Advanced CNN for computer vision

Rodrigo Gonzalez, PhD

Hello!

I am Rodrigo Gonzalez, PhD

You can find me at

rodrigo.gonzalez@ingenieria.uncuyo.edu.ar

Summary

1. The different branches of computer vision: image classification, image segmentation, object detection
2. Modern convnet architecture patterns: residual connections, batch normalization, depthwise separable convolutions

1.

Computer vision tasks

The goals of a CNN

Computer vision tasks

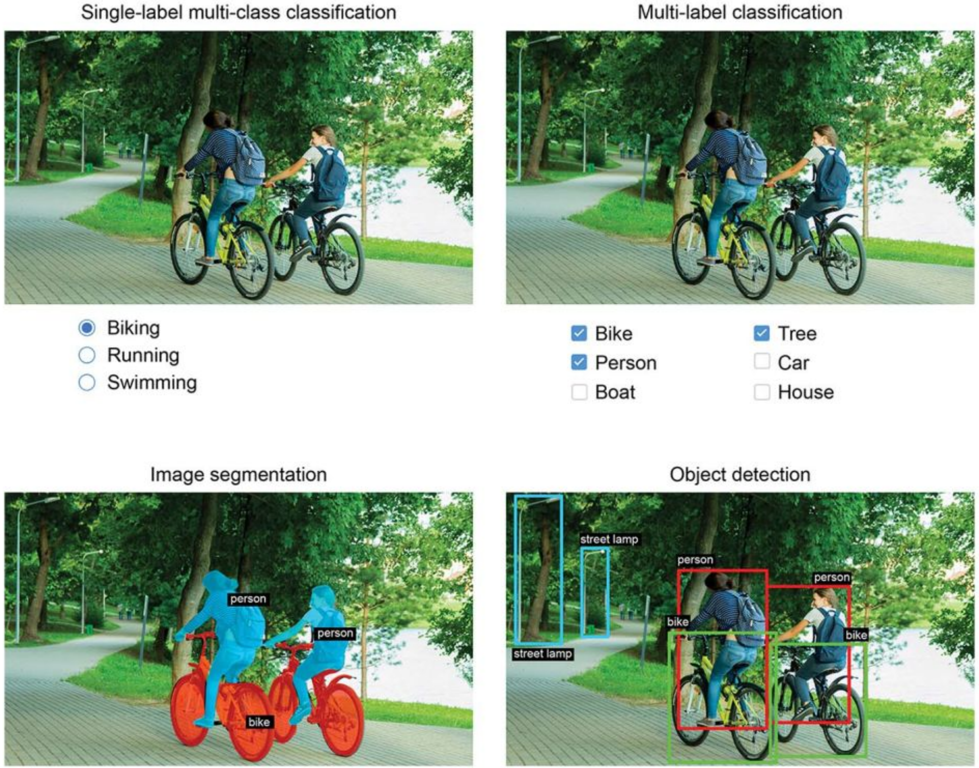


Figure 9.1 The three main computer vision tasks: classification, segmentation, detection

Image segmentation

1 Foreground 2 Background 3 Contour

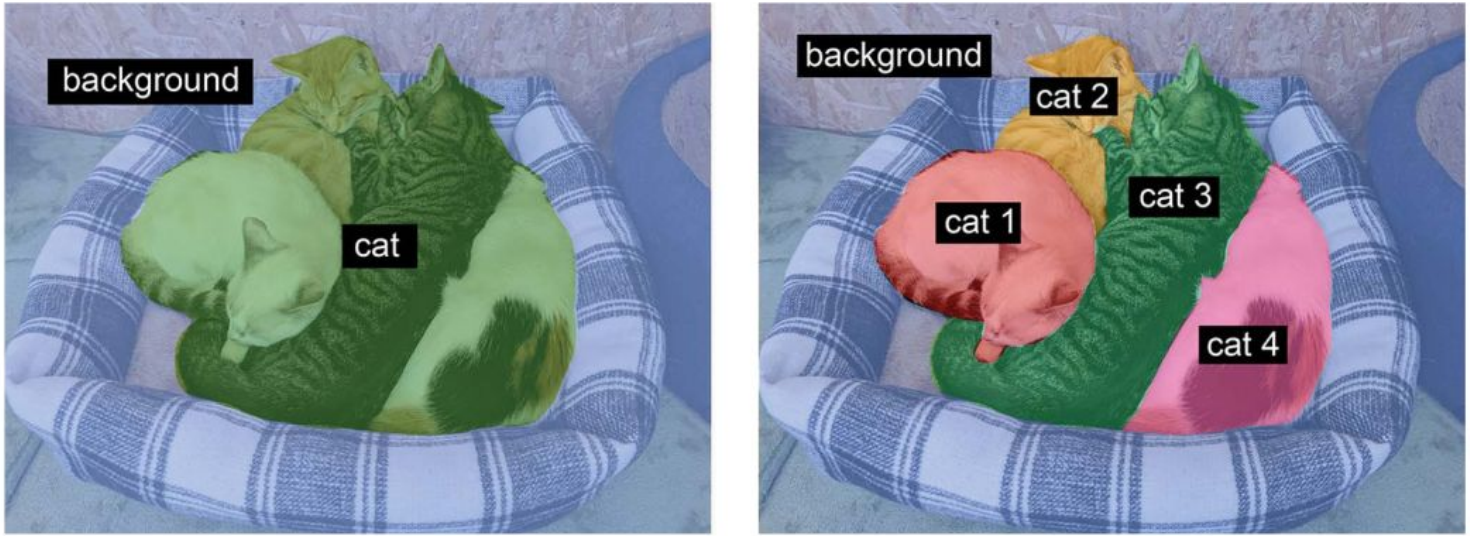


Figure 9.2 Semantic segmentation vs. instance segmentation

Image segmentation

Target mask

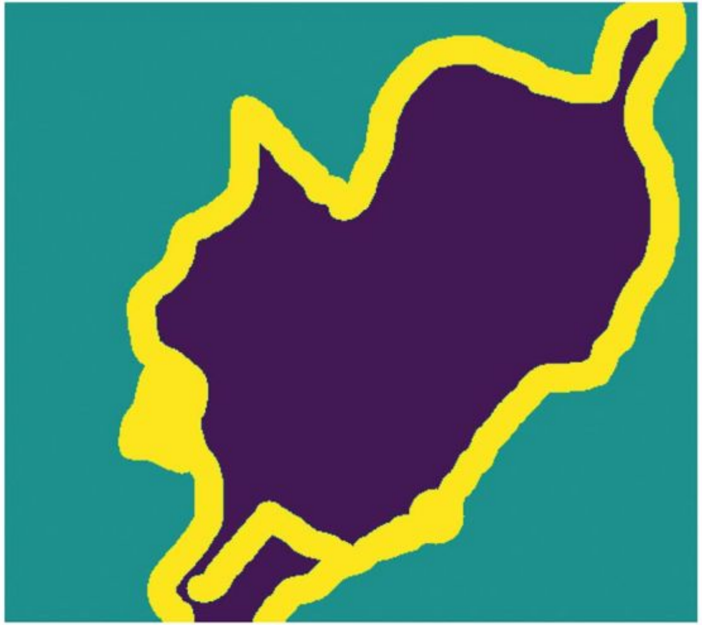


Image segmentation

The pixels of our segmentation masks can take one of three integer values:

1 (foreground)

2 (background)

3 (contour)

In the case of image segmentation, we care a lot about the spatial location of information in the image, we use stride instead of maxpooling

Conv2DTranspose layer

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way

Conv2DTranspose layer

The output of the first half of the model is a feature map of shape (25, 25, 256).

But we want our final output to have the same shape as the target masks, (200, 200,3)

We need to apply a kind of inverse of the transformations

Upsampling

Conv2DTranspose layer is a kind of convolution layer that learns to upsample.

```
(100, 100, 64) --> Conv2D(128,3,strides=2,padding="same") --> (50, 50, 128) -->
Conv2D-Transpose(64,3,strides=2,padding="same") -->  (100, 100, 64)
```

2.

Modern convnet architecture patterns

Model architecture

A good model architecture is one that reduces the size of the **search space** or otherwise makes it easier to converge to a good point of the search space.

Model architecture is all about making the problem simpler for **gradient descent** to solve.

Model architecture is more an **art** than a science.

3.

Modularity, hierarchy, and reuse

The VGG16 architecture

The number of filters grows with layer depth, while the size of the feature maps shrinks accordingly.

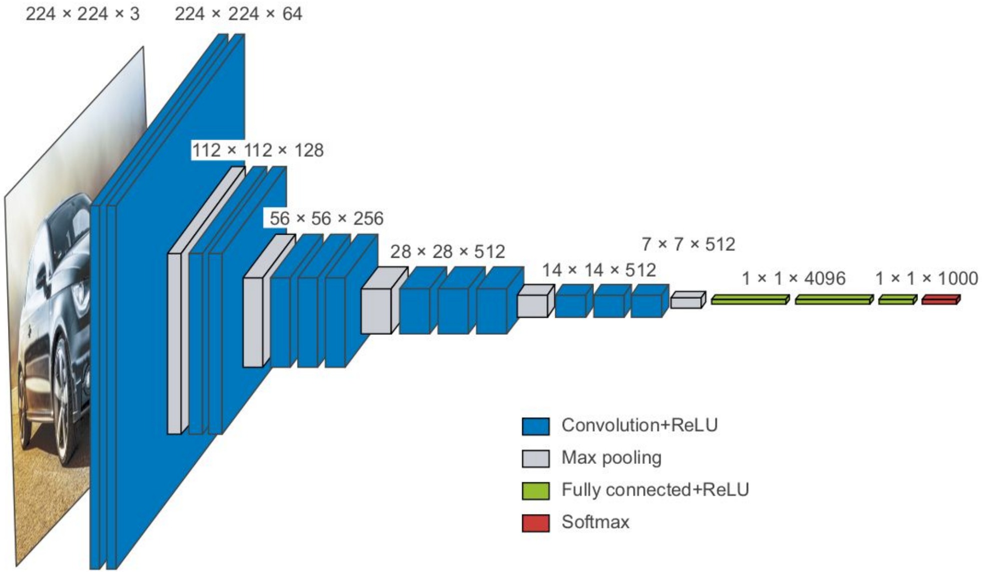


Figure 9.8 The VGG16 architecture: note the repeated layer blocks and the pyramid-like structure of the feature maps

The vanishing gradients problem

- Game of Telephone: The final message ends up bearing little resemblance to its original version
- Backpropagation in a sequential deep learning model is pretty similar to the game of Telephone

```
y = f4(f3(f2(f1(x))))
```

The vanishing gradients problem

- The *name of the game* is to adjust the parameters of each function in the chain based on the error recorded on the output of f4 (the loss of the model).
- To adjust f1, you'll need to percolate error information through f2, f3, and f4.
- However, each successive function in the chain introduces some amount of noise.
- If your function chain is too deep, this noise starts overwhelming gradient information, and backpropagation stops working.

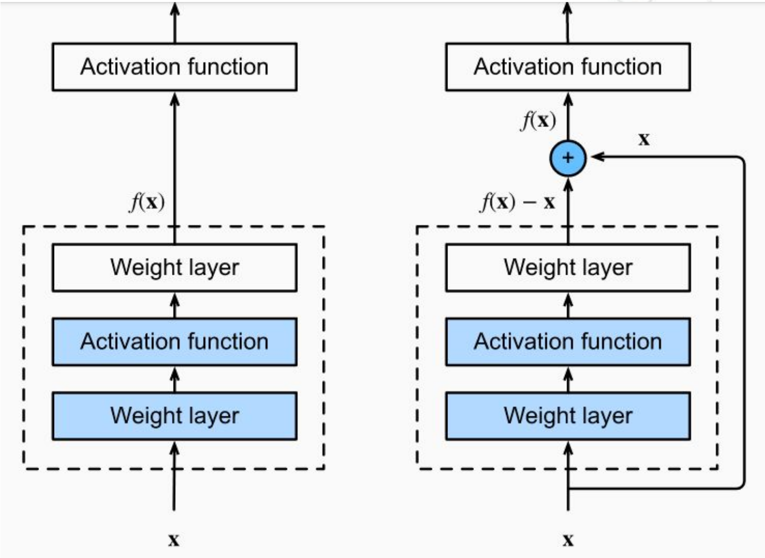
4.

Residual connections

A residual connection

To solve the vanishing gradients problem, just force each function in the chain to retain a noiseless version of the information contained in the previous input.

This technique was introduced in 2015 with the ResNet family of models.



A residual connection

Listing 9.1 A residual connection in pseudocode

```
x = ...
residual = x
x = block(x)
x = add([x, residual])
```

Some input tensor

Save a pointer to the original input. This is called the residual.

This computation block can potentially be destructive or noisy, and that's fine.

Add the original input to the layer's output: the final output will thus always preserve full information about the original input.

A residual connection

Listing 9.1 A residual connection in pseudocode

Some input tensor

Save a pointer to the original input. This is

```
x = ...
residual = x
x = block(x)
x = add([x, residua
```

Add the original input to t
output: the final outpu
always preserve full inf
about the origi

Listing 9.2 Residual block where the number of filters changes

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.

A residual connection

Listing 9.1 A residual connection in pseudocode

Some input tensor

Save a pointer to the original input. This is

```
x = ..
residu
x = bl
x = ad
```

Add the output alway

Listing 9.3 Case where the target block includes a max pooling layer

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

Now the block output and the residual have the same shape and can be added.

5.

Batch normalization

Batch normalization

In previous examples, data was normalized before feeding it into models.

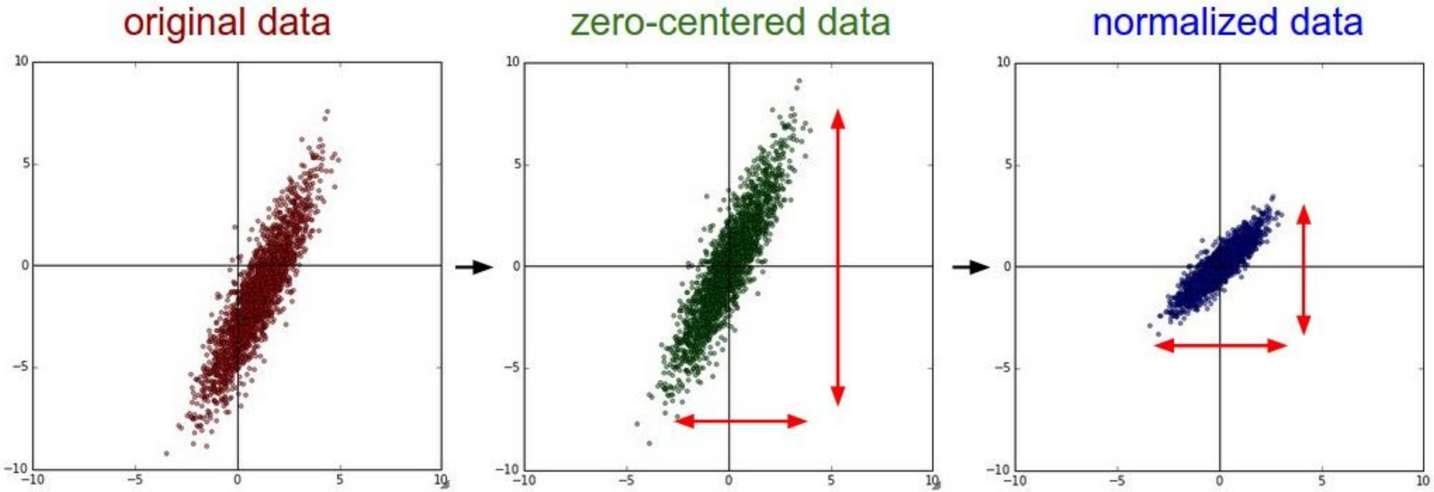
```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

Even if the **data entering** a Dense or Conv2D network has a 0 mean and unit variance, there's no reason to expect a priori that this will be the case for the **data coming out**.

Layer `BatchNormalization`in Keras do the job.

The main effect of batch normalization **appears** to be that it helps with gradient propagation.

Batch normalization



Batch normalization

Because the normalization step will take care of centering the layer's output on zero, the **bias vector** is **no longer needed** when using `BatchNormalization`.

```
x = ...
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
```

Recommended setup:

```
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
x = layers.Activation("relu")(x)
```

We place the activation after the BatchNormalization layer.

Batch normalization will center your inputs on zero, while your relu activation uses zero as a pivot for keeping or dropping activated channels

6.

Colab Notebook

Colab Notebook

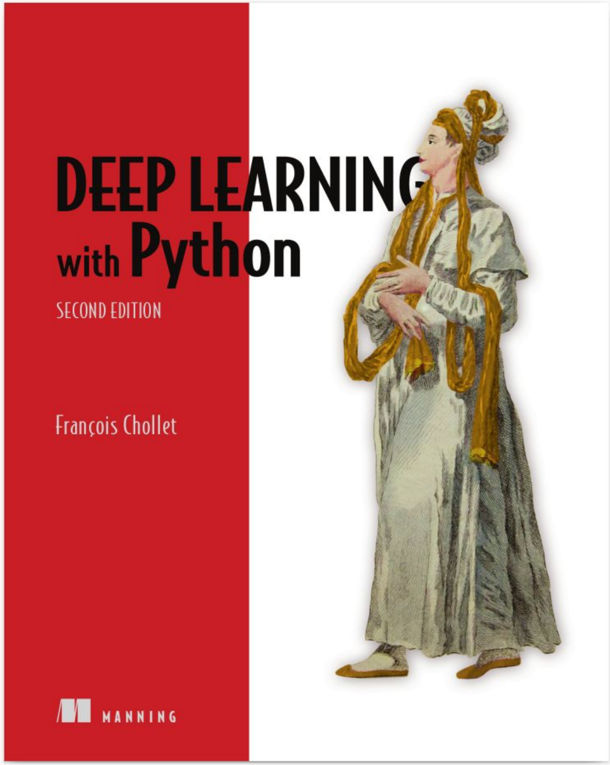
Chapter 9, Advanced deep learning for computer vision:

https://colab.research.google.com/drive/1-QQ_myOpJIbXZCubeD3HP9eQUmea2mSb

7.

Book

Deep Learning with Python, 2nd Ed. by Francois Chollet



- Chapter 9

Thanks!

Any questions?