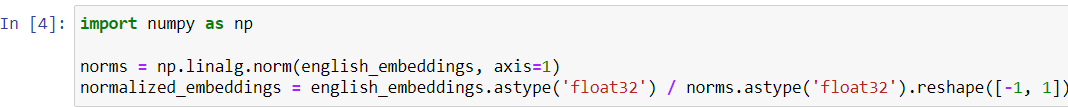
**Introduction to Natural Language Processing (NLP) in PyTorch**

**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. This module has a set of pre-trained word vectors. There exist many sets of pretrained word embeddings: Here we will utilize ConceptNet NumberBatch.
* A computer screen shot of a computer code

  Description automatically generated
* To read an h5 file, we’ll need to use the h5py package
  + Below, we use the package to open the mini.h5 file. We extract from the file a list of utf-8 encoded words, as well as their 300-d vectors
* A screen shot of a computer code

  Description automatically generated
* Now, “all\_words” is a list of “V” strings (what we call our vocabulary), and “all\_embeddings” is a “V x 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 6-character “/c/en/” prefix) and their embeddings
* A screen shot of a computer screen

  Description automatically generated
* 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).
* 
  + We want to look up words easily, so we create ea dictionary that maps us from a word to its index in the word embeddings matrix.
* 
  + Now we are ready to measure the similarity between pairs of words. We use numpy to take the dot product
* A screenshot of a computer code

  Description automatically generated
  + We can also find, for instance, the most similar words to a given word.
* A screenshot of a computer

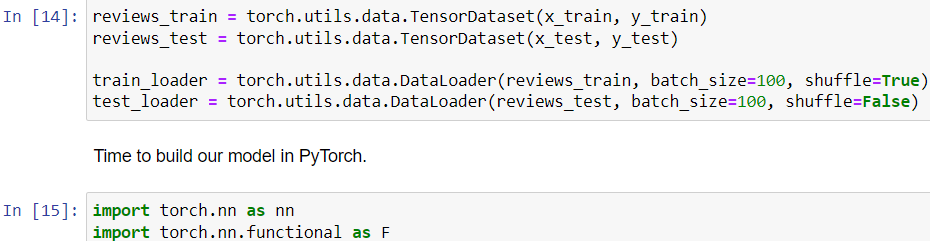
  Description automatically generated
  + We can also use “closet\_to\_vector” to find words “nearby” vectors that we create ourselves. This allows us to solve analogies. For example, in order to solve the analogy “man : brother :: woman : ?”, we can compute a new vector “brother – man + woman:” the meaning of brother, minus the meaning of man, plus the meaning of woman. We can then ask which words are closet, in the embedding space, to that new vector.
* A close-up of a computer screen

  Description automatically generated
  + 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).

**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 revies: 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 embedding 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 task like sentiment analysis, it can be surprisingly effective.
    - 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
      2. By 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.
* **A screenshot of a computer program

  Description automatically generated**
  + Notice that with this set-up, our input words have been converted to vectors as part of our preprocessing. 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 revisit 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
* **A screenshot of a computer code

  Description automatically generated**
  + We could format each batch individually as we feed it into the model, but to make it easier on ourselves, let’s create a Tensor Dataset and Data Loader as we’ve used in the past for MNIST.
* ****
  + 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 “positive” 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
* **A screenshot of a computer program

  Description automatically generated**
  + 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.
* **A screenshot of a computer program

  Description automatically generated**
* **A number on a white background

  Description automatically generated**
  + We can now examine what our model has learned, seeing how it responds to word vectors for different words:
* **A screen shot of a computer code

  Description automatically generated**
  + You can also try changing up the model and re-training it to see how the results change. Better performance | simplicity without reduction in accuracy | classification of mean embeddings directly

**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).
* **A screenshot of a computer program

  Description automatically generated**
  + 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:
* **A computer screen shot of a program

  Description automatically generated**
  + 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.
* **A screenshot of a computer program

  Description automatically generated**
  + 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

**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 he 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:
    - A close-up of a word

      Description automatically generated
  + In the example sequence above, the word “Recurrent” is the “t=1” word, which we denote “w1”; similarly, “neural” is “w2”, and so on. As the preceding sections have hopefully impressed upon you, it is often more advantageous to model words as embedding vectors x1,….,xT, rather than one-hot vectors (which tokens w1,…wT corresponds to), so our first step is often to do an embedding table look-up for each input word. Let’s assume 300-dimensional word embedding and, for simplicity, a minibatch of size 1.
* A screenshot of a computer code

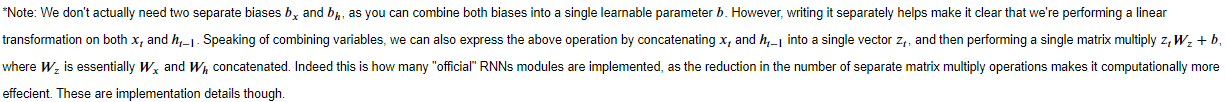
  Description automatically generated
  + Notice that we have formatted our inputs as (words x minibatch x embedding dimension). This is the preferred input ordering for PyTorch RNNs.
  + Let’s say we want to process this example. In our previous sentiment analysis example, we just took the average embedding 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 revisit the fully connected layer that we used in our logistic regression and multilayer perceptron examples, with a few changes in notation:
    - 
  + Instead of calling the results 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 “tanh()” or “o()” nonlinearity. Visualized:
    - A diagram of a mathematical equation

      Description automatically generated
  + 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 “x1,….,xT” 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:
  + 
* Where “tanh()” is the hyperbolic tangent, a nonlinear activation function. RNNs process words one at a time in sequence (x1), producing a hidden state “ht” 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 “xt”, and then applying a nonlinearity. Notice that we apply the same linear transformation (Wx, bx) at every time step. The difference is that we also apply a separate linear transformation (Wh, bh) to the previous hidden state ht-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:
* A diagram of a machine

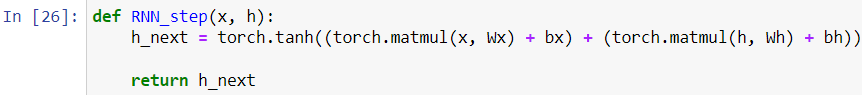
  Description automatically generated with medium confidence
* You can think of these recurrent connections as allowing the model to consider previous hidden states of a sequence when calculating the hidden state for the current input.



**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”
* A close up of words

  Description automatically generated
* In an RNN, we project both the input ”xt” and the previous hidden state ht-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.
* A screenshot of a computer program

  Description automatically generated
* For convenience, we define a function for one time step of the RNN. This function take the current input “xt” and previous hidden state “ht-1” performs the linear transformations  and then a hyperbolic tangent nonlinearity.
* 
* 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:

A screenshot of a computer

Description automatically generated

**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

**A screenshot of a computer

Description automatically generated**

* What do we do with these hidden states? It depends on the model and ask. Just like multilayer perceptron and convolutional neural networks, RNNs can be stacked in multiple layers as well. In this case, the outputs h1,…,ht are the sequential inputs to the next layer. If the RNN layer is the final layer, hT or the mean/max of h1,…,ht 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 https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Note that “RNN” as a name is somewhat overloaded: it can refer to both the basic recurrent model 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.
* **A screen shot of a computer code

  Description automatically generated**

**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.