LSTM Training, Part of Speech Tagging

May 31, 2020

1 LSTM for Part-of-Speech Tagging

In this section, we will use an LSTM to predict part-of-speech tags for words. What exactly is part-of-speech tagging?

Part of speech tagging is the process of determining the *category* of a word from the words in its surrounding context. You can think of part of speech tagging as a way to go from words to their Mad Libs categories. Mad Libs are incomplete short stories that have many words replaced by blanks. Each blank has a specified word-category, such as `"noun"`, `"verb"`, `"adjective"`, and so on. One player asks another to fill in these blanks (prompted only by the word-category) until they have created a complete, silly story of their own. Here is an example of such categories:

```
Today, you'll be learning how to [verb]. It may be a [adjective] process, but I think it will be
If you want to take a break you should [verb] and treat yourself to some [plural noun].
```

... and a set of possible words that fall into those categories:

```
Today, you'll be learning how to code. It may be a challenging process, but I think it will be r
If you want to take a break you should stretch and treat yourself to some puppies.
```

1.0.1 Why Tag Speech?

Tagging parts of speech is often used to help disambiguate natural language phrases because it can be done quickly and with high accuracy. It can help answer: what subject is someone talking about? Tagging can be used for many NLP tasks like creating new sentences using a sequence of tags that make sense together, filling in a Mad Libs style game, and determining correct pronunciation during speech synthesis. It is also used in information retrieval, and for word disambiguation (ex. determining when someone says *right* like the direction versus *right* like "that's right!").

1.0.2 Preparing the Data

Now, we know that neural networks do not do well with words as input and so our first step will be to prepare our training data and map each word to a numerical value.

We start by creating a small set of training data, you can see that this is a few simple sentences broken down into a list of words and their corresponding word-tags. Note that the sentences are turned into lowercase words using `lower()` and then split into separate words using `split()`, which splits the sentence by whitespace characters.

Words to indices Then, from this training data, we create a dictionary that maps each unique word in our vocabulary to a numerical value; a unique index `idx`. We do the same for each word-tag, for example: a noun will be represented by the number `1`.

```
In [1]: # import resources
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        import matplotlib.pyplot as plt

        %matplotlib inline
```

```
In [2]: # training sentences and their corresponding word-tags
        training_data = [
            ("The cat ate the cheese".lower().split(), ["DET", "NN", "V", "DET", "NN"]),
            ("She read that book".lower().split(), ["NN", "V", "DET", "NN"]),
            ("The dog loves art".lower().split(), ["DET", "NN", "V", "NN"]),
            ("The elephant answers the phone".lower().split(), ["DET", "NN", "V", "DET", "NN"])
        ]

        # create a dictionary that maps words to indices
        word2idx = {}
        for sent, tags in training_data:
            for word in sent:
                if word not in word2idx:
                    word2idx[word] = len(word2idx)

        # create a dictionary that maps tags to indices
        tag2idx = {"DET": 0, "NN": 1, "V": 2}
```

Next, print out the created dictionary to see the words and their numerical values!

You should see every word in our training set and its index value. Note that the word "the" only appears once because our vocabulary only includes *unique* words.

```
In [3]: # print out the created dictionary
        print(word2idx)

{'the': 0, 'cat': 1, 'ate': 2, 'cheese': 3, 'she': 4, 'read': 5, 'that': 6, 'book': 7, 'dog': 8,
```

```
In [4]: import numpy as np

        # a helper function for converting a sequence of words to a Tensor of numerical values
        # will be used later in training
        def prepare_sequence(seq, to_idx):
            '''This function takes in a sequence of words and returns a
            corresponding Tensor of numerical values (indices for each word).'''
            idxs = [to_idx[w] for w in seq]
            idxs = np.array(idxs)
            return torch.from_numpy(idxs)
```

```
In [5]: # check out what prepare_sequence does for one of our training sentences:
        example_input = prepare_sequence("The dog answers the phone".lower().split(), word2idx)
        print(example_input)
```

```
tensor([  0,   8,  12,   0,  13])
```

1.1 Creating the Model

Our model will assume a few things: 1. Our input is broken down into a sequence of words, so a sentence will be [w1, w2, ...] 2. These words come from a larger list of words that we already know (a vocabulary) 3. We have a limited set of tags, `[NN, V, DET]`, which mean: a noun, a verb, and a determinant (words like "the" or "that"), respectively 4. We want to predict* a tag for each input word

* To do the prediction, we will pass an LSTM over a test sentence and apply a softmax function to the hidden state of the LSTM; the result is a vector of tag scores from which we can get the predicted tag for a word based on the *maximum* value in this distribution of tag scores.

Mathematically, we can represent any tag prediction $\hat{y}_i$ as:

$$\hat{y}_i = \text{argmax}_j \left(\log \text{Softmax}(Ah_i + b)\right)_j \tag{1}$$

Where A is a learned weight and b, a learned bias term, and the hidden state at timestep i is h_i.

1.1.1 Word embeddings

We know that an LSTM takes in an expected input size and hidden_dim, but sentences are rarely of a consistent size, so how can we define the input of our LSTM?

Well, at the very start of this net, we'll create an `Embedding` layer that takes in the size of our vocabulary and returns a vector of a specified size, `embedding_dim`, for each word in an input sequence of words. It's important that this be the first layer in this net. You can read more about this embedding layer in the PyTorch documentation.

Pictured below is the expected architecture for this tagger model.

```
In [6]: class LSTMTagger(nn.Module):

            def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
                ''' Initialize the layers of this model.'''
                super(LSTMTagger, self).__init__()

                self.hidden_dim = hidden_dim

                # embedding layer that turns words into a vector of a specified size
                self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
```

```
        # the LSTM takes embedded word vectors (of a specified size) as inputs
        # and outputs hidden states of size hidden_dim
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # the linear layer that maps the hidden state output dimension
        # to the number of tags we want as output, tagset_size (in this case this is 3 t
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)

        # initialize the hidden state (see code below)
        self.hidden = self.init_hidden()


    def init_hidden(self):
        ''' At the start of training, we need to initialize a hidden state;
           there will be none because the hidden state is formed based on perviously see
           So, this function defines a hidden state with all zeroes and of a specified s
        # The axes dimensions are (n_layers, batch_size, hidden_dim)
        return (torch.zeros(1, 1, self.hidden_dim),
                torch.zeros(1, 1, self.hidden_dim))

    def forward(self, sentence):
        ''' Define the feedforward behavior of the model.'''
        # create embedded word vectors for each word in a sentence
        embeds = self.word_embeddings(sentence)

        # get the output and hidden state by passing the lstm over our word embeddings
        # the lstm takes in our embeddings and hidden state
        lstm_out, self.hidden = self.lstm(
            embeds.view(len(sentence), 1, -1), self.hidden)

        # get the scores for the most likely tag for a word
        tag_outputs = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_outputs, dim=1)

        return tag_scores
```

1.2 Define how the model trains

To train the model, we have to instantiate it and define the loss and optimizers that we want to use.

First, we define the size of our word embeddings. The EMBEDDING_DIM defines the size of our word vectors for our simple vocabulary and training set; we will keep them small so we can see how the weights change as we train.

Note: the embedding dimension for a complex dataset will usually be much larger, around 64, 128, or 256 dimensional.

Loss and Optimization Since our LSTM outputs a series of tag scores with a softmax layer, we will use `NLLLoss`. In tandem with a softmax layer, NLL Loss creates the kind of cross entropy loss that we typically use for analyzing a distribution of class scores. We'll use standard gradient descent optimization, but you are encouraged to play around with other optimizers!

```
In [7]: # the embedding dimension defines the size of our word vectors
        # for our simple vocabulary and training set, we will keep these small
        EMBEDDING_DIM = 6
        HIDDEN_DIM = 6

        # instantiate our model
        model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word2idx), len(tag2idx))

        # define our loss and optimizer
        loss_function = nn.NLLLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.1)
```

Just to check that our model has learned something, let's first look at the scores for a sample test sentence *before* our model is trained. Note that the test sentence *must* be made of words from our vocabulary otherwise its words cannot be turned into indices.

The scores should be Tensors of length 3 (for each of our tags) and there should be scores for each word in the input sentence.

For the test sentence, "The cheese loves the elephant", we know that this has the tags (DET, NN, V, DET, NN) or `[0, 1, 2, 0, 1]`, but our network does not yet know this. In fact, in this case, our model starts out with a hidden state of all zeroes and so all the scores and the predicted tags should be low, random, and about what you'd expect for a network that is not yet trained!

```
In [8]: test_sentence = "The cheese loves the elephant".lower().split()

        # see what the scores are before training
        # element [i,j] of the output is the *score* for tag j for word i.
        # to check the initial accuracy of our model, we don't need to train, so we use model.ev
        inputs = prepare_sequence(test_sentence, word2idx)
        inputs = inputs
        tag_scores = model(inputs)
        print(tag_scores)

        # tag_scores outputs a vector of tag scores for each word in an inpit sentence
        # to get the most likely tag index, we grab the index with the maximum score!
        # recall that these numbers correspond to tag2idx = {"DET": 0, "NN": 1, "V": 2}
        _, predicted_tags = torch.max(tag_scores, 1)
        print('\n')
        print('Predicted tags: \n',predicted_tags)
```

```
tensor([[-1.4706, -0.7638, -1.1897],
        [-1.5189, -0.7543, -1.1690],
        [-1.4320, -0.7917, -1.1773],
        [-1.5067, -0.7402, -1.1995],
        [-1.5904, -0.6857, -1.2296]])
```

```
Predicted tags:
 tensor([ 1,  1,  1,  1,  1])
```

1.3 Train the Model

Loop through all our training data for multiple epochs (again we are using a small epoch value for this simple training data). This loop:

1. Prepares our model for training by zero-ing the gradients
2. Initializes the hidden state of our LSTM
3. Prepares our data for training
4. Runs a forward pass on our inputs to get tag_scores
5. Calculates the loss between tag_scores and the true tag
6. Updates the weights of our model using backpropagation

In this example, we are printing out the average epoch loss, every 20 epochs; you should see it decrease over time.

```
In [9]: # normally these epochs take a lot longer
        # but with our toy data (only 3 sentences), we can do many epochs in a short time
        n_epochs = 300

        for epoch in range(n_epochs):

            epoch_loss = 0.0

            # get all sentences and corresponding tags in the training data
            for sentence, tags in training_data:

                # zero the gradients
                model.zero_grad()

                # zero the hidden state of the LSTM, this detaches it from its history
                model.hidden = model.init_hidden()

                # prepare the inputs for processing by out network,
                # turn all sentences and targets into Tensors of numerical indices
                sentence_in = prepare_sequence(sentence, word2idx)
                targets = prepare_sequence(tags, tag2idx)

                # forward pass to get tag scores
                tag_scores = model(sentence_in)

                # compute the loss, and gradients
            loss = loss_function(tag_scores, targets)
            epoch_loss += loss.item()
            loss.backward()

            # update the model parameters with optimizer.step()
            optimizer.step()

        # print out avg loss per 20 epochs
        if(epoch%20 == 19):
            print("Epoch: %d, loss: %1.5f" % (epoch+1, epoch_loss/len(training_data)))
```

```
Epoch: 20, loss: 0.92635
Epoch: 40, loss: 0.64539
Epoch: 60, loss: 0.38730
Epoch: 80, loss: 0.21467
Epoch: 100, loss: 0.12886
Epoch: 120, loss: 0.08614
Epoch: 140, loss: 0.06228
Epoch: 160, loss: 0.04768
Epoch: 180, loss: 0.03809
Epoch: 200, loss: 0.03143
Epoch: 220, loss: 0.02658
Epoch: 240, loss: 0.02292
Epoch: 260, loss: 0.02009
Epoch: 280, loss: 0.01784
Epoch: 300, loss: 0.01601
```

1.4 Testing

See how your model performs *after* training. Compare this output with the scores from before training, above.

Again, for the test sentence, "The cheese loves the elephant", we know that this has the tags (DET, NN, V, DET, NN) or `[0, 1, 2, 0, 1]`. Let's see if our model has learned to find these tags!

```
In [10]: test_sentence = "The cheese loves the elephant".lower().split()

         # see what the scores are after training
         inputs = prepare_sequence(test_sentence, word2idx)
         inputs = inputs
         tag_scores = model(inputs)
         print(tag_scores)

         # print the most likely tag index, by grabbing the index with the maximum score!
         # recall that these numbers correspond to tag2idx = {"DET": 0, "NN": 1, "V": 2}
         _, predicted_tags = torch.max(tag_scores, 1)
         print('\n')
         print('Predicted tags: \n',predicted_tags)
```

```
tensor([[-0.0124, -5.7735, -4.6888],
        [-6.9034, -0.0021, -6.7835],
        [-5.6254, -2.8380, -0.0642],
        [-0.0227, -5.8839, -3.9290],
        [-6.2161, -0.0089, -4.9848]])


Predicted tags:
 tensor([ 0,  1,  2,  0,  1])
```

1.5 Great job!

To improve this model, see if you can add sentences to this model and create a more robust speech tagger. Try to initialize the hidden state in a different way or play around with the optimizers and see if you can decrease model loss even faster.

```
In [ ]:
```