1. Embeddings

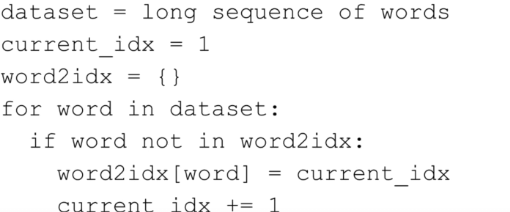
* Dealing with text:
* Text is also sequence data >< not continuous
* X(t)’s are not numerical
* Can’t do regular RNN
* One-hot encoding?
* Create an array of size equal to the number of words in the English language
* Map each word to an index
* A sequence of T words, each word get a one-hot encoded vector of size V
* One sentence – T x V matrix
* Same generic shape as a T x D matrix
* Problem:
* Some English-language datasets have ~ 1 million possible tokens / words
* Large input and large input-hidden weight matrix
* We want our input features to have some structure
  + ML is nothing but a geometry problem
  + This rule relies on the fact that there’s some geometrical structure in the data – that data vectors in the same class are probably near each other
* Problem with one-hot encoding:
* Take any 2 one-hot encoded vectors
* Euclidean distance is sqrt(2)
* All words are an equal distance apart – useless
* Better solution: Embeddings
* Assign each word to a D-dimensional vector (not one-hot encoded)
* How are these vectors found?
* Coding trick: Suppose we multiple a one-hot encoded vector with a matrix
* One-hot encoding an integer k and multiplying that by a matrix is the same as selecting the k’th row of the matrix
* Index the weight matrix (W[k]) – more efficient
* This is exactly what the Embedding layer does
* Tensorflow embedding
* Step 1: convert words into integers
  + [I, like, cats] -> [50, 25, 3]
* Step 2: use integers to index the word embedding matrix to get word vectors for each word
  + [50, 25, 3] -> [[0.3, -0.5], [1.2, -0.7], [-2.1, 0.9]]
  + T length array -> T x D matrix
* How are the weights found?
* We know intuitively that the word vectors must have some useful structure
* Same story as with convolutional filters
* These weights are found automatically with gradient descent when you call model.fit()
* Pre-trained word vectors
* Sometimes, we use pre-trained word vectors (trained using some other algorithm)
* We freeze (fix) the embedding layer’s weights, so only the other layers are trained when we call model.fit()



* Extra reading: word2vec, GloVe
* Summary:
* It’s not possible to multiple a word by a weight matrix, there is no concept of multiplying words
* We must turn words into some kind of numerical representation first
* One-hot encoding – fine first guess
  + Too much space
  + Not geometrically useful (cat is just as close to feline as to airplane)
* Better: convert words to unique integers
* Use those integers to index a weight matrix (embedding matrix)
* Much faster than matrix multiplication
* Each input sentence thus becomes a T x D sequence which we know an RNN can accept
* The embedding is trained like any other layer -> just need to call model.fit()
* Alternative methods for training embedding: word2vec, GloVe

1. Code preparation

* Turn a sequence of words into an acceptable format such that they can be converted into a T x D matrix of numbers
* Words to integers:
* Before converting words to vectors, we must first convert them into integers (indexes into the word embedding matrix)
* Words -> integers -> vectors
* Need a mapping from word -> integer
* Pseudocode:

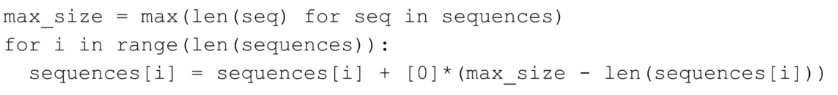


* Why start from 1 instead of 0?
  + In Tensorflow, we have constant sequence length -> all the data can fit into an N x T x D array
  + T – maximum sequence length -> any shorter sentences require padding
  + We use 0 for padding
* Words to integers in Tensorflow
* In Tensorflow, we can use a function to do that for us
* Text is not formatted as a sequence of words
* Just a string
* We want a list of strings where each string is a single word -> tokenization
* Now we have to do: string -> tokens -> integers -> vectors
* Tensorflow Tokenizer
* Converts strings containing multiple words into a list of tokens
* Converts list of tokens into list of integers

A black text on a white background

Description automatically generated

* For a Google-sized dataset, we may have 1 mil+ tokens, most of which are extremely rare and probably useless for classification
* We can assign all these to some generic <RARE> or <UNKNOWN> token
* Padding:
* The sequences still all have different lengths -> padding



* Padding in Tensorflow:
  + Input: list of N lists of integers (max length = T)
  + Output: N x T matrix
  + Or an N x MAXLEN matrix depending on which is smaller

****

* Truncating: Truncating at the beginning or end?
* Setting the truncating argument to ‘pre’ or ‘post’

**A black text on a white background

Description automatically generated**

* Padding at the beginning or end?
* Setting the padding argument to ‘pre’ or ‘post’
  + E.g.: Spam classifier: Might want padding at the beginning – RNNs have trouble learning long distance patterns
  + E.g.: neural machine translation: suppose the target language sentence is longer than the input language sentence, how can RNN predict output words if the only input it has seen so far is a bunch of 0s?
* Recap:
* Data – list of strings
* Each string is one or more sentences
* Tokenizer converts each string into a list of integers
* Each integer corresponds to a unique word, which can then be used to index a weight matrix (an embedding)
* pad the sequence (of integers) to get a 2D array of shape N x T
* X[n, t] is the word (index) that appears in document n, time step t
* Pass through an embedding layer to get an N x T x D tensor (convert each word into a vector)
* Pass into an RNN
* A full RNN for text classification

**A screenshot of a computer

Description automatically generated**