Machine Learning I (DATS 6202)

Recurrent Neural Network

Yuxiao Huang

Data Science, Columbian College of Arts & Sciences
George Washington University

Spring 2020

Reference

- This set of slices was largely built on the following 8 wonderful books and a wide range of fabulous papers:

HML	Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)
PML	Python Machine Learning (3rd Edition)
ESL	The Elements of Statistical Learning (2nd Edition)
PRML	Pattern Recognition and Machine Learning
LFD	Learning From Data
NND	Neural Network Design (2nd Edition)
NNDL	Neural Network and Deep Learning
RL	Reinforcement Learning: An Introduction

- For most materials covered in the slides, we will specify their corresponding books and papers for further reference.

Code Example & Case Study

- See code example of topics discussed in this section in github repository: /p2_c2_s3_recurrent_neural_network/code_example
- See case study of Kaggle Competition related to this section in github repository: /p2_c2_s3_recurrent_neural_network/case_study

Overview

Learning Objectives

- It is **expected** to understand
  - the architecture and idea of short-term RNN cell
  - the good practices for designing and implementing RNN
  - the good practices for using pretrained RNN to address NLP problems
- It is **recommended** to understand
  - the backpropagations particularly designed for RNN (e.g., BPTT and RTRL)
  - the architecture and idea of long-term RNN cells (LSTM and GRU)

Recurrent Perceptron

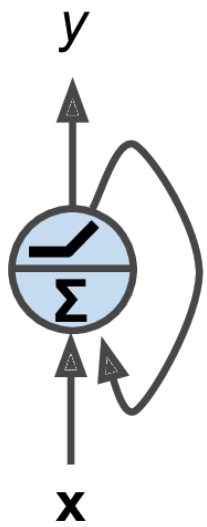

Figure 1: A recurrent perceptron. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition).*

- In *Feedforward Neural Network*, information always passes forward (from lower layer to higher layer).
- In *Recurrent Neural Network* (RNN), information also passes backward (from higher layer to lower layer).
- Fig. 1 shows a recurrent perceptron where the output of the output layer goes back to the input of the input layer.

Unrolling Recurrent Perceptron Through Time

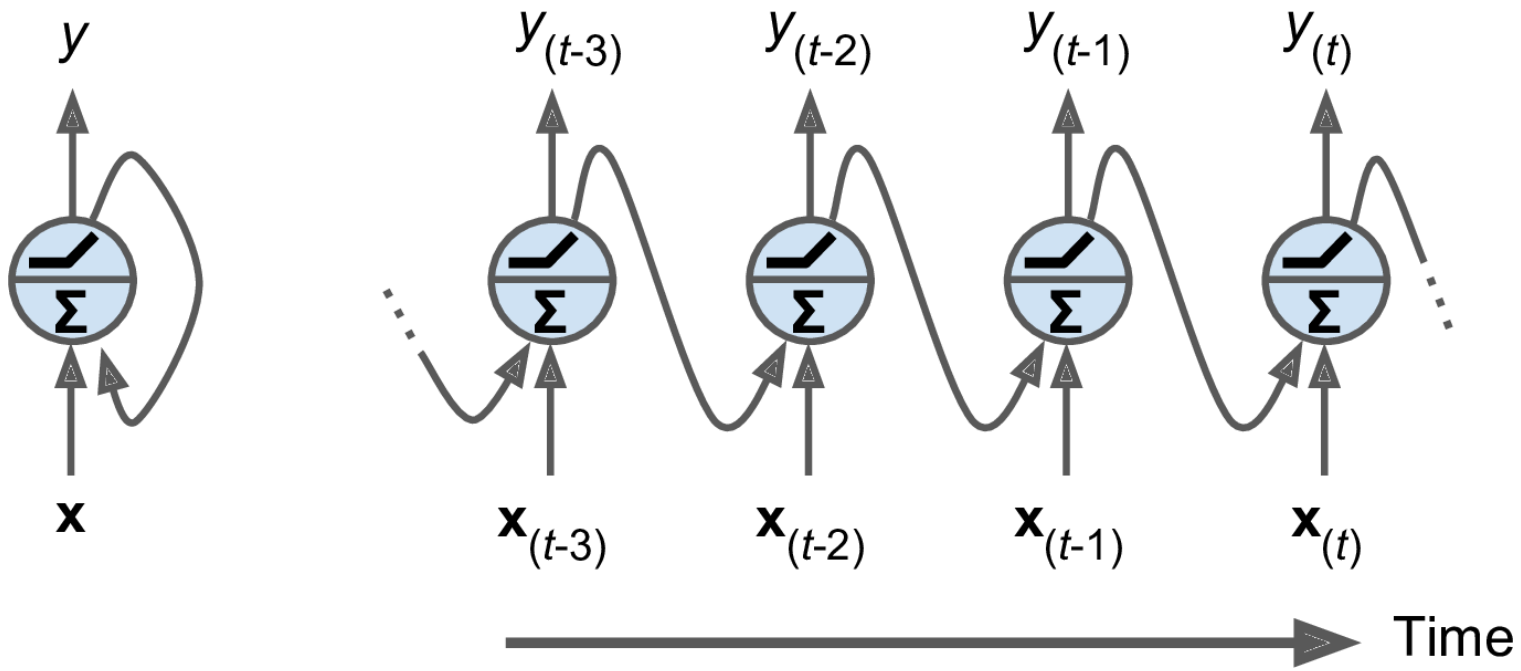

Figure 2: A recurrent perceptron unrolled through time. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- We can represent the left-most recurrent perceptron in fig. 2 as a function of time:
  - at time 0 (the first time step), the perceptron receives input at time 0
  - at time t (where $t > 0$), the perceptron receives not only input at time t, $\mathbf{x}_{(t)}$, but also its output at time $t-1$, $y_{(t-1)}$
- This representation is called *Unrolling the Network Through Time*.

Recurrent Layer

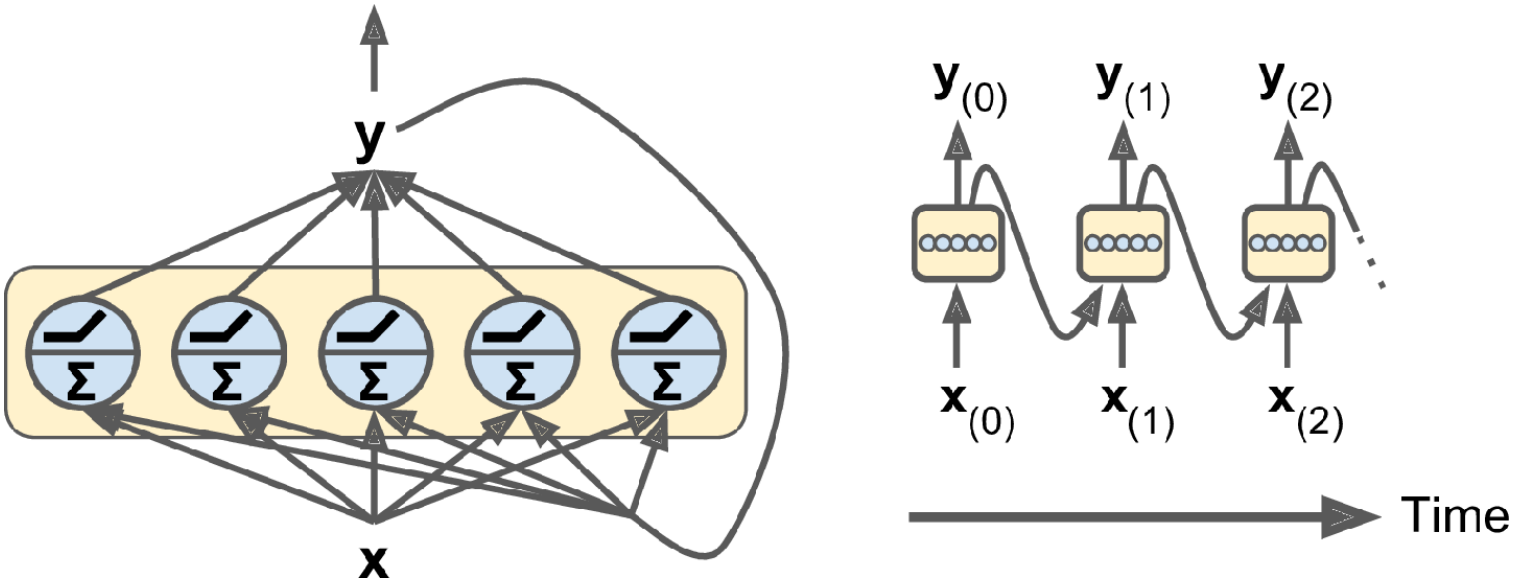

Figure 3: A layer of recurrent perceptrons unrolled through time. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition).*

- We can create a layer of recurrent perceptrons and represent the layer as a function of time:
  - at time 0 (the first time step), each perceptron on the layer receives input at time 0, $\mathbf{x}_{(0)}$
  - at time t (where $t > 0$), each perceptron on the layer receives not only input at time t, $\mathbf{x}_{(t)}$, but also output at time $t-1$, $\mathbf{y}_{(t-1)}$

Output of a Recurrent Layer

- The output of a recurrent layer at time t, $\mathbf{y}_{(t)}$, can be written as

$$\mathbf{y}_{(t)} = \mathbf{f}\left(\mathbf{W}_x \mathbf{x}_{(t)}^{\mathsf{T}} + \mathbf{W}_y \mathbf{y}_{(t-1)}^{\mathsf{T}} + \mathbf{b}\right). \tag{1}$$

  - $\mathbf{f}$ is the activation function (e.g., ReLU) of the layer
  - $\mathbf{x}_{(t)}$ is the input at time t
  - $\mathbf{W}_x$ is the connecting weight between the perceptrons on the layer and $\mathbf{x}_{(t)}$
  - $\mathbf{y}_{(t-1)}$ is the output of the layer at time $t-1$
  - $\mathbf{W}_y$ is the connecting weight between the perceptrons on the layer and $\mathbf{y}_{(t-1)}$
  - $\mathbf{b}$ is the bias of the perceptrons on the layer
- The recursive relationship in eq. (1) shows that $\mathbf{y}_{(t)}$ is a function of the input across all the previous time steps, $\mathbf{x}_{(0)}, \mathbf{x}_{(1)}, \cdots, \mathbf{x}_{(t-1)}$.

Memory Cell

- As mentioned earlier, the output of a recurrent perceptron at a time step is a function of the input across all the previous time steps. That is, a recurrent perceptron has a form of memory that can preserve some state (more on this later) across time steps.
- A recurrent perceptron is one kind of memory cell that can only learn short patterns (typically about 10 steps long). This kind of cell is also called *Short-Term Memory Cell*.
- There are more complex memory cells that allow longer patterns (typically about 10 times longer, more on this later). This kind of cell is also called *Long-Term Memory Cell*.

Output and Hidden State

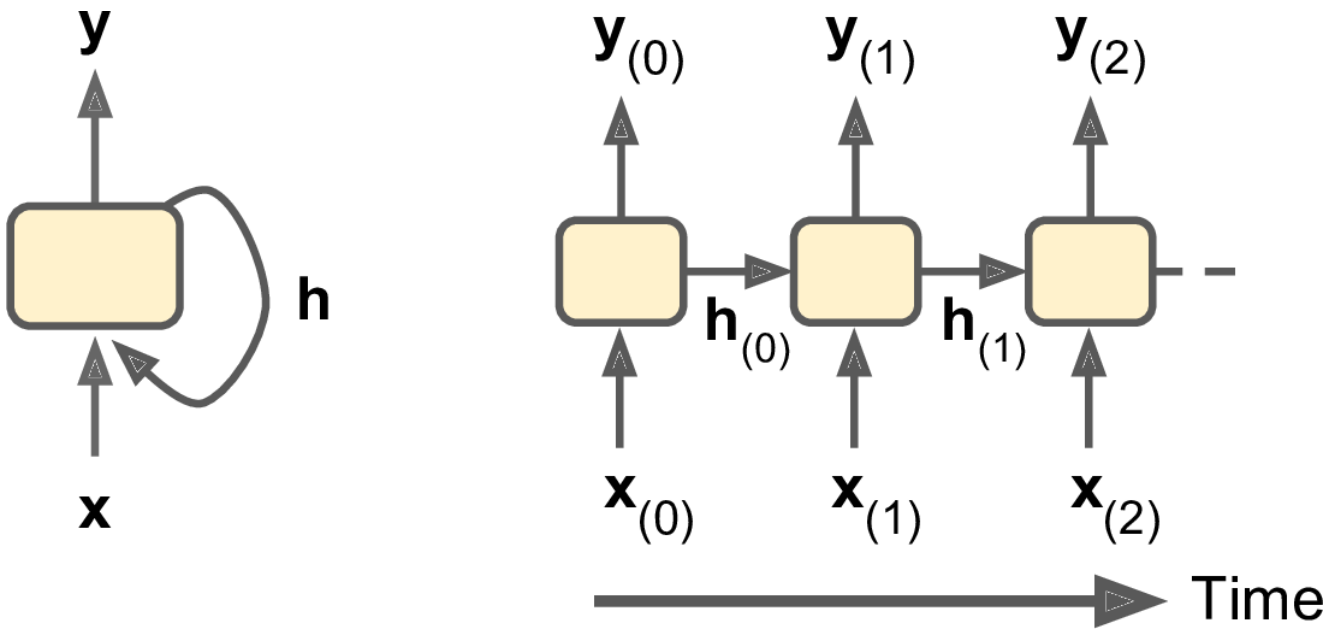

Figure 4: Output and hidden state of a memory cell. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- A memory cell's state at time step t, $\mathbf{h}_{(t)}$, is a function of the input at t, $\mathbf{x}_t$, and its state at $t-1$, $\mathbf{h}_{(t-1)}$.
- A memory cell's output at time step t, $\mathbf{y}_{(t)}$, is also a function of the input at t, $\mathbf{x}_t$, and its output at $t-1$, $\mathbf{y}_{(t-1)}$.
- While $\mathbf{h}_{(t)}$ and $\mathbf{y}_{(t)}$ could be the same in simple cells (left panel in fig. 4), they might be different in more complex cells (right panel in the figure).

Input and Output Sequences

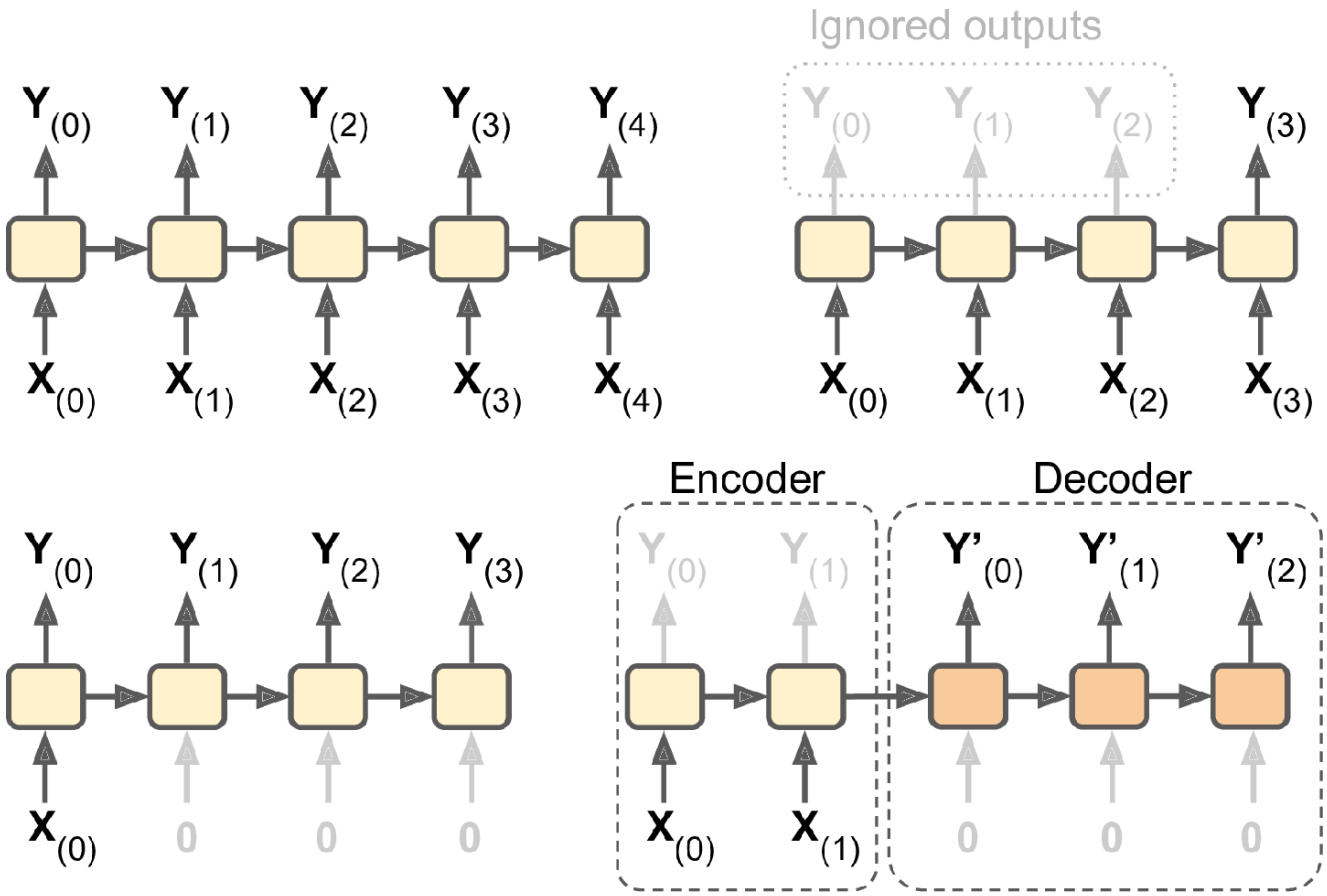

Figure 5: Four types of input and output sequences of a RNN. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition).*

Input and Output Sequences

- *Sequence-to-Sequence Network* (top-left panel in fig. 5):
  - a RNN that takes as input a sequence and outputs a sequence
  - e.g., given the input (a sequence), predict the output (also a sequence) that is some time steps later than the input
- *Sequence-to-Vector Network* (top-right panel in fig. 5):
  - a RNN that takes as input a sequence and outputs a vector
  - e.g., given the book review (a sequence), predict the book rating (a vector)
- *Vector-to-Sequence Network* (bottom-left panel in fig. 5):
  - a RNN that takes as input a vector and outputs a sequence
  - e.g., given an image (a vector), predict its caption (a sequence)
- *Encoder-Decoder Network* (bottom-right panel in fig. 5):
  - a RNN that begins with a sequence-to-vector network, named *Encoder*, and ends with a vector-to-sequence network, named *Decoder*
  - e.g., given a sentence in one language (a sequence), encode it into a latent representation (a vector), which is then decoded into a sentence in another language (a sequence)
  - sequence-to-sequence network may not work here since we may have to see the whole sentence to know its meaning

Deep RNN

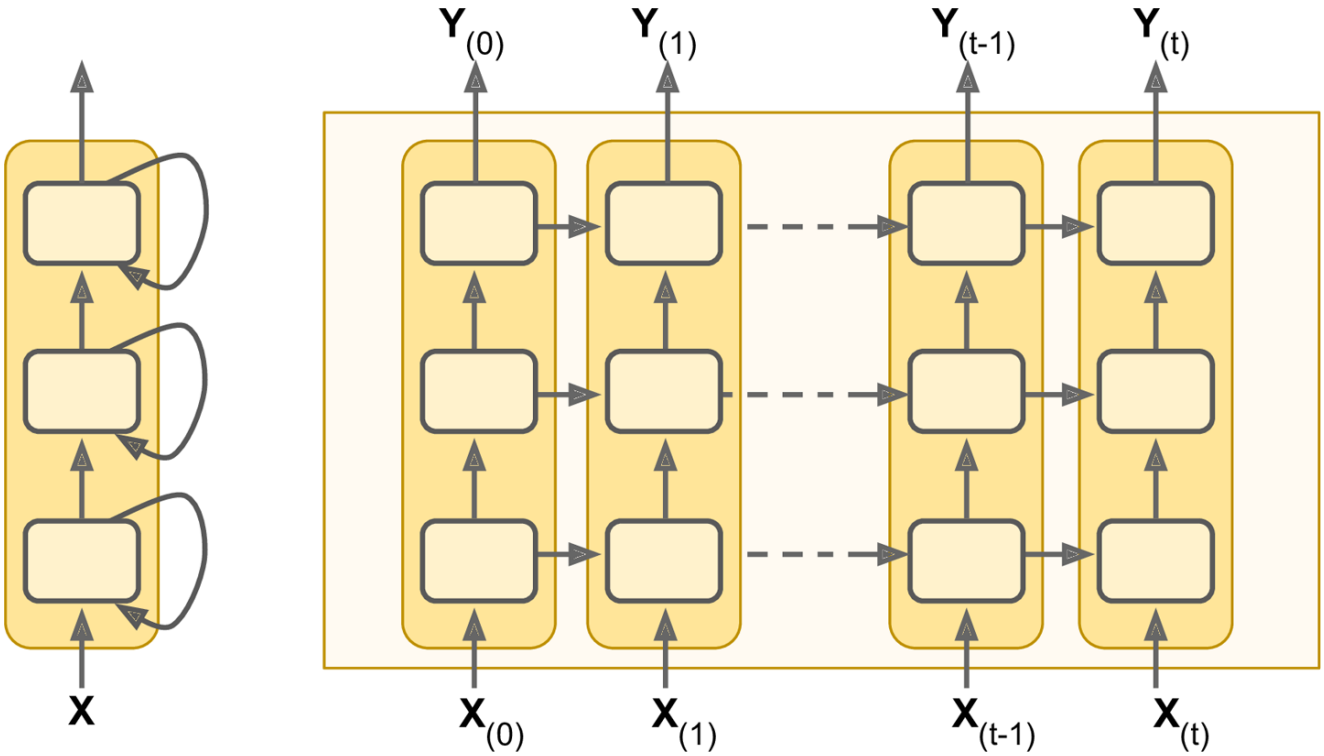

Figure 6: A deep RNN with three layers. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- Previously we have been focusing on RNNs that have only one hidden layer.
- Similar to DNN, a RNN can also have many hidden layers, making it a *Deep RNN*.

Further Reading

- See HML Chapter 15 for a high-level description of a special backpropagation particularly designed for RNN, named *Backpropagation Through Time* (BPTT).
- See NND Chapter 14 for:
  - a detailed discussion of BPTT
  - a detailed discussion of another backpropagation, also particularly designed for RNN, named *Real-Time Recurrent Learning* (RTRL)
  - the pros and cons of BPTT and RTRL

The Short-Term Memory Problem

- After we pass the data to a RNN, it will go through a series of transformations (e.g., when calculating the net input and activation on each layer).
- Such transformations can also repeat themselves since we feed the output of a RNN back to the input layer.
- As a result, when a RNN receives the later elements of a long sequence, the fed back output may be very different from the earlier elements of the long sequence.
- In other words, we can think of a RNN as *Dory the fish* who, when translating the last word in a long sentence, already forgets about the first word in the sentence.
- This is called the *Short-Term Memory Problem*.

Long-Term Memory Cell

- To address the short-term memory problem of short-term memory cell, many cells with long-term memory have been proposed.
- Here we will introduce two popular long-term memory cells:
  - *Long Short-Term Memory* (LSTM) cell [2]
  - *Gated Recurrent Unit* (GRU) cell
- Compared to short-term memory cell, LSTM and GRU usually perform much better:
  - training LSTM and GRU could converge faster
  - LSTM and GRU could detect long-term dependencies in the data

LSTM Cell, As a Blackbox

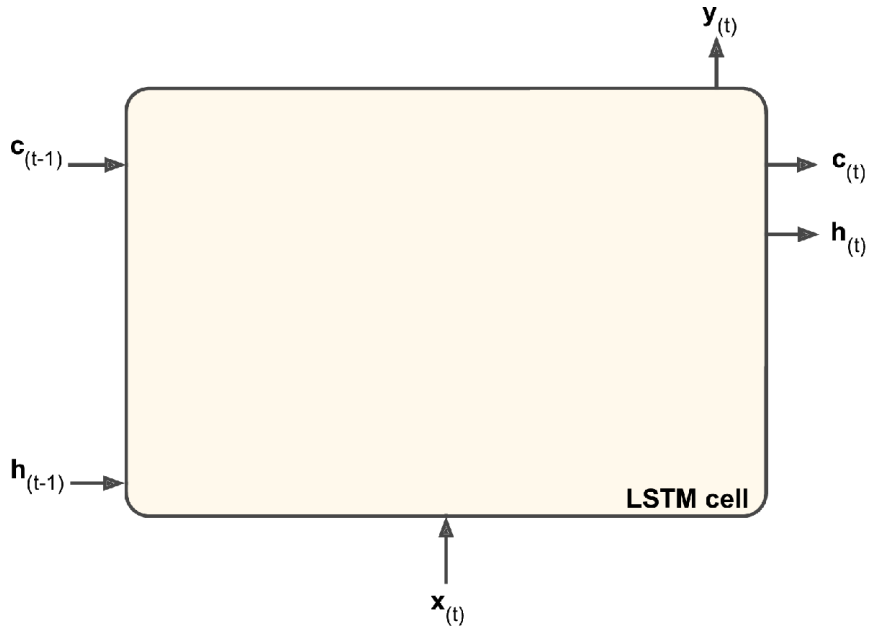

Figure 7: LSTM cell, as a blackbox. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- If we treat LSTM as a blackbox and only focus on its input and output, then the only difference between LSTM and short-term memory cell is
  - short-term memory cell has input $\mathbf{x}_{(t)}$, output $\mathbf{y}_{(t)}$ and short-term states $\mathbf{h}_{(t-1)}$ and $\mathbf{h}_{(t)}$
  - LSTM has extra long-term states $\mathbf{c}_{(t-1)}$ and $\mathbf{c}_{(t)}$
- The long-term states are the reason why LSTM works better on long sequences.

The Architecture of LSTM Cell

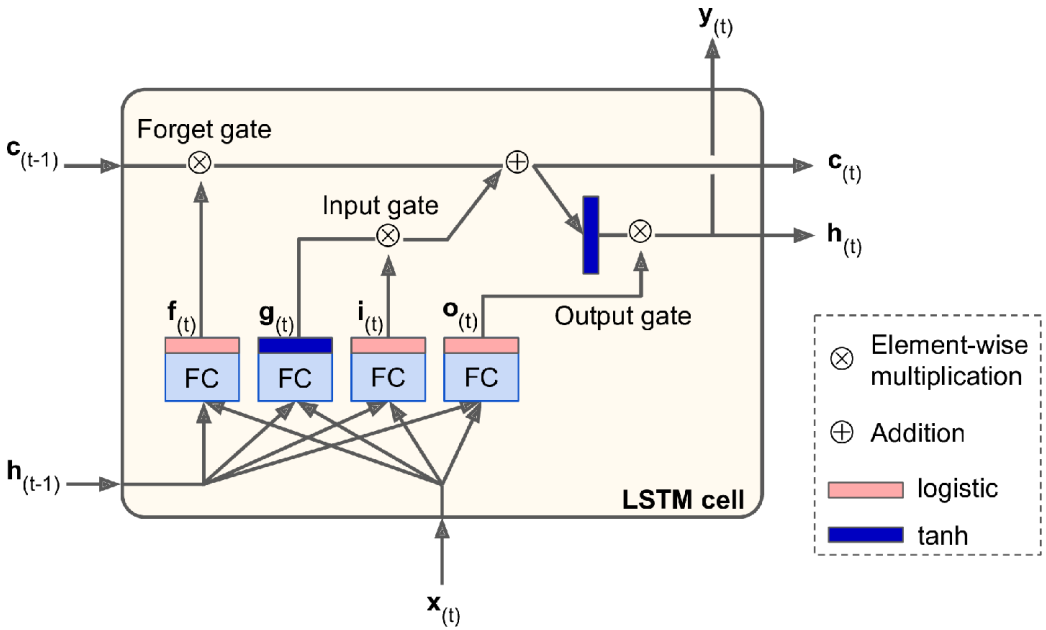

Figure 8: The architecture of LSTM cell. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- There are four kinds of fully connected layers in LSTM, which all take as input $\mathbf{h}_{(t-1)}$ and $\mathbf{x}_{(t)}$. However:
  - the *Main Layer* outputs the activation (`tanh`), $\mathbf{g}_{(t)}$
  - the *Forget Gate Controller* outputs the activation (`logistic`), $\mathbf{f}_{(t)}$
  - the *Input Gate Controller* outputs the activation (`logistic`), $\mathbf{i}_{(t)}$
  - The *Output Gate Controller* outputs the activation (`logistic`), $\mathbf{o}_{(t)}$

The Architecture of LSTM Cell

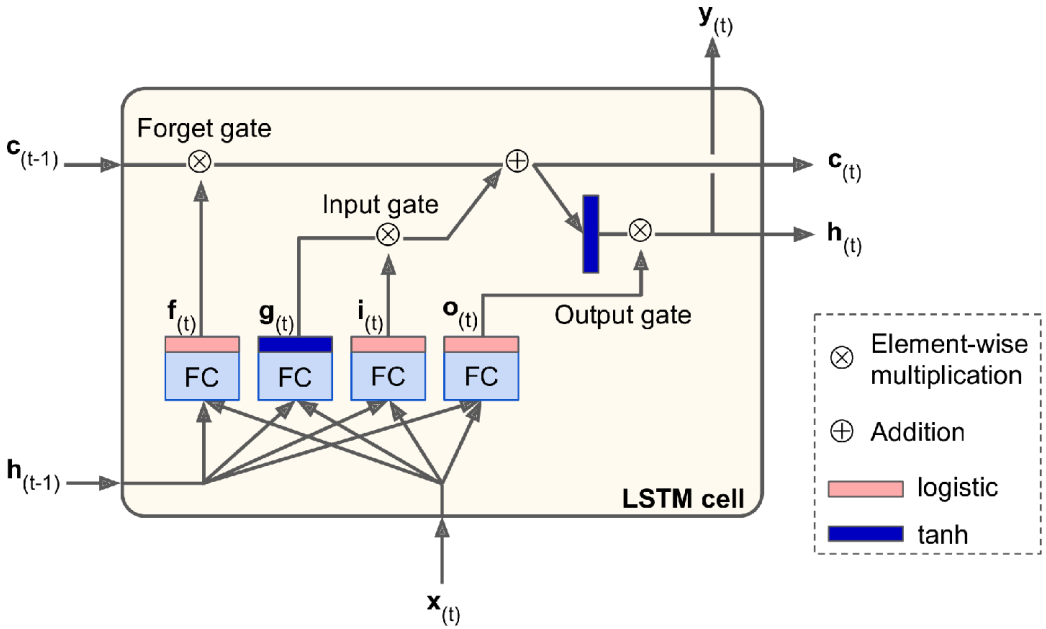

Figure 9: The architecture of LSTM cell. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- There are three major steps in LSTM:
  - recognize important input, with the *Input Gate* (taking as input $\mathbf{g}_{(t)}$ and $\mathbf{i}_{(t)}$)
  - store the important input in the long-term state for as long as it is needed, with the *Forget Gate* (taking as input $\mathbf{c}_{(t-1)}$ and $\mathbf{f}_{(t)}$)
  - extract the important input whenever it is needed, with the *Output Gate* (taking as input $\tanh(\mathbf{c}_{(t-1)})$ and $\mathbf{o}_{(t)}$)

The Math Behind LSTM Cell

- The output of the input gate controller at time step t, $\mathbf{i}_{(t)}$, is

$$\mathbf{i}_{(t)} = \sigma\left(\mathbf{W}_{xi}\mathbf{x}_{(t)}^{\intercal} + \mathbf{W}_{hi}\mathbf{h}_{(t-1)}^{\intercal} + \mathbf{b}_i\right). \tag{2}$$

- The output of the forget gate controller at time step t, $\mathbf{f}_{(t)}$, is

$$\mathbf{f}_{(t)} = \sigma\left(\mathbf{W}_{xf}\mathbf{x}_{(t)}^{\intercal} + \mathbf{W}_{hf}\mathbf{h}_{(t-1)}^{\intercal} + \mathbf{b}_f\right). \tag{3}$$

- The output of the output gate controller at time step t, $\mathbf{o}_{(t)}$, is

$$\mathbf{o}_{(t)} = \sigma\left(\mathbf{W}_{xo}\mathbf{x}_{(t)}^{\intercal} + \mathbf{W}_{ho}\mathbf{h}_{(t-1)}^{\intercal} + \mathbf{b}_o\right). \tag{4}$$

- The output of the main layer at time step t, $\mathbf{g}_{(t)}$, is

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}\mathbf{x}_{(t)}^{\intercal} + \mathbf{W}_{hg}\mathbf{h}_{(t-1)}^{\intercal} + \mathbf{b}_g\right). \tag{5}$$

- The long-term state at time step t, $\mathbf{c}_{(t)}$, is

$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}. \tag{6}$$

- The short-term state and output at time step t, $\mathbf{h}_{(t)}$ and $\mathbf{y}_{(t)}$, are

$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right). \tag{7}$$

The Math Behind LSTM Cell

- In eqs. (2) to (5):
  - $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$ and $\mathbf{W}_{xg}$ are the connecting weights between the input at time step t, $\mathbf{x}_{(t)}$, and the input gate controller, forget gate controller, output gate controller and the main layer
  - $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$ and $\mathbf{W}_{hg}$ are the connecting weights between the short-term state at time step $t-1$, $\mathbf{h}_{(t-1)}$, and the input gate controller, forget gate controller, output gate controller and the main layer
  - $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$ and $\mathbf{b}_g$ are the biases of the input gate controller, forget gate controller, output gate controller and the main layer
- In eq. (6):
  - $\mathbf{f}_{(t)}$, $\mathbf{i}_{(t)}$ and $\mathbf{g}_{(t)}$ are the output of the forget gate controller, input gate controller and the main layer at time step t
  - $\mathbf{c}_{(t-1)}$ is the long-term state at time step $t-1$
- In eq. (7):
  - $\mathbf{o}_{(t)}$ is the output of the output gate controller at time step t
  - $\mathbf{c}_{(t)}$ is the long-term state at time step t

Peephole Connections

- In regular LSTM cell, the input/forget/output gate controller only receives the input at time step t, $\mathbf{x}_{(t)}$, and the short-term state at time step $t-1$, $\mathbf{h}_{(t-1)}$.
- An extension of LSTM with extra connections called *Peephole Connections* was proposed in [1]:
  - the long-term state at time step $t-1$, $\mathbf{c}_{(t-1)}$, is added as an input to the input and forget gate controller
  - the long-term state at time step t, $\mathbf{c}_{(t)}$, is added as an input to the output gate controller

GRU Cell, As a Blackbox

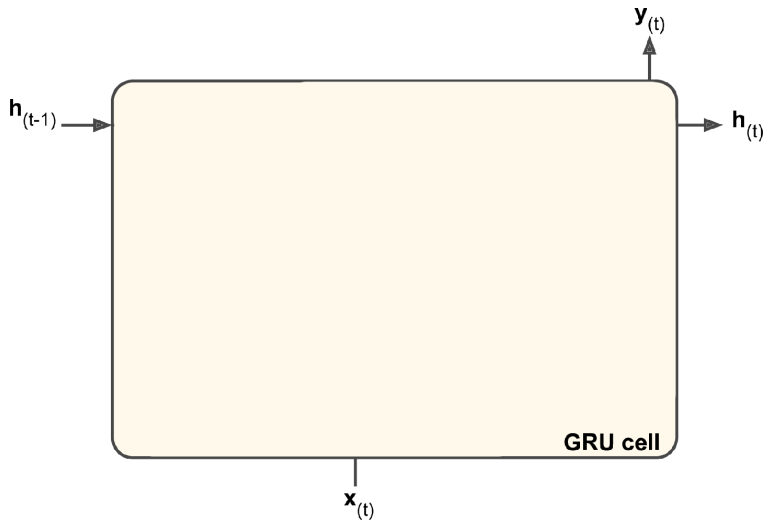

Figure 10: GRU cell, as a blackbox. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- If we treat GRU cell as a blackbox and only focus on its input and output, then there is no difference between GRU cell and short-term memory cell:
  - both cells have input $\mathbf{x}_{(t)}$, output $\mathbf{y}_{(t)}$ and short-term states $\mathbf{h}_{(t-1)}$ and $\mathbf{h}_{(t)}$
- Note that LSTM has extra long-term states $\mathbf{c}_{(t-1)}$ and $\mathbf{c}_{(t)}$.

The Architecture of GRU Cell

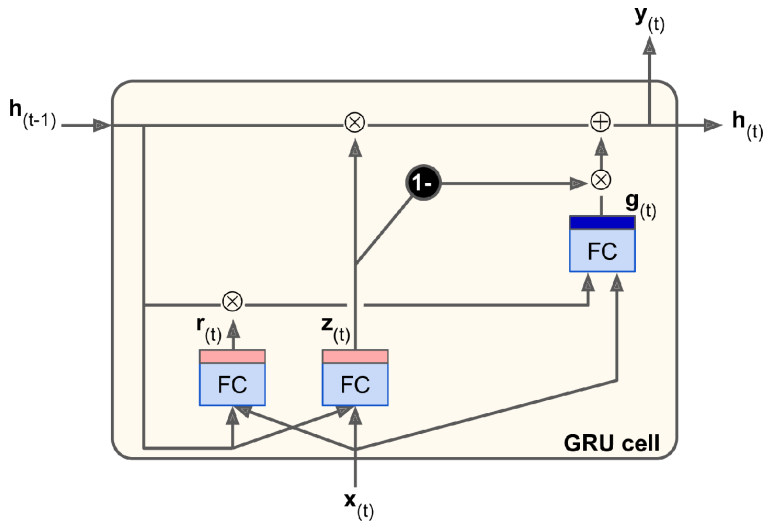

Figure 11: The architecture of GRU cell. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- There are three kinds of fully connected layers in GRU:
  - a gate controller (receiving $\mathbf{h}_{(t-1)}$ and $\mathbf{x}_{(t)}$) outputs the activation (`logistic`), $\mathbf{z}_{(t)}$
  - a gate controller (receiving $\mathbf{h}_{(t-1)}$ and $\mathbf{x}_{(t)}$) outputs the activation (`logistic`), $\mathbf{r}_{(t)}$
  - the *Main Layer* (receiving $\mathbf{r}_{(t)}$ and $\mathbf{x}_{(t)}$) outputs the activation (`tanh`), $\mathbf{g}_{(t)}$

The Architecture of GRU Cell

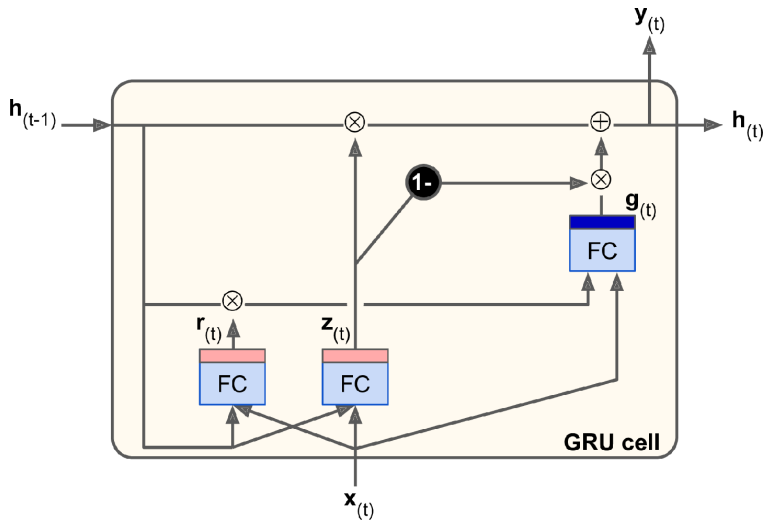

Figure 12: The architecture of GRU cell. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

- There are three major steps in GRU:
  - recognize important input (that should be used when producing the output), with gate controller that outputs $\mathbf{z}_{(t)}$ and main layer that outputs $\mathbf{g}_{(t)}$
  - store the important input (that should be used when producing the output) in the state for as long as it is needed, with gate controller that outputs $\mathbf{z}_{(t)}$ (the 1- operation allows the controller to switch between its two functions)
  - recognize important input (that should be used when producing $\mathbf{g}_{(t)}$), with gate controller that outputs $\mathbf{r}_{(t)}$

The Math Behind the GRU Cell

- The output of a gate controller at time step t, $\mathbf{z}_{(t)}$, is

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}\mathbf{x}_{(t)}^{\mathsf{T}} + \mathbf{W}_{hz}\mathbf{h}_{(t-1)}^{\mathsf{T}} + \mathbf{b}_z\right). \tag{8}$$

- The output of a gate controller at time step t, $\mathbf{r}_{(t)}$, is

$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}\mathbf{x}_{(t)}^{\mathsf{T}} + \mathbf{W}_{hr}\mathbf{h}_{(t-1)}^{\mathsf{T}} + \mathbf{b}_r\right). \tag{9}$$

- The output of the main layer at time step t, $\mathbf{g}_{(t)}$, is

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}\mathbf{x}_{(t)}^{\mathsf{T}} + \mathbf{W}_{hg}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right)^{\mathsf{T}} + \mathbf{b}_g\right). \tag{10}$$

- The short-term state and output at time step t, $\mathbf{h}_{(t)}$ and $\mathbf{y}_{(t)}$, are

$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}. \tag{11}$$

The Math Behind the GRU Cell

- In eqs. (8) to (10):
  - $\mathbf{W}_{xz}$, $\mathbf{W}_{xr}$ and $\mathbf{W}_{xg}$ are the connecting weights between the input at time step t, $\mathbf{x}_{(t)}$, and the two gate controllers and the main layer
  - $\mathbf{W}_{hz}$, $\mathbf{W}_{hr}$ and $\mathbf{W}_{hg}$ are the connecting weights between the short-term state at time step $t-1$, $\mathbf{h}_{(t-1)}$, and the two gate controllers and the main layer
  - $\mathbf{b}_z$, $\mathbf{b}_r$ and $\mathbf{b}_g$ are the biases of the two gate controllers and the main layer
- In eq. (10):
  - $\mathbf{r}_{(t)}$ is the output of a gate controller at time step t
- In eq. (11):
  - $\mathbf{z}_{(t)}$ is the output of a gate controller at time step t
  - $\mathbf{g}_{(t)}$ is the output of the main layer at time step t

Transfer Learning

- Similar to transfer learning with CNN:
  - when there are state-of-the-art RNNs pretrained on similar data, it is suggested to reuse and tweak the pretrained RNN to make it suitable for our data
  - transfer learning will not only speed up designing, training and fine-tuning RNN considerably, but also require significantly less training data
  - it turns out that transfer learning also works particularly well in NLP, since the lower layers of a pretrained RNN will usually capture simple features that are common in many data (hence can be reused)

State-Of-The-Art Pretrained RNN

- See state-of-the-art pretrained RNN in TensorFlow Hub.

Training RNN with Pretrained RNN

- Here we can simply follow the good practice (for training DNN with pretrained DNN) discussed previously (also shown below).
- **Good Practice**:
  - to train a DNN with pretrained model, we should
    1. freeze all the reused layers of the pretrained DNN (i.e., make their weights non-trainable so that backpropagation will not change them) then train the DNN
    2. unfreeze one or two top hidden layers of the pretrained DNN (the more training data we have the more top hidden layers we can unfreeze) and reduce the learning rate when doing so (thus the fine-tuned weights on the lower layers will not change significantly)
  - If the above steps do not produce an accurate DNN
    - if we do not have sufficient data, we can drop the top hidden layers and repeat the above steps
    - otherwise, we can replace (rather than drop) the top hidden layers or even add more hidden layers

Designing and Training DNN with Pretrained DNN

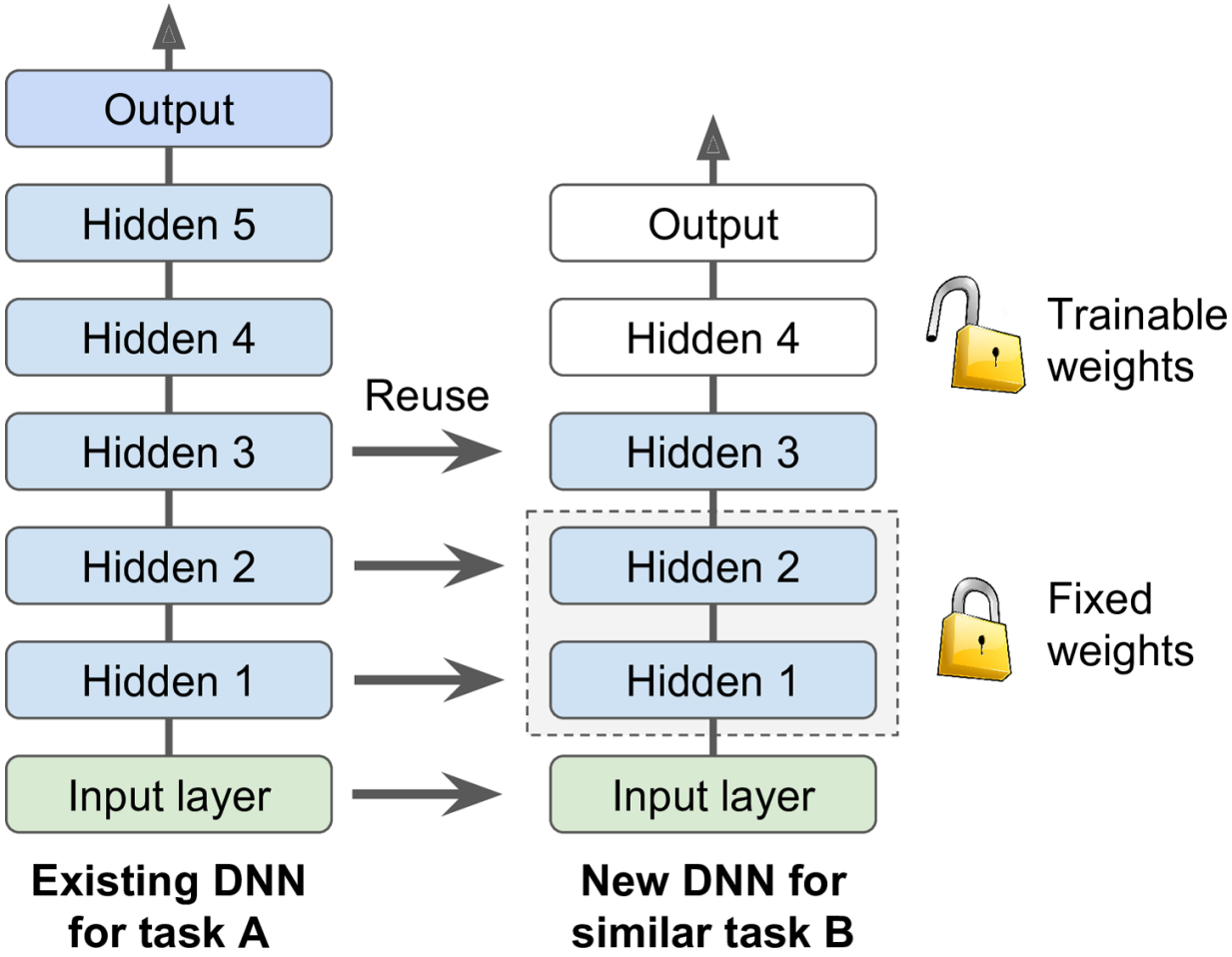

Figure 13: Designing and training DNN with pretrained DNN. Picture courtesy of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Edition)*.

Implementing RNN with Pretrained RNN

- The code below shows how to build a RNN on top of a pretrained nnlm-en-dim50 sentence embedding module using `hub.KerasLayer`, for IMDB-Reviews dataset:

```
1  # Get the model
2  model = keras.Sequential([hub.KerasLayer(
3      'https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1',
4      trainable=False,
5      dtype=tf.string,
6      input_shape=[],
7      output_shape=[50]),
8      keras.layers.Dense(128, activation='relu'),
9      keras.layers.Dense(1, activation='sigmoid')])
```

- Lines 2 to 7: The module is a *Sentence Encoder*, which transforms strings into numerical vectors (with 50 numbers in this case)
- Line 4: `trainable=False` indicates that the parameters are not trainable

Freezing the Pretrained Layers

- The code below shows how to freeze the pretrained layers:

```
1 # Freeze the pretrained layers
2 model.layers[0].trainable = False
```

- Line 2: `model.layers[0].trainable` means whether the parameters of the first layer (the pretrained module) should be trainable (changable). Note that the code above is not necessary here since we already set `trainable=False` when implementing the model.
- The code below shows how to compile the RNN (we need to compile a model after freezing its layers):

```
1 # Compile the model
2 model.compile(
3     optimizer=keras.optimizers.Adam(learning_rate=10 ** -3),
4     loss='binary_crossentropy',
5     metrics=['accuracy'])
```

- Line 3: We use `Adam` optimizer with `learning rate` 0.001.

Model Summary

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 50)                48190600
_________________________________________________________________
dense (Dense)                (None, 128)               6528
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 48,197,257
Trainable params: 6,657
Non-trainable params: 48,190,600
_________________________________________________________________
```

Figure 14: Model summary.

Unfreezing the Pretrained Layers

- The code below shows how to unfreeze the pretrained layers:

```
1 # Unfreeze the pretrained layers
2 model.layers[0].trainable = True
```

- Line 2: `model.layers[0].trainable` means whether the parameters of the first layer (the pretrained module) should be trainable (changable).
- The code below shows how to compile the RNN (we need to compile a model after unfreezing its layers):

```
1 # Compile the model
2 model.compile(
3     optimizer=keras.optimizers.Adam(learning_rate=10 ** -4),
4     loss='binary_crossentropy',
5     metrics=['accuracy'])
```

- Line 3: We use `Adam` optimizer with `learning_rate` 0.0001 (one tenth of the previous learning rate).

Model Summary

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 50)                48190600
_________________________________________________________________
dense (Dense)                (None, 128)               6528
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 48,197,257
Trainable params: 48,197,257
Non-trainable params: 0
_________________________________________________________________
```

Figure 15: Model summary.

References

F. A. Gers and J. Schmidhuber.

Recurrent Nets that Time and Count.

In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.

S. Hochreiter and J. Schmidhuber.

Long Short-Term Memory.

Neural computation, 9(8):1735–1780, 1997.