computerComposer

Intro:

I am training a machine learning model on the classical piano compositional works of Catherine Rollin with the goal of having it create music in her writing style. This research will be used in the presentation of the lecture *Human Imagination or AI?* at the **2024 Music Teachers National Association <https://www.mtna.org/>** conference.

Step 0: Setup

The important libraries in this project are tensorFlow, keras, and music21. Getting tensorFlow/keras to run on an M1 chip in python was a very tedious process. Only certain versions will work with the M1 and even then, some versions of tensorFlow will not work with other versions of python. It will save you 1 million headaches to use Python 3.8 and tensorFlow 2.13.

Finally, with the correct versions of everything you still need to setup a virtual environment using conda for everything to work properly on the M1. **This video < https://www.youtube.com/watch?v=WFIZn6titnc>** will help navigate that task.

I began with a foundation of code based on **this github < https://github.com/Skuldur/Classical-Piano-Composer>** . However, in my experience Skuldur's model could not learn without tweaks. The loss function does not decrease, accuracy does not rise and thus, the model it makes will predict the same note over and over. So, I had to play with changing the layers/dropout/batch normalization/etc. **This blog < https://softologyblog.wordpress.com/2019/09/22/long-short-term-memory-music-composer/>** helped me understand what to tweak and Softology adds matlab plotting which makes everything a lot more readable. I still did things differently in setting up my model, but these sources were super useful for getting started. More details on my methods below!

Step 1: Organize the data

There were 22 pieces already written in Finale, which can then be easily exported to midi. I transposed these pieces all to the key of C major or A minor (depending on if they were major/minor to start with). This seemed like a logical organizational step so that the model learns based on broader musical characteristics than key signature. The first runs just involved those pieces, but it

became clear that overfitting would be an issue without a larger data set.

I then obtained 187 PDFs of other pieces by Catherine Rollin courtesy of Alfred Publishing. Using PlayScore2, I translated those into midi and then transposed them. With 209 examples in the training data set I began running tests.

Step 2: Structuring the Model

My initial goal was to simply get the model to learn. This involved removing the batch normalization layers, adding a dropout layer and removing an activation layer (some of these might get added back in later). The "lstm" and "predict" codes in **this github < https://github.com/skrinsky/melody>** are the first versions of the model that would appear to learn. One thing to note is that the Skuldur base code is considering only one parameter option for each instance in the sequence: note/chord. This means that the model does not consider velocity, rhythm, meter, ADSR, resonance, portamento, etc. Later into testing I will try adding parameters. Another thing to note is that Skuldur's model takes training data 100-note sequences at a time. This means that longer musical works will be broken into *more* training sequences, and therefore the longer pieces will have an outsized effect on the training data. First I began experimenting with different batch sizes, using a window of 200 epochs as a starting point. Here are the results for 64, 128, and 256 batches:

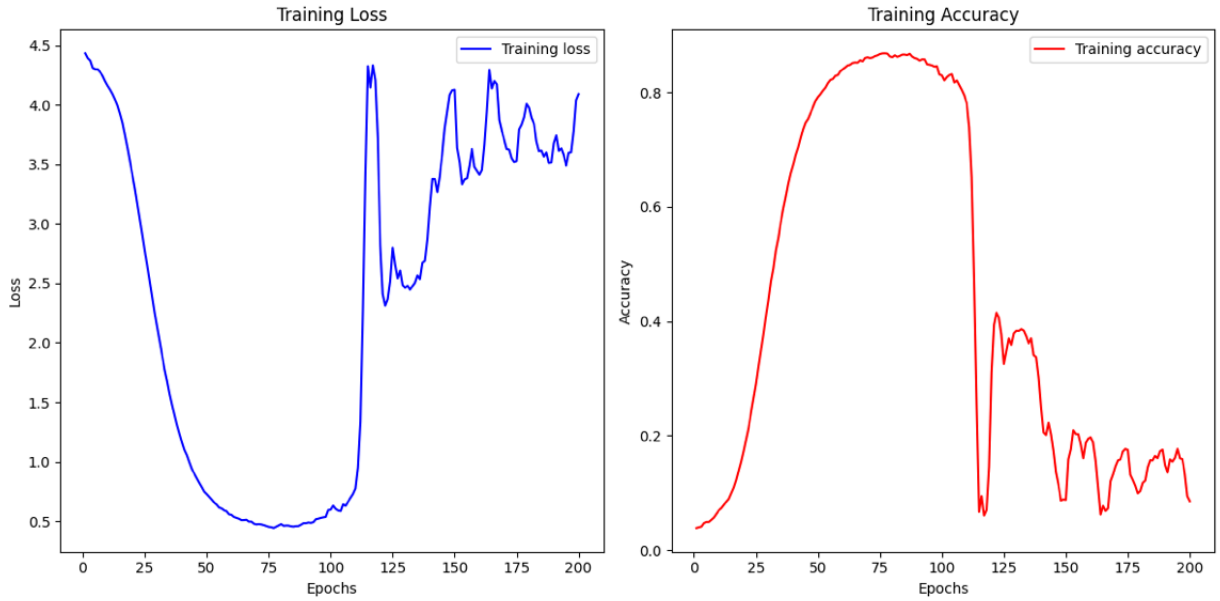

Fig1: 1 Parameter (Notes/Chords), 64 Batch, 200 Epoch *convergence around Epoch 76 (trained to 77) with 86% accuracy*

Listen to three examples of this model **HERE. <https://soundcloud.com/summer-554845429/sets/1lstm64batch/s-sax4UqLT74k?si=183db0bf16254a85bd6741d6770b32e7&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>**

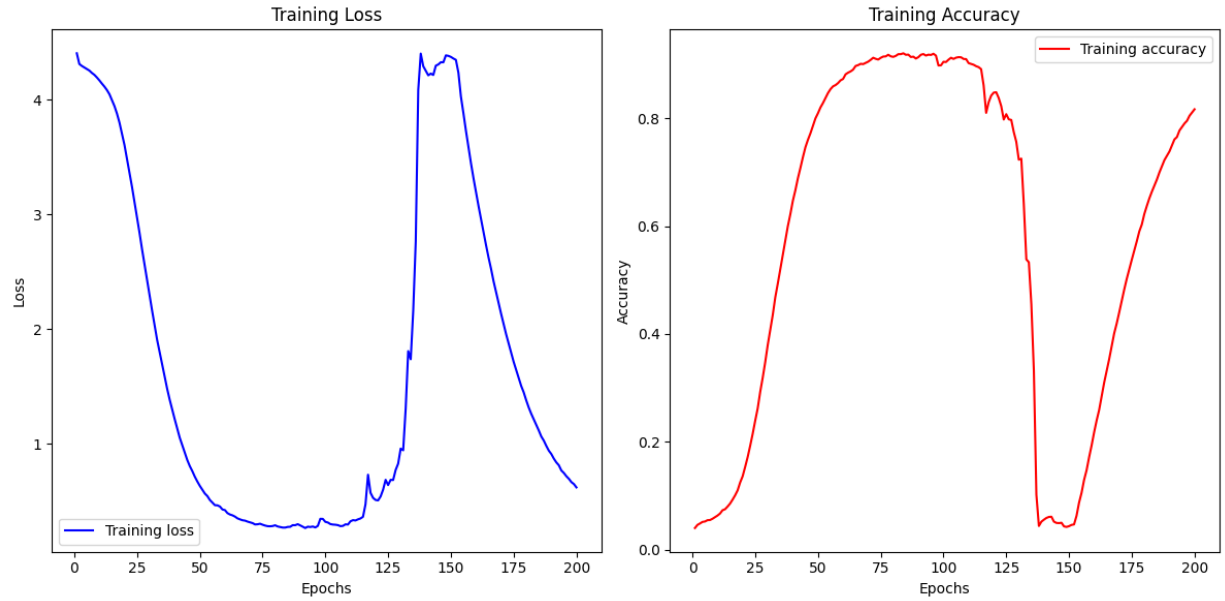

Fig2: 1 Parameter (Notes/Chords), 128 Batch, 200 Epoch *convergence around Epoch 84 (trained to 92) with 92% accuracy*

Listen to three examples of this model **HERE <https://soundcloud.com/summer-554845429/sets/1lstm_128batch/s-ED06iMkYWvx?si=183db0bf16254a85bd6741d6770b32e7&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>** .

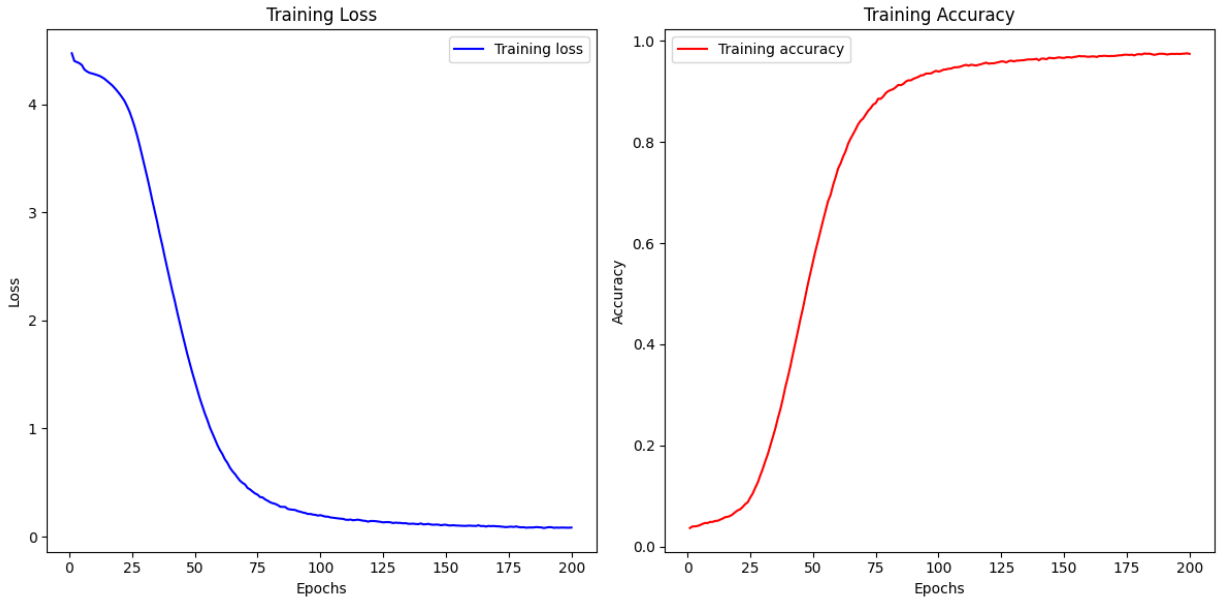

Fig3: 1 Parameter (Notes/Chords), 256 Batch, 200 Epoch *convergence around Epoch 156 (trained to 189) with 97% accuracy*

Listen to three examples of this model **HERE <https://soundcloud.com/summer-554845429/sets/1lstm_256batch/s-AXuxfJ9YD4W?si=183db0bf16254a85bd6741d6770b32e7&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>** .

I want to use a model with the epoch count corresponding to when the graph first hits its limit. This is what I am called the epoch of "convergence" (as opposed to when it finished training.)

Step 3: Adding a parameter

Adding velocity seems like the most straightforward way to squeeze a little more "musicality" out of the machine. If the model is learning correctly, this parameter should result in us

beginning to hear “phrasing” which is the method by which a musician shapes a sequence of notes in a passage with expressive articulation.

The model currently intakes instances of a note as note+octave or instances of an interval/chord as note+octave.note+octave. This means a middle C would be C4 and a major chord built on it would be C4.E4.G4. So to add velocity I decided to separate it by an underscore. This would look like C4_60 or C6_60.E4_80.G4_70.

Upon first run the loss function did not decrease at all. Investigating further, I realized that it is creating possible outcomes for every single velocity+note combination. This means 128 possible notes (not even including chords) combining with 128 possible velocities. Big number! Too big for this little GPU I fear, so I experimented with quantizing the velocities of the training data to the nearest 20. This means there are 6 possible velocities (20,40,60,80,100,120). This code can be found under **“lstmTake2” and “predictTake2” < https://github.com/skrinsky/melody>** .

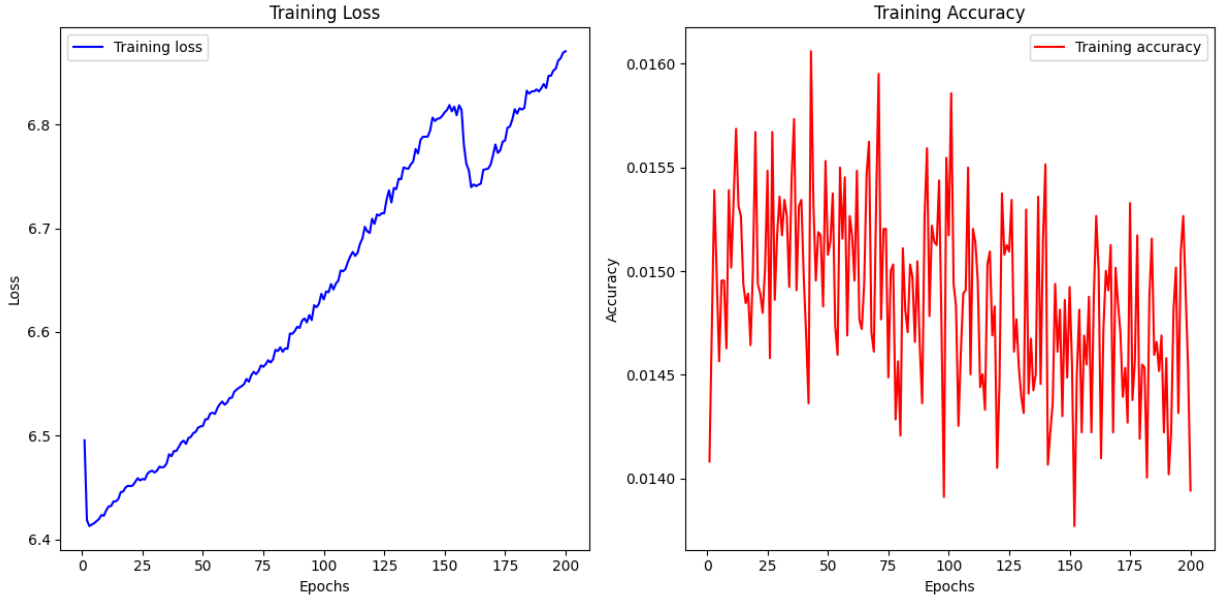

Fig4: 2 Parameters (note+velocity) 64Batch 200Epoch Quant20 *never converged, **unusable***

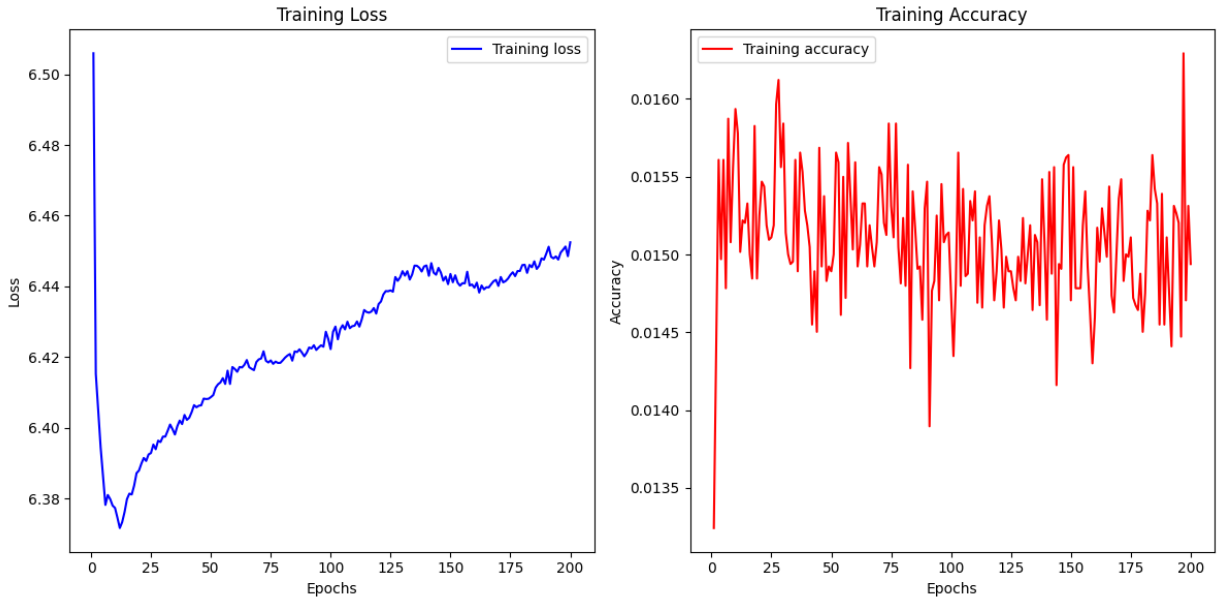

Fig5: 2 Parameters (note+velocity) 128Batch 200Epoch Quant20 *never converged, **unusable***

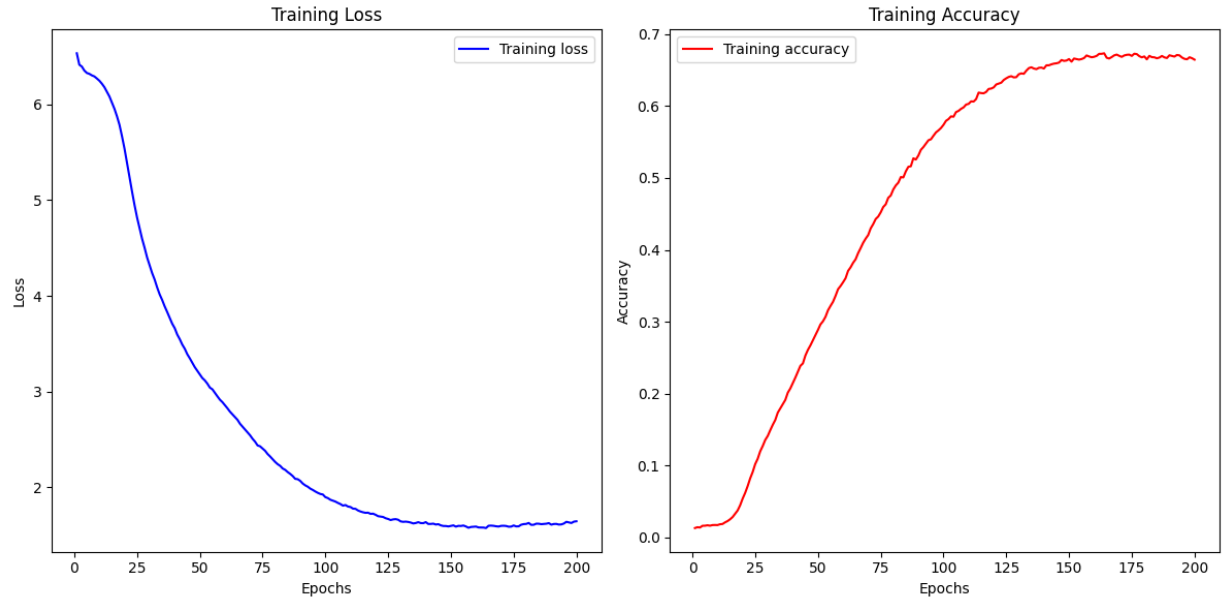

Fig6: 2 Parameters (note+velocity) 256Batch 200Epoch Quant20
convergence around Epoch 157 (trained to 164) with 67% accuracy

Listen to three examples of this model **HERE. <https://soundcloud.com/summer-554845429/sets/2lstm_256batch/s-wVqmKpToTUD?si=28d2067644204faba9398c4c4d2efe65&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>**

The first two batch sizes still don't work, but we begin to see convergence at the 256 Batch! Now this model only has 67% accuracy, before velocity was added the 256 Batch model had 97% accuracy. To try to increase accuracy I run a version quantizing velocity to the nearest 40, so 3 possible velocities (40, 80, 120). This brings accuracy up to 75%, however in my opinion it seems like the output of the model is noticeably less musical with only three shades of volume. Example 2 in this set is especially

repetitive too.

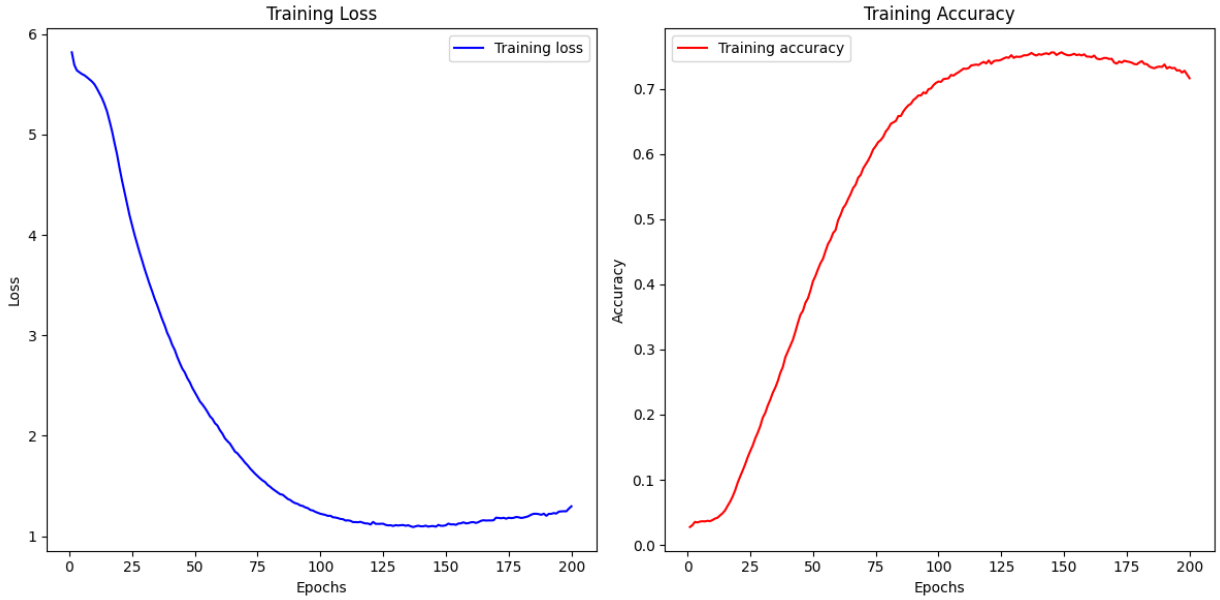

Fig7: 2 Parameters (note+velocity) 256Batch 200Epoch Quant40
convergence around Epoch 129 (trained to 137) with 75% accuracy

Listen to three examples of this model **HERE. <https://soundcloud.com/summer-554845429/sets/2lstm-256batch-quant40/s-10hwQknI9Zq?si=0e7ee83c4f8f4e49b798c033958589c3&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>**

Step 3: Experimenting with batch normalization

Next I try adding several batch normalization layers back in but in different places than they were originally located. The model appears to train faster and at a higher accuracy but the output is definitely the most meandering and I wonder if it has become

too generalized. Then I tried this with both the 40 and 20 velocity quantizations.

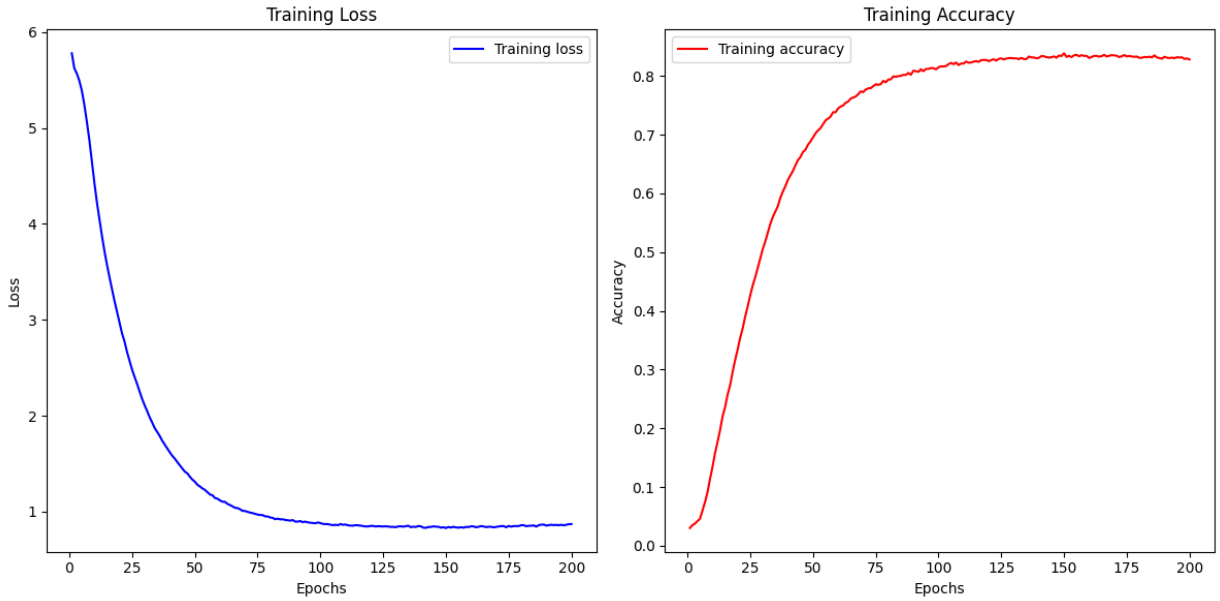

Fig8: 2 Parameters (note+velocity) 256Batch 200Epoch Quant40 added 2 Layers of Batch Normalization *convergence around Epoch 141 (trained to 150) with 83% accuracy*

Listen to three examples of this model **HERE <https://soundcloud.com/summer-554845429/sets/2lstm-add2batchnormlayers-256batch-quant40/s-4PVbgoMgAlw?si=0e7ee83c4f8f4e49b798c033958589c3&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>**.

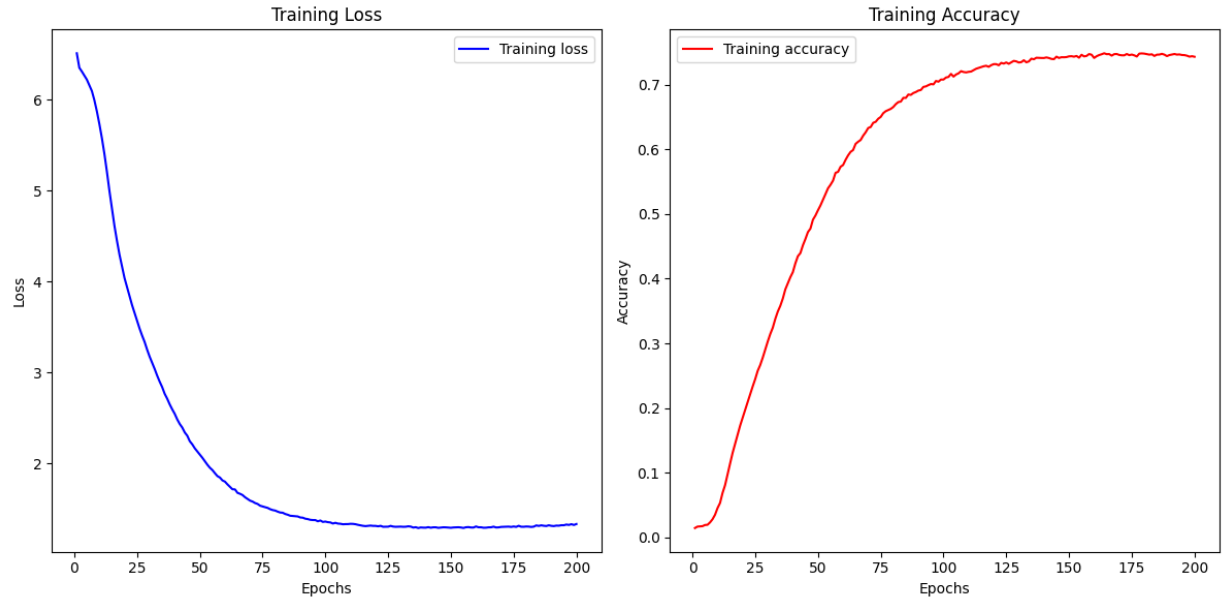

Fig9: 2 Parameters (note+velocity) 256Batch 200Epoch Quant20 2 Layers of Batch Normalization *convergence around Epoch 135 (trained to 137) with 74% accuracy*

Listen to three examples of this model **HERE. <https://soundcloud.com/summer-554845429/sets/2lstm-add2batchnormlayers-256batch-quant20/s-czXpeicB3Ve?si=bccf7d8653594066931328 7aae327930&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>**

Batch normalization is taking training times down but creating over-generalized results. I try decreasing the amount of normalization happening on those layers with this line, setting the momentum (originally at .9) lower:

model.add(BatchNormalization(momentum=0.5, epsilon=1e-5))

I want to see if this version of the model is able to train with 128

Batch sizes (which did not work before in Fig 5). It also does not converge:

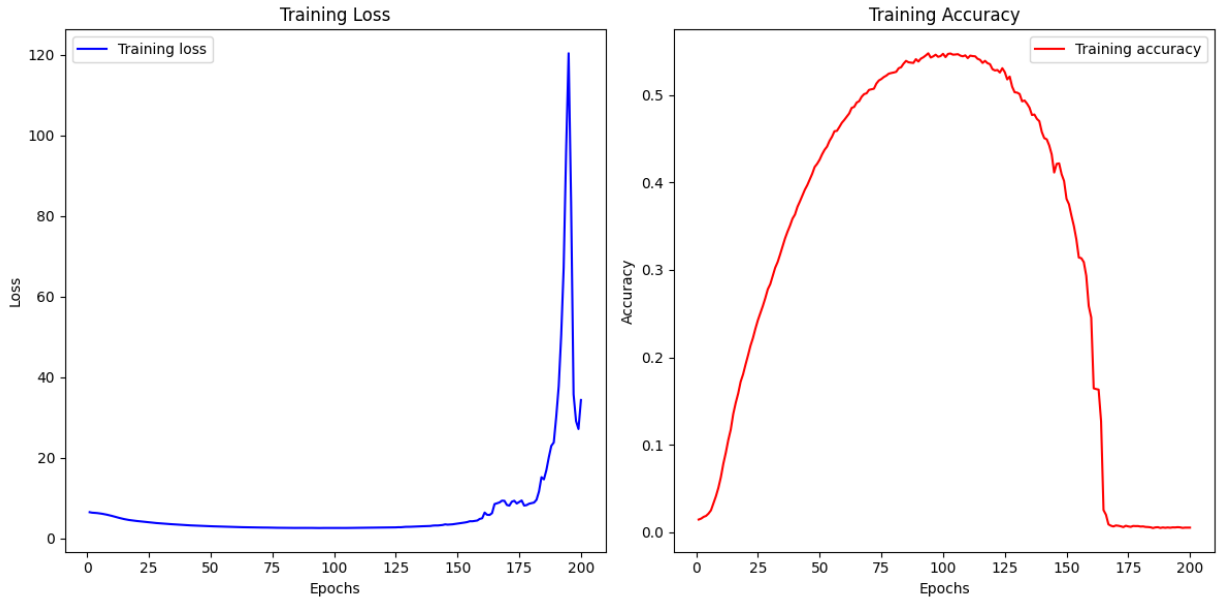

Fig10: 2 Parameters (note+velocity) 128Batch 200Epoch Quant20 2 Layers of Batch Normalization *never converged, **unusable***

Step 4: Experimenting with length of sequence

I tried reducing the length of input sequences from 100 to 50. This decreased training time and worked for 128 and 256 batch sizes. Also, I had to comment out the batch normalization for it to train. Compare 128 batch size in Fig 11 to Fig 5 and 256 batch size in Fig 12 to Fig 6 to see the difference in sequence length (all other aspects of these two models are identical). The 128 batch seems to be way too overgeneralized, it is probably the least musical in terms of note choice of any of the models, the 256 batch is a little better but is still pretty meandaring.

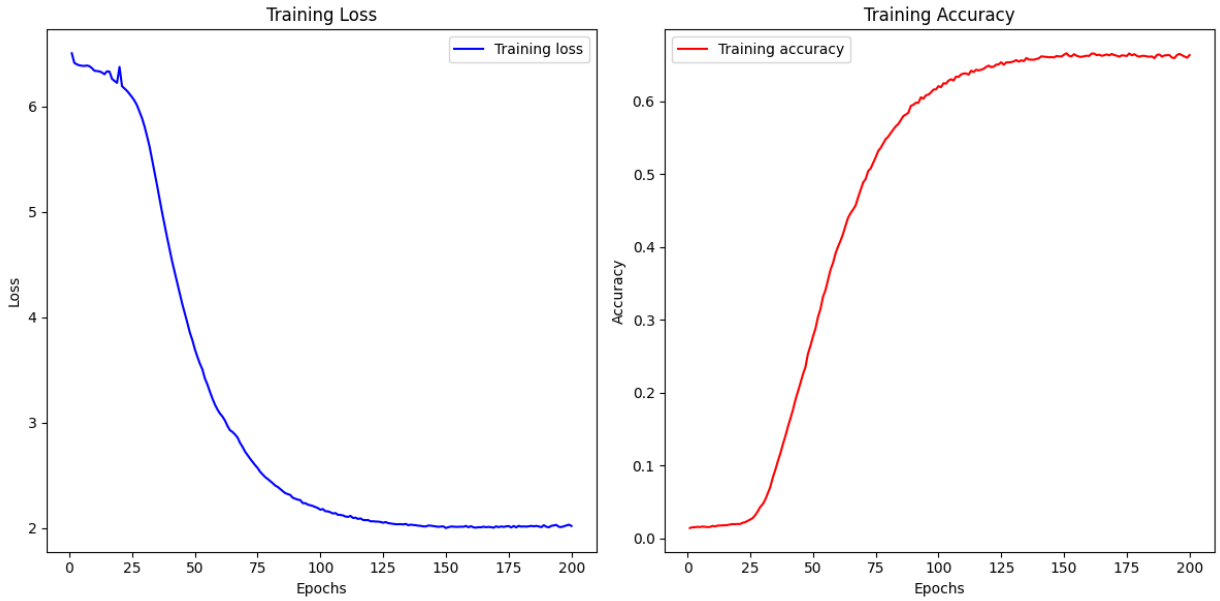

Fig11: 2 Parameters (note+velocity) 128 Batch 200 Epoch Quant20 50SeqLength *convergence around 141 (trained to 150) with 66% accuracy*

Listen to three examples of this model **HERE. <https://soundcloud.com/summer-554845429/sets/2lstm-128batch-20quant-50seq/s-Oup9pdP9SxA?si=5d1d90f89e9d41acb046ce6977cbac69&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>**

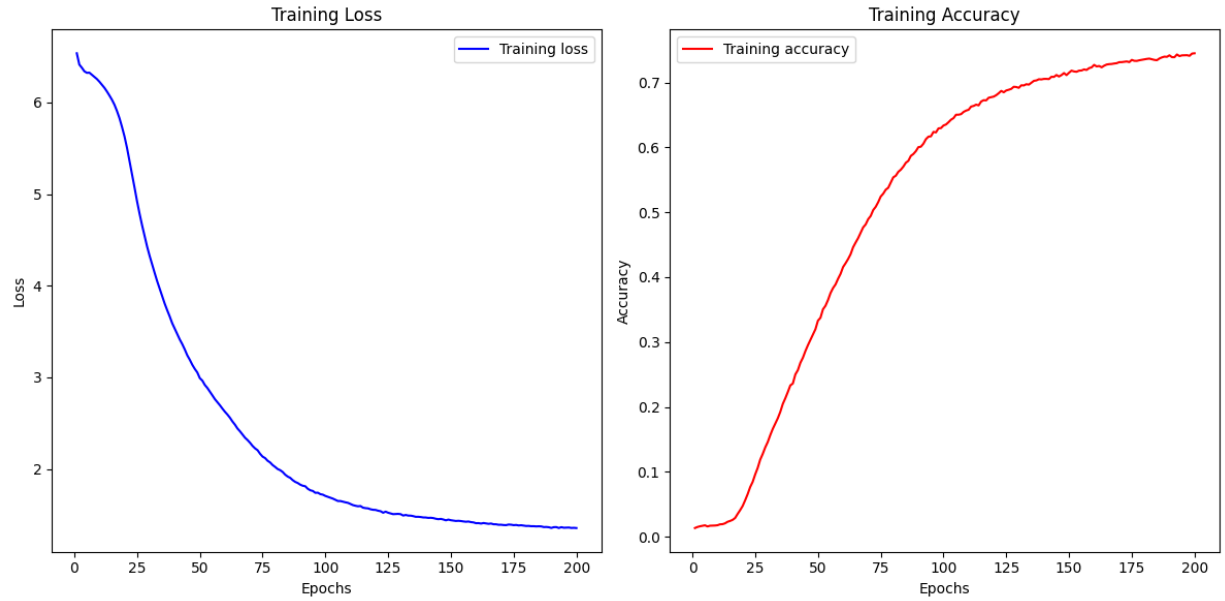

Fig12: 2 Parameters (note+velocity) 256Batch 200Epoch Quant20 50SeqLength *convergence around 190 (trained to 200) with 74% accuracy*

Listen to three examples of this model **HERE. <https://soundcloud.com/summer-554845429/sets/2lstm-256batch-20quant-50seq/s-3sKKqYT54aN?si=5d1d90f89e9d41acb046ce6977cbac69&utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing>**

****Interesting thoughts I am having*****

How could we use the thinking style of mid-side processing and apply it to compositional decision making? Say we added 209 pieces by Rachmaninov to the training data, we would get:

Rollin + Rachmaninov = stronger signal of compositional similarities

In the mid-side analogy, this is the “center” channel. The interesting question becomes, how would we get the side channel? How can someone phase flip compositional decisions? We would need an “inverted_Rollin” signal. For every note or chord event in each of the Rollin midi pieces, there is a set of notes that were not chosen. For each of these instances we would load all of the choices that were not made. So if a chord event was C3.E3.G3 then inverted_Rollin would be every single note possibility but those three.

inverted_Rollin + Rachmaninov = stronger signal of compositional differences

This would sound chaotic, like free jazz. However, what makes mid-side processing useful is that there can be variable amounts of side summed with the center. So, we could take 10 pieces created by the model that reinforces compositional differences and add it to the training data of Rollin + Rachmaninov model. Additionally, we could effectively create a Rollin – Rachmaninov model by adding Rollin + (inverted_Rollin + Rachmaninov). Could this lead to interesting outcomes?