

GRIFFITH UNIVERSITY

3006ICT | ROBOTICS AND COMPUTER VISION

PRACTICE BOOK

Chapter 4

Transfer Learning, Fine-Tuning and Robustness

with ResNet-18

Dr Lei Wang

Griffith University
Student Practice Edition

How to use this chapter

Copy -> run -> inspect -> modify -> explain -> test. The goal is not to memorise PyTorch syntax. The goal is to understand what the model is doing, which parameters are changing, what the evaluation means, and when the result should not be trusted.

Chapter Purpose

Week 4 moves from classical computer-vision pipelines into deep-learning-based visual perception. The central engineering problem is not simply how to call ResNet-18. It is how to reuse a pretrained model on a small robotics dataset, control what is allowed to learn, measure generalisation, and test whether the system remains reliable when the visual environment changes.

The Week 4 learning story

Prepare the data -> establish a pretrained baseline -> learn a new classification head -> fine-tune only where useful -> stress-test under domain shift.

Learning Outcomes

- Inspect a small image-classification dataset and understand the roles of train, validation and domain-shift splits.
- Use a pretrained ResNet-18 and its official preprocessing pipeline.
- Explain the difference between a pretrained representation and a task-specific classification head.
- Freeze a backbone and train only a new classifier.
- Fine-tune a selected high-level block with a smaller learning rate.
- Interpret training loss and validation accuracy as different pieces of evidence.
- Measure accuracy under visual domain shift and inspect individual failures.
- Recognise overfitting, distribution shift and deployment risk in a robotics setting.
- Reuse the Week 4 training/evaluation structure in later machine-learning applications.

The Core Engineering Question

A pretrained model contains useful knowledge, but that knowledge was learned for another task. The engineering question is therefore: how much of the existing model should we keep, and how much should we adapt?

Strategy	What changes?	Main advantage	Main risk
Train from scratch	All parameters	Maximum task-specific flexibility	Usually data-hungry and easy to overfit
Frozen backbone	New classification head only	Stable and data-efficient	Backbone may not fully match the new task
Limited fine-tuning	Head + selected late layers	Adapts useful high-level features	More parameters can overfit
Full fine-tuning	Most/all parameters	Maximum adaptation	Highest data and overfitting risk on a tiny dataset

Important

Do not assume that fine-tuning must improve accuracy. It is an experiment. If the result becomes worse, that is useful evidence about data size, learning rate, model flexibility or domain mismatch.

Part 1: Dataset, Splits and Experimental Design

The supplied Week 4 dataset contains three classes: box, bottle and cup. The training split has 24 images per class; validation and domain-shift each have 8 images per class. The domain-shift split contains changed visual conditions such as clutter, illumination changes, blur, noise and partial occlusion.

Split	Approx. size	Used for	May update parameters?
train	24/class	Learning model parameters	Yes
val	8/class	Checking clean generalisation	No
domain_shift	8/class	Robustness / stress test	No

1 Folder structure

Copy this structure into your notes. The folder names become the class labels.

```
week4_dataset/  
  train/  
    box/  
    bottle/  
    cup/  
  val/  
    box/  
    bottle/  
    cup/  
  domain_shift/  
    box/  
    bottle/  
    cup/
```

2 Device selection

Key code

```
device = torch.device(  
    "cuda" if torch.cuda.is_available() else "cpu"  
)  
print("Using device:", device)
```

This is a small line of code with an important engineering idea: the program should adapt to the available hardware. Do not assume that every computer has CUDA.

3 Loading the dataset

```
from pathlib import Path  
from torchvision import datasets  
from torch.utils.data import DataLoader  
  
DATA_ROOT = Path("week4_dataset")  
BATCH_SIZE = 8  
  
train_set = datasets.ImageFolder(DATA_ROOT / "train", transform=train_tf)  
val_set = datasets.ImageFolder(DATA_ROOT / "val", transform=eval_tf)  
shift_set = datasets.ImageFolder(DATA_ROOT / "domain_shift", transform=eval_tf)  
  
train_loader = DataLoader(  
    train_set, batch_size=BATCH_SIZE,
```

```
        shuffle=True, num_workers=0
    )

    val_loader = DataLoader(
        val_set, batch_size=BATCH_SIZE,
        shuffle=False, num_workers=0
    )
```

Read ImageFolder as a data contract

ImageFolder infers labels from subfolder names. If the directory structure is wrong, the labels are wrong before the neural network even starts training.

4 First practice experiment

Before training anything, inspect the classes and one example from each class.

```
print("Classes:", train_set.classes)
print("Class-to-index:", train_set.class_to_idx)
print("Training images:", len(train_set))
print("Validation images:", len(val_set))
print("Domain-shift images:", len(shift_set))
```

Prediction pause: What should the three class names and dataset sizes be before you run the cell?

Discussion questions

Q1. Why should validation images not update model parameters?

Validation data is intended to estimate generalisation to unseen examples. If it influences training, information leaks from the evaluation set into learning and the reported validation result can become overly optimistic.

Q2. Why keep domain_shift separate?

It is a robustness test. If the model is trained or tuned on those shifted images, the test no longer measures how it handles an unseen change in visual conditions.

Q3. Why select CPU or GPU at runtime?

CUDA availability depends on the machine. Runtime selection makes the same code portable across CPU-only and GPU-enabled systems.

Part 2: What Does a Pretrained ResNet-18 Already Know?

Do not modify the model immediately. First establish a baseline. The original ResNet-18 is an ImageNet classifier, so its labels are not the same as the workshop labels. That is expected.

Baseline before intervention

A strong engineering experiment establishes what the original system does before changing it. Otherwise you cannot clearly explain what the intervention achieved.

1 Load the pretrained model

```
import torch
from torchvision.models import ResNet18_Weights, resnet18

weights = ResNet18_Weights.DEFAULT
model = resnet18(weights=weights).eval()

preprocess = weights.transforms()
categories = weights.meta["categories"]
```

The official preprocessing transform is important. The pretrained model expects inputs prepared in a way consistent with its original training. A common mistake is to resize or normalise inputs arbitrarily.

2 Top-5 prediction experiment

```
from PIL import Image

image = Image.open("path/to/one_image.jpg").convert("RGB")
batch = preprocess(image).unsqueeze(0)

with torch.no_grad():
    probabilities = model(batch).softmax(dim=1)[0]

values, indices = probabilities.topk(5)

for value, index in zip(values, indices):
    print(f"{categories[index]}: {float(value):.3f}")
```

Top-5 means the five ImageNet classes with the highest probabilities. Do not interpret an ImageNet label mismatch as proof that transfer learning is useless. The labels are different, but the internal visual representation can still be valuable.

Key insight: representation versus label space

A neural network can contain reusable visual features even when its original output classes are not your target classes. Transfer learning exploits this separation: keep useful internal features and replace the task-specific output layer.

Discussion questions

Q1. What does a Top-5 prediction list show?

It shows the five ImageNet categories with the highest predicted probabilities. It gives more information than a single Top-1 guess.

Q2. Why may ImageNet labels differ from box, bottle and cup?

ImageNet has a different 1000-class label vocabulary. Its categories can be more specific, more general or simply different from the workshop categories.

Q3. Why can the pretrained model still be useful?

The backbone has learned reusable patterns such as edges, textures, shapes and higher-level visual structures from a much larger dataset.

Part 3: Frozen Feature Extractor

Now replace the original ImageNet classification layer with a three-class layer and freeze the pretrained backbone. Only the new head is allowed to learn.

1 The two most important lines

```
for parameter in model.parameters():
    parameter.requires_grad = False

model.fc = nn.Linear(
    model.fc.in_features,
    len(train_set.classes)
)
```

The first block freezes all existing parameters. The second replaces the original 1000-class ImageNet classifier with a new classifier whose output dimension matches the three workshop classes.

2 Why is this efficient?

- The backbone already contains useful visual features.
- Only a small number of task-specific parameters need to be learned.
- The small dataset is less likely to cause severe overfitting than if the entire network were trained.
- The experiment isolates a clear question: are the pretrained features already good enough for this task?

3 Data augmentation

```
mean, std = weights.meta["mean"], weights.meta["std"]

train_tf = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(0.2, 0.2, 0.2, 0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

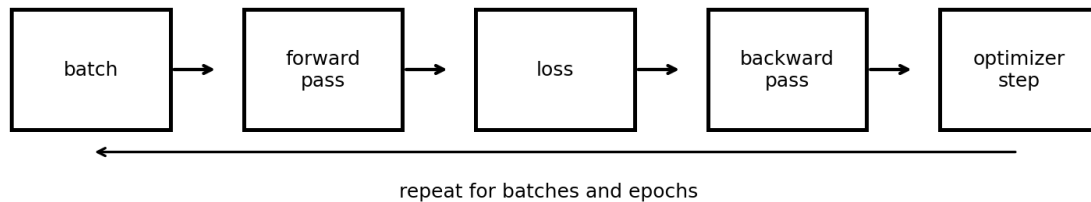
eval_tf = weights.transforms()
```

Training and validation transforms are deliberately different

Training uses random variation to expose the model to multiple appearances. Validation should be deterministic so that the evaluation is stable and comparable.

4 The training loop you should understand

The core supervised training loop



```
def train_one_epoch(model, loader, loss_fn, optimizer, device):
    model.train()
    total_loss = 0.0

    for images, labels in loader:
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        loss = loss_fn(model(images), labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * len(images)

    return total_loss / len(loader.dataset)
```

The key engineering question is not merely what `backward()` does. The important question is: which parameters are allowed to respond to the gradient? In this experiment, the backbone is frozen, so only `model.fc` changes.

5 Evaluation

```
def evaluate(model, loader, device):
    model.eval()
    correct = 0

    with torch.no_grad():
        for images, labels in loader:
            predictions = model(
                images.to(device)
            ).argmax(dim=1)

            correct += (
                predictions.cpu() == labels
            ).sum().item()

    return correct / len(loader.dataset)
```

Loss and accuracy answer different questions

Training loss is the optimisation signal. Validation accuracy estimates performance on held-out data. A falling training loss does not automatically mean that the model generalises better.

6 Optimiser and learning rate

```
loss_fn = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(
    model.fc.parameters(),
    lr=1e-3
)
```

Notice that the optimizer receives `model.fc.parameters()`, not `model.parameters()`. This makes the intended trainable subset explicit.

Practice experiments

- Change batch size from 8 to 4 or 16. What changes in runtime or loss behaviour?
- Temporarily remove `RandomHorizontalFlip`. Does the validation result change?
- Change the learning rate from $1e-3$ to $1e-4$. Does convergence become slower?
- Plot both training loss and validation accuracy. Do not rely on loss alone.
- Print the number of trainable parameters and compare it with the total number of parameters.

Discussion questions

Q1. Which parameters are updated?

Only the new `model.fc` classification layer. The pretrained convolutional backbone remains frozen.

Q2. Why can a frozen backbone work with a small dataset?

The backbone already contains useful general visual features learned from a much larger source dataset. The task-specific problem is reduced to learning how those features map to the three new classes.

Q3. How does augmentation reduce overfitting?

Random crops, flips and colour changes create varied training appearances, making it harder for the classifier to memorise one exact image appearance.

Part 4: Limited Fine-Tuning

Part 3 tested whether a fixed pretrained representation was sufficient. Part 4 relaxes that assumption slightly. We start from the frozen checkpoint, unfreeze only layer4, and use a smaller learning rate.

1 Start from the trained checkpoint

```
checkpoint = torch.load(
    "week4_frozen.pth",
    map_location=device
)

model = resnet18(weights=weights)

for parameter in model.parameters():
    parameter.requires_grad = False

model.fc = nn.Linear(
    model.fc.in_features,
    len(train_set.classes)
)

model.load_state_dict(checkpoint["model_state"])
```

Starting from the frozen checkpoint is important. We are not restarting the experiment. We are taking the Part 3 solution and changing one major experimental factor.

2 Unfreeze only layer4

```
for parameter in model.layer4.parameters():
    parameter.requires_grad = True

optimizer = torch.optim.Adam(
    filter(
        lambda parameter: parameter.requires_grad,
        model.parameters()
    ),
    lr=1e-4
)
```

Why layer4?

Later ResNet blocks contain higher-level representations. Unfreezing the final block gives the model some task-specific flexibility while preserving the earlier general-purpose representation.

3 Why the learning rate becomes smaller

In Part 3, the new classification head starts from random parameters, so a learning rate of 1e-3 is reasonable. In Part 4, layer4 already contains useful pretrained knowledge. A smaller rate such as 1e-4 encourages gradual adaptation rather than large updates that could damage the representation.

4 Verify what is trainable

```
print("Trainable parameters:")
for name, parameter in model.named_parameters():
    if parameter.requires_grad:
        print(name)
```

This is a valuable debugging step. Do not assume that the model is configured correctly just because the training loop runs.

Controlled experiment mindset

Part	Backbone	Head	Learning rate	Question
3	Frozen	Train	1e-3	Are fixed pretrained features sufficient?
4	layer4 trainable	Train	1e-4	Does limited adaptation help?

A good experiment changes one major factor at a time. That makes the comparison easier to interpret.

Discussion questions

Q1. Why use a smaller learning rate for fine-tuning?

The pretrained weights already contain useful information. Smaller updates reduce the risk of damaging those representations.

Q2. Why might limited fine-tuning improve performance?

Late features can adapt to distinctions that matter for box, bottle and cup rather than remaining optimised only for the original ImageNet task.

Q3. Why can fine-tuning too many layers increase overfitting?

A small dataset provides limited evidence to constrain many trainable parameters. The model may memorise training-specific details instead of learning robust patterns.

Part 5: Domain Shift and Robustness

This is the most robotics-oriented part of the chapter. A model can perform well in the conditions represented by its training data and still fail when the camera sees clutter, blur, illumination changes, noise or occlusion.

Core robotics insight

Accuracy on clean images is not the same as reliability in the real world. A deployed robot experiences changing viewpoints, lighting, clutter, partial visibility and sensor imperfections.

1 Build the two evaluation sets

```
val_set = datasets.ImageFolder(
    DATA_ROOT / "val",
    transform=eval_tf
)

shift_set = datasets.ImageFolder(
    DATA_ROOT / "domain_shift",
    transform=eval_tf
)
```

2 Compare four numbers

```
results = {
    "Frozen - clean": evaluate(frozen, val_loader, device),
    "Frozen - shifted": evaluate(frozen, shift_loader, device),
    "Fine-tuned - clean": evaluate(finnetuned, val_loader, device),
    "Fine-tuned - shifted": evaluate(finnetuned, shift_loader, device),
}

for name, accuracy in results.items():
    print(f"{name:24s} {accuracy:.3f}")
```

3 Accuracy drop

```
frozen_drop = (
    results["Frozen - clean"]
    - results["Frozen - shifted"]
)

finetuned_drop = (
    results["Fine-tuned - clean"]
    - results["Fine-tuned - shifted"]
)

print("Frozen drop:", frozen_drop)
print("Fine-tuned drop:", finetuned_drop)
```

A larger clean-to-shifted drop means a larger loss of performance under the tested shift. Do not automatically declare one model 'more robust' from a tiny numerical difference. Consider the dataset size and inspect the actual failures.

4 Visual error analysis

```
def show_predictions(model, dataset, device, count=6):
    figure, axes = plt.subplots(2, 3, figsize=(9, 6))

    for axis, index in zip(
        axes.flat,
        range(min(count, len(dataset)))
    ):
        image, label = dataset[index]
```

```

with torch.no_grad():
    prediction = model(
        image.unsqueeze(0).to(device)
    ).argmax(dim=1).item()

display = image.permute(1, 2, 0)
mean = torch.tensor([0.485, 0.456, 0.406])
std = torch.tensor([0.229, 0.224, 0.225])
display = (display * std + mean).clamp(0, 1)

axis.imshow(display)
axis.set_title(
    f"True: {dataset.classes[label]}\n"
    f"Pred: {dataset.classes[prediction]}"
)
axis.axis("off")

plt.tight_layout()
plt.show()

```

Look at individual mistakes. Ask: Was the object partially hidden? Was the lighting unusual? Was there distracting background structure? Was the object visually ambiguous? This converts a numerical score into engineering evidence.

Discussion questions

Q1. Why can clean accuracy be high while shifted accuracy is low?

The model learns statistical patterns from the training distribution. Changes in lighting, blur, clutter, noise or occlusion can make those patterns less reliable.

Q2. Which visual changes cause the most errors?

Use the displayed results as evidence. Identify whether clutter, darkness, blur, noise or occlusion is associated with the observed mistakes rather than guessing in advance.

Q3. Why does robustness matter for robotics?

A wrong classification can lead to wrong object selection, a missed target, an unsafe action or failure to stop when perception is unreliable.

Reusable Deep-Learning Skeleton

The most valuable part of this chapter is a structure you can reuse later. When you build a different image-classification application, change the dataset, model and task-specific head, but keep the engineering structure.

Stage	Question	Typical code
1. Environment	Can I reproduce the run?	torch / torchvision / device
2. Data	What are the classes and splits?	ImageFolder
3. Transform	What preprocessing does the model expect?	transforms.Compose / weights.transforms
4. Loader	How do samples become batches?	DataLoader
5. Model	What knowledge am I reusing?	pretrained model
6. Adapt	Which parameters should learn?	requires_grad / new head
7. Train	How are parameters updated?	loss.backward / optimizer.step
8. Validate	Does it generalise?	model.eval / no_grad
9. Stress test	Does it survive changed conditions?	domain-shift set
10. Inspect	Why did it fail?	visualisation / error analysis
11. Save	Can I reproduce the trained model?	torch.save

1 Copy-and-adapt notebook skeleton

Reusable reference skeleton - adapt the paths, classes and model carefully.

```
# 1. Imports and device
from pathlib import Path
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.models import ResNet18_Weights, resnet18

DATA_ROOT = Path("my_dataset")
BATCH_SIZE = 8
EPOCHS = 5

device = torch.device(
    "cuda" if torch.cuda.is_available() else "cpu"
)

# 2. Model-specific preprocessing
weights = ResNet18_Weights.DEFAULT
mean, std = weights.meta["mean"], weights.meta["std"]

train_tf = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean, std),
])

eval_tf = weights.transforms()

# 3. Data
train_set = datasets.ImageFolder(
    DATA_ROOT / "train",
    transform=train_tf
)
```

```

val_set = datasets.ImageFolder(
    DATA_ROOT / "val",
    transform=eval_tf
)

train_loader = DataLoader(
    train_set, batch_size=BATCH_SIZE,
    shuffle=True, num_workers=0
)
val_loader = DataLoader(
    val_set, batch_size=BATCH_SIZE,
    shuffle=False, num_workers=0
)

# 4. Pretrained model
model = resnet18(weights=weights)

# 5. Decide what learns
for parameter in model.parameters():
    parameter.requires_grad = False

model.fc = nn.Linear(
    model.fc.in_features,
    len(train_set.classes)
)

model = model.to(device)

# 6. Loss + optimiser
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
    model.fc.parameters(),
    lr=1e-3
)

# 7. Training loop
for epoch in range(EPOCHS):
    model.train()

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        loss = loss_fn(model(images), labels)
        loss.backward()
        optimizer.step()

# 8. Validation
model.eval()
correct = 0

with torch.no_grad():
    for images, labels in val_loader:
        predictions = model(
            images.to(device)
        ).argmax(dim=1)

        correct += (
            predictions.cpu() == labels
        ).sum().item()

accuracy = correct / len(val_set)
print("Validation accuracy:", accuracy)

# 9. Save
torch.save(

```

```
{  
    "model_state": model.state_dict(),  
    "classes": train_set.classes,  
},  
"model_checkpoint.pth",  
)
```

2 The five questions to ask before trusting the result

1. What exactly is the training data, and what does each label mean?
2. Which parameters are trainable right now?
3. What preprocessing is applied at training and evaluation time?
4. What evidence shows that the model generalises rather than memorises?
5. What happens when the input distribution changes?

Common Errors and How to Diagnose Them

Symptom	Likely cause	First check	Do not do this
Model downloads or API error	Torch/torchvision mismatch	Print versions and inspect installed API	Randomly change imports
Wrong class names	Folder structure / class mapping	train_set.classes and class_to_idx	Manually change labels after training
Training does not improve	Wrong trainable parameters / LR / data	requires_grad, loss, sample images	Increase everything at once
Validation is excellent but shifted is poor	Domain shift	Compare clean and shifted images	Assume the model is deployment-ready
Fine-tuning gets worse	Overfitting / LR too high / limited data	Compare Part 3 and Part 4 fairly	Unfreeze the entire network immediately
Displayed images look wrong	Normalisation not reversed	Undo mean/std before imshow	Modify model weights
Results vary unexpectedly	Random augmentation / random initialisation	Fix seeds if a controlled comparison is needed	Compare single noisy runs as proof
GPU code fails on CPU machine	Hard-coded CUDA	Runtime device selection	Assume CUDA exists

1 A debugging order that scales

- Check the environment and imports.
- Check file paths and dataset structure.
- Print class names and counts.
- Display raw images.
- Check transforms and tensor shape.
- Check the model output shape.
- Check which parameters require gradients.
- Run one batch and inspect the loss.
- Run one validation batch before a full experiment.
- Only then change hyperparameters.

Engineering rule

Do not debug five things at once. Change one variable, predict what should happen, run the experiment, and explain the result.

Using AI as an Engineering Assistant

AI can accelerate implementation, explanation and debugging, but generated code is a proposal rather than proof. You remain responsible for checking the data, assumptions, trainable parameters, numerical results and failure cases.

Purpose	Useful prompt
Explain	Explain transfer learning in plain English using the box/bottle/cup example.
Code	Generate only the PyTorch code needed to replace ResNet-18's final layer for a three-class dataset.
Review	Review my training loop and identify logical errors. Do not rewrite the whole program.
Parameters	Explain exactly which parameters are trainable in this model and why.
Testing	Suggest five tests for a small image classifier before I trust its validation accuracy.
Debug	My clean accuracy is high but domain-shift accuracy is low. Give me a diagnostic checklist before suggesting code changes.
Experiment design	Suggest one controlled experiment that changes only the learning rate and explain the expected effect.
Error analysis	Given these true/predicted examples, help me identify patterns without inventing causes not visible in the data.

Good AI habit

Ask AI to explain, review, test and diagnose before asking it to replace your whole program.

Chapter Self-Check

1. Why are training, validation and domain-shift data kept separate?

Answer / notes:

2. What does ImageFolder infer from the directory structure?

Answer / notes:

3. Why is the pretrained model useful even though ImageNet labels differ from the workshop labels?

Answer / notes:

4. What is the difference between a backbone and a classification head?

Answer / notes:

5. What does `requires_grad = False` do?

Answer / notes:

6. Why does Part 3 optimise only `model.fc`?

Answer / notes:

7. Why are training and validation transforms different?

Answer / notes:

8. What is the purpose of `loss.backward()`?

Answer / notes:

9. What is the purpose of `optimizer.step()`?

Answer / notes:

10. Why does validation use `model.eval()` and `torch.no_grad()`?

Answer / notes:

11. Why does Part 4 start from `week4_frozen.pth`?

Answer / notes:

12. Why is `layer4` unfrozen rather than the whole network?

Answer / notes:

13. Why is the fine-tuning learning rate smaller?

Answer / notes:

14. Why might fine-tuning fail to improve accuracy?

Answer / notes:

15. What does an accuracy drop under domain shift tell you?

Answer / notes:

16. Why should individual shifted examples be inspected rather than reporting only one number?

Answer / notes:

17. What could happen if a robot acts on an unreliable classifier?

Answer / notes:

Mini Challenges

Challenge 1 - Count trainable parameters

```
trainable = sum(
    p.numel()
    for p in model.parameters()
    if p.requires_grad
)

total = sum(
    p.numel()
    for p in model.parameters()
)

print("Trainable:", trainable)
print("Total:", total)
print("Trainable fraction:", trainable / total)
```

Before running: predict whether the trainable fraction should be small in Part 3.

Challenge 2 - Compare learning rates

Run the same frozen-backbone experiment with 1e-3 and 1e-4. Keep everything else unchanged.

Run	Learning rate	Final loss	Final val accuracy	Your interpretation
A	1e-3			
B	1e-4			

Challenge 3 - Robustness report

Complete the table using your own results.

Model	Clean accuracy	Shifted accuracy	Accuracy drop	More robust?
Frozen				
Fine-tuned				

Then choose two shifted images and write one sentence for each: what changed visually, what the model predicted, and whether the failure is plausibly related to the visible shift.

One-Page Reference - Week 4

Whole pipeline

DATA -> PREPROCESS -> PRETRAINED MODEL -> ADAPT -> TRAIN -> VALIDATE -> STRESS TEST -> INSPECT -> SAVE

Concept	Remember
ImageFolder	Folder names define class labels.
DataLoader	Turns dataset samples into batches.
weights.transforms()	Use the preprocessing expected by the pretrained weights.
requires_grad	Controls whether a parameter can learn.
model.fc	Task-specific classification head in ResNet-18.
CrossEntropyLoss	Standard loss for multi-class classification.
optimizer.zero_grad()	Clear old gradients.
loss.backward()	Compute gradients.
optimizer.step()	Update trainable parameters.
model.train()	Training behaviour.
model.eval()	Evaluation behaviour.
torch.no_grad()	Avoid gradient computation during inference/evaluation.
domain shift	Input distribution changes while task/labels remain the same.
accuracy drop	Simple measure of sensitivity to the tested shift.

Engineering habits to carry forward

- Understand the data before choosing the model.
- Establish a baseline before modifying a system.
- Make the trainable parameter set explicit.
- Separate training, validation and stress testing.
- Change one important experimental factor at a time.
- Use visualisation to explain errors, not just to make results look attractive.
- Treat clean accuracy as evidence, not as a guarantee of deployment reliability.
- Use AI to accelerate engineering, while retaining responsibility for verification.

Engineering Reflection

What part of the pretrained model would you trust most, and why?

What evidence from your experiment suggests that transfer learning was useful?

Did fine-tuning help, hurt or make little difference? What evidence supports your conclusion?

Which visual shift caused the most concerning errors?

If this classifier controlled a robot, what safety check would you add before allowing an action?

What would you change first if you had ten times more training data?

Week 4 - remember this

A good perception model is not simply a model with high accuracy. It is a model whose data, assumptions, trainable parameters, evaluation procedure and failure modes are understood.

End of Chapter 4 | 3006ICT Robotics and Computer Vision