function is non-convex, as such gradient descent based optimizers are not guaranteed to find a global minimum i.e the Nash Equilibrium. It has been shown that gradient descent optimizers can enter a stable orbit rather than converging to a minimumSalimans et al. [2016]. The results from the fact that a change in strategy from one player will result in a change of strategy from another player. If these changes in strategy become cyclical, the gradient descent algorithm will enter said stable orbit. This can be seen as periodicity in the loss function during training.

3) Diminishing Gradients: If the discriminator begins to overpower the generator i.e it cannot find a successful strategy, the generator gets ”backed up into a corner”. Formally, any change to the generator's weights will result in an increased cost, this is analogous to being stuck in a local minimum. As such the rate at which the generator learns is reduced. This can be visualised as the generator loss being squeezed to a higher value while the discriminator loss approaches zero. This can be avoided by using an optimizer that can escape or avoid local minima.

V. Implementation Details

This section describes the specific implementation of the GAN used in this work. The software and hardware used as well the architectures used are described.

A. *Software and Hardware*

- Tensorflow 2.0.0-alpha0.
- Google Colaboratory.
- The ADAM Optimizer.
- Nvidia Tesla 24GB K80
- 16GB RAM

B. *Architecture*

This section provides a description and motivation of the architectures of the Generator and Discriminator Networks. For full Tensorflow architecture diagrams, refer to the appendix at the end of this document.

1) Generator: The Generator consists of a Fully connected layer followed by four Transpose Convolutional layers. The Transpose Convolutional layers upsample the input noise and produce and image of the required dimensions. All hidden layers use ReLU activations and the output layer uses the Tanh activation function. These activations were chose as it has been shown that these activation functions for the generator stabilize training Radford et al. [2015]. All hidden layers are batch normalized.

2) Discriminator: The Discriminator consists of 4 convolutional layers, 2 with dropout and 2 without, and a fully connected layer at the output. This network uses strided convolution to downsample the image as opposed to pooling layers. LeakyReLU activations are used for all hidden layers as they have been shown to improve GAN performance when used in the Discriminator Radford et al. [2015]. The output layer uses a sigmoid activation function. All hidden layers are batch normalized.

VI. Results

The network was trained over 200 epochs, with an average epoch taking approximately 130 seconds, yet failed to converge. From figure 3 we can see that the loss discriminator network quickly approaches zero. The large spikes in discriminator loss are indicative of modal collapse, however the discriminator recovers from these spikes. This is indicative of the generator oscillating between modes. The loss is roughly periodic, this is evidence that the ADAM optimizer, which is a gradient descent based optimizer, has entered a stable orbit as described by Salimans et al. [2016].

Despite this, from figure 4 we can see the Generator is learning. At epoch 1, the generated images are relatively homogeneous. At epoch 100, we can see distinct blobs and foreground and background with high colour contrast. At epoch 200, we can see the colour palette becomes more refined and there blobs become less jagged. This shows that given enough training time, this network may prove to converge.

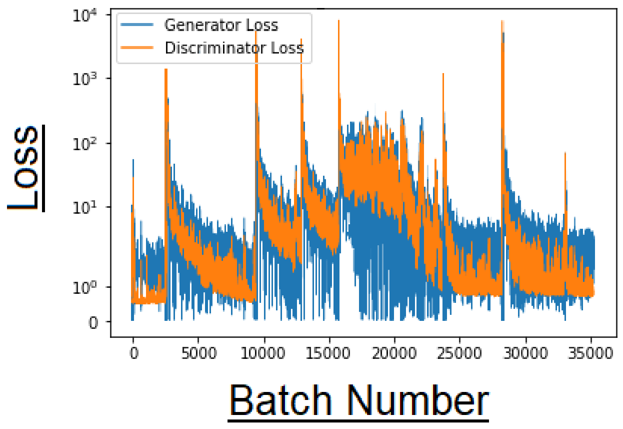


Fig. 3. Generator and Discriminator Loss on a symmetric logarithmic scale.

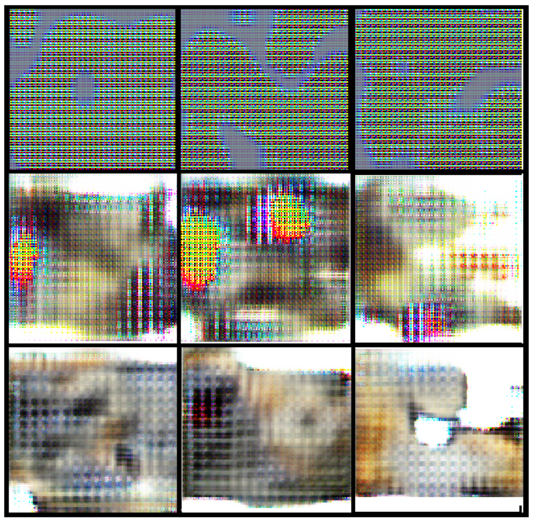


Fig. 4. Evolution of the Generator. Rows correspond to Epoch 1,100 and 200 respectively. Columns correspond to the same random seed.

VII. Conclusion

This goal of this work was to generate 19th century artwork with the use of a Deep Convolutional Generative Adversarial Network. We describe the collection and preprocessing steps of a set of similarly styled 19th century artworks. We discuss the theory and i behind GANs as well as the GAN training algorithm. We examine common issues with training a GAN and provide strategies to overcoming said issues. The Deep convolutional architectures of the Generator and Discriminator networks are discussed and justified. The GAN was trained for 200 epochs yet failed to converge and this failure was linked back to the common GAN training issues discussed earlier in the work. Despite this, we can see the evolution of the Generator through samples of images produced throughout training and this hints at the potential convergence of the network given enough training time.

References

Jon Gauthier. Conditional generative adversarial nets for convolutional face generation. *Class Project for Stanford CS231N: Convolutional Neural Networks for Visual Recognition, Winter semester*, 2014(5):2, 2014.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

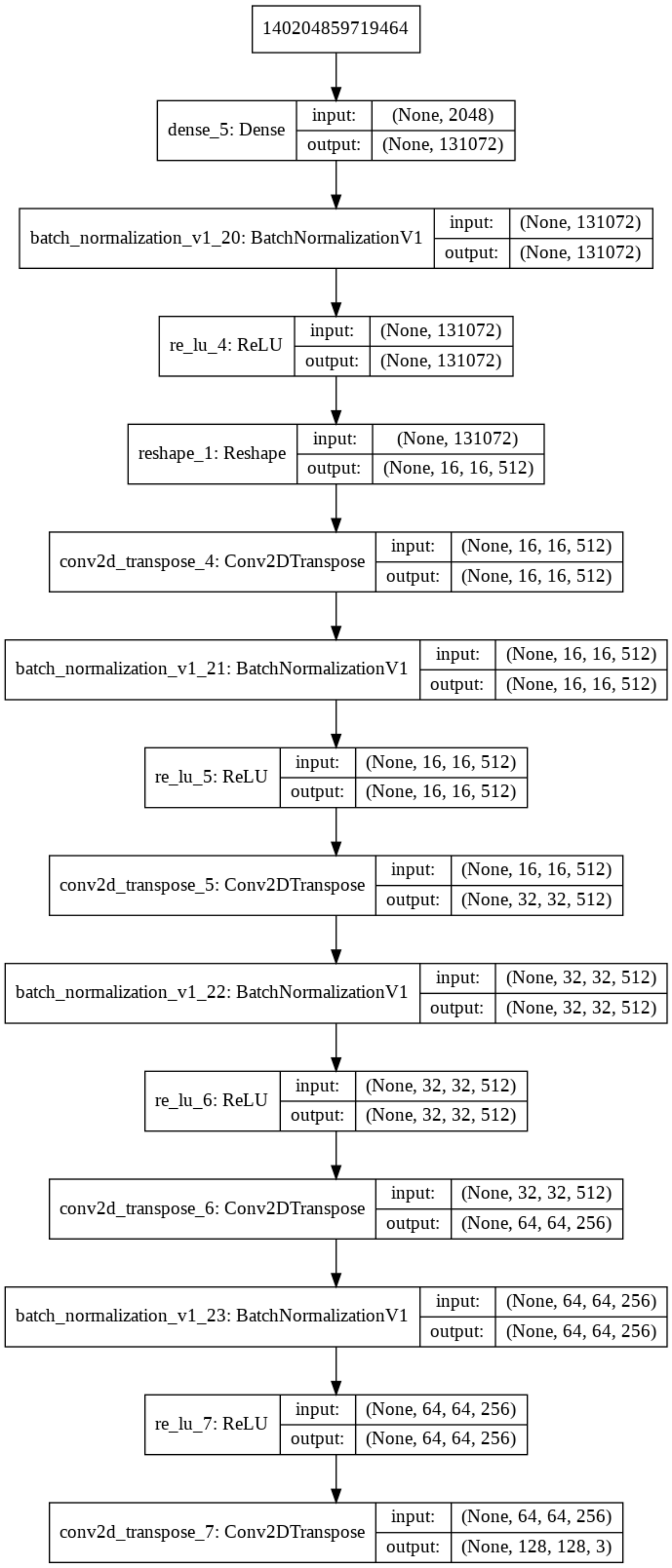
Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

Wei Ren Tan, Chee Seng Chan, Hernán E Aguirre, and Kiyoshi Tanaka. Artgan: Artwork synthesis with conditional categorical gans. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3760–3764. IEEE, 2017.

VIII. Appendix

A. Generator Architecture Diagram



140204859719464
dense_5: Dense | input: (None, 2048) | output: (None, 131072)
batch_normalization_v1_20: BatchNormalizationV1 | input: (None, 131072) | output: (None, 131072)
re_lu_4: ReLU | input: (None, 131072) | output: (None, 131072)
reshape_1: Reshape | input: (None, 131072) | output: (None, 16, 16, 512)
conv2d_transpose_4: Conv2DTranspose | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
batch_normalization_v1_21: BatchNormalizationV1 | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
re_lu_5: ReLU | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
conv2d_transpose_5: Conv2DTranspose | input: (None, 16, 16, 512) | output: (None, 32, 32, 512)
batch_normalization_v1_22: BatchNormalizationV1 | input: (None, 32, 32, 512) | output: (None, 32, 32, 512)
re_lu_6: ReLU | input: (None, 32, 32, 512) | output: (None, 32, 32, 512)
conv2d_transpose_6: Conv2DTranspose | input: (None, 32, 32, 512) | output: (None, 64, 64, 256)
batch_normalization_v1_23: BatchNormalizationV1 | input: (None, 64, 64, 256) | output: (None, 64, 64, 256)
re_lu_7: ReLU | input: (None, 64, 64, 256) | output: (None, 64, 64, 256)
conv2d_transpose_7: Conv2DTranspose | input: (None, 64, 64, 256) | output: (None, 128, 128, 3)

Fig. 5. Generator Architecture

B. Discriminator Architecture Diagram

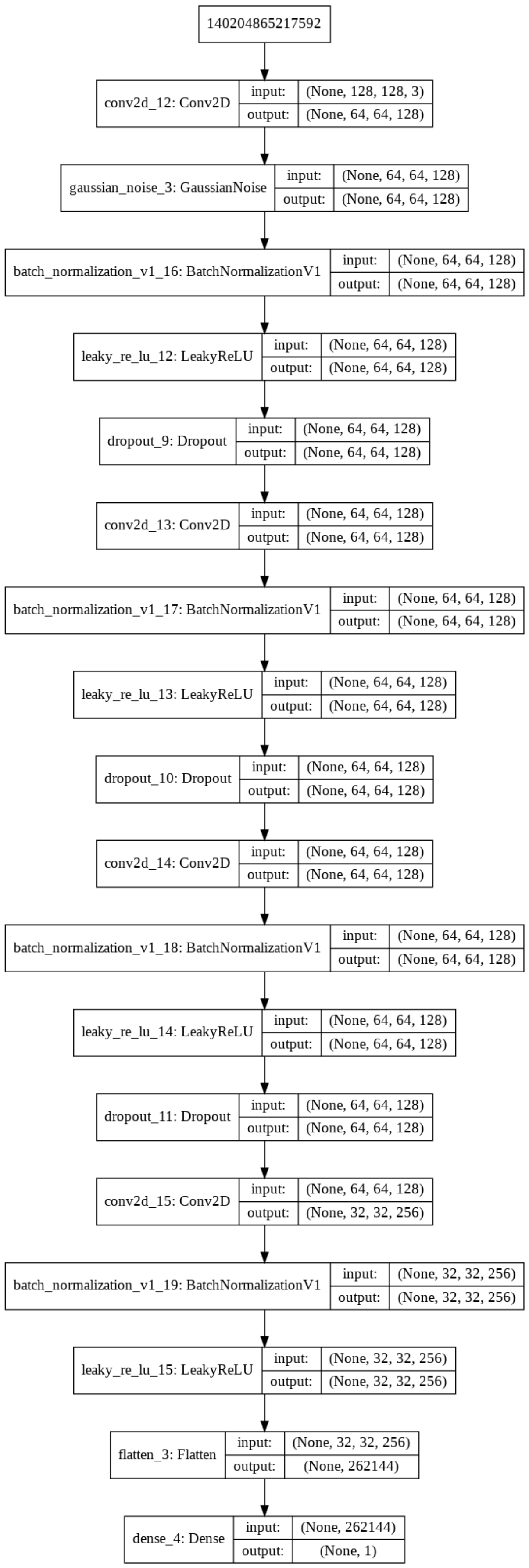

140204865217592
conv2d_12: Conv2D | input: (None, 128, 128, 3) | output: (None, 64, 64, 128)
gaussian_noise_3: GaussianNoise | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
batch_normalization_v1_16: BatchNormalizationV1 | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
leaky_re_lu_12: LeakyReLU | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
dropout_9: Dropout | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
conv2d_13: Conv2D | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
batch_normalization_v1_17: BatchNormalizationV1 | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
leaky_re_lu_13: LeakyReLU | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
dropout_10: Dropout | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
conv2d_14: Conv2D | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
batch_normalization_v1_18: BatchNormalizationV1 | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
leaky_re_lu_14: LeakyReLU | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
dropout_11: Dropout | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
conv2d_15: Conv2D | input: (None, 64, 64, 128) | output: (None, 32, 32, 256)
batch_normalization_v1_19: BatchNormalizationV1 | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
leaky_re_lu_15: LeakyReLU | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
flatten_3: Flatten | input: (None, 32, 32, 256) | output: (None, 262144)
dense_4: Dense | input: (None, 262144) | output: (None, 1)

Fig. 6. Discriminator Architecture