Project Part 2 - RNNs in PyTorch

ARIZONA STATE UNIVERSITY
SCHOOL OF ELECTRICAL, COMPUTER,
AND ENERGY ENGINEERING,
EEE598: Deep Learning for Media Processing & Understanding

1 Objectives

- Learn how to use RNNs, LSTMs, and GRUs in PyTorch
- Use recurrent modules for sequence modeling tasks

2 RNN Background

Convolutional neural networks work well for certain types of data that can be made into fixed size inputs. But what if the data cannot be assumed to be of fixed size? This is common in problems that consider some sort of time series input like speech or text.

If the input data cannot be assumed to be a fixed size, we need some way to have our network automatically adapt to the different sized inputs. One way to do this is to use *recurrent neurons*, where a neuron's output can be used as an input to the same neuron at a different time step (Figure 1). With this formulation we can "unroll" the network for as many time steps as needed to process the data (Figure 2). The unrolled representation can be used to perform backpropagation.

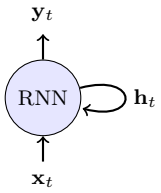

Figure 1: Basic RNN Unit.

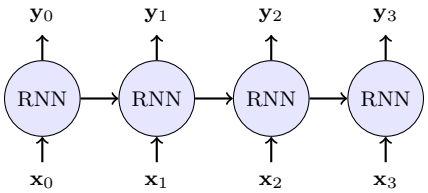

Figure 2: Unrolled Recurrent Network.

There are several "flavors" of recurrent units. In this project we will consider three common variants: vanilla recurrent neurons (RNN), long-short term memory units (LSTM) [1], and gated

recurrent units (GRU) [2]. We will use these recurrent units to create a network that can classify text.

2.1 RNN

The equation for the hidden state of a vanilla RNN layer takes the following form:

$$\mathbf{h}_t = \tanh\left(\mathbf{W}_x\mathbf{x}_t + \mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{b}_h\right) \tag{1}$$

where $\mathbf{x}_t$ is an input at time t, and $\mathbf{h}_t$ is the corresponding state vector. The $\mathbf{W}$ matrices and $\mathbf{b}$ vectors represent learnable parameters.

RNN layers can be used in PyTorch with the `nn.RNN` layer. The main arguments in the initialization of this layer are the input feature dimension and output feature dimension. The layer can infer the batch size and the sequence size from the input data. The output feature dimension isa parameter that you must set yourself to achieve good performance.

In the forward pass of the RNN layer we need to provide both an input, as well as the hidden state tensor. The input is a tensor of size (sequence size, batch size, input feature size), and the hidden state is a tensor of size (1, batch size, output feature size). For our application we can initialize the hidden state to be a tensor of all 0s. The forward pass of the layer will return a hidden state tensor for all positions in the sequence of size (sequence size, batch size, output feature size), and the final value of the hidden state of size (1, batch size, output feature size).

The output of the RNN is based on the value of the hidden state. The form of the output is identical to a fully connected layer:

$$\mathbf{y}_t = \text{softmax}\left(\mathbf{W}_y\mathbf{h}_t + \mathbf{b}_y\right) \tag{2}$$

where $\mathbf{y}_t$ is an output at time t. For this project we are only interested in the last output. The `nn.RNN` layer itself does not compute this output. To implement the output we need to use a separate fully connected layer in PyTorch (`nn.Linear`).

2.2 LSTM

The long-short term memory unit (LSTM) is a more complex recurrent node that incorporates different "gates" to allow or reject information from passing through the network. There is an input gate, and output gate, and a forget gate that controls the flow of information. Additionally, the LSTM has both a memory cell and a hidden state. The LSTM is computed as:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f x_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_c x_t + \mathbf{U}_c h_{t-1} + \mathbf{b}_c) \\
\mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{c}_t)
\end{aligned}$$

where $\circ$ is the Hadamard product (e.g., elementwise product), σ is a sigmoid function, $\mathbf{f}$ is the forget gate, $\mathbf{i}$ is the input gate, $\mathbf{o}$ is the output gate, $\mathbf{c}$ is the memory cell, and $\mathbf{h}$ is the hidden state. $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{b}$ represent learnable parameters.

LSTM layers can be used in PyTorch with the `nn.LSTM`. As with the RNN, the main arguments in the initialization of this layer are the input feature dimension and output feature dimension. The output feature dimension is a parameter that you must set yourself to achieve good performance.

In the forward pass of the LSTM layer we need to provide both an input, as well as a tuple of the memory cell and hidden state tensors. The input is a tensor of size (sequence size, batch size, input feature size). The memory cell and the hidden state are both of size (1, batch size, output feature

size). For our application we can initialize the hidden state and memory cell to be tensors of all 0s. The forward pass of the layer will return a hidden state tensor for all positions in the sequence of size (sequence size, batch size, output feature size), and a tuple of the final value of the hidden state and the memory cell.

The PyTorch LSTM implementation has no output $\mathbf{y}$. We can create an output using the hidden state at a certain time using equation 2. This corresponds to a fully connected layer with the hidden state as the input.

2.3 GRU

The Gated Recurrent unit GRU can be thought of as a simplified version of LSTM. In practice the performance of GRU and LSTM can often be similar. The GRU incorporates only two gates: an update gate and a reset gate. It does not use a memory cell like the LSTM. The GRU is computed by:

$$\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_z\mathbf{x}_t + \mathbf{U}_z\mathbf{h}_{t-1} + \mathbf{b}_z) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{U}_r\mathbf{h}_{t-1} + \mathbf{b}_r) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \circ \mathbf{h}_{t-1} + \mathbf{z}_t \circ \tanh(\mathbf{W}_h\mathbf{x}_t + \mathbf{U}_h(\mathbf{r}_t \circ \mathbf{h}_{t-1}) + \mathbf{b}_h)
\end{aligned}$$

where $\circ$ is the Hadamard product, σ is a sigmoid function, $\mathbf{z}$ is the update gate, $\mathbf{r}$ is the reset gate, and $\mathbf{h}$ is the hidden state. $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{b}$ represent learnable parameters.

GRU can be used in PyTorch with the `nn.GRU` layer. The main arguments in the initialization of this layer are the input feature dimension and output feature dimension. The output feature dimension isa parameter that you must set yourself to achieve good performance.

In the forward pass of the GRU layer we need to provide both an input, as well as the hidden state tensor. The input is a tensor of size (sequence size, batch size, input feature size), and the hidden state is a tensor of size (1, batch size, output feature size). For our application we can initialize the hidden state to be a tensor of all 0s. The forward pass of the layer will return a hidden state tensor for each position in the sequence of size (sequence size, batch size, output feature size), and the final value of the hidden state (1, batch size, output feature size).

The PyTorch GRU implementation has no output $\mathbf{y}$. We can create an output using the hidden state at a certain time using equation 2. This corresponds to a fully connected layer with the hidden state as the input.

3 RNNs for sentence classification

In this project we will utilize recurrent layers to classify sentences.

3.1 Datasets

In this project we will need to use `torchtext` to help us load and preprocess data. To install `torchtext` use the following command:

```
sudo pip install torchtext
```

Each group will be assigned two datasets to work on (see Section 4). Table 1 describes each dataset.

The sentiment dataset consists of movie reviews that are either positive or negative [3]. The spam dataset consists of emails that are either spam or not spam [4].

The questions dataset [5] contains questions classified by what type is the answer to the question. For example the questions "Who is Snoopy's arch-enemy?" and "What actress has received the most

Oscar nominations?" have the same class. As another example the questions "What novel inspired the movie BladeRunner?" and "What's the only work by Michelangelo that bears his signature?" also have the same class. The classes are varied enough that the network has to understand the whole sentence to classify, rather than just looking for key words like "what" or "who"

The newsgroups dataset [6] consists of newsgroup postings from twenty different categories (Newsgroups are where people posted things on the internet before Facebook and Reddit). Examples of the newsgroup classes are "comp.sys.mac.hardware", "talk.politics.guns", "rec.sport.hockey".

Each dataset is stored in two CSV files: `train.csv` and `text.csv`. Each line of the CSV file has a sentence and a label. We have provided a `load_dataset` function in `dataset.py` to load the datasets using `torchtext`. This function takes as arguments the name of the dataset and the batch size. The `load_dataset` function returns two iterators: one for training data and one for testing data. When we loop using the iterators, we will receive batches of data. The sentences and the labels can be extracted from the batch as in the following example:

```
train_iterator, test_iterator = load_dataset(...)
for batch in train_iterator:
    sentence = batch.sentence
    label = batch.label
```

The dataset loader will translate each unique word to a unique number using a lookup table. The number of unique words is the vocabulary size of the dataset.

Table 1: Datasets

Dataset	Description	# Categories	Vocab size	# Train / # Test
Sentiment	Movie review dataset [3]	2	18298	7996 / 2666
Spam	Enron spam dataset [4]	2	28049	3879 / 1293
Questions	Learning question classifiers dataset [5]	50	7694	4464 / 1488
Newsgroups	20 newsgroups dataset [6]	20	160792	14121 / 4707

3.2 Network Design

The input to our network is a batch of sentences and the output is a batch of predicted labels. From the input we first want to perform a word embedding. The embedding layer takes the words, which have been mapped to integers by the dataset loader, and outputs a vector embedding for each word. For example, the word "yes" might be mapped to the vector $[0.1, 0.7, -0.5]$ and the word "no" might be mapped to the vector $[1.1, -0.5, 0.3]$. The dimension of the embedding is a hyper-parameter that must be set. We can use the `nn.Embedding` layer for this. By default the embedding layer uses a random embedding matrix. It is also possible to load a pre-trained embedding matrix where similar words are maps to similar vectors, but for this project the default random embeddings should still give good performance.

Next we pass the embedded words into the recurrent layer. In general, we could utilize several stacks of recurrent layers, but in this project it is only required to consider a single recurrent layer stack. When calling the forward pass of a recurrent layer, the layer requires two inputs: the input $\mathbf{x}$ and the hidden state $\mathbf{h}$. Your code should initialize the hidden state to be a tensor of 0s. We want to obtain a class prediction from the output of the recurrent unit, so we can use a fully connected layer after the recurrent layer. The number of outputs of the fully connected layer should be equal to the number of classes.

Since we are doing classification, we can use the `nn.CrossEntropyLoss` loss function. This loss function incorporates the softmax internally, so we do not need to explicitly add a softmax layer to our model.

Training the sequence models is almost identical to training convolutional neural networks, as in part one of the project. We still need to loop through our data, for each batch call the forwards pass, backwards pass, call an optimizer update step, etc. One difference between the convolutional network and recurrent networks is that we have an internal state. This state must be initialized before we call the forward pass of a new batch.

Table 2: Layers for text processing

Layer	Description	Initialization Arguments
`nn.Embedding`	Embed word vectors	(vocabulary size, embedding size)
`nn.RNN`	Vanilla RNN unit	(input feature size, output feature size)
`nn.LSTM`	LSTM unit	(input feature size, output feature size)
`nn.GRU`	GRU unit	(input feature size, output feature size)
`nn.Linear`	Fully connected output	(input feature size, output feature size)

Hint: Training RNNs

Training RNNs may be a little more difficult than training convolutional neural networks. If training is not working well, you may need to fiddle with the learning rate, or try another optimizer such as ADAM [7]. ADAM can be used with the PyTorch `optim.Adam` class.

4 Deliverable

Depending on your group number, you are assigned two different datasets and one recurrent unit type (Table 3). For the assigned datasets and recurrent unit type, you must determine the hyper parameters (e.g. learning rate, batch size, embedding size, hidden state feature dimension) of the network to achieve good performance.

Submit your project as a folder named `GROUP_NUMBER_PROJECT_PART2` and zip the folder for submission. The grading rubric for this project is shown in Table 4. There are no unit tests for the project, but we will grade your code by attempting to run your code. There should be only one submission per group.

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[2] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[3] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up?: sentiment classification using machine learning techniques," in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pp. 79–86, Association for Computational Linguistics, 2002.

[4] V. Metsis, I. Androutsopoulos, and G. Paliouras, "Spam filtering with naive bayes-which naive bayes?," in *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS 2006)*, 2006.

Table 3: Group tasks

Group number	Datasets	RNN unit
1	Sentiment, Spam	LSTM
2	Sentiment, Questions	LSTM
3	Sentiment, Newsgroups	LSTM
4	Spam, Questions	LSTM
5	Spam, Newsgroups	LSTM
6	Questions, Newsgroups	LSTM
7	Sentiment, Spam	GRU
8	Sentiment, Questions	GRU
9	Sentiment, Newsgroups	GRU
10	Spam, Questions	GRU
11	Spam, Newsgroups	GRU
12	Questions, Newsgroups	GRU
13	Sentiment, Spam	LSTM
14	Sentiment, Questions	LSTM
15	Sentiment, Newsgroups	LSTM
16	Spam, Questions	LSTM
17	Spam, Newsgroups	LSTM
18	Questions, Newsgroups	LSTM
19	Sentiment, Spam	GRU
20	Sentiment, Questions	GRU
21	Sentiment, Newsgroups	GRU
22	Spam, Questions	GRU
23	Spam, Newsgroups	GRU
24	Questions, Newsgroups	GRU

Table 4: Grading rubric

Points	Description
40	Working code for first dataset
40	Working code for second dataset
10	Report final classification accuracy for both datasets
10	Submit saved models for both datasets
Total 100	

[5] X. Li and D. Roth, "Learning question classifiers," in *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pp. 1–7, Association for Computational Linguistics, 2002.

[6] K. Lang, "Newsweeder: Learning to filter netnews," in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 331–339, 1995.

[7] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.