Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

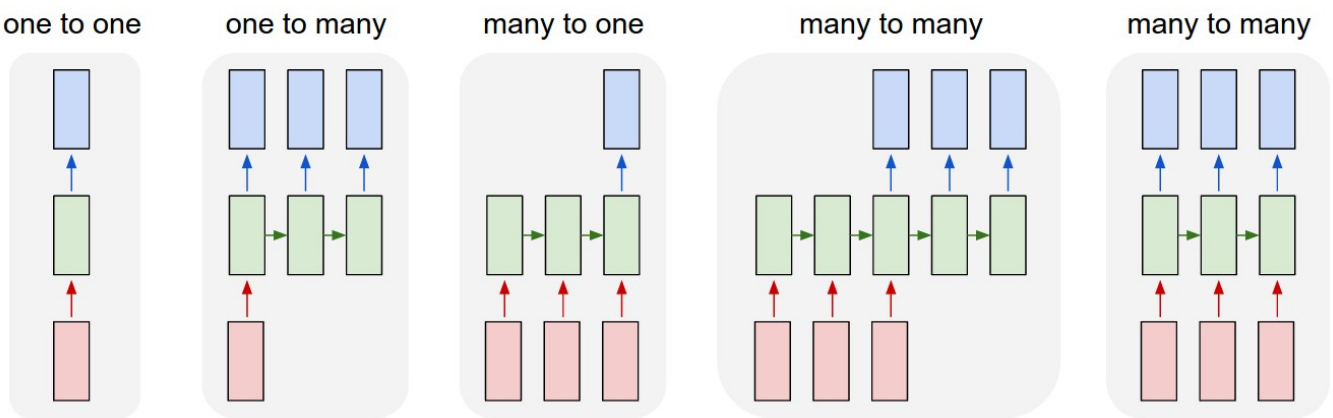
Benjamin Milde, Fabian Barteld, Prof. Dr. Chris Biemann

PART 05: NEURAL TAGGER WITH RNNS IN TENSORFLOW

Introduction

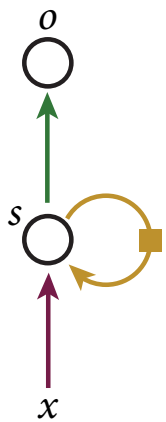
RNNs - different variants

- DNNs have an obvious limitation for NLP, they always need a fixed-sized input. Recurrent Neural Networks (RNNs) on the other hand operate on sequences of varying length ideal for text and speech
- Depending on the task, RNNs can be used in different ways
- All of these are fairly straightforward to do in Tensorflow



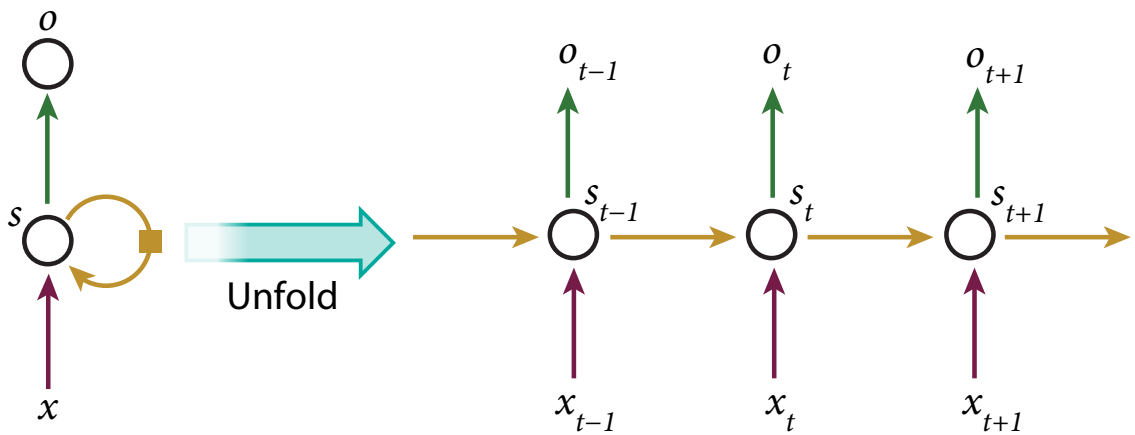
RNNs

- Update a state given inputs at each timestep
- Can be ”unrolled” into a DNN:



RNNs

- Update a state given inputs at each timestep
- Can be ”unrolled” into a DNN:

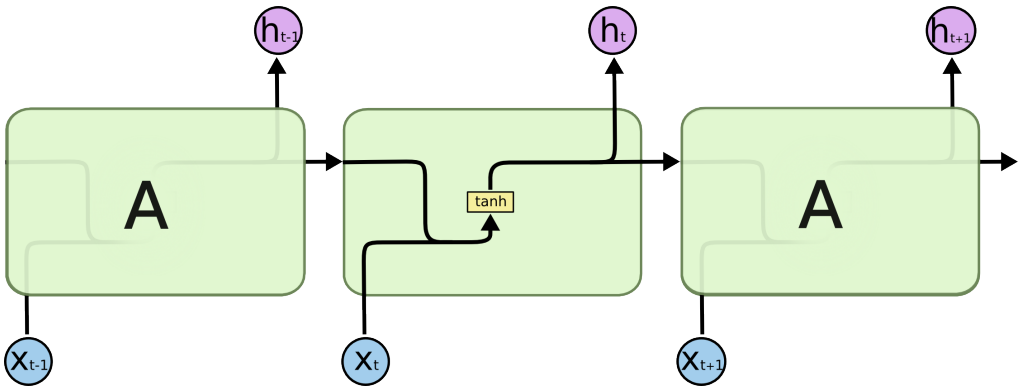


LSTMs - intuition

- Vanilla RNNs are difficult to train: the same operation is applied at ever time step - difficult to optimize for many time steps (vanishing and exploding gradients)
- When we use RNNs, we usually use a variant that has a gating mechanism
- Intuition: sometimes we need to forget things, also we want to learn how much to change our internal state given certain inputs
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

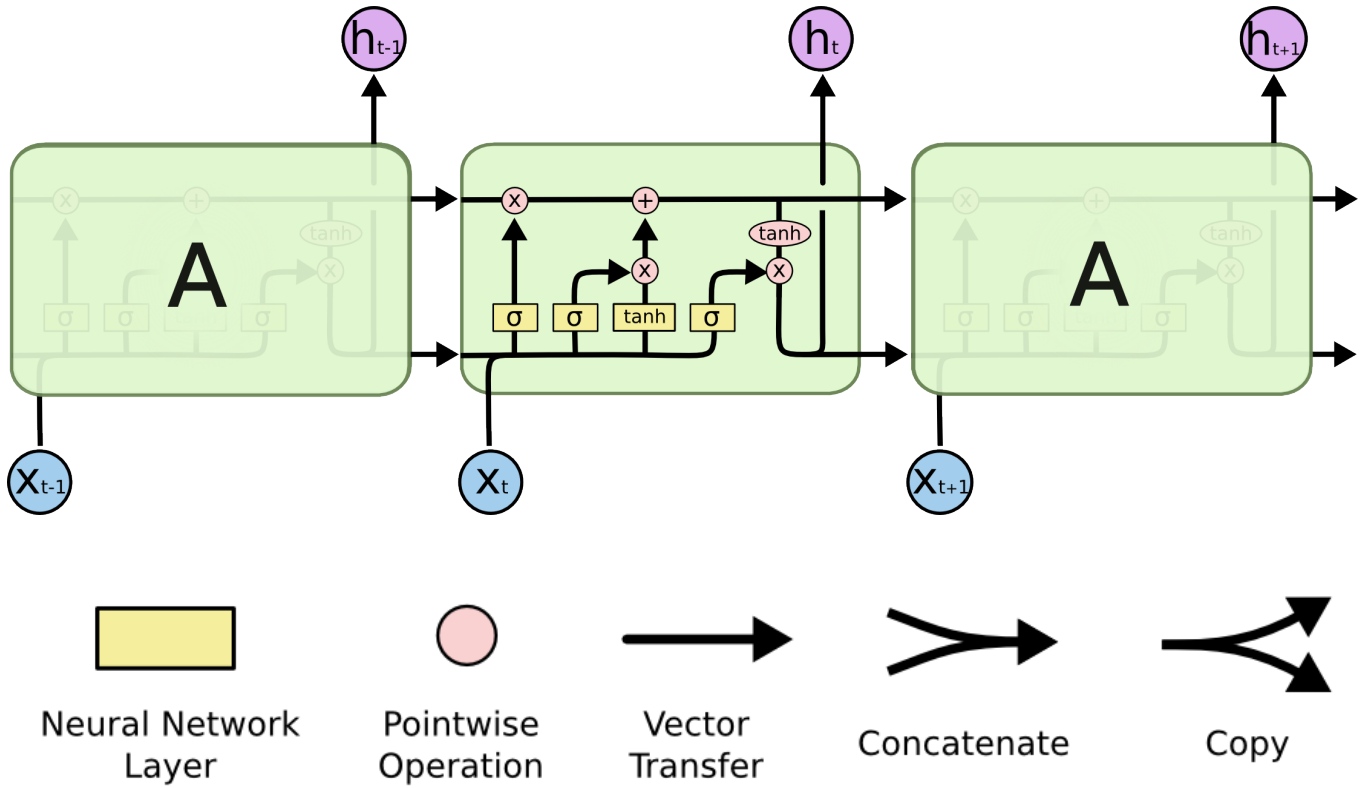
Understanding LSTMs

- All graphics and some of the explanations on this and on the following slides are from: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`
- The repeating module in a standard RNN contains a single layer:



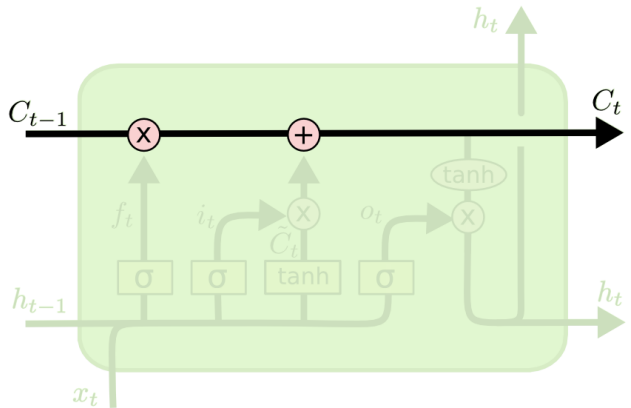
Understanding LSTMs

- An LSTM cell has more complex operations:



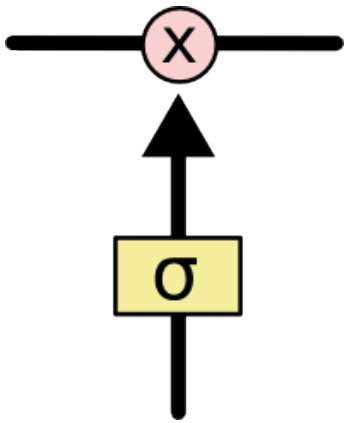
Understanding LSTMs

- The key to LSTMs is the cell state.
- It runs straight down the entire chain, with only some minor linear interactions.
- It makes it easy for information to just flow along unchanged.



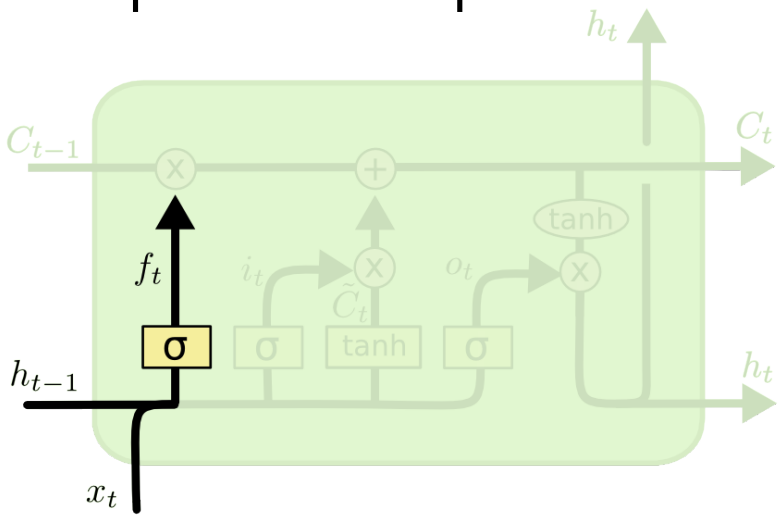
Understanding LSTMs

- Another key element of an LSTM cell is gating. A gate consists of a sigmoid layer and a multiplication.
- The sigmoid pushes all values to be between 0 and 1
- You can think of it as some kind of ”differentiable Transistor”



Understanding LSTMs

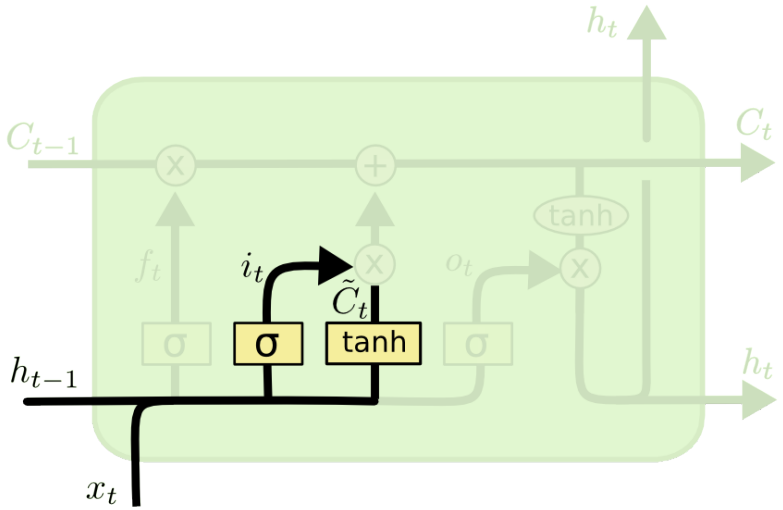
- The first step in an LSTM is to decide what information is going to be thrown away from the cell state.
- This decision is made by the ”forget gate layer”
- It concatenates h_{t-1} and x, applies a sigmoid layer and multiplies the output with the cell state.



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

Understanding LSTMs

- The next step is to decide what new information we're going to store in the cell state.
- First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, *Ct*.

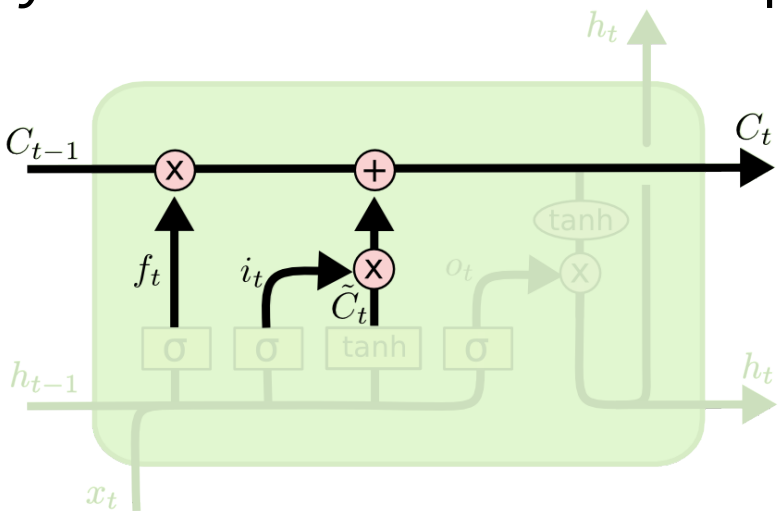


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

Understanding LSTMs

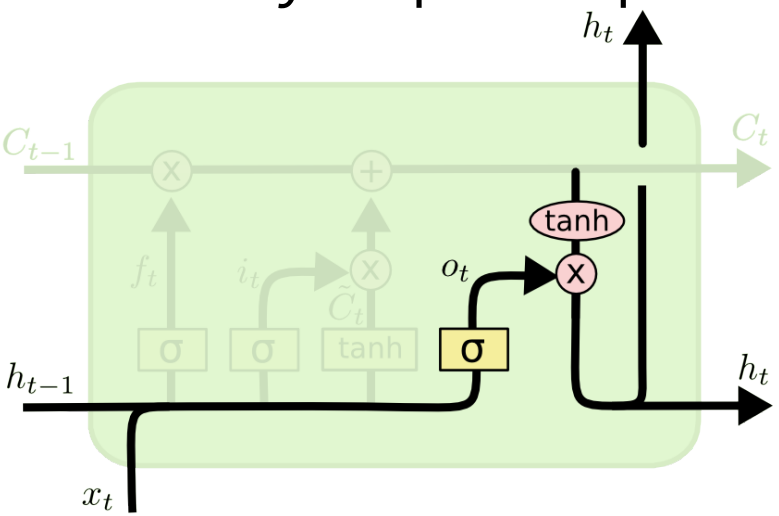
- Now the new cell state is calculated
- We multiply the old state by f_t, forgetting the things we decided to forget earlier.
- Then we add $i_t \cdot Ct$. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Understanding LSTMs

- Finally we compute the output based on the cell state
- The first step is a sigmoid layer which decides what parts of the cell state we're going to output.
- Then, we put the cell state through tanh (values between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

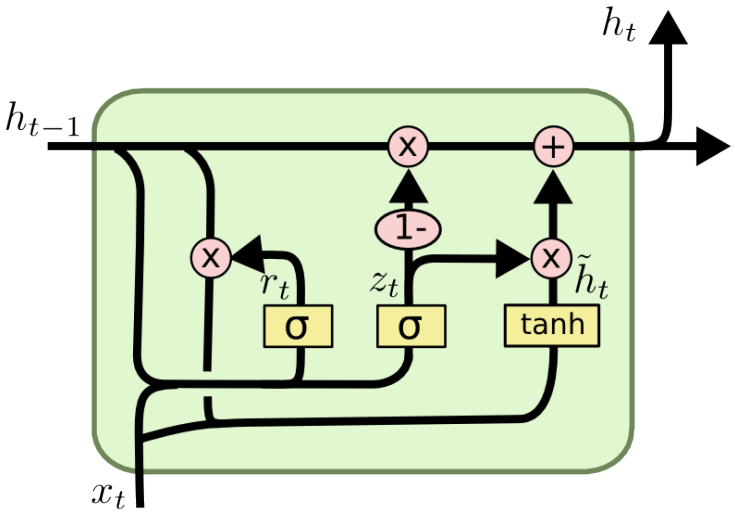


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Popular variant of LSTMs: GRU

- Combines the forget and input gates into a single ”update gate.”
- It also merges the cell state and hidden state, and makes some other changes.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTMs in Tensorflow

- Good news! RNNs are first class citizens in Tensorflow, you don't have to code the gating mechanism on your own
- Two types of RNNs: static_rnn and dynamic_rnn, also many types of cells, e.g. LSTMs and GRUs.
- The so called static_rnn expects a list of tensors, one for each step. Number must be known at graph creation time!
- Sequences of smaller length must be padded!

```
cell = tf.contrib.rnn.LSTMCell(num_hidden)
list_of_tensors = [x_embedd[:,num] for num in range(seq_len)]
outputs, state = tf.nn.static_rnn(cell, list_of_tensors,
 dtype=tf.float32)
```

LSTMs in Tensorflow

```
outputs, state = tf.nn.static_rnn(cell, list_of_tensors,
 dtype=tf.float32)
```

- outputs is a list of tensors (?, hidden_size), one of these for each output.
- For LSTMs, state is a tuple of c and h:

tf.contrib.rnn.LSTMStateTuple

class tf.contrib.rnn.LSTMStateTuple

class tf.nn.rnn_cell.LSTMStateTuple

Defined in `tensorflow/python/ops/rnn_cell_impl.py`.

See the guide: RNN and Cells (contrib) > Classes storing split `RNNCell` state

Tuple used by LSTM Cells for `state_size`, `zero_state`, and output state.

Stores two elements: `(c, h)`, in that order.

Only used when `state_is_tuple=True`.

Exercise 1 - Tagger with static_rnn

- Make sure you have the newest Tensorflow version! (python3 -m pip install - -upgrade tensorflow)
- Extend the neural tagger from the previous exercises so that static_rnn is used, instead of a DNN
- As before, use a fixed length context window (e.g. left_context=4, right_context=0)
- We provide you with a new exercise file (05_dnn_tagger_lstm_ex.py): the solution for the tagger from previous exercises, which you need to extend.

Variable sequence lengths

- Using a fixed length RNN isn't all to practical for NLP
- Ideally, we would like to keep the sequence length dynamic
- E.g. change it from batch to batch, as needed
- Ideally we would like to operate on an input tensor shape like (None, None) - we don't know the batch size nor the sentence lengths in advance!
- That's what dynamic_rnn is for!

```
cell = tf.contrib.rnn.LSTMCell(num_hidden)
output, state = tf.nn.dynamic_rnn(cell, x_embedd,
  dtype=tf.float32, sequence_length=seq_len_in)
```

Variable sequence lengths

```
output, state = tf.nn.dynamic_rnn(cell, x_embedd,
  dtype=tf.float32, sequence_length=seq_len_in)
```

- output is a tensor, e.g. (?, ?, hidden_size)
- For LSTMs, state is again a tuple of c and h. This is dependent on what you use as cell! E.g. with GRUs, there is no separate cell state *c* in the cell.

Exercise 2 - Tagger with dynamic_rnn

- Extend the neural tagger from the previous exercises so that dynamic_rnn is used, instead of a DNN
- There is a new prepare_data_sentences function for preparing the data, which you must use. Instead of a window, one training example is now a complete sentence.
- Hints: you need to adapt inputs, also reshape the outputs of the rnn so that the loss function can be applied correctly.

Exercise 2 - Things to try

- Bidirectional RNNs (e.g. tf.nn.bidirectional_dynamic_rnn)
- Stacking RNNs (tf.contrib.rnn.MultiRNNCell, tf.contrib.rnn.stack_bidirectional_dynamic_rnn)
- Dropout for RNNs (tf.contrib.rnn.DropoutWrapper)