Sequence models

Author: Juvid Aryaman
Last compiled: November 13, 2021

This document contains my personal notes on sequence models.

1. Embeddings

Embeddings are tensors. You interact with that tensor by indexing into it. It is often used to store encodings of collections of words. For example:

```
>>> nn.Embedding(vocab_sz, n_hidden)
```

creates a set of vocab_sz tensors, each of size n_hidden.

A common thing to do is to something like:

```
>>> embedding = nn.Embedding(10, 3)
>>> input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)

tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])
```

so you can see that the input is [sentence1, sentence2], where sentence 1 consists of words [1,2,4,5]. As an output, we get the corresponding 3-vectors for each word. So the output is:

```
[[[embedding_word_1,    # length 3 vector
   embedding_word_2,
   embedding_word_4,
   embedding_word_5],

  [embedding_word_4,
   embedding_word_3,
   embedding_word_2,
   embedding_word_9]
]]
```

2. Linear layer

Applies a linear transformation to the incoming data: $y = xA^T + b$

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

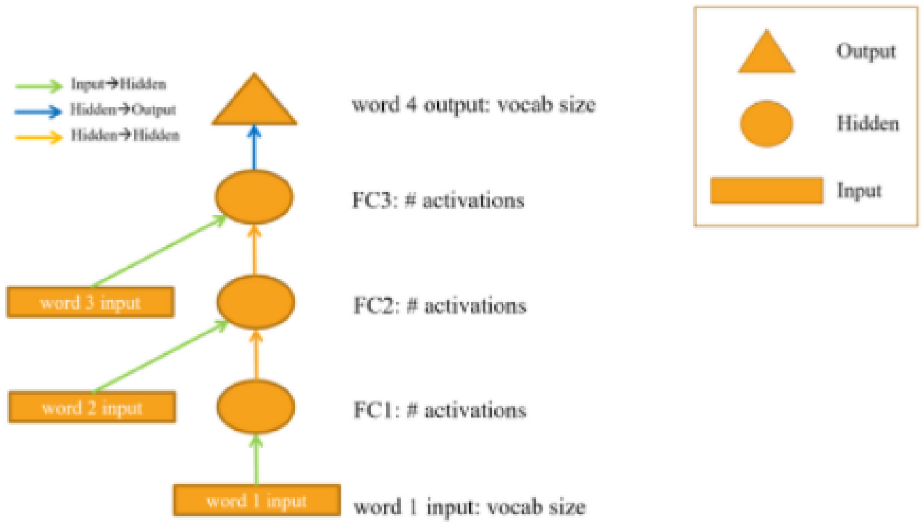

Figure 1. Graphical representation of RNN

3. Recurrent neural network

Torch, by default, applies a multi-layer Elman RNN. This is defined as applying the following function to each element of the input sequence

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h) \tag{3.1}$$
$$y_t = \sigma_y(W_y h_t + b_y) \tag{3.2}$$

where x_t is an input vector, h_t is a hidden layer vector, y_t is an output vector, W, U, b are parameter matrices and vector, σ_h, σ_y are activation functions. Note that we don't actually retain the hidden state between lines – we throw it away after every complete training example (a line). We will typically initialize the hidden state to be $h_{t=0} = 0$. Within a particular training instance, on a particular line, we may have different maximum values of $t = T$.

For example, in word classification, where we construct a character-level RNN, in each training loop we will

1. Get an input and target tensor
2. Create a zeroed initial hidden state
3. Read each letter in and:
   - Keep hidden state for next letter
   - Feed the previous hidden state h_{t-1} in with the current input x_t
4. Compare output at the end of the RNN loop to the target
5. Back-propagate

Then return the output and loss.

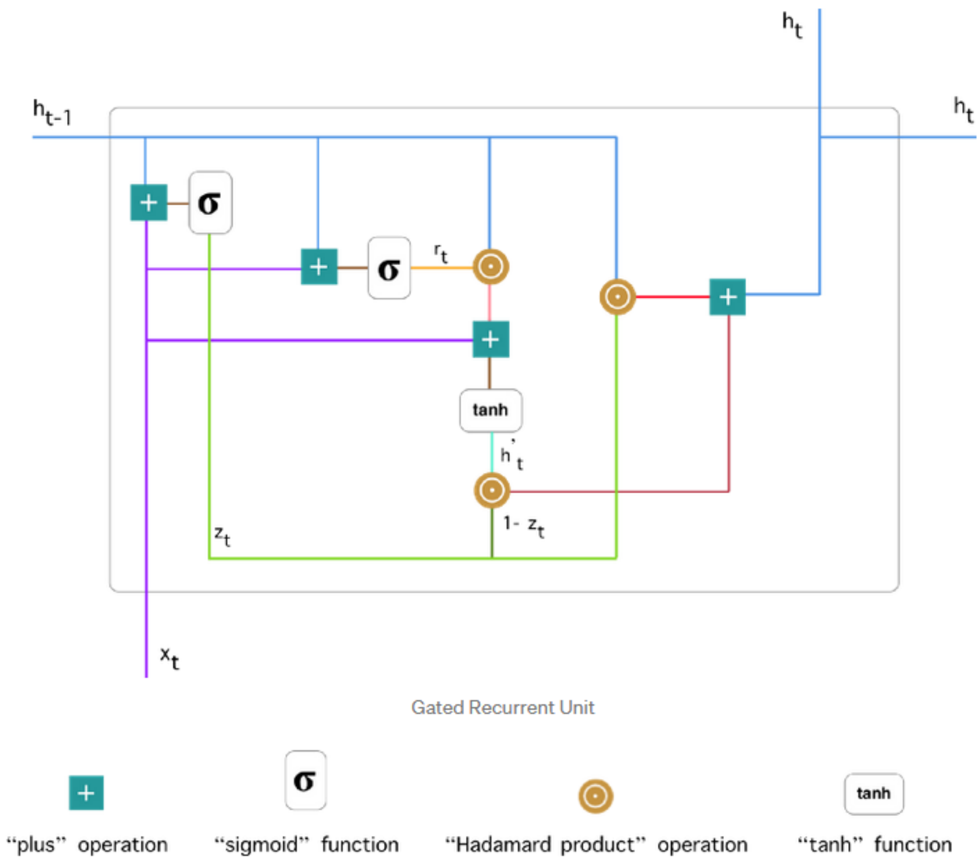

Figure 2. Graphical representation of GRU. I don't actually find these super-helpful.

3.1. Gated recurrent unit

A GRU is a type of RNN. For each element in the input sequence, each layer computes the following function:

$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \tag{3.3}$$
$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \tag{3.4}$$
$$n_t = \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{(t-1)} + b_{hn})) \tag{3.5}$$
$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{(t-1)} \tag{3.6}$$

where x_t is the input at time t, h_t is the hidden state at time t. r_t, z_t, and n_t are the reset, update, and new gates respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

1. Eq.(3.3) is called the **update gate**. The update gate combines the input with the previous hidden state. It determines how much of the previous step's hidden state h_{t-1} is passed onto the new hidden state h_t in Eq.(3.6).
2. Eq.(3.4) is called the **reset gate**. The formula is the same as Eq.(3.3). It will be used to decide how much of the past information to **forget** in Eq.(3.5).
3. Eq.(3.5) is a **candidate** hidden state. It combines the current input with some weighting of the previous hidden state. The reset gate r_t has an element-wise product with h_{t-1}, allowing the network to forget h_{t-1} as $r_t \to 0$.

4. Eq.(3.6) mixes the previous hidden state h_{t-1} with the candidate hidden state n_t through a convex combination weighted by z_t.

Bahdanau *et al.* (2014)

References

Bahdanau, D., K. Cho, and Y. Bengio, 2014 Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 .