# 國立清華大學

# 計算機視覺

# Computer Vision

## Homework 3

系所級:電子所二年級

學號:111063548

姓名:蕭方凱

指導老師:孫民教授

# 目錄

# Problems

## 1. Finish the rest of the codes for Problem 1 and Problem 2 according to the hint. (2 code cells in total.)

(1) Change the output of the model to 10 class:

The dataset used in this assignment is CIFAR10, which contains 60,000 images and is divided into 10 classes. This code converts the output classes of the original resnet18 model and resnet50 model from the default 1000 to 10 to meet the question requirements.

The specific method is as shown below. **Use the torch.nn.Linear syntax** to convert the output classes into 10, and print the results to confirm that the output classes have been successfully converted to 10.

```
# HINT: Remember to change the model to 'resnet50' and the weights to weights="IMAGENET1K_V1" when needed.
small_model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18', weights = None)
big_model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50', weights = None)

small_model.fc = torch.nn.Linear(small_model.fc.in_features, 10)  # model.fc.in_features = 512，替換新的全連接層，從1000個classes變成10個
big_model.fc = torch.nn.Linear(big_model.fc.in_features, 10)  # model.fc.in_features = 512，替換新的全連接層，從1000個classes變成10個
num_classes1 = big_model.fc.out_features
print("output of the big model is", num_classes1)
num_classes2 = small_model.fc.out_features
print("output of the small model is", num_classes2)


# Background: The original resnet18 is designed for ImageNet dataset to predict 1000 classes.
# TODO: Change the output of the model to 10 class.
```

✓ 4.9s

```
Downloading: "https://github.com/pytorch/vision/zipball/v0.10.0" to C:\Users\zeus9/.cache\torch\hub\v0.10.0.zip
output of the big model is 10
output of the small model is 10
Using cache found in C:\Users\zeus9/.cache\torch\hub\pytorch_vision_v0.10.0
```

Fig 1 Change the number of output classes

The two variables of the torch.nn.Linear syntax are the number of input features and the number of output features.

small_model.fc.in_features and big_model.fc.in_features represent the number of input features of the small model and big model fully connected layers respectively. The former is 512. The latter is 2048. The number of input features will affect the training results. More complex models usually require more datasets to avoid generalization. And 10 is the number of output features, which is output classes.

## (2) Training and Testing Models

```python
# TODO: Fill in the code cell according to the pytorch tutorial we gave.
loss_fn = nn.CrossEntropyLoss()
small_model_optimizer = torch.optim.Adam(small_model.parameters(), lr=1e-3)
big_model_optimizer = torch.optim.Adam(big_model.parameters(), lr=1e-3)

def train(dataloader, model, loss_fn, optimizer):
  num_batches = len(dataloader)
  # print("batches:", num_batches)
  # print(dataloader)
  size = len(dataloader.dataset)
  # print("size:", size)
  # print(dataloader.dataset)
  epoch_loss = 0
  correct = 0

  model.train()

  for X, Y in tqdm(dataloader):   # tqdm:進度條
    # X, Y = X.to(device), Y.to(device)

    # Compute prediction error
    pred = model(X)
    loss = loss_fn(pred, Y)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    epoch_loss += loss.item()
    pred = pred.argmax(dim = 1, keepdim = True)
    correct += pred.eq(Y.view_as(pred)).sum().item()

  avg_epoch_loss = epoch_loss / num_batches
  avg_acc = correct / size
  return avg_epoch_loss, avg_acc
```

Fig 2 Training function

This program is a training function. X and Y are input features and output categories respectively. The training steps are as follows:

1. **Pred = model(X):** apply the model to get the predicted value, but this is not yet a label (1~10)
2. **loss = loss_fn(pred, Y):** calculate the error between the predicted value pred and the actual value Y
3. **optimizer.zero_grad(), loss.backward(), optimizer.step():** zero the gradient, calculate the gradient, and update parameters to prevent distortion caused by accumulation of gradient parameters.
4. **epoch_loss += loss.item():** accumulate the loss value of each batch
5. **pred = pred.argmax(dim = 1, keepdim = True):** convert predicted values into class labels
6. **correct += pred.eq(Y.view_as(pred)).sum().item():** calculate the number of samples for which the prediction is correct
7. return avg_loss, avg_accuracy

```
def test(dataloader, model, loss_fn):
    num_batches = len(dataloader)
    size = len(dataloader.dataset)
    epoch_loss = 0
    correct = 0

    model.eval()

    with torch.no_grad():
        for X, Y in tqdm(dataloader):
            # X, Y = X.to(device), Y.to(device)

            pred = model(X)

            epoch_loss += loss_fn(pred, Y).item()
            pred = pred.argmax(dim = 1, keepdim = True)
            correct += pred.eq(Y.view_as(pred)).sum().item()

    avg_epoch_loss = epoch_loss / num_batches
    avg_acc = correct / size
    return avg_epoch_loss, avg_acc
```

Fig 3 Testing Function

This program is a testing function. X and Y are input features and output categories respectively. The steps are as follows:

1. **model.eval()**: The model is set to evaluation mode, which disables some behaviors used during training, such as dropout. Ensure that the behavior of the model at test time is consistent with training
2. **Pred = model(X):** apply the model to get the predicted value, but this is not yet a label (1~10)
3. **epoch_loss += loss_fn(pred, Y).item():** accumulate the loss value of each batch
4. **pred = pred.argmax(dim = 1, keepdim = True):** convert predicted values into class labels
5. **correct += pred.eq(Y.view_as(pred)).sum().item():** calculate the number of samples for which the prediction is correct
6. return avg_loss, avg_accuracy

## 2. Train small model (resnet18) and big model (resnet50) from scratch on ‘sixteenth train dataloader’, half train dataloader’, and ‘train dataloader’ respectively.

(1) Reset function

```
# 跑for迴圈前須重置，避免累積accuracy，導致結果失真
def reset(weight_select):
  set_all_seed(123)
  small_model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18', weights = weight_select)
  big_model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50', weights = weight_select)
  small_model.fc = torch.nn.Linear(small_model.fc.in_features, 10)  # model.fc.in_features = 512，替換新的全連接層，從1000個classes變成10個
  big_model.fc = torch.nn.Linear(big_model.fc.in_features, 10)  # model.fc.in_features = 512，替換新的全連接層，從1000個classes變成10個
  # small_model_optimizer = torch.optim.Adam(small_model.parameters(), lr=1e-3)
  # big_model_optimizer = torch.optim.Adam(big_model.parameters(), lr=1e-3)
  small_model_optimizer = torch.optim.SGD(small_model.parameters(), lr=1e-3)
  big_model_optimizer = torch.optim.SGD(big_model.parameters(), lr=1e-3)
  return small_model, small_model_optimizer, big_model, big_model_optimizer
```

Fig 4 reset code

This program is to prevent the results of the previous size from affecting the next size before training dataloaders of different sizes. For example, if you train for train_dataloader (full size) first and then train for half_traindataloader, the accuracy of half_traindataloader will be higher than the accuracy of training for it from the beginning, so it needs to be reset before training with different sizes.

In the reset function, there are two possibilities for the input weight_select, which are None and IMAGENET1K_V1. These two weights will be used later to train models of different sizes and different amounts of data.。Inside the function, define small model (resnet18), big_model (resnet50) respectively, and two models with different optimizers, and return small_model, big_model, optimizer for later training use.

(2) Train small model (resnet18)

```
epochs = 5
x_train_dataloader = 1.0
x_half_train_dataloader = 0.5
x_sixteenth_train_dataloader = 1/16
data_size = np.array([x_sixteenth_train_dataloader, x_half_train_dataloader, x_train_dataloader])

###############Small Model#################
print("Starting Small Model Training\n")

small_model, small_model_optimizer, big_model, big_model_optimizer = reset(None)

print("==========sixteenth_train_dataloader==========\n")
for epoch in range(epochs):
  small_model_train_loss, small_model_train_acc = train(sixteenth_train_dataloader, small_model, loss_fn, small_model_optimizer)
  small_model_test_loss, small_model_test_acc = test(valid_dataloader, small_model, loss_fn)
  print(f"Epoch {epoch+1:2d}:Loss = {small_model_train_loss:.4f} Acc = {small_model_train_acc:.2f} Test_Loss = {small_model_test_loss:.4f} Test_Acc = {small_model_test_acc:.2f}")
  if (epoch == epochs-1):
    y_small_model_sixteenth_train = small_model_train_acc
    y_small_model_sixteenth_test = small_model_test_acc
print("Done!")
```

Fig 5 Training sixteenth train dataloader

```
small_model, small_model_optimizer, big_model, big_model_optimizer = reset(None)

print("==========half_train_dataloader==========\n")
for epoch in range(epochs):
  small_model_train_loss, small_model_train_acc = train(half_train_dataloader, small_model, loss_fn, small_model_optimizer)
  small_model_test_loss, small_model_test_acc = test(valid_dataloader, small_model, loss_fn)
  print(f"Epoch {epoch+1:2d}:Loss = {small_model_train_loss:.4f} Acc = {small_model_train_acc:.2f} Test_Loss = {small_model_test_loss:.4f} Test_Acc = {small_model_test_acc:.2f}")
  if (epoch == epochs-1):
    y_small_model_half_train = small_model_train_acc
    y_small_model_half_test = small_model_test_acc
print("Done!")
```

Fig 6 Training half train dataloader

```
small_model, small_model_optimizer, big_model, big_model_optimizer = reset(None)

print("==========train_dataloader==========\n")
for epoch in range(epochs):
  small_model_train_loss, small_model_train_acc = train(train_dataloader, small_model, loss_fn, small_model_optimizer)
  small_model_test_loss, small_model_test_acc = test(valid_dataloader, small_model, loss_fn)
  print(f"Epoch {epoch+1:2d}:Loss = {small_model_train_loss:.4f} Acc = {small_model_train_acc:.2f} Test_Loss = {small_model_test_loss:.4f} Test_Acc = {small_model_test_acc:.2f}")
  if (epoch == epochs-1):
    y_small_model_train = small_model_train_acc
    y_small_model_test = small_model_test_acc
print("Done!")
```

Fig 7 Training train dataloader

These three pieces of code train data of different sizes respectively. From top to bottom, they are sixth train dataloader, half train dataloader, and train dataloader. The corresponding data quantities are 3125, 25000, and 50000 photos. The Batch size is preset to 256, which means that each batch will use 256 photos as input data to be input to the model for training.

```
==========sixteenth_train_dataloader==========

100%|██████████| 13/13 [00:20<00:00,  1.56s/it]
100%|██████████| 40/40 [00:07<00:00,  5.68it/s]
Epoch  1:Loss = 2.4940 Acc = 0.01 Test_Loss = 2.3142 Test_Acc = 0.10
  0%|          | 0/13 [00:01<?, ?it/s]
```

The progress bar above is 13, it is because the number of data in the sixth train dataloader is 3125, and 256 images are input in one batch. A total of batches that needed to complete the training is $3125/256 = 13$.

(3) Train big model (resnet50)

```
print("Starting Big Model Training\n")

small_model, small_model_optimizer, big_model, big_model_optimizer = reset(None)

print("==========sixteenth_train_dataloader==========\n")
for epoch in range(epochs):
  big_model_train_loss, big_model_train_acc = train(sixteenth_train_dataloader, big_model, loss_fn, big_model_optimizer)
  big_model_test_loss, big_model_test_acc = test(valid_dataloader, big_model, loss_fn)
  print(f"Epoch {epoch+1:2d}:Loss = {big_model_train_loss:.4f} Acc = {big_model_train_acc:.2f} Test_Loss = {big_model_test_loss:.4f} Test_Acc = {big_model_test_acc:.2f}")
  if (epoch == epochs-1):
    y_big_model_sixteenth_train = big_model_train_acc
    y_big_model_sixteenth_test = big_model_test_acc
print("Done!")
```

Fig 8 Training sixteenth train dataloader

```
small_model, small_model_optimizer, big_model, big_model_optimizer = reset(None)

print("==========half_train_dataloader==========\n")
for epoch in range(epochs):
  big_model_train_loss, big_model_train_acc = train(half_train_dataloader, big_model, loss_fn, big_model_optimizer)
  big_model_test_loss, big_model_test_acc = test(valid_dataloader, big_model, loss_fn)
  print(f"Epoch {epoch+1:2d}:Loss = {big_model_train_loss:.4f} Acc = {big_model_train_acc:.2f} Test_Loss = {big_model_test_loss:.4f} Test_Acc = {big_model_test_acc:.2f}")
  if (epoch == epochs-1):
    y_big_model_half_train = big_model_train_acc
    y_big_model_half_test = big_model_test_acc
print("Done!")
```

Fig 9 Training half train dataloader

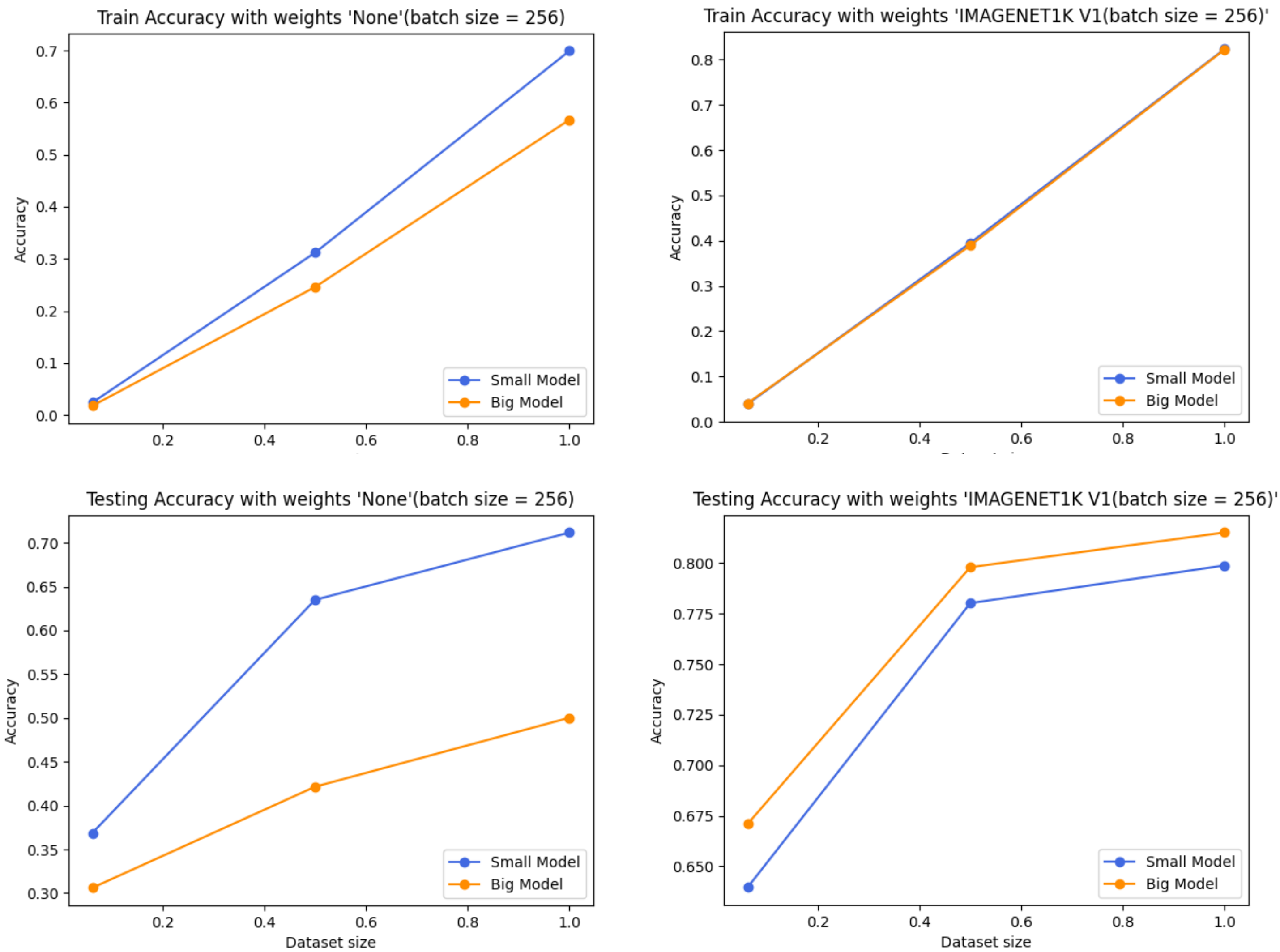
```
small_model, small_model_optimizer, big_model, big_model_optimizer = reset(None)

print("==========train_dataloader==========\n")
for epoch in range(epochs):
  big_model_train_loss, big_model_train_acc = train(train_dataloader, big_model, loss_fn, big_model_optimizer)
  big_model_test_loss, big_model_test_acc = test(valid_dataloader, big_model, loss_fn)
  print(f"Epoch {epoch+1:2d}:Loss = {big_model_train_loss:.4f} Acc = {big_model_train_acc:.2f} Test_Loss = {big_model_test_loss:.4f} Test_Acc = {big_model_test_acc:.2f}")
  if (epoch == epochs-1):
    y_big_model_train = big_model_train_acc
    y_big_model_test = big_model_test_acc
print("Done!")
```

Fig 10 Training train dataloader

These three pieces of code train data of different sizes respectively. From top to bottom, they are sixth train dataloader, half train dataloader, and train dataloader. The corresponding data quantities are 3125, 25000, and 50000 photos. The Batch size is preset to 256, which means that each batch will use 256 photos as input data to be input to the model for training.

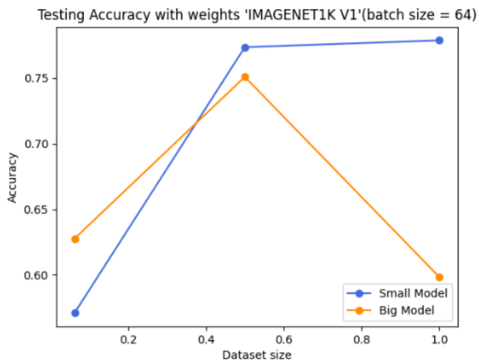
(4) The results of different model, different weights, different data size

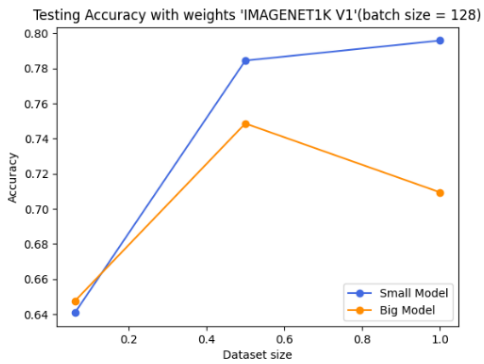


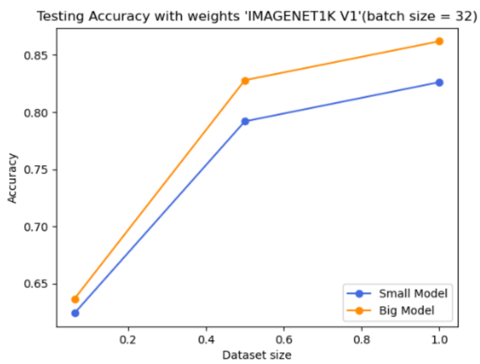

The picture above is the results of training accuracy and testing accuracy for three data sizes, applying weight to None and IMAGENET1K_V1. It can be observed that whether IMAGENET1K_V1 is applied has a great impact on the big model. Why there is such an impact will be discussed later.

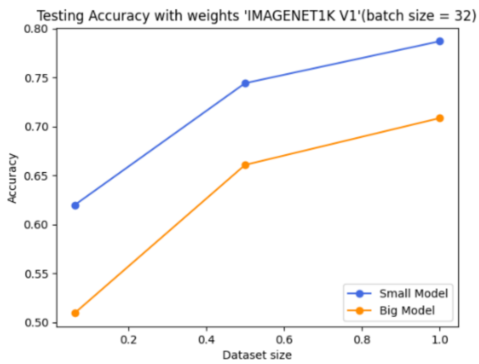
## 3. Achieve the best performance given all training data using whatever model and training strategy.

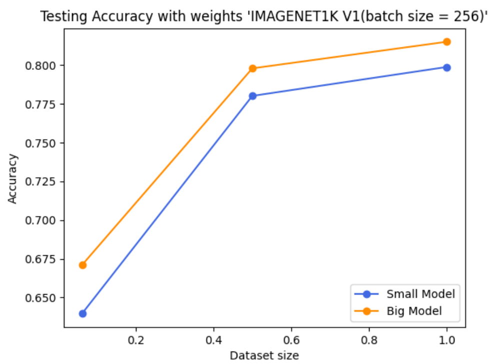
For problem1 and problem2, Adam optimizer with a learning rate of 0.001 was used. Attempts were made to adjust some hyperparameters such as optimizer, learning rate, batch size, etc. Through practical testing, it was found that the performance of Adam is superior to SGD as an optimizer. Therefore, the focus of this assignment was placed on adjusting the batch size and learning rate. The following are the different result graphs corresponding to various batch sizes:



The results indicate that there is overfitting when the batch size is increased from 64 to 128 in the training dataloader. However, with a batch size of 32, the performance is promising. This suggests that a smaller batch size is not necessarily better, as a batch size of 256 performs better than 64 or 128. Nevertheless, theoretically, reducing the batch size still has some positive effects.

| **Small Model Accuracy** | | | |
|---|---|---|---|
| | **Batch size = 32 (lr=1e-3)** | **Batch size = 32 (lr=1e-4)** | **Batch size=256** |
| **Sixteenth dataloader** | 62% | 62.4% | 64% |
| **half dataloader** | 74.4% | 79.2% | 78% |
| **full dataloader** | 78.7% | 82.6% | 79.9% |
| **Big Model Accuracy** | | | |
| | **Batch size = 32 (lr=1e-3)** | **Batch size = 32 (lr=1e-4)** | **Batch size=256** |
| **Sixteenth dataloader** | 50.9% | 63.7% | 67.1% |
| **half dataloader** | 66.1% | 82.8% | 79.8% |
| **full dataloader** | 70.1% | 86.2% | 81.5% |

The green table represents the optimal results, and specific adjustments were made to the following hyperparameters:

(1). batch size = 32

(2). learning rate = 0.0001

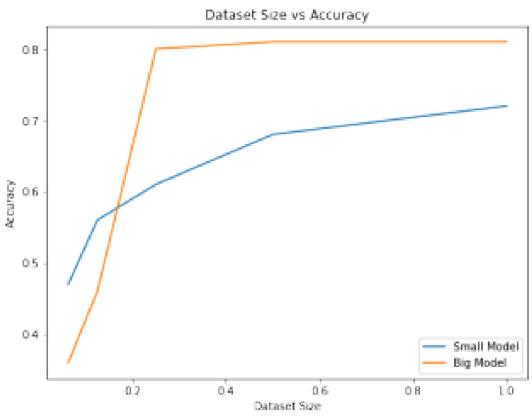
(3). nn.Sequential(nn.Dropout(0.2), nn.Linear(small_model.fc.in_features, 10)), use dropout method

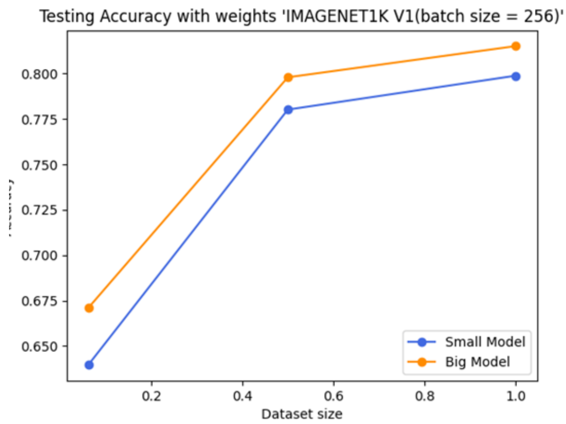
A smaller batch size may improve generalization ability, but may also lead to overfitting. Generalization is a measure of a model's ability to classify unseen data samples. A model has good generalization properties if it can predict data samples from different sets. Overfitting refers to a model that performs well during training but performs poorly on data sets other than the known data, the two(overfitting and generalization) are a bit like the concept of trade off.

A smaller learning rate facilitates the model in converging more easily to either the global minimum or a local minimum. A larger learning rate may lead to skipping optimal points during the optimization process, while a smaller learning rate helps in more accurately pinpointing the optimal points. However, too small of a learning rate can result in the model converging too slowly. Considering the training data, model architecture, and task type comprehensively, selecting an appropriate learning rate is a crucial step in training deep learning models.

## Discussion

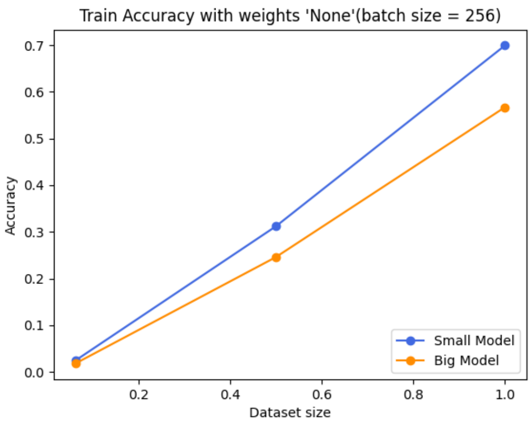
1. The relationship between the accuracy, model size, and the training dataset size. (Total 6 models. Small model trains on the sixteenth, half, and all data. Big model trains on the sixteenth, half, and all data. If the result is different from Fig. 1, please explain the possible reasons.)

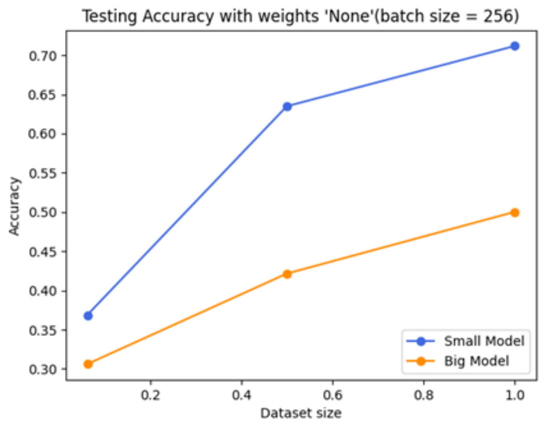



The picture on the left is Fig1 of the example, and the picture on the right is the result of self-training. The two images differ in several aspects. The first notable distinction is that in the left image, there is a significant increase in accuracy between a dataset size of 0.2 and 0.4, while in the right image, the accuracy starts to improve around a dataset size of 0.5 (half train dataloader), reaching approximately 0.8. This suggests that the left image provides a more detailed differentiation of dataset sizes, whereas the right image only includes three dataset size categories. Additionally, it appears that there is no data for dataset sizes around 0.2 in the right image, possibly due to the absence of a dataset with approximately 10,000 photos, which could explain the lack of a discernible accuracy trend at that point.

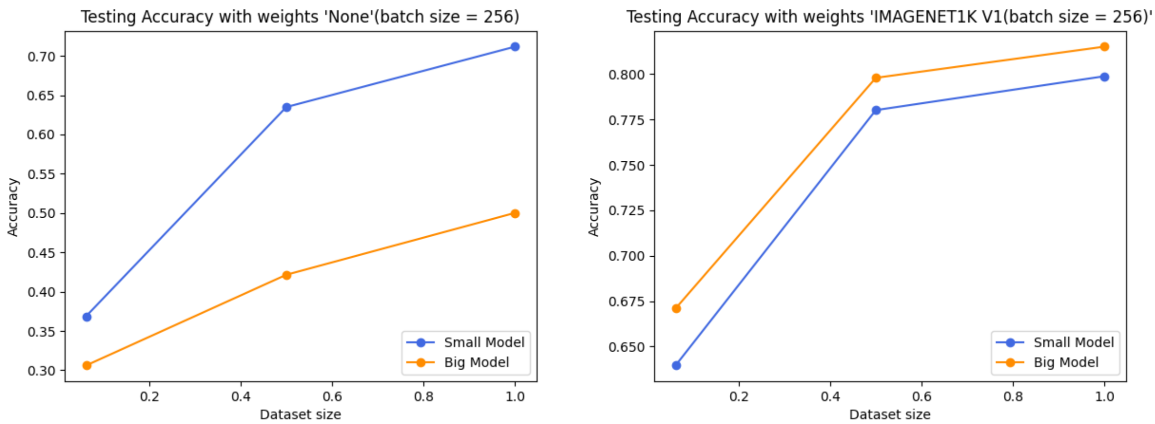
The second difference lies in the performance around a dataset size of 0.2. In this range, the right image exhibits better accuracy compared to the left image. The accuracy in the right image is approximately 65%, while the left image shows less than 50% accuracy. This discrepancy could be attributed to the model using default weights from IMAGENET1K_V1 during translation.






Because the model is more complex, Big model sometimes leads to overfitting. The two figures above both show the case of “weights = None”. The orange line is the big model. It can be seen that the dataset size is 1, which means when train_dataloader is used as input (total 30,000 pictures), the training accuracy is much different from the testing accuracy. This is a typical situation of overfitting. One of the solutions is dropout, this method can randomly turn off a certain proportion of neurons in the NN layer, thereby reducing the model's impact on each neuronal dependency.

2. What if we train the ResNet with ImageNet initialized weights (weights="IMAGENET1K V1"). Please explain why the relationship changed this way?




The picture on the left shows the case where weights are preset to None, and the picture on the right shows IMAGENET1K_V1. The choice of weights plays a very important role in model training. For this operation, when the weight set at the beginning is close to the image category of CIFAR10, It will be more conducive to the model training process, so it is speculated that IMAGENET1K_V1 is a good preset weight.