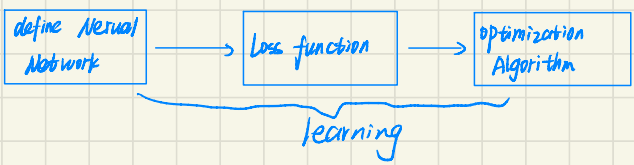
学习目标：

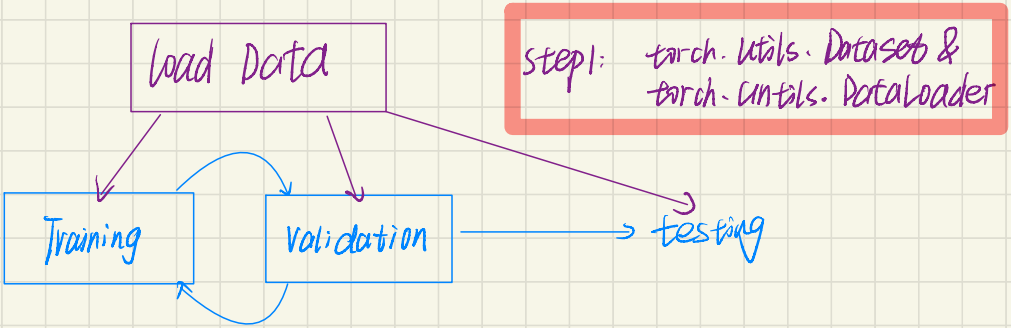
① Background: Prerequisties & what is pytorch?
② Training & testing Nerual Networks in pytorch
③ Dataset & DataLoader
④ Tensors
⑤ torch.nn: Models, Loss functions
⑥ torch.optim: Optimization
⑦ save / Load models

↓

① An machine learning framework in python

②





step1: torch.utils.Dataset & torch.cintils.DataLoader

③ Dataset: stores data samples and expected values
DataLoader: group data in batches, enables multiprocessing

eg: dataset = MyDataset(file)
dataloader = DataLoader(dataset, batch_size, shuffle=True)
Training: True
Testing: False

```
from torch.utils.data import DataSet, DataLoader

class MyDataset(DataSet):
    def __init__(self,file):
```

```
self.data = ...                          } Read data or process
def __getitem__(self, index):            } Returns one simple at a time
    return self.data[index]
def __len__(self):                       } Returns the size of the dataset
    return len(self.data)
```

④

- high-dimensional matrices (arrays)

1-D
audio

2-D
Black or white images

3-D
RGB images

- Check with .shape() 看数据维度
- 另外一些基本计算

transpose( ) 转置
squeeze(0) 消除维度值 dim=0
unsqueeze(1) 加个维度
cat( ) 合并
使用Gpu运算: x = x.to('cuda')
Cpu: x = x.to('cpu')

Tensor- Gradient Calculation

① x = torch.tensor([[1., 0.], [-1., 1.]], require_grad=True)
② z = x.pow(2).sum( )
③ z.backward( ) 计算梯度优化.
④ x.grad( )

① $x=\begin{bmatrix}1 & 0\\ -1 & 1\end{bmatrix}$
② $z=\sum\sum x_{ij}^2$
③ $\frac{\partial z}{\partial x_{ij}}=2x_{ij}$
④ $\frac{\partial z}{\partial x}=\begin{bmatrix}2 & 0\\ -2 & 2\end{bmatrix}$

⑤

Step 2: torch.nn.Module

input Tensor * × 32 → nn.Linear(32, 64) → output Tensor * × 64

layer = torch.nn.Linear(32, 64)

Activation Functions { nn.Sigmoid() / nn.ReLU()

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(10, 32),
            nn.Sigmoid(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.net(x)
```

Initialize your model & define layers

Compute output of your NN

=

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.layer1 = nn.Linear(10, 32)
        self.layer2 = nn.Sigmoid(),
        self.layer3 = nn.Linear(32,1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        return out
```

Step 3: Torch.nn.MSELoss / Torch.nn.CrossEntropy etc

- Mean Squared Error (for regression tasks)

  `criterion = nn.MSELoss()`

- Cross Entropy (for classification tasks)

  `criterion = nn.CrossEntropyLoss()`

- `loss = criterion(model_output, expected_value)`

(6) Step4: torch.optim

```
optimizer = torch.optim.SGD(model.parameters(), lr, momentum = 0)
```

- For every batch of data:
  1. Call `optimizer.zero_grad()` to reset gradients of model parameters.
  2. Call `loss.backward()` to backpropagate gradients of prediction loss.
  3. Call `optimizer.step()` to adjust model parameters.

Step5: Entire Procedure

Neural Network Training Setup

⇓

```
dataset = MyDataset(file)                              read data via MyDataSet
tr_set = DataLoader(dataset, 16, shuffle=True)         put dataset into DataLoader
model = MyModel().to(device)                           construct model and move to device (cpu/Gpu)
criterion = nn.MSELoss()                               set loss function
optimizer = torch.optim.SGD(model.parameters(), 0.1)   set optimizer
```

```
for epoch in range(n_epochs):                          iterate n_epochs
    model.train()                                      set model to train model
    for x, y in tr_set:                                iterate through the DataLoader
        optimizer.zero_grad()                          set gradients to zero
        x, y = x.to(device), y.to(device)
        pred = model(x)                                forward pass (compute output)
        loss = criterion(pred, y)                      compute loss
        loss.backward()                                compute gradient
        optimizer.step()                               update model with optimizer
```

Validation Loop

```
model.eval()                                           set model to evaluation mode
total_loss = 0
for x, y in dv_set:                                    iterate through the dataloader
    x, y = x.to(device), y.to(device)                  move data to device (cpu/cuda)
    with torch.no_grad():                              disable gradient calculation
        pred = model(x)                                forward pass (compute output)
        loss = criterion(pred, y)                      compute loss
    total_loss += loss.cpu().item() * len(x)           accumulate loss
    avg_loss = total_loss / len(dv_set.dataset)        compute averaged loss
```

Testing Loop

```
model.eval()                                           set model to evaluation mode
preds = []
for x in tt_set:                                       iterate through the dataloader
    x = x.to(device)                                   move data to device (cpu/cuda)
    with torch.no_grad():                              disable gradient calculation
        pred = model(x)                                forward pass (compute output)
        preds.append(pred.cpu())                       collect prediction
```

Notice:

- model.eval()

  Changes behaviour of some model layers, such as dropout and batch normalization.

- with torch.no_grad()

  Prevents calculations from being added into gradient computation graph. Usually use to prevent accidental training on validation/testing data.

⑦ Save/load models