Single Image Super Resolution

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

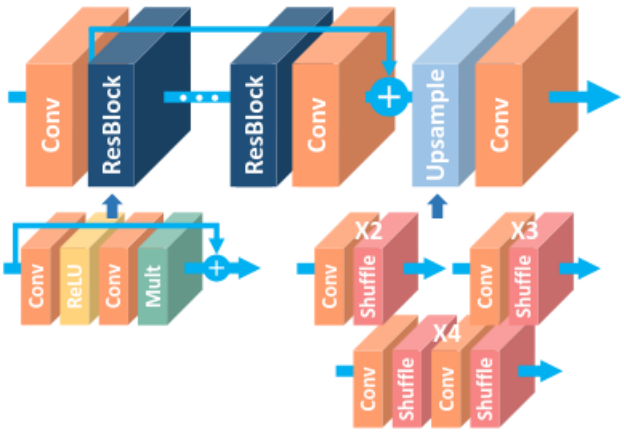
September, 2024

Contents

1 Disclaimer

In this project, we've implemented two models with different hardware setups. The first model trained on a GPU using the standard PyTorch modules. It handles upscaling images from 128×64 to 256×128. The second model is trained on a CPU, where we wrote our own modules. Even though the implementations differ, both models keep the same method signatures. This means you can easily switch between them , allowing you to use checkpoints from either model interchangeably. For the smaller CPU model, we focused on upscaling images from 32×16 to 64×32. In this report we will present the results of both models.

Figure 1: Super Resolution Architecture



2 Introduction

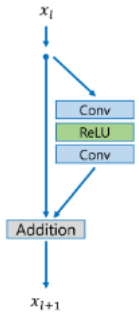
The goal of a Single Image Super Resolution Network is to take a low-resolution image and enhance its resolution by a factor of 2 or more. This is a regression task. The network takes an image in tensor form as input and outputs a larger tensor based on the scaling factor. The output tensor can be converted 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

2.1 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

2.2 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 covolutional layer that "generates" the new pixels and the PixelShuffle that rearange it with the proper dimentions.

```python
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

3 Building Blocks

We developed the modules to ensure compatibility with the PyTorch ones, as they share the same structure. We trained a model on a GPU using PyTorch and a smaller model on a CPU using our modules, employing various techniques to speed up computation. In the end, both checkpoints can be used interchangeably since the state dictionaries have the same entries and architecture.

3.1 Convolution

Convolution is the fundamental operation in a Convolutional Neural Network (CNN). It involves sliding a kernel (or filter) over the input image to produce a feature map, which highlights important patterns and features. The convolution operation can be formally defined as follows:

$$\text{Output}(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \text{Input}(i+m, j+n) \cdot \text{Kernel}(m,n) \tag{1}$$

Implementing convolution in a naive approach, as shown in Listing 3, is formally correct and can be used with any image and kernel, even considering batch size. However, the four nested loops lead to significant inefficiencies. The complexity of implementation grows rapidly with the increase in image size and the number of filters. Additionally, the nested loops result in inefficient memory access patterns, which can cause cache misses and slow down execution. This implementation does not take full advantage of the parallelization capabilities of modern GPUs and even CPUs, as optimized libraries like cuDNN utilize specialized algorithms to perform convolutions much more efficiently. Furthermore, the overhead associated with Python's interpretation and variable management can contribute to the overall slowness, especially in a context with intensive looping. Fortunately, there are several techniques that can be employed to speed up computation, although this often comes at the cost of increased memory

usage. The easiest way to speed up computation is by compiling the code. One strategy that we did not adopt is to utilize Numba, which can compile Python functions to machine code and accelerate operations that use NumPy. However, Numba has limitations; it is primarily effective for NumPy arrays and may not support more complex Python objects or libraries. It can help bypass some limitations of the Global Interpreter Lock (GIL) for the functions it compiles. Additionally, since we are using tensors with frameworks like PyTorch, which already optimize and parallelize tensor operations, the benefits of using Numba in our case may be limited.

```
def slow_forward(self, image):
        image = nn.functional.pad(image, (self.padding,) * 4, "
            constant", 0)
        batch_size, in_channels, height, width = image.shape
        out_channels, in_channels_kernel, m, n = self.weight.shape
        if self.in_channels != in_channels:
        raise ValueError(
        f"Input channels are different: Declared {self.in_channels
            }, but got Image with {in_channels}")
        output_height = height - m + 1
        output_width = width - n + 1
        new_image = torch.zeros((batch_size, out_channels,
            output_height, output_width))

        for b in range(batch_size):
        for c in range(out_channels):
        for i in range(output_height):
        for j in range(output_width):
        new_image[b, c, i, j] = torch.sum(image[b, :, i:i + m, j:j
            + n] * self.weight[c]) + self.bias[c]
        return new_image
```

Listing 3: Slow convolution implementation

To further enhance the efficiency of the convolution operation, we can utilize the `im2col` transformation. This technique reshapes the input tensor into a 2D matrix where each column represents a flattened receptive field of the input corresponding to a specific position of the kernel. This transformation allows us to perform matrix multiplication between the reshaped input and the weights of the convolutional layer, which can be highly optimized and parallelized using libraries like cuBLAS on GPUs, but even CPUs will benefit of this access pattern. The implementation of the `im2col` function is shown in Listing 4:

```
        def _im2col(input, kernel_size, stride=1, padding=0):
        input_padded = torch.nn.functional.pad(input, (padding,
            padding, padding, padding))
        batch_size, in_channels, height, width = input_padded.size
            ()
        kernel_height, kernel_width = kernel_size
        out_height = (height - kernel_height) // stride + 1
        out_width = (width - kernel_width) // stride + 1
        col = torch.empty(batch_size, in_channels, kernel_height,
            kernel_width, out_height, out_width)

        for y in range(kernel_height):
```

```
        for x in range(kernel_width):
        col[:, :, y, x, :, :] = input_padded[:, :, y: y +
            out_height * stride: stride,
        x: x + out_width * stride: stride]

        return col.view(batch_size, in_channels * kernel_height *
            kernel_width, -1)
```

Listing 4: im2col tranformation algorithm

Once the input has been transformed using `im2col`, the convolution operation can be performed efficiently in the conv_forward function, as shown in Listing 5:

```
        def _conv_forward(self, input, weight, bias=None, stride=1,
             padding=0):
        col = _im2col(input, weight.size()[2:], stride, padding)
        weight_col = weight.view(weight.size(0), -1)
        out = torch.matmul(weight_col, col)

        if bias is not None:
        out += bias.view(1, -1, 1)

        batch_size, out_channels = out.size(0), weight.size(0)
        out_height = (input.size(2) + 2 * padding - weight.size(2))
             // stride + 1
        out_width = (input.size(3) + 2 * padding - weight.size(3))
            // stride + 1
        return out.view(batch_size, out_channels, out_height,
            out_width)
```

Listing 5: Implementation of the forward function

By leveraging the `im2col` transformation, we can efficiently compute the convolution operation in a way that is parallel and faster.

3.2 Pixel Shuffle

The primary function of the `PixelShuffle` layer is to rearrange the output tensor from the previous convolutional layer. This output tensor typically contains additional dimensions that facilitate the upscaling process. The `PixelShuffle` layer effectively reorganizes these dimensions and prepares the data for the final convolutional output layer, ensuring that the spatial resolution of the image is increased while maintaining the integrity of the feature maps.

```

class PixelShuffle(nn.Module):
        def __init__(self, upscale_factor: int):
                super().__init__()
                self.upscale_factor = upscale_factor

        def forward(self, x):
                batch_size, channels, height, width = x.shape
                channels //= (self.upscale_factor ** 2)
                x = x.view(batch_size, channels, self.
                    upscale_factor, self.upscale_factor, height,
                    width)
```

```
                x = x.permute(0, 1, 4, 2, 5, 3)
                return x.contiguous().view(batch_size, channels,
                    height * self.upscale_factor, width * self.
                    upscale_factor)
```

Listing 6: Implementation of the PixelShuffle class

4 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.

4.1 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 7: SuperResolutionDataset

As we can see in listing 7, 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.

4.2 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 8: Splitting

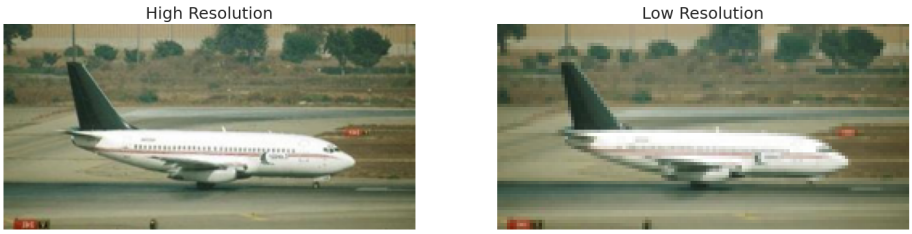
The splitting function shown in listing 8 takes a dictionary with percentages as input (which must sum to 1) and returns three disjoint datasets that have been randomly split.
To load and utilize the dataset, we pass it to the `DataLoader` class, which is a standard approach in PyTorch. We can also specify 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 9.

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

Listing 9: Example of DataLoader

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

Figure 3: Example Output of DataLoader



5 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.

5.1 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

5.2 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 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 large model in validation can be observed in Figure 4 and in Figure 5 for the smaller model. Even if the difference in db is not marked, this model, compared to algorithmic techniques like Biliner , gives a less blurry image.

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

Figure 4: Validation output large model

Figure 5: Validation output small model

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 6 and Figure 7. 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 6: L1 Loss

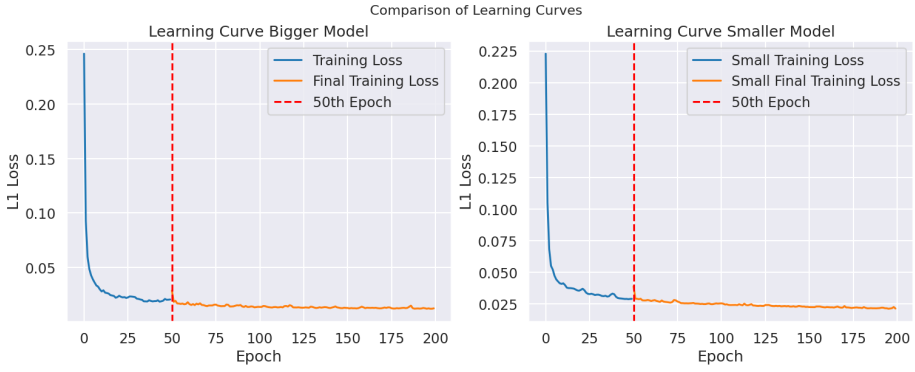
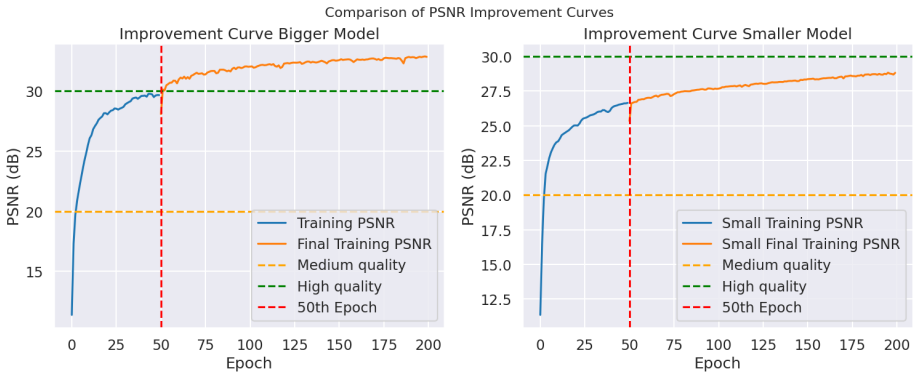


Figure 7: PSNR



6 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.

6.1 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 10, 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)
        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 10: Testing Method

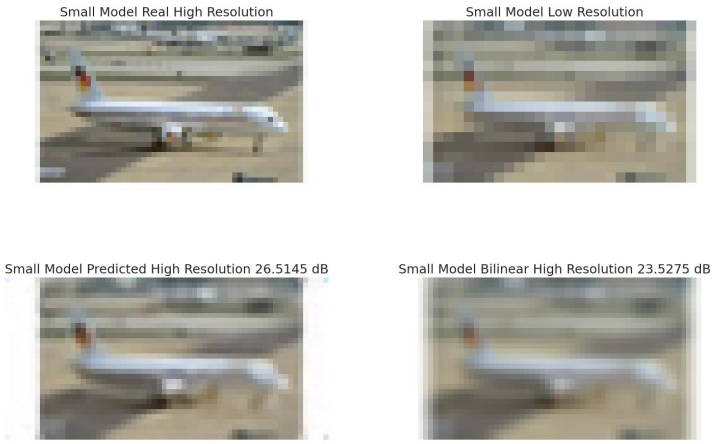
A comparison of the results obtained from testing and validation is presented in Table 6, with the validation results included solely for reference. Additionally, Figure 8 provides a visual comparison of the test results.

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 8: Test output for large model

Figure 9: Test output for small model



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. For the small model, we can also achieve inference in about 20 seconds on CPU, and the results are very satisfying, as the upscaling is more noticeable due to the very low resolution of the images.

List of Figures

List of Tables