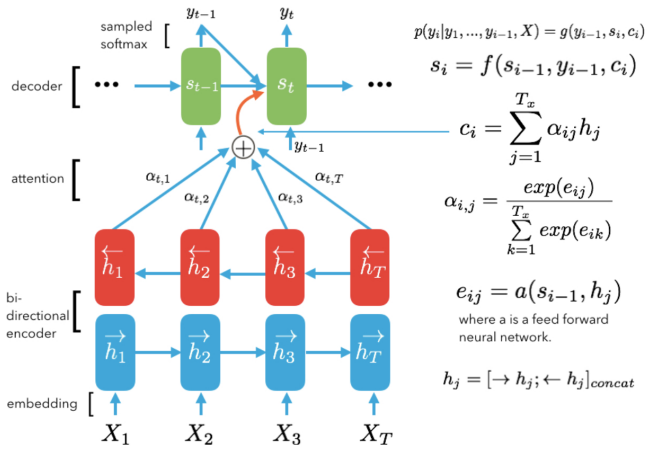
THE NEURAL PERSPECTIVE
DEEP LEARNING SIMPLIFIED

RECURRENT NEURAL NETWORK (RNN) – PART 4: ATTENTIONAL INTERFACES

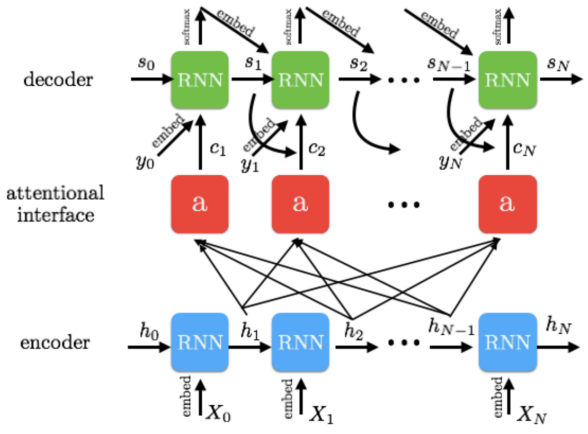
Edit

oIn this post, we will be covering the encoder-decoder architecture with attention for seq-seq tasks. We will loosely follow the implementation from the paper I have simplified **here**.



First we will take a look at the entire model and talk about some interest parts, then we will break down attention and then we will start the implementation with the same model as our encoder-decoder without attention model, which I have detailed **here**. We will slowly introduce attention and we will also implement inferencing. **Note**: This is not going to be a state of the art model, especially since I wrote the data myself in couple minutes. This will purely be a post about understanding the architecture and you can use the implementation with your own larger data sources and you will achieve nice results.

ENCODER-DECODER WITH ATTENTION:



This diagram is a detailed version of the first diagram. We will start from the encoder and move up to the decoder outputs. We have out input data which has been padded and embedded. We feed these into our encoder RNN with a hidden state. The hidden state is initially all zeros but after our first input, it starts changing and holding valuable information. Just know that if you use an LSTM, we will also be passing a cell state c along with our hidden state h. For each input into the encoder we get the hidden state output which is passed for the next input but it is also used as the output for this cell for this input. We can call these h_1 to h_N and these some of our inputs for the attention model.

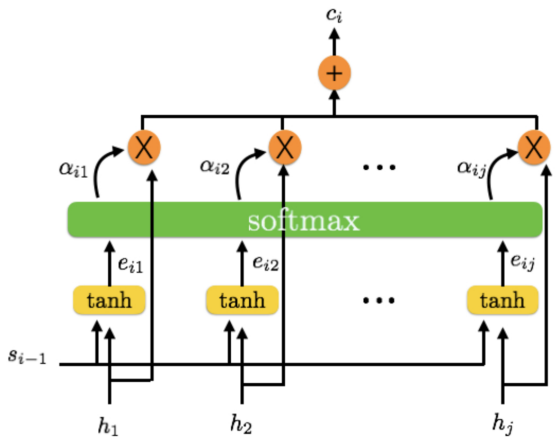
Before we dive deep into the actual attentional interface, let's see how the decoder processes its inputs and generates outputs. Our decoder inputs are the target language inputs with a GO token in the front and an EOS token followed by PADs. We will be embedding these inputs as well. The decoder RNN cell also has a hidden state input. Initially, these will be zeroes and then

change as we feed in inputs. So far, the decoder RNN looks exactly like the encoder RNN, the difference is with an additional input c_i which comes from the attention mechanism. In the next section below, we will take a closer look at how this context c_i is derived but it is essentially the result from the attentional interface based on all of the encoder inputs and our previous decoder hidden state. It tells us how much attention to give to each of the encoder inputs when trying to predict our next output.

From each decoder input, the decoder cell uses the input, previous hidden state and the context vector from attention to predict the target output via softmax. Note that during training, each RNN cell only uses these three inputs to get its target output, but during inference we do not know what the next decoder input. So we will use the previously predicted decoder output as the new decoder input.

Now, let's take a closer look at how the context c_i is calculated from the attentional interface.

ATTENTION MECHANISMS:



Let's initially just focus on the the inputs and outputs from the attention layer. For generating a context c_i for each decoder cell, we use all of the hidden states from all of the encoder inputs and the previous hidden state from the decoder. First, both of these inputs go through a tanh net which produced the output e [NXH]. This happens for all j relevant encoder inputs. We apply softmax of all of the e and now we get a probability for each of the h, which we call alpha. We now multiply each alpha but the originally hidden states h to get a weighted value from the each h. Finally we will sum them up to get our context c_i [NXH] to get the weight representation of the encoder inputs.

Initially, these will be arbitrary contexts but eventually, our model will train and learn which of the encoder inputs are worth weighting in order to accurately predict the decoder target.

TENSORFLOW IMPLEMENTATION:

Now we can focus on implementing this architecture, but most of the focus will be on the attentional interface. We will be using a simple unidirectional GRU encoder and decoder (very similar to the one from the previous **post**) but the decoder will now be using attention. Specifically, we will be using the embedding_attention_decoder() from tensorflow.

First, let take a look at the data that we will process and feed into the encoder/decoder.

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.

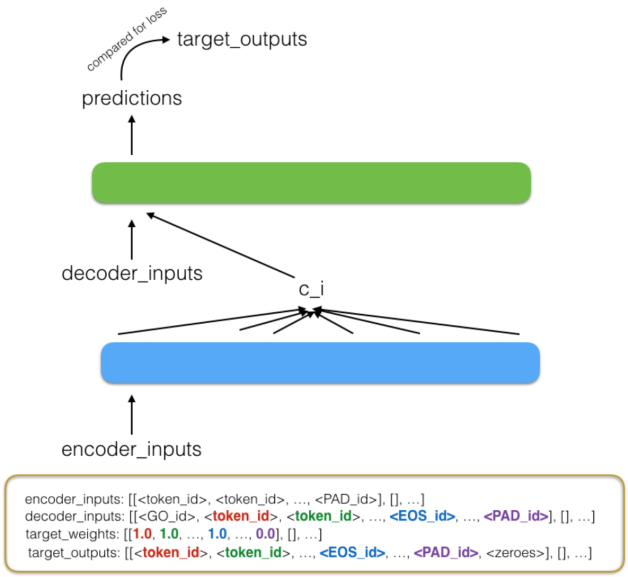
First we separate the sentences into tokens and then convert the tokens into token ids. During this process we collect a vocabulary dict and a reverse vocabulary dict to convert back and forth between tokens and token ids. For our target language (spanish), we will add an extra EOS token. Then we will 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   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.



ENCODER

We feed in the encoder_inputs into the encoder. The inputs are of shape **[N, max_len]** which are embedded into shape **[N, max_len, H]**. The encoder dynamic RNN processes these inputs and seq_lens (**[N,]**) and returns all outputs with shape **[N, max_len, H]** and states of shape **[N, H]** (which is the last relevant state for each input.) We will attend to all of these encoder outputs.

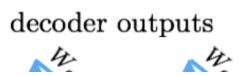
DECODER

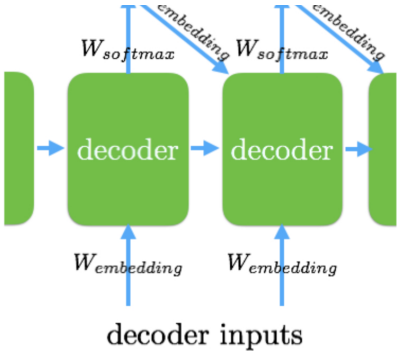
Before talking about the attentional interface, let's quickly see the inputs and outputs of the decoder as well. The decoder's initial state is the encoder's last relevant state for each sample (**[N, H]**). Tensorflow's embedding_attention_decoder() requires the decoder inputs to be time-major list so we convert our decoder inputs from **[N, max_len]** to a max_len sized list **[N]**. We also create our output projection weights, which is another way of calling the softmax weights **[H, C]** for processing the outputs from the decoder. We feed in our time-major list of decoder inputs, initial state, attention states and the projection weights into the embedding_attention_decoder() and we receive all outputs (**[max_len, N, H]**) and state (**[N, H]**). It doesn't matter that our all outputs is time major because we will just be flattening our outputs and applying softmax to convert them to shape ***[Nmax_len, C]***. *We will then also flatten our targets from* ***[N, max_len]*** *to* ***[N****max_len,]* and compute the loss with sparse_softmax_cross_entropy_with_logits(). We will then be masking the loss, in order to remove influence from the predictions where the target was a PAD.

ATTENTION:

Finally, we can focus on the attentional interface. We know it's inputs and outputs but what is happening internally? Our time-major list of decoder inputs, initial_state, attention states (encoder_outputs) all go into our embedded_attention_decoder(). First we will create a new set of weights in order to embed the decoder inputs. Let's call these weights W_embedding. We will then set up a loop function, which will be used after generating a decoder output with a decoder input. The loop function will decide wether to and what to pass into the next decoder cell for processing the next decoder input. Usually, during training we will not pass in the previous decoder output. Here the loop function will just be a None. However, during inference, we will want to pass in the previously predicted decoder output. The loop function we will be using here is _extract_argmax_and_embed() which does exactly what it says. We will take output of the decoder, apply our softmax (output_projection) weights to convert from **[N, H]** to **[N, C]** and then use the same W_embedding to place the embedded output (**[N, H]**) as an input for the next decoder cell.

```
1  # Loop function if using decoder outputs for next predictio
2  loop_function = _extract_argmax_and_embed(
3      W_embedding, output_projection,
4      update_embedding_for_previous) if feed_previous else No
```





One additional option we have with the loop function is update_embedding_for previous which if False, will stop the gradient from flowing through our W_embedding weights when we embed the decoder output (except for the GO token). So, even though we use W-embedding twice, they weights will only depend on the embedding we do on the decoder inputs and NOT on the decoder outputs (except GO token). Now, we can pass our list of embedded time-major decoder inputs, initial_state, attention states and loop function into attention_decoder().

The attention_decoder() is the heart of the attentional interface and there are a few additional processing steps applied not mentioned in the alignment paper. Recall that attention will use our attention states (encoder outputs) and the previous decoder state. It will pass these values into a tanh neural net and we will project e_ij for each of the hidden states. We will then apply softmax to convert to alpha_ij which is multiplied with the original attention state. We take the sum of this value for all attention states and this becomes our new context vector c_i. This context vector is used to eventually produce our new decoder output.

The main difference here is that our attention states (encoder outputs) and the previous decoder state are not processed together with something like _linear() and then applied with a regular tanh. Instead, we do some extra steps. First, the attention states go through a 1×1 convolution. This is a technique to extract meaningful features from our attention states, instead of processing them raw. Recall that conv layers in image recognition acted as excellent feature extractors. The consequence of this step are better features from the attention states but we also now have a 4D representation of the attention states.

```
'''
Transformation in shape:
    original hidden state:
        [N, max_len, H]
    reshaped to 4D hidden:
        [N, max_len, 1, H] = N images of [max_len, 1, H]
        so we can apply filter
    filter:
        [1, 1, H, H] = [height, width, depth, # num filter
    Apply conv with stride 1 and padding 1:
        H = ((H - F + 2P) / S) + 1 =
            ((max_len - 1 + 2)/1) + 1 = height'
        W = ((W - F + 2P) / S) + 1 = ((1 - 1 + 2)/1) + 1 =
        K = K = H
        So we just converted a
            [N, max_len, H] into [N, height', 3, H]
'''

hidden = tf.reshape(attention_states,
    [-1, attn_length, 1, attn_size]) # [N, max_len, 1, H]
hidden_features = []
attention_softmax_weights = []
for a in xrange(num_heads):
    # filter
    k = tf.get_variable("AttnW_%d" % a,
        [1, 1, attn_size, attn_size]) # [1, 1, H, H]
    hidden_features.append(tf.nn.conv2d(hidden, k, [1,1,1,
    attention_softmax_weights.append(tf.get_variable(
        "W_attention_softmax_%d" % a, [attn_size]))
```

This means that, in order to process our transformed 4D attention states with the previous decoder state, we need to convert the previous decoder state to 4D as well. This is easily done by first sending the previous state through an MLP to change it's shape to match the attention states' size and then reshaped to a 4D tensor.

```
y = tf.nn.rnn_cell._linear(
    args=query, output_size=attn_size, bias=True)

# Reshape into 4D
y = tf.reshape(y, [-1, 1, 1, attn_size]) # [N, 1, 1, H]

# Calculating alpha
s = tf.reduce_sum(
    attention_softmax_weights[a] *
    tf.nn.tanh(hidden_features[a] + y), [2, 3])
a = tf.nn.softmax(s)

# Calculate context c
c = tf.reduce_sum(tf.reshape(
    a, [-1, attn_length, 1, 1])*hidden, [1,2])
cs.append(tf.reshape(c, [-1, attn_size]))
```

Now that both the attention states and the previous decoder state have been transformed, we just need to apply the tanh operation. We multiply this with the softmax weights and apply

softmax to give us our alpha_ij. Finally, we reshape our alphas and multiply with original attention states and take the sum to receive our context vectors c_i.

Now we are ready to process our decoder inputs one by one. Let's talk about training first. We do not care about the decode outputs because the input will always be decoder input. So our loop function is none. We process the decoder input with the PREVIOUS context vectors (initially zeroes for first input) with an MLP using _linear(). Then we run the output of that with the previous decoder state through our dynamic_rnn cell. **This** is the reason we needed our decoder inputs to a list of time-major inputs. We processed one time token at a time for all the samples because we need the previous state from the last token at that time index. Time-majored inputs allows us to do this in batch efficiently.

Once we get the dynamic rnn outputs and state, we compute the new context vectors using this new state. The cell outputs are combined with this new context vector and go through an MLP to finally give us our decoder output. All of these additional MLPs are usually not show in decoder diagrams but they are additional steps we apply to get the outputs. Note that the outputs from the cell and the outputs from the attention_decoder itself both have shape **[max_len, N, H]**.

If we were doing inference, our loop function is no longer None but the _extract_argmax_and_append(). This takes in the previous decoder output and our new decoder input is just the previous output with softmax applied to it, following by reembedding. And after we do all the processing with the attention states, the prev is updated to be the newly predicted output.

```
# Process decoder inputs one by one
for i, inp in enumerate(decoder_inputs):

    if i &amp;gt; 0:
        tf.get_variable_scope().reuse_variables()

    if loop_function is not None and prev is not None:
        with tf.variable_scope("loop_function", reuse=True
            inp = loop_function(prev, i)

    # Merge the input and attentions together
    input_size = inp.get_shape().with_rank(2)[1]
    x = tf.nn.rnn_cell._linear(
        args=[inp]+attns, output_size=input_size, bias=Tru

    # Decoder RNN
    cell_outputs, state = cell(x, state) # our stacked cel

    # Attention mechanism to get Cs
    attns = attention(state)

    with tf.variable_scope('attention_output_projection'):
        output = tf.nn.rnn_cell._linear(
            args=[cell_outputs]+attns, output_size=output_
            bias=True)
    if loop_function is not None:
        prev = output
    outputs.append(output)

return outputs, state
```

And of course, we processes the outputs from our attention_decoder with softmax, flatten and then compare with targets to compute loss.

NUANCES:

SAMPLED SOFTMAX

Using attentional interfaces like this are excellent architecture for seq-seq tasks such as machine translation. But a common issue is the large size of the corpus. This especially proves to be computationally expensive during training when we need to compute the softmax with the decoder outputs. The solution here is to use a sampled softmax. You can read more about why and how in my post **here**.

Here is the code for a sampled softmax. Note that the weights are the same as our output_projection weights we are using with the decoder, since they both are doing the same task (converting decoder output H-length vector to num_class length vector).

```
def sampled_loss(inputs, labels):
    labels = tf.reshape(labels, [-1, 1])
    # We need to compute the sampled_softmax_loss using 32
    # avoid numerical instabilities.
    local_w_t = tf.cast(w_t, tf.float32)
    local_b = tf.cast(b, tf.float32)
    local_inputs = tf.cast(inputs, tf.float32)
    return tf.cast(
            tf.nn.sampled_softmax_loss(local_w_t, local_b,
                local_inputs, labels,
                num_samples, self.target_vocab_size),
            dtype)
softmax_loss_function = sampled_loss
```

And then, we can just process the loss with a seq_loss function, where weights are 1 everywhere except 0 where target outputs are PADs. **Note:** Sampled softmax is only used for training but during inference we have to use regular softmax because we want to sample from the entire corpus for the word, not just a select few that best approximate the corpus.

```
else:
    losses.append(sequence_loss(
```

```
        outputs, targets, weights,
        softmax_loss_function=softmax_loss_function))
```

MODEL WITH BUCKETS:

Another common architectural addition is to use tf.nn.seq2seq model_with_buckets(). This is what the official tensorflow NMT **tutorial** uses and the main advantage of these buckets is with the attention states. With our model, we are applying attention to all <max_len> hidden states. We should only be doing them to the relevant states since the PADs do not need any attention anyway. With buckets, we can place our sequence in select buckets so there are very few PADs for a given sequence.

However, I find this method a bit crude and if we really wanted to process out the PADs, I would suggest using seq_lens to filter out the PADs first out of the encoder outputs OR once we compute the attention context vectors_i, we zero out the locations where the hidden state comes from a PAD for each sequence. This is a bit complicated and we won't implement it here but buckets does serve as a decent solution for this extra noise.

CONCLUSION:

There are many more variants to this attentional interface and it is a growing area of research. I like to use this architecture above for many seq-seq processing tasks as it produces decent results for many situations. Just be wary to have large training and validation sets because these models can easily overfit and produce horrible results for validation. In subsequent posts, we will be using these attentional interfaces for more complicated tasks involving memory and logical reasoning.

CODE:

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

SHAPES ANALYSIS:

Encoder inputs are **[N, max_len]** which are embedded and transformed to **[N, max_len, H]** and fed into the encoder RNN. The outputs are **[N, max_len, H]** and state is **[N, H]** holding the last relevant state for each sample.

The encoder outputs are same as the attention states of shape **[N, max_len, H]**.

Decoder inputs is shape **[N, max_len]** which are converted to a **max_len** long time-major list of shape **N**. Decoder initial state is the encoder state of shape **[N, H]**. Before getting input into the decoder RNN, the inputs are embedded in to a **max_len** long time-major list of shape **[N, H]**. The input may be the actual decoder input or the previous output (during inference). If doing inference, the previous output is derived from the decoder. The output from the decoder from the previous time step had shape **[N, H]** which sent into a softmax layer (output projection) into shape **[N, C]**. This output is then reembedded using the same weights we use to embed the inputs, into shape **[N, H].** These inputs are fed into the decoder RNN which produces decoder outputs of shape **[max_len, N, H]** and state **[N, H]**. The outputs are flattened to [**N*max_len, H]** *and compared with flattened targets (also of shape [**N*max_len, H]*). The losses are masked where targets are PADs and backprop ensues.

Inside the decoder RNN, there are quite a few operations. First it takes the attention states (encoder outputs) of shape **[N, max_len, H]** and convert to a 4D **[N, max_len, 1, H]** (so we can apply conv) and applies convolution to extract meaningful features. The shape of these hidden features is 4D **[N, height', 3, H]**. The previous hidden state from the decoder, of shape **[N, H]**, is also an input to the attentional interface. This hidden state goes through an MLP into shape **[N, H]** (this was done incase the previous hidden states second dimension (H) was different from the attention_size, which is also H for us). Then this previous hidden state is converted to 4D **[N, 1, 1, H]** so that we can combine with the hidden features. We apply tanh to this addition to produce e_ij and then take the softmax to produce alpha_ij, of shape **[N, max_len, 1, 1]** (which is representing the probability of each hidden state to use for each sample). This alpha is multiplied with the original hidden states, of shape **[N, max_len, 1, H]**, and then summed to create our context vector c_i, of shape **[N, H]**.

This context vector c_i is combined with the decoder inputs, of shape **[N, H]**, which is the case regardless of wether the input is from the decoder inputs data (training) or from the previous prediction (inference). This inputs is just one from the list of length **max_len** of shape **[N, H]**. First we add the previous context vector (initially it's zeros of shape **[N, H]**) to the input. Recall that the inputs are from decoder inputs which is a time_major length **N** list of **[max_len,]**, which is why each input will be size **[N, H]**. An MLP is assigned to the addition of the input and the context vector to create an output with shape **[N, H]**. This is fed into our dynamic RNN cell along with the state, of shape **[N, H]**. The outputs are cell_outputs of shape **[N, H]** and the state is also shape **[N, H]**. The new state becomes the state we use for the next decoder operation. Recall that we generate these outputs of shape **[N, H]** for all max_len so at the end we shape a **max_len** length list of shape **[N, H].** After getting the output and state from the decoder cell, we process the new context vector by passing in this new state into the attention function. This new context vector of shape **[N, H]** is combined with the outputs of shape **[N, H]** and an MLP is applied to the sum and converted to shape **[N, H]**. Finally, if we are using inference the new prev becomes this output (initially prev was none). This prev is used as an input into loop_function to get the input for the next decoder operation.

Posted in: *Attention*, *Sequence-to-Sequence*
Tagged: *Tutorials*

← *RECURRENT NEURAL NETWORKS (RNN) – PART 3: Encoder-Decoder*

Reinforcement Learning (RL) – Policy Gradients I →

12 THOUGHTS ON "RECURRENT NEURAL NETWORK (RNN) – PART 4: ATTENTIONAL INTERFACES"

WASIM KARANI December 8, 2016 at 10:40 am EDIT REPLY →

Hello Goku Mohandas
I was following your videos and I learnt so many thing about RNN from you github repo and these explanations. Why your github repo not working anymore.

★ 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! And I don't make videos but would it helpful if I did for some topics?

★ Like

Pingback: Highlights and Tutorials for Concepts Discussed in "Richard Socher on the Future of Deep Learning" – The Neural Perspective Edit

RYH April 21, 2017 at 9:27 am EDIT REPLY →

Hi, this post is great but I have a question
What do you mean by saying "Our attentional states become just the encoder outputs because the attention for a simple sample is just all of its hidden states from all the outputs." in encoder, TENSORFLOW IMPLEMENTATION section ?

★ Liked by you

GOKUMOHANDAS April 21, 2017 at 1:18 pm EDIT REPLY →

Hi, my wording is a bit confusing. What I meant to say was that, in this simple case, we will be attending to all of the encoder outputs. I will fix the wording and checkout my new O'Reilly post for a more recent, better written post 🙂
https://www.oreilly.com/ideas/interpretability-via-attentional-and-memory-based-interfaces-using-tensorflow

★ Like

RYH April 24, 2017 at 10:54 am EDIT

Thanks for your reply
and I have a confusion on "shapes analysis" section
In your essay, if an embed vector is H dimensional as input then its output of the encoder_rnn is H dimensional too
But I thinks it is a D dimensional vector where D is the num_unit of LSTM if this encoder rnn is LSTM
So I mean the embedding dimension may not be the same as output dimension of rnn, D is not equal to H in most cases , if so , then output of encoder rnn is [N, max_len, H] may be misleading or just I'm wrong ?

★ Liked by you

GOKUMOHANDAS April 24, 2017 at 5:40 pm EDIT REPLY →

Hey RYH, can't reply to your latest message so hope you see this here. You can have D

Hey RYH, can I reply to your latest message so hope you see this here. You can have D and H be the same number. You'll see some papers use the same throughout the models and others will differentiate with different letters.

★ Like

RYH April 25, 2017 at 5:13 am EDIT REPLY →

Got it
thanks for your patience

★ Liked by you

ARMAN June 9, 2017 at 1:05 am EDIT REPLY →

Hi, thanks for the great post.
The github repo seems to be unavailable. I was wondering if you can put the code back online! Thanks!

★ Like

GOKUMOHANDAS June 9, 2017 at 1:50 pm EDIT REPLY →

Hey arman, these tutorials are outdated and written for Tensorflow. New updated PyTorch tutorials with code will be available soon. But if you specifically interested in code for attentional interfaces checkout my article here: https://www.oreilly.com/ideas/interpretability-via-attentional-and-memory-based-interfaces-using-tensorflow

★ Like

MEHDI August 18, 2017 at 10:41 pm EDIT REPLY →

attention states is not encoder hidden states, it's encoder outputs

★ Liked by you

GOKUMOHANDAS August 18, 2017 at 11:00 pm EDIT REPLY →

Ah good catch, thanks Medhi

★ Like

LEAVE A REPLY

Enter your comment here...