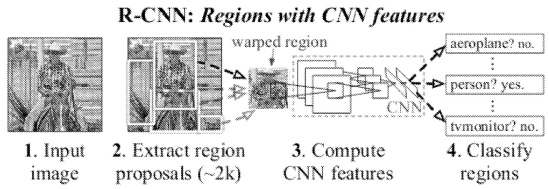
Two shot detectors

- Use two passes on the data:
  - Detect candidates (Region proposals)
  - Classify and regress objects in proposal (Including a no-object class)
- Methods:
  - RCNN - Region Proposals without Network. SVM classifier.
  - Fast RCNN - CNN Classifier
  - RFCN - Fully Convolutional RCNN
  - Faster RCNN - Use Region Proposal Network
  - Fast RCNN - Add Object Segmentation (Instance segmentation)

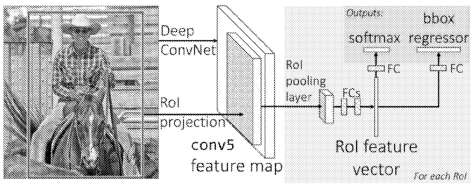
R-CNN

- Instead of processing candidates on a grid, use superpixels (selective search algorithm) to find candidates.
- Extract 2000 proposals.
- Advantages:
  - Candidates have different aspect ratios and scales.
  - Fewer candidates.
- Disadvantage:
  - Save and process each candidate separately (slow)



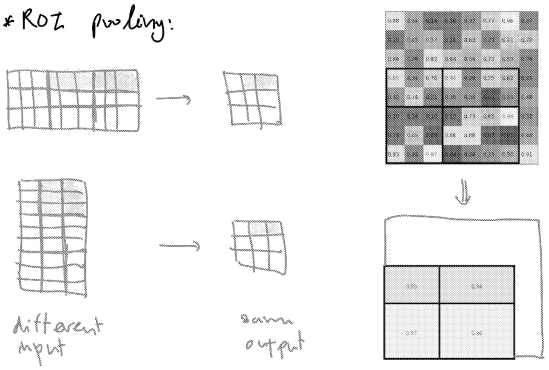
Fast R-CNN

- Convolution implementation instead of processing one candidate box at a time.
- Faster (X200) than R-CNN, but slower than YOLO.
- Proposals from experimental algorithm (selective search).



- ROI Pooling: Use max pooling to convert the region of interest to a fixed size (e.g., 7x7) for the fully connected (FC) layer.

Fast R-CNN

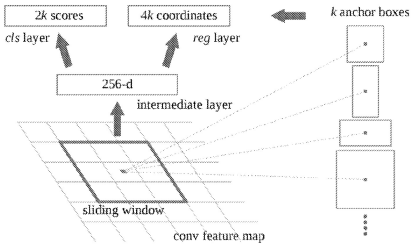


Region Proposal Network (RPN)

- **Region Proposal Network (RPN):**
  - **Part of Faster R-CNN**
  - **Train CNN to produce region proposals**
  - **Lightweight CNN (relatively simple task)**
- **Algorithm:**
  - **Slide window and classify (object/no object) using fast convolution implementation (sliding window).**
  - **Use k-anchor boxes at each location (different size and aspect ratio), around 200k boxes.**
  - **Classify/Regress K-boxes:**
    - **CLS layer- output 2K classifier scores (object/no object)**
    - **REG layer- output 4K regression boxes**

Region Proposal Network (RPN)

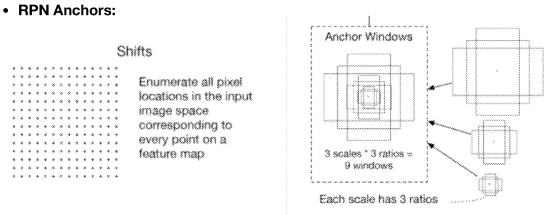
- **RPN Architecture:**



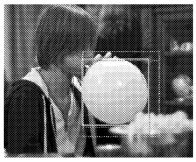
- **E.g., three aspect ratios and three scales give k = 9 anchor boxes at each location, leading to W x H x K total anchors.**

Region Proposal Network (RPN)

- **RPN Anchors:**

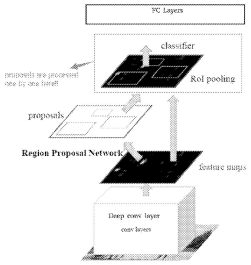


- **NMS - Select Candidates with Highest Confidence and Delete Candidates Overlapping (IOU greater than threshold).**



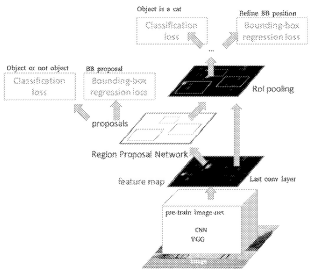
Faster R-CNN

- **Faster R-CNN (Fast R-CNN + RPN):**
  - **Use RPN for region proposals**
  - **Use ROI pooling, then classify using dense layers**
  - **Share feature extraction layers**



Faster R-CNN

- **Faster R-CNN loss:**
  - **RPN - classification (object/no object) + box regression**
  - **Final - classification (class) + refined box regression**



Faster R-CNN

- **Faster R-CNN Algorithm summary:**

  1. **Get pre-trained CNN (e.g., VGG16 or ResNet101)**
  2. **Get feature map from last (deep) convolution layer**
  3. **Train RPN: Object/no object classification and box regression**
  4. **Apply ROI pooling to proposals**
  5. **Use FC (dense) layers for class classification and refined box regression**

R-FCN

- **R-FCN (Region-based Fully Convolutional Network)**
  - **Use convolutions instead of costly dense layers, applied for each region**
  - **Compute position-sensitive score maps where each score map is sensitive to a different region. A (K x K) grid will produce K^2 score maps**
  - **Each score map has C+1 channels (C classes + one no-object class)**

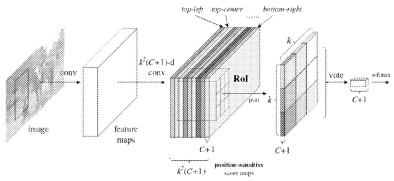


Figure 1: Key idea of **R-FCN** for object detection. In this illustration, there are $k \times k = 3 \times 3$ position-sensitive score maps generated by a fully convolutional network. For each of the $k \times k$ bins in an RoI, pooling is only performed on one of the k^2 maps (marked by different colors).

- **Position-Sensitive ROI Pooling to pool one score map at each location (K x K grid)**

R-FCN

- **Use RPN for proposals**

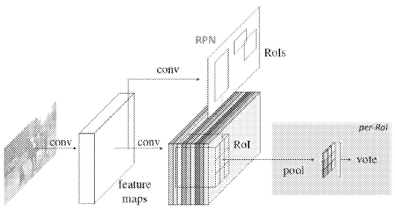
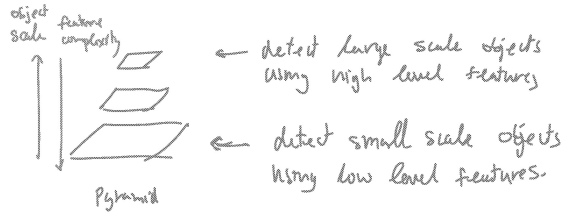


Figure 2: Overall architecture of R-FCN. A Region Proposal Network (RPN) [18] proposes candidate RoIs, which are then applied on the score maps. All learnable weight layers are convolutional and are computed on the entire image; the per-RoI computational cost is negligible.

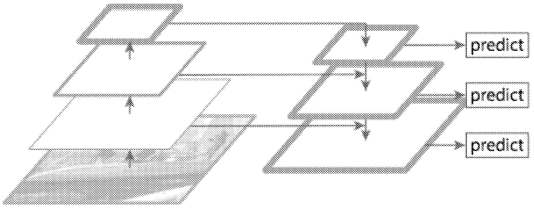
Feature Pyramid Network (FPN)

- Feature Pyramid Network (FPN)
  - Used by Faster R-CNN
  - Goal: Detect objects at different scales using higher-level features



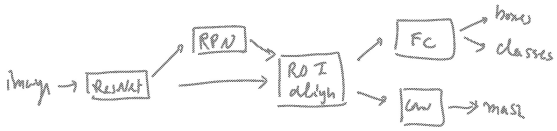
Feature Pyramid Network (FPN)

- Use Lateral Connections in FPN:
  - Maintain strong semantic features at each level
  - Lateral connections: Combine features at corresponding levels



Mask R-CNN

- Mask R-CNN
  - Add instance segmentation to Faster R-CNN

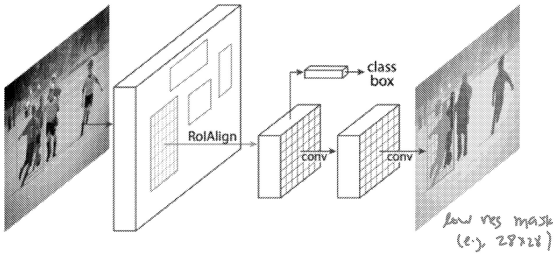


$$Loss = L_{cls} + L_{reg} + L_{seg}$$

$$L_{cls}^{RPN} + L_{cls}^{obj} \qquad L_{reg}^{RPN} + N_{reg}^{obj}$$

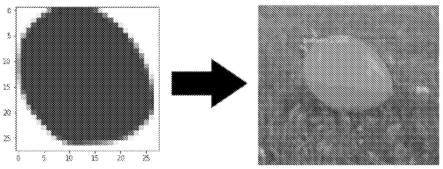
Mask R-CNN

* Mask RCNN architecture:



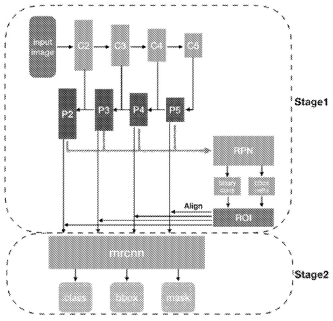
Mask R-CNN

- Predicted masks are scaled up to original resolution.



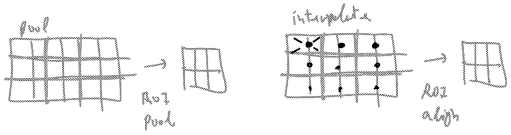
Mask R-CNN

- Mask R-CNN uses FPN as a backbone



Mask R-CNN

- **ROI Align:**
  - **ROI pooling extracts small feature maps of fixed size. It is sensitive to alignment shifting (shifting the grid will change results)**
  - **ROI align solves this by sampling with interpolation**



Object detection example

```
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Dense, Flatten, Conv2D, Reshape, Input
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt

# Load the Oxford-IIIT Pet dataset
dataset, info = tfds.load('oxford_iiit_pet', split='train', with_info=True,
as_supervised=False)

# Preprocessing function
def preprocess_data(sample):
    image = sample['image']
    bbox = sample['objects']['bbox']
    label = sample['objects']['label']

    # Resize image to a fixed shape (224x224)
    image = tf.image.resize(image, (224, 224))
    image = image / 255.0  # Normalize pixel values

    # Convert bbox to [xmin, ymin, xmax, ymax] format and normalize coordinates
    bbox = tf.stack([bbox[:, 1], bbox[:, 0], bbox[:, 3], bbox[:, 2]], axis=1)  #
Convert to [xmin, ymin, xmax, ymax]

    return image, {'bbox': bbox, 'class': tf.one_hot(label, depth=37)}  # 37 classes

# Apply preprocessing to the dataset
train_data = dataset.map(preprocess_data)
```

Object detection example

```
# Batch and prefetch the dataset for performance
batch_size = 32
train_data = train_data.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)

# Sample dataset info
for image, labels in train_data.take(1):
    print("Image shape: ", image.shape)
    print("Bounding box: ", labels['bbox'].shape)
    print("Class label: ", labels['class'].shape)

# Build the object detection model
def build_model(input_shape=(224, 224, 3), num_classes=37):
    # Load pre-trained MobileNetV2 without the top layers
    base_model = MobileNetV2(input_shape=input_shape, include_top=False,
weights='imagenet')
    base_model.trainable = False  # Freeze the base model

    # Input layer
    input_layer = Input(shape=input_shape)

    # Pass inputs through the base model
    x = base_model(input_layer)
    x = Flatten()(x)

    # Bounding box regression head (output 4 values: xmin, ymin, xmax, ymax)
    bbox_regression = Dense(128, activation='relu')(x)
    bbox_regression = Dense(64, activation='relu')(bbox_regression)
    bbox_regression = Dense(4, name='bbox')(bbox_regression)  # 4 outputs for bbox

    # Classification head (output number of classes)
    classification = Dense(128, activation='relu')(x)
    classification = Dense(64, activation='relu')(classification)
    classification = Dense(num_classes, activation='softmax', name='class')  # 37
classes
```

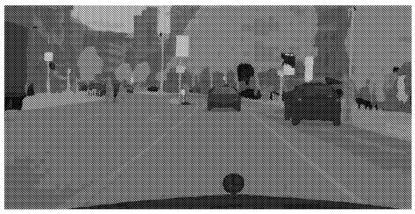
Object detection example

```
    # Create the model
    model = Model(inputs=input_layer, outputs=[bbox_regression, classification])

    return model

# Build the model
model = build_model()

# Compile the model with two loss functions: one for bounding boxes and one for
classification
model.compile(optimizer='adam',
              loss={'bbox': 'mean_squared_error', 'class': 'categorical_crossentropy'},
              metrics={'bbox': 'mse', 'class': 'accuracy'})

# Print model summary
model.summary()

# Train the model (replace with more epochs for better performance)
epochs = 5
history = model.fit(train_data, epochs=epochs)

# Plot training history
def plot_history(history):
    plt.figure(figsize=(12, 5))

    # Loss plot
    plt.subplot(1, 2, 1)
    plt.plot(history.history['bbox_loss'], label='BBox Loss')
    plt.plot(history.history['class_loss'], label='Class Loss')
    plt.title('Loss during Training')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
```

Object detection example

```
    # Accuracy plot
    plt.subplot(1, 2, 2)
    plt.plot(history.history['class_accuracy'], label='Classification Accuracy')
    plt.title('Accuracy during Training')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.show()

plot_history(history)

# Testing the model
def test_model_on_single_image(model, dataset):
    for image, labels in dataset.take(1):  # Take 1 batch for testing
        predictions = model.predict(image)

        # Predicted bounding boxes and class labels
        predicted_bbox = predictions[0]
        predicted_class = predictions[1].argmax(axis=1)

        # Show the first image and its predicted bounding box
        plt.figure(figsize=(6, 6))
        plt.imshow(image[0])

        # Extract predicted bounding box (assuming batch_size=1)
        xmin, ymin, xmax, ymax = predicted_bbox[0]
        plt.gca().add_patch(plt.Rectangle((xmin * 224, ymin * 224), (xmax - xmin) *
224, (ymax - ymin) * 224,
                                          fill=False, edgecolor='red', linewidth=2))

        plt.title(f"Predicted class: {predicted_class[0]}")
        plt.show()

# Test on a single image
test_model_on_single_image(model, train_data)
```

Semantic segmentation



- **Segment (Cluster) According to Class (e.g., road)**
- **Classify each pixel**
- **Evaluate using mean IOU of masks**

- **Methods**
  - **FCN (Fully Convolutional Segmentation)**
  - **U-Net: Add skip connections between encoder and decoder branches**
  - **DeepLab: Add spatial pyramid pooling (SPP)**

FCN (Fully Convolutional Segmentation)

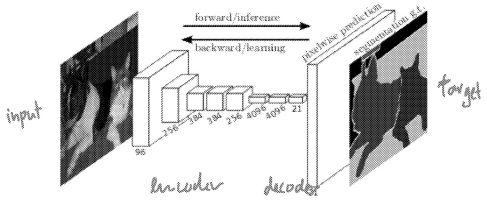


Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

FCN (Fully Convolutional Segmentation)

- **Replace dense layers (for pixel classification) with convolution.**
- **Can segment arbitrary-size images with fewer coefficients compared to dense layers.**

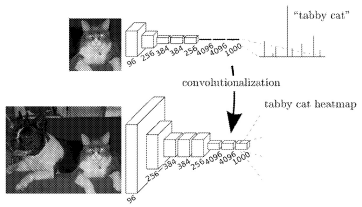


Figure 2. Transforming fully connected layers into convolution layers enables a classification net to output a heatmap. Adding layers and a spatial loss (as in Figure 1) produces an efficient machine for end-to-end dense learning.

FCN (Fully Convolutional Segmentation)

- **FCN versions:**

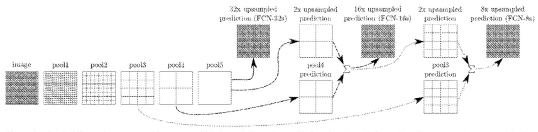


Figure 3. Our DAG nets learn to combine coarse, high layer information with fine, low layer information. Layers are shown as grids that reveal relative spatial coarseness. Only pooling and prediction layers are shown; intermediate convolution layers (including our converted fully connected layers) are omitted. Solid line (FCN-32s): Our single-stream net, described in Section 4.1, upsamples stride 32 predictions back to pixels in a single step. Dashed line (FCN-16s): Combining predictions from both the final layer and the `pool4` layer, at stride 16, lets our net predict finer details, while retaining high-level semantic information. Dotted line (FCN-8s): Additional predictions from `pool3`, at stride 8, provide further precision.

U-Net

- **Encoder-decoder architecture**

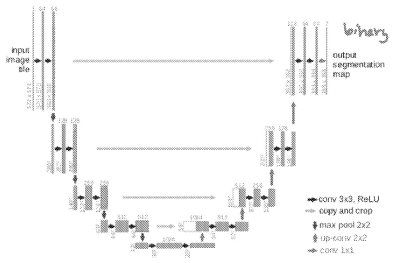


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

U-Net

- **Skip Connections**
  - **Skip connections between encoder and decoder**
  - **Up-sampled output is concatenated with cropped input (cropped due to loss of pixels in every convolution layer)**
- **Fully convolutional**
  - **3x3 convolution produces segmented output (probability)**
  - **The number of filters in the final convolution layer equals the number of classes**

Spatial Pyramid Pooling (SPP)

- **Goal: support objects at different scales**
- **Use multiple pooling levels as input to the classification layer**

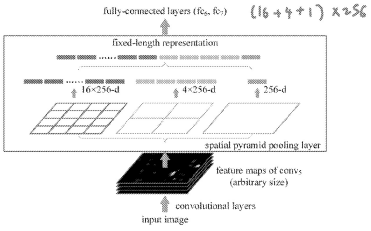
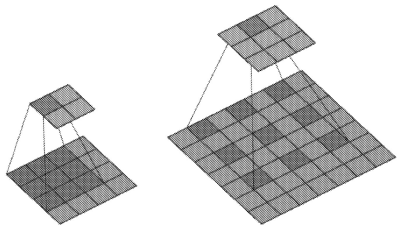


Figure 3: A network structure with a **spatial pyramid pooling layer**. Here 256 is the filter number of the $conv_5$ layer, and $conv_5$ is the last convolutional layer.

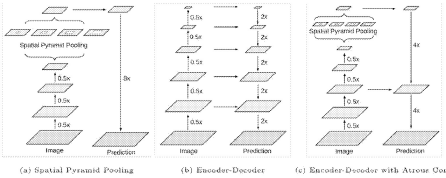
Spatial Pyramid Pooling (SPP)

- **Spatial Pyramid Pooling results in increased dimensions**
- **Use Atrous (Dilated) Convolution**
- **This results in Atrous Spatial Pyramid Pooling (ASPP)**



DeepLab

- **Encoder-Decoder Architecture**
- **Use Atrous Spatial Pyramid Pooling (ASPP) with skip connections**

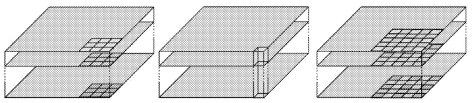


(a) Spatial Pyramid Pooling (b) Encoder-Decoder (c) Encoder-Decoder with Atrous Conv

Fig. 1. We improve DeepLabv3, which employs the spatial pyramid pooling module (a), with the encoder-decoder structure (b). The proposed model, DeepLabv3+, contains rich semantic information from the encoder module, while the detailed object boundaries are recovered by the simple yet effective decoder module. The encoder module allows us to extract features at an arbitrary resolution by applying atrous convolution.

DeepLab

- **Convolution Implemented as stepwise separable convolution (as in MobileNets)**



(a) Depthwise conv. (b) Pointwise conv. (c) Atrous depthwise conv.

Fig. 3. 3×3 Depthwise separable convolution decomposes a standard convolution into (a) a depthwise convolution (applying a single filter for each input channel) and (b) a pointwise convolution (combining the outputs from depthwise convolution across channels). In this work, we explore *atrous separable convolution* where atrous convolution is adopted in the depthwise convolution, as shown in (c) with $rate = 2$.

DeepLab

* DeepLab architecture (v3+):

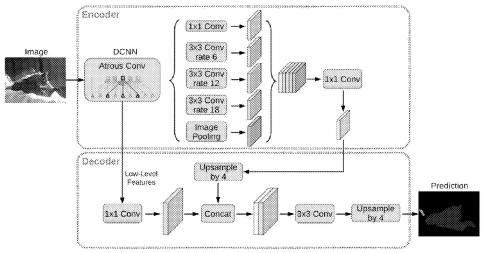
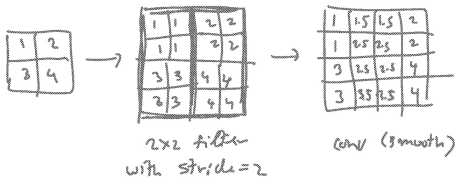


Fig. 2. Our proposed DeepLabv3+ extends DeepLabv3 by employing a encoder-decoder structure. The encoder module encodes multi-scale contextual information by applying atrous convolution at multiple scales, while the simple yet effective decoder module refines the segmentation results along object boundaries.

Upsampling



- **A stride without overlap will produce block artifacts**
- **Use Gaussian convolution or Bilinear interpolation**
- **Goal: Let the network learn how to upsample**
- **Use transpose convolution with learnable parameters**

Transpose convolution

- **1D convolution**

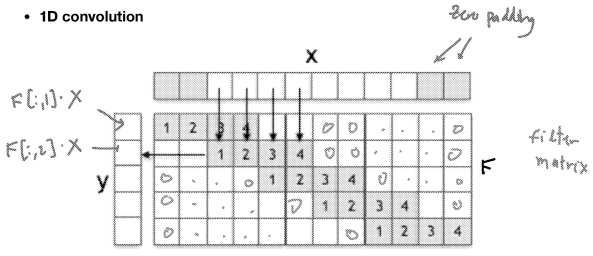
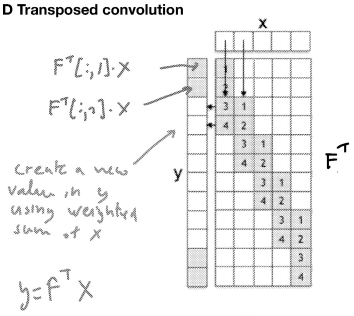


Figure 2: Convolution with stride 2 in 1D

$y = Fx$

Transpose convolution

- **1D Transposed convolution**



(a)

Figure 3: (a) Transposed convolution (stride = 2)