Using Neural Networks to Generate Artistic Images

Owen Mooney

ABSTRACT

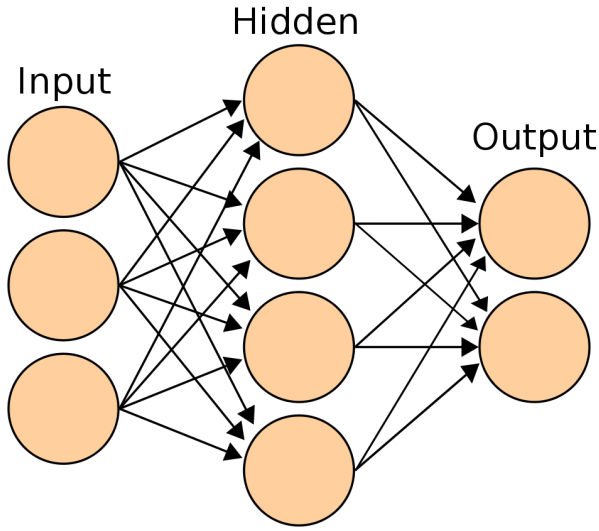
Artificial Neural Networks are computational tools that have found widespread use in a number of varied fields in computer science. In particular they have a wide range of uses in computer graphics, for example in image classification [7], face detection [5] and recognition [8], and image restoration [3]. Neural Networks are particularly efficient for these tasks since they are well suited for recognising and classifying patterns in input data. This paper investigates how it might be possible to generate aesthetically pleasing images using a Neural Network with random input data. The Neural Network could be trained by using either a fitness function or through human interaction, and thus could learn how to construct an image that would be visually appealing.

1. INTRODUCTION

An Artificial Neural Network is a system of independent processing units arranged in layers, with the units in each layer having connections to every other unit in the layers above and below it. Each unit has multiple inputs and outputs, with the outputs from the previous layer being the inputs to the next, thus the structure of the network forms a directed acyclic graph. This structure is known as a *Feed Forward Neural Network*. It is also possible to create a *Recurrent Neural Network*, i.e. one where the graph will contain cycles, however these types of networks will not be discussed in this paper. The structure of a basic Neural Network is shown in figure 1. This network has an input and output layer and one *hidden* layer. The number of hidden layers can be varied depending on the specific problem the network aims to solve.

In designing a Neural Network, there are a number of choices to be made. The first is the choice of combination function. This function determines how the inputs for each unit are combined to produce a *net input* for the unit. Most Neural Networks use either a linear combination of the inputs, or a Euclidean distance function. Feed Forward Neural Net-

Figure 1: A graphical representation of the structure of a Neural Network



works are broken down into two main types depending on the combination function they use. These are the *Multilayer Perceptron (MLP)*, in the case of the linear combination function, and the *Radial Basis Function (RBF)*, in the case of the Euclidean distance function. In this paper, only MLP networks will be discussed. The net input, ν_k, for each unit is calculated as follows

$$\nu_k = \sum_{j=1}^{n} w_{kj} x_j \tag{1}$$

where w_{kj} is the weight associated with the connection from unit j in the previous layer, x_j is the output from the j-th unit in the previous layer, and n is the total number of units in the previous layer.

The second design choice that must be made is the activation function. This function determines how the net input, as calculated above, is converted into the output value for the unit. In many Neural Network implementations, including the one discussed in this paper, before the net input is

transformed by the activation function, it is reduced by an amount know as the *threshold* for the unit. It will be shown in *Section 3* that this value can be altered as part of the learning process for the network. Denoting the threshold for the k-th unit as τ_k, the output for that unit, σ_k will be given by

$$\sigma_k = \phi(\nu_k - \tau_k) \tag{2}$$

where ϕ is the activation function. The activation function should be one that exhibits non-linearity, since a multi-layered network with a linear activation function can be reduced to a single layer network, thus losing the full computational power of a MLP [6]. Two common activation functions are the hyperbolic tangent function, and the logistic function, which is defined as

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

A further design choice to be considered is how the network will learn. Typically the network will learn by adjusting the weights of the connections and the threshold values for each unit. The changes that need to be made to the network can be computed by the difference between the output of the network given a certain input, and the desired output. *Back Propagation* is the technique that is typically used to achieve this [2].The details of its operation are beyond the scope of this paper, since it is not the learning technique used in this project. Other possibilities for learning algorithms are using genetic or evolutionary techniques, these will be discussed further in the following section.

2. RELATED WORK

There are a number of existing methods of image generation and synthesis that rely on the use of Neural Networks. Many of these techniques use the properties of Neural Networks to create textures that model the surface appearance of real-world materials.

2.1 Textured Image Synthesis

The method outlined in the paper *Textured Image Synthesis and Segmentation via Neural Network Probabilistic Modelling*[4], uses Neural Networks to categorise different features in a texture. The network can then be used to generate textures that resemble that texture.

2.2 Random Neural Network Texture Model

This paper [1], also outlines a method for generating textures using a Neural Network. It uses a similar technique to the one outlined in [4]. The network is trained using an image of the desired texture, then the network is used to generate textures that resemble that image. The resulting images share some of the features of the original texture, however the algorithm can generate random variations.

The two papers discussed here are both concerned more with pattern recognition, and then using the knowledge of these patterns stored in the network to synthesise images, than simply generating images from scratch.

3. METHOD OUTLINE

As explained above, the Neural Network used for this project is a feed-forward multi-layer perceptron, using simple linear combination as the combination function. The activation function used is the logistic function defined in equation (3). The inputs to the network will be a set of real numbers between zero and one and will be randomly generated. The random input data should act as a 'seed' for the image generation. The idea behind this is that the network should be able to generate different images for different input, but that each image generated should fulfil the criteria for being an aesthetically pleasing image. Thus the network will be a general algorithm for generating visually appealing images. The possible uses of such an algorithm are discussed in *Section 7*.

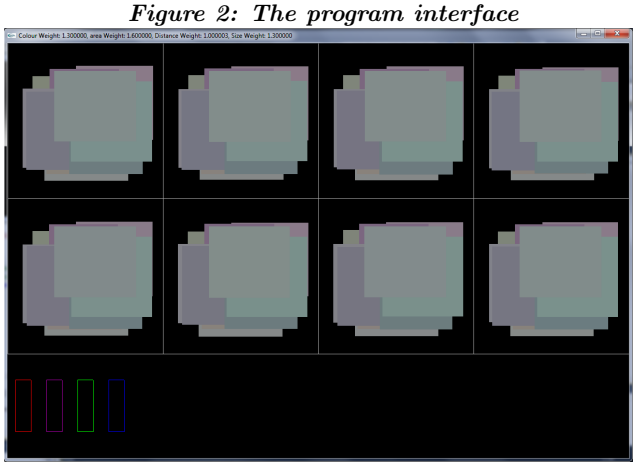
In a network of this type, the learning process consists of altering the connection weights and threshold values of each unit, in such a way that the output performs better as measured by some criteria. Hence these values encode the 'knowledge' of the network. Using back propagation as a training method requires that the desired output be exactly known from a given input. In this case the desired output, (that of a visually appealing image), is too poorly defined to use this technique. Thus an evolutionary algorithm is used to train the network. Every iteration of the application produces a number of distinct images, which are computed by a corresponding number of different variations of the network. The best image (as defined by the user or a fitness function) is chosen, and that network is used as the base for the next iteration. The learning rate of the network can be changed by altering the number of variations to the network's structure generated in each iteration.

The outputs from the network are a set of real numbers between one and zero. Each image is generated using a number of coloured primitives, whose dimensions, position and colour are defined by the outputs from the network. The best image can be chosen by the user, or by calculating some metric for each image. Possible metrics to be used to determine what constitutes a good image are: colour distribution, primitive size, coverage of image space and distance between primitives.

4. IMPLEMENTATION

The application accompanying this paper is written in C++, and uses OpenGL and GLUT for rendering. A screenshot of the program interface is depicted in figure 2. The network used has eight inputs, four hidden layers that consist of eighty units, and eighty outputs. The image is generated from a number of coloured rectangles. Each of these takes eight parameters: x and y coordinates, height, width, and an RGBA colour value. In the current implementation the alpha value has no effect on the rendering. Since there are eighty outputs from the network, each image consists of ten rectangles. As described before, each iteration of the program produces eight different variations of the network, and an image for each one of these is generated. To reduce the

Figure 2: The program interface



memory cost of storing eight different networks, only one network is stored in full in memory. For each variation, a number of different changes to the weights and thresholds are generated. After these changes are applied, they can be reverted. This allows one network to generate eight distinct images using eight different variations. Once the best variation is chosen, the changes that produced that variation are applied permanently to the network and the cycle begins again. The structure of the network can be written to and read from disk, allowing its state to persist after the program exits.

The program can operate in two distinct ways. In one, the user can select the most pleasing image out of the eight using the mouse, and thus pick the best variation of the network. The other mode allows a fitness function to be evaluated for each image, and the best image chosen by the application itself. When using the fitness function, there are four parameters that affect how the different criteria are weighted, one for each of the different criteria by which the images are judged. The criteria are the colour brightness, the average distance between the rectangles, the fraction of the total image space that are covered by rectangles, and the difference between the average area of the rectangles and a target area value. These values can be changed at runtime.

Each unit in the network is an instance of a **Perceptron** object. These objects expose the following functionality:

```
class Perceptron
{
public:
    virtual void Increment(float amount);
    virtual float GetValue();
    virtual void Reset();
    float threshold;
protected:
    float currentValue;
};
```

The **Increment** function applies the inputs to the unit, and the current net input value is stored in the field **currentValue**. The function **GetValue** applies the activation function to the net input and returns the result. The **Reset** function simply resets the net input counter to zero. The input and output units are subclasses of this and are named **InputNode** and **OutputNode** respectively, as shown below:

```
class InputNode : public Perceptron
{
public:
    void SetValue(float amount);
};
```

```
class OutputNode : public Perceptron
{
public:
    float GetValue();
};
```

The **SetValue** function allows the value of the **InputNode** to be set directly, and the **GetValue** function allows the normalised output from each **OutputNode** to be read.

Each connection in the network is an instance of a **Connection** object:

```
class Connection
{
public:
    Connection(int index, float weight,
        Perceptron* endPoint, int outIndex);
    void Activate(float amount);
    int inputIndex;
    float weight;
    int outputIndex;
private:
    Perceptron* connected;
};
```

The constructor takes four parameters, the array index of the input unit, the weight of the connection, a pointer to the **Perceptron** object that it is connected to, and that units array index. The **Activate** function takes the output value from the input unit and calls the **Increment** function on the unit it is connected to with that value as its argument. The **weight** field simply stores the connection weight. The **inputIndex** and **outputIndex** fields are necessary for serialisation of the class when the network is saved to a file.

Each layer of the network is implemented as an array of **Perceptron** objects, and an array of **Connection** objects. To compute the output of the network, first the input layer is initialised with the input data using the **SetValue** function as outlined above. Then each successive layer first computes the outputs from each unit, and the **Activate** method is

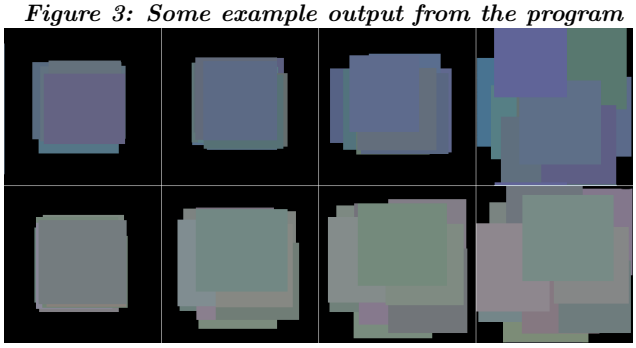
called on each **Connection** object with the output from each unit as its argument. Finally the outputs are collected from the **OutputNode** objects. These outputs are used to create rectangles that are drawn to the screen.

5. RESULTS

Some output from the program is shown in figure 3. The image shows two separate executions of the program. For each execution, four images were generated at different points through the learning process. The learning process used in both cases was through the use of the fitness function outlined in *Section 4*. In the second run (the bottom set of images) the variations were more aggressively selected for images with a large average distance between the rectangles. Thus the rectangles spread out more quickly. The number of iterations between the first and last images is approximately one thousand. Thus it would be infeasible for a human user to supply the information required over such a huge number of generations. Altering the learning rate could ameliorate this problem, however increasing the learning rate to a much larger value can cause instability.

The images generated by the network bear little relation to the input. This means that the neural network is really only learning to generate one particular image. To remedy this, it would be necessary to teach the network to select variations that produce different output depending on the input. This would result in a further increase in the complexity of the program. As it stands, the network must compute results eight times for each iteration, using the same input. To select variations that produce different outputs for each input, each variation would need to produce a set of images for a set of inputs, and these would have to be compared to ensure they were sufficiently different.

Figure 3: Some example output from the program



6. FURTHER ENHANCEMENTS

As mentioned in *Section 5*, the network could be trained to produce varying images dependent on the input data. This would allow it to generate many images, that each have some sort of artistic value. The training time for such a network could be quite large, as it would increase the computation time for each successive generation of the network.

The basic building blocks of the images generated so far are coloured rectangles. It is possible to use many other primitive shapes to build up an image. The network could also be made to output information that determines what type of primitive is drawn, as well as its size, position and colour. Other possible primitives include triangles, circles/ellipses, lines, or arbitrary polygons. Also, each primitive could be drawn with more detail. Instead of simply being a solid colour, they could include gradients, borders etc. Furthermore, alpha blending could be used to allow the colours of each primitive to interact and produce more complex effects.

To generate more complex images, drawing more primitives is necessary. Since a simple rectangle requires at least seven parameters to be drawn (more complex primitives could require more), then the required number of outputs of the network could increase dramatically. This could severely impact computation time. One solution would be to use a different input to generate a further set of primitives and combine the output of multiple computations of the network until a satisfactory image is obtained.

As an alternative to coloured primitives, the network could be used to generate spatial and colour-space distortions in the image. Artistic filters could also be controlled using the network, which could increase the complexity and also the visual appeal of the image.

7. POSSIBLE APPLICATIONS

The procedural generation of images with aesthetic appeal could be useful in many areas of computer graphics. Video games in particular could benefit greatly from an efficient process of generating such images. The cost of content creation for modern video games has increased dramatically in the last decade. The use of procedurally generated artwork could help small developers to create unique and visually appealing video games, in a cost-effective and efficient way.

Images generated in a manner like the one outlined in this paper, are by their nature, quite abstract. These types of images could be particularly suited to graphic design, and in particular web design.

Apart from the possible uses in industry, the generation of images by computers is a valid artistic pursuit. Since computer graphics has existed there have been many people who have used computers to generate images of artistic merit. This paper attempts to use the power of Neural Networks to generate such images. The technique as presented here does not generate particularly complex images, and their aesthetic value is debatable, however a refinement of the technique using some or all of the enhancements outlined in the previous section could generate much more impressive results.

8. REFERENCES

[1] V. Atalay, E. Gelenbe, and N. Yalabik. The random neural network model for texture generation. *IJPRAI*.

[2] D. Bertsekas. *Nonlinear programming*. Optimization and neural computation series. Athena Scientific, 1995.

[3] A. P. A. de Castro and J. D. S. o. da Silva. Restoring images with a multiscale neural network based

technique. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1693–1697, New York, NY, USA, 2008. ACM.

[4] J.-N. Hwang and E. T. yen Chen. Textured image synthesis and segmentation via neural network probabilistic modeling.

[5] B.-H. Lee, K.-H. Kim, Y. Won, and J. Nam. Efficient and automatic faces detection based on skin-tone and neural network model. In *Proceedings of the 15th international conference on Industrial and engineering applications of artificial intelligence and expert systems: developments in applied artificial intelligence*, IEA/AIE '02, pages 57–66, London, UK, UK, 2002. Springer-Verlag.

[6] E. Shenouda. A quantitative comparison of different mlp activation functions in classification. In J. Wang, Z. Yi, J. Zurada, B.-L. Lu, and H. Yin, editors, *Advances in Neural Networks - ISNN 2006*, volume 3971 of *Lecture Notes in Computer Science*, pages 849–857. Springer Berlin / Heidelberg, 2006.

[7] H. Zhang, J. Guan, and G. C. Sun. Artificial neural network-based image pattern recognition. In *Proceedings of the 30th annual Southeast regional conference*, ACM-SE 30, pages 437–441, New York, NY, USA, 1992. ACM.

[8] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Comput. Surv.*, 35:399–458, December 2003.