Topic 4: Hidden representations

The hidden layers of a neural network are typically very difficult to interpret and for this reason we say that neural networks are black box models, that is, you can't inspect what it is that they're doing internally to come up with their output. However, whether we understand them or not, one of the advantages of deep learning is that they are useful on their own. By extracting the activations of hidden layers, we obtain feature vectors that represent the input. These feature vectors can be used to cluster the inputs, can be transferred to other machine learning models (the information in the feature vector is more 'accessible' than in the raw input), or even shared between different models that are being trained at the same time. In this topic, we will see an example of each of these cases.

1. Word embeddings

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/04_-_Hidden_representations/01_-_Word_embeddings.py

How can you feed a word or several words to a neural network as an input (say, to predict the next word in a partial sentence)? Neural networks can only work with numbers. Will you express the individual letters as Unicode numbers? That would imply that Unicode 97 ('a') is equally similar to Unicode 98 ('b') as it is to Unicode 96 ('`'), which would create a built-in bias into the input which the neural network needs to unlearn.

A better way to convert words into numbers that can be fed to a neural network is to replace every word with a one-hot vector, which is a vector consisting of zeros everywhere except for a single solitary 1, such as [0, 0, 1, 0, 0, 0]. If each unique word is replaced with a unique one-hot vector such that different words will have different positions for the solitary 1 in the vector, then the word vectors would all be equally different from each other and there would not be any bias introduced to the neural network.

Each possible unique word is associated with a unique index number which would be the position of the 1 in the one-hot vector. For example, the word with index 0 would have the one-hot vector [1, 0, 0, …], the word with index 1 would have the one-hot vector [0, 1, 0, …], etc. The size of the one-hot vectors must be equal to the amount of different words that can be used. In order to keep the number of words finite, we choose a fixed vocabulary made from words found in the training set.

Although this is not implemented in the given script, in practice we typically exclude rare words as it's difficult to infer the meaning of a rare word. For this reason, we only use the most frequent n words in the training set for the vocabulary, where n is the desired vocabulary size (larger is better but also more memory consuming due to the one-hot vector sizes). Any word that is not part of the vocabulary is replaced with a pseudo-word called an unknown token. This allows us to use words that were not in the vocabulary when using the neural network after training at the cost of just one extra word.

In the given script we train a neural network to predict the word that is likely to be in between two other words. For example, between 'the' and 'barks' there is likely to be the word 'dog'. This is done by feeding in the one-hot vectors for 'the' and 'barks' into the neural network, concatenating the vectors together, passing the concatenated vector through the neural network, and finally using a softmax to predict the word in between.

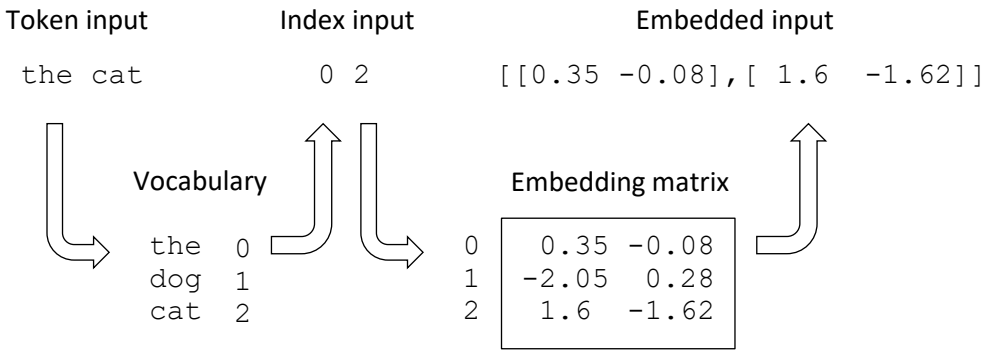
2. Word embeddings 2

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/04_-_Hidden_representations/02_-_Word_embeddings_2.py

One-hot vectors are easy to understand but are also a waste of time to the neural network. If you think about it, multiplying a one-hot vector by a weight matrix is equivalent to picking out a row from the weight matrix. For example:

$$(0 \quad 1 \quad 0)\begin{pmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{pmatrix} = (3 \quad 4)$$

This means that the layer does not 'transform' a vector into another vector but just maps an index into vector. Even the bias does not add any information to the mapping since you can just add the bias vector directly to every row of the weight matrix. One-hot vectors are also very memory inefficient when stored in the training set. For these reasons, we avoiding performing a full matrix multiplication in the first layer and just pick a row out of a matrix called an embedding matrix. The embedding matrix would have a number of rows equal to the vocabulary size and a number of columns equal to the desired word vector size. A special Tensorflow function is then used to pick rows from this matrix and concatenate them together when given a list of word indexes. Here's an example of how to think of the embedding matrix:



In the previous script, we first concatenated the one-hot vectors of two words together and then passed them through a neural network layer which gave us a hidden layer activation vector of size 4. This is equivalent to just embedding each word separately into a vector of size 4 and then adding the two vectors together, followed by an application of the sigmoid function. In this script we will simplify this process by instead embedding the two words into vectors of size 2 and then concatenating the two vectors into a vector of size 4. This is not equivalent to the previous process by it is much more memory efficient and just as expressive.

We usually do not apply an activation function to the embedded vectors and instead allow the number in the word vectors to grow without bound. This does not reduce the expressive power of the neural network as there is no transformation being applied to the input vector in the first layer.

An interesting thing about the embedding matrix is that, after training, the numerical similarity between two word vectors tends to give hints on the similarity between the two words, provided that there is enough data to train on. In fact, neural networks are constructed and trained just to be able to extract this embedding matrix as it is very useful information about the words. An example of this is Word2Vec,

which is a neural network that is trained to predict the middle word from its neighbouring words in a sequence of words taken from a corpus, or predict the surrounding word from the middle word. See this paper for more information on Word2Vec. In the given script, we plot the row vectors of the embedding matrix and label each dot with the word it represents.

3. Transfer learning

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/04_-_Hidden_representations/03_-_Transfer_learning.py

Of course, we don't just collect word vectors for linguistic reasons, we also do it to pass on the embedding matrix into other neural networks. If you have to train a neural network to perform a task such as part-of-speech tagging (classify if a word is a noun, a verb, etc.) then you are likely to have a relatively small dataset of tagged words. This makes it difficult for the neural network to infer the meaning of the words. On the other hand, it is very easy to find lots and lots of unlabelled text to train a neural network to predict the middle word from a sequence of words. The embedding matrix learned by the second neural network is likely to be very high quality and it would be nice to copy over the knowledge encoded in it to other neural networks.

To do this we can initialise the embedding matrix of the part-of-speech tagger (called the target model) with the embedding matrix extracted from the pre-trained word predictor (called the source model). The embedding matrix can then either be kept as is during training (freezing the parameters) or can be further optimised during training (fine-tuning the parameters). Freezing is useful to avoid overfitting whilst fine-tuning is useful to make the representation specialise to the new task.

This process of transferring parameters between tasks or data domains is called transfer learning and is one of the most important fields in deep learning as it allows you to reuse pre-trained neural networks to train new ones. It also makes sense to train a model starting from existing knowledge rather than from randomly set parameters as that is how we usually learn (you wouldn't start from zero every time you wanted to learn something new).

4. Multi-task learning

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/04_-_Hidden_representations/04_-_Multi-task_learning.py

Rather than extracting the parameters from a pre-trained neural network, we can instead train one neural network on two datasets. Of course, the neural network would need separate inputs and outputs for each task, but the hidden layers can be shared, that is, both tasks make use of the exact same parameters internally (although not all parameters need to be used for both tasks). This means that a particular hidden layer would be trained to create useful features for two tasks at once rather than for a single task, which would make it harder for the model to overfit since it will need to overfit on two different tasks.

This is called multi-task learning. One very interesting use of multi-task learning was in a single neural network that performs image object recognition, image description generation, text translation, part-of-speech tagging, and speech recognition. Another example is a single neural network that can translate between many different languages at once.