Machine Learning Assignment Report

Name: Nuha Imran

Student ID: 20696366

Date of submission: 31/03/23

Contents: Page

Task 1: Choice of model and why……3

Task 1: Data Augmentation and why I did it……4

Task 1: What is fine tuning and how we do it……5

Task 1: Hyper-parameters and choice……6

Task 1: Hyper-parameters and choice……11

Task 1: Fine-tuning & Resnet result……12

Task 1: Scratch & Resnet50 scratch result……13

Task 1: Conclusion and discussion……14

Task 2: Resnet18……14

Task 2: Alexnet……15

Task 2: Choice of values for layers(Alexnet & Vgg16)……16

Task 2: Vgg16……17

Task 2: Results……18

Task 2: Conclusion and final discussion……19

Task 3: Feature Extraction code……20

Task 3: Results of simple model on feature extraction……21

Task 3: Final verdict and discussion……22

Implementation of codes……22

Image of all source codes……26

Machine Learning Report

Task 1:

I have attempted the full assignment nothing is missing from any of the tasks.

Note: Use the file I have provided to unzip or the code won't run as per it should

This is simply because when I did not put them in one folder the code gave me very low accuracy although the implementation was entirely correct and just to be on the safe side I tuned the hyper parameters too which made no significant change and later changed my training and testing function too which made no change either so I'm assuming it does not read classes because in one folder they have a different name and in the other its different.

Choice of Model: ResNet50

➔ Why:

1. It has a depth of 50 layers.
2. Deals with vanishing gradient problem. (Able to learn identity function)
3. Model can capture small features and details because of its depth.
4. Presence of skip connection.
5. Makes use of 3-layer bottleneck blocks to ensure improved accuracy and lesser training time.

This model was immensely successful, as can be ascertained from the fact that its ensemble won the top position at the ILSVRC 2015 classification competition with an error of only 3.57%. Additionally, it also came first in the ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation in the ILSVRC & COCO competitions of 2015.
source: https://viso.ai/deep-learning/resnet-residual-neural-network/

Overall, I personally think resnet50 would work best on coral image classification because of these attributes.

- Image size being chosen and why?

  The images provided were not of the same size, they are of various sizes. As shown below this makes it hard for the pre-trained model to process them

and takes more computational time as for every image the model needs to adjust accordingly making the processing time-consuming and we also end up with different-sized images in the recognition process.

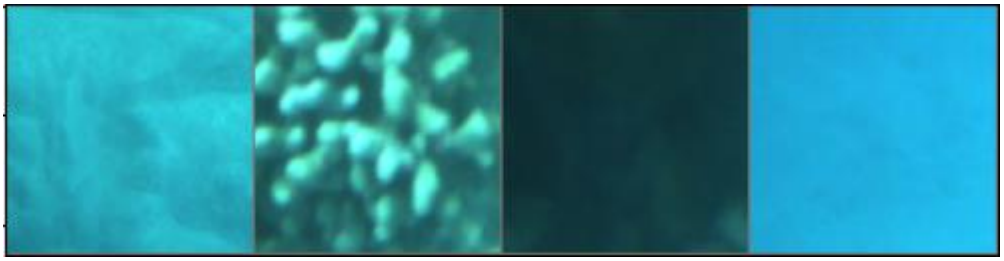
The image size was chosen as 224*224

1. Ensures better performance, captures the features better, and is less time-consuming as a model can process images faster.
2. Ensures consistency: all images are being trained to learn the exact same features.
3. Pre-trained models available in the torch vision library were trained on 224*224.

The code of line that does this transformation:

```
#crops image to a square size of 224*224
transforms.RandomResizedCrop(224),
```

What the images would look like after the resizing:



- <u>Data Augmentation: Why I did it?</u>

  Data Augmentation is done when the dataset is small. This is simply to make your model learn to generalize better on unseen data. For e.g if the dog is facing the right side in an image the model will mostly learn that only a dog facing the right side is a dog which is not true to fix this flip the image so the model can learn better. In the same way, we have a small dataset so we will

flip images because corals facing toward the right or left are still corals. Further, we rotate so it's recognized as the rotated dog is still a dog. In the same way, an upside-down coral is still a coral. Further, I normalized the pixels with a mean and standard deviation simply because various channels of the image have a various range of pixel values this causes issues with the model training and optimization, and I want to mitigate that.

The snippet of code that does all this:

```
[ ]  #the compose function applies transformations to each image in order
     train_transforms = transforms.Compose([
         #randomly rotates image by 30 degrees(for data augmentation purposes)
         transforms.RandomRotation(30),
         #crops image to a square size of 224*224
         transforms.RandomResizedCrop(224),
         #flips image horizontally again for augmentation purposes
         transforms.RandomHorizontalFlip(),
         #converts image to pytorch tensor
         transforms.ToTensor(),
         #normalising pixel of image
         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
     ])
```

- What is fine-tuning?

  Fine-tuning: Refers to a procedure where you take a trained model e.g a model that was trained before on a dataset and then train it further making changes that end up functioning well on your dataset or simply fitting your dataset. We update the weights of the pre-trained model based on the new dataset. This helps reduce the time and effort put in and helps in increasing efficiency and accuracy. Additionally, we also change the hyper parameters To meet our model's requirements.

- What we do:

  During training, we are fine-tuning the model by changing the parameters of the newly added layer. This is due to the new layer's randomly initialized weights, which must be updated to make the model more suitable for our specific classification task.

Where is it in the code:

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.resnet50(pretrained = True)
```

This snippet shows that the pre-trained weights are being used.

```
#replaces last fully connected layer with new layer that has same input
#features equal to number of classes in training data
mod.fc = nn.Linear(mod.fc.in_features, len(train_data.classes))
```

The ResNet50 model has been enhanced with a new linear layer with the same number of output classes as the custom coral image dataset. This is essential since the pre-trained ResNet50 model was initially trained on a different dataset with a different number of output classes. By replacing the old layer with a new layer that has the same number of output classes as the coral image dataset.

- Further completing fine-tuning hyper-parameters
  Since the data set is small, I'm starting with a batch size of 16 and epochs of 10(to prevent overfitting)

- Why only these batch sizes?(16 & 32)
  - These batch sizes 16 & 32 allow faster training overall and allow more frequent changes to model weight making it more accurate.
  - Less memory usage for small sizes this is better when you have less computational power, and my computer has low computational power.
  - Small batches tend to generalize better simply because they introduce more noise and randomness in the training process this then prevents overfitting.

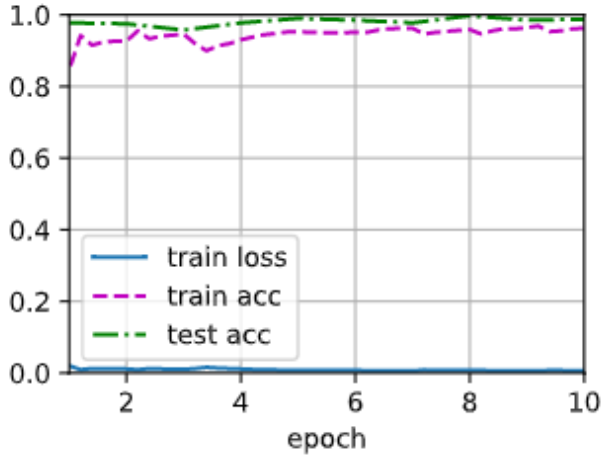
- Why only this many number of epochs? (10 & 20)
- small dataset is usually prone to overfitting when we go for more epochs the model starts memorizing the images from the datasets rather than learning the patterns so to avoid this and help it generalize on unseen data I've opted for a low number of epochs.
- With small datasets the patterns can be learned quickly and can reach optimal performance after a few epochs, this will help the model converge faster while saving time.

WARNING: Every time you see test accuracy in the graph its actually validation accuracy as mentioned in the book

- Tuning the Hyper parameters:
  Batch size = 16 Number of epochs = 10

```
# Train the model
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.006, train acc 0.962, test acc 0.988
94.3 examples/sec on [device(type='cuda', index=0)]
```



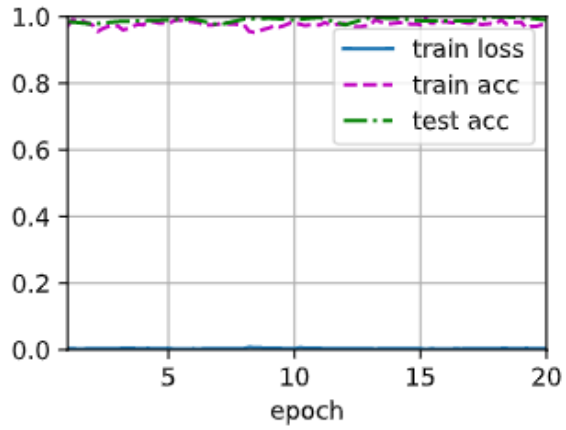

```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.99
```

- Batch size = 16 Number of epochs = 20

```
[48] # Train the model
     num_epochs = 20
     d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
     #d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.004, train acc 0.974, test acc 0.990
97.9 examples/sec on [device(type='cuda', index=0)]
```



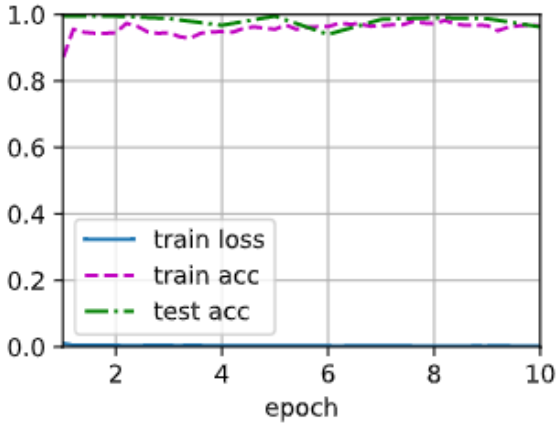

```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.985
```

- Batch size = 32 Number of epochs = 10

```
[38] # Train the model
     num_epochs = 10
     d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
     #d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.003, train acc 0.965, test acc 0.963
101.5 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.9775
```

- Batch size = 32 Number of epochs = 20

```
# Train the model
num_epochs = 20
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.001, train acc 0.981, test acc 0.990
101.7 examples/sec on [device(type='cuda', index=0)]
```


```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.9875
```

All these results are with ResNet50

Epochs	Batch Size	Train Accuracy	Validation Accuracy	Test Accuracy
10	16	96.2%	98.8%	99%
20	16	97.4%	99%	98.5%
10	32	96.5%	96.3%	97.75%
20	32	98.1%	99%	98.75%

16 batch size and 10 epochs give the best accuracy hence I'll continue with that throughout the training testing.

➔ <u>Why learning rate 0.001:</u>

- A very small learning rate can result in the model learning very slowly as it'll converge slowly and be time-consuming, making training time longer whereas a larger learning rate can cause the model's weights to diverge resulting in inaccurate results, small datasets are more prone to facing this

issue as models have less example to learn from by choosing 0.001 it's the best of both worlds where the model will neither converge(make it time consume) nor diverge(making it inaccurate)

➔ Why weight decay as 0.001?

- Small datasets are usually prone to overfitting. The model starts memorizing the training data instead of learning the patterns a smaller number like 0.001 can help regularize the model's weight to prevent overfitting.
- Weight decay helps the model to perform better in the testing stage because it reduces the model's sensitivity to small variations in training data and noise which helps specifically when the dataset is small
- Helps model to learn from existing data and generalise better on small datasets

➔ Why momentum as 0.9?

- Usually with a small dataset its easier to learn the patterns so a momentum of the value 0.9 will help the optimizer converge faster towards the optimal solution this will give good results and speed up the learning process
- Helps optimizer not get into the local minima, good for small datasets since fewer data to learn and sometimes have more noise.
- Can help optimizer lead to a better generalization performance as with small datasets sometimes overfitting become inevitable.

➔ Why SGD as an optimization Algorithm:

- It is computationally effective.
- Although prone to overfitting with small datasets like coral with the right regularization techniques like weight decay it becomes a good choice which is evident in my code.
- Reduces effects of noise.

➔ Why choose cross entropy as a loss function:

- Effective in measuring the difference between true class labels and predicted probabilities.
- Penalizes model for making incorrect predictions.
- Good for multi-class classification.
- Provides smooth gradient.

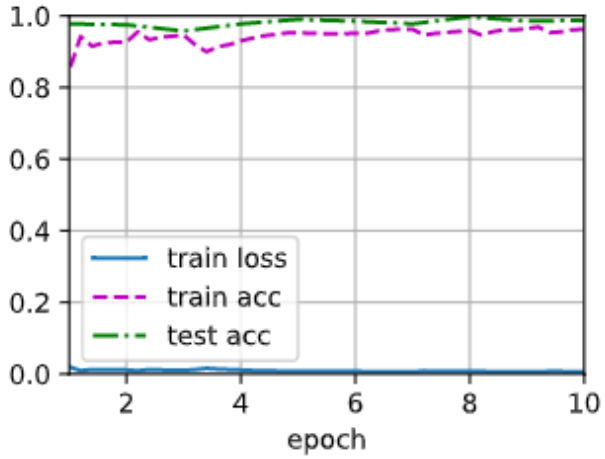
```
#defining the loss function and optimiser
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mod.parameters(), lr=0.001, weight_decay=0.001, momentum=0.9)
```

I did not tune these other hyperparameters as I am getting a good accuracy without tweaking them.

- Resnet50 as a model:
  ResNet50 is a 50-layer deep convolutional neural network design. Microsoft researchers developed it in 2015 as an answer to the issue of vanishing gradients in very deep neural networks. The introduction of residual connections, which allow the network to learn residual functions that describe the difference between a layer's input and output, is the main innovation of ResNet50. This allows the network to propagate gradients and reduces the vanishing gradient problem more effectively.
- Fine-tuning Resnet50 result:

```
# Train the model
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.006, train acc 0.962, test acc 0.988
94.3 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.99
```

The fine-tuned model shows very high accuracies. The training accuracy of 96.2%, validation accuracy of 98.8%, and testing accuracy of 99% show that the model is not overfitting, as performance on the testing set is not considerably greater than performance on the training set. This implies that the refined model is very accurate and capable of detecting meaningful patterns in the data.

➔ What is training from scratch?

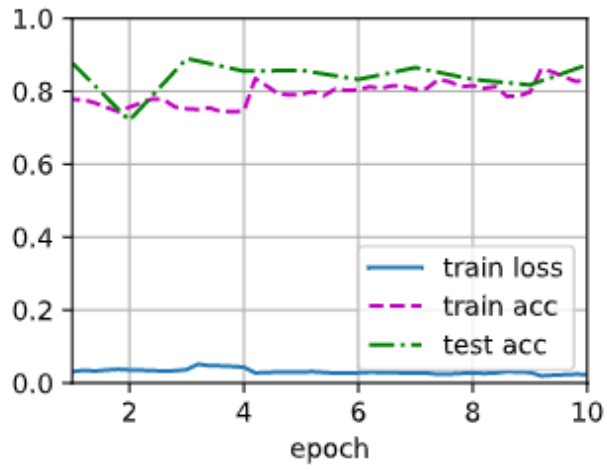
- Does not rely on weights of a pre-trained model.
- The weights are initialized randomly, and the model learns based on that.
- Hyperparameters are tuned.
  Part of the code that shows weights are initialized randomly.

```
rnd = torchvision.models.resnet50()
```

➔ Resnet50 Scratch result:

```
# training the model with the library
num_epochs = 10
#d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer1, num_epochs)
```

```
loss 0.024, train acc 0.831, test acc 0.873
99.0 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(rnd, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.895
```

The scratch model has 83.1% training accuracy, 87.3% validation accuracy, and 89.5% testing accuracy. While testing accuracy is quite good, the training accuracy is substantially lower, implying that the model may not be complex enough to capture all essential patterns in the data. This difference in training and testing accuracies is likewise not highly significant, indicating that the model is not overfitting.

- Comparison of performance:

Implementation	Training	Validation	Testing
Scratch	83.1%	87.3%	89.5%
Fine-tuning	96.2%	98.8%	99%

- Conclusion:
  With a 9.5% boost in accuracy, fine-tuning a pre-trained ResNet50 model on the Coral Net dataset performs better than training the model from scratch. This demonstrates the usefulness of transfer learning and fine-tuning for image classification problems, particularly when the dataset is small. The pre-trained had better accuracy because it had learned a lot of important features from the training it went through previously.

Task 2

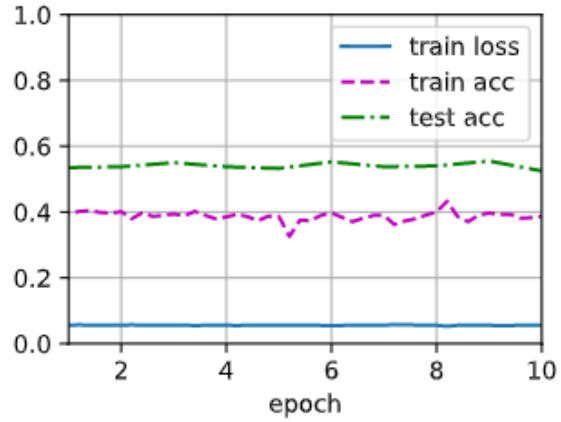
I have concluded from task 2 that fine-tuning is better so for the rest of the models, I'll use fine-tuning.

1. Resnet18

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.resnet18(pretrained = True)
```

```
# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.056, train acc 0.387, test acc 0.525
339.3 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.5225
```

The ResNet18 model is underfitting with a training accuracy of 38.7%, a validation accuracy of 52.5%, and a testing accuracy of 52.25%. The model's relatively low training accuracy suggests that it is failing to capture the underlying patterns in the data, resulting in poor performance on both the validation and testing sets.

2.Alexnet:

In alexnet, we cannot use in_features because the in_features parameter is already set to the correct value for the ImageNet dataset (which has images with size 224x224x3), and so we don't need to modify it when we fine-tune the model for a new task with the same input size. Therefore, we can simply replace the last linear layer of the classifier with a new linear layer that has the appropriate number of output units for our new task (in this case, 2), without changing the in_features parameter.
So I add this line of code there:

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.alexnet(pretrained = True)

#replacing last linear layer with 2 output units
#create new linear layer with 4096 input units and two outputs
mod.classifier[6] = nn.Linear(4096, 2)
#adding activation function with dimension of one
mod.classifier.add_module("7", nn.LogSoftmax(dim = 1))
```

What it does is modify the pre-trained model's classifier for the new task by replacing the last linear layer with a new linear layer that has 2 output units (corresponding to the number of classes in the new task) and adding a Log SoftMax activation function to the end of the classifier.

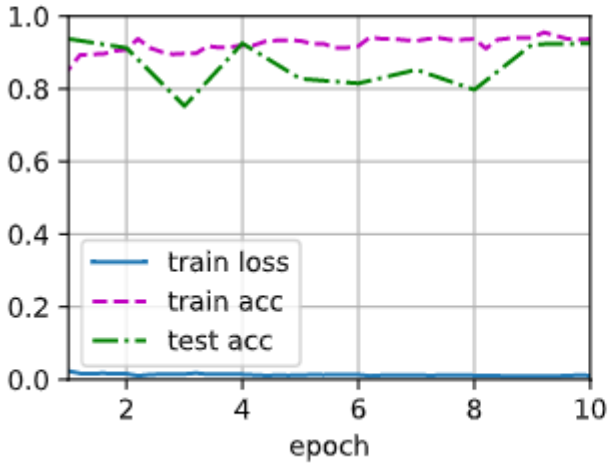
➔ Why these values?

The number of output units in the final linear layer of a neural network model's classifier should correspond to the number of classes in the classification task. Because the classification task contains two classes, the last linear layer is replaced with a new linear layer with two output units.

Because the pre-trained AlexNet model was trained on the ImageNet dataset, which contains 1000 classes, the original last linear layer had 1000 output units. As a result, the model's original classifier included a linear layer with 4096 input units and 1000 output units, followed by a LogSoftmax activation function to output the class probabilities. We want to adapt the pre-trained AlexNet model to a new binary classification task in this fine-tuning example, so we replace the last linear layer with a new linear layer with two output units and add a LogSoftmax activation function to the classifier's end to output the class probabilities for the new task.

```
# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.011, train acc 0.937, test acc 0.925
341.4 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

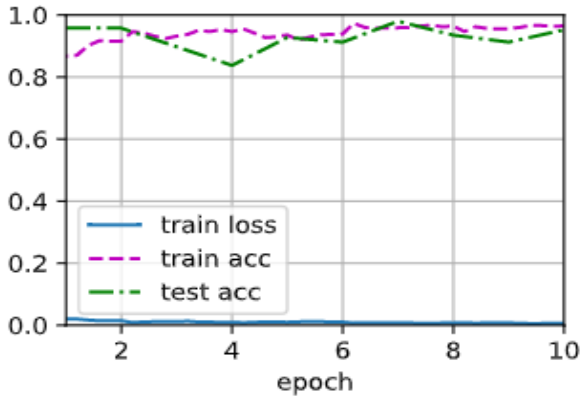
```
Testing accuracy: 0.945
```

Alexnet's training accuracy is 93.7%, validation accuracy is 92.5%, and testing accuracy is 94.5%, indicating that it is somewhat underfitting. While the testing accuracy is very good, the lower training accuracy suggests that the model may be too simple to capture all of the essential patterns in the data.

- Testing vgg16 finetuning:
  Vgg16 has the same snippet of code added that was added to alexnet because like alexnet is also does not have in_features and the justification and explanation would be the same.

```
# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.006, train acc 0.965, test acc 0.953
65.2 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.965
```

The Vgg16 model has 96.5% training accuracy, 95.3% validation accuracy, and 96.5% testing accuracy, indicating that it performs well without overfitting or underfitting.

Model	Train Accuracy	Validation Accuracy	Test Accuracy
ResNet50(fine-tuning)	96.2%	98.8%	99%
ResNet18(fine-tuning)	38.7%	52.5%	52.25%
Vgg16(fine-tuning)	96.5%	95.3%	96.5%
Alexnet(fine-tuning)	93.7%	92.5%	94.5%

Performance from best to worst with reasoning:

1)Resnet50

- Deep architecture: has 50 layers allowing it to learn complex features.
- Skip-connections efficiently deal with the vanishing gradient problem and allow the model to learn better
- Resnet50 learns features well from augmented data and our data has been augmented at the start of the code.

2)Vgg16: performed well but not as much as resnet50

- Is a deep network with 16 layers but not as deep as Resnet 50 with 50 layers.
- Does not have skip layers functionality hence fourth cannot be as good as resnet50.
- Has simpler architecture that helps training fast but this compromises learning complex features.

3)Alexnet: did well but not as well as resnet50 and vgg16

- Simply because alexnet has only 8 layers cannot understand as deeply as vgg16 or resnet50.
- Uses local response normalization in Convolutional layers this is less effective than batch normalization which has been proven by studies.
- Takes the lowest computational time but at the same time is less effective at understanding features of complex images.

4)ResNet18:

- Has a shallow architecture, has few layers, and cannot efficiently capture abstract features and work on complex datasets
- Has few skip connections and cannot learn as deeply as resnet50.
- Is more computationally effective this is where you compromise on performance and efficiency.

➔ <u>Conclusion (final discussion)</u>

The ResNet50 model surpasses the VGG16,Resnet18 and AlexNet models, improving accuracy by 2.5%,46.75% and 4.5%, respectively. This demonstrates ResNet50's ability in deep learning tasks, particularly when dealing with relatively small datasets like the CoralNet dataset.

The VGG16 model works reasonably well, however, it is still less accurate than ResNet50. VGG16 is a more detailed model than AlexNet and resnet18, but its design is simpler than ResNet 50's. This implies that the effectiveness of a neural network architecture is determined by the task and dataset.

The Alexnet models performance falls on the third number of the three models, with slightly lower accuracy than ResNet50. AlexNet was a breakthrough in deep learning when it was proposed in 2012, but its architecture is now considered relatively simplistic in comparison to more recent models like ResNet50 and VGG16.

The worse performance was given by Resnet18 although it has a complex architecture compared to Alexnet this helps us conclude that the models performance simply comes down to the dataset

Overall, the comparison of the three models demonstrates the significance of selecting the appropriate architecture for a given job and dataset. Fine-tuning pre-trained models such as ResNet50 and VGG16 can save time and money when compared to training models from scratch, but the pre-trained model used can have a substantial impact on the model's performance.

Task 3

Code extracting CNN features:

```
#defining a class that takes in from the nn.module and calling it CNN feature extractor
class CNNFeatureExtractor(nn.Module):
  #takes in an argument mod which in my case is a pretrained model resnet50
    def __init__(self, mod):
        super().__init__()
        #the nn.seq object is a container for the layers and uses list to get
        #all the pretrained layers and excludes out the last layer since thats what we need
        self.features = nn.Sequential(*list(mod.children())[:-1])
        #x is an input tensor
    def forward(self, x):
      #input tensor passed through convulutional layer
        x = self.features(x)
        #reshapes based on batch size and number ofc features
        x = x.view(x.size(0), -1)
        #returns the tensor
        return x
```

Code to store those features

```
mod.to(device)
mod.eval()
#creating to empty lists with these names
features_train = []
labels_train = []
#just no gradients being computed
with torch.no_grad():
    #iterating through the trainloader I made initially
    for images, labels in trainloader:
        #using cuda
        images = images.to(device)
        labels = labels.to(device)
        #passing the images  through mod so i can get features
        features = mod(images)
        #converting features from tensor to numpy arrays
        features_train.extend(features.cpu().numpy().astype(np.float32)) # cast to np.float32
        labels_train.extend(labels.cpu().numpy())
```

Training a simple model on those features

```
# training a simple model using extracted CNN features
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(features_train, labels_train)
```

➔ **Why I choose KNN:**

- Can handle non-linear data.
- Can extract high level features.
- Easy to implement.
- It is non-parametric.
- Good for small datasets as easier to keep all instances in memory and compute distance between them.

➔ **The accuracy I got:**

```
[18] # evaluating the performance on validation data
     train_accuracy = knn.score(features_train, labels_train)
     print('Training accuracy:', train_accuracy)

     Training accuracy: 0.9418084153983886
```

```
[19] # evaluating the performance on validation data
     val_accuracy = knn.score(features_val, labels_val)
     print('Validation accuracy:', val_accuracy)

     Validation accuracy: 0.9275
```

```
[20] # evaluating the performance on testing data
     test_accuracy = knn.score(features_test, labels_test)
     print('Testing accuracy:', test_accuracy)


     Testing accuracy: 0.92
```

KNN model has a training accuracy of 94.2%, a validation accuracy of 92.75%, and a testing accuracy of 92%. While the testing accuracy is very high, the lower training accuracy indicates that the model may not be capturing all of the significant patterns in the data, resulting in worse performance on both the validation and testing sets. This implies that the KNN model is slightly underfitting.

➔ Comparison with Resnet50-finetuning:

Model	Training Accuracy	Validation Accuracy	Testing accuracy
Resnet50(fine-tuning)	96.2%	98.8%	99%
KNN	94.2%	92.75%	92%

➔ Final Verdict:

We can compare the performance of the KNN trained on the extracted CNN features with the best model from task 2 which is the fine-tuned ResNet50 model. The ResNet50 model achieved an accuracy of 99% on the testing set in Task 2.

In our example, we trained a KNN on the extracted features and achieved an accuracy of 92% on the testing set. This shows that the features extracted from the last convolution layer of the ResNet50 model are informative and can be used to train a simple model to achieve good performance on the CoralNet dataset. However, the performance is not as good as the fine-tuned ResNet50 model, which indicates that the fine-tuned model can learn more discriminative features than a simple linear model based on the extracted CNN features.

➔ Implementation of code:

1)Make all necessary imports

```
import torch
import torchvision
from torchvision import transforms, datasets
from torch import nn
!pip install d2l
%pip install matplotlib-inline
from d2l import torch as d2l
```

2)Unzip the required file

```
!unzip /content/Training\ and\ Validation\ Data_2023_T1_COMP3010.zip
```

```
!unzip /content/Testing\ Data_2023.zip
```

3) Code defines three sets of transformations to be applied to the training, validation, and testing datasets, respectively.

```
#the compose function applies transformations to each image in order
train_transforms = transforms.Compose([
    #randomly rotates image by 30 degrees(for data augmentation purposes)
    transforms.RandomRotation(30),
    #crops image to a square size of 224*224
    transforms.RandomResizedCrop(224),
    #flips image horizontally again for augmentation purposes
    transforms.RandomHorizontalFlip(),
    #converts image to pytorch tensor
    transforms.ToTensor(),
    #normalising pixel of image
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

```
#doing the same to validation and testing data too
val_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

```
test_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

4) Code loads the data using the datasets.ImageFolder method, which assumes that the images are organized in subdirectories with each subdirectory representing a different class.

```
#getting the data from the path and subdirectories
data_dir = '/content/Training and Validation Data_2023_T1_COMP3010 (2)/Training and Validation Data_2023_T1_COMP3010/Data/coral image classification'
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
val_data = datasets.ImageFolder(data_dir + '/val', transform=val_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
```

5) The torch.utils.data.DataLoader method is used to create batches of data for training, validation, and testing.

```
#number of samples processed in each batch
batch_size = 16
#shuffle is for shuffling data before creating batches
#the train_data,val_data and test_data are datasets
#data loaders iterate over these datasets in batch sizes
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

These steps remain the same for every single model

6)choose a model from torch vision library set pretrained to true if that is the case if not keep the bracket empty()

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.resnet50(pretrained = True)
#this one is not pre-trained has randomised weights
rnd = torchvision.models.resnet50()
```

7)Replace last linear layer yourself for the models without in_features example alexnet and vgg16 for the rest use in_features to replace last linear layer also apply activation function in case of manual.

```
#features equal to number of classes in training data
mod.fc = nn.Linear(mod.fc.in_features, len(train_data.classes))

print(mod.fc.in_features)
print(len(train_data.classes))
rnd.fc = nn.Linear(rnd.fc.in_features, len(train_data.classes))
#checking if GPU is available
```

8) define the loss function (nn.CrossEntropyLoss()) and optimizer (torch.optim.SGD).

```
#defining the loss function and optimiser
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mod.parameters(), lr=0.001, weight_decay=0.001, momentum=0.9)
## optimizer1 = torch.optim.SGD(rnd.parameters(), lr=0.001, momentum=0.9)
```

9) d2l.train_ch13 method is used to train the model for a specified number of epochs (num_epochs) using the training and validation datasets.

```
# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

10) d2l.evaluate_accuracy_gpu method is used to evaluate the accuracy of the model on the testing dataset.

```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

This is the generalised code which will be followed throughout the only changes made will be for scratch and last layer and name of models.
Till step 7 the code is the same for KNN too after that
8)We extract features from the last layer

```
#defining a class that takes in from the nn.module and calling it CNN feature extractor
class CNNFeatureExtractor(nn.Module):
  #takes in an argument mod which in my case is a pretrained model resnet50
    def __init__(self, mod):
        super().__init__()
        #the nn.seq object is a container for the layers and uses list to get
        #all the pretrained layers and excludes out the last layer since thats what we need
        self.features = nn.Sequential(*list(mod.children())[:-1])
        #x is an input tensor
    def forward(self, x):
      #input tensor passed through convulutional layer
        x = self.features(x)
        #reshapes based on batch size and number ofc features
        x = x.view(x.size(0), -1)
        #returns the tensor
        return x
```

9)Store the testing, training and validating features:

```
mod.to(device)
mod.eval()
#creating to empty lists with these names
features_train = []
labels_train = []
#just no gradients being computed
with torch.no_grad():
    #iterating through the trainloader I made initially
    for images, labels in trainloader:
        #using cuda
        images = images.to(device)
        labels = labels.to(device)
        #passing the images  through mod so i can get features
        features = mod(images)
        #converting features from tensor to numpy arrays
        features_train.extend(features.cpu().numpy().astype(np.float32)) # cast to np.float32
        labels_train.extend(labels.cpu().numpy())
```

10)Train a model on your saved features

```
# training a simple model using extracted CNN features
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(features_train, labels_train)
```

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

11)evaluate its performance

```
# evaluating the performance on validation data
train_accuracy = knn.score(features_train, labels_train)
print('Training accuracy:', train_accuracy)
```

```
Training accuracy: 0.9418084153983886
```

```
# evaluating the performance on validation data
val_accuracy = knn.score(features_val, labels_val)
print('Validation accuracy:', val_accuracy)
```

```
Validation accuracy: 0.9275
```

```
# evaluating the performance on testing data
test_accuracy = knn.score(features_test, labels_test)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.92
```

CODE REFERENCES:

https://github.com/bentrevett/pytorch-image-classification/blob/master/3_alexnet.ipynb

https://github.com/mrdbourke/pytorch-deep-learning/blob/main/06_pytorch_transfer_learning.ipynb

https://github.com/udacity/deep-learning-v2-pytorch/blob/master/intro-to-pytorch/Part%208%20-%20Transfer%20Learning%20(Solution).ipynb

Extract features from last hidden layer Pytorch · GitHub

https://d2l.ai/d2l-en.pdf

cs329p_slides_14_1.pdf (d2l.ai)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Imran	Student ID:	20696366
Other name(s):	Nuha		
Unit name:	Machine Learning	Unit ID:	COMP3010
Lecturer / unit coordinator:	Miss Sushma Hans	Tutor:	
Date of submission:	31/03/23	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- ✓ The above information is complete and accurate.
- ✓ The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- ✓ I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- ✓ I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- ✓ Plagiarism and collusion are dishonest, and unfair to all other students.
- ✓ Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- ✓ If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- ✓ Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- ✓ It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Nuha Imran Date of signature: 31/03/23

(By submitting this form, you indicate that you agree with all the above text.)

Double-click (or enter) to edit

Getting the imports I need

```
import torch
import torchvision
from torchvision import transforms, datasets
from torch import nn
!pip install d2l
%pip install matplotlib-inline
from d2l import torch as d2l
```

Unzipping my files so I can get the images in the code

```
!unzip /content/Training\ and\ Validation\ Data_2023_T1_COMP3010.zip
```

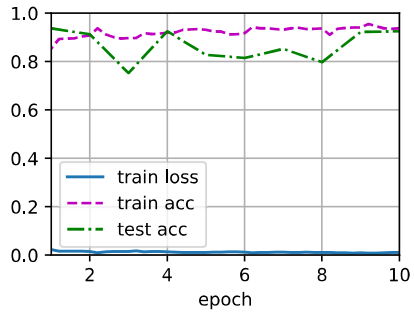
```
#the compose function applies transformations to each image in order
train_transforms = transforms.Compose([
    #randomly rotates image by 30 degrees(for data augmentation purposes)
    transforms.RandomRotation(30),
    #crops image to a square size of 224*224
    transforms.RandomResizedCrop(224),
    #flips image horizontally again for augmentation purposes
    transforms.RandomHorizontalFlip(),
    #converts image to pytorch tensor
    transforms.ToTensor(),
    #normalising pixel of image
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#doing the same to validation and testing data too
val_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#getting the data from the path and subdirectories
data_dir = '/content/Training and Validation Data_2023_T1_COMP3010 (2)/Training and Validation Data_2023_T1_COMP3010/Data/coral image classif
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
val_data = datasets.ImageFolder(data_dir + '/val', transform=val_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
```

```
#number of samples processed in each batch
batch_size = 16
#shuffle is for shuffling data before creating batches
#the train_data,val_data and test_data are datasets
#data loaders iterate over these datasets in batch sizes
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.alexnet(pretrained = True)
```

```
#replacing last linear layer with 2 output units
#create new linear layer with 4096 input units and two outputs
mod.classifier[6] = nn.Linear(4096, 2)
#adding activation function with dimension of one
mod.classifier.add_module("7", nn.LogSoftmax(dim = 1))


#rnd.fc = nn.Linear(rnd.fc.in_features, len(train_data.classes))
#checking if GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")



torch.cuda.is_available()


#defining the loss function and optimiser
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mod.parameters(), lr=0.001, weight_decay=0.001, momentum=0.9)
## optimizer1 = torch.optim.SGD(rnd.parameters(), lr=0.001, momentum=0.9)


!pip uninstall matplotlib
!pip install --upgrade matplotlib


torch.cuda.is_available()


# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.011, train acc 0.937, test acc 0.925
341.4 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.945
```

Double-click (or enter) to edit

Getting the imports I need

```
import torch
import torchvision
from torchvision import transforms, datasets
from torch import nn
!pip install d2l
%pip install matplotlib-inline
from d2l import torch as d2l
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC


torch.manual_seed(0)
np.random.seed(0)
```

Unzipping my files so I can get the images in the code

```
!unzip /content/Training\ and\ Validation\ Data_2023_T1_COMP3010.zip


#the compose function applies transformations to each image in order
train_transforms = transforms.Compose([
    #randomly rotates image by 30 degrees(for data augmentation purposes)
    transforms.RandomRotation(30),
    #crops image to a square size of 224*224
    transforms.RandomResizedCrop(224),
    #flips image horizontally again for augmentation purposes
    transforms.RandomHorizontalFlip(),
    #converts image to pytorch tensor
    transforms.ToTensor(),
    #normalising pixel of image
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])


#doing the same to validation and testing data too
val_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])


test_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])


#getting the data from the path and subdirectories
data_dir = '/content/Training and Validation Data_2023_T1_COMP3010 (2)/Training and Validation Data_2023_T1_COMP3010/Data/coral image classif
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
val_data = datasets.ImageFolder(data_dir + '/val', transform=val_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)




#number of samples processed in each batch
batch_size = 16
#shuffle is for shuffling data before creating batches
#the train_data,val_data and test_data are datasets
#data loaders iterate over these datasets in batch sizes
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
```

```
valloader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)



#loading pre-trained resnet model from the torch library
mod = torchvision.models.resnet50(pretrained = True)
#this one is not pre-trained has randomised weights
rnd = torchvision.models.resnet50()
#replaces last fully connected layer with new layer that has same input features as old but new output features,new output
#features equal to number of classes in training data


print(mod.fc.in_features)
print(len(train_data.classes))
rnd.fc = nn.Linear(rnd.fc.in_features, len(train_data.classes))
#checking if GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")



torch.cuda.is_available()


#defining a class that takes in from the nn.module and calling it CNN feature extractor
class CNNFeatureExtractor(nn.Module):
  #takes in an argument mod which in my case is a pretrained model resnet50
    def __init__(self, mod):
        super().__init__()
        #the nn.seq object is a container for the layers and uses list to get
        #all the pretrained layers and excludes out the last layer since thats what we need
        self.features = nn.Sequential(*list(mod.children())[:-1])
        #x is an input tensor
    def forward(self, x):
      #input tensor passed through convulutional layer
        x = self.features(x)
        #reshapes based on batch size and number ofc features
        x = x.view(x.size(0), -1)
        #returns the tensor
        return x



mod.to(device)
mod.eval()
#creating to empty lists with these names
features_train = []
labels_train = []
#just no gradients being computed
with torch.no_grad():
    #iterating through the trainloader I made initially
    for images, labels in trainloader:
        #using cuda
        images = images.to(device)
        labels = labels.to(device)
        #passing the images  through mod so i can get features
        features = mod(images)
        #converting features from tensor to numpy arrays
        features_train.extend(features.cpu().numpy().astype(np.float32)) # cast to np.float32
        labels_train.extend(labels.cpu().numpy())
#doing the same as above for validation and testing
features_val = []
labels_val = []
with torch.no_grad():
    for images, labels in valloader:
        images = images.to(device)
        labels = labels.to(device)
        features = mod(images)
        features_val.extend(features.cpu().numpy().astype(np.float32)) # cast to np.float32
        labels_val.extend(labels.cpu().numpy())

features_test = []
labels_test = []
with torch.no_grad():
    for images, labels in testloader:
        images = images.to(device)
        labels = labels.to(device)
```

```
    features = mod(images)
    features_test.extend(features.cpu().numpy().astype(np.float32)) # cast to np.float32
    labels_test.extend(labels.cpu().numpy())
```

```
torch.cuda.is_available()
```

```
True
```

```
# training a simple model using extracted CNN features
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(features_train, labels_train)
```

```
KNeighborsClassifier()
```

```
# evaluating the performance on validation data
train_accuracy = knn.score(features_train, labels_train)
print('Training accuracy:', train_accuracy)
```

```
Training accuracy: 0.9418084153983886
```

```
# evaluating the performance on validation data
val_accuracy = knn.score(features_val, labels_val)
print('Validation accuracy:', val_accuracy)
```

```
Validation accuracy: 0.9275
```

```
# evaluating the performance on testing data
test_accuracy = knn.score(features_test, labels_test)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.92
```

Double-click (or enter) to edit

Getting the imports I need

```
import torch
import torchvision
from torchvision import transforms, datasets
from torch import nn
!pip install d2l
%pip install matplotlib-inline
from d2l import torch as d2l
```

Unzipping my files so I can get the images in the code

```
!unzip /content/Training\ and\ Validation\ Data_2023_T1_COMP3010.zip
```

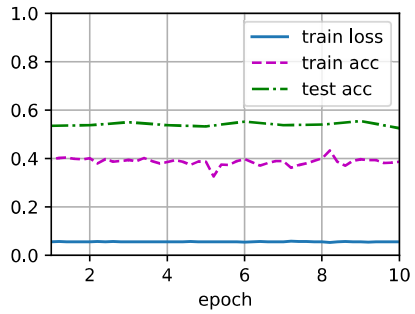
```
!unzip /content/Testing\ Data_2023.zip
```

```
#the compose function applies transformations to each image in order
train_transforms = transforms.Compose([
    #randomly rotates image by 30 degrees(for data augmentation purposes)
    transforms.RandomRotation(30),
    #crops image to a square size of 224*224
    transforms.RandomResizedCrop(224),
    #flips image horizontally again for augmentation purposes
    transforms.RandomHorizontalFlip(),
    #converts image to pytorch tensor
    transforms.ToTensor(),
    #normalising pixel of image
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#doing the same to validation and testing data too
val_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#getting the data from the path and subdirectories
data_dir = '/content/Training and Validation Data_2023_T1_COMP3010 (2)/Training and Validation Data_2023_T1_COMP3010/Data/coral image classif
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
val_data = datasets.ImageFolder(data_dir + '/val', transform=val_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
```

```
#number of samples processed in each batch
batch_size = 16
#shuffle is for shuffling data before creating batches
#the train_data,val_data and test_data are datasets
#data loaders iterate over these datasets in batch sizes
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.resnet18(pretrained = True)
#this one is not pre-trained has randomised weights
rnd = torchvision.models.resnet50()
#replaces last fully connected layer with new layer that has same input features as old but new output features,new output
#features equal to number of classes in training data
mod.fc = nn.Linear(mod.fc.in_features, len(train_data.classes))

print(mod.fc.in_features)
print(len(train_data.classes))
rnd.fc = nn.Linear(rnd.fc.in_features, len(train_data.classes))
#checking if GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")



torch.cuda.is_available()


#defining the loss function and optimiser
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mod.parameters(), lr=0.001, weight_decay=0.001, momentum=0.9)
## optimizer1 = torch.optim.SGD(rnd.parameters(), lr=0.001, momentum=0.9)


!pip uninstall matplotlib
!pip install --upgrade matplotlib


torch.cuda.is_available()


# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.056, train acc 0.387, test acc 0.525
339.3 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.5225
```

Double-click (or enter) to edit

Getting the imports I need

```
import torch
import torchvision
from torchvision import transforms, datasets
from torch import nn
!pip install d2l
%pip install matplotlib-inline
from d2l import torch as d2l
```

Unzipping my files so I can get the images in the code

```
!unzip /content/Training\ and\ Validation\ Data_2023_T1_COMP3010.zip
```

```
!unzip /content/Testing\ Data_2023.zip
```

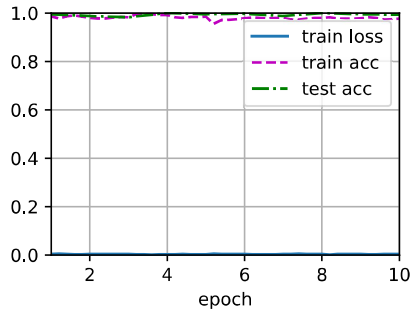
```
#the compose function applies transformations to each image in order
train_transforms = transforms.Compose([
    #randomly rotates image by 30 degrees(for data augmentation purposes)
    transforms.RandomRotation(30),
    #crops image to a square size of 224*224
    transforms.RandomResizedCrop(224),
    #flips image horizontally again for augmentation purposes
    transforms.RandomHorizontalFlip(),
    #converts image to pytorch tensor
    transforms.ToTensor(),
    #normalising pixel of image
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#doing the same to validation and testing data too
val_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#getting the data from the path and subdirectories
data_dir = '/content/Training and Validation Data_2023_T1_COMP3010 (2)/Training and Validation Data_2023_T1_COMP3010/Data/coral image classif
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
val_data = datasets.ImageFolder(data_dir + '/val', transform=val_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)

#number of samples processed in each batch
batch_size = 16
#shuffle is for shuffling data before creating batches
#the train_data,val_data and test_data are datasets
#data loaders iterate over these datasets in batch sizes
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.resnet50(pretrained = True)
#this one is not pre-trained has randomised weights
rnd = torchvision.models.resnet50()
#replaces last fully connected layer with new layer that has same input features as old but new output features,new output
#features equal to number of classes in training data
mod.fc = nn.Linear(mod.fc.in_features, len(train_data.classes))

print(mod.fc.in_features)
print(len(train_data.classes))
rnd.fc = nn.Linear(rnd.fc.in_features, len(train_data.classes))
#checking if GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")



torch.cuda.is_available()


#defining the loss function and optimiser
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mod.parameters(), lr=0.001, weight_decay=0.001, momentum=0.9)
## optimizer1 = torch.optim.SGD(rnd.parameters(), lr=0.001, momentum=0.9)


!pip uninstall matplotlib
!pip install --upgrade matplotlib


torch.cuda.is_available()
```

```
False
```

```
# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.003, train acc 0.977, test acc 0.993
98.0 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.99
```

Double-click (or enter) to edit

Getting the imports I need

```
import torch
import torchvision
from torchvision import transforms, datasets
from torch import nn
!pip install d2l
%pip install matplotlib-inline
from d2l import torch as d2l
```

Unzipping my files so I can get the images in the code

```
!unzip /content/Training\ and\ Validation\ Data_2023_T1_COMP3010.zip
```

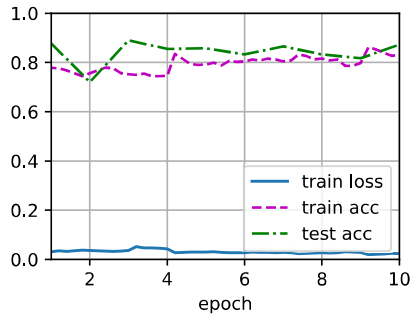
```
!unzip /content/Testing\ Data_2023.zip
```

```
#the compose function applies transformations to each image in order
train_transforms = transforms.Compose([
    #randomly rotates image by 30 degrees(for data augmentation purposes)
    transforms.RandomRotation(30),
    #crops image to a square size of 224*224
    transforms.RandomResizedCrop(224),
    #flips image horizontally again for augmentation purposes
    transforms.RandomHorizontalFlip(),
    #converts image to pytorch tensor
    transforms.ToTensor(),
    #normalising pixel of image
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#doing the same to validation and testing data too
val_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#getting the data from the path and subdirectories
data_dir = '/content/Training and Validation Data_2023_T1_COMP3010 (2)/Training and Validation Data_2023_T1_COMP3010/Data/coral image classif
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
val_data = datasets.ImageFolder(data_dir + '/val', transform=val_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
```

```
#number of samples processed in each batch
batch_size = 16
#shuffle is for shuffling data before creating batches
#the train_data,val_data and test_data are datasets
#data loaders iterate over these datasets in batch sizes
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.resnet50(pretrained = True)
#this one is not pre-trained has randomised weights
rnd = torchvision.models.resnet50()
#replaces last fully connected layer with new layer that has same input features as old but new output features,new output
#features equal to number of classes in training data
mod.fc = nn.Linear(mod.fc.in_features, len(train_data.classes))

print(mod.fc.in_features)
print(len(train_data.classes))
rnd.fc = nn.Linear(rnd.fc.in_features, len(train_data.classes))
#checking if GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")



torch.cuda.is_available()


#defining the loss function and optimiser
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mod.parameters(), lr=0.001, weight_decay=0.001, momentum=0.9)
optimizer1 = torch.optim.SGD(rnd.parameters(), lr=0.001, momentum=0.9, weight_decay=0.01)


!pip uninstall matplotlib
!pip install --upgrade matplotlib


torch.cuda.is_available()


# training the model with the library
num_epochs = 10
#d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer1, num_epochs)
```

```
loss 0.024, train acc 0.831, test acc 0.873
99.0 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(rnd, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.895
```

Double-click (or enter) to edit

Getting the imports I need

```
import torch
import torchvision
from torchvision import transforms, datasets
from torch import nn
!pip install d2l
%pip install matplotlib-inline
from d2l import torch as d2l
```

Unzipping my files so I can get the images in the code

```
!unzip /content/Training\ and\ Validation\ Data_2023_T1_COMP3010.zip
```

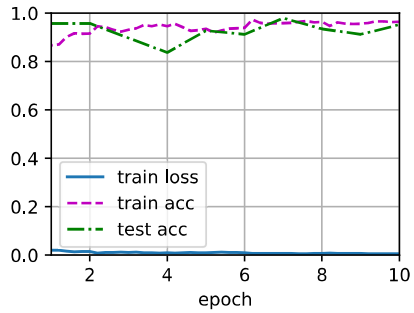
```
#the compose function applies transformations to each image in order
train_transforms = transforms.Compose([
    #randomly rotates image by 30 degrees(for data augmentation purposes)
    transforms.RandomRotation(30),
    #crops image to a square size of 224*224
    transforms.RandomResizedCrop(224),
    #flips image horizontally again for augmentation purposes
    transforms.RandomHorizontalFlip(),
    #converts image to pytorch tensor
    transforms.ToTensor(),
    #normalising pixel of image
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#doing the same to validation and testing data too
val_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transforms = transforms.Compose([
    transforms.Resize(255),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#getting the data from the path and subdirectories
data_dir = '/content/Training and Validation Data_2023_T1_COMP3010 (2)/Training and Validation Data_2023_T1_COMP3010/Data/coral image classif
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
val_data = datasets.ImageFolder(data_dir + '/val', transform=val_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
```

```
#number of samples processed in each batch
batch_size = 16
#shuffle is for shuffling data before creating batches
#the train_data,val_data and test_data are datasets
#data loaders iterate over these datasets in batch sizes
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

```
#loading pre-trained resnet model from the torch library
mod = torchvision.models.vgg16(pretrained = True)
```

```
#replacing last linear layer with 2 output units
#create new linear layer with 4096 input units and two outputs
mod.classifier[6] = nn.Linear(4096, 2)
#adding activation function with dimension of one
mod.classifier.add_module("7", nn.LogSoftmax(dim = 1))


#rnd.fc = nn.Linear(rnd.fc.in_features, len(train_data.classes))
#checking if GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")



torch.cuda.is_available()


#defining the loss function and optimiser
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mod.parameters(), lr=0.001, weight_decay=0.001, momentum=0.9)
## optimizer1 = torch.optim.SGD(rnd.parameters(), lr=0.001, momentum=0.9)


!pip uninstall matplotlib
!pip install --upgrade matplotlib


torch.cuda.is_available()


# training the model with the library
num_epochs = 10
d2l.train_ch13(mod, trainloader, valloader, loss, optimizer, num_epochs)
#d2l.train_ch13(rnd, trainloader,valloader, loss, optimizer, num_epochs)
```

```
loss 0.006, train acc 0.965, test acc 0.953
65.2 examples/sec on [device(type='cuda', index=0)]
```




```
# Evaluate the model on the testing data
test_accuracy = d2l.evaluate_accuracy_gpu(mod, testloader)
print('Testing accuracy:', test_accuracy)
```

```
Testing accuracy: 0.965
```