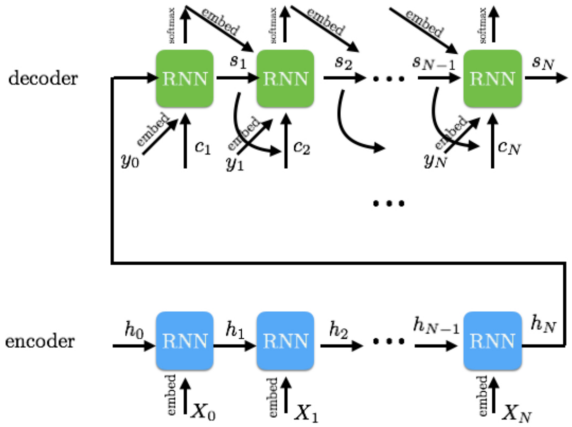
# RECURRENT NEURAL NETWORKS (RNN) – PART 3: ENCODER-DECODER

Edit

In this post, I will cover the basic encoder-decoder which we use to process seq-seq tasks such as machine translation. We will not be covering attention in this post but we will implement it in the next one.

Here we will feed in the input sequence into the encoder which will generate a final hidden state that we will feed into a decoder. The final hidden state from the encoder is the new initial state for the decoder. We will use the decoder outputs with softmax and compare it to the targets to calculate our loss. You can find our more about the paper this model comes from in this **post**. The main difference is that I do not add an EOS token to the encoder inputs and I do not reverse the encoder inputs.




## DATA:

I wanted to create a very short dataset to work with (20 sentences in english and spanish). The point of this tutorial is just to see how to build an encoder-decoder system with soft attention for tasks such as machine translation and other seq-to-seq processing. So I wrote several sentences about me and then translate them to spanish and that is our data.

First we separate the sentences into tokens and then convert the tokens into token ids. During this process we collect a vocabulary dict and a reverse vocabulary dict to convert back and forth between tokens and token ids. For our target language (spanish), we will add an extra EOS token. Then we will pad both source and target tokens to the max length (biggest sentence in the respective datasets). This is the data we feed into our model. We use the padded source inputs as is for the encoder, but we will do further additions to the target inputs in order to get our decoder inputs and outputs.

Finally, the inputs will look like this:

```
encoder inputs:
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   4 141 163   8
 197  80   9 169 206   9  60 190  62  14  28 112 125  38]
decoder inputs
[  1  37 157 139  27   7 197 170  53 196   6   7 161 175   6  78 203   9
 158 140   4   2   0   0   0   0   0   0   0   0   0   0   0]
targets
[ 37 157 139  27   7 197 170  53 196   6   7 161 175   6  78 203   9 158
 140   4   2   0   0   0   0   0   0   0   0   0   0   0   0]
en_seq_lens
18
sp_seq_lens
21
```

This is just one sample from a batch. The 0's are paddings, 1 is a GO token and 2 is an EOS token. A more general representation of the data transformation is below. Ignore the target weights, we do not use them in our implementation.


- SEARCH -

Search …

- FOLLOW ME ON TWITTER -

Tweets by @GokuMohandas

Goku Mohandas
@GokuMohandas

Your model may be performing really well by incorrectly focusing on confounding features (extraneous influencers in the data that aren't accounted for). Check out @johnrzech's post where x-ray stickers unintentionally influenced the classifications: medium.com/@jrzech/what-a…

Embed View on Twitter

- RECENT POSTS -

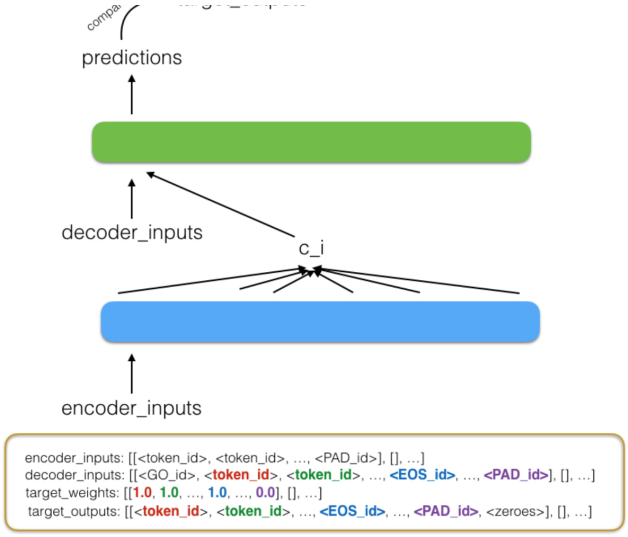
Update on Embeddings (Spring 2017)

Exploring Sparsity in Recurrent Neural Networks

Question Answering from Unstructured Text by Retrieval and Comprehension

Overcoming Catastrophic Forgetting in Neural Networks

Opening the Black Box of Deep Neural Networks via Information




## ENCODER:

The encoder simply taken the encoder inputs and the only thing we care about is the final hidden state. This hidden state holds the information from all of the inputs. We do not reverse the encoder inputs as the paper suggests because we are using seq_len with dynamic_rnn. This automatically returns the last relevant hidden state based on seq_lens.

```
with tf.variable_scope('encoder') as scope:

    # Encoder RNN cell
    self.encoder_stacked_cell = rnn_cell(FLAGS, self.dropo
        scope=scope)

    # Embed encoder inputs
    W_input = tf.get_variable("W_input",
        [FLAGS.en_vocab_size, FLAGS.num_hidden_units])
    self.embedded_encoder_inputs = rnn_inputs(FLAGS,
        self.encoder_inputs, FLAGS.en_vocab_size, scope=sc
    #initial_state = encoder_stacked_cell.zero_state(FLAGS

    # Outputs from encoder RNN
    self.all_encoder_outputs, self.encoder_state = tf.nn.d
        cell=self.encoder_stacked_cell,
        inputs=self.embedded_encoder_inputs,
        sequence_length=self.en_seq_lens, time_major=False
        dtype=tf.float32)
```

We will use this final hidden state as the new initial state for our decoder.

## DECODER:

This simple decoder takes in the final hidden state from the encoder as its initial state. We will also embed the decoder inputs and process them with the decoder RNN. The outputs will be normalized with softmax and then compared with the targets. Note that the decoder inputs starts with a GO token which is to predict the first target token. The decoder input's last relevant token with predict the EOS target token.

```
with tf.variable_scope('decoder') as scope:

    # Initial state is last relevant state from encoder
    self.decoder_initial_state = self.encoder_state

    # Decoder RNN cell
    self.decoder_stacked_cell = rnn_cell(FLAGS, self.dropo
        scope=scope)

    # Embed decoder RNN inputs
    W_input = tf.get_variable("W_input",
        [FLAGS.sp_vocab_size, FLAGS.num_hidden_units])
    self.embedded_decoder_inputs = rnn_inputs(FLAGS, self.
        FLAGS.sp_vocab_size, scope=scope)

    # Outputs from encoder RNN
    self.all_decoder_outputs, self.decoder_state = tf.nn.d
        cell=self.decoder_stacked_cell,
        inputs=self.embedded_decoder_inputs,
        sequence_length=self.sp_seq_lens, time_major=False
        initial_state=self.decoder_initial_state)
```

But what about the paddings, they will also be predicting some output target but we don't really care about those but they will still impact our loss if we factor them in. Here's where we will be masking the loss to remove influence from paddings in the targets.

## LOSS MASKING:

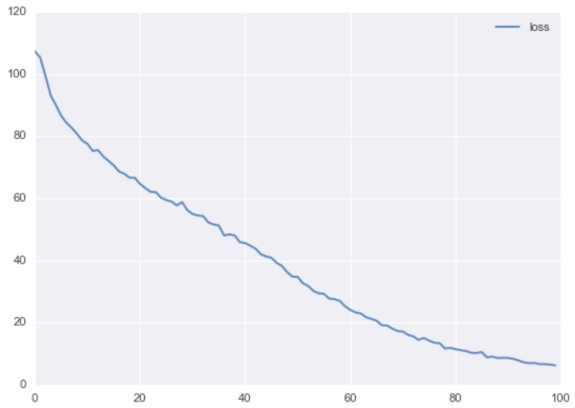
We will use the targets and where ever the target is a PAD, we will mask the loss for that location to 0. So when we get to the last relevant decoder token, the appropriate target will be an EOS token id. For the next decoder input the target will be a PAD id. This is where the masking starts.

```
# Logits
self.decoder_outputs_flat = tf.reshape(self.all_decoder_ou
    [-1, FLAGS.num_hidden_units])
self.logits_flat = rnn_softmax(FLAGS, self.decoder_outputs
    scope=scope)

# Loss with masking
targets_flat = tf.reshape(self.targets, [-1])
losses_flat = tf.nn.sparse_softmax_cross_entropy_with_logi
    self.logits_flat, targets_flat)
mask = tf.sign(tf.to_float(targets_flat))
masked_losses = mask * losses_flat
masked_losses = tf.reshape(masked_losses,  tf.shape(self.t
self.loss = tf.reduce_mean(
    tf.reduce_sum(masked_losses, reduction_indices=1))
```

We will cleverly use the fact that PAD IDs are 0 to apply the loss mask. Once we apply the mask, we just compute the sum of the losses for each row (sample in the batch) and then take the mean of all the sample's losses to get the batch's loss. From here, we can just train on minimizing this loss.

Here are the training results:




We won't be doing any inference here but you can find it the following post with attention. But if you really want to implement inference here, just use the same model as training but you need to feed the predicted target back in as an input for the next decoder rnn cell. You need to embed with the same set of weights used to embed INTO the decoder and have it as another input to the rnn. This means that for the initial GO token, you need to feed in some dummy input token that will be embedded.

## CONCLUSION:

This encoder-decoder model is quite simple but it is a necessary foundation prior to understanding the seq-seq implementation with attention. In the next RNN tutorial, we will cover attentional interfaces and their advantages over this encoder-decoder architecture.

## CODE:

**GitHub Repo** (Updating all repos, will be back up soon!)



*Posted in:* *Machine Translation*, *Sequence-to-Sequence*
*Tagged:* *Tutorials*

← *Recurrent Neural Network (RNN) – Part 5: Custom Cells*

*Recurrent Neural Network (RNN) – Part 4: Attentional Interfaces* →

## 8 THOUGHTS ON "RECURRENT NEURAL NETWORKS (RNN) – PART 3: ENCODER-DECODER"

Pingback: Recurrent Neural Network (RNN) – Part 4: Attentional Interfaces – The Neural Perspective Edit

**NOTHANKS** December 18, 2016 at 10:00 pm EDIT REPLY →

Hi – Github links broken. Thanks!

★ Like

**GOKUMOHANDAS** December 19, 2016 at 11:28 pm EDIT REPLY →

Hi, sorry about that, I was fixing up some old code and didn't want to confuse anyone. It's all up now!

★ Like

**VASILIS B** February 16, 2017 at 11:02 pm EDIT

Nice work, Github link broken again though. Thanks!

★ Like

**JAMESON S** February 16, 2017 at 11:18 pm EDIT

Hey, I love this tutorial! I was able to actually understand most of the tensorflow tutorial after reading this, so to the comment above, code provided there should be good enough to get us started. But I would also like access to your repo but I see this next to the link "(Updating all repos, will be back up soon!)" so I will be waiting patiently!

★ Like

**JATIN KHURANA** March 30, 2017 at 7:37 am EDIT REPLY →

Hello, I am very new to Seq 2 Seq learning. How do we compute the lost function. I mean that for every time instance, output of the decoder is a vector of size vocab_size(of target side) after performing softmax and we have just a number on target output. Should we create one hot vector of size vocab_size and compute the loss ?

I don't know how to do this. Any help would be appreciated!

★ Liked by you

**GOKUMOHANDAS** March 30, 2017 at 2:19 pm EDIT REPLY →

Hey Jatin, good question. So once you have the output of the decoder, you will take that and do TWO things. First is multiple by a set of weights [num_hidden_units out of decoder X target_vocab_size]. This will give you the logits. You can then apply softmax, which is just normalizing the data between 0 and 1. When actually computing the loss in Tensorflow or PyTorch, you'll notice you need to feed in the logits and not the normalized softmax values.

★ Like

**EMBER** October 18, 2017 at 8:52 am EDIT REPLY →

Hi, thanks for your wonderful tutorial. But I do not quite understand how to use tf.nn.dynamic_rnn for inference? Because there is no input sequence but only a start symbol. Thanks.

★ Like

## LEAVE A REPLY

Enter your comment here...