Course: Deep Learning **([24-11-2021])**

Submission Assignment #[3A]

Coordinator: Jakub Tomczak *Name:* [Tiddo Loos, Yoes Ywema, Roel Rotteveel], *Netid:* [tls430, yya230 , rrl620]

1 Introduction

In this report the aim is to design a Convolutional Neural Networks that can be applied on the MNIST data set. Convolutional Neural Networks consist of two main layers that are applied on input data: a "Convolutional layer" and a "pooling layer". We will start having a look at the first one by diving into the mechanics of such a layer, and on how to update all parameters in networks containing these layers.

1.1 Convolutional Layer

The Convolutional layer applies a filter on input data, often a tensor, and gives as output a tensor where the filter is applied over the whole input. Padding is a layer attached to the sides of your inputs in order to use the information that is located there equally well. We need to go from the input tensor x with dimensions; $(batch\_size, input\_channels, input\_width, input\_height)$ to the output tensor y with dimensions $(batch\_size, output\_channels, output\_width, output\_height)$. For this we need an amount of padding, a number of output channels, kernel size, and a stride. The below pseudocode shows how kernels are applied onto the input patches.

```
for sample in batch_size:
    padding_width = input_width + 2*padding
    padding_height = input_height + 2*padding
    # Given that output_width and output_height are the same:
    output_width = output_height = ((input_width - kernel_size + 2*padding)/stride) + 1
    # Create empty output for this layer
    output = zeros([total_kernels, output_width, output_height])
    for layer in input_channels:
        layer_pad = [sample, layer, :, :]
        # Add padding
        for pad in padding:
            add column of zeros to right to layer_pad
            add column of zeros to left to layer_pad
            add column of zeros to top to layer_pad
            add column of zeros to bottom to layer_pad
        # Move over layer:
        for y in range(output_height-1):
            for x in range(output_width-1):
                # If multiple kernels exist, save in different slices of output
                for kernel in total_kernels:
                    patch = layer_pad[y*stride:(y+1)*stride, x*stride:(x+1)*stride]
                    output[kernel, y, x] += sum( patch * kernel[layer, :, :] )
    sample_output[sample, :, :, :] = output
```

Question 1.1

The weights are represented in matrices of the form $(input\_channels, kernel\_size, kernel\_size)$. As can be seen in the last for-loop there is a seperate matrix for each layer. It's also possible that there are multiple filters/kernels. In that case multiple outputs are generated, which are saved apart from each other. This can be seen in the initialisation of the output matrix, whose first index is that of $total\_kernels$. In this pseudo-code the number of output channels is set to one, since the output of one layer is added on top of the other layers

(due to the $+=$ after $output[kernel, y, x]$. This could also be changed to for example $input\_channels$ by not adding the output, but rather saving it in different slices like $output[kernel, layer, y, x]$.

As can be seen in the pseudocode at first the output shape is constructed, the dimensions are computed by the following formula if the input height and width are equal:

Question 1.2

$$output\_width = output\_height = ((input\_width - kernel\_size + 2 * padding)/stride) + 1$$

When the input or kernel is not in a square shape than the output width can be obtained by using the input width and kernel width in this formula and for the output height using the input height and kernel height.

1.2 Unfolding

Now we'll have a look at how all different subsets of the data in a batch are obtained to apply weights on. On the batched input a function is applied which is called "unfolding". The idea is simple, unfolding is the process of splitting an input up in patches that are altogether multiplied by a set of weights. For a single picture this means sliding over it with a window of a certain size (kernel size) with a certain amount of steps each time (in both x and y direction). A simple schematic example is shown in figure 1.

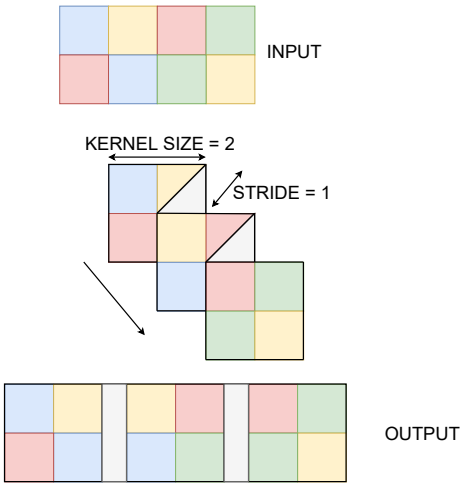

Figure 1: **Unfolding process for a single 2x8 image with a stride of 1 and a kernel size of 2**

To make it a bit more applicable on our problem with 28x28 sized pictures, below the following pseudocode describes how to do this for batches of data (size 16) with multiple channels(3) and with an added padding (1).

```
#input_tensor = [b, c, h, w] #batch_size, channels, height, width
# Pseudo code naive unfold:
def naive_unfold(input_tensor, kernel_size, stride, padding):
    output_size = output_size(input_tensor, kernel_size, stride, padding)
    #1 extract all patches from the input
    for sample in b:
        for channel in c:
            layer_pad = [sample, layer, :, :]
            # Add padding
            for pad in padding:
                add column of zeros to right to layer_pad
                add column of zeros to left to layer_pad
                add column of zeros to top to layer_pad
                add column of zeros to bottom to layer_pad
            n_patches_per_layer = output_size * output_size
            for y in range(output_size-1):
                for x in range(output_size-1):
                    # x+ y = number of patch, total_patch is 0 first time
                    total_patch = (x+y) * channel
```

```
                    patch[sample, x+y + total_patch ,:, :] = \
                    layer_pad[y*stride : (y*stride)+kernel_size, x*stride : (x*stride)+kernel_size]
```

Question 1.3

1.3 Forward pass

The whole forward pass of an image into the neural network can be summarized by the following code (some details left out for illustration purpose):

```
@staticmethod
def forward(ctx, input_batch, kernel, stride=1, padding=1):
    output_channels = kernel.size()[1]
    batch_size, input_channels, height, width = input_batch.size()
    kernel_size = int(math.sqrt(kernel.size()[0]/input_channels))
    h_out = int(output_size(input_batch, kernel_size, stride, padding))
    w_out = int(output_size(input_batch, kernel_size, stride, padding))
    U = F.unfold(input_batch, (kernel_size, kernel_size), padding=padding, stride=stride)
    batch_size, k_values_per_patch, patches = U.size()
    U_reshaped = torch.transpose(U, 1, 2).reshape(-1, k_values_per_patch)
    W = torch.rand((k_values_per_patch, output_channels))
    ctx.save_for_backward(input_batch, U_reshaped, W)
    Y = torch.mm(U_reshaped, W)
    Y_reshaped = Y.reshape((batch_size, patches, output_channels))
    Y_permuted = torch.permute(Y_reshaped, (0, 2, 1))
    output_batch = Y_permuted.reshape(batch_size, output_channels, h_out, w_out)
    return output_batch
```

Question 1.4/5/6

As can be seen the operations that are used are reshapes, matrix multiplication, permutation and transpose. Thus almost all of these operations are ways to redefine the structure of the data (in other words, moving of values from one place to another. For these operations the gradients are easy to obtain, since values aren't actually changed. Let's have a look at the mathematical derivation of such a forward function. We'll work backwards from the output of the forward function. The last operation to get from Y_permuted to the output_batch is a reshape operations where simply gradients are moved from one place to another. Thus reversing this processes suffices to get the correct gradients. In this case this means translating the gradient at index (batch_size, output_channels, width and height) towards a gradient at position (batch_size, output_channels, width*height). So one dimension vanishes here. The next backward step is to go from Y_permuted to Y_reshaped. This is only the swapping of dimensions, thus this can easily be re-done by swapping them back. The dimensions that have to be swapped are the once representing the output_channels and the one representing all gradients per pixel (as created in previous derivative). The next step is again a reshape, thus it requires the reverse of this process again, meaning that gradients for all patches for each batch must be replaced underneath each other in one dimension (instead of being seperated per batch in a seperate dimension). Now we have the gradients for the output of the matrix multiplication. This is where it get's more interesting and here we'll work out the derivative with respect to the weights mathematically.

$$\frac{\partial Y}{\partial W_{k,o}} = \frac{\partial U * W}{\partial W_{k,o}} = \frac{\partial \sum_i U_{i,k} * W_{k,o}}{\partial W_{k,o}} = \sum_i U_{i,k}$$

Now combining this with the output w.r.t. the loss we get:

$$\frac{\partial l}{\partial W_{k,o}} = Y^{\nabla} * \frac{\partial Y}{\partial W_{k,o}} = Y^{\nabla}.T * U$$

The latter part contains the transpose versions in order to let the dimensions match (Y = (torch.Size([16384, 8])), U_reshaped = (torch.Size([16384, 27]))). In updating the weights of a single layer this computation is

enough to do that. However often in CNN structures multiple layers are used, and therefore we need to back-propagate fully towards the input of the layer. Let's start by having a look at the derivative for Y w.r.t. U_reshaped (which is quite similar to the previous one).

$$\frac{\partial Y}{\partial U_{b*p,k}} = \frac{\partial U * W}{\partial U_{b*p,k}} = \frac{\partial \sum_o U_{b*p,k} * W_{k,o}}{\partial U_{b*p,k}} = \sum_o W_{k,o}$$

Again starting form the loss gives:

$$\frac{\partial l}{\partial U_{b*p,k}} = Y^{\nabla} * \frac{\partial Y}{\partial U_{b*p,k}} = Y^{\nabla} * W.T$$

Finally the unfold operation has to be reversed. This can be done by using the fold function (integrated in pytorch).

This results in the following backward function:

```
@staticmethod
def backward(ctx, grad_output):
    input_batch, X_reshaped, W = ctx.saved_tensors
    grad_Y_permuted = grad_output.reshape(grad_output.size()[0], grad_output.size()[1],\
        grad_output.size()[2]*grad_output.size()[2])
    grad_Y_reshaped = torch.permute(grad_Y_permuted, (0, 2, 1))
    grad_Y = grad_Y_reshaped.reshape(grad_Y_reshaped.size()[0]*grad_Y_reshaped.size()[1],\
        grad_Y_reshaped.size()[2])
    kernel_grad = torch.transpose(torch.mm(torch.transpose(grad_Y,0,1), X_reshaped),0,1)
    grad_U_reshaped = torch.mm(grad_Y, torch.transpose(W,0,1))
    grad_U = torch.permute(grad_U_reshaped.reshape(input_batch.size()[0], \
        int(grad_U_reshaped.size()[0]/input_batch.size()[0]), kernel_grad.size()[0]), (0, 2, 1))
    input_batch_grad = F.fold(grad_U, output_size=[input_batch.size()[2], input_batch.size()[3]],\
        kernel_size=(3,3), padding=1)
    return input_batch_grad, kernel_grad, None, None
```

1.4 MNIST

Now suppose we want to perform classification on a the large MNIST dataset by using a CNN. At first we take 2 different datasets one for training purpose and one for testing purpose. We do this by extracting the data from torchvision.MNIST (shuffling the input data). Than the training set is split in 50000 train instances and 10000 validation instances.

Question 2.7

These sets are thrown into a Dataloader object to allow for easy iteration over the training batches. Now the sets can be used in the following training and validation loops (where loaders refers to the Dataloader that is created for each set).:

```
def validate(net, loaders):
    net.eval()
    correct = 0
    with torch.no_grad():
        for x, y in loaders['val_set']:
            output = net(x)
            _,pred_y = torch.max(output, dim = 1)
            correct += (pred_y == y).float().sum()

    print('accuracy on validation set', (correct / 10000)*100, '%')
    return (correct / 10000)*100

def train(net, loaders, epochs, loss_f, opt):
    train_loss = []
    epoch_list = []
    acc_list = []
```

```
    for i in range(epochs):
        print('epoch = ', i)
        for j, (x, y) in enumerate(loaders['train_set']):
            opt.zero_grad()
            x_batch = x
            y_batch = y
            output = net.forward(x_batch)
            loss = loss_f(output, y_batch)
            train_loss.append(loss)
            if j % 1000 == 0:
                print('loss:', loss.item())
            loss.backward()
            opt.step()
        epoch_list.append(i)
        acc_list.append(validate(net, loaders))
    return train_loss, epoch_list, acc_list
```

Using the hyperparameters as described in table 1 we obtained an accuracy of 97.83% on the validation set.

learning rate	0.001
epochs	5
stride	1
padding	1
kernel	(3,3)
batch_size	16

Table 1: Hyperparameters Baseline CNN

Question 2.8

1.5 Data Augmentation

To improve performance we would like to add some variation to the data instances that is trained on. This can be done by performing data augmentation. With data augmentation adjustments are made to a picture (like rotating, zooming in color adjustments, shifts etc) in order to generalise the learned behaviour. We used the following transform function to augment our data:

```
augmented = transforms.Compose(
                    [
                    transforms.RandomRotation((-7.0,7.0),fill=(1,)),
                    transforms.RandomAffine((-3,3),translate =(0.05,0.05)),
                    transforms.ToTensor(),
                    transforms.Normalize((0.1308,), (0.3016,))
                    ])
```

Question 2.9

This function is a combination of transforms that are found by searching for effective approaches online (Kaggle competitions). The transform contributes to the diversity of the dataset and improves it's learning ability. Using the same hyperparamaters as for the baseline CNN the accuracy improved to 98.36%.

1.6 Variable Resolutions

Not always all pictures in a dataset are equal in size (almost always not the case). Therefore the CNN needs to generalise somehow for different image sizes. For example, we have an input image consisting of 3 channels with a dimension of 1024x768. Suppose the image has to be converted to an image with 16 channels by using a 5x5 kernel with stride=2 and padding=1. The output will be an image with 16 layers and a size of 511x383 (torch.Size([1, 16, 511, 383]) (in fact because of the stride of 2 the outcome in dimensions is 511.5, since we

do not perform a half convolution the convolution function stops at 511). This means per image there are 16 layers of a width of 511 and a height of 383. The output for an image with the following dimensions: 1920x1080 (and 3 channels) in the same convolutional layer is torch.Size([1, 16, 959, 539]). Thus the dimensions are very different (and the convolution is able to compute them) which can be a problem when in the next layer one particular shape is expected. However when the input consists of more channels, this convolution operation cannot be used anymore since it expects a different number of values per patch (k) than it gets for a different amount of channels.

Question 3.10

A way to get shapes of images alligned for different inputs is by letting the images go through a global pooling layer. Than the outcome dimension is specified and the input is sort of squeezed into this shape. Max pooling takes the maximum value in a certain window, while mean pooling takes the average. This can be implemented in pytorch the following:

```
input_tensor = torch.rand((b, c, h, w))
# AVG
global_average_pooling = torch.nn.AvgPool2d((h,w), stride=None, padding=0)
# MAX
global_max_pooling = torch.nn.MaxPool2d((h,w), stride=None, padding=0)
# Execute
avg_pooled = global_average_pooling(input_tensor)
max_pooled = global_max_pooling(input_tensor)
```

Question 3.11

The outcomes of these 2 functions will be a (b,c,1,1).
Another approach to fit all images into a single dimension is to resize all pictures by transforming them to the smallest available size. As can be imagined this throws away a lot of useful informational pixels. The results of the learning, with similar hyper parameters as for the baseline, shows this is the case. An accuracy of 91.95% is reached after the 5 epochs (which is significantly less than the baseline and the augmented dataset).

Question 3.12

So when resizing the pictures to a smaller size is no solution, would it be a good idea to upscale them all to the largest available resolution? Besides lots of extra computations a downside is that as with zooming in on a picture you can get very blurry images which barely represent what the label tells it does. This would sometimes only add noise to the dataset.

Question 3.13

2 Global Max and Average Pooling

Question 3.14

To let every image, with a unique image size, pass the CNN successfully, either reshaping the initial resolution down to the lowest resolution in the data or letting every image pass through the network one by one (batch=1) is a solution to this problem. However, as explained earlier, it is possible to feed the neural network different resolutions of images. The achieve this, one can add a Global Pool layer at the end of the network. A Global Max Pool Layer and a Global Average Pool layer are compared in an experiment in terms of the classification performance of the network on the validation set during the training epochs. The implemented network class with the Global Average and Global Max Pooling is presented below.

```
class Net(nn.Module):
    def __init__(self, input_chan, kernel, strd, padding, output, pool):
        super().__init__()
        self.pool = pool
        self.conv1 = nn.Sequential(nn.Conv2d(input_chan, 16, kernel, strd, padding), nn.ReLU(), nn.MaxPool2d(2,2))
        self.conv2 = nn.Sequential(nn.Conv2d(16, 32, kernel, strd, padding), nn.ReLU(), nn.MaxPool2d(2,2))
        self.conv3 = nn.Sequential(nn.Conv2d(32, 81, kernel, strd, padding), nn.ReLU(), nn.MaxPool2d(2,2))
        self.out = nn.Linear(81, output)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        if self.pool == 'avg':
            global_avg_pooling = torch.nn.AvgPool2d((x.shape[2],x.shape[3]))
            x = global_avg_pooling(x)
        if self.pool == 'max':
            global_max_pooling = torch.nn.MaxPool2d((x.shape[2],x.shape[3]))
            x = global_max_pooling(x)
        x = torch.flatten(x, 1)
        output = self.out(x)
        return output
```

The CNN was set up with three convolution layers; each layer applies sequentially a ReLU and then a max-Pool(2,2). After the three convolution layers the Global Max or Average Pool is applied. The last linear layer provides the desired output. The 3rd Convolution layer has input of shape (batch, 32, 7, 7) and an output of shape (batch, 81, 7, 7), resulting in a network with 29029 parameters. The output of the third layer with N=81 was set to compare the CNN with Global Max/Average Pooling with the CNN with the fixed input of image size of 64x64 (containing 29066 parameters).
The Cross Entropy Loss functions is initialised and n Adam optimiser is used with a learning rate of 0.0001. The batch size is set to 64 and the number of epochs is set to 20.

Question 3.16

The results of this experiment are presented in Figures 2 and 3. The results show that Global Max Pooling has a better performance on the validation set in terms of accuracy and average loss, compared to the network that was trained with Global Average Pooling.

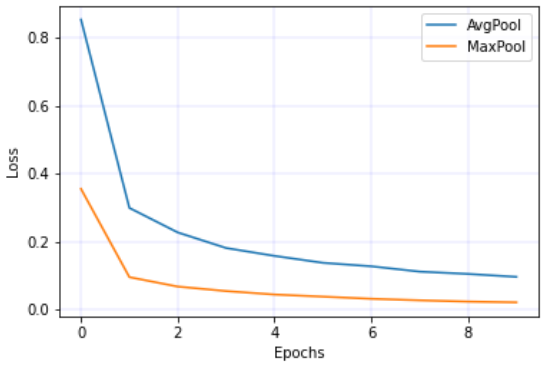

Figure 2: Loss during training epochs with global Max and Avg Pooling

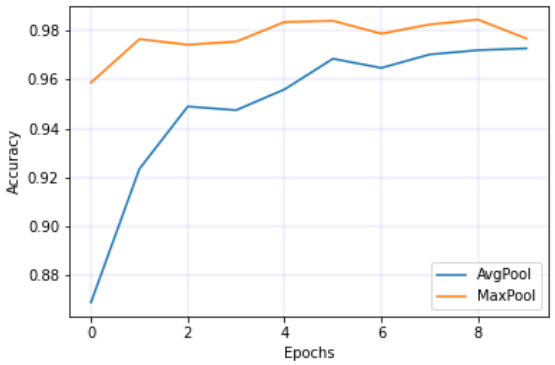

Figure 3: Accuracy during training epochs with global Max and Avg Pooling

2.1 Comparing Two Networks

Question 3.17

Ultimately it is interesting to investigate whether the network with the down shaped reshaped input (28x28) or the network with a Global Max Pooling layer has a better performance in terms of accuracy. For the research both the network are tuned on learning rate, batchsize and epochs. First the learning rate is investigated. For both networks the batchsize is set to 16. Learning rates ranging from 0.0005 to 0.005 are tested with a step size of 0.0005. To cap training time for this experiment, epochs were set to 5. The results of this experiment indicated that the best values for the learning rates were 0.0010 and 0.0015 for the Global Max Pool network and the reshape network respectively. Similarly, to the learning rate experiment, both networks with different batchsize were tested with batchsizes of 4, 16, 32 an 64. Decreasing the batchsize often means that epochs should increase to achieve higher accuracy's (this is not a general rule), but it could be possible to yield higher

accuracy's with lower batchsizes. For testing the batchsizes the epochs were set to 10, again with the training time in mind. From the experiment it was concluded that setting the batchsize to 4 and 16 for the Global Max Pool network and the reshaped input network respectively. After 8 epochs Global Max Pooling network achieved its highest accuracy of 98.9 % on the validation set during training. For the reshaped input network the highest accuracy of 97.4% was achieved after 7 epochs (Figure 4.
Now that both networks are tuned, the networks will predict the labels for the data in the test set. On the test set the Global Max Pooling networks classifies with an accuracy of 98.7%. The Network with the reshaped shaped input of 28x28 achieves an accuracy of 96.9%

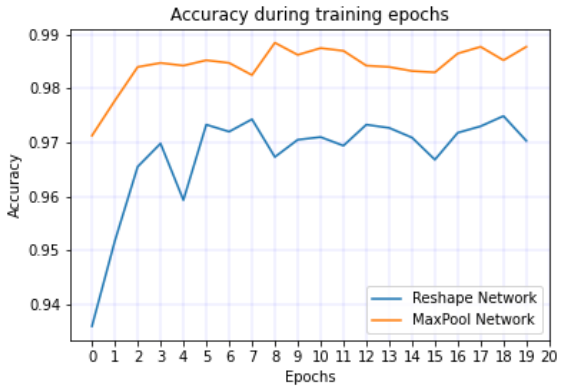

Figure 4: Accuracy during training epochs with the Global Max Pooling and the input Reshape Network