HW7 Fully-Connected Neural Networks

M113040105 劉東霖

壹.Linear_Relu_Layer:

一.Linear class:

1.forward:

如下圖所示。首先，因為 x 的 shape 為(N, d_1,...,d_k)，所以要先 reshape 成(N,D)，D=(d_1*d_2*…*d_k)，才能矩陣相乘。最後再將 x 和 w 矩陣相乘和加上 bias 即為輸出。

```
buff_x=x.reshape(x.shape[0],-1)
out = buff_x @ w + b
```

2.backward:

如下面左邊的圖所示，x 的 backward 依圖上的方法更新，要注意的是，dx 的輸出 shape 為(N, d_1,...,d_k)，所以要輸出成 x 的 shape。下右圖為程式碼。

$$s = x@w+b \quad , x \in \mathbb{R}^{N\times D}, w \in \mathbb{R}^{D\times M}$$

downstream_gradient

= local_gradient * upstream_gradient

$$\Rightarrow \frac{\partial f}{\partial x} = \frac{\partial f}{\partial s}\frac{\partial s}{\partial x} = dout @ w.t()$$

```
dx = dout @ w.t()
dx = dx.reshape(x.shape)
```

如下面左邊的圖所示，w 的 backward 依圖上的方法更新，要注意的是，shape 為(N, d_1,...,d_k)，所以要先 reshape 成(N,D)，才能矩陣相乘。下右圖為程式碼。

$s = x@w + b \quad , x \in \mathbb{R}^{N \times D}, w \in \mathbb{R}^{D \times M}$

downstream_gradient

= local_gradient * upstream_gradient

$\Rightarrow \frac{\partial f}{\partial w} = \frac{\partial s}{\partial w} \frac{\partial f}{\partial s} = x.t()@dout$

```
x  =  x.reshape(x.shape[0],-1)
dw  =  x.t()  @  dout
```

如下面左邊的圖所示，b 的 backward 依圖上的方法更新。下右圖為程式碼。

$s = x@w + b \quad , x \in \mathbb{R}^{N \times D}, w \in \mathbb{R}^{D \times M}$

downstream_gradient

= local_gradient * upstream_gradient

$\Rightarrow \frac{\partial f}{\partial x} = \frac{\partial f}{\partial s} \frac{\partial s}{\partial x} = dout@w.t()$

```
db  =  torch.sum(dout,axis=0)
```

二.Relu class:

1.forward:

把 x 大於 0 的輸出。下圖為程式碼。

```
out=torch.maximum(torch.zeros_like(x),x)
```

2.backward:

relu 的 backward 為如果 x>0，dout 就 pass 過去。下圖為程式碼。

```
mask=x>0
dx=mask*dout
```

三.執行結果:

1.如下圖所示，linear class 的 forward 輸出結果跟現實差異不大。

```
x = torch.linspace(-0.1, 0.5, steps=input_size, dtype=torch.float64, device='cuda')
w = torch.linspace(-0.2, 0.3, steps=weight_size, dtype=torch.float64, device='cuda')
b = torch.linspace(-0.3, 0.1, steps=output_dim, dtype=torch.float64, device='cuda')
x = x.reshape(num_inputs, *input_shape)
w = w.reshape(torch.prod(input_shape), output_dim)

out, _ = Linear.forward(x, w, b)
correct_out = torch.tensor([[1.49834984, 1.70660150, 1.91485316],
                            [3.25553226, 3.51413301, 3.77273372]
                            ).double().cuda()

print('Testing Linear.forward function:')
print('difference: ', usefuns.grad.rel_error(out, correct_out))
```

```
Testing Linear.forward function:
difference:  3.683042917976506e-08
```

2.如下圖所示，linear class 的 backward 輸出結果跟現實差異不大。

```
x = torch.randn(10, 2, 3, dtype=torch.float64, device='cuda')
w = torch.randn(6, 5, dtype=torch.float64, device='cuda')
b = torch.randn(5, dtype=torch.float64, device='cuda')
dout = torch.randn(10, 5, dtype=torch.float64, device='cuda')

dx_num = usefuns.grad.compute_numeric_gradient(lambda x: Linear.
dw_num = usefuns.grad.compute_numeric_gradient(lambda w: Linear.
db_num = usefuns.grad.compute_numeric_gradient(lambda b: Linear.

_, cache = Linear.forward(x, w, b)
dx, dw, db = Linear.backward(dout, cache)

# The error should be around e-10 or less
print('Testing Linear.backward function:')
print('dx error: ', usefuns.grad.rel_error(dx_num, dx))
print('dw error: ', usefuns.grad.rel_error(dw_num, dw))
print('db error: ', usefuns.grad.rel_error(db_num, db))
```

```
Testing Linear.backward function:
dx error:  5.221943563709987e-10
dw error:  3.498388787266994e-10
db error:  5.373171200544344e-10
```

3.如下圖所示，relu class 的 forward 輸出結果跟現實差異不大。

```
out, _ = ReLU.forward(x)
correct_out = torch.tensor([[ 0.,



# Compare your output with ours. Th
print('Testing ReLU.forward function:')
print('difference: ', usefuns.grad.rel_

Testing ReLU.forward function:
difference:  4.5454545613554664e-09
```

4.如下圖所示，relu class 的 backward 輸出結果跟現實差異不大。

```
dx_num = usefuns.grad.compute_numer

_, cache = ReLU.forward(x)
dx = ReLU.backward(dout, cache)

# The error should be on the o
print('Testing ReLU.backward functi
print('dx error: ', usefuns.grad.r

Testing ReLU.backward function:
dx error:  2.6317796097761553e-10
```

5.如下圖所示，把 linear class 和 relu class 串再一起使用，並計算

backward，輸出結果跟現實差異不大。

```
Testing Linear_ReLU.forward and Linear_ReLU.backward:
dx error:  1.210759699545244e-09
dw error:  7.462948482161807e-10
db error:  8.915028842081707e-10
```

6.如下圖所示，測試 a3_helper 裡面的 svm_loss 和 softmax_loss，與現

實差異不大。

```
Testing svm_loss:
loss:  9.000430792478463
dx error:  7.97306008441663e-09

Testing softmax_loss:
loss:  2.3026286102347924
dx error:  1.0417990899757076e-07
```

貳.Two layer network:

一.使用到的 function:

1.__init__:

在 weight 初始化時，題目要求以高斯分布的方式隨機生成亂數，且平均值為 0，標準差為 weight scale，如下圖所示。

```
'W1':torch.normal(mean=0,  std=weight_scale,  size=(input_dim,hidden_dim),device=device,dtype=dtype),
'W2':torch.normal(mean=0,  std=weight_scale,  size=(hidden_dim,num_classes),device=device,dtype=dtype),
```

在 bias 初始化時，題目要求初始值為 0，如下圖所示。

```
'b1':torch.zeros((hidden_dim,),device=device,dtype=dtype),
'b2':torch.zeros((num_classes,),device=device,dtype=dtype)}
```

完整程式碼如下:

```
self.params={
    'W1':torch.normal(mean=0,  std=weight_scale,  size=(input_dim,hidden_dim),device=device,dtype=dtype),
    'W2':torch.normal(mean=0,  std=weight_scale,  size=(hidden_dim,num_classes),device=device,dtype=dtype),
    'b1':torch.zeros((hidden_dim,),device=device,dtype=dtype),
    'b2':torch.zeros((num_classes,),device=device,dtype=dtype)}
```

2.loss:

首先把 w1,w2,b1,b2 拿出來，再用 Linear_Relu 算出第一層輸出，並把中間的變數用 cache 存起來，方便後面 backward。最後用 Linear 算出第二層輸出，並把中間的變數用 cache 存起來，方便後面 backward。程式碼如下:

```
w1,w2,b1,b2=self.params['W1'],self.params['W2'],self.params['b1'],self.params['b2']
caches=[]
out,cache=Linear_ReLU.forward(X,w1,b1)
caches.append(cache)
scores,cache  =  Linear.forward(out,w2,b2)
caches.append(cache)
```

再來下面的更新 loss 和 grad。首先先用 a3_helper 算出 loss 和 dscore，再將 loss 加上權重完成正則化，程式碼如下。

```
loss,dscore=softmax_loss(scores,y)
loss+=self.reg*(torch.sum(w1  *  w1)+torch.sum(w2  *  w2))
```

再來是反向傳播的部分。首先用 Linear.backward()取出最後一層的梯度和更新最後一層的權重和偏差，cache.pop()為傳遞 forward 時的中間數據。最後用 Linear_Relu.backward()取出第一層的梯度和更新第一層的權重和偏差，程式碼如下。

```
dout,dw,db=Linear.backward(dscore,caches.pop())
grads['W2']=dw+self.reg*w2*2
grads['b2']=db
dout,dw,db=Linear_ReLU.backward(dout,  caches.pop())
grads['W1']=dw+self.reg*w1*2
grads['b1']=db
```

全部程式碼如下:

```
w1,w2,b1,b2=self.params['W1'],self.params['W2'],self.params['b1'],self.params['b2']
caches=[]
out,cache=Linear_ReLU.forward(X,w1,b1)
caches.append(cache)
scores,cache  =  Linear.forward(out,w2,b2)
caches.append(cache)
```

```
loss,dscore=softmax_loss(scores,y)
loss+=self.reg*(torch.sum(w1  *  w1)+torch.sum(w2  *  w2))
dout,dw,db=Linear.backward(dscore,caches.pop())
grads['W2']=dw+self.reg*w2*2
grads['b2']=db
dout,dw,db=Linear_ReLU.backward(dout,  caches.pop())
grads['W1']=dw+self.reg*w1*2
grads['b1']=db
```

3.create solver instance:

首先，先建立一個 TwoLayerNet 的 model，再用 Solver instance 去 train model 來讓 validation set 的準確率達到 50%以上。下圖為程式碼:

```
model = TwoLayerNet(hidden_dim=100, dtype=dtype, device=device)
################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least    #
# 50% accuracy on the validation set.
################################################################################
solver = None
# Replace "pass" statement with your code
solver = Solver(model, data_dict,
                update_rule=sgd,
                optim_config={
                  'learning_rate': 0.1,
                },
                lr_decay=0.95,
                num_epochs=10, batch_size=100,
                print_every=100,
                device=device)
solver.train()
```

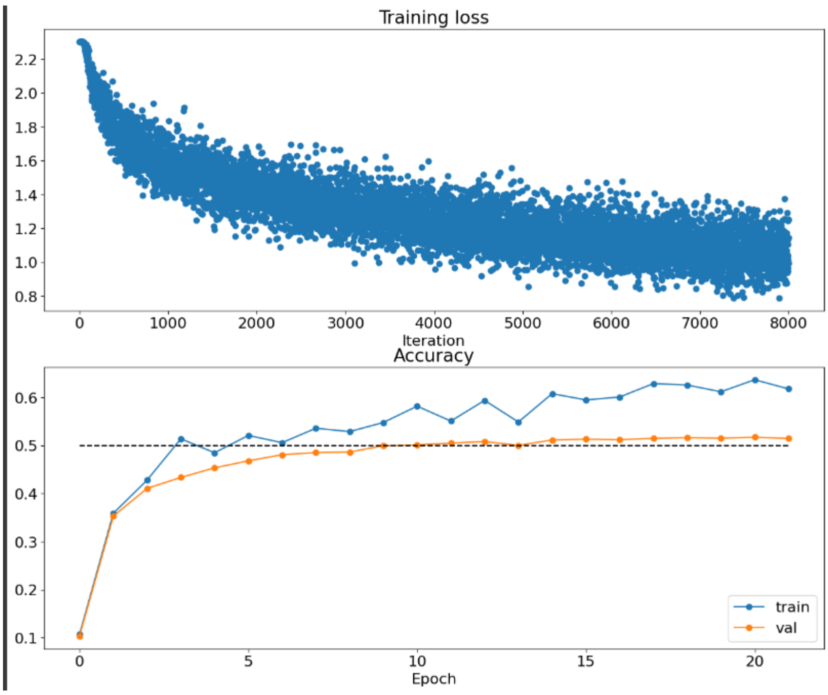
二. 執行結果:

1. 如下圖所示，創建一個 TwoLayerNet 的實例 model，並對其進行初始化，其中包括設定 input size、hidden size、class number、權重標準差、data type 和 device，並檢查權重和偏差是否符合要求，發現數值誤差都很小。

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 2.57e-07
W2 relative error: 1.65e-09
b1 relative error: 1.01e-06
b2 relative error: 2.41e-09
Running numeric gradient check with reg =  0.7
W1 relative error: 2.70e-08
W2 relative error: 9.86e-09
b1 relative error: 2.28e-06
b2 relative error: 2.90e-08
```

2.如下圖所示，利用 solver 來 train Twolayernet，validation 準確率有到 50%，並把 loss 和準確率 plot 出來和把最棒的準確率儲存。

```
(Epoch 15 / 10) train acc: 0.601000; val_acc: 0.512200
(Time 8.20 sec; Iteration 2001 / 4000) loss: 1.016012
(Time 8.57 sec; Iteration 2101 / 4000) loss: 1.249219
(Time 8.92 sec; Iteration 2201 / 4000) loss: 0.984308
(Time 9.28 sec; Iteration 2301 / 4000) loss: 1.157969
(Epoch 16 / 10) train acc: 0.629000; val_acc: 0.514900
(Time 9.70 sec; Iteration 2401 / 4000) loss: 1.227245
(Time 10.05 sec; Iteration 2501 / 4000) loss: 0.867307
(Time 10.41 sec; Iteration 2601 / 4000) loss: 0.999737
(Time 10.76 sec; Iteration 2701 / 4000) loss: 1.122970
(Epoch 17 / 10) train acc: 0.626000; val_acc: 0.516400
(Time 11.19 sec; Iteration 2801 / 4000) loss: 1.060964
(Time 11.55 sec; Iteration 2901 / 4000) loss: 1.037791
(Time 11.91 sec; Iteration 3001 / 4000) loss: 1.062729
(Time 12.27 sec; Iteration 3101 / 4000) loss: 1.123431
(Epoch 18 / 10) train acc: 0.612000; val_acc: 0.515300
(Time 12.70 sec; Iteration 3201 / 4000) loss: 1.032545
(Time 13.06 sec; Iteration 3301 / 4000) loss: 1.175342
(Time 13.42 sec; Iteration 3401 / 4000) loss: 1.065494
(Time 13.79 sec; Iteration 3501 / 4000) loss: 0.992284
(Epoch 19 / 10) train acc: 0.637000; val_acc: 0.517400
(Time 14.22 sec; Iteration 3601 / 4000) loss: 1.274796
(Time 14.58 sec; Iteration 3701 / 4000) loss: 1.064057
(Time 14.94 sec; Iteration 3801 / 4000) loss: 0.916855
(Time 15.29 sec; Iteration 3901 / 4000) loss: 1.150464
(Epoch 20 / 10) train acc: 0.618000; val_acc: 0.514700
```




```
(Epoch 10 / 10) train acc: 0.582000; val_acc: 0.501300
load checkpoint file: drive/My Drive/Colab Notebooks/DL2023/HW7/best_two_layer_net.pth
Saved model's accuracy on validation is 0.5173999667167664
```

參.Multilayer network:

一.使用到的 function:

1.FullyConnectedNet __init__:

首先，先使用串接的方式把所有 layer 的神經元數目用 list 串起來，如下圖所示。

```
dims = [input_dim] + hidden_dims + [num_classes]
```

如下圖所示，for 一個迴圈看有幾個 layer，並把 layer 之間的神經元數目和名字都找出來。

```
dims = [input_dim] + hidden_dims + [num_classes]
for i in range(1, self.num_layers + 1):
    W_name = 'W'+repr(i)
    b_name = 'b'+repr(i)
    fan_in = dims[i-1]
    fan_out = dims[i]
```

初始 weight 和 bias 的方式跟前面 Two layer net 一樣，程式碼如下。

```
self.params[W_name] = torch.normal(mean=0, std=weight_scale, size=(fan_in, fan_out),device=device,dtype=dtype)
self.params[b_name] = torch.zeros(size=(fan_out,), device=device, dtype=dtype)
```

2. FullyConnectedNet loss:

首先，先用前面寫的 linear relu forward 的方式算出每一層的輸出結果，並把權重的部分用 cache 存起來，方便後面 backward。最後一層因為不能用到 relu，所以與前面分開處理，程式如下。

```
out=X
caches=[]
for i in range(self.num_layers-1):
    out,cache=Linear_ReLU.forward(out,self.params['W'+repr(i+1)],self.params['b'+repr(i+1)])
    caches.append(cache)
##########################################################################
#                             END OF YOUR CODE
##########################################################################
scores, cache = Linear.forward(out, self.params['W'+repr(self.num_layers)], self.params['b'+repr(self.num_layers)])
caches.append(cache)
```

再來是計算 loss 的部分，這裡我是直接套 a3_helper 裡面的 softmax_loss 來計算 loss，並加入每一層的 weight 來達到正則化。這裡題目有要求，每次加權重時要乘上 0.5。程式碼如下:

```
data_loss,dscore=softmax_loss(scores,y)
reg_loss=0.0
for i in range(self.num_layers):
    w=self.params['W'+repr(i+1)]
    reg_loss+=self.reg*torch.sum(w**2)*0.5
loss=data_loss+reg_loss
```

再來是反向傳播的部分。首先用 Linear.backward()取出最後一層的梯度和更新最後一層的權重和偏差，cache.pop()為傳遞 forward 時的中間數據。最後用 Linear_Relu.backward()取出每一層的梯度和更新每一層的權重和偏差，程式碼如下。

```
dout,dw,db=Linear.backward(dscore,caches.pop())
grads['W'+repr(self.num_layers)]=dw+self.reg*self.params['W'+repr(self.num_layers)]
grads['b'+repr(self.num_layers)]=db
for i in range(self.num_layers-2,-1,-1):
    dout,dw,db=Linear_ReLU.backward(dout, caches.pop())
    grads['W'+repr(i+1)]=dw+self.reg*self.params['W'+repr(i+1)]
    grads['b'+repr(i+1)]=db
```

完整程式碼如下:

```
out=X
caches=[]
for i in range(self.num_layers-1):
    out,cache=Linear_ReLU.forward(out,self.params['W'+repr(i+1)],self.params['b'+repr(i+1)])
    caches.append(cache)
##############################################################################
#                             END OF YOUR CODE
##############################################################################
scores, cache = Linear.forward(out, self.params['W'+repr(self.num_layers)], self.params['b'+repr(self.num_layers)]
caches.append(cache)
```

```
data_loss,dscore=softmax_loss(scores,y)
reg_loss=0.0
for i in range(self.num_layers):
    w=self.params['W'+repr(i+1)]
    reg_loss+=self.reg*torch.sum(w**2)*0.5
loss=data_loss+reg_loss
dout,dw,db=Linear.backward(dscore,caches.pop())
grads['W'+repr(self.num_layers)]=dw+self.reg*self.params['W'+repr(self.num_layers)
grads['b'+repr(self.num_layers)]=db
for i in range(self.num_layers-2,-1,-1):
    dout,dw,db=Linear_ReLU.backward(dout, caches.pop())
    grads['W'+repr(i+1)]=dw+self.reg*self.params['W'+repr(i+1)]
    grads['b'+repr(i+1)]=db
```

3. get three layer network params 和 get five layer network params:

接下來我用下面這些參數來讓我的 train 準確率達到 100%。

get_three_layer_network_params:

```
weight_scale = 1e-1
learning_rate = 1e-1
```

get_five_layer_network_params:

```
learning_rate  =  1.01e-1
weight_scale  =    1e-1
```

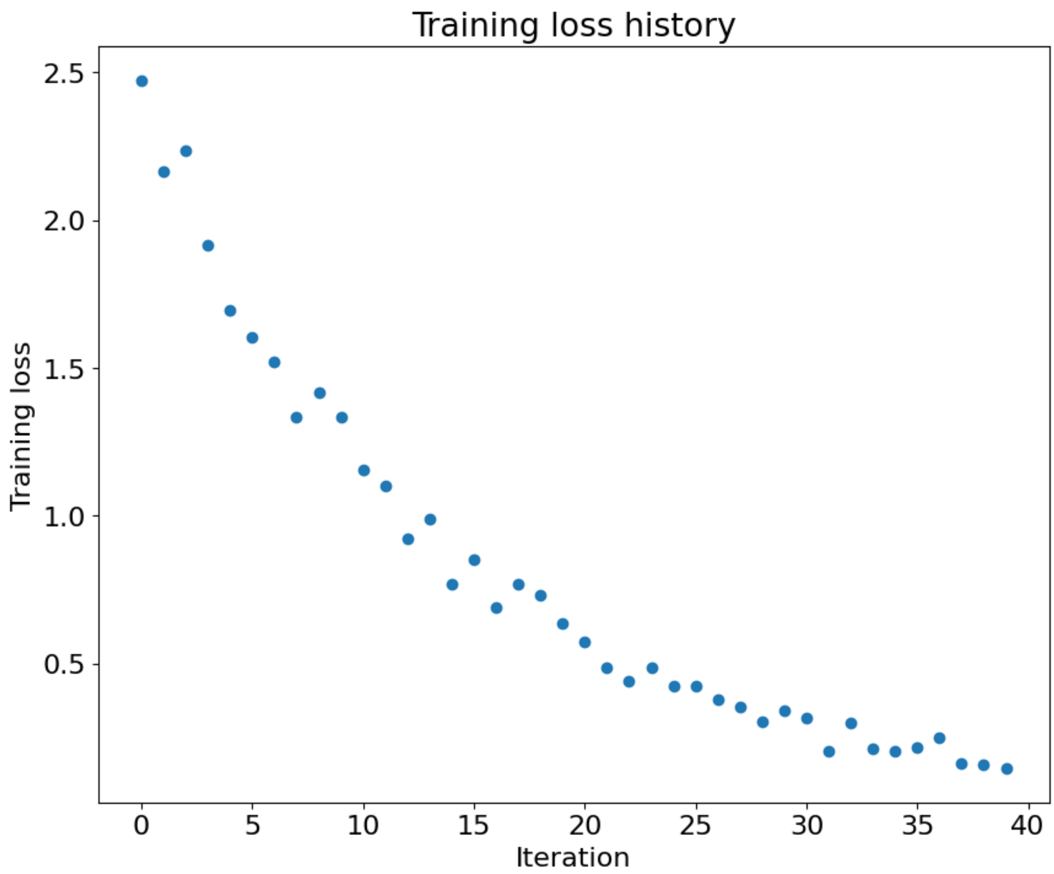
二.執行結果:

1.如下圖所示，創建一個 FullyConnectedNet 的實例 model，並對其進行初始化，其中包括設定 input size、hidden size、class number、權重標準差、data type 和 device，並檢查權重和偏差是否符合要求，發現數誤差都很小。

```
Running check with reg =  0
Initial loss:  2.3053575717037686
W1 relative error: 6.06e-08
W2 relative error: 1.02e-07
W3 relative error: 5.89e-08
b1 relative error: 1.28e-07
b2 relative error: 2.05e-08
b3 relative error: 3.41e-09
Running check with reg =  3.14
Initial loss:  7.29369633719099
W1 relative error: 6.79e-09
W2 relative error: 8.39e-09
W3 relative error: 9.56e-09
b1 relative error: 1.57e-07
b2 relative error: 2.91e-08
b3 relative error: 6.23e-09
```

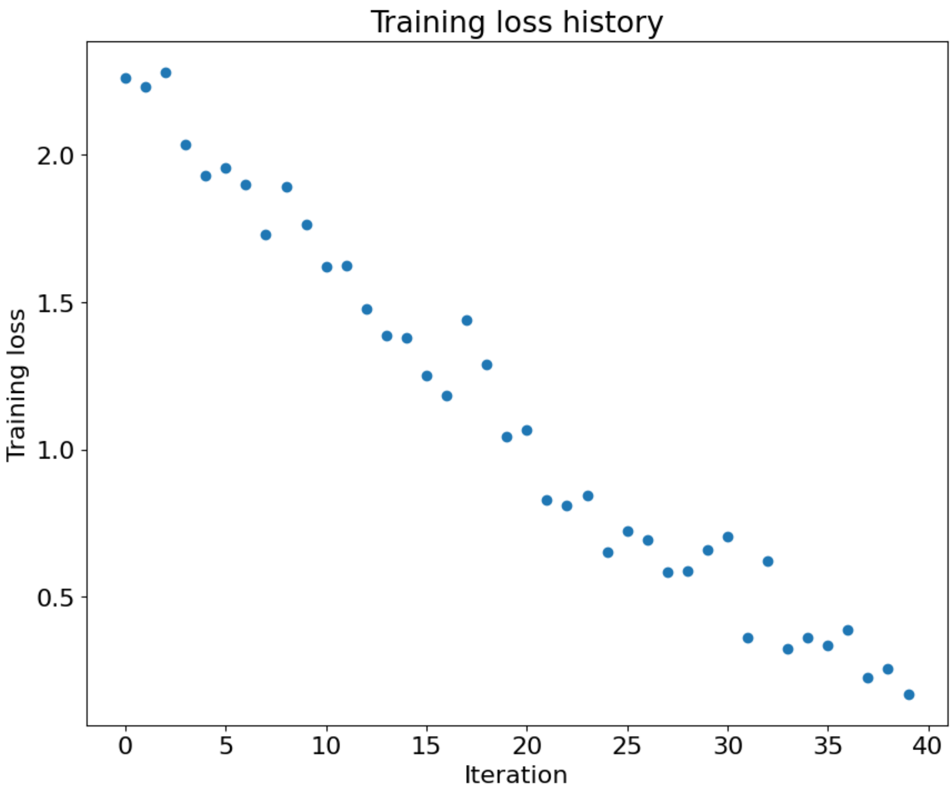
2.如下圖所示，使用三層神經網絡，通過調整學習率和 weight scale，對 50 個訓練示例進行過度擬合，最高可到 100%。

```
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.181900
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.179900
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.181100
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.183700
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.180300
(Time 0.36 sec; Iteration 31 / 40) loss: 0.318150
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.180100
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.183900
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.185100
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.186500
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.185300
```



3. 如下圖所示，使用五層神經網絡，通過調整學習率和 weight scale，對 50 個訓練示例進行過度擬合，最高可到 100%，並把最佳解存起來。

```
(Epoch 11 / 20) train acc: 0.800000; val_acc: 0.175600
(Epoch 12 / 20) train acc: 0.920000; val_acc: 0.176500
(Epoch 13 / 20) train acc: 0.920000; val_acc: 0.177300
(Epoch 14 / 20) train acc: 0.920000; val_acc: 0.172600
(Epoch 15 / 20) train acc: 0.840000; val_acc: 0.173700
(Time 0.55 sec; Iteration 31 / 40) loss: 0.703088
(Epoch 16 / 20) train acc: 0.940000; val_acc: 0.172900
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.185200
(Epoch 18 / 20) train acc: 0.940000; val_acc: 0.193100
(Epoch 19 / 20) train acc: 0.960000; val_acc: 0.185100
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.190600
```




```
Saved in drive/My Drive/Colab Notebooks/DL2023/HW7/best_overfit_five_layer_net.pth
load checkpoint file: drive/My Drive/Colab Notebooks/DL2023/HW7/best_overfit_five_layer_net.pth
Saved model's accuracy on small train is 1.0
```

肆.Update rules:

一.使用到的 function:

1. sgd momentum

更新的公式如下圖所示:

SGD+Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

程式碼如下。

```
momentum=config['momentum']
learning_rate=config['learning_rate']
v=momentum*v-learning_rate*dw
next_w=w+v
```

二.執行結果:

1. 如下圖所示，用 sgd momentim 算出來的 w 和 v 跟實際上沒差很多

```
next_w error:  1.6802078709310813e-09
velocity error:  2.9254212825785614e-09
```

2. 如下圖所示，比較 loss 和準確率發現 sgd momentum 比 sgd 好。

```
running with  sgd
(Time 0.00 sec; Iteration 1 / 200) loss: 2.302603
(Epoch 0 / 5) train acc: 0.099000; val_acc: 0.097100
(Epoch 1 / 5) train acc: 0.103000; val_acc: 0.095200
(Epoch 2 / 5) train acc: 0.099000; val_acc: 0.095200
(Epoch 3 / 5) train acc: 0.096000; val_acc: 0.096300
(Epoch 4 / 5) train acc: 0.126000; val_acc: 0.106100
(Epoch 5 / 5) train acc: 0.102000; val_acc: 0.095200

running with  sgd_momentum
(Time 0.00 sec; Iteration 1 / 200) loss: 2.303541
(Epoch 0 / 5) train acc: 0.082000; val_acc: 0.092400
(Epoch 1 / 5) train acc: 0.105000; val_acc: 0.095200
(Epoch 2 / 5) train acc: 0.150000; val_acc: 0.145600
(Epoch 3 / 5) train acc: 0.181000; val_acc: 0.186100
(Epoch 4 / 5) train acc: 0.250000; val_acc: 0.225300
(Epoch 5 / 5) train acc: 0.295000; val_acc: 0.245700
```