Recurrent neural networks

Training one's own word embeddings as part of the model

- In the previous example, I used pre-trained word embeddings
- This didn't work very well for me when I was trying to prepare these examples (although I didn't try very hard)
- It doesn't change the problem too much to add a word embedding: it's just a collection of vectors, one per possible word
- PyTorch has an way of dealing with learning embeddings that allows us to simply use a numerical word id as input
- Thus, we'll convert our reviews into sequences of word id's
- However, as there are hundreds of thousands of distinct tokens in these reviews, and that would be a very large vocabulary of vectors to learn (and I don't really feel like doing any smarter kind of cleanup on the tokens, which wouldn't be instructive), I'm just going to assign IDs to the most frequent 5000 words, and encode all others to 5001
- By doing this, I am making the input different than in the last examples I did with this problem - a substantially smaller vocabulary, but perhaps a better representation as it will be tuned to the problem - so the results aren't directly comparable

```
from collections import Counter
counts = Counter()
for i, review in enumerate(reviews):
    counts += Counter(review.split())
    if (i+1) % 1000 == 0:
        print(i)
vocab_size = 5000
vocab = list(dict(counts.most_common(vocab_size)).keys())
reviews_indices = []
for review in reviews:
    values = []
    for w in review.split():
        try:
            values.append(vocab.index(w))
```

```
        except ValueError:
            values.append(vocab_size)
    reviews_indices.append(np.array(values))
```

- We will see below what to do with these to train an embedding

Recurrent neural networks

- The idea behind an RNN is as follows
- We process **one token at a time**
- However, unlike in a feed-forward model, we add a **hidden state** (a set of latent variables)
- The hidden state is computed based on **the input token** and **the previous hidden state**
- This is very much like a finite-state model, in which the transition to the next state is determined by the current input symbol and the current state
- However, unlike in a finite state model, the hidden state is a continuous value: thus it is in principle an **infinite-state** machine, which means that it is Turing complete (can encode any computation, even ones which are infeasible or which never terminate)
- If this is true, then why did we bother coming up with these other architectures (LSTM, Transformer?)
- The question, as before, is not about what can in principle be represented, but what tends to be learned from data
- Importantly, there is a literature which investigates the properties of **saturated neural-networks**, that is, networks the values of whose variables are so consistently binary that the cases where they are not are too rare to matter
- If the network is saturated then representations are binary which means **there are only a finite number of states**; in this case, the RNN is just a finite-state machine
- Many argue based on empirical evidence that saturated networks represent islands of stability within the hypothesis space; this would explain why RNNs appear to have limited expressive capacity in practice (see e.g., Merrill et al. 2020, https://arxiv.org/abs/2004.08500; Weiss et al. 2018, https://arxiv.org/abs/1805.04908)

Setting up an RNN in PyTorch

- For our problem, we start by setting up the embedding to be learned, which is orthogonal to the rest
- The following line just sets up a tensor that supports automatic differentiation, and for which PyTorch will help us do some bookkeeping to be able to easily pass in numerical indices and get out the embedding from the table

- I'm picking 32 as the embedding dimension because that's what the Hastie book does

```
emb_dim = 32
emb = torch.nn.Embedding(vocab_size+1, emb_dim)
```

- We have not said anything about how the state transition actually works
- There are a number of ways to do it, but a typical way is to set up two (learned) feed-forward networks, one on the encoding of the token (here, the embedding) which maps it to a vector of the same size as the hidden state, and another on the hidden state, which again maps it to a vector of the same size as the hidden state
- We then take the **sum** of the two
- Of course, to allow the model to be more powerful than a linear model, we pass the result through a non-linearity
- The result is the new hidden state
- I'm picking 16 as the embedding dimension (the Hastie book did 32 as well, but someone on the internet did 16 and got serviceable results on this problem - it surely doesn't matter much - do we really need that many states to do sentiment analysis?)

```
hidden_dim = 16

inp_to_hid = torch.nn.Linear(emb_dim, hidden_dim)
hid_to_hid = torch.nn.Linear(hidden_dim, hidden_dim)

def transition_to_new_state(input, hidden):
    h_0 = inp_to_hid(input)
    h_1 = hid_to_hid(hidden)
    return torch.nn.functional.tanh(torch.Tensor.add(h_0, h_1))
```

- I've used a **tanh** in the above rather than a logistic function - it looks like this

- It has larger gradients than the logistic function when the values are close to zero
- This helps mitigate the **vanishing gradient problem**
- Recall that the gradients are constructed using the chain rule, which means they are a series of multiplications
- As we do a lot of multiplications of very small numbers, the numbers get tinier and tinier, and the learning updates don't move around much
- When we train an RNN, this becomes a particular problem as the function we've applied to the last element in the sequence itself involves many, many multiplicational steps: first apply the network to the first element, then to the second, then to the third, etc etc
- The gradients will become uninformative, hence the particular importance of selecting activation functions that allow for larger update steps
- Now how are we going to use this to classify?

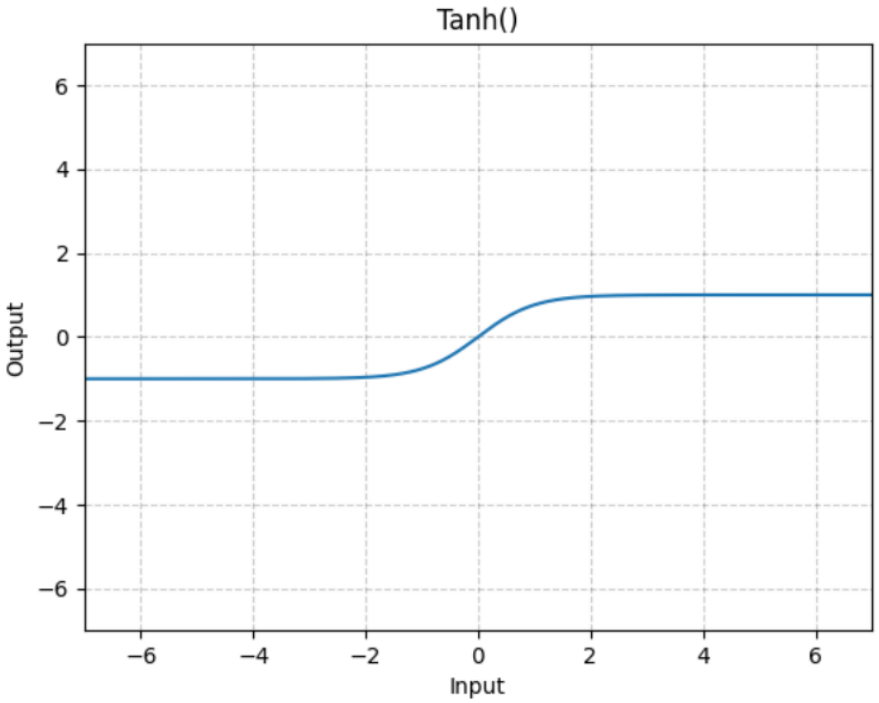

Figure 1: image

- Let's classify at the end of the sequence
- Let's just set up a binary classifier on the hidden state - though instead of accept/reject state, we interpret the classification as **positive/negative** (sentiment)
- You sometimes see other setups, e.g., where the input token also directly conditions the output network

```
lin = torch.nn.Linear(hidden_dim, 1)
```

- So, all in all, to process a string of words (encoded as a sequence of word ids), we do the following (initializing the hidden state to be zero)

```
def do_rnn(input_indices):
    # For Embedding() to work, Torch needs integers
    input_indices_t = torch.LongTensor(input_indices)
    hidden = torch.zeros(hidden_dim)
    e = emb(input_indices_t)
    for i in range(e.shape[0]):
        token = torch.squeeze(e[i,:])
        hidden = transition_to_new_state(token, hidden)
    z = torch.squeeze(lin(hidden))
    return z
```

- One last thing: you're going to need to present only one example at a time
- Yeah: the previous model you could do by matrix-multiplying
- But this model is kind of complicated, and (at least at first glance), it isn't obvious how to do that
- So this is going to be slow
- Now, in each epoch, you should present the examples in a different order
- Or, at the very least, do not present the examples in the order they appear in the data set
- In this data set, all the negative examples precede all the positive examples
- As such, if we let the model train in that order, it will find a super great but super trivial solution, really fast: say no to everything

```
shuffled_indices = train_indices.copy()
random.shuffle(shuffled_indices)
```

Faster RNNs

- It is also super annoying and slow to have to feed the example in one token at a time
- PyTorch has a built-in RNN class which is more efficient: the loop over the tokens is in C++, which means it is faster (on the clock)
- Note that this class returns not only the final hidden states, but all the intermediate hidden states (for your convenience)
- It also supports the possibility of stacking RNNs (feeding the output of

one RNN into another RNN), and, therefore, there is a second axis to the result (because you might want to see the output of each of the RNNs)
- Finally, it also supports the possibility of **bidirectional RNNs** (for reasons that should be clear after our discussion of iterative and subsequential phonological processes), for which the output is determined not by just the hidden states of one RNN, but by the concatenation of those hidden states with the hidden states with those of another RNN that ran over the input backwards
- It's for these reasons that I need to do some indexing when extracting the hidden state at the end of the sequence (see the documentation or do some prints to understand exactly why)

```
rnn = torch.nn.RNN(emb_dim, hidden_dim)
lin = torch.nn.Linear(hidden_dim, 1)

def do_rnn(input_indices):
    input_indices_t = torch.LongTensor(input_indices)
    e = emb(input_indices_t)
    hidden = rnn(e)[1][0,:]
    z = torch.squeeze(lin(hidden))
    return z
```

Batching and packing and padding

- It is, in fact, possible to present multiple examples at once
- This is going to take some doing, and it involves **packing and padding**
- It is not going to be pretty, I warn you
- However, unlike the previous examples, we have too much data to present **all** the data at once without running out of memory
- Thus we are also going to have to split up the data into **batches**
- When you do this, you are getting a (poor) approximation of the gradient, so you can expect slower convergence
- Anyway, these two things together are going to be quite complex to deal with

Batching

- PyTorch has a useful tool for constructing batches of randomly selected training examples, called a **DataLoader**

```
class ReviewDataset(torch.utils.data.Dataset):
    def __init__(self, reviews_indices_t, labels):
        self.reviews_indices_t = reviews_indices_t
        self.y_all = torch.Tensor(np.array(labels, dtype="float16"))
```

```
    def __len__(self):
        return len(self.reviews_indices_t)

    def __getitem__(self, idx):
        X = self.reviews_indices_t[idx]
        y = self.y_all[idx]
        return X, y

training_loader = torch.utils.data.DataLoader(
    ReviewDataset(reviews_indices_tr, labels_code_tr),
    batch_size=batch_size, collate_fn=collate_fn,
    shuffle=True)
```

- I'll then just iterate over the DataLoader in each epoch
- We will see the effect of the *collate_fn* soon

Packing and padding

- My goal is to pass in multiple examples at once into the RNN
- What does that even mean?
- Well, if we were only dealing with the **first** element of each sequence, we could pass the entire data set in at once as before (put them in a matrix and do a matrix multiplication)
- This would yield the hidden states after seeing the first element of each item
- So now, what is keeping us from doing the **second** token of each sequence (assuming each sequence has at least two tokens)?
- We could add an axis to the tensor representing which token we are dealing with and then letting the RNN module iterate, as it did before, over all elements of the sequence
- But of course, not all the sequences necessarily have two elements, and, more generally, not all sequences have the same length
- An approach that one often sees, and which works, is to **pad the sequences** - add zeroes at the beginning or the end
- That way, you can fit all the examples in a large tensor of sequences with the length of the maximal length of the sequence
- This is a pain to deal with yourself, as you then need to keep track of where the "real" end of each sequence is (or beginning if padding at the beginning) so as not to be updating the weights based on the result after pushing the sequence *and then a bunch of zeroes* through the network - it isn't the same thing
- Pytorch has a tool for this called a **PackedSequence**

```
def collate_fn(data):
    X, y = zip(*data)
```

```
    # pack_sequence expects a list of tensors
    # representing sequences,
    # sorted in descending order with respect to
    # length of the sequences;
    # here the tensors are (need to be) already
    # converted to embeddings.
    lengths = [len(r) for r in X]
    X_y_sorted = [(r, lab) for _, r, lab in sorted(
                                zip(lengths, X, y),
                                key=lambda x: x[0],
                                reverse=True)]
    X_emb = torch.nn.utils.rnn.pack_sequence(
        [emb(r) for r, _ in X_y_sorted])
    # targets need to be a tensor as well
    y_t = torch.stack([lab for _, lab in X_y_sorted])
    return X_emb, y_t
```

- It's now remarkably simple to use the RNN class (it was written with this in mind)

```
def do_rnn(e):
    hidden_all = rnn(e)[1]
    z = lin(hidden_all)
    return z
```