Introduction to Pytorch

Thomas Gerald

April 2, 2026

Laboratoire Interdisciplinaire des Sciences du Numérique – LISN, CNRS

Pytorch and Computational graph

- **Dynamic computation:** The graph is re-computed for each input (similarly to what we implement previous week)
- **Eager:** Each operation is immediatly computed (**no** lazy computation)

$\rightarrow$ Since Pytorch 2 an option to compile the code is possible $\implies$ Not eager computation

Interface

- Python (used in the exercise labs)
- C++

Pytorch components

- `torch`: Tensors (a node in the computational graph)
- `torch.nn.functional`: functions to manipulates the Tensors
- `torch.nn`: neural network components/blocks (e.g. Linear/Affine transformation and more complex blocks)
- `torch.optim`: optimization methods (e.g. SGD, Adam, RMSProp,...)

PyTorch Tensors

Introduction to PyTorch: the tensors

torch.Tensor

- **dtype**: the type of the tensor
- **shape**: shape/size of the tensor
- **device**: where the tensor is located (on the cpu/gpu)
- **requires_grad**: do we need to backpropagate gradient to this tensor (e.g, input rarely needs it)

Create an uninitialized tensor:

```
import torch

t = torch.empty(
    (4, 2, 2), # shape
    dtype=torch.float,
    device="cpu",
    requires_grad=True
)
```

```
tensor([[[ 2.1968e-19,  4.5067e-41],
         [ 2.1968e-19,  4.5067e-41]],
        [[-7.6702e+08,  3.3659e-41],
         [ 0.0000e+00,  0.0000e+00]],
        [[ 0.0000e+00,  0.0000e+00],
         [-7.6806e+08,  3.3659e-41]],
        [[-9.8235e+07,  3.3659e-41],
         [ 3.0484e-23,  4.5067e-41]]],
         requires_grad=True)
```

Create an initialized tensor:

```
torch.zeros((4, 2, 2), dtype=torch.float, requires_grad=True)
torch.ones((4, 2, 2), dtype=torch.float, requires_grad=True)
torch.rand((4, 2, 2), dtype=torch.float, requires_grad=True)
```

Using [...]_like:

Create a new tensor with the same attributes than the one given in arguments:

Introduction to PyTorch: Create tensors from tensors

Using [...]_like:

Create a new tensor with the same attributes than the one given in arguments:

- Specific attributes can be overridden (type, value)

Using [...]_like:

Create a new tensor with the same attributes than the one given in arguments:

- Specific attributes can be overridden (type, value)
- Shape are the same

Introduction to PyTorch: Create tensors from tensors

Using [...]_like:

Create a new tensor with the same attributes than the one given in arguments:

- Specific attributes can be overridden (type, value)
- Shape are the same

```
t2 = torch.zeros_like(t)
t_bool = torch.zeros_like(t, dtype=torch.bool)
t3 = torch.full_like(t2,1) # all value are one
```

Introduction to PyTorch: Create tensors from tensors

Using clone:

```
t1 = torch.ones((1,))
t2 = t1.clone()
t2[0] = 3
print(t1, t2)
```

Introduction to PyTorch: Create tensors from tensors

Using clone:

```
t1 = torch.ones((1,))
t2 = t1.clone()
t2[0] = 3
print(t1, t2)
```

Output ?

```
tensor([1.]) tensor([3.])
```

Using clone:

```
t1 = torch.ones((1,))
t2 = t1.clone()
t2[0] = 3
print(t1, t2)
```

Output ?

```
tensor([1.]) tensor([3.])
```

Using new ?

Create a tensor of the same type as the current one (on the same device)

```
x = torch.ones((2,), device="cpu")
y = x.new()
```

Introduction to PyTorch: Create tensors from tensors

Using clone:

```
t1 = torch.ones((1,))
t2 = t1.clone()
t2[0] = 3
print(t1, t2)
```

Output ?

```
tensor([1.]) tensor([3.])
```

Using new ?

Create a tensor of the same type as the current one (on the same device)

```
x = torch.ones((2,), device="cpu")
y = x.new()
```

$\rightarrow$

```
tensor([])
```

Using clone:

```
t1 = torch.ones((1,))
t2 = t1.clone()
t2[0] = 3
print(t1, t2)
```

Output ?

```
tensor([1.]) tensor([3.])
```

Using new ?

Create a tensor of the same type as the current one (on the same device)

```
x = torch.ones((2,), device="cpu")
y = x.new()
```

$\rightarrow$

```
tensor([])
```

```
x = torch.ones((2,))
y = x.new(3,2)
```

Using clone:

```
t1 = torch.ones((1,))
t2 = t1.clone()
t2[0] = 3
print(t1, t2)
```

Output ?

```
tensor([1.]) tensor([3.])
```

Using new ?

Create a tensor of the same type as the current one (on the same device)

```
x = torch.ones((2,), device="cpu")
y = x.new()
```

→

```
tensor([])
```

```
x = torch.ones((2,))
y = x.new(3,2)
```

→

```
tensor([[-9.0229e+08,  3.3659e-41],
        [-7.6788e+08,  3.3659e-41],
        [-9.2275e+17,  7.0065e-45]])
```

Introduction to PyTorch: Create tensors from data

From python list:

```
t1 = torch.tensor([0, 1, 2, 3], dtype=torch.long)
```

- Create a vector with integers 0,1,2,3
- Elements are long integers

Introduction to PyTorch: Create tensors from data

From python list:

```
t1 = torch.tensor([0, 1, 2, 3], dtype=torch.long)
```

- Create a vector with integers 0,1,2,3
- Elements are long integers

From iterables:

```
t2 = torch.tensor(range(10))
t3 = torch.tensor(np.array([1,2,3,4,5]))
```

- t2 is a vector of floats with values from 0 to 9
- t3 is a vector of floats with values from 1 to 5

Creating matrices:

```
t3 = torch.tensor([[0, 1], [2, 3]])
```

- First row: 0, 1
- Second row: 2, 3

Introduction to PyTorch: Create tensors from data

Creating matrices:

```
t3 = torch.tensor([[0, 1], [2, 3]])
```

- First row: 0, 1
- Second row: 2, 3

Creating random matrices:

```
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.randn((4, 4)) # standardized normal law
```

Copy base operations (Out-of-place)

- Create a new tensor, i.e. new memory is allocated to store the result
- Set backpropagation information if required (as in preivous exercise lab)
  if at least one of the inputs require grad

Introduction to PyTorch: operations

Copy base operations (Out-of-place)

- Create a new tensor, i.e. new memory is allocated to store the result
- Set backpropagation information if required (as in preivous exercise lab)
  if at least one of the inputs require grad

Inplace operations

- Modify the data og the tensor (`x.data`)
- Contain underscore in their names
- Can leads to error computing gradient
  - Forget forward value
  - Can break the backpropagation

Introduction to PyTorch: Copy operations

```python
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.rand((4, 4)) # standardized normal law
```

Introduction to PyTorch: Copy operations

```
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.rand((4, 4)) # standardized normal law
```

Addition

```
t3 = torch.add(t1, t2)
t3 = t1.add(t2)
t3 = t1 + t2
```

```
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.rand((4, 4)) # standardized normal law
```

Addition

```
t3 = torch.add(t1, t2)
t3 = t1.add(t2)
t3 = t1 + t2
```

Substraction

```
t3 = torch.sub(t1, t2)
t3 = t1.sub(t2)
t3 = t1 - t2
```

Introduction to PyTorch: Copy operations

```
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.rand((4, 4)) # standardized normal law
```

Addition

```
t3 = torch.add(t1, t2)
t3 = t1.add(t2)
t3 = t1 + t2
```

Substraction

```
t3 = torch.sub(t1, t2)
t3 = t1.sub(t2)
t3 = t1 - t2
```

Multiplication (Adamard)

```
t3 = torch.mul(t1, t2)
t3 = t1.mul(t2)
t3 = t1 * t2
```

Introduction to PyTorch: Copy operations

```python
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.rand((4, 4)) # standardized normal law
```

Addition

```python
t3 = torch.add(t1, t2)
t3 = t1.add(t2)
t3 = t1 + t2
```

Substraction

```python
t3 = torch.sub(t1, t2)
t3 = t1.sub(t2)
t3 = t1 - t2
```

Multiplication (Adamard)

```python
t3 = torch.mul(t1, t2)
t3 = t1.mul(t2)
t3 = t1 * t2
```

⚠ **Not matrix multiplication**

Division

```python
t3 = torch.div(t1, t2)
t3 = t1.div(t2)
t3 = t1 / t2
```

```
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.rand((4, 4)) # standardized normal law
```

Matrix multiplication

```
t3 = torch.matmul(t1, t2)
t3 = t1.matmul(t2)
t3 = t1 @ t2
```

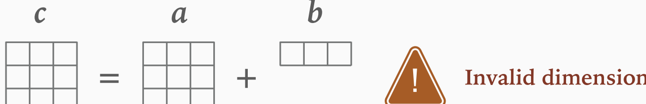
Introduction to PyTorch: Inplace operation

```
t1 = torch.rand((4, 4)) # uniform between 0 and 1
t2 = torch.rand((4, 4)) # standardized normal law
```

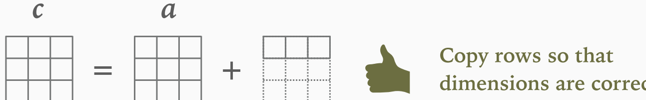
Inplace

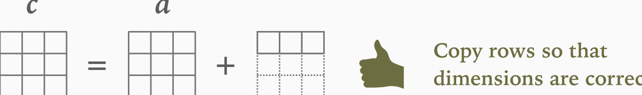
```
t1.add_(t2)
t1.sub_(t2)
t1.mul_(t2)
t1.div_(t2)
```

Introduction to PyTorch: Broadcasting



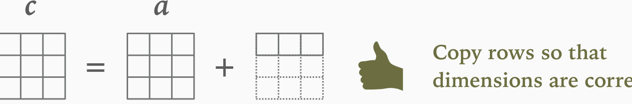
Introduction to PyTorch: Broadcasting





Explicit broadcasting

```
a = torch.rand((3, 3))
b = torch.rand((1, 3))
# explicitly copy the data
b.repeat((3, 1))
# implicit construction
# (no duplicated memory)
b.expand((3, -1))
```



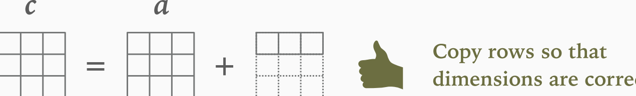
Explicit broadcasting

```
a = torch.rand((3, 3))
b = torch.rand((1, 3))
# explicitly copy the data
b.repeat((3, 1))
# implicit construction
# (no duplicated memory)
b.expand((3, -1))
```

Implicit broadcasting

Many operations will automatically broadcast

Introduction to PyTorch: Broadcasting



Explicit broadcasting

```
a = torch.rand((3, 3))
b = torch.rand((1, 3))
# explicitly copy the data
b.repeat((3, 1))
# implicit construction
# (no duplicated memory)
b.expand((3, -1))
```

Implicit broadcasting

Many operations will automatically broadcast

→ ⚠ **Read the doc**

```
a = torch.rand((3, 3))
b = torch.rand((1, 3))
c = a + b
```

https://pytorch.org/docs/stable/notes/broadcasting.html

Einstein sum:

Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.

$$ij, jk \rightarrow ik$$

i, j and k correspond to dimensions of tensors

Introduction to PyTorch: Einstein notation (Expert)

Einstein sum:

Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.

$$ij, jk \rightarrow ik$$

i, j and k correspond to dimensions of tensors

- The left side (before $\rightarrow$) corresponds to inputs

Einstein sum:

Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.

$$ij, jk \rightarrow ik$$

i, j and k correspond to dimensions of tensors

- The left side (before $\rightarrow$) corresponds to inputs
- The right side (after $\rightarrow$) corresponds to the output

Einstein sum:

Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.

$$ij, jk \rightarrow ik$$

i, j and k correspond to dimensions of tensors

- The left side (before $\rightarrow$) corresponds to inputs
- The right side (after $\rightarrow$) corresponds to the output
- Each input is separated by a comma in the inputs

Einstein sum:

Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.

$$ij, jk \rightarrow ik$$

i, j and k correspond to dimensions of tensors

- The left side (before $\rightarrow$) corresponds to inputs
- The right side (after $\rightarrow$) corresponds to the output
- Each input is separated by a comma in the inputs
- If same dimension are in the input, dim are multiplied

Einstein sum:

Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.

$$ij, jk \rightarrow ik$$

i, j and k correspond to dimensions of tensors

- The left side (before $\rightarrow$) corresponds to inputs
- The right side (after $\rightarrow$) corresponds to the output
- Each input is separated by a comma in the inputs
- If same dimension are in the input, dim are multiplied
- If dimension are in input and not in input it is summed in the output

Examples

```
# outer product
x = torch.randn(5)
y = torch.randn(4)
torch.einsum('i,j->ij', x, y)

# batch matrix multiplication
As = torch.randn(3, 2, 5)
Bs = torch.randn(3, 5, 4)
torch.einsum('bij,bjk->bik', As,
    Bs)
```

```
# permutation of last two
    dimensions
A = torch.randn(2, 3, 4, 5)
torch.einsum('...ij->...ji', A).
    shape

# equivalent to As * Bs
As = torch.randn(3, 2, 5)
Bs = torch.randn(3, 2, 5)
torch.einsum('bij,bij->bij', As,
    Bs)
```

⚠ **Use it only if you master pytorch** See https://pytorch.org/docs/stable/generated/torch.einsum.html

Introduction to PyTorch: Activation functions

Out-of-place

```
t2 = torch.relu(t1)
t2 = torch.tanh(t1)
t2 = torch.sigmoid(t1)
```

Inplace

```
torch.relu_(t1)
torch.tanh_(t1)
torch.sigmoid_(t1)
```

```
a = torch.FloatTensor([2])
b = torch.FloatTensor([4])
c = a * b

c.backward()
```

→ ⚠ **RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn**

→ Does not register the gradient computation function

```
a = torch.FloatTensor([2])
b = torch.FloatTensor([4])
a.requires_grad = True
c = a * b
# after the call a.grad
    contains the gradient
c.backward()
```

→ `a.grad` get the gradient (4)

Providing a gradient to backward

We can provide an initial gradient to the backward

```
a = torch.FloatTensor([2])
b = torch.FloatTensor([4])
a.requires_grad = True
c = a * b
# after the call a.grad
    contains the gradient
c.backward(torch.tensor([2.]))
```

→ a.grad get the gradient (8)

Double backward ?

```
a = torch.FloatTensor([2])
b = torch.FloatTensor([4])
a.requires_grad = True
c = a * b
# after the call a.grad
    contains the gradient
c.backward()

# let's do something else...
b2 = torch.FloatTensor([2])
c2 = a * b2
c2.backward()
```

→

`a.grad` get the gradient (6)

⚠ **The gradient is accumulated !!!**

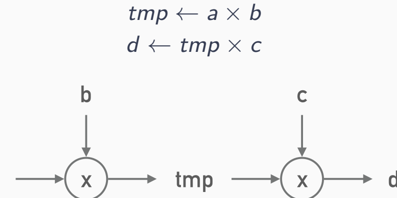
→ use `a.grad.zero_()` to set the gradient to zero !!

Let compute d :

$$
\begin{aligned}
tmp &\leftarrow a \times b \\
d &\leftarrow tmp \times c
\end{aligned}
$$

Let compute d **:**

$$tmp \leftarrow a \times b$$
$$d \leftarrow tmp \times c$$



Let compute d :

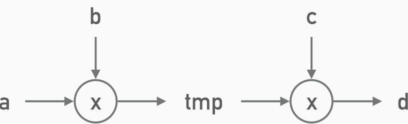
$$tmp \leftarrow a \times b$$
$$d \leftarrow tmp \times c$$

Back-propagation to tmp

- We need the value of `d`
- We don't need the of `tmp`

Let compute d :

$$tmp \leftarrow a \times b$$
$$d \leftarrow tmp \times c$$




```
a = torch.rand((1,), requires_grad=True)
b = torch.rand((1,), requires_grad=True)
c = torch.rand((1,), requires_grad=True)
tmp = a * b
d = tmp * c
# erase the data of tmp
torch.zero_(tmp)
d.backward()
```

Back-propagation to tmp

- We need the value of `d`
- We don't need the of `tmp`

Introduction to PyTorch: Backpropagation and inplace operation

Let compute d :

$$tmp \leftarrow a \times b$$
$$d \leftarrow tmp \times c$$

```
a = torch.rand((1,), requires_grad=True)
b = torch.rand((1,), requires_grad=True)
c = torch.rand((1,), requires_grad=True)
tmp = a * b
d = tmp * c
# erase the data of tmp
torch.zero_(tmp)
d.backward()
```

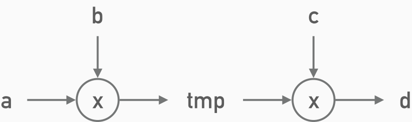
Back-propagation to tmp

- We need the value of `d`
- We don't need the of `tmp`

→ **⚠ RuntimeError: one of the variables needed for gradient computation has been modified by an inplace operation**

Let compute d :

$$tmp \leftarrow a \times b$$
$$d \leftarrow tmp \times c$$




```
a = torch.rand((1,), requires_grad=True)
b = torch.rand((1,), requires_grad=True)
c = torch.rand((1,))
tmp = a * b
d = tmp * c
# erase the data of tmp
torch.zero_(tmp)
d.backward()
```

Back-propagation to tmp

- We need the value of `d`
- We don't need the of `tmp`

$\rightarrow$ It works !!

Introduction to PyTorch: Where are stored the gradient function ?

```
a = torch.FloatTensor([2])
b = torch.FloatTensor([4])
a.requires_grad = True
c = a * b
# after the call a.grad contains the gradient
c.backward()
```

- `d.grad_fn` is the function (¡MulBackward0¿)
- `d.grad_fn._saved_[]` contains saved tensors
- `d.grad_fn.next_functions` contains next function to calls

Neural Network

torch.nn.Module

To build a neural network, we store in a module:

- Parameters of the network
- Other modules

Benefits

- Execution mode: we can set the network in training or in test mode (e.g. to automatically apply or discard dropout)
- Move whole network to a device
- Retrieve all learnable parameters of the network (as in previous exercise lab)

Single Hidden Layer (1st version)

```
class HiddenLayer(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.W = torch.nn.Parameter(
            torch.empty(input_dim, output_dim)
            )
        self.bias = torch.nn.Parameter(torch.empty(output_dim, 1))
    def forward(self, x):
        z = x @ self.W.data.transpose(0, 1) \
        + self.bias.data.transpose(0, 1)
        return torch.relu(z)


nn = HiddenLayer(10, 2)
x = torch.rand((64, 10))
# Shape of y is (64, 5)
y = nn(x)
```

⚠ **Do not call forward explicitly**

Single Hidden Layer (2nd version)

```
class HiddenLayer(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, inputs):
        z = self.linear(inputs)
        return torch.relu(z)


nn = HiddenLayer(10, 2)
x = torch.rand((64, 10))
# Shape of y is (64, 5)
y = nn(x)
```

⚠ **Do not call forward explicitly**

Introduction to PyTorch: One layer NN

- Extend `torch.nn.module`
- Attributes requesting update should be Module

One layer neural network

```
class SimpleNetwork(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, n_classes):
        super().__init__()
        self.z_proj = torch.nn.Linear(input_dim, hidden_dim)
        self.output_proj = torch.nn.Linear(hidden_dim, n_classes)
    def forward(self, inputs):
        z = torch.relu(self.z_proj(inputs))
        o = self.output_proj(z)
        return o
```

Introduction to PyTorch: ModuleList

```
class SimpleNetwork2(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2,
        n_classes):
        super().__init__()
        self.z_projs = [] # Never do this !!!!
        self.z_projs.append(torch.nn.Linear(input_dim, hidden_dim1))
        self.z_projs.append(torch.nn.Linear(hidden_dim1, hidden_dim2
    ))
```

- ⚠ **Never create a Python list of modules in a module**
- At least if you need to get the parameters (later)

Introduction to PyTorch: ModuleList

```
class SimpleNetwork2(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2,
        n_classes):
        super().__init__()
        self.z_projs = [] # Never do this !!!!
        self.z_projs.append(torch.nn.Linear(input_dim, hidden_dim1))
        self.z_projs.append(torch.nn.Linear(hidden_dim1, hidden_dim2
    ))
```

Module inspection

Pytorch will automatically inspect attributes of modules, e.g. to extract all parameters of a network. However, only appropriate containers will be recursively inspected:

- `torch.nn.Module`
- `torch.nn.ModuleList`
- `torch.nn.ParameterDict`
- `torch.nn.ParameterList`

Introduction to PyTorch: ModuleList

```
class SimpleNetwork2(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2,
     n_classes):
        super().__init__()
        self.z_projs = torch.nn.ModuleList()
        self.z_projs.append(torch.nn.Linear(input_dim, hidden_dim1))
        self.z_projs.append(torch.nn.Linear(hidden_dim1, hidden_dim2
     ))
        self.output_proj = torch.nn.Linear(hidden_dim2, n_classes)
    def forward(self, inputs):
        z = inputs
        for nn in self.z_projs:
        z = torch.relu(nn(z))
        o = self.output_proj(z)
        return o
```

- ✅ If you need a list of modules use ModuleList

Optimization

Introduction to PyTorch: Loss function

```
loss_builder = torch.nn.NLLLoss(reduction="mean")
loss = loss_builder(y, gold)

epoch_loss += loss.item() ## use item
```

- If you only want to show a value $\rightarrow$ use item
- Else the computation graph not entirely release (you created a new node)

Introduction to PyTorch: Parameters Update

```
nn = SimpleNetwork2(10, 250, 100, 2)
optimizer = torch.optim.SGD(
        nn.parameters(),
        lr=0.01,
        weight_decay=0.0001,
        momentum=0.9
    )
# Forward
x = torch.rand((64, 10))
y = nn(x)
loss = loss_builder(y, gold)
# backward
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Introduction to PyTorch: optimizer

- Many optimizers are available out of the box

Introduction to PyTorch: optimizer

- Many optimizers are available out of the box
- More recent techniques are often available on GitHub (RAdam, ...)

Introduction to PyTorch: optimizer

- Many optimizers are available out of the box
- More recent techniques are often available on GitHub (RAdam, . . .)

```
torch.optim.Adam(nn.parameters())
torch.optim.Adadelta(nn.parameters())
torch.optim.Adagrad(nn.parameters())
```

Introduction to PyTorch:Train and eval

```python
class SimpleNetwork2(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim1, n_classes):
        super().__init__()
        self.seq = torch.nn.Sequential(
            torch.nn.Linear(input_dim, hidden_dim1),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.2), # 10% component set to 0
            torch.nn.Linear(hidden_dim1, n_classes)
        )
    def forward(self, inputs):
        return self.seq(inputs)
```

Introduction to PyTorch:Train and eval

```
class SimpleNetwork2(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim1, n_classes):
        super().__init__()
        self.seq = torch.nn.Sequential(
            torch.nn.Linear(input_dim, hidden_dim1),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.2), # 10% component set to 0
            torch.nn.Linear(hidden_dim1, n_classes)
        )
    def forward(self, inputs):
        return self.seq(inputs)

nn = SimpleNetwork2(10, 250, 100, 2)
x = torch.rand((64, 10))
nn.train()
y = nn(x)
nn.eval()
y = nn(x)
```

Introduction to PyTorch: Train and eval

```
class SimpleNetwork2(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim1, n_classes):
        super().__init__()
        self.seq = torch.nn.Sequential(
            torch.nn.Linear(input_dim, hidden_dim1),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.2), # 10% component set to 0
            torch.nn.Linear(hidden_dim1, n_classes)
        )
    def forward(self, inputs):
        return self.seq(inputs)

nn = SimpleNetwork2(10, 250, 100, 2)
x = torch.rand((64, 10))
nn.train()
y = nn(x)
nn.eval()
y = nn(x)

nn = SimpleNetwork2(10, 250, 100, 2)
x = torch.rand((64, 10))
nn.train()
y = nn(x)
```