tags: 人工智慧導論

人工智慧導論 HW1

Introduction

我們將本次作業簡單分成兩個部分，分別為以下。

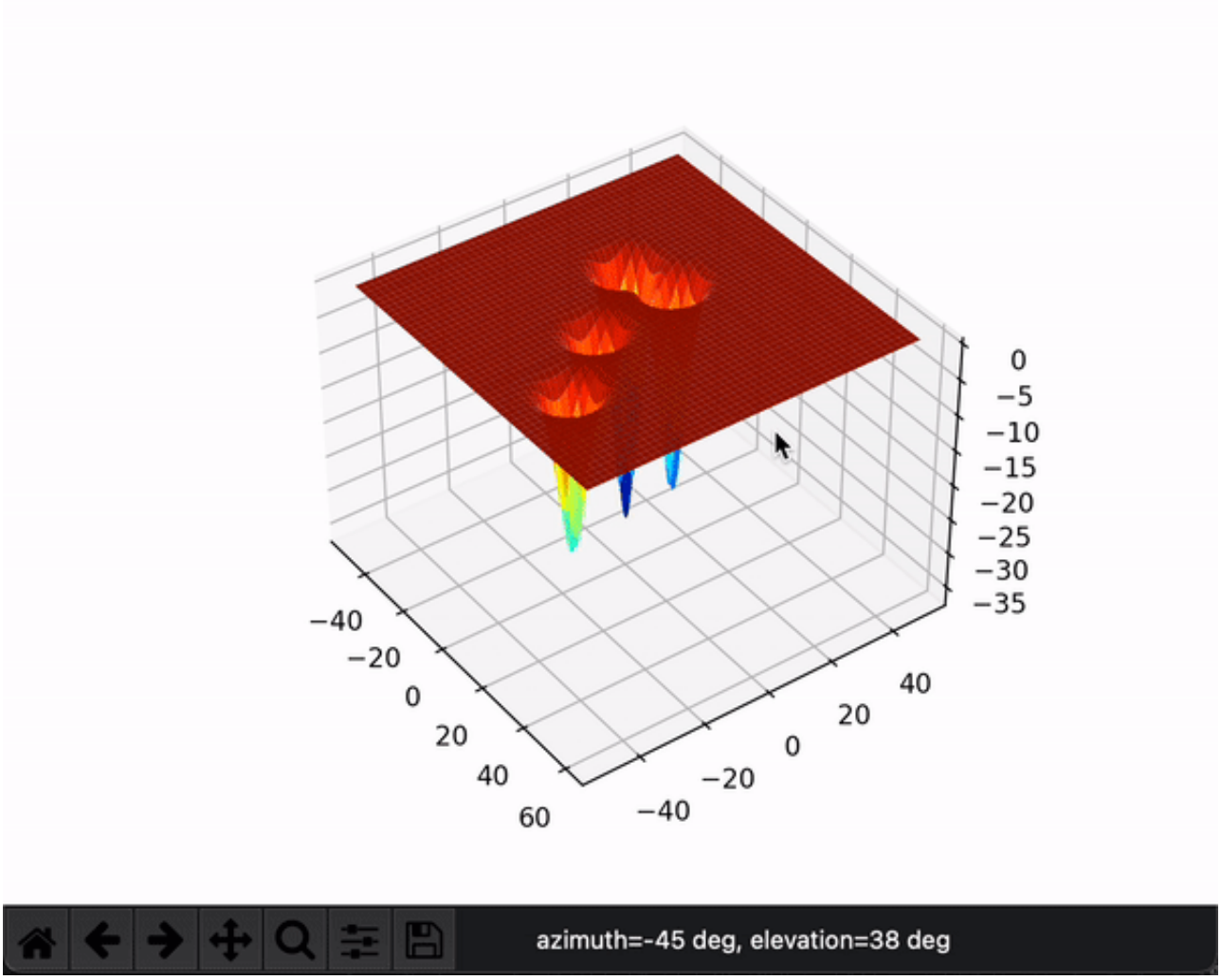
- Pretreatment
  - 使用現成的算法先求得正解，用於比對後面自己實作的答案並且在後續的觀察用以做比較對象。
- Implement
  - 分別實作作業要求的 `Brute Force` 以及 `Local Beam Search` 算法

Pretreatment

在這步驟使用 `Scipy` 的 `Basin hopping` 算法用以求解，需要注意的部分是迭代的次數默認為 100，如果不調整的話就會很容易卡在 `local minimum` ，這邊在嘗試過後選擇將次數調整到 10000 次，下方是得出的結果。

```
                        fun: -37.00136249960556
 lowest_optimization_result:       fun: -37.00136249960556
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
      jac: array([7.10542677e-07, 0.00000000e+00])
  message: 'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 18
      nit: 4
     njev: 6
   status: 0
  success: True
        x: array([-14.99944745,  10.00018416])
                    message: ['requested number of basinhopping iterations
     minimization_failures: 0
                       nfev: 156300
                        nit: 10000
                       njev: 52100
                          x: array([-14.99944745,  10.00018416])
```

接著我們在對整體函數做視覺化 3d 重建，用來確保我們得出的答案沒有錯誤。



Brute Force algorithm

這部分並沒有太多的實作細節，主要將 x 軸以及 y 軸的座標交叉遍歷，需要注意的部分僅有精度的調控，差一位就是 100 倍的差距，所以我在實作上從小數點下第一位開始慢慢往下調控，能得到的最佳精度大概就是小數點下第二位，因為再往下一位就需要花費五小時以上才能得到答案。

> Global minimum: -37.001361997892886
> x axis: -15.00
> y asix: 10.00

Local Beam Search

LBS 在本質上是一種 seq2seq 的算法常用於翻譯等模型上，剛開始我們需要先選定一個 `beam_width` ，假如我選定其為 3 的話，那代表之後我們會挑出三種可能的答案。
首先我們需要從所有備選答案中挑出機率最高的三個詞彙，並且在一個列表內維護他，接下來我們需要針對這三個詞彙各自在備選答案中挑出接他在後面機率最高的詞彙，所以如果我們的備選答案有 10000 個的話在這一步我們就會從三萬種可能中挑出其中三個，最後就是一直重複前面的步驟直到再也沒有答案可選，那麼列表中就是最有可能的三個答案。
回歸到題目，備選答案的值域是一個未知的函數，因此對我們而言每一步的機率都會相同，那麼 LBS 算法在這裡就會退化成每一步只需要找出八方位中最小的值，而且上一步驟並不會影響下一步驟挑選答案的機率，所以答案列表中對於每個可能的答案我們也只需要維護當前的最小值就好了。
經過上述分析在整個算法中我們唯一需要擔心的議題只有如何更好的挑出 `beam_width` 個起點來盡可能的讓找到 `global_minimum` 的機率更高，鑑於這是一個未知函數，所有點找到最佳答案的機率都一樣，那麼將 `beam_width` 個起點平均分配在可能的值域上便會是我們的最佳測略。
在實作上我均攤的做法是將 `beam_width` 開根號得出一個 k，接下來在 `x` 上每隔 $abs(x_{max} - x_{min}) \div k$ 就取出一個點， `y` 也同理，然後將這些點交叉配對，那麼我的起始點就是這 K^2 個點了
最後將答案列表 sort 就能得出 global minimum 為下。

> Global minimum: -37.001361997892886
> x axis: -15.00
> y asix: 10.00

在這邊可以清楚見到我們離真正答案還是有些許誤差的，為了修正這誤差但又不增加太多時間複雜度所以我對 LBS 算後續做了一些修改，我們首先維持大誤差的精度找出我們 $top(width)$ 優良的答案，接下來我們只保存前 $\sqrt{width}$ 個答案，然後將我們每次搜尋的步伐數值乘以 10^{width}，這樣我們就可以變向降低合法值域來提高我們的精度，同時也不會增加太高的時間複雜度，最後得到的答案如下。

```
Global minimum: -37.00136249590535
x axis: -14.999400000000001
y asix: 10.0002
```

函數部分有提供 `width` 可以修改，預設為 10,000，這是考量到所需時間跟精度的 cp 值才這樣設定的，如果要更高精度的話可以改成 1,000,000，但會需要花費約莫 1-2 分鐘 (在 x 及 y 值域約為 100 時)

Comparison

如果我們將精度提高，相對比下就能很清楚地發現在 BF 算法中是用時間來換取精度，而在 LBS 算法中則是反過來用精度來換取時間，下面我們分別列出兩種方法的優劣。

- Brute Force
  只要設定好精度就絕對能夠找到答案，某方面意義上可能是最準確的算法了，但反之帶來的就是時間複雜度的大幅提升，在兩個維度下每提升一次精度就是百倍的差距了，如果在真實多維情況下提升的時間將會是很驚人的，因次我認為 BF 算法是可用，但僅限在特定情況下。
- Local Beam Search
  LBS 的優點就是速度快，並且在犧牲了部分精度下還是能得出不錯的結果，但缺點也隨之而來，如果 Beam width 設定太小就很容易卡在 local minimum，而如果太大就很容易讓每個區塊都重疊進而增加時間複雜，因此在這算法我認為需要去關注的參數便在於 Beam width 的設定。

Conclusion

其實也能將這兩種算法互補用來得到更好的結果，例如可以先用 LBS 算法獲得機率最高的解答所在的區域，再使用 BF 算法來提高精度，等等之類的，我認為還有很多優化的方法可以實作。

在這次作業中也還有很多新的想法想要實作看看，例如將 Basin hopping 的迭代做法應用在 LBS 上或是運用分治想法來優化 LBS 等等，未來有空希望能學到更多相關優化做法。