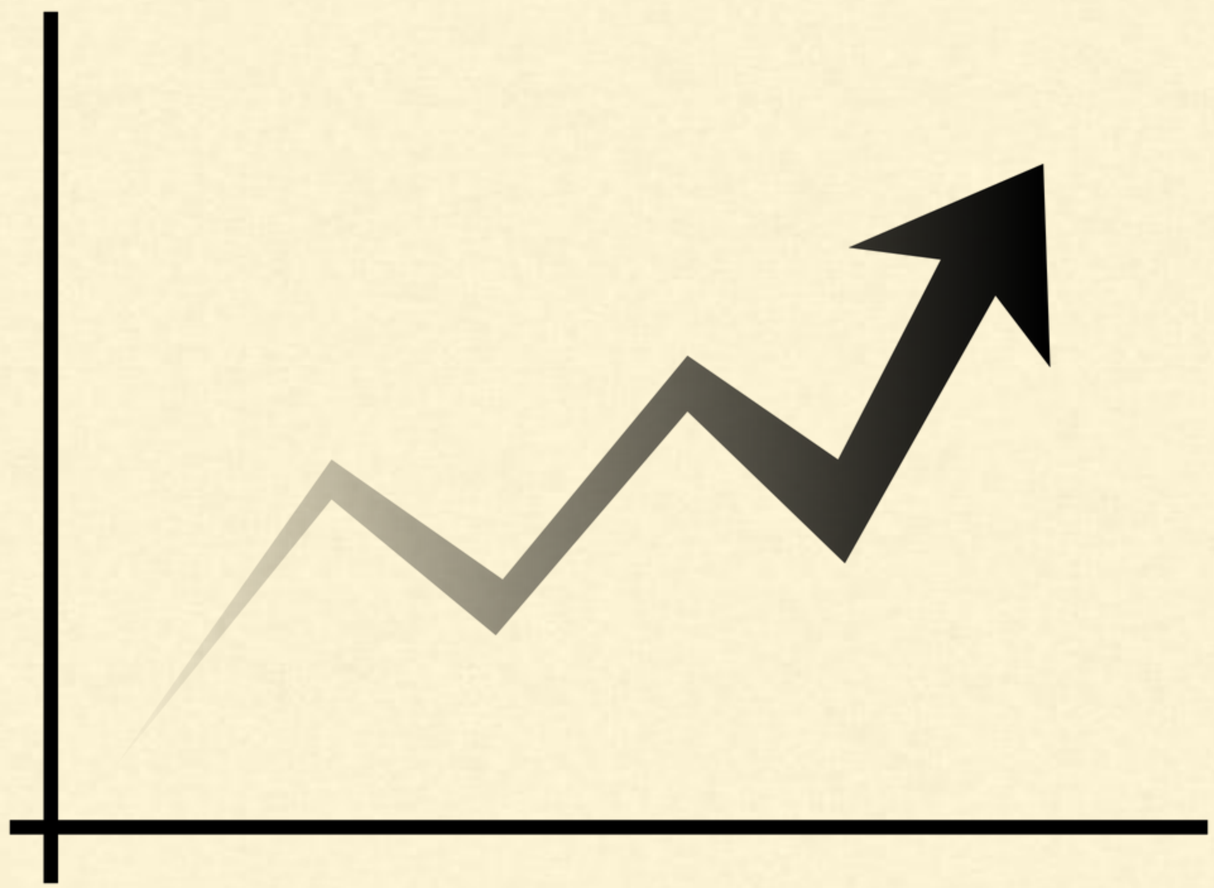
Recurrent Neural Network

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

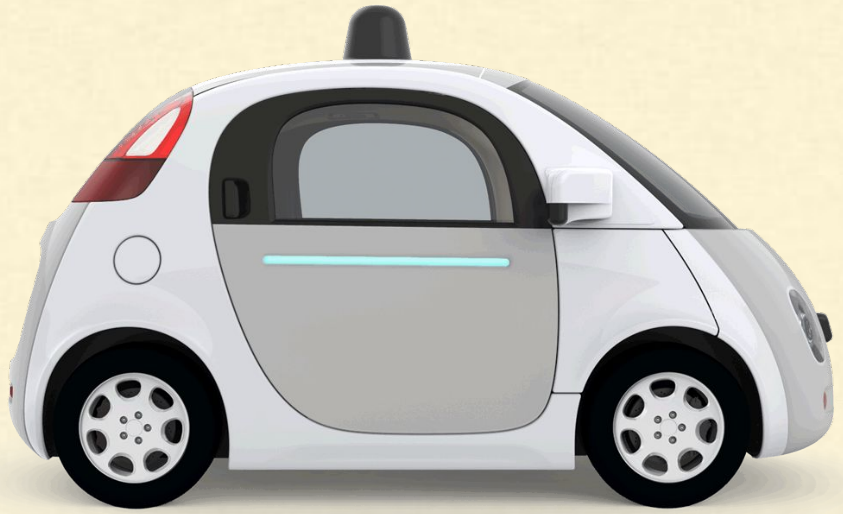
Recurrent Neural Network - Applications

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



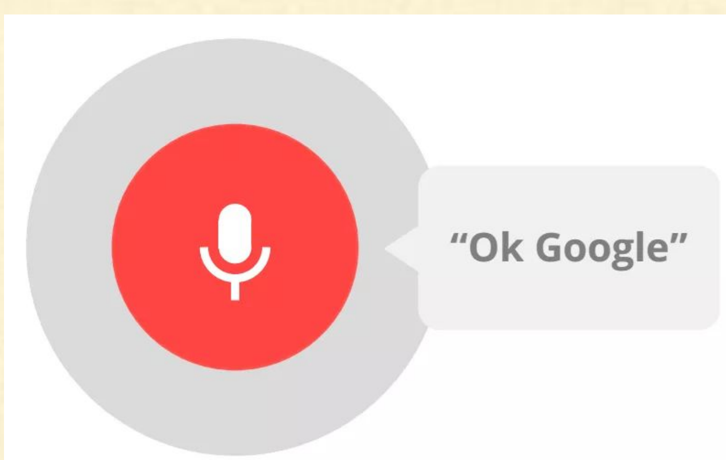
Recurrent Neural Network - Applications

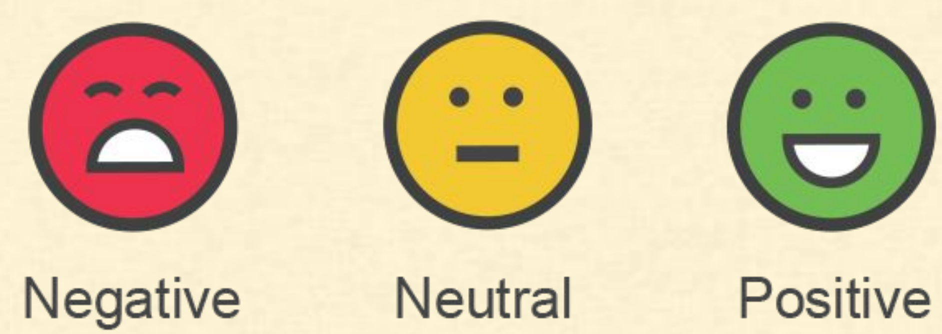
- In autonomous driving systems, RNN can
    - Anticipate car trajectories and
    - Help avoid accidents



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

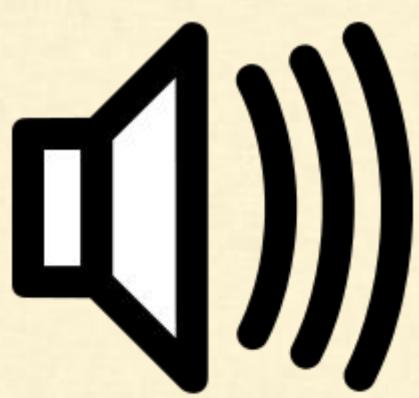




Recurrent Neural Network - Applications

- 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

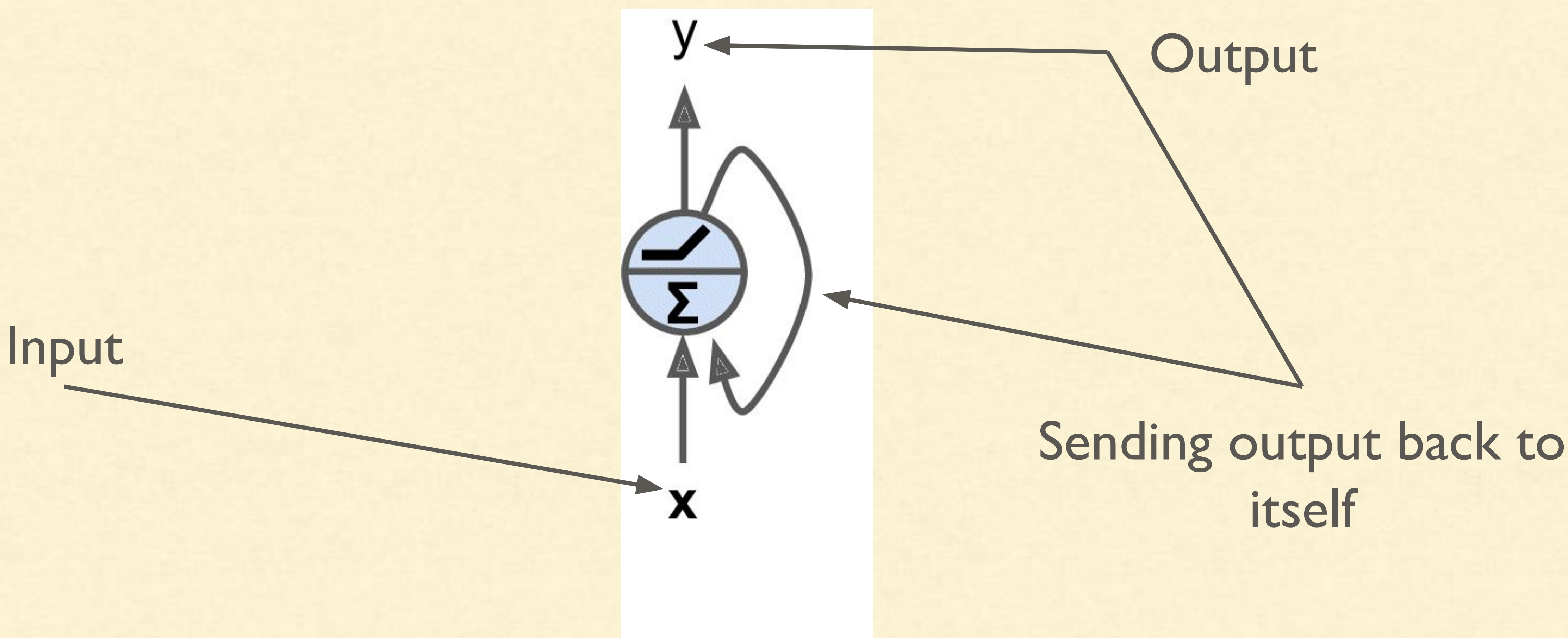
Recurrent Neurons

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

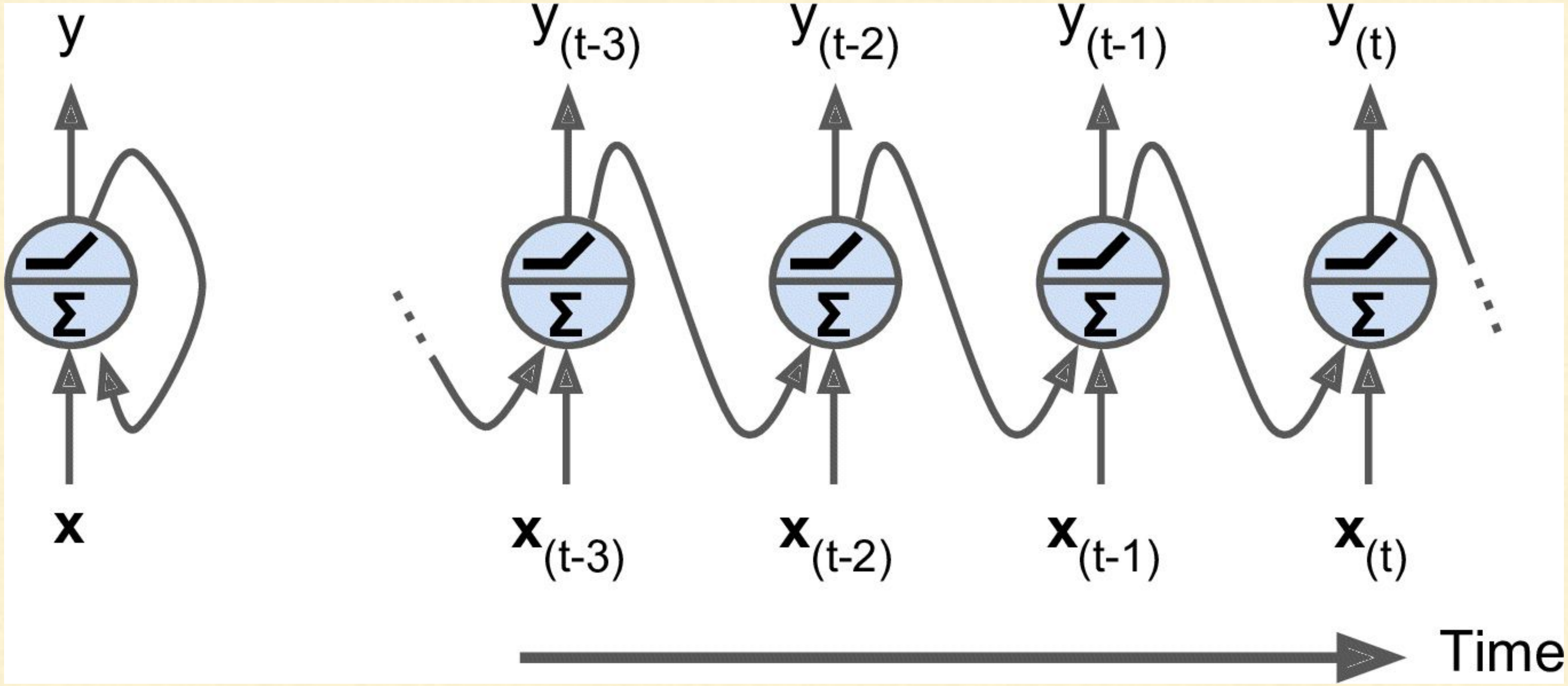
Recurrent Neurons

- 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

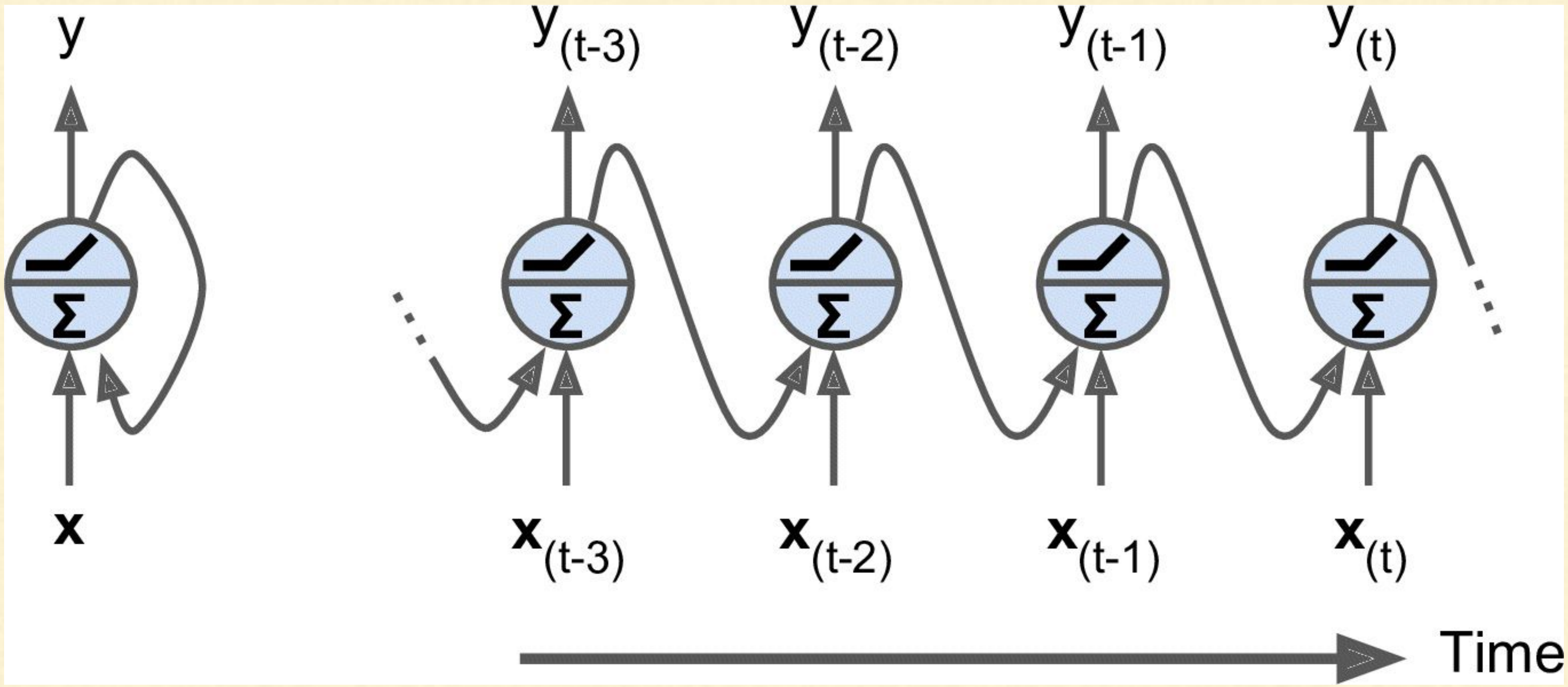
- At each time step **t** (also called a frame)
  - This recurrent neuron receives the inputs $\mathbf{x}_{(t)}$
  - As well as its own output from the previous time step $\mathbf{y}_{(t-1)}$



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

Recurrent Neurons

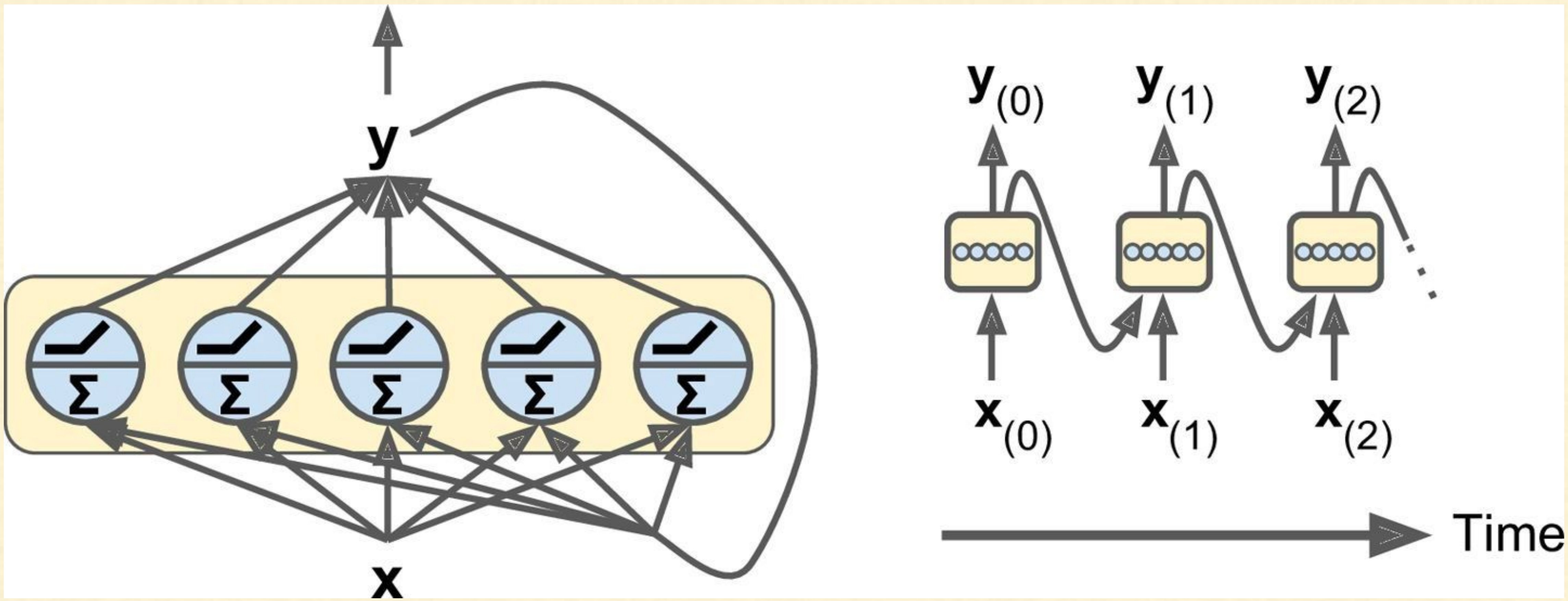
- We can represent this tiny network against the time axis (See below figure)
- This is called ***unrolling the network through time***



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

$$
\begin{aligned}
\mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}\right) \\
&= \phi\left(\begin{bmatrix}\mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)}\end{bmatrix} \cdot \mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}
\end{aligned}
$$

Recurrent Neurons

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

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

- $\mathbf{Y}_{(t)}$ 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

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

- $\mathbf{X}_{(t)}$ is an m × 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

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

- $\mathbf{W}_x$ 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

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

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

Memory Cells

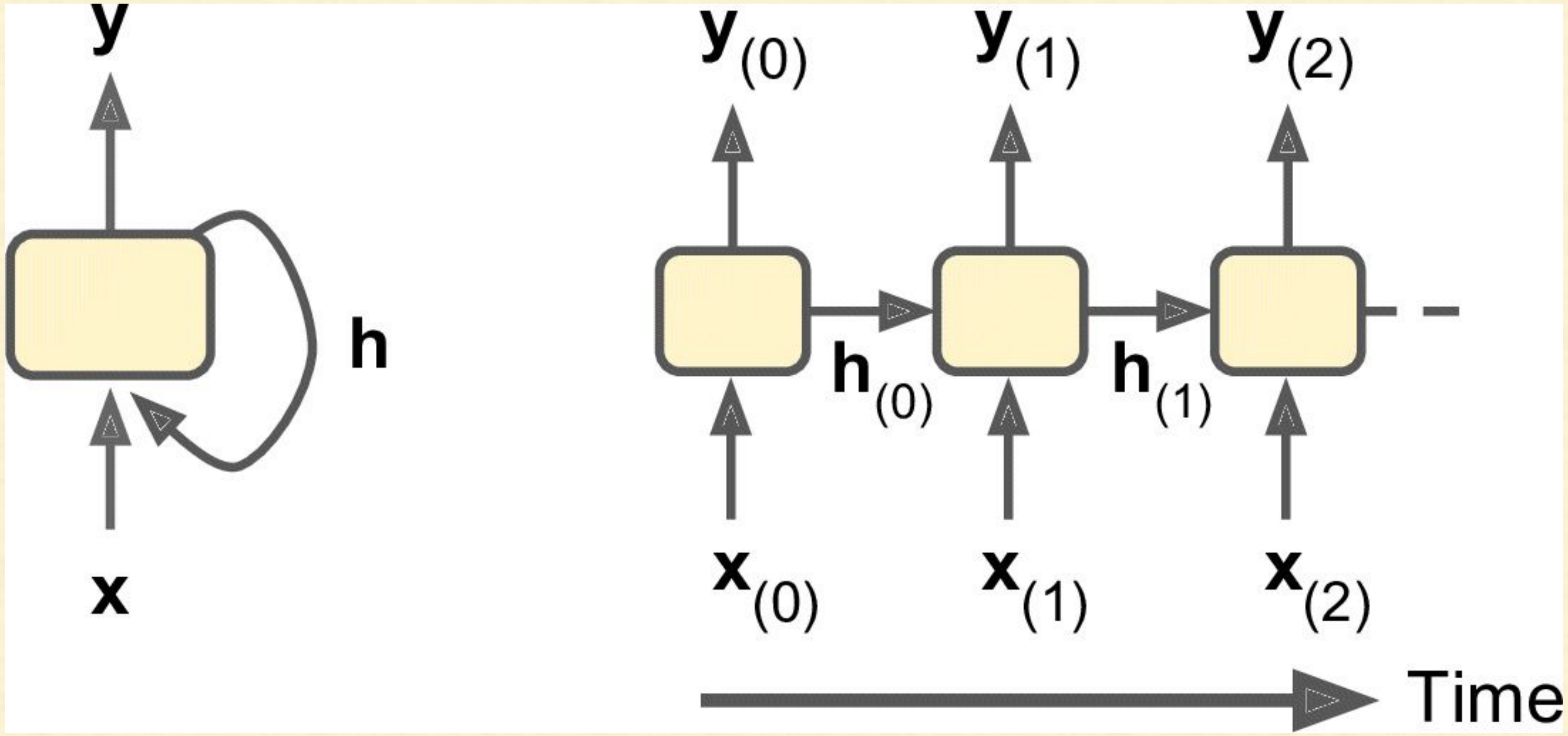
- Since the output of a recurrent neuron at time step t is a
  - Function of all the inputs from previous time steps
  - We can say that it has a form of ***memory***
- A part of a neural network that
  - Preserves some state across time steps is called a **memory cell**

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

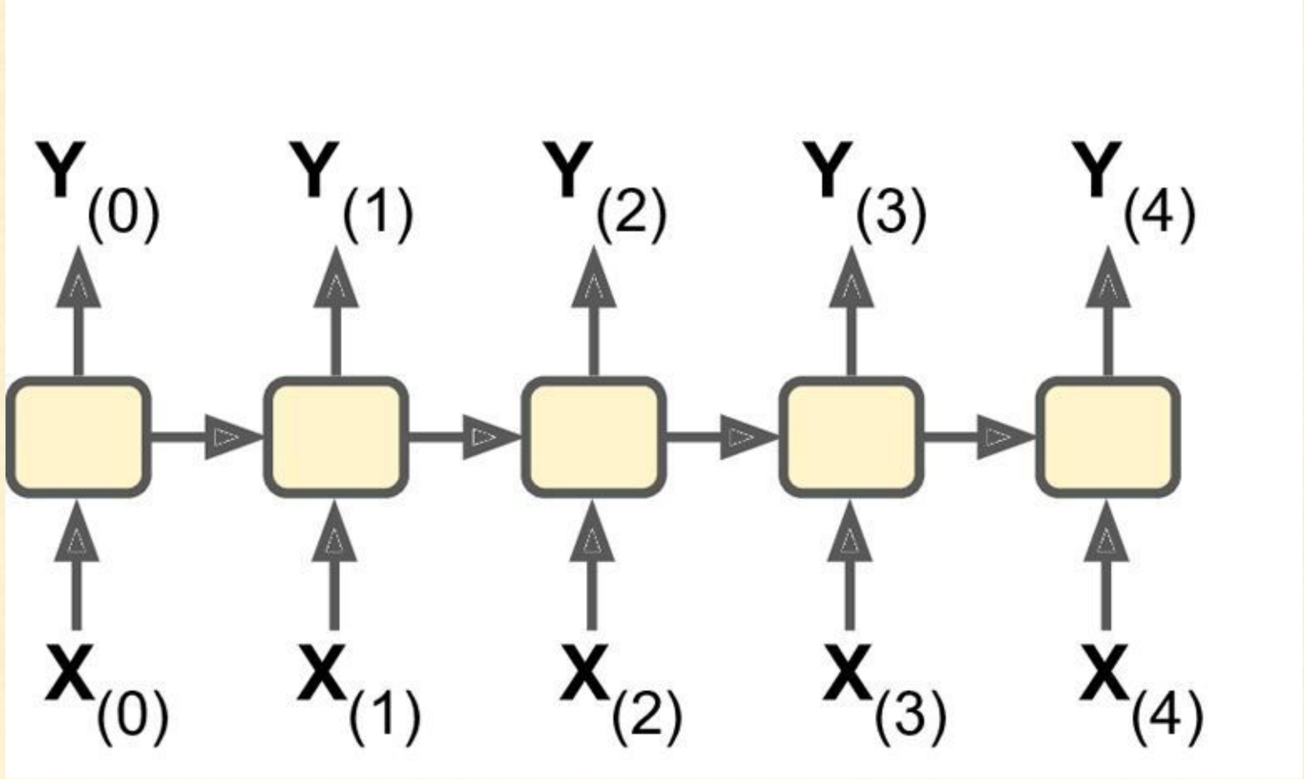


A cell's hidden state and its output may be different

Input and Output Sequences

Sequence-to-sequence Network

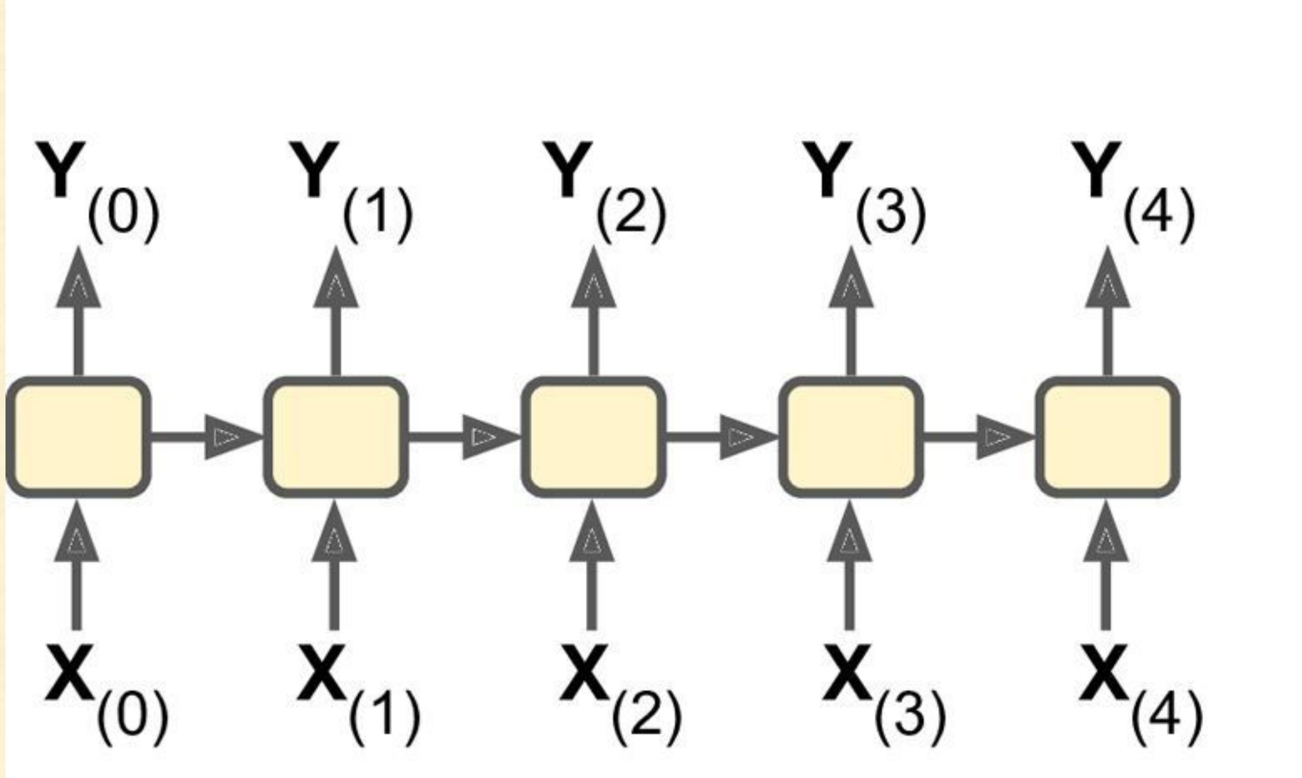
- An RNN can simultaneously take a
  - Sequence of inputs and
  - Produce a sequence of outputs



Input and Output Sequences

Sequence-to-sequence Network

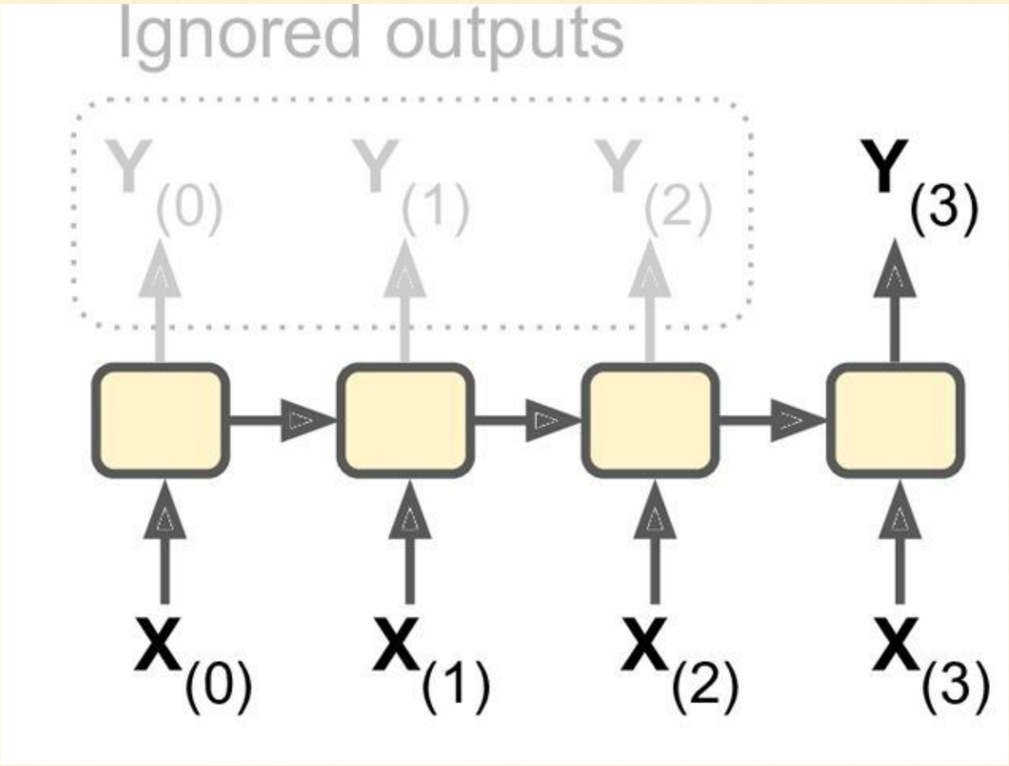
- This type of network is useful for predicting time series
  - Such as stock prices
- We feed it the prices over the last N days and
  - It must output the prices shifted by one day into the future
  - i.e., from N – 1 days ago to tomorrow



Input and Output Sequences

Sequence-to-vector Network

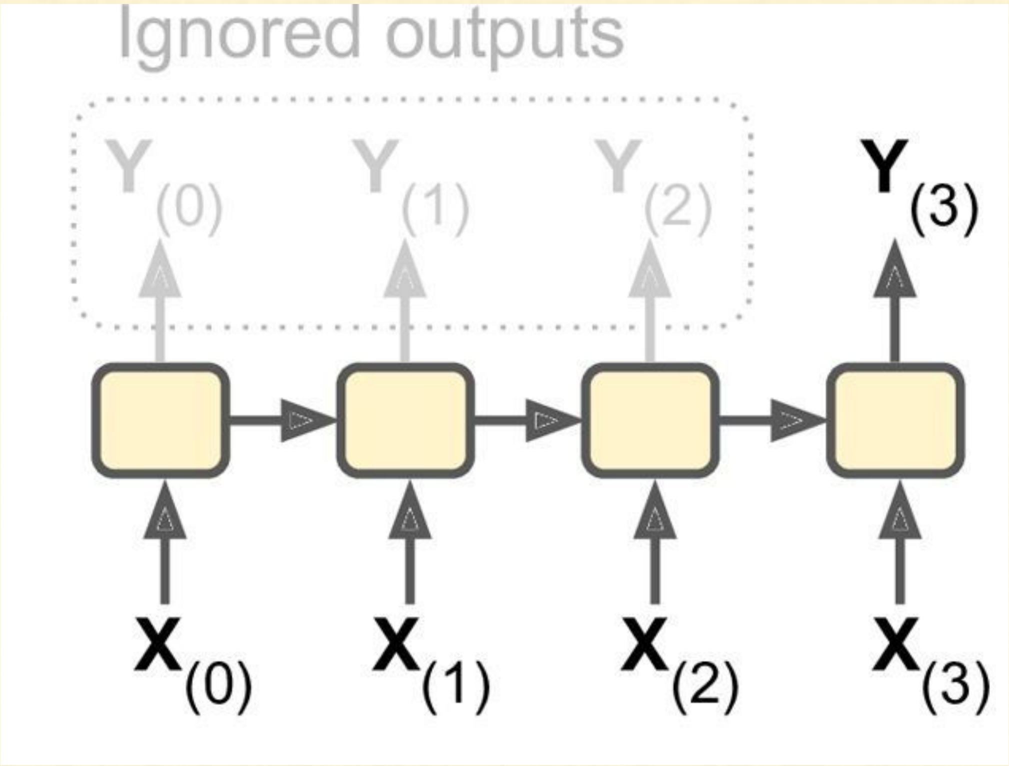
- Alternatively we could feed the network a sequence of inputs and
  - Ignore all outputs except for the last one



Input and Output Sequences

Sequence-to-vector Network

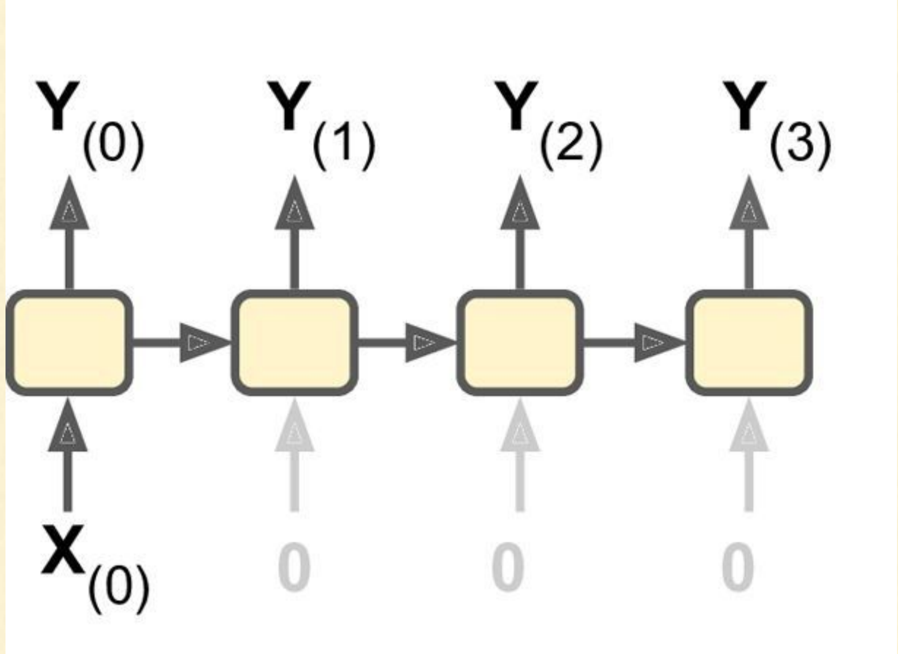
- We can feed this network a sequence of words
  - Corresponding to a movie review and
  - The network would output a sentiment score
  - e.g., from –1 [hate] to +1 [love]



Input and Output Sequences

Vector-to-sequence Network

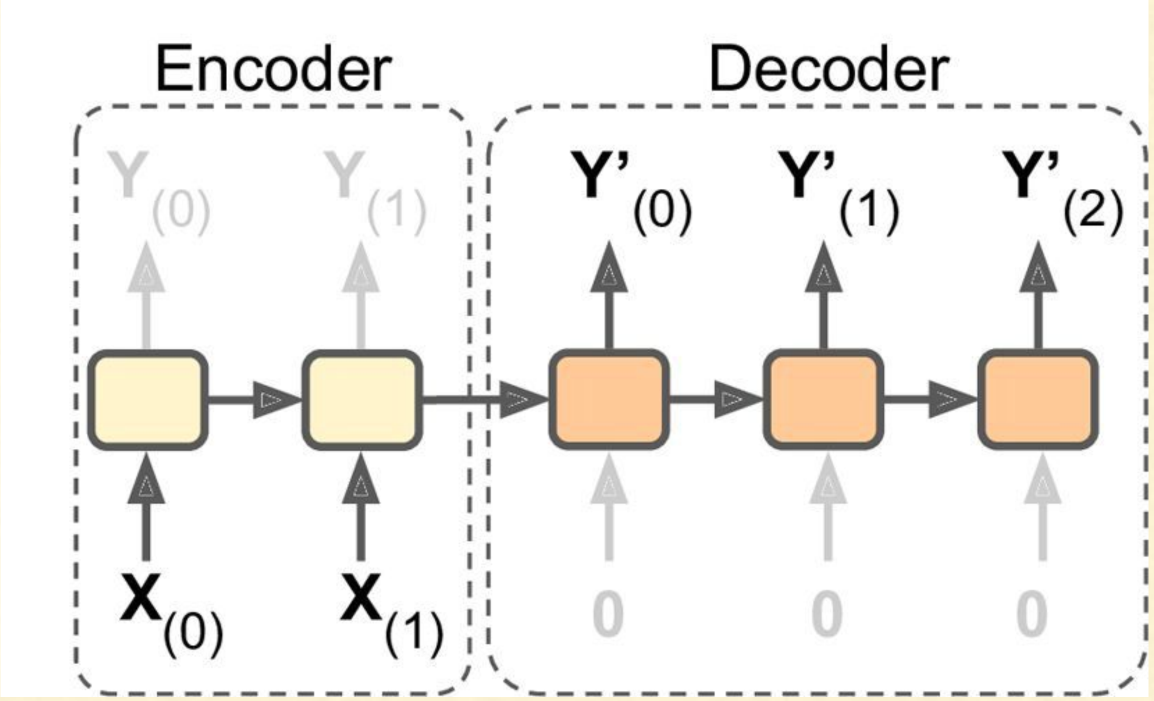
- We could feed the network a single input at the first time step and
  - Zeros for all other time steps and
  - Let is output a sequence
- For example, the input could be an image and the
  - Output could be a caption for the image



Input and Output Sequences

Encoder-Decoder

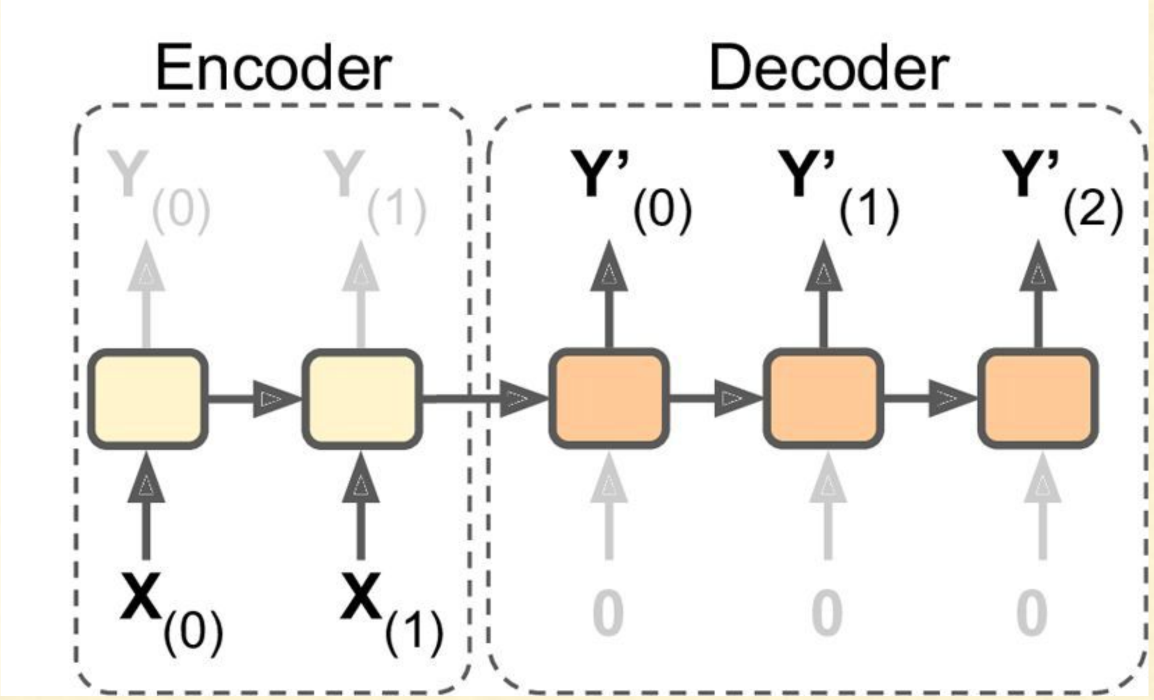
- In this network, we have
  - sequence-to-vector network, called **an encoder** followed by
  - vector-to-sequence network, called **a decoder**



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)

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

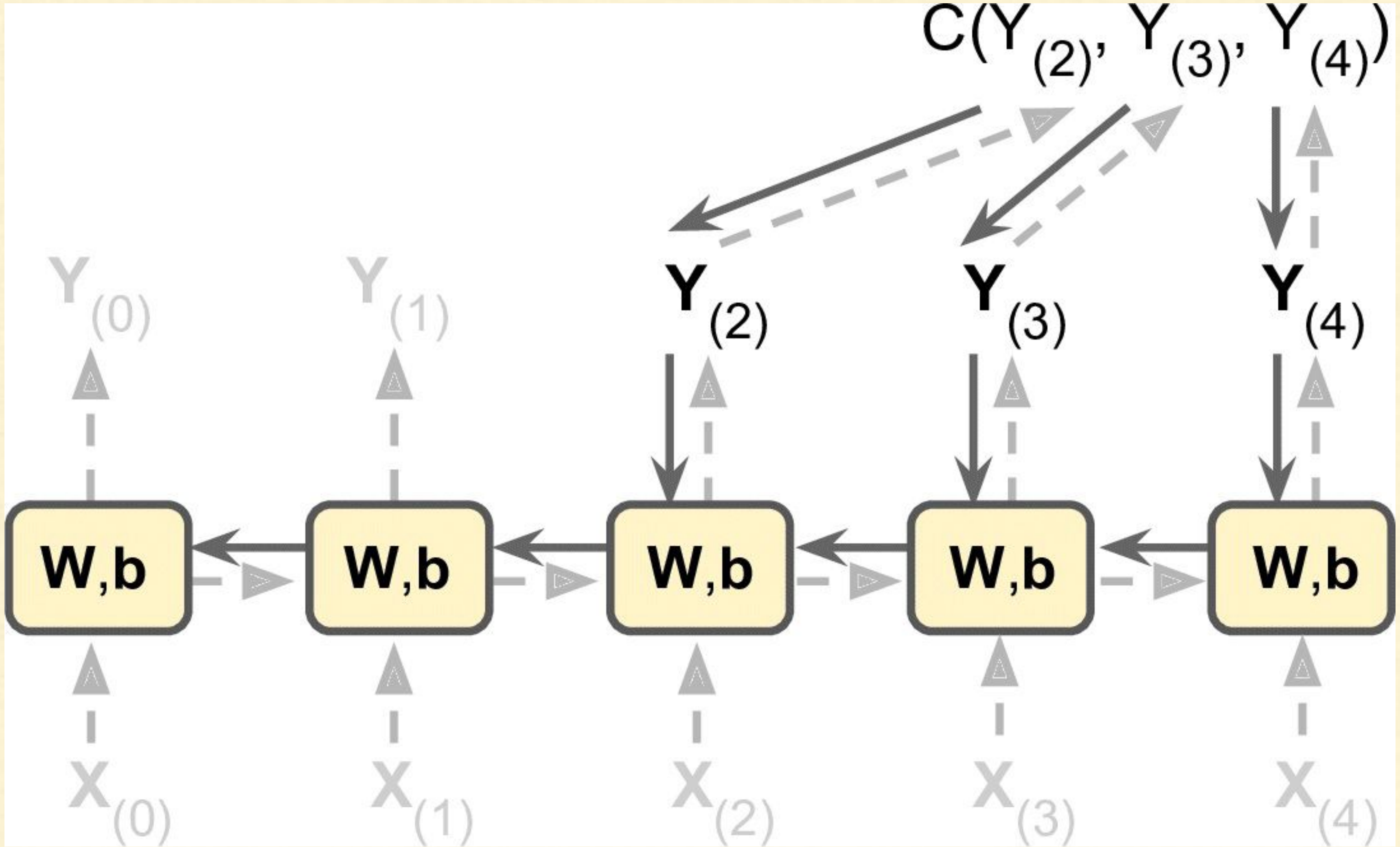
- In this case, the most common solution is to define
  - A special output called an **end-of-sequence token (EOS token)**
- Any output past the EOS should be ignored - We will discuss it later in details

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

Training RNNs

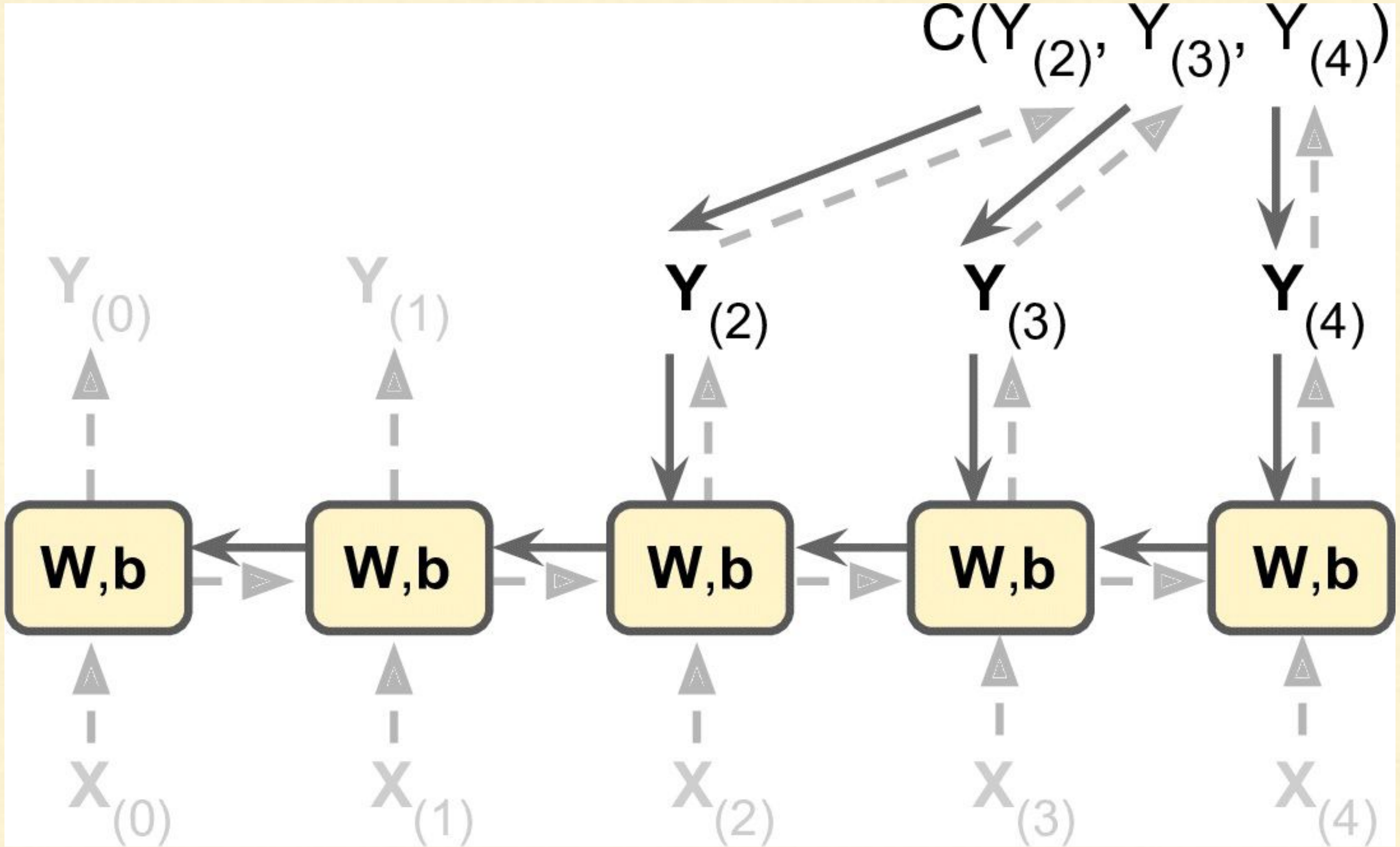
Training RNNs

- 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

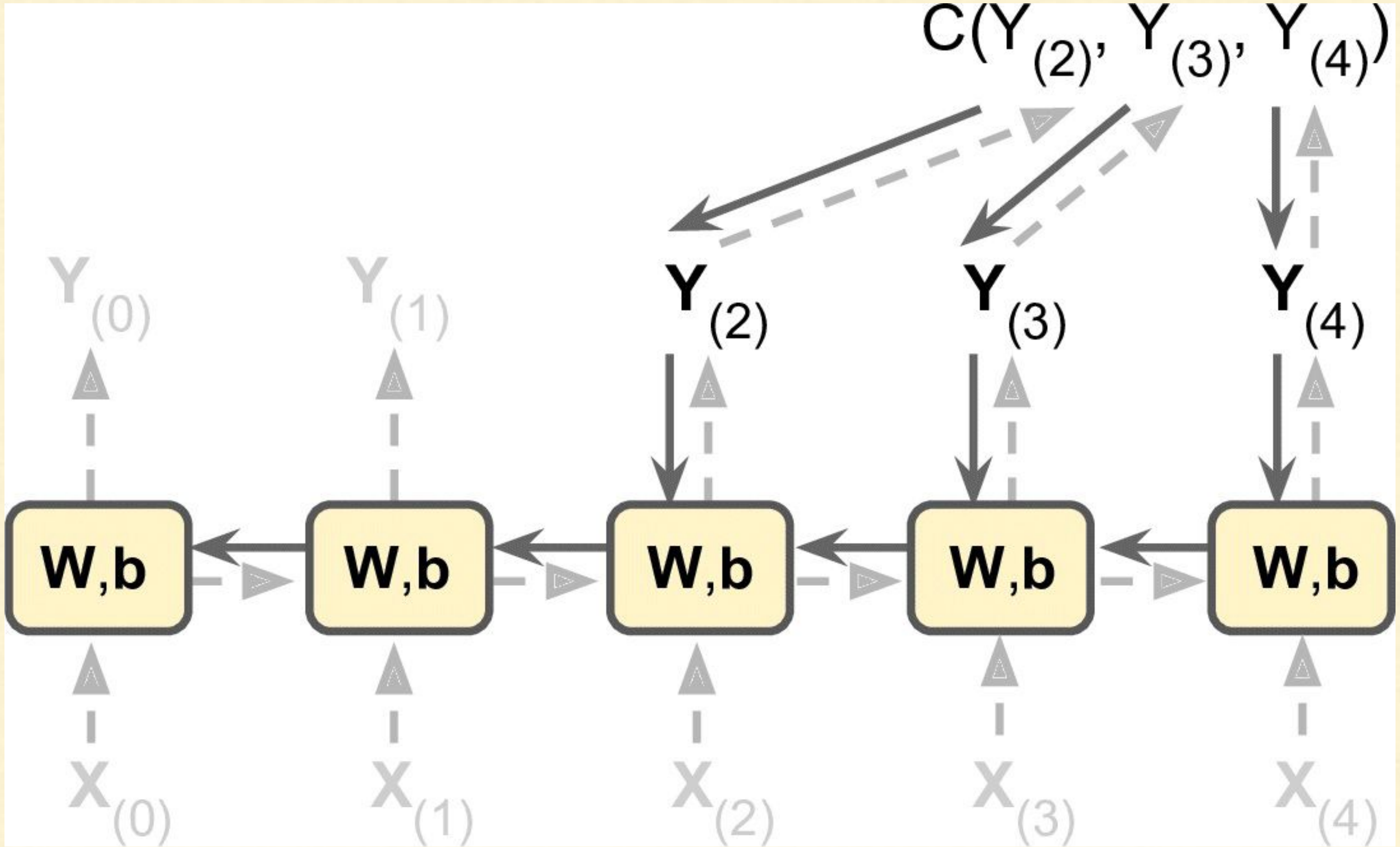
Understanding how RNNs are trained



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

Training RNNs

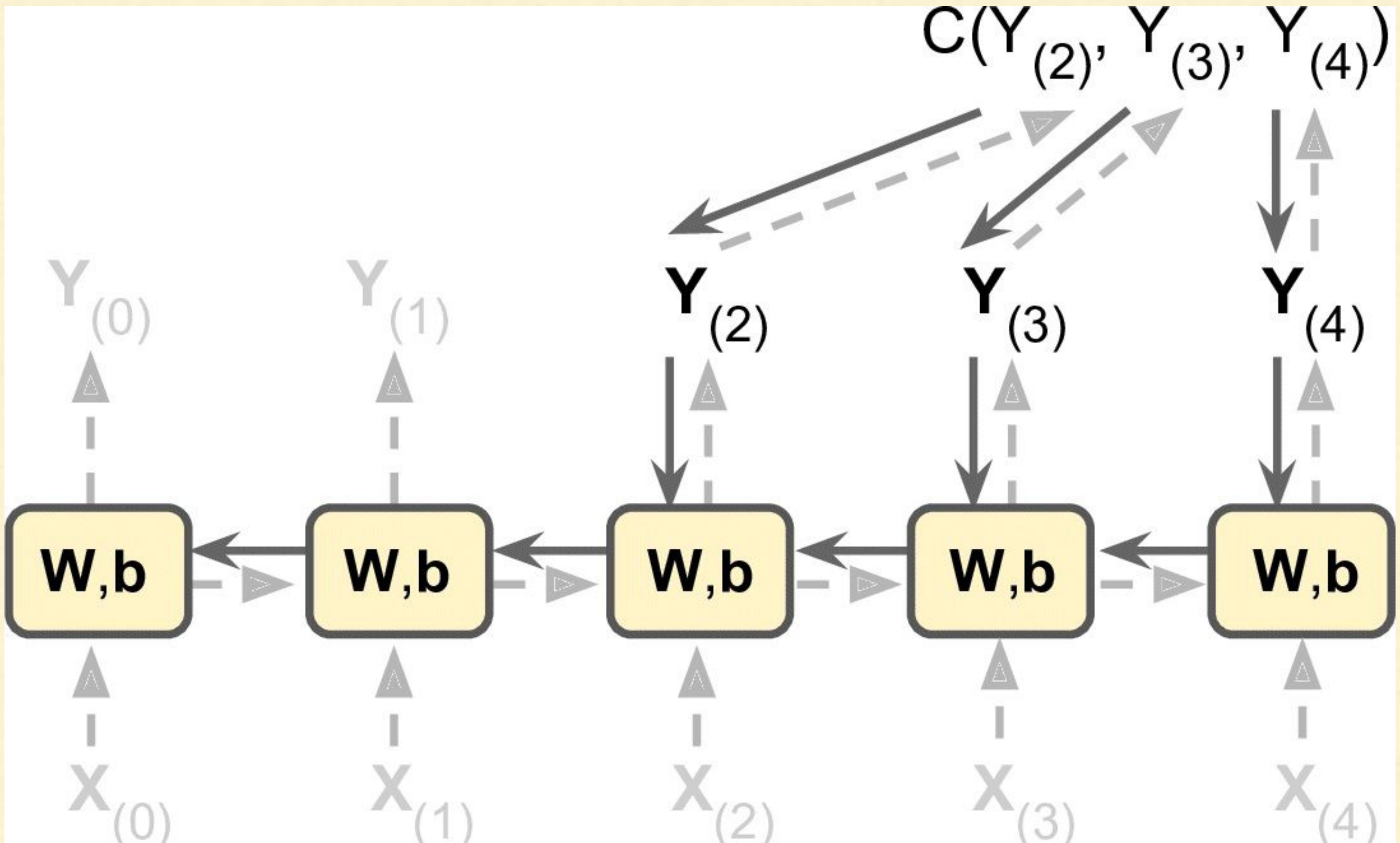
Understanding how RNNs are trained



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

Training RNNs

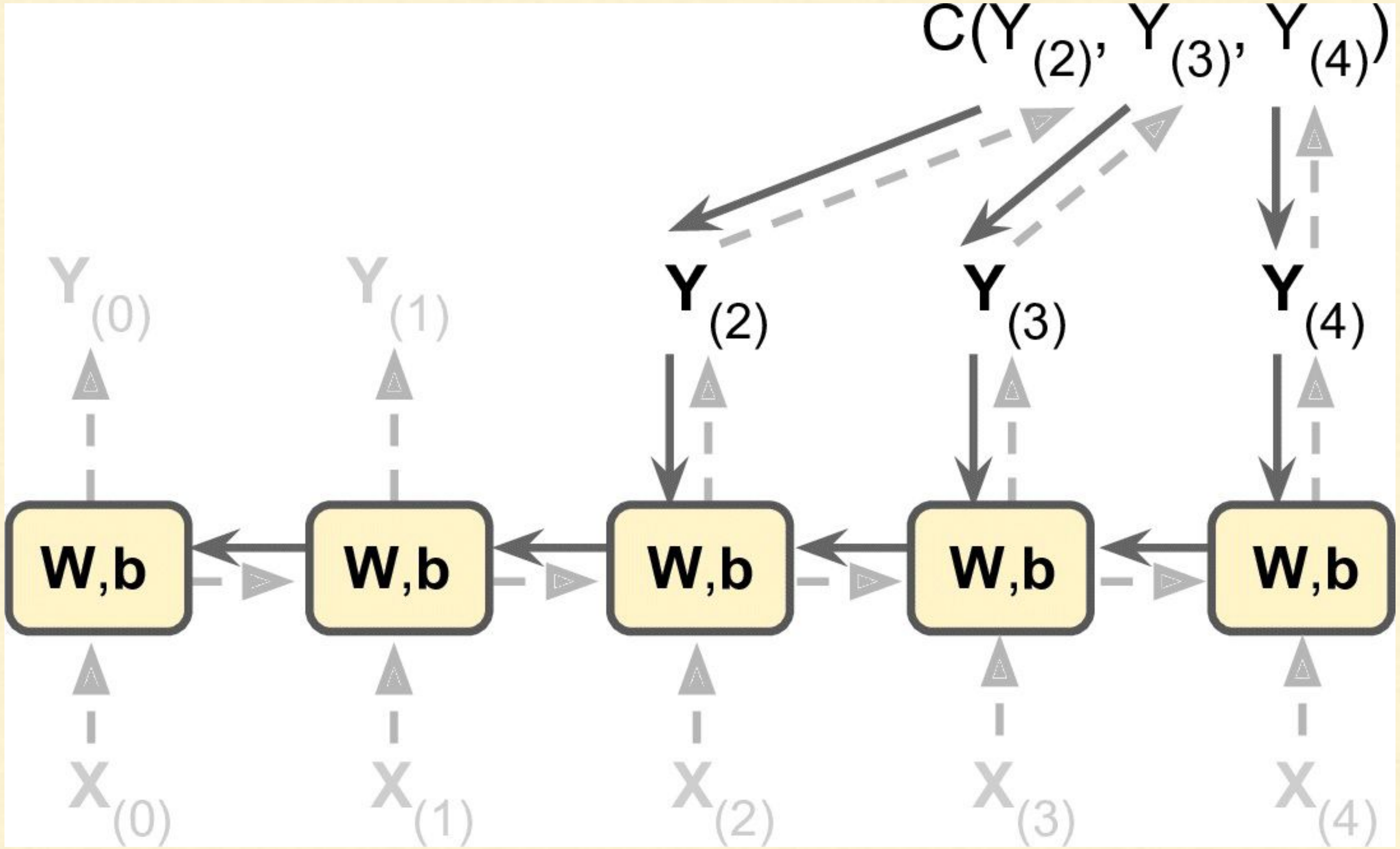
Understanding how RNNs are trained



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

Training RNNs

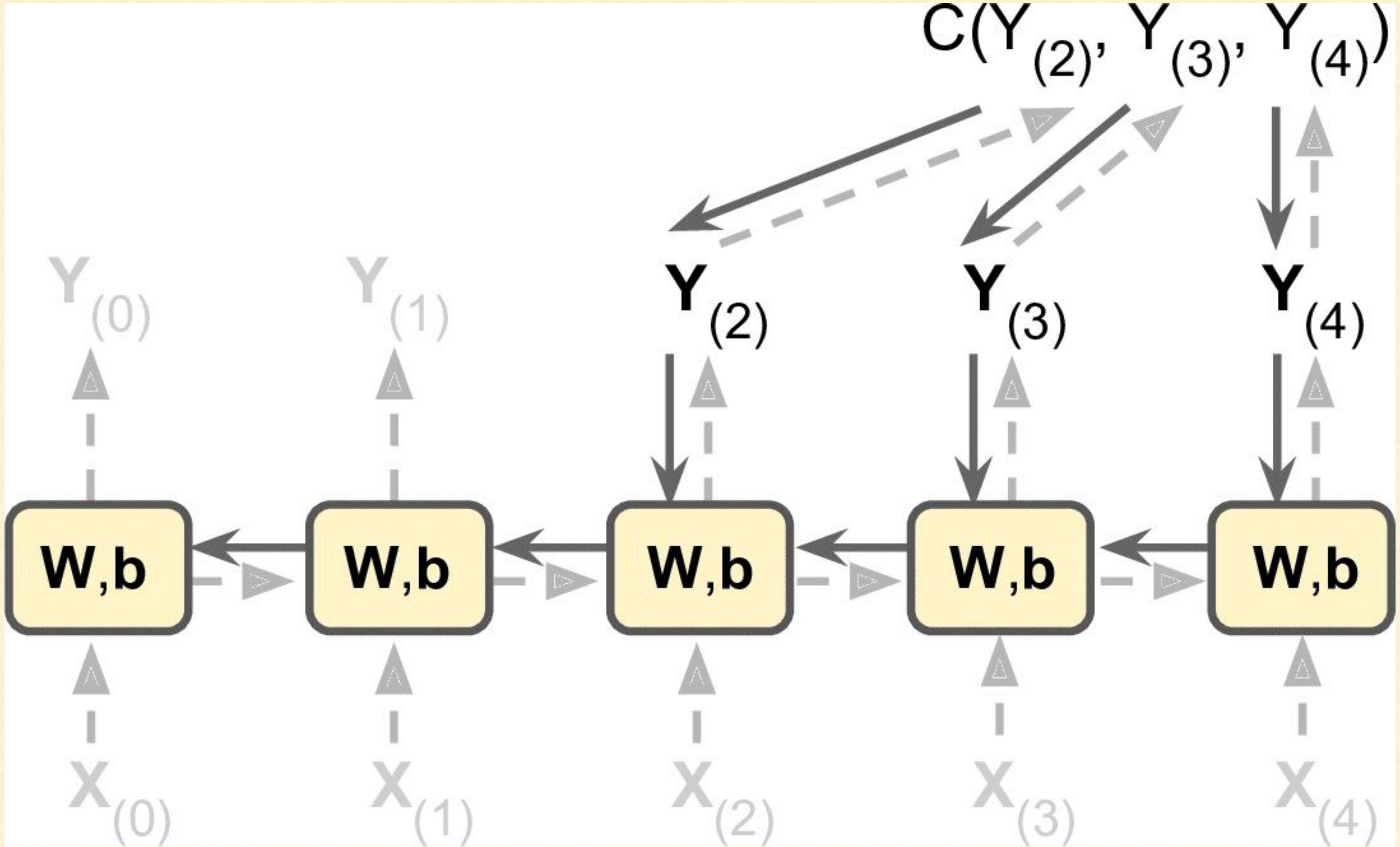
Understanding how RNNs are trained



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

Training RNNs

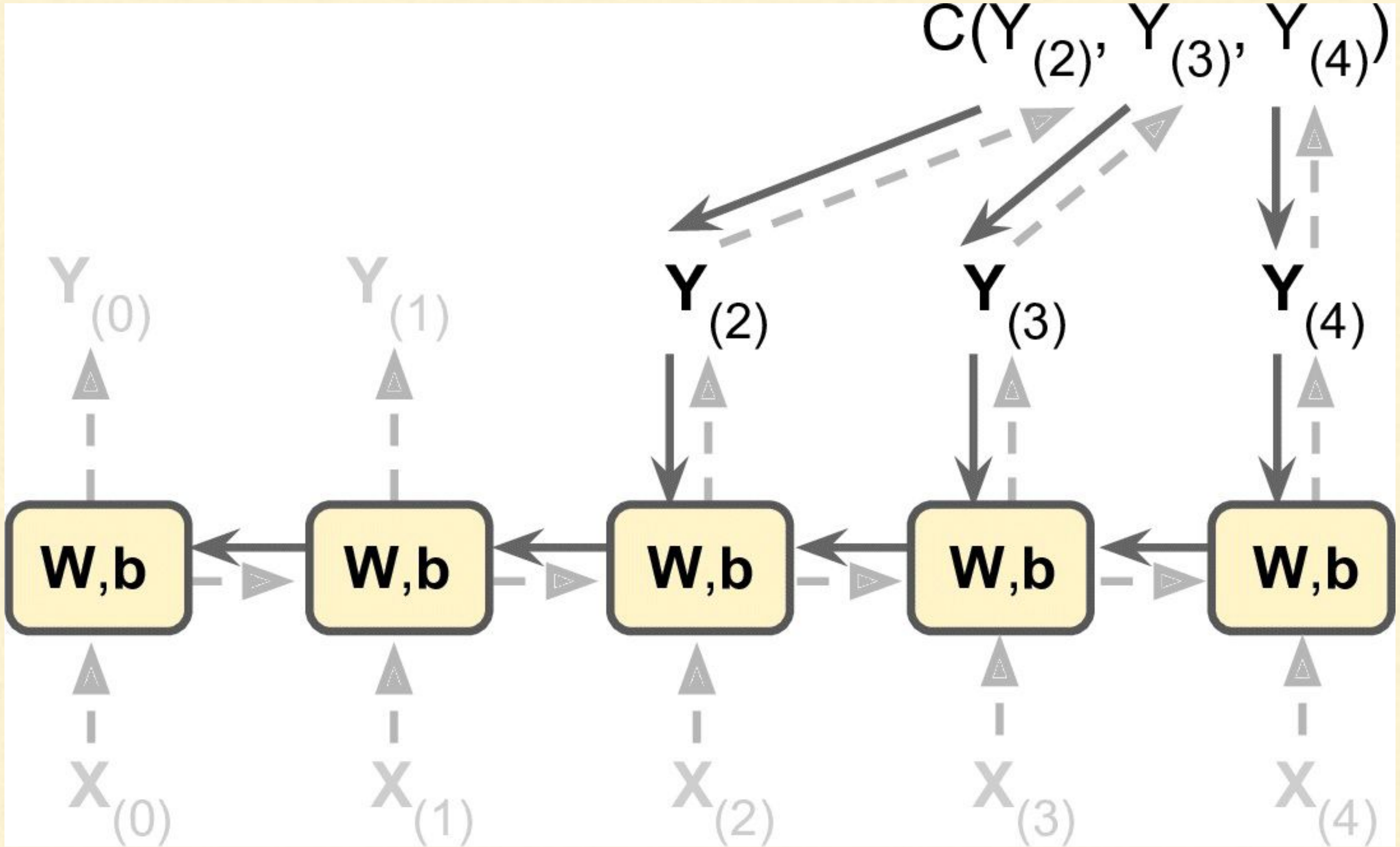
Understanding how RNNs are trained



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

Training RNNs

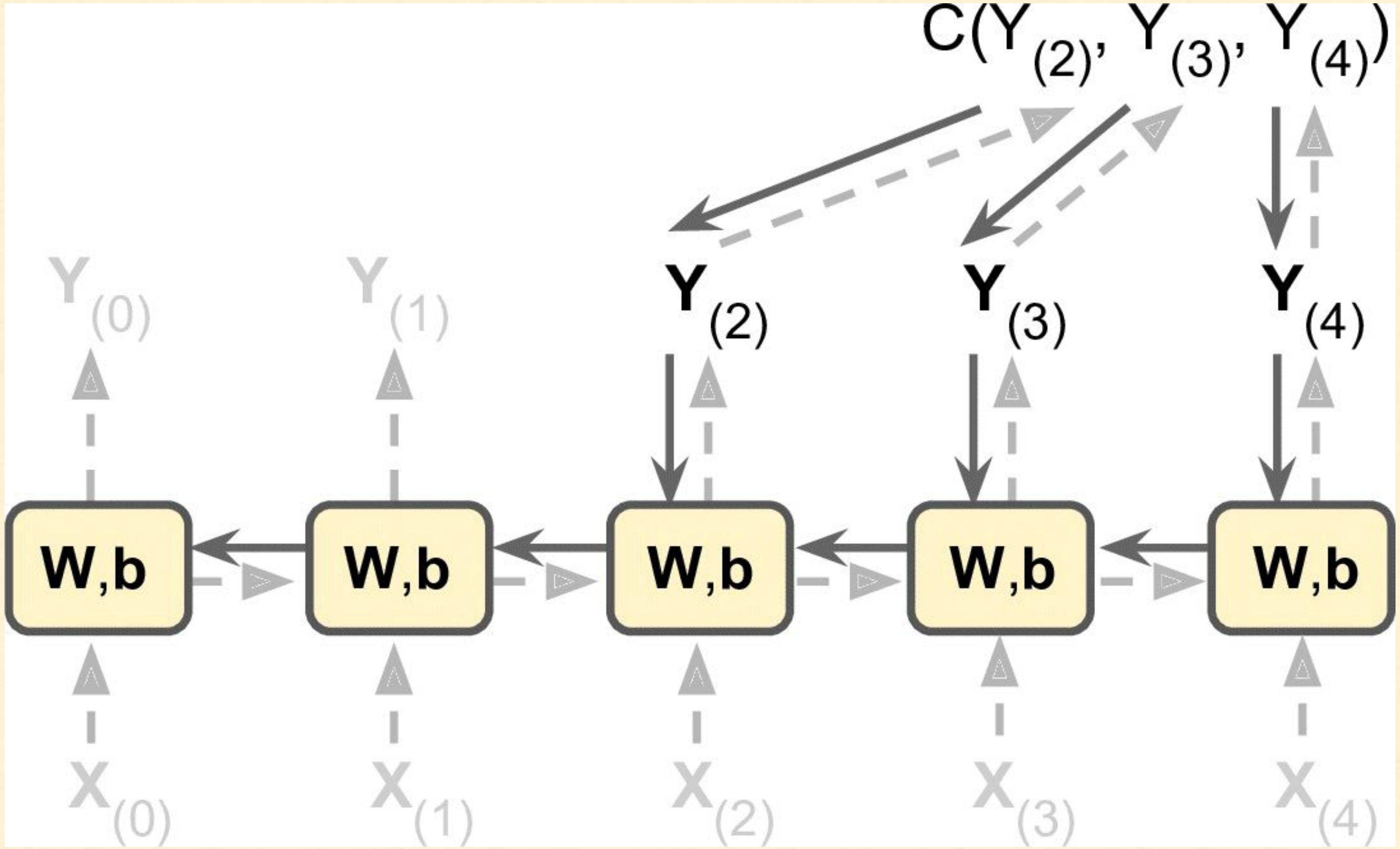
Understanding how RNNs are trained



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

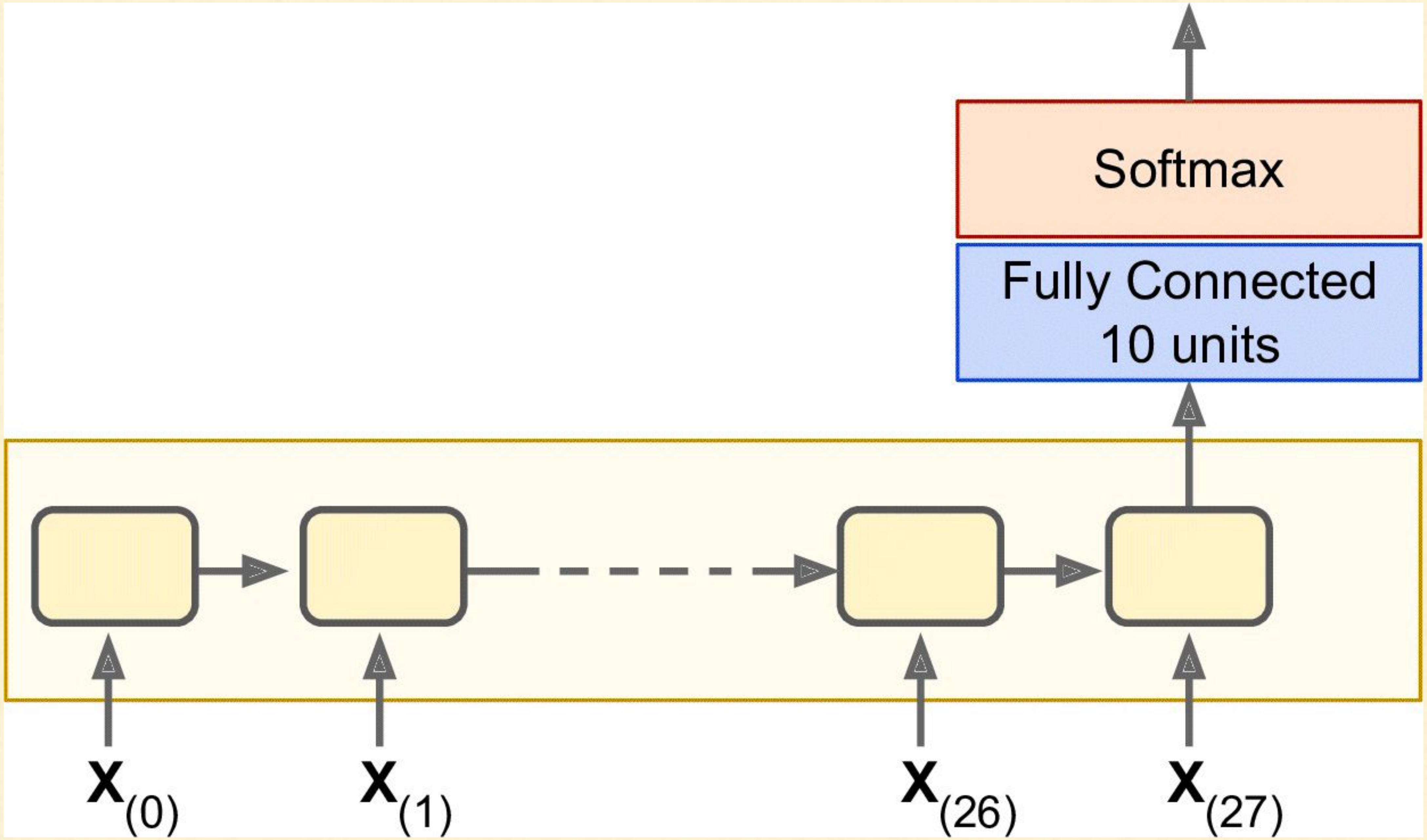
Training a Sequence Classifier

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 = fully_connected(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

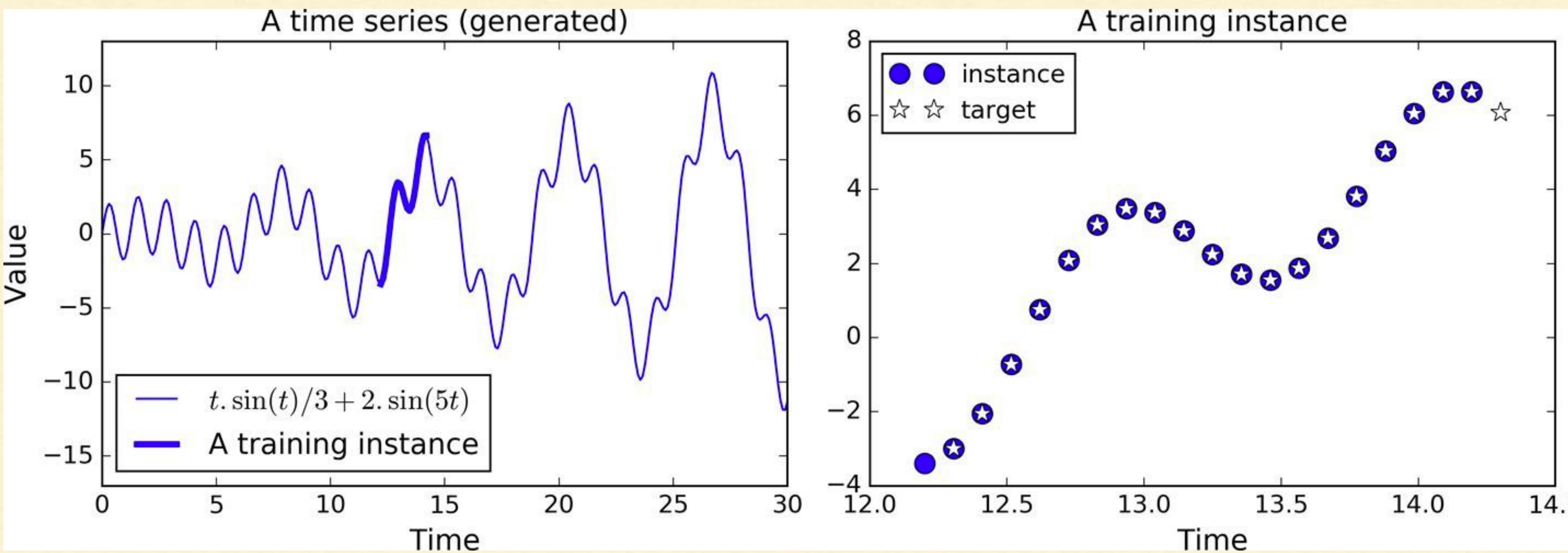
- We get over **98% accuracy** — not bad!
- Plus we would certainly get a better result by
    - Tuning the hyperparameters
    - Initializing the RNN weights using He initialization
    - Training longer
    - Or adding a bit of regularization e.g., dropout

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

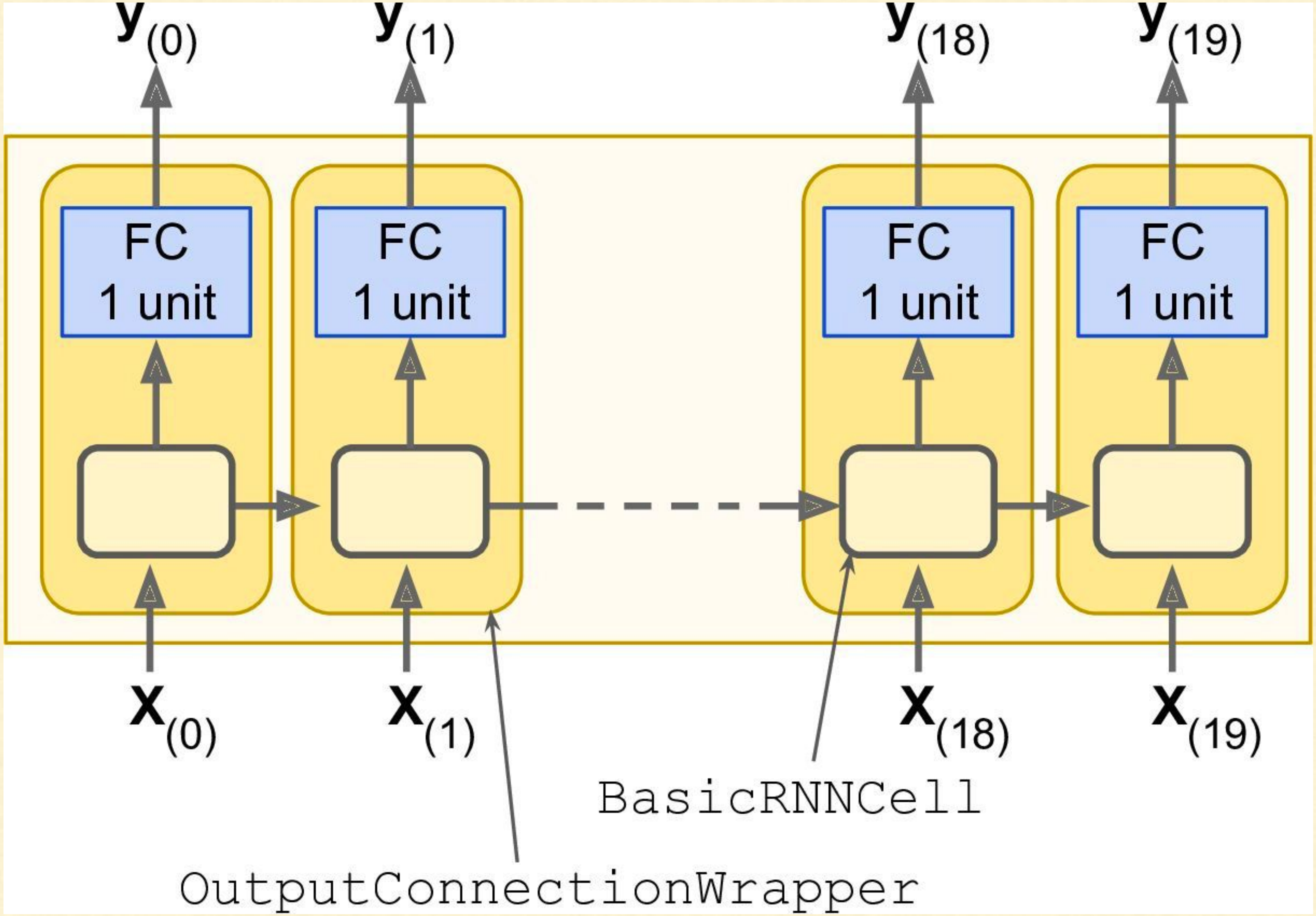
- At each time step we now have an output vector of size 100
- But what we actually want is a single output value at each time step
- The simplest solution is to wrap the cell in an **OutputProjectionWrapper**

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

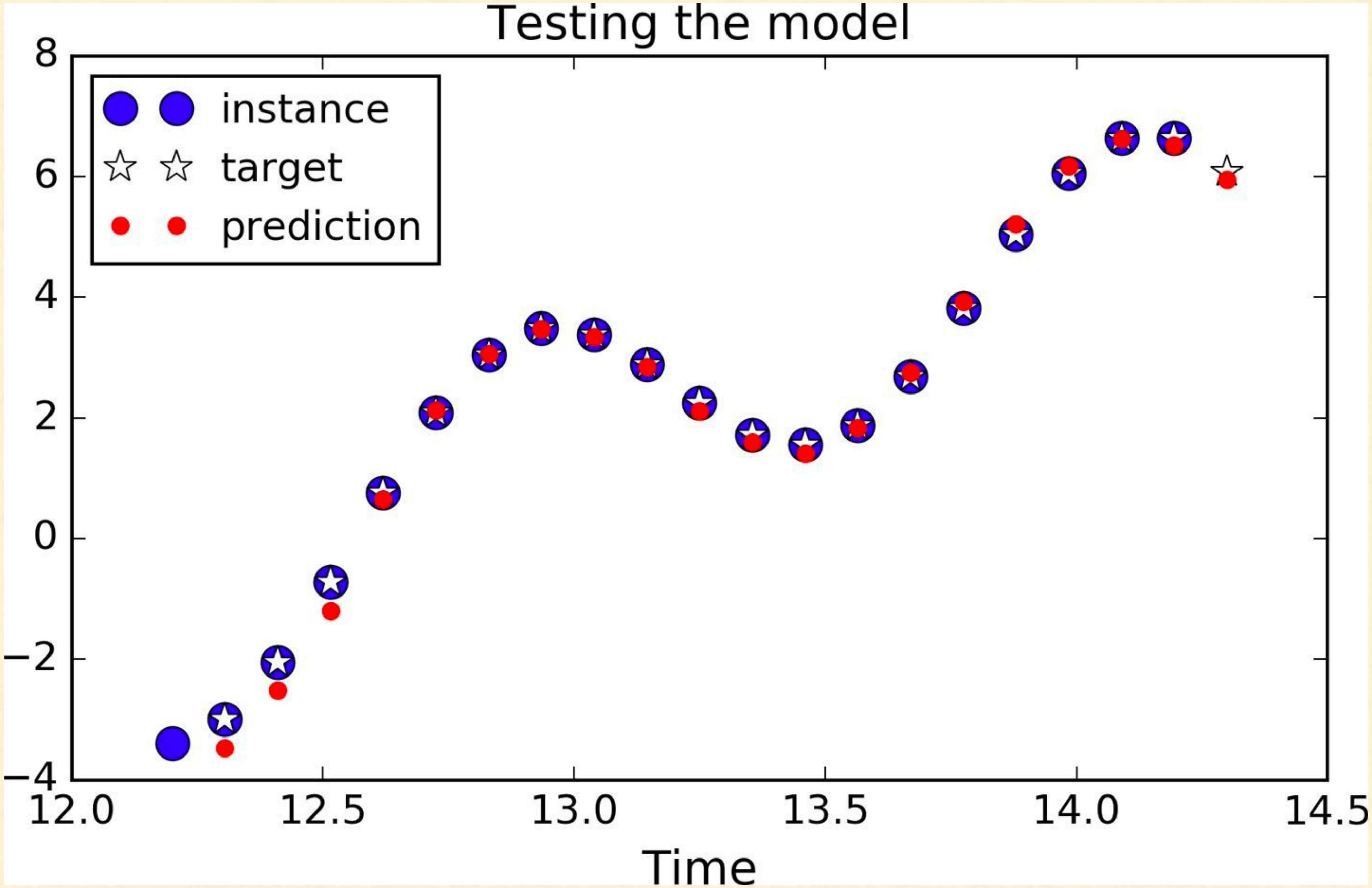
Making Predictions

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

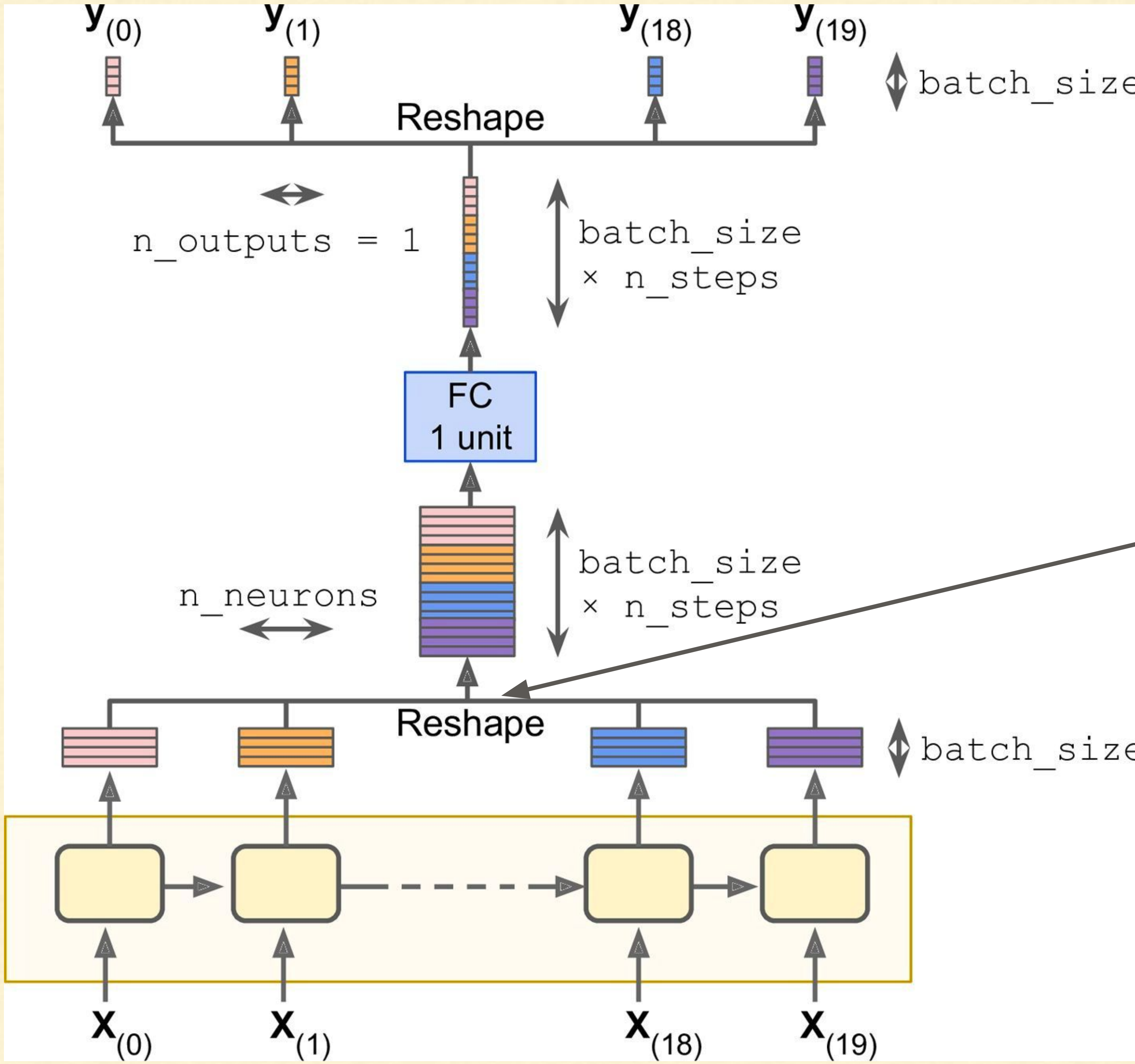
Training to Predict Time Series

- 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

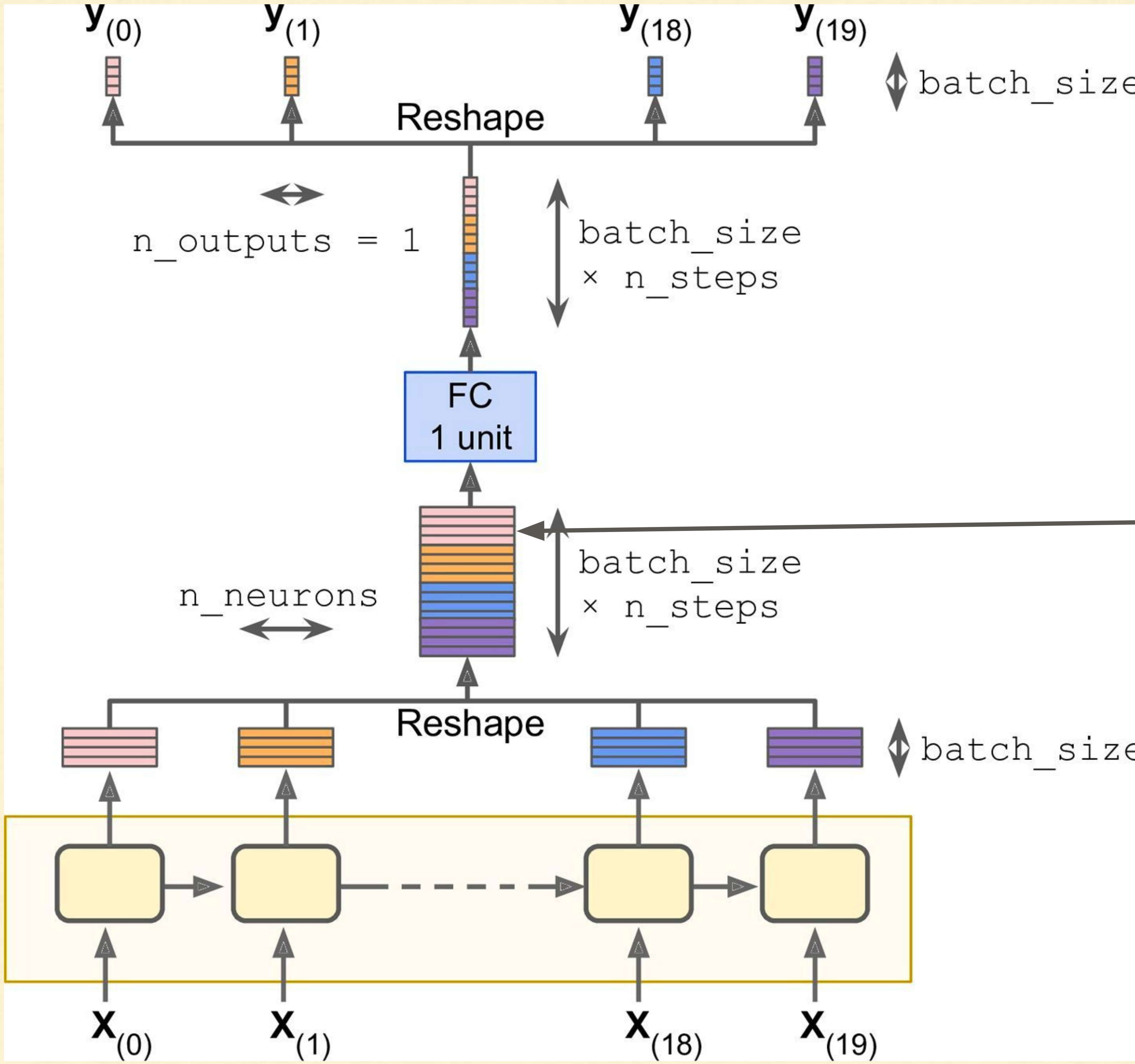
- There is a trickier but more efficient solution: you can reshape the RNN outputs from **[batch_size, n_steps, n_neurons]** to **[batch_size * n_steps, n_neurons]**
- Then apply a single fully connected layer with the appropriate output size in our case just **1**, which will result in an output tensor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



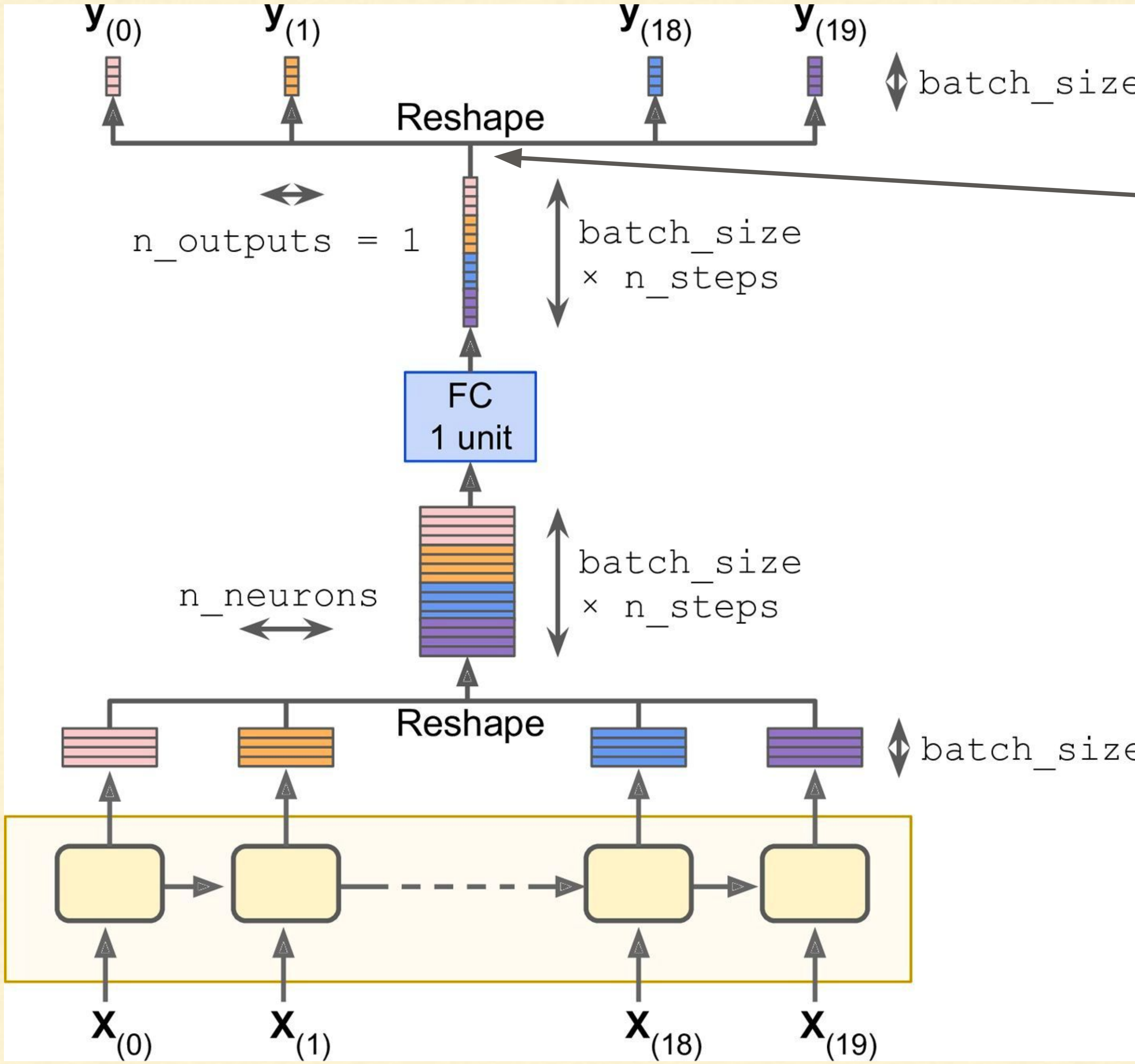
Reshape the RNN outputs from **[batch_size, n_steps, n_neurons]** to **[batch_size * n_steps, n_neurons]**

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

Deep RNNs

LSTM Cell

Peephole Connections

GRU Cell

Natural Language Processing

Word Embeddings

Machine Translation

Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com