Artificial Neural Networks with PyTorch

Aniello Panariello, Emanuele Frascaroli, Matteo Boschini, Lorenzo Bonicelli, Angelo Porrello

November 18, 2022

University of Modena and Reggio Emilia

Introduction

What is Pytorch?

PyTorch is a Python-based scientific computing **package** that is both:

- A replacement for **NumPy** to use the power of GPUs
- a deep learning research platform that provides maximum flexibility and speed

PyTorch's Tensors

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
import torch
```

Construct a randomly initialized matrix:

```
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.8005, 0.7715, 0.1023],
        [0.6883, 0.7806, 0.0691],
        [0.8946, 0.7593, 0.0940],
        [0.4308, 0.9524, 0.6512],
        [0.0237, 0.3999, 0.4181]])
```

PyTorch's Tensors

You can use standard NumPy-like indexing with all bells and whistles!

```
print(x[:, 1])
```

```
tensor([-0.7393, -1.1572, -1.1516, -0.7919,  0.3470])
```

Resizing: If you want to resize/reshape tensor, you can use either `torch.reshape()` or `torch.view()`. The size `-1` is inferred from other dimensions.

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)
print(x.size())
print(y.size())
print(z.size())
```

```
torch.Size([4, 4])
torch.Size([16])
torch.Size([2, 8])
```

Note that `torch.view()` uses the same underlying memory as the original data, while `torch.reshape()` may or may not do the same thing. If you need to explicitly get an independent copy of your data, use `torch.clone()`

PyTorch's Tensors

If you have a one element tensor, use `.item()` to get the value as a Python number.

```
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([1.0897])
1.0896666049957275
```

100+ Tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random numbers, etc., are described here.

PyTorch's Tensors

Converting a Torch Tensor to a NumPy array and vice versa is a breeze. The Torch Tensor and NumPy array will share their underlying memory locations (if the Torch Tensor is on CPU), and changing one will change the other.

```
a = torch.ones(5)                    tensor([1., 1., 1., 1., 1.])
print(a)                             [1. 1. 1. 1. 1.]
b = a.numpy()
print(b)

import numpy as np                   [2. 2. 2. 2. 2.]
a = np.ones(5)                       tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

PyTorch's Tensors

All the Tensors on the CPU support converting to NumPy and back.

```
# let us run this cell only if CUDA is available
# We will use ``torch.device`` objects to move tensors in and out of GPU
if torch.cuda.is_available():
    device = torch.device("cuda:0")          # a CUDA device object
    y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
    x = x.to(device)                       # or just use strings ``.to("cuda:0")``
    z = x + y
    print(z)                               # tensor([2.0897], device='cuda:0')
    print(z.to("cpu", torch.double))       # tensor([2.0897], dtype=torch.float64)
```

Autograd

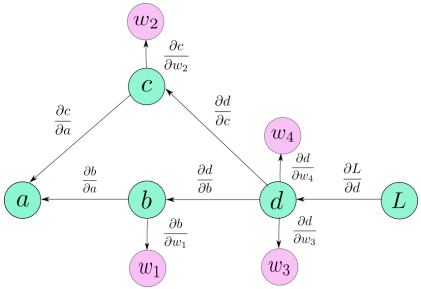
Autograd

Central to all neural networks in PyTorch is the `autograd` package.

The `autograd` package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, **which means that your backprop is defined by how your code is run**, and that every single iteration can be different.

Let us see this in more simple terms with some examples.

Tensor

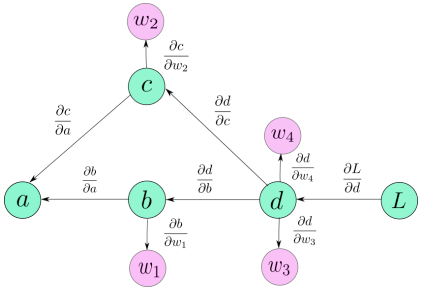


`torch.Tensor` is the central class of the package.

- If you set its attribute `.requires_grad` as `True`, it starts **to track** all operations on it.
- When you finish your computation you can call `.backward()` and **have all the gradients computed automatically**.
- The gradient for this tensor will be **accumulated** into `.grad` attribute.

To stop a tensor from tracking history, you can call `.detach()` to **detach it** from the computation history, and to prevent future computation from being tracked.

Backward

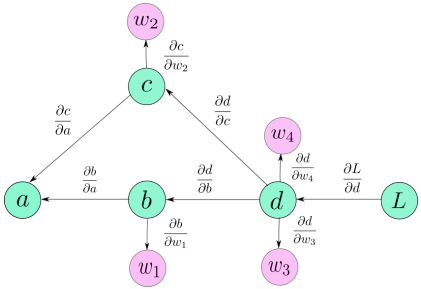


To prevent tracking history (and using memory), you can also wrap the code block in with `torch.no_grad()`.

This can be particularly helpful **when evaluating a model** because the model may have trainable parameters with `requires_grad=True`, but for which we don't need the gradients.

If you want to compute the derivatives, you can call `.backward()` on a `Tensor`. If `Tensor` is a scalar (i.e. it holds a one element data), you don't need to specify any arguments to `backward()`.

Backward



To prevent tracking history (and using memory), you can also wrap the code block in with `torch.no_grad()`.

This can be particularly helpful **when evaluating a model** because the model may have trainable parameters with `requires_grad=True`, but for which we don't need the gradients.

If you want to compute the derivatives, you can call `.backward()` on a `Tensor`. If `Tensor` is a scalar (i.e. it holds a one element data), you don't need to specify any arguments to `backward()`.

Backward

```
import torch
x = torch.ones(2, 2, requires_grad=True)
print(x)         # tensor([[1., 1.], [1., 1.]], requires_grad=True)
y = x + 2        # let's define a tensor operation
print(y)         # tensor([[3., 3.], [3., 3.]], grad_fn=<AddBackward0>)
```

y was created as a result of an operation, so it has a `grad_fn`.

```
print(y.grad_fn)     # <AddBackward0 object at 0x7f27857f7c88>
```

Backward

Do more operations on `y`

```
z = y * y * 3
out = z.mean()

print(z)        # tensor([[27., 27.], [27., 27.]], grad_fn=<MulBackward0>)
print(out)      # tensor(27., grad_fn=<MeanBackward0>)
```

Backward

Do more operations on `y`

```
z = y * y * 3
out = z.mean()

print(z)        # tensor([[27., 27.], [27., 27.]], grad_fn=<MulBackward0>)
print(out)      # tensor(27., grad_fn=<MeanBackward0>)
```

Let's backprop now.

```
out.backward() # print gradients d(out)/dx
print(x.grad)  # tensor([[4.5000, 4.5000], [4.5000, 4.5000]])
```

Please refer to the Pytorch's doc for additional details on Autograd.

Neural Networks

Neural Networks

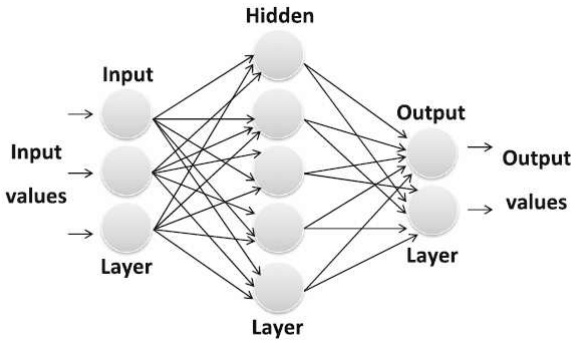
Neural networks can be constructed using the `torch.nn` package.

Now that you had a glimpse of `autograd`, `nn` depends on autograd to define models and differentiate them.

An `nn.Module` contains layers, and a method `forward(input)` that returns the output.

Neural Networks: MLP

Multi-Layer Perceptron: is a simple feed-forward network. It takes the input, feeds it through several layers one after the other, and then finally gives the output.



Training procedure

A typical training procedure for a neural network is as follows:

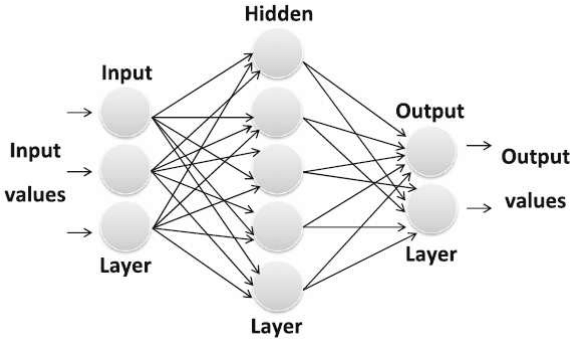
- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule:
  `weight = weight - learning_rate * gradient`

Neural Networks

A typical training procedure for a neural network is as follows:

- **Define the neural network that has some learnable parameters (or weights)**

Let's define this network:



nn.Module skeleton

Let's define the skeleton:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self, **kwargs):
        pass

    def forward(self, x):
        pass

net = Net()
print(net)
```

nn.Module skeleton

Let's define the `__init__()` method:

```
class Net(nn.Module):

    def __init__(self, ...):
        pass

    def forward(self, x):
        pass

net = Net()
print(net)
```

```
def __init__(self, in_size, hidden_size, num_classes):
        # always call the super-constructor
        super(Net, self).__init__()

        # a linear operation: y = Wx + b
        self.fc1 = nn.Linear(in_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, num_classes)
```

nn.Module skeleton

Let's define the `forward(x)` method:

```
class Net(nn.Module):

    def __init__(self, ...):
        pass

    def forward(self, x):
        pass

net = Net()
print(net)
```

```
def forward(self, x):
        # x.shape = (batch_size, in_size)
        x = self.fc1(x)
        # x.shape = (batch_size, hidden_size)
        x = F.relu(x)
        # RELU does not change the shape!
        x = self.fc2(x)
        # x.shape = (batch_size, num_classes)

        return x
```

nn.Module skeleton

Once the network has been defined, we are able to print it:

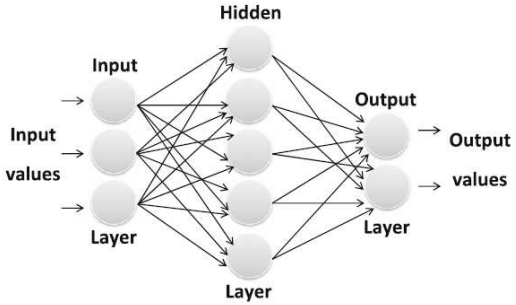
```
class Net(nn.Module):

    def __init__(self, ...):
        self.fc1 = ...
        self.fc2 = ...
        ...

    def forward(self, x):
        x = ...
        x = ...
        return x
```

```
if __name__ == "__main__":
    net = Net(784, 100, 10)
    print(net)
```

```
Net(
  (fc1): Linear(in_features=784, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
)
```



nn.Module skeleton

You just have to define the **forward** function, and the **backward** function (where gradients are computed) is automatically defined for you using `autograd`. You can use any of the `Tensor` operations in the forward function.

The learnable parameters of a model are returned by `net.parameters()`.

```
if __name__ == "__main__":
    net = Net(784, 100, 10)
    params = list(net.parameters())
    print(len(params))
    print(params[0].size())  # fc1's .weight
```

```
4
torch.Size([100, 784])
```

Let try a random 28x28 input. Note: expected input size of this net is 784=28x28.

To use this net on MNIST dataset, please flatten the images from the dataset to 784.

```
if __name__ == "__main__":
    net = Net(784, 100, 10)
    B = 1 # batch size
    in_size = 28*28
    x = torch.randn(B, in_size)
    output = net(x)
    print(output)
```

```
tensor([[-0.0091, -0.3007,  0.2303,
         -0.2525, -0.1047,  0.0665,
         0.0558, -0.4858, 0.4425,
         -0.3851]], grad_fn=<AddmmBackward0>)
```

Batch size axis

A brief note on mini-batch sizes:

- `torch.nn` only supports mini-batches, namely inputs that are a mini-batch of samples, and not a single sample.
- For example, `nn.Linear` will take in a 2D Tensor of `nSamples` x `in_size`.

```
if __name__ == "__main__":
    net = Net(784, 100, 10)
    B = 16 # batch size
    in_size = 28*28
    x = torch.randn(B, in_size)
    output = net(x)
```

If you have a single sample, just use `x.unsqueeze(0)` to add a fake batch dimension.

Recap

Recap:

- `torch.Tensor` - A multi-dimensional array with support for autograd operations like `backward()`. Also holds the gradient w.r.t. the tensor.
- `nn.Module` - Neural network module. Convenient way of encapsulating parameters, with helpers for moving them to GPU, exporting, loading, etc.
- `nn.Parameter` - A kind of `Tensor`, that is automatically registered as a parameter when assigned as an attribute to a `Module`.
- `autograd.Function` - Implements forward and backward definitions of an `autograd` operation.

Still Left:

- Computing the loss
- Updating the weights of the network

Loss Function

Loss Function

A loss function takes the (output, target) pair of inputs, and computes a value that estimates **how far away** the output is from the target.

- There are several different loss functions under the `nn` package.
- A simple loss is: `nn.MSELoss` which computes the mean-squared error between the input x and the target $\widehat{x}$.

$$\text{MSE}(x, \widehat{x}) = \frac{\sum_{i=1}^{n} (x_i - \widehat{x}_i)^2}{n}$$

```
output = net(x)
target = torch.randn(10)  # a dummy target
target = target.view(1, -1)
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

Output:

```
tensor(0.6021, grad_fn=<MseLossBackward>)
```

Computational graph

Now, if you follow loss in the backward direction, using its `.grad_fn` attribute, you will see a graph of computations that looks like this:

```
x    -> fc1 -> relu -> fc2
     -> MSELoss
     -> loss
```

So, when we call `loss.backward()`, the whole graph is differentiated w.r.t. the loss, and all Tensors in the graph that has `requires_grad=True` will have their `.grad` Tensor accumulated with the gradient.

Backward

To **backpropagate the error** all we have to do is to `loss.backward()`. You need to clear the existing gradients, else gradients will be accumulated.

```
# zeroes the gradient buffers
# of all parameters
net.zero_grad()
print('fc1.bias.grad before backward')
print(net.fc1.bias.grad)

loss.backward()
print('fc1.bias.grad after backward')
print(net.fc1.bias.grad)
```

Output:

```
fc1.bias.grad before backward
tensor([0., ..., 0., 0.])
fc1.bias.grad after backward
tensor([-0.0081, ..., -0.0021,
    -0.0041])
```

Cross entropy loss

The `nn.CrossEntropyLoss` combines `nn.LogSoftmax` and `nn.NLLLoss` in one single module.

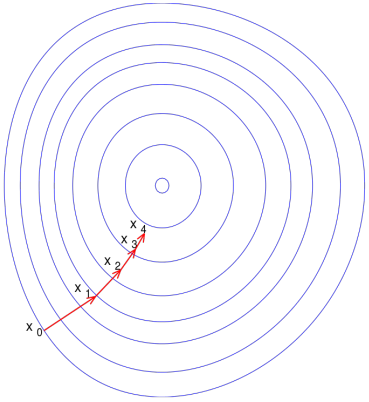
- It is useful when training a **classification problem with C classes**.
- The input is expected to contain raw, unnormalized scores for each class.

```
x = torch.randn(3, 5, requires_grad=True)
"""
tensor([[-1.3930, -0.1162, -0.7232, -1.1799,  0.2584],
        [ 0.2351,  0.9933, -0.8860,  0.7573, -0.3822],
        [-0.8702, -0.0620,  1.3838,  0.4569,  1.6368]],
        requires_grad=True)
"""
target = torch.empty(3, dtype=torch.long)
target = target.random_(5) # tensor([0, 3, 4])
loss = nn.CrossEntropyLoss()
output = loss(x, target)
output.backward()
```

The neural network package contains various modules and loss functions that form the building blocks of deep neural networks. A full list with documentation is here.

Optimization

Gradient Descent



Gradient descent is an iterative optimization algorithm for finding the minimum of a function. How? Take step proportional to the negative of the gradient of the function at the current point.

If we consider a function $f(\boldsymbol{\theta})$, the **gradient descent update** can be expressed as:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} f(\boldsymbol{\theta}) \tag{1}$$

for each parameter θ_j. The size of the step is controlled by **learning rate** α.

Gradient Descent

Gradient descent is often used in machine learning to **minimize a cost function**, usually also called *objective* or *loss* function.

The cost function depends on the model's parameters and is a proxy to evaluate model's performance. Generally speaking, in this framework minimizing the cost equals to maximizing the effectiveness of the model.

Optimization

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

We can implement this using simple Python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data -= f.grad.data * learning_rate
```

Optimization

In practice, it's quite rare to see the procedure described above (**vanilla SGD**).

Moreover, as you use neural networks, you want to use various different cutting-edge optimizers such as Nesterov-SGD, **Adam**, RMSProp, etc.

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)
# optimizer = optim.Adam(net.parameters(), lr=0.01)

# in your training loop:
for i in range(num_epochs):
    optimizer.zero_grad() # zero the gradient buffers
    output = net(input)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step() # Does the update
```

Datasets

Datasets

Generally, when you have to deal with image, text, audio or video data, you can use standard python packages that load data into a numpy array. Then you can convert this array into a torch Tensor.

- For images, packages such as Pillow, OpenCV are useful
- For audio, packages such as scipy and librosa
- For text, either raw Python or Cython based loading, or NLTK and SpaCy are useful

Specifically for vision, a package called `torchvision` provides data loaders for common datasets such as Imagenet, CIFAR10, MNIST, etc. and data transformers for images, viz., `torchvision.datasets` and `torch.utils.data.DataLoader`.

Pipeline

We will:

- Load and normalizing the MNIST training and test datasets using torchvision
- Define a Neural Network
- Define a loss function
- Train the network on the training data
- Test the network on the test data

torchvision

Using torchvision, it's extremely easy to load MNIST.

```
import torch
import torchvision

"""
The output of torchvision datasets are PILImage images of range [0, 1].
We transform them to Tensors of normalized range [-1, 1].
"""
transform = torchvision.transforms.Compose([
     torchvision.transforms.ToTensor(),
     torchvision.transforms.Normalize(0.5, 0.5)])

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
```

torchvision datasets

```
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                       download=True, transform=transform)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                      download=True, transform=transform)
```

These two lines are all you need to have the data processed and setup for you. **Under the hood:**

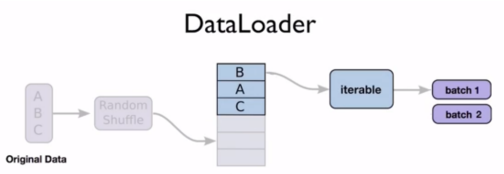
- It downloads the byte files, decodes and converts them into a readable format.
- It also handles the case when the dataset has already been downloaded and processed.
- It cleanly abstracts out all the pestering details.
- It converts PIL images into Pytorch normalized tensors.

Dataloaders

These objects can now be accessed using standard indexes (e.g. `print(trainset[0])`). **However**, most of the time you wouldn't really be accessing such indices but actually forwarding batches of samples to the model. PyTorch provides another wrapper interface called the `torch.utils.data.DataLoader`.

```
dl = DataLoader(dataset=trainset,
                batch_size=32, num_workers=0)

for X, y in dl:
    # perform some operations on batch (X,y)
    print(X.shape, y)
```



Dataloaders

Dataloaders come in handy when you need to prepare data batches (and perhaps shuffle them before every run).

```
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
```

Data-loaders instantiations:

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
                                          shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=128,
                                         shuffle=False, num_workers=2)
```

Dataloaders

Data-loaders usage:

```
for images, labels in trainloader:
    print(images.shape, labels.shape)
    ...
```

```
torch.Size([128, 1, 28, 28]) torch.Size([128])
torch.Size([128, 1, 28, 28]) torch.Size([128])
torch.Size([128, 1, 28, 28]) torch.Size([128])
torch.Size([128, 1, 28, 28]) torch.Size([128])
...
```

Dataloaders

Let us show some of the training images, for fun.


```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

images, labels = iter(trainloader).next()
imshow(torchvision.utils.make_grid(images))
```

Pipeline

We will:

- ~~Load and normalizing the MNIST training and test datasets using torchvision~~
- Define a Neural Network
- Define a loss function
- Train the network on the training data
- Test the network on the test data

Network

Regarding the model, we can copy it from the Neural Network section before and modify it to flatten the 1-channel images.

- ~~Load and normalizing the MNIST training and test datasets using torchvision~~
- ~~Define a Neural Network~~
- Define a loss function
- Train the network on the training data
- Test the network on the test data

```
class Net(nn.Module):
    def __init__(self, ...):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(...)
        self.fc2 = nn.Linear(...)

    def forward(self, x):
        # 28*28 images to 784 vector
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Optimization

Classification with multiple classes $\rightarrow$ let's use the Cross-Entropy loss.
For the optimizer, we could exploit SGD with momentum.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Pipeline

We will:

- ~~Load and normalizing the MNIST training and test datasets using torchvision~~
- ~~Define a Neural Network~~
- ~~Define a loss function~~
- **Train the network on the training data**
- Test the network on the test data

This is when things start to get interesting. We simply have to loop over our data iterator, and feed the inputs to the network and optimize.

Training cycle

Since one epoch is too big to feed at once we divide it in several smaller batches.

Moreover, we need to pass the full dataset multiple times to the same neural network.

```
NUM_EPOCHS = 50

# loop over the dataset NUM_EPOCHS times
for epoch in range(NUM_EPOCHS):
    for i, data in enumerate(trainloader):
        # 0) get the inputs
        # 1) zero the gradients
        # 2) forward
        # 3) backward
        # 4) optimization step
        # optionally: print statistics
        pass
```

Training cycle

```
NUM_EPOCHS = 50

# loop over the dataset NUM_EPOCHS times
for epoch in range(NUM_EPOCHS):
    for i, data in enumerate(trainloader):

        inputs, labels = data                # 0) get the inputs
        optimizer.zero_grad()                # 1) zero the gradients

        outputs = net(inputs)
        loss = criterion(outputs, labels)    # 2) forward

        loss.backward()                      # 3) backward
        optimizer.step()                     # 4) optimization step

        # optionally: print statistics
```

Training cycle

```python
for epoch in range(NUM_EPOCHS):

    running_loss = 0.0
    for i, data in enumerate(trainloader):

        inputs, labels = data                # 0) get the inputs
        optimizer.zero_grad()                # 1) zero the gradients
        outputs = net(inputs)
        loss = criterion(outputs, labels)    # 2) forward
        loss.backward()                      # 3) backward
        optimizer.step()                     # 4) optimization step

        # optionally: print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch+1, i+1, running_loss/2000))
            running_loss = 0.0
```

```
[1,  2000] loss: 2.276
[1,  4000] loss: 2.007
[1,  6000] loss: 1.749
[1,  8000] loss: 1.620
[1, 10000] loss: 1.543
[1, 12000] loss: 1.471
[2,  2000] loss: 1.421
[2,  4000] loss: 1.384
[2,  6000] loss: 1.370
[2,  8000] loss: 1.340
[2, 10000] loss: 1.318
[2, 12000] loss: 1.296
...
```

Evaluation

We will:

- ~~Load and normalizing the MNIST training and test datasets using torchvision~~
- ~~Define a Convolutional Neural Network~~
- ~~Define a loss function~~
- ~~Train the network on the training data~~
- **Test the network on the test data**

We have trained the network for multiple passes over the training dataset, but we need to check if the network has learnt anything at all.

Evaluation

We will check this by predicting the class label for an unseen test-set and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images:
    %d %%' % (100 * correct / total))
```

Training on GPU

Let's first define our device as the first visible cuda device if we have CUDA available:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# Assuming that we are on a CUDA machine, this should print a CUDA device:
print(device)
```

Just like how you transfer a Tensor onto the GPU, you transfer the net onto the GPU.

These methods will recursively go over all modules and convert their parameters and buffers to CUDA tensors.

Finally, remember that you will have to send the inputs and targets at every step to the GPU too.

```
net = net.to(device)
...
for epoch in range(NUM_EPOCHS):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        ...
```