Language Modeling with LSTM and ngram

1 Language Modeling

The task of language modeling, quite simply, deals with building a model for the language(s) concerned. This means that the sequences of tokens (words, sub-words, or characters) that occur frequently in the language should be rated higher with respect to some scoring function. In a probabilistic language model (like the one you will be building), this scoring function is a probability distribution over possible sequences. Hence, a normal sentence like "The dog chased the cat " should be more probable than a non-sensical sentence like "chased the the dog cat":

$$P(\text{“}The\ dog\ chased\ the\ cat\text{''}) >>> P(\text{“}chased\ the\ the\ dog\ cat\text{''})$$

Generally, every word depends on the words that have previously been seen. Therefore, the probability of a sequence is often seen as a product of seeing every word in the sequence given all (or some) of its respective previous words. That is,

$$\begin{aligned} P(\text{“}The\ dog\ chased\ the\ cat\text{''}) = & P(\text{“}cat\text{''}|\text{“}The\ dog\ chased\ the\text{''}) \\ & \times P(\text{“}the\text{''}|\text{“}The\ dog\ chased\text{''}) \\ & \times \ldots \times P(\text{"}The\text{"}|\phi) \end{aligned}$$

or more generally, for a sequence $\{w_1, w_2, \ldots, w_n\}$:

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{i=n} P(w_i | w_1, \ldots, w_{i-1})$$

Hence, a language model defined this way can be seen as a next token predictor given a sequence of previous words. One can build this predictor using deep neural networks.

Evaluation. Language models are evaluated using the Perplexity metric. Perplexity measures the confusion of predicting a given target word.

We can define perplexity in terms of a cross-entropy loss. (As an exercise, convince yourself that the definition shown below is mathematically the same as the one we saw in class.) Let us assume a model with parameters θ which predicts a probability distribution $\hat{y}$ over the vocabulary (V) at every position in the sentence. The cross-entropy loss given for a given token at position i is given by:

$$Loss_i(\theta) = -\sum_{j=1}^{|V|} y_{i,j} \times log(\hat{y}_{i,j})$$

where $y_{i,j}$ is 1 for the index corresponding to the target token and 0 elsewhere.

One can aggregate the loss over all the tokens (T) in the sentence:

$$Loss(\theta) = -\frac{1}{|T|} \sum_{i=1}^{|T|} \sum_{j=1}^{|V|} y_{i,j} \times log(\hat{y}_{i,j})$$

Perplexity is then computed as:

$$\text{Perplexity}(\theta) = 2^{Loss(\theta)}$$

Lower the perplexity, the better the language model models your target set.

2 A Simple N-gram Language Model

A simple statistical model that one can build is the n-gram model. An n-gram is a sequence of n tokens (words, or characters). An n-gram model computes the probability of an n-gram given the previous n-1 tokens. A simple way of computing this probability is to use the relative frequency:

$$P(w_i | w_{i-n+1} \ldots w_{i-1}) = \frac{count(w_{i-n+1} \ldots w_{i-1}, w_i)}{count(w_{i-n+1} \ldots w_{i-1})}$$

where the numerator counts the number of occurrences of the n-gram, and the denominator is the number of times the (n-1)-gram $\mathrm{w}_{i-n+1} \ldots \mathrm{w}_{i-1}$ occurs. These occurrences are usually obtained using a train set. Once, you get these probabilities, one can use them to evaluate the test set via the perplexity metric.

As you can imagine if an n-gram in the test set never occurs, it gets a zero probability. This is a stringent condition considering that many valid n-grams might not occur in the training set. To relax this condition, we modify the probability computation to assign some probability mass to such cases. This is called **smoothing**. The simplest kind of smoothing is add-one or Laplace smoothing. This is done as follows:

$$P(w_i|w_{i-n+1}\dots w_{i-1}) = \frac{count(w_{i-n+1}\dots w_{i-1}, w_i) + 1}{count(w_{i-n+1}\dots w_{i-1}) + |V|}$$

where $|V|$ is the size of the vocabulary.

3 Language Model for the Mini Project

In this mini-project, you will be implementing a *character*-level language model. That is, you will be building a model which predicts a character given a sequence of characters. You will be using an LSTM Hochreiter and Schmidhuber [1997][1] based recurrent neural network to make these predictions.

3.1 The Model

You will be building an LSTM-based recurrent model that takes in a trainable character embedding. A LSTM cell should produce an output vector, a hidden state vector, and a context vector. You should pass the output vector through a two-layer feed-forward network and predict a distribution over the character vocabulary. The following equations describe a forward pass for some position i :

$$emb_i = onehot(w_i).W_{emb}$$

where w_i is the character at the i^{th} position, and W_{emb} is a trainable projection matrix. Just like the previous mini-project, you may use `torch.nn.Embedding` to compute the emb_i. Next, we need to feed the embedding to the LSTM layer. This can be done as follows:

$$out_i, h_i, c_i = LSTM(emb_i, h_{i-1}, c_{i-1})$$

where h_{i-1} and c_{i-1} are the hidden and context vectors respectively output from the preceding position. Note that LSTM layers can be trivially stacked such that the out_i of the first layer is fed as input to the next layer. In multi-layer LSTM, every layer will have its own hidden and context vector. The vector out_i corresponds to the output vector of the LSTM. Next, we feed the output vector to a two-layer feed-forward network to obtain a probability distribution ($\hat{y}_i$) over the vocabulary of characters.

$$\hat{y}_i = softmax((ReLU(out_i.W_1 + b_1)).W_2 + b_2)$$

The distribution $\hat{y}_i$ can then be used to compute training losses. An `argmax` over the distribution should give you the next predicted character.

[1]Refer to Chris Olah's blog for a detailed explanation: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Teacher Forcing. Since we are modeling a recurrent neural network, by definition, the output of a forward pass at position i should serve as input to the forward pass at position $i+1$. Ideally, we would want the output at position i to be correct. However, often, this is not the case. Using erroneous predictions as input for the next forward pass can make training difficult. Hence, we use the gold input from at every position to obtain the predictions during the training phase. This is called teacher forcing. The difference between a forward pass with and without teacher forcing is shown in Figure 1. You cannot use teacher forcing for prediction as the gold predictions are not available. That is, the output at a certain position i should serve as the input for the pass at position $i+1$ even if the prediction is erroneous.

3.2 Files

We have provided the following files to you:

1. **Data files (data/{train,dev,test}/).** Each of these sub-folders contains the files for the respective splits. The files contain a sequence in every line.

2. **Vocab file (data/vocab.pkl).** A pickle file containing a mapping from a string to unique ID (Python dictionary). You can load it as follows:

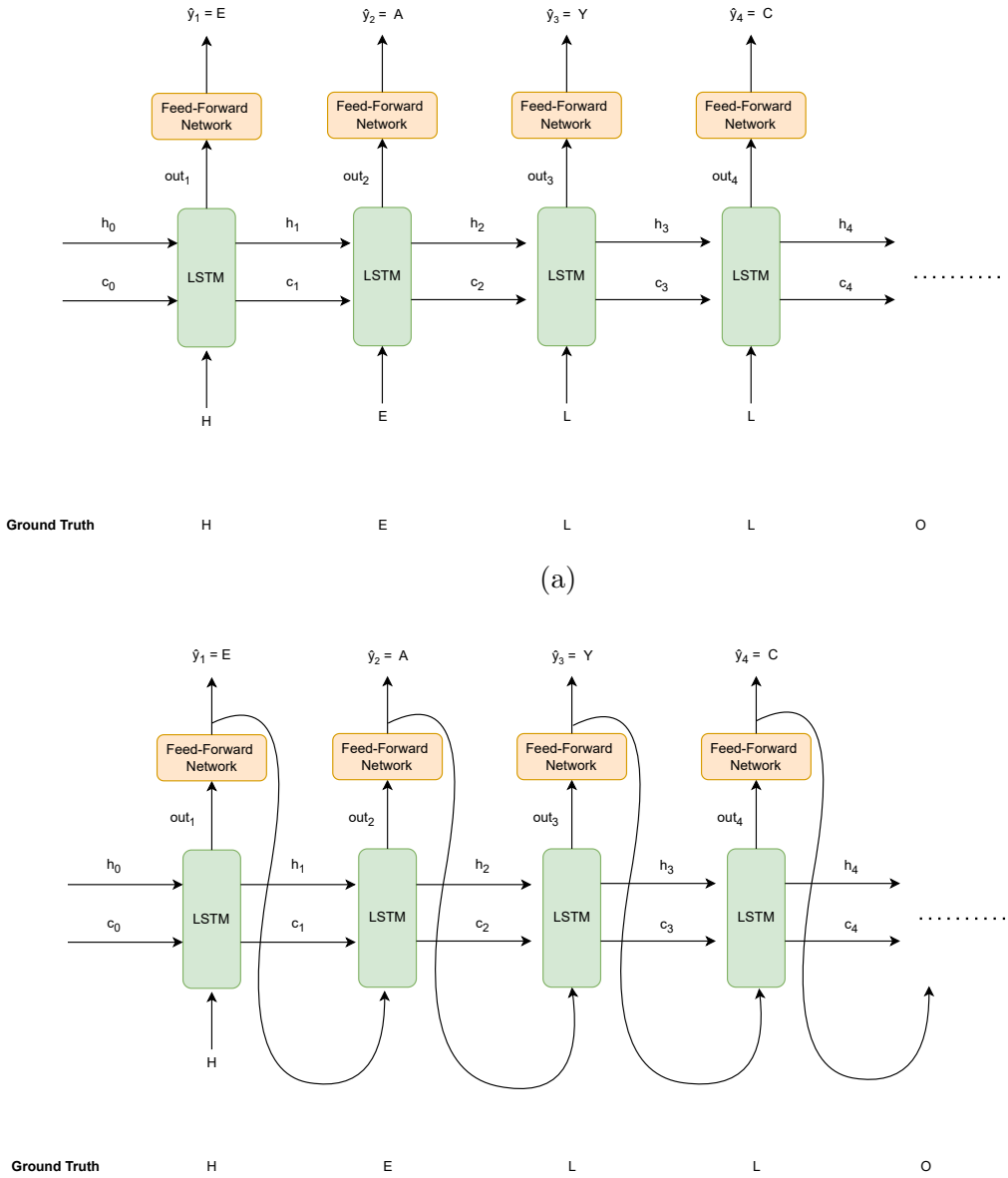
```
import pickle
with open('./data/vocab.pkl', 'rb') as f:
    vocab = pickle.load(f)
```

3. **Helper Methods (scripts/utils.py).** Contains helper methods to get files from a folder, mapping files and lines to indices.

3.3 Pre-processing Data

Sequences in natural language can be arbitrarily long and longer sequences have higher memory requirements. We need to pre-process the data appropriately to avoid these issues. For this mini-project, we will break sequences into fixed-length sub-sequences which can be later used to form data batches. That is, if we have a sequence like "Hello World" and a fixed sub-sequence length(k) of four, the sequence is broken down into three sub-sequences: i) `['H','e','l','l']`, ii) `['o',' ','W','o']`, and iii) `['r','l','d','[PAD]']`. Since the last sub-sequence is incomplete, we pad it appropriately with the '[PAD]' token. Note that for character-level language modeling, each character has a corresponding target which is the next character. Consequently, you can generate the target sub-sequences. For the above case it will be as follows: i) `['e','l','l','o']`, ii) `[' ','W','o','r']`, and iii) `['l','d','[PAD]','[PAD]']`.

<unk> tokens. Many low-frequency characters are not included in the vocabulary file. When a character that is out-of-vocabulary(OOV) is encountered, it has to be converted to the <unk> token present in the vocabulary.



(a)

(b)

Figure 1: Illustration of a forward pass (a) with teacher forcing, and (b) without teacher forcing. Note that the LSTM and feed-forward network weights are shared at each step.

3.4 Useful PyTorch Tools

You may night to use the following torch classes/methods:

1. **torch.nn.LSTM** This is PyTorch's implementation of the LSTM network. You may need to use the `num_layers` and `batch_first` arguments.

2. **torch.nn.CrossEntropyLoss** You will be asked to weight your losses to work-around the frequency imbalance in characters. The loss weight (lw_t) of a certain character t in the vocabulary is defined by:

$$lw_t = 1 - \frac{count(t)}{\sum_j^{|V|} count(j)} \tag{1}$$

   where $|V|$ is the size of the vocabulary. You may use the `weight` argument to set the weighted losses once they are computed.
   Further, we are not interested in predicting the '[PAD]' sequences. You can ignore loss and gradient computation for the '[PAD]' tokens by setting the `ignore_index` argument to the '[PAD]' token index.
   During evaluation, you might need character-wise losses. This can be obtained by setting `reduce=False`.

3.5 Parameters

1. $d_{emb} = 50$. Embedding dimension for the character embeddings.

2. h = 200. The hidden LSTM dimension.

3. |V| = 386. Number of distinct actions.

4. k=500. Subsequent length.

5. max_eps = 5. Maximum training epochs.

6. num_layers={1,2}. Number of LSTM layers.

3.6 Hyper-parameters

Just like the previous mini projects, the only tunable hyper-parameter is the learning rate. Train your model using the following learning rates: {0.0001, 0.00001, 0.000001}. Run the training loop for a maximum of 5 epochs. The best model across learning rate and epochs is the one that gets the lowest dev perplexity. Feel free to choose the batch size. **Please fix the random seed in your implementation.**

Benchmark information. A training epoch roughly takes 40 minutes with a batch size of 64 on a Titan X machine. It requires roughly 6 GB of GPU RAM. You can ask for a GPU by setting the `--gres=gpu:1` flag.

4 What to Report

1. [25 points] Implement a simple 4-gram model (again a character level one). Report the number of parameters (number of conditional probabilities) and the perplexity on the test set. Note that you need to use teacher forcing when computing test perplexity.

2. [40 points] Train an LSTM network like the one explained above. The network should be trained using weighted losses as mentioned in §3.4. Report the test perplexity for a one-layer and a two-layer LSTM network. You should also report the number of parameters in both networks (including the embedding and feed-forward layers). You can do that using:

```
num_param = sum(p.numel() for p in model.parameters())
```

3. [15 points] Compare the n-gram and the LSTM models and comment on your findings.

4. [20 points] Generate the next 200 characters given the seed sequences given below. Note that for prediction you should use a sampling strategy instead of predicting the `argmax`. This can be done using `torch.multinomial`.

   (a) The little boy was

   (b) Once upon a time in

   (c) With the target in

   (d) Capitals are big cities. For example,

   (e) A cheap alternative to

References

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.