Introduction to Neural Network programming with Python and PyTorch

Lecturer: Valerio De Caro
valerio.decaro@phd.unipi.it

Intelligent Systems for Pattern Recognition
Apr 9th 2024

PAI LAB
Università di Pisa
• **Tensor manipulation**: library to manipulate tensors, with MATLAB/Numpy-like API.

• **GPU support**: seamless execution on GPU and CPU devices.

• **Automatic Differentiation**: custom layers only need to define the forward step, because functions is automatically differentiated using the chain rule.

• **High-level API**: ready-to-use high-level API with neural networks layers, losses, and optimizers
• Python 3.x (or C++, good luck with that)

• Cross-platform

• Installation via pip or conda

• Installation available with both CPU-only and CUDA support (for GPU)

• To use GPU, you must check that the PyTorch version you’re installing matches the CUDA version on your machine. Examples:
  • PyTorch 2.2.2 requires CUDA 12.1 or CUDA 11.8;
  • PyTorch 1.13.1 requires CUDA 11.6 or CUDA 11.7);
  • Check it by using the bash command `nvidia-smi`

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

• PyTorch previous versions: https://pytorch.org/get-started/previous-versions/
Tensors operations and manipulation

Some basics
• Tensors are the main data structure. They represent multidimensional arrays

* Equivalent of np.ndarray

* Support **advanced indexing** and **broadcasting**

**Attributes:**

* **dtype:** determine the type of the tensor elements (float{16, 32, 64}, int{8, 16, 32, 64}, uint8). Can be specified during the initialization.

* **device:** memory location (cpu or cuda)

* **layout:** dense tensors (strided) or sparse (sparse_coo)

---

**The basic building block of any ML framework**

---

Tensors
- **torch.tensor**
  - takes any array-like argument and create a new tensor

- **Zero or one initialization**
  - `torch.zeros(*dims)` – `torch.ones(*dims)`

- **Random**
  - `torch.randn(*dims)`
  - `torch.rand(*dims)`

- **Linear range**
  - `torch.linspace(start, end, steps=100)`

- **numpy bridge**
  - `torch.from_numpy(x)`
  - You can also convert a tensor into a ndarray with the `.numpy` method
  - **Note**: the numpy array and the resulting tensor share the memory

---

**Tensors**
• **torch.cuda** API for GPU management (check availability with `torch.cuda.is_available`)

**Using GPU**

1. Create or move to GPU: `torch.tensor(..., device=”cuda”)` or `tensor.to(”cuda”)`
2. All the tensor arguments of an operation **must reside on the same device** → **result on the same device**

• Can take the GPU id as an optional argument if you have multiple GPUs (e.g., `tensor.to(“cuda:3”)`)
• You can move tensors back to the CPU with the `cpu` method

---

**Using PyTorch for what it’s worth**

Tensors in GPU
• On a server you typically have access to multiple shared GPU and you must select one to run your code.
  • Manual selection using the device argument (‘cuda:0’, ‘cuda:1’…)
  • Using the context manager torch.cuda.device
  • Changing the shell environment variable CUDA_VISIBLE_DEVICES to limit the visible GPUs
    • `export CUDA_VISIBLE_DEVICES=0`
    • Note that the indices of torch.device will always start from 0.
    • E.g., `CUDA_VISIBLE_DEVICES=3,4` will give you two gpus s.t. `torch.device(cuda:0)` will use gpu 3 and `torch.device`

• **Always, always, ALWAYS REMEMBER TO DE-ALLOCATE STUFF FROM THE GPU IF YOU’RE NOT USING IT**

  ... or a group of angry phd students will come get you from home 😊

---

**GPU Usage**

10/04/2024

INTRODUCTION TO PyTorch
cudax = torch.device(device('cuda')) # Default CUDA device
cuda0 = torch.device(device('cuda:0'))
cuda2 = torch.device(device('cuda:2')) # GPU 2

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

With torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.tensor([1., 2.], device=cuda)
    # transfers a tensor from CPU to GPU 1
    b = torch.tensor([1., 2.]).cuda()
    # a.device and b.device are device(type='cuda', index=1)
    # You can also use ``Tensor.to`` to transfer a tensor:
    b2 = torch.tensor([1., 2.]).to(device=cuda)
    # b.device and b2.device are device(type='cuda', index=1)

c = a + b  # c.device is device(type='cuda', index=1)
z = x + y  # z.device is device(type='cuda', index=0)

# even within a context, you can specify the device
# (or give a GPU index to the .cuda call)
d = torch.randn(2, device=cuda2)
e = torch.randn(2).to(cuda2)
f = torch.randn(2).cuda(cuda2)
# d.device, e.device, and f.device are all device(type='cuda', index=2)
Tensor operations

- Some operators are overloaded
  - +, - for addition and subtraction (support broadcasting)
  - * is the elementwise multiplication (not the matrix product, supports broadcasting)
  - @ for matrix multiplication (torch.matmul)

- In-place operations are defined with a suffix underscore
  - add_, sub_, matmul_ are the in-place equivalent for the previous operators

- Check the documentation: [http://pytorch.org/docs/stable/torch.html#tensors](http://pytorch.org/docs/stable/torch.html#tensors)
• **Basic** tensor indexing is similar to list indexing, *but with multiple dimensions*

• **Boolean condition**: boolean arrays can be used to filter elements that satisfy some condition

• If the indices are less than the number of dimensions the missing indices are considered complete slices

```
# first k elements
x = a[:k]
# all but the first k
x = a[k:]
# negative indexing
x = a[-k:]
# mixed indexing
a[:t_max, b:b+k, :]
# indexing with Boolean condition
def relu(x):
    x[x < 0] = 0
    return x
```
Reshaping is fundamental in many occasions to achieve results efficiently

- We distinguish:
  - `tensor.squeeze()` → removes all singleton dimensions
  - `tensor.unsqueeze(dim)` → add a singleton dimension at the dim-th position
  - `tensor.transpose(dim1, dim2)` → transposes the two dimensions of the tensor
  - `tensor.permute(*dims)` → re-arranges the dimensions as in *dims

```python
x = torch.randn(5, 1, 5)
# squeeze
x.squeeze() → [5, 5]
# unsqueeze
x.unsqueeze(3) → [5, 1, 5, 1]
# transpose
x.transpose(1, 2) → [5, 5, 1]
# indexing with Boolean condition
x.permute(1, 0, 2) → [1, 5, 5]

"→ " indicates calling `x.size()`
```
• Torch allows to collapse dimensions of tensors via reduce operations

• We distinguish:
  • tensor.sum/prod(dim) → collapses the dim-th dimension by element-wise summing or multiplying tensors
  • tensor.amin/amax(dim) → collapses the dim-th dimension by getting the element-wise min or max

```
x = torch.randn(5,1,5)
# You can do
x.sum(0) → [1,5]
# or
x.amin(2) → [5,1]
```

”→ “ indicates calling x.size()
In [3]: a = torch.rand(3, 3)
In [4]: b = torch.rand(3, 1)
In [5]: s1 = a + b
In [6]: c = torch.rand(3, 1, 1)
In [7]: s2 = a + c
In [8]: d = torch.rand(3, 2)
In [9]: a + d

RuntimeError: inconsistent tensor size, expected r_ [3 x 3], t [3 x 3] and src [3 x 2] to have the same number of elements, but got 9, 9 and 6 elements respectively at d:\projects\pytorch\torch\lib\th\generic\THTensorMath.c:887

ok, b is expanded
this is equivalent to a + b.expand(-1, 3)

ok, a and c are expanded
a.unsqueeze(2).expand(3,3,3) + c.expand(3,3,3)

error, a and d are not broadcastable

INTRODUCTION TO PyTorch
Autograd

Automatic Differentiation in PyTorch
• **torch.autograd** is the package responsible for the automatic differentiation

• Each computation creates a dynamic computational graph. Each operation adds a **Function** node, connected to its **Tensor** arguments

• The graph is used to compute the gradient by calling the method **backward**.
• Tensor objects are the data nodes of the computational graph

• The main attributes related to the graph structure are:
  • **data**: Tensor containing the Variable value
  • **grad**: Tensor containing the gradient (initially set to None)
  • **grad_fn**: the function used to compute the gradient

• Each Function implements two methods:
  • **forward**: function application
  • **backward**: gradient computation
• The **requires_grad** attribute is used to specify if the gradient computation should propagate into the Tensor or stop
  • for model’s parameters requires_grad=True
  • for input data or constant values requires_grad=False

• You can truncate the gradient using **detach**. The method removes the Tensor from the graph, making it a leaf.

• **In-place** modification is not allowed because it breaks the automatic differentiation.

• At inference time you can speed up the computation by using the context manager **torch.no_grad**, which disables the graph construction required for the backward computation, saving space and time.

Back-propagation uses the dynamically built graph

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
next_h.backward(torch.ones(1, 20))
```

Building the Dynamic Graph

Autograd
Building models and pipelines

TORCH.nn, LOSSES, OPTIMIZERS AND DATASETS
torch.nn contains the basic components to define your neural networks, loss functions, regularization techniques and optimizers.

We will see in the next few slides
- What is a nn.Module
- how to define a custom nn.Module
- how to set up a basic training loop

**Basic Interface for Building Models**

**torch.nn**
• **Module** is the base class for all the neural network submodules
  • Linear, convolutional, recurrent layers are all Module subclasses

• A `nn.Module` contains **Parameters**:  
  • These are typically the trainable parameters of your model  
  • **Parameter is a wrapper of a tensor with a name and requires_grad=True**  
  • You can iterate over all the parameters using the `parameters()` method

• You can compute the output of a network by using it like a function (e.g. `y_pred = net(X)`)  
  • That is possible because `__call__` is overridden  
  • **The computation is performed by the forward method**, but if you forward directly the module’s hooks are not activated

• It is possible to define forward and backward hooks  
  • e.g. you can check for NaN gradients after the backward pass  
  • You can register the hook with methods like `register_forward_hook()`

---

**Defining a model (or part of it) in a single class**

`nn.Module`
• Override the `forward` method to define how the computation is performed. Backward is automatically implemented with autograd.

• Override the `__init__` method, defining your parameters
  • remember to call the constructor of the super class!

• When you add a `Parameter` as an attribute it is automatically registered for you. It also works for submodules.

• If you want to add a list of parameters or modules use the `ParameterList` and `ModuleList` containers.
  • If you use a regular list the parameter will not be registered and cannot be iterated with the `parameters` method.

• You can print the network to see the registered parameters and submodules.

**HOW TO SUBCLASS**

`nn.Module`
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # remember to call the superclass
        # 1 input image channel, 6 output channels,
        # 5x5 square convolution
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a
        # single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

Example with a Custom Module

nn.Module

10/04/2024
```python
model = MyModel(**kwargs)

# Correct!
model.x = nn.ParameterList([
    torch.randn((10, 10), requires_grad=True),
    torch.randn((10, 10), requires_grad=True)
])

# Wrong!!!
model.x = [
    torch.randn((10, 10), requires_grad=True),
    torch.randn((10, 10), requires_grad=True)
]
```

**ParameterList usage**

```
```

**nn.Module**
• if we create a Net object and print it we obtain the following output:

```python
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)
)
```

• We can see the two convolutional layers and the three fully connected layers.
• To define a training loop we need a loss and an optimizer

• torch.nn defines many different loss functions
  • nn.MSELoss, nn.CrossEntropyLoss, nn.NLLLoss, nn.BCELoss, ...
  • You can also use the functional version, defined in nn.functional. The only difference is that you don’t need to create an object.
  • **Always** check to documentation for the correct shape and input arguments (does the loss needs logits or probabilities? Which dimension should be the last? Is the average for each element or for each sample?)

```python
import nn.functional as F
net = Net()
out = net(X)
loss = F.MSELoss(out, target)
```
• Simple gradient descent:

```python
sgd = torch.optim.SGD(model.parameters(), lr=0.01)
loss.backward()
sgd.step()
sgd.zero_grad()
```

• Note the call to the `zero_grad` method. It is needed to reset the gradient buffers.

• You can also use other optimizers defined in `torch.optim` (next slide):
  - Adam, RMSProp...
  - they take as arguments the learning rate, momentum, l2 weight decay.

**Optimizing the model with off-the-shelf optimizers**

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

for epoch in range(100):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(dataset):
        inputs, labels = data  # get the inputs
        optimizer.zero_grad()  # zero the parameter gradients

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.data[0]
        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

print('Finished Training')
• Available Modules:
  • Convolutional layers: Conv2D, MaxPool2D
  • Recurrent layers: RNN, LSTM, GRU, {RNN, LSTM, GRU}Cell
  • FeedForward: Linear
  • activation functions defined in torch.nn.functional

**Note: modules have train/eval mode**

• This is useful for layers (e.g. Dropout, BatchNormalization) that define a different behaviour during train and test

• Always set it during training with `net.train()`, and disable it during the test phase (`net.eval()`).
Feedforward Network

```python
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(100, 50),
    nn.ReLU(),
    nn.Linear(50, 50),
    nn.ReLU(),
    nn.Linear(50, 10),
    nn.Softmax()
)

y_out = model(X)
```
Recurrent Neural Network

{LSTM, RNN, GRU}Cell implement a recurrent layer. Combining them we can build a recurrent network.

The default input shape is (time, batch, features)

You also need to keep track of the hidden and cell states.

```python
model = torch.nn.LSTMCell(input_size, hidden_size)

out = []

h_prev = torch.zeros((batch_size, hidden_size))
c_prev = torch.zeros((batch_size, hidden_size))

for t in range(n_steps):
    X_t = X[t]
    h_prev, c_prev = model(X_t, (h_prev, c_prev))
    out.append(o_t)

out = torch.stack(out)
```
• For small datasets, and mostly in cases you want to go batch, you can load the data as a numpy array and convert it to a pytorch tensor (remember to check the dimensions in case you need to transpose some dimension)

• Most of the times, use the tools in in `torch.data.utils`

• `DataLoader` is used to automatize mini-batching, shuffling of the dataset, sampling techniques and any pre-processing. *Allows parallel loading*

• `Sampler` classes for sequential or random sampling from a dataset.

• Check the documentation: [http://pytorch.org/docs/stable/data.html](http://pytorch.org/docs/stable/data.html)
• PyTorch provides some guidelines regarding serialization
  • Save a model
    ```python
    torch.save(the_model.state_dict(), PATH)
    ```
  • Load back the model
    ```python
    the_model = TheModelClass(*args, **kwargs)
    the_model.load_state_dict(torch.load(PATH))
    ```
  • You can also use Tensorboard to log training metrics
    • https://pytorch.org/docs/stable/tensorboard.html
• Implement and train a Convolutional Neural Network to perform image classification on MNIST. Some guidelines:

• Use torchvision to download and use MNIST
  • Note that it’s a multi-class classification problem, so loss and output layer must be initialized accordingly

• Bonuses:
  • Monitor the performance with tensorboard
  • Use batch_norm

Your turn!