How to build CNNs using `PyTorch`?

Dataset

We will use the CIFAR10 dataset for this lab. You should be familiar with the dataset. We have used the dataset in `Lab_1: Introduction to PyTorch`. It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32.

We can use `torchvision` to load CIFAR10 easily. The output of `torchvision` datasets are `PILImage` images of range (0, 1). We need to transform them to `Tensors` of normalized range (-1, 1).

```
import torch
import torchvision
import torchvision.transforms as transforms

# load and transform the dataset.
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

# the labels of the dataset.
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

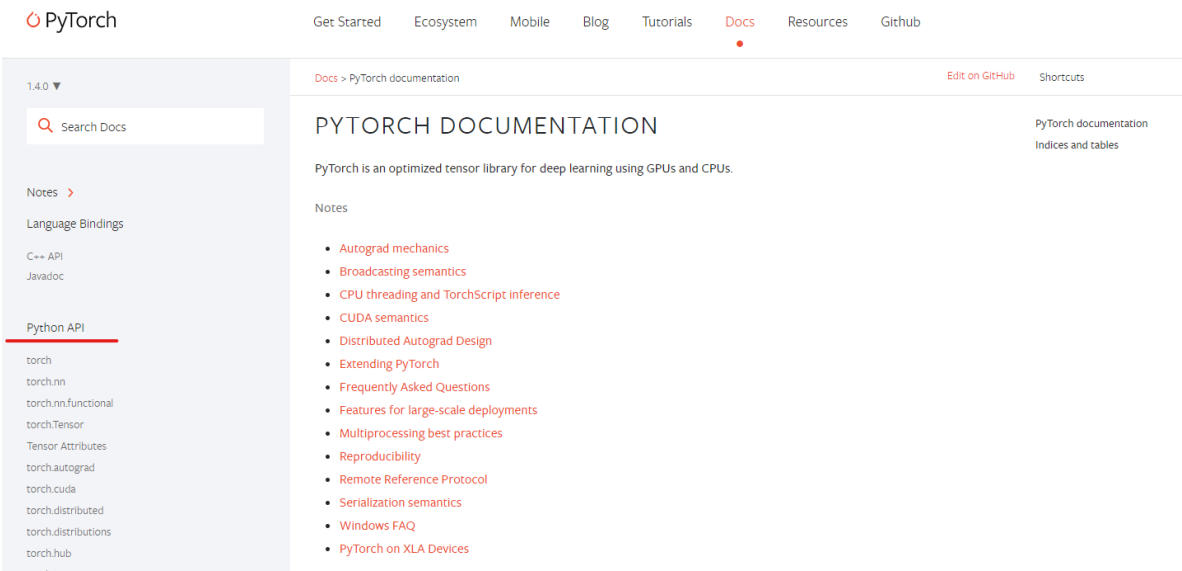
CNN

We will use the code in `cifar10_tutorial.ipynb` of `Lab_1: Introduction to PyTorch` to show how to build `CNNs` using `PyTorch`.

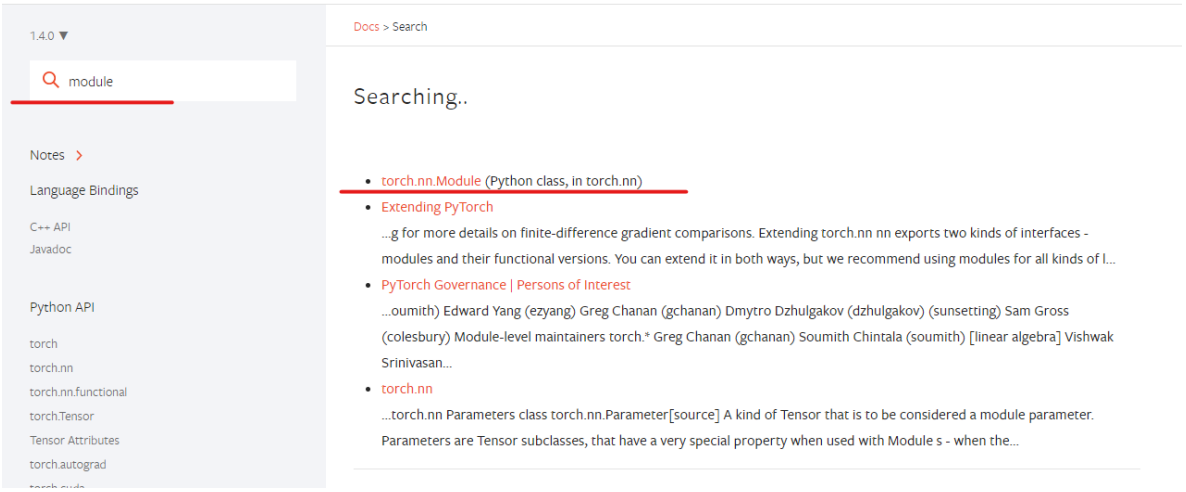
```
# import the modules we need
import torch.nn as nn
import torch.nn.functional as F

# the class Net should extend the Base class nn.Module
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # Convolution Layer: input_channels = 3, output_channels = 6,
kernal_size = 5 * 5
        self.conv1 = nn.Conv2d(3, 6, 5)

```

```
        # Pooling Layer: kernal_size = 2 * 2
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # Linear Layer: in_features = 400, out_features = 120
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # conv -> ReLU -> pool
        # After the operations, the size of x is: 14 * 14
        x = self.pool(F.relu(self.conv1(x)))

        # After the operations, the size of x is: 5 * 5
        x = self.pool(F.relu(self.conv2(x)))

        # Resize x
        x = x.view(-1, 16 * 5 * 5)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        x = self.fc3(x)
        return x

```

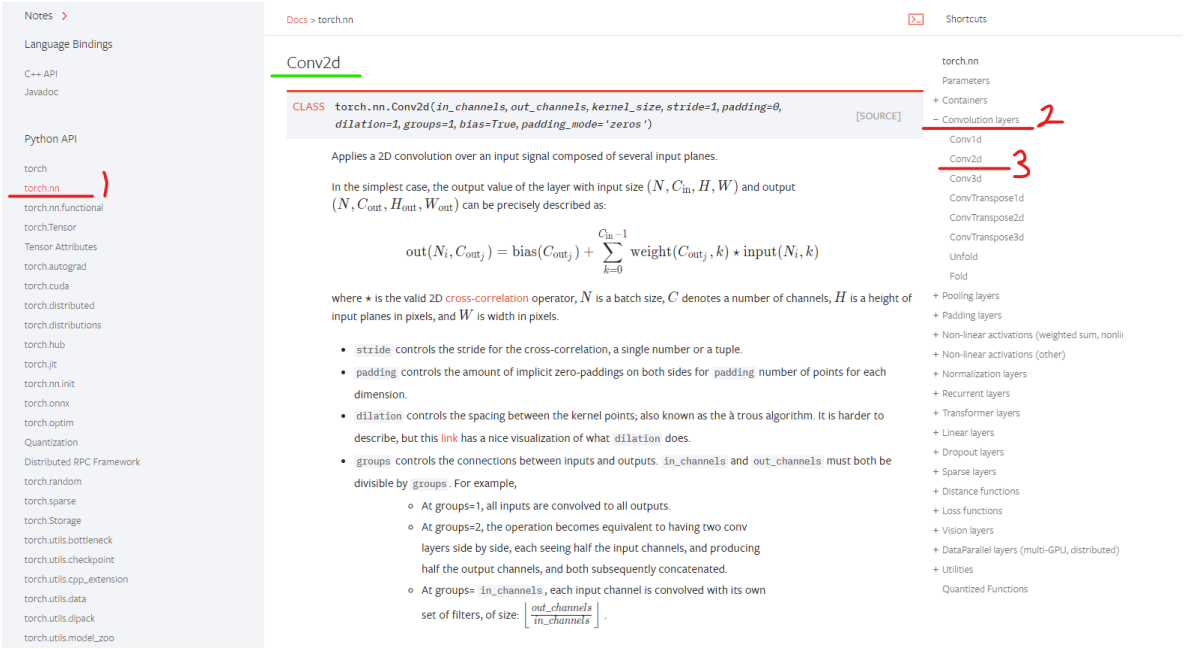
I hope that you can read the Docs of `PyTorch` when you have any problems. You can find the Docs in https://pytorch.org/docs/stable/index.html



You can search the class you want to find.



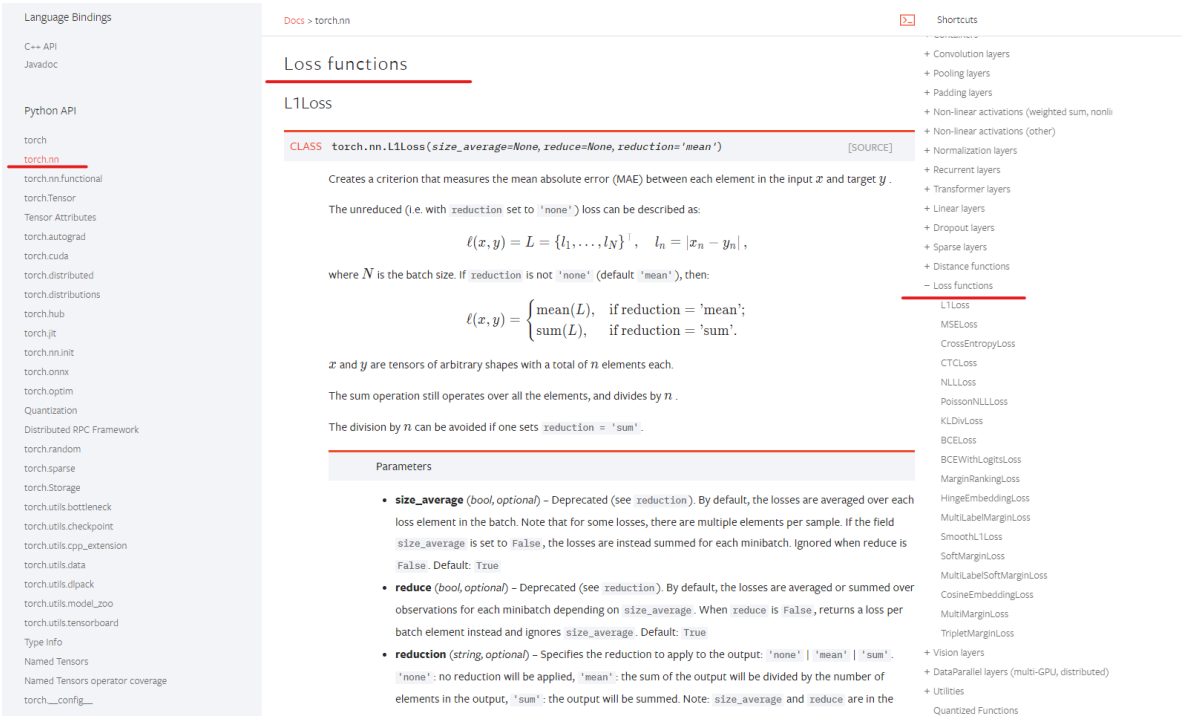
Let's try to search `Conv2d`. It is in `torch.nn`. Here we can find the definition of `Conv2d`. The Docs also provides examples of `Conv2d`.



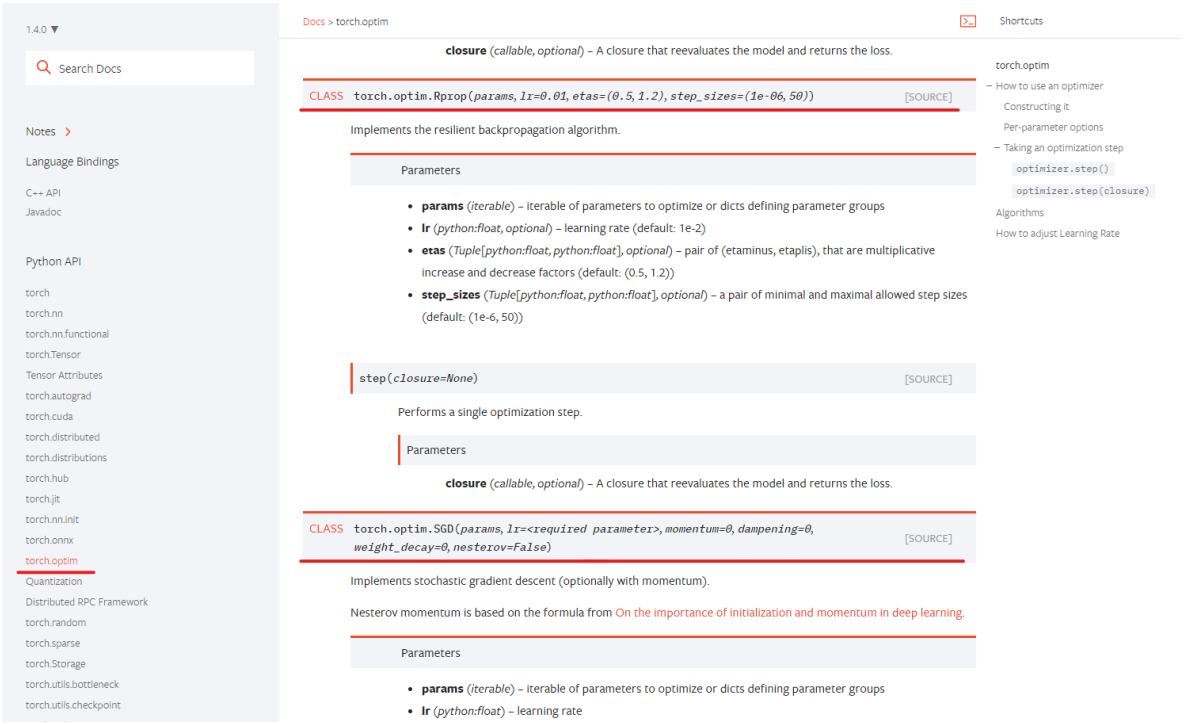
I suggest you to read the information of `MaxPool2d`. It is also in `torch.nn`.

Loss Function and optimizer

Then we need to define a loss function. `PyTorch` has defined common loss functions in `torch.nn`. You can select one loss function to use according to your projects.



`PyTorch` has defined common algorithms in `torch.optim`. I hope that you can read the definition of the algorithms that are common used, such as SGD, Adam.



Then we can use the following code to define a loss function and optimizer in our codes.

```
# Please import torch.optim at the beginning of the .py file.
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Now we can train the network.

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
```

```
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward
        outputs = net(inputs)

        # loss function
        loss = criterion(outputs, labels)

        # backward
        loss.backward()

        # update weights
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

Finally, we can test the network on the test set. In practice, we may need Normalization and Dropout. We can find the definitions in `torch.nn`.

We know how to build a network using `PyTorch` now. The most important I think is that you should learn to how to use the Docs of `PyTorch` when you do your projects.

Some Codes of our Lab

```
if __name__ == '__main__':
  # Command line arguments
  parser = argparse.ArgumentParser()
  parser.add_argument('--learning_rate', type = float, default = LEARNING_RATE_DEFAULT,
                      help='Learning rate')
  parser.add_argument('--max_steps', type = int, default = MAX_EPOCHS_DEFAULT,
                      help='Number of steps to run trainer.')
  parser.add_argument('--batch_size', type = int, default = BATCH_SIZE_DEFAULT,
                      help='Batch size to run trainer.')
  parser.add_argument('--eval_freq', type=int, default=EVAL_FREQ_DEFAULT,
                        help='Frequency of evaluation on the test set')
  parser.add_argument('--data_dir', type = str, default = DATA_DIR_DEFAULT,
                      help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()

  main()
```