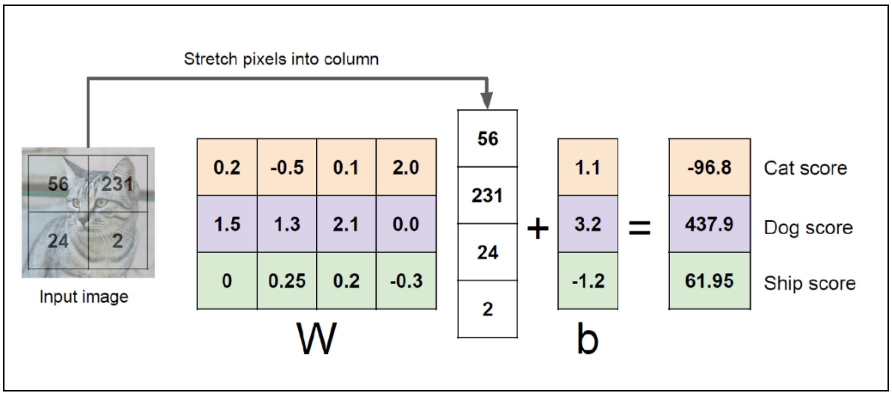
HW3 SVM Classifiers

M113040105 劉東霖

壹.SVM 介紹:

1.首先，我們透過權重矩陣 w 和 bias 矩陣 b 得到了狗和貓和車的score，如下圖所示:



2.定義 SVM 的公式如下圖所示，希望正確的分類的得分高於不正確項的得分，如果正確項≥錯誤項得分+邊界值，我們認為沒有誤差，如果正確項<錯誤項+邊界值，我們認為存在誤差（loss）。

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

以第一張圖來計算 SVM loss，此 loss 的計算方式為 max(0，437.9-(-96.8)+邊界)+ max(0，61.95-(-96.8)+邊界)，邊界通常為 1。

3.為了防止過擬合的情況發生，有時我們會在 loss 加上懲罰項(regularization term)來阻止權重變大。模型越複雜，正則化值就越大。最常見的 regularization term 如下:

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

4.完整的 SVM loss 由兩部分組成：data loss + regularization loss，我們用超參數λ來做為懲罰項的權重，如下圖所示:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

或這種形式，f(xi,w)j=WjXi:

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

5.我們需要計算 L 的梯度來優化 w，進而正確的分類。首先計算 data loss 的梯度，計算此需要分為兩個 part，i=j 和 i!=j。

i!=j 的計算過程如下圖所示:

$$① \ \nabla_{\mathbf{w}_j} L_i = \frac{\partial L_i}{\partial s_j}\frac{\partial s_j}{\partial \mathbf{w}_j} = 1_{(s_j - s_{y_i} + 1 > 0)}\mathbf{x}_i$$

Proof

$$\text{For } j \neq y_i \text{ , if } j = 1: \frac{\partial L_i}{\partial s_1} = \frac{\partial \max\left(0, s_1 - s_{y_i} + 1\right)}{\partial s_1} = 1_{(s_1 - s_{y_i} + 1 > 0)}$$

$$\text{, if } j = 2: \frac{\partial L_i}{\partial s_2} = \frac{\partial \max\left(0, s_2 - s_{y_i} + 1\right)}{\partial s_2} = 1_{\left(s_2 - s_{y_i} + 1 > 0\right)}$$

$$\Rightarrow \frac{\partial L_i}{\partial s_j} = 1_{(s_j - s_{y_i} + 1 > 0)}$$

$$\frac{\partial s_j}{\partial \mathbf{w}_j} = \frac{\partial \mathbf{w}_j^T \mathbf{x}_i}{\partial \mathbf{w}_j} = \mathbf{x}_i$$

i=j 的計算過程如下圖所示:

② $\nabla_{y_i} L_i = \frac{\partial L_i}{\partial s_{y_i}} \frac{\partial s_{y_i}}{\partial \mathbf{w}_{y_i}} = -\sum_{j \neq y_i} 1_{(s_j - s_{y_i} + 1 > 0)} \mathbf{x}_i$

Proof

$$\frac{\partial L_i}{\partial s_{y_i}} = \frac{\partial \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)}{\partial s_{y_i}} = \sum_{j \neq y_i} 1_{(s_j - s_{y_i} + 1 > 0)} \times (-1)$$

$$\frac{\partial s_j}{\partial \mathbf{w}_{y_i}} = \frac{\partial \mathbf{w}_{y_i}^T \mathbf{x}_i}{\partial \mathbf{w}_{y_i}} = \mathbf{x}_i$$

計算 regularization 的梯度如下圖:

$$\frac{\partial(\lambda \sum_k \sum_l W_{k,l}^2)}{\partial W} = 2\lambda W$$

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。

Loss 的梯度計算如下圖:

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

貳.SVM 實作:

1.SVM_loss_navive:

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

1.2 過程 :

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

```
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
    scores = W.t().mv(X[i])
    correct_class_score = scores[y[i]]
```

(2).裡面再一個 loop 為總共幾個 class，假如執行到正確的類別就跳過，不然會多加 1 次。沒有的話就按照 svm 公式計算 loss。

程式碼如下圖所示:

```
for j in range(num_classes):
    if j == y[i]:
        continue
    margin = scores[j] - correct_class_score + 1 # note delta = 1
```

(3).假如 margins>0，代表存在誤差，所以要讓 loss 去累加 margin。dw 的部分剛剛在介紹時有講過了，代入下圖的兩個公式:

$\nabla_{\mathbf{w}_j} L_i = \frac{\partial L_i}{\partial s_j}\frac{\partial s_j}{\partial \mathbf{w}_j} = \mathbb{1}_{(s_j - s_{y_i} + 1 > 0)} \mathbf{x}_i$
$\nabla_{y_i} L_i = \frac{\partial L_i}{\partial s_{y_i}}\frac{\partial s_{y_i}}{\partial \mathbf{w}_{y_i}} = -\sum_{j \neq y_i} \mathbb{1}_{(s_j - s_{y_i} + 1 > 0)} \mathbf{x}_i$

在 dw[:, j]時要累加 xi，在 dw[:, y[i]]時要減掉 xi，如下圖所示:

```
dW[:,y[i]]-=X[i]
dW[:,j]+=X[i]
```

假如 margins<0，代表沒有 loss 存在，直接忽略。

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

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

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
```

(5).全部程式碼如下:

```
dW = torch.zeros_like(W) # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
    scores = W.t().mv(X[i])
    correct_class_score = scores[y[i]]
    for j in range(num_classes):
        if j == y[i]:
            continue
        margin = scores[j] - correct_class_score + 1 # note delta = 1
        if margin > 0:
            loss += margin
            ###########################################################################
            # TODO:
            # Compute the gradient of the loss function and store it dW. (
            # Rather than first computing the loss and then computing the
            # derivative, it is simple to compute the derivative at the sam
            # that the loss is being computed.
            ###########################################################################
            # Replace "pass" statement with your code
            dW[:,y[i]]-=X[i]
            dW[:,j]+=X[i]
            ###########################################################################
            #                              END OF YOUR CODE
            ###########################################################################
```

```
loss /= num_train

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

######################################
# TODO:
# Compute the gradient of the lo
######################################
# Replace "pass" statement with y
dW/=num_train
dW+=reg*W*2
```

1.3 執行結果 :

(1). λ=0.000005 時代入 validation data 算出的 loss 值如下圖:

```
import usefuns
from linear_classifier import svm_loss_naive

usefuns.reset_seed(0)
# generate a random SVM weight tensor of small numbers
W = torch.randn(3073, 10, dtype=data_dict['X_val'].dtype, device=data_dict['X_val'].device) * 0

loss, _grad_ = svm_loss_naive(W, data_dict['X_val'], data_dict['y_val'], 0.000005)
print('loss: %f' % (loss, ))

loss: 9.000888
```

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

```
numerical: 0.031599 analytic: 0.031599, relative error: 3.139148e-07
numerical: 0.111444 analytic: 0.111444, relative error: 9.893848e-10
numerical: 0.011204 analytic: 0.011204, relative error: 1.003052e-06
numerical: -0.046128 analytic: -0.046128, relative error: 8.157190e-08
numerical: 0.071948 analytic: 0.071948, relative error: 1.000117e-07
numerical: 0.025688 analytic: 0.025688, relative error: 1.051337e-06
numerical: 0.185388 analytic: 0.185388, relative error: 7.039118e-09
numerical: -0.021740 analytic: -0.021740, relative error: 3.369463e-07
numerical: -0.159613 analytic: -0.159613, relative error: 6.416943e-08
numerical: 0.092690 analytic: 0.092690, relative error: 1.748534e-07
```

(3). λ=1000 時，numerical gradient 和 analytic gradient 的誤差皆小於 1e-5，如下圖所示:

```
numerical: 0.124849 analytic: 0.124849, relative error: 6.251581e-08
numerical: 0.168915 analytic: 0.168915, relative error: 5.957873e-09
numerical: 0.148752 analytic: 0.148752, relative error: 8.733009e-08
numerical: -0.024936 analytic: -0.024936, relative error: 6.470254e-08
numerical: -0.008570 analytic: -0.008570, relative error: 7.174549e-07
numerical: -0.103155 analytic: -0.103155, relative error: 2.498757e-07
numerical: -0.335573 analytic: -0.335573, relative error: 4.472387e-09
numerical: -0.222176 analytic: -0.222176, relative error: 4.264915e-08
numerical: 0.681163 analytic: 0.681163, relative error: 1.235571e-08
numerical: -0.004090 analytic: -0.004089, relative error: 4.327973e-06
```

2. svm_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

(2).利用下圖的方式把每一筆 data 的正確類別找出來。下面第 1 張圖和第 2 張圖是一樣的方法，但題目說不能使用 for loop，所以我們使用第一張圖的方式找出每一筆 data 的正確類別。

```
corresponds=scores[range(X.shape[0]),y].
```

```
for i in range(X.shape[0]):
    corresponds[i] = scores[i,y[i]]
```

出來的結果的 shape 是(N)，要 reshape 成(N,1)後面才能用廣播的方式運算。

(3).代入 svm 的公式，如下圖所示:

```
margins=torch.maximum(torch.zeros_like(scores),scores-corresponds+1)
```

因上面的公式有重複計算到正確類別的標籤，所以要把重複計算到正確類別的標籤變成 0，如下圖所示:

```
margins[range(X.shape[0]),y]=0
```

(4).代入 loss 的公式，把剛剛算的 margins 加起來除 train 的數量，再加 regularization term 就是 total loss，如下圖所示:

```
loss=torch.sum(margins)/X.shape[0]
loss+=reg*torch.sum(W * W)
```

(5)再來要計算的是 dw 的部分。

首先讓 margin>0 的部分=1 滿足下面公式的 1 那個部分:

$$\nabla_{\mathbf{w}_j} L_i = \frac{\partial L_i}{\partial s_j}\frac{\partial s_j}{\partial \mathbf{w}_j} = 1_{(s_j - s_{y_i} + 1 > 0)}\mathbf{x}_i$$

margins 中各樣本對應於正確標籤位置的值通過減去各行之和得到下圖的結果。

$$\nabla_{y_i} L_i = \frac{\partial L_i}{\partial s_{y_i}} \frac{\partial s_{y_i}}{\partial \mathbf{w}_{y_i}} = -\sum_{j \neq y_i} \mathbb{1}_{(s_j - s_{y_i} + 1 > 0)} \mathbf{x}_i$$

最終梯度就是用 X 與 margins 的矩陣乘積，然後再加上正則化處理項。上述執行過程如下圖所示:

```
binary=margins
binary[binary>0]=1
row_sum=torch.sum(binary,axis=1)
binary[range(X.shape[0]),y]-=row_sum
dW=X.T  @  binary/X.shape[0]
dW+=reg*W*2
```

(6).全部程式碼:

```
scores=X  @  W
#score  shape:(128,10)
corresponds=scores[range(X.shape[0]),y].reshape(-1,1)
#different  x  mapping  different  y, shape  is  (128)->(128,1)
margins=torch.maximum(torch.zeros_like(scores),scores-corresponds+1)
margins[range(X.shape[0]),y]=0
loss=torch.sum(margins)/X.shape[0]
loss+=reg*torch.sum(W  *  W)
```

```
binary=margins
binary[binary>0]=1
row_sum=torch.sum(binary,axis=1)
binary[range(X.shape[0]),y]-=row_sum
dW=X.T  @  binary/X.shape[0]
dW+=reg*W*2
```

2.3 執行結果 :

(1).有無用向量化操作的區別，發現雖然計算有些微小的差異，但 speedup 卻差很多，如下兩圖所示:

```
Naive loss: 9.002394e+00 computed in 212.80ms
Vectorized loss: 9.002394e+00 computed in 2.99ms
Difference: -5.33e-15
Speedup: 71.07X
```

```
Naive loss and gradient: computed in 305.05ms
Vectorized loss and gradient: computed in 2.85ms
Gradient difference: 1.87e-14
Speedup: 107.21X
```

3.SVM 的各種不同的測試:

3.1.過程

(1). sample_batch:題目指定要產生 shape 為(batch_size,)大小的亂數來決定 batch 裡面的 data。產生亂數的方式如下圖:

```
indices=torch.randint(0,num_train,(batch_size,))
X_batch,y_batch=X[indices],y[indices]
```

(2). train_linear_classifier:題目叫我們用 gradient 和 learning rate 去更新權重，程式如下:

```
W-=grad*learning_rate
```

(3). predict linear classifier:先用 X@W 算出 scores，再找出每一個 row 裡面 scores 最大的地方，就完成預測了，程式碼如下:

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

(4). svm_get_search_params:題目要求用不同的 learning_rate 和 regularization_strengths 來找最佳解，我就照著原本 cell 留下來的結果裡面的參數來測試，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]
```

(5). test_one_param_set:題目要我們訓練單個 LinearClassifier 實例，並返回學習到的實例和訓練/驗證的準確度。

首先，先把 train 和 validation 的 data 拿出來。

```
x_train,y_train=data_dict['X_train'],data_dict['y_train']
x_val,y_val=data_dict['X_val'],data_dict['y_val']
```

再來，把測試資料和參數代入 LinearClassifier 訓練和預測結果，並計算準確率。

```
cls.train(x_train,y_train,lr,reg,num_iters)
train_pred=cls.predict(x_train)
train_acc=torch.sum(train_pred==y_train)/y_train.shape[0]
```

把 validation 代入 LinearClassifier 預測結果，並計算準確率。

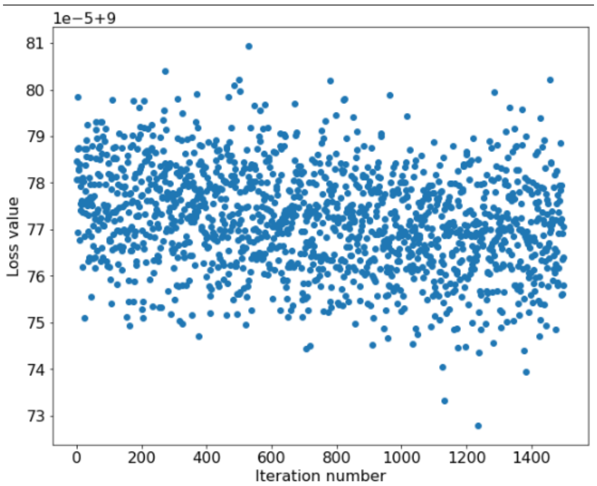
```
val_pred=cls.predict(x_val)
val_acc=torch.sum(val_pred==y_val)/y_val.shape[0]
```

3.3 執行結果:

(1). 在下圖參數下執行的 loss，發現 loss 幾乎都一樣沒什麼改變。並把他每個 loss 都 plot 出來。

```
W, loss_hist = train_linear_classifier(svm_loss_vectorized, None,
                                       data_dict['X_train'],
                                       data_dict['y_train'],
                                       learning_rate=3e-11, reg=2.5e4,
                                       num_iters=1500, verbose=True)

torch.cuda.synchronize()
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 9.000785
iteration 100 / 1500: loss 9.000762
iteration 200 / 1500: loss 9.000777
iteration 300 / 1500: loss 9.000766
iteration 400 / 1500: loss 9.000778
iteration 500 / 1500: loss 9.000771
iteration 600 / 1500: loss 9.000772
iteration 700 / 1500: loss 9.000770
iteration 800 / 1500: loss 9.000772
iteration 900 / 1500: loss 9.000772
iteration 1000 / 1500: loss 9.000770
iteration 1100 / 1500: loss 9.000789
iteration 1200 / 1500: loss 9.000787
iteration 1300 / 1500: loss 9.000769
iteration 1400 / 1500: loss 9.000778
That took 2.439793s
```



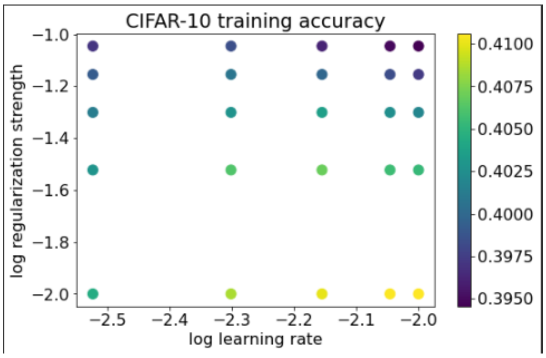
(2).把 train 和 validation 放入 predict linear classifier 預測，發現結果很慘，如下圖所示:

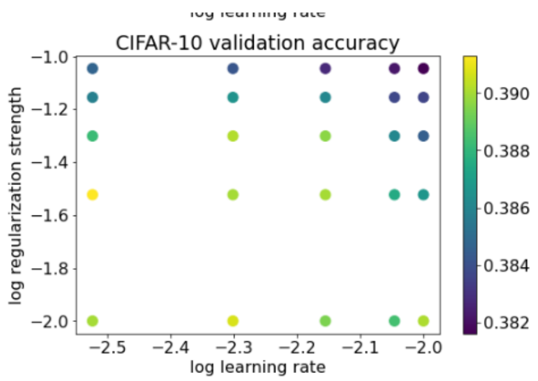
```
y_train_pred = predict_linear_classifier(W, data_dict['X_train'])
train_acc = 100.0 * (data_dict['y_train'] == y_train_pred).double().mean().item()
print('Training accuracy: %.2f%%' % train_acc)

y_val_pred = predict_linear_classifier(W, data_dict['X_val'])
val_acc = 100.0 * (data_dict['y_val'] == y_val_pred).double().mean().item()
print('Validation accuracy: %.2f%%' % val_acc)

Training accuracy: 9.24%
Validation accuracy: 9.00%
```

(3).利用 svm_get_search_params 設的 learning_rate 和 regularization_strengths 來找最佳解，並把他 plot 出來，顏色越深代表準確率越高。

```
Training SVM 1 / 25 with learning_rate=3.000000e-03 and reg=1.000000e-02
Training SVM 2 / 25 with learning_rate=3.000000e-03 and reg=3.000000e-02
Training SVM 3 / 25 with learning_rate=3.000000e-03 and reg=5.000000e-02
Training SVM 4 / 25 with learning_rate=3.000000e-03 and reg=7.000000e-02
Training SVM 5 / 25 with learning_rate=3.000000e-03 and reg=9.000000e-02
Training SVM 6 / 25 with learning_rate=5.000000e-03 and reg=1.000000e-02
Training SVM 7 / 25 with learning_rate=5.000000e-03 and reg=3.000000e-02
Training SVM 8 / 25 with learning_rate=5.000000e-03 and reg=5.000000e-02
Training SVM 9 / 25 with learning_rate=5.000000e-03 and reg=7.000000e-02
Training SVM 10 / 25 with learning_rate=5.000000e-03 and reg=9.000000e-02
Training SVM 11 / 25 with learning_rate=7.000000e-03 and reg=1.000000e-02
Training SVM 12 / 25 with learning_rate=7.000000e-03 and reg=3.000000e-02
Training SVM 13 / 25 with learning_rate=7.000000e-03 and reg=5.000000e-02
Training SVM 14 / 25 with learning_rate=7.000000e-03 and reg=7.000000e-02
Training SVM 15 / 25 with learning_rate=7.000000e-03 and reg=9.000000e-02
Training SVM 16 / 25 with learning_rate=9.000000e-03 and reg=1.000000e-02
Training SVM 17 / 25 with learning_rate=9.000000e-03 and reg=3.000000e-02
Training SVM 18 / 25 with learning_rate=9.000000e-03 and reg=5.000000e-02
Training SVM 19 / 25 with learning_rate=9.000000e-03 and reg=7.000000e-02
Training SVM 20 / 25 with learning_rate=9.000000e-03 and reg=9.000000e-02
Training SVM 21 / 25 with learning_rate=1.000000e-02 and reg=1.000000e-02
Training SVM 22 / 25 with learning_rate=1.000000e-02 and reg=3.000000e-02
Training SVM 23 / 25 with learning_rate=1.000000e-02 and reg=5.000000e-02
Training SVM 24 / 25 with learning_rate=1.000000e-02 and reg=7.000000e-02
Training SVM 25 / 25 with learning_rate=1.000000e-02 and reg=9.000000e-02
```

```
lr 3.000000e-03 reg 1.000000e-02 train accuracy: 0.404425 val accuracy: 0.390000
lr 3.000000e-03 reg 3.000000e-02 train accuracy: 0.402850 val accuracy: 0.391300
lr 3.000000e-03 reg 5.000000e-02 train accuracy: 0.401300 val accuracy: 0.388200
lr 3.000000e-03 reg 7.000000e-02 train accuracy: 0.399175 val accuracy: 0.385800
lr 3.000000e-03 reg 9.000000e-02 train accuracy: 0.397000 val accuracy: 0.384700
lr 5.000000e-03 reg 1.000000e-02 train accuracy: 0.408575 val accuracy: 0.390700
lr 5.000000e-03 reg 3.000000e-02 train accuracy: 0.406175 val accuracy: 0.390000
lr 5.000000e-03 reg 5.000000e-02 train accuracy: 0.402825 val accuracy: 0.390200
lr 5.000000e-03 reg 7.000000e-02 train accuracy: 0.400800 val accuracy: 0.386600
lr 5.000000e-03 reg 9.000000e-02 train accuracy: 0.398425 val accuracy: 0.384200
lr 7.000000e-03 reg 1.000000e-02 train accuracy: 0.410025 val accuracy: 0.389400
lr 7.000000e-03 reg 3.000000e-02 train accuracy: 0.406925 val accuracy: 0.390000
lr 7.000000e-03 reg 5.000000e-02 train accuracy: 0.403550 val accuracy: 0.389600
lr 7.000000e-03 reg 7.000000e-02 train accuracy: 0.399850 val accuracy: 0.386100
lr 7.000000e-03 reg 9.000000e-02 train accuracy: 0.396175 val accuracy: 0.382700
lr 9.000000e-03 reg 1.000000e-02 train accuracy: 0.410525 val accuracy: 0.388400
lr 9.000000e-03 reg 3.000000e-02 train accuracy: 0.405525 val accuracy: 0.387500
lr 9.000000e-03 reg 5.000000e-02 train accuracy: 0.402450 val accuracy: 0.386100
lr 9.000000e-03 reg 7.000000e-02 train accuracy: 0.398325 val accuracy: 0.383700
lr 9.000000e-03 reg 9.000000e-02 train accuracy: 0.394925 val accuracy: 0.382200
lr 1.000000e-02 reg 1.000000e-02 train accuracy: 0.410600 val accuracy: 0.390100
lr 1.000000e-02 reg 3.000000e-02 train accuracy: 0.405275 val accuracy: 0.386700
lr 1.000000e-02 reg 5.000000e-02 train accuracy: 0.401900 val accuracy: 0.384500
lr 1.000000e-02 reg 7.000000e-02 train accuracy: 0.397325 val accuracy: 0.383600
lr 1.000000e-02 reg 9.000000e-02 train accuracy: 0.394550 val accuracy: 0.381600
best validation accuracy achieved during cross-validation: 0.391300
Saved in drive/My Drive/Colab Notebooks/DL2023/HW4/HW4/svm_best_model.pt
```





(4).使用最佳的 learning_rate 和 regularization_strengths 來預測 test 的準確率和 model 預測出來的 label。

```python
import usefuns

usefuns.reset_seed(0)
y_test_pred = best_svm_model.predict(data_dict['X_test'])
test_accuracy = torch.mean((data_dict['y_test'] == y_test_pred).double())
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.392300
```