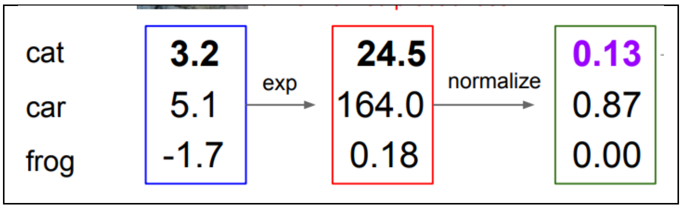
HW4 Softmax classifier

壹.Softmax 介紹:

1.softmax classifier 的定義如下圖，其中 Zj=WjXi，也等於 scores。

$$s_j(z) = \frac{e^{z_j}}{\sum_j e^{z_j}}$$

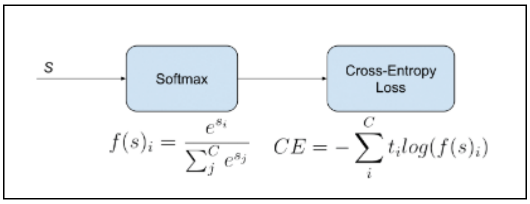
其計算過程如下圖所示，先把 scores 先把它指數化，為了讓 scores 都大於 0。最後把 scores normalize 變成機率。



2.softmax loss 又稱 cross entropy loss，把正確類別的機率代入下圖公式。如果正確類別的機率越大，Li 越小，否則 Li 越大。

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

串起來的流程圖如下:



3.softmax loss 的梯度推導過程如下:

$$\nabla_w L = \frac{1}{N}\sum_{i}^{N} \nabla_w L_i + 2\lambda W$$

$$L_i = -\log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right) = -s_i + \log\left(\sum_j e^{s_j}\right), \quad s = Wx$$

if $i \neq j$

$$\frac{\partial L_i}{\partial w_j} = \frac{\partial L_i}{\partial s_j}\frac{\partial s_j}{\partial w_j} = \frac{e^{s_j}}{\sum_j e^{s_j}} x_i$$

① $\frac{\partial L_i}{\partial s_j} = \frac{e^{s_j}}{\sum_j e^{s_j}}$

② $\frac{\partial s_j}{\partial w_j} = x_i$

if $i == j$

$$\frac{\partial L_i}{\partial w_i} = \frac{\partial L_i}{\partial s_i}\frac{\partial s_i}{\partial w_i} = \left(\frac{e^{s_i}}{\sum_j e^{s_j}} - 1\right) x_i$$

① $\frac{\partial L_i}{\partial s_i} = -1 + \frac{e^{s_i}}{\sum_j e^{s_j}}$

② $\frac{\partial s_i}{\partial w_i} = x_i$

當我們計算出梯度後在用梯度下降法就可以不斷的更新參數了。

貳.Softmax 實作:

1.Softmax_loss_navive:

1.1 目的 : 以雙層 for 的形式來完成 softmax 的 loss function

1.2 過程:

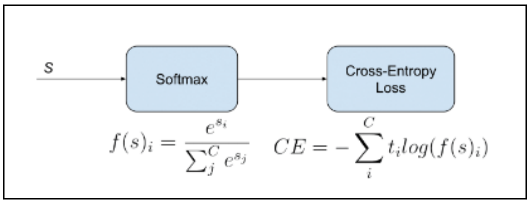
(1). for 一個 loop 為總共幾個 train，並計算 scores，如下圖所示:

```
num_classes = W.shape[1]
num_train = X.shape[0]
for i in range(num_train):
    scores=X[i] @ W
```

(2). 在實現的時候容易發現直接用 Softmax 容易出現梯度爆炸的問題，這是因為浮點數的範圍是有限的。如果指數過大，就直接 overflow 了。為了避免，需要減去最大項，其機率不會變，程式碼如下。

```
scores-=torch.max(scores)
```

(3).依照下圖的流程計算 cross entropy loss。



程式碼如下:

```
exp=torch.exp(scores)
p=exp/torch.sum(exp)
loss+=-torch.log(p[y[i]])
```

(4).再來就是算 dw 的部分，依照下圖的規則來更新。

if $i \neq j$

$$\frac{\partial L_i}{\partial w_j} = \frac{\partial L_i}{\partial s_j}\frac{\partial s_j}{\partial w_j} = \frac{e^{s_j}}{\sum_j e^{s_j}} x_i$$

if $i == j$

$$\frac{\partial L_i}{\partial w_i} = \frac{\partial L_i}{\partial s_i}\frac{\partial s_i}{\partial w_i} = \left(\frac{e^{s_i}}{\sum_j e^{s_j}} - 1\right) x_i$$

程式碼如下:

```
for j in range(num_classes):
    if j==y[i]:
        dW[:,j]+=(p[j]-1)*X[i]
    else:
        dW[:,j]+=p[j]*X[i]
```

(5).根據下圖公式，累加完的 loss 要除以 train 的數量，並加入 regularization term

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} [\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + \Delta)] + \lambda \sum_k \sum_l W_{k,l}^2$$

程式碼如下:

```
loss /= num_train

# Add regularization to the loss.
loss += reg * torch.sum(W * W)
```

dw 也要依據下圖的公式，除以 train 的數量，並加入 regularization 的梯度。

$$\nabla_W L = \frac{1}{N} \sum_{1}^{N} \nabla_W L_i + 2\lambda W$$

程式碼如下:

```
dW/=num_train
dW+=reg*W*2
```

(6).完整程式碼

```
loss = 0.0
dW = torch.zeros_like(W)

########################################
# TODO: Compute the softmax loss
# Store the loss in loss and th
# here, it is easy to run into
# in http://cs231n.github.io/linear-
# regularization!
########################################
# Replace "pass" statement with y
num_classes = W.shape[1]
num_train = X.shape[0]
for i in range(num_train):
    scores=X[i] @ W
    scores-=torch.max(scores)
    exp=torch.exp(scores)
    p=exp/torch.sum(exp)
    loss+=-torch.log(p[y[i]])
    for j in range(num_classes):
        if j==y[i]:
            dW[:,j]+=(p[j]-1)*X[i]
        else:
            dW[:,j]+=p[j]*X[i]
loss/=num_train
dW/=num_train
loss += reg * torch.sum(W * W)
dW+=reg*W*2
```

1.3 執行結果:

(1).沒加 regularization 的 loss:

```
loss, _ = softmax_loss_naive(W, X_batch, y_batch, reg=0.0)

# As a rough sanity check, our loss should be something
print('loss: %f' % loss)
print('sanity check: %f' % (math.log(10.0)))

loss: 2.302826
sanity check: 2.302585
```

(2). λ=0 時，numerical gradient 和 analytic gradient 的誤差皆小於 1e-5，如下圖所示:

```
numerical: 0.003046 analytic: 0.003046, relative error: 6.468497e-07
numerical: 0.006308 analytic: 0.006308, relative error: 1.234992e-07
numerical: 0.005392 analytic: 0.005392, relative error: 2.534459e-07
numerical: 0.002581 analytic: 0.002581, relative error: 3.442300e-08
numerical: 0.007512 analytic: 0.007512, relative error: 8.122736e-07
numerical: 0.006417 analytic: 0.006417, relative error: 2.286038e-08
numerical: 0.011391 analytic: 0.011391, relative error: 2.935470e-07
numerical: 0.001822 analytic: 0.001822, relative error: 2.932218e-06
numerical: -0.014710 analytic: -0.014710, relative error: 8.967622e-08
numerical: -0.005153 analytic: -0.005153, relative error: 4.012889e-07
```

(3). λ=10 時，numerical gradient 和 analytic gradient 的誤差皆小於 1e-5，如下圖所示:

```
numerical: 0.004914 analytic: 0.004914, relative error: 2.907815e-08
numerical: 0.005887 analytic: 0.005887, relative error: 8.060044e-07
numerical: 0.006309 analytic: 0.006309, relative error: 1.166504e-08
numerical: 0.001580 analytic: 0.001580, relative error: 1.087482e-06
numerical: 0.005839 analytic: 0.005839, relative error: 8.457771e-07
numerical: 0.006800 analytic: 0.006800, relative error: 6.637346e-07
numerical: 0.011465 analytic: 0.011465, relative error: 1.624818e-07
numerical: 0.002314 analytic: 0.002314, relative error: 4.575594e-07
numerical: -0.016813 analytic: -0.016813, relative error: 9.876027e-08
numerical: -0.006673 analytic: -0.006673, relative error: 1.797545e-07
```

2. softmax_loss_vectorized:

2.1 目的 : 不用 for 的方式來降低總執行時間。

2.2 過程 :

(1). 利用 X@W 來算出 scores，程式碼如下:

```
num_classes = W.shape[1]
num_train = X.shape[0]
s=X @ W
```

(2). 為了防止數值爆炸，必須減掉 score 裡每一個 row 的最大值，程式如下:

```
s-=torch.max(s,dim=1,keepdim=True).values
```

(3). 利用下圖的方式把每一筆 data 的正確類別找出來，並把 score 指數化。

```
sy=s[range(num_train),y]
s_exp=torch.exp(s)
```

(4).利用下圖公式計算出 loss，並把 loss 除以 train 的個數再加上 regularization term。

$$L = \frac{1}{N}\boxed{\sum_i^N L_i} + \lambda \sum w^2$$

$$\sum_i^N -\log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right) = \sum_i^N \left(-s_i + \log\left(\sum_j e^{s_j}\right)\right) = -\sum_i^N s_i + \sum_i^N \log\left(\sum_j e^{s_j}\right)$$

程式碼如下:

```
loss=-sum(sy)+torch.sum(torch.log(torch.sum(s_exp,dim=1)))
```

```
loss/=num_train
loss+=reg*torch.sum(W * W)
```

(5).依照下圖的公式計算 dw。先算出經過指數化過後的機率 p，再把 p 裡面正確類別的地方等於-1。

最後再將((x 和 p 矩陣相乘)/train 的個數)+regularization 的梯度就是 loss 的梯度了。

$$\text{if } i \neq j \quad \frac{\partial L_i}{\partial w_j} = \frac{\partial L_i}{\partial s_j}\frac{\partial s_j}{\partial w_j} = \frac{e^{s_j}}{\sum_j e^{s_j}} \times x_i \qquad \text{if } i == j \quad \frac{\partial L_i}{\partial w_i} = \frac{\partial L_i}{\partial s_i}\frac{\partial s_i}{\partial w_i} = \left(\frac{e^{s_i}}{\sum_j e^{s_j}} - 1\right) x_i$$

程式碼如下:

```
p=s_exp/torch.sum(s_exp,dim=1).reshape(-1,1)
p[range(num_train),y]-=1
dW=X.T@(p)
dW/=num_train
dW+=reg*W*2
```

(6).全部程式碼如下:

```
num_classes = W.shape[1]
num_train = X.shape[0]
s=X @ W
s-=torch.max(s,dim=1,keepdim=True).values
sy=s[range(num_train),y]
s_exp=torch.exp(s)
loss=-sum(sy)+torch.sum(torch.log(torch.sum(s_exp,dim=1)))
p=s_exp/torch.sum(s_exp,dim=1).reshape(-1,1)
p[range(num_train),y]-=1
dW=X.T@(p)
dW/=num_train
dW+=reg*W*2
loss/=num_train
loss+=reg*torch.sum(W * W)
```

2.3 執行結果:

(1). 有無用向量化操作的區別，發現雖然梯度有些微小的差異，但speedup卻差很多，如下圖所示:

```
naive loss: 2.302841e+00 computed in 110.192537s
vectorized loss: 2.302841e+00 computed in 3.863573s
Loss difference: 0.00e+00
Gradient difference: 6.81e-16
Speedup: 28.52X
```

(2).檢測了同一種參數出來的結果會一樣，不會有數值爆炸的問題出現。

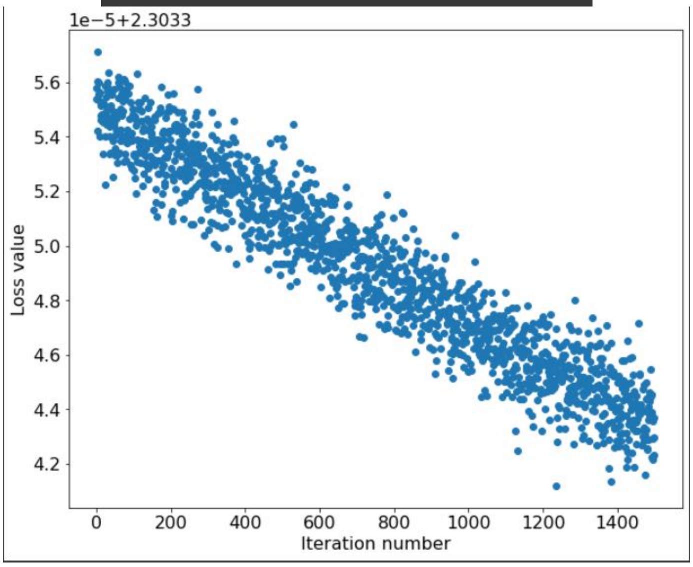
```
W_ones = torch.ones(D, C, device=device, dtype=dtype)
W, loss_hist = train_linear_classifier(softmax_loss_naive, W_ones,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-8, reg=2.5e4,
                                       num_iters=1, verbose=True)

W_ones = torch.ones(D, C, device=device, dtype=dtype)
W, loss_hist = train_linear_classifier(softmax_loss_vectorized, W_ones,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=1e-8, reg=2.5e4,
                                       num_iters=1, verbose=True)
```

```
iteration 0 / 1: loss 768250002.302585
iteration 0 / 1: loss 768250002.302585
```

(3).重複了1500次參數更新後，發現梯度有在緩慢的下降，並把下降過程plot出來。

```
iteration 0 / 1500: loss 2.303356
iteration 100 / 1500: loss 2.303353
iteration 200 / 1500: loss 2.303354
iteration 300 / 1500: loss 2.303352
iteration 400 / 1500: loss 2.303352
iteration 500 / 1500: loss 2.303351
iteration 600 / 1500: loss 2.303350
iteration 700 / 1500: loss 2.303349
iteration 800 / 1500: loss 2.303349
iteration 900 / 1500: loss 2.303348
iteration 1000 / 1500: loss 2.303347
iteration 1100 / 1500: loss 2.303348
iteration 1200 / 1500: loss 2.303347
iteration 1300 / 1500: loss 2.303345
iteration 1400 / 1500: loss 2.303345
That took 5.607013s
```



(4).把 train 和 validation 放入之前在 svm 使用過的 predict linear classifier 預測，發現結果很慘，如下圖所示:

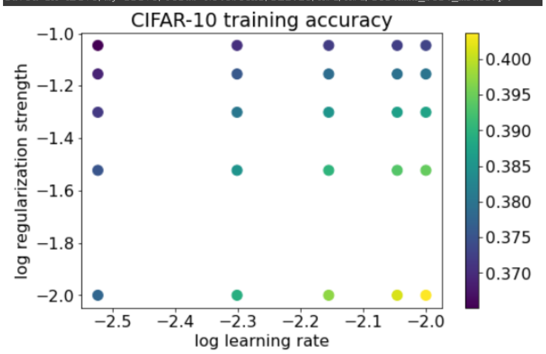
```
training accuracy: 8.90%
validation accuracy: 8.54%
```

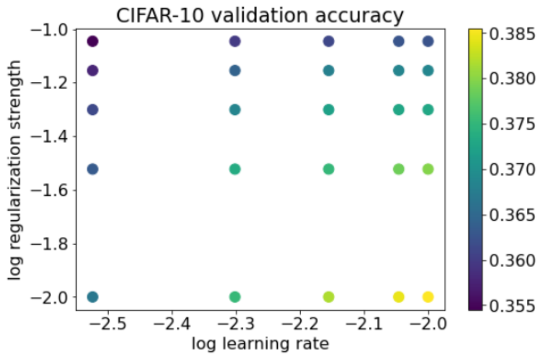
predict linear classifier 的程式碼如下:

```
scores=X @ W
y_pred=torch.argmax(scores,axis=1)
```

(5).利用 softmax_get_search_params 設的 learning_rate 和 regularization_strengths 來找最佳解，並把他 plot 出來，顏色越黃代表準確率越高。softmax_get_search_params 的參數跟之前在 svm_get_search_params 一樣。

```
Training Softmax 1 / 25 with learning_rate=3.000000e-03 and reg=1.000000e-02
Training Softmax 2 / 25 with learning_rate=3.000000e-03 and reg=3.000000e-02
Training Softmax 3 / 25 with learning_rate=3.000000e-03 and reg=5.000000e-02
Training Softmax 4 / 25 with learning_rate=3.000000e-03 and reg=7.000000e-02
Training Softmax 5 / 25 with learning_rate=3.000000e-03 and reg=9.000000e-02
Training Softmax 6 / 25 with learning_rate=5.000000e-03 and reg=1.000000e-02
Training Softmax 7 / 25 with learning_rate=5.000000e-03 and reg=3.000000e-02
Training Softmax 8 / 25 with learning_rate=5.000000e-03 and reg=5.000000e-02
Training Softmax 9 / 25 with learning_rate=5.000000e-03 and reg=7.000000e-02
Training Softmax 10 / 25 with learning_rate=5.000000e-03 and reg=9.000000e-02
Training Softmax 11 / 25 with learning_rate=7.000000e-03 and reg=1.000000e-02
Training Softmax 12 / 25 with learning_rate=7.000000e-03 and reg=3.000000e-02
Training Softmax 13 / 25 with learning_rate=7.000000e-03 and reg=5.000000e-02
Training Softmax 14 / 25 with learning_rate=7.000000e-03 and reg=7.000000e-02
Training Softmax 15 / 25 with learning_rate=7.000000e-03 and reg=9.000000e-02
Training Softmax 16 / 25 with learning_rate=9.000000e-03 and reg=1.000000e-02
Training Softmax 17 / 25 with learning_rate=9.000000e-03 and reg=3.000000e-02
Training Softmax 18 / 25 with learning_rate=9.000000e-03 and reg=5.000000e-02
Training Softmax 19 / 25 with learning_rate=9.000000e-03 and reg=7.000000e-02
Training Softmax 20 / 25 with learning_rate=9.000000e-03 and reg=9.000000e-02
Training Softmax 21 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-02
Training Softmax 22 / 25 with learning_rate=1.000000e-02 and reg=3.000000e-02
Training Softmax 23 / 25 with learning_rate=1.000000e-02 and reg=5.000000e-02
Training Softmax 24 / 25 with learning_rate=1.000000e-02 and reg=7.000000e-02
Training Softmax 25 / 25 with learning_rate=1.000000e-02 and reg=9.000000e-02
lr 3.000000e-03 reg 1.000000e-02 train accuracy: 0.377725 val accuracy: 0.366900
lr 3.000000e-03 reg 3.000000e-02 train accuracy: 0.375275 val accuracy: 0.363300
lr 3.000000e-03 reg 5.000000e-02 train accuracy: 0.371725 val accuracy: 0.361400
lr 3.000000e-03 reg 7.000000e-02 train accuracy: 0.368975 val accuracy: 0.357700
lr 3.000000e-03 reg 9.000000e-02 train accuracy: 0.365125 val accuracy: 0.354500
lr 5.000000e-03 reg 1.000000e-02 train accuracy: 0.389300 val accuracy: 0.375300
lr 5.000000e-03 reg 3.000000e-02 train accuracy: 0.385050 val accuracy: 0.373500
lr 5.000000e-03 reg 5.000000e-02 train accuracy: 0.380450 val accuracy: 0.368600
lr 5.000000e-03 reg 7.000000e-02 train accuracy: 0.375725 val accuracy: 0.364200
lr 5.000000e-03 reg 9.000000e-02 train accuracy: 0.370450 val accuracy: 0.359900
lr 7.000000e-03 reg 1.000000e-02 train accuracy: 0.397075 val accuracy: 0.381700
lr 7.000000e-03 reg 3.000000e-02 train accuracy: 0.390175 val accuracy: 0.375100
lr 7.000000e-03 reg 5.000000e-02 train accuracy: 0.385200 val accuracy: 0.372200
lr 7.000000e-03 reg 7.000000e-02 train accuracy: 0.378775 val accuracy: 0.367900
lr 7.000000e-03 reg 9.000000e-02 train accuracy: 0.371975 val accuracy: 0.361200
lr 9.000000e-03 reg 1.000000e-02 train accuracy: 0.401300 val accuracy: 0.384500
lr 9.000000e-03 reg 3.000000e-02 train accuracy: 0.393275 val accuracy: 0.378600
lr 9.000000e-03 reg 5.000000e-02 train accuracy: 0.386825 val accuracy: 0.372900
lr 9.000000e-03 reg 7.000000e-02 train accuracy: 0.379400 val accuracy: 0.368600
lr 9.000000e-03 reg 9.000000e-02 train accuracy: 0.372225 val accuracy: 0.362700
lr 1.000000e-02 reg 1.000000e-02 train accuracy: 0.403650 val accuracy: 0.385500
lr 1.000000e-02 reg 3.000000e-02 train accuracy: 0.394525 val accuracy: 0.379700
lr 1.000000e-02 reg 5.000000e-02 train accuracy: 0.387500 val accuracy: 0.373000
lr 1.000000e-02 reg 7.000000e-02 train accuracy: 0.379875 val accuracy: 0.368700
lr 1.000000e-02 reg 9.000000e-02 train accuracy: 0.372850 val accuracy: 0.362400
best validation accuracy achieved during cross-validation: 0.385500
Saved in drive/My Drive/Colab Notebooks/DL2023/HW4/HW4/softmax_best_model.pt
```



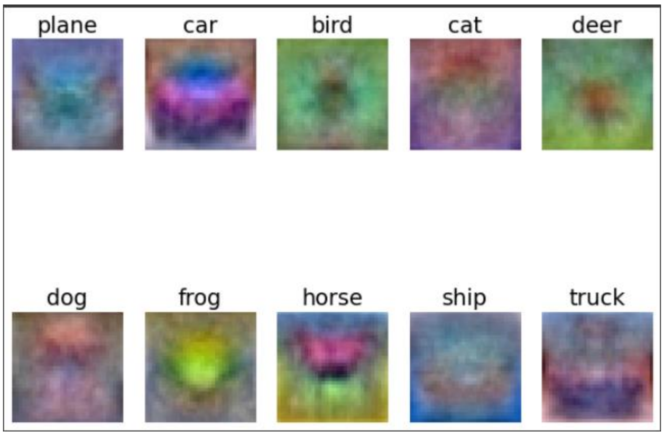


softmax_get_search_params 的程式碼如下：

```
learning_rates = [3e-3,5e-3,7e-3,9e-3,1e-2]
regularization_strengths = [1e-2,3e-2,5e-2,7e-2,9e-2]
```

(6).使用最佳的 learning_rate 和 regularization_strengths 來預測 test 的準確率，並把 w 的每一行的內容都畫出來。

```
softmax on raw pixels final test set accuracy: 0.393300
```



參.參考資料：

https://cs231n.github.io/linear-classify/

http://cs231n.github.io/optimization-1/