PyTorch Tutorial

May 21, 2025

Part 1: Tensor

dtype

1. torch.bool
2. torch.int32, torch.int64
3. torch.float32, torch.float64
4. torch.complex64

dtype

1. torch.bool
2. torch.int32, torch.int64
3. torch.float32, torch.float64
4. torch.complex64

Default floating dtpye is torch.float32, which can be modified by torch.set_default_dtype(torch.float64)

```
x = torch.ones(3, 1)
print (x.dtype)

torch.set_default_dtype(torch.float64)
x = torch.ones(3, 1)
print (x.dtype)

```

device

Data can be stored on CPU, GPU, ...

device

Data can be stored on CPU, GPU, ...

```
x = torch.ones(3, 1)

# show where the data is
print (x.device)

device = torch.device('cuda')

# move x to GPU
print (x.to(device))

# define a tensor on GPU
y = torch.ones(3, 1, device=device)

```

Part 2: Neural networks

Feedforward neural networks

$$f(x;\theta) = f_L(f_{L-1}(\cdots(f_1(x))\cdots)) \tag{1}$$

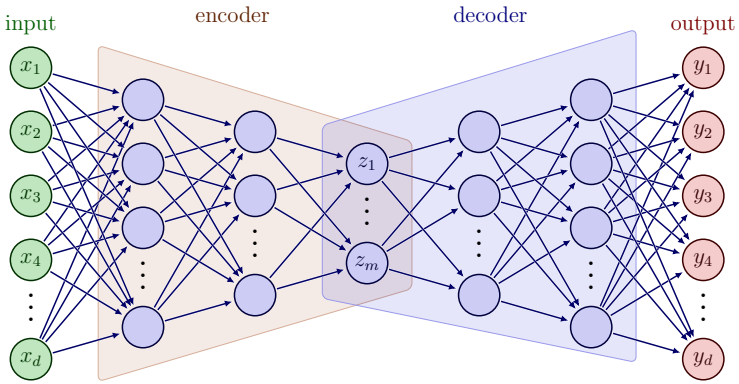
Recursive definition:

$$\begin{aligned} z_0 &= x \\ z_l &= f_l(z_{l-1}) = \varphi(b_l + W_l z_{l-1}), \quad l = 1, 2, \ldots, L-1\,, \\ z_L &= f_L(z_{l-1}) = b_L + W_L z_{L-1}\,. \end{aligned} \tag{2}$$

where

- $z_l, b_l \in \mathbb{R}^{d_l}$, $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$
- φ: activation function
- $\theta = (b_1, b_2, \ldots, b_L, W_1, \ldots, W_L)$

Feedforward neural networks



Building blocks

1. Containers: Sequential, ModuleList
2. nn.Linear
3. Activation: nn.ReLU, nn.Sigmoid, nn.Softplus, nn.Tanh
4. Convolution layers, dropout, recurrent[1]

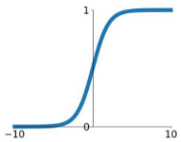
1. `https://docs.pytorch.org/docs/stable/nn.html`

Activation

Activation Functions

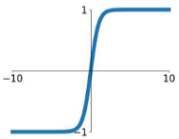
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



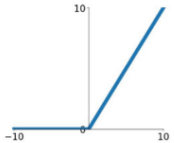
tanh

$\tanh(x)$



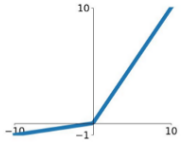
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

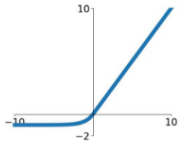


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



torch.nn

```
class MyNet(nn.Module):

    def __init__(self):
        super().__init__()

        self.net = nn.Sequential(
            nn.Linear(1, 3),
            nn.Tanh(),
            nn.Linear(3, 4),
            nn.Tanh(),
            nn.Linear(4, 1),
        )

    def forward(self, x):
        output = self.net(x)
        return output


model = MyNet()
```

torch.nn

A simpler way:

```
model = nn.Sequential(
    nn.Linear(1, 3),
    nn.Tanh(),
    nn.Linear(3, 4),
    nn.Tanh(),
    nn.Linear(4, 1),
)
```

Parameters of the model are PyTorch tensors.

```
# print all (training) parameters of the model
for param in model.parameters():
    print (param)
```

Parameters of the model are PyTorch tensors.

```
# print all (training) parameters of the model
for param in model.parameters():
    print (param)
```

Output:

```
Parameter containing:
tensor([[0.1453],
        [0.5682],
        [0.7924]], requires_grad=True)
        ...
```

Model evaluation

Model can be evaluated on mini-batches.

```
x = torch.ones(3, 1)

model(x)
```

Model evaluation

Input tensor and model tensors must have the same dtypes and devices.

```
# both x and tensors in the model have default dtype
↪  torch.float32
x = torch.ones(3, 1)
# works
model(x)

y = torch.ones(3, 1, dtype=torch.float64)
# doesn't work
model(y)
```

Example: positive function

Adding a positive activation to the output layer:

```
class MyNet(nn.Module):

    def __init__(self):
        super().__init__()

        self.net = nn.Sequential(
            nn.Linear(1, 3),
            nn.Tanh(),
            nn.Linear(3, 1),
            nn.Tanh(),
        )

    def forward(self, x):
        output = self.net(x)
        return output

model = MyNet()
```

Example: features as input

Transform input tensor to features before passing to model.

```
class MyNet(nn.Module):

    def __init__(self):
        super().__init__()

        self.net = nn.Sequential(
            nn.Linear(1, 3),
            nn.Tanh(),
            nn.Linear(3, 1),
            nn.Tanh(),
        )

    def forward(self, x):
        output = self.net(torch.sin(x))
        return output

model = MyNet()
```

Part 3: Automatic differentiation

computational graph

Whenever require_grad=True, PyTorch tracks the operations.

```
a = torch.tensor([2.0], requires_grad=True)

s = 0.5 * a ** 2

# requires_grad=True are set for parameters of model
 model = nn.Sequential(
    nn.Linear(1, 3),
    nn.Tanh(),
    nn.Linear(3, 4),
    nn.Tanh(),
    nn.Linear(4, 1),
)
```

computational graph

Backprogration:

```
a = torch.tensor([1.0], requires_grad=True)
b = torch.tensor([2.0])

s = 0.5 * a ** 2 + b

s.backward()
# output: 1 None
print (a.grad, b.grad)
```

computational graph

Gradient of **leaf nodes** will be accumulated in the grad field.

```
a = torch.tensor([1.0], requires_grad=True)

s = 0.5 * a ** 2
s.backward()
# gradient of s
print (a.grad)

s1 = 3 * a
# Gradients of s and s1 will be sumed up.
# To avoid this, write:
# a.grad = None
s1.backward()

print (a.grad)

```

computational graph

Backward propagation will free intermediate values of the graph. To backward twice, add retain_graph=True.

```
a = torch.tensor([1.0], requires_grad=True)

s = 0.5 * a ** 2

# add retain_graph=True to avoid error below
s.backward(retain_graph=True)

s1 = 3 * s
s1.backward()

print (a.grad)

```

computational graph

Avoid gradient computation:

```
a = torch.tensor([1.0], requires_grad=True)

b = torch.tensor([1.5], requires_grad=True)
b.requires_grad = False

s = 0.5 * a ** 2 + b**2
s.backward()
```

computational graph

Detach: remove nodes from computational graph.

```
a = torch.tensor([1.0], requires_grad=True)

b = a
# quadratic in a
s = a * b
s.backward()

b = a.detach()
# linear in a
s = a * b
s.backward()
```

autograd.grad

autograd.grad allows to compute gradients with given tensors.

Example: compute $\int_0^1 |f'(x)|^2 dx$, where $f : \mathbb{R} \to \mathbb{R}$ is the model.

```
x = torch.linspace(0, 1, 101).reshape(-1,1)
  # we need to compute derivative wrt x
x.requires_grad_()

# f
y = model(x)

y_grad_vec = torch.autograd.grad(outputs=y.sum(), inputs=x,
↪  retain_graph=True)[0]

int_val = torch.sum(y_grad_vec**2) * 0.01
```

Part 3: Dataset

different datasets

1. training dataset: training models
2. validation dataset: early stopping
3. test dataset: evaluation of the trained model

dataset splitting

Method 1: torch.utils.data.random_split

```
x = torch.linspace(0, 1, 101).reshape(-1,1)
seed = 42

ratios = [0.8, 0.1, 0.1]

train_data, val_data, test_data =
↪  torch.utils.data.random_split(x, ratios,
↪  generator=torch.Generator().manual_seed(seed))

training_set = x[train_data.indices]
val_set = x[val_data.indices]
test_set = x[test_data.indices]
```

dataset splitting

Method 1: train_test_split from sklearn

```
1 import numpy as np
2 from sklearn.model_selection import train_test_split
3
4 X, y = np.arange(10).reshape((5, 2)), range(5)
5
6 X_train, X_test, y_train, y_test = train_test_split(X, y,
  ↪  test_size=0.33, random_state=42)
7
8 print (X_train.shape, len(y_train))
```

DataLoader

Example 1:

```
1 from torch.utils.data import DataLoader
2
3 x = torch.linspace(0, 1, 101).reshape(-1,1)
4 batch_size = 21
5 data_loader = DataLoader(x, batch_size=batch_size,
  ↪  shuffle=True, drop_last=True)
6
7 for data in data_loader:
8     print (data.shape)
```

DataLoader

Example 2:

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

print (train_dataset.data.shape, test_dataset.data.shape)

train_loader = DataLoader(train_dataset, batch_size=1001,
    shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1001,
    shuffle=True)

for data in test_loader:
    img, label = data
    print (img.shape)
```

Part 3: Training

Optimizer

Optimization objective:

$$\mathcal{L}(\theta) := \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(x_n; \theta)).$$

Full gradient: $\nabla_\theta \mathcal{L}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \ell(y_n, f(x_n; \theta))$.

SGD:

$$\theta_{t+1} = \theta_t - \frac{\eta_t}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \nabla_\theta \ell(y_n, f(x_n; \theta_t))$$

Optimizer

Optimization objective:

$$\mathcal{L}(\theta) := \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(x_n; \theta)).$$

Full gradient: $\nabla_\theta \mathcal{L}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \ell(y_n, f(x_n; \theta))$.

SGD:

$$\begin{aligned}
\theta_{t+1} =& \theta_t - \frac{\eta_t}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \nabla_\theta \ell(y_n, f(x_n; \theta_t)) \\
=& \theta_t - \eta_t \nabla_\theta \mathcal{L}(\theta_t) + \eta_t \left(\nabla_\theta \mathcal{L}(\theta_t) - \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \nabla_\theta \ell(y_n, f(x_n; \theta_t)) \right) \\
=& \theta_t - \eta_t \nabla_\theta \mathcal{L}(\theta_t) + \eta_t e_t
\end{aligned}$$

Optimizer

SGD:

```
import torch.optim as optim

model = nn.Sequential(nn.Linear(1, 3), nn.Tanh(),
↪  nn.Linear(3, 4))

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

Adam:

```
criterion = nn.MSELoss(reduction='sum')
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

training loop

```
for epoch in range(total_epochs):    # for each epoch

    # loop over all mini-batches
    for data in data_loader:

        y_pred = model(data)
        y = fun(data)

        loss = criterion(y_pred, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```