PyTorch

Introduction to Machine Learning – GIF-7015

Professor: Christian Gagné

Week 10

UNIVERSITÉ LAVAL

10.8 PyTorch basics

PyTorch

- Automatic differentiation library for deep learning
- Early October 2016 by a Facebook team
- Built over the Torch C engine
- More *pythonic* than TensorFlow
- Very close to numpy syntax
- Supports GPU computing (extremely fast, factor 10!)
- Supports dynamic graphs
- Stable and usable for large scale deployment

Tensor concept

- Pytorch is organized around tensor manipulation operations, including automatic derivation
  - Create a tensor from a list with `torch.<type>Tensor()`

```
import torch
a = torch.FloatTensor([[1,2,3], [2,3,4]])
```

  - Creating a random tensor

```
a = torch.randn(2, 3)
print(a)
```

```
>> tensor([[ 0.0991, -0.8607,  0.8124],
          [ 2.1726,  0.7590, -0.2185]])
```

  - Creating a tensor from a Numpy array

```
a = torch.from_numpy(numpy_array)
```

Tensor concept

- We can perform all kinds of operations on the tensors

```
a = torch.FloatTensor([[1,2,3], [2,3,4]])
b = torch.FloatTensor([[4,3,3], [5,3,4]])
c = a + b
print(c)
```

```
>> tensor([[5., 5., 6.],
          [7., 6., 8.]])
```

- Full list here: `https://pytorch.org/docs/stable/torch.html`

Automatic derivation

- During the application of the operations, PyTorch builds a graph of calculation
  - This graph allows to follow all the operations necessary to calculate the result
- Then, easy to automatically calculate the derivative at each step of the graph
  - To indicate the calculation of the derivative with respect to a certain tensor, use parameter `requires_grad`

```
a = torch.FloatTensor([[1,2,3], [2,3,4]], requires_grad=True)
```

  - Or once the tensor is in place

```
a.requires_grad = True
```

Example of a linear regression (1/2)

- Declare weight vector and random bias

```
# 10 dimensions
W = torch.randn(10, requires_grad=True)
b = torch.randn(1, requires_grad=True)
```

- Execute the chain of operation (very close to numpy)

```
# y_hat is the predicted output, x is the input
y_hat = W.dot(x) + b
```

- Calculate the quadratic error

```
# y is the desired output
err = 0.5 * (y_hat - y) ** 2
```

Example of a linear regression (2/2)

- Derive the equation using the method `backward()`

```
err.backward()
```

- We can then recover derivatives in the tensors W and b.

```
W_grad = W.grad
b_grad = b.grad
```

- Take a step in the right direction to do a gradient descent

```
W = W - eta * W.grad
b = b - eta * b.grad
```

Running on GPU

- All tensor operations can be easily performed on a GPU
  - PyTorch defines tensors `torch.cuda.<type>Tensor` in the same way as those previously seen
  - To translate a tensor from a non-GPU type (non-cuda) to a GPU type (cuda) and vice versa, simply use the method `to`:

```
a = a.to('cuda')   # to the GPU
a = a.to('cpu')    # back to the CPU
```

10.9 Defining a network

Defining a network

- PyTorch offers a way to easily declare networks
  - Defining a network with tensors directly would be a difficult task
  - Typically we use the package `torch.nn` and we inherit from `nn.Module`

```
import torch.nn as nn
class MonReseau(nn.Module):
    def __init__(self):
        super().__init__()
        # the network structure is defined here
        # - layers
        # - non-linear operations
        # - regularization methods
    def forward(self, x):
        # we make the inference here
```

Defining a network

- Several types of layers are available
  - Composition of simple modules to create more complex modules
- Examples of basic modules
  - Linear

    ```
    torch.nn.Linear(in_features, out_features, bias=True)
    ```

  - Convolution 2D

    ```
    torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
                    padding=0, dilation=1, groups=1, bias=True)
    ```

  - Dropout

    ```
    torch.nn.Dropout(p=0.5, inplace=False)
    ```

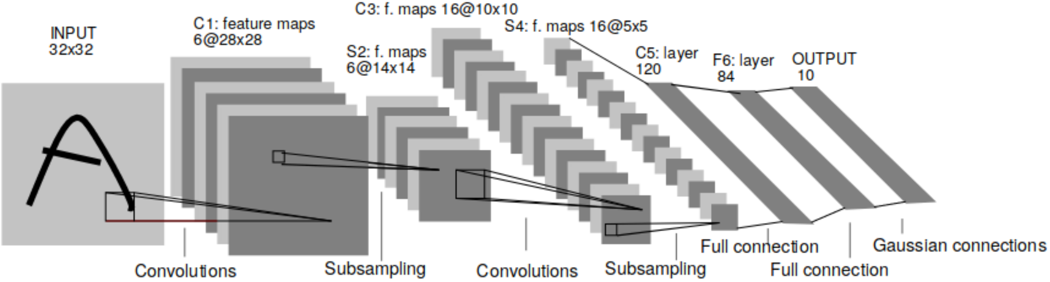
- See `https://pytorch.org/docs/stable/nn.html` for more details

Defining a network

- Most of the layers are also available in functions from `torch.nn.functional`
  - Warning: the module does not register these layers when they are declared directly as a function
  - Parameters of these layers are not taken into account in the list of parameters
  - Some layers like `dropout` or `batchnorm` have different behaviors in training and testing, changing network mode changes the behavior of a class layer, but not of a function layer
- It is therefore better to use function layers only when the layer has no parameters to be optimized and/or the same behaviour between training and test (e.g. activation function)

Defining a network

Let's assume the LeNet-5 network



From *Y. LeCun, L. Bottou, Y. Bengio et P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE, 86(11), 1998.* Accessed online on November 6, 2020 at `http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf`.

Defining a network

- PyTorch implementation of the LeNet-5 network for a dataset with image on a *channel* (2D tensor)

```
import torch.nn as nn
import torch.nn.functional as F
class Lenet5(nn.Module):
    def __init__(self):
        super().__init__()
        self.C1 = nn.Conv2d(1, 6, kernel_size=5)
        self.S2 = nn.MaxPool2d(2)
        self.C3 = nn.Conv2d(6, 16, kernel_size=5)
        self.S4 = nn.MaxPool2d(2)
        self.C5 = nn.Linear(16*4*4, 120)
        self.F6 = nn.Linear(120, 64)
        self.output = nn.Linear(64, 10)

        [...]
```

Defining a network

```
        [...]

    def forward(self, x):
        y = self.S2(F.relu(self.C1(x)))
        y = self.S4(F.relu(self.C3(y)))
        y = y.view(-1, 16*4*4) # resizing
        y = F.relu(self.C5(y))
        y = F.relu(self.F6(y))
        return self.output(y)
```

Defining a network

- In the same way, it is very easy to send a network to the GPU with the `to` method:

```
model = Lenet5()
model.to('cuda') # to the GPU
model.to('cpu')  # back to the CPU
```

- It is also possible to change the network mode, which will change the behaviour of some layers, like this:

```
model = Lenet5()
model.train() # in training mode
model.eval()  # in test mode
```

10.10 Handling datasets

Load and manipulate data

- Class to manage datasets:

```
torch.utils.data.Dataset
```

  - Must define a method `__getitem__(self, index)` to access an instance
  - Must define a method `__len__(self)` to return the size of the dataset

- Class to load batches of data:

```
torch.utils.data.DataLoader
```

  - Must receive a `Dataset` object and a `batch_size`, other arguments allow advanced options
  - `DataLoader` is a python iterator

Load and manipulate images

- Subpackage `torchvision` implements several useful functions for digital vision and image processing
  - `torchvision.datasets` allows to download several popular datasets such as MNIST, CIFAR or SVHN
  - `ImageFolder` and `DatasetFolder` allow to easily load a dataset organized in directories
  - `torchvision.transforms` implements transformations on images
  - `ToTensor` converts to a PyTorch tensor
  - `Normalize` allows to normalize a PyTorch tensor
- Several other functions available, see `https://pytorch.org/vision/stable/datasets.html`

Example with MNIST

```python
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor

batch_size = 64

# download to 'path/to/data'
train_set = MNIST('path/to/data', train=True, transform=ToTensor(), download=True)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
```

10.11 Training a network

Training a network

- Once the data has been loaded, an optimizer and error function is needed to do the training
  - Optimizers in `torch.optim`
  - Error functions in `torch.nn`, such as layers
- For example, to perform multi-class classification, we could use
  - Optimizer by stochastic gradient descent `torch.optim.SGD`
  - Cross-entropy `torch.nn.CrossEntropyLoss`

Training a network

- LeNet-5 training in classification

```
nb_epoch = 10
batch_size = 64
learning_rate = 0.01
momentum = 0.9
# download to 'path/to/data'
train_set = MNIST('path/to/data', train=True, transform=ToTensor(),
                  download=True)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
model = Lenet5()
model.train() # put in training mode
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
                            momentum=momentum)
criterion = torch.nn.CrossEntropyLoss()

[...]
```

Training a network

```
[...]

for i_epoch in range(nb_epoch):
    for i_batch, batch in enumerate(train_loader):
        X, y = batch
        optimizer.zero_grad()       # important! reset the gradients to 0
        y_hat = model(X)            # compute the predictions
        loss = criterion(y_hat, y)  # compute the error
        loss.backward()             # derive the graph
        optimizer.step()            # perform an optimization step
```

Use a pre-trained network

- Possible to backup a network via its state dictionary (`state_dict`) and function `torch.save`

```
state = model.state_dict()
torch.save(state, 'path/to/model')
```

- In the same way, it is possible to load a pre-trained model with the function `torch.load` and the method `load_state_dict`

```
state = torch.load('path/to/model')
model.load_state_dict(state)
```

- It's wise to load a network with a destination indication to first make sure it is on the CPU

```
state = torch.load('path/to/model', map_location=lambda storage, loc: storage)
```

- More details: `https://bit.ly/2Pu0Ibm`

Use a pre-trained network

- Subpackage `torchvision.models` implements several models useful for vision tasks.
- Can be loaded with pre-trained weights on the huge ImageNet natural image dataset
- For example, it is possible to load a ResNet-18 with the pre-trained weights as follows:

```
from torchvision.models import resnet18
model = resnet18(pretrained=True)
```

Use a pre-trained network

- Access to network parameters with their layer name with the method `named_parameters()`
  - Thus, it is possible to analyze the network

```
for name, param in model.named_parameters():
    print(name)
    print(param.grad)
```

  - To modify a network

```
model.nom_de_couche = NouvelleCouche()
```

  - And to freeze layers:

```
for name, param in model.named_parameters():
    if name == nom_de_couche_a_geler:
        param.requires_grad = False
```

Use a pre-trained network

- If you freeze layers, then it is important to only give the parameters to be optimized to the optimizer

```
params = filter(lambda x: x.requires_grad, model.parameters())
optimizer = torch.optim.SGD(params, lr=learning_rate, momentum=momentum)
```