# Applications of Generative Adversarial Networks: Seismic Image Generation

Juan Meléndez

Department of Geoscience

Faculty of Science

University of Calgary

## Abstract

Amongst the various classes of artificial neural networks, Generative Adversarial Networks (GANs) have proven highly effective at creating realistic, completely synthetic data. This is accomplished by competitively pairing two networks, namely a generator and a discriminator, which create random data and then compare it to training data; over many iterations, the network learns to generate data of similar quality to the training set. Though this process can be applied to many scenarios, GANs are especially popular in image generation for several reasons: they can produce realistic images that are nearly indistinguishable from real ones, they do not require labeled data sets, and they can be adapted to dissimilar data sets with relative ease. An application of GANs is studied here as it pertains to geophysics, where generative, high quality results are necessary for the production useful training data, which may be otherwise difficult to obtain. Using various machine learning libraries in Python, a basic GAN network is trained to create artificial seismic images.

## Introduction

Many general-purpose image manipulation programs exist, and for the most part they can manage a broad variety of tasks. Traditional photo-editing software does well when modifying existing images or creating new, singular pieces of work, but when producing large quantities of highly refined images form scratch, the programs tend to preform extremely inefficiently or not at all (Wu et al., 2019). Machine learning can address some of these problem, with GANs being a subset currently enjoying lots of scholarly interest. Quickly after their original debut in 2014 (Goodfellow et al. 2014), their qualities as easily adaptable, unsupervised systems that produce high quality images has resulted in their quick adoption in various disciplines.

### *Related work*

GANs are being implemented in seemingly disparate fields as certain data processing needs are widespread. In practice most uses are related to image management, especially in generation, classification, editing, and resolution up-scaling (Wu et al., 2019). Examples include the creation of non-existent galaxies (Smith & Geach, 2019), generating photo-realistic faces (figure 1), de-noising MRI scans (Yi et al., 2019), increasing photo resolution (Bulat et al., 2018), and the identification of specific faces in crowds (Wang et al., 2016). GANs have also proven useful for other types of data, such as speech synthesis (Kaneko et al., 2017). Research continues into many topics, for example improving image quality by preprocessing training data (Denton et al., 2015), experimenting with semi-supervised methods (Creswell et al., 2018), and improving GAN stability (Metz et al., 2016).

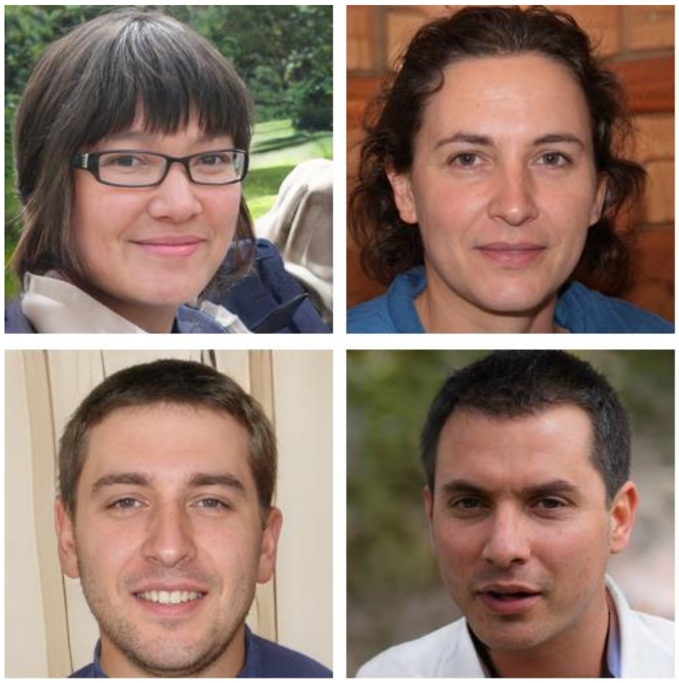


Figure 1: These images have been created using generative adversarial networks. By training on photographs of real people, life-like pictures of non-existent faces can be made. (https://www.thispersondoesnotexist.com)

As it pertains to geoscience, GANs have great potential in the generation of realistic, non-existent geological scenarios, which are useful for training purposes. Generated scenarios are desirable for a few reasons; obtaining new data may require expensive acquisitional techniques, while processing it may be less efficient than generating it artificially (Chan & Elsheikh, 2017) (Jo et al., 2020). Fortunately, geoscientists often work with visual representations of data. Well logs, formation maps, river models, and seismic profiles are all examples of commonly used data that must be represented visually for interpretation purposes, and so all of these can be exploited by GANs. This project presents one example, where machine learning libraries in Python are used with the aim of creating realistic looking artificial seismic profiles.

## Background

### *Generative Adversarial Networks*

GANs are best described as an iterative system of two networks, a generator and discriminator, in direct competition with each other. An analogy is commonly used to describe this game, where the generator is like an art forger, producing fake paintings, and the discriminator is like an art expert, comparing the fake images to real ones with the aim of authenticating the paintings (Dumoulin et al., 2017). Over many cycles, the forger learns to produce more convincing forgeries, while the discriminator learns to better tell the fakes apart, eventually pushing the forger to create perfect or near-perfect paintings.

In technical terms, a standard GAN implicitly models probabilistic distributions of data (Smith & Geach, 2019). The discriminator (figure 2) receives a latent-space representation of an image from the generator, which is then mapped to a probability distribution of the likelihood of an image being real. In the same iteration, an image is taken from the training data, converted to latent space, and similarly mapped to a probability to determine its authenticity. Often the system is left to run in a completely unsupervised fashion, which is an advantage in situations where annotating large amounts of training data would be unfeasible. Once both images have their authenticity probabilities assigned, the probabilities are compared, on which basis the discriminator then decides one image to be real and one to be fake. Importantly, the discriminator is completely blind to the actual real/fake labels, ensuring that the ability to make correct decisions is purely the result of the GAN being trained. Once the real/fake decision has been assigned by the discriminator, it is compared the actual real/fake labels. Whether the decision was correct or not provides the one and only source of feedback for the system, which is used by the discriminator to improve its performance, which after enough iterations should produce appreciably better results. Incidentally, the error rate of the discriminator serves as a measure of the performance of the GAN; in theory the system will eventually reach a balance between the discriminator and generator, as generated images will be indistinguishable from real ones, causing the discriminator to have an error rate of 50%. In practice most GANs do not use this value to directly determine stopping conditions, largely because it is difficult to create a totally stable system that can maintain a 50% balance (Mao et al., 2017).

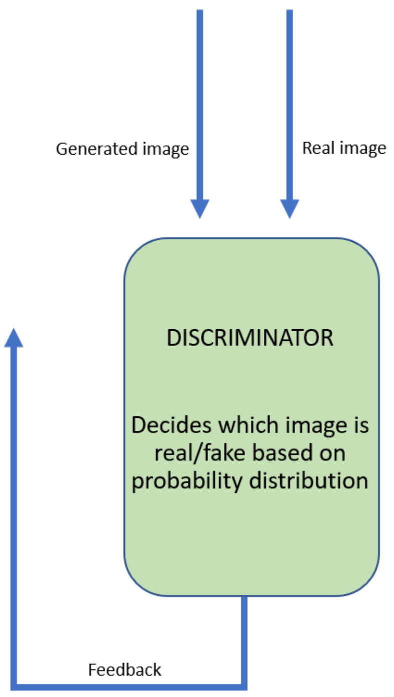

Generated image
Real image
DISCRIMINATOR
Decides which image is real/fake based on probability distribution
Feedback

Figure 2: Summarized representation of the basic functions handled by a discriminator.

Unlike the discriminator, the generator (figure 3) never has access to the training images (Creswell et al., 2018). The only way of improving its performance is by receiving the real/fake feedback response from the discriminator, causing the generator to shift model weights accordingly, usually in small step-sizes to preserve stability. The next cycle's image will be slightly different as a result. Changes made over a single iteration may or may not improve quality, but in time the changes accumulate to produce a much-improved version that more closely resembles the training data. The generation process itself begins by creating a random vector of appropriate dimensions and feeding it to the generator. This vector is the latent-space equivalent of a nonsense image, which a trained generator will adjust to resemble something closer to the training data. This rather simple procedure ensures that once training is complete, random vectors can be continuously fed into the generator to produce a potentially infinite number of good quality images. The entire system can be thought of as looped generator and discriminator, with the generator sending images to the discriminator and receiving feedback in return (figure 4).

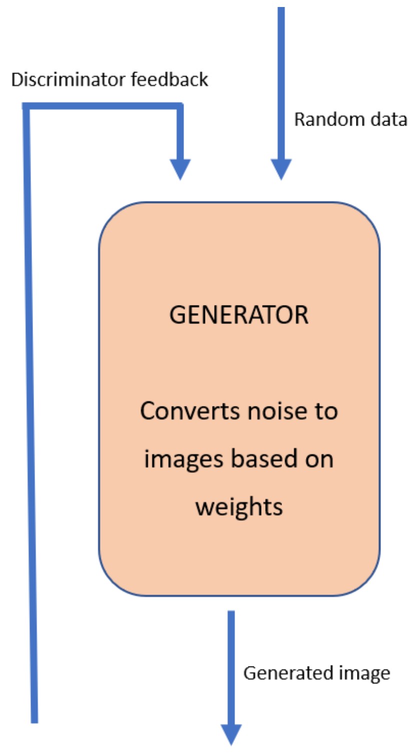


Figure 3: Summarized representation of the basic functions handled by a generator.

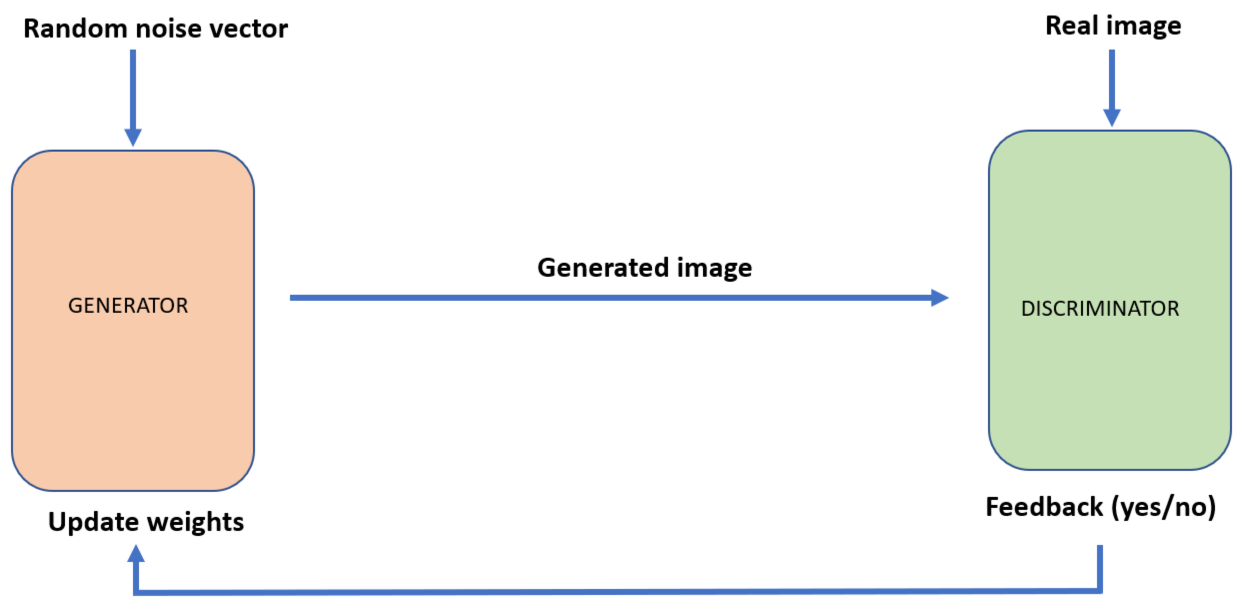


Figure 4: Visual representation of a complete GAN.

*Deep Convolution Generative Adversarial Networks*

Existing examples of image-generating GANs are abound. Generically speaking, these GANS are of a class called Deep Convolution Generative Adversarial Networks, which are a type of Artificial Neural Network (ANN). Inspired by biological neural systems, ANNs describe a computational processing system comprised of many interconnected computational nodes called neurons, which entwine in a distributed manner to collectively learn to optimize the network's output. If many layers of neurons are used it is called deep learning (Albawi & Mohammed, 2017).

Convolutional Neural Networks (CNNs) are very similar to ANNs but are distinct in that there are certain layer-types CNNs must have, namely convolutional layers and pooling layers. Convolutional layers control neuron output by finding the scalar product between neuron weights; the layer comes with an attached rectified linear unit (ReLU) to apply an elementwise activation function. Convolutional layers work by convolving data across small spatial dimensionalities called kernels, which are evenly spread across the input. It is for each kernel element that the scalar product is found, which defines when a kernel will activate. Pooling layers down-sample data to decrease complexity within the activation. Other layers will likely be required, such as an input layer to handle the image data. When these are stacked a CNN is created, and because CNNs require multiple layers, all CNNs are deep by default (O'Shea & Nash, 2015).

An advantage of Deep Convolution Networks over other ANNs is that the use of convolutional layers allows for spatial down-sampling and up-sampling operators to be learned during training, which improves efficiency. This is especially important for generating detailed images (Radford et al., 2016), as they contain a lot of information. Independently creating all the components such a system would normally be a complicated and laborious process, but the availability of libraries in Python greatly simplifies matters. Keras is one such example, which serves as a library for artificial neural networks and so contains functions for the generator, discriminator, the required CNN layers, and many other necessary features.

Francois Chollet, the library's creator, describes a basic convolutional GAN using Keras (*Deep Learning with Python,* Chollet, p. 403, 2017). Designed to work with relatively small 32x32 pixel images, the program produces pictures of frogs after training on an online dataset of real images (figure 5). Many sophisticated techniques exist that define the stopping criterion, such as using the least square loss function of the discriminator (Mao et al., 2017), but the Chollet model simply runs for a predetermined number of cycles. For basic projects, this approach may yield satisfactory results, but a reasonable value must be empirically determined over several runs or known previously from related work. Indeed, many useful tweaks have appeared in literature that further optimize GANs (Radford et al., 2016). Some commonly used ones appear in the Chollet model: sampling the random vector from a Gaussian distribution instead of a uniform distribution, introducing some randomness via dropout layers and random noise, using LeakyReLU activation functions instead of ReLU (to allow for small negative activation values), and replacing max pooling operations with strided convolutions (Chollet, p. 403, 2017). Many such tricks are required to avoid the GAN getting "stuck" before reaching a 50% error rate, as GAN instability makes them susceptible to converging around local optimums instead.

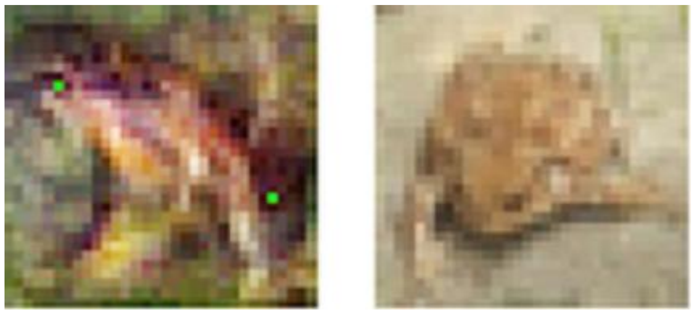


Figure 5: Generated image (left) vs a real image (right).
(*Deep Learning with Python*, Francois Chollet, 2017)

Optimization techniques not explored in the model include heuristic averaging, which penalizes the network for deviating from the running average, and mini-batch discrimination, which groups the results into small batches and finds the distance between samples, both of which help prevent the generator becoming stuck (Creswell et al., 2018). Beyond simple optimization, GANs can be combined with other machine learning tools to create powerful networks, such as using Reinforcement Learning to better train GANs. Reinforcement Learning (RL) works by creating an

RL agent, which must perform some task by interpreting its environment. The agent is not given instructions on how to do this, but rather it learns to do so by being rewarded or punished based on its performance. A Reinforcement Learning Generative Adversarial Network (RL-GAN) therefore compares generator results with training data and rewards or punishes the GAN accordingly, creating a more robust generator model (Sarmad et al., 2019). Another tactic is using evolutionary algorithms to create Evolutionary Generative Adversarial Networks (E-GANs). These work by treating the training process as an evolutionary problem, where generators produce "offspring" as they train. These new generators will adjust weights in slightly different ways while producing children of their own, eventually causing their lineages to diverge. Weakly performing offspring are culled, pushing the system to produce highly effective generators (Wang et al., 2019).

## Methods

### *GAN Architecture*

Necessary framework elements for the GAN are provided by the aforementioned Chollet model and described in *Deep Learning with Python*. Significant increases in runtime were necessary to accommodate the larger 128x128 seismic images instead of the 32x32 frogs, as well as doubling the number of iterations to 20,000 in the pursuit of higher quality results. If desired, the program (Appendix A) is known to run in Python 3.7, and requires the keras, nympy, and opencv libraries be installed. Matplotlib can also be installed if sample images are desired.

The generator (figure 6) (Appendix B) begins every cycle by taking the input, a 128-length latent-space randomized vector, and creating a 64x64 128-channel feature map. This is followed by a convolutional layer. Data is then up-sampled to a 128x128 256-channel feature map using a strided convolution of kernel size 4 and stride 2. Two more convolutional layers follow. For the activation functions, LeakyReLU is used in lieu of ReLU. The final step requires converting the data to a 128x128 array, which matches the dimensions of the desired images. For this step, a tanh activation function is used.

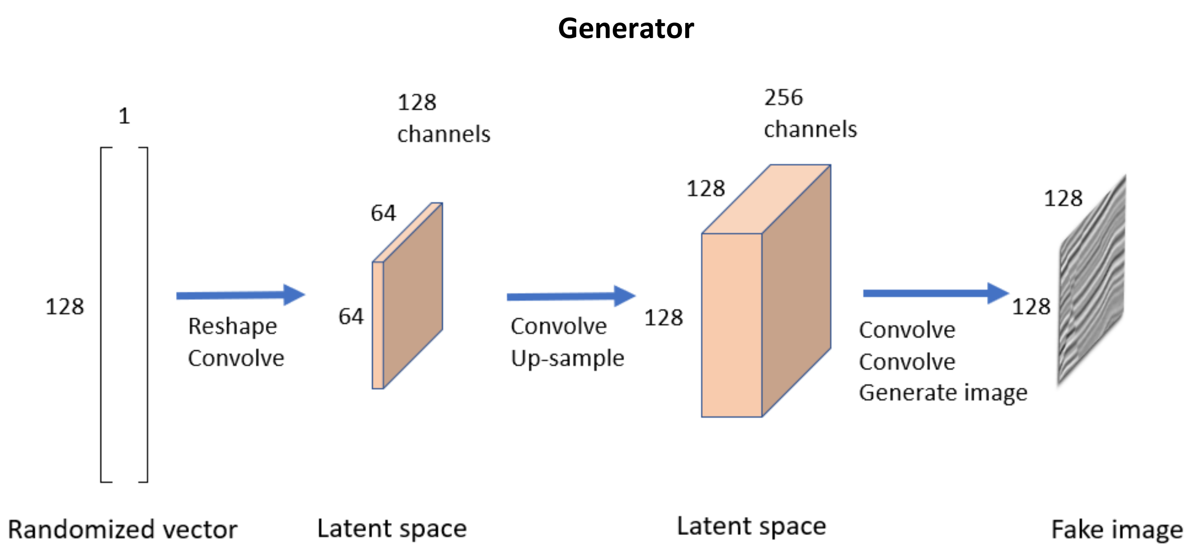


Figure 6: Visual representation of generator architecture. A summary of all the layers can be seen in Appendix C.

After taking a 128x128x1 input, the discriminator (figure 7)(Appendix D) converts it to a 126x126 128-channel feature map and runs the data through four convolutional layers, again using LeakyReLU activation functions. Using the same kernel size and stride as before, the data is down-sampled to 14x14 128 channels; drop-out is intentionally included to add randomness to the system. The data is then flattened back into a vector so the discriminator model can classify the image as real or fake.

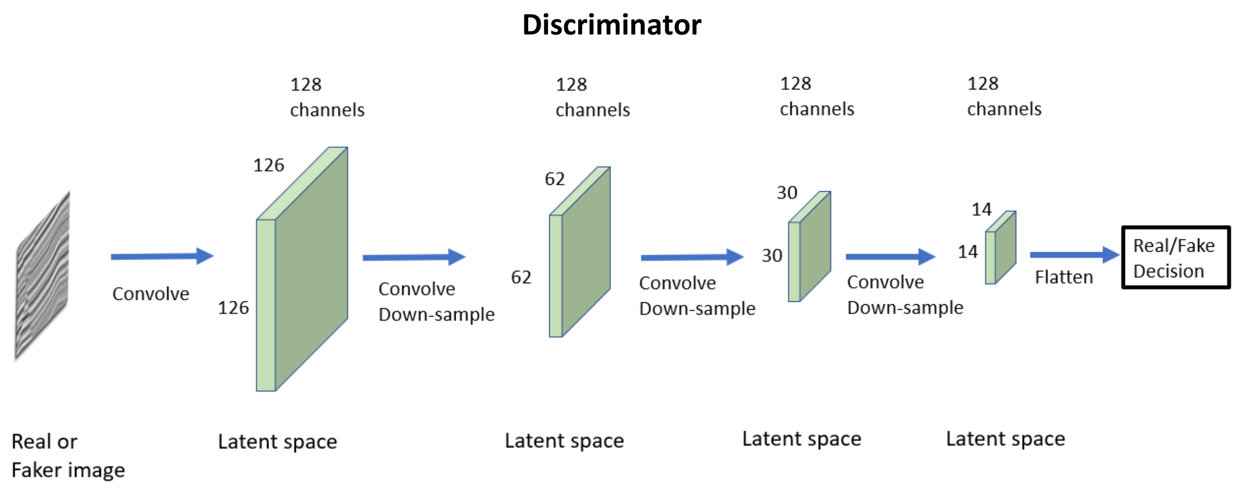


Figure 7: Visual representation of discriminator architecture. A summary of all the layers can be seen in Appendix F.

Unlike the generator, the discriminator handles two images. In every iteration it accepts the generator's output as well as an image from the training data (figure 8), which in this case is a local archive of 2,600 seismic profiles in greyscale, measuring 128x128 pixels. Only once both images have been processed can a probability be assigned, and a decision made. Following this, feedback is sent to the generator and discriminator models in order to train them.

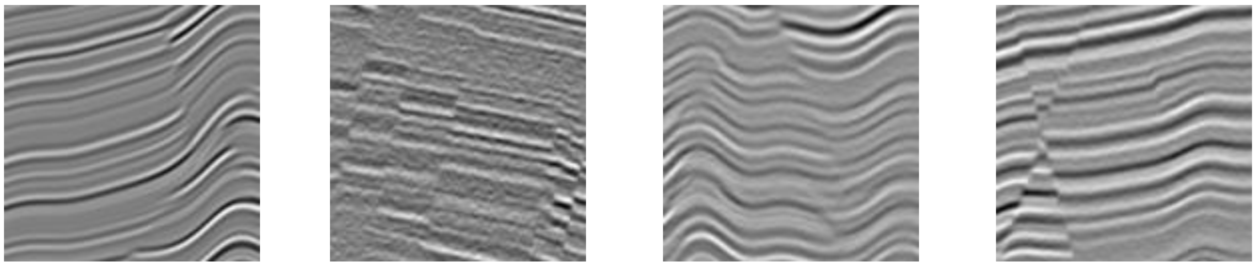


Figure 8: Examples of real images taken from the training set. Before further processing, these PNG files are converted to a modified RGB scale, with each pixel's shade being represented by a number between 0 and 1.

With the generator and the discriminator built, they must be linked inside a for loop. Within this loop are a few more important processes: the creation of the random vector, adding noise to the generated and real images before feeding it to the discriminator, and saving the generated images in a local directory.

**Results**

The first complete run of the GAN was done for 10,000 intervals, this being the number the Chollet model successfully used. Findings are presented (figure 9) in intervals of 1000 cycles, beginning at 0, whose image is the result of a generator that has had no training at all. It can be observed that early on the system shows a clear trend towards better image quality, with image 0 being entirely random, and image 10,000 approximating the near-horizontal features of the training data. The most marked improvements appear to take place sometime before the 5,000th iteration.

Image 10,000 represents a supposedly fully trained result, but clearly the model cannot yet produce images resembling seismic sections. A lack of training was suspected, so a run of 20,000 cycles was completed. The results can be seen in figure 10, with image 20,000 being the fully trained image.

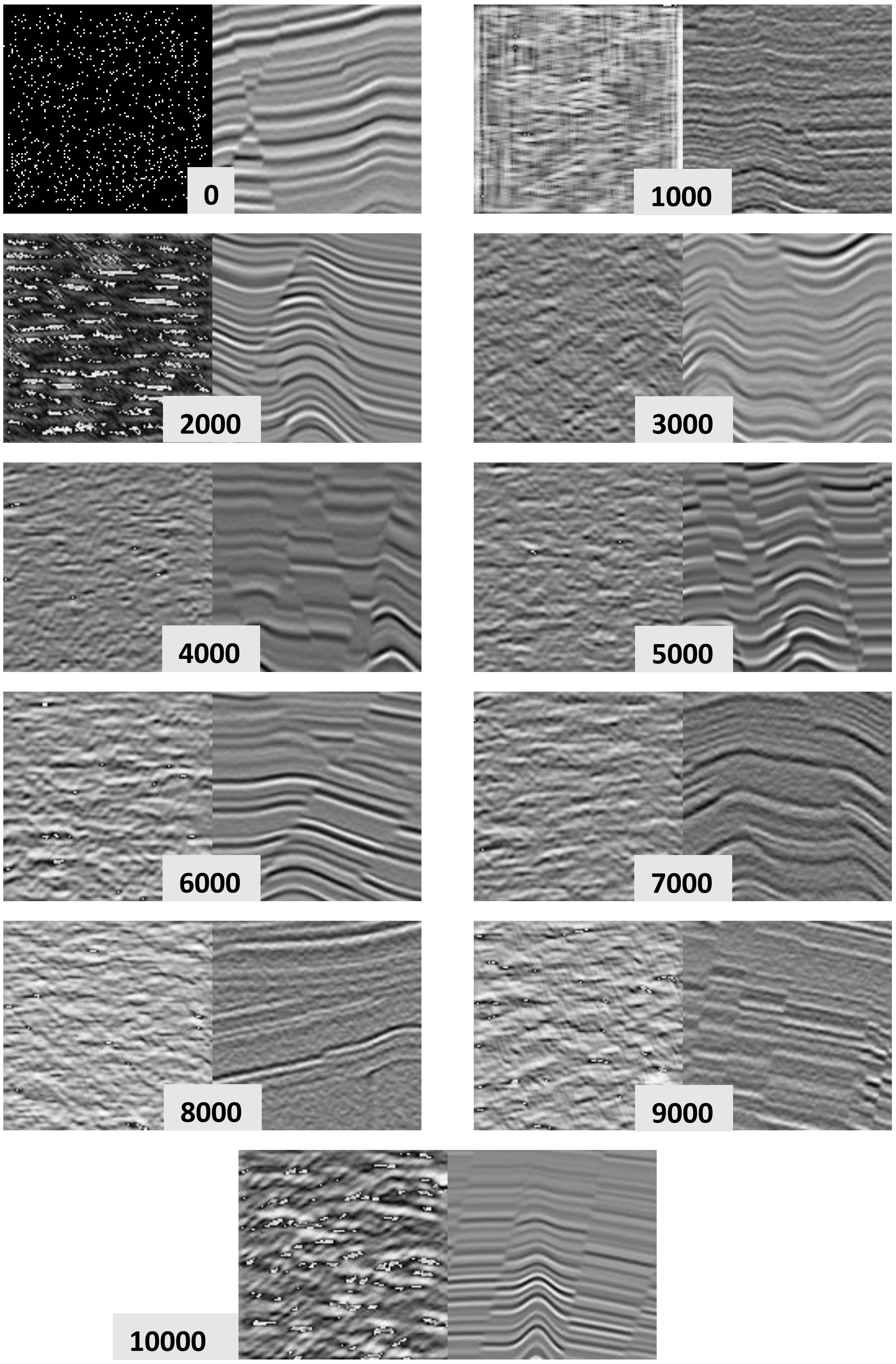


Figure 9: Results for 10,000 iterations, including corresponding real images.

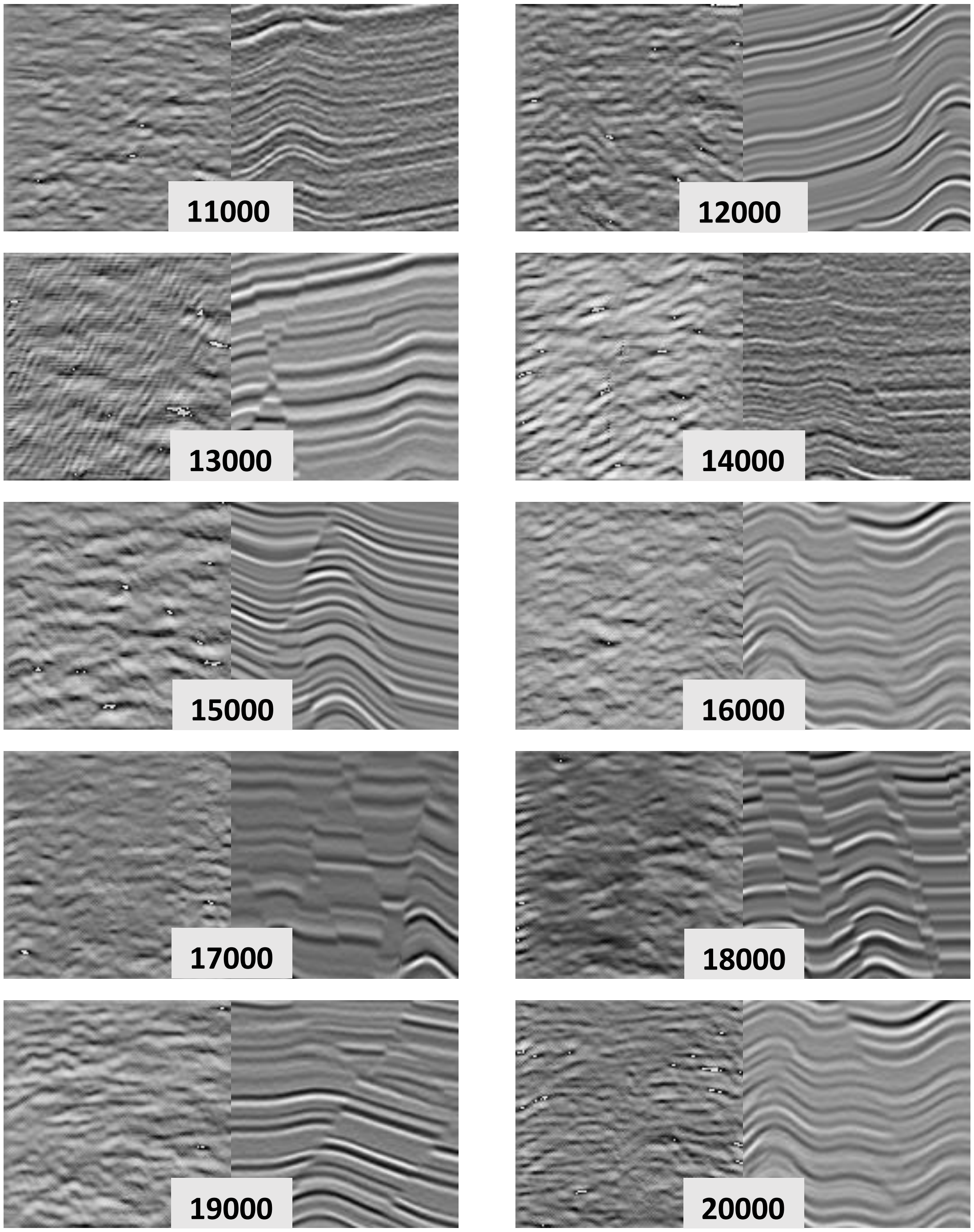


Figure 10: Results for iterations 11,000 to 20,000.

Despite doubling the number of iterations, the newly generated images show little increase in quality (figure 11), implying that the number of iterations was in fact sufficient, and that there are deeper issues that require fixing. Two possibilities come to mind: the networks are being undertrained and so should be be deepened, or the generator has become stuck and is no longer improving image quality. The first problem may be solved by adding more layers, while the second may be addressed by using any of the techniques addressed in the Methods section, such as mini-batch discrimination.

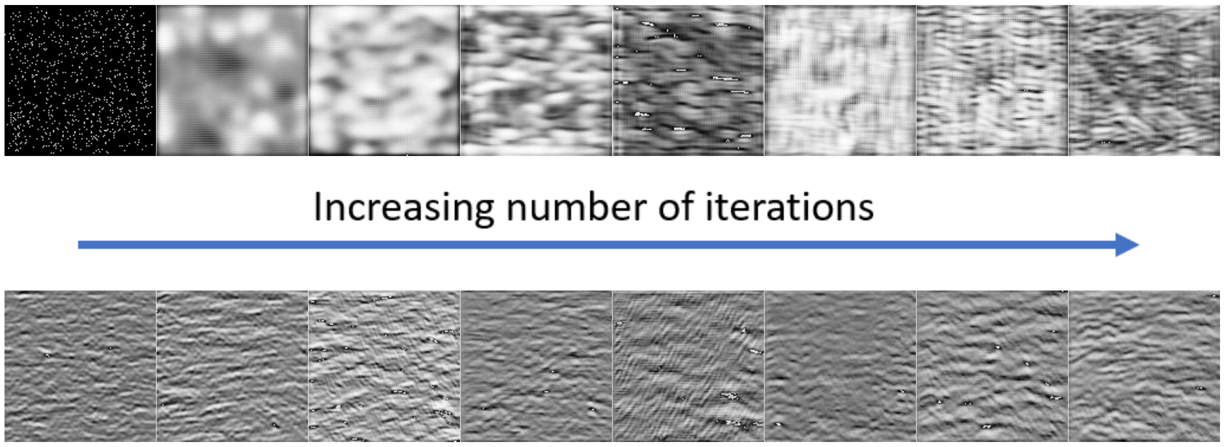


Figure 11: Evolution of generated images (scaled down). The top row contains images from cycles 0 – 700, sampled every 100 iterations, and the bottom row contains images from cycles 5,000 – 19,000, sampled every 2,000 iterations. It is normal for earlier cycles to evolve more quickly, as these have had little training and as such will improve rapidly. However, the bottom row is taken from a very wide range and shows little to no improvement, implying that the GAN has become stuck at a local optimum.

Unfortunately, the runtime is several hours long, requiring nearly an entire day to complete unless powerful hardware is used. This is one of the major pitfalls of the program, as any adjustments become difficult to test. Even basic alterations, such as increasing training weight step-size to reduce runtime, require huge time investments, though this only affects the training process. Once the training weights are set, these runtime issues become irrelevant as new images can be created almost instantaneously. As a proof-of-concept the GAN is still viable, but more man-hours are required to optimize for efficiency and image quality.

## Conclusion

Overall, this application of GANs to generate artificial seismic sections was a moderate success. Thanks to pre-existing libraries, trainable neural networks can be readily created, in this case being complex enough to generate images with some desired characteristics. Realistically, the images fall short of resembling real seismic data, but this is not necessarily due to having an incorrect approach, as it may be that the network requires further adjustment. Seeing how GANs can be computationally expensive, time constraints are the main stumbling block as they prevent meaningful optimization.

Assuming more time can be had to adjust the program and test those modifications, this GAN has great potential. Many techniques already exist to improve efficiency and image quality, such as heuristic averaging or mini-batch discrimination, both of which help GANs avoid becoming stuck. This appears to be the cause of the GAN's mediocre quality, as past a certain number of iterations the generator's output becomes predictable, evidenced by the similarity of most images past 5,000 iterations. Another solution may be to build a deeper neural network with more layers. Fortunately, GANs are popular enough that if neither approach works more sophisticated remedies may be found, be it by using RL-GANS, E-GANs, or through other means.

## Appendix

### *A) Library importation and initialization*

```
import keras
from keras import layers
from keras.preprocessing import image
import numpy as np
import cv2
import os

# Add for GPU BEFORE JSON
from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession

config = ConfigProto()
config.gpu_options.allow_growth = True
session = InteractiveSession(config=config)

# The following is required to resolve a conflicting/duplicate libraries issue, but may not be necessary on all systems:
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

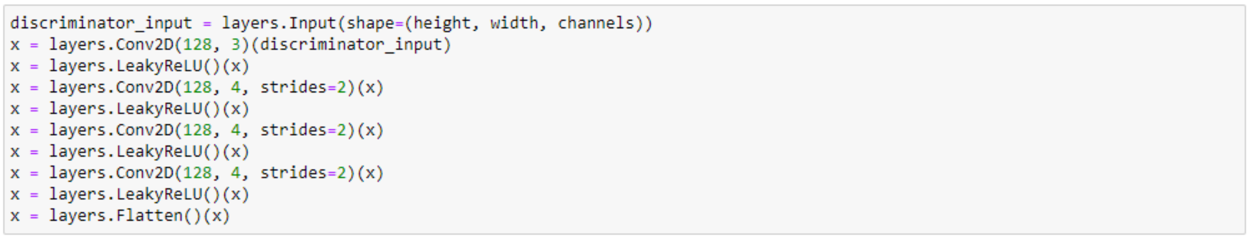
### *B) Generator*

```
# Image parameters are set. In this case, the program is set to accept 128 x 128 pixel images. Larger images are possible,
# but may increase an already lengthy computation time.
latent_dim = 128
height = 128
width = 128
channels = 1

# Now the generator can be created. Keras provides useful functions for both the generator and discriminator.
# Generators work by creating images in the latent space (raw data as opposed to a true image). The uniqueness of each
# image is accomplished sampling a random noise vector every iteration, then running it through the generator which modifies
# it to resemble a real image. With every cycle, changes in the generator's weights will improve this conversion, resulting
# in better images.
generator_input = keras.Input(shape=(latent_dim,))

# The input must be converted into a 64 x 64 128-channels feature map:
x = layers.Dense(128 * 64 * 64)(generator_input)

# LeakyReLU is used in lieu of standard ReLU. It has less severe constraints as it allows small negative activation values.
# Based on empirical evidence, this results in better results for GANs:
x = layers.LeakyReLU()(x)
x = layers.Reshape((64, 64, 128))(x)

# Convolution layers are added, and up-sampling to 128 x 128 is done:
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

# Upsampling to 128 x 128:
x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)


# Produce a 128 x 128 feature map, which is now the shape of the actual images.
# Again, it has been emprically shown that a tanh activation is more effective that the more common sigmoid.
x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()
```

*C) Generator summary*

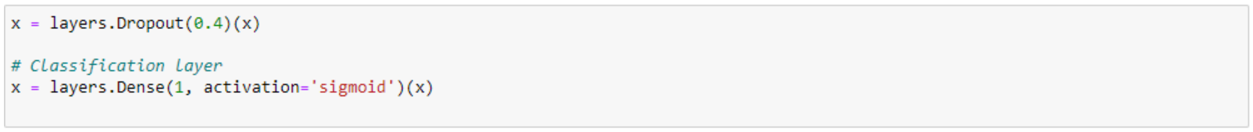
```
Model: "model_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_10 (InputLayer)        (None, 128)               0         
_________________________________________________________________
dense_7 (Dense)              (None, 524288)            67633152  
_________________________________________________________________
leaky_re_lu_28 (LeakyReLU)   (None, 524288)            0         
_________________________________________________________________
reshape_4 (Reshape)          (None, 64, 64, 128)       0         
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 64, 64, 256)       819456    
_________________________________________________________________
leaky_re_lu_29 (LeakyReLU)   (None, 64, 64, 256)       0         
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 128, 128, 256)     1048832   
_________________________________________________________________
leaky_re_lu_30 (LeakyReLU)   (None, 128, 128, 256)     0         
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 128, 128, 256)     1638656   
_________________________________________________________________
leaky_re_lu_31 (LeakyReLU)   (None, 128, 128, 256)     0         
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 128, 128, 256)     1638656   
_________________________________________________________________
leaky_re_lu_32 (LeakyReLU)   (None, 128, 128, 256)     0         
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 128, 128, 1)       12545     
=================================================================
Total params: 72,791,297
Trainable params: 72,791,297
Non-trainable params: 0
_________________________________________________________________
```

## D) Discriminator



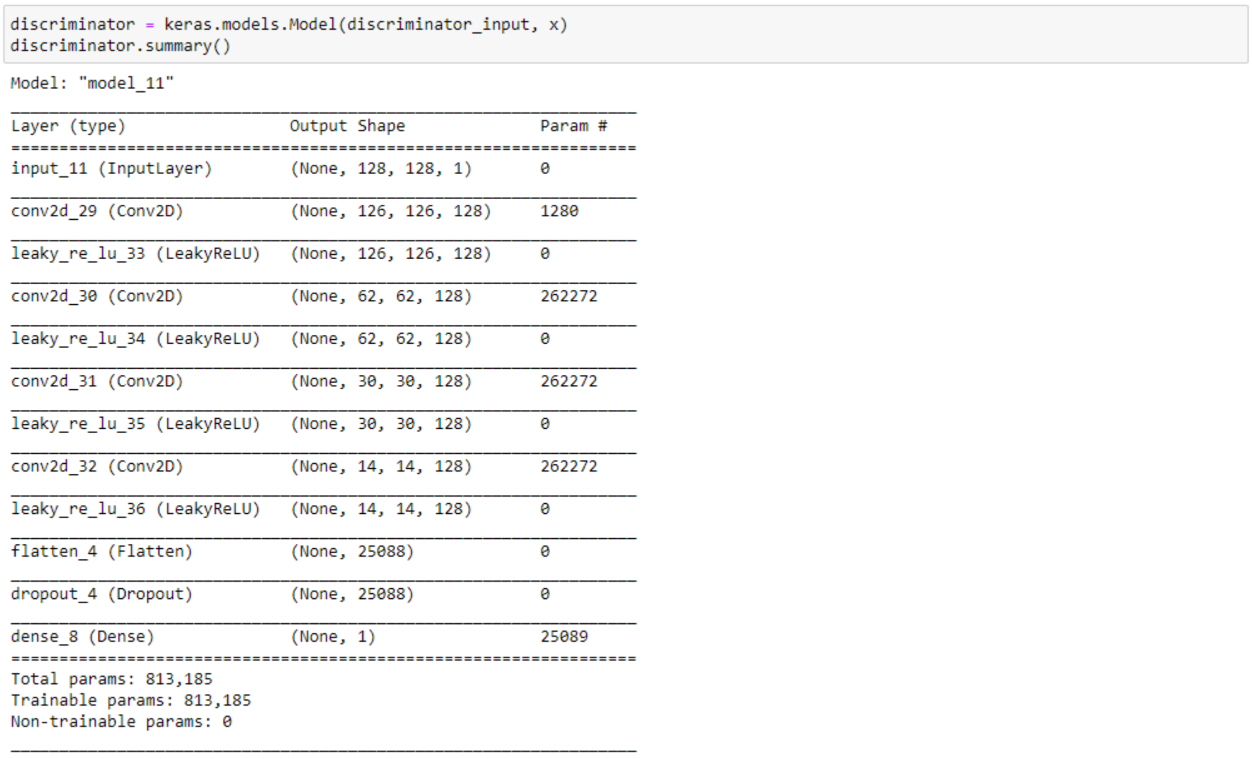
```
discriminator_input = layers.Input(shape=(height, width, channels))
x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Flatten()(x)
```

## E) Discriminator drop-out layer



```
x = layers.Dropout(0.4)(x)

# Classification layer
x = layers.Dense(1, activation='sigmoid')(x)
```

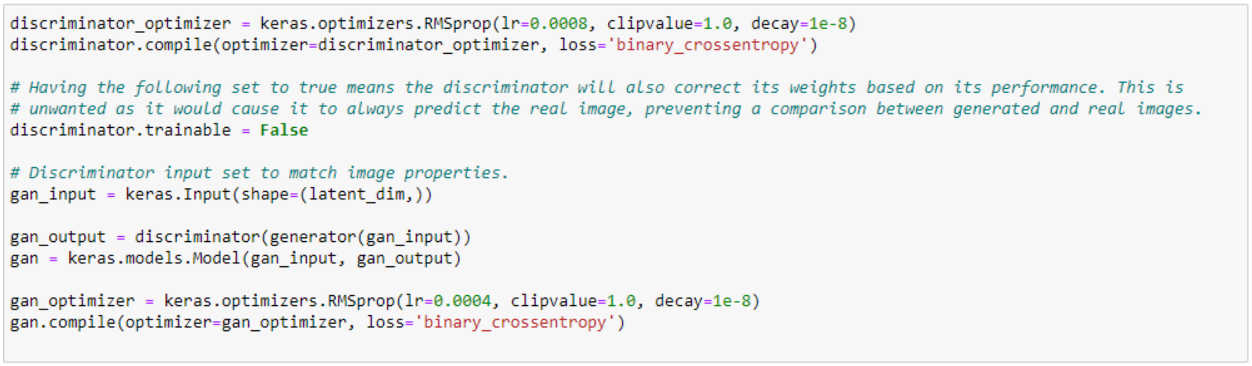
## F) Discriminator summary



```
discriminator = keras.models.Model(discriminator_input, x)
discriminator.summary()
```

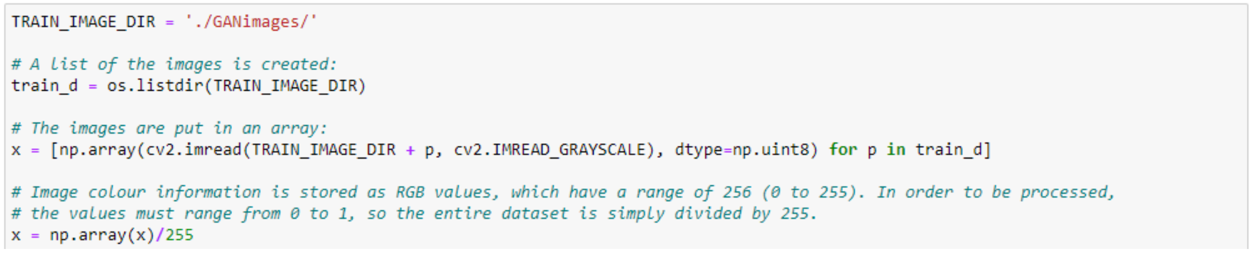
```
Model: "model_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_11 (InputLayer)        (None, 128, 128, 1)       0         
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 126, 126, 128)     1280      
_________________________________________________________________
leaky_re_lu_33 (LeakyReLU)   (None, 126, 126, 128)     0         
_________________________________________________________________
conv2d_30 (Conv2D)           (None, 62, 62, 128)       262272    
_________________________________________________________________
leaky_re_lu_34 (LeakyReLU)   (None, 62, 62, 128)       0         
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 30, 30, 128)       262272    
_________________________________________________________________
leaky_re_lu_35 (LeakyReLU)   (None, 30, 30, 128)       0         
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 14, 14, 128)       262272    
_________________________________________________________________
leaky_re_lu_36 (LeakyReLU)   (None, 14, 14, 128)       0         
_________________________________________________________________
flatten_4 (Flatten)          (None, 25088)             0         
_________________________________________________________________
dropout_4 (Dropout)          (None, 25088)             0         
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 25089     
=================================================================
Total params: 813,185
Trainable params: 813,185
Non-trainable params: 0
_________________________________________________________________
```

## G) Learning rate



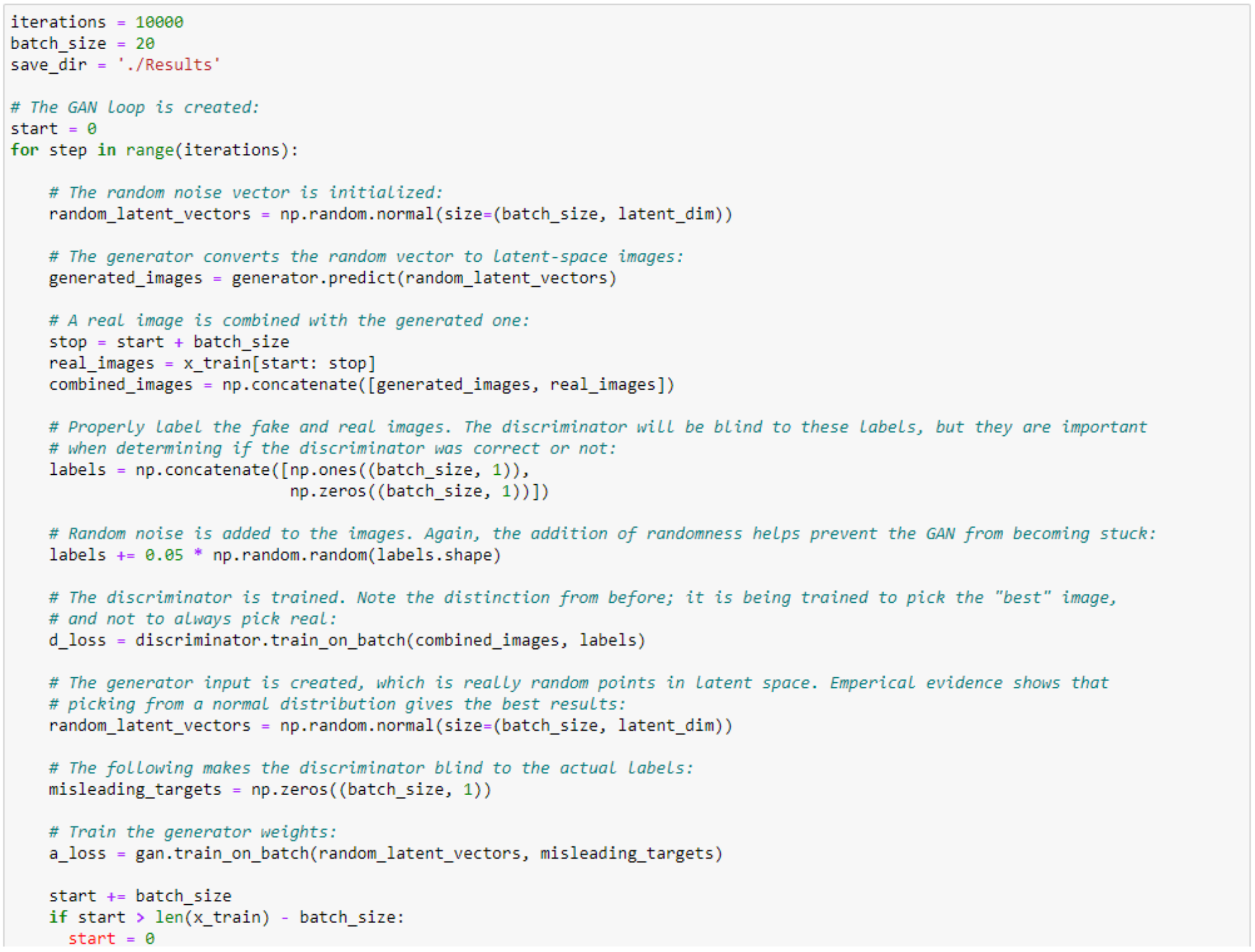
```
discriminator_optimizer = keras.optimizers.RMSprop(lr=0.0008, clipvalue=1.0, decay=1e-8)
discriminator.compile(optimizer=discriminator_optimizer, loss='binary_crossentropy')

# Having the following set to true means the discriminator will also correct its weights based on its performance. This is
# unwanted as it would cause it to always predict the real image, preventing a comparison between generated and real images.
discriminator.trainable = False

# Discriminator input set to match image properties.
gan_input = keras.Input(shape=(latent_dim,))

gan_output = discriminator(generator(gan_input))
gan = keras.models.Model(gan_input, gan_output)

gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')
```

## H) Training data is prepared

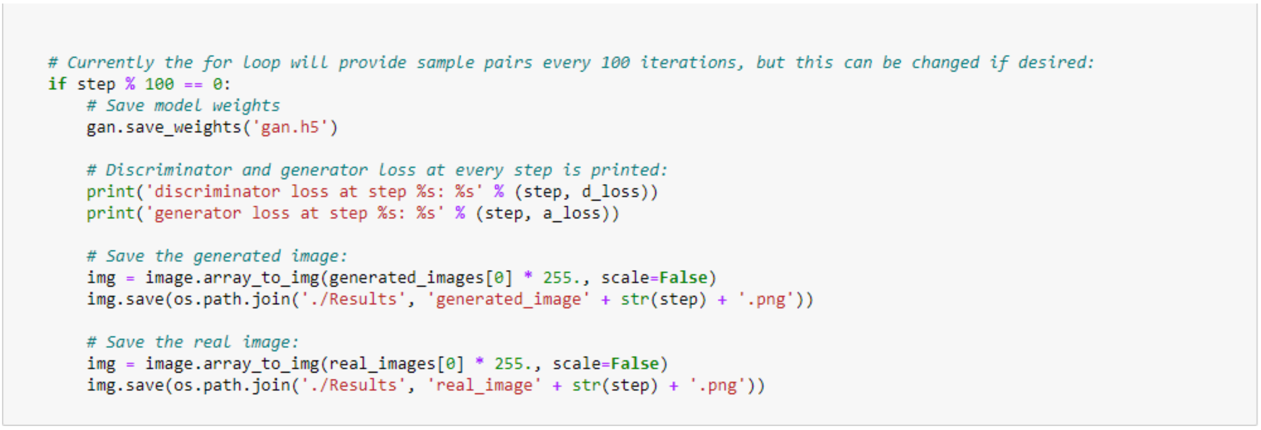


```
TRAIN_IMAGE_DIR = './GANimages/'

# A list of the images is created:
train_d = os.listdir(TRAIN_IMAGE_DIR)

# The images are put in an array:
x = [np.array(cv2.imread(TRAIN_IMAGE_DIR + p, cv2.IMREAD_GRAYSCALE), dtype=np.uint8) for p in train_d]

# Image colour information is stored as RGB values, which have a range of 256 (0 to 255). In order to be processed,
# the values must range from 0 to 1, so the entire dataset is simply divided by 255.
x = np.array(x)/255
```

*I) For loop is created*



```
iterations = 10000
batch_size = 20
save_dir = './Results'

# The GAN loop is created:
start = 0
for step in range(iterations):

    # The random noise vector is initialized:
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))

    # The generator converts the random vector to latent-space images:
    generated_images = generator.predict(random_latent_vectors)

    # A real image is combined with the generated one:
    stop = start + batch_size
    real_images = x_train[start: stop]
    combined_images = np.concatenate([generated_images, real_images])

    # Properly label the fake and real images. The discriminator will be blind to these labels, but they are important
    # when determining if the discriminator was correct or not:
    labels = np.concatenate([np.ones((batch_size, 1)),
                             np.zeros((batch_size, 1))])

    # Random noise is added to the images. Again, the addition of randomness helps prevent the GAN from becoming stuck:
    labels += 0.05 * np.random.random(labels.shape)

    # The discriminator is trained. Note the distinction from before; it is being trained to pick the "best" image,
    # and not to always pick real:
    d_loss = discriminator.train_on_batch(combined_images, labels)

    # The generator input is created, which is really random points in latent space. Emperical evidence shows that
    # picking from a normal distribution gives the best results:
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))

    # The following makes the discriminator blind to the actual labels:
    misleading_targets = np.zeros((batch_size, 1))

    # Train the generator weights:
    a_loss = gan.train_on_batch(random_latent_vectors, misleading_targets)

    start += batch_size
    if start > len(x_train) - batch_size:
      start = 0
```



```
    # Currently the for loop will provide sample pairs every 100 iterations, but this can be changed if desired:
    if step % 100 == 0:
        # Save model weights
        gan.save_weights('gan.h5')

        # Discriminator and generator loss at every step is printed:
        print('discriminator loss at step %s: %s' % (step, d_loss))
        print('generator loss at step %s: %s' % (step, a_loss))

        # Save the generated image:
        img = image.array_to_img(generated_images[0] * 255., scale=False)
        img.save(os.path.join('./Results', 'generated_image' + str(step) + '.png'))

        # Save the real image:
        img = image.array_to_img(real_images[0] * 255., scale=False)
        img.save(os.path.join('./Results', 'real_image' + str(step) + '.png'))
```

## References

1. Wu, H., Zheng, S., Zhang, J., Huang, K. 2019. GP-GAN: Towards Realistic High-Resolution Image Blending. *Proceedings of the 27th ACM International Conference on Multimedia (MM '19)*. (2019)

2. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. Generative Adversarial Nets. *NIPS* (2014).

3. Smith, M., Geach, J. Generative deep fields: arbitrarily sized, random synthetic astronomical images through deep learning, *Monthly Notices of the Royal Astronomical Society*, Volume 490, Issue 4, (2019) pp. 4985–4990

4. Yi, X., Walia, E., Babyn, P. Generative adversarial network in medical imaging: A review, *Medical Image Analysis*, Volume 58, (2019)

5. Creswell, White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., Bharath, A. Generative Adversarial Networks: An Overview, *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53-65, (2018)

6. Mao, X., Li, Q., Xie, H., Lau, R., Wang, Z., Smolley, S. Proceedings of the International Conference on Computer Vision (ICCV), (2017), pp. 2794-2802

7. Bulat, A., Yang, J., Tzimiropoulos, G. Proceedings of the European Conference on Computer Vision (ECCV) (2018), pp. 185-200

8. Wang, M., Fu, W., Hao, S., Tao, D., Wu, X.: Scalable semi-supervised learning by efficient anchor graph regularization. *TKDE* (2016)

9. Kaneko, T., Kameoka, H., Hojo, N., Ijima, Y., Hiramatsu, K., Kashino, K. Generative adversarial network-based postfilter for statistical parametric speech synthesis, *International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2017)

10. Denton, E., Chintala, S., Szlam, A., Fergus, R. Deep generative image models using a laplacian pyramid of adversarial networks. *Advances in Neural Information Processing Systems (NIPS)* (2015), pp. 1486–1494

11. Metz, L., Poole, B., Pfau, D., Sohl-Dickstein, J. Unrolled generative adversarial networks. arXiv:1611.02163, (2016)

12. Chan, S., Elsheikh, A. Parametrization and generation of geological models with generative adversarial networks. arXiv:1708.01810 [stat.ML] (2017)

13. Jo, H., Santos, J., Pyrcz, M. Conditioning well data to rule-based lobe model by machine learning with a generative adversarial network. *Energy Exploration & Exploitation* (2020)

14. Albawi, S., Mohammed, T., Al-Zawi, S. Understanding of a convolutional neural network. *2017 International Conference on Engineering and Technology (ICET)* (2017) pp. 1-6

15. O'Shea, K., Nash, R. An introduction to convolutional neural networks. arXiv preprint arXiv:1511.08458 (2015)

16. Chollet, Francois. 2017. *Deep Learning with Python*. New York, NY: Manning Publications.

17. Radford, A., Metz, L., Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks, *Proceedings of the 5th International Conference on Learning Representations (ICLR) - workshop track* (2016)

18. Sarmad, M., Lee, H., Kim, Y. Rl-gan-net: A reinforcement learning agent controlled gan network for real-time point cloud shape completion. *Proceedings of the Conference on Computer Vision and Pattern Recognition* (2019), pp. 5898-5907

19. Wang, C., Xu, C., Yao, X., Tao, D. Evolutionary generative adversarial networks. *Transactions on Evolutionary Computation* (2019), pp. 921-934