```
import sys
import tensorflow as tf
import matplotlib.pyplot as plt
from pathlib import Path
```

§ 15.5. Exercises

10) Download the [Bach chorales] dataset and unzip it. It is composed of 382 chorales composed by Johann Sebastian Bach. Each chorale is 100 to 640 time steps long, and each time step contains 4 integers, where each integer corresponds to a note's index on a piano (except for the value 0, which means that no note is played). Train a model--recurrent, convolutional, or both--that can predict the next time step (4 notes), given a sequence of time steps from a chorale. Then use this model to generate Bach-like music, 1 note at a time: you can do this by giving the model the start of a chorale and asking it to predict the next time step, then appending these time steps to the input sequence and asking the model for the next note, and so on. Also make sure to check out Google's Coconet model, which was used for a nice Google doodle about Bach.

```
tf.keras.utils.get_file(
    "jsb_chorales.tgz",
    "https://github.com/ageron/data/raw/main/jsb_chorales.tgz",
    cache_dir=".",
    extract=True)
```

```
Downloading data from https://github.com/ageron/data/raw/main/jsb_chorales.tgz
117793/117793 [==============================] - 0s 0us/step
'./datasets/jsb_chorales.tgz'
```

```
jsb_chorales_dir = Path("datasets/jsb_chorales")
train_files = sorted(jsb_chorales_dir.glob("train/chorale_*.csv"))
valid_files = sorted(jsb_chorales_dir.glob("valid/chorale_*.csv"))
test_files = sorted(jsb_chorales_dir.glob("test/chorale_*.csv"))
```

```
import pandas as pd
```

```
def load_chorales(filepaths):
  return [pd.read_csv(filepath).values.tolist() for filepath in filepaths]

train_chorales = load_chorales(train_files)
valid_chorales = load_chorales(valid_files)
test_chorales = load_chorales(test_files)
```

```
train_chorales[0]
```

```
       [77, 65, 60, 53],
       [77, 65, 60, 53],
       [74, 65, 58, 58],
       [74, 65, 58, 58],
       [74, 65, 58, 58],
       [74, 65, 58, 58],
       [72, 67, 58, 51],
       [72, 67, 58, 51],
       [72, 67, 58, 51],
       [72, 67, 58, 51],
       [72, 65, 57, 53],
       [72, 65, 57, 53],
       [72, 65, 57, 53],
       [72, 65, 57, 53],
       [70, 65, 62, 46],
       [70, 65, 62, 46],
       [70, 65, 62, 46],
       [70, 65, 62, 46],
       [70, 65, 62, 46],
       [70, 65, 62, 46],
       [70, 65, 62, 46],
       [70, 65, 62, 46]]
```

```
notes = set()
for chorales in (train_chorales, valid_chorales, test_chorales):
  for chorale in chorales:
    for chord in chorale:
      notes |= set(chord)

n_notes = len(notes)
min_note = min(notes - {0})
max_note = max(notes)

assert min_note == 36
assert max_note == 81
```

```
from IPython.display import Audio
```

```
def notes_to_frequencies(notes):
  # Frequency doubles when you go up 1 octave; there are 12 semi-tones per octave.
  # Note A on octave 4 is 440 Hz, and it is note number 69.
  return 2 ** ((np.array(notes) - 69) / 12) * 440

def frequencies_to_samples(frequencies, tempo, sample_rate):
  note_duration = 60 / tempo # the tempo is measured in beats/minutes
  # To reduce click sound at every beat, we round the frequencies to
  # try to get the samples close to 0 at the end of each note.
  frequencies = (note_duration * frequencies).round() / note_duration
  n_samples = int(note_duration * sample_rate)
  time = np.linspace(0, note_duration, n_samples)
  sine_waves = np.sin(2 * np.pi * frequencies.reshape(-1, 1) * time)
  # Removing all notes with frequencies ≤ 9 Hz (includes note 0 = silence)
  sine_waves *= (frequencies > 9.).reshape(-1, 1)
  return sine_waves.reshape(-1)

def chords_to_samples(chords, tempo, sample_rate):
  freqs = notes_to_frequencies(chords)
  freqs = np.r_[freqs, freqs[-1:]] # make last note a bit longer
  merged = np.mean([frequencies_to_samples(melody, tempo, sample_rate)
                    for melody in freqs.T], axis=0)
  n_fade_out_samples = sample_rate * 60 // tempo # fade out last note
  fade_out = np.linspace(1., 0., n_fade_out_samples)**2
  merged[-n_fade_out_samples:] *= fade_out
  return merged

def play_chords(chords, tempo=160, amplitude=0.1, sample_rate=44100, filepath=None):
  samples = amplitude * chords_to_samples(chords, tempo, sample_rate)
  if filepath:
    from scipy.io import wavfile
    samples = (2**15 * samples).astype(np.int16)
    wavfile.write(filepath, sample_rate, samples)
    return display(Audio(filepath))
  else:
    return display(Audio(samples, rate=sample_rate))
```

```
import numpy as np
```

```
for index in range(3):
  play_chords(train_chorales[index])
```

0:00 / 1:12

0:00 / 1:25

0:00 / 1:18

```
def create_target(batch):
  X = batch[:, :-1]
  Y = batch[:, 1:] # predict next note in each arpegio, at each step
  return X, Y

def preprocess(window):
  window = tf.where(window == 0, window, window - min_note + 1) # shift values
  return tf.reshape(window, [-1]) # convert to arpegio

def bach_dataset(chorales, batch_size=32, shuffle_buffer_size=None,
                 window_size=32, window_shift=16, cache=True):
  def batch_window(window):
    return window.batch(window_size + 1)

  def to_windows(chorale):
    dataset = tf.data.Dataset.from_tensor_slices(chorale)
    dataset = dataset.window(window_size + 1, window_shift, drop_remainder=True)
    return dataset.flat_map(batch_window)

  chorales = tf.ragged.constant(chorales, ragged_rank=1)
  dataset = tf.data.Dataset.from_tensor_slices(chorales)
  dataset = dataset.flat_map(to_windows).map(preprocess)
  if cache:
    dataset = dataset.cache()
  if shuffle_buffer_size:
    dataset = dataset.shuffle(shuffle_buffer_size)
  dataset = dataset.batch(batch_size)
  dataset = dataset.map(create_target)
  return dataset.prefetch(1)
```

```
train_set = bach_dataset(train_chorales, shuffle_buffer_size=1000)
valid_set = bach_dataset(valid_chorales)
test_set = bach_dataset(test_chorales)
```

```
n_embedding_dims = 5

model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_notes, output_dim=n_embedding_dims, input_shape=[None]),
    tf.keras.layers.Conv1D(32, kernel_size=2, padding="causal", activation="relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv1D(48, kernel_size=2, padding="causal", activation="relu", dilation_rate=2),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv1D(64, kernel_size=2, padding="causal", activation="relu", dilation_rate=4),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv1D(96, kernel_size=2, padding="causal", activation="relu", dilation_rate=8),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LSTM(256, return_sequences=True),
    tf.keras.layers.Dense(n_notes, activation="softmax")
])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 5)           235

 conv1d (Conv1D)             (None, None, 32)          352

 batch_normalization (Batch  (None, None, 32)          128
 Normalization)

 conv1d_1 (Conv1D)           (None, None, 48)          3120

 batch_normalization_1 (Bat  (None, None, 48)          192
 chNormalization)

 conv1d_2 (Conv1D)           (None, None, 64)          6208

 batch_normalization_2 (Bat  (None, None, 64)          256
```

```
 chNormalization)

 conv1d_3 (Conv1D)           (None, None, 96)          12384

 batch_normalization_3 (Bat  (None, None, 96)          384
 chNormalization)

 lstm (LSTM)                 (None, None, 256)         361472

 dense (Dense)               (None, None, 47)          12079

=================================================================
Total params: 396810 (1.51 MB)
Trainable params: 396330 (1.51 MB)
Non-trainable params: 480 (1.88 KB)
_________________________________________________________________
```

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-3)
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=optimizer,
              metrics=["accuracy"])
model.fit(train_set,
          epochs=20,
          validation_data=valid_set)
```

```
Epoch 1/20
98/98 [==============================] - 18s 63ms/step - loss: 1.9527 - accuracy: 0.5099 - val_loss: 3.6761 - val_accuracy: 0.0785
Epoch 2/20
98/98 [==============================] - 2s 23ms/step - loss: 0.8912 - accuracy: 0.7649 - val_loss: 3.6367 - val_accuracy: 0.1121
Epoch 3/20
98/98 [==============================] - 2s 19ms/step - loss: 0.7387 - accuracy: 0.7953 - val_loss: 3.4599 - val_accuracy: 0.1199
Epoch 4/20
98/98 [==============================] - 2s 17ms/step - loss: 0.6613 - accuracy: 0.8111 - val_loss: 2.6387 - val_accuracy: 0.2666
Epoch 5/20
98/98 [==============================] - 2s 20ms/step - loss: 0.6070 - accuracy: 0.8232 - val_loss: 1.6374 - val_accuracy: 0.5430
Epoch 6/20
98/98 [==============================] - 2s 17ms/step - loss: 0.5644 - accuracy: 0.8333 - val_loss: 0.9486 - val_accuracy: 0.7330
Epoch 7/20
98/98 [==============================] - 2s 22ms/step - loss: 0.5279 - accuracy: 0.8418 - val_loss: 0.6664 - val_accuracy: 0.8066
Epoch 8/20
98/98 [==============================] - 2s 21ms/step - loss: 0.4949 - accuracy: 0.8504 - val_loss: 0.6134 - val_accuracy: 0.8208
Epoch 9/20
98/98 [==============================] - 2s 18ms/step - loss: 0.4671 - accuracy: 0.8572 - val_loss: 0.5897 - val_accuracy: 0.8265
Epoch 10/20
98/98 [==============================] - 2s 21ms/step - loss: 0.4411 - accuracy: 0.8645 - val_loss: 0.5927 - val_accuracy: 0.8250
Epoch 11/20
98/98 [==============================] - 2s 18ms/step - loss: 0.4172 - accuracy: 0.8711 - val_loss: 0.5844 - val_accuracy: 0.8274
Epoch 12/20
98/98 [==============================] - 2s 18ms/step - loss: 0.3941 - accuracy: 0.8776 - val_loss: 0.5991 - val_accuracy: 0.8260
Epoch 13/20
98/98 [==============================] - 3s 27ms/step - loss: 0.4073 - accuracy: 0.8728 - val_loss: 0.6330 - val_accuracy: 0.8179
Epoch 14/20
98/98 [==============================] - 2s 22ms/step - loss: 0.3854 - accuracy: 0.8786 - val_loss: 0.5873 - val_accuracy: 0.8289
Epoch 15/20
98/98 [==============================] - 2s 18ms/step - loss: 0.3487 - accuracy: 0.8903 - val_loss: 0.5951 - val_accuracy: 0.8263
Epoch 16/20
98/98 [==============================] - 2s 18ms/step - loss: 0.3300 - accuracy: 0.8964 - val_loss: 0.5989 - val_accuracy: 0.8258
Epoch 17/20
98/98 [==============================] - 2s 18ms/step - loss: 0.3116 - accuracy: 0.9023 - val_loss: 0.6040 - val_accuracy: 0.8282
Epoch 18/20
98/98 [==============================] - 2s 18ms/step - loss: 0.2959 - accuracy: 0.9073 - val_loss: 0.6021 - val_accuracy: 0.8266
Epoch 19/20
98/98 [==============================] - 2s 21ms/step - loss: 0.2826 - accuracy: 0.9114 - val_loss: 0.6120 - val_accuracy: 0.8263
Epoch 20/20
98/98 [==============================] - 2s 19ms/step - loss: 0.2670 - accuracy: 0.9163 - val_loss: 0.6291 - val_accuracy: 0.8203
<keras.src.callbacks.History at 0x7c955875ee90>
```

```
model.save("my_bach_model", save_format="tf")
model.evaluate(test_set)
```

```
34/34 [==============================] - 1s 17ms/step - loss: 0.6299 - accuracy: 0.8207
[0.6299295425415039, 0.8207215666770935]
```

```
def generate_chorale(model, seed_chords, length):
  arpegio = preprocess(tf.constant(seed_chords, dtype=tf.int64))
  arpegio = tf.reshape(arpegio, [1, -1])
  for chord in range(length):
    for note in range(4):
      next_note = model.predict(arpegio, verbose=0).argmax(axis=-1)[:1, -1:]
      arpegio = tf.concat([arpegio, next_note], axis=1)
  arpegio = tf.where(arpegio == 0, arpegio, arpegio + min_note - 1)
  return tf.reshape(arpegio, shape=[-1, 4])
```

```
seed_chords = test_chorales[2][:8]
play_chords(seed_chords, amplitude=0.2)
```

0:00 / 0:03

```
new_chorale = generate_chorale(model, seed_chords, 56)
play_chords(new_chorale)
```

0:00 / 0:24

```
def generate_chorale_v2(model, seed_chords, length, temperature=1):
  arpegio = preprocess(tf.constant(seed_chords, dtype=tf.int64))
  arpegio = tf.reshape(arpegio, [1, -1])
  for chord in range(length):
    for note in range(4):
      next_note_probas = model.predict(arpegio)[0, -1:]
      rescaled_logits = tf.math.log(next_note_probas) / temperature
      next_note = tf.random.categorical(rescaled_logits, num_samples=1)
      arpegio = tf.concat([arpegio, next_note], axis=1)
  arpegio = tf.where(arpegio == 0, arpegio, arpegio + min_note - 1)
  return tf.reshape(arpegio, shape=[-1, 4])
```

```
new_chorale_v2_cold = generate_chorale_v2(model, seed_chords, 56, temperature=0.8)
play_chords(new_chorale_v2_cold, filepath="bach_cold.wav")
```

```
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
```

0:00 / 0:24