Deep learning project

Alexander Caning, Lucas Rasmussen, Søren Peddersen

12/2024

1. Introduction

When creating new music, musicians face multiple challenges: It can be hard to get an initial starting point for their music, the artists get stuck in the process or wish to extend the piece. One approach to solve this problem is to take an existing song and make a neural network predict the last part of the song. Attempts to predict and generate piano songs using the LSTM framework have already been made. Still, no one has yet had the idea of trying to use the newly updated framework of the xLSTM to solve the music generation problem. As the territory with the LSTM model has already been explored, a baseline model has been established. This project seeks to discover whether changing the model to the xLSTM framework could increase performance and accuracy in this task.

2. Dataset

Our dataset is comprised of 216,284 unique Irish songs encoded in ABC notation, which is a way encoding musical notation for computers. Keys, pauses, beats per minute, note lengths, among many others, are all expressed in this notation. The data was sourced from an open-source dataset published on Hugging Face. (Wu, S., et. al.)

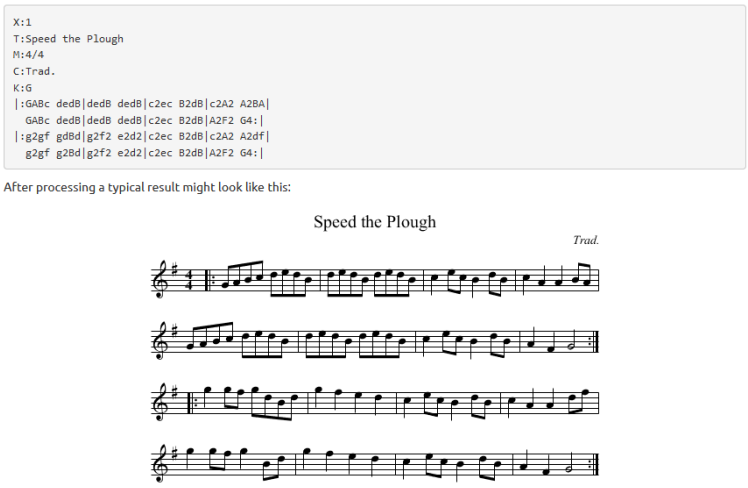

Figure 1: An example of ABC notation translated to sheet music. Taken from abcnotation.com.

To verify the usefulness of the data, we utilized an online ABC editor, which converts ABC notation into its corresponding sheet music and converts it to a playable format. We tested over 50 songs from the dataset which returned no errors in the rendering of the music notation. We had a few minor problems with the data, however: The structure of the songs was inconsistent. Some included comments embedded within the notation. Some specified beats per minute while most did not. We did not standardize these differences, since the dataset is quite large, and we considered variability to be a part of the real world. Fortunately, these inconsistencies seemed rare enough to not critically impact the models' training.

2.1 Data preprocessing:

Each song in the dataset was concatenated into a single text file and each unique character in the dataset was mapped to a corresponding integer value. In figure 2, we observe that white spaces were highly dominant in the dataset. In fact, whitespace was the most frequently occurring character in the data, constituting over 25% of the entire dataset. Deciding to remove all white spaces was a logical choice as they did not provide any useful information for the model, and in terms of the rules for ABC notation is included only for human readability. The model operated without issue in the absence of white spaces, as each character remained unique. At initial stages *before* removing whitespace the model achieved an accuracy of approximately 25%. This seemed to be caused by the model simply predicting whitespaces understandably due to the high frequency in the data.

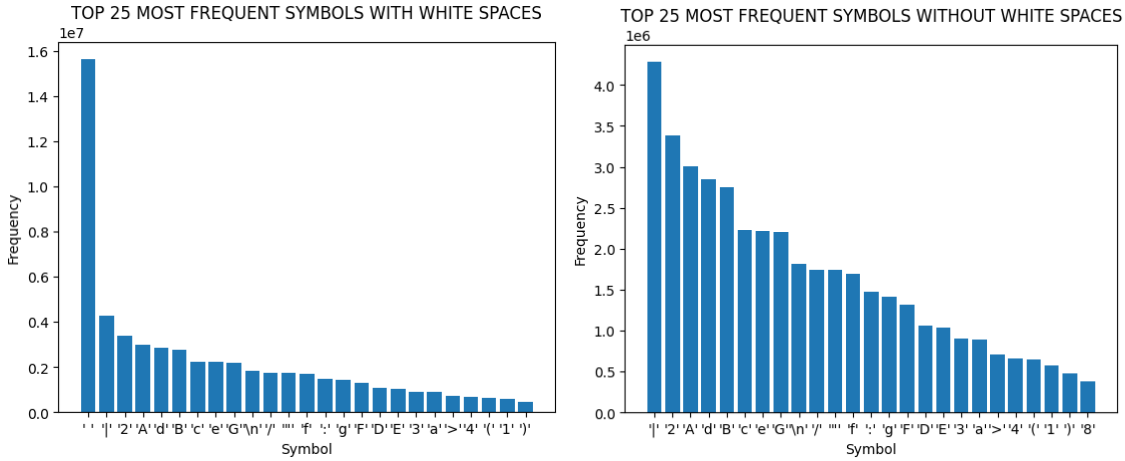

Figure 2: Amount of each character in the dataset. The left image shows the distribution before removing whitespace, while the right image has whitespaces omitted.

To prepare the data for the model we create a MusicDataset class that converts the data to tensors for pyTorch compatibility, and for each item in the dataset delivers a sequence of 128 characters of the concatenated data file, along with a target corresponding to the next character in the sequence. This essentially acts as a rolling window through the entire data file.

3. Network Architectures

As mentioned in the introduction, the territory of using an LSTM model for music generation has already been explored. However, the methodology used in the project is suboptimal as the model completely overfits the training data. Thus, prompting us to establish a baseline, by creating our own Long Short-Term Memory (LSTM) model (S. Hochreiter and J. Schmidhuber). This will be used as a benchmark against the Extended Long Short-Term Memory (xLSTM) model (Beck, M. et. al.). The structure of the networks can be found in the images below.

We created the LSTM model using an embedding of 128 dimensions. The reason for using embedding is to capture semantic relationships between our unique variables. The embedding enables the model to learn meaningful patterns in the data rather than treating the integers as arbitrary numbers. On top of this we built 4 LSTM blocks. While this is quite a lot of LSTM blocks, we deemed this necessary because of the scale of our problem. Lastly, we have two fully connected layers with a ReLU activation function between the layers to capture non-linear patterns. During initial training runs, the model would quickly start overfitting to the training set. Introducing dropout layers with 50% dropout probability in both our LSTM blocks and between the fully connected layers, tended to ease this trend and create a balance in the training and testing performance.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SimpleModel                              [1, 95]                   --
├─Embedding: 1-1                         [1, 128, 128]             12,160
├─LSTM: 1-2                              [1, 128, 256]             1,974,272
├─Dropout: 1-3                           [1, 256]                  --
├─Linear: 1-4                            [1, 256]                  65,792
├─Dropout: 1-5                           [1, 256]                  --
├─Linear: 1-6                            [1, 95]                   24,415
==========================================================================================
Total params: 2,076,639
Trainable params: 2,076,639
```

Figure 3: A general overview of the LSTM model architecture produced by summary from the Torchinfo library

However, LSTM is not a perfect model, and through the years it has received some criticism, which includes the inability to revise storage decisions, meaning it cannot fully re-evaluate or revise earlier storage decisions. Once it is forgotten, it is gone. Furthermore, LSTM has limited storage capabilities, which means when the threshold of the memory has been reached, new input will erase old input.

The xLSTM tries to mitigate these errors by introducing exponential gating with stabilization, which allows more nuanced control over the gates (input, output, and forget). It also adds a “stability” gate, which controls

scaling and avoids exploding or vanishing activations. xLSTM also introduces memory architectures, which are called sLSTM (Scalar Memory) and mLSTM (Matrix Memory). The sLSTM module works with scalars only and tries to reduce the complexity of memory updates, where each memory update is represented as a scalar value instead of a vector, leading to simpler computations. The mLSTM tries to increase storage capacities which is essentially done by increasing the LSTM memory cell from a scalar to a matrix. The retrieval is therefore performed via matrix multiplication.

Like the LSTM model, in our creation of the xLSTM model, we added the same embedding dimensions and two fully connected layers separated by a ReLU activation function. Both fully connected layers are preceded by a dropout layer, as seen in Figure 3. Utilizing the same underlying network structure where we solely modify the blocks in the respective networks grants a clearer comparison between the results of the LSTM and xLSTM model.

```
==========================================================================================
Layer (type:depth-idx)                                  Output Shape              Param #
==========================================================================================
SimpleModelWithxLSTM                                    [1, 95]                   --
├─Embedding: 1-1                                        [1, 128, 128]             12,160
├─xLSTMBlockStack: 1-2                                  [1, 128, 128]             --
│    └─ModuleList: 2-1                                  --                        --
│    │    └─mLSTMBlock: 3-1                             [1, 128, 128]             109,448
│    │    └─sLSTMBlock: 3-2                             [1, 128, 128]             108,032
│    │    └─mLSTMBlock: 3-3                             [1, 128, 128]             109,448
│    │    └─mLSTMBlock: 3-4                             [1, 128, 128]             109,448
│    └─LayerNorm: 2-2                                   [1, 128, 128]             128
├─Dropout: 1-3                                          [1, 128, 128]             --
├─Linear: 1-4                                           [1, 256]                  33,024
├─Dropout: 1-5                                          [1, 256]                  --
├─Linear: 1-6                                           [1, 95]                   24,415
==========================================================================================
Total params: 506,103
Trainable params: 506,103
```

Figure 4: A general overview of the xLSTM model architecture produced by summary from the Torchinfo library

4. Train strategy

We decided, for initial development of the model, that not all the data should be included when training the model. The development would have been slowed significantly if all of the data had been included, as training the model would take up to 4 days. The initial steps was to only include 1/250 of the dataset as we presumed the data was quite homogeneous, and the model would therefore scale nicely with only including small parts of the total training data. We settled on 15 epochs initially, as we quickly found out that more epochs would lead it to overfit. This led us to our next part of our training strategy, which included early stopping of the model. Early stopping was based on test loss, as optimizing the loss function aligns with the mathematical foundation of model training and ensures a fair comparison between models. However, in real-world applications, test accuracy often takes precedence, as it better reflects practical performance.

Using a grid search approach on *learning rate* and *dropout probability* for hyperparameter tuning, we found a learning rate of 0.001 and dropout probability of 70% to have the best performance.

Bibliography

Beck, M., Pöppel, K., Spanring, M., Auer, A., Prudnikova, O., Kopp, M., Klambauer, G., Brandstetter, J., & Hochreiter, S. (2024). xLSTM: Extended Long Short-Term Memory. ArXiv Preprint ArXiv:2405.04517. URL: https://arxiv.org/abs/2405.04517

S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," in *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 15 Nov. 1997, doi: 10.1162/neco.1997.9.8.1735. URL: https://ieeexplore.ieee.org/abstract/document/6795963

Wu, S., Li, X., Yu, F., & Sun, M. (2023). TunesFormer: Forming Irish Tunes with Control Codes by Bar Patching. In L. Porcaro, R. Batlle-Roca, & E. Gómez (Eds.), Proceedings of the 2nd Workshop on Human-Centric Music Information Retrieval 2023 co-located with the 24th International Society for Music Information Retrieval Conference (ISMIR 2023), Milan, Italy, November 10, 2023 (Vol. 3528). CEUR-WS.org. https://ceur-ws.org/Vol-3528/paper1.pdf