AdvNLP Lab 6: Neural Language Models

Julie Weeds

March 1, 2024

1 Getting started

This week we are going to be looking at using the pytorch library to build a simple feedforward neural language model. This notebook is adapted from one of the pytorch tutorials and includes code by Robert Guthrie as well as my own.

https://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html#sphx-glr-beginner-nlp-word-embeddings-tutorial-py

```
# Standard pytorch imports
# Author: Robert Guthrie

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

Word Embeddings in Pytorch

Before we get to a worked example and some exercises, a few quick notes about how to use embeddings in Pytorch. First, we need to define an index for each word when using embeddings. These will be keys into a lookup table. That is, embeddings are stored as a $|V| \times D$ matrix, where D is the dimensionality of the embeddings, such that the word assigned index i has its embedding stored in the i'th row of the matrix. In all of my code, the mapping from words to indices is a dictionary named `word_to_ix`.

The module that allows you to use embeddings is `torch.nn.Embedding`, which takes two arguments: the vocabulary size, and the dimensionality of the embeddings.

To index into this table, you must use `torch.LongTensor` (since the indices are 64-bit integers, not floats).

```
#create a word index for 2 words
word_to_ix = {"hello": 0, "world": 1}
#create 5 dimensional embeddings for 2 words
embeds = nn.Embedding(2, 5) # 2 words in vocab, 5 dimensional embeddings
#identify the index into this embedding matrix for the word of interest - this is stored in a 1-d tensor
lookup_tensor = torch.tensor([word_to_ix["hello"]], dtype=torch.long)
#find the embedding of interest
hello_embed = embeds(lookup_tensor)
print(hello_embed)
```

N-gram Language Modelling

Recall that in an n-gram language model, given a sequence of words w, we want to compute

$$P(w_i|w_{i-1}, w_{i-2}, \dots, w_{i-n+1}) \tag{1}$$

where w_i is the ith word of the sequence.

In this example, we will compute the loss function on some training examples and update the parameters with backpropagation.

First, lets get some text, tokenise it and turn it into pairs of (context,target) where context is a list of the $n-1$ words immediately preceding target in the text.

```
from nltk import word_tokenize as tokenize

CONTEXT_SIZE = 2 #this is the amount of preceding context to consider
EMBEDDING_DIM = 10 #this is the dimension of the embeddings
# We will use Shakespeare Sonnet 2
test_sentence = ["__END","__START"]+tokenize("""When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a totter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
And see thy blood warm when thou feel'st it cold.""")+["__END"]

# build a list of tuples. Each tuple is ([ word_i-2, word_i-1 ], target word)
trigrams = [([test_sentence[i], test_sentence[i + 1]], test_sentence[i + 2])
            for i in range(len(test_sentence) - 2)]
# print the last 3, just so you can see what they look like
print(trigrams[-3:])
```

We need to find the set of words making up the vocabulary and create the `word_to_ix` index. We'll also make a reverse index `ix_to_word` at the same time so that we can look up a word associated with an index.

```
#find the vocabulary and create the index
vocab = set(test_sentence)
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for i, word in enumerate(vocab)}
print(word_to_ix)
```

Now we have our basic NGramLanguageModeler class. It inherits from the nn.Module class
https://pytorch.org/docs/stable/generated/torch.nn.Module.html

Essentially, the `__init__` method is used to define the neural network. We have a set of embeddings (vocab_size by embedding_dim) and then 2 linear layers. The first (or hidden) layer has 128 neurons each with context_size * embedding_dim inputs. The size of the second layer is equal to the vocab_size, where each neuron has 128 inputs (one from each neuron in the preceding layer). The value at each of the neurons in this output layer will tell us the probability of each word in the vocabulary as the next word in the sequence.

The `forward` method is used to run the network in forward mode i.e., give it some inputs and get some outputs. Activation functions are added to each layer - the hidden layer has a relu function applied to each neuron and the output layer outputs go through a softmax in order to create a probability distribution.

The `train` method iterates over the corpus for a certain number of epochs. The embeddings for the current context are selected and passed to the model's `forward` method. The log probability of the current target word according to the output is used to compute the loss (i.e., how likely is the target word given the current parameters) and this is then back-propagated through the network via stochastic gradient descent. It also prints the losses on each epoch - so you can see whether this is decreasing.

```
class NGramLanguageModeler(nn.Module):

    def __init__(self, vocab_size, embedding_dim, context_size):
        super(NGramLanguageModeler, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(context_size * embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((1, -1))
        out = F.relu(self.linear1(embeds))
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs


    def train(self,inputngrams,loss_function=nn.NLLLoss(),lr=0.001,epochs=10):
        optimizer=optim.SGD(self.parameters(),lr=lr)

        losses=[]
        for epoch in range(epochs):
            total_loss = 0
            for context, target in inputngrams:

                # Step 1. Prepare the inputs to be passed to the model (i.e, turn the words
                # into integer indices and wrap them in tensors)
                context_idxs = torch.tensor([word_to_ix[w] for w in context], dtype=torch.long)

                # Step 2. Recall that torch *accumulates* gradients. Before passing in a
                # new instance, you need to zero out the gradients from the old
                # instance
                self.zero_grad()

                # Step 3. Run the forward pass, getting log probabilities over next
                # words
                log_probs = self.forward(context_idxs)

                # Step 4. Compute your loss function. (Again, Torch wants the target
                # word wrapped in a tensor)
                loss = loss_function(log_probs, torch.tensor([word_to_ix[target]], dtype=torch.long))

                # Step 5. Do the backward pass and update the gradient
                loss.backward()
                optimizer.step()

                # Get the Python number from a 1-element Tensor by calling tensor.item()
                total_loss += loss.item()
            losses.append(total_loss)
        print(losses)


model = NGramLanguageModeler(len(vocab), EMBEDDING_DIM, CONTEXT_SIZE)
model.train(trigrams)
```

Now, we are going to some generation with the model. Add the following extra methods to the class which reflect the methods we had in our ngram language model in week 3. See if you can work out what each step is doing in each.

- `get_logprob()`
- `nextlikely()`
- `generate()`

```
def get_logprob(self,context,target):
#return the logprob of the target word given the context
    context_idxs = torch.tensor([word_to_ix[w] for w in context], dtype=torch.long)
    log_probs = self.forward(context_idxs)
    target_idx=torch.tensor(word_to_ix[target],dtype=torch.long)
    return log_probs.index_select(1,target_idx).item()

def nextlikely(self,context):
#sample the distribution of target words given the context
    context_idxs = torch.tensor([word_to_ix[w] for w in context], dtype=torch.long)
    log_probs = self.forward(context_idxs)
    probs=[math.exp(x) for x in log_probs.flatten().tolist()]
    t=random.choices(list(range(len(probs))),weights=probs,k=1)
    return ix_to_word[t[0]]

def generate(self,limit=20):
#generate a sequence of tokens according to the model
    tokens=["__END","__START"]
    while tokens[-1]!="__END" and len(tokens)<limit:
        current=self.nextlikely(tokens[-2:])
        tokens.append(current)
    return " ".join(tokens[2:-1])
```

2 Tasks

1. **Extend** your class so that it can be trained on a corpus. You can adapt some of the code from week 4 but you will need to think about the order in which things are initialised - the whole corpus will need to be read in so that the vocabulary can be determined BEFORE the neural network layers are initialised. **Train** your neural language model on part the training split of the corpus for the Microsoft Research Sentence Completion Challenge (see lab 4). Note that this will take a long time to run even if you only give it one file to process. Reducing the size of the vocabulary (in exercise 2) will improve the run time and the ability of the model to generalise. **Generate** some likely sequences.

2. **Modify** your model so that all words in the vocabulary with frequency less than a threshold (e.g, 5) are replaced by the "__UNK" token. **Generate** some likely sequences

3. Calculate the perplexity of the test corpus according to your NLM.

4. Try some different embedding sizes. Plot a graph of perplexity against embedding size.

5. Extend your model so that you can consider different amounts of context.