# Exercises Week 2

## Introduction to PyTorch

Leandro VON KRANNICHFELDT
Vinay SHARMA
Amaury WEI

12.09.2024

École polytechnique fédérale de Lausanne

# Warm-up Quiz

Go to

https://**menti**.com

Enter the code

5919 7630

# Why PyTorch?


Hardware-accelerated array computation library

## Main Features

- Automatic differentiation
- Hardware acceleration
- Compatible with Numpy, Scipy, …

## Strong Points

- Open-source & active community
- Easy-to-use (and understand)
- Lots of integrated tools

Similar Libraries:

# Learning PyTorch

## Documentation

PyTorch

https://pytorch.org/docs/stable/index.html

https://pytorch.org/tutorials/beginner/ptcheat.html

## Tutorials

PyTorch

https://pytorch.org/tutorials/beginner/basics/intro.html

## YouTube


can be outdated

# Tensors

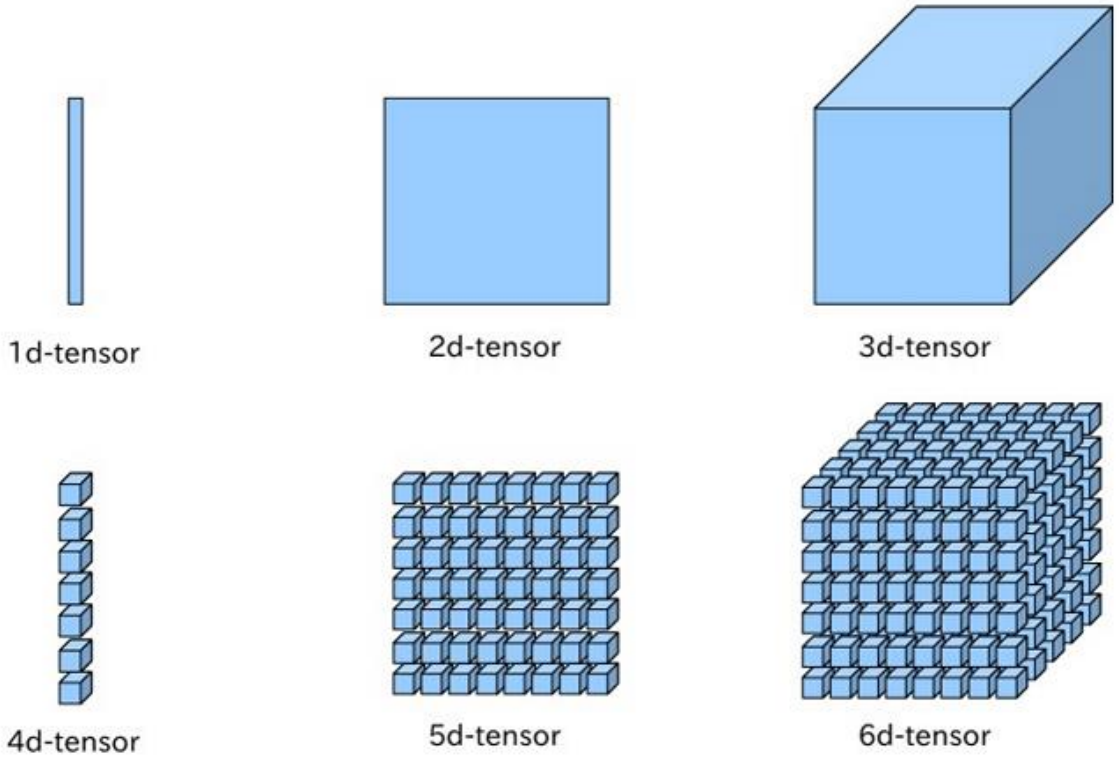
## Declaration

```
x = torch.randn(*size)              # tensor with independent N(0,1) entries
x = torch.[ones|zeros](*size)       # tensor with all 1's [or 0's]
x = torch.tensor(L)                 # create tensor from [nested] list or ndarray L
y = x.clone()                       # clone of x
```

```
np_array = np.array([[1, 2], [3, 4]])
torch_tensor = torch.from_numpy(np_array)
```

## Dimensions

```
>>> t = torch.empty(3, 4, 5)
>>> t.size()
torch.Size([3, 4, 5])
```




Most important concept in DL libraries

# Data Types

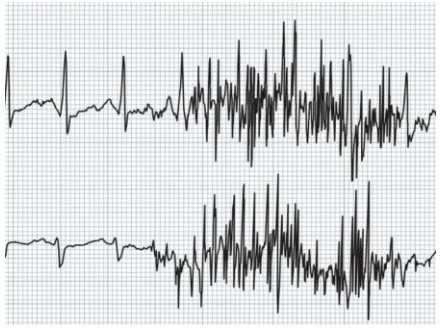
| | |
|---|---|
| 32-bit floating point | `torch.float32` or `torch.float` |
| 64-bit floating point | `torch.float64` or `torch.double` |
| 16-bit floating point | `torch.float16` or `torch.half` |
| 8-bit integer | `torch.int8` |
| 8-bit integer unsigned | `torch.uint8` |
| 16-bit integer | `torch.int16` |
| 32-bit integer | `torch.int32` |

```
a = torch.ones((2, 3), dtype=torch.int16)
b = torch.rand((2, 3), dtype=torch.float64)
```

*Grayscale*

| Pet | Cat | Dog | Turtle | Fish |
|---|---|---|---|---|
| Cat | 1 | 0 | 0 | 0 |
| Dog | 0 | 1 | 0 | 0 |
| Turtle | 0 | 0 | 1 | 0 |
| Fish | 0 | 0 | 0 | 1 |
| Cat | 1 | 0 | 0 | 0 |

*One-hot encoding*



*Timeseries*

# Tensor Operations

## Math Operations

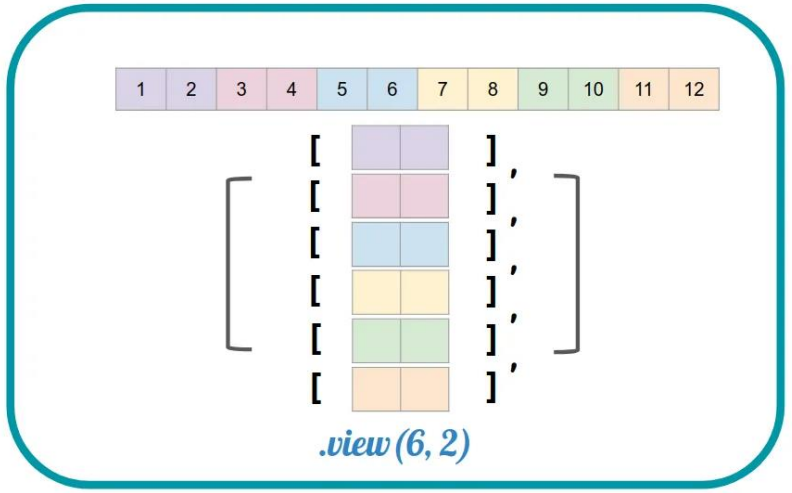
```
ones = torch.zeros(2, 2) + 1
twos = torch.ones(2, 2) * 2
threes = (torch.ones(2, 2) * 7 - 1) / 2
fours = twos ** 2
```

```
torch.abs()
torch.linalg.norm()
torch.matmul()
...
```

## Shape Manipulation

```
torch.squeeze()
torch.unsqueeze()

torch.view()
torch.reshape()
torch.permute()
torch.transpose()
...
```




```
>>> x = torch.zeros(2, 1, 2, 1, 2)
>>> x.size()
torch.Size([2, 1, 2, 1, 2])
>>> y = torch.squeeze(x)
>>> y.size()
torch.Size([2, 2, 2])
>>> y = torch.squeeze(x, 0)
>>> y.size()
torch.Size([2, 1, 2, 1, 2])
```

# Autograd Mechanism

## Autograd

```
import torch

a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
```

## Freezing Model Weights

```
from torch import nn, optim

model = resnet18(weights=ResNet18_Weights.DEFAULT)

# Freeze all the parameters in the network
for param in model.parameters():
    param.requires_grad = False
```

## Gradient Example 1

$$Q = 3a^3 - b^2$$

```
Q = 3*a**3 - b**2
```

$$\frac{\partial Q}{\partial a} = 9a^2$$

$$\frac{\partial Q}{\partial b} = -2b$$

```
a = x + 5
b = a ** 3
c = b + 1
y = c.sum()/3
print("Y", y)
```

```
Y tensor(229., grad_fn=<DivBackward0>)
```

```
y.backward()
print(x.grad)
```

```
tensor([25., 36., 49.])
```

## Gradient Example 2

$$y = \frac{1}{3}\sum_i^3 \left[(x_i + 5)^3 + 1\right]$$

$$x = [0, 1, 2]$$

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial c_i}\frac{\partial c_i}{\partial b_i}\frac{\partial b_i}{\partial a_i}\frac{\partial a_i}{\partial x_i}$$

$$\frac{\partial y}{\partial x_i} = (x_i + 5)^2$$

$$\partial y/\partial \mathbf{x} = [25, 36, 49]$$

$a = x + 5$ $\quad c = b + 1$

- $b = a^3$ $\quad y = c/3$

$$\frac{\partial a_i}{\partial x_i} = 1, \qquad \frac{\partial b_i}{\partial a_i} = 3 \cdot a_i^2 \qquad \frac{\partial c_i}{\partial b_i} = 1 \qquad \frac{\partial y}{\partial c_i} = \frac{1}{3}$$

# Building Layers

## Fully Connected

CLASS `torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

## Pooling

CLASS `torch.nn.MaxPool2d(kernel_size, stride=None, ceil_mode=False)` [SOURCE]

## Convolutional

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)` [SOURCE]

## Memory Blocks

CLASS `torch.nn.LSTM(*args, **kwargs)` [SOURCE]

## Activation Functions

CLASS `torch.nn.ReLU(inplace=False)` [SOURCE]

Applies the rectified linear unit function element-wise:

$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

## Losses

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0)` [SOURCE]

# Building Models

## torch.nn.Module

```
class MLP(nn.Module):
    def __init__(self, input_size, np = 64):
        super(MLP, self).__init__()
        self.Lin1 = nn.Linear(input_size, np)
        self.Lin2 = nn.Linear(np, np)
        self.Lin3 = nn.Linear(np, 1)
        self.ReLU = nn.ReLU()
        self.Sig = nn.Signoid()

    def forward(self,x):
        x = self.ReLU(self.Lin1(x))
        x = self.ReLU(self.Lin2(x))
        return self.Sig(self.Lin4(x))
```

## torch.nn.Sequential

```
class MLP(nn.Module):
    def __init__(self, input_size, np = 64):
        super(MLP, self).__init__()
        self.Net = nn.Sequential(
            torch.nn.Linear(input_size, np),
            nn.ReLU(),
            torch.nn.Linear(np, np),
            nn.ReLU(),
            torch.nn.Linear(np, 1),
            nn.Sigmoid(),
            )

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

## Optimizer

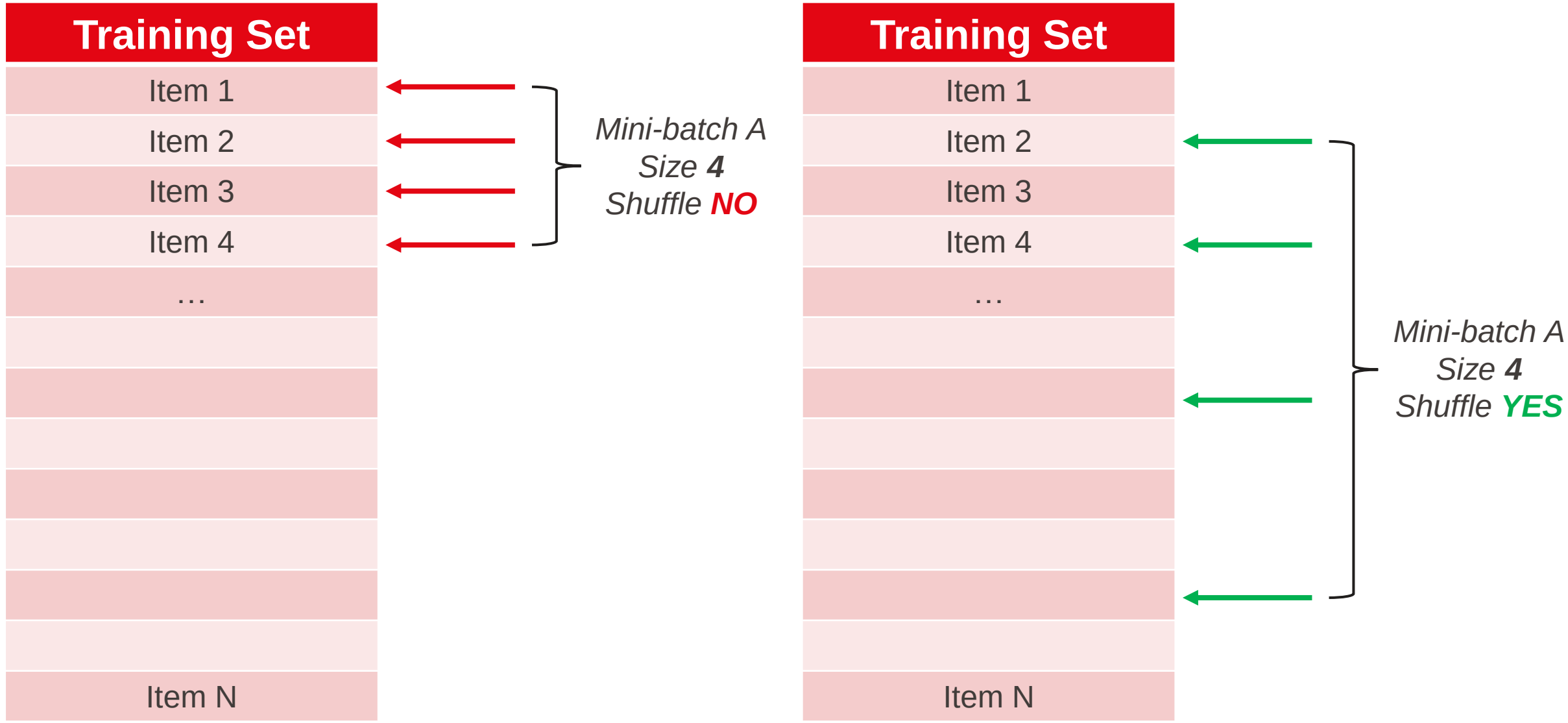
```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

## Loss Function

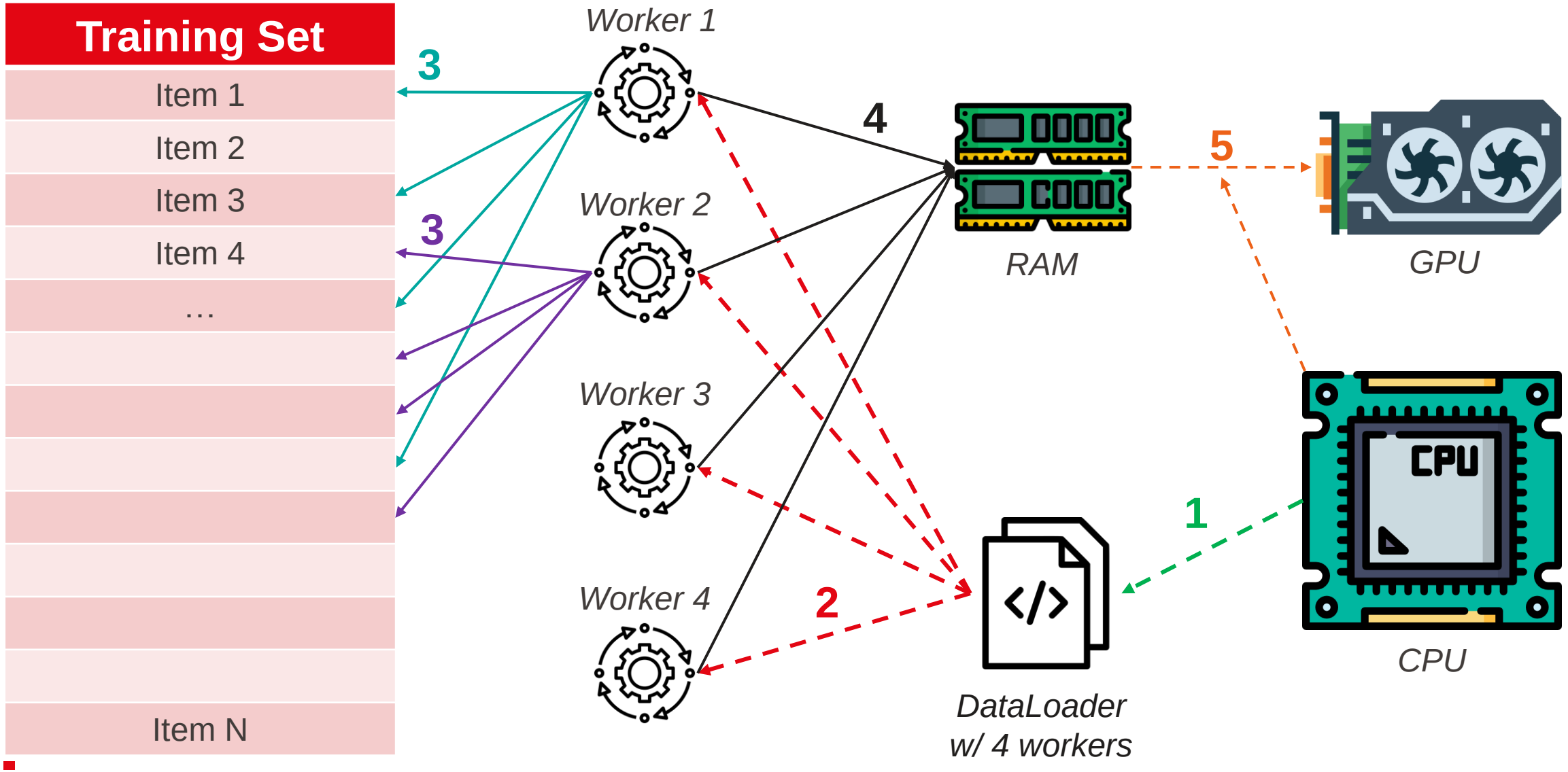
```
>>> loss = nn.MSELoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
```

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

# Mini-batches for training



# Mini-batches for training

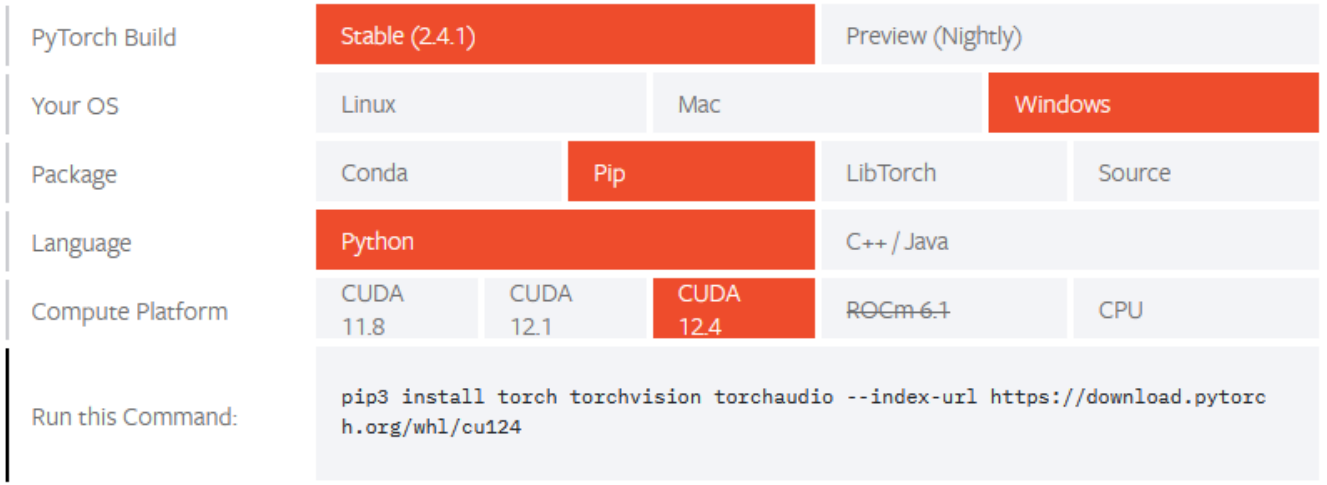


# GPU Acceleration

```
cuda = torch.device('cuda')     # Default CUDA device
```

```
x = torch.tensor([1., 2.], device=cuda0)
# x.device is device(type='cuda', index=0)
```

https://pytorch.org/get-started/locally/




```
if args.cpu:
    device = torch.device("cpu")
elif torch.cuda.is_available():
    device = torch.device("cuda")
elif torch.backends.mps.is_available():
    device = torch.device("mps")  # Apple Silicon
else:
    device = torch.device("cpu")
```

*Automatic selection of GPU acceleration*
*Manual override with `--cpu` CLI argument*

⚠ Make sure to install the right version

⚠ Do not forget **`model.to(device)`**

⚠ Move data to GPU inside the main loop, not inside the **`Dataset`** class

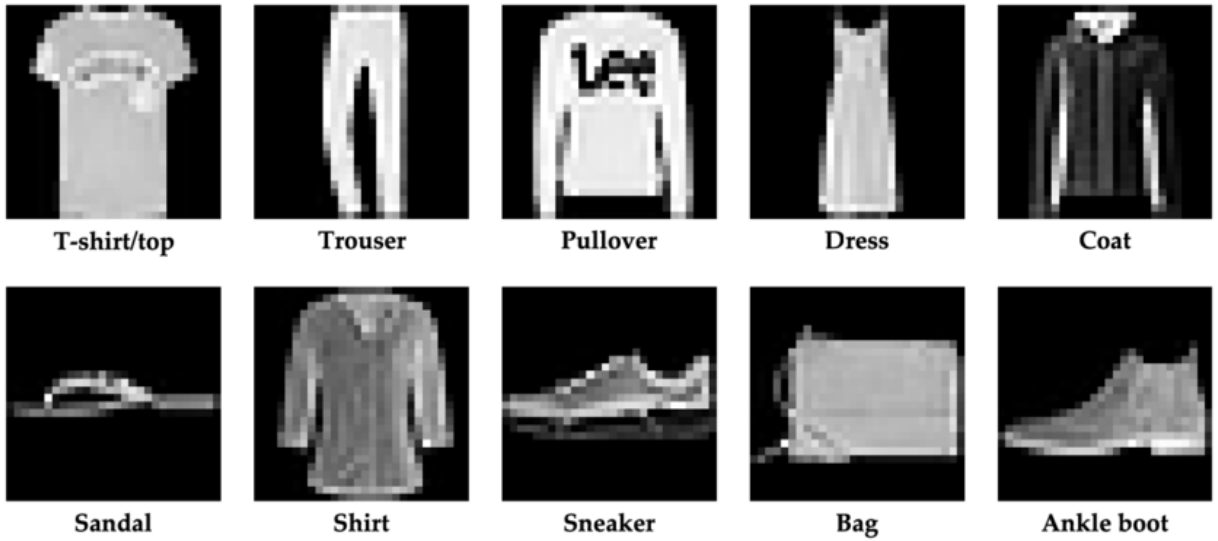
# Example Training Loop

```
# Create datasets for training & validation, download if necessary
training_set = torchvision.datasets.FashionMNIST('./data', train=True, transform=transform,
validation_set = torchvision.datasets.FashionMNIST('./data', train=False, transform=transform,

# Create data loaders for our datasets; shuffle for training, not for validation
training_loader = torch.utils.data.DataLoader(training_set, batch_size=4, shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_set, batch_size=4, shuffle=False)

model = GarmentClassifier()

# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```




*10 digits as output*

*Flatten the convolved images*

```
# PyTorch models inherit from torch.nn.Module
class GarmentClassifier(nn.Module):
    def __init__(self):
        super(GarmentClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

# Example Training Loop

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

*Cycle through the data*

*Could use:*

```
inputs = inputs.to(device)
labels = labels.to(device)
```

`loss_fn` *can be* `CrossEntropyLoss()`

*Key step in the learning process*

# Questions