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

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 1: Super Resolution Architecture

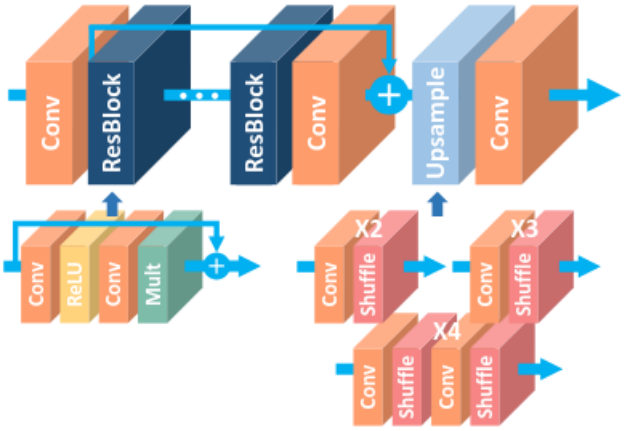
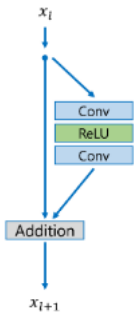


Figure 2: Residual Block Architecture



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.

```python
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.

```python
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.

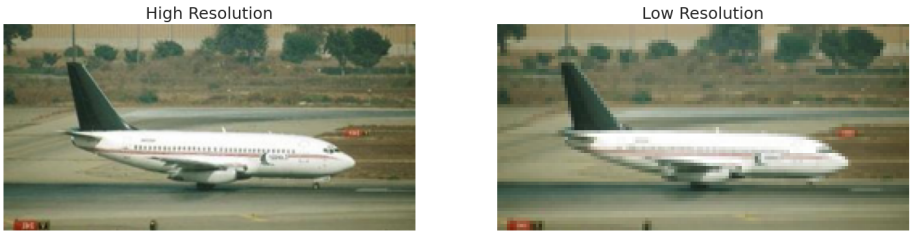
To load and utilize the dataset, we pass it to the `DataLoader` class, which is a standard approach in PyTorch. We can also specify the batch size and whether we want to shuffle the dataset when selecting batches.
An example of how to use it is provided in listing 5.

```
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



List of Figures

List of Tables