2018/04/24 @ CCU

§ 最佳化 HW1 | 403410034 資工四 黃鈺程

TOC

- Abstract
- Environment
- Gradient Descent
  - Numerical Gradient
  - Gradient Descent
  - Momentum
- Experimental Results
  - F1
  - F2
  - F3
  - F4
- Conclusion

1 Abstract

我實作了 2 種最佳化的方法：原始的 Gradient Descent 與帶 Momentum 的 Gradient Descent，然後在 4 個函式上應用這 2 種方法並進行比較。這 2 種方法都需要計算函式的導數（偏微分），我使用數值方法來計算導數而不使用代數方法。最後展示了每個函數的 1. 可視化 2. 隨著迭代的函式值 3. 隨著迭代 x 到全域最佳解的距離。

2 Environment

使用 Python 的科學計算環境在 Fedora 27 上完成這個作業：

1. Python 3.6
2. Matplotlib
3. Numpy
4. Jupyter

程式碼放在 Github 上。如果想復現請在安裝好 Dependencies 後，在 Jupyter 中選 Cell/Run All。

3 Gradient Descent

3.1 Numerical Gradient

我使用以下式子來求偏微分：

$$\frac{\partial}{\partial x_i} f(\mathbf{x}) = \lim_{2h \to 0} \frac{f(x_0, \ldots, x_i + h, \ldots, x_{n-1}) - f(x_0, \ldots, x_i - h, \ldots, x_{n-1})}{2h}$$

用以下式子來求 gradient：

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_0}, \ldots, \frac{\partial f}{\partial x_{n-1}} \right)$$

於是 f 在位置 $\mathbf{x}$ 的 gradient 寫成程式是：

```
def numerical_gradient(f, x):
    h = 1e-4
    grad = np.zeros_like(x)
    for i, val in enumerate(x):
        # f(x + h)
        x[i] = val + h
        f1 = f(x)
        # f(x - h)
        x[i] = val - h
        f2 = f(x)
        # grad
        grad[i] = (f1 - f2) / (2 * h)
        # restore
        x[i] = val
    return grad
```

注意程式碼中 `x` 的型態是 `numpy.ndarray` 。

3.2 Gradient Descent

有了 gradient 後，就能實作出 gradient descent，其更新為：

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n)$$

對應的程式碼是：

```python
def gradient_descent(f, init_x, lr=0.01, step_num=100):
    path = np.zeros((step_num, init_x.size), dtype=np.float32)
    grad = np.zeros((step_num, init_x.size), dtype=np.float32)
    path[0] = init_x
    grad[0] = numerical_gradient(f, path[0])
    for i in range(1, step_num):
        path[i] = path[i - 1] - (lr * grad[i - 1])
        grad[i] = numerical_gradient(f, path[i])
    return path, grad
```

3.3 Momentum

在我實驗的過程中，發現原始的 gradient descent 在大片平坦區域時會停住（因為 grad 為 0），所以我實作了一個帶 Momentum（動量）的版本希望能解決這個問題。Momentum 模擬了球從山上滾下山時的物理現象：會有動量存在，球會順著方向繼續滾而不是馬上停住。

Momentum 方法的更新公式為：

$$\mathbf{v} = \alpha \mathbf{v} - \gamma \nabla f(\mathbf{x}_n)$$
$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}$$

一般建議的 α 是 0.9，所以對應的程式碼為：

```python
def momentum(f, init_x, lr=0.01, step_num=100, alpha=0.9):
    path = np.zeros((step_num, init_x.size), dtype=np.float32)
    grad = np.zeros((step_num, init_x.size), dtype=np.float32)
    velc = np.zeros((step_num, init_x.size), dtype=np.float32)
    path[0] = init_x
    grad[0] = numerical_gradient(f, path[0])
    velc[0] = -lr * grad[0]
    for i in range(1, step_num):
        path[i] = path[i - 1] + velc[i - 1]
        grad[i] = numerical_gradient(f, path[i])
        velc[i] = alpha * velc[i - 1] - lr * grad[i]
    return path, grad, velc
```

4 Experimental Results

我嘗試最小化的函式有 4 個：

$$
\begin{aligned}
f_1(x) &= x^4 - 3x^2 + 2 \\
f_2(\mathbf{x}) &= 100(x_1 - x_0)^2 + (1 - x_0)^2 \\
f_3(\mathbf{x}) &= x_0^2 + x_1^2 \\
f_4(\mathbf{x}) &= \frac{1}{20}x_0^2 + x_1^2
\end{aligned}
$$

利用 numpy，程式碼寫起來很簡單（再次強調，程式碼中的 `x` 是 `ndarray`）：

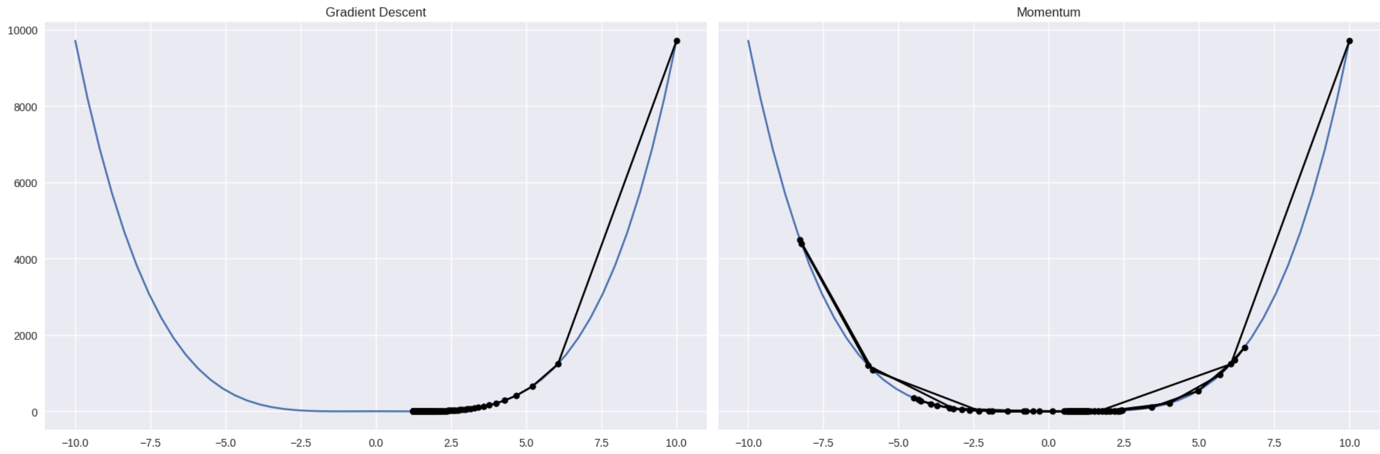
```python
def f1(x):
    return x**4 - 3 * x**2 + 2

def f2(x):
    return 100 * (x[1] - x[0]) ** 2 + (1 - x[0])**2

def f3(x):
    return x[0]**2 + x[1]**2

def f4(x):
    return 1/20 * x[0]**2 + x[1]**2
```

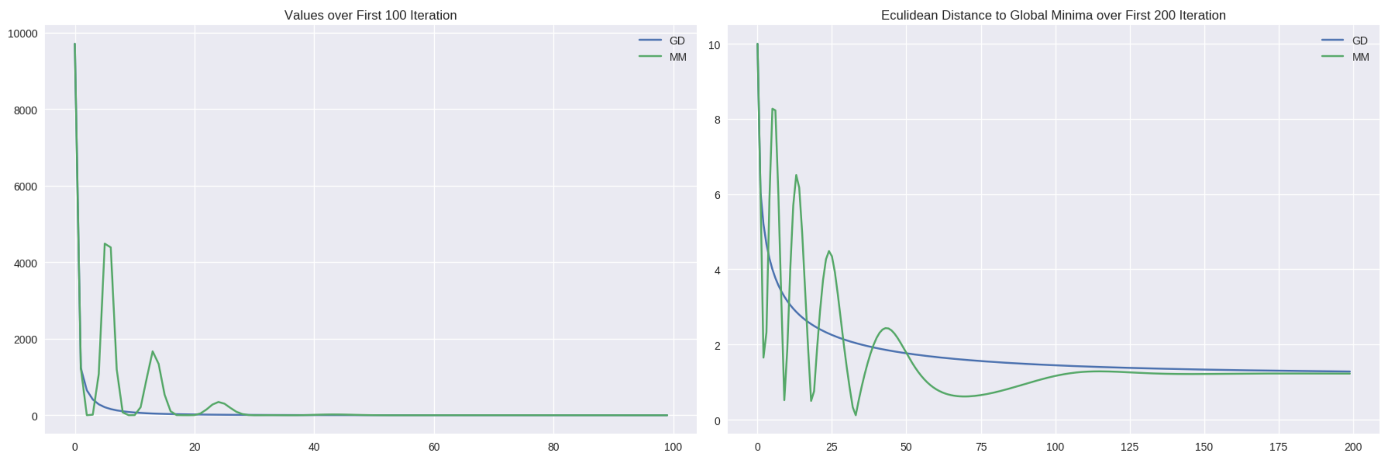
除了 f_1 以外，f_2, f_3, f_4 都是雙變數的函式，因此需要 3D 繪圖來可視化。所幸 matplotlib 有這個功能，利用 `Axes3D` 可以做到 3D 繪圖，並可以透過 `view_init` 調整視角，細節請參考我 Github 上的程式碼。底下我可視化了各函式優化的過程，以下簡稱 Gradient Descent 為 GD、帶 Momentum 的方法為 MM。

4.1 F1

$f_1(x) = x^4 - 3x^2 + 2$ 超參數 `init_x = [10,]`, `lr=0.001`, `iter=1000`。

Optimizer	$\mathbf{x}_n$	$f(\mathbf{x}_n)$
Naive Gradient Descent	1.2249577	-0.249999
Gradient Descent with Momentum	1.2247576	-0.249999





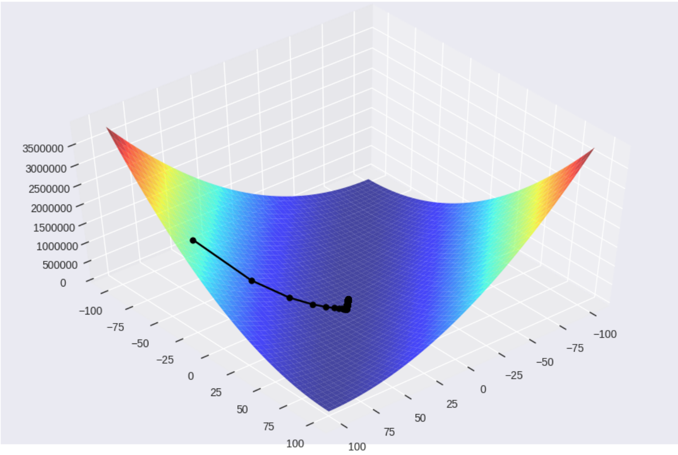
兩者跑出差不多的結果，只是優化的過程非常不同。而負數的產生我猜測原因是浮點數的誤差造成的。從左下的圖可以觀察到函數值的振盪，帶 Momentum 的方法如何預期的因為有動量的存在，衝上了函式的另一測。而從右下的圖可以看到 $\mathbf{x}$ 距離 Global Minima 的歐式距離越來越小，且 MM 優化的速度比 GD 快一些。

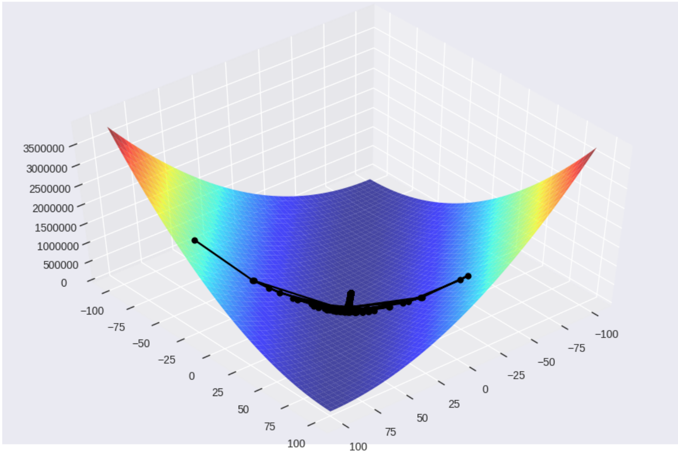
4.2 F2

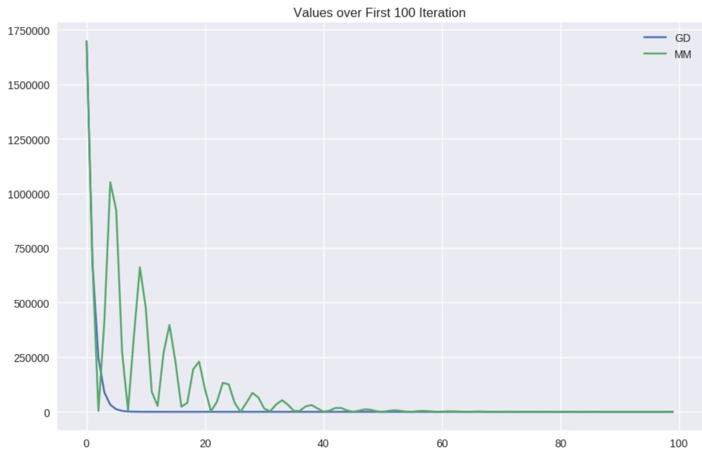
$$f_2(\mathbf{x}) = 100(x_1 - x_0)^2 + (1 - x_0)^2$$

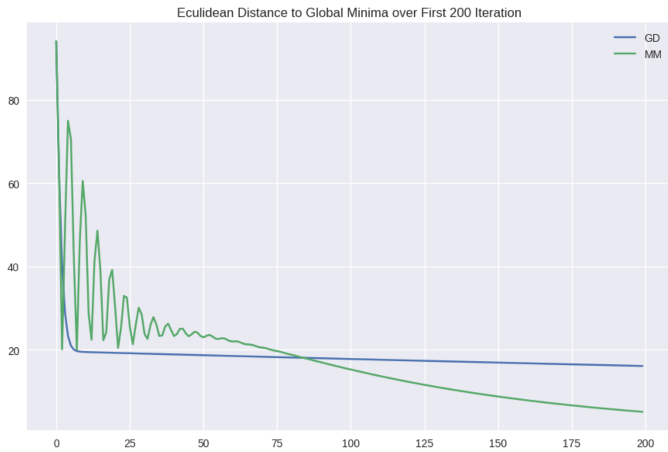
超參數 `init_x = [80.0, -50.0]`，`lr=0.001`，`iter=1000`。

Optimizer	$\mathbf{x}_n$	$f(\mathbf{x}_n)$
Naive Gradient Descent	[6.092 6.117]	25.995
Gradient Descent with Momentum	[1.000 1.000]	2.238e-07







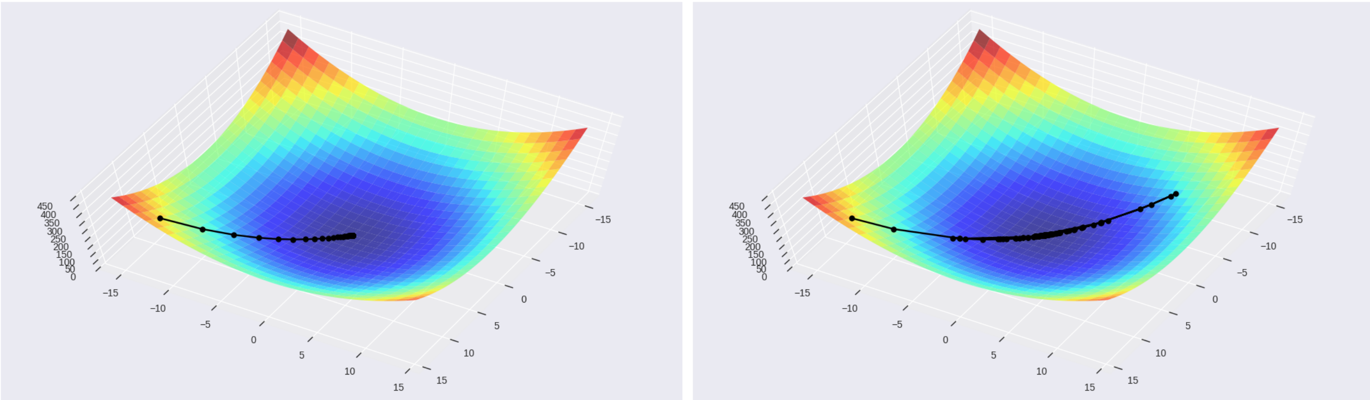


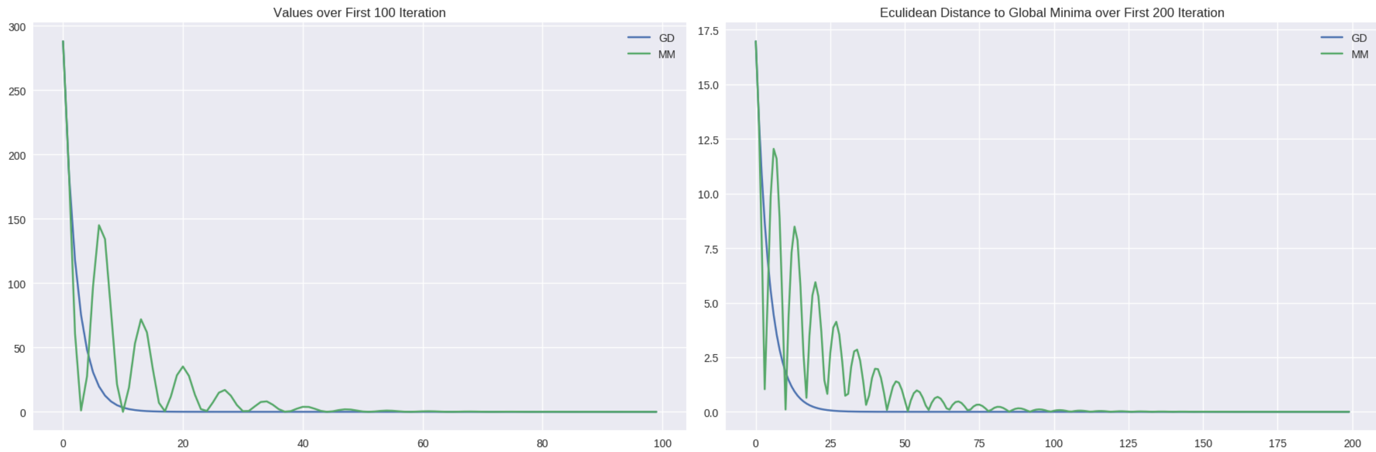
對於 f_2 兩種方法的結果就差距很大了，GD 根本無法找到全域最佳解，進入到平坦區域後就停住了，而 MM 帶有著動量能繼續前進。不過從左下的圖來說，MM 優化的速度比 GD 還要慢，但從右下的圖可以發現 MM 比 GD 更能接近 Global Minima。兩者各有其優點。

4.3 F3

$f_3(\mathbf{x}) = x_0^2 + x_1^2$ 超參數 `init_x = [12.0, -12.0]`, `lr=0.1`, `iter=200`。

Optimizer	$\mathbf{x}_n$	$f(\mathbf{x}_n)$
Naive Gradient Descent	[8.970e-13 -8.970e-13]	1.609e-24
Gradient Descent with Momentum	[-0.00028 0.00028]	1.586e-07



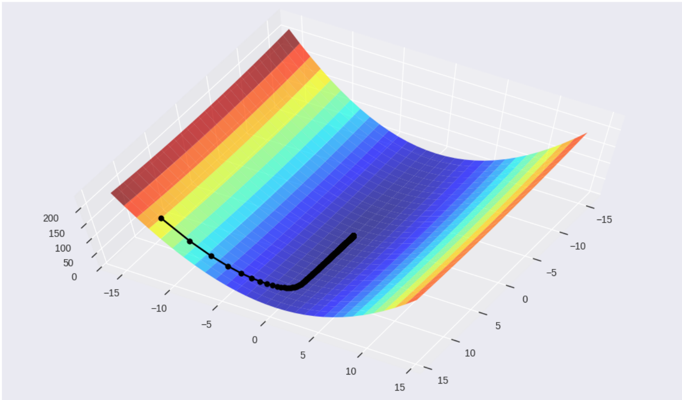


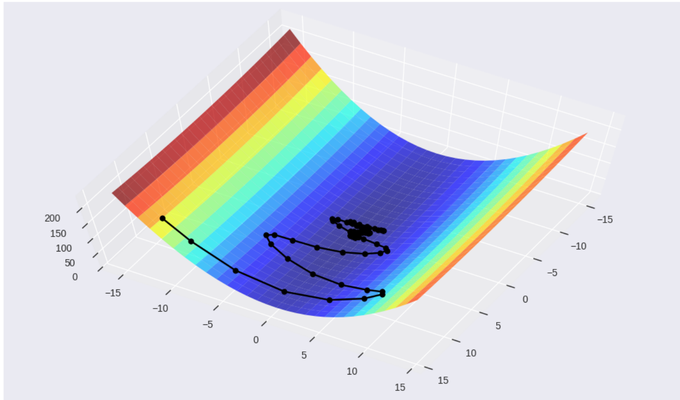
f_3 這種處處都有非零 gradient 的函式就簡單了。GD 在各方面都比 MM 更好。MM 會衝過頭而造成收斂的比較慢。

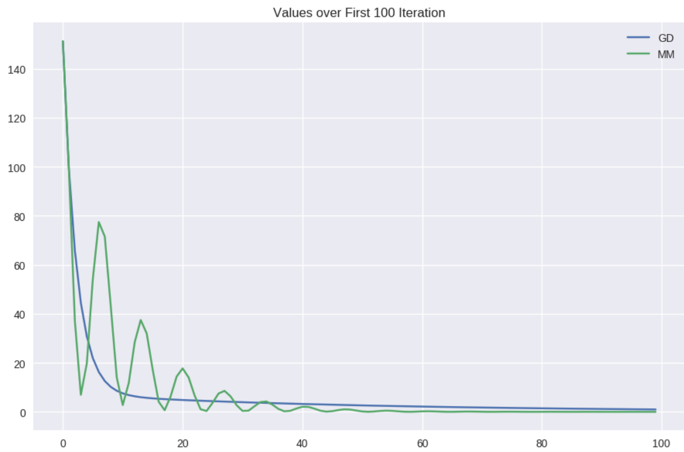
4.4 F4

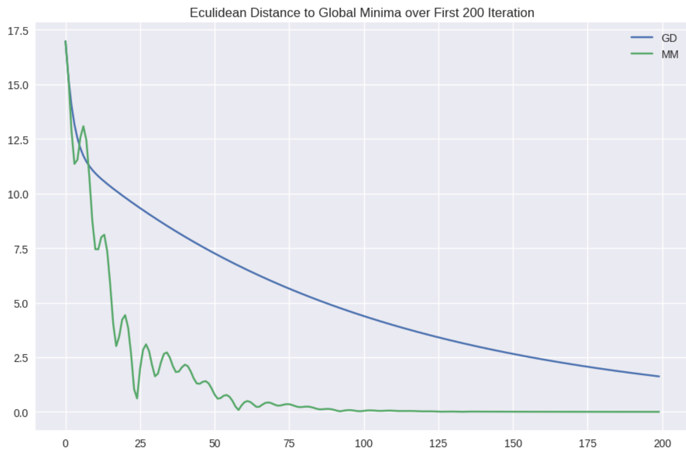
$f_4(\mathbf{x}) = \frac{1}{20}x_0^2 + x_1^2$ 超參數 `init_x = [12.0, -12.0]`, `lr=0.1`, `iter=500`。

Optimizer	$\mathbf{x}_n$	$f(\mathbf{x}_n)$
Naive Gradient Descent	[7.957e-02 -9.299e-13]	0.0003165
Gradient Descent with Momentum	[4.615e-11 4.310e-11]	1.964e-21









f_4 只比 f_3 多了一點係數，結果卻大不相同。MM 比 GD 好上許多，不過是找到 $\mathbf{x}$ 還是 $f(\mathbf{x})$，MM 都比 GD 精準。從左下與右下的圖也可以發現，MM 收斂地比 GD 快，並能跑出較好的解。

5 Conclusion

在整個實驗的過程，我發現不管是 GD 還是 MM 都非常受超參數（$x_0, lr, iter$）的影響。調得好，GD/MM 都能找到不錯的解，但調得不好，GD/MM 就會發散，找到的解非常大或非常小。其中，lr 的影響最大，一旦太大，值根本跑不回來。

這個結論給了我一個重要的觀念，以後訓練 Deep Learning 的模型時，應該**多嘗試幾組超參數**。模型訓練不出來，很可能不是架構的問題，而是沒用對超參數。在這個實驗中，整體而言，MM 表現地比 GD 好上一些，不過 MM 有時容易衝過頭，反而需要迭代更多次才能找到方向。

最後，老師給的 Banana Function 寫錯了，少了一個平方項。真正的 Banana Function 應該一個四次的函式。我在實驗中也嘗試拿我寫的 2 個方法去優化真正的 Banana Function，但結果慘不忍睹，只有少數幾組可以找到最佳解，大部份情況都很慘，所以我就不放上來了。