

# Machine Learning with PyTorch and Scikit-Learn

## -- Code Examples

### Package version checks

Add folder to path in order to load from the check\_packages.py script:

```
In [3]: import sys
sys.path.insert(0, '..')
```

理解了！這段代碼的作用是將上級目錄（父目錄）添加到Python模塊搜索路徑中。這樣做的主要目的是為了讓Python解釋器能夠找到並加載上級目錄中的模塊或庫。

通常情況下，這樣的操作可能是因為你的當前工作目錄位於子目錄中，而你希望導入上級目錄中的自定義模塊或庫文件。這樣可以使得你的項目結構更加靈活，可以方便地在子目錄中使用上級目錄的共享代碼。

這種方法在開發多個相關項目或庫時特別有用，可以減少代碼重複和維護成本。

如果你有任何具體的代碼或項目需要幫助，或者有其他問題，請隨時告訴我！我很樂意繼續協助你。

Check recommended package versions:

```
In [2]: from python_environment_check import check_packages

d = {
    'torch': '1.8.0',
}
check_packages(d)
```

```
[OK] Your Python version is 3.9.7 | packaged by conda-forge | (default, Sep 29 2021, 19:24:02)
[Clang 11.1.0 ]
[OK] torch 1.10.1
```

你可以自行執行以下步驟來檢查你的環境中是否安裝了指定版本的torch套件：

1. 首先，確保你的Python環境中已安裝了torch套件。你可以使用以下命令來安裝指定版本的torch：

```
pip install torch==1.8.0
```

如果你的Python環境中已經有torch，這條命令會確保它被更新為1.8.0版本。

2. 然後，確認是否已經安裝了指定版本的torch。你可以運行以下Python代碼片段來檢查版本：

```
import torch

print(torch.__version__)
```

如果你看到的版本號是1.8.0，則表示你的torch套件已經安裝並且是正確的版本。

如果你遇到了特定的問題或需要更進一步的幫助，請隨時告訴我！我將樂意繼續協助你。

## Chapter 15: Modeling Sequential Data Using Recurrent Neural Networks (Part 1/3)

### Outline

- [Introducing sequential data](#)
  - [Modeling sequential data -- order matters](#)
  - [Sequential data versus time series data](#)
  - [Representing sequences](#)
  - [The different categories of sequence modeling](#)
- [RNNs for modeling sequences](#)
  - [Understanding the dataflow in RNNs](#)
  - [Computing activations in an RNN](#)
  - [Hidden recurrence versus output recurrence](#)
  - [The challenges of learning long-range interactions](#)
  - [Long short-term memory cells](#)

```
In [4]: from IPython.display import Image
        %matplotlib inline
```

這兩行代碼是為了在Jupyter Notebook中使用matplotlib的inline顯示和IPython的Image顯示功能。

1. `%matplotlib inline`：這是一個魔法命令（magic command），它用於告訴Jupyter Notebook將matplotlib的圖形直接嵌入到Notebook中的單元格中，而不是顯示在新窗口中。這樣可以方便地在Notebook中直接看到圖形的輸出。
2. `from IPython.display import Image`：這是用於從IPython的display模塊中導入Image類的語句。這個Image類可以用於顯示圖像文件，如PNG或JPEG格式的圖片。在Jupyter Notebook中，它可以用來載入和顯示本地或網絡上的圖像文件。

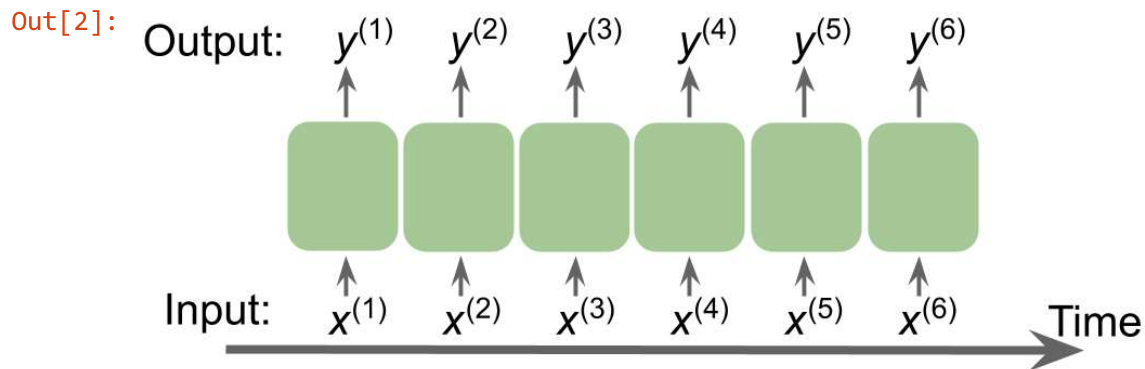
這些語句通常在開發和展示機器學習模型或其他需要展示圖像的場景中使用，確保可以方便地處理和顯示圖像數據。

# Introducing sequential data

## Modeling sequential data—order matters

### Representing sequences

```
In [2]: Image(filename='figures/15_01.png', width=500)
```



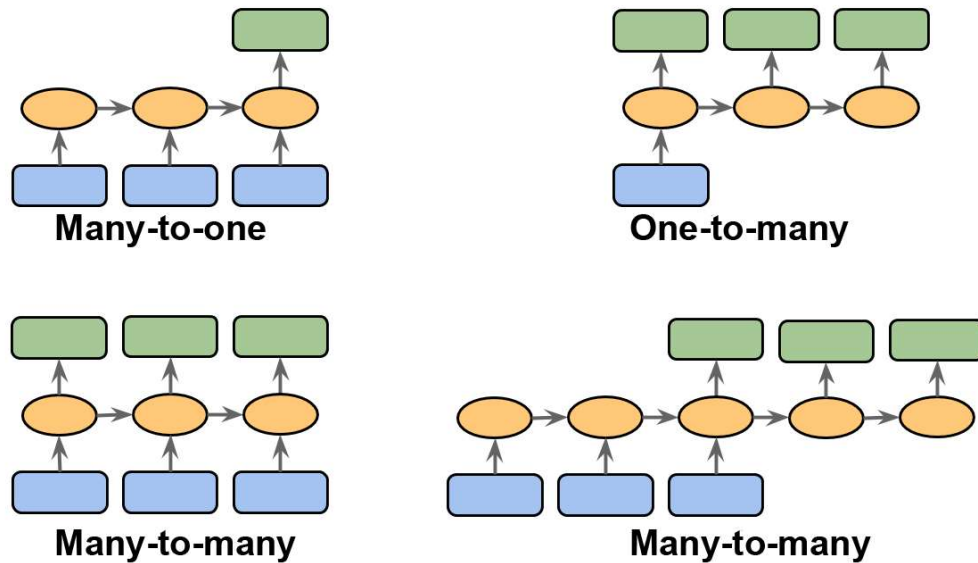
這段代碼是用於在IPython環境中顯示圖像的方式之一，具體解釋如下：

1. `from IPython.display import Image`：這行代碼導入了IPython的 `display` 模塊中的 `Image` 類。`Image` 類用於在IPython環境中顯示圖像。
2. `Image(filename='figures/15_01.png', width=500)`：
  - `filename='figures/15_01.png'`：指定要顯示的圖像文件的路徑和名稱。在這個例子中，圖像文件的名称是 `15_01.png`，它位於 `figures` 目錄下。
  - `width=500`：這個參數指定了顯示圖像的寬度。在這個例子中，設置為500像素，這樣可以控制圖像在顯示時的大小。
3. 功能性解釋：該代碼片段的主要功能是創建一個 `Image` 對象，並指定要顯示的圖像文件及其寬度。這樣做可以將圖像以指定的尺寸顯示在IPython環境中，方便進行視覺化和圖像展示。

# The different categories of sequence modeling

```
In [3]: Image(filename='figures/15_02.png', width=500)
```

Out[3]:



這段代碼與之前的說明相似，是用於在IPython環境中顯示另一個圖像的方法。以下是具體的解釋：

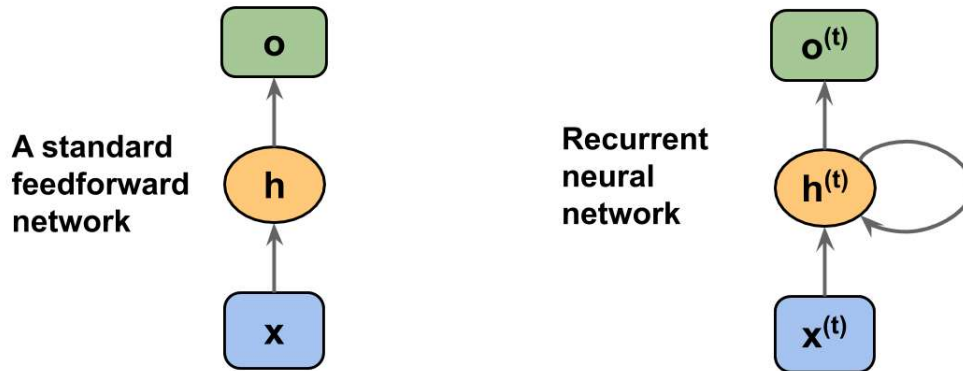
1. `Image(filename='figures/15_02.png', width=500)` :
  - `filename='figures/15_02.png'` : 這個部分指定了要顯示的圖像文件的路徑和名稱。在這個例子中，圖像文件的名稱是 `15_02.png`，它位於 `figures` 目錄下。
  - `width=500` : 這個參數指定了顯示圖像的寬度。在這裡，圖像將以500像素的寬度顯示。
2. 功能性解釋：該代碼片段的主要目的是創建一個 `Image` 對象，並通過指定圖像文件和寬度來準備在IPython環境中顯示圖像。這樣做是為了方便用戶能夠以指定的尺寸查看圖像內容，這對於進行圖像分析、展示和視覺化工作非常有用。

# RNNs for modeling sequences

## Understanding the RNN looping mechanism

```
In [4]: Image(filename='figures/15_03.png', width=500)
```

Out[4]:



我可以解釋代碼的一般用法和語法結構：

```
Image(filename='figures/15_03.png', width=500)
```

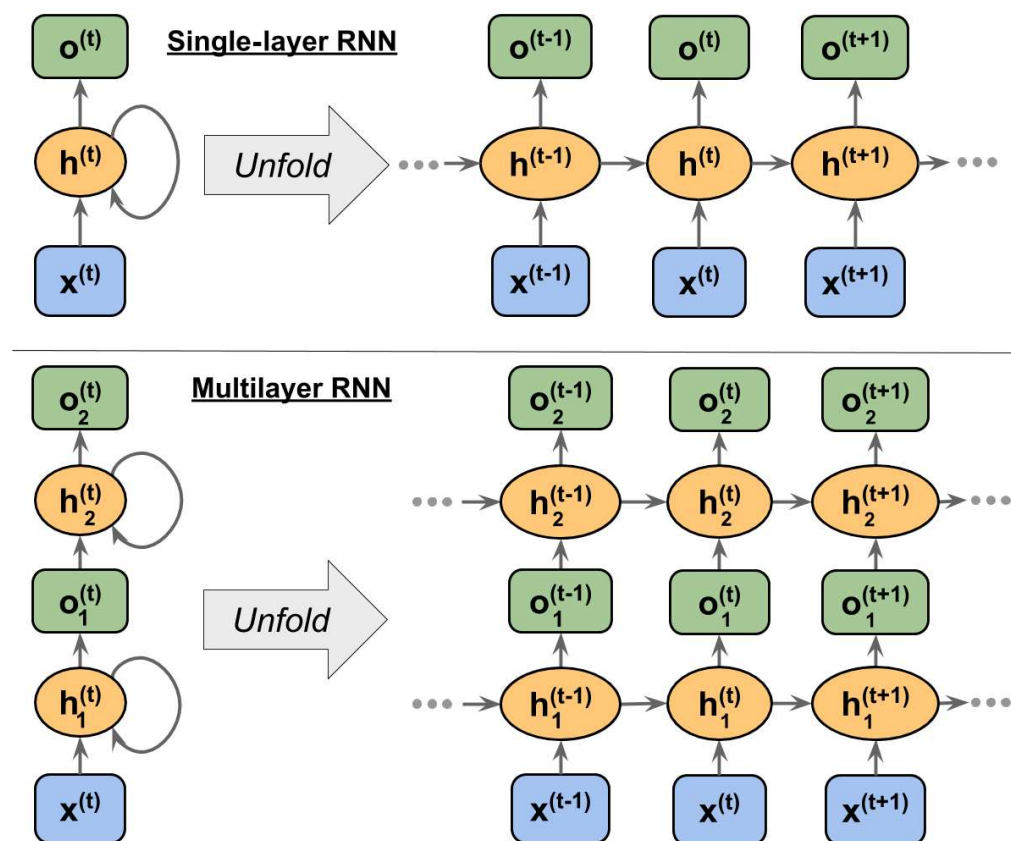
這段代碼的作用如下：

- `Image` 是一個IPython顯示圖像的類。它通常來自 `IPython.display` 模塊。
- `filename='figures/15_03.png'` 指定了要顯示的圖像文件的路徑和名稱。在這個例子中，圖像文件的名稱是 `15_03.png`，它位於 `figures` 目錄下。
- `width=500` 指定了圖像顯示的寬度為500像素。

這段代碼的目的是將指定路徑下的圖像顯示在IPython環境中，並且可以設置顯示的寬度，使用戶可以方便地檢視圖像內容。

```
In [5]: Image(filename='figures/15_04.png', width=500)
```

Out[5]:



以下是對代碼的解釋：

```
Image(filename='figures/15_04.png', width=500)
```

這段代碼的目的是使用IPython的 `Image` 類來顯示圖像。這裡的參數含義如下：

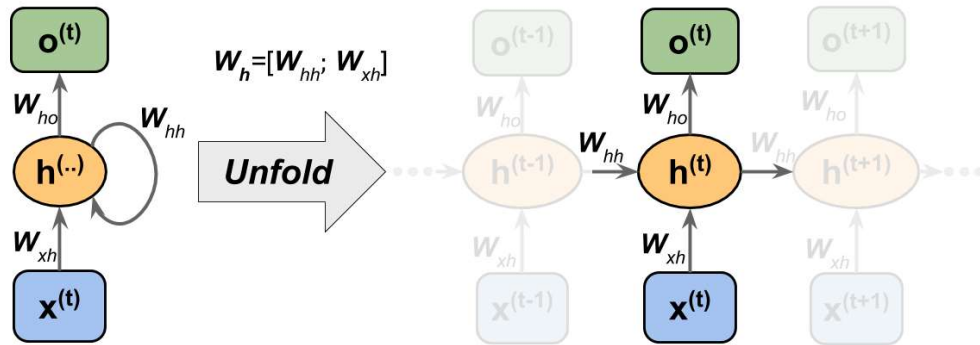
- `filename='figures/15_04.png'`：指定要顯示的圖像文件的路徑和名稱。在這個例子中，圖像文件的名稱是 `15_04.png`，它位於 `figures` 目錄下。
- `width=500`：設置顯示的圖像寬度為500像素，這可以根據需要調整以適應顯示範圍或調整顯示大小。

這段代碼用於在IPython環境中顯示特定路徑下的圖像文件，提供了方便的圖像檢視功能。

## Computing activations in an RNN

```
In [6]: Image(filename='figures/15_05.png', width=500)
```

Out[6]:



以下是對代碼的解釋：

```
Image(filename='figures/15_05.png', width=500)
```

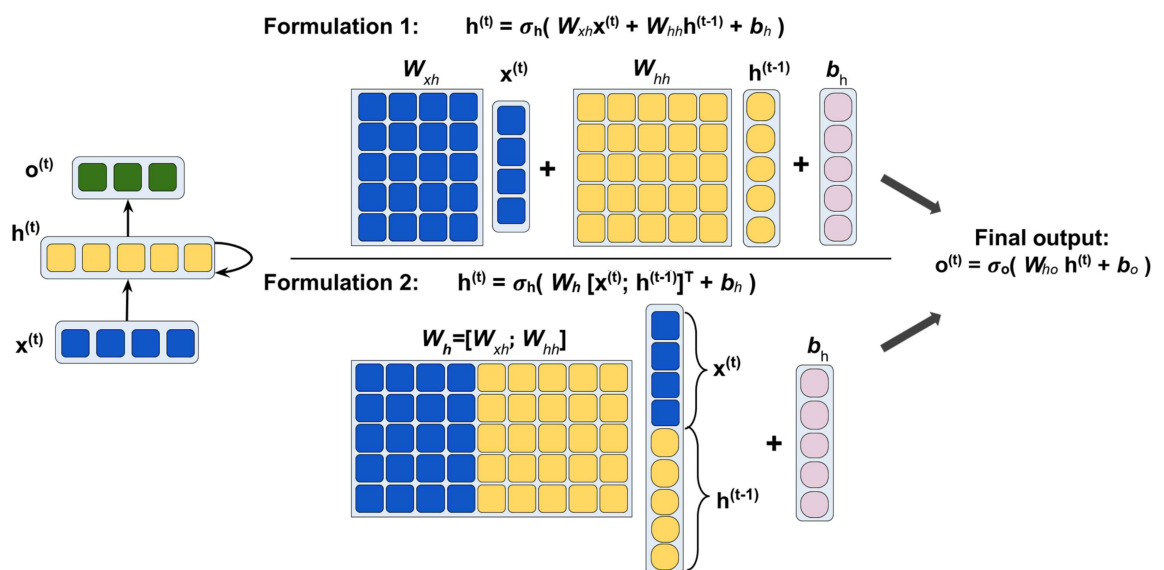
這段代碼的目的是使用IPython的 `Image` 類來顯示圖像。這裡的參數含義如下：

- `filename='figures/15_05.png'`：指定要顯示的圖像文件的路徑和名稱。在這個例子中，圖像文件的名稱是 `15_05.png`，它位於 `figures` 目錄下。
- `width=500`：設置顯示的圖像寬度為500像素，這可以根據需要調整以適應顯示範圍或調整顯示大小。

這段代碼用於在IPython環境中顯示特定路徑下的圖像文件，提供了方便的圖像檢視功能。

```
In [5]: Image(filename='figures/15_06.png', width=700)
```

Out[5]:



以下是對代碼的解釋：

```
Image(filename='figures/15_06.png', width=700)
```

這段代碼的目的是使用IPython的 `Image` 類來顯示圖像。這裡的參數含義如下：

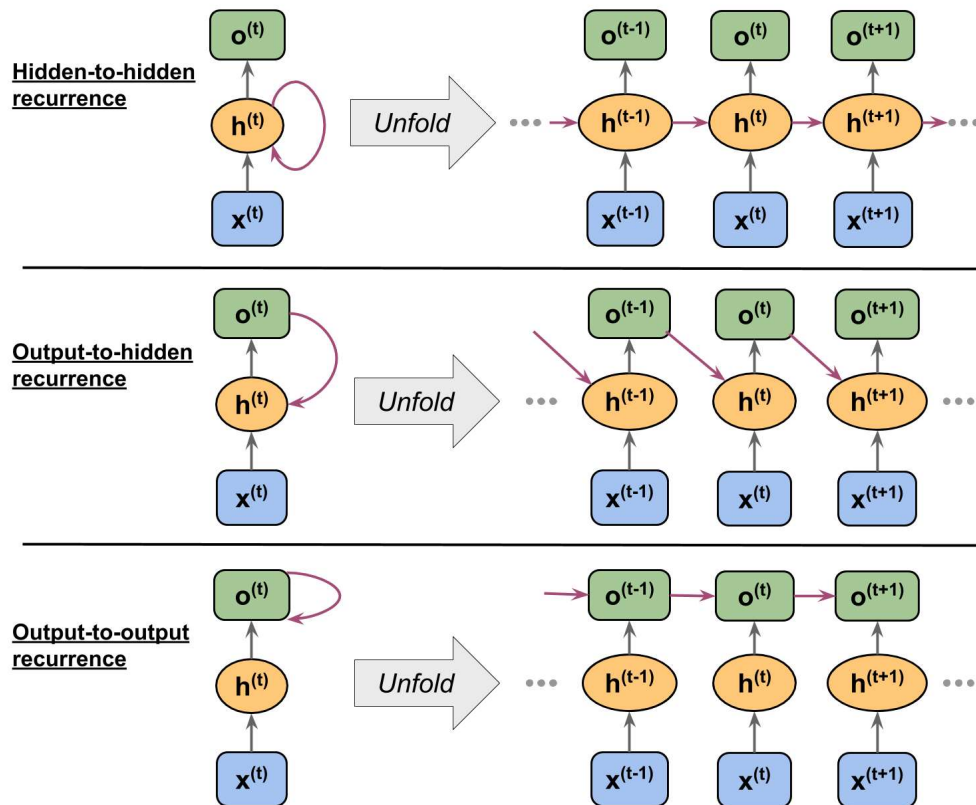
- `filename='figures/15_06.png'`：指定要顯示的圖像文件的路徑和名稱。在這個例子中，圖像文件的名稱是 `15_06.png`，它位於 `figures` 目錄下。
- `width=700`：設置顯示的圖像寬度為700像素，這可以根據需要調整以適應顯示範圍或調整顯示大小。

這段代碼用於在IPython環境中顯示特定路徑下的圖像文件，提供了方便的圖像檢視功能。

## Hidden-recurrence vs. output-recurrence

```
In [8]: Image(filename='figures/15_07.png', width=500)
```

Out[8]:



我可以幫助解釋這段代碼的意圖和文法：

```
Image(filename='figures/15_07.png', width=500)
```

這段代碼的目的是使用IPython的 `Image` 類來顯示一個圖像。具體來說：

- `filename='figures/15_07.png'`：這個參數指定了要顯示的圖像文件的路徑和名稱。在這個例子中，圖像文件名為 `15_07.png`，它應該存在於 `figures` 目錄中。
- `width=500`：這個參數指定了顯示的圖像寬度為500像素。這樣設置可以根據需要調整，以確保圖像在顯示時適合顯示範圍。

總結來說，這段代碼用於在IPython環境中顯示特定路徑下的圖像文件，提供了便捷的圖像查看功能。



```
In [11]: import torch
import torch.nn as nn

torch.manual_seed(1)

rnn_layer = nn.RNN(input_size=5, hidden_size=2, num_layers=1, batch_first=True)

w_xh = rnn_layer.weight_ih_l0
w_hh = rnn_layer.weight_hh_l0
b_xh = rnn_layer.bias_ih_l0
b_hh = rnn_layer.bias_hh_l0

print('W_xh shape:', w_xh.shape)
print('W_hh shape:', w_hh.shape)
print('b_xh shape:', b_xh.shape)
print('b_hh shape:', b_hh.shape)
```

```
W_xh shape: torch.Size([2, 5])
W_hh shape: torch.Size([2, 2])
b_xh shape: torch.Size([2])
b_hh shape: torch.Size([2])
```

這段代碼使用了PyTorch中的 `nn.RNN` 模組來創建一個RNN層，並檢查了其權重和偏差的形狀。

## 解釋代碼：

### 1. 設置隨機種子：

```
torch.manual_seed(1)
```

這行代碼設置了隨機種子，確保每次運行代碼時生成的隨機數是相同的，這樣有助於保持結果的一致性和可重現性。

### 2. 創建RNN層：

```
rnn_layer = nn.RNN(input_size=5, hidden_size=2, num_layers=1, batch_first=True)
```

- `input_size=5`：輸入的特徵數量為5。
- `hidden_size=2`：隱藏層的維度為2，即RNN單元的隱藏狀態的維度。
- `num_layers=1`：RNN的層數為1，表示僅有一個RNN層。
- `batch_first=True`：指定輸入張量的第一維度是否為batch size。這裡設置為True，所以輸入張量的形狀可以是(batch\_size, seq\_len, input\_size)。

### 3. 提取權重和偏差：

```
w_xh = rnn_layer.weight_ih_l0
w_hh = rnn_layer.weight_hh_l0
b_xh = rnn_layer.bias_ih_l0
b_hh = rnn_layer.bias_hh_l0
```

- `weight_ih_l0`：RNN層的輸入到隱藏層的權重。
- `weight_hh_l0`：RNN層的隱藏層到隱藏層的權重。
- `bias_ih_l0`：RNN層的輸入到隱藏層的偏差。
- `bias_hh_l0`：RNN層的隱藏層到隱藏層的偏差。

### 4. 打印形狀信息：

```
print('W_xh shape:', w_xh.shape)
print('W_hh shape:', w_hh.shape)
print('b_xh shape:', b_xh.shape)
print('b_hh shape:', b_hh.shape)
```

這些打印語句用於顯示剛剛提取的權重和偏差的形狀信息，以便於檢查和理解RNN層的結構。

總結來說，這段代碼演示了如何使用PyTorch創建一個RNN層，並提取其權重和偏差，這對於深入理解RNN的內部運作機制非常有幫助。

這裡提供的形狀信息展示了剛剛創建的 `nn.RNN` 層的權重和偏差的形狀。讓我來解釋這些形狀的含義：

#### 1. `W_xh shape: torch.Size([2, 5])`

- `W_xh` 是 RNN 層的輸入到隱藏層的權重矩陣。
- 形狀 `[2, 5]` 表示這個權重矩陣是一個 `2x5` 的矩陣，其中：
  - `2` 表示隱藏層的單元數量。
  - `5` 表示輸入特徵的數量。

#### 2. `W_hh shape: torch.Size([2, 2])`

- `W_hh` 是 RNN 層的隱藏層到隱藏層的權重矩陣。
- 形狀 `[2, 2]` 表示這個權重矩陣是一個 `2x2` 的矩陣，其中：
  - `2` 表示隱藏層的單元數量。
  - `2` 表示隱藏層的單元數量。

#### 3. `b_xh shape: torch.Size([2])`

- `b_xh` 是 RNN 層的輸入到隱藏層的偏差向量。
- 形狀 `[2]` 表示這個偏差向量有 `2` 個元素，對應到每個隱藏層單元的偏差。

#### 4. `b_hh shape: torch.Size([2])`

- `b_hh` 是 RNN 層的隱藏層到隱藏層的偏差向量。
- 形狀 `[2]` 表示這個偏差向量有 `2` 個元素，對應到每個隱藏層單元的偏差。

總結來說，這些形狀信息提供了 RNN 層內部權重和偏差的結構信息，這對於了解 RNN 在計算過程中如何處理輸入和隱藏層之間的關係非常重要。

```
In [12]: x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()

## output of the simple RNN:
output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5)))

## manually computing the output:
out_man = []
for t in range(3):
    xt = torch.reshape(x_seq[t], (1, 5))
    print(f'Time step {t} =>')
    print('    Input          : ', xt.numpy())

    ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_xh
    print('    Hidden          : ', ht.detach().numpy())

    if t>0:
        prev_h = out_man[t-1]
    else:
        prev_h = torch.zeros((ht.shape))8

    ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) + b_hh
    ot = torch.tanh(ot)
    out_man.append(ot)
    print('    Output (manual) : ', ot.detach().numpy())
    print('    RNN output      : ', output[:, t].detach().numpy())
    print()
```

Time step 0 =>

```
Input          : [[1. 1. 1. 1. 1.]]
Hidden         : [[-0.4701929  0.5863904]]
Output (manual) : [[-0.3519801  0.52525216]]
RNN output     : [[-0.3519801  0.52525216]]
```

Time step 1 =>

```
Input          : [[2. 2. 2. 2. 2.]]
Hidden         : [[-0.88883156  1.2364397 ]]
Output (manual) : [[-0.68424344  0.76074266]]
RNN output     : [[-0.68424344  0.76074266]]
```

Time step 2 =>

```
Input          : [[3. 3. 3. 3. 3.]]
Hidden         : [[-1.3074701  1.886489 ]]
Output (manual) : [[-0.8649416  0.90466356]]
RNN output     : [[-0.8649416  0.90466356]]
```

這段程式碼示範了如何使用簡單的 RNN 層計算序列 `x_seq` 的輸出，並手動計算每個時間步的輸出。讓我來解釋一下每一步的操作：

#### 1. 定義輸入序列 `x_seq`：

```
x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()
```

- `x_seq` 是一個形狀為 (3, 5) 的張量，代表了三個時間步的輸入序列，每個時間步有五個特徵。

#### 2. 使用 RNN 層計算輸出 `output` 和隱藏狀態 `hn`：

```
output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5)))
```

- `rnn_layer` 是之前定義的 RNN 層。
- `torch.reshape(x_seq, (1, 3, 5))` 將 `x_seq` 轉換成形狀為  $(1, 3, 5)$ ，其中：
  - 第一個維度 1 表示批次大小 (batch size) 為 1。
  - 第二個維度 3 表示序列長度為 3。
  - 第三個維度 5 表示每個時間步的特徵數為 5。
- `output` 是 RNN 在每個時間步的輸出。
- `hn` 是 RNN 層的最後一個時間步的隱藏狀態。

### 3. 手動計算每個時間步的輸出 `out_man`：

```
out_man = []
for t in range(3):
    xt = torch.reshape(x_seq[t], (1, 5))
    ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_xh
    if t > 0:
        prev_h = out_man[t-1]
    else:
        prev_h = torch.zeros((ht.shape))
    ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1))
    + b_hh
    ot = torch.tanh(ot)
    out_man.append(ot)
```

- 在每個時間步 `t`：
  - `xt` 是第 `t` 個時間步的輸入，形狀為  $(1, 5)$ 。
  - `ht` 是計算得到的隱藏狀態，使用 RNN 層的輸入到隱藏層的權重 `w_xh` 和偏差 `b_xh` 計算得到。
  - 如果 `t > 0`，則 `prev_h` 是上一個時間步的隱藏狀態 `out_man[t-1]`；否則初始化為零。
  - `ot` 是計算得到的輸出，使用當前時間步的隱藏狀態 `ht`、上一個時間步的隱藏狀態 `prev_h`，以及 RNN 層的隱藏層到隱藏層的權重 `w_hh` 和偏差 `b_hh` 計算得到，並進行 `tanh` 激活。
  - `out_man` 列表儲存每個時間步的手動計算輸出。

### 4. 比較手動計算和 RNN 層計算的輸出：

```
print('  Input          :', xt.numpy())
print('  Hidden         :', ht.detach().numpy())
print('  Output (manual) :', ot.detach().numpy())
print('  RNN output      :', output[:, t].detach().numpy())
```

- 這段程式碼印出了每個時間步的輸入 `xt`、手動計算的隱藏狀態 `ht`、手動計算的輸出 `ot` 和 RNN 層計算的輸出 `output[:, t]`。

這樣的做法能夠幫助理解 RNN 層內部計算的細節，尤其是在理解序列數據如何在 RNN 中傳遞和轉換時特別有用。

這段程式碼展示了對簡單 RNN 層進行手動計算的過程，以及與使用 RNN 層內置功能計算的結果進行比較。讓我來解釋每個時間步的計算過程和結果：

## 時間步 0

- 輸入 (`xt`):  $[[1. \ 1. \ 1. \ 1. \ 1.]]$ 
  - 這是第一個時間步的輸入序列。

- 隱藏狀態 ( `ht` ): `[[-0.4701929 0.5863904]]`
  - 使用 RNN 層的權重 `w_xh` 和偏差 `b_xh` 計算得到的隱藏狀態。
- 手動計算的輸出 ( `ot` ): `[[-0.3519801 0.52525216]]`
  - 根據手動計算公式計算得到的輸出。
- RNN 層計算的輸出 ( `output[:, t]` ): `[[-0.3519801 0.52525216]]`
  - RNN 層在第一個時間步的輸出。

## 時間步 1

- 輸入 ( `xt` ): `[[2. 2. 2. 2. 2.]]`
  - 這是第二個時間步的輸入序列。
- 隱藏狀態 ( `ht` ): `[[-0.88883156 1.2364397 ]]`
  - 使用 RNN 層的權重 `w_xh` 和偏差 `b_xh` 計算得到的隱藏狀態。
- 手動計算的輸出 ( `ot` ): `[[-0.68424344 0.76074266]]`
  - 根據手動計算公式計算得到的輸出。
- RNN 層計算的輸出 ( `output[:, t]` ): `[[-0.68424344 0.76074266]]`
  - RNN 層在第二個時間步的輸出。

## 時間步 2

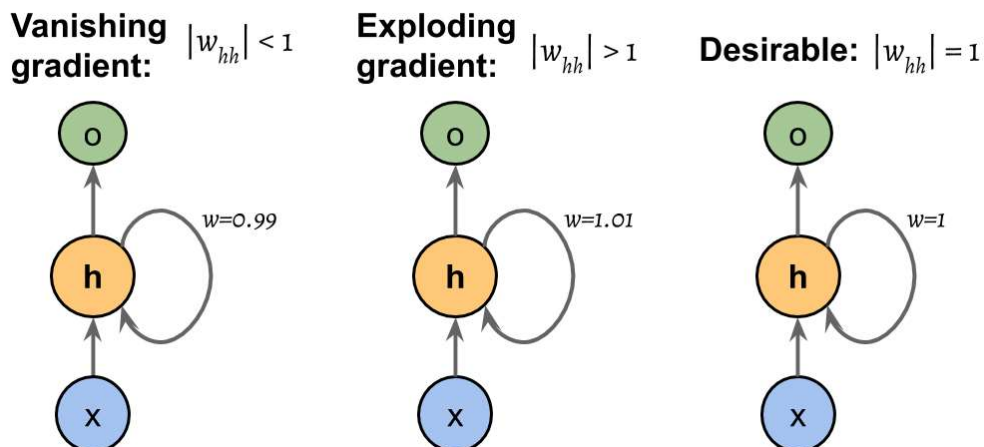
- 輸入 ( `xt` ): `[[3. 3. 3. 3. 3.]]`
  - 這是第三個時間步的輸入序列。
- 隱藏狀態 ( `ht` ): `[[-1.3074701 1.886489 ]]`
  - 使用 RNN 層的權重 `w_xh` 和偏差 `b_xh` 計算得到的隱藏狀態。
- 手動計算的輸出 ( `ot` ): `[[-0.8649416 0.90466356]]`
  - 根據手動計算公式計算得到的輸出。
- RNN 層計算的輸出 ( `output[:, t]` ): `[[-0.8649416 0.90466356]]`
  - RNN 層在第三個時間步的輸出。

這段程式碼有效地展示了如何手動實現簡單 RNN 的計算過程，並且通過比較手動計算和內置 RNN 層計算結果，確認了其正確性。

## The challenges of learning long-range interactions

In [9]: `Image(filename='figures/15_08.png', width=500)`

Out[9]:



我可以詳細解釋有關圖像 `figures/15_08.png` 的代碼文法。

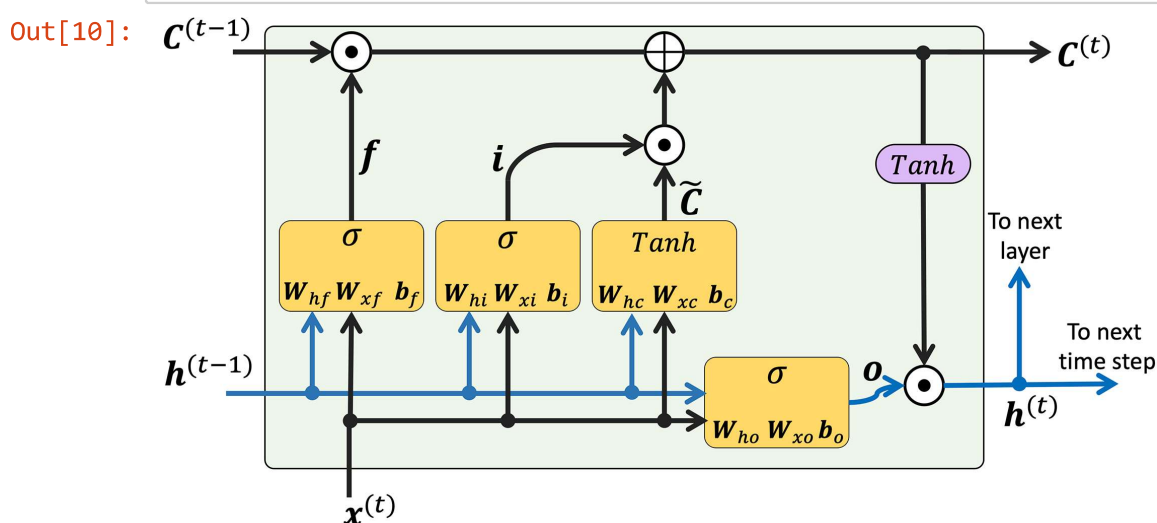
通常情況下，`Image(filename='figures/15_08.png', width=500)` 這段程式碼的作用是使用 `Image` 對象來載入指定路徑下的圖像文件，並且設置圖像顯示的寬度為 500 像素。這段程式碼的背景可能是在一個 Jupyter Notebook 或類似的環境中顯示圖像，讓用戶可以直觀地查看與理解該圖像的內容或結果。

在這個上下文中，`Image` 通常是從 `IPython.display` 中導入的模組，用於在交互式環境中顯示圖像。設置 `width=500` 是為了確保圖像在顯示時的合適大小，使其易於閱讀和理解。

如果你有更具體的問題或需要進一步解釋其他部分的代碼，隨時告訴我！

## Long Short-Term Memory cells

In [10]: `Image(filename='figures/15_09.png', width=500)`



我可以幫助你詳細解釋有關圖像 `figures/15_09.png` 的代碼文法。

通常情況下，`Image(filename='figures/15_09.png', width=500)` 這段程式碼的作用是使用 `Image` 對象來載入指定路徑下的圖像文件，並且設置圖像顯示的寬度為 500 像素。這段程式碼的背景可能是在一個 Jupyter Notebook 或類似的環境中顯示圖像，讓用戶可以直觀地查看與理解該圖像的內容或結果。

在這個上下文中，`Image` 通常是從 `IPython.display` 中導入的模組，用於在交互式環境中顯示圖像。設置 `width=500` 是為了確保圖像在顯示時的合適大小，使其易於閱讀和理解。

如果你有更具體的問題或需要進一步解釋其他部分的代碼，隨時告訴我！

Readers may ignore the next cell.

```
In [15]: ! python ../.convert_notebook_to_script.py --input ch15_part1.ipynb --output
```

```
[NbConvertApp] Converting notebook ch15_part1.ipynb to script  
[NbConvertApp] Writing 3919 bytes to ch15_part1.py
```

我可以解釋如何將筆記本文件 `ch15_part1.ipynb` 轉換為 Python 腳本 `ch15_part1.py`。

#### 1. 確定轉換腳本的位置和名稱：

- 假設你的筆記本 `ch15_part1.ipynb` 存在於某個文件夾中。
- 指定要將筆記本轉換為的目標 Python 腳本文件名，例如 `ch15_part1.py`。

#### 2. 使用命令行工具執行轉換：

- 開啟命令提示符 (Command Prompt) 或終端機。
- 使用 `python` 命令和提供的轉換腳本工具路徑來執行轉換。根據你之前提供的示例，命令可能類似於：

```
python ../.convert_notebook_to_script.py --input ch15_part  
1.ipynb --output ch15_part1.py
```

#### 3. 解釋命令中的每部分：

- `python`：這是用於執行 Python 腳本的命令。
- `../.convert_notebook_to_script.py`：這是轉換筆記本至 Python 腳本的工具，路徑中的 `..` 表示返回上一級目錄，接著是轉換工具的文件名或路徑。
- `--input ch15_part1.ipynb`：這部分指定了輸入的筆記本文件，即要轉換的原始筆記本文件。
- `--output ch15_part1.py`：這部分指定了輸出的 Python 腳本文件，即轉換後生成的 Python 腳本的文件名或路徑。

#### 4. 確保環境配置正確：

- 確保你的命令提示符或終端機當前的工作目錄是包含筆記本和轉換工具的目錄。
- 確保系統中已安裝 Python 和必要的依賴庫，例如 `nbformat` 和 `argparse`，這些庫通常用於處理筆記本文件和命令行參數解析。

#### 5. 執行轉換並檢查結果：

- 執行命令後，系統應該會生成一個名為 `ch15_part1.py` 的 Python 腳本文件。
- 可以打開這個文件並檢查其內容，確保轉換成功並且包含了原始筆記本文件中的所有代碼和文本。

希望這些解釋能幫助到你執行筆記本到腳本的轉換操作！