# Utilizing LSTMs to Generate Sheet Music

**Luke Feng and Kevin Loun**
Math 179: Harvey Mudd College
klaa2020@mymail.pomona.edu
lfeng@students.pitzer.edu

## Abstract

In this paper, we introduce the concepts and implementations of Long Short-Term Memory (LSTMs), a special type of Recurrent Neural Network. We examine an implementation of LSTMs that generates sheet music, which uses a Final Fantasy dataset of Final Fantasy .midi files. To further validate and expand upon our findings, we replicated the experiment using a subset of the GiantMIDI Piano dataset, offering new insights into its potential applications in automated music composition across varied genres.

## 1 Project Motivations:

The primary motivation behind this project stems from the evolving landscape of artificial intelligence in creative domains, particularly in automated music composition. The inception of Long Short-Term Memory (LSTM) networks and their notable success in processing sequential data have opened new avenues for exploring the generation of complex and aesthetically pleasing musical compositions. Our project is driven by the desire to harness the capabilities of LSTM models to generate sheet music, leveraging datasets such containing popular classical .midi files and a subset of the GiantMIDI Piano dataset.

**Bridging Computational Models and Artistic Creation:** The intersection of machine learning and music presents a unique challenge: how to create models that not only understand the structural patterns of music but also capture its emotional and expressive qualities. This project aims to explore this frontier, seeking to understand the extent to which LSTM networks can internalize and replicate the nuances

## 2 Structure and Inner Workings of LSTM Models

### 2.1 The Feed- forward Neural Network

Feedforward Neural Networks (FNNs) constitute the foundational architecture upon which more complex neural network models are built. Characterized by a unidirectional flow of data from input to output, FNNs consist of multiple layers of neurons, including an input layer, one or more hidden layers, and an output layer. Each neuron in a layer is connected to every neuron in the subsequent layer, with these connections embodying the learned weights of the network.

**Mathematical Model:** The operation of a neuron in an FNN can be described mathematically by a weighted sum of its inputs, followed by the application of an activation function. For neuron $i$ in

Submitted to 30th Conference on Neural Information Processing Systems (NIPS 2016). Do not distribute.

layer $l$, the output $a_i^l$ is given by:

$$a_i^l = \sigma\left(\sum_j w_{ij}^l \cdot a_j^{l-1} + b_i^l\right)$$

where $a_j^{l-1}$ represents the output of neuron $j$ in the preceding layer, $w_{ij}^l$ is the weight of the connection from neuron $j$ to neuron $i$, $b_i^l$ is the bias term for neuron $i$, and $\sigma$ denotes the activation function.

**Activation Functions:** The choice of activation function is crucial for enabling the network to capture nonlinear relationships between inputs and outputs. Common choices include the Rectified Linear Unit (ReLU), sigmoid, and tanh functions. The ReLU function, defined as $f(x) = \max(0, x)$, is particularly favored for hidden layers due to its computational efficiency and effectiveness in mitigating the vanishing gradient problem.

**Learning Process:** Training an FNN involves adjusting the weights and biases of the network to minimize a loss function, typically through backpropagation and an optimization algorithm like gradient descent. The loss function quantifies the difference between the network's predicted outputs and the actual target values for a set of training examples.

**Implementation Example:** In Python, utilizing frameworks such as TensorFlow or PyTorch, an FNN can be implemented by defining the model architecture, specifying the loss function, and selecting an optimization algorithm. For instance, a simple FNN for classification might employ cross-entropy as the loss function and stochastic gradient descent (SGD) as the optimizer.

Feedforward Neural Networks serve as the cornerstone for understanding more complex neural architectures. Their simplicity, coupled with their ability to approximate any continuous function given sufficient neurons in the hidden layers, makes them indispensable in the realm of machine learning and artificial intelligence. As such we will be using FNN as the foundation for the RNN developed for this project.

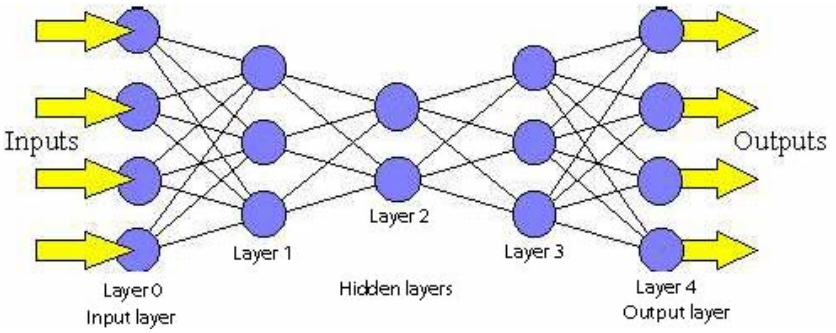



Figure 1: Feed Forward Neural Network

### 2.2 Fully Connected Layers

Fully Connected layers play a crucial role in the architecture of a Convolutional Neural Network. After the initial convolutional and pooling layers have extracted features from the input image, it is the task of the Fully Connected layers to interpret these features and make decisions, such as classifications. Essentially, Fully Connected layers create a high-level reasoning based on the low-level features identified by previous layers.

### 2.3 Linear Transformation of Data

The Fully Connected Layer begins by first conducting a linear transformation on the input vector. The layer works by initializing a perception (neuron) for each of the elements of the input vector. The

elements of the input vector act as inputs for these perceptions which apply a linear transformation. This appears in the form

$$y = Wx + b$$

where x is the input vector, W is the weight matrix, b is the bias vector, and y is the output vector. The weights in the initial weight matrix are initialized randomly and will be adjusted later during back propagation.

The linear transformation in fully connected layers effectively reduces the dimensionality of the data from the high-dimensional feature maps produced by convolutional layers. This process is crucial for integrating these features into a form that is suitable for classification. By performing a matrix multiplication with the weight matrix W, the network combines these features in various ways, condensing the information into a more manageable and useful format. Fully connected layers are designed to learn complex relationships between these features. The linear transformation allows the network to weigh these features differently, understanding which features are more important for making a final decision or classification. The output of the linear function is then used as inputs for the next layer of the Neural Network: The non-linear transformation.

### 2.4 Non-Linear Transformations

After the linear transformation, a non-linear activation function is applied to the output. This step is crucial for introducing non-linearity into the model, enabling it to learn more complex patterns.Common activation functions used in fully connected layers include: ReLU (Rectified Linear Unit): It introduces non-linearity while being computationally efficient. Sigmoid: Often used for binary classification tasks. Softmax: Used in the final layer for multi-class classification, providing probabilities for each class. These can occur in multiple layers of different activation functions but it can be represented as

$$y_{\text{activated}} = \sigma(Wx + b)$$

where $\sigma$ represents the chosen activation function. The final output of the neural network is also a non-linear transformation and outputs the probabilities for each class that the image could be.

## 3 Gradient Descent and Back Propagation

Gradient Descent is an optimization algorithm used to minimize the loss function in a neural network. The idea is to find the set of weights that results in the smallest possible error.

The loss function $L$ measures the difference between the predicted output of the network and the actual target values. For example, in a regression task, the Mean Squared Error (MSE) is a common choice:

$$L = \frac{1}{n}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$$

where $y_i$ is the true value and $\hat{y_i}$ is the predicted value.In classification tasks a common choice is cross entropy loss:

$$-\sum_{c=1}^{M}(y_{o,c}\log(P_{o,c})$$

The gradient descent algorithm optimizes the weights of a Convolutional Neural Network (CNN) through the following steps:

**Initialize Weights:** Start with initial values for the weights $W$ of the neural network. This can be done randomly or based on a specific strategy.

$$W_{\text{initial}} = \text{Random or Specific Initialization}$$

**Set Learning Rate:** Choose a small, positive learning rate $\alpha$. This parameter controls the size of the steps taken towards minimizing the loss function.

$$\alpha = \text{Small Positive Value}$$

**Iterative Optimization:** Repeat the following sub-steps until convergence:**Compute Gradient:** Calculate the gradient of the loss function with respect to the weights $\nabla W$.

$$\nabla W = \frac{\partial \mathcal{L}}{\partial W}$$

**Update Weights:** Adjust the weights in the opposite direction of the gradient.

$$W = W - \alpha \nabla W \tag{1}$$

**Check for Convergence:** Determine if the algorithm has converged based on the change in loss or a fixed number of iterations.

**Finalize Weights:** The process concludes when the stopping criterion is met, finalizing the optimized weights for the network.

The back-propagation process leverages the gradient decent process; back- propagation computes the gradient of the loss function with respect to each weight in the network by applying the chain rule of calculus in reverse order, starting from the output layer and moving backward through the network's layers. This computation effectively determines how much each weight contributes to the error, providing a direction and magnitude for how the weights should be adjusted to reduce the loss. Specifically the process works as follows:

**Begin at the Output Layer**:
Compute the error by determining the difference between the network output and the expected output.
Calculate the gradients of the loss function with respect to the activations of the output layer.

**Propagate the Error Backwards**: For each layer $l$, starting from the last hidden layer and moving towards the input layer:**Compute Gradient with Respect to Weights**:

$$\frac{\partial L}{\partial W_{j,k}^l} = \frac{\partial L}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial W_{j,k}^l} \tag{2}$$

where $a_j^l$ is the activation of neuron $j$ in layer $l$. Accumulate these gradients for all weights in the layer.

1. **Update Weights Across All Layers**: After calculating the gradients for all layers, update the weights by moving in the direction that minimizes the loss.
2. **Repeat Until Convergence**: Continue the process for a sufficient number of iterations or until the network's performance meets predefined criteria.
3. **Conclusion**: The network is now better tuned to make accurate predictions or classifications, having adjusted its weights based on the back-propagation of errors.

# 4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data effectively. Unlike feedforward neural networks, RNNs possess the unique capability to process inputs of varying lengths and maintain an internal state that captures information about the sequence they have processed so far. This characteristic is particularly useful for tasks such as language modeling, speech recognition, and time series analysis.

**Hidden State:** The core mechanism of RNNs is the hidden state, which acts as a memory of the network, retaining information about previous elements in the sequence. The hidden state at time $t$, denoted as $h_t$, is updated based on the previous hidden state $h_{t-1}$ and the current input $x_t$, according to the formula:

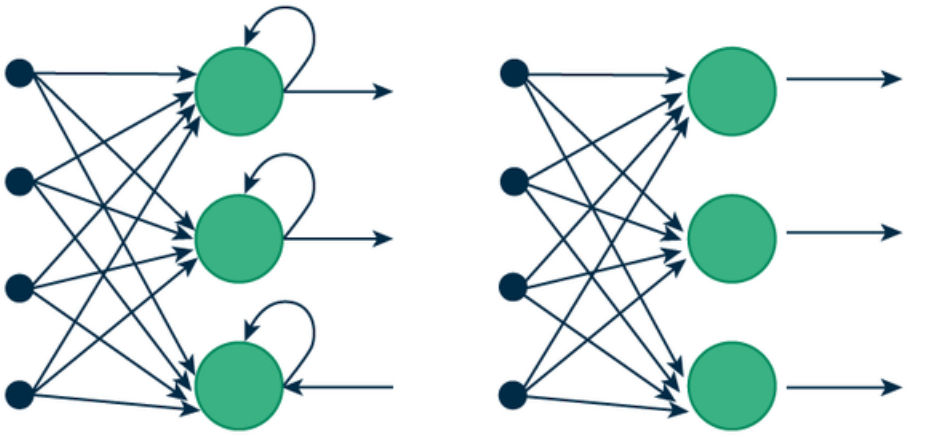
$$h_t = f(W \cdot h_{t-1} + U \cdot x_t + b)$$

where $W$ and $U$ represent the weight matrices for the hidden state and the input, respectively, $b$ is a bias term, and $f$ is a nonlinear activation function such as tanh or ReLU.

**Challenges:** Despite their flexibility and power, RNNs face significant challenges, notably the difficulty in capturing long-term dependencies due to the vanishing gradient problem. As the sequence length increases, gradients propagated back in time tend to either vanish or explode, making it challenging to train RNNs on long sequences.

**Advancements:** To mitigate these challenges, variants of RNNs such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) have been developed. These architectures introduce gating mechanisms to regulate the flow of information, allowing the network to better capture long-term dependencies and mitigate the vanishing gradient problem.

**Applications:** RNNs and their variants have been successfully applied to a variety of tasks that involve sequential data, including text generation, machine translation, voice recognition, and time-series forecasting. Their ability to model sequential dependencies makes them a powerful tool for tackling complex problems in natural language processing, speech analysis, and more.




## 5 Vanishing Gradient Problem

The vanishing gradient problem is a significant challenge encountered in the training of deep neural networks, particularly those involving recurrent architectures like RNNs. It refers to the phenomenon where gradients of the loss function with respect to the network's parameters become increasingly small as the error is propagated back through layers. This results in minimal updates to the weights of the earlier layers during training, hindering the network's ability to learn long-range dependencies.

**Mathematical Explanation:** Consider a neural network with $L$ layers, where the gradient of the loss function $E$ with respect to the weights $W^{(l)}$ of the $l^{th}$ layer is computed using the chain rule of calculus:

$$\frac{\partial E}{\partial W^{(l)}} = \frac{\partial E}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial W^{(l)}}$$

where $a^{(l)}$ and $a^{(L)}$ denote the activations of the $l^{th}$ and the $L^{th}$ (output) layer, respectively. The term $\frac{\partial a^{(L)}}{\partial a^{(l)}}$ represents the product of derivatives through the network's layers from $l$ to $L$, and it is crucial for understanding the vanishing gradient problem.

**Impact of Activation Functions:** The issue becomes pronounced in networks using activation functions like the sigmoid or tanh, where the derivatives can take on small values. Consequently, when these derivatives are multiplied through the layers, the product can become exceedingly small, leading to gradients that are practically zero. This effect compounds as the network depth increases, making it difficult for the network to adjust the weights of earlier layers based on the output error.

**Consequences and Solutions:** The vanishing gradient problem makes it challenging to train deep and recurrent neural networks, as it impedes the learning of long-term dependencies. Solutions to this problem include the use of alternative activation functions such as Rectified Linear Units (ReLU), which have gradients that do not saturate in the same way as sigmoid or tanh functions. Additionally, advanced network architectures like Long Short-Term Memory (LSTM) units and Gated Recurrent Units (GRUs) incorporate mechanisms to control the flow of gradients, effectively mitigating the impact of vanishing gradients by allowing the model to learn which information to pass through.

# 6 LSTMs

Long Short-Term Memory (LSTM) networks are a special kind of Recurrent Neural Network (RNN) capable of learning long-term dependencies. They were introduced by Hochreiter  Schmidhuber in 1997 to specifically address the vanishing gradient problem encountered by traditional RNNs during training. Here's how LSTMs help solve the vanishing gradient problem:

**Cell State**: The core idea behind LSTMs is the cell state, a kind of "conveyor belt" that runs straight down the entire chain of LSTM blocks. It allows information to flow relatively unchanged if needed. The cell state makes it easier for the network to transport information across many time steps, mitigating the vanishing gradient problem.

**Gates:** LSTMs contain three types of gates to control the flow of information:

**Forget Gate**: This gate decides what information should be thrown away or kept. By selectively letting information through, it helps the network to forget irrelevant parts of the previous states, reducing the risk of saturating the gradient during backpropagation.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

**Input Gate**: It updates the cell state with new information, allowing the network to add to its memory based on the new input it receives, which is crucial for learning dependencies.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

**Output Gate**: It decides what the next hidden state should be, which contains information based on the input and the memory of the cell state. This hidden state is used for predictions and is passed to the next time step.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \cdot \tanh(C_t)$$

where $\sigma$ denotes the sigmoid function, $[h_{t-1}, x_t]$ the concatenation of the previous hidden state and current input, and $W$ and $b$ are weights and biases specific to each gate.

**Gradient Flow in LSTMs** The design of the LSTM gates allows gradients to flow through the network more effectively. Since the cell state is designed to let gradients pass through unchanged, LSTM can maintain a stable gradient across many time steps, allowing it to learn long-term dependencies.

The gates of LSTM units use sigmoid functions for gating, which output values between 0 and 1, effectively deciding how much of each component should be let through. This mechanism, combined with the ability to add or remove content from the cell state, helps in mitigating the vanishing gradient problem by preserving the gradient flow over long sequences.

1. The architecture of LSTMs allows them to maintain a more constant error that can be backpropagated through time and layers. By carefully regulating the information that passes through the network, LSTMs can learn on data with long-range temporal dependencies, addressing the vanishing gradient problem that plagues standard RNNs.

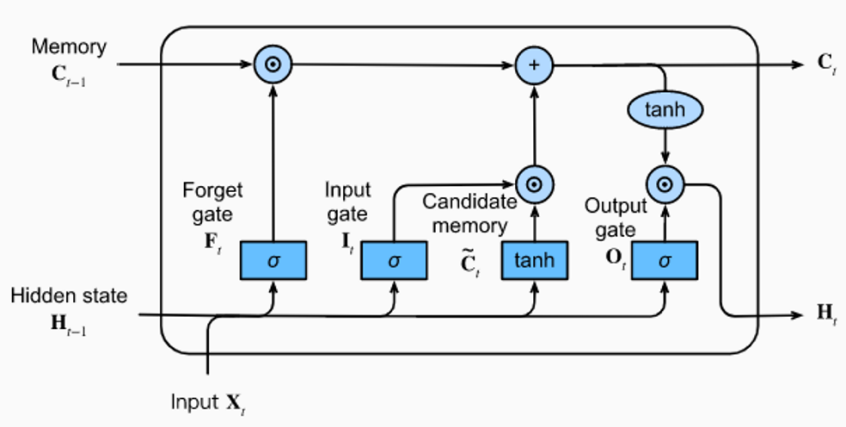



Figure 2: LSTM Model Diagram

# 7 Implementation Examples

In this section we apply the model given in https://github.com/Skuldur/Classical-Piano-Composer to a subset of the GiantMIDI dataset, containing 172 piano pieces from Romantic composers Lizst, Debussy, and Chopin.

## 7.1 Data

The data is split into object types, notes and chords. Each note contains information on the Octave, Pitch, and offset of the Note.

- Pitch refers to the frequency of the sound, represented with the letters [A, B, C, D, E, F, G], with A being the highest and G being the lowest.
- Octave refers to which set of pitches being used on the piano
- Offset refers to where the note is located in the piece.

Chord objects are essentially a container for a set of notes that are played at the same time.

```
...
<music21.note.Note F>
<music21.chord.Chord A2 E3>
<music21.chord.Chord A2 E3>
<music21.note.Note E>
<music21.chord.Chord B-2 F3>
<music21.note.Note F>
<music21.note.Note G>
<music21.note.Note D>
<music21.chord.Chord B-2 F3>
<music21.note.Note F>
<music21.chord.Chord B-2 F3>
<music21.note.Note E>
<music21.chord.Chord B-2 F3>
<music21.note.Note D>
<music21.chord.Chord B-2 F3>
<music21.note.Note E>
<music21.chord.Chord A2 E3>
...
```

Figure 3: excerpt from a midi file read using music21

### 7.2 Model Architecture

The model uses four different types of layers:

- LSTM Layers: Recurrent Neural Net layer that takes a sequence as an input and can return either sequences or a matrix.
- Dropout layers that set a fraction of input units to 0 at each update during the training to mitigate possible overfitting

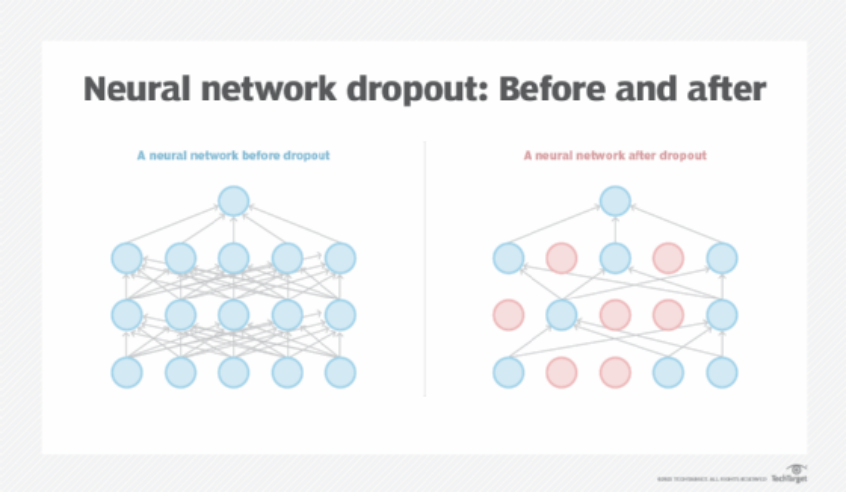



Figure 4: Dropout layer

- Normal Feedforward Neural Network with one Dense Layer using ReLU Function
- An Activation layer, specifically, a Softmax output player, outputting probabilities for a specific note

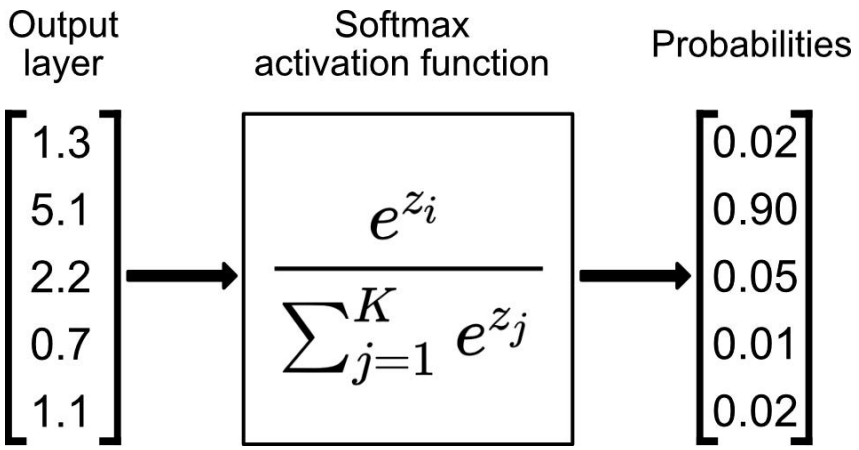



Figure 5: Softmax Activation Function

### 7.3 Getting Code to Run

A major hurdle for this project beyond understanding the methods was preparing the data and training the LSTM on such a large dataset. Usually this would not be a problem, but I (Luke) have a MacBook Pro with an M2 chip, which only has a 10-core GPU. While the code was able to run, it had a training time of about 3,400 hours, which was not feasible for the deadline of this project, with it taking about 17 hours per epoch and there being 200 epochs. As a result, we decided to try to tune the hyperparameters, decreasing the number of epochs from 200 to 50 and even lowering the sequence length from 100 to 10, but no significant advances were made in the runtime. As a result, we turned to using Google Colab, which offers the Nvidia T4 Tensor core GPU with limited availability. However, we found that our LSTM configuration did not meet the requirement for cuDNN (CUDA Deep Neural Network library, a GPU-accelerated library for deep neural networks) since our recurrent dropout value was larger than 0, so our code wasn't able to use the T4 GPU. Thus, we decided to remove the layer altogether. This sped up our runtime, but it was still too slow, since it took about 30 minutes to run each epoch, so it would have taken about 100 hours to run all 200 epochs. Because of this, we opted to subscribe to Google Colab Pro, which provided us access to the NVIDIA A100 Tensor Core GPU. This sped up our time to about 100 seconds per epoch, but we had limited computing resources allocated to us, since Google Colab runs on the cloud, thus we had to reduce our number of epochs to about 50 for our first runtime to avoid using up all of the resources allocated to us. An early stoppage function was also added to prevent us from using up all of our "computing units," which terminates the runtime if the loss does not improve/decrease for more than 5 epochs. Below are some examples of sheet music generated with the input dataset.

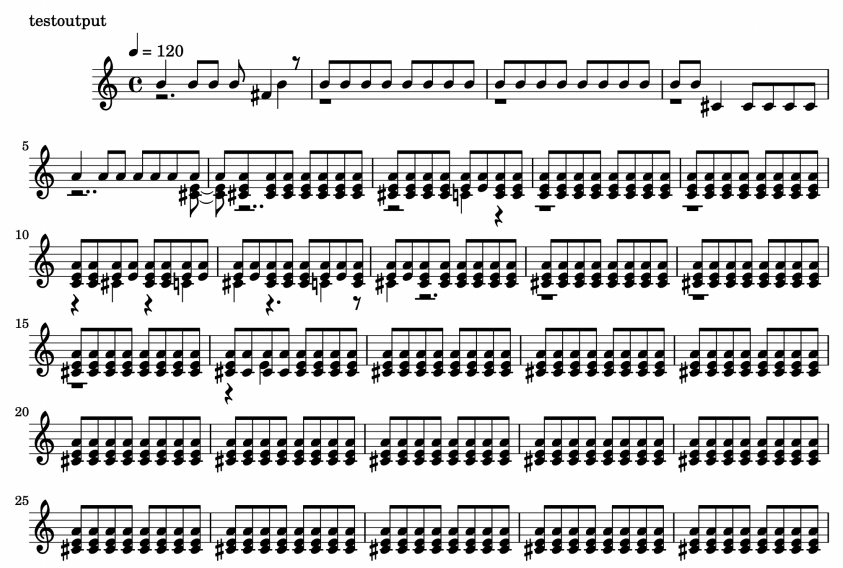



Figure 6: Trained on 50 epochs with sequence length 100

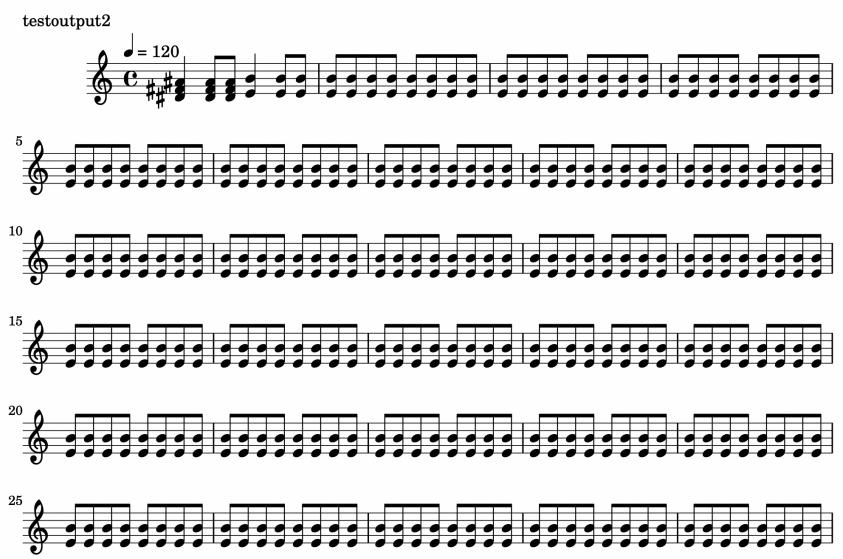



Figure 7: Trained on 200 epochs (terminated early at epoch 24) with sequence length 10

# 8 Future Directions

Removing the recurrent dropout layer and reducing the number of epochs might have led to overfitting, which is unfortunate, but understandable given the circumstances. This might explain the repeated sequence of notes.

## 8.1 Alternate Implementation

Another possible explanation is due to the argmax implementation of the model. While this is meant to detect patterns, sometimes it makes the model "too good" at predicting patterns. This is brought up in a blog by David Exiga (Exiga, 2020). The article gives the following example: "For example, say during training the model picks up on the following note pattern: “A, A, A” In other words, it

thinks that the more As the model sees, the more likely the next note should be A. Now, consider the following scenario with our quirky 'A' model using the past 3 previous notes to predict upon and only 3 possible notes to predict: A, B, or C. In the first iteration it is given "A, B, C" to predict on and the corresponding probability distribution is [0.4, 0.3, 0.3]. Argmax says to predict A. In the second iteration it is given "B, C, A" and the corresponding probability distribution is [0.8, 0.1, 0.1]. Argmax says again to predict A. In the third iteration the model is given "C, A, A" and the corresponding probability distribution is [0.98, 0.01, 0.01]. Argmax once again says to predict A. Now, the model is stuck predicting A forever!"

### 8.2 Positive Feedback System

Positive feedback systems in neural network training can significantly influence the model's learning dynamics, enhancing its ability to reinforce learned patterns and behaviors. Such systems can be implemented in music generation models to amplify the recurrence of motifs and harmonic structures that are deemed favorable or interesting.

**Mathematical Framework:** In the context of LSTMs, a positive feedback mechanism can be mathematically represented by adjusting the weight update rule to incorporate feedback from the output layer. This can be expressed as:

$$W_{new} = W_{old} + \alpha \Delta W + \beta F(y)$$

where $W_{new}$ and $W_{old}$ are the new and old weights, respectively, $\alpha$ is the learning rate, $\Delta W$ is the traditional weight update derived from backpropagation, $\beta$ is the feedback strength, and $F(y)$ is a function of the output $y$ that represents the feedback signal.

**Implementation Insights:** In a Python-based LSTM model, this could involve adjusting the model's training loop to include a feedback term when updating weights. The feedback function $F(y)$ could be designed to measure aspects such as the melodic coherence or rhythmic complexity of generated sequences, rewarding patterns that align with desired musical properties.

**Potential Improvement:** Using a positive feedback system could assist in improving the model and having it learn different patterns from the data. This could result in it learning the distinction between cords and octaves. However, this would require multiple runs of re-training the model which is costly and time inefficient so it would potentially be better to look at alternative options.

### 8.3 Music Theory Implementation

Incorporating music theory into LSTM models for music generation involves encoding theoretical knowledge—such as scale, harmony, and rhythm—into the network's architecture or training process. This can guide the model to generate musically coherent pieces that adhere to established compositional rules.

**Mathematical Representation:** One approach is to augment the input or output layers of the LSTM to include a representation of music theory knowledge. For instance, a vector encoding that represents the key or scale can be appended to the input at each timestep:

$$X_{augmented} = [X; M]$$

where $X$ is the original input vector, $M$ is the music theory vector, and $X_{augmented}$ is the augmented input vector.

**Coding Consideration:** In practice, this might involve modifying the data preprocessing pipeline to include music theory-based features. For a Python implementation using libraries such as TensorFlow or Keras, this could mean extending the input tensors to include additional channels for music theory attributes, which are then processed by the LSTM layers to influence the generation process.

**Practical Example:** When training on datasets like GiantMIDI, additional preprocessing steps can parse the MIDI files to extract theoretical elements (e.g., key signatures, time signatures) and encode these into the input vectors. This guides the LSTM in learning not just from the raw note sequences but also from the underlying music theory, resulting in outputs that are more structured and theoretically sound.

**Potential Improvement:** Implementing music theory in this manner would subvert the issue in previous; having to re-run the model. However, this method requires more extensive knowledge on not only coding but also music theory.

# 9 References

Exiga, D. (2020, May 6). Music Generation Using LSTM Neural Networks. Medium. https://david-exiga.medium.com/music-generation-using-lstm-neural-networks-44f6780a4c5

Skúli, S. (2017, Dec 7). How to Generate Music using a LSTM Neural Network in Keras. Towards Data Science. Available: https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5.