深度學習

HW7

學號 : B103012002

姓名 : 林凡皓

1. Linear layer

* Linear.forward :

1. 解題思路 : 先利用view( )將輸入x變成shape為(N, D)的tensor。接著透過forward propagation的公式來計算forward propagation的輸出結果。
2. 執行結果 :



由此結果可以看出，計算結果與預期結果相差很小，代表說這個function功能正確。

1. 額外討論 :

Forward propagation的公式為。之所以是xW是因為考慮到tensor的形狀。out、x、W的形狀分別為(N, M)、(N, D)、(D, M)，因此透過xW的形狀為(N, M)滿足out的形狀可以得知，矩陣乘法順序應該為xW。

* Linear.backward :

1. 解題思路 : 透過computational graph來幫助我們計算微分，computational graph如下

一張含有 文字, 字型, 行, 圖表 的圖片

自動產生的描述

根據上圖推倒結果，完成dx、dw、db即可。

1. 執行結果 :

一張含有 文字, 字型, 螢幕擷取畫面, 印刷術 的圖片

自動產生的描述

由上面結果可以看出，跟預期結果相比，誤差很小，代表說此function功能正確。

1. 額外討論 :

透過computational graph計算出來的dx為，這其實是reshape後的dx，並不是我們要的。因此計算出此結果後，還需要透過reshape將形狀變成 (N, d1, …, d\_k)。

1. ReLU activation

* ReLU.forward :

1. 解題思路 :

透過torch.relu( )即可完成此題。

1. 執行結果 :



由上面結果可以看出，跟預期結果相比，誤差很小，代表說此function功能正確。

* ReLU.backward :

1. 解題思路 :

由可以知道，對此函數做微分的結果為，當x > 0，微分結果會是upstream derivatives dout，而當x<=0，微分結果為0。因此我先將dx令成dout，再透過index將x<=0的部分改成0。

1. 執行結果 :



由上面結果可以看出，跟預期結果相比，誤差很小，代表說此function功能正確。

1. “Sandwich” layers

在神經網路中，通常會出現一些常見的層模式，像是linear layer後面會接上ReLU。為了定義這些層模式，我們可以定義一個convenience layer，也就是說定義一個layer為linear layer和ReLU的組合。透過這種抽象化的概念，我們可以更方便的去管理code架構。接下來就是將剛剛實現的Linear和ReLU做結合，變成一個叫做Linear\_ReLU的convenience layer。

* Linear\_ReLU.forward :

1. 解題思路 : 將input的x、w、b送進剛剛定義好的Linear.forward。接著再將經過Linear.forward的輸出送進ReLU.forward即可。要注意的是，我們需要將過程中的參數記錄下來，以便在backward path中使用。

* Linear\_ReLU.backward :

1. 解題思路 : 先接收從Linear\_ReLU.forward的過程中存下來的參數，接著分別將這些參數送進先前定義好的ReLU.backward和Linear.backward。由於是back propagate，因此要先經過ReLU.backward再經過Linear.backward。
2. 執行結果 :

一張含有 文字, 字型, 螢幕擷取畫面, 黑色 的圖片

自動產生的描述

由上面結果可以看出，跟預期結果相比，誤差很小，代表說此function功能正確。

1. Two-layer network

* \_\_init\_\_ :

1. 解題思路 : 這個為TwoLayerNet物件的初始化。由於是兩層的神經網路，因此會有四個參數要做初始化，分別為W1、W2、b1、b2(他們都是tensor)。W1和W2我選擇利用randn(normal distribution)的方式做隨機的初始化，b1、b2則是初始化為0。
2. 額外討論 :

為甚麼可以直接將b初始化為0但是W卻不行，甚至還要特別使用weight\_scale來控制初始化大小?

b可以初始化為0是因為它其實只是在做平移，並不需要做過多的變化或是隨機性。但是W就不一樣了。W會直接的影像到神經網路的收斂性，在做gradient descent的時候，W設置不好可能會造成所謂的梯度消失或是梯度爆炸，因此通常在做W的初始化都會採用一些隨機初始化的方法。

* TwoLayerNet.loss :

1. 解題思路 :

一張含有 文字, 行, 字型, 圖表 的圖片

自動產生的描述

根據上面的推導的結果以及先前定義好的function來完成。在計算dw的時候，由於先前並沒有加上regularization的影響，因此經過Linear.backward和Linear\_ReLU.backward後還需要加上的部分。

1. 執行結果 :

一張含有 文字, 螢幕擷取畫面, 字型, 黑與白 的圖片

自動產生的描述

可以看出不管有沒有考慮regularization，計算出來的結果誤差都很小，代表說功能正確。

1. Solver

Solver為一個將訓練分類器所需要的東西都封裝起來的類別。要使用Solver，我們需要先創建一個Solver類別，並將dataset、learning rate、batch size……等參數輸入進去。接著可以透過呼叫train方法來訓練模型。訓練完成後會將參數存放到model.params中，訓練過程的loss、training accuracy、validation accuracy會分別存放到solver.loss\_history、solver.train\_acc\_history和solver.val\_acc\_history中。

* \_\_init\_\_ :

初始化權重的想法和TwoLayerNet相同。不一樣的是這次要針對多層的神經網路做初始化。對於每一層來說，W的形狀為(dim\_of\_last\_layer, dim\_of\_current\_layer)，而b的形狀為(dim\_of\_current\_layer, )。

* loss :

loss的部分分為forward path和backward path。

Forward path的部分就是透過for loop不斷的將計算出來的東西送進Linear\_ReLU.forward，除了最後一層是送入Linear.forward。與先前不同的是，這一次還要考慮使用Dropout layer的情況。Dropout的部分會使用到Dropout.forward，因此先說明Dropout.forward的實現。

* Dropout.forward : 利用torch.rand\_like(x)生成與x相同形狀的tensor，並對這個tensor使用mask，讓大於dropout機率的位置設成1，其餘設成0。

接著是backward path的部分。Backward path的主要實現邏輯與forward path相同，都是透過for loop來進行多次的propagation，差別在於backward path是不斷的經過Linear\_RuLU.backward。由於也要多考慮到Dropout的部分，因此這邊需要多實現Dropout.backward。

* Dropout.backward : Dropout.backward的主要概念就是將在forward path過程中，被dropout的部分設成0，其餘為1。
* create\_solver\_instance :

創建一個Solver物件並將要設定的參數輸入進去。這邊嘗試使用的參數如下 : hidden\_dim = 200、update\_rule = sgd、learning\_rate = 0.4、lr\_decay = 0.96、batch\_size = 512、num\_epochs = 15。

* 訓練結果 :

使用create\_solver\_instance所設置的參數後，訓練結果如下

一張含有 文字, 螢幕擷取畫面, 功能表, 黑與白 的圖片

自動產生的描述

最佳的準確度為52.69 %。

將結果視覺化如下

一張含有 文字, 螢幕擷取畫面, 繪圖, 行 的圖片

自動產生的描述

1. Multilayer network

* get\_three\_layer\_network\_params :

採用三層的神經網路對50筆資料訓練。這次可以調整的參數為weight\_scale和learning\_rate。先將weight\_scale設成1，learning rate設成1e-2。訓練結果如下

一張含有 文字, 功能表, 螢幕擷取畫面, 圖書 的圖片

自動產生的描述

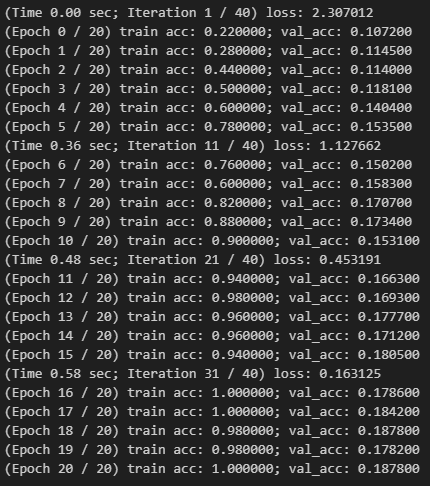
一張含有 文字, 螢幕擷取畫面, 陳列, 數字 的圖片

自動產生的描述

可以看出在20epochs內，training accuracy就可以來到100%，overfitting很嚴重。

* get\_five\_layer\_network\_params :

採用五層的神經網路對50筆資料訓練。這次可以調整的參數為weight\_scale和learning\_rate。先將weight\_scale設成1，learning rate設成0.1。訓練結果如下



一張含有 文字, 螢幕擷取畫面, 圖表, 繪圖 的圖片

自動產生的描述

可以看出在20epochs內，training accuracy就可以來到100%，overfitting很嚴重。

1. Update rules

* sgd + momentum :

根據Nesterov Momentum的公式，如下圖，來完成此函數。

一張含有 文字, 字型, 行, 螢幕擷取畫面 的圖片

自動產生的描述

* 測試結果 :



* 比較sgd與sgd + momentum :

分別透過sgd與sgd + momentum來訓練一個六層的神經網路。執行結果如下 :

一張含有 文字, 螢幕擷取畫面, 字型, 黑與白 的圖片

自動產生的描述

一張含有 文字, 行, 螢幕擷取畫面, 圖表 的圖片

自動產生的描述

由結果可以很明顯地看到，sgd + momentum 的收斂速度快很多。

1. 額外嘗試

* 重新訓練三層、五層的神經網路 :

先前只是利用50張資料來查看模型是否可以overfit，因此在確認能夠overfit後，我嘗試調整參數並用40000張訓練資料來訓練模型。

1. 重新訓練三層神經網路 :

我主要針對learning rate、weight scale和regularization strength去做調整。我採用for loop來對多種參數組合作迭代。訓練完成後，最佳的結果如下

一張含有 文字, 樣式, 螢幕擷取畫面, 布 的圖片

自動產生的描述

一張含有 樣式, 布, 單色, 黑與白 的圖片

自動產生的描述

一張含有 服裝, 布, 樣式, 灰色 的圖片

自動產生的描述

一張含有 服裝, 布, 樣式, 灰色 的圖片

自動產生的描述

神經網路架構為第一層hidden layer size = 512，第二層hidden layer size = 1024。

由上面的結果可以看出，對於三層的神經網路來說，在learning rate = 1、weight scale = 0.01和regularization strength = 0時，會有最好的validation accuracy為56.4 %。

1. 重新訓練五層神經網路 :

這次我主要針對learning rate、weight scale、regularization strength和dropout做調整。最好的訓練結果如下

一張含有 文字, 樣式, 布, 螢幕擷取畫面 的圖片

自動產生的描述

一張含有 服裝, 布, 樣式, 灰色 的圖片

自動產生的描述

一張含有 服裝, 布, 樣式, 灰色 的圖片

自動產生的描述

一張含有 文字, 螢幕擷取畫面, 黑與白, 單色 的圖片

自動產生的描述

網路架構為第一層hidden layer size = 128，第二層hidden layer size = 256、第三層hidden layer size = 512、第四層hidden layer size = 1024。

由上面的圖片可以看出，最好的結果發生在learning rate = 1.1、weight scale = 0.01、regularization strength = 0和dropout = 0.1。最好的準確度為53%。

由重新練模型結果可以發現，對於三層網路與五層網路來說，最好的準確度大約落在53~56%，與之前作業的兩層網路相比，其實相差不遠，代表說對於CIFAR-10來說，只增加1、2層並不會造成甚麼進步。比較讓人意外的是，拿三層網路與五層網路相比，五層網路表現比較不好，這不太合理，因此接下來討論一下原因 :

1. Overfitting : 加大模型後確實會更加容易造成overfitting，但是我認為在這次作業中，造成準確度下降的原因不完全是因為overfitting，因為訓練的時候，我有加上regularization和dropout，在這兩個參數比較大的時候，模型並不會有表現上提升。
2. 資料分布 : 資料分布造成訓練上的困難確實會造成模型表現不如預期。因此我嘗試檢查資料預處理過後的樣子，結果如下

一張含有 文字, 字型, 螢幕擷取畫面 的圖片

自動產生的描述

由上面顯示X\_train的結果可以看到，資料已經有經過normalization的預處理，因此資料分布不均應該也不會是問題。

1. 訓練前沒有shuffle : 想像一下，訓練模型時，假如一個epoch有30個資料，前十個資料都是0，接著十個資料都是1，最後十個資料都是2。這會造成模型在一開始被訓練成只會回答0的懶惰模型，接著雖然會去學習1特徵，但是沒回答幾題後發現答案都是1，又變回懶惰模型……。這就會造成模型學習不起來。為了確認是否有這樣的問題，我將前面以個label顯示出來，結果如下

一張含有 文字, 字型, 螢幕擷取畫面 的圖片

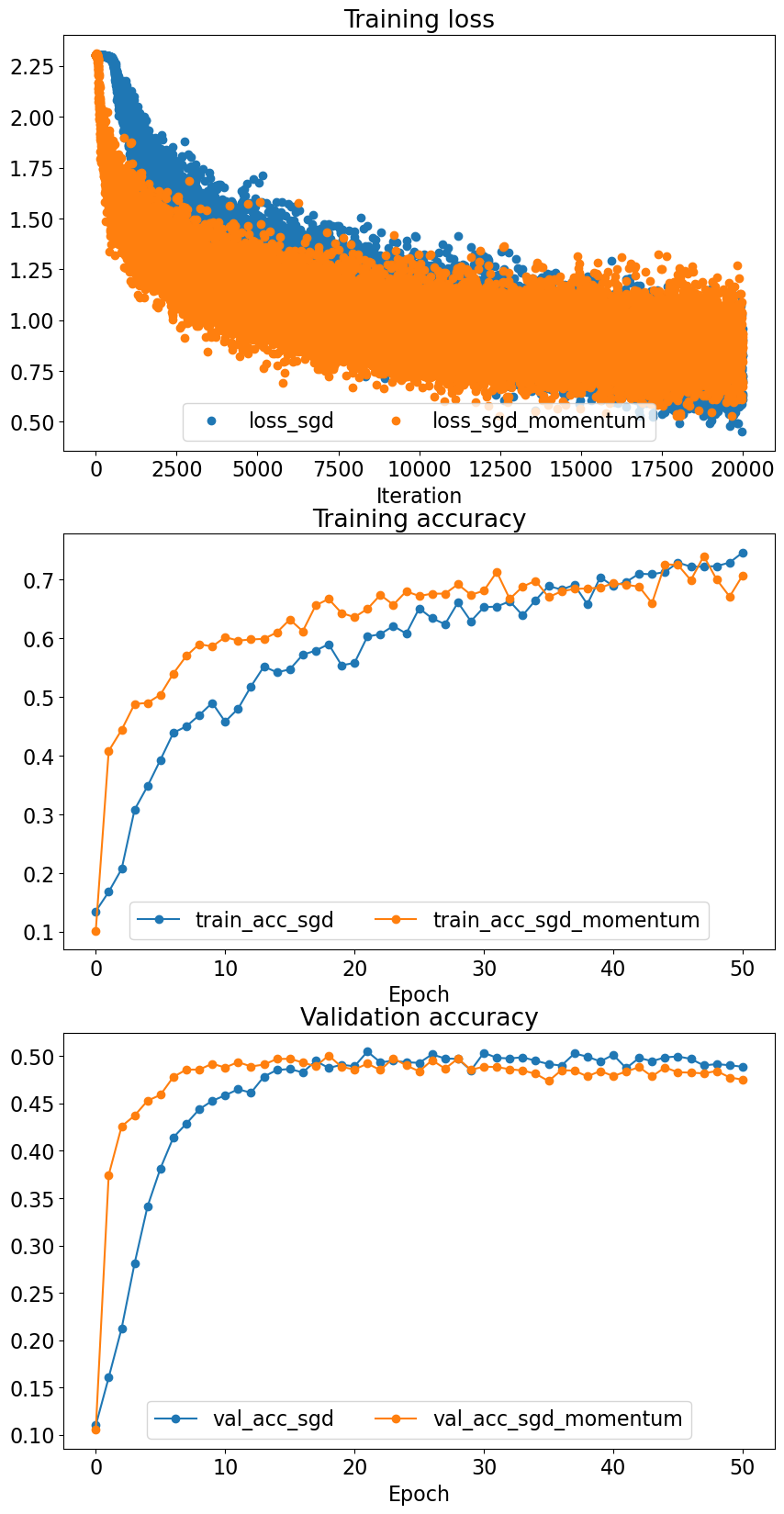
自動產生的描述

由結果可以看出，資料是有經過shuffle的。

經過一系列的檢查，我沒有發現到問題所在，因此我認為會造成這樣的現象是因為單純增加一兩層的hidden layer對於準確度提升並沒有太大幫助，當訓練超參數稍微有一些偏差，就很可能造成準確度不如預期。

* 嘗試使用SGD與SGD + Momentum來訓練六層神經網路

嘗試修改learning rate、weight scale、regularization strength之後訓練模型，發現到模型的準確度基本上都會卡在50%上下。訓練節果如下



由視覺化後結果可以很明顯地看到，準確度會卡在50 %左右。結合前面重新訓練三層、五層神經網路，以及之前作業的兩層網路的結果會發現到，不論是哪一種架構，準確度大約都在50%左右，代表說單純是增加幾層網路其實對於準確度提升很有限。如果需進一步提升準確度，可能需要改變神經網路架構，像是採用CNN或是ResNet等架構。

* Optimizer :

由於這次作業有做到sgd + momentum的實現，因此我希望可以多加了解一下optimizer的原理與種類。

1. Gradient Descent : gradient descent是一種基於凸函數的優化演算法。他透過沿著負gradient方向來降低loss。

一張含有 文字, 字型, 筆跡, 白色 的圖片

自動產生的描述

* 優點 : 很好理解，實做起來簡單。
* 缺點 : 由於他一次要計算所有訓練資料的梯度並一次更新所有參數，因此計算速度很慢，而且需要花費很多記憶體空間。

1. Stochastic Gradient Descent (SGD) : 主要概念與gradient descent相同，差別在於他一次只更新一個參數。

一張含有 圓形, 寫生, 字型, 筆跡 的圖片

自動產生的描述

* 優點 : 他可以對大型的資料做計算，且需要比較少的記憶體空間。
* 缺點 : 由於SGD更新參數時只有使用一個樣本，因此可能有noise，使模型更新不穩定。此外，計算成本也比較高。

1. Mini-Batch Gradient Descent : 將訓練資料分成batches並對這些batches做參數更新。

一張含有 圓形, 行, 圖表, 設計 的圖片

自動產生的描述

* 優點 : 收斂過程會更佳穩定，計算複雜度也比較低。
* 缺點 : 容易受到learning rate的影響。

1. SGD with Momentum : 在SGD的基礎上加上momentum。Momentum使參數在更新一定程度的保留前一次更新時的方向，並利用當前的梯度對該方向做調整。

一張含有 文字, 字型, 筆跡, 白色 的圖片

自動產生的描述

一張含有 圓形, 寫生, 設計 的圖片

自動產生的描述

* 優點 : momentum可以幫助減少noise，而且exponential weighted average可以讓曲線更平滑。
* 缺點 : 需要額外的超參數。

1. AdaGrad (Adaptive Gradient Descent ) : 先前討論的所有方法中，learning rate都是固定的。AdaGrad使我們可以對不同的神經元、迭代次數使用不一樣的learning rate。

一張含有 文字, 字型, 筆跡, 行 的圖片

自動產生的描述

* 優點 : learning rate可以根據迭代次數做調整。對於稀疏的資料也可以有不錯的表現。
* 缺點 : 對於深層的神經網路來說，learning rate會變得很小，導致一些神經元權重更新很慢，也就是所謂的dead neuron problem。

1. RMS-Prop (Root Mean Square Propagaion) : 為一種特殊版本的AdaGrad。Learning rate為梯度的exponential average。通常會結合momentum一起使用。

一張含有 文字, 字型, 筆跡, 白色 的圖片

自動產生的描述

* 優點 : learning rate可以自動更新，而且對於每一個參數都可以有不一樣的learning rate。
* 缺點 : 訓練比較慢。

1. Adam (Adaptive Moment Estimation) : 為目前最流行的optimizer。他是一種可以為每一個參數都計算adaptive learning rate的方法。他結合了momentum和RMS-Prop的使用。

一張含有 字型, 行, 圖表, 筆跡 的圖片

自動產生的描述

* 優點 : 計算效率高、需要比較少的記憶體空間。

1. 如何選擇optimizer :

宗傑以上幾種optimizer的特性，可以歸納出幾個選擇標準。

* 對於稀疏的資料，採用adaptive learning rate的optimizer。
* 在很多情況下，Adam、RMSprop有差不多的表現。
* Adam通常會比RMSprop表現更好一點，因為Adam可以透過momentum來保持更新的方向的一致性。
* 對於較大型的模型，建議使用Adam，因為Adam具有更好的收斂性能。

1. 心得總結

這次作業嘗試使用多層的神經網路來對CIFAR-10資料集做訓練。透過convenience layer和封裝的概念，將需要的東西裝成一包來幫助我們更好的管理code以及網路架構。此外也實現了SGD + momentum optimizer來幫助我們訓練模型。

接下來嘗試從新訓練作業中的幾個神經網路。經過一番嘗試之後發現到僅僅增加幾層神經網路對於準確度提升並沒有太大幫助，如果希望有更大的提升，可能要嘗試更換網路架構，像是採用ResNet、CNN等等之類的架構。

最後我自己到網路上搜尋與整理有關optimizer的資料，並對多種optimizer的原理與優缺做比較。基本上RMSprop與Adam都有很不錯的表現，而Adam通常會表現得比RMSprop還要好，因此很多情況下Adam會是一個default的選擇。

1. Reference

[1] Musstafa “Optimizers in Deep Learning” <https://musstafa0804.medium.com/optimizers-in-deep-learning-7bf81fed78a0>

[2] LS\_learner “訓練及準確度很高，驗證集準確率低的問題” <https://blog.csdn.net/qq_39777550/article/details/108965486>

[3] ML Glossary “Optimizers” <https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html>

[4] OpenAI. (2023). ChatGPT (Mar 14 version) [Large language model]. <https://chat.openai.com/>