Deep vs Shallow

1. Simulate a function:

我們使用三種不同的DNN，使用RELU激活函數，參數均為523個

Net1(
  (fc1): Linear(in_features=1, out_features=174, bias=True)
  (fc2): Linear(in_features=174, out_features=1, bias=True)
)

Net2(
  (fc1): Linear(in_features=1, out_features=8, bias=True)
  (fc11): Linear(in_features=8, out_features=16, bias=True)
  (fc12): Linear(in_features=16, out_features=16, bias=True)
  (fc13): Linear(in_features=16, out_features=5, bias=True)
  (fc2): Linear(in_features=5, out_features=1, bias=True)
)

Net3(
  (fc1): Linear(in_features=1, out_features=6, bias=True)
  (fc11): Linear(in_features=6, out_features=5, bias=True)
  (fc12): Linear(in_features=5, out_features=6, bias=True)
  (fc13): Linear(in_features=6, out_features=6, bias=True)
  (fc14): Linear(in_features=6, out_features=7, bias=True)
  (fc15): Linear(in_features=7, out_features=7, bias=True)
  (fc16): Linear(in_features=7, out_features=7, bias=True)
  (fc17): Linear(in_features=7, out_features=7, bias=True)
  (fc18): Linear(in_features=7, out_features=6, bias=True)
  (fc19): Linear(in_features=6, out_features=6, bias=True)
  (fc110): Linear(in_features=6, out_features=6, bias=True)
  (fc111): Linear(in_features=6, out_features=6, bias=True)
  (fc2): Linear(in_features=6, out_features=1, bias=True)
)

我們使用兩種擬合函數，函數1是: **sin(5*x)+2*x**；函數2是: **5*sin(5*x)**

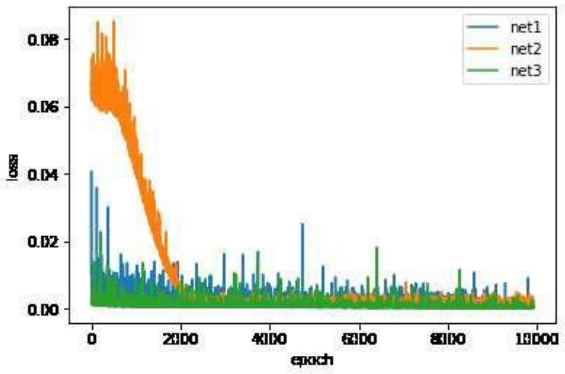


圖1.1.1 函數1的loss

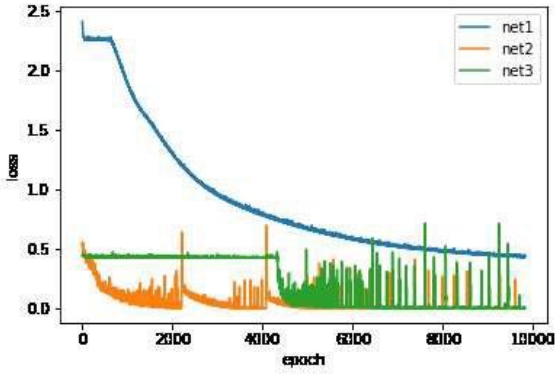


圖1.1.2 函數2的loss

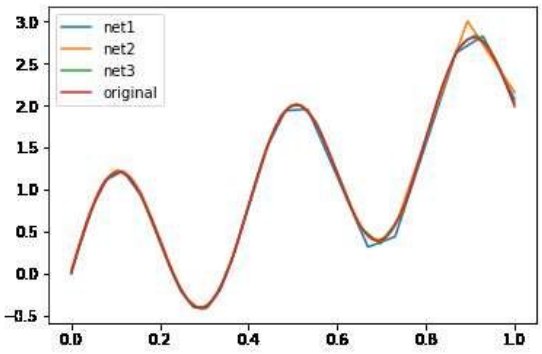


圖1.1.3 函數1的擬合曲線圖

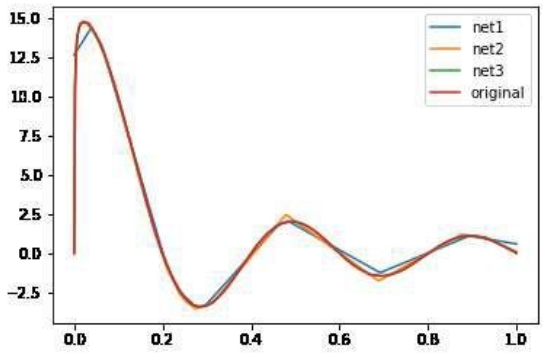


圖1.1.4 函數2的擬合曲線圖

我們發現對於較為簡單的函數（函數1），三個神經網絡最後擬合效果接近一致；對於較為複雜的函數（函數2），淺層網絡（net1）不能很好擬合。在高度彎折的部分，淺層網絡（net1）也無法很好擬合，出現尖角。而深層網絡（net3）可以完全擬合，無尖角。因此層數越深，可表述的特徵越多（相同參數個數的情況下）

2. Train on actual task:

CIFAR-10

用了三個network:

1. Shallow CNN, **2937374** parameters

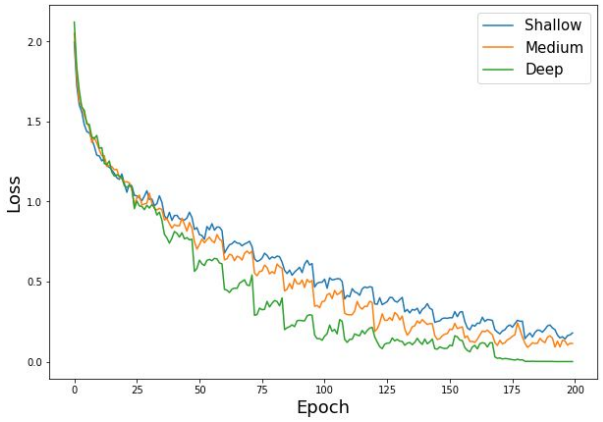
```
Net(
  (pool): MaxPool2d(kernel_size=(8, 8), stride=(8, 8), dilation=(1, 1))
  (conv1): Conv2d (3, 250, kernel_size=(9, 9), stride=(1, 1))
  (fc1): Linear(in_features=4096, out_features=700)
  (final): Linear(in_features=700, out_features=10)
)
```

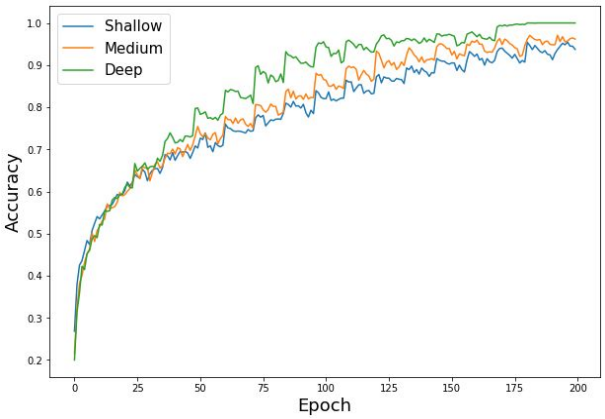
2. Medium CNN, **2913866** parameters

```
Net(
  (pool): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (conv1): Conv2d (3, 128, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d (128, 64, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=1600, out_features=1024)
  (fc2): Linear(in_features=1024, out_features=1024)
  (final): Linear(in_features=1024, out_features=10)
)
```

3. Deep CNN, **2897162** parameters

```
Net(
  (pool): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (conv1): Conv2d (3, 96, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d (96, 64, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d (64, 64, kernel_size=(1, 1), stride=(1, 1))
  (fc1): Linear(in_features=1600, out_features=1024)
  (fc2): Linear(in_features=1024, out_features=1024)
  (final): Linear(in_features=1024, out_features=10)
)
```





我使用CNN來train CIFAR10，從圖中可以看到，越深的model能越快的收斂。

MNIST

1. Shallow DNN, total 7950010 parameters

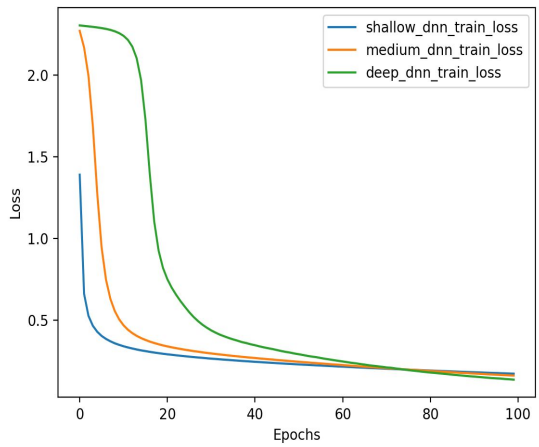
```
Net_DNN1(
  (fc1): Linear(in_features=784, out_features=10000)
  (fc2): Linear(in_features=10000, out_features=10)
)
how many parameters: 7950010
```

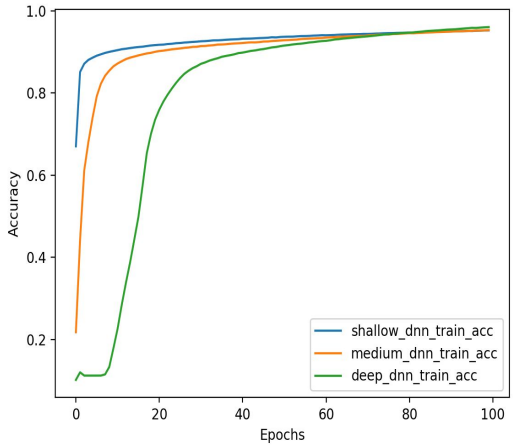
2. Medium DNN, total 7794410 parameters

```
Net_DNN2(
  (fc1): Linear(in_features=784, out_features=1000)
  (fc2): Linear(in_features=1000, out_features=5000)
  (fc3): Linear(in_features=5000, out_features=400)
  (fc4): Linear(in_features=400, out_features=10)
)
how many parameters: 7794410
```

3. Deep DNN, total 7798710 parameters

```
Net_DNN3(
  (fc1): Linear(in_features=784, out_features=1300)
  (fc2): Linear(in_features=1300, out_features=1300)
  (fc3): Linear(in_features=1300, out_features=1300)
  (fc4): Linear(in_features=1300, out_features=1300)
  (fc5): Linear(in_features=1300, out_features=1300)
  (fc6): Linear(in_features=1300, out_features=10)
)
how many parameters: 7798710
```





結果：前半個epochs中，越deep的DNN會越久到達跟shallow的DNN一樣的效果，但是最後幾個epochs，deep的DNN表現得比shallow的DNN還要好。

Optimization

1. Visualize the optimization process:

我們用CNN在MNIST上進行訓練，使用ADAM算法（learning rate = 0.001）。我們每兩個epoch記錄一次數據，最後使用PCA降維，我們使用不同的初始化條件進行訓練。見圖2.1.1和2.1.2。其中不同點點顏色表示不同初始化條件，點的大小表示loss（cross entropy）的值（點大代表loss較大），紅色數字表示loss值。

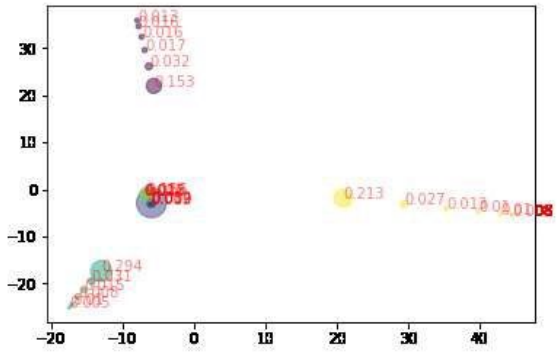


图 2.1.1 Parameters of the whole layer

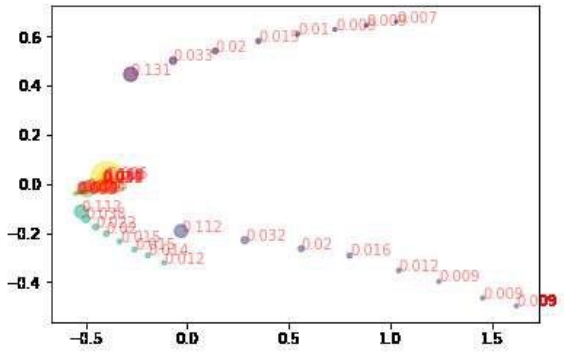


图 2.1.2 Parameters of the second convolutional layer

我們可以看到每次訓練都在沿一條近似直線的方向遠離初始點，而某一特定層的參數向著特定的方向移動。

2. Observe gradient norm during training

我們先針對 f(x) = sin(pi * 5 * x) / (x + 0.001)，取出 (0, 1) 間均勻的10000個點為training data，用三層皆為**21** neurons的network進行擬合，使用Adam來train。

圖 2.2.1 batch size 64

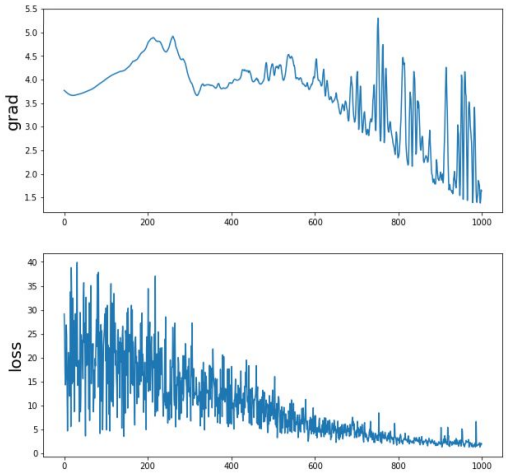


圖 2.2.2 full batch

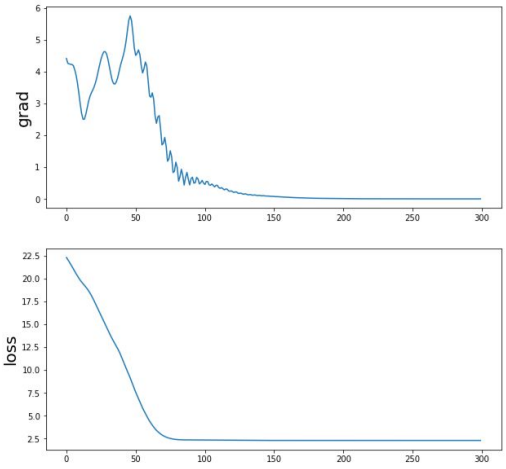


圖2.2.1為batch size = 64的訓練過程(橫軸為epoch)，可以看到震盪的幅度很大，但loss跟grad都有漸漸下降。訓練後期grad norm常有大幅度擺動，這是因為訓練為stochastic，每次僅看少量data然後

針對該筆data更新，對於不同batch更新的方向不同外，error surface也不同，因此算出來的grad norm會巨幅震盪。當我們用full batch來train，則如圖2.2.2，比起small batch size的train法，這裡grad norm很穩定的下降，畢竟圖中每個點都是在同一個error surface上對grad norm做gradient descent。不過穩定的grad norm也不代表比較好(只是圖看起來比較漂亮)，因為用large batch size來train通常generalization較差。我想可能因為用full batch train時，error surface就只有一個。只看一個surface來train，model就較經不起考驗。可以看到圖2.2.1中，儘管train loss都很低了，grad norm還是瘋狂震盪。代表每個batch的error surface都一再考驗model是否能同時朝每個surface的最低點走，而不只是僅在一個surface上得到好的「平均的」結果，但有可能是在某些batch上極好，某些batch上極差。另外我們也有在MNIST上觀察：

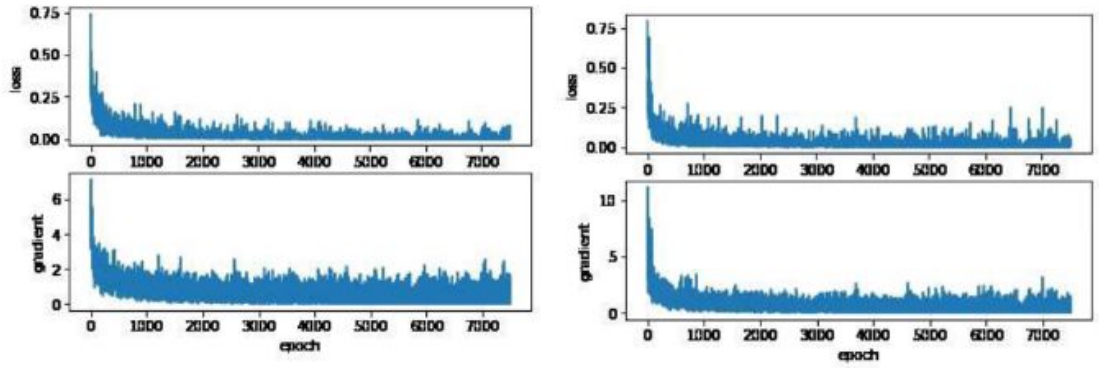


图 2.2.3 MNIST上CNN的loss和gradient norm（左图 learning rate=0.001 右图 learningrate=0.0005）

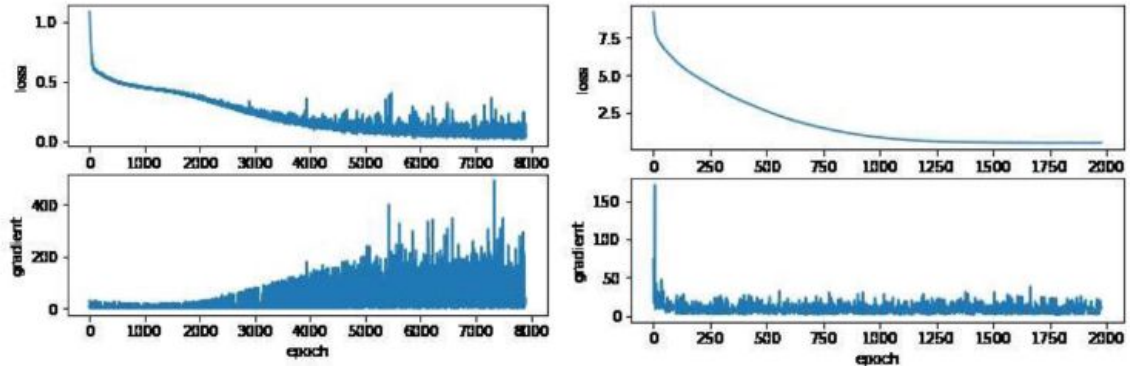


圖 2.2.4 擬合函數曲線的DNN的loss和gradient norm (左圖 learning rate=0.001 右圖 learningrate=0.0005)
我們發現loss降到接近0時gradient依舊在震蕩，而且learning rate越大gradient越不穩定。

3. What happens when gradient is almost zero:

我們同樣針對 f(x) = sin(pi * 5 * x) / (x + 0.001)，取出 (0, 1) 間均勻的10000個點為training data，用三層皆為3 neurons的network進行擬合。原本採取的train法是：**<u>先以RMSE為目標函數，再換成以gradient norm為目標函數 (作法A)</u>。**並沒有使用Newton's method，因為實作後發現用Hessian matrix來update參數，雖然在某些時候的確有快速、一步到低點的效果，但相當不穩定，有時反而會讓loss直線飆升到overflow。這可能是因為train到越後期Hessian matrix解出來的eigenvalues時常

出現越多0而deganerate，單用二次微分無法知道要往哪走最好。且若一開始就處在concave上的一點 (而不是convex)，則loss會飆升也很合理。作為修正後來我們試著實作Levenberg-Marquardt，的確能比Newton's method穩定很多，且收斂得較做法A快。圖2.3.1將其訓練過程畫成如上一題的圖。因此最後也採取：**先以RMSE為目標函數，再用Levenberg-Marquardt (作法B)**。第二階段中尋找critical point時，Levenberg-Marquardt通常僅需要3~5 epoch即可找到gradient norm夠小的點，而以gradient norm為目標函數的做法則需要至少數十個epoch。訓練結果如圖2.3.2，作法A畫了100個點，做法B畫了20個點。兩種做法都能看到類似的結果：loss越低，min ratio越高。我們的min ratio的定義和助教投影片一樣：**（大於零的eigenvalues #）/（所有eigenvalues #）**。

圖 2.3.1 Leveberg-Marquardt

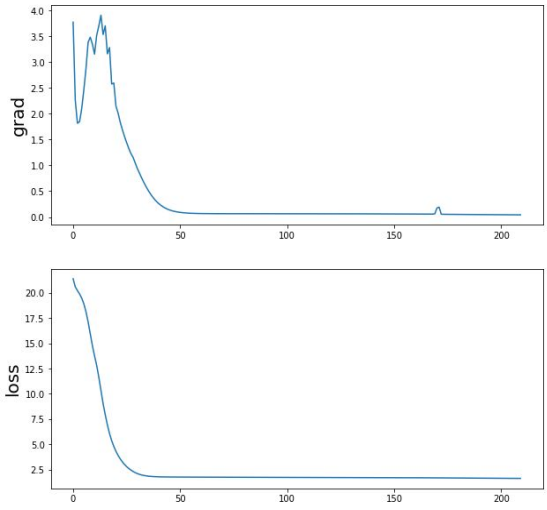


圖 2.3.2 loss v.s. Minimum ratio

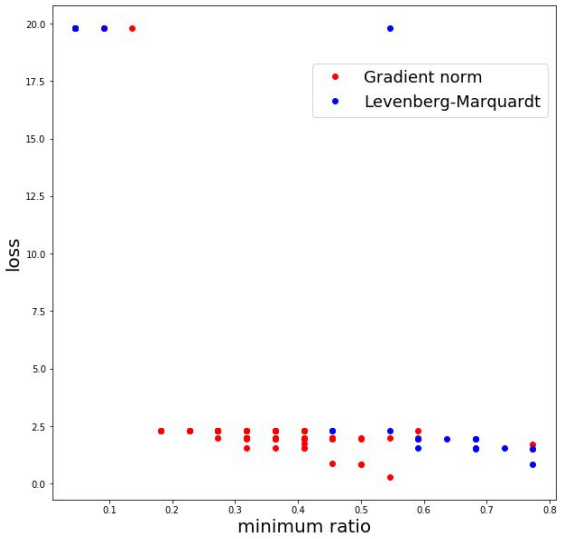
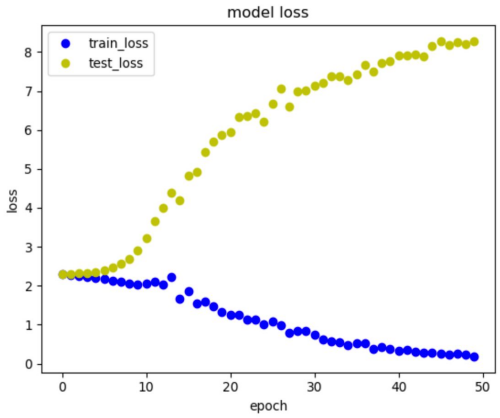


圖2.3.2中，藍色點有一個特別靠右(簡稱X)，似乎顯示loss = 19附近也有個很接近local min的位置。然而實際觀察其eigenvalues發現為一半正數一半負數，且數值不小，為一個saddle point。而其他loss = 19的點，eigenvalues幾乎為0，代表相當平坦，也說明了X其實還沒到達高原的中心，還在外圍，因此尚且可以用Hessian matrix判斷為saddle point。對於那些已經跑進高原中心的點，就只能看到一堆0的eigenvalues，不確定該點的性質。X還在高原外圍，gradient norm應該會稍大才對，然而其grad norm卻比一些真正在高原中心的點還要略小。實際看了X的gradient數值後發現，大部分為0，僅兩個數值特別高(0.005506和0.005601)。代表雖然我以為grad norm小於0.008已經夠小(code中小於0.008後就會認定該點為critical point。對於22個參數的network，平均每個參數的gradient要小於約0.001)，但沒考慮到如果grad norm的貢獻完全只來自一個參數，則該參數的gradient仍然相當大(最大可以就是0.008)。雖然只有0.008，但相對於其他位在高原中心的點(所有參數gradient都很小)明顯還沒有train好。因此會出現X點是我們設定gradient norm threshold時較寬鬆的緣故。

Generalization

1. Can network fit random labels?

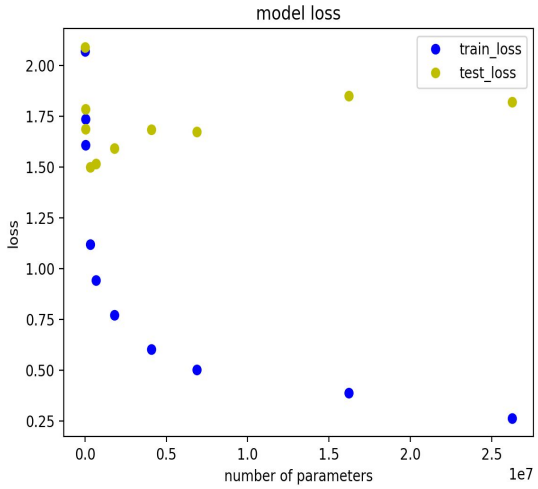
我們在CIFAR10上做訓練，epochs為128，batch size為100，optimizer為SGD，learning rate為0.1，momentum為0.5，架構為2個hidden layer，再加上一個output layer，每個layer之間皆為DNN。

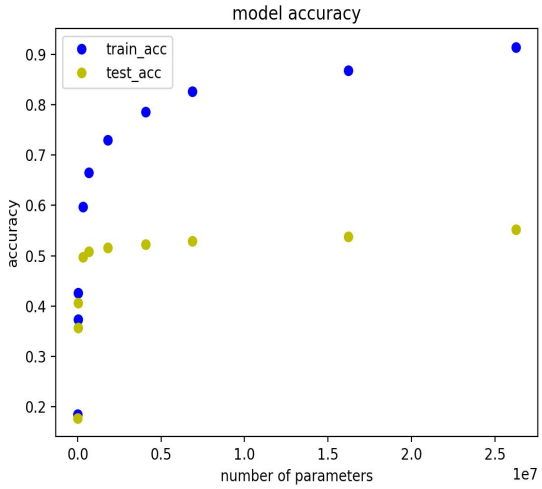


結果：epoch越多，train loss越低，代表network有從random shuffle training data強記東西，但在test上，test loss反而越來越高。

2. Number of parameters v.s. Generalization

我們在CIFAR10上做訓練，epochs為10，batch size為100，optimizer為SGD，learning rate為0.2，momentum為0.5，架構均為2個hidden layer，再加上一個output layer，每個layer之間皆為DNN





結果：做的model數不夠多，無法像上課時老師所說的，參數量越大，test_loss仍是下降的，但是test_acc卻是有稍微增加的趨勢，顯然當遇到參數量過多而導致overfitting時，不妨可以讓參數量增加更多，即使overfitting的情況更加嚴重，test_acc還是會增加的。

3. Flatness v.s. Generalization

PART1

我們在MNIST上訓練，網絡結構與初始化參數相同，使用ADAM算法，learningrate也保持為0.001，紅色表示正確率，藍色表示cross-entropy，其中cross-entropy為log後的結果。圖3.3.1中alpha=0表示batch-size為8，alpha=1表示batch-size為512；圖3.3.2中alpha=0表示batch-size為16，alpha=1表示batch-size為32；圖3.3.3中alpha=0表示batch-size為1024，alpha=1表示batch-size為16。

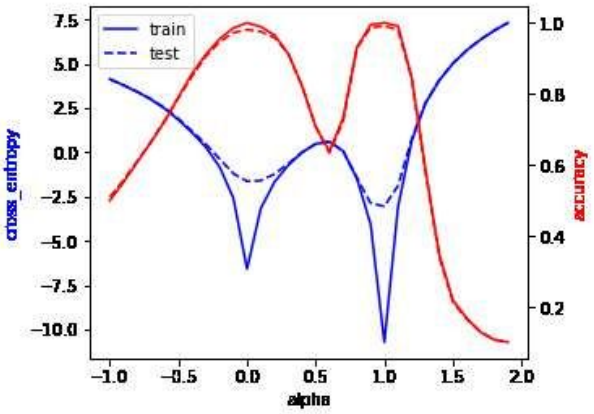


圖 3.3.1

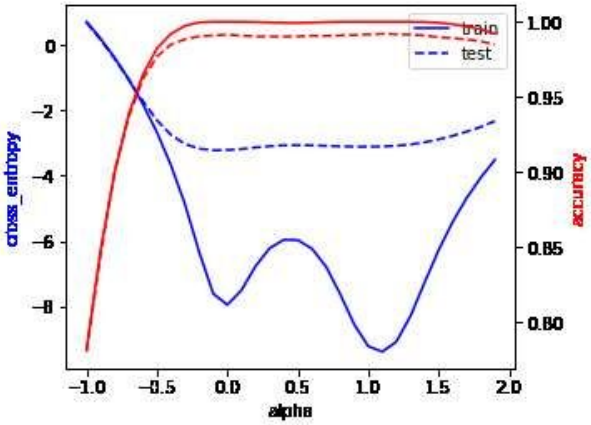


圖 3.3.2

可以看到，batch-size較大時loss函數很尖銳，而batch-size越小loss越平滑，說明generalization效果越好。

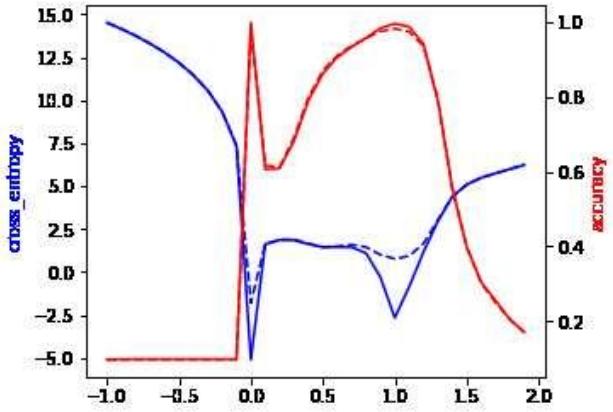


圖 3.3.3

PART2

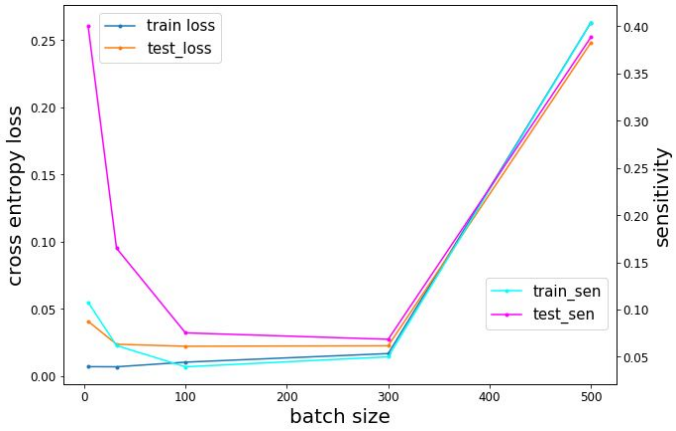


圖 3.3.4

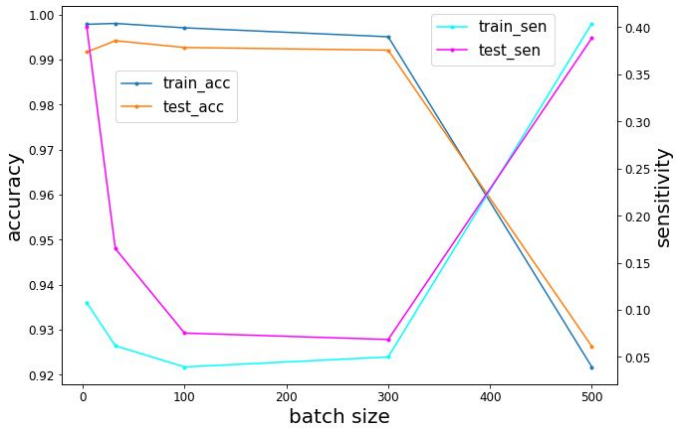


圖 3.3.5

Sensitivity部分也是在MNIST上使用相同的network(但與PART1不同)訓練。train了五個不同batch size的approach，分別是4, 32, 100, 300, 500。都是用Adam，不過有不斷調整learning試圖讓model有完全train到底。從圖3.3.4和圖3.3.5中可以看到，sensitivity相當能作為判斷model是否有generalize的依據。在batch size為512的時候，因為batch size太大，loss走不下去造成在train set和test set上的loss和accuracy表現都不好，而此時sensitivity也的確都相當的大。而在batch size為4的時候，雖然train loss極低且train accuracy極高，但可以看到在training set的sensitivity較高，甚至testing set的sensitivity超高，同時test上的loss和accuracy也明顯比batch size為32時差。因此sensitivity的確有助於反應model是否generalize，且不只在testing set，在training set上也能有所反應，因此也的確可以把sensitivity嵌入optimizer來試著找到最能generalize的model。雖然這次實驗似乎與上一個的結論有所衝突，不過其實batch = 32的表現仍然是最好的。在後面的bonus中也看得到，batch size太小的時候，似乎會overfit且走到較尖銳的地方。

BONUS

我們在MNIST上進行訓練，網絡結構與初始化參數相同，使用ADAM算法，learningrate也保持為0.001。我們使用一種特殊方法計算flatness：首先讓一個特定的model按照一種訓練途徑（batch size固定）訓練到底（正確率達到100 %），然後計算loss（cross entropy）關於input的梯度，然後對此梯度進行normalization，即得到loss關於input的方向導數，沿此方向即為loss增加最快的方向。然後沿著這個方向，按照固定步長畫出loss面，如圖3.4.1, 2, 3。其中，不同顏色線條表示不同的batch-size的loss。我們設距離原點為-2.5（stepsize=-2.5）的loss與原點loss的差為sharpness。得到圖3.5，藍線表示數據的loss，紅線表示sharpness。在圖3.5中，我們發現batchsize超過一個範圍後，loss面開始顯著陡峭，sharpness急速增加。

圖 3.4.1

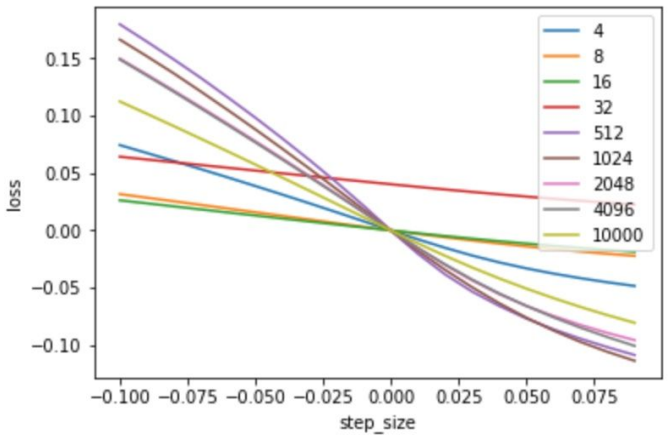


圖 3.4.2

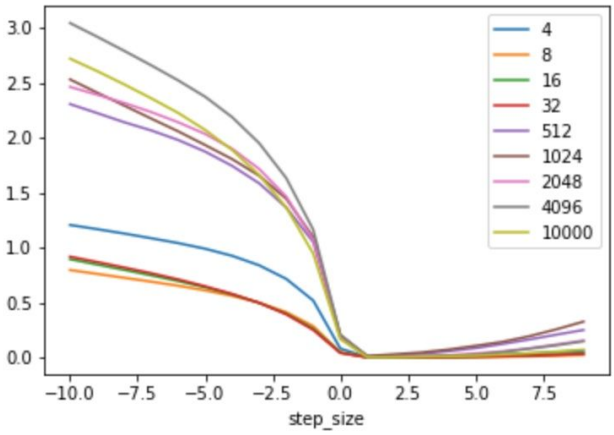


圖 3.4.3

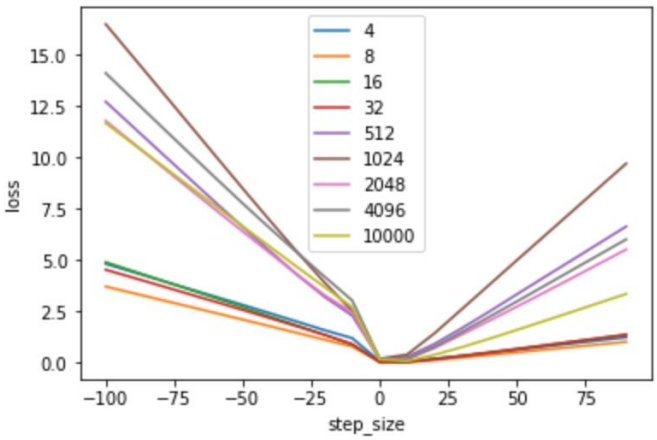


圖 3.5