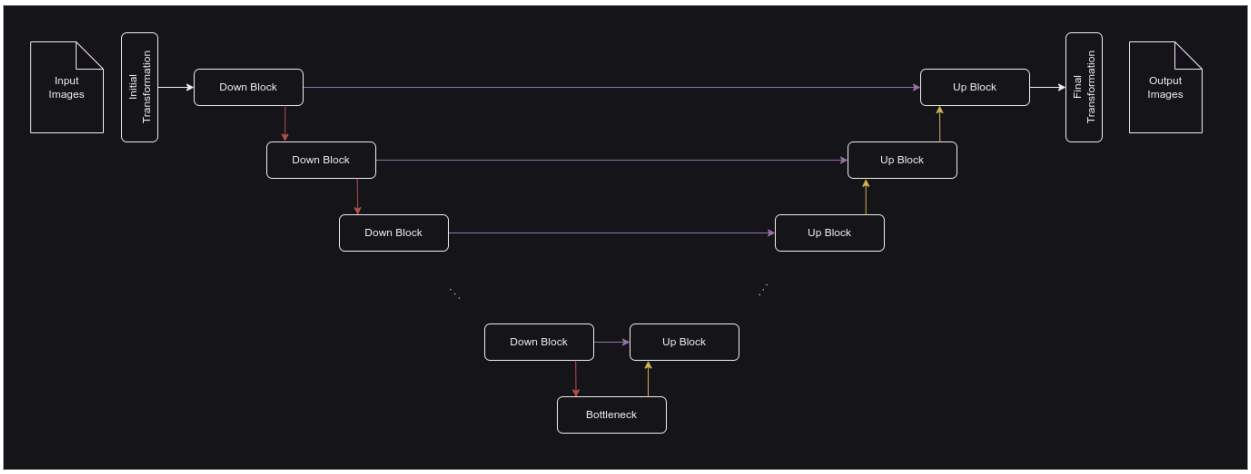
1. Our Own UNet

In this practical, we will be coding our own UNet using Keras and custom layers.

1.1. The UNet Architecture



UNet is a well known Neural Network architecture with a ton of uses, it consists of:

1. An Encoder path, that will generate multiple views of the input image at multiple resolutions.
2. A Decoder path, that will be responsible for transforming

One of the difficulties of this architecture is the long-range residuals that make the network highly non sequential. We will go step by step.

2. A base Code

2.1. Dataset

A basic code base is provided in order to focus only on the UNet architecture. A small dataset is provided, it consists of photographic images of people's hand and the objective is to produce a network that will be able to do automatic nail segmentation.

The dataset comprise of two directories:

1. The raw images
2. The segmentation masks

2.2. Datagenerator

A basic datagenerator python file is provided, it is not in the scope of this practical to see how it works, but I encourage you to look at it.

2.3. Predictions

Another file `tp5_predict.py` is provided that will be used to test the models you build on your own images.

It takes 4 arguments

- `-i` the input image filepath.
- `-o` the output image filepath.
- `-m` the tested model directory path.
- `-d` the depth of the model.

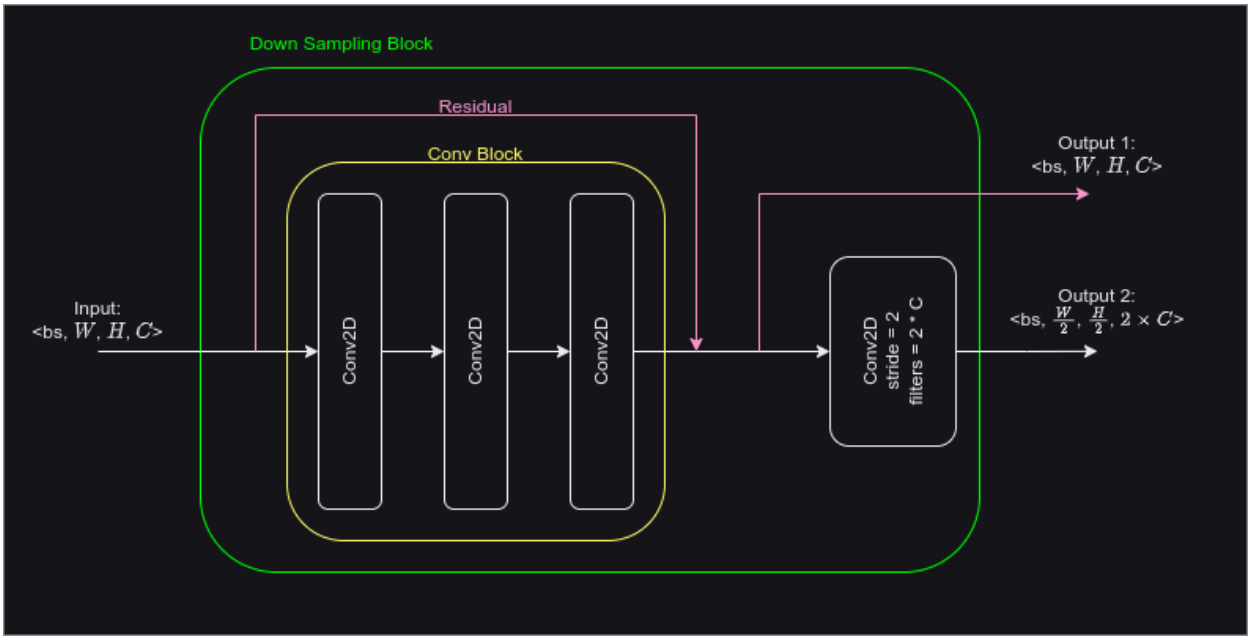
2.4. Utils

A `Utils` file is provided but the classes it serves are missing their core bodies. It's one of your job in this practival to fill in the missing parts.

2.5. Start

Finally, a `tp5_start.py` file is provided as the file that will generate the model, train it and save it. One of your jobs is to fill in as well the missing parts of the model structure.

3. Encoder Blocks



Encoder Blocks (or Down Sampling Blocks) are made of a residual connection around a Convolution Block, followed by a special `stride = (2, 2)`, `filters = 2xC` convolution that serves the purpose of down sampling the image data.

This layer has two outputs that need to be taken into account:

1. The “original-sized image” just following the residual block, that will be concatenated further in the structure.
2. The Down sampled data that will be used in the next layer.

3.1. The Residual Convolution Block

Fill in the custom layer class that we will be using as our residual convolution block. The exact content of the block is not that important (number of convolutions, kernel_sizes, etc…) what matters is that you will be able to change it easily with a single customization point. For now, you can reuse the custom layer from last practical.

3.2. The Down Sampling Layer

This one easy, it consists of only one convolution with a stride of 2. You can write a custom layer if you want to be fancy about it, but for simplicity sake’s we will stay with the single convolution. Write the corresponding conv2d layer.

3.3. Putting Things Together

Now that the first and second parts of the layer are done, let’s use them together to form the Encoder Block. As a reminder, you can have multiple outputs in a custom layer, for example:

```
import tensorflow as tf

class MultipleOutputCustomLayer(tf.keras.layers.Layer):
    def call(self, inputs):
        return inputs, inputs * 2
```

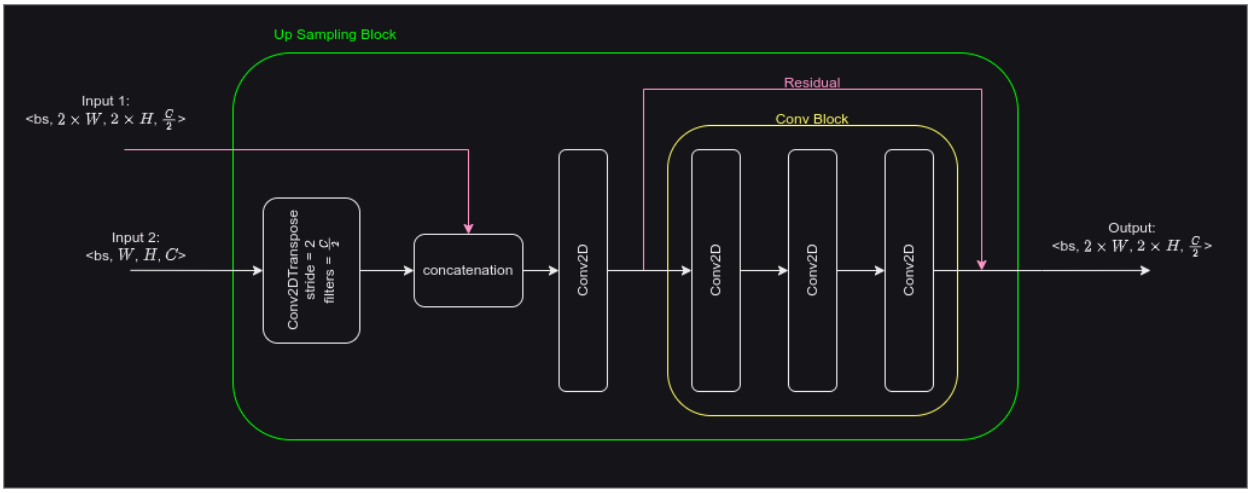
And you can collect the outputs separatly like this:

```
... # previous layers

X, doubleX = MultipleOutputCustomLayer()(previous_layer)

... # rest of the code
```

Here, `X` and `doubleX` will create two divergent layer (like the “forking” at the start of a residual path).

Fill in the custom `UNetEncoder` layer that will call a `ResidualConv2D`, down sample the result and return the two corresponding outputs.

4. Decoder Block



4.1. Transposed Convolution

The transposed convolution is essentially the inverse of a standard convolution. You can find more informations here: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose.

4.2. Multiple Inputs

The Decoder block takes two inputs and generate only one outputs.

Here is an example of a two inputs custom Keras layer:

```
class TwoInputCustomLayer(tf.keras.layers.Layer):
    def call(self, inputs):
        input1, input2 = inputs
        return input1 + input2

    def build(self, inputs_shape):
        input_shape1, input_shape2 = inputs_shape
        ... # rest of the code
```

The way you use it is as follow:

```
... # previous layers

# input1 and input2 are two previous layers,
# don't forget to enclose them in a list !
X = TwoInputCustomLayer()([input1, input2])

... # rest of the code
```

4.3. The Concatenate Layer

As always, I encourage you too go and look for the documentation for yourself: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Concatenate.

The Concatenate layer will "fuse" the two inputs layer altogether along a given axis (here the last one).

4.4. All In One

Fill in the custom "Decoder" layer that take two inputs and generate the output following the provided schematics.

5. The Whole Picture

You are now ready to write the missing parts in the `tp5_start.py` file for the model code and complete the structure.

5.1. DEPTH

The `DEPTH` constant is the number of Down sampling and Up sampling will the network go.

As a hint, you can iterate over a python list in reverse order using the following:

```
mylist = [1, 2, 3, 4, 5, 6]

for element in mylist[::-1]:
    print(element)
    # will print 6, 5, 4, 3, 2, 1
```

Complete the missing code part and run a training and save a model