

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).

```
1 import torch
2 import torchvision
3 import torchvision.transforms as transforms
4
5 # load and transform the dataset.
6 transform = transforms.Compose(
7     [transforms.ToTensor(),
8      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
9
10 trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
11                                         download=True, transform=transform)
12 trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
13                                           shuffle=True, num_workers=2)
14
15 testset = torchvision.datasets.CIFAR10(root='./data', train=False,
16                                         download=True, transform=transform)
17 testloader = torch.utils.data.DataLoader(testset, batch_size=4,
18                                          shuffle=False, num_workers=2)
19
20 # the labels of the dataset.
21 classes = ('plane', 'car', 'bird', 'cat',
22            '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`.

```
1 # import the modules we need
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 # the class Net should extend the Base class nn.Module
6 class Net(nn.Module):
7     def __init__(self):
8         super(Net, self).__init__()
9
10         # Convolution Layer: input_channels = 3, output_channels = 6,
11         # kernel_size = 5 * 5
12         self.conv1 = nn.Conv2d(3, 6, 5)
```

```

13     # Pooling Layer: kernal_size = 2 * 2
14     self.pool = nn.MaxPool2d(2, 2)
15     self.conv2 = nn.Conv2d(6, 16, 5)
16
17     # Linear Layer: in_features = 400, out_features = 120
18     self.fc1 = nn.Linear(16 * 5 * 5, 120)
19     self.fc2 = nn.Linear(120, 84)
20     self.fc3 = nn.Linear(84, 10)
21
22     def forward(self, x):
23         # conv -> ReLU -> pool
24         # After the operations, the size of x is: 14 * 14
25         x = self.pool(F.relu(self.conv1(x)))
26
27         # After the operations, the size of x is: 5 * 5
28         x = self.pool(F.relu(self.conv2(x)))
29
30         # Resize x
31         x = x.view(-1, 16 * 5 * 5)
32
33         x = F.relu(self.fc1(x))
34         x = F.relu(self.fc2(x))
35
36         x = self.fc3(x)
37         return x
38

```

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>



The screenshot shows the PyTorch documentation website. At the top, there's a navigation bar with links like 'Get Started', 'Ecosystem', 'Mobile', 'Blog', 'Tutorials', 'Docs' (highlighted), 'Resources', and 'Github'. Below this, the main content area is titled 'PYTORCH DOCUMENTATION' and includes a search bar. On the left, a sidebar shows the 'Python API' section expanded, listing various modules like `torch`, `torch.nn`, `torch.nn.functional`, `torch.Tensor`, `Tensor Attributes`, `torch.autograd`, `torch.cuda`, `torch.distributed`, `torch.distributions`, and `torch.hub`. The main content area lists 'Notes' with links to various topics like 'Autograd mechanics', 'Broadcasting semantics', 'CPU threading and TorchScript inference', 'CUDA semantics', 'Distributed Autograd Design', 'Extending PyTorch', 'Frequently Asked Questions', 'Features for large-scale deployments', 'Multiprocessing best practices', 'Reproducibility', 'Remote Reference Protocol', 'Serialization semantics', 'Windows FAQ', and 'PyTorch on XLA Devices'.

You can search the class you want to find.

1.4.0 ▼

Q module

Notes >
Language Bindings
C++ API
Javadoc
Python API
torch
torch.nn
torch.nn.functional
torch.Tensor
Tensor Attributes
torch.autograd
torch.utils

Docs > Search

Searching..

- torch.nn.Module (Python class, in torch.nn)
- Extending PyTorch

...g for more details on finite-difference gradient comparisons. Extending torch.nn nn exports two kinds of interfaces - modules and their functional versions. You can extend it in both ways, but we recommend using modules for all kinds of l...
- PyTorch Governance | Persons of Interest

...oumith) Edward Yang (ezyang) Greg Chanan (gchanan) Dmytro Dzhulgakov (dzhulgakov) (sunsetting) Sam Gross (colesbury) Module-level maintainers torch.* Greg Chanan (gchanan) Soumith Chintala (soumith) [linear algebra] Vishwak Srinivasan...
- torch.nn

...torch.nn Parameters class torch.nn.Parameter[source] A kind of Tensor that is to be considered a module parameter. Parameters are Tensor subclasses, that have a very special property when used with Module s - when the...

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`.

Notes >
Language Bindings
C++ API
Javadoc
Python API
torch
torch.nn
torch.nn.functional
torch.Tensor
Tensor Attributes
torch.autograd
torch.cuda
torch.distributed
torch.distributions
torch.hub
torch.jit
torch.nn.init
torch.onnx
torch.optim
Quantization
Distributed RPC Framework
torch.random
torch.sparse
torch.storage
torch.utils.bottleneck
torch.utils.checkpoint
torch.utils.cpp_extension
torch.utils.data
torch.utils.dlpack
torch.utils.model_zoo

Docs > torch.nn

Conv2d

```

CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
dilation=1, groups=1, bias=True, padding_mode='zeros')

```

[SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) * input(N_i, k)$$

where $*$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this [link](#) has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
 - At `groups=1`, all inputs are convolved to all outputs.
 - At `groups=2`, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
 - At `groups= in_channels`, each input channel is convolved with its own set of filters, of size: $\left\lfloor \frac{out_channels}{in_channels} \right\rfloor$.

Shortcuts

torch.nn
Parameters
Containers
- Convolution layers
Conv1d
Conv2d
Conv3d
Conv/Transpose1d
Conv/Transpose2d
Conv/Transpose3d
Unfold
Fold
+ Pooling layers
+ Padding layers
+ Non-linear activations (weighted sum, nonli
+ Non-linear activations (other)
+ Normalization layers
+ Recurrent layers
+ Transformer layers
+ Linear layers
+ Dropout layers
+ Sparse layers
+ Distance functions
+ Loss functions
+ Vision layers
+ DataParallel layers (multi-GPU, distributed)
+ Utilities
Quantized Functions

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.

Language Bindings

C++ API
Javadoc

Python API

torch

torch.nn

torch.nn.functional
torch.Tensor
Tensor Attributes
torch.autograd
torch.cuda
torch.distributed
torch.distributions
torch.hub
torch.jit
torch.nn.init
torch.nnix
torch.optim
Quantization
Distributed RPC Framework
torch.random
torch.sparse
torch.Storage
torch.utils.bottleneck
torch.utils.checkpoint
torch.utils.cpp_extension
torch.utils.data
torch.utils.dlpack
torch.utils.model_zoo
torch.utils.tensorboard
Type Info
Named Tensors
Named Tensors operator coverage
torch.__config__

Docs > torch.nn

Loss functions

L1Loss

CLASS `torch.nn.L1Loss(size_average=None, reduce=None, reduction='mean')` [\[SOURCE\]](#)

Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y .

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^T, \quad l_n = |x_n - y_n|,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

x and y are tensors of arbitrary shapes with a total of n elements each.

The sum operation still operates over all the elements, and divides by n .

The division by n can be avoided if one sets `reduction = 'sum'`.

Parameters

- size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when `reduce` is `False`. Default: `True`
- reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduction` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the

Shortcuts

- + Convolution layers
- + Pooling layers
- + Padding layers
- + Non-linear activations (weighted sum, nonli
- + Non-linear activations (other)
- + Normalization layers
- + Recurrent layers
- + Transformer layers
- + Linear layers
- + Dropout layers
- + Sparse layers
- + Distance functions
- Loss functions
- L1Loss
- MSELoss
- CrossEntropyLoss
- CTCLoss
- NLLLoss
- PoissonNLLoss
- KLDivLoss
- BCELoss
- BCEWithLogitsLoss
- MarginRankingLoss
- HingeEmbeddingLoss
- MultiLabelMarginLoss
- SmoothL1Loss
- SoftMarginLoss
- MultiLabelSoftMarginLoss
- CosineEmbeddingLoss
- MultiMarginLoss
- TripletMarginLoss
- + Vision layers
- + DataParallel layers (multi-GPU, distributed)
- + Utilities
- Quantized Functions

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.

1.4.0 ▼

Search Docs

Notes >

Language Bindings

C++ API
Javadoc

Python API

torch

torch.nn

torch.nn.functional
torch.Tensor
Tensor Attributes
torch.autograd
torch.cuda
torch.distributed
torch.distributions
torch.hub
torch.jit
torch.nn.init
torch.nnix

torch.optim

Quantization
Distributed RPC Framework
torch.random
torch.sparse
torch.Storage
torch.utils.bottleneck
torch.utils.checkpoint

Docs > torch.optim

closure (callable, optional) – A closure that reevaluates the model and returns the loss.

CLASS `torch.optim.Rprop(params, lr=0.01, etas=(0.5, 1.2), step_sizes=(1e-06, 50))` [\[SOURCE\]](#)

Implements the resilient backpropagation algorithm.

Parameters

- params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- lr** (*python:float, optional*) – learning rate (default: 1e-2)
- etas** (*Tuple[python:float, python:float], optional*) – pair of (etaminus, etaplus), that are multiplicative increase and decrease factors (default: (0.5, 1.2))
- step_sizes** (*Tuple[python:float, python:float], optional*) – a pair of minimal and maximal allowed step sizes (default: (1e-6, 50))

step(closure=None)

[SOURCE]

Performs a single optimization step.

Parameters

closure (callable, optional) – A closure that reevaluates the model and returns the loss.

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [\[SOURCE\]](#)

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from [On the importance of initialization and momentum in deep learning](#).

Parameters

- params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- lr** (*python:float*) – learning rate

Shortcuts

torch.optim

- How to use an optimizer
 - Constructing it
 - Per-parameter options
- Taking an optimization step
 - `optimizer.step()`
 - `optimizer.step(closure)`
- Algorithms
- How to adjust Learning Rate

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

```

1 # Please import torch.optim at the beginning of the .py file.
2 import torch.optim as optim
3
4 criterion = nn.CrossEntropyLoss()
5 optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

```

Now we can train the network.

```

1 for epoch in range(2): # loop over the dataset multiple times
2
3     running_loss = 0.0

```

```

4     for i, data in enumerate(trainloader, 0):
5         # get the inputs
6         inputs, labels = data
7
8         # zero the parameter gradients
9         optimizer.zero_grad()
10
11        # forward
12        outputs = net(inputs)
13
14        # loss function
15        loss = criterion(outputs, labels)
16
17        # backward
18        loss.backward()
19
20        # update weights
21        optimizer.step()
22
23        # print statistics
24        running_loss += loss.item()
25        if i % 2000 == 1999:    # print every 2000 mini-batches
26            print('[%d, %5d] loss: %.3f' %
27                  (epoch + 1, i + 1, running_loss / 2000))
28            running_loss = 0.0
29
30    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

```

1  if __name__ == '__main__':
2      # Command line arguments
3      parser = argparse.ArgumentParser()
4      parser.add_argument('--learning_rate', type = float, default =
LEARNING_RATE_DEFAULT,
5                          help='Learning rate')
6      parser.add_argument('--max_steps', type = int, default =
MAX_EPOCHS_DEFAULT,
7                          help='Number of steps to run trainer.')
8      parser.add_argument('--batch_size', type = int, default =
BATCH_SIZE_DEFAULT,
9                          help='Batch size to run trainer.')
10     parser.add_argument('--eval_freq', type=int, default=EVAL_FREQ_DEFAULT,
11                         help='Frequency of evaluation on the test set')
12     parser.add_argument('--data_dir', type = str, default = DATA_DIR_DEFAULT,
13                         help='Directory for storing input data')
14     FLAGS, unparsed = parser.parse_known_args()
15
16     main()

```