Single Image Super Resolution

Christian Mancini
7110459
christian.mancini1@stud.unifi.it

August, 2024

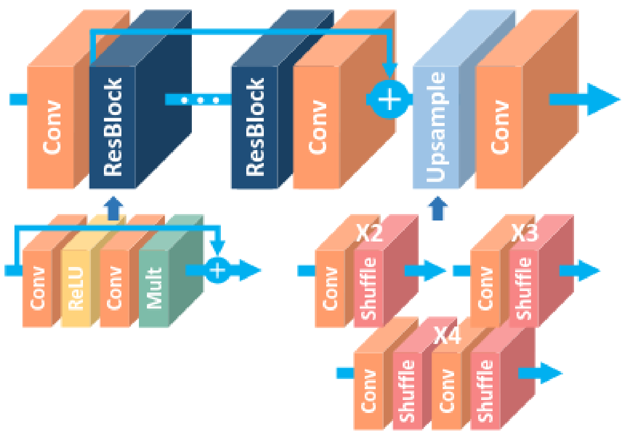
Contents

1 Introduction

The goal of a Single Image Super Resolution Network is to take a low-resolution image and enhance its resolution by a factor of 2 or more. This is a regression task. The network takes an image in tensor form as input and outputs a larger tensor based on the scaling factor. The output tensor can be converted back into an image to visualize the upscaling result.
We can utilize a Convolutional Neural Network that employs several convolutional filters (learned during training) to leverage the regular structure of an image. A proposed architecture for this type of problem is shown in Figure 1. The network contains 3 main layers:

- Convolutional
- Residual Block
- Upsample

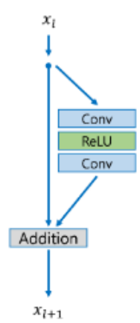
Figure 1: Super Resolution Architecture



1.1 Convolutional And Residual Block Layers

As shown in Figure 1, the first part of the network consists of an alternation between Convolutional and Residual blocks. The architecture of the Residual Blocks is illustrated in Figure 2.

Figure 2: Residual Block Architecture



In the residual block layer, the input is processed through a convolutional filter, followed by a ReLU activation, another convolutional filter, and then summed with the original input for that layer before being passed to the next layer. The number of residual blocks in the network determines the depth of the network and is a parameter to be chosen during model selection. The implementation of the Residual Block is shown in listing 1 .

```
from torch import nn


```

```
class ResidualBlock(nn.Module):

        def __init__(self, num_channels):
                super().__init__()
                self.conv1 = nn.Conv2d(num_channels, num_channels,
                    kernel_size=3, padding=1)
                self.relu = nn.ReLU()
                self.conv2 = nn.Conv2d(num_channels, num_channels,
                    kernel_size=3, padding=1)

        def forward(self, x):
                res = x
                out = self.conv1(x)
                out = self.relu(out)
                out = self.conv2(out)
                out += res
        return out
```

Listing 1: Residual Block implementation

1.2 Upsample and Output Layer

The Upsample layer is the last layer responsible for increasing the dimensions of the tensor before passing it through the output convolutional layer. The implementation of the Upsample layer is shown in Listing 2. The component that actually performs the upscaling is the pixel shuffle operation.

```
from torch import nn

class Upsample(nn.Module):

        def __init__(self, num_channel):
                super().__init__()
                self.conv = nn.Conv2d(num_channel, num_channel * 4,
                     kernel_size=3, padding=1)
                self.shuffle = nn.PixelShuffle(2)

        def forward(self, x):
                out = self.conv(x)
                out = self.shuffle(out)
        return out
```

Listing 2: Upsample implementation

2 Manage Data

In PyTorch, there are conventions to be followed when using datasets. In our case, the dataset consists of images from the **Caltech101 dataset** (found in `torchvision.datasets`), specifically the **airplane dataset**. In the package dataset, there is a module called `data_preparation.py`, which contains a function to download this dataset into the **data** folder of the project.

2.1 Dataset class

As mentioned before, in PyTorch, we should use a convention to easily manage datasets. We need to extend the dataset class. The result is a **SuperResolutionDataset** class found in dataset/super_resolution_dataset.py.

```
class SuperResolutionDataset(Dataset):
        def __init__(self, root_dir, transform=transforms.Compose([transforms.ToTensor()])) -> None:
                self.root_dir = root_dir
                self.transform = transform
                self.file_names = os.listdir(root_dir)

        def __len__(self) -> int:
        return len(self.file_names)

        def __getitem__(self, idx) -> tuple[torch.Tensor, torch.Tensor]:
                img_path = os.path.join(self.root_dir, self.file_names[idx])
                image = Image.open(img_path)
                low_res = image.resize((128, 64))
                high_res = image.resize((256, 128))

                if self.transform:
                low_res = self.transform(low_res)
                high_res = self.transform(high_res)

        return low_res, high_res
```

Listing 3: SuperResolutionDataset

As we can see in listing 3, we have to define a function to get the length of the dataset and a function to specify what happens during access with an index. In our case, we return two tensors containing the low-resolution and high-resolution representations of the images.

2.2 Splitting and loading the dataset

For splitting and loading the dataset we have to use the class that we have implemented.

```
def split_dataset(dataset: Dataset, sizes: dict[str, float]) -> list[Subset]:
        required_keys = {'train', 'validation', 'test'}
        if set(required_keys) != set(sizes.keys()):
        raise ValueError(f"Dictionary of sizes must contain 'train', 'validation', and 'test' keys")
        if not sum(sizes.values()) == 1.0:
        raise ValueError(f"Sizes do not summ up to 1.0, but got {sum(sizes.values()):.2f}")
        train_size = int(sizes["train"] * len(dataset))
        validation_size = int(sizes["validation"] * len(dataset))
        test_size = len(dataset) - train_size - validation_size
```

```
return torch.utils.data.random_split(dataset, [train_size,
    validation_size, test_size])
```

Listing 4: Splitting

The splitting function shown in listing 4 takes a dictionary with percentages as input (which must sum to 1) and returns three disjoint datasets that have been randomly split.
To load and utilize the dataset, we pass it to the `DataLoader` class, which is a standard approach in PyTorch. We can also specify the batch size and whether we want to shuffle the dataset when selecting batches.
An example of how to use it is provided in listing 5.

```
download("./data", "airplanes")
root_dir = 'data/airplanes'
dataset = SuperResolutionDataset(root_dir=root_dir)
dataset_dataloader = DataLoader(dataset, batch_size=16, shuffle=
    True)
```

Listing 5: Example of DataLoader

Iterating through the DataLoader we can get a similar output as shown in Fig. 3.

Figure 3: Example Output of DataLoader

3 Training the best model

For training the best model over all possible model (or a subsample of them) we have to apply the model selection principles.

3.1 Model Selection

Model selection involves training different models using the same training dataset. This could include using the same model architecture with different parameters or employing entirely different architectures. After training, we utilize a validation set that does not contain any samples from the training set to calculate the **loss** and **metrics** for each model.
Once we have selected the **best model** based on these metrics, we continue training only that model using the combined training and validation set, as the validation set is no longer needed. To clarify, the final performance metrics of

the model (e.g., **loss**, **PSNR**) have not yet been evaluated at this stage. The proportion of the splitting is reported in Table 1

Table 1: Data Splitting proportion

Dataset Split	**Value**
Train	0.50
Validation	0.30
Test	0.20

For simplicity (and lack of computational resources), we perform model selection on the two possible parameter of the Network, i.g. the number of channels (feature channels), and the number of residual block (that regulates the depth of the network). The combinations used are in Table 2.

Table 2: Validation Parameters

Number of Channels	**Number of Residual Blocks**
16	4
16	8
16	16
32	4
32	8
32	16
64	4
64	8
64	16

3.2 Training for Validation

The training needs to have a training set, a Loss and an Optimizer. We choose L1 Loss and ADAM optimizer with hyperparameters reported in Table 3. We

Table 3: Adam Hyperparameters

Hyperparameter	**Value**
Learning Rate	1×10^{-4}
Betas	(0.9, 0.999)
Epsilon	1×10^{-8}

trained each model for 50 epochs and choose the best model based on the result found in validation. The state of the model is saved for future use and demonstration. The best model is described in Table 4, and we can see that the model

Table 4: Validation Results

Metric	**Value**
L1 Loss	0.017452
PSNR	30.2676 dB
Number of Channels	64
Number of Residual Blocks	8

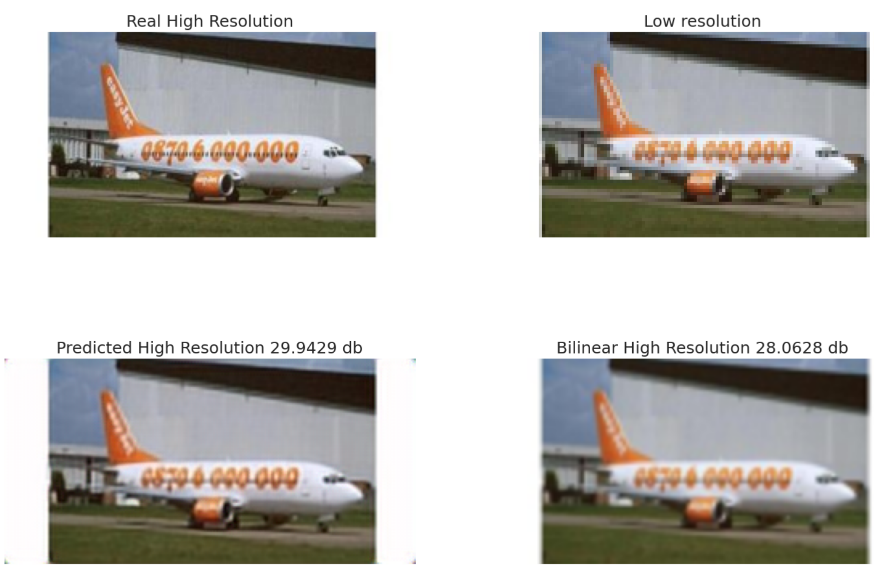
Table 5: Training Recap

Parameter	**Value**
Loss Function	L1
Optimizer	Adam
Epochs	50
DataLoader	training set

is smaller than the one in the paper that used 16 Residual Blocks. Indeed we are working with much smaller images.

To recap, what has been used during all the trainings is reported in Table 5. A visual comparison of the result of the best model in validation can be observed in Figure 4. Even if the difference in db is not marked, this model, compared to algorithmic techniques like Biliner , gives a less blurry image.

Figure 4: Validation output



Continuing to train the best model for an additional 150 epochs using the combined training and validation set leads to the results shown in terms of loss and PSNR, as seen in Figure 5 and Figure 6. The classification of image quality levels as low, medium, and high based on PSNR values is a matter of convention and does not relate to resolution; it only indicates how much the generated image differs from the original.

Figure 5: L1 Loss

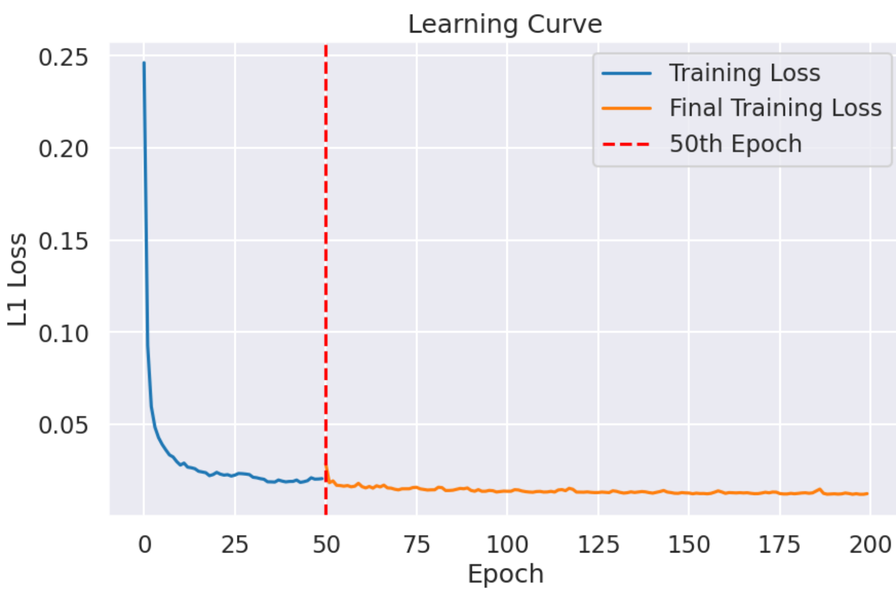
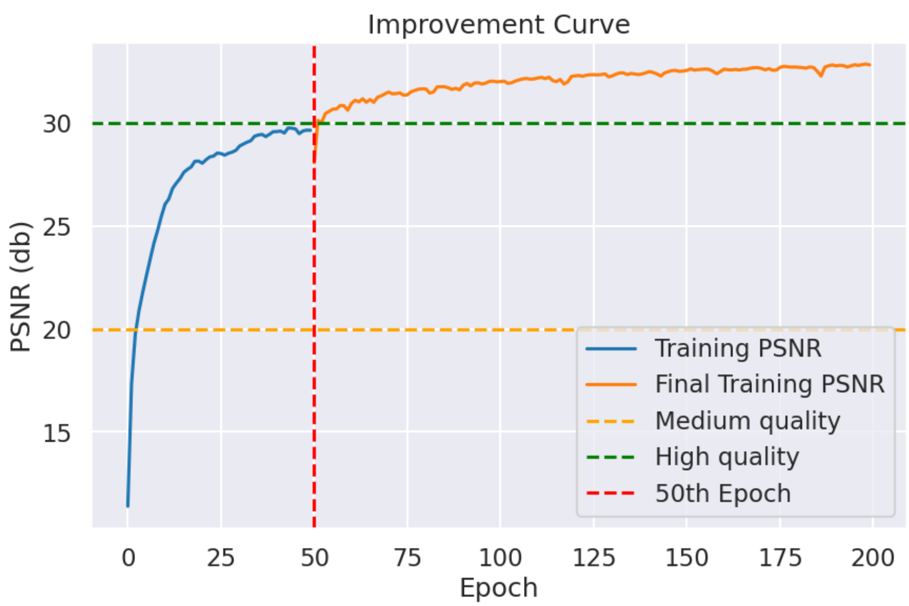


Figure 6: PSNR



4 Test the model

Upon the completion of the validation and final training phases, we proceed to the model assessment stage. In this phase, we conduct a comprehensive evaluation of the model using data samples that the model has not encountered during training. This assessment is crucial for determining the model's generalization capabilities.

4.1 Testing Methodologies

It is considered good practice to utilize the same metrics during testing as those employed during validation. The testing method is a function of the **SuperResolution** class. As illustrated in Listing 6, we set the model to evaluation mode (which prevents weight updates) and compute the L1 loss and Peak Signal-to-Noise Ratio (PSNR) on the test dataset.
Validation and testing essentially execute the same code; however, their purposes are distinct.

```
def test(self, loss_fn, test_dataloader, device='cpu'):
        self.to(device)
        self.eval()
        total_loss = 0.0
        total_psnr = 0.0
        for low_res, high_res in test_dataloader:
        low_res = low_res.to(device)
```

```python
        high_res = high_res.to(device)
        with torch.no_grad():
        predicted_high_res = self(low_res)
        loss = loss_fn(predicted_high_res, high_res)
        total_loss += loss.item()
        total_psnr += peak_signal_noise_ratio(predicted_high_res,
            high_res)

        avg_loss = total_loss / len(test_dataloader)
        avg_psnr = total_psnr / len(test_dataloader)
return avg_loss, avg_psnr
```

Listing 6: Testing Method

A comparison of the results obtained from testing and validation is presented in Table 6, with the validation results included solely for reference. Additionally, Figure 7 provides a visual comparison of the test results.

Table 6: Comparison of Test and Validation Metrics

Metric	**Validation**	**Testing**
L1 Loss	0.017452	0.012140
PSNR	30.2676 dB	32.9143 dB

Figure 7: Test output

We focused on images of airplanes seen from the front. Even though these images had low resolution, we were able to achieve good upscaling results in a relatively short time, about 12 seconds per epoch on a 3050 Ti GPU. This shows that the model works well for improving image quality while being efficient.

List of Figures

List of Tables