SOI1010 Machine Learning II - Assignment #2

작성자: 2023036299김진욱

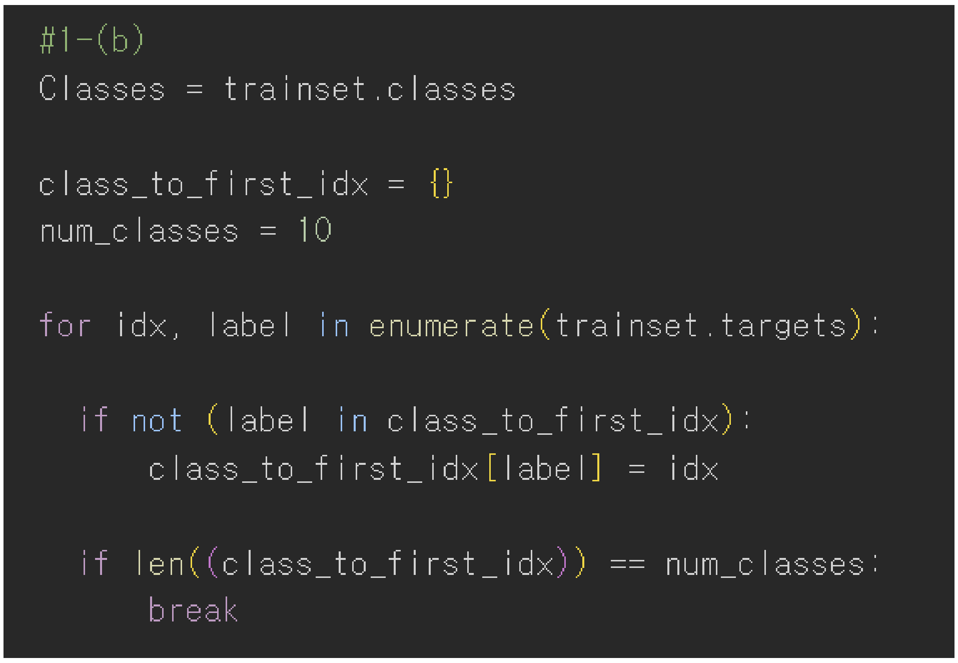
Colab URL Link:

https://colab.research.google.com/drive/1JB4B1ezh5W695VJQyKI01hb_NM3kR1up?usp=sharing

(I have allowed access to users affiliated with Hanyang University, but I am also including ipynb and py just in case.)

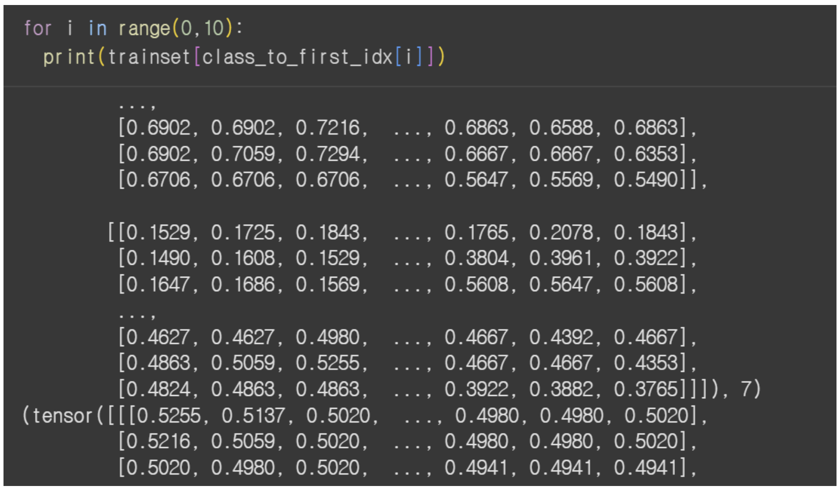
Problem #1: Binary Classification via soft-margin SVM on CIFAR10

b) Visualize at least one image for each class. You may need to look into how dataset is implemented in PyTorch.



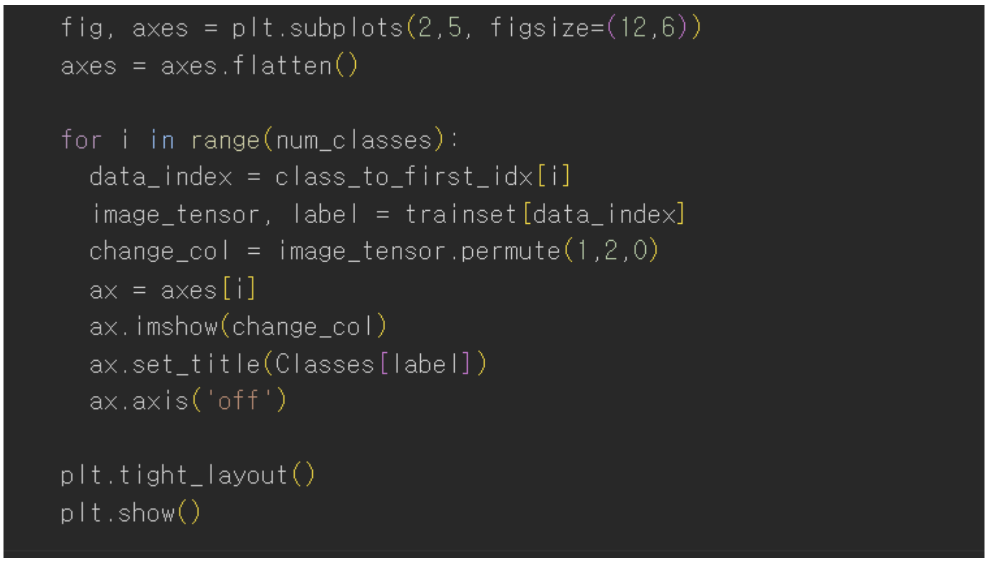
```
#1-(b)
Classes = trainset.classes

class_to_first_idx = {}
num_classes = 10

for idx, label in enumerate(trainset.targets):

  if not (label in class_to_first_idx):
      class_to_first_idx[label] = idx

  if len((class_to_first_idx)) == num_classes:
      break
```

Torch's CIFAR10 doesn't assign indices to class data by region, so we need to iterate through the data to find the label-specific index. Therefore, we created a Classes variable to store the names of each class in the trainset, and stored the index information for each class to be visualized through a dictionary.



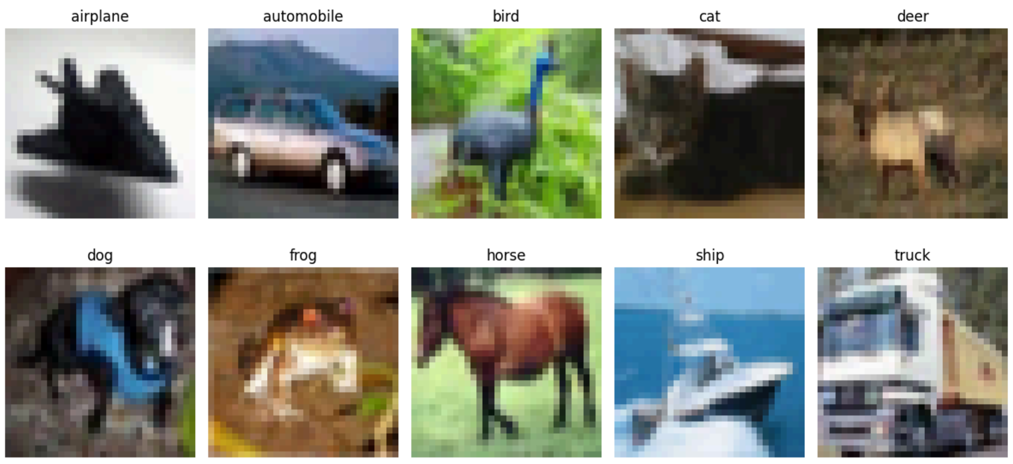
```
for i in range(0,10):
  print(trainset[class_to_first_idx[i]])

         ...,
         [0.6902, 0.6902, 0.7216,  ..., 0.6863, 0.6588, 0.6863],
         [0.6902, 0.7059, 0.7294,  ..., 0.6667, 0.6667, 0.6353],
         [0.6706, 0.6706, 0.6706,  ..., 0.5647, 0.5569, 0.5490]],

        [[0.1529, 0.1725, 0.1843,  ..., 0.1765, 0.2078, 0.1843],
         [0.1490, 0.1608, 0.1529,  ..., 0.3804, 0.3961, 0.3922],
         [0.1647, 0.1686, 0.1569,  ..., 0.5608, 0.5647, 0.5608],
         ...,
         [0.4627, 0.4627, 0.4980,  ..., 0.4667, 0.4392, 0.4667],
         [0.4863, 0.5059, 0.5255,  ..., 0.4667, 0.4667, 0.4353],
         [0.4824, 0.4863, 0.4863,  ..., 0.3922, 0.3882, 0.3765]]]), 7)
(tensor([[[0.5255, 0.5137, 0.5020,  ..., 0.4980, 0.4980, 0.5020],
         [0.5216, 0.5059, 0.5020,  ..., 0.4980, 0.4980, 0.5020],
         [0.5020, 0.4980, 0.5020,  ..., 0.4941, 0.4941, 0.4941],
```

Referring to the visualization of fashinMNIST using matplotlib in Datasets & DataLoaders in week4_Labtime, I visualized it using matplotlib.

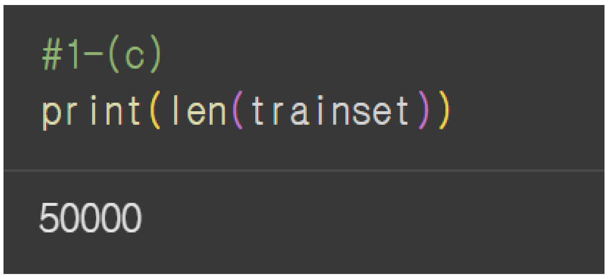


```
fig, axes = plt.subplots(2,5, figsize=(12,6))
axes = axes.flatten()

for i in range(num_classes):
  data_index = class_to_first_idx[i]
  image_tensor, label = trainset[data_index]
  change_col = image_tensor.permute(1,2,0)
  ax = axes[i]
  ax.imshow(change_col)
  ax.set_title(Classes[label])
  ax.axis('off')

plt.tight_layout()
plt.show()
```

Unlike FASHIONMNIST, which was a black and white photo so there was no need to consider color, CIFAR10 had color, so before visualizing it using matplotlib, the column order of (channel, height, width) had to be changed to (height, channel, width).



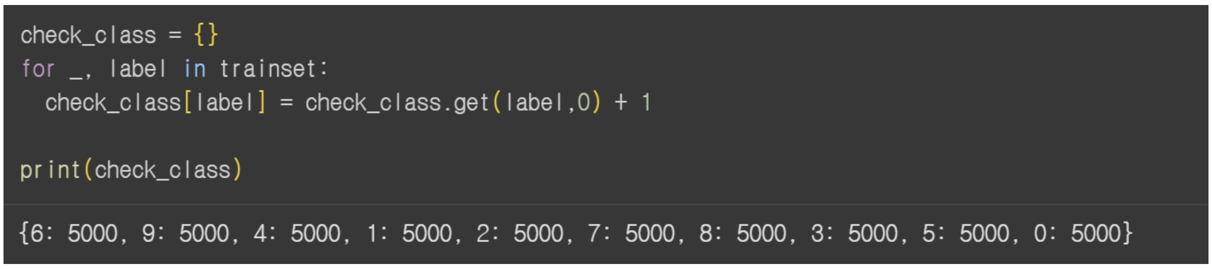
c) Split the trainset into training set and validation set with 90% : 10% ratio. Implement dataloaders for CIFAR10



```
#1-(c)
print(len(trainset))
```

```
50000
```

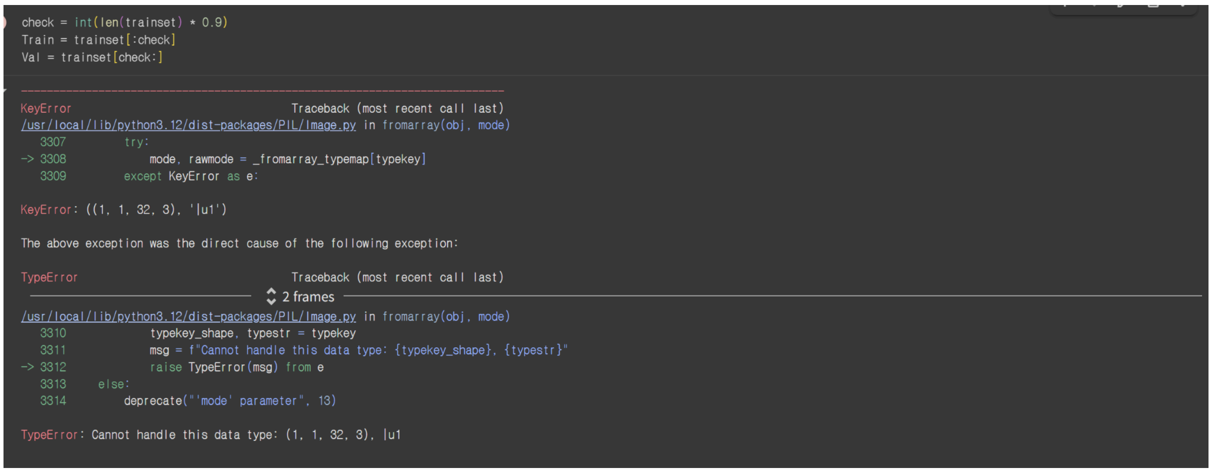
First, we checked the size of the entire dataset and confirmed that there were a total of 5,000 photo data.



```
check_class = {}
for _, label in trainset:
  check_class[label] = check_class.get(label,0) + 1

print(check_class)
```

```
{6: 5000, 9: 5000, 4: 5000, 1: 5000, 2: 5000, 7: 5000, 8: 5000, 3: 5000, 5: 5000, 0: 5000}
```

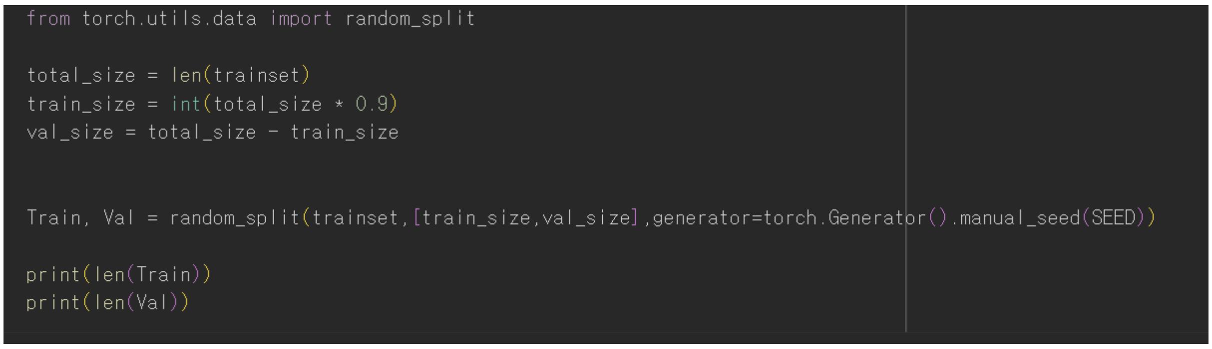
We also confirmed that 5,000 were evenly distributed per label.



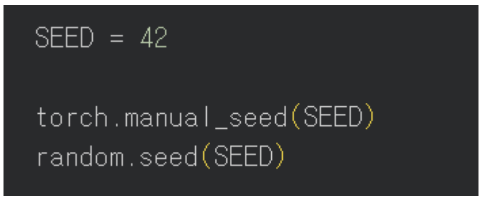
```
check = int(len(trainset) * 0.9)
Train = trainset[:check]
Val = trainset[check:]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.12/dist-packages/PIL/Image.py in fromarray(obj, mode)
   3307         try:
-> 3308             mode, rawmode = _fromarray_typemap[typekey]
   3309         except KeyError as e:

KeyError: ((1, 1, 32, 3), '|u1')

The above exception was the direct cause of the following exception:

TypeError                                 Traceback (most recent call last)
                                  2 frames
/usr/local/lib/python3.12/dist-packages/PIL/Image.py in fromarray(obj, mode)
   3310             typekey_shape, typestr = typekey
   3311             msg = f"Cannot handle this data type: {typekey_shape}, {typestr}"
-> 3312             raise TypeError(msg) from e
   3313     else:
   3314         deprecate("'mode' parameter", 13)

TypeError: Cannot handle this data type: (1, 1, 32, 3), |u1
```

I just tried to do simple slicing and got an error.
->When slicing torchdataset, it is said that slicing is usually done through random_split (for randomness).

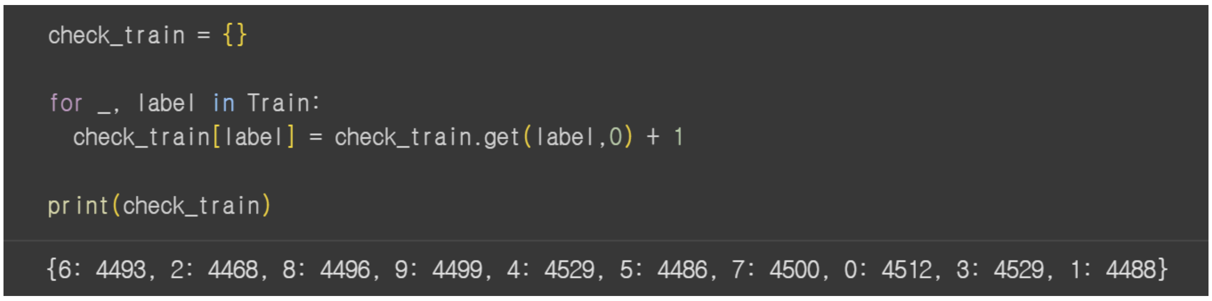


```
from torch.utils.data import random_split

total_size = len(trainset)
train_size = int(total_size * 0.9)
val_size = total_size - train_size


Train, Val = random_split(trainset,[train_size,val_size],generator=torch.Generator().manual_seed(SEED))

print(len(Train))
print(len(Val))
```



```python
SEED = 42

torch.manual_seed(SEED)
random.seed(SEED)
```

Don't forget to fix the SEED for reproduction.



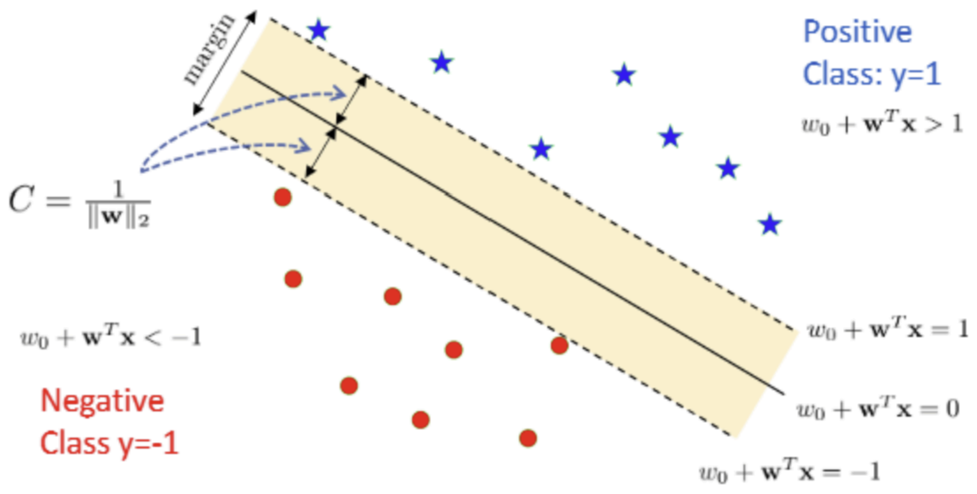
```python
check_train = {}

for _, label in Train:
  check_train[label] = check_train.get(label,0) + 1

print(check_train)
```

```
{6: 4493, 2: 4468, 8: 4496, 9: 4499, 4: 4529, 5: 4486, 7: 4500, 0: 4512, 3: 4529, 1: 4488}
```

The label distribution in the Training_Set is as follows. I was a little concerned because it wasn't exactly 4,500, but it turns out that this is actually a case where the random split worked well.

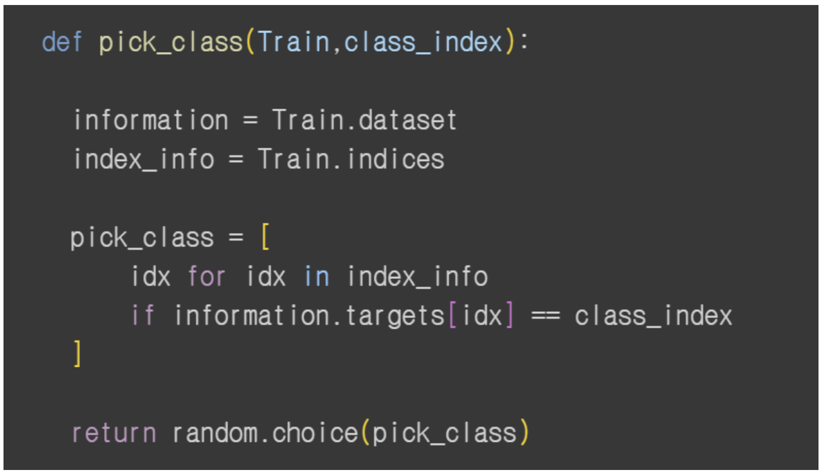
1-d) Choose any two classes. Then, make a SVM classifier

I'm planning to use SVM to classify airplanes and ships as binary classes. This is because the dataset contains colors, and airplanes and ships are likely to have similar backgrounds (probably blue during the day, and dark at night). I thought it would be good to check whether the model has truly learned the unique shape of the class, rather than just the color information, so I decided to compare.

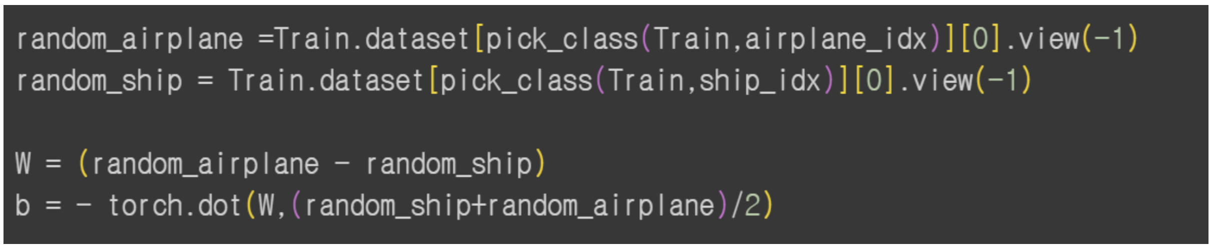

margin
Positive
Class: y=1
$w_0 + \mathbf{w}^T\mathbf{x} > 1$
$C = \frac{1}{\|\mathbf{w}\|_2}$
$w_0 + \mathbf{w}^T\mathbf{x} = 1$
$w_0 + \mathbf{w}^T\mathbf{x} < -1$
$w_0 + \mathbf{w}^T\mathbf{x} = 0$
Negative
Class y=-1
$w_0 + \mathbf{w}^T\mathbf{x} = -1$

SVM is a relatively simple model with excellent classification ability, as it uses small weights and large margins for HyperPlane. I was concerned about how to set the initial θ of HyperPlane to compare these two classes. (To be clear, I approached this completely wrongly. However, I will discuss this further down.)

First, we decided to select one from the airplane set and one from the ship set, and initialize the weights and bias based on the distance between these two.



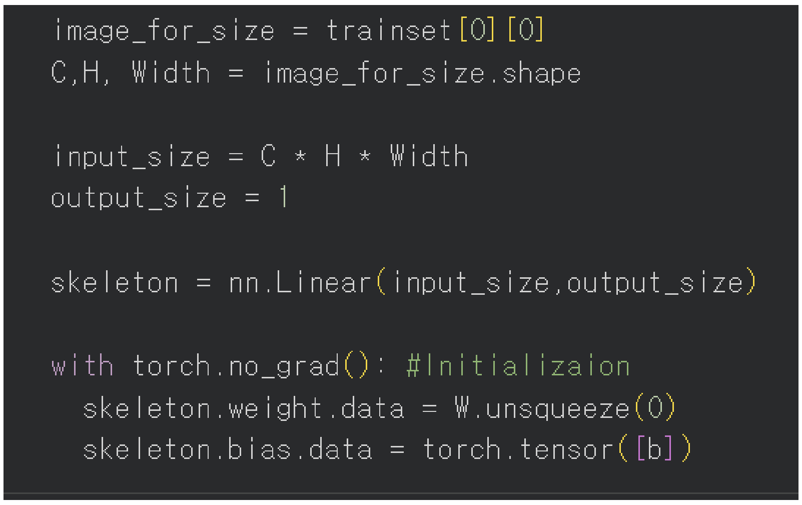
```
def pick_class(Train,class_index):

  information = Train.dataset
  index_info = Train.indices

  pick_class = [
      idx for idx in index_info
      if information.targets[idx] == class_index
  ]

  return random.choice(pick_class)
```



```
random_airplane =Train.dataset[pick_class(Train,airplane_idx)][0].view(-1)
random_ship = Train.dataset[pick_class(Train,ship_idx)][0].view(-1)

W = (random_airplane - random_ship)
b = - torch.dot(W,(random_ship+random_airplane)/2)
```

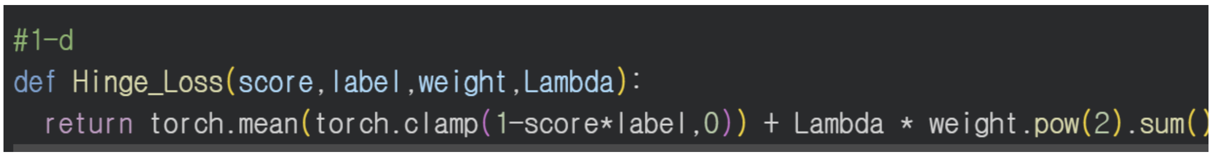
The weights were created by flattening the image data of the random data ([0].view(-1)) and then subtracting them, and the bias was created by taking the inner product of the midpoint between the weights and the two labels.

The skeleton of the model that will be responsible for subsequent learning was also defined as follows.



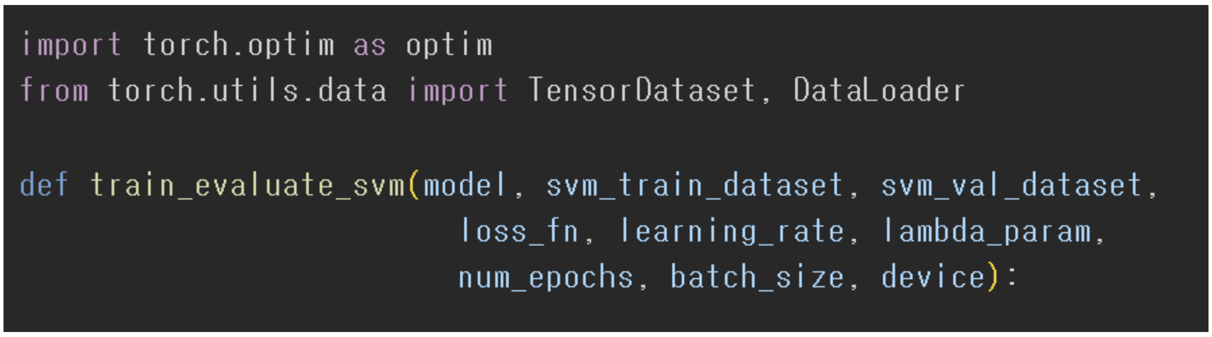
```
  image_for_size = trainset[0][0]
  C,H, Width = image_for_size.shape

  input_size = C * H * Width
  output_size = 1

  skeleton = nn.Linear(input_size,output_size)

  with torch.no_grad(): #Initializaion
    skeleton.weight.data = W.unsqueeze(0)
    skeleton.bias.data = torch.tensor([b])
```

1-(d) Choose any two classes. Then, make a SVM classifier



```
#1-d
def Hinge_Loss(score,label,weight,Lambda):
  return torch.mean(torch.clamp(1-score*label,0)) + Lambda * weight.pow(2).sum()
```

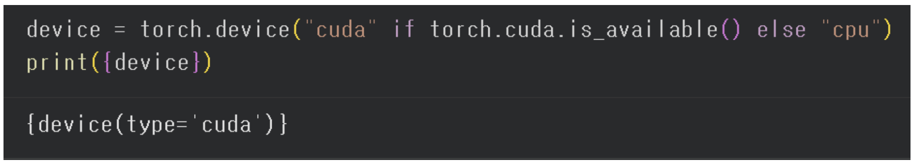
I did not apply sqrt to L2 Norm. sqrt is monotonically increasing anyway, and to prevent the situation where 0 is included in the denominator of the derivative,



```
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

def train_evaluate_svm(model, svm_train_dataset, svm_val_dataset,
                       loss_fn, learning_rate, lambda_param,
                       num_epochs, batch_size, device):
```

In order to efficiently proceed with the process from e to j in the future, I thought it would be good to make the learning process into a single function, so I defined a new train_evaluate_svm function. (And luckily, this function was also useful in Problem #2.)

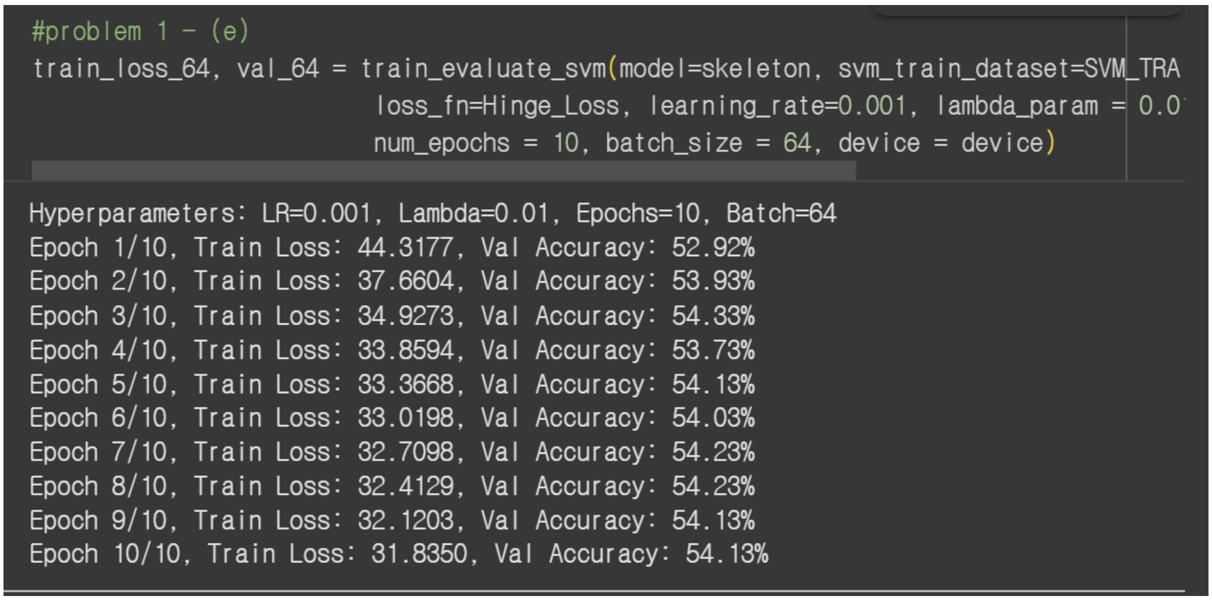
This function receives the model, dataset, and loss function as parameters (learning rate, regularization, total number of learnings, size) and whether to use CUDA as arguments, quickly trains the model using powerful parallel computing based on CUDA, and then returns the loss function value and verification accuracy during the learning process.



```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print({device})

{device(type='cuda')}
```

1-e) Train for 10 epochs with batch size 64.

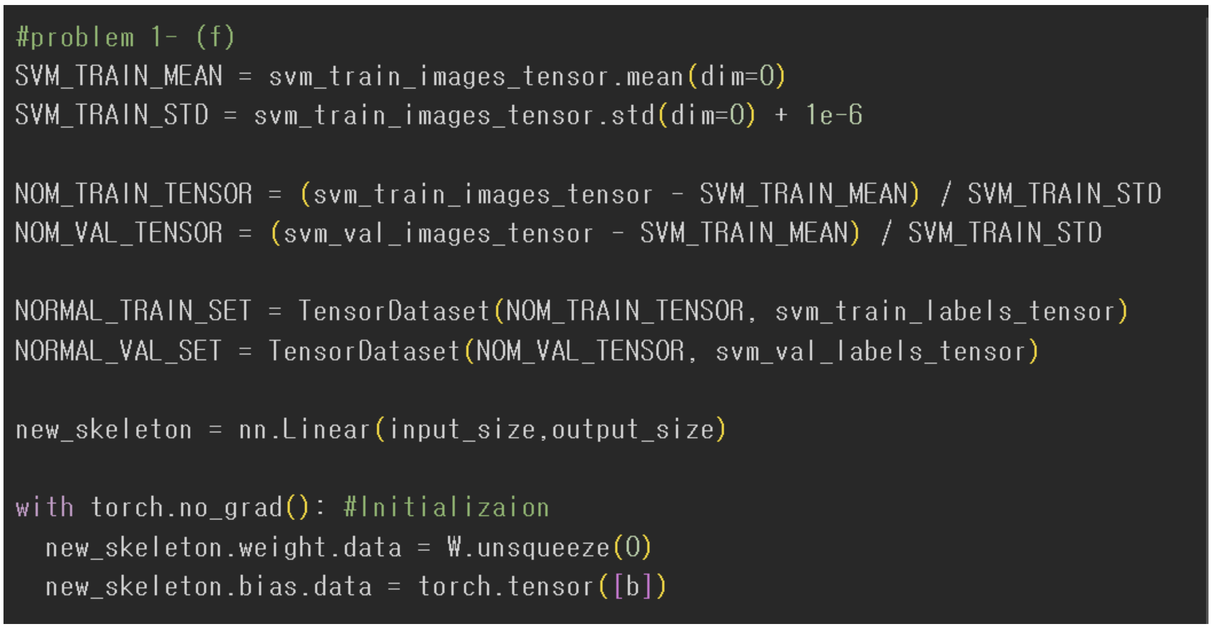
Except for Epoch and batch_size, the remaining hyperparameter values were set to 0.01 for lambda and 0.001 for LEARNING_RATE by default.



```
#problem 1 - (e)
train_loss_64, val_64 = train_evaluate_svm(model=skeleton, svm_train_dataset=SVM_TRA
                        loss_fn=Hinge_Loss, learning_rate=0.001, lambda_param = 0.0
                        num_epochs = 10, batch_size = 64, device = device)

Hyperparameters: LR=0.001, Lambda=0.01, Epochs=10, Batch=64
Epoch 1/10, Train Loss: 44.3177, Val Accuracy: 52.92%
Epoch 2/10, Train Loss: 37.6604, Val Accuracy: 53.93%
Epoch 3/10, Train Loss: 34.9273, Val Accuracy: 54.33%
Epoch 4/10, Train Loss: 33.8594, Val Accuracy: 53.73%
Epoch 5/10, Train Loss: 33.3668, Val Accuracy: 54.13%
Epoch 6/10, Train Loss: 33.0198, Val Accuracy: 54.03%
Epoch 7/10, Train Loss: 32.7098, Val Accuracy: 54.23%
Epoch 8/10, Train Loss: 32.4129, Val Accuracy: 54.23%
Epoch 9/10, Train Loss: 32.1203, Val Accuracy: 54.13%
Epoch 10/10, Train Loss: 31.8350, Val Accuracy: 54.13%
```

As the EPOCH increases, the loss decreases and the accuracy increases, but the difference is not significant.

1-f) Perform data normalization

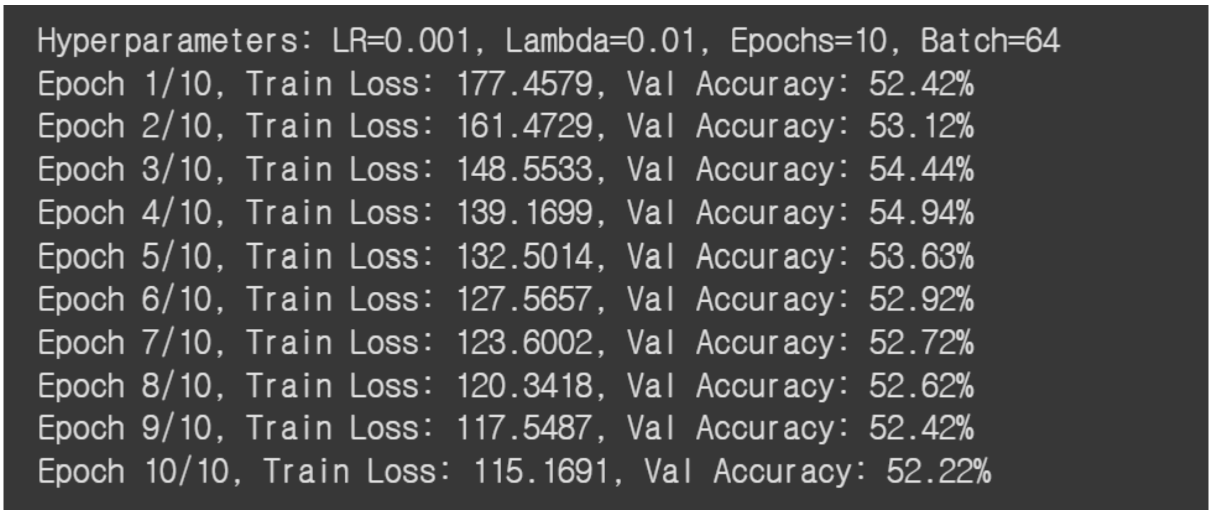


```
#problem 1- (f)
SVM_TRAIN_MEAN = svm_train_images_tensor.mean(dim=0)
SVM_TRAIN_STD = svm_train_images_tensor.std(dim=0) + 1e-6

NOM_TRAIN_TENSOR = (svm_train_images_tensor - SVM_TRAIN_MEAN) / SVM_TRAIN_STD
NOM_VAL_TENSOR = (svm_val_images_tensor - SVM_TRAIN_MEAN) / SVM_TRAIN_STD

NORMAL_TRAIN_SET = TensorDataset(NOM_TRAIN_TENSOR, svm_train_labels_tensor)
NORMAL_VAL_SET = TensorDataset(NOM_VAL_TENSOR, svm_val_labels_tensor)

new_skeleton = nn.Linear(input_size,output_size)

with torch.no_grad(): #Initializaion
  new_skeleton.weight.data = W.unsqueeze(0)
  new_skeleton.bias.data = torch.tensor([b])
```

The mean and standard deviation of the train data were calculated for each feature (dim=0), and then the normalization process was performed according to broadcasting and i.i.d. (basically, train and val are independent but have the same distribution), and a correction value of 1e-6 was added to the standard deviation to prevent division by 0.

The existing skeleton was already learned through 1-(e), so a new one was created.

1-g) Again, train for 10 epochs with batch size 64 after data normalization.



```
Hyperparameters: LR=0.001, Lambda=0.01, Epochs=10, Batch=64
Epoch 1/10, Train Loss: 177.4579, Val Accuracy: 52.42%
Epoch 2/10, Train Loss: 161.4729, Val Accuracy: 53.12%
Epoch 3/10, Train Loss: 148.5533, Val Accuracy: 54.44%
Epoch 4/10, Train Loss: 139.1699, Val Accuracy: 54.94%
Epoch 5/10, Train Loss: 132.5014, Val Accuracy: 53.63%
Epoch 6/10, Train Loss: 127.5657, Val Accuracy: 52.92%
Epoch 7/10, Train Loss: 123.6002, Val Accuracy: 52.72%
Epoch 8/10, Train Loss: 120.3418, Val Accuracy: 52.62%
Epoch 9/10, Train Loss: 117.5487, Val Accuracy: 52.42%
Epoch 10/10, Train Loss: 115.1691, Val Accuracy: 52.22%
```

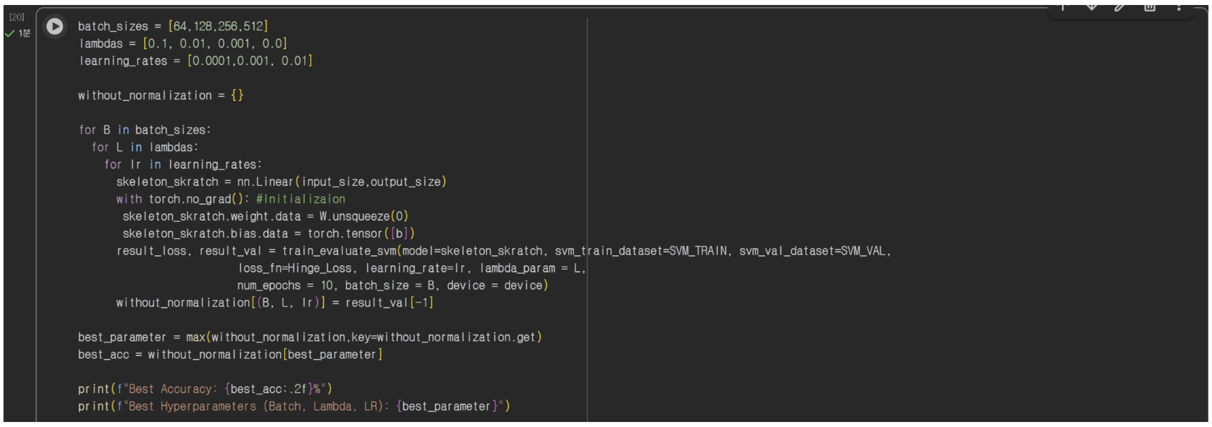
After repeating the epchos after normalization and seeing the accuracy fluctuate, I realized at this point that something was wrong. Normalization is supposed to stabilize the domain and improve model performance, but it produced unexpected results, which made me realize something was wrong.

1-h) What are the hyperparameters you can tune?

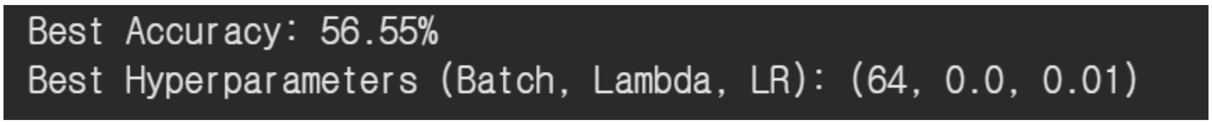
```
batch_sizes = [64,128,256,512]
lambdas = [0.1, 0.01, 0.001, 0.0]
learning_rates = [0.0001,0.001, 0.01]
```

To prevent the model from overfitting to the train_set and to prevent the training time from becoming too long, epichocs were excluded from hyperparameter tuning.

1-i) Try to obtain find optimal hyperparameters.

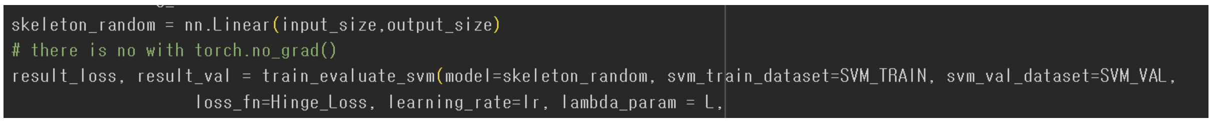


```
batch_sizes = [64,128,256,512]
lambdas = [0.1, 0.01, 0.001, 0.0]
learning_rates = [0.0001,0.001, 0.01]

without_normalization = {}

for B in batch_sizes:
  for L in lambdas:
    for lr in learning_rates:
      skeleton_skratch = nn.Linear(input_size,output_size)
      with torch.no_grad(): #Initializaion
       skeleton_skratch.weight.data = W.unsqueeze(0)
       skeleton_skratch.bias.data = torch.tensor([b])
      result_loss, result_val = train_evaluate_svm(model=skeleton_skratch, svm_train_dataset=SVM_TRAIN, svm_val_dataset=SVM_VAL,
                          loss_fn=Hinge_Loss, learning_rate=lr, lambda_param = L,
                          num_epochs = 10, batch_size = B, device = device)
      without_normalization[(B, L, lr)] = result_val[-1]

best_parameter = max(without_normalization,key=without_normalization.get)
best_acc = without_normalization[best_parameter]

print(f"Best Accuracy: {best_acc:.2f}%")
print(f"Best Hyperparameters (Batch, Lambda, LR): {best_parameter}")
```



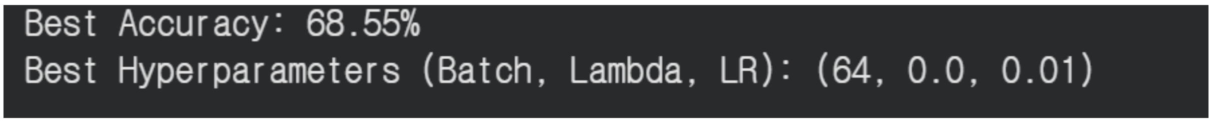
```
Best Accuracy: 56.55%
Best Hyperparameters (Batch, Lambda, LR): (64, 0.0, 0.01)
```

My faulty model didn't even get past 60% of the maximum performance.



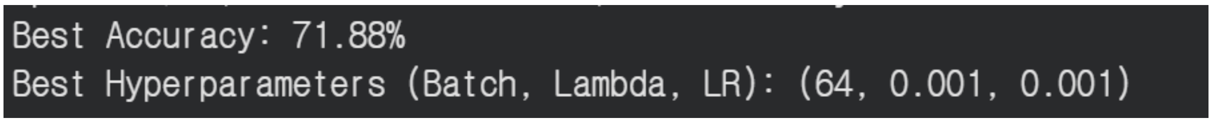
```
skeleton_random = nn.Linear(input_size,output_size)
# there is no with torch.no_grad()
result_loss, result_val = train_evaluate_svm(model=skeleton_random, svm_train_dataset=SVM_TRAIN, svm_val_dataset=SVM_VAL,
                          loss_fn=Hinge_Loss, learning_rate=lr, lambda_param = L,
```

The performance of the model with θ initialized at a neutral random location was as follows.



```
Best Accuracy: 68.55%
Best Hyperparameters (Batch, Lambda, LR): (64, 0.0, 0.01)
```

θ initialized at a neutral random location + Normalization



```
Best Accuracy: 71.88%
Best Hyperparameters (Batch, Lambda, LR): (64, 0.001, 0.001)
```

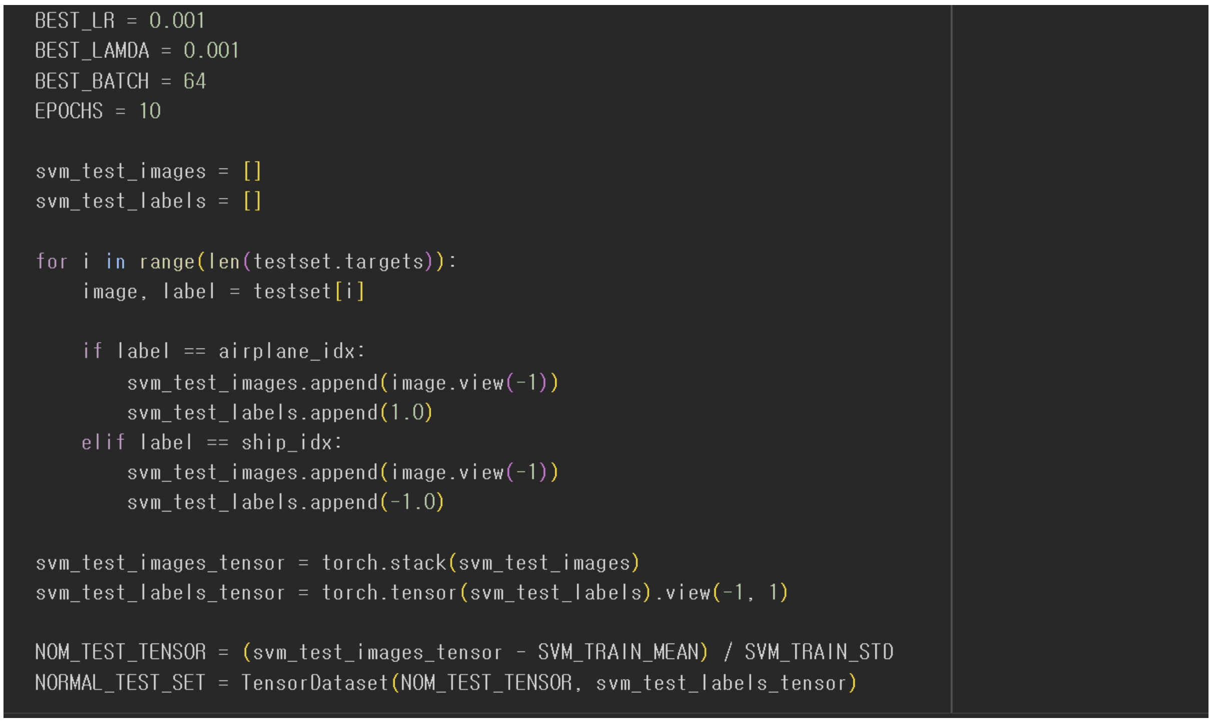
Interestingly, my method and the unnormalized method yielded good results at (64, 0.0, 0.01), but after normalization, the best result was at (64, 0.001, 0.001).

The very act of picking two random points is a profoundly flawed approach (ignoring the entire plane and the entire ship).

The best result was at (64, 0.0, 0.01), but after normalization, the best result was at (64, 0.001, 0.001). This is because normalization forces the data to fall within a certain distribution.

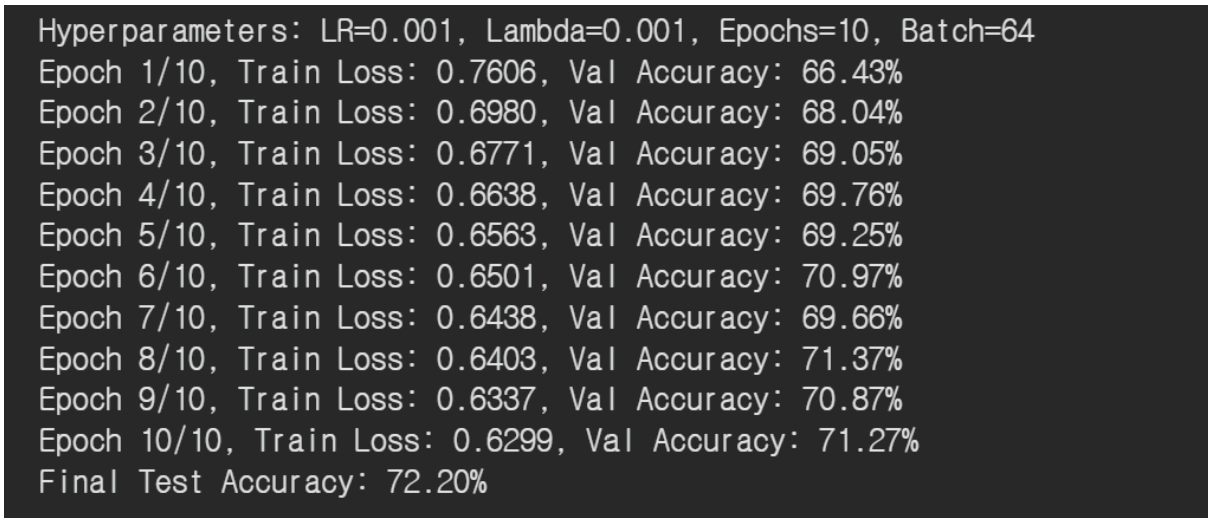
We were able to observe the effect of regularization, which was not properly observed in

1 - j) What is the final test accuracy?
The final test was conducted with a model that was randomly initialized and then generalized, with a parameter combination of (64, 0.001, 0.001).



```
BEST_LR = 0.001
BEST_LAMDA = 0.001
BEST_BATCH = 64
EPOCHS = 10

svm_test_images = []
svm_test_labels = []

for i in range(len(testset.targets)):
    image, label = testset[i]

    if label == airplane_idx:
        svm_test_images.append(image.view(-1))
        svm_test_labels.append(1.0)
    elif label == ship_idx:
        svm_test_images.append(image.view(-1))
        svm_test_labels.append(-1.0)

svm_test_images_tensor = torch.stack(svm_test_images)
svm_test_labels_tensor = torch.tensor(svm_test_labels).view(-1, 1)

NOM_TEST_TENSOR = (svm_test_images_tensor - SVM_TRAIN_MEAN) / SVM_TRAIN_STD
NORMAL_TEST_SET = TensorDataset(NOM_TEST_TENSOR, svm_test_labels_tensor)
```

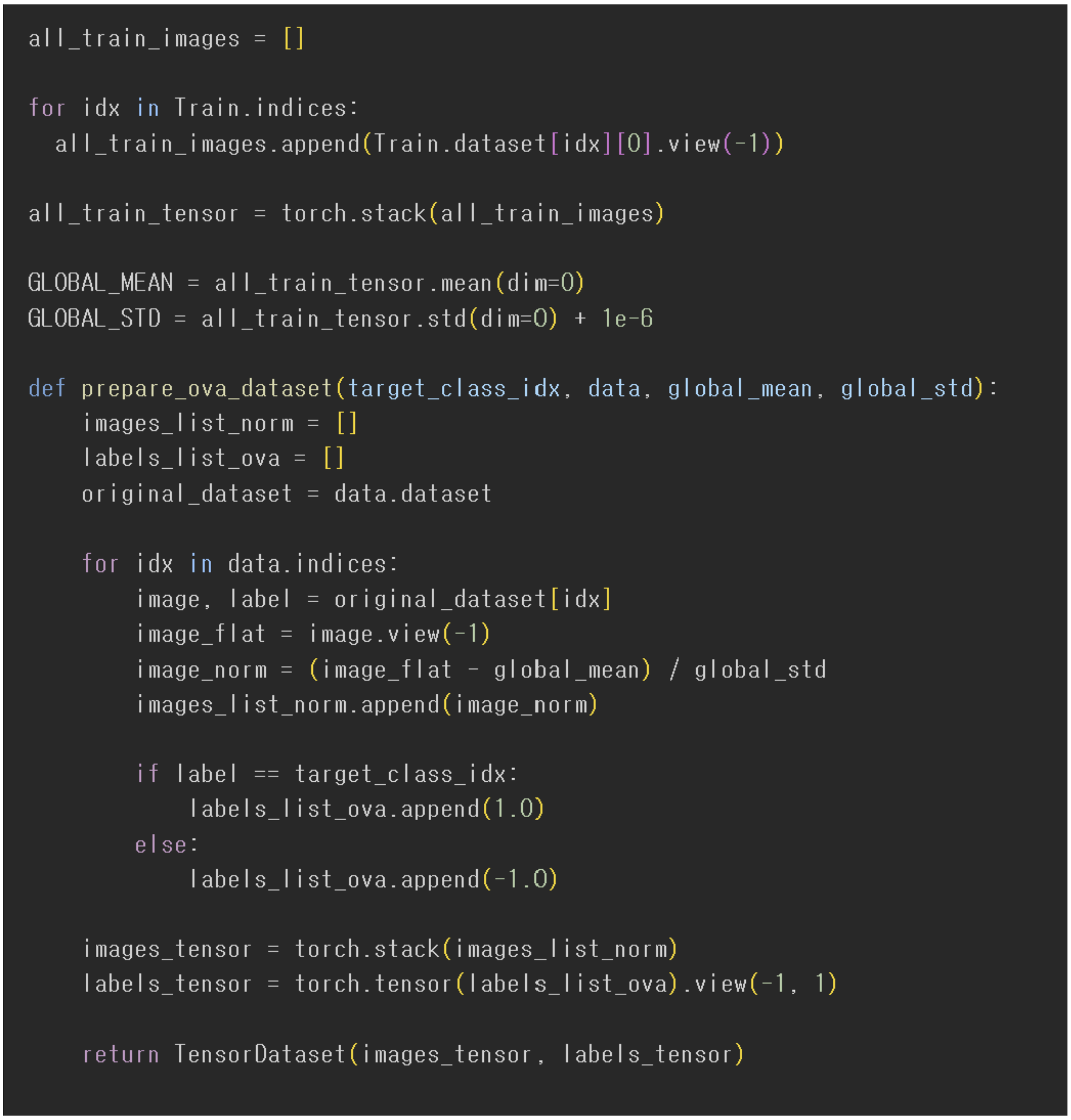
After selecting only the data of airplane and ship, normalize it by the mean and standard deviation of the train to ensure i.i.d.



```
Hyperparameters: LR=0.001, Lambda=0.001, Epochs=10, Batch=64
Epoch 1/10, Train Loss: 0.7606, Val Accuracy: 66.43%
Epoch 2/10, Train Loss: 0.6980, Val Accuracy: 68.04%
Epoch 3/10, Train Loss: 0.6771, Val Accuracy: 69.05%
Epoch 4/10, Train Loss: 0.6638, Val Accuracy: 69.76%
Epoch 5/10, Train Loss: 0.6563, Val Accuracy: 69.25%
Epoch 6/10, Train Loss: 0.6501, Val Accuracy: 70.97%
Epoch 7/10, Train Loss: 0.6438, Val Accuracy: 69.66%
Epoch 8/10, Train Loss: 0.6403, Val Accuracy: 71.37%
Epoch 9/10, Train Loss: 0.6337, Val Accuracy: 70.87%
Epoch 10/10, Train Loss: 0.6299, Val Accuracy: 71.27%
Final Test Accuracy: 72.20%
```

This experiment attempted to initialize the weights by setting the geometric relationship between the two samples, but found that this was a biased starting point that overfitted a small number of data.
After changing to standard random initialization, the performance was improved, and after stabilizing learning through data normalization and performing hyperparameter tuning again, the model reached a final performance of 72.20%.
In conclusion, the dramatic performance contrast with the failed first attempt clearly demonstrates how essential unbiased initialization, stable data scale (normalization), and optimal hyperparameters are for model performance.

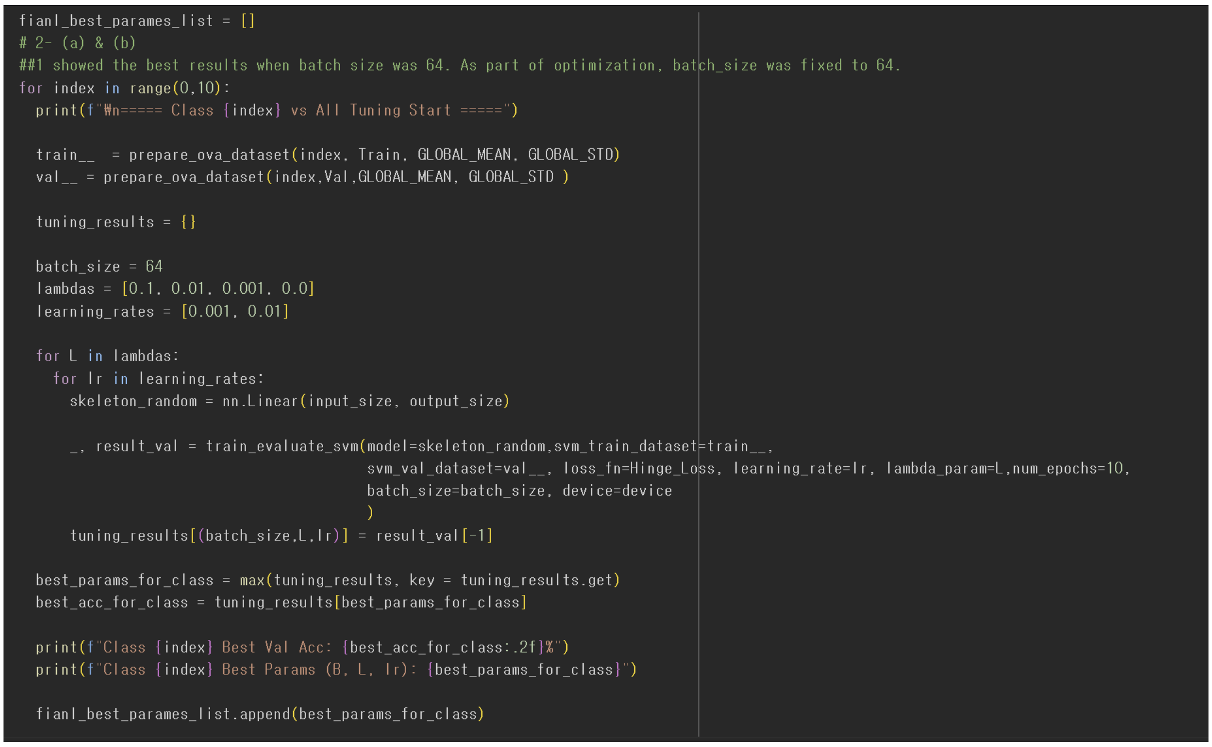
Problem #2: Multiclass Classification via multiple one-vs-all soft-margin SVM on CIFAR10



```python
all_train_images = []

for idx in Train.indices:
  all_train_images.append(Train.dataset[idx][0].view(-1))

all_train_tensor = torch.stack(all_train_images)

GLOBAL_MEAN = all_train_tensor.mean(dim=0)
GLOBAL_STD = all_train_tensor.std(dim=0) + 1e-6

def prepare_ova_dataset(target_class_idx, data, global_mean, global_std):
    images_list_norm = []
    labels_list_ova = []
    original_dataset = data.dataset

    for idx in data.indices:
        image, label = original_dataset[idx]
        image_flat = image.view(-1)
        image_norm = (image_flat - global_mean) / global_std
        images_list_norm.append(image_norm)

        if label == target_class_idx:
            labels_list_ova.append(1.0)
        else:
            labels_list_ova.append(-1.0)

    images_tensor = torch.stack(images_list_norm)
    labels_tensor = torch.tensor(labels_list_ova).view(-1, 1)

    return TensorDataset(images_tensor, labels_tensor)
```

Since I confirmed that normalization performs well in step 1,
I first perform normalization and then define a function that classifies a specific idx and the remainder.

2 - a) Perform multiclass classification using multiple one-vs-all soft-margin SVM, using your SVM implementation.
2 - b) Perform hyperparameter search for each SVM separately.

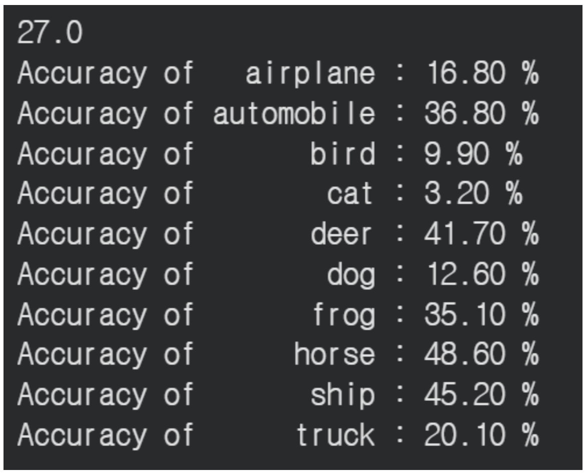


```python
fianl_best_parames_list = []
# 2- (a) & (b)
##1 showed the best results when batch size was 64. As part of optimization, batch_size was fixed to 64.
for index in range(0,10):
  print(f"₩n===== Class {index} vs All Tuning Start =====")

  train__  = prepare_ova_dataset(index, Train, GLOBAL_MEAN, GLOBAL_STD)
  val__ = prepare_ova_dataset(index,Val,GLOBAL_MEAN, GLOBAL_STD )

  tuning_results = {}

  batch_size = 64
  lambdas = [0.1, 0.01, 0.001, 0.0]
  learning_rates = [0.001, 0.01]

  for L in lambdas:
    for lr in learning_rates:
      skeleton_random = nn.Linear(input_size, output_size)

      _, result_val = train_evaluate_svm(model=skeleton_random,svm_train_dataset=train__,
                                         svm_val_dataset=val__, loss_fn=Hinge_Loss, learning_rate=lr, lambda_param=L,num_epochs=10,
                                         batch_size=batch_size, device=device
                                         )
      tuning_results[(batch_size,L,lr)] = result_val[-1]

  best_params_for_class = max(tuning_results, key = tuning_results.get)
  best_acc_for_class = tuning_results[best_params_for_class]

  print(f"Class {index} Best Val Acc: {best_acc_for_class:.2f}%")
  print(f"Class {index} Best Params (B, L, lr): {best_params_for_class}")

  fianl_best_parames_list.append(best_params_for_class)
```

Since the basic iteration is 10 times (the number of classes), I thought it would take a long time if I ran too many nested loops, so I fixed the batch_size to 64 and the epochs to 10. This is because when solving **Problem #1**, the best batch_size came out to be 64, and the epochs were fixed to 10 before tuning the hyperparameters.

```
Class 0 Best Val Acc: 90.26%
Class 0 Best Params (B, L, lr): (64, 0.0, 0.001)
Class 1 Best Val Acc: 90.54%
Class 1 Best Params (B, L, lr): (64, 0.0, 0.01)
Class 2 Best Val Acc: 89.36%
Class 2 Best Params (B, L, lr): (64, 0.1, 0.001)
Class 3 Best Val Acc: 90.58%
Class 3 Best Params (B, L, lr): (64, 0.1, 0.001)
Class 4 Best Val Acc: 90.58%
Class 4 Best Params (B, L, lr): (64, 0.1, 0.001)
Class 5 Best Val Acc: 89.72%
Class 5 Best Params (B, L, lr): (64, 0.1, 0.001)
Class 6 Best Val Acc: 89.86%
Class 6 Best Params (B, L, lr): (64, 0.1, 0.001)
Class 7 Best Val Acc: 90.76%
Class 7 Best Params (B, L, lr): (64, 0.01, 0.01)
Class 8 Best Val Acc: 89.92%
Class 8 Best Params (B, L, lr): (64, 0.1, 0.001)
Class 9 Best Val Acc: 90.14%
Class 9 Best Params (B, L, lr): (64, 0.0, 0.01)
```

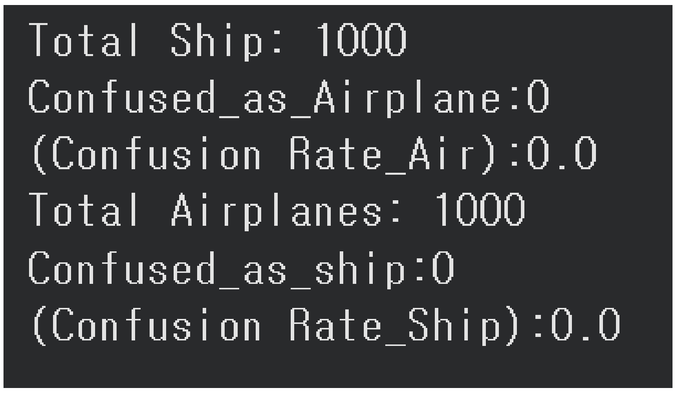
Animal classes suppressed model complexity (overfitting) by applying strong regularization with a high lambda coefficient of 0.1, and non-animal classes showed high Val Acc even without strong regularization.



```
27.0
Accuracy of   airplane : 16.80 %
Accuracy of automobile : 36.80 %
Accuracy of       bird : 9.90 %
Accuracy of        cat : 3.20 %
Accuracy of       deer : 41.70 %
Accuracy of        dog : 12.60 %
Accuracy of       frog : 35.10 %
Accuracy of      horse : 48.60 %
Accuracy of       ship : 45.20 %
Accuracy of      truck : 20.10 %
```

I had high expectations for the test because the accuracy on the validation set was very high, but I was a little surprised because it was so low. The results for the test set were significantly worse than for the test set, but there was a pitfall in the accuracy of val. Since each class was trained on 1 vs. the remaining 9, even if it was determined that the class corresponding to 1 was not present, an accuracy close to 90% was achieved due to the characteristics of the dataset. However, there was no significant correlation between the hyperparameter tuning results (animals are subject to relatively large restrictions) and the actual test results. This is because the best results were achieved by horses, which received relatively small restrictions among the animals.

Additionally, when solving Problem #1 earlier, I targeted ships and airplanes, which have a high probability of having similar background colors, as 1vs1 targets. I was curious how a model trained on airplanes with 1vs all would recognize ships, and how a model trained on ships with 1vs all would recognize airplanes, so I ran it.



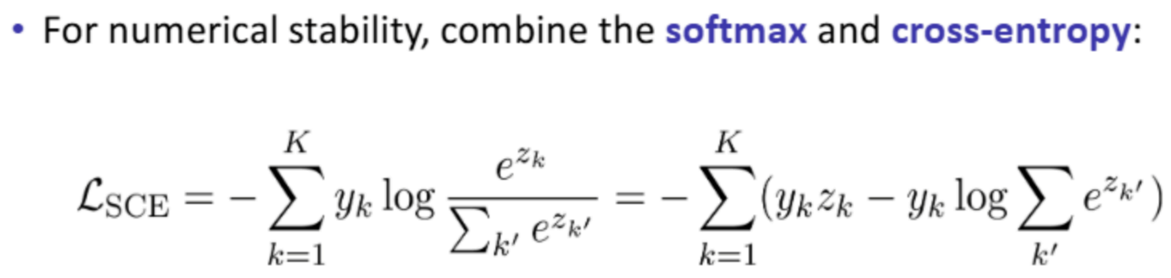
```
Total Ship: 1000
Confused_as_Airplane:0
(Confusion Rate_Air):0.0
Total Airplanes: 1000
Confused_as_ship:0
(Confusion Rate_Ship):0.0
```

Just like the testset, the accuracy is 90% even if the model simply sets it as 'not the target class', so the model will not mistake other classes with similar colors as the target class.

Additionally, for the 1-vs-1 SVM (nn.Linear) experiment, we confirmed that the similar background colors (sky/sea) shared by 'airplane' and 'ship' acted as a major noise factor that significantly hindered the perfect classification of the two classes, intertwined with the inherent limitations of the linear model.

Problem #3: Multiclass Classification via multinomial/multiclass logistic regression on CIFAR10

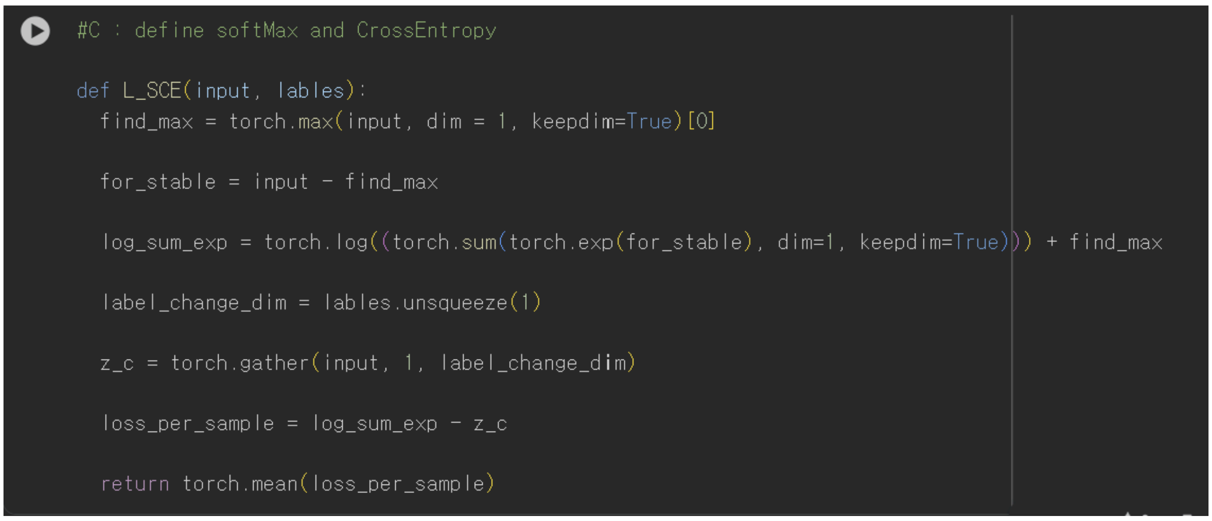
3 - a) Implement your own multinomial/multiclass logistic regression from scratch. Do not use activation functions from PyTorch libraries. Perform multiclass classification.



- For numerical stability, combine the **softmax** and **cross-entropy**:

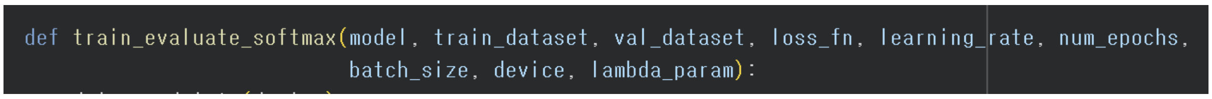
$$\mathcal{L}_{\text{SCE}} = -\sum_{k=1}^{K} y_k \log \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}} = -\sum_{k=1}^{K} \left(y_k z_k - y_k \log \sum_{k'} e^{z_{k'}}\right)$$

Rather than defining Softmax and CEE separately, we efficiently defined L_SCE, which combines Softmax (multinomial/multiclass logistic regression) and CEE, by referring to the lecture material (Lecture 8 Multiclass Classification).



```
#C : define softMax and CrossEntropy

def L_SCE(input, lables):
  find_max = torch.max(input, dim = 1, keepdim=True)[0]

  for_stable = input - find_max

  log_sum_exp = torch.log((torch.sum(torch.exp(for_stable), dim=1, keepdim=True))) + find_max

  label_change_dim = lables.unsqueeze(1)

  z_c = torch.gather(input, 1, label_change_dim)

  loss_per_sample = log_sum_exp - z_c

  return torch.mean(loss_per_sample)
```

I didn't quite understand the process of defining a function, where instead of using the input as is, we use the value obtained by subtracting the input from the maximum value in the input. However, since exp itself increases significantly even slightly after x>0, the function value increases significantly to prevent overflow, so it is said that the range of x entering exp is set to x<=0 to prevent overflow. This was added for stability to prevent disconnection during hyperparameter tuning after the function definition was completed. In fact, there may be errors caused by the longer decimal point even when x<0, but it is said that this method is used because it is better than overflow. After correcting like this, find_max is added again when calculating log_sum_exp.

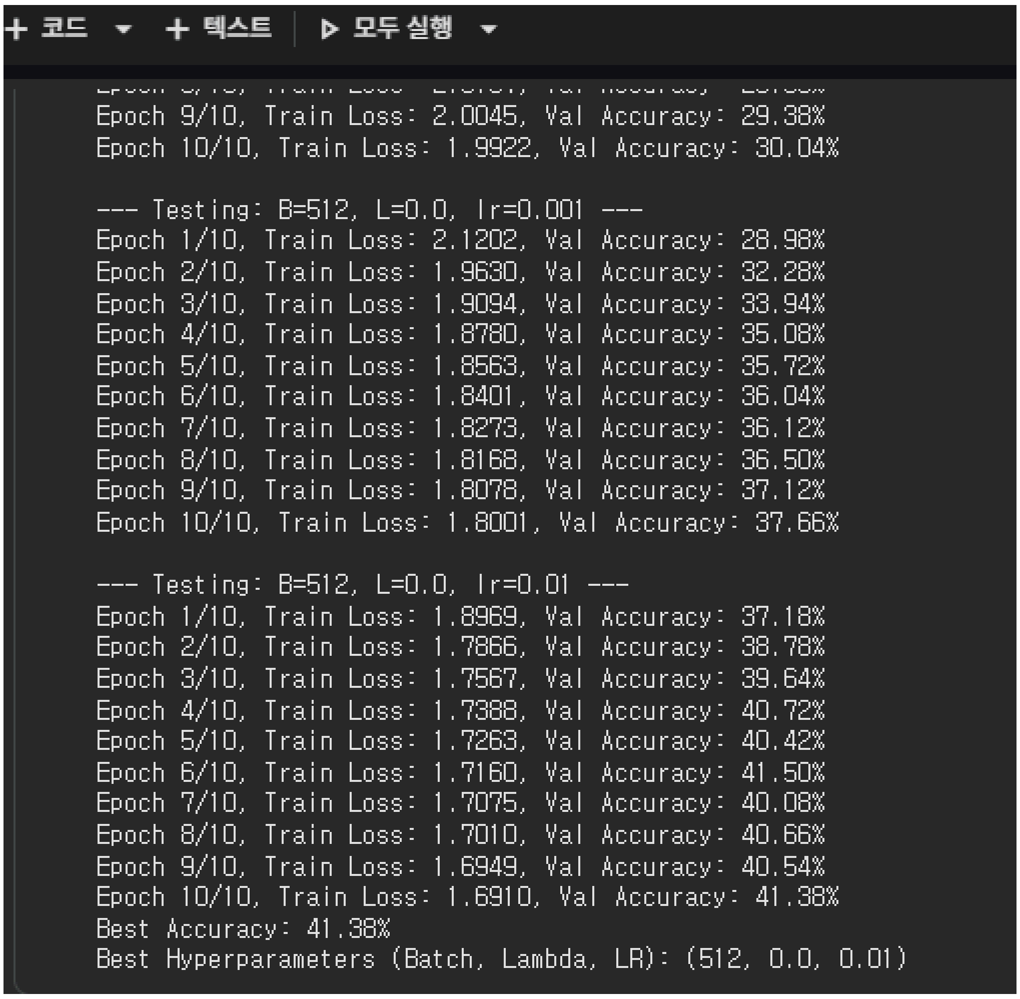


```
def train_evaluate_softmax(model, train_dataset, val_dataset, loss_fn, learning_rate, num_epochs,
                           batch_size, device, lambda_param):
```

The train function was adjusted to suit the purpose of multi-class.

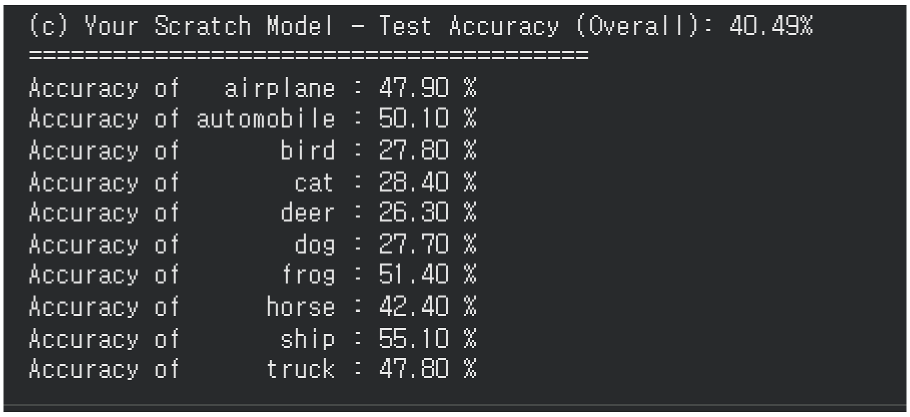
3 - b) Perform hyperparameter search

Hyperparameter tuning was performed under the following conditions.
batch_sizes = [64,128,256,512]
lambdas = [0.1, 0.01, 0.001, 0.0]
learning_rates = [0.0001, 0.001, 0.01]



```
    Epoch 9/10, Train Loss: 2.0045, Val Accuracy: 29.38%
    Epoch 10/10, Train Loss: 1.9922, Val Accuracy: 30.04%

    --- Testing: B=512, L=0.0, lr=0.001 ---
    Epoch 1/10, Train Loss: 2.1202, Val Accuracy: 28.98%
    Epoch 2/10, Train Loss: 1.9630, Val Accuracy: 32.28%
    Epoch 3/10, Train Loss: 1.9094, Val Accuracy: 33.94%
    Epoch 4/10, Train Loss: 1.8780, Val Accuracy: 35.08%
    Epoch 5/10, Train Loss: 1.8563, Val Accuracy: 35.72%
    Epoch 6/10, Train Loss: 1.8401, Val Accuracy: 36.04%
    Epoch 7/10, Train Loss: 1.8273, Val Accuracy: 36.12%
    Epoch 8/10, Train Loss: 1.8168, Val Accuracy: 36.50%
    Epoch 9/10, Train Loss: 1.8078, Val Accuracy: 37.12%
    Epoch 10/10, Train Loss: 1.8001, Val Accuracy: 37.66%

    --- Testing: B=512, L=0.0, lr=0.01 ---
    Epoch 1/10, Train Loss: 1.8969, Val Accuracy: 37.18%
    Epoch 2/10, Train Loss: 1.7866, Val Accuracy: 38.78%
    Epoch 3/10, Train Loss: 1.7567, Val Accuracy: 39.64%
    Epoch 4/10, Train Loss: 1.7388, Val Accuracy: 40.72%
    Epoch 5/10, Train Loss: 1.7263, Val Accuracy: 40.42%
    Epoch 6/10, Train Loss: 1.7160, Val Accuracy: 41.50%
    Epoch 7/10, Train Loss: 1.7075, Val Accuracy: 40.08%
    Epoch 8/10, Train Loss: 1.7010, Val Accuracy: 40.66%
    Epoch 9/10, Train Loss: 1.6949, Val Accuracy: 40.54%
    Epoch 10/10, Train Loss: 1.6910, Val Accuracy: 41.38%
    Best Accuracy: 41.38%
    Best Hyperparameters (Batch, Lambda, LR): (512, 0.0, 0.01)
```

3-c) What is the final test accuracy?

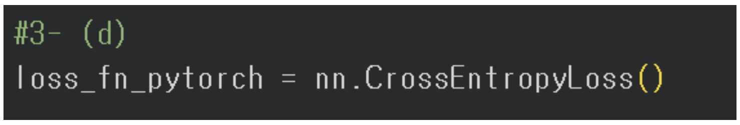


```
(c) Your Scratch Model - Test Accuracy (Overall): 40.49%
========================================
Accuracy of   airplane : 47.90 %
Accuracy of automobile : 50.10 %
Accuracy of       bird : 27.80 %
Accuracy of        cat : 28.40 %
Accuracy of       deer : 26.30 %
Accuracy of        dog : 27.70 %
Accuracy of       frog : 51.40 %
Accuracy of      horse : 42.40 %
Accuracy of       ship : 55.10 %
Accuracy of      truck : 47.80 %
```

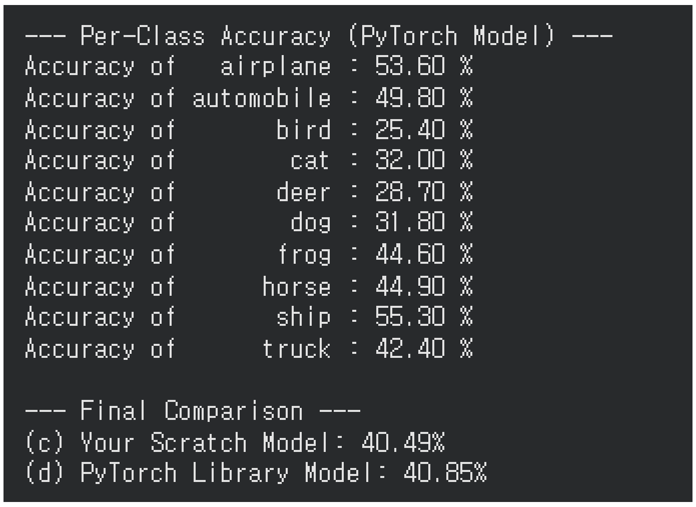
This is the result of calculating the softmax values for all 10 testset images, then classifying the largest value as the testset image's class and measuring its accuracy. Accuracy for bird and cat, which previously suffered in the 1 vs. All test, has significantly improved, and classes that previously recorded accuracy in the 20% range have all increased to the

40-50% range. However, there are classes, such as deer and horse, where accuracy has actually declined.

3-d) Compare against a model implemented using functions from PyTorch libraries



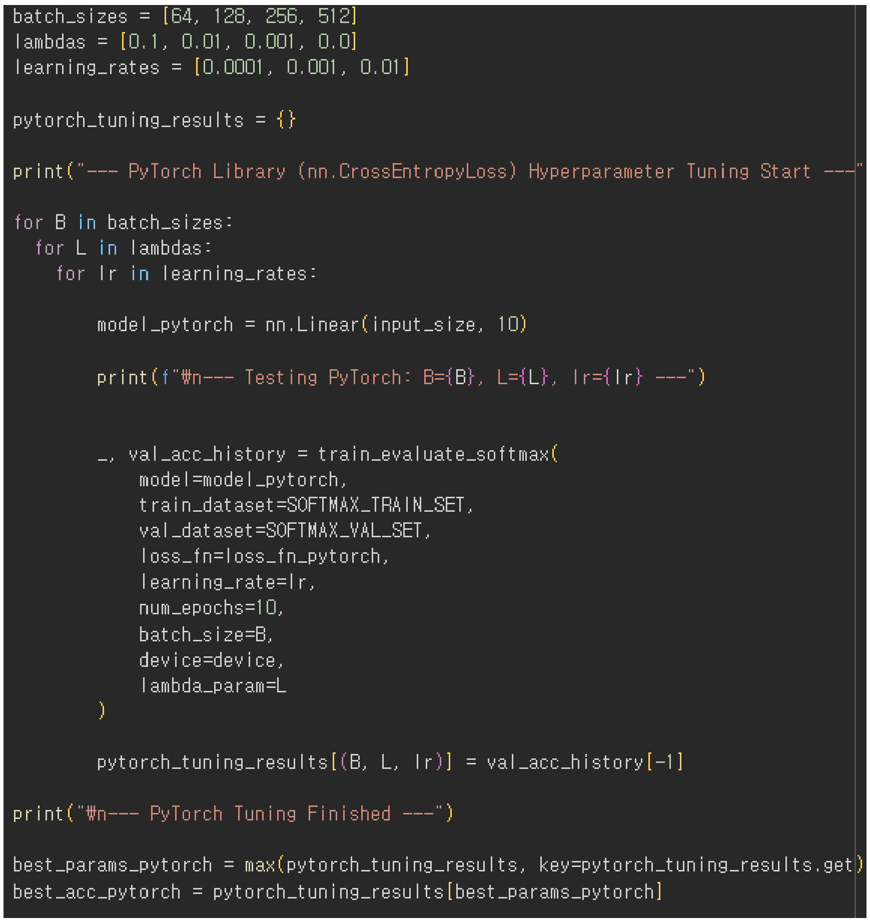
```
#3- (d)
loss_fn_pytorch = nn.CrossEntropyLoss()
```



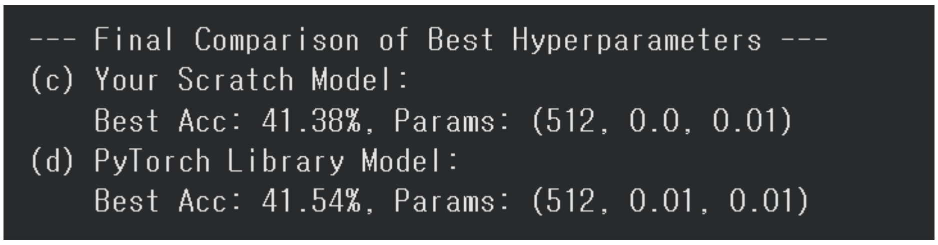
```
--- Per-Class Accuracy (PyTorch Model) ---
Accuracy of   airplane : 53.60 %
Accuracy of automobile : 49.80 %
Accuracy of       bird : 25.40 %
Accuracy of        cat : 32.00 %
Accuracy of       deer : 28.70 %
Accuracy of        dog : 31.80 %
Accuracy of       frog : 44.60 %
Accuracy of      horse : 44.90 %
Accuracy of       ship : 55.30 %
Accuracy of      truck : 42.40 %

--- Final Comparison ---
(c) Your Scratch Model: 40.49%
(d) PyTorch Library Model: 40.85%
```

After changing only the loss function to the PyTorch library's L_SCE and training with the optimal combination of hyperparameter tuning of the L_SCE I created above, I conducted a test. The overall accuracy was similar to that of the L_SCE I created, and similar to the results of the L_SCE I created, it failed to distinguish between birds, cats, and deer.

Finally, to verify that nn.CrossEntropyLoss and L_SCE are truly similar, we performed hyperparameter tuning on nn.CrossEntropyLoss and compared the optimal combination to see if it was similar to L_SCE and whether the final accuracy was similar.



```python
batch_sizes = [64, 128, 256, 512]
lambdas = [0.1, 0.01, 0.001, 0.0]
learning_rates = [0.0001, 0.001, 0.01]

pytorch_tuning_results = {}

print("--- PyTorch Library (nn.CrossEntropyLoss) Hyperparameter Tuning Start ---"

for B in batch_sizes:
  for L in lambdas:
    for lr in learning_rates:

        model_pytorch = nn.Linear(input_size, 10)

        print(f"₩n--- Testing PyTorch: B={B}, L={L}, lr={lr} ---")


        _, val_acc_history = train_evaluate_softmax(
            model=model_pytorch,
            train_dataset=SOFTMAX_TRAIN_SET,
            val_dataset=SOFTMAX_VAL_SET,
            loss_fn=loss_fn_pytorch,
            learning_rate=lr,
            num_epochs=10,
            batch_size=B,
            device=device,
            lambda_param=L
        )

        pytorch_tuning_results[(B, L, lr)] = val_acc_history[-1]

print("₩n--- PyTorch Tuning Finished ---")

best_params_pytorch = max(pytorch_tuning_results, key=pytorch_tuning_results.get)
best_acc_pytorch = pytorch_tuning_results[best_params_pytorch]
```



```
--- Final Comparison of Best Hyperparameters ---
(c) Your Scratch Model:
    Best Acc: 41.38%, Params: (512, 0.0, 0.01)
(d) PyTorch Library Model:
    Best Acc: 41.54%, Params: (512, 0.01, 0.01)
```

Unfortunately, there are some differences, which are said to be due to errors that occur during the process of iterating through the loop for hyperparameter tuning.

Through this experiment, I implemented an SVM using Torch, visualized data more complex than MNIST, and learned about preprocessing, which transforms the data into a form suitable for model learning. The pseudocode for hyperparameter tuning learned in class was simple, but when actually implementing it, there were more considerations than expected (including considering image dimensions and rearranging column order).
It was surprising to discover that initializing the initial value of θ in an SVM by randomly selecting two points between the two classes for a 1vs1 comparison could actually be the main culprit in lowering model performance. Furthermore, I learned that simply unfolding the image information and drawing a hyperplane based on it to classify the two classes has limitations. Furthermore, in 1vsall comparisons, I learned that high validation accuracy does

not necessarily indicate the general performance of the model. I found that if the ratio of the target to the rest is too skewed to one side, the model can actually become stupid.