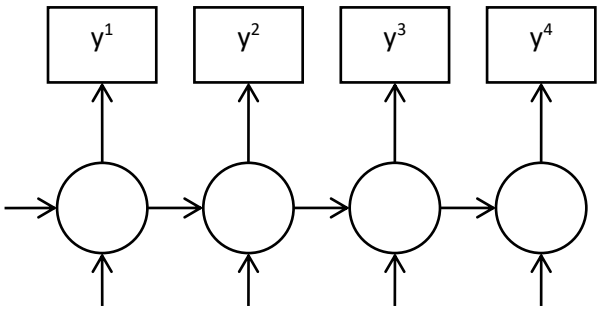
Topic 8: Sequence generators

One of the more interesting uses of neural networks is in sequence generation rather than classification. This is usually used to generate text although any sequence can be generated.

1. Deterministic generators

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/08_-_Sequence_generators/01_-_Deterministic_generators.py

The simplest way to generate a sequence is to use an RNN that always receives the same input at every time step (use a sequence of zero vectors for example). All the intermediate states (which would still be different at every time step even though the inputs are the same) can then be used to classify a sequence item at every time step. For example, each intermediate state would be sent to a softmax layer in order to classify a word, resulting in a different word being selected at every time step. By changing the length of the blank input sequence you would be changing the maximum length of the output sentence. Here's an illustration:



In the code, the constant input at each time step is a learned vector, just like the initial state. It is then replicated for each time step needed. The number of time steps needed is an integer input. The output of the neural network is a sequence of softmaxes which are then reduced to their argmax in order to get the position of the maximum element in each softmax. The neural network is trained to generate an alternating bit.

2. One output at a time

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/08_-_Sequence_generators/02_-_One_output_at_a_time.py

One thing that might be confusing you about using RNNs to generate a sequence is how you have to generate a sequence from beginning to end rather than one output at a time. Shouldn't RNNs be able to work sequentially for any number of steps without needing to establish the number beforehand? This is true, but it requires that you get the RNN's state and supply it as an input rather than allow the dynamic_rnn function to manage that state for you.

The way it works is that, given a neural network, take the RNN and supply it with a state vector. At first the state vector would be the initial state. The RNN cell would then take in the supplied state together with the fixed input vector and return a new state. This new state is then passed to the softmax layer in order to predict the next output and is also returned to the user. The user reads out the new state vector

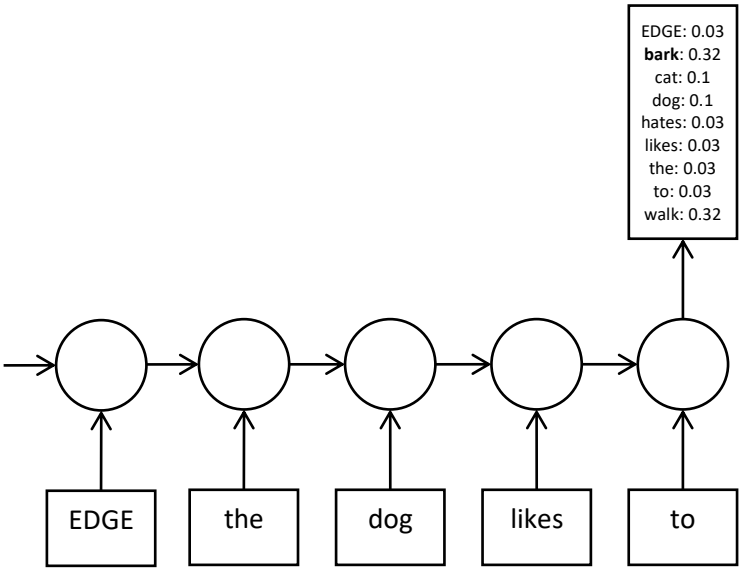
and predicted output, and returns the new state vector as an input. This process can keep going indefinitely.

One important point to note is that this is only possible once the neural network is trained. You cannot train a neural network's RNN in this way as the RNN needs to know all of the neural network's sequence output in order to make sure that it learns to create a state that correctly evolves with every time step. This means that the state cannot be an input but must be learned and in order to learn the state, the RNN must be trained to generate the entire sequence at once, as has been done in the previous code.

3. Neural language models

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/08_-_Sequence_generators/03_-_Neural_language_models.py

The above method of generating a sequence is called a deterministic generator because it always generates the same sequence and only the length of the sequence can be varied. If you want to instead be able to generate different sequences (such as different sentences) then you will need to construct a neural language model. A neural language model is a neural network that takes in a prefix of a sentence (an initial part of a sentence) and returns a probability for each possible next word. You can then pick one word at random based on their probabilities. Here's an example of a neural language model predicting the next word after the prefix "the dog likes to…" when the vocabulary consists of the words bark, cat, dog, hates, likes, the, to, and walk:



The edge tokens are used to indicate the beginning and end of the sentence. Following the prefix, we randomly pick one of the words using the probabilities given by the neural network (which sum to 1). The most likely word to pick is either 'bark' or 'walk' but even one of the least probable next words can be selected, just very rarely. If after selecting a word we attach it to the end of the prefix, we can then select the next word after that and so on until we have a complete sentence.

The edge token allows us to pick a first word in the sentence by using the edge token as a prefix. If every sentence starts with the edge token then its probability of being selected is 100% which means that it

doesn't need to be predicted. It also allows us to be able to terminate the sentence by checking if the sampled word is the edge token, indicating that we've selected to end the sentence.

The randomness in the generation process allows us to generate a different sentence each time. This is called sequence sampling and the probability of sampling a particular sentence is equal to the probability of that sentence according to the neural network (which is found by multiplying the probabilities of each word in the sentence).

The basic way you train a neural language model is by taking a corpus of sentences and breaking up each sentence into all its possible prefix-next word pairs. For example, given the sentence "the dog likes to bark", we break it up into the following:

Prefix	Next word
EDGE	the
EDGE the	dog
EDGE the dog	likes
EDGE the dog likes	to
EDGE the dog likes to	bark
EDGE the dog likes to bark	EDGE

The whole neural network can be seen as just a sequence classifier that takes in a prefix of a sentence and classifies which word can follow the prefix using a softmax. You might be confused on how to train such a neural network when there are multiple possible next words given the same prefix. For example, there are many possible first words that can be used after the edge token. How can a classifier classify multiple classes? An example is shown below:

Input	Target output
EDGE	the
EDGE	a

The neural network's optimiser does not break in such a situation since its goal is not to reach 100% accuracy but only to find the minimum possible error. In this case the minimum possible error would be to give 50% of the total softmax probability to 'the' and 50% of the probability to 'a', with all the other words in the softmax having a probability of 0%. If there were more words that can follow the edge token then the probability will be evenly divided between all of them.

What happens if, for the same input, there is a particular target output that appears more than once? For example, multiple sentences can start with the word 'the', resulting in a training set that looks like this:

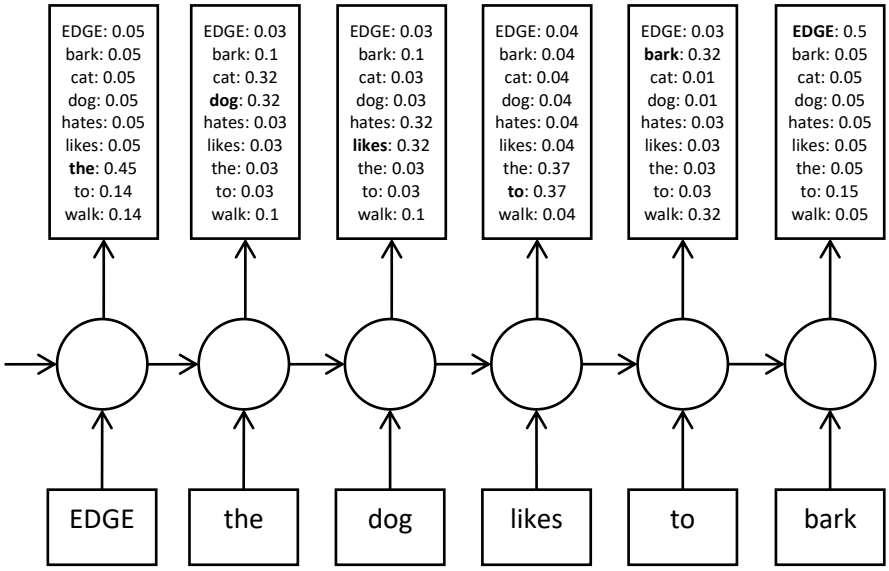
Input	Target output
EDGE	the
EDGE	the
EDGE	a

In this case, the optimiser will make the neural network give the word 'the' a probability that is twice as large as that given to 'a', that is, 'the' will be 66% and 'a' will be 33%. This makes the neural network give probabilities that are proportionate to the frequency of the words, making them reflect the actual probability of finding the given word after the given prefix in the corpus training set.

4. Faster neural language models

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/08_-_Sequence_generators/04_-_Faster_neural_language_models.py

The previous neural language model shown requires that every sentence gets broken down into its prefixes which results in an inflated dataset and slow training time. This can be solved by instead leaving the sentences whole in the training set but instead of training the neural network to predict a single next word, you train it to predict every next word after every prefix. This can be done by using the outputs vectors of the dynamic_rnn function rather than the final state which gives a representation for each prefix in the sentence. Here is an illustration:



The training set now looks more compact:

Prefixes	Next words
EDGE the dog likes to bark	the dog likes to bark EDGE

Rather than being a table of prefixes and their next word, the training set is now a mapping from the whole sentence to the same sentence but shifted by one word. The input is the sentence with the edge token at the beginning, allowing the first edge token to be used as a prefix to predict the first word, whilst the output sentence is the same sentence but with the edge token at the end, allowing the longest prefix to predict the end of sentence. In this way, the neural network is trained to predict all the next words in the sentence at once. Of course, during testing the neural network still needs to predict each word one at a time and is used in exactly the same way as the previous neural language model. This modification is only applicable for training.

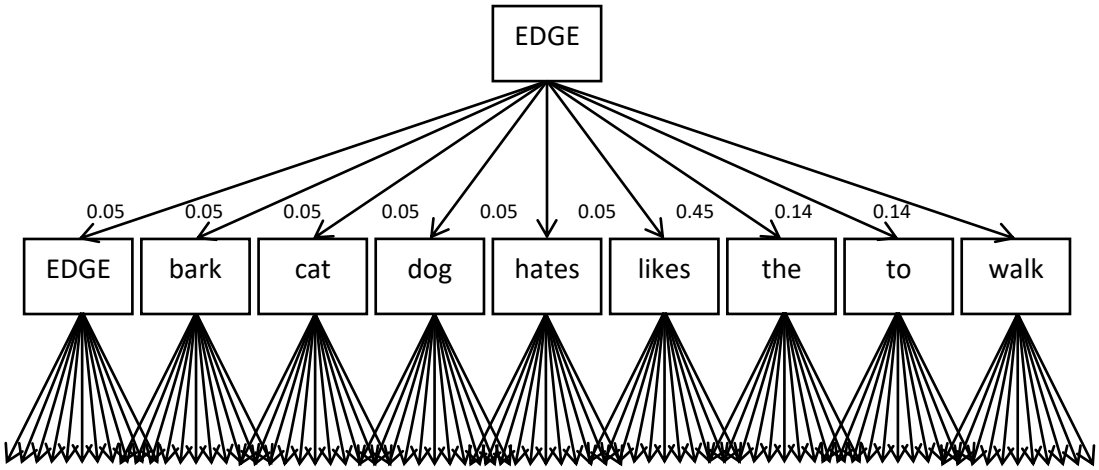
5. Beam search

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/08_-_Sequence_generators/04_-_Beam_search.py

Up to now, we have been generating sentences by randomly sampling words, which allows us to generate different sentences each time. Sometimes this is not desirable. Sometimes we just want to generate the most probable sentence and only generate that one sentence.

The simplest way to try to generate the most probable sentence is to use a greedy search, which is when we always pick the most probable word predicted by the neural network. By doing this, we are selecting the most probable word given the prefix but this does not necessarily result in the most probable sentence as the most probable sentence might not consist of the most probable word at every position.

A better way to find the most probable sentence is to use tree searching algorithms in order to find the most probable path in the probability tree of the language model. A language model can be used to make a probability tree with the root being the edge token, the children of each node being the entire vocabulary of words, the links between nodes contain the probability of the child node given all its ancestor nodes, and the leaves being all edge tokens. Here is an illustration:



What we want to do is find a path from the root to a leaf which has the maximum product of probabilities. We know when to stop looking when we find a complete path whose probability is greater than that of all other complete or incomplete paths that are of the same length of shorter. What this means is that, if we found a complete four-word sentence but there are other four-word prefixes we saw that have a greater probability than the sentence, then we cannot stop looking just yet. On the other hand, if a complete five-word sentence was found that has a greater probability than all other prefixes of five words or less, then that five-word sentence is the maximum probability sentence. This is because the probability of a prefix never increases as the number of words increases since multiplying a probability by another probability creates a smaller probability. So if all prefixes found up till now have a smaller probability than a complete sentence found, making the other prefixes longer will only make their probabilities even smaller.

We can exploit this termination condition in order to search all possible paths in the tree using something like breadth first search but that would take too long if the vocabulary is large. Instead, we do an

approximate search called beam search in order to search only the most promising paths. We choose a beam width (such as 3 or 5) and only explore the 'beam width' most probable prefixes found up to now. This eliminates a lot of search time but also makes it less likely that the most probable sentence will be found.

A beam width of 1 makes the beam search algorithm equivalent to greedy search. The only reason why we were just finding the maximum probability word in greedy search rather than finding the maximum probability prefix, as we do in beam search, is because all the prefixes considered are the same except for the new token. This means that we only need to find the maximum probability token in order to find the maximum probability prefix.