lions@epfl

EE-556: MATHEMATICS OF DATA: FROM THEORY TO COMPUTATION
LABORATORY FOR INFORMATION AND INFERENCE SYSTEMS
FALL 2018

EPFL

INSTRUCTOR
PROF. VOLKAN CEVHER

HEAD TA
YA-PING HSIEH

PYTORCH TUTORIAL (FOR EXERCISE 2)

This tutorial introduces the basics of the deep learning framework **PyTorch 0.4** required for **Exercise 2**.

1 Installing PyTorch

The installation of PyTorch is straightforward via a package manager like *conda* (`https://conda.io`) or *pip*. The exact installation command depends on your OS, package manager, python version, and whether you have a GPU with CUDA support. In the PyTorch homepage (`https://pytorch.org`) you can find the correct command in your case. For example, if you use the conda package manager, running the following command in a Linux terminal

```
conda install pytorch-cpu torchvision-cpu -c pytorch
```

will install PyTorch without GPU support.

2 Defining a neural network architecture

In PyTorch it is easy to define a network architecture. We only have to define a subclass of the `torch.nn.Module` class and implement a *forward* method that defines the output of the network given an input. We can reuse the components in the `torch.nn` submodule and the functions in the `torch.nn.functional` submodule. The following example shows how to define a two layer fully connected network with ReLU activations.

```
import torch.nn.functional as F
from torch import nn

class TwoLayerNet(nn.Module):
    def __init__(self, input, hidden, output):
        """
        create two nn.Linear objects and assign them as attributes

        :param input: dimension of the input
        :param hidden: number of hidden neurons
        :param output: dimension of the output
        """
        super().__init__()
        self.linear1 = nn.Linear(input, hidden)
        self.linear2 = nn.Linear(hidden, output)

    def forward(self, x):
        """
        In the forward method we define what is the output of the network
        given an input x. In this example we use the ReLU as our activation function
        """
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x
```

Now we can create an instance of the architecture we just defined, and compute its output on some tensor. Notice that we can pass different values for the input dimension, number of hidden neurons and output dimension, when creating an instance of the class.

```
net = TwoLayerNet(input=784, hidden=100, output=10)
x = torch.randn(784)
result = net(x)
print('output of the network at input x: ' + str(result))
```

3 Loading the training data

PyTorch implements several classes that make it easy to work with large datasets. Also, the `torchvision` module lets us download popular benchmark datasets. The following example shows how to load the MNIST dataset, and create two instances of `torch.utils.data.Dataloader`, which is an object that lets us iterate through the dataset in batches. Because MNIST consists of image-label pairs, each batch is made of a tensor containing the images, and a tensor containing the corresponding labels.

```
import torch, torchvision
from torchvision import transforms

train_dataset = torchvision.datasets.MNIST(
        root='~/data',
        train=True,
        transform=transforms.ToTensor(),
        download=True)

train_loader = torch.utils.data.DataLoader(
        dataset=train_dataset,
        batch_size=100,
        shuffle=True)

for x, y in train_loader:
    print('batch size: ' + str(x.shape[0]))
    print('input dimension: ' + str(x[0].shape))
```

Note that the batch of data is the tensor `x` which has dimensions batch_size × input_dimension. In our particular example we have chosen a batch size of 100 and the MNIST images have dimension 28×28. The network we defined in section 2 has an input dimension of $784 = 28 \times 28$. In order to reshape the batch tensor `x` as a tensor of dimensions batch_size × 784 we can use the following

```
    x = x.view(x.shape[0], -1) # reshape 28x28 matrix to a 1x784 vector
```

with this shape we can feed the tensor to our network.

4 Defining a loss function and obtaining its gradient with respect to the network's parameters

Now that we have a neural network architecture and some labeled data, we turn to the task of choosing a set of parameters that make the network perform well on the image classification task. One popular loss function for image classification is the cross-entropy loss, and PyTorch makes it easy to obtain the gradient of such a function with respect to each of the network's parameters, in order to use first-order optimization methods. In the following example we will obtain the gradient on each mini-batch of the training data

```
for x, y in train_loader:
    loss_fn = nn.CrossEntropyLoss()
    x = x.view(x.shape[0], -1)
    net.zero_grad() # set the gradients to 0
    output = net(x)
    loss = loss_fn(output, y)
    loss.backward() # backpropagation
    for p in net.parameters():
        gradient = p.grad
        # perform an update based on the gradient
```

notice that the gradient is stored as an attribute `p.grad` on each of the network's parameters. Also, we have to set all the gradients to zero with the line `net.zero_grad()`. This is because everytime the method `loss.backward()` is called, the gradients are accumulated rather than replaced.

5 Using Pytorch's Optimizer class

In order to provide useful abstractions for an optimization algorithm, PyTorch provides the `Optimizer` interface as well as different classes implementing well known methods like SGD or ADAM. This is provided in the `optim` submodule. You can import it as follows

```
from torch import optim
```

An optimizer takes as input a list of variables for an optimization problem (usually the parameters of a PyTorch module), and the hyperparameters of the optimization algorithm. For example if we want to train the weights of our neural network with SGD and learning rate (step size) 0.1 we would initialize an optimizer as follows:

```
optimizer = optim.SGD(net.parameters(), lr=0.1)
```

The Optimizer class works by calling two different methods `optimizer.zero_grad()` and `optimizer.step()`. The first one sets as zero all the gradients corresponding to the variables that were assigned at initialization (in this case, the network's parameters), and the second one updates the variables according to some rule. In the case of SGD this just means subtracting a multiple of the gradient from each variable. A typical training loop using an optimizer looks as follows:

```
for x, y in train_loader:
    loss_fn = nn.CrossEntropyLoss()
    x = x.view(x.shape[0], -1)  # reshape 28x28 image to a 1x784 vector
    optimizer.zero_grad()  # set the gradients to 0
    output = net(x)
    loss = loss_fn(output, y)  # compute loss function
    loss.backward()  # backpropagation
    optimizer.step()  # update
```

6 Some useful low-level functions

- `torch.zeros(*sizes)`: returns a tensor of zeros with dimensions specified by a list of integers.
- `torch.ones(*sizes)`: similar to `torch.zeros`
- `torch.sqrt(x)`: returns the element-wise square root of the tensor x.
- `(*, **, /, +, -)`: python's math operators work element-wise with tensors.
- see `https://pytorch.org/docs/stable/torch.html#math-operations` for a complete review of PyTorch low-level functions.