



PPCES: Machine and Deep Learning

Case Study – PyTorch

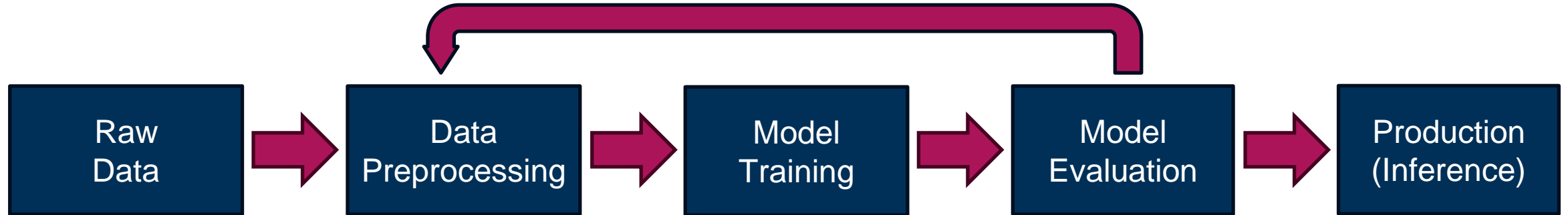


- **PyTorch was introduced in 2016**
 - Designed for Deep Learning workloads
 - Latest stable version: 2.6.0
 - Frontends for Python and C++
- **Extensive documentation and tutorials ([here](#))**
 - Performance tuning guides
 - Distributed training
- **Libraries (transform, datasets, models, ...)**
 - torchaudio (e.g., signal processing + sampling)
 - torchtext (e.g., tokenizers, metrics, vocabular)
 - torchvision (e.g., segmentation, video, image)

– Key points

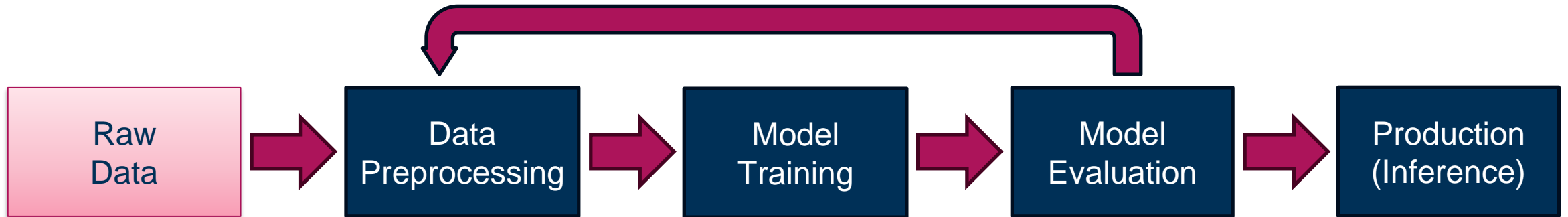
- High flexibility and freedom
- Define layers with nn package
- Manual training and evaluation loop
- Stable interface

```
# create model to be trained
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
# model training
for t in range(n_epochs):
    y_pred = model(data)           # forward pass
    loss = loss_fn(y_pred, y)     # determine loss
    optimizer.zero_grad()        # zero the gradients
    loss.backward()               # compute gradients
    optimizer.step()              # update weights
```



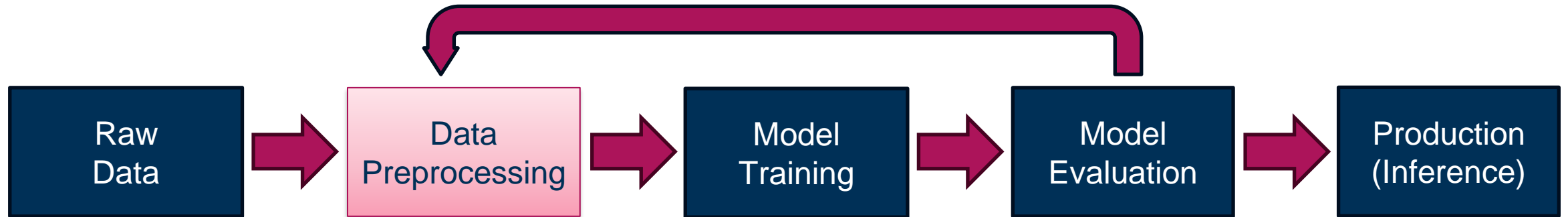
- **Build your pipelines**

- Data pipeline
- Training pipeline



- **Utilize your own data dataset**
 - Some interfaces work with X (features) and y (labels)
 - However, more complex data: Images, time series
 - Use Dataset classes from `torch.utils.data` ([Link](#))
- **Predefined datasets for vision, audio, ...**
 - ImageNet
 - CIFAR (10 and 100)
 - MNIST
 - ...

<code>Caltech101(root[, target_type, transform, ...])</code>	Caltech 101 Dataset.
<code>Caltech256(root[, transform, ...])</code>	Caltech 256 Dataset.
<code>CelebA(root[, split, target_type, ...])</code>	Large-scale CelebFaces Attributes (CelebA) Dataset Dataset.
<code>CIFAR10(root[, train, transform, ...])</code>	CIFAR10 Dataset.
<code>CIFAR100(root[, train, transform, ...])</code>	CIFAR100 Dataset.
<code>Country211(root[, split, transform, ...])</code>	The Country211 Data Set from OpenAI.



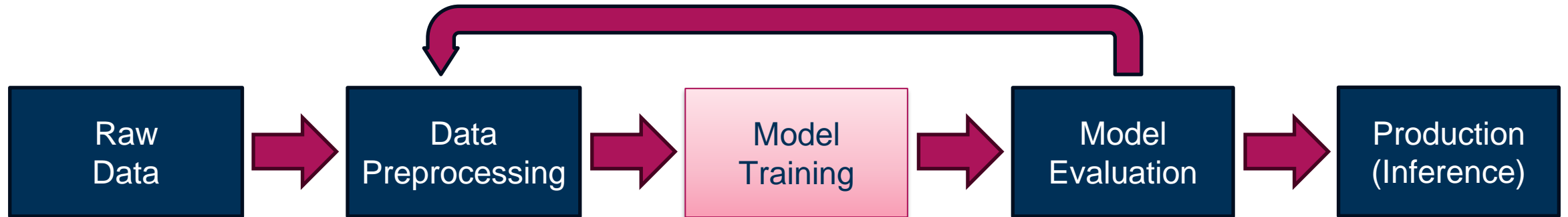
– Various map / transformation operations available

- Manual data manipulations (e.g., standardization)
- Apply raw functions in `torch.nn.functional` to tensors
- Several transformations in `torchvision.transforms`
 - Normalize
 - RandomHorizontalFlip
 - Resize
 - Padding
 - Full list → [here](#)
- `DataLoader` (batching, shuffling)

```
import torchvision.transforms as trf

transf = trf.Compose([
    trf.RandomHorizontalFlip(),
    trf.ToTensor(),
    trf.Normalize(mean, std)
])
```

```
import torch
# create data loader
loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=32,
    shuffle=False,
    num_workers=0,
    pin_memory=False,
)
# iterate data loader
for idx, samples in enumerate(loader):
    print(batch_idx, samples)
```



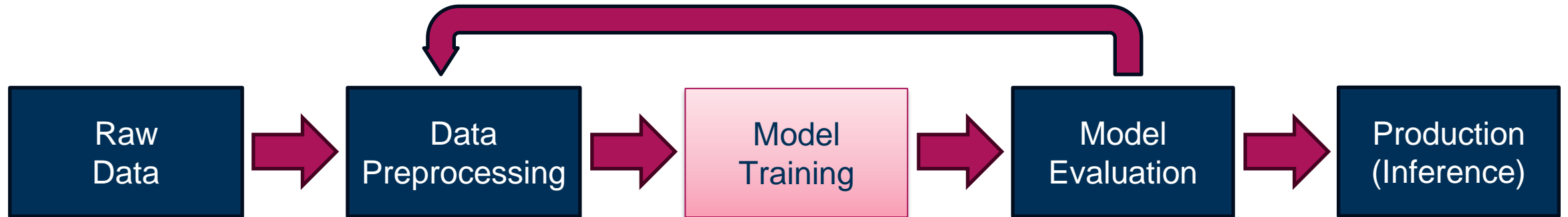
- **Loss function** (MSE, CrossEntropy, ...)
- **Optimizers** (SGD, Adam, ...)
- **Create or load a model**
 - **Sequential API**
 - **Functional API: Define your own `torch.nn.Module`**
 - **Predefined modules ([Catalogue](#))**
 - AlexNet, ResNet, VGG, ShuffleNet, ...

```
# create model to be trained
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
```

```
import torch

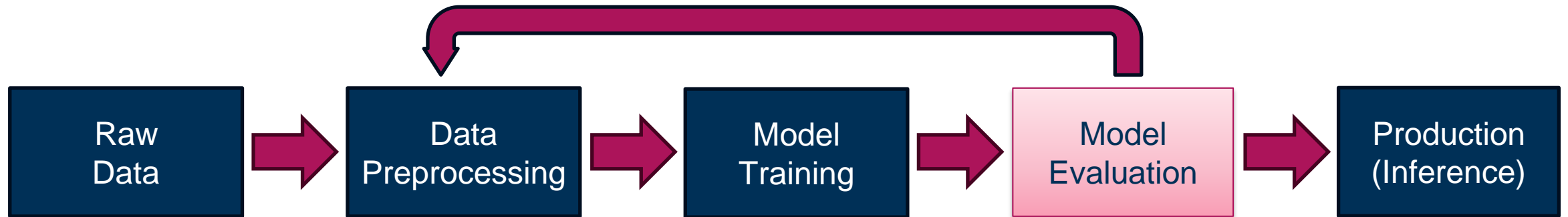
class TinyModel(torch.nn.Module):
    def __init__(self):
        super(TinyModel, self).__init__()
        self.linear1 = torch.nn.Linear(100, 200)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(200, 10)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.softmax(x)
        return x
```



- **Loss function** (MSE, CrossEntropy, ...)
- **Optimizers** (SGD, Adam, ...)
- **Create or load a model**
 - Sequential API
 - Functional API: Define your own `torch.nn.Module`
 - Predefined modules ([Catalogue](#))
 - AlexNet, ResNet, VGG, ShuffleNet, ...

```
# create model to be trained
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
# model training
for t in range(n_epochs):
    y_pred = model(data)           # forward pass
    loss = loss_fn(y_pred, y)     # determine loss
    optimizer.zero_grad()        # zero the gradients
    loss.backward()              # compute gradients
    optimizer.step()             # update weights
```



- **Standard PyTorch:** Explicit training / test loops
 - Full control / allows customization
- **Extensions:** `torcheval` or `TorchMetrics`
 - `BinaryAccuracy`, `BinaryPrecision`
 - `MulticlassAccuracy`, `MulticlassPrecision`
 - `MeanSquaredError`, `R2Score`

```
# calculate accuracy in a single line  
acc = (y_pred.round() == y_batch).float().mean()
```

- **Metrics**
 - `Aggregation Metrics`
 - `Classification Metrics`
 - `Ranking Metrics`
 - `Regression Metrics`
 - `Text Metrics`
 - `Windowed Metrics`

([Doc torcheval](#))

([Doc TorchMetrics](#))



Example: Classification with MNIST ([Source Code](#))

```
def main():
    # parsing arguments and other preparations
    ...

    # load and preprocess MNIST dataset
    loader_train, loader_test = get_data_loaders(args)

    # load the model
    model = Net().to(device)

    # create optimizer instance
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    # training
    for epoch in range(1, args.epochs + 1):
        train(args, model, device, loader_train, optimizer, epoch)
        test (args, model, device, loader_test)
```

```
def get_data_loaders(args):
    # define arguments for data loaders
    ds_kwargs = {'batch_size': args.batch_size}

    # transformation
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    # train and test dataset
    dataset1 = datasets.MNIST('../data', train=True, download=True, transform=transform)
    dataset2 = datasets.MNIST('../data', train=False, transform=transform)
    # create data loaders
    loader_train = torch.utils.data.DataLoader(dataset1, **ds_kwargs)
    loader_test = torch.utils.data.DataLoader(dataset2, **ds_kwargs)
    return loader_train, loader_test
```

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

```
def train(args, model, device, loader_train, optimizer, epoch):
    # set model into training mode
    model.train()

    # iterative over batches
    for batch_idx, (data, target) in enumerate(loader_train):
        data, target = data.to(device), target.to(device) # move them to GPU
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

    if batch_idx % args.log_interval == 0:
        print('Train Epoch: {} [{} / {} ( {:.0f} % )] \t Loss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(loader_train.dataset),
            100. * batch_idx / len(loader_train), loss.item()))
```

Example: Classification with MNIST ([Source Code](#))

```
def test(args, model, device, loader_test):
    # set model into evaluation mode
    model.eval()
    test_loss, correct = 0.0, 0.0
    with torch.no_grad():
        for data, target in loader_test:
            data, target = data.to(device), target.to(device) # move them to GPU
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(loader_test.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(loader_test.dataset),
        100. * correct / len(loader_test.dataset)))
```