Transfer learning

Flowers102

LECTURE 13 | GÉRON CH. 12

Applications of Machine Learning

BSc Mathematics of Artificial Intelligence · Third year

Where we are

Lecture 12 built a convolutional network for the flower brief: weight sharing, so one filter is reused at every position; equivariance, so the feature map follows a shift of the image; pooling, which turns that into invariance; then an architecture, trained from scratch. Rebuilt, retrained and scored on the test set in this lecture's notebook:

Model	Test accuracy
Always the commonest species	3.87%
Convolutional network from scratch, 4,807,494 parameters, 20 epochs	9.82%
What the operator needs	✗ 90%

THE CONDITION

We asked **1,020 labelled images** to pay for **4,807,494 parameters**. Most of those parameters encode facts about photographs in general — edges, textures, colour gradients — rather than facts about flowers, and there is nowhere near enough labelled data here to learn them.

Which suggests the repair: do not learn them. Somebody already has

Today

1. Why features transfer at all, and what that diagnoses about Lecture 12's network
2. **The method** — a pretrained backbone, a frozen probe, unfreezing the last block at its own learning rate, and augmentation
3. A failure condition: one transform handed to two loaders
4. The scoreboard, and when transfer fails
5. **Design exercise** — an agritech company's plant-disease app for farmers' phone photos, in nine steps, with the pitfalls and how to address them

No new derivation today — Lecture 12's parameter count is the whole argument, and this lecture is what you do about it. the method, and the conditions under which it fails.

EXAMINABLE

Open Lecture 13 in Colab

What a pretrained backbone has actually learned

Not “flowers”. A hierarchy, and only the bottom of it is general:

Depth	What the filters respond to	Transfers?
first block	edges, colour opponency, simple gradients	**to essentially anything ✓**
middle blocks	textures, repeated motifs, part-like shapes	**to most natural images ✓**
last blocks	object-specific configurations	**only to similar objects ✗**
the head	the 1,000 ImageNet classes	**not at all — it is replaced ✗**

Which is the whole recipe: **keep the bottom, adapt the top, replace the head.** Freezing is not a compromise forced by compute — it is a statement that those layers already encode what you would otherwise be trying to learn from 1,020 images.

The measurement that names the problem

	Accuracy
Training set	27.75%
✗ **Validation set**	✗ **11.67%**
✗ **Gap**	✗ **16 points**

This shape has a name: a gap between the images the network has seen and the ones it has not is **variance** — the fitted function depending on which images it was trained on.

Not a capacity problem. Twenty epochs is a short run and the training accuracy is still climbing when it stops — but it is already 16 points ahead of validation, and fitting harder widens that gap rather than closing it.

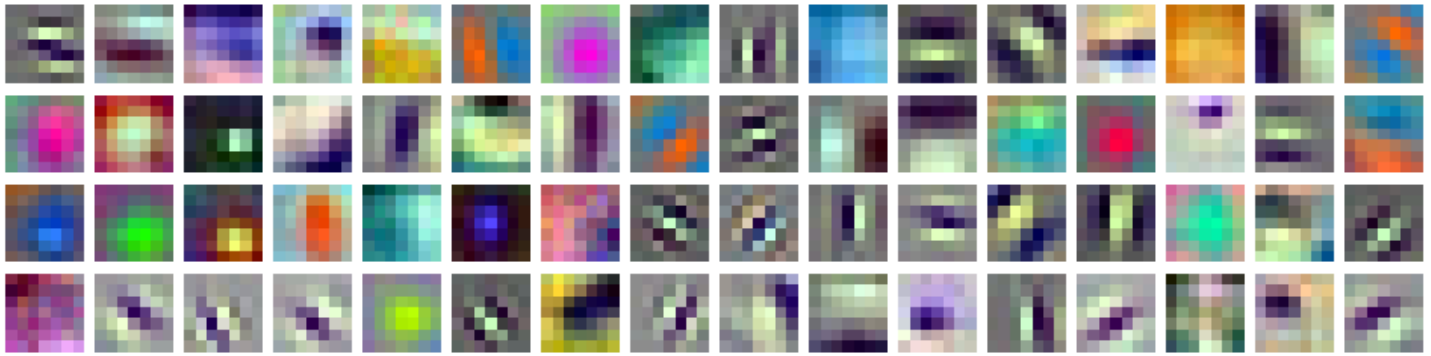
The two cures for variance

Cure	Can we?
More training data	**✗ no — labelling is the whole cost**
More constraint on the hypothesis space	**✓ yes, and we have already done it once**

Weight sharing was a constraint, and it bought a factor of 5,478,275. The question is where the next constraint comes from.

1.28 million labels, spent on this

resnet18, first layer: 64 filters of 7 × 7 × 3, trained on 1.28 million labelled photographs



The first layer of a ResNet-18 trained on ImageNet: oriented edges and opponent-colour blobs. 9,408 numbers, and every one of them is a fact about photographs rather than about the 1,000 classes they were fitted on.

The diagnosis

> **ONE SENTENCE**
>
> We asked 1,020 labelled images to pay for 4,807,494 parameters, most of which encode facts about photographs in general rather than facts about flowers.

The repair is therefore not a better architecture, a longer run or a cleverer schedule. It is **starting from weights that already work**.

Which backbone?

Family	Params	Use it when
MobileNet / EfficientNet-B0	~5 M	the budget is CPU, or inference has to be fast
ResNet-18 / ResNet-50	11 / 25 M	the default; well understood and widely reproduced
ConvNeXt, ViT-B	~88 M	plenty of data and plenty of compute

THE RULE THAT MATTERS MORE THAN THE FAMILY

A **bigger backbone is not automatically better on a small corpus.** You are fitting a head on frozen features; more features means more parameters in the head and more chance to overfit 1,020 images. Start small, and grow it only if the validation curve says the features are the bottleneck.

The names change every year. The trade-off does not.

NOT EXAMINABLE — ENGINEERING

Two decisions, in this order

1. **Linear probe.** Freeze everything, extract features once, fit a linear classifier. Cheap, and it tells you whether the backbone understands your images at all
2. **Fine-tune.** Unfreeze the last block or two, small learning rate, and only if the probe was promising

> **WHY THE ORDER IS NOT ARBITRARY**
>
> The probe is a *measurement of the features*, uncontaminated by anything you do to them. If it barely beats the trivial baseline, the features are wrong for this domain and fine-tuning is polishing the wrong object. If it is already close, fine-tuning has little left to buy and may cost you generalisation.

Freezing weights does not freeze a network

`requires_grad = False` stops the *gradients*. It does not stop batch normalisation updating its running mean and variance, which happens in the forward pass and has nothing to do with autograd.

> **THE CONDITION**
>
> A “frozen” backbone in `train()` mode keeps rewriting its normalisation statistics from your batches. The weights are fixed and the network still drifts — and it drifts towards a corpus of 1,020 images, which is the opposite of what you froze it for.

The fix is `model.eval()` on the frozen part — the same call that switches dropout off for evaluation, and for the same underlying reason: some modules behave differently in the two modes, and the mode is not implied by anything else you have set.

The recipe

1. Load a pretrained backbone and **its** preprocessing — the normalisation constants are part of the weights
2. Freeze it, put it in `eval()` mode, extract features once
3. Fit a linear head. **Measure.** This is the number everything else has to beat
4. Unfreeze the last block; learning rate an order of magnitude smaller than the head's. Measure again
5. Add augmentation, one transform at a time, each justified by an invariance you can state. Measure each
6. Report the ablation, not just the best number

✗ Steps 3, 4 and 5 each end in *measure*. A recipe without them is a ritual.

The backbone we will use

```
from torchvision.models import resnet18, ResNet18_Weights

weights = ResNet18_Weights.DEFAULT
net = resnet18(weights=weights)
```

	Value
Parameters	11,689,512
Trained on	ImageNet, 1,000 classes
Output before the classifier	512 numbers

Residual networks are Chapter 12's own example architecture. We use it; the residual connection itself is the book’s, not ours to re-derive.

The weights come with their own preprocessing

```
>>> ResNet18_Weights.DEFAULT.transforms()
ImageClassification(
    crop_size=[224]
    resize_size=[256]
    mean=[0.485, 0.456, 0.406]
    std=[0.229, 0.224, 0.225]
    interpolation=InterpolationMode.BILINEAR
)
```

✗ Use these statistics, not ones computed from your own training images. The network's first layer expects inputs on the scale it was trained with, and substituting your own is a silent, uncrashing degradation.

What is frozen and what moves

Freeze the general, train the specific

ImageNet weights, 11,689,512 parameters, downloaded not trained



2,782,784 parameters never move. 8,446,054 do.

Two learning rates, one optimiser: the early layers already hold edges and colours, and 1,020 flower photographs cannot improve on them.

2,782,784 parameters never move. 8,446,054 do.

Step 1 · freeze everything, replace the head

```
for p in net.parameters():
    p.requires_grad = False

net.fc = nn.Linear(512, 102)      # a fresh module: requires_grad is True
print(sum(p.numel() for p in net.parameters() if p.requires_grad))
# 52326

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

Replacing `fc` after the loop is not an accident of ordering — a newly constructed module has `requires_grad=True`, so the head is trainable and nothing else is.

If nothing before the head is training, run it once

```
body = nn.Sequential(*list(net.children())[:-1]).eval()

with torch.no_grad():
    F_train = torch.cat([body(xb).flatten(1) for xb, _ in train_loader])

# F_train is (1020, 512). Now train a linear model on a table.
```

The 1,020 images pass through the backbone once rather than once per epoch. The 60 epochs of the head then run on a table, which costs almost nothing.

The frozen backbone, measured

Model	Test accuracy
From scratch, 20 epochs	9.82%
✓ Frozen backbone, linear head	**81.09% ✓**

+71.3 points

and cheaper: one backbone pass.

Why a linear model on somebody else's features beats your network

- The 52,326 trainable parameters are being fitted with 1,020 examples — about 51 parameters per image, against 4713 before
- The 11,176,512 frozen ones were fitted with 1.28 million examples, by somebody else

The variance problem was not solved. It was moved to a dataset large enough to absorb it.

Step 2 · let the last block move

The early layers hold edges and colours, which are not about flowers. The *late* layers hold parts and textures, which partly are.

```
for p in net.layer4.parameters():
    p.requires_grad = True

opt = torch.optim.Adam([
    {"params": net.layer4.parameters(), "lr": 1e-4},
    {"params": net.fc.parameters(),     "lr": 1e-3},
])
```

One optimiser, two parameter groups, two learning rates. That is all a “differential learning rate” is.

Why the two rates have to differ

	Starting point	Rate
new head	random	**✓ large — it has everything to learn**
pretrained block	already good	**✓ small — it has something to lose**

✗ Give the backbone the head's learning rate and the first few batches — whose gradients come through a randomly initialised head — will damage the representation you downloaded. It does not crash; the accuracy simply comes out lower and you conclude that transfer learning does not work.

What a large learning rate destroys

Fine-tune the whole backbone at the rate you would use to train from scratch, and the first gradient step overwrites the features you came for. The network then relearns them from 1,020 images, badly.

> **THE CONDITION**
>
> The pretrained weights are a *good* starting point, which is exactly what makes them fragile: a step large enough to be useful on random weights is far too large for weights that are already nearly right.

This is why differential learning rates exist, and why the usual recipe warms up the new head first while everything else is frozen: the head starts random, and its early gradients are large enough to damage anything they reach.

The trap inside the frozen layers

```
net.train()
for xb, yb in train_loader:
    opt.zero_grad()
    lossf(net(xb), yb).backward()
    opt.step()
```

Every parameter in `layer1` is frozen. Something in `layer1` changes anyway.

Thirty seconds before the next slide.

Batch-norm running statistics are not parameters

- They are buffers, updated in the **forward** pass whenever the module is in training mode
- `requires_grad = False` does nothing to them — no gradient is involved
- So a frozen backbone in `train()` mode quietly replaces ImageNet's statistics with the statistics of 1,020 flower photographs, in batches of 32

```
net.train()
for m in [net.bn1, net.layer1, net.layer2, net.layer3]:
    m.eval()                          # freeze the buffers too
```

Step 3 · manufacture more data

We cannot buy labels. We can show the network a different view of the same labelled image every epoch.

- **Random resized crop** — a random sub-rectangle, scaled back to 224. Teaches scale and position tolerance
- **Horizontal flip** — a flower is still that species mirrored

Each is a transformation the label is known to survive. That is the entire criterion, and it is a modelling decision, not a library call.

One photograph, seven training examples



The label is the same in all eight. That is the only thing that makes this legitimate.

Applying it — and only where it belongs

```
train_tf = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.55, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(IMAGENET_MEAN, IMAGENET_STD),
])

eval_tf = transforms.Compose([          # val AND test
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(IMAGENET_MEAN, IMAGENET_STD),
])
```

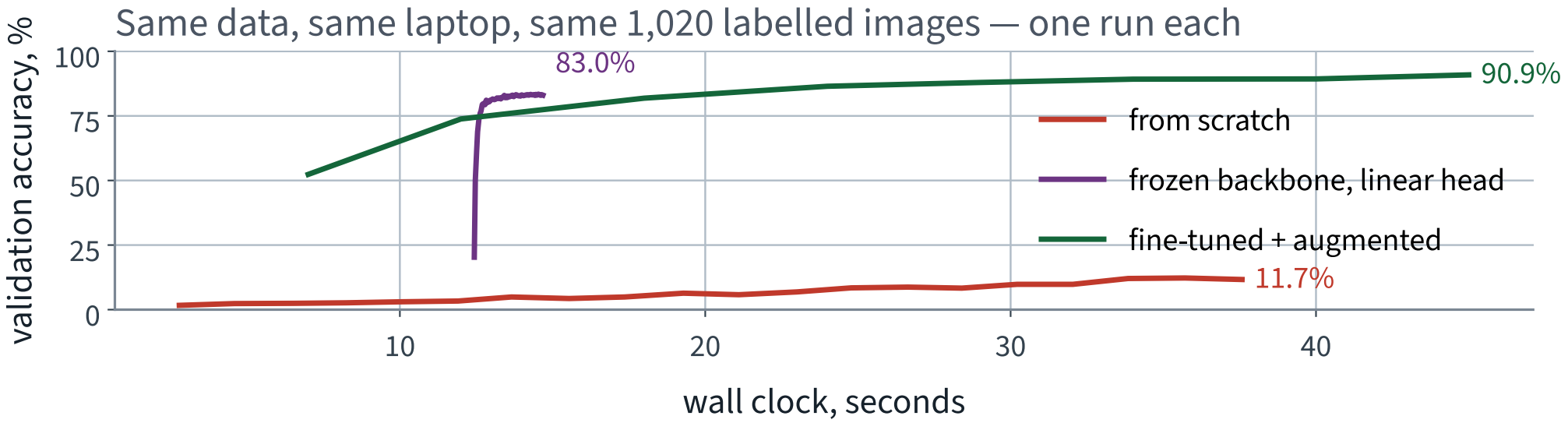
Two pipelines, deliberately. The evaluation one has no randomness in it; later in the lecture we measure what it costs when it does.

Fine-tuned, measured

Model	Test accuracy
From scratch	9.82%
Frozen backbone, linear head	81.09%
✓ layer4 unfrozen + augmentation	**87.82% ✓**

9.82% to 87.82%. It is not cheaper than training from scratch — a larger network on larger images, two validation passes per epoch included — it is better.

Same data, same laptop, same labels



Read both axes. The probe finishes first; the fine-tune passes it while the from-scratch run is still climbing near the floor.

Which part of the repair did the work

Change	Test accuracy	Gain
from scratch	9.82%	—
+ pretrained backbone, frozen	81.09%	**✓ 71.3 pts**
✓ + layer4 unfrozen, and augmentation	**87.82% ✓**	6.7 pts

Report the ablation, not just the final number. Almost all of the gain is in one row, and saying which row is what makes the result useful to somebody else.

All three rows are single runs, and none is replicated. The notebook unfreezes layer4 and augments in the same run, so it cannot say how the 6.7 points divide between the two; that takes one more fine-tune, without augmentation.

Augmentation is a training-time transform

It is easy to build one pipeline and hand it to both loaders. The training loader wants random crops and flips; the evaluation loader must not have them.

> **THE CONDITION, AND WHAT IT COSTS**
>
> An augmented evaluation set makes the score a **random variable** rather than a function of the weights. Score one fixed model twice and you get two different numbers. Measured here: the reported accuracy is **1.87 points pessimistic** and carries **2.16 points of noise** — a third of the gain that fine-tuning added on top of the frozen probe.

Nothing raises. The only symptom is that the number moves when nothing else did.

The score stops being a function of the weights

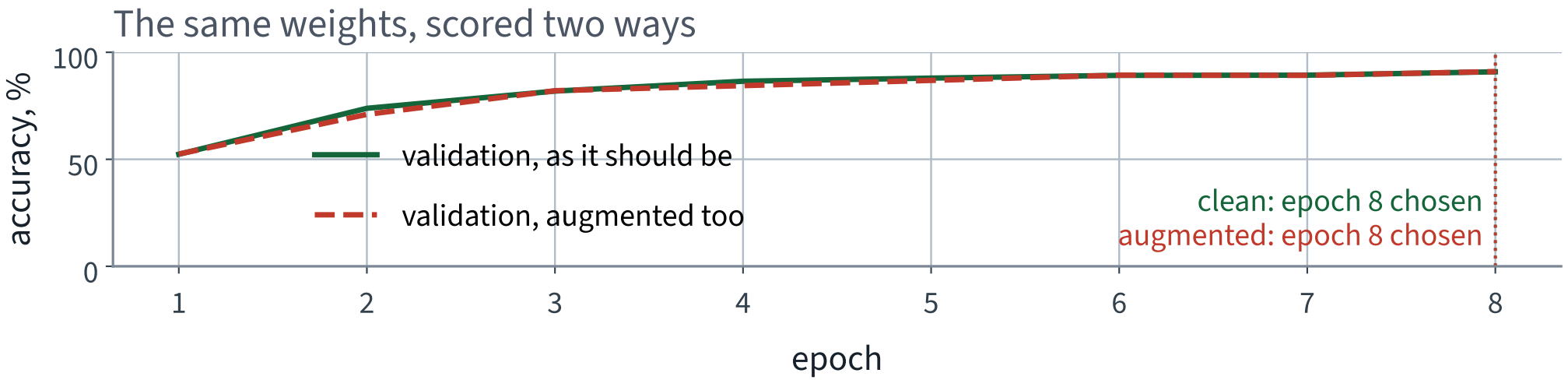
Score one fixed set of weights — the fine-tuned model above — on the augmented validation set, 10 times:

	Accuracy
on the clean validation set	90.88%
✗ augmented, mean of 10	**✗ 89.01%**
✗ augmented, spread	**✗ 2.16 points**

✗ The same weights, the same set, 2.16 points apart.

A deterministic function of fixed weights and fixed data does not wobble. When the score moves, something in its path is random.

Does it select a different model?



Not in this run: both curves are still rising and both peak at the last epoch. Where a curve flattens, 2.16 points of noise is enough to order two epochs by the evaluator's seed, and early stopping — keeping the best validation epoch — keeps the wrong one.

The check, in two lines

```
a = evaluate(model, val_loader)
b = evaluate(model, val_loader)
assert a == b, f"evaluation is not deterministic: {a} vs {b}"
```

Run every evaluation twice. If the two disagree, something in the evaluation path is random — augmentation, dropout still active, or a shuffled loader whose batches are averaged rather than pooled.

> **WHY AN ASSERT AND NOT A COMMENT**
>
> A comment saying “do not augment the validation set” is read once. This runs every time, costs two forward passes, and fails loudly at the moment the mistake is made rather than three lectures later.

Name three things that make an evaluation non-deterministic, and the check that catches all three.

EXAMINABLE

The whole application, in one table

Model	Test accuracy
Uniform guess	0.98%
Commonest species	3.87%
Convolutional net, from scratch	9.82%
Frozen backbone, linear head	81.09%
✓ Fine-tuned + augmented	**87.82% ✓**
Operator’s requirement	90%

Be precise about “cheaper”

	Passes per image	Trained parameters
From scratch, 128 × 128	20	4,807,494
✓ Frozen backbone	**1, no gradients ✓**	**52,326 ✓**
Fine-tuned at 224 × 224	8	**✗ layer4 and the head**

The fine-tune is no cheaper than from scratch: fewer epochs, but a much larger network on much larger images, validated twice an epoch. The frozen probe is the cheap one — and the one to run first, every time.

When this recipe does not work

Situation	Why
Your images are unlike the corpus — radar, spectrograms, microscopy	the learned features are about natural photographs
Your task is not classification	the backbone is trained to discard position
✗ You have millions of labelled examples	**✗ then the constraint is costing you, not helping**

Test it the same way: freeze, probe, and compare with the from-scratch baseline. The frozen probe is cheap, so there is never an excuse for not knowing.

The notebook for this lecture

Open Lecture 13 in Colab

- **Runs on free CPU.** The backbone is small and mostly frozen, which is exactly why this lecture is cheap and the last one was not
- Features extracted once and cached, then a linear head fitted on them — the step that does most of the work
- The comparison: from scratch, frozen probe, and fine-tuned with augmentation, on the same splits
- The augmented validation set scored ten times on fixed weights, and the two-line check that rules it out

> **WHAT TO CHANGE**
>
> Freeze *everything* including the final block, and refit. The accuracy drops by a few points — which tells you how much of the gain came from adapting the last block rather than from the backbone itself.

Design exercise · in the manner of an ML system-design interview

The task: an agritech company’s plant-disease app

> **THE BRIEF**
>
> An agritech company sells a crop-protection app: a farmer photographs a leaf, the app names the **disease** — or says healthy — and recommends a treatment. One class, on a phone in a field. Users: farmers and the company’s agronomists. The business measures renewals, the cost of a wrong recommendation — a spray not needed, a blight missed — and its exposure where such advice is regulated.

- **Given:** a few dozen labelled photographs per disease, most from its trial plots; a trickle from pilot customers’ fields; a pretrained backbone
- **Scale:** tens of thousands of farmers, many phones, every light and background
- **Consumers of the output:** the farmer, the agronomists who follow up, the outbreak map sold to cooperatives

✓ Nine steps follow, each with the pitfall that catches a first attempt and what to do about it. The model at the centre is today’s: a pretrained backbone, probed frozen and then fine-tuned.

Step 1 — Frame it: one class per leaf, and a label budget that is the whole constraint

- One row per photograph; the label is the disease, or healthy; the output is one class from the list
- The metric is per disease: the recall of each, since a missed blight spreads and a false alarm costs a farmer's trust; the baseline is "healthy", the commonest label
- Count the labels per class before choosing anything: a few dozen per disease is not a from-scratch budget

PITFALL — FEW LABELS, PLANNED AS IF THERE WERE MANY

The first plan budgets a network trained from scratch and a labelling campaign to feed it. From scratch on a few dozen images per class the network memorises the training set and scores near the baseline on everything else, and the campaign spends the agronomists' hours where they buy least. The method for this budget is a pretrained backbone, and the agronomists' hours go to the validation set, where a label is worth most.

Step 2 — The data: where the photographs come from

- The trial plots' photographs: one camera, a white sheet behind the leaf, even light. The customers': many phones, soil and sky behind the leaf, shadow, dust, a thumb in the corner
- The backbone's early layers transfer to both; the task is to make the labelled set look like the app's input

PITFALL — DOMAIN SHIFT FROM PHONE CAMERAS AND FIELDS

Fit and validate on the trial plots' photographs and the probe looks excellent; in the field the model meets a distribution it never saw, and the app is wrong in the one place it is used. Label field photographs from the pilot, even fewer of them, and let the trial plots' images be training data only: the number reported is the number on farmers' photographs.

Step 3 — Split by farmer, not by photograph

- A farmer photographs the same plant several times, from the same phone, in the same light: the photographs are near twins
- Hold whole farmers out: the validation set is farmers — their phones, fields and light — the model never saw; the test set is another such group, read once at the end

PITFALL — A RANDOM SPLIT OF PHOTOGRAPHS

The same diseased plant lands on both sides, and the score measures whether the model recognises a plant it has already seen. On the validation set it does; the pilot customers say otherwise. Split by farmer, and check that no phone model, field or day of the held-out group appears in training.

Step 4 — Representation: the backbone's features, with the backbone's own preprocessing

- The representation of a leaf is the vector the pretrained backbone emits before its classifier: a few hundred numbers, learned by somebody else on a corpus a thousand times larger
- The weights ship with their preprocessing — the resize, the crop, the per-channel mean and spread — and those constants are part of the weights

PITFALL — YOUR OWN NORMALISATION

Standardising with the mean and spread of your own leaf images feeds the first layer inputs on a scale it was not trained for; nothing raises, and the probe comes out worse. The same with the phone's full-resolution image fed unresized. Use `weights.transforms()` and nothing else — in the app exactly as in training.

Step 5 — The model: probe frozen first, then the last block at its own rate

- Freeze the backbone, put it in `eval()` mode, extract the features once, fit a linear head: cheap, and the number everything after has to beat
- If the probe is far above the baseline, unfreeze the last block at a rate an order of magnitude below the head's, and keep the frozen batch-norm layers in `eval()`: freezing gradients does not freeze their running statistics

> **PITFALL — FINE-TUNING FIRST, AND EVERYTHING AT ONCE**
>
> The whole backbone at the head's learning rate: the first batches' gradients come through a random head and overwrite the features you downloaded; the network relearns them from a few hundred leaves, badly, and the team concludes transfer does not work, and buys the labelling campaign. Probe, measure, then move only the last block, slowly.

Step 6 — Failure condition: augmentation, and the loader it belongs to

- Manufacture views of the training leaves: rotations and flips (a leaf has no up), random crops, brightness (phones differ). Not hue: a yellowing *is* the disease, and the label does not survive a colour shift
- Two pipelines, deliberately: the evaluation loader has no randomness in it

> **PITFALL — ONE TRANSFORM HANDED TO BOTH LOADERS**
>
> The validation score becomes a random variable: the same weights scored twice give two numbers, pessimistic on average and noisy by points, and a release is signed off on it. Run every evaluation twice and assert that the two agree; the assert fails at the moment the mistake is made, and catches dropout left on as well.

Step 7 — Evaluation: stop on the right curve, read the test farmers once

- Choose the epoch to keep on the clean validation curve of held-out farmers
- Report the ablation — probe, then the last block, then augmentation — because the row the gain came from is what the product team needs for the next crop
- Then the test farmers, once, with the sample size beside the number

> **PITFALL — EARLY STOPPING ON THE WRONG CURVE**
>
> The training curve rises for as long as you let it and selects the last epoch; an augmented validation curve carries enough noise to order two epochs by the evaluator’s seed, so where it flattens the checkpoint kept is the wrong one. Keep the checkpoint by the clean validation curve.

Step 8 — In the field: new farmers, new seasons, and a test set that must stay unread

- Photographs arrive with the season: new crops, diseases, phones. The frozen probe is cheap enough to rerun on each month's labelled field photographs, to see whether the features hold
- Release to a share of farmers first; retrain on the diagnoses the agronomists confirm, monitor per-disease recall, an agronomist on call for outbreak alerts

> **PITFALL — THE TEST SET REREAD AT EVERY RELEASE**
>
> Each version is compared on the same test farmers; after a dozen readings the test set has chosen the model. Retire it: each release is scored once on farmers who arrived since, and the old test group joins training.

Step 9 — Ship it: the pipeline, and the pitfalls in one table

Labels counted → field photographs labelled → split by farmer → the weights' own preprocessing → frozen probe, measured → last block unfrozen slowly, batch norms in `eval()` → augmentation on the training loader only → early stopping on the clean curve, ablation → test farmers once → few farmers first, retrain on confirmed labels.

Pitfall	Address it by
Few labels, a from-scratch plan	a pretrained backbone; labels spent on validation
Trial-plot photographs, field deployment	field photographs labelled, for validation
A random split of photographs	whole farmers held out
Your own normalisation	the weights' own transforms, in the app too
Fine-tuning everything at once	probe first; the last block at a rate ten times lower
One transform for both loaders	an evaluation loader with no randomness, asserted
Early stopping on the wrong curve	the clean validation curve
The test set reread every release	fresh farmers each release; the old group retired

What we did

1. Named the condition that made Lecture 12's network fall short: 1,020 images paying for 4.8 million parameters, most of which encode general facts about photographs
2. Reused a backbone trained on ImageNet with its own preprocessing, froze it, and fitted a linear head on features extracted once — the probe that every later step has to beat
3. Unfroze the last block at a learning rate ten times smaller than the head's, and kept the frozen batch-norm layers in `eval()`, because freezing weights does not freeze their running statistics
4. Augmented the training set only, with transformations the label survives, and measured what an augmented validation set costs
5. Read the ablation and its costs, and named the conditions under which transfer fails
6. Designed the plant-disease app an agritech company sells to farmers in nine steps — a label budget that chose transfer, field photographs for validation in place of the trial plots', a split by farmer, the weights' own preprocessing, the probe before the fine-tune, augmentation on the training loader only, early stopping on the clean curve, and a test group retired after one reading — each step named for the pitfall it avoids

NOT PRESENTED — FOR AFTERWARDS

Exercises Lecture 13

five, in the style of the written paper

Exercises — Lecture 13

1 State the order in which a pretrained network's head must be replaced and its body frozen, and what goes wrong in the other order. [4]

2 Fine-tuning uses 10^{-4} on the pretrained block and 10^{-3} on the new head. Justify the difference in one sentence. [4]

3 Freezing a batch-norm layer's weights does not freeze the layer. State what still changes, and the call that stops it. [4]

Solutions on Lecture 14's deck.

Exercises — Lecture 13 (continued)

4 An augmented validation loader is used to select the epoch to keep. State the two things this costs, and the one-line test that detects it. [5]

5 A frozen-backbone probe is both more accurate and faster than training from scratch on 1,020 images. Explain why both, and name the cost that must still be counted. [5]

Solutions on Lecture 14’s deck.

THE FIVE SET AT THE END OF LECTURE 12

Solutions Lecture 12

not part of this lecture

Solutions — Lecture 12

1. A dense layer mapping a $3 \times 128 \times 128$ input to a $32 \times 128 \times 128$ output has…

✓ $32 \times 3 \times 7 \times 7 = 4{,}704$.

Neither H nor W appears.

- One kernel per output channel, reused at every position
- The image size is genuinely absent — that is what weight sharing means

2. State the difference between equivariance and invariance, say which of the two pooling builds, and why a…

✓ Equivariance: $f(Tx) = Tf(x)$, the output moves with the input. Invariance: $f(Tx) = f(x)$, the output does not move. Pooling builds invariance, and a classifier wants it because the species is the same wherever the flower is.

- A max pool keeps the largest value in a window, so a shift that keeps that value inside its window changes nothing; four stages tolerate more, and a global pool tolerates every shift
- The position is a nuisance variable for naming a species, so discarding it costs nothing; a task that must say *where* could not afford it

Solutions — Lecture 12 (2)

3. Convolution is equivariant to translation on the interior of an image but not at the border. Give the reason.

✓ Zero padding invents input that was not there, so a shift moves real content into invented content.

- The identity assumes the operator sees the same neighbourhood before and after the shift
- At the edge it does not, so the equality is an interior statement

4. The lecture's network has 4,807,494 parameters. Name the layer that holds most of them, give its count…

✓ The first dense layer, from the flattened $256 \times 8 \times 8 = 16{,}384$ values to 256 units: $16{,}384 \times 256 + 256$ weights and biases, 4,194,560 in all, 87.3% of the total.

- Each of its units has its own weight for every value it reads, so its size grows with the map it flattens
- A convolution reuses one kernel at every position, so its count, $C_{\text{out}} C_{\text{in}} k^2$, holds no image size: all ten with their batch norms come to 586,720

Solutions — Lecture 12 (3)

5. After thirty epochs the first layer's 32 filters are still almost the vector they started as: cosine…

✓ Little gradient reaches it, and its scale does not matter. It needs a far larger corpus: those filters come from one about a thousand times this size.

- Ten layers of chain rule separate it from the loss, and 1,020 images give 960 optimiser steps in total
- A batch norm follows it, so the overall size of its weights does not change the function, and nothing pushes on it

LECTURE 14 · GÉRON CH. 12

Detection and segmentation

Where the object is, and how many: a detector returns a set of boxes
Intersection over union, why it stops giving a gradient, and average precision
Running a pretrained detector, choosing its threshold, non-maximum suppression
Tracking in brief, and segmentation: a label for every pixel

Run the Lecture 13 notebook first