CS78/278, Winter 2023, Problem Set # 3
Text Generation

February 16, 2023

Due: March 2, 2023, 11:59pm ET

This problem set requires you to implement functions in PyTorch for creating and training a model that predicts the next character given a sequence of past characters. Such a model can be used to compose essays by recursively predicting the next character. Section 6 lists the deliverables of this assignment, which include code and data. The code ('*.py') and data ('*.pt') files must be submitted electronically via Canvas as a single zipped directory. The directory must not contain any subdirectories. The name of the directory should be in the format 'First_Middle_Last_HW3', where 'First', 'Middle' (if it exist), and 'Last' match your student name on Canvas. Your zipped directory should therefore have the format 'First_Middle_Last_HW3.zip'.

You will receive points for correctly implementing and training base models. There are no **"improvement" points** for this assignment. As you may remember, you are given a total of 4 free late days to be used for homework assignments. The late days that you use will be deducted from your total of 4 days. Once these 4 days are used up, any homework turned in late will receive 0 point. Make sure to upload all of your code and the base models to Canvas. We cannot accept forgotten code or data files past the submission deadline without penalizing for late days.

1 Overview

Text generation is an important and active area of research within natural language modeling. Text generally contains long-term temporal structure, which is practically difficult to learn because of memory limitations. In this task, you will write code that trains models to learn temporal dependencies in text. The goal is to learn to predict the next character given a sequence of past characters. In particular, you will build and train two types of recurrent neural networks for text generation: a basic recurrent network utilizing a hyperbolic tangent activation function, and a long short term memory (LSTM) network. We hope that the models can learn basic patterns found in the corpus. Like most natural language processing problems, this task is quite difficult and usually requires a large amount of data and computational time. Since you are constrained by a small dataset and limited computational power, it is normal for your models not to generate coherent essays. Please note that although we focus on text generation in this assignment, the models you will implement can be used for any sequence-to-sequence task with only a few or no modifications.

1.1 Notation

Extending the notation in the lecture slides, we express a dataset $\mathcal{D}$ in the following way:

$$\mathcal{D} = \left\{ \left(\mathbf{X}^{(1)}, \mathbf{y}^{(1)} \right), \left(\mathbf{X}^{(2)}, \mathbf{y}^{(2)} \right), \cdots \left(\mathbf{X}^{(m)}, \mathbf{y}^{(m)} \right) \right\} \tag{1}$$

$$\mathbf{X}^{(i)} = \left[\mathbf{x}_1^{(i)}, \mathbf{x}_2^{(i)}, \cdots, \mathbf{x}_t^{(i)} \right], \text{where the indices } 1, 2, \cdots, t \text{ denote time steps} \tag{2}$$

$$\mathbf{x}_t^{(i)} = \left[x_{t,1}^{(i)}, x_{t,2}^{(i)}, \cdots, x_{t,d}^{(i)} \right] \in \{0, 1\}^d \tag{3}$$

$$\mathbf{y}^{(i)} = \left[y_1^{(i)}, y_2^{(i)}, \cdots, y_t^{(i)} \right], \text{where the indices } 1, 2, \cdots, t \text{ denote time steps} \tag{4}$$

$$y_t^{(i)} \in \{0, 1, 2, \cdots, d-1\} \tag{5}$$

Furthermore, the observations $\mathbf{X}^{(i)}$ are time-delayed vectors of $\mathbf{y}^{(i)}$ such that $y_{t-1}^{(i)}$ denotes the only index of $\mathbf{x}_t^{(i)}$ which has 1 (i.e., $\mathbf{x}_t^{(i)} = \texttt{one-hot-encode}(y_{t-1}^{(i)})$.) In other words, at each time step t the model will infer the event occurring at time step $t+1$. We define the pre-activation and activation for layer k at time step t respectively as $\mathbf{a}^{(k)}(\mathbf{x}_t)$ and $\mathbf{h}^{(k)}(\mathbf{x}_t)$. Descriptions of the datasets, the model architectures, and the deliverables follow.

1.2 Data

The dataset for this assignment is composed of William Shakespeare's plays *King Lear*, *Macbeth* and *Hamlet*. These plays have been compiled into a text file `shakespeare.txt` which has been provided to you. We have also provided you with two optional corpuses (Shakespeare's sonnets and the tweets of a public figure) located in the folder `Optional Datasets` to play. You can (should) play around with these optional datasets but note that the models you submit must be trained exclusively on `shakespeare.txt`.

We have also provided a function `create_sequence` in `create_sequence.py` which will create a sequence-to-sequence dataset as described in Section 1.1 given the path to the dataset, a sequence length and a validation ratio. We skip encoding the observations $\mathbf{X}^{(i)}$ into one-hot vectors in `create_sequence.py` until training time to reduce the memory requirements of the assignment. In

addition to the data, `create_sequence.py` also returns the mapping from characters to integers as well as the reverse mapping. You should inspect `create_sequence.py` and ensure that you understand the code and all the objects it returns.

2 Layer Implementations

This section refers to the code stubs which must be completed by you in the following files:

- `basic_rnn_cell.py`
- `lstm_cell.py`

The `Sequential` API is not suitable for RNNs since they are naturally recurrent. We will extend `nn.Module` and define the forward propagation. The following subsections give details of the function specifications of our recurrent models.

2.1 Basic RNN Layer

This section describes the forward propagation of a basic RNN layer. In this assignment, a basic RNN layer is an RNN layer with `tanh` as the activation function. A basic recurrent layer computes a new pre-activation $\mathbf{a}^{(k)}(\mathbf{x}_t)$ by combining the activation from the previous layer $\mathbf{h}^{(k-1)}(\mathbf{x}_t)$ with the activation from the previous time step $\mathbf{h}^{(k)}(\mathbf{x}_{t-1})$. Basic recurrent neural networks may model short-term temporal dependencies but are usually not good at modeling patterns expressed over more than a few time steps. You will implement the forward propagation of a basic recurrent layer. You are not required to implement the backward propagation since PyTorch automatically does that for you. Complete the class `BasicRNNCell` in `basic_rnn_cell.py` using the following definition of the forward propagation of a basic recurrent hyperbolic tangent layer.

Given parameters $\mathbf{b}, \mathbf{W}$, and $\mathbf{V}$, the forward propagation implements the following operation:

$$\mathbf{a}^{(k)}(\mathbf{x}_t) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x}_t) + \mathbf{V}^{(k)}\mathbf{h}^{(k)}(\mathbf{x}_{t-1}) \tag{6}$$

$$\mathbf{h}^{(k)}(\mathbf{x}_t) = \tanh(\mathbf{a}^{(k)}(\mathbf{x}_t)), \tag{7}$$

where the `tanh` function is applied element-wise.

The method implementing the forward propagation of `BasicRNNCell` has the signature

```
def forward(x, h):
```

where

- `x` is the activation from the previous layer, $\mathbf{h}^{(k-1)}(\mathbf{x}_t)$. For the first layer, this is the input volume $\mathbf{x}_t$ at time step t
- `h` is the hidden state from the previous time step, $\mathbf{h}^{(k)}(\mathbf{x}_{t-1})$. In this assignment, we set `h` to a tensor of zeros for the first time step.

Implement Eqs. 6-7 in the forward method. The method should return the output activation (new hidden state) $\mathbf{h}^{(k)}(\mathbf{x}_t)$.

The constructor of `BasicRNNCell` has the signature

```
def __init__(vocab_size, hidden_size):
```

where

- `vocab_size` is the number of input features
- `hidden_size` is the number of units in the layer.

Complete the constructor by declaring instance variables for the parameters $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{n \times m}$ and $\mathbf{V} \in \mathbb{R}^{m \times m}$ where n is the number of input features and m is the number of hidden units. Initialize all the parameters uniformly at random from the range $[-k, k]$ where $k = \sqrt{\frac{1}{m}}$.

2.2 Recurrent Long Short Term Memory

This section describes the forward propagation of an LSTM layer. LSTMs are recurrent layers that are associated with a memory cell. At every time step, the layer removes some information from the memory cell using a *forget gate*; it computes a cell pre-activation and adds some of the pre-activation to the cell using an *input gate*; finally, it propagates part of the information in the cell forward using an *output gate*. Like the `BasicRNNCell` you will implement only the forward propagation of `LSTMCell`. PyTorch will implement the backward propagation for you.

The forward propagation of an LSTM layer is defined as follows:

$$\mathbf{a}^{(k)}(\mathbf{x}_t) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x}_t) + \mathbf{V}^{(k)}\mathbf{h}^{(k)}(\mathbf{x}_{t-1}) \tag{8}$$

$$= \left[\mathbf{a}_i^{(k)}(\mathbf{x}_t),\, \mathbf{a}_f^{(k)}(\mathbf{x}_t),\, \mathbf{a}_o^{(k)}(\mathbf{x}_t),\, \mathbf{a}_g^{(k)}(\mathbf{x}_t)\right]^T \tag{9}$$

$$\mathbf{i}^{(k)}(\mathbf{x}_t) = \sigma\left(\mathbf{a}_i^{(k)}(\mathbf{x}_t)\right) \tag{10}$$

$$\mathbf{f}^{(k)}(\mathbf{x}_t) = \sigma\left(\mathbf{a}_f^{(k)}(\mathbf{x}_t)\right) \tag{11}$$

$$\mathbf{o}^{(k)}(\mathbf{x}_t) = \sigma\left(\mathbf{a}_o^{(k)}(\mathbf{x}_t)\right) \tag{12}$$

$$\mathbf{g}^{(k)}(\mathbf{x}_t) = \tanh\left(\mathbf{a}_g^{(k)}(\mathbf{x}_t)\right) \tag{13}$$

$$\mathbf{c}^{(k)}(\mathbf{x}_t) = \mathbf{i}^{(k)}(\mathbf{x}_t) \odot \mathbf{g}^{(k)}(\mathbf{x}_t) + \mathbf{f}^{(k)}(\mathbf{x}_t) \odot \mathbf{c}^{(k)}(\mathbf{x}_{t-1}) \tag{14}$$

$$\mathbf{h}^{(k)}(\mathbf{x}_t) = \mathbf{o}^{(k)}(\mathbf{x}_t) \odot \tanh\left(\mathbf{c}^{(k)}(\mathbf{x}_t)\right), \tag{15}$$

where $\odot$ is the element-wise product. The sigmoid and hyperbolic tangent functions are also applied element-wise.

Fill in the method implementing the forward propagation of `LSTMCell` which has the signature

```
def forward(x, h, c):
```

where `c` is the cell state from the previous time step, $\mathbf{c}^{(k)}(\mathbf{x}_{t-1})$. `x` and `h` are the activations from the previous layer and previous time step respectively as described in Section 2.1. Like `h`, `c` is set to a tensor of zeros for the first time step. Implement Eqs. 8-15 in the forward method. The method should return the output activation $\mathbf{h}^{(k)}(\mathbf{x}_t)$ and the new cell state $\mathbf{c}^{(k)}(\mathbf{x}_t)$.

Please remember to complete the constructor of `LSTMCell` by creating the parameters $\mathbf{b} \in \mathbb{R}^p, \mathbf{W} \in \mathbb{R}^{n \times p}$ and $\mathbf{V} \in \mathbb{R}^{m \times p}$ where $p = 4 \cdot m$. Initialize all the parameters uniformly at random from the range $[-k, k]$. n, m and k have the same meaning as defined in Section 2.1.

3 Specifying the Network

You will now build a recurrent neural network using the layers you have defined. This section details the code stub that is partially implemented for you.

1. Read the constructor of `CustomRNN` in `create_rnn.py` which has the signature

   ```
   def __init__(vocab_size, hidden_size, num_layers, rnn_type):
   ```

   Where

   - `vocab_size` is an integer describing the number of input features.
   - `hidden_size` is an integer describing the number of units in each layer of the network. In this assignment, we will follow the common design paradigm of RNNs by assigning the same number of units to all layers of the network.
   - `num_layers` is an integer describing the number of layers in the RNN.
   - `rnn_type` $\in$ {`basic_rnn`, `lstm_rnn`} is a string indicating the type of RNN layers. If `rnn_type` is '`basic_rnn`', the network should use layers of type `BasicRNNCell` . `LSTMCell` layers should be used if `rnn_type` is '`lstm_rnn`'.

   We start by creating a `ModuleList` and then add the required number of layers of the appropriate type to the list. `ModuleList` is one of the specialized data structures in PyTorch along with `ModuleDict` for organizing modules. `ModuleList` operates just like a Python list. Thus it implements all the operations of normal python lists such as `append`.

   The number of input features for the first layer is the `vocab_size`; the number of input features for all other layers is the `hidden_size`.

2. Complete the method implementing the forward propagation of `CustomRNN` which has the signature

   ```
   def forward(x, h, c):
   ```

   where

   - `x` $\in \mathbb{R}^{B \times T \times n}$ is a mini-batch of input sequences where B is the mini-batch size, T is the length of the sequence (number of time steps) and n is the number of input features. For this assignment, the number of input features is equivalent to the number of unique characters in the corpus.
   - `h` $\in \mathbb{R}^{l \times B \times m}$ is the initial hidden state(s). l and m are the number of layers and number of hidden units in the network respectively.
   - `c` $\in \mathbb{R}^{l \times B \times m}$ is the initial cell state(s). c is used only if the RNN uses layers of type `LSTMCell`.

The `forward` method should return

- `out` $\in \mathbb{R}^{B \times T \times m}$ the hidden states of the last layer at each time step.
- `h` $\in \mathbb{R}^{l \times B \times m}$ the hidden states of each layer at the end of the last time step.
- `c` $\in \mathbb{R}^{l \times B \times m}$ the cell state of each layer at the end of the last time step. Return the `c` passed as input if the RNN uses `BasicRNNCell` layers.

4 Training Tasks

4.1 Baseline Models

This section refers to the following code stubs which you must complete to create and train the networks. You will specify values to read the data, create the appropriate RNN models and then train the models on the data. Complete the function `text_generation` in `text_generation.py` which has the signature.

```
def text_generation(rnn_type, input_file, sample_len, seed):
```

where

- `rnn_type` $\in$ {`basic_rnn`, `lstm_rnn`} is the type of RNN.
- `input_file` is the file path to the dataset.
- `sample_len` is the number of characters you want to generate. You can set this parameter to any value you want. Note that this parameter is different from the sequence length as described above.
- `seed` is a prompt for the model to start generating new characters. We will set the seed to "KING LEAR" if you do not provide a seed or if the seed is invalid.

Fill in the code blocks for a `basic RNN` and an `LSTM RNN` using the following model parameters and `train_opts`:

- sequence length: 100
- number of layers: 2
- number of hidden units
  - `Basic RNN`: 128
  - `LSTM RNN`: 256
- batch size: 128
- number of epochs: 120
- learning rate: 0.001
- weight decay: 0.0001

Run the script `text_generation.py` once you have completed the function to train a `basic RNN` and `LSTM RNN` models. The function will automatically invoke the sampling script after the models have been trained.

Besides the baseline training schedule we have recommended, you may experiment with other training heuristics such as implementing gradient clipping which may lead to better results. Table 1 shows the average classification accuracy on the training and the validation sets of our base models.

model	train	validation
basic rnn	53.00%	50.16%
lstm rnn	55.35%	49.84%

Table 1: Final Accuracy

4.2 Generating Text One Character at a Time

We have provided a utility function `sample.py` which will enable your models to compose essays given a prompt. The prompt (seed) can be anything between one character and several characters. The sampling script samples from the output of the model and uses the sample as the next input until the desired text length is reached. In addition to the prompt, you may specify other relevant arguments such as the length of text to generate and whether or not to include the prompt as part of the generated text. The script will print the text after it has finished generating it. It will also save the generated text to file.

You will not be graded based upon the generated output. However, you may find it helpful/interesting to read the essays your models compose. We have included the output text from our two baseline models which were trained on `shakespeare.txt` (named '`basic_rnn.txt`' and '`lstm_rnn.txt`') as reference. You can use the optional datasets or compile your own dataset, train RNN models and then compose essays with them for fun.

5 Academic integrity

This homework assignment must be done individually. Sharing code or model specifications are strictly prohibited. Homework discussions are allowed only on Piazza. You are not allowed to search online for auxiliary software, reference models, architecture specifications, or additional data to solve the homework assignment.

Your submission must be entirely your own work. That is, the code and the answers that you submit must be created, typed, and documented by you alone, based exclusively on the materials discussed in class, and released with the homework assignment. You can obviously consult the class slides posted in Canvas, your lecture notes, and the textbook. **Important:** the models for this homework assignment must be trained **exclusively** on the data provided with this assignment. The improvements made to the base model must be your own. These rules will be strictly enforced, and any violation will be treated seriously.

6 Submission instructions and Grading

You must submit the following code and data files:

- `basic_rnn_cell.py`: Complete the code stub for computing a recurrent layer having a hyperbolic tangent activation function. (**30 points**)
- `lstm_cell.py`: Complete the code stub for computing a recurrent layer having an LSTM cell. (**30 points**)
- `create_rnn.py`: Complete the code stub for building an RNN. (**20 points**)
- `text_generation.py`: Complete the script that trains and evaluates a recurrent network using hyperbolic tangent activations and LSTM cells. (**10 points**)
- `basic_rnn.pt`: Submit the data file storing your basic RNN model after 120 training epochs. (**5 points**)
- `lstm_rnn.pt`: Submit the data file storing your LSTM model after 120 training epochs. (**5 points**)
- `rnn.py`: The RNN wrapper we provided.
- `train.py`: The training script we provided.
- `sample.py`: The sampling script we provided.
- `create_sequence.py`: The script for creating sequences we provided.
- `shakespeare.txt`: The training data we provided.

Your code will be graded based on whether it runs without error and whether it produces expected outputs given expected inputs. In general, partial credit will not be awarded for code that does not pass our evaluations. Your models will be graded based upon whether they meet or exceed our baseline performance on a hidden test set. If the final error of your model on the validation set is similar to the performance shown in Table 1 then it will likely meet the baseline performance on the test set.

We suggest that you keep all of your work in the same subdirectory and use `create_submission.py` to create your zipped submission. The script will fail if you have forgotten to include any of the required code or data files. Please be careful and make sure that the script includes the correct files. For instance, if you have multiple files named '`create_rnn.py`' you may inadvertently submit the wrong code. We cannot accommodate such mistakes.