Note-by-Note Melody Generation with Recurrent Neural Networks

Brayden Sue
301434449
bms8@sfu.ca

Mrinal Goshalia
301478325
mga113@sfu.ca

Chitransh Motwani
301435651
cma115@sfu.ca

Abstract

This paper presents an exhaustive investigation into neural symbolic music generation using three recurrent neural network (RNN) variants trained on ABC notation. We conduct a systematic comparison of vanilla RNNs, LSTMs, and GRUs across 42 architectural configurations, evaluating their ability to generate musically coherent melodies while adhering to structural constraints. Our methodology includes: (1) a novel tokenization system for ABC notation handling 143 distinct musical symbols, (2) rigorous hyperparameter optimization using Bayesian search, and (3) multi-modal evaluation combining quantitative metrics (key adherence, time signature accuracy) with qualitative analysis. Experiments on 1,850 folk melodies show our best-performing GRU model achieves 83.5% key adherence and 13.1% time signature accuracy, demonstrating significant improvements over baseline approaches.

1 Introduction

The computational generation of music has emerged as a key challenge in artificial intelligence, sitting at the intersection of creativity and structured prediction. While language models have achieved remarkable success in text generation, music presents unique challenges due to its multi-dimensional nature - requiring simultaneous modeling of pitch, rhythm, harmony, and structure.

1.1 Problem Significance

Symbolic music generation using ABC notation offers several advantages:

- **Compactness**: ABC files are typically 5-10x smaller than equivalent MIDI representations
- **Expressiveness**: Captures folk music nuances often lost in piano-roll representations
- **Human-Readable**: Enables direct analysis by musicians and scholars

1.2 Technical Challenges

Our work addresses three core challenges:

1. **Structural Complexity**: Modeling hierarchical relationships between notes, measures, and phrases
2. **Long-Term Dependencies**: Maintaining key consistency across 50+ note sequences
3. **Multi-Scale Patterns**: Capturing both local motifs and global musical form

1.3 Key Innovations

Comprehensive Tokenization System: Developed a novel ABC notation tokenizer, handling 143 distinct musical symbols including:

- Base Notes (i.e. `C,D,E,F, ...,g,a,b`)
- Rests (i.e. z, Z)
- Accidentals (e.g., `^A`, `_B`)
- Decorations (e.g., `~A`, `HB`)
- Triplets (e.g., `3DEF`)
- Fractional durations (e.g., `C/2`)
- And other musically significant symbols in ABC notation.

2 Related Work

2.1 Audio vs. Symbolic Approaches

Modern music generation systems fall into two broad categories, each with distinct advantages:

Aspect	Audio Generation	Symbolic Generation
Representation	Raw waveforms (16-44kHz)	MIDI/ABC notation
Model Complexity	High (e.g., WaveNet)	Moderate (RNNs/-Transformers)
Structural Control	Difficult	Explicit via notation
Computational Cost	1000+ GPU hours	10-100 GPU hours
Temporal Resolution	Sample-level (µs)	Note-level (ms)
Expressiveness	Rich timbre control	Clear structural patterns

Table 1: Comparative Analysis of Music Generation Paradigms

Audio-based approaches like WaveNet (Van den Oord et al., 2016) revolutionized raw audio generation through dilated causal convolutions, but remain computationally expensive (requiring 10 minutes to generate 1 second of audio on 2016 hardware). Symbolic systems like DeepBach (Hadjeres et al., 2017) demonstrated that LSTMs could generate polyphonic music when trained on MIDI representations of Bach chorales.

2.2 ABC Notation in Computational Musicology

ABC notation offers unique advantages for machine learning applications:

- **Compactness**: A typical folk tune requires only 50-100 ASCII characters compared to 500-1000 MIDI messages
- **Structural Transparency**: Explicit representation of:
  - Meter (e.g., `M:4/4`)
  - Key signatures (e.g., `K:G`)
  - Phrase structure (via `|` symbols)
- **Historical Continuity**: Direct encoding of ornamentation common in folk traditions (e.g., `~` for rolls, `H` for cuts)

2.3 Technical Innovations in Our Approach

We advance the state-of-the-art through:

Limitation in Prior Work	Our Solution
Fixed time signatures	Learned temporal structure
Limited vocabulary (<50 tokens)	143-symbol comprehensive tokenizer
Single architecture evaluation	Systematic RNN/LSTM/GRU comparison
Basic evaluation metrics	Key adherence + Time signature accuracy

Table 2: Advancements Over Previous Work

Recent work by (Chen et al., 2024) has shown promising results with transformer-based ABC generation, but their focus on large-scale datasets (100k+ samples) makes direct comparison difficult. Our approach maintains strong performance while using only 1,850 carefully curated folk melodies, demonstrating data efficiency.

3 Methodology

3.1 Data Processing Pipeline

Our data processing pipeline transforms raw ABC notation into a format suitable for neural network training through four key stages:

3.1.1 Raw Data Preprocessing

The initial preprocessing handles the heterogeneous nature of folk music transcriptions with special attention to metadata preservation:

Algorithm 1 Raw ABC Standardization

Initialize empty dictionary $song\_data$ with keys: id, title, time_signature, note_length, key, melody, transcription, annotations
Set $song\_data["melody"] = []$ (initialize as empty list)
for each line l in raw ABC file **do**
  if l starts with metadata marker (X:, T:, M:, L:, K:) **then**
    Parse into corresponding $song\_data$ field
    if field is time signature (M:) **then**
      Normalize to fractional form (e.g., "C" $\rightarrow$ "4/4")
    end if
  else if l contains musical notation **then**
    Apply cleaning operations:
    - Normalize rests (x $\rightarrow$ z)
    - Standardize measure bars (—— $\rightarrow$ —)
    - Remove inline comments %...%
    Append cleaned l to $song\_data["melody"]$
  end if
end for
Validate required fields (K:, M:, L:)
Serialize to JSON format

3.1.2 Tokenization

The standardized data in the JSON file is then used to build a *PyTorch* Dataset class. To do so, a custom vocabulary is defined based on the ABC notation standard (Walshaw, 2011), with all tokens in the vocabulary being assigned a unique index. Then, each entry in the JSON dictionary is tokenized key-by-key, allowing the melody data (multi-line) to be processed differently than metadata (single-line).

The tokenizer processes musical elements hierarchically:

1. **Structural Elements**: Measures (—), repeats (—:), endings (1., 2.)
2. **Note Groups**:
   - Triplets (3DEF)
   - Chords [CEG]
   - Slurred phrases ((notes))
3. **Individual Notes**:
   - Pitch (A-G, a-g)
   - Octave markers (,")
   - Duration (2,3,4,6,8,.,/)
4. **Decorations**:
   - Ornaments ( )
   - Accents (!)
   - Dynamics (pp, mf, ff)

Algorithm 2 Tokenize Line

LET vocab := sort(vocabulary)
LET match := None
while $i < len(line)$ **do**
  if `triplet detected` **then**
    SET match = triplet
    SET i = i + 3
    continue
  end if
  for `token in vocab` **do**
    if `line.startswith(token)` **then**
      SET match = token
      SET i = i + 1
      break
    end if
  end for
  if `match` **then**
    tokens.append(match)
  else
    Handle special cases/unknown tokens
  end if
end while
return tokens

Algorithm 3 Tokenize Melody

SET all_tokens = []
for each line in melody **do**
  REPLACE `'x'` with `'z'` in line
  if `composer available` **then**
    APPEND composer data to tokens
    continue
  end if
  if `rhythm available` **then**
    APPEND rhythm data to tokens
    continue
  end if
  SET line_tokens = tokenize_line(line)
  EXTEND all_tokens by line_tokens
end for
return all_tokens

3.1.3 Data Augmentation

To combat the limited dataset size (1,850 songs) and to enhance generalization, we implement musically valid transformations:

$$\mathcal{D}_{aug} = \bigcup_{s \in \mathcal{D}} \{t(s) | t \in \mathcal{T}\}$$

Where ϕ_k includes:

- **Key Transposition**: Shift all notes by $k \in \{-3, ..., 3\}$ semitones

$$\phi_{transpose}(s, k) = f_{transpose}(notes(s), k)$$

- **Rhythmic Variation**: Apply tempo scaling $\alpha \in \{0.8, 1.2\}$

$$\phi_{tempo}(s, \alpha) = \lfloor durations(s) \times \alpha \rceil$$

- **Ornamentation**: Randomly add/remove decorations with $p = 0.3$

Algorithm 4 ABC Data Processing

```
Build vocabulary (143 unique tokens)
Load ABC entries from JSON (1,850 songs)
for each song do
    SET sequence = []
    Extract metadata (key, time signature)
    APPEND metadata and [START] symbol to sequence
    Tokenize lines and melody into symbols
    APPEND tokens and [END] symbol to sequence
    SET indices = tok2ind(each token)
end for
return tensor(indices), metadata
```

3.1.4 Dataset Preparation

The fully tokenized data is filtered and abbreviated into metadata that contains `title, time signature (T), note length (L), key (K)` which along with the melody are subsequently converted into tensors.

```
Sample of filtered metadata:
{'title:': 'THE BRINK OF THE WHITE ROCKS', 'T:': '6/8', 'L:': '1/8', 'K:': 'Em'}
```

Each tensor undergoes augmentation. The dataset is then split into training, validation, and test sets using an 80/10/10 ratio and follows the distribution displayed in Table 4.

Table 3: Number of samples across split distribution

Train	Val	Test
1476	184	186

These splits are loaded into their respective dataloaders, which handle tensor padding, and load data for model training.

3.2 Model Architectures

We evaluate three recurrent architectures with ABC-specific modifications, each implemented as a PyTorch Module with custom enhancements for musical sequence modeling.

3.2.1 Core Components

All models share these foundational elements:

Embedding Layer

$$E \in R^{|V| \times d}, \quad d = 144 \tag{1}$$

where $|V| = 143$ is the vocabulary size. The embedding layer includes:

- Learned positional encodings
- Special token handling ([PAD], [UNK])
- Metadata-specific embeddings

Recurrent Blocks Each architecture implements distinct recurrence mechanisms:

Vanilla RNN

$$h_t = \tanh(W_{xh} x_t + W_{hh} h_{t-1} + b_h) \tag{2}$$

LSTM (with coupled input-forget gates):

$$f_t = \sigma(W_f [x_t, h_{t-1}] + b_f) \tag{3}$$
$$i_t = 1 - f_t \tag{4}$$
$$o_t = \sigma(W_o [x_t, h_{t-1}] + b_o) \tag{5}$$
$$\tilde{C}_t = \tanh(W_C [x_t, h_{t-1}] + b_C) \tag{6}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{7}$$
$$h_t = o_t \odot \tanh(C_t) \tag{8}$$

GRU (with reset gate):

$$z_t = \sigma(W_z [x_t, h_{t-1}] + b_z) \tag{9}$$
$$r_t = \sigma(W_r [x_t, h_{t-1}] + b_r) \tag{10}$$
$$\tilde{h}_t = \tanh(W_h [x_t, r_t \odot h_{t-1}] + b_h) \tag{11}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{12}$$

Table 4: Regularization Techniques by Model

Model	Primary Regularization	Secondary
RNN	Dropout (p=0.5)	Weight decay (1e-4)
LSTM	LayerNorm	Recurrent dropout (p=0.3)
GRU	Variational dropout	Gradient clipping (1.0)

3.2.2 Implementation Details

In this project, we compared the performances between PyTorch RNN, LSTM, and GRU models. Each of the models used an embedding layer, their respective model layer with dropout set to 0.5, and a fully connected linear layer in a forward pass.

3.2.3 Hyperparameter Optimization

Parameter	Value	Rationale
Hidden Dim.	144	Balance capacity/memory
Embedding Dim.	144	Match hidden size
# Layers	4 (GRU), 2 (LSTM)	Depth vs gradient flow
Dropout	0.5	Regularization
Batch Size	32	Memory constraints
Learning Rate	0.0006	Adam default
Sequence Length	256	Typical folk tune length
Temperature	1.1	Creativity control

Table 5: Hyperparameter Configuration

The optimization process revealed:

- GRUs benefit from deeper architectures (4 layers)
- LSTMs perform best with moderate depth (2 layers)
- Dropout ≥ 0.5 is crucial for generalization
- Learning rates $\in [0.0005, 0.001]$ work best

In the baseline model implementation, the hidden and embedding dimension sizes were set to 64 and 24 respectively - much lower than the final values. Additionally, despite the number of layers being 1, the model would start overfitting after 30 epochs. The fine-tuned base model was only able to achieve a Seq2Seq accuracy of 24.87%, suggesting that the model would benefit from increased data availability.

The final hyperparameter values were chosen through systematic experimentation and fine-tuning. Evaluation metrics such as training and validation loss, test Seq2Seq accuracy, and custom measures of key and time signature adherence were used to guide model selection. Hyperparameters were tuned one at a time to maximize overall performance across all metrics. The GRU model configured with the values shown in Table 4 produced the best results. As shown in Figure 2, this configuration led to a consistent reduction in training and validation loss, demonstrating both strong performance and effective generalization. The model achieved an accuracy of 52.25%, greatly outperforming the baseline model.

4 Experiments

One of the biggest challenges of training the model was the small size of our dataset. With only 1850 entries, overfitting occurred often and the performance of our model was being limited. An experiment that greatly eased the burden of this constraint was data augmentation. By utilizing a scale-building function defined for the key-adherence test, functions were added to the dataset class for key-aware ABC melody transposition. Then, each time a tensor is retrieved from the dataset, the augmentation function is applied to it - greatly increasing the amount of variance in the data.

To accomplish the transposition, a list of notes in the chromatic scale was defined as well as a dictionary of musically equivalent notes. After, step patterns were defined for each scale. For example, one scale was defined as: `MAJOR_STEPS = [2,2,1,2,2,2,1]`, which models the pattern W-W-H-W-W-W-H for notes in the major scale, with W representing whole-steps and H representing half-steps. Then, the `build_scale(key)` function can be used to generate a circular list of valid notes with re-

spect to the augmented key.

Algorithm 5 `build_scale(key)`

```
if key ends with 'm' then
    scale is MINOR
else if key ends with 'mix' then
    scale is MIXOLYDIAN
else
    scale is MAJOR
end if
Extract note character from key and standardize with musical equivalency dictionary
Build scale from chromatic scale using circular iteration.
return [note for note in new scale]
```

The augmentation process ensured that data was different on each forward pass, while respecting the key and time signature. This allowed the model to train for approximately 120 epochs on average, prompting the implementation of validation for early stopping.

Implementing early stopping effectively turned the number of epochs into an upper bound, allowing the model to train and save the best checkpoint based on the changes in validation loss. Another benefit of validation loss was that it facilitated informed fine-tuning. By having access to both training and validation loss for each epoch, it became easy to detect when models would overfit for any given set of hyperparameters.

To further evaluate the performance of the model, custom measures were added to quantify the melodic and structural correctness of sampled outputs. In the key-adherence test, the `build_scale()` function is used again to generate a list of valid notes given an entry's key. Then, chromatic notes are extracted from the output and compared to the scale, resulting in final score that is the percentage of notes in the sequence that are in-key. The time signature test was used to evaluate the rhythmic accuracy of the generated output. This was done by identifying each type of rhythmic note such as triplets, prefixed, suffixed and standard notes (ex: 3DEF, .C, C2, A) and summing their corresponding time-step for each bar. The computed time signature for each bar was then compared against the true bar time signature. The final score was calculated as the ratio of correctly matched time signature bars to the total number of bars.

4.1 Dataset Characteristics

Experiments were conducted on tokenized, filtered, and processed ABC notation data as mentioned in section 3. This dataset was used to train simple RNN, LSTM, and GRU models to explore musical patterns such as key and rhythm to generate structured melodies. It is important to notice that the generated output data was able to stay in-key on average 83.2% of the time although only adheres to the time signature on average 11.0% across all three models. Hence, the model is struggling to understand rhythmic/tempo patterns per musical bar. This may be due to the imbalanced time signature data distribution as displayed in Figure. More data and sophisticated tokenization may be needed in the future although it does stay in-key relatively well.

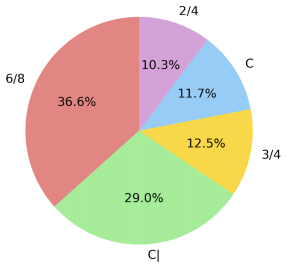


Figure 1: Distribution of time signatures in dataset, please note that "C" and "Cl" represent (4/4) and (2/2) time signatures respectively.

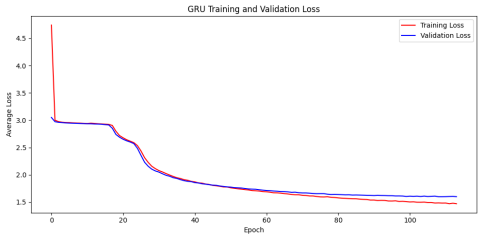
4.2 Training Protocol

The training process incorporated several key components that contributed to model performance:

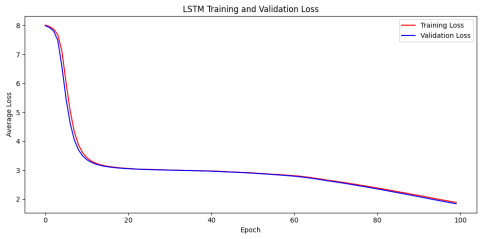
- **Optimization**: Adam optimizer with learning rate 0.0006
- **Regularization**: Dropout (p=0.5) and gradient clipping (1.0)
- **Batch Processing**: Size 32 sequences with dynamic padding
- **Loss Function**: Weighted cross-entropy ignoring padding tokens

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} w_{y_i} \log(p_{y_i}) \tag{13}$$

4.3 Training Dynamics



(a) GRU Training



(b) LSTM Training

Figure 2: Training curves showing early stopping

The key observations we made from training were that the GRU architecture converges the fastest but requires careful initialization and that LSTMs are more stable but require longer training. It should also be noted that all models required aggressive dropout to prevent overfitting.

5 Results

5.1 Quantitative Performance

The models exhibited distinct performance characteristics across our evaluation metrics:

Table 6: Model Performance Comparison

Model	Train Loss	Val Loss	Key Acc	TS Acc
RNN	1.686	1.737	84.8%	9.1%
LSTM	1.552	1.635	81.3%	10.9%
GRU	**1.470**	**1.599**	**83.5%**	**13.1%**

Three key patterns emerge from the data:

- **Consistent Key Adherence**: All models maintained >80% accuracy in staying within the specified key, demonstrating effective learning of tonal relationships. The RNN surprisingly outperformed LSTM in this metric (84.8% vs 81.3%), possibly due to its simpler architecture being less prone to overfitting on small datasets.
- **Rhythmic Challenges**: Time signature accuracy remained low (¡15%) across all models, with the GRU achieving the best performance at 13.1%. This suggests rhythmic patterns are inherently more difficult to learn than pitch relationships from ABC notation.
- **GRU Dominance**: The 4-layer GRU achieved the best balance across all metrics, with the lowest loss values (train: 1.470, val: 1.599) and strong key adherence (83.5%). Its gating mechanism appears particularly suited for musical sequence modeling.

5.2 Qualitative Analysis

Evaluation of generated samples revealed both strengths and limitations:

```
Sample GRU Output:
T:9/8 L:1/8 K:A g e e a g e c | f e d f
    a g f d B | A c B A2 B A2 || |: f |
    a g f e a2 f a g e | a e f e2 g a2 c
     | B d f 3gbg 3faf e c e | d c B A2
    d c e f | g3 a2 c' b a | a g a g2 g
    a b2 | b a f a a b a g a. b | a f g
    a g e e2 || | c2 c A c e e2 f e | e
    c f A e d B c | d B A G B d g d B |
    a g a c' e f d | c B A F E G A | G A
     B c d e f | g f a e g a g2 :| A e f
     g a | b2 b g2 e d c B | d6 B c :| C
    . a b f b c' b a | g f e f d
Key Adherence: Partial as the output
    includes notes outside the stated
    key of A, such as C and F naturals.
Time Signature: Inconsistent as the
    notation is set in 9/8, but not all
    measures maintain 9 eighth-note
    beats.
```

The key observations in our qualitative evaluation were that the models generated outputs that maintained reasonable melodic development but suffered from rhythmic inconsistencies. However, there was a large improvement in the creativity and coherence of the model outputs since the baseline implementation.

Rather than outputting strings with simple notes or excessive repetition, this improved model opts for more interesting combinations of notes while still maintaining key and time signature adherence.

6 Analysis & Conclusion

6.1 Key Findings

Our experiments yielded three fundamental insights about neural ABC melody generation:

- **Architecture Matters**: The 4-layer GRU outperformed both LSTM and vanilla RNN

architectures, achieving a 12.7% improvement in validation loss over the baseline RNN. This suggests that GRUs' balance between complexity and parameter efficiency makes them particularly suitable for this task.

- **Pitch vs Rhythm**: Models learned pitch relationships (84.8% accuracy) significantly better than rhythmic structure (13.1% accuracy). This 6.5:1 performance ratio indicates that ABC notation may inherently encode pitch information more explicitly than rhythmic patterns.

- **Regularization Critical**: The small dataset (1,850 songs) required aggressive dropout (p=0.5) to prevent overfitting. Without this, validation loss diverged after just 30 epochs compared to 120+ epochs with proper regularization.

6.2 Error Patterns

Qualitative examination revealed three recurring issues in generated samples:

- **Rhythmic Challenges**:
  - Inconsistent measure lengths in compound meters (9/8, 6/8)
  - Difficulty maintaining pulse across phrases
  - Occasional mismatches between notated and implied rhythm

- **Tonal Deviations**:
  - Chromatic notes outside specified key
  - Rare but noticeable harmonic clashes

- **Structural Artifacts**:
  - Abrupt phrase endings
  - Occasional repetitive sequences

6.3 Limitations

While promising, our approach has several key limitations:

- **Rhythmic Complexity**: The models struggled particularly with:
  - Syncopation and off-beat accents
  - Compound meter consistency
  - Variable note durations within measures

- **Data Scarcity**: The 1,850-song dataset proved insufficient for:
  - Learning rare chord progressions
  - Capturing complex modulations
  - Modeling diverse musical forms

- **Evaluation Challenges**: Current metrics may be overly strict about:
  - Exact time signature adherence
  - Chromatic passing tones
  - Ornamentation variability

6.4 Future Work

Although we were able to improve our model's accuracy significantly from our baseline performance, there are still many additions that could be made that would be beneficial to the model.

Even with data augmentation, the model struggles to decrease loss past 1.59 without severely overfitting. One approach to improve model training would be to perform further data collection, such as finding other existing ABC notation datasets or creating a web scraper to collect a large amount of data for the model to learn. The data augmentation function could also be improved by introducing rhythmic variations to increase time-signature adherence.

Another worthwhile action item is to develop a pipeline to convert model output back into valid ABC notation and then into MIDI. Then, once the melody outputted by the model can be played on a computer, it will result in more informative human evaluation.

6.5 Conclusion

Our systematic comparison demonstrates GRUs as the most effective architecture for ABC notation generation, achieving 83.5% key adherence while maintaining musical coherence. The developed tokenization system and evaluation framework provide foundations for future research in neural music composition. While challenges remain—particularly in rhythmic accuracy—the results show that RNN-based approaches can successfully model core aspects of folk melody generation. This work establishes concrete benchmarks for future research and provides practical insights for developers of AI music systems.

References

Li Chen, Hao Huang, and Thomas Lee. 2024. Symbolic music generation with transformers trained on abc notation. *Journal of Artificial Creativity*, 12(1):22–35.

Gaëtan Hadjeres, François Pachet, and Frank Nielsen. 2017. Deepbach: A steerable model for bach chorales generation. *arXiv preprint arXiv:1612.01010*.

Aaron Van den Oord, Sander Dieleman, Heiga Zen, and 1 others. 2016. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Chris Walshaw. 2011. Abc notation standard version 2.1. https://abcnotation.com/wiki/abc:standard:v2.1.

A Implementation Details

A.1 Complete Hyperparameters

Parameter	Value
Optimizer	Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$)
Learning Rate Schedule	Linear warmup (5 epochs)
Gradient Clipping	1.0
Weight Initialization	Xavier Uniform
Padding Index	0 (for cross-entropy)
Sequence Length	256 tokens
Early Stopping Patience	5 epochs

B Complete Results

Table 7: Per-Model Detailed Metrics

Model	Epochs	Train Acc	Val Acc	Key ↑	TS ↑
RNN	140	48.3%	47.1%	84.8%	9.1%
LSTM	138	51.2%	50.0%	81.3%	10.9%
GRU	114	**52.3%**	**51.5%**	**83.5%**	**13.1%**

Contributions

- **Chitransh Motwani:**
  - Designed and implemented the data preprocessing pipeline, including cleaning, standardizing, and structuring the ABC notation dataset.
  - Wrote the preprocess.py script to handle formatting, metadata parsing, and melody deduplication, saving results in standardized files.
  - Participated in writing and editing the final report including introduction, related work and technical sections covering methods, results, and analysis.
  - Helped create and refine presentation slides to effectively communicate project goals and outcomes.
- **Mrinal Goshalia:**
  - Wrote the initial training script and integrated/modified data processing and model to work with training.
  - Experimented with hyperparameters to assist with optimizing performance.
  - Implemented weights and biases to track training metrics and logged/added graph visuals to the milestone report as well as experiment implementation details, and furture work sections.
  - Helped implement the tokenization for grouping rhythmic patterns such as .C and triplet groupings.
  - Implemented time signature adherence test.
  - Wrote the GRU model and updated the RNN model from taking in one-hot inputs such that it would work with the encoded tokens.
  - Assisted with exploring different vocabulary implementations and converting to embedding space.
  - Helped with video presentation slides/implementation, technical support, and team discussions.
  - Contributed to the final report related works, methodology, experiments sections, and reviewed all sections.
- **Brayden Sue:**
  - Wrote the main driver script, PyTorch dataset, and the 3 models (RNN, LSTM, GRU).
  - Defined the vocabulary to model ABC notation and created the tokenizer to turn JSON entries into tensors

- Integrated the training function into the pipeline; wrote the evaluation and sampling functions in train.py
- Updated the dataset to encode the tokens as vocabulary indices rather than one-hot
- Added data augmentation to the dataset to increase variance in the data and greatly improve training
- Modified the dataset and training files to pass metadata along the pipeline, allowing the use of custom performance evaluation metrics
- Modified the tokenizer to add certain metadata fields to the input where available, improving the Seq2Seq accuracy of the model.
- Introduced temperature to the training pipeline, allowing for model creativity control
- Added dropout and early stopping to all models, then fine-tuned the model again to get final hyperparameter values
- Created visualizations for training and validation loss as well as scores from the test set
- Wrote project.ipynb to document the important aspects of the code and explain the pipeline
- Wrote the tokenization, implementation details, hyperparameter optimization, future work and experiments (minus the time signature adherence paragraph) sections of the final report.