UNIVERSITAT POLITÈCNICA DE VALÈNCIA

MIARFID Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Session 2. Introduction to PyTorch

Aplicaciones de Reconocimiento de Formas (ARF)

Curso 2023/2024

Departamento de Sistemas Informáticos y Computación

Index

Index

Overview of PyTorch

PyTorch

End-to-end Machine Learning Framework

Provided high level features:

- Tensor computation with GPU extension
- Deep Neural Networks built on tape-based autograd system

PyTorch enables fast, flexible experimentation and efficient production through a user-friendly front-end, distributed training, and ecosystem of tools and libraries.

Overview of PyTorch

Most important PyTorch library components:

- `torch`: `Tensor` library with GPU support
- `torch.autograd`: automatic differentiation library for any `torch Tensor`
- `torch.nn`: neural network library integrated with `autograd`
- `torch.cuda`: support for CUDA GPU tensors
- `torch.optim`: optimization methods (SGD, RMSProp, LBFGS, Adam, . . .)
- `torch.linalg`: lineal algebra operations
- `torch.distributions`: probability distributions
- `torch.utils`: `DataLoader`, `tensorboard`, model zoo, and other
- `torch.library`: user defined operations
- `torch.distributed`: for distributed systems computation
- `torch.onnx`: export to ONNX format
- `torch.hub`: pre-trained models repository

Overview of PyTorch

PyTorch features: current stable version (2.2.0)

- Production ready
- Torchserve: tool for deploying PyTorch models at scale
- Distributed training
- Robust ecosystem
- Native support of ONNX
- C++ front-end
- Cloud support
- Mobile support (experimental): for iOS and Android

Full documentation

Index

PyTorch basics

Based on Soumith Chintala's Deep Learning with PyTorch: A 60 Minute Blitz

Tensor:

- `Tensor` objects: n-dimensional arrays (similar to NumPy's `ndarray`)
- Usable with GPUs

```
(base) arf@arf-carmarhi-student-2024:/tmp$ python
Python 3.11.5 (main, Sep 11 2023, 13:54:46) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> x = torch.empty(5,3)
>>> print(x)
tensor([[5.4457e-38, 4.5556e-41, 5.5168e-38],
        [4.5556e-41, 5.4457e-38, 4.5556e-41],
        [5.4457e-38, 4.5556e-41, 5.5190e-38],
        [4.5556e-41, 5.5191e-38, 4.5556e-41],
        [6.2173e-38, 4.5556e-41, 5.5169e-38]])
```

PyTorch basics

Tensor creation from data:

```
>>> d = [[1, 2],[3, 4]]
>>> x_d = torch.tensor(d)
>>> print(x_d)
tensor([[1, 2],
        [3, 4]])
```

Tensor creation from NumPy array:

```
>>> import numpy as np
>>> np_a = np.array(d)
>>> x_np = torch.from_numpy(np_a)
>>> print(x_np)
tensor([[1, 2],
        [3, 4]])
```

Tensor creation from other Tensor (keeping dimensions):

```
>>> x_ones = torch.ones_like(x_d)
>>> print(x_ones)
tensor([[1, 1],
        [1, 1]])
>>> x_rand = torch.rand_like(x_d, dtype=torch.float)
>>> print(x_rand)
tensor([[0.5075, 0.1500],
        [0.0492, 0.7871]])
```

PyTorch basics

Tensor random initialisation:

```
>>> x = torch.rand(5, 3)
>>> print(x)
tensor([[0.6546, 0.9033, 0.2316],
        [0.4486, 0.2093, 0.9956],
        [0.4243, 0.1832, 0.9598],
        [0.4879, 0.5403, 0.5412],
        [0.4111, 0.9258, 0.7138]])
```

`torch.ones(...)` and `torch.zeros(...)` for unitary or null `Tensor`

Tensor attributes:

```
>>> print(x.shape)
torch.Size([5, 3])
>>> print(x.dtype)
torch.float32
>>> print(x.device)
cpu
```

Parallelisation for GPU with CUDA: transform `Tensor` into CUDA `Tensor`

```
>>> if torch.cuda.is_available():
>>>     x = x.to("cuda")
>>>     y = y.to("cuda")
```

PyTorch basics

Tensor indexing/slicing:

```
>>> t = torch.ones(4, 4)
>>> t[:,1] = 0
>>> print(t)
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Tensor joining:

```
>>> t = torch.ones(2,2)
>>> t1 = torch.cat([t, t], dim=1)
>>> print(t1)
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.]])
>>> t0 = torch.cat([t, t], dim=0)
>>> print(t0)
tensor([[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]])
```

PyTorch basics

Tensor multiplication:

```
>>> t1 = torch.ones(4, 4)
>>> t1[:, 1] = 0
>>> print(t1)
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

```
>>> t2 = torch.ones(4, 4)
>>> t2[1, :] = 0
>>> print(t2)
tensor([[1., 1., 1., 1.],
        [0., 0., 0., 0.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

- Element-wise: `mul`, `*`

```
>>> print(t1.mul(t2))
tensor([[1., 0., 1., 1.],
        [0., 0., 0., 0.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

```
>>> print(t1*t2)
tensor([[1., 0., 1., 1.],
        [0., 0., 0., 0.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

- Matrix product: `matmul`, `@`

```
>>> print(t1.matmul(t2))
tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])
```

```
>>> print(t1@t2)
tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])
```

PyTorch basics

Tensor math operations:

- In-line operands: `x + y`
- Tensor methods: `torch.add(x, y)`, `torch.add(x, y, out=result)`
- In-place operands (with _): `y.add_(x)` (same than `y = y + x`)

Some important methods for Tensor math operations:

abs	add	atan	ceil	clamp	cos	cosh	div
exp	floor	lerp	log	mul	neg	pow	remainder
round	sigmoid	sign	sin	sinh	sqrt	tan	tanh
dist	mean	median	norm	prod	std	sum	var
eq	equal	ge	gt	isfinite	isinf	isnan	isreal
	le	lt	maximum	minimum	ne	sort	
	diag	histc	renorm	trace	tril	triu	
det	dot	inverse	mm	mv	qr	svd	symeig

Complete list

Index

Autograd

autograd: package that provides automatic differentiation for Tensor

Full autograd documentation

When creating a Tensor:

- With requires_grad=True makes operations on Tensor to be tracked
- Call to backward() allows to compute gradients
- grad attribute stores gradients

```
>>> import torch
>>> a = torch.tensor([2., 3.], requires_grad=True)
>>> print(a)
tensor([2., 3.], requires_grad=True)
```

Autograd

Each `Tensor` with `requires_grad=True` is connected to the function that created it

`Tensor` attribute `grad_fn` (`None` for user-defined `Tensor`)

From the basic `Tensor` (the one with `requires_grad=True`), the corresponding functions that create the final composed object can be defined:

```
>>> b = torch.tensor([6., 4.], requires_grad=True)
>>> Q = 3*a**3 - b**2
>>> print(Q)
tensor([-12.,  65.], grad_fn=<SubBackward0>)

>>> print(Q.grad_fn)
<SubBackward0 object at 0x7efe01954dc0>
```

Now `Q` has the function $Q = 3a^3 - b^2$

Autograd

Gradients: by using the `backward` method

Derivatives with respect to `a` and `b`:

$$\frac{\mathrm{d}Q}{\mathrm{d}a} = 9a^2 \qquad \frac{\mathrm{d}Q}{\mathrm{d}b} = -2b$$

Specific values:

$$a = (2,3) \rightarrow \frac{\mathrm{d}Q}{\mathrm{d}a} = (36,81) \qquad b = (6,4) \rightarrow \frac{\mathrm{d}Q}{\mathrm{d}b} = (-12,-8)$$

It is necessary to indicate the gradient to do the derivatives:

```
>>> Q.backward(gradient=torch.tensor([1., 1.]))
>>> a.grad
tensor([36., 81.])
>>> b.grad
tensor([-12.,  -8.])
```

Index

Neural networks

By using the `torch.nn` package:

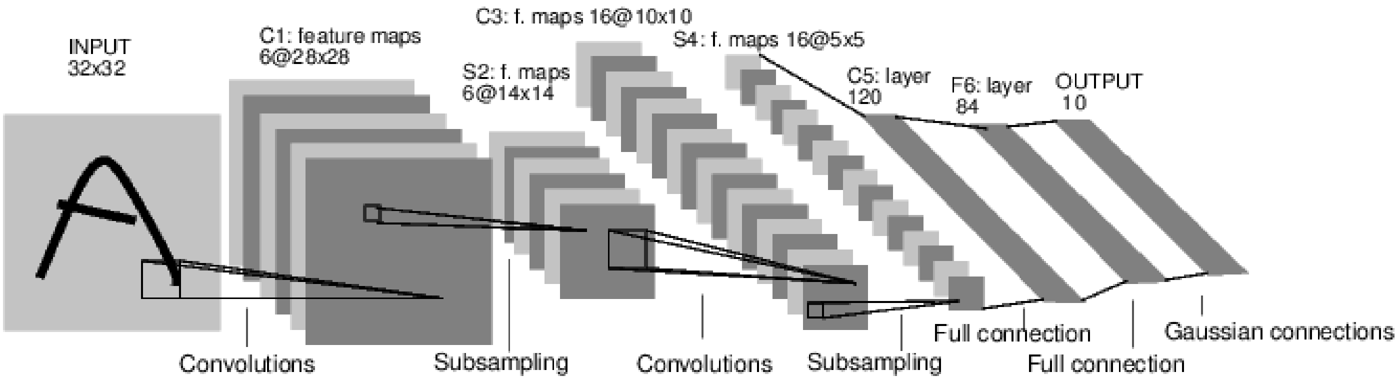
- `nn` employs `autograd` for model definition and differentiation
- `nn.Module` contains
  - The layers
  - A `forward(input)` method that returns an `output`

Network training process:

- Define network with its weights
- Iterate over input dataset and process it by the network
- Compute network loss
- Propagate gradients back into network
- Update network weights

Neural networks

Example of feed-forward network (LeNet)



Definition of the network includes:

- Convolutional layers
- Subsampling (max pool)
- Linear fully-connected layers

Neural networks

Structure for the network code:

- Import libraries and classes (`torch`, `nn`, `functional`)

- Define class for the network, derived from `nn.Module`
  - Constructor (`__init__`): define network elements
  - `forward` method: define network connections and operations
  - Auxiliar methods (if needed)

- `backward` method not necessary (provided by `autograd`)

Neural networks

Code:

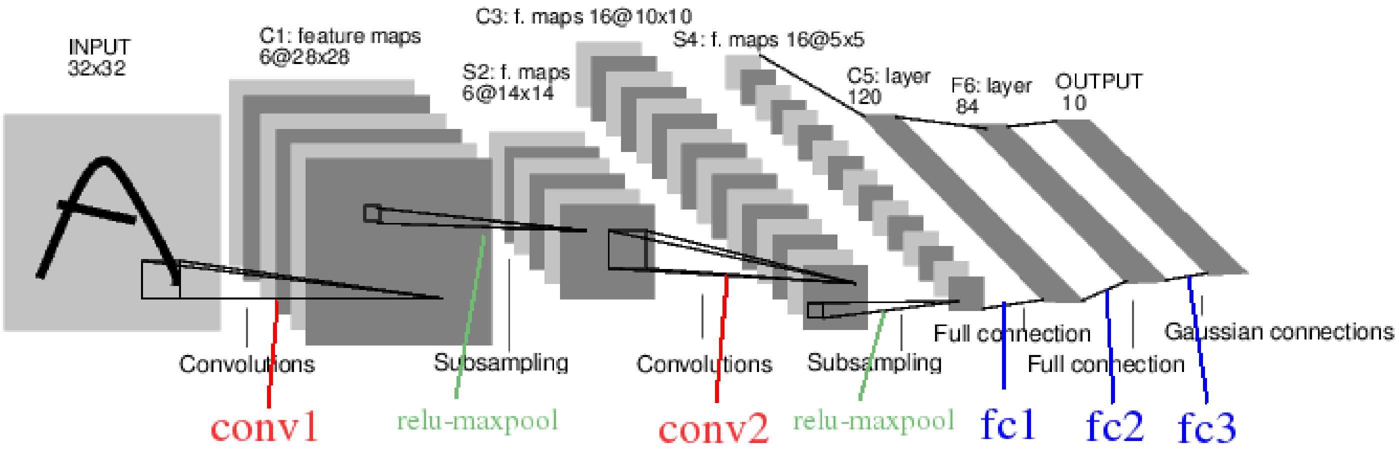
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Neural networks



For creating and visualising the net:

```
>>> net = Net()
>>> print(net)
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Neural networks

Visualising the internal parameters of the network:

```
>>> params = list(net.parameters())
>>> print(len(params))
10

>>> print(params[0].size())
torch.Size([6, 1, 5, 5])

>>> print(params[0])
Parameter containing:
tensor([[[[ 0.0727,  0.1421, -0.0468, -0.0630, -0.1446],
          [ 0.0773, -0.0762, -0.1289, -0.1441,  0.1419],
          [-0.0679, -0.1911, -0.1467,  0.1722,  0.0006],
          [ 0.1494, -0.1786, -0.1229, -0.1727,  0.1350],
          [-0.0813,  0.1056, -0.1841,  0.1526, -0.1622]]],

          ...

        [[[ 0.0104, -0.0567,  0.0613, -0.0571, -0.1645],
          [ 0.1841,  0.1473,  0.0132, -0.0322,  0.0823],
          [-0.0269,  0.1671, -0.0737,  0.0866,  0.1922],
          [ 0.0688,  0.0618,  0.0238, -0.0139, -0.0216],
          [ 0.1568, -0.0625, -0.1545, -0.0483, -0.1220]]]], requires_grad=True)
```

Neural networks

Types of layers provided by `torch.nn`:

- Convolutional: `Conv1d`, `Conv2d`, `Conv3d`, ...
- Pooling: `MaxPool1d`, `MaxUnpool1d`, `AvgPool1d`, ...
- Padding: `ReflectionPad2d`, `ReplicationPad2d`, `ZeroPad2d`, ...
- Non-linear activators: `ReLU`, `SELU`, `Threshold`, `Sigmoid`, `Softmax`, ...
- Normalisation: `BatchNorm1d`, `InstanceNorm1d`, `LayerNorm`, ...
- Recurrent: `RNN`, `LSTM`, `GRU`, ...
- Transformer: `Transformer`, `TransformerEncoder`, ...
- Linear: `Linear`, `Bilinear`, ...
- Dropout: `Dropout`, `Dropout2d`, `Dropout3d`, ...
- Sparse: `Embedding`, `EmbeddingBag`
- Other types: vision, shuffle, parallel

Full documentation

Neural networks

The network uses `Tensor` objects as input and output data

```
>>> input = torch.randn(1, 1, 32, 32)
>>> out = net(input)
>>> print(out)
tensor([[-0.0732, -0.0575, -0.1335, -0.0140, -0.0610,  0.0496, -0.0550,  0.0494,
          0.0619,  0.0087]], grad_fn=<AddmmBackward0>)
```

For gradient propagation, reset (`zero_grad`) and backpropagate (`backward`)

```
>>> net.zero_grad()
>>> out.backward(torch.randn(1, 10))
```

Neural networks

Loss functions:

- Provided by `torch.nn`
- Many different functions available
  - `MSELoss`: mean square error
  - `CrossEntropyLoss`
  - `CTCLoss`: Connectionist Temporal Classification
  - `NLLLoss`: negative log likelihood
  - `BCELoss`: Binary Cross Entropy
  - `KLDivLoss`: Kullback-Leibler
  - ...

Full documentation

Neural networks

Example with `MSELoss`:

```
>>> output = net(input)
>>> target = torch.randn(10)
>>> target = target.view(1, -1)
>>> criterion = nn.MSELoss()
>>> loss = criterion(output, target)
>>> print(loss)

tensor(0.8269, grad_fn=<MseLossBackward0>)
```

Neural networks

Computation graph for `loss`:

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> flatten -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

Some steps backward:

```
>>> print(loss.grad_fn)
<MseLossBackward0 object at 0x7f62ac864b20>
>>> print(loss.grad_fn.next_functions[0][0])
<AddmmBackward0 object at 0x7f62ac864b50>
>>> print(loss.grad_fn.next_functions[0][0].next_functions[0][0])
<AccumulateGrad object at 0x7f62ac864b50>
```

Neural networks

Now the loss must be backpropagated to re-estimate the network weights

```
net.zero_grad()
print(net.conv1.bias.grad)
None

loss.backward()
print(net.conv1.bias.grad)
tensor([0.0150, 0.0043, 0.0250, 0.0049, 0.0022, 0.0150])
```

Neural networks

Weight update: via optimisers in `torch.optim`

- `Adam`
- `LGBFS`
- `RMSprop`
- `SGD`
- `...`

Full documentation

Neural networks

Example of use: with `SGD` and learning rate of 0.01

```
>>> net.zero_grad()
>>> import torch.optim as optim
>>> optimizer = optim.SGD(net.parameters(), lr=0.01)
>>> optimizer.zero_grad()
>>> output = net(input)
>>> loss = criterion(output, target)
>>> loss.backward()
>>> optimizer.step()
```

Neural networks

Network training: for each epoch and each training sample

- Obtain `inputs` and `labels` as `Tensor` objects
- Make the optimizer reset (`optimizer.zero_grad()`)
- Pass input to network and obtain output (`outputs = net(inputs)`)
- Compute loss (`loss = criterion(outputs, labels)`)
- Backpropagate (`loss.backward()`)
- Change network weights (`optimizer.step()`)

Network testing: for each test sample

- Obtain `inputs` and `labels`, `inputs` as a `Tensor` object
- Obtain network output (`outputs = net(inputs)`)
- Get hypothesis from output (`_, hyp = torch.max(outputs.data, 1)`)
- Compare real and hypothesis label to check error

Neural networks

Summary:

1. Define and create network
2. Define loss function
3. Define optimiser
4. Load training data
5. Train network
6. Load test data
7. Test network

Index

Practical task: speech recognition

Based on José A. Rodríguez Fonollosa Google commands code

Task description: Spanish digit recognition

- 10 speakers, 10 repetitions each speaker
- A total of 1000 samples (100 for each class)
- Format: `NSR.wav` (`N` number, `S` speaker, `R` repetition)
- Partitions:
  - Training: speakers 0, 1, 2 (female), 5, 6, 7 (male)
  - Validation: speakers 3 (female) and 8 (male)
  - Test: speakers 4 (female) and 9 (male)

Practical task: speech recognition

Download database from PoliformaT (`DIG.tgz`)

Download Python code from Poliformat:

- `DIG_loader.py`: to read wav files and organise data
- `model.py`: models definitions
- `train.py`: train and test methods
- `run.py`: full experiment code

Put all downloaded files into the same folder (e.g., `02-PYTORCH`)

Uncompress `DIG.tgz`

Practical task: speech recognition

Run full experiment:

```
python run.py --arc LeNet --num_workers 1
```

Approximated running time: 2 minutes

Final test result: 91.5% accuracy

Check the code and play with the different options

Be careful!: logs for models can make your disk space full (`checkpoint` directory) and some models are so large that do not fit in the virtual machine memory