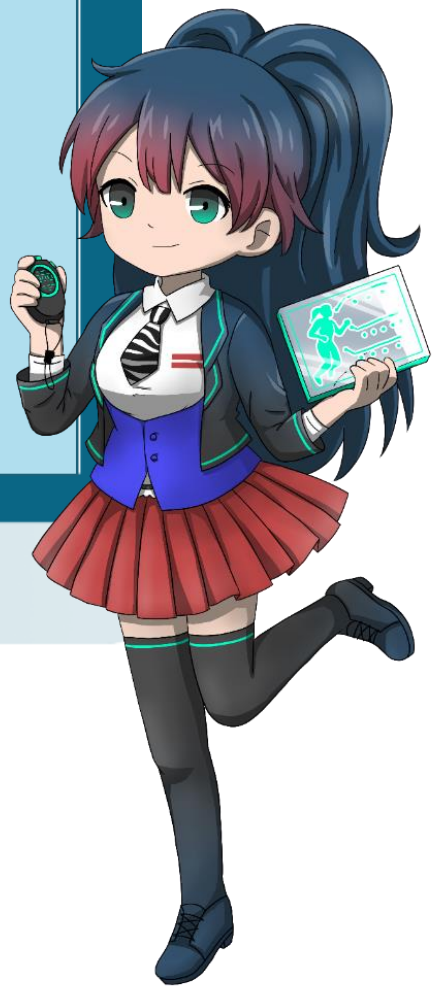
主題三： Pytorch 套件介紹

單元1

Pytorch Tensor基本概念



什麼是PyTorch？

- PyTorch是一個基於Torch的框架(framework)，由Facebook的人工智能小組開發以開源方式移植成Python機器學習庫。
- PyTorch 於 2017 年早期推出，短時間被學術研究領域大量採用，它是訓練神經網絡的最佳框架，並且使用起來很方便。
- 但因為起步晚，在此之前由Google所發展的TensorFlow框架為主流，在不太願意改變自己的業界，採用的還是以 TensorFlow 為大宗，但Pytorch有逐漸迎頭趕上之勢。

什麼是PyTorch？

- Numpy和PyTorch比較：Pytorch可以稱為機器學習領域的Numpy

資料套件	NumPy	PyTorch
資料名稱	NdArray	Tensor
運算	僅支持 CPU 計算	支持GPU加速計算
神經網路模組	x	實現動態神經網路
自動求導	x	具備反向傳播求導
函式	大部分的函式名稱和使用方法相似	

Tensor基本運算操作

- 和Numpy比對，可以快速學會Pytorch的tensor操作

資料套件	NumPy	PyTorch
Import套件	import numpy as np	import torch
元素都設置為1	np.ones((2, 3, 4))	torch.ones((2, 3, 4))
元素都设置为0	np.zeros((2, 3, 4))	torch.zeros((2, 3, 4))
產生連續值列	np.arange(12)	torch.arange(12)
由列表產生	np.array([[2,1,4,3],[1,2,3,4]])	torch.tensor([[2,1,4,3],[1,2,3,4]])
從均值為0、標準差為1的高斯分佈中隨機採樣	np.random.normal(0,1, size=(3,4))	torch.randn(3, 4)

Tensor基本運算操作

- 和Numpy比對，可以快速學會Pytorch的tensor操作

資料套件	NumPy	PyTorch
張量的形狀(沿每個軸的長度)	x.shape	x.shape
張量中元素的總數，即形狀的所有元素乘積	x.size	x.numel()
改變一個張量的形狀(-1可自動計算維度)	X = x.reshape(3, 4)	X = x.reshape(3, 4) 或 X = x.view(3, 4)

資料型態-常量、向量、矩陣、張量

資料型態	NumPy	PyTorch
常量：只有一個元素的資料。 $[10.0]$	A = np.array(3.0)	A = torch.tensor([3.0])
向量：由幾個常量組成一維的資料 $\begin{bmatrix}5.0\\4.0\\3.0\end{bmatrix}$	B= np.arange(24) 比較len(B)和B.shape	B = torch.arange(24) 比較len(B)和B.shape
矩陣：由幾個向量組成一維的資料，二維的常量 $\begin{bmatrix}5.0 & 2.0 & 1.0\\4.0 & 8.0 & 6.0\\3.0 & 9.0 & 7.0\end{bmatrix}$	C=B.reshape(6,4) 比較len(C)和C.shape	C=B.reshape(6,4) 比較len(C)和C.shape
張量：由幾個矩陣組成多維的資料 $\begin{bmatrix}\begin{bmatrix}2.0 & \cdots & 3.0\\ \vdots & \ddots & \vdots\\4.0 & \cdots & 5.0\end{bmatrix} & \begin{bmatrix}1.0 & \cdots & 2.0\\ \vdots & \ddots & \vdots\\6.0 & \cdots & 5.0\end{bmatrix}\end{bmatrix}$	D=B.reshape(2,3,4) 比較len(D)和D.shape	D=B.reshape(2,3,4) 比較len(D)和D.shape

Tensor和其它資料格式間的轉換

資料套件	NumPy	PyTorch
互相轉換 Torch張量和numpy矩陣會共享記憶體空間，改變其中一個也將改變另一個	A為NdArray=>Tensor torch.tensor(A) 或 torch.from_numpy(A)	B為Tensor=>NdArray B.numpy()
要將元素個數為1的張量轉換為數值 A=np.array([9.8]) 或 A=torch.tensor([9.8])	調用 item() 函數 或 Python 的內置函數 A.item()或float(A)	

單元2

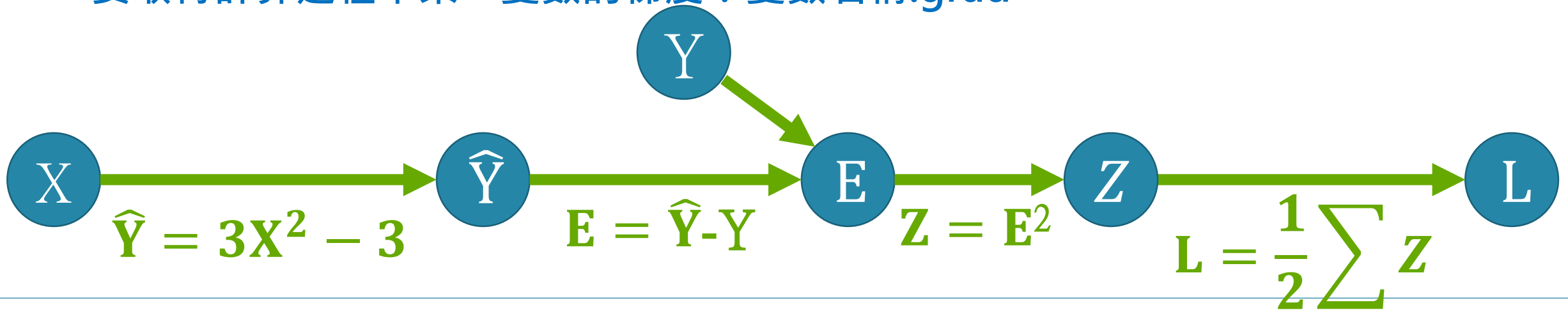
自動求導
AutoGrad

自動求導(autograd)

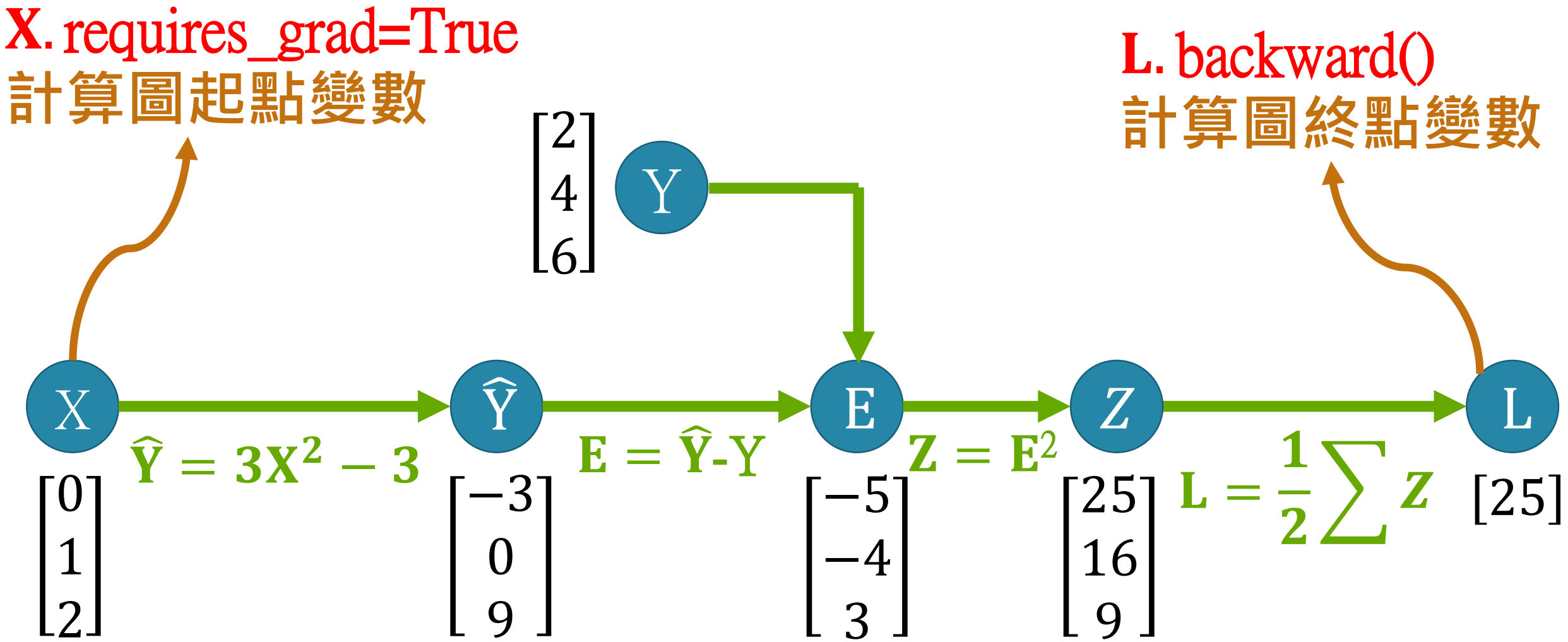
- 模型訓練時，計算到最後比較訓練資料集的標準解答，我們會得到所謂的損失函數(Loss function)，必須透過梯度下降，來更新模型的權重和偏值，使損失函數下降。
- 求梯度是幾乎所有機器學習優化算法的關鍵步驟，雖然求梯度的計算很簡單，只需要一些基本的微分，但對於復雜的模型，反覆使用連鎖規則，手工進行更新是一件很痛苦的事情。
- 自動求導可以為張量上的所有操作，運算提供了每一個步驟自動算梯度求導，這樣一來不用為反向傳播繁複的連鎖規則傷腦筋，只需專心於前向傳播．

自動求導(autograd)

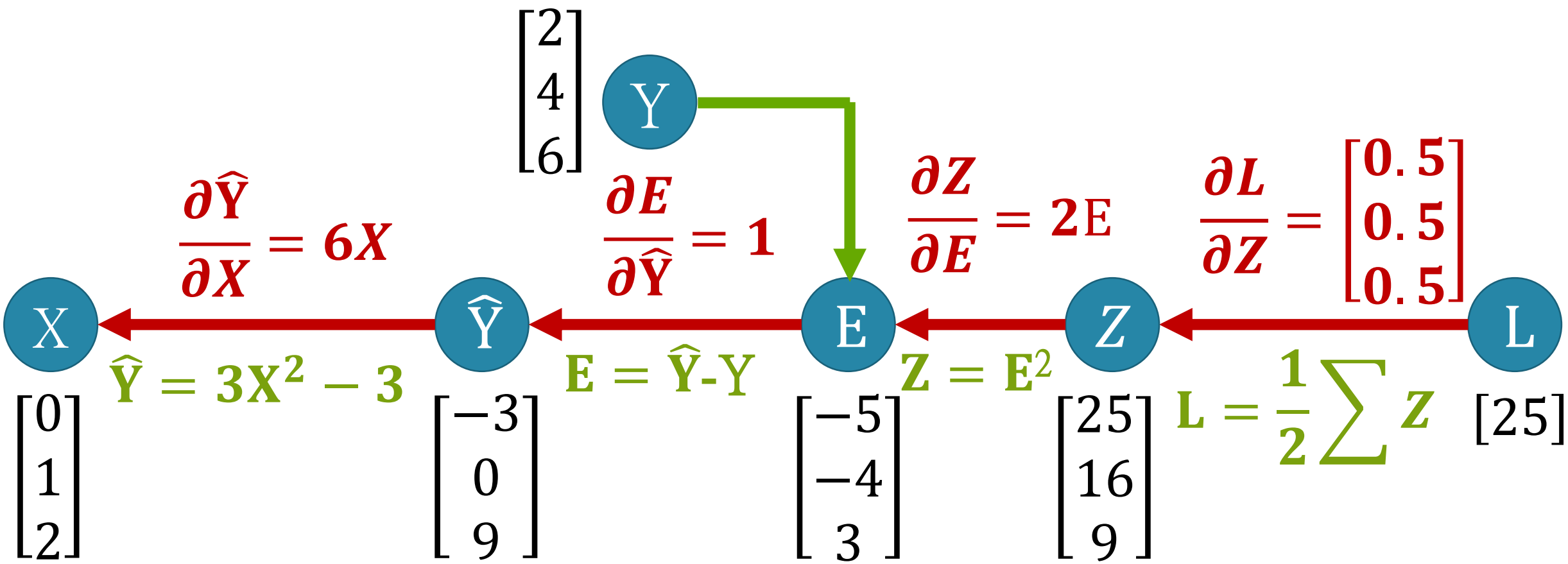
- 自動求導的原理：
  - 根據我們設計的模型，系統會構建一個計算圖(computational graph)，來跟踪計算是哪些數據？通過哪些操作？組合起來產生輸出。
  - 自動求導使系統能夠隨後反向傳播各個操作的梯度。
- 自動求導步驟：
  - 計算圖起點變數須設定為requires_grad(True)。
  - 若要記錄中間過程變數的梯度須執行retain_grad()
  - 計算圖終點變數須執行backward()取得反向傳播梯度。
  - 要取得計算過程中某一變數的梯度：變數名稱.grad



自動求導(autograd)



自動求導(autograd)



自動求導(autograd)

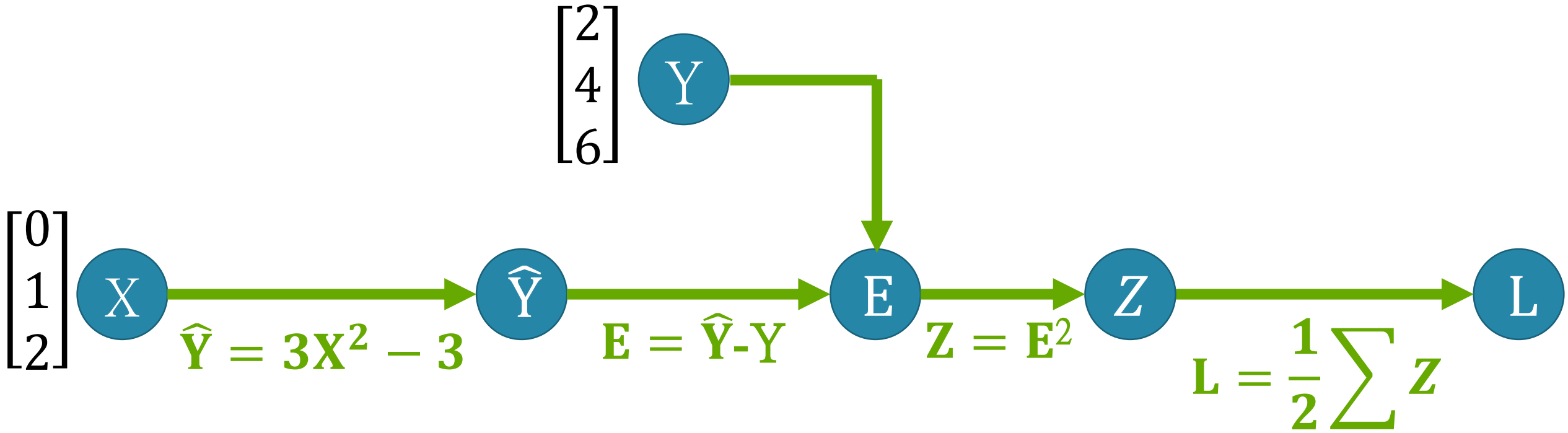
- 在預設情況下，PyTorch會累積梯度，使用：變數.grad.zero_() 可清除之前的梯度。

- 因為模型在訓練時的參數可能具有requires_grad=True，但在評估模型時，不需要在此過程中對他們進行梯度計算，要停止一個張量被跟踪梯度歷史，可用下面兩個方法：
  - 可以調用.detach()方法將其與計算歷史分離，並阻止它未来的計算記錄被跟踪。
  - 可以將程式碼包裝在 with torch.no_grad(): 的縮排中。

- 即使構建函數的計算圖需要通過 Python控制流（例如，條件、迴圈或任意函式呼叫），我們仍然可以計算得到的變數的梯度。

實作3-1 利用Pytorch的AutoGrad計算梯度

- 輸入特徵矩陣X經預估模型$\hat{Y} = 3X^2 - 3$後，與實際值矩陣 Y 計算誤差 E ，再計算損失函數 L 。
- 計算完損失函數後，利用自動求導，求出損失函數對各變數的梯度。



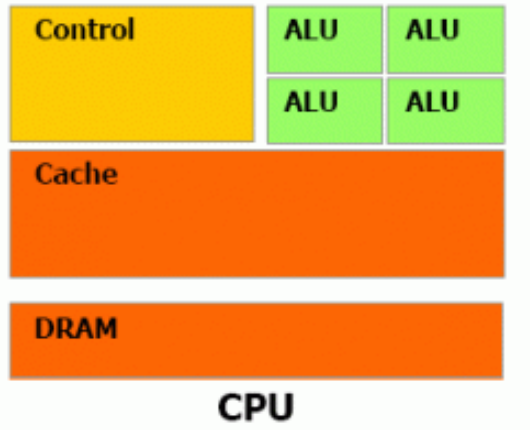
單元3

使用GPU
加速運算

CPU 與 GPU

- CPU (中央處理器 Central Processing Unit)由數百萬個電晶體打造而成，它負責執行電腦與作業系統所需的指令與程序，可視為電腦的大腦。

- CPU的內部組成：
  - 由運算器(ALU)：用來執行算術運算、移位操作、地址運算和轉換。
  - 控制器(CU)：負責對指令解碼，完成每條指令所要執行的各個操作的控制信號。
  - 寄存器：用於保存運算過程數據以及指令。
  - 高速緩衝存儲器：可從主記憶體中預載數據，加速處理速度。

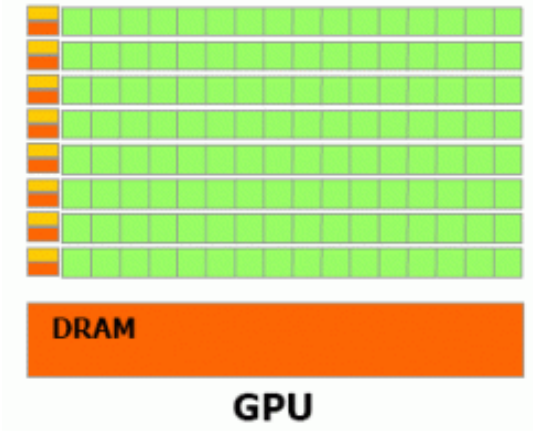


- 綠色的是計算單元
- 橙紅色的是存儲單元
- 橙黃色的是控制單元

- CPU是序列處理數據的，在處理大規模與高速數據時，CPU很難滿足需要，後來發展出多個處理核心的CPU，目前大概是4核、8核為主流。

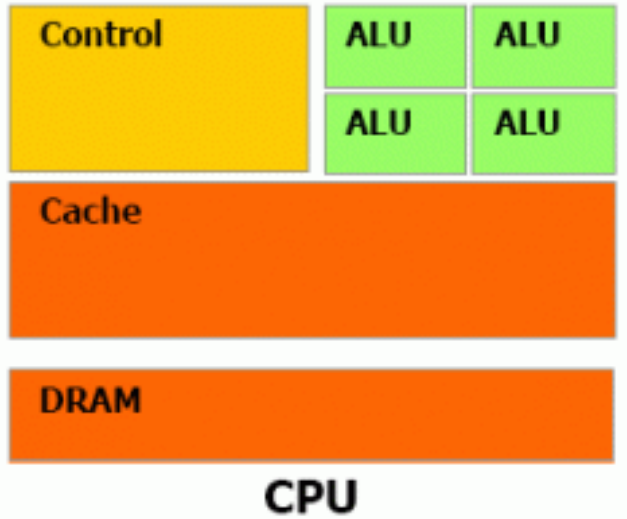
CPU 與 GPU

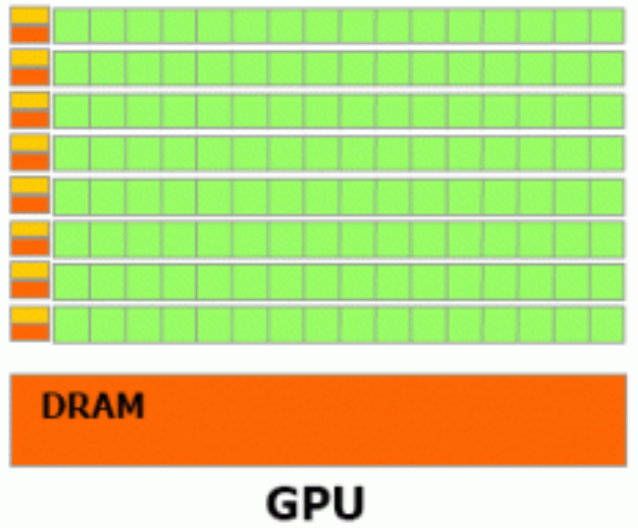
- 圖形處理對電腦應用來說越來越重要，圖形上的每個像素都要被處理，這就是一個大數據，於是一個專門處理圖形的核心處理器GPU(Graphics Processing Unit)應運而生。
- 圖形資料就是一個矩陣，所以對於影像或圖片的編輯，即是對一個巨大的數值矩陣做運算，現代機器學習的成果也是基於巨大數據矩陣的處理，GPU恰好在這方面能夠提供助力。
- 普通的GPU就包含了幾百個運算核，高端的有上萬個核，這對於多媒體處理中大量的重複處理過程有著天生的優勢，同時更重要的是，它可以用來做大規模並行數據處理。



CPU 與 GPU

CPU	GPU
功能較為全面 控制、運算、暫存約各佔30%	功能較為專一 運算功能就佔80%以上
Cache佔據了大量空間，	省去了Cache或極少量
有複雜的控制邏輯	只有非常簡單的控制邏輯
處理的資料複雜不一 只能序列是的處理	處理的資料較有一致性 可以併發平行運算
運算速度慢 0.1~1TFLOPS	運算速度快 10~100TFLOPS
頻寬較小 資料流寬度64bits	頻寬較大 資料流寬度可達512bits

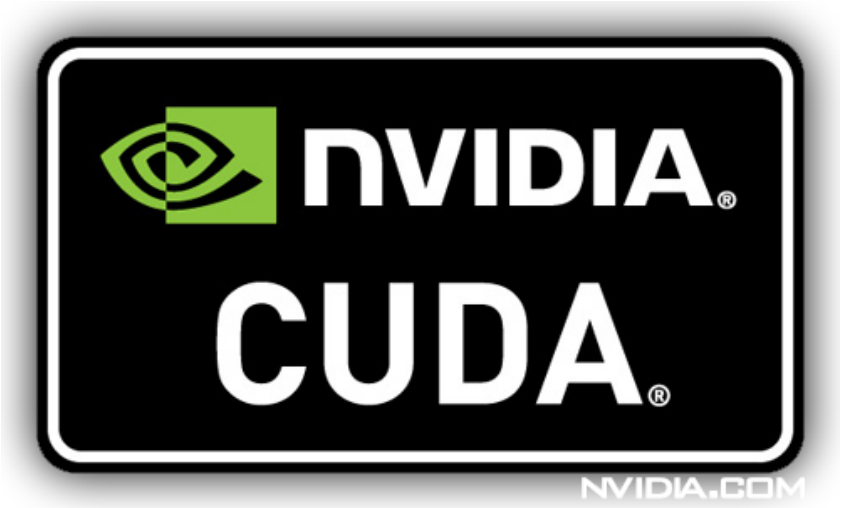




GPU 和 Cuda

- 目前全球最大的GPU大廠主要為Nvidia和AMD兩家：
  - AMD GPU主攻電玩遊戲方面的圖形處理，故較佔價格優勢。
  - Nvidia GPU除在遊戲方面外，在超級計算機、數據中心、深度學習硬體等方面的應用更大幅領先AMD，可以說是這方面的主流。

- 什麼是Cuda(Compute Unified Device Architecture)：
  為了把中央處理器(CPU)的運算工作，移到圖形處理器(GPU)上進行， 由NVIDIA提出來利用GPU運算的函式庫及API稱為Cuda。



使用GPU要注意的事項

- 使用GPU可以提升我們訓練的速度，如果使用不當，可能影響使用效率，具體使用時要注意以下幾點：
  （1）GPU的數量儘量為偶數，奇數的GPU有可能會出現異常中斷的情況。
  （2）GPU很快，但資料量較小時，效果可能沒有單GPU好，甚至還不如CPU。
  （3）如果記憶體不夠大，使用精度稍微低一點的資料類型，有時也效果很好。

Colab上如何啟用GPU?

- 高階運算的 GPU 降格昂貴，配合安裝軟體複雜，在 Google Colab 中提供了免費的 GPU 提供給想進行實驗的使用者進行運算。

- Google Colab 的GPU使用限制：
  - GPU只有一顆
  - 一天只能使用12小時的時限

- 如何將Colab後端調整為使用GPU 的類型：
  - 點擊 Runtime(執行階段)
    -> Change Runtime Type(變更執行階段)
    -> 將 Hardware accelerator(硬體加速) 選擇為 GPU。
    -> Save 存檔後，你應該就會連接上有 GPU 的後端。
- 查看Colab GPU的訊息：
  - 在 單元格 輸入「!nvidia-smi」。

Pytorch上使用GPU的語法

- Pytorch上查詢是否有GPU可供使用：
  - 語法：torch.cuda.is_available()
- 寫程式時為了保持彈性，讓不管是否有GPU的環境都能執行，我們會先判斷是否有GPU：
  - 語法：device=torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
- 要在GPU上運算：
  - 把資料張量一律直接使用：tensor物件.to(device)或tensor物件.cuda()即可。
  - 對於Pytorch所建立的神經網路型：模型物件.to(device)或模型物件.cuda()即可。
- 要在GPU上資料張量放回主記憶體：
  - GPU上的tensor物件.cpu()即可

單元4

將資料載入模型訓練的方式

不同資料量的訓練方式

- 訓練神經網路所需要的資料量是非常龐大的，我們依據每一次梯度下降更新權重使用的不同訓練資料量區分為：

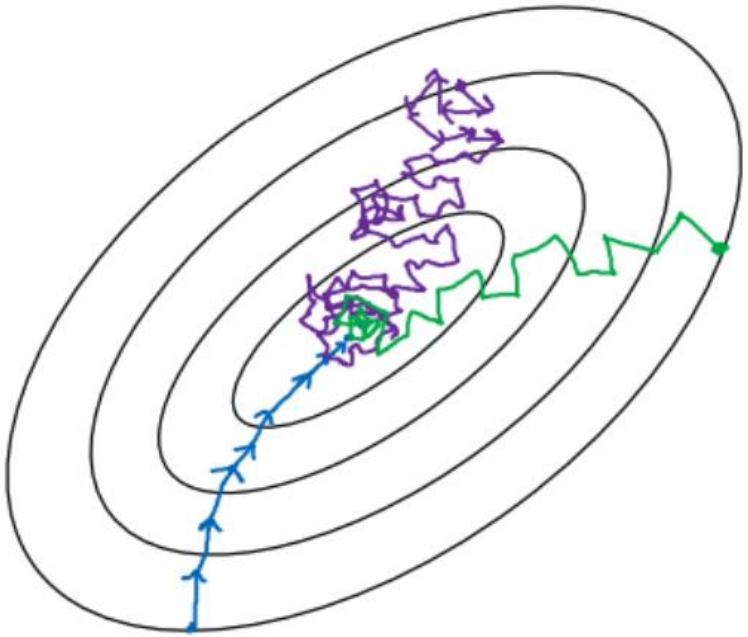
  - 批量梯度下降 (BGD，batch gradient descent)：
    一次把所有數據送進神經網路做訓練。

  - 随机梯度下降 (SGD，stochastic gradient descent)：
    一次只送一筆數據到神經網路做訓練。

  - 小批量梯度下降 (MBGD，min-batch gradient descent)：
    將數據分成幾等分，每次送一等分到神經網路做訓練。

不同資料量訓練方式的比較

- 批量梯度下降 ：一次把所有數據送進神經網路做訓練。
  - 優點：因為計算所有數據而得到的梯度是比較準確的，也就是每次都能找到最佳路徑往下走。
  - 缺點：當數據量非常巨大的時候，每一輪訓練都會花非常久的時間。
- 随机梯度下降 ：一次只送一筆數據到神經網路做訓練。
  - 優點：每一步計算都非常快速。
  - 缺點：是找到的梯度修正不一定指向正確方向，路徑會經過多次調整，花費時間。
- 小批量梯度下降 ：
  - 優點：每一步計算可以很快速，梯度修正方向大致正確；利用硬體可平行訓練。
  - 目前最多人採用的方法。



- 批量梯度
- 随机梯度下降
- 小批量梯度下降

如何將資料送入模型做訓練

- 在深度學習中，數據的預處理是第一步，pytorch提供了非常規範的處理接口工具。
- pytorch數據預處理三個重要基本類別：

- 注意這三個類別均由torch.utils.data 中載入
- 這三者是依次封裝的關係，Dataset被裝進DataLoader，DataLoder被裝進DataLoaderIter。

Dataset類別介紹

- Dataset是一個抽象類別，Pytorch中所有數據集加載類別中應該繼承的父類別。
- 其中Dataset類中的兩個私有成員函數必須被重載，否則將會觸發錯誤提示：
  - def __init__(self):
    - 由檔案(例:文字檔、圖片、語音、、、)去讀取數據集(特徵和標示)的程式碼。
    - 建構函數一般情況下我們也是要自己定義的，但是不是強制性的，可以留空在下面完成。
  - def __getitem__(self, index):(必須被重載)
    - 從資料集中得到一個數據片段（如：資料，標籤），並做資料處理(縮放、變形、、、)
    - 編寫支持數據集索引的函數，例如通過dataset[i]可以得到數據集中的第i+1個數據。
  - def __len__(self):(必須被重載)
    - 用來返回數據集的大小

Dataset類別介紹

■ 自定義Dataset類別的程式碼模板：

```
1  class MyDataset(Dataset):                    #需要繼承Dataset
2      def __init__(self):
3          #從檔案去讀取數據集
4          self.data = ◦ ◦ ◦ ◦ ◦ ◦ ◦
5          self.label = ◦ ◦ ◦ ◦ ◦ ◦ ◦
6      def __getitem__(self, index):
7          # 處理資料(縮放、變形、、、)
8          return self.data[index], self.label[index]
9      def __len__(self):
10         #資料集的總筆數
11         DataSize=len(self.data)
12         return DataSize
```

實作3-2 將鳶尾花資料集分別建立訓練、測試的DataSet

- 建立一個類別(class)，能由檔案(iris_dataset.csv)讀入鳶尾花資料集，並可以分別實體化建立訓練和測試的DataSet。
- 利用產生的Dataset物件兩個私有成員函數__len__()和__getitem__()，取出資料集的總筆數，並調用某一筆的資料。
- 練習從建立的DataSet，讀取資料的特徵資料，和標籤資料。

DataLoader類別介紹

- DataLoader原生類別和參數：

```
class torch.utils.data.DataLoader(
        dataset,
        batch_size=1,
        shuffle=False,
        sampler=None,
        batch_sampler=None,
        num_workers=0,
        collate_fn=<function default_collate>,
        pin_memory=False,
        drop_last=False
    )
```

- 主要參數有這麼幾個：
  - dataset : 即上面自定義的dataset物件。
  - batch_size：每一批次的資料筆數。
  - Shuffle：設定是否資料順序洗牌。
  - collate_fn: 這個函式用來打包batch。
  - num_worker: 非常簡單的多執行緒方法, 只要設定為>=1, 就可以多執行緒預讀資料。

DataLoader讀取數據的方式

- dataloader本質是一個可迭代物件，可以使用for迴圈進行可迭代物件的讀取：

```
for data,label in 要讀取的DataLoader物件:
    print("特徵資料：",data)
    print("分類標示：",label)
```

- 先使用內建私有成員函數__iter__()對dataloader進行第一步包裝，變成一個DataLoaderIter迭代器，然後就可以使用next讀取了。

```
DataLoaderIter迭代器=DataLoader物件.__iter__()
for Batch in range(len(DataLoader物件)):
    print("批次：",Batch+1)
    data,label=next(DataLoaderIter迭代器)
    print("特徵資料：",data)
    print("分類標示：",label)
```

如何將資料送入模型做訓練

- Pytorch加載數據到模型的流程有三步驟：

創建一個 Dataset物件

- 必須實現 __len__() __getitem__() 這兩個方法。

創建一個 DataLoader 物件

- 它是對DataSet對象進行迭代的。一般不需要改寫的其他方法了。

循環遍歷 DataLoader 對象

- 將資料特徵data, 資料標籤label加載到模型中進行訓練

如何將資料送入模型做訓練

■ Pytorch加載數據到模型的程式框架：

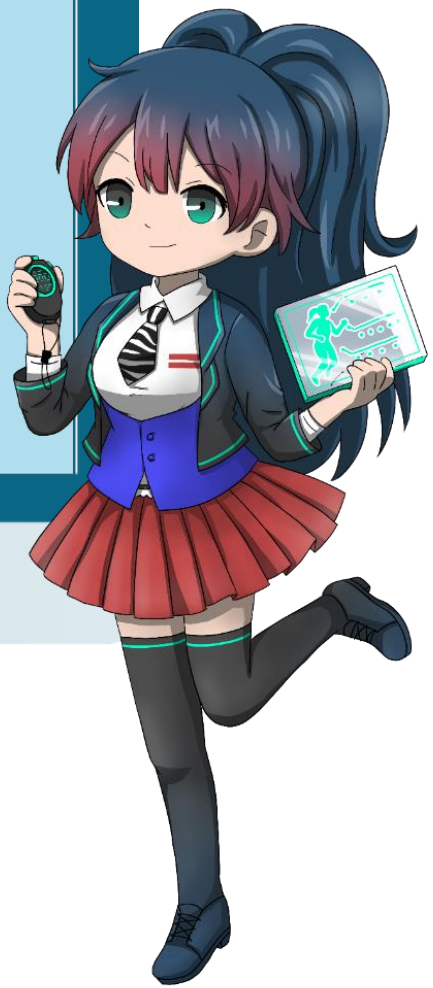
```
1 from torch.utils.data import Dataset,DataLoader    # 載入套件
2 dataset = MyDataset()                              # 第一步：構造Dataset對象
3 dataloader = DataLoader(dataset)                   # 第二步：通過DataLoader來構造迭
                                                     代對象
4 num_epoches = 100
5 for epoch in range(num_epoches):                   # 第三步：逐步迭代數據
6       for data, label in dataloader:
7             # 訓練代碼
8             。 。 。 。 。 。 。
9             。 。 。 。 。 。 。
```

實作3-3 將鳶尾花訓練資料集建立訓練DataLoader

- 將實作3-2的鳶尾花DataSet，包裝成DataLoader，指定batch_size=15，shuffle=True 。
- 練習從建立的DataLoader，用迴圈讀取每一批次的特徵資料，和標籤資料。
- 練習從建立的DataLoader使用__iter__()函式包裝成DataLoaderIter ，再用迴圈讀取每一批次的特徵資料，和標籤資料。

單元5

損失函數
Loss Function



損失函數(Loss Function)

- 深度學習中的Loss Function有很多，常見的有MSE、CrossEntropy，其最終目的就是計算預測值與實際值間的差別，而優化器的目的就是最小化Loss，當Loss的值穩定後，便是模型的權重和偏值最優的時候。

- Pytorch總共約有19種的Loss Function，不同的Loss Function適用不同問題型態，這裡只針對常見的Loss Function進行闡述。

- 均方誤差MSE(Mean Square Error)：模型預測值$\hat{y}_k$與真實樣本值y_k之間差值平方的平均值。
  - 類別函數：
    torch.nn.MSELoss(參數詳見官網)
  - *數學運算：*

$$\mathbf{L}(\hat{y}_k, y_k) = \frac{\sum_{k=1}^{K}(y_k - \hat{y}_k)^2}{K}$$

損失函數(Loss Function)

- 二元分類交叉熵：
  - 類別函數：
    torch.nn.BCELoss(預測值,標準值,、、、其他參數詳見官網)
  - 數學運算：

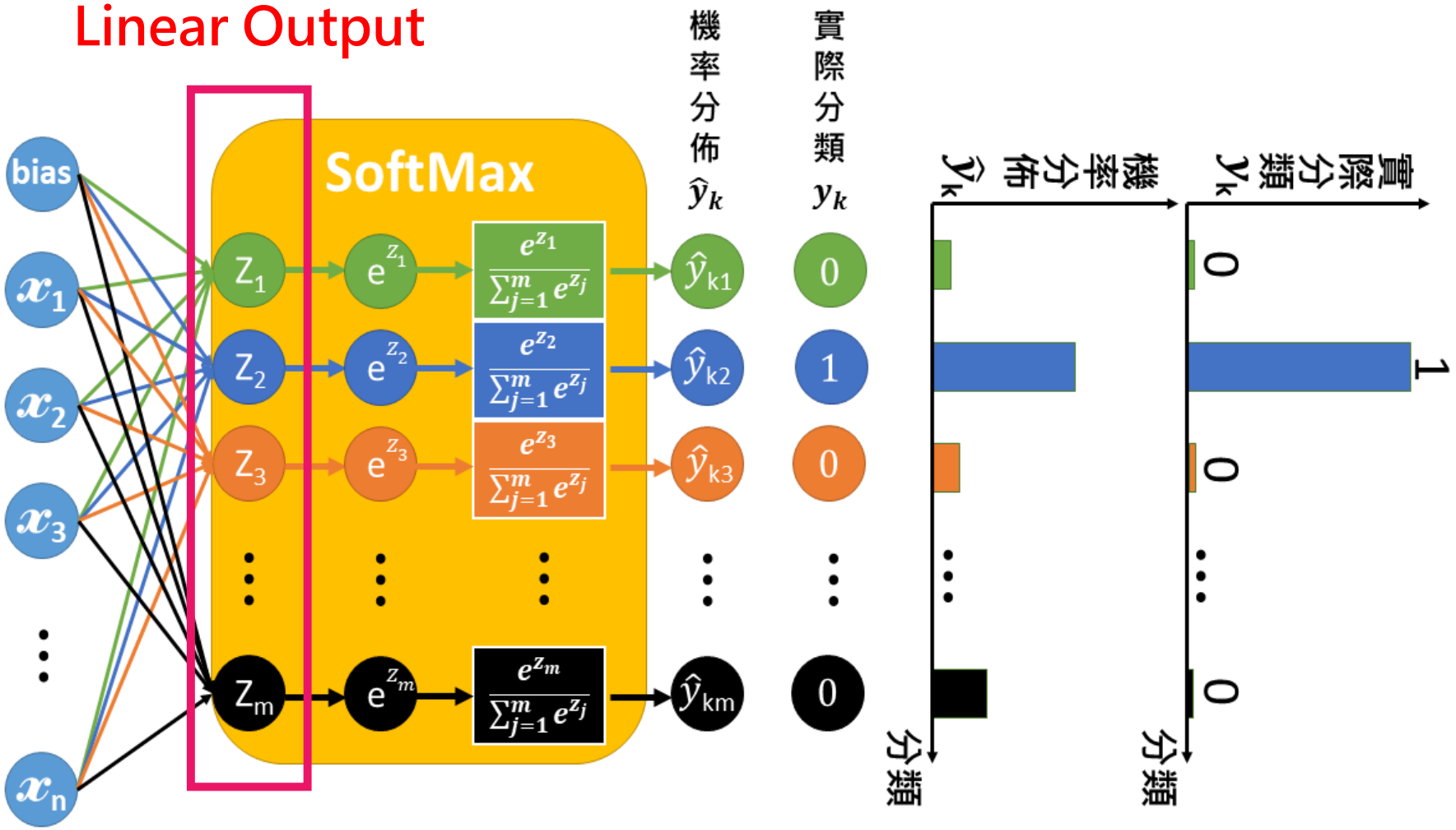
$$L(\hat{y}_k, y_k) = -\frac{1}{K}\sum_{k=1}^{K} y_k \cdot \ln(\hat{y}_{kj}) + (1 - y_k) \cdot \ln(1 - \hat{y}_{kj})$$

- 交叉熵CrossEntropy：
  - 類別函數：
    torch.nn.CrossEntropyLoss(預測值,標準值,、、、其他參數詳見官網)
  - 數學運算：

$$L(\hat{y}_{kj}, y_k) = \frac{1}{K}\sum_{k=1}^{K} l_k = -\frac{1}{K}\sum_{k=1}^{K}\sum_{j=1}^{m} y_{kj} \cdot \ln(\hat{y}_{kj})$$

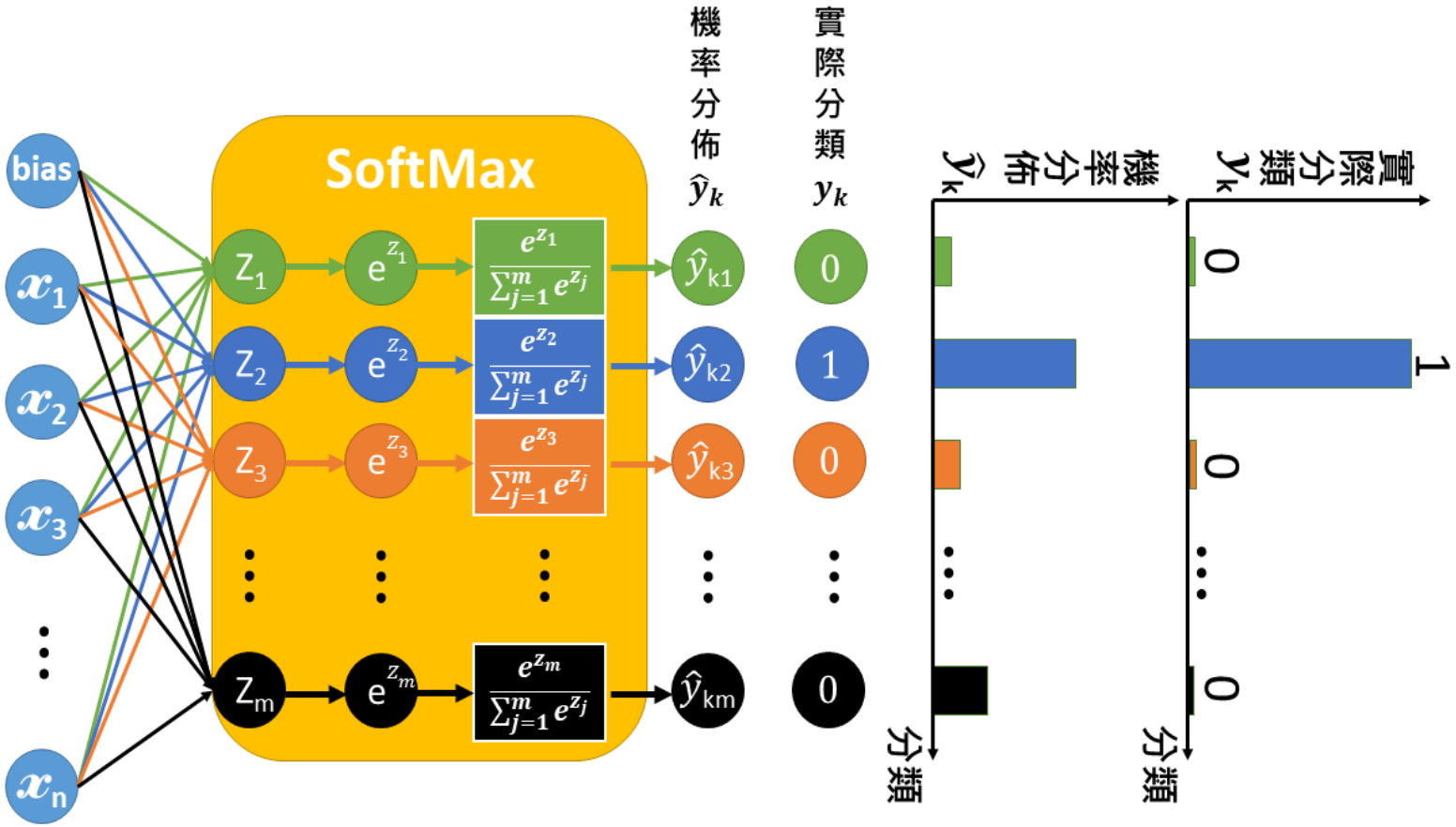
損失函數(Loss Function)

最後一層線性輸出
Linear Output



損失函數(Loss Function)

- 使用nn.CrossEntropyLoss() 時：
  - 預測值使用最後一層線性運算的輸出(Linear Output)。
  - 標準值不需做one hot encode

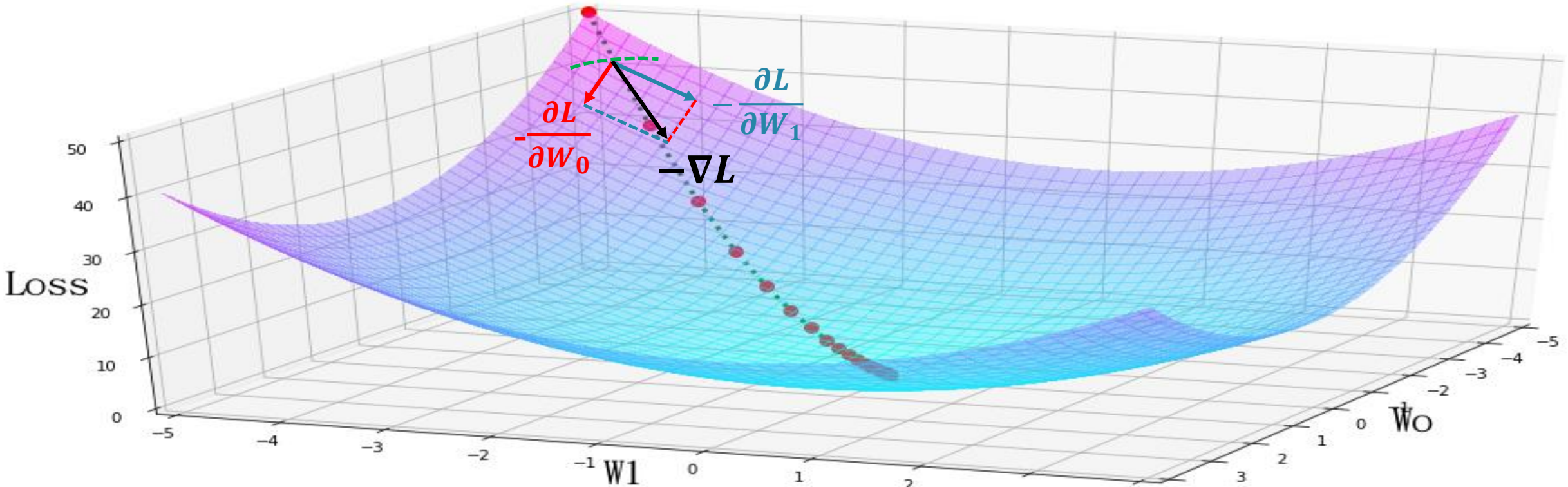


單元6

優化器
Optimizer

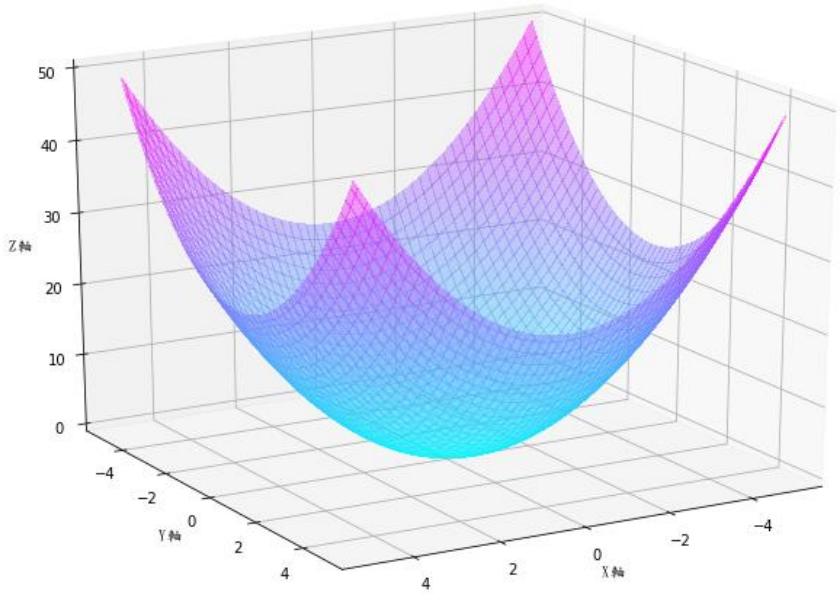
梯度下降算法Gradient Descent

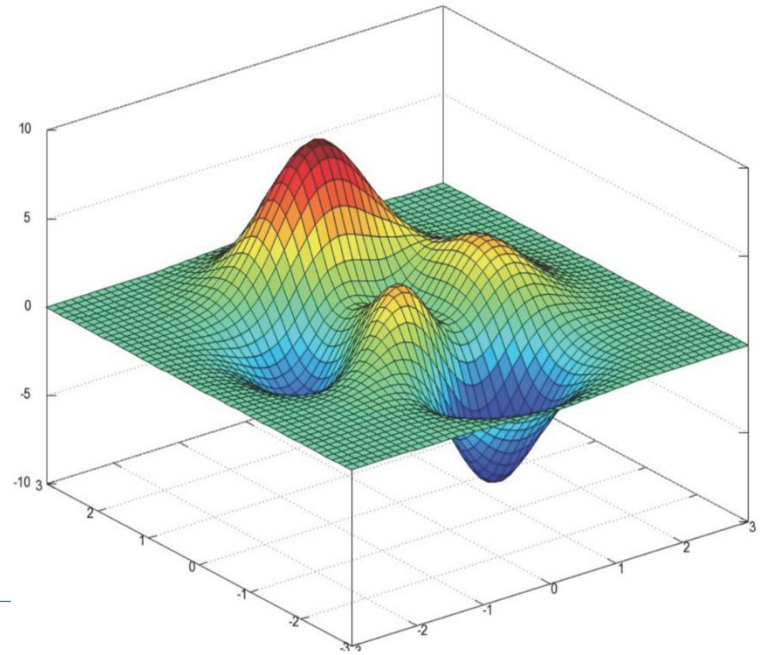
- 偏微分$\frac{\partial L}{\partial W_0}$為損失函數L對權重$W_0$的變化率，偏微分$\frac{\partial L}{\partial W_1}$為損失函數L對權重$W_1$的變化率，由這兩個變化率組成梯度向量$\nabla L$。
  - 梯度向量∇L和等高線互相垂直，為下降趨勢最快速的方向。
- 利用梯度下降算法修正權重：$W = W + \eta(-\nabla L)$
  - η為學習率控制下降的幅度。

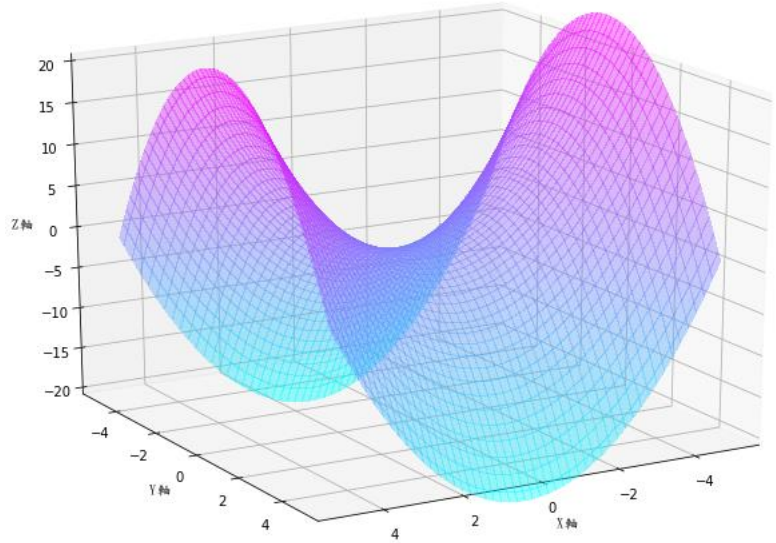


梯度下降算法的難題

- 學習率的調整：學習率過小時收斂速度慢，而過大時導致訓練震盪，而且可能會發散。
- 局部極小值點：梯度下降的過程中有可能陷入到局部極小值中。
- 鞍點：在某些變數方向往上彎曲，在其它方向往下彎曲，由於此點周圍的梯度都近似為0，梯度下降如果到達了鞍點就很難逃出來，高維問題中會更加常見。







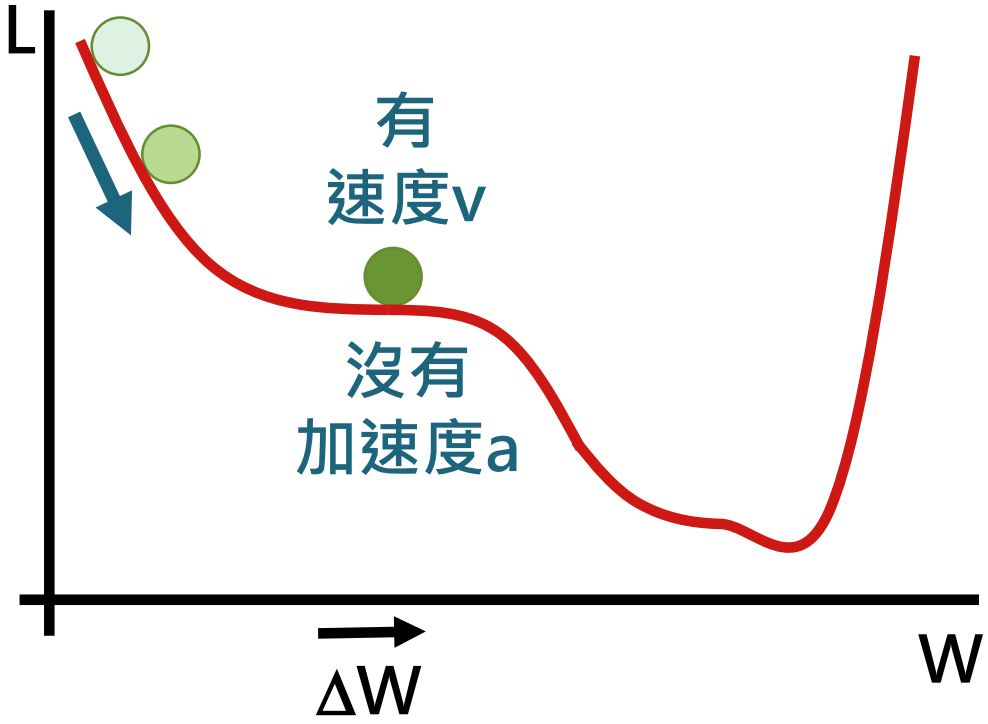
傳統梯度下降的問題

- 梯度向量∇L代表的是會產生下滑力，類似於物理上的加速度a。

- 權重位置變化(位移)ΔW
  =(新位置W-舊位置W)
  $=-\eta(\nabla L)$

- 參考上式，所以單純梯度下降只有考慮每個位置的加速度，沒有任何速度。

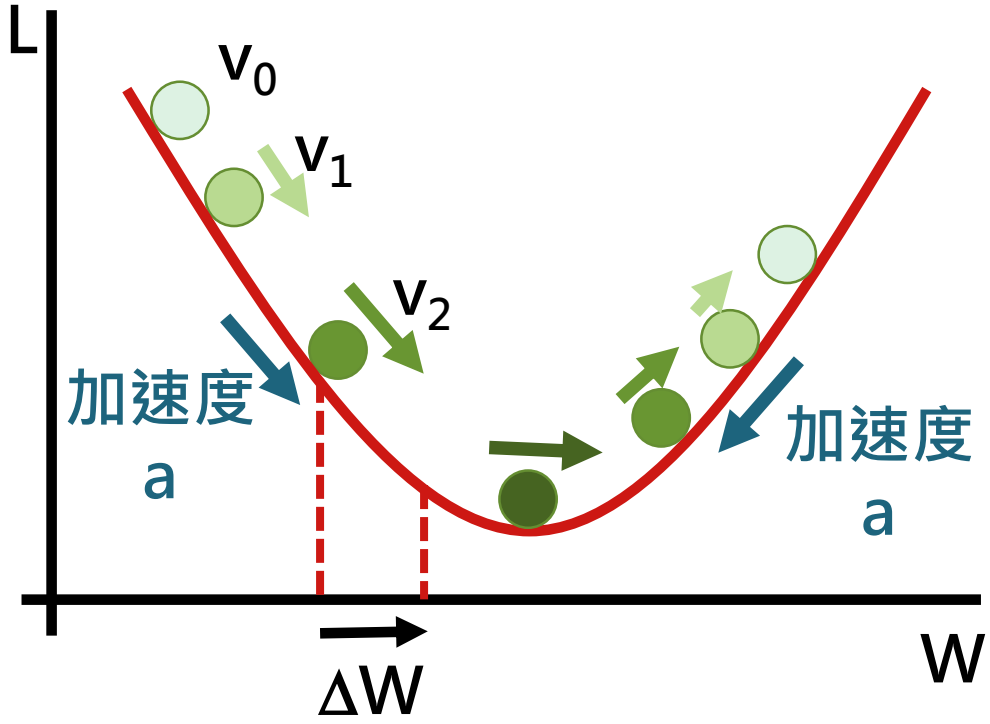
- 傳統梯度下降每移動到新的位置都是先停止一下，再朝梯度的方向移動一小步，然後又停止、、、。
  ⇨這樣跟想像成一個小球從山坡到山谷　的過程，是不同的。



動量momentum梯度下降

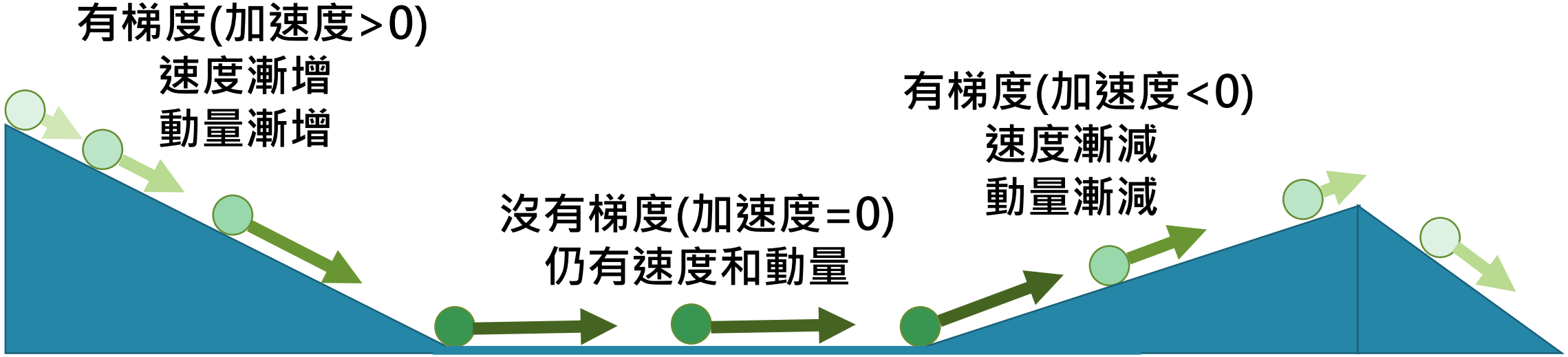
- 動量是物理上的物理量，其定義為：動量P=物體的質量x速度
  那麼滾動的小球質量都視為定值，其實只要考慮速度就好。

- 動量梯度下降法：
  - 每一個位置的速度，除了要考慮之前的累積速度外，還要考慮當下的加速度a
    目前速度v=ρx前面速度v+$(-\nabla L)$xη
  - ρ的物理意義為摩擦損失率，有效地抑制速度，降低了系統的動能，不然永遠不會停下來。
  - 以目前速度v(動量p)，來造成權重位置變化
    (位移)Δ W =(新位置W-舊位置W) =ρx初速v+$(-\nabla L)$xη



動量momentum梯度下降

- 動量(momentum)梯度下降
  - 摩擦損失率ρ通常設為[0.5,0.9,0.95,0.99]中的一個，一個典型的設置是剛開始將動量設為0.5而在後面的多個週期（epoch）中慢慢提升到0.99。
  - 這樣跟想像成一個小球從山坡到山谷的過程，就比較相似，球滾動時遇到小凹地形，是可以靠之前累積的速度來越過的。
  - 缺點：靠近極小值(谷底)時，動量法會來回震盪，直到動量損失後停在谷底。



實作3-4 利用動量梯度下降法逼近多項式函數最小值

- 實作目的：透過此實作能理解動量梯度下降法逼近函數最小值的方法，並用動畫展示如何一步步逼近函數最小值。
- 實作要求：
  - 能完成動量梯度下降的程式，並能動畫展示。
  - 能修改摩擦損失率的大小，觀察對梯度下降的速率和結果會造成什麼影響。

PyTorch 優化器(Optimizer)

- Optimizer是深度學習模型訓練中非常重要的一個模塊，它決定參數更新的方向、快慢和大小，好的Optimizer算法和合適的參數使得模型收斂又快又準。

- 優化器從最簡易的 SGD ,Momentum, RMSprop 到 Adam，都是為了解決：
  - 梯度下降的優化過程，找到的 loss 值不會被侷限在局部最小或鞍點，而是我們想要的全域最小
  - 學習率不易調整

  而加入了不同的構想，值得深入了解。
- PyTorch中的幾種optimizer優化方法，可以分為2大類：
  - SGD及其改進(加Momentum)：所有的參數都是一個學習率。
  - 自適應學習率方法：對不同的參數有不同的學習率，包括AdaGrad、RMSProp、Adam等。

PyTorch 中常用的優化器

- Pytorch的torch.optim是包含各種優化器算法的套件，使用 Optimizer 非常的容易，一行程式碼即可搞定，使用前先import。

```
1 from torch import optim
```

- 為了使用torch.optim，需先構造一個優化器物件Optimizer，用來保存當前的狀態，並能夠根據計算得到的梯度來更新參數。

- 要構建一個優化器optimizer，你必須給它一個可進行反覆運算優化的包含了所有參數的列表，即指定要優化的目標。

- 您可以指定程式優化特定的選項，例如學習速率，權重衰減等。

torch.optim.SGD

```
1 optimizer = optim.SGD(model.parameters(), 關鍵字參數、、、、)
```

- 關鍵字參數：
  - lr：學習率
  - Momentum： 動量-通常設定為0.9，0.8
  - Dampening：若採用nesterov，dampening必須為 0.
  - weight_decay：權值衰減係數，也就是L2正則項的係數，初步嘗試可以使用 1e-4 或者 1e-3
  - Nesterov：是否使用NAG(Nesterov accelerated gradient)
- Momentum若設定為0，沒有參酌之前的動量，就是傳統的梯度下降。
- NAG(Nesterov accelerated gradient)
  - 不使用當前位置的梯度，而是計算未來位置的梯度，來計算動量。
  - 能提前看到前方的梯度，若前面的梯度比當前位置大，就可以把步子邁得大一點，若前方梯度小，就可以邁小步一點， 收斂會更快。

torch.optim.Adagrad

```
1 optimizer = optim.Adagrad(model.parameters(), 關鍵字參數、、、、)
```

- 關鍵字參數：
  - lr_decay=0
  - weight_decay=0
  - initial_accumulator_value=0
- Adagrad是一種自適應優化方法，是自適應的為各個參數分配不同的學習率。
- 這個學習率的變化，會受到梯度的大小和迭代次數的影響。
- 梯度越大，學習率越小；梯度越小，學習率越大。
- 缺點是訓練後期，學習率過小，因為Adagrad累加之前所有的梯度平方作為分母。

torch.optim.Adadelta

```
1 optimizer = optim.Adadelta(model.parameters(), 關鍵字參數、、、、)
```

- 關鍵字參數：
  - rho=0.9
  - eps=1e-06
  - weight_decay=0
- 是Adagrad的改進。
- Adadelta分母中採用距離當前時間點比較近的累計項，這可以避免在訓練後期，學習率過小。
- 用於平滑的式子eps（一般設為1e-4到1e-8之間）是防止出現除以0的情況。

torch.optim. RMSprop

```
1 optimizer = optim. RMSprop(model.parameters(), 關鍵字參數、、、、)
```

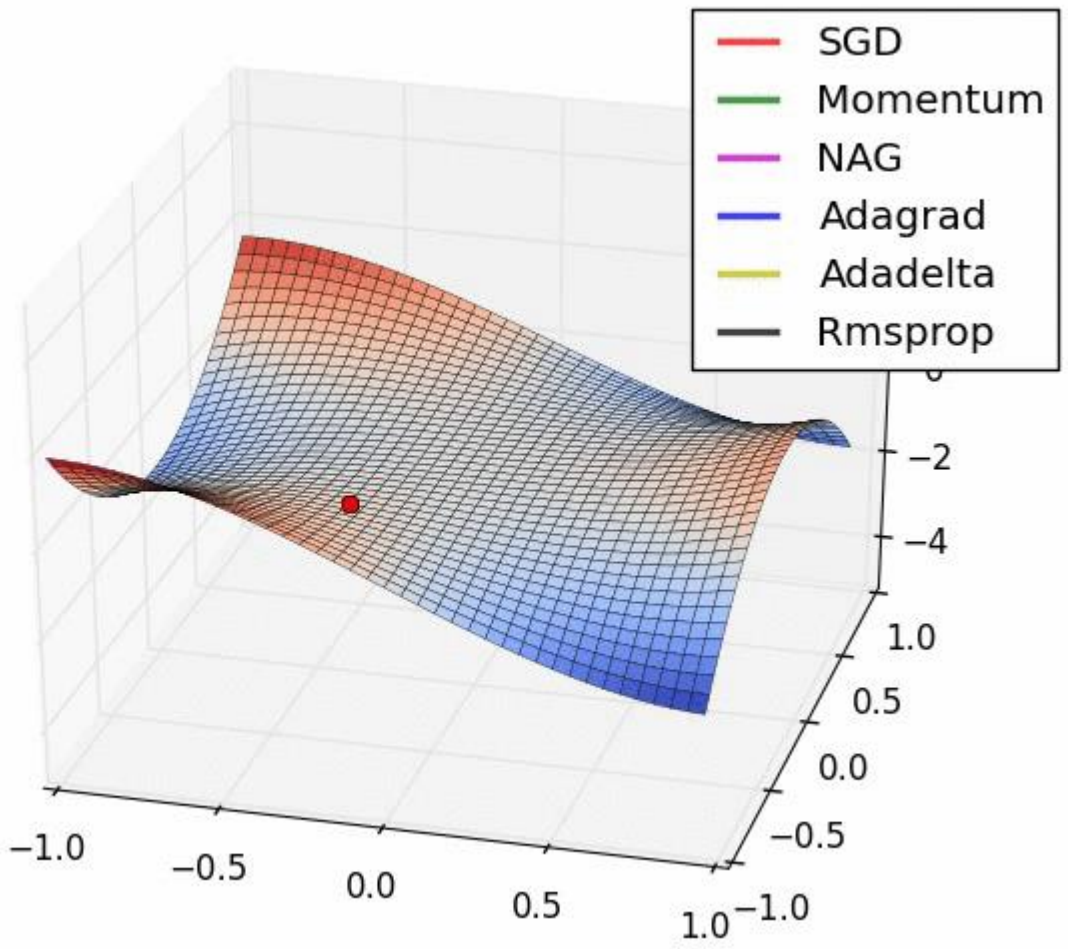
- 關鍵字參數：
  - alpha=0.99
  - eps=1e-08
  - weight_decay=0
  - momentum=0
  - centered=False
- RMS是方均根(root meam square)的意思，也是對Adagrad的一種改進。
- RMSprop採用均方根作為分母，可緩解Adagrad學習率下降較快的問題，並且引入均方根，可以減少擺動。

torch.optim.Adam

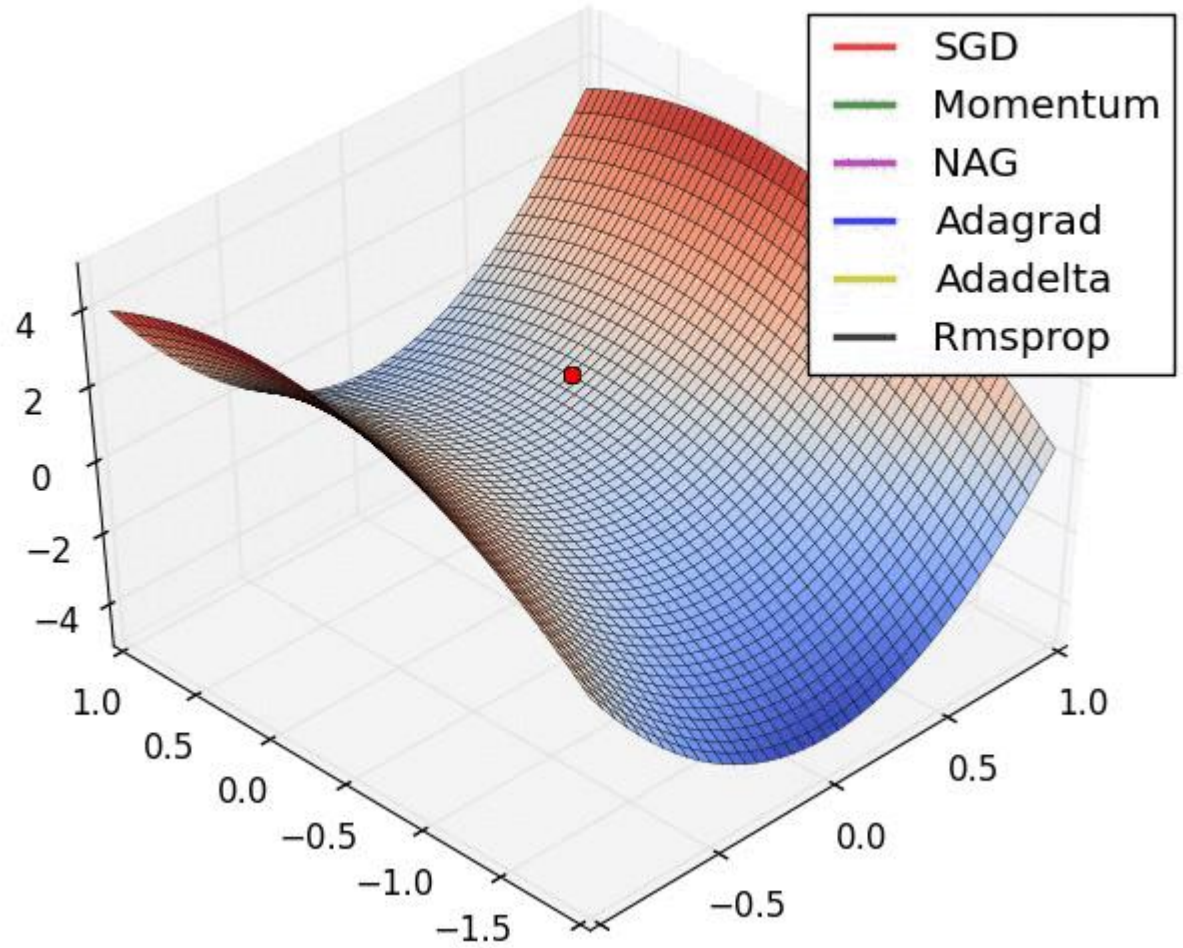
```
optimizer = optim.Adam(model.parameters(), 關鍵字參數、、、、)
```

- 關鍵字參數：
  - betas=(0.9, 0.999)
  - eps=1e-08
  - weight_decay=0
  - amsgrad=False
- amsgrad- 是否採用AMSGrad優化方法，asmgrad優化方法是針對Adam的改進，通過新增額外的約束，使學習率始終為正值。
- Adam是帶有動量項的RMSprop，利用梯度的一階矩估計和二階矩估計動態調整每個參數的不同學習率。
- Adam的特點有：
  - 每一次反覆運算學習率都有個確定範圍，使得參數比較平穩。
  - 結合了Adagrad善於處理稀疏梯度和RMSprop善於處理非平穩目標的優點。
  - 對記憶體需求較小。

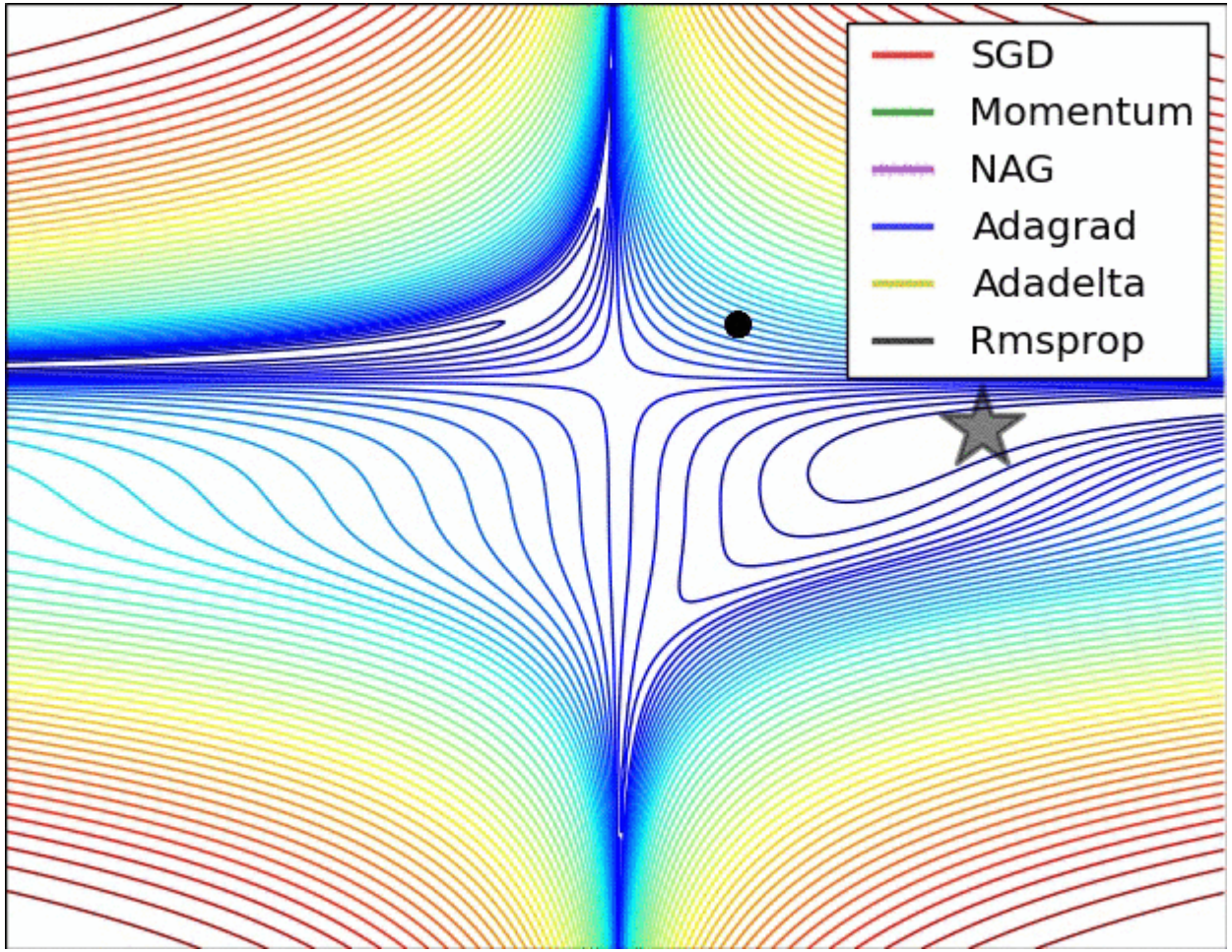
各種優化算法的比較


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

各種優化算法的比較


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

各種優化算法的比較



優化器要如何選擇？

- 在實際操作中，我們推薦Adam作為默認的演算法，一般而言跑起來比RMSProp要好一點。
- 但是也可以試試SGD+Nesterov動量。
- 完整的Adam更新演算法也包含了一個偏置（bias）矯正機制，因為m,v兩個矩陣初始為0，在沒有完全熱身之前存在偏差，需要採取一些補償措施。

訓練模型時的優化流程

```
   #定義使用的優化器並設定參數
1  optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
   #透過DataLoader迭代取出批量資料
2  for data,label in dataset:            #使用迴圈讀取批量資料來訓練
3      optimizer.zero_grad()              #把前面的梯度歸零，否則會累積前一輪的
4      output = model(data)               #由模型算出預測值
5      loss = loss_fn(output, label)      #利用預測值、目標值，由損失函數算損失值
6      loss.backward()                    #由 loss值反向傳播自動求各參數的梯度
7      optimizer.step() #利用各參數的梯度，已指定優化法更新參數
```