UNIVERSITÀ DELLA CALABRIA

Time Series

Examples

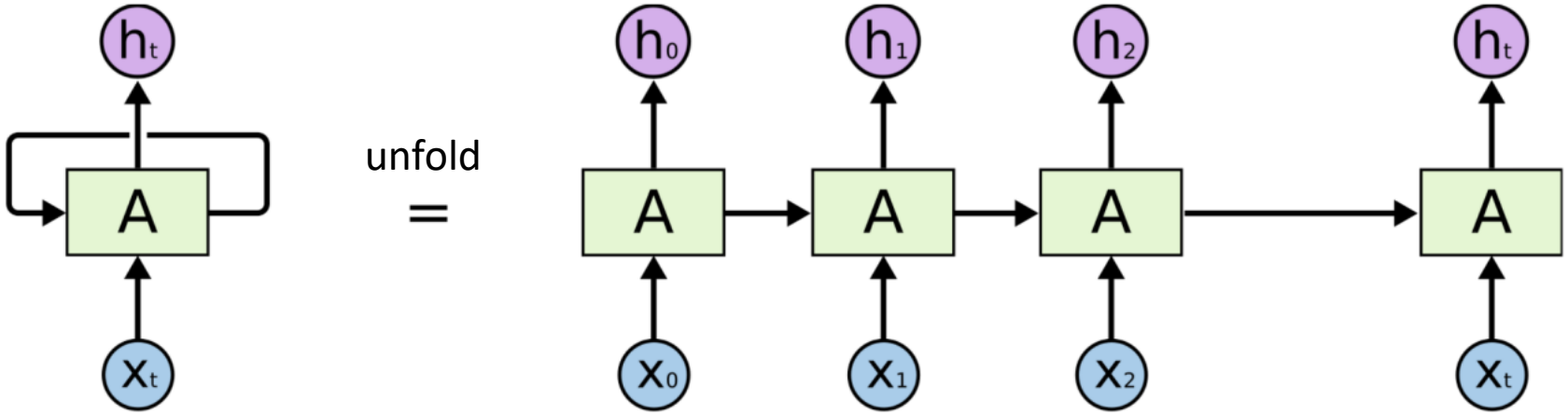
- Stock price prediction
  - A correct stock price prediction requires a lot of past information; it's impossible to predict the price evolution without knowing its previous trends
- Text generation
  - Text classification
  - Translation
  - Chatbot
  - …

Recurrent Networks

- Feedforward networks don’t consider temporal states
  - Sequences must be passed entirely, not word by word
- A RNN processes sequences by iterating over the elements and by maintaining a state containing information about what has been seen so far

Recurrent Neural Networks

- A RNN is a graph with cycles
  - Perceptrons can benefit from feedback loops
  - The output of a perceptron at time t is involved in its input at time $t + 1$



Equations

- Elman network

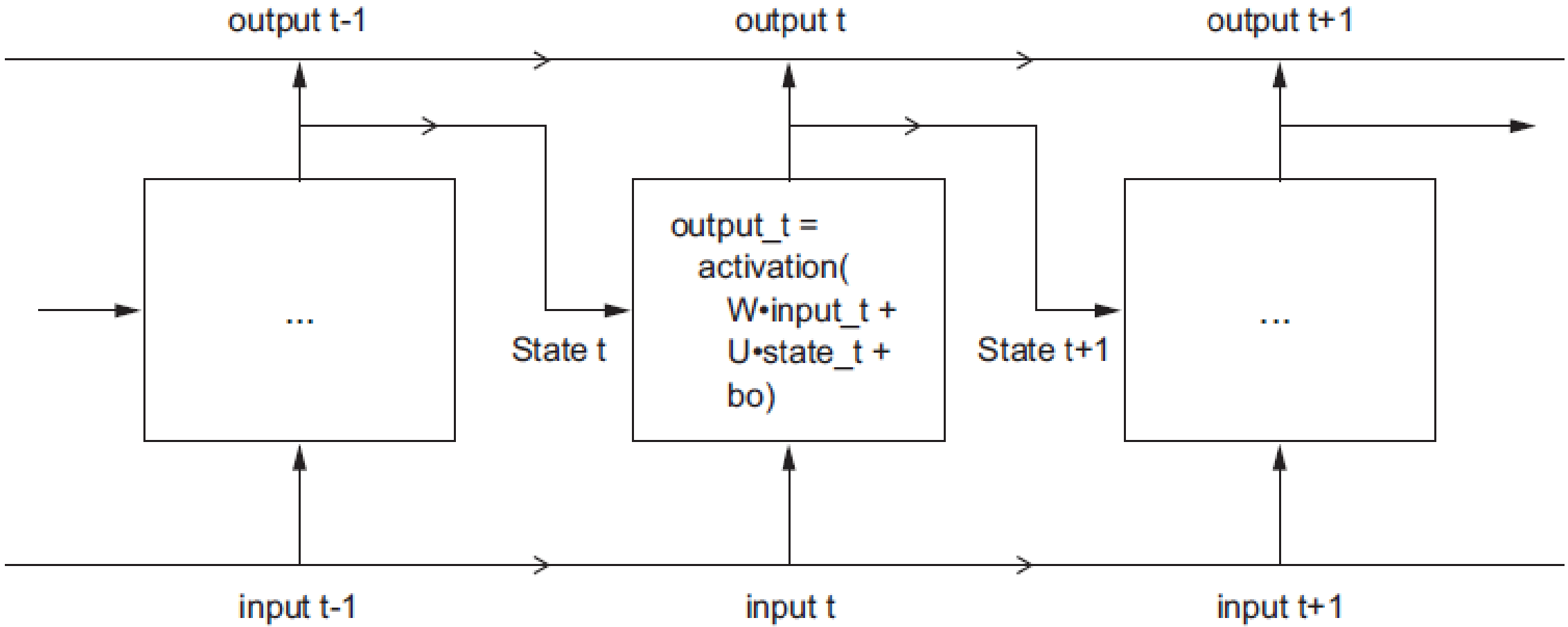
$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

- Jordan network

$$h_t = \sigma_h(W_h x_t + U_h y_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

$\boldsymbol{x_t}$: input vector
$\boldsymbol{y_t}$: output vector
$\boldsymbol{h_t}$: hidden layer vector
$\boldsymbol{W}$, $\boldsymbol{U}$, $\boldsymbol{b}$: weights and bias
$\boldsymbol{\sigma_h}$, $\boldsymbol{\sigma_y}$: activation functions

Unroll RNN


output t-1
output t
output t+1
...
State t
output_t =
activation(
W•input_t +
U•state_t +
bo)
State t+1
...
input t-1
input t
input t+1

Implementation

```
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)
```

Number of timesteps in the input sequence

Dimensionality of the input feature space

Dimensionality of the output feature space

Input data: random noise for the sake of the example

Initial state: an all-zero vector

Creates random weight matrices

input_t is a vector of shape (input_features,).

Combines the input with the current state (the previous output) to obtain the current output

Stores this output in a list

Updates the state of the network for the next timestep

The final output is a 2D tensor of shape (timesteps, output_features).

Usage

```
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))  # This last layer only returns the last output
model.summary()
```

These layers returns a list

Example

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000  # number of words to consider as features
maxlen = 500
batch_size = 32

(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)

input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
```

Example

```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(16))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Backpropagation Through Time

- Similarly to any neural network, RNNs needs to learn the weight values according to a training set
- The learning algorithm, namely backpropagation, needs a slightly change due to the feedback loops
- Its variant is called Backpropagation through time (BPTT)

Backpropagation Through Time

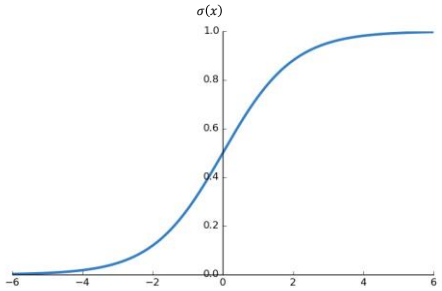
- The idea is very simple:

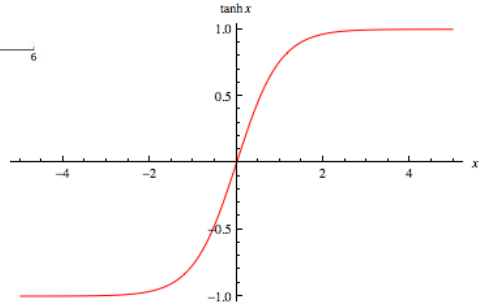
1. Unfold the recurrent neural network in time
   - The unfolded neurons share the same parameters
2. Backpropagate the loss gradient like in a feedforward neural network

Vanishing gradient

- According to the number of time steps, an unfolded RNN may be a very long sequence

- To simulate neuron activations, perceptrons are typically equipped with close-to-step activation functions:
  - Sigmoid, $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent, $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$





Vanishing gradient

- Stochastic gradient descent:

$$w^* = w - \eta \nabla E_j\big(o, f(i, w)\big)$$

- where:
  - w^*, the new weights
  - w, the old weights
  - η, the learning rate
  - $E_j(\cdot)$ the error (loss) function of the j-th observation
  - f, the activation function
  - ∇, the gradient operator
  - i, the input
  - o, the output

Vanishing gradient

- In backpropagation we have:

$$\nabla E_j\big(o, f_l(i, w_l)\big) = \nabla E_j\big(o, f_l(f_{l-1}(f_{l-2}(f_{l-3}(\dots), w_{l-2}), w_{l-1}), w_l)\big)$$

where l is a layer out of L

- For a recurrent perceptron (feedback loop)

$$f_l = f_{l-1} = f_{l-2} = \cdots = f$$

Vanishing gradient

- Hence a RNN perceptron has gradient:

$$\nabla E_j\big(o, f(f(f(f(\dots), w), w), w), w\big)$$

- By chain rule of derivatives we know:

$$\frac{d}{dx} f(g(x)) = f'\big(g(x)\big) \cdot g'(x)$$

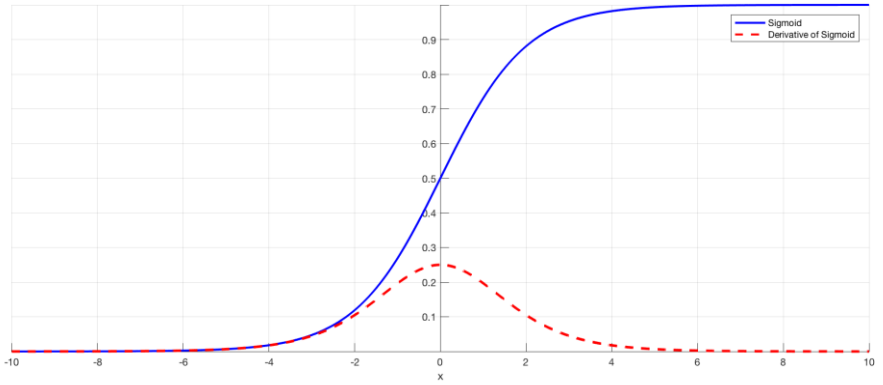
- This means that in a RNN we need to multiply the same derivative function many times:

$$[f'(i)]^T$$

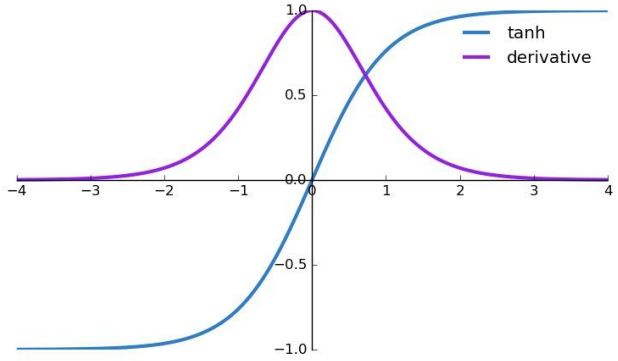
- where T is the number of time steps

Vanishing gradient

- The derivative of the sigmoid is:
  - $\sigma'(x) = \sigma(x) \cdot [1 - \sigma(x)]$



- The derivative of the tanh is:
  - $\tanh(x) = 1 - \tanh^2(x)$



Vanishing gradient

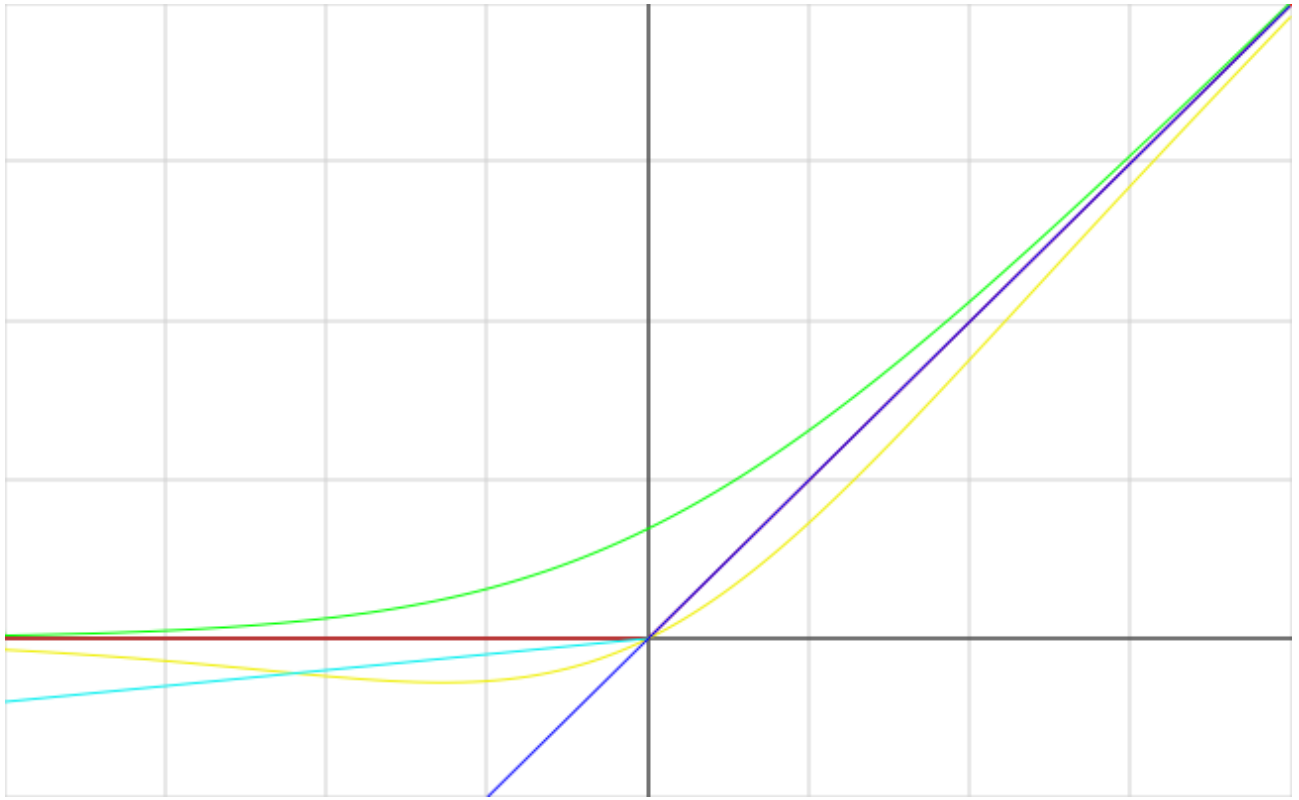
- In both cases we have quantities that are at most 1
- In $[f'(i)]^T$, let's assume that $f'(i) = 0.99$ and $T = 300$

$$0.99^{300} = 0.049$$

- The gradient is close to 0, it is disappearing… it is vanishing!
  - The network is not able to find out the optimal values for the weights

Possible solutions

- Changing the activation function
  - Linear
    - $Linear(x) = x$
  - REctified Linear Unit (RELU)
    - $RELU(x) = \begin{cases} x & if\ x \geq 0 \\ 0 & otherwise \end{cases}$
  - Leaky ReLU
    - $LRELU(x; \alpha \ll 1) = \begin{cases} x & if\ x \geq 0 \\ \alpha x & otherwise \end{cases}$
  - Softplus
    - $\text{Softplus}(x) = \ln(1 + e^x)$
  - Swish
    - $\text{Swish}(x) = x \cdot \sigma(x)$
- There is no control in what to remember
  - The long-term dependency problem

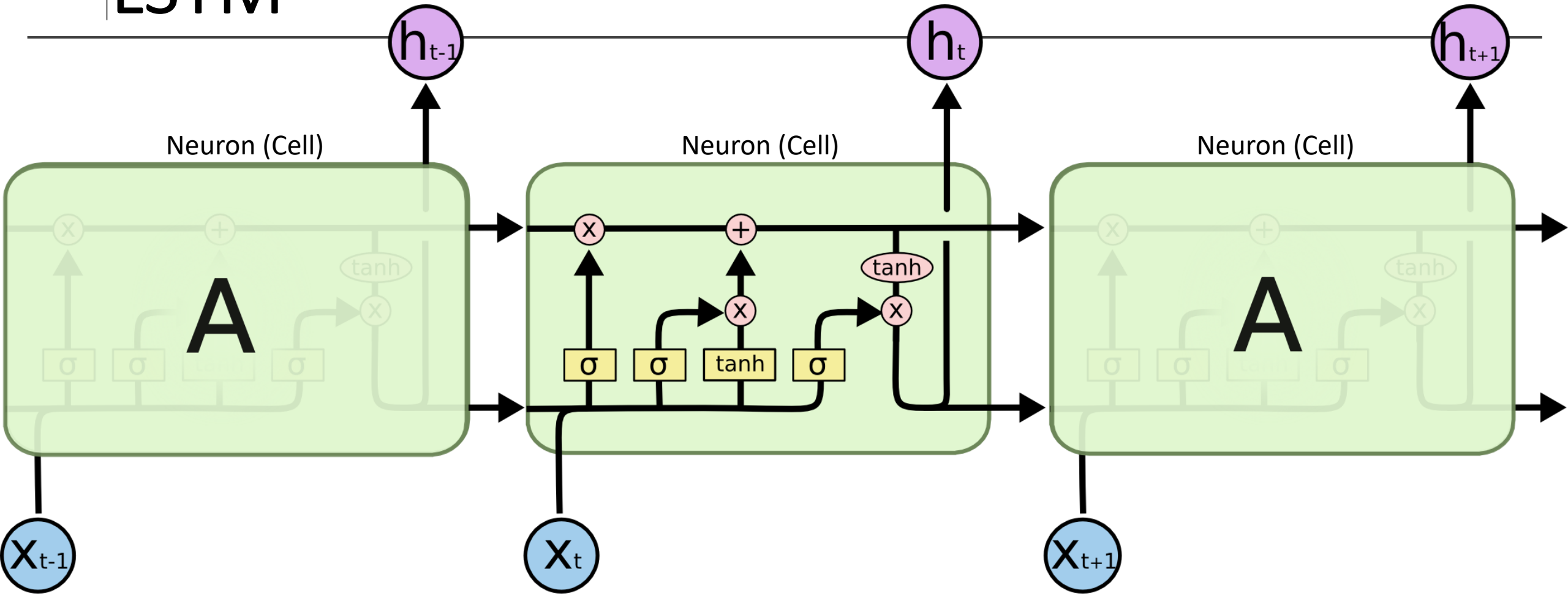


Solution: Long Short Term Memory

- Long Short Term Memory networks (LSTMs) are a special kind of RNN, capable of effectively learning long-term dependencies

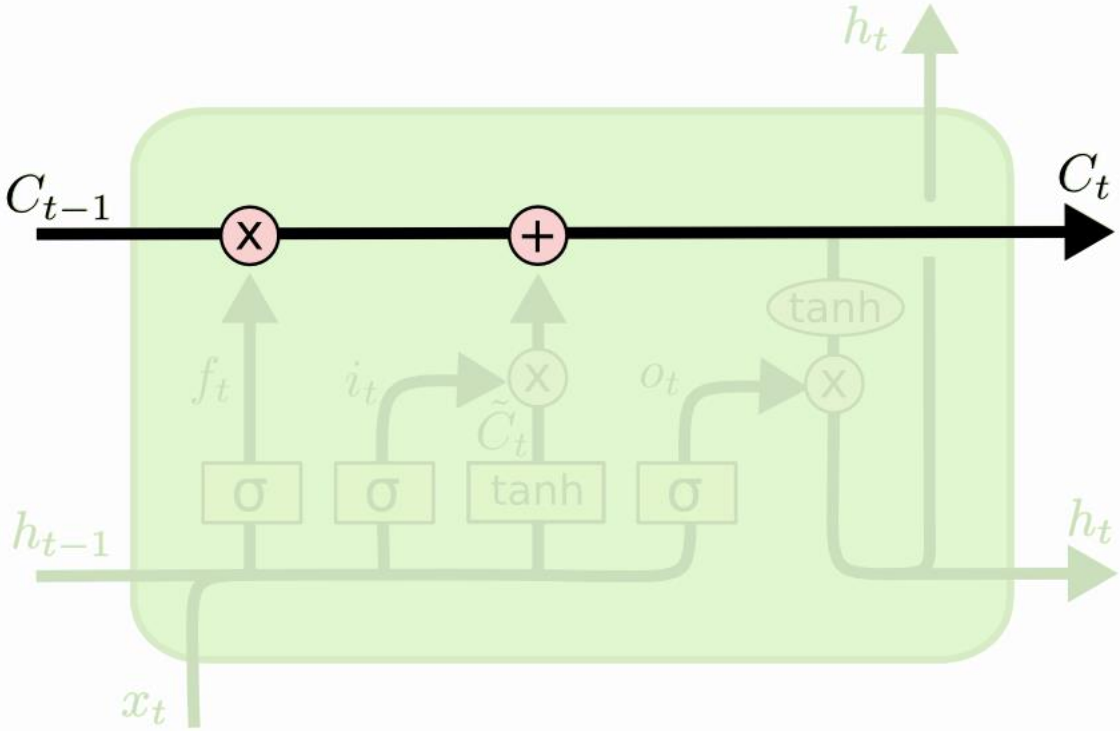
- LSTMs are explicitly designed to:
  - Control the vanishing gradient
  - Avoid the long-term dependency problem
    - Remembering information for long periods of time is practically their default behavior, not something they struggle to learn

LSTM


h_{t-1}
h_t
h_{t+1}
Neuron (Cell)
Neuron (Cell)
Neuron (Cell)
A
A
σ
σ
tanh
σ
tanh
X_{t-1}
X_t
X_{t+1}
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

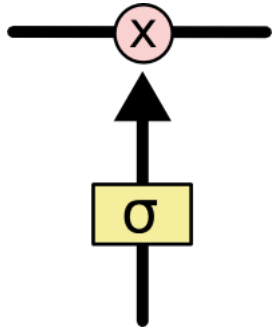
LSTM

- The cell state stores an inner information of the chain (inner history)
  - It runs straight down the entire chain, with only minor linear interactions
  - It's very easy for information to just flow along it unchanged



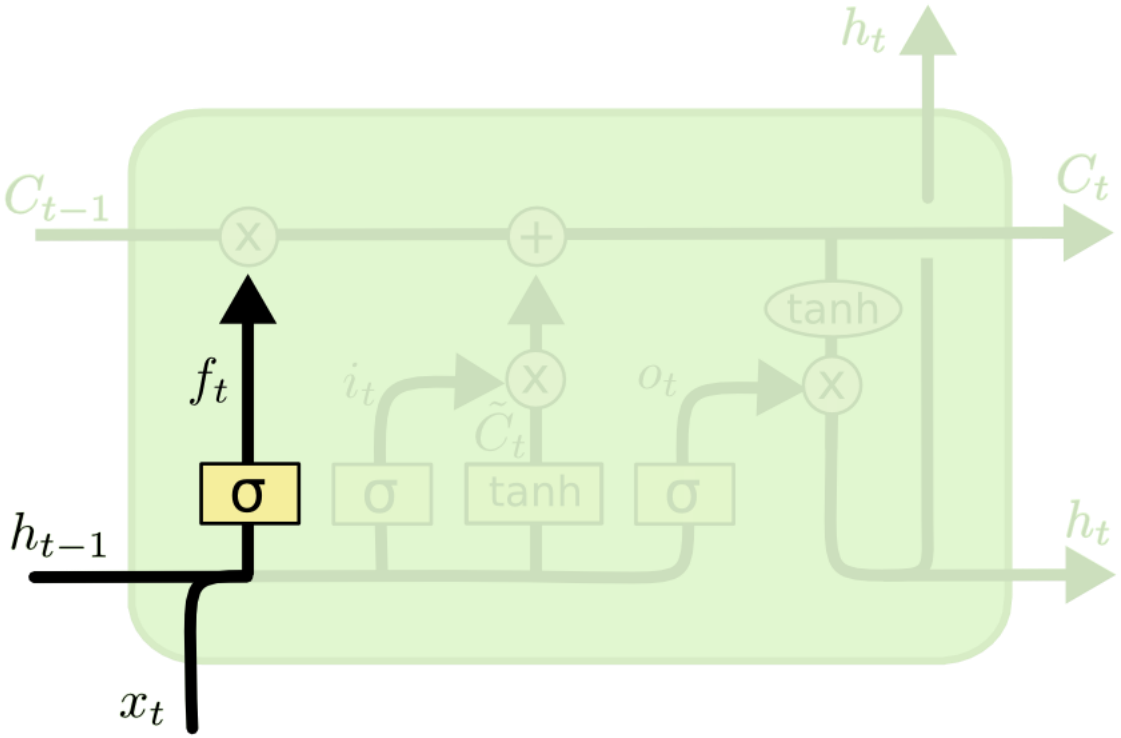
LSTM

- The LSTM does have the ability to remove or add information to the state
  - The information control system is the gate
- Gates are a way to optionally let information through.
  - They are composed out of a sigmoid neural net layer and a pointwise multiplication



LSTM

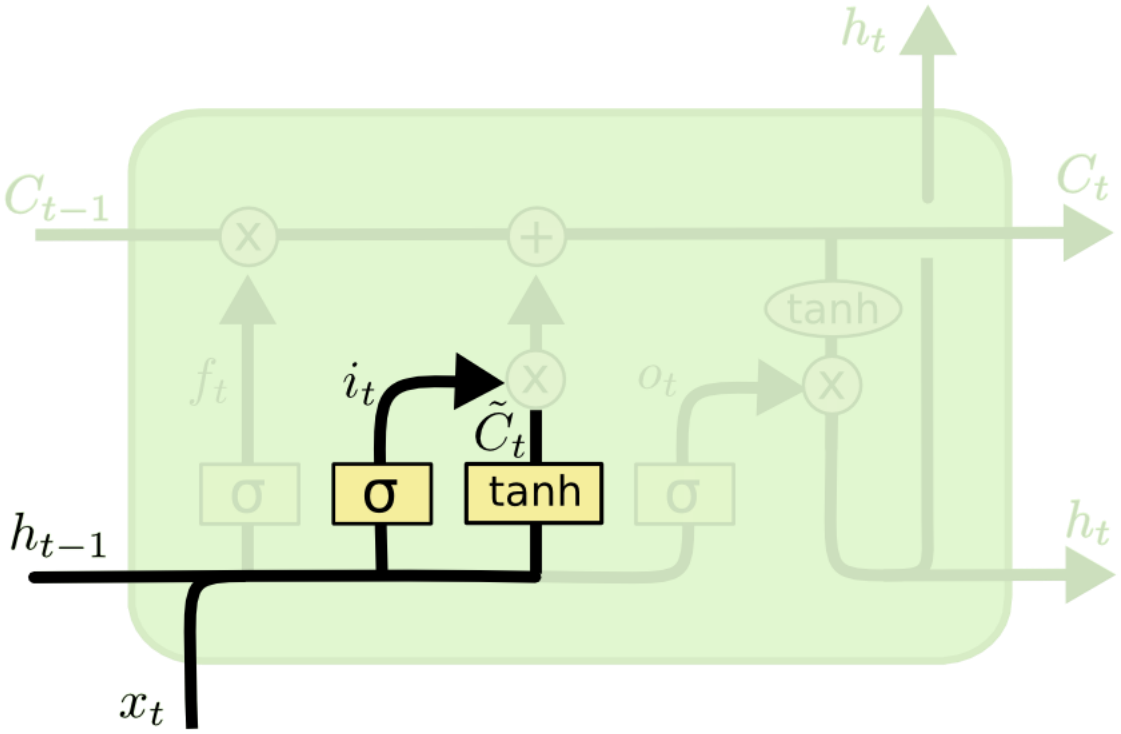
- The forget gate
  - It determines how much previous inner history to consider from now on



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

LSTM

- Input gate and contribution to the current inner history
  - How much the new input can influence the inner history

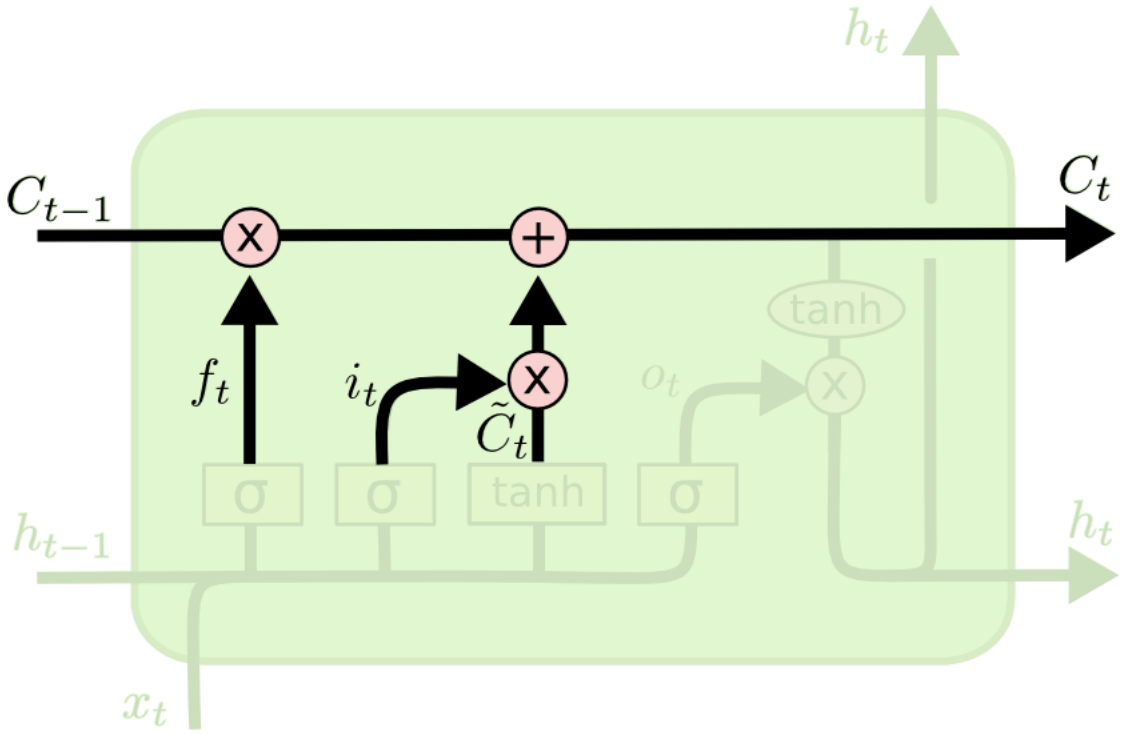


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

LSTM

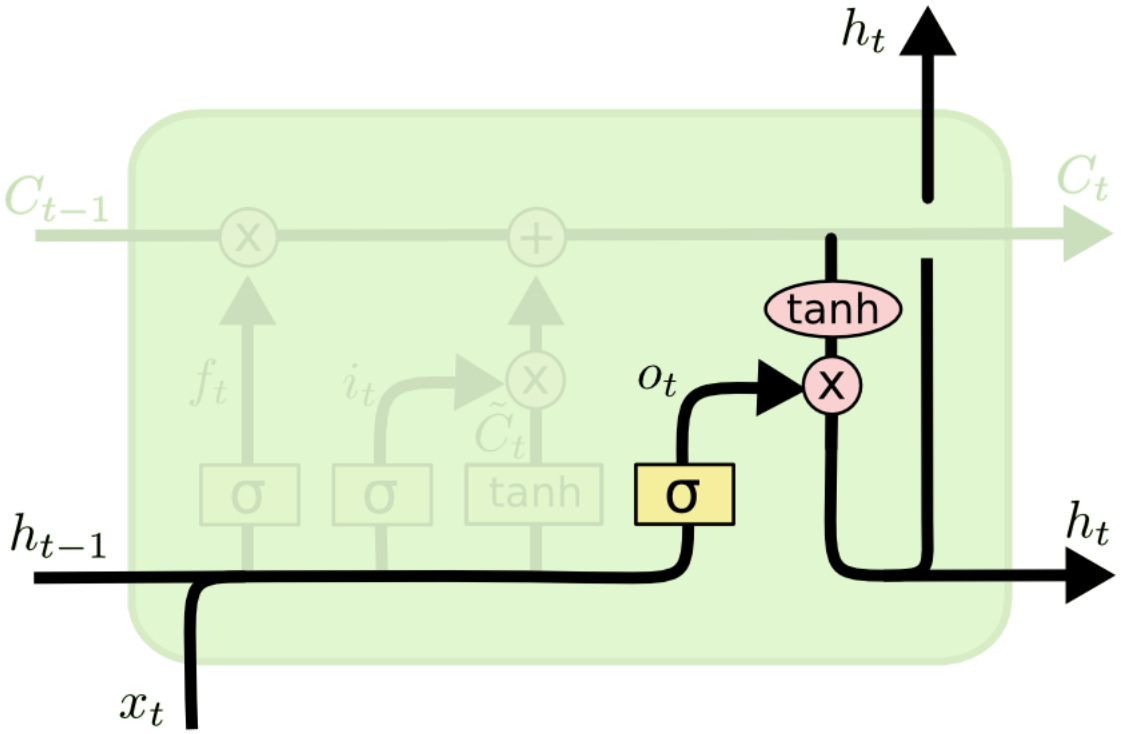
- Update the inner state
  - Here LSTMs are tackling the vanishing gradient problem



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

LSTM

- Generating the output and output gate
  - The output is dependent on the cell inner status
  - LSTMs control how much history to use for generating the output

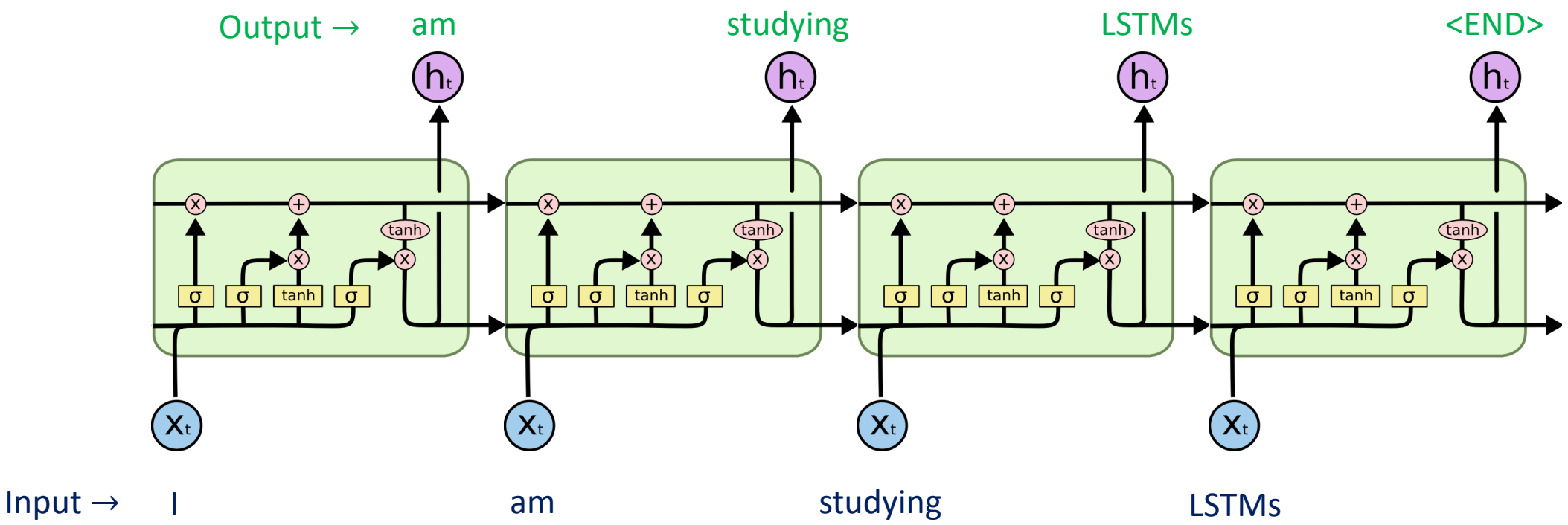


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

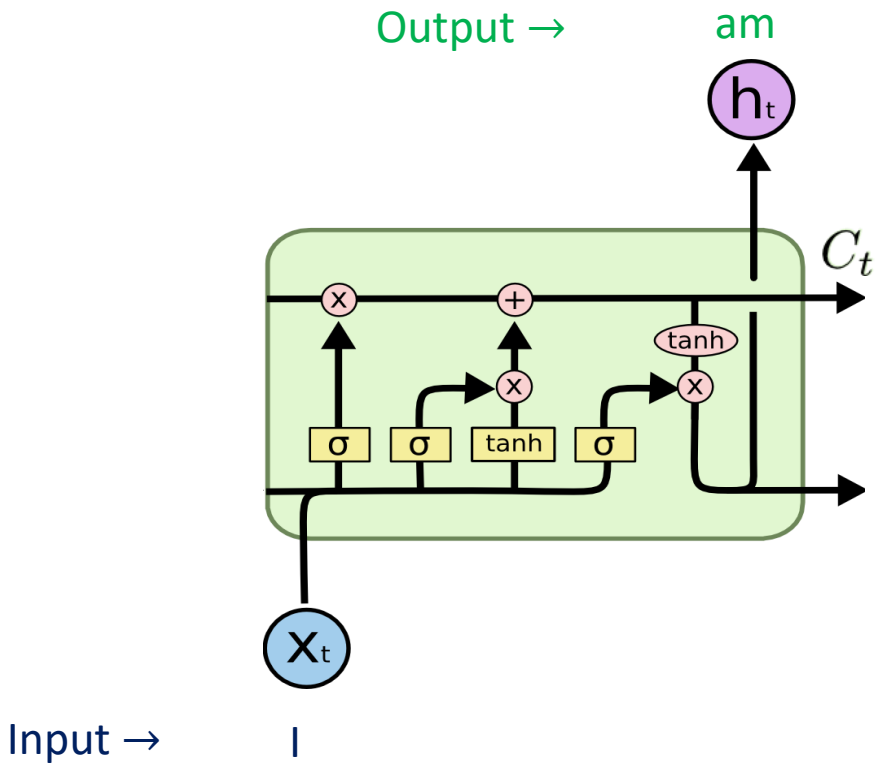
$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM network: how does it work?

- Consider the sentence: “I am studying LSTMs”



LSTM network: how does it work?



Vector representation of the LSTM knowledge (inner history)

Vector representation of the word “am”

Example

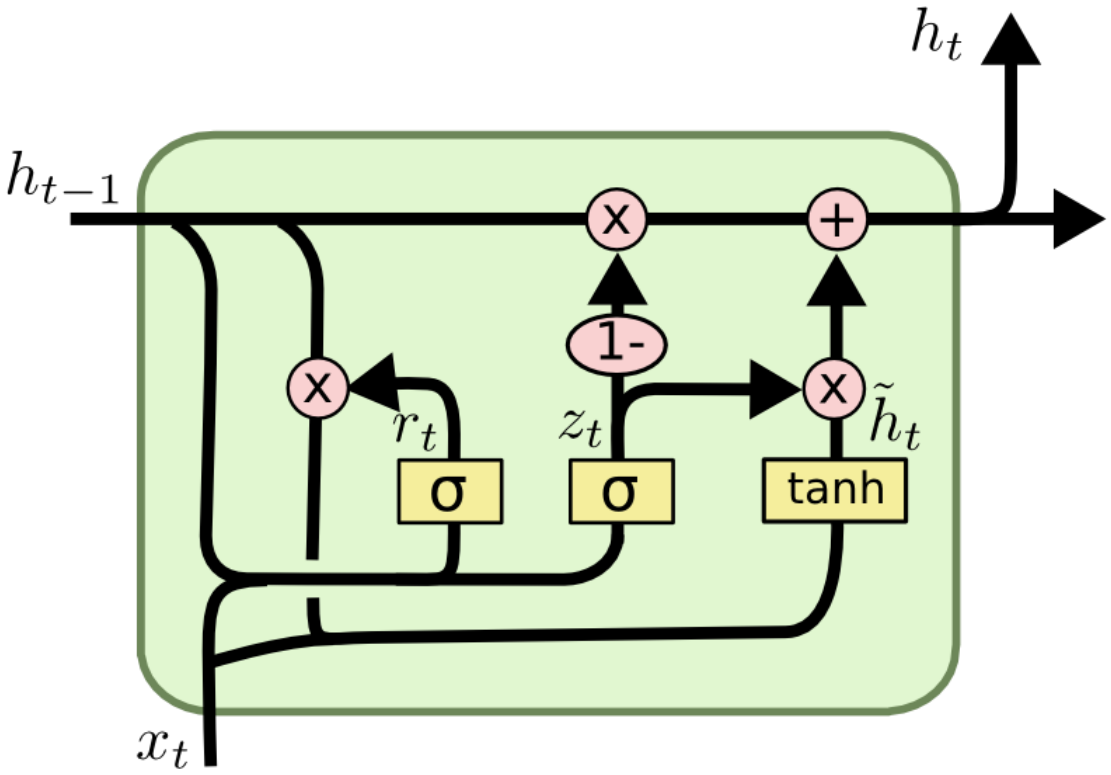
```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Gated Recurrent Unit (GRU)

- Cho, et al. (2014)
- Combine the forget and input gates into a single “update gate”



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

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

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

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

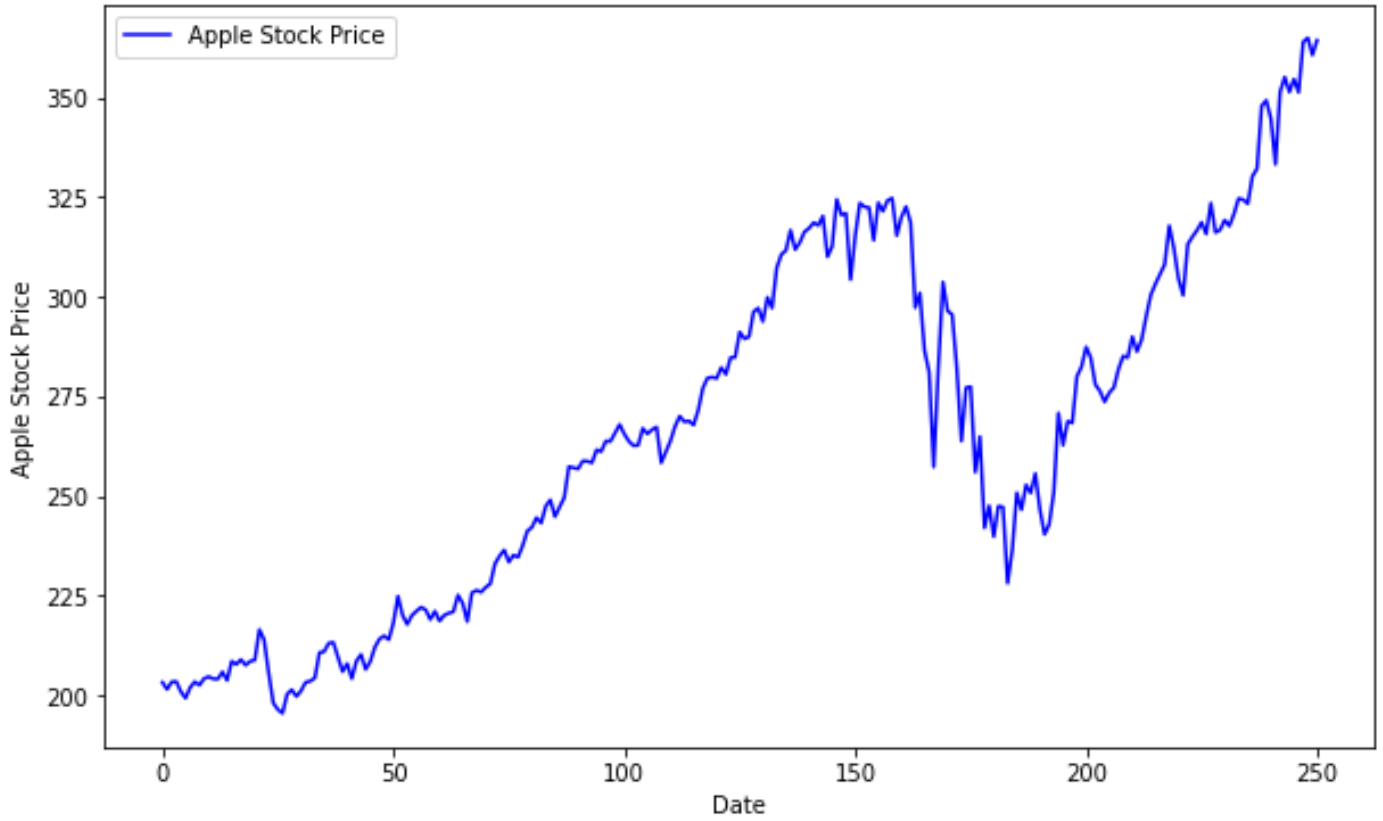
Setup - Yahoo Finance

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from google.colab import files
files.upload()
apple_training_complete = pd.read_csv(r'AAPL.csv')
apple_training_processed = apple_training_complete.iloc[:, 1:1].values
```

Plot the Dataset

```
plt.figure(figsize=(10,6))
plt.plot(apple_training_processed, color='blue', label='Apple Stock Price')
plt.xlabel('Date')
plt.ylabel('Apple Stock Price')
plt.legend()
plt.show()
```



Preprocessing

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range = (0, 1))
apple_training_scaled = scaler.fit_transform(apple_training_processed)
features_set = []
labels = []
for i in range(60, 1259):
    features_set.append(apple_training_scaled[i-60:i, 0])
    labels.append(apple_training_scaled[i, 0])
features_set, labels = np.array(features_set), np.array(labels)
features_set = np.reshape(features_set,
                          (features_set.shape[0], features_set.shape[1], 1))
```

Building the Network

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(features_set.shape[1], 1)))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(units = 1))
model.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

Fit

```
model.fit(features_set, labels, epochs = 100, batch_size = 32)
```

Prediction

```
from google.colab import files
files.upload()

apple_testing_complete = pd.read_csv(r'AAPL-Test.csv')
apple_testing_processed = apple_testing_complete.iloc[:, 1:1].values
apple_total = pd.concat((apple_training_complete['Open'], apple_testing_complete['Open']), axis=0)
test_inputs = apple_total[len(apple_total) - len(apple_testing_complete) - 60:].values
test_inputs = test_inputs.reshape(-1,1)
test_inputs = scaler.transform(test_inputs)
test_features = []

for i in range(60, 80):
    test_features.append(test_inputs[i-60:i, 0])

test_features = np.array(test_features)
test_features = np.reshape(test_features, (test_features.shape[0], test_features.shape[1], 1))
```

Architectures