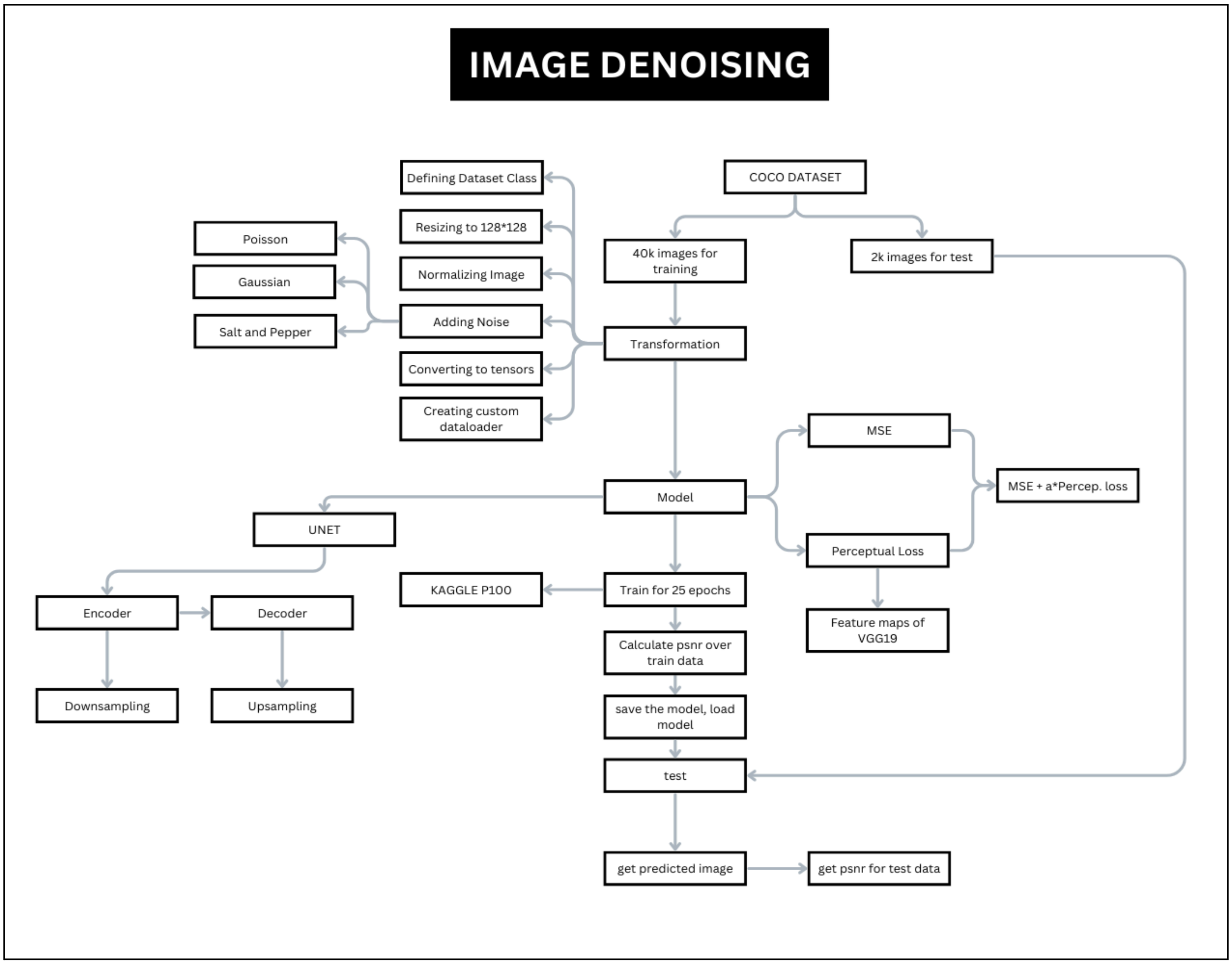
Image Denoising

Ayush Singh - 23410010 - ayush_s@mt.iitr.ac.in

TRAINED ON : KAGGLE P100 (128 batch size)
PAPER REFERRED :

- Perceptual Loss : https://arxiv.org/abs/1603.08155v1
- UNet Architecture : https://arxiv.org/abs/1505.04597

Note : Please read the training subtopic of the report before reading model.ipynb.



DATASET

https://cocodataset.org/#download

The dataset used for model training is the Microsoft Coco dataset. The dataset had various subsets so the one used for training containing 40k images is the test2014 and the dataset used for testing containing 2k images is the subset of val2017.

```
class NoiseImageDataset(Dataset):

    def __init__(self, image_paths, transform=None):
        self.image_paths = image_paths

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        target_size = (128,128)
        image = cv2.resize(image,target_size,interpolation=cv2.INTER_LINEAR)

        image = image.astype(np.float32) / 255.0

        noise_image = add_salt_and_pepper_noise(image)
        noise_image = add_poisson_noise(noise_image)
        noise_image = add_gaussian_noise(noise_image)

        image = torch.tensor(image).permute(2, 0, 1)
        noise_image = torch.tensor(noise_image).permute(2, 0, 1)

        return  noise_image.to(device) ,image.to(device)
```

The above code is for creating a custom dataset class which involves the following steps :

- The image is read using the **imread** function and then converted from BGR to RGB because in further steps when we visualize the dataset, the **imshow** function takes the image as RGB.
- Then we use **resize** function to resize images to a standard 128*128 size.

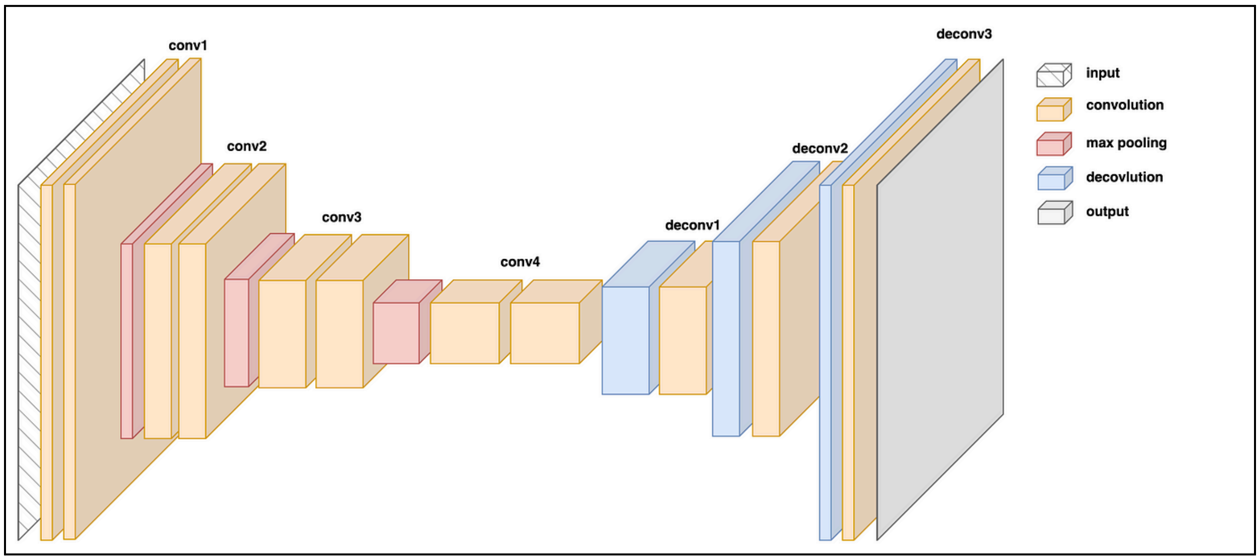
- The next step involves converting the pixel values to float and then dividing each pixel value by 255. This process is called normalization. Normalization is a common preprocessing step that can help speed up training and lead to better performance of neural networks.
- Then we add different types of noise (salt-and-pepper, Poisson, and Gaussian) to the image. This step is essential for training models on noisy data, which can help improve the model's generalization ability.
- Then the permute converts the images to PyTorch tensors and permutes the dimensions from (height, width, channels) to (channels, height, width). PyTorch models typically expect input in the format (channels, height, width).
- After that we return the image and the noisy image.

```
image_directory = '/kaggle/working/test2014'
image_paths = glob.glob(os.path.join(image_directory, '*.jpg'))
dataset = NoiseImageDataset(image_paths)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)
```

- Now the dataset class is initialized with image directory passed as argument.
- Then a custom **dataloader** is created that helps in performing mini batch gradient descent during model training.

MODEL ARCHITECTURE

The model architecture here is a UNet architecture without skip connections. A intuitive diagram of UNet architecture is given below :



*THIS IS JUST FOR REFERENCE THE ACTUAL MODEL ARCHITECTURE IS DIFFERENT.

The model is divided into two parts encoder and decoder :

Encoder : The encoder is responsible for compressing the input data into a more compact representation. It essentially encodes the input into a format that the decoder can later use to reconstruct the original input.

Decoder : The decoder takes the compact representation created by the encoder and attempts to reconstruct the original input. It essentially decodes the compressed data back into a form similar to the original and in this process learns the representation of the image.

The model architecture used is explained in detail below :

ENCODER

Input : The input to the model are images of batch size 128 each of standardized size of 128*128.

Convolution Block : A single of convolution block consists of various sub parts which are stated below :

- A 2D convolution with kernel size 3 and padding 1.
- A 2D batch normalization that normalizes the image batch.
- A Leaky ReLU(max(0.01*x,x)).
- A 2D convolution with kernel size 3 and padding 1.
- A 2D batch normalization that normalizes the image batch.
- A Leaky ReLU(max(0.01*x,x)).
- A 2D max pooling layer.

With each convolution the number of feature maps doubles.

Convolution Block : Same as above.

Convolution Block : Same as above.

DECODER :

Upsampling Block : A single upsampling block consists of various sub parts which are stated below :

- An Upsample block that increases the size by a factor of 2.
- A 2d convolution reduces the feature maps to half.
- A 2d batch normalization.
- A Leaky ReLU(max(0.01*x,x)).
- ¸A 2d convolution that keeps the number of feature maps same.
- A 2d batch normalization.
- A Leaky ReLU(max(0.01*x,x)).

- And a max pool operation.

Upsampling Block : Same as above.

Final Block :
- An upsampling operation that increases the size by 2.
- A 2d convolution that yields a 3 channel tensor.
- A final ReLU layer.

*Haven't pasted the entire model code due to the large length.

PERCEPTUAL LOSS

```
class PerceptualLoss(nn.Module):
  def __init__(self):
    super(PerceptualLoss ,self).__init__()
    vgg19 = models.vgg19(pretrained=True).features[:36].eval()
    for param in vgg19.parameters():
      param.requires_grad = False
    self.vgg19 = vgg19
    self.criterion = nn.MSELoss()

  def forward(self, output, target):
    output_fea = self.vgg19(output)
    target_fea = self.vgg19(target)

    return self.criterion(output_fea , target_fea)
```

Mean Squared Error :
- Mean Squared Error (MSE) is a straightforward way to measure how close our predictions are to the actual values.
- In simpler terms, MSE tells us how much our predictions deviate from the actual values, with larger errors being penalized more heavily due to the squaring step.
- The problem with MSE is that it treats all pixel errors equally. In image processing, this can be a problem because human eyes are more sensitive

to certain types of errors, like those affecting edges and textures, than others.

- Also, since MSE squares the errors, a single large error can disproportionately affect the overall MSE, making it very sensitive to outliers.

Perceptual Loss :

- Perceptual loss measures the difference between images in a way that better matches human perception. Instead of comparing pixels directly, it compares high-level features extracted by a deep neural network (often a pre-trained network like VGG).
- Therefore we measure MSE of these high-level features instead of the pixel values and sum these differences to get the perceptual loss.
- Hence Perceptual Loss provides a measure that aligns better with human perception.

In our model we used a combination of both of them :

Loss = MSE + 0.1 * Perceptual Loss

- We used this equation because the MSE part ensures that image is closer to ground truth on a pixel-by-pixel basis and the Perceptual Loss term ensures that good textures, edges that are more closer to human perception.
- A weight like 0.1 gives a modest influence to the Perceptual Loss, ensuring it improves visual quality without overpowering the MSE component.

Training

- The training is done using Kaggle P100 GPUs.
- At first I started to train my model on 30 epochs but after 5-6 hours of training it used to stop multiple times.
- So I went for the approach of model checkpointing in which I saved the models at intervals of 5 after 10 epochs. This time the same thing happened. The training got stuck at 21 epochs with even epoch 20 model not saving.
- So i went for 15_unet_ploss_vgg19.pth model, downloaded it, renamed it to “15epoch”
- And continued to train it on 10 more epochs to get 20_unet_ploss_vgg19.pth and 25_unet_ploss_vgg19.pth models.

Peak Signal to Noise Ratio (PSNR)

$$\boxed{PSNR = 10\log_{10}\frac{255^2}{MSE}\,\mathrm{dB}}$$

- The Peak Signal-to-Noise Ratio (PSNR) is a metric commonly used to evaluate the quality of reconstructed or compressed images.
- Here 255 is the maximum value of the image signal.
- MSE is the mean squared error between the ground truth image and the predicted image.
- This is measured in dB.

```python
def psnr(gt_image, pred_images, max_val=1.0):
    mse = torch.mean((gt_image - pred_images) ** 2)
    psnr_val = 10 * torch.log10((max_val ** 2) / mse)
    return psnr_val

def calculate_psnr(data_loader, model):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()
    psnr_sum = 0.0
    num_batches = 0

    with torch.no_grad():
        for batch_idx, (noisy, gt_image) in enumerate(data_loader):
            noisy = noisy.to(device)
            pred_images = model(noisy)

            psnr_batch = psnr(gt_image, pred_images)

            psnr_sum += torch.sum(psnr_batch)
            num_batches += 1

    avg_psnr = psnr_sum / num_batches
    return avg_psnr.item()
```

The given two functions are used to calculate the PSNR in one forward pass of the entire data through our trained model.

- The psnr() function first calculates MSE between the ground truth and predicted image and then fits it in the formula to get PSNR.
- This function is called in every batch and then the average PSNR is calculated by calculating the mean over all the batches.

PSNR : 27.962 (test dataset 2k images) ; 14 (training dataset)

Further Improvements :

- Further improvement to this very architecture could be using skip connections by concatenating filters from corresponding encoder part with decoder part.
- Another improvement could be to use GAN architecture presented in the paper by Justin Johnson and Fei Fei : https://arxiv.org/abs/1603.08155v1.