Two shot detectors

* use two passes on the data:
  - detect candidates (region proposals) region proposal
  - classify and regress objects in proposal (including a no-object class)

* Methods:

More complex

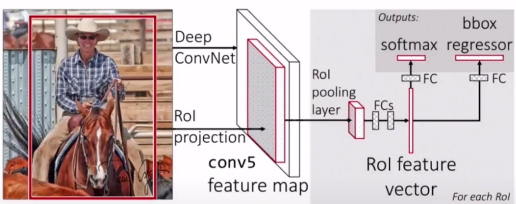
- Fast-RCNN - CNN classifier
- R-FCN - Fully convolutional RCNN
- Faster RCNN - use region proposal network
- Mask RCNN - add object segmentation (instance

Two shot detectors

* RCNN:
  - Instead of processing candidates on a grid use super pixels (selective search alg.) to find candidates
  - Extract 2000 proposals
  - Advantage:
    - candidates have different aspect ratios and scales
    - Fewer candidates

This page does not seem important

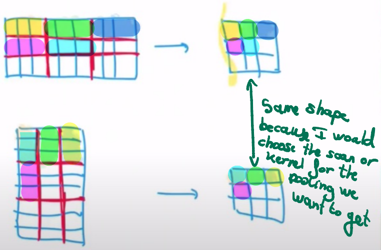
* Fast-RCNN architecture:



* ROI-pooling - use max pooling to convert the region of interest to a fixed size (e.g. 7x7) to the FC classifier

Two shot detectors

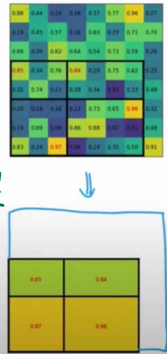
* ROI pooling:



Same shape because I would choose the scan or kernel for the pooling we want to get

different input

same output



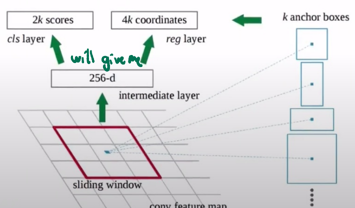
Two shot detectors

* Region proposal Network (RPN)
  - part of faster-RCNN
  - train CNN to produce region proposals
  - Light weight CNN (relatively simple task)

* Algorithm:
  - slide window and classify (object / no-object) — main idea
    use fast convolution implementation (sliding window)
  - use k anchor boxes at each location
    (different size and aspect ratio)
    ⇒ around 200k boxes

Two shot detectors

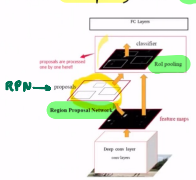
* RPN algorithm contd.
- classify / regress k-boxes
  - cls-output 2k classification scores (obj/no-obj)
  - reg-output 4k regression boxes



e.g. 3 aspect ratios and 3 scales gives k=9 anchor boxes at each location
⇒ W×H×k total anchors

Two shot detectors

* Faster RCNN (fast RCNN + RPN):
  - use RPN for region proposals
  - use ROI-pooling then classify using dense layers



- share feature extraction layer

Two shot detectors

* RFCN (Region based fully convolutional Network)
  - use convolutions instead of costly dense layers that are applied for each region)
  - create position sensitive score maps where each score map is sensitive to another region.

  k×k grid ⇒ k^2 score maps

  each score map has C+1 channels (c classes + one no-obj class)

Two shot detectors

* Position sensitive RoI Pooling:

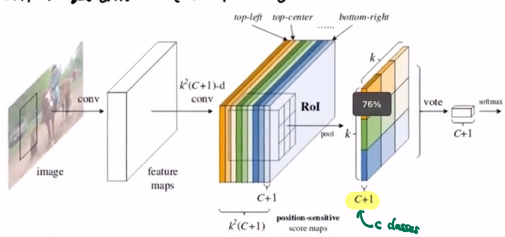


Figure 1: Key idea of **R-FCN** for object detection. In this illustration, there are $k \times k = 3 \times 3$ position-sensitive score maps generated by a fully convolutional network. For each of the $k \times k$ bins in an RoI, pooling is only performed on one of the k^2 maps (marked by different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

* RFCN architecture:
  - use RPN for proposals

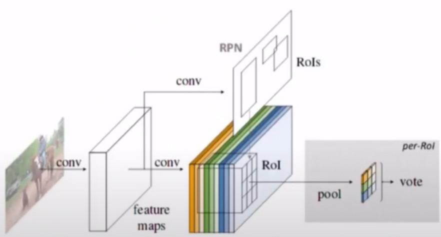
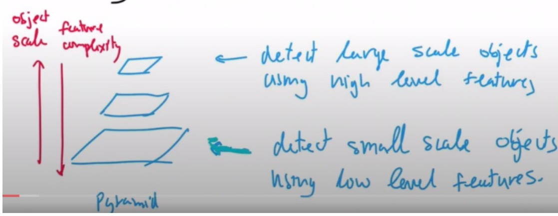


Figure 2: Overall architecture of R-FCN. A Region Proposal Network (RPN) [18] proposes candidate RoIs, which are then applied on the score maps. All learnable weight layers are convolutional and are computed on the entire image; the per-RoI computational cost is negligible.

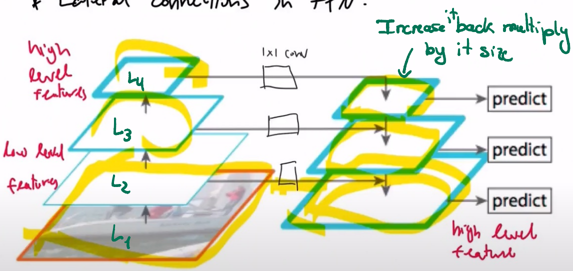
Two shot detectors

* Feature pyramid network (FPN)
  - used by faster-RCNN
  - Goal: detect objects at different scales using higher level features



Two shot detectors

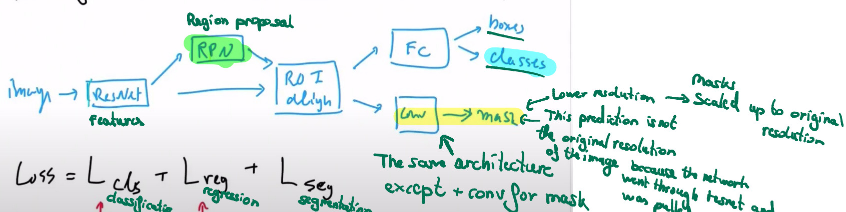
* Lateral connections in FPN:



- maintain strong semantic features at each level
- lateral connections combine features at

Two shot detectors

* Mask RCNN: → Try to solve the problem of instant segmentation
  - Add segmentation to faster RCNN → instance segmentation

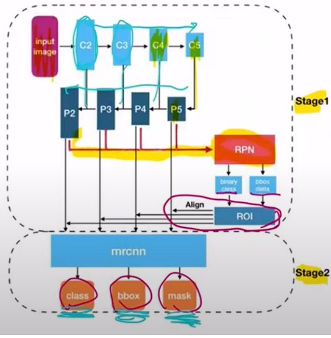


$$Loss = \underbrace{L_{cls}}_{\text{classification}} + \underbrace{L_{reg}}_{\text{regression}} + \underbrace{L_{seg}}_{\text{segmentation}}$$

$L_{cls} = L^{RPN}_{cls} + L^{obj}_{cls}$

$L_{reg} = L^{RPN}_{reg} + L^{obj}_{reg}$

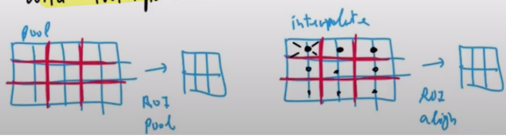
* Mask - RCNN uses FPN as back bone:



Two shot detectors

* ROI align: ← Try to solve the problem of ROI pulling → You are sensitive to alignment

- RoI-pooling extracts small feature maps of fixed size.
- RoI-pooling is sensitive to alignment ← Because regions are fixed
  shifting the grid will change results.
- RoI-align solves this by sampling with interpolation




```python
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
```

breeds of dogs ('oxford_iiit_pet')

} Three things that I need (image, bbox, label)