Assignment 1
NLP 202: Natural Language Processing II

University of California Santa Cruz

Due: January 24, 2023, 11:59pm

This assignment is to be done in Python 3.

Most of your grade will depend on the quality of the report. You should submit it as a PDF. We *strongly* recommend typesetting your scientific writing using LaTeX. Some free tools that might help: Overleaf (online), TexLive (cross-platform), MacTex (Mac), and TexStudio (Windows).

1 Task

The purpose of this homework is to understand how to implement minibatching in PyTorch. You will need to implement a logistic regression model and a LSTM model in PyTorch that uses minibatching on the sentiment analysis task. One goal of sentiment analysis is to determine if a text (e.g., a sentence) has positive or negative sentiment. You will be using the IMDB reviews dataset (IMDB; [1]); this is a standard sentiment analysis dataset based on movie reviews. We provided a starter colab notebook that loads the data and build, train and evaluate the model for the single instance with a model correctness test to initialize model randomly (with a set seed) on some example instances then run forward on each instance and print score.

2 Padding and masking the inputs

To perform minibatching using DataLoader, all the inputs in a batch need to be of the same length. This is done by a process called **padding**, where an input sequence is padded with a padding token to make it of the same length as the maximum length input in the batch. Another method, which we don't allow on this assignment, is to group instances of the same length to formulate a batch. The downside of batching sentences of the same length is that short (and long) sentences are batched together rather than being spread evenly through the training batches, which can be less good for training. It is more common to use padding, which is the method you need to use in this assignment.
In many network architectures, in addition to padding, you must also do **masking**, which masks out the extra padding tokens so that the network computation and loss computation are the same as if you had processed the sentence without padding. This is necessary if you have a model with attention and you don't want it to attend beyond the sentence length, or if you are doing sequence labeling and you don't want add the loss for the padding tokens. In this assignment, if you use the zero vector for padding, you probably won't need to use masking. However, you will need to verify that the minibatched version of your code computes the same loss as the single-instance version (see below).

When preprocessing the data, we convert the input sequence into a sequence of indices. A sentence after preprocessing may look like this:

$$[12, 1, 5, 6]$$

If the max sentence length in a batch is 10, and the padding_id is 0, then the padded version of the sentence would be:

$$[12, 1, 5, 6, 0, 0, 0, 0, 0, 0]$$

To do the padding, you may use the *pad_sequence()*, *pack_padded_sequence()* and *pad_packed_sequence()* methods defined in torch.nn.utils.rnn

3 Building a logistic regression model(50%)

The model should contain 2 components as the following:

- (Embedding layer) Embed the input text as a sequence of vectors.
- (Linear output layer) Apply a feed-forward liner layer on that vector to obtain a label.

We have already provided starter code for the non-batched version of the code in the starter colab notebook. Modify the code to perform minibatching.

Model correctness testing You will need to verify that the minibatched version of your code computes the same loss as the single-instance version. In the stater code, we have provided code to output the loss of the single instance on some test examples with the initial parameters. Note that the model parameters will be the same across runs because we set a fixed seed. You should write your own code to test your model with minibatching for the same number of instances and check if output scores approximately the same with the single instance model with the same fixed model parameters.

Hyperparameter tuning You will need to make several hyperparameter decisions, including but not limited to: number of training epochs, training batch size, choice of nonlinearity, choice of optimization method, choice of loss function, dropout rate, L2 regularization strength, and learning rate. You are required to at least tune the batch size and learning rate and record the corresponding properties:

- Record training time of different batch sizes.
- Record model accuracy on dev set of different batch sizes.
- For a specific batch size, record model accuracy on dev set of different learning rate.

Deliverables In the writeup, be sure to fully describe your models and experimental procedure. Provide graphs, tables, charts or other summary evidence to support any claims you make.

1. Are your minibatching output scores exactly the same as the single instanced model? Why or why not?
2. Make a plot to compare training time of different batch sizes.
3. Make a plot to compare model accuracy on dev set of different batch sizes.
4. Tune the learning rate for a specific batch size and make a plot to compare model accuracy on dev set of different learning rates.
5. Report the accuracy of your best model on the dev and test sets after training.
6. Submit the outputs of the best model for both the dev and test sets.

7. Look at the output on the dev set and compare to the gold labels. What are your observations about the errors your model makes?

4 Building a model with LSTM feature extractor(50%)

Implement a classification model that runs an LSTM on the word embeddings, followed by average pooling on the resulting hidden state vectors, and passes the vector to a feedforward neural network to make the prediction. To do this, you will need to add an LSTM layer between the embedding layer and the linear output layer. In PyTorch, before you pass your embeddings to the LSTM, you need to pack them, which you do with *packed_padded_sequence*. This will cause our LSTM to only process the non-padded elements of our sequence. The LSTM will then return packed_output (a packed sequence) as well as the hidden and cell states (both of which are tensors). You then unpack the output sequence, with *pad_packed_sequence*, to transform it from a packed sequence to a tensor. The elements of output from padding tokens will be zero tensors (tensors where every element is zero).

Model correctness testing Check your minibatch implementation for correctness as you did in the logistic regression model.

Hyperparameter tuning As in the logistic regression model, you will need to make several hyperparameter decisions. You are required to at least tune the batch size and learning rate and record the corresponding properties:

- Record training time of different batch sizes.
- Record model accuracy on dev set of different batch sizes.
- For a specific batch size, record model accuracy on dev set of different learning rate.

Deliverables In the writeup, be sure to fully describe your models and experimental procedure. Provide graphs, tables, charts or other summary evidence to support any claims you make.

1. Are your minibatching output scores exactly the same as the single instanced model? Why or why not?
2. Make a plot to compare training time of different batch sizes.
3. Make a plot to compare model accuracy on dev set of different batch sizes.
4. Tune the learning rate for a specific batch size and make a plot to compare model accuracy on dev set of different learning rates.
5. Report the accuracy of your best model on the dev and test sets after training.
6. Submit the outputs of the best model for both the dev and test sets.
7. Look at the output on the dev set and compare to the gold labels. What are your observations about the errors your model makes?

5 Useful PyTorch Functions

To handle and manipulate your tensors, you may need to use some of the following tensor operations/methods. Further documentation at https://pytorch.org/docs/stable/tensors.html

- **view() and reshape()**
  tensor.view() or tensor.reshape() returns a new tensor with the same data and number of elements as the

old one, but a different shape specified in the parameters. Refer to the examples here. -1 is used as a parameter to derive the length of one of the dimensions based on the other dimensions and the number of elements, and only one dimension may be specified as -1.

- **squeeze(), unsqueeze()**
  squeeze() and unsqueeze() are reverse operations of each other. tensor.squeeze(input) returns a tensor with all the dimensions of input of size 1 removed. tensor.unsqueeze(input, dim) returns a new tensor with a dimension of size 1 inserted at the specified position (dim). Documentation for squeeze and unsqueeze
- **contiguous()**
  tensor.contiguous() makes a copy of tensor so the order of the elements would be the same as if a tensor of the same shape was created from scratch.
- **transpose()**
  transpose(input, dim0, dim1) returns a tensor that is a transposed version of input. The dimensions dim0 and dim1 are swapped.
- **permute()**
  tensor.permute(*dims) rearranges the original tensor according to the desired ordering (*dims) and returns a new multidimensional rotated tensor
- **flatten()**
  flatten(input, start_dim=0, end_dim=-1) flattens a contiguous range of dims in input. Examples here
- **expand()**
  tensor.expand() returns a new view of the self tensor with singleton dimensions expanded to a larger size. Passing -1 as the size for a dimension does not change the size of that dimension.
- **scatter_() and gather()**
  scatter_() and gather() are reverse operations of each other. gather(input, dim, index) gathers values of indices along an axis specified by dim. scatter_(dim, index, src) writes all values from the tensor src into self at the indices specified in the index tensor.
- **masked_select()**
  masked_select(input, mask) returns a new 1-D tensor which indexes the input tensor according to the boolean mask (which is a BoolTensor).
- **masked_scatter()**
  tensor.masked_scatter_(mask, source) copies elements from source into tensor at positions where the mask is one.
- **eq(), ge()**
  tensor.eq(other) or tensor.ge(other) computes element-wise equality or $\geq$ for tensor and other, where other can be a number or a tensor whose shape is broadcastable with the first argument..

Submission Instructions

Submit a zip file (`A1.zip`) on Canvas, containing the following:

- **Output**: Submit the output of your code on the dev and test sets if it is asked for in each section.
- **Code**: You code should be implemented in Python 3, and needs to be runnable. Submit your code together with a neatly written README file to instruct how to run your code with different settings. We assume that you always follow good practice of coding (commenting, structuring), and these factors are not central to your grade. However, please provide well commented code if you want partial credit. If you have multiple files, provide a short description in the preamble of each file.
- **Report**: As noted above, your writeup should be four pages long, or less, in PDF (one-inch margins,

reasonable font sizes). Include your name at the top of the report. Part of the training we aim to give you in this class includes practice with technical writing. Organize your report as neatly as possible, and articulate your thoughts as clearly as possible. We prefer quality over quantity. Do not flood the report with tangential information such as low-level documentation of your code that belongs in code comments or the README. Similarly, when discussing the experimental results, do not copy and paste the entire system output directly to the report. Instead, create tables and figures to organize the experimental results.

References

[1] 2011 Andrew L. Maas et al. Imdb large movie review dataset. URL `http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf`.