Assignment 3

Arturo Abril Martinez (2813498)

December 2024

1 Answers

question 1 The learnable weights of the conv layer are represented as a tensor of shape:

`(output_channels, input_channels, kernel_height, kernel_width)`

This is the set of *filters* of the convolutional layer. More explicitly (and excluding the bias terms) we will have `W0, W1, W2, ...` and so on up until the number given by `output_channels`. The total number of parameters will then be:

`kernel_width * kernel_height * input_channels * output_channels.`

The pseudocode to implement the convolution is:

```
Y = \
torch.zeros((batch_size, output_channels, output_height, \
    output_width))

for b in range(batch_size):
    for ch in range(output_channels)
        for row in range(output_height):
            for col in range(output_width):
                Y[b,ch,row,col] = \
                ( \
                    X[b,:,row*S:hk+row*S,col*S:wk+col*S] \
                    * W[ch,:,:,:]
                ).sum()
return Y
```

Where `S` represents the stride and `hk` and `wk` the kernel (or filter) height and width respectively. Note that the multiplication inside the `sum()` is element-wise, and in the pseudocode `W` is the set of all weight tensors, so that `W[0,:,:,:] = W0`. *Note: I assume the input images are already padded, if they need to be.*

question 2 Given that the size of the input is (C_{in}, H_{in}, W_{in}), we can calculate the size of the output as:

$$W_{out} = \frac{W_{in} - K + 2P}{S} + 1$$

H_{out} can be calculated using the same formula by symmetry, changing $W_{in} \to H_{in}$. The size of the output is then $(C_{out}, H_{out}, W_{out})$ where C_{out} can be chosen to be any number. K, S and P are the kernel size, stride and padding respectively[1]. *If the division in the formula above results in a non-integer number, one can take the integer part of the division.*

question 3 Below is the pseudocode for the unfold operation. `p`, `k` and `S` represent the total number of patches, number of values per patch and stride respectively. `w` (`wk`) and `h` (`hk`) are the input (kernel) width and height respectively. *Note: I assume the input images are already padded, if needed to be.*

```
Y = torch.zeros((batch_size, p, k))
for b in range(batch_size):
    # iterate over rows in the output
    # each row will be a flattened image patch
    for row in range(p):
        # select patch
        i = row//w_out  # transition row every `w_out` patches
        j = row%(w-wk+1) # when `row` exceeds maximum, start over
        start_row, start_col = i*S, j*S
        patch = X[
            b, :, start_row : start_row+hk, \
            start_col : start_col+wk
        ]

        # flatten patch
        counter = 0
        for ch_patch in range(input_channels):
            for row_patch in range(hk):
                for col_patch in range(wk):
                    Y[b, row, counter] = \
                    patch[ch_patch, row_patch, col_patch]
                    counter += 1
# This way the output will have size (batch_size, p, k)
return Y
```

question 4 To calculate the backward of the con2d operation, we first present some notation. We first present all the tensors involved and what each of their

[1] I have ommited the batch size in the calculation. Of course, for every tensor in the input batch we would have an output tensor, which dimensions can be calculated with the formula provided.

indices represent. Later, we will use dummy indices for the caclculations, always keeping in mind what each of them represent, which will be determined by their postion. First, the constants:

- b, c, h, w: Batch size, input channels, input width and input height.
- h_k, w_k: Kernel height and kernel width.
- $c_{out}, h_{out}, w_{out}$: Output channels, output height and output width.
- p: Total number of patches. Resulting from doing $h_{out} * w_{out}$.
- k: Total number of values per patch. Resulting from doing $c * h_k * w_k$

Now we define the tensors involved. The indices are now named with what they represent. We will later have to remember this:

- Input batch: $\mathbf{X} = X_{bchw}$
- Unfolded input: $\mathbf{U} = U_{bkp}$
- Unfolded input transposed: $\tilde{\mathbf{U}} = \tilde{U}_{bpk} = \mathbf{U}\text{.transpose}(1, 2)$
- Kernel matrix: $\mathbf{W} = W_{c_{out} c h_k w_k}$
- Kernel matrix reshaped and transposed: $\tilde{\mathbf{W}} = \tilde{W}_{k c_{out}} =$
  $= \mathbf{W}\text{.reshape}(c_{out}, c * h_k * w_k)\text{.transpose}()$
- Output without reshaping: $\mathbf{Y}' = Y'_{bpc_{out}} = \sum_k \tilde{U}_{bpk} \tilde{W}_{kc_{out}}$
- Output: $\mathbf{Y} = Y_{bc_{out}h_{out}w_{out}} =$
  $= \mathbf{Y}'\text{.transpose}(1, 2)\text{.reshape}(b, c_{out}, h_{out}, w_{out})$

Before calculating $\mathbf{W}^\nabla$ and $\mathbf{X}^\nabla$ we present some useful derivatives. Note that there will be matrices which elements have a one to one correspondence, but their shapes are different. This is the case for example of $\mathbf{Y}$ and $\mathbf{Y}'$. When doing derivatives of one w.r.t. the other, we will have simple deltas for the dimensions that match, and "special" deltas for those dimensions that not match, but we still have a one to one correspondence of elements. This will become clear in the following derivatives.

$$\frac{\partial Y_{abcd}}{\partial Y'_{efg}} = \delta_{ae}\delta_{bg}\delta_{cd,f} \;\; ; \;\; \frac{\partial Y'_{efg}}{\partial \tilde{W}_{hi}} = \tilde{U}_{efh}\delta_{gi} \;\; ; \;\; \frac{\partial \tilde{W}_{hi}}{\partial W_{abcd}} = \delta_{ia}\delta_{h,bcd}$$

The deltas of the type $\delta_{ab,c}$ appear when we are dealing with different shapes but there is still a one to one correspondence of elements. In this case we want to pick the element ab that corresponds to c[2].

[2]If this is not still clear, imagine that we have a tensor $\mathbf{X}$ with elements X_{11}, X_{12}, X_{21} and X_{22}. We can "flatten" this into a vector of four values $\mathbf{x}$. X_{11} corresponds to x_1, X_{12} to x_2 and so on. The derivative $\partial X_{ab}/\partial x_c = \delta_{ab,c}$.

First, we do $\mathbf{W}^\nabla$:

$$
\begin{aligned}
[\mathbf{W}^\nabla]_{\alpha\beta\gamma\eta} \equiv \frac{\partial \mathcal{L}}{\partial W_{\alpha\beta\gamma\eta}} &= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \sum_{efg} \frac{\partial Y_{abcd}}{\partial Y'_{efg}} \sum_{hi} \frac{\partial Y'_{efg}}{\partial \tilde{W}_{hi}} \frac{\partial \tilde{W}_{hi}}{\partial W_{\alpha\beta\gamma\eta}} \\
&= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \sum_{efg} \delta_{ae}\delta_{bg}\delta_{cd,f} \sum_{hi} \tilde{U}_{efh}\delta_{gi}\delta_{i\alpha}\delta_{h,\beta\gamma\eta} = \\
&= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \tilde{U}_{a,cd,\beta\gamma\eta}\delta_{b\alpha} = \sum_{acd} \frac{\partial \mathcal{L}}{\partial Y_{a\alpha cd}} \tilde{U}_{a,cd,\beta\gamma\eta}
\end{aligned}
$$

In case indices become unclear, one just have to look up what each tensor represents, check the number of indices that tensor must have and what each of them represent. For example, in the result above, take $\tilde{U}_{a,cd,\beta\gamma\eta}$. We know that $\tilde{\mathbf{U}}$ is the unfolded input and then transposed. We know that its first index represents the batch dimension, the second the total number of patches p and the third the total number of values per patch k. That's all we need. We can easily vectorize the result above to get $\mathbf{W}^\nabla$. We take the matrices $\mathbf{Y}^\nabla$ and $\tilde{\mathbf{U}}$ and sum over the right dimensions. This can be done with some reshaping and transposing with the following pseudo code:

```
# Y_nabla will come with shape: (b, c_out, h_out, w_out)
# U_tilde, as indicated above, (b, p, k)
W_nabla = Y_nabla
    .view(b, c_out, p)
    .transpose(0, 1).reshape(c_out, b*p)
    .matmul(U_tilde.reshape(b*p, k))
    .reshape(c_out, c, hk, wk)
```

We basically group the dimensions over which we have to do the sumation and then reshape back so that $\mathbf{W}^\nabla$ matches the dimensions of $\mathbf{W}$.

question 5 To work out the gradient w.r.t. the input, a lot of the notation and derivatives presented in question 4 are helpful. The approach we take is the following: We note that the elements of the unfolded input $\mathbf{U}$ and the input $\mathbf{X}$ have a one to one correspondence. This is, for every U_{bkp}, *and assuming a certain kernel size*, there is exactly one X_{bchw} and vice versa. The derivative $\partial\mathbf{U}/\partial\mathbf{X}$ will be a delta indicating the right dimensions to be matched[3], namely: $\partial U_{abc}/\partial X_{\alpha\beta\gamma\eta} = \delta_{a\alpha}\delta_{b\beta}\delta_{c,\gamma\eta}$[4]. This relationship assumes a certain fixed kernel size, that $\mathbf{X}$ is already padded, if needed, and a suitable choice of padding

[3]One can think about it a as a generalization of a transposition. Let $\mathbf{A}$ and $\mathbf{B} = \mathbf{A}^T$ square matrices. Then, $\partial A_{ij}/\partial B_{kl} = \delta_{il}\delta_{jk}$. If $\mathbf{B}$ was $\mathbf{A}$ unfolded, the derivative would result also in a delta, with a more complicated relationship between indices (since shapes don't match).

[4]Please take this result with a pinch of salt. Except the first dimension of the tensors, which is indeed the same for both (the batch size), the rest of dimensions are not *strictly* a delta, in the sense that they have to be the same number. They rather mean that whatever the correspondence between both tensors is w.r.t. the indicated dimension, we have to choose the element in the first matrix (and for that dimension) that matches its corresponding element in the second matrix (considering the right dimension).

and stride so that patches can fit nicely given the image size. Once we have realised this, we can work out $\mathbf{U}^{\nabla}$ and *rehsape it back* to match the original shape of the input. If done correctly, the resulting matrix will be exactly $\mathbf{X}^{\nabla}$.
Before we present the result, one more useful derivative:

$$\frac{\partial \tilde{U}_{efm}}{\partial U_{\alpha\beta\gamma}} = \delta_{e\alpha}\delta_{f\gamma}\delta_{m\beta}$$

(Remember that $\tilde{\mathbf{U}}$ is just the transpose of $\mathbf{U}$ w.r.t. the last two indices).
Now we're ready to do $\mathbf{U}^{\nabla}$:

$$\begin{aligned}
[\mathbf{U}^{\nabla}]_{\alpha\beta\gamma} \equiv \frac{\partial \mathcal{L}}{\partial U_{\alpha\beta\gamma}} &= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \sum_{efg} \frac{\partial Y_{abcd}}{\partial Y'_{efg}} \sum_{m} \frac{\partial \tilde{U}_{efm}}{\partial U_{\alpha\beta\gamma}} \tilde{W}_{mg} = \\
&= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \sum_{efg} \delta_{ae}\delta_{bg}\delta_{cd,f}\delta_{e\alpha}\delta_{f\gamma}\tilde{W}_{\beta g} = \\
&= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \delta_{a\alpha}\delta_{cd,\gamma}\tilde{W}_{\beta b} = \\
&= \sum_{b} \frac{\partial \mathcal{L}}{\partial Y_{\alpha b,\gamma}} \tilde{W}_{\beta b}
\end{aligned}$$

Again, this is easy to vectorize. We take $\mathbf{Y}^{\nabla}$ and $\tilde{\mathbf{W}}$ and we sum along the right dimensions (second dimension for both matrices). In pseudo code:

```
U_nabla = W
    .reshape(c_out, c*hk*wk)
    .transpose()
    .matmul(
    Y_nabla
    .reshape(b, c, p)
    .transpose(0, 1)
    .reshape(c_out, b*p)
    )
    .reshape(k, b, p)
    .transpose(0, 1)
# U_nabla is shape (b, k, p)
```

All the reshape / transpose trickery is done so that we can perform a simple matrix multiplication usnig `matmul` taking into acount the dimensions we want to sum. As justified before, we just need to "undo" the unfold operation to get the matrix $\mathbf{U}^{\nabla}$ to the shape expected by $\mathbf{X}^{\nabla}$, this is, the shape of $\mathbf{X}$. For this, we need the constants used for the unfolding: kernel height and width (h_k, w_k) stride and padding. In pseudo code:

```
X_nabla = fold(
    U_nabla,
    output_size=(h, w),
    kernel_size=(hk, wk),
    padding=padding,
    stride=stride
)
```

question 6 Below is the implementation of the 2D convolution forward and backward pass as a `torch.autograd.Function`.

```
class Conv2DFunc(torch.autograd.Function):
"""
We can implement our own custom autograd Functions by subclassing
torch.autograd.Function and implementing the forward and backward
passes which operate on Tensors.
"""

@staticmethod
def forward(
    ctx,
    input_batch: torch.Tensor,
    kernel: torch.Tensor,
    padding: int = 1,
    stride: int = 1,
) -> torch.Tensor:
    """
    In the forward pass we receive a Tensor containing the input
    and return a Tensor containing the output. ctx is a context
    object that can be used to stash information for backward
    computation. You can cache arbitrary objects for use in the
    backward pass using the ctx.save_for_backward method.
    """

    # store objects for the backward
    ctx.save_for_backward(input_batch, kernel)
    ctx.padding = padding
    ctx.stride = stride

    params = Conv2DParams(
        kernel_size=kernel.size(),
        input_size=input_batch.size(),
        padding=padding,
        stride=stride,
    )

    unfolded = F.unfold(  # U
        input_batch,
        (params.hk, params.wk),
        padding=params.padding,
        stride=params.stride,
    ).transpose(
        1, 2
    )  # \tilde{U}

    assert unfolded.size() == (
        params.b,
        params.p,
        params.k,
    ), f"Expected: {(params.b, params.p, params.k)}, but got: {unfolded.size()}"

    output_batch = unfolded.matmul(
        kernel.reshape(params.c_out, -1).t()  # \tilde{U} @ \tilde{W} = Y'
    ).transpose(
        1, 2
    )  # This op together with the next reshape form Y

    assert output_batch.size() == (params.b, params.c_out, params.p)

    return output_batch.view(
        params.b, params.c_out, params.h_out, params.w_out
    )  # Y

@staticmethod
def backward(ctx, grad_output) -> Tuple[torch.Tensor, torch.Tensor, None, None]:
    """
    In the backward pass we receive a Tensor containing the
    gradient of the loss with respect to the output, and we need
    to compute the gradient of the loss with respect to the
    input
    """
    # retrieve stored objects
    (input_batch, kernel) = ctx.saved_tensors
    padding = ctx.padding
    stride = ctx.stride
    # your code here

    params = Conv2DParams(
        kernel_size=kernel.size(),
        input_size=input_batch.size(),
        padding=padding,
        stride=stride,
    )
```

```
grad_output = grad_output.view(
    params.b, params.c_out, -1
)  # shape: (b, c_out, p)

assert grad_output.size() == (
    params.b,
    params.c_out,
    params.p,
), f"Expected: {(params.b, params.c_out, params.p)}, got: {grad_output.size()}"

# Gradient w.r.t. kernel
unfolded = F.unfold(
    input_batch,
    kernel_size=(params.hk, params.wk),
    padding=params.padding,
    stride=params.stride,
).transpose(1, 2)

assert unfolded.size() == (
    params.b,
    params.p,
    params.k,
), f"Expected: {(params.b, params.p, params.k)}, got: {unfolded.size()}"

grad_kernel = (
    grad_output.transpose(0, 1)
    .reshape(params.c_out, -1)
    .matmul(unfolded.reshape(params.b * params.p, params.k))
)

assert grad_kernel.size() == (
    params.c_out,
    params.k,
), f"Expected: {(params.c_out, params.k)}, got: {grad_kernel.size()}"

grad_kernel = grad_kernel.view(params.c_out, params.c, params.hk, params.wk)

# Gradient w.r.t. input
assert grad_output.size() == (
    params.b,
    params.c_out,
    params.p,
), f"Expected: {(params.b, params.c_out, params.p)}, got: {grad_output.size()}"

grad_input_unfolded = (
    kernel.view(params.c_out, -1)
```

```
        .t()  # this is \tilde{W}, shape: (k, c_out)
        .matmul(grad_output.transpose(0, 1).reshape(params.c_out, -1))
    ).reshape(params.k, params.b, params.p)

    assert grad_input_unfolded.size() == (
        params.k,
        params.b,
        params.p,
    ), f"Expected: {(params.k, params.b, params.p)}, got: {grad_input_unfolded.size()}"

    grad_input = F.fold(
        grad_input_unfolded.transpose(0, 1),
        output_size=(params.h, params.w),
        kernel_size=(params.hk, params.wk),
        padding=params.padding,
        stride=params.stride,
    )

    assert grad_input.size() == (
        params.b,
        params.c,
        params.h,
        params.w,
    ), f"Expected: {(params.b, params.c, params.h, params.w)}, got: {grad_input.size()}"

    return (
        grad_input,
        grad_kernel,
        None,
        None,
    )
```

For the forward implementation as a `torch.nn.Module`, see the Appendix (this is really similar of course to the static `forward`) method of the `Conv2DFunc` above. The kernel is directly what we called $\tilde{\mathbf{W}}$, instead of $\mathbf{W}$.

A Appendix

implementation of 2D convolution as a `torch.nn.Module`

```python
class Conv2DModule(nn.Module):

    def __init__(
        self, kernel_size: SizeTuple, stride: int = 1, padding: int = 1
    ) -> None:
        super().__init__()
        # Pytorch's kernel would have shape (out_channles, in_channels, *kernel_size)
        # Ours is built to be a simple 2d matrix, but is equivalent if reshaped
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        self.kernel = torch.randn(
            math.prod(kernel_size[1:]), kernel_size[0]
        )  # shape: (k, c_out)

    def forward(self, input_batch: torch.Tensor) -> torch.Tensor:
        # parameters
        params = Conv2DParams(
            kernel_size=self.kernel_size,
            input_size=input_batch.size(),
            padding=self.padding,
            stride=self.stride,
        )

        unfolded = F.unfold(
            input_batch,
            kernel_size=(params.hk, params.wk),
            padding=params.padding,
            stride=params.stride,
            # We transpose, because Pytorch returns flattened patches as columns
            # I want them as rows
        ).transpose(1, 2)

        assert unfolded.size() == (
            params.b,
            params.p,
            params.k,
        ), f"Expected: {(params.b, params.p, params.k)} but got: {unfolded.size()}"

        # First reshape, then Y = XW, then reshape back
        output = (
            unfolded.reshape(params.b * params.p, params.k)
            .matmul(self.kernel)
            .reshape(params.b, params.p, params.c_out)  # This is actually Y
            .transpose(1, 2)  # This is needed for later
        )

        assert output.size() == (
            params.b,
            params.c_out,
            params.p,
        ), f"Expected: {(params.b, params.c_out, params.p)} but got: {output.size()}"

        return output.reshape(params.b, params.c_out, params.h_out, params.w_out)
```