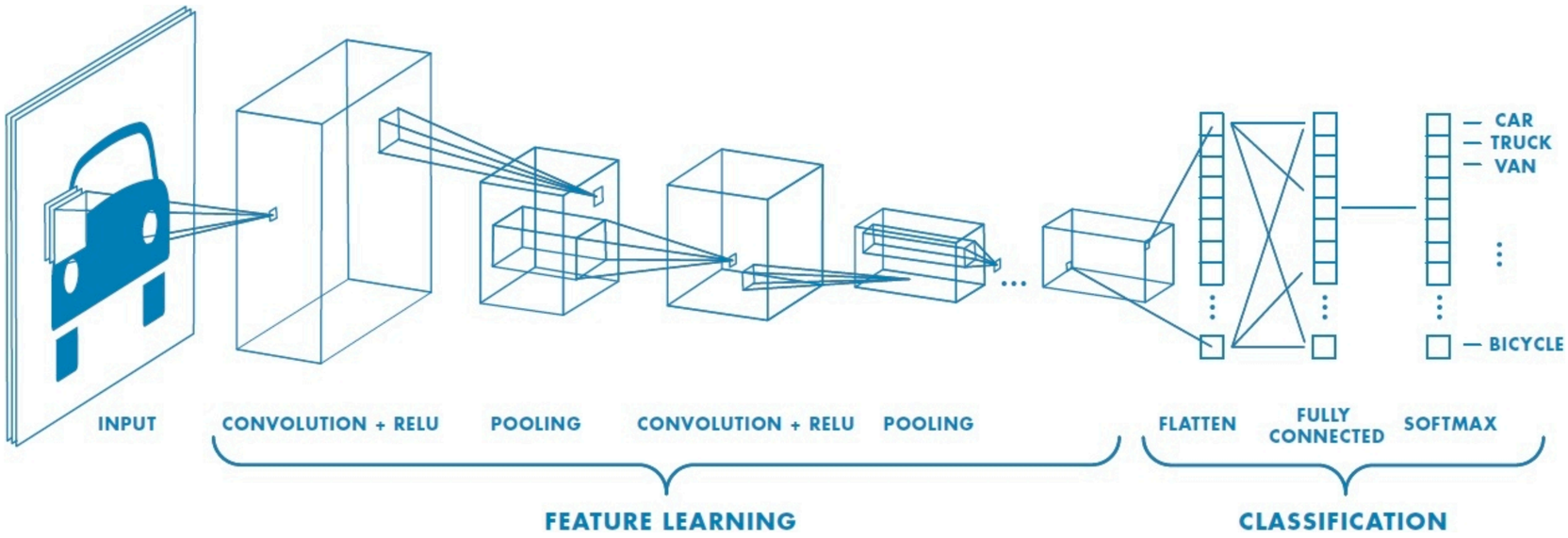
Convolutional Neural Networks

Transfer learning, Object Detection, ...

Rafał Nowak April 15, 2025

Convolutional Neural Networks



https://www.mathworks.com/discovery/convolutional-neural-network.html

Le-Net 1998

- Y. LeCun et al., Gradient-based learning applied to document recognition, 1998
- Average pooling
- Sigmoid + tanh
- Dense Top Layers
- MNIST (60k) dataset

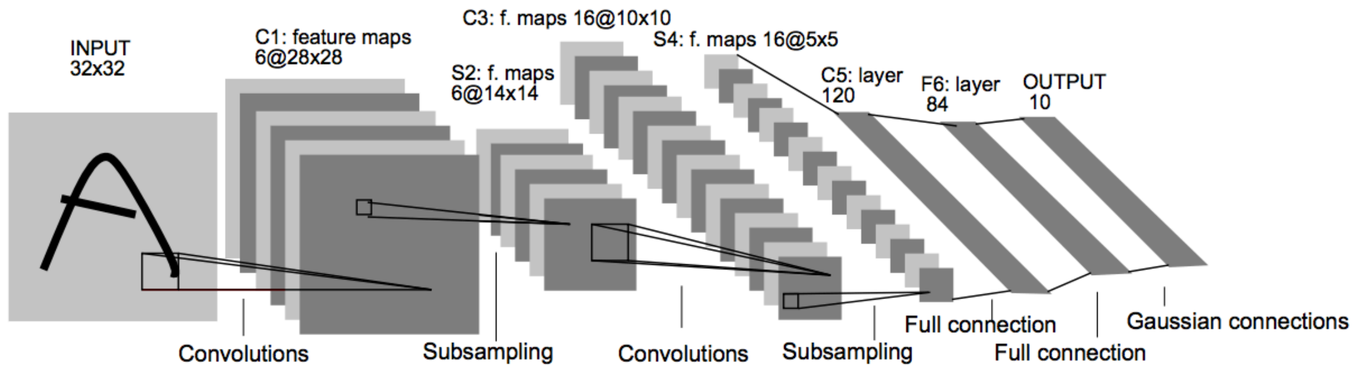


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Some CNN architectures

- LeNet-5
- AlexNet
- VGGNet
- GoogLeNet
- ResNet
- and some others like: ResNeXt, DenseNet, SqueezeNet, MobileNet, etc

torchvision

models

- torchvision.models provides a number of **pre-trained** models
- These models have been trained on the ImageNet dataset
- The models can be used for classification, segmentation, detection, etc.
- The models can be used for transfer learning
- see https://pytorch.org/vision/stable/models.html

Classification

The following classification models are available, with or without pre-trained weights:

- AlexNet
- ConvNeXt
- DenseNet
- EfficientNet
- EfficientNetV2
- GoogLeNet
- Inception V3
- MaxVit
- MNASNet
- MobileNet V2
- MobileNet V3
- RegNet
- ResNet
- ResNeXt
- ShuffleNet V2
- SqueezeNet
- SwinTransformer
- VGG
- VisionTransformer
- Wide ResNet

Table of all available classification weights

Accuracies are reported on ImageNet-1K using single crops:

Weight	Acc@1	Acc@5	Params	GFLOPS	Recipe
AlexNet_Weights.IMAGENET1K_V1	56.522	79.066	61.1M	0.71	link
ConvNeXt_Base_Weights.IMAGENET1K_V1	84.062	96.87	88.6M	15.36	link
ConvNeXt_Large_Weights.IMAGENET1K_V1	84.414	96.976	197.8M	34.36	link
ConvNeXt_Small_Weights.IMAGENET1K_V1	83.616	96.65	50.2M	8.68	link
ConvNeXt_Tiny_Weights.IMAGENET1K_V1	82.52	96.146	28.6M	4.46	link
DenseNet121_Weights.IMAGENET1K_V1	74.434	91.972	8.0M	2.83	link
DenseNet161_Weights.IMAGENET1K_V1	77.138	93.56	28.7M	7.73	link
DenseNet169_Weights.IMAGENET1K_V1	75.6	92.806	14.1M	3.36	link
DenseNet201_Weights.IMAGENET1K_V1	76.896	93.37	20.0M	4.29	link
EfficientNet_B0_Weights.IMAGENET1K_V1	77.692	93.532	5.3M	0.39	link
EfficientNet_B1_Weights.IMAGENET1K_V1	78.642	94.186	7.8M	0.69	link
EfficientNet_B1_Weights.IMAGENET1K_V2	79.838	94.934	7.8M	0.69	link
EfficientNet_B2_Weights.IMAGENET1K_V1	80.608	95.31	9.1M	1.09	link
EfficientNet_B3_Weights.IMAGENET1K_V1	82.008	96.054	12.2M	1.83	link
EfficientNet_B4_Weights.IMAGENET1K_V1	83.384	96.594	19.3M	4.39	link
EfficientNet_B5_Weights.IMAGENET1K_V1	83.444	96.628	30.4M	10.27	link
EfficientNet_B6_Weights.IMAGENET1K_V1	84.008	96.916	43.0M	19.07	link
EfficientNet_B7_Weights.IMAGENET1K_V1	84.122	96.908	66.3M	37.75	link
EfficientNet_V2_L_Weights.IMAGENET1K_V1	85.808	97.788	118.5M	56.08	link
EfficientNet_V2_M_Weights.IMAGENET1K_V1	85.112	97.156	54.1M	24.58	link
EfficientNet_V2_S_Weights.IMAGENET1K_V1	84.228	96.878	21.5M	8.37	link
GoogLeNet_Weights.IMAGENET1K_V1	69.778	89.53	6.6M	1.5	link
Inception_V3_Weights.IMAGENET1K_V1	77.294	93.45	27.2M	5.71	link
MNASNet0_5_Weights.IMAGENET1K_V1	67.734	87.49	2.2M	0.1	link
MNASNet0_75_Weights.IMAGENET1K_V1	71.18	90.496	3.2M	0.21	link
MNASNet1_0_Weights.IMAGENET1K_V1	73.456	91.51	4.4M	0.31	link
MNASNet1_3_Weights.IMAGENET1K_V1	76.506	93.522	6.3M	0.53	link
MaxVit_T_Weights.IMAGENET1K_V1	83.7	96.722	30.9M	5.56	link
MobileNet_V2_Weights.IMAGENET1K_V1	71.878	90.286	3.5M	0.3	link
MobileNet_V2_Weights.IMAGENET1K_V2	72.154	90.822	3.5M	0.3	link
MobileNet_V3_Large_Weights.IMAGENET1K_V1	74.042	91.34	5.5M	0.22	link
MobileNet_V3_Large_Weights.IMAGENET1K_V2	75.274	92.566	5.5M	0.22	link

CNNs

Overview & Best Practices

- What Are CNNs For?
    - Primarily used for image and video analysis, also effective in:
    - Object detection
    - Facial recognition
    - Image classification & segmentation
    - Medical imaging
- Work by learning spatial hierarchies through filters (kernels)

CNNs

Choosing the Kernel Size

- Common sizes: 3x3, 5x5, 7x7 (for very early layers)
- Best Practice: Use stacked 3×3 filters instead of a single large one
  - Stacked 3×3s preserve more non-linearity and reduce parameters
- Larger kernels can be used for aggressive downsampling or to capture more global context

CNNs

Choosing the Kernel Size

- Kernel count (filters per layer): start with 16–32, double in deeper layers
- Use stride 1 and padding=`same` to preserve spatial dimensions early on
- End with a Global Average Pooling + Dense layer for classification

CNNs

How Many Layers

- Depends on the task complexity:
  - Simple tasks: 3–10 layers
  - State-of-the-art models (ResNet, VGG, etc.): 20–100+ layers
- Deep networks use residual connections (ResNets) to avoid vanishing gradients

CNNs

Best Practices

- Start small: Begin with a shallow network, then scale
- ReLU activation is a standard; consider Leaky ReLU or ELU if dead neurons arise
- Use Batch Normalization to stabilize and accelerate training
- Dropout helps with overfitting (0.25–0.5 commonly used)
- Pooling layers (MaxPooling) reduce spatial size and computation
- Always monitor overfitting — use validation loss and early stopping
- Use data augmentation for image tasks (flipping, cropping, rotation)

Feature selector and classifier

Example: AlexNet

- ALEXNET

torchvision.models.alexnet(*, *weights: Optional[AlexNet_Weights] = None*, *progress: bool = True*, ***kwargs: Any*) → AlexNet [SOURCE]

AlexNet model architecture from One weird trick for parallelizing convolutional neural networks.

• NOTE

AlexNet was originally introduced in the ImageNet Classification with Deep Convolutional Neural Networks paper. Our implementation is based instead on the "One weird trick" paper above.

Parameters:

- **weights** (`AlexNet_Weights`, optional) – The pretrained weights to use. See `AlexNet_Weights` below for more details, and possible values. By default, no pre-trained weights are used.
- **progress** (*bool, optional*) – If True, displays a progress bar of the download to stderr. Default is True.
- ****kwargs** – parameters passed to the `torchvision.models.squeezenet.AlexNet` base class. Please refer to the source code for more details about this class.

see https://github.com/pytorch/vision/blob/main/torchvision/models/alexnet.py

```python
class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 1000, dropout: float = 0.5) -> None:
        super().__init__()
        _log_api_usage_once(self)
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Feature selector and classifier

Example: VGG

VGG16

torchvision.models.vgg16(*, *weights: Optional[VGG16_Weights] = None*, *progress: bool = True*, ***kwargs: Any*) → VGG [SOURCE]

VGG-16 from Very Deep Convolutional Networks for Large-Scale Image Recognition.

Parameters:

- **weights** (`VGG16_Weights`, optional) – The pretrained weights to use. See `VGG16_Weights` below for more details, and possible values. By default, no pre-trained weights are used.
- **progress** (*bool, optional*) – If True, displays a progress bar of the download to stderr. Default is True.
- ****kwargs** – parameters passed to the `torchvision.models.vgg.VGG` base class. Please refer to the source code for more details about this class.

VGG19

torchvision.models.vgg19(*, *weights: Optional[VGG19_Weights] = None*, *progress: bool = True*, ***kwargs: Any*) → VGG [SOURCE]

VGG-19 from Very Deep Convolutional Networks for Large-Scale Image Recognition.

Parameters:

- **weights** (`VGG19_Weights`, optional) – The pretrained weights to use. See `VGG19_Weights` below for more details, and possible values. By default, no pre-trained weights are used.
- **progress** (*bool, optional*) – If True, displays a progress bar of the download to stderr. Default is True.
- ****kwargs** – parameters passed to the `torchvision.models.vgg.VGG` base class. Please refer to the source code for more details about this class.

```python
class VGG(nn.Module):
    def __init__(
        self, features: nn.Module, num_classes: int = 1000, init_weights: bool = True, dropout:
float = 0.5
    ) -> None:
        super().__init__()
        _log_api_usage_once(self)
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            for m in self.modules():
                if isinstance(m, nn.Conv2d):
                    nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
                    if m.bias is not None:
                        nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.BatchNorm2d):
                    nn.init.constant_(m.weight, 1)
                    nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight, 0, 0.01)
                    nn.init.constant_(m.bias, 0)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Global Average Pooling (GAP) in CNNs

- What is GAP?
  - Global Average Pooling (GAP) replaces the flatten + fully connected layers
  - It reduces each feature map to a single scalar — by averaging all spatial values (H × W) in that channel
- Why Use GAP?
  - ✅ No trainable parameters
  - ✅ Reduces overfitting
  - ✅ More lightweight models
  - ✅ Makes the model spatially invariant
  - ✅ Common in architectures like GoogLeNet, ResNet, MobileNet

`[Batch, Channels, Height, Width]` → `[Batch, Channels]`

Each channel is averaged:

$$y_c = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} x_{c,i,j}$$

Global Average Pooling (GAP) in CNNs

```python
import torch.nn as nn

# Global Average Pooling 2D
gap = nn.AdaptiveAvgPool2d((1, 1))

# Apply and flatten
out = gap(x)              # [B, C, 1, 1]
out = out.view(out.size(0), -1)  # [B, C]
```

```python
self.classifier = nn.Sequential(
    nn.AdaptiveAvgPool2d((1, 1)),
    nn.Flatten(),  # Handles the flattening for you
    nn.Linear(128, num_classes)
)
```

Transfer learning

Possible scenarios

- Small dataset and similar to the original:
  - train only the (last) fully connected layer
- Small dataset and different to the original:
  - train only the fully connected layers
- Large dataset and similar to the original:
  - freeze the earlier layers (simple features) and train the rest of the layers

```
# Load the pretrained model from pytorch
vgg16 = models.vgg16_bn()
```

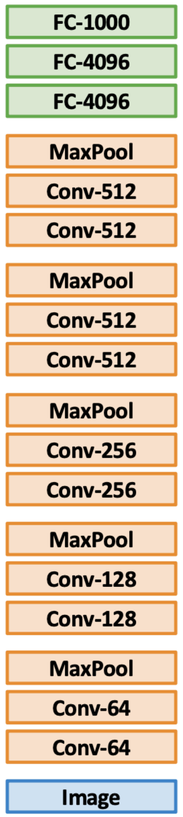
```
# Freeze training for all layers
for param in vgg16.features.parameters():
    param.require_grad = False
```

- Large dataset and different to the original:
  - train the model from scratch and reuse the network architecture (using the trained weights as a started point).

https://towardsdatascience.com/a-practical-example-in-transfer-learning-with-pytorch-846bb835f2db

Transfer Learning: Feature Extraction

1. Train on ImageNet

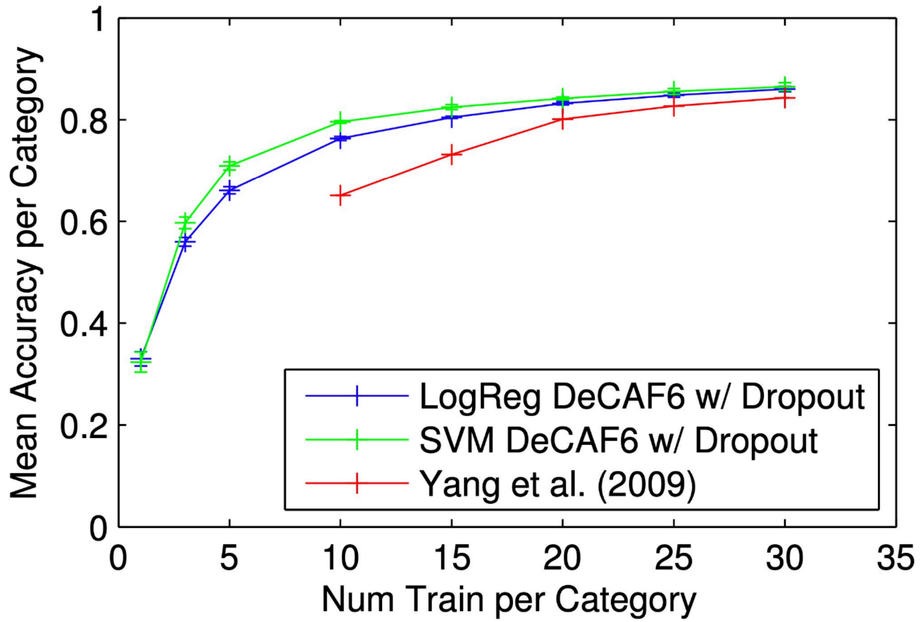


2. Extract features with CNN, train linear model

Remove last layer

Freeze these

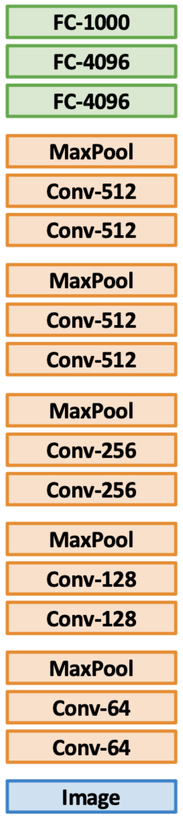
Classification on Caltech-101



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

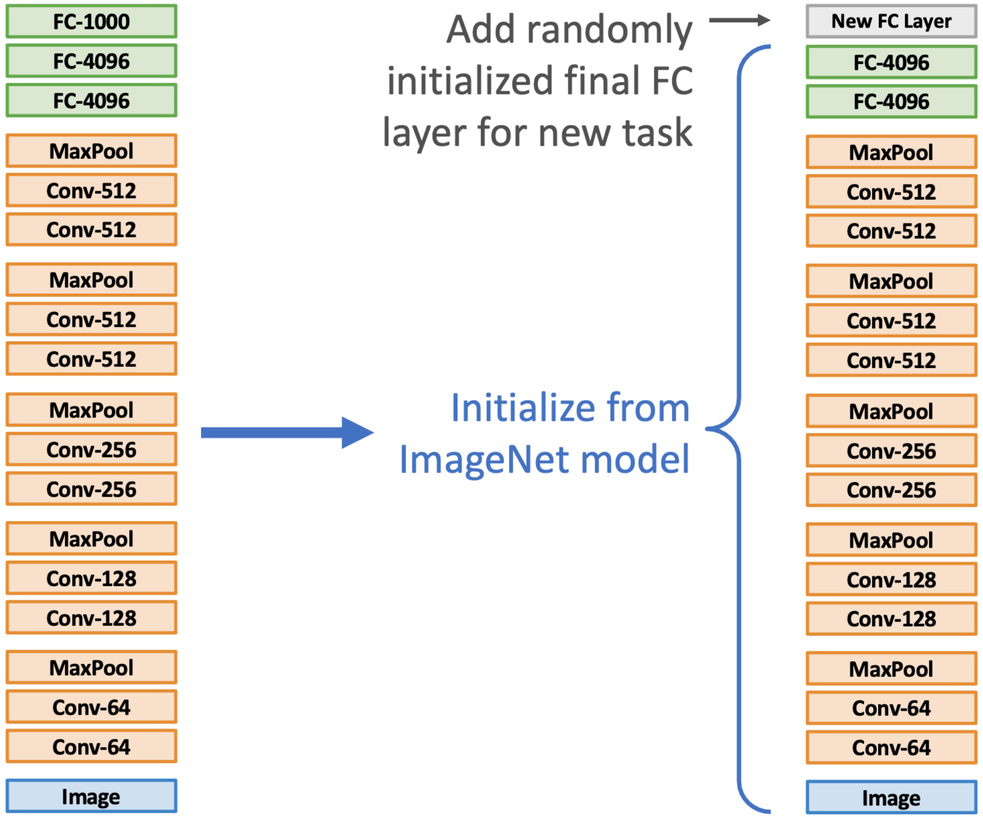
Transfer Learning: Fine-Tuning

1. Train on ImageNet



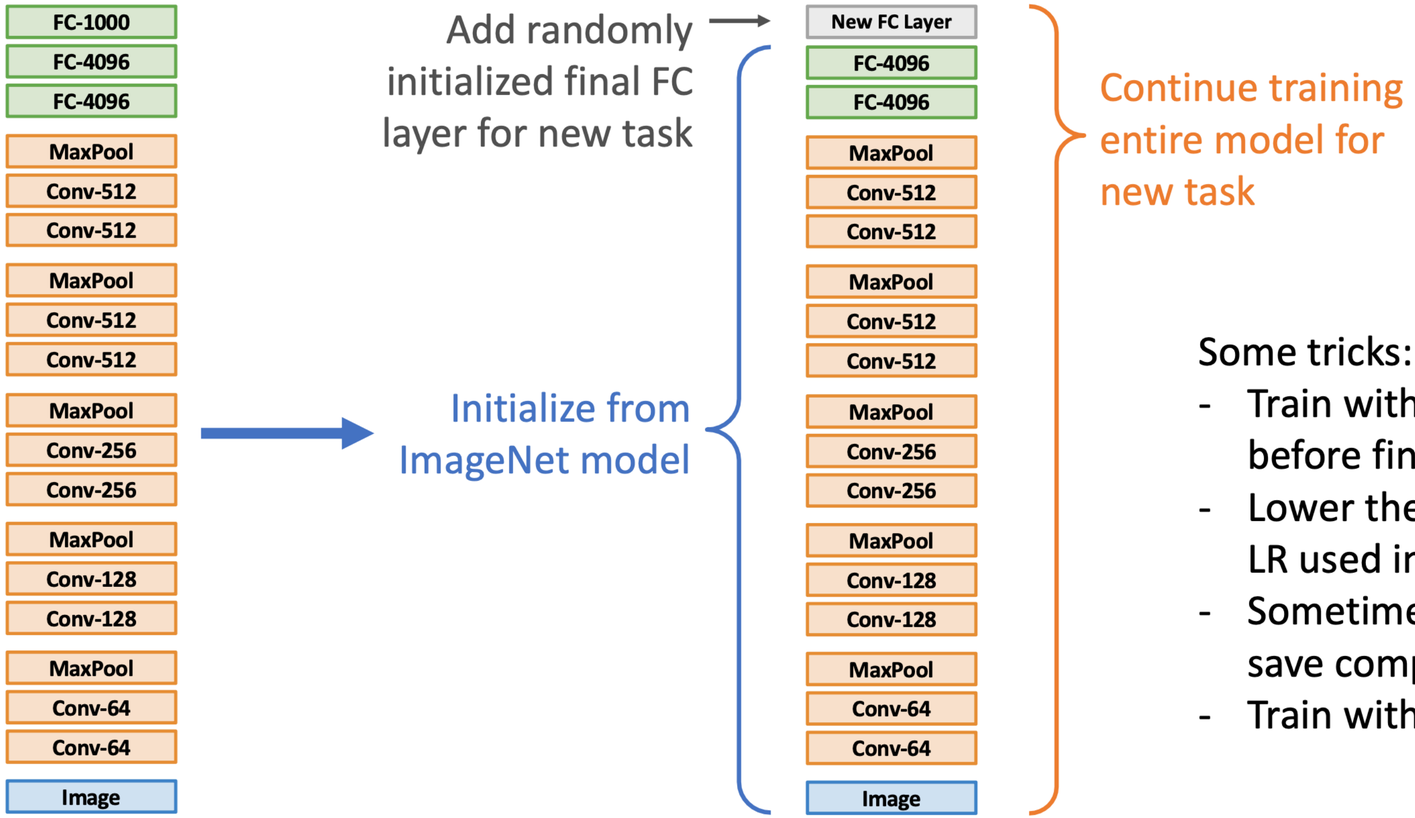
Transfer Learning: Fine-Tuning

1. Train on ImageNet



Transfer Learning: Fine-Tuning

1. Train on ImageNet

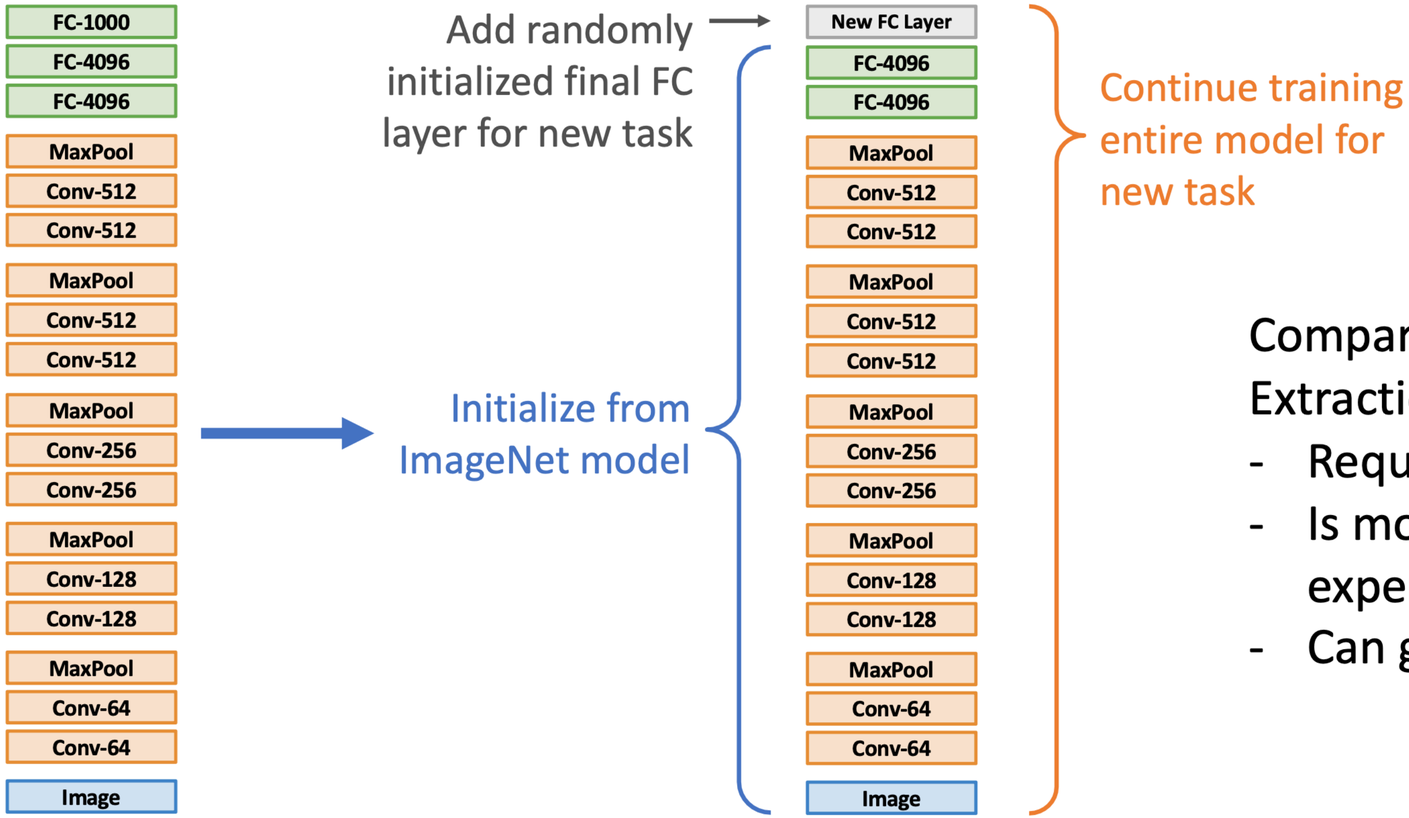


Some tricks:

- Train with feature extraction first before fine-tuning
- Lower the learning rate: use ~1/10 of LR used in original training
- Sometimes freeze lower layers to save computation
- Train with BatchNorm in “test” mode

Transfer Learning: Fine-Tuning

1. Train on ImageNet



Compared with Feature Extraction, Fine-Tuning:

- Requires more data
- Is more computationally expensive
- Can give higher accuracies

1x1 Convolutions

- 🤔 Not intuitive at first glance
- 🚀 Widely used in top architectures like:
- GoogLeNet (Inception)
- ResNet (bottleneck blocks)
- MobileNet (depthwise separable convs)
- 💡 Help CNNs become deeper, faster, and smarter

What is a 1x1 Convolution?

- A convolution with a 1×1 kernel
- Operates across channels, not spatially
- At each pixel:
- Takes a vector of size C_in
- Produces a vector of size C_out
  - Like a fully connected layer at each pixel

Conv 1x1

Key Applications

- ✅ Dimensionality Reduction
  → Reduce number of channels before expensive 3x3 or 5x5 convs
- ✅ Dimensionality Expansion
  → Increase representation power before merging with residuals
- ✅ Feature Mixing
  → Allows interaction across channels
- ✅ Non-linearity Insertion
  → Use with ReLU to learn richer representations

Conv 1x1

Use in ResNet (Bottleneck Block)

- 🧱 Structure:
  - 1x1 Conv (reduce)
  - 3x3 Conv (process)
  - 1x1 Conv (expand)
- 🔁 Skip connection added
- Why?
  - Reduce computation
  - Allow deeper networks to train effectively

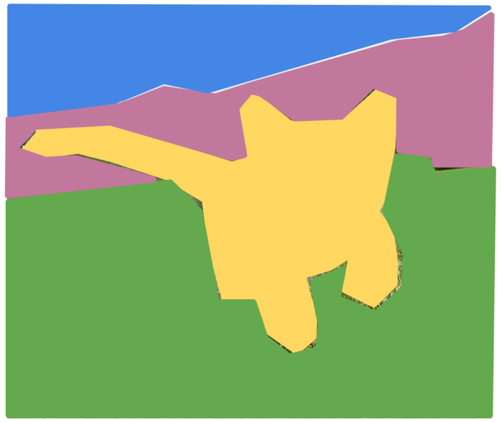
Computer Vision Tasks

Classification

CAT

No spatial extent

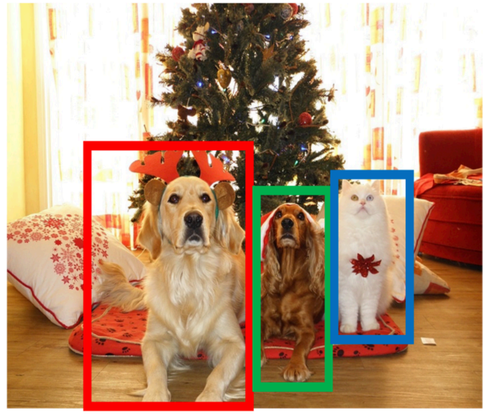
Semantic Segmentation



GRASS, CAT, TREE, SKY

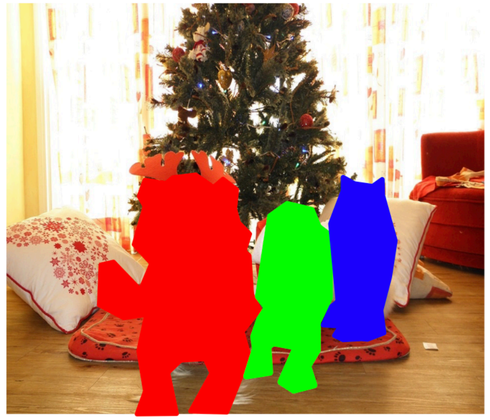
No objects, just pixels

Object Detection



DOG, DOG, CAT

Instance Segmentation



DOG, DOG, CAT

Multiple Objects

Object detection

- 🎯 Object Detection is a task where we:
  - **Locate objects** in an image (bounding boxes 📦)
  - **Classify** them into categories (e.g., "cat", "car", "person")
- 🧠 It's a mix of:
  - Image classification (what is it?)
  - Localization (where is it?)

Object detection

Output fomat

- For each detected object:
  - A **bounding box**: (x, y, width, height)
  - A **class label**
  - Often: a **confidence score**

```
[
  { "label": "dog", "bbox": [100, 200, 150, 250], "score": 0.97 },
  { "label": "ball", "bbox": [300, 120, 80, 90], "score": 0.87 }
]
```

Object detection

Applications

- Applications Across Industries:
  - 🔍 Surveillance (detect intruders)
  - 🚗 Autonomous Vehicles (detect other cars, pedestrians)
  - 📦 E-commerce (find products in photos)
  - 🧠 Robotics, AR, Smart Cameras

Object detection

Two-Stage vs One-Stage Detectors

- ⚡ One-Stage:
  - Examples: YOLO, SSD
  - Directly predict boxes and classes in one shot
- 🧪 Two-Stage:
  - Examples: R-CNN, Faster R-CNN
  - Step 1: Generate region proposals
  - Step 2: Classify and refine boxes
- 💡 Trade-off:
  - One-stage: faster, real-time
  - Two-stage: more accurate

Object detection

Evaluation Metrics

- **IoU** (Intersection over Union):
  - How well boxes overlap ground truth
- **mAP** (mean Average Precision):
  - How accurate predictions are across classes

IoU

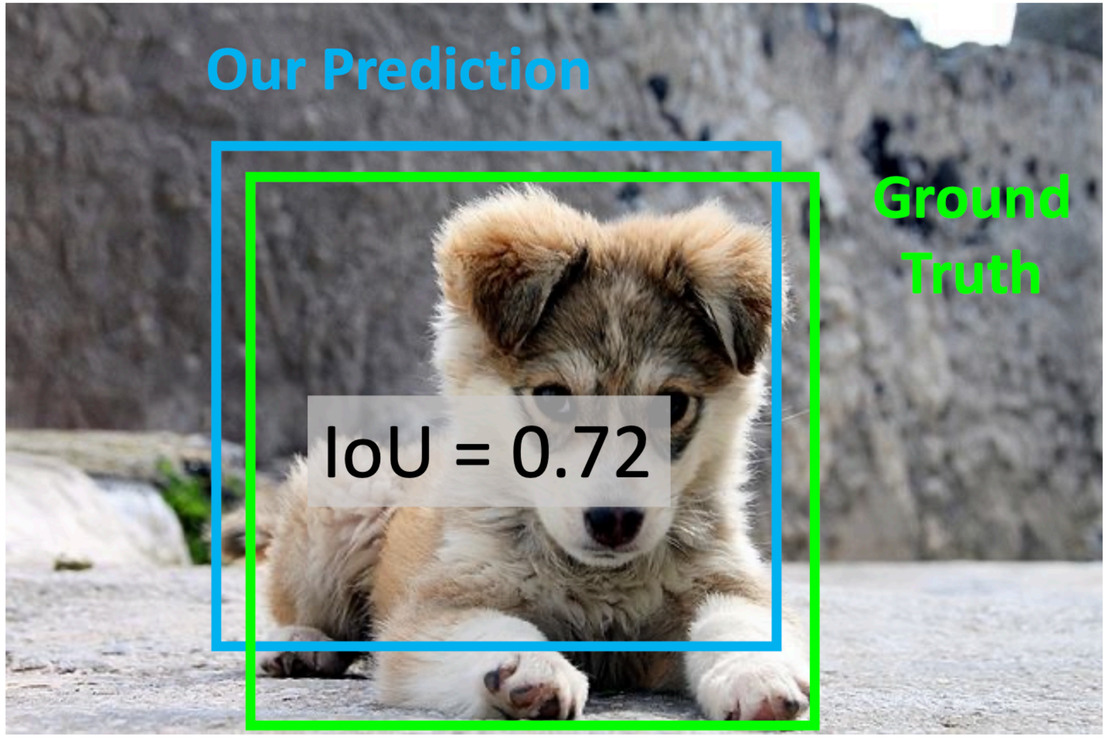
Comparing Boxes: Intersection over Union (IoU)

How can we compare our prediction to the ground-truth box?

Intersection over Union (IoU)
(Also called “Jaccard similarity” or “Jaccard index”):

$$\frac{\textit{Area of Intersection}}{\textit{Area of Union}}$$

IoU > 0.5 is “decent”,
IoU > 0.7 is “pretty good”,



NMS

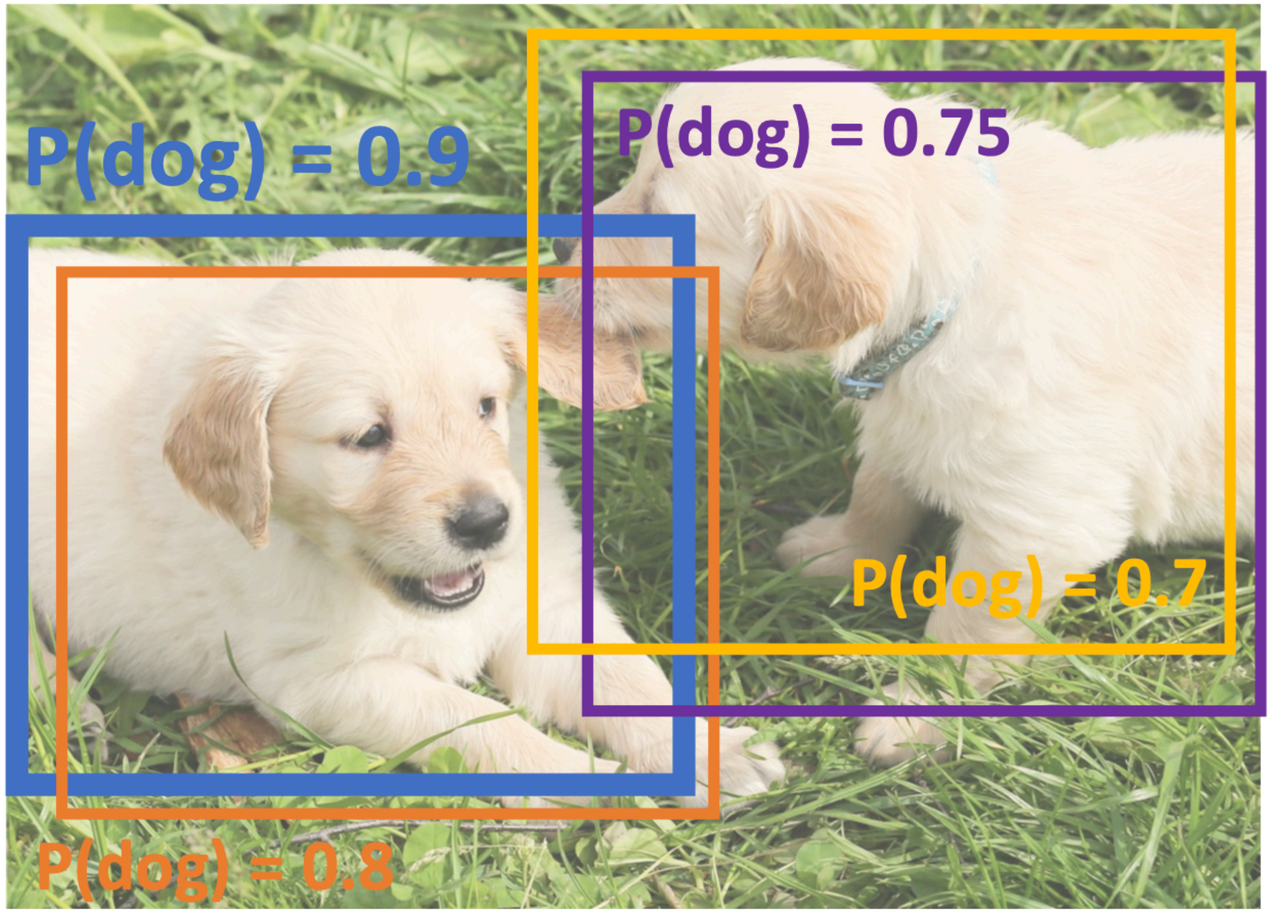
Overlapping Boxes: Non-Max Suppression (NMS)

Problem: Object detectors often output many overlapping detections:

Solution: Post-process raw detections using **Non-Max Suppression (NMS)**

1. Select next highest-scoring box
2. Eliminate lower-scoring boxes with IoU > threshold (e.g. 0.7)
3. If any boxes remain, GOTO 1

IoU(■, ■) = **0.78**
IoU(■, ■) = 0.05
IoU(■, ■) = 0.07



YOLO

You Only Look Once: Unified, Real-Time Object Detection

- https://arxiv.org/abs/1506.02640

You Only Look Once:
Unified, Real-Time Object Detection

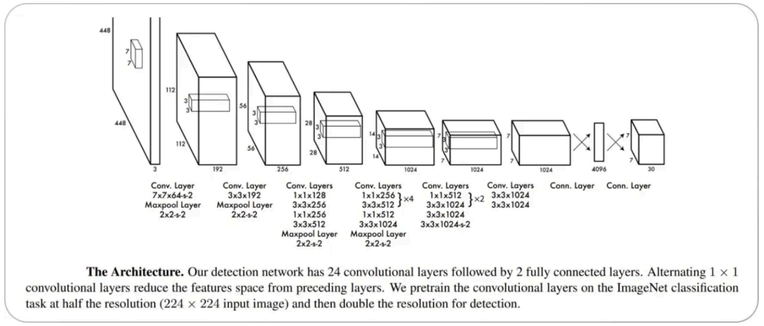
Joseph Redmon*, Santosh Divvala*†, Ross Girshick¶, Ali Farhadi*†
University of Washington*, Allen Institute for AI†, Facebook AI Research¶
http://pjreddie.com/yolo/

YOLO

You Only Look Once: Unified, Real-Time Object Detection

- ### How does YOLO work? YOLO Architecture

  The YOLO algorithm takes an image as input and then uses a simple deep convolutional neural network to detect objects in the image. The architecture of the CNN model that forms the backbone of YOLO is shown below.

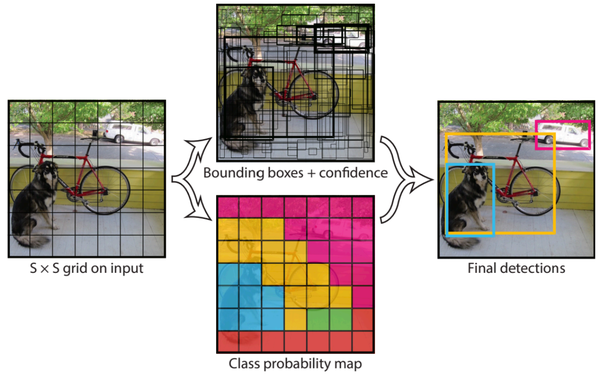
  

  The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

- https://medium.com/@v7.editors/yolo-algorithm-for-object-detection-explained-examples-90ccfd8a5642

YOLO

Object Detection Algorithm



- YOLO = You Only Look Once
- It's a one-stage object detection algorithm that treats object detection as a regression problem:
- From an image → directly to bounding boxes + class probabilities
- All in one forward pass of a neural network
- This makes it extremely fast, suitable for real-time applications.

YOLO

Object Detection Algorithm

- YOLO divides the input image into a grid — S × S (like 7x7 or 13x13).
- Each grid cell:
  - Predicts B bounding boxes
    - Each box: (x, y, w, h) + **confidence score**
  - Predicts **class probabilities** for the object it contains
- Final prediction per cell = B × (x, y, w, h, c_score) + C class scores

YOLO

Confidence

- For each predicted box:
  - (x, y) are offsets within the cell
  - (w, h) are relative to the whole image
- **Confidence score** $\approx$ IoU(box, ground truth) $\times$ P(object)
- Network learns to output ***confidence score***

YOLO

Inference

- Input image → CNN
- CNN outputs a tensor of predictions (bounding boxes + scores)
- Apply **confidence thresholding**
- Use **Non-Maximum Suppression** (NMS) to remove overlapping boxes
- Return final predictions

YOLO

Localization loss (bbox regression)

This part of the loss penalizes errors in the **predicted bounding box coordinates** (center `x`, `y`, width `w`, and height `h`) for boxes that are responsible for detecting an object.

Formally:

$$\lambda_{coord} * \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

- `𝟙_{ij}^{obj}` is 1 if the object exists in cell `i` and box `j` is responsible for prediction.
- Square roots of `w` and `h` are used to reduce the impact of large errors for bigger boxes.
- `λ_coord` is typically set to **5** to increase emphasis on localization accuracy.

YOLO

Classification loss

This penalizes the error in **class probability predictions** within each grid cell that contains an object.

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$$

- Only computed for cells with objects.
- Measures the difference between predicted class probability and ground truth.

YOLO

Confidence loss

This term measures how confident the network is about **object presence** in each grid cell.

- **For cells with objects**:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2$$

- **For cells without objects**:

$$\lambda_{noobj} * \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2$$

- `C_i` is the predicted confidence score.
- `λ_{noobj}` is usually **0.5** to reduce the influence of many background cells dominating the loss.

YOLO

Full Loss

YOLO treats **object detection as a regression problem**, and the total loss is made up of **three components**:

$$\mathcal{L}_{\text{total}} = \lambda_{\text{coord}} \cdot \mathcal{L}_{\text{bbox}} + \mathcal{L}_{\text{conf}} + \mathcal{L}_{\text{class}}$$

-
- λ_coord (typically 5)

- see example: link