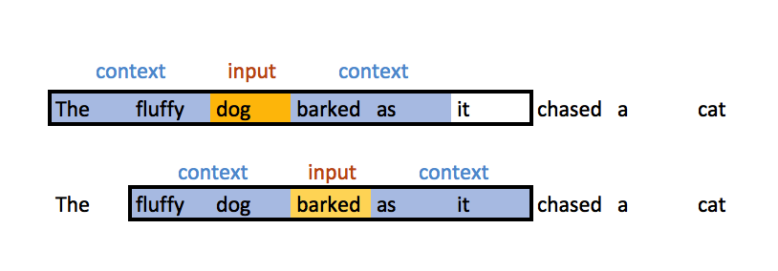
Word2Vector

**Word2Vector -** Traditional approaches to NLP, such as one-hot encoding and bag-of-words models (i.e. using dummy variables to represent the presence or absence of a word in an observation, i.e. a sentence). In essence, traditional approaches to NLP such as one-hot encodings do not capture syntactic (structure) and semantic (meaning) relationships across collections of words and, therefore, represent language in a very naive way. For example, a one-hot encoding cannot capture simple relationships, such as determining that the words "dog" and "cat" both refer to animals that are often discussed in the context of household pets. Such encodings often provide sufficient baselines for simple NLP tasks (for example, email spam classifiers), but lack the sophistication for more complex tasks such as translation and speech recognition.

The beauty of representing words as vectors is that they lend themselves to mathematical operators. For example, we can add and subtract vectors — the canonical example here is showing that by using word vectors we can determine that: **king - man + woman = queen.** In other words, we can subtract one meaning from the word vector for king (i.e. maleness), add another meaning (femaleness), and show that this new word vector (king - man + woman) maps most closely to the word vector for queen. In simpler terms, a word vector is a row of real-valued numbers (as opposed to dummy numbers) where each point captures a dimension of the word's meaning and where semantically similar words have similar vectors. This means that words such as wheel and engine should have similar word vectors to the word car (because of the similarity of their meanings), whereas the word banana should be quite distant. Put differently, words that are used in a similar context will be mapped to a proximate vector space

The numbers in the word vector represent the word's distributed weight across dimensions. In a simplified sense, each dimension represents a meaning and the word's numerical weight on that dimension captures the closeness of its association with and to that meaning. Thus, the semantics of the word are embedded across the dimensions of the vector.



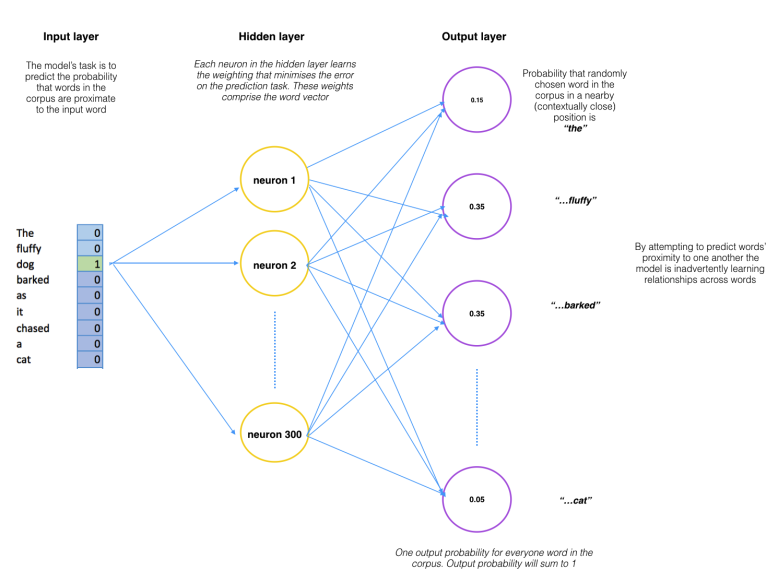


One of the fascinating things about word vectors created by word2vec models is that they are the side effects of a predictive task, not its output. In other words, a word vector is not predicted, (it is context probabilities that are predicted), the word vector is a learned representation of the input that is used on the predictive task — i.e. predicting a word given a context.

A word2vec model, therefore, accepts as input a single word (represented as a one-hot encoding amongst all words in the corpus) and the model attempts to predict the probability that a randomly chosen word in the corpus is at a nearby position to the input word. This means that for every input word there are n output probabilities, where n is equal to the total size of the corpus. The magic here is that the training process includes only the word's context, not all words in the corpus. This means in our simple example above, given the word "dog" as input, "barked" will have a higher probability estimate than "cat" because it is closer in context, i.e. it is learned in the training process. Put differently, the model attempts to predict the probability that other words in the corpus belong to the context of the input word. Therefore, given the sentence above ("The fluffy dog barked as it chased a cat") as input a run of the model would look like this:

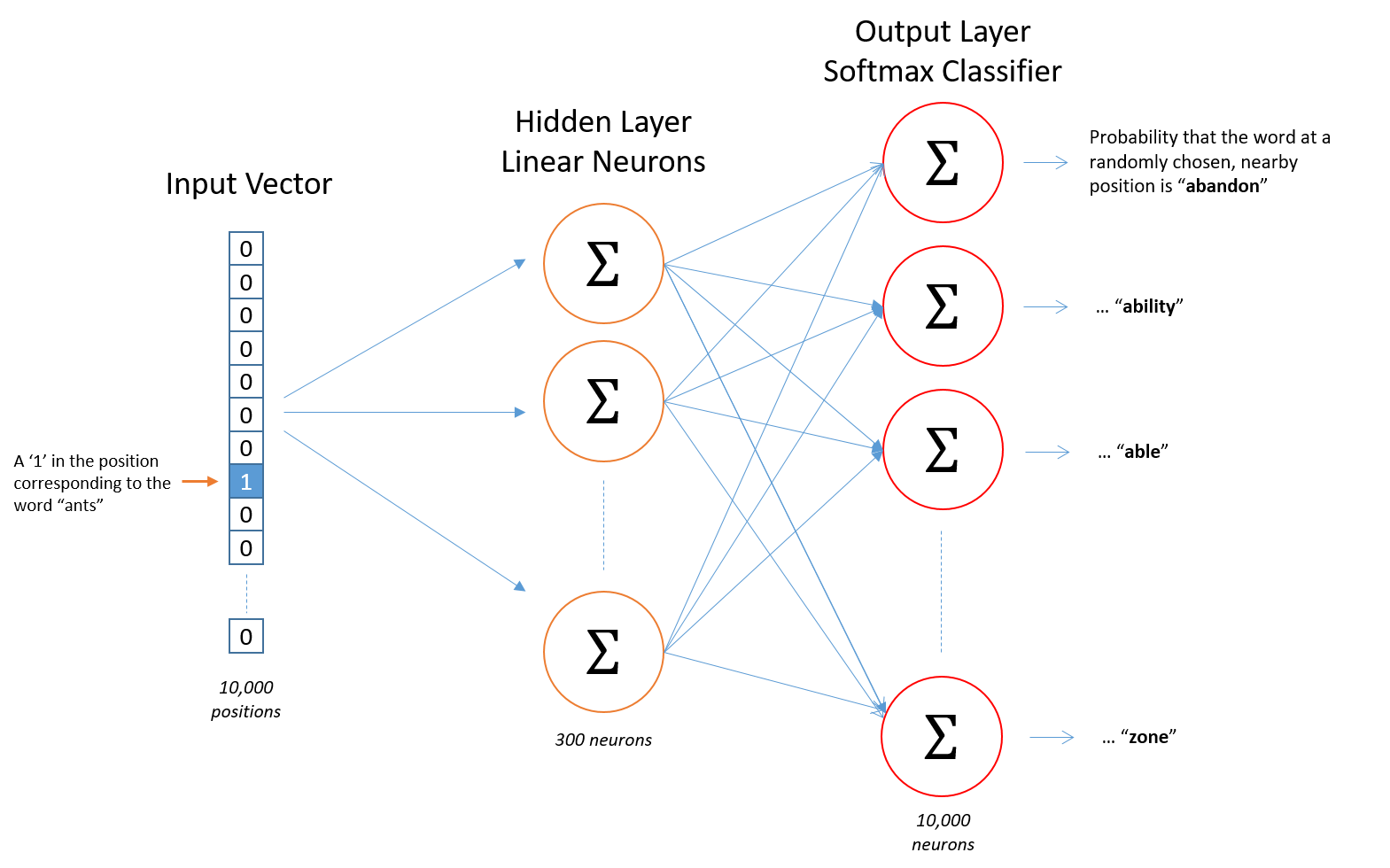
The value of going through this process is to extract the weights that have been learned by the neurons of the model's hidden layer. It is these weights that form the word vector, i.e. if you have a 300-neuron hidden layer, you will create a 300-dimension word vector for each word in the corpus. The output of this process, therefore, is a word-vector mapping of size n-input words \* n-hidden layer neurons.

The word vector is the model's attempt to learn a good numerical representation of the word in order to minimize the loss (error) of its predictions. As the model iterates, it adjusts its neurons' weights in an attempt to minimize the error of its predictions and in doing so, it gradually refines its representation of the word. In doing so, the word's "meaning" becomes embedded in the weight learned by each neuron in the hidden layer of the network.



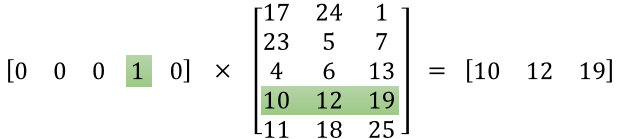
Let’s take another example. We first build a vocabulary of words from our training documents–let’s say we have a vocabulary of 10,000 unique words. We’re going to represent an input word like “ants” as a one-hot vector. This vector will have 10,000 components (one for every word in our vocabulary) and we’ll place a “1” in the position corresponding to the word “ants”, and 0s in all of the other positions.

The output of the network is a single vector (also with 10,000 components) containing, for every word in our vocabulary, the probability that a randomly selected nearby word is that vocabulary word. Here’s the architecture of our neural network.



**The Hidden Layer -** For our example, we’re going to say that we’re learning word vectors with 300 features. So the hidden layer is going to be represented by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for every hidden neuron).

Now, you might be asking yourself–“That one-hot vector is almost all zeros… what’s the effect of that?” If you multiply a 1 x 10,000 one-hot vector by a 10,000 x 300 matrix, it will effectively just select the matrix row corresponding to the “1”. Here’s a small example to give you a visual.



Finally, in order to get the outputs to sum up to 1, we divide this result by the sum of the results from all 10,000 output nodes.