# Deep Learning

# Practical 3

# 1. From last Practicals

The base toy model we used from last two practicals is:

```python
import tensorflow as tf
import numpy as np

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train, x_test = x_train / 255.0 , x_test / 255.0

input_layer = tf.keras.layers.Input(name="input_layer", shape=(28, 28))
flatten_layer = tf.keras.layers.Flatten()(input_layer)
hidden_layer = tf.keras.layers.Dense(units=128)(flatten_layer)
hidden_layer = tf.keras.layers.Activation("relu")(hidden_layer)
hidden_layer = tf.keras.layers.Dropout(0.2)(hidden_layer)
hidden_layer = tf.keras.layers.Dense(units=10)(hidden_layer)
output_layer = tf.keras.layers.Activation(
    "softmax", name="output_layer"
)(hidden_layer)

model = tf.keras.models.Model(inputs=[input_layer], outputs=[output_layer])

model.summary(150)

model.compile(
    optimizer = "Adam",
    loss = {
        "output_layer": tf.keras.losses.SparseCategoricalCrossentropy()
    },
    metrics = ["acc"],
)

model.fit(
    x_train,
    y_train,
    validation_data=(x_test, y_test),
    batch_size = 32,
    epochs = 5
)
```

The objectif of this practical is to transform it into an encoder/ Decoder Sequentialy Separated Convolutional Residual Variadic Neural Network !

Let's proceed step by step.

**Provide a working `.py` file for each part.**

# 2. Convolutional Transformation

### 2.1. The Idea

The idea here is to transform the existing dense layers into convolutional layers. Convolutional layers are more suited for image data as they can extract spatial hierarchies of features from the images.

### 2.2. Methodology

1. Input Shape: Change the input shape to include the channel dimension. For MNIST, this would be (28, 28, 1) as the images are grayscale.
2. Convolutional Layers: Replace dense layers with convolutional layers (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D). You'll still need a Flatten layer before the final Dense layer to convert the multi-dimensional convolutional output to a flat form.

Replace the first dense layer with a convolutional layer with `kernel_size = (1, 1)` and `filters = 128`. For now, keep the Flatten and final Dense layer.

Add 3 more convolutions, Use 7×7 kernels, 128 filters and `padding = "same"` for this layers.

## 3. Variadic Transformation

### 3.1. The Idea

The goal is to make the network capable of handling images of varying sizes.

### 3.2. Methodology

1. Input Shape: Change the input shape to have `None` for the width and height dimensions, allowing for variability in image size.
2. GlobalAveragePooling2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalAveragePooling2D): This layer will help to transform variably shaped inputs into a fixed-size vector, making it possible to output a fixed-size probability vector.

Change input shape to `(None, None, 1)`.

Change the final Dense layer by a Conv2D layers with 10 filters. Include a GlobalAveragePooling2D layer just after to have a 10-sized vector.

# 4. Residual

## 4.1. The Idea

The concept behind adding residual layers is to enable the learning of identity functions, which aids in addressing the vanishing gradient problem in deep neural networks. By incorporating residuals, or “shortcut connections”, around each convolutional block, the network becomes fully residual. These connections allow the network to preserve information from earlier layers and pass it through deeper layers without significant degradation.

## 4.2. Methodology

For each convolutional block, you’ll introduce a shortcut that bypasses the block and adds its input directly to its output. This can be done as follows:

1. Before a convolutional block, save the input to that block.
2. After the convolutional block, add the saved input to the output of the block.

Here’s a conceptual example:

```
input_to_block = x  # Save input before the block
# Convolutional block
x = tf.keras.layers.Conv2D(...)(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.Activation("relu")(x)

# Adding the residual
x = x + input_to_block
```

Repeat this pattern around each convolutional block to ensure the entire network is fully residual from top to bottom.

# 5. Sequentialy Separated

## 5.1. The Idea

Sequentially separated convolutions involve breaking down larger convolutions into smaller, more manageable parts, which can increase depth and reduce parameters.

## 5.2. Disclaimer

We will only be dealing with square odd-sized kernels, but the methodology is the same for any sized kernel, just do it separatly for the two dimensions and use a 2-sized kernel for the final one in case of even-sized requests.

### 5.3. Methodology

1. Get the receptive field ($\varphi$) size you want to achieve (for example 9x9 gives $\varphi = 9$)
2. Compute the number of 3x3 convolutions needed to achieve such receptive field, using the formula: $n = \frac{\varphi - 1}{2}$ (here $n = 4$)
3. Replace the $\varphi$-sized kernel convolution by $n$ 3x3-sized kernel convolutions. In our study case, replace

```
x = tf.keras.layers.Conv2D(filters=c, kernel_size=(9, 9), padding="same")(x)
```

   by

```
for _ in range((9-1)//2):
  x = tf.keras.layers.Conv2D(filters=c, kernel_size=(3, 3), padding="same")(x)
```

4. Add activations, normalizations and regularization as you see fit. For example:

```
for _ in range((9-1)//2):
  x = tf.keras.layers.Conv2D(filters=c, kernel_size=(3, 3), padding="same")(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation("tanh")(x)
  x = tf.keras.layers.Dropout(0.2)(x)
```

5. Replace all 3x3-sized convolution by a 3x1 followed by a 1x3:

```
for _ in range((9-1)//2):
  x = tf.keras.layers.Conv2D(filters=c, kernel_size=(3, 1), padding="same")(x)
  x = tf.keras.layers.Conv2D(filters=c, kernel_size=(1, 3), padding="same")(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation("tanh")(x)
  x = tf.keras.layers.Dropout(0.2)(x)
```

   In your code, change all convolutions to Sequentialy Separated convolutions !

## 6. Encoder/Decoder Convolutions Blocks

### 6.1. The Idea

Apply a channel reduction transformation before applying the sequential separation and transform back to the original size at the end to sum for the residual block, thus lowering the number of parameters of the network.

### 6.2. Methodology

1. Add a 1x1-sized kernel convolution at the beginning of each convolution block, with a low `filters` value in order to drasticaly lower the computation size (like $n = 32$).
3. Change all `filters` values in the convolutions that will be compressed to the choosen $n$.
2. Add a 1x1-sized kernel convolution at the end of each convolution block, with the original number of `filters` value in order to get back the original number of channels and be able to add them together for the residual block.

Apply this methodology for each sequentialy separated convolution block in your code, try different value for $n$ and see what works best.

## 7. Cifar-10

Update your code to run on the Cifar-10 dataset (https://www.tensorflow.org/api_docs/python/tf/keras/datasets/cifar10/load_data)