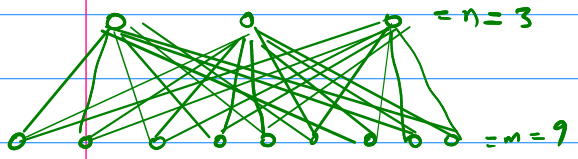
Imp / basics week 9.

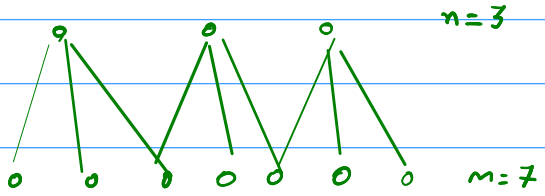
- Type of info which can be extracted out from an image
  - Semantic
  - metric
- Semantic tasks – object recognition, localization, Segmentation, tracking
  metric tasks – depth estimation, construction
- Challenges – viewpoint variation, illumination, Scale, deformation, Occlusion, background clutter and motion
- focus in course - image preprocessing, object recognition, object detection, estimation of depth from single image.

CNN → multilevel NN with 2 constraints. local connectivity & parameter sharing.



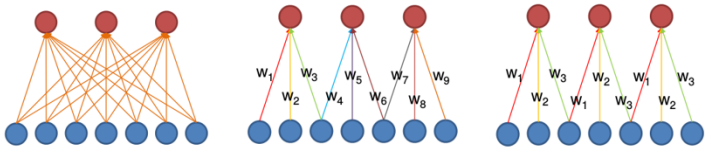
No. of parameter = mn = 27

k = 3



no. of parameters = kn = 3 × 3 = 9.

CNN: Parameter sharing (PS)



MLNN (21 parameters)

MLNN-LC (3 X 3 = 9 parameters)
2.3X runtime and storage efficient.

MLNN-LC-PS (3 parameters)
2.3X faster,
& 7X storage efficient.

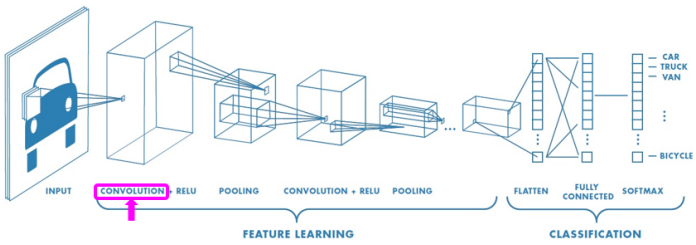
In general for a level with *m* input and *n* output nodes and CNN-local connectivity of *k* nodes *(k<m)*:

MLNN have
1. *m x n* parameters to store.
2. O(*m x n*) runtime

MLNN-LC have:
1. *k x n* parameters to store.
2. O(*k x n*) runtime

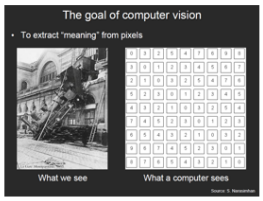
MLNN-LC-PS have:
1. *k* parameters to store.
2. O(*k x n*) runtime

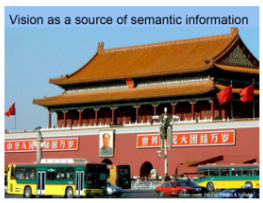
Different layers of CNN architecture

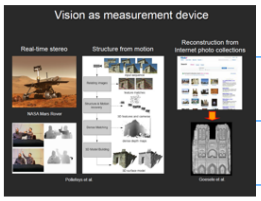


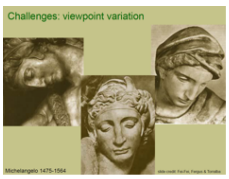
$$O = \frac{I + 2P - F}{S} + 1$$

Summary:









- Our focus in this course
  - Object Recognition
  - Object Detection
  - Depth from single image
  - Image pre-processing
- Advantage of CNN over MLP
  - Local connectivity
  - Parameter sharing
  - Efficient runtime and storage
- Different layers in CNN
  - Convolutional layer
  - ReLU
  - Pooling
  - Fully connected layer

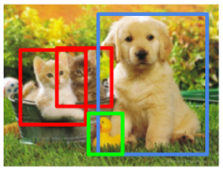
DLP: CV module
Object Detection

- Fast RCNN.
- Faster RCNN.
- YOLO (You only look once)

how can it be done.

Object Detection

- Another important task in computer vision



The task of assigning a **label** and a **bounding box** to all objects in the image

CAT, DOG, DUCK

- We can do this as image classification

Labels = [cat, dog, duck]

Cat? No

Dog? No

Duck? No

- We can do this as image classification

Labels = [cat, dog, duck]

Cat? Yes

Dog? No

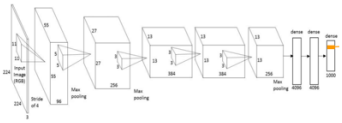
Duck? No

Problem: Too many positions & scales to test

Solution: If your classifier is fast enough, go for it

RCNN → Region based CNN.
- Fast RCNN.
- Faster RCNN.

- We can do this with a image classification network



Convnets are computationally demanding. We can't test all positions & scales !

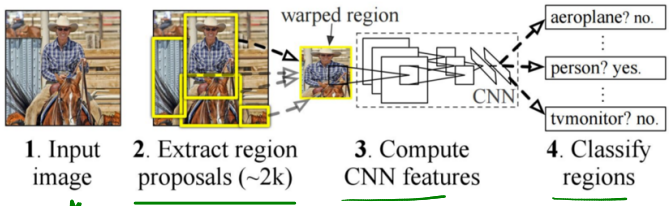
Solution: Look at a tiny subset of positions. Choose them wisely :)

Object Detection via Region Proposals

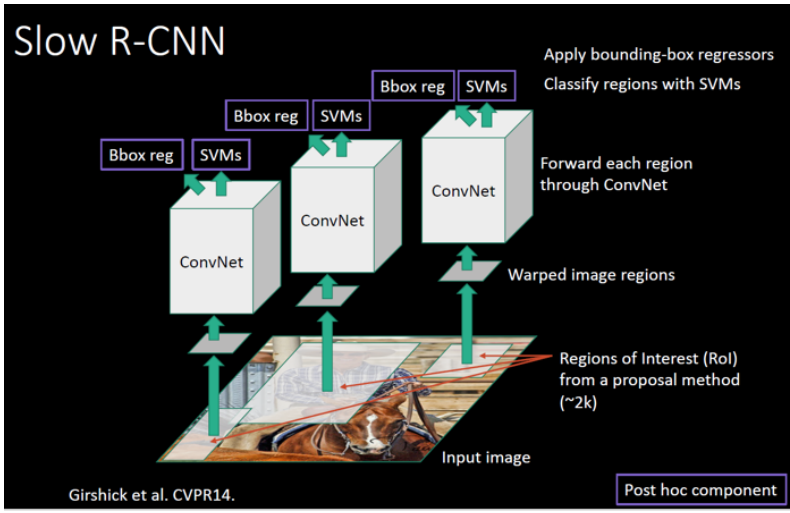
- Find "blobs" that are likely to contain objects
- "Class-agnostic" proposals

Object Detection with ConvNets: RCNN

- Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. CVPR 2014



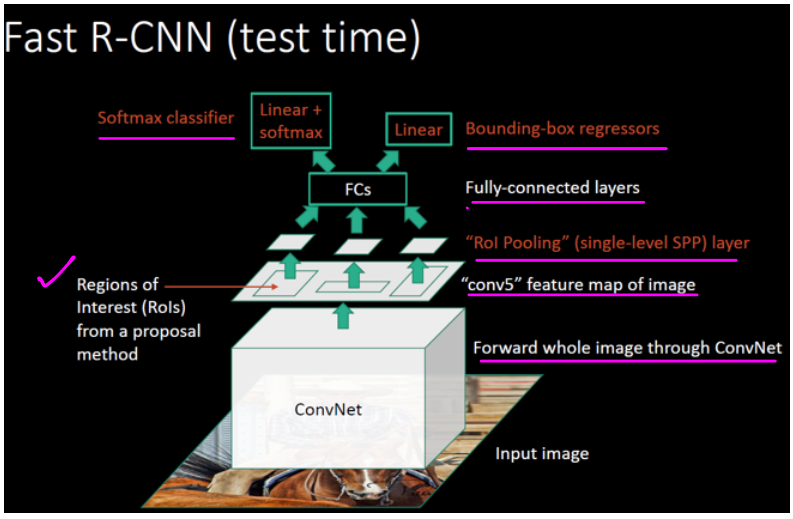
Not all

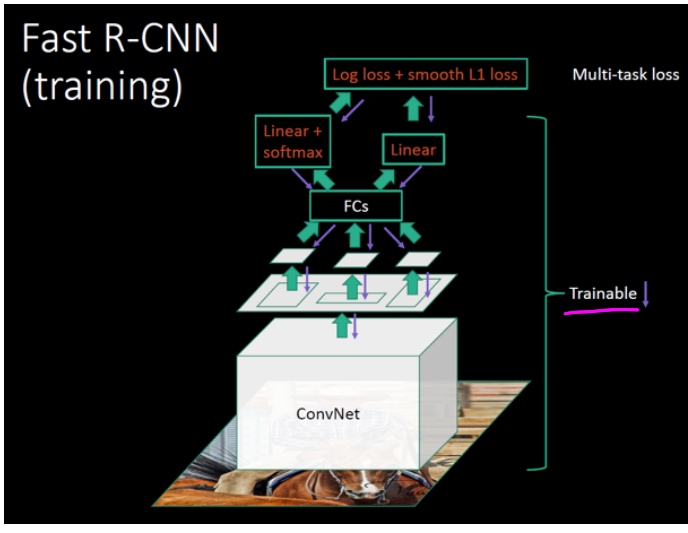


What's wrong with slow R-CNN?

- Ad hoc training objectives
  - Fine-tune network with softmax classifier (log loss)
  - Train post-hoc linear SVMs (hinge loss)
  - Train post-hoc bounding-box regressions (least squares)
- Training is slow (84h), takes a lot of disk space
- Inference (detection) is slow
  - 47s / image with VGG16 [Simonyan & Zisserman. ICLR15]
  - Fixed by SPP-net [He et al. ECCV14]

Fast R-CNN.





R-CNN (Regions with CNN features)

- **Step-by-step pipeline**:
  1. Uses **Selective Search** to generate ~2000 region proposals.
  2. Each region is **cropped and resized** to a fixed size.
  3. Each region is passed **individually** through a CNN (e.g., AlexNet).
  4. CNN features are used to train SVMs for classification and a regressor for bounding box refinement.
- **Drawback**: Each proposal is processed **separately** → very slow!

Fast R-CNN

- **Unified pipeline**:
  1. The entire image is passed **once** through a CNN to get a **feature map**.
  2. Region proposals (from selective search) are mapped onto this feature map.
  3. Uses **ROI Pooling** to extract fixed-size features from the shared feature map.
  4. A single network handles **classification + bounding box regression**.
- **Improvement**: Reuses feature maps → much **faster** than R-CNN.

⏱ 2. Speed

Aspect	R-CNN	Fast R-CNN
Inference	Very slow (e.g., 47s/image)	Much faster (e.g., 2s/image)
Training	Multi-stage & slow	Single-stage, end-to-end

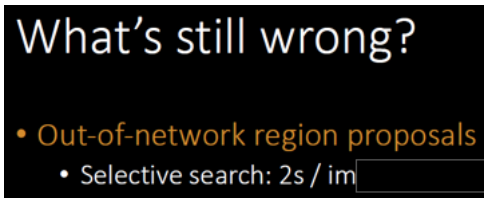
🎯 3. Training Strategy

- **R-CNN**:
  - 3 steps: CNN fine-tuning → SVM training → bounding box regression.
  - Complex and **not end-to-end**.
- **Fast R-CNN**:
  - **Single model**, trained end-to-end using a **multi-task loss** (classification + regression).

🧩 4. Storage & Memory

- **R-CNN**: Stores feature vectors for each region on disk → **high storage overhead**.
- **Fast R-CNN**: No such storage needed.

Feature	R-CNN	Fast R-CNN
Feature Extraction	Per region	Shared over full image
Speed	Slow	Fast
Training Complexity	Multi-stage	End-to-end
ROI Pooling	❌ Not used	✅ Used
Classifier	SVM	Softmax layer
Bounding Box Regressor	Separate model	Integrated into the network

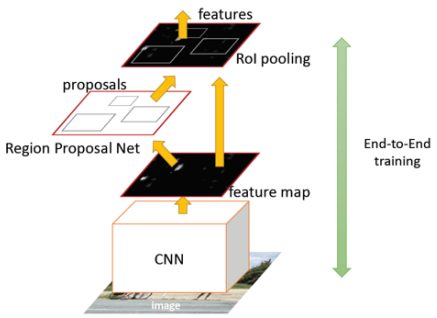


Faster – RCNN

Working.

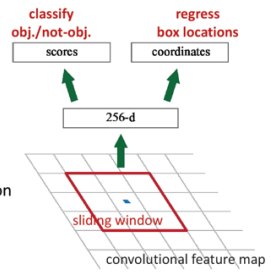
Object Detection: Faster R-CNN

- Faster R-CNN
  - Solely based on CNN
  - No external modules
  - Each step is end-to-end



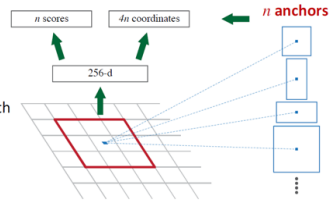
(RPN) Region Proposal Network

- Slide a small window on the feature map
- Build a small network for:
  - classifying object or not-object, and
  - regressing bbox locations
- Position of the sliding window provides localization information with reference to the image
- Box regression provides finer localization information with reference to this sliding window



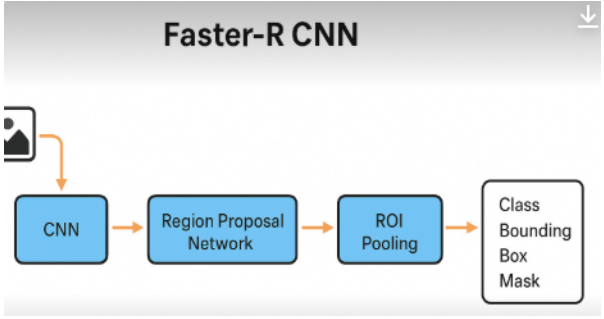
system	time	07 data	07+12 data
R-CNN	~50s	66.0	-
Fast R-CNN	~2s	66.9	70.0
Faster R-CNN	198ms ✓	**69.9**	**73.2**

- ✓ RPN is fully convolutional [Long et al. 2015]
- ✓ RPN is trained end-to-end
- ✓ RPN shares convolutional feature maps with the detection network (covered in Ross's section)



Feature	Fast R-CNN	Faster R-CNN
Region Proposal	Selective Search	Region Proposal Network
Proposal Time	~2 seconds	~10 milliseconds
Proposal Generation Method	Non-learnable	Learnable
Feature Sharing	Partial	Full
Anchors	❌ Not used	✅ Used
End-to-End Training	Partial	✅ Yes

There is mask-RCNN also, which adds instance segmentation on top of faster-RCNN.



Hyper Parameters in Faster. RCNN. *

RPN → Anchor Scales (Size of anchor box) 128 / 256 / 512

Anchor aspect ratio 1:1, 1:2, 2:1

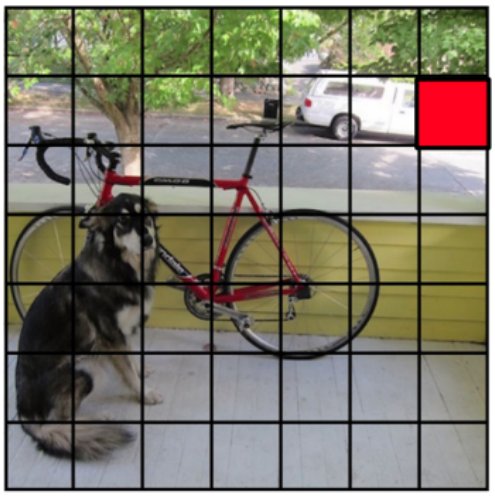
YOLO

Advantage of YOLO over R-CNN versions

Previous Approaches	YOLO
1. A separate model for generating bounding boxes and for classification (more complicated model pipeline)	1. A single neural network for localization and for classification (less complicated pipeline)
1. Need to run classification many times (expensive computation)	1. Need to inference only once (efficient computation)
1. Looks at limited part of the image (lacks contextual information for detection)	1. Looks at the entire image each time leading to less false positives (has contextual information for detection)

Overview of You Only Look Once (YOLO)

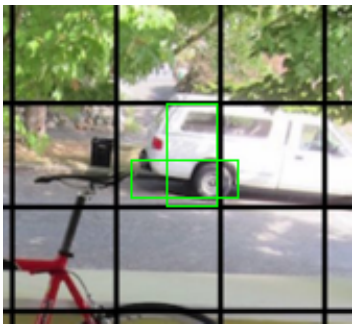
1. First, image is split into a SxS cells.
2. For each cell, generate B bounding boxes
3. For each bounding box, there are 5 predictions: x, y, w, h, confidence
4. For each cell, we need to also do object classification.



The image is divided into a 7 x 7 grid. For example, the red box is a cell in the grid.

For each cell, generate B bounding boxes. For example B=2.
For each bounding box, YOLO outputs x,y,w,h, and confidence of a presence of an object P(object).

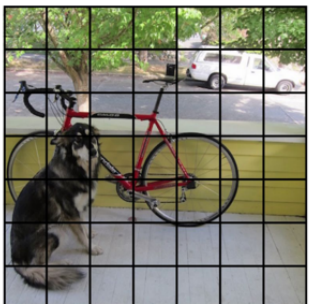
Ex.

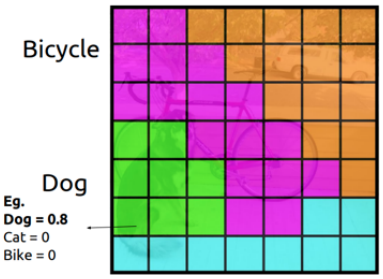


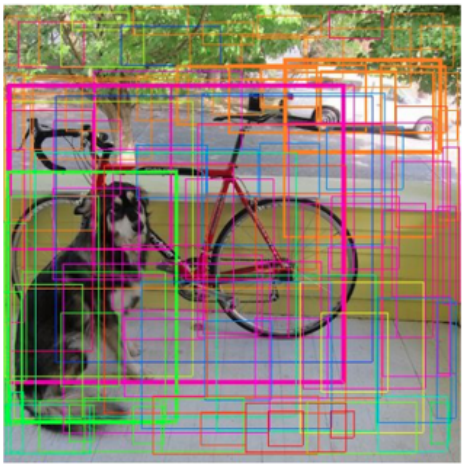
So for all cells (7x7=49), we have 98 boxes

and each box → x, y, w, h, confidence.

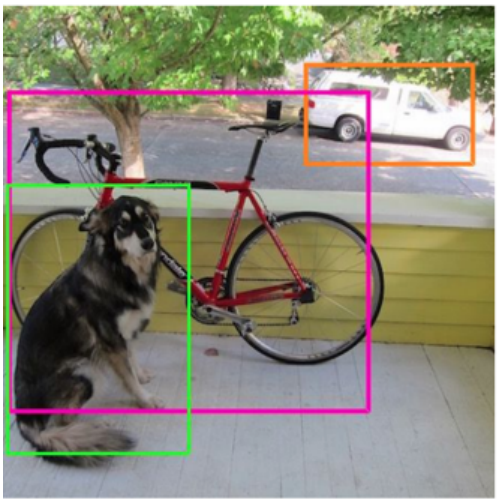
Each cell also predicts a class probability conditioned on object being present.
For example P(Car | Object)







P(class|Object) * P(Object) =P(class)



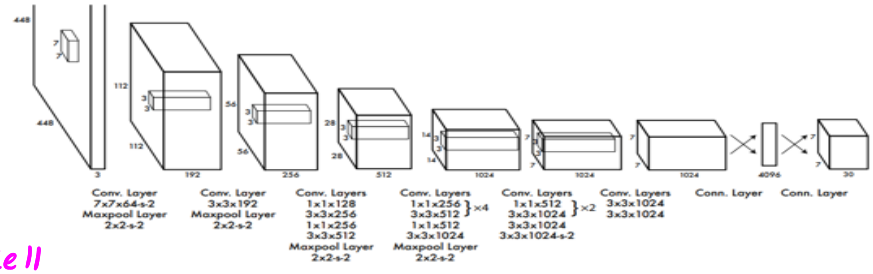
Further Thresholding operation and NMS

Yolo Architecture

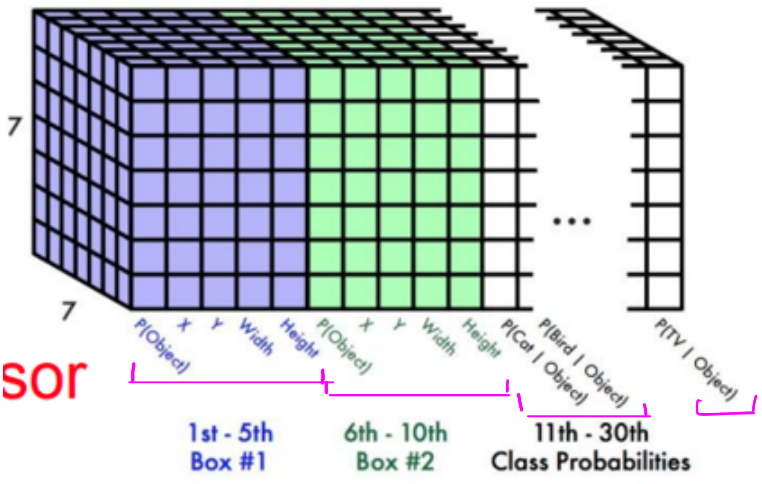
- Input image size is 448x448x3
- 24 convolution layers
- Default parameters: S = 7, B = 2, C = 20
- Output is S*S*(5B+C) = 7*7*(5*2+20) = 7*7*30

Each cell predicts:

- For each bounding box:
  - 4 coordinates (x, y, w, h)
  - 1 confidence value
- Some number of class probabilities



One cell



Loss Function in YOLO Model

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(x_i - \hat{x}_i\right)^2 + \left(y_i - \hat{y}_i\right)^2$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2$$

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

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

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

🎯 Main Hyperparameters in YOLO Training

1. Model Architecture Parameters

Hyperparameter	Description
`img-size`	Input image resolution (e.g., 640, 416, 320) — affects accuracy and speed
`depth_multiple`	Scales model depth (YOLOv5/yolov8s vs m vs l vs x)
`width_multiple`	Scales model width (number of channels per layer)
`nc`	Number of classes in your dataset
`anchors`	Predefined anchor boxes for each detection layer (in older YOLOs)

2. Optimization Hyperparameters

Hyperparameter	Description
`epochs`	Number of training epochs
`batch-size`	Number of images per training batch
`learning_rate` or `lr0`	Initial learning rate (e.g., 0.01 or 0.001)
`momentum`	Momentum in SGD optimizer (e.g., 0.937)
`weight_decay`	L2 regularization (e.g., 0.0005)
`optimizer`	Common choices: SGD, Adam, AdamW
`lr_scheduler`	Step, cosine, or one-cycle learning rate scheduling

3. Augmentation & Regularization

Hyperparameter	Description
`hsv_h`, `hsv_s`, `hsv_v`	Color jittering in hue/saturation/value
`flipud`, `fliplr`	Vertical & horizontal flips
`degrees`, `scale`, `translate`	Geometric transformations
`mosaic`	Combines 4 images into one (YOLOv4 and later)
`mixup`	Blends two images together (sometimes used)
`label_smoothing`	Smooths labels to prevent overconfidence

how to use YOLO

```
# Clone YOLOv5 repository
git clone https://github.com/ultralytics/yolov5.git
cd yolov5

# Install required dependencies
pip install -r requirements.txt
```

📁 2. Prepare Dataset (YOLO Format)

📁 Directory structure:

```
kotlin

datasets/
├── images/
|   ├── train/
|   └── val/
├── labels/
|   ├── train/
|   └── val/
├── data.yaml
```

📄 Sample `data.yaml`:

```
yaml

train: datasets/images/train
val: datasets/images/val

nc: 2  # number of classes
names: ['cat', 'dog']
```

🧠 3. Train YOLOv5

```
bash

python train.py \
  --img 640 \
  --batch 16 \
  --epochs 50 \
  --data data.yaml \
  --weights yolov5s.pt \
  --name yolov5-custom
```

- `--img`: input image size
- `--weights`: use pretrained weights (e.g., `yolov5s.pt`, `yolov5m.pt`, etc.)
- `--name`: experiment folder name in `runs/train/`

🧪 4. Run Inference (Test on New Images)

```bash
python detect.py \
  --weights runs/train/yolov5-custom/weights/best.pt \
  --img 640 \
  --conf 0.25 \
  --source path/to/test/images
```

Results will be saved in `runs/detect/exp/`.

📊 5. Evaluate on Validation Set

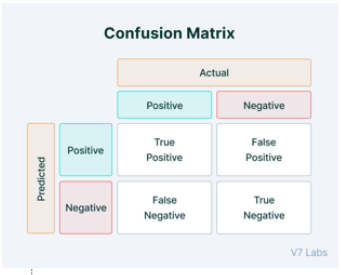
```bash
python val.py \
  --weights runs/train/yolov5-custom/weights/best.pt \
  --data data.yaml \
  --img 640
```

💻 6. Use Trained Model in Python

```python
import torch

# Load model
model = torch.hub.load('ultralytics/yolov5', 'custom', path='runs/train/yolov5-custom/weights/best

# Inference
results = model('path/to/image.jpg')
results.print()
results.show()  # display
results.save()  # save to file
```

Object detection metric: Mean Average Precision (mAP)

mAP is measure of how well the detector is able to localize and classify the objects in a image. It is based on the concepts of **Confusion Matrix**, **Intersection Over Union (IoU)**, **Precision** and **Recall**



$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

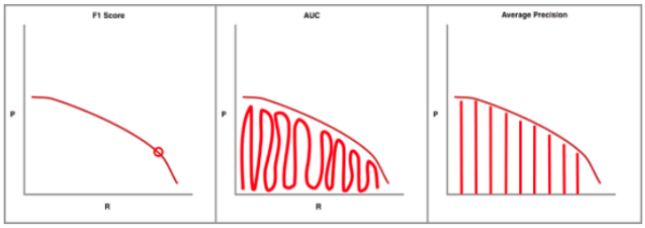
Precision measures how well you can find true positives(TP) out of all positive predictions (TP+FP).

Recall measures how well you can find true positives(TP) out of all predictions(TP+FN).

- If IoU > a certain threshold (e.g., 0.5), the prediction is considered a **True Positive (TP)**.
- If IoU ≤ threshold, the prediction is considered a **False Positive (FP)**.

Steps to Calculate mAP for Object Detection

1. **For Each Class**:
   - **Compute IoU**: For each predicted box, compute its IoU with the ground truth box.
   - **Assign True Positives and False Positives**:
     - If IoU > threshold (e.g., 0.5), it's a True Positive (TP).
     - If IoU ≤ threshold or there is no ground truth box for the prediction, it's a False Positive (FP).
     - If there's a ground truth box with no corresponding prediction, it's a False Negative (FN).
   - **Compute Precision and Recall**
2. **Precision-Recall Score Computation**: For each class, plot a Precision-Recall (P-R) curve, which is created by varying the confidence threshold used for accepting predictions (lower threshold = more detections, but possibly more FPs).



3. **Average Precision (AP)**: The area under the Precision-Recall curve is calculated. It is essentially the average of precision values across a range of recall values, typically from 0 to 1. A common method for calculating AP is by integrating over evenly spaced recall points (e.g., 0, 0.1, 0.2, ..., 1). This is called **Average Precision (AP)** for that specific class.

4. **Mean Average Precision (mAP)**: Once the AP for each class is computed, mAP is simply the mean of the AP values across all object classes.

Mean Average Precision (mAP) is the most widely used metric for evaluating object detection models. It takes into account not just whether objects are correctly detected, but also the localization quality of the bounding boxes. By averaging precision over different recall levels and across all classes, mAP provides a robust measure of detection performance.

Precision, Recall, False positive, False Negative, F1.

Recall : Out of total '100' positives, how many have been correctly classified.

Precision : Out of total '100' positives (classified), how many are 'truely' positive.

Classified \ (Actual)	1	0
1	TP	FP
0	FN	TN

$$\text{Recall}_p = \frac{TP}{TP + FN}$$

$$\text{Precision}_p = \frac{TP}{TP + FP}$$

- **Precision (positive)** =

$$\frac{TP}{TP + FP}$$

→ Out of all predicted positives, how many were actually correct.

- **Recall (positive)** =

$$\frac{TP}{TP + FN}$$

→ Out of all actual positives, how many were predicted correctly.

$$F_1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Practical : how to implement Faster RCNN. (Steps).

```
# 1. Load a backbone (e.g., ResNet50 with FPN)
backbone = torchvision.models.detection.backbone_utils.resnet_fpn_backbone(
    'resnet50', pretrained=True
)
```

```
# 2. Define anchor sizes and aspect ratios
anchor_generator = AnchorGenerator(
    sizes=((32, 64, 128, 256, 512),),
    aspect_ratios=((0.5, 1.0, 2.0),)
)
```

```
# 3. ROI Pooler settings
roi_pooler = torchvision.ops.MultiScaleRoIAlign(
    featmap_names=['0'], output_size=7, sampling_ratio=2
)
```

```
# 4. Build the Faster R-CNN model
model = FasterRCNN(
    backbone,
    num_classes=2,  # e.g., 1 class + background
    rpn_anchor_generator=anchor_generator,
    box_roi_pool=roi_pooler
)
```

```
# Optional: Freeze backbone layers
for name, parameter in model.backbone.body.named_parameters():
    if "layer2" not in name and "layer3" not in name and "layer4" not in name:
        parameter.requires_grad = False
```

```
# 5. Define transforms
transform = T.Compose([
    T.ToTensor(),
    T.Resize((800, 800))
])
```

```python
# 6. Optimizer and training hyperparameters
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)
```

```python
# 7. Training loop (simplified)
# for images, targets in dataloader:
#     images = [transform(img) for img in images]
#     loss_dict = model(images, targets)
#     losses = sum(loss for loss in loss_dict.values())
#     optimizer.zero_grad()
#     losses.backward()
#     optimizer.step()
#     lr_scheduler.step()
```

YOLO vs Faster R-CNN

Feature	YOLO (e.g., v3, v4, v5)	Faster R-CNN
Architecture Type	Single-stage detector	Two-stage detector
Speed	Very fast (real-time capable)	Slower, not suitable for real-time
Accuracy (mAP)	Slightly lower (esp. for small objects)	Higher accuracy
Inference Time	15–45 ms per image (GPU)	100–200+ ms per image (GPU)
Use Case	Real-time detection (e.g., drones, surveillance)	High-accuracy tasks (e.g., medical imaging)
Region Proposals	No proposals, direct predictions	Uses RPN to generate region proposals
Anchor Boxes	Yes (learned or pre-defined)	Yes (used in RPN and detection heads)
Input Processing	Entire image processed once	Proposal-based, multiple regions
End-to-End Trainable	Yes	Yes, but more complex training
Model Complexity	Lighter & more optimized (especially in v5/v8)	Heavier due to multi-stage pipeline
Output per Grid Cell	Class probabilities + box coords	ROI classification + box regression

PA : Week 9

3) For an RGB image with dimensions 300x300 from a mobile phone, how many parameters would need to be handled by a traditional Multi-Layer Perceptron (MLP)?

$300 \times 300 \times 3 = 270000$ (Input size.)

4) What are the features of a CNN?

- [x] (a) Parameter sharing.
- [ ] (b) Full connectivity.
- [x] (c) Local connectivity.
- [x] (d) Weight sharing.

5) Consider a 4x4 input matrix:

$$\begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

3x3

$$\begin{matrix} 6 & 7 & 8 \\ 10 & 11 & 12 \\ 14 & 15 & 16 \end{matrix}$$

Max. with a 2x2 kernel

What will be the output matrix after applying an average pooling with a stride of 1?

$O = \frac{I + 2P - F}{S} + 1 \therefore O = 4 - 2 + 1 = 3$

6) Mark the gate(s) which are not solvable by a normal perceptron?

- [ ] (a) NOR.
- [ ] (b) NAND.
- [ ] (c) AND.
- [x] (d) XNOR.

7) *For a CNN network with 10 input nodes and 5 output nodes with a local connectivity of 3 nodes, what do you think should be the runtime and storage requirements without parameter sharing?*

- [x] (a) Runtime: 15 operations, Storage: 15 parameters.
- [ ] (b) Runtime: 50 operations, Storage: 15 parameters.
- [ ] (c) Runtime: 30 operations, Storage: 30 parameters.
- [ ] (d) Runtime: 10 operations, Storage: 5 parameters.

Runtime = kn

$= 3 \times 5 = 15$

Storage = kn = 15

8) For a CNN network with 10 input nodes and 5 output nodes with a local connectivity of 3 nodes, what do you think should be the runtime and storage requirements with parameter sharing ?

- [ ] (a) Runtime: 15 operations, Storage: 15 parameters.
- [ ] (b) Runtime: 5 operations, Storage: 3 parameters.
- [ ] (c) Runtime: 30 operations, Storage: 30 parameters.
- [x] (d) Runtime: 15 operations, Storage: 3 parameters.

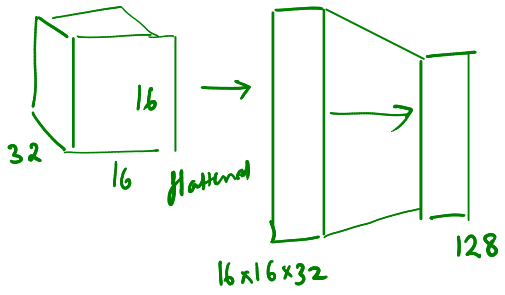
operations = kn = 15

Storage only 'k' parameters = 3

9) Why is the ReLU (Rectified Linear Unit) preferred over the Sigmoid activation function in CNNs?

- (a) ReLU is computationally cheaper compared to the Sigmoid function. ✓
- (b) ReLU helps mitigate the vanishing gradient problem, which is common with Sigmoid. ✓
- (c) Sigmoid outputs are bounded between 0 and 1, which can limit learning in deeper networks. ✓
- (d) ReLU introduces sparsity in activations, which can improve computational efficiency. ✓
- (e) All of the above. ✓

10) For a CNN layer, suppose the output after the convolutional and pooling operations is a feature map of shape (16,16, 32) (height = 16, width = 16, depth = 32). ***1 point***
If this feature map is flattened and connected to a fully connected to a fully connected layer with 128 output nodes, how many parameters (weights and biases) will this layer have?



∴ No. of Parameters will be

$16 \times 16 \times 32 \times 128 + 128$ (biases.)

$= 1048704$

Without biases 1048576

11) In a CNN, why is the Softmax activation function typically used in the output layer?

- (a) To convert the output feature maps into class probabilities.
- (b) To ensure that the outputs of the CNN represent probabilities that sum to 1 across all classes. ✓
- (c) To allow multi-class classification by emphasizing the highest probability class.
- (d) To improve computational efficiency in training the CNN. ✗

12) In a CNN, what is the primary distinction between the feature learning layers (convolutional and pooling layers) and the classification layers (fully connected layers)?

- (a) Feature learning layers reduce the size of the input, while classification layers increase it.
- ✓ (b) Feature learning layers extract spatial and hierarchical features, while classification layers map these features to specific output categories.
- (c) Feature learning layers use activation functions, while classification layers do not.
- (d) Feature learning layers perform probabilistic operations, while classification layers perform linear transformations. ✗

13) Consider a CNN layer with the following parameters:
- **Input size**: 32×32×3 (height × width × depth),
- **Filter size**: 5×5,
- **Number of filters**: 16,
- **Stride**: 2,
- **Padding**: 1.

What will be the output size of the feature map?

$O = \frac{I + 2P - F}{S} + 1 = \frac{32 + 2 - 5}{2} + 1 = \frac{29}{2} + 1 = 15$

∴ output size $= 15 \times 15 \times 16$

14) You are given an image of dimensions 227 × 227 × 3 representing its height, width, and color channels. You want to flatten it into a 1D array using Python. Which of the following code snippets will correctly achieve this transformation?

(A) import numpy as np
img = np.random.rand(227, 227, 3)
flat_img = img.flatten() ✓

(B) import numpy as np
img = np.random.rand(227, 227, 3)
flat_img = img.reshape(-1) ✓

(C) import numpy as np
img = np.random.rand(227, 227, 3)
flat_img = img.reshape((227*227*3,)) ✓

(D) import numpy as np
img = np.random.rand(227, 227, 3)
flat_img = img.flatten((227, 227, 3)) ✗

Which option(s) are correct?

A
- img.flatten() → flatten into 1D vector (array)
- img.reshape(-1) → flatten into 1D vector (array)
- img.reshape((227 × 227 × 3))

Both are similar

15) In a CNN, parameter sharing ensures that the same set of weights is applied across different regions of the input, enabling spatial feature detection. Consider a 7 × 7 × 3 input image passed through a convolutional layer with the following characteristics:

4 filters
Kernel size of 3 × 3
Stride 1
No padding
Which of the following PyTorch code snippets correctly demonstrates parameter sharing in this convolutional layer?

A.

```
import torch
import torch.nn as nn
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.conv = nn.Conv2d(in_channels=3, out_channels=4
                kernel_size=(3, 3), stride=1, padding=0)
  def forward(self, x):
    return self.conv(x)

model = CNN()
x = torch.randn(1, 3, 7, 7)  # Batch size 1, 3 channels, 7x7 ir
output = model(x)
print(output.shape)
```

B.

```
import torch
import torch.nn as nn
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.conv = nn.Conv2d(in_channels=3, out_channels=4
                kernel_size=(3, 3), stride=1, padding=1)
  def forward(self, x):
    return self.conv(x)

model = CNN()
x = torch.randn(1, 3, 7, 7)  # Batch size 1, 3 channels, 7x7 ir
output = model(x)
print(output.shape)
```

C.

```
import torch
import torch.nn as nn
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.conv = nn.Conv2d(in_channels=3, out_channe
                kernel_size=(3, 3), stride=1, padding
  def forward(self, x):
    # Manually perform convolution
    weight = self.conv.weight
    bias = self.conv.bias
    return nn.functional.conv2d(x, weight, bias=bias,
                stride=1, padding=0)
```

D.

```
import torch
import torch.nn as nn
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.conv = nn.Conv3d(in_channels=3, out_channels=4,
                kernel_size=(3, 3, 3), stride=1, padding=0)
  def forward(self, x):
    return self.conv(x)

model = CNN()
x = torch.randn(1, 3, 7, 7)  # Batch size 1, 3 channels, 7x7 image
output = model(x)
print(output.shape)
```

16) For the given below code snippet:

```
import torch
import torch.nn as nn
input_tensor = torch.tensor([[[[1, 2, 3, 4, 5, 6],
                [7, 8, 9, 10, 11, 12],
                [13, 14, 15, 16, 17, 18],
                [19, 20, 21, 22, 23, 24],
                [25, 26, 27, 28, 29, 30],
                [31, 32, 33, 34, 35, 36]]]], dtype=torch.float32)
conv_layer = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3, 3), stride=(1, 1), padding=0)
output_tensor = conv_layer(input_tensor)
print("Output shape:", output_tensor.shape)
```

17) If the padding is changed to 'padding='same'', what happens next?

- (a) The input image size remains 6 ×6, and the output size also stays 6 ×6.
- (b) The output size reduces to 4 ×4 because of the 3×3 filter and the stride of 1, but with extra padding.
- (c) The output size increases to 8 × 8 due to the padding.
- (d) The convolution operation cannot be performed because the padding is invalid.

18) The following code applies a custom pooling operation with a pool size of 2×2 and a stride of 2×2 using PyTorch:

```
import torch
import torch.nn.functional as F
input_tensor = torch.tensor([[[[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12],
                [13, 14, 15, 16]]]], dtype=torch.float32)
def min_pool2d(input_tensor, kernel_size, stride):
  return -F.max_pool2d(-input_tensor, kernel_size=kernel_size, stride=stride)
output_tensor = min_pool2d(input_tensor, kernel_size=2, stride=2)
print("Output after pooling:")
print(output_tensor)
```

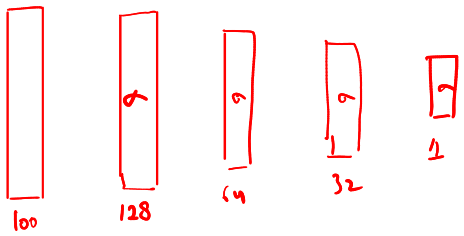
```
import torch

import torch.nn as nn

input_tensor = torch.tensor([[[[1, 2, 3, 4, 5, 6, 7, 8],
                               [9, 10, 11, 12, 13, 14, 15, 16],
                               [17, 18, 19, 20, 21, 22, 23, 24],
                               [25, 26, 27, 28, 29, 30, 31, 32],
                               [33, 34, 35, 36, 37, 38, 39, 40],
                               [41, 42, 43, 44, 45, 46, 47, 48],
                               [49, 50, 51, 52, 53, 54, 55, 56],
                               [57, 58, 59, 60, 61, 62, 63, 64]]]], dtype=torch.float32)

conv_layer = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(3, 3), stride=(2, 2), padding=0)

output_tensor = conv_layer(input_tensor)

print("Output shape:", output_tensor.shape)
```

I = 8×8

$O = \frac{8-3}{2} + 1 = \frac{5}{2} + 1 = 2 + 1 = 3$

$\therefore O = 3 \times 3$

Q.20



Input 10 × 100

The model will experience slow convergence or stagnation due to the vanishing gradient problem caused by sigmoid activation.

What is wrong?

```
import torch
import torch.nn as nn
import torch.optim as optim

class DeepNN(nn.Module):
    def __init__(self):
        super(DeepNN, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(100, 128),
            nn.Sigmoid(),
            nn.Linear(128, 64),
            nn.Sigmoid(),
            nn.Linear(64, 32),
            nn.Sigmoid(),
            nn.Linear(32, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.layers(x)

model = DeepNN()
input_tensor = torch.randn(10, 100)  # Batch size of 10, 100 input features
labels = torch.ones(10, 1)  # Target values

criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

output = model(input_tensor)
loss = criterion(output, labels)
loss.backward()
print("Loss:", loss.item())
```

find

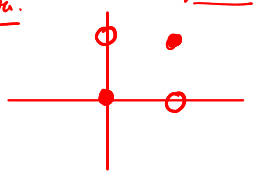
```
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32) # Inputs
y = torch.tensor([[1], [0], [0], [1]], dtype=torch.float32) # Outputs (XNOR)

class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.fc = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.fc(x))

model = SimpleMLP()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

for epoch in range(1000):
    optimizer.zero_grad()
    outputs = model(X)
    loss = criterion(outputs, y)
    loss.backward()
    optimizer.step()
```

model

X → [0, 0] → o/p

y = 1

data. XNOR



At every stage for (0,0)

Whatever no. we run, model will not be able to correctly identify (classify)

$\hat{y} = \sigma(0) = \frac{1}{1+e^{0}} = \frac{1}{2} = 0.5$ ✓

```
predictions = model(X).round()
print("Predicted outputs:")
print(predictions)
```

What do you think will be the output of the model?

1) Given an image dataset for object detection, how would you preprocess Fast R-CNN? Consider the following code snippet:

```
import cv2
import numpy as np
from skimage.feature import hog

image = cv2.imread('path_to_image.jpg')
resized_image = cv2.resize(image, (600, 600))
hog_features = hog(resized_image, orientations=9, pixels_per_cell=(8, 8),
    cells_per_block=(2, 2), visualize=False, multichannel=True)

def generate_proposals(image, window_size=(128, 128), step_size=32):
    proposals = []
    for y in range(0, image.shape[0] - window_size[1], step_size):
        for x in range(0, image.shape[1] - window_size[0], step_size):
            proposals.append((x, y, x + window_size[0], y + window_size[1]))
    return proposals

region_proposals = generate_proposals(resized_image)
```

Which of the following steps is NOT a part of Fast R-CNN preprocessing?

- (a) Resize the input image to a fixed size.
- (b) Extract features using HOG.
- (c) Generate region proposals.
- (d) Classify each region proposal directly.

part of fast R-CNN process.

HOG was used in old CV models.
↓
Histogram of Oriented Gradients.

Fast-RCNN uses deep CNN model for extraction of features.

2) Which of the following statements correctly differentiates Fast R-CNN from YOLO?

- (a) Fast R-CNN is a single-stage detector, while YOLO is a two-stage detector. X opposite.
- (b) Fast R-CNN relies on external region proposals, whereas YOLO divides the image into a grid for detection. Yes
- (c) YOLO uses Selective Search for region proposals, while Fast R-CNN performs end-to-end detection in one step. false opposite.
- (d) Both Fast R-CNN and YOLO perform detection and classification in a single stage. No.

```
import cv2
import numpy as np
from sklearn.svm import SVC
from skimage.feature import hog

image = cv2.imread('dog_cat.jpg')
ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()
    ss.setBaseImage(image)
    ss.switchToSelectiveSearchFast()
region_proposals = ss.process()

## Extract features using convnet
# Step 3: Train a classifier (SVM) on the extracted features
classifier = SVC(kernel='linear')
classifier.fit(features, labels) # labels are pre-defined for training
```

R-CNN. uses SVM

Which architecture follows this approach of using region proposals, feature extraction, and a classifier for object detection?

in Faster R-CNN:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# Input feature map from a backbone network (e.g., ResNet)
feature_map = torch.randn(1, 512, 50, 50) # (Batch, Channels, Height, Width)

# RPN layers
class RegionProposalNetwork(nn.Module):
    def __init__(self, in_channels, num_anchors):
        super(RegionProposalNetwork, self).__init__()
        self.conv = nn.Conv2d(in_channels, 512, kernel_size=3, stride=1, padding=1)
        self.cls_layer = nn.Conv2d(512, num_anchors * 2, kernel_size=1) # Classification
        self.reg_layer = nn.Conv2d(512, num_anchors * 4, kernel_size=1) # Regression

    def forward(self, x):
        x = F.relu(self.conv(x))
        cls_logits = self.cls_layer(x) # Objectness scores
        reg_deltas = self.reg_layer(x) # Bounding box adjustments
        return cls_logits, reg_deltas

# Initialize the RPN with 9 anchors per location
rpn = RegionProposalNetwork(in_channels=512, num_anchors=9)

# Generate RPN outputs
cls_logits, reg_deltas = rpn(feature_map)
print("Classification logits shape:", cls_logits.shape) # (1, 18, 50, 50)
print("Regression deltas shape:", reg_deltas.shape) # (1, 36, 50, 50)
```

Which of the following architectures incorporates this type of Region Proposal Network (RPN) for object detection?

RPNs are used in Faster-RCNN

```python
# Predicted bounding boxes and confidence scores
predictions = [
    {'bbox': [50, 50, 100, 100], 'score': 0.9},
    {'bbox': [30, 30, 70, 70], 'score': 0.75},
    {'bbox': [200, 200, 250, 250], 'score': 0.6}
]

# Ground truth bounding boxes
ground_truths = [
    {'bbox': [50, 50, 100, 100]},
    {'bbox': [30, 30, 70, 70]}
]

# Function to calculate IoU (Intersection over Union)
def calculate_iou(box1, box2):
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[2], box2[2])
    y2 = min(box1[3], box2[3])
    intersection = max(0, x2 - x1) * max(0, y2 - y1)
    area1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
    area2 = (box2[2] - box2[0]) * (box2[3] - box2[1])
    union = area1 + area2 - intersection
    return intersection / union if union > 0 else 0
```

```python
# Match predictions with ground truths (IoU > 0.5 considered correct)
threshold = 0.5
true_positives = 0
false_positives = 0
false_negatives = len(ground_truths)
for pred in predictions:
    matched = False
    for gt in ground_truths:
        iou = calculate_iou(pred['bbox'], gt['bbox'])
        if iou >= threshold:
            true_positives += 1
            false_negatives -= 1
            matched = True
            break
    if not matched:
        false_positives += 1

# Calculate precision and recall using the precision_recall function.
precision, recall = find_precission(true_positives, false_positives, true_negative, false_negative)
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

Pred 1 = [50, 50, 100, 100] matches with gt 1 [50, 50, 100, 100] ∴ TP

Pred 2 [30, 30, 70, 70] matches with gt 2 ∴ TP

Pred 3 does not match ∴ it is FP.

∴ TP = 2, FP = 1

TN = 0 FN = 0.

Now precision is out of total 'P' how may 'TP'

∴ Precision = $\frac{2}{3} = 0.67$

and recall is out of how many 'P', classified correctly TP

$= 2/2 = 1$

∴ Precision = 0.67, recall = 1

```python
import numpy as np

# Predicted boxes with confidence scores and ground truth boxes
predictions = [
    {'bbox': [50, 50, 100, 100], 'score': 0.9, 'class': 'cat'},
    {'bbox': [30, 30, 70, 70], 'score': 0.8, 'class': 'dog'},
    {'bbox': [200, 200, 250, 250], 'score': 0.7, 'class': 'cat'}
]

# Ground truth boxes
ground_truths = [
    {'bbox': [50, 50, 100, 100], 'class': 'cat'},
    {'bbox': [30, 30, 70, 70], 'class': 'dog'}
]

# Function to calculate IoU (Intersection over Union)
def calculate_iou(box1, box2):
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[2], box2[2])
    y2 = min(box1[3], box2[3])
    intersection = max(0, x2 - x1) * max(0, y2 - y1)
    area1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
    area2 = (box2[2] - box2[0]) * (box2[3] - box2[1])
    union = area1 + area2 - intersection
    return intersection / union if union > 0 else 0
```

```python
# Sort predictions by score (descending)
predictions = sorted(predictions, key=lambda x: x['score'], reverse=True)

# Evaluate precision and recall
tp, fp = 0, 0
matched_gt = set()
precision_recall_curve = []
for pred in predictions:
    matched = False
    for i, gt in enumerate(ground_truths):
        if gt['class'] == pred['class'] and i not in matched_gt:
            iou = calculate_iou(pred['bbox'], gt['bbox'])
            if iou >= 0.5: # IoU threshold
                tp += 1
                matched_gt.add(i)
                matched = True
                break
    if not matched:
        fp += 1

precision, recall = precision_recall(tp, fp, tn, fn)
# Calculate mAP as the average precision
mAP = np.mean([p for p, r in precision_recall_curve])
print(f"Mean Average Precision (mAP): {mAP:.2f}")
precision_recall_curve.append((precision, recall))
```

Given the predictions and ground truths, what is the Mean Average Precision (mAP)?

P_1 = 50, 50, 100, 100 0.9 cat → ∴ P = 1 R = 1/2 = 0.5

matches with gt 1

P_2 = 30, 30, 70, 70, 0.8 dog → gt 2 → P = 2/2 = 1 R = 2/2 = 1

P_3 does not match with anyone → P = 2/3 = 0.67 R = 2/2 = 1.

Precision-Recall Curve

Let's log the values as they're processed (pretend curve is built iteratively):

Step	TP	FP	Precision	Recall
1	1	0	1 / (1 + 0) = 1.0 1	1/2 1 / 2 = 0.5
2	2	0	2 / (2 + 0) = 1.0 1	1 2 / 2 = 1.0
3	2	1	2 / (2 + 1) ≈ 0.67 2/3	1 still 1.0

mAP = mean Average Precision

$= \frac{1 + 1 + 0.67}{3}$

$= \frac{2.67}{3} = 0.89$

```
Step 1
feature_map = # Input feature map (e.g., from a CNN backbone like ResNet)
# Region proposals as (x1, y1, x2, y2)
region_proposals = torch.tensor([
    [0, 0, 7, 7],
    [2, 2, 10, 10],
    [4, 4, 14, 14]
], dtype=torch.float32)

Step 2
# ROI Pooling
pooled_features = roi_pool(
    feature_map,
    [region_proposals],
    output_size=(7, 7) # Fixed size for each region
)
```

Region Proposal & RoI pooling is done in Fast R-CNN

```
Step 3
# Fully connected layer for classification and regression
class FastRCNNHead(nn.Module):
    def __init__(self, in_features, num_classes):
        super(FastRCNNHead, self).__init__()
        self.fc1 = nn.Linear(in_features, 1024)
        self.fc2 = nn.Linear(1024, num_classes + 4) # +4 for bb
    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Which architecture incorporates the above components for

- ( ) (a) R-CNN (Slow R-CNN)
- (✓) (b) Fast R-CNN
- ( ) (c) Faster R-CNN
- ( ) (d) YOLO

```
# Predicted labels and ground truth labels
predictions = [1, 0, 1, 1, 0, 1, 0]
ground_truth = [1, 0, 1, 0, 0, 1, 1]

# Confusion matrix components
true_positive = sum([1 for p, g in zip(predictions, ground_truth) if p == 1 and g == 1])
false_positive = sum([1 for p, g in zip(predictions, ground_truth) if p == 1 and g == 0])
false_negative = sum([1 for p, g in zip(predictions, ground_truth) if p == 0 and g == 1])

# F-score calculation find_f1_Score()
f_score = find_f1_Score(true_positive, false_positive, false_negative)
print(f"F-Score: {f_score:.2f}")
```

TP = 3, FN = 1
TN = 2, FP = 1

Predicted \ Actual	1	0
1	3 (TP)	1 (FP)
0	1 (FN)	2 (TN)

Precision → out of Predicted P, how many actual 'P'
$= \frac{TP}{TP+FP} = \frac{3}{4} = 0.75$

Recall → out of total 'P', how many correctly Predicted P
$= \frac{TP}{TP+FN} = \frac{3}{4} = 0.75$

F1 Score : $2 \cdot \frac{P \times R}{P+R} = \frac{2 \times 0.75 \times 0.75}{0.75 + 0.75}$
$= \frac{2 \times 0.75 \times 0.75}{2 \times 0.75} = 0.75$

Given the predictions and ground truth, what is the calculated F-score?

```
# Bounding box format: [x1, y1, x2, y2]
box1 = [50, 50, 150, 150]
box2 = [100, 100, 200, 200]

# Calculate the intersection coordinates
x1 = max(box1[0], box2[0])
y1 = max(box1[1], box2[1])
x2 = min(box1[2], box2[2])
y2 = min(box1[3], box2[3])

# Intersection area
intersection_area = max(0, x2 - x1) * max(0, y2 - y1)

# Areas of the two boxes
area_box1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
area_box2 = (box2[2] - box2[0]) * (box2[3] - box2[1])

# Union area
union_area = area_box1 + area_box2 - intersection_area

# IoU calculation
iou = intersection_area / union_area if union_area > 0 else 0
print(f"IoU: {iou:.2f}")
```

$x_1 = 100, y_1 = 100$
$x_2 = 150, y_2 = 150$

∴ Intersection area : $50 \times 50 = 2500$

area_box1 $= (150-50) \times (150-50) = 10000$
area box2 $= 100 \times 100 = 10000$

∴ Union area $= 10000 + 10000 - 2500 = 17500$

IoU $= \frac{2500}{17500} = \frac{1}{7} = 0.142$

Given the bounding boxes [50,50,150,150] and [100,100,200,200], what is the calculated IoU?

```
# Example predictions and ground truths
# Predictions: [x, y, w, h, confidence, class_scores...]
predictions = torch.tensor([[0.5, 0.5, 0.2, 0.2, 0.8, 0.7, 0.3]])
ground_truth = torch.tensor([[0.4, 0.4, 0.3, 0.3, 1.0, 1.0, 0.0]])

# Hyperparameters
lambda_coord = 5
lambda_noobj = 0.5
lambda_obj = 1.0
lambda_class = 1.0
lambda_iou = 2.0

# Loss functions
mse_loss = nn.MSELoss()

# Coordinate loss (x, y, w, h)
coord_loss = lambda_coord * mse_loss(predictions[:, :4], ground_truth[:, :4])

# Confidence loss (objectness vs no-object)
object_loss = lambda_obj * mse_loss(predictions[:, 4], ground_truth[:, 4])
no_object_loss = lambda_noobj * mse_loss(1- predictions[:, 4], 1- ground_truth[:, 4])

# IoU loss (for predicted and ground truth boxes)
def calculate_iou_loss(pred_bbox, gt_bbox):
x1 = torch.max(pred_bbox[:, 0]- pred_bbox[:, 2] / 2, gt_bbox[:, 0]- gt_bbox[:, 2] / 2)
y1 = torch.max(pred_bbox[:, 1]- pred_bbox[:, 3] / 2, gt_bbox[:, 1]- gt_bbox[:, 3] / 2)
x2 = torch.min(pred_bbox[:, 0] + pred_bbox[:, 2] / 2, gt_bbox[:, 0] + gt_bbox[:, 2] / 2)
y2 = torch.min(pred_bbox[:, 1] + pred_bbox[:, 3] / 2, gt_bbox[:, 1] + gt_bbox[:, 3] / 2)
intersection = torch.clamp(x2- x1, min=0) * torch.clamp(y2- y1, min=0)
union = (pred_bbox[:, 2] * pred_bbox[:, 3]) + (gt_bbox[:, 2] * gt_bbox[:, 3])- intersection
iou = intersection / union
return 1- iou.mean()

iou_loss = lambda_iou * calculate_iou_loss(predictions[:, :4], ground_truth[:, :4])

# Classification loss
class_loss = lambda_class * mse_loss(predictions[:, 5:], ground_truth[:, 5:])

# Total loss
total_loss = coord_loss + object_loss + no_object_loss + iou_loss + class_loss
print(f"Total Loss: {total_loss:.4f}")
```

Which of the following loss components are included in the YOLO loss function illustrated in the code snippet?

- (a) Coordinate loss, Object loss, No-object loss, Classification loss.
- (b) Coordinate loss, Object loss, IoU loss, Classification loss.
- ✓ (c) Coordinate loss, IoU loss, Object loss, No-object loss, Classification loss.
- (d) IoU loss, Object loss, Classification loss, Regularization loss.

YOLO loss →
- Coordinate loss
- iou loss
- Confidence loss — object loss, no object loss
- Classification loss

11) Which of the following networks can be used as feature extractors in Faster R-CNN?

- (a) VGG16, ResNet50, MobileNet ✓
- (b) YOLO, SSD, RetinaNet ✗
- (c) AlexNet, LeNet, GAN ✗
- (d) RPN, FPN, DenseNet ✗

12) Which of the following statements correctly differentiates Fast R-CNN from Slow R-CNN? ✓ *1 point*

- (a) Fast R-CNN uses external region proposals, while Slow R-CNN generates region proposals directly during training. ✗
- (b) Slow R-CNN uses ROI Pooling for fixed-size feature extraction, while Fast R-CNN processes entire images in a single pass. ✗
- (c) Fast R-CNN performs feature extraction on the entire image once, while Slow R-CNN extracts features sepa rately for each region proposal. ✓
- (d) Slow R-CNN integrates region proposal generation into the network, while Fast R-CNN relies on Selective Search for proposals. ✗

Key take aways

RCNN — Slow
Fast RCNN — uses proposals (region)
Faster CNN → has RPNs
YOLO → do in one. (single pipeline)

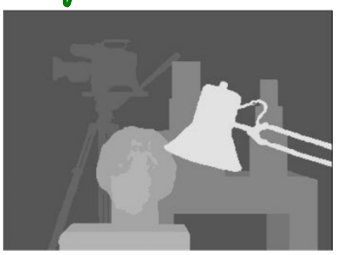
Precision → out of total predicted 'p', how many True P = $\frac{TP}{TP+FP}$

Recall → out of how many actual 'p', how many predicted 'p' = $\frac{TP}{TP+FN}$

Week 11: basics Depth estimation

Depth Estimation : meaning.

RGB Image of the scene



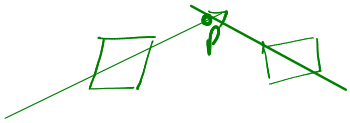
Depth map of the scene

- Estimate Pixel-level depth relative to camera position !

Applications

- Autonomous Navigation
  - Indoor (Humanoid)
  - Roads (Cars)
  - Aerial (Drones)
- Virtual/Augmented Reality
  - Virtual Meeting Rooms
  - Gaming
  - Robotic Surgeries
- Scene Understanding
  - Semantic Summarization
  - Large Scale 3D Mapping
  - Architecture/Heritage Capture

Depth from Stereo



Monocular depth estimation

- Challenging task
- Depth cues like parallax is missing
- Some cues that exist are vanishing points, object sizes etc
- These cues cannot be quantified through a rule based method

Monocular depth estimation methods

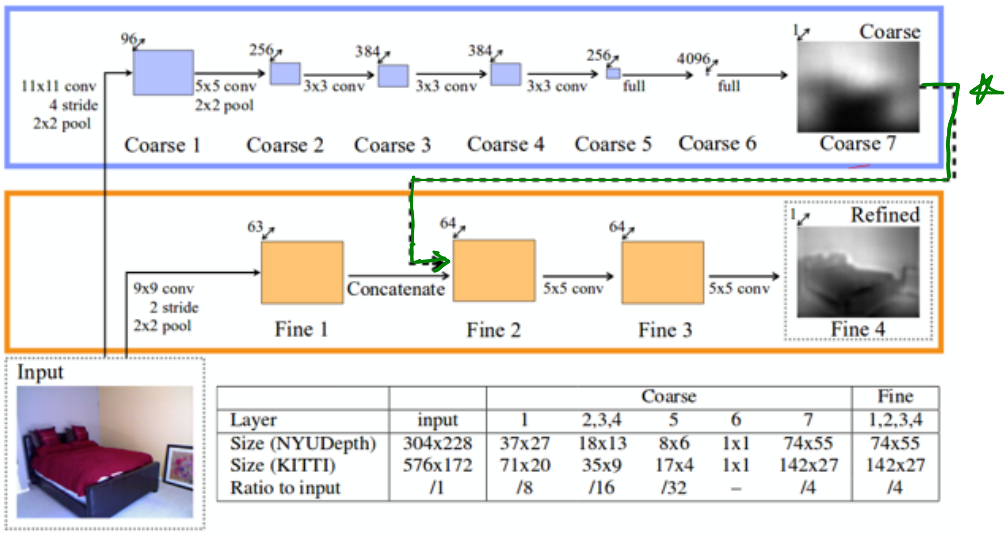
- ➢ Supervised techniques
  - Multi-Scale Deep Network
  - UNet
- ➢ Unsupervised
  - Depth from left-right consistency

Depth map from a single image using multi-scale deep network

Entire network is divided into 2 parts:

- Coarse network
- Fine Network

- Coarse network predicts the overall depth for the scene
- Fine network refines the global prediction locally



		Coarse					Fine
Layer	input	1	2,3,4	5	6	7	1,2,3,4
Size (NYUDepth)	304x228	37x27	18x13	8x6	1x1	74x55	74x55
Size (KITTI)	576x172	71x20	35x9	17x4	1x1	142x27	142x27
Ratio to input	/1	/8	/16	/32	–	/4	/4

Coarse network needs larger receptive field and is hence more deep compared to the fine network

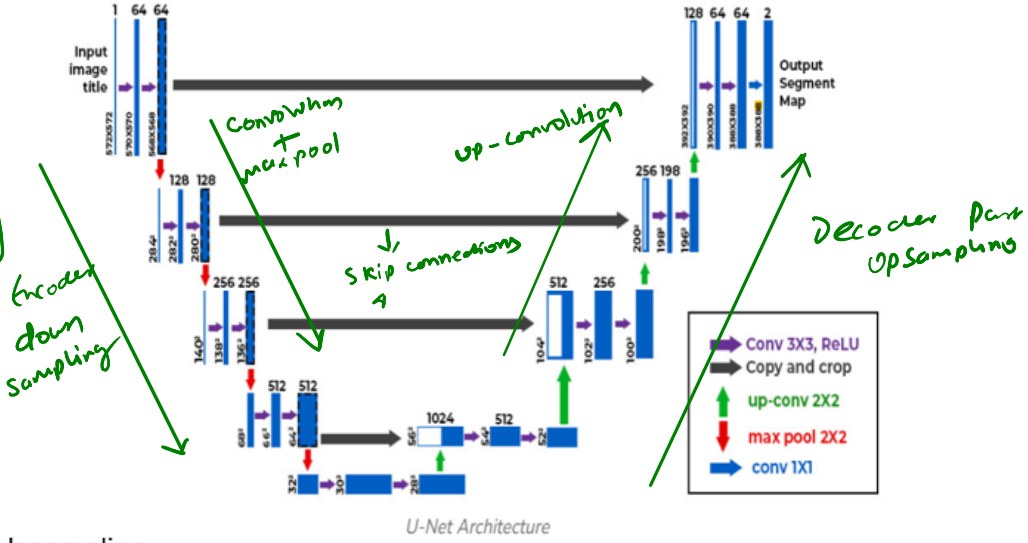
Results

Original Image | Coarse estimate | Fine estimate | Ground Truth

Depth map from a single image using a UNet based Architecture

Entire network is divided into 2 parts:

- Encoder Downsampling Network
- Decoder Upsampling Network



U-Net Architecture

Encoder Downsampling Network

- Uses convolutional layers followed by max pooling to progressively reduce spatial dimensions while increasing the number of feature channels. Captures high-level features and context in the image.

Decoder Upsampling Network

- Involves transposed convolutions (or upsampling layers) to increase the spatial dimensions of feature maps. Combines low-level features from the downsampling path with high-level features to produce detailed output depth maps.

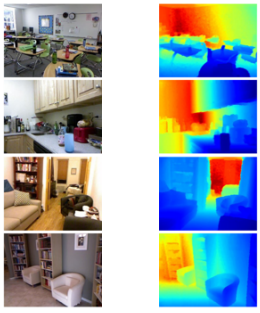
Skip Connections

- Directly connect corresponding layers in the downsampling path to the upsampling path. Preserve spatial information lost during downsampling, allowing for more precise reconstructions in the output.

Loss Function : MSE $\frac{1}{N}(\hat{y}-y)^2$, MAE = $\frac{1}{N}|\hat{y}-y|$

Unsupervised Monocular depth estimation

- Deep learning requires tons of training data for supervised learning
- For depth estimation you need the image depth map pairs



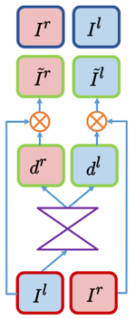
NYU Depth dataset

Unsupervised depth estimation with left-right consistency

- Can we use the relation between left - right image pairs and depth map of the scene to learn in an unsupervised setting?

~~consistency~~

- Use left image of the stereo pair predict the disparity
- True disparity is not available for training
- Now, transform the left image to right image using the predicted disparity
- True right image is available and can be used for supervision



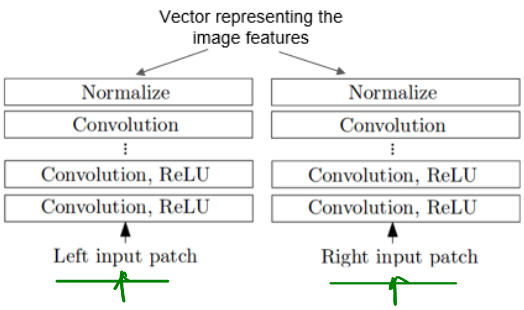
Ours

Godard et. al. 2016

- At test time network predicts disparity map from only a single image.

Matching Cost-CNN → used for 'Stereo' depth estimation

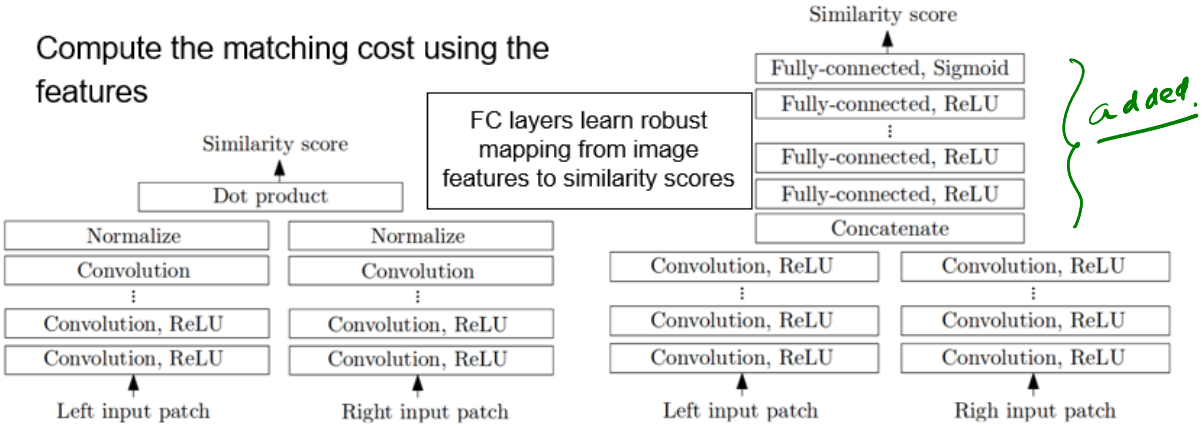
- Compute matching cost based on CNN features for an image



Parameters are shared between the two networks (Siamese network)

MC- CNN

Compute the matching cost using the features



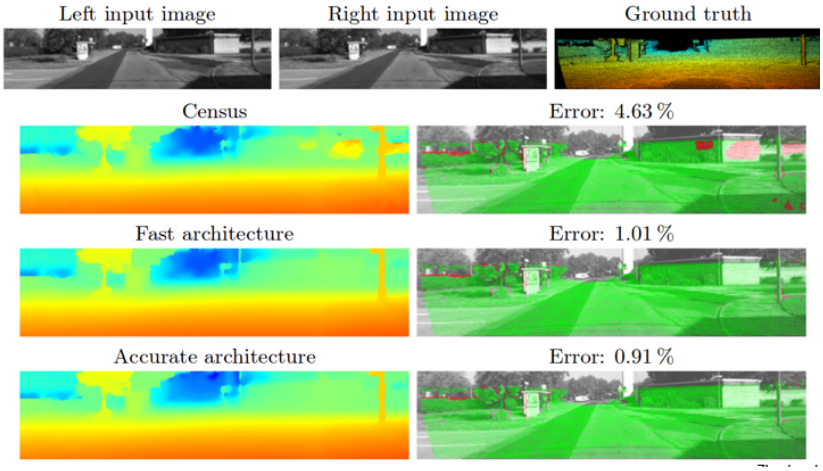
Fast Architecture

Accurate Architecture

FC layers require lot of parameters to be learned and are also slower in terms of computation time

Zbontar et. al. 201

Results for MC-CNN



☑ Best Architectures for Monocular Depth Estimation

1. **MiDaS** (by Intel)
   - 🧠 **State-of-the-art performance** across diverse scenes.
   - Uses a pre-trained encoder like **ResNet** or **ViT**.
   - Trained on multiple datasets to generalize well.
   - 📦 MiDaS GitHub
2. **Monodepth2** (by Godard et al.)
   - Uses self-supervised learning from stereo or monocular video.
   - Good if you're training from scratch and have access to video/stereo data.
   - 📦 Monodepth2 GitHub
3. **DenseDepth**
   - Based on U-Net with a DenseNet encoder.
   - Pretrained on NYU Depth v2 dataset.
   - Good for indoor scene depth prediction.

'Imp. points from PA : week 11 :1. In 'U Net' Encoder convolution + Maxpool
Decoder No Maxpool.

2. MSE 3. for unsupervised : loss

A.

```
import torch

import torch.nn.functional as F

def left_right_loss(depth_left, depth_right, disparity_left):

    depth_reconstructed = F.grid_sample(depth_right, disparity_left)
```

5) In depth estimation from stereo images, what is the primary role of the "disparity map"?

- (a) To directly compute the depth values without requiring any other information. ✗
- ✓ (b) To identify matching points between the left and right images and measure their displacement.
- (c) To smooth the textures in the images for better feature matching. ✗
- (d) To estimate the camera's intrinsic parameters for depth calculation. ✗

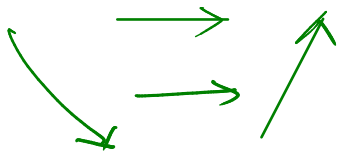
6) What are the primary challenges of estimating a depth map from a single image using a multiscale deep neural network?

- ✓ (a) Understanding global context and handling variations in scene geometry and textureless regions.
- ✗ (b) Accurately aligning stereo image pairs and computing disparity for depth estimation.
- ✗ (c) Reducing computational complexity while matching points across two images
- ✗ (d) Using depth sensors to directly capture depth information instead of estimating it.

7) What kind of pooling layer is used in Unet architecture?

- (a) Min pool of 2x2.
- ✓ (b) Max pool of 2x2.
- (c) Average pool of 2x2.
- (d) Average pool of 4x4.

Max pool in Encoder down sampling



Week 12

DLP: CV module
Image Pre-processing

Most Common image pre-processing tasks — Denoising, deblurring, Super resolution

Image denoising

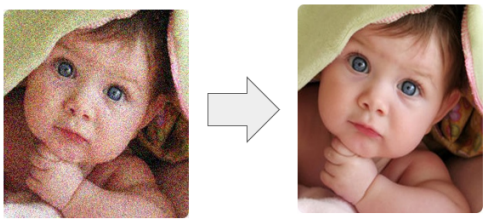


Image formation model

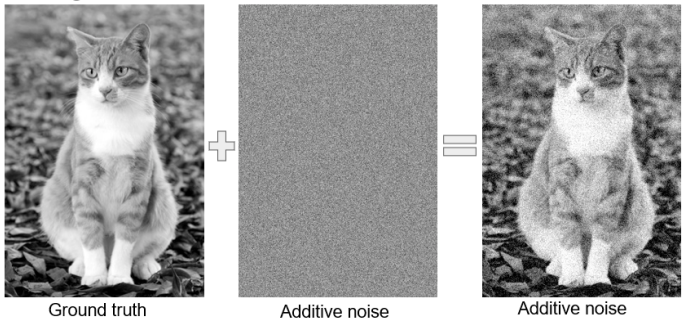
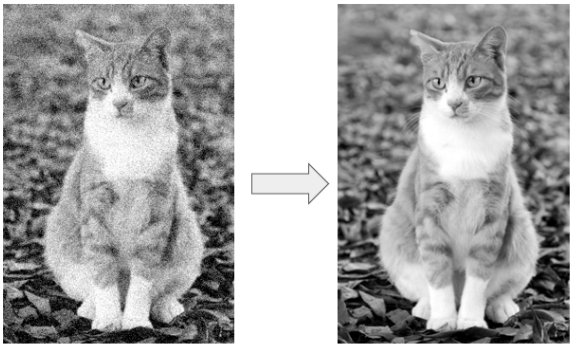


Image denoising

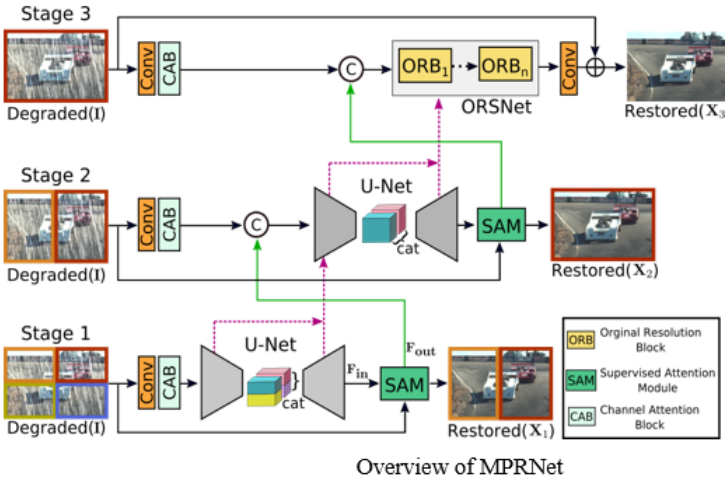


Noisy image | Clean image

MPRNet: Multi-Stage Progressive Image Restoration[1]

* MPR Net for denoising

- Learns the contextualized features using encoder-decoder architectures and later combines them with a high-resolution branch that retains local information.
- Multi-stage architecture progressively refines image quality across several stages.
- Parallel branches extract multi-scale features for capturing both local and global information.
- Each stage outputs intermediate results, passed to the next for further enhancement.



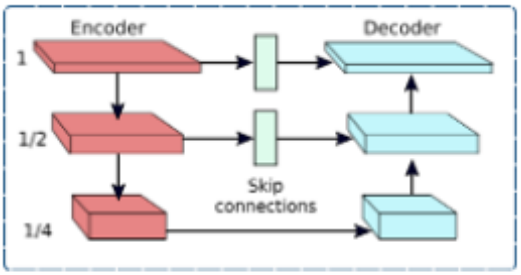
Overview of MPRNet

[1] Zamir, S.W., Arora, A., Khan, S., Hayat, M., Khan, F.S., Yang, M.H. and Shao, L., 2021. Multi-stage progressive image restoration. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 14821-14831).

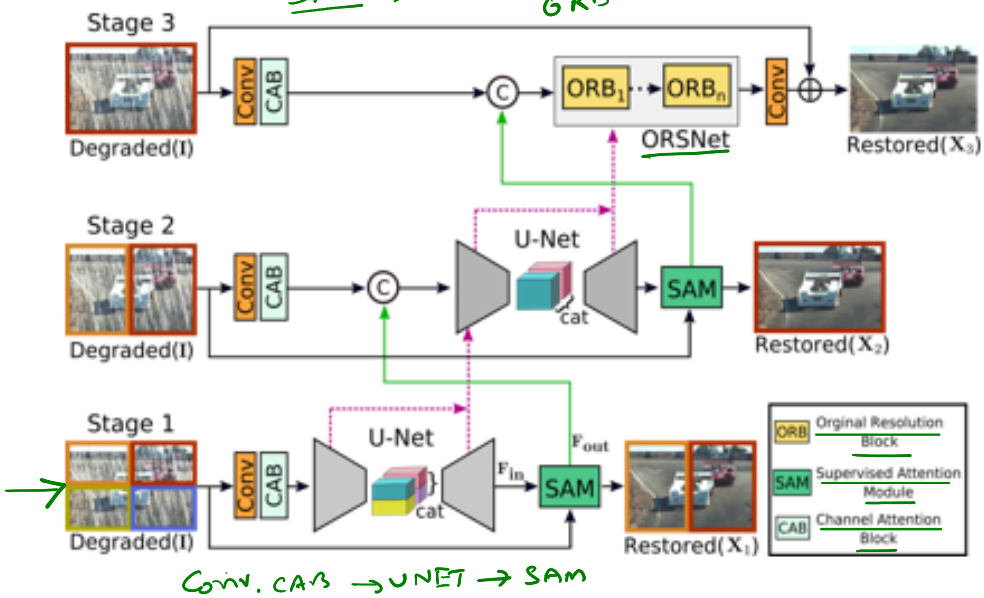
CAB → Channel Attention Block
SAM → Supervised Attention Module
ORB → Orginal resolution block.

Encoder-Decoder Architecture:

- The encoder-decoder subnetwork is based on the standard U-Net architecture.
- It incorporates Channel Attention Blocks (CABs) for enhanced feature extraction at multiple scales.
- Bilinear upsampling followed by convolution is used to increase spatial resolution.



Encoder-decoder subnetwork



Conv. CAB → UNET → SAM

Channel Attention Block (CAB)

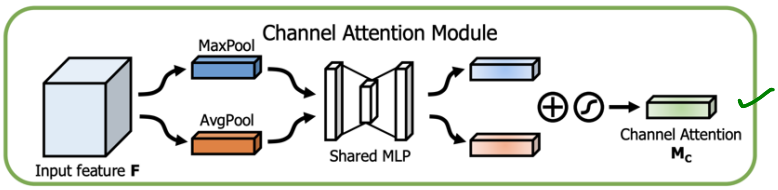


Figure courtesy CBAM: Convolutional Block Attention Module

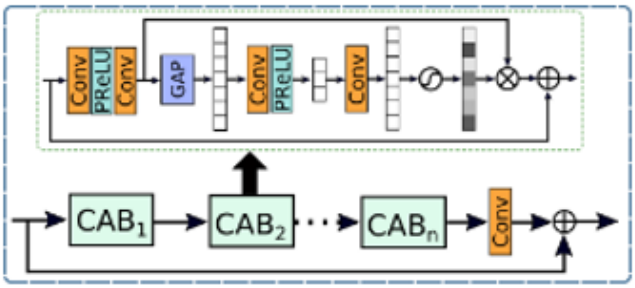
Channel attention map is for exploiting the inter-channel relationship of features.

Channel attention focuses on 'what' is meaningful given an input image. To compute the channel attention, the spatial dimension of the input feature map is squeezed using average-pooling and max-pooling, generating two different spatial context descriptors.

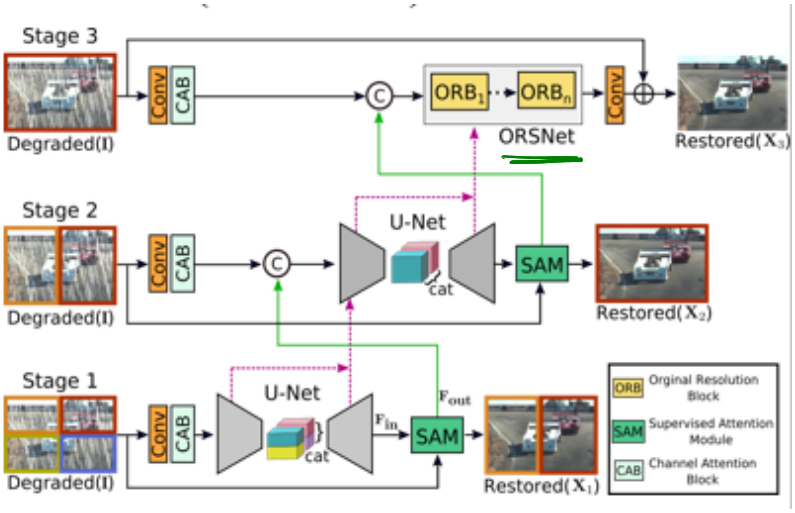
Both descriptors are then forwarded to a shared network to produce the channel attention map.

Original Resolution Subnetwork (ORSNet):

- Preserves fine details in image restoration.
- Operates without downsampling, producing spatially-enriched high-resolution features.
- Composed of multiple Original-Resolution Blocks (ORBs), each containing Channel Attention Blocks (CABs).

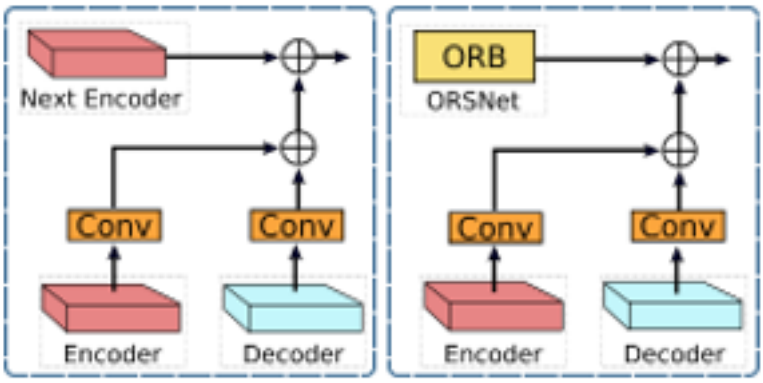


Original resolution block (ORB) in ORSNet subnetwork



Cross-stage Feature Fusion (CSFF) module:

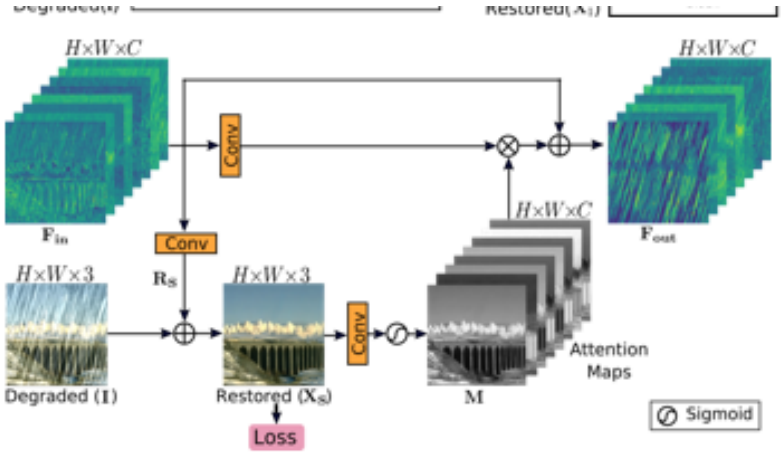
- CSFF is integrated between encoder-decoders and between the encoder-decoder and ORSNet.
- Features from one stage are refined using 1×1 convolutions before being passed to the next stage for aggregation.
- Reduces information loss, enhances feature enrichment across stages, and stabilizes the network optimization process, facilitating the addition of more stages.



Cross-stage Feature Fusion (CSFF) module

Supervised Attention Module (SAM):

- Introduced between stages to improve performance in multi-stage image restoration networks.
- SAM provides ground-truth supervisory signals for progressive image restoration at each stage.
- It generates attention maps to suppress less informative features, allowing only useful features to propagate to the next stage.



Supervised Attention Module (SAM)

MPRNet: Multi-Stage Progressive Image Restoration

At any given stage S, instead of directly predicting a restored image Xs , the model predicts a residual image Rs to which the degraded input image I is added to obtain: Xs = I + Rs .

$$\mathcal{L} = \sum_{S=1}^{3} [\mathcal{L}_{char}(\mathbf{X}_S, \mathbf{Y}) + \lambda \mathcal{L}_{edge}(\mathbf{X}_S, \mathbf{Y})],$$

$$\mathcal{L}_{char} = \sqrt{\|\mathbf{X}_S - \mathbf{Y}\|^2 + \varepsilon^2},$$

$$\mathcal{L}_{edge} = \sqrt{\|\Delta(\mathbf{X}_S) - \Delta(\mathbf{Y})\|^2 + \varepsilon^2},$$

Y = ground-truth image

L_{char} = Charbonnier loss

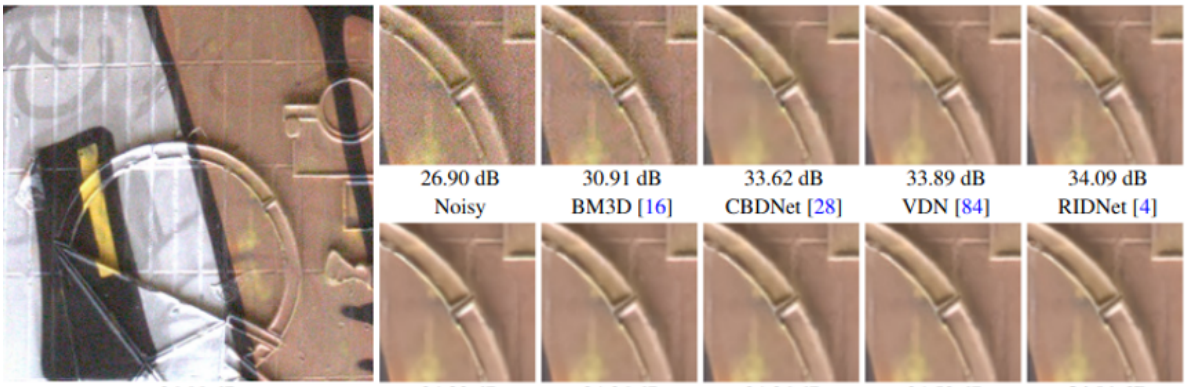
L_{edge} = edge loss

Δ denotes the Laplacian operator

ε = 10−3

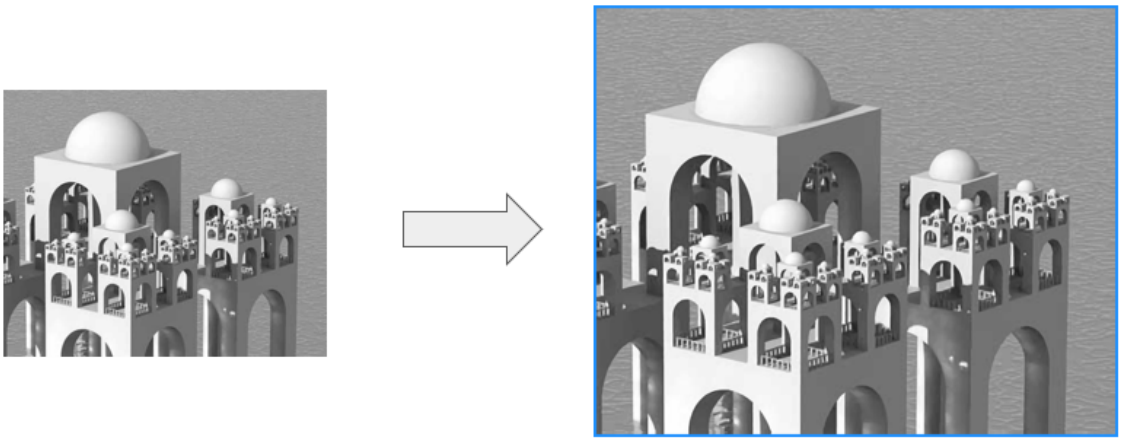
λ = 0.05

MPRNet results



Next task

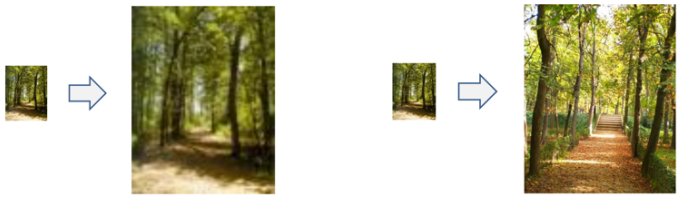
Image super-resolution



Single image super-resolution

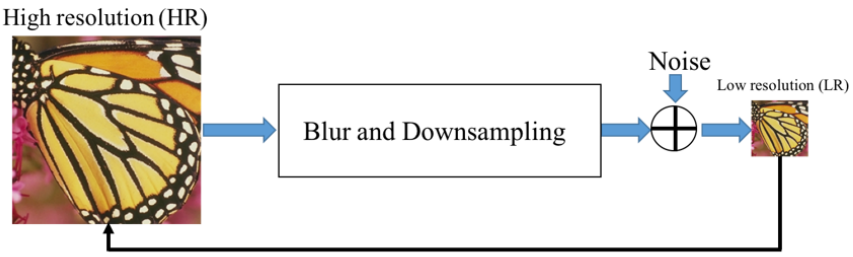
Main goals

- Be faithful to the low resolution input image
- Produce a detailed, realistic output image



SISR → Single Image Super resolution

Image super-resolution is ill-posed



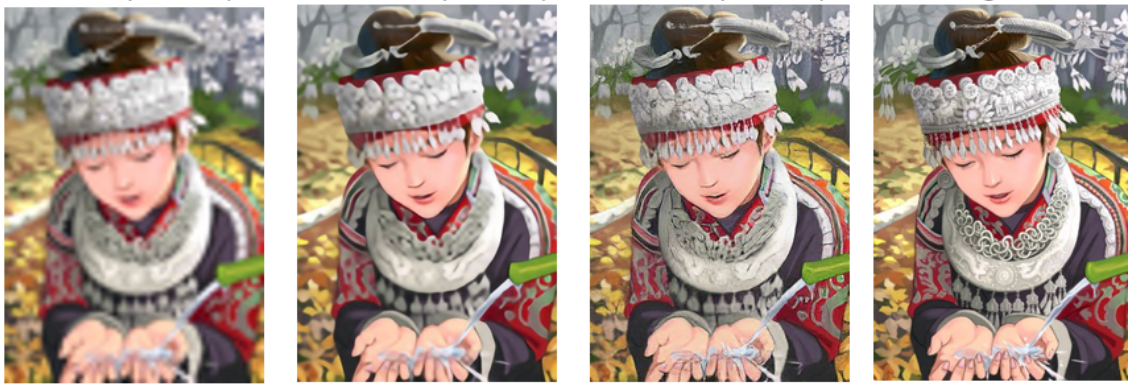
SISR: Try to recover HR from its LR counterpart

Many solutions are possible. DL comes to rescue

Challenges in SR

Figure courtesy SRGAN paper

bicubic (21.59dB) | SRResNet (23.53dB) | SRGAN (21.15dB) | original



PSNR - Peak signal to Noise ratio.

Challenge in Super-Resolution: Despite advancements in faster and deeper CNNs, recovering fine texture details during large upscaling is very difficult.

Problem with Current Methods: Most methods focus on minimizing mean squared error, leading to high PSNR but lacking high-frequency details and perceptual quality.

GAN based approach better than MSE based solution

esrgan used in npees

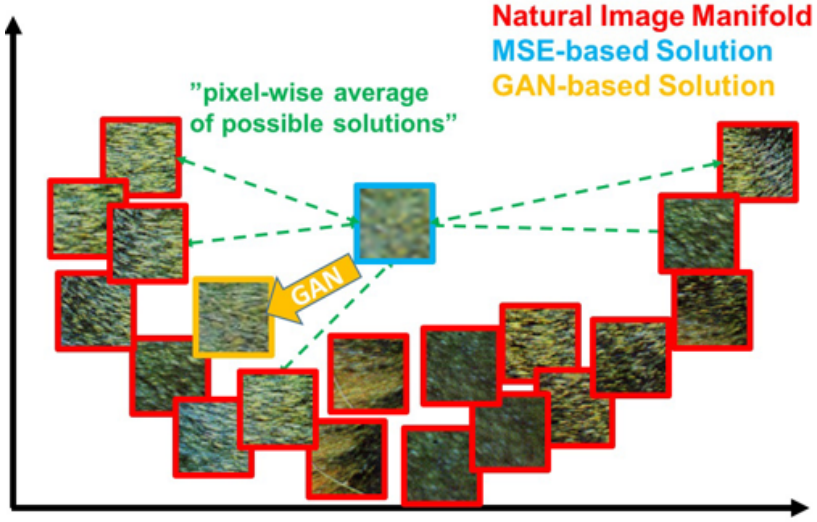


Illustration of patches from the natural image manifold (red) and super-resolved patches obtained with MSE (blue) and GAN (orange).

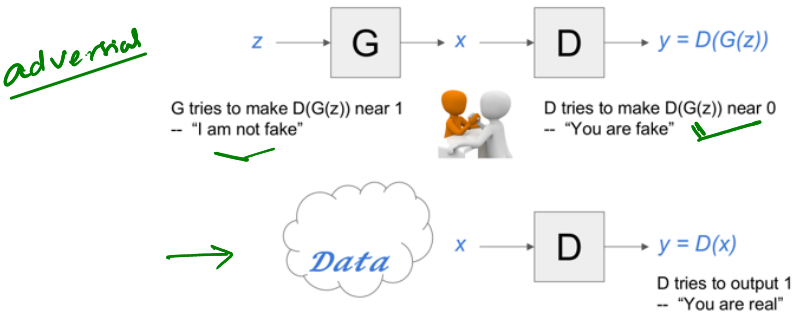
The MSE-based solution appears overly smooth due to the pixel-wise average of possible solutions in the pixel space.

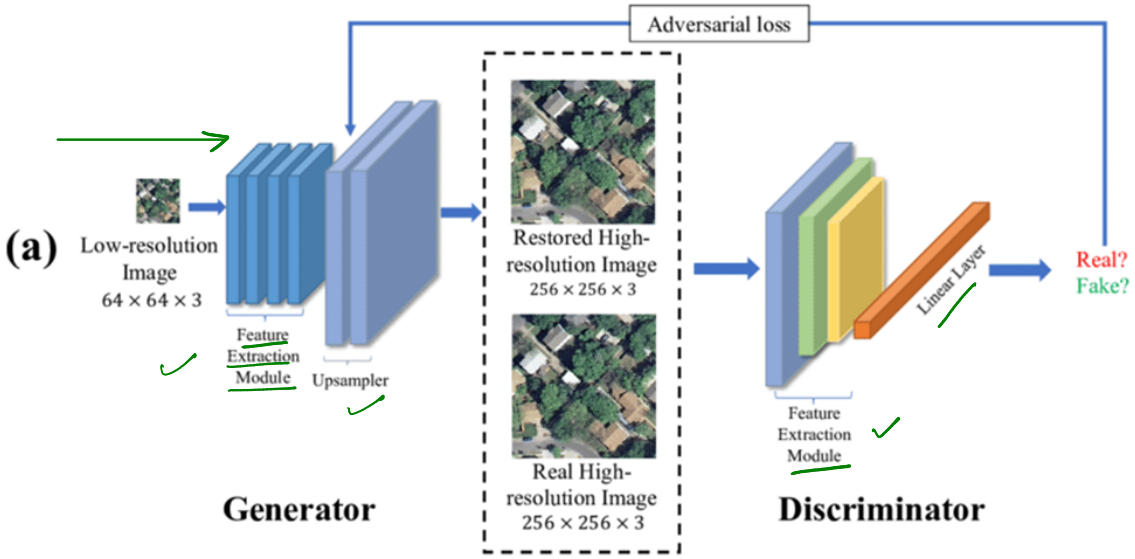
GAN drives the reconstruction towards the natural image manifold producing perceptually more convincing solutions.

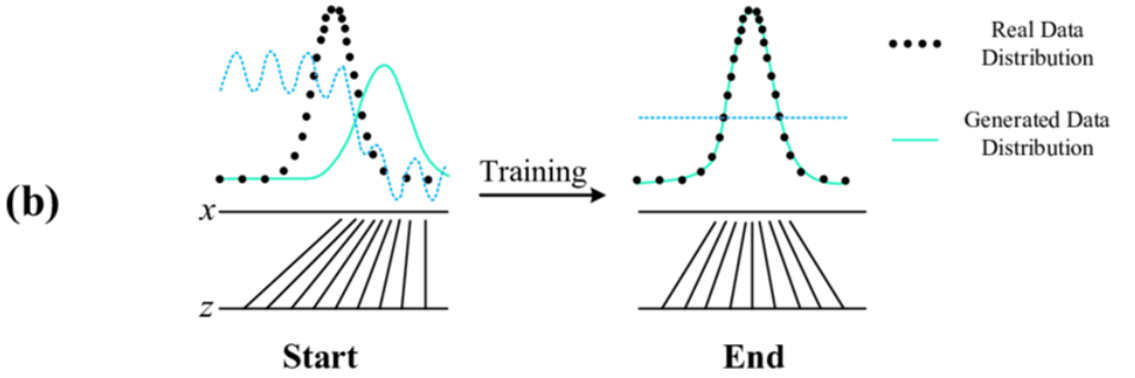
Generative Adversarial Network (GAN)

Two player game

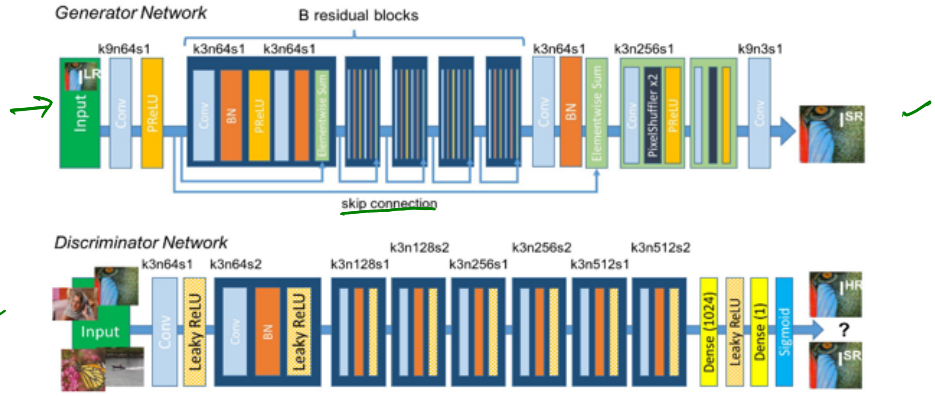
adversial







SRGAN: Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network[2]



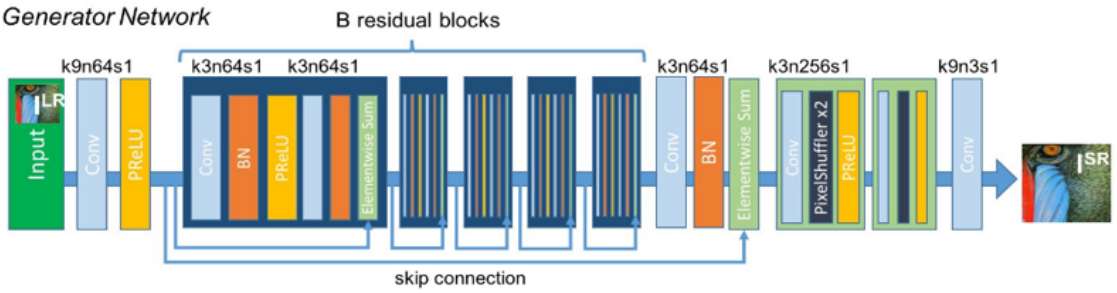
SRGAN is the first framework capable of generating photo-realistic images with 4× upscaling factors.

Perceptual Loss Function: Combines adversarial loss (trained to differentiate super-resolved images from real images) and content loss (based on perceptual similarity rather than pixel-wise similarity).

Adversarial Network Setup:

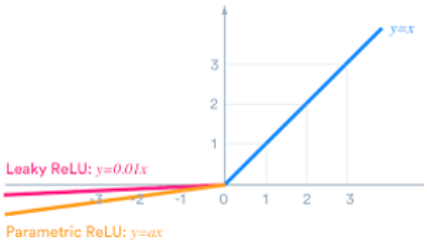
- Utilizes a **generator (G)** and **discriminator (D)** to solve an adversarial min-max problem where **G** aims to fool **D** into classifying generated images as real.
- **G** generates realistic images while **D** tries to distinguish between super-resolved and real images, encouraging perceptually superior solutions over pixel-wise error methods like MSE.

SRGAN: Generator network



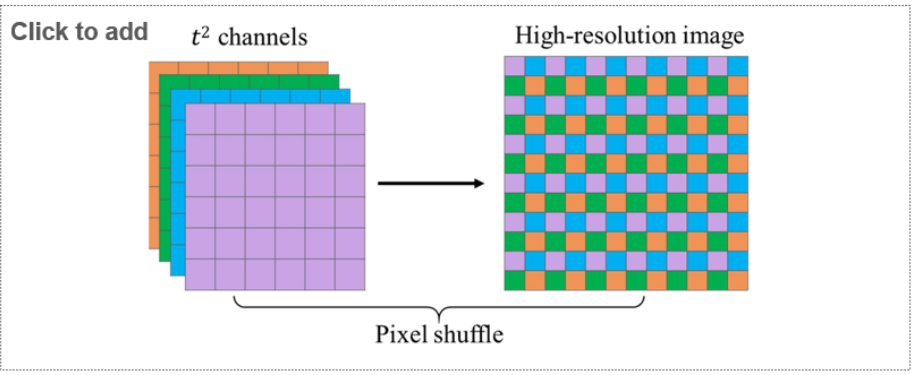
Generator Architecture:

- Built using a very deep network with B residual blocks, employing 3×3 convolutional layers, batch normalization, and ParametricReLU activations.
- Resolution is increased using Pixel Shuffle layers for high-quality super-resolution.



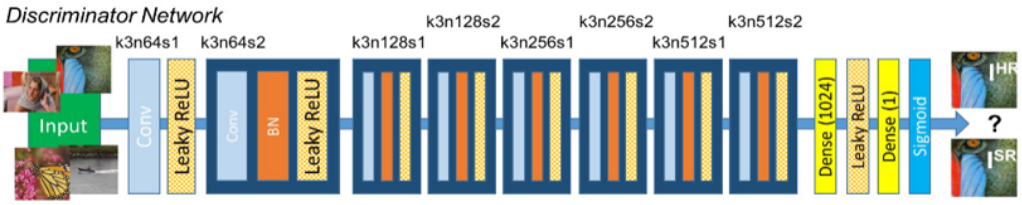
Parametric ReLU

Upsampling is done via Pixel shuffle



Rearranges elements in a tensor of shape $(*,C\times r^2,H,W)$ to a tensor of shape $(*,C,H\times r,W\times r)$

SRGAN: Discriminator architecture



Discriminator Architecture:

- Consists of 8 convolutional layers, with 3×3 filters increasing from 64 to 512, followed by two dense layers and a sigmoid activation for binary classification between real and super-resolved images.
- Uses LeakyReLU ($\alpha = 0.2$) to maintain gradient flow for better learning.
- Replaces max-pooling with strided convolutions to preserve more spatial detail during downsampling.

SRGAN: Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network

$$l^{SR} = \underbrace{\underbrace{l_{\mathrm{X}}^{SR}}_{\text{content loss}} + \underbrace{10^{-3} l_{Gen}^{SR}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

$$l_{VGG/i.j}^{SR} = \frac{1}{W_{i,j} H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

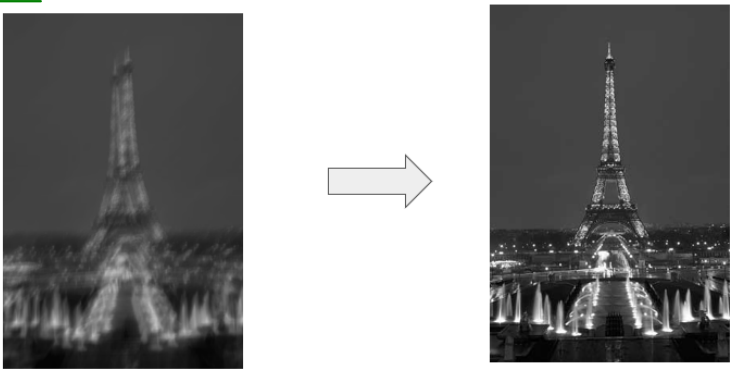
$$l_{Gen}^{SR} = \sum_{n=1}^{N} -\log D_{\theta_D}(G_{\theta_G}(I^{LR}))$$

Next task → Image deblurring

Till now

Denoising → MPRNE

Super resolution → SRGAN



Sources of blur object motion, Camera motion, defocus. (4)

Point Spread Function (PSF) or blur kernel

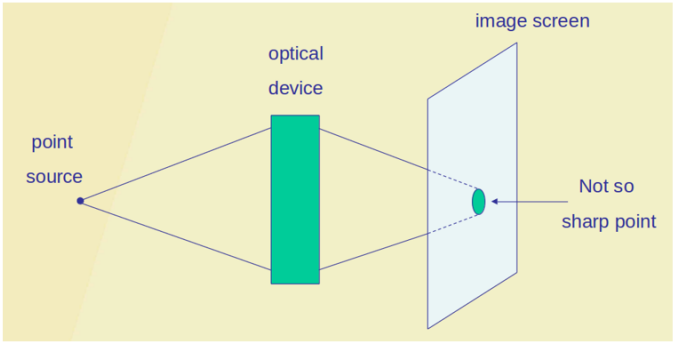
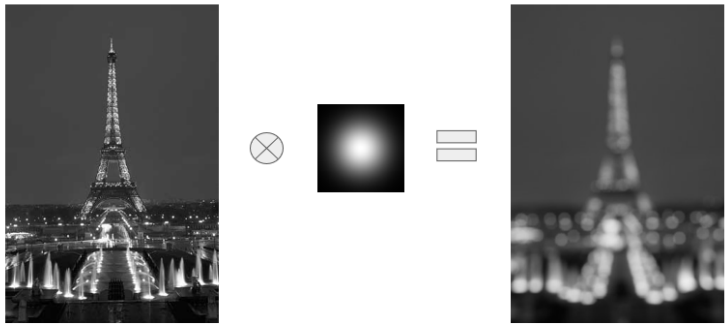
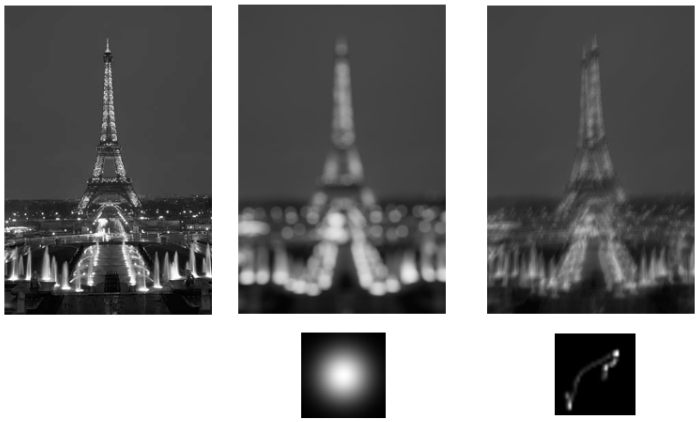


Image formation model

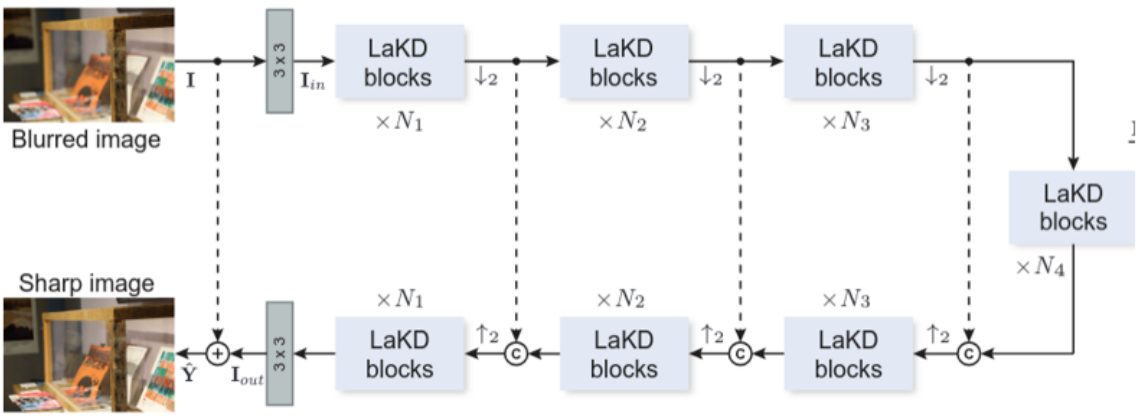


Ground truth PSF Blurred image

PSF - Defocus blur vs Motion blur



LaKDNet: Revisiting Image Deblurring with an Efficient ConvNet[4]



A pure CNN model (LaKDNet) to restore sharp, high-resolution images from blurry versions.

Utilizes depth-wise convolution with large kernels to maintain efficiency while modeling long-range pixel dependencies.

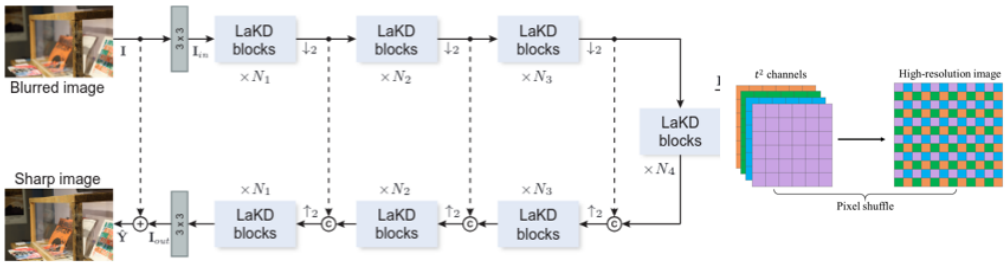
Architecture Type: U-shaped hierarchical network consisting of symmetric encoder-decoder modules.

Levels: Comprises 4 levels, each containing N LaKD blocks, where N varies across levels (N_1,N_2,N_3,N_4).

Input Processing: Takes an input image I with dimensions H×W×3 and extracts low-level features through an initial convolutional layer.

Feature Extraction: Passes the features into the encoder-decoder structure for blur removal, followed by a convolution layer to recover output features.

LaKDNet: Revisiting Image Deblurring with an Efficient ConvNet



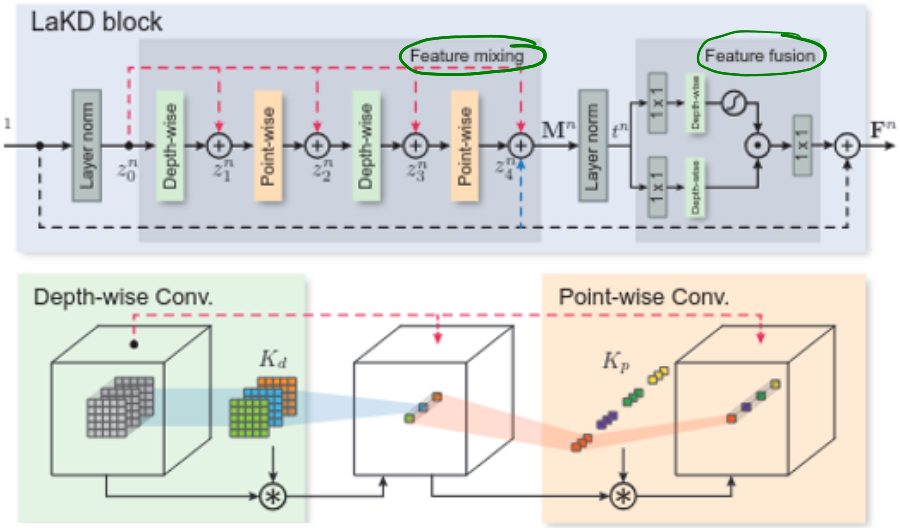
Downsampling/Upsampling: Utilizes pixel unshuffle for downsampling and pixel shuffle for upsampling.

Output Formation: Implements skip connections from input I to output I_{out}, resulting in a global residual structure

LaKD block

LaKD Block Motivation: Designed to explore local and global dependencies while achieving a large effective receptive field (ERF) using a fully convolutional approach.

Submodules: Comprises two main submodules—**feature mixer** and **feature fusion**.

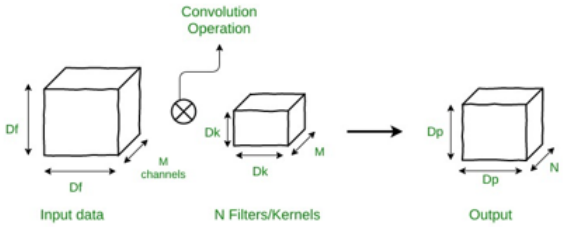


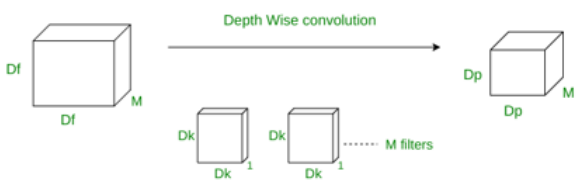
Feature Mixer:

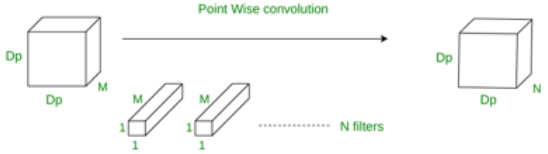
- Similar to depth-wise separable convolution, utilizing unusually large kernel sizes (e.g., 9×9).
- Incorporates a point-wise convolution (1×1)
- Separately mixes spatial intra-channel and depth-wise inter-channel features, enabling distant spatial location mixture.

Normal convolution vs (Depth-wise and Point-wise convolution)

Normal convolution: mixes spatial and channel information







Depth-wise convolution: only mixes spatial information

Point-wise convolution: only mixes channel information

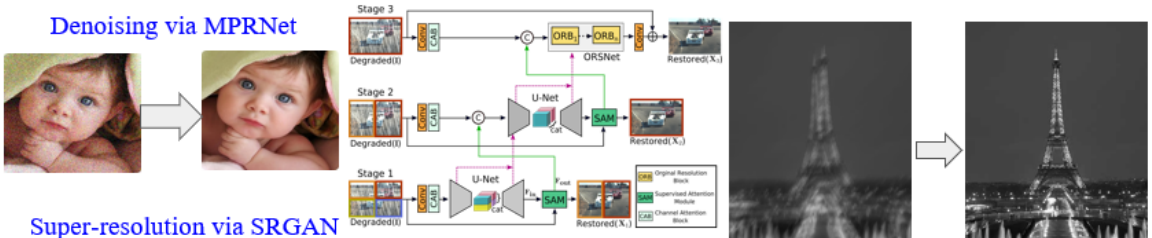
Lot of computation is saved by using depth-wise and point-wise convolution

Feature Fusion Module:

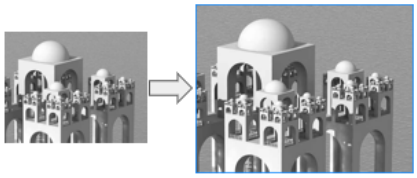
- Utilizes depth-wise convolution layers with 3×3 kernels for efficient local information encoding.
- Incorporates a gating mechanism with GELU activation to propagate and fuse features effectively.

Summary on Image Pre-processing

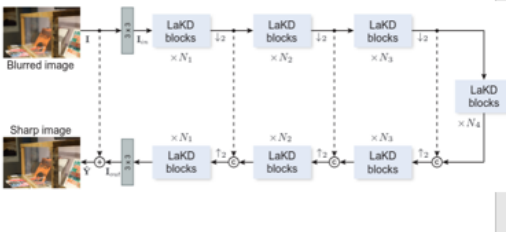
Denoising via MPRNet

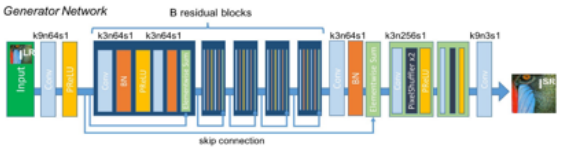


Super-resolution via SRGAN



Deblurring via LaKDNet





1) Which technique is utilized by LAKDNet to enhance feature extraction and aggregation?

- ○ (a) Multi-Scale Convolutional Layers
- ✓ (b) Dual Attention Mechanism (Spatial and Channel Attention)
- ○ (c) Recurrent Neural Networks (RNNs)
- ○ (d) Depthwise Separable Convolutions

2) Suppose you want to add random noise to an image, where the noise values are uniformly distributed in the range [0, 20]. Which code snippet will correctly accomplish this?

noise is to be uniform.

(a)

```
import torch

def add_random_noise(image):
    noise = torch.rand_like(image) * 20  # Uniform noise in [0, 20]
    noisy_image = image + noise
    return noisy_image
```

Correct

3) What is the role of the **discriminator** in a Super-Resolution GAN (SRGAN)?

- ○ (a) To generate high-resolution images from low-resolution inputs.
- ○ (b) To compare generated images with ground truth and assign a similarity score.
- ✓ (c) To distinguish between real high-resolution images and generated high-resolution images.
- ○ (d) To refine the generator's output by applying post-processing techniques.

you are fake

4) What is a common challenge faced by CNN-based techniques for capturing fine textures in super-resolution tasks?

- (a) Difficulty in preserving high-frequency details like edges and textures, leading to overly smooth results.
- (b) Inability to generate accurate low-resolution representations for input images.
- (c) Overfitting to fine textures, resulting in loss of structural consistency.
- (d) Reduced performance due to high dependency on pre-trained models.

5) How many levels are there in the LAKDNet architecture?

- (a) 2
- (b) 3
- (c) 4
- (d) 5

Consider the following code snippets for loading and modifying VGGNet-16 classification task with 10 classes:

```
import torch
import torch.nn as nn
from torchvision.models import vgg16, vgg19

# Block A: VGGNet-16 with modified classification head
class VGG16Modified(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG16Modified, self).__init__()
        self.vgg16 = vgg16(pretrained=True)
        self.vgg16.classifier[6] = nn.Linear(4096, num_classes)

    def forward(self, x):
        return self.vgg16(x)

model_a = VGG16Modified()

# Block B: VGGNet-19 with modified classification head
class VGG19Modified(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG19Modified, self).__init__()
        self.vgg19 = vgg19(pretrained=True)
        self.vgg19.classifier[6] = nn.Linear(4096, num_classes)

    def forward(self, x):
        return self.vgg19(x)

model_b = VGG19Modified()
```

```
# Block C: Loading VGGNet-16 and freezing feature extraction layers
vgg16_model = vgg16(pretrained=True)
for param in vgg16_model.features.parameters():
    param.requires_grad = False
vgg16_model.classifier[6] = nn.Linear(4096, 10)

# Block D: VGGNet-19 with feature extraction layers unfrozen
vgg19_model = vgg19(pretrained=True)
vgg19_model.classifier[6] = nn.Linear(4096, 10)
```

Which of the following statements are true about the provided blocks?

(a) Block A uses VGGNet-16 and modifies the classification head to support 10 classes.
(b) Block B and Block C both use VGGNet-19 but differ in how feature extraction layers are handled.
(c) Block C freezes the feature extraction layers in VGGNet-16 for transfer learning.
(d) Block D unfreezes feature extraction layers in VGGNet-19, making it trainable end-to-end.

Select the correct options: a, c and d.

The given below picture is likely to describe which architecture:

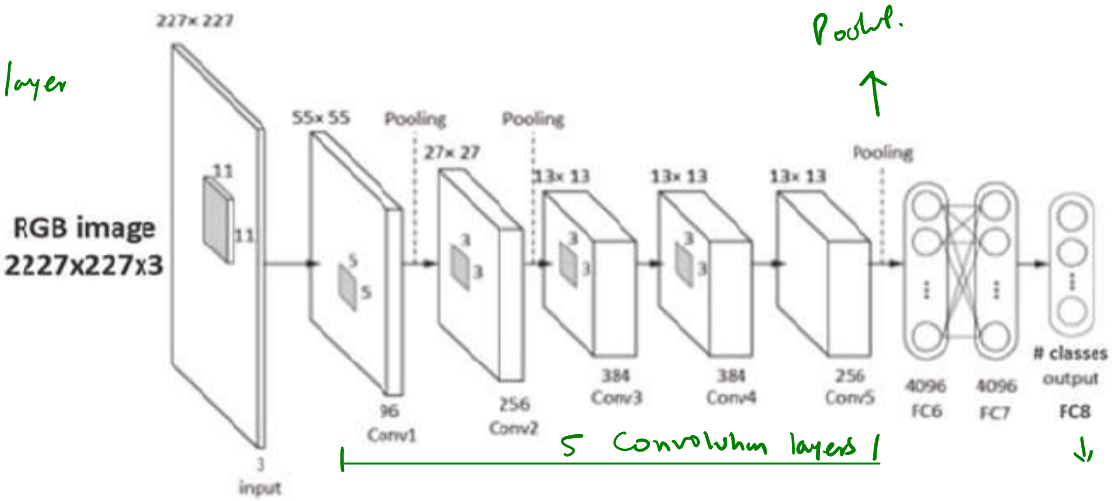


Figure 1: A simple CNN architecture for object detection

🔍 Why it's AlexNet:

- **Input size**: $227 \times 227 \times 3$ (close to AlexNet's original $224 \times 224 \times 3$, often 227 used due to padding/strides)
- **Conv1 layer**: 11×11 kernel with stride 4
- **Conv2 layer**: 5×5 kernel with stride 1 and padding
- **Conv3–5 layers**: 3×3 filters
- **ReLU activations** (implied)
- **Three fully connected layers (FC6, FC7, FC8)**: ending in classification
- **96, 256, 384, 384, 256 filters** in conv layers

Suppose I have an image of size 227×227. Which of the following code snippets correctly implements a Min Pooling operation with a window size of 2 × 2, stride of 2, and padding of 1 in PyTorch?

for minpooling we use max(-input, _ _ _)

```
import torch
import torch.nn.functional as F

# Min Pooling
def min_pool2d(x, kernel_size, stride, padding):
    return -F.max_pool2d(-x, kernel_size=kernel_size, stride=stride, padding=padding)

input_tensor = torch.randn(1, 3, 227, 227)
output = min_pool2d(input_tensor, kernel_size=2, stride=2, padding=1)
```

Consider a black-and-white image of dimension 227×227. Which of the following correctly demonstrates a function to flatten it along with the dimension of the newly formed vector?

img.flatten()

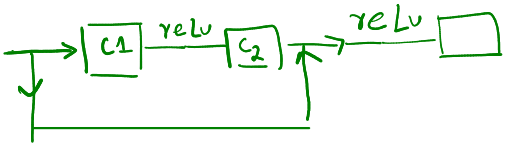
img.reshape(-1)

img.reshape((227*227))

```
img.reshape(-1) -> (227, 227, 1)    (51529,)

img.flatten() -> (51529,)

                                    (51529, )
img.reshape((227*227,)) -> (227*227*3, 1)

.reshape
img.flatten((227, 227)) -> (51529,)
```

Which of the following code snippets correctly implements the skip connection in ResNet?

```
import torch
import torch.nn as nn

class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

    def forward(self, x):
        residual = x
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x += residual
        return self.relu(x)
```

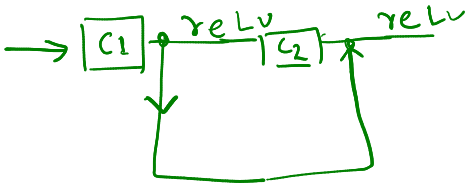


Correct.

```
import torch
import torch.nn as nn

class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x += x
        return self.relu(x)
```

X

```
import torch
import torch.nn as nn

class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

    def forward(self, x):
        residual = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x += residual
        return self.relu(x)
```



No this is not Skip Connection.

```
import torch
import torch.nn as nn

class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        return self.relu(x)
```

X

Suppose you are implementing a YOLO function with the following classes - Dog, cat, car, house.

Consider this image.

Max No. of bounding box = max no. of objects of classes.

Dog – 1
Cat – 0
Car – 6
House – 3

∴ 10

What is the maximum number of bounding box that can be found in this image?

The following code snippets represent different blocks in the Fast R-CNN pipeline. Identify the correct arrangement of these blocks in the Fast R-CNN architecture:

how does Fast R-CNN work

- Feature extractor.
- Region proposal
- RoI pooling
- FC Head

A. Block A

```
import torch
import torch.nn as nn

class RegionProposal(nn.Module):
    def __init__(self, in_channels):
        super(RegionProposal, self).__init__()
        self.conv = nn.Conv2d(in_channels, 256, kernel_size=3, stride=1, padding=1)
        self.cls_layer = nn.Conv2d(256, 18, kernel_size=1)
        self.reg_layer = nn.Conv2d(256, 36, kernel_size=1)

    def forward(self, x):
        features = torch.relu(self.conv(x))
        cls_logits = self.cls_layer(features)
        reg_deltas = self.reg_layer(features)
        return cls_logits, reg_deltas
```

∴ B A C D

B. Block B

```
import torchvision.models as models

class FeatureExtractor(nn.Module):
    def __init__(self):
        super(FeatureExtractor, self).__init__()
        vgg = models.vgg16(pretrained=True)
        self.features = vgg.features  # Use pre-trained VGG16 convolutional layers

    def forward(self, x):
        return self.features(x)
```

C. Block C

```
from torchvision.ops import roi_pool

class ROIPooling(nn.Module):
    def __init__(self, output_size=(7, 7)):
        super(ROIPooling, self).__init__()
        self.output_size = output_size

    def forward(self, feature_map, proposals):
        return roi_pool(feature_map, proposals, output_size=self.output_size)
```

D. **Block D**

```
class FullyConnectedHead(nn.Module):
    def __init__(self, in_features, num_classes):
        super(FullyConnectedHead, self).__init__()
        self.fc1 = nn.Linear(in_features, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.cls_score = nn.Linear(4096, num_classes)
        self.bbox_pred = nn.Linear(4096, num_classes * 4)  # 4 coordinates per class

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        cls_logits = self.cls_score(x)
        bbox_deltas = self.bbox_pred(x)
        return cls_logits, bbox_deltas
```

Which of the following code snippets correctly models the loss function in YOLO?

Options :

```
coord_loss = lambda_coord * mse_loss(predictions[..., :2], targets[..., :2])
conf_loss = mse_loss(predictions[..., 4], targets[..., 4])
noobj_loss = lambda_noobj * mse_loss(predictions[..., 4], torch.zeros_like(targets[..., 4]))
class_loss = mse_loss(predictions[..., 5:], targets[..., 5:])
total_loss = coord_loss + conf_loss + noobj_loss + class_loss
```

YOLO has.
- Coordinate loss
- Confidence — object
  — no object loss
- Classification loss
-

```
coord_loss = lambda_coord * mse_loss(predictions[..., :4], targets[..., :4])
conf_loss = lambda_noobj * mse_loss(predictions[..., 4], targets[..., 4])
class_loss = mse_loss(predictions[..., 5:], targets[..., 5:])
total_loss = coord_loss + conf_loss + class_loss
```

```
coord_loss = lambda_coord * mse_loss(predictions[..., :2], targets[..., :2])
conf_loss = mse_loss(predictions[..., 4], targets[..., 4])
class_loss = mse_loss(predictions[..., 5:], targets[..., 5:])
total_loss = coord_loss + conf_loss + class_loss
```

```
coord_loss = lambda_coord * mse_loss(predictions[..., :2], targets[..., :2])
noobj_loss = mse_loss(predictions[..., 4], torch.zeros_like(targets[..., 4]))
total_loss = coord_loss + noobj_loss
```

Consider the following predictions and ground truths for a binary classification problem:
- True Positives (TP): 30 - False Positives (FP): 10 - False Negatives (FN): 20
Which of the following correctly calculates the Precision and F1-Score for this classification task?

$$\text{Precision} = \frac{TP}{TP+FP} = \frac{30}{40} = 0.75$$

$$\text{Recall} = \frac{TP}{TP+FN} = \frac{30}{50} = 0.6$$

$$\therefore F1 = 2 \times \frac{0.75 \times 0.6}{0.7 + 0.6} = 2 \times \frac{0.45}{1.3} = 0.69$$

```
precision = TP / (TP + FP)  # 30 / (30 + 10) = 0.75
recall = TP / (TP + FN)     # 30 / (30 + 20) = 0.60
f1_score = 2 * (precision * recall) / (precision + recall)
```

How many convolutional layers are there in the original YOLO architecture? — 24

```
def calculate_iou(box1, box2):
    # Calculate intersection
    x1 = torch.max(box1[0], box2[0])
    y1 = torch.max(box1[1], box2[1])
    x2 = torch.min(box1[2], box2[2])
    y2 = torch.min(box1[3], box2[3])
    intersection = torch.clamp(x2 - x1, min=0) * torch.clamp(y2 - y1, min=0)

    # Calculate union
    area1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
    area2 = (box2[2] - box2[0]) * (box2[3] - box2[1])
    union = area1 + area2 - intersection

    # Compute IoU
    iou = intersection / union if union > 0 else 0
    return iou

# Bounding boxes: [x1, y1, x2, y2]
box1 = torch.tensor([0, 0, 2, 2])
box2 = torch.tensor([1, 1, 3, 3])

print(calculate_iou(box1, box2))
```

box1 = [0, 0, 2, 2] box2 = [1 1 3 3]

$x_1 = 1,\ y_1 = 1,\ x_2 = 2,\ y_2 = 2$

$\therefore$ intersection $= 1 \times 1 = 1$

area1 $= 2 \times 2 = 4$

area2 $= (3-1) \times (3-1) = 4$

$\therefore$ union $= 4 + 4 - 1 = 7$

$\therefore$ IOU $= \frac{1}{7} = 0.142$

What will the function output for the given bounding boxes?

Consider the following code snippets used in a multiscale deep network for single-image depth estimation. Both blocks play different roles in the network.

Which of the following best describes the roles of the two blocks?

A. Block A:

```
import torch
import torch.nn as nn

class DepthNetwork(nn.Module):
    def __init__(self):
        super(DepthNetwork, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=5, stride=2, padding=2),
            nn.ReLU()
        )
        self.fc = nn.Sequential(
            nn.Linear(128 * 28 * 28, 1024),
            nn.ReLU(),
            nn.Linear(1024, 128 * 56 * 56)
        )

    def forward(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        x = x.view(x.size(0), 128, 56, 56)
        return x
```

B. Block B:

```
import torch
import torch.nn as nn

class DepthRefinementNetwork(nn.Module):
    def __init__(self):
        super(DepthRefinementNetwork, self).__init__()
        self.refinement = nn.Sequential(
            nn.Conv2d(131, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 1, kernel_size=3, padding=1)
        )

    def forward(self, coarse_depth, rgb):
        x = torch.cat((coarse_depth, rgb), dim=1)  # Concatenate coarse depth and RGB
        x = self.refinement(x)
        return x
```

Options :

6406533531454. ✔ Block A generates high-level global features and outputs an initial coarse depth map, while Block B refines the depth map using local details from the RGB image and the coarse depth map.

6406533531455. ✖ Block A performs refinement of the depth map using concatenated coarse depth and RGB features, while Block B generates the coarse depth map from the input RGB image.

6406533531456. ✖ Both Block A and Block B are coarse networks, with Block B performing an additional refinement step.

6406533531457. ✖ Block A generates a low-resolution coarse depth map but does not include global features, while Block B produces fine-grained depth without refinement.

Which of the following code snippets correctly implements the **downsampling** operation in a U-Net architecture?

```
import torch
import torch.nn as nn

class DownSampleBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownSampleBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)
        return x
```

```
import torch
import torch.nn as nn

class DownSampleBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownSampleBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.conv(x)
        x = self.relu(x)
        return x
```

```
class DownSampleBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownSampleBlock, self).__init__()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.pool(x)
        return x
```

```
import torch
import torch.nn as nn

class DownSampleBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownSampleBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv(x)
        return x
```

Consider the following predictions and ground truth values:
- Predictions: [3.0,−0.5, 2.0, 7.0] - Ground Truth: [2.5, 0.0, 2.0, 8.0]
The Mean Squared Error (MSE) is calculated using one of the following code snippets. Identify the correct implementation and its result.

```
predictions = torch.tensor([3.0, -0.5, 2.0, 7.0])
targets = torch.tensor([2.5, 0.0, 2.0, 8.0])

mse = ((predictions - targets).abs()).mean()
print(mse.item())
# Output: 0.500
```

```
mse = ((predictions - targets) ** 2).mean()
print(mse.item())
# Output: 0.625
```

```
mse = ((predictions - targets) ** 2).sum() / predictions.size(0)
print(mse.item())
# Output: 0.625
```

```
mse = ((predictions - targets) ** 2).sum()
print(mse.item())
# Output: 2.500
```

In a stereo vision system, left-right disparity is used to compute depth by comparing corresponding points in the left and right images. Which of the following statements about left right disparity is correct?

6406533530101. ✖ Disparity is the absolute difference in pixel intensities between the left and right images.

6406533530102. ✖ Disparity increases as the depth of an object increases from the cameras.

6406533530103. ✔ Disparity is the horizontal shift between corresponding points in the left and right images.

6406533530104. ✖ Disparity is the vertical difference between points in the left and right images.

6406533530110. ✔ The generator produces high-resolution images from low-resolution inputs, while the discriminator distinguishes real high-resolution images from generated high-resolution images.

6406533530111. ✖ The generator distinguishes real high-resolution images from generated images, while the discriminator produces high-resolution images from low-resolution inputs.

6406533530112. ✖ Both the generator and discriminator produce high-resolution images from low-resolution inputs.

6406533530113. ✖ Both the generator and discriminator distinguish real high-resolution images from generated high-resolution images.

Which of the following code snippets correctly adds gaussian noise to an image?

```
import torch

def add_noise(image, noise_std=0.1):
    noise = torch.randn_like(image) * noise_std
    noisy_image = image + noise
    return noisy_image
```

What are the benefits of using depthwise and pointwise convolutions (as in depthwise separable convolutions) compared to standard convolutions?

Options :

6406533530118. ✔ They reduce the number of parameters and computational cost, making the model more efficient.

6406533530119. ✖ They improve the accuracy of the model by learning richer spatial features.

6406533530120. ✖ They reduce overfitting by performing regularization during convolution operations.

6406533530121. ✖ They allow convolutions to operate across both spatial dimensions and depth channels simultaneously.

```
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.upsample = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=9, stride=1, padding=4),
            nn.ReLU(),
            nn.Conv2d(64, 256, kernel_size=3, stride=1, padding=1),
            nn.PixelShuffle(upscale_factor=2),
            nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1)
        )
        self.refine = nn.Sequential(
            nn.Conv2d(3, 3, kernel_size=3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.upsample(x)
        return self.refine(x)
```

Which of the following are common use cases of super-resolution using SRGAN?

Options :

6406533530105. ✔ Enhancing medical imaging scans for improved diagnosis.

6406533530106. ✔ Increasing the resolution of satellite images for geographical analysis.

6406533530107. ✔ Generating high-quality images for video streaming and upscaling.

6406533530108. ✖ Translating text from one language to another in images.

6406533530109. ✔ Improving the resolution of historical or archival images.