In [1]:

[View on TensorFlow.org (https://www.tensorflow.org/tutorials/audio/music_generation)](https://www.tensorflow.org/tutorials/audio/music_generation) [Run in Goog (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/audio/music_generatic](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/audio/music_generatic)

This tutorial shows you how to generate musical notes using a simple recurrent neural network (RNN). You will train a model using a collection of piano MIDI files from the [MAESTRO dataset (https://magenta.tensorflow.org/datasets/maestro)](https://magenta.tensorflow.org/datasets/maestro). Given a sequence of notes, your model will learn to predict the next note in the sequence. You can generate longer sequences of notes by calling the model repeatedly.

This tutorial contains complete code to parse and create MIDI files. You can learn more about how RNNs work by visiting the [Text generation with an RNN (https://www.tensorflow.org/text/tutorials/text_generation)](https://www.tensorflow.org/text/tutorials/text_generation) tutorial.

This tutorial uses the [`pretty_midi` (https://github.com/craffel/pretty-midi)](https://github.com/craffel/pretty-midi) library to create and parse MIDI files, and [`pyfluidsynth` (https://github.com/nwhitehead/pyfluidsynth)](https://github.com/nwhitehead/pyfluidsynth) for generating audio playback in Colab.

In [2]:
```
!sudo apt install -y fluidsynth
```

```
  fluid-soundfont-gm fluidsynth libdouble-conversion3 libfluidsynth2
  libinstpatch-1.0-2 libpcre2-16-0 libqt5core5a libqt5dbus5 libqt5gui5
  libqt5network5 libqt5svg5 libqt5widgets5 libsdl2-2.0-0 qsynth
  qt5-gtk-platformtheme qttranslations5-l10n timgm6mb-soundfont
0 upgraded, 17 newly installed, 0 to remove and 93 not upgraded.
Need to get 136 MB of archives.
After this operation, 202 MB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu (http://us-central1.gce.archive.ubuntu.com/ubuntu) focal/
universe amd64 libdouble-conversion3 amd64 3.1.5-4ubuntu1 [37.9 kB]
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu (http://us-central1.gce.archive.ubuntu.com/ubuntu) focal-
updates/main amd64 libpcre2-16-0 amd64 10.34-7ubuntu0.1 [181 kB]
Get:3 http://us-central1.gce.archive.ubuntu.com/ubuntu (http://us-central1.gce.archive.ubuntu.com/ubuntu) focal-
updates/universe amd64 libqt5core5a amd64 5.12.8+dfsg-0ubuntu2.1 [2006 kB]
Get:4 http://us-central1.gce.archive.ubuntu.com/ubuntu (http://us-central1.gce.archive.ubuntu.com/ubuntu) focal-
updates/universe amd64 libqt5dbus5 amd64 5.12.8+dfsg-0ubuntu2.1 [208 kB]
Get:5 http://us-central1.gce.archive.ubuntu.com/ubuntu (http://us-central1.gce.archive.ubuntu.com/ubuntu) focal-
updates/universe amd64 libqt5network5 amd64 5.12.8+dfsg-0ubuntu2.1 [673 kB]
Get:6 http://us-central1.gce.archive.ubuntu.com/ubuntu (http://us-central1.gce.archive.ubuntu.com/ubuntu) focal-
updates/universe amd64 libqt5gui5 amd64 5.12.8+dfsg-0ubuntu2.1 [2971 kB]
Get:7 http://us-central1.gce.archive.ubuntu.com/ubuntu (http://us-central1.gce.archive.ubuntu.com/ubuntu) focal-
```

In [3]:
```
!pip install --upgrade pyfluidsynth
```

```
Collecting pyfluidsynth
  Downloading pyFluidSynth-1.3.2-py3-none-any.whl (19 kB)
Requirement already satisfied: numpy in /tmpfs/src/tf_docs_env/lib/python3.9/site-packages (from pyfluidsynth) (1.
26.1)
Installing collected packages: pyfluidsynth
Successfully installed pyfluidsynth-1.3.2
```

In [4]:

```
!pip install pretty_midi
```

```
Collecting pretty_midi
  Downloading pretty_midi-0.2.10.tar.gz (5.6 MB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy>=1.7.0 in /tmpfs/src/tf_docs_env/lib/python3.9/site-packages (from pretty_mid
i) (1.26.1)
Collecting mido>=1.1.16 (from pretty_midi)
  Downloading mido-1.3.0-py3-none-any.whl.metadata (5.1 kB)
Requirement already satisfied: six in /tmpfs/src/tf_docs_env/lib/python3.9/site-packages (from pretty_midi) (1.16.
0)
Requirement already satisfied: packaging~=23.1 in /tmpfs/src/tf_docs_env/lib/python3.9/site-packages (from mido>=
1.1.16->pretty_midi) (23.2)
Downloading mido-1.3.0-py3-none-any.whl (50 kB)
Building wheels for collected packages: pretty_midi
  Building wheel for pretty_midi (setup.py) ... done
  Created wheel for pretty_midi: filename=pretty_midi-0.2.10-py3-none-any.whl size=5592287 sha256=f7d88e5b16376925
e8b98b6963d75a807b486489a1dc0823c57786b22cd52d52
  Stored in directory: /home/kbuilder/.cache/pip/wheels/75/ec/20/b8e937a5bcf1de547ea5ce465db7de7f6761e15e6f0a01e25
f
Successfully built pretty_midi
Installing collected packages: mido, pretty_midi
Successfully installed mido-1.3.0 pretty_midi-0.2.10
```

In [5]:

```
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
```

```
2023-10-27 05:49:15.925119: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register cuD
NN factory: Attempting to register factory for plugin cuDNN when one has already been registered
2023-10-27 05:49:15.925168: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register cuFF
T factory: Attempting to register factory for plugin cuFFT when one has already been registered
2023-10-27 05:49:15.926725: E external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to register cu
BLAS factory: Attempting to register factory for plugin cuBLAS when one has already been registered
```

In [6]:
```
seed = 42
tf.random.set_seed(seed)
np.random.seed(seed)

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

Download the Maestro dataset

In [7]:
```
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-v2.0.0-midi.zip',
      extract=True,
      cache_dir='.', cache_subdir='data',
  )
```

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

The dataset contains about 1,200 MIDI files.

In [8]:
```
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 [9]:
```
sample_file = filenames[1]
print(sample_file)
```

```
data/maestro-v2.0.0/2008/MIDI-Unprocessed_05_R1_2008_01-04_ORIG_MID--AUDIO_05_R1_2008_wav--4.midi
```

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

In [10]:
```
pm = pretty_midi.PrettyMIDI(sample_file)
```

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

In [11]:
```
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 [12]:
```
display_audio(pm)
```

```
fluidsynth: warning: SDL2 not initialized, SDL2 audio driver won't be usable
fluidsynth: error: Unknown integer parameter 'synth.sample-rate'
```

Out[12]:

0:00 / 0:00

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

In [13]:
```
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 [14]:
```
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=54, note_name=F#3, duration=0.0612
1: pitch=51, note_name=D#3, duration=0.0781
2: pitch=58, note_name=A#3, duration=0.0898
3: pitch=39, note_name=D#2, duration=0.0703
4: pitch=46, note_name=A#2, duration=0.1029
5: pitch=39, note_name=D#2, duration=0.0495
6: pitch=51, note_name=D#3, duration=0.0599
7: pitch=46, note_name=A#2, duration=0.0443
8: pitch=54, note_name=F#3, duration=0.0651
9: pitch=63, note_name=D#4, duration=0.9219
```

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 [15]:

```python
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 [16]:

```python
raw_notes = midi_to_notes(sample_file)
raw_notes.head()
```

Out[16]:

	pitch	start	end	step	duration
0	63	0.910156	1.832031	0.000000	0.921875
1	58	1.320312	1.410156	0.410156	0.089844
2	51	1.330729	1.408854	0.010417	0.078125
3	46	1.330729	1.433594	0.000000	0.102865
4	54	1.334635	1.395833	0.003906	0.061198

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 [17]:

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

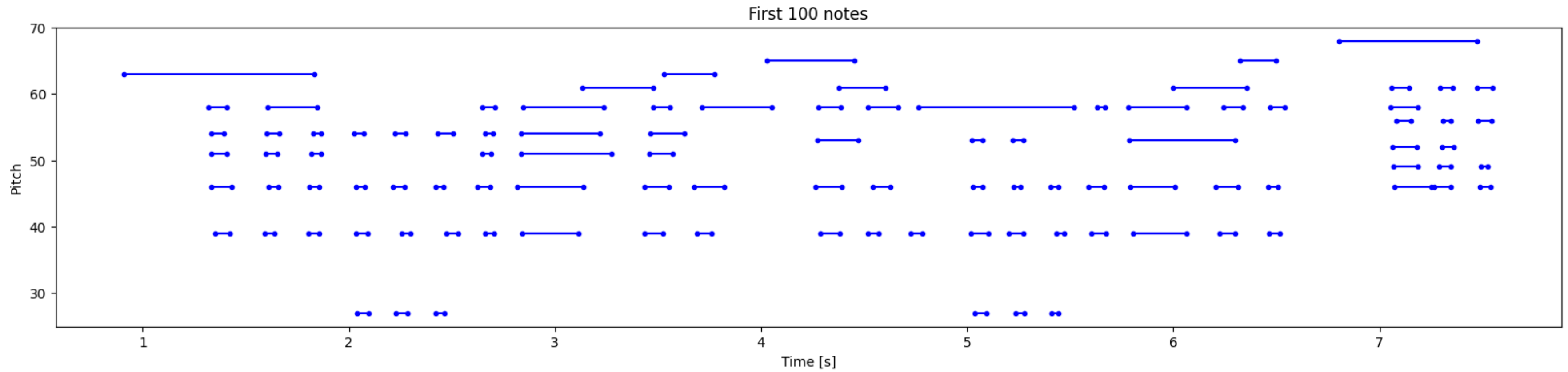
Out[17]:

```
array(['D#4', 'A#3', 'D#3', 'A#2', 'F#3', 'D#2', 'D#2', 'D#3', 'F#3',
       'A#3'], 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 [18]:
```
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)
```

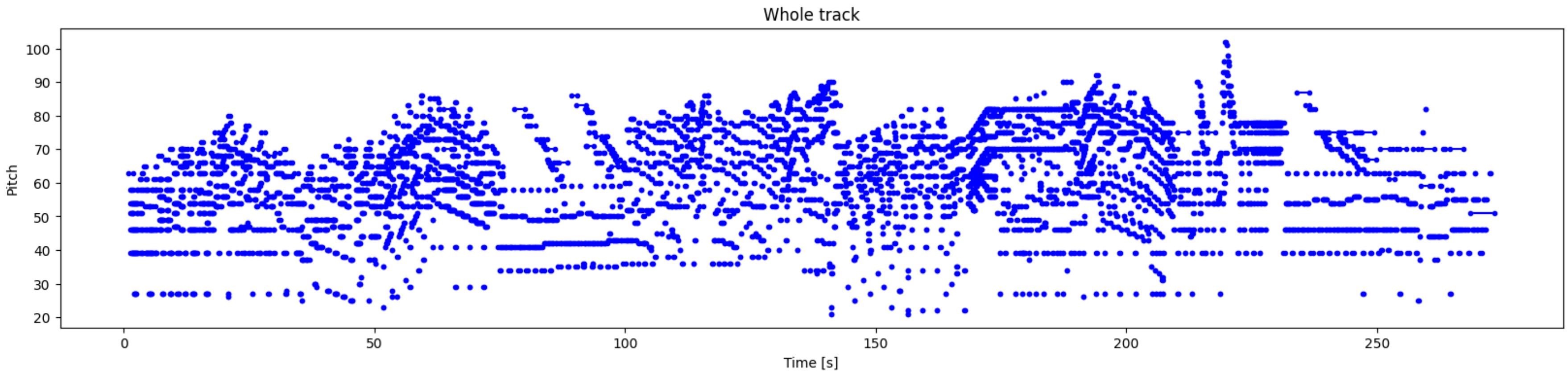
In [19]:
```
plot_piano_roll(raw_notes, count=100)
```



Plot the notes for the entire track.

In [20]:

```
plot_piano_roll(raw_notes)
```



Check the distribution of each note variable.

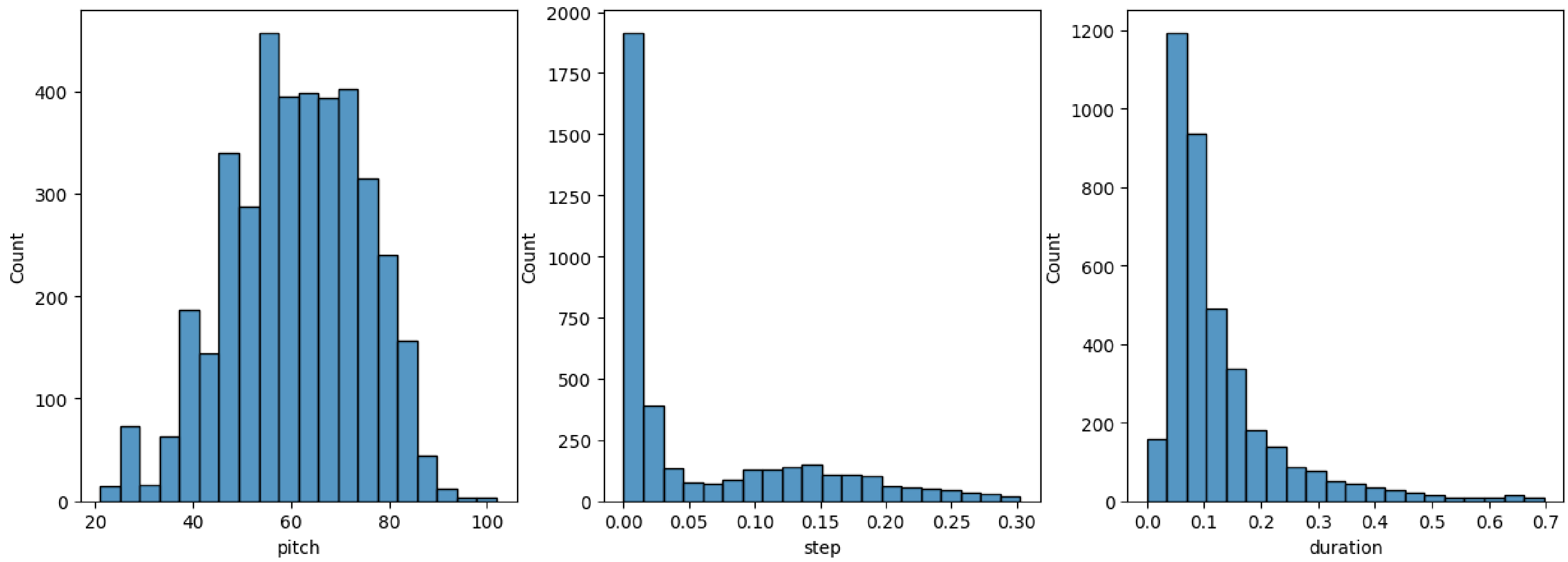
In [21]:

```
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 [22]:
```
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 [23]:
```python
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 [24]:
```python
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 [25]:
```python
display_audio(example_pm)
```

```
fluidsynth: warning: SDL2 not initialized, SDL2 audio driver won't be usable
fluidsynth: error: Unknown integer parameter 'synth.sample-rate'
```

Out[25]:

0:00 / 0:00

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 [26]:
```
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 [27]:
```
n_notes = len(all_notes)
print('Number of notes parsed:', n_notes)
```

```
Number of notes parsed: 15315
```

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

In [28]:
```
key_order = ['pitch', 'step', 'duration']
train_notes = np.stack([all_notes[key] for key in key_order], axis=1)
```

In [29]:
```
notes_ds = tf.data.Dataset.from_tensor_slices(train_notes)
notes_ds.element_spec
```

Out[29]:
```
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 (https://www.tensorflow.org/text/tutorials/text_generation)](https://www.tensorflow.org/text/tutorials/text_generation).

You can use the handy [window (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#window)](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#window) function with size `seq_length` to create the features and labels in this format.

In [30]:
```python
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 (https://www.tensorflow.org/tutorials/keras/keras_tuner)](https://www.tensorflow.org/tutorials/keras/keras_tuner). The size of the vocabulary (`vocab_size`) is set to 128 representing all the pitches supported by `pretty_midi` .

In [31]:
```python
seq_length = 25
vocab_size = 128
seq_ds = create_sequences(notes_ds, seq_length, vocab_size)
seq_ds.element_spec
```

Out[31]:
```
(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 [32]:
```
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(
[[0.625      0.         0.23828125]
 [0.6015625  0.04036458 0.2421875 ]
 [0.5859375  0.22395833 0.06510417]
 [0.5625     0.09505208 0.0703125 ]
 [0.53125    0.11067708 0.1640625 ]
 [0.5859375  0.05598958 0.12760417]
 [0.5625     0.09244792 0.08333333]
 [0.625      0.08333333 0.17317708]
 [0.6015625  0.01822917 0.15104167]
 [0.5859375  0.109375   0.04036458]], shape=(10, 3), dtype=float64)

target: {'pitch': <tf.Tensor: shape=(), dtype=float64, numpy=68.0>, 'step': <tf.Tensor: shape=(), dtype=float64, n
umpy=0.11588541666666652>, 'duration': <tf.Tensor: shape=(), dtype=float64, numpy=0.15494791666666696>}
```

Batch the examples, and configure the dataset for performance.

In [33]:
```
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 [34]:
```
train_ds.element_spec
```

Out[34]:
```
(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 [35]:

```
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 [36]:

```python
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 [37]:
```
losses = model.evaluate(train_ds, return_dict=True)
losses
```

```
238/238 [==============================] - 4s 3ms/step - loss: 6.1272 - duration_loss: 0.6039 - pitch_loss: 4.8544
- step_loss: 0.6689
```

Out[37]:
```
{'loss': 6.127169609069824,
 'duration_loss': 0.603919267654419,
 'pitch_loss': 4.854353427886963,
 'step_loss': 0.6688962578773499}
```

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

In [38]:
```
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 [39]:
```
model.evaluate(train_ds, return_dict=True)
```

```
238/238 [==============================] - 2s 3ms/step - loss: 1.5155 - duration_loss: 0.6039 - pitch_loss: 4.8544
- step_loss: 0.6689
```

Out[39]:
```
{'loss': 1.515533447265625,
 'duration_loss': 0.603919267654419,
 'pitch_loss': 4.854353427886963,
 'step_loss': 0.6688962578773499}
```

Train the model.

In [40]:
```
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 [41]:
```
%%time
epochs = 50

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

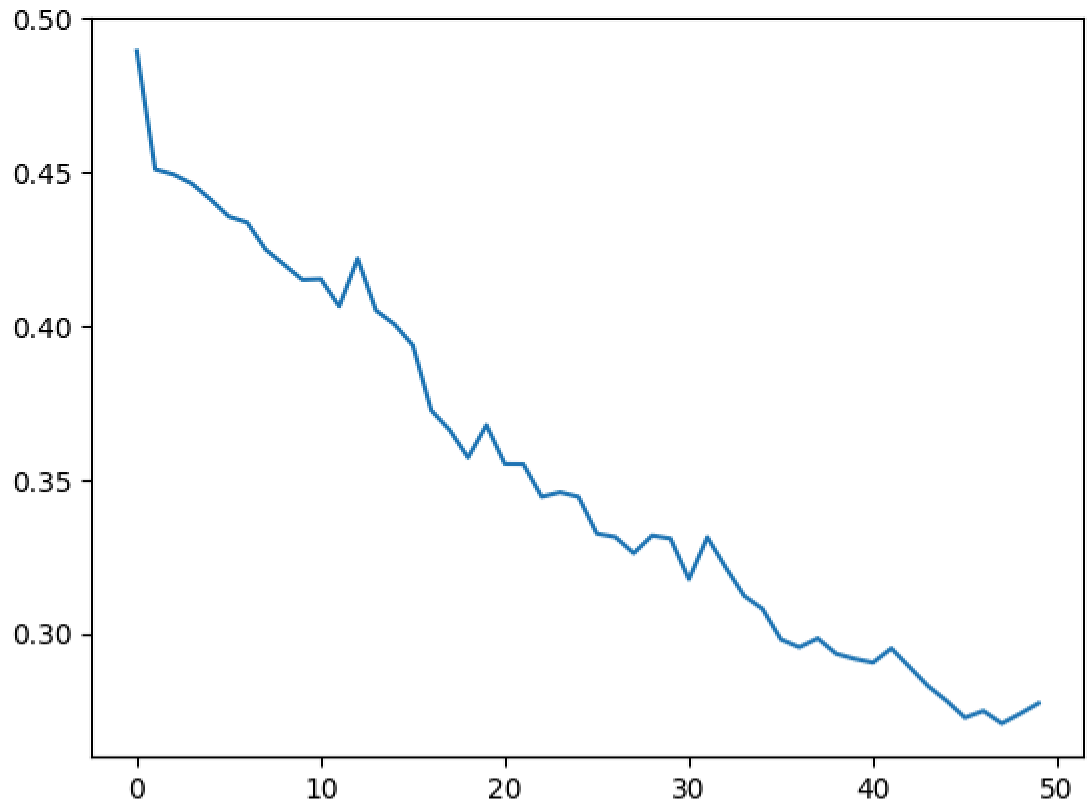
```
Epoch 1/50

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1698385783.436050  470386 device_compiler.h:186] Compiled cluster using XLA!  This line is logged at
most once for the lifetime of the process.

238/238 [==============================] - 4s 5ms/step - loss: 0.4896 - duration_loss: 0.2237 - pitch_loss: 4.25
69 - step_loss: 0.0530
Epoch 2/50
238/238 [==============================] - 1s 4ms/step - loss: 0.4510 - duration_loss: 0.2010 - pitch_loss: 4.07
47 - step_loss: 0.0463
Epoch 3/50
238/238 [==============================] - 1s 4ms/step - loss: 0.4494 - duration_loss: 0.2006 - pitch_loss: 4.05
95 - step_loss: 0.0457
Epoch 4/50
238/238 [==============================] - 1s 4ms/step - loss: 0.4464 - duration_loss: 0.1989 - pitch_loss: 4.04
58 - step_loss: 0.0451
Epoch 5/50
238/238 [==============================] - 1s 4ms/step - loss: 0.4413 - duration_loss: 0.1948 - pitch_loss: 4.03
42 - step loss: 0.0448
```

In [42]:

```
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 (https://www.tensorflow.org/text/tutorials/text_generation)](https://www.tensorflow.org/text/tutorials/text_generation).

In [43]:

```python
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 [44]:

```python
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 386ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 41ms/step
```

In [45]:
```
generated_notes.head(10)
```

Out[45]:

	pitch	step	duration	start	end
0	47	0.097688	0.179285	0.097688	0.276973
1	72	0.858761	0.163204	0.956448	1.119652
2	84	1.089159	0.000000	2.045608	2.045608
3	84	1.095052	0.000000	3.140660	3.140660
4	89	1.108157	0.000000	4.248817	4.248817
5	89	1.118506	0.000000	5.367323	5.367323
6	84	1.135083	0.000000	6.502405	6.502405
7	90	1.128108	0.000000	7.630513	7.630513
8	84	1.129703	0.000000	8.760216	8.760216
9	84	1.155163	0.019010	9.915379	9.934389

In [46]:
```
out_file = 'output.mid'
out_pm = notes_to_midi(
    generated_notes, out_file=out_file, instrument_name=instrument_name)
display_audio(out_pm)
```

```
fluidsynth: warning: SDL2 not initialized, SDL2 audio driver won't be usable
fluidsynth: error: Unknown integer parameter 'synth.sample-rate'
```

Out[46]:

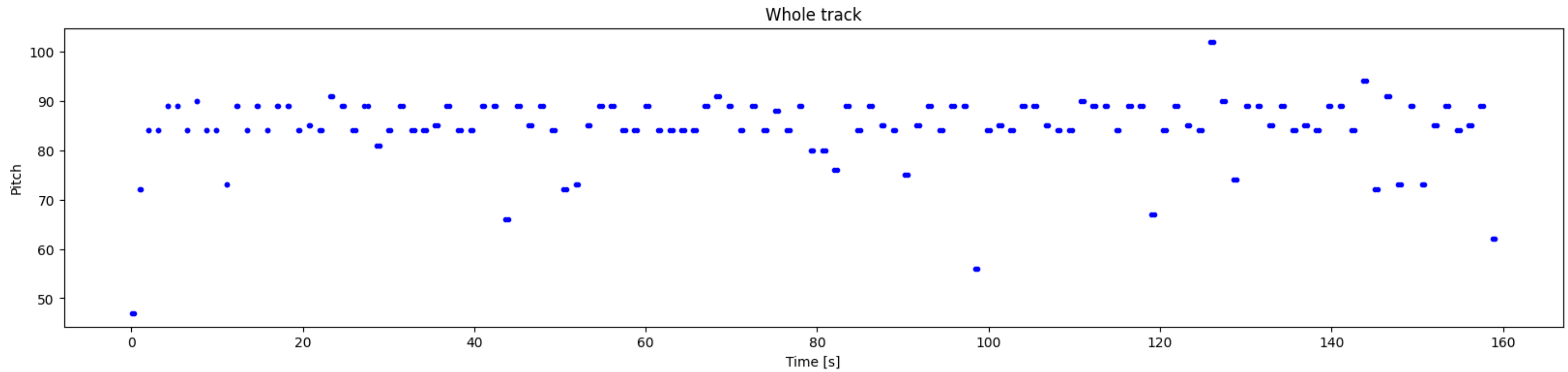
0:00 / 0:00

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

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

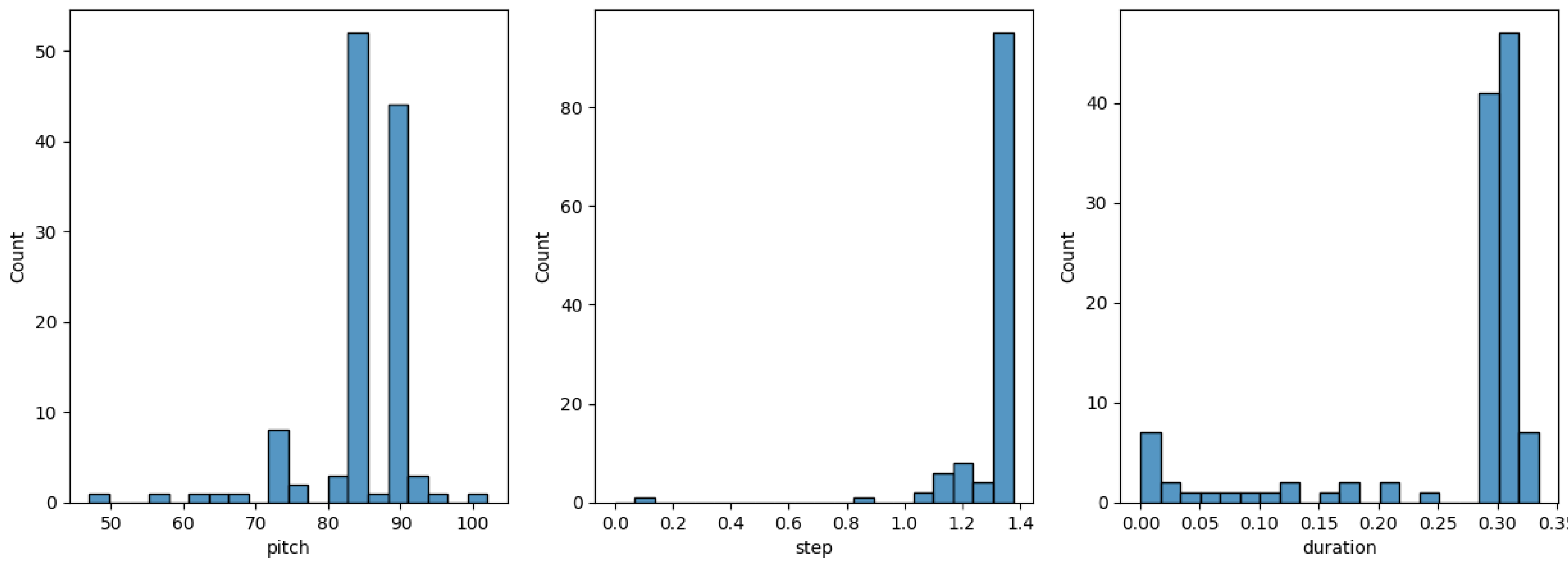
Visualize the generated notes.

In [47]:
```
plot_piano_roll(generated_notes)
```



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

In [48]:
```
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` .

Next steps

This tutorial demonstrated the mechanics of using an RNN to generate sequences of notes from a dataset of MIDI files. To learn more, you can visit the closely related Text generation with an RNN (https://www.tensorflow.org/text/tutorials/text_generation) tutorial, which contains additional diagrams and explanations.

One of the alternatives to using RNNs for music generation is using GANs. Rather than generating audio, a GAN-based approach can generate an entire sequence in parallel. The Magenta team has done impressive work on this approach with GANSynth (https://magenta.tensorflow.org/gansynth). You can also find many wonderful music and art projects and open-source code on Magenta project website (https://magenta.tensorflow.org/).