Course: Deep Learning *Coordinator:* Jakub Tomczak

Submission Assignment 3A

Name: [Priyakshi Goswami, Tanya Kaintura, Payanshi Jain], *Student ID:* [pgo207, tka204, pja204]

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 Implementing a convolution through a set of nested loops. Suppose,

- Batch size = B
- input channels = C
- input width = W
- input height = H
- output channels = C'
- output width = W'
- output height = H'
- kernel width = w
- kernel height = h

Now, the input tensor $\mathbf{X}$ is a tensor of dimensions (B, C, W, H). And the output tensor will have (B, C', W', H'). Then the weights/parameters can be represented by a tensor $\mathbf{F}$ with dimensions (C, C', w, h) where (w, h) is the kernel-size, each kernel has C channels and there are a total of C' number of kernels. Now we can write the pseudocode for convolution as:

Algorithm 1 Pseudo-code for convolution:

Input Tensor $\mathbf{X}$, Weights $\mathbf{F}$
Initialize output tensor $\mathbf{O}(B, C', W', H')$
for $b = 1$ to B (% loop over a batch) **do**
  for $j = 1$ to C' (% loop over output channels) **do**
    for $i = 1$ to C (% loop over input channels) **do**
      for $m = 1$ to H' (% loop over rows of output tensor) **do**
        for $n = 1$ to W' (% loop over columns of output tensor) **do**
          for $p = 1$ to h (% loop over rows of kernel) **do**
            for $q = 1$ to w (% loop over columns of kernel) **do**
              $\mathbf{O}[b, j, n, m] = \mathbf{O}[b, j, n, m] + \mathbf{X}[b, i, m + p - 1, n + q - 1] \times \mathbf{F}[i, j, p, q]$
            end for
          end for
        end for
      end for
    end for
  end for
end for

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 If we have the input tensor, kernel size, stride and padding, the output tensor size can be easily calculated.

```
output_height = (input_height - filter_height + 2 x padding)/stride + 1
output_width = (input_width - filter_width + 2 x padding)/stride + 1
output_channels = number of kernels
```

Writing this into a single function that takes input tensor, filter, stride and padding as input and returns the output tensor size, we check our results by comparing it with *conv2d* function of PyTorch.

In [3]:
```
inputs = torch.randn(16, 5, 5, 5)#(batch size, channels, width, heights)
filters = torch.randn(10, 5, 3, 3)
padding = 0
stride = 1
```

Output using Conv2d (torch)

In [4]:
```
output = F.conv2d(inputs, filters, padding=padding, stride=stride)
```

In [5]:
```
output.shape
```

Out[5]:
```
torch.Size([16, 10, 3, 3])
```

Output using the Function

In [6]:
```
def output_size(p,s,inputs,filters):
    output_width = (inputs.shape[2] - filters.shape[2] + (2*p))/s + 1
    output_heights = (inputs.shape[3] - filters.shape[3] + (2*p))/s + 1
    output_dimension = [inputs.shape[0],filters.shape[0], int(output_width), int(output_heights)]
    return output_dimension
```

In [7]:
```
output_dimension = output_size(padding, stride, inputs, filters)
output_dimension
```

Out[7]:
```
[16, 10, 3, 3]
```

Question 3

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

Answer The *Unfold* function transforms the Input of size (b, c, h, w) into a matrix of size (b, p, k) by extracting all patches and flattening them. Here, k is the number of values per patch and p is the number of patches. If the kernel is of size (H, W), then we have

$$k = H \times W \times c$$

The number of patches (p) can be written as a product of number of patches along the height of the input tensor and number of patches along the weight of the input tensor.

```
patches_h = (h - H + 2 x padding)/stride + 1
patches_w = (w - W + 2 x padding)/stride + 1
p = patches_h x patches_w
```

Algorithm 2 Pseudo-code for *Unfold*

```
Input X
Initialize a matrix M(b, p, k)
for all b images in a batch do
  for all patches along the width of the input do
    for all patches along the height of the input do
      Extract current patch of size (c, H, W)
      Flatten it into a vector v of length k
      Insert v into the matrix M in the next empty row
    end for
  end for
end for
```

This is same as the size of the output tensor height and weight as we saw in Question 2.

We will implement convolution as a PyTorch Module and a PyTorch Function.For the module, all we need is a python object that inherits from nn.Module, and implements a method called *forward*.

The forward pass of the convolution consists of the following steps:

```
1. Unfold X(b, c, h, w) to U(b, k, p)
2. Reshape U(b, k, p) to U(b*p, k)
3. Matrix multiplication of U(b*p, k) with W(c', k) to get Y'(b*p, c')
4. Reshape Y'(b*p, c') to Y'(b, c', p)
5. Fold Y'(b, c', p) into Y(b, c', h_out, w_out)
```

In [2]:
```python
class Conv2D(nn.Module):

    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super(Conv2D, self).__init__()

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        self.conv = nn.Parameter(torch.ones(out_channels, in_channels, kernel_size, kernel_size))

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)

    def forward(self, input_batch):
        b, c, h, w = input_batch.size()
        k = self.kernel_size
        p = self.padding
        s = self.stride

        h_out = (h + 2*p - k)/s + 1
        w_out = (w + 2*p - k)/s + 1
        h_out, w_out = int(h_out), int(w_out)

        #Unfold
        x = torch.nn.functional.unfold(input_batch, (k, k), padding=p)
        x = x.transpose(1, 2)

        P = x.shape[1]

        # Reshape to (b*p, k)
        x = torch.reshape(x, (x.shape[0] * x.shape[1], x.shape[2]))

        #Matmul
        W = torch.reshape(self.conv , (self.conv.size(0), -1))
        y = x.matmul(W.t())

        # Reshape to (b, l, p)
        y = torch.reshape(y, (input_batch.shape[0], P, y.shape[1]))

        y = y.transpose(1, 2)

        out = torch.nn.functional.fold(y, (h_out, w_out), (1, 1))
        print(out.shape)

        r = self.conv1(input_batch)
        assert r.shape == out.shape

        return out
```

In [3]:
```python
conv = Conv2D(in_channels = 3, out_channels = 16)
input_batch = torch.randn(16, 3, 32, 32)
output_batch = conv(input_batch)
```

```
torch.Size([16, 16, 32, 32])
```

Question 4

Work out the backward with respect to the kernel weights $\mathbf{W}$.

Answer To find the backward with respect to kernel weights(W), we first need to find Y'^{∇} as a function of Y^{∇} and then W^{∇} as a function of Y'^{∇}.

- **Backward of the reshape function** $Y'(b, c', p) \longrightarrow Y(b, c', h', w')$ Now suppose we are doing a reshape function over a matrix A into B.
  $A = [a_1, a_2, a_3, a_4, a_5, a_6]$ to $B = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \\ a_5 & a_6 \end{bmatrix}$ Now, if we try to compute the backward $B^{\nabla} \longrightarrow A^{\nabla}$, we get

  $$A^{\nabla}_{ij} = \sum_{k,l} B^{\nabla}_{kl} . \frac{\partial B_{kl}}{\partial A_{ij}}$$

  We know that the reshape function does not change any of the values in a tensor but just the positions which will give us $A^{\nabla}_{11} = B^{\nabla}_{11}$, $A^{\nabla}_{12} = B^{\nabla}_{12}$, $A^{\nabla}_{13} = B^{\nabla}_{21}$, ...$A^{\nabla}_{16} = B^{\nabla}_{33}$ i.e. $A^{\nabla} = [B^{\nabla}_{11}, B^{\nabla}_{12}, B^{\nabla}_{21}, B^{\nabla}_{22}, ..B^{\nabla}_{32}]$ Therefore, we can write

  $$A^{\nabla} = B^{\nabla}.reshape(shape(A))$$

  Now, extending this to the reshape of $Y \longrightarrow Y'$, we get

  $$\mathbf{Y'^{\nabla} = Y^{\nabla}.reshape(shape(Y'))}$$

- **Backward of the matmul** First, we know that if we have two simple matrices W - f x m matrix , X - n x f matrix and outputs $Y = XW$ is n x m matrix . Here $Y = XW$ is matrix multiplication which is defined by

  $y_{ab} = \sum_c x_{ac} w_{cb}$
  $w^{\nabla}_{ij} = \sum_{a,b} y_{ab} \frac{\partial y_{ab}}{\partial w_{ij}} = \sum_{a,b} y^{\nabla}_{ab} \frac{\partial \sum_c x_{ac} w_{cb}}{\partial w_{ij}} = \sum_{a,b} y^{\nabla}_{ab} \frac{\partial x_{ai} w_{ib}}{\partial w_{ij}} = \sum_a y^{\nabla}_{aj} \frac{\partial x_{ai} w_{ij}}{\partial w_{ij}} = \sum_a y^{\nabla}_{aj} x_{ai}$
  $W^{\nabla} = X^T . Y^{\nabla}$
  $x^{\nabla}_{ij} = \sum_{a,b} y^{\nabla}_{ab} \frac{\partial y_{ab}}{\partial x_{ij}} = \sum_{a,b} y^{\nabla}_{ab} \frac{\partial \sum_c x_{ac} w_{cb}}{\partial x_{ij}} = \sum_{a,b} y^{\nabla}_{ab} \frac{\partial x_{aj} w_{jb}}{\partial x_{ij}} = \sum_b y^{\nabla}_{ib} \frac{\partial x_{ij} w_{jb}}{\partial x_{ij}} = \sum_b y^{\nabla}_{ib} w_{jb}$
  $X^{\nabla} = Y^{\nabla} . W^T$

  Here, our weights/kernel W is a matrix of size (k, c') where k is the number of nodes per patch and c' is the number of output channels. The U we get from the unfold function is of shape (b, p, k). We can easily reshape this into $U(b*p, k)$. Now our matrix multiplication becomes $T = UW$ where T is of shape$(b*p, c')$. And finally $Y' = T.reshape(b, c', p)$. So for the gradients, we can use all the results we have derived above:

  $$T^{\nabla} = Y'^{\nabla}.reshape(b * p, c')$$
  $$W^{\nabla} = (U.reshape(b * p, k))^T . T^{\nabla}$$
  $$\mathbf{W^{\nabla} = (U.reshape(b * p, k))^T . (Y'^{\nabla}.reshape(b * p, c'))}$$

Question 5

If the input to the convolution is not the input to the network, but an intermediate value, we'll need gradients over the input as well. Work out the backward with respect to the input $\mathbf{X}$.

Answer We can also get the backward of U in the same way using Chain rule-

$$\mathbf{U^{\nabla} = ((Y'^{\nabla}.reshape(b * p, c')).W^T).reshape(b, p, k)}$$

Using Chain Rule, Backward of X :

$\frac{\partial l}{\partial X} = \frac{\partial l}{\partial Y} * \frac{\partial Y}{\partial Y'} * \frac{\partial Y'}{\partial U} * \frac{\partial U}{\partial X}$

$X^{\nabla} = Y^{\nabla} * Y'^{\nabla} * W * \frac{\partial U}{\partial X}$....1

For $\frac{\partial U}{\partial X}$:
Let X(b, c, h, w) , kernel size(kh, kw), stride(sh, sw), and Y'(b, c, hout*wout) , Y(b, c, hout,wout), where
$hout = (h + 2 * padding - kh)//sh + 1$
$wout = (w + 2 * padding - kw)//sw + 1$

If input to unfold function is $X(b, c, hw)$, output will be $X(b, c, kh * kw, hout * wout)$ That is :
$Unfold(X(b, c, h, w)) = U(b, c * kh * kw, hout * wout)$
then backward of unfold will be fold depended on intput size(h, w) :
$\mathbf{Fold}(\mathbf{U}^{\nabla}(\mathbf{b}, \mathbf{c} * \mathbf{kh} * \mathbf{kw}, \mathbf{hout} * \mathbf{wout})) = \mathbf{X}^{\nabla}(\mathbf{b}, \mathbf{c}, \mathbf{h}, \mathbf{w})$

In equation 1: $\mathbf{X}^{\nabla} = \mathbf{Y}^{\nabla} * \mathbf{Y'}^{\nabla} * \mathbf{W} * \mathbf{X}$

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

In [4]:
```
class Conv2DFunc(torch.autograd.Function):

    @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)

        h = input_batch.shape[2]
        w = input_batch.shape[3]
        c = input_batch.shape[1]
        k = kernel.shape[2]
        C = kernel.shape[0]


        #output_batch = F.conv2d(input_batch, kernel, stride=stride, padding=padding)
        h_out = (h + 2*padding - k)/stride + 1
        w_out = (w + 2*padding - k)/stride + 1
        h_out, w_out = int(h_out), int(w_out)


        #Unfold
        x = torch.nn.functional.unfold(input_batch, (k, k), padding=padding)
        x = x.transpose(1, 2)

        P = x.shape[1]

        # Reshape to (b*p, k)
        x = torch.reshape(x, (x.shape[0] * x.shape[1], x.shape[2]))
        ctx.save_for_backward(x)

        #Matmul
        W = torch.reshape(kernel , (kernel.size(0), -1))
        #ctx.save_for_backward(W)
        y = x.matmul(W.t())


        # Reshape to (b, l, p)
        y = torch.reshape(y, (input_batch.shape[0], P, y.shape[1]))
        yshape = y.shape
        ctx.yshape = yshape
        y = y.transpose(1, 2)

        out = torch.nn.functional.fold(y, (h_out, w_out), (1, 1))
        print(out.shape)

        r = F.conv2d(input_batch, kernel, stride=stride, padding=padding)
        assert r.shape == out.shape

        return out


    @staticmethod
    def backward(ctx, grad_output):

        # retrieve stored objects
        input, kernel, U = ctx.saved_tensors


        # backward of Kernel
        yshape = ctx.yshape

        y_grad = torch.reshape(grad_output, yshape)
        y_grad = torch.reshape(yshape, (yhape[0] * yshape[1], yshape[2]))

        kernel_grad = U.t().matmul(y_grad)

        input_batch_grad = None

        return input_batch_grad, kernel_grad, None, None
```

In [5]:
```
input_batch = torch.randn(16, 3, 32, 32,  requires_grad=True, dtype=torch.double)
kernel = torch.randn(10, 3, 3, 3,  requires_grad=True, dtype=torch.double)
out = Conv2DFunc.apply(input_batch, kernel)
```

```
torch.Size([16, 10, 32, 32])
```

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.

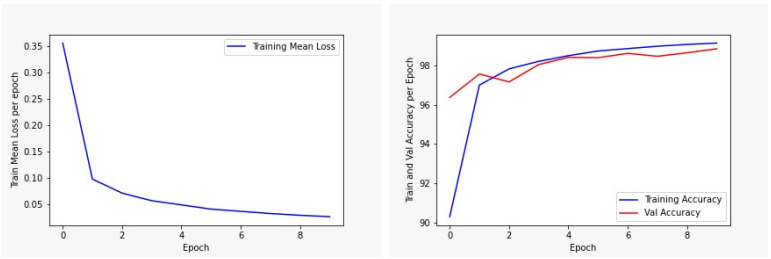
Answer MNIST training data has a total of 60,000 training instances. We split this data into train (50,000) and validation data (10,000) before loading them into the dataloaders for training.

```
# load the dataset
train_dataset = datasets.MNIST(root=data_dir, train=True,
            download=True, transform=train_transform)

valid_dataset = datasets.MNIST(root=data_dir, train=True,
            download=True, transform=valid_transform)

num_train = len(train_dataset)
indices = list(range(num_train))
split = int(np.floor(valid_size * num_train))

if shuffle == True:
    np.random.seed(random_seed)
    np.random.shuffle(indices)

train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(train_dataset,
                batch_size=batch_size, sampler=train_sampler,
                num_workers=num_workers, pin_memory=pin_memory)

valid_loader = torch.utils.data.DataLoader(valid_dataset,
                batch_size=batch_size, sampler=valid_sampler,
                num_workers=num_workers, pin_memory=pin_memory)
```

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 We build the network and used dataloader from pytorch to load the data into train, validation and test. The model was trained for 10 epochs, 16 batch size, 0.0001 learning rate and used cross entropy loss function and Adam optimiser. Loss is decreasing over epochs and accuracy is increasing for train and validation approximately to 98%. Evaluating the model on the test set, we get a test accuracy of **98.68**%.



(a) Training Mean Loss (b) Accuracy

Figure 1: Answer 8: Baseline Performance

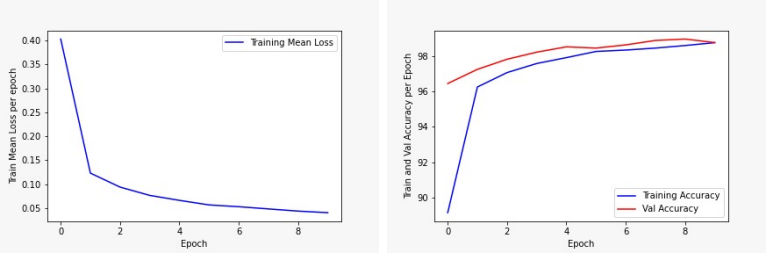
```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        #
        self.conv1 = nn.Sequential(nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1),
                                   nn.ReLU(),
                                   nn.MaxPool2d(kernel_size=2),
                                  )
        self.conv2 = nn.Sequential(nn.Conv2d(16, 32, 3, 1, 1),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=2),
                                  )
        self.conv3 = nn.Sequential(nn.Conv2d(32, 64, 3, 1, 1),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=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 = torch.flatten(x, 1)
        x = self.out(x)

        return x
```

Figure 2: Answer 8: Fixed Resolution CNN Network

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 tried various data augmentations for our model - Horizontal flip, Random crop, Random rotation etc. Most of them did not improve the performance of our model except Random Rotation which showed a slight increase in our test accuracy. We implemented Random Rotation data augmentation to the data loader in the training set. The model was trained for 10 epochs, 16 batch size, 0.0001 learning rate and used cross entropy loss function and Adam optimiser. Loss is decreasing over epochs and accuracy is increasing for train and validation approximately to 98-99%.
With augmentation, although the training accuracy was almost same as baseline, the model became a little more accurate, with test and validation accuracy increasing by almost 1%. As shown in table 1, with augmentation the accuracy is 99.07% and without is 98.93%. Please refer to question 8 above to check the plots for baseline.



(a) Training Mean Loss (b) Accuracy

Figure 3: Answer 9: With Augmentation Performance

	Without Augmentation	With Augmentation
Test set	98.93%	99.07%

Table 1: Answer9 : Test set results

Data augmentation is done typically only on train data because the purpose of it is to increase the size of

the data-set and get different more images so that the model learns more robustly and generalize well to avoid over-fitting. Since the test data or validation data are not used in this generalization but just to measure the model performance, their augmentation is unnecessary.

Question 10

Assume we have a convolution with a 5x5 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 1024x768. What are the dimensions of the output tensor? What if we apply the same convolution to an image with 3 channels and resolution 1920x1080? Could we apply the convolution to an image with resolution 1920x1080 and 8 channels?

Answer To determine the dimensions of the output tensor after applying the given convolution to an image with 3 channels and resolution 1024x768, you need to calculate the dimensions of the output tensor using the following formula:

```
output_height = (input_height + 2 * padding - kernel_size) / stride + 1
output_width = (input_width + 2 * padding - kernel_size) / stride + 1
output_channels = output_channels
```

If the input tensor has dimensions (3, 1024, 768). Plugging these values into the formula above, we get:

```
output_height = (1024 + 2 * 1 - 5) / 2 + 1 = 513
output_width = (768 + 2 * 1 - 5) / 2 + 1 = 385
output_channels = 16
```

Therefore, the dimensions of the output tensor after applying the convolution will be (16, 513, 385).

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

```
output_height = (1920 + 2 * 1 - 5) / 2 + 1 = 961
output_width = (1080 + 2 * 1 - 5) / 2 + 1 = 641
output_channels = 16
```

The dimensions of the output tensor will be (16, 961, 641).
If we apply the same convolution to an image with 8 channels and resolution 1920x1080, the output tensor will have dimensions (16, 961, 641), but the convolution will not be applied correctly, as the number of input channels (8) does not match the number of input channels expected by the convolution (3). In order to apply the convolution correctly, the number of input and output channels must match the dimensions expected by the convolution.

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 Below (Figure 4) is the PyTorch code for implementation of global max pool and global mean pool.

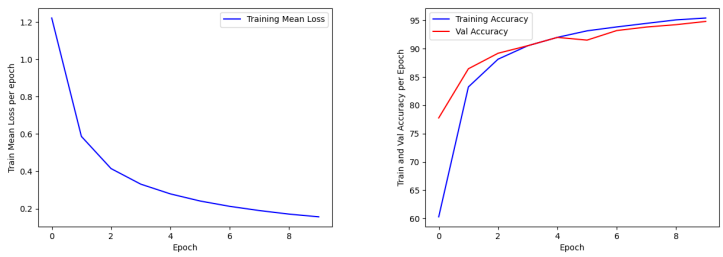
Question 12

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 use the data from ImageFolder and pass through the network after resizing the images to (28,28). The model was trained for 10 epochs, 16 batch size, 0.0001 learning rate and used cross entropy loss function and Adam optimiser. Loss is decreasing over epochs and accuracy is increasing for train and validation approximately to 95%. We achieve a validation accuracy of 94.32% and a test accuracy of 95.04%. The loss curve per epoch along with training and validation accuracy vs epochs is given below in Figure 5.

```
In [60]: x = torch.randn(10, 5, 3, 4)

In [61]: x_mean_pool = torch.mean(x.view(x.size(0), x.size(1), -1), dim=2)
         x_max_pool = torch.max(x.view(x.size(0), x.size(1), -1), dim=2)

In [62]: x_mean_pool.shape

Out[62]: torch.Size([10, 5])

In [63]: x_max_pool[0].shape

Out[63]: torch.Size([10, 5])

In [64]: x_mean_pool

Out[64]: tensor([[ 0.0467,  0.0175,  0.1895, -0.7514,  0.0242],
                 [-0.1610,  0.2377, -0.2689,  0.4620,  0.0795],
                 [-0.2015, -0.0645,  0.2027,  0.2001, -0.3461],
                 [ 0.1308,  0.0481,  0.1180,  0.1716,  0.0136],
                 [ 0.1147, -0.0550, -0.1865, -0.0901,  0.0964],
                 [-0.0595, -0.4492, -0.3744,  0.1736, -0.3727],
                 [ 0.2964, -0.2718, -0.0246,  0.2071, -0.0179],
                 [ 0.0160, -0.4862,  0.2450,  0.2930,  0.0456],
                 [ 0.3892, -0.4105, -0.4212, -0.4591,  0.7573],
                 [-0.2444, -0.1751, -0.4091,  0.2015,  0.2022]])

In [65]: x_max_pool

Out[65]: torch.return_types.max(
         values=tensor([[0.8930, 1.2958, 2.0468, 0.5708, 1.7492],
                 [1.3677, 1.7183, 1.3215, 1.3267, 2.7351],
                 [1.3198, 2.4796, 2.1402, 1.3001, 1.7870],
                 [1.4887, 1.2393, 1.9484, 2.5183, 1.6162],
                 [2.1297, 1.9673, 1.8664, 1.7344, 0.9601],
                 [2.3047, 1.0591, 1.4477, 1.6938, 1.3646],
                 [2.3246, 1.9541, 1.5654, 2.2761, 1.2131],
                 [1.8424, 1.9899, 1.9133, 1.8392, 1.8689],
                 [1.9576, 1.0952, 0.7398, 1.7301, 2.7018],
                 [0.9203, 0.6774, 1.6847, 1.6686, 1.4751]]),
         indices=tensor([[10,  1, 10,  5,  9],
                 [ 7,  0,  8,  2,  0],
                 [ 9, 11, 10,  4,  9],
                 [ 9,  1,  5, 10,  6],
                 [ 1, 11,  3,  4,  9],
                 [11,  7,  0,  9, 10],
                 [ 3,  3,  9,  0,  9],
                 [ 6, 10,  1,  1,  0],
                 [ 4,  6,  4,  4,  4],
                 [ 1,  1,  9,  1,  9]]))
```

Figure 4: Answer 11:Implementation of global meanpool and global maxpool



(a) Training Mean Loss (b) Accuracy

Figure 5: Answer 12: Performance

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 The downsides of resizing all the data to a uniform resolution of 64x64 include potential loss of information or distortion in the images, unnecessary computational overhead, and potentially suboptimal performance of the network.
Resizing all the data to a uniform resolution may not be necessary if the original data already has a consistent resolution, and it may result in unnecessary computational overhead. Also, resizing all the data to a uniform resolution of 64x64 may not be necessary if the network is designed to handle variable input resolutions which may not provide any benefits and may even hinder the network's performance.

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 We have implemented a custom collate functions to return 3 tensors depending upon the resolution which is called in the dataloader which is shown in the Figure 6(a). In Figure 6 (b) The training loop is looped over epochs(line 16) then over batch and within each batch(line 22) the inner loop works over the tensors for different resolution(line 25).

```python
# Create a custom collate function
def collate_fn(data):
    # Initialize empty tensors for each resolution
    resolution1_tensor = []
    resolution2_tensor = []
    resolution3_tensor = []

    # Traverse through each image in the data
    for image in data:
        transformer = transforms.ToPILImage()
        transformer1 = transforms.ToTensor()
        image_tensor = transformer1(transformer(image[0]))
        # Check the resolution of the image
        if (image_tensor.shape[1] == resolution1 & image_tensor.shape[2] == resolution1):
            resolution1_tensor.append(image)
        elif (image_tensor.shape[1] == resolution2 & image_tensor.shape[2] == resolution2):
            resolution2_tensor.append(image)
        elif (image_tensor.shape[1] == resolution3 & image_tensor.shape[2] == resolution3):
            resolution3_tensor.append(image)

    # Return the tensors for each resolution
    return resolution1_tensor, resolution2_tensor, resolution3_tensor

# Define the resolutions for the images
resolution1 = 32
resolution2 = 48
resolution3 = 64
```

(a) Custom Collate Function

```python
def train(num_epochs,cnn,loaders):

    cnn.train()

    # Train the model
    total_step = len(loaders['train'])
    total_step_val = len(loaders['val'])
    train_loss_list = []
    train_mean_loss_list = []
    train_acc_list = []
    val_acc_list = []

    for epoch in range(num_epochs):
        mean_loss = []
        train_correct = 0
        train_total = 0
        val_correct = 0
        val_total = 0
        for step in range(0,total_step):
            tensor1, tensor2, tensor3 = next(iter(loaders['train']))
            tensors = [tensor1, tensor2, tensor3]
            for j, tensor in enumerate(tensors):
                if(len(tensor)==0):
                    continue
                images = [x[0] for x in tensor]
                labels = [x[1] for x in tensor]
                images = torch.stack(images)
                labels = torch.tensor(labels)

                outputs = cnn(images)
                loss = criterion(outputs, labels)
                # clear gradients for this training step
                optimizer.zero_grad()
                # backpropagation, compute gradients
                loss.backward()
                # apply gradients
                optimizer.step()
```

(b) Training loop over tensors

Figure 6: Answer 14: Code snippet to load the data as tensors and write a training loop

If the dataset contains images with a wide range of resolutions, One potential solution would be to use a variable-sized input tensor, where the dimensions of the tensor are determined by the size of the input image. This would allow the model to handle images of different resolutions without the need for a fixed-sized tensor.

Another approach would be to resize all of the images to a fixed size before training the model. This would require resizing the images to a common resolution, which may introduce some distortion or loss of detail.

```python
class CNN(nn.Module):
    def __init__(self, input_channels, num_classes, N):
        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(input_channels, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(2)

        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(2)

        self.conv3 = nn.Conv2d(32, N, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        self.maxpool3 = nn.MaxPool2d(2)

        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.linear = nn.Linear(N, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.maxpool1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.maxpool2(x)

        x = self.conv3(x)
        x = self.relu3(x)
        x = self.maxpool3(x)

        x = self.global_pool(x)
        x = x.view(x.size(0),-1)
        x = self.linear(x)

        return x
```

Figure 7: Answer 14:Variable Resolution CNN Network

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).

Answer Using the code in figure 4, we found when N =81, we have approximately equal number of parameters. The number of parameters are 29066 and 29029 for fixed resolution network and variable resolution network respectively.

```python
# Initialize a counter
total_params = 0

# Loop over the model's parameters
for param in cnn1.parameters():
  # Get the no. of elements in the parameter and sum it
  total_params += param.numel()

# Print the total number of parameters
print(total_params)
```

```
29066
```

(a) Fixed Resolution Network

```python
# Initialize a counter
total_params = 0

# Loop over the model's parameters
for param in cnn.parameters():
   # Get the no. of elements in the parameter and sum it
  total_params += param.numel()

# Print the total number of parameters
print(total_params)
#29066
```

```
29029
```

(b) Variable Resolution Network

Figure 8: Answer 15: Code snippet to calculate the number of parameters

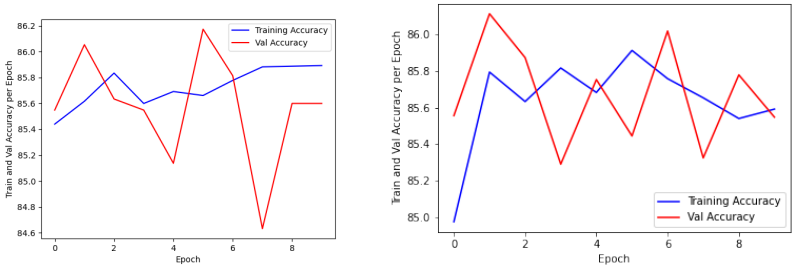
Question 16

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

Answer We used the network given in the Figure 7 and compared the validation performance of the global max pool and global mean pool in the network. We used "nn.AdaptiveAvgPool2d(1)" for global average pooling and "nn.AdaptiveMaxPool2d(1)" for global max pooling.

The model was trained for 10 epochs, 16 batch size, 0.001 learning rate and used cross entropy loss function and SGD optimiser. The network has 3 input channels, 10 classes and N = 81.

Overall, global max pooling and global mean pooling are similar in that they both reduce the spatial dimensions of the feature map, but they differ in how they summarize the information in the feature map. Global max pooling retains the most important information, while global mean pooling retains a summary of the information. Based on the validation accuracy we will choose Global mean pooling for this case.



(a) Global Max Pooling (b) Global Mean Pooling

Figure 9: Answer 16: Plots of train accuracy vs validation accuracy per epoch

	Global Max Pooling	Global Mean Pooling
Validation Performance	86.002%	86.259%

Table 2: Answer 16 : Test set results

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.

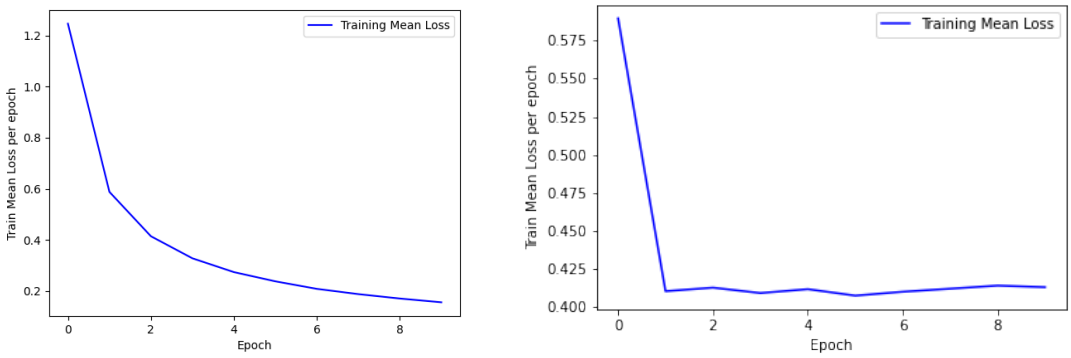
Answer For fixed resolution network as give in Figure 2, The model was trained for 10 epochs, 16 batch size, 0.0001 learning rate and used cross entropy loss function and Adam optimiser.
For variable resolution network as given in Figure 7, The model was trained for 10 epochs, 16 batch size, 0.001 learning rate and used cross entropy loss function and SGD optimiser. The network has 3 input channels, 10 classes and N = 81.
We used N = 81 to make the parameters approximately same for both networks but still see that fixed resolution network was performing better than the variable resolution one based on the test set accuracy.

	Fixed Resolution Network	Variable Resolution Network
Test Set Performance	95.07%	85.87%

Table 3: Answer17 : Test set results

In Figure 10(b), the loss curve steeply decreases and then becomes constant, this could indicate that the model has reached a local minimum. This is generally a good sign, as it means that the model has learned the training data and is not making significant progress in reducing the loss.

However, it is important to note that reaching a local minimum does not guarantee that the model is performing well. Since, the Variable resolution Network is giving test accuracy of 85% which is less than fixed resolution network. In this cases, the model may be stuck at a local minimum that is not optimal, which can lead to poor performance on unseen data.



(a) Fixed Resolution Network (b) Variable Resolution Network

Figure 10: Answer 17: Plots of Loss per epoch

References

Indicate papers/books you used for the assignment. References are unlimited. I suggest to use `bibtex` and add sources to `literature.bib`. An example citation would be Eiben et al. (2003) for the running text or otherwise (Eiben et al., 2003).

References

Eiben, A. E., Smith, J. E., et al. (2003). *Introduction to evolutionary computing*, volume 53. Springer.

Appendix

The TAs may look at what you put here, **but they're not obliged to**. This is a good place for, for instance, extra code snippets, additional plots, hyperparameter details, etc.