NATURAL LANGUAGE PROCESSING WITH DEEP LEARNING

Dr.Minaei, IUST, Fall 2020

Presenting by Sara Rajaee

Most materials of this mini-course are provided by "Natural Language Processing" course, Mohammad Taher Pilehvar, IUST, Fall 2019 and "Natural Language Processing with Deep Learning" course, Stanford, Winter 2020

REVIEW-RECURRENT NEURAL NETWORKS

- Dense has no memory!
- However, we read sentences word by word, keeping a memory of what came before
- RNNs process sequences by iterating through the sequence elements and maintaining a state containing information relative to what they have seen so far.

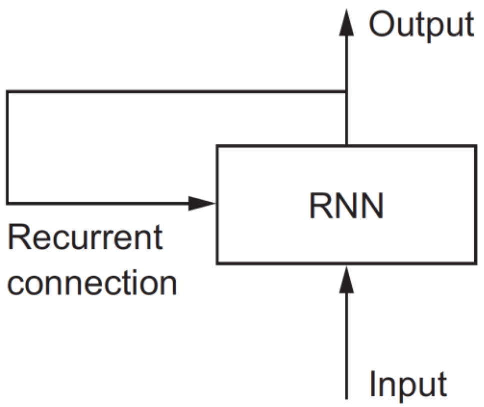


Figure 6.9 A recurrent network: a network with a loop

REVIEW-RECURRENT NEURAL NETWORKS

```
state_t = 0                          <— The state at t
for input_t in input_sequence:       <— Iterates over sequence elements
    output_t = f(input_t, state_t)
    state_t = output_t               <— The previous output becomes the state for the next iteration.
```

RECURRENT NEURAL NETWORKS

- You can even flesh out the function `f`: the transformation of the input and state into an output will be parameterized by two matrices, `W` and `U`, and a bias vector.

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

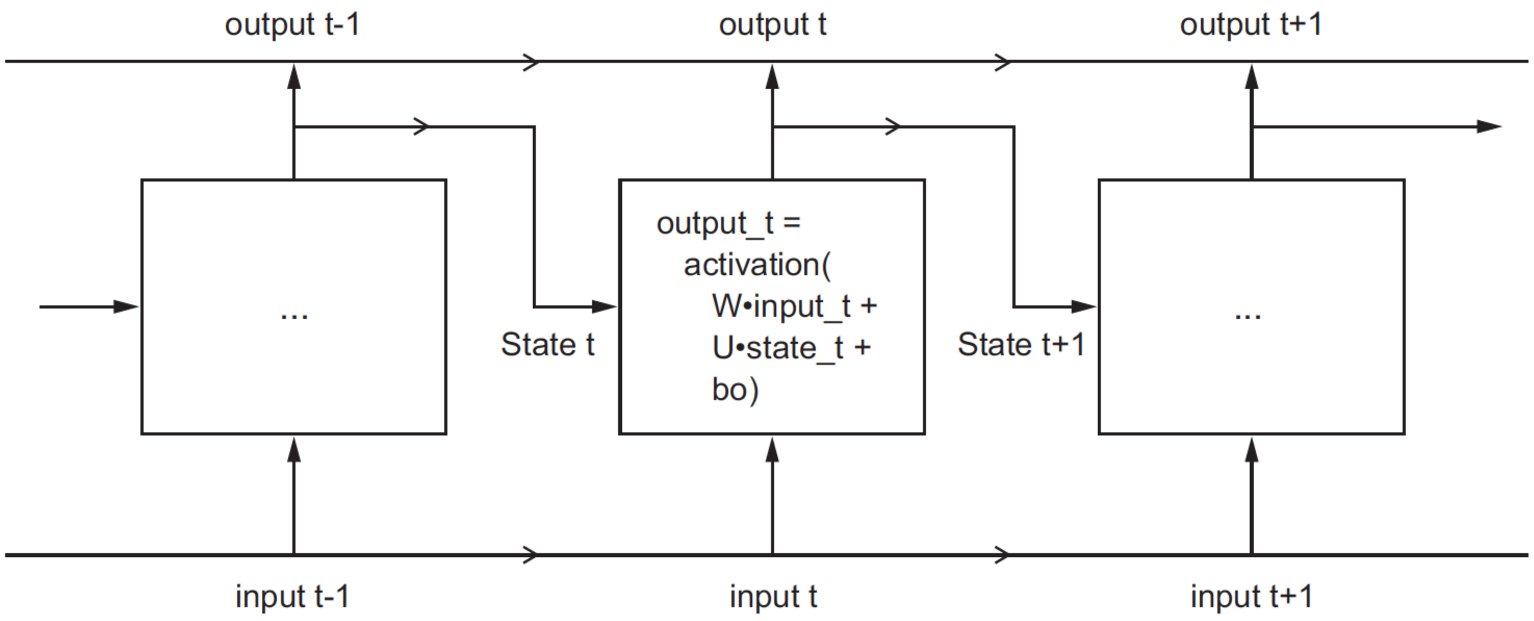
RNN

```
import numpy as np

timesteps = 100
input_features = 32
output_features = 64

inputs = np.random.random((timesteps, input_features))

state_t = np.zeros((output_features,))

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)

    successive_outputs.append(output_t)

    state_t = output_t

final_output_sequence = np.concatenate(successive_outputs, axis=0)
```

RECURRENT NEURAL NETWORKS

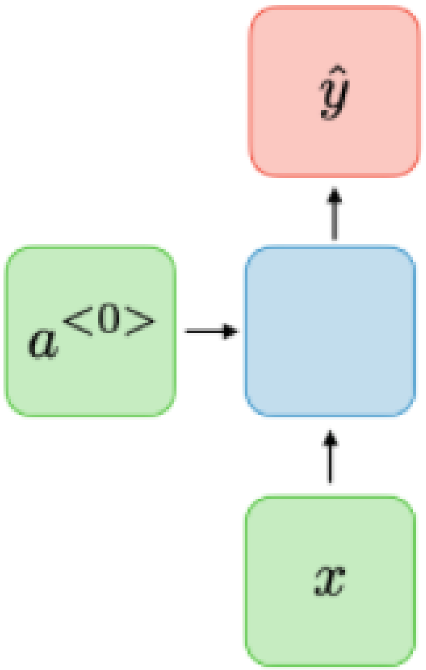
- RNNs are characterized by their step function, such as the following function in this case:

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```



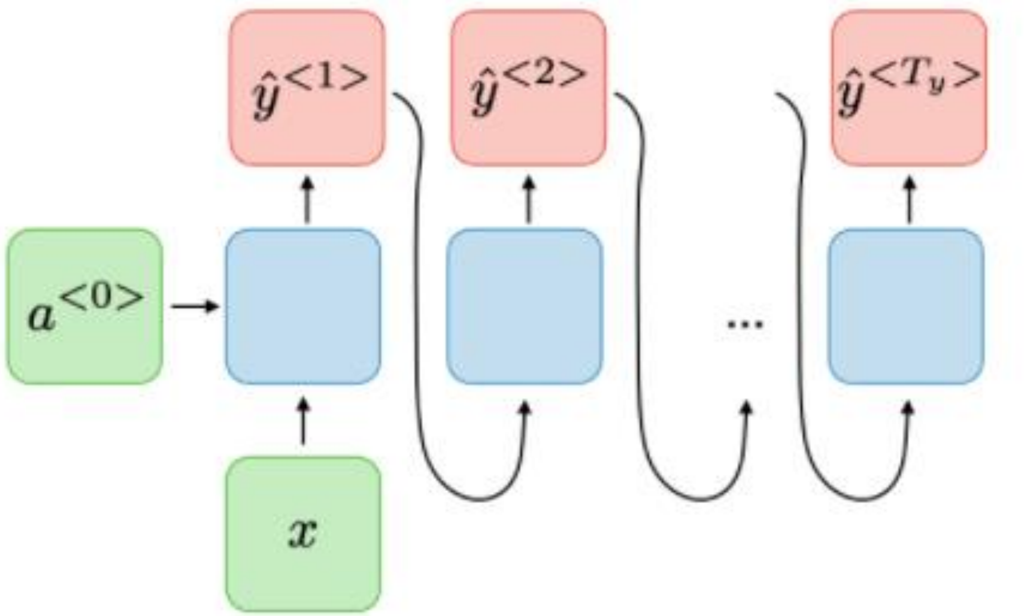
RECURRENT NEURAL NETWORKS
TYPES

- One-to-One
- Traditional neural network



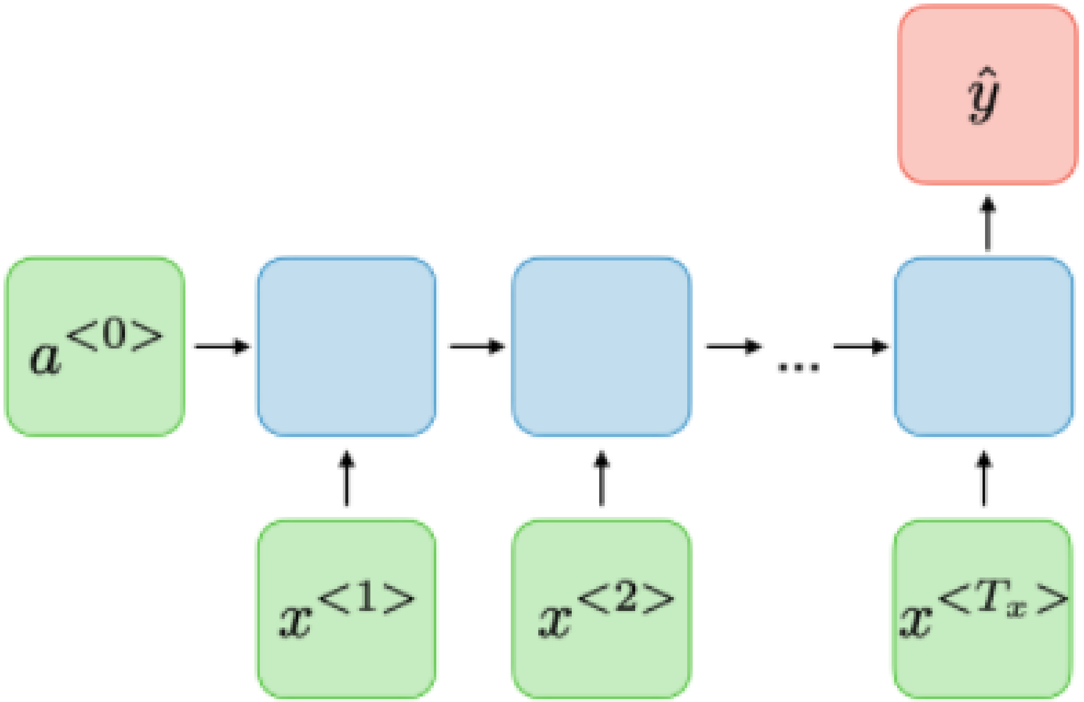
RECURRENT NEURAL NETWORKS
TYPES

- One-to-Many
- Music Generation



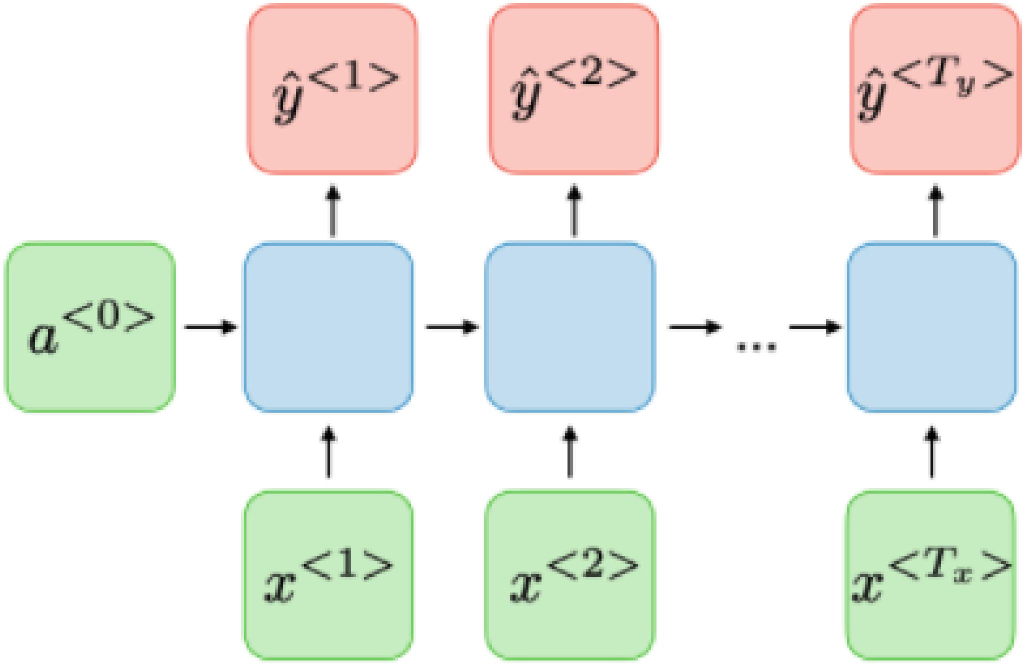
RECURRENT NEURAL NETWORKS
TYPES

- Many-to-One
- Sentiment Classification



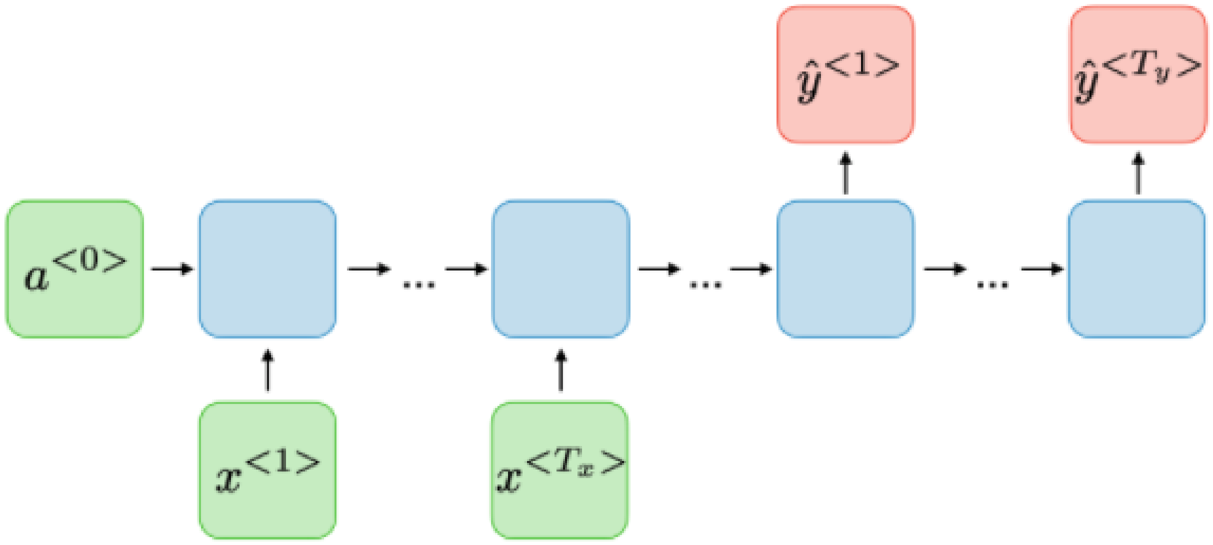
RECURRENT NEURAL NETWORKS
TYPES

- Many-to-Many
  - *input length* $=$ *output length*
- Named Entity Recognition



RECURRENT NEURAL NETWORKS
TYPES

- Many-to-Many
  - *input length* $\neq$ *output length*
- Machine Translation



RNN IN TENSORFLOW

- This implementation is `SimpleRNN` in Tensorflow, with one difference:
  - `SimpleRNN` processes batches of sequences, like all other layers, not a single sequence as in the Numpy example.
  - This means it takes inputs of shape `(batch_size, timesteps, input_features)`, rather than `(timesteps, input_features)`.

RNN IN TENSORFLOW

Two options for outpus:

- Return either the full sequences of successive outputs for each timestep
  - A 3D tensor of shape `(batch_size, timesteps, output_features)`
- Or only the last output for each input sequence
  - A 2D tensor of shape `(batch_size, output_features)`

RNN IN TENSORFLOW

Two options for outpus:

- Return either the full sequences of successive outputs for each timestep
  - A 3D tensor of shape `(batch_size, timesteps, output_features)`
- Or only the last output for each input sequence
  - A 2D tensor of shape `(batch_size, output_features)`

RNN IN TENSORFLOW

First Option

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 32)          320000
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, 32)                2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
_________________________________________________________________
```

RNN IN TENSORFLOW

Second Option

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences = True))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, None, 32)          2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
_________________________________________________________________
```

RNN IN TENSORFLOW

- It is sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network.

```
[8]
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences = True))
model.add(SimpleRNN(32, return_sequences = True))
model.add(SimpleRNN(32, return_sequences = True))
model.add(SimpleRNN(32, return_sequences = True))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simple_rnn_5 (SimpleRNN)     (None, None, 32)          2080
=================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
_________________________________________________________________
```

RNNs

- Advantages
  - Possibility of processing input of any length
  - Model size not increasing with size of input
  - Computation takes into account historical information
  - Weights are shared across time
- Disadvantages
  - Computation being slow
  - Difficulty of accessing information from a long time ago
  - Cannot consider any future input for the current state

SIMPLE RNN
TENSORFLOW 2.0

LSTM AND GRU LAYERS

SimpleRNN is generally too simplistic to be of real use.

- Although it should theoretically be able to retain at time `t` information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn.

VANISHING GRADIENT

- An effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable.

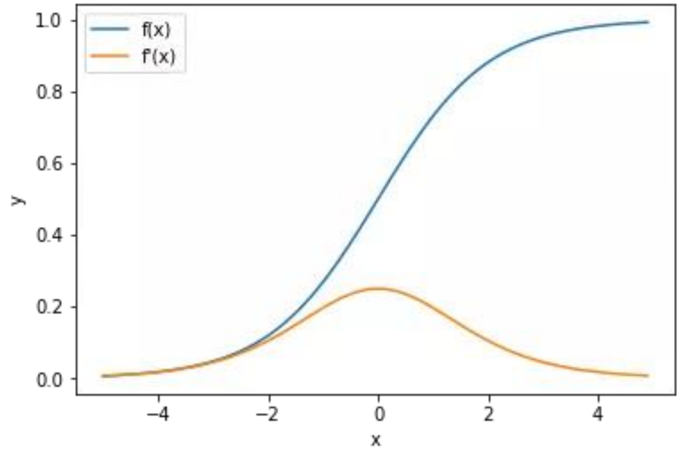
VANISHING GRADIENT

Arises due to the nature of the back-propagation optimization

- The weight and bias values in the various layers within a neural network are updated each optimization iteration by stepping in the direction of the *gradient* of the weight/bias values with respect to the loss function.

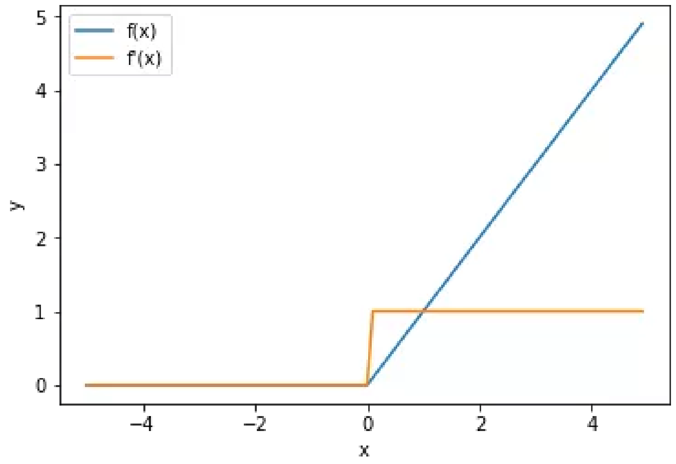
Problematic for deep feedforward networks

- Particularly with sigmoid activation functions.
- When the sigmoid function value is either too high or too low, the derivative (orange line) becomes very small i.e. << 1.



VANISHING GRADIENT

- ReLU activation can partly solve the problem
- No degradation of the error signal
- But, certain weights can be cancelled out whenever there is a negative input into a given neuron

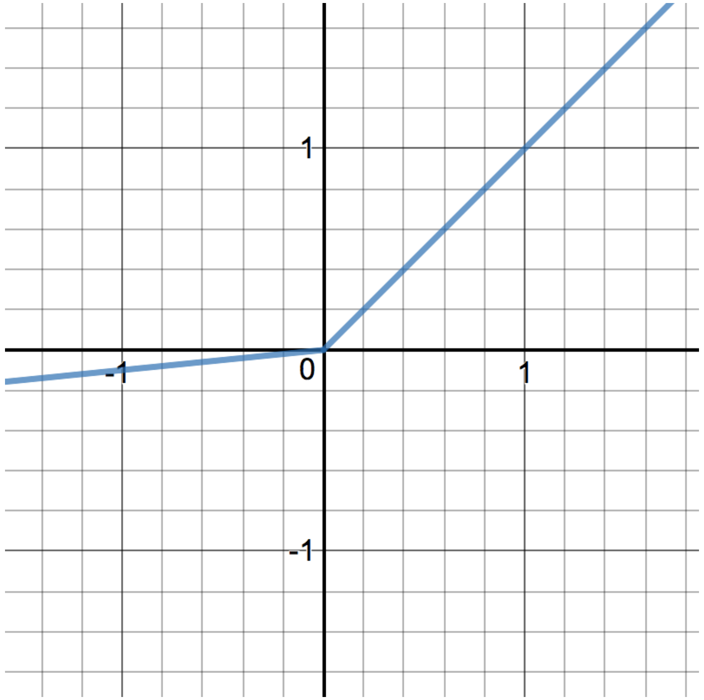


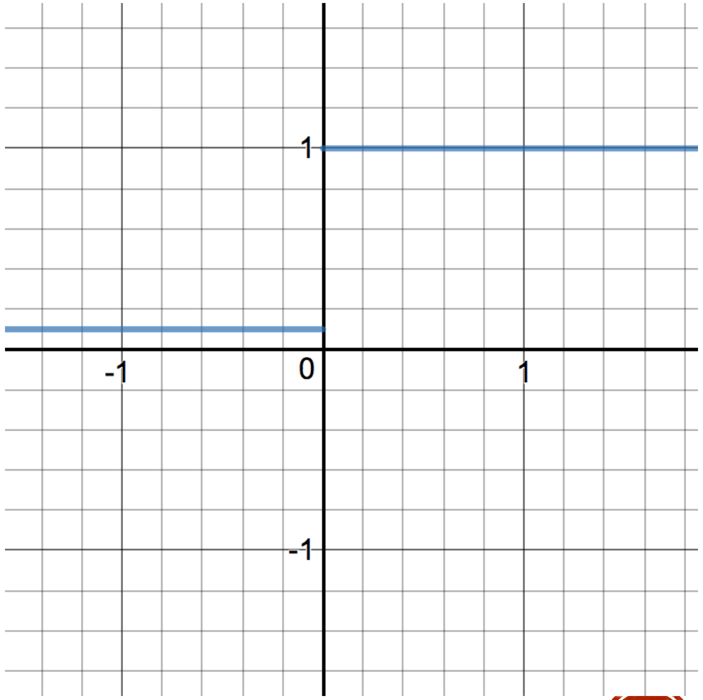
VANISHING GRADIENT

- Leaky ReLU

$$f(x) = \max(\alpha x, x)$$

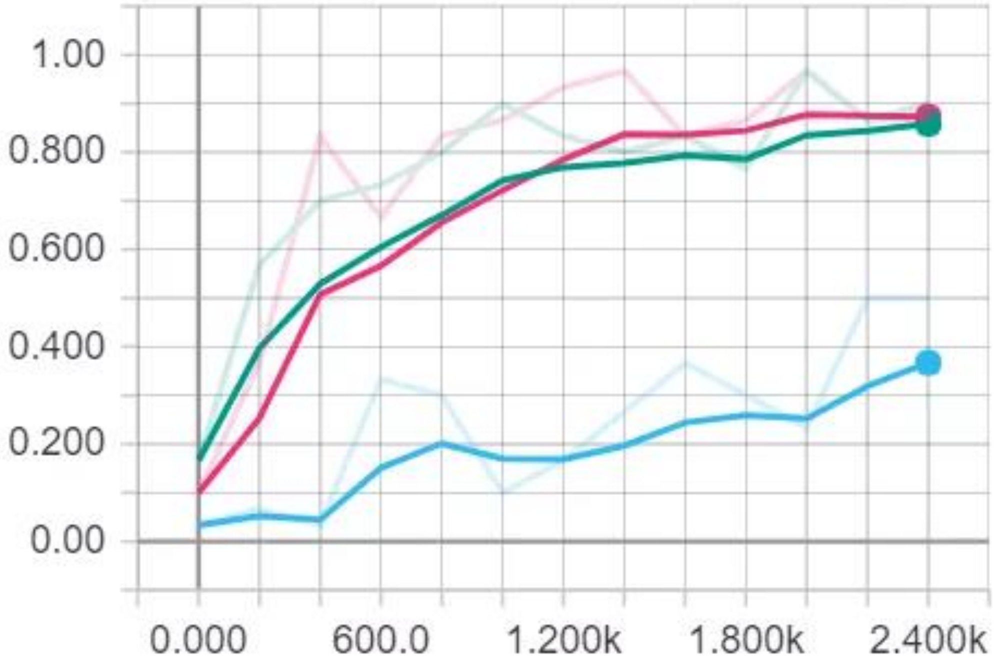
- The good thing about the Leaky ReLU activation function is that the derivative when x is below zero is alpha – i.e. it is a small but no longer 0.





VANISHING GRADIENTS (IN PRACTICE)

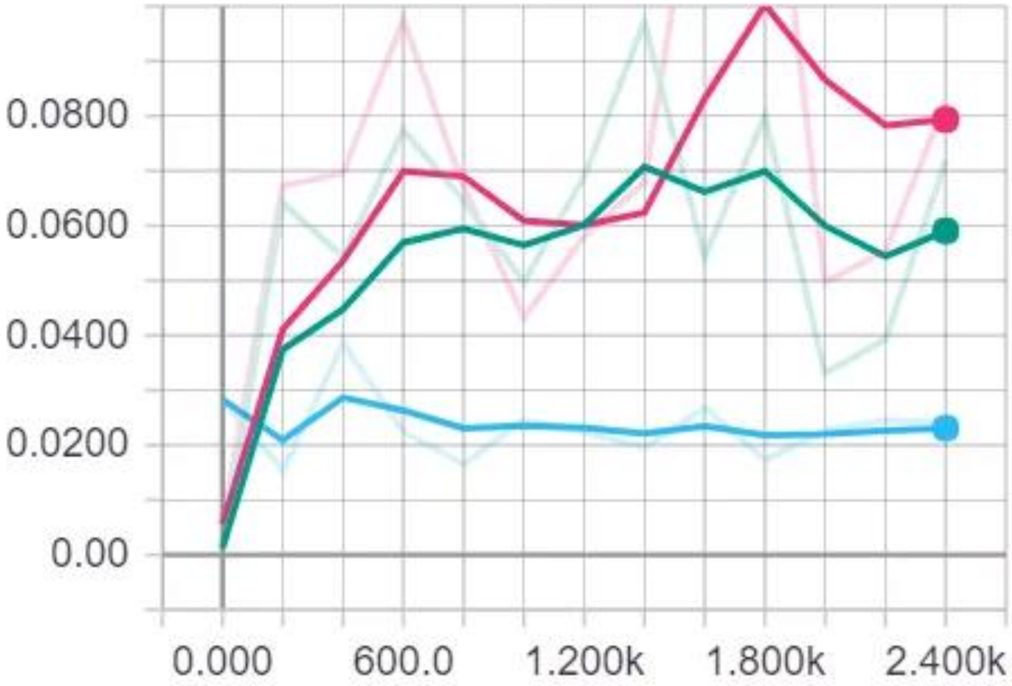
- MNIST using a 7-layer densely connected network



Accuracy of the three activation scenarios
sigmoid (blue),
ReLU (red),
Leaky ReLU (green)

VANISHING GRADIENTS (IN PRACTICE)

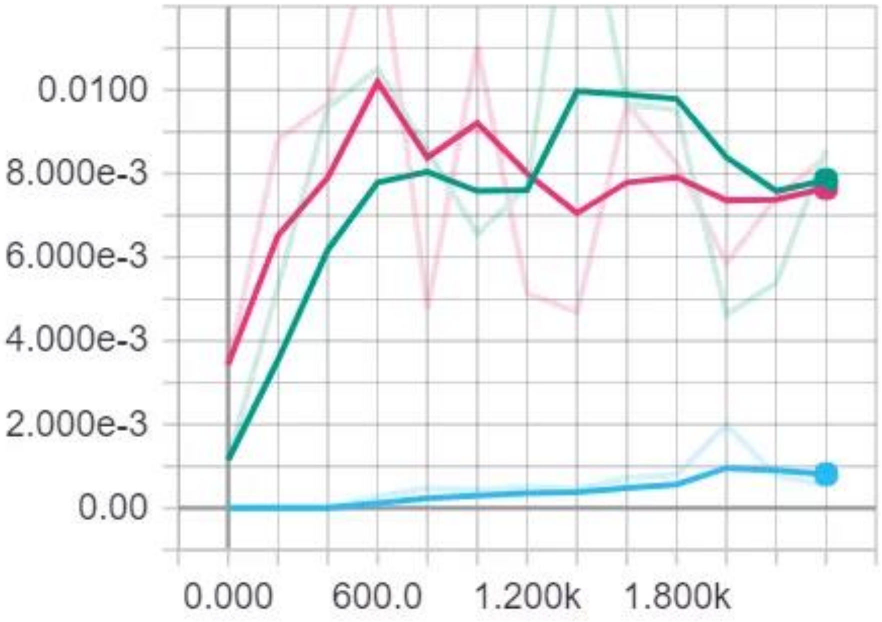
- MNIST using a 7-layer densely connected network
- Mean absolute gradient logs during training



Mean absolute gradients
Output layer (6th layer)
sigmoid (blue),
ReLU (red),
Leaky ReLU (green)

VANISHING GRADIENTS (IN PRACTICE)

- MNIST using a 7-layer densely connected network
- Mean absolute gradient logs during training



Mean absolute gradients
1st layer
sigmoid (blue),
ReLU (red),
Leaky ReLU (green)

FROM SIMPLERNN TO LSTM

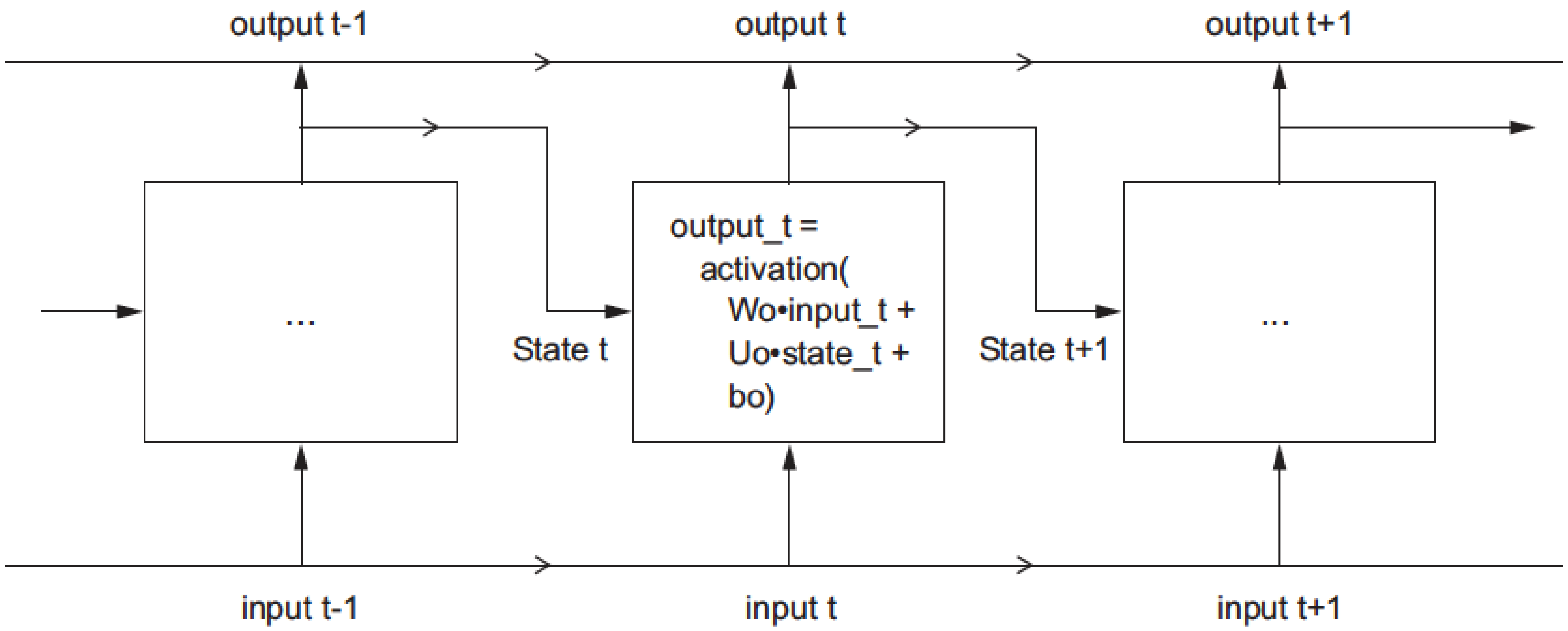


Figure 6.13 The starting point of an LSTM layer: a SimpleRNN

FROM SIMPLERNN TO LSTM

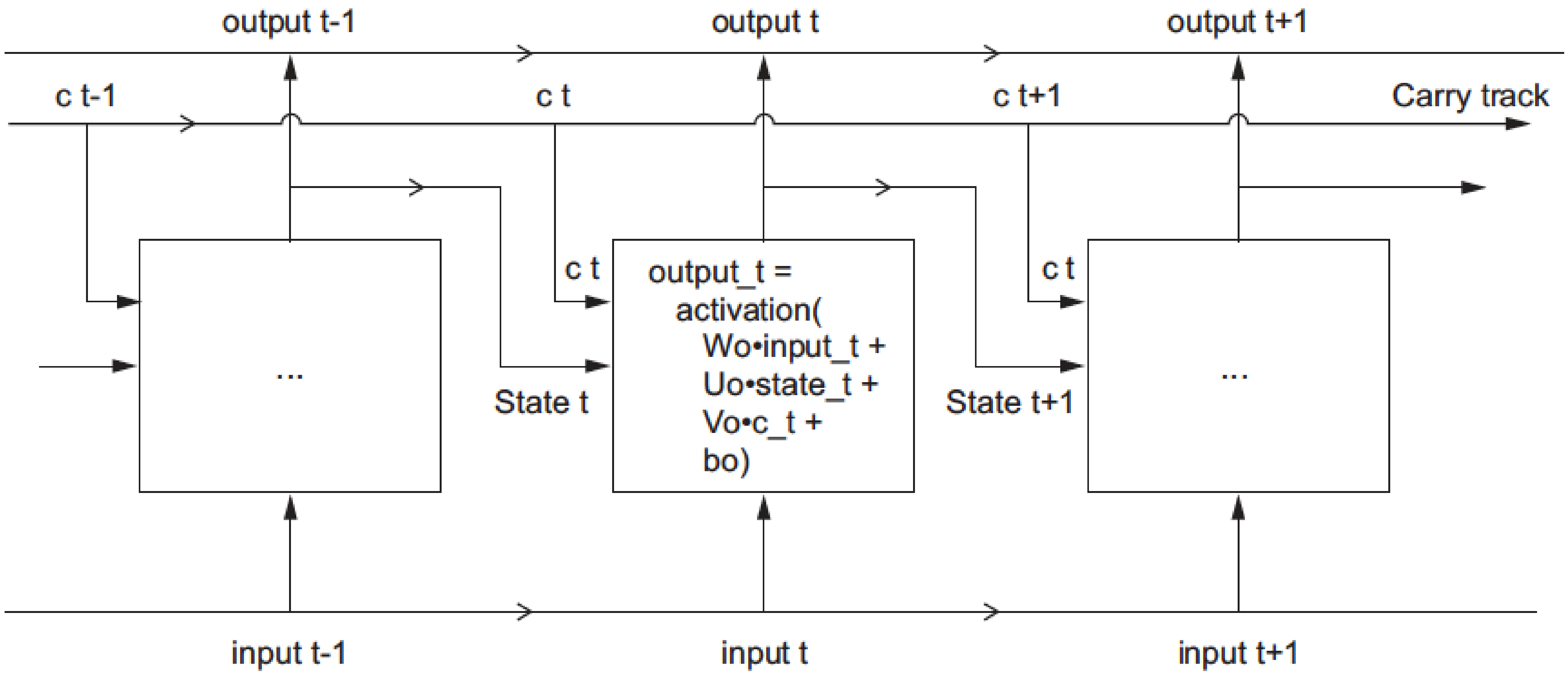


Figure 6.14 **Going from a `SimpleRNN` to an `LSTM`: adding a carry track**

FROM SIMPLERNN TO LSTM

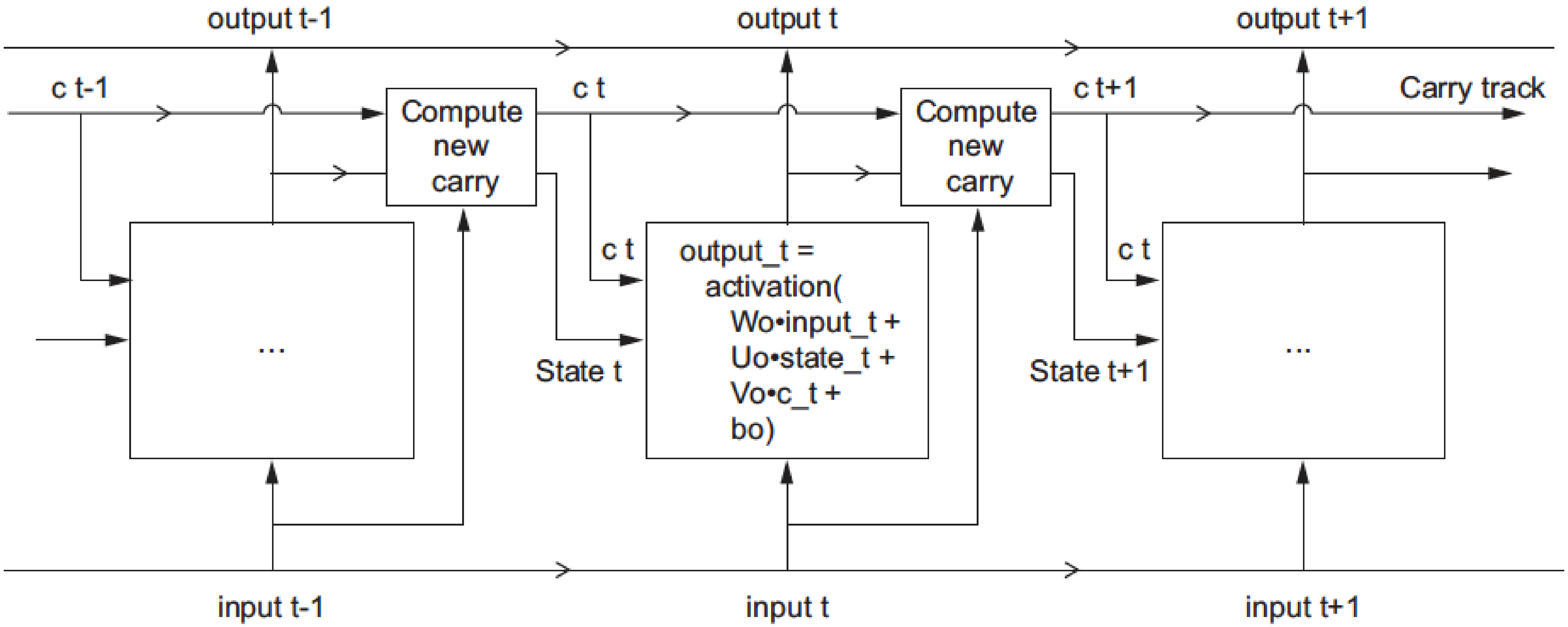
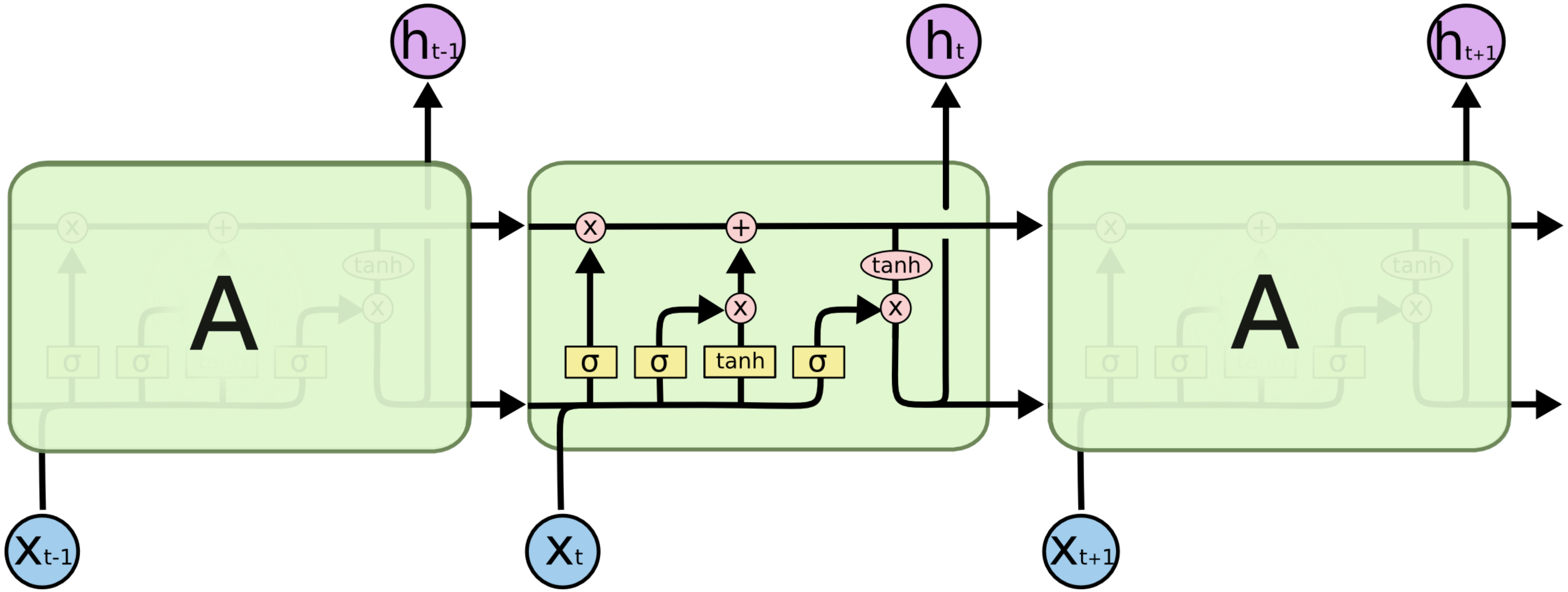


Figure 6.15 Anatomy of an LSTM

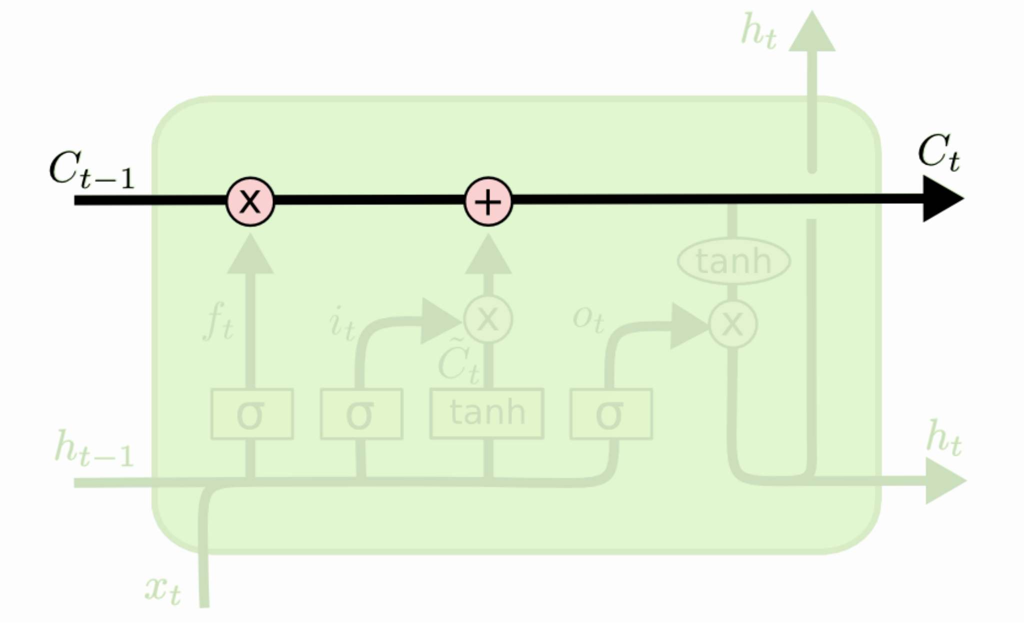
LSTMs



LSTMs

Cell state

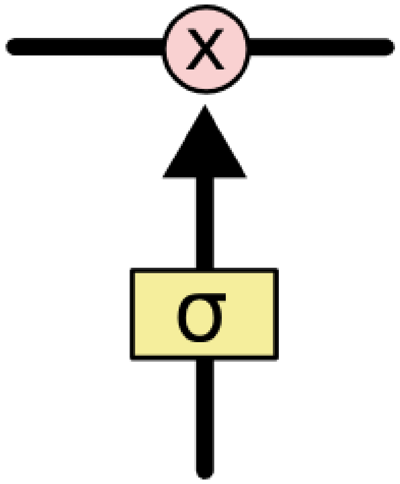
- kind of like a conveyor belt



- It runs straight down the entire chain, with only some minor linear interactions.

- It's very easy for information to just flow along it unchanged.

LSTMs



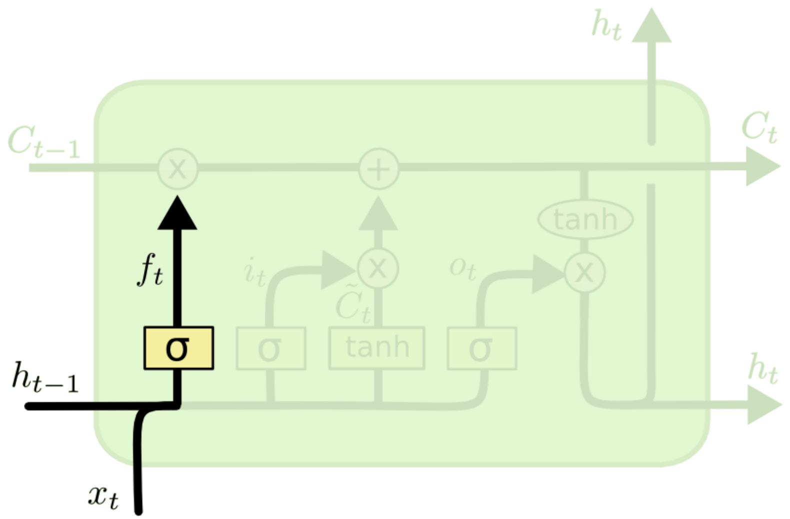
Gates

- A way to optionally let information through.
- They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
  - A value of zero means “let nothing through,” while a value of one means “let everything through!”

LSTMs

Forget gate layer

- What information we're going to throw away from the cell state?



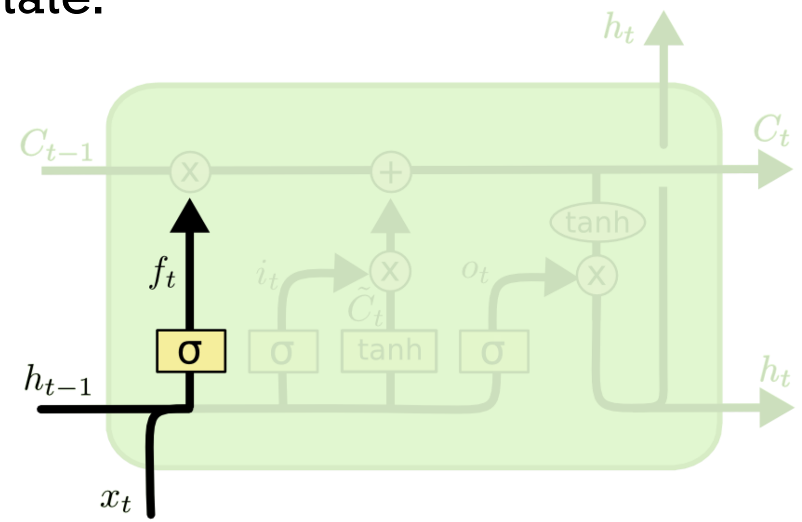
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

LSTMs

What new information we're going to store in the cell state?

Two parts:

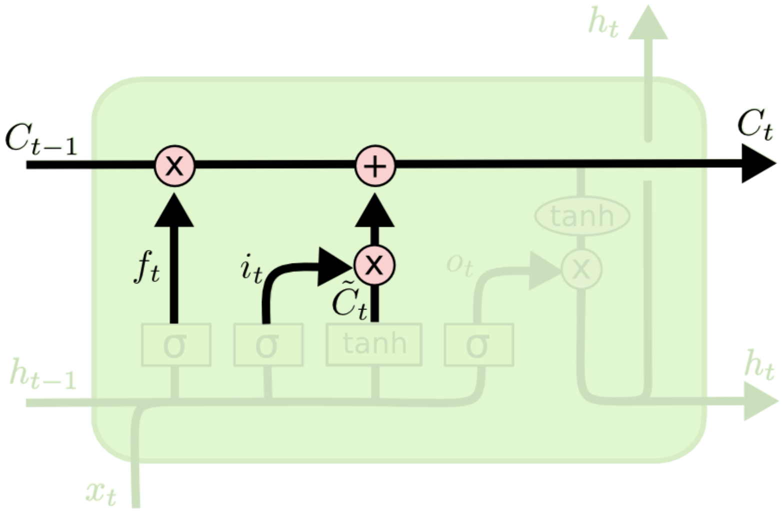
1. A sigmoid layer called the *input gate layer* decides which values we'll update.
2. A tanh layer creates a vector of new candidate values, that could be added to the state.



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

LSTMs

It's now time to update the old cell state C_{t-1} to C_t

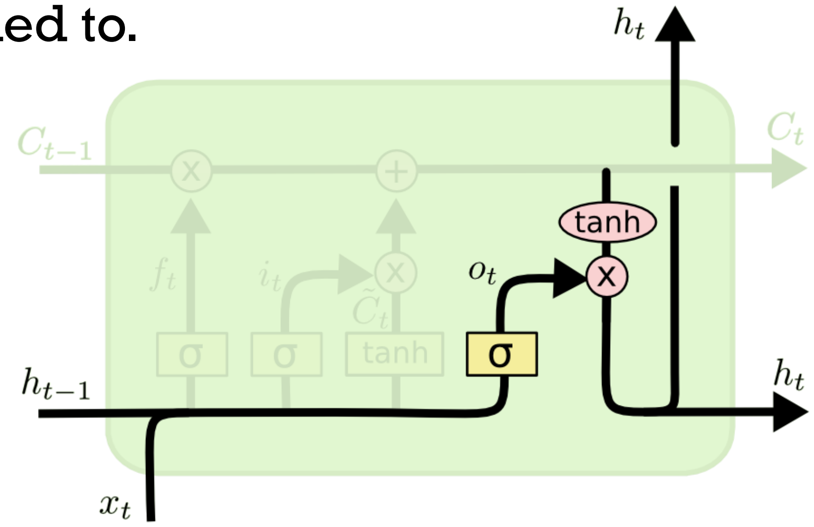


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTMs

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version.

1. A sigmoid layer to decide what parts of the cell state to output.
2. Put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

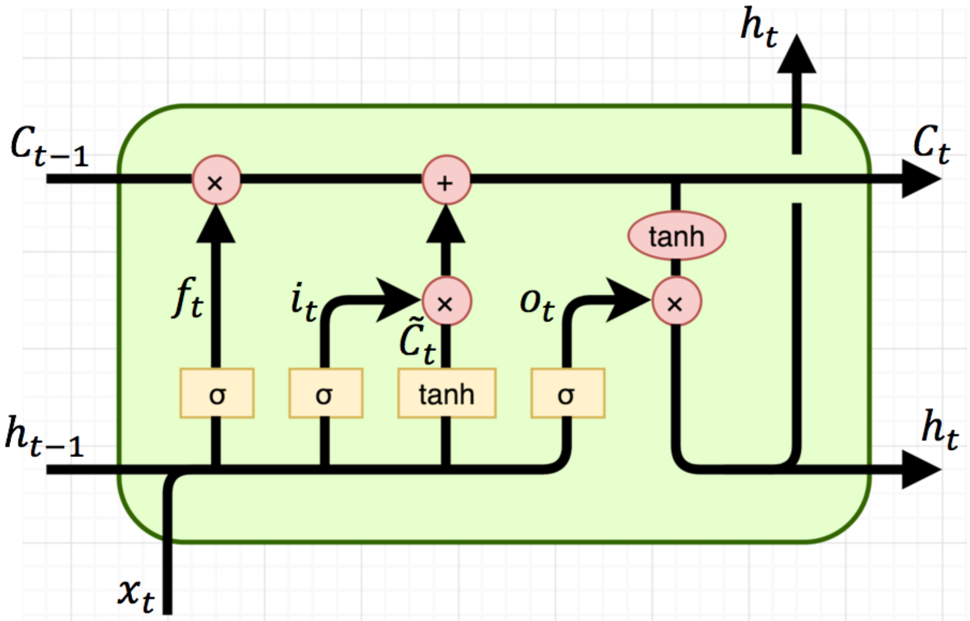


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

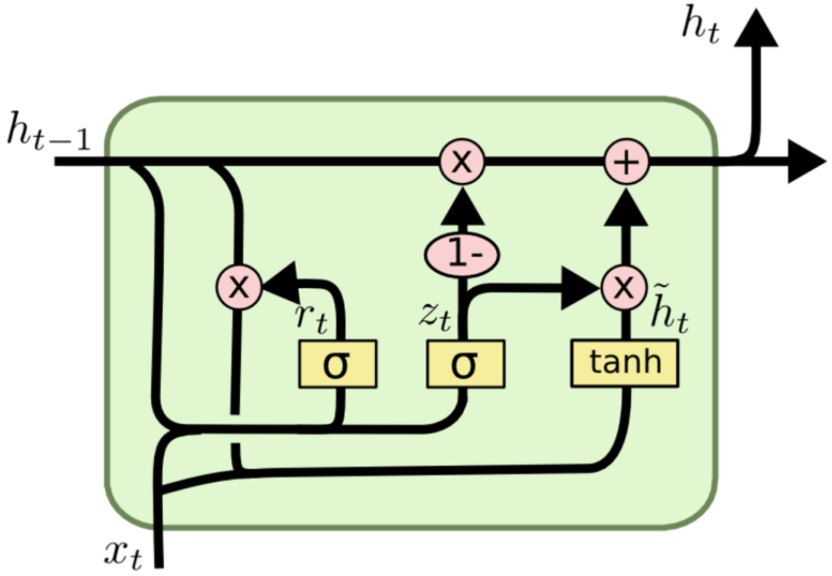
$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM
TENSORFLOW 2.0

GRU: GATED RECURRENT UNITS

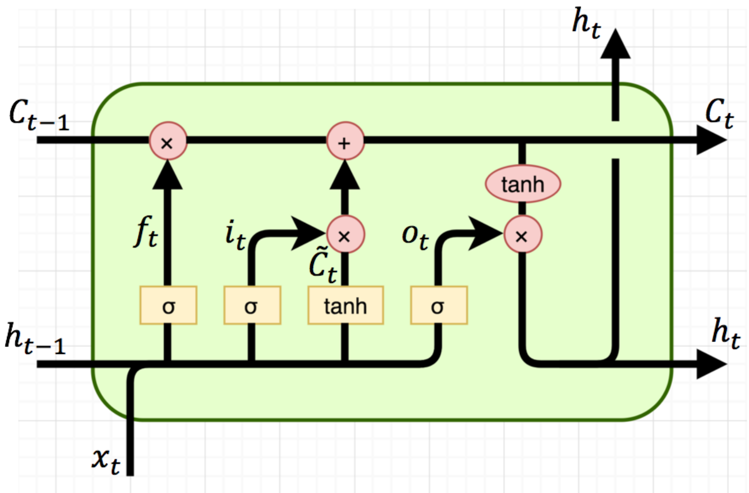


(a) Long Short-Term Memory

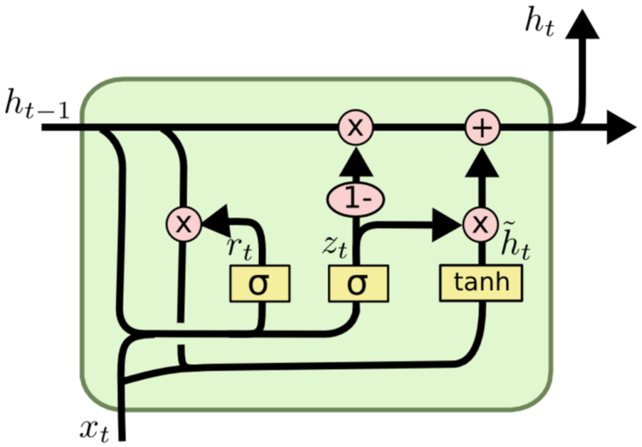


(b) Gated Recurrent Unit

GRU: GATED RECURRENT UNITS



(a) Long Short-Term Memory



(b) Gated Recurrent Unit

$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right)$$
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right)$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$
$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right)$$
$$h_t = \tanh(C_t) * o_t$$

$$z_t = \sigma\left(x_t U^z + h_{t-1} W^z\right)$$
$$r_t = \sigma\left(x_t U^r + h_{t-1} W^r\right)$$
$$\tilde{h}_t = \tanh\left(x_t U^h + (r_t * h_{t-1}) W^h\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

DROPOUT VS. RECURRENT DROPOUT

```
tf.keras.layers.LSTM(
    units, activation='tanh', recurrent_activation='sigmoid', use_bias=True,
    kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
    bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
    recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, recurrent_constraint=None, bias_constraint=None,
    dropout=0.0, recurrent_dropout=0.0, implementation=2, return_sequences=False,
    return_state=False, go_backwards=False, stateful=False, time_major=False,
    unroll=False, **kwargs
)
```

- Regular dropout masks the inputs
  - Add a `Dropout` layer after the recurrent layer if you want to mask the outputs.

- Recurrent dropout masks the connections between the recurrent units (the cell state)

STACKING RNN LAYERS

- It is a generally a good idea to increase the capacity of your network until overfitting becomes your primary obstacle (assuming that you are already taking basic steps to mitigate overfitting, such as using dropout).

- Recurrent layer stacking is a classic way to build more powerful recurrent networks:
  - For instance, what currently powers the Google translate algorithm is a stack of seven large LSTM layers -- that's huge.

STACKING RNN LAYERS

- To stack recurrent layers on top of each other in Tensorflow, all intermediate layers should return their full sequence of outputs (a 3D tensor) rather than their output at the last timestep.

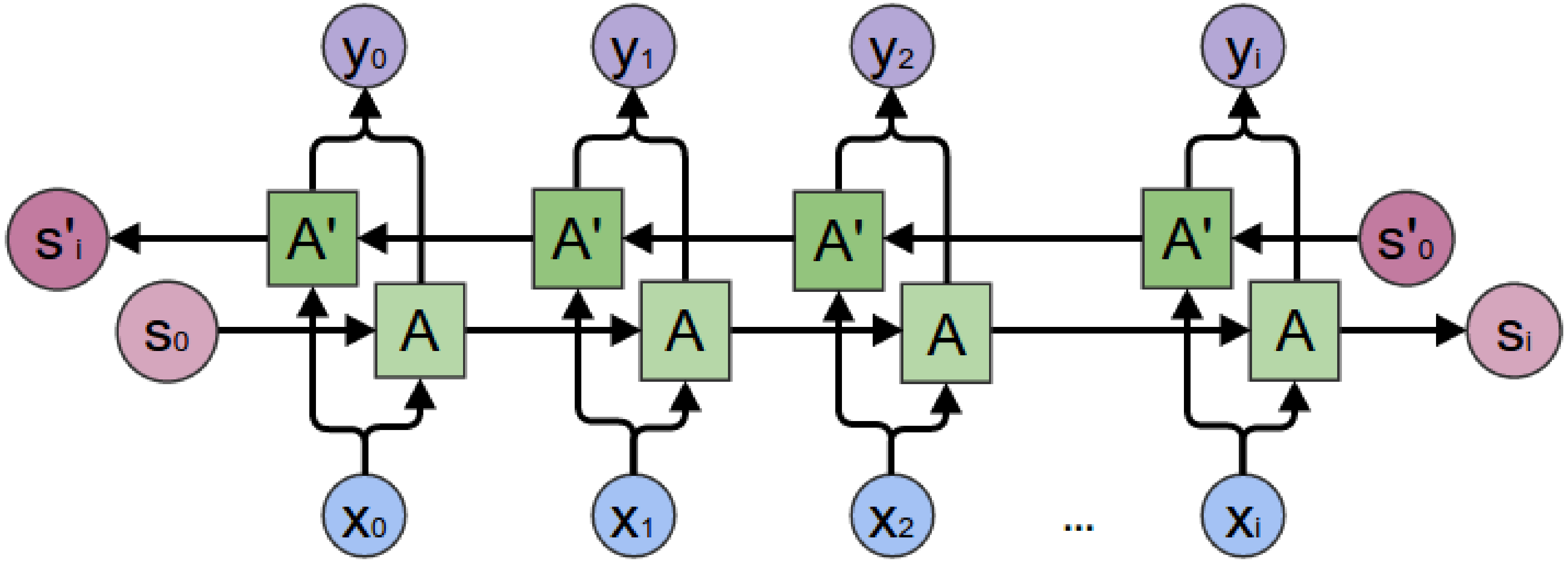
- This is done by specifying `return_sequences=True`:

BIDIRECTIONAL RNNs

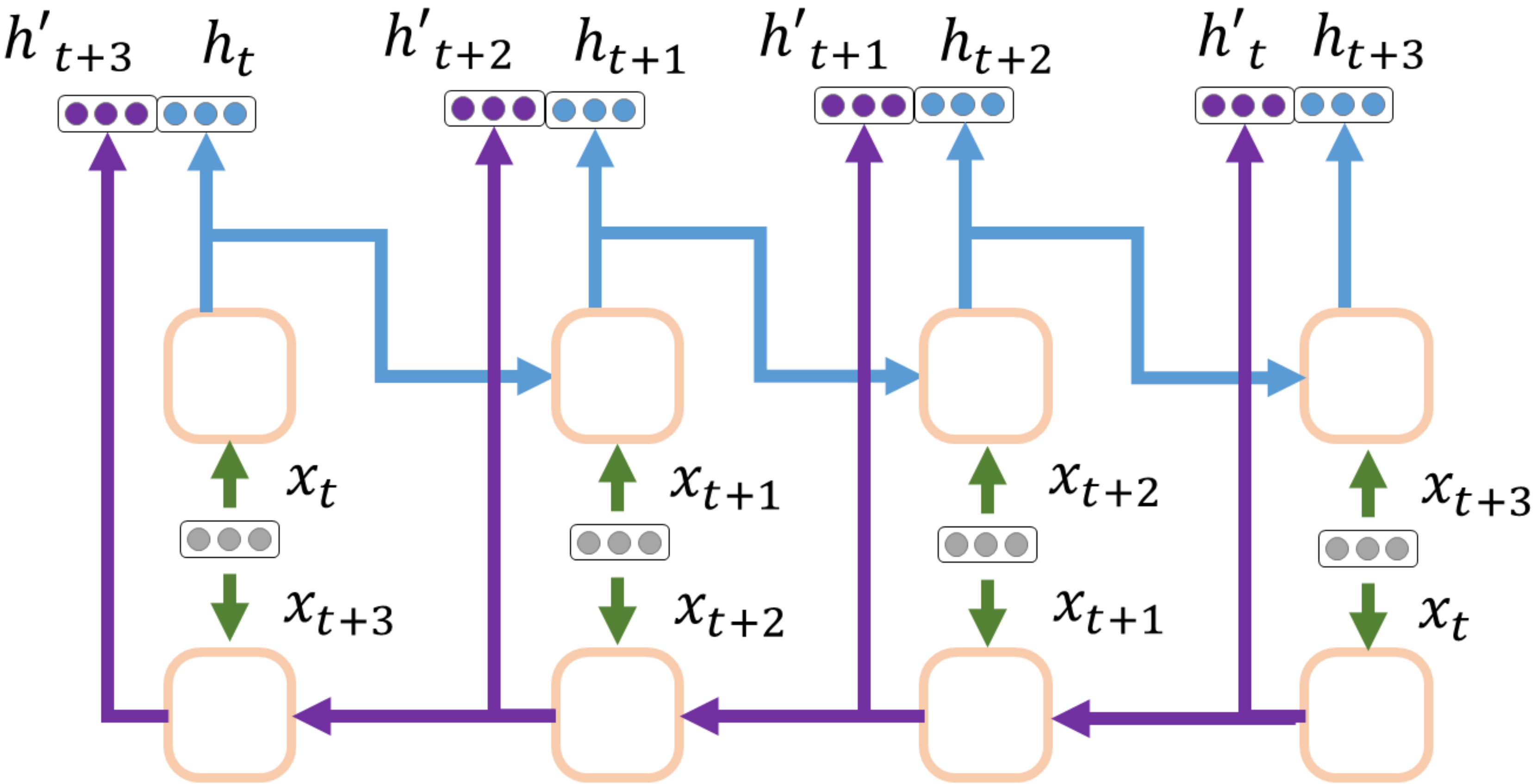
- A bidirectional RNN is common RNN variant which can offer higher performance than a regular RNN on certain tasks.

- It is frequently used in natural language processing -- you could call it the Swiss army knife of deep learning for NLP.

BIDIRECTIONAL RNNs



BIDIRECTIONAL RNNs



BIDIRECTIONAL RNNs

- A bidirectional RNN exploits the order-sensitivity of RNNs.
- It simply consists of two regular RNNs, such as the GRU or LSTM layers that you are already familiar with, each processing input sequence in one direction (chronologically and anti-chronologically), then merging their representations.

- By processing a sequence both way, a bidirectional RNN is able to catch patterns that may have been overlooked by a one-direction RNN.

WRAP UP

- It is good to first establish *common sense baselines* for your metric of choice. If you don't have a baseline to beat, you can't tell if you are making any real progress.
- Try *simple models* before expensive ones, to justify the additional expense. Sometimes a simple model will turn out to be your best option.
- On data where *temporal ordering* matters, recurrent networks are a great fit and easily outperform models that first flatten the temporal data.
- *Stacked RNNs* provide more representational power than a single RNN layer. They are also much more expensive, and thus not always worth it.
- *Bidirectional RNNs*, which look at a sequence both ways, are very useful on natural language processing problems.

REFERENCES AND FURTHER RESOURCES

Websites:

1. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks
2. https://www.youtube.com/watch?v=qhXZsFVxGKo
3. https://medium.com/@himanshuxd/activation-functions-sigmoid-relu-leaky-relu-and-softmax-basics-for-neural-networks-and-deep-8d9c70eed91e
4. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html
5. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
6. https://isaacchanghau.github.io/post/lstm-gru-formula/