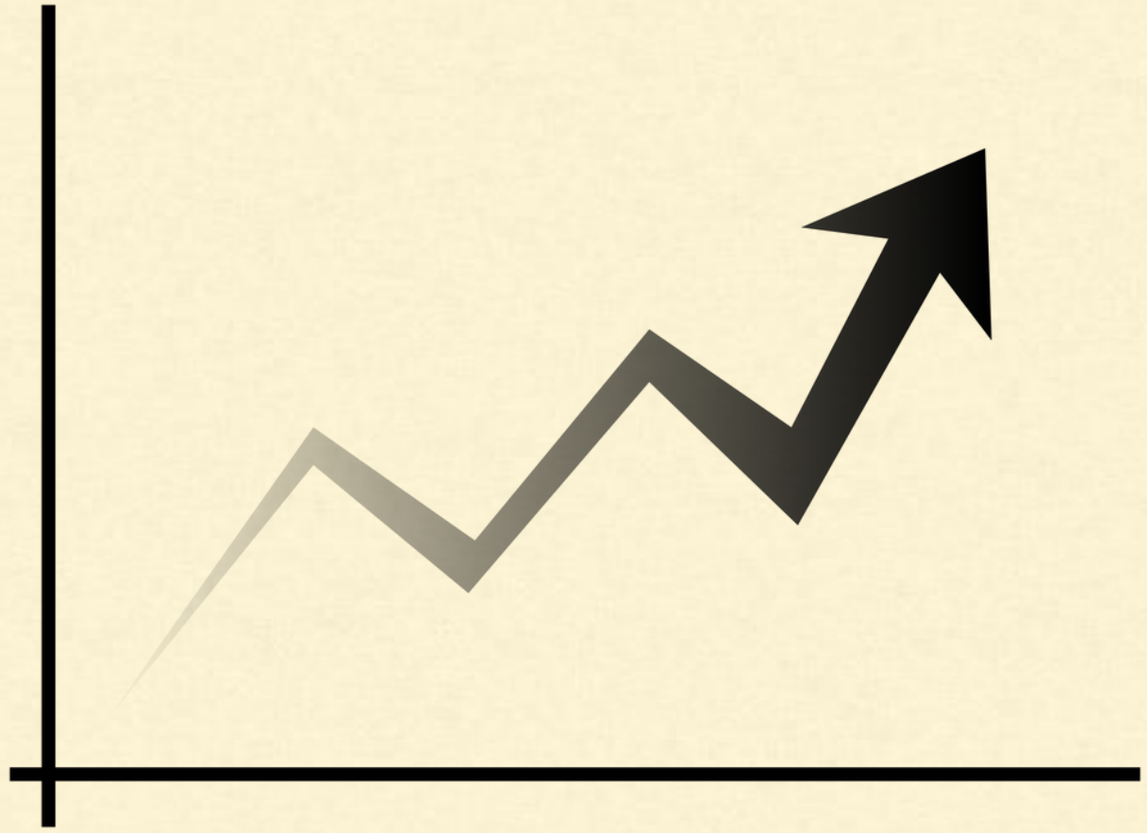
Recurrent Neural Network

Recurrent Neural Network

- Predicting the future is what we do all the time
  - Finishing a friend's sentence
  - Anticipating the smell of coffee at the breakfast or
  - Catching the ball in the field
- In this chapter, we will cover RNN
  - Networks which can predict future
- Unlike all the nets we have discussed so far
  - RNN can work on sequences of arbitrary lengths
  - Rather than on fixed-sized inputs

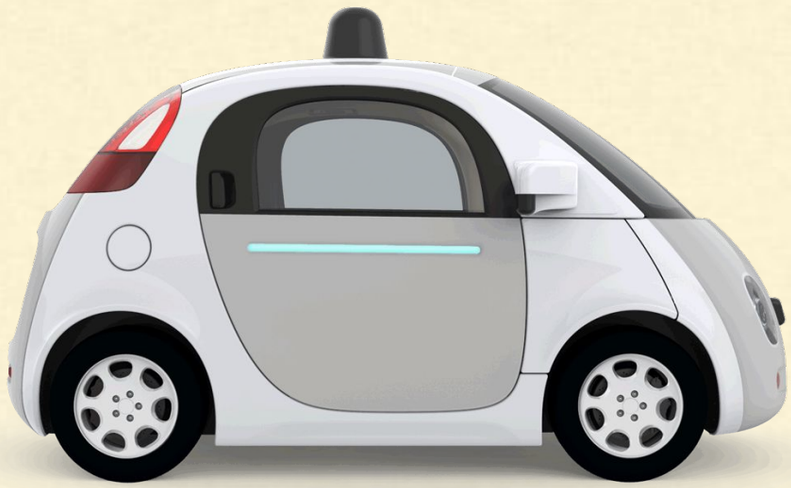
Recurrent Neural Network - Applications

- RNN can analyze time series data
  - Such as stock prices, and
  - Tell you when to buy or sell



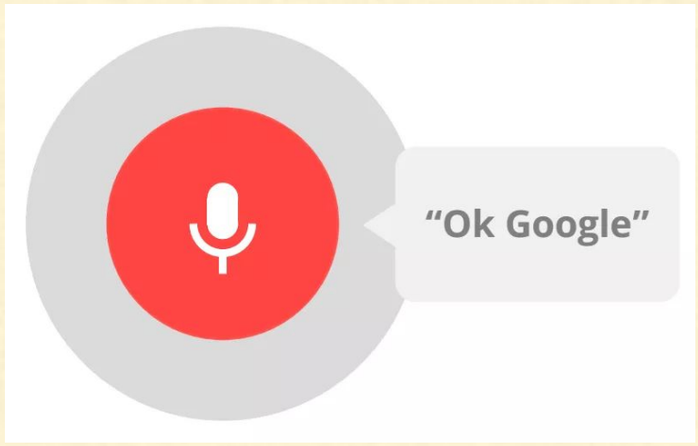
Recurrent Neural Network - Applications

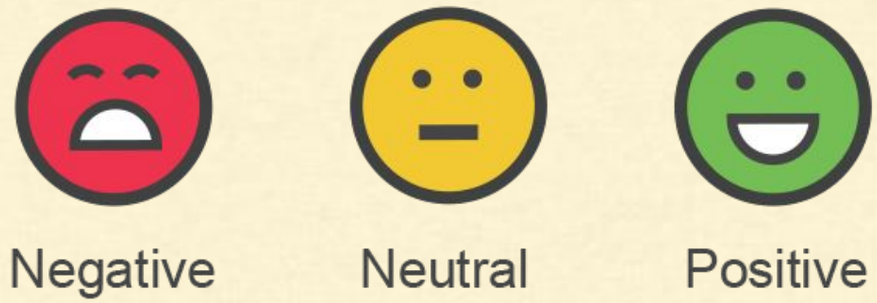
- In autonomous driving systems, RNN can
  - Anticipate car trajectories and
  - Help avoid accidents



Recurrent Neural Network - Applications

- RNN can take sentences, documents, or audio samples as input and
  - Make them extremely useful
  - For natural language processing (NLP) systems such as
    - Automatic translation
    - Speech-to-text or
    - Sentiment analysis

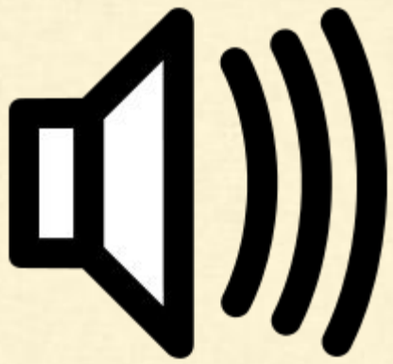




Recurrent Neural Network - Applications

- RNNs' ability to anticipate also makes them capable of surprising creativity.
  - You can ask them to predict which are the most likely next notes in a melody
  - Then randomly pick one of these notes and play it.
  - Then ask the net for the next most likely notes, play it, and repeat the process again and again.

Here is an example melody produced by Google's Magenta project



Recurrent Neural Network

- In this chapter we will learn about
  - Fundamental concepts in RNNs
  - The main problem RNNs face
  - And the solution to the problems
  - How to implement RNNs
- Finally, we will take a look at the
  - Architecture of a machine translation system

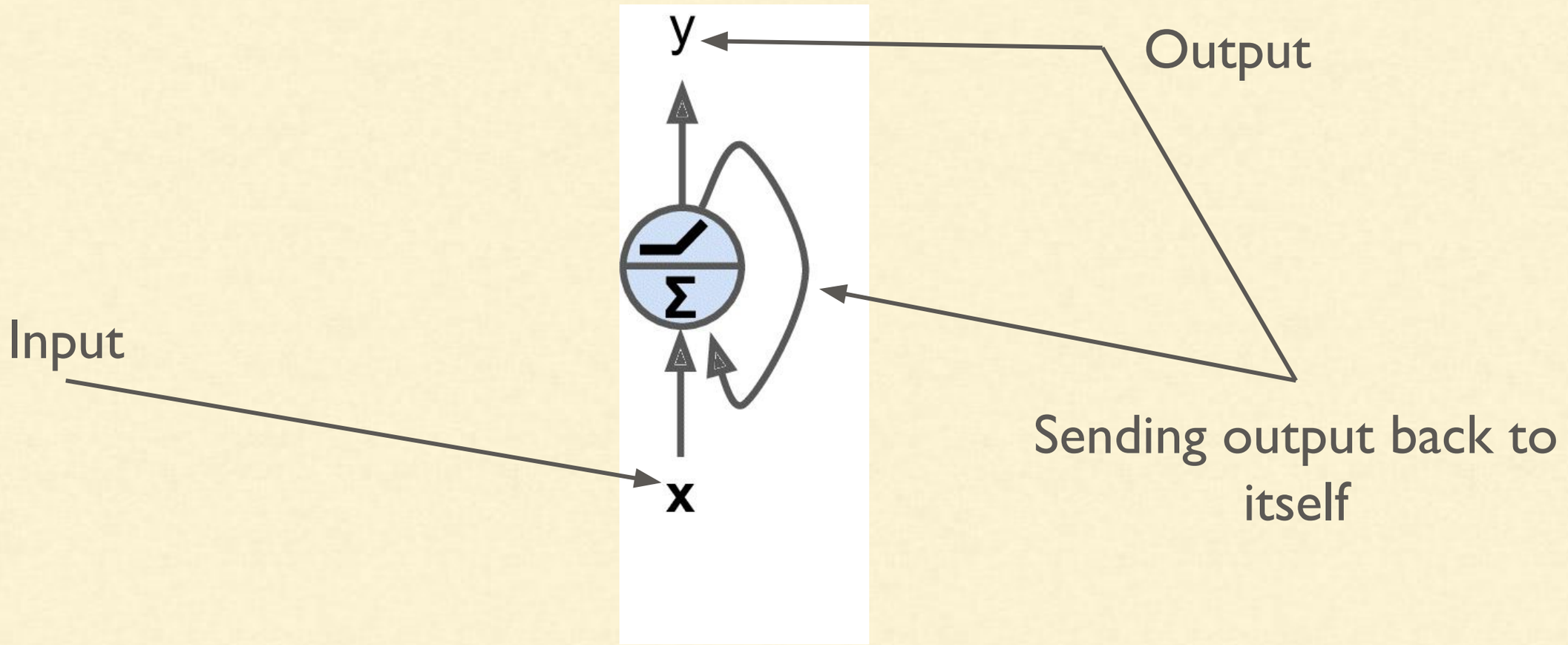
Recurrent Neurons

Recurrent Neurons

- Up to now we have mostly looked at feedforward neural networks
  - Where the activations flow only in one direction
  - From the input layer to the output layer
- RNN looks much like a feedforward neural network
  - Except it also has connections pointing backward

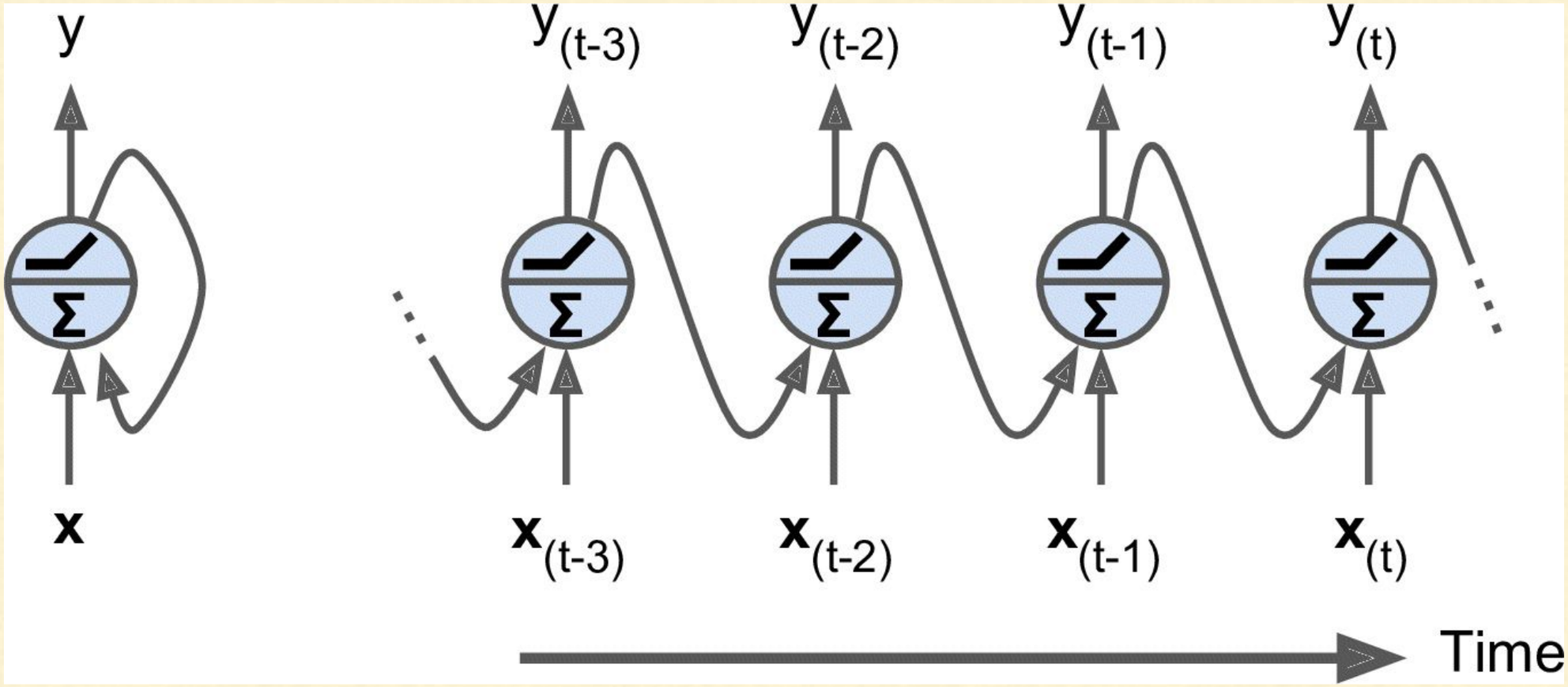
Recurrent Neurons

- Let's look at the simplest possible RNN
  - Composed of just one neuron receiving inputs
  - Producing an output, and
  - Sending that output back to itself



Recurrent Neurons

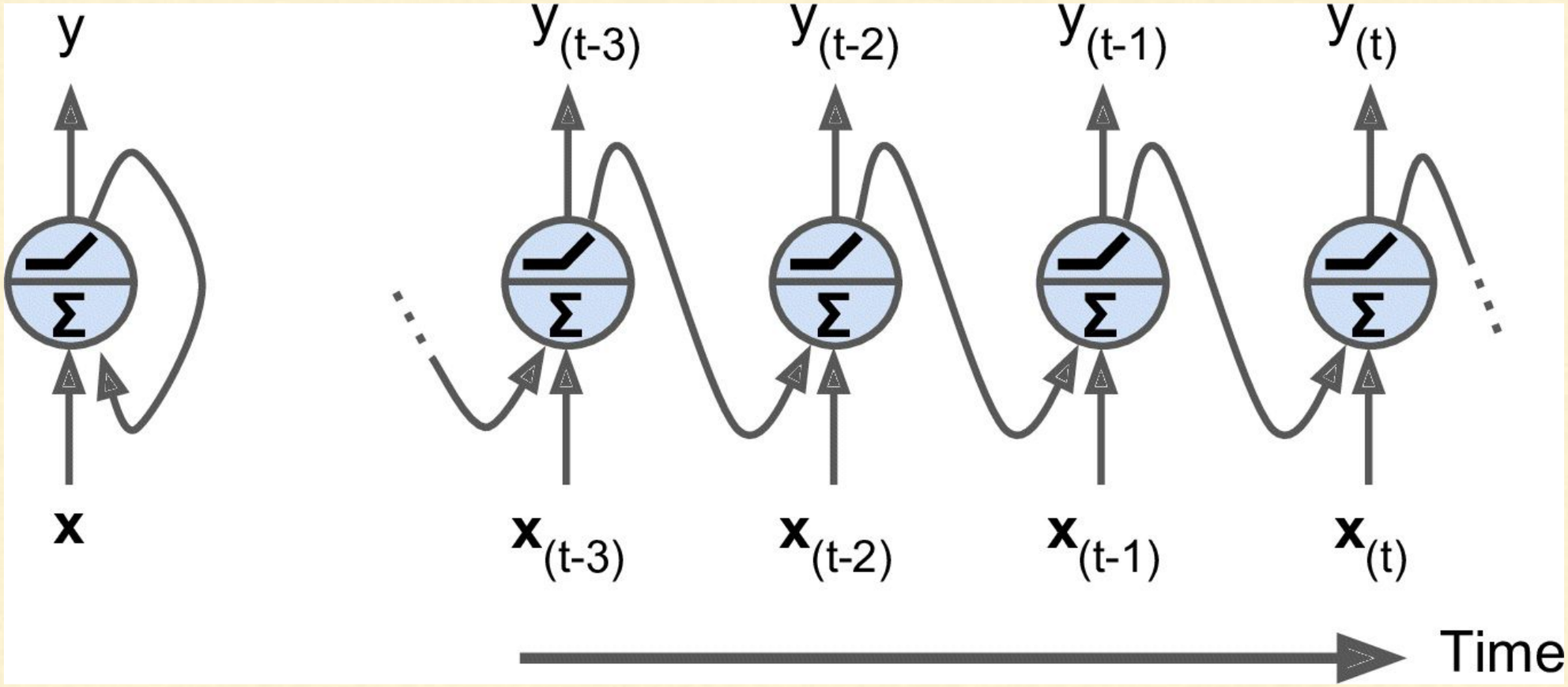
- At each time step **t** (also called a frame)
  - This recurrent neuron receives the inputs $\mathbf{x}_{(t)}$
  - As well as its own output from the previous time step $\mathbf{y}_{(t-1)}$



A recurrent neuron (left), unrolled through time (right)

Recurrent Neurons

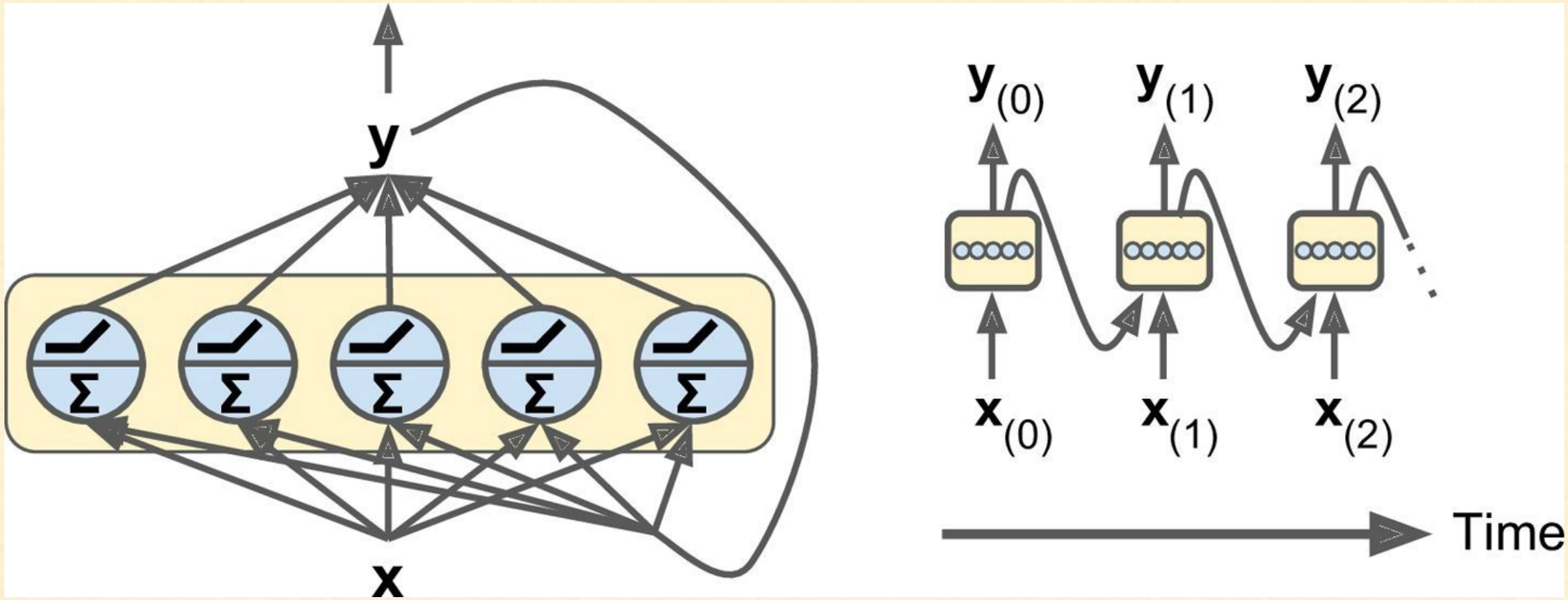
- We can represent this tiny network against the time axis (See below figure)
- This is called ***unrolling the network through time***



A recurrent neuron (left), unrolled through time (right)

Recurrent Neurons

- We can easily create a layer of recurrent neurons
- At each time step t, every neuron receives both the
  - Input vector $x_{(t)}$ and
  - Output vector from the previous time step $y_{(t-1)}$



A layer of recurrent neurons (left), unrolled through time(right)

Recurrent Neurons

- Each recurrent neuron has two sets of weights
  - One for the inputs $x_{(t)}$ and the
  - Other for the outputs of the previous time step, $y_{(t-1)}$
- Let's call these weight vectors w_x and w_y
- Below equation represents the output of a single recurrent neuron

Output of a single recurrent neuron for a single instance

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}_{(t)}^{T} \cdot \mathbf{w}_x + \mathbf{y}_{(t-1)}^{T} \cdot \mathbf{w}_y + b\right)$$

ϕ() is the activation function like ReLU

bias

Recurrent Neurons

- We can compute a whole layer's output
  - In one shot for a whole mini-batch
  - Using a vectorized form of the previous equation

Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b})$$
$$= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \cdot \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b})$$

$$= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \cdot \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

- $\mathbf{Y}_{(t)}$ is an m **x** $n_{neurons}$ matrix containing the
  - Layer's outputs at time step t for each instance in the minibatch
  - m is the number of instances in the mini-batch
  - $n_{neurons}$ is the number of neurons

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b})$$

$$= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \cdot \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

- $\mathbf{X}_{(t)}$ is an $m \times n_{inputs}$ matrix containing the inputs for all instances
  - n_{inputs} is the number of input features

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b})$$

$$= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \cdot \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

- $\mathbf{W}_x$ is an $n_{inputs} \times n_{neurons}$ matrix containing the connection weights for the inputs of the current time step
- $\mathbf{W}_y$ is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights for the outputs of the previous time step

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b})$$

$$= \phi([\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}] \cdot \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated into a single weight matrix $\mathbf{W}$ of shape $(n_{inputs} + n_{neurons}) \times n_{neurons}$
- $\mathbf{b}$ is a vector of size $n_{neurons}$ containing each neuron's bias term

Memory Cells

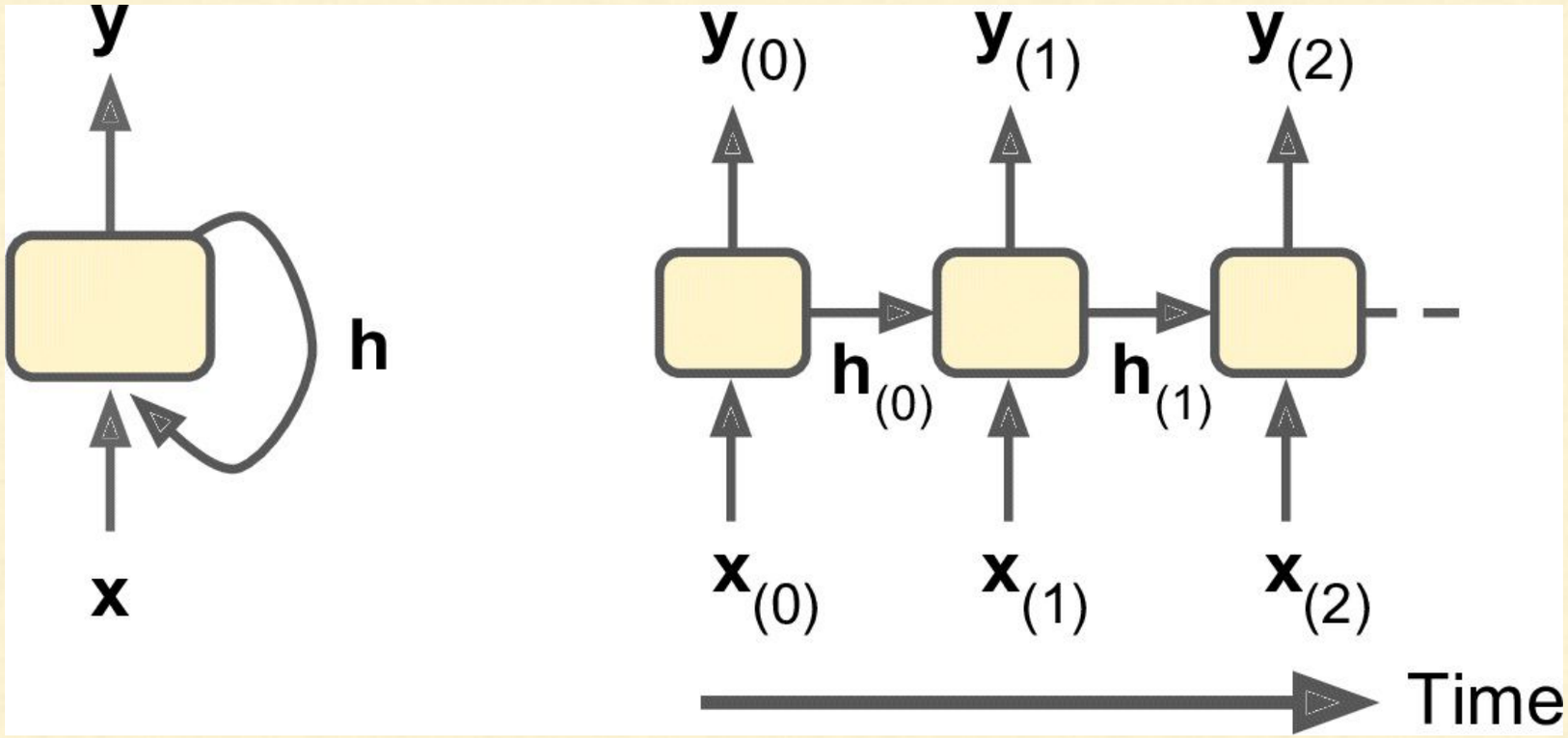
- Since the output of a recurrent neuron at time step t is a
  - Function of all the inputs from previous time steps
  - We can say that it has a form of ***memory***
- A part of a neural network that
  - Preserves some state across time steps is called a **memory cell**

Memory Cells

- In general a cell's state at time step t, denoted $h_{(t)}$ is a
  - Function of some inputs at that time step and
  - Its state at the previous time step $h_{(t)} = f(h_{(t-1)}, x_{(t)})$
- Its output at time step t, denoted $y_{(t)}$ is also a
  - Function of the previous state and the current inputs

Memory Cells

- In the case of basics cells we have discussed so far
  - The output is simply equal to the state
  - But in more complex cells this is not always the case

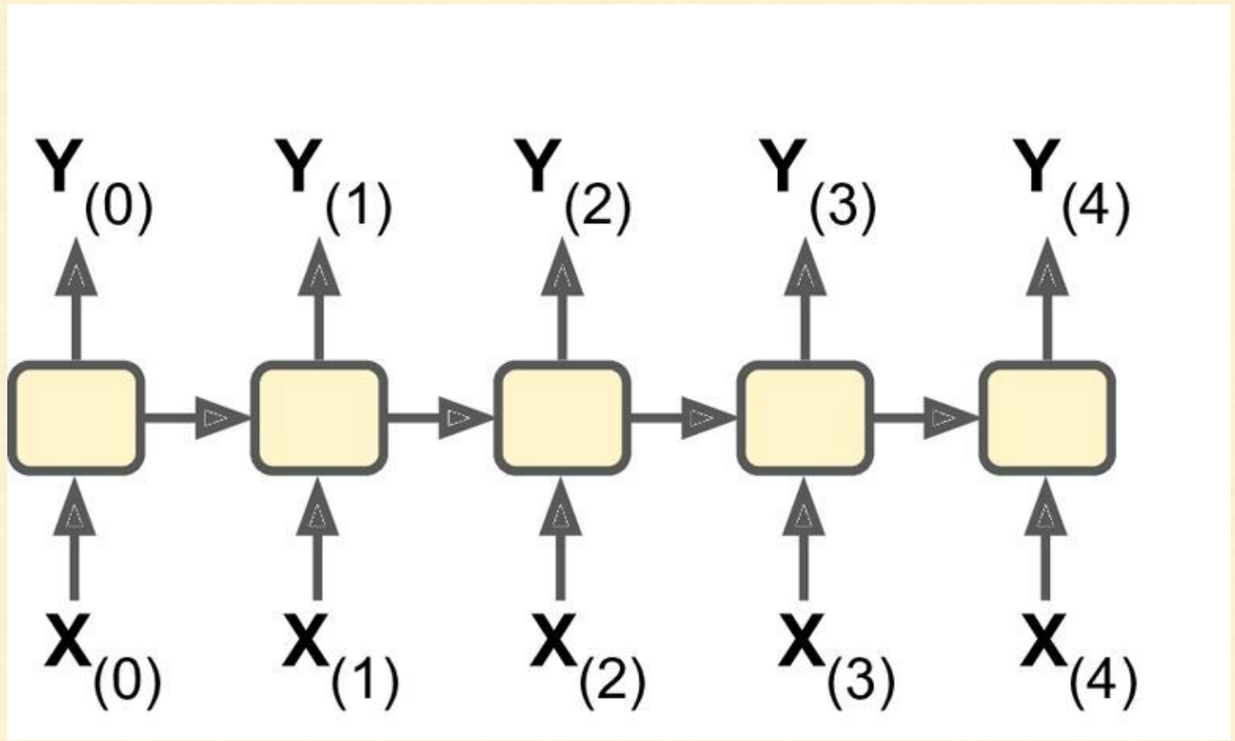


A cell's hidden state and its output may be different

Input and Output Sequences

Sequence-to-sequence Network

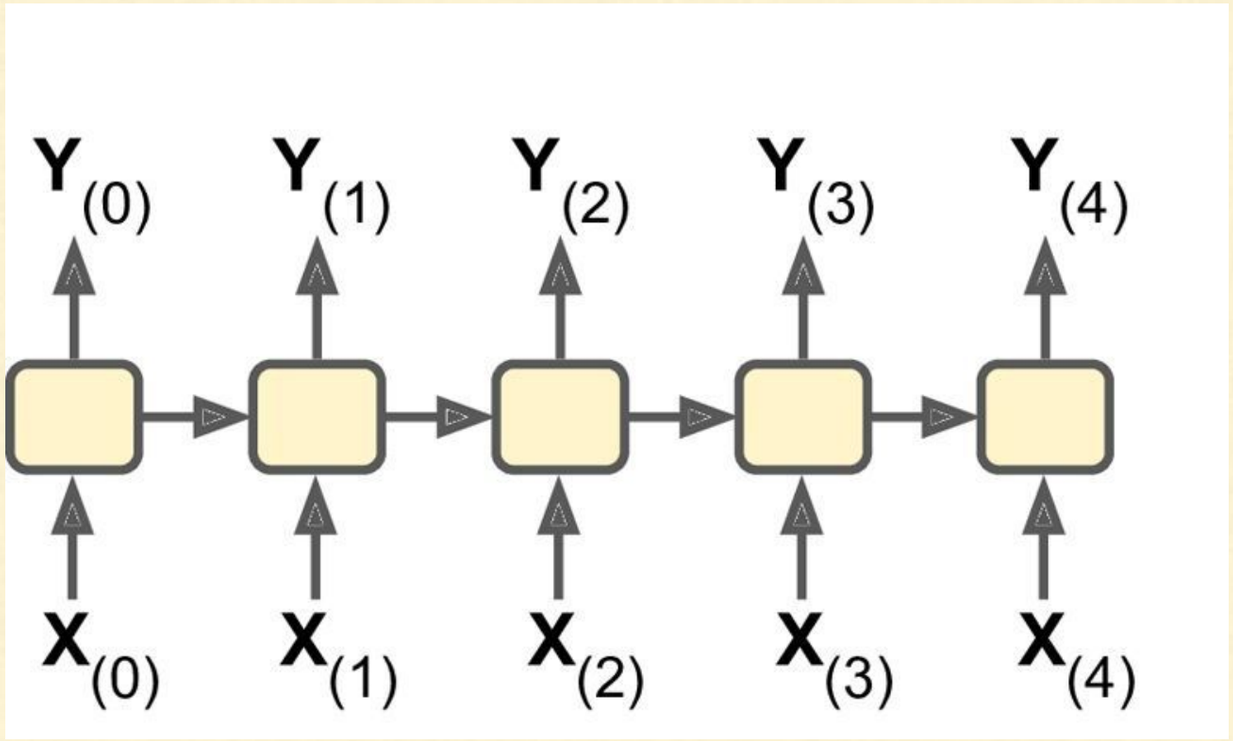
- An RNN can simultaneously take a
  - Sequence of inputs and
  - Produce a sequence of outputs



Input and Output Sequences

Sequence-to-sequence Network

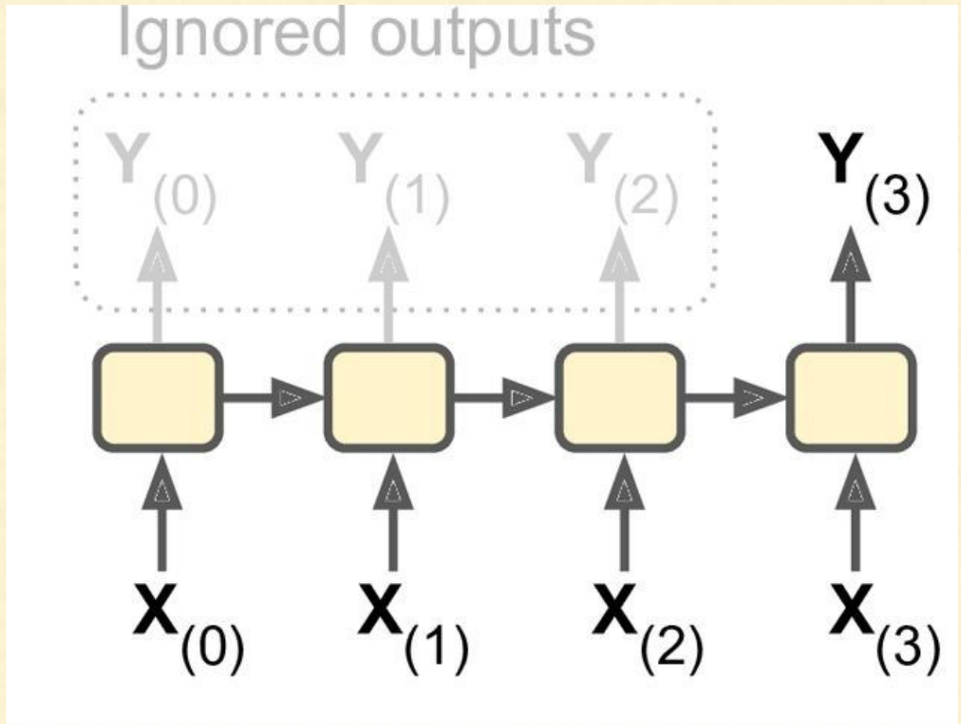
- This type of network is useful for predicting time series
  - Such as stock prices
- We feed it the prices over the last N days and
  - It must output the prices shifted by one day into the future
  - i.e., from N – 1 days ago to tomorrow



Input and Output Sequences

Sequence-to-vector Network

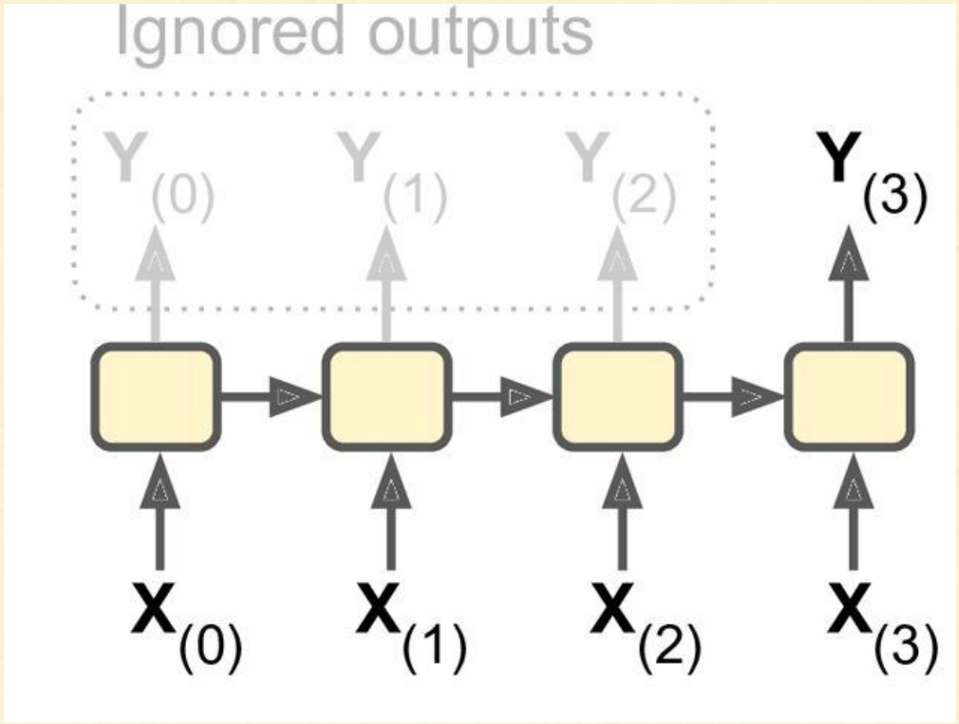
- Alternatively we could feed the network a sequence of inputs and
  - Ignore all outputs except for the last one



Input and Output Sequences

Sequence-to-vector Network

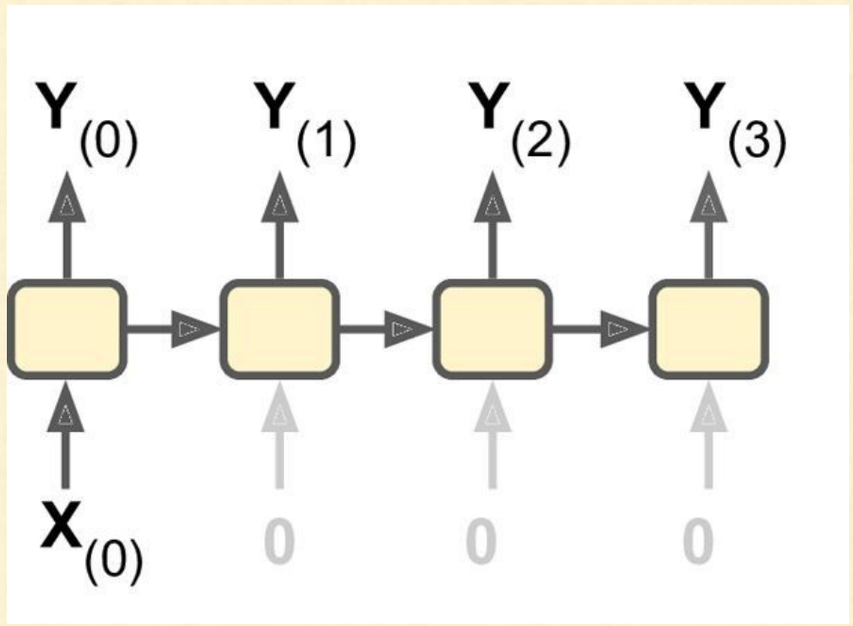
- We can feed this network a sequence of words
  - Corresponding to a movie review and
  - The network would output a sentiment score
  - e.g., from –1 [hate] to +1 [love]



Input and Output Sequences

Vector-to-sequence Network

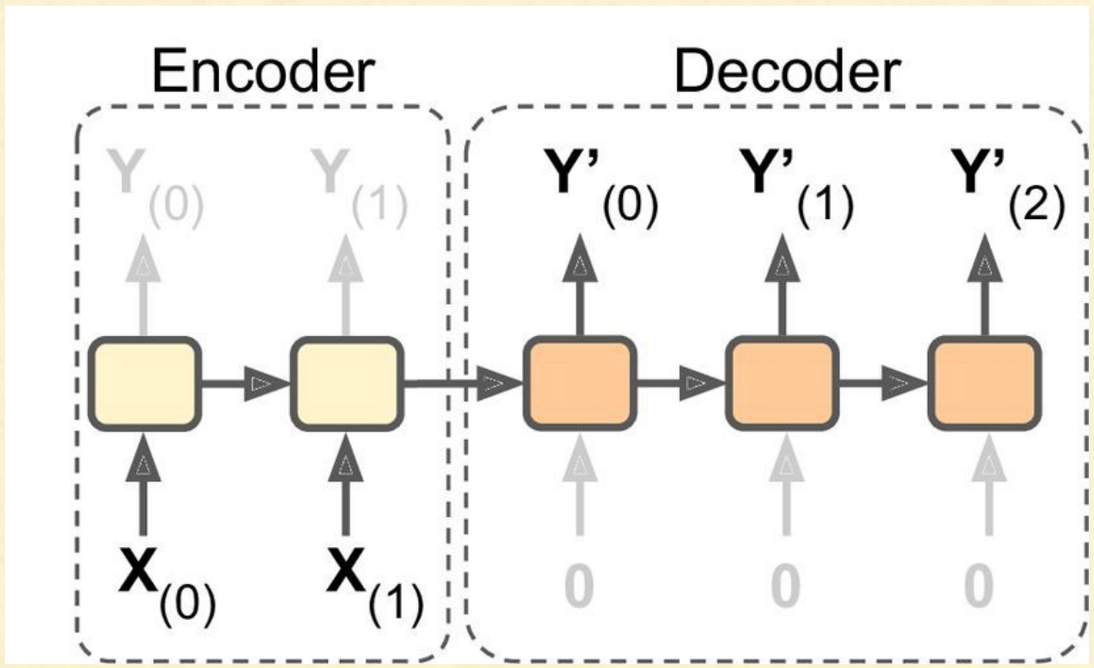
- We could feed the network a single input at the first time step and
  - Zeros for all other time steps and
  - Let is output a sequence
- For example, the input could be an image and the
  - Output could be a caption for the image



Input and Output Sequences

Encoder-Decoder

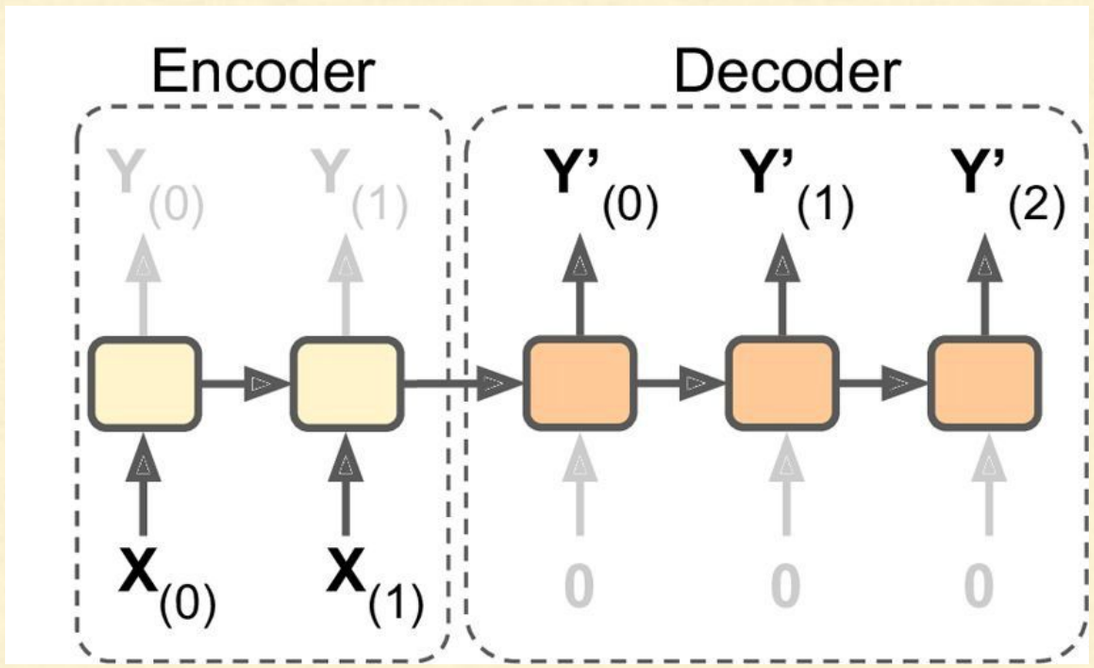
- In this network, we have
  - sequence-to-vector network, called **an encoder** followed by
  - vector-to-sequence network, called **a decoder**



Input and Output Sequences

Encoder-Decoder

- This can be used for translating a sentence
  - From one language to another
- We feed the network sentence in one language
  - The encoder converts this sentence into single vector representation
  - Then the decoder decodes this vector into a sentence in another language



Input and Output Sequences

Encoder-Decoder

- This two step model works much better than
  - Trying to translate on the fly with a
  - Single sequence-to-sequence RNN
- Since the last words of a sentence can affect the
  - First words of the translation
  - So we need to wait until we know the whole sentence

Basic RNNs in TensorFlow

Basic RNNs in TensorFlow

- Let's implement a very simple RNN model
  - Without using any of the TensorFlow's RNN operations
  - To better understand what goes on under the hood
- Let's create an RNN composed of a layer of five recurrent neurons
  - Using the tanh activation function and
  - Assume that the RNN runs over only two time steps and
  - Taking input vectors of size 3 at each time step

Basic RNNs in TensorFlow

```
n_inputs = 3
n_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

init = tf.global_variables_initializer()
```

- This network looks like a two-layer feedforward neural network with two differences
  - The same weights and bias terms are shared by both layers and
  - We feed inputs at each layer, and we get outputs from each layer

Basic RNNs in TensorFlow

```
import numpy as np

# Mini-batch:         instance 0,instance 1,instance 2,instance 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

- To run the model, we need to feed it the inputs at both time steps
- Mini-batch contains four instances
  - Each with an input sequence composed of exactly two inputs

Basic RNNs in TensorFlow

```
import numpy as np

# Mini-batch:         instance 0,instance 1,instance 2,instance 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

- At the end, Y0_val and Y1_val contain the outputs of the network
  - At both time steps for all neurons and
  - All instances in the mini-batch

Checkout the complete code under **“Manual RNN”** section in notebook

Static Unrolling Through Time

- Let's look at how to create the same model
  - Using TensorFlow's RNN operations
- The `static_rnn()` function creates
  - An unrolled RNN network by chaining cells
- The below code creates the exact same model as the previous one

```
>>> X0 = tf.placeholder(tf.float32, [None, n_inputs])
>>> X1 = tf.placeholder(tf.float32, [None, n_inputs])
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, [X0, X1], dtype=tf.float32
    )
>>> Y0, Y1 = output_seqs
```

Static Unrolling Through Time

```
>>> X0 = tf.placeholder(tf.float32, [None, n_inputs])
>>> X1 = tf.placeholder(tf.float32, [None, n_inputs])
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, [X0, X1], dtype=tf.float32
    )
>>> Y0, Y1 = output_seqs
```

- First we create the input placeholders

Static Unrolling Through Time

```
>>> X0 = tf.placeholder(tf.float32, [None, n_inputs])
>>> X1 = tf.placeholder(tf.float32, [None, n_inputs])
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, [X0, X1], dtype=tf.float32
    )
>>> Y0, Y1 = output_seqs
```

- Then we create a `BasicRNNCell`
  - It is like a factory that creates
  - Copies of the cell to build the unrolled RNN
    - One for each time step

Static Unrolling Through Time

```
>>> X0 = tf.placeholder(tf.float32, [None, n_inputs])
>>> X1 = tf.placeholder(tf.float32, [None, n_inputs])
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, [X0, X1], dtype=tf.float32
    )
>>> Y0, Y1 = output_seqs
```

- Then we call `static_rnn()`, giving it the cell factory and the input tensors
- And telling it the data type of the inputs
  - This is used to create the initial state matrix
  - Which by default is full of zeros

Static Unrolling Through Time

```
>>> X0 = tf.placeholder(tf.float32, [None, n_inputs])
>>> X1 = tf.placeholder(tf.float32, [None, n_inputs])
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, [X0, X1], dtype=tf.float32
    )
>>> Y0, Y1 = output_seqs
```

- The `static_rnn()` function returns two objects
- The first is a Python list containing the output tensors for each time step
- The second is a tensor containing the final states of the network
- When we use basic cells
  - Then the final state is equal to the last output

Static Unrolling Through Time

Checkout the complete code under **“Using static_rnn()”** section in notebook

Static Unrolling Through Time

- In the previous example, if there were 50 time steps then
  - It would not be convenient to define
  - 50 place holders and 50 output tensors
- Moreover, at execution time we would have to feed
  - Each of the 50 placeholders and manipulate the 50 outputs
- Let's do it in a better way

Static Unrolling Through Time

```
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, X_seqs, dtype=tf.float32
    )
>>> outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

- The above code takes a single input placeholder of
  - shape [None, n_steps, n_inputs]
  - Where the first dimension is the mini-batch size

Static Unrolling Through Time

```
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, X_seqs, dtype=tf.float32
    )
>>> outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

- Then it extracts the list of input sequences for each time step
- X_seqs is a Python list of n_steps tensors of shape [None, n_inputs]
  - Where first dimension is the minibatch size

Static Unrolling Through Time

```
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, X_seqs, dtype=tf.float32
    )
>>> outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

- To do this, we first swap the first two dimensions
  - Using the `transpose()` function so that the
  - Time steps are now the first dimension

Static Unrolling Through Time

```
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, X_seqs, dtype=tf.float32
    )
>>> outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

- Then we extract a Python list of tensors along the first dimension
  - i.e., one tensor per time step
  - Using the unstack() function

Static Unrolling Through Time

```
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, X_seqs, dtype=tf.float32
    )
>>> outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

- The next two lines are same as before

Static Unrolling Through Time

```
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> output_seqs, states = tf.contrib.rnn.static_rnn(
        basic_cell, X_seqs, dtype=tf.float32
    )
>>> outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

- Finally, we merge all the output tensors into a single tensor
  - Using the stack() function
- And then we swap the first two dimensions to get a
  - Final outputs tensor of shape [None, n_steps, n_neurons]

Static Unrolling Through Time

- Now we can run the network by
  - Feeding it a single tensor that contains
  - All the mini-batch sequences

```
X_batch = np.array([
         # t = 0      t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 0
        [[3, 4, 5], [0, 0, 0]], # instance 1
```

```
        [[6, 7, 8], [6, 5, 4]], # instance 2
        [[9, 0, 1], [3, 2, 1]], # instance 3
    ])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
```

Static Unrolling Through Time

- And then we get a single outputs_val tensor for
  - All instances
  - All time steps, and
  - All neurons

```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]
  [-0.70553327 -0.11918639  0.48885304  0.08917919 -0.26579669]]

 [[ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]
  [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]]

 [[ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]
  [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]]
```

Static Unrolling Through Time

Checkout the complete code under **“Packing sequences”** section in notebook

Static Unrolling Through Time

- The previous approach still builds a graph
  - Containing one cell per time step
- If there were 50 time steps, the graph would look ugly
- It is like writing a program without using for loops
  - `Y0=f(0,X0); Y1=f(Y0, X1); Y2=f(Y1, X2); ...; Y50=f(Y49, X50))`
- With such a large graph
  - Since it must store all tensor values during the forward pass
  - So it can use them to compute gradients during the reverse pass
  - We may get out-of-memory (OOM) errors
  - During backpropagation (in GPU cards because of limited memory)

Dynamic Unrolling Through Time

Let's look at the better solution than previous approach using the dynamic_rnn() function

Dynamic Unrolling Through Time

- The dynamic_rnn() function uses a while_loop() operation to
  - Run over the cell the appropriate number of times
- We can set swap_memory=True
  - If we want it to swap the GPU's memory to the CPU's
  - Memory during backpropagation to avoid out of memory errors
- It also accepts a single tensor for
  - All inputs at every time step (shape [None, n_steps, n_inputs]) and
  - It outputs a single tensor for all outputs at every time step
    - (shape [None, n_steps, n_neurons])
  - There is no need to stack, unstack, or transpose

Dynamic Unrolling Through Time

RNN using dynamic_rnn

```
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> outputs, states = tf.nn.dynamic_rnn(basic_cell, X,
dtype=tf.float32)
```

Dynamic Unrolling Through Time

Checkout the complete code under **“Using dynamic_rnn()”** section in notebook

Dynamic Unrolling Through Time

Note

- During backpropagation
  - The while_loop() operation does the appropriate magic
  - It stores the tensor values for each iteration during the forward pass
  - So it can use them to compute gradients during the reverse pass

Handling Variable Length Input Sequences

- So far we have used only fixed-size input sequences
- What if the input sequences have variable lengths (e.g., like sentences)
- In this case we should set the sequence_length parameter
  - When calling the dynamic_rnn() function
  - It must be a 1D tensor indicating the length of the
  - Input sequence for each instance

```
seq_length = tf.placeholder(tf.int32, [None])

[...]
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
                                    sequence_length=seq_length)
```

Handling Variable Length Input Sequences

- Suppose the second input sequence contains
  - Only one input instead of two
  - Then It must be padded with a zero vector
  - In order to fit in the input tensor X

```
X_batch = np.array([
        # step 0     step 1
        [[0, 1, 2], [9, 8, 7]], # instance 0
        [[3, 4, 5], [0, 0, 0]], # instance 1 (padded with a zero vector)
        [[6, 7, 8], [6, 5, 4]], # instance 2
        [[9, 0, 1], [3, 2, 1]], # instance 3
    ])
seq_length_batch = np.array([2, 1, 2, 2])
```

Handling Variable Length Input Sequences

- Now we need to feed values for both placeholders X and seq_length

```
with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

Handling Variable Length Input Sequences

- Now the RNN outputs zero vectors for
  - Every time step past the input sequence length
  - Look at the second instance's output for the second time step

```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]  # final state

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]   # final state
  [ 0.          0.          0.          0.          0.        ]]  # zero vector

 [[ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]
  [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]]  # final state

 [[ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]
  [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]] # final state
```

Handling Variable Length Input Sequences

- Moreover, the states tensor contains the final state of each cell
  - Excluding the zero vectors

```
>>> print(states_val)
[[ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]    # t = 1
 [-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]    # t = 0 !!!
 [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]    # t = 1
 [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]   # t = 1
```

Handling Variable Length Input Sequences

Checkout the complete code under **“Setting the sequence lengths”** section in notebook

Handling Variable-Length Output Sequences

- What if the output sequences have variable lengths
- If we know in advance what length each sequence will have
  - For example if we know that it will be the same length as the input sequence
  - Then we can set the **sequence_length** parameter as discussed
- Unfortunately, in general this will not be possible
  - For example,
    - The length of a translated sentence is generally different from the
    - Length of the input sentence

Handling Variable-Length Output Sequences

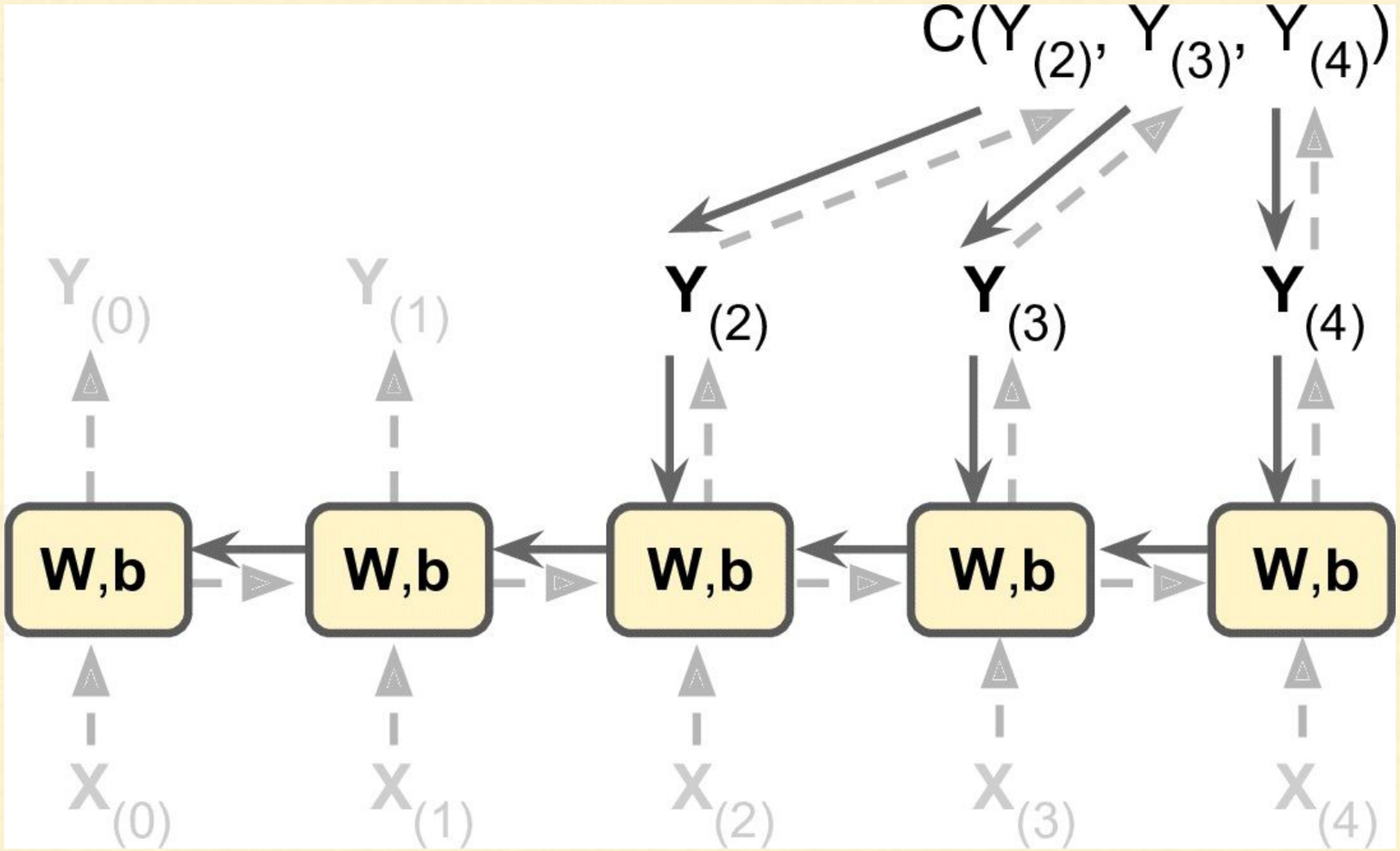
- In this case, the most common solution is to define
  - A special output called an **end-of-sequence token (EOS token)**
- Any output past the EOS should be ignored - We will discuss it later in details

Till now we have learnt how to build an RNN network. But how do we train it?

Training RNNs

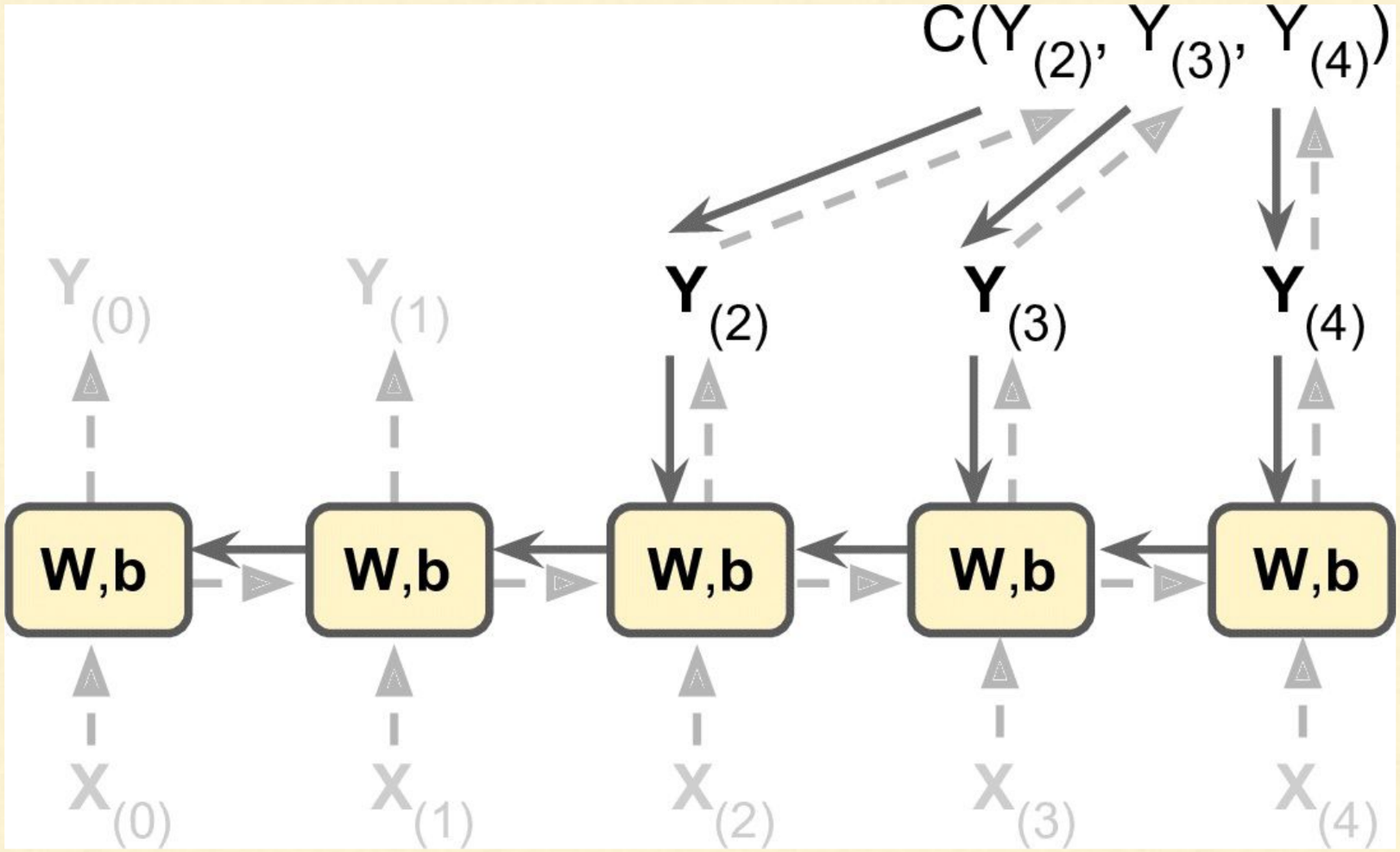
Training RNNs

- To train an RNN, the trick is to unroll it through time and then simply use **regular backpropagation**
- This strategy is called **backpropagation through time (BPTT)**



Training RNNs

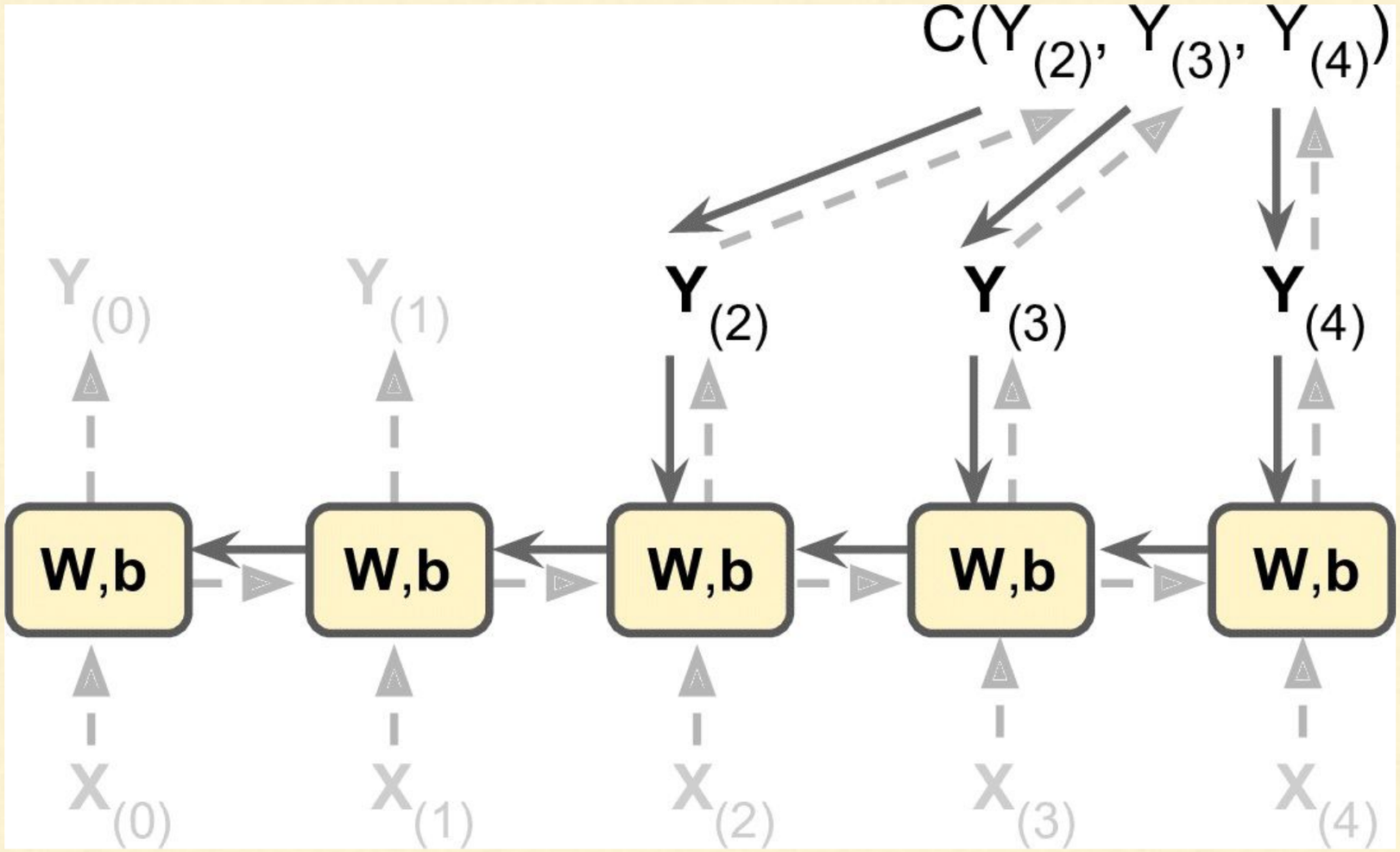
Understanding how RNNs are trained



Just like in regular backpropagation, there is a first forward pass through the unrolled network, represented by the dashed arrows

Training RNNs

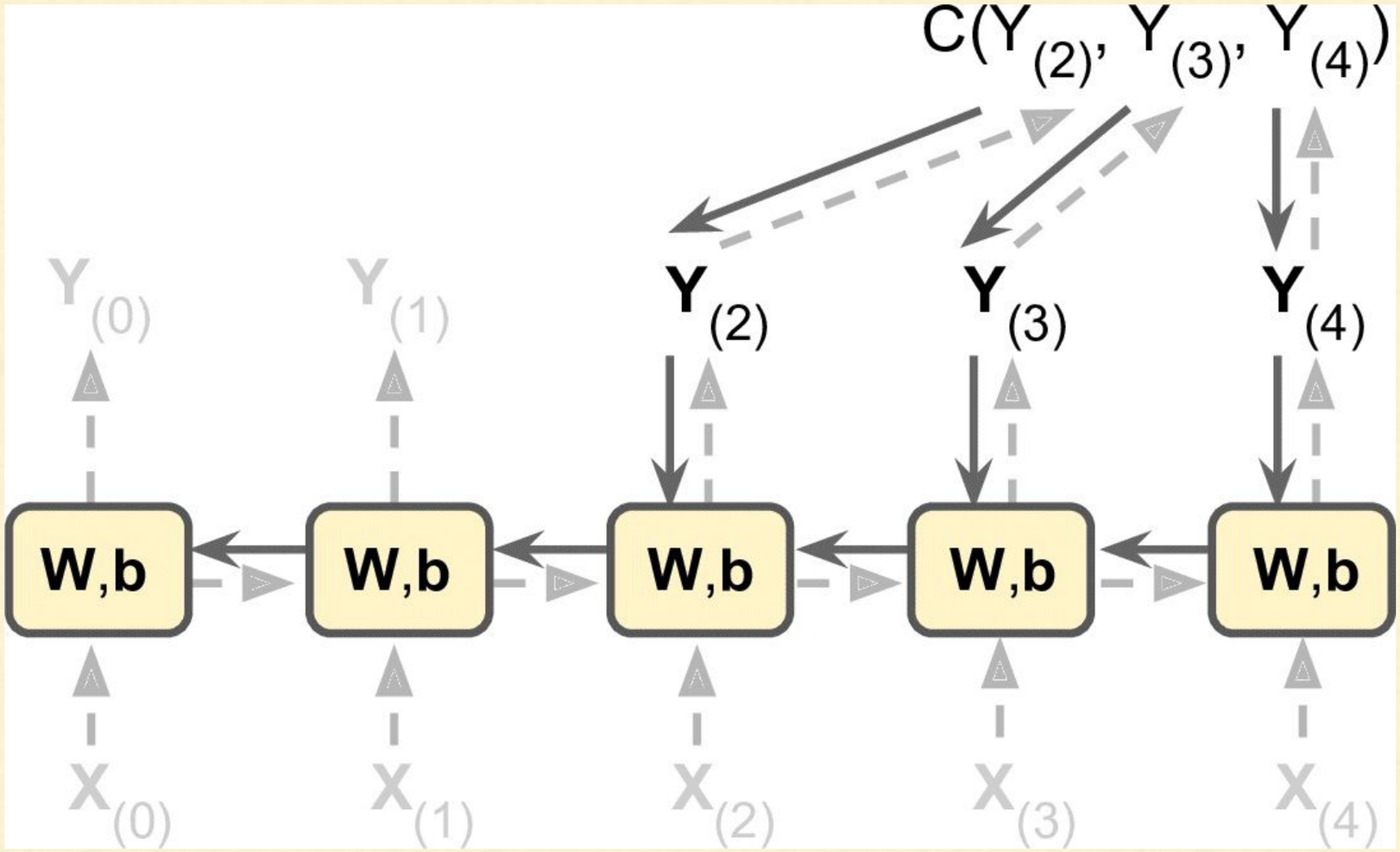
Understanding how RNNs are trained



Then the output sequence is evaluated using a cost function $C(\boldsymbol{Y}_{(t_{\min})}, \boldsymbol{Y}_{(t_{\min}+1)}, \cdots, \boldsymbol{Y}_{(t_{\max})})$ where t_{min} and t_{max} are the first and last output time steps, not counting the ignored outputs

Training RNNs

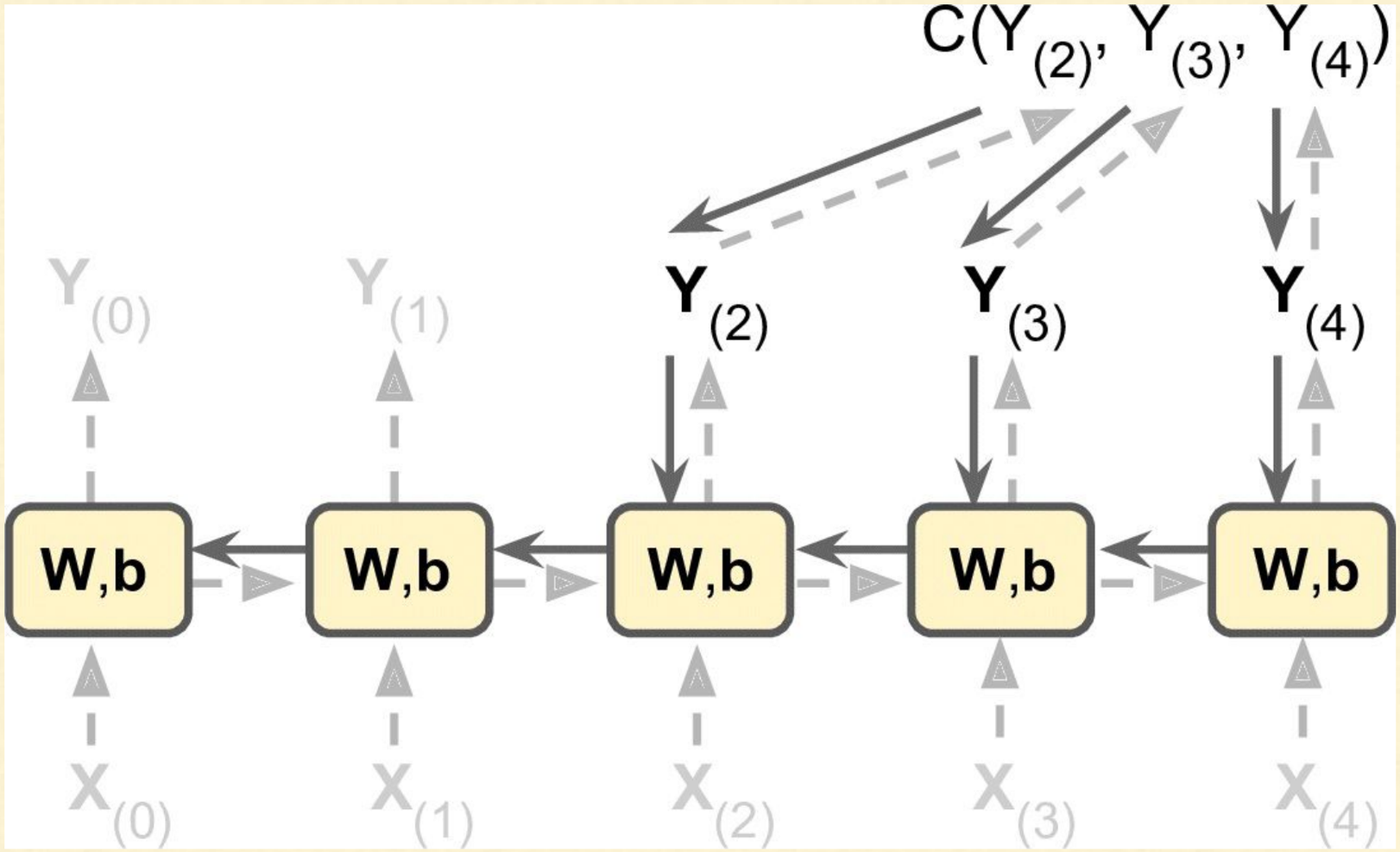
Understanding how RNNs are trained



Then the gradients of that cost function are propagated backward through the unrolled network, represented by the solid arrows

Training RNNs

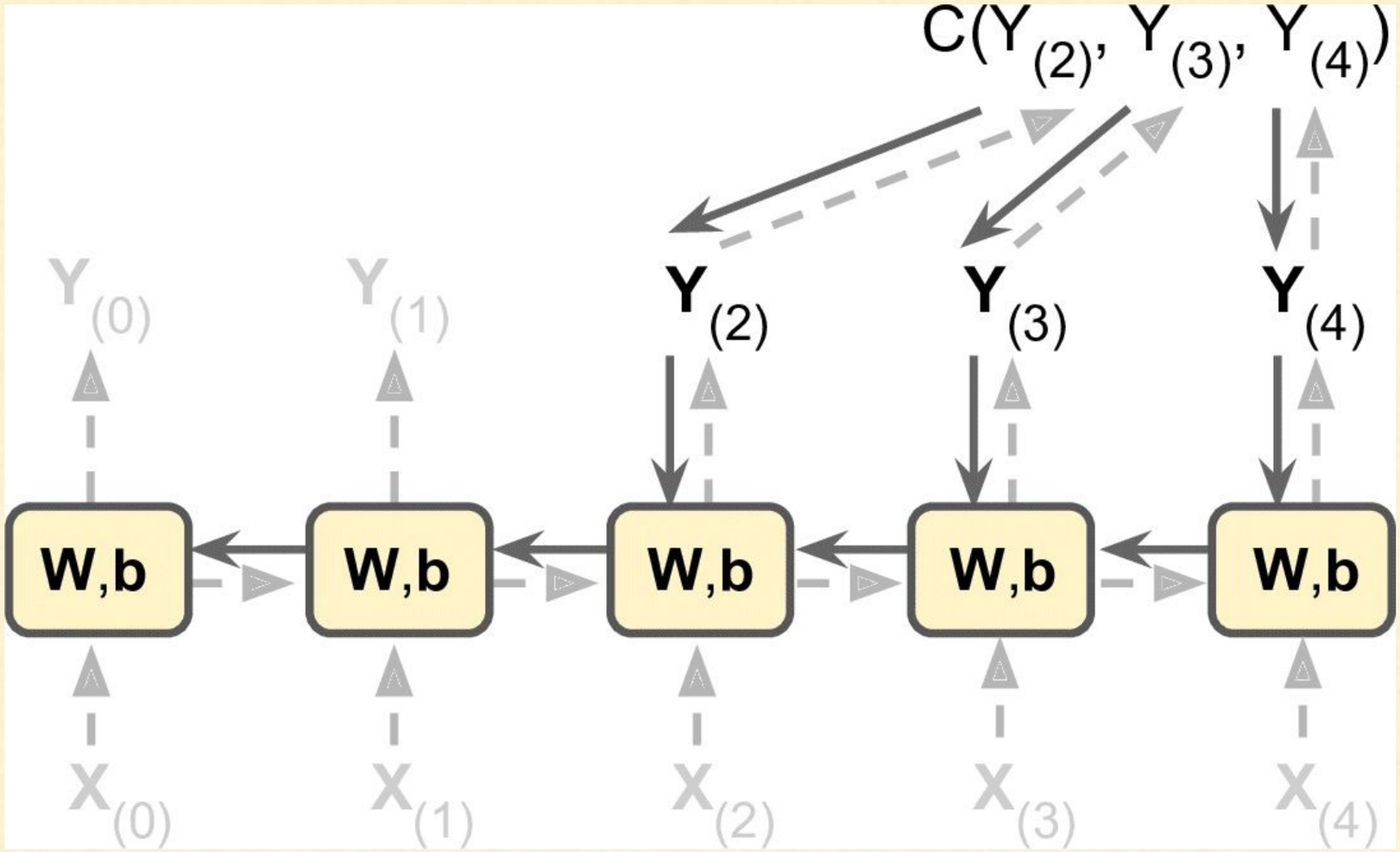
Understanding how RNNs are trained



And finally the model parameters are updated using the gradients computed during **BPTT**

Training RNNs

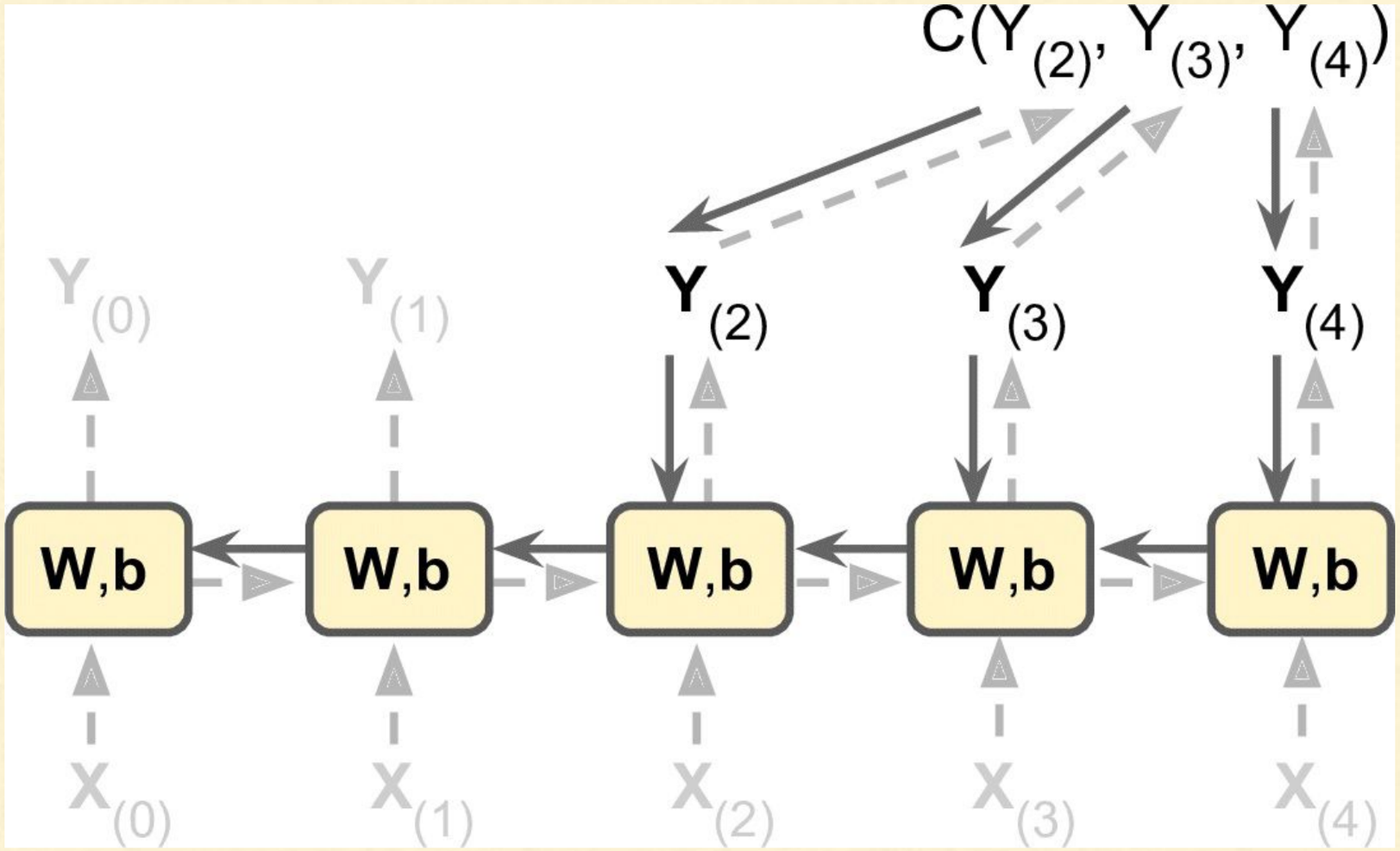
Understanding how RNNs are trained



Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output

Training RNNs

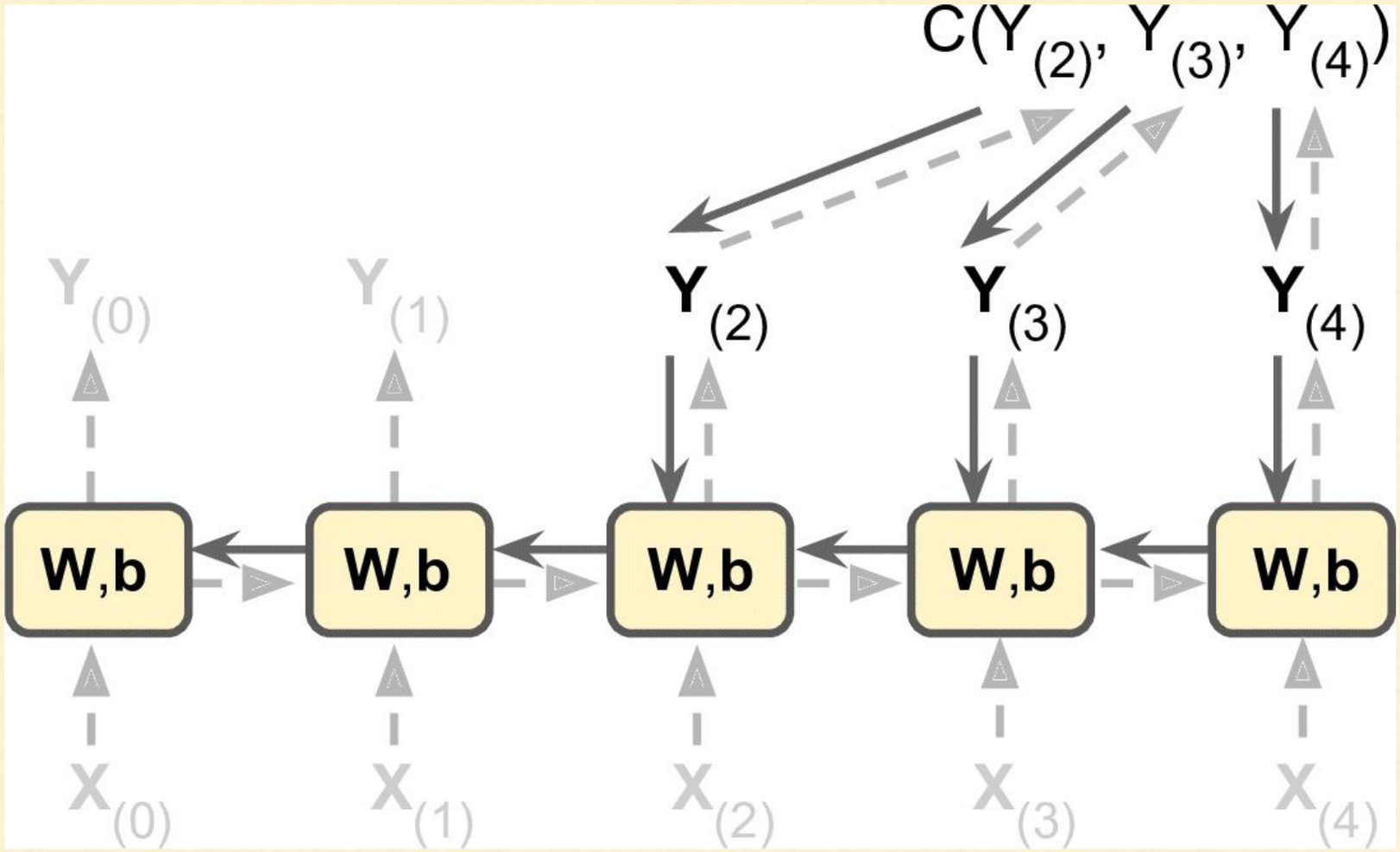
Understanding how RNNs are trained



Here, the cost function is computed using the last three outputs of the network, $\mathbf{Y}_{(2)}$, $\mathbf{Y}_{(3)}$, and $\mathbf{Y}_{(4)}$, so gradients flow through these three outputs, but not through $\mathbf{Y}_{(0)}$ and $\mathbf{Y}_{(1)}$

Training RNNs

Understanding how RNNs are trained



Moreover, since the same parameters **W and b** are used at each time step, backpropagation will do the right thing and sum over all time steps

Training a Sequence Classifier

Let's train an RNN to classify MNIST images

Training a Sequence Classifier

- A convolutional neural network would be better suited for image classification
- But this makes for a simple example that we are already familiar with

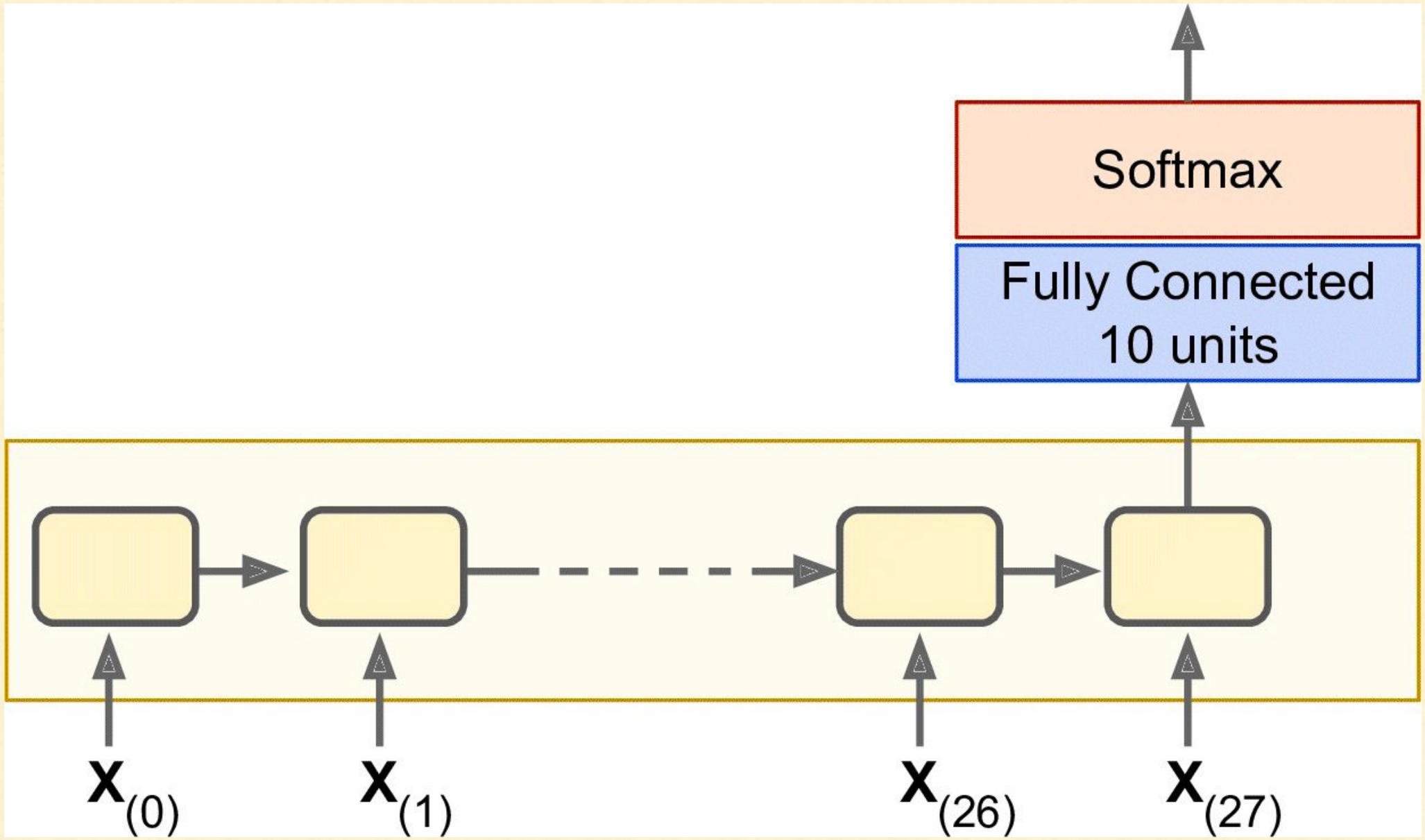
Training a Sequence Classifier

Overview of the task

- We will treat each image as a sequence of **28 rows of 28 pixels** each, since each MNIST image is **28 × 28 pixels**
- We will use cells of **150 recurrent neurons**, plus a fully connected layer containing **10 neurons**, one per class, connected to the output of the last time step
- This will be followed by a softmax layer

Training a Sequence Classifier

Overview of the task



Training a Sequence Classifier

Construction Phase

- The construction phase is quite straightforward
- It's pretty much the same as the MNIST classifier we built previously, except that an unrolled RNN replaces the hidden layers
- Note that the fully connected layer is connected to the states tensor, which contains only the final state of the RNN i.e., the 28th output

Training a Sequence Classifier

Construction Phase

```
>>> from tensorflow.contrib.layers import fully_connected
>>> n_steps = 28
>>> n_inputs = 28
>>> n_neurons = 150
>>> n_outputs = 10
>>> learning_rate = 0.001
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> y = tf.placeholder(tf.int32, [None])
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> outputs, states = tf.nn.dynamic_rnn(basic_cell, X,
dtype=tf.float32)
```

Run it on Notebook

Training a Sequence Classifier

Construction Phase

```
>>> logits = tf.layers.dense(states, n_outputs, activation_fn=None)
>>> xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
labels=y, logits=logits)
>>> loss = tf.reduce_mean(xentropy)
>>> optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
>>> training_op = optimizer.minimize(loss)
>>> correct = tf.nn.in_top_k(logits, y, 1)
>>> accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
>>> init = tf.global_variables_initializer()
```

Run it on Notebook

Training a Sequence Classifier

Load the MNIST data and reshape it

Now we will load the MNIST data and reshape the test data to [batch_size, n_steps, n_inputs] as is expected by the network

```
>>> from tensorflow.examples.tutorials.mnist import
input_data
>>> mnist = input_data.read_data_sets("data/mnist/")
>>> X_test = mnist.test.images.reshape((-1, n_steps,
n_inputs))
>>> y_test = mnist.test.labels
```

Run it on Notebook

Training a Sequence Classifier

Training the RNN

We reshape each training batch before feeding it to the network

```
>>> n_epochs = 100
>>> batch_size = 150
>>> with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```

Run it on Notebook

Training a Sequence Classifier

The Output

The output should look like this:

```
0 Train accuracy: 0.713333 Test accuracy: 0.7299

1 Train accuracy: 0.766667 Test accuracy: 0.7977

                          ...

98 Train accuracy: 0.986667 Test accuracy: 0.9777

99 Train accuracy: 0.986667 Test accuracy: 0.9809
```

Training a Sequence Classifier

Conclusion

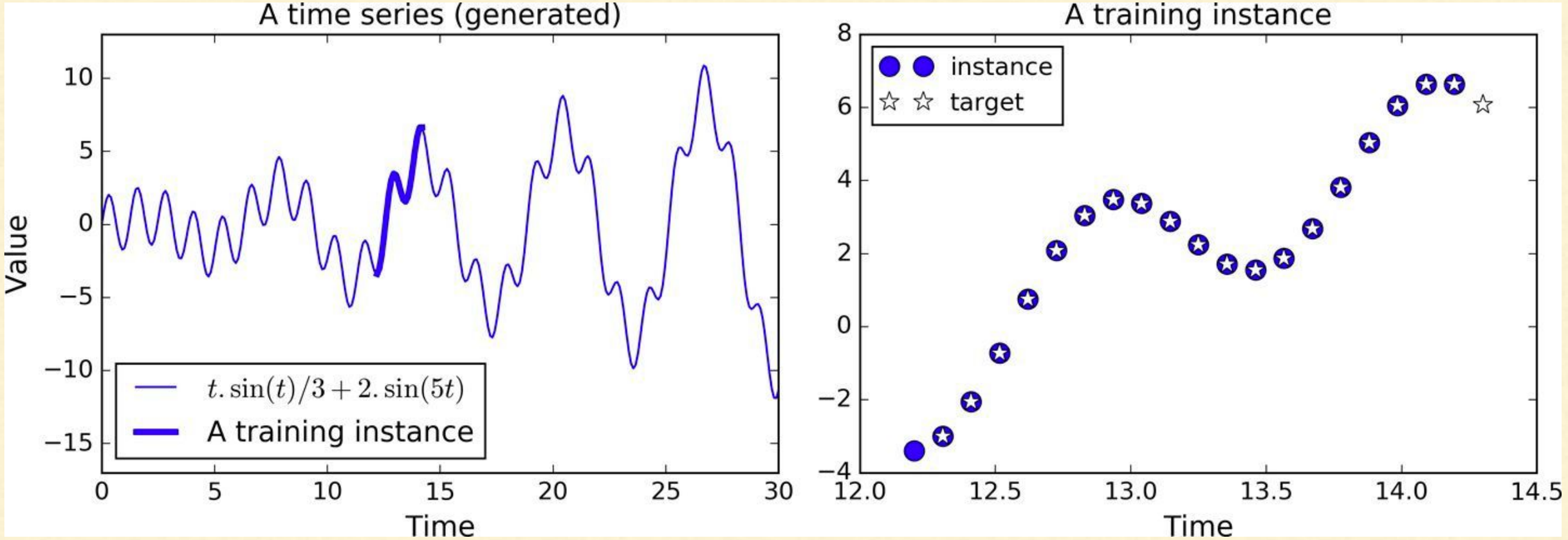
- We get over **98% accuracy** — not bad!
- Plus we would certainly get a better result by
  - Tuning the hyperparameters
  - Initializing the RNN weights using He initialization
  - Training longer
  - Or adding a bit of regularization e.g., dropout

Training to Predict Time Series

Now, we will train an RNN to predict the next value in a generated time series

Training to Predict Time Series

- Each training instance is a randomly selected sequence of 20 consecutive values from the time series
- And the target sequence is the same as the input sequence, except it is shifted by **one time step into the future**



Training to Predict Time Series

Construction Phase

- It will contain **100 recurrent neurons** and we will unroll it over **20 time steps** since each training instance will be **20 inputs long**
- Each input will contain only one feature, the value at that time
- The targets are also sequences of **20 inputs**, each containing a single value

Training to Predict Time Series

Construction Phase

```
>>> n_steps = 20
>>> n_inputs = 1
>>> n_neurons = 100
>>> n_outputs = 1
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
>>> cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
activation=tf.nn.relu)
>>> outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

Run it on Notebook

Training to Predict Time Series

Construction Phase

- At each time step we now have an output vector of size **100**
- But what we actually want is a single output value at each time step
- The simplest solution is to wrap the cell in an

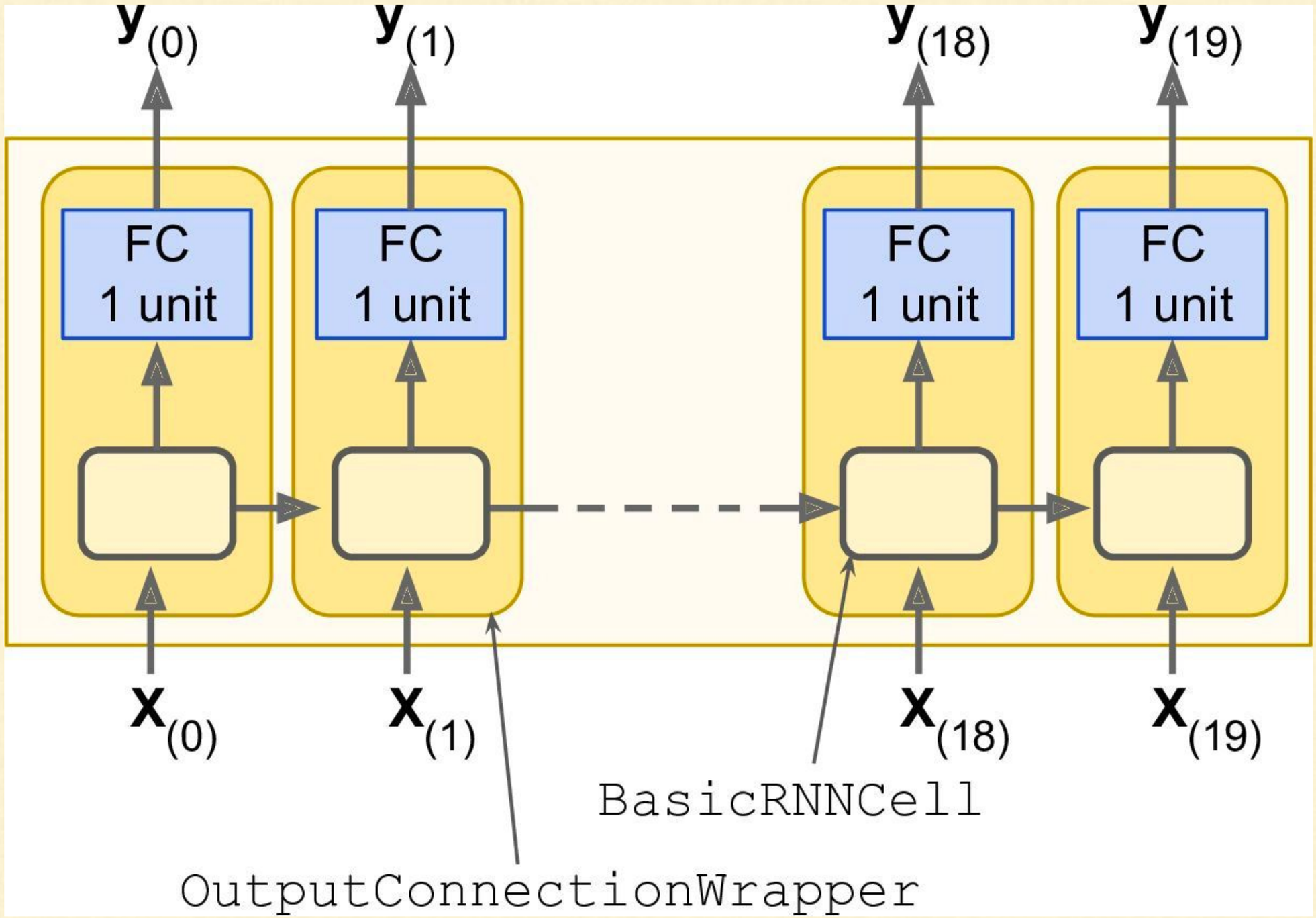
OutputProjectionWrapper

Training to Predict Time Series

Construction Phase

- A cell wrapper acts like a normal cell, proxying every method call to an underlying cell, but it also adds some functionality
- The **OutputProjectionWrapper** adds a fully connected layer of linear neurons i.e., **without any activation function** on top of each output, but it does not affect the cell state
- All these fully connected layers share the same trainable weights and bias terms.

Training to Predict Time Series



RNN cells using output projections

Training to Predict Time Series

Wrapping a cell is quite easy

Let's tweak the preceding code by wrapping the **BasicRNNCell** into an

OutputProjectionWrapper

```
>>> cell = tf.contrib.rnn.OutputProjectionWrapper(
        tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
activation=tf.nn.relu),output_size=n_outputs)
```

Run it on Notebook

Training to Predict Time Series

Cost Function and Optimizer

- Now we will define the cost function
- We will use the Mean Squared Error (MSE)
- Next we will create an Adam optimizer, the training op, and the variable initialization op
-

```
>>> learning_rate = 0.001
>>> loss = tf.reduce_mean(tf.square(outputs - y))
>>> optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
>>> training_op = optimizer.minimize(loss)
>>> init = tf.global_variables_initializer()
```

Run it on Notebook

Training to Predict Time Series

Execution Phase

```
>>> n_iterations = 10000
>>> batch_size = 50
>>> with tf.Session() as sess:
        init.run()
        for iteration in range(n_iterations):
            X_batch, y_batch = [...] # fetch the next training batch
            sess.run(training_op, feed_dict={X: X_batch, y:y_batch})
            if iteration % 100 == 0:
                mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
                print(iteration, "\tMSE:", mse)
```

Run it on Notebook

Training to Predict Time Series

Execution Phase

The program's output should look like this

```
0 MSE: 379.586

100 MSE: 14.58426

200 MSE: 7.14066

300 MSE: 3.98528

400 MSE: 2.00254

[...]
```

Training to Predict Time Series

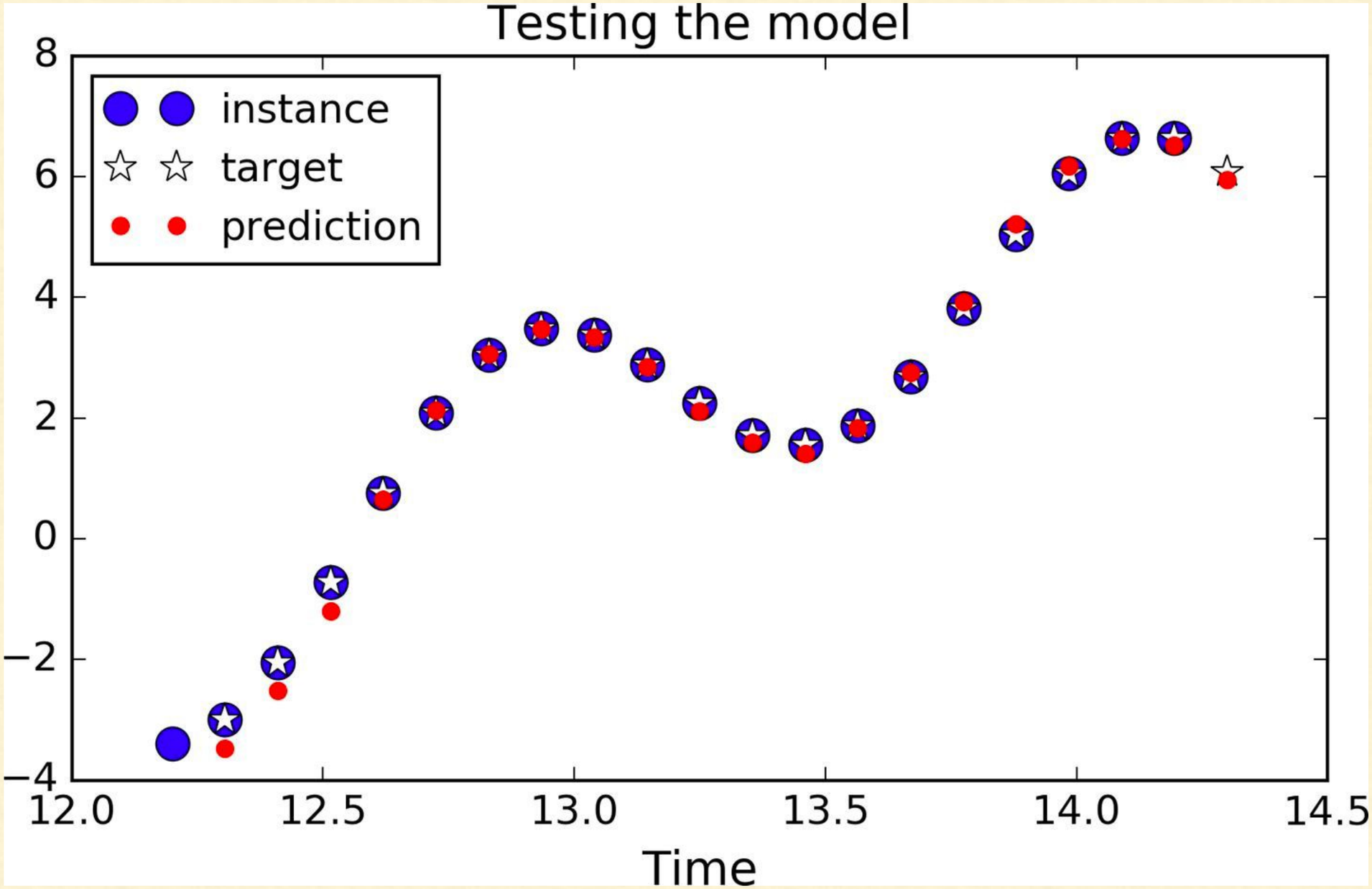
Making Predictions

Once the model is trained, you can make predictions:

```
>>> X_new = [...] # New sequences
>>> y_pred = sess.run(outputs, feed_dict={X: X_new})
```

Training to Predict Time Series

Making Predictions



Shows the predicted sequence for the instances, after 1,000 training iterations

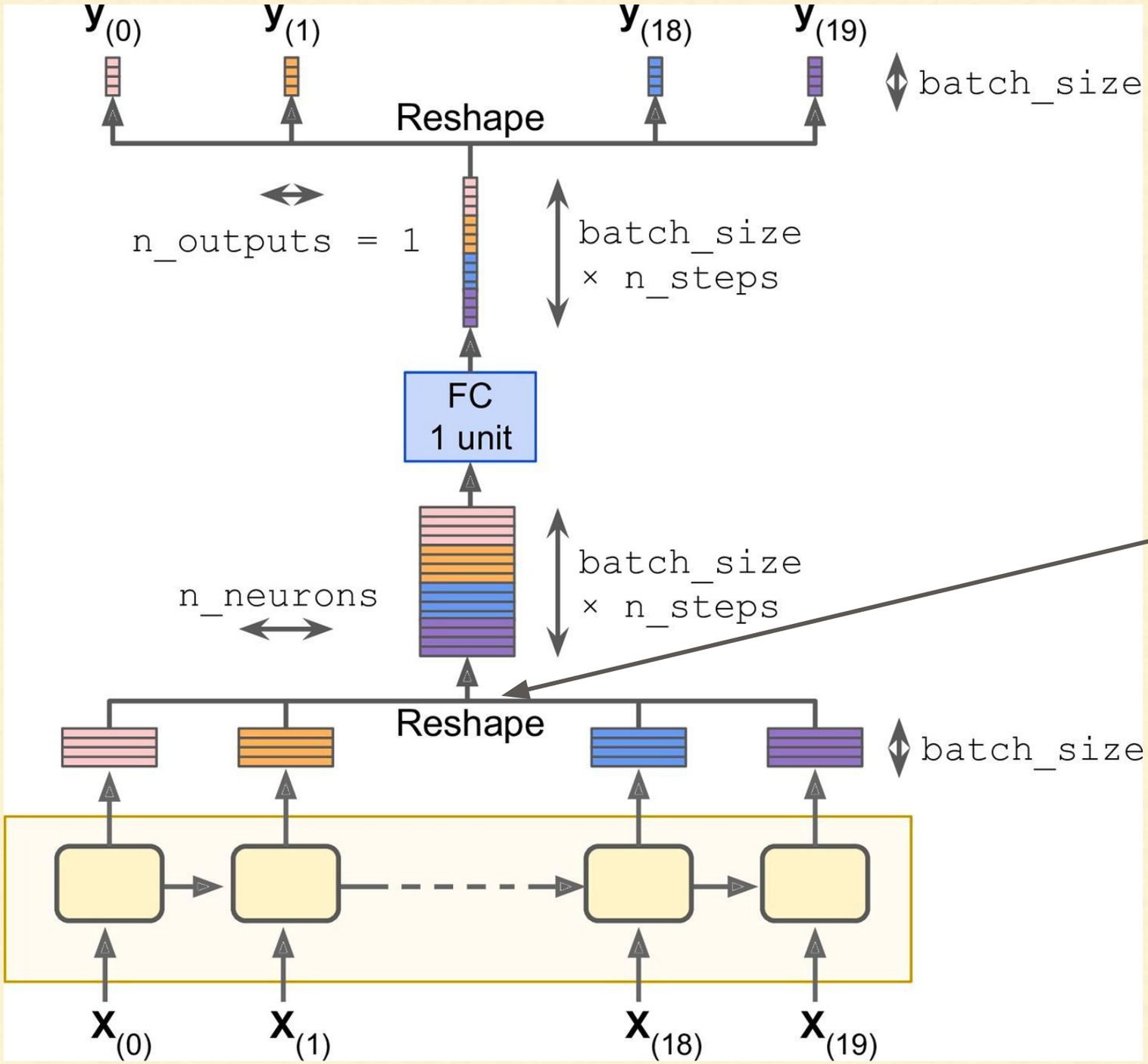
Training to Predict Time Series

- Although using an **OutputProjectionWrapper** is the simplest solution to reduce the dimensionality of the RNN's output sequences down to just one value per time step per instance
- **But** it is not the most efficient

Training to Predict Time Series

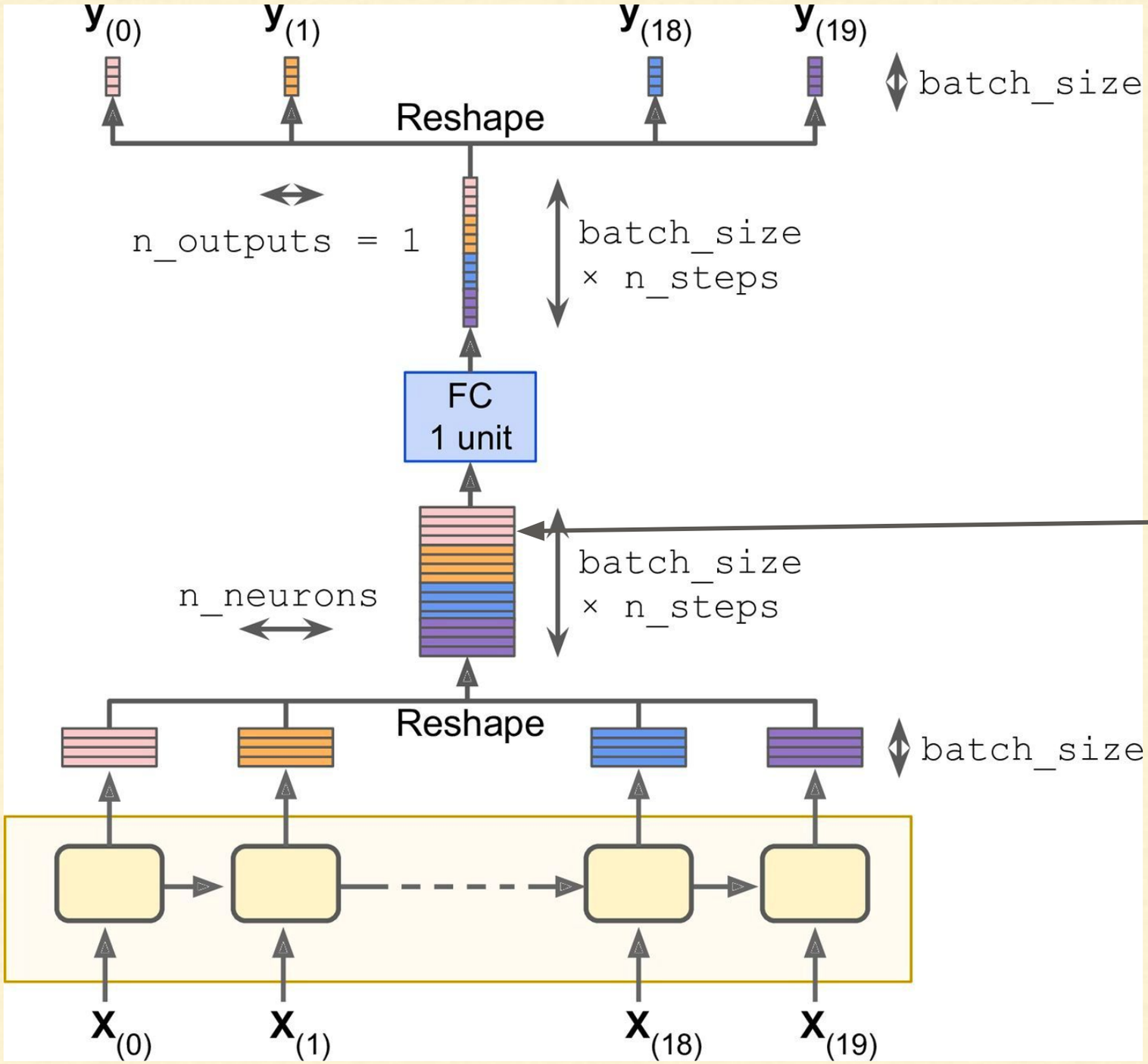
- There is a trickier but more efficient solution:
  - We can reshape the RNN outputs from **[batch_size, n_steps, n_neurons]** to **[batch_size * n_steps, n_neurons]**
  - Then apply a single fully connected layer with the appropriate output size in our case just **1**, which will result in an output tensor of shape **[batch_size * n_steps, n_outputs]**
  - And then reshape this tensor to **[batch_size, n_steps, n_outputs]**

Training to Predict Time Series



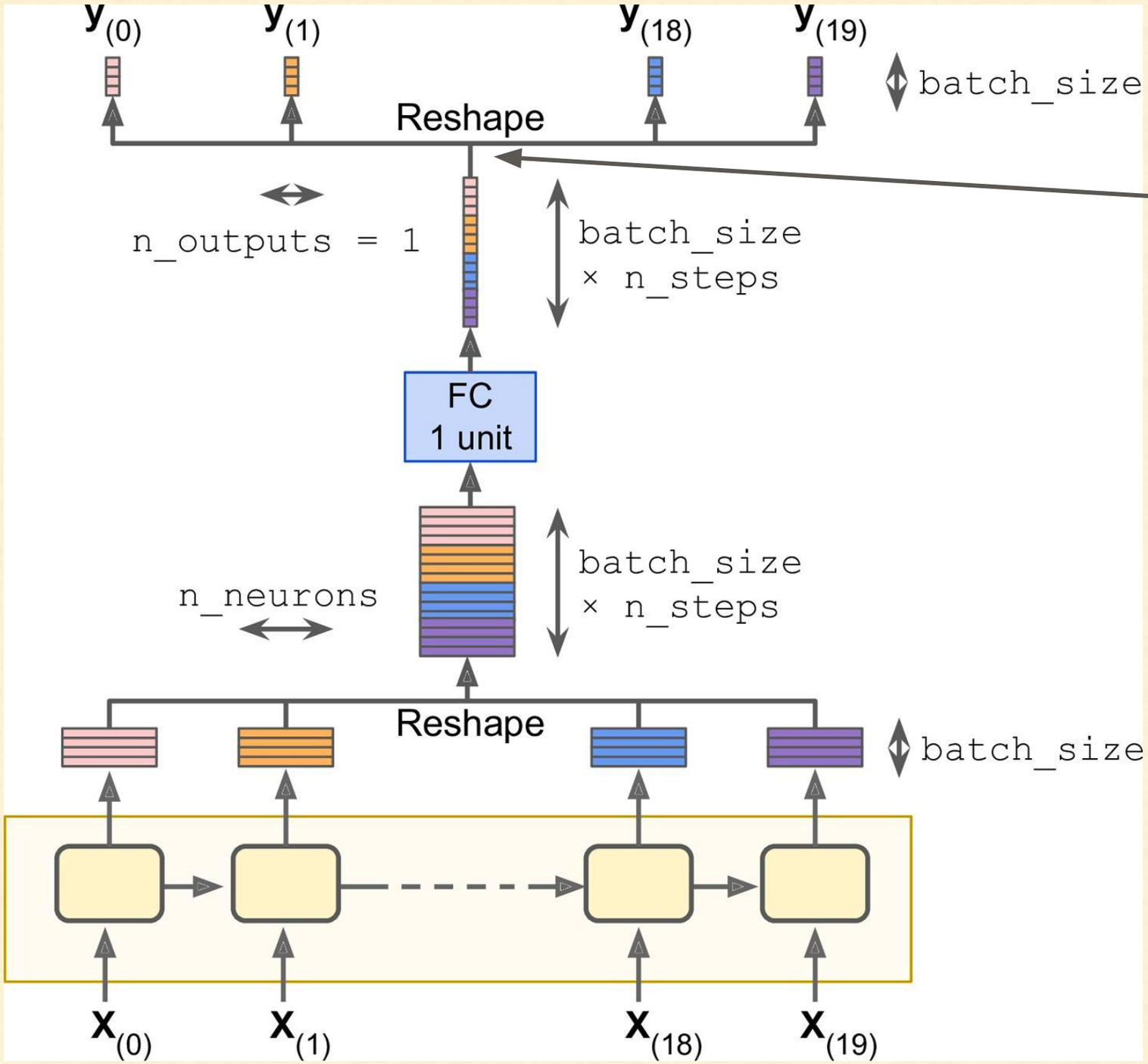
Reshape the RNN outputs from **[batch_size, n_steps, n_neurons]** to **[batch_size * n_steps, n_neurons]**

Training to Predict Time Series



Apply a single fully connected layer with the appropriate output size in our case just **1**, which will result in an output tensor of shape **[batch_size * n_steps, n_outputs]**

Training to Predict Time Series



And then reshape this tensor to **[batch_size, n_steps, n_outputs]**

Training to Predict Time Series

Let's implement this solution

- We first revert to a basic cell, without the **OutputProjectionWrapper**

```
>>> cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
activation=tf.nn.relu)
>>> rnn_outputs, states = tf.nn.dynamic_rnn(cell, X,
dtype=tf.float32)
```

Run it on Notebook

Training to Predict Time Series

Let's implement this solution

- Then we stack all the outputs using the **reshape()** operation, apply the fully connected linear layer without using any activation function; this is just a projection, and finally unstack all the outputs, again using **reshape()**

```
>>> stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
>>> stacked_outputs = fully_connected(stacked_rnn_outputs,
n_outputs, activation_fn=None)
>>> outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])
```

Run it on Notebook

Training to Predict Time Series

Let's implement this solution

- The rest of the code is the same as earlier. This can provide a significant speed boost since there is just one fully connected layer instead of one per time step.

Creative RNN

Let’s use our to generate some creative sequences

Creative RNN

- All we need is to provide it a **seed sequence** containing **n_steps values** e.g., full of zeros
- Use the model to predict the next value
- Append this predicted value to the sequence
- Feed the last **n_steps** values to the model to predict the next value
- And so on

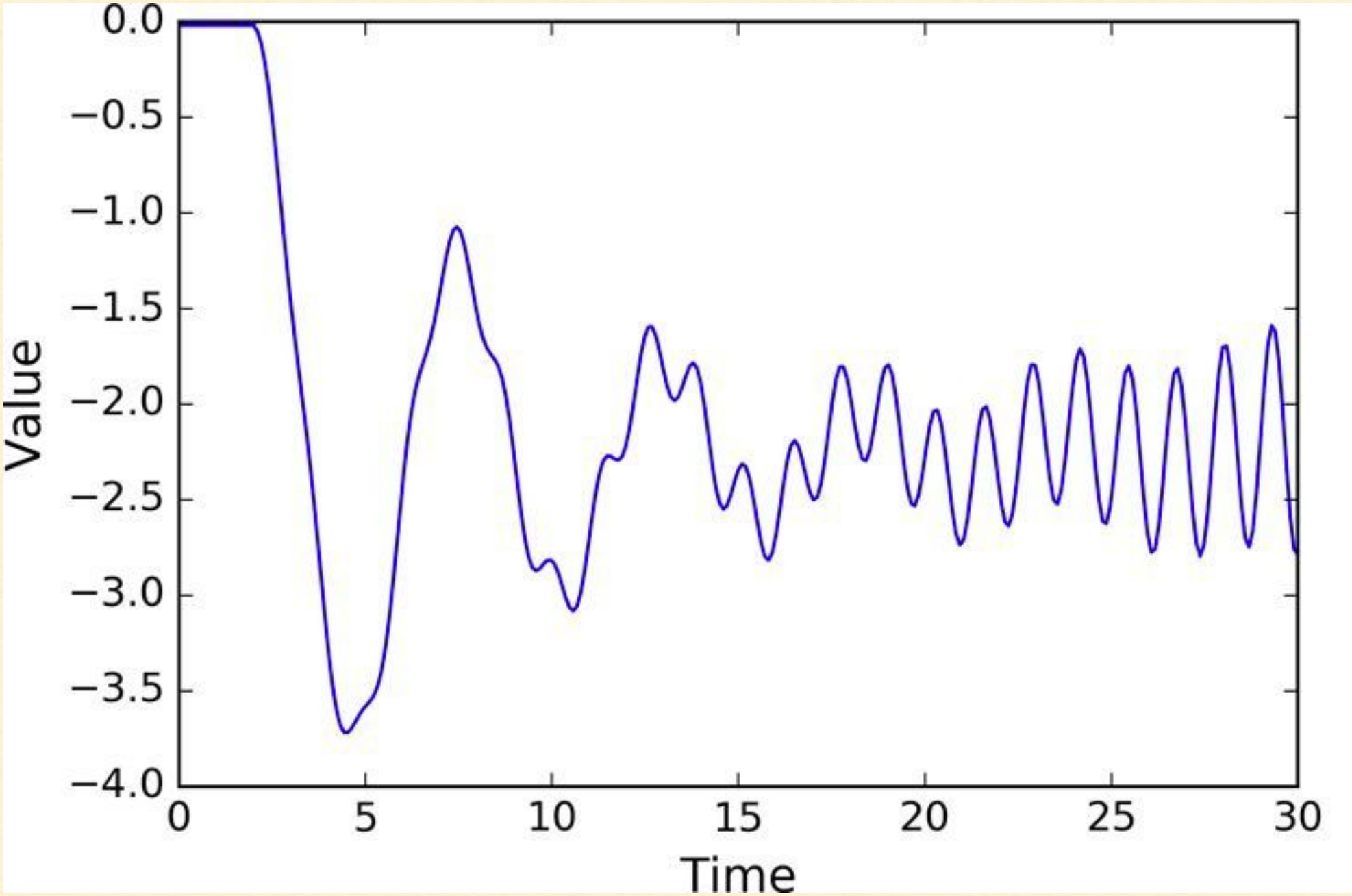
This process generates a new sequence that has some resemblance to the original time series

Creative RNN

```
>>> sequence = [0.] * n_steps
>>> for iteration in range(300):
        X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence.append(y_pred[0, -1, 0])
```

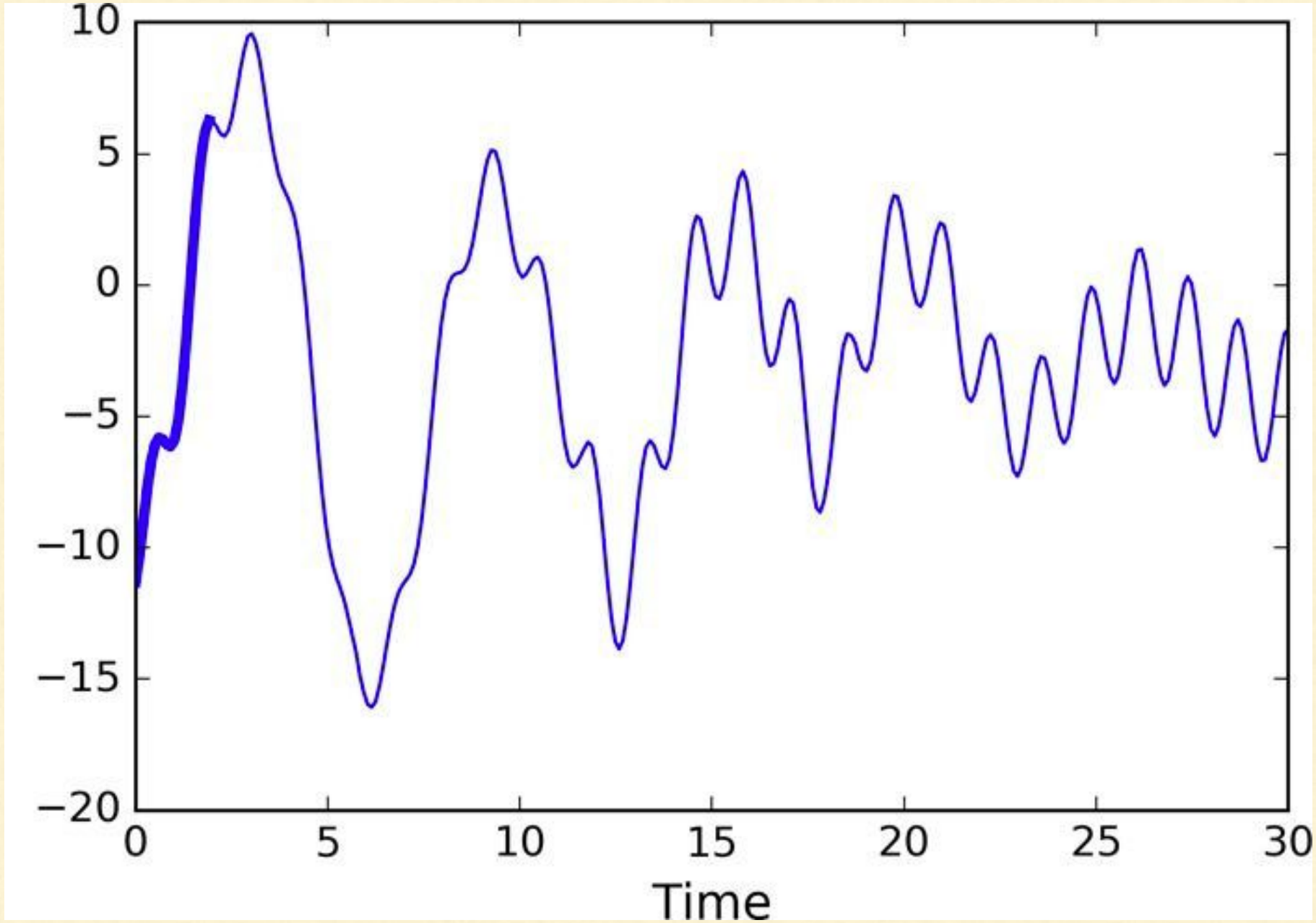
Run it on Notebook

Creative RNN



Creative sequences seeded with zeros

Creative RNN

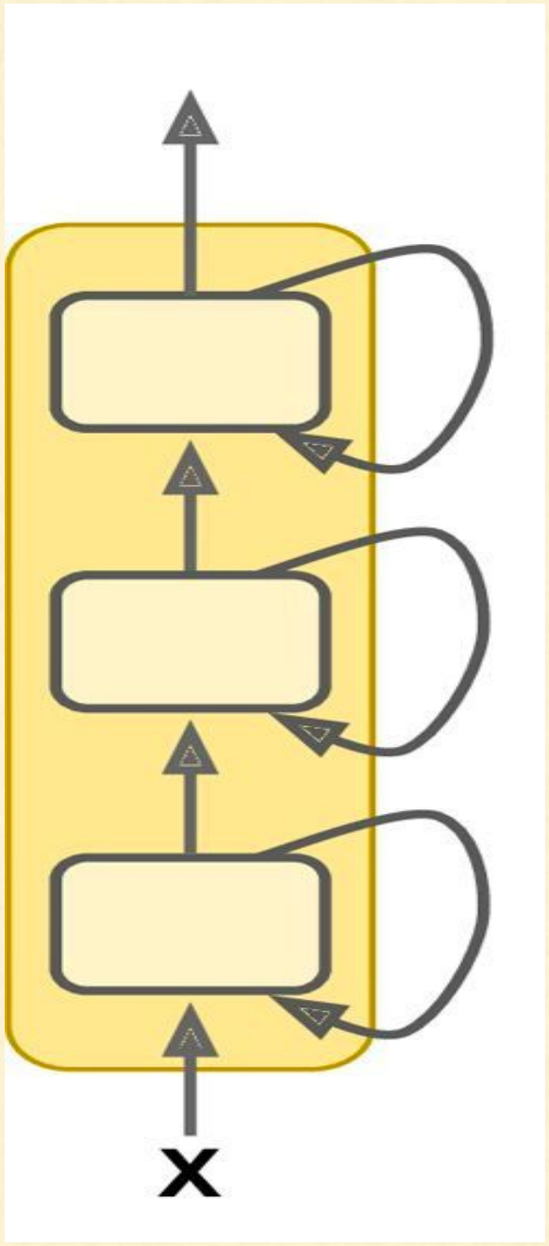


Creative sequences seeded with an instance

Deep RNNs

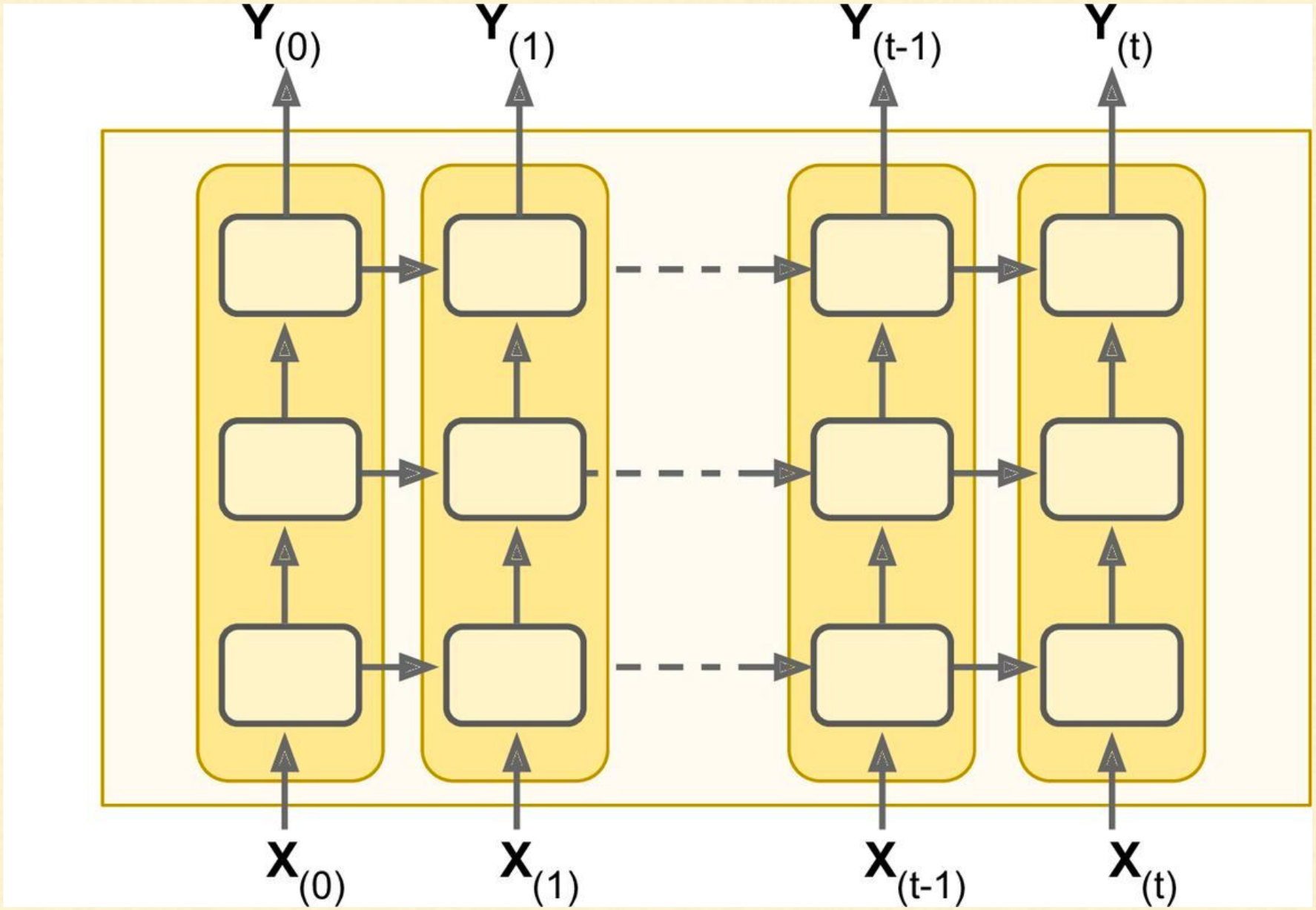
Deep RNNs

- It is quite common to stack multiple layers of cells.
- This gives you a **Deep RNN**



A Deep RNN

Deep RNNs



Deep RNN unrolled through time

Deep RNNs

How to implement Deep RNN in TensorFlow

Deep RNNs - Implementation in TensorFlow

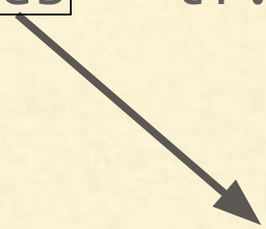
- To implement a deep RNN in TensorFlow
- We can create several cells and stack them into a **MultiRNNCell**
- In the following code we stack three identical cells

```
>>> n_neurons = 100
>>> n_layers = 3
>>> basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> multi_layer_cell = tf.contrib.rnn.MultiRNNCell([basic_cell] *
n_layers)
>>> outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X,
dtype=tf.float32)
```

Run it on Notebook

Deep RNNs - Implementation in TensorFlow

```
>>> outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X,
dtype=tf.float32)
```



- The *states* variable is a tuple containing one tensor per layer, each representing the final state of that layer's cell with shape **[batch_size, n_neurons]**
- If you set **state_is_tuple=False** when creating the **MultiRNNCell**, then states becomes a single tensor containing the states from every layer, concatenated along the column axis i.e., its shape is **[batch_size, n_layers * n_neurons]**

Deep RNNs - Applying Dropout

- If you build a very deep RNN, it may end up overfitting the training set
- To prevent that, a common technique is to apply **dropout**
- You can simply add a dropout layer before or after the RNN as usual
- But if you also want to apply dropout between the RNN layers, you need to use a **DropoutWrapper**

Deep RNNs - Applying Dropout

- The following code applies dropout to the inputs of each layer in the RNN, dropping each input with a 50% probability

```
>>> keep_prob = 0.5
>>> cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> cell_drop = tf.contrib.rnn.DropoutWrapper(cell,
input_keep_prob=keep_prob)
>>> multi_layer_cell = tf.contrib.rnn.MultiRNNCell([cell_drop] *
n_layers)
>>> rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X,
dtype=tf.float32)
```

Run it on Notebook

Deep RNNs - Applying Dropout

- It is also possible to apply dropout to the outputs by setting **output_keep_prob**
- The main problem with this code is that it will apply dropout not only during training but also during testing, which is **not what we want**
- **Since dropout should be applied only during training**

Deep RNNs - Applying Dropout

- Unfortunately, the **DropoutWrapper** does not support an is_training placeholder
- So we must either write our own dropout wrapper class, or have two different graphs:
  - One for training
  - And the other for testing

Let's implement the second option

Deep RNNs - Applying Dropout

```
>>> import sys
>>> is_training = (sys.argv[-1] == "train")
>>> X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
>>> y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
>>> cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
>>> if is_training:
        cell = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
>>> multi_layer_cell = tf.contrib.rnn.MultiRNNCell([cell] * n_layers)
>>> rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X,
dtype=tf.float32)
    [...] # build the rest of the graph
>>> init = tf.global_variables_initializer()
>>> saver = tf.train.Saver()
>>> with tf.Session() as sess:
>>> if is_training:
        init.run()
        for iteration in range(n_iterations):
            [...] # train the model
        save_path = saver.save(sess, "/tmp/my_model.ckpt")
    else:
        saver.restore(sess, "/tmp/my_model.ckpt")
        [...] # use the model
```

Run it on Notebook

Deep RNNs

The Difficulty of Training over Many Time Steps

- To train an RNN on long sequences, we will need to run it over many time steps, making the unrolled RNN a very deep network
- Just like any deep neural network it may suffer from the vanishing/exploding gradients problem and take forever to train

Deep RNNs

The Difficulty of Training over Many Time Steps

- Many of the tricks we discussed to alleviate this problem can be used for deep unrolled RNNs as well:
  - good parameter initialization,
  - nonsaturating activation functions e.g., ReLU
  - Batch Normalization,
  - Gradient Clipping,
  - And faster optimizers

Deep RNNs

The Difficulty of Training over Many Time Steps

- However, if the RNN needs to handle even moderately long sequences e.g., 100 inputs, then training will still be very slow
- The simplest and most common solution to this problem is to unroll the RNN only over a limited number of time steps during training
- This is called **truncated backpropagation through time**

Deep RNNs

The Difficulty of Training over Many Time Steps

- However, if the RNN needs to handle even moderately long sequences e.g., 100 inputs, then training will still be very slow
- The simplest and most common solution to this problem is to unroll the RNN only over a limited number of time steps during training
- This is called **truncated backpropagation through time**

Deep RNNs

The Difficulty of Training over Many Time Steps

- In TensorFlow you can implement **truncated backpropagation through time** by simply by truncating the input sequences
- For example, in the time series prediction problem, you would simply reduce **n_steps** during training
- The problem with this is that the model will not be able to learn long-term patterns

How can we solve this problem?

Deep RNNs

The Difficulty of Training over Many Time Steps

- One workaround could be to make sure that these shortened sequences contain both **old and recent data**
- So that the model can learn to use both
- E.g., the sequence could contain monthly data for the last five months, then weekly data for the last five weeks, then daily data over the last five days
- But this workaround has its limits:
  - What if fine-grained data from last year is actually useful?
  - What if there was a brief but significant event that absolutely must be taken into account, even years later
  - E.g., the result of an election

Deep RNNs

The Difficulty of Training over Many Time Steps

- Besides the long training time
  - A second problem faced by long-running RNNs is the fact that the memory of the first inputs gradually fades away
  - Indeed, due to the transformations that the data goes through when traversing an RNN, some information is lost after each time step.
- After a while, the RNN's state contains virtually no trace of the first inputs

Let's understand this with an example

Deep RNNs

The Difficulty of Training over Many Time Steps

- Say you want to perform sentiment analysis on a long review that starts with the four words **"I loved this movie,"**
- But the rest of the review lists the many things that could have made the movie even better
- If the RNN gradually forgets the first four words, it will completely misinterpret the review

Deep RNNs

The Difficulty of Training over Many Time Steps

- To solve this problem, various types of cells with long-term memory have been introduced
- They have proved so successful that the basic cells are not much used anymore

Let's study about these long memory cells

LSTM Cell

LSTM Cell

- The Long Short-Term Memory (LSTM) cell was proposed in 19973 by **Sepp Hochreiter and Jürgen Schmidhuber**
- And it was gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak, Wojciech Zaremba, and many more

Sepp Hochreiter

Jürgen Schmidhuber

LSTM Cell

- If you consider the LSTM cell as a black box, it can be used very much like a basic cell
- **Except**
  - It will perform much better
  - Training will converge faster
  - And it will detect long-term dependencies in the data

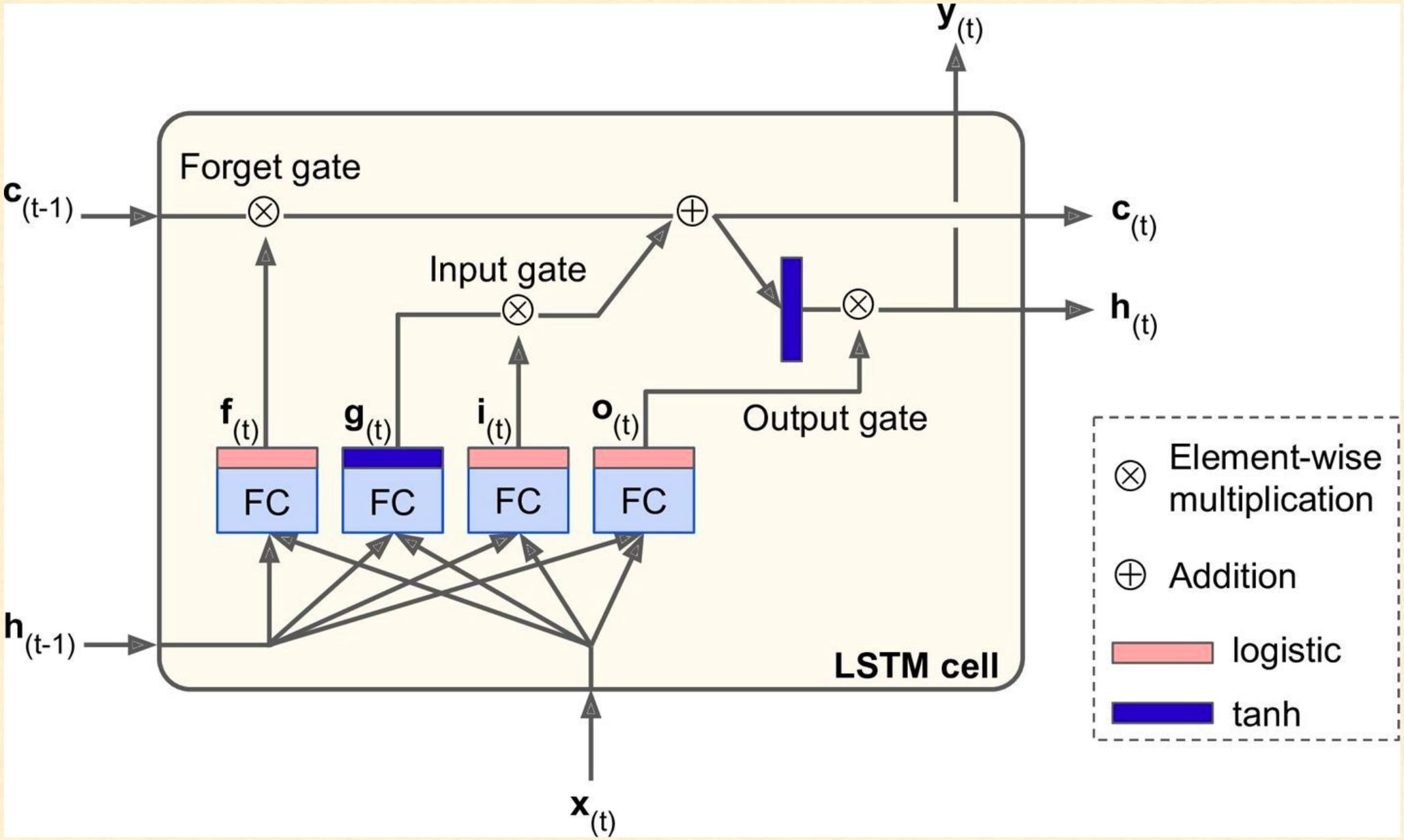
In TensorFlow, you can simply use a **BasicLSTMCell** instead of a **BasicRNNCell**

```
>>> lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

LSTM Cell

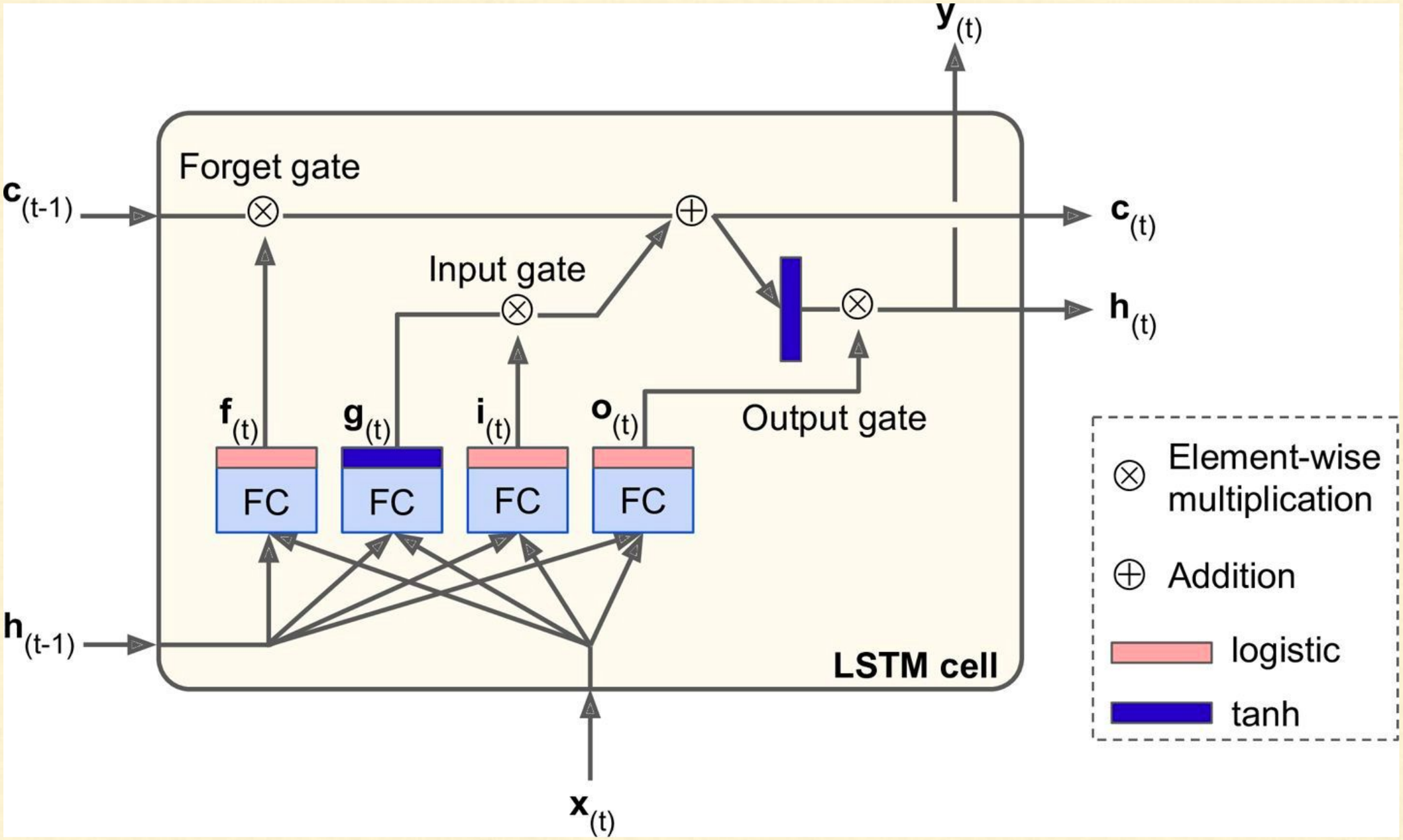
- LSTM cells manage two state vectors, and for performance reasons they are kept separate by default
- We can change this default behavior by setting **state_is_tuple=False** when creating the **BasicLSTMCell**

LSTM Cell



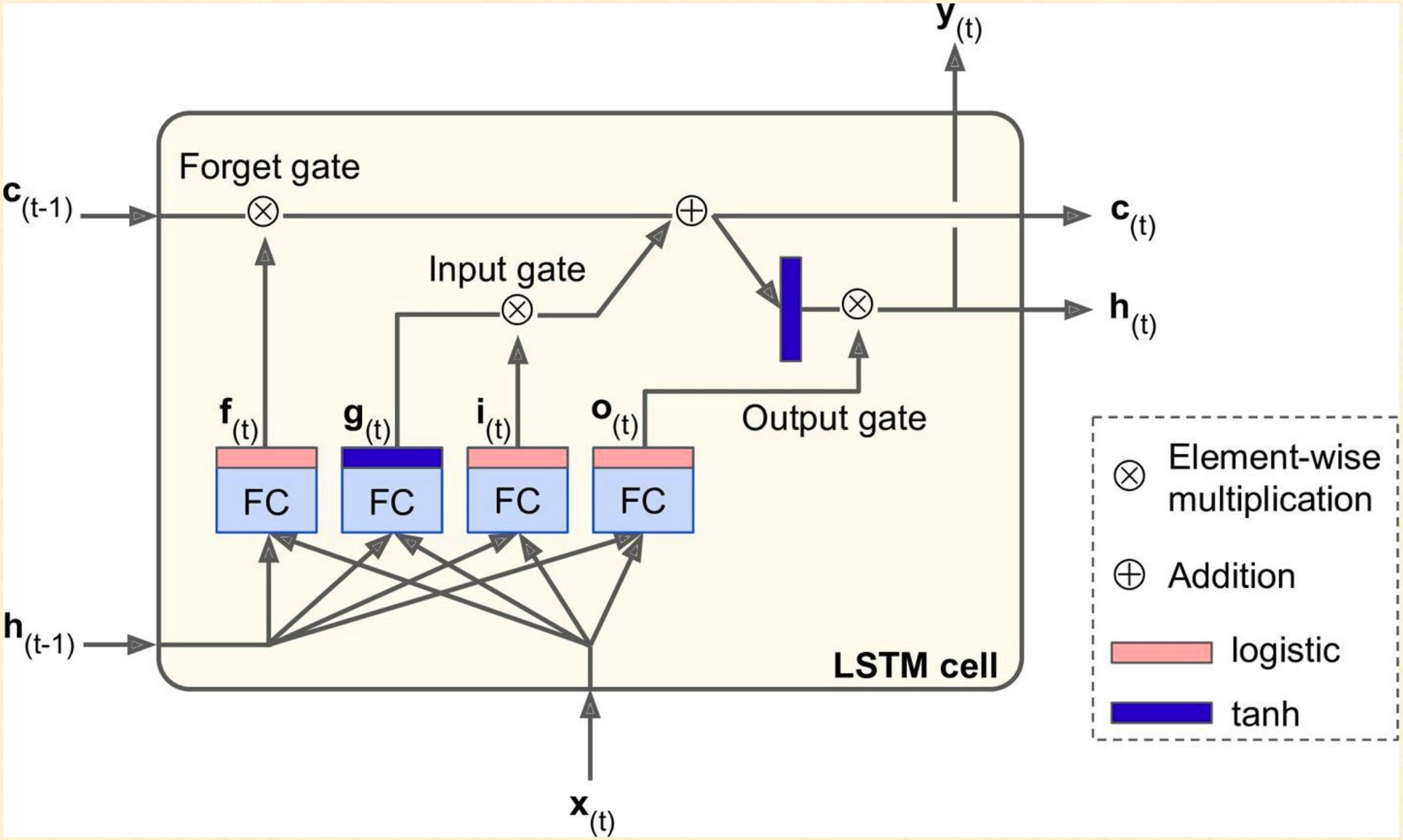
The architecture of a basic LSTM cell

LSTM Cell



- The **LSTM cell** looks exactly like a regular cell, except that its state is split in two vectors: $h_{(t)}$ and $c_{(t)}$, here "c" stands for "cell"

LSTM Cell



- We can think of $h_{(t)}$ as the short-term state and $c_{(t)}$ as the long-term state

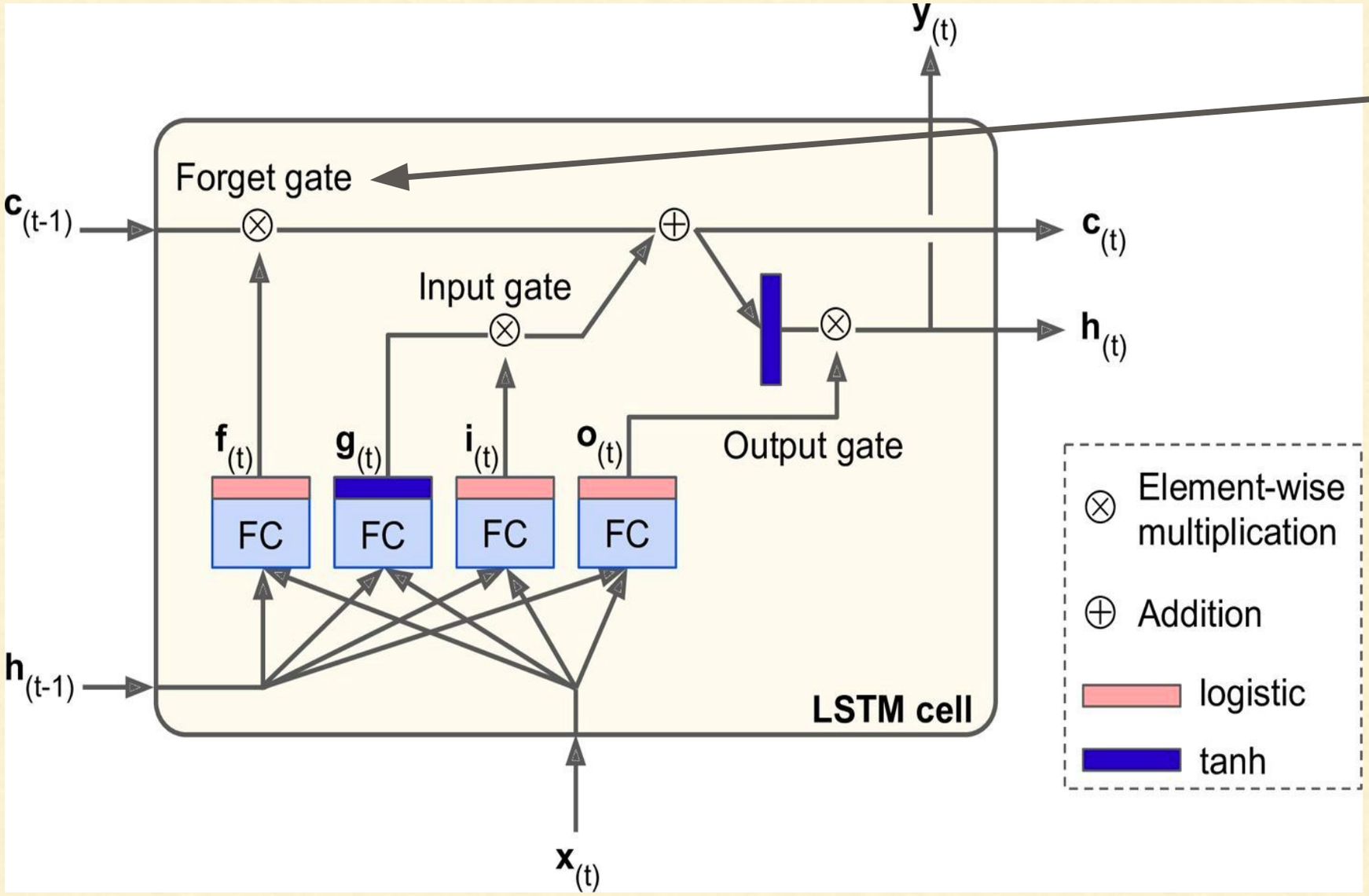
LSTM Cell

Understanding the LSTM cell structure

- The key idea is that the network **can learn**
  - What to store in the long-term state,
  - What to throw away,
  - And what to read from it

LSTM Cell

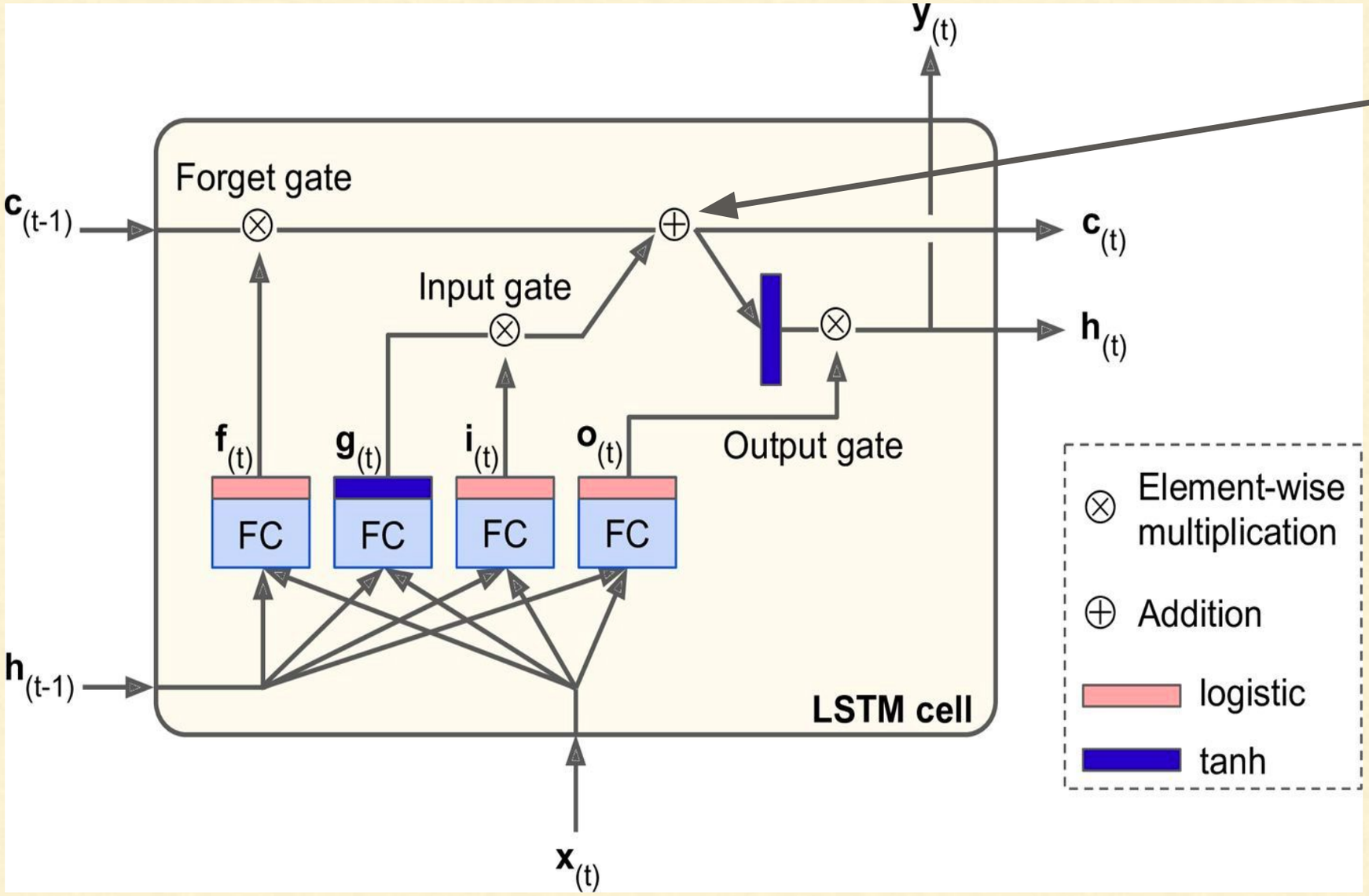
Understanding the LSTM cell structure



As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right, it first goes through a **forget gate**, dropping some memories

LSTM Cell

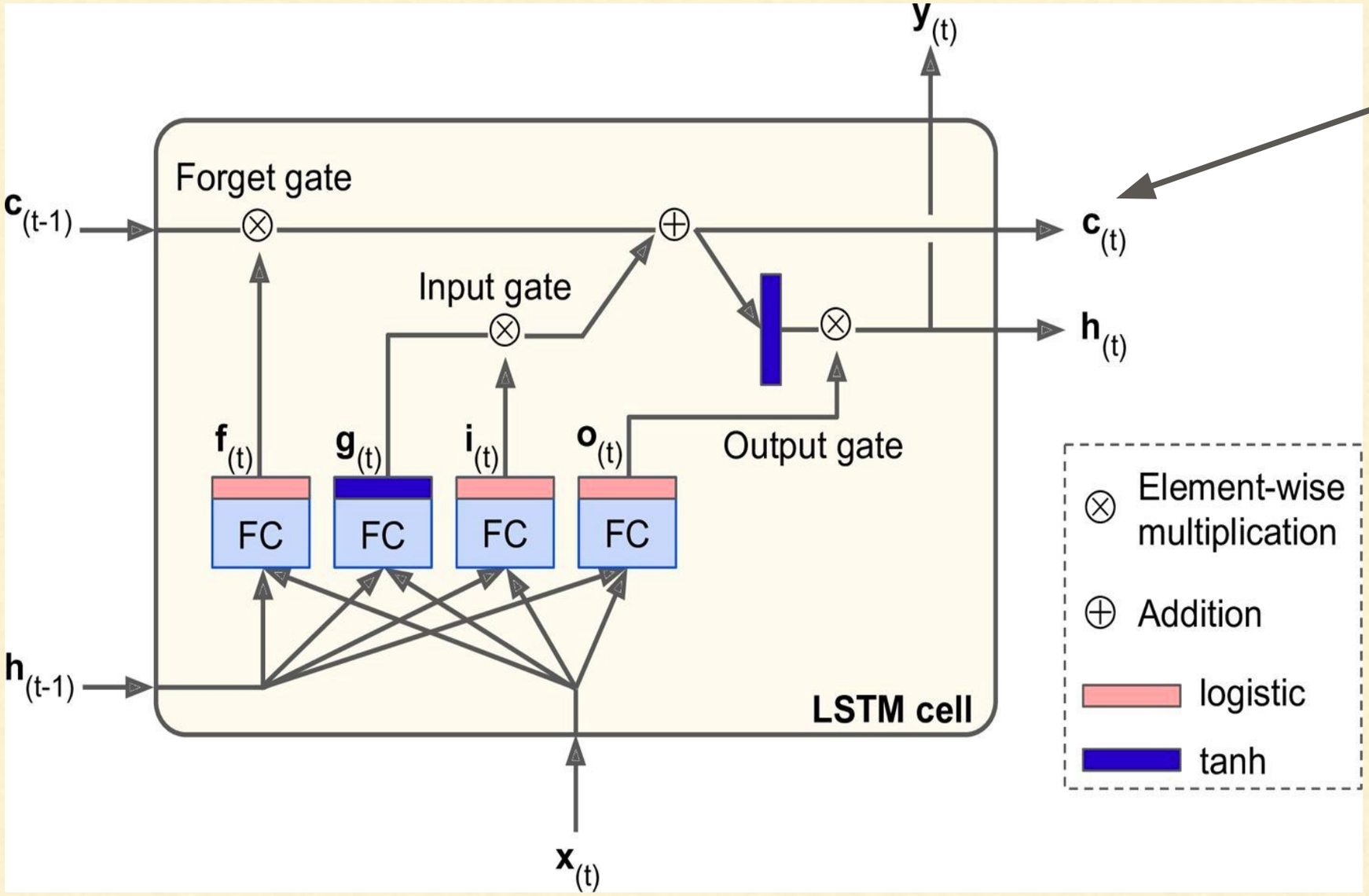
Understanding the LSTM cell structure



And then it adds some new memories via the addition operation, which adds the memories that were selected by an input gate

LSTM Cell

Understanding the LSTM cell structure

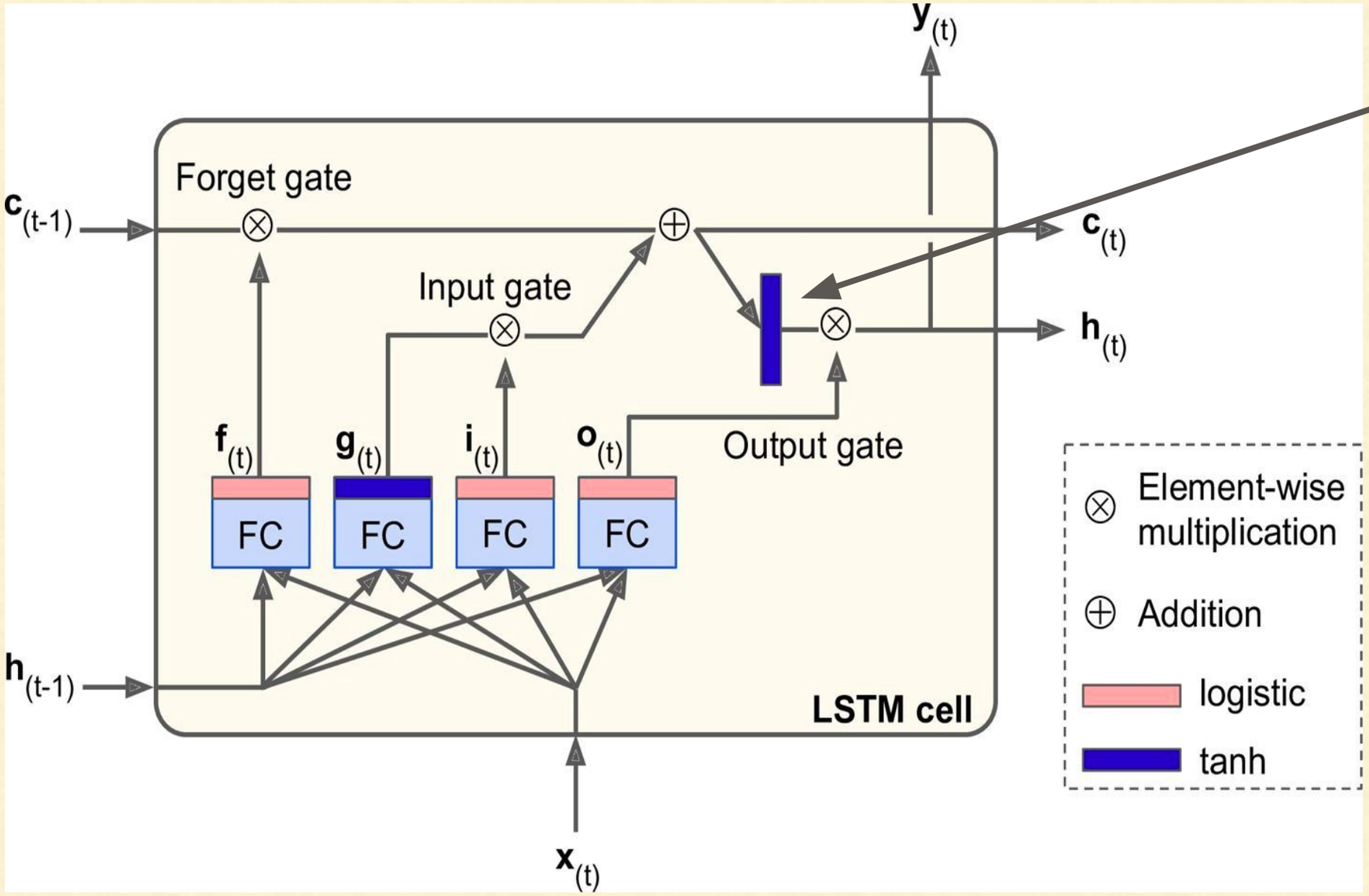


The result $\mathbf{c}_{(t)}$ is sent straight out, without any further transformation.

So, at each time step, some memories are dropped and some memories are added

LSTM Cell

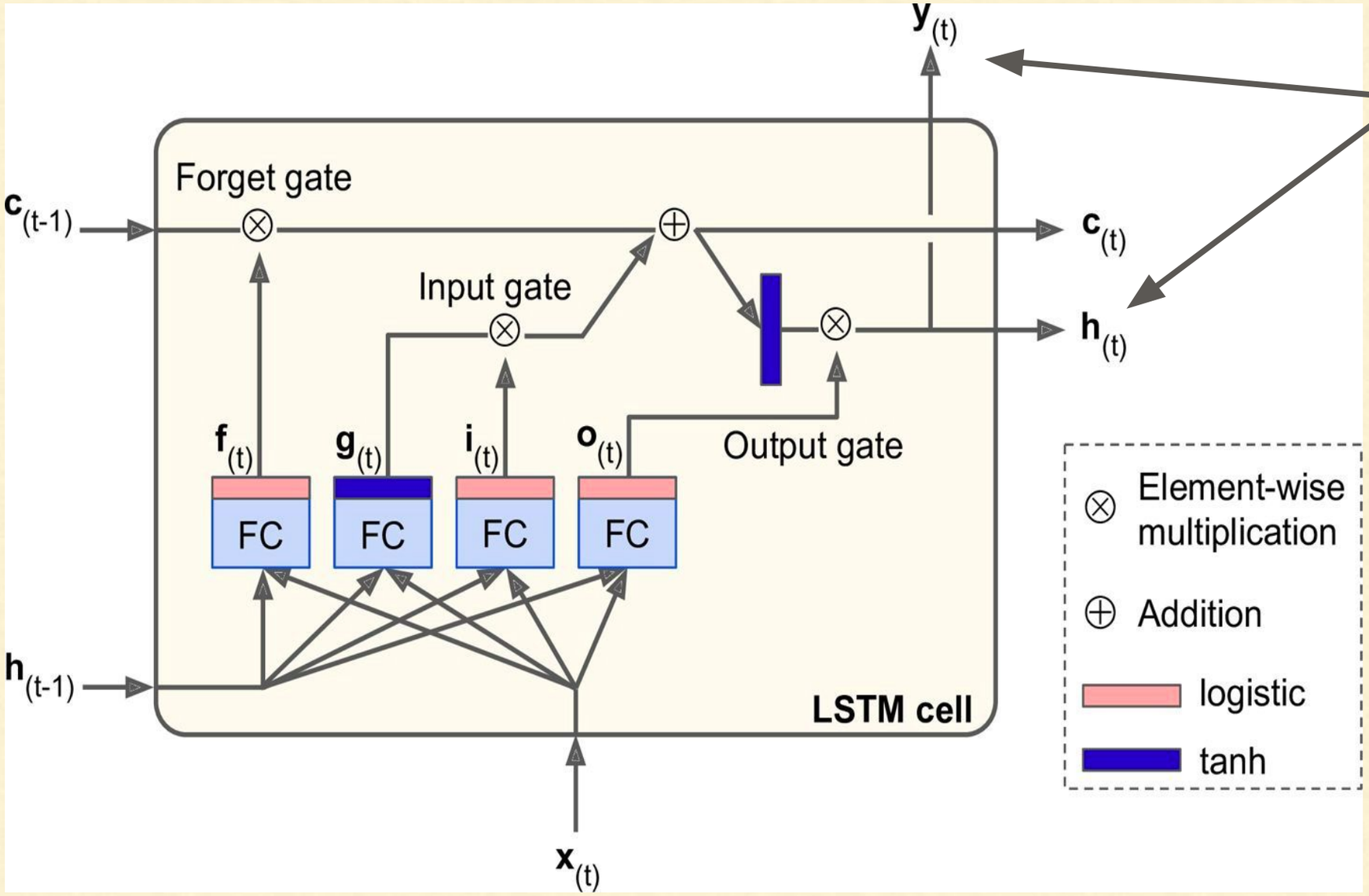
Understanding the LSTM cell structure



Moreover, after the addition operation, the long term state is copied and passed through the **tanh** function, and then the result is filtered by the output gate.

LSTM Cell

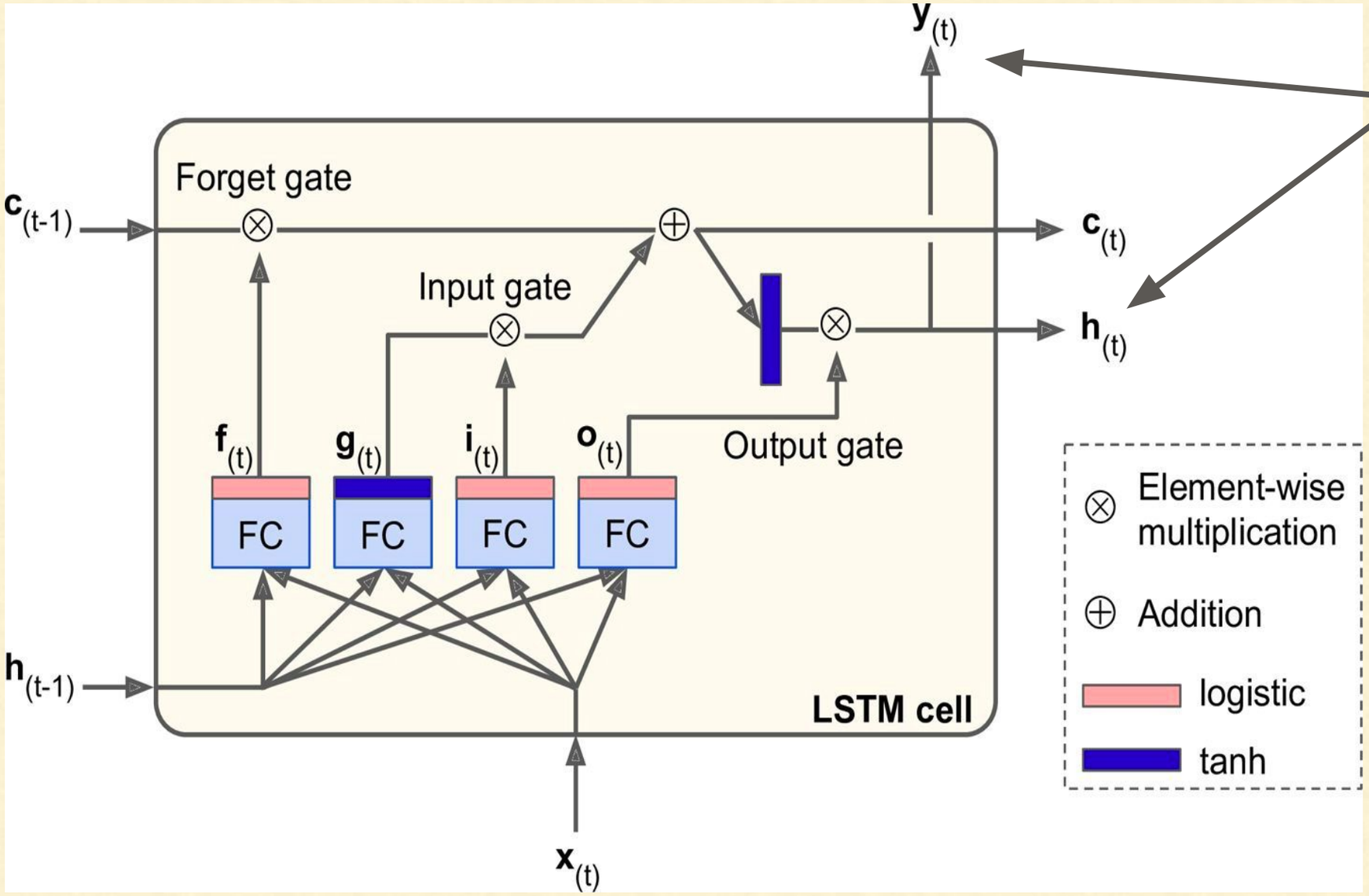
Understanding the LSTM cell structure



This produces the short-term state $\mathbf{h}_{(t)}$, which is equal to the cell's output for this time step y(t)

LSTM Cell

Understanding the LSTM cell structure



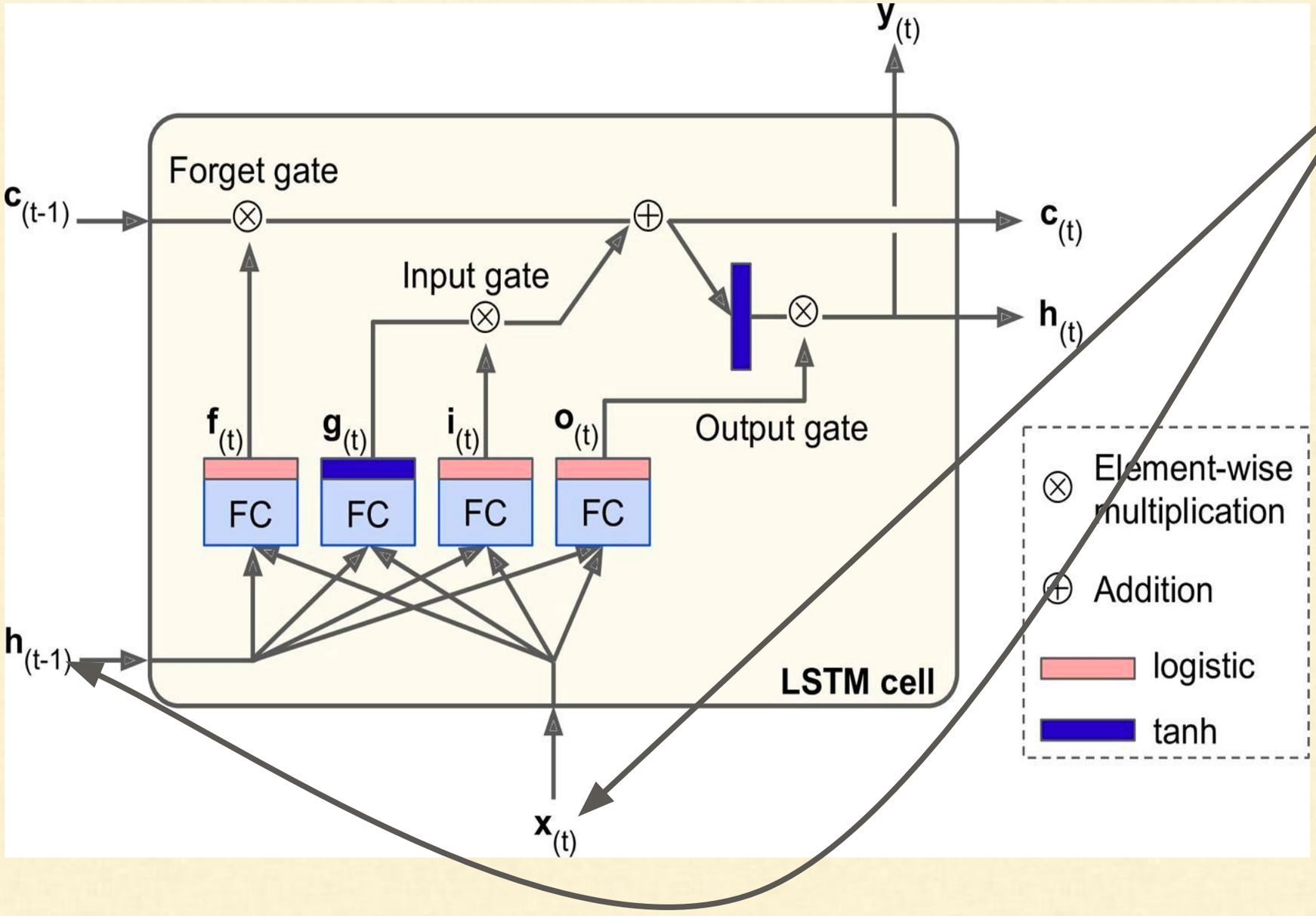
This produces the short-term state $\mathbf{h}_{(t)}$, which is equal to the cell's output for this time step y(t)

LSTM Cell

Now let's look at where new memories come from and how the gates work

LSTM Cell

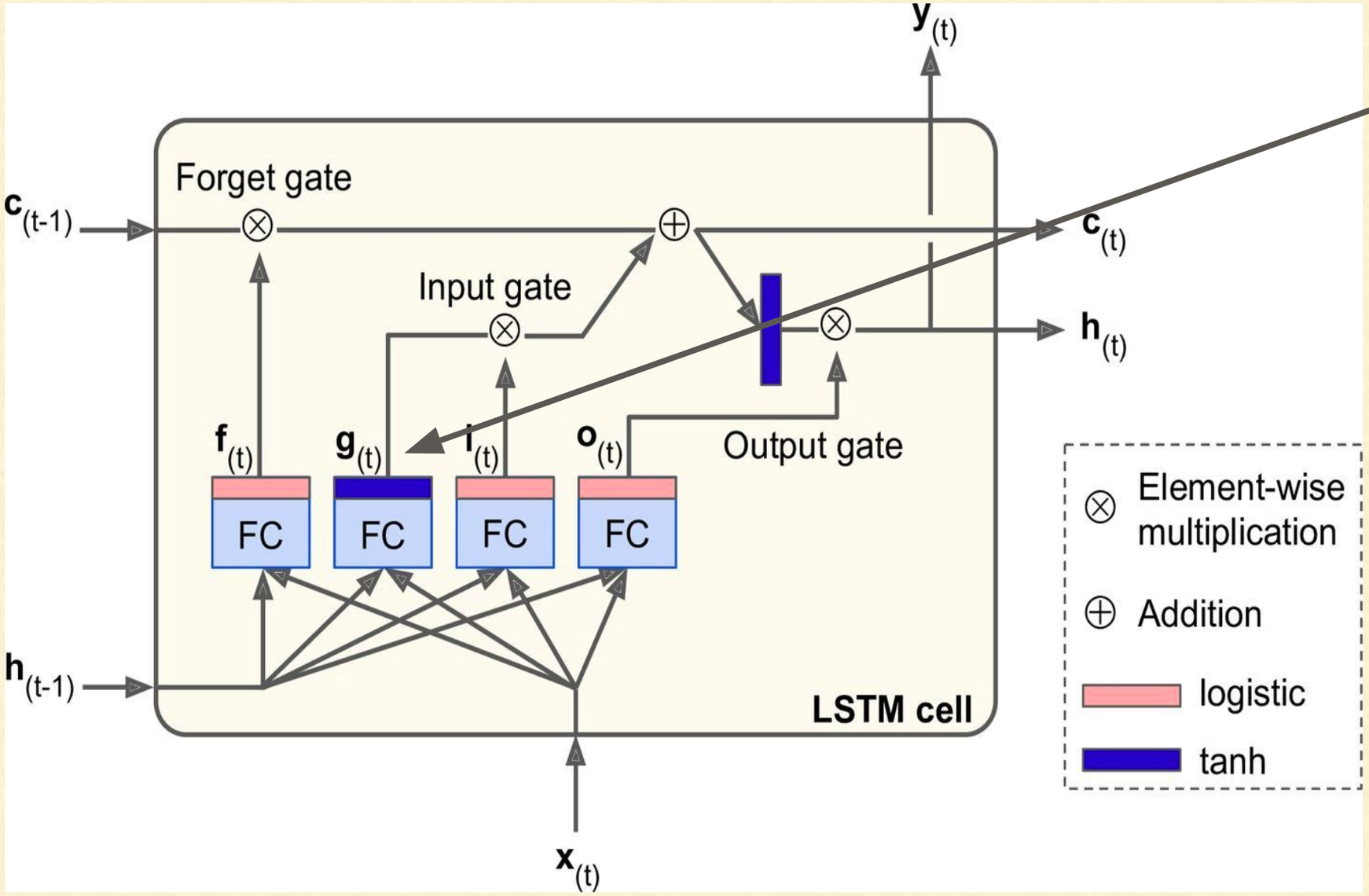
Understanding the LSTM cell structure



First, the current input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$ are fed to four different fully connected layers. They all serve a different purpose

LSTM Cell

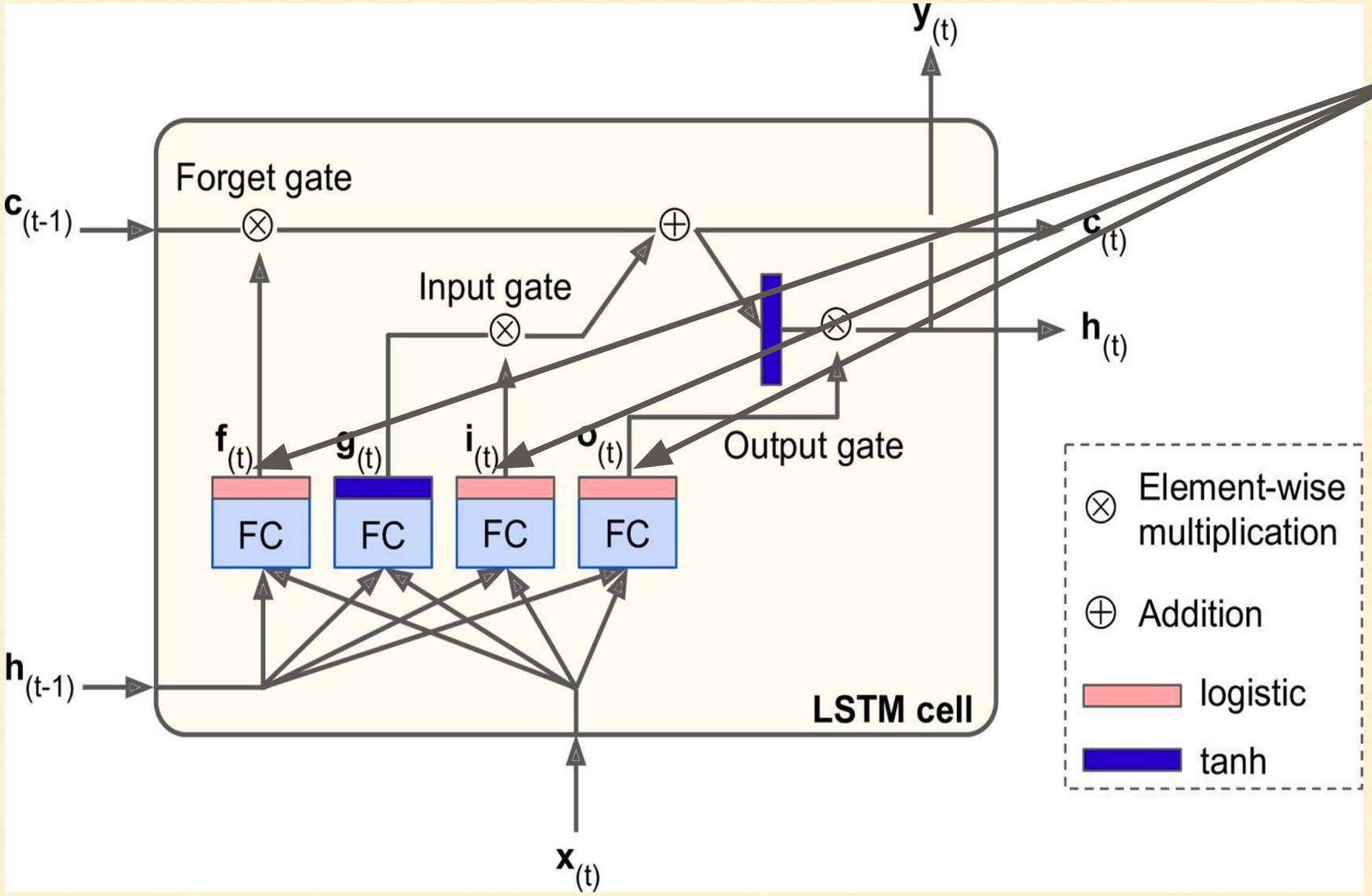
Understanding the LSTM cell structure



The main layer is the one that outputs $\mathbf{g}_{(t)}$. It has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$. In an LSTM cell this layer's output is partially stored in the long-term state.

LSTM Cell

Understanding the LSTM cell structure



The three other layers are gate controllers. Since they use the logistic activation function, their outputs range from 0 to 1.

LSTM Cell

Understanding the LSTM cell structure

$$\mathbf{i}_{(t)} = \sigma(\mathbf{W}_{xi}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i)$$

$$\mathbf{f}_{(t)} = \sigma(\mathbf{W}_{xf}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f)$$

$$\mathbf{o}_{(t)} = \sigma(\mathbf{W}_{xo}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o)$$

$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g)$$

$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$

$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})$$

- This summarizes how to compute the cell's **long-term state**, its **short-term state**, and its **output** at each time step for a single instance
- The equations for a whole mini-batch are very similar

LSTM Cell

Conclusion

- A **LSTM cell can learn** to
  - Recognize an important input, that's the role of the input gate,
  - Store it in the long-term state,
  - Learn to preserve it for as long as it is needed, that's the role of the forget gate,
  - And learn to extract it whenever it is needed

This explains why they have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

Peephole Connections

- In a **basic LSTM cell**, the gate controllers can look only at the input $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$
- It may be a good idea to give them a bit more context by letting them peek at the **long-term state as well**
- This idea was proposed by **Felix Gers and Jürgen Schmidhuber** in 2000

Peephole Connections

- They proposed an **LSTM variant** with extra connections called **peephole connections**:
  - The previous long-term state $\mathbf{c}_{(t-1)}$ is added as an input to the controllers of the forget gate and the input gate,
  - And the current long-term state $\mathbf{c}_{(t)}$ is added as input to the controller of the output gate.

Peephole Connections

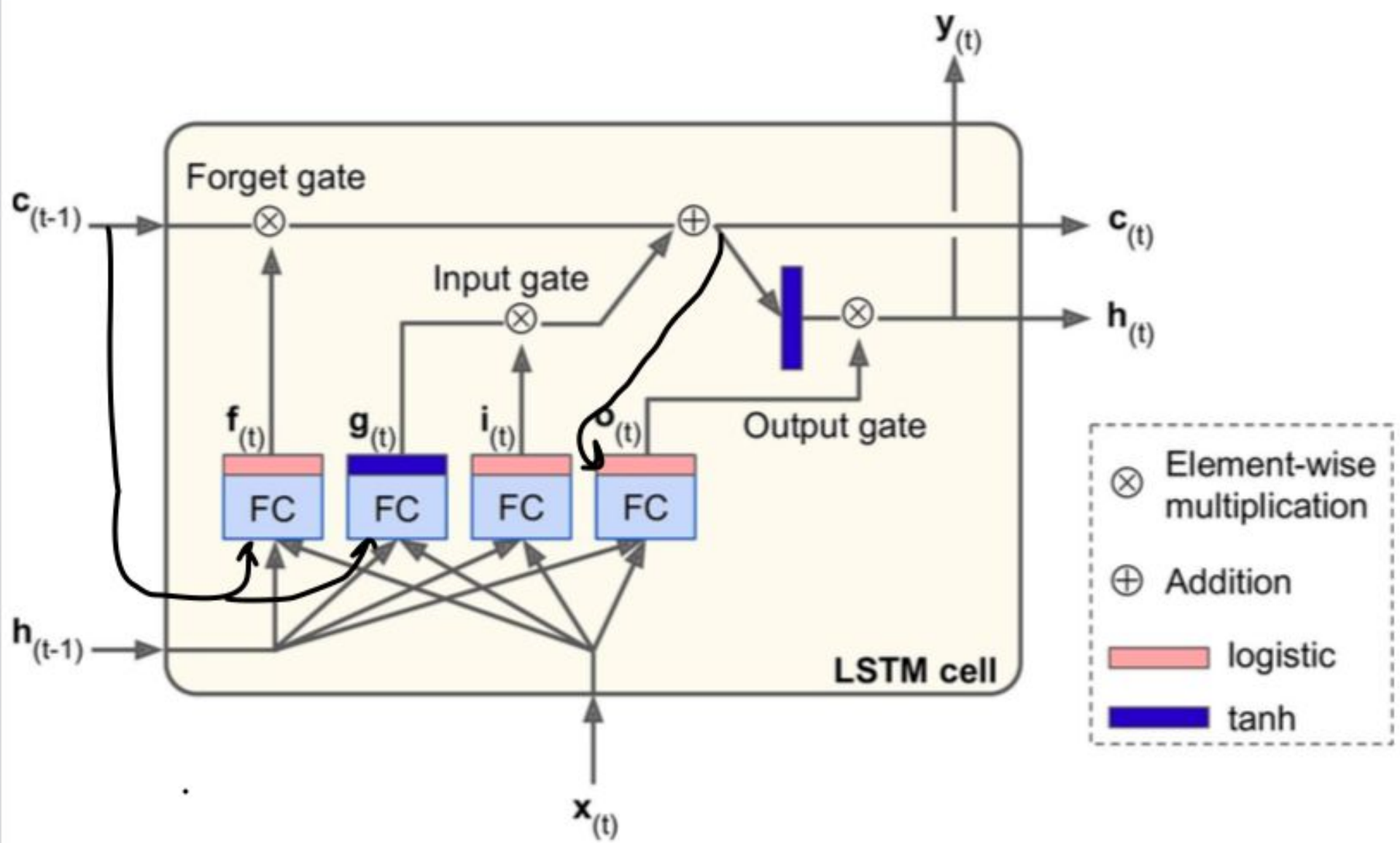


Figure 14-13. LSTM cell

Peephole Connections

To implement **peephole connections in TensorFlow**, you must use the **LSTMCell** instead of the **BasicLSTMCell** and set **use_peepholes=True**:

```
>>> lstm_cell = tf.contrib.rnn.LSTMCell(num_units=n_neurons,
use_peepholes=True)
```

There are many other variants of the **LSTM cell**.

One particularly popular variant is the **GRU cell**, which we will look at now.

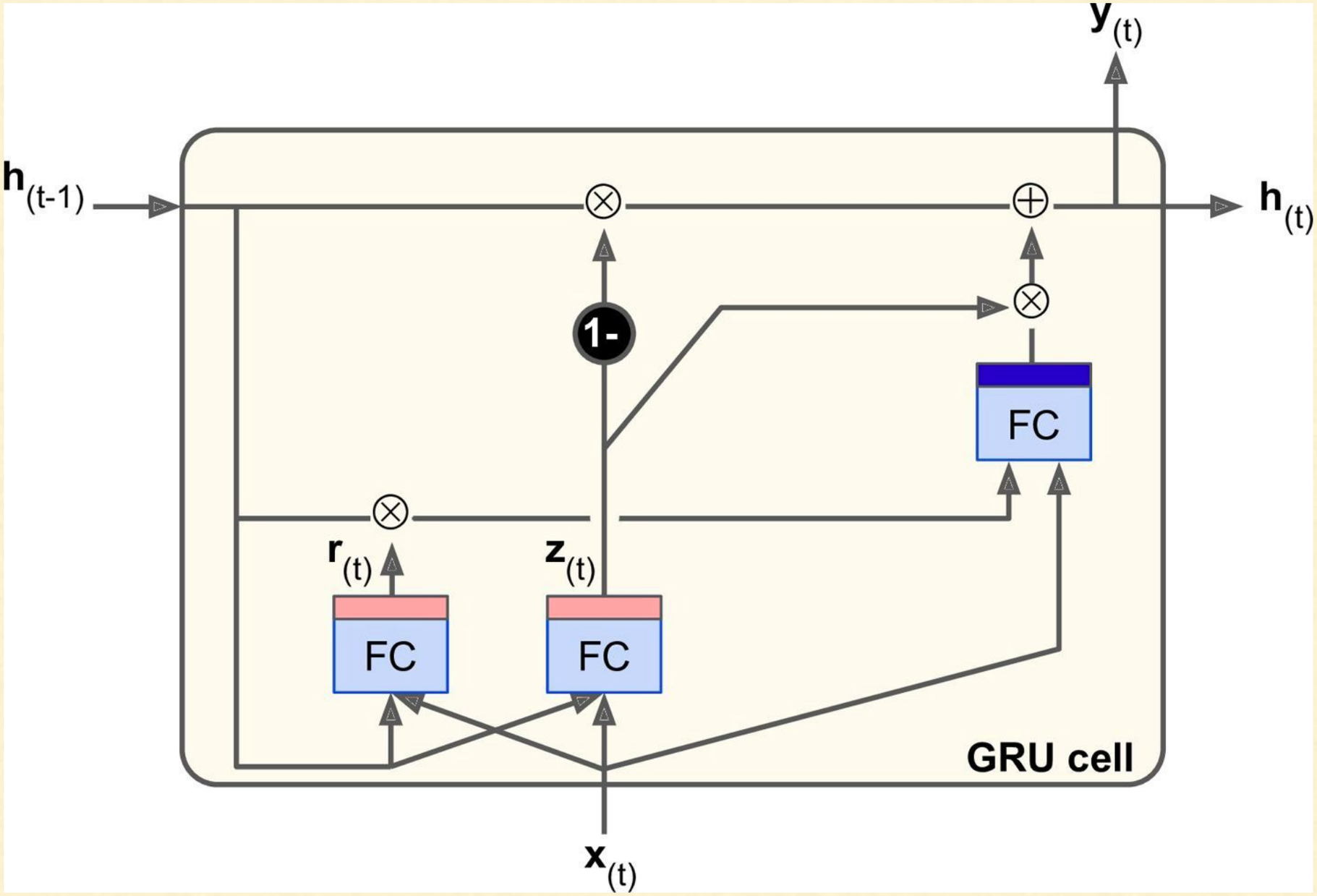
GRU Cell

GRU Cell

The Gated Recurrent Unit (GRU) cell was proposed by **Kyunghyun Cho** et al. in a 2014 paper that also introduced the Encoder–Decoder network we discussed earlier

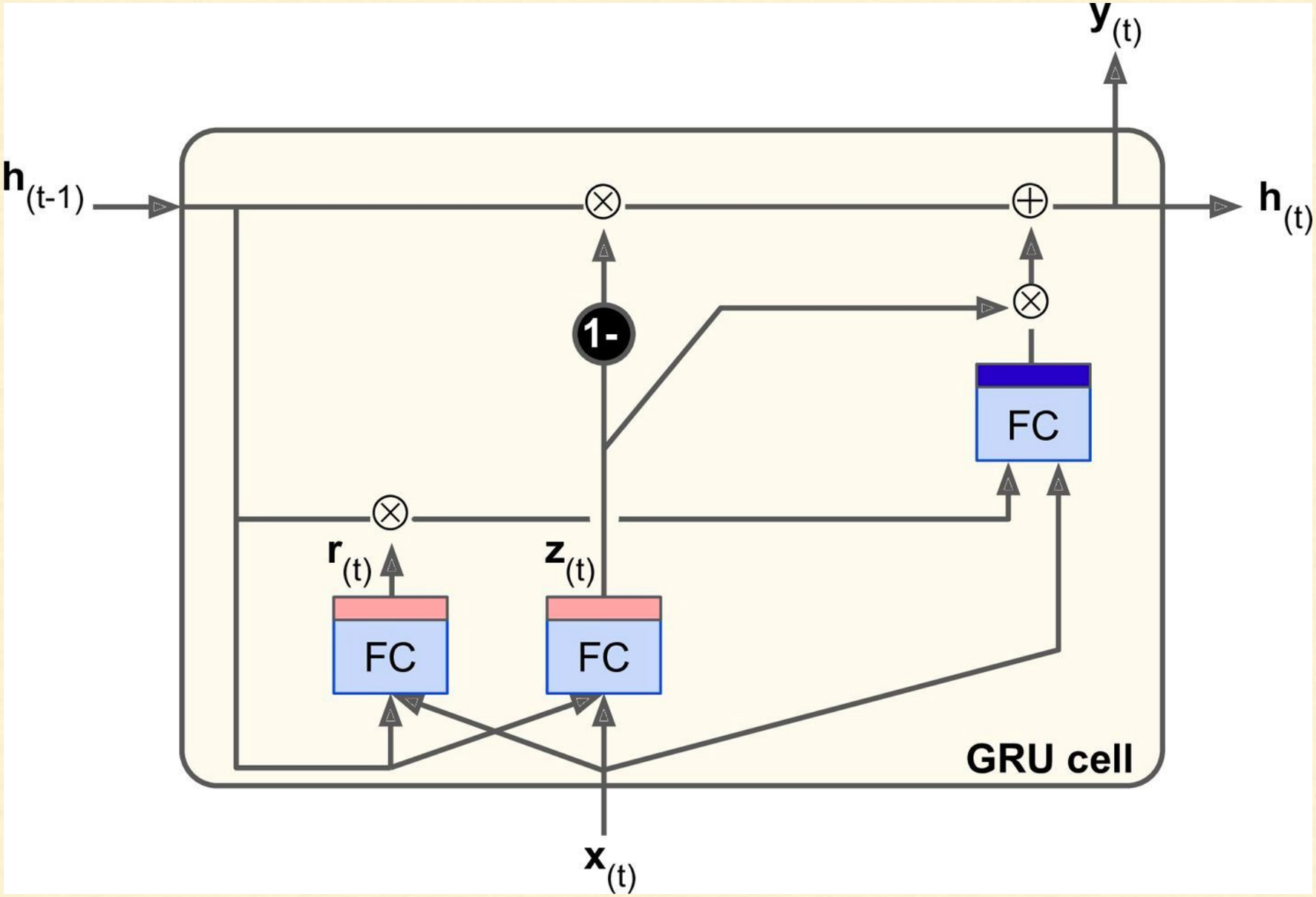
Kyunghyun Cho

GRU Cell



- The **GRU cell** is a simplified version of the **LSTM cell**
- It seems to perform just as well
- This explains its growing popularity

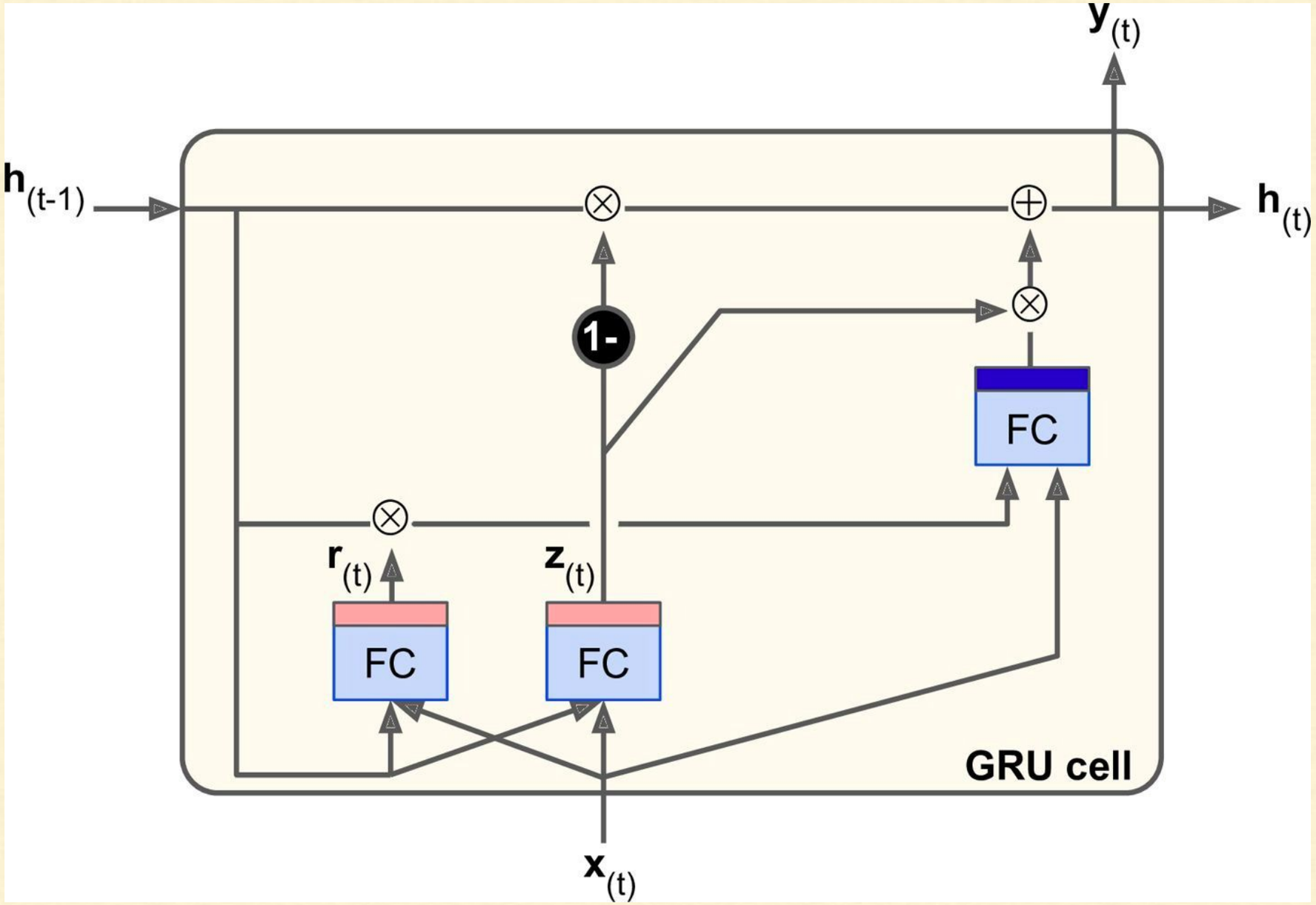
GRU Cell



The main simplifications are:

- Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$

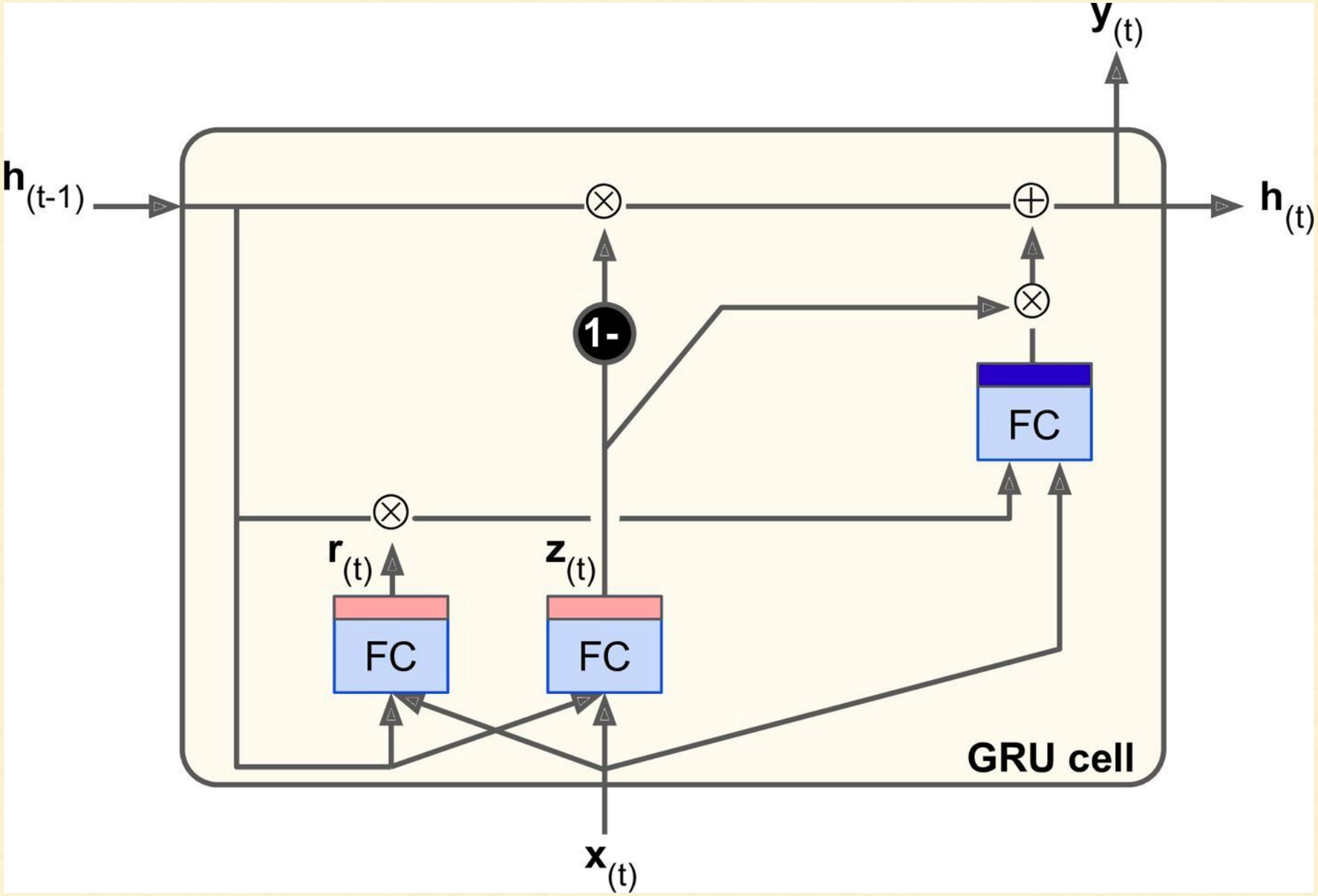
GRU Cell



The main simplifications are:

- A single gate controller controls both the forget gate and the input gate.
  If the gate controller outputs a 1, the input gate is open and the forget gate is closed.

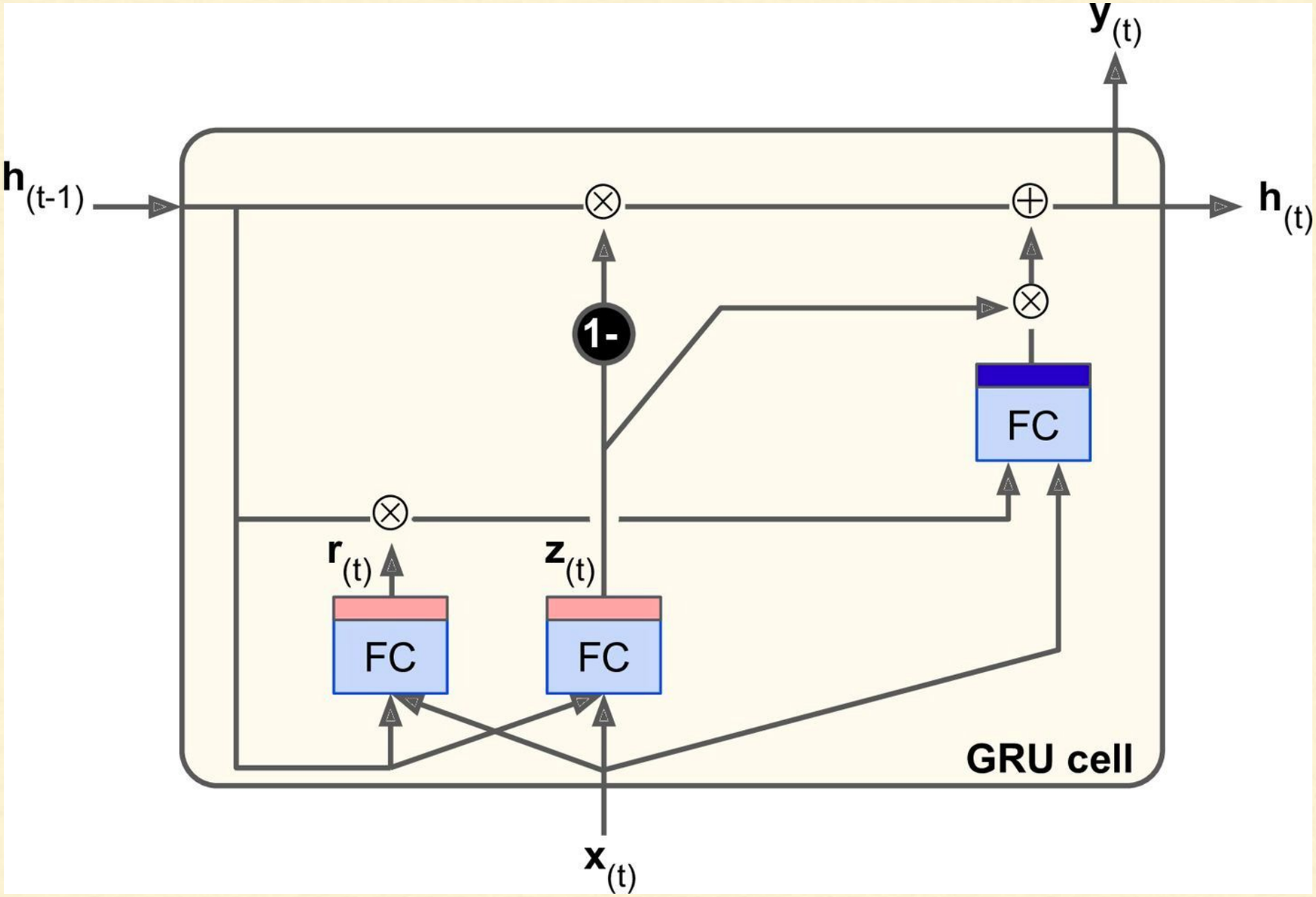
GRU Cell



The main simplifications are:

If it outputs a 0, the opposite happens
In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself

GRU Cell



The main simplifications are:

- There is no output gate; the full state vector is output at every time step. There is a new gate controller that controls which part of the previous state will be shown to the main layer.

GRU Cell

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}^{T} \cdot \mathbf{h}_{(t-1)}\right)$$

$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}^{T} \cdot \mathbf{h}_{(t-1)}\right)$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^{T} \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)})\right)$$

$$\mathbf{h}_{(t)} = (1 - \mathbf{z}_{(t)}) \otimes \tanh\left(\mathbf{W}_{xg}^{T} \cdot \mathbf{h}_{(t-1)}\right) + \mathbf{z}_{(t)} \otimes \mathbf{g}$$

Equations to compute the cell's state at each time step for a single instance

GRU Cell

Implementing GRU cell in TensorFlow

```
>>> gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

- LSTM or GRU cells are one of the main reasons behind the success of **RNNs** in recent years
- In particular for applications in **natural language processing (NLP)**

Natural Language Processing

Natural Language Processing

- Most of the state-of-the-art NLP applications, such as
  - Machine translation,
  - Automatic summarization,
  - Parsing,
  - Sentiment analysis,
  - and more, are **now based on RNNs**

Now we will take a quick look at what a machine translation model looks like.

This topic is very well covered by **TensorFlow's** awesome Word2Vec and Seq2Seq tutorials, so you should definitely check them out

Natural Language Processing - Word Representation

Before we start, we need to answer this important question

How do we represent a “word” ??

Natural Language Processing - Word Representation

In order to apply algorithms,

We need to convert everything in numbers.

What can we do about climate?

temp	climate	comments
12	Cold	Very nice place to visit in summers
30	Hot	Do not visit. This is a trap

Natural Language Processing - Word Representation

In order to apply algorithms,

We need to convert everything in numbers.

What can we do about climate?

We can convert it into One-Hot vector

temp	climate	comments
12	Cold	Very nice place to visit in summers
30	Hot	Do not visit. This is a trap

temp	climate_cold	climate_hot	comments
12	1	0	Very nice place to visit in summers
30	0	1	Do not visit. This is a trap

Natural Language Processing - Word Representation

In order to apply algorithms,

We need to convert everything in numbers.

And what can we do about comments?

temp	climate	comments
12	Cold	Very nice place to visit in summers
30	Hot	Do not visit. This is a trap

Natural Language Processing - Word Representation

One option could be to represent each word using a one-hot vector.

But consider this :

- Suppose your vocabulary contains **50,000 words**
- Then the **nth word** would be represented as a **50,000-dimensional vector**, full of **0s** except for a **1** at the **nth** position
- However, with such a large vocabulary, this sparse representation would **not be efficient at all**

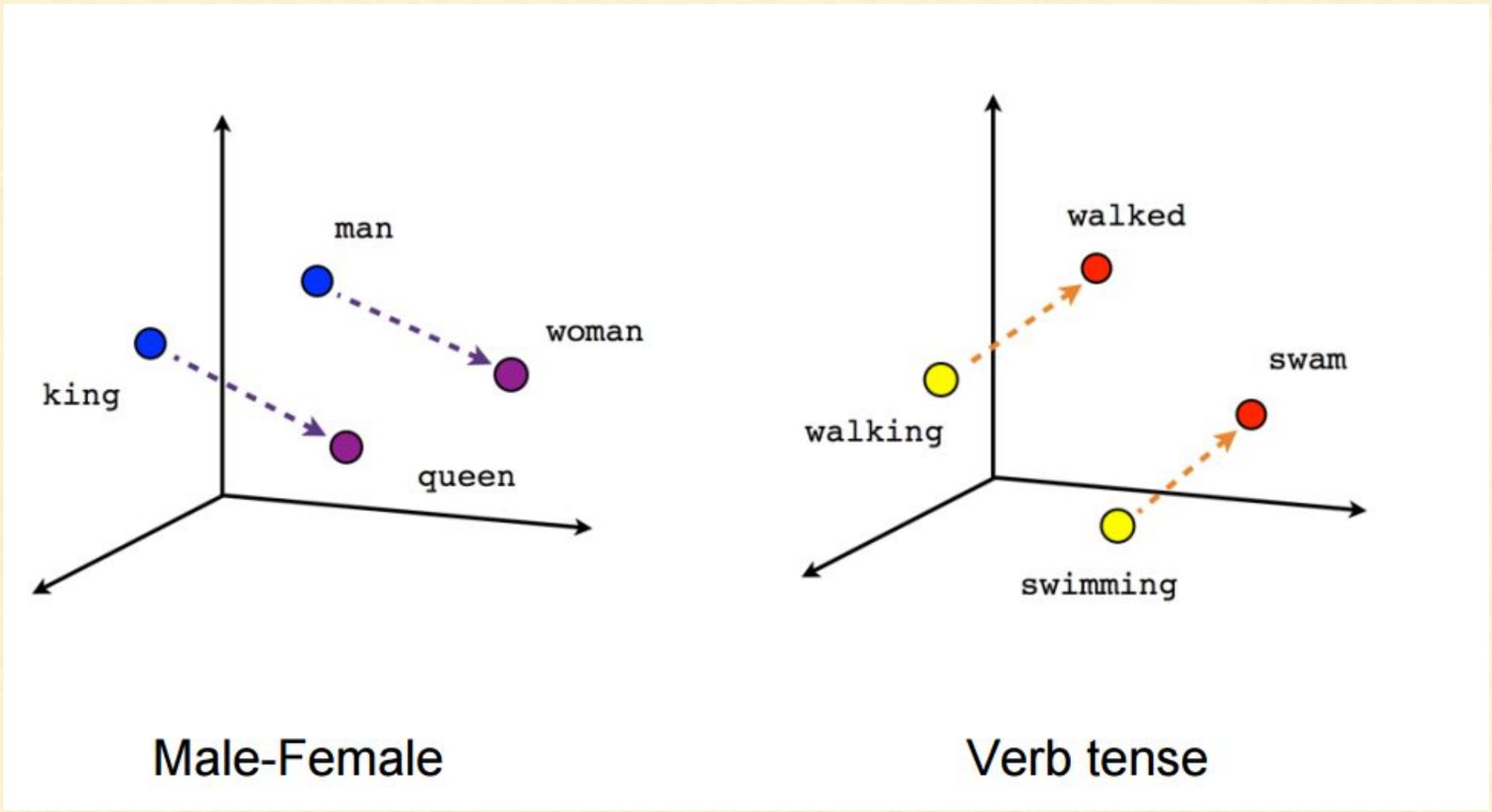
Natural Language Processing - Word Representation

- Ideally, we want **similar words** to have **similar representations**, making it easy for the model to generalize what it learns about a word to all similar words
- For example,
  - If the model is told that **“I drink milk”** is a valid sentence, and if it knows that **“milk”** is close to **“water”** but far from **“shoes”**
  - Then it will know that **“I drink water”** is probably a valid sentence as well
  - While **“I drink shoes”** is probably not

But how can you come up with such a meaningful representation?

Natural Language Processing - Word Embedding

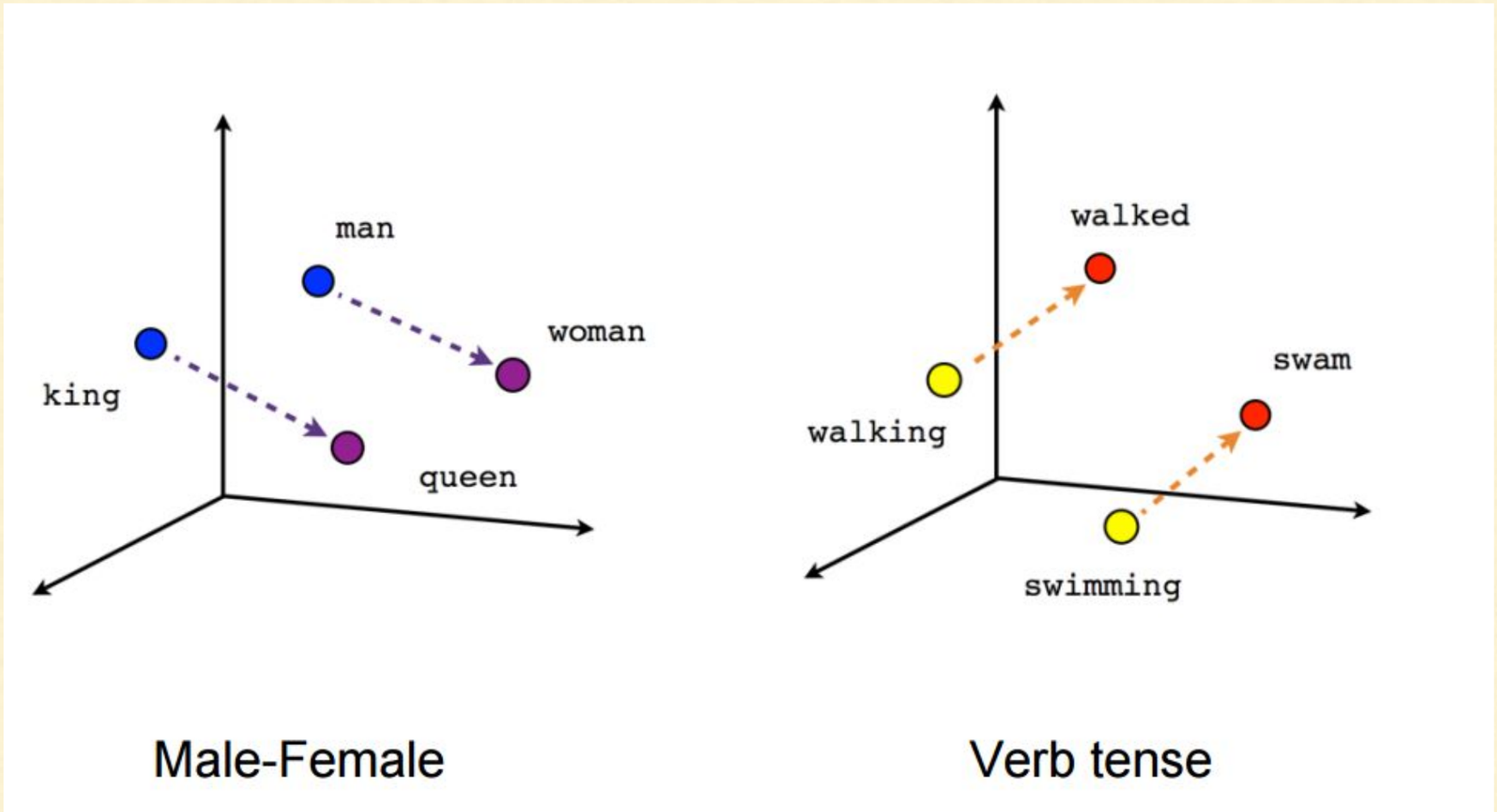
- The most common solution is to represent each word in the vocabulary using a fairly small and dense vector e.g., 150 dimensions, called an **Embedding**
- And just let the neural network learn a good embedding for each word during training



Natural Language Processing - Word Embedding

With word embeding a lot of magic is possible:

king - man + woman == queen

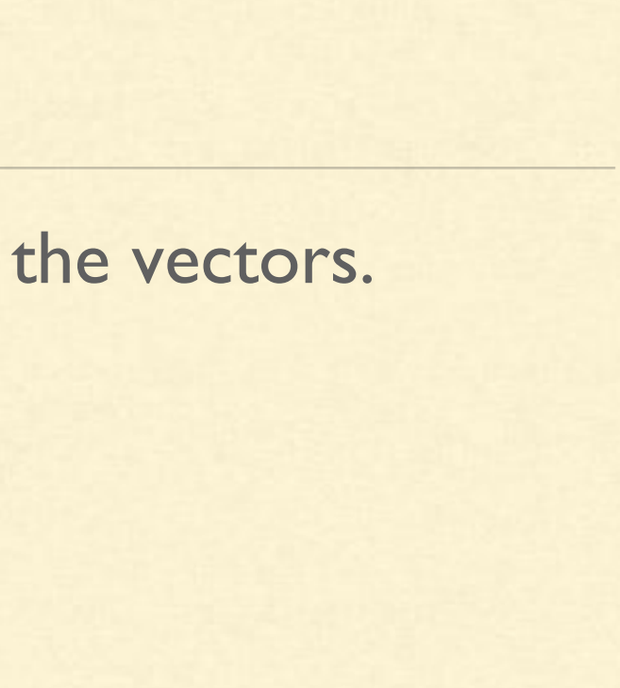


Word Embedding - word2vec

- Based on the context of word, people have generated the vectors.
- One such vector is word2vec and other is Glove

```
from gensim.models import KeyedVectors
# load the google word2vec model
filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'],
   negative=['man'], topn=1)
print(result)
```

```
[('queen', 0.7118192315101624)]
```



Word Embedding - Vector space models (VSMs)

Based on the Distributional Hypothesis:

- words that appear in the same contexts share semantic meaning.

Two Approaches:

1. Count-based methods (e.g. Latent Semantic Analysis)
2. Predictive methods (e.g. neural probabilistic language models)

Word Embedding - word2vec - Approaches

1. **Count-based methods (e.g. Latent Semantic Analysis)**
    - Compute the statistics of how often some word co-occurs with its neighbor words in a large text corpus
    - Map these count-statistics down to a small, dense vector for each word

Word Embedding - word2vec - Approaches

2. Predictive models

- Directly try to predict a word from its neighbors
- in terms of learned small, dense embedding vectors
- (considered parameters of the model).

word2vec

Computationally-efficient predictive model

for learning word embeddings from raw text.

Comes in two flavors:

1. *Continuous Bag-of-Words model (CBOW)*
2. *Skip-Gram model*

word2vec

Computationally-efficient predictive model

for learning word embeddings from raw text.

Comes in two flavors:

1. *Continuous Bag-of-Words model (CBOW)*
    - *predicts target words (e.g. 'mat') from source context words*
    - *e.g ('the cat sits on the'),*
2. *Skip-Gram model*

word2vec

Computationally-efficient predictive model

for learning word embeddings from raw text.

Comes in two flavors:

1. Continuous Bag-of-Words model (CBOW)
    - predicts target words (e.g. 'mat') from source context words
    - e.g ('the cat sits on the'),
2. Skip-Gram model
    - Predicts source context-words from the target words
    - Treats each context-target pair as a new observation
    - Tends to do better when we have larger datasets.
    - Will focus on this

word2vec: Scaling up Noise-Contrastive Training

Neural probabilistic language models

- *are traditionally trained using the maximum likelihood (ML) principle*
- *to maximize the probability of the next word w_t (for "target")*
- *given the previous words h (for "history") in terms of a softmax function,*

word2vec: Scaling up Noise-Contrastive Training

Neural probabilistic language models

- *are traditionally trained using the maximum likelihood (ML) principle*
- *to maximize the probability of the next word w_t (for "target")*
- *given the previous words h (for "history") in terms of a softmax function,*

$$P(w_t|h) = \text{softmax}(\text{score}(w_t, h)) = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word w' in Vocab}} \exp\{\text{score}(w', h)\}}$$

where score(w_t, h) computes the compatibility of word w_t with the context h(a dot product is commonly used). We train this model by maximizing its log-likelihood i.e. $\log P(w_t|h)$

word2vec: Scaling up Noise-Contrastive Training

Neural probabilistic language models

- *are traditionally trained using the maximum likelihood (ML) principle*
- *to maximize the probability of the next word w_t (for "target")*
- *given the previous words h (for "history") in terms of a softmax function,*

$$P(w_t|h) = \text{softmax}(\text{score}(w_t, h)) = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word w' in Vocab}} \exp\{\text{score}(w', h)\}}$$

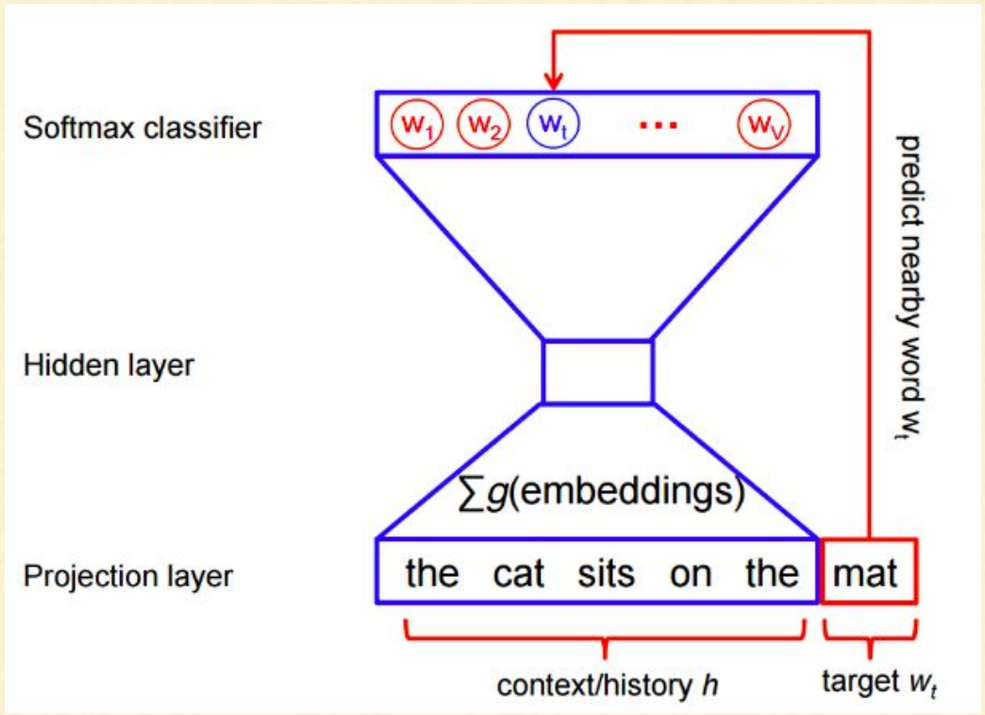
where score(w_t, h) computes the compatibility of word w_t with the context h(a dot product is commonly used). We train this model by maximizing its log-likelihood i.e. $\log P(w_t|h)$

This is very expensive, because we need to compute and normalize each probability using the score for all other V words w' in the current context , at every training step.

word2vec: Scaling up Noise-Contrastive Training

Neural probabilistic language models

- are traditionally trained using the maximum likelihood (ML) principle
- to maximize the probability of the next word w_t (for "target")
- given the previous words h (for "history") in terms of a softmax function,

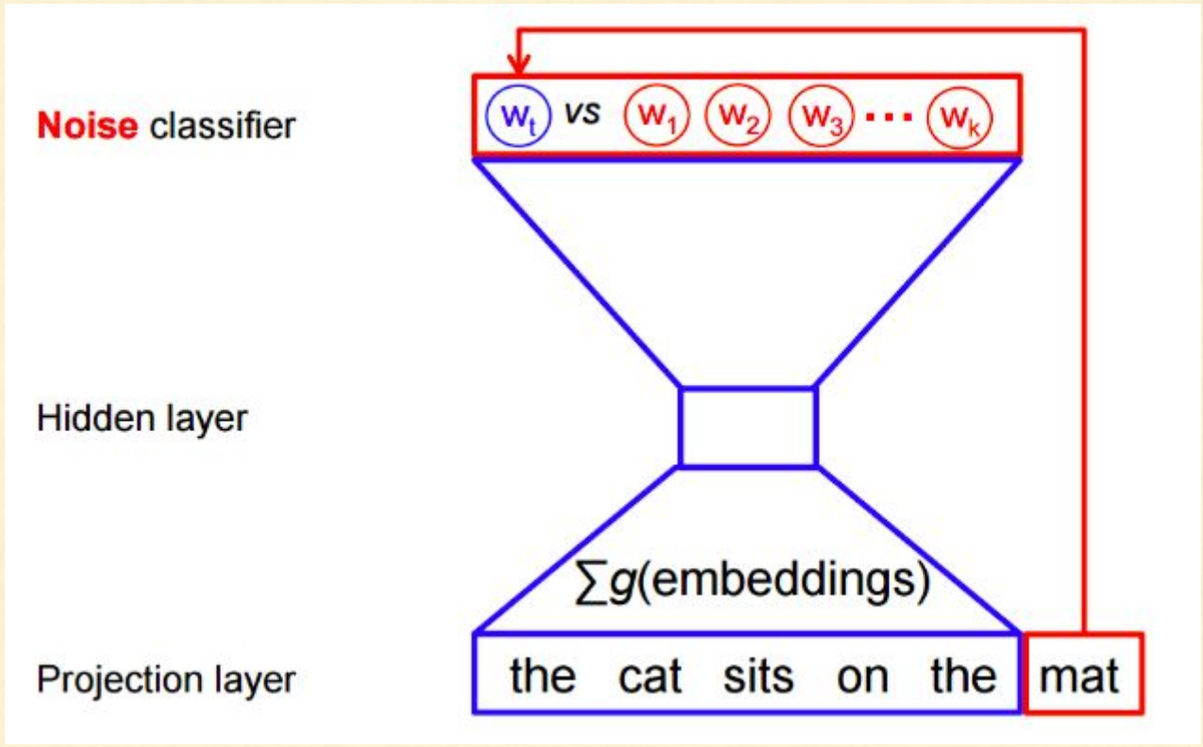


This is very expensive, because we need to compute and normalize each probability using the score for all other V words w' in the current context , at every training step.

word2vec: Scaling up Noise-Contrastive Training

Instead models trained using a binary classification objective (logistic regression)

to discriminate the real target words w_t from k imaginary (noise) words w, in the same context.



1. Computing the loss function now scales only with the number of noise words that we select and not all words in the vocabulary
2. This makes it much faster to train.
3. will use similar noise-contrastive estimation (NCE) loss - tf.nn.nce_loss().

Word2vec: Context Example

the quick brown fox jumped over the lazy dog

Context: word to the left and word to the right.

([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...

Word2vec: Skip Gram Model

Skip-gram

- *inverts contexts and targets, and*
- *tries to predict each context word from its target word*

the quick brown fox jumped over the lazy dog

Task becomes to predict 'the' and 'brown' from 'quick', 'quick' and 'fox' from 'brown', etc.

(quick, the), (quick, brown), (brown, quick), (brown, fox), ...

Natural Language Processing - Word Embedding

Let's imagine at training step t

- For first case above, the goal is to predict ***the*** from ***quick***.
- We select num_noise number
  - of noisy (contrastive) examples
  - by drawing from some noise distribution,
  - typically the unigram distribution,
- For simplicity let's say num_noise=1 and we select sheep as a noisy example. Next we compute the loss for this pair of observed and noisy examples

Natural Language Processing - Word Embedding

The objective at time step t becomes:

$$J_{\text{NEG}}^{(t)} = \log Q_\theta(D = 1|\text{the, quick}) + \log(Q_\theta(D = 0|\text{sheep, quick}))$$

where $Q_\theta(D = 1|w, h)$ is the binary logistic regression probability under the model of seeing the word w in the context h in the dataset D, calculated in terms of the learned embedding vectors θ. In practice we approximate the expectation by drawing k contrastive words from the noise distribution (i.e. we compute a Monte Carlo average).

Natural Language Processing - Word Embedding

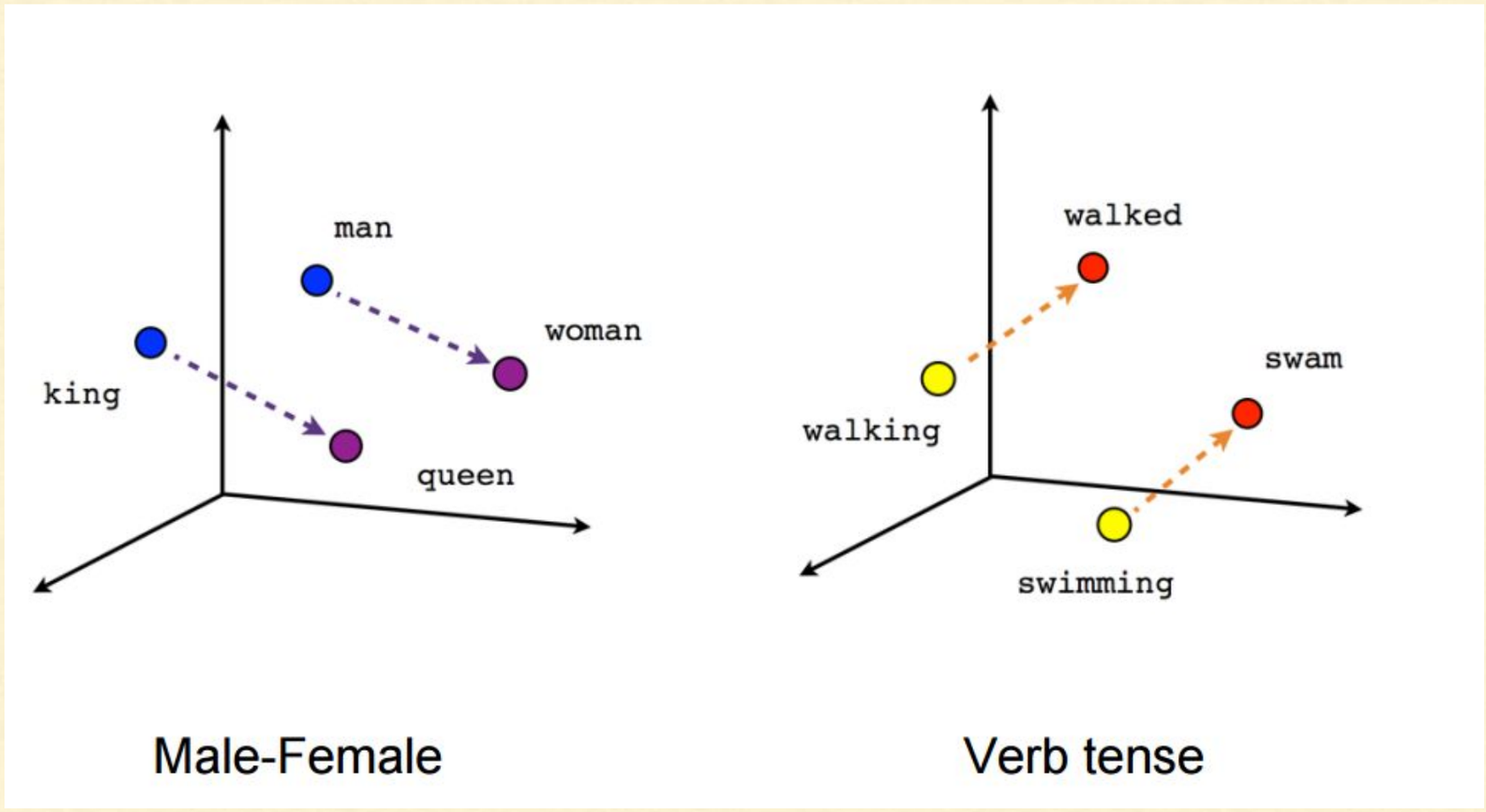
- The goal is to make an update to the embedding parameters
- to improve (in this case, maximize) the objective function
- We do this by deriving the gradient of the loss with respect to the embedding parameters , i.e. (luckily TensorFlow provides easy helper functions for doing this!).
- We then perform an update to the embeddings by taking a small step in the direction of the gradient. When this process is repeated over the entire training set, this has the effect of 'moving' the embedding vectors around for each word until the model is successful at discriminating real words from noise words.

Natural Language Processing - Word Embedding

- At the beginning of training, embeddings are simply chosen randomly,
- But during training, **backpropagation** automatically moves the embeddings around in a way that helps the neural network perform its task

Natural Language Processing - Word Embedding

- Typically this means that similar words will gradually cluster close to one another, and even end up organized in a rather meaningful way.
- For example, embeddings may end up placed along various axes that represent
  - gender,
  - singular/plural,
  - adjective/noun,
  - and so on



Male-Female

Verb tense

Natural Language Processing - Word Embedding

How to do it in TensorFlow

In TensorFlow, we first need to create the variable representing the

embeddings for every word in our vocabulary which is initialized randomly

```
>>> vocabulary_size = 50000
>>> embedding_size = 150
>>> embeddings = tf.Variable(
tf.random_uniform([vocabulary_size, embedding_size],
-1.0, 1.0))
```

Natural Language Processing - Word Embedding

How to do it in TensorFlow - Preprocessing

Suppose we want to feed the sentence **"I drink milk"** to your neural network.

- We should first preprocess the sentence and break it into **a list of known words**
- For example
  - We may remove unnecessary characters, replace unknown words by a predefined token word such as "[UNK]",
  - Replace numerical values by "[NUM]",
  - Replace URLs by "[URL]",
  - And so on

Natural Language Processing - Word Embedding

How to do it in TensorFlow

- Once we have a list of known words, we can look up each word's integer identifier from **0 to 49999** in a dictionary, for example [72, 3335, 288]
- At that point, you are ready to feed these word identifiers to TensorFlow using a placeholder, and apply the **embedding_lookup()** function to get the corresponding embeddings

```
>>> train_inputs = tf.placeholder(tf.int32, shape=[None]) # from ids...
>>> embed = tf.nn.embedding_lookup(embeddings, train_inputs) # ...to
embeddings
```

Natural Language Processing - Word Embedding

- Once our model has learned good word embeddings, it can actually be reused fairly efficiently in any **NLP application**
- In fact, instead of training your own word embeddings, we may want to download pre-trained word embeddings
- Just like when reusing pretrained layers, we can choose to
  - Freeze the pretrained embeddings
  - Or let backpropagation tweak them for your application
- The first option will speed up training, but the second may lead to slightly higher performance

Machine Translation

Machine Translation

We now have almost all the tools we need to implement a machine translation system

Let's look at this now

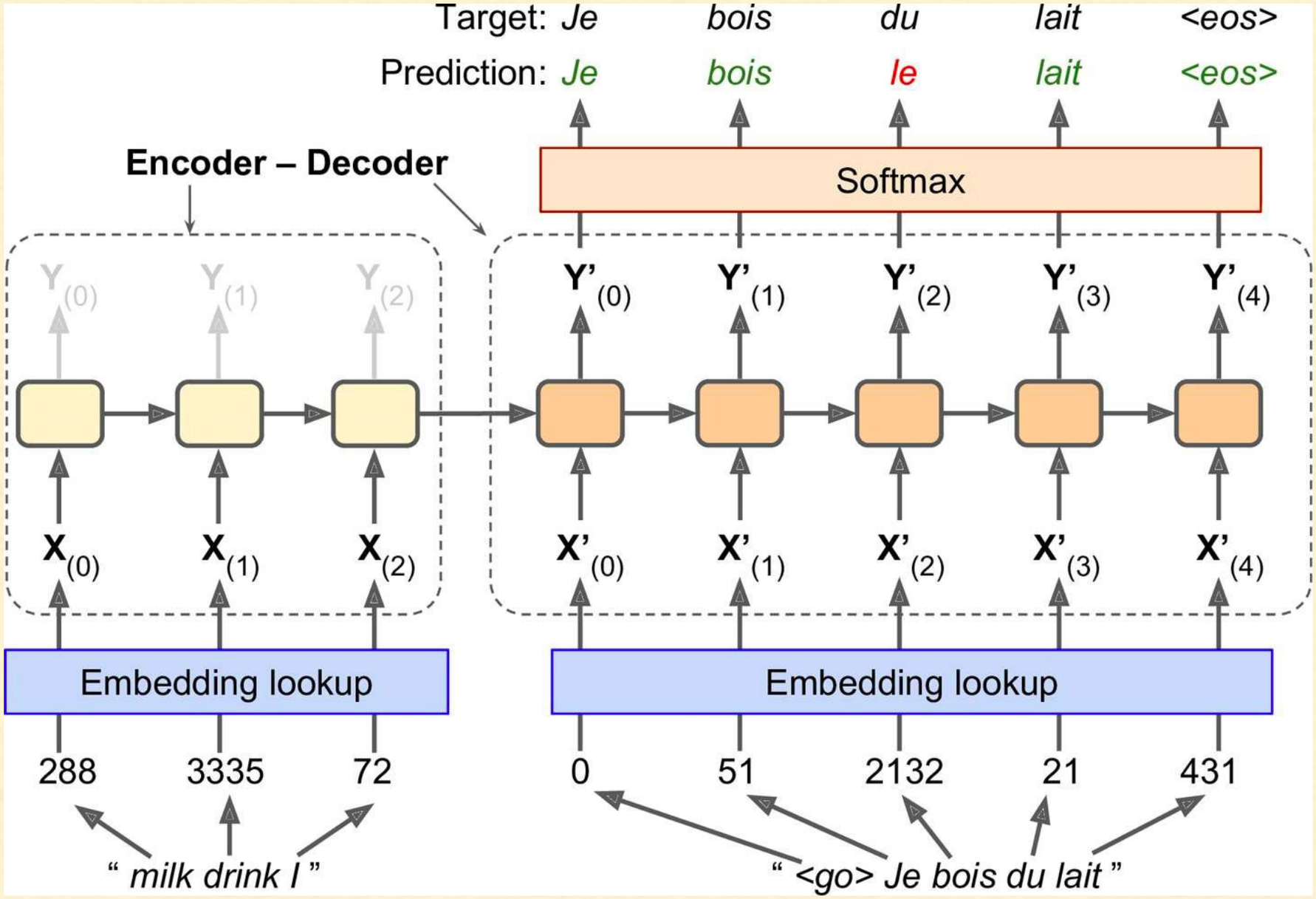
Machine Translation

An Encoder–Decoder Network for Machine Translation

Let's take a look at a simple machine translation model that will translate English sentences to French

Machine Translation

An Encoder–Decoder Network for Machine Translation



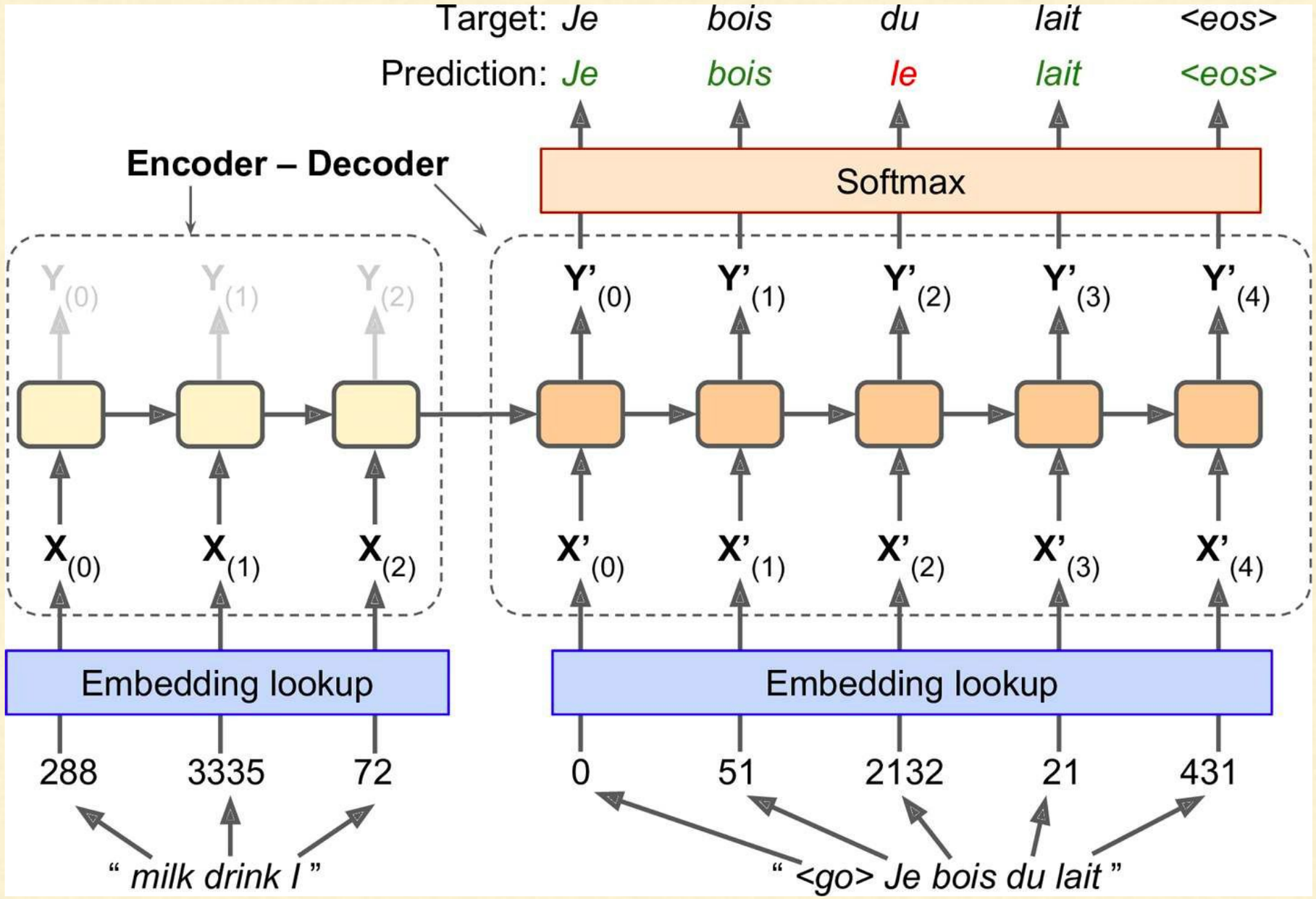
A simple machine translation model

Machine Translation

Let's learn how this Encoder–Decoder Network for Machine Translation is trained

Machine Translation

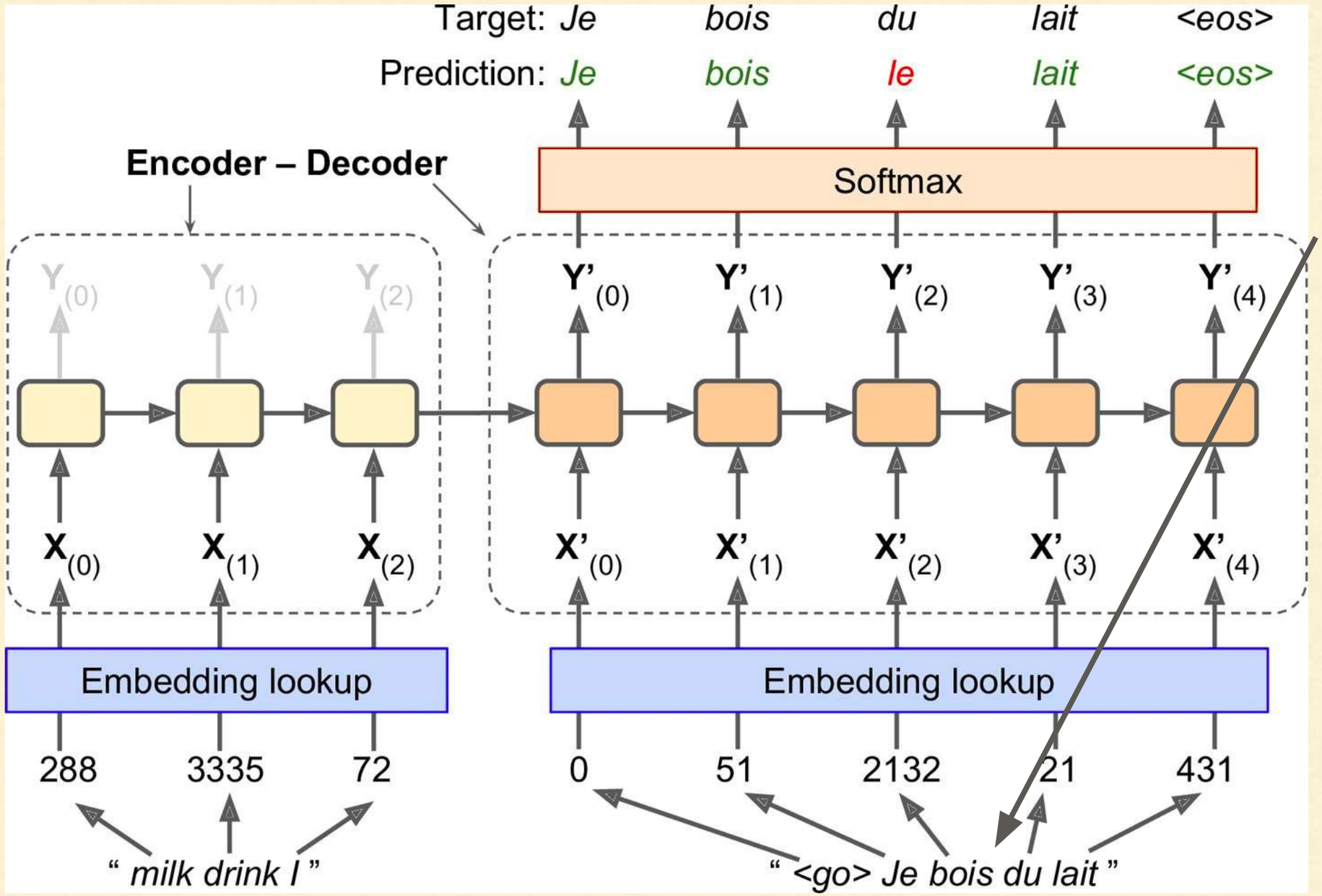
An Encoder–Decoder Network for Machine Translation



The English sentences are fed to the **encoder**, and the **decoder** outputs the French translations

Machine Translation

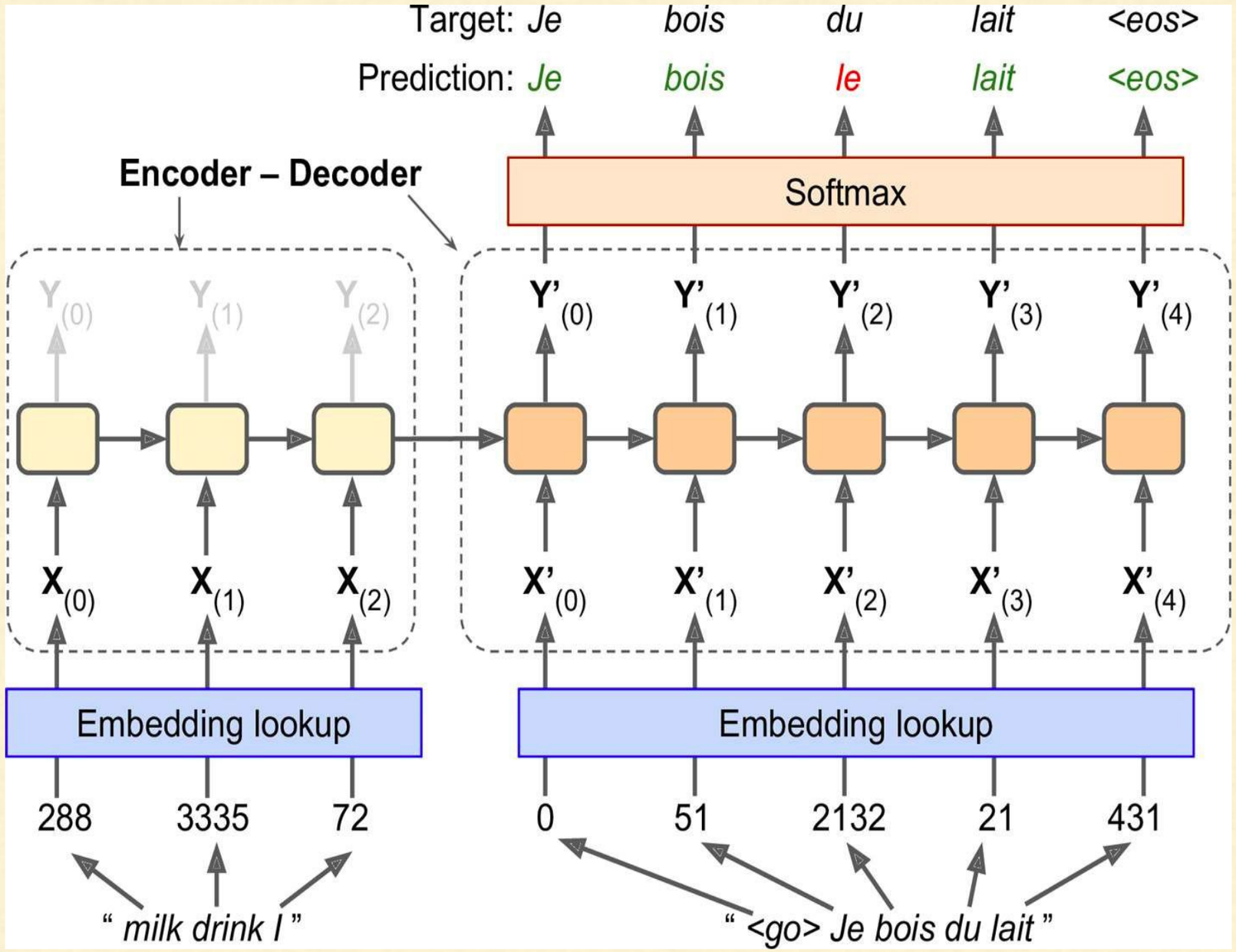
An Encoder–Decoder Network for Machine Translation



Note that the French translations are also used as inputs to the decoder, but pushed back by one step

Machine Translation

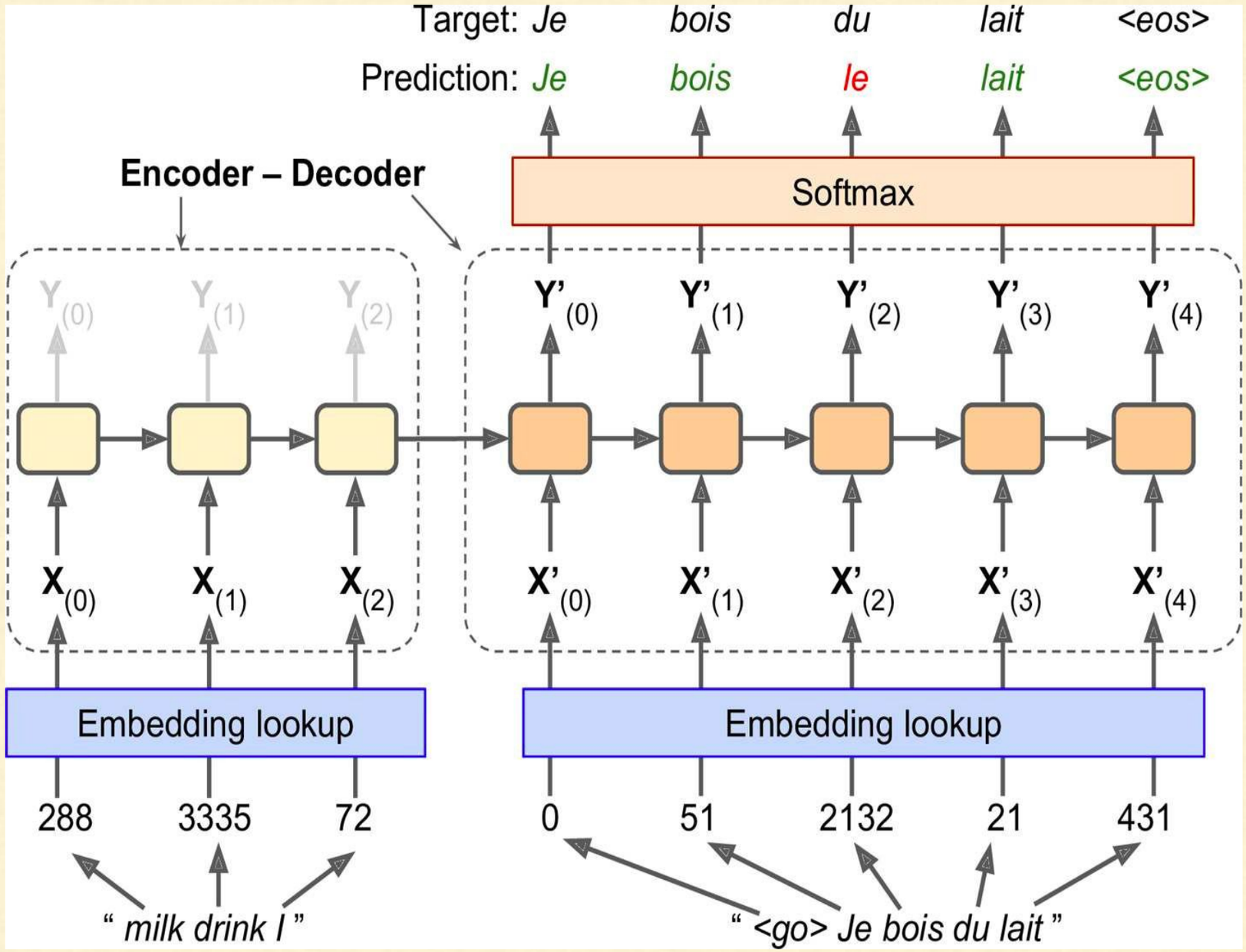
An Encoder–Decoder Network for Machine Translation



In other words, the decoder is given as input the word that **it should** have output **at the previous step.** Regardless of what it actually output at the current step

Machine Translation

An Encoder–Decoder Network for Machine Translation

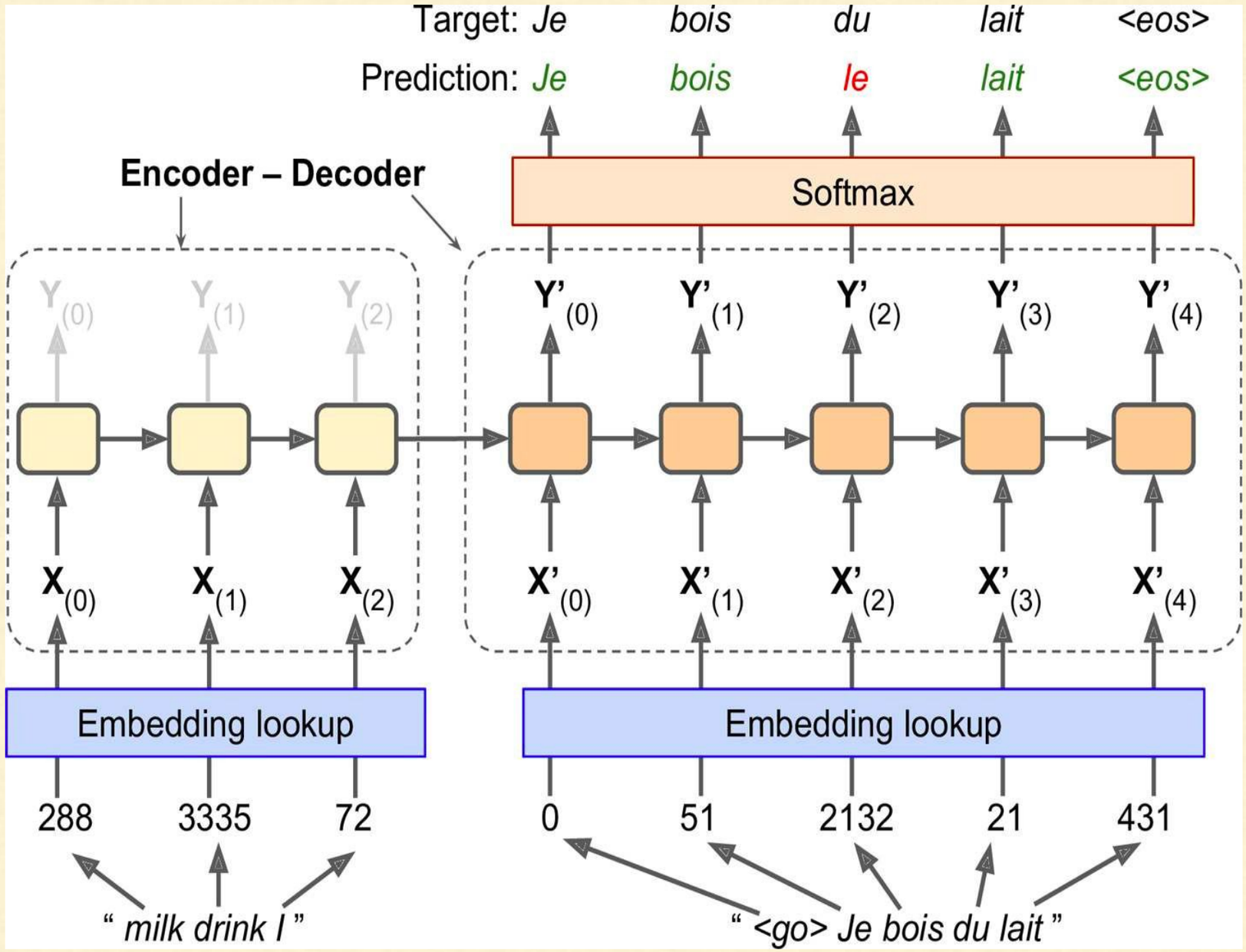


For the very first word, **the decoder** is given a token that represents the beginning of the sentence (here, **"<go>"**)

The decoder is expected to end the sentence with an end-of-sequence (EOS) token (here, **"<eos>"**)

Machine Translation

An Encoder–Decoder Network for Machine Translation



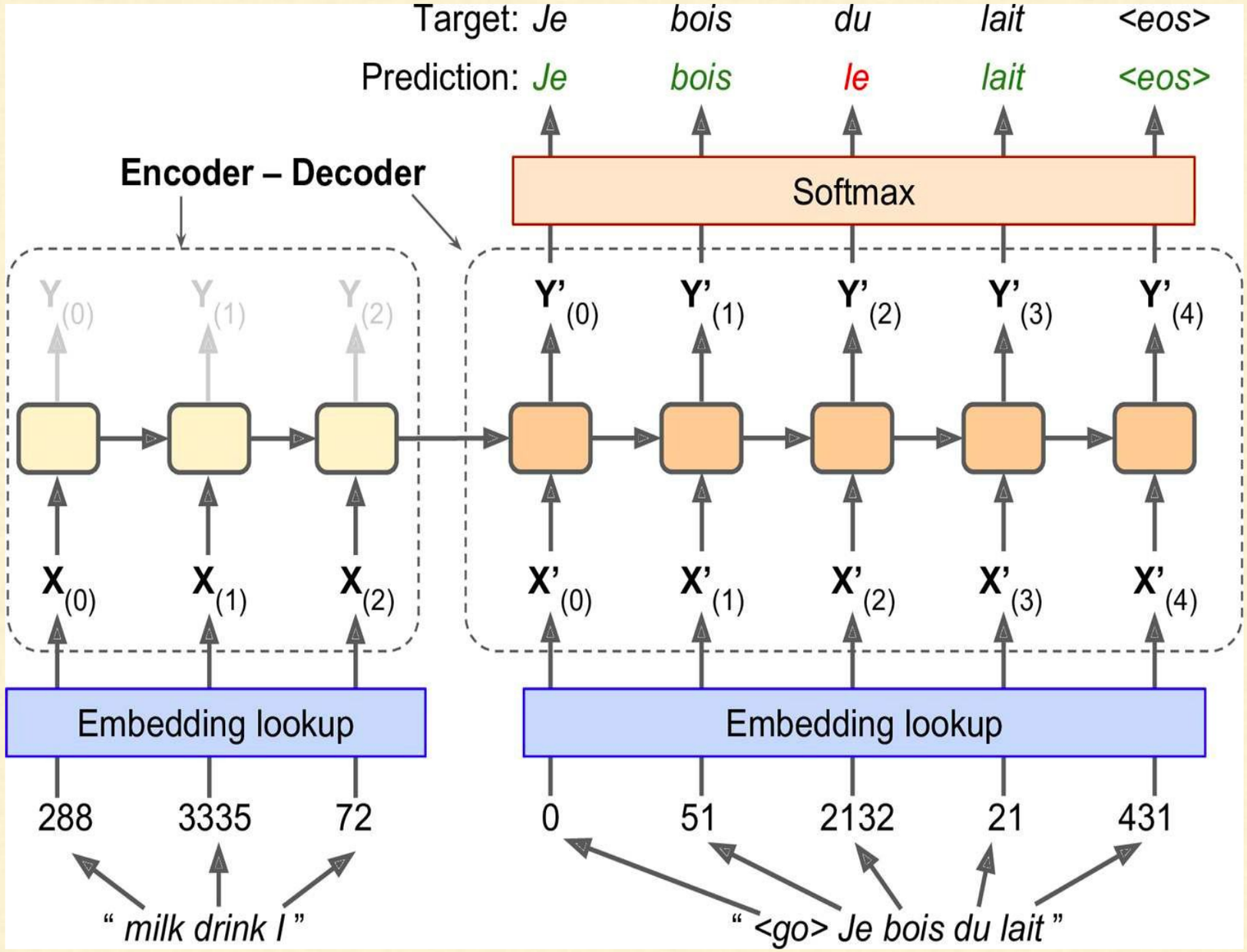
Question:

Why are the English sentences reversed before feeding it to the encoder??

Here **"I drink milk"** is reversed to **"milk drink I"**

Machine Translation

An Encoder–Decoder Network for Machine Translation

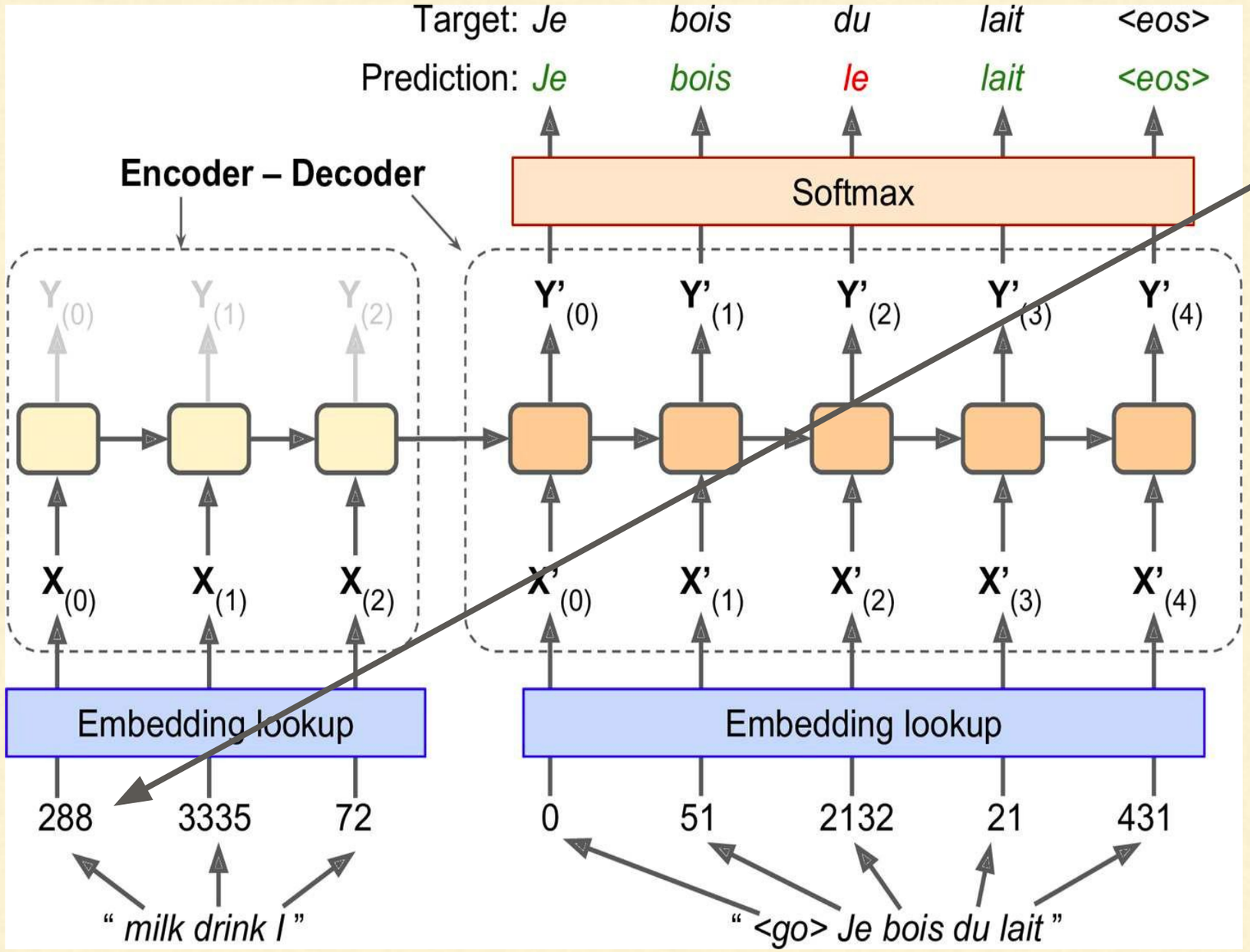


Answer:

This ensures that the beginning of the English sentence will be fed last to the encoder, which is useful because that's generally the first thing that the decoder needs to translate

Machine Translation

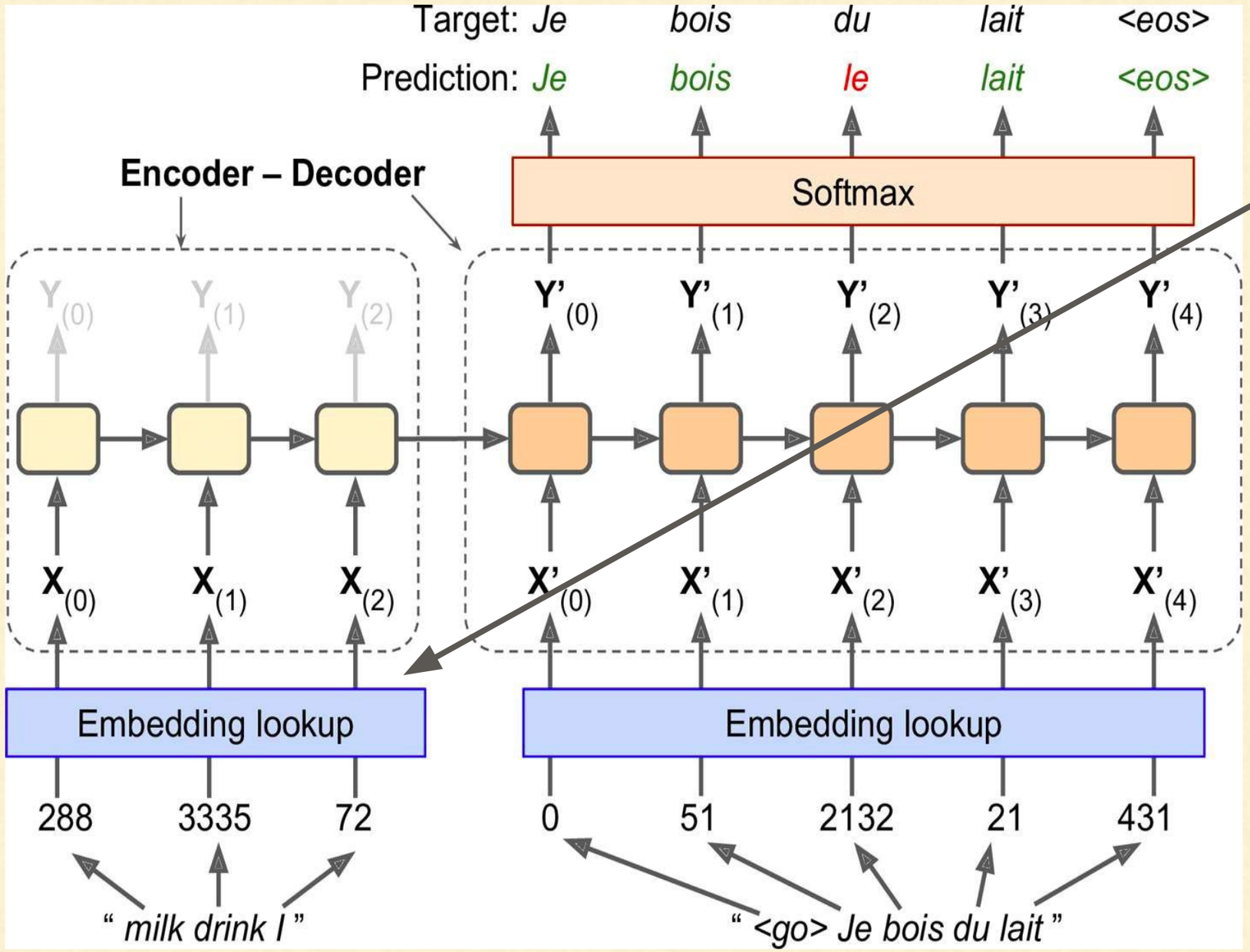
An Encoder–Decoder Network for Machine Translation



- Each word is initially represented by a simple integer identifier
- e.g., 288 for the word "milk"

Machine Translation

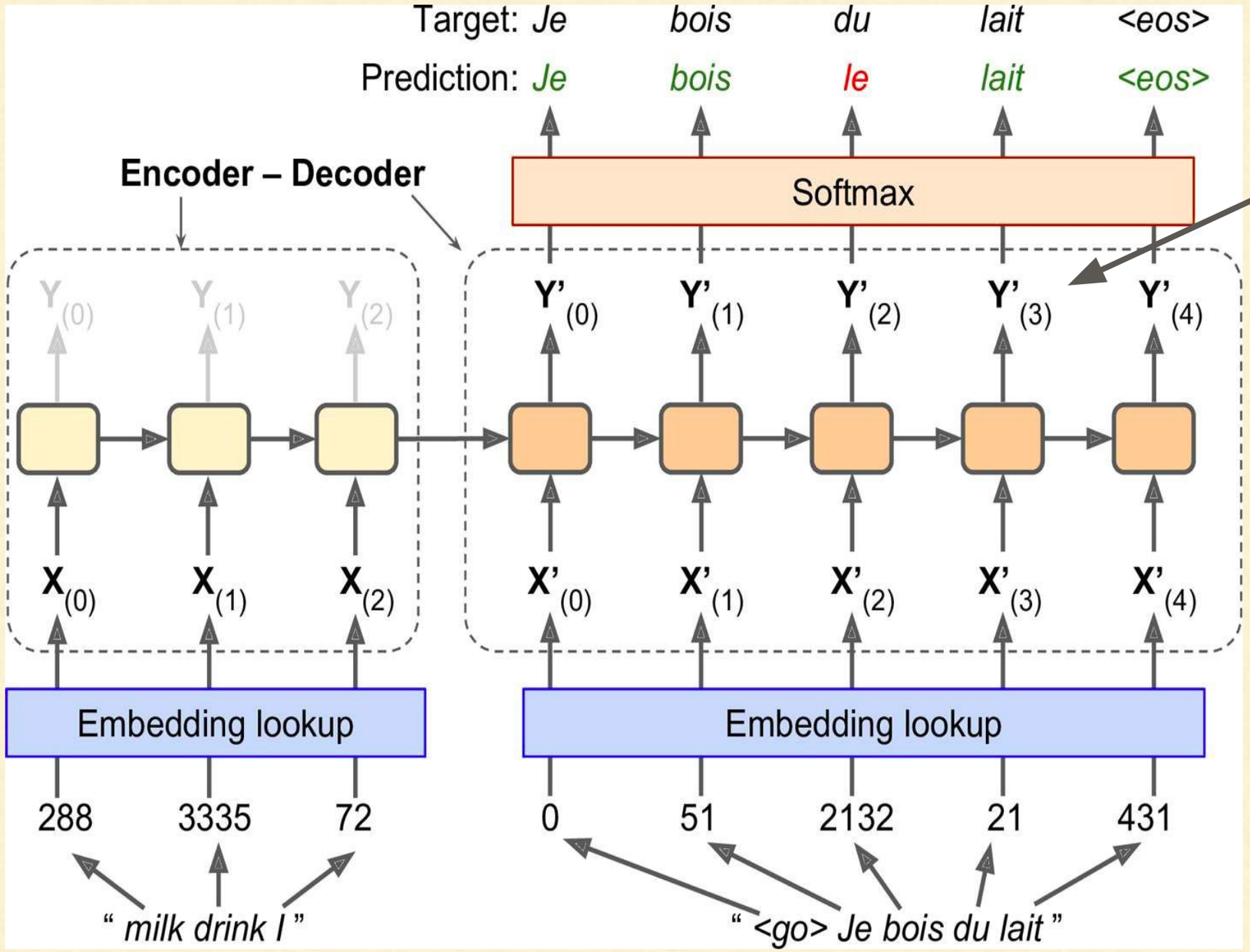
An Encoder–Decoder Network for Machine Translation



- Next, an embedding lookup returns the word embedding
- This is a dense, fairly low-dimensional vector
- These word embeddings are what is actually fed to the encoder and the decoder

Machine Translation

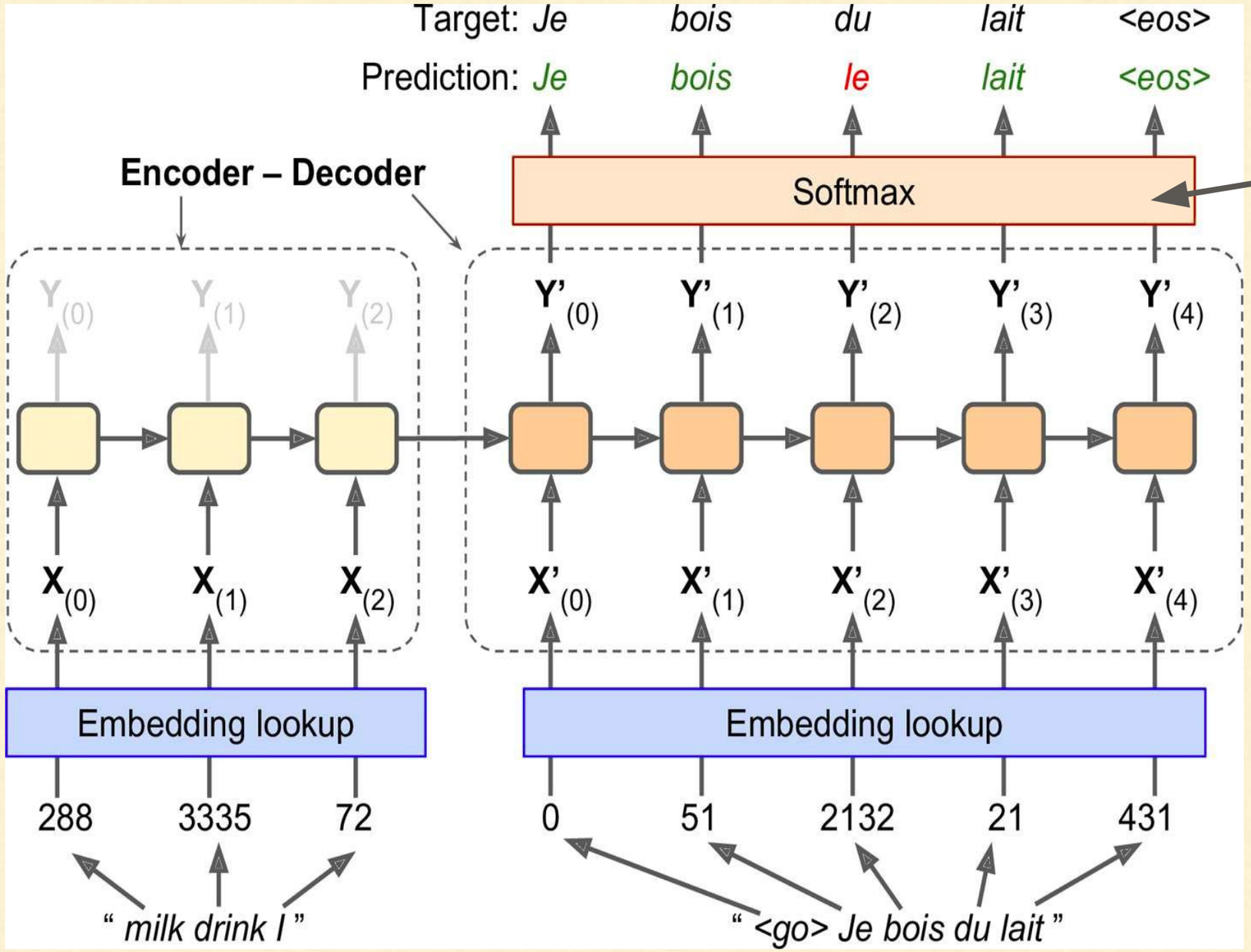
An Encoder–Decoder Network for Machine Translation



- At each step, the decoder outputs a score for each word in the output vocabulary i.e., French,

Machine Translation

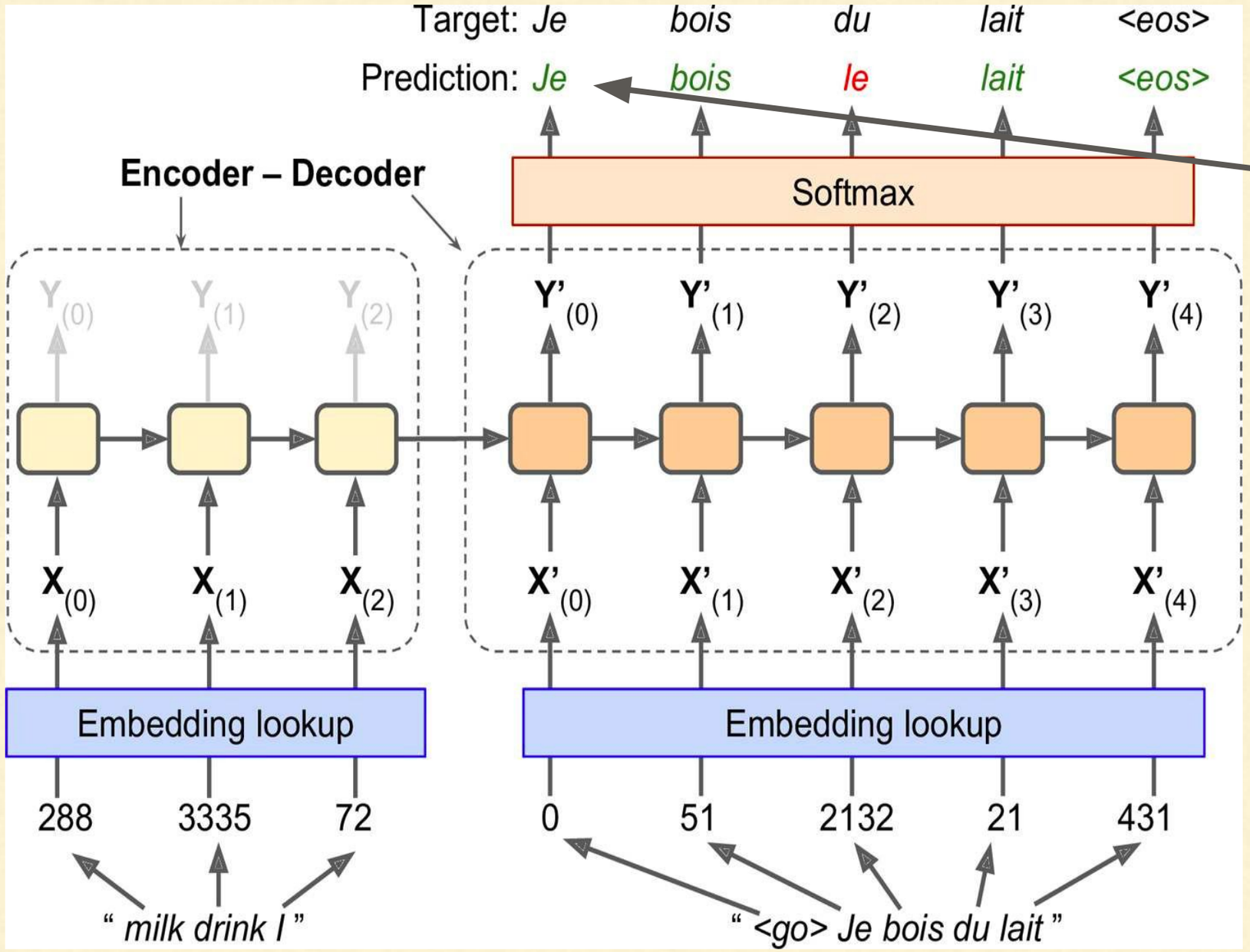
An Encoder–Decoder Network for Machine Translation



- And then the **Softmax layer** turns these scores into probabilities

Machine Translation

An Encoder–Decoder Network for Machine Translation



- For example, at the first step the word "Je" may have a probability of 20%, "Tu" may have a probability of 1%, and so on
- The word with the highest probability is output

Machine Translation

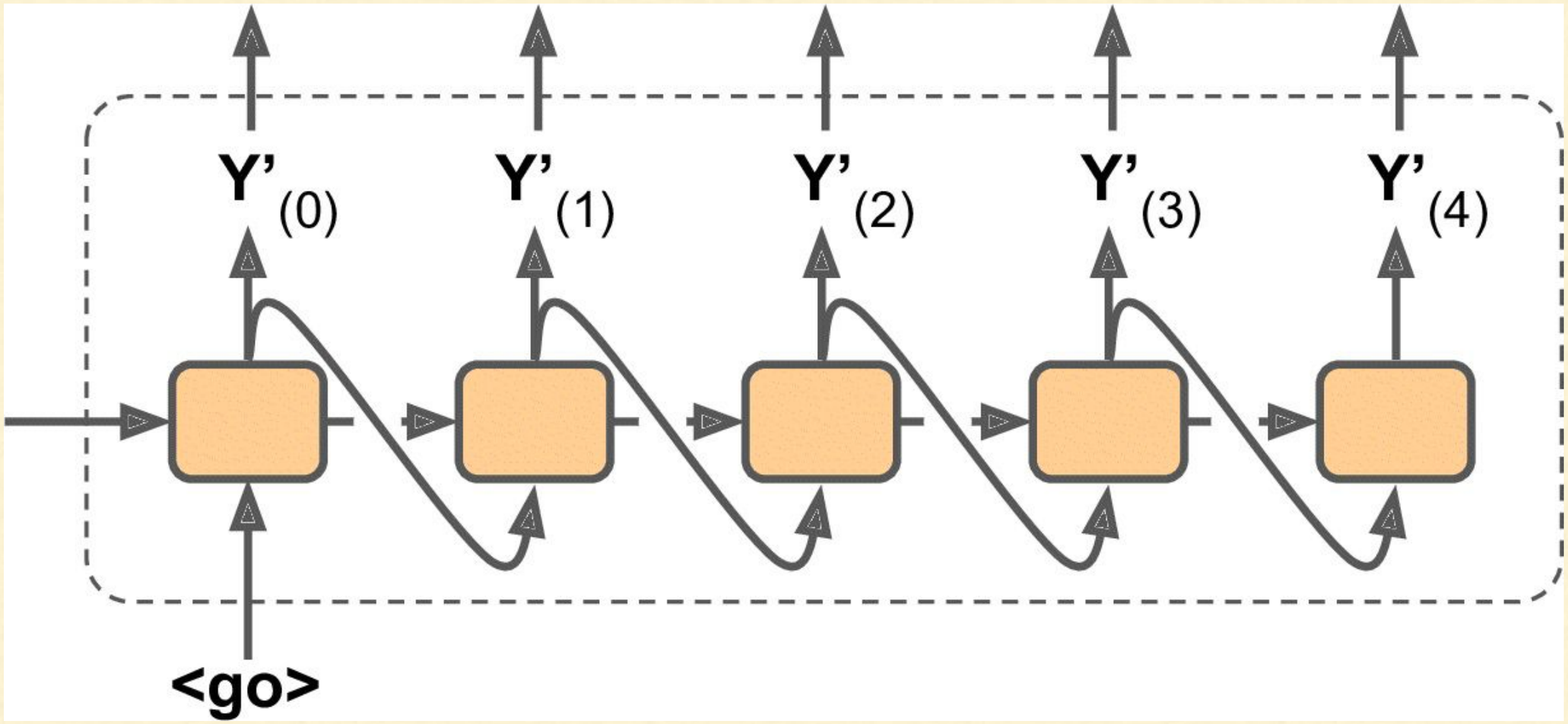
An Encoder–Decoder Network for Machine Translation

How can we use this Encoder–Decoder Network for Machine Translation at the inference time, since we will not have the target sentence to feed to the decoder ??

Machine Translation

An Encoder–Decoder Network for Machine Translation

- We will simply feed the decoder the word that it output at the previous step
- This will require an embedding lookup that is not shown on the diagram



Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com