Group number: 530

- Rahabu Mwang’amba rahabum@es.aau.dk
- Yurii Iotov yio@create.aau.dk
- Galadrielle Humblot-Renaux gegeh@create.aau.dk

PhD course Deep learning - assignment 2

Note: we realized a mistake in the code of assignment 1 which led to under-performance (as also pointed out in the assignment feedback) - the softmax layer should not be included in the model definition since torch’s cross entropy loss expects unnormalized logits. We have corrected this, and thus all the results presented here are without the softmax layer.

Questions

1. **Input layer:** Unlike for fully connected layers, the configuration of convolutional layers does not depend on the spatial resolution of the input, but only on the number of input channels. We have grayscale/single-channel images as input, so the input layer has a single input channel.

2. **Size of flattened vector:** See the code snippet below.
   For a convolutional layer, the output “depth” (ie. the number of 2D feature maps) is determined by the number of output channels/filters. Since we have a stride of 1 but we do not apply any padding, the spatial resolution of the output is similar to the input but slightly smaller (we lose 4 pixels in height and width with a kernel size of 5).
   With max pooling of stride 2, we spatially downsample feature maps by a factor of 2 (and the depth is kept the same).

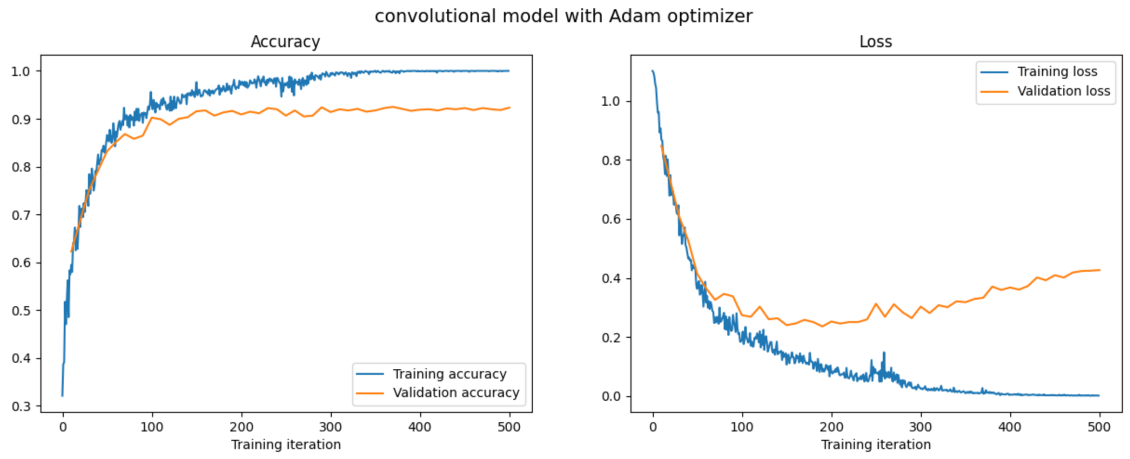
```
# given an input size and kernel parameters, calculate the output size for a single layer
    def get_output_size_single_layer(input_size=(1,101,40), out_channels=None, kernel_size=(5,5),
stride=(1,1), padding=(0,0)):
        Din,Hin,Win = input_size # input depth, height, width
        Hout = (Hin - kernel_size[0] + 2*padding[0])//stride[0] + 1 # output height
        Wout = (Win - kernel_size[1] + 2*padding[1])//stride[1] + 1 # output width
        Dout = out_channels if out_channels is not None else Din # if no output_channels specified,
assume same as input_channels (e.g. for pooling)
        return (Dout,Hout,Wout)

    # define the layer parameters for our model
    layer_params = {
            "conv1": {"out_channels": 32, "kernel_size": (5,5), "stride": (1,1), "padding": (0,0)},
            "pool1": {"kernel_size": (2,2), "stride": (2,2), "padding": (0,0)},
            "conv2": {"out_channels": 16, "kernel_size": (5,5), "stride": (1,1), "padding": (0,0)},
            "pool2": {"kernel_size": (2,2), "stride": (2,2), "padding": (0,0)},
        }
    # loop through the layers, passing the output size from one layer to the next
    def get_output_size_from_layer_params(input_size=(1,101,40), layer_params=layer_params):
        output_size = input_size
        for layer in layer_params:
            output_size = get_output_size_single_layer(input_size=output_size, **layer_params[layer])
            print(f"Output size after {layer}: {output_size}")

        flattened_size = torch.LongTensor(output_size).prod()
        print(f"Final output size: {output_size} -> when flattened {flattened_size}")
        return flattened_size

    get_output_size_from_layer_params(input_size=(1,101,40), layer_params=layer_params)
```

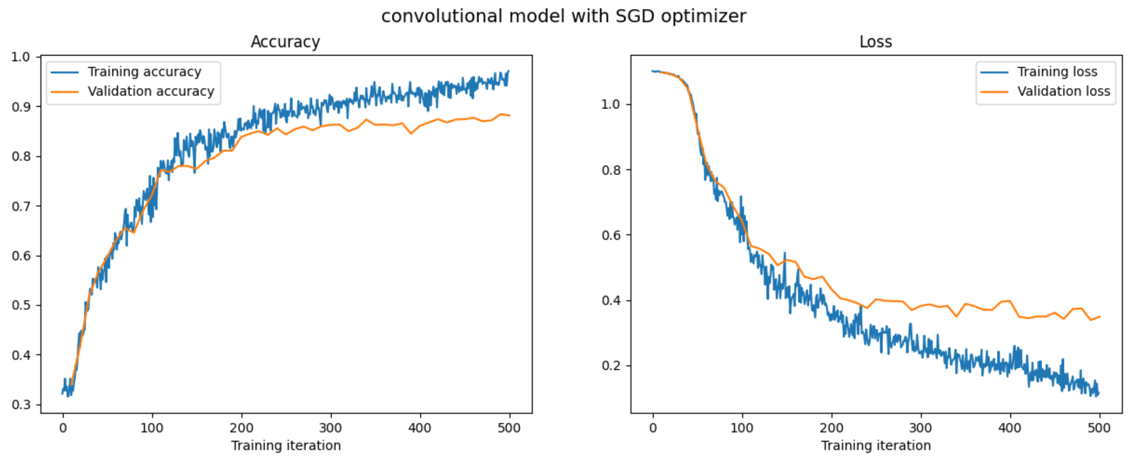
This gives the following:

```
Output size after conv1: (32, 97, 36)
Output size after pool1: (32, 48, 18)
Output size after conv2: (16, 44, 14)
Output size after pool2: (16, 22, 7)
Final output size: (16, 22, 7) -> 2464
```

We thus use 2464 as the input size for the first fully connected layer.

3. **Loss curves with SGD vs. Adam:** we compare the accuracy & training curves of both optimizers below. With Adam, we get a significantly faster convergence.





4. **Test accuracy with SGD vs. Adam:** see *Table 1* below - using the Adam optimizer improves accuracy by over 2 percentage points for both architectures.

5. **Number of parameters**: The number of parameters n_{conv} in a convolutional layer depends on the kernel height k_h and width k_w, the dimensionality of the input (ie. number of input channels) k_c, and the number of filters/output channels K. For each filter, we also have a bias term.

$$n_{conv} = K(k_h \times k_w \times k_c + 1)$$

So;

1) for the first convolutional layer, we have 32*(5x5x1+1) = 832
2) for the 2nd convolutional layer, we have 16*(5x5x32+1)=12816

For the fully connected layers, we repeat the same procedure as the previous assignment. The input size of the first fully connected layer is the flattened size calculated in question 2.

3) for the 1st fully connected layer, we have 128*(2464+1) = 315520
4) for the 2nd fully connected layer, we have…128*(128+1) = 16152
5) for the output layer, we have…3*(128+1) = 387

This adds up to a total of 346067 parameters.

6. **Performance and computational complexity of the convolutional vs. fully-connected model:** See *Table 1* below. The complexity of the two models can be compared in terms of number of parameters. In the fully-connected neural network, we have approximately 1.65 times more parameters than in the convolutional neural network. This is because of the parameter sharing properties of convolutional layers: the same filter is applied across the whole input.
   Despite the smaller model size, the classification performance of the convolutional neural network is significantly higher than the fully-connected model (with both optimisers). The convolutional neural network is better suited to extract relevant features from images as it captures spatial structure and local correlation, and it introduces translation invariance (the same parameters are used regardless of input location).

Table 1: Summary of performance (test accuracy after 50 epochs) for the 2 architectures & optimizers in our comparison (best result in bold):

	SGD	*Adam*
Fully-connected (567171 params)	85.27%	87.96%
Convolutional (346067 params)	89.59%	**91.78%**