Assignment 3

Arturo Martinez (2813498)
Paul Kroeger (2829079)
Michele Vannucci (2819493)

Dicember 2024

The code used to obtain the results of this report can be found here[1].

Answers

Part 1

question 1 Having defined the loss as l, and $f(\mathbf{X},\mathbf{Y}) = \mathbf{X}/\mathbf{Y} = \mathbf{S}$, the derivation for $X^{\nabla}_{i,j}$is the following:

$$\begin{aligned}
X^{\nabla}_{i,j} &= -\frac{\partial l}{\partial X^{\nabla}_{i,j}} = \sum_{k,l} \frac{\partial l}{\partial S_{k,l}} \frac{\partial S_{k,l}}{\partial X_{i,j}} \\
&= \sum_{k,l} S^{\nabla}_{k,l} \frac{\partial\left[X_{k,l}/Y_{k,l}\right]}{\partial X_{i,j}} \\
&= S^{\nabla}_{i,j} \frac{\partial\left[X_{i,j}/Y_{i,j}\right]}{\partial X_{i,j}} \\
&= S^{\nabla}_{i,j} \frac{1}{Y_{i,j}} \quad \rightarrow \quad X^{\nabla}_{i,j} = \frac{S^{\nabla}_{i,j}}{Y_{i,j}} \quad \rightarrow \mathbf{X}^{\nabla} = S^{\nabla}/\mathbf{Y}
\end{aligned}$$

Where the division between $\mathbf{S}^{\nabla}$ and $\mathbf{Y}$ is element-wise. Similarly for $Y^{\nabla}_{i,j}$ we get

$$\begin{aligned}
Y^{\nabla}_{i,j} &= S^{\nabla}_{i,j} \frac{\partial\left[X_{k,l}/Y_{k,l}\right]}{\partial Y_{i,j}} \\
&= S^{\nabla}_{i,j} \cdot X_{i,j} \cdot -\frac{1}{Y^{2}_{i,j}} \quad \rightarrow \quad \mathbf{Y}^{\nabla} = -\left[\frac{\mathbf{S}^{\nabla} \odot \mathbf{X}}{\mathbf{Y}^{2}}\right]
\end{aligned}$$

Where all the operations are performed element-wise.

question 2 Given that $F(X) = \mathbf{Y}$, the generic gradient $\mathbf{X}^{\nabla}$ is given by the following derivation:

$$\begin{aligned}
X^{\nabla}_{ij} &= \sum_{kl} \frac{\partial l}{\partial Y_{kl}} \frac{\partial Y_{kl}}{\partial X_{ij}} \\
&= \sum_{kl} Y^{\nabla}_{kl} \frac{\partial\left[f(X_{kl})\right]}{\partial X_{ij}} \\
&= Y^{\nabla}_{ij} \cdot f'(X_{ij}) \quad \rightarrow \quad \mathbf{X}^{\nabla} = \mathbf{Y}^{\nabla} \odot F'(\mathbf{X})
\end{aligned}$$

[1]https://github.com/michelexyz/DL-group-assignments

Where $F'(\mathbf{X})$ is a function that applies element-wise f' to the elements of $\mathbf{X}$.

question 3 Having defined $\mathbf{Y} = \mathbf{W}\mathbf{X}^T, \quad \mathbf{W} \in \mathbb{R}^{m\times f}, \quad \mathbf{X} \in \mathbb{R}^{n\times f}$, the derivation for $\mathbf{W}^\nabla$ is the following:

$$\begin{aligned}
W^\nabla_{i,j} &= \sum_{k,l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial Y_{k,l}}{\partial W_{i,j}} \\
&= \sum_{k,l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial \left[\sum_z W_{k,z} X^T_{z,l}\right]}{\partial W_{i,j}} \\
&= \sum_{k,l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial \left[W_{k,j} X^T_{j,l}\right]}{\partial W_{i,j}} \\
&= \sum_{l} \frac{\partial l}{\partial Y_{k,l}} \frac{\partial \left[W_{i,j} X^T_{j,l}\right]}{\partial W_{i,j}} \\
&= \sum_{l} Y^\nabla_{i,j} X^T_{j,l} = \sum_{k,l} Y^\nabla_{i,j} X_{j,l} \quad \rightarrow \quad \mathbf{W}^\nabla = \mathbf{Y}^\nabla \mathbf{X}
\end{aligned}$$

$\mathbf{X}$ is transposed in the forward to match with the dimensions of W that is defined as $W \in \mathbb{R}^{m\times f}$, this of course affects the derivation of the gradient.

question 4 Having defined $f(\mathbf{x}) = \mathbf{Y}$ where $Y_{i,j} = x_i$ and $\mathbf{x} \in \mathbb{R}^h, \quad \mathbf{Y} \in \mathbb{R}^{h\times 16}$. The derivation for $\mathbf{X}^\nabla$ is the following.

$$\begin{aligned}
X^\nabla_i &= \sum_j Y^\nabla_{i,j} \frac{\partial Y_{i,j}}{\partial X_i} \\
X^\nabla_i &= \sum_j Y^\nabla_{i,j} \frac{\partial X_i}{\partial X_i} \\
&= \sum_j Y^\nabla_{i,j} \cdot 1 \quad \rightarrow \quad \mathbf{X}^\nabla = \mathbf{Y}^\nabla \mathbf{1}_{16}
\end{aligned}$$

Where $\mathbf{1}_{16}$ is a vertical vector of ones of length 16.

Part 1

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 pseudo code 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 pseudo code `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 $W_{in} \rightarrow H_{in}$. The size of the output is then $(C_{out}, H_{out}, W_{out})$ where C_{out} can be chosen to be any number $\in \mathbb{N}^+$. K, S and P are the kernel size, stride and padding respectively[2]. *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 pseudo code 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]
```

[2]I have committed 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.

```
                    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 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: $\mathbf{W} = W_{c_{out}ch_kw_k}$
- Kernel matrix reshaped and transposed: $\tilde{\mathbf{W}} = \tilde{W}_{kc_{out}} =$
  $= \mathbf{W}.\text{reshape}(c_{out}, c * h_k * w_k).\text{transpose}()$
- 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.

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

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[3].

[3]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 $\mathbf{W}^{\nabla}$:

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

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[4], namely: $\partial U_{abc}/\partial X_{\alpha\beta\gamma\eta} = \delta_{a\alpha}\delta_{b\beta}\delta_{c,\gamma\eta}$[5]. This relationship assumes a certain fixed kernel size, that $\mathbf{X}$ is already padded, if needed, and a suitable choice of padding 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).

[4]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).

[5]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).

Now we're ready to do $\mathbf{U}^{\nabla}$:

$$[\mathbf{U}^{\nabla}]_{\alpha\beta\gamma} \equiv \frac{\partial \mathcal{L}}{\partial U_{\alpha\beta\gamma}} = \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \sum_{efg} \frac{\partial Y_{abcd}}{\partial Y'_{efg}} \sum_{m} \frac{\partial \tilde{U}_{efm}}{\partial U_{\alpha\beta\gamma}} \tilde{W}_{mg} =$$

$$= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \sum_{efg} \delta_{ae}\delta_{bg}\delta_{cd,f}\delta_{e\alpha}\delta_{f\gamma}\tilde{W}_{\beta g} =$$

$$= \sum_{abcd} \frac{\partial \mathcal{L}}{\partial Y_{abcd}} \delta_{a\alpha}\delta_{cd,\gamma}\tilde{W}_{\beta b} =$$

$$= \sum_{b} \frac{\partial \mathcal{L}}{\partial Y_{\alpha b,\gamma}} \tilde{W}_{\beta b}$$

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
```

```python
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 $\mathbf{W}$.

Part 2

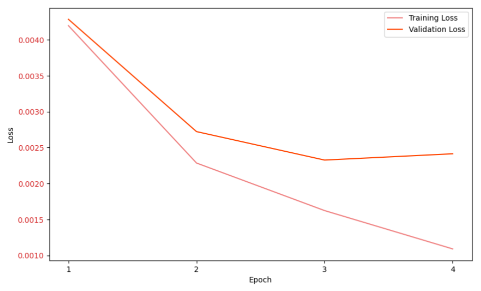
question 7 We load all the data in memory by setting the batch size of the data loader to the same size of the entire dataset. Therefore, when we call the `next` function on the loader we retrieve a batch containing all the data points. The Python implementation is as follows:

```
    train_loader = DataLoader(train_data, batch_size=len(train_data))
    test_loader = DataLoader(test_data, batch_size=len(test_data))

    x_train_data, y_train_data = next(iter(train_loader))
    x_test_data, y_test_data = next(iter(test_loader))
```

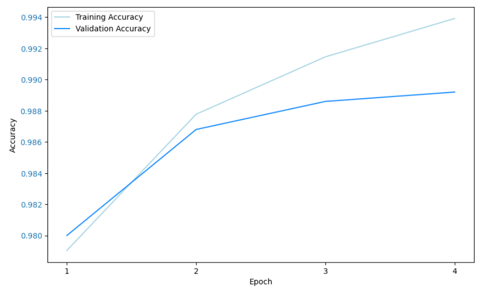
We are not going to use the test data yet, but are splitting the train data into 50 000 training instances and 10 000 validation instances. This is simply done with the following code:

```
    x_train_data, x_val_data = x_train_data[:50000], x_train_data[50000:]
    y_train_data, y_val_data = y_train_data[:50000], y_train_data[50000:]
```

Finally, the training loop is implemented with the function `train` displayed in the appendix together with implementation of the network. We trained the network with a batch size of 16, Adam optimizer, and learning rate $\alpha = 0.001$. Figure 1 shows the resulting performances across 4 epochs.



(a) Training and validation loss computed after each epoch



(b) Training and validation accuracy computed after each epoch

Figure 1: Initial performance of the network defined for question 7 (see appendix for implementation). We used a batch size of 16 and $\alpha = 0.001$

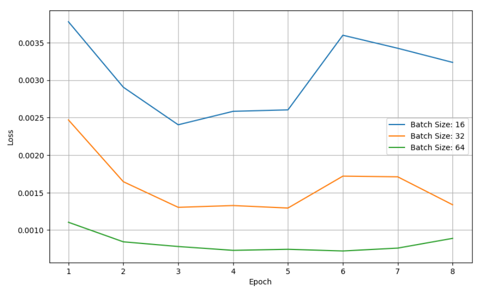
Since in figure 1 we display the loss and accuracy only after we have fully trained the network on one epoch, in figure X (see appendix) we display the loss over the single batches within the first epoch to give a better measure of the network learning process from the start.

question 8 Figure 2 shows the performance of the network defined in question 7 over the validation set for eight epochs and three different batch sizes.

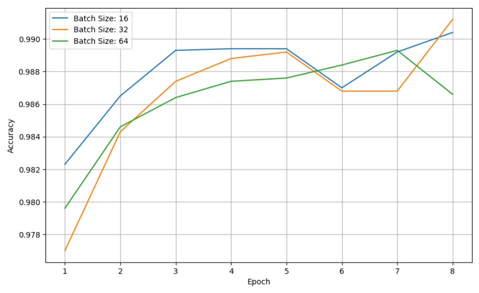
We can observe that even though we have clearly a better loss for an increased batch size, the validation accuracy it's comparable across all epochs for different settings. Finally, the best accuracy a_* we get is $99\% \lesssim a_*$, for a 8 epochs and a batch size of 32. This result is not significant given the smallest sample size of one.

Question 9 We player around a bit with various augmentations available in torch. The final set of augementations are as follows :

```
train_transform = transforms.Compose([
        transforms.RandomAffine(degrees=15, translate=(0.1, 0.1), scale=(0.9, 1.1)),  # Rotation,
        transforms.RandomPerspective(distortion_scale=0.1, p=0.5),  # Perspective distortion
        transforms.GaussianBlur(kernel_size=3, sigma=(0.1, 1.0)),  # Add Gaussian blur
        transforms.ToTensor(),  # Convert to tensor
        transforms.Normalize((0.5,), (0.5,))  # Normalize to [-1, 1]

    ])
```



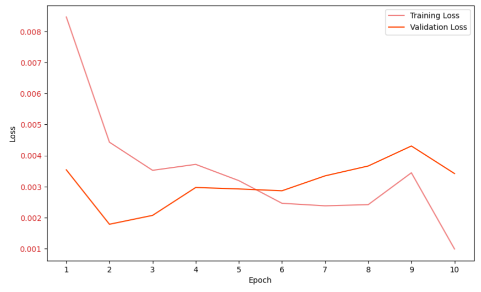
(a) Comparison of the losses



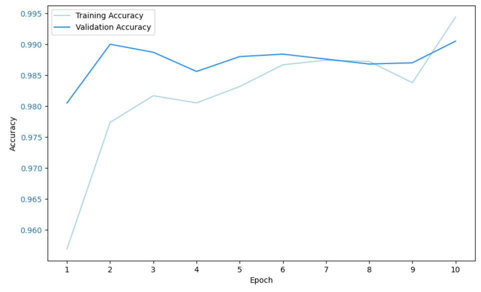
(b) Comparison of the accuracies

Figure 2: Performance comparison over different batch sizes of the network defined for question 7 (see appendix for implementation). We used a batch size in the set $\{16, 32, 64\}$ and $\alpha = 0.001$

The random affine transformation randomly rotates the picture within the specified range of degree, translates it which means shifting in the x and y dimension (within the specified range), and scales it, which is a random zoom (in or out). We then applied some perspective distortion, which simulates taking a different viewing angle. This seemed really interesting as a concept, even though for this particular problem there probably is not a change in perspective all too often. The last transformation applied was just some good old gaussian blur, really cant go wrong with that. Interestingly, we got comparable performance by simply rotating the picture by 20 degrees with a chance of 0.5, but this set of augmentations seemed more fun. Overall we got a decent performance out of it, with the test accuracy hitting 99% after 8 epochs and a batch size of 16. The loss and accuracy curves can be observed in figure 3.



(a) Training and validation loss computed after each epoch with Augmentations



(b) Training and Test accuracy computed after each epoch with Augmentations

Figure 3: Performance for our model using augmentations. We used a batch size of 16 and $\alpha = 0.001$

Part 3

question 10 We define: a kernel $\mathbf{K}_o, \mathbf{K} \in \mathbb{R}^{5\times5\times3}$, that takes three channels as input, where o indicates the index of the output channel in the range $[0; 16)$; a padding of size $p = 1$ and stride $s = 2$. Given that the input image is a tensor, $\mathbf{X}, \mathbf{X} \in \mathbb{N}_0^{h\times w\times 3}$, the output tensor will be of size $(h_{new} \times w_{new} \times 16)$, where $h_{new} = \left\lfloor \frac{h+2p-k}{s} \right\rfloor + 1$ and $w_{new} = \left\lfloor \frac{h+2p-k}{s} \right\rfloor$, $k = 5$ is the size of the kernel. With $h = 768$ and $w = 1024$, we then have an output tensor of size $(383 \times 511 \times 16)$. If we apply the convolution to an image of size 1920x1080 where the first number, 1920, indicates the width, the resulting output tensor would have size

$(539 \times 959 \times 16)$. Finally, we could apply a convolution to an image with the same size and 8 channels, but we would need to increase the dimension of our kernel to have 8 channels as well, so that $\mathbf{K}_o, \mathbf{K} \in \mathbb{R}^{5\times5\times8}$. The resulting output size would be the same as for the 1920x1080 image with 3 channels.

Question 11 The following code show how perform mean and max pooling with PyTorch.

```python
import torch.nn.functional as F
b, c, h, w = 4, 10, 32, 32
x1 = torch.randn(b, c, h, w)

# squeeze because otherwise the shape would be (b, c, 1, 1)
x1_max_pool_func = F.max_pool2d(x1, kernel_size=x1.size()[2:]).squeeze()
x1_mean_pool_func = F.avg_pool2d(x1, kernel_size=x1.size()[2:]).squeeze()
```

The shape of the generated tensors is (b, c).

Question 12 The following is the code that shows how we resized and loaded the train data into memory.

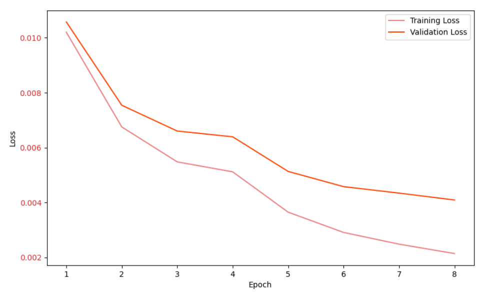
```python
varres_transform = Compose([
    Resize((28, 28)),
    Grayscale(num_output_channels=1),
    ToTensor()
])

vatters_train = ImageFolder(root='./data/MNIST/mnist-varres/train', transform=varres_transform)

# Split into training and validation sets
train_size = int(0.8 * len(vatters_train))
val_size = len(vatters_train) - train_size
train_varres_dataset, val_varres_dataset = torch.utils.data.random_split(vatters_train, [train_s

# Create data loaders
train_varres_loader = DataLoader(train_varres_dataset, batch_size=len(train_varres_dataset), shu
val_varres_loader = DataLoader(val_varres_dataset, batch_size=len(val_varres_dataset), shuffle=Fa

#load the whole dataset into tensors
x_train_resized, y_train_resized= next(iter(train_varres_loader))
x_val_resized, y_val_resized = next(iter(val_varres_loader))
```

This is similar to what we have done in question 7. The only differences are that we applied the additional transforms `Resize((28, 28))` and `Grayscale(num_output_channels=1)`, and we used `torch.utils.data.random_split` to randomly split the training data each run.

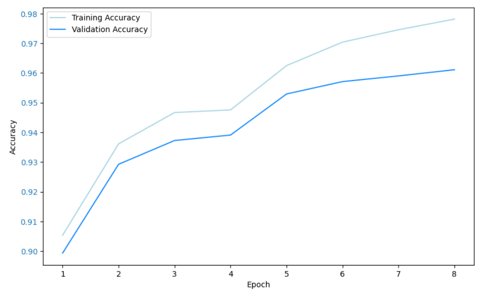
Figure 4 shows the resulting performance of the network trained over 8 epochs with a batch size of 32. For the sake of an additional and more fair comparison, we added the results of the training with a batch size of 16 in Figure 11 of the appendix.

We can see that, even though we have a small difference with respect to the loss, the validation accuracy that we reach after 8 epochs is almost the same for a batch size of 16 and 32:$\approx 96\%$. The performance is decreased as expected, in fact, we reached an accuracy of $> 98\%$ using the regular dataset after only three epochs.

Question 13 Resizing all images could lead to two potential drawbacks that we see. The first one is that it increases computational costs, so if the computational budget is tight



(a) Training and validation loss computed after each epoch



(b) Training and validation accuracy computed after each epoch

Figure 4: Performance of the network as defined in question 7 trained over the varres dataset with a resize transform to a resolution of 28x28. We used a batch size of 32 and $\alpha = 0.001$

this might be not a good choice! The second one is the potential loss of information, since resizing smaller images could lead to distortions and make valuable features such as edges disappear.

Question 14 If our data contains images of all kinds of shapes, we have to think of some kind of strategy to create batches of the same size. Resizing all of them to the same input size does not seem feasible, seeing as the original size could vary greatly! One possibility would be to introduce padding to all images, in order to equalize the dimensions of all images. Again however this could be inefficient if the image with the biggest size is way larger for example than all others, so a lot of padding would be needed. Another would be to group the images by a similar enough (however we want to define this in the particular case) image size and then resize (or pad) per batch. That way we are making sure that the images we are resizing do not differ in size all too much, so not a lot of information should be lost in the process!

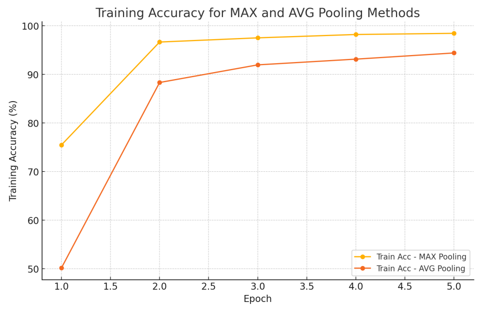
Question 15 We implmented the following function to sum over all relevant parts of a model.

```python
def count_parameters(model):
    total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    return total_params
```

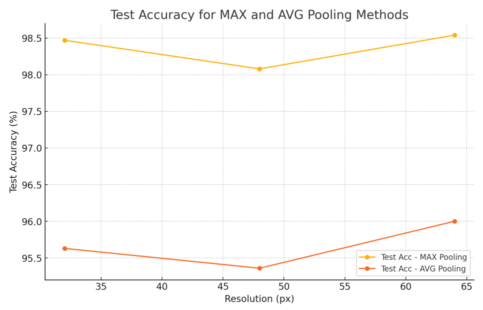
Using this we found the original model to have 29066 parameters with the variable input model having 23946 parameters for N = 64. Looping over various values for N, we found the two closes matches to be N = 81 with 29029 parameters and N = 82 with 29328 parameters respectively. Due to N = 81 being the closest match, this is the number we used going forward.

Question 16 We proceeded to implement a training loop for both variants of global pooling. The code can be found in the appendix. The results seem to be rather clear, with max pooling clearly outperforming average pooling. as can be seen in figure 5. Both max poolings training and test accuracy converged in the 98.X % range, while the average pooling stayed in the upper 95.X % range. So going forward we decided to use max pooling.

Question 17 We tuned the variable size model, finding that a learning rate of 0.0009 working best, while training on whole batches. When training both networks for 15 episodes we managed to get a Test accuracy for the variable-resolution classifier of 98.89%. The



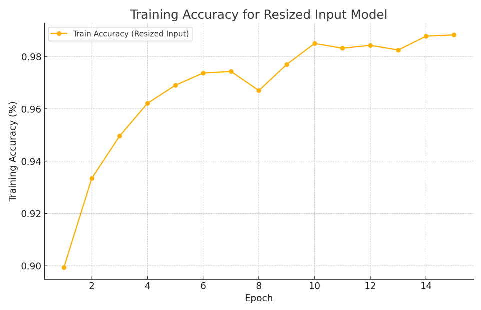
(a) Training loss computed after each epoch



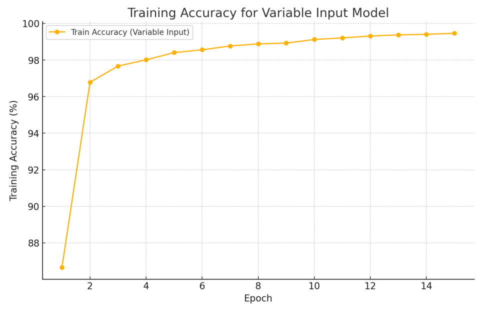
(b) Test accuracy per resolution size as input

Figure 5: Comparison of variable input model with avg and max global pooling

network from question 12 with resized images managed to get an accuracy of 96.8%. Interestingly enough the training accuracy between the two seemed to be comparable, as can be seen in figure 6, so there seems to be some information lost when resizing. This seems to



(a) Training loss per epoch for resized resolution network



(b) Training accuracy per epoch for variable resolution network

Figure 6: Comparison of variable input model with avg and max global pooling

be more severe for some digits over others, as can be seen in the confusion matrices for the variable size network (figure 7) and the original network (figure 8) respectively. .

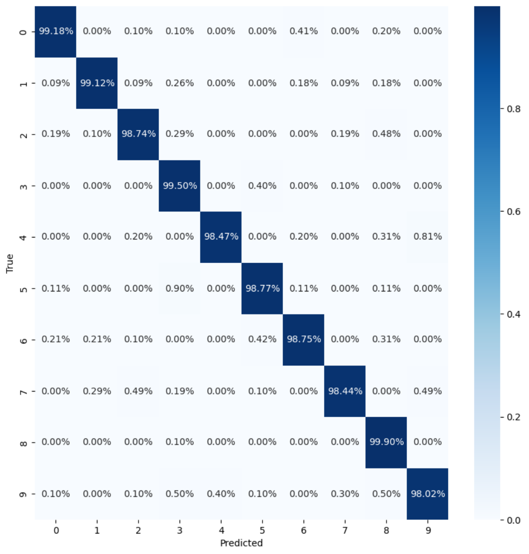


Figure 7: Confusion matrix variable-resolution network

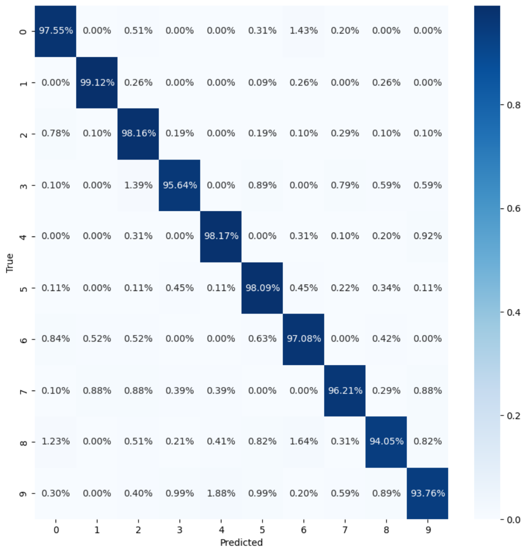


Figure 8: Confusion matrix resized-resolution network

Appendix

Part 1

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

```
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' = \tilde{U}\tilde{W}, 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) # Y
```

Part 2

Question 7

For this question we implemented the following CNN:

```
class MNISTConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        ↪  # 1 input channel, 16 output channels
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        ↪  # 16 input channels, 32 output channels
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        ↪  # 32 input channels, 64 output channels
        self.fc1 = nn.Linear(64 * 3 * 3, 10)  # Flattened to a fully
        ↪  connected layer with 10 outputs

    def forward(self, x):
        x = F.relu(self.conv1(x))  # First convolution + ReLU
        x = F.max_pool2d(x, 2)     # Max pooling 2x2
        x = F.relu(self.conv2(x))  # Second convolution + ReLU
        x = F.max_pool2d(x, 2)     # Max pooling 2x2
        x = F.relu(self.conv3(x))  # Third convolution + ReLU
        x = F.max_pool2d(x, 2)     # Max pooling 2x2
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc1(x)            # Fully connected layer
        return x
```

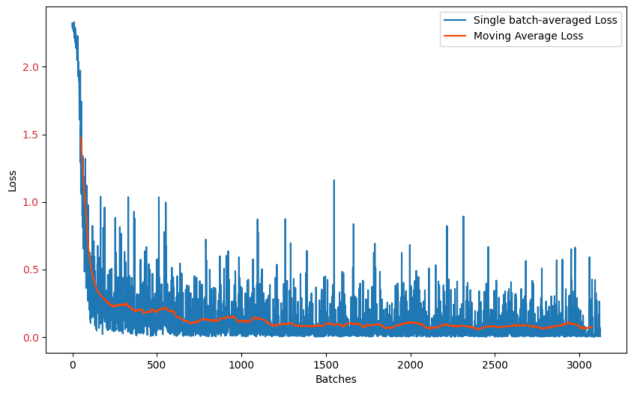


Figure 9: Loss averaged over the single batch within the epoch of training process performed for question 7

The latter was trained with following function, it assumes that the train and validation data was stored into tensors.

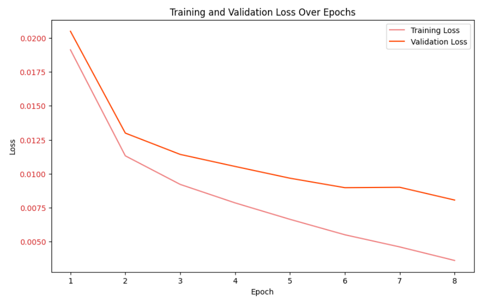
```
def train(model, x_train, y_train, x_val, y_val, optimizer, criterion,
↪   epochs=10, batch_size=64):

    first_epoch_running_loss = []
    train_evaluations = np.zeros((epochs, 2))
    val_evaluations = np.zeros((epochs, 2))

    for epoch in range(epochs):
        model.train()
        for i in tqdm(range(0, len(x_train), batch_size), desc=f'Batches
        ↪   for epoch {epoch + 1}/{epochs}'):

            x_batch = x_train[i:i + batch_size]
            y_batch = y_train[i:i + batch_size]

            optimizer.zero_grad()
            output = model(x_batch)
            loss = criterion(output, y_batch)
            loss.backward()
            optimizer.step()

            if epoch == 0:
                first_epoch_running_loss.append(loss.item())

        train_loss, train_acc = calculate_loss_and_accuracy(model, x_train,
        ↪   y_train, criterion, batch_size)
        val_loss, val_acc = calculate_loss_and_accuracy(model, x_val,
        ↪   y_val, criterion, batch_size)

        train_evaluations[epoch] = [train_loss, train_acc]
        val_evaluations[epoch] = [val_loss, val_acc]

    return first_epoch_running_loss, train_evaluations, val_evaluations
```

This function stores and returns both the loss and accuracy of the train and validation set over the epochs, this data is then used for the plots.

Part 3

Question 12

For a fair comparison with the network trained in question 7 with the regular MNIST dataset we also train the network over the resized varres dataset with the same parameters. Figure 11 shows the results of these process. Finally, figure Z shows the loss averaged over the


(a) Training and validation loss computed after each epoch

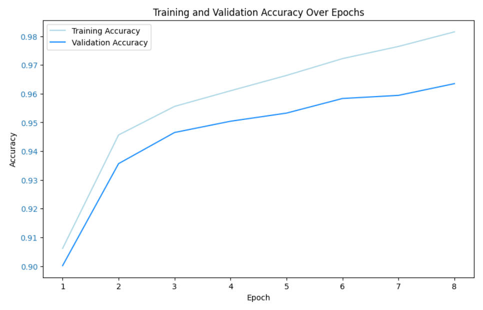
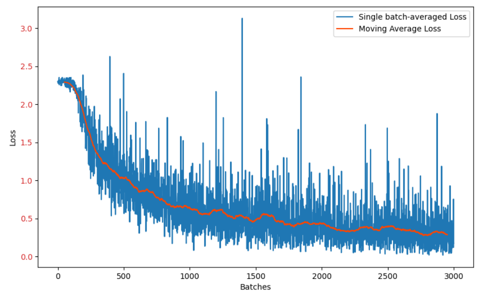

(b) Training and validation accuracy computed after each epoch

Figure 10: Performance of the network as defined in question 7 trained over the varres dataset with a resize transform to a resolution of 28x28. We used a batch size of 16 and $\alpha = 0.001$

single batch for the training performed with a batch size of 16 and 32 respectively.


(a) Results for a batch size 16

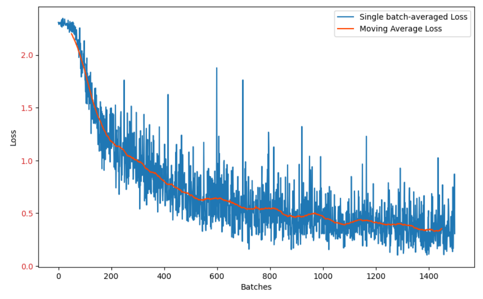

(b) Results for a batch size 32

Figure 11: Loss averaged over the single batch within the epoch of training process performed for question 12

Question 16

Code for comparing between two pooling methods:

```
def train_and_compare_pooling(train_data_by_resolution, test_data_by_resolution, num_epochs=5):
    pooling_methods = ['max', 'avg']
    results = {}
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    for pooling_type in pooling_methods:
        print(f"\nTraining with {pooling_type.upper()} pooling:")

        # Initialize model, criterion, and optimizer
        model = VariableInputNetwork(pooling_type=pooling_type).to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.001)

        # Training
        for epoch in range(num_epochs):
            print(f"Epoch {epoch + 1}/{num_epochs}")
            for res, (train_data, train_labels) in train_data_by_resolution.items():
                print(f"Training on resolution {res}x{res}:")
                train_loss, train_acc = train_on_resolution(model, train_data, train_labels, crit
                print(f"  Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%")
            print("-" * 50)

        # Evaluation
        print(f"\nEvaluating {pooling_type.upper()} pooling on test data:")
        pooling_results = {}
        for res, (test_data, test_labels) in test_data_by_resolution.items():
            test_loss, test_acc = evaluate_on_resolution(model, test_data, test_labels, criterion
            print(f"Resolution {res}x{res}: Test Loss: {test_loss:.4f}, Test Acc: {test_acc:.2f}%
            pooling_results[res] = (test_loss, test_acc)

        results[pooling_type] = pooling_results

    # Print Summary
    print("\nComparison of Pooling Methods:")
    for pooling_type, pooling_results in results.items():
        print(f"\n{pooling_type.upper()} Pooling:")
        for res, (loss, acc) in pooling_results.items():
            print(f"Resolution {res}x{res}: Test Loss = {loss:.4f}, Test Acc = {acc:.2f}%")

    return results
```