Image Segmentation

Topics Covered

1. Image Segmentation Introduction
2. Encoder Decoder Approach for Image Segmentation
3. Working With Unet – Encoder Decoder Approach
4. Implementation with Python
5. Summary

Image segmentation introduction

If we have the question of finding the multiple objects in the same image and their location but instead, we want the exact location of the objects.

We can give the exact location by identifying every pixel that represents the object. In simple words, instead of drawing a rough rectangular box around the object, we draw a polygon around the object and also color every pixel of that object as can be seen here. This can be formulated as an Image Segmentation problem.

Q. What are the objects present in the image? Where are they exactly?

There is a **dog** and a **cat**!

The dog → coloured as **blue**
The cat → coloured as **red**

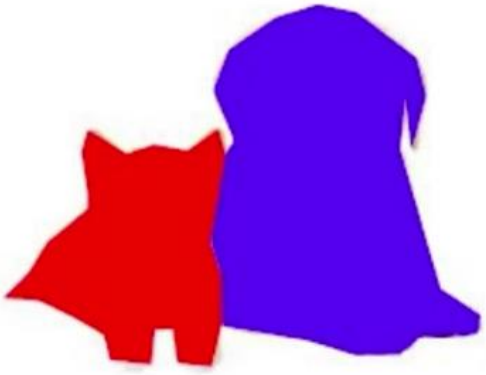


Image segmentation is the task of partitioning an image based on the objects present and their semantic importance.

This makes it a whole lot easier to analyze the given image, because instead of getting an approximate location from a rectangular box. We can get the exact pixel-wise location of the objects.

Applications of image segmentation

Now there are multiple applications where image segmentation can be applied, Let's discuss a few interesting image segmentation applications.

Medical Imaging

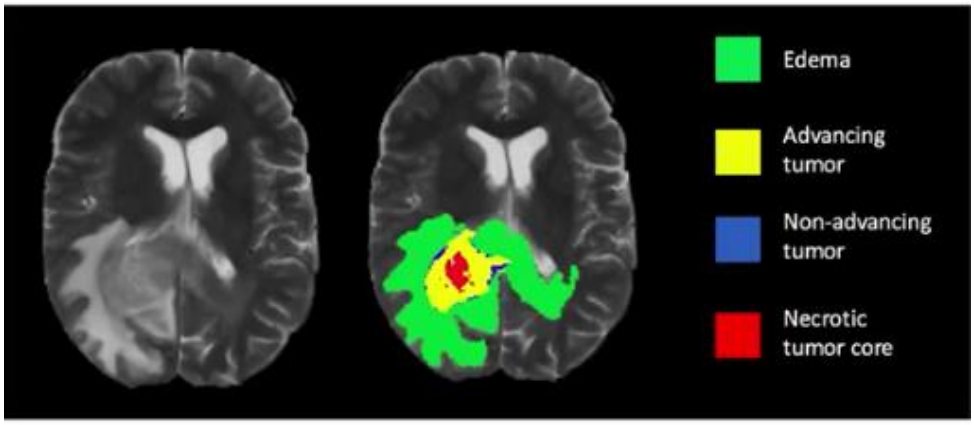


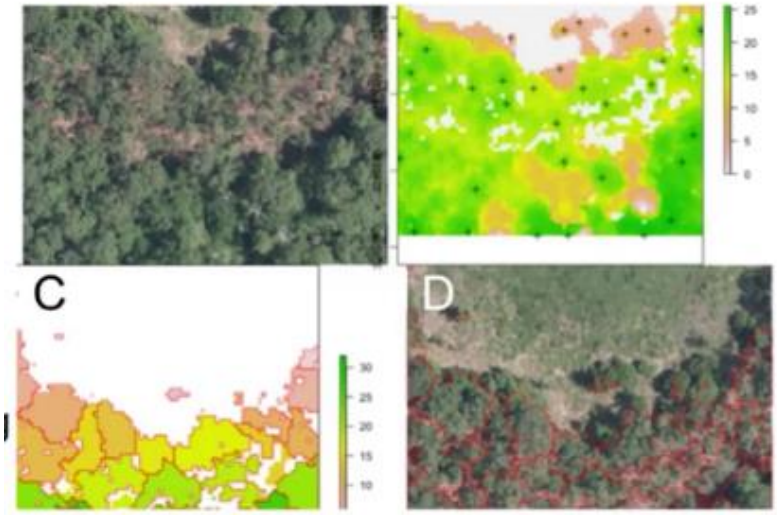
Image segmentation can be applied for medical imaging tasks such as cancer cell segmentation. Where it is of utmost importance that we identify the exact location of the tumours or cancerous cells.

Self-driving cars

Source: Marius Cordts et al: "The Cityscapes Dataset for Semantic Urban Scene Understanding", 2016

Other applications of image segmentation could be self-driving systems. for lane segmentation or pedestrian identification. By precisely predicting the location of the object of importance like roads or a person, a self-driving system can take appropriate steps to handle the downstream task like applying the breaks or slowing down the car.

Satellite Imaging/ Remote sensing



Source: McMahon CA. 2019. Remote sensing pipeline for tree segmentation and classification in a mixed softwood and hardwood system

Another application could be in the remote sensing domain. Where we can identify the composition of the ground such as what is the forest cover in the region or finding illegal activities like mining or even forest fires.

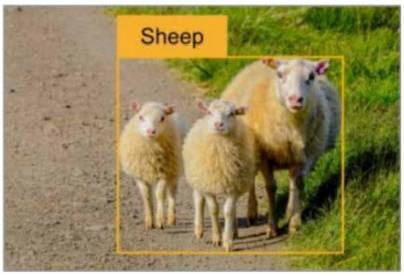
Types of Image Segmentation

Now even in Image segmentation, there are a few types of problems that you should get yourself acquainted with. First and foremost is the semantic segmentation then coming instance segmentation. The next one is panoptic segmentation. Let's see each of them.

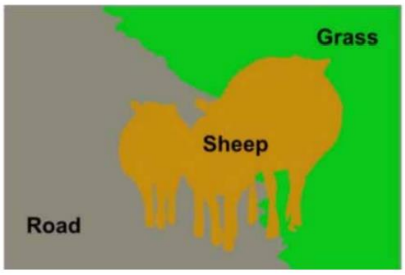
Semantic segmentation

Semantic segmentation describes the process of associating each pixel with a class label. So simply, here we just care about a coarse representation of all the objects present in the image. Here you can see all the cars represented in blue, the pedestrian with red, and the street as slightly pink.

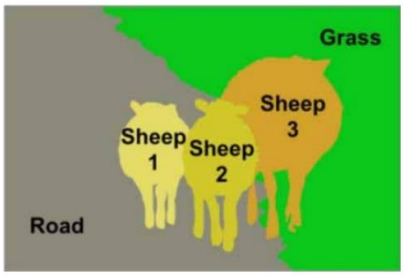
So there is no clear distinction between the cars, which means all the cars are coloured blue. This is the simplest way to define an image segmentation problem.



Classification and Localization



Segmantic segmentation



Instance segmentation

Instance segmentation

Now If you want to go a bit further and want to represent each instance of a class differently, this problem will be formally known as instance segmentation.

Unlike semantic segmentation, in image segmentation, we mask each instance of an object contained in an image independently. So this implies, that we will focus on the object of importance first and then identify each instance of the object separately.

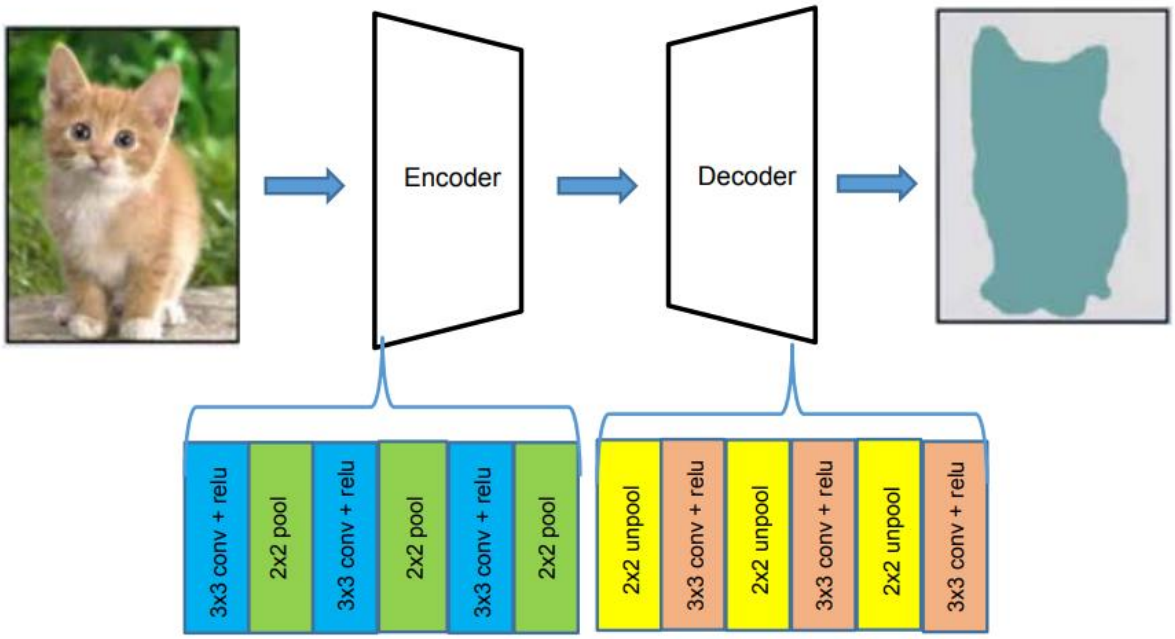
You can see that all objects in the image cars and persons are highlighted and all of these are different colors. This is what instance segmentation is.

Encoder Decoder approach for Image segmentation

A general semantic segmentation architecture can be broadly thought of as an **encoder** network followed by a **decoder** network:

- The encoder is usually is a pre-trained classification network like VGG/ResNet followed by a decoder network.
- The task of the decoder is to semantically project the discriminative features (lower resolution) learnt by the encoder onto the pixel space (higher resolution) to get a dense classification.

Semantic segmentation not only requires discrimination at pixel level but also a mechanism to project the discriminative features learnt at different stages of the encoder onto the pixel space. Different approaches employ different mechanisms as a part of the decoding mechanism.

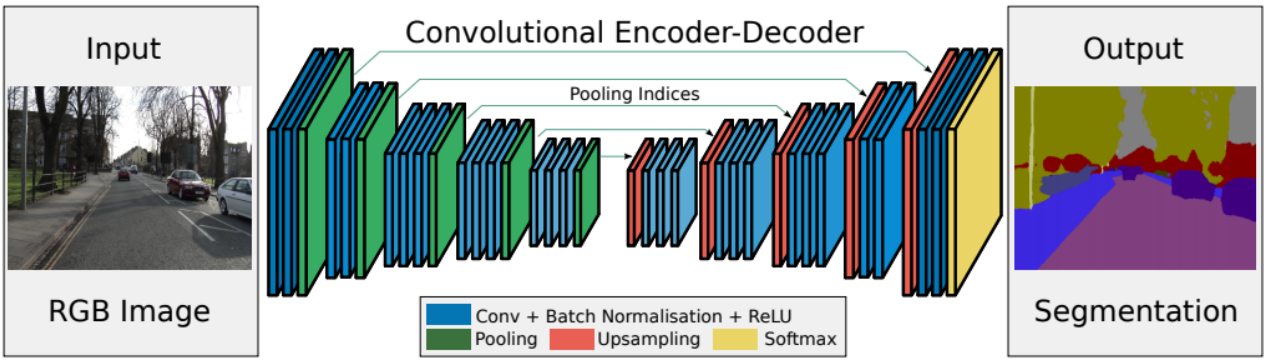


Deep Understanding of SegNet Model for Semantic Segmentation

SegNet, is designed to be an efficient architecture for pixel-wise semantic segmentation. It is primarily motivated by road scene understanding applications which require the ability to model appearance (road, building), shape (cars,pedestrians) and understand the spatial-relationship (context) between different classes such as road and side-walk. In typical road scenes, the majority of the pixels belong to large classes such as road, building and hence the network must produce smooth segmentation.

Architecture of SegNet

There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification



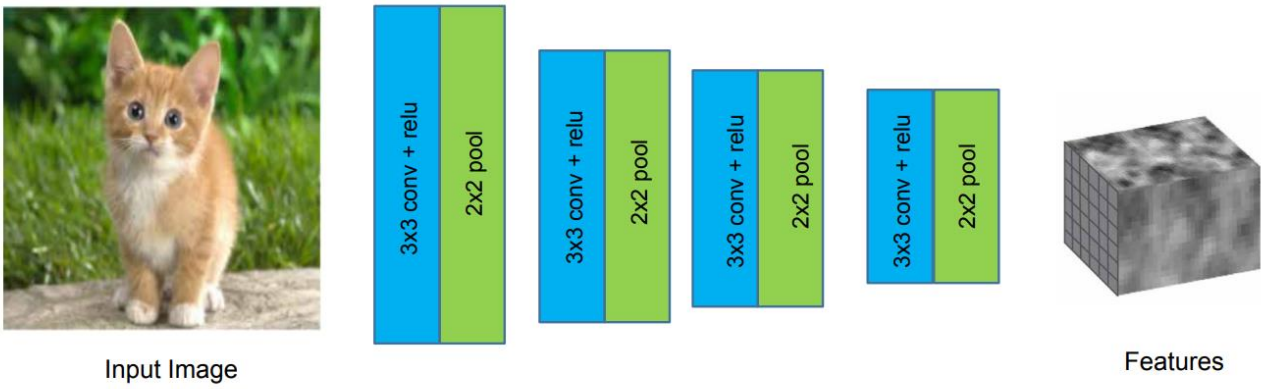
SegNet has an encoder network and a corresponding decoder network, followed by a final pixelwise classification layer.

This architecture is illustrated in above figure. The encoder network consists of 13 convolutional layers which correspond to the first 13 convolutional layers in the VGG16 network designed for object classification. They discard the fully connected layers in favour of retaining higher resolution feature maps at the deepest encoder output.

This also reduces the number of parameters in the SegNet encoder network significantly (from 134M to 14.7M) as compared to other recent architectures. Each encoder layer has a corresponding decoder layer and hence the decoder network has 13 layers. The final decoder output is fed to a multi-class soft-max classifier to produce class probabilities for each pixel independently.

Encoder network

It performs convolution with a filter bank to produce a set of feature maps. These are then batch normalized . Then an element-wise rectified linear non-linearity (ReLU) max(0, x) is applied. Following that, max-pooling with a 2 × 2 window and stride 2 (non-overlapping window) is performed and the resulting output is sub-sampled by a factor of 2.Max-pooling is used to achieve translation invariance over small spatial shifts in the input image. Sub-sampling results in a large input image context (spatial window) for each pixel in the feature map.

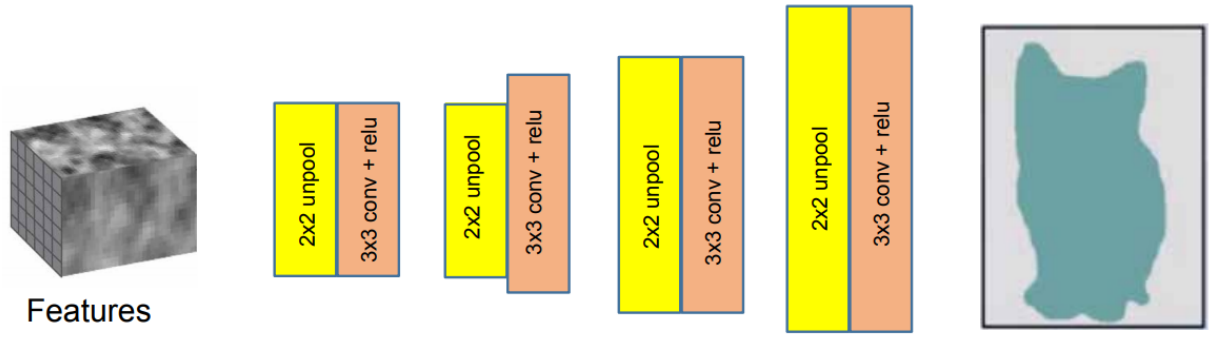


While several layers of max-pooling and sub-sampling can achieve more translation in-variance for robust classification correspondingly there is a loss of spatial resolution of the feature maps. The increasingly lossy (boundary detail) image representation is not beneficial for segmentation where boundary delineation is vital.

Therefore, it is necessary to capture and store boundary information in the encoder feature maps before sub-sampling is performed. If memory during inference is not constrained, then all the encoder feature maps (after sub sampling) can be stored. This is usually not the case in practical applications and hence we propose a more efficient way to store this information. It involves storing only the max-pooling indices, i.e, the locations of the maximum feature value in each pooling window is memorized for each encoder feature map. In principle, this can be done using 2 bits for each 2 × 2 pooling window and is thus much more efficient to store as compared to memorizing feature map(s) in float precision.

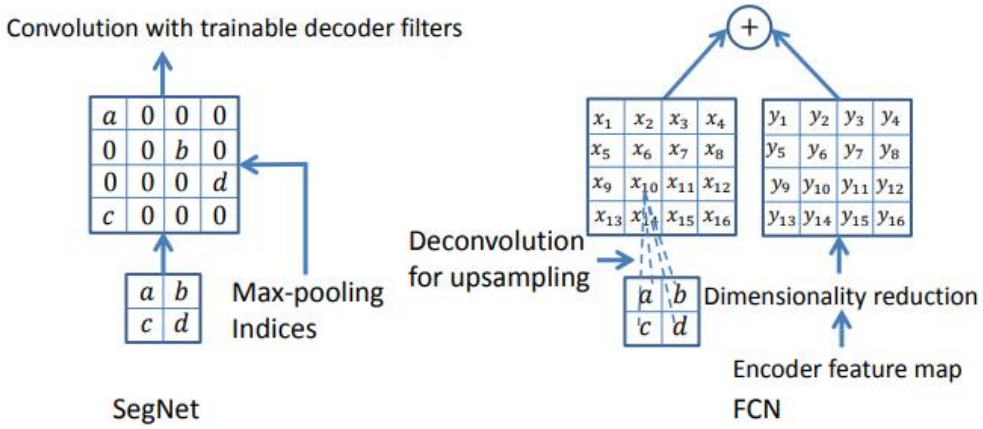
Decoder network

It up-samples its input feature map using the memorized max-pooling indices from the corresponding encoder feature map(s). This step produces sparse feature map(s). This SegNet decoding technique is illustrated in Fig. below. These feature maps are then convolved with a trainable decoder filter bank to produce dense feature maps. A batch normalization step is then applied to each of these maps.



Note that the decoder corresponding to the first encoder (closest to the input image) produces a multi-channel feature map, although its encoder input has 3 channels (RGB). This is unlike the other decoders in the network which produce feature maps with the same number of size and channels as their encoder inputs.

The high dimensional feature representation at the output of the final decoder is fed to a trainable soft-max classifier. This soft-max classifies each pixel independently. The output of the soft-max classifier is a K channel image of probabilities where K is the number of classes. The predicted segmentation corresponds to the class with maximum probability at each pixel.



SegNet and FCN decoders. a, b, c, d correspond to values in a feature map. SegNet uses the max pooling indices to upsample (without learning) the feature map and convolves with a trainable decoder filter bank. FCN upsamples by learning to deconvolve the input feature map and adds the corresponding encoder feature map to produce the decoder output. This feature map is the output of the max-pooling layer (includes sub-sampling) in the corresponding encoder. Note that there are no trainable decoder filters in FCN.

Working With Unet – Encoder Decoder Approach

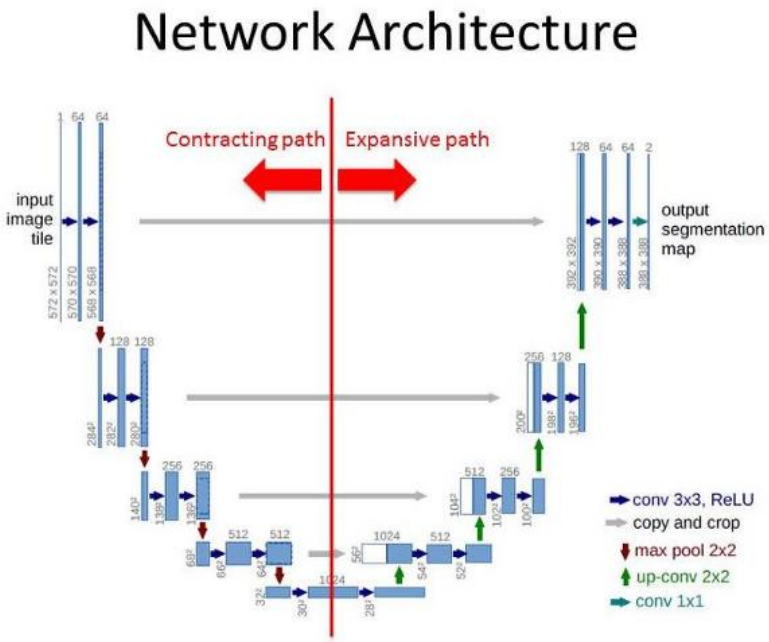
A U-Net is a convolutional neural network architecture that was developed for biomedical image segmentation. its architecture can be broadly thought of as an **encoder** network followed by a **decoder** network.

Unlike classification where the end result of the deep network is the only important thing, semantic segmentation not only requires discrimination at pixel level but also a mechanism to project the discriminative features learnt at different stages of the encoder onto the pixel space.

U-Nets have been found to be very effective for tasks where the output is of similar size as the input and the output needs that amount of spatial resolution. This makes them very good for creating segmentation masks and for image processing/generation such as super resolution or colourisation.

When convolutional neural nets are commonly used with images for classification, the image is taken and downsampled into one or more classifications using a series of stride two convolutions reducing the grid size each time.

To be able to output a generated image of the same size as the input, or larger, there needs to be an upsampling path to increase the grid size. This makes the network layout resemble a U shape.



A U-Net the downsampling/encoder path forms the left-hand side of the U and the upsampling /decoder path forms the right-hand part of the U.

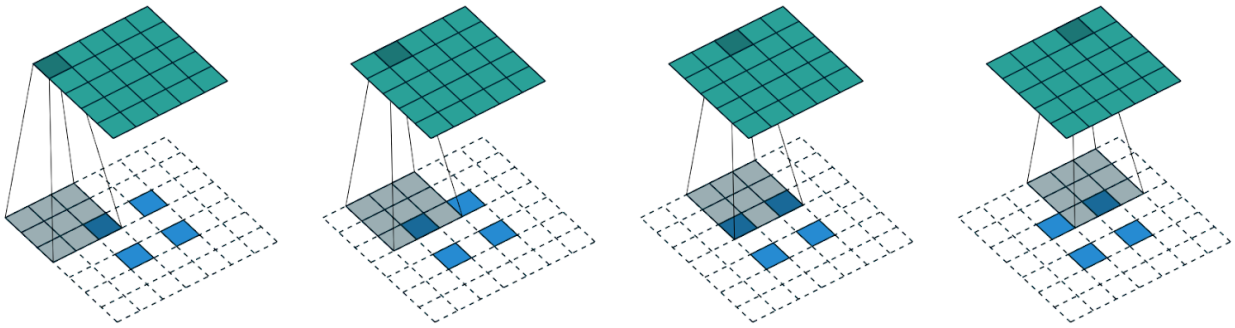
For the upsampling/decoder path several transposed convolutions accomplish this, each adding pixels between and around the existing pixels. Essentially the reverse of the downsampling path is carried out. The options for the upsampling algorithms are discussed further on.

Note that this model's U-Net based architecture also has cross connections which are detailed further on, these weren't part of the original U-Net architecture.

Upsampling/ transposed convolutions

Each upsample in the decoder/upsampling part of the network (right hand part of the U) needs to add pixels around the existing pixels and also in-between the existing pixels to eventually reach the desired resolution.

This process can be visualised as below from the paper "A guide to convolution arithmetic for deep learning" where zeros are added between the pixels. The blue pixels are the original 2x2 pixels being expanded to 5x5 pixels. 2 pixels of padding around the outside are added and also a pixel between each pixel. In this example all new pixels are zeros (white).



The transpose of convolving a 3×3 kernel over a 5×5 input using 2×2 strides (i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 0$). It is equivalent to convolving a 3×3 kernel over a 2×2 input (with 1 zero inserted between inputs) padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $\tilde{i}' = 3$, $k' = k$, $s' = 1$ and $p' = 2$).

This could have been improved with some simple initialisation of the new pixels by using the weighted average of the pixels (using bi-linear interpolation), as otherwise it is unnecessarily making it harder for the model to learn.

In this model it instead uses an improved method known as pixel shuffle or sub-pixel convolution with ICNR initialisation, which results in the gaps between the pixels being filled much more effectively. This is described in the paper "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network".

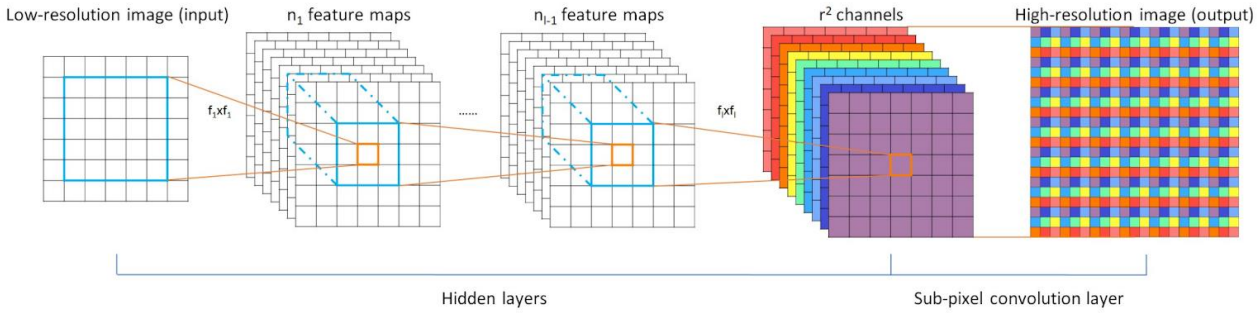


Figure 1. The proposed efficient sub-pixel convolutional neural network (ESPCN), with two convolution layers for feature maps extraction, and a sub-pixel convolution layer that aggregates the feature maps from LR space and builds the SR image in a single step.

The pixel shuffle upscales by a factor of 2, doubling the dimensions in each of the channels of the image (in its current representation at that part of the network). Replication padding is then performed to provide an extra pixel around the image. Then average pooling is performed to extract features smoothly and avoid the checkerboard pattern which results from many super resolution techniques.

After the representation for these new pixels are added, the subsequent convolutions improve the detail within them as the path continues through the decoder path of the network before then upscaling another step and doubling the dimensions.

U-Nets and fine image detail

When using a only a U-Net architecture the predictions tend to lack fine detail, to help address this cross or skip connections can be added between blocks of the network.

Rather than adding a skip connection every two convolutions as is in a ResBlock, the skip connections cross from same sized part in downsampling path to the upsampling path. These are the grey lines shown in the diagram above.

The original pixels are concatenated with the final ResBlock with a skip connection to allow final computation to take place with awareness of the original pixels inputted into the model. This results in all of the fine details of the input image are at the top on the U-Net with the input mapped almost directly to the output.

The outputs of the U-Net blocks are concatenated making them more similar to DenseBlocks than ResBlocks. However there are stride two convolutions that reduce the grid size back down, which also helps to keep memory usage from growing too large.

Implementation with Python

After downloading the data set- oxford iiit pet dataset we have saved the images folder to the drive so that we can read them if you want to read that folder locally. First, we need to import the libraries that we require.

```
import cv2
import matplotlib.pyplot as plt
import os from PIL
import Image
import numpy as np
import pandas as pd
```

Once we are done importing the libraries we initialize the directory where the images are stored. After that, we store the list directory in a variable and check what all is present in the list. Use the below code to obtain that and once you print the list you will see something similar to the content shown in the image.

```
os.chdir('/content/drive/My Drive/Images')
lst    = os.listdir('/content/drive/My Drive/Images')
print(lst)
```

['newfoundland_177.png', 'pomeranian_120.png', 'american_pit_bull_terrier_31.png', 'keeshond_88.png', 'german_shorthaired_17
3.png', 'Persian_262.png', 'english_setter_130.jpg', 'american_pit_bull_terrier_118.jpg', 'Ragdoll_121.png', 'Maine_Coon_78.p
ng', 'Maine_Coon_91.jpg', 'saint_bernard_146.jpg', 'Egyptian_Mau_63.jpg', 'Abyssinian_37.png', 'shiba_inu_28.jpg', 'Siamese_1
04.jpg', 'american_pit_bull_terrier_127.png', 'staffordshire_bull_terrier_90.jpg', 'newfoundland_188.jpg', 'Ragdoll_75.png',
'boxer_77.png', 'havanese_184.png', 'Birman_19.png', 'wheaten_terrier_20.jpg', 'chihuahua_127.jpg', 'havanese_190.png', 'sain
t_bernard_150.jpg', 'Birman_25.jpg', 'Egyptian_Mau_88.jpg', 'Russian_Blue_212.png', 'saint_bernard_39.jpg', 'keeshond_63.pn
g', 'Bengal_1.png', 'american_bulldog_139.png', 'german_shorthaired_8.jpg', 'Sphynx_239.jpg', 'staffordshire_bull_terrier_41.
png', 'Bengal_33.png', 'newfoundland_8.png', 'yorkshire_terrier_64.png', 'Persian_260.jpg', 'japanese_chin_42.png', 'american
_bulldog_111.png', 'British_Shorthair_51.png', 'german_shorthaired_161.jpg', 'german_shorthaired_20.png', 'keeshond_62.jpg',
'Persian_107.jpg', 'boxer_99.png', 'basset_hound_147.png', 'Egyptian_Mau_1.jpg', 'pomeranian_132.jpg', 'Bengal_182.jpg', 'Rag
doll_250.jpg', 'pomeranian_126.jpg', 'Abyssinian_20.png', 'Abyssinian_103.png', 'Ragdoll_255.png', 'Bombay_39.jpg', 'Sphynx_1
62.jpg', 'British_Shorthair_68.jpg', 'Abyssinian_25.jpg', 'pomeranian_21.png', 'american_bulldog_114.jpg', 'english_cocker_sp
aniel_138.jpg', 'saint_bernard_196.png', 'miniature_pinscher_27.jpg', 'Bengal_187.png', 'beagle_74.jpg', 'shiba_inu_205.png',
'english_setter_108.png', 'wheaten_terrier_4.jpg', 'keeshond_98.jpg', 'english_cocker_spaniel_139.jpg', 'Sphynx_205.png', 'Ab
yssinian_32.png', 'pug_185.jpg', 'keeshond_103.jpg', 'yorkshire_terrier_71.png', 'japanese_chin_42.jpg', 'beagle_72.jpg', 'po
meranian_132.png', 'great_pyrenees_59.jpg', 'pug_150.png', 'Maine_Coon_219.jpg', 'beagle_58.png', 'Russian_Blue_162.png', 'ba
sset_hound_142.png', 'american_bulldog_110.png', 'Bengal_187.jpg', 'chihuahua_132.jpg', 'havanese_95.jpg', 'Sphynx_61.png',
'pug_144.jpg', 'yorkshire_terrier_186.jpg', 'staffordshire_bull_terrier_41.jpg', 'Maine_Coon_187.jpg', 'Birman_24.jpg', 'beag
le_125.png', 'english_cocker_spaniel_107.jpg', 'english_setter_76.jpg', 'american_bulldog_148.png', 'pug_187.jpg', 'Persian_1

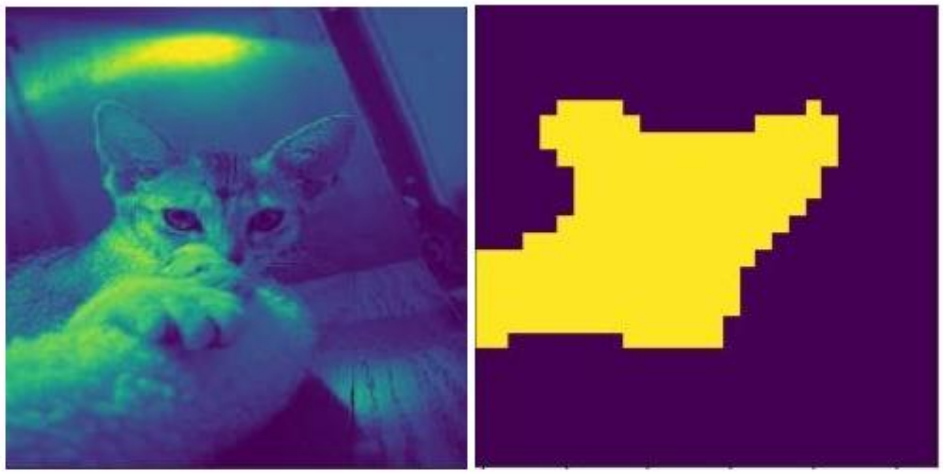
After this, we have created two lists one for storing masks and the other for storing the image. (mask & img). After storing the image and mask we have picked only 1000 images with their corresponding masks. Use the code shown below to do the same.

```
mask = []
img = []
for filename in lst:
   if filename.endswith('.jpg'):
   img.append(filename)
   if filename.endswith('.png'):
   mask.append(filename)

img.sort()
mask.sort()
img = img[:1000]
masks = mask[:1000]
```

After sorting has been done we are reading the image and label mask in X and y respectively. We have captured the index of the image file and stored the directory of that index image. After that, we open that image and resize it also then we convert that image into a grayscale image and store its index. We then store the mask of the image corresponding to the index we stored the grayscale image. After that, we give the directory of the mask and read the mask. Finally, we have pre-processed the mask image by resizing it and normalizing the pixel value then stored it at the pre-processed mask image at the output array at the same index position. X[n] stores the image and y[n] stores the corresponding mask.

Pre-processing of the image and mask

```
y = np.zeros((1000, 28,28), dtype=np.float32)
X = np.zeros((1000,224, 224, 1), dtype=np.float32)

for file in img:
index = img.index(i)
dir_img = os.path.join('/content/drive/My Drive/Images', i)
img = Image.open(dir1)
img = img.resize((224, 224))
img = np.reshape(img.convert('L'), (224,224,1))
X[n] = img            mask = masks[index]
dir_mask = os.path.join('/content/drive/My Drive/Images, mask)
mask_img = cv2.imread(dir_mask)
mask_img = (mask!=2)*1.0        mask_img = cv2.resize(mask, (28, 28))
     mask_img = 1.0*(mask[:,:,0]>0.2)
     y[n] = mask
```



If you will plot the pre-processed image and mask you will see something similar to the image shown above. The above image shows the pre-processed image as well as its mask. Now we install the pre-trained model for segmentation and load all the useful libraries from that segmentation model as shown in the code below.

We will have to add a few convolution libraries as well to add our own custom layers.

```
!pip install git+https://github.com/qubvel/segmentation_models
from segmentation_models import Unet
from segmentation_models.backbones import get_preprocessing
rom segmentation_models.losses import bce_jaccard_loss
from segmentation_models.metrics import iou_score from sklearn.model_selection
import train_test_split import tensorflow as tf
from keras.optimizers import Adam
from tensorflow.keras.losses import binary_crossentropy from keras.models
import model_from_json

from keras.layers import Input, Conv2D, Reshape
from keras.models import Model
```

We then divide the data into training and testing X, y respectively. After dividing we have imported ResNet as a backbone network and loaded the weights. After this, we pre-process the input and output accordingly.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
BACKBONE = 'resnet34' preprocess_input = get_preprocessing(BACKBONE)
X_train = preprocess_input(X_train)
X_test = preprocess_input(X_test)
```

Building The UNet Model

We have then imported the U-net model being ResNet as a backbone network and loaded weights of image net. We have then defined the input shape that is expected by the base model and the custom layer that takes that base mode input whose output is then passed to the UNet model. The output of UNet model is then passed to other defined ConvNet layers having activation as ReLU. The final output is then reshaped to 28X28. At last, we have defined the model that takes input (inp) and gives us the output (x_out) using the base_model.

```
from keras.layers import Reshape N = x_train.shape[-1]

base_model = Unet(backbone_name='resnet34', encoder_weights='imagenet')

input_base_model = Input(shape=(224, 224, N))

l1 = Conv2D(3, (1, 1))(inp)

out = base_model(l1)

x1 = Conv2D(10, kernel_size =3,strides=2,padding = "same",
activation="relu")(out)
x1 =layers.BatchNormalization()

x2= Conv2D(10, kernel_size=3,strides=2,padding = "same", activation="relu")(x1)
x2 =layers.BatchNormalization()

x3 = Conv2D(10, kernel_size=3,strides=2,padding = "same",
activation="relu")(x2)
x3 =layers.BatchNormalization()

x4 = Conv2D(1, kernel_size=2,strides=2,padding = "same", activation="relu")(x3)

x_out = Reshape((28,28))(x4)

model = Model(input_base_model, x_out, name=base_model.name)
```

We have then defined the function for metric, loss and optimizer that we will be using. Dice coefficient as the metric, loss function as binray_cross_entropy and sgd as an optimizer. After defining everything we have compiled the model and fitted the training and validation data to the model. The code illustration for the same is given below.

```
def dice_coefficient(y_true, y_pred):
    numerator = 2 * tf.reduce_sum(y_true * y_pred)
    denominator = tf.reduce_sum(y_true + y_pred)

return numerator / (denominator + tf.keras.backend.epsilon())

def loss(y_true, y_pred):
    return binary_crossentropy(y_true, y_pred) - tf.log(dice_coefficient(y_true,
y_pred) + tf.keras.backend.epsilon())

model.compile(optimizer='sgd', loss=loss, metrics=[dice_coefficient])

model.fit(X_train,y_train,batch_size=32,epochs=30,validation_data=(X_test, y_test))
```

```
Epoch 1/20
850/850 [==============================] - 12s 14ms/step - loss: 0.9316 - dice_coefficient: 0.6165 - val_loss: 0.9889 - val_dic
e_coefficient: 0.5956
Epoch 2/20
850/850 [==============================] - 12s 14ms/step - loss: 0.9036 - dice_coefficient: 0.6256 - val_loss: 0.9745 - val_dic
e_coefficient: 0.5999
Epoch 3/20
850/850 [==============================] - 12s 14ms/step - loss: 0.8812 - dice_coefficient: 0.6331 - val_loss: 0.9594 - val_dic
e_coefficient: 0.6065
Epoch 4/20
850/850 [==============================] - 12s 15ms/step - loss: 0.8532 - dice_coefficient: 0.6427 - val_loss: 0.9598 - val_dic
e_coefficient: 0.6122
Epoch 5/20
850/850 [==============================] - 12s 14ms/step - loss: 0.8230 - dice_coefficient: 0.6528 - val_loss: 0.9183 - val_dic
e_coefficient: 0.6183
Epoch 6/20
850/850 [==============================] - 12s 14ms/step - loss: 0.8005 - dice_coefficient: 0.6605 - val_loss: 1.0867 - val_dic
e_coefficient: 0.5595
Epoch 7/20
850/850 [==============================] - 12s 14ms/step - loss: 0.7751 - dice_coefficient: 0.6696 - val_loss: 0.8959 - val_dic
e_coefficient: 0.6334
Epoch 8/20
850/850 [==============================] - 12s 14ms/step - loss: 0.7441 - dice_coefficient: 0.6800 - val_loss: 0.8400 - val_dic
e_coefficient: 0.6491
Epoch 9/20
850/850 [==============================] - 12s 14ms/step - loss: 0.7160 - dice_coefficient: 0.6898 - val_loss: 0.7916 - val_dic
e_coefficient: 0.6652
Epoch 10/20
850/850 [==============================] - 12s 14ms/step - loss: 0.6898 - dice_coefficient: 0.6992 - val_loss: 0.7662 - val_dic
e_coefficient: 0.6738
Epoch 11/20
850/850 [==============================] - 12s 14ms/step - loss: 0.6650 - dice_coefficient: 0.7083 - val_loss: 0.7455 - val_dic
e_coefficient: 0.6813
Epoch 12/20
850/850 [==============================] - 12s 14ms/step - loss: 0.6425 - dice_coefficient: 0.7165 - val_loss: 0.7216 - val_dic
e_coefficient: 0.6896
Epoch 13/20
```

Once the model is trained we can then evaluate its performance over the testing set using the code shown below which will give us the least loss and highest accuracy. After evaluation, we have saved the trained model weights by serializing it.

```
model.evaluate(x_test, y_test)
```

```
200/200 [==============================] - 1s 5ms/step

[0.6069157894452413, 0.7474195226033529]
```

```
from keras.models import model_from_json model_json = model.to_json()

with open("model.json", "w") as json_file:
    json_file.write(model_json)
model.save_weights("model.h5")
print("Saved model to disk")
```

Prediction by the UNet model

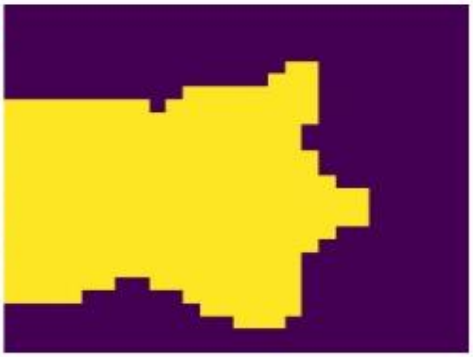
After saving the model we made predictions on X_train and X_test using the trained model and stored it. After making predictions we then have defined a function to visualize the prediction made by the model. The function expects input array and output array and the predictions. We have defined k to be none so to randomly pick the images from the training data and for the same index of the picked training image we have taken the mask. We have then defined the figure size and plotted all three that are image, mask, and predicted mask.

```
training_pred = model.predict(X_train)
testing_pred = model.predict(X_test)

def prediction(X, y, pred, k=None):
    if k  == 'None':
        k = np.random.randint(0, len(X))

    has_mask = y[k].max() > 0

    figure, j = plt.subplots(1, 3, figsize=(20, 20))
    j[0].imshow(X[k, ..., 0])
    if has_mask:
        j[0].contour(y[i].squeeze())
    k[1].imshow(y[i].squeeze())
    k[2].imshow(pred[i].squeeze())
    if has_mask:
        k[2].contour(preds[i].squeeze())
```

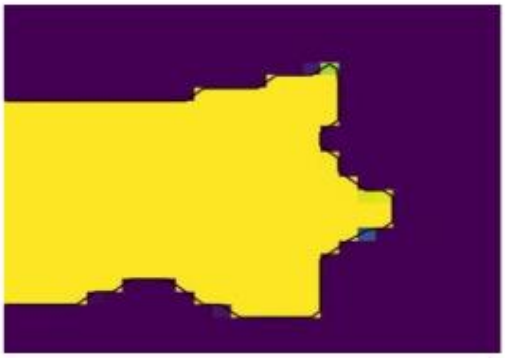
We now plot the images to see the model results. Use the below code to do so

```
prediction(X_train, y_train, training_prediction)
```

Cat



Ground Truth



Predicted Mask

The above images show the randomly picked images, corresponding ground truth of the mask and predicted mask by the trained UNet model.

Summary

- Image segmentation is the task of partitioning an image based on the objects present and their semantic importance.
- Semantic segmentation describes the process of associating each pixel with a class label. There is no clear distinction between the different object of same class. All objects of same class are colored same.
- In instance segmentation, we mask each instance of an object contained in an image independently.
- The encoder is usually is a pre-trained classification network like VGG/ResNet followed by a decoder network.
- The task of the decoder is to semantically project the discriminative features (lower resolution) learnt by the encoder onto the pixel space (higher resolution) to get a dense classification.
- A U-Net is a convolutional neural network architecture that was developed for biomedical image segmentation. its architecture can be broadly thought of as an encoder network followed by a decoder network
- U-Nets have been found to be very effective for tasks where the output is of similar size as the input and the output needs that amount of spatial resolution.