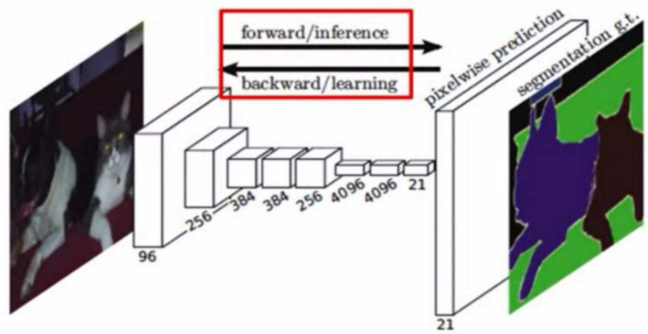
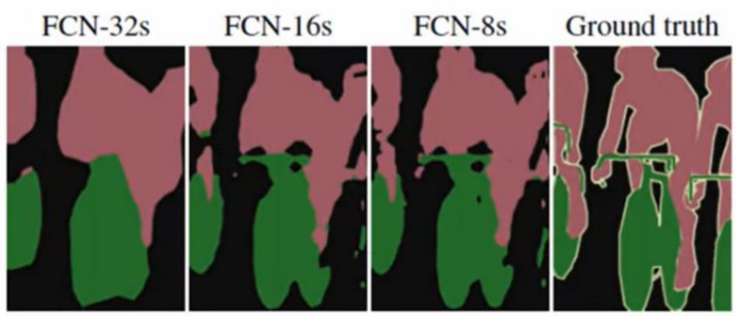
Fully Convolutional Neural Networks

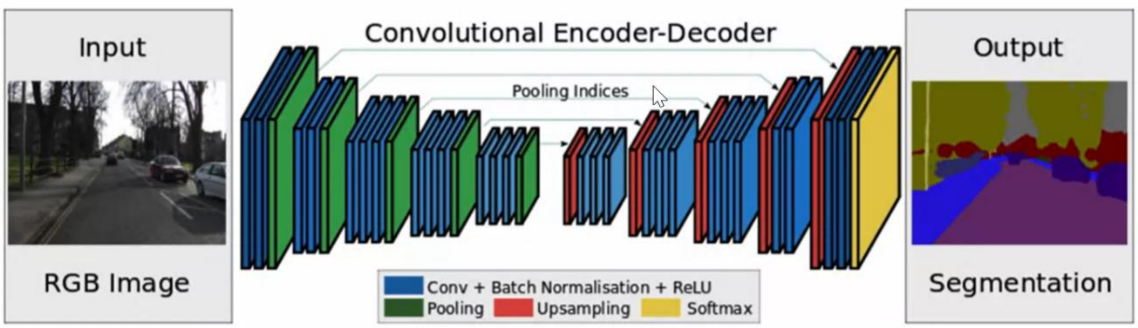


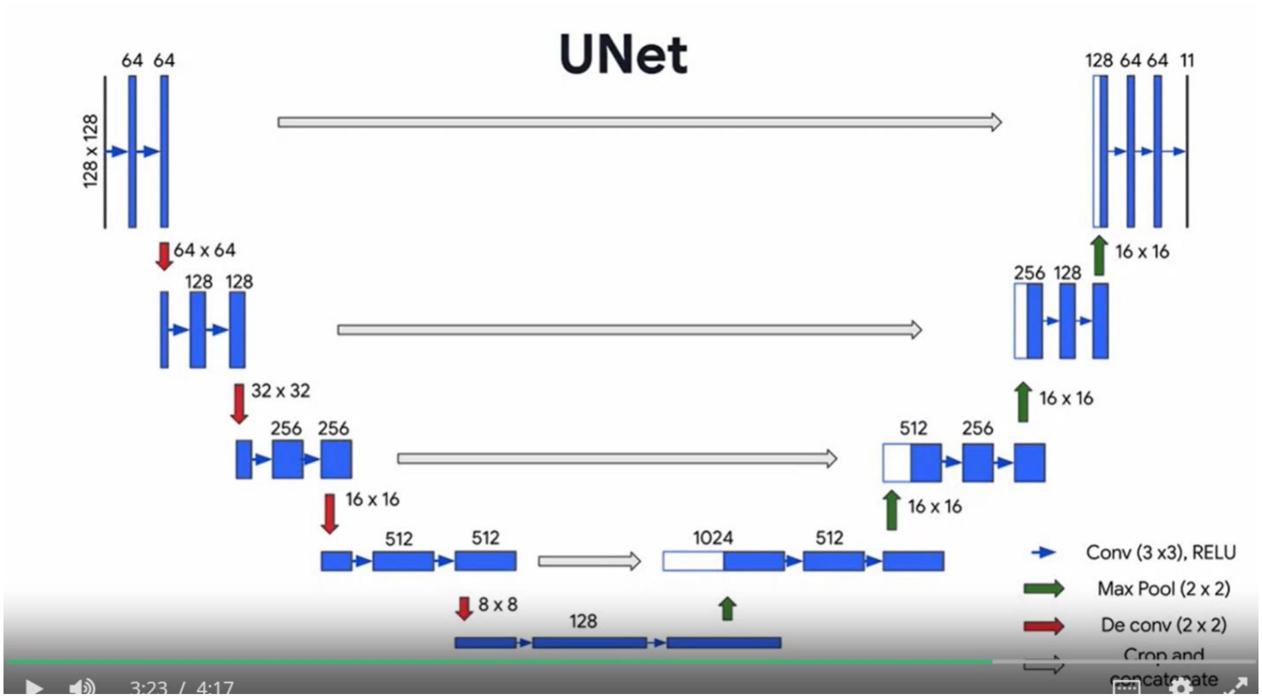
https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf

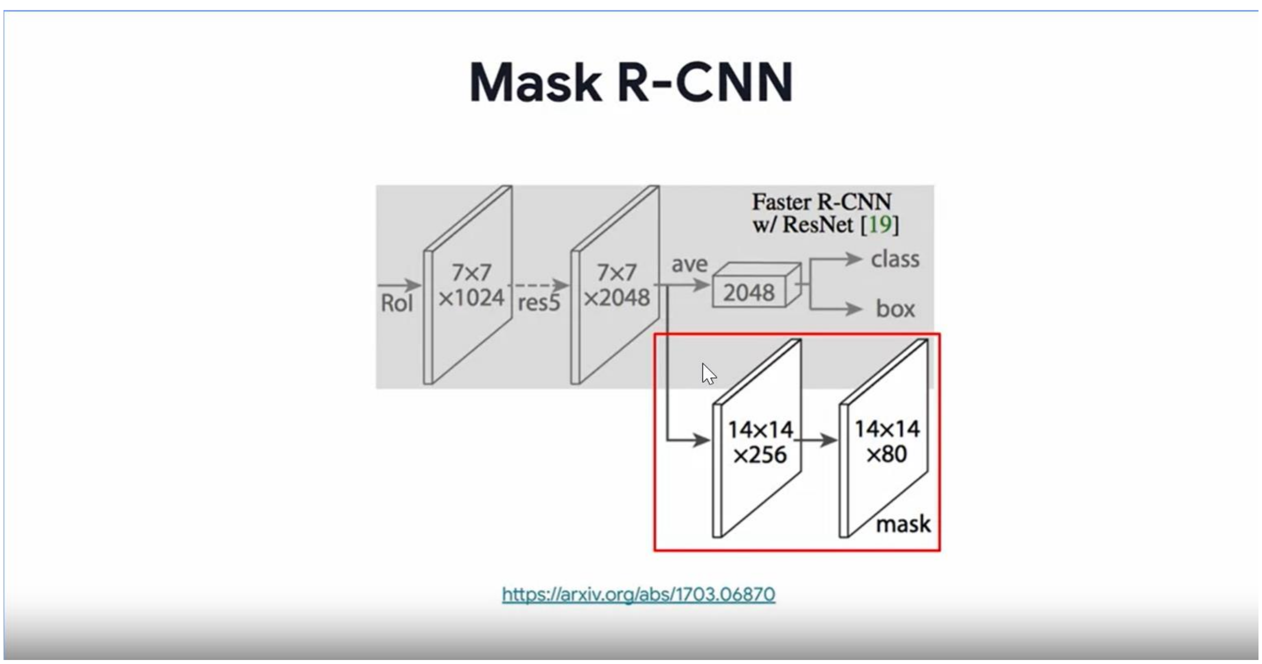
Comparison of Different FCNs



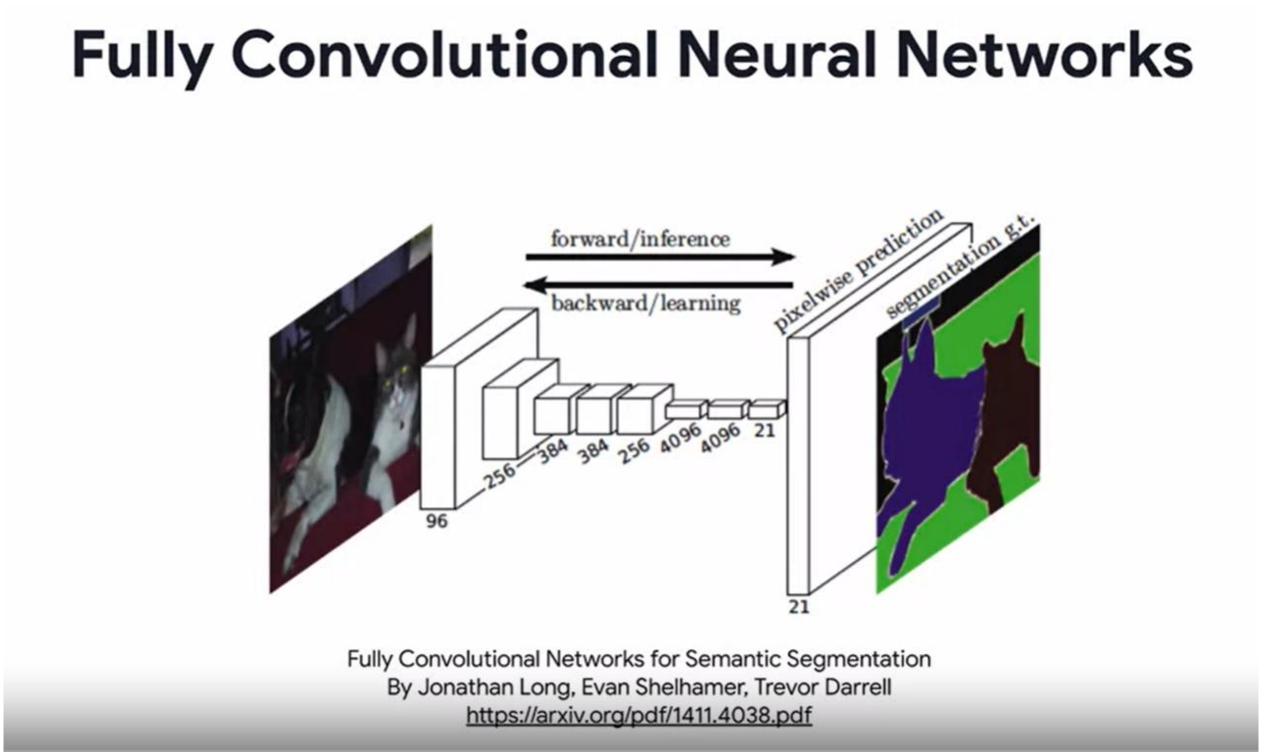
SegNet




UNet
64 64
128 x 128
64 x 64
128 128
32 x 32
256 256
16 x 16
512 512
8 x 8
128
1024 512
512 256
16 x 16
256 128
16 x 16
128 64 64 11
Conv (3 x3), RELU
Max Pool (2 x 2)
De conv (2 x 2)
3:23 / 4:17


Mask R-CNN
Faster R-CNN
w/ ResNet [19]
RoI
7×7
×1024
res5
7×7
×2048
ave
2048
class
box
14×14
×256
14×14
×80
mask
https://arxiv.org/abs/1703.06870

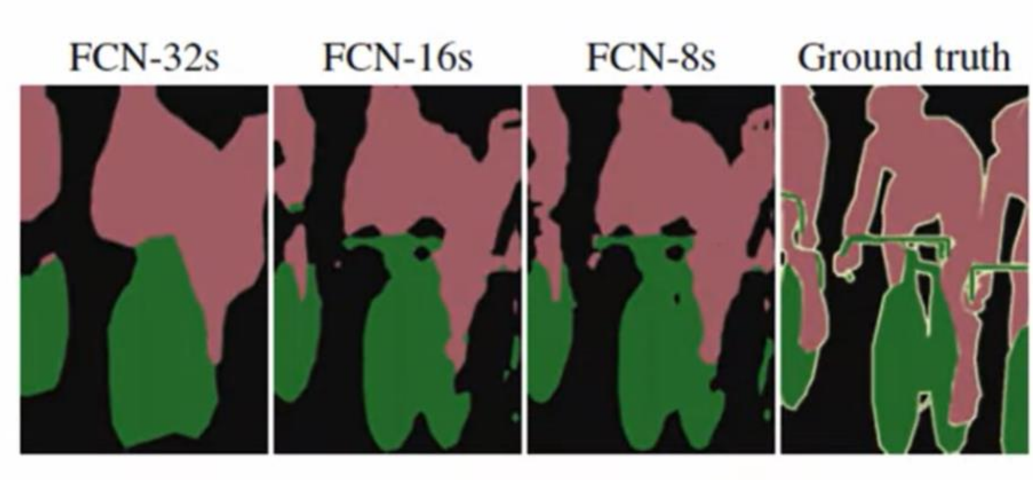
FCN Architecture Details



Encoders

- Popular encoder architectures:
  - VGG-16
  - ResNet-50
  - MobileNet
- Reuse convolutional layers for feature extraction.
  - Do not reuse fully connected layers

Decoders


FCN-32s
FCN-16s
FCN-8s
Ground truth

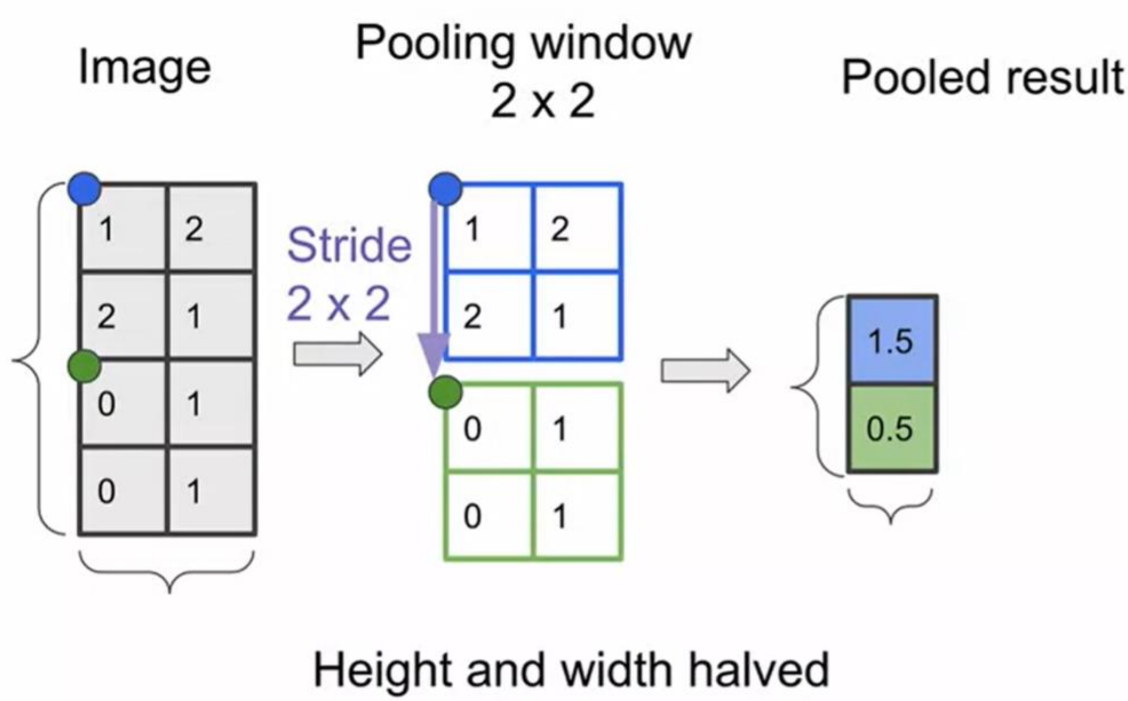
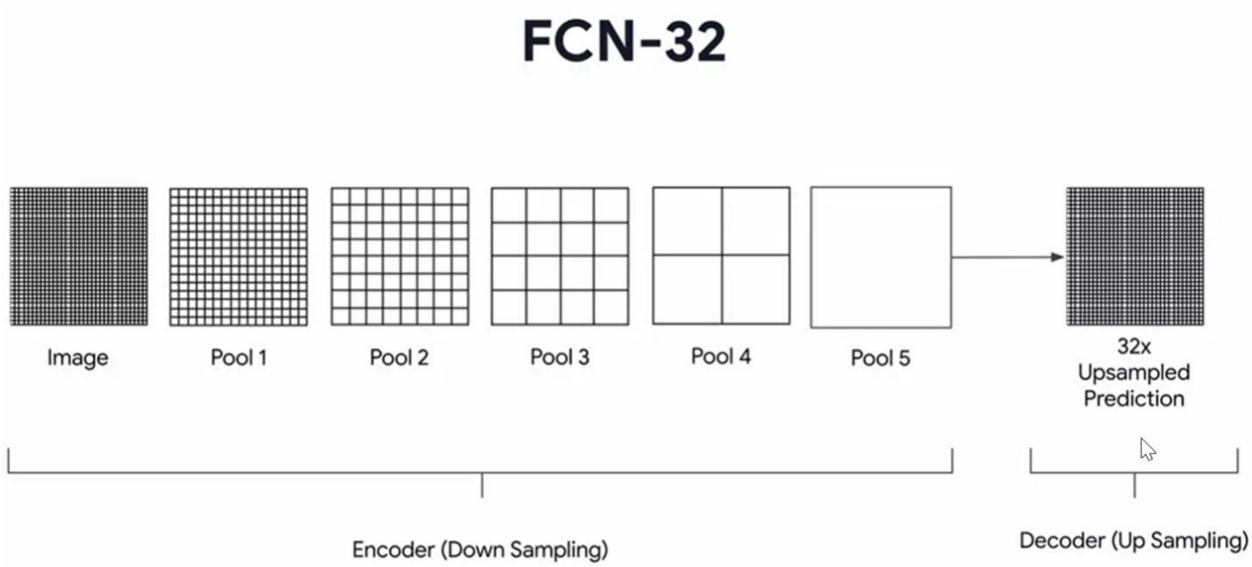
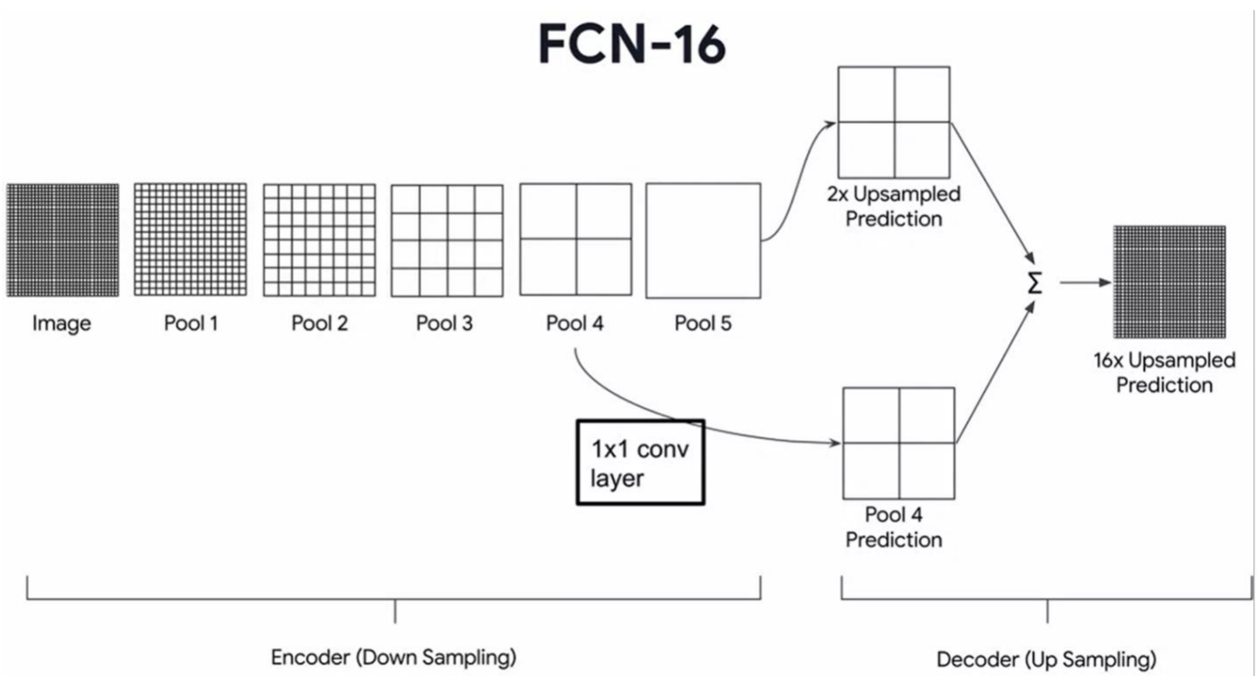
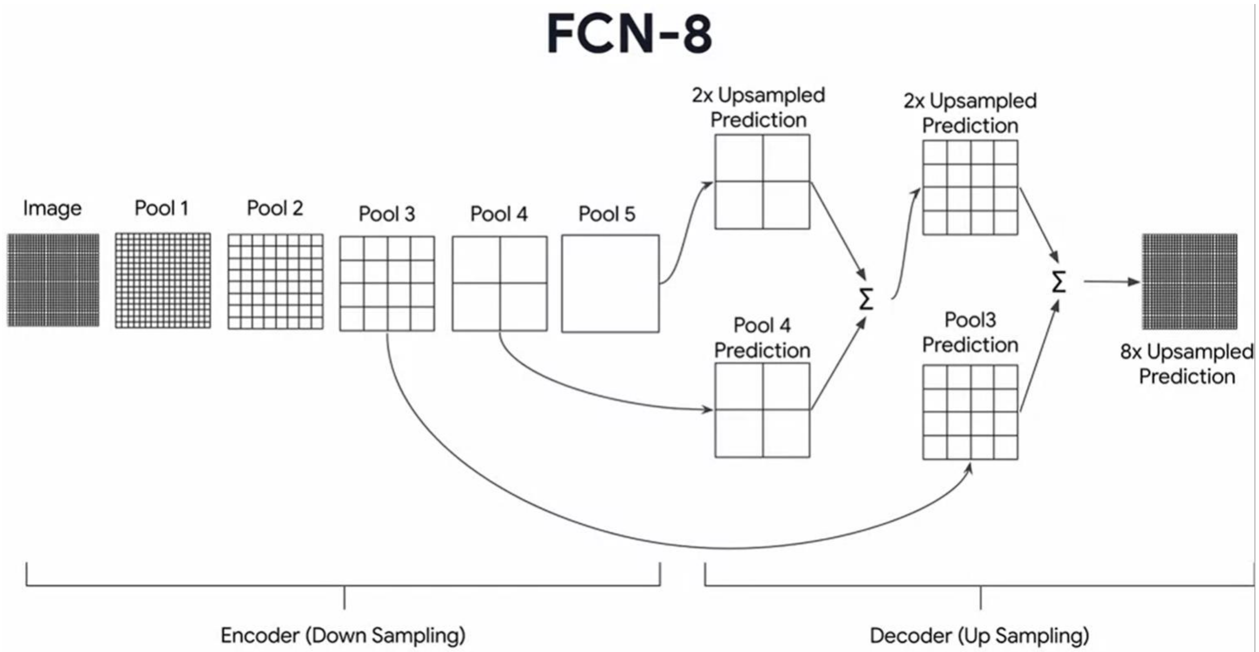


Image
Pooling window
2 x 2
Pooled result
Stride
2 x 2
1
2
2
1
0
1
0
1
1
2
2
1
0
1
0
1
1.5
0.5
Height and width halved


FCN-32
Image
Pool 1
Pool 2
Pool 3
Pool 4
Pool 5
32x Upsampled Prediction
Encoder (Down Sampling)
Decoder (Up Sampling)


FCN-16
2x Upsampled Prediction
Image
Pool 1
Pool 2
Pool 3
Pool 4
Pool 5
Σ
16x Upsampled Prediction
1x1 conv layer
Pool 4 Prediction
Encoder (Down Sampling)
Decoder (Up Sampling)



Upsampling

- Upsampling is increasing height and width of the feature map.
- Two types of layers used in TensorFlow:
  - Simple Scaling - UpSampling2D
  - Transposed Convolution(Deconvolution) - Conv2DTranspose

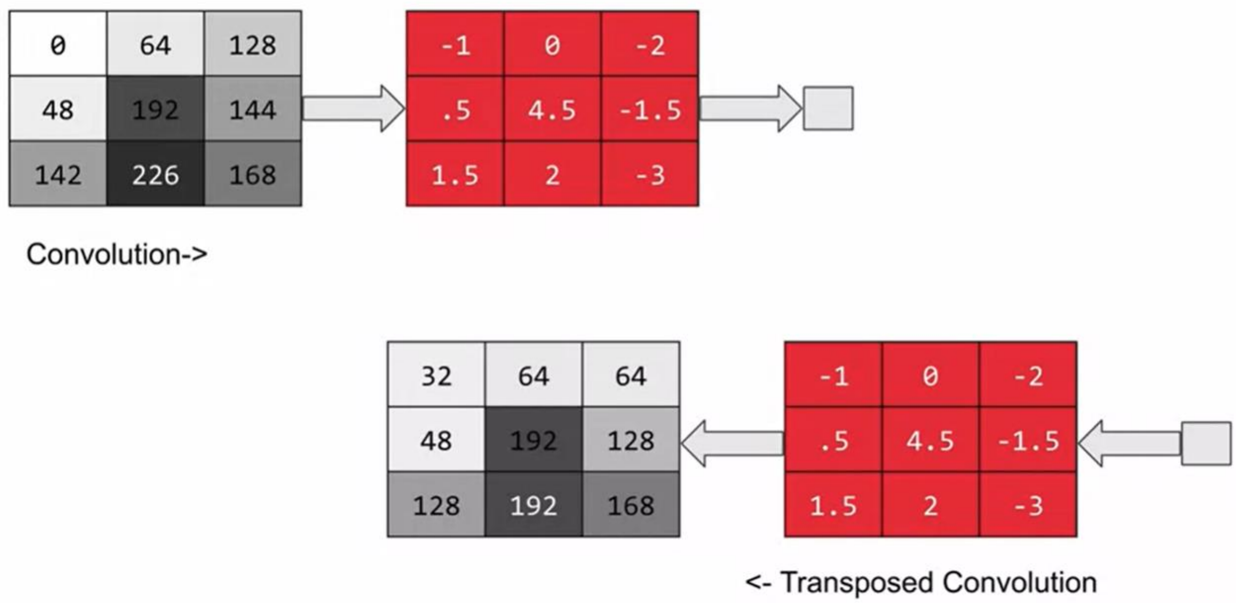
Simple Scaling - UpSampling2D

- Upsampling2D scales up the image
- Two Types of scaling:
    - Nearest
        - Copies value from nearest pixel.
    - Bilinear
        - linear interpolation from nearby pixels.

UpSampling2D - Usage

```
x = UpSampling2D(
    size=(2, 2),
    data_format=None,
    interpolation='nearest')(x)

size: int or tuple of two ints
data_format: 'channels_first', 'channels_last' or None
interpolation: 'nearest' or 'bilinear'
```

Transposed Convolution



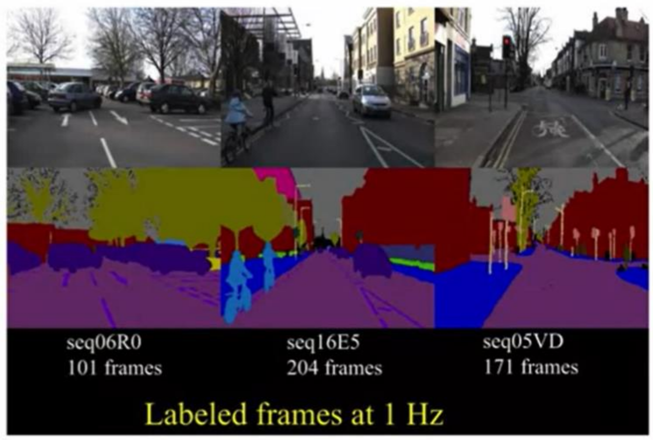
Conv2DTranspose

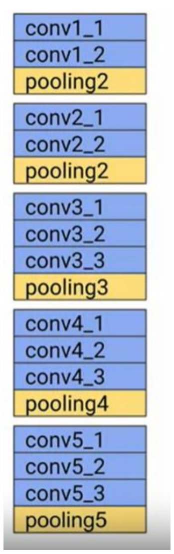
- Reverse of Convolution.
- Applied to output of a convolution operation.
- Uses a kernel of a specified size and stride in order to recreate the original input before the convolution operation.

```
Conv2DTranspose(
    filters=32,
    kernel_size=(3, 3)
)
```

Encoder in Code

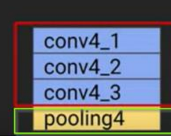
- The Cambridge Driving, labelled video database (aka CamVid) contains 10 minutes of 30fps video, segmented and labelled with 32 classes
- GitHub account divamgupta has taken a subsample of the CamVid dataset to create a smaller dataset.



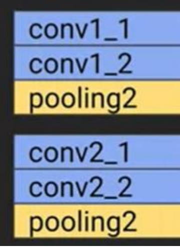


- Divam Gupta's GitHub account containing a subsample of the CamVid dataset to create a smaller dataset.

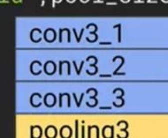
```
def block(x, n_convs, filters, kernel_size, activation, pool_size, pool_stride, block_name):
  for i in range(n_convs):
      x = tf.keras.layers.Conv2D(filters=filters,
                                 kernel_size=kernel_size, activation=activation,
                                 padding='same',
                                 name="{}_conv{}".format(block_name, i + 1))(x)

  x = tf.keras.layers.MaxPooling2D(pool_size=pool_size, strides=pool_stride,
                                   name="{}_pool{}".format(block_name, i+1 ))(x)

  return x
```

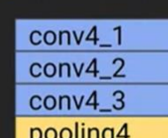


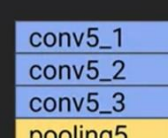
```
def VGG_16(image_input):
  x = block(image_input, n_convs=2, filters=64, kernel_size=(3,3),
            activation='relu',pool_size=(2,2), pool_stride=(2,2),
            block_name='block1')
  p1= x

  x = block(x, n_convs=2, filters=128, kernel_size=(3,3),
            activation='relu',pool_size=(2,2), pool_stride=(2,2),
            block_name='block2')
  p2 = x

  ...
```



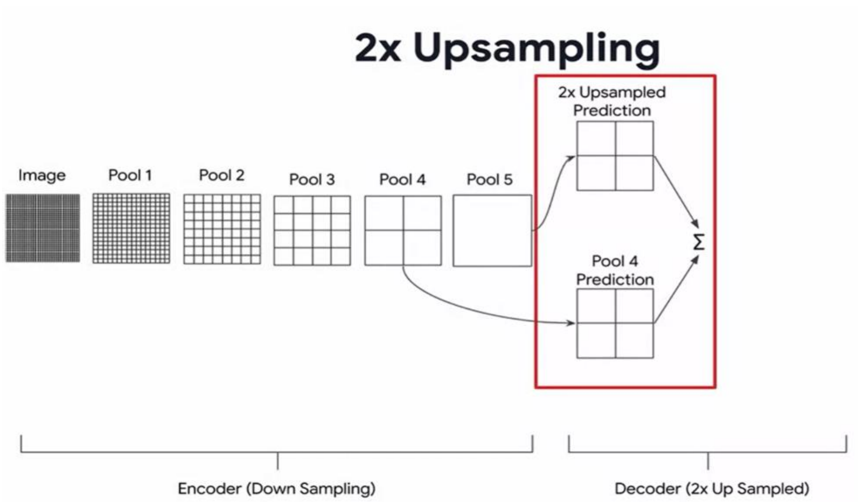

```
 x = block(x, n_convs=3, filters=256, kernel_size=(3,3), activation='relu',pool_size=(2,2),
          pool_stride=(2,2), block_name='block3')
p3 = x

x = block(x, n_convs=3, filters=512, kernel_size=(3,3), activation='relu',pool_size=(2,2),
          pool_stride=(2,2), block_name='block4')
p4 = x

x = block(x, n_convs=3, filters=512, kernel_size=(3,3), activation='relu',pool_size=(2,2),
          pool_stride=(2,2), block_name='block5')
p5 = x

..
```







Decoder in Code (FCN8)



Define Decoder - 2x UpSampling

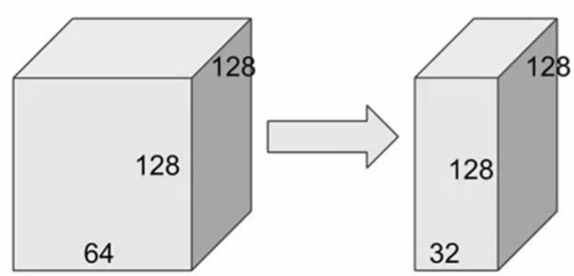
```python
def fcn8_decoder(convs, n_classes):
  f1, f2, f3, f4, f5 = convs

  o = tf.keras.layers.Conv2DTranspose(n_classes, kernel_size=(4,4),
                                      strides=(2,2), use_bias=False )(f5)

  o = tf.keras.layers.Cropping2D(cropping=(1,1))(o)

  o2 = f4
  o2 = ( tf.keras.layers.Conv2D(n_classes, (1,1),
                                activation='relu', padding='same'))(o2)

  o = tf.keras.layers.Add()([o, o2])
  ...
```

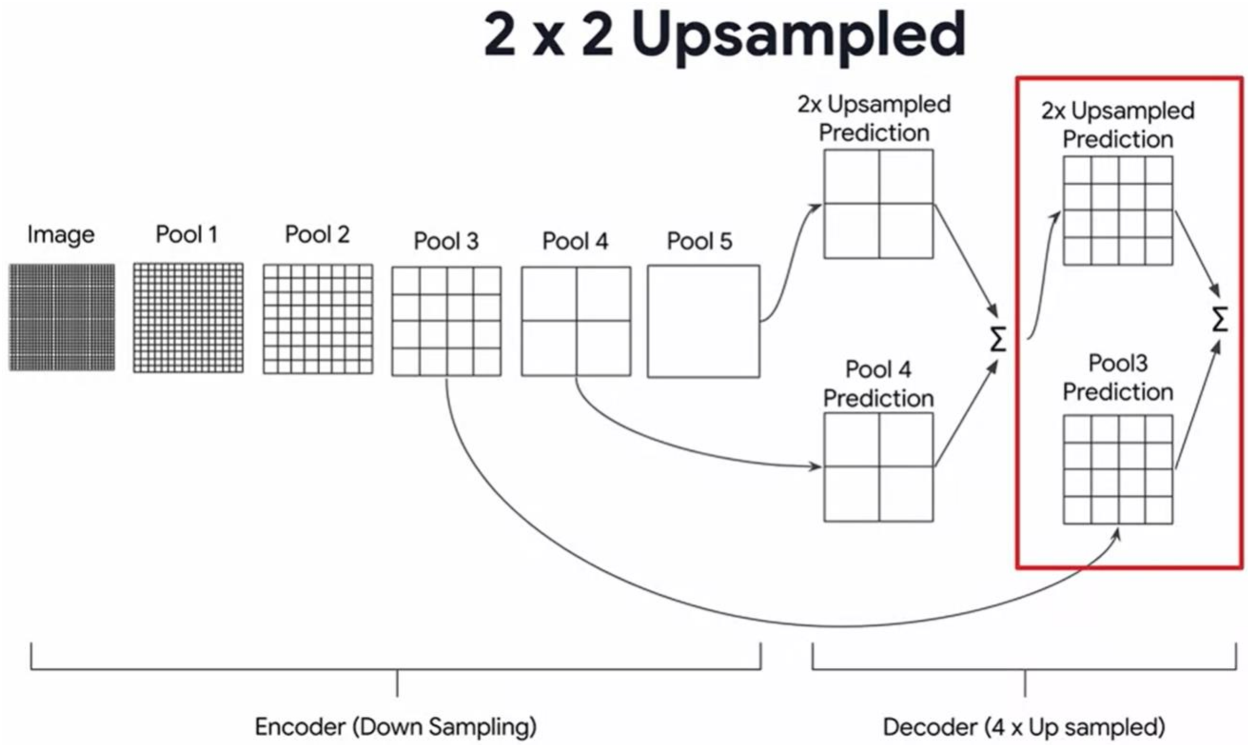
1 x 1 Convolutions

(B, F, H, W) - *B = # batches; F= # filters, H, W = Height/Width*

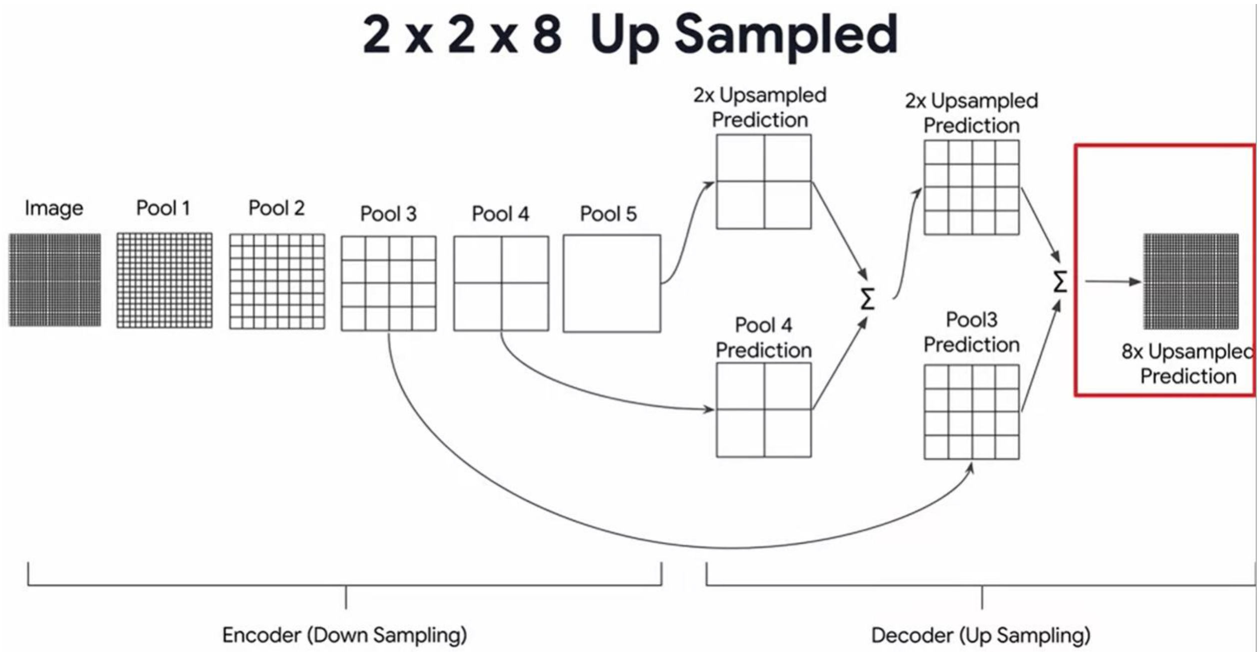
Apply a layer with N 1x1 Convolutions with stride of 1:

(B, N, H, W) - *B = # batches; N= # filters, H, W = Height/Width*






```
def fcn8_decoder(convs, n_classes):
  ...
  o = (tf.keras.layers.Conv2DTranspose( n_classes, kernel_size=(4,4),
                                        strides=(2,2))(o)

  o = tf.keras.layers.Cropping2D(cropping=(1, 1))(o)

  o2 = ( tf.keras.layers.Conv2D(n_classes,(1,1), activation='relu',
                                padding='same'))(f3)


  o = tf.keras.layers.Add()([o, o2])
```



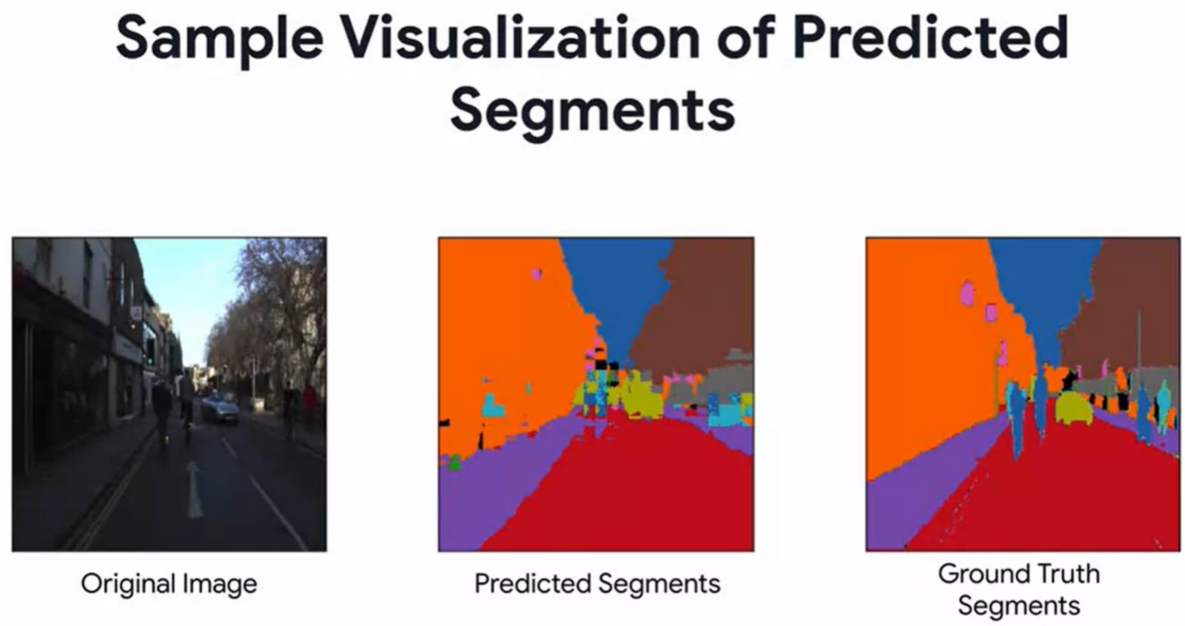
Define Decoder

```
def fcn8_decoder(convs, n_classes):
  …

  o = tf.keras.layers.Conv2DTranspose(n_classes , kernel_size=(8,8),
                                      strides=(8,8))(o)

  o = (tf.keras.layers.Activation('softmax'))(o)

  return o
```

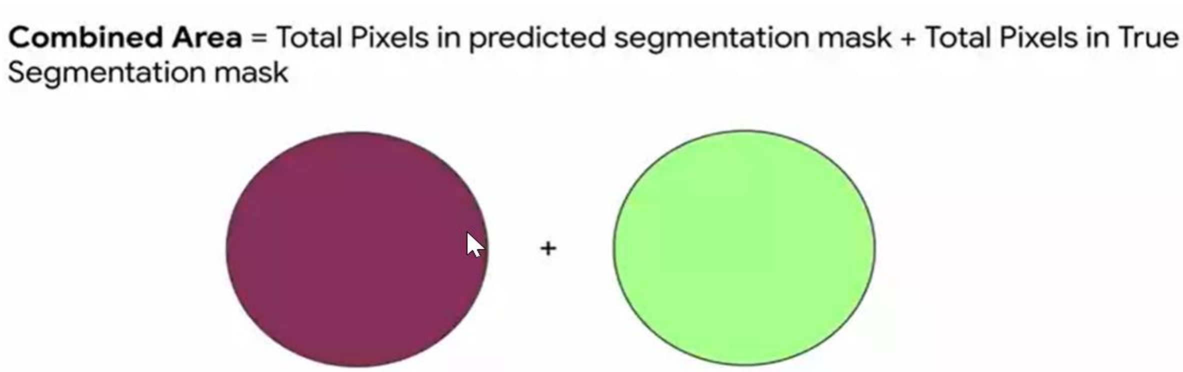
Define Final Model

```
def segmentation_model():
  inputs = tf.keras.layers.Input(shape=(224,224,3,))
  convs = VGG_16(image_input=inputs)
  outputs = fcn8_decoder(convs, 12)
  model = tf.keras.Model(inputs=inputs, outputs=outputs)
  return model

model = segmentation_model()
```

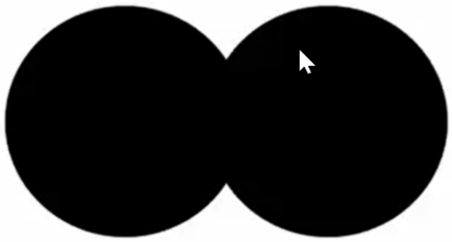
Evaluation with IoU and Dice Score



Area of Overlap



Area of Union = Total Pixels in predicted segmentation mask + Total Pixels in True Segmentation mask - Area of Overlap

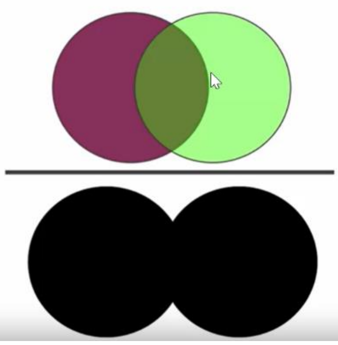


Calculate Areas

```python
def class_wise_metrics(y_true, y_pred):
 ...
  smoothening_factor = 0.00001

  for i in range(n_classes):
    intersection = np.sum((y_pred == i) * (y_true == i))
    y_true_area = np.sum((y_true == i))
    y_pred_area = np.sum((y_pred == i))
    combined_area = y_true_area + y_pred_area
    union_area = combined_area - intersection
```

Intersection Over Union

$$\text{IOU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

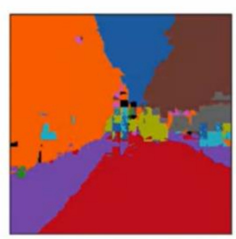


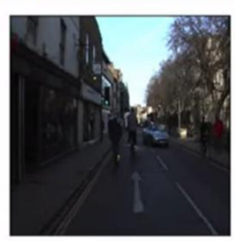
Calculate IOU

```python
def class_wise_metrics(y_true, y_pred):
  class_wise_iou = []

  ...
  for i in range(n_classes):
    ...
    iou = (intersection) / (union_area)
    class_wise_iou.append(iou)

  return class_wise_iou
```

IOU Results

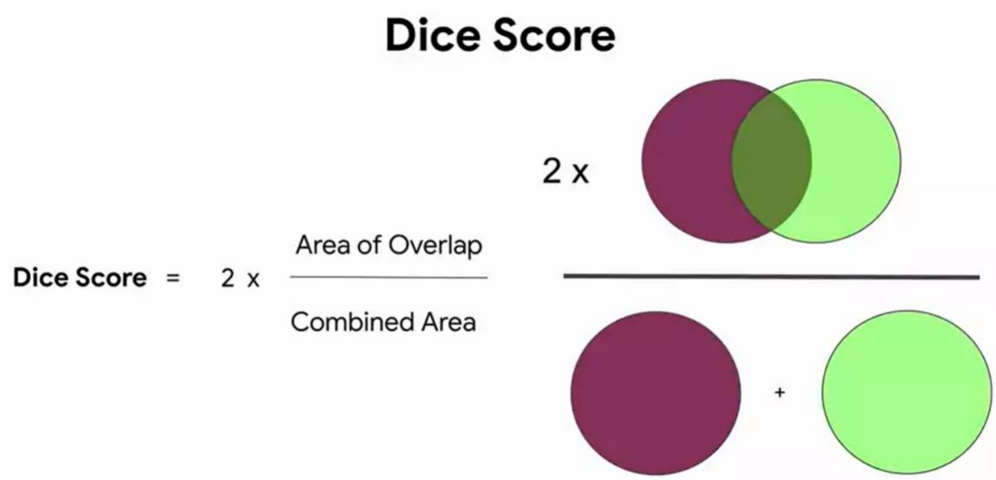
sky	0.8779669959482955
building	0.7570989578412737
column/pole	4.578754577665808e-10
road	0.9155431555822588
side walk	0.7235628237658467
vegetation	0.7664541807647628
traffic light	3.0202657798187055e-05
fence	0.006380242448568188
vehicle	0.2950299461448835
pedestrian	0.0001264333276608086
byciclist	0.023621930993270864
void	0.16456276759816527





You'll also use the dice score as one of the evaluation metrics in the Colab. The dice score is twice the area of overlap divided by the combined area. It can be used in similar circumstances to the intersection over union score, and they're often both used. The subtle difference between them is that the dice score tends to veer towards the average performance. Whereas the IOU helps you understand worst case performance. Code for this calculated class wise is also pretty straightforward. Iterate through the classes, calculate the intersection and the combined area for that class, and then calculate the score according to the formula. Again, we can get a metric for our classification using it. As you can see here, we did very well in some classes and we did quite poorly in others. Particularly alarming is that the pedestrian and cyclist scores are very poor.

You would not want to use a model like this in production.
Hopefully, that was a useful introduction into how the fully convolutional layers architecture can be used for image segmentation.
Now there's a lot to cover and there's no better way to do that than to dig into the code.
Next, you're going to practice everything that you've seen so far.
Building an image segmentation model to segment dashcam views for self-driving car tasks.
You're going to use VGG16 for the encoder and FCN8 for the decoder, and you'll also use things like IOU and the dice score to evaluate your model's performance. In the next lesson, we'll switch gears and we'll look at the unit architecture for image segmentation

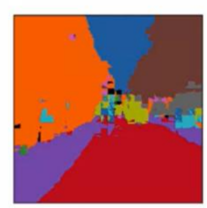


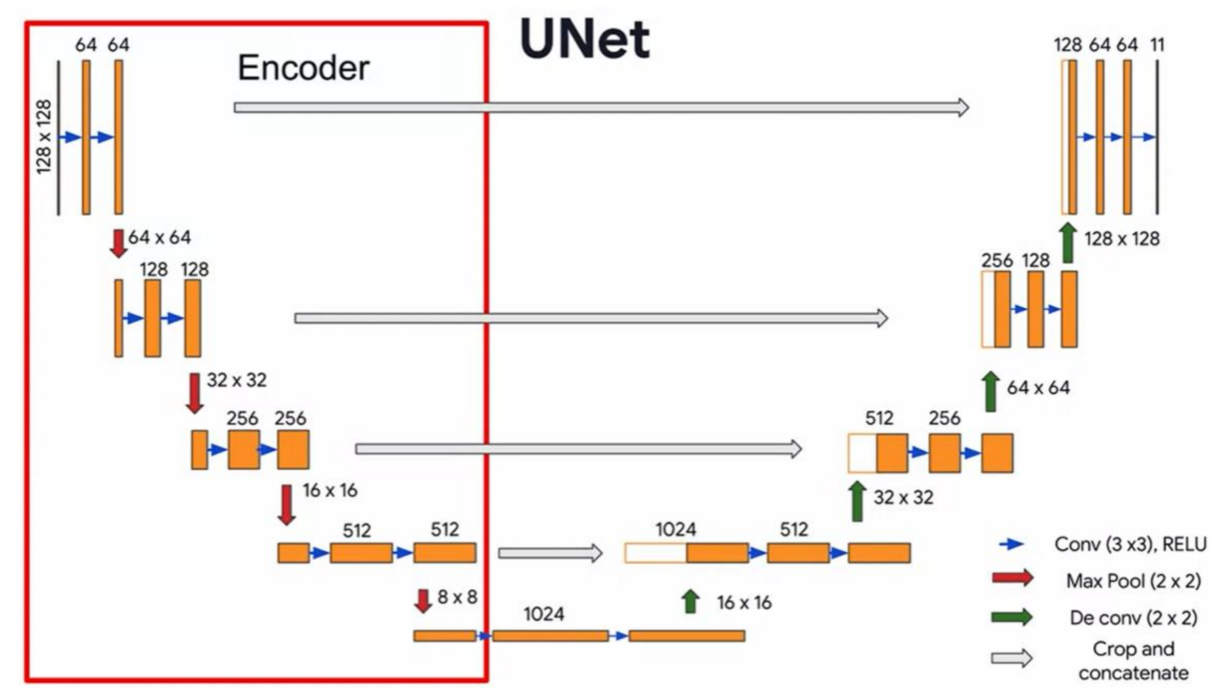
Calculate Dice Score

```
def class_wise_metrics(y_true, y_pred):
  class_wise_dice_score = []

  ...
  for i in range(n_classes):
    ...
    dice_score = 2 * (intersection) / (combined_area)
    class_wise_dice_score.append(dice_score)

  return class_wise_dice_score
```

Dice Score Results

sky	0.9350185576821789
building	0.861760180856767
column/pole	9.15750915331616e-10
road	0.9559097147402678
side walk	0.8396129387346395
vegetation	0.8677883515092748
traffic light	6.040349126864078e-05
fence	0.012679586065167121
vehicle	0.45563416820437186
pedestrian	0.0002528346886971326
byciclist	0.04615362426586659
void	0.2826172572009191

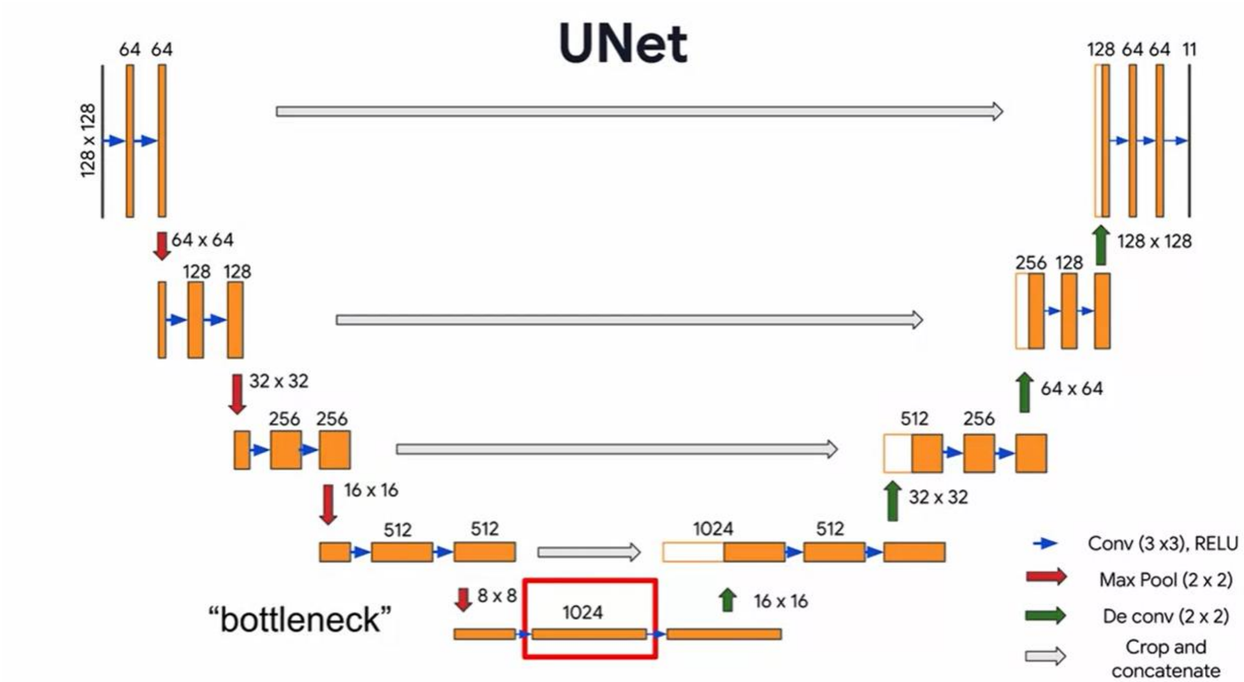




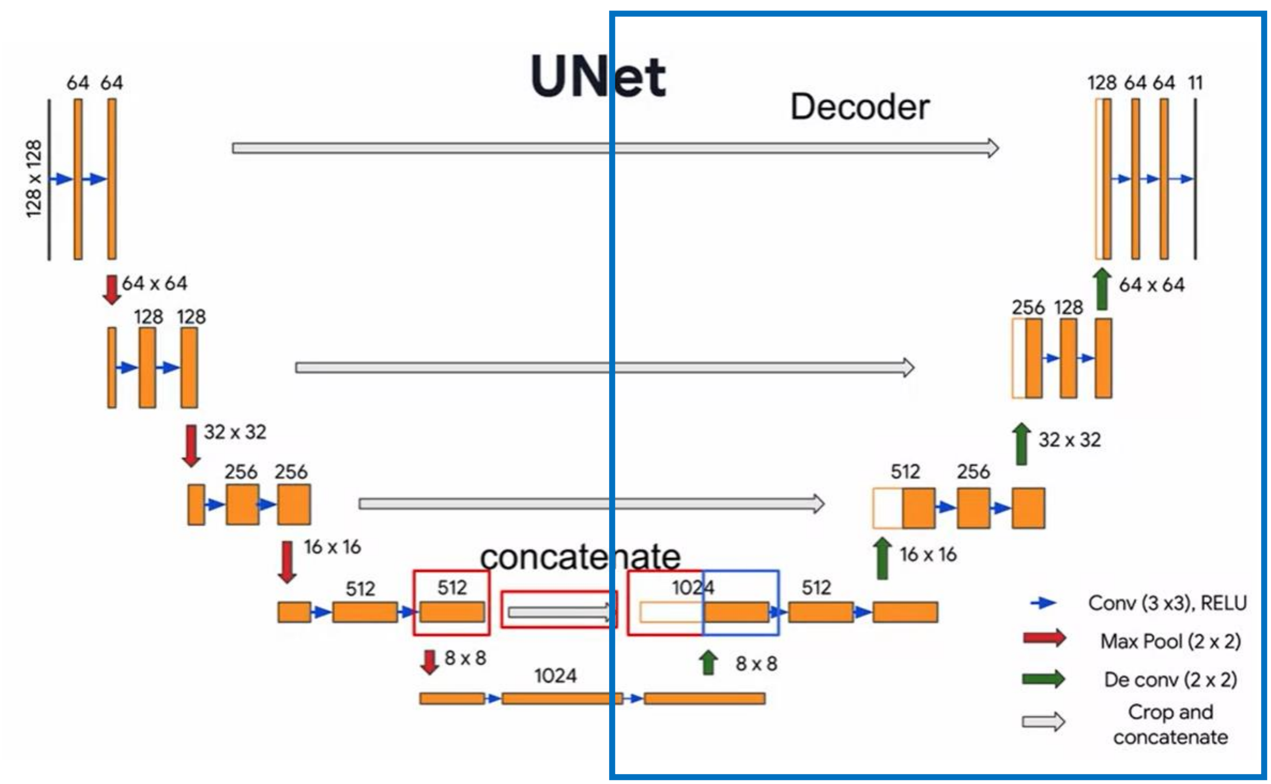
First is the encoder path, on the left of the U.
Here, similar to the FCNs, we have an image fed into convolution layers and then down sampled using max pooling.
So, for example at the 1st level, if the image is fed in as a 128 by 128 and pass through two convolutional layers that have 64 filters each, the subsequent images when pooled, will be 64 by 64.

Note that this isn't because of the number of filters, but because a max pooling layer with a 2 by 2 window and a stride of 2, can reduce the dimensionality by half. So 128 by 128 becomes 64 by 64.

In the 2nd level, the images are passed through two layers of convolutions with 128 filters each and they are then pulled from 64 by 64 to 32 by 32. In the 3rd level, the 32 by 32 matrices are passed through two layers of 256 filters each and then pulled to 16 by 16. And in the 4th level, the 16 by 16 images are fed through two layers of 512 filters each, before being pulled into an 8 by 8 at the fifth level.
Thus from the 1st level to the 5th, a 128 by 128 image is filtered and down sampled into 8 by 8 blocks.



An additional element in the unit architecture, is a simple convolutional layer, which can further extract features, but it doesn't have a pooling layer to follow it. This convolutional layer is named the bottleneck. Data flows through that, before we get to the decoder side of the ear.



And that's here and this is the right hand side of the U shape. Starting at the bottom, the 5th level with the 8 by 8 blocks gets up samples to 16 by 16. So, now were moving up to the 4th level of the U shape, but notice what happens next.

You'll take the 512 filters from the layer of the encoder that's at the same level as this decoder layer, in this case it's the 4th level. Notice that since the encoder layer and decoder layer are the same level in the unit, they also have the same height and width of 16 by 16 and they also have the same number of filters at 512 each.

So, you'll then concatenate the filters from the encoder, with the filters of the decoder, for a total of 1024 filters. You'll then pass this concatenated set of 1024 filters, through 2 convolutional layers.

And this pattern continues through the decoder, you'll up sample the blocks to 32 by 32 and move up to level 3. You'll take the filters from the encoder on the same level, concatenate them to the blocks from the decoder and pass the entire thing through 2 convolutional layers. And of course this works as well as up sampling the image, it will get fed through two layers of convolutional filters,
that match the one on the previous level. But with the additional ones that are the skip over, from the equivalent filters on the encoder side.

So up sampled to 64 by 64, move up to level 2, combine the filters from the encoder with a decoder and pass them through two convolution layers. And finally, up sample to 128 by 128 and move up to level 1, concatenate the filters from the encoder and decoder and pass them through the two convolution layers.