University of Paris-Saclay - UFR Sciences
Lab 2 report - Language modeling

Diyun LU - Marwan MASHRA - Yihan ZHONG

December 6, 2022

1 Introduction

1.1 Neural n-gram model

N-gram is a language model, which is a probability model. The input of this model is a sentence, and the output is the probability of this sentence, which is the joint probability of the words in this sentence.

$$P(S) = p(w_1, w_2, w_3, ...w_n) = p(w_1)p(w_2|w_1)...p(w_n|w_n - 1...w_2w_1)$$

However, there are two problems :

1. The parameter space is too large, and the probability P has $O(n)$ parameters.
2. The data is sparse, and words may not appear at the same time, especially when the combination order is high.

In order to solve the first problem, we introduce the Markov Assumption: the appearance of a word is only related to n words before it. The n of n-gram can be taken very high, but in reality, bi-gram and tri-gram are usually enough. For the sparsity problem, assume that we have 20k words, then we will have C^2_{20k} almost 0.2 billion combinations. Many of these combinations do not appear in the corpus, and the combination probability obtained according to the maximum likelihood estimation will be 0. Therefore, we need to perform data smoothing, so that to make the sum of all n-gram probabilities be 1, and make all n-gram probabilities not be 0. Its essence is to redistribute the entire probability space, which reduces the probability of n-grams that have appeared, and supplements n-grams that have not appeared before.

N-gram can be used in many NLP tasks, such as part-of-speech tagging, text classification, words segmentation, machine translation etc.

On the other hand, n-grams has another disadvantage which is that languages have a long-distance dependency. For example, consider the following sentence: "The dog, I saw on the street last Tuesday, is cute." If we want to predict the probability of the last word "cute" with the n-grams model, the possibility of the actual association between "cute" and "Tuesday" should be very small. On the contrary, the subject "dog" of this sentence is highly correlated with cute, but the n-grams models could not capture this information. In order to alleviate the above problems, we can use Long short-term memory(LSTM).

1.2 Long short-term memory(LSTM)

LSTM is a type of RNN. Different from CNN, RNN will give an output for each moment of input combined with the state of the current model. In theory, RNN are absolutely capable of handling long-term dependency. We can carefully choose parameters for them to solve this. Unfortunately, in practice, RNN does not seem to be able to learn them.

As the neural network structure deepens, it will still bring about the problem of long-term dependence. The larger the number of network layers, the easier it is for the gradient to vanish or explode after passing through many stages of propagation during the propagation process. Then the network loses the ability to learn previous information, and it is difficult to optimize. The reason why this is more likely to occur in RNN is that as sequence data, RNN naturally has the problem that the longer the sequence, the deeper the network structure.

LSTM is explicitly designed to avoid long-term dependency problems. Memorizing information for a long time is actually their default behavior. LSTM proposes a gating mechanism with forget gate, input gate, output gate, which can selectively let information pass through. The forget gate value of LSTM can be selected between $[0, 1]$, allowing LSTM to improve the gradient vanish. We can choose it to be close to 1 to saturate the forget gate, and the long-distance information gradient does not disappear at this time. We can also choose it to be close to 0, the model deliberately blocks the gradient flow and forgets the previous information. Although LSTM can alleviate the problem of gradient vanish, it cannot avoid the problem of gradient explosion, and gradient explosion may still occur. However, due to the many gating structures of LSTM, the frequency of gradient explosion in LSTM is much lower than that of normal RNN. Gradient explosion can be solved by gradient clipping.

2 Implementation

2.1 Data preprocessing

2.1.1 Word Dictionary

We built a word dictionary that maps words and integers.

```python
class WordDict:
    def __init__(self, words):
        assert type(words) == set
        self.words_dict = dict(zip(words, range(len(words))))
        self.index_dict = dict(zip(self.words_dict.values(),self.words_dict.keys()))

    def word_to_id(self, word):
        assert type(word) == str
        return self.words_dict[word] if word in self.words_dict else self.words_dict["<unk>"]

    def id_to_word(self, idx):
        assert type(idx) == int
        return self.index_dict[idx] if idx in self.index_dict else "<unk>"

    def __len__(self):
        return len(self.words_dict)
```

Now we add all words to our vocab dictionary. We also add two special tokens $< bos >$ and $< eos >$ to the dictionary. Notice that we only add words from our train set and not dev nor test set. Later, any new unseen word will be replaced by the special token $< unk >$.

```python
train_words = set()
for sentence in train_data:
    train_words.update(sentence["text"])
train_words.update(["<bos>", "<eos>", "<unk>"])
word_dict = WordDict(train_words)
```

Finally, we process our datasets

```python
sentence_to_id = lambda sentence: list(map(word_dict.word_to_id, sentence))
process_data = lambda data: list(map(sentence_to_id, [sentence['text'] for sentence in data] ))
x_train_idx = process_data(train_data)
x_dev_idx = process_data(dev_data)
x_test_idx = process_data(test_data)
```

2.1.2 Generate input for N-grams

We are building a tri-gram model. Therefore, the context size is 2. The input data will be a tensor of shape two which contains the two token to base the predictions on. The output is a tensor of shape one, which contains the next token of the 2 previous token of the input data. Note that we also add two $< bos$ tokens in order to accept sentences of length 1. We'll detail more the batching process when talking about the training.

2.1.3 Generate input for LSTM

In the case of LSTM, the input will just be the sentences themselves (tensor of words IDs). Since each tensor is a sentence of different length, we can't have the dataset as a two-dimensional tensor. Therefore, we keep it as a list of tensors. We'll detail more the batching process when talking about the training.

2.2 Modelling

In this lab, we have implemented two different methods, N-grams and LSTM.

2.2.1 N-grams

2.2.1.1 Architecture

The N-grams model gets the embedding representation of the input tensor, stack the embeddings of the input tokens (two tokens for 3-grams), forward it through the classifier (MLP), then outputs the log probability for each word to be the next token.

```python
class nGramModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim=10,context_size=2):
        super(nGramModel, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.MLP = nn.Sequential(
            nn.Linear(context_size * embedding_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, vocab_size)
        )
    def forward(self, inputs):
        emb=self.embeddings(inputs)
        logits=self.MLP(emb)
        log_probs = F.log_softmax(logits,dim=1)
        return log_probs
```

2.2.1.2 Training

First, we need to understand the preprocessed dataset for this model. The input x is a two-dimensional tensor of shape (num_ngrams $\times$ 2). The gold output y is a one-dimensional tensor of shape (num_ngrams), corresponding to the token to be predicted from the n-gram. Therefore, batching here is quite simple, and could be done in the same way we're used to, i.e. taking a subset of size batch_size of both input and gold.

The output of the model is a two-dimensional tensor of shape (batch_size $\times$ vocab_size), i.e. for each n-gram, we have a log-probabilities vector describing the probability for each word in the vocab to follow that n-gram. Finally, we pass the output and the gold to our loss function, which is a negative log-likelihood.

2.2.2 LSTM

2.2.2.1 Architecture

This model gets the embedding representation of the input sentence, apply a variational dropout, forward it through an LSTM, gets the output representation of each token, apply another variational dropout, forward it through the classifier (MLP), then outputs a log probability vector for each token in the sentence.

2.2.2.2 Training

First, we need to understand the preprocessed dataset for this model. The input x is a list of length (num_sentences), containing all sentences. Those sentences are tensors of different sizes. Therefore, we need to pad them in order to get the embeddings. To pad them, we can either use the function *pack_padded_sequence* from PyTorch, or we can pad manually as we saw in the class:

First, we need to account for the padded token when creating the embeddings table :

```python
self.emb_table = nn.Embedding( self.vocab_size+1, self.embedding_dim, padding_idx=self.vocab_size)
```

then, we create an empty two-dimentaional tensor w and fill it with the padding tokens :

```python
w = torch.empty(len(inputs), max(len(s) for s in inputs), dtype=torch.long).to(device)
w.fill_(self.emb_table.padding_idx)
```

After that, we fill w with our input sentences, leaving the padding at the end :

```python
for i,input in enumerate(inputs):
    w[i,:input.shape[0]] = input
```

Now, we can use w to get the padded embeddings, forward through the different layers of our network as described previously, and obtain the log-probabilities, which is a three-dimensional tensor of shape (batch_size $\times$ max_len_sentence $\times$ vocab_size). We can't pass it yet to the loss function because it contains the log-prob of the padding tokens. Therefore, we need to remove them or mask them somehow. We decided to remove them, and thus, return a list of tensors :

```python
list_log_probs = []
for i in range(log_probs.shape[0]):
    list_log_probs.append(log_probs[i, :(inputs[i].shape[0])])
```

The output of the model is a list of tensor that is very similar to the input, but has the log-prob of a token instead of the token id.

Now, let's go back to the training loop. Regarding batching, a batch of our input x will be a list of length (batch_size) containing tensors of different sizes (sentences). Then we can construct y directly from that batch by creating a one-dimensional tensor of a size (total_num_words). Notice that the shape of y doesn't depend directly on the batch_size, but on the total number of words in the batch. Then the output of the model is a list of tensors of different sizes, similar to the input, but this one has log-prob instead of token id, i.e. the shape of a given tensor will be (len_sentence $\times$ vocab_size). By contacting all those tensors, we get a log-prob for each word in our batch (all sentences combined), which is a two-dimensional tensor of size (total_num_words $\times$ vocab_size). Finally, we pass that tensor to the loss function (negative log likelihood) along with the y. Here is the code from batching until computing the loss :

```python
batch = x_train[i:i+batch_size]
list_log_probs = model(batch)
log_probs = torch.cat(list_log_probs)
gold = torch.cat(batch)
loss = loss_function(log_probs, gold)
```

2.2.3 Dropout

During the model training, if the model has too many parameters and too few training samples, the trained model is prone to overfit. We can use dropout to alleviate it. During forward propagation, dropout lets the activation value of a certain neuron stop working with a certain probability p. This can make the model more generalized because it will not be too dependent on some local features.

But the problem is that we will randomly discard some neurons during training, but they cannot be randomly discarded during prediction, so this will cause unstable results. Then a compensation method is to multiply the weight of each neuron by p. For example, if the output of a neuron is x, then it has a probability of p participating during training, and the probability of $(1-p)$ is discarded. Then its output expectation is $px * 1 + (1-p) * 0 = px$. So when testing, multiplying the weight of this neuron by p can get the same expectation.

2.2.4 Variational dropout

In a normal dropout, we apply a different mask to each token (word embedding) in each sentence. Therefore, we have a number of masks equal to the total number of tokens in the batch, a mask per token. The idea of a variational dropout is to apply the same mask to all tokens of the sentence, i.e. we have one mask per sentence. We applied a variational dropout layer to the input embeddings of the LSTM layer, as well as its output embeddings. Here is the code to create a VariationalDropout module in pytorch.

```python
class VariationalDropout(nn.Module):
    def __init__(self, dropout: float):
        super().__init__()
        self.dropout = dropout

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        if not self.training or self.dropout <= 0. or self.dropout > 1.:
            return x
        batch_size, num_words, emb_dim = x.size()
        m = x.new_empty(batch_size, 1, emb_dim, requires_grad=False).bernoulli_(1 - self.dropout)
        x = (x*m) / (1 - self.dropout)
        return x
```

2.3 Evaluation – Perplexity

For evaluation, we use the metric of perplexity. We know that the language model is used to calculate the probability of a sentence or a word, but in practice, we will not use the original probability as a measure of the language model. That is when the perplexity comes in.

$$\begin{aligned} perplexity(S) &= p(w_1, w_2, w_3, ...w_N)^{-\frac{1}{N}} \\ &= (\frac{1}{p(w_1, w_2, w_3, ...w_N)})^{\frac{1}{N}} \\ &= (\prod_{i=1}^{N} \frac{1}{p(w_i|w_1, w_2, w_3, ...w_{i-1})})^{\frac{1}{N}} \end{aligned}$$

S represents the sentence, N is the length of the sentence, $p(w_i)$ is the probability of i_{th} word.

In addition, perplexity is the exponential form of cross entropy.

$$perplexity(S) = 2^(- \frac{1}{M} \sum_{i=1}^{M} log2p(yt|yt-1)$$

In our case, it takes a vector of log likelihood of size $[batchsize, 1]$, which signifies the probability of the correct next token. We then sum over the batch and do the average. The perplexity then can be obtained with the above function.

```python
class Perplexity:
    def __init__(self):
        self.log_probs = torch.tensor([]).to(device)

    def reset(self):
        self.log_probs = torch.tensor([]).to(device)

    def add_sentence(self, log_probs):
        # log_probs: vector of log probabilities of words in a sentence
        self.log_probs = torch.cat([self.log_probs, log_probs])

    def compute_perplexity(self):
        return 2**((-1/self.log_probs.size(0))*self.log_probs.sum())
```

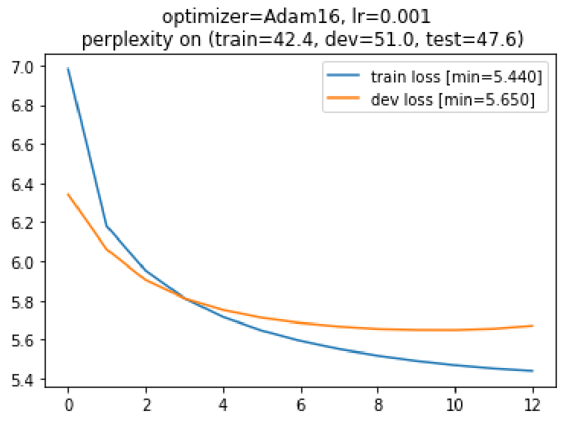
Better models will have higher log-likelihoods and lower perplexities. If the model is perfect and assigns a probability of 1 to the correct token, the log probability is zero and the perplexity is 1. On the other hand, the model does not know anything about the data. Thus, all tokens have the same probability. The perplexity will be $|N|$.

To evaluate our model, we will forward the data through the trained model to get the log probabilities. We then get the log probability of gold tokens using advanced indexing. Here is an example for the n-grams model:

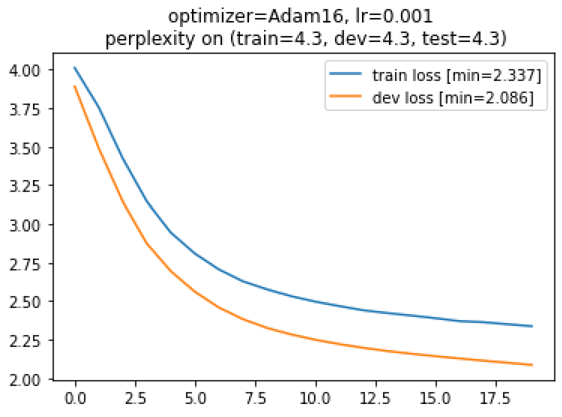
```python
batch = x[i:i + batch_size]
log_probs = model(batch)
log_probs_gold = log_probs[range(0,gold.shape[0]),gold]  # advanced indexing
perplexity.add_sentence(log_probs_gold)
```

3 Results of our experiments

We note that we used early stopping to avoid overfitting for the n-grams model. The LSTM Model was only trained for 20 epochs because of the long time it takes, although the loss was still going down.



(a) n-grams model with early stopping



(b) LSTM Model with Variational Dropout of 0.3

Figure 1: Learning curve of both models