Experiment: 2

Aim

To generate musical notes using a Recurrent Neural Network (RNN) trained on the MAESTRO dataset, where the model predicts the next note in a sequence based on previous notes, enabling the creation of new music compositions.

Theory

Music generation involves predicting the next note in a sequence of musical notes, a task well-suited for Recurrent Neural Networks (RNNs) due to their ability to model sequential data. This experiment uses a simple RNN to train on piano MIDI files from the MAESTRO dataset, which contains approximately 1,200 MIDI files of piano music. The model is designed to learn the relationship between notes and predict the next note in a sequence, represented by three key features: pitch (note's frequency), step (time elapsed from the previous note), and duration (how long the note is played).

The dataset is preprocessed by extracting these features from the MIDI files and forming sequences of notes to train the model. The RNN learns to predict the next note in the sequence, making it possible to generate longer sequences of notes through repeated predictions. The model is optimized using a custom loss function that ensures non-negative values for step and duration, while the pitch is predicted from a softmax distribution to introduce variety in the generated music. Finally, the trained model is used to generate new music sequences, which can be converted back into MIDI files for playback

This tutorial uses the `pretty_midi` library to create and parse MIDI files, and `pyfluidsynth` for generating audio playback in Colab.

In [ ]:
```python
import collections
import datetime
import fluidsynth
import glob
import numpy as np
import pathlib
import pandas as pd
import pretty_midi
import seaborn as sns
import tensorflow as tf

from IPython import display
from matplotlib import pyplot as plt
from typing import Optional
```

In [ ]:
```python
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)

# Sampling rate for audio playback
_SAMPLING_RATE = 16000
```

Download the Maestro dataset

In [ ]:

```
data_dir = pathlib.Path('data/maestro-v2.0.0')
if not data_dir.exists():
  tf.keras.utils.get_file(
      'maestro-v2.0.0-midi.zip',
      origin='https://storage.googleapis.com/magentadata/datasets/maestro/v2.0.0/maestro-v2.0
      extract=True,
      cache_dir='.', cache_subdir='data',
  )
```

```
Downloading data from https://storage.googleapis.com/magentadata/datasets/maestro/v2.0.0/maes
tro-v2.0.0-midi.zip
59243107/59243107 [==============================] - 1s 0us/step
```

The dataset contains about 1,200 MIDI files.

In [ ]:

```
filenames = glob.glob(str(data_dir/'**/*.mid*'))
print('Number of files:', len(filenames))
```

```
Number of files: 1282
```

Process a MIDI file

First, use `pretty_midi` to parse a single MIDI file and inspect the format of the notes. If you would like to download the MIDI file below to play on your computer, you can do so in colab by writing `files.download(sample_file)`.

In [ ]:

```
sample_file = filenames[1]
print(sample_file)
```

```
data/maestro-v2.0.0/2017/MIDI-Unprocessed_059_PIANO059_MID--AUDIO-split_07-07-17_Piano-e_2-03
_wav--3.midi
```

Generate a `PrettyMIDI` object for the sample MIDI file.

In [ ]:

```
pm = pretty_midi.PrettyMIDI(sample_file)
```

Play the sample file. The playback widget may take several seconds to load.

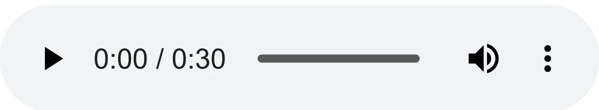
In [ ]:

```
def display_audio(pm: pretty_midi.PrettyMIDI, seconds=30):
  waveform = pm.fluidsynth(fs=_SAMPLING_RATE)
  # Take a sample of the generated waveform to mitigate kernel resets
  waveform_short = waveform[:seconds*_SAMPLING_RATE]
  return display.Audio(waveform_short, rate=_SAMPLING_RATE)
```

In [ ]:

```
display_audio(pm)
```

Out[ ]:



Do some inspection on the MIDI file. What kinds of instruments are used?

In [ ]:

```
print('Number of instruments:', len(pm.instruments))
instrument = pm.instruments[0]
instrument_name = pretty_midi.program_to_instrument_name(instrument.program)
print('Instrument name:', instrument_name)
```

```
Number of instruments: 1
Instrument name: Acoustic Grand Piano
```

Extract notes

In [ ]:
```
for i, note in enumerate(instrument.notes[:10]):
  note_name = pretty_midi.note_number_to_name(note.pitch)
  duration = note.end - note.start
  print(f'{i}: pitch={note.pitch}, note_name={note_name},'
        f' duration={duration:.4f}')
```

```
0: pitch=48, note_name=C3, duration=0.1031
1: pitch=48, note_name=C3, duration=0.0990
2: pitch=55, note_name=G3, duration=0.0854
3: pitch=60, note_name=C4, duration=0.0854
4: pitch=64, note_name=E4, duration=0.0531
5: pitch=60, note_name=C4, duration=0.0896
6: pitch=67, note_name=G4, duration=0.0740
7: pitch=72, note_name=C5, duration=0.0729
8: pitch=76, note_name=E5, duration=0.0563
9: pitch=72, note_name=C5, duration=0.0865
```

You will use three variables to represent a note when training the model: `pitch`, `step` and `duration`. The pitch is the perceptual quality of the sound as a MIDI note number. The `step` is the time elapsed from the previous note or start of the track. The `duration` is how long the note will be playing in seconds and is the difference between the note end and note start times.

Extract the notes from the sample MIDI file.

In [ ]:
```
def midi_to_notes(midi_file: str) -> pd.DataFrame:
  pm = pretty_midi.PrettyMIDI(midi_file)
  instrument = pm.instruments[0]
  notes = collections.defaultdict(list)

  # Sort the notes by start time
  sorted_notes = sorted(instrument.notes, key=lambda note: note.start)
  prev_start = sorted_notes[0].start

  for note in sorted_notes:
    start = note.start
    end = note.end
    notes['pitch'].append(note.pitch)
    notes['start'].append(start)
    notes['end'].append(end)
    notes['step'].append(start - prev_start)
    notes['duration'].append(end - start)
    prev_start = start

  return pd.DataFrame({name: np.array(value) for name, value in notes.items()})
```

In [ ]:
```
raw_notes = midi_to_notes(sample_file)
raw_notes.head()
```

Out[ ]:

	pitch	start	end	step	duration
0	48	0.977083	1.080208	0.000000	0.103125
1	36	0.978125	2.641667	0.001042	1.663542
2	48	1.184375	1.283333	0.206250	0.098958
3	55	1.278125	1.363542	0.093750	0.085417
4	60	1.368750	1.454167	0.090625	0.085417

It may be easier to interpret the note names rather than the pitches, so you can use the function below to convert from the numeric pitch values to note names. The note name shows the type of note, accidental and octave number (e.g. C#4).

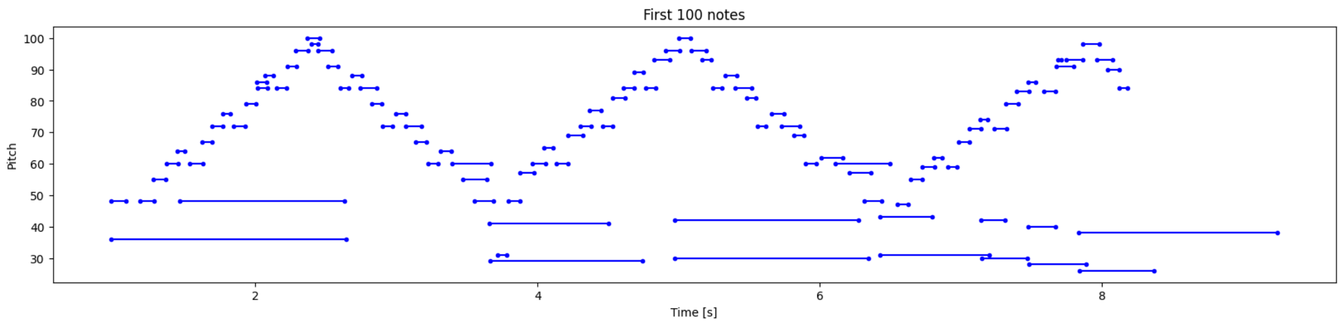
In [ ]:
```
get_note_names = np.vectorize(pretty_midi.note_number_to_name)
sample_note_names = get_note_names(raw_notes['pitch'])
sample_note_names[:10]
```

Out[ ]:
```
array(['C3', 'C2', 'C3', 'G3', 'C4', 'E4', 'C3', 'C4', 'G4', 'C5'],
      dtype='<U3')
```

To visualize the musical piece, plot the note pitch, start and end across the length of the track (i.e. piano roll). Start with the first 100 notes

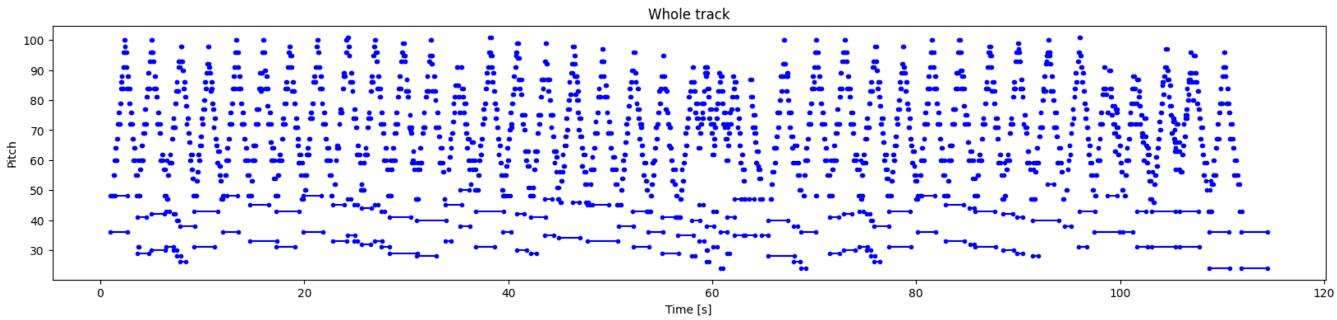
In [ ]:
```
def plot_piano_roll(notes: pd.DataFrame, count: Optional[int] = None):
  if count:
    title = f'First {count} notes'
  else:
    title = f'Whole track'
    count = len(notes['pitch'])
  plt.figure(figsize=(20, 4))
  plot_pitch = np.stack([notes['pitch'], notes['pitch']], axis=0)
  plot_start_stop = np.stack([notes['start'], notes['end']], axis=0)
  plt.plot(
      plot_start_stop[:, :count], plot_pitch[:, :count], color="b", marker=".")
  plt.xlabel('Time [s]')
  plt.ylabel('Pitch')
  _ = plt.title(title)
```

In [ ]:
```
plot_piano_roll(raw_notes, count=100)
```



Plot the notes for the entire track.

In [ ]:
```
plot_piano_roll(raw_notes)
```

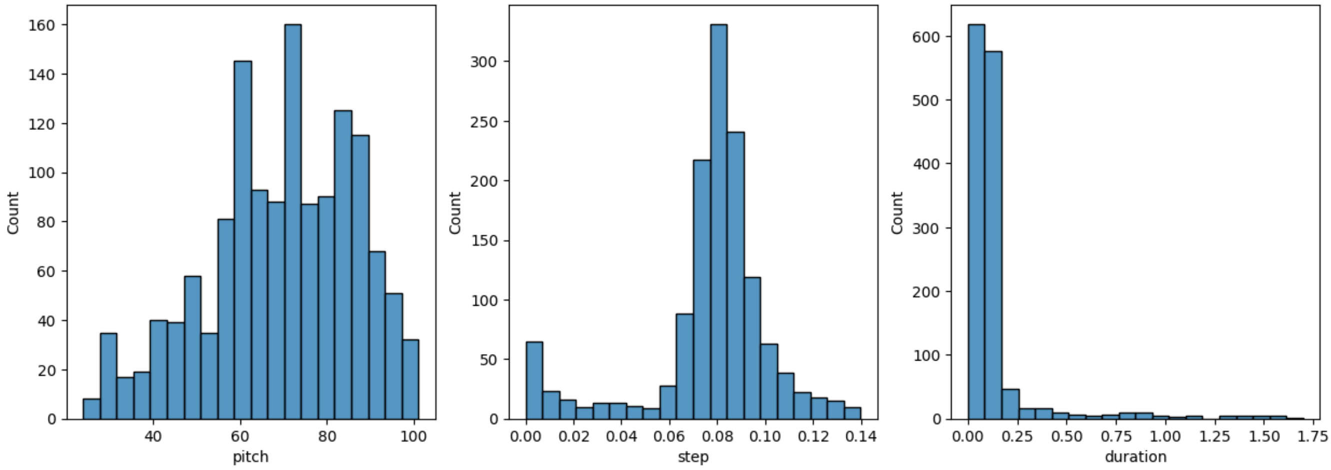


Check the distribution of each note variable.

In [ ]:
```
def plot_distributions(notes: pd.DataFrame, drop_percentile=2.5):
  plt.figure(figsize=[15, 5])
  plt.subplot(1, 3, 1)
  sns.histplot(notes, x="pitch", bins=20)

  plt.subplot(1, 3, 2)
  max_step = np.percentile(notes['step'], 100 - drop_percentile)
  sns.histplot(notes, x="step", bins=np.linspace(0, max_step, 21))

  plt.subplot(1, 3, 3)
```

```
    max_duration = np.percentile(notes['duration'], 100 - drop_percentile)
    sns.histplot(notes, x="duration", bins=np.linspace(0, max_duration, 21))
```

In [ ]:
```
plot_distributions(raw_notes)
```



Create a MIDI file

You can generate your own MIDI file from a list of notes using the function below.

In [ ]:
```
def notes_to_midi(
  notes: pd.DataFrame,
  out_file: str,
  instrument_name: str,
  velocity: int = 100,  # note loudness
) -> pretty_midi.PrettyMIDI:

  pm = pretty_midi.PrettyMIDI()
  instrument = pretty_midi.Instrument(
      program=pretty_midi.instrument_name_to_program(
          instrument_name))

  prev_start = 0
  for i, note in notes.iterrows():
    start = float(prev_start + note['step'])
    end = float(start + note['duration'])
    note = pretty_midi.Note(
        velocity=velocity,
        pitch=int(note['pitch']),
        start=start,
        end=end,
    )
    instrument.notes.append(note)
    prev_start = start

  pm.instruments.append(instrument)
  pm.write(out_file)
  return pm
```

In [ ]:
```
example_file = 'example.midi'
example_pm = notes_to_midi(
    raw_notes, out_file=example_file, instrument_name=instrument_name)
```

Play the generated MIDI file and see if there is any difference.

In [ ]:
```
display_audio(example_pm)
```

Out[ ]: 0:00 / 0:30

As before, you can write `files.download(example_file)` to download and play this file.

Create the training dataset

Create the training dataset by extracting notes from the MIDI files. You can start by using a small number of files, and experiment later with more. This may take a couple minutes.

In [ ]:
```
num_files = 5
all_notes = []
for f in filenames[:num_files]:
  notes = midi_to_notes(f)
  all_notes.append(notes)

all_notes = pd.concat(all_notes)
```

In [ ]:
```
n_notes = len(all_notes)
print('Number of notes parsed:', n_notes)
```

```
Number of notes parsed: 20994
```

Next, create a `tf.data.Dataset` from the parsed notes.

In [ ]:
```
key_order = ['pitch', 'step', 'duration']
train_notes = np.stack([all_notes[key] for key in key_order], axis=1)
```

In [ ]:
```
notes_ds = tf.data.Dataset.from_tensor_slices(train_notes)
notes_ds.element_spec
```

Out[ ]:
```
TensorSpec(shape=(3,), dtype=tf.float64, name=None)
```

You will train the model on batches of sequences of notes. Each example will consist of a sequence of notes as the input features, and the next note as the label. In this way, the model will be trained to predict the next note in a sequence. You can find a diagram describing this process (and more details) in Text classification with an RNN.

You can use the handy window function with size `seq_length` to create the features and labels in this format.

In [ ]:
```
def create_sequences(
    dataset: tf.data.Dataset,
    seq_length: int,
    vocab_size = 128,
) -> tf.data.Dataset:
  """Returns TF Dataset of sequence and label examples."""
  seq_length = seq_length+1

  # Take 1 extra for the labels
  windows = dataset.window(seq_length, shift=1, stride=1,
                              drop_remainder=True)

  # `flat_map` flattens the" dataset of datasets" into a dataset of tensors
  flatten = lambda x: x.batch(seq_length, drop_remainder=True)
  sequences = windows.flat_map(flatten)

  # Normalize note pitch
  def scale_pitch(x):
    x = x/[vocab_size,1.0,1.0]
```

```
    return x

  # Split the labels
  def split_labels(sequences):
    inputs = sequences[:-1]
    labels_dense = sequences[-1]
    labels = {key:labels_dense[i] for i,key in enumerate(key_order)}

    return scale_pitch(inputs), labels

  return sequences.map(split_labels, num_parallel_calls=tf.data.AUTOTUNE)
```

Set the sequence length for each example. Experiment with different lengths (e.g. 50, 100, 150) to see which one works best for the data, or use hyperparameter tuning. The size of the vocabulary (`vocab_size`) is set to 128 representing all the pitches supported by `pretty_midi` .

In [ ]:

```
seq_length = 25
vocab_size = 128
seq_ds = create_sequences(notes_ds, seq_length, vocab_size)
seq_ds.element_spec
```

Out[ ]:

```
(TensorSpec(shape=(25, 3), dtype=tf.float64, name=None),
 {'pitch': TensorSpec(shape=(), dtype=tf.float64, name=None),
  'step': TensorSpec(shape=(), dtype=tf.float64, name=None),
  'duration': TensorSpec(shape=(), dtype=tf.float64, name=None)})
```

The shape of the dataset is `(100,1)` , meaning that the model will take 100 notes as input, and learn to predict the following note as output.

In [ ]:

```
for seq, target in seq_ds.take(1):
  print('sequence shape:', seq.shape)
  print('sequence elements (first 10):', seq[0: 10])
  print()
  print('target:', target)
```

```
sequence shape: (25, 3)
sequence elements (first 10): tf.Tensor(
[[4.29687500e-01 0.00000000e+00 1.54479167e+00]
 [3.98437500e-01 7.29166667e-03 1.74062500e+00]
 [3.75000000e-01 1.04166667e-03 1.35833333e+00]
 [5.62500000e-01 8.23958333e-01 7.34375000e-01]
 [5.85937500e-01 1.37812500e+00 5.83333333e-02]
 [4.29687500e-01 1.05208333e-01 1.68437500e+00]
 [3.90625000e-01 6.25000000e-03 1.66770833e+00]
 [5.78125000e-01 2.08333333e-03 1.78125000e-01]
 [3.67187500e-01 1.14583333e-02 1.48541667e+00]
 [5.23437500e-01 7.78125000e-01 6.11458333e-01]], shape=(10, 3), dtype=float64)

target: {'pitch': <tf.Tensor: shape=(), dtype=float64, numpy=48.0>, 'step': <tf.Tensor: shape
=(), dtype=float64, numpy=0.005208333333333037>, 'duration': <tf.Tensor: shape=(), dtype=floa
t64, numpy=1.6906249999999998>}
```

Batch the examples, and configure the dataset for performance.

In [ ]:

```
batch_size = 64
buffer_size = n_notes - seq_length  # the number of items in the dataset
train_ds = (seq_ds
            .shuffle(buffer_size)
            .batch(batch_size, drop_remainder=True)
            .cache()
            .prefetch(tf.data.experimental.AUTOTUNE))
```

In [ ]:

```
train_ds.element_spec
```

Out[ ]:
```
(TensorSpec(shape=(64, 25, 3), dtype=tf.float64, name=None),
 {'pitch': TensorSpec(shape=(64,), dtype=tf.float64, name=None),
  'step': TensorSpec(shape=(64,), dtype=tf.float64, name=None),
  'duration': TensorSpec(shape=(64,), dtype=tf.float64, name=None)})
```

Create and train the model

The model will have three outputs, one for each note variable. For `step` and `duration`, you will use a custom loss function based on mean squared error that encourages the model to output non-negative values.

In [ ]:
```
def mse_with_positive_pressure(y_true: tf.Tensor, y_pred: tf.Tensor):
  mse = (y_true - y_pred) ** 2
  positive_pressure = 10 * tf.maximum(-y_pred, 0.0)
  return tf.reduce_mean(mse + positive_pressure)
```

In [ ]:
```
input_shape = (seq_length, 3)
learning_rate = 0.005

inputs = tf.keras.Input(input_shape)
x = tf.keras.layers.LSTM(128)(inputs)

outputs = {
  'pitch': tf.keras.layers.Dense(128, name='pitch')(x),
  'step': tf.keras.layers.Dense(1, name='step')(x),
  'duration': tf.keras.layers.Dense(1, name='duration')(x),
}

model = tf.keras.Model(inputs, outputs)

loss = {
      'pitch': tf.keras.losses.SparseCategoricalCrossentropy(
          from_logits=True),
      'step': mse_with_positive_pressure,
      'duration': mse_with_positive_pressure,
}

optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

model.compile(loss=loss, optimizer=optimizer)

model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 25, 3)]              0         []

 lstm (LSTM)                 (None, 128)                  67584     ['input_1[0][0]']

 duration (Dense)            (None, 1)                    129       ['lstm[0][0]']

 pitch (Dense)               (None, 128)                  16512     ['lstm[0][0]']

 step (Dense)                (None, 1)                    129       ['lstm[0][0]']

==================================================================================================
Total params: 84354 (329.51 KB)
Trainable params: 84354 (329.51 KB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

Testing the `model.evaluate` function, you can see that the `pitch` loss is significantly greater than the `step` and `duration` losses. Note that `loss` is the total loss computed by summing all the other losses and is currently dominated by the `pitch` loss.

In [ ]:
```
losses = model.evaluate(train_ds, return_dict=True)
losses
```

```
327/327 [==============================] - 20s 29ms/step - loss: 6.2875 - duration_loss: 1.0652 - pitch_loss: 4.8477 - step_loss: 0.3746
```

Out[ ]:
```
{'loss': 6.287472248077393,
 'duration_loss': 1.065152645111084,
 'pitch_loss': 4.847684383392334,
 'step_loss': 0.3746379315853119}
```

One way balance this is to use the `loss_weights` argument to compile:

In [ ]:
```
model.compile(
    loss=loss,
    loss_weights={
        'pitch': 0.05,
        'step': 1.0,
        'duration':1.0,
    },
    optimizer=optimizer,
)
```

The `loss` then becomes the weighted sum of the individual losses.

In [ ]:
```
model.evaluate(train_ds, return_dict=True)
```

```
327/327 [==============================] - 8s 22ms/step - loss: 1.6822 - duration_loss: 1.0652 - pitch_loss: 4.8477 - step_loss: 0.3746
```

Out[ ]:
```
{'loss': 1.682174801826477,
 'duration_loss': 1.065152645111084,
 'pitch_loss': 4.847684383392334,
 'step_loss': 0.3746379315853119}
```

Train the model.

In [ ]:
```
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        filepath='./training_checkpoints/ckpt_{epoch}',
```

```
        save_weights_only=True),
    tf.keras.callbacks.EarlyStopping(
        monitor='loss',
        patience=5,
        verbose=1,
        restore_best_weights=True),
]
```

In [ ]:

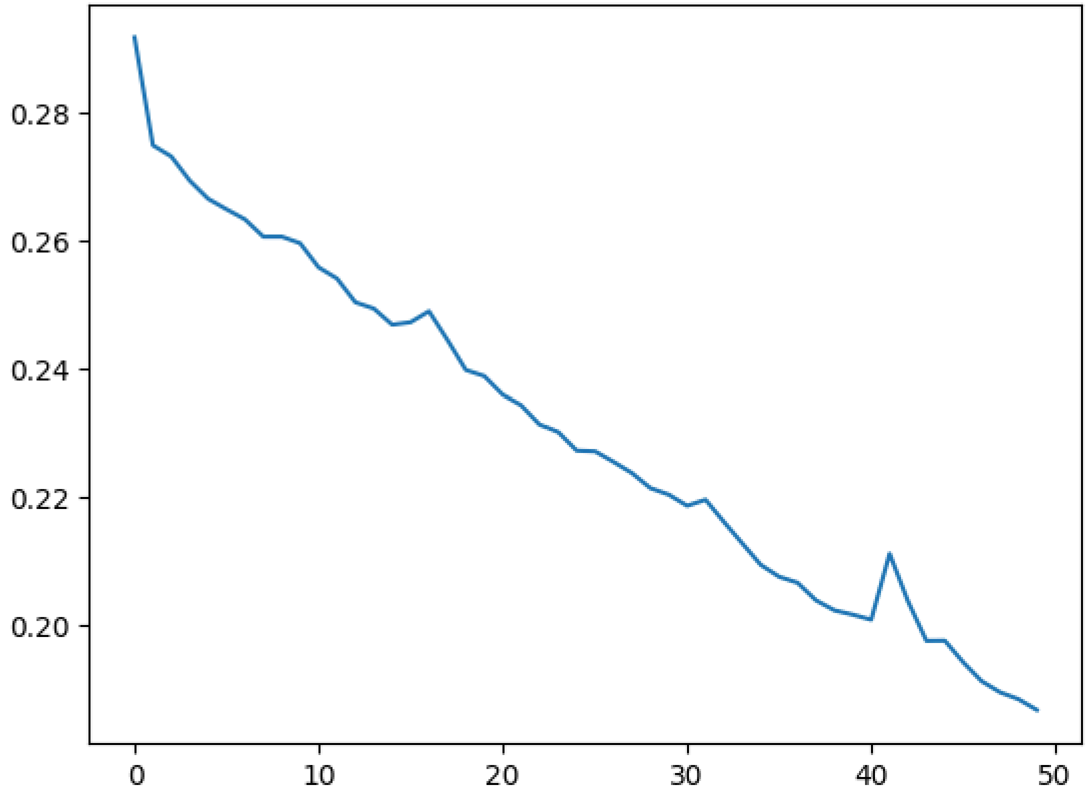
```
%%time
epochs = 50

history = model.fit(
    train_ds,
    epochs=epochs,
    callbacks=callbacks,
)
```

```
Epoch 1/50
327/327 [==============================] - 18s 46ms/step - loss: 0.2917 - duration_loss: 0.06
49 - pitch_loss: 4.2075 - step_loss: 0.0164
Epoch 2/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2748 - duration_loss: 0.05
85 - pitch_loss: 4.0703 - step_loss: 0.0129
Epoch 3/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2731 - duration_loss: 0.05
80 - pitch_loss: 4.0463 - step_loss: 0.0128
Epoch 4/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2693 - duration_loss: 0.05
79 - pitch_loss: 3.9755 - step_loss: 0.0126
Epoch 5/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2665 - duration_loss: 0.05
74 - pitch_loss: 3.9313 - step_loss: 0.0125
Epoch 6/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2649 - duration_loss: 0.05
65 - pitch_loss: 3.9182 - step_loss: 0.0124
Epoch 7/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2633 - duration_loss: 0.05
59 - pitch_loss: 3.9002 - step_loss: 0.0123
Epoch 8/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2606 - duration_loss: 0.05
41 - pitch_loss: 3.8871 - step_loss: 0.0121
Epoch 9/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2606 - duration_loss: 0.05
41 - pitch_loss: 3.8883 - step_loss: 0.0121
Epoch 10/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2596 - duration_loss: 0.05
37 - pitch_loss: 3.8771 - step_loss: 0.0120
Epoch 11/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2558 - duration_loss: 0.05
15 - pitch_loss: 3.8496 - step_loss: 0.0118
Epoch 12/50
327/327 [==============================] - 15s 47ms/step - loss: 0.2541 - duration_loss: 0.05
10 - pitch_loss: 3.8294 - step_loss: 0.0116
Epoch 13/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2504 - duration_loss: 0.04
78 - pitch_loss: 3.8140 - step_loss: 0.0118
Epoch 14/50
327/327 [==============================] - 15s 47ms/step - loss: 0.2494 - duration_loss: 0.04
76 - pitch_loss: 3.8084 - step_loss: 0.0114
Epoch 15/50
327/327 [==============================] - 15s 44ms/step - loss: 0.2469 - duration_loss: 0.04
56 - pitch_loss: 3.7962 - step_loss: 0.0114
Epoch 16/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2472 - duration_loss: 0.04
66 - pitch_loss: 3.7862 - step_loss: 0.0113
Epoch 17/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2489 - duration_loss: 0.04
82 - pitch_loss: 3.7937 - step_loss: 0.0111
Epoch 18/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2445 - duration_loss: 0.04
60 - pitch_loss: 3.7524 - step_loss: 0.0108
Epoch 19/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2398 - duration_loss: 0.04
28 - pitch_loss: 3.7315 - step_loss: 0.0105
Epoch 20/50
327/327 [==============================] - 15s 47ms/step - loss: 0.2389 - duration_loss: 0.04
28 - pitch_loss: 3.7168 - step_loss: 0.0102
Epoch 21/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2360 - duration_loss: 0.04
09 - pitch_loss: 3.6940 - step_loss: 0.0104
Epoch 22/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2344 - duration_loss: 0.04
03 - pitch_loss: 3.6795 - step_loss: 0.0101
Epoch 23/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2314 - duration_loss: 0.03
86 - pitch_loss: 3.6603 - step_loss: 0.0098
```

```
Epoch 24/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2302 - duration_loss: 0.03
87 - pitch_loss: 3.6399 - step_loss: 0.0095
Epoch 25/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2273 - duration_loss: 0.03
73 - pitch_loss: 3.6167 - step_loss: 0.0091
Epoch 26/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2272 - duration_loss: 0.03
79 - pitch_loss: 3.6061 - step_loss: 0.0089
Epoch 27/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2255 - duration_loss: 0.03
69 - pitch_loss: 3.5886 - step_loss: 0.0092
Epoch 28/50
327/327 [==============================] - 15s 47ms/step - loss: 0.2238 - duration_loss: 0.03
68 - pitch_loss: 3.5609 - step_loss: 0.0089
Epoch 29/50
327/327 [==============================] - 15s 47ms/step - loss: 0.2214 - duration_loss: 0.03
58 - pitch_loss: 3.5433 - step_loss: 0.0085
Epoch 30/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2204 - duration_loss: 0.03
52 - pitch_loss: 3.5385 - step_loss: 0.0083
Epoch 31/50
327/327 [==============================] - 14s 43ms/step - loss: 0.2187 - duration_loss: 0.03
44 - pitch_loss: 3.4982 - step_loss: 0.0094
Epoch 32/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2196 - duration_loss: 0.03
60 - pitch_loss: 3.5056 - step_loss: 0.0083
Epoch 33/50
327/327 [==============================] - 15s 44ms/step - loss: 0.2162 - duration_loss: 0.03
45 - pitch_loss: 3.4680 - step_loss: 0.0082
Epoch 34/50
327/327 [==============================] - 14s 43ms/step - loss: 0.2128 - duration_loss: 0.03
26 - pitch_loss: 3.4396 - step_loss: 0.0082
Epoch 35/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2095 - duration_loss: 0.03
13 - pitch_loss: 3.4053 - step_loss: 0.0079
Epoch 36/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2076 - duration_loss: 0.03
03 - pitch_loss: 3.3880 - step_loss: 0.0079
Epoch 37/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2067 - duration_loss: 0.03
03 - pitch_loss: 3.3678 - step_loss: 0.0079
Epoch 38/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2039 - duration_loss: 0.02
94 - pitch_loss: 3.3410 - step_loss: 0.0075
Epoch 39/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2024 - duration_loss: 0.02
86 - pitch_loss: 3.3268 - step_loss: 0.0074
Epoch 40/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2017 - duration_loss: 0.02
82 - pitch_loss: 3.3196 - step_loss: 0.0075
Epoch 41/50
327/327 [==============================] - 15s 46ms/step - loss: 0.2009 - duration_loss: 0.02
83 - pitch_loss: 3.2997 - step_loss: 0.0076
Epoch 42/50
327/327 [==============================] - 15s 45ms/step - loss: 0.2112 - duration_loss: 0.03
31 - pitch_loss: 3.4097 - step_loss: 0.0076
Epoch 43/50
327/327 [==============================] - 14s 44ms/step - loss: 0.2038 - duration_loss: 0.03
07 - pitch_loss: 3.3141 - step_loss: 0.0074
Epoch 44/50
327/327 [==============================] - 15s 45ms/step - loss: 0.1976 - duration_loss: 0.02
71 - pitch_loss: 3.2632 - step_loss: 0.0074
Epoch 45/50
327/327 [==============================] - 14s 44ms/step - loss: 0.1976 - duration_loss: 0.02
77 - pitch_loss: 3.2513 - step_loss: 0.0074
Epoch 46/50
327/327 [==============================] - 14s 44ms/step - loss: 0.1943 - duration_loss: 0.02
62 - pitch_loss: 3.2178 - step_loss: 0.0071
```

```
Epoch 47/50
327/327 [==============================] - 15s 44ms/step - loss: 0.1913 - duration_loss: 0.02
51 - pitch_loss: 3.1863 - step_loss: 0.0069
Epoch 48/50
327/327 [==============================] - 14s 44ms/step - loss: 0.1896 - duration_loss: 0.02
37 - pitch_loss: 3.1817 - step_loss: 0.0068
Epoch 49/50
327/327 [==============================] - 15s 45ms/step - loss: 0.1885 - duration_loss: 0.02
32 - pitch_loss: 3.1732 - step_loss: 0.0066
Epoch 50/50
327/327 [==============================] - 15s 45ms/step - loss: 0.1868 - duration_loss: 0.02
33 - pitch_loss: 3.1405 - step_loss: 0.0065
CPU times: user 17min 16s, sys: 60 s, total: 18min 16s
Wall time: 14min 48s
```

In [ ]:
```
plt.plot(history.epoch, history.history['loss'], label='total loss')
plt.show()
```



Generate notes

To use the model to generate notes, you will first need to provide a starting sequence of notes. The function below generates one note from a sequence of notes.

For note pitch, it draws a sample from the softmax distribution of notes produced by the model, and does not simply pick the note with the highest probability. Always picking the note with the highest probability would lead to repetitive sequences of notes being generated.

The `temperature` parameter can be used to control the randomness of notes generated. You can find more details on temperature in Text generation with an RNN.

In [ ]:
```
def predict_next_note(
    notes: np.ndarray,
    model: tf.keras.Model,
    temperature: float = 1.0) -> tuple[int, float, float]:
  """Generates a note as a tuple of (pitch, step, duration), using a trained sequence model."

  assert temperature > 0
```

```
  # Add batch dimension
  inputs = tf.expand_dims(notes, 0)

  predictions = model.predict(inputs)
  pitch_logits = predictions['pitch']
  step = predictions['step']
  duration = predictions['duration']

  pitch_logits /= temperature
  pitch = tf.random.categorical(pitch_logits, num_samples=1)
  pitch = tf.squeeze(pitch, axis=-1)
  duration = tf.squeeze(duration, axis=-1)
  step = tf.squeeze(step, axis=-1)

  # `step` and `duration` values should be non-negative
  step = tf.maximum(0, step)
  duration = tf.maximum(0, duration)

  return int(pitch), float(step), float(duration)
```

Now generate some notes. You can play around with temperature and the starting sequence in `next_notes` and see what happens.

In [ ]:

```
temperature = 2.0
num_predictions = 120

sample_notes = np.stack([raw_notes[key] for key in key_order], axis=1)

# The initial sequence of notes; pitch is normalized similar to training
# sequences
input_notes = (
    sample_notes[:seq_length] / np.array([vocab_size, 1, 1]))

generated_notes = []
prev_start = 0
for _ in range(num_predictions):
  pitch, step, duration = predict_next_note(input_notes, model, temperature)
  start = prev_start + step
  end = start + duration
  input_note = (pitch, step, duration)
  generated_notes.append((*input_note, start, end))
  input_notes = np.delete(input_notes, 0, axis=0)
  input_notes = np.append(input_notes, np.expand_dims(input_note, 0), axis=0)
  prev_start = start

generated_notes = pd.DataFrame(
    generated_notes, columns=(*key_order, 'start', 'end'))
```

```
1/1 [==============================] - 0s 496ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 39ms/step
```

```
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
```

In [ ]:

```
generated_notes.head(10)
```

Out[ ]:

	pitch	step	duration	start	end
0	79	0.026249	0.105218	0.026249	0.131467
1	97	0.027427	0.161812	0.053676	0.215488
2	97	0.043140	0.142999	0.096816	0.239816
3	86	0.024314	0.171399	0.121130	0.292530
4	88	0.038276	0.193428	0.159407	0.352835
5	86	0.092009	0.116320	0.251415	0.367735
6	84	0.069781	0.180908	0.321196	0.502104
7	86	0.049032	0.188873	0.370227	0.559100
8	67	0.057262	0.160340	0.427489	0.587829
9	86	0.063030	0.115808	0.490520	0.606328

In [ ]:

```
out_file = 'output.mid'
out_pm = notes_to_midi(
    generated_notes, out_file=out_file, instrument_name=instrument_name)
display_audio(out_pm)
```

Out[ ]:

0:00 / 0:08

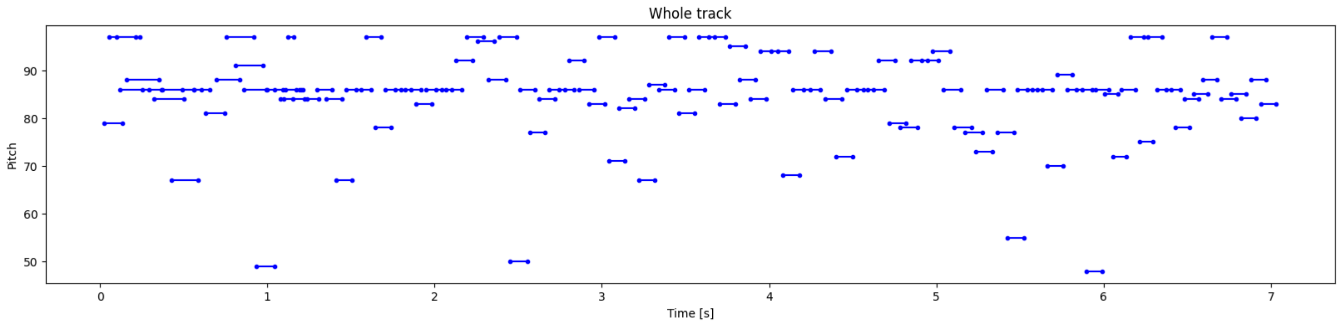
You can also download the audio file by adding the two lines below:

```
from google.colab import files
files.download(out_file)
```

Visualize the generated notes.

In [ ]:

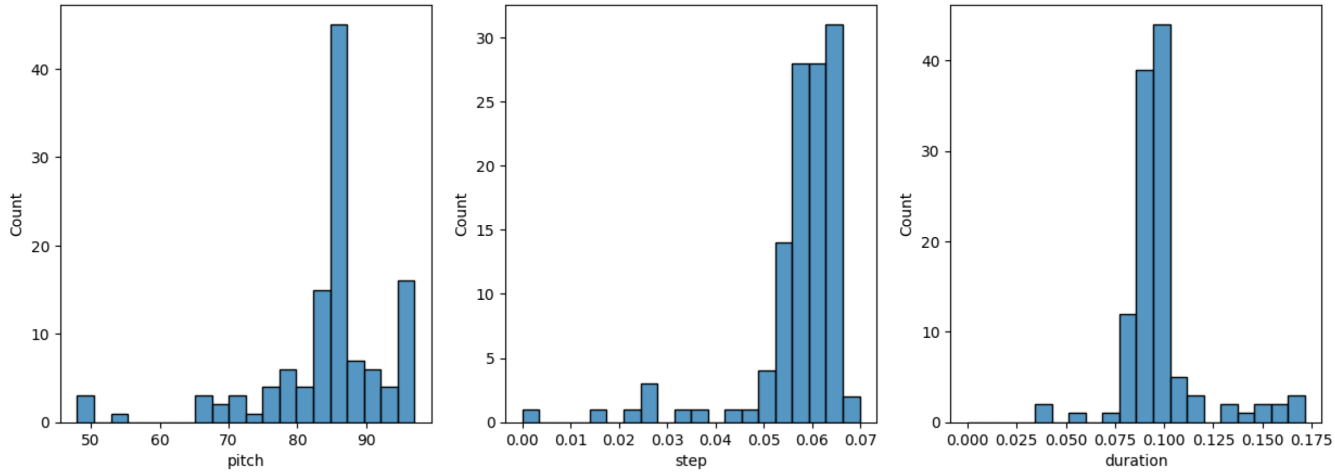
```
plot_piano_roll(generated_notes)
```



Check the distributions of `pitch` , `step` and `duration` .

In [ ]:

```
plot_distributions(generated_notes)
```



In the above plots, you will notice the change in distribution of the note variables. Since there is a feedback loop between the model's outputs and inputs, the model tends to generate similar sequences of outputs to reduce the loss. This is particularly relevant for `step` and `duration`, which uses the MSE loss. For `pitch`, you can increase the randomness by increasing the `temperature` in `predict_next_note`.