Module creation

Solve the following exercises and upload your solutions to Moodle (unless specified otherwise) until the specified due date. Make sure to use the *exact filenames* that are specified for each individual exercise. Unless explicitly stated otherwise, you can assume correct user input and correct arguments. You are allowed to write additional functions, classes, etc. to improve readability and code quality.

Exercise 1 – Submission: `a4_ex1.py` 40 Points

Write a class `SimpleNetwork` that extends `torch.nn.Module` and represents a basic neural network that is able to handle batches of 1-dimensional samples. This neural network should only use four fully-connected layers: one input, two hidden, and one output layer, whose numbers of neurons are specified upon creation of the network. Implement the following two methods:

- ```
  __init__(
      self,
      input_neurons: int,
      hidden_neurons: int,
      output_neurons: int,
      activation_function: torch.nn.Module = torch.nn.ReLU()
  )
  ```
  - `input_neurons` specifies the number of neurons in the fully-connected input layer of the neural network.
  - `hidden_neurons` specifies the number of neurons in all of the fully-connected hidden layers of the neural network.
  - `output_neurons` specifies the number of output neurons of the last fully-connected layer of the neural network.
  - `activation_function` specifies which non-linear activation function is used after all non-output layers of the network. By default, the network should use the `torch.nn.ReLU` (module) function.
  ```

- `forward(self, x: torch.Tensor) -> torch.Tensor`

  This method is used for forward passes through the neural network. It should work as follows:

  - `x` is a 2D tensor of samples with shape `(N, F)`, where `N` is the minibatch size and `F` is the number of features per sample.
  - In the function, a full forward pass through all four layers and activations should be performed.
  - The function should return the result of the output layer, which should be a 2D tensor of shape `(N, output_neurons)`.

Example program execution:

```
if __name__ == "__main__":
    torch.random.manual_seed(0)
    simple_network = SimpleNetwork(10, 20, 5)
    input = torch.randn(1, 10)
    output = simple_network(input)
    print(output)
```

Example output (might differ due to PyTorch version differences):

```
tensor([[-0.2576, -0.0579, -0.1965,  0.0738, -0.0100]],
       grad_fn=<AddmmBackward0>)
```

Hints:

- Although we will not train the network in this assignment, all trainable layers need to be registered for training.
- Do not apply an activation function to the output of the network.
- You can assume that only vectors of appropriate shape, i.e., `F` = `input_neurons`, are fed into the network.

Exercise 2 – Submission: `a4_ex2.py` 60 Points

Write a class `SimpleCNN` that extends `torch.nn.Module` and represents a simple neural network that utilizes convolutional layers with 2-dimensional kernels. Do not use any fully-connected layers in this network apart from the output layer. Implement the following two methods:

- ```
  __init__(
      self,
      input_channels: int,
      hidden_channels: int,
      num_hidden_layers: int,
      use_batchnormalization: bool,
      num_classes: int,
      kernel_size: int = 3,
      activation_function: torch.nn.Module = torch.nn.ReLU()
  )
  ```
  - `input_channels` specifies the number of channels (e.g. color-channels) in the input to the neural network.
  - `hidden_channels` specifies the number of feature channels that are computed by the convolutional layers.
  - `num_hidden_layers` specifies the number of hidden convolutional layers the network should have.
  - `use_batch_normalization` controls whether 2-dimensional batch normalization is used after every convolutional layer.
  - `num_classes` specifies the number of output neurons of the fully-connected output layer of the neural network.
  - `activation_function` specifies which non-linear activation function is used after all non-output layers of the network. By default, the network should use the `torch.nn.ReLU` (module) function.
  ```

- `forward(self, input_images: torch.Tensor)`

  This method is used for forward passes through the neural network. It should work as follows:

  - `input_images` is a 4D tensor of stacked images with a separate brightness channel. Its shape is `(N, C, H, W)`, where `N` is the minibatch size, `C` is the number of channels (e.g., color channels), and `H` and `W` are the height and width of the images (both `H` and `W` can be assumed to be 64).
  - The function should return the result of the output layer, which should be of shape `(N, output_neurons)`. In all convolutional layers, the inputs should be padded with zeros to keep their height and width from changing.

Example program execution:

```
if __name__ == "__main__":
    torch.random.manual_seed(0)
    network = SimpleCNN(3, 32, 3, True, 10, activation_function=nn.ELU())
    input = torch.randn(1, 3, 64, 64)
    output = network(input)
    print(output)
```

Example output (might differ due to PyTorch version differences):

```
tensor([[ 0.3485, -0.0793, -0.1733, -0.9075,  0.4231, -0.0460, -0.4666, -0.1664,
         -0.0804, -0.6130]], grad_fn=<AddmmBackward0>)
```

Hints:

- Although we will not train the network in this assignment, all trainable layers need to be registered for training.
- To convert the output of the convolutional layers to a suitable shape for the fully-connected output layer, you can, e.g, use `Tensor.view` or `Tensor.flatten`. Remember that the height and width of the input can be assumed to be 64 each, and these remain the same even after all convolutional layers, since you must use padding. This means that the flattened representation has a size of `hidden_channels` $\cdot 64 \cdot 64$, where `hidden_channels` is the number of channels of the last convolutional layer.
- You should make heavy use of PyTorch's modules and functions, do not implement everything yourself.
- For batch normalization, refer to https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html.