50.039 Theory and Practice of Deep Learning

W3-2 Introduction to Deep Learning using the PyTorch framework

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Introduction (Week 3)

1. What is the **PyTorch library** and its **benefits**?
2. What is a **PyTorch tensor object** and its typical **attributes**?
3. How to implement some typical **tensor operations**?
4. What is **broadcasting** on tensors?
5. What are **tensor locations** in terms of computation?
6. How to **transform our original NumPy shallow Neural Network** class so it uses **PyTorch** now instead?
7. How to implement a **forward**, **loss** and **accuracy** metric in PyTorch?
8. What are some measurable **performance benefits** of using **PyTorch** over NumPy and **GPUs** over CPUs?

Introduction (Week 3)

9. What is the **autograd/backprop** module in PyTorch, and how does it use a **computational graph** to **compute all derivatives**?
10. How to use the **autograd** to implement **derivatives** and a **vanilla gradient descent**?
11. How to implement **backprop** in PyTorch for our **shallow Neural Network** class?
12. How to use **PyTorch** to implement **advanced optimizers**?
13. How to use **PyTorch** to implement **advanced initializers**?
14. How to use **PyTorch** to implement **regularization**?
15. How to finally revise our **trainer** function to obtain a minimal, yet complete Neural Network in PyTorch?

Introduction (Week 3)

16. What are the **Dataset** and **Dataloader** objects in **PyTorch**?
17. How to implement a custom **Dataloader** and **Dataset** object in PyTorch?
18. How to move from binary classification to **multi-class classification**?
19. How to adjust output probabilities using the **softmax** function?
20. How to change the **cross-entropy loss** so it works in **multi-class classification**?
21. How to implement **building blocks** in PyTorch?
22. How to implement and train our first **Deep Neural Network**?
23. What are **additional good practices** in PyTorch?

```python
class ShallowNeuralNet_PT(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y, device):
        super().__init__()
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        self.device = 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()

    def forward(self, inputs):
        A1 = torch.sigmoid(torch.matmul(inputs, self.W1) + self.b1)
        return torch.sigmoid(torch.matmul(A1, self.W2) + self.b2)

    def CE_loss(self, pred, outputs):
        eps = 1e-10
        losses = outputs * torch.log(pred + eps) + (1 - outputs) * torch.log(1 - pred + eps)
        return -torch.sum(losses)/outputs.shape[0]

    def accuracy(self, pred, outputs):
        return ((pred >= 0.5).int() == outputs).float().mean()
```

Autograd, the PyTorch beast

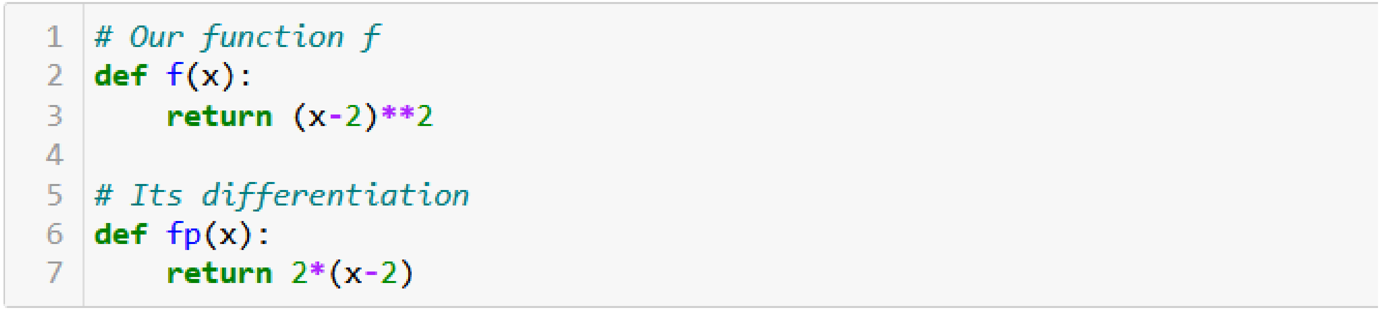
A feature that makes PyTorch extremely powerful is the **autograd**, a **computational graph able to provide automatic differentiation for all operations made on Tensors**.

- To demonstrate, let us consider the function $f(x) = (x-2)^2$.
- Let us compute $\frac{\partial f}{dx}$ and then compute $f'(1)$.
- The **automatic differentiation** is done by PyTorch, for any operation involving a tensor, that has been tracked by setting its attribute **requires_grad** to **True**.
- Any operation can then be tracked back by using the **backward()** method on the resulting variable (here, $y = f(1)$).

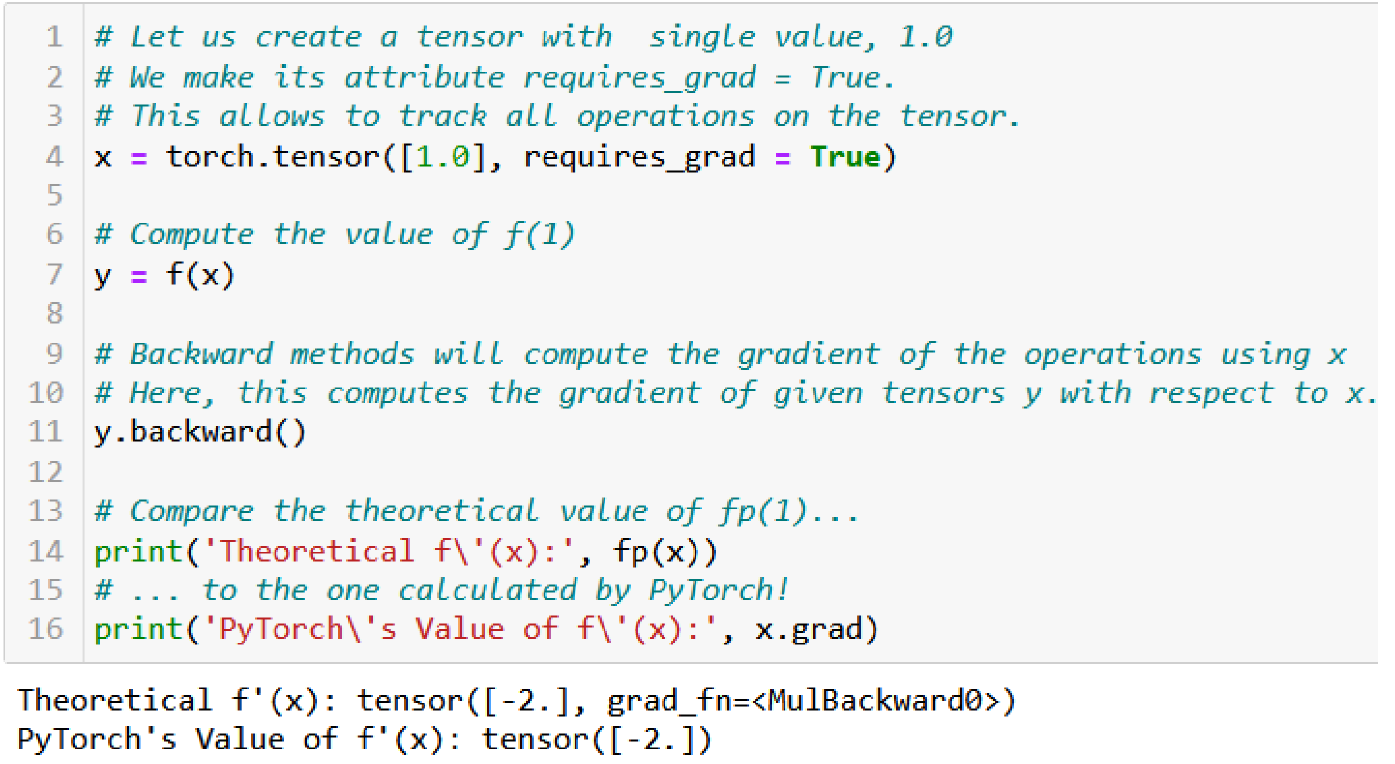
Autograd in action

Define our function and its derivative (for reference).

- Create tensor x, containing a single scalar with value 1.
- Requires/Retain grad is set to True for tensor x.
- Compute $y = f(x = 1)$.
- Use backward() to ask PyTorch to compute $\frac{\partial y}{\partial x} = f'(x = 1)$.
- Derivative value stored in **x.grad**.



```python
# Our function f
def f(x):
    return (x-2)**2

# Its differentiation
def fp(x):
    return 2*(x-2)
```



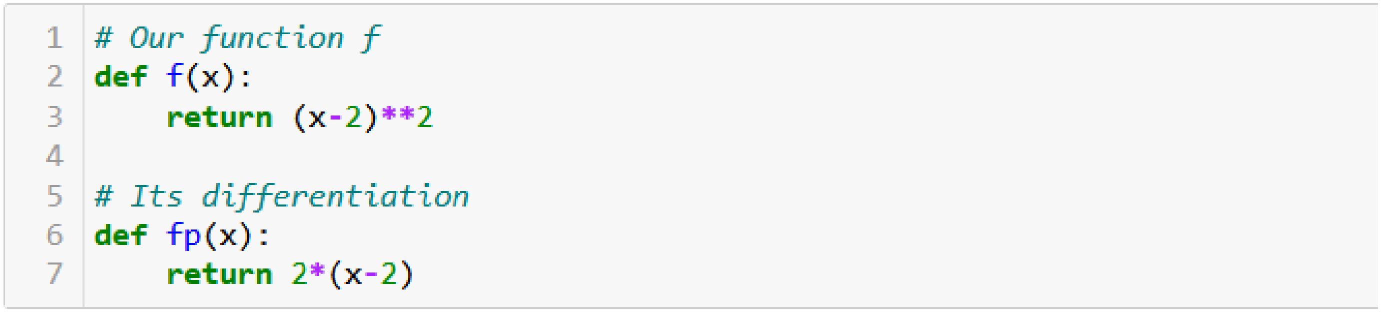
```python
# Let us create a tensor with  single value, 1.0
# We make its attribute requires_grad = True.
# This allows to track all operations on the tensor.
x = torch.tensor([1.0], requires_grad = True)

# Compute the value of f(1)
y = f(x)

# Backward methods will compute the gradient of the operations using x
# Here, this computes the gradient of given tensors y with respect to x.
y.backward()

# Compare the theoretical value of fp(1)...
print('Theoretical f\'(x):', fp(x))
# ... to the one calculated by PyTorch!
print('PyTorch\'s Value of f\'(x):', x.grad)
```

```
Theoretical f'(x): tensor([-2.], grad_fn=<MulBackward0>)
PyTorch's Value of f'(x): tensor([-2.])
```

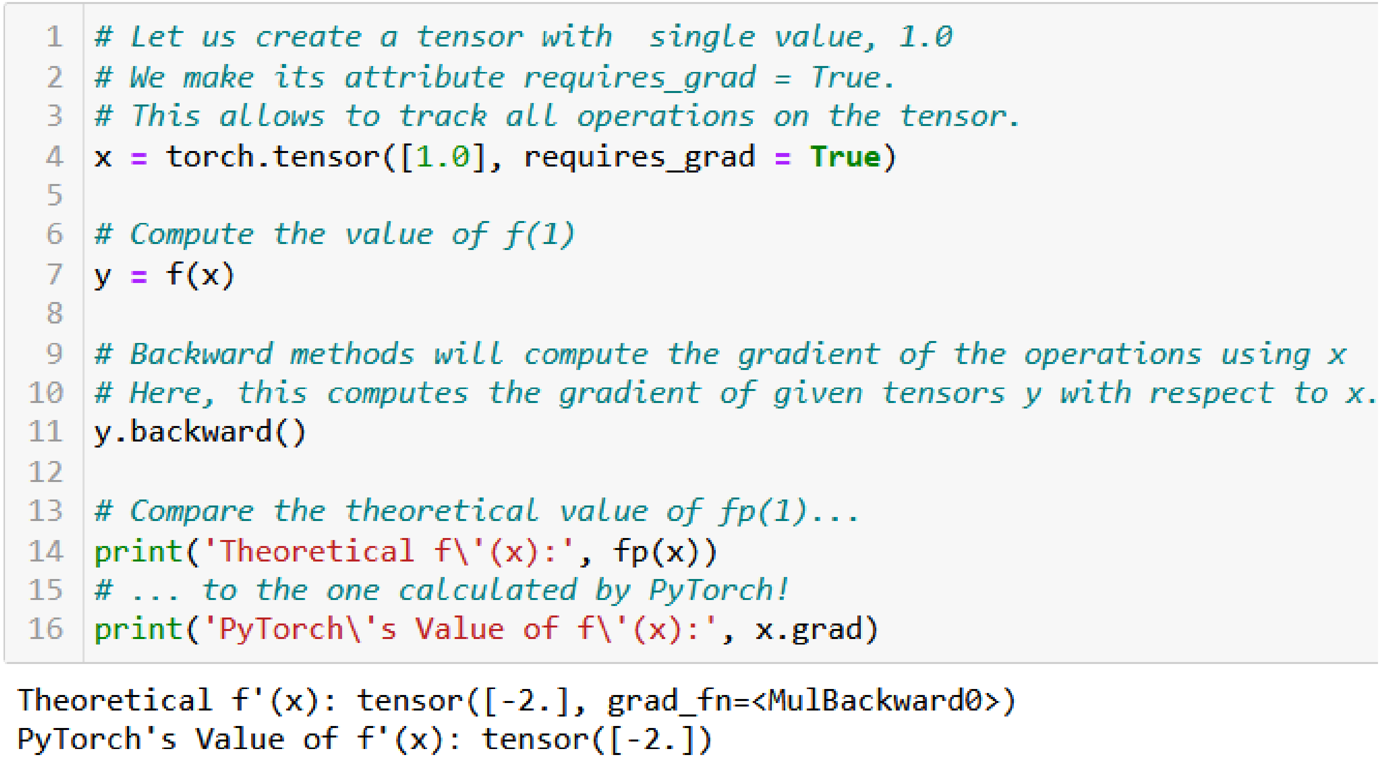
Autograd in action

Whenever backward() is used:

- Autograd will look at the computational graph,
- And it will compute the partial derivatives of the variable y on which backward() is applied,
- With respect to all the tensors x whose requires_grad parameter was set to True,
- And store these derivatives wrt. to each tensor x, in said tensor x, under the grad attribute.



```python
# Our function f
def f(x):
    return (x-2)**2

# Its differentiation
def fp(x):
    return 2*(x-2)
```



```python
# Let us create a tensor with  single value, 1.0
# We make its attribute requires_grad = True.
# This allows to track all operations on the tensor.
x = torch.tensor([1.0], requires_grad = True)

# Compute the value of f(1)
y = f(x)

# Backward methods will compute the gradient of the operations using x
# Here, this computes the gradient of given tensors y with respect to x.
y.backward()

# Compare the theoretical value of fp(1)...
print('Theoretical f\'(x):', fp(x))
# ... to the one calculated by PyTorch!
print('PyTorch\'s Value of f\'(x):', x.grad)
```

```
Theoretical f'(x): tensor([-2.], grad_fn=<MulBackward0>)
PyTorch's Value of f'(x): tensor([-2.])
```

Autograd in action

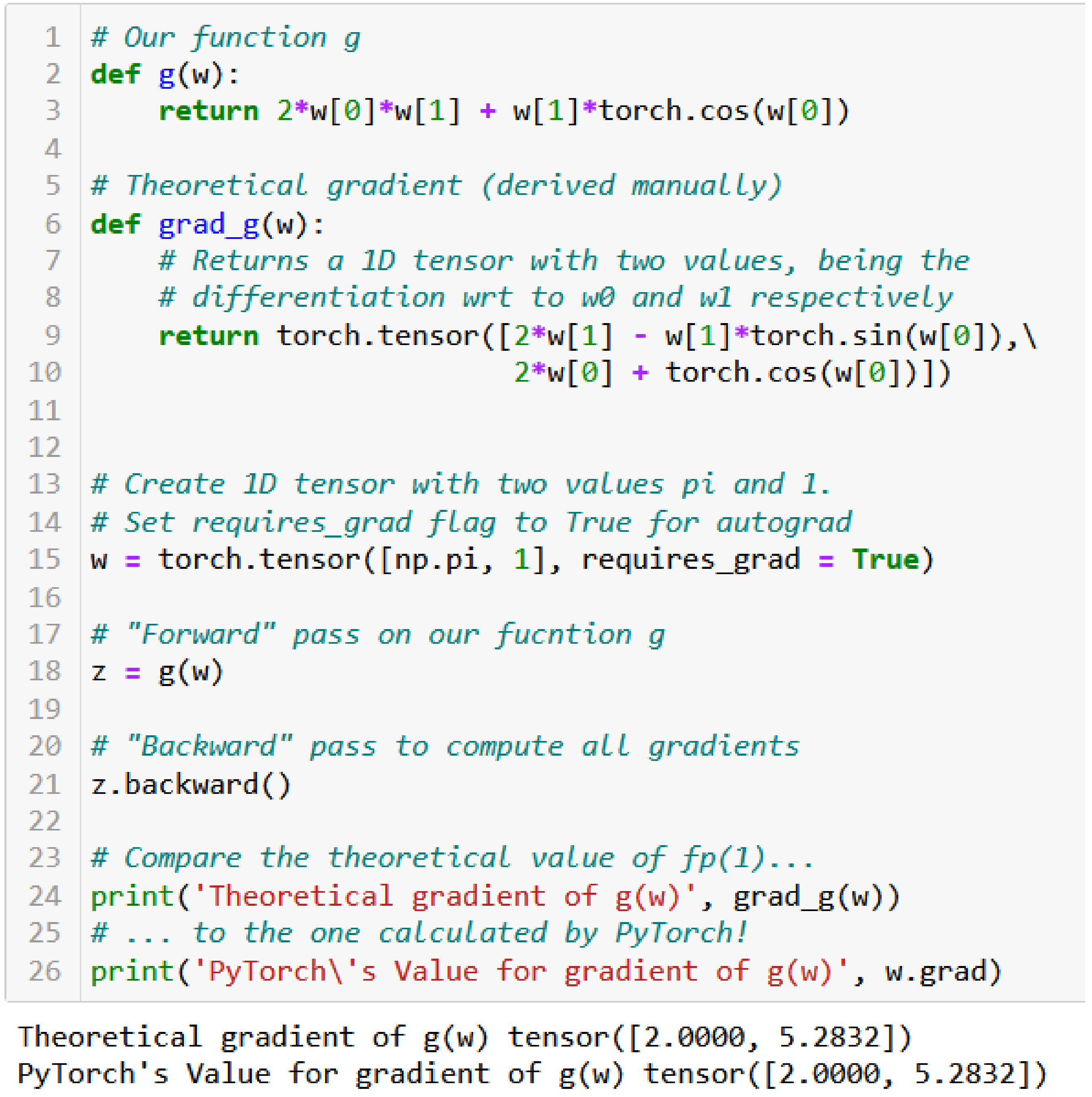
Similarly,

- It can also find gradients of functions of multiple variables!
- For instance, let us define a 1D tensor $w = [w_0, w_1]^T$.
- And function

$$g(w) = 2w_0w_1 + w_1\cos(w_0).$$

- We can use the autograd to compute $\nabla_w\, g(w)$ and find that

$$\nabla_w\, g(\pi, 1) = (2, 2\pi - 1)$$



```python
# Our function g
def g(w):
    return 2*w[0]*w[1] + w[1]*torch.cos(w[0])

# Theoretical gradient (derived manually)
def grad_g(w):
    # Returns a 1D tensor with two values, being the
    # differentiation wrt to w0 and w1 respectively
    return torch.tensor([2*w[1] - w[1]*torch.sin(w[0]),\
                         2*w[0] + torch.cos(w[0])])


# Create 1D tensor with two values pi and 1.
# Set requires_grad flag to True for autograd
w = torch.tensor([np.pi, 1], requires_grad = True)

# "Forward" pass on our fucntion g
z = g(w)

# "Backward" pass to compute all gradients
z.backward()

# Compare the theoretical value of fp(1)...
print('Theoretical gradient of g(w)', grad_g(w))
# ... to the one calculated by PyTorch!
print('PyTorch\'s Value for gradient of g(w)', w.grad)
```

```
Theoretical gradient of g(w) tensor([2.0000, 5.2832])
PyTorch's Value for gradient of g(w) tensor([2.0000, 5.2832])
```

Autograd, in short

The **autograd** is one of the most powerful features of the PyTorch framework, and why we like it so much for training Neural Networks.

- **Thanks to autograd, we do not need to calculate the gradients for any of our future gradient descent rules, manually again! (yay!)**
- The counterpart however is that all operation must involve tensors and must use PyTorch functions and methods... (but that is ok).
- You do not need to know how it implements the computation of all gradients in the background, but if you are curious, have a look at: https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/.

Implementing gradient descent with autograd

Now that we have an automated way to compute gradients, we can use gradient descent to find the (local) minima of any differentiable function. To demonstrate, we use $f(x) = (x-2)^2$.

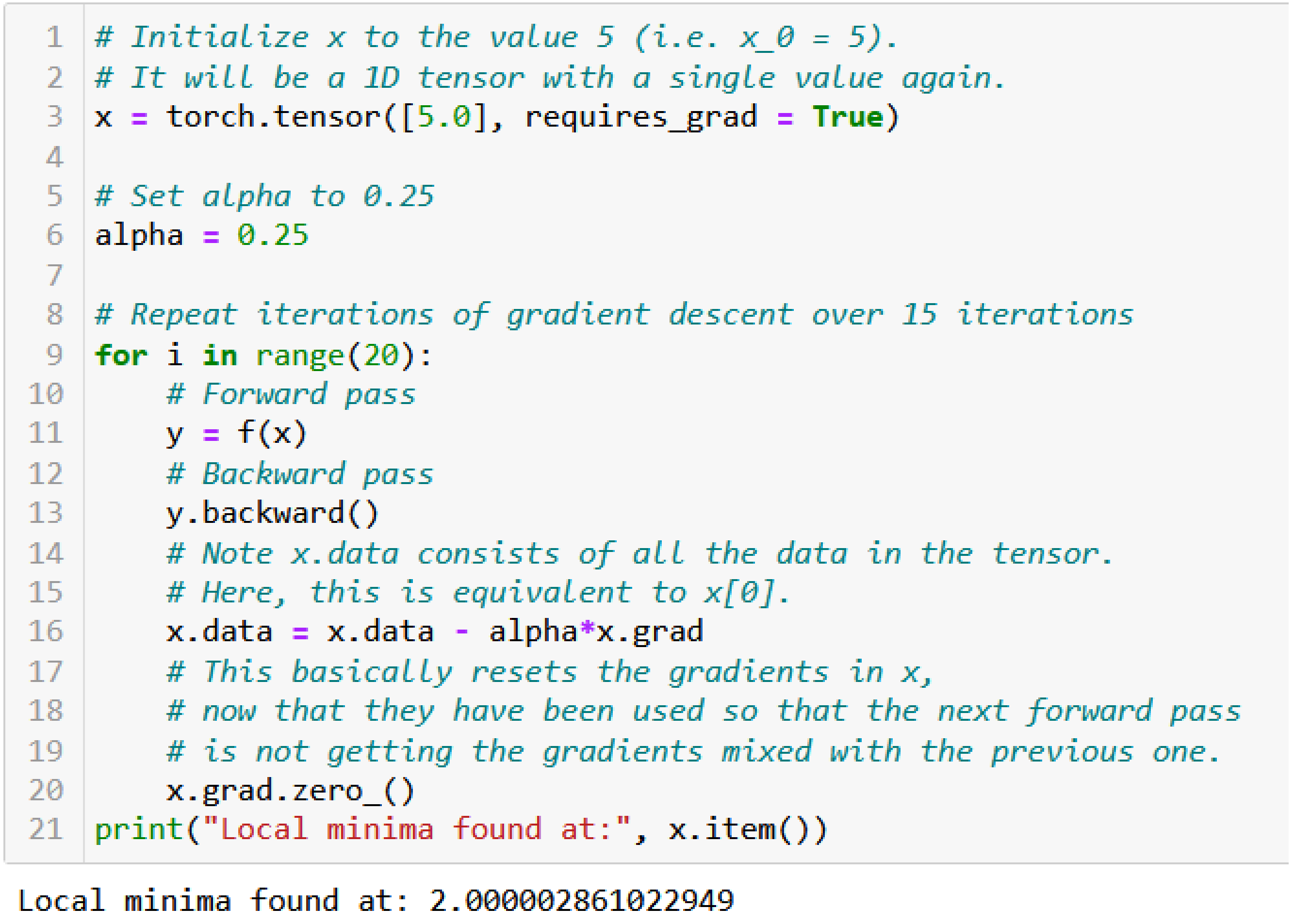
We find the minimum by using the **gradient descent algorithm**:

- **Define a tensor with retained gradients and starting value $\boldsymbol{x_0}$, learning rate $\boldsymbol{\alpha}$**, and set iteration number t to 0.
- **Forward pass:** compute $f(x_t)$
- **Backward pass:** Using **autograd,** compute $x_{t+1} = x_t - \alpha f'(x_t)$.
- Increment t by 1, and **repeat forward-backward passes** for a given number of iterations (or until convergence is observed).

Implementing gradient descent with autograd

<u>Gradient descent algorithm steps:</u>

- **Define a tensor with retained gradients and starting value x_0, learning rate α,** and set iteration number t to 0.
- **Forward pass:** compute $f(x_t)$
- **Backward pass:** Using **autograd,** compute $x_{t+1} = x_t - \alpha f'(x_t)$.
- Increment t by 1 and **repeat forward-backward passes** for a given number of iterations (or until convergence is observed).



```python
# Initialize x to the value 5 (i.e. x_0 = 5).
# It will be a 1D tensor with a single value again.
x = torch.tensor([5.0], requires_grad = True)

# Set alpha to 0.25
alpha = 0.25

# Repeat iterations of gradient descent over 15 iterations
for i in range(20):
    # Forward pass
    y = f(x)
    # Backward pass
    y.backward()
    # Note x.data consists of all the data in the tensor.
    # Here, this is equivalent to x[0].
    x.data = x.data - alpha*x.grad
    # This basically resets the gradients in x,
    # now that they have been used so that the next forward pass
    # is not getting the gradients mixed with the previous one.
    x.grad.zero_()
print("Local minima found at:", x.item())
```

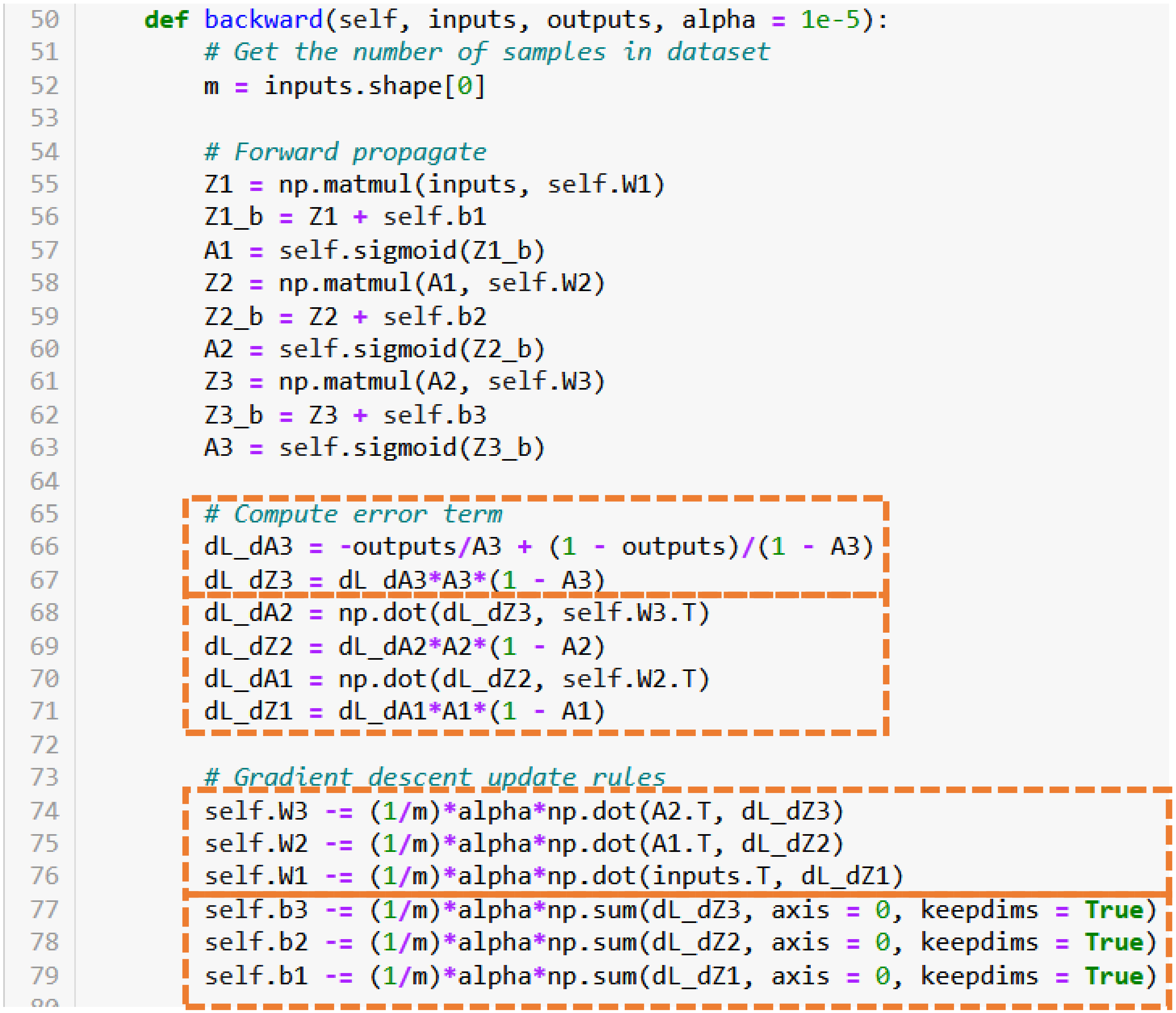
```
Local minima found at: 2.000002861022949
```

Intuition for our NN

Recall Week 2 Notebook 8, our three-layers model.

- We can **recognize patterns** in the way to calculate derivatives and gradient descent update rules.
- **Hints that there should be a way to automate differentiation!**

Practice: automate the gradient descent rule process for any number N of layers?



```
    def backward(self, inputs, outputs, alpha = 1e-5):
        # Get the number of samples in dataset
        m = inputs.shape[0]

        # Forward propagate
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        A1 = self.sigmoid(Z1_b)
        Z2 = np.matmul(A1, self.W2)
        Z2_b = Z2 + self.b2
        A2 = self.sigmoid(Z2_b)
        Z3 = np.matmul(A2, self.W3)
        Z3_b = Z3 + self.b3
        A3 = self.sigmoid(Z3_b)

        # Compute error term
        dL_dA3 = -outputs/A3 + (1 - outputs)/(1 - A3)
        dL_dZ3 = dL_dA3*A3*(1 - A3)
        dL_dA2 = np.dot(dL_dZ3, self.W3.T)
        dL_dZ2 = dL_dA2*A2*(1 - A2)
        dL_dA1 = np.dot(dL_dZ2, self.W2.T)
        dL_dZ1 = dL_dA1*A1*(1 - A1)

        # Gradient descent update rules
        self.W3 -= (1/m)*alpha*np.dot(A2.T, dL_dZ3)
        self.W2 -= (1/m)*alpha*np.dot(A1.T, dL_dZ2)
        self.W1 -= (1/m)*alpha*np.dot(inputs.T, dL_dZ1)
        self.b3 -= (1/m)*alpha*np.sum(dL_dZ3, axis = 0, keepdims = True)
        self.b2 -= (1/m)*alpha*np.sum(dL_dZ2, axis = 0, keepdims = True)
        self.b1 -= (1/m)*alpha*np.sum(dL_dZ1, axis = 0, keepdims = True)
```

Rewriting our backpropagation

In order to train our Neural Network using backprop, we rely on autograd to compute the gradient updates for us automatically.

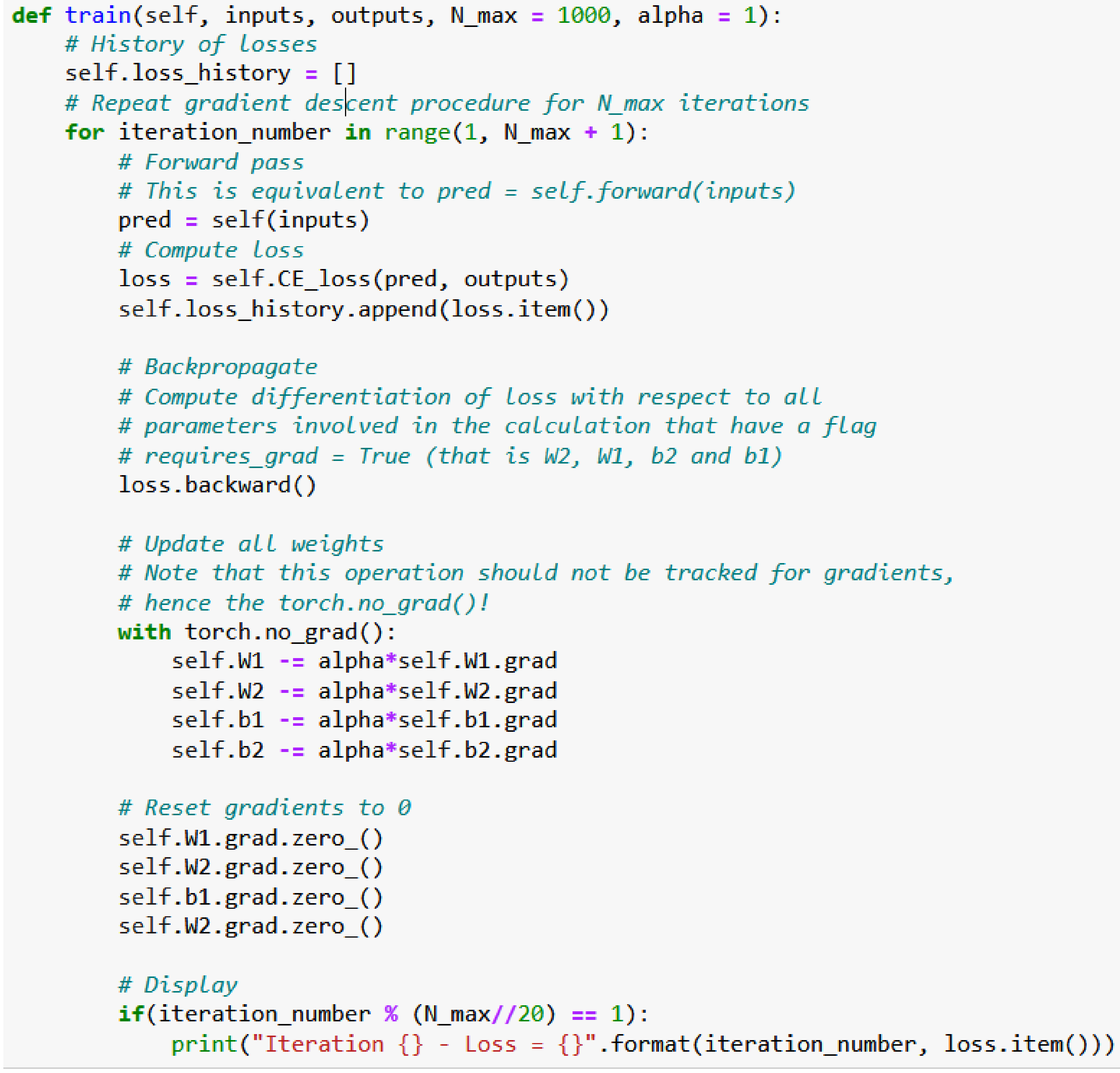
For this reason,

- There is no need for a **backward()** method as before to compute gradients and gradient descent update rules (built-in method for nn.Module classes, covered automatically by PyTorch autograd, yay!).
- Our **train()** method will now consist of several iterations of
  - **forward()** pass and loss calculation,
  - gradient computation with the **backward()** method, used on loss,
  - and **gradient descent updates** on trainable parameters.

Backpropagation

Our **train()** method will be rewritten and now consists of several iterations of

- **forward()** pass and loss calculation,
- gradient computation with the **backward()** method, used on loss,
- and **gradient descent updates** on trainable parameters.



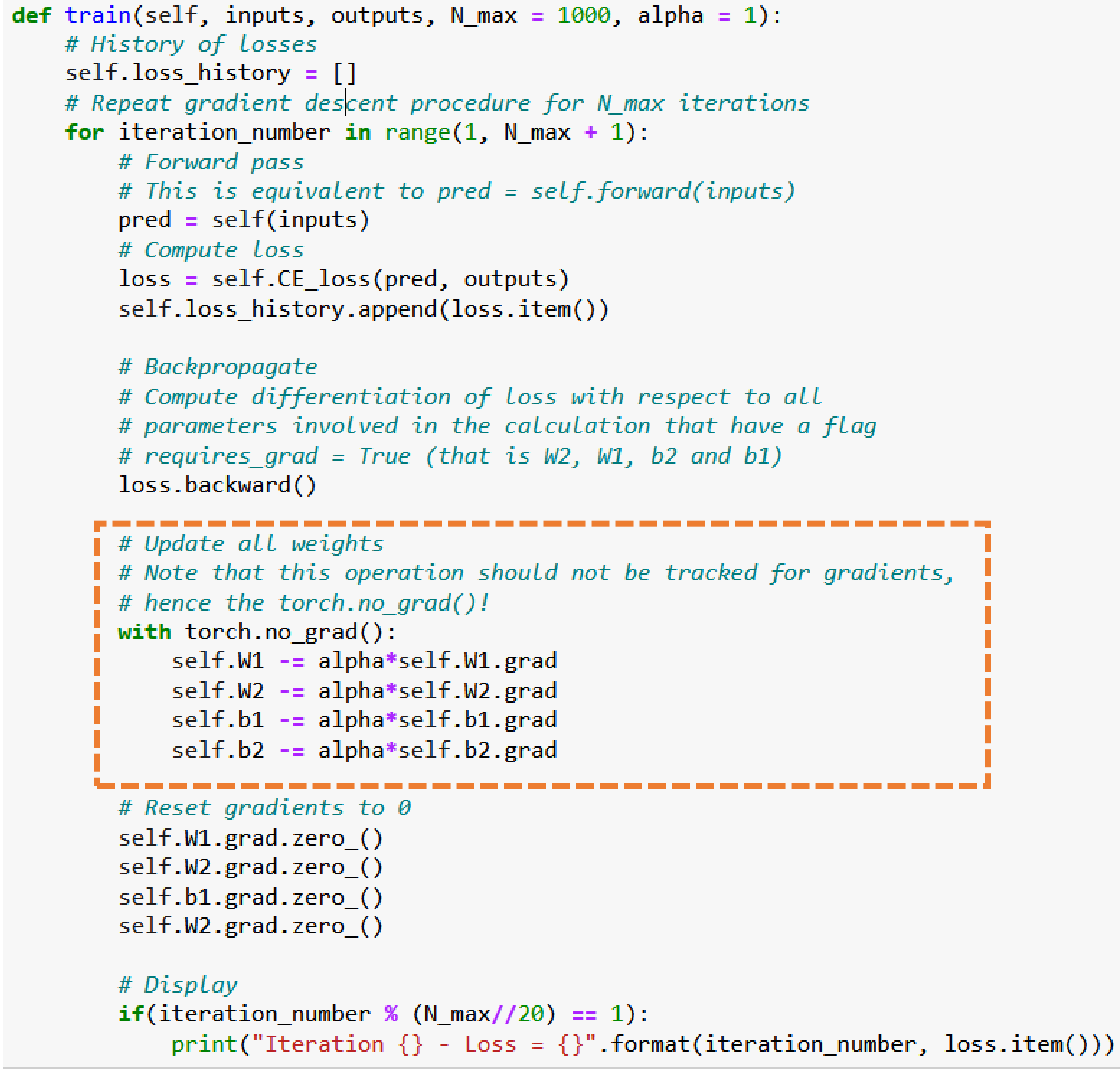
```python
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # History of losses
    self.loss_history = []
    # Repeat gradient descent procedure for N_max iterations
    for iteration_number in range(1, N_max + 1):
        # Forward pass
        # This is equivalent to pred = self.forward(inputs)
        pred = self(inputs)
        # Compute loss
        loss = self.CE_loss(pred, outputs)
        self.loss_history.append(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)
        loss.backward()

        # Update all weights
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # Display
        if(iteration_number % (N_max//20) == 1):
            print("Iteration {} - Loss = {}".format(iteration_number, loss.item()))
```

Backpropagation

Important note:

Using **with torch.no_grad():** tells PyTorch that the operations should not be tracked by the autograd.

Indeed, when performing gradient update for a trainable parameter, we do not want to update gradients for other trainable parameters!



```python
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # 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 = self.CE_loss(pred, outputs)
        self.loss_history.append(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)
        loss.backward()

        # Update all weights
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # Display
        if(iteration_number % (N_max//20) == 1):
            print("Iteration {} - Loss = {}".format(iteration_number, loss.item()))
```

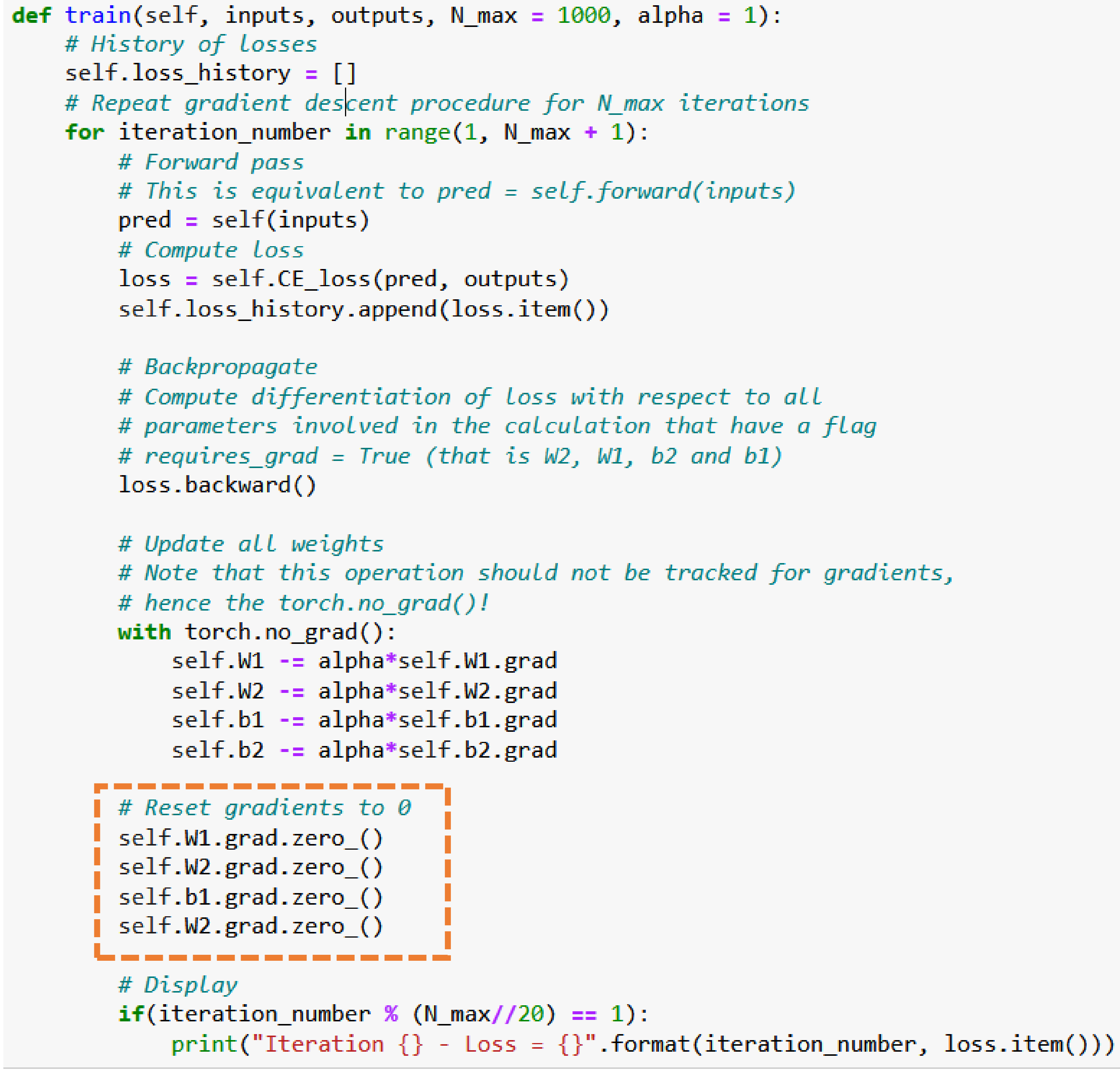
Backpropagation

Important note #2:

Also, do not forget to **reset your gradients to zero** before the next iteration!

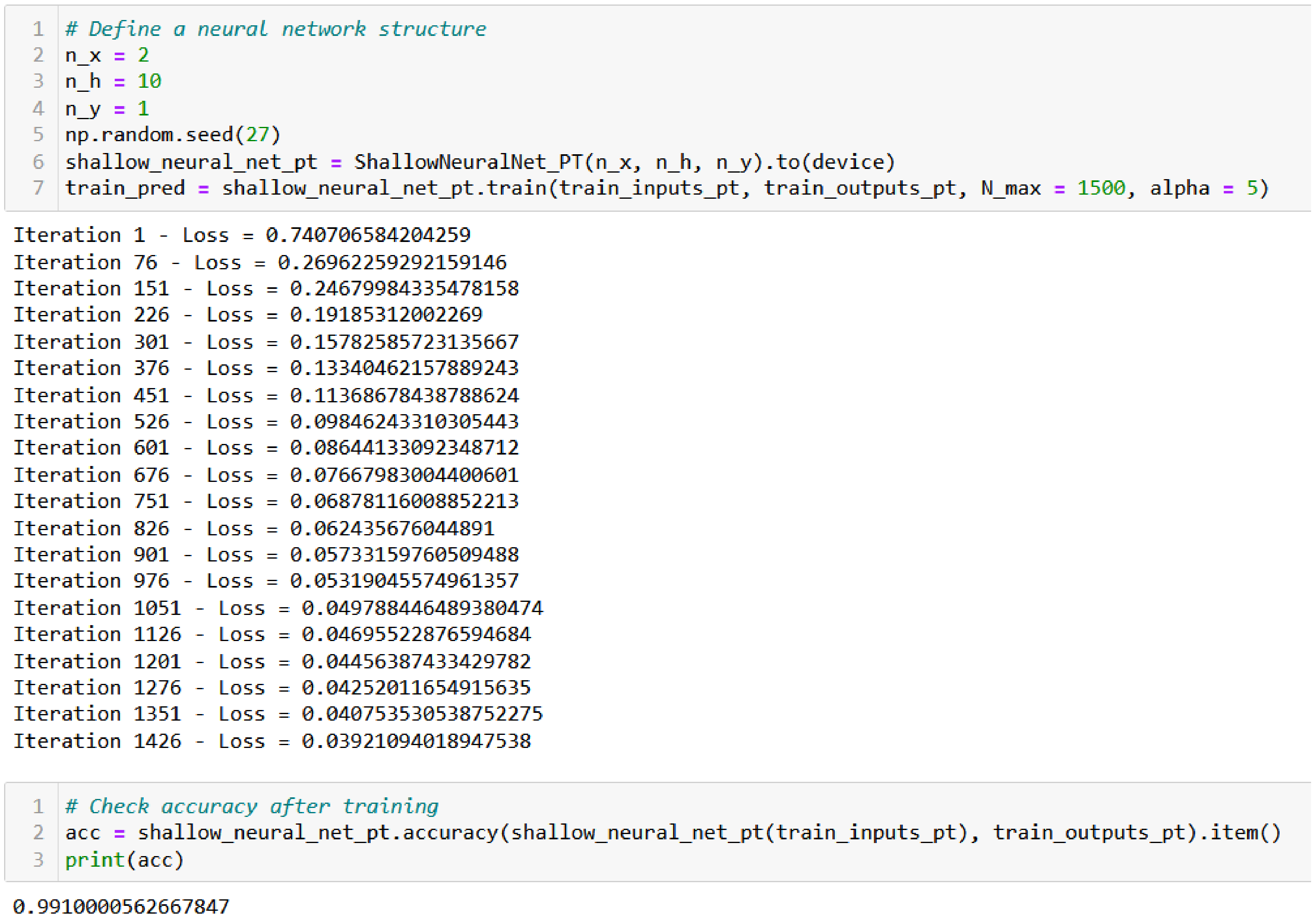
Otherwise, they will accumulate (and we do not want that!)

This can be simply done with **grad.zero_().**



```
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # 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 = self.CE_loss(pred, outputs)
        self.loss_history.append(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)
        loss.backward()

        # Update all weights
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # Display
        if(iteration_number % (N_max//20) == 1):
            print("Iteration {} - Loss = {}".format(iteration_number, loss.item()))
```

Trying our trainer function... Works!



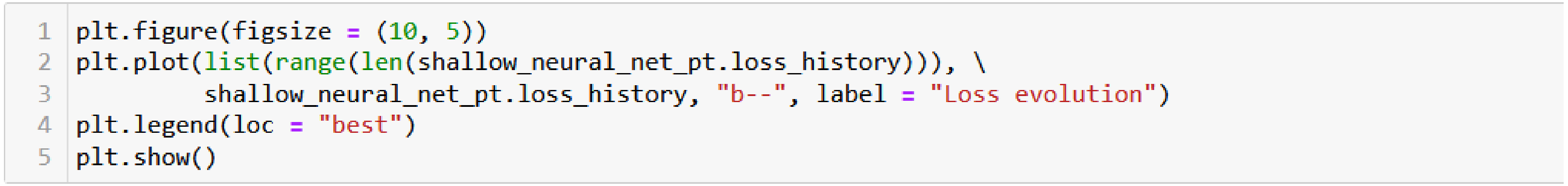
```python
# Define a neural network structure
n_x = 2
n_h = 10
n_y = 1
np.random.seed(27)
shallow_neural_net_pt = ShallowNeuralNet_PT(n_x, n_h, n_y).to(device)
train_pred = shallow_neural_net_pt.train(train_inputs_pt, train_outputs_pt, N_max = 1500, alpha = 5)
```

```
Iteration 1 - Loss = 0.740706584204259
Iteration 76 - Loss = 0.26962259292159146
Iteration 151 - Loss = 0.24679984335478158
Iteration 226 - Loss = 0.19185312002269
Iteration 301 - Loss = 0.15782585723135667
Iteration 376 - Loss = 0.13340462157889243
Iteration 451 - Loss = 0.11368678438788624
Iteration 526 - Loss = 0.09846243310305443
Iteration 601 - Loss = 0.08644133092348712
Iteration 676 - Loss = 0.07667983004400601
Iteration 751 - Loss = 0.06878116008852213
Iteration 826 - Loss = 0.062435676044891
Iteration 901 - Loss = 0.05733159760509488
Iteration 976 - Loss = 0.05319045574961357
Iteration 1051 - Loss = 0.049788446489380474
Iteration 1126 - Loss = 0.04695522876594684
Iteration 1201 - Loss = 0.04456387433429782
Iteration 1276 - Loss = 0.04252011654915635
Iteration 1351 - Loss = 0.040753530538752275
Iteration 1426 - Loss = 0.03921094018947538
```

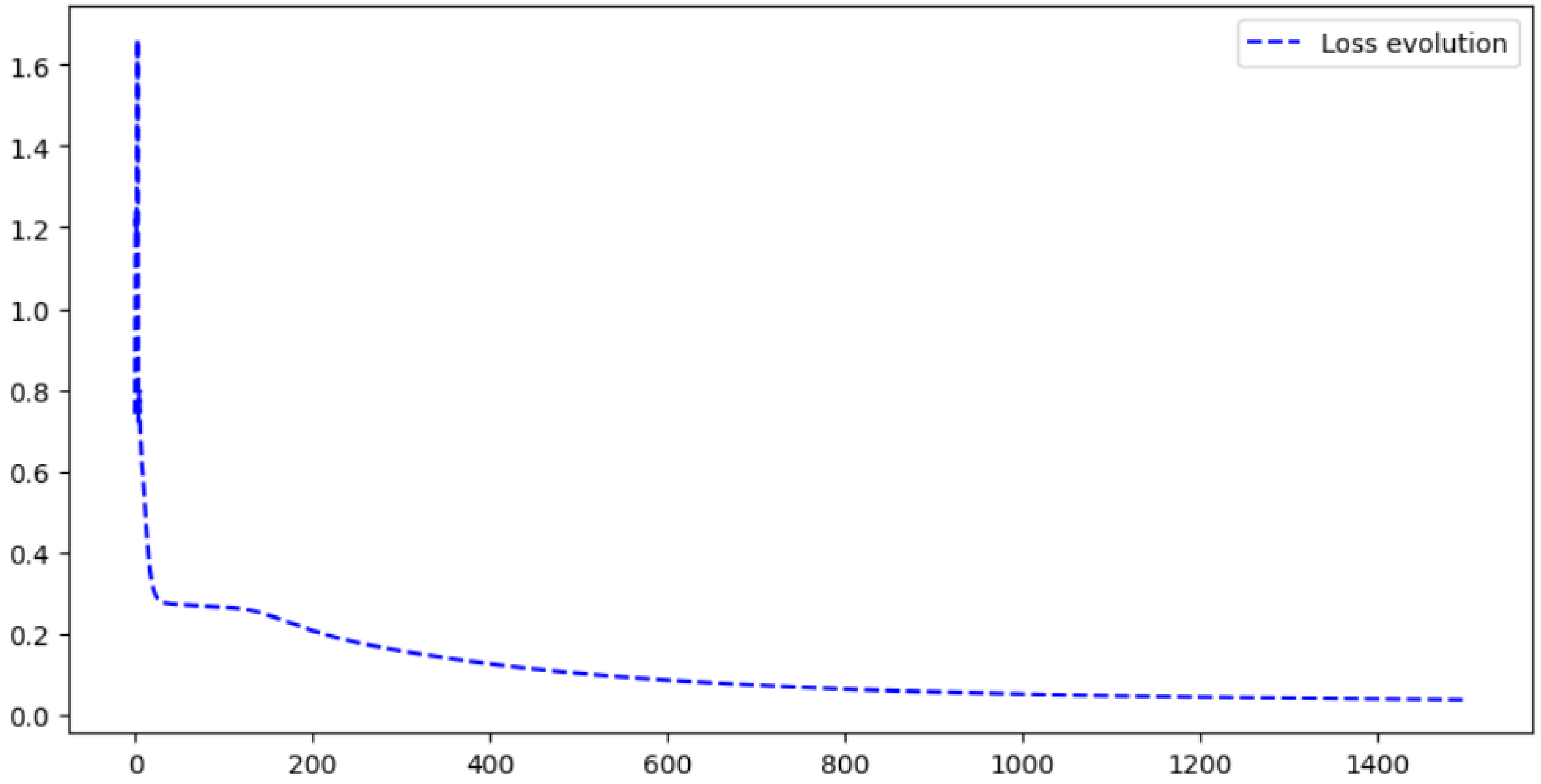
```python
# Check accuracy after training
acc = shallow_neural_net_pt.accuracy(shallow_neural_net_pt(train_inputs_pt), train_outputs_pt).item()
print(acc)
```

```
0.9910000562667847
```

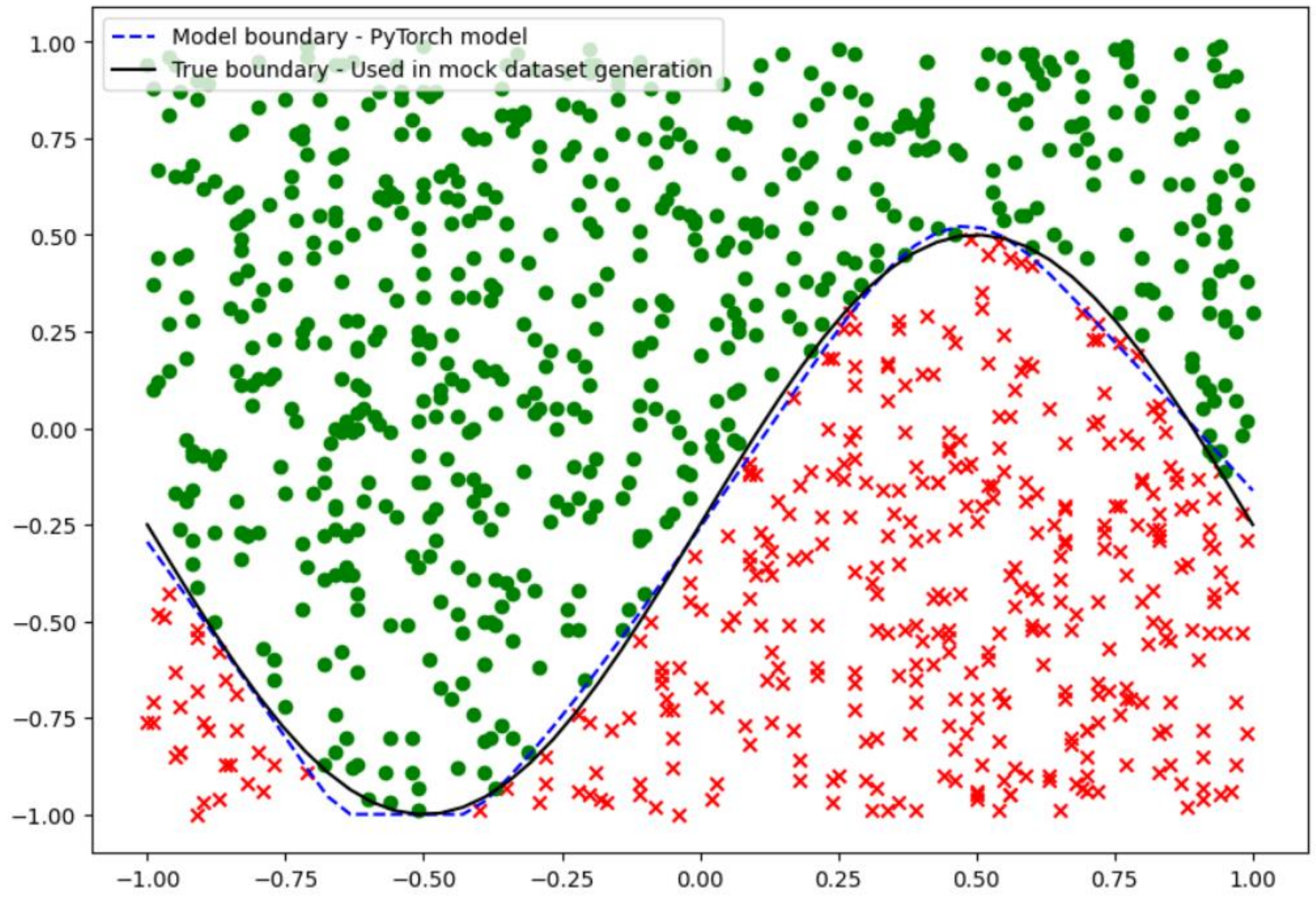
Trying our trainer function... Works!



```
plt.figure(figsize = (10, 5))
plt.plot(list(range(len(shallow_neural_net_pt.loss_history))), \
         shallow_neural_net_pt.loss_history, "b--", label = "Loss evolution")
plt.legend(loc = "best")
plt.show()
```



Trying our trainer function… Works!



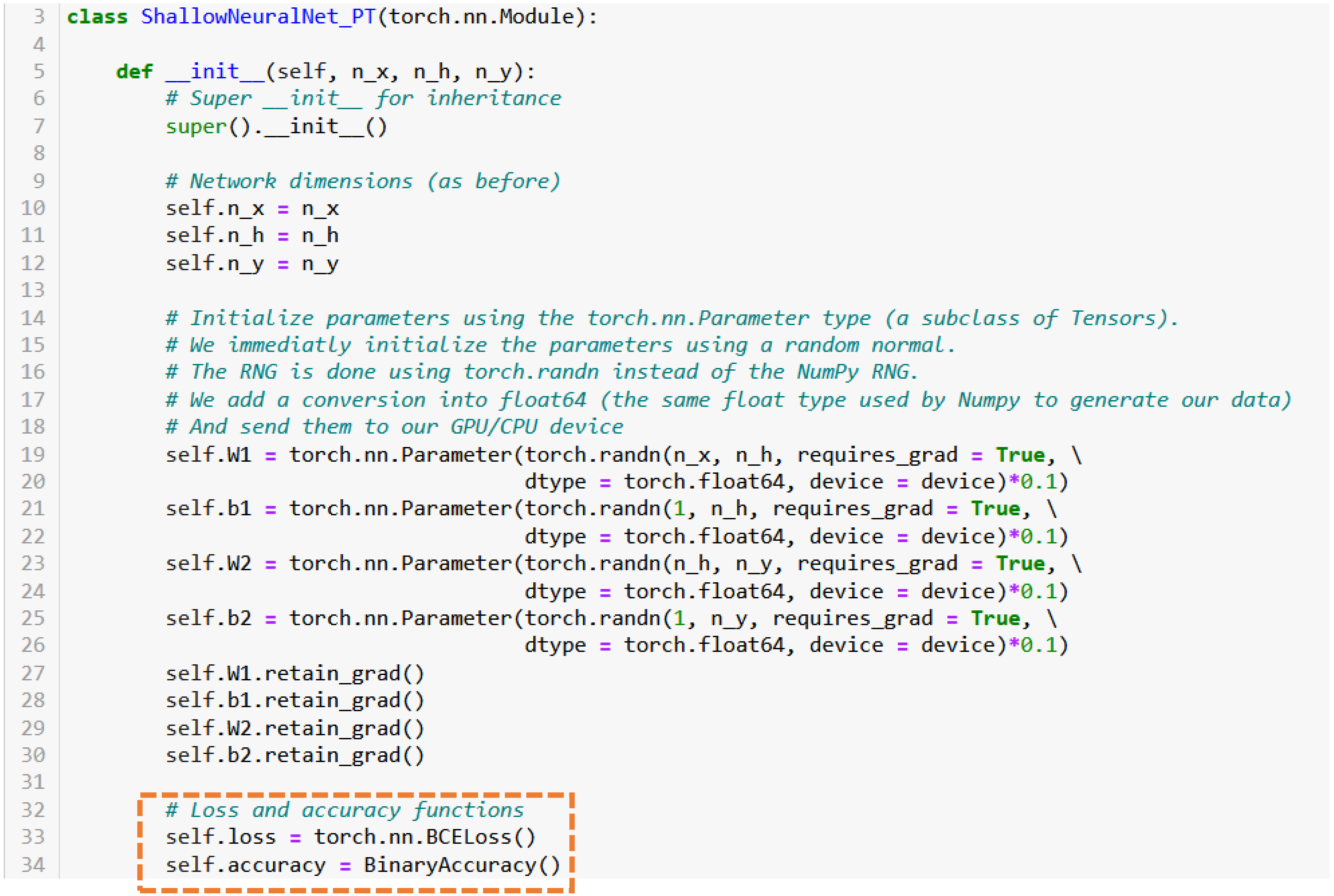
Using PyTorch losses and metrics

In order to make our model even simpler and more efficient, we will use the loss functions and evaluation functions from PyTorch.

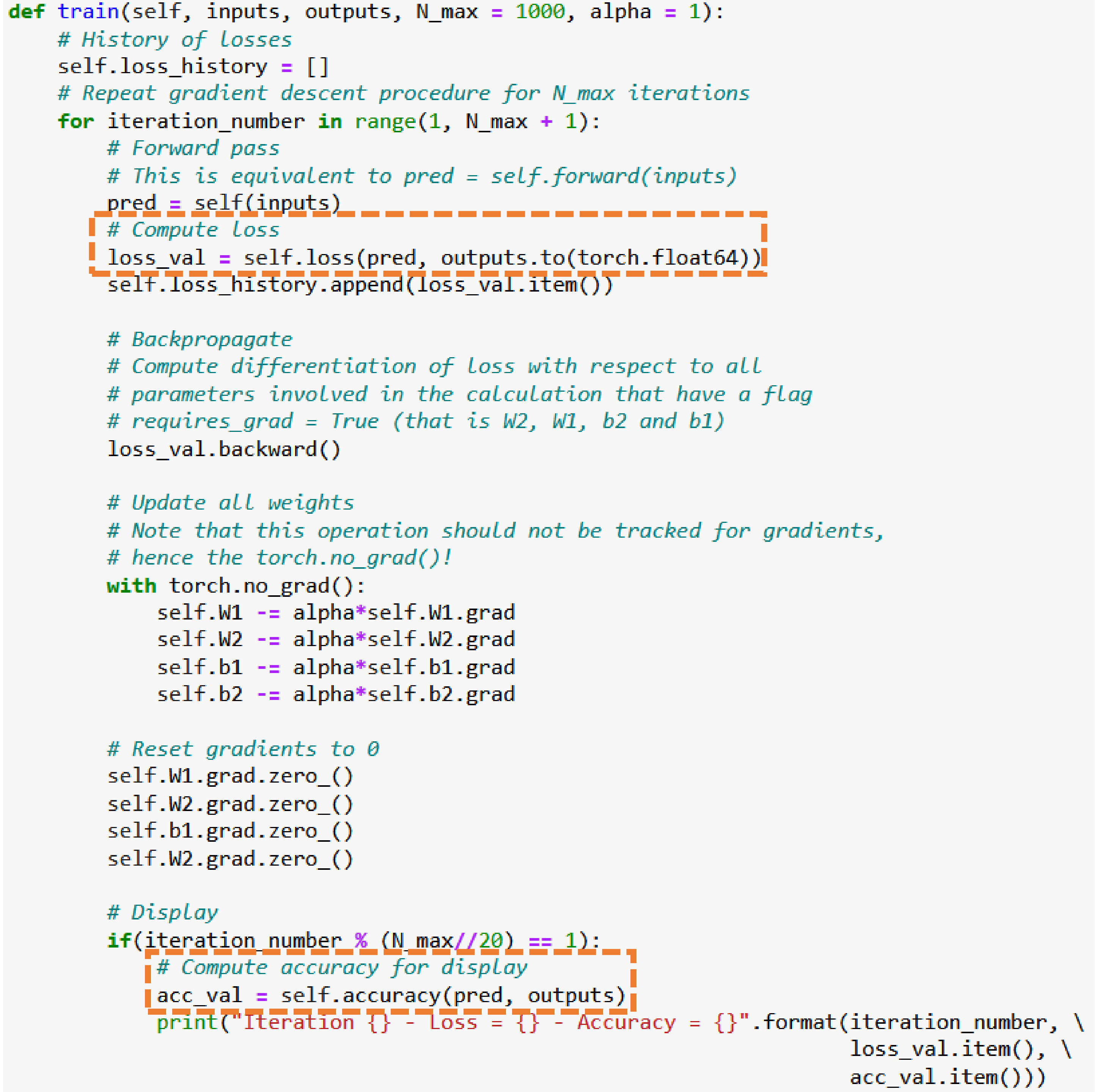
Our **CE_loss()** and **accuracy()** methods will therefore be replaced with the built-in **nn.BCELoss()** and the **BinaryAccuracy()** functions from the **PyTorch** and **TorchMetrics** libraries.

Feel free to have a look at the loss functions available in **PyTorch**, here: https://pytorch.org/docs/stable/nn.html#loss-functions.

Torchmetrics is a supporting library, which provides a few additional losses and metrics functions, ready to use with **PyTorch**: https://torchmetrics.readthedocs.io/en/stable/all-metrics.html.



```python
class ShallowNeuralNet_PT(torch.nn.Module):

    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()

        # Loss and accuracy functions
        self.loss = torch.nn.BCELoss()
        self.accuracy = BinaryAccuracy()
```



```python
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # 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
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # 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()))
```

```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 = 1001, alpha = 5)
```

```
Iteration 1 - Loss = 0.7197759687513233 - Accuracy = 0.37400001287460327
Iteration 51 - Loss = 0.27012799843420826 - Accuracy = 0.874000012874603
Iteration 101 - Loss = 0.26844662460163987 - Accuracy = 0.8759999871253967
Iteration 151 - Loss = 0.2514640191969197 - Accuracy = 0.878000020980835
Iteration 201 - Loss = 0.20506982178051603 - Accuracy = 0.9070000052452087
Iteration 251 - Loss = 0.17680756703390124 - Accuracy = 0.9160000085830688
Iteration 301 - Loss = 0.1579480030878312 - Accuracy = 0.925000011920929
Iteration 351 - Loss = 0.14387554010094675 - Accuracy = 0.9359999895095825
Iteration 401 - Loss = 0.13283459121849153 - Accuracy = 0.9440000057220459
Iteration 451 - Loss = 0.12385125501615053 - Accuracy = 0.949999988079071
Iteration 501 - Loss = 0.11634485891740215 - Accuracy = 0.9580000042915344
Iteration 551 - Loss = 0.10992562465020217 - Accuracy = 0.9620000123977661
Iteration 601 - Loss = 0.10428099036393183 - Accuracy = 0.968999981880188
Iteration 651 - Loss = 0.09912382760500893 - Accuracy = 0.9700000286102295
Iteration 701 - Loss = 0.09418616337556611 - Accuracy = 0.9729999899864197
Iteration 751 - Loss = 0.08925529750401755 - Accuracy = 0.9739999771118164
Iteration 801 - Loss = 0.08424532485311009 - Accuracy = 0.9760000109672546
Iteration 851 - Loss = 0.0792171201563788 - Accuracy = 0.9760000109672546
Iteration 901 - Loss = 0.07430772516730857 - Accuracy = 0.9760000109672546
Iteration 951 - Loss = 0.06965367567190604 - Accuracy = 0.9789999723434448
Iteration 1001 - Loss = 0.0653535992262251 - Accuracy = 0.9800000190734863
```

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

```
0.9800000190734863
```

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!

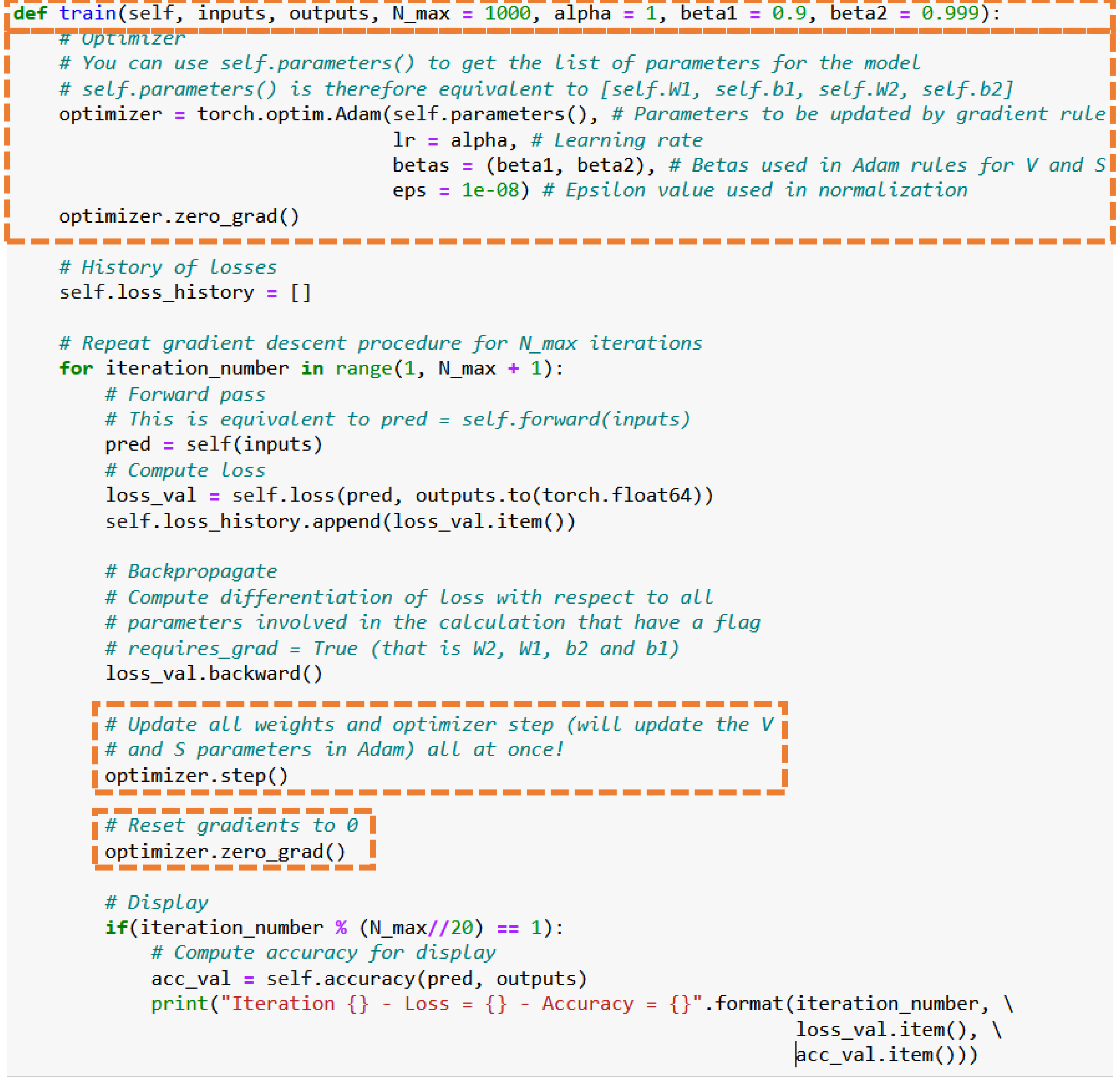
Using advanced optimizers (Adam, etc.)

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

- **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!



```python
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

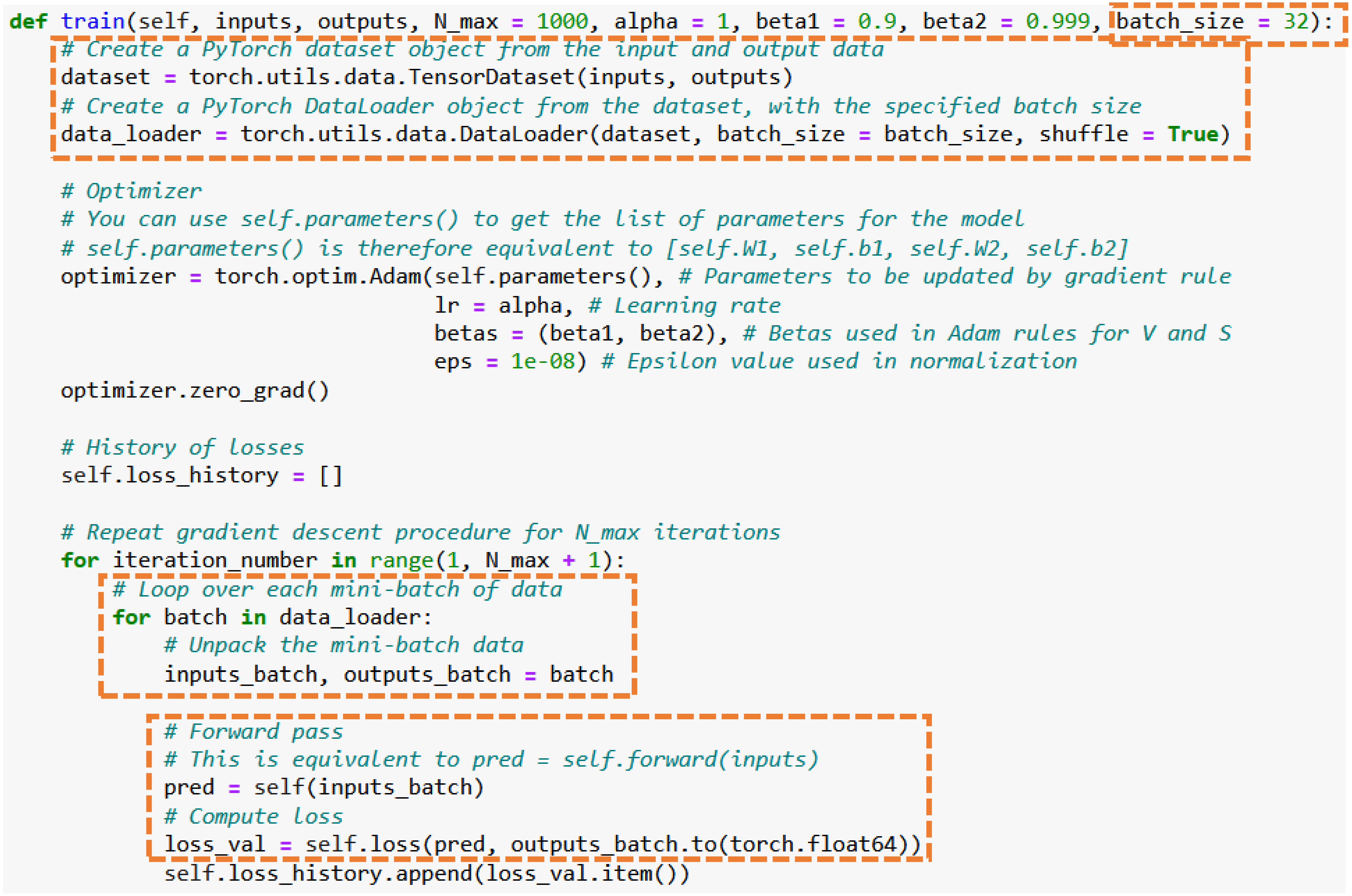
Implementing a stochastic mini-batch GD

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
```

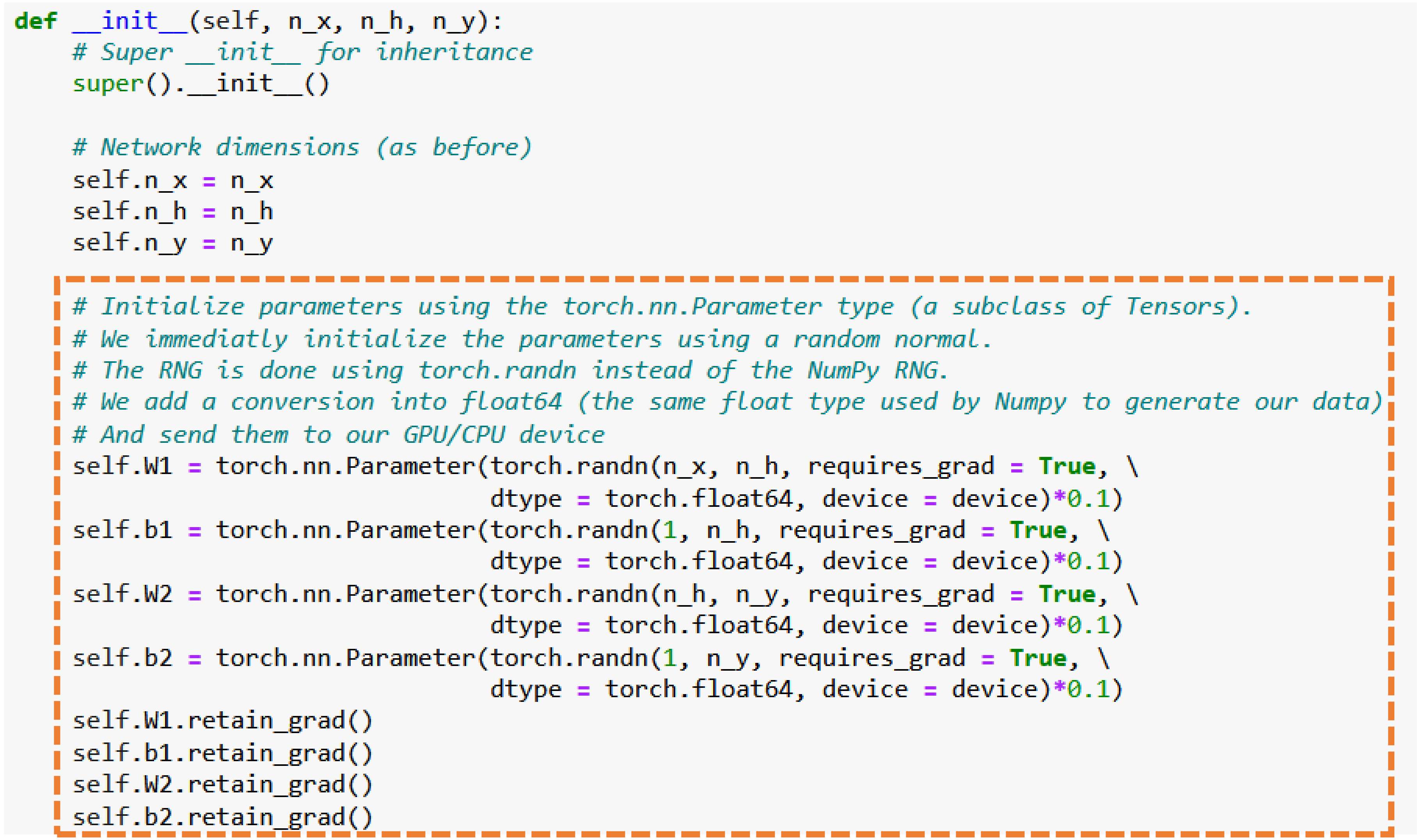
```python
# Check accuracy after training
acc = shallow_neural_net_pt.accuracy(shallow_neural_net_pt(train_inputs_pt), train_outputs_pt).item()
print(acc)
```

```
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!**



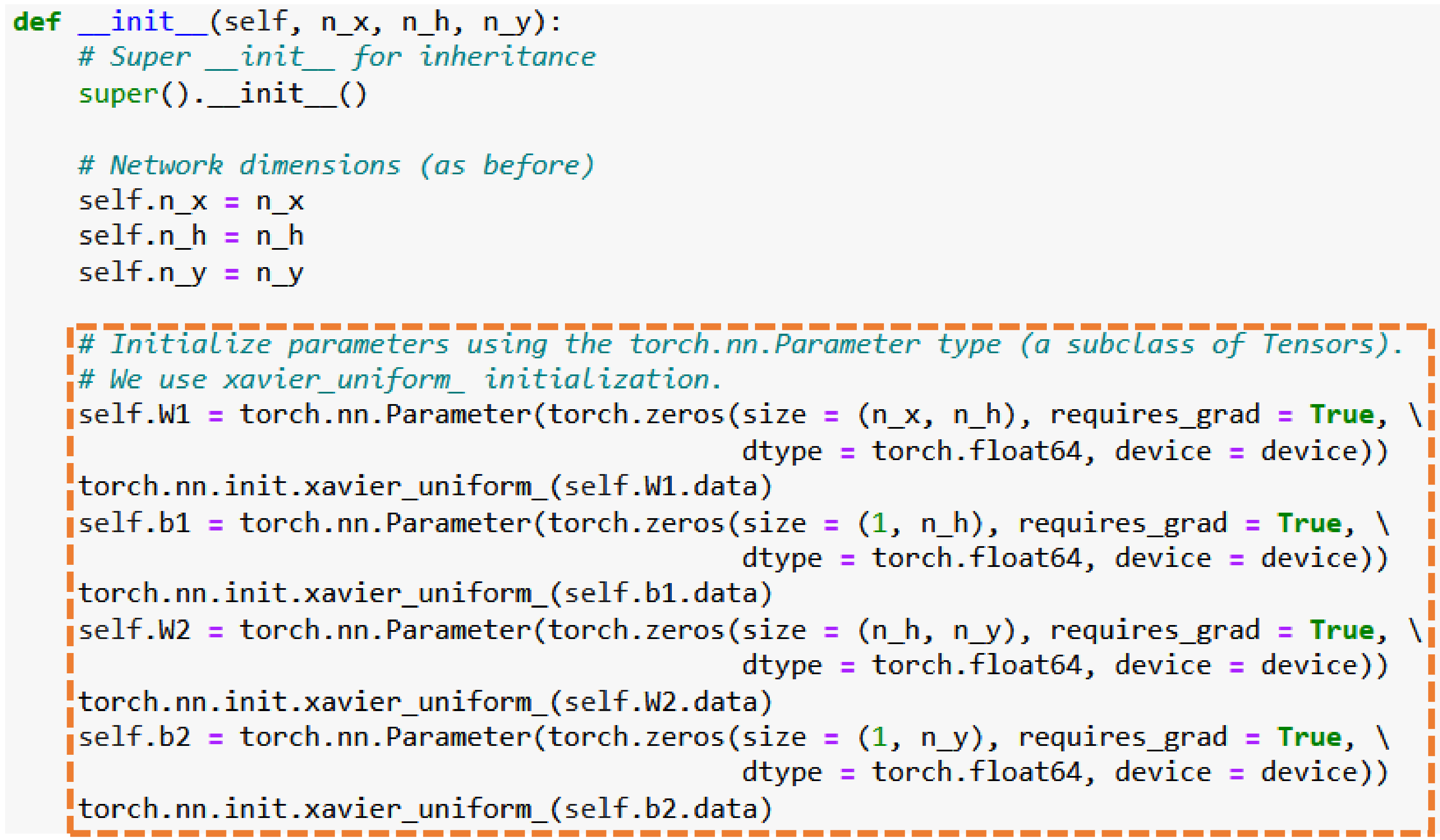
```
def __init__(self, n_x, n_h, n_y):
    # Super __init__ for inheritance
    super().__init__()

    # 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

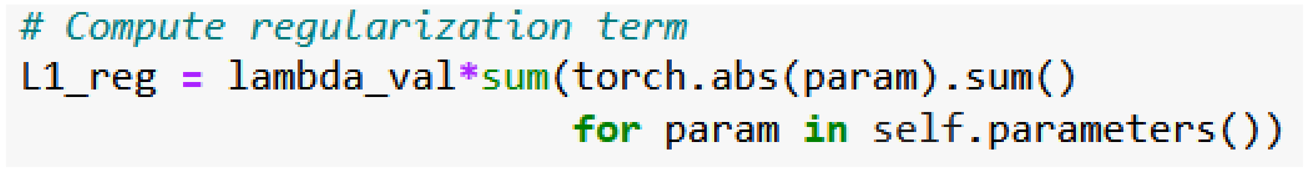


```python
def __init__(self, n_x, n_h, n_y):
    # Super __init__ for inheritance
    super().__init__()

    # 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

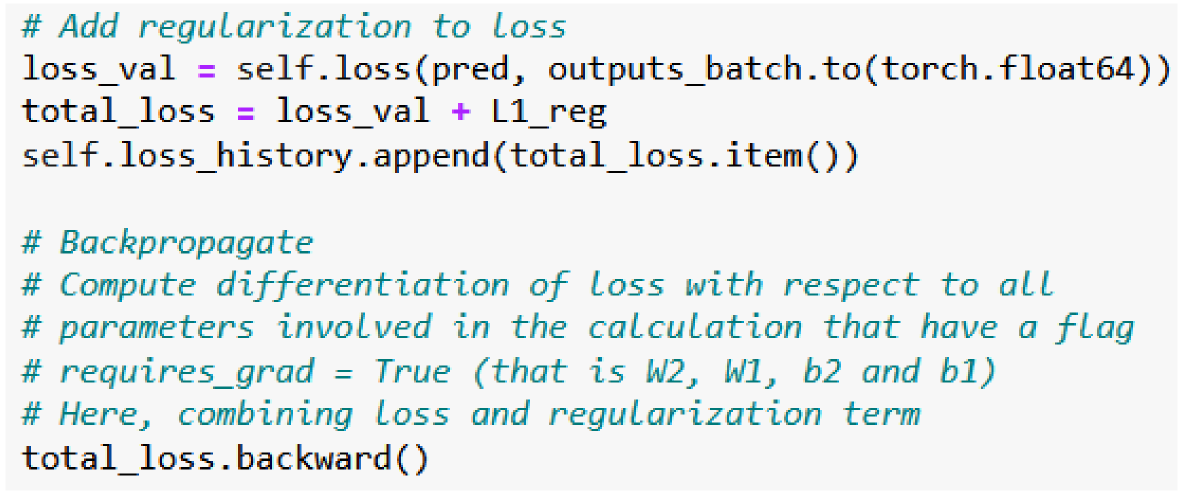
We have also seen during **Ridge Regression** (on Week 1) that it is sometimes beneficial to add a **regularization** term to our loss functions.

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



```python
# 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**.



```python
# 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()
```

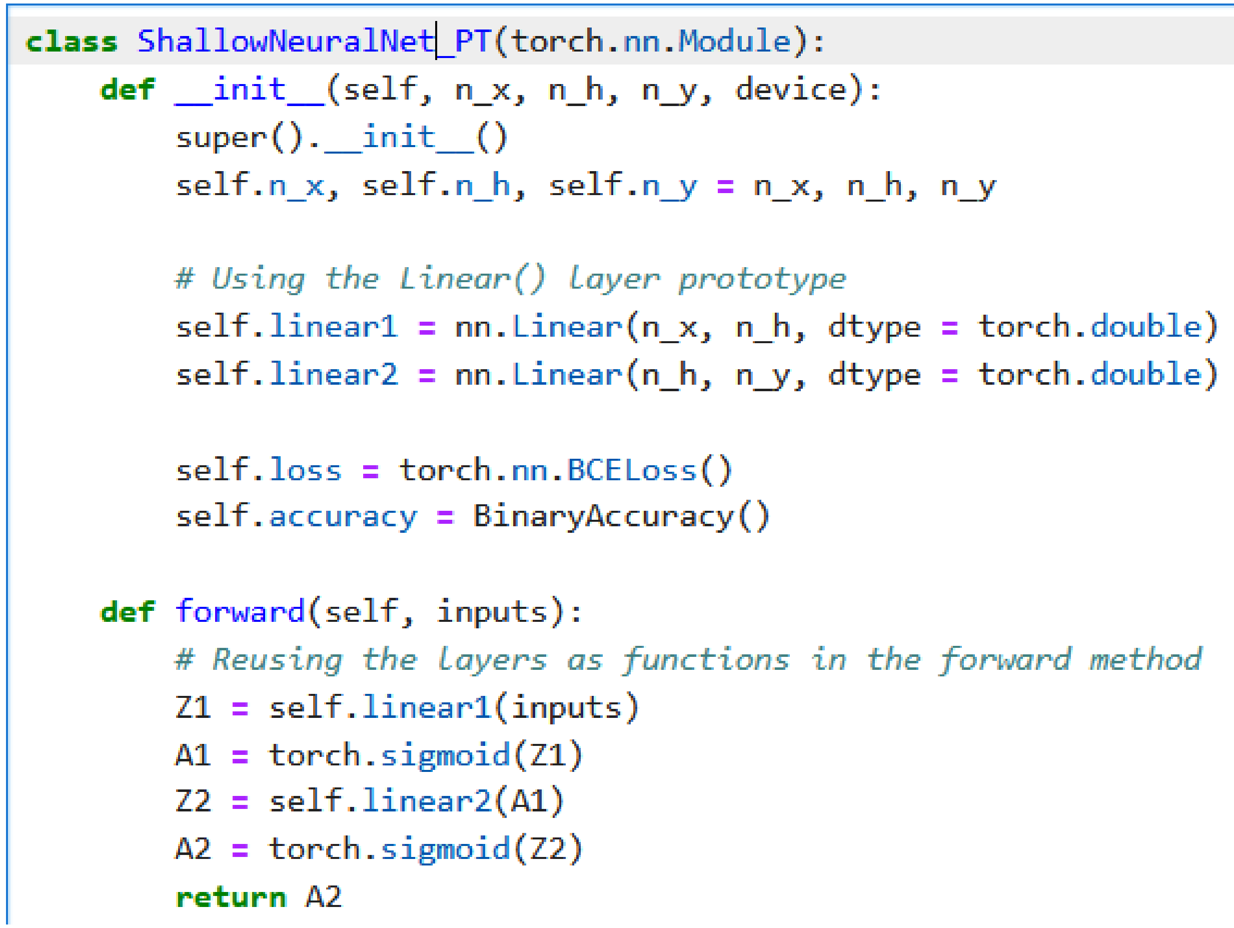
Last but not least, using layers prototypes

We could

- Define the trainable parameters,
- And the operation $WX + b$ to use in a given layer,

All at once, by using the **Linear()** layer available in PyTorch!

- We simply use the Linear() layer and call it as a function in the forward method!
- And remove the Parameter objects in our __init__, as they will now be attributes of the Linear() objects.



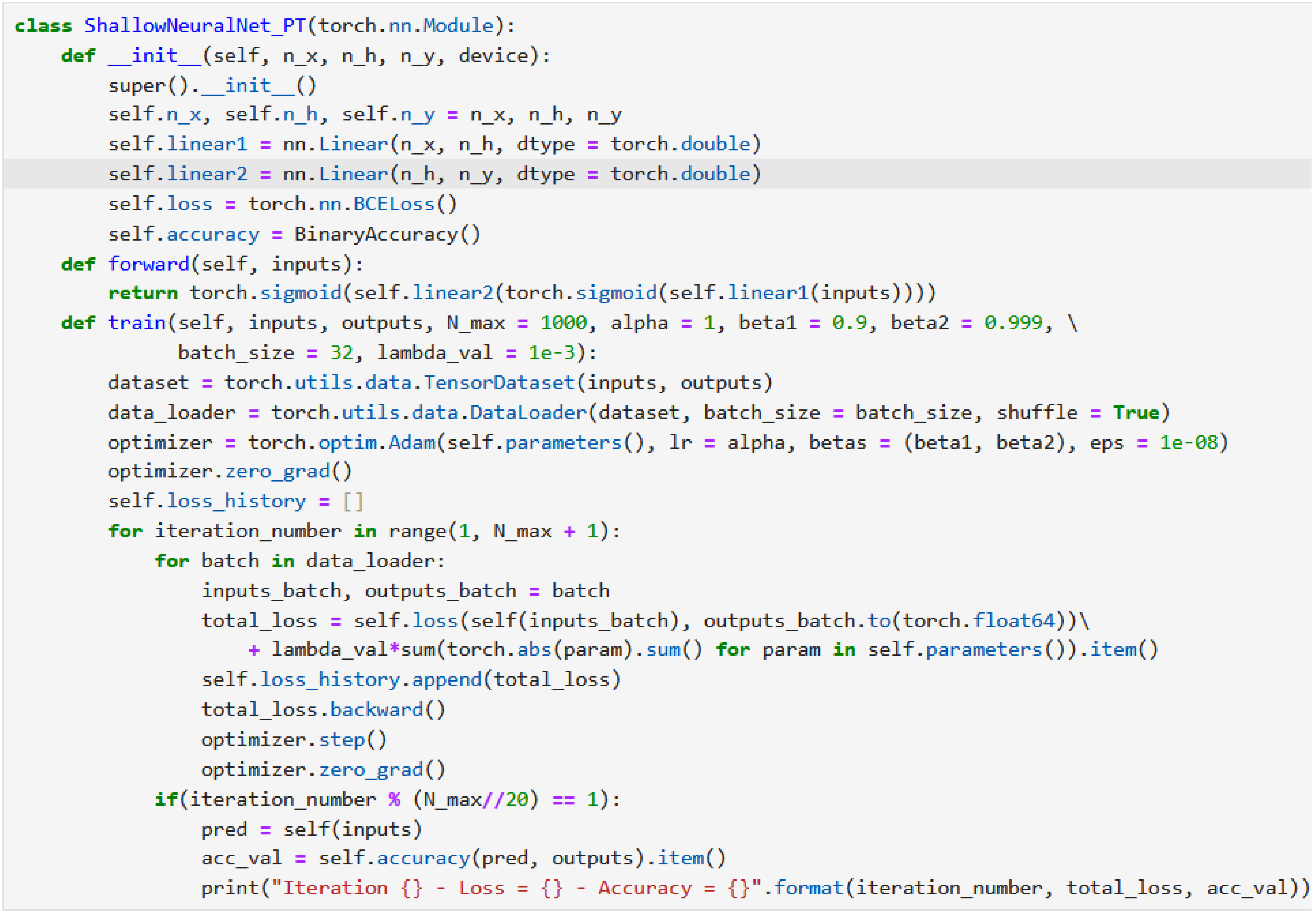
```python
class ShallowNeuralNet_PT(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y, device):
        super().__init__()
        self.n_x, self.n_h, self.n_y = n_x, n_h, n_y

        # Using the Linear() layer prototype
        self.linear1 = nn.Linear(n_x, n_h, dtype = torch.double)
        self.linear2 = nn.Linear(n_h, n_y, dtype = torch.double)

        self.loss = torch.nn.BCELoss()
        self.accuracy = BinaryAccuracy()

    def forward(self, inputs):
        # Reusing the layers as functions in the forward method
        Z1 = self.linear1(inputs)
        A1 = torch.sigmoid(Z1)
        Z2 = self.linear2(A1)
        A2 = torch.sigmoid(Z2)
        return A2
```

To summarize

We now have a full Neural Network class, written in PyTorch, with:

- 2 linear layers, sigmoid activation functions,
- Xavier uniform initialization on trainable parameters,
- Forward pass method,
- Autograd backpropagation and trainer method,
- Adam optimizer,
- Dataloader allowing for stochastic mini-batches,
- Cross-entropy loss and accuracies,
- L1 regularization.

And it runs/trains at the speed of light (almost…) on GPU!



```python
class ShallowNeuralNet_PT(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y, device):
        super().__init__()
        self.n_x, self.n_h, self.n_y = n_x, n_h, n_y
        self.linear1 = nn.Linear(n_x, n_h, dtype = torch.double)
        self.linear2 = nn.Linear(n_h, n_y, dtype = torch.double)
        self.loss = torch.nn.BCELoss()
        self.accuracy = BinaryAccuracy()
    def forward(self, inputs):
        return torch.sigmoid(self.linear2(torch.sigmoid(self.linear1(inputs))))
    def train(self, inputs, outputs, N_max = 1000, alpha = 1, beta1 = 0.9, beta2 = 0.999, \
              batch_size = 32, lambda_val = 1e-3):
        dataset = torch.utils.data.TensorDataset(inputs, outputs)
        data_loader = torch.utils.data.DataLoader(dataset, batch_size = batch_size, shuffle = True)
        optimizer = torch.optim.Adam(self.parameters(), lr = alpha, betas = (beta1, beta2), eps = 1e-08)
        optimizer.zero_grad()
        self.loss_history = []
        for iteration_number in range(1, N_max + 1):
            for batch in data_loader:
                inputs_batch, outputs_batch = batch
                total_loss = self.loss(self(inputs_batch), outputs_batch.to(torch.float64))\
                    + lambda_val*sum(torch.abs(param).sum() for param in self.parameters()).item()
                self.loss_history.append(total_loss)
                total_loss.backward()
                optimizer.step()
                optimizer.zero_grad()
            if(iteration_number % (N_max//20) == 1):
                pred = self(inputs)
                acc_val = self.accuracy(pred, outputs).item()
                print("Iteration {} - Loss = {} - Accuracy = {}".format(iteration_number, total_loss, acc_val))
```