DL Seminar Project 2: Super-resolution with Deep Learning through Perceptual Loss Optimization

Shivam Gard, EID: sg48957 **Sagnik Majumder, EID: sm72878**

1 Data for training and inference

We combined a total number of 37,500 images in our dataset. These comprise of 15,000 images from ImageNet, 15,000 from Places-2, 5000 from the Elvis dataset and 2500 from the Labeled Faces in the Wild (LFW) dataset. We made the images diverse enough because our model was going to be finally tested on a blind dataset. Making our model overfit a very restricted dataset and getting a good performance on that doesn't necessarily guarantee that our model will perform well on the blind dataset. We also had to keep the number of images from each dataset low enough so that we could finish training on the combined data in time using the limited Colab compute that we have. Not having too many images of every dataset but having a lot of diversity made training on this dataset pretty difficult.

2 Training loss functions

We trained all our models by using a sum of 4 losses: *perceptual loss computed from the VGG-16 model, L1 loss, L2 loss and negative of the SSIM score*. We heuristically gave a weight of 1 to all the losses. In addition to the perceptual losses, we used the L1 loss and the negative of the SSIM score to ensure a good score on the L1 and SSIM metrics. However, using L1 loss often introduced artificats/grains in the output images, so we had to balance that with an L2 loss which would help in smoothing out these artifacts. Although these loss functions should be highly correlated, optimizing all of these at the same time is not trivial. While SSIM and perceptual losses are feature based losses, L1 and L2 are pixel-based losses. A network may optimize one or both of these pixel based losses and still give a bad performance with respect to the feature based losses. But we anyway felt it's in our best interests to optimize these four losses together to get a decent score on each of these metrics during grading.

3 Techniques Used

3.1 Residual CNN with Perceptual Loss

The model which we finally settled was a simple model based on principles of residual connections. The upsampling part involved a bilinear interpolation followed by a convolution layer. Our network is 10 layer deep with 3 residual blocks. We started of with a deeper network anticipating the need for larger capacity for super resolution. But the deeper models proved difficult to be optimise even after we added batchnorm layers, swish activations and Leaky ReLU activations. We progressively trimmed down the network to arrive at the final network.

Layer	Output	Details
conv_1	64x64x128	3x3, 128
conv_2	64x64x128	3x3, 128
conv_3	64x64x128	3x3, 128
upsample_0	128x128x128	bilinear
conv_4	128x128x64	3x3, 64
conv_5	128x128x64	3x3, 64
conv_6	128x128x64	3x3, 64
upsample_1	256x256x64	bilinear
conv_7	256x256x64	3x3, 64
conv_8	256x256x64	3x3, 64
conv_9	256x256x64	3x3, 64
conv_10	256x256x3	3x3, 3

Table 1: Encoder Architecture

ReLU activation was used after every convolution layer. BatchNorm was used after conv_2, conv_3, conv_5, conv_6, conv_8 and conv_9. Residual blocks comprised of (conv_2, conv_3), (conv_5, conv_6) and (conv_8, conv_9). Before arriving at this architecture we tried a couple of more things we thought would prove useful to the super resolution task but we failed to achieve good results with them.

3.2 Adaptation of UNet architecture of Pix2Pix and SRCNN architecture

We tried a modified version of the UNet architecture mentioned in Pix2Pix. It has an equal number of layers in the encoder and decoder modules for downsampling an image to a certain feature size and again upsampling it back to an image of the same size. However, we had to use fewer layers in both the encoder and decoder with fewer channels to make it fit on our Colab GPUs with a batch size of 16 to make it not take too long to finish a single run. We also didn't use dropout in the decoder module and replaced all transpose-convolutional layers with upsampling layers. This is because using transpose-convolutional layers often introduce chequered box patters in the images and upsampling layers help in mitigating this effect.
We also trained the SRCNN (Super-resolution CNN) architecture and a very basic 3-layered architecture mentioned in a PyTorch example which uses pixel-reshuffling. Unfortunately this also failed to fit our training data well enough.

3.3 Generative Adversarial Networks

We wrote architecture for generator and discriminator based on similar principles listed above. But GAN network proved really hard to optimise. Most common problem we got was the generator collasping and producing garbage images. We used L1 + L2 + SSIM + GAN loss for generator and standard GAN loss for discriminator. We also tried various varying the ratio of discriminator to generator updates, but most networks either collasped or gave very poor results.The motivation of selecting GAN in the first place was that the criteria of scoring involves diversity which GANs would factor in by design. Plus by adding L1, L2 and SSIM loss, we thought the system would learn to optimise them well alongwith producing diverse set of images. We were able to train one GAN network but the results were not good. A score of 20 was achieved on our validation set. So we decided to shift to perceptual loss implementation.

4 Possible pitfalls

After running multiple experiments and not getting a score of 60 points, we realized a few possible pitfalls with our implementation.

- The dataset we are training on is a very diverse dataset to ensure that it generalizes well to other datasets. The sheer diversity and complexity of the dataset might be preventing the model from fitting the data completely.

- We have been able to train each of our models for only 20 epochs on Colab. It is highly possible that they haven't reached full convergence while training and training them for more epochs could have helped.

- We couldn't achieve very high diversity in the output images. We got a diversity score of around 12 points on our test set of 115 images. This is because we are adding random noise to our input during inference using the seed provided in the grader scripts, to induce diversity in the outputs. However, using too much noise for the sake of a good diversity score heavily affects our loss based scores.

- Moreover, we were trying to achieve 4 losses at the same time some of which might be competing against the others during optimization. A better approach would have been to weight the losses differently and do a rigorous hyperparameter search on the weights based on our validation-cum-test set.

- Finally, we have been using **bicubic** interpolation during training and have been getting zero scores during grading for a very long time. We didn't pay much attention to the interpolation technique that is being used in the grading scripts ('nearest' in *val_grader/tests.py*) because we thought that the interpolation technique wouldn't matter much and our network will be robust enough. But as it turns out, when we change the interpolation in *tests.py* to **bicubic** we get a score of around 42 points on our test set. Thus, we humbly request the instructors to kindly run our grading using **bicubic** interpolation. We have also written a separate email to both the instructors in this regard. Nevertheless, this experience provides the realization that the best way to train the network would have been to use all possible interpolation methods on the training set and augment the data to make it more robust.