Music Generation with RNNs

2.1 Dependencies

First, let's download the course repository, install dependencies, and import the relevant packages we'll need for this lab.

```
# Import Tensorflow 2.0
%tensorflow_version 2.x
import tensorflow as tf
```

```
    Colab only includes TensorFlow 2.x; %tensorflow_version has no effect.
```

```
# Download and import the MIT Introduction to Deep Learning package
!pip install mitdeeplearning
import mitdeeplearning as mdl
```

```
    Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
    Collecting mitdeeplearning
      Downloading mitdeeplearning-0.3.0.tar.gz (2.1 MB)
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 28.8 MB/s eta 0:00:00
      Preparing metadata (setup.py) ... done
    Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from mitdeeplearning) (1.22.4)
    Requirement already satisfied: regex in /usr/local/lib/python3.10/dist-packages (from mitdeeplearning) (2022.10.31)
    Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from mitdeeplearning) (4.65.0)
    Requirement already satisfied: gym in /usr/local/lib/python3.10/dist-packages (from mitdeeplearning) (0.25.2)
    Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.10/dist-packages (from gym->mitdeeplearning) (0.0.8)
    Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gym->mitdeeplearning) (2.2.1)
    Building wheels for collected packages: mitdeeplearning
      Building wheel for mitdeeplearning (setup.py) ... done
      Created wheel for mitdeeplearning: filename=mitdeeplearning-0.3.0-py3-none-any.whl size=2117316 sha256=919a3c6dd56637cea47b7508dbda6aa
      Stored in directory: /root/.cache/pip/wheels/9c/9f/b5/0e31f83bc60a83625e37429f923934e26802d0d68cf3ef3216
    Successfully built mitdeeplearning
    Installing collected packages: mitdeeplearning
    Successfully installed mitdeeplearning-0.3.0
```

```
import numpy as np
import os
import time
import functools
from IPython import display as ipythondisplay
from tqdm import tqdm
!apt-get install abcmidi timidity > /dev/null 2>&1
```

```
# Check that we are using a GPU, if not switch runtimes
# using Runtime > Change Runtime Type > GPU
assert len(tf.config.list_physical_devices('GPU')) > 0
```

```
    ---------------------------------------------------------------------------
    AssertionError                            Traceback (most recent call last)
    <ipython-input-6-b69c7ea72189> in <cell line: 3>()
          1 # Check that we are using a GPU, if not switch runtimes
          2 # using Runtime > Change Runtime Type > GPU
    ----> 3 assert len(tf.config.list_physical_devices('GPU')) > 0

    AssertionError:
```

SEARCH STACK OVERFLOW

2.2 Dataset

```
# Download the dataset
songs = mdl.lab1.load_training_data()
```

```
    Found 817 songs in text
```

```
# Print one of the songs to inspect it in greater detail!
example_song = songs[0]
```

```
print("\nExample song: ")
print(example_song)
```

```
Example song:
X:1
T:Alexander's
Z: id:dc-hornpipe-1
M:C|
L:1/8
K:D Major
(3ABc|dAFA DFAd|fdcd FAdf|gfge fefd|(3efe (3dcB A2 (3ABc|!
dAFA DFAd|fdcd FAdf|gfge fefd|(3efe dc d2:|!
AG|FAdA FAdA|GBdB GBdB|Acec Acec|dfaf gecA|!
FAdA FAdA|GBdB GBdB|Aceg fefd|(3efe dc d2:|!
```

```
# Convert the ABC notation to audio file and listen to it
mdl.lab1.play_song(example_song)
```

0:01 / 1:05

```
# Join our list of song strings into a single string containing all songs
songs_joined = "\n\n".join(songs)

# Find all unique characters in the joined string
vocab = sorted(set(songs_joined))
print("There are", len(vocab), "unique characters in the dataset")
```

```
There are 83 unique characters in the dataset
```

2.3 Process the dataset for the learning task

Vectorize the text

```
### Define numerical representation of text ###

# Create a mapping from character to unique index.
# For example, to get the index of the character "d",
#   we can evaluate `char2idx["d"]`.
char2idx = {u:i for i, u in enumerate(vocab)}

# Create a mapping from indices to characters. This is
#   the inverse of char2idx and allows us to convert back
#   from unique index to the character in our vocabulary.
idx2char = np.array(vocab)
```

This gives us an integer representation for each character. Observe that the unique characters (i.e., our vocabulary) in the text are mapped as indices from 0 to len(unique). Let's take a peek at this numerical representation of our dataset:

```
print('{')
for char,_ in zip(char2idx, range(20)):
    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))
print('  ...\n}')
```

```
{
  '\n':   0,
  ' ' :   1,
  '!' :   2,
  '"' :   3,
  '#' :   4,
  "'" :   5,
  '(' :   6,
  ')' :   7,
  ',' :   8,
  '-' :   9,
  '.' :  10,
  '/' :  11,
  '0' :  12,
  '1' :  13,
  '2' :  14,
        '3' :  15,
        '4' :  16,
        '5' :  17,
        '6' :  18,
        '7' :  19,
        ...
      }
```

```
### Vectorize the songs string ###

'''TODO: Write a function to convert the all songs string to a vectorized
    (i.e., numeric) representation. Use the appropriate mapping
    above to convert from vocab characters to the corresponding indices.

  NOTE: the output of the `vectorize_string` function
  should be a np.array with `N` elements, where `N` is
  the number of characters in the input string
'''
def vectorize_string(string):
  vectorized_output = np.array([char2idx[char] for char in string])
  return vectorized_output

# def vectorize_string(string):
  # TODO

vectorized_songs = vectorize_string(songs_joined)
```

We can also look at how the first part of the text is mapped to an integer representation:

```
print ('{} ---- characters mapped to int ----> {}'.format(repr(songs_joined[:10]), vectorized_songs[:10]))
# check that vectorized_songs is a numpy array
assert isinstance(vectorized_songs, np.ndarray), "returned result should be a numpy array"
```

```
     'X:1\nT:Alex' ---- characters mapped to int ----> [49 22 13  0 45 22 26 67 60 79]
```

Create training examples and targets

The batch method will then let us convert this stream of character indices to sequences of the desired size.

```
### Batch definition to create training examples ###

def get_batch(vectorized_songs, seq_length, batch_size):
  # the length of the vectorized songs string
  n = vectorized_songs.shape[0] - 1
  # randomly choose the starting indices for the examples in the training batch
  idx = np.random.choice(n-seq_length, batch_size)

  '''TODO: construct a list of input sequences for the training batch'''
  input_batch = [vectorized_songs[i:i+seq_length] for i in idx]
  '''TODO: construct a list of output sequences for the training batch'''
  output_batch = [vectorized_songs[i+1:i+seq_length+1] for i in idx]

  # x_batch, y_batch provide the true inputs and targets for network training
  x_batch = np.reshape(input_batch, [batch_size, seq_length])
  y_batch = np.reshape(output_batch, [batch_size, seq_length])
  return x_batch, y_batch


# Perform some simple tests to make sure your batch function is working properly!
test_args = (vectorized_songs, 10, 2)
if not mdl.lab1.test_batch_func_types(get_batch, test_args) or \
   not mdl.lab1.test_batch_func_shapes(get_batch, test_args) or \
   not mdl.lab1.test_batch_func_next_step(get_batch, test_args):
   print("======\n[FAIL] could not pass tests")
else:
   print("======\n[PASS] passed all tests!")
```

```
     [PASS] test_batch_func_types
     [PASS] test_batch_func_shapes
     [PASS] test_batch_func_next_step
     ======
     [PASS] passed all tests!
```

We can make this concrete by taking a look at how this works over the first several characters in our text:

```
x_batch, y_batch = get_batch(vectorized_songs, seq_length=5, batch_size=1)

for i, (input_idx, target_idx) in enumerate(zip(np.squeeze(x_batch), np.squeeze(y_batch))):
    print("Step {:3d}".format(i))
    print("  input: {} ({:s})".format(input_idx, repr(idx2char[input_idx])))
    print("  expected output: {} ({:s})".format(target_idx, repr(idx2char[target_idx])))
```

```
Step   0
  input: 59 ('d')
  expected output: 82 ('|')
Step   1
  input: 82 ('|')
  expected output: 60 ('e')
Step   2
  input: 60 ('e')
  expected output: 14 ('2')
Step   3
  input: 14 ('2')
  expected output: 59 ('d')
Step   4
  input: 59 ('d')
  expected output: 1 (' ')
```

2.4 The Recurrent Neural Network (RNN) model

Define the RNN model

Now, we will define a function that we will use to actually build the model.

```
def LSTM(rnn_units):
  return tf.keras.layers.LSTM(
    rnn_units,
    return_sequences=True,
    recurrent_initializer='glorot_uniform',
    recurrent_activation='sigmoid',
    stateful=True,
  )
```

The time has come! Fill in the TODOs to define the RNN model within the build_model function, and then call the function you just defined to instantiate the model!

```
### Defining the RNN Model ###

'''TODO: Add LSTM and Dense layers to define the RNN model using the Sequential API.'''
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential([
    # Layer 1: Embedding layer to transform indices into dense vectors
    #   of a fixed embedding size
    tf.keras.layers.Embedding(vocab_size, embedding_dim, batch_input_shape=[batch_size, None]),

    # Layer 2: LSTM with `rnn_units` number of units.
    # TODO: Call the LSTM function defined above to add this layer.
    LSTM(rnn_units),

    # Layer 3: Dense (fully-connected) layer that transforms the LSTM output
    #   into the vocabulary size.
    # TODO: Add the Dense layer.
    tf.keras.layers.Dense(units = vocab_size)
  ])

  return model

# Build a simple model with default hyperparameters. You will get the
#   chance to change these later.
model = build_model(len(vocab), embedding_dim=256, rnn_units=1024, batch_size=32)
```

Test out the RNN model

```
model.summary()

     Model: "sequential"
     _________________________________________________________________
      Layer (type)                Output Shape              Param #
     =================================================================
      embedding (Embedding)       (32, None, 256)           21248

      lstm (LSTM)                 (32, None, 1024)          5246976

      dense (Dense)               (32, None, 83)            85075

     =================================================================
     Total params: 5,353,299
     Trainable params: 5,353,299
     Non-trainable params: 0
     _________________________________________________________________
```

We can also quickly check the dimensionality of our output, using a sequence length of 100. Note that the model can be run on inputs of any length.

```
x, y = get_batch(vectorized_songs, seq_length=100, batch_size=32)
pred = model(x)
print("Input shape:      ", x.shape, " # (batch_size, sequence_length)")
print("Prediction shape: ", pred.shape, "# (batch_size, sequence_length, vocab_size)")

     Input shape:       (32, 100)  # (batch_size, sequence_length)
     Prediction shape:  (32, 100, 83) # (batch_size, sequence_length, vocab_size)
```

Predictions from the untrained model

```
sampled_indices = tf.random.categorical(pred[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()
sampled_indices

     array([20,  2, 14, 44, 37,  1,  8, 55, 64, 50, 51, 48, 43, 34, 60, 15, 69,
             4, 64, 10, 72, 49, 64, 81, 21, 19, 65, 28, 28, 38, 54,  8, 18, 60,
            67,  8,  1, 22, 70, 72, 28, 12, 49, 61, 52, 43,  8, 73, 64, 40, 40,
            48, 12, 53, 79, 82, 81,  3, 76, 19, 33,  8, 75,  6, 58, 30,  9, 28,
            37, 73, 61, 36, 72, 28, 38, 46, 12, 73, 30, 47, 15, 64, 53,  6, 15,
            44, 43, 47, 43, 23, 50, 48, 78, 60, 51, 47, 71, 23, 12, 46])
```

We can now decode these to see the text predicted by the untrained model:

```
print("Input: \n", repr("".join(idx2char[x[0]])))
print()
print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices])))

     Input:
      '|AFF dFF|ABc def|!\n[1 eBe fdB|eBe fdB|AFF dFF|FED E2:|!\n[2 edB BAF|AFE E2F|AFF dFF|FED E2|]!\n\nX:165\n'

     Next Char Predictions:
      '8!2SL ,_iYZWRIe3n#i.qXiz97jCCM^,6el, :oqC0Xf[R,riOOW0]x|z"u7H,t(cE-CLrfKqCMU0rEV3i](3SRVR<YWweZVp<0U'
```

2.5 Training the model: loss and training operations

Let's first compute the loss using our example predictions from the untrained model:

```
### Defining the loss function ###

'''TODO: define the loss function to compute and return the loss between
    the true labels and predictions (logits). Set the argument from_logits=True.'''
def compute_loss(labels, logits):
  loss = tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True) # TODO
  return loss
```

```
'''TODO: compute the loss using the true next characters from the example batch
    and the predictions from the untrained model several cells above'''
example_batch_loss = compute_loss(y, pred) # TODO

print("Prediction shape: ", pred.shape, " # (batch_size, sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())
```

```
     Prediction shape:  (32, 100, 83)  # (batch_size, sequence_length, vocab_size)
     scalar_loss:       4.4190063
```

Let's start by defining some hyperparameters for training the model. To start, we have provided some reasonable values for some of the parameters. It is up to you to use what we've learned in class to help optimize the parameter selection here!

```
### Hyperparameter setting and optimization ###

# Optimization parameters:
num_training_iterations = 2000  # Increase this to train longer
batch_size = 4  # Experiment between 1 and 64
seq_length = 100  # Experiment between 50 and 500
learning_rate = 5e-3  # Experiment between 1e-5 and 1e-1

# Model parameters:
vocab_size = len(vocab)
embedding_dim = 256
rnn_units = 1024  # Experiment between 1 and 2048

# Checkpoint location:
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "my_ckpt")
```

```
### Define optimizer and training operation ###

'''TODO: instantiate a new model for training using the `build_model`
  function and the hyperparameters created above.'''
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size)

'''TODO: instantiate an optimizer with its learning rate.
  Checkout the tensorflow website for a list of supported optimizers.
  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/
  Try using the Adam optimizer to start.'''
optimizer = tf.keras.optimizers.Adam(learning_rate)

@tf.function
def train_step(x, y):
  # Use tf.GradientTape()
  with tf.GradientTape() as tape:

    '''TODO: feed the current input into the model and generate predictions'''
    y_hat = model(x)

    '''TODO: compute the loss!'''
    loss = compute_loss(y, y_hat)

  # Now, compute the gradients
  '''TODO: complete the function call for gradient computation.
      Remember that we want the gradient of the loss with respect all
      of the model parameters.
      HINT: use `model.trainable_variables` to get a list of all model
      parameters.'''
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the optimizer so it can update the model accordingly
  optimizer.apply_gradients(zip(grads, model.trainable_variables))
  return loss

##################
# Begin training!#
##################

history = []
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Loss')
if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
```

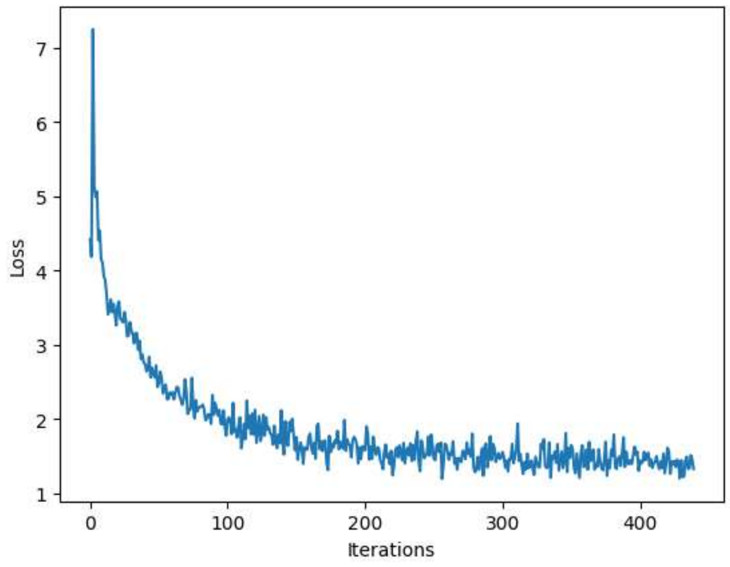
```
for iter in tqdm(range(num_training_iterations)):

  # Grab a batch and propagate it through the network
  x_batch, y_batch = get_batch(vectorized_songs, seq_length, batch_size)
  loss = train_step(x_batch, y_batch)

  # Update the progress bar
  history.append(loss.numpy().mean())
  plotter.plot(history)

  # Update the model with the changed weights!
  if iter % 100 == 0:
    model.save_weights(checkpoint_prefix)

# Save the trained model and the weights
model.save_weights(checkpoint_prefix)
```



```
22%|██          | 440/2000 [35:58<2:01:54,  4.69s/it]
```

2.6 Generate music using the RNN model

Restore the latest checkpoint

```
'''TODO: Rebuild the model using a batch_size=1'''
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

# Restore the model weights for the last checkpoint after training
model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
model.build(tf.TensorShape([1, None]))

model.summary()
```

The prediction procedure

```
### Prediction of a generated song ###

def generate_text(model, start_string, generation_length=1000):
  # Evaluation step (generating ABC text using the learned RNN model)

  '''TODO: convert the start string to numbers (vectorize)'''
  input_eval = [char2idx[num] for num in start_string]
  input_eval = tf.expand_dims(input_eval, 0)

  # Empty string to store our results
  text_generated = []
  # Here batch size == 1
  model.reset_states()
  tqdm._instances.clear()

  for i in tqdm(range(generation_length)):
      '''TODO: evaluate the inputs and generate the next character predictions'''
      predictions = model(input_eval)

      # Remove the batch dimension
      predictions = tf.squeeze(predictions, 0)

      '''TODO: use a multinomial distribution to sample'''
      predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()

      # Pass the prediction along with the previous hidden state
      #   as the next inputs to the model
      input_eval = tf.expand_dims([predicted_id], 0)

      '''TODO: add the predicted character to the generated text!'''
      # Hint: consider what format the prediction is in vs. the output
      text_generated.append(idx2char[predicted_id])

  return (start_string + ''.join(text_generated))
```

```
'''TODO: Use the model and the function defined above to generate ABC format text of length 1000!
    As you may notice, ABC files start with "X" - this may be a good start string.'''
generated_text = generate_text(model, start_string="X", generation_length=30000) # TODO
```

Play back the generated music!

```
### Play back generated songs ###

generated_songs = mdl.lab1.extract_song_snippet(generated_text)
for i, song in enumerate(generated_songs):
  # Synthesize the waveform from a song
  mdl.lab1.play_song(song)
  print(song, end="\n\n\n\n")
  #print(song)
```

Double-click (or enter) to edit

● Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.