DEEP LEARNING

A Cooking Guide

With PRACTICAL SESSION

Plant Species Classification

Sonia TIEO

Today's Menu

01 **The Ingredients**
Data · Labels · Splits

02 **The Recipe**
Perceptron · Your first neuron

03 **The Cooking**
Loss · Optimizer · Backprop · Training loop

HANDS-ON **Your First Dish**
Toy TP: blob vs ring · MLP · Challenges

01 02 03 **PlantNet Dataset + CNN**
Images · Preprocessing · Augmentation · Conv layers

HANDS-ON **PlantNet TP**
Build CNN · Train · Evaluate on real species

04 **Tasting**
Predictions · Confusion matrix · Precision/Recall

05 **Inside the Casseroles**
Feature Maps · PCA Embeddings

06 **Standing on Giants**
Transfer Learning · ResNet · Fine-tuning

SECTION 01

The Ingredients — Our Dataset

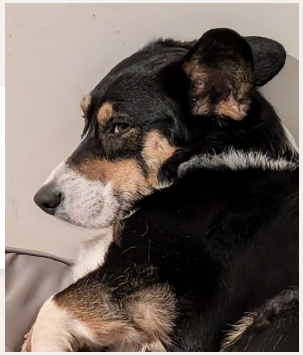
Data loading, splits, augmentation, and first look

The Ingredients

Data

Raw ingredients

Input samples (images, text, tabular rows)



The Ingredients

Data

Raw ingredients

Input samples (images, text, tabular rows)

Labels

Ingredient labels

Ground truth y for each sample x

Species : Canis familiaris
Age = 2 years
Name: Valky

SECTION 2

The Recipe — Build the Model

Perceptron · Neural networks · CNN architecture

The Perceptron — A Simple Taster

Kitchen Analogy

A simple taster who takes a few cues and makes a binary decision,
Or a one-step recipe:

INPUT

Enough sugar
+
fruity
+
acidic

→

NEURON

combine,
threshold,
decide.

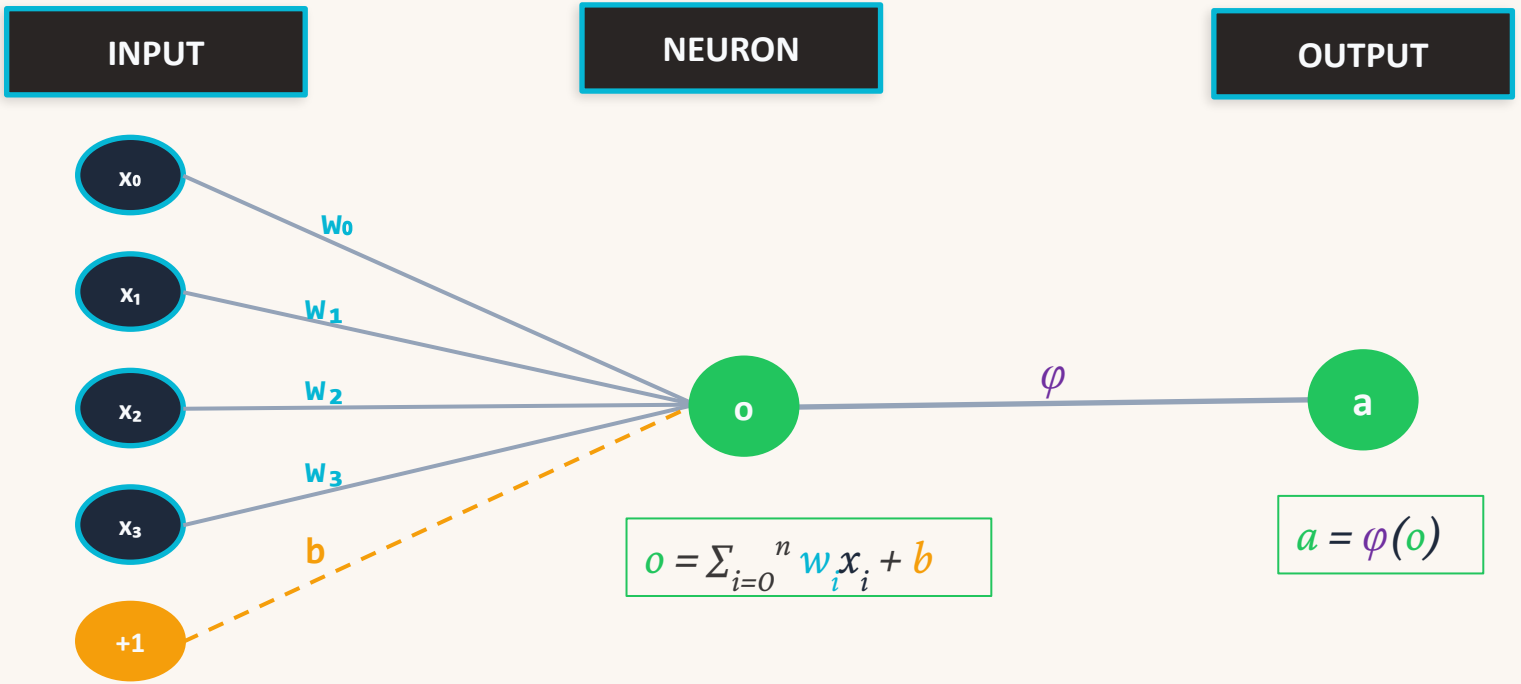
→

OUTPUT

It's a
dessert.

The Perceptron — A Simple Taster

The fundamental computational unit of a neural network



b : bias term · φ : activation function

PYTORCH

Your First Neuron

The perceptron — one line of PyTorch

```
import torch
import torch.nn as nn

# A perceptron = one linear layer
perceptron = nn.Linear(
    in_features=3,   # 3 inputs
    out_features=1    # 1 decision
)

# Forward pass
x = torch.tensor([1.0, 0.5, -0.3])
output = perceptron(x)
# output = Σ wᵢxᵢ + b
```

- nn.Linear wraps weights + bias automatically
- in_features = number of ingredients the taster receives
- No activation yet — just the raw weighted sum

SECTION 03

The Cooking — Training

Loss functions · Optimizers · Training loop · Learning curves

The Loss Function — How Bad Is the Dish?

You prepare a dish, taste it, and measure how far it is from what you wanted.

Dessert

Dessert –ish ??

NOT A DESSERT

The Loss Function — How Bad Is the Dish?

You prepare a dish, taste it, and measure how far it is from what you wanted.

The loss function is exactly that score: the distance between your prediction and reality.

Choosing the Right Loss

Match the loss to the task — like choosing the right seasoning for the dish.

Task	Output Activation	Loss Function	Formula
Regression	Linear / ReLU	**MSE**	$(1/n)\ \Sigma\ \|\|\hat{y}-y\|\|^2$
Binary Classification	Sigmoid $\sigma(x)$	**Binary Cross-Entropy**	$-(1/n)\ \Sigma\ \log\ p(\hat{y}=y)$
Multi-class Classification	Softmax	**Categorical Cross-Entropy**	$-(1/n)\ \Sigma\ \log$ $\text{softmax}[y]$

PYTORCH

Defining the Loss

Tell PyTorch how to measure mistakes

```
import torch.nn as nn

# Regression → MSE
mse_loss = nn.MSELoss()

# Classification → Cross-Entropy
# (includes softmax internally!)
ce_loss = nn.CrossEntropyLoss()

# Usage
predictions = model(x)        # forward
loss = ce_loss(predictions, labels)
print(f'Loss: {loss.item():.4f}')
```

- MSE for regression
- CrossEntropyLoss combines softmax + negative log-likelihood
- loss.item() extracts the scalar value

Loss & Optimizer — Two Different Jobs

One measures the mistake, the other fixes it.

The Loss Function

"How wrong is the model?"

- Compares predictions to true labels
- Returns a single number: the error
- Higher = the model is more wrong
- Does NOT change anything in the model

The Optimizer

"How should the model improve?"

- Uses gradients to update model weights
- Decides step size (learning rate)
- Many flavours: SGD, Adam, AdamW...
- This is what actually changes the model

`Loss = how wrong the model is` | `Optimizer = how the model improves`

PYTORCH

Setting Up the Optimizer

The algorithm that adjusts the recipe

```
import torch.optim as optim

# The go-to default
optimizer = optim.Adam(
    model.parameters(),
    lr=1e-3    # learning rate
)

# Or classic SGD with momentum
optimizer = optim.SGD(
    model.parameters(),
    lr=0.01,
    momentum=0.9
)
```

- model.parameters() automatically finds all weights & biases
- Adam = adaptive learning rate per parameter
- SGD + momentum for fine-tuning: often +1-2% accuracy

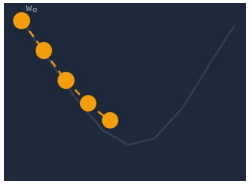
Choosing the Right Learning Rate

🐢

ρ too small

Converges, but very slowly. May need many epochs.

Slow but safe

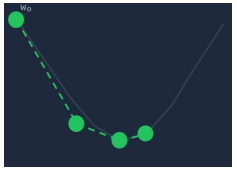


✅

ρ just right

Steady convergence to the minimum.

The sweet spot

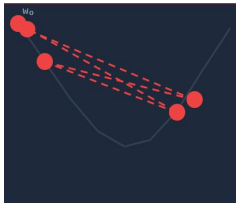


💥

ρ too large

Diverges — steps overshoot the minimum.

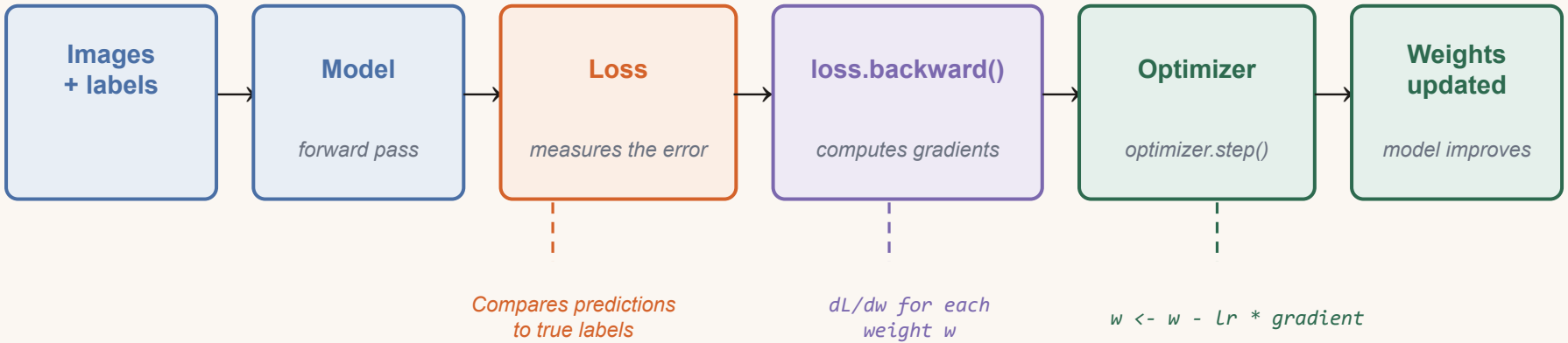
Unstable



In practice: use learning rate schedulers or adaptive optimizers (Adam, RMSProp) to tune ρ automatically.

Backpropagation — Tracing Back Mistakes

The dish is bad. The head chef traces back: ***too much cooking? bad proportions upstream? wrong mix at step 3?***
Backprop distributes the blame across every station in the chain.



`Loss = error` | `Backprop = how to reduce it` | `Optimizer = changes the weights`

Backpropagation — Tracing Back Mistakes

The dish is bad. The head chef traces back: ***too much cooking? bad proportions upstream? wrong mix at step 3?***
Backprop distributes the blame across every station in the chain.

Training mode

PYTORCH

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Training: forward + backward + step

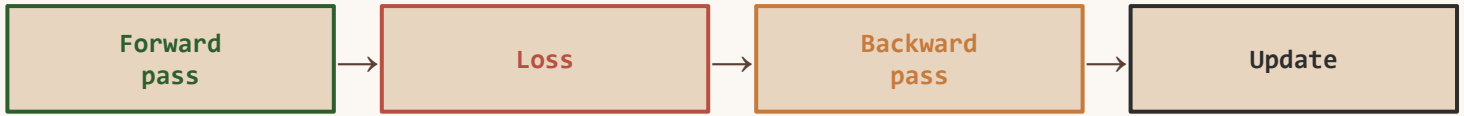
Evaluation mode

```
NO backward()
NO optimizer.step()
```

Evaluation: forward only

The Training Loop

Epoch after epoch: forward → loss → backward → update. Repeat until convergence.

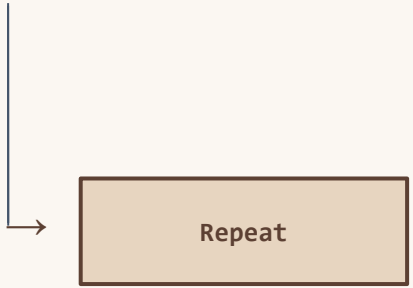


Feed batch through model → logits

Compare logits to true labels

runs backpropagation and computes $\partial L/\partial w$, which measures how the loss L changes when each weight w is modified.

$w \leftarrow w - \eta \cdot \partial L/\partial w$ (Adam step)



Next batch. All batches = 1 epoch.

Key concepts

`Batch size`	32 images at once → GPU parallelism
`Learning rate η`	0.001 typical → too big = diverge
`Epoch`	1 full pass over all training data
`model.eval()`	Disable dropout & batchnorm updates

PYTORCH

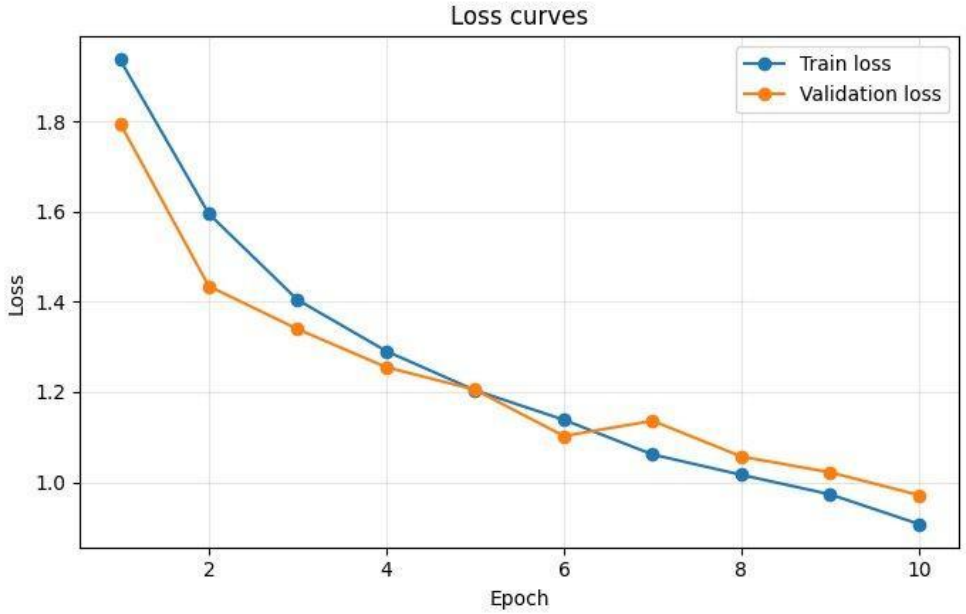
The Full Training Loop

```
for epoch in range(num_epochs):
  model.train()
  for images, labels in train_loader:
    images, labels = images.to(device), labels.to(device)

    # 1. Forward pass
    logits = model(images)
    loss = criterion(logits, labels)

    # 2. Backward pass (autograd)
    optimizer.zero_grad()
    loss.backward()

    # 3. Update weights
    optimizer.step()

  # Evaluate on validation set
  val_loss, val_acc = evaluate(
    model, val_loader, criterion, device)
```

- **zero_grad()** clears previous gradients
- **loss.backward( ):** computes the gradients - it tells the model how each weight should change to reduce the error.
- optimizer.step() changes the weights to make the model's predictions better.

Learning Curves — Loss

Track training and validation loss over 10 epochs (from scratch).



What we observe

Both train and validation loss decrease together → the model is learning.

Val loss stays close to train loss → no severe overfitting (yet).

Loss drops fast in early epochs, then slower: typical convergence.

How to read loss curves

Train ↓ + Val ↓ = learning OK

Train ↓ + Val ↑ = overfitting

Both stay high = underfitting or bad learning rate

Training — Practice Makes Perfect

Training = repeated practice in the kitchen. Epoch = one full pass through every recipe. Batch = a small lot of dishes before correction.

Epoch

One full pass through
training data

Batch

Mini-lot of samples
processed before update

Training Loss

Should always decrease
over epochs

Validation Loss

Diverging up = overfitting!
Save best checkpoint

Overfitting fixes: Dropout · BatchNorm · Data augmentation · Early stopping · Reduce model complexity

DL-Specific Regularization

Dropout

During training: randomly zero out neurons with probability p (0.2–0.5)

→ Forces redundant representations
→ Acts like an ensemble of 2^n networks
→ At inference: all active, weights × (1−p)

```
model.train()
  ← dropout ON
model.eval()
  ← dropout OFF
```

Batch Normalization

Normalizes activations within each mini-batch:

→ Stabilizes training
→ Allows higher learning rates
→ Acts as regularizer
→ Key ingredient inside every ResNet block

Rule of thumb: ReLU in hidden layers · Softmax in output · BatchNorm after every conv · Dropout before classifier

HANDS-ON

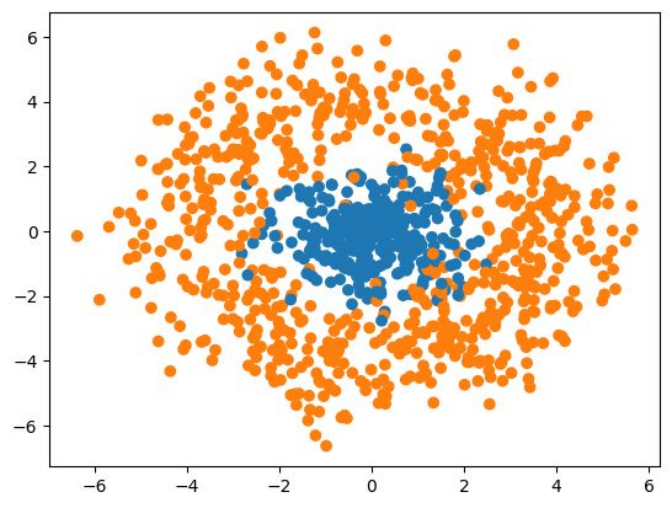
Your First Dish — Simple Classification

Synthetic data · MLP · Training loop · Evaluation

The Toy Dataset — Blob vs Ring

Synthetic 2D Data

Class 0: Gaussian blob (300 pts) at origin
Class 1: Ring distribution (700 pts) around it



PYTORCH

The Model — A Simple MLP

3 Linear layers, sigmoid output — but something is missing!

```
class Model(torch.nn.Module):
    def __init__(self,n_neurons = 20) -> None:
        super(Model, self).__init__()
        self.hidden0 = nn.Linear(2, n_neurons)
        self.hidden1 = nn.Linear(n_neurons, n_neurons)
        self.classif = nn.Linear(n_neurons, 1)

   def forward(self, x):
        x = self.hidden0(x)
        x = self.hidden1(x)    # No activation!
        return self.classif(x).sigmoid()
```

- 2 inputs (x, y coords)
  n_neurons hidden units
- stacking linear layers
  = one big linear
  transformation!
- super()
  initializes the PyTorch model
  structure.

A class = a mold

The CLASS

The blueprint

- The recipe for the cake
- Describes what an object IS
- Describes what it can DO

An OBJECT (instance)

The actual cake

- Built from the class
- You can make as many as you want!
- e.g. model = Model()

The two ingredients of a class

1 ATTRIBUTES

Data stored inside the object

```
self.n_neurons = 20
```

→ Value 20 is stored in the object
→ Accessible via self.n_neurons

__init__ = the constructor
Runs automatically when the object is created

2 METHODS

Functions belonging to the class

```
def forward(self, x):
```

→ What the object knows how to DO
→ Always takes self as first argument

forward = the main method
Describes how data flows through the network

PYTORCH

The Model — A Simple MLP

3 Linear layers, sigmoid output — but something is missing!

```
class Model(torch.nn.Module):
    def __init__(self,n_neurons = 20) -> None:
        super(Model, self).__init__()



    self.hidden0 = nn.Linear(2, n_neurons)
    self.hidden1 = nn.Linear(n_neurons, n_neurons)
    self.classif = nn.Linear(n_neurons, 1)


   def forward(self, x):
    x = self.hidden0(x)
    x = self.hidden1(x)    # No activation!
    return self.classif(x).sigmoid()
```

- 2 inputs (x, y coords)
  n_neurons hidden units
- stacking linear layers
  = one big linear
  transformation!
- super()
  initializes the PyTorch model
  structure.

How to use it?

01 Create an instance

```
model = Model(n_neurons=20)
```

__init__ runs automatically.

Layers hidden0, hidden1, classif are created.

02 Make a prediction

```
prediction = model(x)
```

forward() is called automatically.

Returns a value between 0 and 1.

Key takeaway — You don't have to write the class. Just understand it wraps the network and gives you a ready-to-use object.

PYTORCH

Custom Dataset + DataLoader

How PyTorch loads your data in batches

```
class CustomDataset(Dataset):
    def __init__(self, x_data, y_data):
        self.x = torch.tensor(x_data, dtype=torch.float32)
        self.y = torch.tensor(y_data, dtype=torch.float32)

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]
```

- __init__: store data as tensors
- __getitem__: return one (x, y) pair by index
- tells DataLoader the dataset size.

PYTORCH

Custom Dataset + DataLoader

How PyTorch loads your data in batches

```
dataset = CustomDataset(x_train, y_train)

loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

- Groups samples into batches of 32
- Shuffles data each epoch
- You just iterate: for x, y in loader

PYTORCH

Training Loop — Simple TP

```
net = Model(n_neurons=160).to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=0.00001)

for epoch in range(num_epochs):
    total_loss = 0.0
    for batch_x, batch_y in dataloader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        # 1. Forward pass
        probability = net.forward(batch_x)

        # 2. Compute loss (MSE here)
        loss = torch.mean((batch_y - probability.T)**2)

        # 3. Backward + Update
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

- **Setup:** model + optimizer.
  SGD with very small lr.
  160 neurons per hidden layer.
- **loss.backward( ):** computes the gradients - it tells the model how each weight should change to reduce the error.
- optimizer.step() changes the weights to make the model's predictions better.

Time to Code!

Open the notebook and follow along.

- ✓ GPU runtime enabled (Runtime → Change runtime type → T4 GPU)
- ✓ Run each section in order — don't skip ahead
- ✓ Compare results at each stage
- ✓ Ask questions — that's what we're here for

Challenge: What Is Missing?

Run the notebook — your accuracy will be low. Why?

Challenge: What Is Missing?

Run the notebook — your accuracy will be low. Why?

The Problem

Without activation functions between layers...

- Linear(Linear(x)) = one big Linear
- The model cannot learn non-linear boundaries
- A blob inside a ring needs curves, not lines!

Challenge: What Is Missing?

Run the notebook — your accuracy will be low. Why?

The Problem

Without activation functions between layers...

- Linear(Linear(x)) = one big Linear
- The model cannot learn non-linear boundaries
- A blob inside a ring needs curves, not lines!

The Fix

Add ReLU activation between hidden layers:

- x = self.hidden0(x)
- x = torch.relu(x) # This changes everything!
- x = self.hidden1(x)
- x = torch.relu(x)

ACT 2

A Real Kitchen — From Pixels to Plants

PlantNet dataset · CNNs · Transfer learning · Uncertainty

The Dataset — Plant Species

Pl@ntNet-300K

This repository contains the code used to produce the benchmark in the paper ***"Pl@ntNet-300K: a plant image dataset with high label ambiguity and a long-tailed distribution"***.

Download the dataset

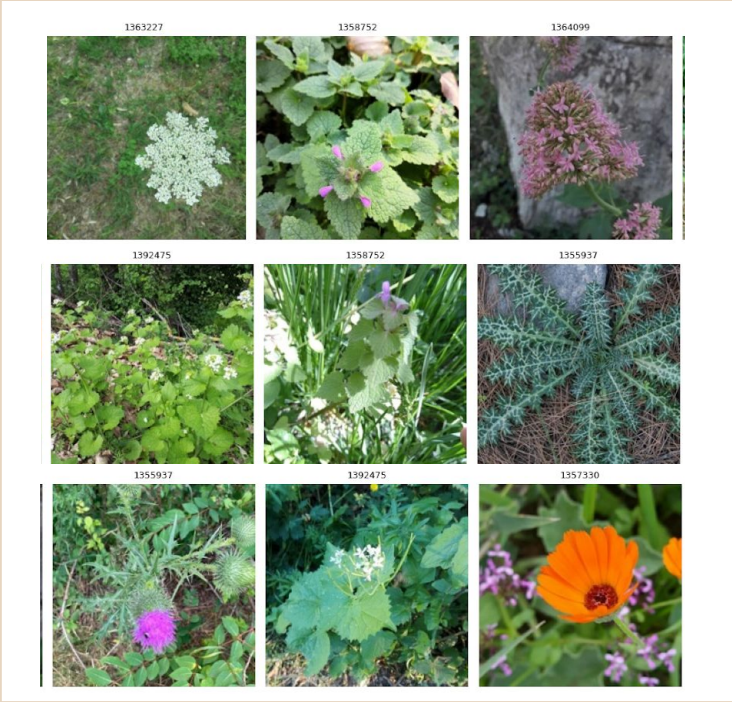
In order to train a model on the PlantNet-300K dataset, you first have to download the dataset on Zenodo.

Scientific Publication

You can find detailed information about the dataset as well as extensive experiments in the NeurIPS 2021 paper. If you use this work for your research, please cite the paper:

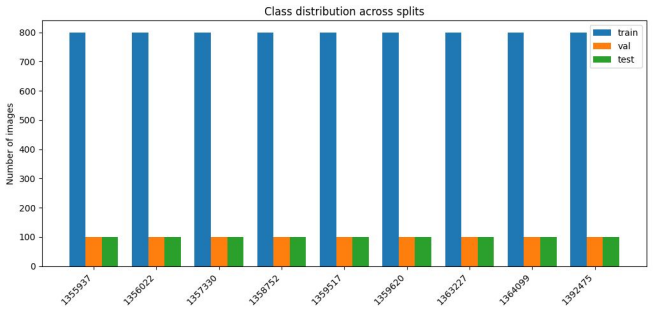
```
@inproceedings{plantnet-300k,
author    = {Garcin, Camille and Joly, Alexis and Bonnet, Pierre and Lombardo, Jean-Christophe and Affou
booktitle = {NeurIPS Datasets and Benchmarks 2021},
title     = {{Pl@ntNet-300K}: a plant image dataset with high label ambiguity and a long-tailed distribu
year      = {2021},
}
```

The Dataset — Plant Species



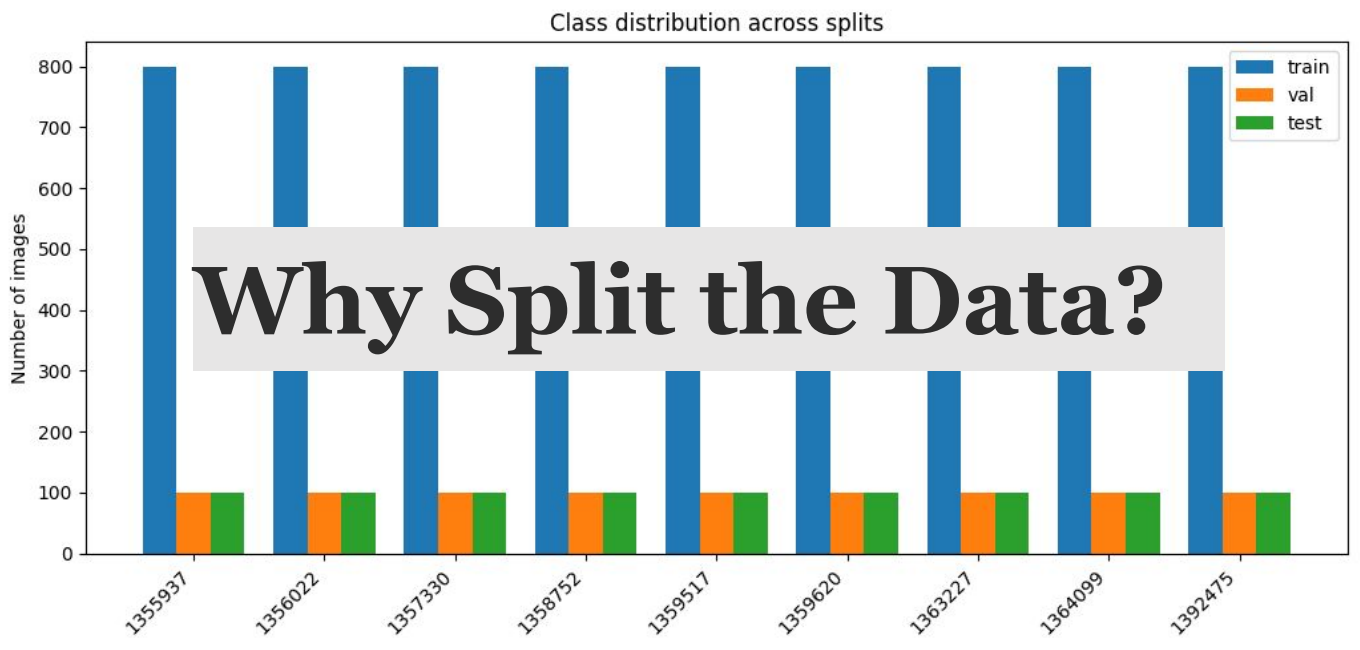
Ecoweek subset

Classes	9 species
Train	800 img/class
Val	100 img/class
Test	100 img/class
Resolution	224×224



The Dataset — Plant Species

Real plant images. Your task: classify them by species ID.



Why Split the Data?

A single image cannot be both 'new for training' and 'reserved for evaluation'.

Train

Learn model parameters
(weight updates via backprop)

Validation

Tune hyperparameters,
monitor overfitting

Test

Final, unbiased
performance estimate

Golden rule: the test set must NEVER influence any modeling decision. That is the validation set's job.

How the Data is Organised — ImageFolder

One folder per class, pre-split into train / val / test

```
ecoweek/
  train/
    sp001/
      img001.jpg
      img002.jpg
    sp002/
      ...
  val/
    sp001/
      ...
  test/
    ...
```

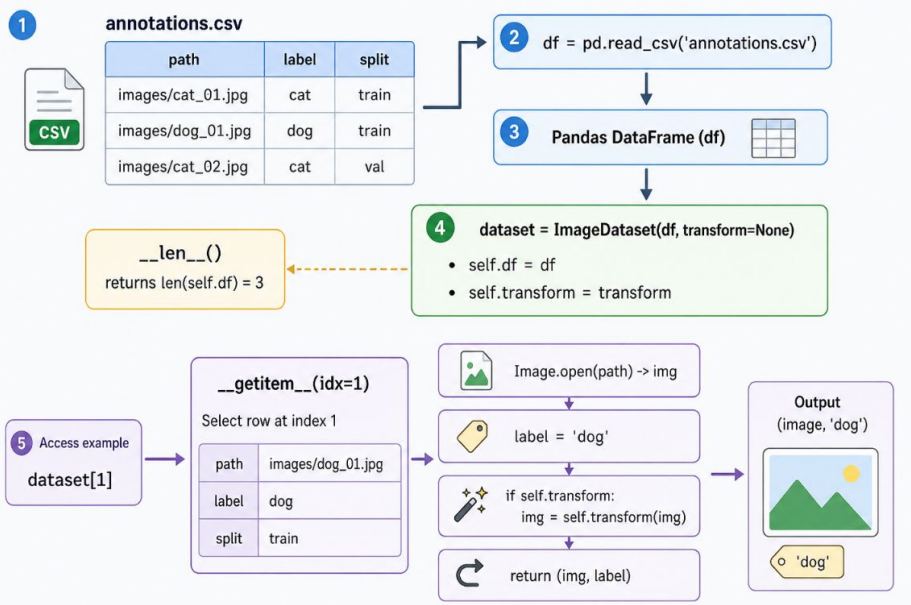
PYTORCH

```
torchvision import datasets, transforms

train_dataset = datasets.ImageFolder(
    root='ecoweek/train',
    transform=train_transform
)
val_dataset = datasets.ImageFolder(
    root='ecoweek/val',
    transform=eval_transform
)
# classes auto-detected from folder names
# label = folder index
```

How the Data is Organised — Alternative

What If You Start From a CSV?

Custom Dataset + DataFrame



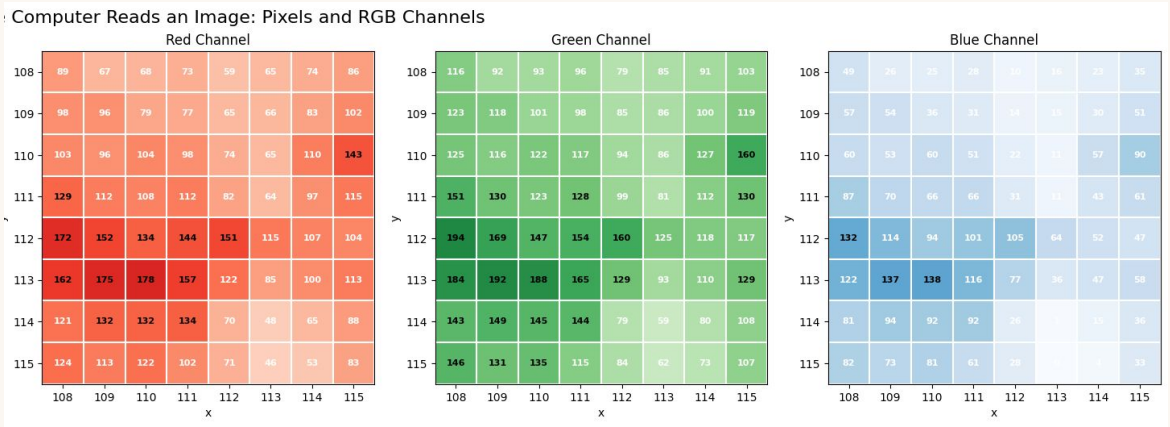
PYTORCH

```
import pandas as pd
from torch.utils.data import Dataset

df = pd.read_csv('annotations.csv')
# df: path | label | split

class ImageDataset(Dataset):
  def __init__(self, df, transform=None):
    self.df = df
    self.transform = transform
  def __len__(self):
    return len(self.df)
  def __getitem__(self, idx):
    img = Image.open(self.df.iloc[idx].path)
    label = self.df.iloc[idx].label
    if self.transform:
      img = self.transform(img)
    return img, label
```

Preparing the Ingredients — Transforms

Neural networks expect fixed-size tensors — not raw JPEG or PNG files

Raw image



The Preprocessing Pipeline

From Image to Tensor

Raw image (any size)
JPEG file from dataset, e.g. 1200×900×3

→

Resize (224)
Shrink to 224×224 pixels

→

ToTensor
Convert to float32 tensor [0.0, 1.0]

→

DataLoader
Group into batches of 32 for GPU

PYTORCH

```
eval_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])
```

Resize: neural networks expect tensors with the same shape within a batch. 224×224 is a common standard.

ToTensor: converts the PIL image (uint8, 0–255) to a float32 tensor scaled to [0, 1]. Also reorders dimensions from H×W×C to C×H×W.

Training transforms add randomness (augmentation). Val/test transforms are deterministic.

The Dataset — Plant Species

Each image = a tensor of shape [3, 224, 224].
torch.Size([32, 3, 224, 224]) = one batch of 32 images.

PYTORCH

```
train_transform = transforms.Compose([
    transforms.Resize(image_size),
    transforms.ToTensor(),
])

train_dataset = datasets.ImageFolder(
    os.path.join(data_dir, 'train'),
    transform=train_transform
)
train_loader = DataLoader(train_dataset,
    batch_size=32, shuffle=True)
```

Data Augmentation

Why ?

The model risks **memorizing** instead of learning.

Augmentation creates virtually infinite training data from the same images.

Augmentation is applied to TRAIN only — val and test must be deterministic for a fair evaluation.

Data Augmentation

PYTORCH

```
augmented_train_transform = transforms.Compose([
    transforms.Resize(image_size),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.1),
    transforms.ToTensor(),
])
```

Augmentation is applied to TRAIN only — val and test must be deterministic for a fair evaluation.

Summary - Inspect Before Cooking

1. Visualise a batch

→ Are classes visually distinct?
→ Any corrupt or mislabelled images?
→ What does augmented data look like?

1. Check class distribution

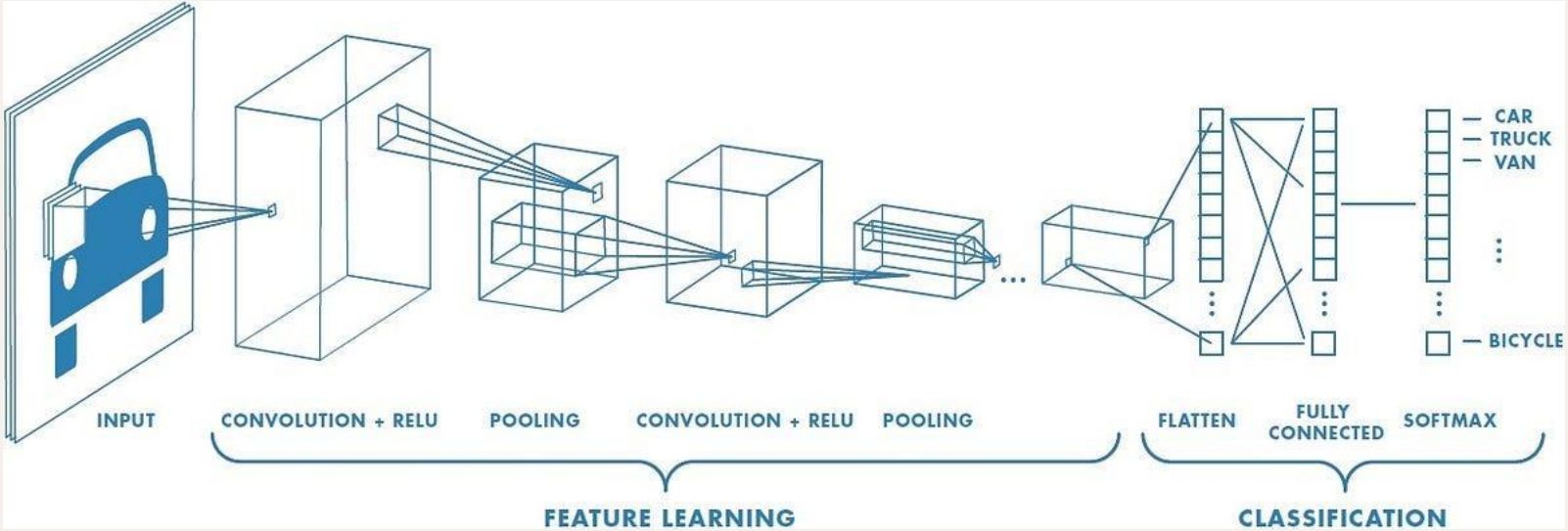
→ Is the data balanced or skewed?
→ Imbalanced → model biased toward majority
→ Fix: class weights, oversampling, WeightedRandomSampler

Time to Code!

Open the notebook and follow along.

- ✓ GPU runtime enabled (Runtime → Change runtime type → T4 GPU)
- ✓ Run each section in order — don't skip ahead
- ✓ Compare results at each stage
- ✓ Ask questions — that's what we're here for

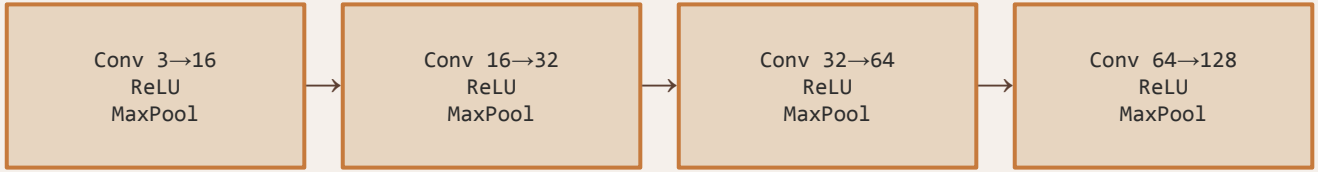
A Simple CNN — The Architecture



Understanding of Convolutional Neural Network (CNN) — Deep Learning | by Prabhu Raghav | Medium

A Simple CNN — The Architecture

Feature extractor



PYTORCH

Building the CNN

```
# ------------------------------------------------------
# Feature extractor: convolutions + pooling
# ------------------------------------------------------

features = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),

    nn.Conv2d(16, 32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),

    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),

    nn.Conv2d(64, 128, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
)
)
```

- **Conv2d:** slides a 3×3 filter across the image and the padding adds a border around the image to preserve its spatial size after convolution.
- **MaxPool2d:** reduces the spatial size of the image by keeping only the maximum value in each 2×2 region, making the representation smaller while preserving the strongest features.
- **ReLu:** activation function that keeps positive values and replaces negative values with zero, helping the neural network learn non-linear patterns.

The Convolution Operation

A small filter slides over the image, computing a weighted sum at each position.

Input image (4×4)

3	1	2	0
2	4	3	1
1	2	5	2
0	1	3	4

Filter (3×3)

1	0	-1
2	0	-2
1	0	-1

(edge detector)

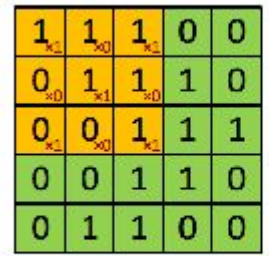
⊛

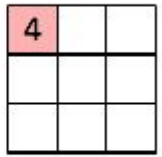
Feature map

4	-2	3
-1	5	-3
2	-4	1

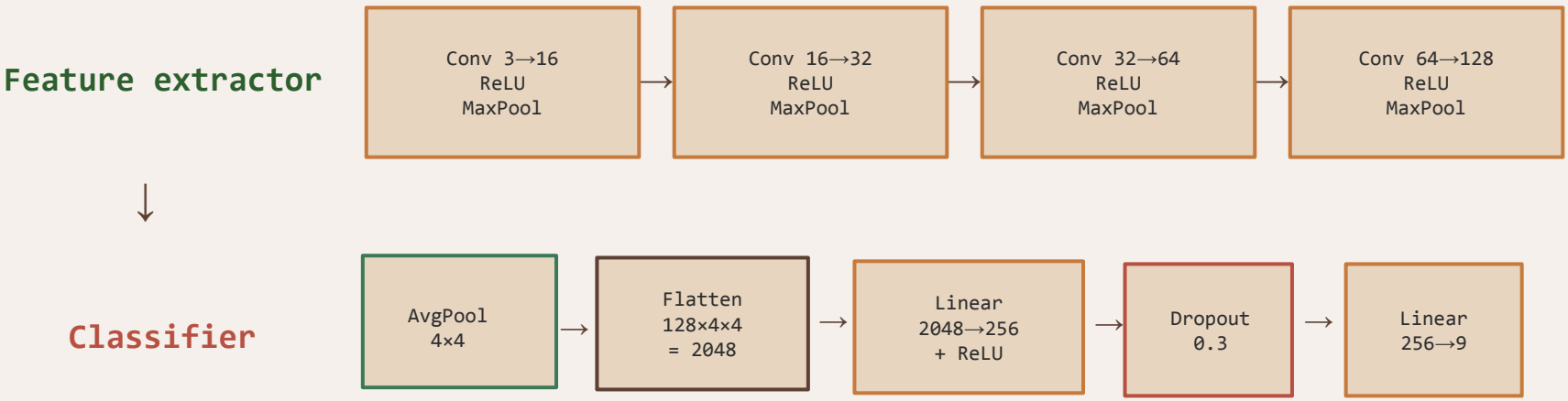
Slide the filter across the entire image:

- Each output position = how much the filter PATTERN matches the LOCAL image patch.
- Same filter weights everywhere → "weight sharing". An edge detector works at any position.
- Stack many filters (different patterns) → output depth = number of filters.





A Simple CNN — The Architecture



PYTORCH

Building the CNN

```
# ----------------------------------------------------------
# Classification head: pooling + linear layers
# ----------------------------------------------------------
classifier = nn.Sequential(
    nn.AdaptiveAvgPool2d((4, 4)),
    nn.Flatten(),
    nn.Linear(128 * 4 * 4, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, num_classes),
)

# ----------------------------------------------------------
# Full model: features → classifier
# ----------------------------------------------------------
model = nn.Sequential(features, classifier).to(device)

print(model)
```

- **AdaptiveAvgPool2d**: Resizes each feature map to a fixed 4×4 size by averaging values, so the classifier receives a consistent input size.
- **Flatten** converts the 2D feature maps into a 1D vector that can be passed to fully connected layers.
- **nn.Linear(128 * 4 * 4 = 2048, 256)** This fully connected layer takes the flattened feature vector and learns a compact 256-dimensional representation.
- **Dropout:** Dropout randomly deactivates 30% of neurons during training to reduce overfitting.

From Scratch to Shortcuts

You already know everything. PlantNet just uses built-in tools for the same ideas.

Concept	TP Simple	TP PlantNet
Load data	`class CustomDataset(Dataset)` you write `__getitem__`	`ImageFolder(root)` auto-loads from folders
Batching	`DataLoader(dataset, batch_size=32)`	`DataLoader` (identical!)
Preprocessing	(nothing – 2D points)	`transforms.Compose([` `Resize, ToTensor, Normalize])`
Model	`class Model(nn.Module)` you write `forward()`	`nn.Sequential(Conv2d, ReLU, ...)` no class needed
Loss	`torch.mean((y - pred)**2)` MSE by hand	`nn.CrossEntropyLoss()`
Training loop	`forward > backward > step`	`forward > backward > step` (identical!)
Evaluation	`accuracy = (pred == y).mean()`	`classification_report` `confusion_matrix`

SECTION 05

Tasting — Evaluation

Test set · Predictions · Confusion matrix · Classification report

The Final Tasting — Evaluating the Model

Inspecting concrete predictions: true label, predicted label, confidence score.

Example Predictions

What to look for

Correct + confident (0.90+):
model is sure and right.

Correct + low confidence (0.32):
right answer, but uncertain.

Wrong predictions:
which class gets confused?

From the output

Some images show True=1355937 but Pred=1392475 with Conf=0.36.

Low confidence + wrong → the model is genuinely uncertain.

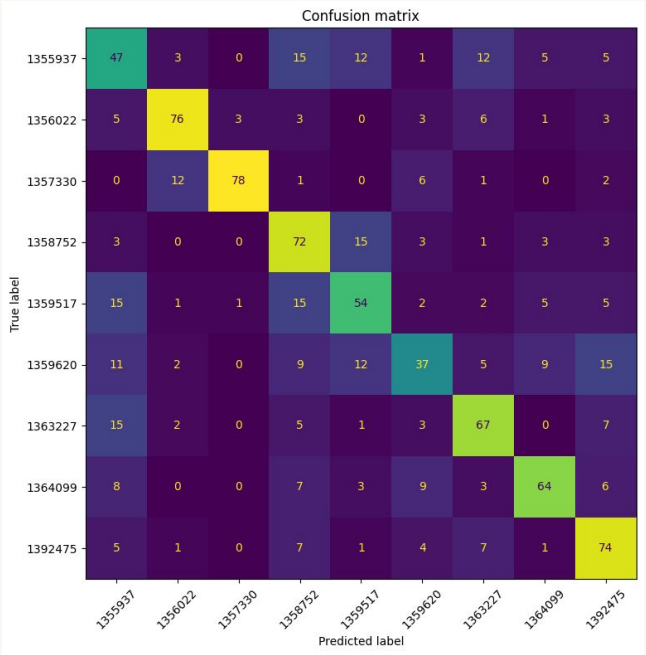
High confidence + right → the model has learned that species well.

The Final Tasting — Evaluating the Model

Confusion matrix

Diagonal = correct predictions.
Off-diagonal = errors (confusions).

Which species pairs get confused? That tells you what the model finds hard — often visually similar species.



The Final Tasting — Evaluating the Model

Accuracy

`correct / total`

Correct predictions / total
Simple but misleading when classes are imbalanced.

Precision

`TP / (TP + FP)`

Of all predicted Positives, how many were truly Positive?
Focus: avoiding false alarms.

Recall

`TP / (TP + FN)`

Of all actual Positives, how many did the model find?
Focus: not missing real cases.

F1-Score

`2·P·R / (P+R)`

Harmonic mean of Precision & Recall.
Balanced metric for imbalanced data.

PYTHON

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report
print(classification_report(y_true, y_pred, target_names=classes))
```

Time to Code!

Open the notebook and follow along.

- ✓ GPU runtime enabled (Runtime → Change runtime type → T4 GPU)
- ✓ Run each section in order — don't skip ahead
- ✓ Compare results at each stage
- ✓ Ask questions — that's what we're here for

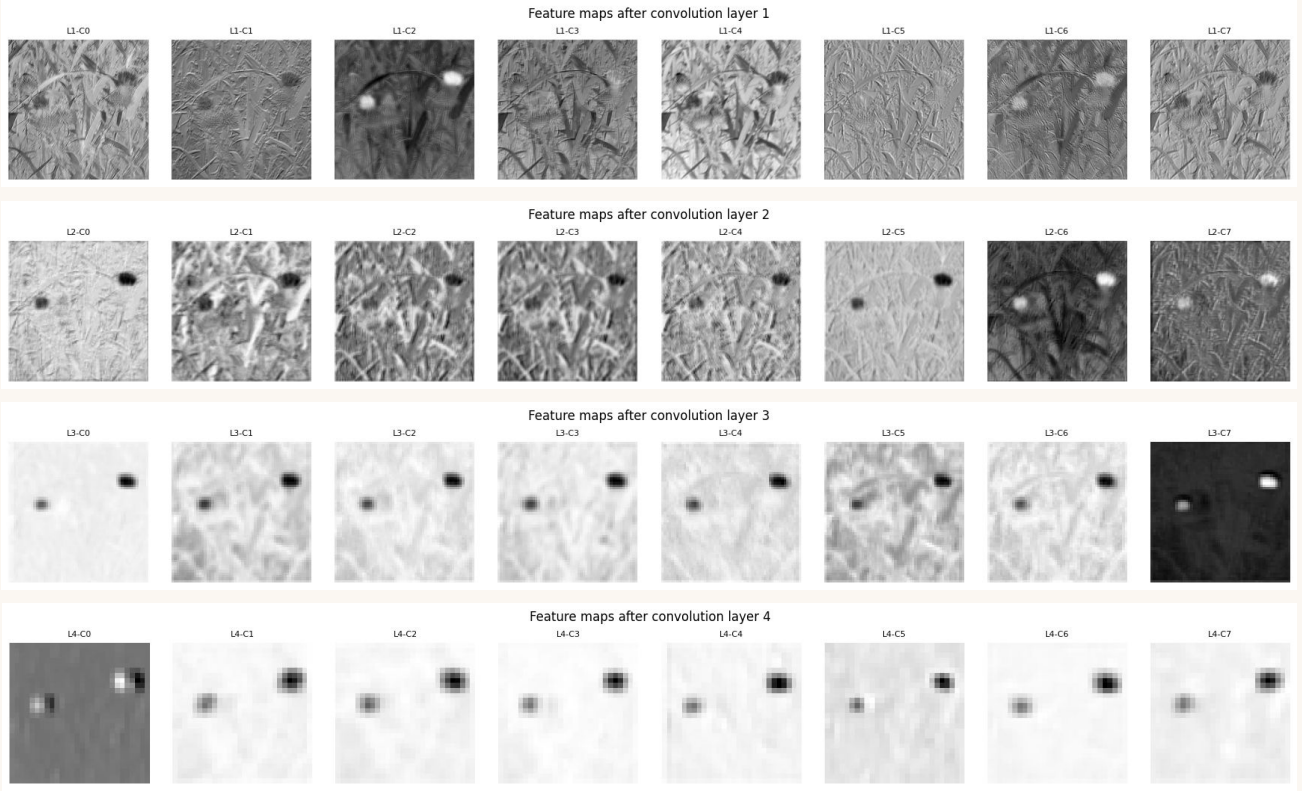
SECTION 06

Inside the Casseroles

Feature maps · Embedding visualisation with PCA

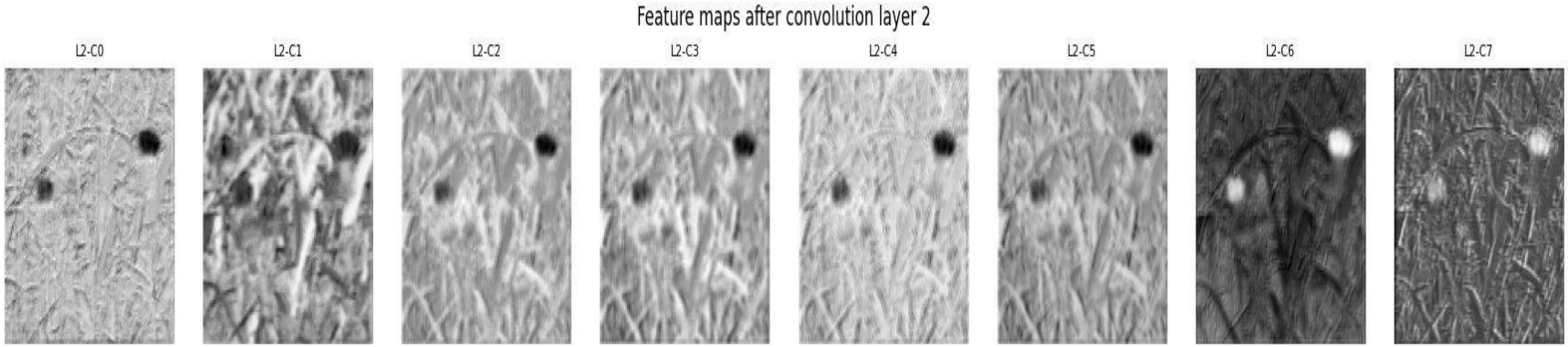
Inspecting Feature Maps

What does the CNN see inside?



Inspecting Feature Maps

What does the CNN see inside? Intermediate activations after convolution layer 2.



What we see

Each channel in Conv2 detects different patterns: some highlight edges, others respond to textures or shapes.

Note how the flower (dark blob) is detected differently across channels.

Early vs. deep layers

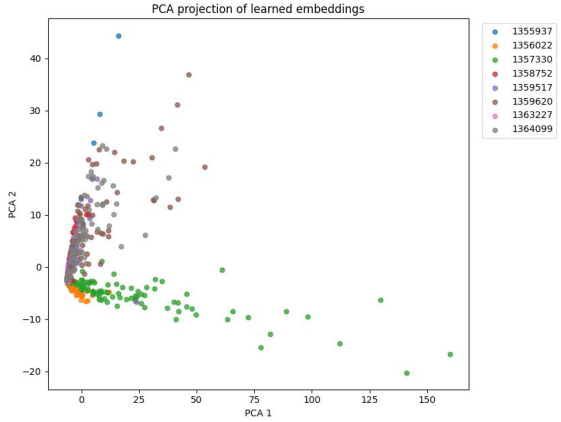
Early layers (Conv1-2): simple patterns close to raw pixels — edges, gradients.

Deep layers (Conv3-4): abstract, class-relevant features — object parts, shapes. No longer recognizable as an image.

Embedding Visualization — PCA

Project the CNN's internal representations to 2D. If classes cluster, the model learned structure.

The notebook produces a PCA scatter plot of the embeddings extracted before the final classifier. Each dot = one image, colored by class.



What to look for

Separated clusters = the model learned meaningful representations.

Overlapping clusters = those species are hard to distinguish.

Compare: scratch model vs. pre-trained

This is visual proof that transfer learning works.

Compare PCA plots: scratch model (no clusters) vs. pre-trained model (clear separation). Visual proof that transfer learning works.

SECTION 07

Standing on Giants — Transfer Learning

ResNet · Pretrained models · Fine-tuning with timm

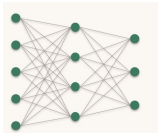
Transfer Learning: Reusing a Pre-trained Model

Training from scratch is informative, but often inefficient when the dataset is small.

1 Large source dataset

ImageNet
1.2M images
1000 classes

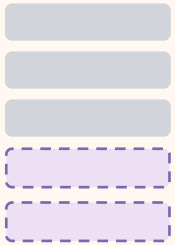
2 Pre-trained model learns features



edges, textures
shapes, objects

3 Reuse backbone

4 Replace classifier



New classes

5 Train on our dataset

Model adapted to our task

Key idea:

Reuse features from a model trained on millions of images. Adapt only the final layers to your specific task.

Transfer Learning: Reusing a Pre-trained Model

From scratch (random weights)

```
model = timm.create_model('resnet18')
# All 11.7M parameters are random
# Must learn everything from zero
```

Must learn: edges → textures → parts → species.
With ~6,400 train images? Hard.

Pre-trained + fine-tune

```
pretrained_model = timm.create_model(
  'resnet18',
  pretrained=True,    # ImageNet weights
  num_classes=9       # replace FC head
)
```

Conv layers already know visual features.
Only the FC head needs training.

What timm does under the hood:

1- Downloads pre-trained weights (trained on 1.2M ImageNet images, 1000 classes)

2- Loads them into the ResNet18 architecture (conv1 + 4 blocks + avgpool)

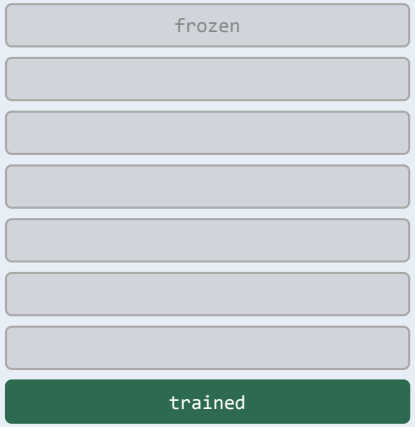
3- Replaces the last FC layer: nn.Linear(512, 1000) → nn.Linear(512, 8)

4 - The new FC layer has random weights — everything else is frozen/fine-tuned

Three Ways to Transfer

Feature Extraction

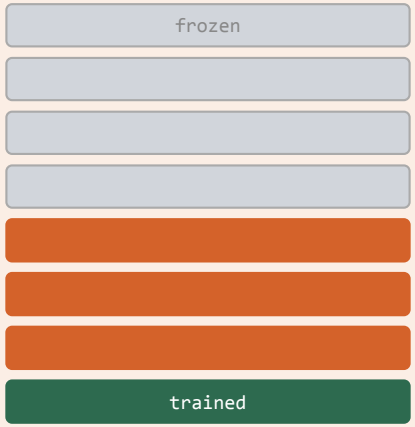
Freeze everything
Train only last layer



Small data, similar domain

Partial Fine-Tuning

Freeze early layers
Unfreeze last blocks



Medium data

Full Fine-Tuning

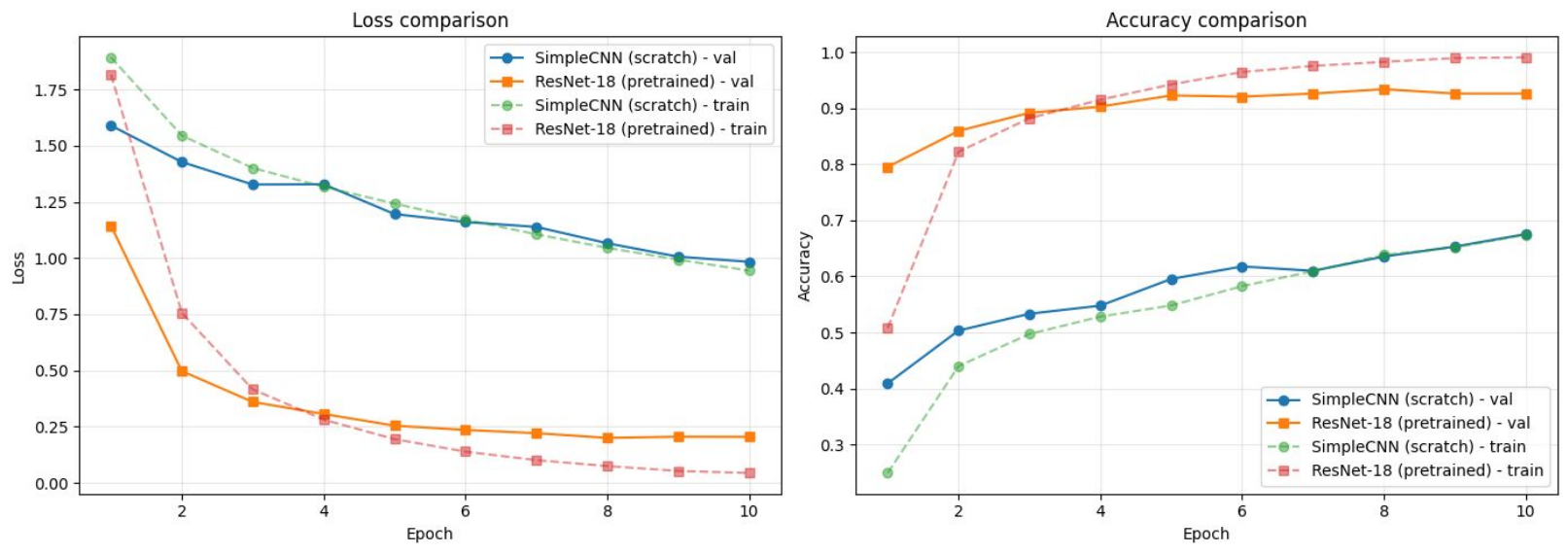
All layers trained
Small learning rate

Large data, different domain

= frozen (kept from ImageNet) = unfrozen (retrained) = new classifier layer

Scratch vs. Pre-trained — Results

> *What do you observe? What do you deduce? Compare loss curves, accuracy, convergence speed.*



PYTORCH

Transfer Learning in 5 Lines

```
import timm

# 1. Load pretrained ResNet-18
model = timm.create_model(
    'resnet18',
    pretrained=True,
    num_classes=1000)

# 2. Replace classifier head
model.fc = nn.Linear(512, num_species)

# 3. Feature extraction: freeze backbone
for p in model.parameters():
    p.requires_grad = False

# 4. Unfreeze only the head
for p in model.fc.parameters():
    p.requires_grad = True

# 5. Train with small lr
optimizer = optim.Adam(model.fc.parameters(), lr=1e-4)
```

Time to Code!

Open the notebook and follow along.

- ✓ GPU runtime enabled (Runtime → Change runtime type → T4 GPU)
- ✓ Run each section in order — don't skip ahead
- ✓ Compare results at each stage
- ✓ Ask questions — that's what we're here for

SECTION 08

BONUS

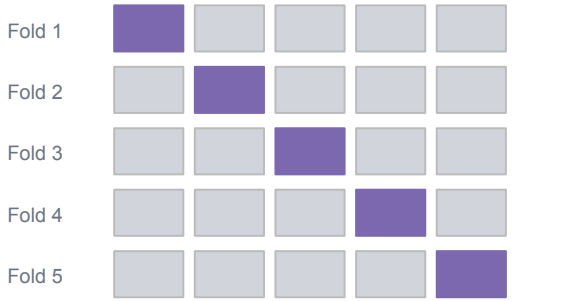
Top-K · Average-K · Conformal prediction

Train / Val / Test & Cross-Validation

1 Why split the data?

Set	Purpose	When
Train	Learn weights	Every epoch
Val	Tune & monitor	Each epoch (no grad)
Test	Final score	Once at the end

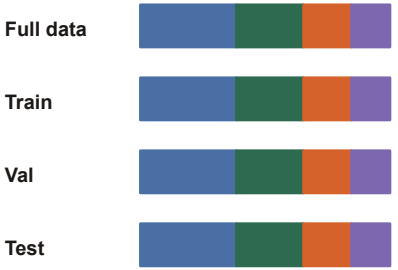
2 K-Fold Cross-Validation



Rotate the val fold each time.
Average scores across K runs.
More robust but slower.

3 Stratified splits

Each split keeps the same class proportions.



ecoweek dataset has pre-defined splits

Train / Val / Test & Cross-Validation

PYTORCH

```
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
    fold_train = Subset(train_dataset, train_idx)
    fold_val   = Subset(train_dataset, val_idx)

    model = timm.create_model('resnet18', pretrained=True,
                              num_classes=num_classes).to(device)
    # train + evaluate on this fold...
```