CHAPTER 6

Advanced Neural Networks

In this chapter, we will look at some of the advanced concepts and models in deep learning being used recently. Image segmentation and object localization and detection are some of the key areas that have garnered a lot of importance lately. Image segmentation plays a crucial role in detecting diseases and abnormalities through the processing of medical images. At the same time, it is equally crucial in industries such as aviation, manufacturing, and other domains to detect anomalies such as cracks or other unwanted conditions in machinery. Similarly images of the night sky can be segmented to detect previously unknow galaxies, stars and planets. Object detection and localization has profound use in places requiring constant automated monitoring of activities, such as in shopping malls, local stores, industrial plants, and so on. Also, it can be used to count objects and people in an area of interest and estimate various densities, such as traffic conditions at various signals.

We will begin this chapter by going through a few of the traditional methods of image segmentation so that we can appreciate how neural networks are different from their traditional counterparts. Then, we will look at object detection and localization techniques, followed by generative adversarial networks, which have gained lot of popularity recently because of their use and potential as a generative model to create synthetic data. This synthetic data can be used for training and inference in case there is not much data available or the data is expensive to procure. Alternatively, generative models can be used for style transfer from one domain to another. Finally, we end with some guidelines as to how TensorFlow models can be implemented in production with ease using TensorFlow's serving capabilities.

Image Segmentation

Image segmentation is a computer-vision task involving the partitioning of an image into pertinent segments, such as pixels within the same segment that share some common attributes. The attributes can differ from domain to domain and from task to task, with the major attributes being pixel intensity, texture, and color. In this section, we will go through some basic segmentation techniques, such as thresholding methods based on a histogram of pixel intensities, watershedding thresholding techniques, and so on, to get some insights about image segmentation before we start with the deep learning–based image-segmentation methods.

Binary Thresholding Method Based on Histogram of Pixel Intensities

Often in an image there are only two significant regions of interest—the object and the background. In such a scenario, a histogram of pixel intensities would represent a probability distribution that is bimodal; i.e., have high density around two-pixel intensity values. It would be easy to segment the object and the background by choosing a threshold pixel and setting all pixel intensities below the threshold as 255 and those above the threshold as 00. This activity would ensure that we have a background and an object

S. Pattanayak, *Pro Deep Learning with TensorFlow,* https://doi.org/10.1007/978-1-4842-3096-1_6

represented by white and black colors, not necessarily in that order. If an image is represented as $I(x, y)$ and a threshold t is selected based on the histogram of pixel intensities, then the new segmented image $I'(x, y)$ can be represented as

$$I'(x,y) = 0 \; when \; I(x,y) > t$$
$$= 255 \; when \; I(x,y) \leq t$$

When the bimodal histogram is not distinctly separated by a region of zero density in between, then a good strategy to choose a threshold t is to take the average of the pixel intensities at which the bimodal regions peak. If those peak intensities are represented by p_1 and p_2 then the threshold t can be chosen as

$$t = \frac{(p_1 + p_2)}{2}$$

Alternately, one may use the pixel intensity between p_1 and p_2 at which histogram density is minimum as the thresholding pixel intensity. If the histogram density function is represented by $H(p)$, where $p \in \{0,1,2..,255\}$ represents the pixel intensities, then

$$t = \underbrace{Arg \; Min}_{p \in [p_1, p_2]} H(p)$$

This idea of binary thresholding can be extended to multiple thresholding based on the histogram of pixel intensities.

Otsu's Method

Otsu's method determines the threshold by maximizing the variance between the different segments of the images. If using binary thresholding via Otsu's method, here are the steps to be followed:

- Compute the probability of each pixel intensity in the image. Given that N pixel intensities are possible, the normalized histogram would give us the probability distribution of the image.

$$P(i) = \frac{count(i)}{M} \qquad \forall i \in \{0,1,2,\ldots,N-1\}$$

- If the image has two segments C_1 and C_2 based on the threshold t, then the set of pixels $\{0, 2 \ldots . t\}$ belong to C_1 while the set of pixels $\{t+1, t+2 \ldots .. L-1\}$ belong to C_2. The variance between the two segments is determined by the sum of the square deviation of the mean of the clusters with respect to the global mean. The square deviations are weighted by the probability of each cluster.

$$var(C_{1,}C_2) = P(C_{1,})(u_1 - u)^2 + P(C_2)(u_2 - u)^2$$

where u_1, u_2 are the means of cluster 1 and cluster 2 while u is the overall global mean.

$$u_1 = \sum_{i=0}^{t} P(i)i \quad u_2 = \sum_{i=t+1}^{L-1} P(i)i \quad u = \sum_{i=0}^{L-1} P(i)i$$

The probability of each of the segments is the number of pixels in the image belonging to that class. The probability of segment C_1 is proportional to the number of pixels that have intensities less than or equal to the threshold intensity t, while that of segment C_2 is proportional to the number of pixels with intensities greater than threshold t. Hence,

$$P(C_1)=\sum_{i=0}^{t}P(i) \quad P(C_2)=\sum_{i=t+1}^{L-1}P(i)$$

- If we observe the expressions for u_1,u_2, $P(C_1)$, and $P(C_2)$, each of them is a function of the threshold t while the overall mean u is constant given an image. Hence, the between-segment variance $var(C_1,C_2)$ is a function of the threshold pixel intensity t. The threshold $\hat{t}$ that maximizes the variance would provide us with the optimal threshold to use for segmentation using Otsu's method:

$$\hat{t}=\underbrace{Arg\max}_{t} var\left(C_1,C_2\right)$$

Instead of computing a derivative and then setting it to zero to obtain $\hat{t}$, one can evaluate the $var(C_1,C_2)$ at all values of $t=\{0,1,2,\ldots,L-1\}$ and then choose the $\hat{t}$ at which the $var(C_1,C_2)$ is maximum.

Otsu's method can also be extended to multiple segments where instead of one threshold one needs to determine $(k-1)$ thresholds corresponding to k segments for an image.

The logic for both methods just illustrated—i.e., binary thresholding based on histogram of pixel intensities as well as Otsu's method—has been illustrated in Listing 6-1 for reference. Instead of using an image-processing package to implement these algorithms, the core logic has been used for ease of interpretability. Also, one thing to note is that these processes for segmentation are generally applicable on grayscale images or if one is performing segmentation per color channel.

Listing 6-1. Python Implementation of Binary Thresholding Method Based on Histogram of Pixel Intensities and Otsu's Method

```
## Binary thresholding Method Based on Histogram of Pixel Intensities
import cv2
import matplotlib.pyplot as plt
#%matplotlib inline
import numpy as np
img = cv2.imread("coins.jpg")
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
plt.imshow(gray,cmap='gray')
row,col = np.shape(gray)
gray_flat = np.reshape(gray,(row*col,1))[:,0]
ax = plt.subplot(222)
ax.hist(gray_flat,color='gray')
gray_const = []
## 150 pixel intensity seems a good threshold to choose since the density is minimum
##  round 150
for i in xrange(len(gray_flat)):
    if gray_flat[i] < 150 :
        gray_const.append(255)
    else:
```

```
        gray_const.append(0)
gray_const = np.reshape(np.array(gray_const),(row,col))
bx = plt.subplot(333)
bx.imshow(gray_const,cmap='gray')

## Otsu's thresholding Method

img = cv2.imread("otsu.jpg")
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
plt.imshow(gray,cmap='gray')
row,col = np.shape(gray)
hist_dist = 256*[0]
## Compute the frequency count of each of the pixels in the image
for i in xrange(row):
    for j in xrange(col):
        hist_dist[gray[i,j]] += 1
# Normalize the frequencies to produce probabilities
hist_dist = [c/float(row*col) for c in hist_dist]
plt.plot(hist_dist)
## Compute the between segment variance
def var_c1_c2_func(hist_dist,t):
    u1,u2,p1,p2,u = 0,0,0,0,0
    for i in xrange(t+1):
        u1 += hist_dist[i]*i
        p1 += hist_dist[i]
    for i in xrange(t+1,256):
        u2 += hist_dist[i]*i
        p2 += hist_dist[i]
    for i in xrange(256):
        u += hist_dist[i]*i
    var_c1_c2 = p1*(u1 - u)**2 + p2*(u2 - u)**2
    return var_c1_c2
## Iteratively run through all the pixel intensities from 0 to 255 and choose the one that
## maximizes the variance

variance_list = []
for i in xrange(256):
    var_c1_c2 = var_c1_c2_func(hist_dist,i)
    variance_list.append(var_c1_c2)
## Fetch the threshold that maximizes the variance
t_hat = np.argmax(variance_list)

## Compute the segmented image based on the threshold t_hat
gray_recons = np.zeros((row,col))
for i in xrange(row):
    for j in xrange(col):
        if gray[i,j] <= t_hat :
            gray_recons[i,j] = 255
        else:
            gray_recons[i,j] = 0
plt.imshow(gray_recons,cmap='gray')

--output --
```

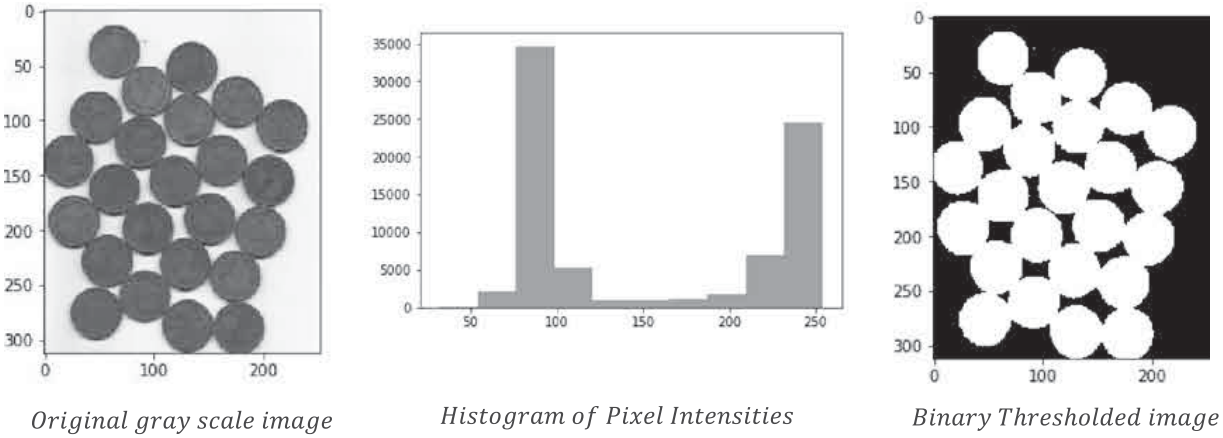


Figure 6-1. *Binary thresholding method based on histogram of pixel intensities*

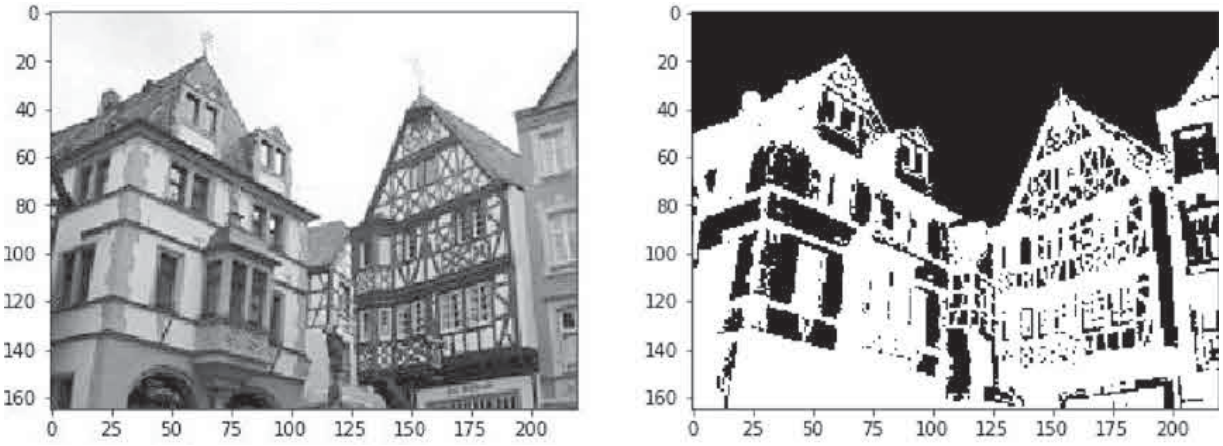


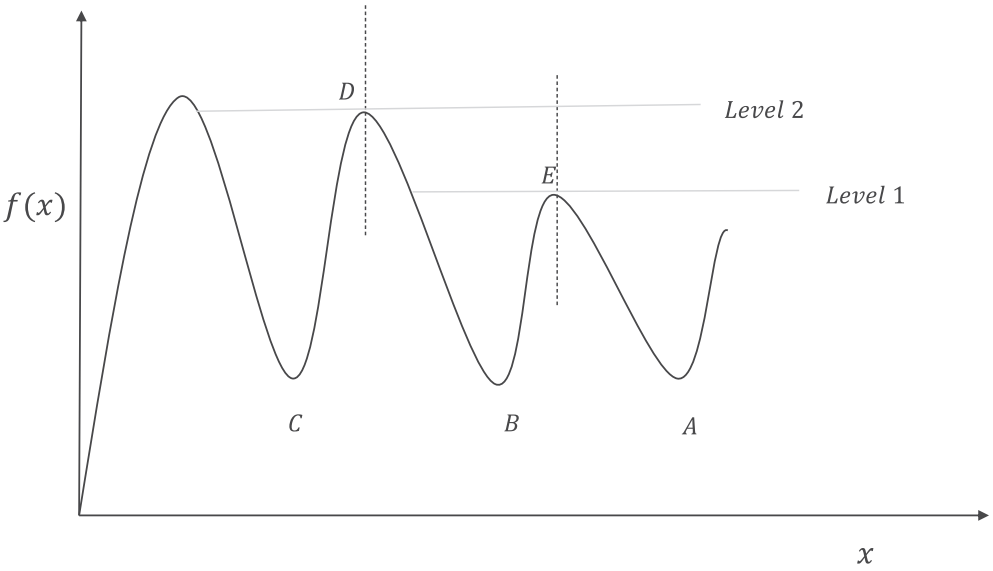
Figure 6-2. *Otsu's method of thresholding*

In Figure 6-1, the original grayscale image of the coin has been binary thresholded based on the histogram of pixel intensities to separate the objects (i.e., the coins) from the background. Based on the histogram of pixel intensities, the pixel intensity of 150 has been chosen as the thresholding pixel intensity. Pixel intensities below 150 have been set to 255 to represent the objects, while pixel intensities above 150 have been set to 0 0to represent the background.

Figure 6-2 illustrates Otsu's method of thresholding for an image to produce two segments determined by the black and white colors. The black color represents the background while white represents the house. The optimal threshold for the image is a pixel intensity of 143.

Watershed Algorithm for Image Segmentation

The Watershed algorithm aims at segmenting topologically-placed local regions around local minima of pixel intensities. If a grayscale image pixel intensity value is considered a function of its horizontal and vertical coordinates, then this algorithm tries to find regions around local minima called basins of attraction or catchment basins. Once these basins are identified, the algorithm tries to separate them by constructing separations or watersheds along high peaks or ridges. To get a better idea of the method, let's show this algorithm with a simple illustration as represented in Figure 6-3.



A, B, C — *Minima of Catchment Basins*
D, E — *Peaks or Maximas where watersheds need to be constructed*

Figure 6-3. *Watershed algorithm illustration*

If we start filling water in the catchment basin with its minima as *B*, water would keep on filling the basin up to Level 1, at which point an extra drop of water has a chance of spilling over to the catchment basin at *A*. To prevent the spilling of water, one needs to build a dam or watershed at *E*. Once we have built a watershed at *E*, we can continue filling water in the catchment basin *B* till Level 2, at which point an extra drop of water has a chance of spilling over to the catchment basin *C*. To prevent this spilling of water to *C*, one needs to build a watershed at *D*. Using this logic, we can continue to build watersheds to separate such catchment basins. This is the principal idea behind the Watershed algorithm. Here, the function is univariate, whereas in the case of a grayscale image the function representing the pixel intensities would be a function of two variables: the vertical and the horizontal coordinates.

The Watershed algorithm is particularly useful in detecting objects when there is overlap between them. Thresholding techniques are unable to determine distinct object boundaries. We will work through illustrating this in Listing 6-2 by applying Watershed techniques to an image containing overlapping coins.

Listing 6-2. Image Segmentation Using Watershed Algorithm

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from scipy import ndimage
from skimage.feature import peak_local_max
from skimage.morphology import watershed

## Load the coins image
im = cv2.imread("coins.jpg")
## Convert the image to grayscale
imgray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
plt.imshow(imgray,cmap='gray')
# Threshold the image to convert it to Binary image based on Otsu's method
```

```
thresh = cv2.threshold(imgray, 0, 255,
     cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
## Detect the contours and display them
im2, contours, hierarchy = cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
y = cv2.drawContours(imgray, contours, -1, (0,255,0), 3)
## If we see the contour plots in the display of "y"
## we see that the coins have a common contour and hence it is not possible to separate them
plt.imshow(y,cmap='gray')
## Hence we will proceed with the Watershed algorithm so that each of the coins form its own
## cluster and thus it's possible to have separate contours for each coin.
## Relabel the thresholded image to be consisting of only 0 and 1
## as the input image to distance_transform_edt should be in this format.
thresh[thresh == 255] = 5
thresh[thresh == 0] = 1
thresh[thresh == 5] = 0
## The distance_transform_edt and the peak_local_max functions help building the markers by
detecting
## points near the center points of the coins. One can skip these steps and create a marker
## manually by setting one pixel within each coin with a random number representing its
cluster
D = ndimage.distance_transform_edt(thresh)
localMax = peak_local_max(D, indices=False, min_distance=10,
     labels=thresh)
markers = ndimage.label(localMax, structure=np.ones((3, 3)))[0]
# Provide the EDT distance matrix and the markers to the watershed algorithm to detect the
cluster's
# labels for each pixel. For each coin, the pixels corresponding to it will be filled with
the cluster number
labels = watershed(-D, markers, mask=thresh)
print("[INFO] {} unique segments found".format(len(np.unique(labels)) - 1))
# Create the contours for each label(each coin) and append to the plot
for k in np.unique(labels):
    if k != 0 :
        labels_new = labels.copy()
        labels_new[labels == k] = 255
        labels_new[labels != k] = 0
        labels_new = np.array(labels_new,dtype='uint8')
        im2, contours, hierarchy = cv2.findContours(labels_new,cv2.RETR_TREE,cv2.CHAIN_
APPROX_SIMPLE)
        z = cv2.drawContours(imgray,contours, -1, (0,255,0), 3)
        plt.imshow(z,cmap='gray')

--output --
```

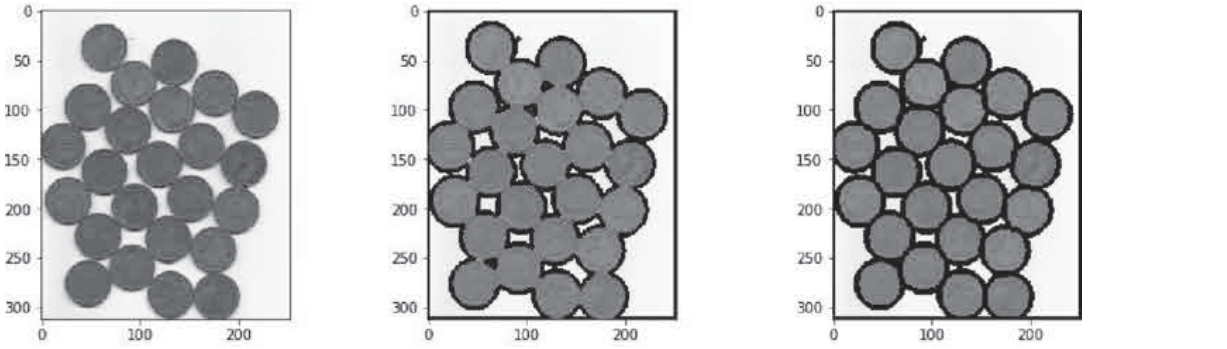


Figure 6-4. *Illustration of Watershed algorithm for image segmentation)*

As we can see from Figure 6-4, the borders for overlapping coins are distinct after applying the Watershed algorithm, whereas the other thresholding methods are not able provide a distinct border to each of the coins.

Image Segmentation Using K-means Clustering

The famous *K*-means algorithm can be used to segment images, especially medical images. The term *K* is a parameter to the algorithm which determines the number of distinct clusters to be formed. The algorithm works by forming clusters, and each such cluster is represented by its cluster centroids based on specific input features. For image segmentation through *K*-means, generally the segmentation is based on input features such as pixel intensity and its three spatial dimensions; i.e., horizontal and vertical coordinates and the color channel. So, the input feature vector can be represented as $u \in \mathbb{R}^{4\times 1}$, where

$$u = [I(x,y,z), x, y, z]^T$$

Similarly, one can ignore the spatial coordinates and take the pixel intensities along the three color channels as the input feature vector; i.e.,

$$u = [I_R(x,y), I_G(x,y), I_B(x,y)]^T$$

where $I_R(x,y), I_G(x,y)$, and $I_B(x,y)$ represent the pixel intensities along the Red, Green, and Blue channels respectively at the spatial coordinates (x,y).

The algorithm uses a distance measure such as an L^2 or L^1 norm, as shown here:

$$D\left(u^{(i)}, u^{(j)} / L^2\right) = \left\| u^{(i)} - u^{(j)} \right\|_2 = \sqrt{u^{(i)} - u^{(j)})^T \left(u^{(i)} - u^{(j)}\right)}$$

$$D(u_i, u_j / L^1) = \left\| u^{(i)} - u^{(j)} \right\|_1$$

The following are the working details of the *K*-means algorithm

- *Step 1* – Start with *K* randomly selected cluster centroids $C_1, C_2 \ldots C_k$ corresponding to the *K* clusters $S_1, S_2 \ldots S_k$.
- *Step 2* – Compute the distance of each pixel feature vector $u^{(i)}$ from the cluster centroids and tag it to the cluster S_j if the pixel has a minimum distance from its cluster centroid C_j:

$$j = \underbrace{Arg\ Min}_{j} \left\| u^{(i)} - C \right\|_{j2}$$

- This process needs to be repeated for all the pixel-feature vectors so that in one iteration of *K*-means all the pixels are tagged to one of the *K* clusters.
- *Step 3* – Once the new centroids clusters have been assigned for all the pixels, the centroids are recomputed by taking the mean of the pixel-feature vectors in each cluster:

$$C_j = \sum_{u^{(i)} \in S_j} u^{(i)}$$

- Repeat *Step 2* and *Step 3* for several iterations until the centroids no longer change. Through this iterative process, we are reducing the sum of the intra-cluster distances, as represented here:

$$L = \sum_{j=1}^{K} \sum_{u^{(i)} \in S_j} \left\| u^{(i)} - C_j \right\|_2$$

A simple implementation of the *K*-means algorithm is shown in Listing 6-3, taking the pixel intensities in the three color channels as features. The image segmentation is implemented with $K = 3$. The output is shown in grayscale and hence may not reveal the actual quality of the segmentation. However, if the same segmented image as produced in Listing 6-3 is displayed in a color format, it would reveal the finer details of the segmentation. One more thing to add: the cost or loss functions minimized—i.e., the sum of the intra-cluster distances—is a non-convex function and hence might suffer from local minima problems. One can trigger the segmentation several times with different initial values for the cluster centroids and then take the one that minimizes the cost function the most or produces a reasonable segmentation.

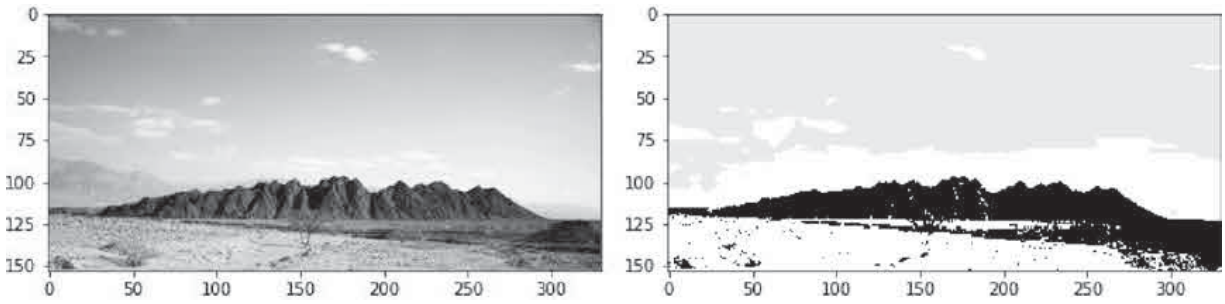
Listing 6-3. Image Segmentation Using K-means

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#np.random.seed(0)
img = cv2.imread("kmeans.jpg")
imgray_ori = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
plt.imshow(imgray_ori,cmap='gray')
## Save the dimensions of the image
row,col,depth = img.shape
## Collapse the row and column axis for faster matrix operation.
img_new = np.zeros(shape=(row*col,3))
glob_ind = 0
for i in xrange(row):
    for j in xrange(col):
        u = np.array([img[i,j,0],img[i,j,1],img[i,j,2]])
```

```
        img_new[glob_ind,:] = u
        glob_ind += 1
# Set the number of clusters
K = 5
# Run the K-means for
num_iter = 20
for g in xrange(num_iter):
# Define cluster for storing the cluster number and out_dist to store the distances from
centroid
    clusters = np.zeros((row*col,1))
    out_dist = np.zeros((row*col,K))
    centroids = np.random.randint(0,255,size=(K,3))
    for k in xrange(K):
        diff = img_new - centroids[k,:]
        diff_dist = np.linalg.norm(diff,axis=1)
        out_dist[:,k] = diff_dist
# Assign the cluster with minimum distance to a pixel location
    clusters = np.argmin(out_dist,axis=1)
# Recompute the clusters
    for k1 in np.unique(clusters):
        centroids[k1,:] = np.sum(img_new[clusters == k1,:],axis=0)/np.sum([clusters == k1])
# Reshape the cluster labels in two-dimensional image form
clusters = np.reshape(clusters,(row,col))
out_image = np.zeros(img.shape)
#Form the 3D image with the labels replaced by their correponding centroid pixel intensities
for i in xrange(row):
    for j in xrange(col):
        out_image[i,j,0] = centroids[clusters[i,j],0]
        out_image[i,j,1] = centroids[clusters[i,j],1]
        out_image[i,j,2] = centroids[clusters[i,j],2]

out_image = np.array(out_image,dtype="uint8")
# Display the output image after converting into grayscale
# Readers advised to display the image as it is for better clarity
imgray = cv2.cvtColor(out_image,cv2.COLOR_BGR2GRAY)
plt.imshow(imgray,cmap='gray')

---Output ---
```



Original Image *Segmented Image with $K = 3$*

Figure 6-5. *Illustration of segmentation through the K-means algorithm*

We can see from Figure 6-5 that K-means clustering has done a good job segmenting the image for $K = 3$.

Semantic Segmentation

Image segmentation through convolutional neural networks has gained a lot of popularity in recent years. One of the things significantly different when segmenting an image through neural networks is the annotation process of assigning each pixel to an object class so that the training of such a segmentation network is totally supervised. Although the process of annotating images is a costly affair, it simplifies the problem by having a ground truth to compare to. The ground truth would be in the form of an image with the pixels holding a representative color for a specific object. For example, if we are working with a cats and dogs image and the images can have a background, then each pixel for an image can belong to one of the three classes—cat, dog, and background. Also, each class of object is generally represented by a representative color so that the ground truth can be displayed as a segmented image. Let's go through some convolutional neural networks that can perform semantic segmentation.

Sliding-Window Approach

One can extract patches of images from the original image by using a sliding window and then feeding those patches to a classification convolutional neural network to predict the class of the central pixel for each of the image patches. Training such a convolutional neural network with this sliding-window approach is going to be computationally intensive, both at training and at test time, since at least N number of patches per image need to be fed to the classification CNN, where N denotes the number of pixels in the image.

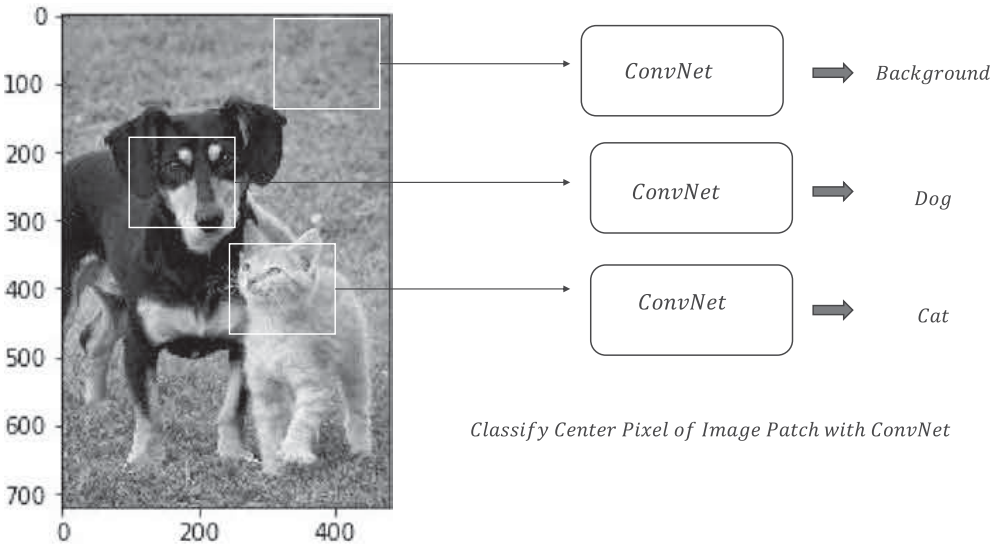


Figure 6-6. *Sliding-window semantic segmentation*

Illustrated in Figure 6-6 is a sliding-window semantic segmentation network for segmenting images of cat, dog, and background. It crops out patches from the original image and feeds it through the classification CNN for classifying the center pixel in the patch. A pre-trained network such as AlexNet, VGG19, Inception V3, and so forth can be used as the classification CNN, with the output layer replaced to have only three 3classes pertaining to the labels for dog, cat, and background. The CNN can then be fine-tuned through backpropagation, with the image patches as input and the class label of the center pixel of the input image patch as output. Such a network is highly inefficient from a convolution of images standpoint since image patches next to each other will have significant overlap and re-processing them every time independently leads to unwanted computational overhead. To overcome the shortcomings of the above Network one can use a Fully Convolutional Network which is our next topic of discussion.

Fully Convolutional Network (FCN)

A fully convolutional network consists of a series of convolutional layers without any fully connected layers. The convolutions are chosen such that the input image is transmitted without any change in the spatial dimensions; i.e., the height and width of the image remains the same. Rather than having individual patches from an image independently evaluated for pixel category, as in the sliding-window approach, a fully convolutional network predicts all the pixel categories at once. The output layer of this network consists of C feature maps, where C is the number of categories, including the background, that each pixel can be classified into. If the height and width of the original image are h and w respectively, then the output consists of C number of $h \times w$ feature maps. Also, for the ground truth there should be C number of segmented images corresponding to the C classes. At any spatial coordinate (h_1, w_1), each of the feature maps contains the score of that pixel pertaining to the class the feature map is tied to. These scores across the feature maps for each spatial pixel location (h_1, w_1) form a SoftMax over the different classes.

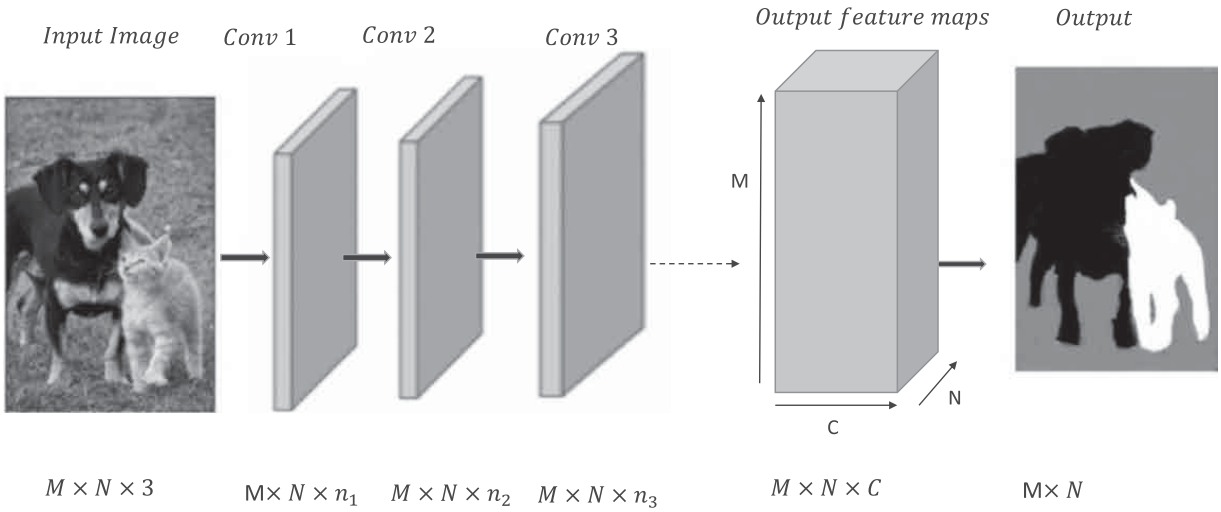


Figure 6-7. *Fully convolutional network architecture*

Figure 6-7 contains the architectural design of a fully convolutional network. The number of output feature maps as well as ground truth feature maps would be three, corresponding to the three classes. If the input net activation or score at the spatial coordinate (i,j) for the *kth* class is denoted by $s_k^{(i,j)}$, then the probability of the *kth* class for the pixel at spatial coordinate (i,j) is given by the SoftMax probability, as shown here:

$$P_k(i,\mathrm{j}) = \frac{e^{s_k^{(i,j)}}}{\sum_{k'=1}^{C} e^{s_{k'}^{(i,j)}}}$$

Also, if the ground truth labels at the spatial coordinate (i,j) for the *kth* class are given by $y_k(i,\mathrm{j})$, then the cross-entropy loss of the pixel at spatial location (i,j) can be denoted by

$$L(i,j) = -\sum_{k=1}^{C} y_k(i,\mathrm{j}) \log P_k(i,\mathrm{j})$$

If the height and width of the images fed to the network are M and N respectively, then the total loss for an image is

$$L = -\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\sum_{k=1}^{C} y_k(i,\mathrm{j}) \log P_k(i,\mathrm{j})$$

The images can be fed as a mini batch to the network, and hence the average loss per image can be taken as the loss or cost function to be optimized in each epoch of mini-batch learning through gradient descent.

The output class $\hat{k}$ for a pixel at spatial location (i,j) can be determined by taking the class k for which the probability $P_k(i,\mathrm{j})$ is maximum; i.e.,

$$\hat{k} = \underbrace{Arg\max}_{k} P_k(i,\mathrm{j})$$

The same activity needs to be performed for pixels at all spatial locations of an image to get the final segmented image.

In Figure 6-8, the output feature maps of a network for segmenting images of cats, dogs, and background are illustrated. As we can see, for each of the three categories or classes there is a separate feature map. The spatial dimensions of the feature maps are the same as those of the input image. The net input activation, the associated probability, and the corresponding ground label have been shown at the spatial coordinate (i,j) for all the three classes.

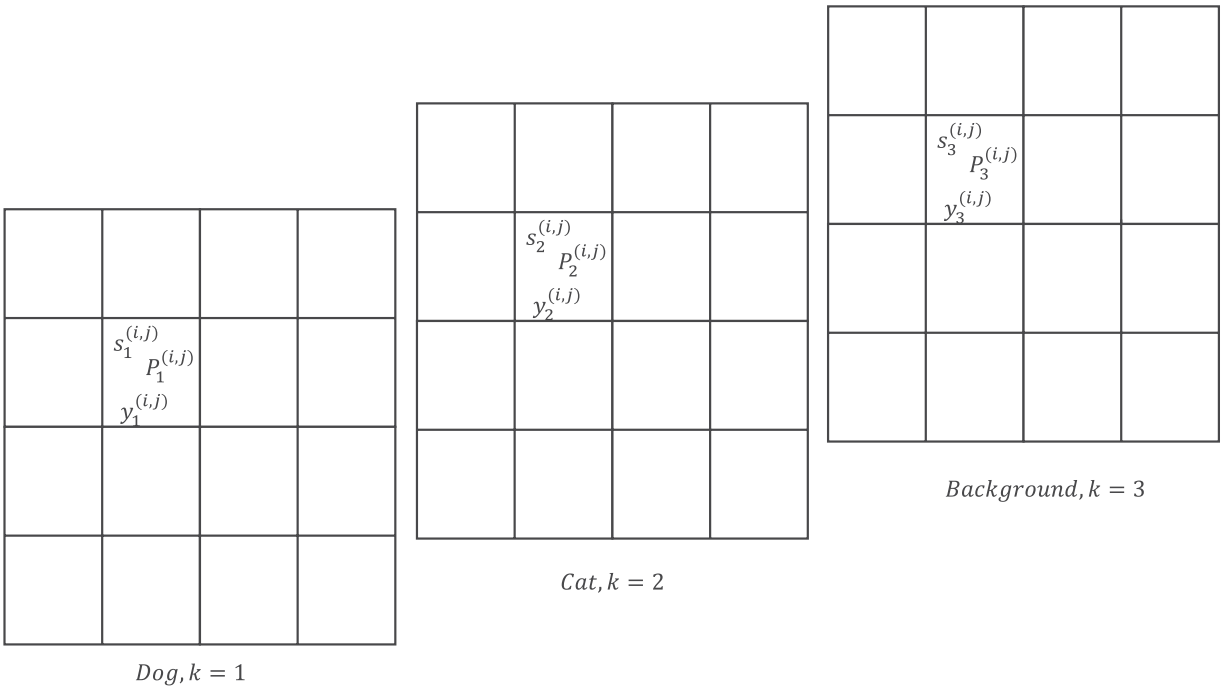


Figure 6-8. *Output feature maps corresponding to each of the three classes for dog, cat, and background*

All the convolutions in this network retain the spatial dimensions of the input image. So, for high-resolution images the network would be computationally intensive, especially if the number of feature maps or channels in each convolution is high. To address this problem, a different variant of a fully convolutional neural network is more widely used that both downsamples the image in the first half of the network and then upsamples the images within the second half of the network. This modified version of the fully convolutional network is going to be our next topic of discussion.

Fully Convolutional Network with Downsampling and Upsampling

Instead of preserving the spatial dimensions of the images in all convolutional layers as in the previous network, this variant of the fully convolutional network uses a combination of convolutions where the image is downsampled in the first half of the network and then upsampled in the final layers to restore the spatial dimensions of the original image. Generally, such a network consists of several layers of downsampling through strided convolutions and/or pooling operations and then a few layers of upsampling. Until now, through the convolution operation we have either downsampled the image or kept the spatial dimensions of the output image the same as those of the input. In this network, we would need to upsample an image, or rather the feature maps. Illustrated in Figure 6-9 is a high-level architectural design of such a network.

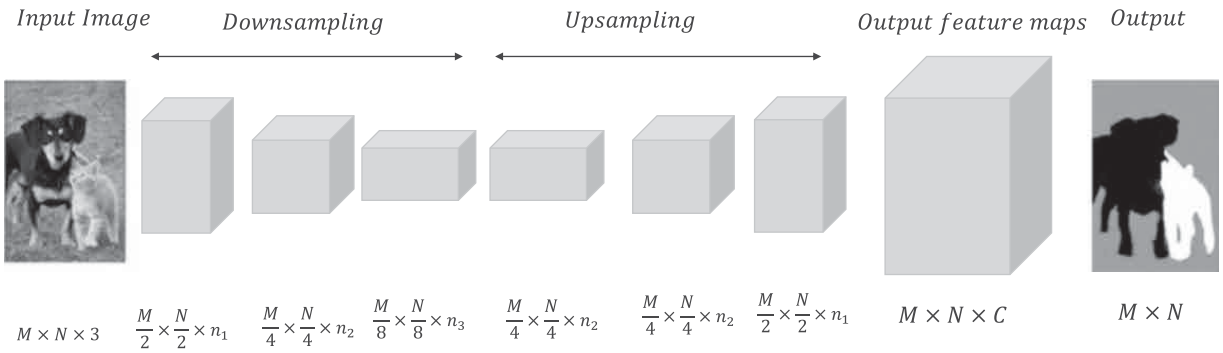


Figure 6-9. *Fully convolutional network with downsampling and upsampling*

The techniques that are commonly used to upsample an image or feature map are as discussed next.

Unpooling

Unpooling can be treated as the reverse operation to pooling. In max pooling or average pooling we reduce the spatial dimensions of the image by either taking the maximum or the average of the pixel value based on the size of the pooling kernel. So, if we have a 2×2 kernel for pooling, the spatial dimensions of the image get reduced by $\frac{1}{2}$ in each spatial dimension. In unpooling, we generally increase the spatial dimensions of the image by repeating a pixel value in a neighborhood, as shown in Figure 6-10 (A).

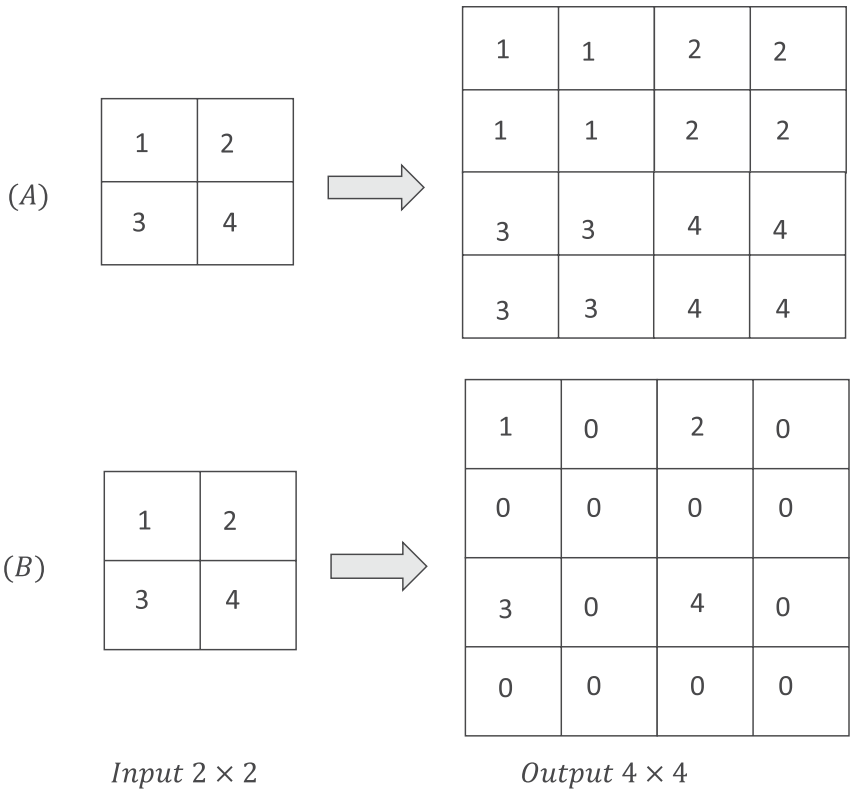


Figure 6-10. *Unpooling operation*

Similarly, one may choose to populate only one pixel in the neighborhood and set the rest to zero, as illustrated in Figure 6-10 (B).

Max Unpooling

Many of the fully convolutional layers are symmetric, as a pooling operation in the first half of the network would have a corresponding unpooling in the second half of the network to restore the image size. Whenever pooling is performed, minute spatial information about the input image is lost because of the summarizing of the results of neighboring pixels by one representative element. For instance, when we do max pooling by a 2×2 kernel, the maximum pixel value of each neighborhood is passed on to the output to represent the 2×2 neighborhood. From the output, it would not be possible to infer the location of the maximum pixel value. So, in this process we are missing the spatial information about the input. In semantic segmentation, we want to classify each pixel as close to its true label as possible. However, because of max pooling, a lot of information about edges and other finer details of the image is lost. While we are trying to rebuild the image through unpooling, one way we can restore a bit of this lost spatial information is to place the value of the input pixel in the output location corresponding to the one where the max pooling output got its input from. To visualize it better, let's look at the illustration in Figure 6-11.

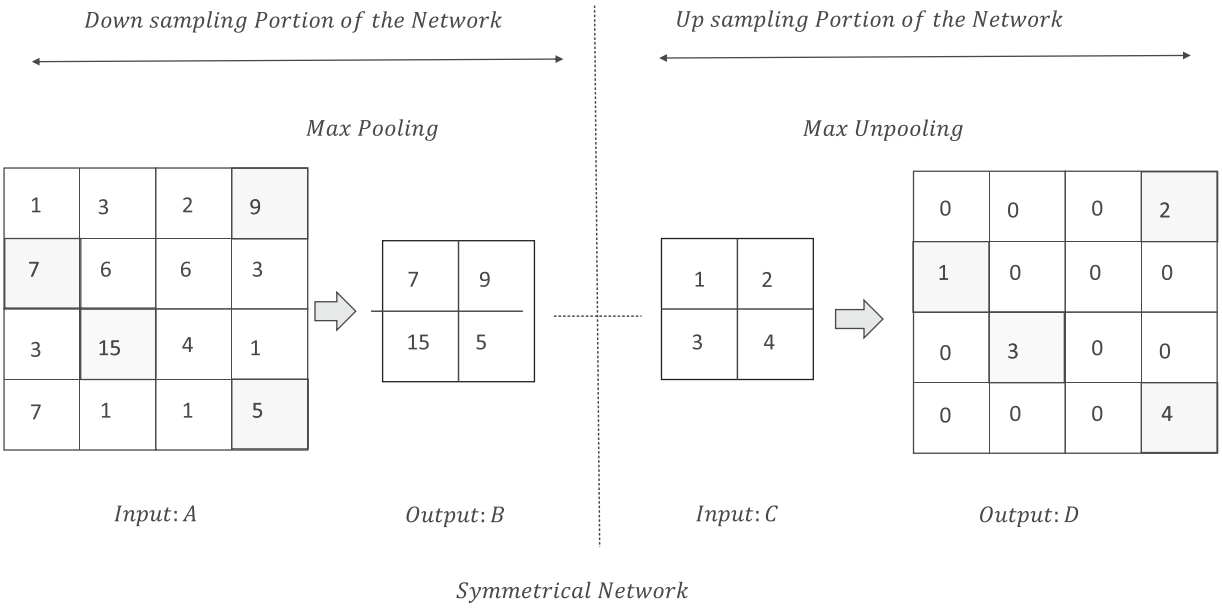


Figure 6-11. *Max unpooling illustration for a symmetric fully connected segmentation network*

As we can see from the Figure 6-11 while Unpooling only the locations in output map *D* corresponding to the location of the maximal elements in Input A with respect to MaxPooling are populated with values. This method of unpooling is generally called *max unpooling*.

Transpose Convolution

The upsampling done through unpooling or max unpooling are fixed transformations. These transformations don't involve any parameters that the network needs to learn while training the network. A learnable way to do upsampling is to perform upsampling though transpose convolution, which is much like convolution operations that we know of. Since transpose convolution involves parameters that would be learned by the network, the network would learn to do the upsampling in such a way that the overall cost function on which the network is trained reduces. Now, let's get into the details of how transpose convolution works.

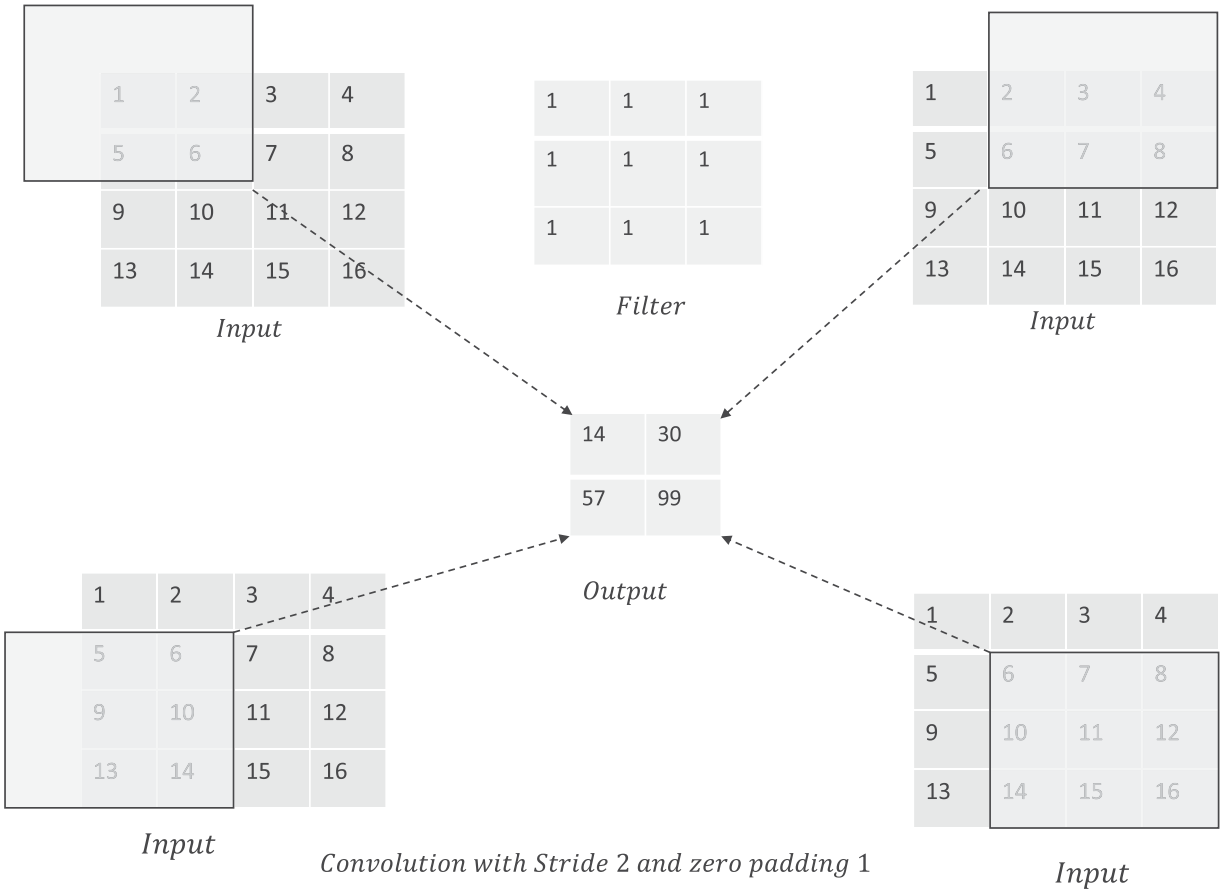


Figure 6-12. *Strided convolution operation for downsampling an image*

In strided convolution, the output dimensions are almost half those of the input for each spatial dimension for a stride of 2. Figure 6-12 illustrates the operation of convolving a 2D input of dimension 5×5 with a 4×4 kernel with a stride of 2 and zero padding of 1. We slide the kernel over the input, and at each position the kernel is on, the dot product of the kernel is computed with the portion of the input the kernel is overlapping with.

In transpose convolution, we use the same kind of logic, but instead of downsampling, strides greater than 1 provide upsampling. So, if we use a stride of 2, then the input size is doubled in each spatial dimension. Figures 6-13a, 6-13b, and 6-13c illustrate the operation of transpose convolution for an input of dimension 2×2 by a filter or kernel size of 3×3 to produce a 4×4 output. Unlike the dot product between the filter and the portions of input as in convolution, in transpose convolution, at a specific location the filter values are weighted by the input value at which the filter is placed, and the weighted filter values are populated in the corresponding locations in the output. The outputs for successive input values along the same spatial dimension are placed at a gap determined by the stride of the transpose convolution. The same is performed for all input values. Finally, the outputs corresponding to each of the input values are added to produce the final output, as shown in Figure 6-13c.

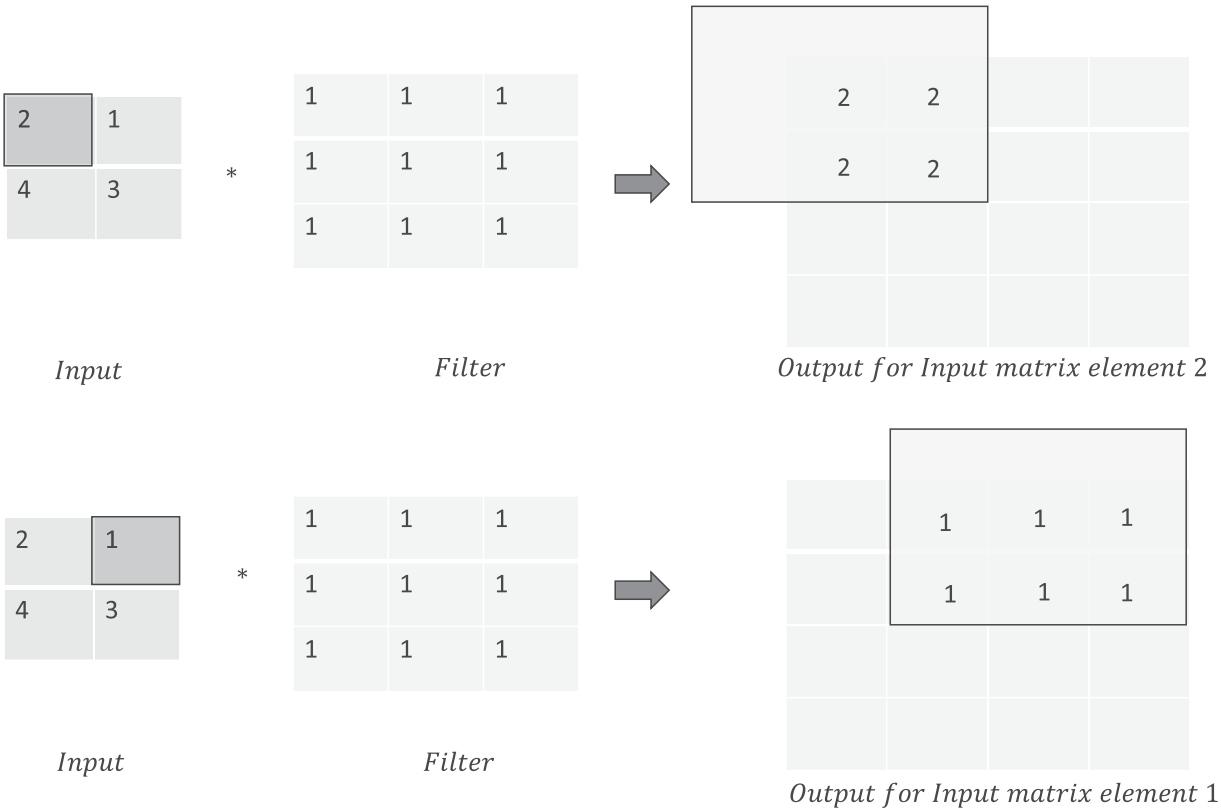
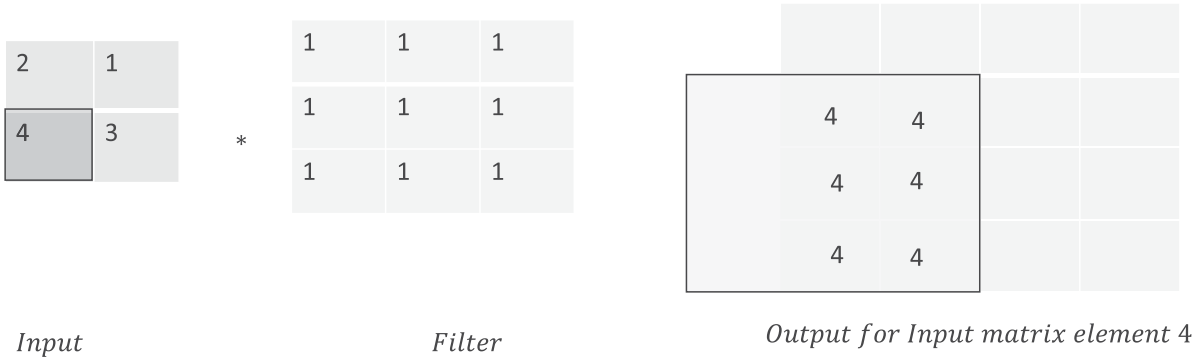


Figure 6-13a. *Transpose convolution for upsampling*



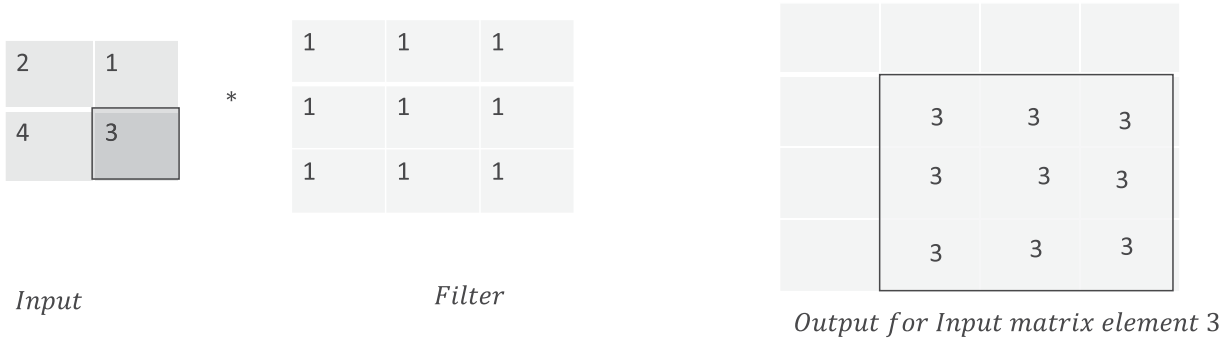


Figure 6-13b. *Transpose convolution for upsampling*

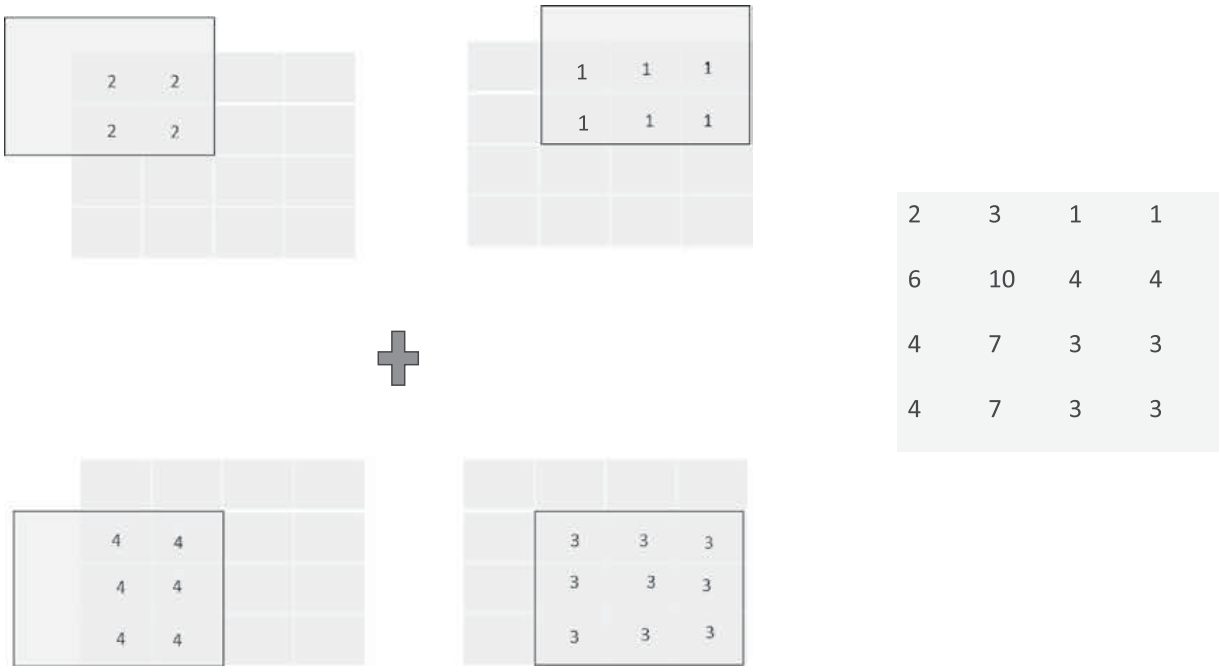


Figure 6-13c. *Transpose convolution for upsampling*

In TensorFlow, the function `tf.nm.conv2d_transpose` can be used to perform upsampling through transpose convolution.

U-Net

The U-Net convolutional neural network is one of the most efficient architectures in recent times for the segmentation of images, especially medical images. This U-Net architecture won the Cell Tracking Challenge at ISBI 2015. The network topology follows a U-shape pattern from the input to the output layers, hence the name U-Net. Olaf Ronneberger, Philipp Fischer, and Thomas Brox came up with this convolutional neural network for segmentation, and the details of the model are illustrated in the white paper "U-Net: Convolutional Networks for Biomedical Image Segmentation." The paper can be located at `https://arxiv.org/abs/1505.04597`.

In the first part of the network, the images undergo downsampling through a combination of convolution and max-pooling operations. The convolutions are associated with pixel-wise ReLU activations. Every convolution operation is with 3×3 filter size and without zero padding, which leads to a reduction of two pixels in each spatial dimension for the output feature maps. In the second part of the network, the downsampled images are upsampled till the final layer, where output feature maps correspond to specific classes of objects being segmented. The cost function per image would be pixel-wise categorical cross-entropy or log loss for classification summed over the whole image, as we saw earlier. One thing to note is that the output feature maps in a U-Net have less spatial dimension than those of the input. For example, an input image having spatial dimensions of 572×572 produces output feature maps of spatial dimension 388×388. One might ask how the pixel-to-pixel class comparison is done for loss computation for training. The idea is simple—the segmented output feature maps are compared to a ground truth segmented image of patch size 388×388 extracted from the center of the input image. The central idea is that if one has images of higher resolution, say 1024×1024, one can randomly create many images of spatial dimensions 572×572 from it for training purposes. Also, the ground truth images are created from these 572×572 sub-images by extracting the central 388×388 patch and labeling each pixel with its corresponding class. This helps train the network with a significant amount of data even if there are not many images around for training. Illustrated in Figure 6-14 is the architecture diagram of a U-Net.

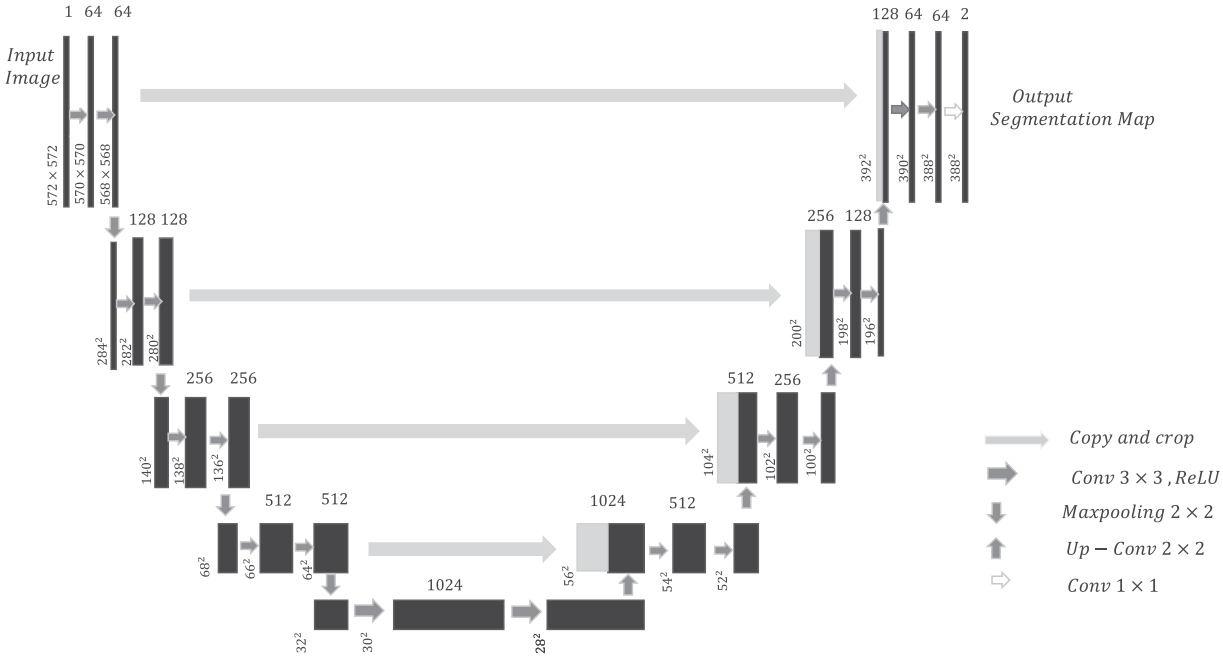


Figure 6-14. *U-Net architecture diagram*

We can see from the architecture diagram that in the first part of the network the images undergo convolution and max pooling to reduce the spatial dimensions and at the same time increase the channel depth; i.e., to increase the number of feature maps. Every two successive convolutions, along with their associated ReLU activations, are followed by a max-pooling operation, which reduces the image size by $\frac{1}{4}$.

Every max-pooling operation brings the network down to the next set of convolutions and contributes to the U-shape in the first part of the network. Similarly, the upsampling layers increase the spatial dimensions by two in each dimension and hence increase the image size by four times. Also, it provides the U structure to the network in the second part. After every upsampling, the image goes through two convolutions and their associated ReLU activations.

The U-Net is a very symmetric network as far as the operations of max pooling and upsampling are concerned. However, for a corresponding pair of max pooling and upsampling, the image size before the max pooling is not the same as the image size after upsampling, unlike with other fully convolutional layers. As discussed earlier, when a max-pooling operation is done a lot of spatial information is lost by having a representative pixel in the output corresponding to a local neighborhood of the image. It becomes difficult to recover this lost spatial information when the image is upsampled back to its original dimensions, and hence the new image lacks a lot of information around the edges and other finer aspects of the image too. This leads to sub-optimal segmentation. Had the upsampled image been of the same spatial dimensions as the image before its corresponding max-pooling operation, one could have just appended a random number of feature maps before max pooling with the output feature maps after upsampling to help the network recover a bit of the lost spatial information. Since in the case of U-Net these feature-map dimensions don't match, U-Net crops the feature maps before max pooling to be of the same spatial dimensions as the output feature maps from upsampling and concatenates them. This leads to better segmentation of images, as it helps recover some spatial information lost during max pooling. One more thing to note is that upsampling can be done by any of the methods that we have looked at thus far, such as unpooling, max unpooling, and transpose convolution, which is also known as deconvolution.

Few of the big wins with U-Net Segmentation are as below:

- A significant amount of training data can be generated with only a few annotated or hand-labeled segmented images.
- The U-Net does good segmentation even when there are touching objects of the same class that need to be separated. As we saw earlier with traditional image-processing methods, separating touching objects of the same class is tough, and methods such as the Watershed algorithm require a lot of input in terms of object markers to come up with reasonable segmentation. The U-Net does good separation between touching segments of the same class by introducing high weights for misclassification of pixels around the borders of touching segments.

Semantic Segmentation in TensorFlow with Fully Connected Neural Networks

In this section, we will go through the working details of a TensorFlow implementation for the segmentation of car images from the background based on the Kaggle Competition named Carvana. The input images, along with their ground truth segmentation, are available for training purposes. We train the model on 80 percent of the training data and validate the performance of the model on the remaining 20 percent of the data. For training, we use a fully connected convolutional network with a U-Net-like structure in the first half of the network followed by upsampling through transpose convolution. A couple of things different in this network from a U-Net is that the spatial dimensions are kept intact while performing convolution by using padding as *SAME*. The other thing different is that this model doesn't use the skip connections to concatenate feature maps from the downsampling stream to the upsampling stream. The detailed implementation is provided in Listing 6-4.

Listing 6-4. Semantic Segmentation in TensorFlow with Fully Connected Neural Network

```
## Load the required packages
import tensorflow as tf
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
import os
from subprocess import check_output
import numpy as np
from keras.preprocessing.image import array_to_img, img_to_array, load_img,
ImageDataGenerator
from scipy.misc import imresize

# Define downsampling - 2 (Conv+ReLU) and 1 Maxpooling
# Maxpooling can be set to False when needed

x = tf.placeholder(tf.float32,[None,128,128,3])
y = tf.placeholder(tf.float32,[None,128,128,1])

def down_sample(x,w1,b1,w2,b2,pool=True):
    x = tf.nn.conv2d(x,w1,strides=[1,1,1,1],padding='SAME')
    x = tf.nn.bias_add(x,b1)
    x = tf.nn.relu(x)
    x = tf.nn.conv2d(x,w2,strides=[1,1,1,1],padding='SAME')
    x = tf.nn.bias_add(x,b2)
    x = tf.nn.relu(x)
    if pool:
        y = tf.nn.max_pool(x,ksize=[1,2,2,1],strides=[1,2,2,1],padding='SAME')
        return y,x
    else:
        return x

# Define upsampling
def up_sample(x,w,b):
    output_shape  = x.get_shape().as_list()
    output_shape[0] = 32
    output_shape[1] *= 2
    output_shape[2] *= 2
    output_shape[1] = np.int(output_shape[1])
    output_shape[2] = np.int(output_shape[2])
    output_shape[3] = w.get_shape().as_list()[2]
    conv_tf = tf.nn.conv2d_transpose(value=x,filter=w,output_shape=output_shape,strides=
    [1,2,2,1],padding="SAME")
    conv_tf = tf.nn.bias_add(conv_tf,b)
    return tf.nn.relu(conv_tf)
```

```
## Define weights

weights = {
    'w11': tf.Variable(tf.random_normal([3,3,3,64],mean=0.0,stddev=0.02)),
    'w12': tf.Variable(tf.random_normal([3,3,64,64],mean=0.0,stddev=0.02)),
    'w21': tf.Variable(tf.random_normal([3,3,64,128],mean=0.0,stddev=0.02)),
    'w22': tf.Variable(tf.random_normal([3,3,128,128],mean=0.0,stddev=0.02)),
    'w31': tf.Variable(tf.random_normal([3,3,128,256],mean=0.0,stddev=0.02)),
    'w32': tf.Variable(tf.random_normal([3,3,256,256],mean=0.0,stddev=0.02)),
    'w41': tf.Variable(tf.random_normal([3,3,256,512],mean=0.0,stddev=0.02)),
    'w42': tf.Variable(tf.random_normal([3,3,512,512],mean=0.0,stddev=0.02)),
    'w51': tf.Variable(tf.random_normal([3,3,512,1024],mean=0.0,stddev=0.02)),
    'w52': tf.Variable(tf.random_normal([3,3,1024,1024],mean=0.0,stddev=0.02)),
    'wu1': tf.Variable(tf.random_normal([3,3,1024,1024],mean=0.0,stddev=0.02)),
    'wu2': tf.Variable(tf.random_normal([3,3,512,1024],mean=0.0,stddev=0.02)),
    'wu3': tf.Variable(tf.random_normal([3,3,256,512],mean=0.0,stddev=0.02)),
    'wu4': tf.Variable(tf.random_normal([3,3,128,256],mean=0.0,stddev=0.02)),
    'wf': tf.Variable(tf.random_normal([1,1,128,1],mean=0.0,stddev=0.02))
}

biases = {
    'b11': tf.Variable(tf.random_normal([64],mean=0.0,stddev=0.02)),
    'b12': tf.Variable(tf.random_normal([64],mean=0.0,stddev=0.02)),
    'b21': tf.Variable(tf.random_normal([128],mean=0.0,stddev=0.02)),
    'b22': tf.Variable(tf.random_normal([128],mean=0.0,stddev=0.02)),
    'b31': tf.Variable(tf.random_normal([256],mean=0.0,stddev=0.02)),
    'b32': tf.Variable(tf.random_normal([256],mean=0.0,stddev=0.02)),
    'b41': tf.Variable(tf.random_normal([512],mean=0.0,stddev=0.02)),
    'b42': tf.Variable(tf.random_normal([512],mean=0.0,stddev=0.02)),
    'b51': tf.Variable(tf.random_normal([1024],mean=0.0,stddev=0.02)),
    'b52': tf.Variable(tf.random_normal([1024],mean=0.0,stddev=0.02)),
    'bu1': tf.Variable(tf.random_normal([1024],mean=0.0,stddev=0.02)),
    'bu2': tf.Variable(tf.random_normal([512],mean=0.0,stddev=0.02)),
    'bu3': tf.Variable(tf.random_normal([256],mean=0.0,stddev=0.02)),
    'bu4': tf.Variable(tf.random_normal([128],mean=0.0,stddev=0.02)),
    'bf': tf.Variable(tf.random_normal([1],mean=0.0,stddev=0.02))
}

## Create the final model

def unet_basic(x,weights,biases,dropout=1):

    ## Convolutional 1
    out1,res1 = down_sample(x,weights['w11'],biases['b11'],weights['w12'],biases['b12'],
    pool=True)
    out1,res1 = down_sample(out1,weights['w21'],biases['b21'],weights['w22'],biases['b22'],
    pool=True)
    out1,res1 = down_sample(out1,weights['w31'],biases['b31'],weights['w32'],biases['b32'],
    pool=True)
    out1,res1 = down_sample(out1,weights['w41'],biases['b41'],weights['w42'],biases['b42'],
    pool=True)
```

```
    out1      = down_sample(out1,weights['w51'],biases['b51'],weights['w52'],biases['b52'],
    pool=False)
    up1       = up_sample(out1,weights['wu1'],biases['bu1'])
    up1       = up_sample(up1,weights['wu2'],biases['bu2'])
    up1       = up_sample(up1,weights['wu3'],biases['bu3'])
    up1       = up_sample(up1,weights['wu4'],biases['bu4'])
    out       = tf.nn.conv2d(up1,weights['wf'],strides=[1,1,1,1],padding='SAME')
    out       = tf.nn.bias_add(out,biases['bf'])
    return out

## Create generators for pre-processing the images and making a batch available at runtime
## instead of loading all the images and labels in memory
# set the necessary directories
data_dir = "/home/santanu/Downloads/Carvana/train/"  # Contains the input training data
mask_dir = "/home/santanu/Downloads/Carvana/train_masks/" # Contains the grouth truth labels
all_images = os.listdir(data_dir)
# pick which images we will use for testing and which for validation
train_images, validation_images = train_test_split(all_images, train_size=0.8, test_
size=0.2)
# utility function to convert grayscale images to rgb
def grey2rgb(img):
    new_img = []
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            new_img.append(list(img[i][j])*3)
    new_img = np.array(new_img).reshape(img.shape[0], img.shape[1], 3)
    return new_img

# generator that we will use to read the data from the directory
def data_gen_small(data_dir, mask_dir, images, batch_size, dims):
        """
        data_dir: where the actual images are kept
        mask_dir: where the actual masks are kept
        images: the filenames of the images we want to generate batches from
        batch_size: self explanatory
        dims: the dimensions in which we want to rescale our images
        """
        while True:
            ix = np.random.choice(np.arange(len(images)), batch_size)
            imgs = []
            labels = []
            for i in ix:
                # images
                original_img = load_img(data_dir + images[i])
                resized_img = imresize(original_img, dims+[3])
                array_img = img_to_array(resized_img)/255
                imgs.append(array_img)

                # masks
                original_mask = load_img(mask_dir + images[i].split(".")[0] + '_mask.gif')
                resized_mask = imresize(original_mask, dims+[3])
```

```
                array_mask = img_to_array(resized_mask)/255
                labels.append(array_mask[:, :, 0])
            imgs = np.array(imgs)
            labels = np.array(labels)
            yield imgs, labels.reshape(-1, dims[0], dims[1], 1)

train_gen = data_gen_small(data_dir, mask_dir, train_images,32, [128, 128])
validation_gen = data_gen_small(data_dir, mask_dir, validation_images,32, [128, 128])

display_step=10
learning_rate=0.0001

keep_prob = tf.placeholder(tf.float32)
logits = unet_basic(x,weights,biases)
flat_logits = tf.reshape(tensor=logits, shape=(-1, 1))
flat_labels = tf.reshape(tensor=y,shape=(-1, 1))
cross_entropies = tf.nn.sigmoid_cross_entropy_with_logits(logits=flat_logits,
                                                          labels=flat_labels)
cost = tf.reduce_mean(cross_entropies)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Evaluate model

## initializing all variables

init = tf.global_variables_initializer()

## Launch the execution Graph

with tf.Session() as sess:
    sess.run(init)
    for batch in xrange(500):
        batch_x,batch_y= next(train_gen)
        sess.run(optimizer, feed_dict={x:batch_x,y:batch_y})
        loss = sess.run([cost],feed_dict={x:batch_x,y:batch_y})
        ## Validation loss and store the result for display at the end
        val_x,val_y = next(validation_gen)
        loss_val = sess.run([cost],feed_dict={x:val_x,y:val_y})
        out_x = sess.run(logits,feed_dict={x:val_x})
        print('batch:',batch,'train loss:',loss,'validation loss:',loss_val)

## To judge the segmentation quality of the Model we plot the segmentation for couple of the
validation images.
## These validation images were evaluated in the last batch of training. Bear in mind that
the model hasn't been
## trained on these validation images.

img = (out_x[1] > 0.5)*1.0
plt.imshow(grey2rgb(img),alpha=0.5)
plt.imshow(val_x[1])
plt.imshow(grey2rgb(val_y[1]), alpha=0.5)
```

```
img = (out_x[2] > 0.5)*1.0
plt.imshow(grey2rgb(img),alpha=0.5)
plt.imshow(val_x[2])
plt.imshow(grey2rgb(val_y[2]), alpha=0.5)

-output-

''batch:', 400, 'train loss:', [0.044453222], 'validation loss:', [0.058442257])
('batch:', 401, 'train loss:', [0.049510699], 'validation loss:', [0.055530164])
('batch:', 402, 'train loss:', [0.048047166], 'validation loss:', [0.055518236])
('batch:', 403, 'train loss:', [0.049462996], 'validation loss:', [0.049190756])
('batch:', 404, 'train loss:', [0.047011156], 'validation loss:', [0.051120583])
('batch:', 405, 'train loss:', [0.046235155], 'validation loss:', [0.052921098])
('batch:', 406, 'train loss:', [0.051339123], 'validation loss:', [0.054767497])
('batch:', 407, 'train loss:', [0.050004266], 'validation loss:', [0.052718181])
('batch:', 408, 'train loss:', [0.048425209], 'validation loss:', [0.054115709])
('batch:', 409, 'train loss:', [0.05234601],  'validation loss:', [0.053246532])
('batch:', 410, 'train loss:', [0.054224499], 'validation loss:', [0.05121265])
('batch:', 411, 'train loss:', [0.050268434], 'validation loss:', [0.056970511])
('batch:', 412, 'train loss:', [0.046658799], 'validation loss:', [0.058863375])
('batch:', 413, 'train loss:', [0.048009872], 'validation loss:', [0.049314644])
('batch:', 414, 'train loss:', [0.053399611], 'validation loss:', [0.050949663])
('batch:', 415, 'train loss:', [0.047932044], 'validation loss:', [0.049477436])
('batch:', 416, 'train loss:', [0.054921247], 'validation loss:', [0.059221379])
('batch:', 417, 'train loss:', [0.053222295], 'validation loss:', [0.061699588])
('batch:', 418, 'train loss:', [0.047465689], 'validation loss:', [0.051628478])
('batch:', 419, 'train loss:', [0.055220582], 'validation loss:', [0.056656662])
('batch:', 420, 'train loss:', [0.052862987], 'validation loss:', [0.048487194])
('batch:', 421, 'train loss:', [0.052869596], 'validation loss:', [0.049040388])
('batch:', 422, 'train loss:', [0.050372943], 'validation loss:', [0.052676879])
('batch:', 423, 'train loss:', [0.048104074], 'validation loss:', [0.05687784])
('batch:', 424, 'train loss:', [0.050506901], 'validation loss:', [0.055646997])
('batch:', 425, 'train loss:', [0.042940177], 'validation loss:', [0.047789834])
('batch:', 426, 'train loss:', [0.04780338],  'validation loss:', [0.05711592])
('batch:', 427, 'train loss:', [0.051617432], 'validation loss:', [0.051806655])
('batch:', 428, 'train loss:', [0.047577277], 'validation loss:', [0.052631289])
('batch:', 429, 'train loss:', [0.048690431], 'validation loss:', [0.044696849])
('batch:', 430, 'train loss:', [0.046005826], 'validation loss:', [0.050702494])
('batch:', 431, 'train loss:', [0.05022176],  'validation loss:', [0.053923506])
('batch:', 432, 'train loss:', [0.041961089], 'validation loss:', [0.047880188])
('batch:', 433, 'train loss:', [0.05004932],  'validation loss:', [0.057072558])
('batch:', 434, 'train loss:', [0.04603707],  'validation loss:', [0.049482994])
('batch:', 435, 'train loss:', [0.047554974], 'validation loss:', [0.050586618])
('batch:', 436, 'train loss:', [0.046048313], 'validation loss:', [0.047748547])
('batch:', 437, 'train loss:', [0.047006462], 'validation loss:', [0.059268739])
('batch:', 438, 'train loss:', [0.045432612], 'validation loss:', [0.051733252])
('batch:', 439, 'train loss:', [0.048241541], 'validation loss:', [0.04774794])
('batch:', 440, 'train loss:', [0.046124499], 'validation loss:', [0.048809234])
('batch:', 441, 'train loss:', [0.049743906], 'validation loss:', [0.051254783])
('batch:', 442, 'train loss:', [0.047674596], 'validation loss:', [0.048125759])
('batch:', 443, 'train loss:', [0.048984651], 'validation loss:', [0.04512443])
```

```
('batch:', 444, 'train loss:', [0.045365792], 'validation loss:', [0.042732101])
('batch:', 445, 'train loss:', [0.046680171], 'validation loss:', [0.050935686])
('batch:', 446, 'train loss:', [0.04224021],  'validation loss:', [0.052455597])
('batch:', 447, 'train loss:', [0.045161027], 'validation loss:', [0.045499101])
('batch:', 448, 'train loss:', [0.042469904], 'validation loss:', [0.050128322])
('batch:', 449, 'train loss:', [0.047899902], 'validation loss:', [0.050441738])
('batch:', 450, 'train loss:', [0.043648213], 'validation loss:', [0.048811793])
('batch:', 451, 'train loss:', [0.042413067], 'validation loss:', [0.051744446])
('batch:', 452, 'train loss:', [0.047555752], 'validation loss:', [0.04977461])
('batch:', 453, 'train loss:', [0.045962822], 'validation loss:', [0.047307629])
('batch:', 454, 'train loss:', [0.050115541], 'validation loss:', [0.050558448])
('batch:', 455, 'train loss:', [0.045722887], 'validation loss:', [0.049715079])
('batch:', 456, 'train loss:', [0.042583987], 'validation loss:', [0.048713747])
('batch:', 457, 'train loss:', [0.040946022], 'validation loss:', [0.045165032])
('batch:', 458, 'train loss:', [0.045971408], 'validation loss:', [0.046652604])
('batch:', 459, 'train loss:', [0.045015588], 'validation loss:', [0.055410333])
('batch:', 460, 'train loss:', [0.045542594], 'validation loss:', [0.047741935])
('batch:', 461, 'train loss:', [0.04639449],  'validation loss:', [0.046171311])
('batch:', 462, 'train loss:', [0.047501944], 'validation loss:', [0.046123035])
('batch:', 463, 'train loss:', [0.043643478], 'validation loss:', [0.050230302])
('batch:', 464, 'train loss:', [0.040434662], 'validation loss:', [0.046641909])
('batch:', 465, 'train loss:', [0.046465941], 'validation loss:', [0.054901786])
('batch:', 466, 'train loss:', [0.049838047], 'validation loss:', [0.048461676])
('batch:', 467, 'train loss:', [0.043582849], 'validation loss:', [0.052996978])
('batch:', 468, 'train loss:', [0.050299261], 'validation loss:', [0.048585847])
('batch:', 469, 'train loss:', [0.046049926], 'validation loss:', [0.047540378])
('batch:', 470, 'train loss:', [0.042139661], 'validation loss:', [0.047782935])
('batch:', 471, 'train loss:', [0.046433724], 'validation loss:', [0.049313426])
('batch:', 472, 'train loss:', [0.047063917], 'validation loss:', [0.045388222])
('batch:', 473, 'train loss:', [0.045556825], 'validation loss:', [0.044953942])
('batch:', 474, 'train loss:', [0.046181824], 'validation loss:', [0.045763671])
('batch:', 475, 'train loss:', [0.047123503], 'validation loss:', [0.047637179])
('batch:', 476, 'train loss:', [0.046167117], 'validation loss:', [0.051462833])
('batch:', 477, 'train loss:', [0.043556783], 'validation loss:', [0.044357236])
('batch:', 478, 'train loss:', [0.04773742],  'validation loss:', [0.046332739])
('batch:', 479, 'train loss:', [0.04820114],  'validation loss:', [0.045707334])
('batch:', 480, 'train loss:', [0.048089884], 'validation loss:', [0.052449297])
('batch:', 481, 'train loss:', [0.041174423], 'validation loss:', [0.050378591])
('batch:', 482, 'train loss:', [0.049479648], 'validation loss:', [0.047861829])
('batch:', 483, 'train loss:', [0.041197944], 'validation loss:', [0.051383432])
('batch:', 484, 'train loss:', [0.051363751], 'validation loss:', [0.050520841])
('batch:', 485, 'train loss:', [0.047751397], 'validation loss:', [0.046632469])
('batch:', 486, 'train loss:', [0.049832929], 'validation loss:', [0.048640732])
('batch:', 487, 'train loss:', [0.049518026], 'validation loss:', [0.048658002])
('batch:', 488, 'train loss:', [0.051349726], 'validation loss:', [0.051405452])
('batch:', 489, 'train loss:', [0.041912809], 'validation loss:', [0.046458714])
('batch:', 490, 'train loss:', [0.047130216], 'validation loss:', [0.052001398])
('batch:', 491, 'train loss:', [0.041481428], 'validation loss:', [0.046243563])
('batch:', 492, 'train loss:', [0.042776003], 'validation loss:', [0.042228915])
('batch:', 493, 'train loss:', [0.043606419], 'validation loss:', [0.048132997])
('batch:', 494, 'train loss:', [0.047129884], 'validation loss:', [0.046108384])
```

```
('batch:', 495, 'train loss:', [0.043634158], 'validation loss:', [0.046292961])
('batch:', 496, 'train loss:', [0.04454672],  'validation loss:', [0.044108659])
('batch:', 497, 'train loss:', [0.048068151], 'validation loss:', [0.044547819])
('batch:', 498, 'train loss:', [0.044967934], 'validation loss:', [0.047069982])
('batch:', 499, 'train loss:', [0.041554678], 'validation loss:', [0.051807735])
```

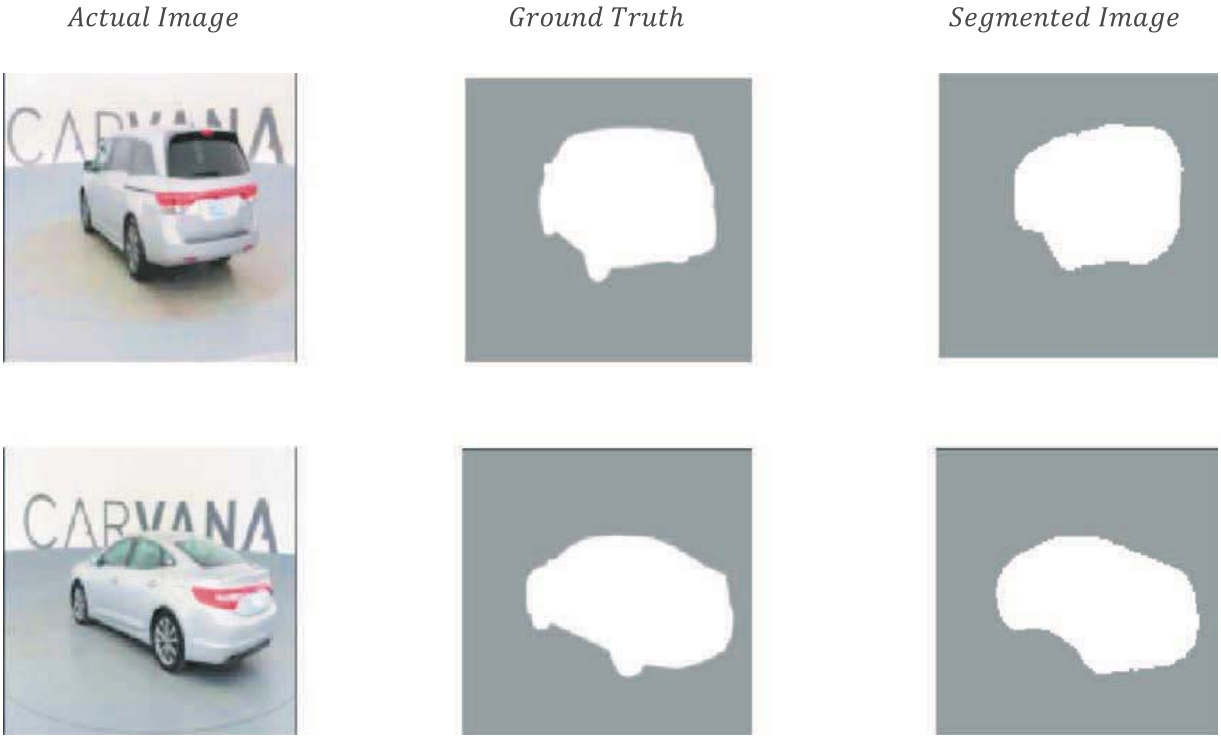


Figure 6-15a. *Segmentation results on the validation dataset with model trained on* 128×128 *–size images*

The average training loss and the validation loss are almost the same, which indicates that the model is not overfitting and is generalizing well. As we can see from Figure 6-15a, the results of segmentation look convincing based on the provided ground truths. The spatial dimensions of the images used for this network are 128×128 . On increasing the spatial dimensions of the input images to 512×512 , the accuracy and segmentation increase significantly. Since it's a fully convolutional network with no fully connected layers, very few changes in the network are required to handle the new image size. The output of segmentation for a couple of validation dataset images are presented in Figure 6-15b to illustrate the fact that bigger image sizes are most of the time beneficial for image segmentation problems since it helps capture more context.

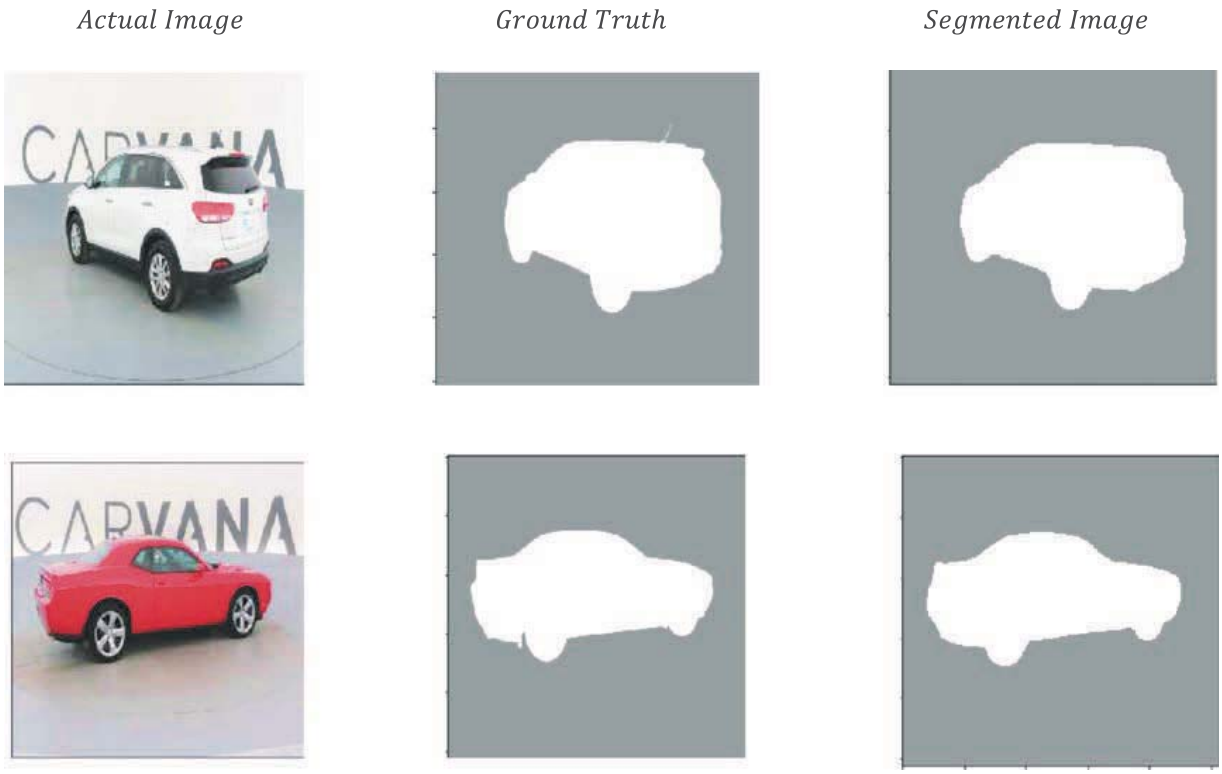


Figure 6-15b. *Segmentation results from the validation dataset with model trained on* 512×512 *–size images*

Image Classification and Localization Network

All the classification models predict the class of the object in the image but don't really tell us the location of the object. A bounding box can be used to represent the location of the object in the image. If the images are annotated with bounding boxes and information about them is available with the output class, it would be possible to train the model to predict these bounding boxes along with the class of the object. These bounding boxes can be represented by four numbers, two corresponding to the spatial coordinates of the leftmost upper part of the bounding box and the other two to denote the height and width of the bounding box. Then, these four numbers can be predicted by regression. One can use one convolutional neural network for classification and another for predicting these bounding-box attributes through regression. However, generally the same convolutional neural network is used for predicting the class of the object as well as for predicting the bounding-box location. The CNN up to the last fully connected dense layers would be the same, but in the output, along with the different classes for objects, there would be four extra units corresponding to the bounding-box attributes. This technique of predicting bounding boxes around objects within an image is known as *localization*. Illustrated in Figure 6-16 is an image classification and localization network pertaining to images of dogs and cats. The *Apriori* assumption in this type of neural network is that there is only one class object in an image.

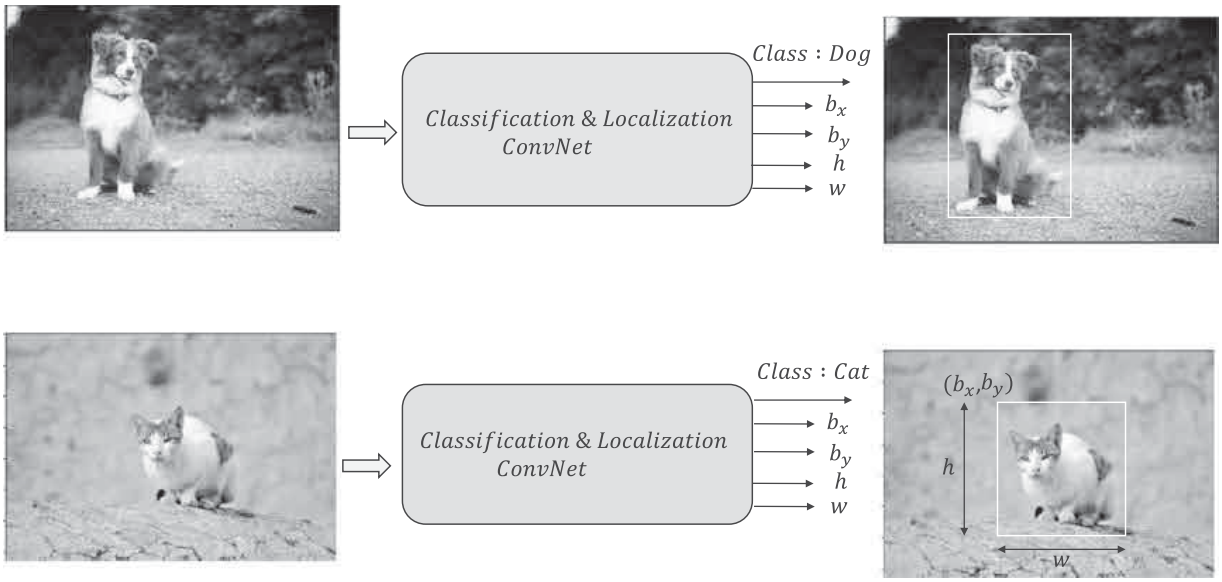


Figure 6-16. *Classification and localization network*

The cost function for this network would be a combination of classification loss/cost over the different object classes and the regression cost associated with the prediction of the bounding-box attributes. Since the cost to optimize is a multi-task objective function, one needs to determine how much weight to assign to each task. This is important since the different costs associated with these tasks—let's say categorical cross-entropy for classification and RMSE for regression—would have different scales and hence could drive the optimization haywire if the costs are not weighed properly to form the total cost. The costs need to be normalized to a common scale and then assigned weights based on the complexity of the tasks.
Let the parameters of the convolutional neural network that deals with n classes and a bounding box determined by the four numbers be represented by θ. Let the output class be represented by the vector $y=[y_1 y_2 .. y_n]^T \in \{0,1\}^{n\times 1}$ since each of the $y_j \in \{0,1\}$. Also, let the bounding-box numbers be represented by the vector $s=[s_1\ s_2\ s_3\ s_4]^T$ where s_1 and s_2 denote bounding-box coordinates for the upper leftmost pixel while s_3 and s_4 denote the height and width of the bounding box. If the predicted probabilities of the classes are represented by $p=[p_1 p_2 .. p_n]^T$ while the predicted bounding-box attributes are represented by $t=[t_1 t_2 t_3 t_4]^T$, then the loss or cost function associated with an image can be expressed as

$$c(\theta)=-\alpha\sum_{j=1}^{n} y_j \log p_j + \beta\sum_{j=1}^{4}\left(s_j - t_j\right)^2$$

The first term in the preceding expression denotes categorical cross-entropy for the SoftMax over the n classes while the second term is the regression cost associated with predicting the bounding-box attributes. The parameters α and β are the hyper-parameters of the network and should be fine-tuned for obtaining reasonable results. For a mini batch over m data points the cost function can be expressed as follows:

$$C(\theta)=\frac{1}{m}\left[-\alpha\sum_{i=1}^{m}\sum_{j=1}^{n} \mathrm{y}_j^{(i)} \log \mathrm{p}_j^{(i)} + \beta\sum_{i=1}^{m}\sum_{j=1}^{4}\left(\mathrm{s}_j^{(i)} - \mathrm{t}_j^{(i)}\right)^2\right]$$

where the suffix over *i* represents different images. The preceding cost function can be minimized through gradient descent. Just as an aside, when comparing the performance of different versions of this network with different hyper-parameter values for (α, β), one should not compare the cost associated with these networks as criteria for selecting the best network. Rather, one should use some other metrics such as precision, recall, F1-score, area under the curve, and so on for the classification task and metrics such as overlap area of the predicted and the ground truth bounding boxes,and so on for the localization task.

Object Detection

An image in general doesn't contain one object but several objects of interest. There are a lot of applications that benefit from being able to detect multiple objects in images. For example, object detection can be used to count the number of people in several areas of a store for crowd analytics. Also, at an instant the traffic load on a signal can be detected by getting a rough estimate of the number of cars passing through the signal. Another area in which object detection is being leveraged is in the automated supervision of industrial plants to detect events and generate alarms in case there has been a safety violation. Continuous images can be captured in critical areas of the plant that are hazardous and critical events can be captured from those images based on multiple objects detected within the image. For instance, if a worker is working with machinery that requires him to wear safety gloves, eye glasses, and helmet, a safety violation can be captured based on whether the objects mentioned were detected in the image or not.

The task of detecting multiple objects in images is a classical problem in computer vision. To begin with, we cannot use the classification and localization network or any variants of it since images can have varying numbers of objects within them. To get ourselves motivated toward solving the problem of object detection, let's get started with a very naïve approach. We can randomly take image patches from the existing image by a brute-force sliding-window technique and then feed it to a pre-trained object classification and localization network. Illustrated in Figure 6-17 is a sliding-window approach to detecting multiple objects within an image.

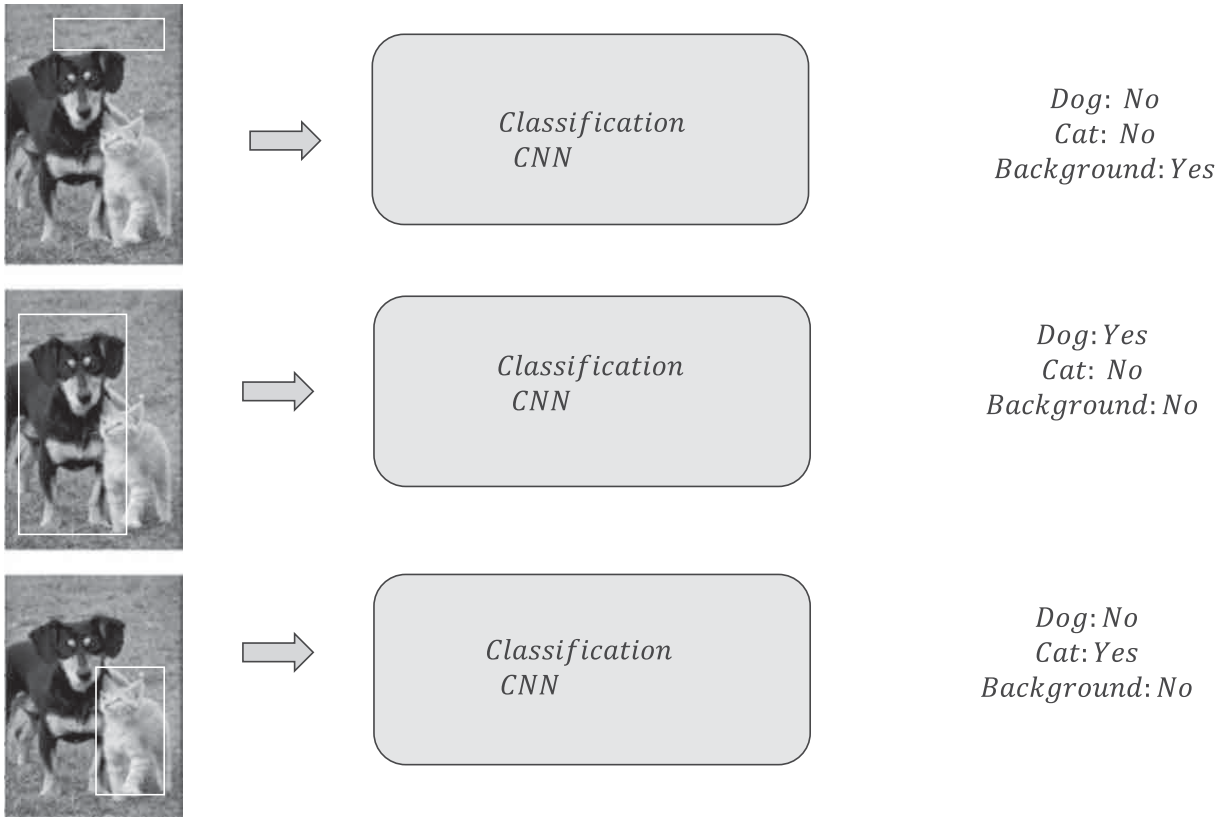


Figure 6-17. *Sliding-window technique to object detection*

Although this method would work, it would be computationally very expensive, or rather computationally intractable, since one would have to try thousands of image patches at different locations and scales in the absence of good region proposals. The current advanced methods in object detection propose several regions in which objects can be located and then feeds those image-proposal regions to the classification and localization network. One such object-detection technique is called R-CNN, which we will discuss next.

R-CNN

In an R-CNN, *R* stands for *region* proposals. The region proposals are usually derived in through an algorithm called *selective search*. A selective search on an image generally provides around 2000 region proposals of interest. Selective search usually utilizes traditional image-processing techniques to locate blobby regions in an image as prospective areas likely to contain objects. The following are the processing steps for selective search on a broad level:

- Generate many regions within the image, each of which can belong to only one class.
- Recursively combine smaller regions into larger ones through a greedy approach. At each step, the two regions merged should be most similar. This process needs to be repeated until only one region remains. This process yields a hierarchy of successively larger regions and allows the algorithm to propose a wide variety of likely regions for object detection. These generated regions are used as the candidate region proposals.

These 2000 regions of interest are then fed to the classification and localization network to predict the class of the object along with associated bounding boxes. The classification network is a convolutional neural network followed by a support-vector machine for the final classification. Illustrated in Figure 6-18 is a high-level architecture for an R-CNN.

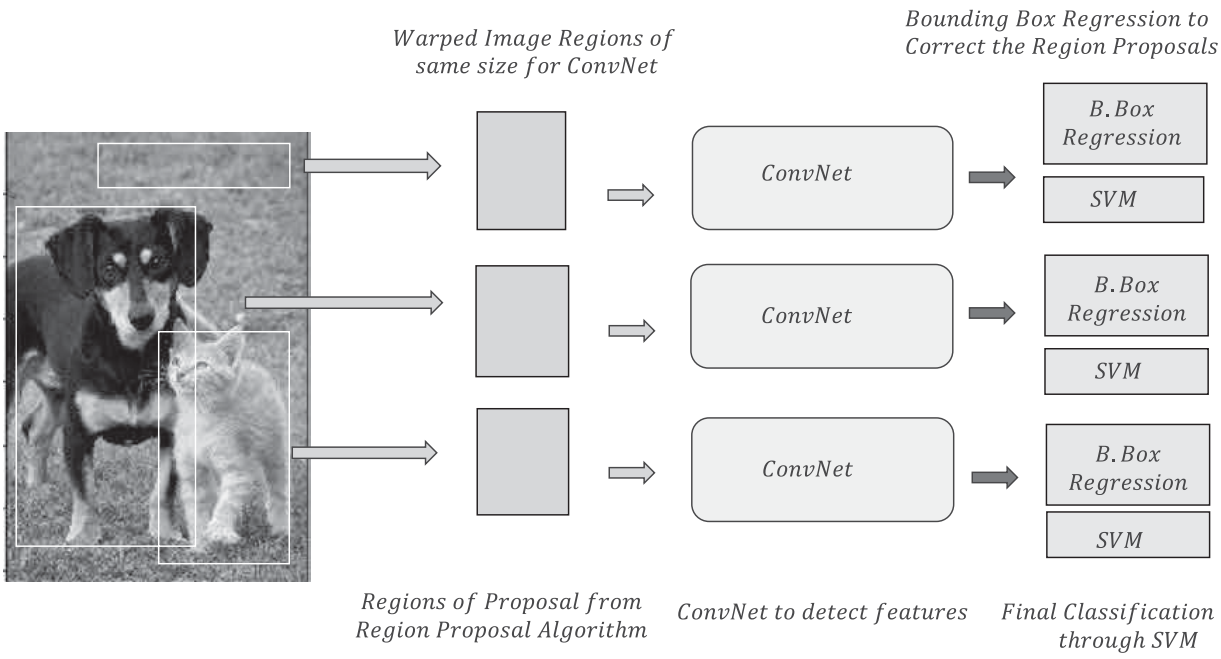


Figure 6-18. *R-CNN network*

The following are the high-level steps associated with training an R-CNN:

- Take a pre-trained ImageNet CNN such as AlexNet and retrain the last fully connected layer with the objects that need to be detected, along with backgrounds.
- Get all the region proposals per image (2000 per image as per selective search), warp or resize them to match the CNN input size, process them through CNN, and then save the features on disk for further processing. Generally, the pooling layer output maps are saved as features to disk.
- Train SVM to classify either object or background based on the features from CNN. For each class of objects there should be one SVM that learns to distinguish between the specific object and background.
- Finally, bounding-box regression is done to correct the region proposals.

Although the R-CNN does a good job in at object detection, the following are some of its drawbacks:

- One of the problems with R-CNN is the huge number of proposals, which makes the network very slow since each of these 2000 proposals would have independent flows through convolution neural networks. Also, the region proposals are fixed; the R-CNN is not learning them.
- The localization and bounding boxes predicted are from separate models and hence during model training we are not learning anything specific to localization of the objects based on the training data.
- For the classification task, the features generated out of the convolutional neural network are used to fine-tune SVMs, leading to a higher processing cost.

Fast and Faster R-CNN

Fast R-CNN overcomes some of the computational challenges of R-CNN by having a common convolution path for the whole image up to a certain number of layers, at which point the region proposals are projected to the output feature maps and relevant regions are extracted for further processing through fully connected layers and then the final classification. The extraction of relevant region proposals from the output feature maps from convolution and resizing of them to a fixed size for the fully connected layer is done through a pooling operation known as ROI pooling. Illustrated in Figure 6-19 is an architecture diagram for Fast R-CNN.

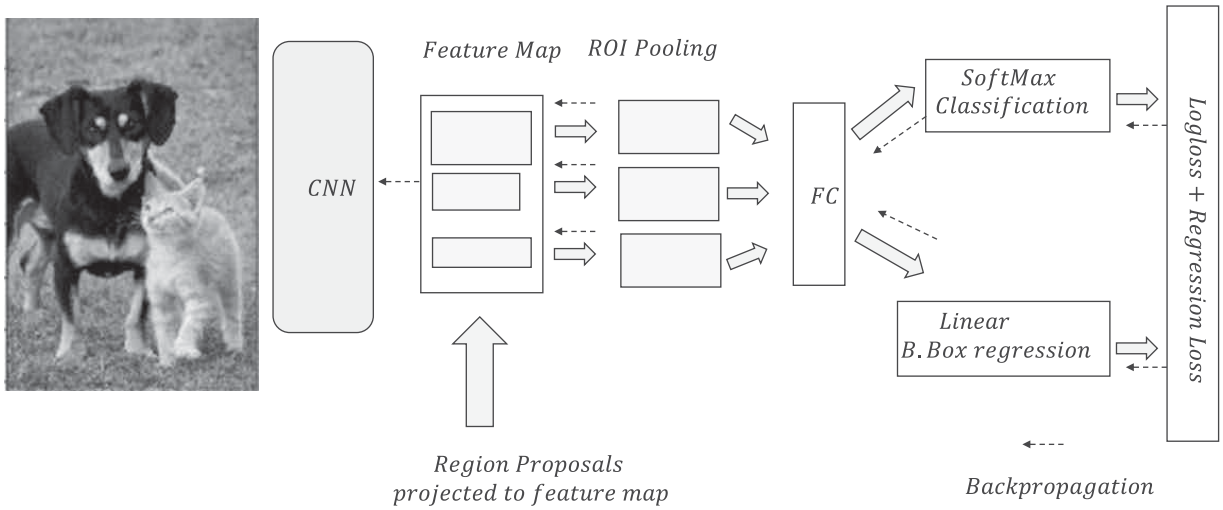


Figure 6-19. *Fast R-CNN schematic diagram*

Fast R-CNN saves a lot of costs associated with multiple convolution operations (2000 per image per selective search) in R-CNN. However, the region proposals are still dependent on the external region-proposal algorithms such as selective search. Because of this dependency on external region-proposal algorithms, Fast R-CNN is bottlenecked by the computation of these region proposals. The network must wait for these external proposals to be made before it can move forward. These bottleneck issues are eliminated by Faster R-CNN, where the region proposals are done within the network itself instead of depending on external algorithms. The architecture diagram for Faster R-CNN is almost like that of Fast R-CNN, but with a new addition—a region-proposal network that eliminates the dependency on an external region-proposal scheme such as selective search.

Generative Adversarial Networks

Generative adversarial networks, or GANs, are one of the biggest advances in deep learning in recent times. Ian Goodfellow and colleagues first introduced this network in 2014 in an NIPS paper titled "Generative Adversarial Networks." The paper can be located at https://arxiv.org/abs/1406.2661. Since then, there has been a lot of interest and development in generative adversarial networks. In fact, Yann LeCun, one of the most prominent deep-learning experts, considers the introduction of generative adversarial networks to be the most important breakthrough in deep learning in recent times. GANs are used as generative models for producing synthetic data like the data produced by a given distribution. GAN has usages and potential in several fields, such as image generation, image inpainting, abstract reasoning, semantic segmentation, video generation, style transfer from one domain to another, and text-to-image generation applications, among others.

Generative adversarial networks are based on the two agents zero-sum game from game theory. A generative adversarial network has two neural networks, the generator (G) and the discriminator (D), competing against each other. The generator (G) tries to fool the discriminator (D) such that the discriminator is not able to distinguish between real data from a distribution and the fake data generated by the generator (G). Similarly, the discriminator (D) learns to distinguish the real data from the fake data generated by the generator (G). Over a certain period, both the discriminator and the generator improve on their own tasks while competing with each other. The optimal solution to this game-theory problem is given by the Nash equilibrium wherein the generator learns to produce fake data that has a distribution the same as that of the original data distribution, and at the same time the discriminator outputs $\frac{1}{2}$ probability for both real and fake data points.

Now, the most obvious question is how are the fake data constructed. The fake data is constructed through the generative neural network model (G) by sampling noise z from a prior distribution P_z. If the actual data x follows distribution P_x and the fake data $G(z)$ generated by the generator follows distribution P_g, then at equilibrium $P_x(x)$ should equal $P_g(G(z))$; i.e.,

$$P_g\left(G\left(z\right)\right) \sim P_x\left(x\right)$$

Since at equilibrium the distribution of the fake data would be almost the same as the real data distribution, the generator would learn to sample fake data that would be hard to distinguish from the real data. Also, at equilibrium the discriminator D should output $\frac{1}{2}$ as the probability for both classes—the real data and the fake data. Before we go through the math for a generative adversarial network, it is worthwhile to gain some understanding about the zero-sum game, Nash equilibrium, and Minimax formulation.

Illustrated in Figure 6-20 is a generative adversarial network in which there are two neural networks, the generator (G) and the discriminator (D), that compete against each other.