1 Recurrent Neural Networks

Lab Objective: *Recurrent Neural Networks are powerful machine learning algorithms that accept sequences as inputs and can process temporal data. In this lab, we generate a Mozart-like piano sonata using the Long Short-term Memory RNN.*

Recurrent Neural Networks

Convolution neural networks work well for problems like image classification because the inputs and outputs are independent and of fixed size. However many problems do not have these constraints. For example, what if we want to predict the next word in a sentence? This is clearly not independent since the output for one iteration becomes the input for the next iteration. Recurrent neural networks, RNNs, address these issues by using sequences as the input, output, or both, allowing for temporal dynamic behavior. They perform the same task for every element of the sequence, hence the use of recurrent in the name. Each task uses the input as well as recent previous information, called memory, from the network to create the output. Even if the input is not sequential, it is possible to process it sequentially using RNNs, resulting in powerful learning algorithms.

Data

For this lab, we will use Google Colab and its GPU capability. To enable the GPU in a Colab notebook, select the *Runtime* tab and then *Change runtime type*. This will open a popup called `Notebook Settings`. Under *Hardware Settings*, select GPU. We recommend mounting a Google Drive to the notebook to make loading and saving data easier. This will save the data if the notebook is disconnected; if the data is saved to the Colab directory, the entire project must be rerun. To mount a Google Drive, run

```
>>> from google.colab import drive
>>> drive.mount('/content/drive')
```

Follow the instructions in the cell to authorize the account.

If you need to refresh your drive connection, you can run

```
>>> drive.mount('/content/drive', force_remount = True).
```

Download Data

We will be using a collection of piano sonatas from Mozart as the data. For easy download, run the following function to save the files to `filepath` in the Google Drive folder.

```
def download_data(filepath):
    if not os.path.exists(os.path.join(filepath, 'mozart_sonatas.tar.gz')):
      datasets.utils.download_url('https://github.com/Foundations-of-Applied-↩
          Mathematics/Data/raw/master/Volume3/mozart_sonatas.tar.gz', filepath,↩
           'mozart_sonatas.tar.gz', None)

      print('Extracting {}'.format('mozart_sonatas.tar.gz'))
      gzip_path = os.path.join(filepath, 'mozart_sonatas.tar.gz')
      with open(gzip_path.replace('.gz', ''), 'wb') as out_f, gzip.GzipFile(↩
          gzip_path) as zip_f:
        out_f.write(zip_f.read())

      print('Untarring {}'.format('mozart_sonatas.tar'))
      tar_path = os.path.join(filepath,'mozart_sonatas.tar')
      z = tarfile.TarFile(tar_path)
      z.extractall(tar_path.replace('.tar', ''))

>> download_data('drive/MyDrive/Colab')
Downloading https://raw.githubusercontent.com/Foundations-of-Applied-↩
    Mathematics/Data/master/RNN/mozart_sonatas.tar.gz to drive/MyDrive/Colab/↩
    mozart_sonatas.tar.gz

Extracting mozart_sonatas.tar.gz
Untarring mozart_sonatas.tar
```

Parsing the Data

Music21 is musical toolkit for Python developed by MIT. [1] It can read and write music files with the .mid extension, which is a MIDI, Musical Instrument Digital Interface file. Midi files contain information on music, like which notes are played, how loud each note is, and how long each note is held.

There are two important object types: Notes and Chords. A note object is comprised of three attributes. The `pitch` and `octave` give information about the frequency of the note. There are seven pitches, A,B,C,D,E,F, and G. These are repeated, doubling the frequency of the vibration of the previous matching pitch. When the frequency of a note is doubled, it is called an octave. A piano has seven octaves, and the middle of the keyboard is called `middle c`. In Music21, it is represented by C4, where 4 is the octave. Lastly, the `offset` is the location of the note in the file. It describes how far apart the notes are, and thus how fast or slow they are played.

Chord objects are containers for multiple notes that are played at the same time.

```
from music21 import converter, instrument, note, chord,  stream
```

[1] `https://web.mit.edu/music21/doc/index.html`.

```
# Read the file piano_sonota_279.mid
midi = converter.parse('piano_sonata_279.mid')
notes_to_parse = instrument.partitionByInstrument(midi).parts[0].recurse()

# Display the Note and Chord objects, their pitches and offsets
for element in notes_to_parse:
    if isinstance(element, note.Note):
        print(element, element.pitch, element.offset)
    elif isinstance(element, chord.Chord):
        print(element, element.pitches, element.offset)

<music21.note.Note E> E5 803.0
<music21.note.Note F> F5 803.5
<music21.chord.Chord B3 B2> (<music21.pitch.Pitch B3>, <music21.pitch.Pitch B2↩
    >) 803.5
<music21.note.Note G> G5 804.0
<music21.note.Note F> F5 804.5
<music21.chord.Chord C4 C3> (<music21.pitch.Pitch C4>, <music21.pitch.Pitch C3↩
    >) 804.5
<music21.note.Note E-> E-5 805.0
<music21.note.Note D> D5 805.5
<music21.chord.Chord E-3 E-4> (<music21.pitch.Pitch E-3>, <music21.pitch.Pitch ↩
    E-4>) 805.5
```

Problem 1. Download the data. Write a function that accepts the path to the .mid files, parses the files, and returns a list of the 119348 pitches as strings. For the Chords, join the pitches of the notes in the chords with a . as in ('D3.D2').

Print the length of your list.

```
# Sample of part of the list
['G5', 'A5', 'C6', 'G3.C4', 'B-5', 'A5', 'G5', 'D3.D2']
```

In order for the data to be applied to an RNN, we need to create the sequences. We do this by looping through the list of notes and slicing it into lists of a given length. The labels for each sequence, or the correct answer that we want the RNN to learn, is the element immediately following the sequence. So if we have a list of 200 pitches and we want each sequence to be 100 elements long, then the label for the first sequence, elements 0-99 would be the 100th element. The label for elements 1-100, the second sequence, would be the 101st element and so on.

Since RNNs only accept numbers, we also convert the pitches to integers. Using the sample list above, we can map $'G5'$ to 0, $'A5'$ to 1, $'C6'$ to 2, and $'G3.C4'$ to 3. Then the sequences of strings become sequences of integers. The PyTorch DataLoader accepts a list of lists, where each element is of the form `[sequence, label]`. The sequence is a PyTorch Long tensor, and label is an integer. So in this case, the data is a sequence of integers representing the notes while the label is the integer representing the first note that follows the sequence.

```
# Example Sequence
example_sequence = [169, 269, 165, 187,  24, 366, 353, 269, 260, 233, 223, 169,↩
     162, 366,
        353, 269, 260, 233, 223, 169, 162,  24,   8, 269, 260,  91,  79, 269,
        260, 366, 353, 233, 223,  24,   8, 269, 260, 169, 162,  79,  72, 233,
        223, 114, 110,   8,   2,   8,   2,   8,   2, 187, 269,   8, 187, 269,
          2, 187, 269,   8, 187, 269,   2, 122, 233,   8, 122, 233,   2,   8,
          2, 278, 187,   8, 278, 187,   2, 278, 187,   8, 278, 187,   2, 246,
        122,   8, 246, 122,   2,   8,   2,  39, 278,   8,  39, 278,   2,  39,
        278,   8, 382,  79]

# Convert sequences to Long Tensor

first_sequence = torch.LongTensor(example_sequence[0:99])
second_sequence = torch.LongTensor(example_sequence[1:100])
first_label = example_sequence[100]
second_label = example_sequence[101]

# Example of a data point formatted for the DataLoader, [sequence, label]
>>> [first_sequence,first_label]
[tensor([169, 269, 165, 187,  24, 366, 353, 269, 260, 233, 223, 169, 162, 366,
         353, 269, 260, 233, 223, 169, 162,  24,   8, 269, 260,  91,  79, 269,
         260, 366, 353, 233, 223,  24,   8, 269, 260, 169, 162,  79,  72, 233,
         223, 114, 110,   8,   2,   8,   2,   8,   2, 187, 269,   8, 187, 269,
           2, 187, 269,   8, 187, 269,   2, 122, 233,   8, 122, 233,   2,   8,
           2, 278, 187,   8, 278, 187,   2, 278, 187,   8, 278, 187,   2, 246,
         122,   8, 246, 122,   2,   8,   2,  39, 278,   8,  39, 278,   2,  39,
         278]), 382]
```

Problem 2. Using the list returned in Problem 1, create the training and testing DataLoaders. Make sure to do all of the following steps:

- Convert the pitches to integers.
- Split the data into Long tensors of length 100.
- Create the labels.
- Randomly split the data into training and test sets using an 80/20 split.
- Create the DataLoaders for both sets of data, using `batch_size`= 128 for the training data and `batch_size`= 1 for the test data.

Print the length of each DataLoader.

Hint: To keep all batches the same size, drop the last training batch with the parameter `drop_last=True`.

LSTM

While RNNs have the ability to look at short-term history, like the previous word in a sentence, they lack longer term contexts. For example, predicting the last word in the "The boat is in the `water`" is relatively easy. Consider the following two sentences separated by some other text: "I grew up in France ... I speak fluent `French`." It's clear that the last word will be a language, but we need the previous information of France to correctly identify which language. RNNs can't remember this information due to exploding and vanishing gradients.

Long short-term memory, LSTM, networks are a popular RNN variation capable of long-term memory that solve this problem. They are used extensively in speech recognition, machine translation, and text-to-speech programs. Every step in the LSTM has three inputs: the current input, the short-term memory (hidden state) from the previous input, and the long-term memory (cell state). There are three gates that regulate these three types of memory. The Input Gate decides what information will be added to the long-term memory, and the Forget Gate chooses which information should be kept in the long-term memory. The Output Gate creates the new short-term memory.

Defining the Network Layers

In PyTorch, the memory is a tuple (hidden state, cell state) and must be initialized before the LSTM layer is called. Usually, the hidden state initialization function is defined in the network class and is called during the training loop for each batch. The LSTM layer can be stacked, with the input from one layer going directly to the next layer; `num_layers` is how many stacked LSTM layers there are in the model. The `hidden_size` is the number of features in the hidden layer. This can be any size, and for this lab, we will use 512.

```
class RNN(nn.Module):
    '''
    Recurrent Neural Network Class
    '''
    def __init__(self):
      super(Network, self).__init__()

    # define function to initialize hidden states
    def init_hidden(self, batch_size):
            weight = next(self.parameters()).data
            h0 = weight.new(self.num_layers, batch_size, self.hidden_size).↩
                zero_().to(device)
            h1 = weight.new(self.num_layers, batch_size, self.hidden_size).↩
                zero_().to(device)

            return (h0, h1)

# Initialize the model
model = RNN()

for epoch in range(30):
    for x, y in train_loader:
        # initialize the hidden states
        (h0, h1) = model.initHidden(128)
        # pass data through the model
        output, (h0, h1) = model(x, (h0,h1))

        h0 = h0.detach()
        h1 = h1.detach()
```

Before calling the LSTM layer, we will use an embedding layer to store the words. The embedding layer is a lookup table that takes in indices and outputs the word embeddings. This is PyTorch's method of one-hot encoding, a process in which variables are converted to binary for better predictions. The first parameter is the number of words in the dictionary; in our case, there are 741 possible notes and chords. The second parameter is the embedding dimension. 32 and 64 are good choices for the embedding dimension.

The LSTM layer has 5 parameters. The first three have already been discussed. The parameter `batch_first` is a boolean that indicates if the batch size is the first or the second dimension in the input tensor. Since we are using the DataLoader, the batch size will be the first dimension and `batch_first=True`. If the last parameter, `dropout`, is defined, a Dropout layer is added after each LSTM layer, except the last. During a Dropout layer, elements of the input tensor are randomly zeroed out with probability p, and the output is scaled. This is used for regularization to improve the network.

Another layer we will use is the BatchNorm1d. It normalizes the input and has as parameter the number of features of the input. The last layer should be softmax activation function. Softmax rescales a tensor to $[0, 1]$ with sum of all elements equal to 1. Thus the output of Softmax can be thought of as a probability vector. Notice that all of the layers: Embedding, LSTM, Linear, BatchNorm1d, Dropout, and LogSoftmax are initialized in the *__init__()* function.

During training, when the model is called, the input is embedded and then passed to the LSTM layer with the hidden states. The hidden state output is saved for the next batch while the lstm output from the final step is sent through the rest of the model. To prevent differentiating the hidden states, we must call the `detach()` method before taking a backwards step. This disables automatic differentiation on the hidden states. Note that because we don't do a backwards step during the testing

```
class Network(nn.Module):
  """ Example class for LSTM model """
  def __init__(self,vocab_size,embedding_dim):
    super(Network, self).__init__()

    self.hidden_size = 512
    self.num_layers = 3
    self.dropout = 0.3
    self.vocab_size = vocab_size
    self.embedding = nn.Embedding(vocab_size,embedding_dim)
    self.lstm = nn.LSTM(embedding_dim, self.hidden_size, self.num_layers, ↩
        batch_first=True, dropout=.3)
    self.batch1 = nn.BatchNorm1d(self.hidden_size)
    self.dropout = nn.Dropout(.3)
    self.linear = nn.Linear(self.hidden_size,self.vocab_size)
    self.softmax = nn.LogSoftmax(dim=1)
```

```
def forward(self, x, hidden):

  embeds = self.embedding(x)
  lstm_out, hidden = self.lstm(embeds, hidden)
  out = self.dropout(self.batch1(lstm_out[:,-1])) #output from final step is ↩
      passed forward
  return self.softmax(self.linear(out)), hidden
```

Problem 3. Create the network class. Include at least 3 LSTM layers, each followed by Dropout layers with probability .3. Also have at least 2 Linear layers. The last LSTM layer and each of the Linear layers should be followed by a BatchNorm1d layer, for at least 3 total batchnorm layers. The final layer should be a Softmax activation.

Initialize the model, loss as CrossEntropyLoss, and optimizer as RMSprop,

```
optimizer = torch.optim.RMSprop(model.parameters(),lr=.001)
```

Train the model for at least 30 epochs, saving the weights every epoch with

```
torch.save({
    'epoch': epoch_number,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,},filepath)
```

After taking a backwards step during training, scale the gradients using

```
nn.utils.clip_grad_norm_(model.parameters(), 5)
```

This will ensure that the gradients are reasonably sized so that the model can learn.

At the end of each epoch, calculate the accuracy and mean loss on the test data. Remember to change the model to *eval*() mode when running the test data and *train*() when running on the training data. You will also need to reinitialize the hidden states $(h0, h1)$ since the batch sizes are different.

After the accuracy is above 95%, plot the training and test losses versus epochs on the same plot. Plot the accuracy versus epochs.

Achtung!

Colab has a 12 hour limit on the amount of GPU available and the longer one runs, the less priority one has. Plan on spending at least 2 hours training. Carefully watch the loss and accuracy to ensure that the model is training correctly.

Generating Music

Now that we have trained the model, we can create our own piano sonata excerpt. If the notebook has disconnected, reinitialize the model, loss, and optimizer. Then load the saved model so that we don't have to retrain the model.

```
def load_model(filename):
  """ Load a saved model to continue training or evaluate """
  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

  model = Network(n_vocab,32)
  model = model.to(device)
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.RMSprop(model.parameters(),lr=.001)

  checkpoint = torch.load(filename,map_location=torch.device('cpu'))
  model.load_state_dict(checkpoint['model_state_dict'])
  optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
  last_epoch = checkpoint['epoch']
  loss = checkpoint['loss']
  model.eval() # For evaluation only

  return model, criterion, optimizer
```

To start predicting, we will use a random sequence in our test data as our initial point. Then we will initialize the hidden states and pass them into the model with the sequence, just as we did in the training step. A new input sequence is created by appending the `argmax` of the output to the sequence and dropping the first entry. To create the sequence of notes, we convert the max of the output back to the corresponding pitch. This process is repeated to generate a sequence of pitches.

Problem 4. Write a function that randomly chooses a sequence in the test data and predicts the next 500 elements. Return a list of the 600 pitches in the sequence. It should look similar to

```
['D4', 'C#4', 'F#5', 'G5', 'A5', 'C6', 'G3.C4', 'B-5', 'A5', 'G5', 'A5', '↩
    G5', 'F5']
```

Now we just need to convert the pitches into Music21 Notes and Chords objects. For each element in the list of pitches, we first determine if it's a Note or Chord. For each Note, we create a Note object with the pitch and instrument. Chords can be create from the individual note objects.

```
notes = [ ]

# Create note objects for each pitch in the chord
for pitch in chord_pitches:
    new_note = note.Note(pitch)
    new_note.storedInstrument = instrument.Piano()
    notes.append(new_note)

# Create a Chord object
new_chord = chord.Chord(notes)
new_chord.offset = offset
```

Ater creating the objects, the last attribute to add is the offset. To prevent the notes from playing at the same time, we increase the offset after each note or chord. The simplest way to do this is look at the distribution of offsets and increase by a set amount each time. Since the most common offset difference, 0.0, results in notes being played at the same time, we'll ignore it and choose to increase the offset by either .25 or .5. For a more advanced option, you could randomly generate which offset difference to use based on the probability of the distribution.

0.0	1999
0.25	1167
0.5	507

Table 1.1: The three most common offset distance and their frequency in the Mozart data.

Lastly, we write the the notes to a midi file and save it.

```
midi_stream = stream.Stream(output_notes)
midi_stream.write('midi', fp=file_location)
```

Problem 5. Convert the sequence from Problem 4 into note and chord objects and save it to 'mozart.mid' file.