Artificial Intelligence for Composing Polyphonic Music

Using Bi-axial Recurrence in Deep Neural Networks

Binaya Raj Bhattarai

Computer Science and Information Systems Department, Minnesota State University Moorhead, 1104 7th Avenue South, Moorhead, MN 56563

Abstract:

With the high amount of computing power available today, it has been possible to synthesize complex neural networks that provide heuristic solutions to a wide variety of problems. One such problem is that of using the computer to compose music. By making two halves of a recurrent neural network recurrent along two different axes, it is possible to capture both temporal, and harmonic dependencies in music. Using the compositions of at least 30 different classical music composers of different era to train the neural network, it is possible to generate musical pieces that sound quite realistic. However, the nature of music is such that a composition which might be considered sound in the context of a certain musical style or era, but not so in the context of others. For this reason, it is hypothesized that training the network on compositions of a particular composer, or style, or era, would produce better music than training it on an amalgam of styles at the same time.

An *algorithm* is a sequence of tasks that can be performed to accomplish a task. Computer programs rely heavily on algorithms because they are guaranteed to produce the right result every single time. For example, an algorithm is employed to verify your password every time you log in to your email.

There are also many problems for which algorithmic solutions don't exist; for example, the problem of composing music using a computer. An algorithmic solution isn't possible because there are no set rules to follow to accomplish the task. *Heuristics* are used to solve such problems. A heuristic is a solution which produces something close to the expected result, but does not guarantee it.

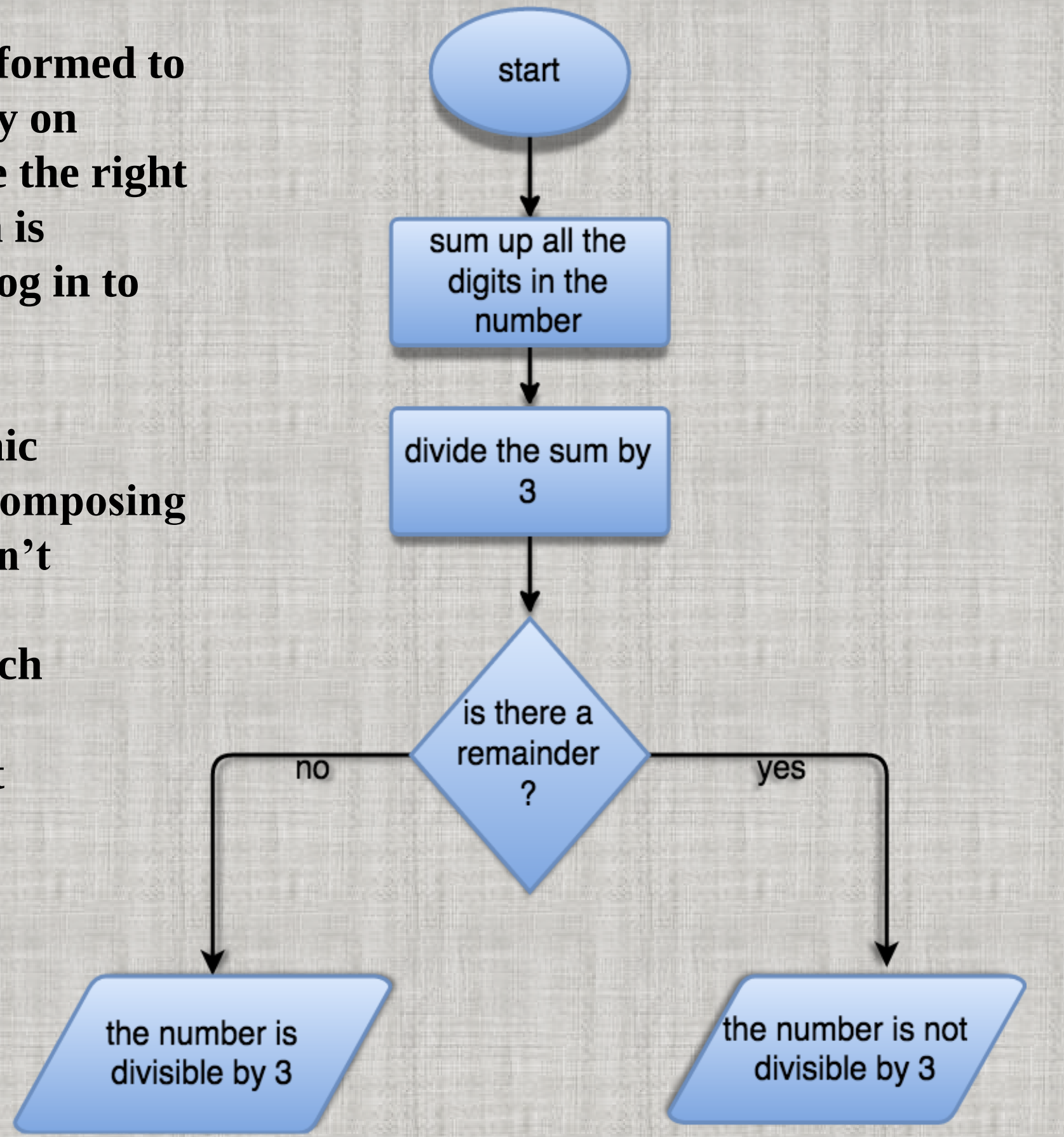

start
sum up all the digits in the number
divide the sum by 3
is there a remainder ?
no
yes
the number is divisible by 3
the number is not divisible by 3

Fig: An algorithm for checking if a number is divisible by 3

The image on the right shows an algorithm for checking if a given integer is divisible by 3. It is guaranteed to produce the correct result every time. *Try it!*

One form of heuristic problem solving in computer science is called *Supervised Machine Learning.* The training data for this technique consists of input samples for which the correct output are already known.
For example, the input could be the pixel values of a picture, and the output could be the label of the image. In the beginning, when the model makes a prediction, it produces an error because its weights are initialized to random values. However, that error is calculated and used to update the weights towards their optimal values. By repeating updating the weights many times, the model can be fine tuned to substantially decrease the total error.

The abundance of training data lies at the heart of supervised machine learning. However, because training requires processing large volumes of data, the process is very resource-extensive. Thankfully, most of the computations involve matrix multiplication operations, which can be sped up quite significantly using hardware and software components specialized for parallel processing. For example, the *Graphics Processing Unit(GPU)* can multiply matrices at least 10 times faster than the CPU.

To efficiently use components like the GPU, it is necessary to vectorize training data. *Vectorization* of data refers to the process of arranging data in vectors and matrices in order to allow computations to be performed using methods of linear algebra. Vectorized code can be run on the *Graphics Processing Unit(GPU)* at speeds at least 10 times greater than what can be achieved from a normal CPU.

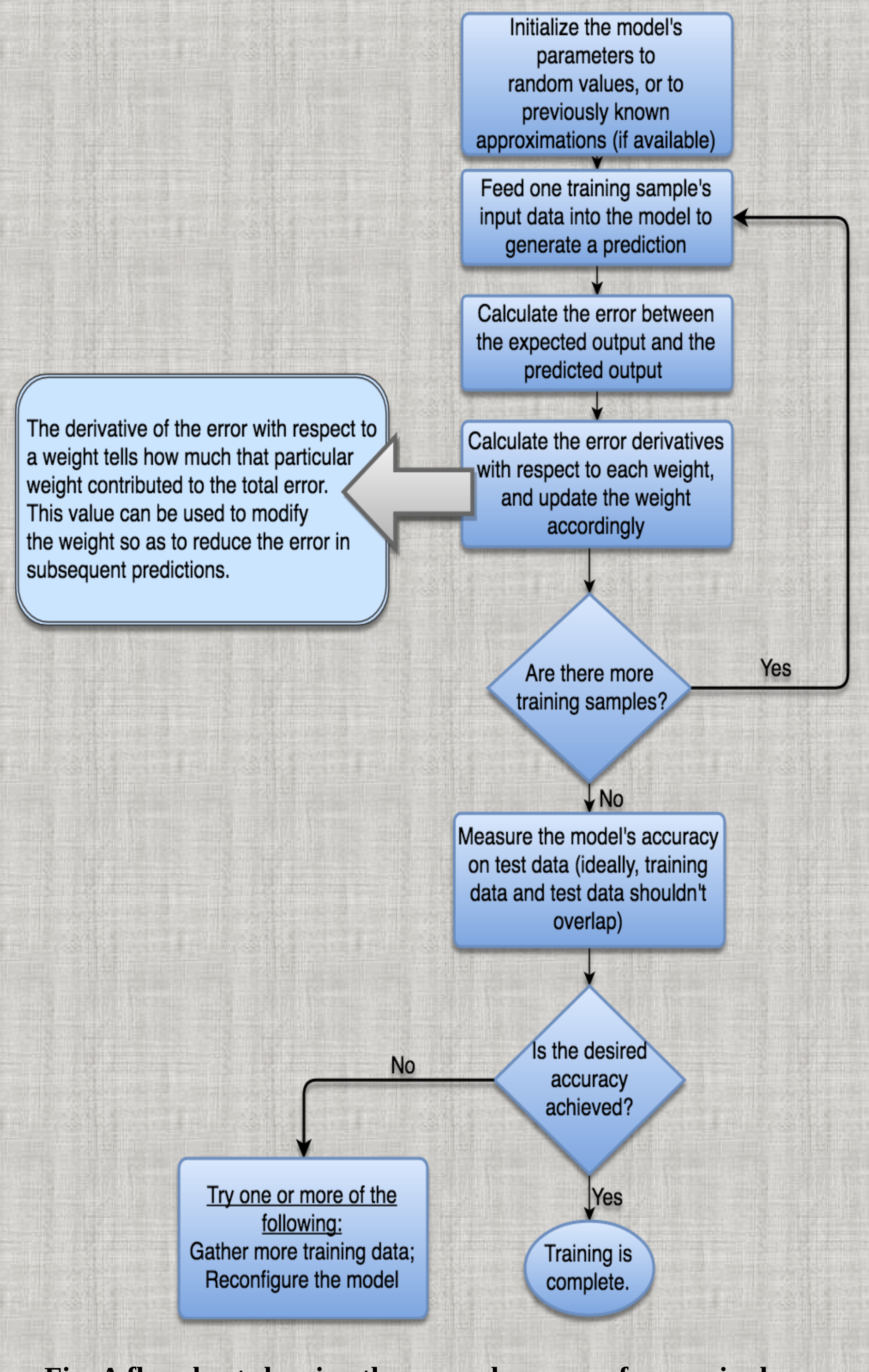

Initialize the model's parameters to random values, or to previously known approximations (if available)
Feed one training sample's input data into the model to generate a prediction
Calculate the error between the expected output and the predicted output
Calculate the error derivatives with respect to each weight, and update the weight accordingly
The derivative of the error with respect to a weight tells how much that particular weight contributed to the total error. This value can be used to modify the weight so as to reduce the error in subsequent predictions.
Are there more training samples?
Yes
No
Measure the model's accuracy on test data (ideally, training data and test data shouldn't overlap)
Is the desired accuracy achieved?
No
Yes
Try one or more of the following: Gather more training data; Reconfigure the model
Training is complete.

Fig: A flowchart showing the general process of supervised machine learning (many details are omitted in favor of simplicity)

An *Artificial Neuron* is a center of computation within a machine learning model called *Artificial Neural Network*. A neuron has a set of weights, one for each of its inputs, and a bias value. The image on the right shows a neuron with inputs x1, x2,..., x_m, and bias *b*.

When the neuron receives inputs, it performs the following computation on them:

$$F(X) = x_1w_1 + x_2w_2 + x_3w_3 + \ldots + x_mw_m + b$$

The value F(X), called the *activation state* of the neuron, is then passed through a (usually non-linear) *activation function*. If the result of the activation function exceeds a predefined *Threshold*, the neuron produces an output. This is referred to as *firing of the neuron*. In some cases however, the desired output might simply be the value of F(X), so the activation function is omitted.

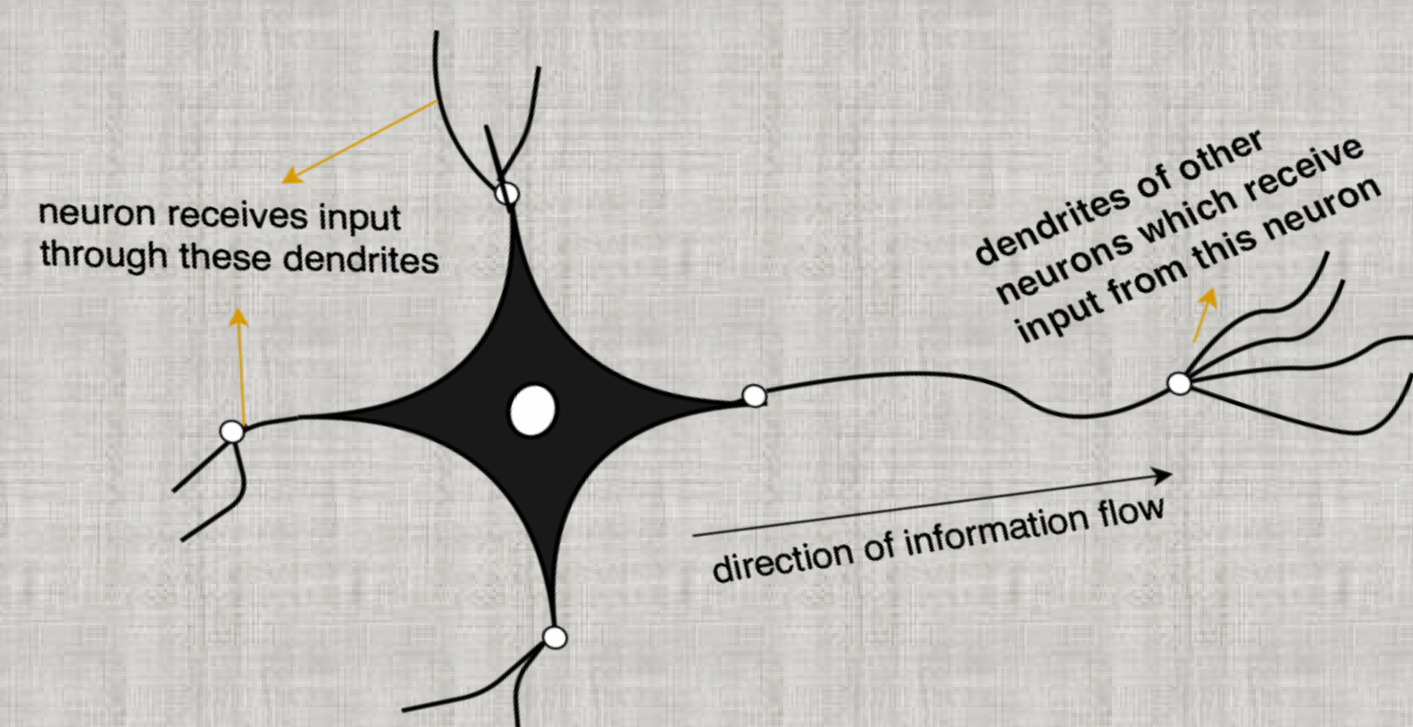

neuron receives input through these dendrites
dendrites of other neurons which receive input from this neuron
direction of information flow

Fig: A Biological Neuron

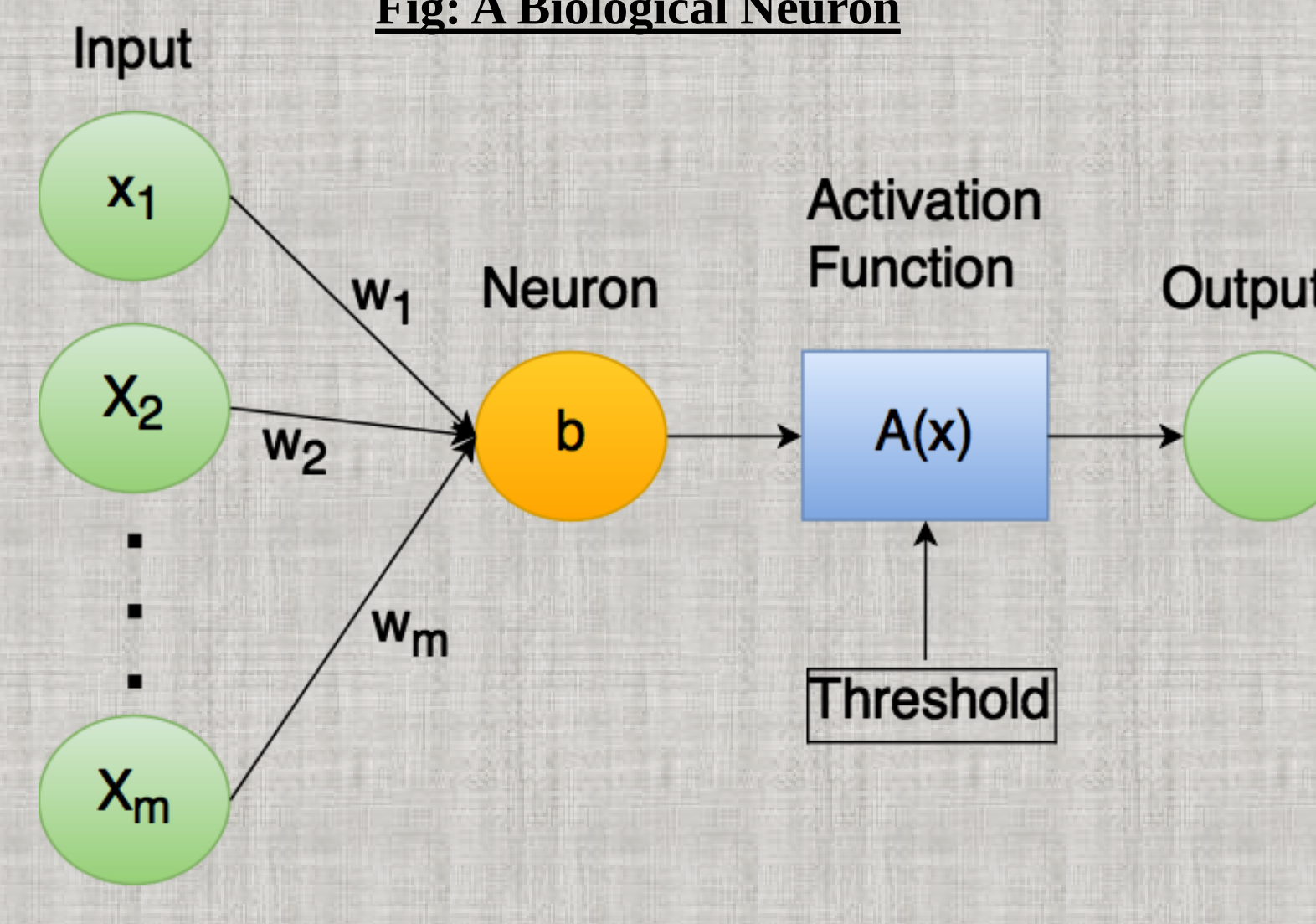


Fig: How a neuron converts input data into an output value

A *neural network* is created by stacking neurons together in layers then stacking several such layers. The layers in between the input and output layers are called hidden layers. Mathematically, each layer transforms applies a transformation to its input values. Intuitively, each transformation can be thought of as a higher representation of the same data. The neurons learn about underlying patterns in the input data by constructing such representations. Neural networks with more than one hidden layer are called *deep neural networks*.

A slightly advanced neural network is the *recurrent neural network*, which can predict sequential data. In order to achieve this, the neurons are fed in not only the data from the input layer, but also their own activation states from prior points in the sequence. A variation of model is called the *Long Short Term Memory (LSTM) model*, which specialized for retaining information over long sequences.

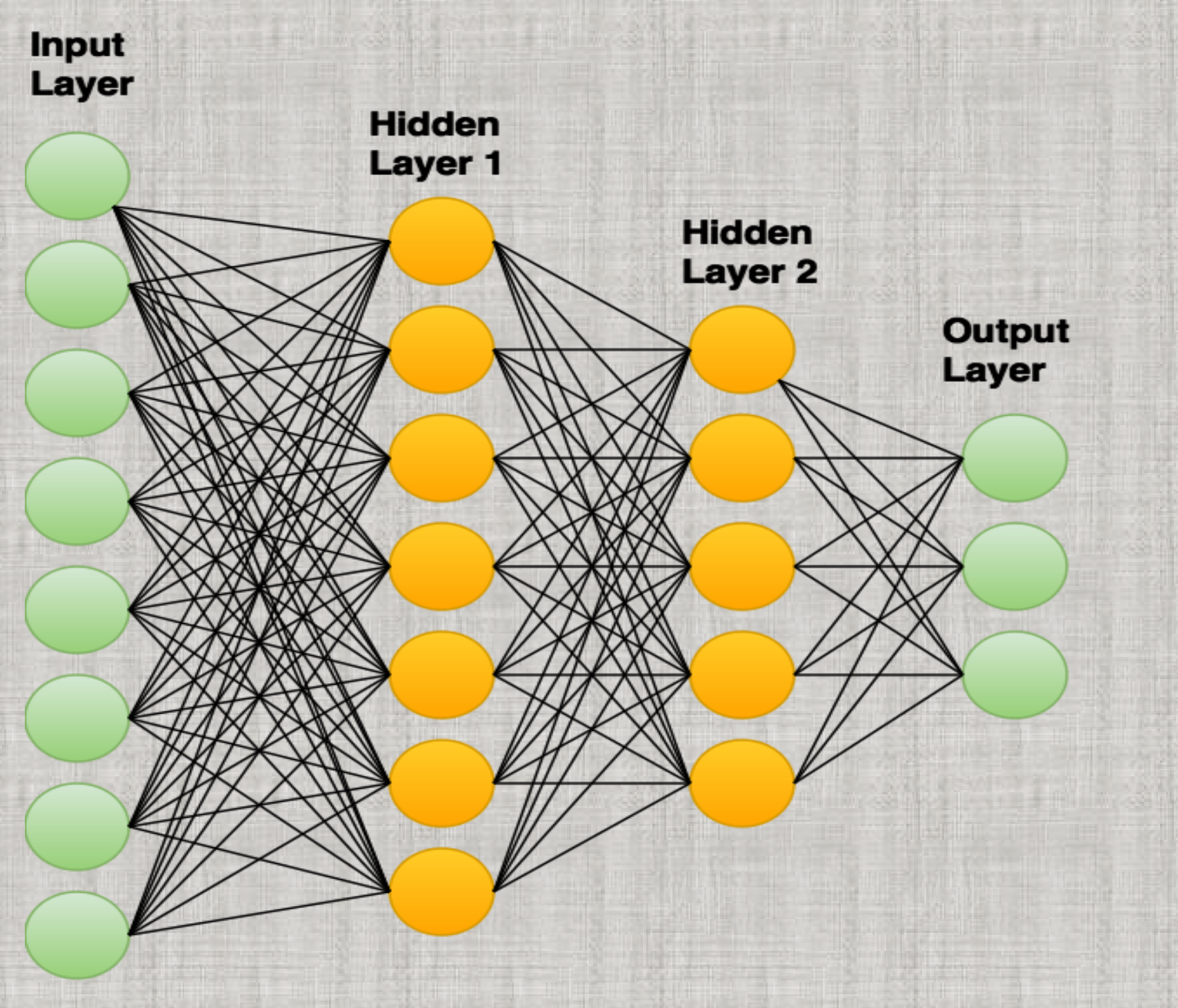


Fig: A fully-connected neural-network with 2 Hidden Layers

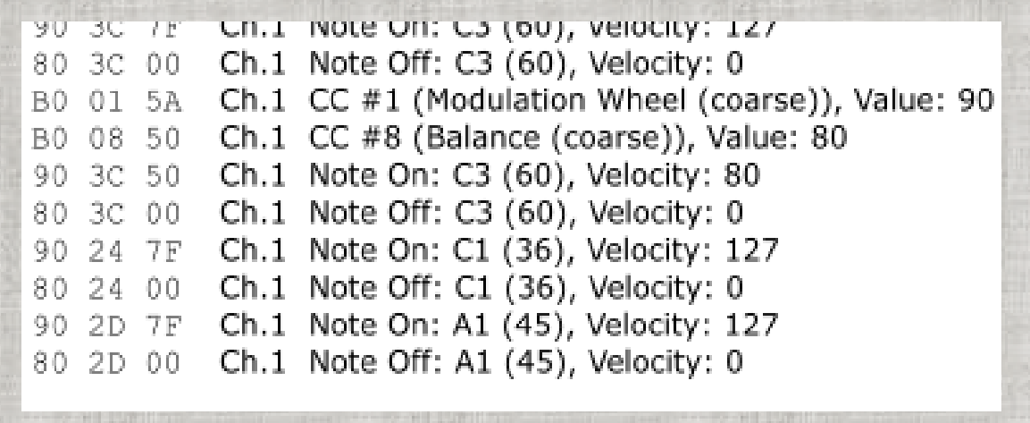
Training a Neural Net to Compose Music:

One form of digital audio representation is called MIDI(Musical Instrument Digital Interface), which represents music as a series of events occurring over time. The *Note On* and *Note Off* events specify exactly which pitches should be playing at any instance during the music's length. The advantage of MIDI over traditional audio formats is that MDI acknowledges the musical pitch-classes (C, D, E, …), which was essential for training this neural network.

It was quite obvious that a recurrent neural network would learn temporal dependencies in music by learning the music at each time-step as a function of the music at prior time-steps. The challenging part was to teach the network harmonic dependencies; the relationships between the different pitch-classes which form the basis of scales, and chords in music theory. The solution is to split the model into two recurrent parts: one recurrent along the temporal axis, and the other recurrent along the pitch-axis[1].

The machine learning library *Theano*, written in programming language *Python*, was chosen for implementing the model. All experiments were conducted on a remote *g2X2_large* GPU instance from Amazon AWS services[2].

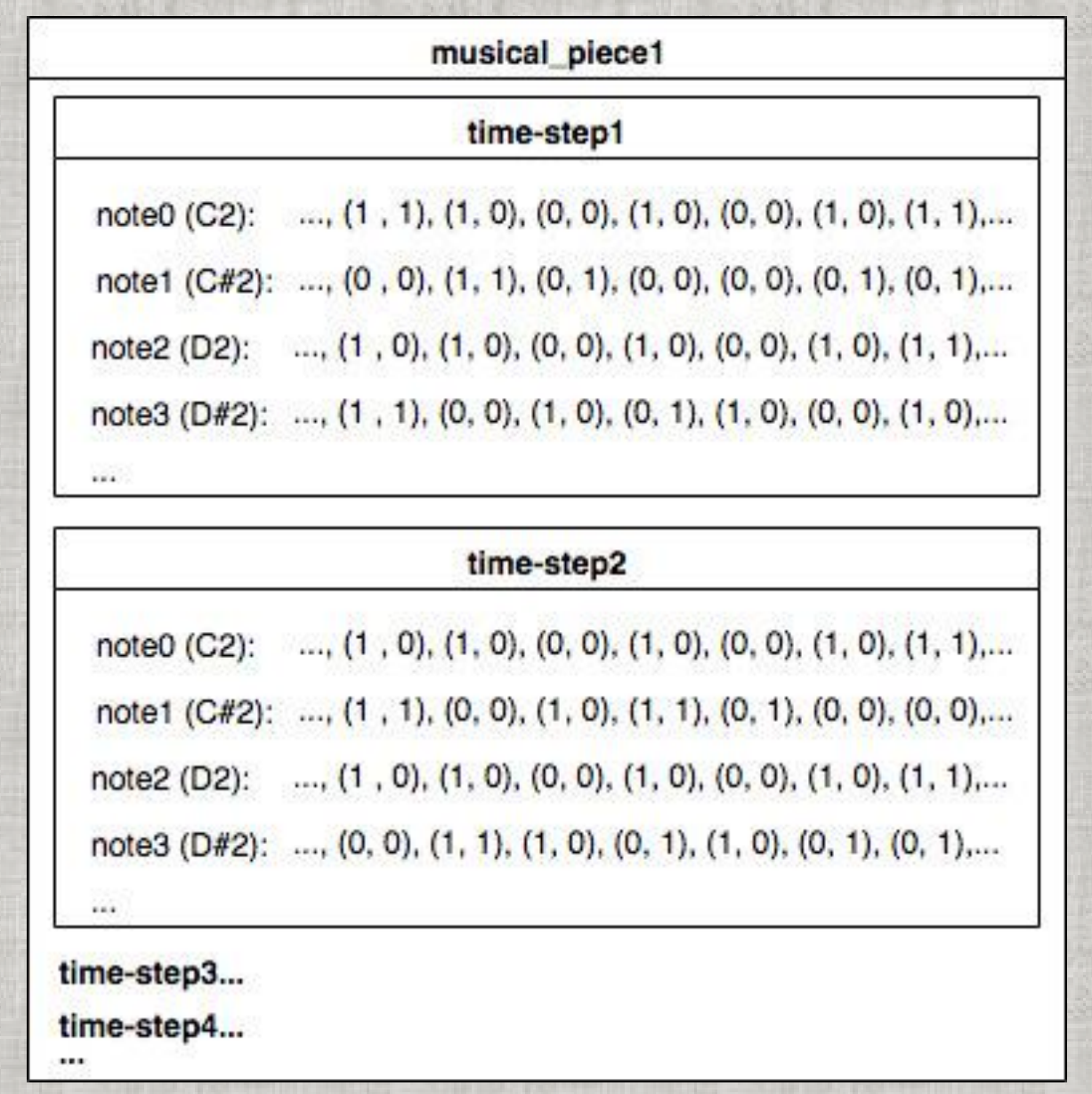
Data was captured from the MIDI files so that the MIDI Note *Note On* and *Note Off* events were captured, with respect to each pitch, for each time-step of each musical piece. The input is first fed to the layers recurrent along the temporal axis, then to those recurrent along the harmonic axis.



```
90 3C 7F   Ch.1  Note On: C3 (60), Velocity: 127
80 3C 00   Ch.1  Note Off: C3 (60), Velocity: 0
B0 01 5A   Ch.1  CC #1 (Modulation Wheel (coarse)), Value: 90
B0 08 50   Ch.1  CC #8 (Balance (coarse)), Value: 80
90 3C 50   Ch.1  Note On: C3 (60), Velocity: 80
80 3C 00   Ch.1  Note Off: C3 (60), Velocity: 0
90 24 7F   Ch.1  Note On: C1 (36), Velocity: 127
80 24 00   Ch.1  Note Off: C1 (36), Velocity: 0
90 2D 7F   Ch.1  Note On: A1 (45), Velocity: 127
80 2D 00   Ch.1  Note Off: A1 (45), Velocity: 0
```

Fig: A snapshot of MIDI data

The image above shows what MIDI data might look like for a short segment of music. The hexa-decimal values at the beginning of each specify the time at which each event would occur.



```
musical_piece1
  time-step1
    note0 (C2):   ..., (1 , 1), (1, 0), (0, 0), (1, 0), (0, 0), (1, 0), (1, 1),...
    note1 (C#2):  ..., (0 , 0), (1, 1), (0, 1), (0, 0), (0, 0), (0, 1), (0, 1),...
    note2 (D2):   ..., (1 , 0), (1, 0), (0, 0), (1, 0), (0, 0), (1, 0), (1, 1),...
    note3 (D#2):  ..., (1 , 1), (0, 0), (1, 0), (0, 1), (1, 0), (0, 0), (1, 0),...
    ...
  time-step2
    note0 (C2):   ..., (1 , 0), (1, 0), (0, 0), (1, 0), (0, 0), (1, 0), (1, 1),...
    note1 (C#2):  ..., (1 , 1), (0, 0), (1, 0), (1, 1), (0, 1), (0, 0), (0, 0),...
    note2 (D2):   ..., (1 , 0), (1, 0), (0, 0), (1, 0), (0, 0), (1, 0), (1, 1),...
    note3 (D#2):  ..., (0, 0), (1, 1), (1, 0), (0, 1), (1, 0), (0, 1), (0, 1),...
    ...
  time-step3...
  time-step4...
  ...
```

Fig: A snapshot of the data after pre-processing

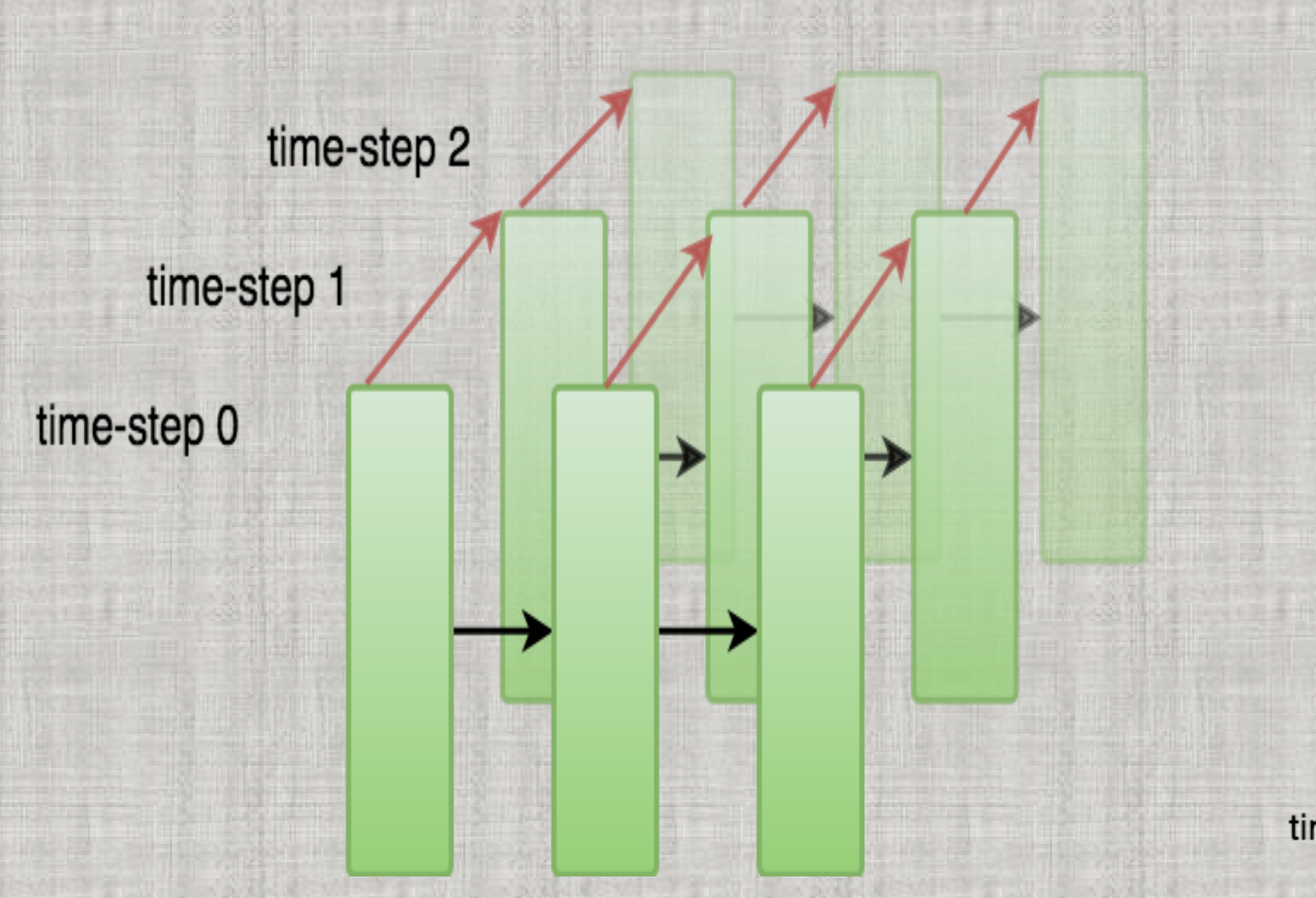


Fig: A Recurrent Neural Network having recurrence along the z-axis (shown in red arrows)

The image above shows a simple recurrent network with recurrence along the z-axis. However, the model employed for this experiment is reflected more accurately by the image to the right (the number of neural layers shown is hypothetical). For each temporal step, the note-axis layer recurs across its entire length, i.e., all the note-steps occur for each of the time-steps. This model captures both harmonic dependencies (scales/chords) as well as temporal dependencies (beat, time-signature, etc.)

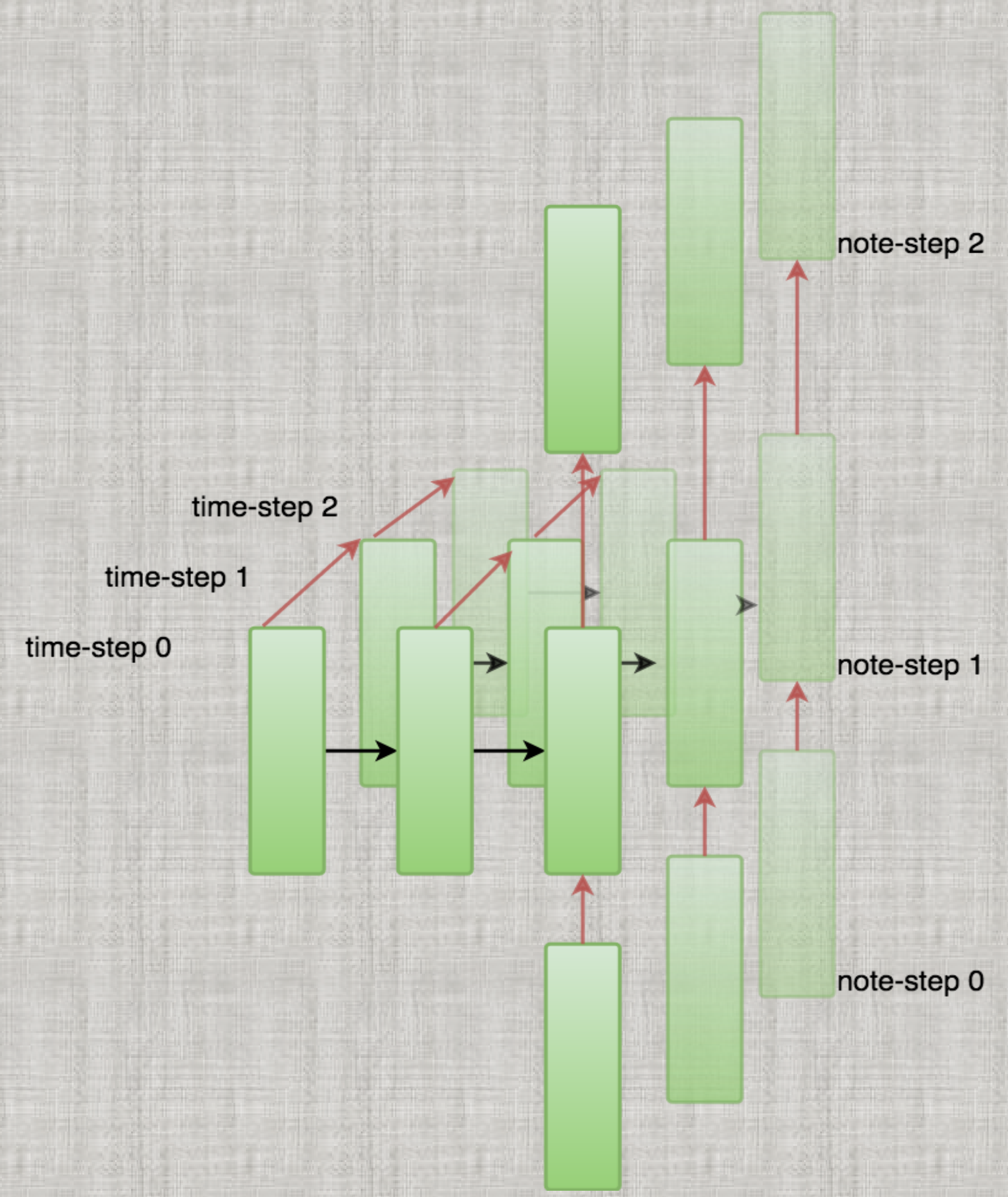


Fig: A Recurrent Neural Network having recurrence along two axes(shown in red arrows): the first two layers are recurrent along the z-axis, while the last layer along the y-axis

Results and Conclusion:

The model was trained using MIDI files of classical piano music downloaded from various open-source repositories. It was first trained using a collection of around 300 musical pieces by around 30 different composers of different musical eras and styles. Then it was trained once using only the works of L.V.Beethoven, then once using the works of W.A.Mozart, and F.J.Haydn together. The error minimization graph for the three training sessions are shown in the image to the right.
It is evident that the latter two training sessions resulted in a much lower error rate. However, because the amount of training data was quite low, the model didn't learn quite as many patterns; it's output sometimes tends to get stuck playing the same chords for prolonged periods of time.

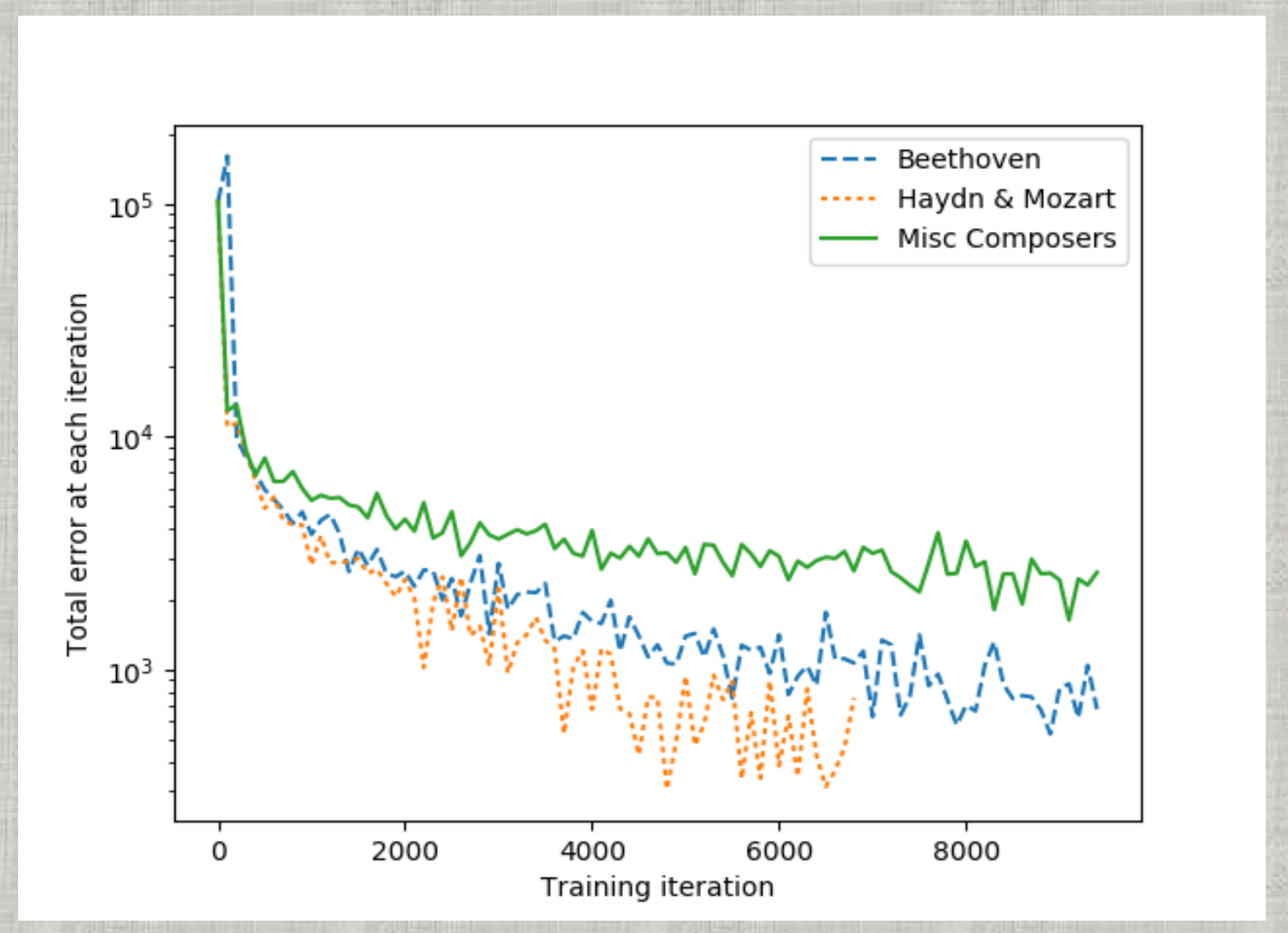


Fig: Comparison of error minimization when using data sets containing works of different composers for training

Another thing to notice when listening to the generated music is that the music is really lacking in the concepts of song structure and form. The neurons haven't learned when to start a song, when to end it, and how to create sub-sections within a composition. I believe this to be caused because the model was trained using 8-bar segments of songs, rather than entire songs. Unfortunately, training using entire songs isn't practical because it would take an exponentially larger computer memory to hold that much data. A practical approach towards improving the model would perhaps be to re-think how, and what input data gets extracted from MIDI files.

Acknowledgements:

All flowcharts and neural network illustrations were created using *draw*, an open-source tool. available at http://draw.io . The error minimization graph was created using *MatPlitLib*, a *Python* library.

Works Cited:

[1] (2015) Hexahedria: Composing Music With Recurrent Neural Networks. [Online]. Available: http://www.hexahedria.com/2015/08/03/composing-music-with- recurrent-neural-networks/
[2] (2017) Amazon Web Services. [Online]. Available: http://www.aws.amazon.com/

MINNESOTA STATE UNIVERSITY MOORHEAD