PyTorch Short Tutorial

Mirela Popa

Mirela.popa@maastrichtuniversity.nl

PyTorch

- A popular open-source deep learning framework, providing fast and flexible implementations of various problems in CV and not only,
- Several tutorials and examples available on pytorch.org,
- Prerequisites: Python and Anaconda framework
- Pytorch – replacement of numpy for taking advantage of GPU power (import torch)
- Offers dynamic computational graphs (can be changed during runtime).

PyTorch

- Levels of abstraction:
  - Tensor – n-dimensional array running on GPU
  - Variable – node in the computational graph, used to store data and gradient
  - Module - layer in the neural network, which can store state and weights

PyTorch - Tensors

- In Numpy we have ndarrays – the equivalent in Pytorch are **tensors** which are multi-dimensional matrices containing elements of a single type (e.g. 9 CPU and 9 GPU tensor types),
  - x= torch.rand(4,2)
  - x= torch.zeros(4,2, dtype=torch.long)
  - x = torch.tensor([2.3,1.2,1])
  - print(x)
- Tensor operations:
  - Addition - torch.add(x, y, out=result)
  - in-place: y.add_(x)
  - Indexing (the same as in Numpy)

Common Modules

- Autograd Module (store gradients for a particular tensor, when operations are performed on it)

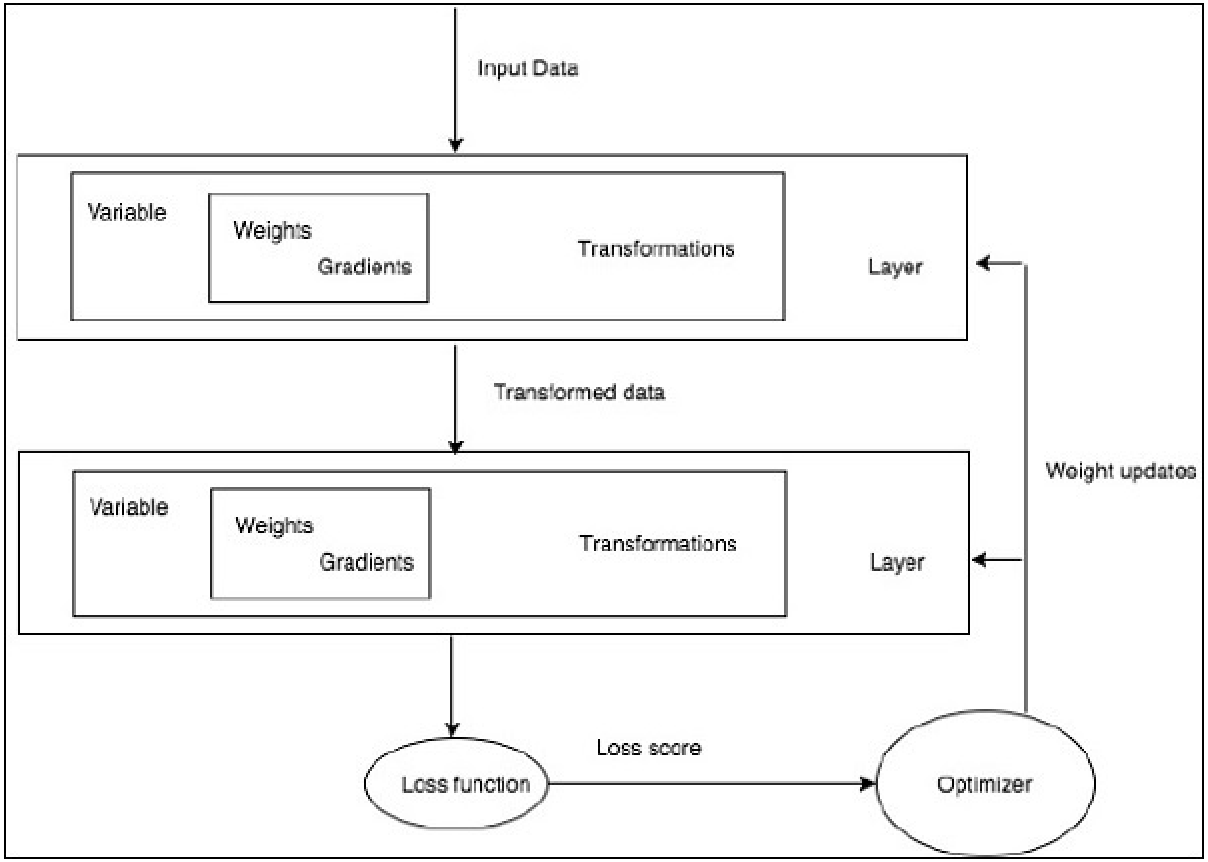
```
x=torch.randn(2,3,requires_grad=True)
```

- Optim Module (provides pre-implemented optimizers)

Includes SGD, Adam, Adadelta, Adagrad, SparseAdam, Adamax, Rprop (resilient backpropagation), etc.

```
from torch import optim
#adam
adam = optim.Adam(model.parameters(), lr=learning_rate)
```

PyTorch – Neural Networks



https://www.tutorialspoint.com/pytorch/pytorch_quick_guide.htm

PyTorch – Neural Networks

- Import the Neural Networks library:

```
import torch
import torch.nn as nn
```

- Define the input layer size, hidden layer size, output layer size and batch size (N):

```
D_in, H, D_out, N = 10, 5, 1, 10
```

- Create random input and output data:

```
x= torch.randn(N, D_in)
y=torch.randn(N, D_out)
```

PyTorch – Neural Networks

- Create a sequential model

```
model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out),
      )
```

- Construct the loss function (Mean Squared Error)

```
loss_fn = torch.nn.MSELoss(reduction='sum')
or loss_fn = torch.nn.CrossEntropy()
```

- Construct the optimizer

```
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

PyTorch – Neural Networks

- Implement the forward pass

```
for epoch in range(50):
        y_pred = model(x) # compute the prediction
        loss = loss_fn(y_pred, y) # compute the difference between
                                    prediction and ground truth
        if epoch % 10 == 9:
            print(t, loss.item())
        optimizer.zero_grad() # zero the gradients before running the
                                  backward pass
        loss.backward() #perform a backward pass

        optimizer.step() #update the parameters
```

PyTorch - Datasets

- The torchvision package is dedicated to loading and preparing datasets:
- Advantage: **efficient data generation scheme**
- (https://pytorch.org/docs/stable/torchvision/datasets.html)

trainset = torchvision.datasets.MNIST(root='./data', train = True, download=true, transform = transform)

Parameters

- **root** (*string*) – Root directory of dataset where `MNIST/processed/training.pt` and `MNIST/processed/test.pt` exist.
- **train** (*bool, optional*) – If True, creates dataset from `training.pt`, otherwise from `test.pt`.
- **download** (*bool, optional*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.

PyTorch - Datasets

- The torchvision package is dedicated to loading and preparing datasets:
  (https://pytorch.org/docs/stable/torchvision/datasets.html)

  ```
  trainset = torchvision.datasets.MNIST(root='./data', train =
  True, download=true, transform = transform)
  ```

- DataLoader – used to shuffle and batch the data

  ```
  trainloader = torch.utils.data.DataLoader(trainset,
        batch_size=…, shuffle = True, num_workers = ..)
  ```

PyTorch – CNN

- Load an existing dataset (eg. MNIST) - slides 10-11
- Create a CNN Model
- Train the model (define the criterion, optimizer) – similar to slide 8-9
- Test the model
- According to the results, fine-tune the parameters

CNN – Conv1d

Check the documentation for Convolutional layers:

https://pytorch.org/docs/stable/nn.html#convolution-layers

- Check and adjust the parameters for your CNN layers, taking into account: input channels, output channels – number of feature maps, the kernel size, the stride, the padding and the padding mode, whether to apply dilation, groups (which means that each input channel is convolved with its own set of filters):

    torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')

CNN Model

Decide the operations after applying each convolutional layer, such as normalization, ReLU or max pooling.

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=N):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(nn.Conv2d(input_dim, output_dim,
                kernel_size=.., stride=.., padding=..),
            nn.BatchNorm2d(output_dim),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=.., stride=..))
        self.layer2 = nn.Sequential(nn.Conv2d(input_dim2, output_dim2,
                kernel_size=..., stride=..., padding=..),
            nn.BatchNorm2d(output_dim2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=.., stride=..))
        self.fc = nn.Linear(output_dim3, num_classes)
```

Be careful about the size of the fully connected layer (FC).

CNN Model (2)

```
def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.reshape(out.size(0), -1)
    out = self.fc(out)
    return out
```

Check different loss functions and select the best one for your model.

```
model = ConvNet(num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),
lr=learning_rate)
```

Check torch.optim module for a complete set of optimization models.

CNN – test a model

After you achieved a satisfactory training set error (loss value defined in slide 9), you can check the model performance on the test set, by comparing the predictions - model(test_images) – with the ground truth.

```
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for img, labels  in test_loader:
            outputs = model(img)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print('Test Accuracy: {} %'.format(100 * correct / total))
```