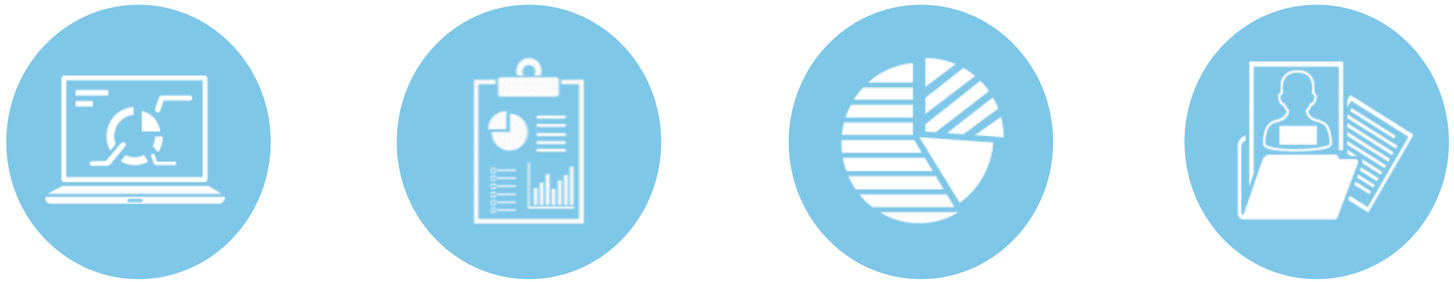
Deep Learning

Lesson 8—Recurrent Neural Networks



Learning Objectives

- Explore the meaning of Recurrent Neural Networks (RNN)
- Understand the working of recurrent neurons and their layers
- Interpret how memory cells of recurrent neurons interact
- Implement RNN in TensorFlow
- Demonstrate variable length input and output sequences
- Explore how to train recurrent neural networks
- Review time series predictions
- Describe training for Long Short Term Memory (LSTM) and Deep RNNs
- Analyze word embeddings

Recurrent Neural Networks

Topic 1: Meaning of Recurrent Neural Networks

Sequence Prediction

You predict based on the sequence or past experience all the time.

A fielder tries to predict where the cricket ball will land

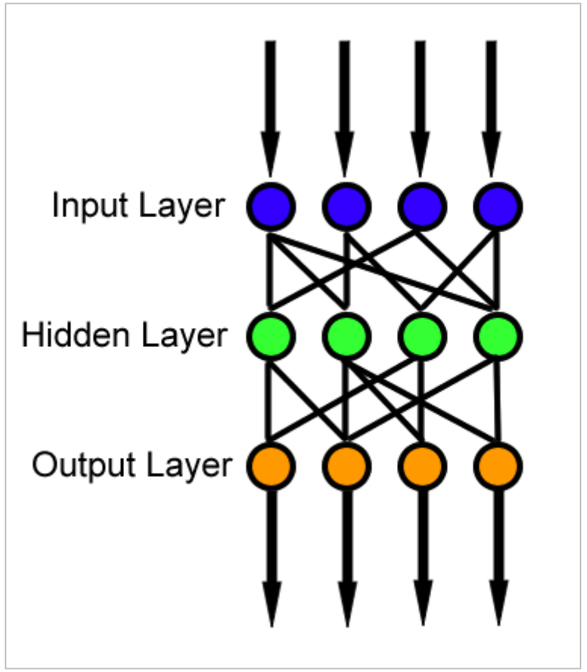
You finish your friend's sentence

Other examples include stock market predictions, an autonomous car trying to decide car trajectory to stay safe, or predicting words in a sentence.

Sequence Prediction in Deep Learning

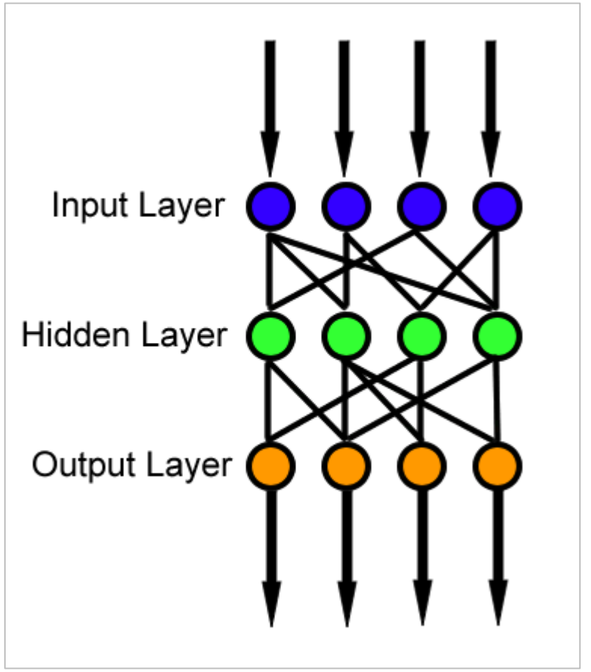
Let’s see if feedforward networks can be used for sequence prediction.

- Feedforward networks are widely used for image classification and statistical analysis.
- In these networks, connections between the units do not form a reverse circular loop.
- As the name suggests, information moves in one direction— from input nodes, through hidden nodes, to the output nodes.



Sequence Prediction in Deep Learning

- The limitation of feedforward networks is that the output of the complete network does not directly depend on the previous output of the same network. This means that the concept of "recurrence" is missing.
- They have separate parameters for each input feature.
- Thus the sequence of outputs are independent of each other, which might not be suitable for predictions of words in a sentence.
- Hence, Recurrent Neural Networks (RNN) prove to be a better choice for sequence prediction than feedforward networks.



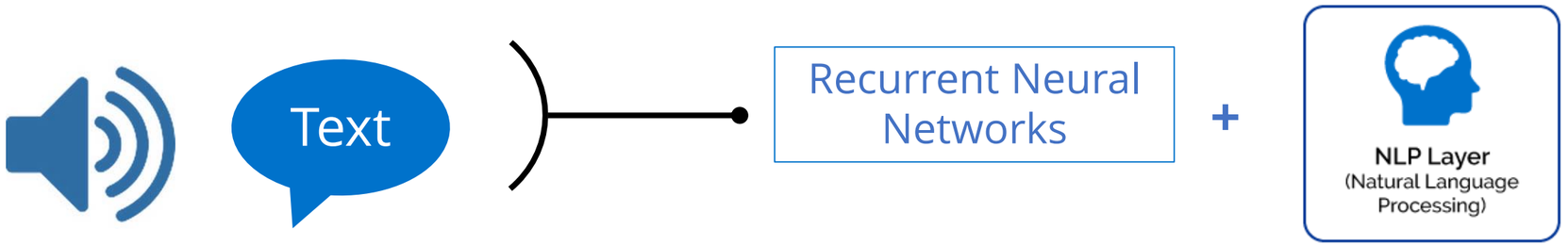
Recurrent Neural Networks: Definition

Recurrent neural networks are a class of artificial neural networks that create cycles in the network graph in order to exhibit dynamic temporal behavior. In simple words, RNNs involve recurrent or circular loops between the neurons, where the output of the network is fed back as an additional input to the network for subsequent processing.

Need for Recurrent Neural Networks

1. Recurrent Neural Networks can process sequences of large size as well as those with variable length.
2. It is based on sharing of parameters across different parts of the model.
3. Parameters share the same weights across several time steps.
4. Each member of the output is a function of previous member's output. It is produced using the same update rule that is applied to all previous outputs.
5. This recurrence imparts a memory to the network topology.

Use Cases: Translation, Speech-to-text, Sentiment Analysis

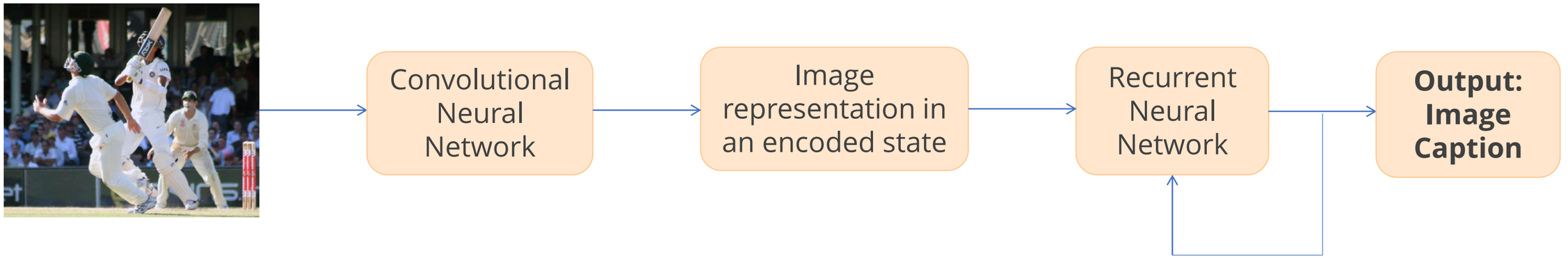


Sentences, documents, or audio files are taken as inputs.

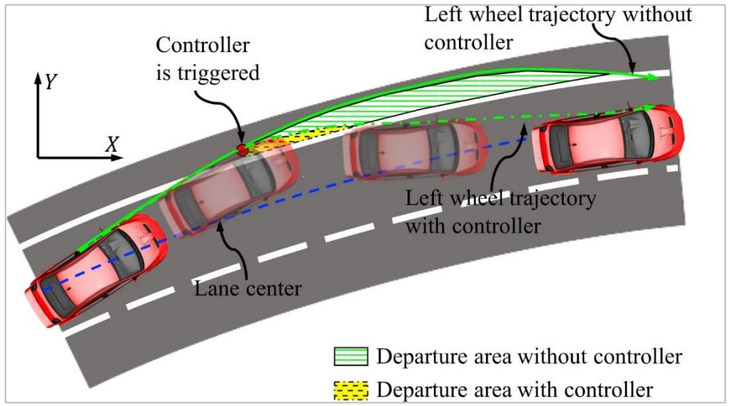
RNN then can take these inputs through NLP applications like translation, speech-to-text, or sentiment analysis.

Use Case: Image Captioning

RNNs can generate sentences, image captions, or even notes of a melody.



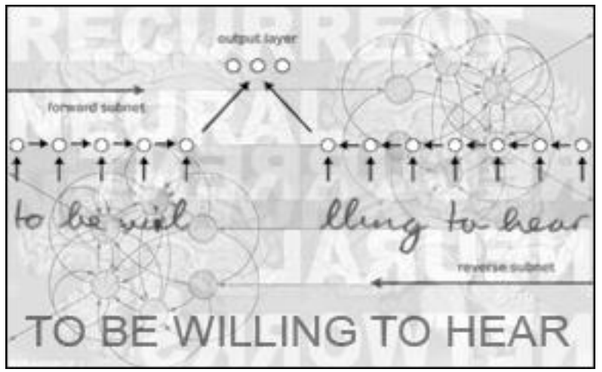
Some More Use Cases of RNN



Anticipating car trajectories to avoid accidents in autonomous cars

Analyzing time series data such as stock prices, and suggesting to brokers when to buy or sell stocks

Google's Magenta project lets you create art and music using RNN algorithms



Enabling handwriting and speech recognition

Recurrent Neural Networks

Topic 2: Layers of Recurrent Neurons

Recurrent Neurons

Meaning

- An RNN looks like a feedforward network except it has connections pointing backward.
- This tiny network can be plotted against the time axis as shown in the figure.
- The figure shows one neuron that receives inputs, produces an output, and sends that output back to itself.
- This is called unrolling the network through time.

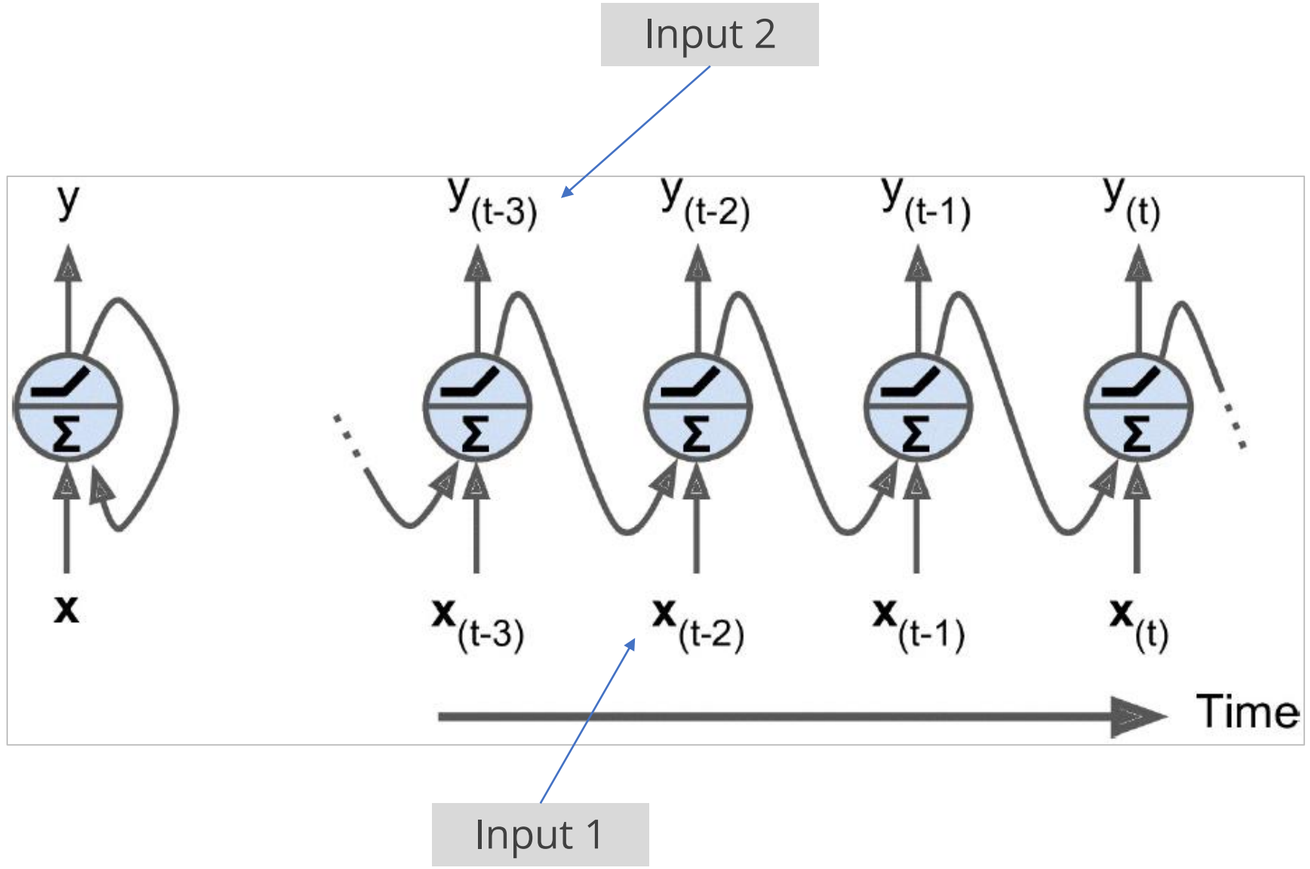


Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Recurrent Neurons

Understanding the Diagram

- ***t*** represents a time step also called frame.
- The recurrent neuron receives two inputs:
  1. **Input 1:** Input 1 is the actual input to the network at a particular time step. It is represented by x_t at each increment of ***t***.
  2. **Input 2:** Input 2 is the output of the previous time step fed as an additional input to the current time step. It is represented by y_{t-1}.

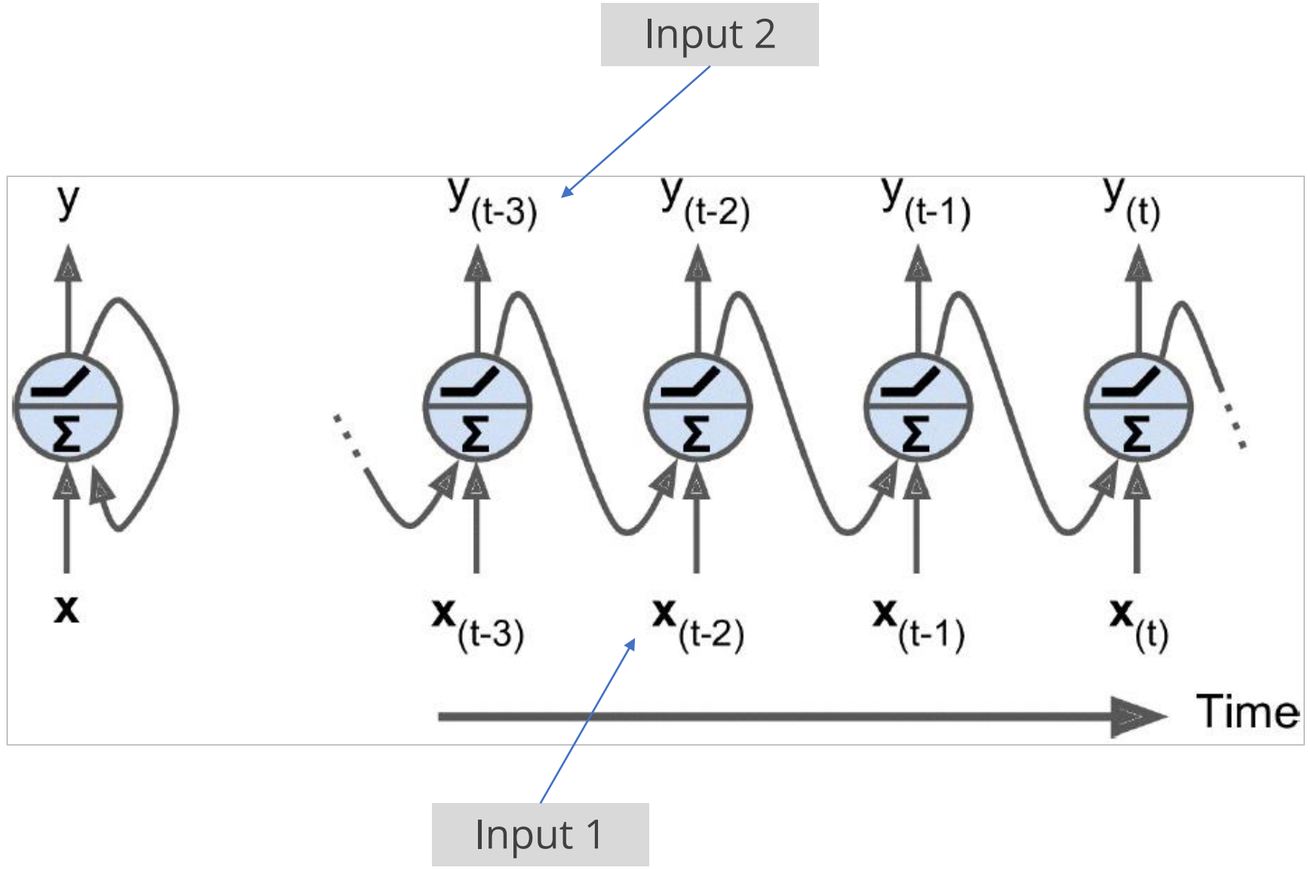
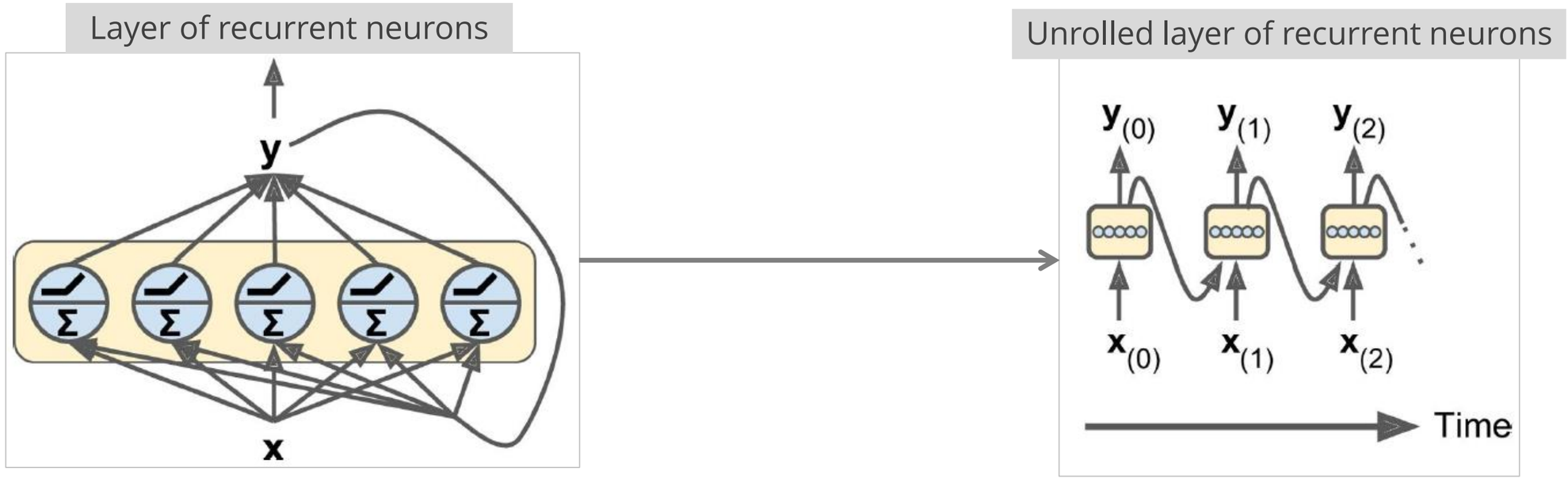


Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Layer of Recurrent Neurons



- Using the recurrent neurons, you can create a layer of recurrent neurons.
- Once unrolled, it looks like the image on the right.
- In the single recurrent neuron shown before, the input x was a vector and output y was a scalar.
- Here, in a layer of recurrent neurons, both **x** and **y** are vectors.

Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Adding Weights to Recurrent Neurons

- Each recurrent neuron has two sets of weights, one for the input x_t and the other for the outputs of the previous time step, y_{t-1}.
- Let's call these weight vectors $\mathbf{w}_x$ and $\mathbf{w}_y$.

- All the weight vectors can be placed into two weight matrices if the entire recurrent layer is considered instead of one recurrent neuron.
- These weight matrices can be represented as follows:

$\mathbf{W}_x$ $\mathbf{W}_y$

Computing Output for a Single Input

- The output vector of the recurrent layer containing a single neuron can be computed as follows:

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

Activation function

Bias vector

Computing Output for Mini-batch

- A recurrent layer's output can be computed for a whole mini-batch of data samples by placing all the inputs at time step ***t*** in an input matrix $\boldsymbol{X_t}$.

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

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

Computing Output for Mini-batch

n_{inputs} X $n_{neurons}$ matrix containing the connection weights for the inputs of current time step

$n_{neurons}$ X $n_{neurons}$ matrix containing the connection weights for the outputs of previous time step

Vector of size $n_{neurons}$ containing each neuron's bias terms

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

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

m X $n_{neurons}$ matrix containing the layer's outputs at time step **t** for each instance in the mini-batch

m - Number of instances in mini-batch
n_{inputs} **-** Number of input features
$n_{neurons}$ **-** Number of neurons

m X n_{inputs} matrix containing the inputs for all instances

Horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$

Weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ concatenated vertically into a single weight matrix **W** of shape $(n_{inputs} + n_{neurons})$ x $n_{neurons}$

Recurrent Neural Networks

Topic 3: Memory Cells of Recurrent Neurons

Memory Cells

- A recurrent neuron is called a ***memory cell*** owing to the fact that this neuron tends to preserve memory across multiple time steps.
- A recurrent neuron has memory. This is evident from the fact that its output at time step ***t*** is a function of all inputs from previous time steps.
- The part of neural network that preserves some state across time steps is referred to as a memory cell.

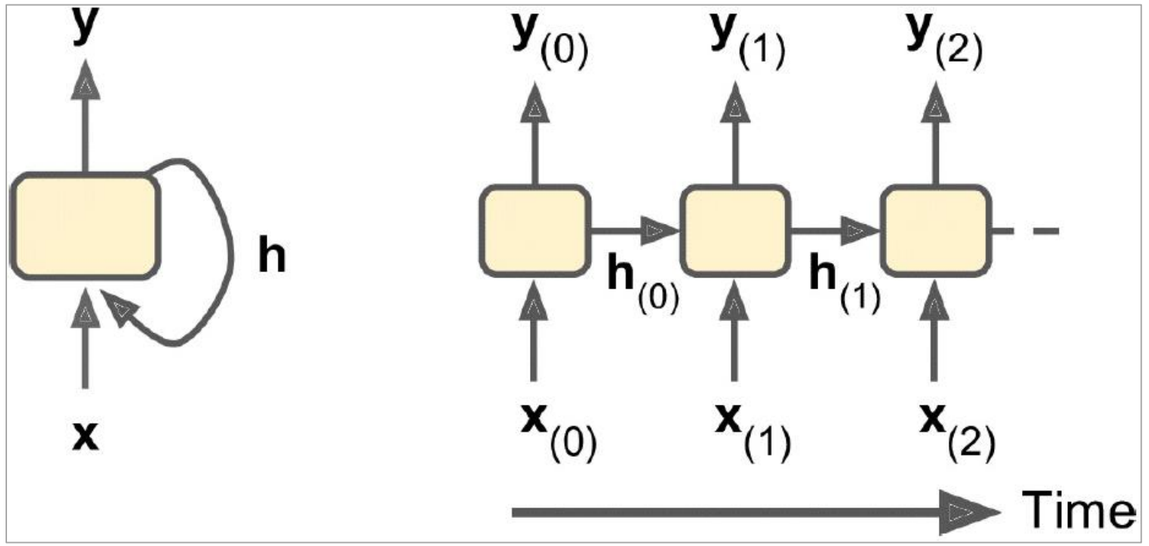


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Hidden State of Memory Cells

- A cell's hidden state can be different from its output as shown in the image. ***y(t)*** is different than ***h(t)***. At other times, ***y(t)*** and ***h(t)*** can be same.
- A cell's state at time step ***t***, denoted by ***h(t)***, is a function of some inputs at that time step and its state at the previous time step:

$$h_{(t)} = f(h_{(t-1)}, x_{(t)})$$

- The output ***y*** at time step ***t*** is also a function of all previous states and the current input.

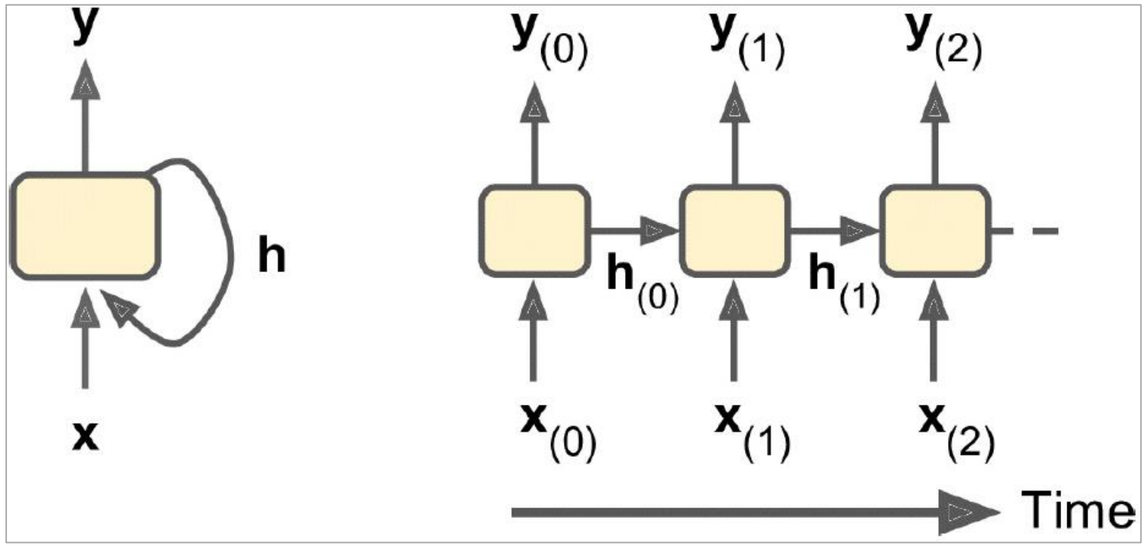


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Various Configurations of RNNs

RNNs can be arranged in multiple configurations to serve various purposes.

Sequence to Sequence

- Here, you feed a sequence of inputs to the RNN, and you get a sequence of outputs. For example, RNN is used for time series predictions such as stock price.
- In this case, the input is stock price pattern until step ***t*** and output is a time step shifted into the future.

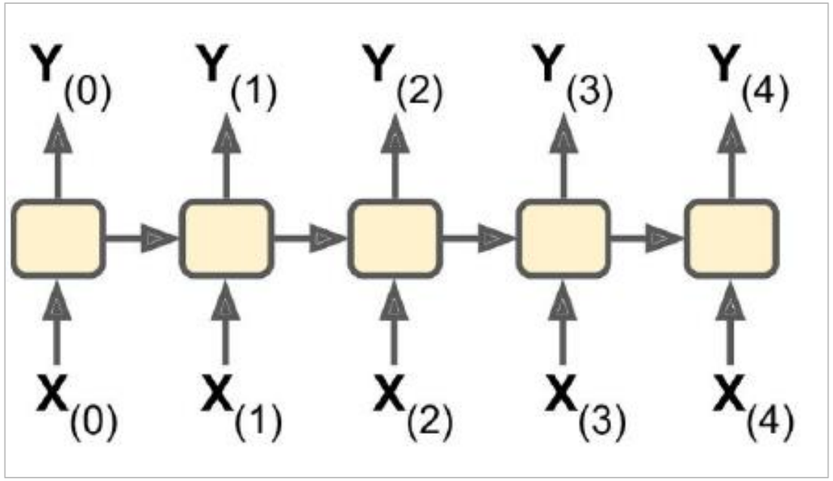


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Various Configurations of RNNs

Sequence to Vector

- Here, you take a sequence of inputs and generate just a single output; for example, sentiment score (+1 or -1) for a movie review.

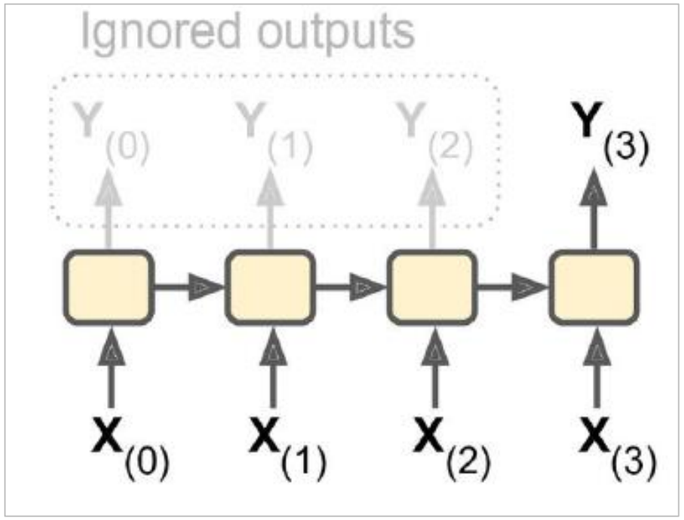


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Various Configurations of RNNs

Vector to Sequence

- Here, the RNN takes one input and produces a sequence of outputs; for example, producing a caption for an image.

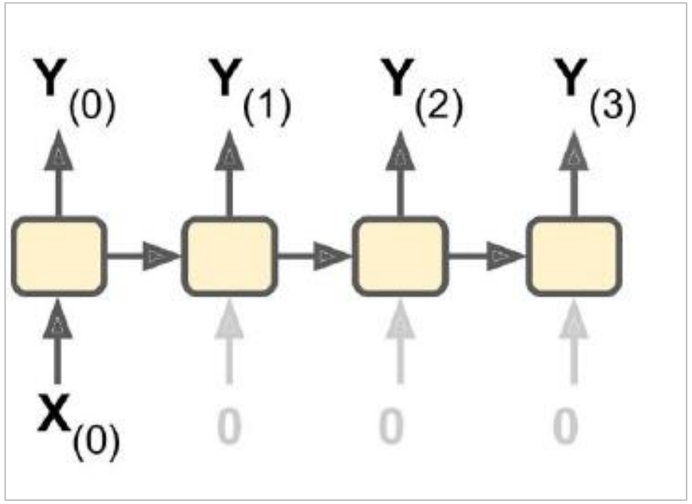


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Various Configurations of RNNs

Sequence to Sequence

- This type has two networks embedded together.
- The first RNN network takes a sequence of inputs to produce an output vector.
- This output vector is fed to the second RNN that produces a sequence of outputs.
- The first box in the diagram that is Encoder is sequence to vector and Decoder is vector to sequence; for example, language translation.

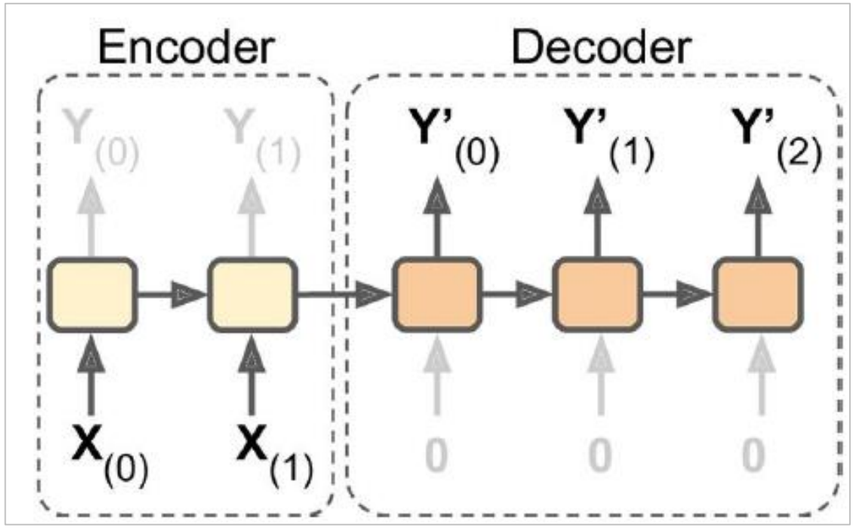
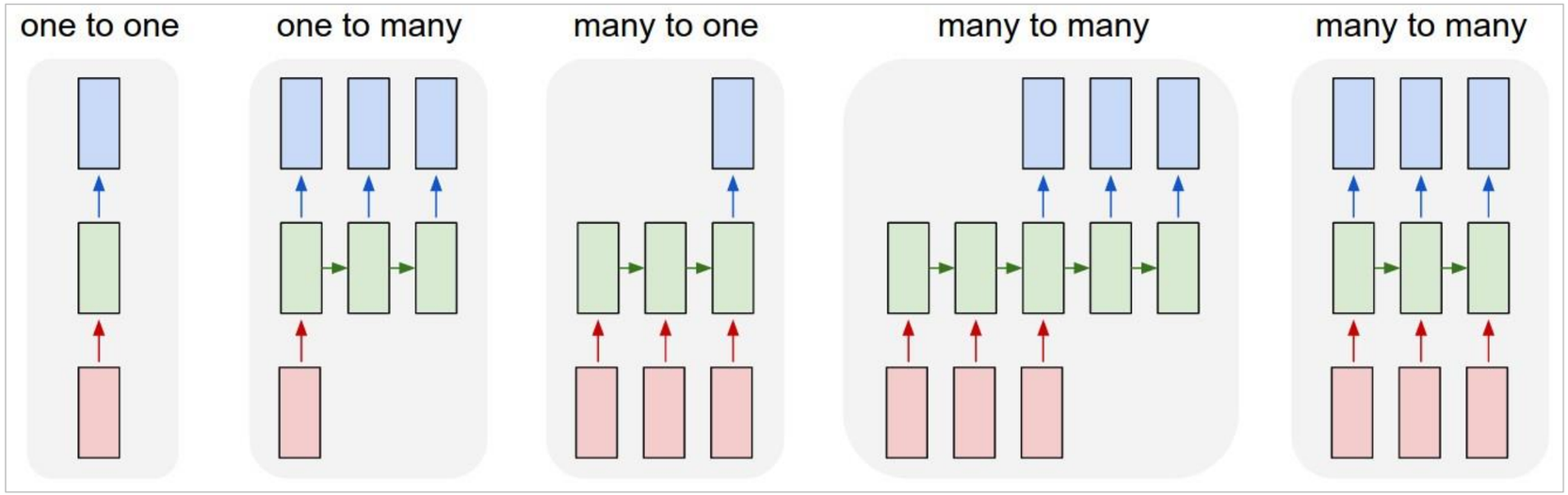


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Various Configurations of RNNs

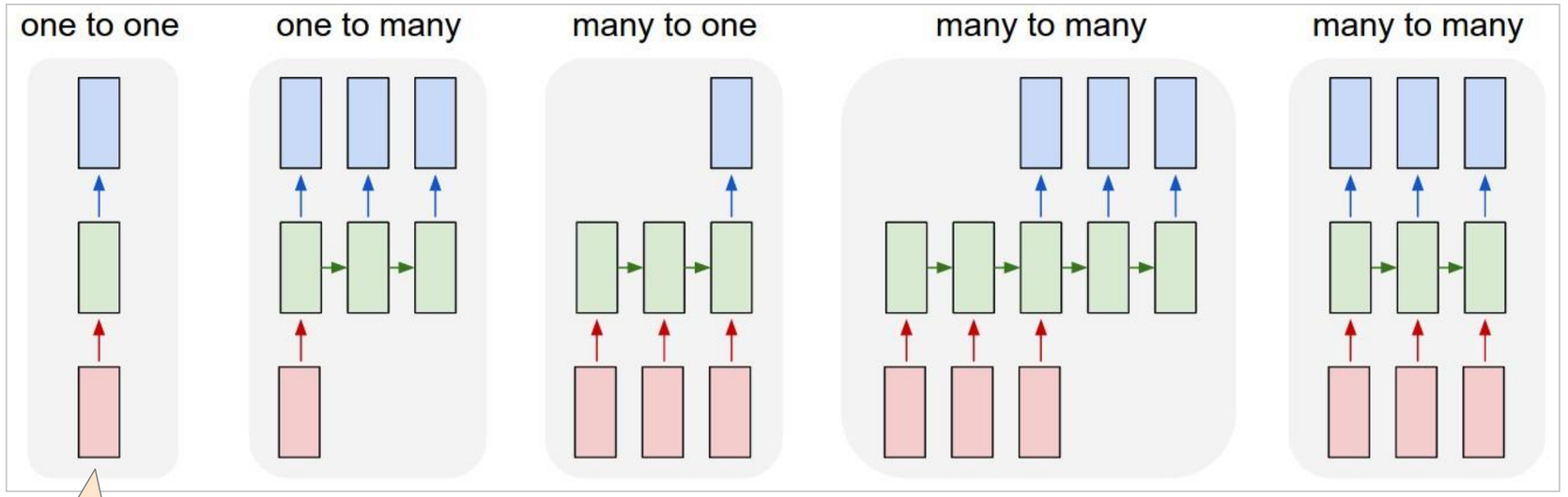
To summarize, here is a high-level view of various RNN configurations:



Rectangles – Represent vectors
Arrows – Represent functions
Red Rectangles – Input vectors
Blue Rectangles – Output vectors
Green Rectangles – RNN's state

Image Source: karpathy.github.io

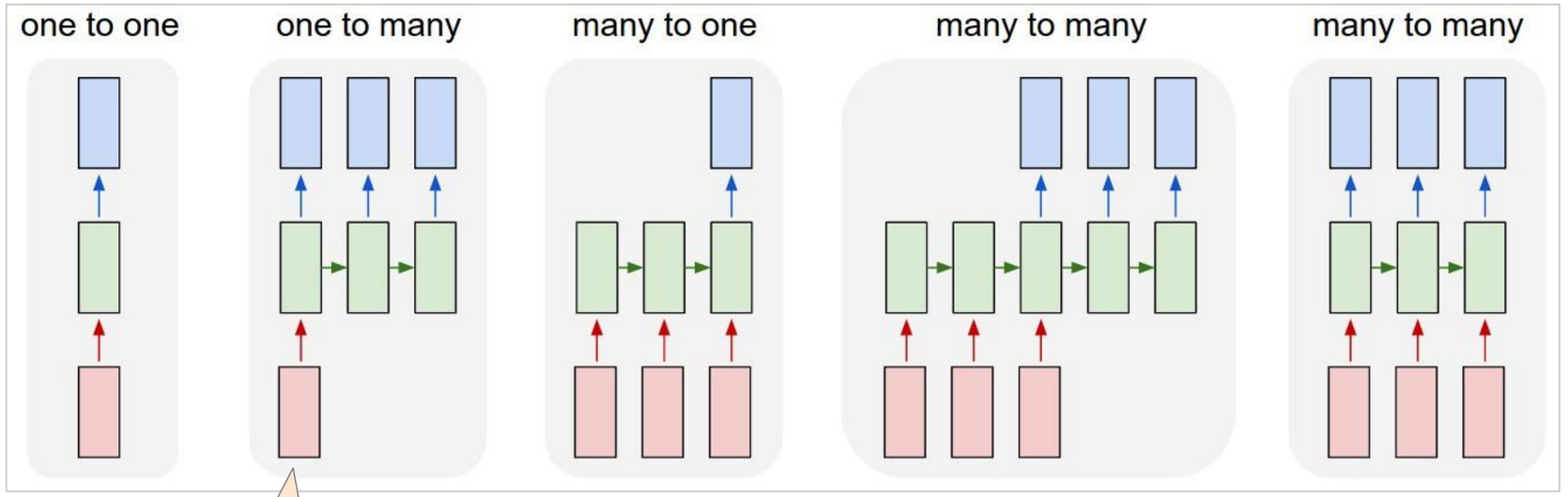
Various Configurations of RNNs



Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output

For example: Image classification

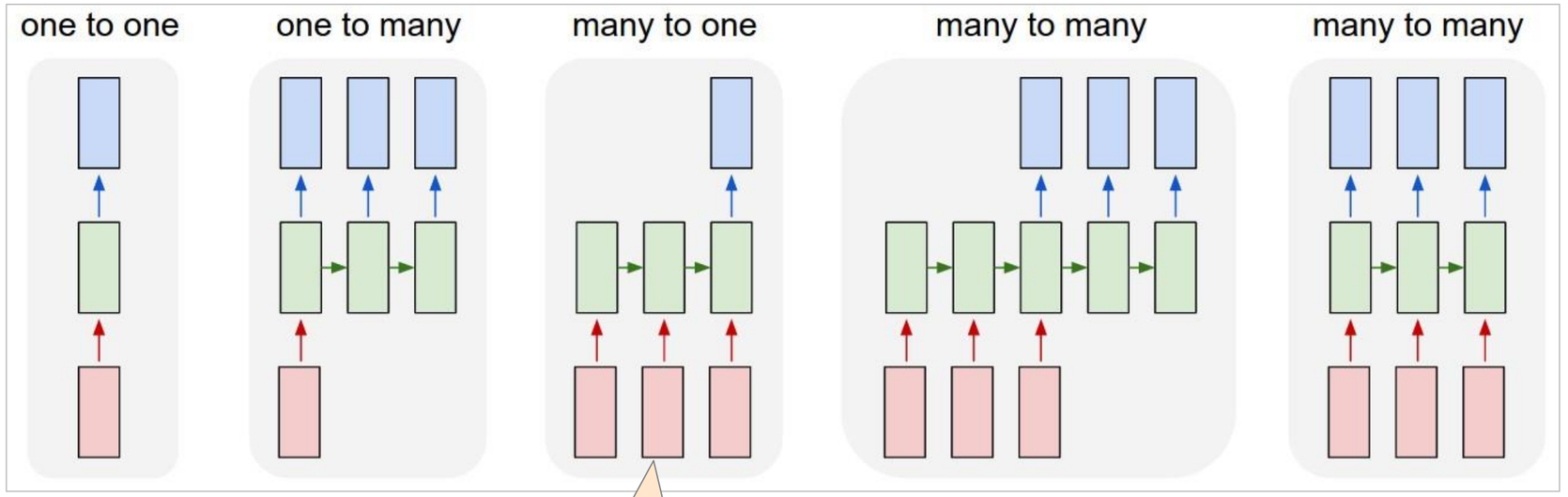
Various Configurations of RNNs



Sequence output

For example: Image captioning takes an image and outputs a sentence of words

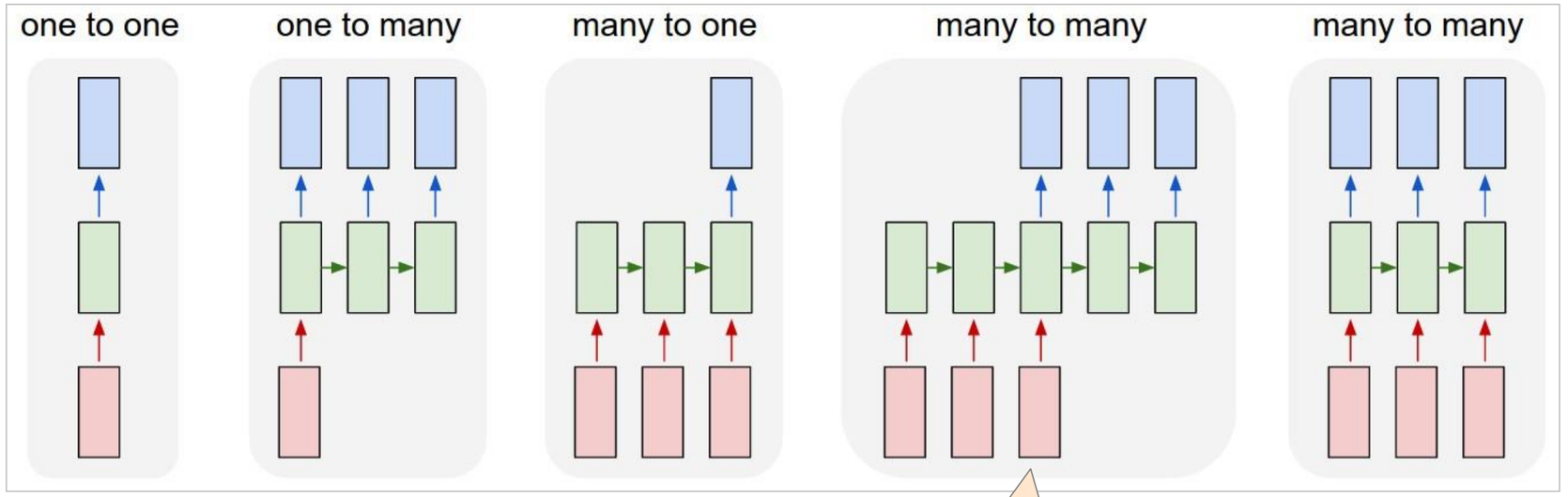
Various Configurations of RNNs



Sequence input

For example: Sentiment analysis, where a given sentence is classified as expressing positive or negative sentiment

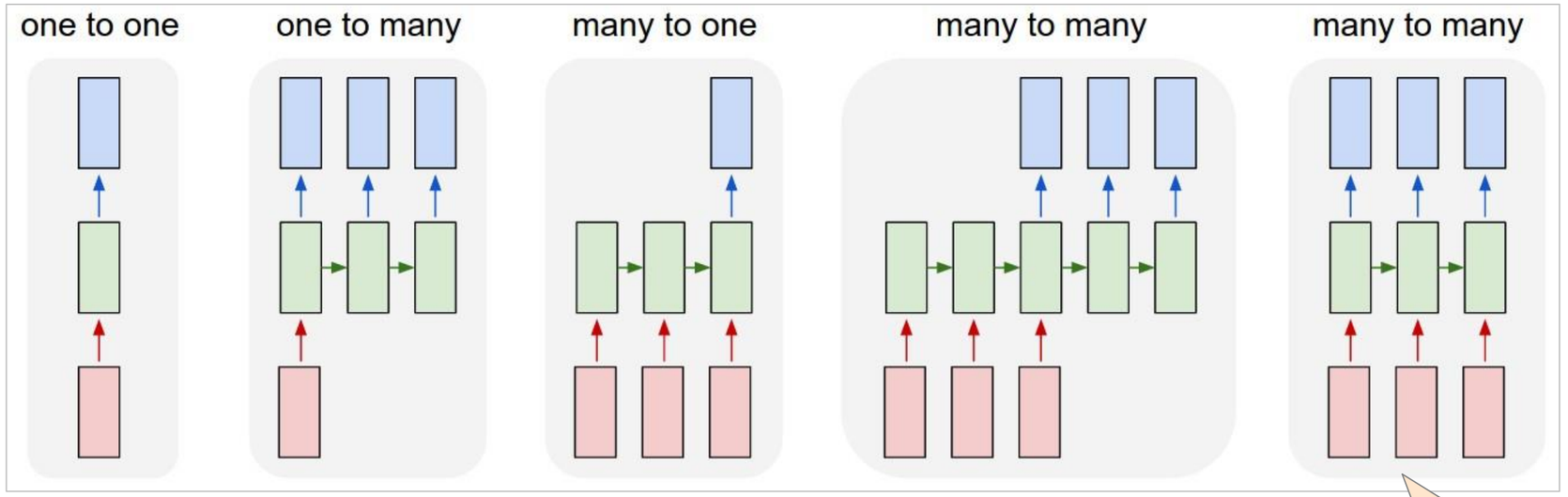
Various Configurations of RNNs



Sequence input and sequence output

For example: Machine translation, where an RNN reads a sentence in English and then outputs a sentence in French

Various Configurations of RNNs



Synced sequence input and output
For example: Video classification, where you wish to label each frame of the video

Recurrent Neural Networks

Topic 3: RNN in TensorFlow

Basic RNN with TensorFlow

Develop a Simple RNN Model

- The code given below builds a basic RNN (without using RNN operations of TensorFlow) with two time steps.
- At each time step, there are five recurrent neurons and the input is a vector with three units.
- You use **tanh** activation function at each time step.

```
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()
```

Basic RNN with TensorFlow

Develop a Simple RNN Model

- The network on the previous slide looks like a regular two layer feedforward network with the following differences:
  - Weights are shared by both layers
  - You feed inputs at each layer and get output at each layer
- Let's feed a mini-batch of inputs, which consists of the input vectors at both time steps.
- Each input would have four data instances and you get outputs at both time steps.

```
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})
```

Basic RNN with TensorFlow

Develop a Simple RNN Model

- The output contains vectors for entire mini-batch for all samples.
- Since n_neurons per layer was five, you get an output vector containing five output values per data instance.

```
>>> print(Y0_val)  # output at t = 0
[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]  # instance 0
 [-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]  # instance 1
 [ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]  # instance 2
 [ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]] # instance 3
>>> print(Y1_val)  # output at t = 1
[[ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]  # instance 0
 [-0.70553327 -0.11918639  0.48885304  0.08917919 -0.26579669]  # instance 1
 [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]  # instance 2
 [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]] # instance 3
```

- For larger number of time steps like 100 steps, the graph gets very big.
- This can be simplified by using TensorFlow RNN operations.

Basic RNN Cell

Definition

A Basic RNN Cell in TensorFlow is the basic RNN unit which when unrolled creates copies of the cell for many time steps.

Implement Basic RNN Cell with TensorFlow

- The function static_rnn creates an unrolled network by chaining basic cells as shown in the code given below.
- The static_rnn function returns:
  - **output_seq** – List of output tensors for every time step
  - **states** – Tensor containing final states of the network
- For basic cell, a state is same as output of previous time step. For example, $y_i=h_i$

```
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
```

Implement Basic RNN Cell with TensorFlow

- The following more efficient code defines input of shape [None, n_steps, n_inputs], where first parameter is mini-batch size.
- The ***unstack*** command extracts input sequences for each time step, that is, *X_seqs* which has shape [*None, n_inputs*].
- ***transpose*** command is used to swap first two dimensions, that is, n_steps and mini-batch size.
- The rest of the code stays the same as before.
- Final line here stacks the output tensors into a single tensor of shape [*None, n_steps, n_inputs*].

```
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])
```

Implement Basic RNN Cell with TensorFlow

- As evident in the following code, you feed a mini-batch as before.

```
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})
```

Dynamic Unrolling to Simplify the Logic and Avoid OOM Errors

- The output of the code is:

```
>>> 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]]]
```

- So far you were still building one cell per time step.
- The graph can turn out to be complex for 50 time steps.
- This can lead to Out of Memory (OOM) errors during backpropagation, since the memory must store all output tensors in feedforward pass to calculate gradients during the reverse pass.
- This is solved by dymanic_rnn function.

Dynamic Unrolling to Simplify the Logic and Avoid OOM Errors

- The ***dynamic_rnn*** function accepts a tensor for all inputs at every time step of shape [None, n_steps, n_inputs], and it outputs a single tensor for all outputs at each time step of shape [None, n_steps, n_inputs].
- There is no need to stack, unstack, or transpose.
- The following code is cleaner and creates the same network as before.

```
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)
```

Demo 1

Develop RNN to demonstrate sequence data processing

- **Objective:** Develop RNN to demonstrate sequence data processing

- **Steps:**
  1. Build an RNN with two time steps, taking input vectors of size 3 at each step, with 5 neurons per cell
  2. Run this RNN with some sample data in the code itself
  3. Print the output at each time step of the RNN
  4. Demonstrate the use of Basic RNN Cell and Dynamic RNN Cell
  5. Print the output for various time steps of the RNN

- **Skills required:** Knowledge of RNNs with unfolded computation graph, number of time steps, basic RNN cells, and dynamic RNNs

- **Dataset:** Sample data is generated within the code

- **Components of the code to be executed:** Run the tutorial titled “Recurrent Neural Networks”

Recurrent Neural Networks

Topic 4: Handling Sequences of Variable Lengths

Variable Length Input Sequences

- Instead of fixed-sized input sequence, one could have variable length input sequence.
- The length of input sequence is provided by ***sequence_length*** parameter.

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

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

Variable Length Input Sequences

- If the length of one input sequence is shorter than others, you can zero-pad it so that it fits the input tensor. *(Zero padding means filling up with zeros to make the length of the element consistent with those of the other elements).*

```
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])
```

2 elements in data instance 0,
1 element in data instance 1,
2 elements in data instance 2,
2 elements in data instance 3

```
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})
```

Variable Length Input Sequences

- The output tensor is shown in the output below.
- The outputs are zero vector if input sequence is zero input.

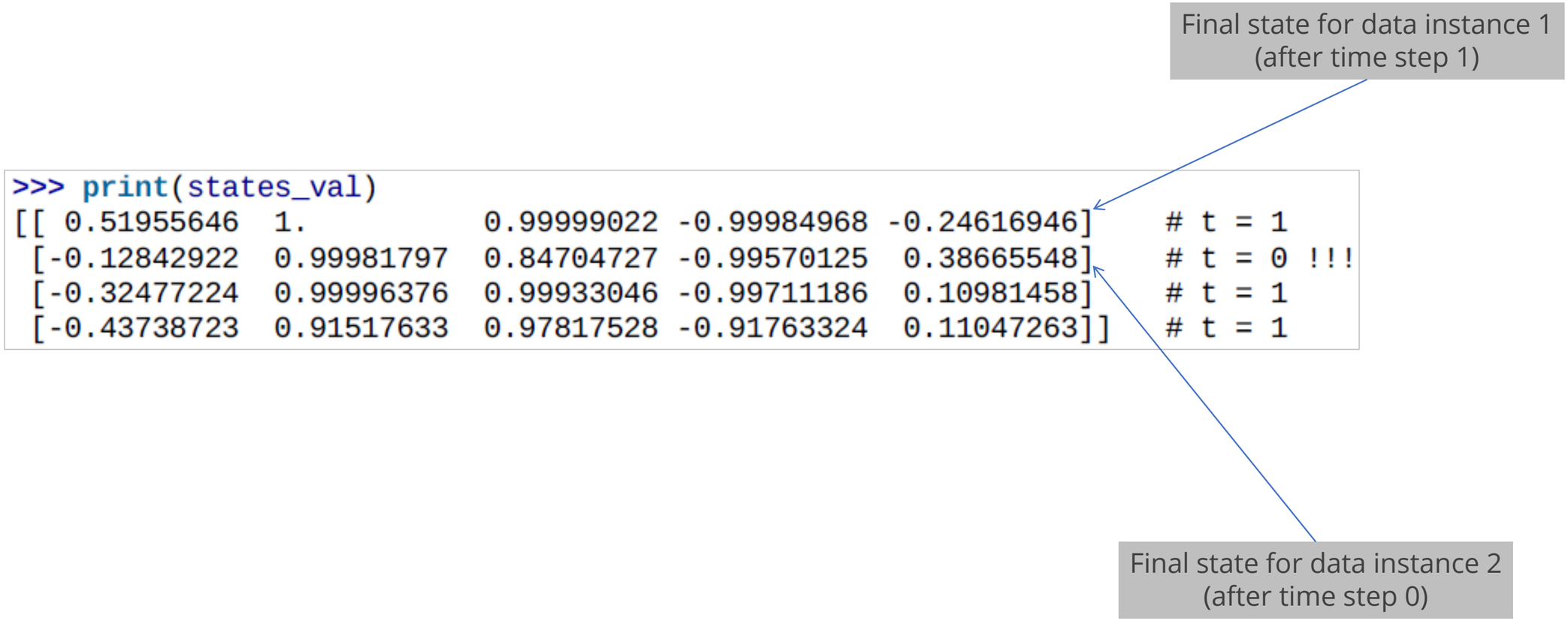
```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]  # final state

 [[-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
```

Output for data instance 1

Output for data instance 2

Variable Length Input Sequences

- The ***states*** tensor (which contains final state of each cell) looks as follows:



Variable Length Output Sequences

- The ***sequence_length*** parameter can be set as before if the output sequence has same length as input sequence.
- If output length sequence varies as in language translation model, you can define a special output called an ***end-of-sequence token*** (EOS token), which will mark the end of output sequence.

Recurrent Neural Networks

Topic 5: Training Recurrent Neural Networks

Backpropagation through Time

- To train an RNN, unroll it through time and then apply backpropagation. This is called ***backpropagation through time (BPTT).***

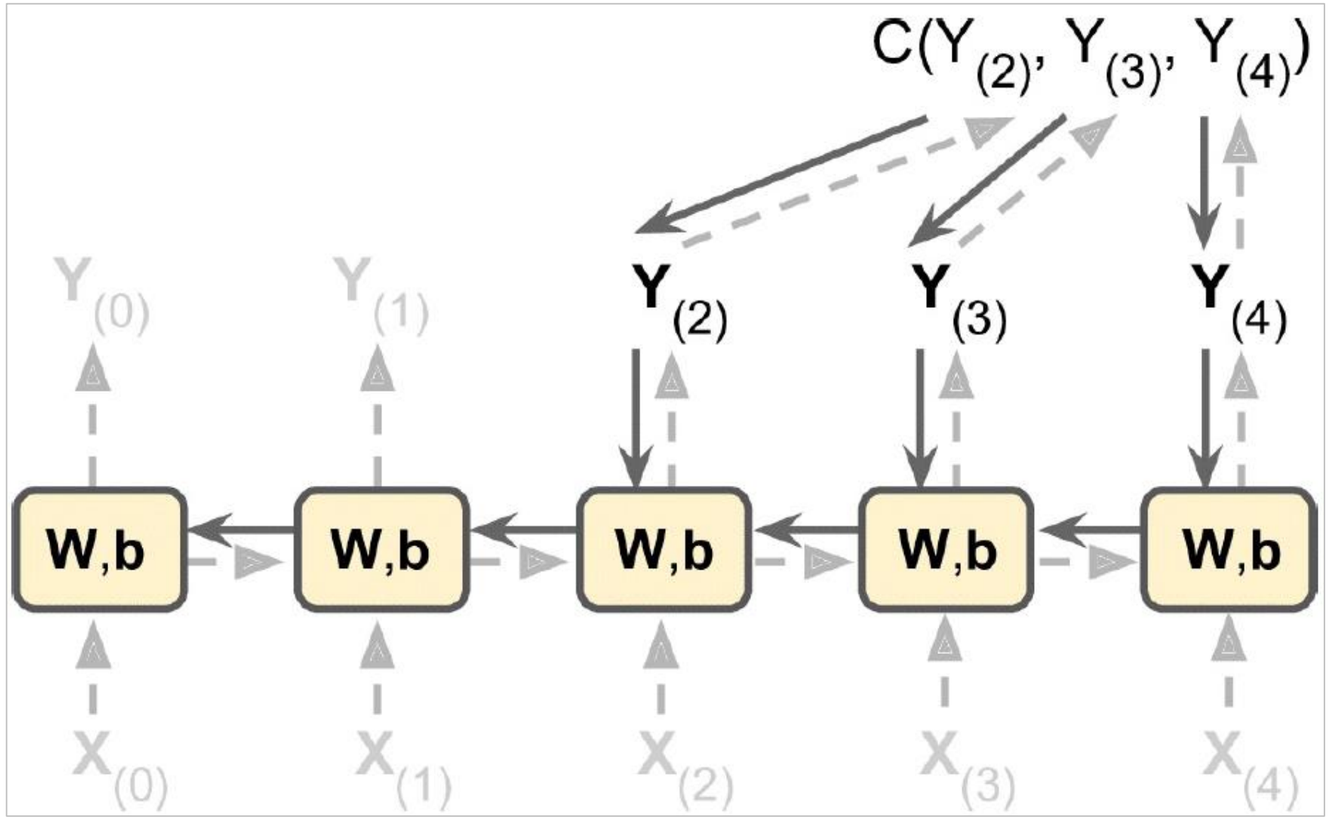


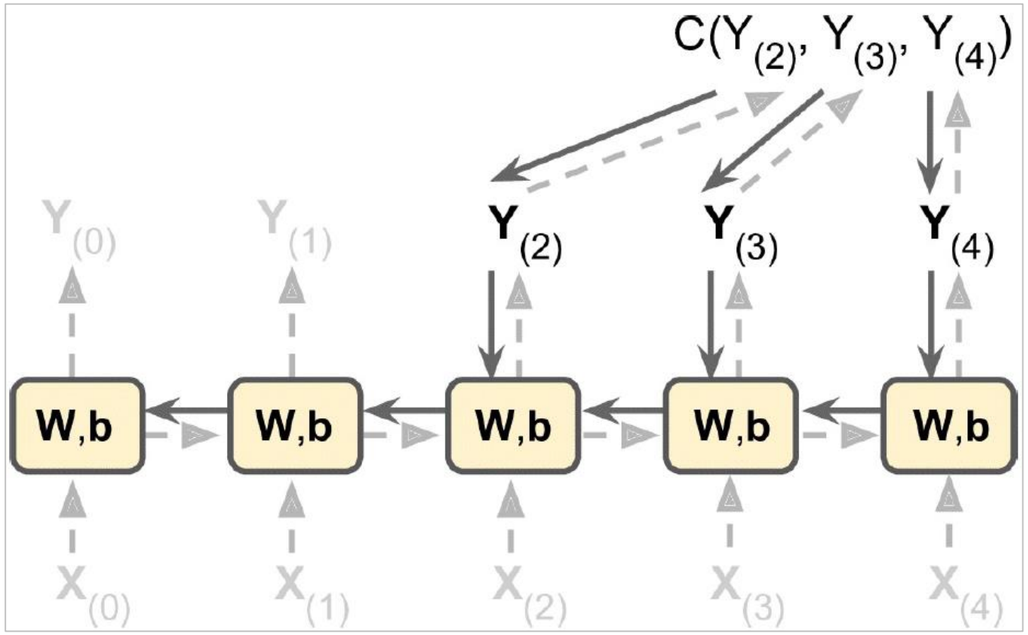
Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Training Recurrent Neural Networks

- First, there is a forward pass through the unrolled network (dashed arrows).
- Then, the output sequence is evaluated using a cost function:

  $C(Y_{t\text{-}min}, Y_{t\text{-}min}+1, \ldots, Y_{t\text{-}max})$

- The sequence starts with the t-min time step, and the outputs at some time steps may be ignored.
- Then the gradients of the unrolled network are propagated backward as shown by solid arrows.
- Finally, the model parameters are updated using the computed gradients.
- Note that the gradients flow backward through all the outputs used by the cost function.



Training a Sequence Classifier

MNIST Classifier

- Let’s use RNNs to train an MNIST classifier.
- Each handwritten digit image is 28 by 28 pixels, which is treated as 28 rows of 28 pixels each for purposes of RNN.
- You can use cells of 150 recurrent neurons.
- Finally, there is a fully connected layer with 10 neurons (for 10 classes of digits), followed by a SoftMax layer for classification.

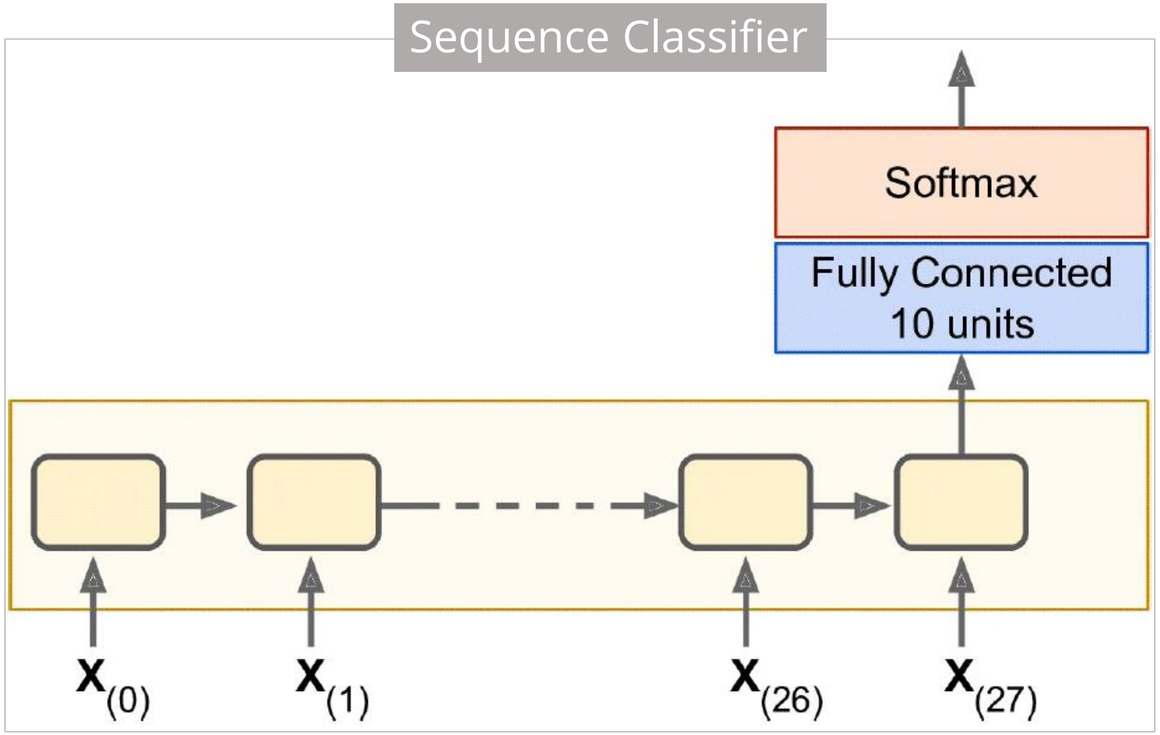


Image Source: “Hands-on Machine Learning with Scikit-Learn and TensorFlow ” by Aurelien Geron

Training a Sequence Classifier

- The construction phase of a sequence classifier is similar to a regular feedforward network, except the fact that the unrolled RNN replaces the hidden layers.
- Notice that the fully connected final layer is connected to the final state tensor *states*, which contains the 28th or the final output.

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)

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()
```

Training a Sequence Classifier

Reshaping Test Data

- Now, let's load MNIST data and reshape the test data to [batch_size, n_steps, n_inputs].

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels
```

Training a Sequence Classifier

Execution Phase

- The execution phase is similar to a regular neural network, except that you 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)
```

Training a Sequence Classifier

Output

- The output is:

```
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
```

- The output demonstrates an accuracy of 98%.
- You can tune it further by adjusting the hyperparameters, adding more epochs, or using He initialization or regularization (for example, dropout).

Recurrent Neural Networks

Topic 6: Time Series Predictions

Time Series Predictions

- You can train an RNN to predict time series data such as stock prices, house values, air temperature, or even brain wave patterns.
- Let's develop a time series RNN where each training instance is randomly selected sequence of 20 consecutive values.
- The target sequence is the same data shifted one step forward in time.

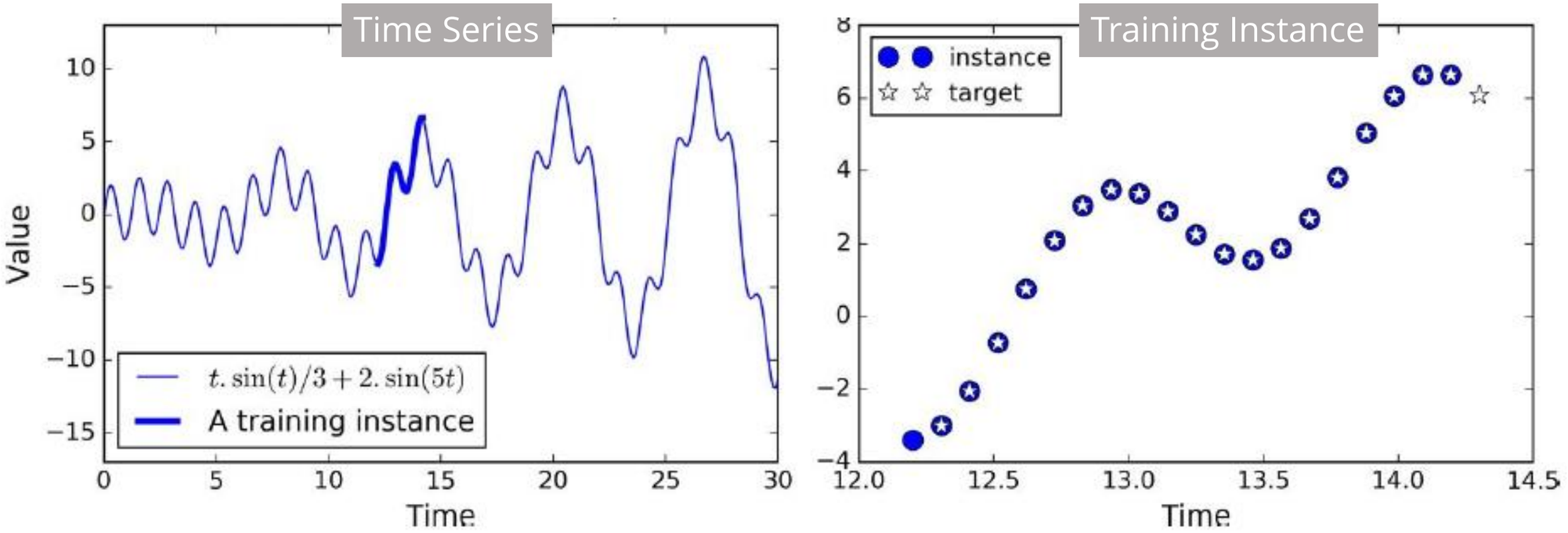


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Time Series Predictions

Training

Let's create the RNN:

```
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)
```

- The number of time steps is 20 since there are 20 points in each input sample.
- Each input has only one feature.
- The target is also a sequence of 20 values, with stock prices shifted one step into the future.
- In practice, for a model like stock market prediction, you might have more than one feature. For example, each input might have not only stock price but also price of competing stocks, analyst's ratings, overall market cap of the entire market, etc.

Time Series Predictions

Training

- At each time step, there is an output vector of size 100.
- But you want only one output value at each step.
- The simplest solution for this is an ***OutputProjectionWrapper***.
- This adds a fully connected layer for each time step output (linear neurons without any activation function).
- The hidden states are not impacted.
- All the fully connected layers share the same trainable weights and bias terms.

RNN Cells using Output Projection

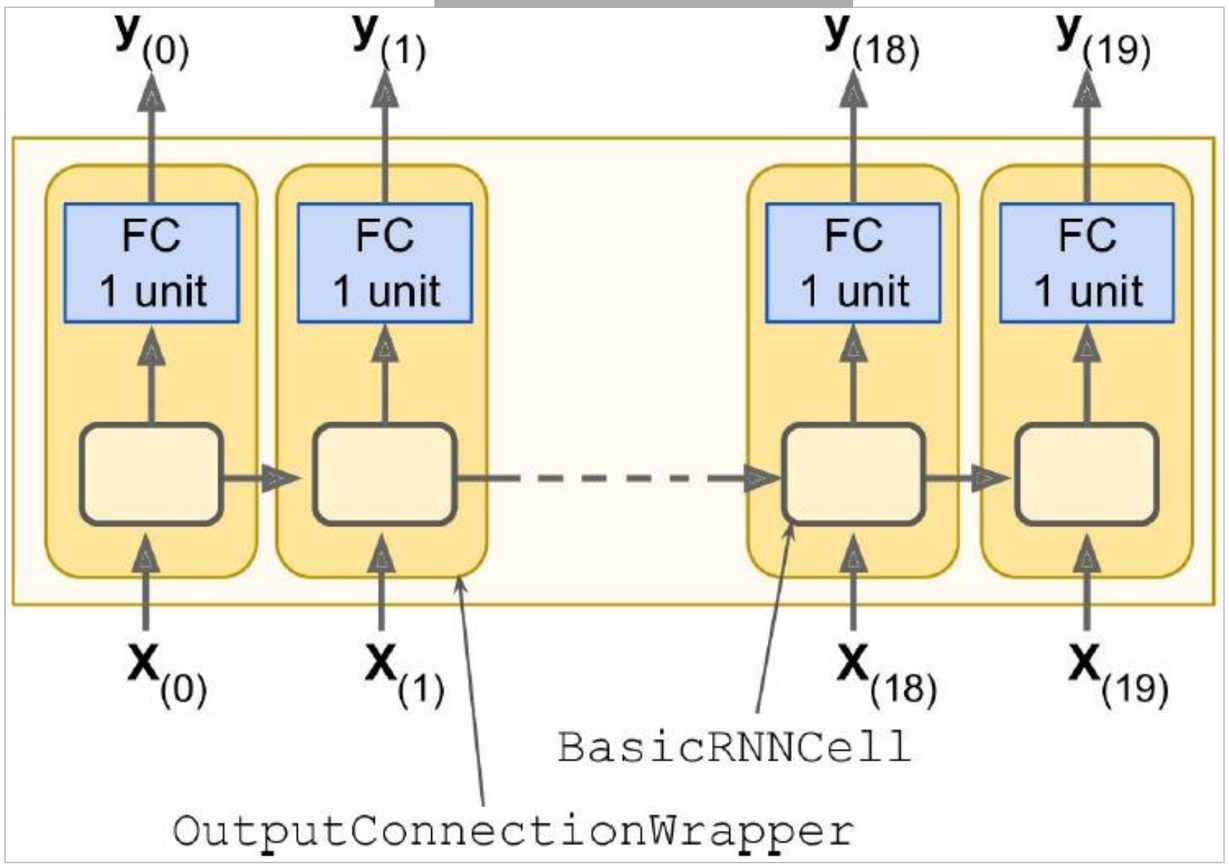


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Time Series Predictions

Training

The code to apply an ***OutputProjectionWrapper:***

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

You define the cost function using ***MSE*** and ***AdamOptimizer***:

```
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()
```

Time Series Predictions

Execution

Then comes the execution phase of the time series:

```
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)
```

Time Series Predictions

Output

The output of the program is as follows:

```
0     MSE: 379.586
100   MSE: 14.58426
200   MSE: 7.14066
300   MSE: 3.98528
400   MSE: 2.00254
[...]
```

Time Series Predictions

Predictions

Once the model is trained, one can make predictions:

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

Time Series Predictions

Testing the Model

The figure given below shows the predicted sequence for the instance after 1000 training iterations:

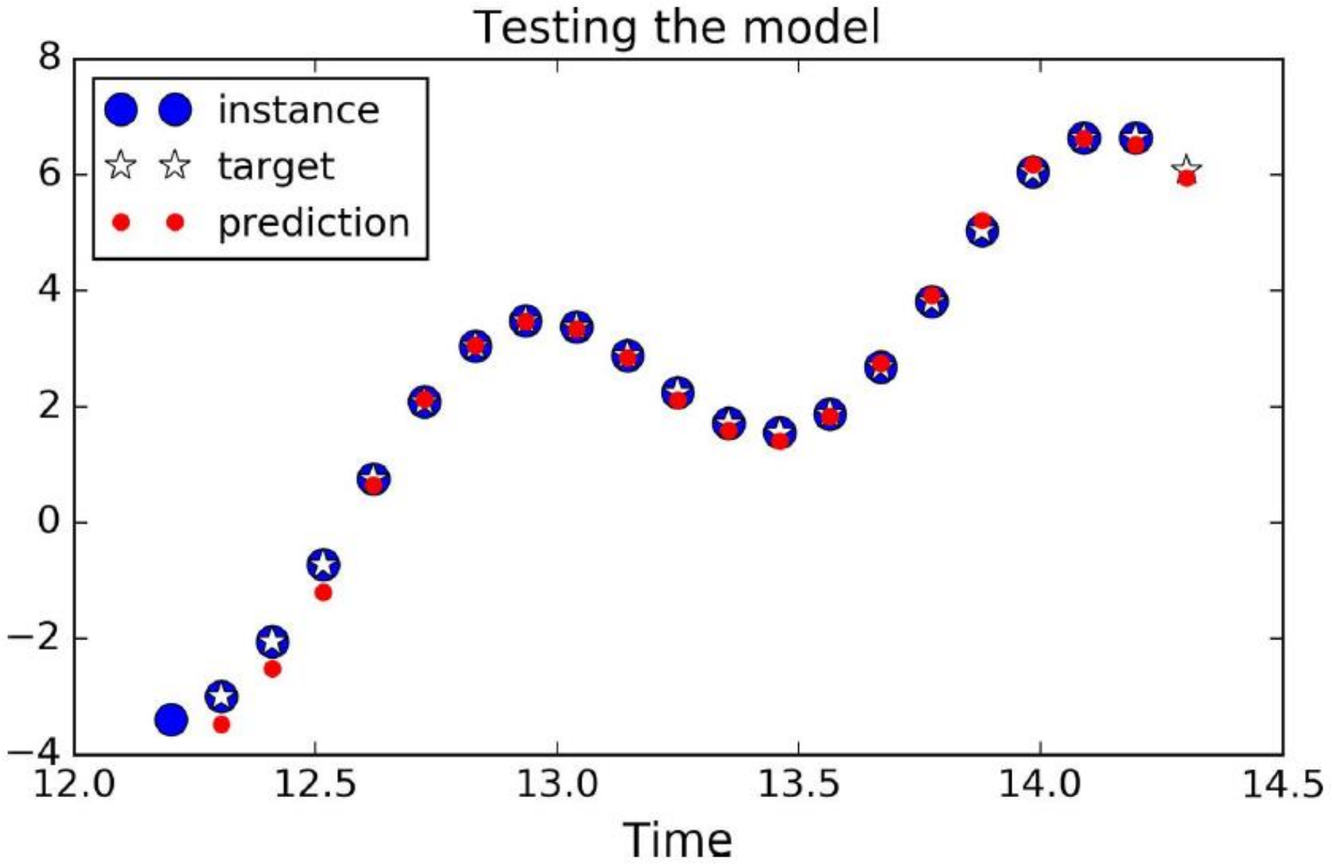


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Demo 2

Develop RNN to demonstrate sequence data processing

- **Objective:** Develop RNN to demonstrate sequence data processing.

- **Steps**:

1. Demonstrate how to handle variable sequence lengths.
2. Develop an MNIST classifier using 1-layer RNN, Dynamic RNN and AdamOptimizer.
3. Then, develop the classifier with 3-layer RNN.

- **Skills required**: Knowledge of RNNs, including concepts like unfolded computation graph, number of time steps, basic RNN cells, dynamic RNNs, variable sequence lengths, etc.

- **Dataset**: Sample data is generated within the code itself. You also use MNIST dataset to test a classification network.

- **Components of the code to be executed:** Run the tutorial titled “Recurrent Neural Networks”

Recurrent Neural Networks

Topic 7: Deep RNN and LSTM

Deep RNNs

- The model explained in prior sections was pertaining to one layered RNN.
- It is possible to stack individual RNN layers to create a multi-layered RNN or Deep RNN.
- Deep RNNs can be created by stacking many layers of cells.

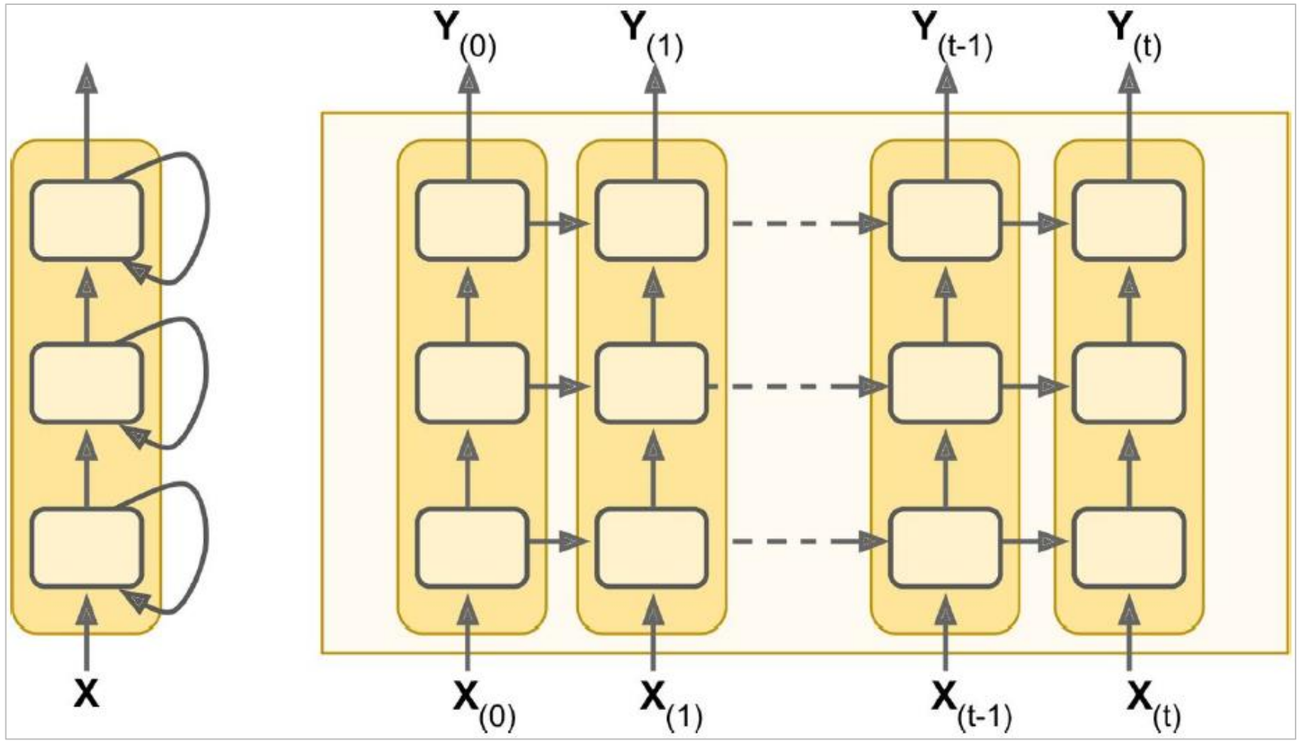


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Three Layered Deep RNN

- In TensorFlow, one can use ***MultiRNNCell***, which is created by stacking several ***BasicRNNCells***.
- The stacked cells can be of different kinds.
- Let’s create a three layered deep RNN.

Three Layer Deep RNN

```
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)
```

- For the code given above, states will be a tuple containing one tensor per layer, with each tensor representing final state of the cell.
- The shape of each state tensor will be [batch_size, n_neurons].
- If you set the parameter state_is_tuple = False while creating a MultiRNNCell, then states becomes a single tensor, containing states of all three layers and with shape [batch_size, n_layers * n_neurons].

Vanishing/Exploding Gradient Problem

- The vanishing or exploding gradient issue crops up with Deep RNNs.
- The techniques that can be used to prevent this are:
  - Good parameter initialization
  - Nonsaturating activation functions (e.g., ReLU)
  - Batch Normalization
  - Gradient Clipping
  - Faster optimizers

Vanishing/Exploding Gradient Problem

- However, Deep RNN training is slow, even with as many as 100 input sequences.
- One solution is truncated backpropagation over time.
- To do this, you can truncate the input sequence.
- But the issue with this is that long sequences cannot be learned.
- Another issue with RNNs is that memory of initial inputs gradually gets diluted and finally lost in later time steps.
- In some cases, the initial memory has significant value. For example, the first few words of a movie review given by audience might hold the key to user sentiment, regardless of later wordings.
- This is where LSTM (Long Short-term Memory) cells come to rescue.

LSTM Cells

- LSTM cells manage long- and short-term memory by means of logic gates, which control what information gets passed forward and what gets dropped.
- LSTMs are very successful in modeling long-term time series, long texts, audio recordings, etc.

- In TensorFlow, LSTM cell can be created via use of ***BasicLSTMCell***:

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

- LSTM cells perform much better, converge faster, and learn long-term dependencies.
- An LSTM cell manages two state vectors instead of just one.

State of LSTM Cells

- An LSTM cell has two states:
  - $c_{(t)}$ - long-term state
  - $h_{(t)}$ - short-term state
- The long-term state passes through forget gate to forget some memories and input gate to allow some memories.
- The long-term state is then passed through gate without transformation.
- The long-term state is also taken through tan h operation. The result is filtered by the output gate to produce short-term state $h_{(t)}$. This is equal to the cell's output for the time step $y_{(t)}$.

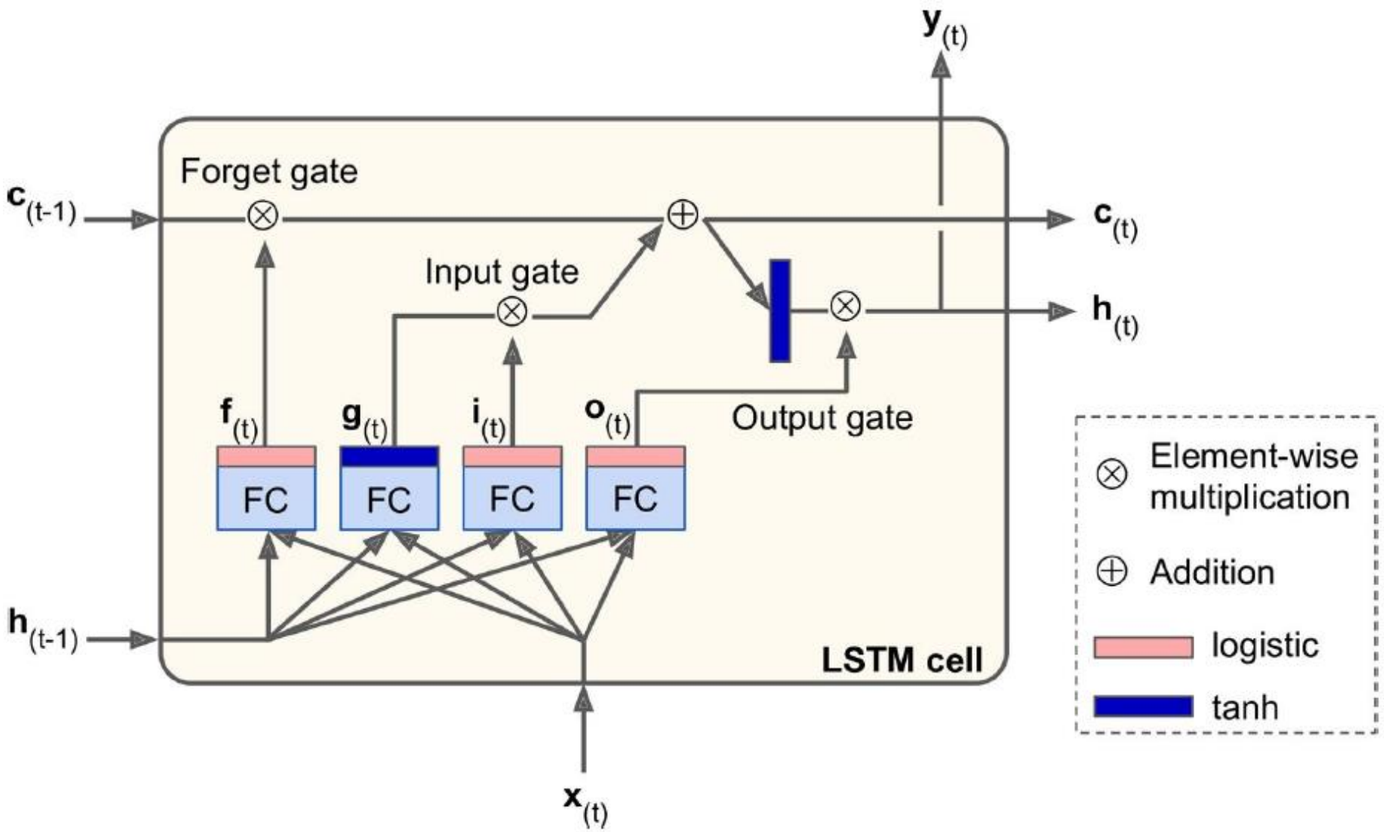


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

State of LSTM Cells

- Let’s see how the gates work.
- The current input $x_{(t)}$ and previous short-term state $h_{(t)}$ are fed to four fully connected layers.
- The main layer is the one producing output $g_{(t)}$. This layer is partially stored in the long-term state (controlled by the input gate).

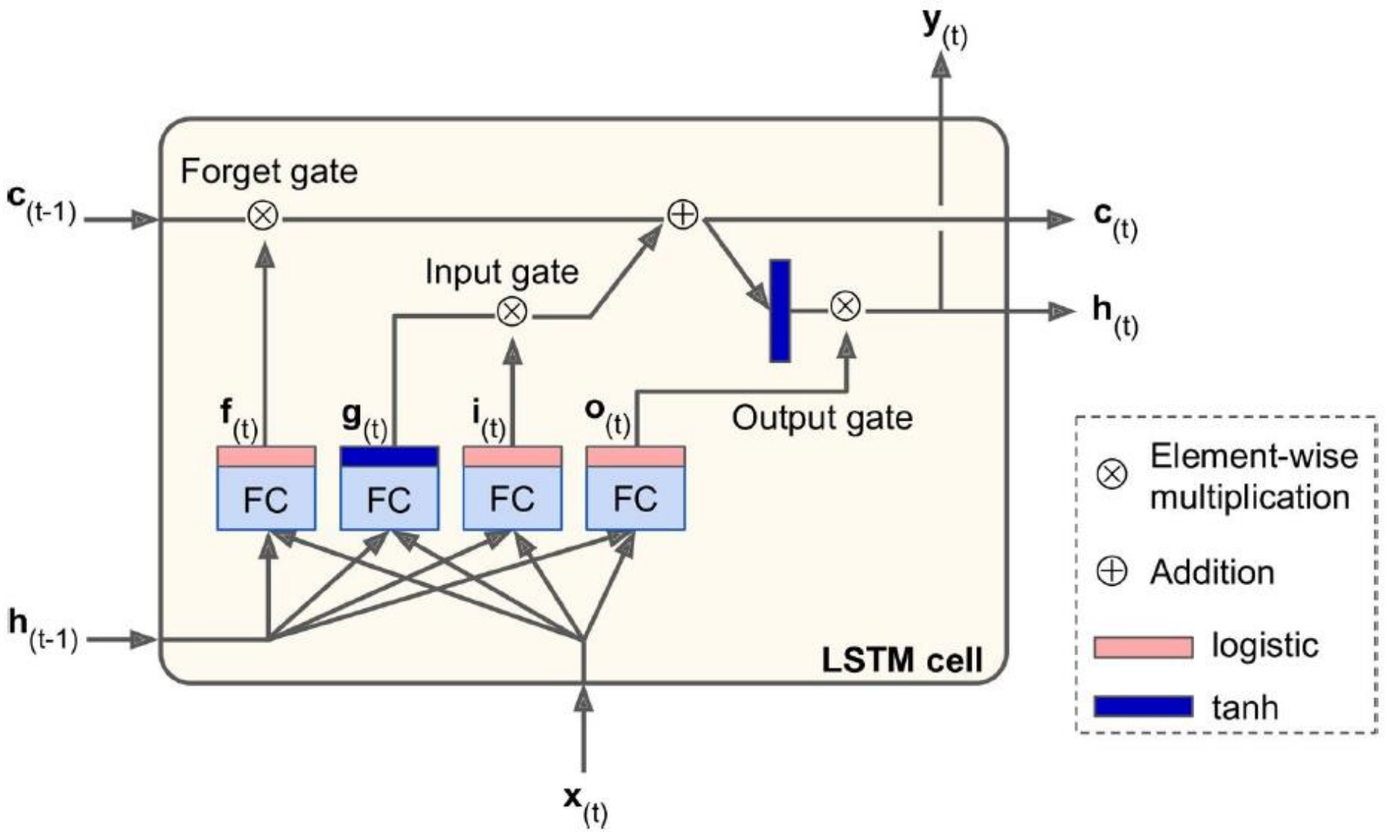


Image Source: “Hands-on Machine Learning with Scikit-Learn and TensorFlow ” by Aurelien Geron

State of LSTM Cells

- The other three layers are ***gate controllers***.
- They use logistic activation function, so their output can be 0 or 1.
- If the output is 0, the corresponding gate is closed or else it is left open.

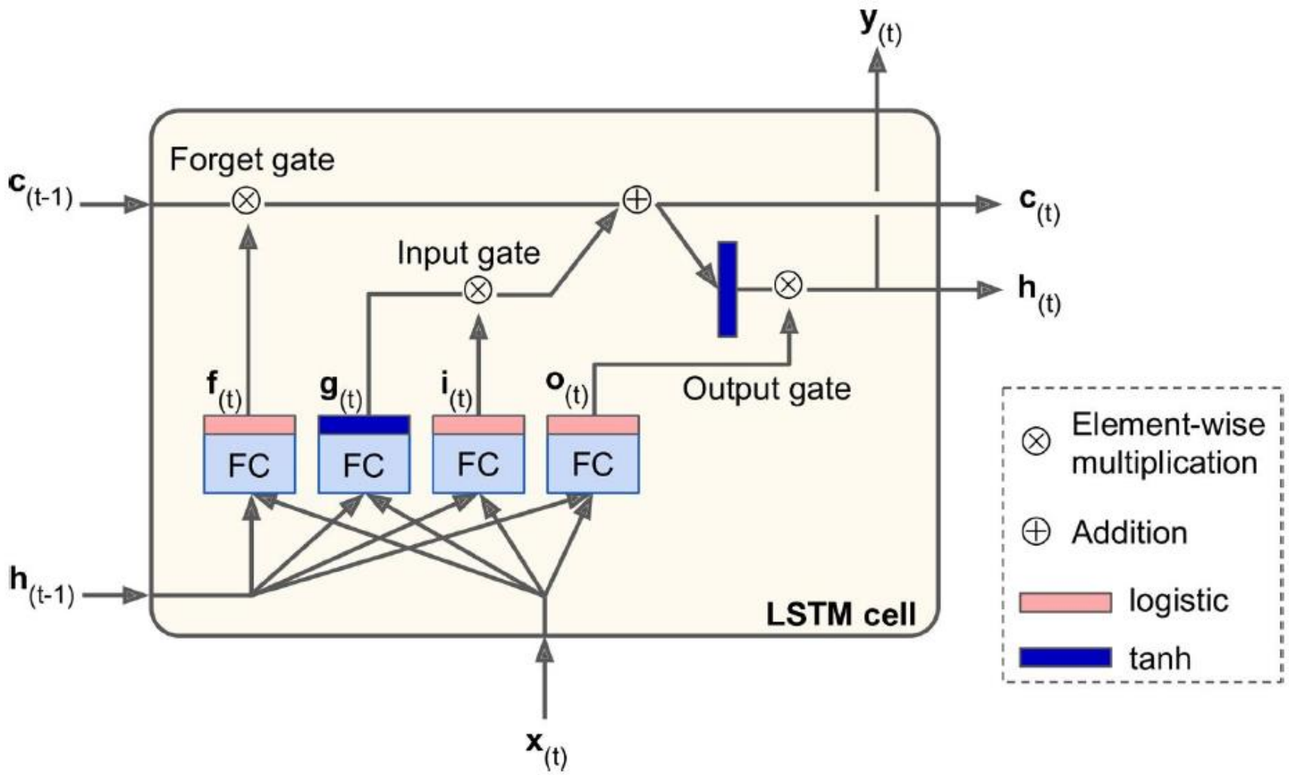


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

State of LSTM Cells

- The output $f_{(t)}$ controls the forget gate, which decides what part of long-term state is deleted.
- The output $i_{(t)}$ controls the input gate, which decides what part of $g_{(t)}$ gets added to the long-term state.
- The output $o_{(t)}$ controls the output gate, which decides what part of long-term state can be stored in $h_{(t)}$ and $y_{(t)}$.

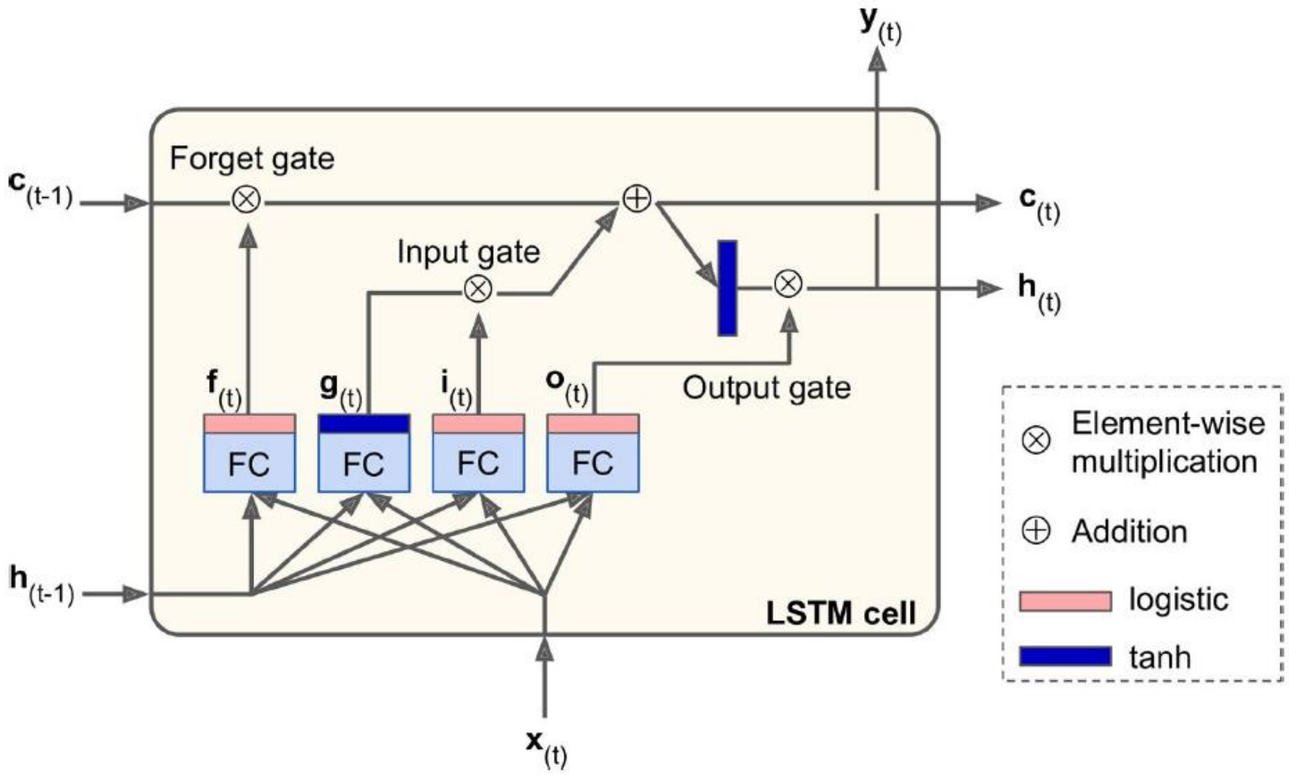


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

LSTM Cell Computations

$$
\begin{aligned}
\boldsymbol{i}_{(t)} &= \sigma\left(\boldsymbol{W}_{xi}^{T} \cdot \boldsymbol{x}_{(t)} + \boldsymbol{W}_{hi}^{T} \cdot \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_{i}\right) \\
\boldsymbol{f}_{(t)} &= \sigma\left(\boldsymbol{W}_{xf}^{T} \cdot \boldsymbol{x}_{(t)} + \boldsymbol{W}_{hf}^{T} \cdot \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_{f}\right) \\
\boldsymbol{o}_{(t)} &= \sigma\left(\boldsymbol{W}_{xo}^{T} \cdot \boldsymbol{x}_{(t)} + \boldsymbol{W}_{ho}^{T} \cdot \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_{o}\right) \\
\boldsymbol{g}_{(t)} &= \tanh\left(\boldsymbol{W}_{xg}^{T} \cdot \boldsymbol{x}_{(t)} + \boldsymbol{W}_{hg}^{T} \cdot \boldsymbol{h}_{(t-1)} + \boldsymbol{b}_{g}\right) \\
\boldsymbol{c}_{(t)} &= \boldsymbol{f}_{(t)} \otimes \boldsymbol{c}_{(t-1)} + \boldsymbol{i}_{(t)} \otimes \boldsymbol{g}_{(t)} \\
\boldsymbol{y}_{(t)} &= \boldsymbol{h}_{(t)} = \boldsymbol{o}_{(t)} \otimes \tanh\left(\boldsymbol{c}_{(t)}\right)
\end{aligned}
$$

LSTM Cell Computations

- $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, $\mathbf{W}_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, and $\mathbf{W}_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$, and $\mathbf{b}_g$ are the bias terms for each of the four layers. Note that TensorFlow initializes $\mathbf{b}_f$ to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

Demo 3

Demonstrate use of Deep RNNs with Multi RNN Cell

- **Objective:** Demonstrate use of Deep RNNs with Multi RNN Cell, using a random uniform distribution between 0 and 1.
- **Steps**:
  1. Demonstrate use of Dropout and DropoutWrapper in deep RNNs with keep_prob = 0.5.
  2. Plot the output to see if the Dropout helped.
  3. Develop an MNIST classifier using LSTM RNNs.
  4. Print training and test accuracy.

- **Skills required**: Understanding of RNNs and LSTMs in detail, including concepts like unfolded computation graph, number of time steps, basic RNN cells, dynamic RNNs, variable sequence lengths, time-series predictions using RNNs, deep RNNs, etc.
- **Dataset**: You generate some sample data within the code itself. You also use MNIST dataset to test an LSTM-based classification network.
- **Code to be executed:** Run the tutorial titled “Recurrent Neural Networks”.

Recurrent Neural Networks

Topic 8: Word Embedding

Word Embeddings

- RNNs are very useful for language modelling or language translation.
- For a vocabulary that is 50000 words strong, each *n*th word can be represented by a sparse vector 50000 long, with all 0 values except one in the *n*th position.
- This is very inefficient for large word set.
- One solution is to use Word Embeddings. Here, one creates a word map with proximity of like or similar words. A sentence like “I like milk” can be easily be replaced by “I like water” as milk and water can be closely placed in the word map.
- An embedding is a very small but dense vector with 150 dimensions.
- To train word embeddings, initially, it is seeded randomly with words. During training with a neural network, the similar words end up huddling close to each other.
- The results are amazing. For example, words might get placed on dimensions like verb/noun, singular/plural, etc.

Word Embeddings

- Let’s see the code to create word embeddings.
- Let’s create the embedding and initialize it randomly:

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

Word Embeddings

- Assume a sentence “I drink milk.” You wish to get its embeddings.
- First, you need to break this into a list of known words.
- Unknown words may be replaced by token “[UNK]”, numerical values may be replaced by “[NUM]”, URLs by “[URL]”, etc.
- Look up integer identifier for each word (0 to 49,999) in the dictionary.
- The embedding_lookup function will get the corresponding embeddings for this sentence.

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

- One can train one’s own embeddings or use previously trained embeddings (much like pre-trained layers of a neural net).

Machine Translation Model

- An Encoder-Decoder RNN can translate across languages; for example, English to French.
- Notice that target output is also fed to the Decoder but pushed back by one step.
- <go> is a token for the beginning of a sentence and <eos> is for the end of a sentence.
- Also, input English sentence is reversed first so that Decoder starts working with first word “I” first.

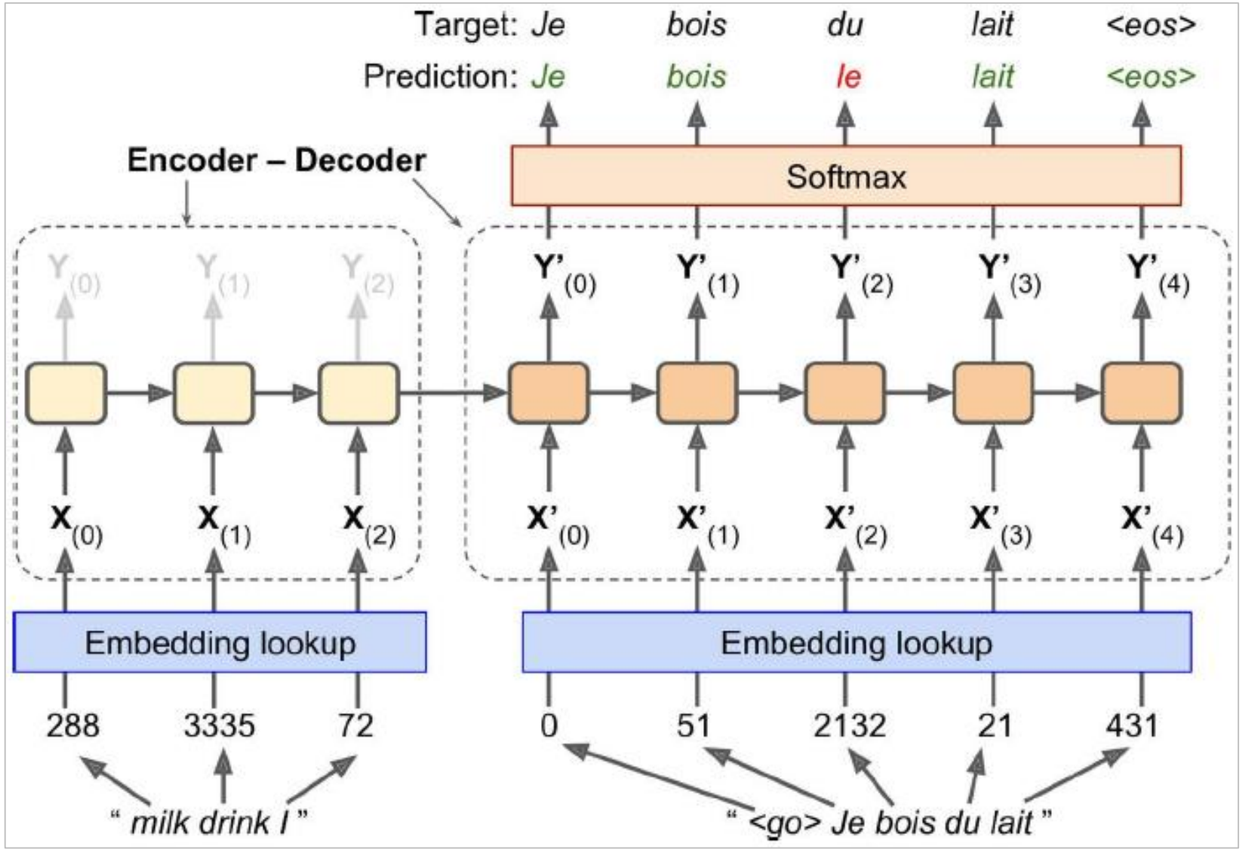


Image Source: “Hands-on Machine Learning with Scikit-Learn and TensorFlow ” by Aurelien Geron

Machine Translation Model

- The Embeddings are what are fed to both Encoder and Decoder.
- The Decoder outputs a score for each word in the output vocabulary, and Softmax converts this into a probability. For each step in the Decoder, the word with highest probability is output.

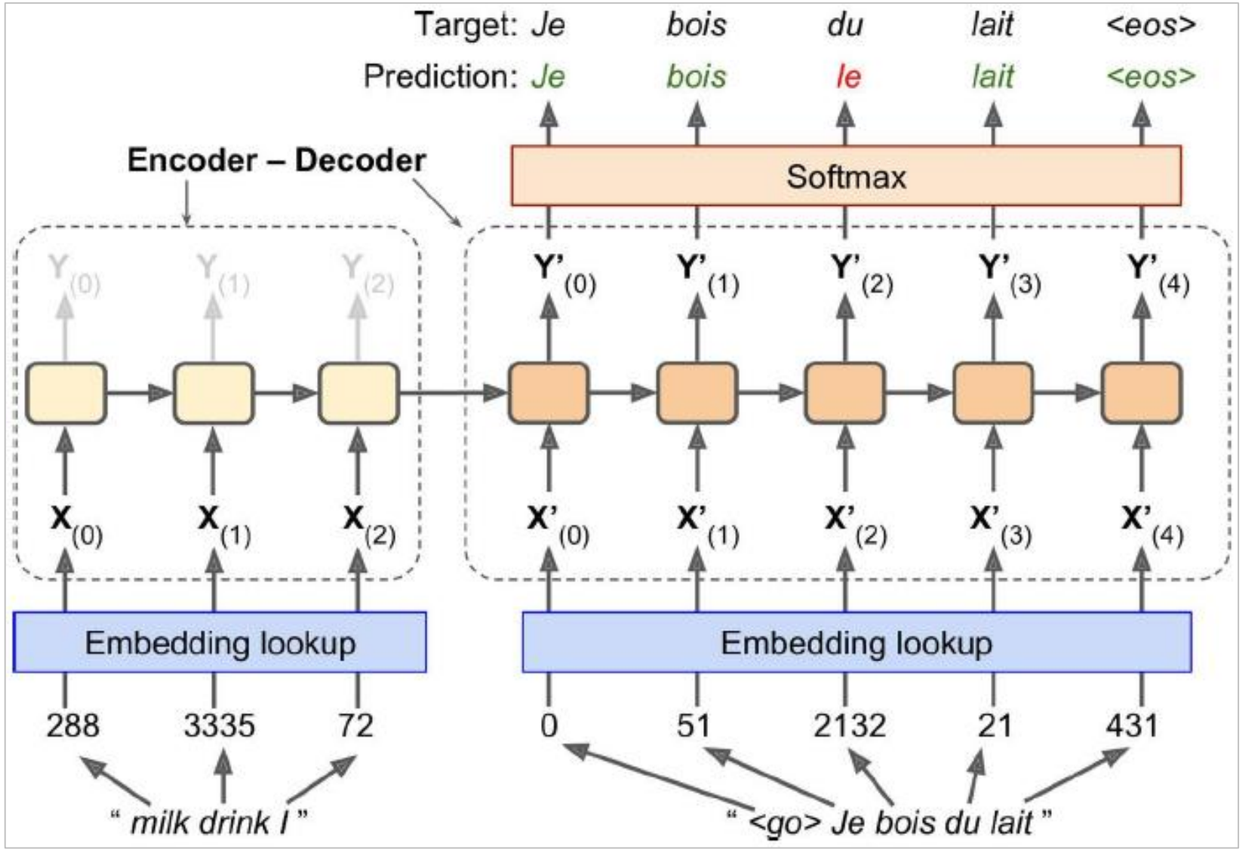


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Machine Translation Model

- Note that during production (inference time), the target sentence will not be available to feed to the Decoder.
- Here, you simply feed the output word from previous time step to the current time step. (This will require an embedding lookup that is not shown in the figure).

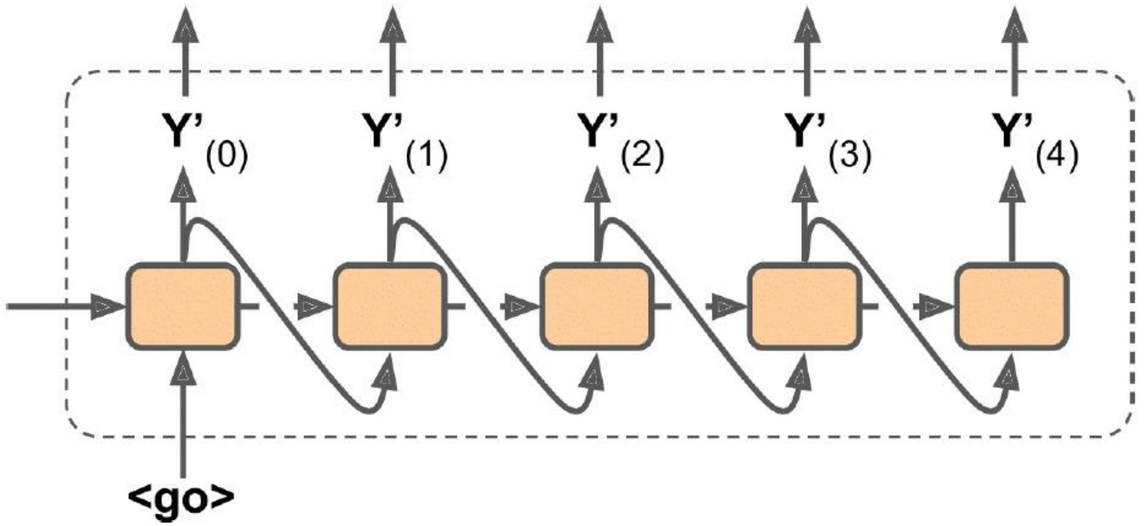


Image Source: "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Key Takeaways

- RNNs are useful for processing data that is sequential in nature.
- An RNN involves a recurrent neuron, where the output of the neuron is fed back as input to itself.
- RNNs can exist in various configurations.
- TensorFlow can be used to implement basic RNNs or dynamic RNNs.
- RNNs are trained via backpropagation through time or BPTT, similar to backpropagation logic of classic ANNs.
- RNNs can be used to predict time-series data like stock prices.
- LSTMs are a special form of RNN that use gate controllers to selectively preserve or delete memory of data.
- Word Embeddings simplify processing of large word dictionaries, allowing development of sophisticated NLP applications.
- An Encoder-Decoder model of RNN can be used for language translation models.

This concludes "Recurrent Neural Networks."

The next lesson is "Other Forms of Deep Learning."