深度學習 - 第十章

Status	Book
Created time	@January 9, 2025 3:06 PM

第十章

10-1 各種時間序列任務

時間序列是具固定時間間隔的資料。時間序列無處不在，不論是自然現象或是人類活動的模式都存在該類型的資料。處理該類資料需要理解一個系統的動態運作方式:即該系統的週期性循環、它如何隨時間變化以及該系統的常規行為等。目前為止，最常見的時間序列任務是預測任務:即預測一個序列中接下來會出現甚麼(本章的重點)。不過實際上，還有很多其他種類的時間序列任務:

- 分類:為一個時間序列指定一個或多個類別標籤。
- 事件偵測:在連續的資料流中，識別出特定的預期事件是否發生。常見的應用:熱詞偵測
- 異常偵測:偵測連續資料流中發生的任何異常情形。

10-2 溫度預測任務

本章的所有內容都是為了解決一個問題:在給定每小時量測值之時間序列下，預測未來24小時的溫度。密集連接網路和卷積神經網路並不具備處理這種資料集的能力，而循環神經網路(RNN)則能在這種類型的問題上發揮作用。

```
#使用德國耶拿(jena)的馬克斯.普朗克生物地球化學研究院氣象站紀錄的天氣時間序列資
#記錄了以十分鐘為區間，14個不同量測值(溫度、壓力...)數年的紀錄結果。
#https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
```

```
import os
fname = os.path.join(r"C:\Users\11146023\Downloads\jena_climate_2009_2016.cs

with open(fname) as f:
    data=f.read()
lines = data.split("\n")
#取得標頭
header = lines[0].split(",")
#取得紀錄內容
lines =lines[1:]
#output:標頭內容
print(header)


#output:資料筆數
print (len(lines))
```

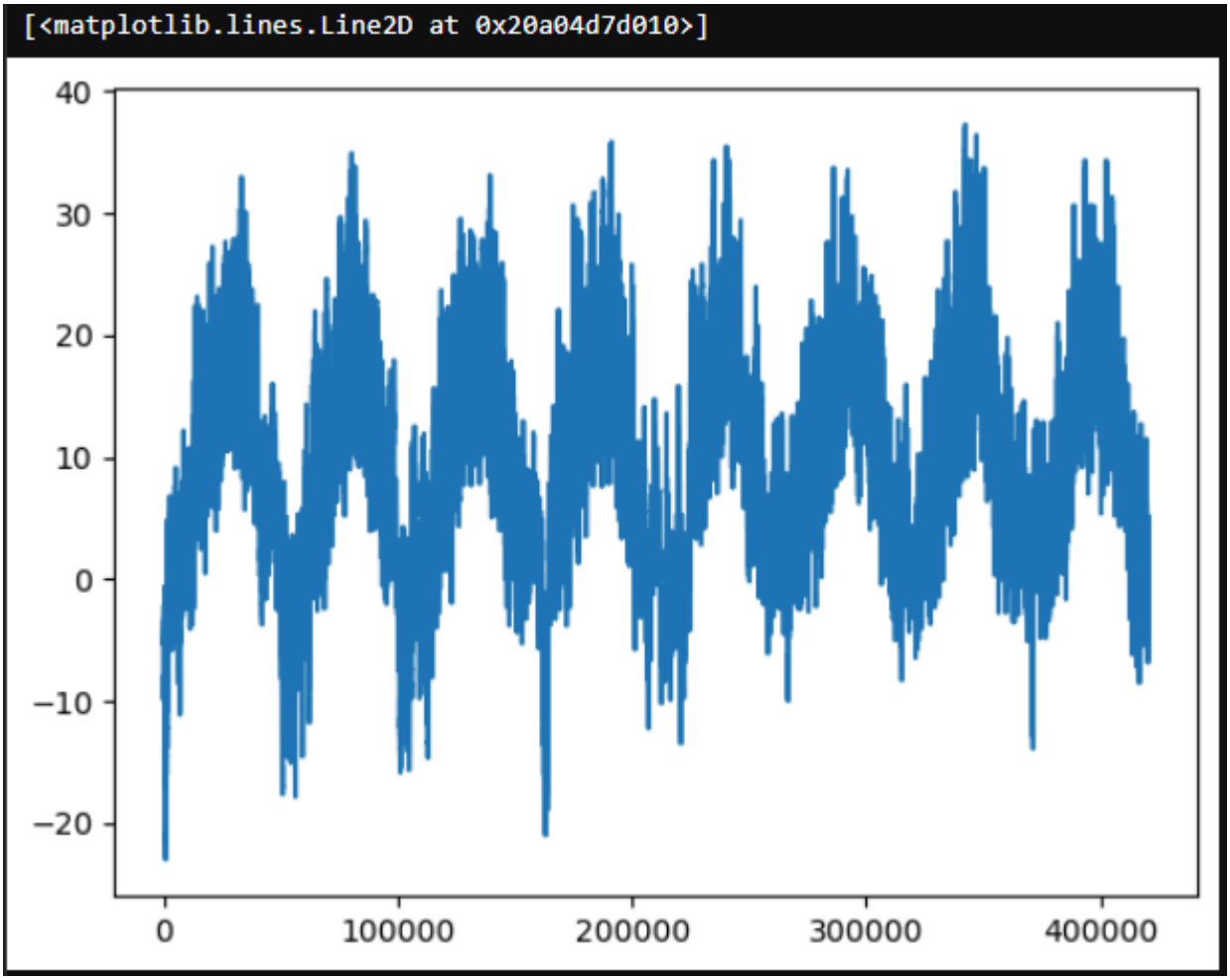
```
['"Date Time"', '"p (mbar)"', '"T (degC)"', '"Tpot (K)"', '"Tdew (degC)"', '"rh (%)"', '"VPmax (mbar)"', '"VPact (mbar)"', '"VPdef (mbar)"', '"sh (g/kg)"',
'"H2OC (mmol/mol)"', '"rho (g/m**3)"', '"wv (m/s)"', '"max. wv (m/s)"', '"wd (deg)"']
420451
```

```
#將資料轉換成兩個NumPy陣列

import numpy as np

#創建兩個空的NumPy陣列
temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines),len(header)-1))

#走訪不同資料
for i, line in enumerate(lines):
#取出首個欄位(Date Time欄位)以外的資料，並存成values陣列
    values = [float(x) for x in line.split(",")[1:]]

#取出溫度欄位的資料，並存進temperature陣列
```

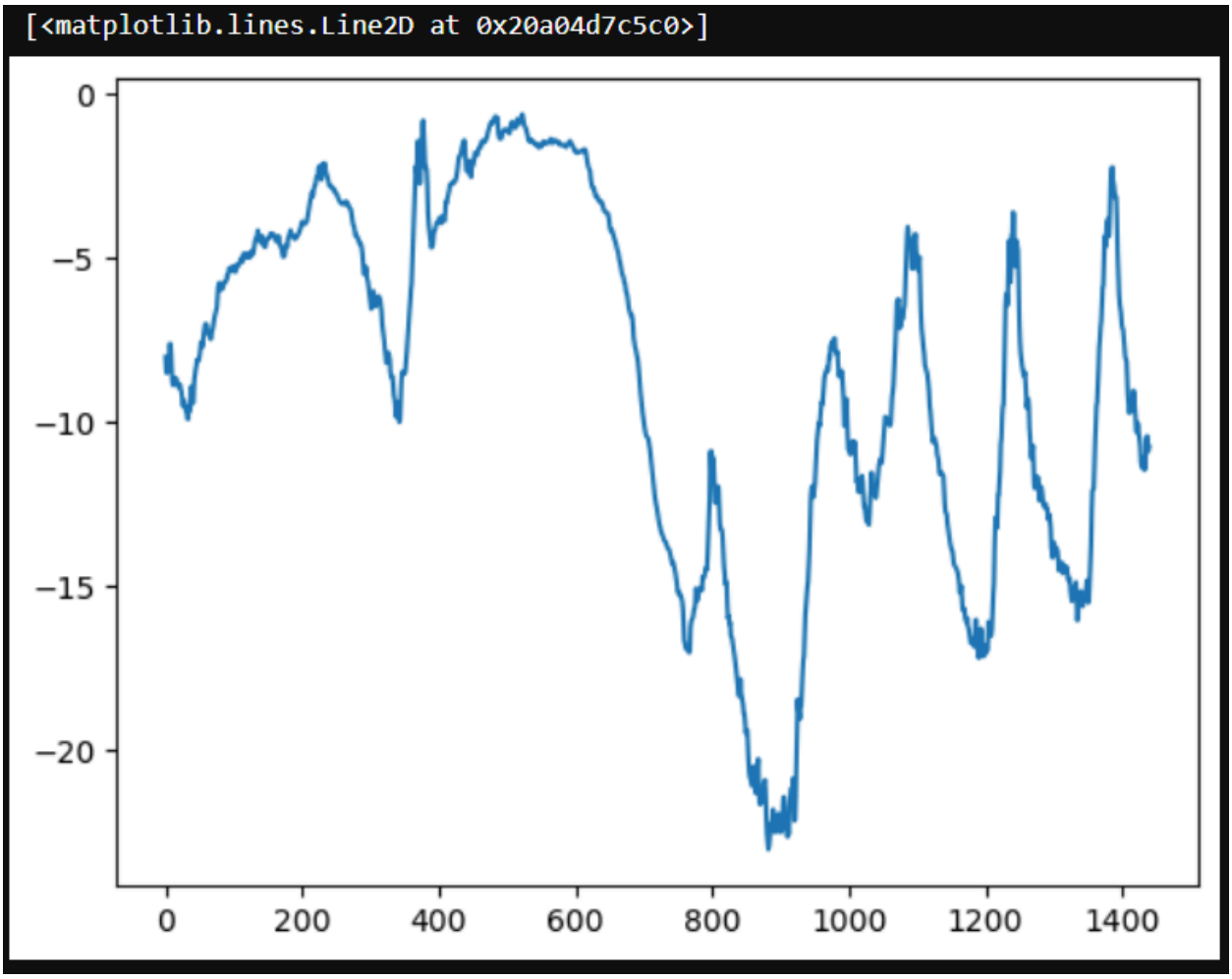
```
    temperature[i] = values[1]
#把values中的所有資料(包含溫度)存進raw_data陣列中
    raw_data[i, :] = values[:]
```

```
#繪製資料集中溫度的變化曲線

from matplotlib import pyplot as plt
plt.plot(range(len(temperature)),temperature)
```

```
[<matplotlib.lines.Line2D at 0x20a04d7d010>]
```




```
#繪製前10天溫度的時間序列

plt.plot(range(1440),temperature[:1440])
```



從上圖中可以看到前面10天的溫度變化大致是以天為循環週期

記得在資料中尋找週期性

在處理時間序列時，驗證資料和測試資料發生的時間點應該比訓練資料晚，因為我們是在嘗試根據過去預測未來。不過也有少數例外，因為對某些問題而言(最短路徑問題:終點固定，起點可變)，有時把時間軸倒過來反而會簡單很多。

```
#計算每個資料子集的樣本數

#50%的資料用來訓練
num_train_samples = (0.5 * len(raw_data))
#25%的資料用來驗證
num_val_samples = int(0.25 * len(raw_data))
#剩下的資料用來測試
```

```
num_test_samples = len(raw_data)-num_train_samples-num_val_samples
#output
print("num_train_samples:",num_train_samples)
print("num_val_samples:",num_val_samples)
print("num_test_samples:",num_test_samples)
```

```
num_train_samples: 210225.5
num_val_samples: 105112
num_test_samples: 105113.5
```

10-2-1 準備資料

問題:若給定前5天的資料(每小時採樣一次)，能否預測24小時候的溫度?

由於資料集中儲存的是數值資料，所以不用再做任何的向量化(one-hot編碼)。不過由於特徵資料具有不同尺度(mbar(值約為1000);mmol/mol(值為3))，因此還需要正規化，讓它們都落在相似尺度的小數值上。

```
#使用前210225個時步作為訓練資料
num_train_samples = int(num_train_samples)
mean = raw_data[:num_train_samples].mean(axis=0)
raw_data -= mean
std = raw_data[:num_train_samples].std(axis=0)
raw_data /=std
```

接著創建一個Dataset物件，它每次可生成一批次的資料。由於批次內各筆資料間有較多重複內容，為所有的批次資料都分配記憶體會很浪費。因此會即時生成批次資料，並只將原始的raw_data陣列和temperature陣列保留在記憶體中。Keras內建的timeseries_dataset_from_array()可以處理上述問題。

timeseries_dataset_from_array()函式介紹:基本上只需要提供一個時間序列資料的陣列(透過傳遞data參數)，timeseries_dataset_from_array()就能傳回，從該時間序列中，不同位置所萃取出的窗格(window,稱它為"序列",sequence)，序列的長度由sequence_length參數所決定。

Ex: data=[0 1 2 3 4 5 6]，sequence_length=3，timeseries_dataset_from_array()就會生成樣本為:[0 1 2]、[1 2 3]、[2 3 4]、[3 4 5]、[4 5 6]

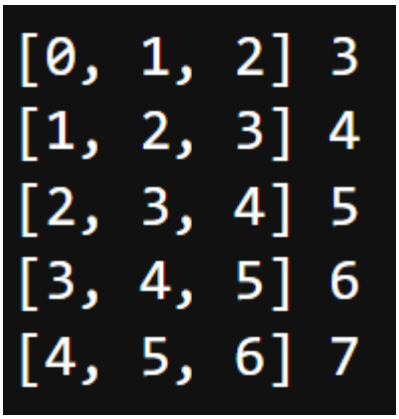
也可以傳遞targets參數(儲存著目標值的陣列)給timeseries_dataset_from_array()，用來指定在生成包含多個樣本(序列，sequence)的批次時也同時生成每個樣本所對應的目標值。在進行時間序列預測時，targets的內容通常是源自於data陣列，不過會與其對應的樣本有一些位移量

Ex:when data=[0 1 2 3 4 5 6.....]，sequence_length=3時，就可以透過傳遞targets=[3 4 5 6....]來創建一個資料集，讓演算法學習如何預測接下來出現的目標值
(!!!Note:timeseries_dataset_from_array()所傳回的Dataset物件，在形式上是一個資料集，但在每次跟它要資料時，會動態生成並傳回一批次的樣本及目標值)

```
import numpy as np
from tensorflow import keras
#生成一個0到9，排序過的整數陣列
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
#生成的序列將從[0 1 2 3 4 5 6]採樣
data=int_sequence[:-3],
#data[N]開始的序列之目標值，會是data[N+3]
targets=int_sequence[3:],
#序列的長度是3
sequence_length=3,
#每次傳回包含2個序列的批次
batch_size=2,
)
#每次讀取一批次資料，直到全部取完
for inputs,targets in dummy_dataset:
#走訪目前批次中的每一筆資料
for i in range(inputs.shape[0]):
print([int (x) for x in inputs[i]],int(targets[i]))
```

第二行為生成的序列;第四行為對應的目標值

預設會由前往後依序生成序列，但也可用shuffle=True來打亂




```
#實例化訓練集、驗證集和測試、
sampling_rate = 6
sequence_length = 120
delay = sampling_rate *(sequence_length + 24 - 1)
batch_size = 256

train_dataset = keras.utils.timeseries_dataset_from_array(
data=raw_data[:-delay],
targets=temperature[delay:],
sampling_rate=sampling_rate,
sequence_length=sequence_length,
#打亂Dataset傳回序列的順序
shuffle=True,
batch_size=batch_size,
start_index=0,
end_index=num_train_samples)

val_dataset = keras.utils.timeseries_dataset_from_array(
raw_data[:-delay],
targets=temperature[delay:],
sampling_rate=sampling_rate,
sequence_length=sequence_length,
shuffle=True,
batch_size=batch_size,
start_index=num_train_samples,
```

```
end_index=num_train_samples + num_val_samples)

test_dataset = keras.utils.timeseries_dataset_from_array(
raw_data[:-delay],
targets=temperature[delay:],
sampling_rate=sampling_rate,
sequence_length=sequence_length,
shuffle=True,
batch_size=batch_size,
start_index=num_train_samples + num_val_samples)
```

參數解釋:

- sampling_rate=6：觀測結果以每小時一個資料點進行採樣，也就是只會保留原始資料集中六分之一的資料點。(單位:十分鐘)
- sequence_length=120：每個序列樣本包含連續五天的資料(單位:小時)
- delay=sampling_rate*(sequence_length+24-1)：每個序列樣本的目標值，就是該序列結束後24小時的溫度(Ex:第0個序列樣本的內容是原始資料的前120筆資料，而其目標值則為原始資料的第120+23筆資料)

每個資料集在讀取時都會動態生成一個tuple : (samples,targets)。資料集內的序列樣本已隨機打亂過，所以批次中的兩個連續序列(ex:samples[0]和samples[1])在時間上不一定很接近

```
#檢視訓練集的輸出情形
for samples , targets in train_dataset:
    print("samples shape:", samples.shape)
    print("targets shape:" ,targets.shape)
    break
```

#samples中包含256個樣本(序列)，其中每個樣本紀錄了120個小時的資料，而每個資料點有著14個量測值

```
samples shape: (256, 120, 14)
targets shape: (256,)
```

10-2-2 一個符合常識、非機器學習的基準線(baseline)

在使用模型預測溫度之前先試試一個簡單、常識性的做法。這個做法可被視為一種完整性檢查，並使我們具備一個基準線:我們必須超越它，才能確保更進階的機器學習模型有在發揮作用。當面臨新問題(目前沒有解決方案)，這種常識性的基準線會很有用。Ex:

不平衡的分類任務:有一些類別的樣本比其他類別普遍很多。如果資料集中有90%屬於類別A，而類別B只佔10%。常識性的預測做法就是在遇到新樣本時，永遠都預測為類別A。這樣分類準確度就能達到90%，而任何更好的方法都應該超越90%以證明它有用。有時候這種很基本的基準線會出乎意料的難以超越。在這個案例中可以安心假設每天在同一時間點的溫度是連續且緩慢變化的。因此，一個符合常識的作法是:預測24小時後的溫度將等於現在的溫度。

```
#計算基準線的平均絕對誤差(MAE)
def evaluate_naive_method(dataset):
    total_abs_err = 0.
    samples_seen = 0
    for samples,targets in dataset:
        preds = samples[:, -1 , 1] * std[1] + mean[1]
#加總各預測值與目標值的絕對誤差
        total_abs_err += np.sum(np.abs(preds - targets))
#加總已處理序列樣本的數量
        samples_seen += samples.shape[0]
#除以總樣本數，計算平均絕對誤差
    return total_abs_err / samples_seen

print(f"Validation MAE:{evaluate_naive_method(val_dataset): .2f}")
print(f"Test MAE:{evaluate_naive_method(test_dataset): .2f}")
```

preds = samples[: , -1 , 1] * std[1] + mean[1]

:代表批次中所有的串列；-1代表串列中最後一小時的天氣資料；1代表天氣資料中的第1個元素(溫度)

mean[1] : 溫度特徵位於第1欄，所以simples[: , -1 , 1]是批次中每個序列最後一小時的溫度。因為已經對特徵進行正規化，若要取回攝氏溫度，要先把它乘以標準差再加回平均值

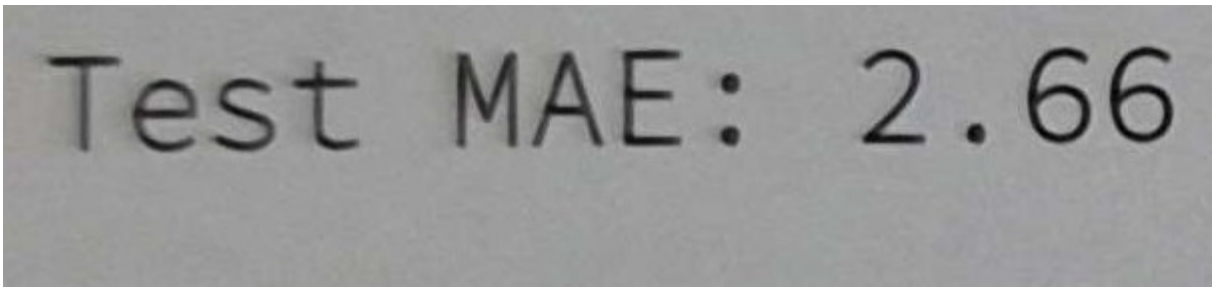
```
Validation MAE: 2.44
Test MAE: 2.62
```

嘗試運用深度學習知識來取得更好的結果

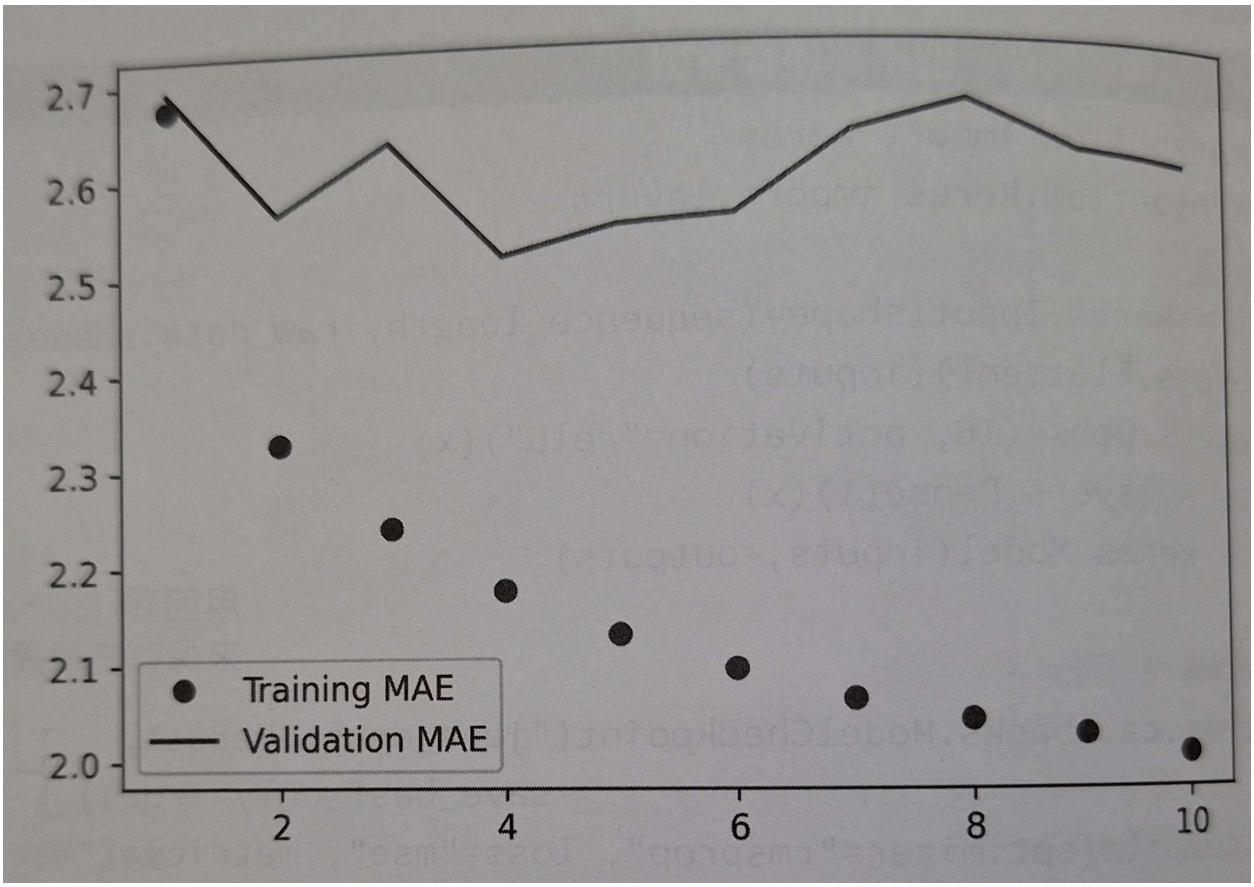
10-2-3 嘗試基本的機器學習模型

與使用機器學習方法之前先建立基準線一樣，探究複雜、運算量高的模型之前，先試試簡單、運算量低的模型。

```
#全連接模型
#步驟:先扁平化資料，接著通過兩個Dense層
#最後一個Dense層沒有激活函數，這在迴歸問題中是很典型的做法
#使用MSE而不是MAE作為損失函數的原因是因為它在零的附近會是平滑的，這對梯度
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)
#用回呼(callbacks)物件來保存表現最好的模型
callbacks = [
keras.callbacks.ModelCheckpoint("jena_dense.keras",
save_best_only=True)]
model.compile(optimizer="rmsprop",loss="mse",metrics=["mae"])
history = model.fit(train_dataset,
epochs=10,
validation_data=val_dataset,
callbacks=callbacks)
#重新載入最佳模型，並用測試資料來評估
model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]: .2f)}")

#未完成
```




```
import matplotlib.pyplot as plt
loss = history.history["mae"]
val_loss = history.history["val_mae"]
epochs = range(1, len(loss) +1)
plt.figure()
plt.plot(epochs,loss,"bo",label="Training MAE")
plt.plot(epochs,val_loss,"b",label="Validation MAE")
plt.title("Training and validation MAE")
plt.legend()
plt.show()
#未完成
```



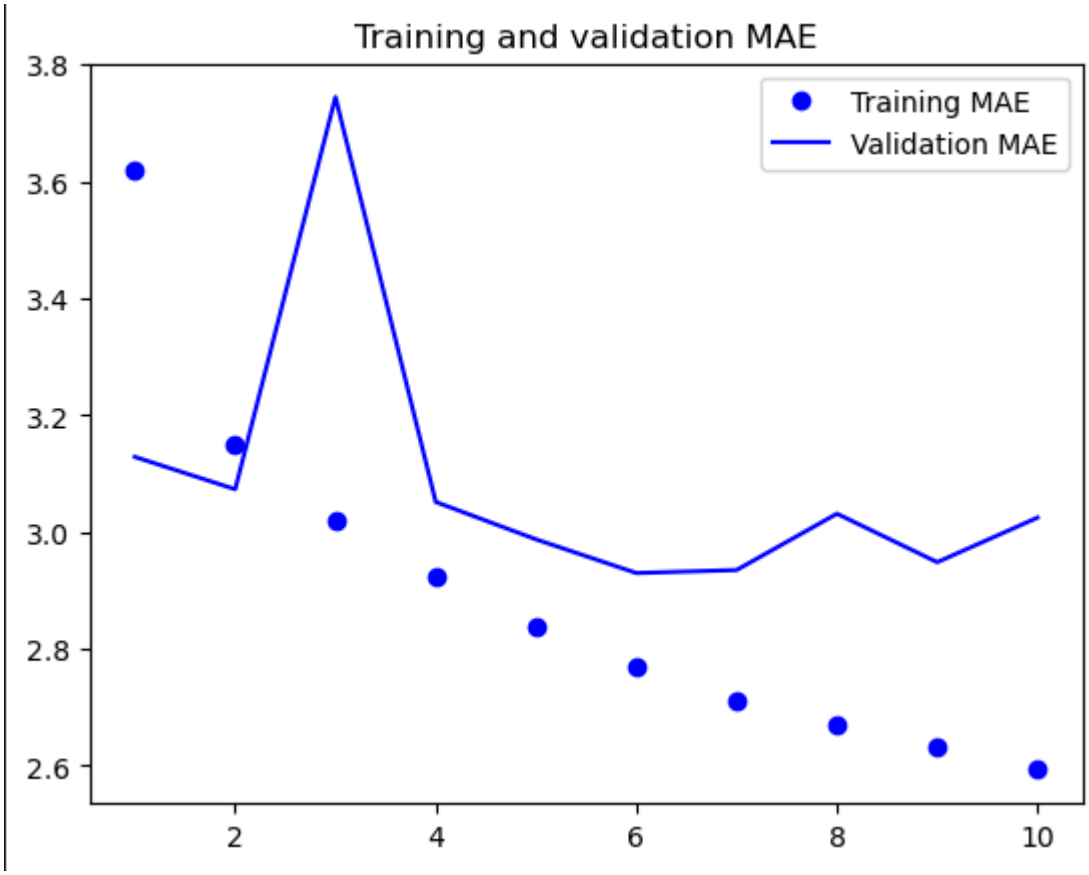
如果存在一個簡單、表現良好的模型(基準線)，能讓我們直接從資料一路通往目標，為甚麼在訓練模型時卻無法發現它?這是因為所蒐尋的模型空間(假設空間)是所有可能的雙層網路空間。而符合基準線或更好的模型，可能只是這個空間內數百萬種模型的其中一個。因此就算假設空間中確實存在好的模型，也不代表就能透過梯度下降法找到它。這就是機器學習普遍存在的一個明顯限制:除非演算法是透過全面搜尋法來尋找特定的優良模型，否則它有時甚至無法找到一個簡單問題的解決方案。這就是為甚麼好的特徵工程和模型架構的先驗知識這麼重要:必須精準的告訴模型它應該尋找的是甚麼

10-2-4 嘗試使用1D卷積模型

既然輸入序列以天為週期，或許卷積模型就能發揮作用(卷積模型適合用來找出資料中的pattern)時間卷積網路可以橫跨不同天重複使用相同表示法，就像空間卷積網路能在影像的不同位置使用相同表示法一樣。先前說過Conv2D和SeparableConv2D會透過在2D網路上滑動的小窗格來檢視輸入。而Conv1D層靠的是在輸入序列上滑動的1D窗格;Conv3D(SeparableConv3D層並不存在)則是在輸入立體資料上滑動的3D窗格。因此可以用跟2D卷積層非常相似的方式建構出1D卷積神經網路，它們十分適用於任何具平移不變性的序列資料

```
#將Conv1D層應用於溫度預測，初始窗格長度為24，這樣就能每次檢視24小時的資料
from tensorflow import keras
from tensorflow.keras import layers
inputs = keras.Input(shape=(sequence_length,raw_data.shape[-1]))
#Conv1D(過濾器數量,過濾器的窗格長度)
x = layers.Conv1D(8,24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
#經過降採樣後，窗格長度變成12
x = layers.Conv1D(8,12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
#經過降採樣後，窗格長度變成6
x = layers.Conv1D(8,6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs,outputs)

callbacks = [
keras.callbacks.ModelCheckpoint("jena_conv.keras",
save_best_only=True)
]
model.compile(optimizer="rmsprop",loss="mse",metrics=["mae"])

history = model.fit(train_dataset,
epochs=10,
validation_data=val_dataset,
callbacks=callbacks)
model = keras.models.load_model("jena_conv.keras")
#output
print(f"Test MAE:{model.evaluate(test_dataset)[1]:.2f}")
```

```
Test MAE:3.14
```



Val_MAE約等於2.9

該驗證模型的驗證MAE與基準線相差甚遠，甚至比密集連接模型表現的還差，原因如下:

- 在天氣資料中僅有某些特性具備平移不變性。雖然資料確實有以一天為一個循環的特點，但是早上資料跟晚上或半夜資料的特性卻不同。天氣資料只在某些特定的時間尺度上具有平移不變性
- 若要預測明天的溫度，今天的溫度資料會比5天前的資料更具參考價值，而1D卷積神經網路卻沒辦法利用這項事實。尤其是最大池化及全局平均池化層在很大程度上破壞了順序資訊

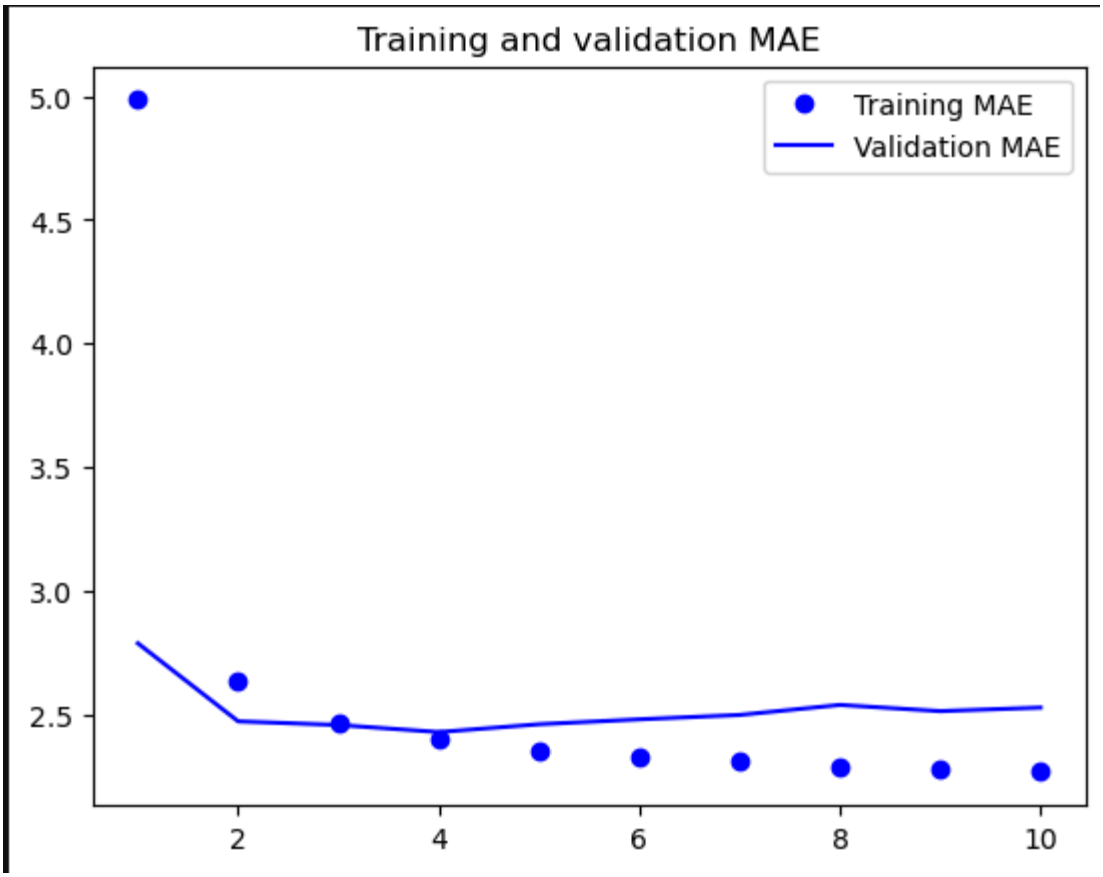
10-2-5 循環神經網路的基準線

雖然密集連接層和卷積網路都沒辦法做得更好，但不代表機器學習不適用於這個問題。在密集連接網路的做法中，第一步就將時間序列扁平化，這樣會喪失輸入資料中的時間順序資訊。卷積神經網路也使用相同的方式處理每段資料，甚至還用了池化層。在溫度案例中資料的本質是一個序列，其內各筆資料的前後因果關係和順序都很重要。有一系列專門為

此設計的神經網路架構，也就是循環神經網路(RNN)。其中長短期記憶(LSTM)層是長久以來都非常受歡迎的RNN神經層

```
#使用LSTM層的簡單模型
inputs = keras.Input(shape=(sequence_length,raw_data.shape[-1]))
#創建有著16個神經單元的LSTM層
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)

model = keras.Model(inputs,outputs)
callbacks = [
   keras.callbacks.ModelCheckpoint("jena_conv.keras",
                           save_best_only=True)
]
model.compile(optimizer="rmsprop",loss="mse",metrics=["mae"])

history = model.fit(train_dataset,
                epochs=10,
                validation_data=val_dataset,
                callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
#output
print(f"Test MAE:{model.evaluate(test_dataset)[1]:.2f}")
```

```
Test MAE:2.52
```



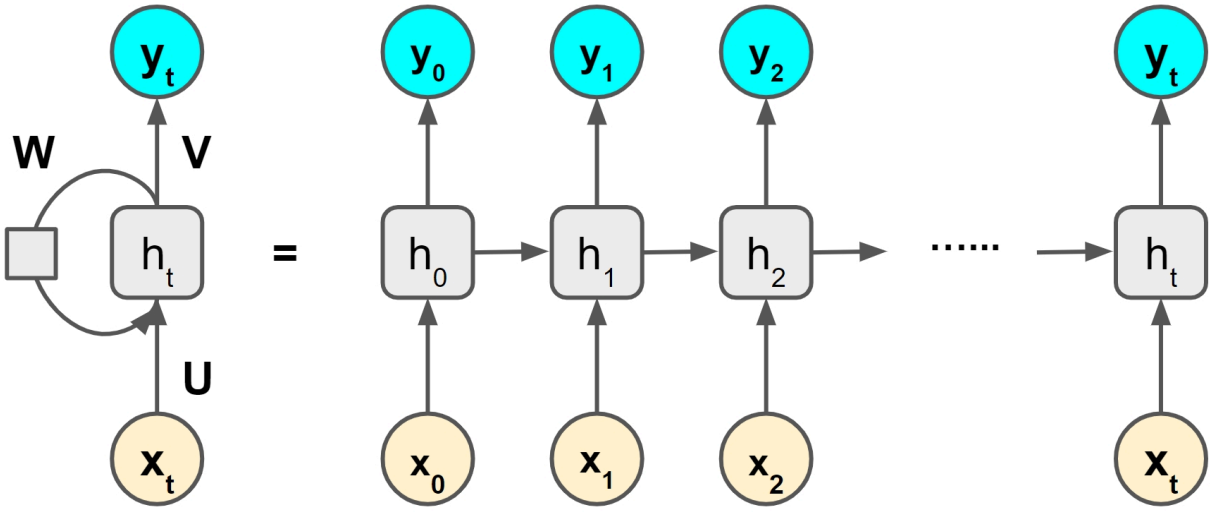
Val_MAE約等於2.35

使用LSTM層的模型終於打敗了基準線，證明了機器學習在這項任務上的價值

10-3 認識循環神經網路(recurrent neural networks)

前面所學的神經網路都有一個主要的特徵，就是它們都沒有記憶(不會記住之前輸入資料的任何訊息)。它們會獨立處理每一筆輸入，而沒有保留不同輸入之間的任何關聯狀態。對這樣的網路來說，為了能處理資料點間的時序關係，必須一次性向網路展示整個序列，也就是把它變成單一資料點。Ex:在前面密集連接網路中，就把5天的資料扁平化成一個大向量，並一次性進行處理，這樣的網路被稱為前饋式神經網路。然而在讀一個句子時，通常會逐字處理並同時保留對前面內容的記憶。這樣可以更流暢的理解句子的意義。生物智能在處理資訊時是漸進的，並且會保留一個記憶處理內容的內部模型。該模型是根據過去的資訊而建立，並且會隨著所接收的新資訊而不斷更新。

循環神經網路(RNN):採用的就是上述的原理，透過對序列元素進行迭代來處理序列，並保留迄今所見相關資訊的狀態。RNN是一種具有內部迴路的神經網路



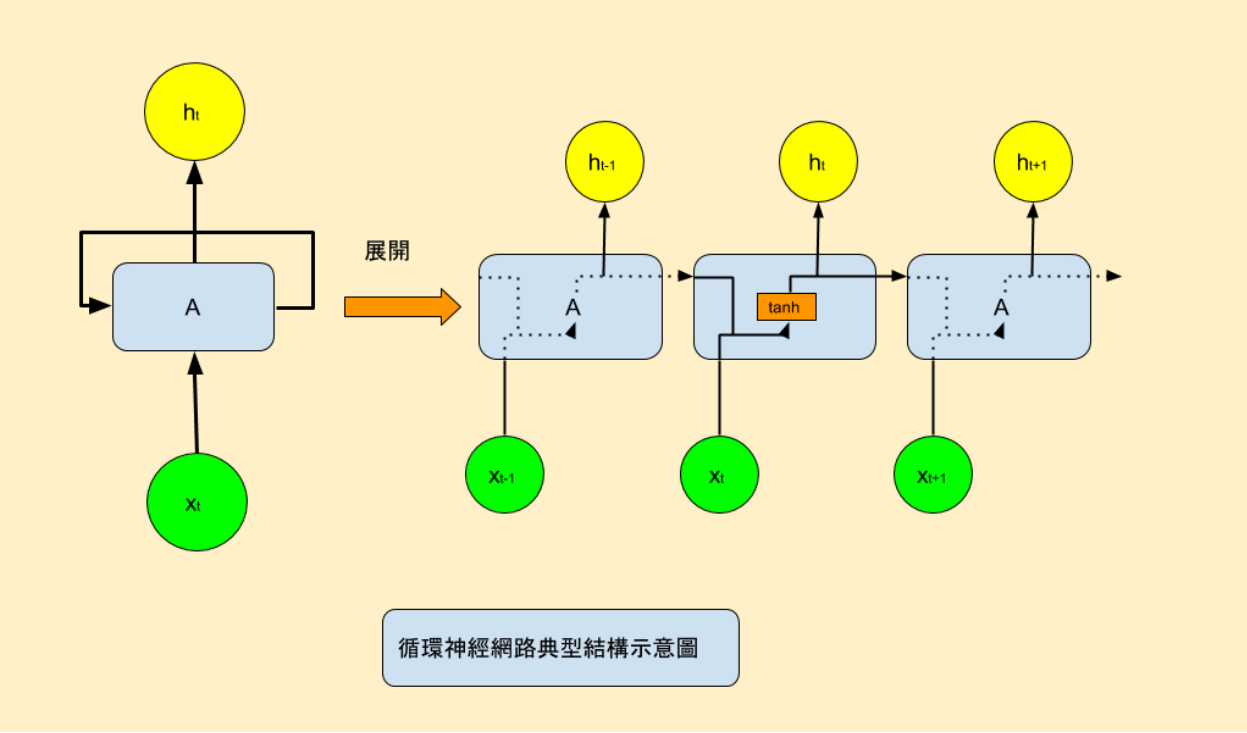
在處理兩個不同且獨立的序列(如一個批次中的兩個樣本)時，RNN的狀態會重置，所以仍把一個序列看成單一個資料點，也就是網路的單一輸入。不同的是，這個資料點不再用單一步驟一次性處理;相反地，RNN會對序列的內部元素進行迭代。

```
#詳細的RNN虛擬碼:以一個向量序列作為輸入，把它編碼成大小為(timesteps,input_fe
#接著會將該輸出設為下一步的狀態。對第一個時步來說，由於沒有之前的輸出，也就不

#在t時的狀態
state_t = 0
#對序列中的元素進行迭代
for input_t in input_sequence:
output_t = activation(dot(W,input_t) + dot(U, state_t)+ b)
#將輸出值設為下一個迭代的狀態
state_t = output_t
```

```
#RNN的NumPy程式碼
import numpy as np
#輸入序列中的時步數
timesteps = 100
#輸入特徵空間的維度
input_features = 32
#輸出特徵空間的維度
```

```
output_features = 64
#輸入資料:隨機數值
inputs = np.random.random((timesteps, input_features))
#初始狀態:一個全零向量
state_t = np.zeros((output_features,))
#W、U創建隨機權重矩陣
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
#創建隨機偏值向量
b = np.random.random((output_features, ))
successive_outputs = []
#input_t是一個shape為(input_features,)的向量
for input_t in inputs:
#合併輸入跟當前狀態(也就是之前的輸出)，獲取當前的輸出(使用tanh來增加非線性特
    output_t = np.tanh(np.dot(W,input_t) + np.dot (U,state_t ) + b)
    #將輸出存進一個串列
    successive_outputs.append(output_t)
    #更新下一個時步的狀態
    state_t = output_t
#最後的輸出是一個2軸張量，shape為(timesteps,output_features)
final_output_sequence = np.stack(successive_outputs,axis = 0)
```

簡單來說RNN就是會重複使用前一次迭代的輸出值的for迴圈。其特點在於它的步驟函數



10-3-1 Keras中的循環層

使用NumPy實作出的運算過程會對應到Keras中的一種神經層:SimpleRNN層。這兩者有細微的不同之處:與其它Keras層一樣，SimpleRNN處理的是批次的序列，而非如NumPy實作中的單一序列。表示它接收的是shape為(batch_size,timesteps,input_features)的輸入，而非(timesteps,input_features)

!!!在指定初始Input()的shape參數，可以把timesteps設置為None，這樣網路就可以處理任意長度的序列

```
#一個能處理任意長度序列的RNN層
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

Keras中的所有循環層(SimpleRNN、LSTM、GRU)都能在兩種不同的模式下運行:它們可以傳回每個時步中，連續輸出的完整序列(shaper=(batch_size,timesteps,output_features));或只傳回每個輸入序列的最終輸出(shape=(batch_size,output_features))。這兩種模式是由return_sequences參數所控制。

```
#只傳回最終輸出的RNN層
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps,num_features))
#return_sequences預設值為False
outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
#output
print(outputs.shape)
```

```
(None, 16)
```

```
#傳回完整輸出序列的RNN層
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps,num_features))
outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)

print(outputs.shape)
```

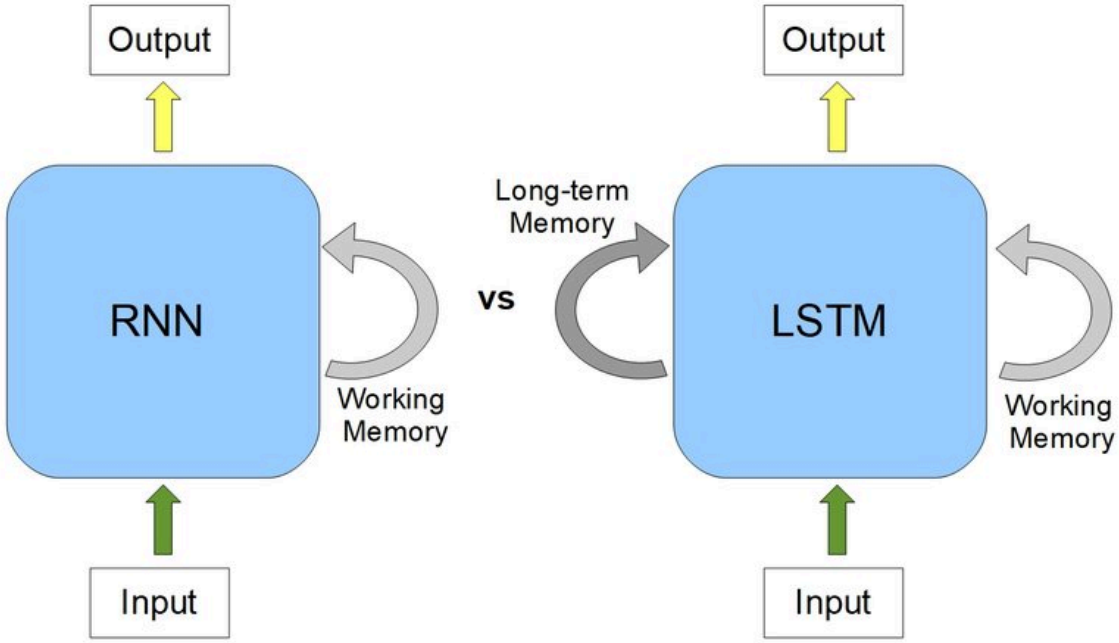
```
(None, 120, 16)
```

```
#堆疊RNN層
inputs = keras.Input(shape=(steps,num_features))
#所有中間層的return_sequences參數都要設為True，以傳回完整的輸出序列
x=layers.SimpleRNN(16,return_sequences=True)(inputs)
x=layers.SimpleRNN(16,return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

10-3-2 Keras中的LSTM及GRU層

實際案例中很少會用到SimpleRNN層，因為它有一個大問題:理論上它應該要能保留許多時步之前看過的輸入資訊，但在實際案例中，序列中的長期依存性被證明是無法學習到

的。這是由梯度消失問題所導致的，它跟非循環網路(前饋式網路)中觀察到的深層情況相似:當不斷添加層數，網路最終會變得無法訓練。LSTM是研究梯度消失問題的最終成果，它是SimpleRNN層的一種變形，透過加入了新方法來承載橫跨許多時步的資訊。LSTM在做的事:儲存資訊以供稍後使用，避免先前的信號在處理過程中逐漸消失，與第9章學到的殘差連接幾乎是相同的概念



一個RNN單元的規格決定了假設空間，但它不能決定單元的作用，因為這取決於單元的權重。因此最好將構成RNN單元的操作組合，看作是搜索模型組態時所加上的

一組限制，而非工程意義上的某項功能設計。總結來說，身為人類不需要了解任何關於LSTM單元的具體架構，要做的就只有記住LSTM單元的功能:允許在未來重新使用過去的資訊，從而對抗梯度消失的問題。

10-4 循環神經網路的進階運用

本節內容將探討進階RNN的應用

- 循環丟棄法:丟棄法的一種變形，用來對抗循環層中的過度配適
- 堆疊多個循環層:增加模型的表徵能力(代價是更高的運算量)

- 雙向循環層:將序列內資料分別以正向及反向的順序，來訓練2個結構相同的RNN層，然後再將兩者學到的成果合併，以提高準確度並減輕記憶喪失的問題

10-4-1 使用循環丟棄法來對抗過度配適

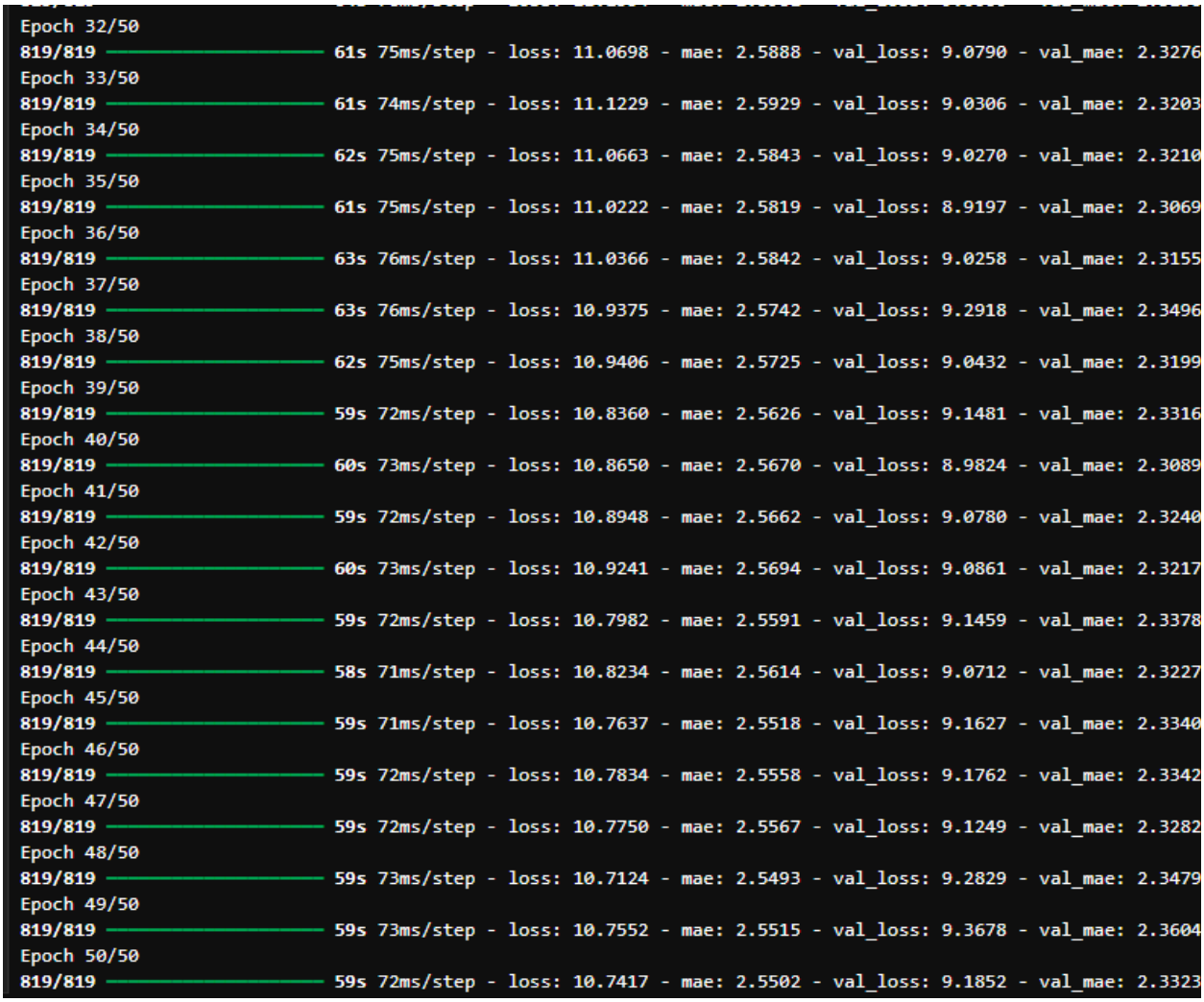
丟棄法:會隨機把神經層的部分輸入單元變成零，以破壞訓練資料間的偶然關聯性，但是要在循環網路中正確使用丟棄法並沒有那麼簡單。Yarin Gal在他的博士論文中確立了循環網路中使用丟棄法的正確方式:在每個時步上都應該採用相同的丟棄遮罩(也就是相同的丟棄單元模式)，而非在不同時步上使用隨機變換的丟棄遮罩。為了常規化GRU和LSTM等神經層的循環閘所學到的表示法，應該在該層內部的循環激活結果上，使用一個恆定的丟棄遮罩(循環丟棄遮罩)。在每個時步上使用相同的丟棄遮罩，可以讓網路正確地隨著時間來反向傳播學習誤差。相反地，隨機的丟棄遮罩則會擾亂這種誤差信號，對學習過程造成不良影響。

Keras中的每種循環層都有兩個跟丟棄法相關的參數

1. dropout:浮點數，用來指定輸入單元的丟棄率
2. recurrent_dropout:浮點數，用於指定循環單元的丟棄率

```
#訓練並評估使用丟棄法的常規化LSTM
inputs = keras.Input(shape=(sequence_length,raw_data.shape[-1]))
#丟棄25%的循環單元
x=layers.LSTM(32,recurrent_dropout=0.25)(inputs)
#為了將Dense層常規化，在LSTM層之後添加了一個Dropout層(丟棄輸入Dense層的5
x=layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model= keras.Model(inputs,outputs)
callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop",loss="mse",metrics=["mae"])

history = model.fit(train_dataset,
#由於使用丟棄法的網路需更多的訓練才能完全收斂，所以使用50個週期數進行訓練
                    epochs=50,
```

```
            validation_data=val_dataset,
            callbacks=callbacks)
```



驗證MAE達到了2.33(基準線為2.44，改善了9%左右)

RNN的運行效能:在GPU上使用Keras的LSTM或GRU層時，神經層將會使用cuDNN核。它是經過高度優化、由NVIDIA所提供的低階演算法實作。cuDNN雖然速度很快但卻缺乏彈性。這或多或少會強迫只能使用NVIDIA所支援的功能。Ex:LSTM和GRU的cuDNN核不支援循環丟棄法，因此若把它添加到層中，則會自動退回或成為普通的TensorFlow實作，以致運行時間比純粹的GPU運行多出2-5倍。當無法使用cuDNN時，可以嘗試展開(untrolling)RNN層的內部迴圈以進行加速。不過這種作法會大大增加RNN的記憶體消耗

量，因此只在序列相對較小(100個時步左右)的情況下是可行的。!!!只有在模型提前知道時步數量的情況下(初始Input()的shape不含None)，才能使用這種方法。運作方法如下:

```
#sequence_length不能是None
inputs = keras.Input(shape=(sequence_length,num_features))
#使用展開法(unrolling)
x=layers.LSTM(32,recurrent_dropout=0.2,unroll=Ture)(inputs)
```

10-4-2 堆疊循環層

堆疊循環層是建立更強大循環神經網路的經典方法。Ex:Google翻譯演算法就是由7個大型LSTM層堆疊而成的。

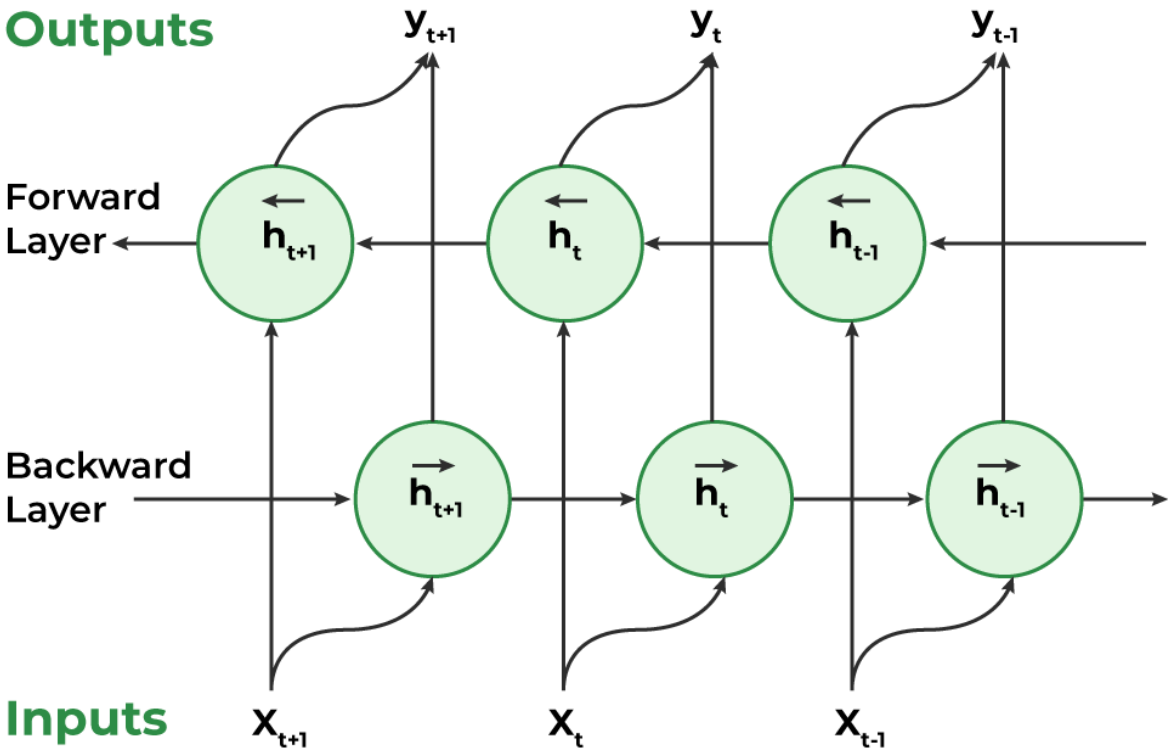
```
#訓練並評估使用了丟棄法的GRU堆疊模型
#GRU跟LSTM非常相似，可以把它視為LSTM架構的精簡版本

inputs = keras.Input(shape=(sequence_length,raw_data.shape[-1]))
x=layers.GRU(32,recurrent_dropout=0.5,return_sequences=True)(inputs)
x=layers.GRU(32,recurrent_dropout=0.5)(x)
x=layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model= keras.Model(inputs,outputs)
callbacks = [
   keras.callbacks.ModelCheckpoint("jena_stacked_gru_dropout.keras",
                           save_best_only=True)
]
model.compile(optimizer="rmsprop",loss="mse",metrics=["mae"])

history = model.fit(train_dataset,
                epochs=50,
                validation_data=val_dataset,
                callbacks=callbacks)
model = keras.models.load_model("jena_acked_gru_dropout.keras")
print(f"Test MAE:{model.evaluate(test_dataset)[1]:.2f}")
```

Test MAE=2.39(比基準線好8.8%)

10-4-3 使用雙向RNN(bidirectional RNN)

雙向RNN是一種常見的RNN變形，經常被用於自然語言處理，可以說它是自然語言處理領域的萬用瑞士刀。RNN十分仰賴順序，它們會依序處理其輸入序列的時步，因此隨機打亂或反轉時步，都會完全改變RNN從序列中萃取到的表示法。雙向RNN充分利用了RNN的順序敏感性:它會使用兩個常規的RNN層，各從一個方向(順時間軸或逆時間軸)處理輸入序列，並整合得到的表示法。

使用反向序列訓練的LSTM表現極差，甚至不如基準線。代表對於LSTM而言依時序處理是成功與否的關鍵。因為相比於較久遠的資料，LSTM層通常在記憶近期的資料上會有更好的表現。然而對於其他問題而言就不是這樣了。Ex:某單字對於理解句子所扮演的重要性，通常不取決於它在句子中的位置。在文字資料中，逆時序與順時序一樣好用。就算倒著讀文字也不會有問題。重要的是，使用反向序列訓練RNN所學到的表示法，會跟使用正向序列所學到的表示法不同。在機器學習的世界中不同但有用的表示法都是值得利用的，而且差異越大越好。因為它們可以提供觀察資料的新視角，並捕捉其他方法所遺漏的資料面向，因此有助於提升效能。這就是模型集成背後的概念。




```
#Bidirectional層會創建該循環層的第二個獨立實例
inputs = keras.Input(shape=(sequence_length,raw_data.shape[-1]))
#Bidirectional層第一個引數是循環層的實例(Ex:layers.GRU or layers.LSTM)
x=layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model= keras.Model(inputs,outputs)

model.compile(optimizer="rmsprop",loss="mse",metrics=["mae"])

history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset)
```

它的表現並沒有普通LSTM來的好是因為所有預測能力一定都來自於網路中依時序處理的那一半，因為已知逆時序那一半在該任務上的表現非常差。同時雙向RNN的容量增加了一倍，這會導致過度配適的情形提早出現。儘管如此雙向RNN非常適合文字資料或任何順序很重要但是哪一種順序並不重要的資料類型。