Word Embeddings and Sequence-to-Sequence Learning

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 2nd April, 2024 23:11

Preamble

To TOC To HowTo

In the year 2013, a group of researchers at Google created a revolution in the text processing community with the publication of the *word2vec* neural network for generating numerical representations for the words in a text corpus. Here are their papers on that discovery:

https://arxiv.org/abs/1301.3781
https://arxiv.org/pdf/1310.4546.pdf

The word-to-numeric representations created by the word2vec network are vectors of real numbers, the size of the vectors being a hyperparameter of the network. These vectors are called word embeddings.

The word embeddings generated by word2vec allow us to establish word similarities on the basis of word contexts.

To elaborate, if two different words, $word_i$ and $word_j$, are frequently surrounded by the same set of other words (called the context words), the word embedding vectors for $word_i$ and $word_j$ would be numerically close.

Preamble (contd.)

What's amazing is that when you look at the similarity clusters produced by word2vec, they create a powerful illusion that a computer has finally solved the mystery of how to automatically learn the semantics of the words.

Word2vec word representations were followed by FastText representations from the folks at Facebook AI Research. The FastText representations remove one of the main limitations of word2vec: **no** vectors for words that did not exist in the corpus used for learning the representations.

Each word in FastText is represented by a sequence of what are known as n-grams, the idea of which is best illustrated with an example. Here is a 3-gram sequence based representation of the word "hello": `<he, hel, ell, llo, lo>` where the characters '<' and '>' are special and used as marker characters to indicate the beginning of a word and its end.

The word vectors in FastText are generated for each n-gram separately and the word vector for a full word is taken to be the vector sum of its component n-gram based word vectors.

Preamble (contd.)

The ploy of creating word vectors for the n-gram components of words allows FastText to generate word vectors even for unseen words since the n-gram decomposition of such words are likely to have been seen during training.

The following paper explains the n-gram based subword decomposition idea on which the FastText algorithm is based:

https://arxiv.org/abs/1607.04606

And the following paper is the main publication on FastText:

https://arxiv.org/pdf/1802.06893.pdf

Regard the topic of word embeddings, in addition to introducing word2vec and FastText, I'll also mention some of my lab's research on using the Word2vec embeddings in the context of software engineering for automatic bug localization.

****

The second major part of this lecture is on sequence-to-sequence (seq2seq) learning that is used in automatic machine translation and automatic question-answering systems.

Preamble (contd.)

For seq2seq, I'll start by presenting the RNN-based Encoder-Decoder architectures that are used for such learning. The Encoder RNN scans one-word-at-a-time through a sentence from the source language and generates a fixed-sized "hidden" state representation of the sentence. Subsequently, the Decoder RNN translates the hidden state produced by the Encoder into the corresponding sentence in the target language. Each word at the output of the Decoder is a function of the hidden state produced by the Encoder and the previous word produced by the Decoder.

Modern versions of the Encoder-Decoder RNNs use the notion of attention for aligning sentence fragments in the two languages that carry the same meaning. Simply stated, the attention is a separate network that learns the associations between the corresponding sentence fragments in the two languages.

The latest approaches in seq2seq learning do away entirely with recurrence and solve the whole problem using just the attention networks. One of the most famous of these is known as the BERT model that is described in:

`https://arxiv.org/abs/1810.04805`

I'll be discussing such purely attention based networks in my Week 14 lecture.

Preamble – How to Learn from These Slides

Here's some help with what to focus on during your first reading of this material. At the beginning, just focus on understanding the following three items:

- I believe that the most beautiful idea presented in this lecture deals with how to create a numeric representation for the words so that a computer acquires what's usually referred to as the semantics. If someone were to ask you: "What's a car?" How would you respond? This core concept is explained in Slides 17 through 34, for a total of 17 slides.

- How to make this concept of semantic understanding of words even more powerful by first breaking up the words into n-grams is explained on Slides 36 through 48, for a total of just 12 slides.

- The more you understand how to use a GRU, especially when a GRU is used in the bidirectional mode, the more likely that you will acquire a powerful tool for your own research. So take the trouble to understand what's on Slides 49 through 58, for a total of 19 slides.

That makes for a total of 48 slides that you need to focus on in your first reading of this material. Of the rest of the material, the discussion on Seq2Seq learning itself is not so critical at the moment since that type of learning is carried out with Transformers that I'll cover next week. Nonetheless, it is good to know it is this work in this lecture that first introduced the idea of Attention in neural learning.

Outline

Outline

Role of Text Search and Retrieval in our Lives

- In order to underscore the importance of word embeddings, I am going to start by highlighting the role played by text search and retrieval in our day-to-day lives. I'll also bring to your attention the major forums where this type of research is presented.
- A good indicator of the central importance of text search and retrieval is the number of times you google something in a 24-hour period.
- Even without thinking, we now bring up an app on our mobile device or on a laptop to get us instantaneously what it is that we are looking for. We could be looking for how to get to a restaurant, how to spell a long word, how to use an idiom properly, or about any of a very large number of other things.
- You could say that instantaneous on-line text search has now become an important part of our reflexive behavior.

Text Search and Retrieval and Us (contd.)

- In my own lab (RVL), we now have a history of around 15 years of research during which we have used text search and retrieval tools to develop ever more powerful algorithms for automatic bug localization for large software libraries. [Purdue University was recently granted a patent based on our work. The title of the patent is "Bug Localization Using Version History", US Patent 10108526.]

- The earliest approaches to text search and retrieval used what are known as the Bag-of-Words (BoW) algorithms. With BoW, you model each document by a histogram of its word frequencies. You think of the histogram as a vector whose length equals the size of the vocabulary. Given a document, you place in each cell of the histogram the number of times the corresponding word appears in the document.

- The document vector constructed as described above is called the term frequency vector. And the term-frequency vectors for all the documents in a corpus define what is known as the term-frequency matrix.

Text Search and Retrieval and Us (contd.)

- You construct a similar vector representation for the user's query for document retrieval. Using cosine distance between the vectors as a similarity metric, you return the documents that are most similar to the query.
- The BoW based text retrieval algorithms were followed by approaches that took term-term order into account. This was a big step forward. The best of these were based on what is known as the Markov Random Fields (MRF) model. Here is a link to our publication on this topic

  `https://engineering.purdue.edu/RVL/Publications/SismanAkbarKak_2016.pdf`

- In an MRF based framework, in addition to measuring the frequencies of the individual words, you also measure the frequencies of ordered pairs of words. The results obtained with MRF based approach were a significant improvement over those obtained with BoW based methods.

Text Search and Retrieval and Us (contd.)

- Most modern approaches to text search and retrieval also include what I called "contextual semantics" in the modeling process. The best-known approaches for contextual semantics use the distributed numeric representations for the words in the form of word2vec and FastText word vectors.
- Before ending this section, I just wanted to mention quickly that the most important annual research forums on general text retrieval are the ACM's SIGIR conference and the NIST's TREC workshops. Here are the links to these meetings:

  `https://sigir.org/sigir2019/`

  `https://trec.nist.gov/pubs/call2020.html`

- Since my personal focus is on text retrieval in the context of extracting information from software repositories, the go-to meeting for me is the annual MSR (Mining Software Repositories) conference.

Text Search and Retrieval and Us (contd.)

- MSR has emerged as the world's premier conference for "data science, machine learning, and AI in software engineering".

- For those interested, here is a link to the main conference. The second link is to our publication in 2020 edition of the conference. Hopefully, you won't be too offended by a bit of shameless self-promotion here.

https://conf.researchr.org/home/msr-2022

https://2020.msrconf.org/details/msr-2020-papers/35/A-Large-Scale-Comparative-Evaluation-of-IR-Based-Tools-for-Bug-Localization

Outline

Word2vec

- I'll explain the word2vec neural network with the help of the figure shown on the next slide.
- The files in a text corpus are scanned with a window of size $2W + 1$. The word in the middle of the window is considered to be the focus word. And the W words on either side are referred to as the context words for the focus word.
- We assume that the size of the vocabulary is V.
- As a text file is scanned, the V-element long one-hot vector representation of each focus word is fed as input to the neural network.
- Each such input goes through the first linear layer where it is multiplied by a matrix, denoted $W_{V \times N}$, of learnable parameters.

Word2vec (contd.)

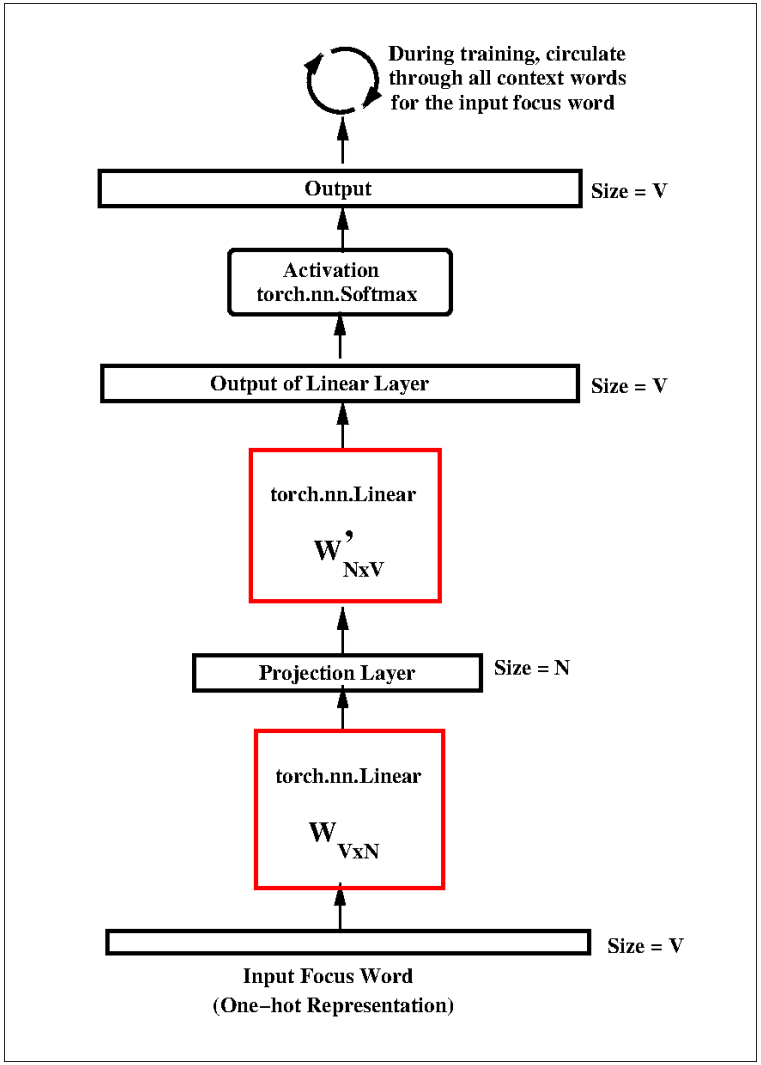


Figure: The SkipGram model for generating the word2vec embeddings for a text corpus.

Word2vec (contd.)

- Overall, in the word2vec network, a V-element tensor at the input goes into an N-element tensor in the middle layer and, eventually, back into a V-element final output tensor.
- Here is a very important point about the first linear operation on the input in the word2vec network: In all of the DL implementations you have seen so far, a `torch.nn.Linear` layer was always followed by a nonlinear activation, but that's NOT the case for the first invocation of `torch.nn.Linear` in the word2vec network.
- The reason for the note in red above is that the sole purpose of the first linear layer is for it to serve as a projection operator.
- But what does that mean?
- To understand what we mean by a projection operator, you have to come to terms with the semantics of the matrix $W_{V \times N}$. And that takes us to the next slide.

Word2vec (contd.)

- You see, the matrix $W_{V \times N}$ is actually meant to be a stack of the word embeddings that we are interested in. The i^{th} row of this matrix stands for the N-element word embedding for the i^{th} word in a sorted list of the vocabulary.
- Given the one-hot vector for, say, the i^{th} vocab word at the input, the purpose of multiplying this vector with the matrix $W_{V \times N}$ is simply to "extract" the current value for the embedding for this word and to then present it to the neural layer that follows.
- You could say that, for the i^{th}-word at the input, the role of the $W_{V \times N}$ matrix is to project the current value of the word's embedding into the neural layer that follows. It's for this reason that the middle layer of the network is known as the projection layer.
- In case you are wondering about the size of N vis-a-vis that of V, that's a hyperparameter of the network whose value is best set by trying out different values for N and choosing the best.

Word2vec (contd.)

- After the projection layer, the rest of the word2vec network as shown in Slide 16 is standard stuff. You have a linear neural layer with `torch.nn.Softmax` as the activation function.
- **To emphasize, the learnable weights in the $N \times V$ matrix W' along with the activation that follows is the ONLY neural layer in word2vec.**
- The next section has further details regarding the `torch.nn.Softmax` activation function.
- To summarize, word2vec is a single-layer neural network that uses a projection layer as its front end.
- While the figure on Slide 16 is my visualization of how the data flows forward in a word2vec network, a more common depiction of this network as shown on the next slide.

Word2vec (contd.)

- This figure is from the following publication by Shayan Akbar and myself:

https://engineering.purdue.edu/RVL/Publications/Akbar_SCOR_Source_Code_Retrieval_2019_MSR_paper.pdf

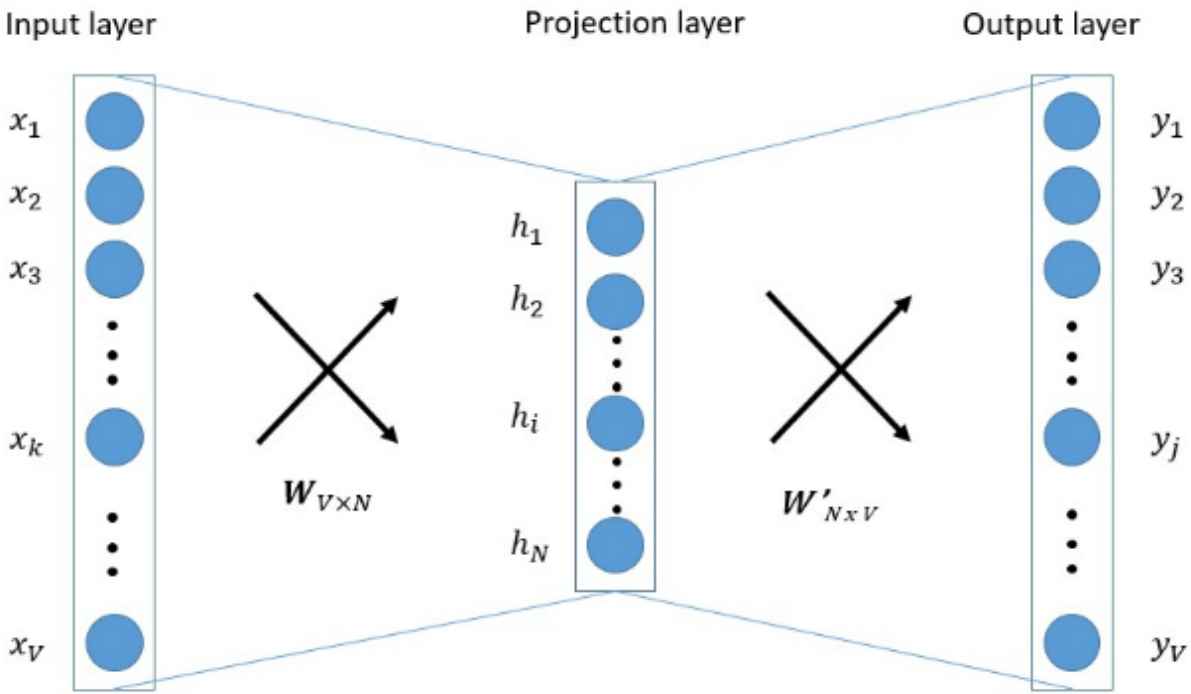


Figure: A more commonly used depiction for the SkipGram model for generating the word2vec embeddings for a vocabulary

Outline

The Softmax Activation Function

- An important part of education is to see how the concepts you have already learned relate to the new concepts you are about to learn. The following comment is in keeping with that precept.
- As you saw on Slide 30 of the Week 12 Lecture on Recurrent Neural Networks, the *activation* function `LogSoftmax` and the *loss* function `NLLLoss` are typically used together and their joint usage amounts to the same thing as using the *loss* function `CrossEntropyLoss` that I had presented previously in the Week 7 lecture on multi-instance object detection and localization (see Slides 28 and 29 of that lecture).
- In the sense mentioned above, we can say that `NLLLoss`, `LogSoftmax`, and `CrossEntropyLoss` are closely related concepts, despite the fact that two of them are loss functions and one an activation function.
- On the next slide, I'll add to the mix of those three related concepts the *activation* function `Softmax`.

The Softmax Activation Function (contd.)

- The Softmax activation function shown below looks rather similar to the cross-entropy loss function presented on Slides 28 and 29 of the Week 7 lecture on Multi-Instance Object Detection:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

yet the two functions carry very different meanings, which has nothing to do with the fact that the cross-entropy formula requires you take the negative log of a ratio that looks like what is shown above.

- The cross-entropy formula presented in the Object Detection lecture focuses specifically on just that output node that is supposed to be the true class label of the input notwithstanding the appearance of all the nodes in the denominator that is used for the normalization of the value at the node that the formula focuses on.

- On the other hand, the Softmax formula shown above places equal focus on all the output nodes. That is, the values at all the nodes are normalized by the same denominator.

The Softmax Activation Function (contd.)

- The best interpretation of the formula for Softmax shown on the previous slide is that it converts all the output values into a probability distribution.
- As to why, the value of the ratio shown in the formula is guaranteed to be positive, is guaranteed to not exceed 1, and the sum of the ratios calculated at all the output nodes is guaranteed to sum to 1 exactly.
- That the output of the activation function can be treated as a probability is important to word2vec because it allows us to talk about each output as being the conditional probability of the corresponding word in the vocab being the context word for the input focus word.
- To elaborate, let w_i represent the i^{th} row of the W_{VxN} matrix of weights for the first linear layer and let w'_j represent the j^{th} column of the W'_{NxV} matrix of weights for the second linear layer in the word2vec network.

The Softmax Activation Function (contd.)

- If we use x_j to denote the output of the second linear layer (that is, prior to its entering the activation function) at the j^{th} node, we can write [Each column of the matrix W'_{NxV} contributes to a single node at the output.]

$$x_j = w'^{T}_{j} w_i \tag{2}$$

- In light of the probabilistic interpretation given to the output of the activation function, we now claim the following: If we let $p(j|i)$ be the conditional probability that the j^{th} vocab word at, obviously, the j^{th} output node is a context word for the i^{th} focus word, we have

$$p(j|i) = \frac{e^{w'^{T}_{j} w_i}}{\sum_k e^{w'^{T}_{k} w_i}} \tag{3}$$

- The goal of training a word2vec network is to maximize this conditional probability *for the actual context words* for any given focus word.

The Softmax Activation Function (contd.)

- That takes us to the following issues:
  1. How to best measure the loss between the true conditional probability for the context words and the current estimates for the same; and
  2. How to backpropagate the loss?
- These issues are addressed in the next section.

Outline

Training for Word2vec

- As you now know, the word2vec network is trained by scanning the text corpus with a window of size $2W+1$, where W is typically between 5 and 10, for each focus word, testing the output against the one-hot representations for the $2W$ context words for the focus word.
- That raises the issue of how to represent the context words at the output of the neural network for calculating the loss.
- In keeping with the conditional probability interpretation of the forward-projected output as presented in the last section, the target output could be a V-element tensor that is a $2W$-version of a one-hot tensor: A V-element tensor in which just those elements are 1 that correspond to the context words, with the rest of the elements being 0s.

Calculating the Loss

- A target tensor such as the one described on the previous slide would look like

$$p_T(j|i) = \begin{cases} \frac{1}{2W+1} & j \text{ is context word for } i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

- The calculation of loss now amounts to comparing the estimated probability distribution on Slide 25 against the target probability distribution shown above. This can be done by measuring the cross-entropy between the two distributions using the formula on Slide 22 of my Week 7 lecture. [The distribution $p_T(j|i)$ shown above is the same as p in the Week 7 lecture and $p(j|i)$ shown on Slide 25 here is the same as q in the Week 7 slide.]. Therefore,

$$\begin{aligned} Loss_i(p_T, p) &= -\sum_{j \in context(i)} p_T(j|i) \cdot \log_2 p(j|i) \\ &= -\sum_{j \in context(i)} \frac{1}{2W+1} \cdot \log_2 \left(\frac{e^{w'_j{}^T w_i}}{\sum_k e^{w'_k{}^T w_i}} \right) \\ &= \sum_{j \in context(i)} -\log_2 \left(e^{w'_j{}^T w_i} \right) + \log_2 \left(\sum_k e^{w'_k{}^T w_i} \right) \qquad \text{ignoring inconsequential terms} \\ &= \sum_{j \in context(i)} -w'_j{}^T w_i + \log_2 \left(\sum_k e^{w'_k{}^T w_i} \right) \end{aligned}$$

Computing the Gradients of the Loss

- The subscript i in the notation $Loss_i$ is meant to indicate that the i^{th} word of the vocabulary is the focus word at the moment and that the loss at the output is being calculated for that focus word.
- Despite the fact that the Loss formula shown on the previous slide is based entirely on the notion of cross-entropy as applied to an estimated probability distribution on Slide 25 and the target probability distribution shown at the top on the previous slide, it is not possible to use PyTorch's `nn.CrossEntropyLoss` for measuring this loss on account of the assumptions that the PyTorch's implementation makes about the input to the network and how its class labels are represented implicitly.
- As I see it, you have no choice but to write your own custom implementation for the loss shown on the previous slide. As shown on the next slide, it is actually quite easy to write your own code for updating the learnable parameters based on this loss.

The Gradients of the Loss (contd.)

- For arbitrary values for the indices s and t, we get the following expression for the gradients of the loss with respect to the elements of the matrix W:

$$\frac{\partial Loss_i}{\partial w_{st}} = \sum_{j \in context(i)} -\frac{\partial(\vec{w'}_j^T \vec{w}_i)}{\partial w_{st}} + \frac{1}{\sum_k e^{\vec{w'}_k^T \vec{w}_i}} \frac{\partial(\sum_k e^{\vec{w'}_k^T \vec{w}_i})}{\partial w_{st}} \tag{5}$$

  where I have introduced the arrowed vector notation $\vec{w}_i$ so that you can distinguish between the elements of the matrix and the row and column vectors of the same matrix.

- You will end up with a similar form for the loss gradients $\frac{\partial Loss_i}{\partial w'_{st}}$.

- I leave it to the reader to simplify further these expressions. You might find useful the derivations presented in the following paper:

  `https://arxiv.org/abs/1411.2738`

Outline

Incorporating Negative Examples for Context

- In this and the next slide, I will show an alternative formulation of the loss function that also incorporates examples of randomly chosen negative context words. Incorporating negative examples can significantly improve the convergence during training.
- I'll start with the realization that whatever exponent $0 \leq s \leq 1$ maximizes e^s, which stands for the numerator in Eq. (3) on Slide 25, the same s will minimize the value of $1 + e^{-s}$. With regard to just the positive context words, this amounts to *minimizing* the following expression:

$$\log \left(1 + e^{-w'_j{}^T w_i} \right) \tag{6}$$

- Now we can write the following expression for loss that includes both the positive and the negative examples of context.

$$Loss_i \quad = \quad \sum_j \log \left(1 + e^{-w'_i{}^T w_j} \right) \quad + \quad \sum_{n \in \mathcal{N}_{i,C}} \log \left(1 + e^{w_i^T n} \right) \tag{7}$$

See the next slide for the notation used in the second term at right.

Incorporating Negative Examples (contd.)

- In Eq. (7), I have used the symbol n to represent a negative context word and the notation $\mathcal{N}_{i,c}$ to represent a set of such negative words chosen randomly from the vocabulary for a given focus word w_i. The index j spans the context neighborhood for the focus word i:
- **Note that in Eq. (7) the exponents for the negative words are opposite to what you see for the positive context word w_j.**
- The loss described in Eq. (7) can be implemented with the PyTorch function `nn.BCEwithLogitsLoss`, which is fundamentally the same as what I presented for the Binary Cross Entropy Loss (`nn.BCELoss`) in Slides 33 through 37 of my Week 7 lecture, except that the former combines the `nn.Sigmoid` activation with the loss calculation. [As you will recall from the Week 7 slides, the calculation of `nn.BCELoss` is preceded by `nn.Sigmoid` activation.]
- For further information regarding how negative context words are used in the word2vec algorithm, see the two publications mentioned on Slide 2 in the Preamble section.

Outline

FastText Word Embeddings

- Much of what I have mentioned about word2vec also applies to the FastText word embeddings. The configuration of the neural network, the loss function, and the training routine are all the same in both cases.
- However, there is one very important difference between the two: representing each word as a sequence of n-grams in FastText.
- This one difference makes for a significant difference between the two types of embeddings with regard to how they are used in an application: Word2vec can only supply embeddings for words seen at the training time. Therefore, with word2vec, if a word is encountered that was not present in the corpus used for training, you will just have to ignore that word. On the other hand, FastText is highly likely to return an embedding for a previously unseen word because it would have learned the embeddings for at least some of the n-grams of such a word.

FastText Word Embeddings (contd.)

- That FastText is able to return embeddings for unseen words is illustrated in the script presented on Slide 40. In line (L) I ask word2vec to give me the embedding for my first name "avinash", and, as you can see in the next line, it comes back with the message that it does not have it.
- On the other hand, when I ask FastText the same question in line (q) on Slide 42, it has no trouble returning an embedding as you can see in the output shown partially in the commented out section that starts in line (r). For FastText, the word "avinash" is actually the following *running* sequence of 3-grams:

```
<av  avi  vin  ina  nas  ash  sh>
```

- As I mentioned in the Preamble, the characters '<' and '>' play an important role in this decomposition of a full word into a sequence of n-grams — they represent the word boundaries.

FastText Word Embeddings (contd.)

- It's because of the marker characters '<' and '>' that FastText can distinguish between "ash" as a 3-gram component in the sequence shown on the previous slide and "ash" as in the full word "ash".
- In the example shown on the previous slide, FastText considers the embedding vector for the word "avinash" to be a vector sum of the vectors for the individual 3-grams shown above and it is highly likely that FastText would have learned the embeddings for those through other words in the training corpus.
- A word in FastText is decomposed into a sequence of n-grams for values of n between 3 and 6, both ends inclusive.
- About the script that follows, the goal is to make you familiar with the functions you need to call in order to carry out lookups and reverse lookups of the embeddings. A lookup here means getting hold of the numeric vector for a given word and a reverse lookup means finding the word whose embedding is closest to the vector you have supplied.

FastText Word Embeddings (contd.)

- Lines (H) and (e) illustrate looking up the word embeddings for the words 'king' and 'hola' respectively. That reminds me to mention that FastText embeddings database I have used in the script is for Spanish.
- The reverse lookup is illustrated by lines (Q) and (S) for word2vec and by lines (g) through (k) for FastText. Note the role played by the argument `topn` for word2vec.
- The reverse lookups allow you find the nearest symbolic words for a given pure noise vector. This is illustrated by lines (T) through (W) for word2vec and by lines (l) through (o) for FastText. Note that for the FastText case, I am using the same noise vectors that I had constructed for word2vec.

Using word2vec and FastText

```
##  word_emdeddings_demo.py

import gensim.downloader as api
from gensim.models import KeyedVectors
import fasttext.util
import numpy
import sys,os

print("\n\n\nDemonstrating word2vec: ")

if os.path.exists('/home/kak/TextDatasets/word2vec/vectors.kv'):                                    ## (A)
    word_vectors = KeyedVectors.load('/home/kak/TextDatasets/word2vec/vectors.kv')                  ## (B)
else:
    # download the word2vec model and return as object ready for use                                ## (C)
    word_vectors = api.load("word2vec-google-news-300")                                             ## (D)
    word_vectors.save('/home/kak/TextDatasets/word2vec/vectors.kv')                                 ## (E)

size_vecs = len(word_vectors)                                                                       ## (F)
print(size_vecs)                                        ## 3000000                                 ## (G)

wv1 = word_vectors['king']                                                                          ## (H)
print(wv1.shape)                                        ## (300,)                                  ## (I)

# With the numpy dot operator (cosine distance)
car_similars = word_vectors.most_similar(positive=['car'], topn=5)                                  ## (J)
print(car_similars)                                                                                 ## (K)
                                                  ##  [('vehicle', 0.7821096777915955),
                                                  ##   ('cars', 0.7423828840255737),
                                                  ##   ('SUV', 0.7160965204238892),
                                                  ##   ('minivan', 0.690703272819519),
                                                  ##   ('truck', 0.6735787987709045)]
## Let's now try a word that is very likely to not have been used in training:
word = 'avinash'                                                                                    ## (L)
if word in word_vectors.key_to_index:                                                               ## (M)
    wv2 = word_vectors[word]                                                                        ## (N)
    print("shape of the word vector for 'avinash': ", wv2.shape)                                    ## (O)
else:
    print("the word '%s' does not exist in the model" % word)   ## Does not exist                   ## (P)

##  Let's now attempt a REVERSE LOOK-UP:
wv_king = word_vectors['king']                                                                      ## (Q)
print(word_vectors.most_similar(positive=[wv_king], topn=1))   ##  [('king', 1.0)]                  ## (R)

##  Let's find similar words using the REVERSE LOOK-UP word vectors:
print(word_vectors.most_similar(positive=[wv_king], topn=5))                                        ## (S)
                                                ## [('king', 1.0),
                                                ##  ('kings', 0.7138045430183411),
                                                ##  ('queen', 0.6510959267616272),
                                                ##  ('monarch', 0.6413195133209229),
                                                ##  ('crown_prince', 0.6204220056533813)]
```

(Continued on the next slide .....)

word2vec and FastText (contd.)

(..... continued from the previous slide)

```
##  Doing reverse look-up on noise vectors:
print("\n\nfrom noise: ")
noise1 = numpy.random.normal(0.0, 1.0, 300)    # mean=0.0, sigma-1.0                    ## (T)
print(word_vectors.most_similar(positive=[noise1], topn=5))                            ## (U)
                                                ## [('X##_motherboard', 0.31045278906822205),
                                                ##  ('Safari_#.#.#', 0.3070869743824005),
                                                ##  ('Mac_OS_X_v##.#', 0.30088552832603455),
                                                ##  ('Mac_OS_X_##.#.##', 0.2867443263530731),
                                                ##  ('QuickTime_X.', 0.2844076156616211)]
print("\n\nfrom noise: ")
noise2 = numpy.random.normal(0.0, 1.0, 300)         # mean=0.0, sigma-1.0               ## (V)
print(word_vectors.most_similar(positive=[noise2], topn=5))                            ## (W)
                                                ## [('Haminu_Draman', 0.2874550521373749),
                                                ##  ('Bono_Manso', 0.2849479019641876),
                                                ##  ('Whacks_Museum', 0.26943090558052063),
                                                ##  ('Covadonga', 0.26873889565467834),
                                                ##  ('Biblioteca', 0.258213073015213)]


#########################################################################################################################

print("\n\n\nDemonstrating FastText: ")

if os.path.exists("/home/kak/TextDatasets/fastText/cc.es.300.bin"):                    ## (a)
    word_vecs = fasttext.load_model("/home/kak/TextDatasets/fastText/cc.es.300.bin")   ## (b)
else:
    fasttext.util.download_model('es', if_exists='ignore')                             ## (c)
    word_vecs = fasttext.load_model("/home/kak/TextDatasets/fastText/cc.es.300.bin")   ## (d)

wv3 = word_vecs['hola']                                                                ## (e)
print(wv3)                                                                             ## (f)
                            ##  [-5.66945970e-02  5.34497127e-02 -6.12863861e-02 -2.43984938e-01
                            ##   -1.36605650e-01  5.66679165e-02  5.90700731e-02 -1.81467310e-02
                            ##   -9.96223241e-02 -1.20628618e-01 -5.05264848e-02  7.94697106e-02
                            ##    1.05264515e-01  6.81780279e-02  1.32493883e-01  3.31371576e-02
                            ##    ...
                            ##    ...

##  For REVERSE LOOK-UPS with FastText:
all_words = list(word_vecs.get_words())                                                ## (g)
all_vecs = numpy.array([word_vecs[x] for x in all_words])                              ## (h)

word_idx = fasttext.util.util.find_nearest_neighbor(wv3, all_vecs, [])                 ## (i)
print(word_idx)                                                 ## 3448               ## (j)
print(all_words[word_idx])                                      ## hola               ## (k)

##  REVERSE LOOK-UP from the noise vectors constructed earlier:
word_idx = fasttext.util.util.find_nearest_neighbor(noise1, all_vecs, [])              ## (l)
print(all_words[word_idx])                                      ## L (unicode)        ## (m)
```

(Continued on the next slide .....)

word2vec and FastText (contd.)

(..... continued from the previous slide)

```
word_idx = fasttext.util.util.find_nearest_neighbor(noise2, all_vecs, [])                    ## (n)
print(all_words[word_idx])                                                                   ## (o)

print("Asking FastText to return the word embedding for 'avinash'")                          ## (p)
wv4 = word_vecs['avinash']                                                                   ## (q)
print(wv4)                                                                                   ## (r)
                              ## [-0.09531539  0.00596228  0.09024861  0.01947961  0.00944935 -0.00325132
                              ##  -0.01753134 -0.01016876  0.02512926  0.00074796 -0.0085186   0.03262738
                              ##   0.0706223  -0.01324025 -0.00633249 -0.07459014 -0.03304705 -0.06904534
                              ##   0.03225741  0.05071883  0.01590452  0.02105784  0.07789602 -0.0056043
                              ##  -0.06514778 -0.00764755 -0.00491822  0.03812853  0.02453358 -0.02561558
                              ##    ...
                              ##    ...
similars = word_vecs.get_nearest_neighbors('avinash', k=5)                                   ## (s)
print(similars)                                                                              ## (t)
                                    ##  [(0.47731107473373413, 'avinado'),
                                    ##   (0.46117398142814636, 'avinagra'),
                                    ##   (0.4502473473548889, 'avin'),
                                    ##   (0.434394508600235, 'Avinash'),
                                    ##   (0.4335462152957916, 'Honna') ]
```

Outline

Using Word2Vec for Text Retrieval

- The numerical word embeddings generated by the word2vec network allow us to establish an easy-to-calculate similarity criterion for the words: We consider two words to be similar if their embedding vectors are close using, say, the Euclidean distance between them.
- Just to show an example of how powerful word2vec is at establishing word similarities (that a lot of people would refer to as "semantic similarities"), the figure on the next slide shows three of the word clusters discovered by word2vec in the context of mining software repositories.
- The similarity clusters shown on the next slide were obtained by Shayan Akbar in his research on automatic bug localization. To generate the word embeddings, he downloaded 35,000 Java repositories from GitHub which resulted in 0.5 million software-centric terms. So 500,000 is the size of the vocabulary here. He used $N = 500$ for the size of the embedding vectors.

Using Word2Vec for Text Retrieval (contd.)

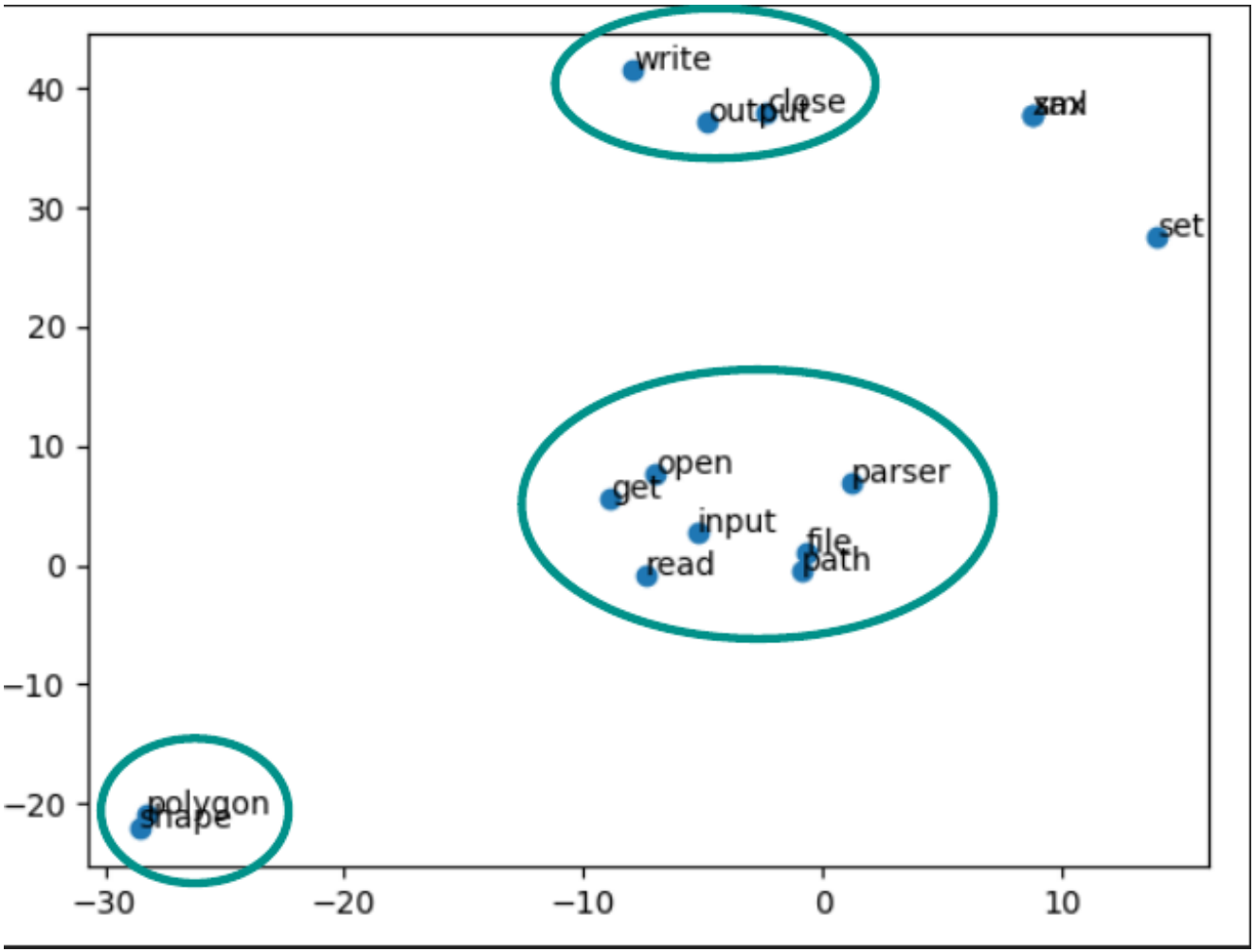


Figure: Some examples of word clusters obtained through the similarity of their embeddings.

Using Word2Vec for Text Retrieval (contd.)

- Show below are additional examples of word similarities discovered through the same word embeddings as mentioned on Slide 44. In this display, we seek the three words that are most similar to the words listed in the top row.

- You've got to agree that it is almost magical that after digesting half a million software-centric words, the system can figure out automatically that "parameter", "param", "method", and "argument" are closely related concepts. The same comment applies to the other columns in the table.

alexnet	delete	rotation	add	parameter
resnet	remove	angle	list	param
lenet	update	rot	set	method
imagenet	copy	lhornang	create	argument

Figure: Additional examples of software-centric word similarities based on learned their embeddings.

Using Word2Vec for Text Retrieval (contd.)

- Now that we know how to establish "semantic" word similarities, the question remains how to use the similarities for improving the quality of retrieval.
- How that problem was solved in the context of retrieval from software repositories is described in our 2019 MSR publication:

https://engineering.purdue.edu/RVL/Publications/Akbar_SCOR_Source_Code_Retrieval_2019_MSR_paper.pdf

Outline

A Quick Review of What You Know About GRU

- Based on what you learned from my Week 12 material, especially from its Section 9 entitled "*Understanding the torch.nn.GRU API*", you probably think you have already conquered the GRU beast. As it turns out, there's more to a GRU than what you have seen so far.

- Based on the material covered earlier, you already know that the behavior of a GRU is determined by its constructor parameters **and** by the shape of the input data.

- You also know already that the GRU constructor has two required parameters: `input_size` and `hidden_size`, where `input_size` refers to the size of the tensor that represents *each element* of the input sequence. For example, if the RNN is meant to process a sequence of words, `input_size` is the size of the tensor for each word. The other required constructor parameter, `hidden_size`, refers to the size of the hidden state for the RNN.

A Quick GRU Review (contd.)

- About the shape of the input to a GRU, you should already know from Week 12 that it must be a tensor of 3 axes, with the dimensionality of the first axis being equal to the length of the input sequence, the dimensionality of the second axis the batch size, and the dimensionality of the third axis the input size, which would be the same as in the constructor call..
- You also know from Week 12 that the GRU constructor takes on an optional parameter named `num_layers` whose default value is 1. When its value is greater than 1, that has a bearing on the shape of the tensor used for initializing the hidden state of the GRU. Setting `num_layers` to greater than 1 means you want to use "stacked GRUs" in which the output of one GRU is fed as input into the next GRU. For the case of stacked GRUs, you must initialize the hidden state separately in each of the GRU layers. The shape of the overall tensor you supply for initializing the hidden state is `(num_layers, batch_size, hidden_size)`.

A Quick GRU Review (contd.)

- You have **two choices** for how you feed data into a GRU: one sequence element at a time in your own loop, or an entire sequence in one fell swoop. If the dimensionality of the first axis of the input data tensor is 1, then you are feeding one element at a time into the GRU. On the other hand, if that dimensionality is *N*, you are feeding a sequence of *N* elements into the GRU.

- Finally — and this is the last bit of review of what you learned in Week 12 — after it has done its thing, the GRU outputs two things: *output* and *hidden*. **If you feed only one sequence element into the GRU, the values emitted through output and hidden are identical, they are both the hidden state after one state transition. On the other hand, when you feed an entire sequence into the GRU at one time, the output emits the time-evolution of the hidden state and the hidden emits the final value of the same hidden state.**

Bidirectional GRU

- From the standpoint of sequence-to-sequence learning (seq2seq), we need to bring into play another optional GRU constructor parameter: *bidirectional*. When you set `bidirectional` to True for an instance of a GRU, you are asking the GRU to carry out both a left-to-right and a right-to-left scan of the input sequence. You can think of it as constructing two logical GRUs, one for forward scanning and the other for backward scanning.

- **When you run a GRU in the bidirectional mode, it changes the nature of what the GRU emits at its output. Now, each element of the time-evolution of the hidden that is emitted at the GRU output is a concatenation of the forward hidden and the backward hidden.** Very confusing, isn't it? The examples that follow will clarify this very important point.

- In addition, in the bidirectional mode, when you initialize the hidden state, you must provide initializations for both the forward-scanning GRU and the backward-scanning GRU.

Bidirectional GRU (contd.)

- Starting with this slide, I'll illustrate the bidirectional operation of a GRU with a simple example.
- So that you can directly compare the unidirectional and the bidirectional modes, let's first construct a unidirectional GRU with the following constructor call:

```
##                  input_size     hidden_size     num_layers
rnn    =    nn.GRU(    4,              7,              1       )
```

- Fundamentally, this GRU is meant for an input-data tensor whose first axis is of dimensionality 1 and the third axis of dimensionality 4. (The second axis of the input is to allow for batching.) **However, the GRU will allow you to feed into it a data tensor whose first axis has dimensionality greater than 1. If you do that, the GRU assumes that you are feeding into it an entire sequence as opposed to a single element of a sequence in your own loop for scanning a sequence.**

Bidirectional GRU (contd.)

- At this time, I'll assume that we want to feed into the GRU a one-element sequence as generated by the following call to `torch.randn()`:

```
##                       sequence length    batch_size     input_size
input  =  torch.randn(   1,                    1,             4         )

print(input)      ##  tensor([[[ 0.7012, -0.8035, -0.4593,  0.3234]]])
```

- Before we can run this GRU, we must also initialize its hidden state. Since `num_layers=1` and `hidden_size=7` in the GRU constructor specification, we need a 7-valued tensor for this initialization. Embedding those 7 values in the tensor shape that the GRU expects to see for its hidden entails the following statement:

```
##                              num_layers    batch_size    hidden_size
hidden_initial  =  torch.randn(    1,            1,          7        )

print(hidden_initial)  ##  tensor([[[-0.2320, 0.6029, -0.2614, 0.5694, 0.4084, -0.8387, 1.0738]]])
```

Bidirectional GRU (contd.)

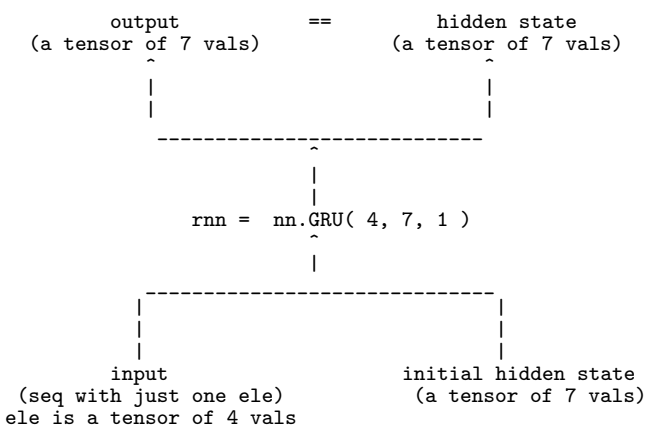
- Note that I used `torch.randn()` for generating data for both the input and the initializer for the hidden. In general, `torch.randn()` generates as many random numbers as needed and then shapes them into a tensor whose shape is based on the specification implied by its arguments.
- So far we have put in place an instance of `nn.GRU` and created data for its input sequence and for the initialization of its hidden state. The input sequence consists of just one element. Let's now run the GRU:

```
output, hidden   =  rnn(input, hidden_initial)

print(output)      ##  tensor([[[-0.2539,  0.5069, -0.1797,  0.2584,  0.0355, -0.2593,  0.1481]]],
                       grad_fn=<StackBackward>)

print(hidden)      ##  tensor([[[-0.2539,  0.5069, -0.1797,  0.2584,  0.0355, -0.2593,  0.1481]]],
                       grad_fn=<StackBackward>)
```

- **Note that the output of the GRU is identical to its final hidden. Also note that they are both of the same size — the size of the hidden-state.** Also note that since the input sequence has only one element, the GRU can go through only one state transition.

Bidirectional GRU (contd.)

- What you have seen is fundamental to the operation of a GRU: After the GRU has finished doing its thing, the *output* and the *hidden* emit the same information, with the only difference that the *output* is the time-evolution of the hidden-state and the *hidden* its final value.
- For the example we have gone through, since only one state transition can take place when the input sequence has just one element, the data emitted for the *output* is exactly the same as for the *hidden*.
- The following 'diagram' may help some visualize the GRU action just presented for the case of a one-element input sequence:



Bidirectional GRU (contd.)

- I'll now run the same experiment as described above, but with a **bidirectional version** of the GRU. Here is new constructor declaration:

```
##             input_size   hidden_size        num_layers
rnn = nn.GRU(      4,           7,                1,          bidirectional=True   )
```

- We will use an input of the same shape as earlier:

```
##                    sequence length    batch_size     input_size
input  =  torch.randn(   1,                 1,             4       )

print(input)         ##  tensor([[[-0.6126, -0.1930, -0.0133,  1.6677]]])
```

  So, we again have an input sequence with just one element in it. The element is a tensor with four values in it.

- Next, we must specify the hidden state of the GRU. And that's where you will see your first difference between the unidirectional GRU and the bidirectional GRU, as described on the next slide.

Bidirectional GRU (contd.)

- While we could use an input of the same shape as earlier, we cannot do that for the hidden state of the bidirectional GRU.
- That's because a bidirectional GRU is, logically speaking, two GRUs, one for forwarding scanning through the input sequence and the other for backward scanning. We must supply each GRU with its own initializer for the hidden state. Our initializer for the hidden state now takes the form:

```
    ##                                num_direcs    batch_size    hidden_size
hidden_initial  =  torch.randn(         2,            1,           7       )

print(hidden_initial)   ## tensor([[[-0.5219,  0.2482, -2.0191, -0.5873, -0.0695, -0.4062,  2.4795]],
                        ##
                        ##         [[-0.0858, -0.8242,  0.0205,  2.0175,  0.0234,  0.6766, -1.6794]]])
```

- Now we are ready to run the bidirectional GRU.

Bidirectional GRU (contd.)

- Shown below are the call to the GRU instance we created and two quantities it returns:

```
output, hidden   =  rnn(input, hidden_initial)


print(output)         ## tensor([[[-0.0863,  0.1825, -1.5969, -0.4559, -0.2648,  0.0814,  1.0873,
                                    0.0889, -0.5061, -0.1972,  1.1838,  0.3071,  0.5952, -1.1093]]],
                              grad_fn=<CatBackward>)

print(output.shape)   ## torch.Size([1, 1, 14])

print(hidden)         ## tensor([[[-0.0863,  0.1825, -1.5969, -0.4559, -0.2648,  0.0814,  1.0873]],

                                  [[ 0.0889, -0.5061, -0.1972,  1.1838,  0.3071,  0.5952, -1.1093]]],
                              grad_fn=<StackBackward>)

print(hidden.shape)   ## torch.Size([2, 1, 7])
```

- Pretty interesting that, for the case of a one-element sequence, the numerical data emitted for both the *output* and the *hidden* are identical, **but the shapes of the two are different.**

Bidirectional GRU (contd.)

- **From the standpoint of sequence-to-sequence learning that I'm going to take up in the next section, the most important thing to note on the previous slide is that, for the case of a one-element input sequence, the final hidden returned by the GRU is a one-element sequence of 14 values.**
- **These 14 values in the hidden returned by the GRU are a concatenation of the 7 values of the final hidden state in the forward direction and the 7 values of the final hidden state in the backward direction. Don't forget that the "job" of the GRU output is to emit the time evolution of the hidden states and, since our input sequence has only one element in it, there can only be one state transition in either direction.**
- The above comments were about the *output* returned by the GRU. About the *hidden* returned by the GRU, its "job" is to emit the just the final hidden states in the forward and the backward directions.

Bidirectional GRU (contd.)

- The last bullet on the previous slide implies that what is returned by the GRU for *hidden* must be a sequence of two final hidden states, each with 7 values in it. That should explain the shape of the *hidden* returned by the GRU as shown on Slide 59.
- The important lesson learned from the "experiment" with a one-element input sequence so far is that while the *output* and the *hidden* returned by the GRU contain the same data values, the meanings to be associated with the two are totally different, as made evident by the shapes of the two returned tensors.
- Our discussion so far has been limited to a one-element input sequence. Starting with the next slide, let's generalize that discussion to an input sequence of 9 elements. We will use the same bidirectional GRU as in the previous experiment:

Bidirectional GRU (contd.)

- Shown below is a call to the GRU constructor — it's same as you saw earlier for the case of a one-element input sequence and when we chose the bidirectional option. Also shown below is the 9-element input sequence.

```
##             input_size     hidden_size        num_layers
rnn = nn.GRU(     4,              7,                1,         bidirectional=True)

##                     sequence length    batch_size     input_size
input  =  torch.randn(   9,                  1,              4      )

print(input)    ## tensor([[[-0.0280,  0.3677, -0.3888, -0.0923]],        ele 1 of input sequence
                ##
                ##          [[ 0.4612,  0.6639, -0.7859,  0.0536]],        ele 2 of input sequence
                ##
                ##          [[-0.0087, -0.5726, -1.2763,  1.1785]],        ele 3 of input sequence
                ##
                ##          [[ 0.1211,  0.4611, -0.0459, -0.0799]],        ele 4 of input sequence
                ##
                ##          [[-0.7091, -0.4798,  0.6764, -0.9395]],        ele 5 of input sequence
                ##
                ##          [[-0.8453, -1.0426,  0.2485, -0.8438]],        ele 6 of input sequence
                ##
                ##          [[ 0.8945,  0.2282,  0.1700,  0.7814]],        ele 7 of input sequence
                ##
                ##          [[ 0.3222,  1.3334,  0.0482,  0.4818]],        ele 8 of input sequence
                ##
                ##          [[ 0.4647, -0.8540, -1.5642, -0.1572]]])       ele 9 of input sequence
```

Bidirectional GRU (contd.)

- Next we need to initialize the hidden states in the forward scanning GRU and the backward scanning GRU. The following initialization is exactly the same as for the previous case of a one-element input sequence.

```
##                              num_directions    batch_size    hidden_size
hidden_initial  =  torch.randn(      2,               1,            7     )
```

- What that, we are ready to run the bidirectional GRU on the input sequence of 9 elements:

```
output, hidden   =  rnn(input, hidden_initial)
```

- The *output* and the *hidden* returned by the GRU are shown on the next slide.

Bidirectional GRU (contd.)

- Shown below are the *output* and the *hidden* returned by the call to the bidirectional GRU on the previous slide:

```
output, hidden   =  rnn(input, hidden_initial)

print(output)
                        ##  tensor([[[ 1.2178, -0.9199, -0.9202,  0.1650, -0.7796,  0.4539,  0.1888,
                        ##             0.0424,  0.0231,  0.4536,  0.5964,  0.6010,  0.5277, -0.0217]],
                        ##
                        ##          [[ 0.7494, -0.8732, -0.5087,  0.3077, -0.5332,  0.1886,  0.0397,
                        ##            -0.0245,  0.0999,  0.4346,  0.5812,  0.6121,  0.5463,  0.1162]],
                        ##
                        ##          [[ 0.5919, -0.3561, -0.3210,  0.2675, -0.4829, -0.1454,  0.2200,
                        ##            -0.1770,  0.3050,  0.4632,  0.5174,  0.4344,  0.5180,  0.0798]],
                        ##
                        ##          [[ 0.3993, -0.3687, -0.4272,  0.1480, -0.3141, -0.2276, -0.0882,
                        ##             0.2014, -0.1510,  0.4245,  0.5002,  0.2798,  0.4132, -0.2530]],
                        ##
                        ##          [[ 0.1298, -0.1016, -0.7360,  0.1579, -0.0390, -0.1244, -0.3113,
                        ##             0.1414, -0.1505,  0.4799,  0.6357,  0.0446,  0.3271, -0.4953]],
                        ##
                        ##          [[-0.0847,  0.1141, -0.8309,  0.2015,  0.1495, -0.0727, -0.1176,
                        ##             0.0327, -0.0036,  0.4124,  0.6920,  0.1982,  0.4531, -0.3890]],
                        ##
                        ##          [[ 0.0234, -0.0262, -0.3654, -0.0411, -0.0215, -0.5593, -0.1840,
                        ##             0.1269, -0.0063,  0.2586,  0.6776,  0.4045,  0.7701, -0.0504]],
                        ##
                        ##          [[ 0.1731, -0.2131, -0.1409, -0.1020, -0.2105, -0.4998, -0.3507,
                        ##             0.2566, -0.0137,  0.3215,  0.8463,  0.2388,  0.8366,  0.0329]],
                        ##
                        ##          [[-0.3638, -0.3396, -0.2189, -0.0195, -0.0921, -0.4463, -0.0333,
                        ##             0.2711,  0.0267,  0.5689,  1.5115, -0.4199,  0.9507, -0.2853]]],
                        ##         grad_fn=<CatBackward>)

print(output.shape)     ##  torch.Size([9, 1, 14])

print(hidden)

                        ##  tensor([[[-0.3638, -0.3396, -0.2189, -0.0195, -0.0921, -0.4463, -0.0333]],
                        ##
                        ##          [[ 0.0424,  0.0231,  0.4536,  0.5964,  0.6010,  0.5277, -0.0217]]],
                        ##         grad_fn=<StackBackward>)

print(hidden.shape)     ##  torch.Size([2, 1, 7])
```

Bidirectional GRU (contd.)

- With the *output* shown on the previous slide, you can better visualize the time evolution of the hidden state in the GRU. **But note that each element of this time evolution is a concatenation of the 7-valued hidden state in the forward direction and the 7-valued hidden state in the backward direction at the same location in the 9-element input sequence.**

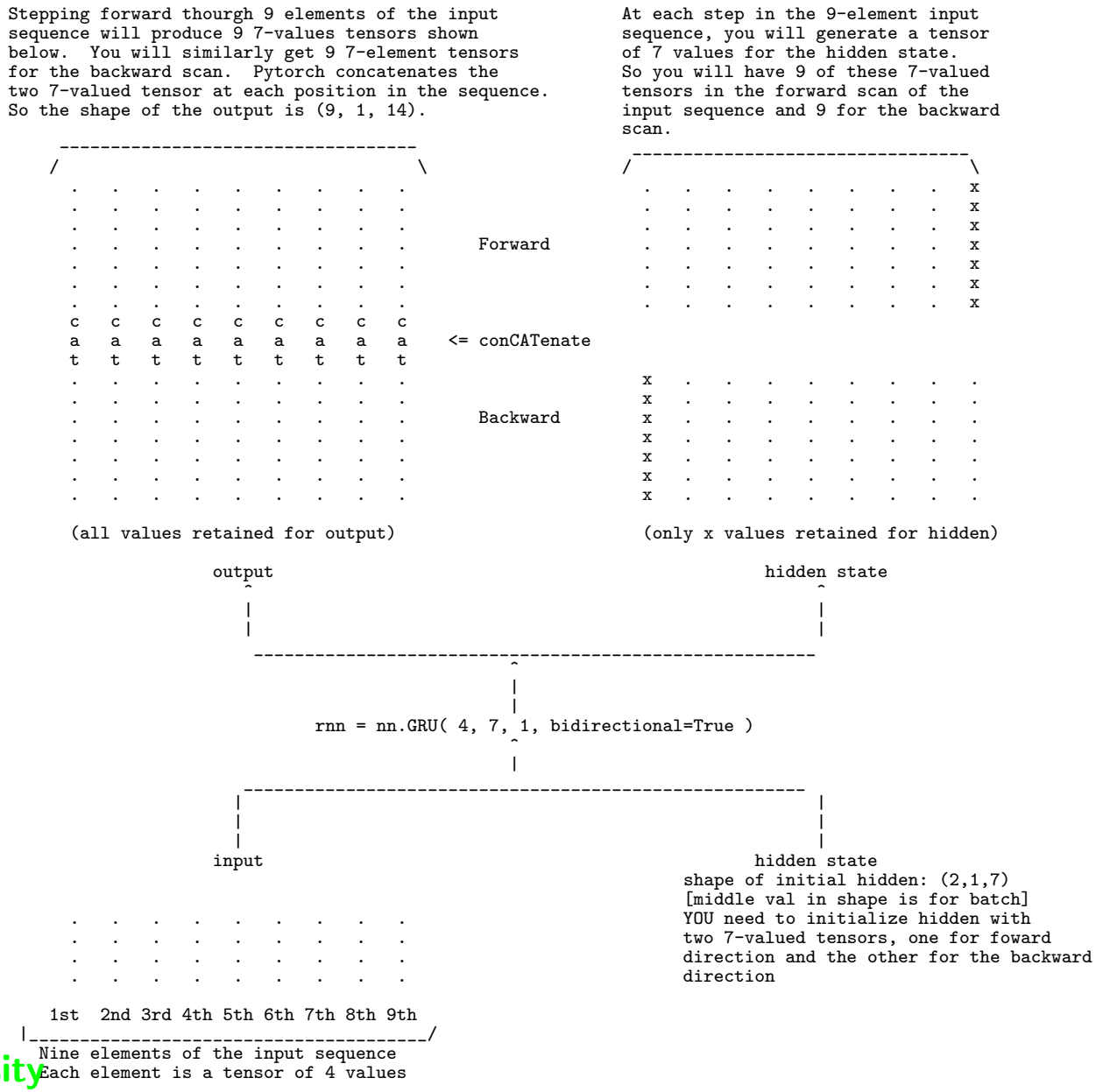
- Therefore, despite the fact that the hidden state in either direction is meant to be defined by a 7-valued tensor, what you get in each element of the time evolution is a 14-valued tensor, as confirmed by the shape [9,1,14] of the *output* shown on the previous slide.

- Note also that what is emitted by the *hidden* of the bidirectional GRU for the case of a 9-element input sequence is of exactly the same shape as for the case of a 1-element input sequence.

Bidirectional GRU (contd.)

- **In the context of the hidden returned by the GRU, a most important observation is that it is a sequence of TWO 7-valued tensors and the first of these is exactly the FIRST SEVEN VALUES in the last element of the output emitted by the GRU and the second is exactly the SECOND SEVEN VALUES in the first element of that output.**
- Perhaps the visualization shown on the next slide would help you consolidate your insights related to the production of the *output* and the *hidden* that are returned by a bidirectional GRU for the case of a 9-element input sequence.
- In the depiction shown on the next slide, a vertical column of 7 stands for the 7 values of the hidden state.
- The concatenation of the 7-valued hidden states during the forward scan of the input sequence with the 7-valued hidden states during the backward scan is depicted by the vertically arranged "cat".

Bidirectional GRU (contd.)



Bidirectional GRU (contd.)

- In conclusion, when the hidden-size declared in the constructor of a bidirectional GRU is 7 and you see a tensor with 14 values corresponding to a certain position in the *output* sequence, the first 7 of those are for the hidden state in the forward-scanning GRU at that position and the second 7 of those are for the hidden state backward-scanning GRU at the same position.
- In sequence-to-sequence learning, we will refer to the individual elements of what is returned for *output* by a bidirectional GRU as the **attention units**. As to why they are called that will be made clear in the section that follows.
- In general, when the input sequence consists of N elements, the GRU will return N **attention units** for the sequence.

Outline

Sequence-to-Sequence Learning

- From Week 12 lecture, you already know about Recurrent Neural Networks (RNN) for modeling variable length sequence data. When an RNN steps through, say, the words in a sequence (which might represent a sentence in some language), it outputs a hidden state, which is a fixed-sized encoding of the input sequence. **In Seq2Seq learning, this encoding is also referred to as the context vector.**

- In seq2seq learning, the goal is to use a Decoder RNN that works just like the Encoder RNN except that the Decoder puts to use the hidden state produced by the Encoder to generate, one word a time, each word for the output sequence that depends on the previous output word and the hidden state that was produced by the Encoder.

- This Encoder-Decoder operation is illustrated in the figure on the next slide.

Sequence-to-Sequence Learning (contd.)

- Early work on seq2seq learning for automatic machine translation was based on the approach depicted in the figure shown below.
- In the figure, C is the final value for the hidden state in the Encoder. It represents a fixed-sized encoding of a variable length input sentence. Subsequently, as shown in the figure, the stepping action in the decoder RNN generates an output sequence, one word at at time, with each output word a function of the previous output word and the encoding C of the input sequence.

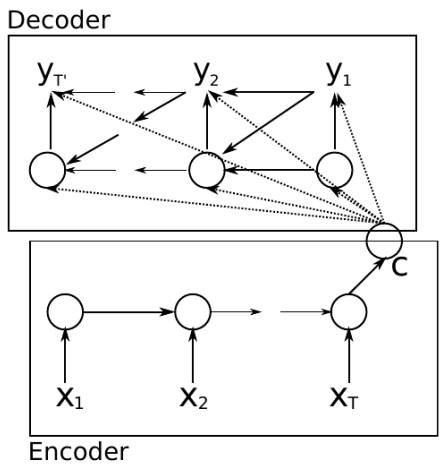


Figure: Using two RNNs for seq2seq learning for machine translation. Figure taken from https://arxiv.org/pdf/1507.01053.pdf.

Sequence-to-Sequence Learning (contd.)

- Following Bahdanau et al. (https://arxiv.org/pdf/1409.0473.pdf), let $(x_1, x_2, \ldots, x_{T_x})$ represents the input sequence and h_i the hidden state at the i^{th} step of the encoder, the operation of the Encoder RNN is described by the following equation for some nonlinear $f()$:

$$h_t = f(x_t, h_{t-1}) \tag{8}$$

- The quantity h_{T_x} is the final hidden state produced by the Encoder.
- To describe the operation of the Decoder RNN, at each step t, it outputs a word denoted y_t that depends on the word y_{t-1} produced at the previous step $t-1$, its own hidden state denoted s_t and, of course, the final Encoder hidden state h_{T_x}.
- During training we compare the words that are output by the Decoder against the words in the target sequence. We sum the losses for each output word and backpropagate the error over the Encoder-Decoder combine.

Sequence-to-Sequence Learning (contd.)

- If we use an appropriate loss function for the Decoder RNN, we can estimate the conditional probabilities at the output of the Decoder:

$$prob(y_t \mid y_{t-1}, s_t, h_{T_x}) \quad = \quad prob(y_t \mid \{y_1, y_2, \ldots, y_{t-1}\}, h_{T_x}) \tag{9}$$

- An Encoder-Decoder framework based solely on the logic presented so far has a fundamental weakness in the rather limited capacity of the final Encoder hidden state to represent the structure of arbitrarily long input sequences in the source language.
- This problem was solved by Bahdanau et al. (https://arxiv.org/pdf/1409.0473.pdf) by proposing an **attention mechanism** that informs the Decoder about the most relevant part of the input sequence to use for a given output word.
- Theoretically, any attention mechanism can be represented by changing the step-invariant context vector h_{T_x} to a context vector that customizes itself to what's needed at each decoder step.

Bahdanau et al. Attention

- The context customization mentioned in the last bullet on the previous slide can be expressed by changing the last of the conditioning arguments in Eq. (9) to

$$prob(y_t \mid y_{t-1}, s_t, c_t) \quad = \quad prob(y_t \mid \{y_1, y_2, \ldots, y_{t-1}\}, c_t) \tag{10}$$

  where c_t is allowed to be whatever it needs to be for the minimization of the loss.

- With a conventional RNN for the encoder, the context vector c_i will be a nonlinear function of all the hidden states discovered during the operation of the RNN.

- In the method proposed by Bahdanau et al., however, c_i is set to a concatenation of the corresponding hidden states discovered in a left-to-right scan of the input sequence and the hidden states discovered during the right-to-left scan of the same sentence.

Bahdanau et al. Attention

- The figure shown below illustrates the components $\overrightarrow{h_i}$ and $\overleftarrow{h_i}$ of each context vector c_i. The symbols $\overrightarrow{h_i}$ represent the hidden states in a forward scan of the input sentence and the symbols $\overleftarrow{h_i}$ represent the corresponding hidden states in a backward scan of the same sentence.

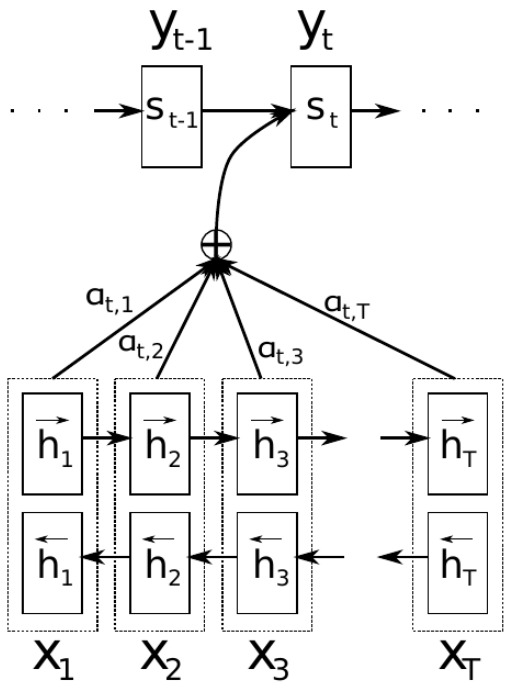


Figure: Figure taken from `https://arxiv.org/pdf/1409.0473.pdf`.

Bahdanau et al. Attention

- What the figure illustrates so well is that ordinarily what goes into the computation at each time step of the Decoder RNN would be a constant context vector, which is generally the last hidden state produced by the Encoder, but now it is a weighted contribution from each of the combined $h_j = [\overrightarrow{h_i}, \overleftarrow{h_i}]$ hidden states that were computed during a bidirectional scan of the input sentence. The quantity in the brackets is supposed to be a concatenation of the forward and the backward hidden states.

- The following equation shows us calculating the context vector c_i needed at each time step of the decoder from the h_j units defined above:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{11}$$

Bahdanau et al. Attention

- In the Bahdanau framework, each attention weight α_{ij} is set to

$$\alpha_{ij} = \frac{e^{a(s_{i-1}, h_j)}}{\sum_{k=1}^{T_x} e^{a(s_{i-1}, h_k)}} \tag{12}$$

  where $a(s_{i-1}, h_j)$ represents the alignment model that tells us how well the input words around the position j match the words around the position i in the decoder output.

- That opens up question of where to get the weights $\alpha_{t,1}, \alpha_{t,2}, \ldots, \alpha_{t,T}$ shown in the figure.

- These weights are learned in a separate network called the Attention Network.

Outline

Programming Issues in Seq2Seq

- Even apart from the attention and how you may go about coding it, there are important other issues you need to be thinking about as you sit down to write code for solving a problem in sequence-to-sequence learning.
- For the sake of shining light on these issues, I'll assume that you are trying to solve a problem in language translation. I'll refer to the languages involved as the source language and the target language.
- First and foremost, you will need to decide what to do about embeddings. Embeddings are the numeric vector representations for the words in the languages.
- Regarding embeddings, the two choices you'd need to consider are learnable embeddings and pre-trained embeddings.

Programming Issues in Seq2Seq (contd.)

- If you decide to go with learnable embeddings, you are likely to be using the `nn.Embedding` layer in the encoder and the decoder for generating the embeddings for the words in your training data. Learning the best embedding vectors for the words then becomes a part of what you get by minimizing the overall loss that is estimated by comparing the predicted target words with the ground-truth target words.
- On the other hand, if you decide to go with pre-trained embeddings, you're likely to have to choose between word2vec embeddings and the Fasttext embeddings. Obviously, you could also use the two types of embeddings both at the same time, one for the source language and the other for the target language.
- **A big disadvantage of seq2seq with learnable embeddings:** The learned embeddings will be specific to the training dataset used. That can be a serious problem since languages in general are always in a state of flux.

Programming Issues in Seq2Seq (contd.)

- To elaborate the last point made on the previous slide, let's say you have created a translation system for a medical specialty domain using all the training data you could get hold of in that domain. Now consider the problems that would arise if a medical doctor wrote up a report using words and idioms that were not included in the training data.

- **A big disadvantage of seq2seq with pre-trained embeddings:** The pre-trained embeddings like word2vec and Fasttext are learned from humongous general-purpose datasets, Google news for the former and Wikipedia for the latter. When you use such embeddings for seq2seq training in a specialty domain, the embeddings may not represent all of the word nuances specific to that domain. On the basis of a non-rigorous study, my to-be-taken-with-a-grain-of-salt observation is that an seq2seq based on pre-trained embeddings is more likely to generate nonsense translations for a given source sentence than the one based on learnable embeddings.

Programming Issues in Seq2Seq (contd.)

- Another disadvantage of using pre-trained embeddings is they can take up a lot of space in the fast memory in your computer. Typically, word2vec needs around 3.5GB of space in your RAM and Fasttext around 8GB.

- The other programming issue you have to resolve is something that is common to all problems in deep learning: The best strategy to use for computing the loss when comparing the words of a predicted translation with the words in the actual ground-truth translation.

- When using pre-trained embeddings, it is tempting to think (at least for a few brief moments) of seq2seq as an exercise in regression. [Let's say you decide to use embeddings of size, say, 300 for both the source language and the target language. For a given sentence in the source language, each predicted word in the target language would be a vector of size 300 that you could compare with the ground-truth embedding vector of the same size. The distance between the two vectors could be construed as loss that could be backproped for updating the learnable parameters, just as you would do in solving a regression problem. I have tried this approach for fun and my experience was that the results you get are generally quite dismal.]

Programming Issues in Seq2Seq (contd.)

- You get much better results if you implement seq2seq as a solution to a classification problem using `nn.LogSoftmax` activation and `nn.NLLLoss`.
- In a classification based soluition, as you execute the decoder code on the encoder's final hidden state for a given source sentence, at each step in the decoder you would estimate the log-probabilities over all possible words in the target language by applying `nn.LogSoftmax` to the values accumulated in the final layer of the decoder. For obvious reasons, this requires that the size of final output layer of your decoder network be the size of the target language vocabulary. The integer index of each node in this layer would stand for a word in the target language vocab.
- By supplying the output of `nn.LogSoftmax` and the integer index of the ground-truth target word at that step of the decoder RNN to `nn.NLLLoss`, you would in effect be calculating the cross-entropy loss related to that prediction.

Outline

Seq2Seq with Learnable Embeddings

- Version 2.0.9 of DLStudio comes with educational material on sequence-to-sequence learning (seq2seq). To that end, I have included the following two new classes in DLStudio:

  1. `Seq2SeqWithLearnableEmbeddings` for seq2seq with learnable embeddings;
  2. `Seq2SeqWithPretrainedEmbeddings` for doing the same with pre-trained embeddings.

- Both seq2seq implementations include the attention mechanism based on my understanding of the original paper on the subject by Bahdanau, Cho, and Bengio. You will find the attention code in a class named `Attention_BCB`. For the sake of comparison, I have also included an implementation of the attention used in the popular NLP tutorial by Sean Robertson. You will find that code in a class named `Attention_SR`.

Seq2Seq with Learnable Embeddings (contd.)

- **The main idea in both the classes listed on the previous slide is to demonstrate the basic PyTorch structures and idioms to use for seq2seq learning.**

- The specific example of seq2seq considered in both those classes is translation from English to Spanish. (I chose this example because learning and keeping up with Spanish is one of my hobbies.)

- In the `Seq2SeqWithLearnableEmbeddings` class, the learning framework learns the best embedding vectors to use for the two languages involved from the training data provided. On the other hand, in the `Seq2SeqWithPretrainedEmbeddings` class, I use the word2vec embeddings provided by Google for the source language. As to why I use the pre-training embeddings for just the source language is explained in the next section.

Seq2Seq with Learnable Embeddings (contd.)

- The dataset that I have used for my seq2seq code in DLStudio is a slighted curated version of the English-Spanish (`spa-eng.zip`) archive posted at `http://www.manythings.org/anki/` by the folks at `tatoeba.org`.
- My alterations to the original dataset consist of expanding the contractions like "it's", "I'm", "don't", "didn't", etc., into their "it is", "i am", "do not", "did not", etc. Another alteration I made to the original data archive is to surround each sentence in both English and Spanish by the `SOS` and `EOS` tokens, with the former standing for "Start of Sentence" and the latter for "End of Sentence".
- In order to run the seq2seq related example scripts in DLStudio, you will need to download the following archive from the DLStudio webpage:

  `en_es_corpus_for_seq2sq_learning.tar.gz`

- In the rest of this section, I'll present DLStudio's seq2seq code based on learnable embeddings.

Seq2Seq with Learnable Embeddings (contd.)

- Shown below is the implementation of the encoder in the `Seq2SeqWithLearnableEmbeddings` part of DLStudio.

```
class EncoderRNN(nn.Module):
    def __init__(self, dls, s2s, embedding_size, hidden_size, max_length):
        super(DLStudio.Seq2SeqWithLearnableEmbeddings.EncoderRNN, self).__init__()
        self.dl_studio = dls
        self.source_vocab_size = s2s.vocab_en_size
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size
        self.max_length = max_length
        self.embed = nn.Embedding(self.source_vocab_size, embedding_size)
        self.gru = nn.GRU(embedding_size, hidden_size, bidirectional=True)                    ## (A)

    def forward(self, sentence_tensor, hidden):
        word_embeddings = torch.zeros(self.max_length, 1,
                                      self.hidden_size).float().to(self.dl_studio.device)     ## (B)
        for i in range(sentence_tensor.shape[0]):
            word_embeddings[i] = self.embed(sentence_tensor[i].view(1, 1, -1))
        output, hidden = self.gru(word_embeddings, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(2, 1, self.hidden_size).float().to(self.dl_studio.device)
```

- An explanation for the code shown starts in the next slide.

Seq2Seq with Learnable Embeddings (contd.)

- As shown in line (A) in the constructor section of the classes `EncoderRNN`, I am using a bidirectional GRU for the encoder.
- To fully understand the operation of the encoder, I'll assume that you have gone through and understood the material in Section 8 of this lecture on the subject of Bidirectional GRUs.
- The `forward()` method of `EncoderRNN` requires two inputs through the parameters `sentence_tensor` and `hidden`. Whereas the second, `hidden`, is just the initializer for the two hidden states of the bidirectional GRU of the encoder, the first, `sentence_tensor`, requires some explaining that's presented on the next slide.

Seq2Seq with Learnable Embeddings (contd.)

- Shown below is the implementation of the method `sentence_to_tensor()` that returns a tensor that becomes `sentence_tensor` for feeding into the `forward()` of the encoder:

```
def sentence_to_tensor(self, sentence, lang):
    list_of_embeddings = []
    words = sentence.split(' ')
    sentence_tensor = torch.zeros(len(words), 1, dtype=torch.long)
    if lang == "en":
        for i,word in enumerate(words):
            sentence_tensor[i] = self.en_vocab_dict[word]
    elif lang == "es":
        for i,word in enumerate(words):
            sentence_tensor[i] = self.es_vocab_dict[word]
    return sentence_tensor
```

- To explain the logic of the method shown above, say there are N words in a sentence (recall each sentence starts with the `SOS` token and ends with the `EOS` token; N is inclusive of these two marker tokens), the tensor produced by this function will be of shape $[N, 1]$. The next slide considers a specific example to show the nature of this tensor.

Seq2Seq with Learnable Embeddings (contd.)

- To illustrate the behavior of `sentence_to_tensor()` with an example, consider the following English sentence from the dataset:

```
SOS they live near the school EOS
```

- Including the two marker tokens, the source sentence shown above has 7 words in it. The contract of the `sentence_to_tensor()` method is to return the following tensor of shape `torch.Size([7, 1])` for such a sentence:

```
tensor([[    0],
        [10051],
        [ 5857],
        [ 6541],
        [10027],
        [ 8572],
        [    1]])
```

in which each integer is the index of the corresponding word in the sorted vocabulary for all the English sentences in the corpus.

Seq2Seq with Learnable Embeddings (contd.)

- Note that we manually insert the tokens `SOS` and `EOS` at the beginning of the sorted vocabulary mentioned on the previous slide. That's why the first and the last entries in the tensor shown on the previous slide are 0 and 1. For the other integers in the tensor, obviously, 10051 must be the index of the word "they" in the sorted vocabulary; 5857 the index for the word "live"; and so on.
- During training, similar tensors are constructed for the Spanish sentences. The integer indexes in those tensors serve as targets in the `nn.NLLLoss` based loss function.
- Getting back to the `EncoderRNN` implementation, it returns two things: The *output* and the *hidden*. Both of these are the same as returned by the bidirectional GRU of `EncoderRNN`. As explained in great detail in Section 8 of this lecture, the *output* of a bidirectional GRU is a time-evolution of the encoder hidden-state in which each element is a concatenation of the hidden state in the forward scanning GRU and the hidden state in the backward scanning GRU.

Seq2Seq with Learnable Embeddings (contd.)

- For a given source sentence, the sequence of the values returned through the *output* of the encoder are referred to as the attention units in the source sentence, as mentioned in the previous section of these slides. The other thing returned by `EncoderRNN` – *hidden* – is the final value of the hidden state in the encoder GRU and is meant to be used as a starting point by the decoder.

- One additional factor that is highly relevant to the action of `EncoderRNN`: the `max_length` parameter in line (B). As you will see shortly, this parameter plays an important role in the calculation of the attention weights that will eventually be needed by the decoder for producing the target sequence.

- Attention weights tell us how much contribution each attention unit in the source sentence makes to production of each output word.

Seq2Seq with Learnable Embeddings (contd.)

- That brings me to the presentation of the decoder class shown below:

```
    class DecoderRNN(nn.Module):
        def __init__(self, dls, s2s, embedding_size, hidden_size, max_length):
            super(DLStudio.Seq2SeqWithLearnableEmbeddings.DecoderRNN, self).__init__()
            self.hidden_size = hidden_size
            self.target_vocab_size = s2s.vocab_es_size
            self.max_length = max_length
            self.embed = nn.Embedding(self.target_vocab_size, embedding_size)
            self.attn_calc = DLStudio.Seq2SeqWithLearnableEmbeddings.Attention_BCB(
                                                dls,embedding_size,hidden_size,max_length)          ## (C)
#           self.attn_calc = DLStudio.Seq2SeqWithLearnableEmbeddings.Attention_SR(
#                                              dls,embedding_size, hidden_size, max_length)         ## (D)
            self.gru = nn.GRU(self.hidden_size, self.hidden_size)                                   ## (E)
            self.out = nn.Linear(self.hidden_size, self.target_vocab_size)                          ## (F)

        def forward(self, word_index, decoder_hidden, encoder_outputs):                             ## (G)
            embedding = self.embed(word_index).view(1, 1, -1)
            attentional_hidden, attn_weights = self.attn_calc(embedding,
                                                    decoder_hidden, encoder_outputs)                ## (H)
            output, hidden = self.gru(attentional_hidden, decoder_hidden)                           ## (I)
            output = nn.LogSoftmax(dim=0)(self.out(output.view(-1)))                                ## (J)
            output = torch.unsqueeze(output, 0)
            return output, hidden, attn_weights
```

Seq2Seq with Learnable Embeddings (contd.)

- As you can see in the code shown in the previous slide, the decoder factors in the attention weights returned by a separate Attention network that is called in line (E). The goal of the attention network is to modify the current hidden state in `DecoderRNN` taking into account all the attention units produced previously by `EncoderRNN`.

- The decoder has two choices for the attention mechanism, the one provided by the class `Attention_BCB` called in line (C) and the other provided by the commented-out call in line (D) to the attention class `Attention_SR`.

- The class `Attention_BCB` is based on my interpretation of the attention mechanism proposed by Bahdanau, Cho, and Bengio. The other attention class, `Attention_SR`, is based on Sean Robertson's implementation of attention in his very popular NLP tutorial at the PyTorch website.

Seq2Seq with Learnable Embeddings (contd.)

- During the training phase, the current ground-truth target word (needed for the prediction of the next target word) is fed into the decoder through the first argument in line (G) and the predicted next target word accessed through the value obtained in line (I). Both of these must be word index values, that is, integers that correspond to the word positions in the sorted vocabulary list for the target language.

- As was done for `EncoderRNN`, the input word (as a word index integer) is mapped to its embedding produced by the `nn.Embedding` layer and then supplied to the Attention class in line (H) that returns an *attentioned* version of the current decoder hidden state.

- The output returned by the GRU in line (I) is first sent through a linear layer, `self.out`, in line (I) that maps it into a vector whose size equals that of the target vocabulary size.

Seq2Seq with Learnable Embeddings (contd.)

- A most interesting thing about the decoder `output` is that it is kind-of wasted during training. During the evaluation phase, however, we apply `torch.max()` to the output of the decoder to find the integer index for the emitted output word. Subsequently, this integer index becomes the input to the decoder for the production of the next output word. However, during training, since the next input to the decoder will be the next word from the target sequence, we have no use for the current decoder output.

- That brings me to the implementation of the attention classes. As mentioned above, the model of attention in the class `Attention_BCB` is based on my interpretation of the logic presented in the following article by Bahdanau, Cho, and Bengio:

`https://arxiv.org/pdf/1409.0473.pdf`

That should explain the suffix "BCB" in the name of the class.

Seq2Seq with Learnable Embeddings (contd.)

- More specifically, the implementation in `Attention_BCB` corresponds to the Global Attention model described in Section 3.1 of the following paper by Luong, Pham, and Manning:

  `https://arxiv.org/pdf/1508.04025.pdf`

- Eq. (7) of the paper by Luong et al. says that if h_t represents the current hidden state in the `DecoderRNN` and if h_{s_i}, for $i = 1, \ldots, max\_length$, represent the attention units returned by the encoder, the contribution that each encoder attention unit h_{s_i} makes to the decoder h_t is proportional to

$$c_t(s_i) = \frac{e^{\text{score}(h_t, h_{s_i})}}{\sum_j e^{\text{score}(h_t, h_{s_j})}}$$

  where a "general" approach to estimating the score() for a given h_t with respect to all the encoder attention units h_{s_i}, $i = 1, 2, \ldots$, is given by

$$\text{score}(h_t, h_s) = h_t^T \cdot W_a \cdot h_s$$

Seq2Seq with Learnable Embeddings (contd.)

- Shown below is the implementation of the attention class `Attention_BCB`:

```
class Attention_BCB(nn.Module):
    def __init__(self, dl_studio, embedding_size, hidden_size, max_length):
        super(DLStudio.Seq2SeqWithLearnableEmbeddings.Attention_BCB, self).__init__()
        self.dl_studio = dl_studio
        self.max_length = max_length
        self.WC1 = nn.Linear( 2 * hidden_size, hidden_size )
        self.WC2 = nn.Linear( 2*hidden_size + embedding_size, embedding_size )

    def forward(self, prev_output_word, decoder_hidden, encoder_outputs):                          ## (K)
        contexts = torch.zeros(self.max_length).float().to(self.dl_studio.device)                  ## (L)
        for idx in range(self.max_length):                                                          ## (M)
            contexts[idx] = decoder_hidden.view(-1) @ self.WC1(encoder_outputs[idx].view(-1))       ## (N)
        weights = nn.LogSoftmax(dim=-1)(contexts)                                                   ## (O)
        attentioned_hidden_state =  weights @ encoder_outputs                                       ## (P)
        attentioned_hidden_state = nn.Softmax(dim=-1)(attentioned_hidden_state)
        output = self.WC2(torch.cat( (attentioned_hidden_state.view(-1),
                                                          prev_output_word.view(-1)), 0 ) )
        output = torch.unsqueeze(torch.unsqueeze(output, 0), 0)
        weights = torch.unsqueeze(weights, 0)
        output = nn.ReLU()(output)
        return output, weights
```

- There exist two matrix products in the code shown above: one in line (N) and the other in line (P). The one in line (N) is what's called for by the second of the two equations shown on the previous slide.

Seq2Seq with Learnable Embeddings (contd.)

- The matrix product in line (N) of the code on the previous slide amounts to multiplying each element of the current decoder hidden with a linear combination of all the attention units produced by the encoder for the source sentence in question. What you see in the first of the two equations on Slide 98 is implemented with the `nn.LogSoftmax` normalization in line (O).

- For comparison, I have also included in the seq2seq section of DLStudio the following attention class, `Attention_SR` that's based on Sean Robertson's popular NLP tutorial:

  `https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html`

  For the details regarding this attention model, it would be best to go to the tutorial directly.

- The code for `Attention_SR` is shown on the next slide.

Seq2Seq with Learnable Embeddings (contd.)

```
class Attention_SR(nn.Module):
    def __init__(self, dl_studio, embedding_size, hidden_size, max_length):
        super(DLStudio.Seq2SeqWithLearnableEmbeddings.Attention_SR, self).__init__()
        self.W = nn.Linear(embedding_size + hidden_size, max_length)
        self.attn_combine = nn.Linear(3*hidden_size, hidden_size)

    def forward(self, prev_output_word, decoder_hidden, encoder_outputs):
        contexts = self.W(torch.cat((prev_output_word[0], decoder_hidden[0]), 1))
        attn_weights = nn.Softmax(dim=1)( contexts )
        attn_applied = torch.unsqueeze(attn_weights, 0) @  torch.unsqueeze(encoder_outputs, 0)
        output = torch.cat((prev_output_word[0], attn_applied[0]), 1)
        output =  torch.unsqueeze(self.attn_combine(output), 0)
        output = nn.ReLU()(output)
        return output, attn_weights
```

Training Loss vs. Iterations for the Learnable Embeddings Case

Figure: Loss vs. iterations during training over 100,000 random draws from a pool of 98,988 sentence pairs. This plot was produced by the script `seq2seq_with_learnable_embeddings.py` in the `Examples` directory of DLStudio, version 2.0.9, with the learning rate set to 0.001.

Some Results Obtained with the Seq2Seq Network

- I'll now show some results obtained with the EncoderRNN-DecoderRNN-Attention_BCB network presented in this section for some example sentences.

- For the results shown, the sentences as extracted from the original dataset were limited in length to a maximum of 8 words. Including the tokens `SOS` and `EOS`, that meant any given sentence had a maximum of 10 words in it. Therefore, the `max_length` parameter in the code shown in this section was set to 10. As you know, this parameter plays an important role for "attentionizing" the hidden state of the decoder during its stepping action for the production of the words in the target language.

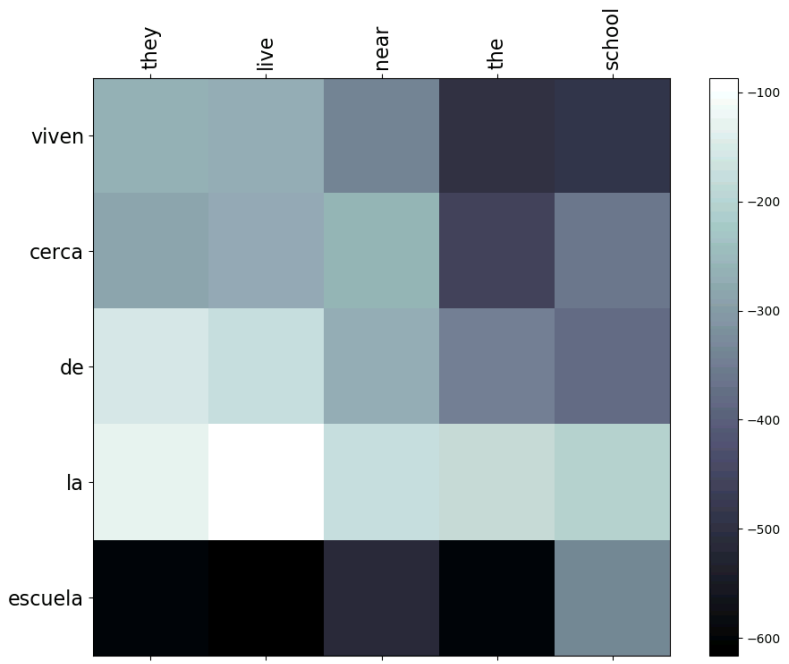
- The results shown in this section were obtained after training the overall network with 100,000 random draws from from a pool of 98988 sentence pairs. The learning rate was set to 0.001.

Some Results Obtained with the Seq2Seq Network

```
Original sentence in English:                          SOS they live near the school EOS
Spanish Translation in the "manythings" Corpus:        SOS viven cerca del colegio EOS

Translation Generated by the seq2seq network:          SOS viven cerca de la escuela EOS
```

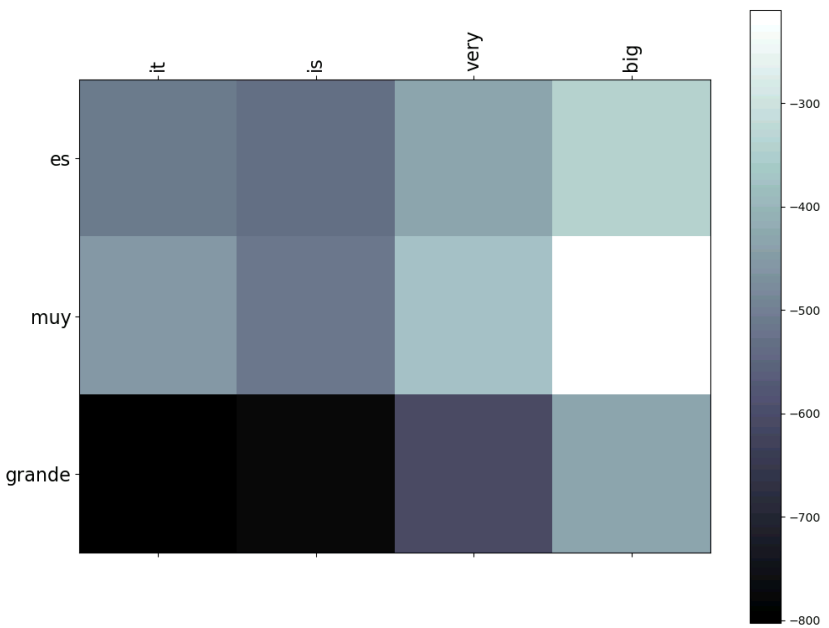
Comment: Easy case. Seq2Seq translation is excellent. "Colegio" and "escuela" are generally considered to be synonyms, although in some Spanish speaking countries one may stand for the elementary school and the other for the high school.

Attention depiction:



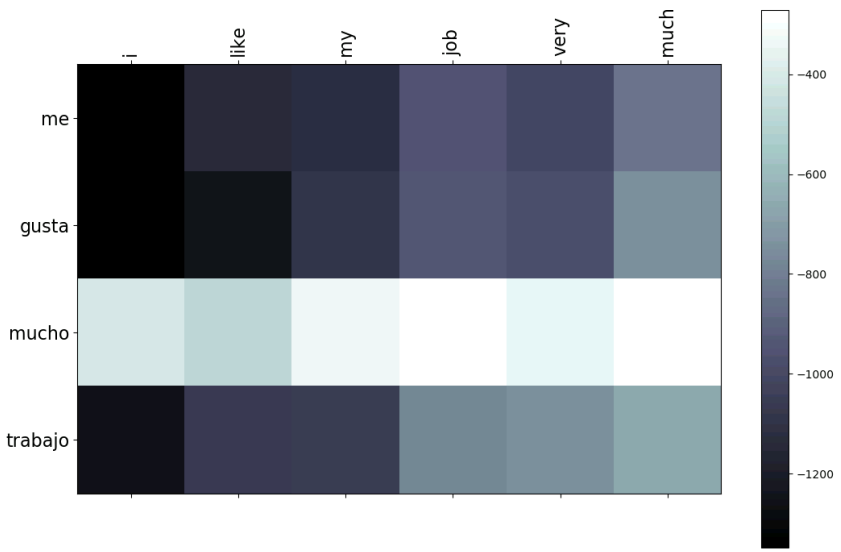
Some Results Obtained with Seq2Seq (contd.)

```
Original sentence in English:                          SOS it is very big EOS
Spanish translation in the "manythings" corpus:        SOS es muy grande EOS

Translation generated by the seq2seq network:          SOS es muy grande EOS

Comment:   Very easy case. Seq2seq translation is excellent.

Attention depiction:
```



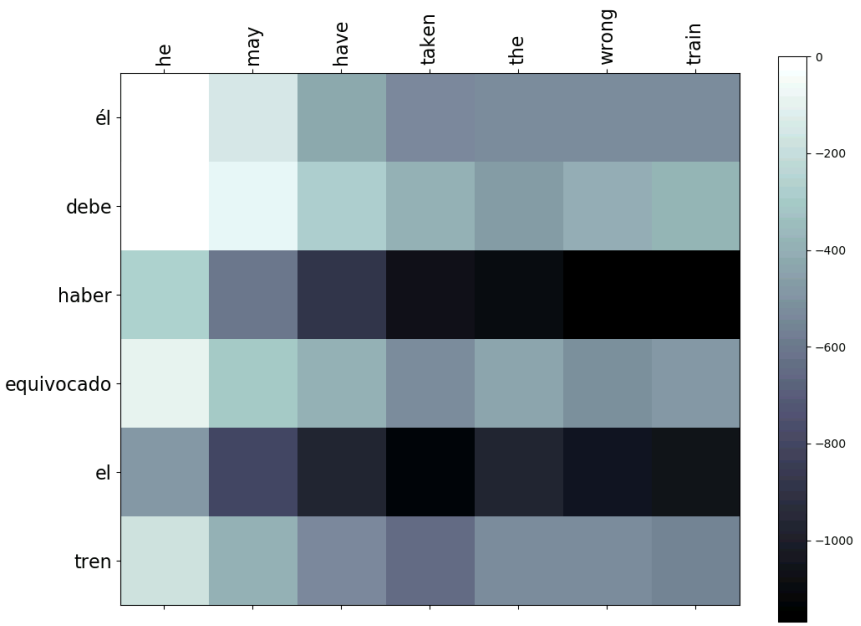
Some Results Obtained with Seq2Seq (contd.)

```
Original sentence in English:                          SOS i like my job very much EOS
Spanish translation in the "everything" corpus:        SOS me gusta mucho mi trabajo EOS

Translation generated by the seq2seq network:          SOS me gusta mucho trabajo EOS

Comment:    Easy case. Seq2Seq translation is passable.  One error: The seq2seq missed "mi" in "mi trabajo"
            which means "my work" or "my job" in English.

Attention depiction:
```



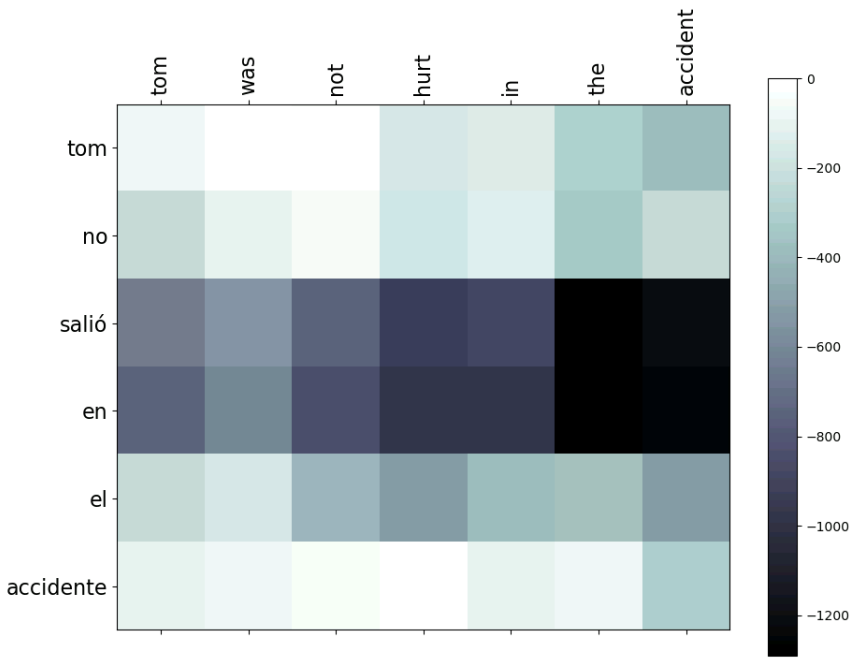
Some Results Obtained with Seq2Seq (contd.)

```
Original sentence in English:                          SOS he may have taken the wrong train EOS
Spanish translation in the "everything" corpus:        SOS él debe haber tomado el tren equivocado EOS

Translation generated by the seq2seq network:          SOS él debe haber equivocado el tren EOS

Comment:    Difficult case because it requires the use of the subjunctive case in Spanish. Seq2Seq translation
            is of pidgin quality.  Error: The seq2seq translation missed the verb "tomar" which means
            "to take".  NOTE:  The corpus translation itself is not entirely correct.  The corpus translation
            means "he should have taken the wrong train" -- which sounds stupid in English.

Attention depiction:
```



Some Results Obtained with Seq2Seq (contd.)

```
Original sentence in English:                          SOS tom was not hurt in the accident EOS
Spanish translation in the "everything" corpus:        SOS tom no salió lastimado en el accidente EOS

Translation generated by the seq2seq network:          SOS tom no salió en el accidente EOS

Comment:   Relatively easy. Seq2Seq translation is WRONG;  The Spanish sentence generated by seq2seq means
           "tom did not go out in the accident".

Attention depiction:
```



Examples of Absurd Translations Generated by the Seq2Seq Network

ENGLISH	SPANISH (produced by seq2seq)	WHY ABSURD
SOS tom you are useless EOS	SOS tom estás enfadado EOS	"enfadado" means "angry"
SOS we went to boston by car EOS	SOS nos vamos a jugar en boston EOS	"jugar" means "to play"
SOS i want to write a book EOS	SOS quiero un libro que quiero EOS	whither writing?
SOS tom flirted with mary EOS	SOS tom se ha con mary EOS	no flirtation in Spanish
SOS we can not sleep because of the noise EOS	SOS no podemos ganar el lunes de tiempo EOS	multiple reasons
SOS i must go it is getting dark EOS	SOS me gustaría ir a quedar EOS	what happened to "dark"??
SOS he wrote a lot of books on china EOS	SOS él tiene mucho de libros EOS	no mention of writing
SOS i must leave early tomorrow EOS	SOS me encanta volver a australia EOS	"australia" ???
SOS it belongs to my brother EOS	SOS era el hermano de mi hermano EOS	brother of my brother????

Outline

Seq2Seq with Pre-Trained Embeddings

- The previous section of this lecture shows how to carry out seq2seq learning when you allow the framework to learn the word embeddings on its own by using the `nn.Embedding` layer.
- In this section, I'll focus on using pre-trained embeddings for the words — but only for the words of the source language.
- The reason for using the pre-training embeddings for only one of the two languages involved has to do with the fast-memory (RAM) constraints that come into existence otherwise. My original plan was to use word2vec embeddings for the source language English and the Fasttext embeddings for the target language Spanish. The pre-trained word2vec embeddings for English occupy nearly 4GB of RAM and the pre-trained Fasttext embeddings another 8GB. The two objects co-residing in the fast memory brings down to heel a 32GB machine.

Seq2Seq with Pre-Trained Embeddings (contd.)

- Using the word2vec embeddings for the source language means that the method `sentence_to_tensor()` shown in the previous section would no longer work. Here is a new implementation that is more appropriate to using pre-trained embeddings for the source-language words:

```
def sentence_to_tensor(self, sentence, lang):
    list_of_embeddings = []                                                  ## (A)
    words = sentence.split(' ')
    if lang == "en":
        ## The corpus sentences come with prefixed 'SOS' and 'EOS' tokens. We
        ##  need to drop them for now and later insert their embedding-like
        ##  tensor equivalents:
        words = words[1:-1]                                                  ## (B)
        for i,word in enumerate(words):
            if word in self.word_vectors_en.key_to_index:                    ## (C)
                embedding = self.word_vectors_en[word]                       ## (D)
                list_of_embeddings.append(np.array(embedding))               ## (E)
        list_of_embeddings.insert(0,self.sos_tensor_en.numpy())              ## (F)
        list_of_embeddings.append(self.eos_tensor_en.numpy())                ## (G)
        sentence_tensor = torch.FloatTensor( list_of_embeddings )
    elif lang == "es":
        sentence_tensor = torch.zeros(len(words), 1, dtype=torch.long)
        for i,word in enumerate(words):                                      ## (H)
            sentence_tensor[i] = self.es_vocab_dict[word]
    return sentence_tensor
```

Seq2Seq with Pre-Trained Embeddings (contd.)

- The method shown on the previous slide treats the source and the target languages differently. When a sentence is in English, it gets the word2vec embeddings for its words by the call in line (D). If you are curious as to where the array `self.word_vectors_en` in that line comes from, here is a segment of the definition of the constructor of `Seq2SeqWithPretrainedEmbeddings`:

```
        if embeddings_type == 'word2vec':
            import gensim.downloader as genapi
            from gensim.models import KeyedVectors
            if os.path.exists(path_to_saved_embeddings_en + 'vectors.kv'):
                self.word_vectors_en = \
                        KeyedVectors.load(path_to_saved_embeddings_en + 'vectors.kv')       ## (I)
            else:
                self.word_vectors_en = genapi.load("word2vec-google-news-300")              ## (J)
                self.word_vectors_en.save(path_to_saved_embeddings_en + 'vectors.kv')
        elif embeddings_type == 'fasttext':
            import fasttext.util
            if os.path.exists(path_to_saved_embeddings_en + "cc.en.300.bin"):
                self.word_vectors_en = fasttext.load_model(path_to_saved_embeddings_en
                                                           + "cc.en.300.bin")
            else:
                os.chdir(path_to_saved_embeddings_en)
                fasttext.util.download_model('en', if_exists='ignore')
                os.chdir(".")
                self.word_vectors_en = fasttext.load_model(path_to_saved_embeddings_en
                                                           + "cc.en.300.bin")               ## (K)
```

Seq2Seq with Pre-Trained Embeddings (contd.)

- The statements that you see in lines (I) and (J) are very specific to the API provided by the `gensim` library for downloading and interacting with word2vec.

- As should be obvious from the code segment shown on the previous slide, the class `Seq2SeqWithPretrainedEmbeddings` is already set to work with the Fasttext embeddings also. Feel free to give it a try if you so wish. But note that I have not yet tested the use of Fasttext embeddings for seq2seq2.

- Apart from the implementation of the `sentence_to_tensor()` method and the specialized code you need to access the word2vec (or Fasttext) embeddings, the only other significant difference between the learnable embeddings case and the pre-trained embeddings case is the code in `EncoderRNN`. The next slide shows the definition of `EncoderRNN` for the case of pre-trained embeddings.

Seq2Seq with Pre-Trained Embeddings (contd.)

- Shown below is the implementation of `EncoderRNN` for the case of pre-trained embeddings. Note that we no longer needs the layer `nn.Embedding` in the encoder:

```
class EncoderRNN(nn.Module):
    def __init__(self, dls, s2s, embedding_size, hidden_size, max_length):
        super(DLStudio.Seq2SeqWithPretrainedEmbeddings.EncoderRNN, self).__init__()
        self.dl_studio = dls
        self.source_vocab_size = s2s.vocab_en_size
        self.hidden_size = hidden_size
        self.max_length = max_length
        self.gru = nn.GRU(embedding_size, hidden_size, bidirectional=True)

    def forward(self, sentence_tensor, hidden):
        word_embeddings = torch.zeros(self.max_length, 1,
                                      self.hidden_size).float().to(self.dl_studio.device)
        for i in range(sentence_tensor.shape[0]):
            word_embeddings[i] = sentence_tensor[i].view(1, 1, -1)
        output, hidden = self.gru(word_embeddings, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(2, 1, self.hidden_size).float().to(self.dl_studio.device)
```

- As was the case with learnable embeddings, I am again using a bi-directional GRU for the encoder.

Seq2Seq with Pre-Trained Embeddings (contd.)

- To remind the reader, if the number of words in a sentence is N and the size of the hidden state is, say, 300, the output of the encoder will emit a time-evolution of the hidden represented by a tensor of shape $[N, 600]$ in which each 600-valued tensor is a concatenation of the forward hidden and the backward hidden during the forward and the backward scan of the input sentence.
- With regard to what the encoder returns, both *output* and *hidden* are critical to the operation of the decoder. As you know well by this time, *output* is the time-evolution of the hidden in the GRU and *hidden* is the final value of the encoder hidden state. The former is needed for calculating the attention weights and the latter becomes the initial hidden for the decoder.

Training Loss vs. Iterations for the Pre-Trained Embeddings Case

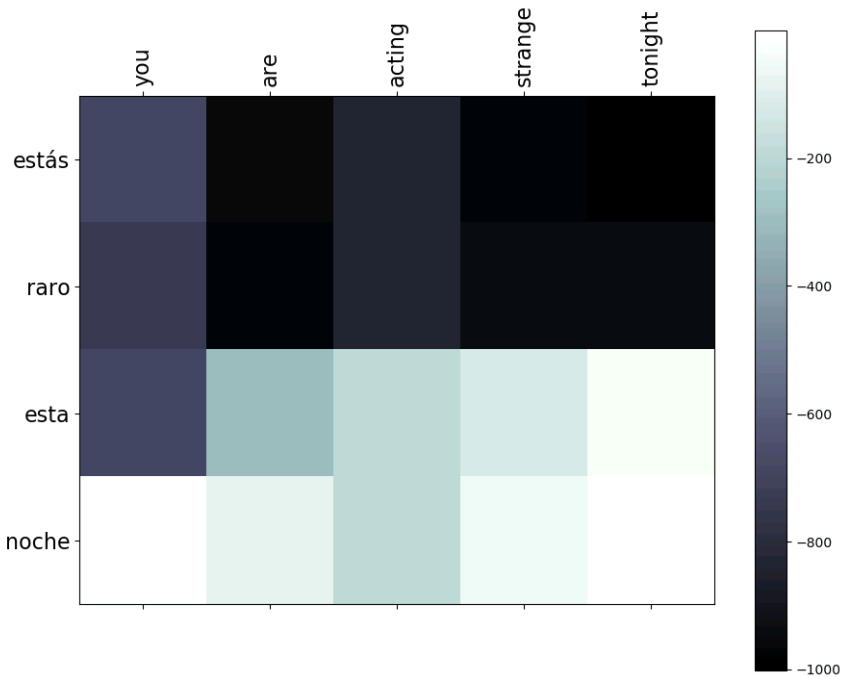
Figure: Loss vs. iterations during training over 100,000 random draws from a pool of 98,988 sentence pairs. This plot was produced by the script `seq2seq_with_pretrained_embeddings.py` in the `Examples` directory of DLStudio, version 2.0.9, with the learning rate set to 0.001.

Some Results with word2vec Embeddings for English

- As with the result obtained for the case of learnable embeddings, what you get when you use pre-trained embeddings for English is a mixed bag: You get a combination of decent, passable, and absurd translations.
- At some point in the future, I plan to analyze more carefully the performance of seq2seq when using pre-trained embeddings for both the source and the target languages. It would also be interesting to study the choice of the embeddings on the performance of seq2seq.
- The next few examples show some results for the case of using word2vec embeddings for English.
- I will not show the "absurd translations" category again because you get very similar absurdities when seq2seq is based on pre-trained embeddings.

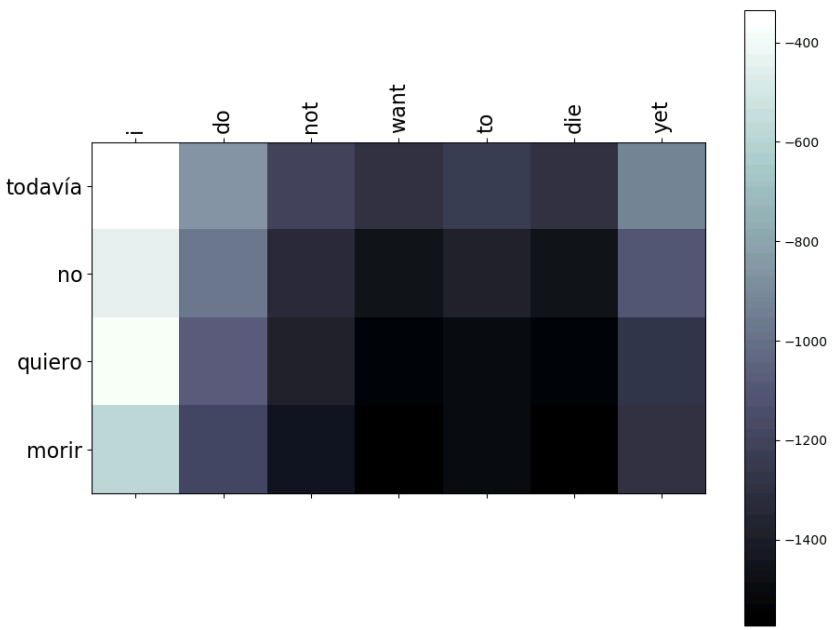
Some Results with word2vec Embeddings (contd.)

```
Original sentence in English:                          SOS you are acting strange tonight EOS
Spanish translation in the "everything" corpus:        SOS estás actuando raro esta noche EOS

Translation generated by the seq2seq network:          SOS estás raro esta noche EOS

Comment:    Easy case. Seq2Seq translation is not too bad. It conveys the essence of the sentence
            -- in a pidgin way. Error: The seq2seq missed the word "acting".

Attention depiction:
```



Some Results with word2vec for English (contd.)

```
Original sentence in English:                          SOS i do not want to die yet EOS
Spanish translation in the "everything" corpus:        SOS aún no quiero morir EOS

Translation generated by the seq2seq network:          SOS todavía no quiero morir EOS

Comment:    Easy case. Seq2Seq translation is excellent because it is the same as the
            corpus translation.  The Spanish words "aún" and "todavia" have very similar
            meanings.

Attention depiction:
```



Some Results with word2vec for English (contd.)

```
Original sentence in English:                          SOS what did tom say he needed EOS
Spanish translation in the "everything" corpus:        SOS qué dijo tomás que necesitaba EOS

Translation generated by the seq2seq network:          SOS tom dijo que lo necesitaba EOS

Comment:    Easy case. Seq2Seq translation is quite good except that it missed the
            interrogative sense.

Attention depiction:
```