Simple RNN Music Generator

Baseline I

Sangjee Dondrub

2023/07/15

Contents

Introduction

> A simple RNN-based model for Tibetan music generation.

Imports

```
import glob
import random

import numpy as np
from pretty_midi import Instrument, Note, PrettyMIDI, instrument_name_to_program
from tqdm import tqdm

import torch
from torch.utils.data import DataLoader, Dataset
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

Let' s define some util functions

```
def GetNoteSequence(instrument: Instrument) -> np.ndarray:
    sorted_notes = sorted(instrument.notes, key=lambda x: x.start)
    assert len(sorted_notes) > 0
    notes = []
    prev_start = sorted_notes[0].start
    for note in sorted_notes:
        notes.append([note.pitch, note.start - prev_start, note.end - note.start])
        prev_start = note.start
    return np.array(notes)


def CreateMIDIInstrumennt(notes: np.ndarray, instrument_name: str) -> Instrument:
    instrument = Instrument(instrument_name_to_program(instrument_name))
    prev_start = 0
    for note in notes:
        prev_start += note[1]
        note = Note(
            start=prev_start, end=prev_start + note[2], pitch=note[0], velocity=100
        )
        instrument.notes.append(note)
```

```
    return instrument
```

Define a simple-as-f* RNN model

```
class SimpleRNNMusicGeneratorModel(torch.nn.Module):
    def __init__(self):
        super(SimpleRNNMusicGeneratorModel, self).__init__()
        self.lstm = torch.nn.LSTM(3, 128, num_layers=1, batch_first=True)
        self.pitch_linear = torch.nn.Linear(128, 128)
        self.pitch_sigmoid = torch.nn.Sigmoid()
        self.step_linear = torch.nn.Linear(128, 1)
        self.duration_linear = torch.nn.Linear(128, 1)

    def forward(self, x):
        x, _ = self.lstm(x)
        pitch = self.pitch_sigmoid(self.pitch_linear(x[:, -1]))
        step = self.step_linear(x[:, -1])
        duration = self.duration_linear(x[:, -1])
        return {"pitch": pitch, "step": step, "duration": duration}
```

Custom Loss

```
class CustomLoss(torch.nn.Module):
    def __init__(self, weight):
        super(CustomLoss, self).__init__()
        self.weight = torch.Tensor(weight)
        self.pitch_loss = torch.nn.CrossEntropyLoss()
        self.step_loss = self.mse_with_positive_pressure
        self.duration_loss = self.mse_with_positive_pressure

    @staticmethod
    def mse_with_positive_pressure(pred, y):
        mse = (y - pred) ** 2
        positive_pressure = 10 * torch.maximum(-pred, torch.tensor(0))
        return torch.mean(mse + positive_pressure)

    def forward(self, pred, y):
        a = self.pitch_loss(pred["pitch"], y["pitch"])
        b = self.step_loss(pred["step"], y["step"])
        c = self.duration_loss(pred["duration"], y["duration"])
```

```
        return a * self.weight[0] + b * self.weight[1] + c * self.weight[2]
```

Dataset

```
class MusicDataset(Dataset):
    def __init__(self, files, seq_len, max_file_num=None):
        notes = None
        filenames = glob.glob(files)
        print(f"Find {len(filenames)} files.")
        if max_file_num is None:
            max_file_num = len(filenames)
        print(f"Reading {max_file_num} files...")
        for f in tqdm(filenames[:max_file_num]):
            pm = PrettyMIDI(f)
            instrument = pm.instruments[0]
            new_notes = GetNoteSequence(instrument)
            new_notes /= [128.0, 1.0, 1.0]
            if notes is not None:
                notes = np.append(notes, new_notes, axis=0)
            else:
                notes = new_notes

        self.seq_len = seq_len
        self.notes = np.array(notes, dtype=np.float32)

    def __len__(self):
        return len(self.notes) - self.seq_len

    def __getitem__(self, idx) -> (np.ndarray, dict):
        label_note = self.notes[idx + self.seq_len]
        label = {
            "pitch": (label_note[0] * 128).astype(np.int64),
            "step": label_note[1],
            "duration": label_note[2],
        }
        return self.notes[idx : idx + self.seq_len], label

    def getendseq(self) -> np.ndarray:
        return self.notes[-self.seq_len :]
```

```
# note feature: pitch, step, duration
```

```
batch_size = 64
sequence_lenth = 25
max_file_num = 1200
epochs = 200
learning_rate = 0.005

loss_weight = [0.1, 20.0, 1.0]

save_model_name = "model.pth"

device = "cuda" if torch.cuda.is_available() else "cpu"

trainning_data = MusicDataset(
    "datasets/*.mid", sequence_lenth, max_file_num=max_file_num
)
print(f"Read {len(trainning_data)} sequences.")
```

```
loader = DataLoader(trainning_data, batch_size=batch_size)
```

```
next(iter(loader))
```

```
for X, y in loader:
    print(f"X: {X.shape} {X.dtype}")
    print(f"y: {y}")
    break

model = SimpleRNNMusicGeneratorModel().to(device)
loss_fn = CustomLoss(loss_weight).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
print(model)
print(loss_fn)

print("Start trainning...")
size = len(loader.dataset)
for t in range(epochs):
    model.train()
    avg_loss = 0.0
    print(f"Epoch {t+1}\n-----------------")
    for batch, (X, y) in enumerate(tqdm(loader)):
        X = X.to(device)
        for feat in y.keys():
            y[feat] = y[feat].to(device)
```

```
        pred = model(X)
        loss = loss_fn(pred, y)
        avg_loss = avg_loss + loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    avg_loss /= len(loader)
    print(f"average loss = {avg_loss}")
    if (t + 1) % 10 == 0:
        torch.save(model.state_dict(), "model%d.pth" % (t + 1))
print("Done!")

torch.save(model.state_dict(), save_model_name)
print(f"Saved PyTorch Model State to {save_model_name}")
```

Predict

```
sample_file_name = "sample.mid"
sample_file_name = 'little_star.mid'
output_file_name = "sample-out-1.mid"
save_model_name = "models/model110.pth"
predict_length = 128
sequence_lenth = 10
```

```
def WeightedRandom(weight, k=100000) -> int:
    sum = int(0)
    for w in weight:
        sum += int(k*w)
    x = random.randint(1, sum)
    sum = 0
    for id, w in enumerate(weight):
        sum += int(k*w)
        if sum >= x:
            return id
    return


def PredictNextNote(model: SimpleRNNMusicGeneratorModel(), input: np.ndarray):
    model.eval()
```

```
    with torch.no_grad():
        input = torch.tensor(input, dtype=torch.float32).unsqueeze(0)
        pred = model(input)
        pitch = WeightedRandom(np.squeeze(pred['pitch'], axis=0))
        step = np.maximum(np.squeeze(pred['step'], axis=0), 0)
        duration = np.maximum(np.squeeze(pred['duration'], axis=0), 0)
    return pitch, float(step), float(duration)


model = SimpleRNNMusicGeneratorModel()
model.to(device)

model.load_state_dict(torch.load(save_model_name,map_location=device))

sample_data = MusicDataset(sample_file_name, sequence_lenth)

cur = sample_data.getendseq()
res = []
prev_start = 0
for i in tqdm(range(predict_length)):
    pitch, step, duration = PredictNextNote(model, cur)
    res.append([pitch, step, duration])
    cur = cur[1:]
    cur = np.append(cur, [[pitch, step, duration]], axis=0)
    prev_start += step

pm_output = PrettyMIDI()
pm_output.instruments.append(
    CreateMIDIInstrumennt(res, "Acoustic Grand Piano"))
pm_output.write(output_file_name)
```