4A_Natural_Language_Processing

May 12, 2021

1 Introduction to Natural Language Processing (NLP) in PyTorch

1.0.1 Word Embeddings

Word embeddings, or word vectors, provide a way of mapping words from a vocabulary into a low-dimensional space, where words with similar meanings are close together. Let's play around with a set of pre-trained word vectors, to get used to their properties. There exist many sets of pretrained word embeddings; here, we use ConceptNet Numberbatch, which provides a relatively small download in an easy-to-work-with format (h5).

[ ]:
```
# Download word vectors
from urllib.request import urlretrieve
import os
if not os.path.isfile('datasets/mini.h5'):
    print("Downloading Conceptnet Numberbatch word embeddings...")
    conceptnet_url = 'http://conceptnet.s3.amazonaws.com/precomputed-data/2016/
↪numberbatch/17.06/mini.h5'
    urlretrieve(conceptnet_url, 'datasets/mini.h5')
```

To read an h5 file, we'll need to use the `h5py` package. If you followed the PyTorch installation instructions in 1A, you should have it downloaded already. Otherwise, you can install it with

```
# If you environment isn't currently active, activate it:
# conda activate pytorch

pip install h5py
```

You may need to re-open this notebook for the installation to take effect.

Below, we use the package to open the `mini.h5` file we just downloaded. We extract from the file a list of utf-8-encoded words, as well as their 300-d vectors.

[ ]:
```
# Load the file and pull out words and embeddings
import h5py

with h5py.File('datasets/mini.h5', 'r') as f:
    all_words = [word.decode('utf-8') for word in f['mat']['axis1'][:]]
    all_embeddings = f['mat']['block0_values'][:]
```

```
print("all_words dimensions: {}".format(len(all_words)))
print("all_embeddings dimensions: {}".format(all_embeddings.shape))

print("Random example word: {}".format(all_words[1337]))
```

Now, `all_words` is a list of V strings (what we call our *vocabulary*), and `all_embeddings` is a $V \times 300$ matrix. The strings are of the form `/c/language_code/word`—for example, `/c/en/cat` and `/c/es/gato`.

We are interested only in the English words. We use Python list comprehensions to pull out the indices of the English words, then extract just the English words (stripping the six-character `/c/en/` prefix) and their embeddings.

[ ]:
```
# Restrict our vocabulary to just the English words
english_words = [word[6:] for word in all_words if word.startswith('/c/en/')]
english_word_indices = [i for i, word in enumerate(all_words) if word.
 ↪startswith('/c/en/')]
english_embeddings = all_embeddings[english_word_indices]

print("Number of English words in all_words: {0}".format(len(english_words)))
print("english_embeddings dimensions: {0}".format(english_embeddings.shape))

print(english_words[1337])
```

The magnitude of a word vector is less important than its direction; the magnitude can be thought of as representing frequency of use, independent of the semantics of the word. Here, we will be interested in semantics, so we *normalize* our vectors, dividing each by its length. The result is that all of our word vectors are length 1, and as such, lie on a unit circle. The dot product of two vectors is proportional to the cosine of the angle between them, and provides a measure of similarity (the bigger the cosine, the smaller the angle).

[ ]:
```
import numpy as np

norms = np.linalg.norm(english_embeddings, axis=1)
normalized_embeddings = english_embeddings.astype('float32') / norms.
 ↪astype('float32').reshape([-1, 1])
```

We want to look up words easily, so we create a dictionary that maps us from a word to its index in the word embeddings matrix.

[ ]:
```
index = {word: i for i, word in enumerate(english_words)}
```

Now we are ready to measure the similarity between pairs of words. We use numpy to take dot products.

[ ]:
```
def similarity_score(w1, w2):
    score = np.dot(normalized_embeddings[index[w1], :],␣
 ↪normalized_embeddings[index[w2], :])
```

```
    return score

# A word is as similar with itself as possible:
print('cat\tcat\t', similarity_score('cat', 'cat'))

# Closely related words still get high scores:
print('cat\tfeline\t', similarity_score('cat', 'feline'))
print('cat\tdog\t', similarity_score('cat', 'dog'))

# Unrelated words, not so much
print('cat\tmoo\t', similarity_score('cat', 'moo'))
print('cat\tfreeze\t', similarity_score('cat', 'freeze'))

# Antonyms are still considered related, sometimes more so than synonyms
print('antonym\topposite\t', similarity_score('antonym', 'opposite'))
print('antonym\tsynonym\t', similarity_score('antonym', 'synonym'))
```

We can also find, for instance, the most similar words to a given word.

[ ]:
```
def closest_to_vector(v, n):
    all_scores = np.dot(normalized_embeddings, v)
    best_words = list(map(lambda i: english_words[i], reversed(np.
 ↪argsort(all_scores))))
    return best_words[:n]

def most_similar(w, n):
    return closest_to_vector(normalized_embeddings[index[w], :], n)
```

[ ]:
```
print(most_similar('cat', 10))
print(most_similar('dog', 10))
print(most_similar('duke', 10))
```

We can also use `closest_to_vector` to find words "nearby" vectors that we create ourselves. This allows us to solve analogies. For example, in order to solve the analogy "man : brother :: woman : ?", we can compute a new vector `brother - man + woman`: the meaning of brother, minus the meaning of man, plus the meaning of woman. We can then ask which words are closest, in the embedding space, to that new vector.

[ ]:
```
def solve_analogy(a1, b1, a2):
    b2 = normalized_embeddings[index[b1], :] - normalized_embeddings[index[a1],␣
 ↪:] + normalized_embeddings[index[a2], :]
    return closest_to_vector(b2, 1)

print(solve_analogy("man", "brother", "woman"))
print(solve_analogy("man", "husband", "woman"))
print(solve_analogy("spain", "madrid", "france"))
```

These three results are quite good, but in general, the results of these analogies can be disappointing.

Try experimenting with other analogies, and see if you can think of ways to get around the problems you notice (i.e., modifications to the `solve_analogy()` algorithm).

1.0.2 Using word embeddings in deep models

Word embeddings are fun to play around with, but their primary use is that they allow us to think of words as existing in a continuous, Euclidean space; we can then use an existing arsenal of techniques for machine learning with continuous numerical data (like logistic regression or neural networks) to process text. Let's take a look at an especially simple version of this. We'll perform *sentiment analysis* on a set of movie reviews: in particular, we will attempt to classify a movie review as positive or negative based on its text.

We will use a Simple Word Embedding Model (SWEM, Shen et al. 2018) to do so. We will represent a review as the *mean* of the embeddings of the words in the review. Then we'll train a two-layer MLP (a neural network) to classify the review as positive or negative. As you might guess, using just the mean of the embeddings discards a lot of the information in a sentences, but for tasks like sentiment analysis, it can be surprisingly effective.

If you don't have it already, download the `movie-simple.txt` file. Each line of that file contains

1. the numeral 0 (for negative) or the numeral 1 (for positive), followed by
2. a tab (the whitespace character), and then
3. the review itself.

Let's first read the data file, parsing each line into an input representation and its corresponding label. Again, since we're using SWEM, we're going to take the mean of the word embeddings for all the words as our input.

[ ]:
```python
import string
remove_punct=str.maketrans('','',string.punctuation)

# This function converts a line of our data file into
# a tuple (x, y), where x is 300-dimensional representation
# of the words in a review, and y is its label.
def convert_line_to_example(line):
    # Pull out the first character: that's our label (0 or 1)
    y = int(line[0])

    # Split the line into words using Python's split() function
    words = line[2:].translate(remove_punct).lower().split()

    # Look up the embeddings of each word, ignoring words not
    # in our pretrained vocabulary.
    embeddings = [normalized_embeddings[index[w]] for w in words
                  if w in index]

    # Take the mean of the embeddings
    x = np.mean(np.vstack(embeddings), axis=0)
    return x, y
```

```
# Apply the function to each line in the file.
xs = []
ys = []
with open("datasets/movie-simple.txt", "r", encoding='utf-8', errors='ignore')␣
↪as f:
    for l in f.readlines():
        x, y = convert_line_to_example(l)
        xs.append(x)
        ys.append(y)

# Concatenate all examples into a numpy array
xs = np.vstack(xs)
ys = np.vstack(ys)
```

[ ]:
```
print("Shape of inputs: {}".format(xs.shape))
print("Shape of labels: {}".format(ys.shape))

num_examples = xs.shape[0]
```

Notice that with this set-up, our input words have been converted to vectors as part of our pre-processing. This essentially locks our word embeddings in place throughout training, as opposed to learning the word embeddings. Learning word embeddings, either from scratch or fine-tuned from some pre-trained initialization, is often desirable, as it specializes them for the specific task. However, because our data set is relatively small and our computation budget for this demo, we're going to forgo learning the word embeddings for this model. We'll revist this in a bit.

Now that we've parsed the data, let's save 20% of the data (rounded to a whole number) for testing, using the rest for training. The file we loaded had all the negative reviews first, followed by all the positive reviews, so we need to shuffle it before we split it into the train and test splits. We'll then convert the data into PyTorch Tensors so we can feed them into our model.

[ ]:
```
print("First 20 labels before shuffling: {0}".format(ys[:20, 0]))

shuffle_idx = np.random.permutation(num_examples)
xs = xs[shuffle_idx, :]
ys = ys[shuffle_idx, :]

print("First 20 labels after shuffling: {0}".format(ys[:20, 0]))
```

[ ]:
```
import torch

num_train = 4*num_examples // 5

x_train = torch.tensor(xs[:num_train])
y_train = torch.tensor(ys[:num_train], dtype=torch.float32)

x_test = torch.tensor(xs[num_train:])
```

```
y_test = torch.tensor(ys[num_train:], dtype=torch.float32)
```

We could format each batch individually as we feed it into the model, but to make it easier on ourselves, let's create a TensorDataset and DataLoader as we've used in the past for MNIST.

```
[ ]: reviews_train = torch.utils.data.TensorDataset(x_train, y_train)
     reviews_test = torch.utils.data.TensorDataset(x_test, y_test)

     train_loader = torch.utils.data.DataLoader(reviews_train, batch_size=100,␣
      ↪shuffle=True)
     test_loader = torch.utils.data.DataLoader(reviews_test, batch_size=100,␣
      ↪shuffle=False)
```

Time to build our model in PyTorch.

```
[ ]: import torch.nn as nn
     import torch.nn.functional as F
```

First we build the model, organized as a `nn.Module`. We could make the number of outputs for our MLP the number of classes for this dataset (i.e. 2). However, since we only have two output classes here ("positive" vs "negative"), we can instead produce a single output value, calling everything greater than 0 "postive" and everything less than 0 "negative". If we pass this output through a sigmoid operation, then values are mapped to $[0, 1]$, with 0.5 being the classification threshold.

```
[ ]: class SWEM(nn.Module):
         def __init__(self):
             super().__init__()
             self.fc1 = nn.Linear(300, 64)
             self.fc2 = nn.Linear(64, 1)

         def forward(self, x):
             x = self.fc1(x)
             x = F.relu(x)
             x = self.fc2(x)
             return x
```

To train the model, we instantiate the model. Notice that since we are only doing binary classification, we use the binary cross-entropy (BCE) loss instead of the cross-entropy loss we've seen before. We use the "with logits" version for numerical stability.

```
[ ]: ## Training
     # Instantiate model
     model = SWEM()

     # Binary cross-entropy (BCE) Loss and Adam Optimizer
     criterion = nn.BCEWithLogitsLoss()
     optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
# Iterate through train set minibatchs
for epoch in range(250):
    correct = 0
    num_examples = 0
    for inputs, labels in train_loader:
        # Zero out the gradients
        optimizer.zero_grad()

        # Forward pass
        y = model(inputs)
        loss = criterion(y, labels)

        # Backward pass
        loss.backward()
        optimizer.step()

        predictions = torch.round(torch.sigmoid(y))
        correct += torch.sum((predictions == labels).float())
        num_examples += len(inputs)

    # Print training progress
    if epoch % 25 == 0:
        acc = correct/num_examples
        print("Epoch: {0} \t Train Loss: {1} \t Train Acc: {2}".format(epoch,␣
 ↪loss, acc))

## Testing
correct = 0
num_test = 0

with torch.no_grad():
    # Iterate through test set minibatchs
    for inputs, labels in test_loader:
        # Forward pass
        y = model(inputs)

        predictions = torch.round(torch.sigmoid(y))
        correct += torch.sum((predictions == labels).float())
        num_test += len(inputs)

print('Test accuracy: {}'.format(correct/num_test))
```

We can now examine what our model has learned, seeing how it responds to word vectors for different words:

[ ]:
```
# Check some words
words_to_test = ["exciting", "hated", "boring", "loved"]
```

```

for word in words_to_test:
    x = torch.tensor(normalized_embeddings[index[word]].reshape(1, 300))
    print("Sentiment of the word '{0}': {1}".format(word, torch.
 ↪sigmoid(model(x))))
```

Try some words of your own! You can also try changing up the model and re-training it to see how the results change. Can you modify the architecture to get better performance? Alternatively, can you simplify the model without sacrificing too much accuracy? What if you try to classify the mean embeddings directly?

1.0.3 Learning Word Embeddings

In the previous example, we used pre-trained word embeddings, but didn't learn them. The word embeddings were part of preprocessing and remained unchanged throughout training. If we have enough data though, we might prefer to learn the word embeddings along with our model. Pre-trained word embeddings are typically trained on large corpora with unsupervised objectives, and are often non-specific. If we have enough data, we may prefer to learn the word embeddings, either from scratch or with fine-tuning, as making them specific to the task may improve performance.

How do we learn word embeddings? To do so, we need to make them a part of our model, rather than as part of loading the data. In PyTorch, the preferred way to do so is with the `nn.Embedding`. Like the other `nn` layers we've seen (e.g. `nn.Linear`), `nn.Embedding` must be instantiated first. There are two required arguments for instantiation are the number of embeddings (i.e. the vocabulary size V) and the dimension of word embeddings (300, in our previous example).

[ ]:
```
VOCAB_SIZE = 5000
EMBED_DIM = 300

embedding = nn.Embedding(VOCAB_SIZE, EMBED_DIM)
```

Under the hood, this creates a word embedding matrix that is 5000×300.

[ ]:
```
embedding.weight.size()
```

Notice that this matrix is basically a 300 dimensional word embedding for each of the 5000 words, stacked on top of each other. Looking up a word embedding in this embedding matrix is simply selecting a specific row of this matrix, corresponding to the word.

When word embeddings are learned, `nn.Embedding` look-up is often one of the first operations in a model module. For example, if we were to learn the word embeddings for our previous SWEM model, the model might instead look like this:

[ ]:
```
class SWEMWithEmbeddings(nn.Module):
    def __init__(self, vocab_size, embedding_size, hidden_dim, num_outputs):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_size)
        self.fc1 = nn.Linear(embedding_size, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, num_outputs)

    def forward(self, x):
        x = self.embedding(x)
        x = torch.mean(x, dim=0)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return x
```

Here we've abstracted the size of the various layers of the model as constructor arguments, so we need to specify those hyperparameters at initialization.

[ ]:
```
model = SWEMWithEmbeddings(
    vocab_size = 5000,
    embedding_size = 300,
    hidden_dim = 64,
    num_outputs = 1,
)
print(model)
```

Note that by making embedding part of our model, the expected input to the `forward()` function is now the word tokens for the input sentence, so we would have to modify our data input pipeline as well. We'll see how this might be done in the next notebook (4B).

1.0.4 Recurrent Neural Networks (RNNs)

In the context of deep learning, sequential data is commonly modeled with Recurrent Neural Networks (RNNs). As natural language can be viewed as a sequence of words, RNNs are commonly used for NLP. As with the fully connected and convolutional networks we've seen before, RNNs use combinations of linear and nonlinear transformations to project the input into higher level representations, and these representations can be stacked with additional layers.

Sentences as sequences The key difference between sequential models and the previous models we've seen is the presence of a "time" dimension: words in a sentence (or paragraph, document) have an ordering to them that convey meaning:

In the example sequence above, the word "Recurrent" is the $t = 1$ word, which we denote w_1; similarly, "neural" is w_2, and so on. As the preceding sections have hopefully impressed upon you, it is often more advantageous to model words as embedding vectors $x_1, ..., x_T$, rather than one-hot vectors (which tokens $w_1, ...w_T$ correspond to), so our first step is often to do an embedding table look-up for each input word. Let's assume 300-dimensional word embeddings and, for simplicity, a minibatch of size 1.

[ ]:
```
mb = 1
x_dim = 300
sentence = ["recurrent", "neural", "networks", "are", "great"]
```

```python
xs = []
for word in sentence:
    xs.append(torch.tensor(normalized_embeddings[index[word]]).view(1, x_dim))

xs = torch.stack(xs, dim=0)
print("xs shape: {}".format(xs.shape))
```

Notice that we have formatted our inputs as (words $\times$ minibatch $\times$ embedding dimension). This is the preferred input ordering for PyTorch RNNs.

Let's say we want to process this example. In our previous sentiment analysis example, we just took the average embedding across time, treating the input as a "bag-of-words." For simple problems, this can work surprisingly well, but as you might imagine, the ordering of words in a sentence is often important, and sometimes, we'd like to be able to model this temporal meaning as well. Enter RNNs.

Review: Fully connected layer Before we introduce the RNN, let's first again revist the fully connected layer that we used in our logistic regression and multilayer perceptron examples, with a few changes in notation:

$$h = f(xW + b)$$

Instead of calling the result of the fully connected layer y, we're going to call it h, for hidden state. The variable y is usually reserved for the final layer of the neural network; since logistic regression was a single layer, using y was fine. However, if we assume there is more than one layer, it is more common to refer to the intermediate representation as h. Note that we also use $f()$ to denote a nonlinear activation function. In the past, we've seen $f()$ as a ReLU, but this could also be a $\sigma()$ or $\tanh()$ nonlinearity. Visualized:

The key thing to notice here is that we project the input x with a linear transformation (with W and b), and then apply a nonlinearity to the output, giving us h. During training, our goal is to learn W and b.

A basic RNN Unlike the previous examples we've seen using fully connected layers, sequential data have multiple inputs $x_1, ..., x_T$, instead of a single x. We need to adapt our models accordingly for an RNN. While there are several variations, a common basic formulation for an RNN is the Elman RNN, which is as follows*:

$$h_t = \tanh((x_t W_x + b_x) + (h_{t-1} W_h + b_h)) \tag{1}$$

where $\tanh()$ is the hyperbolic tangent, a nonlinear activation function. RNNs process words one at a time in sequence (x_t), producing a hidden state h_t at every time step. The first half of the above equation should look familiar; as with the fully connected layer, we are linearly transforming each input x_t, and then applying a nonlinearity. Notice that we apply the same linear transformation (W_x, b_x) at every time step. The difference is that we also apply a separate linear transform (W_h, b_h) to the previous hidden state h_{t-1} and add it to our projected input. This feedback is called a *recurrent* connection.

These directed cycles in the RNN architecture gives them the ability to model temporal dynamics, making them particularly suited for modeling sequences (e.g. text). We can visualize an RNN layer as follows:

We can unroll an RNN through time, making the sequential nature of them more obvious:

You can think of these recurrent connections as allowing the model to consider previous hidden states of a sequence when calculating the hidden state for the current input.

*Note: We don't actually need two separate biases b_x and b_h, as you can combine both biases into a single learnable parameter b. However, writing it separately helps make it clear that we're performing a linear transformation on both x_t and h_{t-1}. Speaking of combining variables, we can also express the above operation by concatenating x_t and h_{t-1} into a single vector z_t, and then performing a single matrix multiply $z_t W_z + b$, where W_z is essentially W_x and W_h concatenated. Indeed this is how many "official" RNNs modules are implemented, as the reduction in the number of separate matrix multiply operations makes it computationally more effecient. These are implementation details though.

RNNs in PyTorch How would we implement an RNN in PyTorch? There are quite a few ways, but let's build the Elman RNN from scratch first, using the input sequence "recurrent neural networks are great".

[ ]:
```
# As always, import PyTorch first
import numpy as np
import torch
```

In an RNN, we project both the input x_t and the previous hidden state h_{t-1} to some hidden dimension, which we're going to choose to be 128. To perform these operations, we're going to define some variables we're going to learn.

[ ]:
```
h_dim = 128

# For projecting the input
Wx = torch.randn(x_dim, h_dim)/np.sqrt(x_dim)
Wx.requires_grad_()
bx = torch.zeros(h_dim, requires_grad=True)

# For projecting the previous state
Wh = torch.randn(h_dim, h_dim)/np.sqrt(h_dim)
Wh.requires_grad_()
bh = torch.zeros(h_dim, requires_grad=True)

print(Wx.shape, bx.shape, Wh.shape, bh.shape)
```

For convenience, we define a function for one time step of the RNN. This function take the current input x_t and previous hidden state h_{t-1}, performs the linear transformations xW_x+b_x and hW_h+b_h, and then a hyperbolic tangent nonlinearity.

```
[ ]: def RNN_step(x, h):
         h_next = torch.tanh((torch.matmul(x, Wx) + bx) + (torch.matmul(h, Wh) + bh))

         return h_next
```

Each step of our RNN is going to require feeding in an input (i.e. the word representation) and the previous hidden state (the summary of preceding sequence). Note that at the beginning of a sentence, we don't have a previous hidden state, so we initialize it to some value, for example all zeros:

```
[ ]: # Word embedding for first word
     x1 = xs[0, :, :]

     # Initialize hidden state to 0
     h0 = torch.zeros([mb, h_dim])
```

To take one time step of the RNN, we call the function we wrote, passing in x_1 and h_0. In this case,

```
[ ]: # Forward pass of one RNN step for time step t=1
     h1 = RNN_step(x1, h0)

     print("Hidden state h1 dimensions: {0}".format(h1.shape))
```

We can call the `RNN_step` function again to get the next time step output from our RNN.

```
[ ]: # Word embedding for second word
     x2 = xs[1, :, :]

     # Forward pass of one RNN step for time step t=2
     h2 = RNN_step(x2, h1)

     print("Hidden state h2 dimensions: {0}".format(h2.shape))
```

We can continue unrolling the RNN as far as we need to. For each step, we feed in the current input (x_t) and previous hidden state (h_{t-1}) to get a new output.

Using `torch.nn` In practice, much like fully connected and convolutional layers, we typically don't implement RNNs from scratch as above, instead relying on higher level APIs. PyTorch has RNNs implemented in the `torch.nn` library.

```
[ ]: import torch.nn

     rnn = nn.RNN(x_dim, h_dim)
```

```python
print("RNN parameter shapes: {}".format([p.shape for p in rnn.parameters()]))
```

Note that the RNN created by `torch.nn` produces parameters of the same dimensions as our from scratch example above.

To perform a forward pass with an RNN, we pass the entire input sequence to the `forward()` function, which returns the hidden states at every time step (`hs`) and the final hidden state (`h_T`).

[ ]:
```python
hs, h_T = rnn(xs)

print("Hidden states shape: {}".format(hs.shape))
print("Final hidden state shape: {}".format(h_T.shape))
```

What do we do with these hidden states? It depends on the model and task. Just like multilayer perceptrons and convolutional neural networks, RNNs can be stacked in multiple layers as well. In this case, the outputs $h_1, ..., h_T$ are the sequential inputs to the next layer. If the RNN layer is the final layer, h_T or the mean/max of $h_1, ..., h_T$ can be used as a summary encoding of the data sequence. What is being predicted can also have an impact on what the RNN outputs are ultimately used for.

Gated RNNs While the RNNs we've just explored can successfully model simple sequential data, they tend to struggle with longer sequences, with vanishing gradients an especially big problem. A number of RNN variants have been proposed over the years to mitigate this issue and have been shown empirically to be more effective. In particular, Long Short-Term Memory (LSTM) and the Gated Recurrent Unit (GRU) have seen wide use recently in deep learning. We're not going to go into detail here about what structural differences they have from vanilla RNNs; a fantastic summary can be found here. Note that "RNN" as a name is somewhat overloaded: it can refer to both the basic recurrent model we went over previously, or recurrent models in general (including LSTMs and GRUs).

LSTMs and GRUs layers can be created in much the same way as basic RNN layers. Again, rather than implementing it yourself, it's recommended to use the `torch.nn` implementations, although we highly encourage that you peek at the source code so you understand what's going on under the hood.

[ ]:
```python
lstm = nn.LSTM(x_dim, h_dim)
print("LSTM parameters: {}".format([p.shape for p in lstm.parameters()]))

gru = nn.GRU(x_dim, h_dim)
print("GRU parameters: {}".format([p.shape for p in gru.parameters()]))
```

1.0.5 Torchtext

Much like PyTorch has Torchvision for computer vision, PyTorch also has Torchtext for natural language processing. As with Torchvision, Torchtext has a number of popular NLP benchmark datasets, across a wide range of tasks (e.g. sentiment analysis, language modeling, machine translation). It also has a few pre-trained word embeddings available as well, including the popular

Global Vectors for Word Representation (GloVe). If you need to load your own dataset, Torchtext has a number of useful containers that can make the data pipeline easier.

You'll need to install TorchText to use it:

```
# If you environment isn't currently active, activate it:
# conda activate pytorch

pip install torchtext
```

1.0.6 Other materials:

Natural Language Processing can be several full courses on its own at most universities, both with or without neural networks. Here are some additional reads:

- Fantastic introduction to LSTMs and GRUs
- Popular blog post on the effectiveness of RNNs