Assignment 5: PyTorch Basics

Solve the following exercises and upload your solutions to Moodle by the due date.

Important Information!

Please try to *exactly match the outputs* provided in the examples.

Use the *exact filenames* specified for each exercise (the default suggestions from the heading). Your main code (example prints, etc.) must be guarded by `if __name__ == '__main__':`. Unless explicitly stated otherwise, you can assume correct user input and correct arguments.

You may use **only standard libraries**, plus modules covered in the lecture (including Programming in Python 1).

Exercise 1 – Submission: `a5_ex1.py` 25 Points

Write a function `regression(x: torch.Tensor, y: torch.Tensor, lr: float, num_steps:int, logistic: bool)-> tuple(float, float)` that performs a simple linear/logistic regression from scratch. The function should initialise a trainable weight parameter and a trainable bias parameter and perform the following procedure for `num_steps` iterations.

- Compute the current predictions $\hat{y}$ with the parameters of the regression. If `logistic` is true, then this should involve the sigmoid function (https://en.wikipedia.org/wiki/Sigmoid_function).
- Compute the error between $\hat{y}$ and y. If `logistic` is false, then this should be the mean squared error, if it is true it should be the cross entropy error.
- Compute the gradients for the parameters (you can use .backward() for this)
- Update the parameters of the regression based on the error and the learning rate.
- Zero the gradients of the parameters.
- Every 50 steps, beginning with step 0, print the current loss.

All steps apart from the gradient computation should be done manually, i.e., do not use `torch.nn.MSELoss`, or anything from `torch.optim`. Training should be done with the full batch. x and y are 1-dimensional, thus you can expect inputs of the shape [N,1], where N is the size of the training data.

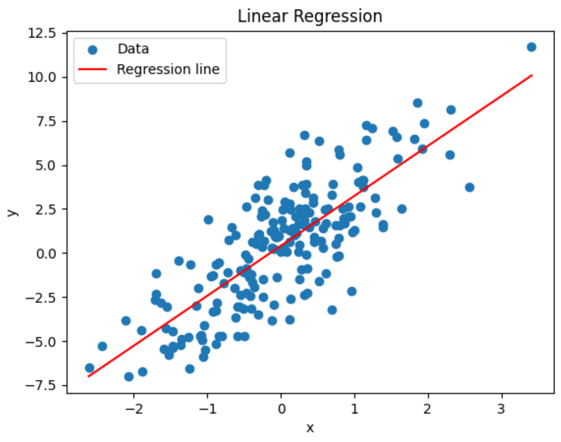
After training, plot the regression in relation to the data and save the figure to the file system. Then return the learned weight and bias.

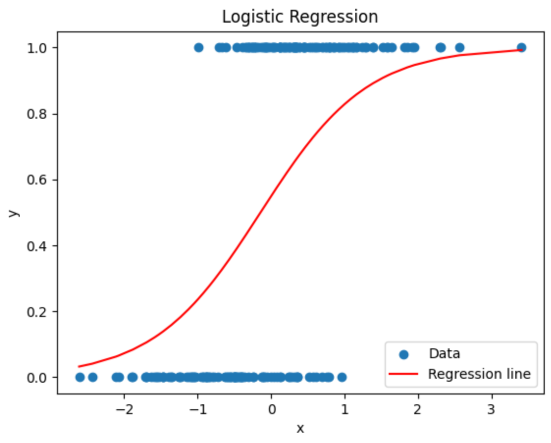
Example code:

```
if __name__ == "__main__":
    lr = 0.01
    num_steps = 300

    # Data
    torch.manual_seed(0)
    N = 200
    x = torch.randn(N, 1)  # Random input data
    noise = 2*torch.randn_like(x)
    y = 3 * x + 0.5 + noise

    w, b = regression(x, y, lr, num_steps, logistic=False)
    print(f"Learned weight: {w:.4f}, Learned bias: {b:.4f}")

    plt.clf()
    y[y > 0.3] = 1
    y[y <= 0.3] = 0

    num_steps = 1000
    w, b = regression(x, y, lr, num_steps, logistic=True)
    print(f"Learned weight: {w:.4f}, Learned bias: {b:.4f}")
```

Example output:

```
Step 0: loss = 12.1586
Step 50: loss = 5.6775
Step 100: loss = 4.6795
Step 150: loss = 4.5258
Step 200: loss = 4.5021
Step 250: loss = 4.4984
Learned weight: 2.8353, Learned bias: 0.4096
Step 0: loss = 0.6931
Step 50: loss = 0.6526
Step 100: loss = 0.6203
Step 150: loss = 0.5943
Step 200: loss = 0.5732
Step 250: loss = 0.5559
Step 300: loss = 0.5415
Step 350: loss = 0.5293
Step 400: loss = 0.5190
Step 450: loss = 0.5102
Step 500: loss = 0.5025
Step 550: loss = 0.4959
Step 600: loss = 0.4900
Step 650: loss = 0.4849
Step 700: loss = 0.4803
Step 750: loss = 0.4763
Step 800: loss = 0.4726
Step 850: loss = 0.4694
Step 900: loss = 0.4664
Step 950: loss = 0.4638
Learned weight: 1.3760, Learned bias: 0.1966
```


Linear Regression
Data
Regression line
x
y


Logistic Regression
Data
Regression line
x
y

Exercise 2 – Submission: `a5_ex2.py` 25 Points

Write a class `SimpleMLP(input_size: int, hidden_size: int, output_size: int)` that implements a basic fully-connected feed-forward neural network and a function `train(model: torch.nn.Module, x: torch.Tensor, y: torch.Tensor, lr: float, num_steps:int)` that trains the model on a regression task.

- The FFNN should have two fully-connected layers created with torch.nn.Linear. The layers should be connected with ReLU activations.
- The input size, hidden size, and output size are given when initialising the model.
- `train` should optimise the model by stochastic gradient descent with the given learning rate and use the mean squared error.
- Every 50 steps, beginning with step 0, print the current loss.

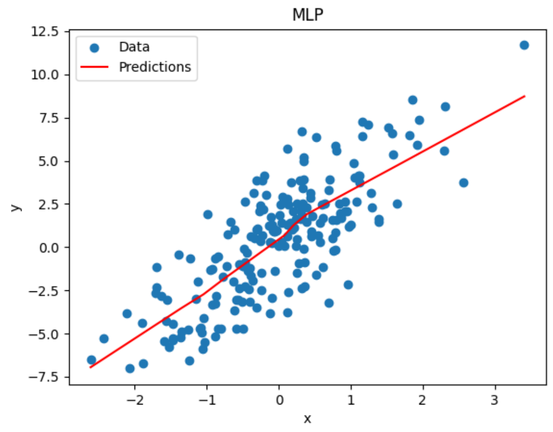
After training the model, plot the predictions and save them under a2.png.

Example code:

```python
if __name__ == "__main__":
    torch.manual_seed(0)

    lr = 0.01
    num_steps = 300
    N = 200

    x = torch.randn(N, 1)
    noise = 2*torch.randn_like(x)
    y = 3 * x + 0.5 + noise

    model = SimpleMLP(input_size=1, hidden_size=10, output_size=1)
    train(model, x, y, lr, num_steps)
    print(f"Learned weights layer 1: {model.fc1.weight.data}")
    print(f"Learned bias layer 1: {model.fc1.bias.data}")
    print(f"Learned weights layer 2: {model.fc2.weight.data}")
    print(f"Learned bias layer 2: {model.fc2.bias.data}")
```

Example output:

```
Step 0: loss = 10.9620
Step 50: loss = 4.7748
Step 100: loss = 4.6562
Step 150: loss = 4.6192
Step 200: loss = 4.5857
Step 250: loss = 4.5554

Learned weights layer 1: tensor([[ 1.3977],
        [ 0.7252],
        [-0.6834],
        [-0.0221],
        [ 0.2624],
        [-0.8336],
        [ 0.4416],
        [-1.0034],
        [-1.3952],
        [-1.3016]])
Learned bias layer 1: tensor([-0.1115,  0.7633, -0.9306,  0.8920, -0.3805,  0.1856, -0.0515,
-0.8091, 0.5421,  0.2626])
Learned weights layer 2: tensor([[ 1.1430,  0.8241, -0.1911,  0.2136,  0.0792, -0.2987,  0.1237,
Learned bias layer 2: tensor([0.3488])
```



Exercise 3 – Submission: `a5_ex3.py` 25 Points

Write a class `SimpleCNN(input_channels:int, hidden_channels_1:int, hidden_channels_2:int, kernel_size:int, num_classes:int, input_width:int, input_height:int)` that implements a basic convolutional neural network and a function `train(model: torch.nn.Module, x: torch.Tensor, y: torch.Tensor, lr: float, num_steps:int)` that trains the model on a classification task.

- The CNN should have two convolutional layers created with torch.nn.Conv2d and a fully-connected output layer. The layers should be connected with ReLU activations.
- The size of all layers is given when initialising the model.
- The convolutional layers should pad the input to preserve the spatial shape.
- `train` should optimise the model by stochastic gradient descent with the given learning rate and use a cross-entropy loss.
- Make sure that the model works with different spatial input dimensions!
- Every 50 steps, beginning with step 0, print the current loss.

Example code:

```python
if __name__ == "__main__":
    torch.manual_seed(0)

    N = 200
    inp_w = inp_h = 28
    x = torch.randn(N, 3, inp_w, inp_h)
    y = torch.randint(0, 10, (N,))

    model = SimpleCNN(
        input_channels=3,
        hidden_channels_1=4,
        hidden_channels_2=8,
        kernel_size=3,
        num_classes=10,
        input_width=inp_w,
        input_height=inp_h
    )
    lr = 0.01
    num_steps = 300
    train(model, x, y, lr, num_steps)
```

Example output:

```
Step 0: loss = 2.3082
Step 50: loss = 1.9787
Step 100: loss = 1.3593
Step 150: loss = 0.4399
Step 200: loss = 0.1284
Step 250: loss = 0.0597
```

Exercise 4 – Submission: `a5_ex4.py` 25 Points

Write a function `get_mnist_loaders(batch_size: int, collate_fn = mnist_collate, root: str = ".") -> tuple[DataLoader, DataLoader]` that returns a DataLoader with the training data of MNIST and a DataLoader with the test data of MNIST, both with the specified batch size and collate function. In addition, write a function `mnist_collate(batch: list) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]` that defines a custom collate function for the dataloader (`https://docs.pytorch.org/docs/stable/data.html`):

- `get_mnist_loaders` should get the MNIST dataset from torchvision.datasets and return appropriate dataloaders with the train and test split of MNIST that use the given collate function.
- `mnist_collate` should standardise the inputs, i.e., subtract the mean and divide by the standard deviation.
- `mnist_collate` should return the standardised inputs, the labels as a 1D Long tensor, and the labels as a 2D one-hot encoding tensor.

Example code:

```
if __name__ == "__main__":
    torch.manual_seed(0)

    train_loader, test_loader = get_mnist_loaders(32)
    x, y, y_onehot = next(iter(train_loader))
    print(x[0,:5,:5])
    print(y[0])
    print(y_onehot[0])
```

Example output:

```
tensor([[[-0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226],
         [-0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226],
         [-0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.3842,  1.8453,  2.3322, -0.2689, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226],
         [-0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.3329,  2.0247,  2.8063, -0.1536, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226],
         [-0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226,  0.3718,  2.6141,  2.8063, -0.1536, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226,
          -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226, -0.4226]]])
tensor(6)
tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.])
```