Assignment 3A: Implementing convolutional neural networks

Goal: Implement convolutional neural networks (CNNs) in PyTorch.

Submission: The assignment consists of two parts: implementation and an analysis. You are supposed to present results for both parts in the following manner:

1. Upload your code.
2. Prepare a report with an analysis of results obtained at home.

The code and the report must be uploaded due to the deadline to Canvas.

UPLOAD A **SINGLE FILE** (a zip file) containing your code and the report. Name your file as follows: [vunetid]_[assignment number].

Changelog

Dec 1. Fixed the dimensions of tensors in the architecture for Q7. Fixed row/column confusion in the initial pseudocode.

Introduction

In this assignment, we will look at convolutional neural networks for images. We will start by implementing a convolution from scratch in pytorch, then build (using the pytorch implementations) a complete classifier for MNIST digits, and finally, we will build a classifier that can handle input images of variable sizes.

The deliverable for this assignment is a *report*. The code contains various questions, but the answers to these should be structured into a coherent story rather than just a list of answers. [1]

Part 1: Implementing a Convolutional layer

In this part, we will investigate how to implement the convolutional layer efficiently. We'll start with a non-vectorized implementation. The convolution works as follows: given an input tensor x with dimensions

```
    (batch_size, input_channels, input_width, input_height),
```

[1] You don't have to pretend that you invented the convolutional layer. Just write a report on how you came to your implementation of the convolutions and the various models.

an amount of padding, a number of output channels, kernel size, and a stride, we produce an output tensor with dimensions

```
(batch_size, output_channels, output_width, output_height).
```

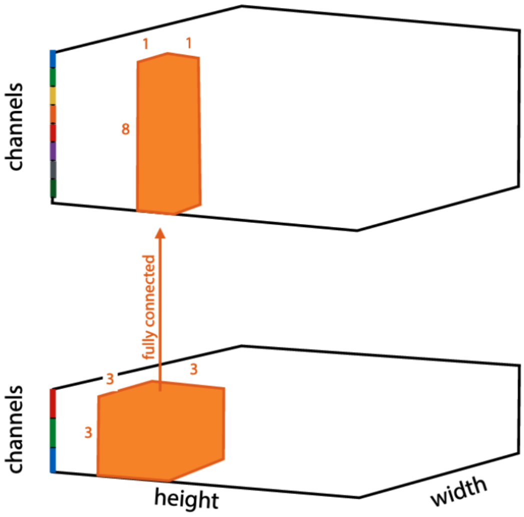
Question 1: Write a pseudo-code for how you would implement this with a set of nested `for` loops. The convolution is defined by a set of weights/parameters which we will learn. How do you represent these weights?[2]

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.

> You can check if your answer is correct by passing a tensor with random values through the conv2d function of PyTorch:
> https://pytorch.org/docs/stable/generated/torch.nn.functional.conv2d.html
> and check the size of the result.

A lot of nested loops are obviously a very inefficient way of implementing the convolution. We'll try to figure out a *vectorized* implementation.

To start with, imagine a convolution with 1 input channel and 1 output channel. In this case, what we compute for each image patch, is essentially a dot product between the patch and the kernel. If we have a 3x3 kernel and stride 1, then we are taking all possible patches of 3x3 pixels from the input. For each of these patches, we element-wise multiply by 9 parameters and sum to create the value for one dimension of the output.



[2] Most convolution layers have a bias term, but we'll ignore that for now. This is always just a set of parameters of the exact shape of the output, which is added to the result just before returning.

If we have *multiple* input channels, then our kernel has multiple input channels to match. If we have multiple output channels, then our kernel has an extra dimension with extra output channels to match. This seems complicated, but if we look at what happens between one image patch and its output pixel, it's just a fully connected layer: each node of the input is connected to each node of the output with a single weight. Everything else is just dimensions to keep track of the shapes.

Using this insight, we can implement the convolution with the following steps:

1. Extract all patches from the input
2. Flatten these patches (with all channels) into vectors, arranged as the rows of a matrix **X**.
3. Multiply this matrix by a weight matrix **Y = XW**.
4. Reshape the matrix **Y**, so that its rows become the pixels of the output tensor.

Extracting all patches is a difficult operation to vectorize, so we'll defer to pytorch for that one. This is done by the *unfold* operation:
https://pytorch.org/docs/stable/generated/torch.nn.functional.unfold.html

This operation takes an image with size `(b, c, h, w)`, extracts all patches that a convolution (with given kernel size, stride, and padding) would extract, flattens them and returns the result in a matrix with size `(b, k, p)`, where `p` is the number of patches, and `k` is the number of values per patch. Note that the patches are flattened into vectors: for a 3x3 kernel, and 5 input channels, we would get `k = 45`. If the images are 32x32 pixels, and padding and stride are 1, then `p = 32 * 32 = 1024`.

Question 3: Write a naive (non-vectorized) implementation of the unfold function in pseudocode. Include the pseudocode in your report.

We will implement the convolution twice using the algorithm above. First, as a PyTorch *Module*, leaving PyTorch to work out the backward, and then as a PyTorch *Function*, implementing both the forward and backward ourselves.

For the module, all we need is a python object that inherits from `nn.Module`, and implements a method called `forward`. Here is some skeleton code that you can build on:

```
import torch
import torch.nn.functional as F
from torch import nn

class Conv2D(nn.Module):

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

```
    def forward(self, input_batch):

        b, c, h, w = input_batch.size()

        # your code here

        return ...

# We use the Conv2D module by instantiating it, and applying it to an input.
conv = Conv2D()
input_batch = torch.randn(16, 3, 32, 32)
output_batch = conv(input_batch)
```

Some tips:

- Make sure to use assert statements liberally. For instance, before you return the result of the convolution, you can compute the expected output size with the function described above. If you add an assert, these two must always match.
- After folding, we have a `(b, p, k)` tensor that we want to multiply by our parameter matrix, treating both the b and p dimensions as batch dimensions. The simplest solution is to reshape to a `(b*p, k)` tensor, apply a batched matrix multiply, and reshape the result back from a `(b*p, l)` tensor to a `(b, p, k)` tensor.
- For our implementation, we can think of the kernel as a matrix (**W** in the algorithm above). This means that self.kernel can just be a simple matrix. The number of rows is the number of nodes in one patch of the input, and the number of columns is the number of nodes in one pixel in the output.
- After multiplying by **W**, and reshaping, the result **Y** contains one row-vector for each pixel in the output. A permutation and a reshape is enough to turn **Y** into the output tensor. You should be able to work out the output resolution from the parameters.

If we implement this operation as a Module, we don't need to implement the backward. However, to see how the gradients operate in a convolution, it's instructive to work out the backward and to implement it ourselves. For this, we'll use a Function, the PyTorch equivalent of the operation in the previous exercise. Here is some skeleton code to work from:

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

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

        # store objects for the backward
        ctx.save_for_backward(input)
        ctx.save_for_backward(kernel)

        # your code here
        ...

        return output_batch

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the
        gradient of the loss with respect to the output, and we need
        to compute the gradient of the loss with respect to the
        input
        """

        # retrieve stored objects
        input, = ctx.saved_tensors

        # your code here
        ...

        # The gradients of the inputs. For anything that doesn't have
        # a gradient (the stride and padding) you can
        # return None.
        return input_batch_grad, kernel_grad, None, None

input_batch = torch.randn(16, 3, 32, 32)
kernel = torch.randn(...)
Conv2DFunc.apply(input_batch, kernel)
output_batch = conv(input_batch)
```

At first blush, this looks similar to the Module. Note, however, that this is an object with only static methods. We never instantiate the object, we just call a static *apply* method, which calls the forward in such a way that a computation graph is built up.

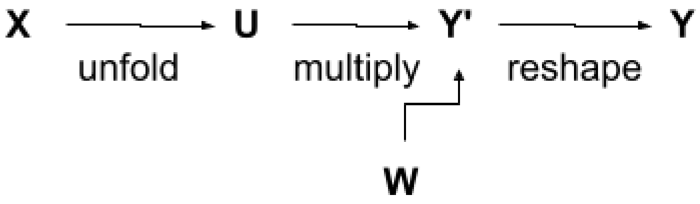
Question 4: Work out the backward with respect to the kernel weights **W**.

Notes:

- We recommend that you draw the basic operation of the convolution as implemented above, and be precise in naming the dimensions of all tensors. Then follow the approach for working out a backward function very mechanically, and in small steps.
- For this backward, you don't need to backpropagate through the unfolded input image. The basic operations are a reshape and matrix multiplication.
- You can work out the backward in one go, or draw a computation graph for the unfold, matrix-mulitplication and reshape operations, and work out the backward of these and chain them manually in the backward of the `Function`.

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 **X**.

For this question you should separate the three steps in a computation graph and work out separate backwards first. I.e. let the computation be



Then you should work out $\mathbf{Y'}^{\triangledown}$ as a function of $\mathbf{Y}^{\triangledown}$, $\mathbf{U}^{\triangledown}$ as a function of $\mathbf{Y'}^{\triangledown}$, and then $\mathbf{X}^{\triangledown}$ as a function of $\mathbf{U}^{\triangledown}$**.** Once you have these, you can chain them together (autograd would normally do this for us, but here the exercise is to do it yourself).

Notes:

- This question requires you to backpropagate through the unfold operation, which gets a little hairy. As ever, start with the scalar formulation and see what you can vectorize.
- Once you've worked out the scalar formulation, you can see what's required to vectorize it. As in the forward, we'll need some special functions. The fold operation should do the trick.
    - In the previous exercise, we saw that when we don't change the values in a tensor, but just repeat them in another shape, the resulting backward becomes a sum over the upstream gradients of all the repeats. Note that the unfold operation is of this kind: it doesn't change any values from the input, it just repeats them.

- In general, it's a good bet that if you use some function f() in the forward, you'll need its inverse f^{-1}() in the backward. It's not guaranteed, but it often happens.

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).

Notes:

- If you don't quite understand how the `Function` works, have a look at the pytorch example, which implements the ReLU as a `Function`. You can also go back to the `Op` class in the `vugrad` system of the last assignment, which works very similarly.

Part 2: Implementing a simple Convolutional classifier

Now that we have a sound understanding of convolutions, let's build a simple convolutional neural network to classify the MNIST data.

MNIST is available as part of pytorch, we can load it as follows:

```
import torchvision
train = torchvision.datasets.MNIST(root=arg.data, train=True,\
                               download=True, transform=ToTensor())
trainloader = torch.utils.data.DataLoader(train, batch_size=arg.batch,\
                               shuffle=True, num_workers=2)

test = torchvision.datasets.MNIST(root=arg.data, train=False,\
                               download=True, transform=ToTensor())
testloader = torch.utils.data.DataLoader(test, batch_size=arg.batch,\
                               shuffle=True, num_workers=2)
```

This gives us dataloaders that we can loop over as we did in the 60 minute blitz in the last exercise.

However, if we want to do this properly, we should split the training data into train and validation when we are designing our network. The dataloaders don't allow us to do this easily. There are ways to achieve it, but it's good to know that you can build a perfectly fine training loop without the data loaders, when a little more flexibility is required. This is especially true for small datasets like MNIST, that we can easily load entirely into memory.

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.

Notes:

- Note that the range() function in python can be called with three arguments, to iterate in large steps.

- It's useful to know that the slice syntax is forgiving if the second argument is out of bounds. If you have an array with 32 elements then `arr[30:36]`, will just give you the last two elements without complaining.
- If you have a GPU, make sure not to load these data tensors into GPU memory. It's better to load each batch into the GPU memory as you loop over the data.

We'll use the following network.

input size	**layer / operation**
(batch, 1, 28, 28)	**`nn.Conv2d`** kernel size 3x3, stride 1, padding 1, 16 output channels
(batch, 16, 28, 28)	`ReLU`
(batch, 16, 28, 28)	`2x2 Max pool`
(batch, 16, 14, 14)	**`nn.Conv2D`** kernel size 3x3, stride 1, padding 1, 32 output channels
(batch, 32, 14, 14)	`ReLU`
(batch, 32, 14, 14)	`2x2 Max pool`
(batch, 32, 7, 7)	**`nn.Conv2D`** kernel size 3x3, stride 1, padding 1, 64 output channels
(batch, 64, 7, 7)	`ReLU`
(batch, 64, 7, 7)	`2x2 Max pool`
(batch, 64, 3, 3)	`Reshape to flatten the image`
(batch, 64 * 3 * 3)	`Linear layer`
(batch, 10)	

Use binary cross-entropy as the loss function, and Adam as the optimizer. Most of this was covered in the PyTorch 60 minute blitz.

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.

In the lectures, we briefly discussed the idea of data augmentation. This is easily added to our 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.

Notes:

- When we use data augmentation, loading the whole data into memory creates a problem. If we use the dataloader, then the data augmentation is randomly applied each time. If we preload the data the dataloader only applies sthe data augmentation once, defeating the purpose. Here, we'll need to use pytorch's random split function instead: https://pytorch.org/docs/stable/data.html#torch.utils.data.random_split
    - Apply this to the dataset object before passing it to the dataloader.

Part 3: A variable-resolution classifier

In the real world, images rarely all come in the same resolution. The nice thing about the convolutional layer is that we can apply the same kernel to images of different resolutions.

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?

With this principle in hand, we can stack a series of convolutions together without worrying about the input resolution. For each input with a different resolution, all the intermediate tensors will have different resolutions, but the whole thing will still work.

At some point in the network, however, we need an operation that takes such a tensor of a variable resolution and maps it to a tensor that has a fixed size, regardless of the input dimensions. The simplest way to do this is a **global pooling operation**. You can think of this as a max pool, but instead of applying the max over a patch of the image, it applies the max over the whole of the height and width dimension, removing them from the output.

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. The result should be a tensor with dimensions (b, c).

To use these ideas in practice, we'll need a variable resolution dataset. We've created a simple version of MNIST, where all the images have resolution either 32x32, 48x48 or 64x64. You can download the dataset here:

https://dlvu.github.io/data/mnist-varres.tgz

The data is provided as image files collected by class in folders. This is a standard way of arranging data, which you can load with the ImageFolder dataset.

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.

Have a look at the 64x64 images in the data. As you can see, we didn't blow up the digits, we just added extra black pixels. This means we shouldn't expect much to remain when we resize this to 28x28.

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?

Let's build a variable resolution network. First, we need to load the data. We can't use a standard dataloader with making the resolution uniform. Somehow we'll need to group the data by resolution, and then batch together images with the same dimensions. It's possible to do this in a dataloader, but it's easier to just load the whole data in memory again.

The network we'll build can deal with images of any resolution, but within a single batch, the resolution needs to be uniform (because it needs to be one tensor). for this dataset, that is relatively simple to achieve.

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?

Notes:

- The python package `glob` is a good tool for iterating over files.

Construct the following network:

input size	**layer / operation**
(batch, 1, 28, 28)	**`nn.Conv2d`** kernel size 3x3, stride 1, padding 1, 16 output channels
(batch, 16, 28, 28)	`ReLU`

(batch, 16, 28, 28)	`2x2 Max pool`
(batch, 16, 14, 14)	**`nn.Conv2D`** kernel size 3x3, stride 1, padding 1, 32 output channels
(batch, 32, 14, 14)	`ReLU`
(batch, 32, 14, 14)	`2x2 Max pool`
(batch, 32, 14, 14)	**`nn.Conv2D`** kernel size 3x3, stride 1, padding 1, N output channels
(batch, N, 14, 14)	`ReLU`
(batch, N, 14, 14)	`2x2 Max pool`
(batch, N, 7, 7)	`Global max pool or global mean pool`
(batch, N)	`Linear layer`
(batch, 10)	

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).

Notes:

- model.parameters() loops over all parameters tensors of a network. Just multiply all dimensions of these tensors, and sum the result.
- Note that the input resolution doesn't affect the number of parameters of the model.

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.

Question 17: Tune both networks (without data augmentation), and then compare the test set performance of the fixed-resolution network with 64x64 input resolution to the performance of the variable resolution network. Report your findings.

Notes:

- You may expect the variable network to perform better, but that is not a given. The global max pool is a fairly crude operation, and it may well harm performance in this case.

- If you feel like running further experiments, try increasing the depth and the amount of 2x2 maxpooling, so that the final resolution gets closer to 1x1. That should reduce the downsides of the global pooling.

Grading (10pt)

- Question 1: 0.25pt
- Question 2: 0.25pt
- Question 3: 0.25pt
- Question 4: 0.75pt
- Question 5: 0.75pt
- Question 6: 0.5pt
- Question 7: 0.5pt
- Question 8: 1.25pt
- Question 9: 1.25pt
- Question 10: 0.25pt
- Question 11: 0.25pt
- Question 12: 0.5pt
- Question 13: 0.25pt
- Question 14: 0.5pt
- Question 15: 0.5pt
- Question 16: 1pt
- Question 17: 1pt