COMP3547: Deep Learning Practical 3

Designing architectures

christopher.g.willcocks@durham.ac.uk & amir.atapour-abarghouei@durham.ac.uk

This is the version *with* answers!

1 Overview

Welcome to the third deep learning practical. This is a coding practical with the aim of introducing some best-practices in designing deep neural networks along with some new tools to use in your armoury. In this practical, we will be covering:

- Tensor manipulation with Einstein operations.
- Convolutional architecture design.
- Residual architectures.
- Block design.

2 Tensor manipulation with Einstein operations.

A running joke in our lab a few years ago was that deep learning research involves 90% tensor wrangling and 10% actual research. In other words it used to be very difficult getting your input data processed and arranged in the right memory layout, such that it's suitable for feeding into a neural network.

Inspired by Einstein notation, 'einops' makes the process of reshaping and manipulating tensors much easier, more optimised, and less error-prone.

Let's start with a colour image of some coffee, where we want to simulate a batch of data by repeating it 10 times across the batch dimension:

```
!pip install einops
!pip install scikit-image

import skimage
import torch
import einops

from skimage import data, io
image = data.coffee()
io.imshow(image)

x = torch.tensor(image)
x = x.float()/x.max()
print(x.size())

> torch.Size([400, 600, 3])
```

Oh no—this is no good! This is $H \times W \times C$ (height, width, channel) but we want $B \times C \times H \times W$, and we also want to repeat it 10 times across the batch dimension. While cumbersome and sometimes inefficient to permute the tensor to the desired memory layout manually in PyTorch, it is very easy to achieve this with einops:

```
from einops import rearrange, reduce, repeat

coffee_batch = repeat(x, 'h w c -> b c h w', b=10)
print(coffee_batch.size())
```

- Run the above code and confirm it has worked as intended.

Einops supports built-in layers for all mainstream deep learning libraries (torch, tensorflow, JAX via flax etc):

```
from einops.layers.torch import Rearrange, Reduce

nn.Sequential(
    Rearrange(...),
    nn.Linear(...),
    Reduce(...)
)
```

Exercise

- Create a new multilayer perceptron `MLP(nn.Module)` class. This will just be a simple network with two `nn.Linear(...)` layers.
- Given your `coffee_batch` tensor of $B \times C \times H \times W$, and using the einops layer 'Rearrange', construct a 2-layer MLP whose first layer has $3 \times 400 \times 600$ input features (for each colour pixel) and 5 outputs.
  - In einops, you can flatten dimensions by putting them in circular brackets.
  - For example '`b c h w -> b (c h w)`' will keep the batch dimensions and flatten the channels and spatial dimensions into a vector, making it suitable for input to a linear layer.
- Use a ReLU nonlinearity and make the second linear layer map from 5 inputs to 1 output. Put all these layers in a `self.layers = nn.Sequential(...)` and return a call to this in the forward pass.
- Initialise `net = MLP()`
- Evaluate `net(cofee_batch).size()` and confirm the output is `torch.Size([10, 1])`.

Exercise solution

```
from einops.layers.torch import Rearrange, Reduce                                    </> copy code

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            Rearrange('b c h w -> b (c h w)'),
            nn.Linear(3*400*600, 5),
            nn.ReLU(),
            nn.Linear(5,1)
        )
    def forward(self, x):
        return self.layers(x)

net = MLP()
print(net(coffee_batch).size())
print('Number of MLP parameters: ' +
    str(len(torch.nn.utils.parameters_to_vector(net.parameters()))))

> torch.Size([10, 1])
> Number of MLP parameters: 3,600,011
```

- Have a look through `http://einops.rocks/pytorch-examples.html` and study the different einops uses cases of 'rearrange' and 'reduce'.
- You can do very complicated manipulations easily. Run the following and explain what is happening:

```
img_grid = rearrange(coffee_batch, '(b1 b2) c h w -> (b1 h) (b2 w) c', b1=2)
io.imshow((img_grid*255).int().numpy())
```

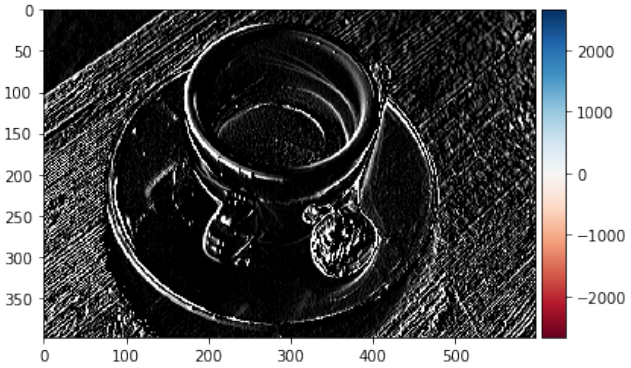
The batch dimension is split into two new dimensions labeled b1 and b2. Only the size of b1 is defined b1=2 so it'll workout that b2 needs to be 5 as our batch has 10 images. By itself this would yield e.g.: `torch.Size([2,5,3,400,600])`. However, in the output we also rearrange b1 with the height, b2 with the width, and also move the channel to the last dimension as the imshow function requires a $H \times W \times C$. This creates a new single image showing a grid of the previous images. Doing all that manually would require a lot of code!

3 Convolutional neural networks

A convolution is a sliding dot product between a matrix (kernel) and the section of the signal (image, audio...) that it's aligned with. The kernel is typically something small, like $4 \times 4 = 16$ parameters, and it remains the same size regardless of how big or small the input image is. In contrast, a linear layer requires as many parameters as there are pixel inputs in the image, multiplied by the number of desired output neurons. In other words, a linear layer requires *a lot* of parameters for large images, text sequences, or audio clips.

Recall that when you apply a convolution to a colour image, you need to do it separately for each input channel. Assuming we would like to learn a different convolution operation for each RGB input channel, we would need to learn $3 \times 4 \times 4 = 48$ parameters for a 4×4 kernel. We can do simple 2D convolution in PyTorch manually, as in traditional image processing, like this Sobel filter to detect edges:

```
conv = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=3, bias=False)           </> copy code

print(conv.weight.size())
print('Number of parameters: ' + str(len(torch.nn.utils.parameters_to_vector(conv.parameters()))))
print(conv.weight.requires_grad)

w = repeat(torch.tensor([
        [-1,0.,1],
        [-2,0.,2],
        [-1,0.,1]
    ]), 'h w -> out_f in_f h w', out_f=1, in_f=3)

conv.weight.data = w
out = conv(coffee_batch)

io.imshow((repeat(out[0,0], 'h w -> h w c', c=3)*255).int().numpy())
```



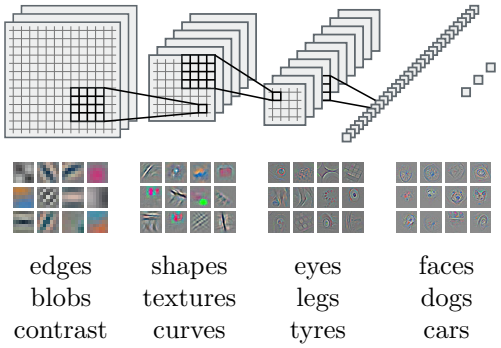
The main difference in deep learning is that we wish to learn these weights, so they're initialised randomly and have `requires_grad=True` set by default. Also, just like we need to convolve separately for each input channel, we also need to convolve separately for each output channel (detector) that we wish to learn. For example, the Sobel filter example above only detects vertical edges in one orientation, but if we wish to learn to detect 24 different types of feature (e.g. horizontal edges, blobs, textures, corners, ...), we would need to learn 24 separate $3 \times 4 \times 4$ kernels, so our final weights tensor would be rank 4 with $24 \times 3 \times 4 \times 4 = 1152$ parameters for a 4×4 kernel. But that's still far fewer than the 3.6 million parameters we required for our linear layer earlier.

Convolutional design

So how many output channels? Well, generally there are two things we wish to achieve: encoding and decoding. When we encode, we wish to learn a latent representation (feature vector) that describes the image abstractly. So the image may be represented by a feature vector with a small compressed set of neurons that activate if the image contains faces, red hair, trees, cars etc.

To achieve this latent encoding, we generally gradually reduce the spatial dimensions (e.g. the image halves in number of pixels each layer) while gradually increasing the number of channels:

Encoder input: $B \times 3 \times 128 \times 128$
Middle layer: $B \times 64 \times 8 \times 8$
Encoder output: $B \times 512 \times 1 \times 1$



Similarly, we can decode a latent representation back to something of the same shape as the original input, e.g.:

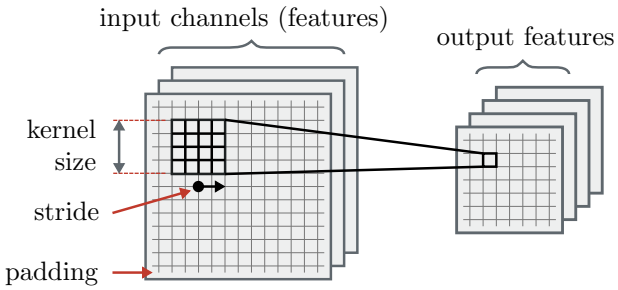
Decoder input: $B \times 512 \times 1 \times 1$
Middle layer: $B \times 64 \times 8 \times 8$
Decoder output: $B \times 3 \times 128 \times 128$

How gradual do we reduce the spatial dimensions and increase the channels? Because of the 'no free lunch theorem' that we'll cover later, it depends on the dataset and the task. If you're training a neural network to classify textures, you would likely want more channels in the earlier layers. If you were training a classifier to detect happy and sad facial expressions, you would likely want more capacity in the later layers—reflecting the deeper and more abstract decision boundaries of the task.

How do we reduce the spatial dimensions? Either use max or mean pooling layers, equivalent to this einops:

```
reduce(x, 'b c (h1 h2) (w1 w2) -> b c h1 w1', 'max', h2=2, w2=2)
```

Or we can change the stride and padding of our convolutional layers. These are best illustrated:



Instead of sliding the kernel over each pixel (stride=1), changing the stride allows us to skip pixels. Setting stride=2 has the effect of halving the spatial dimension of the output. Alternatively, instead of `nn.Conv2d(..., kernel_size=4, stride=2, padding=1)` we could use `nn.ConvTranspose2d(..., kernel_size=4, stride=2, padding=1)` giving the effect of doubling the spatial dimensions of the output—used in decoders.

- To gain some further intuition of convolution and transpose convolution, visit `https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md` and look at the visualisations.
  - Blue are the inputs (images) and green are the outputs.

The two most common patterns

For simplicity and excellent results in the majority of convolutional architectural designs, you can stick to the following two common patterns of kernel size, stride, and padding accordingly:

- The 3,1,1 pattern (kernel size = 3, stride = 1, padding = 1).
  - Allows you to change output channels to anything, without changing the spatial dimension(s).
- The 4,2,1 pattern.
  - Halves the spatial dimension(s).

Sometimes in the last layer, when there are no spatial dimensions, you can use a 1,1,1 pattern—equivilent to a linear layer. Poorly chosen kernel size, stride, and padding parameters leads to checkboard artifacts. There is an excellent interactive visualisation of this here: `https://distill.pub/2016/deconv-checkerboard/` —such artifacts in the coursework will result in lost marks.

Exercise

1. Design a 5 layer convolutional encoder that takes as input a rank 4 tensor of shape $B \times 3 \times 32 \times 32$ and outputs a representation of shape $B \times 1024 \times 1 \times 1$. Do not use max or mean pooling.
   - Start with `x0 = torch.zeros(10, 3, 32, 32)`
   - ```
     c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=4, stride=2, padding=1)
     x1 = c1(x0)
     print(x1.size())
     ```
   - ```
     c2 = nn.Conv2d(...)
     x2 = c2(x1)
     print(x2.size())
     ...
     ```
2. After you've designed the network, make an 'Encoder' class and aggregate the layers in `nn.Sequential(...)`. Feed `x0` into the network, and confirm it outputs a tensor with `torch.Size([10, 1024, 1, 1])`.

Exercise solution

```
x0 = torch.zeros(10, 3, 32, 32)                                    </> copy code

# design approach
c1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=4, stride=2, padding=1)
c2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2, padding=1)
c3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=1)
c4 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=4, stride=2, padding=1)
c5 = nn.Conv2d(in_channels=256, out_channels=1024, kernel_size=4, stride=2, padding=1)

x1 = c1(x0)
x2 = c2(x1)
x3 = c3(x2)
x4 = c4(x3)
x5 = c5(x4)

print(x1.size())
print(x2.size())
print(x3.size())
print(x4.size())
print(x5.size())

print('c1 params: ' + str(len(torch.nn.utils.parameters_to_vector(c1.parameters()))))
print('c2 params: ' + str(len(torch.nn.utils.parameters_to_vector(c2.parameters()))))
print('c3 params: ' + str(len(torch.nn.utils.parameters_to_vector(c3.parameters()))))
print('c4 params: ' + str(len(torch.nn.utils.parameters_to_vector(c4.parameters()))))
print('c5 params: ' + str(len(torch.nn.utils.parameters_to_vector(c5.parameters()))))

# implementation after design
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(3, 32, 4,2,1),
            nn.Conv2d(32, 64, 4,2,1),
            nn.Conv2d(64, 128, 4,2,1),
            nn.Conv2d(128, 256, 4,2,1),
            nn.Conv2d(256, 1024, 4,2,1)
        )
    def forward(self, x):
        return self.layers(x)

net = Encoder()
print(net(x0).size())
print('Number of encoder parameters: ' + str(len(torch.nn.utils.parameters_to_vector(
    net.parameters()))))

> torch.Size([10, 1024, 1, 1])
> Number of encoder parameters: 4,885,472
```

The above encoder is not appropriate for real use as does not have nonlinearities between convolutions.

3. How many parameters does each layer in your architecture use?

   For the above architecture:
   Layer 1 parameters: 1568
   Layer 2 parameters: 32832
   Layer 3 parameters: 131200
   Layer 4 parameters: 524544
   Layer 5 parameters: 4195328

4. How would you design an architecture that has fewer parameters, but still outputs a 1024-dimensional latent representation?

   Adjust the input/output channels such that there is less capacity, especially in the last layers, e.g. [32-64, 64-128, 128-128, 128-128, 128-1024] would have 2.5 million parameters.

5. How would you adjust the architecture for non-square input images of size $B \times 3 \times 32 \times 64$? Hint: the kernel sizes, strides and padding do not have to be integers.

   You could design a kernel that is a 3,1,1 pattern (not changing spatial dimensions) on one axis, but 4,2,1 pattern (halving spatial dimension) along the other axis, e.g:

   `c0 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=[3,4], stride=[1,2], padding=[1,1])`

   would take $B \times 3 \times 32 \times 64$ as input, and output $B \times 16 \times 32 \times 32$.

6. Design a 5 layer convolutional decoder that transforms the $B \times 1024 \times 1 \times 1$ latent representation back to $B \times 3 \times 32 \times 32$. Use `nn.ConvTranspose2d(...)` for this.

Exercise solution

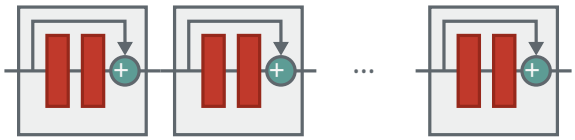
```
class Decoder(nn.Module):                                                    </> copy code
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.ConvTranspose2d(1024, 256, 4,2,1),
            nn.ConvTranspose2d(256, 128, 4,2,1),
            nn.ConvTranspose2d(128, 64, 4,2,1),
            nn.ConvTranspose2d(64, 32, 4,2,1),
            nn.ConvTranspose2d(32, 3, 4,2,1)
        )
    def forward(self, x):
        return self.layers(x)

enc = Encoder()
dec = Decoder()
z = enc(x0)
p = dec(z)
```

The above decoder is not appropriate for real use as does not have nonlinearities between convolutions.

4 Residual architectures

Sometimes it is necessary to go very deep in order to solve very challenging and abstract problems. The problem is that the optimisation landscape becomes more noisy the deeper we go, as the parameters are optimised randomly through all the transformations. Further, the gradients tend to weaken as you back propagate through many non-linearities. Batch normalisation after the convolutional layers helps alleviate this slightly.

An important breakthrough for this problem is to use residual connections, which are shortcuts that can skip over two or three layers. These are typically implemented by adding the input onto the output of several convolutional blocks, which don't change the spatial dimensions (e.g. using 3,1,1 patterns).



```
class ResidualBlock(nn.Module):                                              </> copy code
    def __init__(self, n):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(n,n,3,1,1),
            nn.BatchNorm2d(n),
            nn.ReLU(),
            nn.Conv2d(n,n,3,1,1),
            nn.BatchNorm2d(n)
        )
    def forward(self, x):
        return torch.relu(x + self.block(x))
```

These can then be inserted as with any other layer, for example as part of a `nn.Sequential` block—you just need to specify the number of input features.

5 Block design

Exercise

- Add `ResidualBlock` layers to the `nn.Sequential` part of your `Encoder` and `Decoder` classes. Check the networks still work as intended and print the number of parameters.
- Create an `Autoencoder` class that initialises `self.encoder = Encoder()` and `self.decoder = Decoder()`. Evaluate your autoencoder on `x0` and confirm it returns a tensor of the same shape.

Exercise solution

```
class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()
    def forward(self, x):
        z = self.encoder(x)
        return self.decoder(z)

net = Autoencoder()
prediction = net(x0)
print(prediction.size())
```

```
> torch.Size([10, 3, 32, 32])
```

- This approach of designing and assembling deep neural networks hierarchically from blocks can be powerful and intuitive. The majority of state-of-the-art architectures introduce new blocks and then assemble them sequentially or hierarchically to create very deep and expressive high-capacity models.

Exercise solution

```
print(net.encoder)
```

```
>
```

```
Encoder(
  (layers): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (4): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (5): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (6): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): ResidualBlock(
      (block): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (8): Conv2d(256, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  )
)
```