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

Convolutional Neural Networks work well for problems like image classification where 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 their recurrent nature. 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 pop-up 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 Mozart piano sonatas as the data to train on. 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 a musical toolkit for Python developed by MIT.[1] It can read and write music files with the .mid extension, which are MIDI files, standing for Musical Instrument Digital Interface files. Midi files contain information on music, like which notes are played, how loud each note is, and for 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 pitches repeat, doubling the frequency of the vibration of the previous matching pitch. The interval over which the frequency of a note is doubled 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 temporal location of the Note in the file. Chord objects contain multiple Note objects 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.stream().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
```

```
# Helper function to parse through a Chord object
def order_pitches(pitches):
    """ pitches: element.pitches object where element is a chord.Chord
        returns: sorted list of strings for each pitch in the chord
    """
    return sorted(list(set([str(n) for n in pitches])))
```

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 114208 Notes and Chords as strings. There are many element types in MIDI files, so be sure to only look for Notes and Chords. For the Chords, join the pitches of the Notes in the Chords with a . as in ('D3.D2').

Print the length of your list and the number of unique Notes and Chords.

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

Hint: An easy way to get the list of mozart sonata file names is with the following code.

```
import glob
```

```
>>> glob.glob(filepath + "/mozart_sonatas/mozart_sonatas/*.mid")
```

Also, you'll want to wrap `element.pitch` with `str()` to convert it into a string. Furthermore, the `.join()` method may be useful when constructing the Chord strings.

For the remainder of this lab, we will refer to the notes and chords in the list created in Problem 1 simply as pitches. In order for this data to be applied to an RNN, we need to create sequences. We do this by looping through the list of pitches and slicing it into lists of a given length. The label for each sequence, or the correct pitch we want the RNN to predict, is the element immediately following the sequence. So for a sequence length of 100, given elements 0 through 99 as a sequence, element 100 would be the label.

Since RNNs only accept numbers, we should convert the pitches to integers. Using the sample list in Problem 1 as an example, we would map `'A5'` to 0, `'C6'` to 1, `'G3.C4'` to 2, `'A5'` to 0 again, and so on. The PyTorch DataLoader accepts a list of lists, where each element is of the form `[sequence, label]`, where the sequence is a PyTorch Long tensor, and the label is an integer. So in our case, the sequence will be a tensor of integers representing pitches while the label will be the integer representing the first pitch that follows the sequence.

```
# Example Data
example_data = [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]

# Create sequences as Long Tensors
first_sequence = torch.LongTensor(example_data[0:100])
second_sequence = torch.LongTensor(example_data[1:101])
first_label = example_data[100]
second_label = example_data[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 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, validation, and test sets using a 70/15/15 split.
- Create the DataLoaders for these sets of data, using `batch_size=128` for the training data and `batch_size=32` for the validation and test data; also, set `shuffle=True` for the training data and `False` for the validation and test data (this is common practice in Deep Learning).

Print the length of each DataLoader (they should be 624, 535, and 535, respectively).

Hint: To keep all batches the same size, drop the last training batch in the DataLoader 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, the *Forget Gate* chooses which information should be kept in the long-term memory, and 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, but for this lab we will use 256.

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

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 around 668 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 sometimes used for regularization to improve the network. However, we will *NOT* add a Dropout layer to our model, because we will instead use a BatchNorm1d layer. BatchNorm1d layers normalize the input and have as parameters the number of features of the input. Thus, if we were to use both Dropout to BatchNorm1d layers, the input would be normalized over fewer nodes than the input actually contains, which would throw off the scaling of the model.

The last layer should be a softmax activation function. Softmax rescales a tensor to $[0, 1]$ with the 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, and LogSoftmax are initialized in the `__init__()` function.

```
  class RNN(nn.Module):
    """ Example class for LSTM model """

    def __init__(self, n_notes, embedding_dim):
      super(RNN, self).__init__()

      self.hidden_size = 256
      self.num_layers = 3      # num_layers is the number of layers in the LSTM
      self.n_notes = n_notes   # n_notes is the number of unique pitches
      self.embedding = nn.Embedding(n_notes, embedding_dim)
      self.lstm = nn.LSTM(embedding_dim, self.hidden_size, self.num_layers, ↩
          batch_first=True)
      self.batch1 = nn.BatchNorm1d(self.hidden_size)
      self.linear = nn.Linear(self.hidden_size, self.n_notes)
      self.softmax = nn.LogSoftmax(dim=1)
    def forward(self, x, hidden):
      embeds = self.embedding(x)
      lstm_out, hidden = self.lstm(embeds, hidden)
      out = self.batch1(lstm_out[:,-1])
      # Output from final step is passed forward
      return self.softmax(self.linear(out)), hidden
```

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 during training. Because we don't do a backwards step during testing, we don't need to worry about detaching the hidden states during testing.

```
# Initialize the model
model = RNN()

# Hidden state training demonstration
for epoch in range(30):

    # Initialize the hidden states
    (h0, h1) = model.init_hidden(128)

    for x_truth, y_truth in train_loader:

        # Pass data through the model to get output and new hidden states
        output, (h0, h1) = model(x_truth, (h0, h1))

        # Disable automatic differentiation on the hidden states
        h0 = h0.detach()
        h1 = h1.detach()
```

A Faster Way to Calculate Validation Accuracy

We would like to periodically check our model's validation accuracy as our model is training, but this takes time. One way to save time is to only calculate validation accuracy every n epochs. Another way is to increase the batch size of the validation DataLoader; this is the method we use in this lab, which is why we set the validation and test DataLoader batch sizes to 32 in Problem 2. The problem is, if we compare the predicted labels of a large batch to their true values at the same time, the probability that every data point is correct is small, so the validation accuracy will be mostly 0. The way to work around this is to compare the predicted labels of each batch with their true values seperately, not together. An example of how this might be done is demonstrated in the following:

```
validation = 0
model.eval()
for x_truth, y_truth in validation_loader:
    x_truth, y_truth = x_truth.to(device), y_truth.to(device)
    (h0, h1) = model.init_hidden(val_batch_size)
    y_hat, _ = model(x_truth, (h0, h1))

    # sum how many elements equal each other between the true and predicted
    # batches, then divide by the number of elements in the batch
    validation += sum(torch.eq(y_truth, y_hat.argmax(1))) / val_batch_size

mean_validation_accuracy = validation.item() / len(validation_loader)
```

Problem 3. Create an LSTM network class. Have a hidden layer size of 256, and include at least 3 LSTM layers. 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. Define the loss as CrossEntropyLoss, and define the 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 every epoch, calculate the validation accuracy and mean loss on the validation data. Remember to change the model to `eval()` mode when running the validation data and then `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 validation accuracy is above 60%, plot the training and validation losses versus epochs on the same plot. Also, plot the validation accuracy versus epochs. Then, print the final test accuracy by running the finished model on the test data.

Hint: Training this model for 30 epochs on a GPU will take at least 15 minutes. You may want to test your code by only training it for 2 epochs, and then when everything works the way it should, train it for the whole 30 epochs.

Achtung!

Colab has a 12 hour limit on the amount of GPU available and the longer one runs, the less priority one has. If you follow the instructions given in this lab correctly, training should take less than half an hour. 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. You can then load the saved model with the following code, so that you don't have to retrain the model.

```python
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")

    # n_notes is the number of unique pitches
    model = RNN(n_notes, embedding_dim)
    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() # Toggle evaluation mode

    return model, criterion, optimizer
```

We can now use the following method to predict a new sequence of notes. We will start with an initial sequence of notes, predict what note should follow, and then shift the sequence to include this new note in order to predict the next one, and we'll repeate this process for as long as we like.

Specifically, first select a random sequence from the test data, and initialize an empty list of predictions. Then, for each note we wish to predict, initialize the hidden states using `model.init_hidden(batch_size)`, and then get a prediction by inputting the sequence and these hidden states into the model, just as we did in the training step. The argmax of this prediction (`prediction.argmax().item()`) is an integer representing a pitch. Append this integer to our list of predictions, and then update the sequence by appending this integer to it, and then by dropping the first entry of the sequence. Repeat this process for each note we wish to predict. The list of predictions (which is a list of integers) can then be converted into pitches using the same method we used to initially convert the pitches into integers.

Problem 4. Write a function that randomly chooses a sequence in the test data (which has length 100) and predicts the next n elements, defaulting to 500. Combine the initial sequence with the predicted elements, convert them to pitches, and return this list of 600 pitches. It should look similar to

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

Now we need to convert our list of pitches into Music21 Notes and Chords objects. For each element in the list of pitches, we first determine if it's a note or a chord by the presence of a . (period) in the string. Music21 Note objects are created using the pitch and instrument type. If the element is a chord, we can create a Muisc21 Chord object by first creating a list of Note objects for each note in the chord.

Music21 Note and Chord objects must also have a specified *offset*. The offset indicates at what timestep each object is to be played. The first object will have an offset of 0, and the offset will increment for each following object. The simplest way to choose each offset is look at the distribution of offsets in the original dataset and choose a set amount to increment the offset each time. Since the most common offset (0.0) results in notes being played at the same time, we'll ignore it and choose to increment the offset by either 0.25 or 0.5. For a more advanced option, you could randomly generate which offset to use based on a probability distribution that reflects the following:

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.

In summary, while looping through our predicted pitches, if we should come accross a Chord, the code to create a Music21 Chord object would look like the following:

```python
notes = [ ]

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

# Create a Chord object using list of Note objects
new_chord = chord.Chord(notes)

# Specify offset for this object
new_chord.offset = offset
```

Finally, we write the list of Music21 objects to a midi file and save it.

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

Problem 5. Convert the predictions from Problem 4 into Music21 Note and Chord objects and save it as `'mozart.mid'`.