# Embeddings and Word2Vec

When you're dealing with words in text, you end up with tens of thousands of word classes to analyze; one for each word in a vocabulary. Trying to one-hot encode these words is massively inefficient because most values in a one-hot vector will be set to zero. So, the matrix multiplication that happens in between a one-hot input vector and a first, hidden layer will result in mostly zero-valued hidden outputs.

A close up of text on a white background

Description automatically generated

To solve this problem and greatly increase the efficiency of our networks, we use what are called **embeddings**. Embeddings are just a fully connected layer like you've seen before. We call this layer the embedding layer and the weights are embedding weights. We skip the multiplication into the embedding layer by instead directly grabbing the hidden layer values from the weight matrix. We can do this because the multiplication of a one-hot encoded vector with a matrix returns the row of the matrix corresponding the index of the "on" input unit.

A close up of a clock

Description automatically generated

Instead of doing the matrix multiplication, we use the weight matrix as a lookup table. We encode the words as integers, for example "heart" is encoded as 958, "mind" as 18094. Then to get hidden layer values for "heart", you just take the 958th row of the embedding matrix. This process is called an **embedding lookup** and the number of hidden units is the **embedding dimension**.

A screenshot of a cell phone

Description automatically generated

There is nothing magical going on here. The embedding lookup table is just a weight matrix. The embedding layer is just a hidden layer. The lookup is just a shortcut for the matrix multiplication. The lookup table is trained just like any weight matrix.

Embeddings aren't only used for words of course. You can use them for any model where you have a massive number of classes. A particular type of model called **Word2Vec** uses the embedding layer to find vector representations of words that contain semantic meaning.

## Word2Vec

The Word2Vec algorithm finds much more efficient representations by finding vectors that represent the words. These vectors also contain semantic information about the words.

A picture containing person, ball, game

Description automatically generated

Words that show up in similar **contexts**, such as "coffee", "tea", and "water" will have vectors near each other. Different words will be further away from one another, and relationships can be represented by distance in vector space.

There are two architectures for implementing Word2Vec:

* CBOW (Continuous Bag-Of-Words) and
* Skip-gram

A close up of a map

Description automatically generated

In this implementation, we'll be using the **skip-gram architecture** because it performs better than CBOW. Here, we pass in a word and try to predict the words surrounding it in the text. In this way, we can train the network to learn representations for words that show up in similar contexts.

Sentiment RNN Class

I hope you tried out defining this model on your own and got it to work! Below, is how I completed this model.

*I know I want an embedding layer, a recurrent layer, and a final, linear layer with a sigmoid applied; I defined all of those in the \_\_init\_\_ function, according to passed in parameters.*

**def** **\_\_init\_\_**(self, vocab\_size, output\_size, embedding\_dim, hidden\_dim, n\_layers, drop\_prob=0.5):

"""

Initialize the model by setting up the layers.

"""

super(SentimentRNN, self).\_\_init\_\_()

self.output\_size = output\_size

self.n\_layers = n\_layers

self.hidden\_dim = hidden\_dim

*# embedding and LSTM layers*

self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

self.lstm = nn.LSTM(embedding\_dim, hidden\_dim, n\_layers,

dropout=drop\_prob, batch\_first=**True**)

*# dropout layer*

self.dropout = nn.Dropout(0.3)

*# linear and sigmoid layers*

self.fc = nn.Linear(hidden\_dim, output\_size)

self.sig = nn.Sigmoid()

### \_\_init\_\_ explanation

First I have an **embedding layer**, which should take in the size of our vocabulary (our number of integer tokens) and produce an embedding of embedding\_dim size. So, as this model trains, this is going to create and embedding lookup table that has as many rows as we have word integers, and as many columns as the embedding dimension.

Then, I have an **LSTM layer**, which takes in inputs of embedding\_dim size. So, it's accepting embeddings as inputs, and producing an output and hidden state of a hidden size. I am also specifying a number of layers, and a dropout value, and finally, I’m setting batch\_first to True because we are using DataLoaders to batch our data like that!

Then, the LSTM outputs are passed to a dropout layer and then a fully-connected, linear layer that will produce output\_size number of outputs. And finally, I’ve defined a sigmoid layer to convert the output to a value between 0-1.

## Feedforward behavior

Moving on to the forward function, which takes in an input x and a hidden state, I am going to pass an input through these layers in sequence.

**def** **forward**(self, x, hidden):

"""

Perform a forward pass of our model on some input and hidden state.

"""

batch\_size = x.size(0)

*# embeddings and lstm\_out*

embeds = self.embedding(x)

lstm\_out, hidden = self.lstm(embeds, hidden)

*# stack up lstm outputs*

lstm\_out = lstm\_out.contiguous().view(-1, self.hidden\_dim)

*# dropout and fully-connected layer*

out = self.dropout(lstm\_out)

out = self.fc(out)

*# sigmoid function*

sig\_out = self.sig(out)

*# reshape to be batch\_size first*

sig\_out = sig\_out.view(batch\_size, -1)

sig\_out = sig\_out[:, -1] *# get last batch of labels*

*# return last sigmoid output and hidden state*

**return** sig\_out, hidden

### forward explanation

So, first, I'm getting the batch\_size of my input x, which I’ll use for shaping my data. Then, I'm passing x through the embedding layer first, to get my embeddings as output

These embeddings are passed to my lstm layer, alongside a hidden state, and this returns an lstm\_output and a new hidden state! Then I'm going to stack up the outputs of my LSTM to pass to my last linear layer.

Then I keep going, passing the reshaped lstm\_output to a dropout layer and my linear layer, which should return a specified number of outputs that I will pass to my sigmoid activation function.

Now, I want to make sure that I’m returning only the **last** of these sigmoid outputs for a batch of input data, so, I’m going to shape these outputs into a shape that is batch\_size first. Then I'm getting the last bacth by called `sig\_out[:, -1], and that’s going to give me the batch of last labels that I want!

Finally, I am returning that output and the hidden state produced by the LSTM layer.

### init\_hidden

That completes my forward function and then I have one more: init\_hidden and this is just the same as you’ve seen before. The hidden and cell states of an LSTM are a tuple of values and each of these is size (n\_layers by batch\_size, by hidden\_dim). I’m initializing these hidden weights to all zeros, and moving to a gpu if available.

**def** **init\_hidden**(self, batch\_size):

''' Initializes hidden state '''

*# Create two new tensors with sizes n\_layers x batch\_size x hidden\_dim,*

*# initialized to zero, for hidden state and cell state of LSTM*

weight = next(self.parameters()).data

**if** (train\_on\_gpu):

hidden = (weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_().cuda(),

weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_().cuda())

**else**:

hidden = (weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_(),

weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_())

**return** hidden

After this, I’m ready to instantiate and train this model, you should see if you can decide on good hyperparameters of your own, and then check out the solution code, next!