Music Generation with Deep Learning

Clement Jin

April 2024

Abstract

Deep learning is highly effective at recognising patterns in data, to the extent that it can exhibit signs of "creativity". Similarly, music composition involves the application of intricate patterns in creative ways. Given the widespread availability of digital music datasets, music generation is a highly pertinent use case for deep learning research and application. Automatic music generation would make bespoke compositions readily accessible to the public, whilst serving as a creative tool for composers. In this project, I explore deep neural networks in mathematical detail, before implementing one to extend melodies from JS Bach's Chorales. I found that a simple neural network alone was insufficient to extend melodies musically, and that more sophisticated models are necessary. I then describe one such model: the recurrent neural network, and how its features make it a promising area for future breakthroughs in artificial music generation.

Contents

1 Introduction

The astronomical development of deep learning has redefined many diverse fields of work [Wang and Siau, 2019], and has been facilitated by the exponential rise in computing power [Moore, 2021] as well as the availability of big data [Vassakis et al., 2018]. Deep learning algorithms are able to process vast amounts of data to recognise patterns so subtle that their abilities begin to resemble that of a human. Recent developments in Generative AI models, such as GPT-4 [OpenAI et al., 2024], have seen machines excel in creative fields such as image and text generation, indicating a shift away from machines as tools consigned to menial tasks, and towards critical and inventive agents.

Composing music well requires detailed knowledge of rules, including those about rhythm, harmony and structure, as well as creativity to produce something original. As a result, composition has been restricted to the few who possess this specialised skill set. Artificial music generation could democratize composition to the extent that one only requires access to internet and a computer, making custom music widely available. It could also serve as inspiration for composers and songwriters, professional or amateur, as a tool to enhance their own creative process.

Indeed, deep learning is a suitable way to go about automatic music generation. The rules of music theory are complex, yet largely consistent, making them learnable by a machine. Moreover, the wide digitization of music into many accessible forms provide a plethora of training data from a variety of different genres. The success of deep learning in music generation would further our understanding of this technology, whilst demonstrating its immense versatility and potential.

In this project, I first describe deep neural networks in detail, including their design and how they are trained via gradient descent and backpropagation (Chapters 2-5). Next, I implement a deep neural network which aims to extend a given melody from the Bach Chorales Dataset [Conklin, 2016], and analyse the results after training. I explore the limitations of using such a network and dataset (Chapter 6), before introducing the recurrent neural network: a refinement of the deep neural network with greater sophistication to the task at hand (Chapter 7).

2 Deep Neural Networks

2.1 Neurons

In order to create a predictive model, it will be necessary to understand the core of machine learning: the deep neural network. A deep neural network consists of many layers of nodes, called neurons. Each neuron takes in a series of numerical inputs $x_1, x_2 \cdots x_n$ and produces a single numerical output [Nielsen, 2015] (Figure 1).

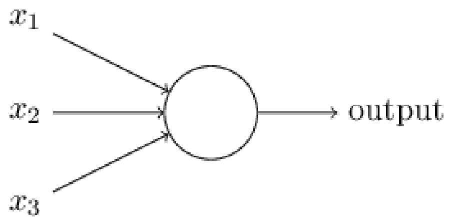


Figure 1: A single neuron with 3 inputs. Credit: *[Nielsen, 2015], Chapter 1*

Neurons in a network are arranged in vertically stacked layers (Figure 2), and can be connected together so that the output of one neuron becomes the input of the next. In a fully connected network, each neuron in layer l is connected to all the neurons in the previous layer $l-1$. That is to say, each neuron takes as input all of the outputs of the previous layer. The only exception to this is the first layer, where the values of the neurons are the inputs to the network. Hence, it is called the *input layer*, while the final layer which produces the output of the network is called the *output layer.* Any layers in-between are called *hidden layers.* In application, if we wanted a network to extend a given melody, the input layer may contain the note values for the start of a melody, while the output layer may contain the next notes of the melody, hence extending it.

Referring to Figure 2, the values in the input layer i are fed as input to the neurons in hidden layer h_1. In turn, the values of neurons in h_1 are input to the next hidden layer h_2, and so on, until an output is produced in layer o. This process of giving the network an input, and allowing the input values to feed into subsequent layers of neurons is called the *feed forward* step [Nielsen, 2015], and it is how the network will produce an output.

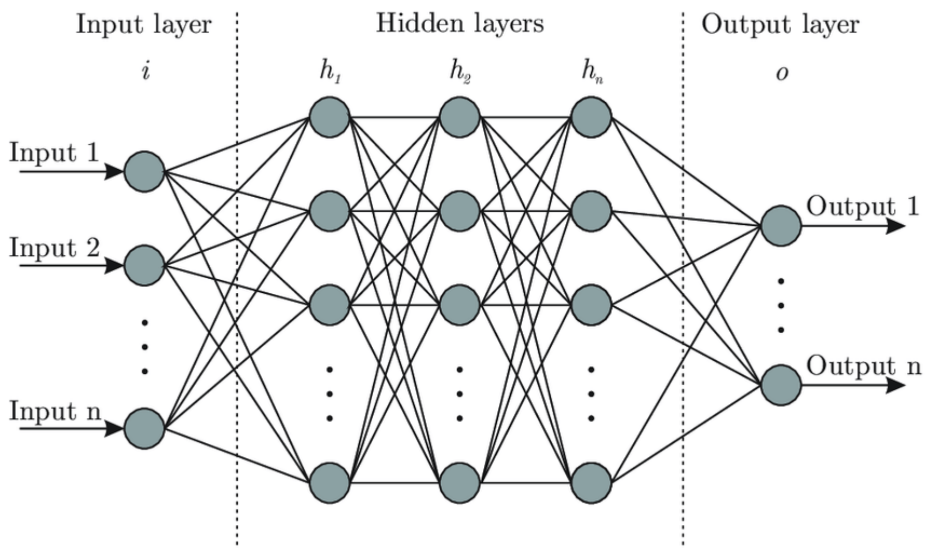


Figure 2: A neural network consists of layers of connected neurons. Credit: *L. Shukla, Towards Data Science*

2.2 Feed Forwards

Since some inputs will naturally be more important than others, they should have a larger effect on the final output [Nielsen, 2015]. We can assign such an importance with weights $w_1, w_2 \cdots w_n$, one for each input. The higher the weight value, the greater the impact on the output. We can start to define what is called the *weighted input*, z, of a neuron which determines the neuron's output. Let us first define z' as the sum of the neuron's inputs multiplied by their respective weights.

$$z' = \sum_{i=1}^{n} x_i \cdot w_i \tag{1}$$

We can concisely write this operation by representing the inputs and weights of the neuron as vectors, $\mathbf{x} = [x_1, x_2, x_3, \cdots x_n]$ and $\mathbf{w} = [w_1, w_2, w_3 \cdots w_n]$ respectively, and then performing a dot product[1]. Hence, an equivalent representation of z' is

$$z' = \mathbf{x} \cdot \mathbf{w} \tag{2}$$

In fact, the *weighted input* z of the neuron has one extra component: the bias, b. Each neuron has a single bias value, and is added to z' to create more

[1]The dot product takes the element-wise product of the two vectors and sums them, which is exactly the operation we want.

flexibility in the network[2]. Thus we define the *weighted input* z as

$$z = (\mathbf{x} \cdot \mathbf{w}) + b \tag{3}$$

In each feed forward pass of the network, the z values of each neuron is calculated using the neurons in the previous layer as inputs. Furthermore, we can represent the weighted inputs of an entire layer of neurons using vector-matrix multiplication

$$\mathbf{z} = (\mathbf{x}\mathbf{W}) + \mathbf{b} \tag{4}$$

Here, $\mathbf{z}$ is a column vector representing the weighted inputs of one layer of neurons; $\mathbf{W}$ is the weight matrix whose rows are the weight vectors $\mathbf{w}_i$ for each neuron in the layer, and $\mathbf{b}$ is a column vector containing the bias values for each neuron in same layer.

2.3 Activation Function

If we used the weighted inputs of the previous layer to compute the weighted inputs of the current layer, no combination of neurons would produce any meaningful output. This is because the weighted input is a *linear* function of the form $\mathbf{w} \cdot \mathbf{x} + b$. The issue with this is that any composition of linear functions, e.g. by connecting many neurons together, will always result in another linear function with different coefficients, $\mathbf{w}' \cdot \mathbf{x} + b'$. Hence, the entire network could be replaced by just a single linear neuron which could only return linear outputs [Goyal et al., 2020], i.e., the output is directly proportional to the input[3]. For even relatively simple tasks, this is not the case. There are often complex relationships between the inputs and the outputs which cannot be predicted using linear functions. Hence, some non-linearity must be introduced to de-trivialize the structure of a neural network. That is the work of an activation function.

The output of a neuron is called its *activation*, a. To compute this, we take the weighted input z and pass it through a function σ, called an *activation function*, which provides the crucial non-linearity we desire [Nielsen, 2015]. The activation of a neuron is thus defined as

$$a = \sigma(z) = \sigma(\mathbf{x} \cdot \mathbf{w} + b) \tag{5}$$

[2]This is expounded in Figure 3.

[3]Since the network has many inputs and outputs, we would say that the partial derivative of any single output A with respect to any single input X, $\dfrac{\partial A}{\partial X}$ is equal to a constant. We can see this easily, since $A = cX + c_1x_1 + c_2x_2 + \cdots + c_nx_n + b$, and since everything but X is a constant, we have $\dfrac{\partial A}{\partial X} = c$. (See Section 3.2 for more information)

This can be generalised to computing the activation of an entire layer of neurons, with $\mathbf{a} = \sigma(\mathbf{z})^4$.

Several different activation functions can be used for different networks. One of the most common functions is the sigmoid function, defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

Moreover, it is now possible to see how the bias adds "flexibility" to each neuron, since varying this bias translates the activation function to the left or right. This is illustrated in Figure 3.

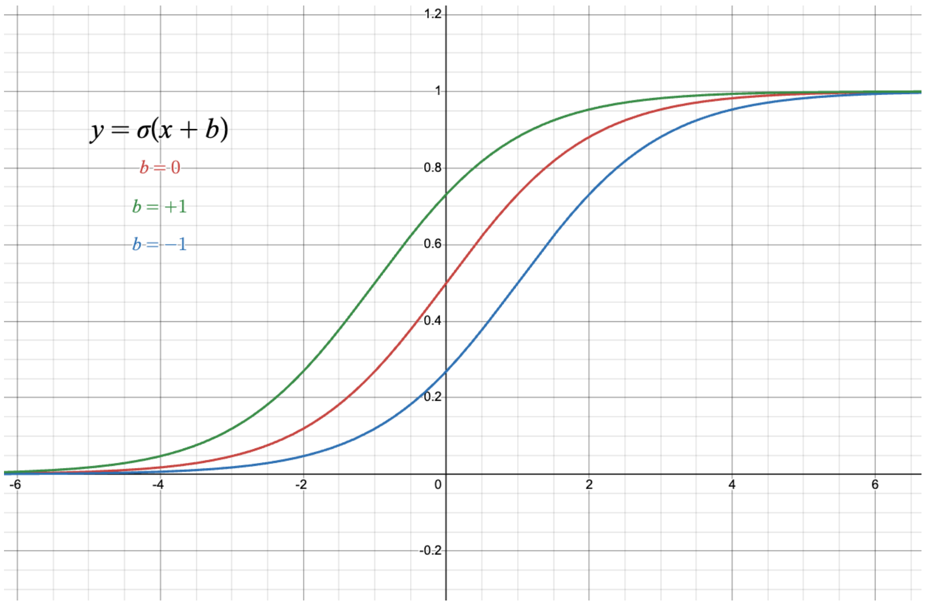


Figure 3: The Sigmoid Function is translated horizontally with different bias values $(0, +1, -1)$. Credit: *Desmos*.

[4]We understand that applying a function to a vector means applying the function to every element within that vector.

2.4 Training the Network

Initially, all the weights and biases are set to arbitrary values, and hence the network's outputs will be random. The aim is to choose a set of weights and biases such that, for any meaningful input, the network can generate some meaningful output. This can be done by incrementally changing the values of each weight and bias such that the network returns better outputs after the change [Nielsen, 2015] [Bishop and Bishop, 2024]. Working out the exact changes to the weights, $\Delta \mathbf{W}$ and biases $\Delta \mathbf{b}$ will cause the network to learn.

2.5 Notation

Due to the large number of variables in the network, it is common to implement a systematic naming convention [Nielsen, 2015], illustrated in Figure 4.

- z_j^l is the *weighted input* of the j^{th} neuron in the l^{th} layer.
- a_j^l is the *activation* of the j^{th} neuron in the l^{th} layer.
- b_j^l is the *bias* of the j^{th} neuron in the l^{th} layer.
- $w_{k,j}^l$ is the *weight* that connects the k^{th} neuron in the l^{th} layer to the j^{th} neuron in the $l-1^{\text{th}}$ layer[5].
- L denotes the output layer. E.g. a_j^L is the j^{th} neuron in the final layer.

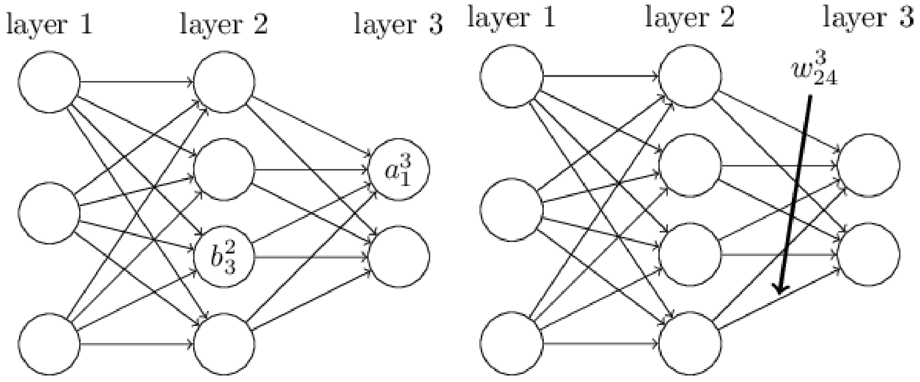


Figure 4: Illustrations of index notation. Credit: [Nielsen, 2015] *Chapter 2*.

[5]Note that the notation here is backwards, i.e. it is more natural to label weights going from layer l to $l+1$. However, specifying weights this way makes the calculation much more elegant when the weight matrix is multiplied by the input vector.

3 Learning

3.1 The Cost Function

To quantify the error in the network, we use a cost function C. There are many possible cost functions, including *"quadratic cost"* , defined as

$$C = \frac{1}{2}\sum_{k=1}^{N}(y_k - a_k^L)^2 \tag{7}$$

Where a_k^L represents each of the network's outputs, and y_k represents each of the *target outputs*, i.e., the output we want the network to return[6]. Now, we can formalise the idea of improving the network as minimizing the cost for all meaningful inputs to the network[7] [Ray, 2019].

3.2 Partial Derivatives

For a given input, we can decrease the value of the cost function by varying the parameters, i.e., the weights and biases. The idea is to work out how to change these parameters in the right way so it minimizes cost [Bishop and Bishop, 2024]. In fact, partial differentiation gives us exactly the right tools to do this.

An ordinary derivative is used to find the gradient of a single variabled function, $f(x)$. It describes how quickly the function $f(x)$ changes with respect to the variable x.

$$f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \tag{8}$$

The partial derivative deals with functions of many parameters such as $g(x, y)$. Here the function $g(x, y)$ defines a surface over a two-dimensional plane. At any point (x, y), the slope is different depending on which direction we approach (x, y) from. Hence, multi-variabled functions have many gradients that are described using *partial derivatives.* For example, the function $g(x, y)$ has 2 parameters[8], and hence has 2 partial derivatives, $\dfrac{\partial g}{\partial x}$ and $\dfrac{\partial g}{\partial y}$, which describe the slope as approached from the x and y directions respectively. This can be formalised as the following [Strang, 1991].

[6]The target output can be thought of as the "correct answer" to a given input question.
[7]C is halved because it gives a nicer expression when it is differentiated. See 4.2.1
[8]Note that x and y are independent.

$$
\begin{aligned}
g_x(x,y) &= \frac{\partial g}{\partial x}(x,y) = \lim_{h\to 0}\frac{g(x+h,y)-g(x,y)}{h}\\
g_y(x,y) &= \frac{\partial g}{\partial y}(x,y) = \lim_{h\to 0}\frac{g(x,y+h)-g(x,y)}{h}
\end{aligned}
\tag{9}
$$

In essence, we fix all variables except one, and calculate the gradient of $g(x, y)$ as if it were a single valued function. By fixing the other variables, we effectively take a 2-dimensional "slice" of the function, for which we can apply the same rules as single-variabled calculus. This is illustrated in Figure 5.

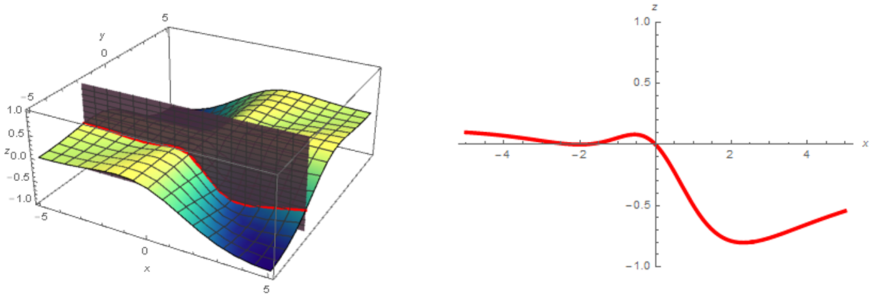


Figure 5: Taking the gradient of a multi-valued function means taking the ordinary derivative of a "slice" of the function. Credit: *Brilliant Wiki: Partial Derivatives.*

To compute the partial derivative in one direction, we simply treat the other variables as constants and differentiate as normal. For example, if $g(x, y) = x^2 + 2xy + 3y^2$, we can find the partial derivatives accordingly.

$$\frac{\partial g}{\partial x} = \frac{\partial}{\partial x}(x^2+2xy+3y^2) = \frac{\partial}{\partial x}(x^2+2xc_1+c_2)$$

Where $c_1 = y$ and $c_2 = 3y^2$ are both constants. Hence, by the power rule,

$$\frac{\partial g}{\partial x} = 2x + 2c_1 = 2x + 2y$$

Moreover, this can be generalised to functions of any number of inputs [Weisstein, 2024]. Let C be a function of n inputs $C(a_1, a_2, \cdots a_n)$. Then we have

$$\frac{\partial C}{\partial a_k} = \lim_{h\to 0}\frac{C(a_1,a_2\cdots a_k+h,a_{k+1},\cdots a_n)-C(a_1,a_2,\cdots a_n)}{h} \tag{10}$$

3.3 Gradient Descent

Now, with the tools of multivariable calculus, we can find a method to minimize the cost function $C(\mathbf{v})$ by tweaking its parameters $\mathbf{v} = [v_1, v_2 \cdots v_n]$, i.e., all of the weights and biases.

Imagine a simple cost function with just two parameters v_1 and v_2 (Figure 6). The function can be visualised as a surface whose height is the value of the cost function. Now imagine a ball at the point (u_1, u_2). We want to move the ball in the "downhill" direction to minimize cost. This means adding small values Δu_1 and Δu_2 to u_1 and u_2 respectively, such that $C(u_1+\Delta u_1, u_2+\Delta u_2) < C(u_1, u_2)$. The changes $\Delta u_1, \Delta u_2$ can be thought of as the "direction" to travel in order to minimize cost [Bishop and Bishop, 2024] [Nielsen, 2015].

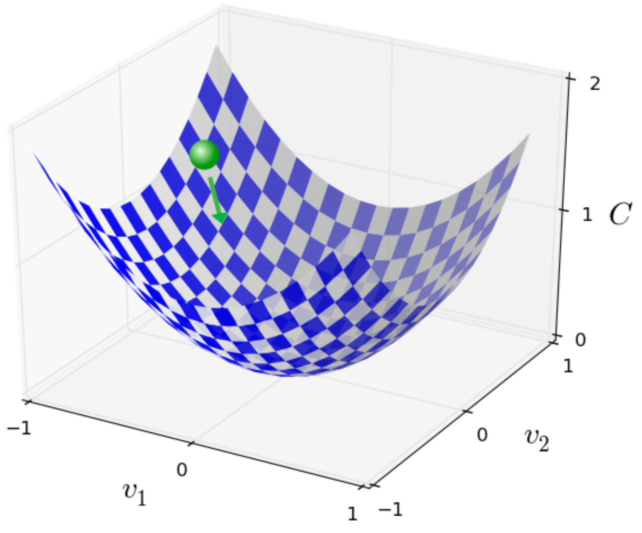


Figure 6: Gradient descent can be thought of as making a ball roll down a hill into a valley. Credit: [Nielsen, 2015] *Chapter 2.*

In general, a neural network has tens of thousands of such parameters that we cannot visualise in 3 dimensions, yet the principle remains the same. Let us now see how we can quantify the change to calculate this "downhill" direction using gradient descent using linear approximations.

3.4 Linear Approximations

In single-variable calculus, we can approximate the change of function $f(x)$ using its derivative $\frac{\mathrm{d}f}{\mathrm{d}x}$. Let us plot $f(x)$ against x, and consider the point $(a, f(a))$. Around this point, the function behaves like a straight line $L(x)$ with gradient $\frac{\mathrm{d}f}{\mathrm{d}x}$, so after making a small change to the x coordinate, Δa, the corresponding change to $f(a)$, Δf can be approximated as

$$\Delta f \approx \frac{\mathrm{d}f}{\mathrm{d}x}\Delta a \tag{11}$$

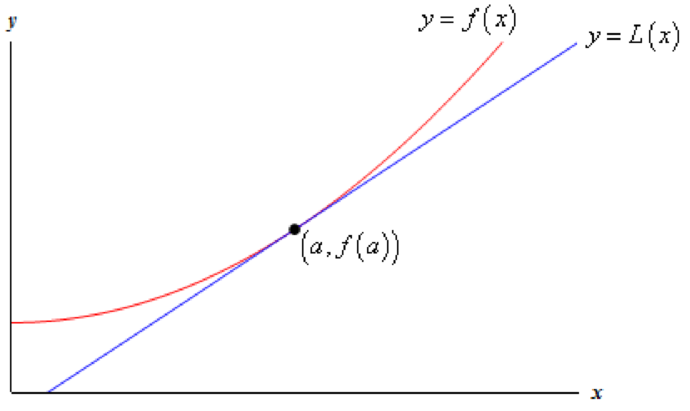


Figure 7: A Linear Approximation of a function.

The same principle can be generalised to multivariable functions. As before, let $C(\mathbf{v})$ be a function of $\mathbf{v} = [v_1, v_2, \cdots v_n]$. For a point $\mathbf{v}$ on C, we can make a small change $\Delta\mathbf{v} = [\Delta v_1, \Delta v_2 \cdots \Delta v_n]$ to define a new point, $C(\mathbf{v} + \mathbf{\Delta v}) = C([v_1 + \Delta v_1], [v_2 + \Delta v_2] + \cdots [v_n + \Delta v_n])$. This causes a small change to the cost function, $\Delta C = C(\mathbf{v} + \Delta\mathbf{v}) - C(\mathbf{v})$, which can be approximated in terms of partial derivatives in a similar way to the above equation 11 [Strang and Herman, 2016].

$$\Delta C \approx \sum_i \frac{\partial C}{\partial v_i}\Delta v_i \tag{12}$$

The gradient of C , ∇C ("nabla C") is the vector of all the partial derivatives.

$$\nabla C = \left[\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \cdots \frac{\partial C}{\partial v_n}\right]$$

Hence, we can write ΔC equivalently in vector form [Nielsen, 2015].

$$\Delta C \approx \nabla C \cdot \Delta\mathbf{v}$$

3.5 Parameters to Minimize Cost

Notice that by setting $\Delta\mathbf{v} = -\nabla C$, our expression for ΔC becomes

$$\Delta C \approx \nabla C \cdot (-\nabla C) = -\sum_{i=1}^{n} \left(\frac{\partial C}{\partial v_i}\right)^2$$

Every $\left(\frac{\partial C}{\partial v_i}\right)^2 \geq 0$, so their sum is greater than or equal to zero. After multiplying by -1, it becomes clear that $\Delta C \leq 0$, hence decreasing value of the cost function [Nielsen, 2015]. Moreover, we can vary increase the magnitude of ΔC by scaling $\Delta\mathbf{v}$ according to a scaling factor, η.

$$\Delta\mathbf{v} = -\nabla C \cdot \eta$$

In doing so, it is understood that every element in $-\nabla C$ is multiplied by η. When we do this, we enlarge $\Delta\mathbf{v}$ by scale factor η, which accordingly enlarges ΔC by scale factor η. Now, we have

$$\Delta C \approx \nabla C \cdot (-\nabla C) \cdot \eta = -\eta \sum_{i=1}^{n} \left(\frac{\partial C}{\partial v_i}\right)^2$$

Choosing larger values of η will lead to larger changes to the cost function. Hence, we call η the learning rate. However, we cannot make η too large, or else the cost function will not converge upon a local minimum [Nielsen, 2015].

Hence, the parameters $\mathbf{v}$ are updated according to the rule:

$$\begin{aligned} \text{define} \quad & \Delta\mathbf{v} = -\eta \cdot \nabla C \\ \text{now redefine} \quad & \mathbf{v} \to \mathbf{v} + \Delta\mathbf{v} \\ \text{resulting in} \quad & C(\mathbf{v} + \Delta\mathbf{v}) \leq C(\mathbf{v}) \end{aligned} \tag{13}$$

4 Backpropagation

Next, we compute the partial derivatives of the cost function using the backpropagation algorithm first designed by [Rumelhart et al., 1986].

4.1 Error

In this derivation, as presented in [Nielsen, 2015], I shall introduce the concept of the *error* δ of a neuron, which measures how "badly" a neuron is performing. Then, using extensive use of the chain rule [Strang, 1991], I will derive the 4 fundamental equations that allow us to compute the partial derivatives. These will consist of formulae to:

1. Compute the error values of the neurons in the output layer, layer L.
2. Use the error values of the neurons in layer l to compute the error values of the neurons in layer $l-1$. Hence by induction, we can find the error values of all the neurons in the network.
3. Use the error of the neurons to compute the partial derivatives of the biases
4. Use the error of the neurons to compute the partial derivatives of the weights.

First, let us define the error δ_j^l of the j^{th} neuron in the l^{th} layer as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{14}$$

Where z_j^l is the weighted input of the same neuron. To gain an intuition for why it is called "error", let us consider z as a parameter. Modifying z by the change $\Delta z = -\delta^2$ would correspond to a decrease in cost, specifically, $\Delta C = -\frac{\partial C}{\partial z}\delta^2$. If δ is large then we can greatly decrease cost by modifying z, suggesting that z is far from optimal, and implying that the neuron is performing worse than it could [Nielsen, 2015]. The opposite is true when δ is small.

4.2 4 Fundamental Equations

4.2.1 Error of the Output Layer

From the chain rule, the error of any neuron in the output layer, δ_j^L is

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \tag{15}$$

We can simplify this expression by thinking about each partial derivative separately. For the second half, recall that since $a_j^L = \sigma(z_j^L)$, we have

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'$$

The derivative σ' can be derived from the chain rule to simply be

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

For the first half, $\frac{\partial C}{\partial a_j^L}$ is dependent on the cost function we use, which in our case, is quadratic cost. Recall that cost can be defined as

$$C = \frac{1}{2}\sum_{k=1}^{N}(y_k - a_k^L)^2$$

Then, we can evaluate $\frac{\partial C}{\partial a_j^L}$ by treating all variables as constant except for the activation a_j^L.

$$\frac{\partial C}{\partial a_j^L} = \frac{\partial}{\partial a_j^L}\left(\frac{1}{2}(y_j - a_j^L)^2\right) = a_j^L - y_j$$

Therefore, we can write the error of a neuron in the output layer as

$$\delta_j^L = (a_j^L - y_j) \cdot \sigma'(z_j^L) \tag{16}$$

This expression is wonderfully simple and intuitive. If the output is close to the "correct" answer, then the error will be small, and vice versa. Furthermore, this is easily computable. We already computed z_j^L and a_j^L to obtain the output, and y_k is given to us in training.

4.2.2 Error of the previous layer

We can express δ_j^l in terms of the error of all the neurons in the $l+1^{th}$ layer.

$$\delta_j^l = \sigma'(z_j^l) \cdot \sum_{k=1}^{n} w_{k,j}^{l+1} \cdot \delta_k^{l+1} \tag{17}$$

In words, this expression states that the error of neuron j in the l^{th} layer is found by first multiplying the error of each neuron in the $l+1^{\text{th}}$ layer by the weight that connects the two neurons. Then, sum these products across all the neurons in the $l+1^{\text{th}}$ layer, and multiply by $\sigma'(z_j^l)$. An illustration of this implementation is included in the appendix (Figure 13).

This expression is essential. We have the errors of layer L, and hence we can compute the errors of layer $L-1$. By induction, we can calculate the errors of all the layers in the network by working backwards; hence the name, *backpropagation.*

To prove this, let us differentiate using the chain rule.

$$\frac{\partial C}{\partial z_j^l} = \sum_k \left(\frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l}\right) \tag{18}$$

To compute the second half of the sum, recall[9] that $z_k^{l+1} = \sum_n \left(w_{k,n}^{l+1} \cdot \sigma(z_n^l)\right) + b$. Since we are computing the partial derivative w.r.t a specific z_j^l, we can treat all other z and w values as constant except for when $n = j$. Thus we have $z_k^{l+1} = w_{k,j}^{l+1} \cdot \sigma(z_j^l) + \text{constant}$. Differentiating,

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{k,j}^{l+1} \cdot \sigma'(z_j^l) \tag{19}$$

[9]This is equivalent to the expression 3 in section 2.2, but with notation from section 2.5.

Substituting this expression into equation 18 and factoring out the constant $\sigma'(z_j^l)$, we obtain our desired expression.

4.2.3 Derivative of Cost w.r.t. any Bias

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{20}$$

This says that the partial derivative of cost with respect to the bias on neuron j in layer l^{th} is exactly equal to the error of that neuron. This is a beautiful result.

To prove it, let us apply the chain rule.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \cdot \frac{\partial z_j^l}{\partial b_j^l} \tag{21}$$

In computing $\frac{\partial z_j^l}{\partial b_j^l}$, we have $z_j^l = (\mathbf{w} \cdot \mathbf{x}) + b_j^l$, where $\mathbf{w}$ and $\mathbf{x}$ are constants. Hence differentiating, $\frac{\partial z_j^l}{\partial b_j^l} = 1$, and we have our expression as required.

4.2.4 Derivative of Cost w.r.t. any Weight

$$\frac{\partial C}{\partial w_{k,j}^l} = \delta_k^l \cdot a_j^{l-1} \tag{22}$$

In words, the error of one neuron multiplied by the activation of another neuron in the subsequent layer is equal to the partial derivative w.r.t. the weight that connects the two neurons. An illustration is included in Figure 14 in the appendix.

To prove this, let us use the chain rule once more.

$$\frac{\partial C}{\partial w_{k,j}^l} = \frac{\partial C}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{k,j}^l} = \delta_k^l \cdot \frac{\partial z_k^l}{\partial w_{k,j}^l} \tag{23}$$

We have $z_k^l = \sum_n \left(w_{k,n}^l \cdot a_n^{l-1}\right) + b$, with everything being constant except when $n = j$. Hence, $z_k^l = w_{k,j}^l \cdot a_j^{l-1} + \text{constant}$. Differentiating,

$$\frac{\partial z_j^l}{\partial w_{k,j}^l} = a_j^{l-1} \tag{24}$$

Substituting this gives our expression.

4.3 Mini-Batching

The four equations described above allow us to compute ∇C from a single training example. However, a single example may not be a good representation of the dataset as a whole. As a result, two different examples may yield slightly different costs despite the parameters $\Delta\mathbf{v}$ remaining unchanged [Nielsen, 2015]. To obtain a more representative value of the gradient, multiple values of ∇C are computed from different training examples, before being averaged[10]. This average, $\nabla\bar{C}$ better represents the cost function with respect to the entire dataset, and hence will lead to changes $\Delta\mathbf{v}$ which decrease cost for inputs across the whole dataset, rather than for a specific input. Thus, gradient descent is applied using this average $\nabla\bar{C}$. The set of training examples used to calculate $\nabla\bar{C}$ is called a *mini-batch*, and the number of examples used is called the *mini-batch size*.

5 The Machine Learning Algorithm

Now, we have all the tools to assemble a deep learning algorithm.

1. Define a cost function and randomly initiate the weights and biases.
2. Input a training example into the network and feed forwards to compute an output.
3. Use the output to compute the error of all the neurons in the network.
4. Use the error of the neurons to compute the partial derivatives of the cost function w.r.t the weights and biases. This gives the gradient of C, ∇C.
5. Compute slightly different values of ∇C using multiple training examples and average them to obtain $\nabla\bar{C}$.
6. Tweak the weights and biases using gradient descent, setting $\Delta\mathbf{v}$ equal to $-\eta\cdot\nabla\bar{C}$ for some learning rate η. This yields a set of weights and biases which decreases the cost.
7. Repeat the process for many training examples until the cost function reaches a local minimum.

[10]The average, $\bar{\mathbf{x}}$ of many vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ is found by taking an element-wise average, i.e., $\bar{\mathbf{x}} = [\text{avg}(\mathbf{a}_0, \mathbf{b}_0, \mathbf{c}_0), \text{avg}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1), \cdots]$.

6 Extending Bach Chorales

Having established the deep learning algorithm, I now apply it to music generation by implementing a neural network to extend musical melodies.

6.1 Dataset

I chose the Bach Chorales dataset [Conklin, 2016] as a starting point for generation due to the relative homogeneity of Baroque music compared to more modern styles. Chorales consist in a soprano melody accompanied by harmony from alto, tenor and bass. In order to simplify the task, I removed the harmonies from the data and focused on extending the melody line.

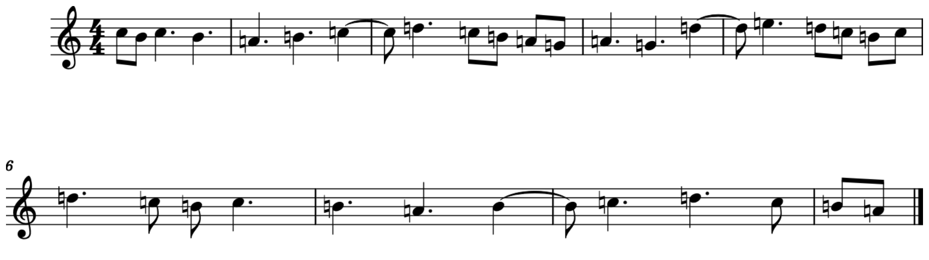


Figure 8: The melody line from a Bach Chorale

The dataset encoded notes as MIDI note values, where a number between 0 and 127 represented note pitch. Each pitch had a duration of $\frac{1}{8}^{\text{th}}$ of a beat, with longer notes being represented by repetitions of the same pitch. One disadvantage is that notes repeatedly played and held notes were indistinguishable.

6.2 Methodology

I re-encoded the data into a `NoteEmbedding` object, which consisted of two attributes: note pitch and note duration. MIDI note values between 36 and 81 were used in the melody, so only 46 different pitches needed to be represented. Moreover, using a "duration" attribute removed redundancy of repeated notes. To prevent numerically higher pitch values being interpreted differently to numerically lower pitch values simply by virtue of their representation, encoding note pitch using a single integer was avoided. Indeed using this encoding method resulted in the network giving the same note output for any input. Instead, I used a one-hot vector, consisting of a string of zeroes, with a single "1" at the index of the note pitch. E.g., $[0, 1, 0, \cdots 0, 0, 0]$ would represent the second lowest note. As such, input pitches would be treated equally regardless of being higher or lower [Briot, 2021]. Similarly, note duration was encoded as a one-hot vector representing the number of eighth notes it should be held for. These two vectors were concatenated together to create a `NoteEmbedding` object.

The input to the network would be any 32 consecutive notes of a melody, while the target output would be the next 32 notes of the melody [Todd, 1989], both represented as `NoteEmbedding` objects. The output `NoteEmbedding` object would be a vector of floating-point values, interpreted as a probability distribution for the likelihood of each note being chosen. The pitch and duration vectors were first split, then individually modified so their probabilities summed to 1. Then, a single pitch/duration was sampled based on this distribution.

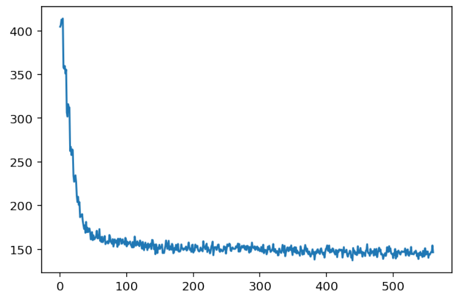
The output of the network could be fed back recursively as the input allowing melodies of arbitrarily long length to be generated. In this model, I used a vanilla deep neural network as described earlier, learning through gradient descent and backpropagation. This made extensive use of the `NumPy` library [Harris et al., 2020] and was inspired by code from [Nielsen, 2015]. Music was played using the `pretty_midi` library [Raffel and Ellis, 2014], and displayed using the `music21` library [Cuthbert and Ariza, 2010].

6.3 Results

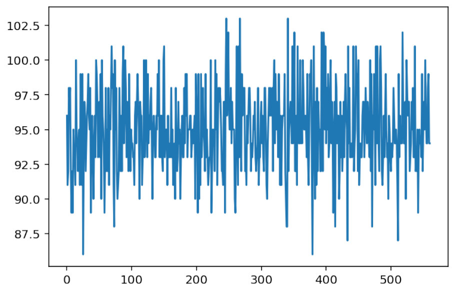
During training, cost decreased from 400 to 150, but converged around this value, suggesting the network had learnt some imperfect strategy (Figure 9). As expected, the output probability distributions started off as largely uniform, with equal probability given to each note pitch. On average, there were around 30 values with similar probabilities, loosely implying a 1/30 chance of choosing the right note. After training, however, only 4 values shared high probabilities, with the rest being close to zero. This implies the network was relatively sure that the target output was one of those 4 notes: a higher 1/4 chance of choosing correctly. Figure 10 displays the probability distributions for some of the output `NoteEmbeddings`. The more bars of high probability, the less “sure” the network is of its choice, and a marked decrease in the number of bars is seen after training.

Despite the decrease in number of bars, the remaining bars were spaced very far apart, spanning about 2-3 octaves. This is uncharacteristic of any music: even if there were many different notes that could follow a given melody, one would expect them to be generally grouped close together, e.g., by a third or a fifth. The large gaps imply that the melody generated will tend to make unmusically large jumps between notes which is highly undesirable.

Indeed, upon listening to the melodies generated before and after training, no distinguishable musicality could be discerned from the trained model compared to the untrained one. Subjectively, training made the melodies sound no better than random. This suggests that the network had not learnt the correct features of the music to produce coherent melodies (Figure 11).



(a) Initial epoch of training.



(b) A later epoch of training.

Figure 9: Plots of cost against progress of training at two different epochs. Cost initially decreased but then preceded to converge, despite giving incoherent melodies.

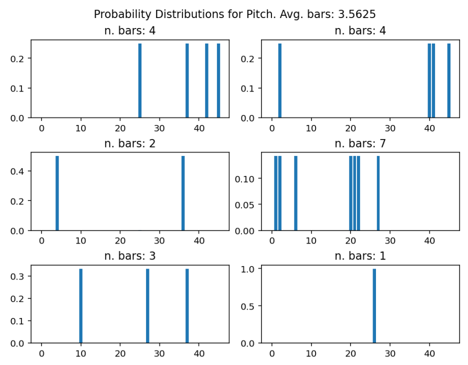
Moreover, a "score" was calculated by taking the average difference between the MIDI note value of the network's output and the MIDI value of the target output (so the lower the score, the better). This metric showed that "score" decreased from 500 to 440, suggesting that not very much had been learnt at all. In fact, the network only output the exact note pitch of the target output 3% of the time, i.e., 1/32 notes.

6.4 Analysis

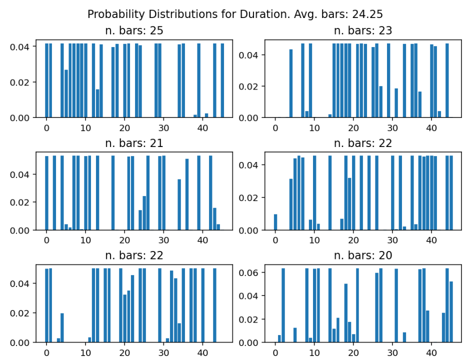
Despite apparent indications of learning, the network failed to capture musical features in its input.

One possible explanation for this is that the quality of the dataset was too low. The original melodies used in training were not extremely musical, and were rhythmically unstable. However, the pitches did form recognizable harmonies, which the network failed to pick up on. Hence dataset quality may have contributed, but was not sufficient to explain the lack of musicality of the network's outputs.

Another explanation is the incorrect setting of hyper-parameters, such as learning rate and network shape. However, I trained many different versions of the network with various hyper-parameter settings. Shapes of the network ranged from a single hidden layer of 50 neurons to three hidden layers of 500 neurons each, and learning rate varied from 0.25 to 20. In all cases, the same under-performing results occurred, suggesting that no combination of hyper-parameters could drastically increase performance with all else being equal.



(a) After training.

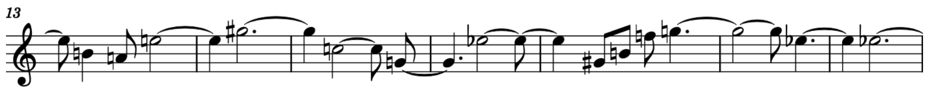


(b) Before training.

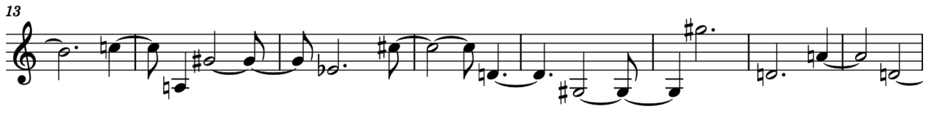
Figure 10: Probability distributions of some output notes, before and after training. Fewer bars after training show that the model is more confident about its prediction, indicating learning.

A more convincing, yet unsuccessful explanation is the small quantity of training data and short training time. The model was trained on just 561 examples, which meant that cost converged quickly within minutes of training and failed to decrease afterwards, despite a further 8 hours of training. A larger dataset, and a corresponding longer training time may have given the necessary musical information for the network to produce superior results. However, a model designed by [Huang et al., 2019] was able to harmonize a melody line in the style of Bach, having been trained on only 306 Bach Chorales. This suggests that small datasets *are* sufficient to learn the necessary musical features in the small domain of Bach Chorales, and that it was simply a shortcoming of the network architecture that led to the difference in performance. Indeed, the model by [Huang et al., 2019] was highly sophisticated and went far beyond the architecture of a vanilla neural network (though this was still the key component).

As a result, the most convincing reason for the network's failure was the inadequacy of the model itself. One problem with the vanilla neural network is that there is no clear order ascribed to the note embeddings. In music, the value of one note is highly dependent on the value of notes around it. Hence, to produce any coherent melody, the network must account for the relationship between nearby notes. However, the network receives all 32 notes of input at once, and must learn the input's sequentially connected nature of its own accord: something it clearly failed to do. To overcome this issue, two strands of refinement on the vanilla neural network become candidates for development. First, *convolutional neural networks* pre-process the input before feeding it into the deep



(a) Random outputs before training (above).



(b) Outputs after training.

Figure 11: Outputs after training are no more musical than the random outputs produced before training.

neural network, in hope of ascertaining the salient features. This is done by passing a filter over the input, which "summarises" the data from nearby points into something the network can more easily recognise [O'Shea and Nash, 2015]. The beauty of it is that this filter can be trained along with the network, and hence does not need to be explicitly specified. This is one of the key elements of the model by [Huang et al., 2019], showing that it has clear promise. Second, *recurrent neural networks* (RNNs) take sequences as inputs, treating each sequence element according to its value and position in the sequence. As a result, RNNs have been highly successful in music generation [Nayebi and Vitelli, 2015] and in the related domain of natural language processing [Sutskever et al., 2011]. I explore RNNs in more detail in the next section.

7 Recurrent Neural Networks

Recurrent neural networks (RNNs) take as their input a sequence, such as a string of words or a melody of musical notes. These inputs are "tokenised" into discrete chunks that are individually fed into the network, starting from the first token [Turner, 2024], and the network returns one output for each token that is input. In music generation, a melody can be tokenised by splitting it up into individual notes. At the heart of RNNs are still neural networks, but what distinguishes them from vanilla neural networks is their ability to reuse data from previous tokens to influence its output for the current token [Schmidt, 2019]. Hence it is able to more readily recognise patterns in sequences of data and perform tasks such as extending a given sequence.

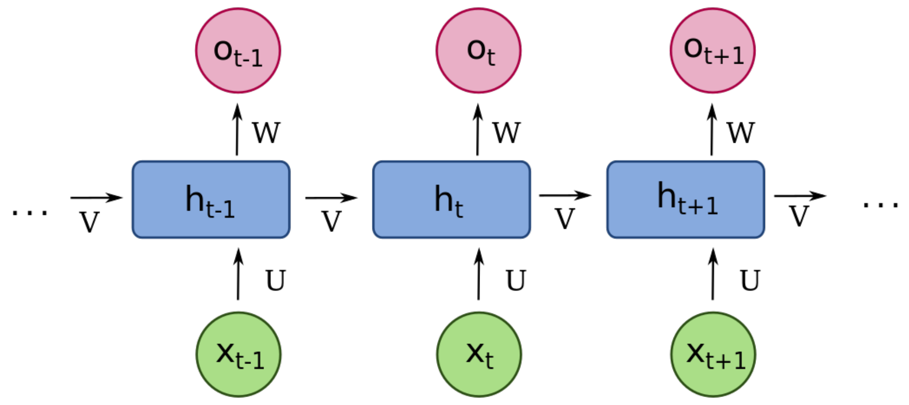


Figure 12: An unfolded RNN in feed forwards. Credit: *Wikipedia: Recurrent Neural Networks*

7.1 RNN Feed forwards

Imagine a simple RNN with an input layer, one hidden layer, and an output layer, as shown in Figure 12. The $(t-1)^{\text{th}}$ input token $\mathbf{x}_{t-1}$ is fed forwards according to a set of weights and biases, $\mathbf{W}_{x,h}$ and $\mathbf{b}_h$ respectively[11] to give the activations of the hidden layer, $\mathbf{h}_{t-1}$, also called the hidden state. Like a vanilla neural network, the activations of $\mathbf{h}_{t-1}$ are fed forwards with another set of weights and biases $\mathbf{W}_{h,o}$ and $\mathbf{b}_o$ respectively to produce the output $\mathbf{o}_{t-1}$. Unlike the vanilla network, however, we now move to the second token $\mathbf{x}_t$. To compute the activations of the hidden layer $\mathbf{h}_t$, we not only use the weighted inputs of the current token $\mathbf{x}_t$, but also the weighted hidden state of the previous layer $\mathbf{h}_{t-1}$. This recurrent step uses a different set of weights $\mathbf{W}_{h,h}$ connecting the previous hidden state to the current one. The output $\mathbf{o}_t$ is then computed as usual from the hidden state $\mathbf{h}_t$, and we move to the next input token $\mathbf{x}_{t+1}$. In general, the hidden state of the t^{th} token $\mathbf{h}_t$ is computed by taking into account the weighted current input $\mathbf{x}_t$ and the weighted hidden state from the previous input $\mathbf{h}_{t-1}$ and a bias $\mathbf{b}_h$. This can be expressed by the formula

$$\mathbf{h}_t = \sigma(\mathbf{x}_t\mathbf{W}_{x,h} + \mathbf{h}_{t-1}\mathbf{W}_{h,h} + \mathbf{b}_h) \tag{25}$$

[11]Recall that capital letters are matricies and lowercase letters are vectors. Also, the subscript $\mathbf{W}_{x,h}$ denotes that this is the weight matrix connecting the inputs x to the hidden layer h.

Recall that σ is the activation function. $\mathbf{x}_t$, $\mathbf{h}_t$ and $\mathbf{b}_h$ are column vectors, while $\mathbf{W}_{x,h}$ and $\mathbf{W}_{h,h}$ are weight matrices. Performing vector-matrix multiplication will weight an entire row of neurons, as explained in equation (4)[12]. The same idea can be generalised to multiple hidden layers [Schmidt, 2019].

The output $\mathbf{o}_t$ will depend directly on $\mathbf{h}_t$. $\mathbf{h}_t$ depends on $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$, while $\mathbf{h}_{t-1}$ depends on $\mathbf{x}_{t-1}$ and $\mathbf{h}_{t-2}$, and so on. From this we can see that the output at time t is affected by all the inputs that came before it, with inputs that are temporally closer to t having the greatest impact. This is highly desirable music generation, since it reflects the tendency of notes that are close together to have a greater connection with each other[13].

7.2 Training RNNs

RNNs can be trained to extend sequences by setting the target output of the network to be the input, but shifted forwards by one step. For example, the melody [A, B, C, D, E, F] could be used to train the network by giving as input the slice [A, B, C, D] and using as target output the slice [B, C, D, E]. As such, the network learns to predict the next token given the previous tokens. Extending the melody by a single note can recursively extend the melody forever. In the same example, we can input the melody [C, D, E, F], and may obtain the output [D, E, F, G], with G being the predicted next note. Next, using the output [D, E, F, G] as input we may obtain a new output of [E, F, G, A], with A being the predicted next note after G. The output melody can be input back to the network, and thus extended, an arbitrary number of times.

Gradient descent can be performed in the same way as vanilla neural networks, but the backpropagation algorithm must be modified slightly to account for the recurrent structure of the neurons. Though the principles of using chain rule to find the partial derivatives remain the same as described previously, its detail is beyond the scope of this project, but can be found in work by [Chen, 2016]. One problem RNNs face in backpropagation is that of exploding and vanishing gradients. Broadly, this occurs because the algorithm must compute the partial derivatives of very long sequences as a result of recurrence, and this involves performing matrix multiplication many times [Schmidt, 2019]. Thus, small values (< 1) in the matrices eventually approach zero (vanish), and large values (> 1) that are greater than one become very large (explode). Hence, the longer the chain of backpropagation, the more extreme are the magnitudes of the partial derivatives [Pascanu et al., 2013]. This has led to the rise of Long Short Term Memory (LSTM) modules [Staudemeyer and Morris, 2019]

[12]In fact, we can further extend this equation to act on the entire dataset by replacing the vector $\mathbf{x}_t$ with the matrix $\mathbf{X}_t$, where each row in $\mathbf{X}_t$ is the $\mathbf{x}_t$ vector for a single training example. This is applied similarly to $\mathbf{h}_t$.

[13]One potential limitation is that RNNs do not analyse inputs that come *after* the current input; something that convolutional neural networks achieve.

and Transformers [Vaswani et al., 2017] to resolve this issue.

Indeed, RNNs and their variants have been applied successfully in music generation, such as in work by [Gunawan et al., 2020] and [Jagannathan et al., 2022], promising further breakthroughs in the future.

8 Conclusion

In this project, I implemented a deep neural network to extend melodies from J.S. Bach's Chorales, but found that this simple architecture alone is insufficient for the task. Despite showing considerable signs of "learning", the model failed to pick up on relevant musical features, suggesting that a refinement in the model is required. One promising direction of development is the recurrent neural network, which may be more effective in adapting to the sequential nature of melodies.

References

[Bishop and Bishop, 2024] Bishop, C. M. and Bishop, H. (2024). *Gradient Descent*, pages 209–232. Springer International Publishing, Cham.

[Briot, 2021] Briot, J.-P. (2021). From artificial neural networks to deep learning for music generation: history, concepts and trends. *Neural Computing and Applications*, 33(1):39–65.

[Chen, 2016] Chen, G. (2016). A gentle tutorial of recurrent neural network with error backpropagation. *CoRR*, abs/1610.02583.

[Conklin, 2016] Conklin, D. (2016). Bach Chorales. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5GC7P.

[Cuthbert and Ariza, 2010] Cuthbert, M. S. and Ariza, C. (2010). Music21: A toolkit for computer-aided musicology and symbolic music data. In Downie, J. S. and Veltkamp, R. C., editors, *ISMIR*, pages 637–642. International Society for Music Information Retrieval.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Goyal et al., 2020] Goyal, M., Goyal, R., Venkatappa Reddy, P., and Lall, B. (2020). *Activation Functions*, pages 1–30. Springer International Publishing, Cham.

[Gunawan et al., 2020] Gunawan, A. A. S., Iman, A. P., and Suhartono, D. (2020). Automatic music generator using recurrent neural network. *International Journal of Computational Intelligence Systems*, 13(1):645–654.

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

[Hermans and Schrauwen, 2013] Hermans, M. and Schrauwen, B. (2013). Training and analysing deep recurrent neural networks. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc.

[Huang et al., 2019] Huang, C.-Z. A., Cooijmans, T., Roberts, A., Courville, A., and Eck, D. (2019). Counterpoint by convolution.

[Jagannathan et al., 2022] Jagannathan, A., Chandrasekaran, B., Dutta, S., Patil, U. R., and Eirinaki, M. (2022). Original music generation using recurrent neural networks with self-attention. In *2022 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 56–63.

[Jo, 2023] Jo, T. (2023). *Recurrent Neural Networks*, pages 247–275. Springer International Publishing, Cham.

[Moore, 2021] Moore, G. (2021). Cramming more components onto integrated circuits (1965).

[Nayebi and Vitelli, 2015] Nayebi, A. and Vitelli, M. (2015). Gruv: Algorithmic music generation using recurrent neural networks. *Course CS224D: Deep Learning for Natural Language Processing (Stanford)*, 52.

[Nielsen, 2015] Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

[OpenAI et al., 2024] OpenAI, Achiam, J., and et al., S. A. (2024). Gpt-4 technical report.

[O'Shea and Nash, 2015] O'Shea, K. and Nash, R. (2015). An introduction to convolutional neural networks. *CoRR*, abs/1511.08458.

[Pascanu et al., 2013] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA. PMLR.

[Raffel and Ellis, 2014] Raffel, C. and Ellis, D. P. W. (2014). Intuitive analysis, creation and manipulation of midi data with `pretty_midi`.

[Ray, 2019] Ray, S. (2019). A quick review of machine learning algorithms. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 35–39.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.

[Schmidt, 2019] Schmidt, R. M. (2019). Recurrent neural networks (rnns): A gentle introduction and overview. *CoRR*, abs/1912.05911.

[Staudemeyer and Morris, 2019] Staudemeyer, R. C. and Morris, E. R. (2019). Understanding lstm – a tutorial into long short-term memory recurrent neural networks.

[Strang, 1991] Strang, G. (1991). *Calculus*. Number v. 1 in Open Textbook Library. Wellesley-Cambridge Press.

[Strang and Herman, 2016] Strang, G. and Herman, E. (2016). *Calculus Volume 3*. OpenStax.

[Sutskever et al., 2011] Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1017–1024.

[Todd, 1989] Todd, P. M. (1989). A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43.

[Turner, 2024] Turner, R. E. (2024). An introduction to transformers.

[Vassakis et al., 2018] Vassakis, K., Petrakis, E., and Kopanakis, I. (2018). Big data analytics: applications, prospects and challenges. *Mobile big data: A roadmap from models to technologies*, pages 3–20.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

[Wang and Siau, 2019] Wang, W. and Siau, K. (2019). Artificial intelligence, machine learning, automation, robotics, future of work and future of humanity: A review and research agenda. *Journal of Database Management (JDM)*, 30(1):61–79.

[Weisstein, 2024] Weisstein, E. (2024). Partial derivative.

[Yunoki et al., 2024] Yunoki, I., Berreby, G., D'Andrea, N., Lu, Y., and Qu, X. (2024). Exploring ai music generation: A review of deep learning algorithms and datasets for undergraduate researchers. In Stephanidis, C., Antona, M., Ntoa, S., and Salvendy, G., editors, *HCI International 2023 – Late Breaking Posters*, pages 102–116, Cham. Springer Nature Switzerland.

A Appendix

A.1 Infographics for Equations 4.2.2 and 4.2.4

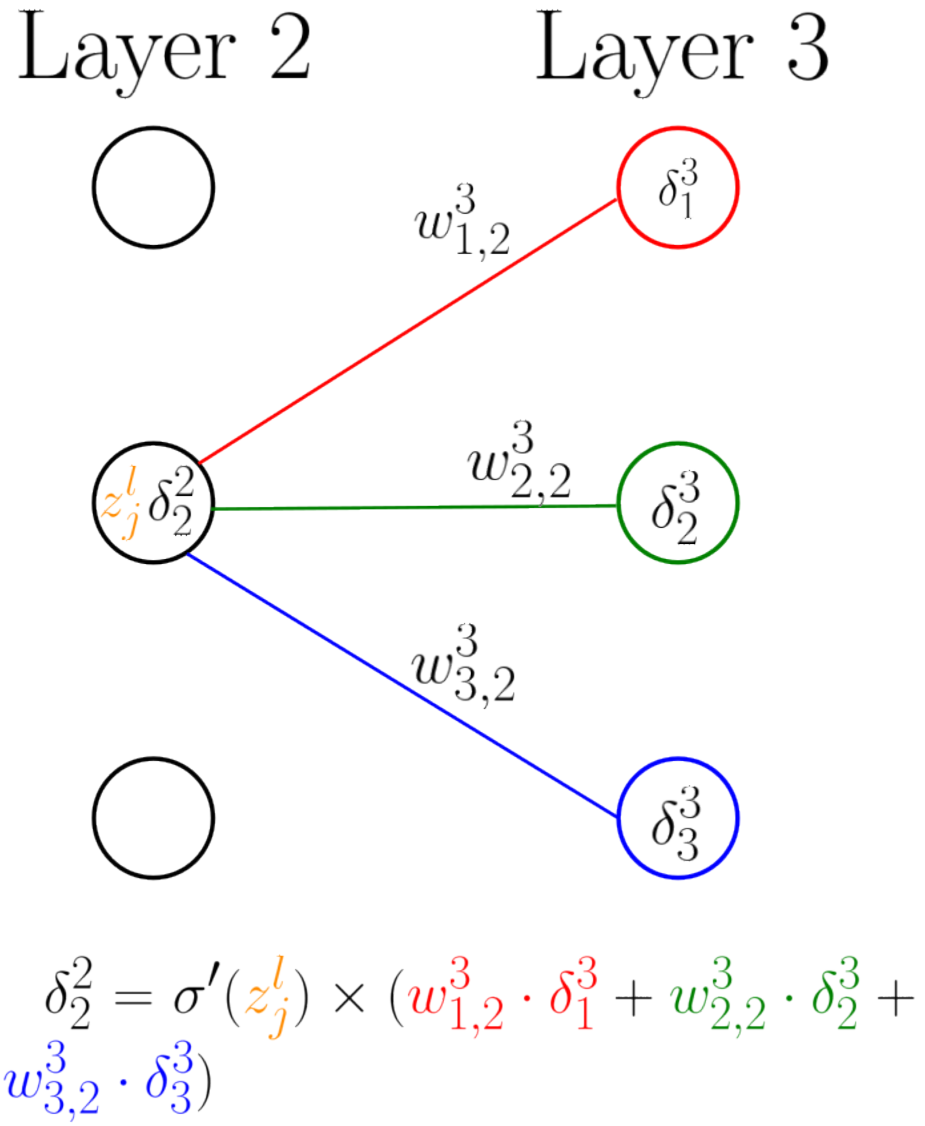
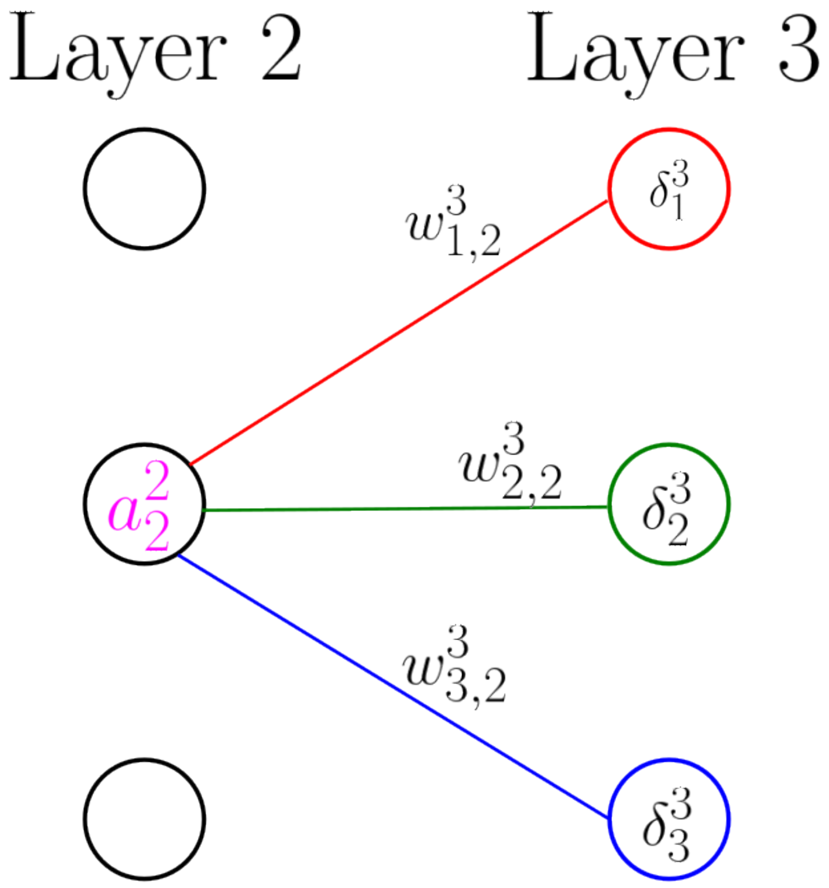

Layer 2
Layer 3
z_j^l
δ_2^2
δ_1^3
δ_2^3
δ_3^3
$w_{1,2}^3$
$w_{2,2}^3$
$w_{3,2}^3$
$\delta_2^2 = \sigma'(z_j^l) \times (w_{1,2}^3 \cdot \delta_1^3 + w_{2,2}^3 \cdot \delta_2^3 + w_{3,2}^3 \cdot \delta_3^3)$

Figure 13: The error of a neuron in layer 2 is calculated using the errors of all the neurons in layer 3 and their corresponding weights.



$$\frac{\partial C}{\partial w^3_{1,2}} = a^2_2 \times \delta^3_1 \qquad \frac{\partial C}{\partial w^3_{2,2}} = a^2_2 \times \delta^3_2$$

$$\frac{\partial C}{\partial w^3_{3,2}} = a^2_2 \times \delta^3_3$$

Figure 14: The activation of one neuron multiplied by the error of a neuron in the subsequent layer gives the partial derivative of cost w.r.t. the weight connecting the two neurons.