Deep Learning
Automatic Differentiation and Pytorch
Recall: Optimizing ML Models
### A Recipe for Machine Learning

1. Given training data:

\[
\{x_i, y_i\}_{i=1}^{N}
\]

2. Choose each of these:
   - Decision function
   \[\hat{y} = f_\theta(x_i)\]
   - Loss function
   \[\ell(\hat{y}, y_i) \in \mathbb{R}\]

3. Define goal:

\[
\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)
\]

4. Train with SGD:
(take small steps opposite the gradient)

\[
\theta^{(t+1)} = \theta^{(t)} - \eta_i \nabla \ell(f_\theta(x_i), y_i)
\]
Gradients

1. Given training data
   \[ \{ x_i, y_i \}_{i=1}^{N} \]

2. Choose each of the following:
   - Decision function
     \[ \hat{y} = f_\theta(x_i) \]
   - Loss function
     \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

**Backpropagation** can compute this gradient!
And it’s a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!

\[ \theta(\hat{y}(x_i), y_i) - \eta_t \nabla \ell(f_\theta(x_i), y_i) \]
How to Compute Gradients
Motivations

● Backpropagation in NN models
  ○ Implementing Backprop by hand is like programming in Assembly. You can do it, but… why?
  ○ Still, if you know assembly you are better off!
Terminology

● **Automatic differentiation** (autodiff) refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value.
  ○ In this lecture, we focus on reverse mode autodiff. There is also a forward mode, which is for computing directional derivatives.
● **Backpropagation** is the special case of autodiff applied to neural nets
  ○ But in machine learning, we often use backprop synonymously with autodiff
● **Autograd** is the name of a particular autodiff package.
  ○ But lots of people, including the PyTorch developers, got confused and started using “autograd” to mean “autodiff”
What Autodiff is not

- Autodiff is not finite differences.
  - Finite differences are expensive, since you need to do a forward pass for each derivative.
  - It also induces huge numerical error.
- Autodiff is both efficient (linear in the cost of computing the value) and numerically stable.
What Autodiff is not

- Autodiff is not symbolic differentiation (e.g. Mathematica).
  - Symbolic differentiation can result in complex and redundant expressions.
- The goal of autodiff is not a formula, but a procedure for computing derivatives.
What Autodiff is

- An autodiff system will convert the program into a sequence of **primitive operations** which have specified routines for computing derivatives.
  - In this representation, backprop can be done in a completely mechanical way.

<table>
<thead>
<tr>
<th>Original program:</th>
<th>Sequence of primitive operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ z = wx + b ]</td>
<td>[ t_1 = wx ]</td>
</tr>
<tr>
<td>[ y = \frac{1}{1 + \exp(-z)} ]</td>
<td>[ z = t_1 + b ]</td>
</tr>
<tr>
<td>[ \mathcal{L} = \frac{1}{2}(y - t)^2 ]</td>
<td>[ t_3 = -z ]</td>
</tr>
<tr>
<td></td>
<td>[ t_4 = \exp(t_3) ]</td>
</tr>
<tr>
<td></td>
<td>[ t_5 = 1 + t_4 ]</td>
</tr>
<tr>
<td></td>
<td>[ y = 1/t_5 ]</td>
</tr>
<tr>
<td></td>
<td>[ t_6 = y - t ]</td>
</tr>
<tr>
<td></td>
<td>[ t_7 = t_6^2 ]</td>
</tr>
<tr>
<td></td>
<td>[ \mathcal{L} = t_7/2 ]</td>
</tr>
</tbody>
</table>
Computation Graph
Compution Graph: Logistic Regression

```python
def logistic(z):
    return 1. / (1. + np.exp(z))
```

$Z = 1.5$
$Y = \text{logistic}(1.5)$
Original program:

\[
\begin{align*}
  z &= wx + b \\
  y &= \frac{1}{1 + \exp(-z)} \\
  \mathcal{L} &= \frac{1}{2} (y - t)^2
\end{align*}
\]
(Reverse Mode) Autodiff starts at an output of the graph and moves towards the beginning.

At each node, it merges all paths which originated at that node.

Example: \((a+b)(b+1)\)
Backprop on the Computation Graph: Reverse Mode

<table>
<thead>
<tr>
<th>SPLIT</th>
<th>ADDITION</th>
<th>FUNCTION</th>
<th>MATRIX MULTIPLY</th>
<th>HADAMARD PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a; b = a$</td>
<td>$c = a + b$</td>
<td>$b = f(a)$</td>
<td>$b = Wa$</td>
<td>$c = a \ast b$</td>
</tr>
<tr>
<td>$\delta_{a} = \delta_{b} + \delta_{c}$</td>
<td>$\delta_{a} = \delta_{c}; \delta_{b} = \delta_{c}$</td>
<td>$\delta_{a} = \delta_{b} \ast f'(a)$</td>
<td>$\delta_{a} = \delta_{b} \ast W^{T}$</td>
<td>$\delta_{a} = \delta_{c} \ast b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\delta_{b} = \delta_{c} \ast a$</td>
</tr>
</tbody>
</table>

*SAPIENZA Università di Roma*
Things can get complicated :)
pyTorch
(slides inspired by this deck)
Do you have pyTorch installed?

```python
>>> import numpy as np
>>> import torch
>>> import sys
>>> import matplotlib
>>> print(f'Python version: {sys.version}')
Python version: 3.8.1 | packaged by conda-forge | (default, Jan 29 2020, 14:55:04) [GCC 7.3.0]

>>> print(f'Numpy version: {np.version.version}')
Numpy version: 1.17.5

>>> print(f'PyTorch version: {torch.version.__version__}')
PyTorch version: 1.4.0

>>> print(f'Matplotlib version: {matplotlib.__version__}')
Matplotlib version: 3.1.2

>>> print(f'GPU present: {torch.cuda.is_available()}')
GPU present: False
```
pyTorch packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>torch</td>
<td>The top-level PyTorch package and tensor library.</td>
</tr>
<tr>
<td>torch.nn</td>
<td>A subpackage that contains modules and extensible classes for building neural networks.</td>
</tr>
<tr>
<td>torch.autograd</td>
<td>A subpackage that supports all the differentiable Tensor operations in PyTorch.</td>
</tr>
<tr>
<td>torch.nn.functional</td>
<td>A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations.</td>
</tr>
<tr>
<td>torch.optim</td>
<td>A subpackage that contains standard optimization operations like SGD and Adam.</td>
</tr>
<tr>
<td>torch.utils</td>
<td>A subpackage that contains utility classes like data sets and data loaders that make data preprocessing easier.</td>
</tr>
<tr>
<td>torchvision</td>
<td>A package that provides access to popular datasets, model architectures, and image transformations for computer vision.</td>
</tr>
</tbody>
</table>
torch.tensor

- PyTorch’s tensors are very similar to NumPy’s ndarrays
  - but they have a **device** attached, 'cpu', 'cuda', or 'cuda:X'
- They might require gradients

```python
>>> t = torch.tensor([1,2,3], device='cpu',
                    requires_grad=False, dtype=torch.float32)
```

```python
>>> print(t.dtype)
torch.float32
```

```python
>>> print(t.device)  
cpu
```

```python
>>> print(t.requires_grad)  
False
```

```python
>>> t2 = t.to(torch.device('cuda'))
```

```python
>>> t3 = t.cuda()  # or you can use shorthand
```

```python
>>> t4 = t.cpu()
```
### pyTorch Data Types

<table>
<thead>
<tr>
<th>Data type</th>
<th>dtype</th>
<th>CPU tensor</th>
<th>GPU tensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating point</td>
<td>torch.float32 or torch.float32</td>
<td>torch.FloatTensor</td>
<td>torch.cuda.FloatTensor</td>
</tr>
<tr>
<td>64-bit floating point</td>
<td>torch.float64 or torch.double</td>
<td>torch.DoubleTensor</td>
<td>torch.cuda.DoubleTensor</td>
</tr>
<tr>
<td>16-bit floating point</td>
<td>torch.float16 or torch.float16</td>
<td>torch.HalfTensor</td>
<td>torch.cuda.HalfTensor</td>
</tr>
<tr>
<td>8-bit integer (unsigned)</td>
<td>torch.int8</td>
<td>torch.ByteTensor</td>
<td>torch.cuda.ByteTensor</td>
</tr>
<tr>
<td>8-bit integer (signed)</td>
<td>torch.int8</td>
<td>torch.CharTensor</td>
<td>torch.cuda.CharTensor</td>
</tr>
<tr>
<td>16-bit integer (signed)</td>
<td>torch.int16 or torch.int16</td>
<td>torch.ShortTensor</td>
<td>torch.cuda.ShortTensor</td>
</tr>
<tr>
<td>32-bit integer (signed)</td>
<td>torch.int32 or torch.int32</td>
<td>torch.IntTensor</td>
<td>torch.cuda.IntTensor</td>
</tr>
<tr>
<td>64-bit integer (signed)</td>
<td>torch.int64 or torch.int64</td>
<td>torch.LongTensor</td>
<td>torch.cuda.LongTensor</td>
</tr>
</tbody>
</table>

Conversion in numpy and in PyTorch:

```python
new_array  = old_array.astype(np.int8)  # numpy array
new_tensor = old_tensor.to(torch.int8)  # torch tensor
```

Remarks: Almost always torch.float32 or torch.int64 are used. Half does not work on CPUs and on many GPUs (hardware limitation).
Creating Tensors

- **eye**: creating diagonal matrix / tensor
- **zeros**: creating tensor filled with zeros
- **ones**: creating tensor filled with ones
- **linspace**: creating linearly increasing values
- **arange**: linearly increasing integers

```python
>>> torch.eye(3, dtype=torch.double)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]], dtype=torch.float64)
>>> torch.arange(6)
tensor([0, 1, 2, 3, 4, 5])
```
pyTorch functions, dimensionality

```python
x.size()  # return tuple-like object of dimensions, old codes
x.shape  # return tuple-like object of dimensions, numpy style
x.ndim   # number of dimensions, also known as .dim()

x.view(a,b,...)  # equivalent with .view()
x.view(-1,a)  # swaps dimensions a and b
x.reshape(a,b,...)  # permutes dimensions; missing in numpy
x.transpose(a,b)  # tensor with added axis; missing in numpy
x.permute(*dims)  # (a,b,c) tensor -> (a,b,1,c) tensor; missing in numpy
x.unsqueeze(dim)  # concatenates tensors along dim
x.unsqueeze(dim=2)

torch.cat(tensor_seq, dim=0)

# For instance:
>>> t = torch.arange(6)
tensor([[ 0, 1, 2, 3, 4, 5]])

>>> t.reshape(2,3)  # same as t.view(2,3) or t.view(2,-1)
tensor([[ 0, 1, 2],
        [ 3, 4, 5]])

>>> t.reshape(2,3).unsqueeze(1)
tensor([[[ 0, 1, 2]],
        [[ 3, 4, 5]]])

>>> t.reshape(2,3).unsqueeze(1).shape
torch.Size([2, 1, 3])
```
Indexing

- Standard numpy indexing works:

```python
>>> t = torch.arange(12).reshape(3,4)
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> t[1,1:3]
tensor([5, 6])
>>> t[:,:] = 0 # fill everything with 0, a.k.a. t.fill_(0)
tensor([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```
Memory: Sharing vs. Copying

- **Copy Data:**
  - `torch.Tensor()`
  - `torch.tensor()`
  - `torch.clone()`
  - *type casting*

- **Share Data**
  - `torch.as_tensor()`
  - `torch.from_numpy()`
  - `torch.view()`
  - `torch.reshape()`

Most shape changing operators keep data.
Memory: Sharing vs. Copying

- How to test it?
  - create a tensor
  - copy/clone/view it
  - modify an element
  - compare the elements

```python
>>> a = np.arange(6)  # [0,1,2,3,4,5]
>>> t = torch.from_numpy(a)
>>> t[2] = 11
>>> t
    tensor([ 0,  1, 11,  3,  4,  5])
>>> a
    array([ 0,  1, 11,  3,  4,  5]) # Changed the underlying numpy array too!
>>> b = a.copy()
>>> p = t.clone()
>>> t[0] = 7
    # a, t change, b, p remain intact.
```
Creating Instances of \texttt{torch.Tensor} w/o Data

```python
>>> torch.eye(2)
tensor([[1., 0.],
        [0., 1.]]
>>> torch.zeros(2,2)
tensor([[0., 0.],
        [0., 0.]]
>>> torch.ones(2,2)
tensor([[1., 1.],
        [1., 1.]]

>>> torch.rand(2,2)
tensor([[0.6849, 0.1091],
        [0.4953, 0.8975]])

>>> torch.empty(2,2) # NEVER USE IT! Creates uninitialized tensor.
tensor([[ -2.2112e-16,  3.0693e-41],
        [ -3.0981e-16,  3.0693e-41]])

>>> torch.arange(6)
tensor([0, 1, 2, 3, 4, 5])
```
Interacting with numpy

```python
>>> import imageio
>>> img = imageio.imread('example.png')  # reading data from disk
>>> t = torch.from_numpy(a)  # input from numpy array
>>> out = model(t)  # processing
>>> result = out.numpy()  # converting back to numpy

# tuples, lists, arrays, etc. can be converted automatically:
>>> t2 = torch.tensor(...)  
```

- Remarks:
  - arrays / tensors must be on the same device.
  - only detached arrays can be converted to numpy (see later)
  - if data types are not the same, casting might be needed (v1.1 or older)
    - E.g. adding an integer and a float tensor together.
Autograd Example

```python
>>> import torch
>>> from torch import autograd
>>> x1 = torch.tensor(2, requires_grad=True, dtype=torch.float32)
>>> x2 = torch.tensor(3, requires_grad=True, dtype=torch.float32)
>>> x3 = torch.tensor(1, requires_grad=True, dtype=torch.float32)
>>> x4 = torch.tensor(4, requires_grad=True, dtype=torch.float32)
>>> # Forward propagation
>>> z1 = x1 * x2
>>> z2 = x3 * x4
>>> f = z1 + z2
>>> df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4])
>>> df_dx
(tensor(3.), tensor(2.), tensor(4.), tensor(1.))
```
Under the Hood (a little bit)

```python
>>> df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4])
>>> df_dx
(tensor(3.), tensor(2.), tensor(4.), tensor(1.))
```
Autograd (a little bit better)

```python
>>> import torch
>>> from torch import autograd
>>> x1 = torch.tensor(2, requires_grad=True, dtype=torch.float32)
>>> x2 = torch.tensor(3, requires_grad=True, dtype=torch.float32)
>>> x3 = torch.tensor(1, requires_grad=True, dtype=torch.float32)
>>> x4 = torch.tensor(4, requires_grad=True, dtype=torch.float32)
>>> # Forward propagation
>>> z1 = x1 * x2
>>> z2 = x3 * x4
>>> f = z1 + z2
>>> # df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4]) # inconvenient
>>> f.backward() # that is better!
>>> print(f"f's derivative w.r.t. x1 is \{x1.grad}"
) tensor(3.)
```
Autograd (a little bit better)

```python
>>> import torch
>>> from torch import autograd
>>> x1 = torch.tensor(2, requires_grad=True, dtype=torch.float32)
>>> x2 = torch.tensor(3, requires_grad=True, dtype=torch.float32)
>>> x3 = torch.tensor(1, requires_grad=True, dtype=torch.float32)
>>> x4 = torch.tensor(4, requires_grad=True, dtype=torch.float32)
>>> # Forward propagation
>>> z1 = x1 * x2
>>> z2 = x3 * x4
>>> f = z1 + z2
>>> # df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4]) # inconvenient
>>> f.backward()  # that is better!
>>> print(f"f's derivative w.r.t. x1 is {x1.grad}")
tensor(3.)
```
Under the Hood (a little bit)
Context managers, decorators

- We can locally disable/enable gradient calculation with
  - `torch.no_grad()`
  - `torch.enable_grad()`
- or using the `@torch.no_grad @torch.enable_grad` decorators

```python
>>> x = torch.tensor([1], requires_grad=True)
>>> with torch.no_grad():
...    y = x * 2
>>> y.requires_grad
False

>>> with torch.no_grad():
...    with torch.enable_grad():
...        y = x * 2
>>> y.requires_grad
True
```
Example: Linear Regression

- Generating data:

  ```python
  >>> a_ref = -1.5
  >>> b_ref = 8
  >>> noise = 0.2 * np.random.randn(50)
  >>> x = np.linspace(1, 4, 50)
  >>> y = a_ref * x + b_ref + noise
  ```

- Defining loss function:

  ```python
  >>> def MSE_loss(prediction, target):
  ...     return (prediction-target).pow(2).mean()
  ```
Example: Linear Regression

- Data as torch tensors and the unknown variables:

```python
xx = torch.tensor(x, dtype=torch.float32)
yy = torch.tensor(y, dtype=torch.float32)

a = torch.randn(0, requires_grad=True, dtype=torch.float32)
b = torch.randn(5, requires_grad=True, dtype=torch.float32)
```
Example: Linear Regression

- Training Loop:

```python
number_of_epochs = 1000
learning_rate = 0.01

for iteration in range(number_of_epochs):
    y_pred = a * xx + b
    loss = MSE_loss(y_pred, yy)
    loss.backward()

    with torch.no_grad():
        a = a - learning_rate * a.grad
        b = b - learning_rate * b.grad
    a.requires_grad = True
    b.requires_grad = True

print(a)
print(b)
```
Example: Linear Regression

- Result:
  
  ```
  tensor(-1.5061, requires_grad=True)
  tensor(8.0354, requires_grad=True)
  ```

![Graph showing a linear regression trend](image)
Other useful pyTorch’s tensor functions

- If you want to detach a tensor from the graph, you can use « detach() »
- If you want to get a python number from a tensor, you can use « item() »
- But if you just take an element, it still will be part of the computational graph!

```python
>>> x=torch.tensor([2.5,3.5], requires_grad=True)
tensor([2.5000, 3.5000], requires_grad=True)
>>> x.detach()
tensor([2.5000, 3.5000])
>>> x[0]    # still part of the graph!
tensor(2.5000, grad_fn=<SelectBackward>)
>>> x[0].item()
2.5

>>> # a frequent line when you go back to numpy:
>>> x.detach().cpu().numpy()
array([[2.5, 3.5], dtype=float32])
```
The «torch.nn.functional» package is the functional interface for Pytorch features. Most features exist both as a function and as a class. Structural parts, or objects with internal state, are usually used as objects. Stateless or simple expressions are usually used in functional form. Activation functions, losses, convolutions, etc. are a huge module.

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

x = torch.rand(2,2)
y = F.relu(x)
relu = nn.ReLU()  # creating the object first
z = relu(x)  # then using it
y == z  # they should be the same

# Similarly:
mseloss = nn.MSELoss()
F.mse_loss(...) == mse_loss(...)```
A typical (pyTorch) ML Workflow

1. creating dataset
2. creating a neural network (model)
3. defining a loss function
4. loading samples (data loader)
5. predicting with the model
6. comparison of the prediction and the target (loss)
7. backpropagation: calculating gradients from the error
8. updating the model (optimizer)
9. checking the loss: if low enough, stop training
Data Loading
Data loading and preprocessing

- The `torch.utils.data` package has two useful classes for loading and preprocessing data:
  - `torch.utils.data.Dataset`
  - `torch.utils.data.DataLoader`

- For more information visit:
  - [https://pytorch.org/tutorials/beginner/data_loading_tutorial.html](https://pytorch.org/tutorials/beginner/data_loading_tutorial.html)
### torch.utils.data.Dataset: Regression Ex. revisited

```python
import torch
class LinearRegressionDataset(torch.utils.data.Dataset):

    def __init__(self, N = 50, m = -3, b = 2, *args, **kwargs):
        # N: number of samples, e.g. 50
        # m: slope
        # b: offset
        super().__init__(*args, **kwargs)

        self.x = torch.rand(N)
        self.noise = torch.rand(N)*0.2
        self.m = m
        self.b = b

    def __getitem__(self, idx):
        y = self.x[idx] * self.m + self.b + self.noise[idx]
        return {'input': self.x[idx], 'target': y}

    def __len__(self):
        return len(self.x)
```

---

**Note:**
- The code snippet above demonstrates how to create a custom dataset for a linear regression problem using PyTorch's `Dataset` class. The dataset generates random samples with a specified number of samples, slope, and offset, and adds noise to simulate real-world data.
- The `__init__` method initializes the dataset with the specified parameters and generates the data.
- The `__getitem__` method returns a sample as a dictionary with the 'input' and 'target' fields.
- The `__len__` method returns the total number of samples in the dataset.
import torch
import imageio

class ImageDataset(torch.utils.data.Dataset):
    def __init__(self, root, N, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.input, self.target = [], []
        for i in range(N):
            t = imageio.imread(f'{root}/train_{i}.png')
            t = torch.from_numpy(t).permute(2, 0, 1)
            l = imageio.imread(f'target_{i}.png')
            l = torch.from_numpy(l).permute(2, 0, 1)
            self.input.append(t)
            self.target.append(l)

    def __getitem__(self, idx):
        return {'input': self.input[idx], 'target': self.target[idx]}

    def __len__(self):
        return len(self.input)
torch.utils.data.Dataset: Image Datasets

```python
import torch
import ImageDataset

datapath = 'data_directory'
myImageDataset = ImageDataset(dataPath, 50)
# iterating through the samples
for sample in myImageDataset:
    input = sample['input'].cpu()  # or .cuda()
    target = sample['target'].cpu()  # or .to(device)
    ....
```

Never ever use .cuda() in the dataset or data loaders!
torch.utils.data.DataLoader

```python
import torch
import ImageDataset
datapath = 'data_directory'
myImageDataset = ImageDataset(dataPath, 50)
# iterating through the samples
train_loader = DataLoader(dataset=myImageDataset, batch_size=32,
                           shuffle=False, num_workers=2)
for sample in train_loader:
    ...

• «DataLoader» is used to:
  ◦ Batching the dataset
  ◦ Shuffling the dataset
  ◦ Utilizing multiple CPU cores/ threads
```
Data augmentation

- modifying the dataset for better training (more robust, etc.)
- data set can have a transform parameter

More details can be found here:
https://pytorch.org/tutorials/beginner/data_loading_tutorial.html
import torch
import imageio

class ImageDataset(torch.utils.data.Dataset):
    def __init__(self, root, N, transform = None, *args,**kwargs):
        super().__init__(*args,**kwargs)
        self.transform = transform
...

    def __getitem__(self, idx):
        sample = {'input': self.input[idx], 'target': self.target[idx]}
        if self.transform:
            sample = self.transform(sample)
        return sample

    def __len__(self):
        return len(self.input)
Data transformation

```python
import torchvision.transforms as T
composed = transforms.Compose([T.Rescale(256),
                                 T.RandomCrop(224),
                                 T.ToTensor()])

... dataset = Mydataset(..., transform = composed)

# another version, needs different dataset
dataset = Mydataset(..., transform = {'input' : composed,
                                       'target' : None})
```
Creating the Model
nn.Module

- A model is of a `nn.Module` class type. A model can contain other models. E.g. we can create the class “Model” based on the stacking `nn.Module`'s of type `nn.Linear`
- The `nn.Module`'s weights as called “Parameters”, and are similar to tensors with “requires_grad=True”.
- A `nn.Module` consists of an initialization of the Parameters and a `forward` function.

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        # structure definition and initialization

    def forward(self, x):
        # actual forward propagation
        result = processing(x)
        return result
```
Model

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        # let's assume 28x28 input images, e.g. MNIST characters
        self.fc1 = nn.Linear(in_features = 28 * 28, out_features = 128, bias=True)
        self.fc2 = nn.Linear(in_features = 128, out_features = 64, bias=True)
        self.fc3 = nn.Linear(in_features = 64, out_features = 10, bias=True)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
class Model2(nn.Module):
    def __init__(self):
        super().__init__()
        # let's assume 28x28 input images, e.g. MNIST characters
        self.fc1 = nn.Linear(in_features = 28 * 28, out_features = 128, bias=True)
        self.activation1 = nn.ReLU()
        self.fc2 = nn.Linear(in_features = 128, out_features = 64, bias=True)
        self.activation2 = nn.ReLU()
        self.fc3 = nn.Linear(in_features = 64, out_features = 10, bias=True)
        self.activation3 = nn.ReLU()

    def forward(self, x):
        x = self.activation1(self.fc1(x))
        x = self.activation2(self.fc2(x))
        x = self.activation3(self.fc3(x))
        return x

What is the difference?
nn.Module’s member functions

- Access model information

```python
>>> model = Model()
>>> model.eval() # see below
>>> list(model.children())
[Linear(in_features=784, out_features=128, bias=True),
 Linear(in_features=128, out_features=64, bias=True),
 Linear(in_features=64, out_features=10, bias=True)]
```

- Children: the parameters and modules / layers defined in the constructor.
- Parts defined in the forward method will not be listed.
- Forward is called many times, expensive objects should not be recreated.

Some layers as e.g. "dropout" and "batch_norm" should operate differently during training and evaluation of the model. We can set the model in different state by the .train() and .eval() functions.
Model’s Parameters

```python
>>> for key, value in model.state_dict().items():
...   print(f'layer = {key:<10s} | feature shape = {value.shape}')

layer = fc1.weight | feature shape = torch.Size([128, 784])
layer = fc1.bias   | feature shape = torch.Size([128])
layer = fc2.weight | feature shape = torch.Size([64, 128])
layer = fc2.bias   | feature shape = torch.Size([64])
layer = fc3.weight | feature shape = torch.Size([10, 64])
layer = fc3.bias   | feature shape = torch.Size([10])
```

- The `.state_dict()` contains all the trainable parameters of the model, this is used for optimization and saving/restoring the model.
So far...

device = torch.device('cpu')
dataset = CustomDataset()
dataloader = DataLoader(dataset, ...)
model = MyModel()
model.to(device)
for i in range(epochs):
    training_loss = 0
    for sample in dataloader:
        input = sample['input'].to(device)
target = sample['target'].to(device)
prediction = model(input)
        loss = loss_function(prediction, target)
        training_loss += loss.item()
        loss.backward()
        # updating the model
    print(f'Current training loss: {training_loss}')
    # validation loop
...
# saving the model
Optimizers
Choosing an Optimizer

Using PyTorch’s optimizers is easy!

```python
import torch
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
...
for sample in dataloader:
    input = sample['input'].to(device)
    target = sample['target'].to(device)
    prediction = model(input)
    loss = loss_fn(prediction, target)

    optimizer.zero_grad()  # clears the gradients
    loss.backward()
    optimizer.step()  # performs the optimization
```
Accumulating Gradients (a Trick)

- If we don’t clear the gradients, they sum up.
- This is often source of bugs, but it can be exploited for larger effective batch sizes:

```python
import torch
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

optimizer.zero_grad()
for idx, sample in enumerate(dataloader):
    input = sample['input'].to(device)
    target = sample['target'].to(device)

    prediction = model(input)
    loss = loss_fn(prediction, target)

    loss.backward()
    if idx % 10 == 9:
        optimizer.step()
        optimizer.zero_grad()
```
Save/Load Models
Saving the internal state of a pyTorch model

- Saving and loading can easily be done using “torch.save” and “torch.load”
- pyTorch uses “pickling” to serialize the data.

```python
>>> state = {'model_state' : model.state_dict(),
            'optimizer': optimizer.state_dict()}

>>> torch.save(state, 'state.pt')
```

Restoring state:

```python
>>> model = Model()
>>> optimizer = optim.SGD(model_parameters(), lr=0.01)
>>> checkpoint = torch.load('state.pt')
>>> model.load_state_dict(checkpoint['model_state'])
>>> optimizer.load_state_dict(checkpoint['optimizer_state'])
```
A Typical ML Pipeline
import json
config = json.load(open('config.cfg'))
device = torch.device(config['device'])
training_data = CustomDataset(..., **config['train'])
validation_data = CustomDataset(..., **config['valid'])
train_loader = DataLoader(training_data, **config['loader'])
validation_loader = DataLoader(validation_data, **config['loader'])
model = MyModel(**config['model'])
model.to(device)
optimizer = Optimizer(model.parameters(), **config['optimizer'])
for i in range(config['epochs']):
    model.train()
    for sample in train_loader:
        optimizer.zero_grad()
        input, target = sample['input'].to(device), sample['target'].to(device)
        prediction = model(input)
        loss = loss_function(prediction, target)
        print(f'Current training loss: {loss.item()}')
        loss.backward()
        optimizer.step()
All the pieces together, part 2

```python
# validation loop
model.eval()
validation_loss = 0
for sample in validation_loader:
    input, target = sample['input'].to(device), sample['target'].to(device)
    prediction = model(input)
    loss = loss_function(prediction, target)
    validation_loss += loss.item()
print(f'Current validation loss: {validation_loss}')
if validation_loss < config['loss_threshold']: # or other condition
    break
full_state = {'model_state': model.state_dict(), 'optimizer': optimizer.state_dict()}
torch.save(full_state, 'parameters.pt')
```
Reproducibility

- Sometimes it is hard to reproduce bugs because of the randomness in the training. The solution is using fixed random seeds.
- For debugging purposes, you should start your codes with these lines:

  ```python
  import numpy as np
  np.random.seed(42)  # your favourite integer

  import torch
  torch.manual_seed(42)  # your favourite integer
  torch.backends.cudnn.deterministic = True  # disable optimizations
  torch.backends.cudnn.benchmark = False
  ```

  But remove them when you are done with debugging, otherwise all the models will be the same!
Colab Example

https://colab.research.google.com/drive/1HUWge-hdUQZMdIqX0fys77c9pUycxSPj#scrollTo=Y9gcgKUmyBXD
Deep Learning
Automatic Differentiation and Pytorch