Practice of Deep Learning
Bonus 3 – Implementing Advanced Concepts in PyTorch

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Introduction (Week 3)

1. How to use **PyTorch** to implement **advanced optimizers**?
2. How to use **PyTorch** to implement **advanced initializers**?
3. How to use **PyTorch** to implement **regularization**?
4. What are the **Dataset** and **Dataloader** objects in **PyTorch**?
5. How to implement a custom **Dataloader** and **Dataset** object in PyTorch?

Using advanced optimizers (Adam, etc.)

Our next step is logically to **replace our vanilla gradient descent with some more advanced optimizers**, e.g. Adam, AdaGrad, RMSProp, etc.

Again, we will be relying on PyTorch as much as possible.

Feel free to have a look at all the available **built-in optimizers**, available in PyTorch, here: https://pytorch.org/docs/stable/optim.html

Let us now demonstrate how to use them in our Neural Network!

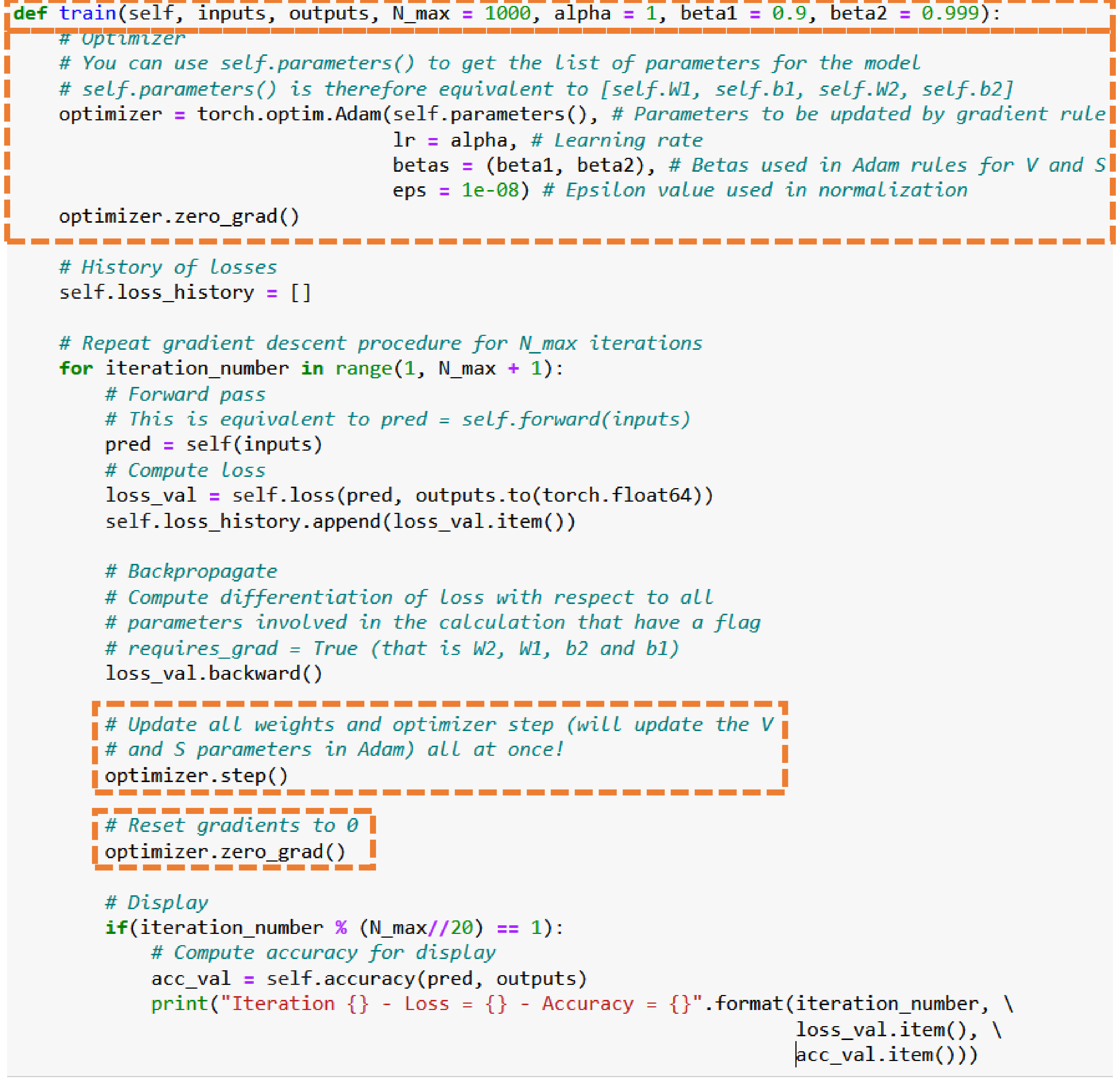
Using advanced optimizers (Adam, etc.)

Three modifications are to be considered in order to use **Adam** optimizer instead of the Vanilla gradient descent.

- **Adam** will be added as an optimizer object and can be passed to the **train()** method. It will simply consist of an additional variable.

optimizer = torch.optim.Adam(self.parameters(), lr = alpha,
betas = (beta1, beta2), eps = 1e-08)

- The **optimizer.step()** is used to update the V and S parameters in Adam. **This also realizes the gradient rule update procedure entirely (damn!).**
- You should remember to **reset gradients in optimizer to 0**, like you would in the parameters tensors. The operation **optimizer.zero_grad()** replaces all four **self.Xx.grad.zero_()** operations we had earlier!



```
def train(self, inputs, outputs, N_max = 1000, alpha = 1, beta1 = 0.9, beta2 = 0.999):
    # Optimizer
    # You can use self.parameters() to get the list of parameters for the model
    # self.parameters() is therefore equivalent to [self.W1, self.b1, self.W2, self.b2]
    optimizer = torch.optim.Adam(self.parameters(), # Parameters to be updated by gradient rule
                                 lr = alpha, # Learning rate
                                 betas = (beta1, beta2), # Betas used in Adam rules for V and S
                                 eps = 1e-08) # Epsilon value used in normalization
    optimizer.zero_grad()

    # History of losses
    self.loss_history = []

    # Repeat gradient descent procedure for N_max iterations
    for iteration_number in range(1, N_max + 1):
        # Forward pass
        # This is equivalent to pred = self.forward(inputs)
        pred = self(inputs)
        # Compute loss
        loss_val = self.loss(pred, outputs.to(torch.float64))
        self.loss_history.append(loss_val.item())

        # Backpropagate
        # Compute differentiation of loss with respect to all
        # parameters involved in the calculation that have a flag
        # requires_grad = True (that is W2, W1, b2 and b1)
        loss_val.backward()

        # Update all weights and optimizer step (will update the V
        # and S parameters in Adam) all at once!
        optimizer.step()

        # Reset gradients to 0
        optimizer.zero_grad()

        # Display
        if(iteration_number % (N_max//20) == 1):
            # Compute accuracy for display
            acc_val = self.accuracy(pred, outputs)
            print("Iteration {} - Loss = {} - Accuracy = {}".format(iteration_number, \
                                                                     loss_val.item(), \
                                                                     acc_val.item()))
```

Using advanced optimizers (Adam, etc.)

Feel free to also have a look at the implementation of these optimizers. (You should be able to recognize some concepts from Week 2!)

These optimizers come with a few more features that might be worth exploring if you are curious (but we will consider them out-of-scope)!

https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam

Note: implementing your own optimizer is generally a "bad" idea (takes lots of work!), but feel free to look at source codes to see what it takes!
https://pytorch.org/docs/stable/_modules/torch/optim/adam.html#Adam

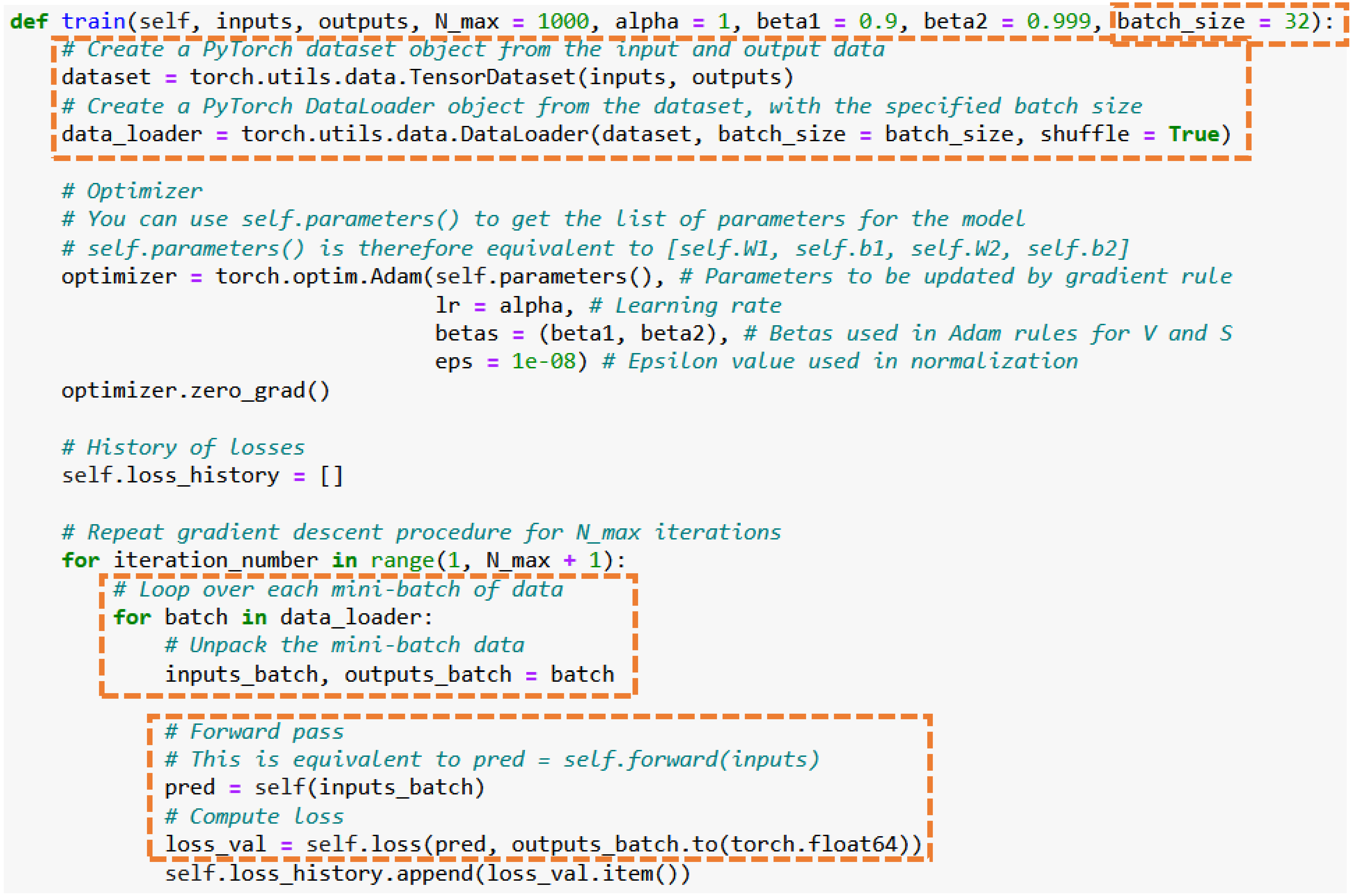
Implementing a stochastic mini-batch GD

We can also define a Stochastic Mini-Batches Gradient Descent procedure. This is typically done with two steps.

1. **Create a Dataset and a Dataloader** using the inputs and outputs provided in the **train()**.

(We will learn more about these Dataset and Dataloader objects in the next notebook and lecture. For now, just consider that it allows to conveniently zip the data in an object that is able to shuffle and draw random mini-batches of data for us.)

2. **Loop over the mini-batches of data**, instead of using the entire inputs/outputs at once, like in (full) batch gradient descent.



```python
def train(self, inputs, outputs, N_max = 1000, alpha = 1, beta1 = 0.9, beta2 = 0.999, batch_size = 32):
    # Create a PyTorch dataset object from the input and output data
    dataset = torch.utils.data.TensorDataset(inputs, outputs)
    # Create a PyTorch DataLoader object from the dataset, with the specified batch size
    data_loader = torch.utils.data.DataLoader(dataset, batch_size = batch_size, shuffle = True)

    # Optimizer
    # You can use self.parameters() to get the list of parameters for the model
    # self.parameters() is therefore equivalent to [self.W1, self.b1, self.W2, self.b2]
    optimizer = torch.optim.Adam(self.parameters(), # Parameters to be updated by gradient rule
                                 lr = alpha, # Learning rate
                                 betas = (beta1, beta2), # Betas used in Adam rules for V and S
                                 eps = 1e-08) # Epsilon value used in normalization
    optimizer.zero_grad()

    # History of losses
    self.loss_history = []

    # Repeat gradient descent procedure for N_max iterations
    for iteration_number in range(1, N_max + 1):
        # Loop over each mini-batch of data
        for batch in data_loader:
            # Unpack the mini-batch data
            inputs_batch, outputs_batch = batch

            # Forward pass
            # This is equivalent to pred = self.forward(inputs)
            pred = self(inputs_batch)
            # Compute loss
            loss_val = self.loss(pred, outputs_batch.to(torch.float64))
            self.loss_history.append(loss_val.item())
```

```python
# Define a neural network structure
n_x = 2
n_h = 10
n_y = 1
np.random.seed(37)
shallow_neural_net_pt = ShallowNeuralNet_PT(n_x, n_h, n_y).to(device)
train_pred = shallow_neural_net_pt.train(train_inputs_pt, train_outputs_pt, N_max = 150, \
                                         alpha = 1, beta1 = 0.9, beta2 = 0.999, batch_size = 32)
```

```
Iteration 1 - Loss = 0.9099342965146197 - Accuracy = 0.8539999723434448
Iteration 8 - Loss = 0.1299862032760467 - Accuracy = 0.906000018119812
Iteration 15 - Loss = 0.07326909912245186 - Accuracy = 0.9509999752044678
Iteration 22 - Loss = 0.016308031142150254 - Accuracy = 0.9559999704360962
Iteration 29 - Loss = 0.029047451953357055 - Accuracy = 0.9340000152587891
Iteration 36 - Loss = 0.003057486766665189 - Accuracy = 0.9430000185966492
Iteration 43 - Loss = 0.004698124321150583 - Accuracy = 0.9470000267028809
Iteration 50 - Loss = 0.0025877519264911903 - Accuracy = 0.9860000014305115
Iteration 57 - Loss = 0.02274350334612207 - Accuracy = 0.9779999852180481
Iteration 64 - Loss = 0.0113862540185388 - Accuracy = 0.9739999771118164
Iteration 71 - Loss = 0.001624697958015368 - Accuracy = 0.9810000061988831
Iteration 78 - Loss = 0.05513155958647538 - Accuracy = 0.9909999966621399
Iteration 85 - Loss = 0.5313654791985021 - Accuracy = 0.9639999866485596
Iteration 92 - Loss = 0.008424374201609155 - Accuracy = 0.9769999980926514
Iteration 99 - Loss = 0.013359109129942799 - Accuracy = 0.9909999966621399
Iteration 106 - Loss = 0.009827488601077239 - Accuracy = 0.9810000061988831
Iteration 113 - Loss = 0.06809013059489735 - Accuracy = 0.9879999756813049
Iteration 120 - Loss = 0.018464219389096287 - Accuracy = 0.9779999852180481
Iteration 127 - Loss = 0.0018662527435126706 - Accuracy = 0.9800000190734863
Iteration 134 - Loss = 0.005060994996136089 - Accuracy = 0.9810000061988831
Iteration 141 - Loss = 0.0011841075081343137 - Accuracy = 0.968999981880188
Iteration 148 - Loss = 0.004148195102886403 - Accuracy = 0.9819999933242798
```

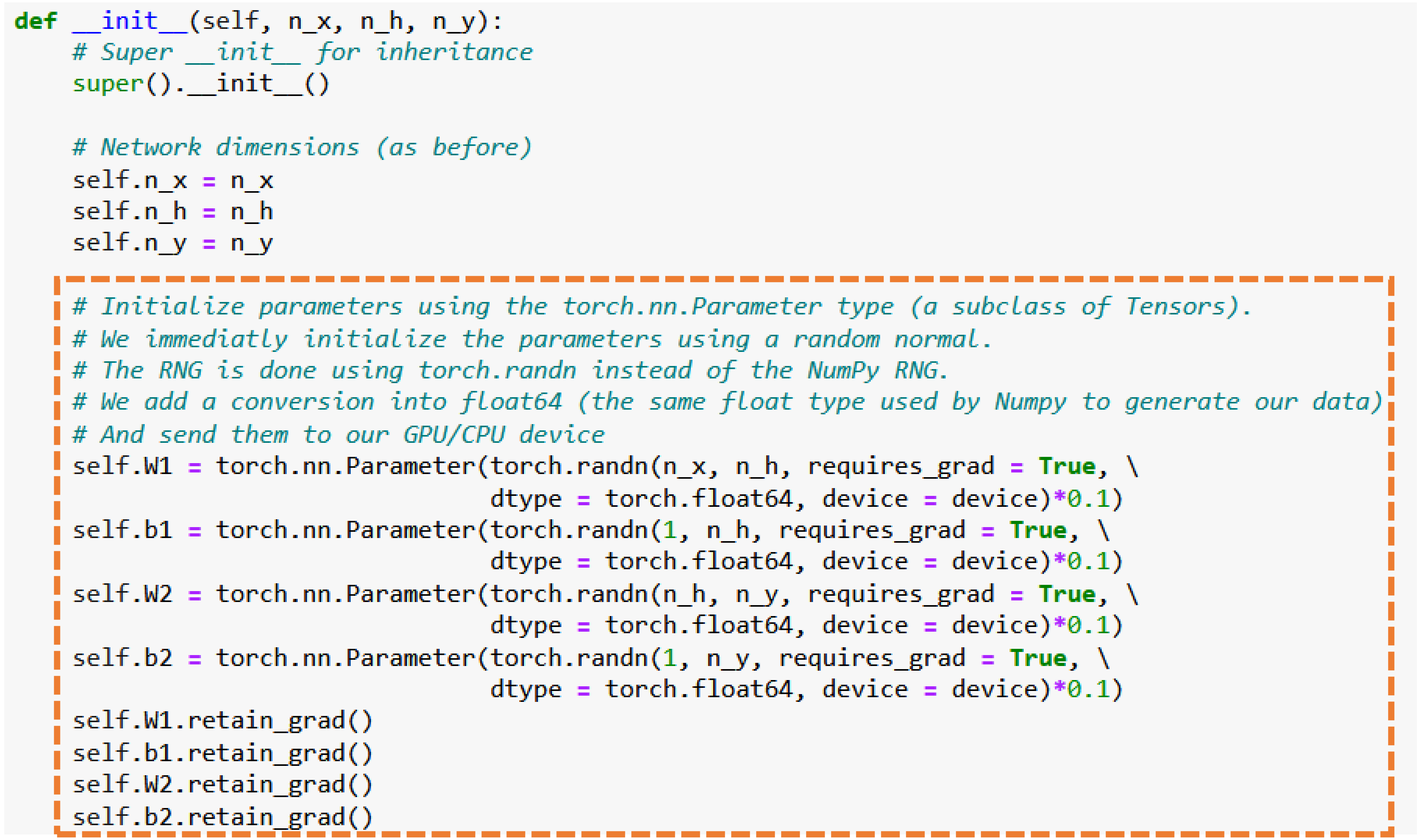
```python
# Check accuracy after training
acc = shallow_neural_net_pt.accuracy(shallow_neural_net_pt(train_inputs_pt), train_outputs_pt).item()
print(acc)
```

```
0.9869999885559082
```

Better initializers in PyTorch

This part is a bit tedious and relies on our own manual implementation of a random normal initializer.

- **Need to replace these RNG with built-in PyTorch initializers!**



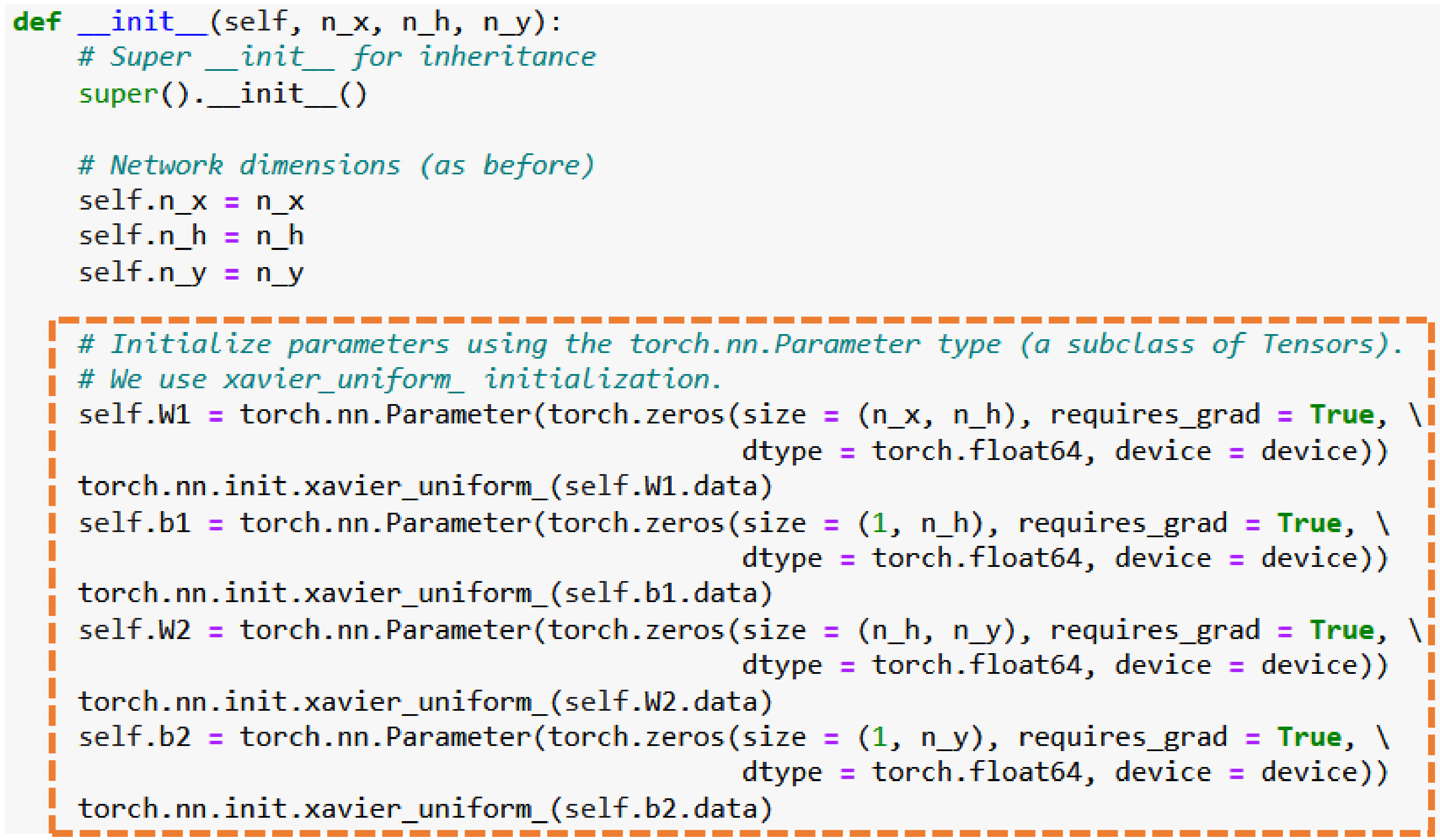
```python
def __init__(self, n_x, n_h, n_y):
    # Super __init__ for inheritance
    super().__init__()

    # Network dimensions (as before)
    self.n_x = n_x
    self.n_h = n_h
    self.n_y = n_y

    # Initialize parameters using the torch.nn.Parameter type (a subclass of Tensors).
    # We immediatly initialize the parameters using a random normal.
    # The RNG is done using torch.randn instead of the NumPy RNG.
    # We add a conversion into float64 (the same float type used by Numpy to generate our data)
    # And send them to our GPU/CPU device
    self.W1 = torch.nn.Parameter(torch.randn(n_x, n_h, requires_grad = True, \
                                             dtype = torch.float64, device = device)*0.1)
    self.b1 = torch.nn.Parameter(torch.randn(1, n_h, requires_grad = True, \
                                             dtype = torch.float64, device = device)*0.1)
    self.W2 = torch.nn.Parameter(torch.randn(n_h, n_y, requires_grad = True, \
                                             dtype = torch.float64, device = device)*0.1)
    self.b2 = torch.nn.Parameter(torch.randn(1, n_y, requires_grad = True, \
                                             dtype = torch.float64, device = device)*0.1)
    self.W1.retain_grad()
    self.b1.retain_grad()
    self.W2.retain_grad()
    self.b2.retain_grad()
```

Better initializers in PyTorch

Fixed!

- Feel free to have a look at this for **additional initializers** in PyTorch.

https://pytorch.org/cppdocs/api/file_torch_csrc_api_include_torch_nn_init.h.html#file-torch-csrc-api-include-torch-nn-init-h

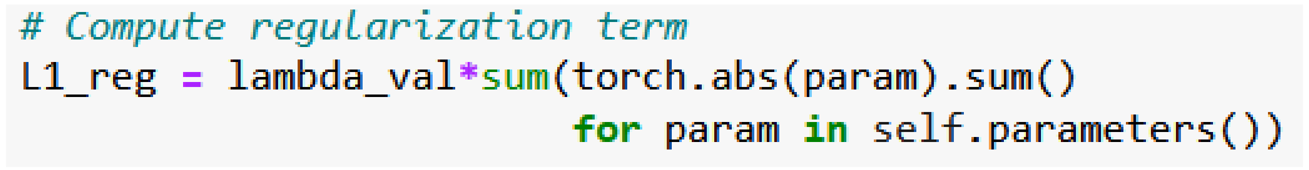


```python
def __init__(self, n_x, n_h, n_y):
    # Super __init__ for inheritance
    super().__init__()

    # Network dimensions (as before)
    self.n_x = n_x
    self.n_h = n_h
    self.n_y = n_y

    # Initialize parameters using the torch.nn.Parameter type (a subclass of Tensors).
    # We use xavier_uniform_ initialization.
    self.W1 = torch.nn.Parameter(torch.zeros(size = (n_x, n_h), requires_grad = True, \
                                            dtype = torch.float64, device = device))
    torch.nn.init.xavier_uniform_(self.W1.data)
    self.b1 = torch.nn.Parameter(torch.zeros(size = (1, n_h), requires_grad = True, \
                                            dtype = torch.float64, device = device))
    torch.nn.init.xavier_uniform_(self.b1.data)
    self.W2 = torch.nn.Parameter(torch.zeros(size = (n_h, n_y), requires_grad = True, \
                                            dtype = torch.float64, device = device))
    torch.nn.init.xavier_uniform_(self.W2.data)
    self.b2 = torch.nn.Parameter(torch.zeros(size = (1, n_y), requires_grad = True, \
                                            dtype = torch.float64, device = device))
    torch.nn.init.xavier_uniform_(self.b2.data)
```

Adding a regularization to our loss

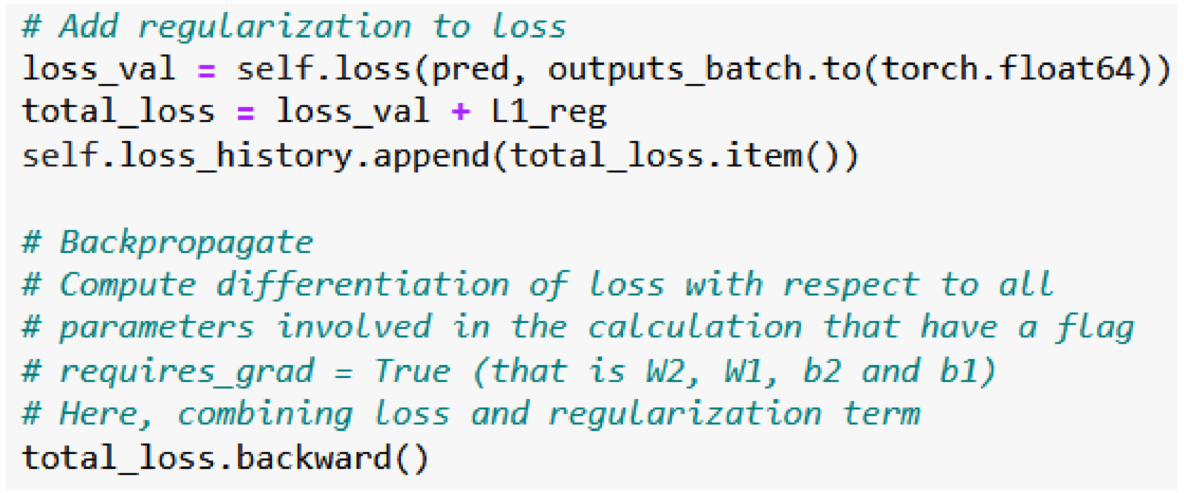
We have also seen during **Ridge Regression** (on Week 1) that it is sometimes beneficial to add a **regularization** term to our loss functions.

- Our first step would be to simply compute our **regularization** term by using the PyTorch functions, for instance the **L1 loss**.



```
# Compute regularization term
L1_reg = lambda_val*sum(torch.abs(param).sum()
                        for param in self.parameters())
```

- We would then simply **add it to the loss before backpropagating**.



```
# Add regularization to loss
loss_val = self.loss(pred, outputs_batch.to(torch.float64))
total_loss = loss_val + L1_reg
self.loss_history.append(total_loss.item())

# Backpropagate
# Compute differentiation of loss with respect to all
# parameters involved in the calculation that have a flag
# requires_grad = True (that is W2, W1, b2 and b1)
# Here, combining loss and regularization term
total_loss.backward()
```