HW4

Introduction

This report covers symbolic music generation task, which in specific uses midi-like tokens with language-model-alike RNN model, and reviews a PerformanceRNN model along with custom improvements via temporal conditioning and augmentation. One of the full piano performance MIDI data from Saarland Music Data where 50 piano performances are recorded as audio-MIDI pairs will be used to overfit the single performance.

Question 1: Review the MIDI-like tokenization method

1.1 Dataset

Bach_BWV888-01_008_20110315-SMD from Saarland Music Data will be used to train the model on the single piece throughout the process. The dataset were recorded precisely with automatic piano-to-MIDI conversion program with real performance. Each data contains notes with pitch from 21 to 108, where 60 corresponds to C4, start time, end time, and velocity information, a value related to strength of pressing the note, where 90 is about mf.

1.2 Tokenization

This MIDI preprocessing pipeline transforms raw note sequences into a sequence of discrete events suitable for training autoregressive sequence models. The EventSeq class consists of time-ordered sequence of four types of events: note_on, note_off, velocity, and time_shift. Each event is mapped to an integer index within a unified vocabulary space, allowing for simple tokenization and decoding.

The entire token vocabulary size is 240 tokens with possible values as:

- note_on: 88 possible values, for pitches from MIDI note 21 (A0) to 108 (C8)
- note_off: same as above, 88 values
- velocity: 32 discrete steps, quantized from the MIDI velocity range 21–108
- time_shift: 32 bins, representing different temporal intervals

The time_shift tokens are designed to represent time intervals between successive events. The bin values follow an exponential progression, which provides finer resolution for short timing intervals while covering long pauses with fewer tokens. It reflects perceptual timing sensitivity and enables efficient sequence modeling.

Decoded MIDI files can differ from the original due to quantization error of timing and velocity, simplified heuristics logic of default note length when note-off is missing, or minimum duration for too short notes. Nonetheless, chords and overlapping notes can be represented with consecutive note_on events without time_shift event in between, or cumulated note_on events, thus can be reconstructed without loss.

Question 2: Implement the softmax temperature

2.1 Architecture

The softmax temperature scaling adjusts the confidence of the model's output distribution. Dividing logits by a temperature tau before softmax makes the distribution sharper when tau is smaller than 1, and flatter when tau is larger than 1. Smaller tau means more confidence, whereas larger tau means more uncertainty and diversity, which is useful for controlling randomness during generation. The highlighted part of the following code is the implemented temperature scaling:

```
def forward(self, x, hidden, tau):
    # TO DO: incoporate tau into the following code
    x_encoder = self.encoder(x)
    x_encoder, x_hidden = self.rnn(x_encoder, hidden)
    x_decoder = self.decoder(x_encoder)
    x_decoder = x_decoder / tau # temperature scaling
    x_pred = self.log_softmax(x_decoder)

    return x_pred, x_hidden
```

2.2 Experiment

To examine the effect of temperature scaling, an experiment is conducted by varying τ and evaluating the quality and diversity of unconditionally generated MIDI sequences.

2.2.1 Hypothesis

Adjusting the softmax temperature affects the trade-off between fidelity and diversity. Lower τ (<1) should yield more accurate, deterministic outputs, while higher τ (>1) promotes diversity at the cost of structure. Since τ = 1 corresponds to the unadjusted learned distribution, it is expected to have the distribution with lowest distance.

2.2.2 Setup

Temperature scaling is applied during decoding in a MIDI sequence generation model, with τ ∈ {0.6, 0.8, 1.0, 1.2, 1.4}. The generated sequences are compared to ground truth using L1, L2 distances and KL divergences for pitch, duration, and velocity.

2.2.3 Quantitative Result

	$\tau = 0.6$	$\tau = 0.8$	$\tau = 1.0$ (baseline)	$\tau = 1.2$	$\tau = 1.4$
Pitch	L1: 0.0766 L2: 0.0336 KL(p||q): 0.0039 KL(q||p): 0.0039 average: 0.0295	L1: 0.0932 L2: 0.0406 KL(p||q): 0.0088 KL(q||p): 0.0090 average: 0.0379	L1: 0.1272 L2: 0.0526 KL(p||q): 0.0147 KL(q||p): 0.0144 average: 0.0522	L1: 0.2709 L2: 0.1091 KL(p||q): 0.0511 KL(q||p): 0.0500 average: 0.1203	L1: 0.1666 L2: 0.0726 KL(p||q): 0.0293 KL(q||p): 0.0293 average: 0.0745
Duration	L1: 0.1890 L2: 0.1180 KL(p||q): 0.0328 KL(q||p): 0.0389 average: 0.0947	L1: 0.0824 L2: 0.0508 KL(p||q): 0.0122 KL(q||p): 0.0268 average: 0.0430	L1: 0.1620 L2: 0.0973 KL(p||q): 0.0703 KL(q||p): 0.1310 average: 0.1152	L1: 0.2156 L2: 0.1263 KL(p||q): 0.0462 KL(q||p): 0.1328 average: 0.1302	L1: 0.1121 L2: 0.0636 KL(p||q): 0.0307 KL(q||p): 0.0741 average: 0.0701
Velocity	L1: 0.4493 L2: 0.1622 KL(p||q): 0.1625 KL(q||p): 0.2231 average: 0.2493	L1: 0.4471 L2: 0.1624 KL(p||q): 0.1578 KL(q||p): 0.2097 average: 0.2442	L1: 0.6960 L2: 0.2501 KL(p||q): 0.3263 KL(q||p): 0.4358 average: 0.4271	L1: 0.4726 L2: 0.1743 KL(p||q): 0.1704 KL(q||p): 0.2274 average: 0.2612	L1: 0.3821 L2: 0.1502 KL(p||q): 0.1196 KL(q||p): 0.1218 average: 0.1934
Average	0.1245	**0.1084**	0.1982	0.1706	0.1127

τ = 0.8 achieves the best overall performance. Lower temperatures generally yield smaller errors in pitch and duration. Velocity errors are higher across the board, but τ = 1.4 shows relatively lower divergence.

2.2.4. Qualitative Result

Lower τ (0.6, 0.8) generates rhythmically stable, classical-style outputs that sound natural but monotonous. Higher τ introduces temporal variation, but often in unnatural timings.

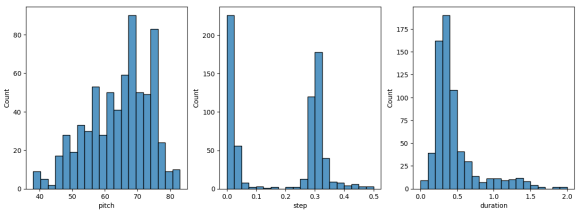
2.2.5 Analysis

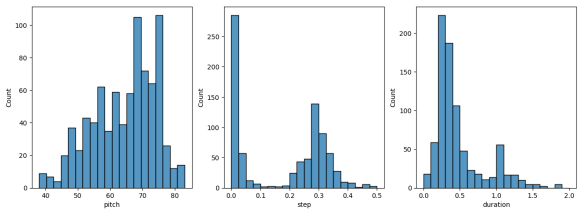
Lower temperatures produce outputs that are statistically closer to the training distribution, indicating better memorization or overfitting. As τ increases, output diversity rises, but the generated distributions diverge more from ground truth. The distribution comparison plots below confirm this claim. Velocity is notably more difficult to match, likely due to its higher variability in human performances and sensitivity to expressive nuance. Moreover, since the velocity distribution doesn't account the timing information, lower distance doesn't strictly tell that velocity sounds in natural timing.

2.2.6 Discussion

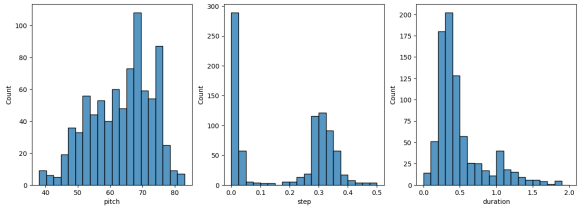
While τ = 1.0 reflects the original learned distribution, it does not guarantee optimal performance. τ = 0.8 yielded the lowest average error, suggesting that a slight sharpening of the distribution improves accuracy. This challenges the assumption that τ = 1.0 is ideal, implying that mild temperature tuning can enhance generation quality without sacrificing too much diversity.

Ground truth $\tau = 0.6$

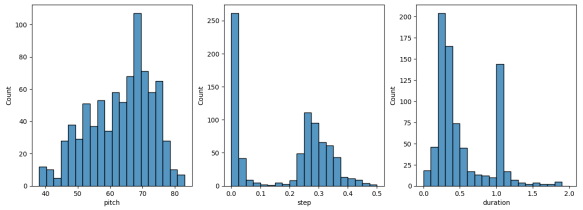




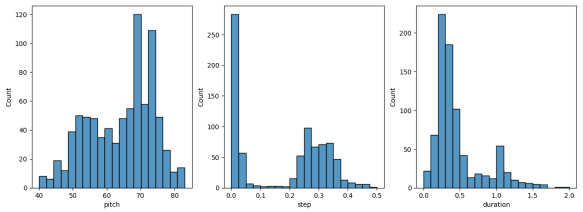
$\tau = 0.8$



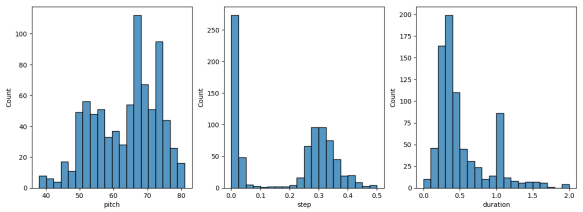
$\tau = 1.0$ (baseline)



$\tau = 1.2$



$\tau = 1.4$



Question 3: Implement the primed generation

3.1 Architecture

Given primer sequence conditions the model's hidden state before generating new tokens. It first processes the primer to update the hidden state accordingly with context. Generation then proceeds

token-by-token from the final primer token, using the updated hidden state. Rest of the process is identical with unconditional generation. The highlighted part of the following code is the implemented primed generation:

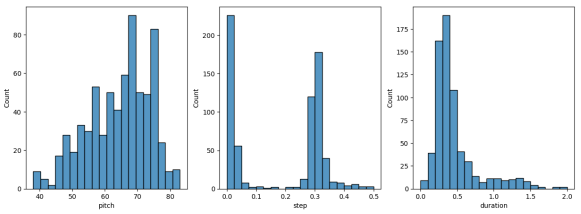
```
def test_primer(self, primer, sequence, tau):
# TO DO: implement the primed generation
  with torch.no_grad():
    self.model.eval()

    # regenerate the hidden state using the primer
    batch_c0, batch_h0 = self.model.init_hidden(batch_size=1, random_init=False)
    init_hidden = (batch_c0.to(self.device), batch_h0.to(self.device))
    _, hidden = self.model(x=primer.unsqueeze(0).to(self.device), hidden=init_hidden, tau=tau)

    # start generation from last token of primer
    pred = primer[-1].unsqueeze(0).to(self.device)
    preds = primer.tolist()

    for step in range(sequence - len(primer)):
        pred, hidden = self.model(x=pred.unsqueeze(0), hidden=hidden, tau=tau)
        pred_dist = pred.data.view(-1).exp()
        pred = torch.multinomial(pred_dist, 1)
        preds.append(pred.cpu().numpy()[0])

    return preds
```

3.2 Generation Result

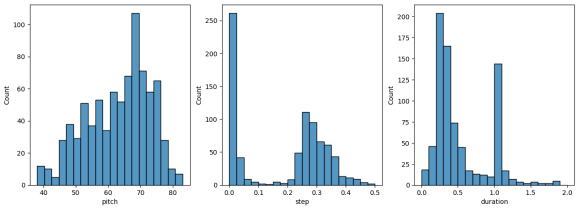
The statistics and plots indicate that the conditional distribution differs subtly due to primer bias, and the generated audio sounds more closely aligned with the given context.

	Uncond	Cond
Pitch	L1: 0.1272 L2: 0.0526 KL(p||q): 0.0147 KL(q||p): 0.0144 average: 0.0522	L1: 0.2241 L2: 0.0972 KL(p||q): 0.0369 KL(q||p): 0.0360 average: 0.0986
Duration	L1: 0.1620 L2: 0.0973 KL(p||q): 0.0703 KL(q||p): 0.1310 average: 0.1152	L1: 0.2192 L2: 0.1400 KL(p||q): 0.3029 KL(q||p): 0.0976 average: 0.1899
Velocity	L1: 0.6960 L2: 0.2501 KL(p||q): 0.3263 KL(q||p): 0.4358 average: 0.4271	L1: 0.3374 L2: 0.1292 KL(p||q): 0.1032 KL(q||p): 0.1337 average: 0.1759

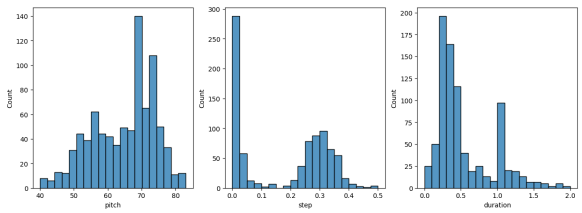
Groundtruth



Uncond (baseline)



Cond



Question 4: Improve the result

4.1 Architecture

To enhance the model's ability to capture timing-related contexts, a tempo-conditioned LSTM architecture combined with tempo-based data augmentation is implemented for comparison on the baseline model.

4.1.1 Tempo conditioned LSTM

Conditioning on a tempo value is applied to the standard LSTM model. A scalar tempo input is projected into the same embedding space as the token encoder via a linear layer and added to each token embedding. This allows the model to modulate its generation according to the specified tempo.

```
class TempoLSTM(nn.Module):
    def __init__(self, n_layers, n_hidden, n_dict, n_enc_dim, tempo_cond=True):

        # Tempo embedding
        if self.tempo_cond:
            self.tempo_proj = nn.Linear(1, n_enc_dim)

        ...

    def forward(self, x, hidden, tau, tempo=None):

        x_encoder = self.encoder(x)

            # Tempo embedding
        if self.tempo_cond:
            if isinstance(tempo, float):
                tempo = torch.FloatTensor([tempo]).unsqueeze(0).to(x_encoder.device)
            elif isinstance(tempo, torch.Tensor) and len(tempo.shape) == 1:
                tempo = tempo.unsqueeze(1)
            tempo_emb = self.tempo_proj(tempo)
            tempo_emb = tempo_emb.unsqueeze(1).expand_as(x_encoder)
            x_encoder = x_encoder + tempo_emb

        ...
```

4.1.2 Tempo augmentation

To help the model generalize across tempo variations, we apply tempo augmentation to the training dataset. Each MIDI file is duplicated and preprocessed multiple times with its note timing scaled by tempo factors ranging from 0.8 to 1.2. This simulates performance at different speeds while preserving musical

content. The corresponding tempo factor is encoded in the filename and parsed as a conditioning label during data loading.

```
# Tempo augmentation
tempo_factors = [0.8, 0.9, 1.0, 1.1, 1.2]

for midi_file in midi_files:
    ...

    base_name = os.path.basename(midi_file).split('.')[0]

    for tempo_factor in tempo_factors:
        aug_notes = []
        for note in sorted_notes:
            aug_note = pretty_midi.Note(
                velocity=note.velocity,
                pitch=note.pitch,
                start=note.start * tempo_factor,
                end=note.end * tempo_factor
            )
            aug_notes.append(aug_note)

                ...

        # save the tokenized file
        suffix = '' if tempo_factor == 1.0 else f'_tempo{tempo_factor}'
        save_name = os.path.join(save_path, out_fmt.format(base_name, suffix))
```

```
class TempoAugDataset(Dataset):
    def __init__(self, window_size, augment_tempo=True):
        ...

        self.data_list = list(find_files_by_extensions(self.data_path, ['.data']))
        if not augment_tempo:
            self.data_list = [f for f in self.data_list if '_tempo' not in f]
                ...

    def __getitem__(self, idx):

        ...

        # Extract tempo factor from filename
        base_name = os.path.basename(file_name)
        tempo_factor = float(base_name.split('_tempo')[1].split('.data')[0]) if '_tempo' in base_name else 1.0

        return torch.LongTensor(data), torch.FloatTensor([tempo_factor])
```

4.2 Experiment

A set of experiments is conducted to evaluate the effectiveness of tempo conditioning and tempo augmentation, both individually and in combination.

4.2.1 Hypothesis

Explicitly conditioning on tempo and training with tempo-scaled data will encourage the model to learn tempo-invariant representations, leading to improved generation quality and robustness to timing variations.

4.2.2 Setup

Four experimental settings are evaluated, with or without tempo conditioning and with or without augmentation. For models using conditioning, tempo values are sampled uniformly from [0.7, 1.3] during training. During generation, tempo is fixed at 1.0. All other training parameters and evaluation metrics are kept consistent with prior experiments, except for the number of training iterations, which is increased to 20,000 for more capacity.

4.2.3 Quantitative Result

The following table shows the distribution distance metrics (L1, L2, KL divergences) between generated and real data for pitch, duration, and velocity. The last row shows the average across all categories.

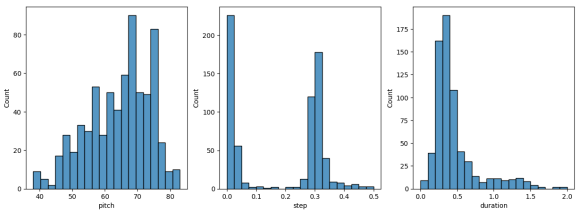
	Baseline	Tempo conditioning + augmentation	Tempo conditioning only	Tempo augmentation only
Pitch	L1: 0.0753 L2: 0.0294 KL(p||q): 0.0073 KL(q||p): 0.0071 average: 0.0298	L1: 0.0950 L2: 0.0410 KL(p||q): 0.0076 KL(q||p): 0.0074 average: 0.0377	L1: 0.0599 L2: 0.0249 KL(p||q): 0.0032 KL(q||p): 0.0033 average: 0.0228	L1: 0.0698 L2: 0.0247 KL(p||q): 0.0056 KL(q||p): 0.0050 average: 0.0263
Duration	L1: 0.1175 L2: 0.0722 KL(p||q): 0.0676 KL(q||p): 0.0383 average: 0.0739	L1: 0.0661 L2: 0.0421 KL(p||q): 0.0108 KL(q||p): 0.0093 average: 0.0321	L1: 0.0519 L2: 0.0301 KL(p||q): 0.0080 KL(q||p): 0.0072 average: 0.0243	L1: 0.0733 L2: 0.0403 KL(p||q): 0.0522 KL(q||p): 0.0578 average: 0.0559
Velocity	L1: 0.3503 L2: 0.1236 KL(p||q): 0.1091 KL(q||p): 0.1465 average: 0.1824	L1: 0.3377 L2: 0.1243 KL(p||q): 0.1048 KL(q||p): 0.1370 average: 0.1760	L1: 0.3822 L2: 0.1324 KL(p||q): 0.1262 KL(q||p): 0.1674 average: 0.2021	L1: 0.3289 L2: 0.1187 KL(p||q): 0.1053 KL(q||p): 0.1346 average: 0.1719
Average	0.0954	**0.0819**	0.0830	0.0847

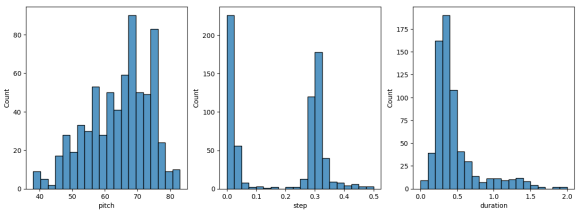
Tempo conditioning only shows the best alignment in pitch and duration distributions while augmentation only improves duration and velocity compared to the baseline. The combined model achieves the best overall average across all features though not always the best in individual metrics. The baseline performs consistently worse across all dimensions confirming the advantage of tempo-aware modeling.

4.2.4 Qualitative Result

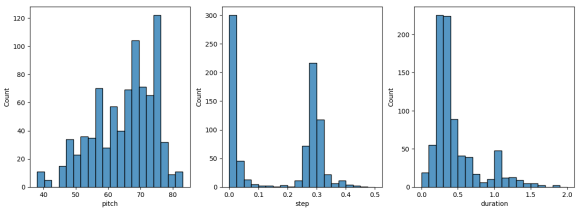
Samples have more natural timing while baseline outputs often lack rhythmic stability especially under tempo variation.

groundtruth baseline

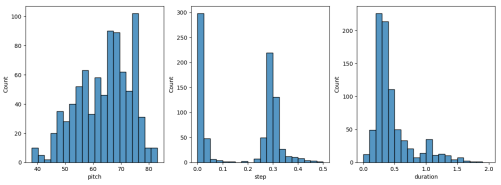




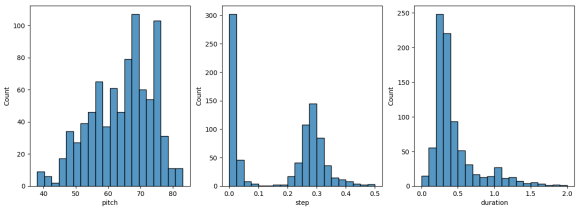
both



Tempo conditioning only



Tempo augmentation only



4.2.5 Analysis

Conditioning enables the model to capture timing structures more effectively by providing explicit tempo cues. Augmentation enhances robustness by exposing the model to tempo-scaled data leading to better generalization. Combining both approaches allows the model to adapt and generalize across tempo variations. Velocity remains harder to model suggesting it is less dependent on tempo and more influenced by expressive nuance.

4.2.6 Discussion

Tempo conditioning and augmentation complement each other by combining explicit control and data diversity. Together they improve the temporal quality and consistency of generated music. Velocity modeling remains a limitation indicating the need for dynamics-specific features or disentangled representations. Future work may explore additional conditioning variables such as articulation or performer style to enhance expressiveness.

Conclusion

The report explored unconditional and conditional symbolic music generation using a MIDI token-based RNN model with temperature scaling and temporal enhancement techniques. Temperature tuning significantly influenced generation quality, with $\tau = 0.8$ yielding the best trade-off. Tempo-conditioned models and tempo-augmented dataset further improved rhythmic naturalness and context awareness.