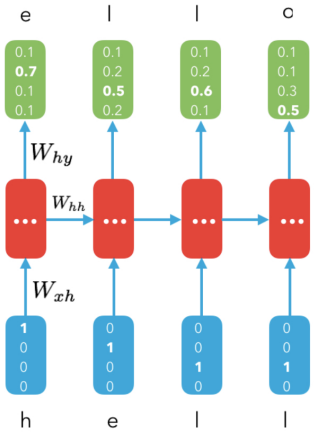
RECURRENT NEURAL NETWORK (RNN) – PART 5: CUSTOM CELLS

Edit

In this post, we will explore the idea of creating our own custom RNN cells. But first, we will take a closer look at the simple RNN and then more complicated units such as LSTM and GRU. We will also analyze the tensorflow code for these units and draw from them to eventually create our own custom cells. In this post, I will be using images from one of the best posts out there on RNNs/LSTMS by Chris Olah. I highly urge you to read the **post** and in my post I will be reiterating a lot of material but I will move rather quickly and focus more on the tf code. I will be referring back to this code in a future post on applying layer normalization to these RNN architectures, which can be found **here**.

BASIC RNNS:

With traditional RNNs, the main issue is that we cannot adequately learn long term dependencies because the operations that we repeat at each cell unit for each input are static. If you think back to the basic RNN cell, the operations all involve the single tanh operation.



Forward Pass:

$$X_t$$

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}X_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

$$P = \frac{e^y}{\sum e_y}$$

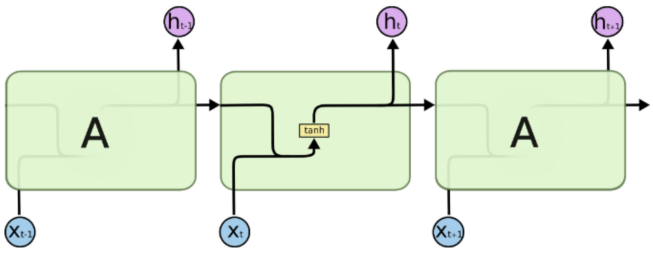
$$L = -h(P)$$

X: [NXC] W_{xh}: [CXH]
H: [NXH] W_{hh}: [HXH]
y: [NXC] W_{hy} : [HXC]

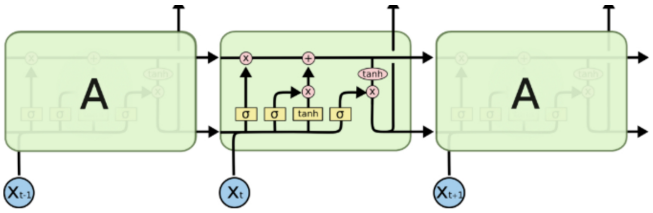
This architecture is suitable inputs where the solutions are based on short term dependencies but if we wish to utilize long term memory efficiently to predict the right targets, we will need a rnn cell unit that is more robust. Cue the LSTM.

LONG SHORT TERM MEMORY NETWORKS (LSTMS):

The architecture of the LSTMs allows us to have long term information control at the expensive of more operations. Our traditional RNNs had one output which served as both the hidden state representation and the output from the cell.

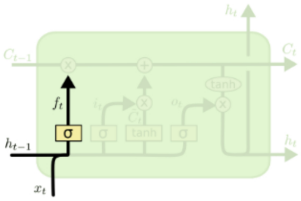


There is an absence of information control with this basic architecture that prevents us from holding on to useful information for many steps down the line. The LSTM, instead, has two different types of outputs. We still the traditional state output which acts as the hidden state representation and the cell's output but the cell also outputs a cell state C. Here is the LSTM in all its glory, time to break it down into pieces.



FORGET GATE:

The very first gate is the forget gate. This gate allows us to selectively pass information to determination of the cell state. I will break down the notation below once and you can reapply for all the other gates as well.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

$$\begin{aligned} f_t &= \sigma(W_{\sigma_h} h_{t-1} + W_{\sigma_x} x_t + b_t) \\ &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_t) \end{aligned}$$

W_f: [D+H X H]

h_{t-1} : [N X H]

x_t : [N X D]

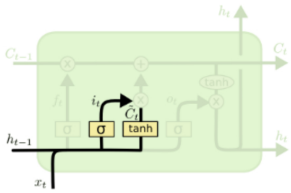
b_t : [H]

And of course to implement this, you could follow something like tf's _linear function. But the main idea is that we are applying this sigmoid operation to both the input and the previous hidden state. But what exactly is applying this sigmoid operation doing? Recall that sigmoid outputs in the range [0, 1] and here we are applying it to a matrix of shape [N X H], which means we will produce NXH values with sigmoid applied to them. If the sigmoid operation results in 0, then that hidden value is nullified and it it is 1, we completely let that value be used. Anything in between allows parts of the information to go through. This is an nice way to control the information that is flowing through by effectively blocking and selectively passing parts of the inputs to the cell.

This forget gate, however, is only the first operation that we do to ultimately calculate our cell state. The next operation involves the input gate.

INPUT GATE:

The input gate takes in our input X and the previous hidden state and computes two operations. First it selectively allows parts of the inputs to pass through with a sigmoid gate and then we multiply it by the tanh of the inputs.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

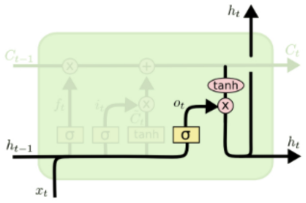
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

What the tanh is doing here is a bit different from the sigmoid operation. Recall that tanh changes our inputs into the range [-1, 1]. This essentially changes the underlying representation of our inputs with this nonlinearity. This is the exact same step as what we were doing with the basic RNN cell. But now we take the product of these two values and add it to the value from the forget gate to calculate our cell state.

These operations with the forget and input gate can be translated to the fact that we keep parts of the old cell state (C_{t-1}) and keep parts of the new transformed (tanh) cell state C~_t. These weights are trained with our data to learn exactly how much information to keep and how to perform the correct transformation.

OUTPUT GATE:

The last gate is the output gate and it uses the input, previous hidden state and the new cell state to determine the new hidden state representation.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

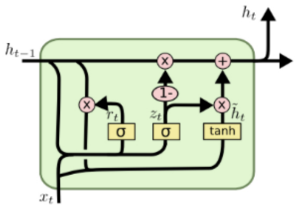
This operation again involves the selective information barrier sigmoid which is multiplied with tanh of the cell state. Note that this tanh operation is not a neural network as with the tanh operation in the input gate. This is simply applying the tangent to the cell state without any modifications with weights. We are merely forcing the cell states [NXH] values to be in the range [-1, 1].

VARIATIONS:

There are literally hundreds of variations for RNN cells so I suggest checking our Chris Olah's **blog** again for more information. A few note worthy one's he discussed were the peephole model (allow all gates to see the cell state available at that point in time (C_{t-1} or C_t is already calculated) and coupled cell states (only update when we forget and forget when we update). But the current rival to the LSTM, which is heavily based off of the LSTM and it rapidly growing in use is the Gated Recurrent Unit (GRU).

GATED RECURRENT UNIT (GRU):

The main idea behind the GRU is that is combines the forget and input gate into one update gate.



$$z_t = \sigma\left(W_z \cdot \left[h_{t-1}, x_t\right]\right)$$
$$r_t = \sigma\left(W_r \cdot \left[h_{t-1}, x_t\right]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot \left[r_t * h_{t-1}, x_t\right]\right)$$
$$h_t = \left(1 - z_t\right) * h_{t-1} + z_t * \tilde{h}_t$$

Empirically, the GRU's performance on most tasks is on par with the LSTM and also computationally less expensive. These tradeoffs are the reason behind it's surging popularity.

TENSORFLOW NATIVE IMPLEMENTATIONS:

Now we will take a look at the official Tensorflow code the GRU unit and we will mostly focus on the function calls, inputs and outputs. From here, we will replicate the structure to create our own unique cells. If you're interested in the other cells available, you can find them all at this **link**. We will just focus on the GRU because it's performance is as good as the LSTM in more cases and significantly less complex.

```
class GRUCell(RNNCell):
  """Gated Recurrent Unit cell (cf. http://arxiv.org/abs/1

  def __init__(self, num_units, input_size=None, activatio
    if input_size is not None:
      logging.warn("%s: The input_size parameter is deprec
    self._num_units = num_units
    self._activation = activation

  @property
  def state_size(self):
    return self._num_units

  @property
  def output_size(self):
    return self._num_units

  def __call__(self, inputs, state, scope=None):
    """Gated recurrent unit (GRU) with nunits cells."""
    with vs.variable_scope(scope or type(self).__name__):
      with vs.variable_scope("Gates"):  # Reset gate and u
        # We start with bias of 1.0 to not reset and not u
        r, u = array_ops.split(1, 2, _linear([inputs, stat
                                              2 * self._num
        r, u = sigmoid(r), sigmoid(u)
      with vs.variable_scope("Candidate"):
        c = self._activation(_linear([inputs, r * state],
                                     self._num_units, True
      new_h = u * state + (1 - u) * c
    return new_h, new_h
```

The GRUCell class start with the __init__ function which defines the number of units and the activation function it will use. This is the activation function that is usually tanh but the sigmoid activations are fixed since the [0,1] range allows us to control the information flow. Then we have two properties that both return self._num_units when invoked. And finally, we have out __call__ function which is what processes the input and churns out the new hidden state. Recall that GRU does not have a cell state like the LSTM.

First, we compute r and u (u = z in colah's notation above). Instead of separately doing them, we just merge the weights and do it with 2*num_units and then we split it by two. **split(dim, num_splits, value)**. Then we apply our sigmoid activate on the values to selectively control the information flow. Then we calculate the candidate c and use it to calculate out new hidden state representation. You may see that the order for calculating new_h is switched, either way works fine, because the weights will train accordingly.

All of the other cells' codes look very similar to this, so you will easily be able to interpret them.

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

← *Embeddings (skipgram and CBOW) Implementations*

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

ONE THOUGHT ON "RECURRENT NEURAL NETWORK (RNN) – PART 5: CUSTOM CELLS"

Pingback: RECURRENT NEURAL NETWORKS (RNN) – PART 3: Neural Machine Translation (NMT) – The Neural Perspective Edit

LEAVE A REPLY

Enter your comment here...