# → PROBA-V Super Resolution

(Enhance the vegetation payload performances, competition)

## Problem:

As a part of this assignment, we have been asked to participate in **PROBA-V** super-resolution image competition organized by the **European Space Agency**. PROBA-V is an ESA earth observation satellite that is designed to track vegetation growth across the entire earth. It has the ability to capture **300m** low-resolution images as well as **100m** high-resolution images. But the capacity of taking low-resolution images is quite high than that of high-resolution images (taken every 5 days). We have been given a data set that consists of both low-resolution images and high-resolution images taken from various earth locations. We have been asked to combine all low-resolution images to generate a high-resolution image. We are open to trying any tactic to combine low-resolution images to generate a high-resolution image.

**Source:** **https://kelvins.esa.int/proba-v-super-resolution/home/**

## Data:

Data consists of **74 hand-selected** earth regions around the globe for which both low-resolution 300m and high-resolution 100m images are collected. In total, there are 1450 image sets where each image set consists of images from one location which are split into **1160 scenes for training** and **290 scenes for testing**. Each image set in training data consists of on average 16 low-resolution images(**128x128** pixels) and one high-resolution image(**384x384** pixels). Each low resolution and the high-resolution image has a quality map. The quality map is a map that represents which part of the image is proper to consider for analysis. We have been asked to ignore high-resolution images with less than **75%** clear pixels and low-resolution images with less than **60%** clear pixels. Data has been give for RED and NIR channels.

## Technologies:

We have performed this whole analysis in **Jupyter notebook** with **python** as our language of choice. We have used image processing library **scikit-image** for loading images and **PyTorch** for designing and training deep learning models. We also have used a few supporting libraries like Pillow, matplotlib, numpy, zip file, warnings, os, and collections. We have designed the whole analysis to be **reproducible** in the jupyter notebook. As analysis uses complicated **deep learning models**, it's recommended to have **GPU** for training to complete training fast.

## Approaches:

We have started with visualization of low-resolution images as well as high-resolution images for a few images in trainset to get an idea about the quality of low-resolution as well as high-resolution images. We have then designed function which splits train data into training and validation sets. We have kept 98% of the training data(1136 image sets) for training and 2% of training data for

validation (24 image sets). Function also returns us the path to each image in train and validation sets which can then be processed by deep learning models by reading them from disk.

We have tried than 3 different Deep Learning models for our purpose:

1. **DCGAN (Deep Convolutional Generative Adversarial Network)**
2. **CNN with Residual Blocks v1**
3. **CNN with Residual Blocks v2**

We'll be explaining below the architecture of each model as well as the training process in detail for each one of them as well as the evaluation process.

## 1. DCGAN

DCGAN approach consists of 2 neural network models competing against each other. It has 2 neural nets in process. (**1. Generator 2. Discriminator.)** Generator CNN is responsible for processing low-resolution images to high-resolution images whereas discriminator is responsible for guessing the newly generated high-resolution image is taken from that same high-resolution camera or not. Generator takes as input low-resolution 300m images and processes them to generate high-resolution images. Discriminator takes as input high-resolution image generated by the generator and predicts whether it's taken from the same 100m high-resolution image cam or not. Our training process has been trained for 2 epochs as error rate has dropped completely after 2 epochs. Generator in the process tries to generate an as accurate high-resolution image as possible to confuse discriminator into thinking that it's generated from the same camera which generated high-resolution images.

We have taken into consideration the **BCELoss** function which is used in binary classification problems to calculate losses for the discriminator. We have used **Adam** optimizer for our process with a very low **learning rate** of **0.0002** as we don't want to modify original images too much so that it loses important information.

**Generator Architecture:**

Generator architecture consists of **6 convolutional transpose** layers. Except for the first and last convolutional layers, each one is followed by the **batch norm** layer. The first convolutional layer is followed by **Relu** and the last convolutional layer is followed by **Tanh** to keep output in the range[-1,1]. After each batch norm layer also there exist one Relu layer. Each convolutional transpose layer has a window size of **3x3** with a stride of one. We also have added padding of 1 pixel in the image so that input and output images have the same size. We also have followed a particular input-output channel structure which goes in this sequence (**1->3->6->9->6->3->1**) from layer to layer.

```
Generator(
  (model): Sequential(
    (0): ConvTranspose2d(1, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): ReLU(inplace=True)
    (2): ConvTranspose2d(3, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (3): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ReLU(inplace=True)
    (5): ConvTranspose2d(6, 9, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (6): BatchNorm2d(9, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): ReLU(inplace=True)
    (8): ConvTranspose2d(9, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (9): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): ConvTranspose2d(6, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (12): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): ConvTranspose2d(3, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (15): Tanh()
  )
)
```

**Discriminator Architecture:**

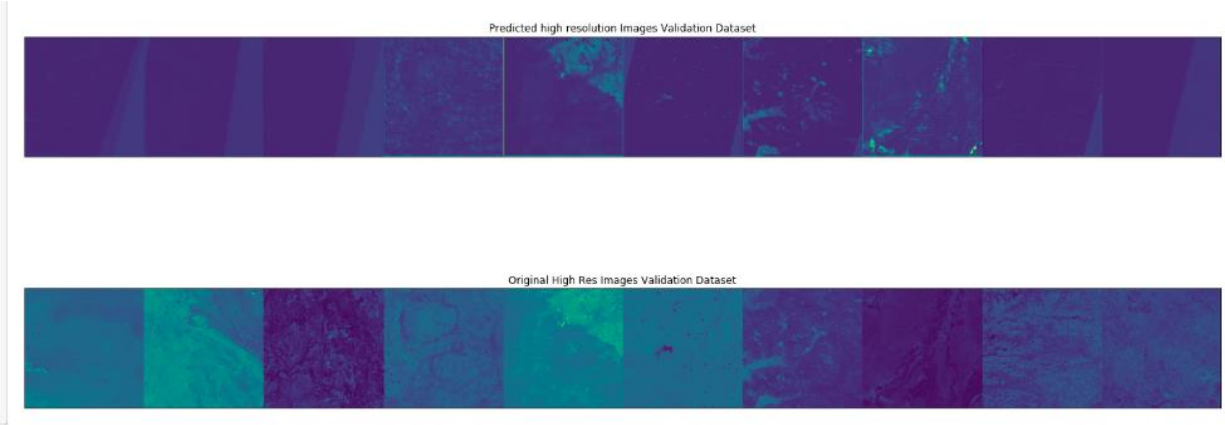
Discriminator architecture consists of **7 convolutional layers** which are followed by **batch norm** layers except for the first and last convolution layers. The first convolution layer is followed by **Relu** whereas the last convolution layer is followed by a sigmoid function. Each batch norm layer is followed by relu layers for intermediate convolution layers. **Sigmoid** layer output probability between 0-1. Each convolutional layer has a window size of **3x3** with a stride of 2 except the last convolutional layer which has a window size of **6x6** and stride of 1 which was done purposefully to get dimensions matching. We have also kept padding of 1 to keep image size the same as the original. We also have followed a particular input-output channel structure which goes in this sequence (**1->3->6->9->6->3->2->1**) from layer to layer.

```
Discriminator(
  (model): Sequential(
    (0): Conv2d(1, 3, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): ReLU(inplace=True)
    (2): Conv2d(3, 6, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ReLU(inplace=True)
    (5): Conv2d(6, 9, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(9, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): ReLU(inplace=True)
    (8): Conv2d(9, 6, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(6, 3, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (12): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): Conv2d(3, 2, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (15): BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(2, 1, kernel_size=(6, 6), stride=(1, 1), bias=False)
    (18): Sigmoid()
  )
)
```

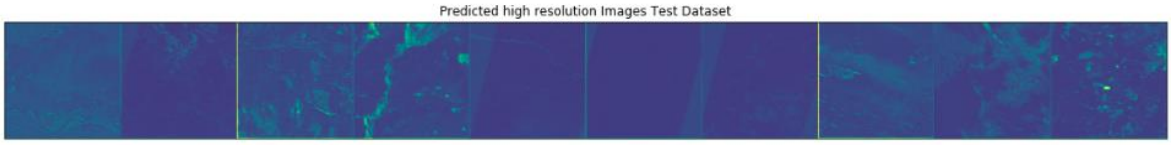
The training process goes on for 2 epochs with standard GANs training which starts with training discriminator with a real high-resolution image and then trains it using image generated through generator using the low-resolution image. It then trains discriminator again using low-resolution images so that it can better differentiate between processed high-resolution and actual high-resolution images. We have also resized all low-resolution images from 128x128 to 384x384 size before giving it to models as that is the resolution of high-resolution images. We loop through training data twice for each epoch. We loop through training data with one image set as a time. We have designed a few helper methods which read images of one imageset and return us so that we can use them for our training. Once we have gone through all image sets of train data then we test its performance on validation data by plotting high-resolution images generated from low-resolution images of validation data. We combine all high-resolution images generated by the model for one imageset into one image by averaging them.

Once the training process has completed, then we use the trained model on test image sets to generate a high-resolution image for each imageset using low-resolution images. We then zip all high-resolution images generated for the test set so that it can be submitted as well. We have to save all images as PNG format of unit16 format which is required for the project. It takes around **35-40** mins for the training process to complete using **GPU**.

**Sample Validation Output:**




**Sample Test Output:**




## 2. CNN with Residual Blocks v1

```
CNNWithResBlocks(
  (res_layers): Sequential(
    (0): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (1): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (2): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (3): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (4): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (5): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (6): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (7): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
    (8): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
    )
  )
  (block): Sequential(
    (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): PReLU(num_parameters=1)
    (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): Tanh()
  )
)
```

CNN with the residual block **v1** is a Convolutional Neural Network based on the concept of the presence of more than 1 **residual block** in a neural network. Residual block is a block of few

**convolutional layers** followed by **relu** layers. The input image is passed through residual block layers and then the original output is added to the original image to generate a composite image.

Our model consists of such **9 such residual blocks** followed by convolutional, relu, convolutional and tanh layers. Tanh layer is responsible for keeping image values in a range of **[-1,1]**. As a part of convolutional layers, we have kept input and out channels for each Conv layers the same as 1. **Greyscale** images have only 1 channel and we have kept it constant across convolutional layers. All of our convolutional layers have a use window size of **3x3** which strides 1 step at a time. We also have used the padding of 1 pixel around images to keep image size the same as the input to avoid image resizing in the training process.

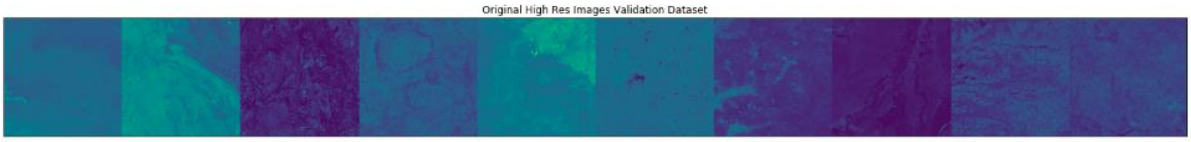
We have used **MSELoss** as our loss function for the model which tries to reduce mean squared error loss between predicted and original high-resolution images. We have used the same Adam optimizer as that of the DCGAN approach with a **learning rate** of **0.0002** for the training process.

Training process loops through training image sets for **2 epochs**. For each epoch, it loops through each image set of training data and predicting high-resolution images from low-resolution images. For each imageset, all high-resolution images generated from low-resolution images are combined by averaging over them. Once the model is trained using train data, it then tries to predict high-resolution images for validation data as well. We also plot the top few images generated for validation data.

Once the training process is completed then we use the same trained model to do the prediction of high-resolution images for test images. We also zip all test set images so that it can be submitted for ranking.

It takes around **50-60** mins for the training process to complete using **GPU**.

**Sample Validation Output:**


**Sample Test Output:**


## 3. CNN with Residual Blocks v2

```
CNNWithResBlocksv2(
  (res_layers): Sequential(
    (0): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
      (conv_block): Sequential(
        (0): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
    (1): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(4, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(4, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
      (conv_block): Sequential(
        (0): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
    (2): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
      (conv_block): Sequential(
        (0): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
    (3): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
        (2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): PReLU(num_parameters=1)
      )
      (conv_block): Sequential(
        (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
  )
  (deconv): Sequential(
    (0): ConvTranspose2d(32, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Tanh()
  )
)
```

CNN with the residual block **v2** is a Convolutional Neural Network based on the concept of the presence of more than 1 **residual block** in a neural network. It has almost the same concept as the previous v1 model with a slight twist in the case of channels used in convolutional layers.

Our model consists of such **4 such residual blocks** each followed by the convolution layer which increases the number of channels incrementally **(1->4->8->16->32)**. After the last residual block, we have one **convolution transpose layer** which reduces channels from 32 to 1 followed by the Tanh layer at last. Each residual block uses relu layers in-between convolutional layers. Tanh layer is responsible for keeping image values in a range of **[-1,1]**. All of our convolutional layers have a use window size of **3x3** which strides 1 step at a time. We also have used the padding of 1 pixel around images to keep image size the same as the input to avoid image resizing in the training process.

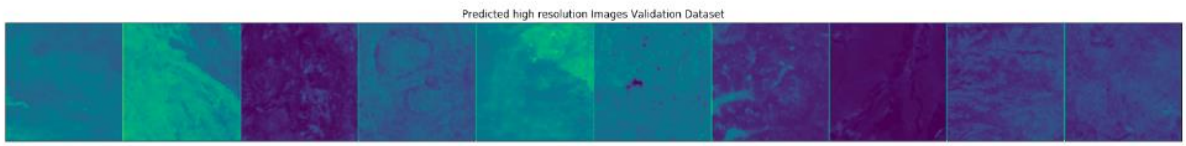
We have used **MSELoss** as our loss function for the model which tries to reduce mean squared error loss between predicted and original high-resolution images. We have used the same Adam optimizer as that of the DCGAN approach with a **learning rate** of **0.0002** for the training process.

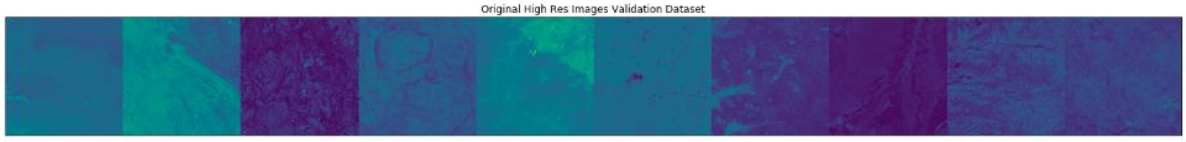
Training process loops through training image sets for 1 epoch. For each epoch, it loops through each image set of training data and predicting high-resolution images from low-resolution images. For each imageset, all high-resolution images generated from low-resolution images are combined by averaging over them. Once the model is trained using train data, it then tries to predict high-resolution images for validation data as well. We also plot the top few images generated for validation data.

Once the training process is completed then we use the same trained model to do the prediction of high-resolution images for test images. We also zip all test set images so that it can be submitted for ranking.

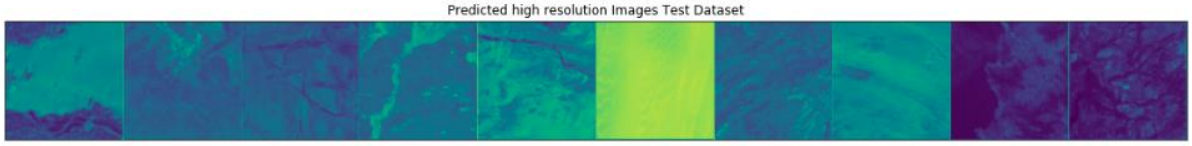
It takes around **50-60** mins for the training process to complete using **GPU**.

**Sample Validation Output:**




**Sample Test Output:**




**Conclusion:**

After trying all 3 approaches we have come to the conclusion that models with residual block did perform better than the DCGAN approach. Models with residual blocks kept the original image as intact as possible while increasing resolution which was not the case with DCGAN much though.