# DSCI 565: COMPUTER VISION



Ke-Thia Yao

Lecture 20: 2025 November 10

# Computer Vision

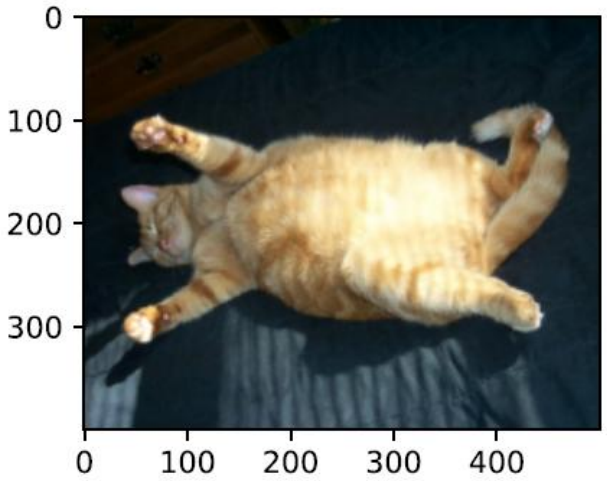


- Techniques to improve image classification
  - Image Augmentation
  - Fine-tuning
- Computer vision tasks
  - Object detection
  - Semantic segmentation
  - Style transfer

# Image Augmentation



- Deep learning networks require large datasets
- Image augmentation generates similar by distinct training examples by performing random transformations
- Image augmentation allows networks to generalize better
- An image of a cat is still a cat, if we applied transformations
  - Cropping the image
  - Flipping or rotating the image
  - Changing the color



# Notebook



- chapter_computer-vision/image-augmentation.ipynb
  - What did you do that cat?!?

# Transfer Learning

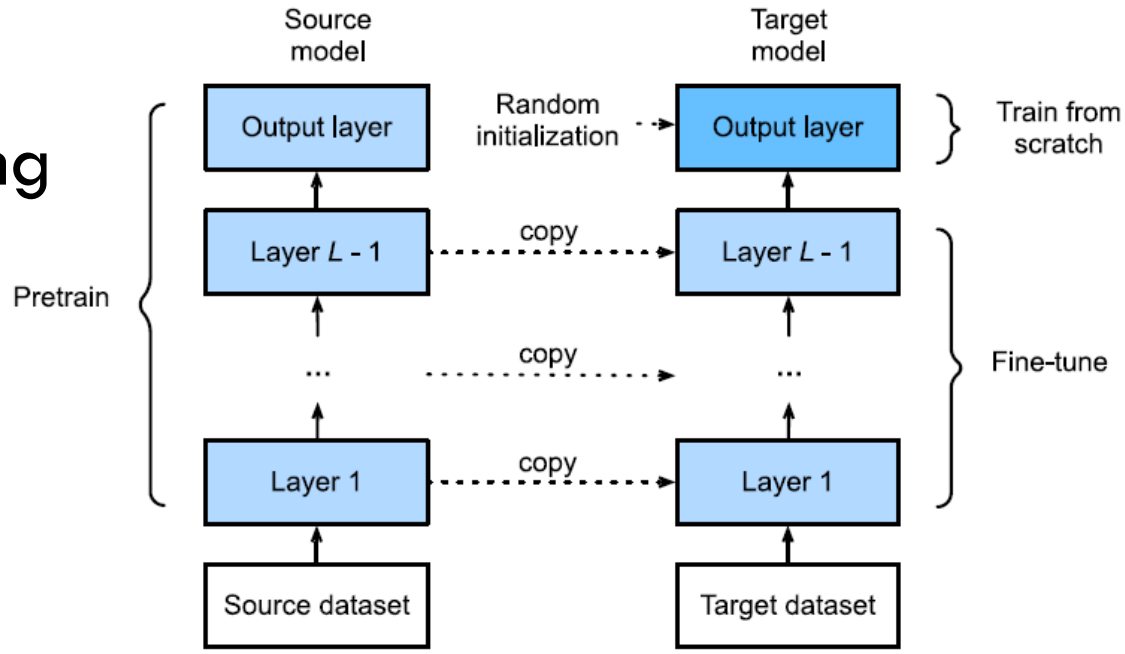


- Suppose we want to create an app to recognize architectural styles of landmarks
- We could
  - Create a labeled dataset with say 100 architectural styles and with say 1000 images for each architectural style
  - Train a new deep learning model on this dataset
- But a deep model may overfit on this small data set
  - This dataset is only 10% of the size of ImageNet
  - Gathering additional labeled landmark images is time consuming
- Better to apply transfer learning
  - Train the model on a large source dataset
  - Then fine-tuning the model of the target dataset

# Fine-Tuning



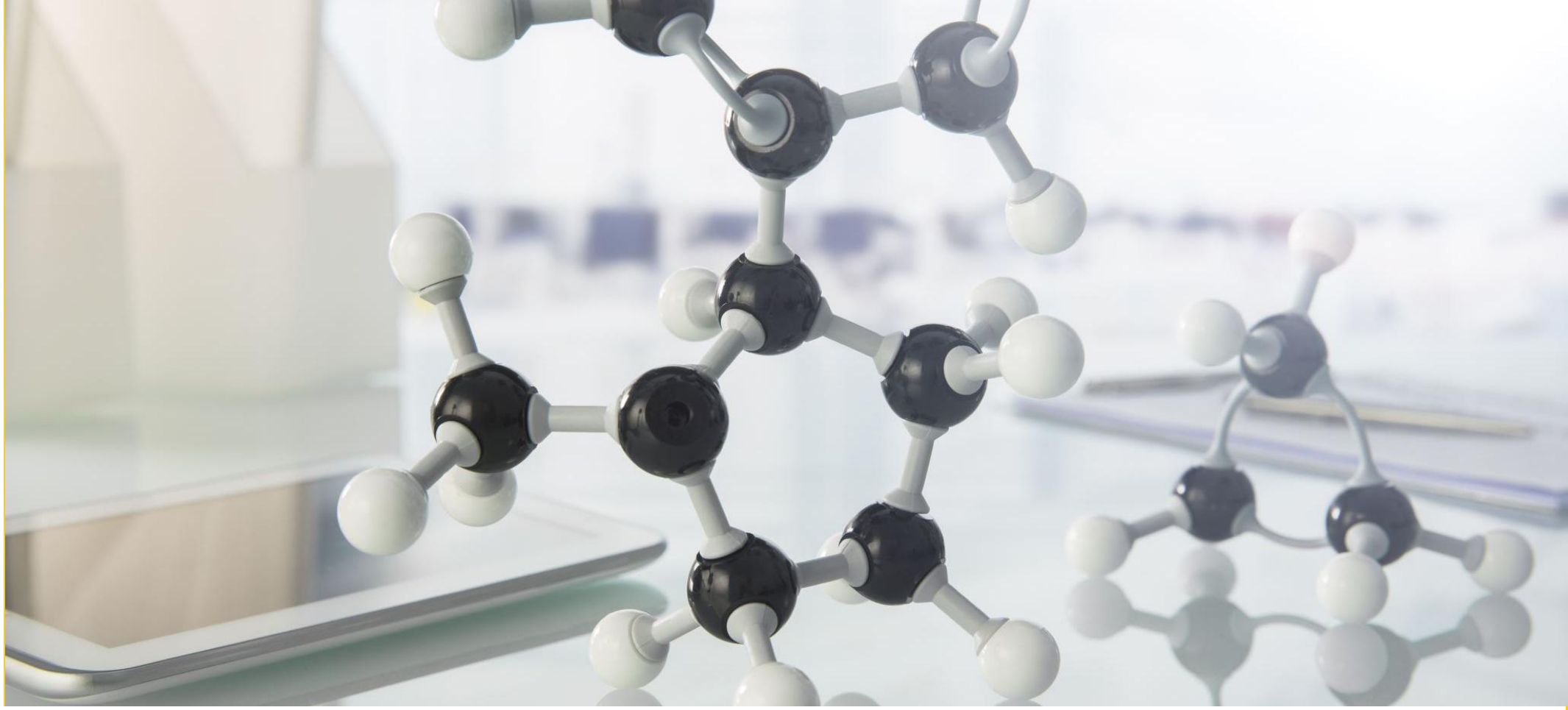
- Pretrain a source model using the source dataset
- Create a new target model by copying from the source model (layers and parameters), except the output layer
- Create a new output layer to the target model
- Train the target model on the target dataset



# Notebook



- chapter_computer-vision/fine-tuning.ipynb
  - Hot dog recognition



8

# Object Detection

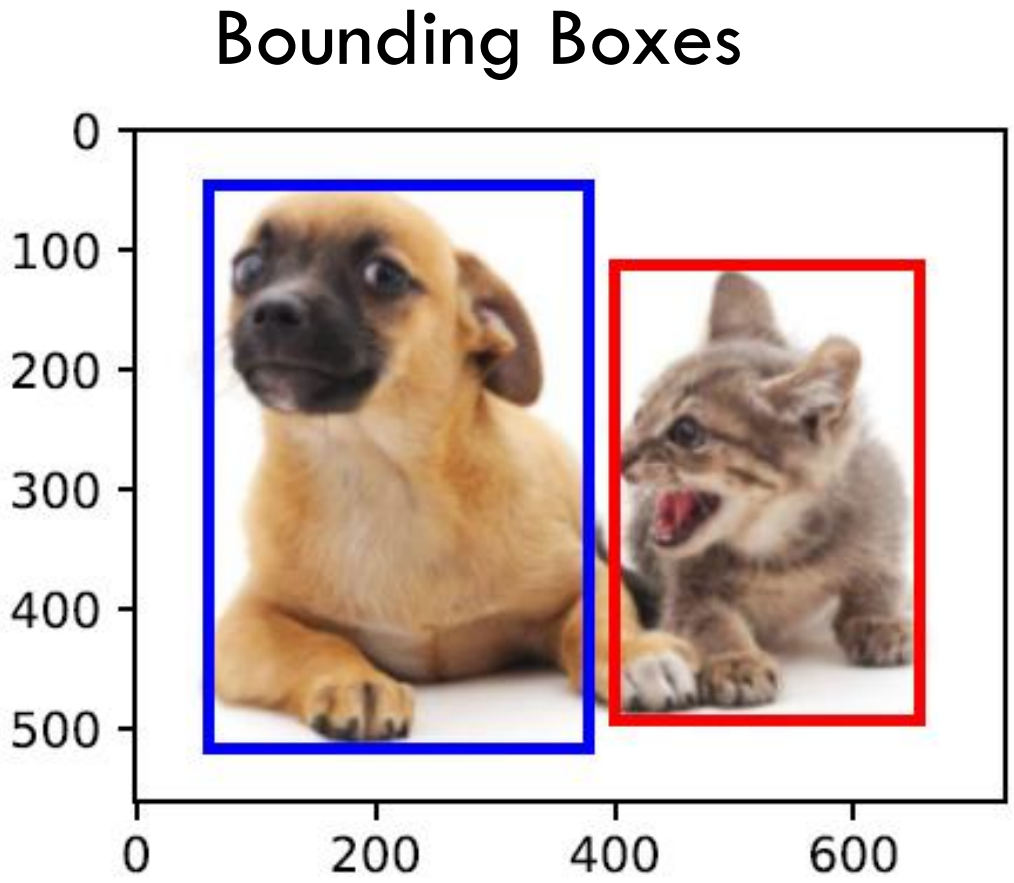
DALL-E generated image for “object detection”

# Object Detection (aka Object Recognition)



- In image classification, we assume an image consists of one major object
- In object detection, the are may be multiple objects of interest
- Model must create one or more bounding boxes to locate the objects

↳ bounding box + label



# Anchor Boxes



- Object detection algorithms usually sample many regions of the input image to determine whether these regions contain objects of interest
- Using sliding windows of varying sizes to scan the image is time consuming
- Anchor boxes provides a way to sample these regions

# Anchor Boxes



- Anchor boxes are predefined bounding boxes
  - with varying scales $(s_1, s_2, \ldots, s_n)$
  - and aspect ratios $(r_1, r_2, \ldots, r_m)$
  - centered on each pixels $(w \times h)$
- Total number of anchor boxes is $whnm$
- Simplify by choosing a subset of scale and ratio combinations

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1)$$

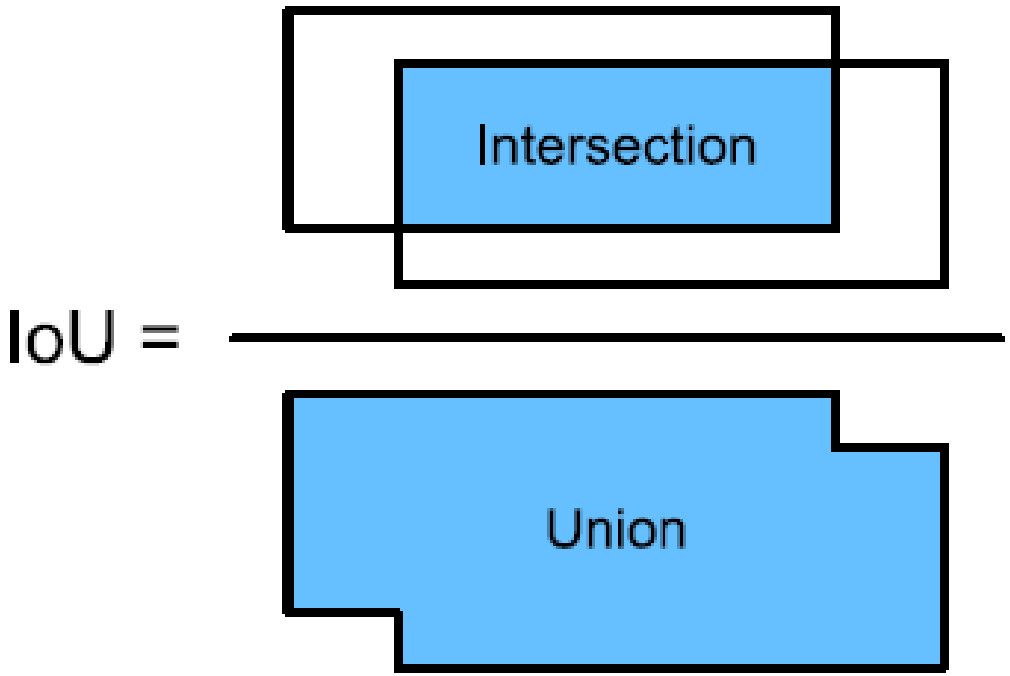
- Reduced number of anchor boxes: $wh(n + m - 1)$

# Intersection over Union (IoU) Metric



- To measure how well a bounding box predict the ground truth bounding box, we use the IoU metric
- The IoU metric is also know as the Jaccard Index

$$J(\mathcal{A}, \mathcal{B}) = \frac{\left| \mathcal{A} \cap \mathcal{B} \right|}{\left| \mathcal{A} \cup \mathcal{B} \right|}$$



# Notebooks

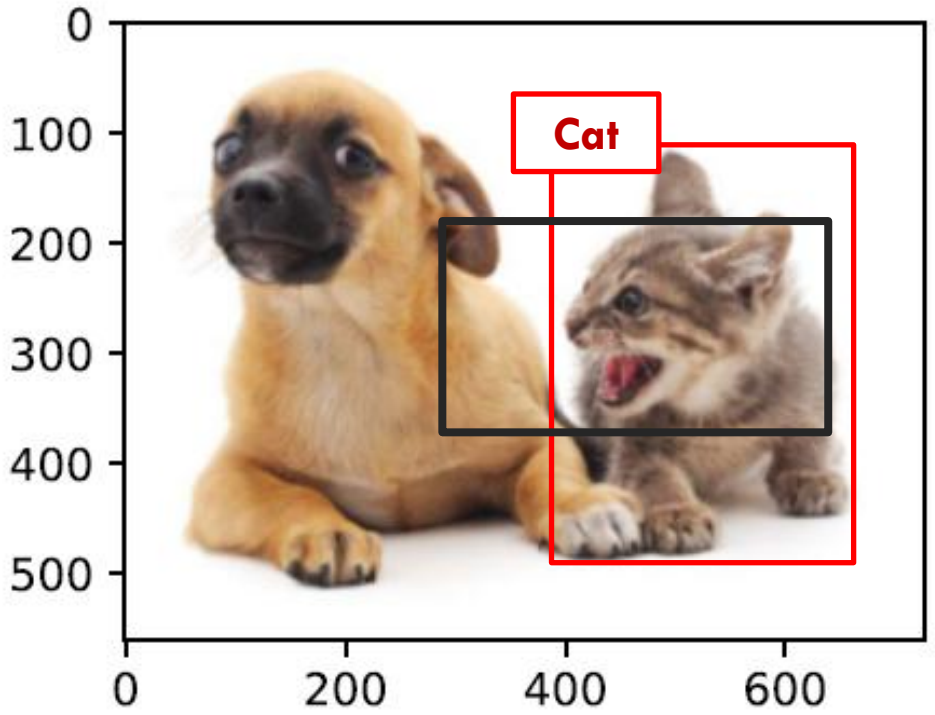


- chapter_computer-vision/bounding-box.ipynb
- chapter_computer-vision/anchor.ipynb

# Labeling Anchor Boxes in Training Data



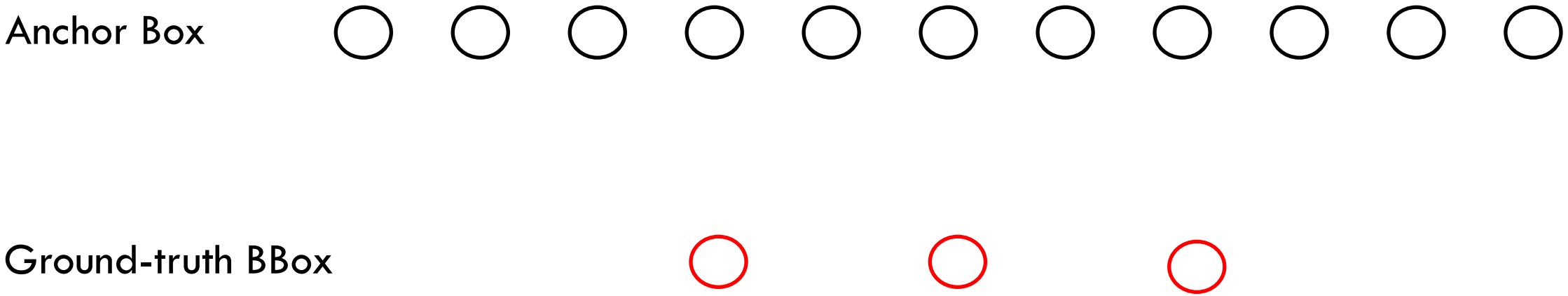
- We want to use anchor boxes for training
- For each anchor box we weed to generate class and offset labels



# Anchor Box Labeling Procedure



- Let anchor boxes be $A_1, \ldots, A_{n_a}$
- Let ground-truth bounding boxes be $B_1, \ldots, B_{n_b}$, where $n_a \geq n_b$
- Let $X \in \mathbb{R}^{n_a \times n_b}$, where $x_{ij} = \mathrm{IoU}(A_i, B_j)$

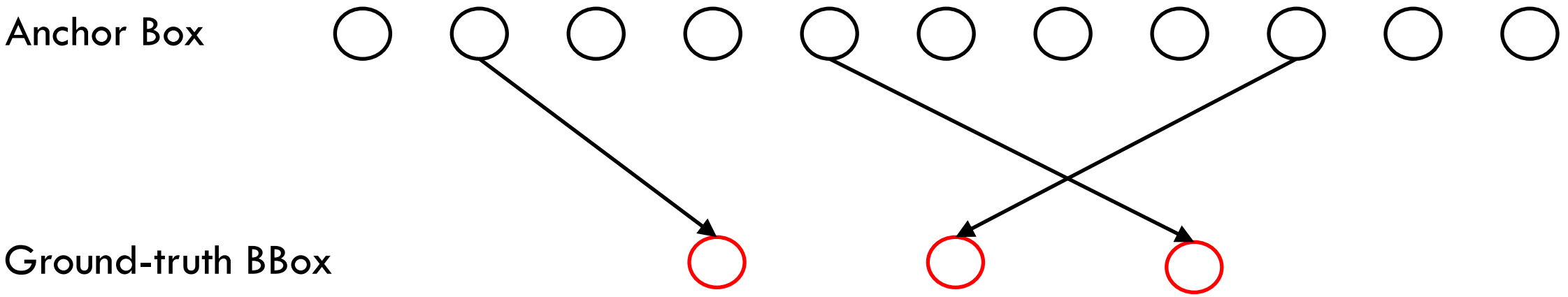


# Anchor Box Labeling Procedure



Step 1

- Find maximal matching
- Assign matched anchor box with the class of the corresponding ground-truth bounding box

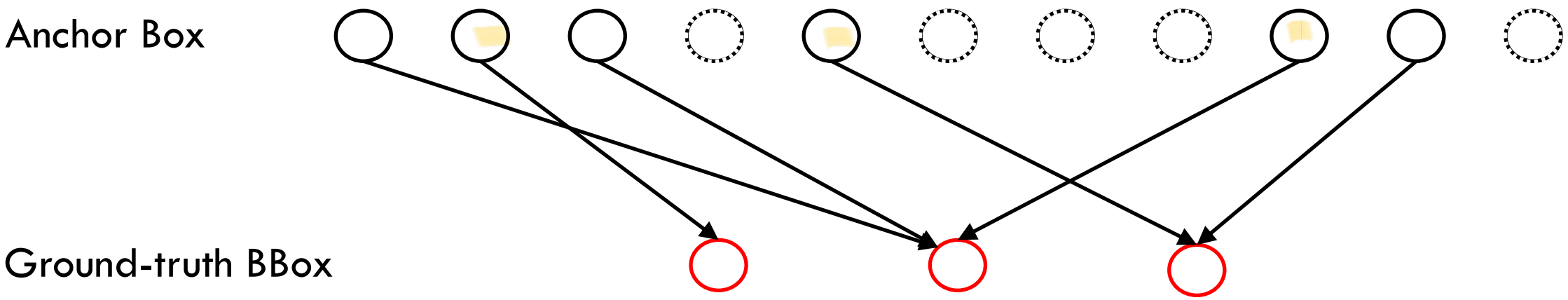


# Anchor Box Labeling Procedure



Step 2

- Assign the unmatched anchor boxes $(n_a - n_b)$ to its closes ground truth bounding box, only if the IoU is above a predefined threshold
- Remaining anchor boxes are labeled “background” or “negative”



# Anchor Box Labeling Procedure



Step 3

- For "positive" (non-background) anchor boxes compute offsets to each of their ground-truth bounding boxes
- Offset should incorporate differences in x, y, width and height
- Let central coordinates of A and B be $(x_a, y_a)$ and $(x_b, y_b)$, widths be $w_a, w_b$ and heights be $h_a, h_b$. The offset is:

$$\left( \frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right),$$

- Where $\mu_x = \mu_y = \mu_w = \mu_h = 0$, $\sigma_x = \sigma_y = 0.1$ and $\sigma_w = \sigma_h = 0.2$

# Non-Maximum Suppression

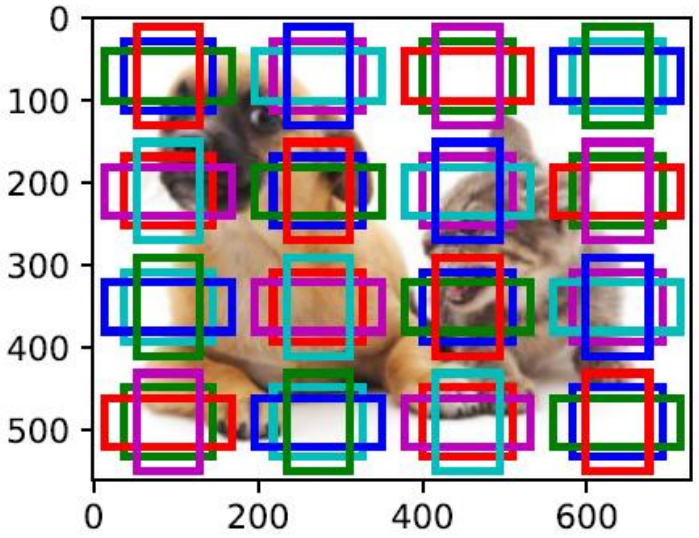


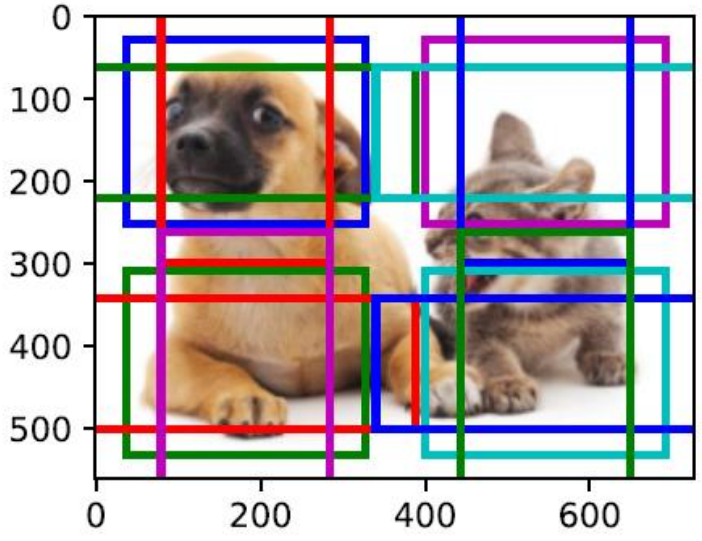
- During prediction there may be many overlapping anchor boxes predicting the same object
- Non-maximum suppression select the best proposed bounding box among similar proposed bounding boxes
- Procedure
  1. Create a sorted list L of predicted bounding boxes based on their confidence scores (e.g., probability output of the model)
  2. Select the highest confidence element in L as a bounding boxes
  3. Remove all elements in L that are similar to this bounding boxes
  4. Repeat until the list L is empty

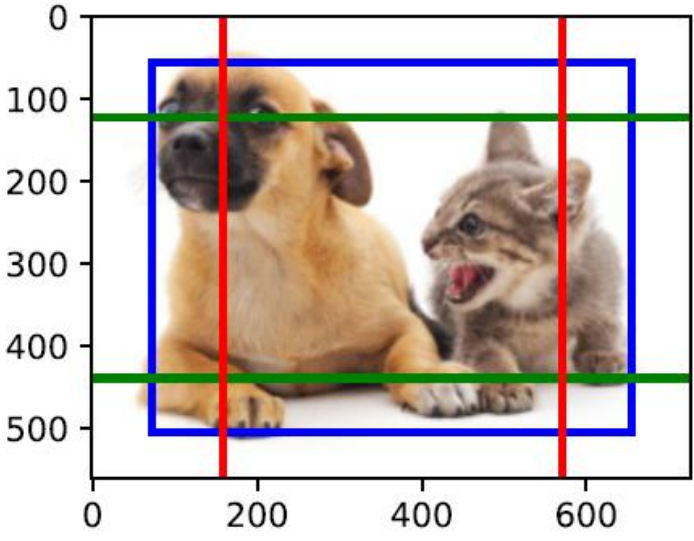
# Multiscale Anchor Boxes



- Compute complexity of creating anchor boxes for each pixel is high
- Instead define anchor boxes at various scales (at the same scale the size $wh$ is the same but the width-height ratio is different)
- Then sample the pixels uniformly with these anchor boxes, i.e., require more boxes for smaller boxes



# Multiscale Detection

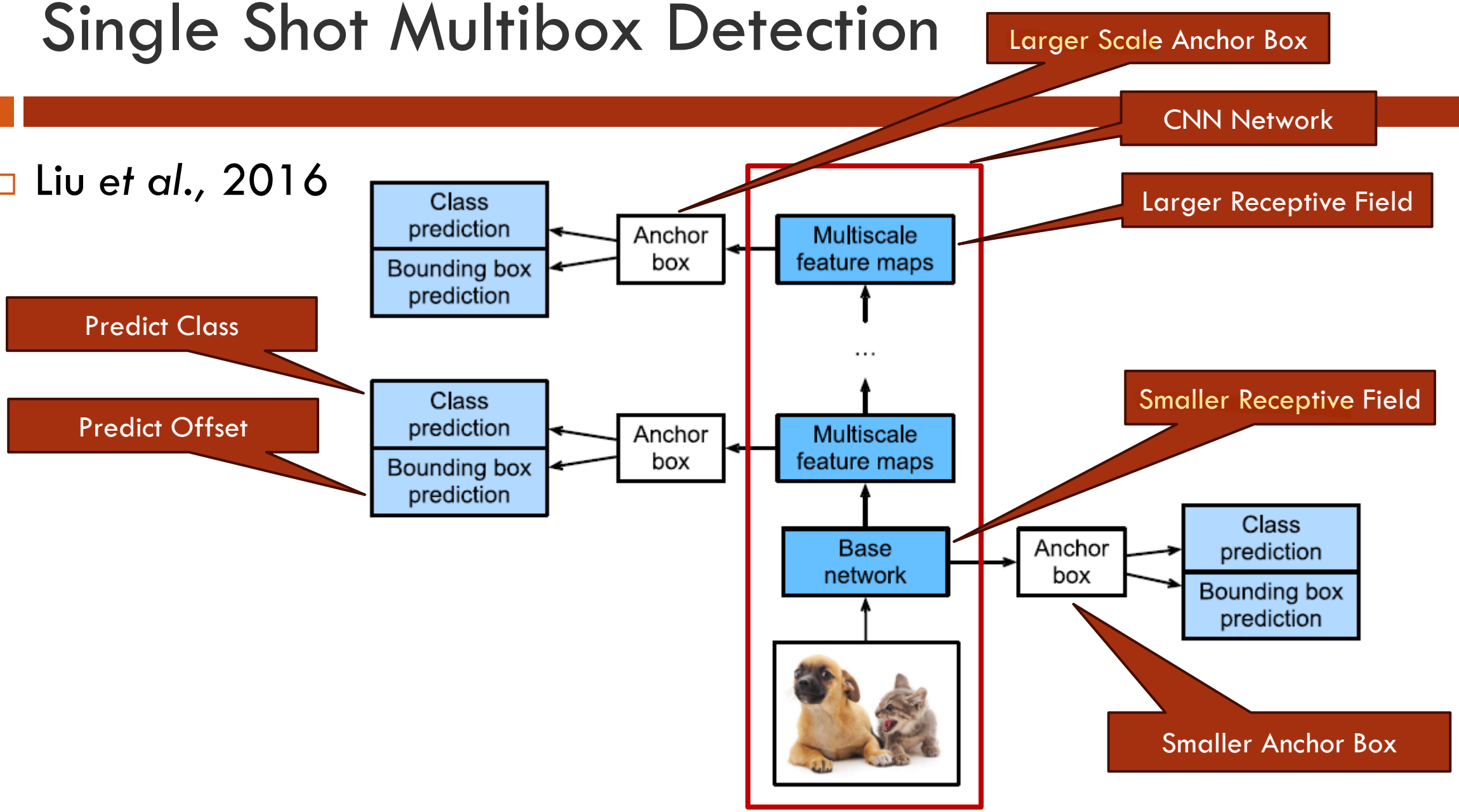


- Use CNN-based network to implement multiscale detection
- Recall nodes in higher layers of CNN network have greater receptive fields, i.e., cover more pixels in the original image
- Use multiple layers of the CNN network for multiscale detection
- Use the features of each node to generate class and offset predictions

# Single Shot Multibox Detection

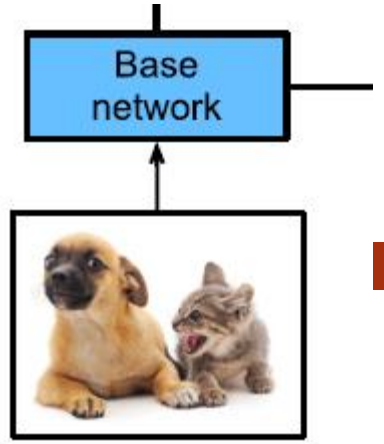


- Liu *et al.*, 2016



# Base Network Block






```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i+1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

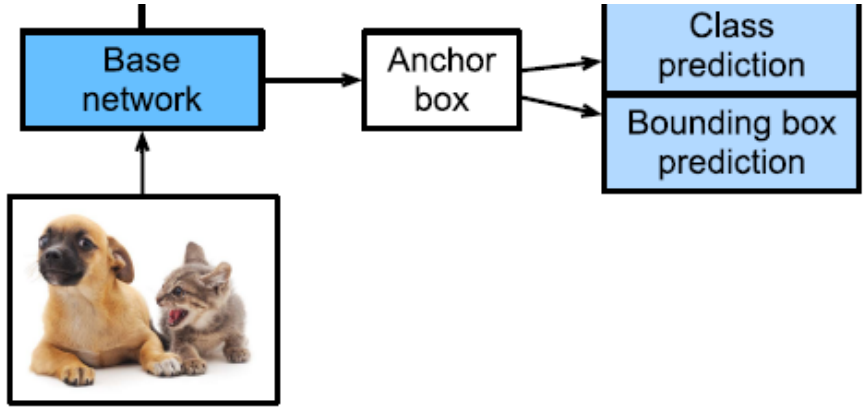
```
torch.Size([2, 64, 32, 32])
```

Similar to VGG
blocks of 3x3 Conv

```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(nn.Conv2d(in_channels, out_channels,
                             kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

# Class and Bbox Predictor






```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

Generate class prediction for each anchor box

```
def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)
```

Generate bbox prediction for each anchor box

```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

# Single Shot Multibox Detection : SSD



Global Max Pooling to reduce height and width to 1

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1,1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
        for i in range(5):
            # Equivalent to the assignment statement `self.blk_i = get_blk(i)`
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(idx_to_in_channels[i],
                                                    num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(idx_to_in_channels[i],
                                                      num_anchors))
```

# Single Shot Multibox Detection



```
def forward(self, X):
    anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
    for i in range(5):
        # Here `getattr(self, 'blk_%d' % i)` accesses `self.blk_i`
        X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
            X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
            getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
    anchors = torch.cat(anchors, dim=1)
    cls_preds = concat_preds(cls_preds)
    cls_preds = cls_preds.reshape(
        cls_preds.shape[0], -1, self.num_classes + 1)
    bbox_preds = concat_preds(bbox_preds)
    return anchors, cls_preds, bbox_preds
```

Returns anchors with corresponding class and offsets

# Loss and Evaluation Functions



Class: Cross entropy loss
BBox: L1 loss

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                   cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

Class: Accuracy
BBox: MaxAbsErr

```
def cls_eval(cls_preds, cls_labels):
    # Because the class prediction results are on the final dimension,
    # `argmax` needs to specify this dimension
    return float((cls_preds.argmax(dim=-1).type(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

# Notebook/PDF



- chapter_computer-vision/object-detection-dataset.ipynb
- chapter_computer-vision/ssd.ipynb

- Note: YOLO (You Only Look Once) is another popular object detection algorithm using similar approach as SSD. It is fast enough for real time video object detection.

# Region-based CNNs (R-CNNs)

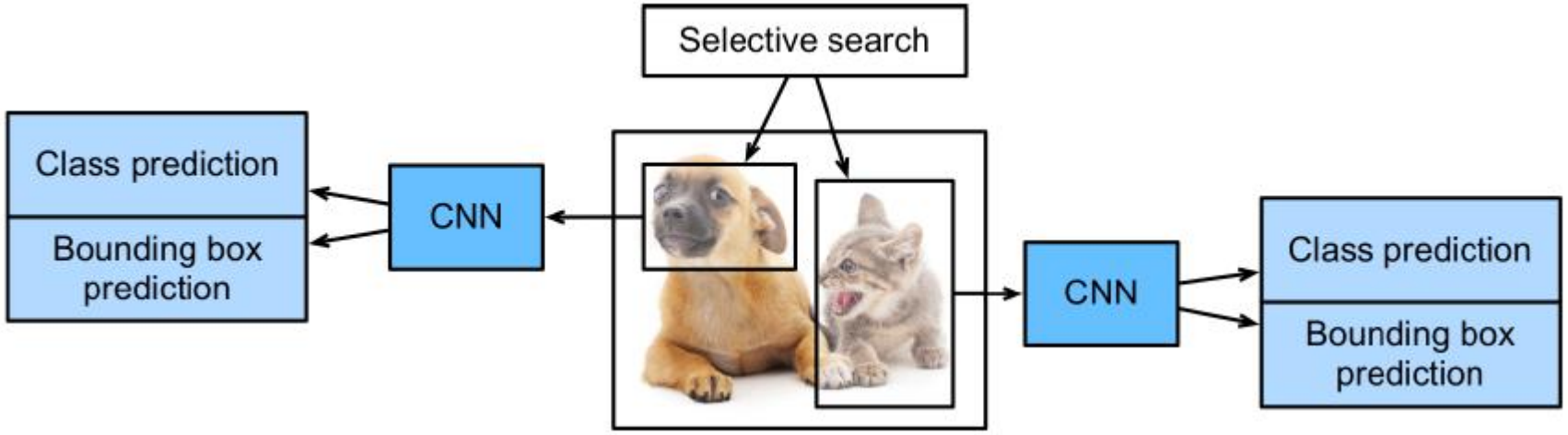


- Like Single Shot Multibox Detection, Region-based CNNs is another approach to object detection
- Several R-CNNs were introduced
  - R-CNN (Girshick *et al.*, 2014)
  - Fast R-CNN (Girshick, 2015)
  - Faster R-CNN (Ren *et al.*, 2015)
  - Masked CNN (He *et al.*, 2017)

# R-CNN



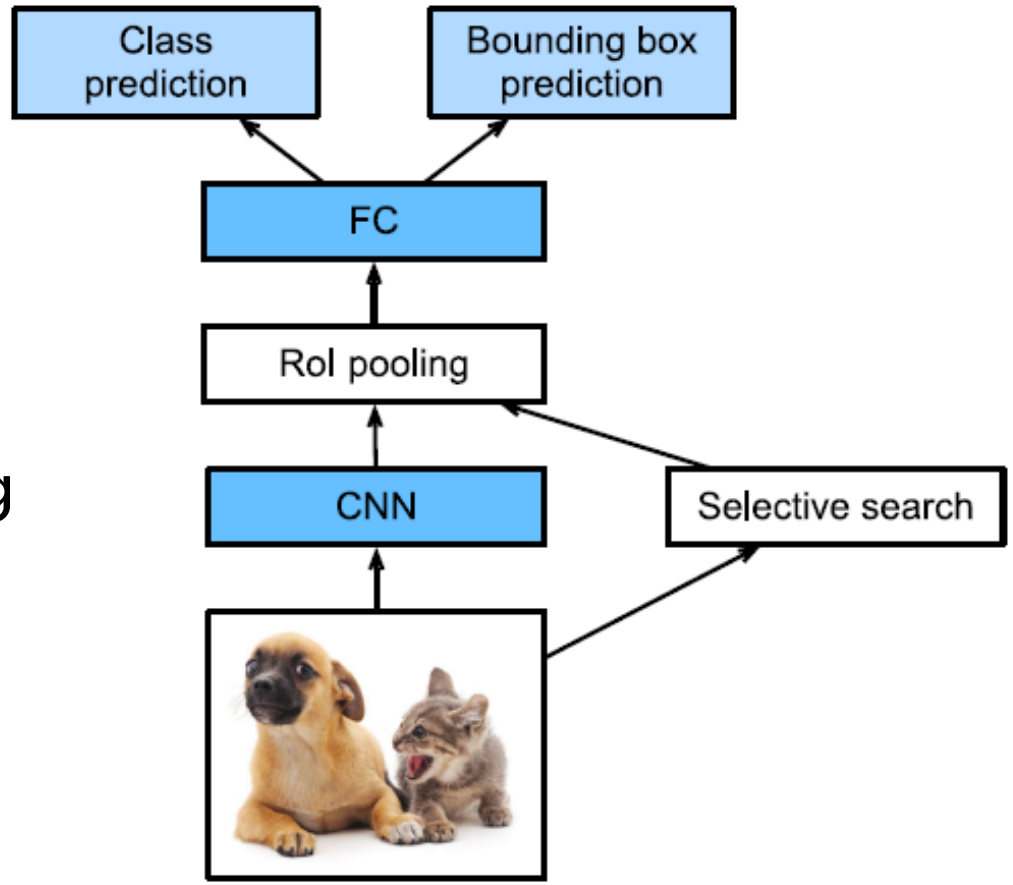
- Use *selective search*, a separate greedy based algorithm, to generate region proposals. Label region proposal with class and ground-truth bounding box
- For each region proposal perform a forward pass on a pretrained CNN to generate features
- Train SVMs to classify region proposals. One SVM per class.
- Train linear regression model to predict ground-truth bounding box given CNN features and labeled bounding box



# Fast R-CNN



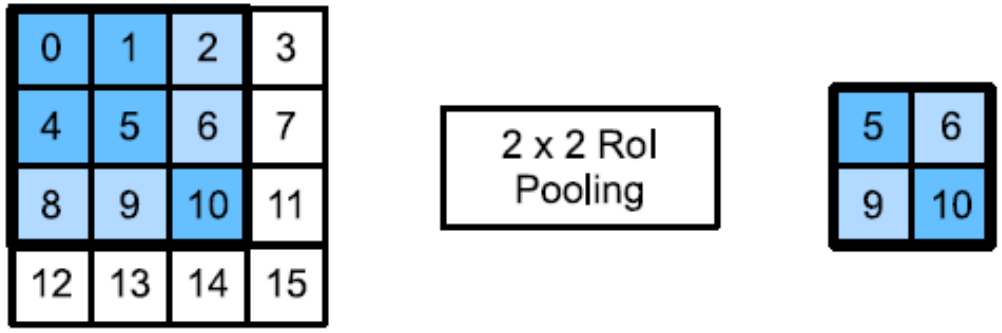
- R-CNN has to perform a forward pass for each region proposal
- These proposals often overlap, but no computation is shared
- Fast R-CNN performs CNN forward on the entire image
- Introduced region of interest (RoI) pooling layer to extract region proposal features
- Network directly outputs class and bbox prediction



# Region of Interest Pooling



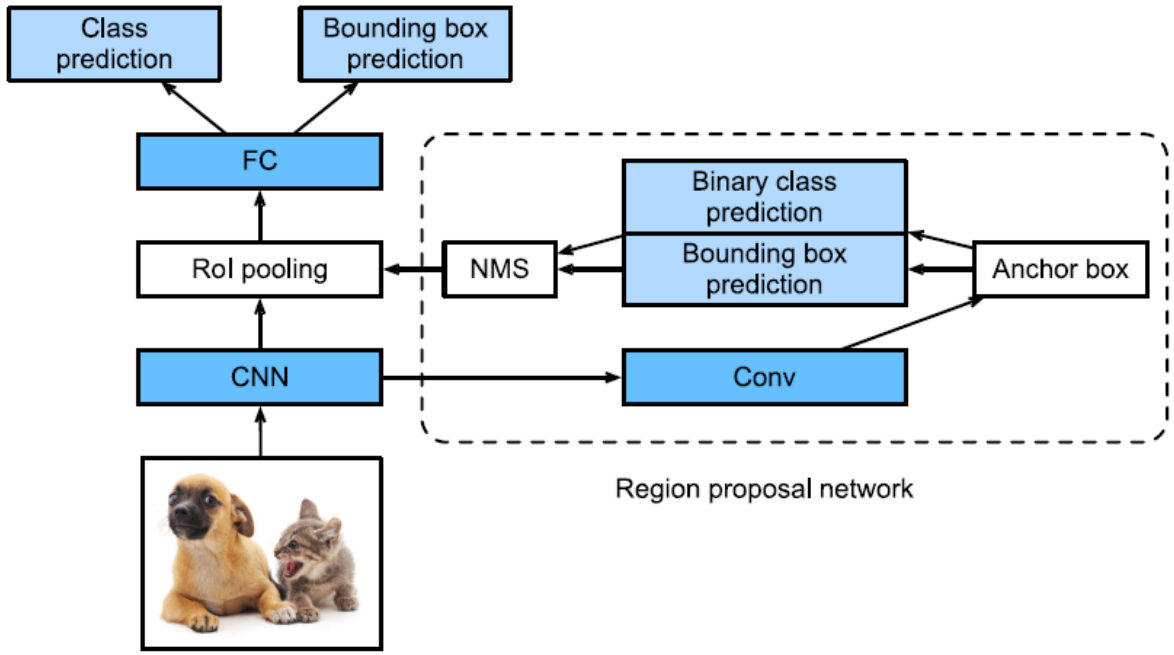
- Directly specify the size of the output, instead of the pooling parameters
- RoI pooling finds the appropriate max pooling parameter to generate desired output shape
- For example,
  - given input shape of 3x3 and desired output shape of 2x2
  - RoI performs a 2x2 max pooling



# Faster R-CNN



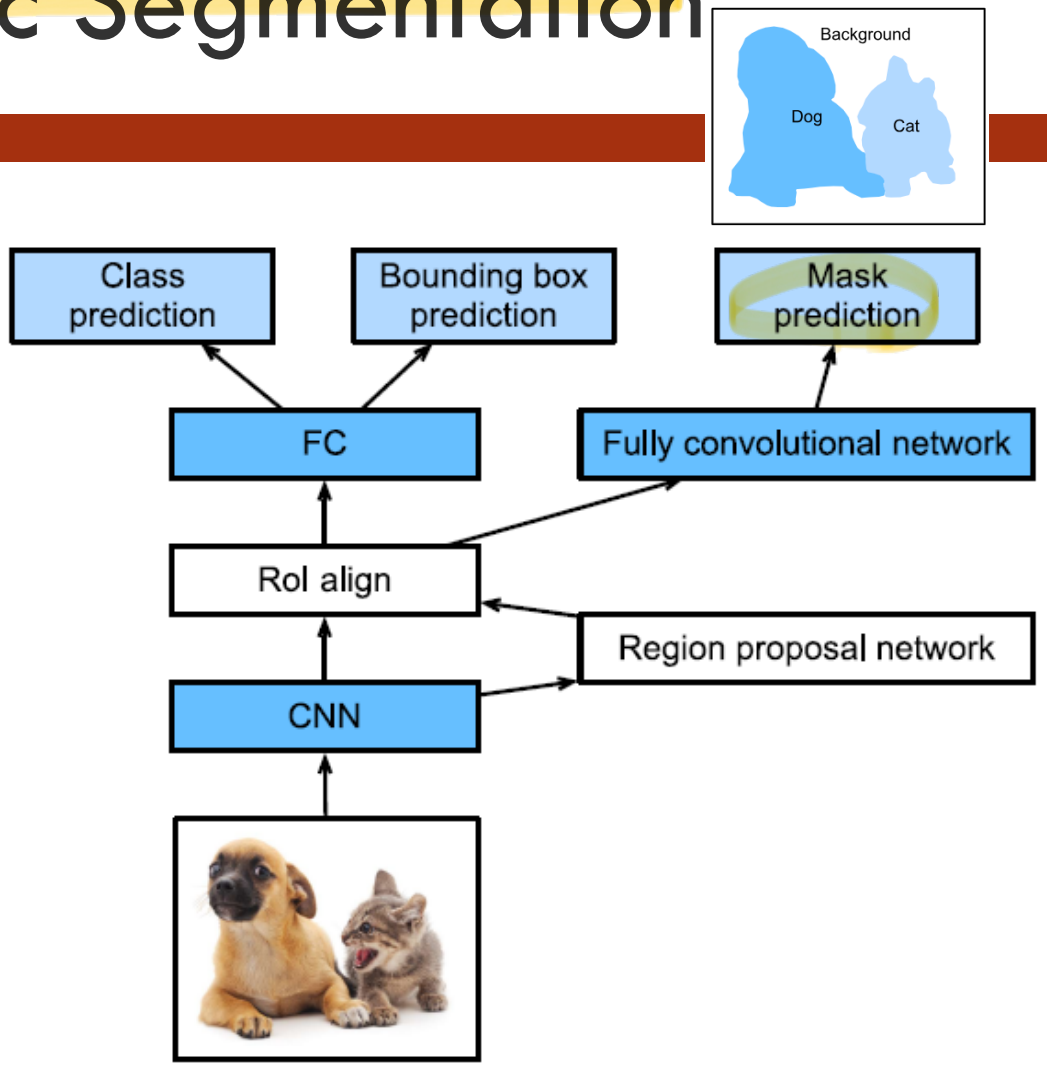
- Replaces selective search with region proposal network
- Region proposal network is similar to SSD, except only applied to last CNN layer
- End-to-end training, region proposal network is jointly trained with the rest of the network



# Mask R-CNN for Semantic Segmentation



- Ground truth is at the pixel-level, not bounding box-level
- RoI align layers using bilinear interpolation, instead of max pooling, to preserve spatial information
- Adds fully convolutional network to generate pixel-level predictions



# Notebook



- chapter_computer-vision/semantic-segmentation-and-dataset.ipynb

# Transposed Convolution

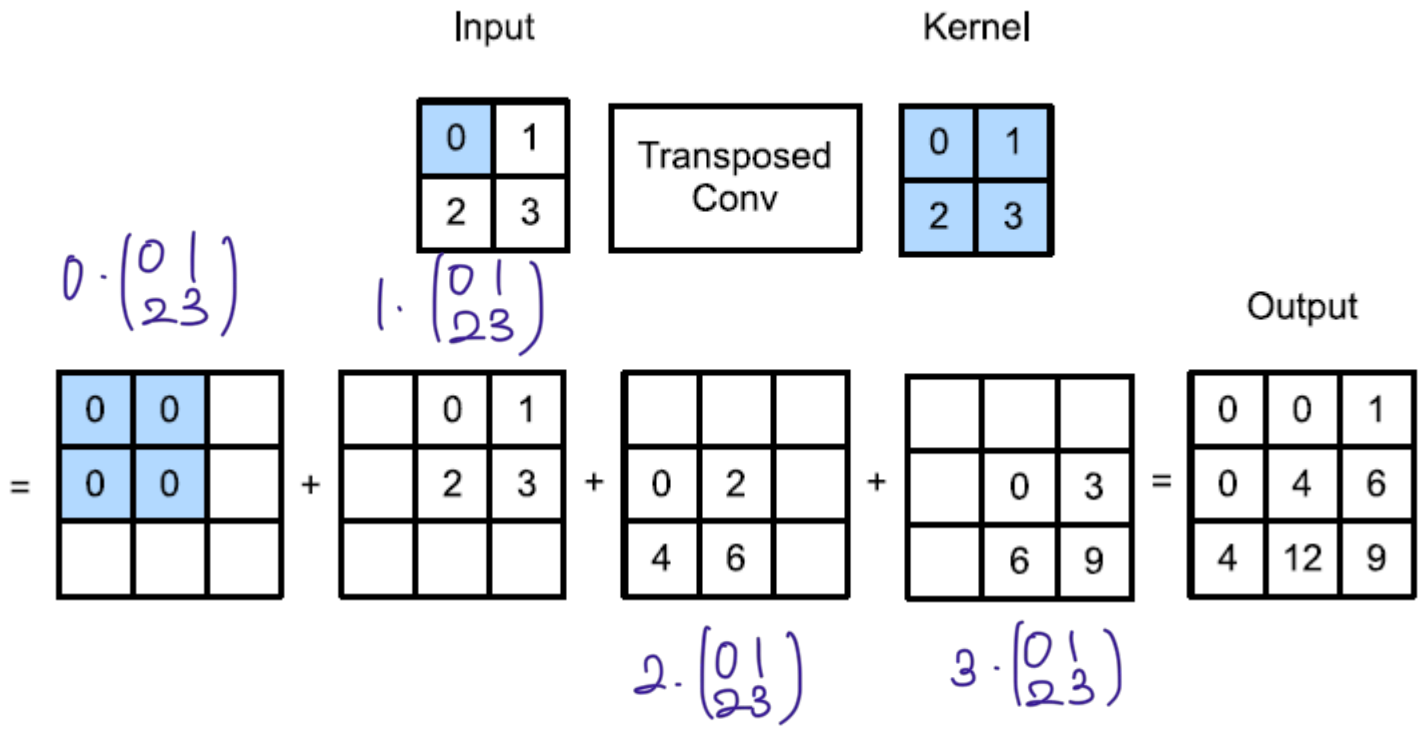


- The CNN networks we have seen either reduce (downsample) the spatial dimensions, or keep them unchanged
- For semantic segmentation, we want the output dimensions to be the same as the input dimensions
- Transposed convolution reverses downsampling operation performed by a regular convolution layer

# Transposed Convolution Basic Operation



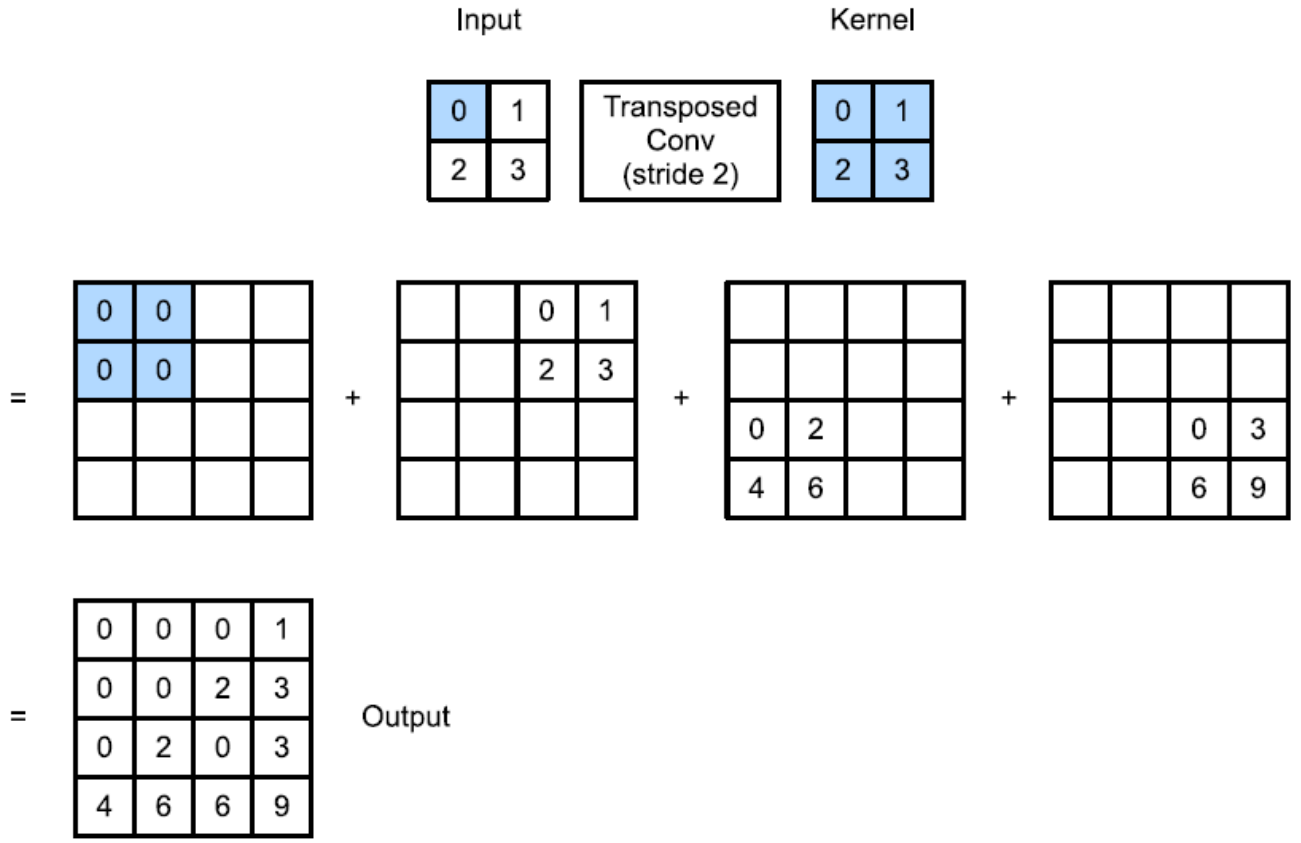
- Given $n_h \times n_w$ input tensor and $k_h \times k_w$ kernel generates $(n_h+k_h-1) \times (n_w+k_w-1)$ output sensor



# Transposed Convolution With Stride



- Given $n_h \times n_w$ input tensor, $k_h \times k_w$ kernel and $s_h, s_w$ stride generate $((n_h-1)s_h + k_h) \times ((n_w-1)s_w) + k_w)$ output sensor



# Convolution as Matrix Multiplication



Convolution compute → matrix multiplication

Consider input matrix $X$ and kernel $K$

$$X = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix} \rightarrow \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 \end{bmatrix}^T$$

nx1

$$K = \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix} \rightarrow \begin{bmatrix} k_1 & k_2 & 0 & k_3 & k_4 & 0 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & 0 & k_3 & k_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & k_1 & k_2 & 0 & k_3 & k_4 & 0 \\ 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & k_3 & k_4 \end{bmatrix}$$

kxn

$$Y = \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix} \rightarrow \begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix}^T \qquad \hat{Y} = \hat{K}\hat{X}$$

nx1 kxn nx1

# Transpose Convolution as Matrix Multiply



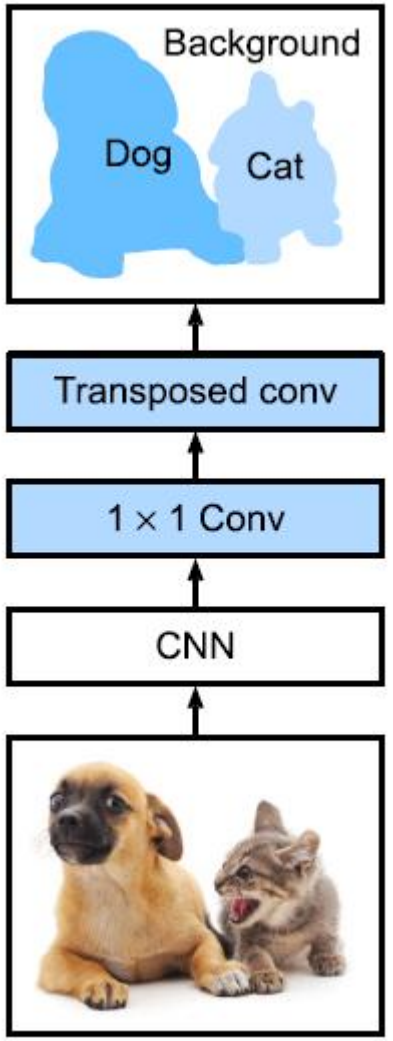
- It turns out transpose convolution can be implemented as matrix multiplication as well
- If $Z = \text{trans\_conv}(Y, K)$, then $\hat{Z} = \hat{K}^T \hat{Y}$

- See notebook:
  - chapter_computer-vision/transposed-conv.ipynb

# Fully Convolutional Networks



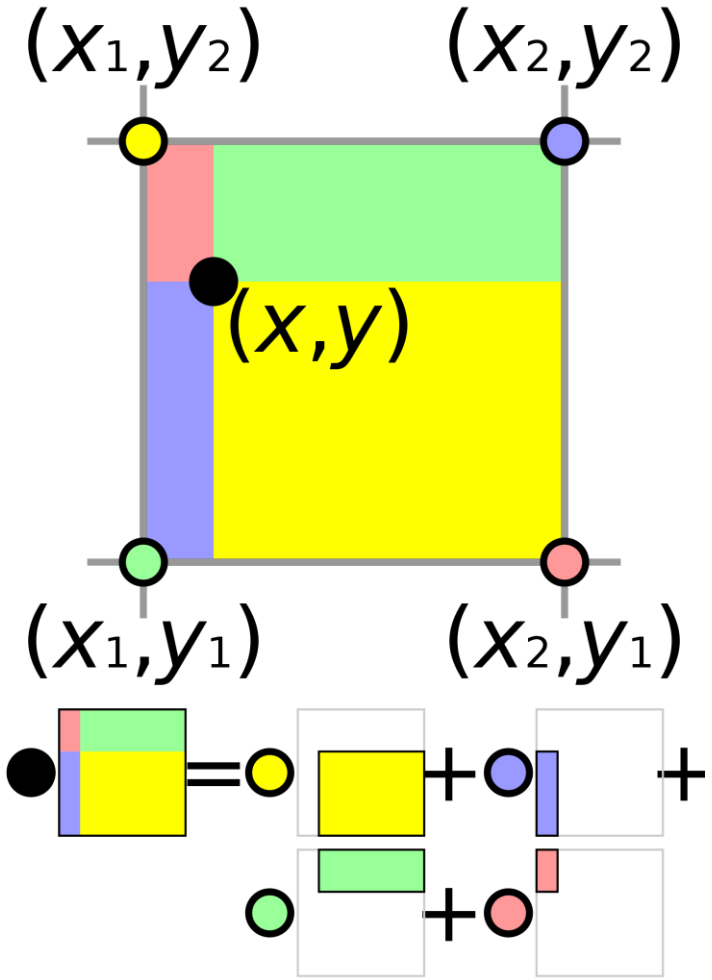
- A fully convolution network uses convolutional layers and transpose convolutional layers to transform image pixels to pixel classes
- Each pixel is given a class label
- Size of the last layer is the same size as the input
- Channel dimension of the last layer corresponds to the number of classes
- The 1x1 Conv layer transforms the number of channels to the number of classes



# Bilinear Interpolation



- During the upsampling process, we want compute the value at pixel location $(x, y)$ of the output image
- This pixel location is not defined in the input image
- Bilinear linear interpolation compute the value at $(x, y)$
  using the closest 4 points in the input image
  $(x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2)$




https://en.wikipedia.org/wiki/Bilinear_interpolation

# Initializing Transpose Convolution



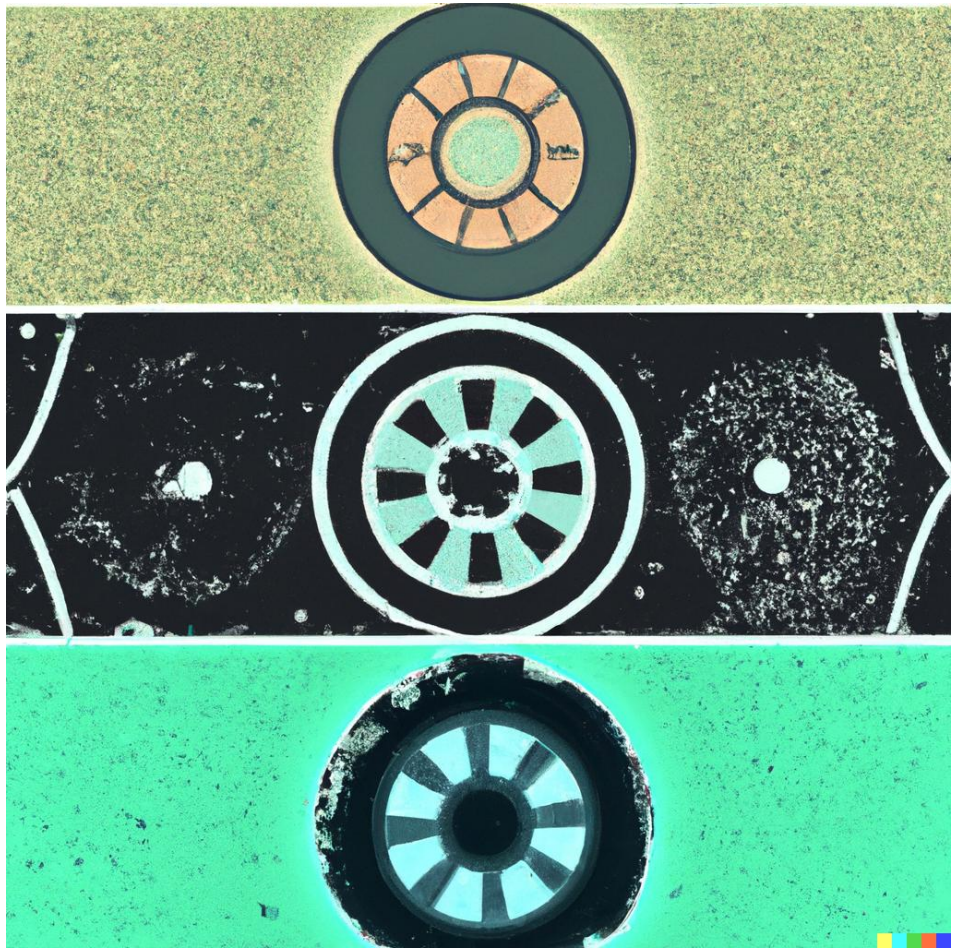
- During the training the process, the network is supposed to learn the best values for the transpose kernel
- We help the training process by initializing the transpose kernel with values from bilinear interpolation

- Notebook
  - chapter_computer-vision/fcn.ipynb



# Style Transfer

DALL-E generated image for "style transfer"



# Neural Style Transfer

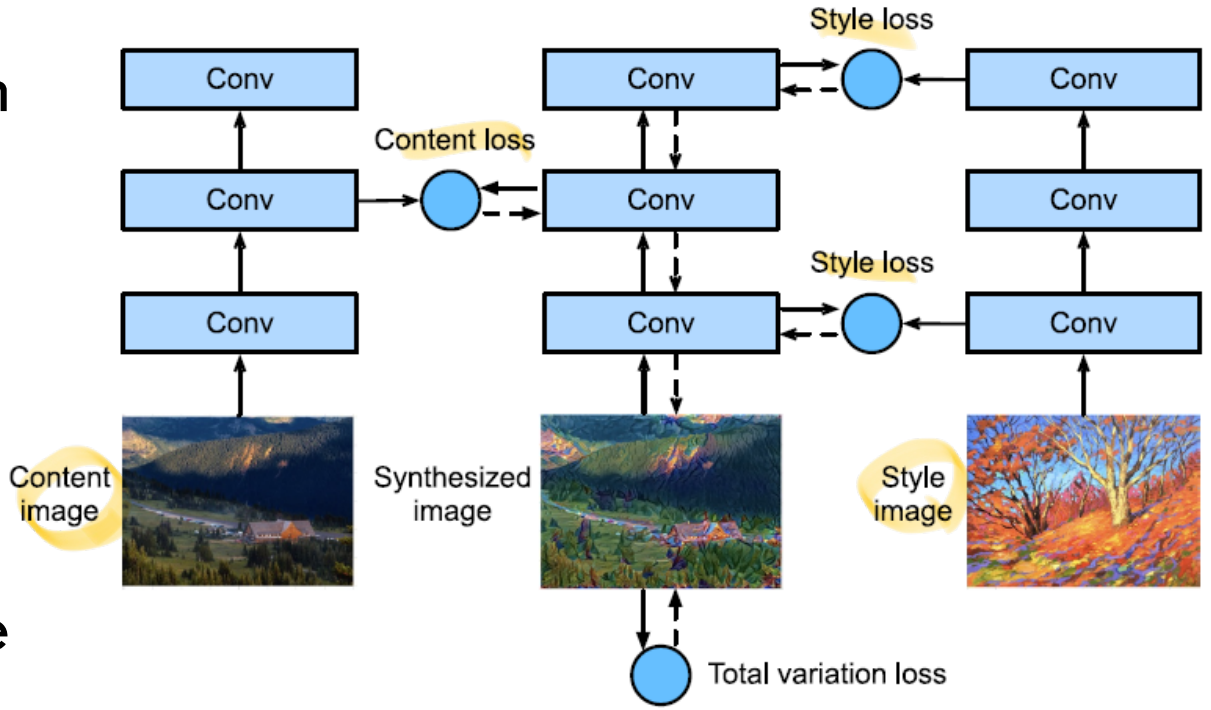


- Given two images: a content image and a style image
- Generate a synthesized image that transfer the style of style image to the content image
- Gatys *et al.*, 2016

# Approach



- Choose a pretrained network. Fix its parameters.
- Use network to extract features from content and style images
- Select some layer outputs as style outputs and some as content outputs
- Compute loss: content loss, style loss and total variation loss
- Train by updating synthesized image to minimize loss



# Content Loss



- Content loss is mean square loss over all individual position-level (pixel) features

```
def content_loss(Y_hat, Y):
    # We detach the target content from the tree used to dynamically compute
    # the gradient: this is a stated value, not a variable. Otherwise the loss
    # will throw an error.
    return torch.square(Y_hat - Y.detach()).mean()
```

# Style Loss



- Use Gram matrix to represent the style of a model layer
- Let the layer output be c channels, h height and w width
- Let $X \in \mathbb{R}^{c \times wh}$, i.e. one row of $X$ represents a channel
- Gram matrix is $XX^T \in \mathbb{R}^{c \times c}$, i.e. correlation of styles between channels
- Style loss is mean square loss between Gram matrices

```
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

# Total Variation Loss



- Synthesize image sometimes has high-frequency noise, i.e., large variations between adjacent pixels
- Total variation denoising by penalize against such large variations

$$\sum_{i,j} \left|x_{i,j} - x_{i+1,j}\right| + \left|x_{i,j} - x_{i,j+1}\right|$$

```
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

# Synthesized Image



- The only parameters in the model are the parameters to define the synthesized image

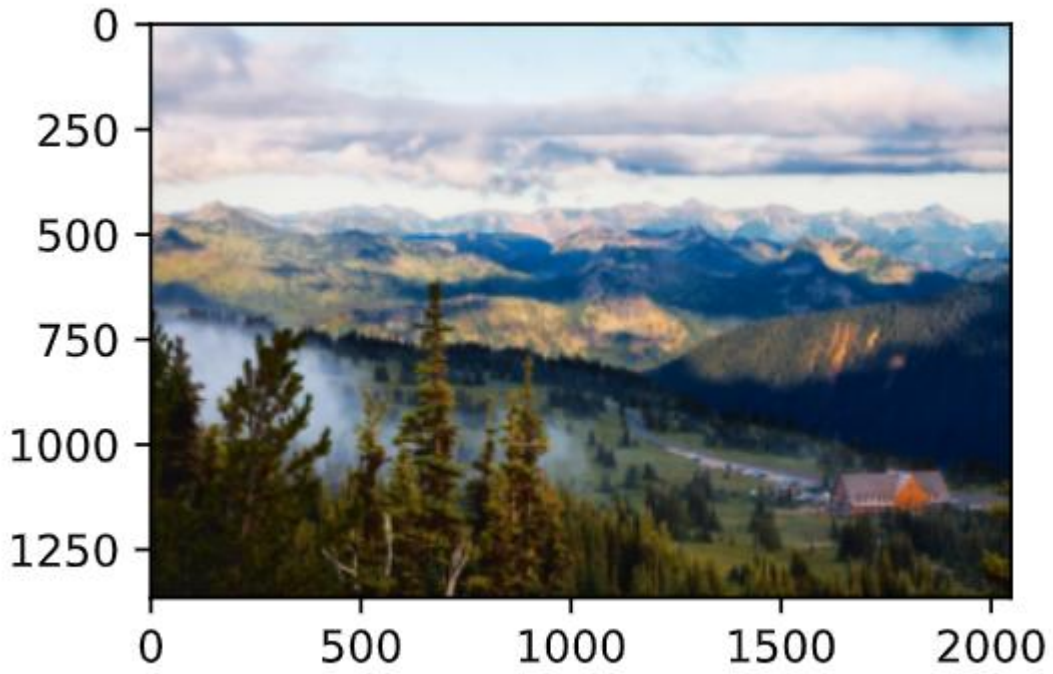
```
class SynthesizedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
        super(SynthesizedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

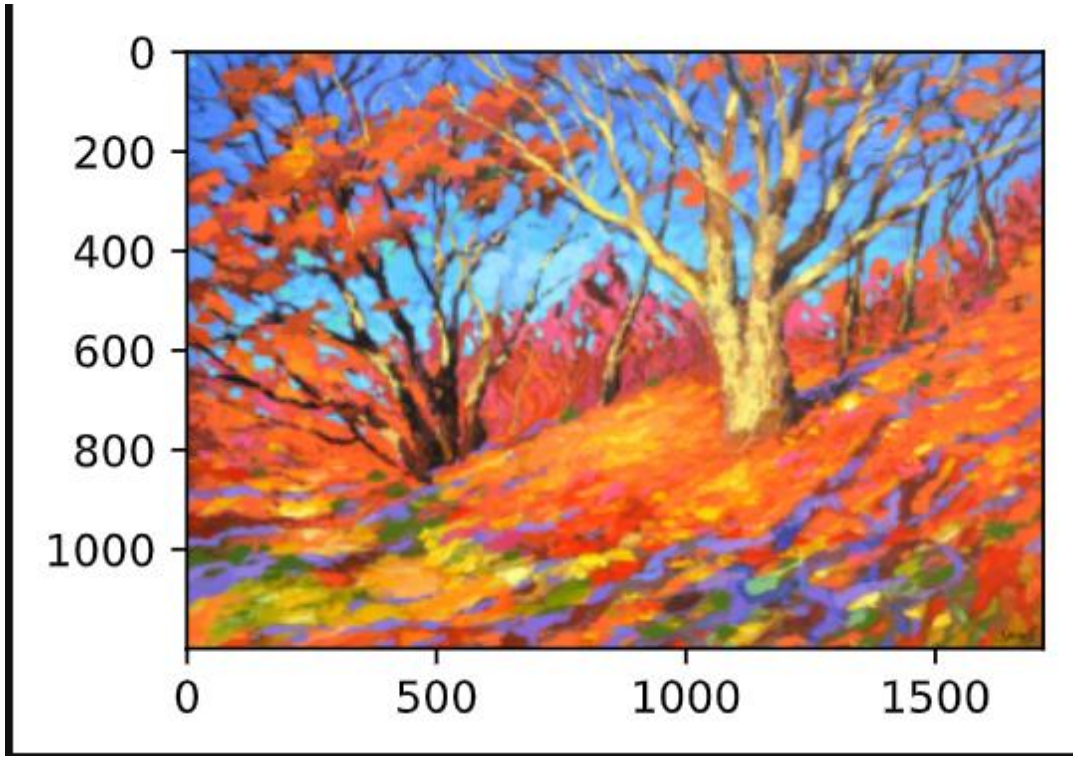
# Training Example

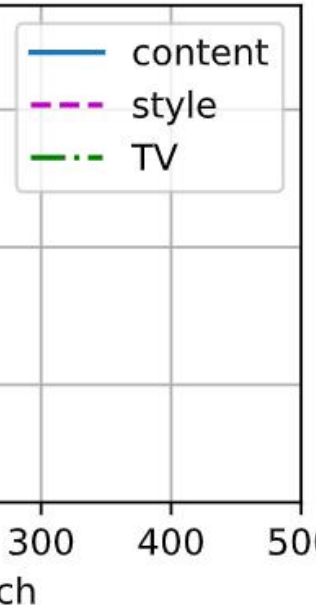


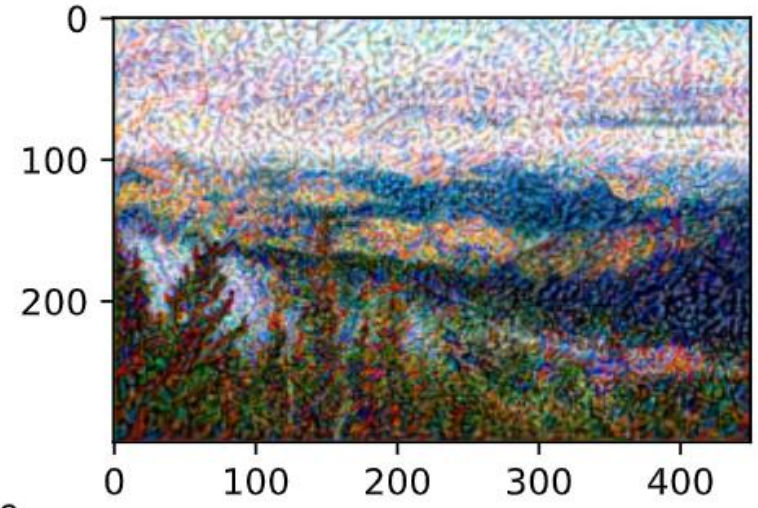
**Content**

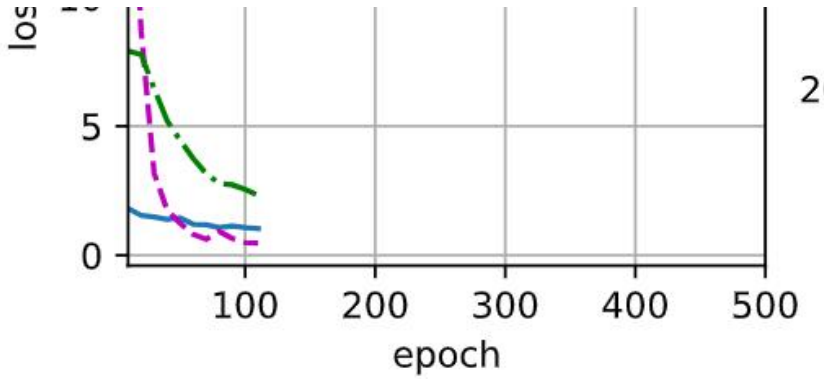


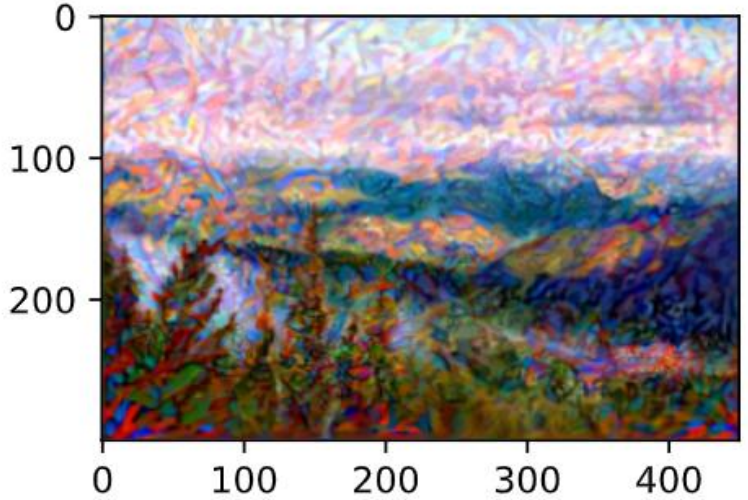

**Style**

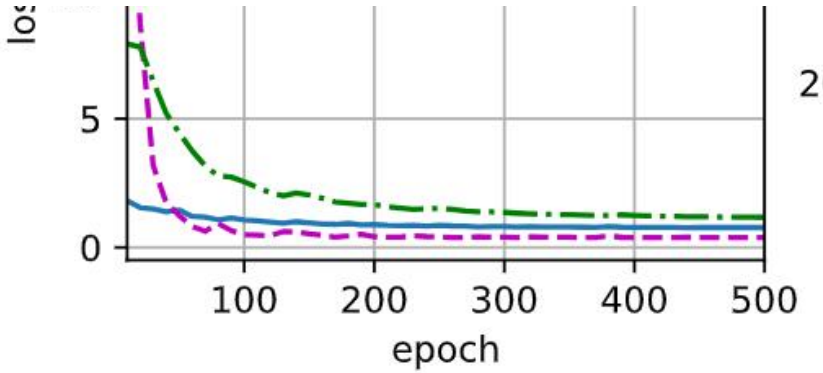


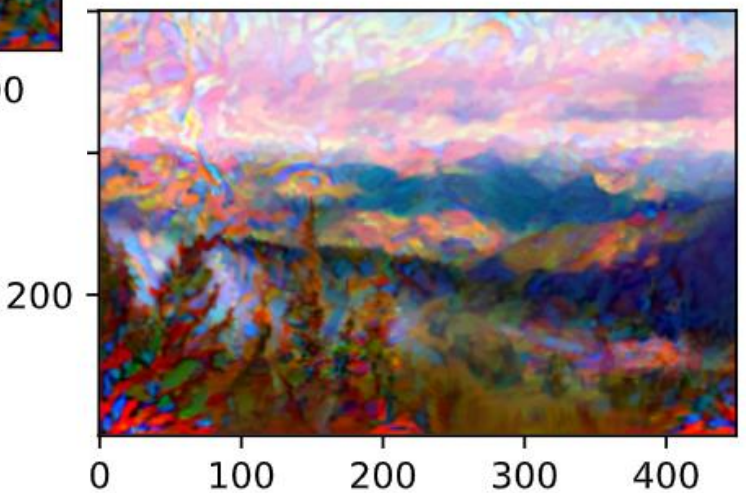

content
style
TV
300
400
500
ch


0
100
200
0
100
200
300
400


los
5
0
100
200
300
400
500
epoch


0
100
200
0
100
200
300
400

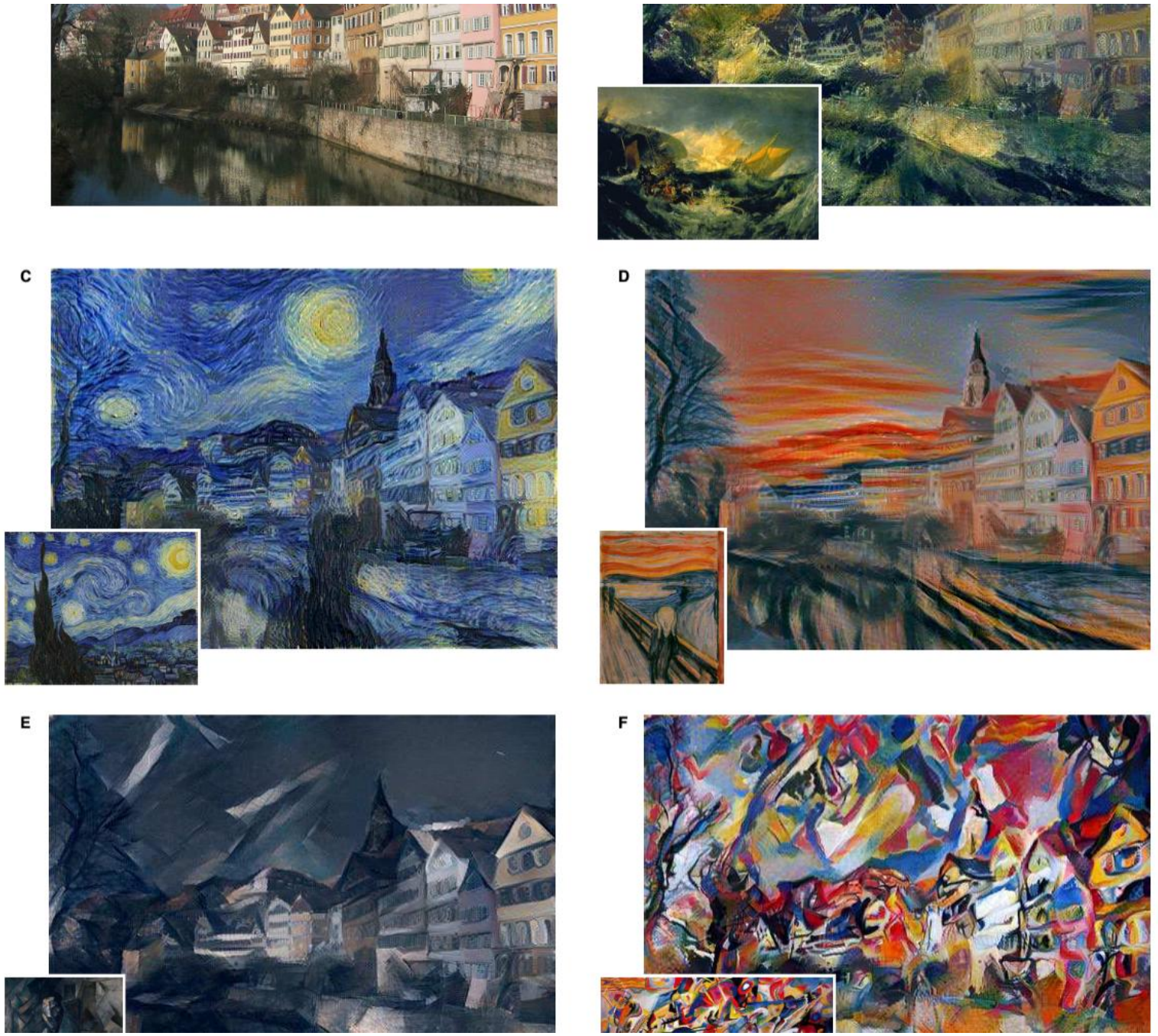

los
5
0
100
200
300
400
500
epoch


200
0
100
200
300
400

# Examples



- Getys *et al.*, 2016



# Tradeoff



- Content vs Style

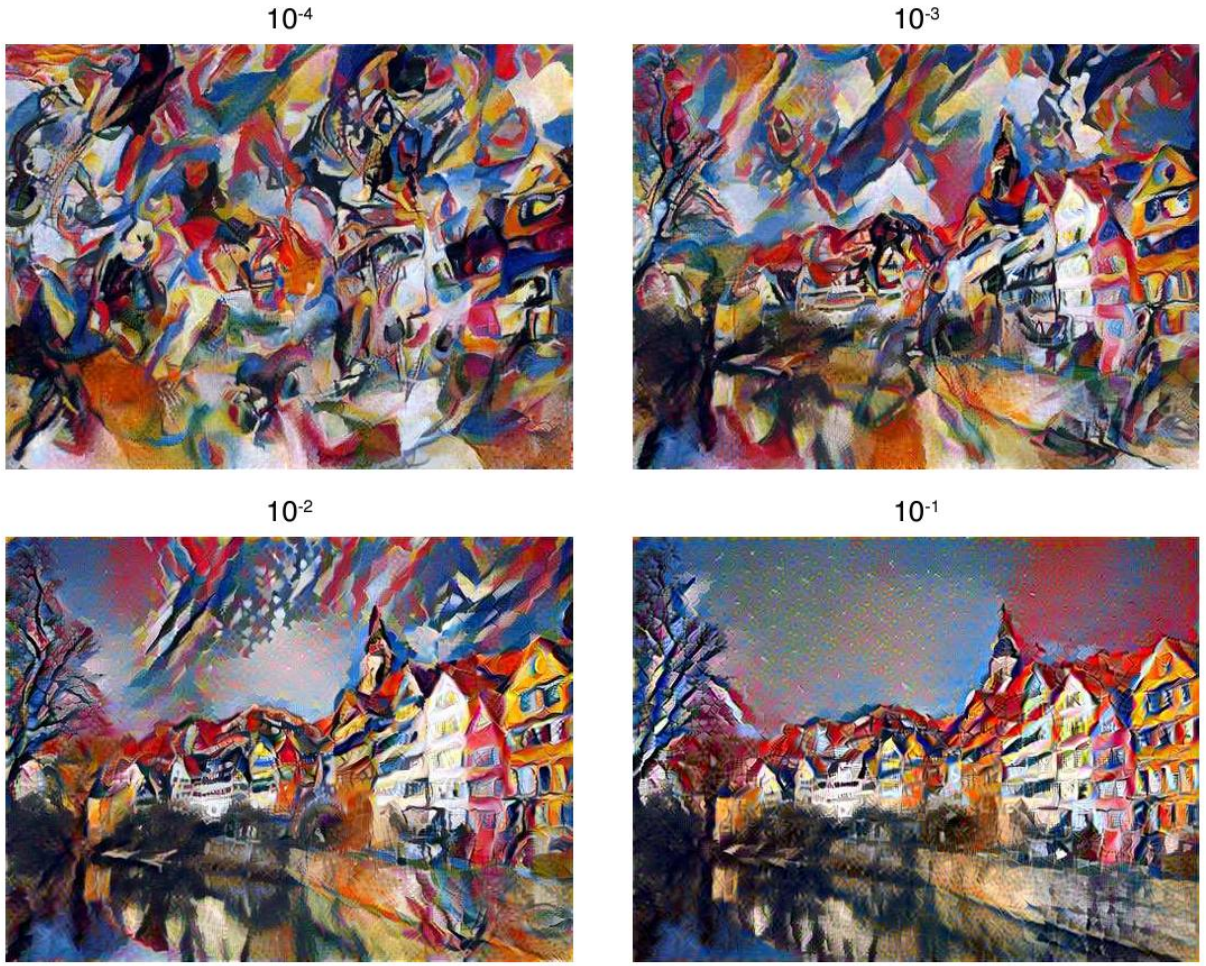



Figure 4. Relative weighting of matching content and style of the respective source images. The ratio $\alpha/\beta$ between matching the content and matching the style increases from top left to bottom right. A high emphasis on the style effectively produces a texturised version of the style image (top left). A high emphasis on the content produces an image with only little stylisation (bottom right). In practice one can smoothly interpolate between the two extremes.

# Choosing Layers



- Style layers
  - Choose range of layers from low to high for visual smoothness
- Content layer
  - Choosing low level layer enforce pixel level details
  - Choosing high level layer enforce concept

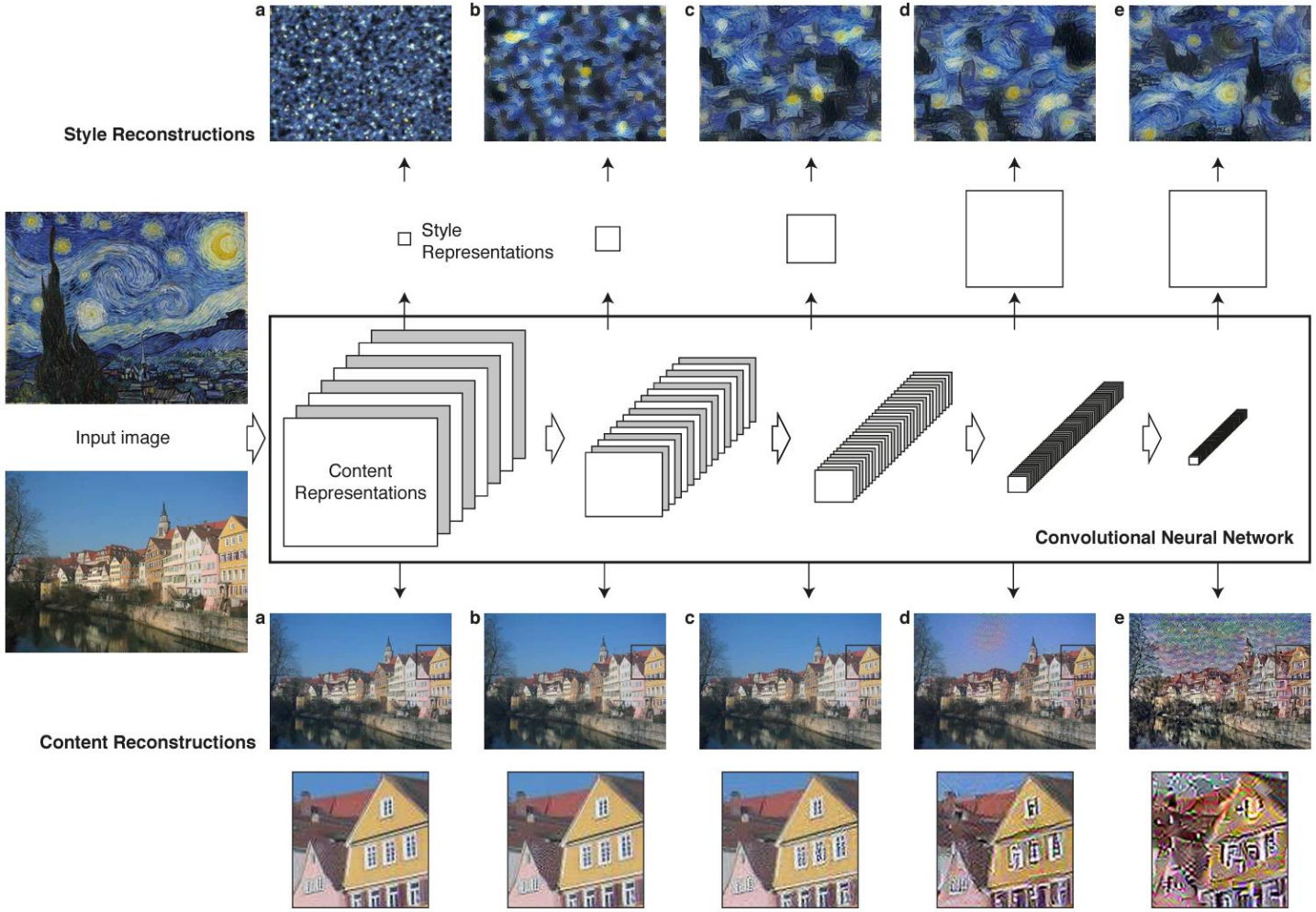



Figure 1. Image representations in a Convolutional Neural Network (CNN). A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network. **Content Reconstructions**. We can visualise the information at different processing stages in the CNN by reconstructing the input image from only knowing the network's responses in a particular layer. We reconstruct the input image from from layers 'conv1_2' (a), 'conv2_2' (b), 'conv3_2' (c), 'conv4_2' (d) and 'conv5_2' (e) of the original VGG-Network. We find that reconstruction from lower layers is almost perfect (a–c). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (d,e). **Style Reconstructions**. On top of the original CNN activations we use a feature space that captures the texture information of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from a style representation built on different subsets of CNN layers ( 'conv1_1' (a), 'conv1_1' and 'conv2_1' (b), 'conv1_1', 'conv2_1' and 'conv3_1' (c), 'conv1_1', 'conv2_1', 'conv3_1' and 'conv4_1' (d), 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1' and 'conv5_1' (e). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

# Content Loss



- Low vs high layer


Figure 5. The effect of matching the content representation in different layers of the network. Matching the content on layer 'conv2_2' preserves much of the fine structure of the original photograph and the synthesised image looks as if the texture of the painting is simply blended over the photograph (middle). When matching the content on layer 'conv4_2' the texture of the painting and the content of the photograph merge together such that the content of the photograph is displayed in the style of the painting (bottom). Both images were generated with the same choice of parameters ($\alpha/\beta = 1 \times 10^{-3}$). The painting that served as the style image is shown in the bottom left corner and is named *Jesuiten III* by Lyonel Feininger, 1915.

# DALL-E: Der Schrei in the style van Gogh

# DALL-E: Variations of “Der Schrei in the style van Gogh“


# DALL-E: Der Schrei painting in the style van Gogh