Using A Generative Adversarial Network (GAN) To Make Video Game Assets

Luke Tolchard
17024063

University of the West of England

August 1, 2020

Neural networks can be trained to create their own images based on a given dataset of images. This implementation of a GAN explores that idea and tries to create assets for game developers.

1 Introduction

There is a large demand for software tools that facilitate game developers to create unique assets easily without artistic skills (Fuchs, 2020). There are a lot of assets available online, but developers shy away from purchasing these as anyone could be able to acquire them which would make their final product less original. Using programming skills to create a neural network capable of learning and producing assets to be used in games would not only save production companies time and money, but it affords them a more diverse outcome. This implementation of a Generative Adversarial Network (GAN) will specifically focus on trying to create football teams that could then be used in a Unity project, using the open source library, Tensorflow (GoogleBrainTeam, 2020), and the open source neural network library, Keras (Microsoft, 2020).

2 Related Work

Fuchs, 2020, created a GAN to generate low resolution assets for video games such as potions and other items. The output files are only 16x16, but they are indistinguishable from the training data. He cites Geitgey, 2017, in his work as another example of a GAN being used to make assets, and his work is closer to the proposed implementation as this project produced 64x64 sprites of monsters.

3 Background

A GAN is a type of machine learning system conceptualised by Goodfellow et al., 2014. A GAN is made up of two components in order to produce images. The first is the generator, which learns to create fake images based on a learned dataset and feedback from its counterpart, the discriminator, which tries to distinguish real data from the data created by the generator. As the program loops, the quality of image produced by the generator slowly improves based on the accuracy value assigned to each iteration by the discriminator.

Figure 1: *Output files exported to an external game engine*

3.1 Convolutional Neural Network (CNN)

So that the discriminator can identify if an image presented to it is real or fake, it must be told what the standard is for a real image and compare the image in question with that standard. To this end, a Convolutional Neural Network (CNN) must be formed, which is a neural network designed to identify certain features in images. As described by Nielsen, 2015, they are an example of a feed-forward network which means that the output from one layer is used as the input for the next layer and so on. They are composed of an in input, an output, and hidden layers that have the task of identifying the correct output from the given input using convolution, pooling, activation functions and normalisation layers.

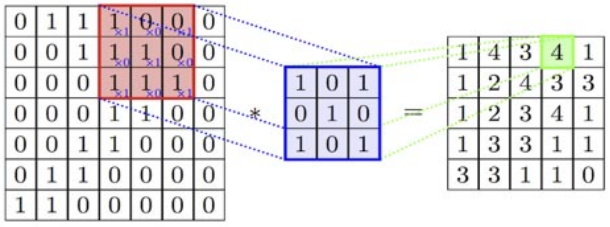

Figure 2: *Example of filters, or weights, within a CNN*

3.1.1 Convolution

A convolution is a linear operation used in traditional neural networks that involved the multiplication of a set of weights by an input (Brownlee, 2019). It is always the first layer in a CNN, and it detects low level features in an image such as edges and curves. The weights, or filters, are the sections of the input image that are currently being evaluated with them 'convolving' around the image (Figure 2). When the weights are multiplied by the input values, the resulting return is also multiplied together. An array with a dimension of 32x32 becomes 28x28 after the convolution, for example.

3.1.2 Pooling

The role of the pooling section of a CNN is to combine features detected in a convolution into a smaller representation of data to reduce the amount of parameters and computation in the network (Pokharna, 2020)

3.1.3 Activation Functions

For a singular neuron with a CNN, activation functions determine whether a value is is valid or not. There are 4 different activation functions being utilised in the implementation: ReLu (Rectified Linear Units, Nair and Hinton, 2010) (1), Leaky ReLu(2) , Tanh (3), and Sigmoid(4)

$$A(x) = max(0, x) \tag{1}$$

$$A(x) = \alpha max(0, x) \tag{2}$$

$$A(x) = \frac{1}{1 + e^{-2x}} - 1 \tag{3}$$

$$A(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

The ReLu equation when differentiated reveals a problem commonly known as the 'dead' ReLu problem (LeCun, Bengio, and Hinton, 2015):

$$ReLu'(x) = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ if } x \leq 0 \end{cases} \tag{5}$$

As shown in Figure 3 , extremely small values are not possible from a ReLu function, instead it's either a 0, causing gradients to return nothing, or 1. If too many values are below 0 when calculating gradients, a lot of weights and biases will not be updated, since the update will be equal to 0.

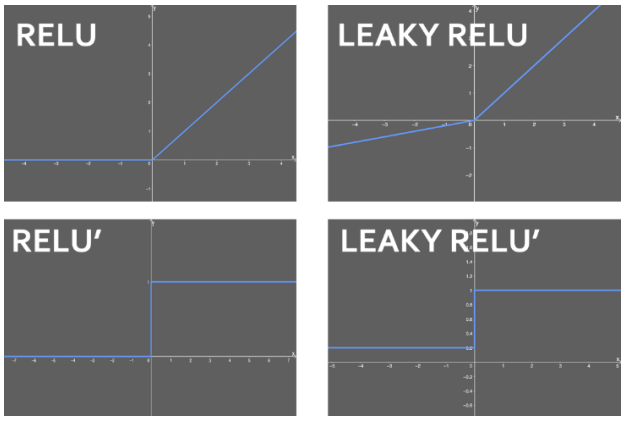

Figure 3: *ReLu and Leaky ReLu graphed with their differentials*

It has been observed by LeCun, Bengio, and Hinton, 2015, that this component of ReLu's makes a neural network perform better, due to increased sparsity. In this context, sparsity means that many activations are saturated which would lead to an increase of efficiency with regards to time and space complexity. However, if too many neurons tend towards this 'dead' state then the neural network will not recover and there will be no more optimisation or improvement of output.

A solution to this is to use Leaky ReLu(4). The differential of this activation function is:

$$ReLu'(x) = \begin{cases} x \text{ if } x > 0 \\ \alpha x \text{ if } x \leq 0 \end{cases} \tag{6}$$

The α value here usually lies within the region of 0.1 to 0.3, which prevents the gradients being stuck at 0. This prevents the 'dead' ReLu problem, at a cost of creating some others such as the exploding

gradient problem (Nielsen, 2020), which shows a rapidly increasing value of gradient. However, this is less of a problem in this implementation as the 'dead' ReLu problem, which is why Leaky ReLu will be implemented in the discriminator.

As this project contains quite a few neurons, the Sigmoid activation will only be used in one layer to help keep the network efficient and cost-effective on the systems hardware. The ReLu function can only return a value of 0 or greater, which means that if a large number of neurons return a 0 activation the network will have less to go through and will result in a faster computation.

3.1.4 Loss and Back-Propagation driven learning

A back-propagation algorithm computes the gradient of the cost function which is calculated with respect to the loss function root-mean-squared error (RMSE) and through evaluating the result of the current neuron and previous results (LeCun, Bengio, and Hinton, 2015). The loss function is the square root of the evaluated node, and this value is returned to adjust the weight for the next iteration.

4 Implementation

A dataset containing 24,000 football player and staff profiles used in a game series (Manager, 2020) was downloaded to be used as training data. The original versions of the images used in this training data had varying sizes and an alpha channel included in their bit depth, so a batch image resizer was used to set every image to the same size and remove the alpha so that the training data array could remain 2 dimensional.

```
GENERATE_RES = 2 # (1=32, 2=64, 3=96,)
GENERATE_SQUARE = 32 * GENERATE_RES
```

The generator uses $model.add(Conv2D())$ (upsampling) layers to produce an image from a seed (random noise). The first layer contains a dense layer that specifies the input shape, which then takes this seed as input, then upsamples several times until the desired image size is generated. The generator was built with an automatically expanding amount of layers depending on the target output resolution set. This allowed it to complete additional upsampling to cope with an increased amount of neurons.

```
for i in range(GENERATE_RES):
model.add(UpSampling2D())
model.add(Conv2D(128,kernel_size=3))
model.add(BatchNormalization(mom=0.8))
model.add(Activation("relu"))
```

All layers in the generator have a kernel size of 3x3, and use the ReLu activation function apart from the final layer which uses tanh.

The discriminator contains 6 layers, the first 5 use the Leaky ReLu activation function with an α value of 0.2, and the last uses Sigmoid. It also contains kernel sizes of 3x3, with the first 3 layers containing 2 strides, and the last 3 layers only 1 stride. The discriminator returned a model that took in the original input and a validity score for the last test.

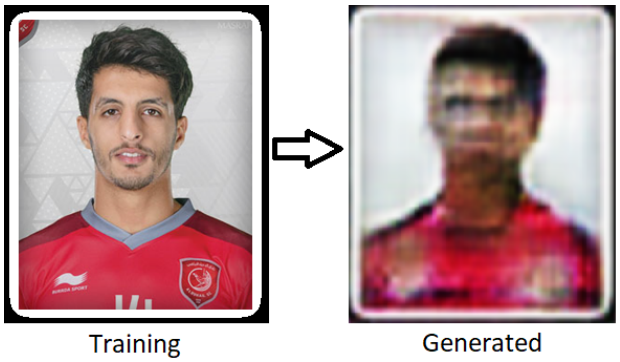

Figure 4: *An example of a generated image next to a sample from the training data. (3000 epochs)*

4.1 Output to a Game Engine

For each epoch, a sheet of faces on the same output were created in the dimension of 7x4 (Figure 5). This was mainly so that the generator could have multiple attempts using the same seed, and reduce the chance that an epoch would produce entirely poor results. Once the network had generated 3000 epochs, the last few generations were chosen and cropped to be the basis for the football teams. Figures 1 and 7 shows the chosen images loaded into Unity as textures on plane objects. Specular reflections were disabled on each texture to prevent the engines sunlight from washing out the image, and a demonstration stadium was quickly constructed to show the textures being used as potential 2D representations of players in a game scenario.

Figure 5: *Generation 299/300 from the neural network*

Some of the faces lack detail to make out certain features, but the potential for the quality of the returned image is only really limited by the hardware capabilities of the machine being used to render, and the time willing to be spent.

5 Evaluation

At the end of 3000 epochs, the final loss was approximately 0.09. It is suspected that the network suffered from mode collapse as a result of the data set used. The majority of players in the data set wear red and blue kits, as they are the most common colours for football teams to sport. This perhaps meant that the generator had learned less about how to make players wearing other colours such as yellow and black, and they appeared much more distorted and lossy (Figure 6, left). Whilst the sample size was broad, the images contained a wide variety of backgrounds behind the players, and they were also subject to a large variety in lighting levels, which sometimes led the backgrounds to 'bleed' into the skin tones of the players (Figure 6, right). An improvement to the consistency of the data set, or an improvement to the generator layers might have solved this issue.

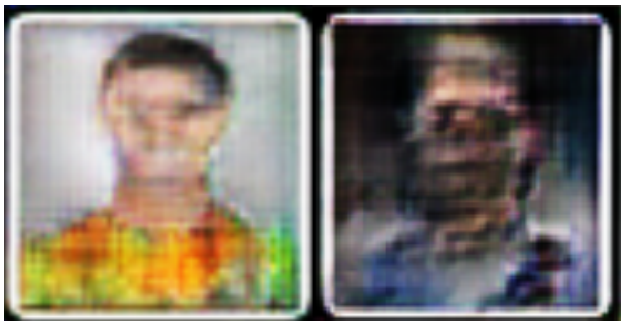

Figure 6: *Possible mode collapse and background bleed*

6 Conclusion

This report outlines the creation of a GAN to create low-resolution sprite textures fit for video game implementation, using a convolutional neural network in Python. The method used to train the neural network was discussed, in relation to other GANs completed. The resultant image data was exported to a game engine and improvements have been suggested to yield better results for future implementations.

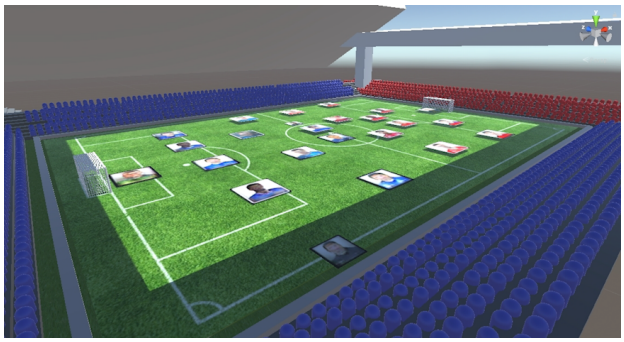

Figure 7: *Generated images representing football players in a simple stadium*

Bibliography

Brownlee, Jason (2019). *A Gentle Introduction to Object Recognition With Deep Learning*. URL: https://machinelearningmastery.com/object-recognition-with-deep-learning/.

Fuchs, Fionn (2020). URL: https://medium.com/@fionn.fuchs/game-asset-gan-using-unsupervised-machine-learning-to-generate-game-assets-d1d24af9eb0d.

Geitgey, Adam (2017). *Building and Deploying Deep Learning Applications with TensorFlow*.

Goodfellow, Ian et al. (2014). "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., pp. 2672–2680. URL: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

GoogleBrainTeam (2020). URL: https://www.tensorflow.org/.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (May 2015). "The multidisciplinary nature of machine intelligence". In: pp. 436–444.

Manager, Football (2020). URL: https://fminside.net/football-manager-downloads/football-manager-2019-downloads/fm19-facepacks/df11-2019-facepack/.

Microsoft (2020). URL: https://keras.io.

Nair, Vinod and Geoffrey Hinton (June 2010). "Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair". In: vol. 27, pp. 807–814.

Nielsen, Michael A (2015). *Neural networks and deep learning*. Vol. 2018. Determination press San Francisco, CA.

Nielsen, Michael A. (2020). *Neural Networks and Deep Learning*. URL: http://neuralnetworksanddeeplearning.com/chap5.html.

Pokharna, Harsh (2020). URL: https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8.