3. Deep learning tools

3.11 Pytorch

Manel Martínez-Ramón
Meenu Ajith
Aswathy Rajendra Kurup

Introduction

- Developed by Facebook's AI Research Lab in 2016.
- It offers a substitution for NumPy by using GPU-optimized tensors.
- Pytorch has automatic differentiation to train neural networks by automatically computing the gradients.
- The graphs are built dynamically to allow changes during run time.

Elements of Pytorch

Tensors

- Tensors are similar to Numpy arrays, but they use both CPU and GPU.
- Tensors are initially assigned to the GPU memory with the help of a specific API named CUDA, developed for GPU parallel computing by NVIDIA.
- If no GPUS are available, CPUs are used.
- Currently, the CUDA API is limited to NVIDIA GPUs.

```
1 s=torch.tensor([[8,2,3],[1, 2, 3]])
2 print(s)
3 print('#dims:' ,s.ndim)
```

```
1 tensor([[8, 2, 3],
2         [1, 2, 3]])
3 #dims: 2
```

Elements of Pytorch

Variables

- Variables use the same API for operations but PyTorch has an Autograd package for the variables to compute the gradients automatically.
- If v is a variable, the tensor value and the gradient of v can be accessed by $p.data$ and $p.grad$ commands respectively.
- In a training, the variables are used in a directed acyclic graph to compute the output. Then, the *backward()* command constructs a graph to with the gradients to compute the backpropagation.
- Trainable variables are initialized by passing the parameter $requires\_grad = \text{True}$.

- In this example from Python.org, a very complex pre-trained neural network is imported (more details on it very soon):

```
1 import torch
2 from torchvision.models import resnet18, ResNet18_Weights
3 model  = resnet18(weights=ResNet18_Weights.DEFAULT)
4 data   = torch.rand(1, 3, 64, 64)
5 labels = torch.rand(1, 1000)
6 optim = torch.optim.SGD(model.parameters(), lr=1e-2,
                          momentum=0.9)
```

- We want to retrain the network with the synthesized data using a stochastic gradient descent (SDG) algorithm (do not worry about this line just yet).

- Thus, we first compute the corresponding output.

```
prediction = model(data) # forward pass
```

- Then we define the loss and we apply a backward pass to it

```
loss = (prediction - labels).sum()
loss.backward() # backward pass
```

- This propagates the error backwards by using a graph with the gradients of the variables.
- Finally, we update the weights

```
 optim.step() #gradient descent
```

- The process should be repeated until a stop criterion is satisfied.

Elements of Pytorch

What sorcery is this?

- Well, let us define **w**, **x**, y, and an arbitrary error

$$E = x_1 w_1^3 - x_2 w_2^2 - y \tag{1}$$

and then, its gradient will be defined by

$$\begin{aligned} \frac{\partial E}{\partial w_1} &= 3x_1 w_1^2 \\ \frac{\partial E}{\partial w_2} &= -2x_2 w_2 \end{aligned} \tag{2}$$

- Let us give two instance values to x_1 and x_2 and define E in Pytorch:

```
1 w = torch.nn.Parameter(data=torch.Tensor([2., 3.]),
                                        requires_grad=True)
2 x=torch.Tensor([1,2])
3 E=x[0]*w[0]**3-x[1]*w[1]**2-y
```

Elements of Pytorch

What sorcery is this?

- Notice that we pass the attribute *requires_grad*.

```
w = torch.nn.Parameter(data=torch.Tensor([2., 3.]),
                                         requires_grad=True)
x=torch.Tensor([1,2])
E=x[0]*w[0]**3-x[1]*w[1]**2-y
optim = torch.optim.SGD([w], lr=1, momentum=0)
```

- Define a loss as the sum of errors and execute the backward pass.

```
loss = E.sum()
loss.backward()
```

- Let us check the *.grad* attributes and compare them with Eq. (2).

```
print(w.grad)
```

```
tensor([ 12., -12.])
```

Elements of Pytorch

What sorcery is this?

- Pytorch has computed the gradients of the loss function with respect to each one of the parameters and stored in the corresponding attributes. These gradients can be now used to change the values of the parameters by gradient descent.

```
print('Before update', w)
optim.step()
print(w.grad)
print('After update', w)
```

```
Before update: Parameter containing:
tensor([2., 3.], requires_grad=True)
Gradient: tensor([ 12., -12.])
After update: Parameter containing:
tensor([-10.,  15.], requires_grad=True)
```

- In this example we defined the model parameters, by explicitly using method *.nn.Parameter* and we passed the attribute $requires\_grad = True$.
- The parameters must be sent to the optimized as an iterable object (an array) by using the *nn* module (introduced next).
- We also defined the algorithm for optimization as SDG, but other variants are available.

Elements of Pytorch

The *nn* package

- All the elements of a network including the learnable parameters should inherit from the *nn* package.
- It provides a higher level of abstraction over the graphs that are used for creating the neural networks.
- The *nn* package contains the modules needed to implement neural networks, as neural network layers, loss functions, and others, including different kinds of containers for structures (e.g. a sequential structure).
- See https://pytorch.org/docs/stable/nn.html.