採用爬山演算法自動建構神經網路模型 – 以 MNIST 為例

開放原始碼論文

陳鍾誠
國立金門大學資訊工程學系
ccc@nqu.edu.tw

2021/10/15

摘要

本論文的開放原始碼專案網址為：https://github.com/cccresearch/nnModelAuto/

以人腦建構神經網路或深度學習模型，通常得依賴研究者的直覺。

但如果能由程式自動建構神經網路的架構，除了不需要依賴人腦的直覺之外，還有可能建構出人腦所難以想出來的模型。

本論文針對手寫數字辨識問題，在 MNIST 資料集上，採用爬山演算法進行了初步的《自動建構神經網路》實驗！

關鍵字 神經網路 · 深度學習 · MNIST · 自動建模

1 簡介

深度學習的神經網路的學習演算法，目前仍然以梯度下降法為主流，透過自動微分的反傳遞方式，計算出梯度之後，像梯度方向邁出微小的步伐。

然而若要自動學習神經網路模型，由於架構的調整通常無法化成連續函數，因此難以使用梯度下降法自動調整網路架構。

不過若採用傳統人工智慧中的《搜尋法或優化算法》，則不需要透過梯度，對網路架構調整這樣的問題而言，採用傳統方法似乎更適合！

本文嘗試使用傳統人工智慧中簡單的《爬山演算法》，針對手寫辨識 MNIST 測試集，自動建構神經網路模型。

我們的實驗結果顯示，使用簡單的爬山演算法，就能從一個單層線性網路的架構開始，在正確率上逐步攀爬，最後得到一個還不錯的多層架構，讓正確率從 91.79% 提升到 98.05%。

當然，若採用其他的優化方法，例如 Best-First Search 等，或許會比爬山演算法表現更好，因此本實驗只能算是利用優化方法自動建模的一個初步嘗試。

2 背景

神經網路的研究從早期的《單層感知器》開始，到了 Hinton 等人在 1986 年 [1] 重新發明反傳遞演算法，並成功應用在語音辨識等問題之後，有過一陣研究熱潮。

熱潮退去後，仍有些研究者持續改良神經網路模型，像是 LeCun [2] 等人所提出的卷積神經網路，在影像識別領域就有優異的表現。

2011 年繪圖處理器硬體上的進展，以及 ImageNet 等大量測試集的出現，引發了神經網路的新一波研究發展，很多論文改良了卷積神經網路的架構，讓神經網路的層數加深並在許多領域表現優異，這些新發產被統稱為《深度學習技術》[3] [4] [5] 。

然而這些神經網路架構，通常是由創造者的知識經驗所設計出來的。

既然神經網路可以建構出《影像辨識、語音辨識、語言生成》等模型，那麼我們能否用程式自動建構出神經網路呢？

目前、這類的研究並不多 [6] [7] ，但已經逐漸有研究者投入，本研究也是我們的一個初步嘗試。

3 方法

爬山演算法是模仿爬山的動作，只要看到附近有更高的點，就往那個方向爬，寫成演算法如下所示：

```
Algorithm HillClimbing(f, x)
    x = 隨意設定一個解。
    while (x 有鄰居 x' 比 x 更高)
        x = x';
    end
    return x;
end
```

由於要比較高低，因此通常會設定固定的高度函數 height()，透過 height() 去比較兩個解答的高度，然後決定新解達是否比舊解答的高度更高；若新的更高則移動過去，否則就繼續找下一個鄰居，直到找到更高的鄰居，或者嘗試了很多次都找不到，那就認為已經爬到某個山頂，於是結束離開。

以下 Python 程式是上述演算法的更詳細版本，也是本文實驗所採用的方法！

```python
def hillClimbing(s, maxGens, maxFails):    # 爬山演算法的主體函數
    global file
    file = open('./model/hillClimbing.log', 'w')
    log(f"start: {str(s)}")                # 印出初始解
    fails = 0                              # 失敗次數設為 0
    # 當代數 gen<maxGen，且連續失敗次數 fails < maxFails 時，就持續嘗試尋找更好的解。
    for gens in range(maxGens):
        snew = s.neighbor()                #  取得鄰近的解
        # log(f'snew={str(snew)}')
        sheight = s.height()               #  sheight=目前解的高度
        nheight = snew.height()            #  nheight=鄰近解的高度
        # log(f'sheight:{sheight} nheight:{nheight}')
        if (nheight > sheight):            #  如果鄰近解比目前解更好
            log(f'{gens}:{str(snew)}')     #    印出新的解
            s = snew                       #    就移動過去
            fails = 0                      #    移動成功，將連續失敗次數歸零
        else:                              #  否則
            fails = fails + 1              #    將連續失敗次數加一
        if (fails >= maxFails):
            log(f'fail {fails} times!')
            break
    log(f"solution: {str(s)}")             #  印出最後找到的那個解
    file.close()
    return s                               #    然後傳回。
```

上述演算法有兩個重要的函數未交代清楚，一個是 height() ，另一個是 neighbor() 。

在我們的實驗中，採用《正確率 - 神經網路複雜度》作為高度的衡量。其中的神經網路複雜度設定為《網路的參數數量/一百萬》，對應的 Python 程式碼如下。

```
def height(self):
    net = self.net
    if not model.exist(net):          # 如果之前沒訓練過這個模型
        trainer.run(net)              # 那麼就先訓練並紀錄正確率
    else:
        jsonObj = model.load(net)     # 載入訓練結果
        net.model['accuracy'] = jsonObj['model']['accuracy'] # 取得正確率

    # 傳回高度 = 正確率 - 網路的參數數量/一百萬
    return net.model['accuracy']-(net.model['parameter_count']/1000000)
```

這樣的高度設計並非是最好的，而且有人為調整的空間，目前採用這個公式只是個初步嘗試。

鄰居函數 neighbor() 的設計，則是採用《隨機選取操作》的方式，可用的操作有《新增與修改》，其中新增是增加一個神經網路層，而修改則是將一個網路層取代為另一個隨機產生的網路層。

```
def neighbor(self):
    model = copy.deepcopy(self.net.model)     # 複製模型
    layers = model["layers"]                  # 取得網路層次
    in_shapes = self.net.in_shapes            # 取得各層次的輸入形狀
    ops = ["insert", "update"]                # 可用的操作有新增和修改
    success = False
    while not success:                        # 直到成功產生一個合格鄰居為止
        i = random.randint(0, len(layers)-1) # 隨機選取第 i 層 (進行修改或新增)
        layer = layers[i]
        op = random.choice(ops)               # 隨機選取操作 (修改或新增)
        newLayer = randomLayer()              # 隨機產生一個網路層
        if not compatable(in_shapes[i], newLayer["type"]): # 若新層不相容 (輸入維度不對)
            continue                          #   那麼就重新產生
        if op == "insert":                    # 如果是新增操作
            layers.insert(i, newLayer)        #   就插入到第 i 層之後
        elif op == "update":                  # 如果是修改操作
            if layers[i]["type"] == "Flatten": # 不能把 Flatten 層改掉
                continue                      # (因為我們強制只能有一個 Flatten 層)
            else:
                layers[i] = newLayer          # 若不是 Flatten 層則可以修改之
        break

    nNet = Net()                              # 創建新網路物件
    nNet.build(model)                         # 根據調整後的 model 建立神經網路
    return SolutionNet(nNet)                  # 傳回新建立的爬山演算法解答
```

隨機產生網路層時，為了避免過多的可能鄰居，導致爬山演算法過慢，我們限縮了鄰居的可能性，只有下列程式中 types 指定的六種網路層可以選取。

而在參數方面，大小與通道數也都不能任意選，基本上都是以 2 的次方為選擇項，這樣才不會有太多的鄰居可選。

Table 1: 不同模型的 MNIST 正確率

編號	模型	正確率	參數數量	高度	說明
0	Flatten	91.79%	7850	91.782151	起點
1	ReLU+Flatten	91.91%	7850	91.902154	
5	ConvPool2d(8)+Flatten	92.08%	13610	92.066392	
9	ConvPool2d(8)+ReLU+Flatten	93.37%	13610	93.356393	
40	Conv2d(4)+ReLU+Flatten	95.98%	27090	95.952913	
47	Conv2d(4)+Conv2d(8)+ReLU+Flatten	97.64%	46426	97.593573	
49	Conv2d(4)+Conv2d(8)+Conv2d(32)+ReLU+Flatten	98.05%	157562	97.892441	

```
types = ["ReLU", "Linear", "Conv2d", "AvgPool2d", "LinearReLU", "ConvPool2d"]
sizes = [ 8, 16, 32, 64, 128, 256 ] # 限縮大小選取範圍，不是所有整數都可以
channels = [ 1, 2, 4, 8, 16, 32 ]   # 限縮通道數範圍

def randomLayer():
    type1 = random.choice(types)              # 隨機選一種層次
    if type1 in ["Linear", "LinearReLU"]:     # 如果是 Linear 或 LinearReLU
        k = random.choice(sizes)              #   就隨機選取 k 作為輸出節點數
        return {"type":type1, "整里程表tures":k}
    elif type1 in ["Conv2d", "ConvPool2d"]:   # 如果是 Conv2d 或 ConvPool2d
        out_channels = random.choice(channels) # 就隨機選取 channels 數量
        return {"type":type1, "out_channels": out_channels}
    else:
        return {"type":type1}                 # 否則不須設定參數，直接傳回該隨機層。
```

必須小心的是，並不是所有隨機產生的層都可以任意插入，因此必須先檢查相容性 (輸入維度是否正確) 後才能進行《新增與修改》動作，以下是相容性檢查的算法。

```
types2d = ["Conv2d", "ConvPool2d", "AvgPool2d", "Flatten"]
types1d = ["Linear", "LinearReLU"]

def compatable(in_shape, newLayerType):
    if newLayerType in ["ReLU"]: # 任何維度都可以使用 ReLU 操作
        return True
    elif len(in_shape) == 4 and newLayerType in types2d:
        # 這些層的輸入必須是 4 維的 (1. 樣本數 2. 通道數 3. 寬 4. 高)
        return True #
    elif len(in_shape) == 2 and newLayerType in types1d:
        # 這些層的輸入必須是 2 維 (1. 樣本數 2. 輸出節點數)
        return True
    return False
```

最後，爬山演算法必須有個起點，我們選擇用單層網路作為起點，也就是只有《Flatten + Linear》所形成的網路，其中 Flatten 是為了將原本 MNIST 輸入的多維《通道 + 影像》結構，攤平成單一維度的結構 (在 PyTorch 中得加上樣本數這個維度，所以是二維結構)。

4 實驗結果

本實驗中的爬山演算法，會在發現更好的模型時，印出該模型，以下是某次實驗整理成表格後的結果：

從表格 1 中，您可以看到爬山演算法從初始模型開始，將正確率從 91.79% 開始逐步提升，每次只會新增或修改一層，最後建構出了 Conv2d(4)+Conv2d(8)+Conv2d(32)+ReLU+Flatten 這個模型，其正確率為 98.05%。

但是模型的參數數量，也從 7850 開始，一路提高到 157562，但套上我們的高度公式得到 (157562/1000000) 大約只扣了 0.158 分，因此該模型分數為 98.05-0.158 = 97.892 ，還是在高度上勝過其他模型。

我們可以透過爬山演算法的高度函數設計，控制參數數量的多寡，例如我們若偏好小型模型，那麼或許可以將高度公式改為《高度 = 正確率 - 參數數量/十萬》，但若我們認為模型大無所謂，那麼使用《高度 = 正確率》這樣的公式也就可以了。

參考文獻

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[2] Yann André LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. 1998.

[3] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, D. Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv*, abs/1502.03167, 2015.

[5] François Chollet. Xception: Deep learning with depthwise separable convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807, 2017.

[6] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, Matthias Urban, Michael Burkart, Maximilian Dippel, Marius Thomas Lindauer, and Frank Hutter. Towards automatically-tuned deep neural networks. In *Automated Machine Learning*, 2019.

[7] Steven Abreu. Automated architecture design for deep neural networks. *ArXiv*, abs/1908.10714, 2019.