1 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.

1.1 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 1, 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.

```
1 def slow_forward(self, image):
2         image = nn.functional.pad(image, (self.padding,) * 4, "
              constant", 0)
3         batch_size, in_channels, height, width = image.shape
4         out_channels, in_channels_kernel, m, n = self.weight.shape
5         if self.in_channels != in_channels:
6         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 1: 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 2:

```
        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 2: 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 3:

```
        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)
```

```
5
6           if bias is not None:
7               out += bias.view(1, -1, 1)
8
9           batch_size, out_channels = out.size(0), weight.size(0)
10          out_height = (input.size(2) + 2 * padding - weight.size(2))
                // stride + 1
11          out_width = (input.size(3) + 2 * padding - weight.size(3))
                // stride + 1
12          return out.view(batch_size, out_channels, out_height,
                out_width)
```

Listing 3: 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.

1.2 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.

```
1
2   class PixelShuffle(nn.Module):
3           def __init__(self, upscale_factor: int):
4                   super().__init__()
5                   self.upscale_factor = upscale_factor
6
7           def forward(self, x):
8                   batch_size, channels, height, width = x.shape
9                   channels //= (self.upscale_factor ** 2)
10                  x = x.view(batch_size, channels, self.
                        upscale_factor, self.upscale_factor, height,
                        width)
11                  x = x.permute(0, 1, 4, 2, 5, 3)
12                  return x.contiguous().view(batch_size, channels,
                        height * self.upscale_factor, width * self.
                        upscale_factor)
```

Listing 4: Implementation of the PixelShuffle class