

Submission Assignment 3A

Names (Student IDs): Hong Huo (hho270), Manuel G. Scanferla (mgi248) & Ángel A. Gil (agi239)

Please provide (concise) answers to the questions below. If you don't know an answer, please leave it blank. If necessary, please provide a (relevant) code snippet. If relevant, please remember to support your claims with data/figures.

Question 1

Write a pseudo-code for how you would implement this with a set of nested for loops. The convolution is defined by a set of weights/parameters which we will learn. How do you represent these weights?

Answer We are given the following elements:

- x : input tensor with dimensions $(batch_size, input_channels, input_width, input_height)$
- An amount of *padding*, a number of *output_channels*, a *kernel_size*, and a *stride*.
- y : output tensor with dimensions $(batch_size, output_channels, output_width, output_height)$

We can represent our non vectorized version of the convolutional layer with the following pseudocode:

Algorithm 1 Non vectorized implementation of convolutional layer

```

procedure CONVOLUTION( $x, padding, output\_channels, kernel\_size, stride$ )
   $Output\_width \leftarrow (input\_width + 2 * padding - kernel\_width) // stride + 1$ 
   $Output\_height \leftarrow (input\_height + 2 * padding - kernel\_height) // stride + 1$ 
  load  $x$ , initialize weights  $\equiv kernel$ , initialize  $y$  (zeros)
   $x' \leftarrow Pad(x, padding)$ 
  Nested loop:
  for  $b$  in  $batch\_size$  do
    for  $c$  in  $output\_channels$  do
      for  $i$  in  $range(0, input\_width + 2 * padding, stride)$  do
        for  $j$  in  $range(0, input\_height + 2 * padding, stride)$  do
          for  $k$  in  $input\_channels$  do
            for  $l$  in  $kernel\_width$  do
              for  $m$  in  $kernel\_height$  do
                 $y[b, c, i, j] \leftarrow x'[b, k, i + l, j + m] * kernel[c, k, l, m]$ 

```

Notice that we divided the *kernel_size* into *kernel_width* and *kernel_height*, although we could also assume that the filter is squared. The weights of this convolution are represented as a 4-dimensional tensor with dimensions $(output_channels, input_channels, kernel_width, kernel_height)$. This tensor is often called the "kernel" or "filter" of the convolutional layer.

Question 2

For a given input tensor, kernel size, stride and padding (no dilutions) work out a general function that computes the size of the output.

Answer The *batch_size* represents the number of times we would do the training in a single forward and backward pass i.e. it stays the same as that of the input. The *output_channels* are given. The size of the *width* and *height* of the output are obtained as follows:

$$\text{output_height} = \frac{[\text{input_height} - \text{kernel_height} + 2 * \text{padding}]}{\text{stride}} + 1$$

$$\text{output_width} = \frac{[\text{input_width} - \text{kernel_width} + 2 * \text{padding}]}{\text{stride}} + 1$$

Why is that so? For each of the spatial dimensions, we have a given input size, which is augmented by padding from both sides. Then, we need to subtract the size of the kernel, because it has to be applied from the leftmost to the rightmost part of x_{padded} , so there are that many less options. Finally, we divide by the stride (and sum 1) since it represents the size of the skips that will be made when computing the patches along the given dimension. We use the square brackets although the aim of padding is precisely avoiding a not exact division.

Lets work out an example assuming that $\text{input_height} = \text{input_width} = 227$, $\text{kernel_height} = \text{kernel_width} = 11$, $\text{padding} = 0$ and $\text{stride} = 4$. Using the methods above:

$$\text{output_height} = \frac{[227 - 11 + 2 * 0]}{4} + 1 = 55 \qquad \text{output_width} = \frac{[227 - 11 + 2 * 0]}{4} + 1 = 55$$

We obtain an output tensor with size $(\text{batch_size}, \text{output_channels}, 55, 55)$ where the first two parameters are given.

Question 3

Write a naive (non-vectorized) implementation of the unfold function in pseudocode. Include the pseudocode in your report.

Answer We can represent our naive non vectorized version dividing the pseudocode in two parts: the first one applies the padding to the input tensor; the second one extracts the patches from the padded input tensor.

Algorithm 2 Non vectorized implementation of the unfold function: part one

```

procedure PAD(x,padding)
  input_padded ← batch size, channels, width + 2 * padding, height + 2 * padding
  loop:
    for b in batch size do
      for c in channels do
        for i in width do
          for j in height do
            input_padded[b, c, i + padding, j + padding] ← input[b, c, i, j]
```

Algorithm 3 Non vectorized implementation of the unfold function: part two

```

procedure UNFOLD(x_padded,kernel_size,stride)
  patches ← empty list
  loop:
    for b in batch_size do
      for i in range(0, input_width, stride) do
        for j in range(0, input_height, stride) do
          patch ← empty list
          for c in channels do
            for l in kernel_width do
              for m in kernel_height do
                append x_padded[b, c, i+l, j+m] to patch
          append patch to patches
```

This function extracts patches of size $k = (\text{input_channels}, \text{kernel_width}, \text{kernel_height})$ using the given stride, and returns the patches in a matrix with dimensions $(\text{batch_size}, k, p)$. The variable p is the total number of patches, and k is the number of features per patch.

Questions 4 & 5

Work out the backward with respect to the kernel weights \mathbf{W} and the backward with respect to the input \mathbf{X} .

Answer Figure 1 shows the computation graph for this question. In order to work out the derivatives, let's first discuss the dimensions of the tensors and the operations that take place in the proposed convolution:

- **Tensors**

- \mathbf{X} : input tensor with 4 dimensions (*batch_size*, *input_channels*, *input_width*, *input_height*)
- \mathbf{U} : unfolded input tensor - *Unfold*(\mathbf{X}) - with 3 dimensions (*batch_size*, *patches*, *kernel_features*)
- \mathbf{W} : weight matrix (kernel whose features have been stretched into a vector to match the unfolded input). Its 2 dimensions are (*kernel_features*, *output_channels*)
- \mathbf{Y}' : Output tensor resulting from the multiplication of \mathbf{U} and \mathbf{W} . Its 3 dimensions are (*batch_size*, *patches*, *output_channels*)
- \mathbf{Y} : resized output tensor - *Reshape*(\mathbf{Y}') - so that its rows become the pixels of the output tensor. Its 4 dimensions are (*batch_size*, *output_channels*, *output_width*, *output_height*)
- l : loss scalar. Although this convolution layer would not be directly connected to the loss (one or many fully connected layers are usually implemented after the convolution layers), we add l so that we can give some context to the gradient of \mathbf{Y} .

- **Operations**

- *Unfold* : Used to extract all of the patches from \mathbf{X} and obtain \mathbf{U} as specified in Question 3.
- *Reshape*: Used to turn the dimension *patches* of \mathbf{Y}' into (*output_width*, *output_height*), obtaining \mathbf{Y} .

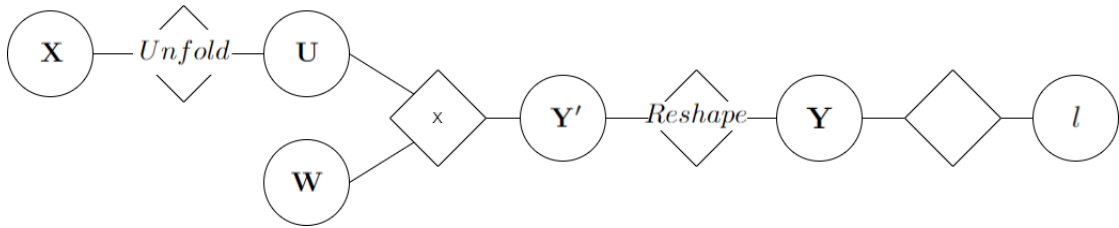


Figure 1: computation graph for Questions 4 & 5

The last steps before working out the derivatives are setting aside the dimensions that do not come into play (for notation simplification) and discussing \mathbf{Y}'^∇ in terms of \mathbf{Y}^∇ . As for the dimensions, we can get rid of *batch_size*, *input_channels* and *output_channels* for now, since they will remain unaltered during the operations. The tensors' shapes would now be: $\mathbf{X} \equiv (w_i, h_i)$ $\mathbf{U} \equiv (k, p)$ $\mathbf{W} \equiv (k)$ $\mathbf{Y}' \equiv (p)$ $\mathbf{Y} \equiv (w_o, h_o)$.

As for \mathbf{Y}'^∇ in terms of \mathbf{Y}^∇ , the reshape operation takes the row dimension $p \in \text{patches}$ and rearranges its values into a matrix of dimensions $w_o \times h_o \in \text{weight_output} \times \text{height_output}$. The derivative of an element $y_{w_o h_o}$ of \mathbf{Y} with respect to an element y'_p of \mathbf{Y}' would be 1 when the row-major ordering of the reshape function of $y_{w_o h_o}$ is exactly y'_p and 0 otherwise, since there are no transformations (there is only a rearrangement). In other words, it would be the indicator function that maps \mathbf{Y} back to \mathbf{Y}' . Then, the derivative of \mathbf{Y}'^∇ in terms of \mathbf{Y}^∇ would consist on applying the inverse reshaping operation on \mathbf{Y}^∇ :

- $\mathbf{Y}'^\nabla = \text{reshape}(\mathbf{Y}^\nabla)$, where *reshape* turns $\text{weight_output} \times \text{height_output}$ back to *patches* and $\mathbf{Y}^\nabla = \frac{\partial l}{\partial \mathbf{Y}}$.

We are finally in position to work out \mathbf{W}^∇ :

$$\bullet \mathbf{W}_i^\nabla = \sum_p \mathbf{Y}'^\nabla_p \frac{\partial \mathbf{Y}'_p}{\partial \mathbf{W}_i} = \sum_p \mathbf{Y}'^\nabla_p \frac{\partial \sum_k \mathbf{U}_{pk} \mathbf{W}_k}{\partial \mathbf{W}_i} = \sum_{pk} \mathbf{Y}'^\nabla_p \frac{\partial \mathbf{U}_{pk} \mathbf{W}_k}{\partial \mathbf{W}_i} = \sum_p \mathbf{Y}'^\nabla_p \frac{\partial \mathbf{U}_{pi} \mathbf{W}_i}{\partial \mathbf{W}_i} = \boxed{\sum_p \mathbf{Y}'^\nabla_p \mathbf{U}_{pi}}$$

Then, the vectorization is $\mathbf{W}^\nabla = \mathbf{Y}'^\nabla \mathbf{U}$.

In order to work out \mathbf{X}^∇ , we first need to discuss \mathbf{U}^∇ .

$$\bullet \mathbf{U}_{ij}^\nabla = \sum_p \mathbf{Y}'_p \frac{\partial \mathbf{Y}'_p}{\partial \mathbf{U}_{ij}} = \sum_p \mathbf{Y}'_p \frac{\partial \sum_k \mathbf{U}_{pk} \mathbf{W}_k}{\partial \mathbf{U}_{ij}} = \sum_{pk} \mathbf{Y}'_p \frac{\partial \mathbf{U}_{pk} \mathbf{W}_k}{\partial \mathbf{U}_{ij}} = \mathbf{Y}'_i \frac{\partial \mathbf{U}_{ij} \mathbf{W}_j}{\partial \mathbf{U}_{ij}} = \boxed{\mathbf{Y}'_i \mathbf{W}_j}$$

In order to get the vectorized version of \mathbf{U}^∇ , we can compute it as the outer product of the gradient for \mathbf{Y}' , which we've computed already, and the vector \mathbf{W} , which we can save during the forward pass:

$$\mathbf{U}^\nabla = \mathbf{Y}'^\nabla \mathbf{W}^T$$

Lets finally work out \mathbf{X}^∇ :

$$\bullet \mathbf{X}_{ij}^\nabla = \sum_{kl} \mathbf{U}_{kl}^\nabla \frac{\partial \mathbf{U}_{kl}}{\partial \mathbf{X}_{ij}} = \sum_{kl} \mathbf{U}_{kl}^\nabla \frac{\partial \sum_{\tau=-l}^l \sum_{v=-m}^m \mathbf{X}_{k+\tau, l+v}}{\partial \mathbf{X}_{ij}} = \sum_{kl} \sum_{\tau=-l}^l \sum_{v=-m}^m \mathbf{U}_{kl}^\nabla \frac{\partial \mathbf{X}_{k+\tau, l+v}}{\partial \mathbf{X}_{ij}}$$

We are almost done, but lets discuss in words what the derivative is going to look like: since it is the derivative of \mathbf{X} with respect to itself, it will be 1 when $k + \tau = i$ and $l + v = j$. This will happen a total of $2l * 2m$ times, with those values being the height and width of the kernel minus 1 divided by 2 (f.i.: $k_h = 5 \implies l = \frac{5-1}{2} = 2$). Then, the derivative of the unfold operation in scalar terms can be written like this:

$$\bullet \mathbf{X}_{ij}^\nabla = \sum_{ab} \mathbf{U}_{ab}^\nabla \quad \text{where } a \in [i-l, i+l] \text{ and } b \in [j-m, j+m]$$

We also have to take into account that, for example, for $\mathbf{X}_{1,1}^\nabla$ we cannot take negative terms of \mathbf{U}^∇ , so a and b would have to be defined carefully for the edges.

The vectorization would then be $\mathbf{X}^\nabla = \sum_{ab} \mathbf{U}_{ab}^\nabla$, with a and b defined as explained.

Question 6

Implement your solution as a PyTorch Function. (If you didn't manage question 5, just set the backward for the input to None).

Answer Figure 2 shows the implementation of the forward and backward passes that we have deduced following the previous questions (we hope you do not have to zoom in and out too much, but putting the captions one on top of the other took way too much space!). The method can be found in the script called *Q6.py*.

```
@staticmethod
def forward(ctx, input_batch, kernel, stride=1, padding=1):
    # store objects for the backward
    ctx.save_for_backward(input_batch)
    ctx.save_for_backward(kernel)

    # Working out the different dimensions
    b, c_in, h_in, w_in = input_batch.size()
    c_out, h_k, w_k = kernel.size()
    h_out = (h_in - h_k + 2 * padding) // stride + 1
    w_out = (w_in - w_k + 2 * padding) // stride + 1

    # Extracting patches (U)
    u = F.unfold(input_batch, (h_k, w_k), dilation=1, padding=padding, stride=stride)
    b, k, p = u.size()

    # Computing Y'
    uflat = torch.permute(u, (0, 2, 1))
    uflat = uflat.reshape(b * p, k)
    weights = torch.randn(c_out, k)
    yprime = torch.inner(uflat, weights) # (b*p, k) x (k, c_out) --> (b*p, c_out)

    # Store final U and W for the backward
    ctx.save_for_backward(uflat)
    ctx.save_for_backward(weights)

    # Computing Y
    y = yprime.reshape(b, h_out, w_out, c_out)
    output_batch = torch.permute(y, (0, 3, 1, 2))
    return output_batch
```

```
@staticmethod
def backward(ctx, grad_output):
    # retrieve stored objects
    input, kernel, uflat, weights, u = ctx.saved_tensors

    # Working out the different dimensions
    b, c_in, h_in, w_in = input.size()
    c_out, h_k, w_k = kernel.size()
    b, c_out, h_out, w_out = grad_output.size()
    b, k, p = u.size()

    # Gradient of Y'
    reorder_grad_output = torch.permute(grad_output, (0, 2, 3, 1))
    yprime_grad = reorder_grad_output.reshape(b * h_out * w_out, c_out) # (b*p, c_out)

    # Gradient of W
    kernel_grad = torch.inner(yprime_grad.T, uflat.T) # (c_out, b*p) x (b*p, k) --> (c_out, k)

    # Gradient of U
    u_grad = torch.matmul(yprime_grad.T, weights.T) # (b*p, c_out) x (c_out, k) --> (b*p, k)

    # Gradient of X
    u_grad_deflat = uflat.reshape(b, p, k)
    u_grad_orig_shape = torch.permute(u_grad_deflat, (0, 2, 1))
    fold = torch.nn.Fold((h_in, w_in), (h_k, w_k), dilation=1, padding=0, stride=1)
    input_batch_grad = fold(u_grad_orig_shape)

    return input_batch_grad, kernel_grad, None, None
```

Figure 2: Forward (left) and backward (right) of our PyTorch Function

Question 7

Use the dataloaders to load both the train and the test set into large tensors: one for the instances, one for the labels. Split the training data into 50 000 training instances and 10 000 validation instances. Then write a training loop that loops over batches of 16 instances at a time.

Answer Figure 3 shows the way we load the train and test set into large tensors and how we split the training data into 50000 training instances and 10000 validation instances (left caption), together with the training loop that loops over batches of 16 instances at a time (right caption). The notebook *Q789.ipynb* contains our implementation of the whole network.

```
# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = transform,
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = transform,
    download = True
)

# Split the training data into 50 000 training instances and 10 000 validation instances
traindata, valdata = train_test_split(train_data, test_size=10000, random_state=42)
batch_size = 16
trainloader = torch.utils.data.DataLoader(traindata, batch_size=batch_size, shuffle=True, num_workers=2)
valloader = torch.utils.data.DataLoader(valdata, batch_size=batch_size, shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True, num_workers=2)

def train(dataloader, model, loss_func, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        y_hot = F.one_hot(y, 10)
        y_hot = y_hot.float()
        # y_hot = torch.zeros(batch_size, 10)
        # y_hot[range(y_hot.shape[0]), y]=1

        X, y_hot = X.to(device), y_hot.to(device)
        # Compute prediction error
        pred = model(X)
        loss = loss_func(pred, y_hot)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            # print(f"Loss: {loss:>7f} [{current:>5d}/{size:>5d}]*")
    return loss
```

Figure 3: Left, how we load and split data for our network. Right, training loop

Question 8

Build this network and tune the hyperparameters until you get a good baseline performance you are happy with. You should be able to get at least 95% accuracy. If training takes too long, you can reduce the number of channels in each layer.

Answer Following the guidelines of the assignment, we obtain the model shown in Figure (4).

```
# Define the model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1,16,3,1,1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16,32,3,1,1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(32,64,3,1,1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.out = nn.Linear(64*3*3, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.size(0), -1) # Flatten , same as x = torch.flatten(x, 1)
        output = self.out(x)
        return output

model = CNN()
```

Figure 4: Model cnn Q8

To tune the hyperparameters and get a good performance we run our network for different learning rates and batch sizes. The tests considered the following learning rates: 0.00001; 0.0001, 0.001, 0.01; and batch sizes: 4; 8; 16.

The plots in Figure 5 represent respectively the performance of the model considering the loss over time for different learning rates and for different batch sizes. From the two plots and some extra texting, we ended up

choosing a learning of 0,001 and a batch size of 16. Running the model with these hyperparameters we obtain an accuracy of 99.2% after 10 epochs, as shown in Figure 6.

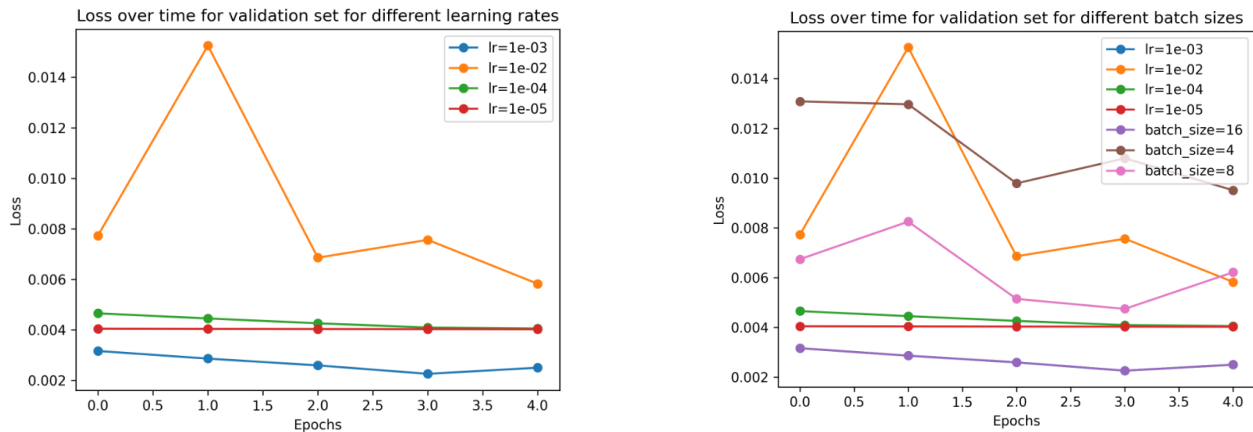


Figure 5: Left, loss curves for different learning rates. Right, added different batch sizes

```
Epoch 5
-----
Test Error:
Accuracy: 98.8%, Average loss: 0.005130

Epoch 6
-----
Test Error:
Accuracy: 99.2%, Average loss: 0.003669

Epoch 7
-----
Test Error:
Accuracy: 99.2%, Average loss: 0.004128

Epoch 8
-----
Test Error:
Accuracy: 99.1%, Average loss: 0.004014

Epoch 9
-----
Test Error:
Accuracy: 99.1%, Average loss: 0.003809

Epoch 10
-----
Test Error:
Accuracy: 99.2%, Average loss: 0.003740
```

Figure 6: Accuracy of baseline considering the last 6 epochs

Question 9

Add some data augmentations to the data loader for the training set. Why do we only augment the training data? Play around with the augmentations available in torchvision. Try to get better performance than the baseline. Once you are happy with your choice of augmentations, run both the baseline and the augmented version on the test set and report the accuracies in your report.

Answer We only augment the training data and not the validation or test data because augmenting the training data can help reduce overfitting, as it introduces more variation and noise into the training set, making the model more resilient to small changes in the input data. Moreover, augmenting the training data can help make the training process more efficient, as it allows the model to learn from more data in less time. Augmenting the validation and test data can introduce bias and distort the results.

For the augmentation of our training data after different tests we chose ColorJitter, an augmentation provided by pytorch that randomly change the brightness, contrast, saturation and hue of an image. The code is shown in Figure 7.

Figure 8 show the results of running both the baseline and the augmented version on the validation sets we obtain a better accuracy for the latter, with an accuracy of 99.4% after 5 epochs, while the baseline version has an accuracy of 99.2% after 5 epochs. Once the models were trained, we tested the accuracy for the test set and

obtained 99.4% and 99.1% respectively, which comes out really close to the previous accuracies obtained with the validation set.

```
transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,))])

# data augmentation for training data
train_transform = transforms.Compose([
    # transforms.ToPILImage(),
    # transforms.CenterCrop(21),
    # transforms.RandomRotation(30),
    # transforms.GaussianBlur(1),
    # transforms.RandomAdjustSharpness(12),
    # transforms.RandomAutocontrast(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    # transforms.RandomAffine(degrees=20, translate=(0.1,0.1), scale=(0.9, 1.1)),
    # transforms.AugMix(),
    # transforms.RandomHorizontalFlip(),
    # transforms.RandomCrop(32, 4),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
])

# Download and load the training data
trainset0 = datasets.MNIST('data_augm', download=True, train=True, transform=train_transform)
testset = datasets.MNIST('data_augm', download=True, train=False, transform=transform)

# Also create a validation set
trainset_augm, valset_augm = random_split(trainset0, [50000, 10000], generator=torch.Generator().manual_seed(42))
# trainset, valset = train_test_split(trainset0, test_size=10000, random_state=42)

batch_size = 16

trainloader_augm = torch.utils.data.DataLoader(trainset_augm, batch_size=batch_size, shuffle=True, num_workers=2)
valloader_augm = torch.utils.data.DataLoader(valset_augm, batch_size=batch_size, shuffle=True, num_workers=2)
testloader_augm = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=True, num_workers=2)
```

Figure 7: Code for augmentation Q9

Epoch 1 ----- Test Error: Accuracy: 99.6%, Average loss: 0.002557	Epoch 1 ----- Test Error: Accuracy: 99.1%, Average loss: 0.006623
Epoch 2 ----- Test Error: Accuracy: 99.4%, Average loss: 0.002688	Epoch 2 ----- Test Error: Accuracy: 99.0%, Average loss: 0.007429
Epoch 3 ----- Test Error: Accuracy: 99.5%, Average loss: 0.002517	Epoch 3 ----- Test Error: Accuracy: 99.2%, Average loss: 0.008580
Epoch 4 ----- Test Error: Accuracy: 99.4%, Average loss: 0.002333	Epoch 4 ----- Test Error: Accuracy: 99.1%, Average loss: 0.008206
Epoch 5 ----- Test Error: Accuracy: 99.4%, Average loss: 0.002619	Epoch 5 ----- Test Error: Accuracy: 99.2%, Average loss: 0.007959

Figure 8: Left, accuracy of the augmented version. Right, accuracy of baseline version

Question 10

Assume we have a convolution with a 5×5 kernel, padding 1 and stride 2, with 3 input channels and 16 output channels. We apply the convolution to an image with 3 channels and resolution 1024×768 . What are the dimensions of the output tensor? What if we apply the same convolution to an image with 3 channels and resolution 1920×1080 ? Could we apply the convolution to an image with resolution 1920×1080 and 8 channels?

Answer If we apply the convolution to an image with 3 channels and resolution 1024×768 , using the formula that we worked out previously (Question 2), we obtain:

$$\text{output_height} = \frac{[1024 - 5 + 2 * 1]}{2} + 1 = 511 \qquad \text{output_width} = \frac{[768 - 5 + 2 * 1]}{2} + 1 = 383$$

We end up with an output tensor with dimensions 16x511x383.

If we apply the same convolution to an image with 3 channels and resolution 1920x1080, using the formula:

$$\text{output_height} = \frac{[1920 - 5 + 2 * 1]}{2} + 1 = 959 \quad \text{output_width} = \frac{[1080 - 5 + 2 * 1]}{2} + 1 = 539$$

We obtain an output tensor with dimensions 16x959x539.

We could apply the convolution to an image with resolution 1920x1080 and 8 channels because the output tensor would have the same number of channels as the number of output channels specified in the convolution (thus, we would still get an output tensor with dimensions 16x959x539), regardless of the number of channels in the input image. It would just make the patches thicker.

Question 11

Let x be an input tensor with dimensions (b, c, h, w) . Write the single line of PyTorch code that implements a global max pool and a global mean pool (one line for each of the two poolings). The result should be a tensor with dimensions (b, c) .

Answer There are at least four ways to do each of the implementations that we know of. For global max pool, these are `torch.nn.MaxPool2D`, `torch.nn.adaptiveMaxPool2D`, `torch.nn.functional.max_pool2d` and `torch.nn.functional.adaptive_max_pool2d`, with the `torch.nn` and the `torch.nn.functional` versions being very similar to their counterpart.

The difference between the adaptive and the non-adaptive methods is that in the latter you specify the pool dimensions while in the former you specify the desired output and the method "adapts" the pool dimensions accordingly. The second option is preferred, since it does not depend on the input. We will show the single-line implementation using `torch.nn.Adaptive` and `torch.nn.functional.adaptive_max_pool2d` as examples for global max pooling, provided that we have imported `nn` and `torch.nn.functional` as `F` from `torch`:

- `output = torch.flatten(nn.AdaptiveMaxPool2d(1), 1)`
- `output = torch.flatten(F.adaptive_max_pool2d(input, (1, 1)), 1)`

In both cases, the pool size would be that of the input height and width, obtaining two singleton dimensions that are squeezed to obtain the desired output.

The implementation of global average pooling using `torch.nn.functional` as `F`, for example, consist on writing "avg" instead of "max":

- `output = torch.flatten(F.adaptive_avg_pool2d(input, (1, 1)), 1)`

Question 12

Use an `ImageFolder` dataset to load the data, and pass it through the network we used earlier. Use a `Resize` transform before the `ToTensor` transform to convert the data to a uniform resolution of 28x28 and pass it through the network of the previous section. See what kind of performance you can achieve.

Answer We load the data from an `ImageFolder` dataset and we use the `transforms.Resize(28,28)` transform before the `ToTensor` transform, as shown in Figure 9. Passing the data through the network of the previous section, we obtain an accuracy of 96.6% after 5 epochs. We can see the performance in Figure 10

```
transform = transforms.Compose([transforms.Resize((28, 28)), transforms.ToTensor(), transforms.Normalize((0.5, ), (0.5, ))])
|
train_data = datasets.ImageFolder('mnist-varres/train', transform=transform)
test_data = datasets.ImageFolder('mnist-varres/test', transform=transform)

# Split the training data into 50 000 training instances and 10 000 validation instances
traindata, valdata = train_test_split(train_data, test_size=10000, random_state=42)
batch_size = 16
trainloader = torch.utils.data.DataLoader(traindata, batch_size=batch_size, shuffle=True, num_workers=2)
valloader = torch.utils.data.DataLoader(valdata, batch_size=batch_size, shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True, num_workers=2)
```

Figure 9: Code for data load and resize


```

Epoch 1
-----
Test Error:
Accuracy: 92.7%, Average loss: 0.016008
Epoch 2
-----
Test Error:
Accuracy: 94.7%, Average loss: 0.010846
Epoch 3
-----
Test Error:
Accuracy: 95.2%, Average loss: 0.009639
Epoch 4
-----
Test Error:
Accuracy: 95.9%, Average loss: 0.008704
Epoch 5
-----
Test Error:
Accuracy: 96.6%, Average loss: 0.007389
Val loss: [0.01600767950466834, 0.01084595177595038, 0.009638688578593428, 0.00870442762532366, 0.007389417464035796]

```

Figure 10: Accuracy of the fixed resolution network

Question 13

We could, of course, resize everything to 64x64. Apart from the fact that running the network would be more expensive, what other downsides do you see?

Answer Apart from the fact that running the network would be more expensive, if we resize everything to a uniform resolution of 64x64, the larger images would require more memory to be handled by the network, increasing the size of the model and making it more difficult to store and transport.

Furthermore, resizing the images to a larger resolution size may introduce unwanted artifacts or distortion in the images, which could negatively impact the performance of the network. In fact, in some cases it can actually decrease performance if the model is not able to effectively process the additional information provided by the higher resolution images.

Question 14

Load the data into a memory as three tensors: one for each resolution. Then write a training loop with an inner loop over the three resolutions. In a more realistic scenario, we could have a dataset where every almost every image has a unique resolution. How would you deal with this situation?

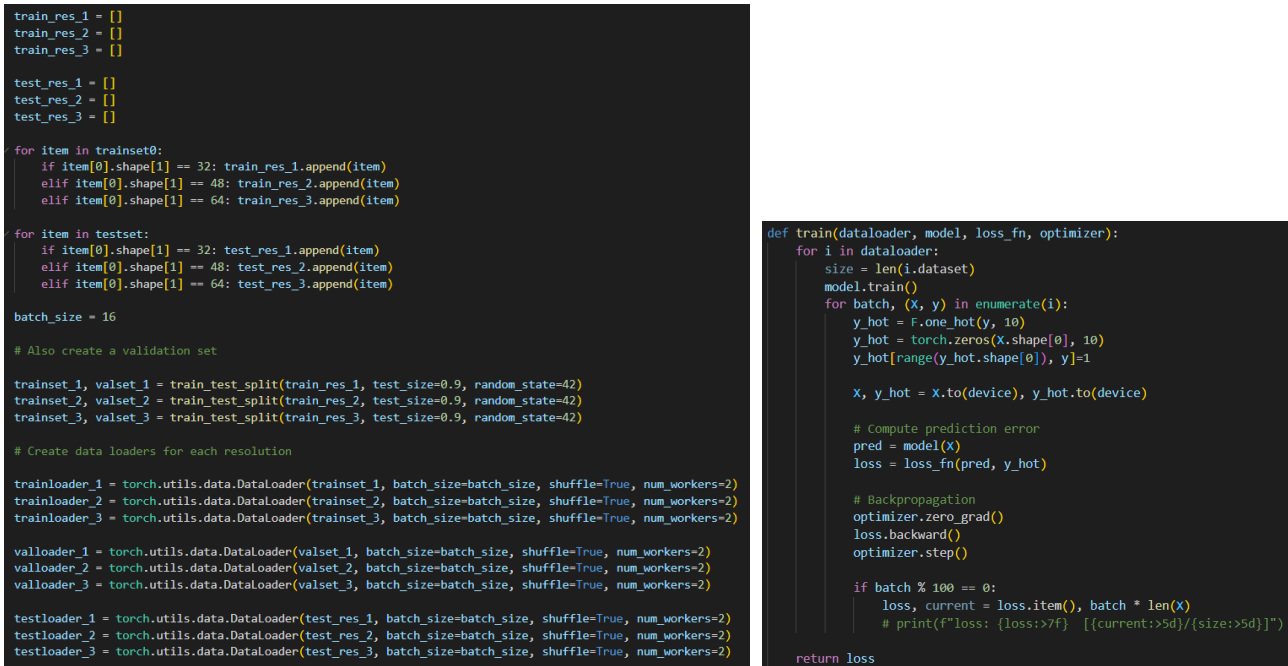
Answer Figure 11 shows how we loaded the data into a memory as three tensors and how we iterate in the training loop. In order to write an inner loop we take into consideration the following lists:

- `train_res = [trainloader_1, trainloader_2, trainloader_3]`, which contains the train data for the three resolutions
- `val_res = [valloader_1, valloader_2, valloader_3]`, which contains the validation data for the three resolutions
- `test_res = [testloader_1, testloader_2, testloader_3]`, which contains the test data for the three resolutions

To deal with a situation where almost every image has a unique resolution it would not be practical to load all the data into memory as separate tensors for each resolution. Instead, it would be more efficient to load the data into a single tensor, where the images are arranged in some order (e.g. by increasing resolution), and use this tensor in a training loop.

Question 15

Note that if we set $N=64$, as we did for the fixed resolution network the last linear layer has fewer parameters here than it did in the first one. Either by trial and error, or through computing the parameters, find the value of N for which both networks have roughly the same number of parameters (this will allow us to fairly compare their performances).



```

train_res_1 = []
train_res_2 = []
train_res_3 = []

test_res_1 = []
test_res_2 = []
test_res_3 = []

for item in trainset0:
    if item[0].shape[1] == 32: train_res_1.append(item)
    elif item[0].shape[1] == 48: train_res_2.append(item)
    elif item[0].shape[1] == 64: train_res_3.append(item)

for item in testset:
    if item[0].shape[1] == 32: test_res_1.append(item)
    elif item[0].shape[1] == 48: test_res_2.append(item)
    elif item[0].shape[1] == 64: test_res_3.append(item)

batch_size = 16

# Also create a validation set

trainset_1, valset_1 = train_test_split(train_res_1, test_size=0.9, random_state=42)
trainset_2, valset_2 = train_test_split(train_res_2, test_size=0.9, random_state=42)
trainset_3, valset_3 = train_test_split(train_res_3, test_size=0.9, random_state=42)

# Create data loaders for each resolution

trainloader_1 = torch.utils.data.DataLoader(trainset_1, batch_size=batch_size, shuffle=True, num_workers=2)
trainloader_2 = torch.utils.data.DataLoader(trainset_2, batch_size=batch_size, shuffle=True, num_workers=2)
trainloader_3 = torch.utils.data.DataLoader(trainset_3, batch_size=batch_size, shuffle=True, num_workers=2)

valloader_1 = torch.utils.data.DataLoader(valset_1, batch_size=batch_size, shuffle=True, num_workers=2)
valloader_2 = torch.utils.data.DataLoader(valset_2, batch_size=batch_size, shuffle=True, num_workers=2)
valloader_3 = torch.utils.data.DataLoader(valset_3, batch_size=batch_size, shuffle=True, num_workers=2)

testloader_1 = torch.utils.data.DataLoader(test_res_1, batch_size=batch_size, shuffle=True, num_workers=2)
testloader_2 = torch.utils.data.DataLoader(test_res_2, batch_size=batch_size, shuffle=True, num_workers=2)
testloader_3 = torch.utils.data.DataLoader(test_res_3, batch_size=batch_size, shuffle=True, num_workers=2)

def train(dataloader, model, loss_fn, optimizer):
    for i in dataloader:
        size = len(i.dataset)
        model.train()
        for batch, (X, y) in enumerate(i):
            y_hot = F.one_hot(y, 10)
            y_hot = torch.zeros(X.shape[0], 10)
            y_hot[range(y_hot.shape[0]), y]=1

            X, y_hot = X.to(device), y_hot.to(device)

            # Compute prediction error
            pred = model(X)
            loss = loss_fn(pred, y_hot)

            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

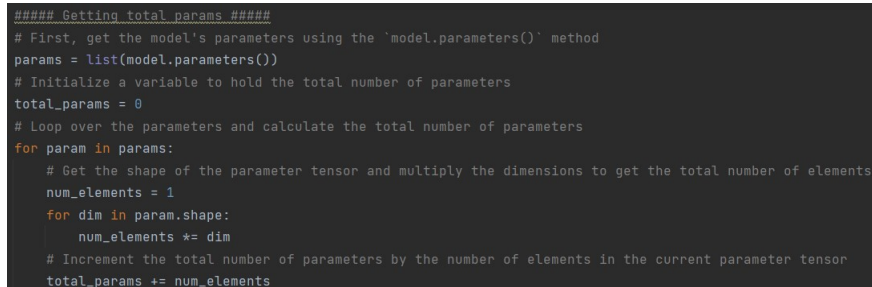
            if batch % 100 == 0:
                loss, current = loss.item(), batch * len(X)
                # print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

    return loss

```

Figure 11: 3-Tensor data loading (left) and training loop (right) of our variable resolution network

Answer We calculated the number of parameters of both networks using the loop showed in Figure 12. For the fixed resolution network, with $N=64$, we obtained a total of 29.066 parameters. After some trial and error, we found that the value of N that makes variable resolution network have roughly the same number of parameters as the fixed resolution network is 81, with 29.029 parameters.



```

##### Getting total params #####
# First, get the model's parameters using the 'model.parameters()' method
params = list(model.parameters())
# Initialize a variable to hold the total number of parameters
total_params = 0
# Loop over the parameters and calculate the total number of parameters
for param in params:
    # Get the shape of the parameter tensor and multiply the dimensions to get the total number of elements
    num_elements = 1
    for dim in param.shape:
        num_elements *= dim
    # Increment the total number of parameters by the number of elements in the current parameter tensor
    total_params += num_elements

```

Figure 12: Loop to obtain the number of parameters of a network

Question 16

Compare the validation performance of global max pooling to that of global mean pooling. Report your findings, and choose a global pooling variant.

Answer Figure 13 shows the comparison for the training of the variable resolution network with global max pooling against global mean pooling. The validation performance of the version with global max pooling shows considerably better results than the version of global mean pooling in the range of 5 epochs. With global max pooling, the network reaches an accuracy of 96.4% over 5 epochs, while the best performance for the validation with global mean pooling is an accuracy of 74.1%.

We ran the training few more times and obtained similar outcomes: the addition of global max pooling was significantly better (i.e. less harmful) than that of the global mean pooling, and the variability was not big enough to justify running an inferential statistic, since the difference in performance is evident. Thus, we chose to incorporate global max pooling for the last question.

<p>Epoch 1</p> <p>-----</p> <p>Test Error: Accuracy: 87.8%, Avg loss: 0.011322</p> <p>Epoch 2</p> <p>-----</p> <p>Test Error: Accuracy: 93.9%, Avg loss: 0.005461</p> <p>Epoch 3</p> <p>-----</p> <p>Test Error: Accuracy: 95.6%, Avg loss: 0.003958</p> <p>Epoch 4</p> <p>-----</p> <p>Test Error: Accuracy: 96.4%, Avg loss: 0.003251</p> <p>Epoch 5</p> <p>-----</p> <p>Test Error: Accuracy: 96.4%, Avg loss: 0.003346</p>	<p>Epoch 1</p> <p>-----</p> <p>Test Error: Accuracy: 23.2%, Avg loss: 0.034074</p> <p>Epoch 2</p> <p>-----</p> <p>Test Error: Accuracy: 26.7%, Avg loss: 0.031450</p> <p>Epoch 3</p> <p>-----</p> <p>Test Error: Accuracy: 47.3%, Avg loss: 0.028926</p> <p>Epoch 4</p> <p>-----</p> <p>Test Error: Accuracy: 56.8%, Avg loss: 0.024344</p> <p>Epoch 5</p> <p>-----</p> <p>Test Error: Accuracy: 74.1%, Avg loss: 0.020220</p>
--	--

Figure 13: Performance of global max pooling (left) vs global mean pooling (right) for the variable resolution network

Question 17

Tune the variable resolution network and the fixed resolution network from question 12 and then compare the test set performance of both. Report your findings.

Answer Figure 14 shows the loss curves obtained during the tuning of the variable resolution network. As for the learning rates, the best performing was 0.003, which came out slightly ahead of 0.001. A lower learning rate was not fast enough to get to good results for the margin of 5 epochs set. We then fixed the learning rate to 0.003 and tested different batch sizes to see if we could further improve the performance of the network. The result was that with a batch size of 16, the network performed better than with 32 and 8. We expected the performance to improve as we reduced the batch size, since the gradients would be computed more frequently (and thus the training was slower), but 8 performed the worst among all batch sizes tested. The chosen hyperparameter set for the final training was *learning rate* = 0.003, *batch size* = 16.

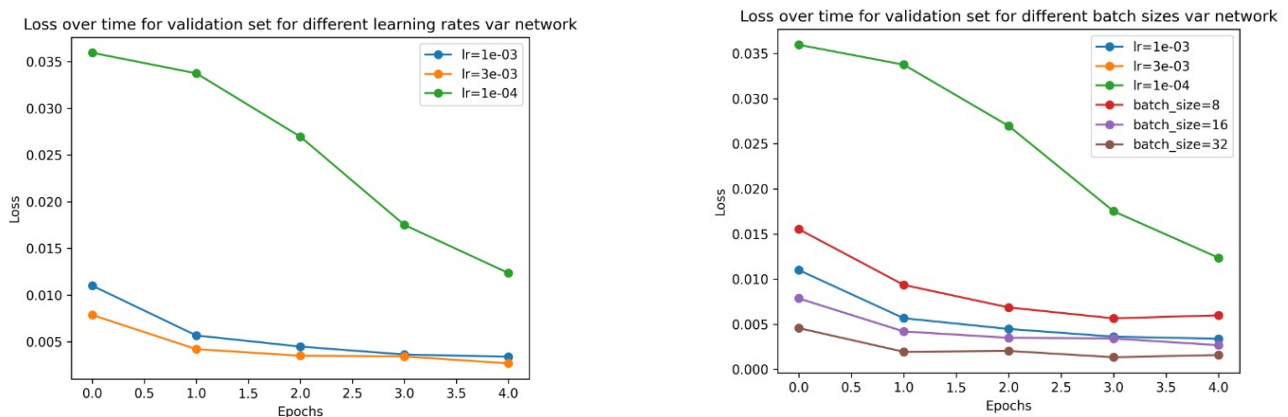


Figure 14: Loss curves for different learning rates and batch sizes for the variable resolution network

Figure 15 shows the loss curves obtained during the tuning of the fixed resolution network. As for the learning rates, the best performing this time was 0.001, which came out slightly ahead of 0.003. Again, a lower learning rate is not ideal for only 5 epochs. We fixed the learning rate to 0.001 and tested different batch sizes to see if we could further improve the performance of the network. This time, the results were in favour of

the most generous batch size, 32. Nevertheless, the chosen hyperparameter set for the final training was *learning rate* = 0.001, *batch size* = 16 since the accuracy obtained was slightly superior.

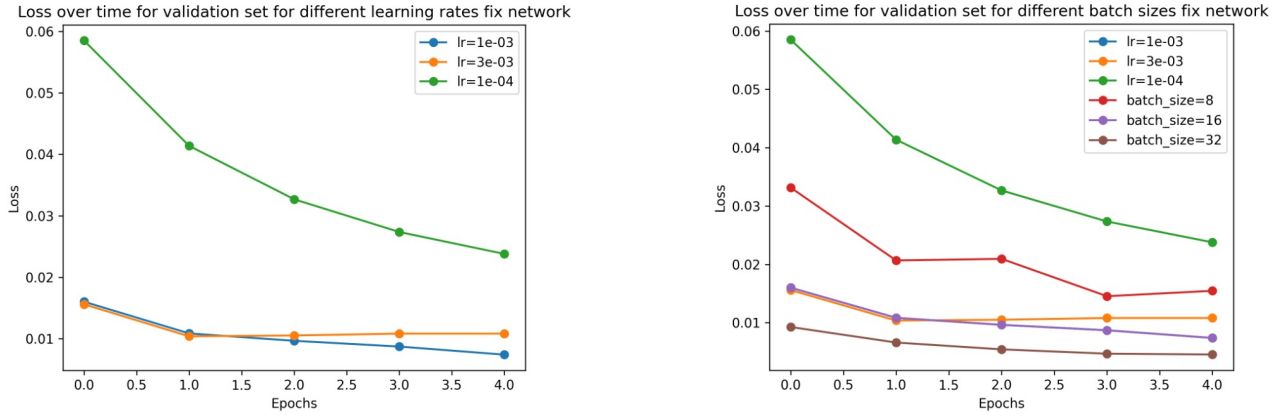


Figure 15: Loss curves for different learning rates and batch sizes for the fixed resolution network

Figure 16 shows the final results for the tuned version of the networks. The test set accuracy was 97.69% for the variable resolution network and 96.50% for the fixed resolution network. In the end, the accuracy obtained came out close, but the fixed resolution network managed to perform slightly better for each of the hyperparameters tested. We did not implement additional "smoothing" the resolutions by adding some extra 2x2 pooling to the variable resolution network, but we think this could have helped the performance. Nevertheless, the tests required some time and we decided to end our discussion here.

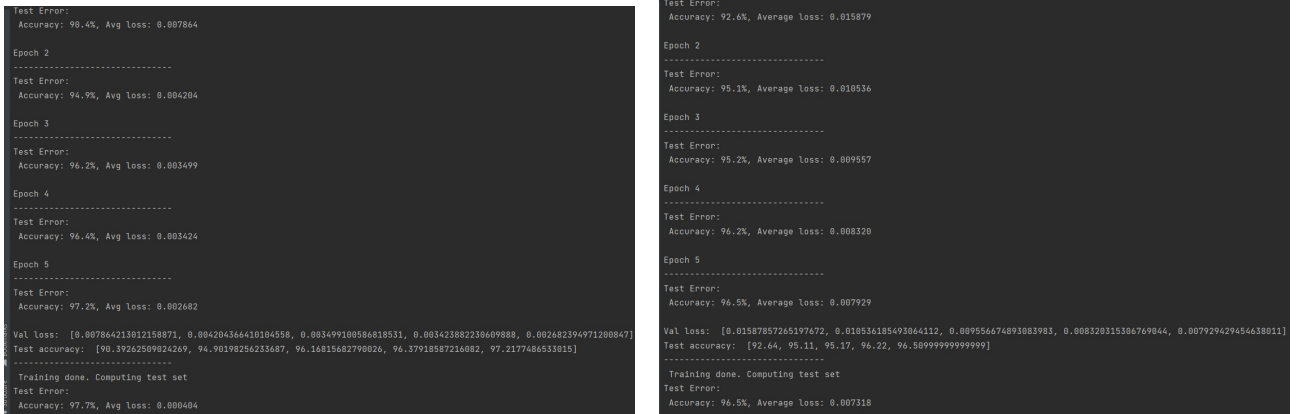


Figure 16: Final run for the variable resolution (left) and the fixed resolution (right) networks