EMBEDDINGS (SKIPGRAM AND CBOW) IMPLEMENTATIONS

Edit

In this post, we will take a closer look at the idea of using embeddings to encode any inputs. Specifically we will be looking at embeddings words for natural language processing (NLP) tasks. Let's get started.

EMBEDDING WEIGHTS

When we have an NLP task, we often want to represent our text inputs in a structured manner. One common strategy was to use one-hot encoding where each word/string would be represents by a N-long vector where N represents the total number of words in the vocabulary. Each index in the N-long vector would belong to a particular word and so for each word only that index will have a value of 1 while the rest are 0. Let's say there are only 3 words in our vocabulary: "I am good". Here's how we would represent each unique word with one-hot encoding.

I: [**1**, 0, 0]
am: [0, **1**, 0]
good: [0, 0, **1**]

This may seem like a good idea but think about what this will look like when our text has lots of word (ie. over a million). We will still be using a N-long vector with just one of the values filled in, creating a high dimensional yet sparse feature vector. Additionally, this method of encoding offers no ability to capture the word's meaning. In the above example there is no discernible way to understand the relationships between the words. So let's see how using embeddings weights will help us here.

When using word embeddings, we will still start with a one-hot encoded vector but we don't actually have to create it. Let's say there are 2000 unique words in our corpus, then we will have one-hot encodings of shape [NX2000] to represent the entire corpus where N is the total number of words in the text. The embedding weights are essentially a set of weights that we will multiply our inputs by to receive a new encoded input representation. The weights will be of shape [2000X300], which means each of the 2000 unique words are now represented by 300 floats. When we take the dot product of our input with these weights our new input becomes shape [NX300] which is significantly smaller than our [NX2000] and now we are able to map the relationship between words because each word is represented by floats (weights) that can be altered during training.

Now, I know I said we never have to actually make the one-hot encoded representation in the first place. If you look at the weights matrix [2000X300], each row is represented by a word. So when processing our input data, just go directly from the word index to the specific row in the weights matrix. This will save computational time and space.

TRAINING

So now how do we actually train these weights and learn the relationship between the words on our high dimensional space. There are two popular methods that we can use to train our embeddings weights: skip-gram and continuous bag of words (CBOW).

SKIP-GRAM

The idea behind skip-gram is to take in a word and predict the context words. Let's see how we would generate training data with the sentence below:

```
the quick brown fox jumped over the lazy dog
```

We need to predict context words from a target word where the context words are the words to the left and to the right of the target word. We will choose a window size of 1 to decide how far left and right to go. The training data for the sentence above will look like this:

```
(quick, the), (quick, brown), (brown, quick), (brown, fox), ...
```

Since our window size is 1, we start with the word on the 1st index because it has 1 word to the

- SEARCH -

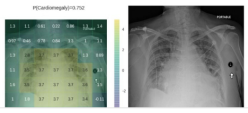
Search …

- FOLLOW ME ON TWITTER -

Tweets by @GokuMohandas

Goku Mohandas
@GokuMohandas

Your model may be performing really well by incorrectly focusing on confounding features (extraneous influencers in the data that aren't accounted for). Check out @johnrzech's post where x-ray stickers unintentionally influenced the classifications: medium.com/@jrzech/what-a…



Embed View on Twitter

- RECENT POSTS -

Update on Embeddings (Spring 2017)

Exploring Sparsity in Recurrent Neural Networks

Question Answering from Unstructured Text by Retrieval and Comprehension

Overcoming Catastrophic Forgetting in Neural Networks

Opening the Black Box of Deep Neural Networks via Information

Since our window size is 1, we start with the word on the 1st index because it has 1 word to the left and to the right. The training data is of manner (input, output) and above our last point would be (lazy, dog). The idea is to be able to train our weights so that we predict context words from target words. So when we input 'quick', we should receive 'the'. We do this across the entire training set (so for the next sentence you might see another words and expected to predict 'the'), which is fine because we want to adjust the weights for all of our words using all the context we have available.

CBOW

The continuous bag of words method allows us to create the training data for adjusting our embedding weights in a different way. Instead of predicting context words from target, we will feed in context words and try to predict the target word. So now the training data (with window of 1) will look something like ('the brown', 'quick') where we sum the embeddings of 'the' and 'brown' and expect to predict 'quick'.

COMPARISON

Though CBOW (predict target from context) and skip-gram (predict context words from target) are just inverted methods to each other, they each have their advantages/disadvantage. Since CBOW can use many context words to predict the 1 target word, it can essentially **smooth out** over the distribution. This is essentially like regularization and is offer very good performance when our input data is not so large. However the skip-gram model is more **fine grained** so we are able to extract more information and essentially have more accurate embeddings when we have a large data set (large data is always the best regularizer).

TRAINING

Regardless of the method we use to generate our training data, the end objective is to alter our randomly initialized embedding weights into meaningful weights that capture the linguistic relationship between the words. For both techniques, we will be predicting the outcome word by embedding the input word(s) and then applying softmax on them. Here's the tf code for what that might look like:

```
# Weights
W_input = tf.Variable(
          tf.random_uniform([config.VOCAB_SIZE,
          config.EMBEDDING_SIZE], -1.0, 1.0)) # embedding
W_softmax = tf.Variable(
          tf.truncated_normal([config.VOCAB_SIZE,
          config.EMBEDDING_SIZE]))
b_softmax = tf.Variable(tf.zeros([config.VOCAB_SIZE]))

# Train
embeddings = tf.nn.embedding_lookup(W_input, train_X)
loss = tf.reduce_mean(tf.nn.sampled_softmax_loss(W_softmax
                b_softmax, embeddings, train_y,
                config.NUM_SAMPLED, config.VOCAB_SIZE))
optimizer = tf.train.AdagradOptimizer(1.0).minimize(loss)
```

In the train phase, you can see we are converting our inputs train_X into the encoded embeddings using our embedding weights. We then use softmax weights and apply a sampled softmax to pick the appropriate output and then compute the mean loss. Before we take a closer look at the sampled softmax approach, let's see how the loss would be computed using traditional softmax.

SAMPLED SOFTMAX APPROACH

You have an input [NX300] which get's processed with softmax weights [300 X] which transforms our output to [N X]. We then compare our softmax results matrix with the ground truth labels [N X] (but iwth 0/1's) to compute the cross entropy loss. The computational expense, however, lies in the softmax operation.

The denominator of the softmax is the sum of all the exponential values of the vectors multiplied by the softmax weights. This term actually becomes very expensive when our vocabulary is especially large. (We can't just the max index in logits because we need to compute probabilities from softmax so we can compute loss.) The solution here is a sampled softmax approach. There are many different sampling approaching but they all try to solve the same issue: replace this denominator with an approximation for its true value. **Note:** this can only be used during training because during inference you actually need to compute the score for all the classes in order to determine the correct result.

I'd like to first give credit to **Sebastian Ruder** and **Goldberg/Levy** for their material on this topic.

We will start by representing our softmax loss as follows:

$$J = -log(\frac{e^{(h^T v'_w)}}{\sum_{w_i \epsilon V} e^{(h^T v'_{w_i})}})$$

$$J = -h^T v'_w + log(\sum_{w_i \epsilon V} e^{(h^T v'_{w_i})}$$

$$w_i \epsilon V$$

We can then compute the gradient which will result in below.

$$\nabla J = \nabla(h^T v'_w) + \frac{e^{(-h^T v'_{w_i})}}{\sum_{w_i \epsilon V} e^{(-h^T v'_{w_i})}} \nabla(-h^T v'_{w_i})$$

$$\nabla J = \nabla(h^T v'_w) - \sum_{w_i \epsilon V} P(w_i) \nabla(h^T v'_{w_i})$$

We can think of this gradient as two halves. The first term is for the actual target class and is positive. The second term is negative which acts as a negative influence for all the other classes and it is also weighted by P.

$$\sum_{w_i \epsilon V} P(w_i) \nabla(h^T v'_{w_i}) = E_{w_i \sim P}[\nabla h^T v'_{w_i}]$$

The heart of all sampling approaches is to approximate this P which will allow us to skip calculating the sum of probabilities for all the words in set V.

In order to approximate P, we will use a proposal distribution Q which will be basis for our Monte-Carlo sampling. As we can see from the main arXiv paper of this post, our proposal distribution will be something simple based off of the unigram distribution of our training data.

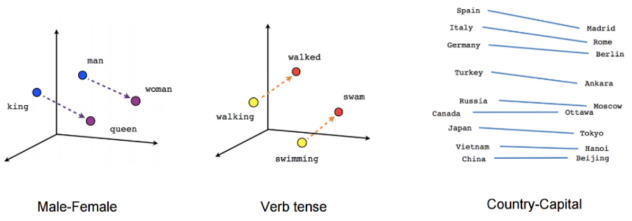
Specifically focusing on negative sampling, the sampled softmax can be represented by:

$$P = \frac{e^{(h^T v'_{w_i})}}{e^{(h^T v'_{w_i})} + kQ(w)}$$

where k is the number of terms we will to sample with (instead of all the words). The higher k is, the better of an approximation Q will be for P. There are many variations to sampled softmax but these general principles hold.

LINGUISTIC MAPPINGS

So how do know these trained weights even captured the linguistic relationships in our words? Well, we can actually map out the relationship with distance and capture some truly amazing patterns. Here are a few examples from Google's Word2Vec.



Male-Female | Verb tense | Country-Capital

In fact, even from our simple implementation in the repo below with a Harry Potter text, we can map closest words (based on cosine similary of the emebdding vectors) and see which words associate most with eachother.

```
Close to see : ['ask', 'explain', 'realized', 'teach', 'surprise']
Close to harry : ['guess', 'ollivander', 'saturday', 'glancing', 'bowed']
Close to we : ['i', 'they', 'you', 'everybody', 'jus']
Close to have : ['ve', 'take', 'speak', 'manage', 'mother']
Close to ron : ['sparkling', 'twitched', 'shout', 'furiously', 'hole']
```

Look's like we captured Ron's personality really well 🙂

TRAINING FROM SCRATCH, PRETRAINING, WORD2VEC/GLOVE

For example of word embeddings, we have many options. Regardless of our task (learning embeddings, text classification, NMT, etc.) we can always choose to just feed in random embedding weights and train the network end-to-end. In this case, the word embeddings get trained along with the overall main task. The advantage is that our embeddings can be trained specifically for this task but the disadvantage is that they are trained from scratch and the relationship between the embeddings may not be similar to what you may find with a much larger general corpus.

The second option is to pretrain the random embeddings first under a schema such as skip-gram or CBOW and then freeze those weights and conduct our main task. This allows us to once again learn the true linguistic relationship in our text and then apply those directly to our task.

The third option is to use official trained embeddings which have been trained on massive corpus (ie. millions of wikipedia articles or Twitter). The most prominent embeddings are from Google's **word2vec** and Stanford's **GloVe**, both of which have their own advantages and disadvantages but for most NLP tasks, either will suffice.

FUTURE WORK

The building upon concepts behind embeddings have not reached completion and there is plenty of active research surrounding them. One aspect I would like to one area which is based on context-dependent word embeddings which I explained in this **post**. We know that many words can have multiple meanings depending on the context. We need to be able to alter the embedded representation depending on the context to have the most accurate representation of the input word. In this paper, they took some trained word embeddings (300D) and used PCA to convert to a 2D representation. Here's what that looked like for two of the words:

Word x'	Axis	Nearest Neighbours
notebook	1	diary notebooks (notebook) sketchbook jottings
	2	palmtop notebooks (notebook) ipaq laptop
power	1	powers authority (power) powerbase sovereignity
	2	powers electrohydraulic microwatts hydel (power)

Table 1: The placements of the word embedding vectors along the principals axes in the local charts of "notebook" and "power". In the case of "notebook", it is clear that the first axis corresponds to different types of books for writing a note, while the second axis corresponds to portable computing devices. In the case of "power", the first axis corresponds to political or social authority, while the second axis to physical energy.

These different definitions for the same word were identified to be on two separate axis. From this, we can choose to mask one of the axis in order to get the accurate representation that represents the meaning of the word in a particular context.

- contextualizing the word embeddings essentially masks these extra dimensions that do not represent the meaning of the word in a particular context.

$$c^x = \frac{1}{T}\sum_{t=1}^{T}\text{MLP}(x_t)$$

$$MLP : \mathbb{R}^E \rightarrow \mathbb{R}^C$$

$$encoder : x_t = x_t \odot sigmoid(W_x c^x + b_x)$$

$$decoder : y_t = y_t \odot sigmoid(W_y c^x + b_y)$$

- Contextualizing the original word embeddings involves the following:
  1. context: average of the nonlinearly transformed source word embeddings.
  2. We take this context and use it to the mask the embedded inputs we put into the encoder and decoder. Note: for the decoder, the word embeddings coming into play when we are feeding in the previously predicted word into the decoder as the input. This goes through an embedding layer similar to the encoding layer.
- The masking involves using an element wise sigmoid operation that produces a float between [0,1] which we multiply by the original embeddings.

There are a few more particulars to the paper such as the fact that we only apply the context when embedding the deocder outputs as the input to the next decoder state but you can find out the more specific details in the paper and in my post.

CODE:

GitHub Repo (Updating all repos, will be back up soon!**)**

Posted in: *Uncategorized*
Tagged: *Tutorials*

← *Learning to Protect Communications with Adversarial Neural Cryptography*

Recurrent Neural Network (RNN) – Part 5: Custom Cells →

3 THOUGHTS ON "EMBEDDINGS (SKIPGRAM AND CBOW) IMPLEMENTATIONS"

CBOW) IMPLEMENTATIONS

Pingback: Recurrent Neural Network (RNN) – Part 4: Attentional Interfaces – The Neural Perspective Edit

GOKUMOHANDAS August 9, 2017 at 2:40 pm EDIT REPLY →

You can make the input sparse but if we look at the one-hot encoded results as is, they are a sequence of dense vectors.

★ Like

RAHUL KUMAR March 25, 2018 at 7:49 pm EDIT REPLY →

According to Mikolov, Skip gram works better than CBOW in case of small amount of training data while the article seems to support the opposite.

★ Like

LEAVE A REPLY

Enter your comment here...