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.

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)
```

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 $requires_grad = True$.

Elements of Pytorch Autograd

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

```
import torch
from torchvision.models import resnet18, ResNet18_Weights
model = resnet18(weights=ResNet18_Weights.DEFAULT)
data = torch.rand(1, 3, 64, 64)
labels = torch.rand(1, 1000)
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).

Autograd

• Thus, we first compute the corresponding output.

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

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

```
1 loss = (prediction - labels).sum()
2 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.

What sorcery is this?

 \bullet Well, let us define \mathbf{w} , \mathbf{x} , \mathbf{y} , and an arbitrary error

$$E = x_1 w_1^3 - x_2 w_2^2 - y (1)$$

and then, its gradient will be defined by

$$\frac{\partial E}{\partial w_1} = 3x_1 w_1^2
\frac{\partial E}{\partial w_2} = -2x_2 w_2$$
(2)

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

What sorcery is this?

• Notice that we pass the attribute requires_grad.

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

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

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

```
1 print (w.grad) 1 tensor([ 12., -12.])
```

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)
print(w.grad)
print('After update', w)
```

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

What sorcery is this?

- 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.

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.