CSC 470: Introduction to Neural Networks

Programming Project #4

Text Generation Using a Generative Recurrent Neural Network

Background

As discussed in the lecture, recurrent neural networks (RNNs) are unique among all the types of networkws we've covered in that RNNs are capable of learning from sequential data, as opposed to an ensemble of data like is used for multilayer perceptrons and convolutional neural networks. There are many scenarios where data are present in sequences. For example, tracking stock market values over time is a sequence; predicting which letter comes next in a string would also involve sequential data; learning the sequence of actions required to move a finite state machine from some initial state to different state again involves sequential processing.

The support for feedback mechanisms in RNNs inherently allows recurrent networks to learn from sequential data by incorporating a time factor into the learning process. Feedforward multilayer perceptrons can use regression analysis to find patterns in sequential data, but they do not fare as well when there is a sequence dependence on the input variables. The environment has a state that can change based on certain actions or stimuli. Each change in state is associated with a time step; during a state change, an environment's previous state resides in the past, while the state it changes into resides in the future. The time doesn't have to be measured in terms of actual time; i.e., seconds, etc. Instead, a series of time steps can be perceived as simply being a chronological sequence of state changes, with the initial state being time point 0, the next state being time point 1, and so on.

So far in this course we have focused on neural networks as **predictive**; i.e., they try to learn patterns and trends that are present in an ensemble of data, and then try to predict the outcome for inputs they had never before seen. Neural networks can also be **generative**, meaning they not only learn patterns and trends from data, they also attempt to generate new data based on what they've learned.

Any neural network that is trained through back-propagation, including recurrent networks that are trained using back-propagation through time (BPTT), can suffer from two types of gradient problems: 1) **exploding gradients**, and 2) **vanishing gradients**. Updates to the synaptic weights are typically computed using partial derivatives of the error function. When this results in very large update values, the gradient explodes, destabilizing the learning process. If the update values become very small, the gradient vanishes, and the learning stalls because it is no longer able to significantly affect the synaptic weight values.

Long Short-Term Memory Recurrent Networks

We've already established that the state of an environment plays a central role in allowing recurrent networks to efficiently process sequential data. Since each state in a sequence is a function of time, the sequence of states constitutes the **memory** of the system as it evolves over time. A special type of recurrent network, known as a long short-term memory (or LSTM, for short) network, trained using BPTT, takes advantage of the inherent memory capacity of recurrent networks to effectively deal with complex sequence problems and address the vanishing gradient problem.

Unlike most neural networks, LSTM networks are not comprised of neurons. Instead, they consist of **memory units** connected into layers. Each unit has additional components and a capacity for remembering recent sequences that give the unit greater power than a standard neuron. In particular, a unit can make use of three different types of **gates** to control the output of the unit:

1. **Input Gates**, which conditionally determines which input values to use to update the memory state of a unit

2. **Output Gates**, which conditionally determine what value a unit should output based on the inputs and the unit's memory of recent sequences

3. **Forget Gates**, which conditionally determine what information to discard from the unit

Each type of gate has its own weight which is updated during learning, making the memory blocks more sophisticated and "intelligent" than the standard neuron.

The Problem of Text Generation

As mentioned previously, recurrent neural networks can be generative in nature, meaning that not only are they able to make predictions like other neural networks, they can also use these predictions to create new data. The **text generation problem** is an example of a problem that requires a generative solution. Simply stated, the problem is to take a corpus of text, such as a book, magazine article, essay, etc., containing sequences of words that have meaning; i.e., that are capable of communicating thoughts and information, and based on the sequences of words present in the text generate new text that will also hopefully have meaning. The text generation problem can also be extended to analyze text at the character level, and in this way potentially generate words that are not already present in the text. As you can tell, this is definitely a sequential problem, so a recurrent neural network should be well suited to take it on. In this project, you will build a LSTM recurrent neural network to learn character sequences from a corpus of text and attempt to use that learned knowledge to generate new sequences of characters.

****** A word of caution: Training a recurrent neural network requires significantly more computing power than the other types of neural networks we used for the previous projects. I have adapted the design of this project so that you should be able to successfully get it to work even if you do not have a computer with a high-end graphics card, but you may find the training will still require several hours to complete.***

Installing Needed Third-Party Libraries

The only third-party libraries this project needs are NumPy and Keras, which you should already have installed from the previous projects.

Scripts Provided:

I am providing you with 2 Python scripts for this project:

`lstm_large.py` — This script creates the LSTM recurrent network to be used.

`lstm_large_generate_text.py` — This script generates text using the LSTM network.

The Dataset

For the dataset for this project I have included a text file containing the complete text of the children's book, *The Cat in the Hat*, written by Dr. Seuss. I chose this book to serve as the corpus of text because of its size and limited vocabulary. (Dr. Seuss was usually given a short list of words by his publisher that he was allowed to draw from when writing his books.) The length of the text used as the corpus is a major determining factor in how long it will take to train the network. A larger corpus will increase the learning time. There is a tradeoff, however, in using a shorter corpus. The ability of the network to generate new text increases as the vocabulary of the corpus increases, so a shorter corpus will tend to cause generation of text that doesn't vary as much from the patterns in the original text.

Exploratory Analysis of the Dataset

You won't need to perform any exploratory analysis for the text I have provided, but you may need to do a little preprocessing if you substitute another text source. For example, you might need to remove things like page numbers, headers and footers, figures and tables, etc. You may also need to make additional modifications if the text contains a lot of misspelled words, grammatical errors, etc.

Training and Testing Sets

For this project the dataset doesn't need to be split into training and testing sets. The entire text corpus will be used for training, and since the goal is to generate new data there will not be any need for predicting outcomes for unseen inputs. The network still needs to be trained, however, and this will be done using a sliding window of a specified length, similar to the filters used with CNNs.

Building the LSTM Recurrent Network

Not a lot of code is required to implement the LSTM network thanks to the Keras framework. Since I've included the base script, please refer to that script for the full code. I will only reprint certain sections here for explanatory purposes.

Reading the Dataset

The first step in building the network is to read in the text from the data file. The text is read and converted to all lower case, since in most cases upper case doesn't appreciably change the semantics of a word, and we need to try to keep the vocabulary to a minimum in order to make training as practical as possible. The vocabulary in this case refers to the number of unique characters, including punctuation marks. After converting the text to lowercase, the unique characters need to be identfied and mapped to unique integers, since the network operates on numerical data. Once this is done, the total number of characters is diaplyed, along with the number of unique characters. For the Cat in the Hat text, the total number of characters is 7353, and the number of unique characters that comprise the vocabulary is 32. Note that the vocabulary can be larger than the 26 letters in the English alphabet, since spaces and punctuation marks are included.

Creating the Training Patterns

To train the network, instead of using the usual training set and testing set, we will use a technique more akin to the filters used in a CNN project for extracting features form images. We will use a sliding window of a fixed length to scan the entire text, shifting the window one character at a time, until the sliding window reaches the end of the text. The text inside the window acts as the input, and the single character that immediately follows the window acts as the desired output. Each input/output pair is referred to as a **training pattern**. So for example, the first line of the Cat in the Hat text is:

```
the sun did not shine.
```

If we use a sliding window of fixed length equal to 10, we would get the following input/output pairs:

Input	Desired Output
the sun di	d
he sun did	<space>
e sun did	n
sun did n	o
un did no	t
n did not	<space>
did not s	s
id not sh	h
d not shi	i
not shin	n
not shine	e

This allows each character in the text a chance to be learned from the sequence of characters that precedes it, except of course for those characters that appear in the first position of the window.

There is tremendous flexibility in how you set the length for the sliding window, as well as how far the window shifts for each new training pattern. You can base the length on a single character, a group of characters, entire words, entire sentences, etc.

Transformation of the Training Patterns

In order to use the training patterns with a LSTM network in Keras, we must transform the input sequences into a form the LSTM network expects, which is a matrix with the dimensions:

```
[samples, time steps, features]
```

where `samples` is the number of samples, `time steps` is the number of time steps, and `features` is the number of features. In our case, the dimensions will be:

```
[n_patterns, seq_length, 1]
```

where `n_patterns` is the number of training patterns, `seq_length` is the length of the sequence in the sliding window, and the number of features represents the length the sliding window will shift each time, which is 1 character. The `reshape` function of the numpy package will do this for us with nothing but a simple function call:

```
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
```

Since we will be using a sigmoid activation function in the LSTM network, we next need to convert all the integers in the training patterns to floating point values in the range 0-to-1. This is accomplished using the following statement:

```
X = X / float(n_vocab)
```

Finally, we need to **one hot encode** the output values, since we are dealing with categorical data (characters which are mapped to integers, in this case). One hot encoding is simply a way to uniquely label each of the possible output values in cases where there is a mapping of values to integers, but there is no ordinality involved in the mapping. An easy way to think of one hot encoding is that each value is identified by a unique binary string or vector. For example, suppose we have 10 different characters, with each character mapped to an integer value in the range 0-9, and suppose the character 'g' is mapped to the integer 3, and the character 'm' is mapped to the integer 7. One hot encoding of the integer values for these characters would look like this:

```
g: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

m: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
```

The length of the binary string or vector will be equal to the number of different data values that need to be encoded, and for the integer value, *n*, to which a data value is mapped, the *n*th position in the string or vector will be a 1, and the rest of the positions will be 0.

Assembling the Layers of the LSTM Network

With the training patterns created and encoded, we can assemble the layers of the LSTM network. The network is sequential, so we begin by creating a `Sequential` object:

```
model = Sequential()
```

Next, we add a hidden LSTM layer containing 256 memory units. We use the parameter, `input_shape`, to define the dimensions of the inputs, and we set the `return_sequences` argument to `True` to instruct the layer to return the last output. The statement for doing this is:

```
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_
sequences=True))
```

After this we add a dropout layer with a dropout probability of 20%:

```
model.add(Dropout(0.2))
```

The dropout layer will cause 20% of the inputs and connections to be excluded from weight updates during training, as a regularization method to help prevent overfitting.

We then add a second hidden LSTM layer, followed by a second dropout layer:

```
model.add(LSTM(256))
model.add(Dropout(0.2))
```

We don't need to specify the input shape this time since the second LSTM layer will already be receiving its input from the preceding layers, which are already shaped correctly.

Lastly, we add an output layer, which is a Dense layer containing a node for each of the possible outputs. The layer uses a softmax activation function, which outputs the probability predicition for each of the possible outputs, in the range 0 to 1.

```
model.add(Dense(y.shape[1], activation='softmax'))
```

With the model assembled, the next step is to compile it:

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

We are once again using the Adam optimizer, but for this problem we use a categorical crossentropy function as the loss function, since this problem is bascially an exercise in classifying inputs to categories of outputs.

Checkpointing

Training recurrent networks like this one often takes a significant amount of time. Depending on the hardware configuration, the size of the text corpus, and the network parameters, training can take anywhere from several hours to several days. Because of this, we use a technique known as **checkpointing** to periodically save the current network state to a file, each time the loss decreases, indicating the network's performance has improved. When we later attempt to use the network to generate new text, we will use the state that gave us the lowest loss. Creating the checkpoint involves the following statements:

```
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1,
                             save_best_only=True, mode='min')
callbacks_list = [checkpoint]
```

The epoch number and the loss value are both incorporated into the name of the checkpoint file so we can easily identify the checkpoint with the lowest loss later on.

Training the Network

We are now ready to train the network. We use the `fit` function for this, passing it the inputs and outputs. For this project we will use 50 epochs and a batch size of 64. We also specify the list of checkpoints created in the preceding statements. The complete function call is:

```
model.fit(X, y, epochs=50, batch_size=64, callbacks=callbacks_list)
```

As the network trains, you should see a progress bar that shows the estimated time left to complete the epoch, and the current loss value. When the epoch has completed, the display will show how long it took to complete the epoch. For example:

```
Epoch 1/50
114/114 [==============================] - 88s 743ms/step - loss: 3.0190
```

This shows there were 114 iterations, or steps, and it took 88s to complete the epoch (743ms/step). The loss value at the end of the epoch was 3.0190. Not all epochs take the same amount of time to complete. In my dry run, the average time to complete an epoch was around 95 seconds, so 50 epochs required approximately 79 minutes. Keep in mind this is for a simple text like Cat in the Hat. Longer texts will require more time, and you can quickly run into practical time considerations if you try to process too much text. As mentioned above, training will produce a series of checkpoint files. ***Be sure to save the checkpoint file with the lowest loss. You will need that file for the text generation part of the project.***

Text Generation

Assembling the LSTM Network and Loading the Checkpoint

Once the LSTM network has been trained, we are ready to use it to try generating new text based on the patterns the network learned. To do this, we will need to recreate the LSTM network we used for training, as well as read in the text file and do all the text preprocessing steps. However, we will not need to train the network, we will just need to load the checkpoint state with the lowest loss value that was created during training. (We could use any of the checkpoint files, actually, but the one with the lowest loss should give us the best results.) I have provided another script for this part of the project:

```
lstm_large_generate_text.py
```

If you look at the script you will notice it is mostly a copy of the script we used for training the network. I won't rehash the copied portions, instead I will just discuss the portions that are new.

Everything in the new script is the same as the previous script until just after the LSTM network is recreated. Once the network has been assembled, instead of training it we load the checkpoint file with the lowest loss, which will most likely be the file created after the last training epoch. The statements for loading the checkpoint look something like this:

```
filename = "weights-improvement-50-0.1250-bigger.hdf5"
model.load_weights(filename)
```

The name given to the checkpoint file will depend on how many epochs you use, and the loss value at the time the checkpoint file was created. In the example above, the checkpoint file is the one create after the 50th epoch, with a loss value of 0.1250. You will need to look at the checkpoint files created when you trained your network for the filename to use, as it may differ from mine. Following this, the network is compiled, just as it was before, during training.

Generating New Text

The remaining lines of code in the script are responsible for generating the new text. We start by setting a random seed, which will be one of the training patterns selected at random:

```
start = numpy.random.randint(0, len(dataX) - 1)

pattern = dataX[start]
```

Next, we generate 1000 characters of new text using the knowledge learned by the LSTM network, and display the generated text to the screen:

```
for i in range(1000):
      x = numpy.reshape(pattern, (1, len(pattern), 1))
      x = x / float(n_vocab)
      prediction = model.predict(x, verbose=0)
      index = numpy.argmax(prediction)
      result = int_to_char[index]
      seq_in = [int_to_char[value] for value in pattern]
      sys.stdout.write(result)
      pattern.append(index)
      pattern = pattern[1:len(pattern)]
```

Results

The quality of the newly generated text will depend on many factors. The diversity of the patterns in the text corpus will be the major determining factor. The Cat in the Hat text, while useful in that it allows for comparatively quick training, is unfortunately quite short, and due to the writing style of Dr. Seuss, has many repetitive patterns. This will cause the text that gets generated to look a lot like the original text.

The length of the sliding window will also play a major role. The longer the window, the fewer training patterns there will be, and this could have a negative impact on the quality of the ext that is generated. Decreasing the size of the window will compensate for this, to an extent, but at the cost of increasing the training time.

The number of epochs used for training also affects the quality of the generated text, as running more epochs will tend to reduce the loss further. However, there is danger of overfitting, which for this problem would mean that the network wouldn't be able to generate text that is valid, but that isn't already present in the original text corpus. Ideally, we want the network to be able to produce new text that is different from the original content.

What You Need to Do:

1. Download the 2 scripts I provided, along with the Cat in the Hat text. Use the `lstm_large.py` script to create and train the network on the text. **Be sure to save the final checkpoint file!** You can delete the other checkpoint files that get generated if you wish.

2. Run the `lstm_large_generate_text.py` script to generate new text. You will need to paste the filename of the checkpoint file created during training, at line 91 in this script. Save a record of the output for your submission.

3. Time permitting, make at least one change to the network to see what effect it has. I will give you 5 bonus points for each change you submit after the first one, up to a maximum of 15 bonus points. Use a separate script for each change, as you did for the previous projects. Suggestions include:

    a. Eliminate the second LSTM layer and its associated dropout layer.

    b. Decrease the size of the sliding window (`seq_length`). Try a value like 25, or whatever value will allow you to still train the network given the time available.

    c. Try using a different text corpus for the input dataset. (***Caution: Don't use a corpus that is too large, or you may not have time to train the network!***)

4. Submit your scripts on Canvas. If you used a different text corpus, just tell me where you found it, you don't need to upload it.

Point Distribution

Item	Points Possible	Points
Trained original network using the provided scripts + results.	10	
One required change to the network + results.	10	
Bonuses:		
5 points per additional change, up to 15 points max	0-15*	
Total:	**20 (up to 35 with bonuses)**	