# Poutyne

# An Easy and Flexible Deep Learning Framework for PyTorch

Frédérik Paradis*, David Beauchemin, Mathieu Godbout, François Laviolette

Department of Computer Science and Software Engineering, Université Laval

*frederik.paradis.1@ulaval.ca

June 9, 2020

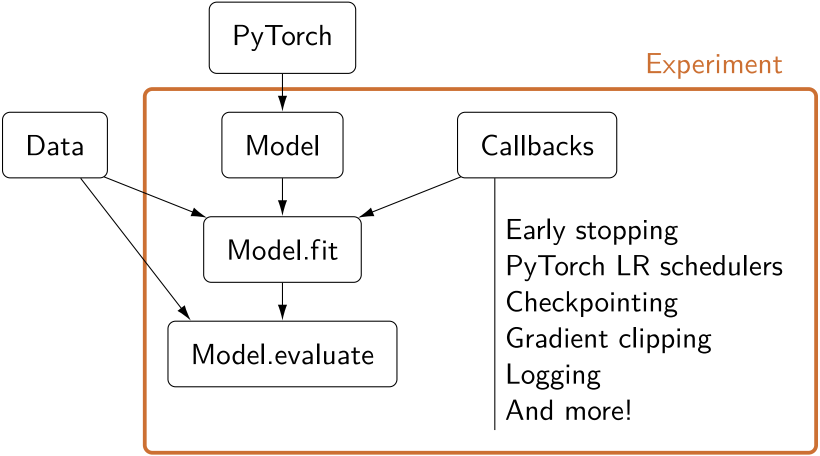
# What Is Poutyne?

- Framework for training neural networks with PyTorch
- Includes checkpointing and logging mechanisms
- Allows to setup experiments quickly

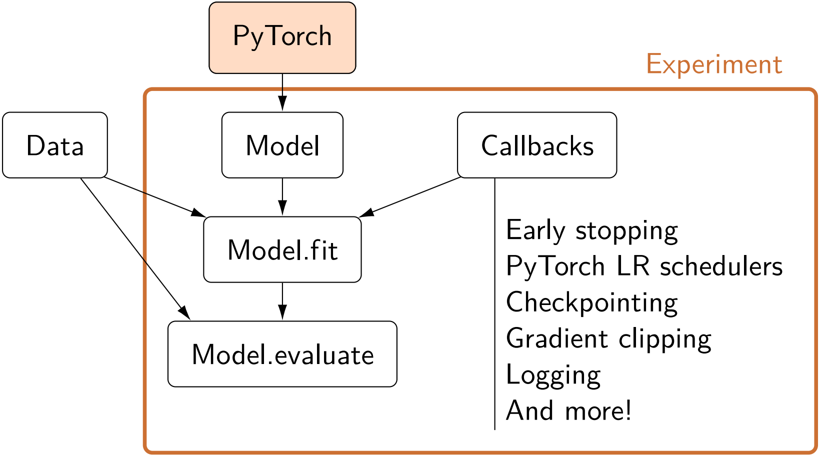
# Core Principles

- Easy to use for simple use cases
- Flexible enough for more complex use cases
- Callbacks are your friends

# Poutyne Flow



# Poutyne Flow



# PyTorch

- Automatic differentiation Python library
- For every differentiable operation done in the “forward” pass, backpropagation is done in the “backward” pass.

```
from torch import nn

class MnistLogistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(784, 10) /
                                    math.sqrt(784)) # W
        self.bias = nn.Parameter(torch.zeros(10)) # b

    def forward(self, xb):
        return xb.matmul(self.weights) + self.bias # xW + b
```

# Usual Code for Training With PyTorch

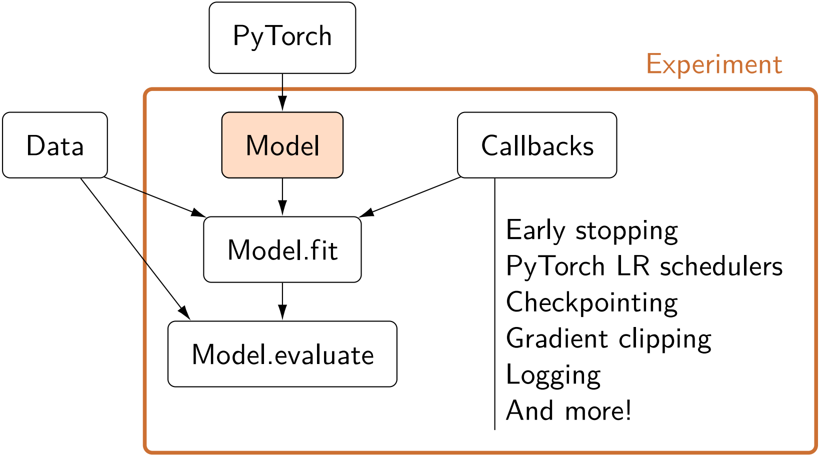
```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

net = nn.Sequential(
    nn.Linear(100, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
)

num_features = 100
num_classes = 10

train_dataset = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32)

test_dataset = TensorDataset(x_test, y_test)
test_loader = DataLoader(test_dataset, batch_size=32)

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

```python
for epoch in range(5):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        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


correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correct / total))
```

# Equivalent Code for Training With Poutyne

```python
import torch.nn as nn
import torch.optim as optim
from poutyne import Model

net = nn.Sequential(
    nn.Linear(100, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
)

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

model = Model(net, optimizer, criterion, batch_metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5,  batch_size=32)
loss, accuracy = model.evaluate(x_test, y_test, batch_size=32)
```

# Poutyne Flow

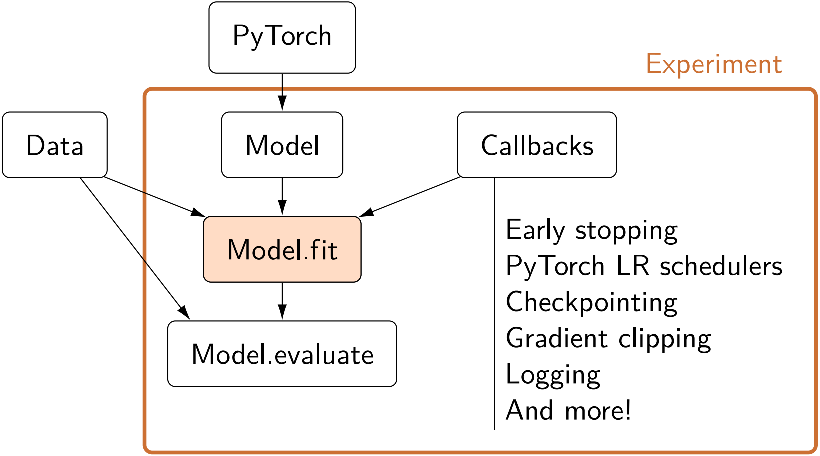


# Model Class

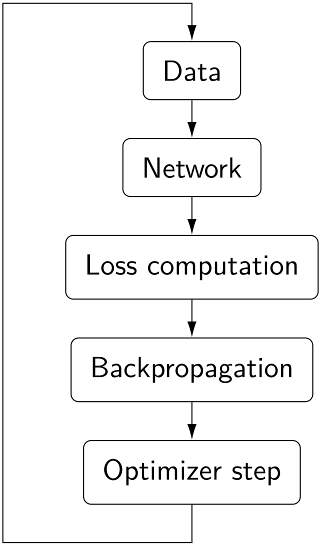
- Main class of the framework
- Plays well with Numpy
- No restrictions on the input or the output format of the network
- Manages devices (GPUs)

```
from poutyne import Model

model = Model(network, optimizer, loss_function)
model.to(device)

model.fit_generator(train_loader, valid_loader,
                    epochs=num_epochs, callbacks=callbacks)

test_loss = model.evaluate_generator(test_loader)
```

# Poutyne Flow



# Poutyne Training Flow



# Poutyne Training Flow

**for** $n$ epochs **do**

**for** each drawn batch $(x, y)$ in training dataset **do**

$$\hat{y} = f(x; \theta)$$
$$\ell = \mathcal{L}(\hat{y}, y)$$
$$g = \nabla_\theta \ell$$

Update $\theta$ with $g$ using chosen optimizer.
Compute and accumulate metrics with $\hat{y}$ and $y$.

**end for**

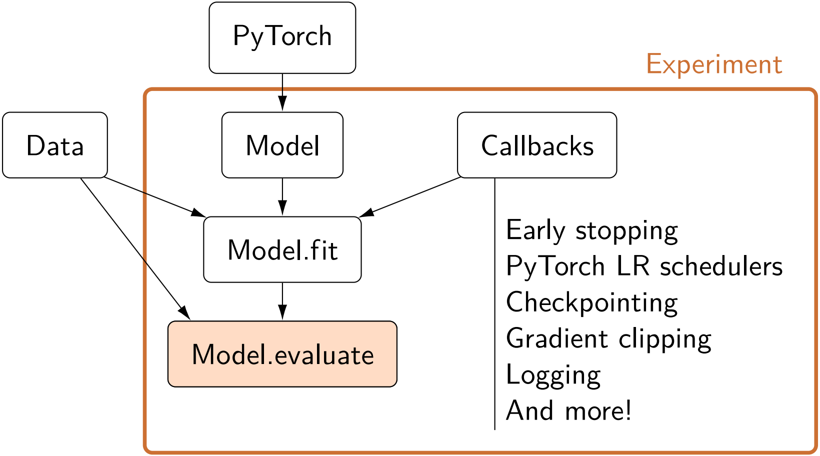
Compute loss and metrics on validation dataset.

**end for**

# Poutyne Training Flow

**for** $n$ epochs **do**
    **Callback on epoch begin**
    **for** each drawn batch $(x, y)$ in training dataset **do**
        **Callback on batch begin**
        $\hat{y} = f(x; \theta)$
        $\ell = \mathcal{L}(\hat{y}, y)$
        $g = \nabla_\theta \ell$
        **Callback on backward end**
        Update $\theta$ with $g$ using chosen optimizer.
        Compute and accumulate metrics with $\hat{y}$ and $y$.
        **Callback on batch end**
    **end for**
    Compute loss and metrics on validation dataset.
    **Callback on epoch end**
**end for**

# Poutyne Flow



# Evaluation Metrics

Using metrics with Poutyne allows easy early stopping, checkpointing and logging.

# Evaluation Metrics

Using metrics with Poutyne allows easy early stopping, checkpointing and logging.

- Batch metrics
  - Decomposable metrics (e.g. accuracy, mse, etc.)
  - Any PyTorch loss function

# Evaluation Metrics

Using metrics with Poutyne allows easy early stopping, checkpointing and logging.

- Batch metrics
  - Decomposable metrics (e.g. accuracy, mse, etc.)
  - Any PyTorch loss function
- Epoch metrics
  - Non-decomposable metrics (e.g. F1, AUC ROC, etc.)
  - Provide a scikit-learn wrapper

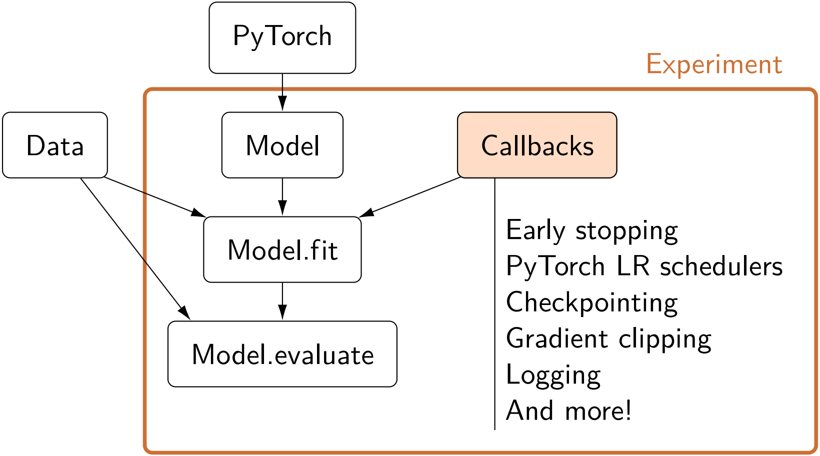
# Evaluation Metrics

Using metrics with Poutyne allows easy early stopping, checkpointing and logging.

- Batch metrics
  - Decomposable metrics (e.g. accuracy, mse, etc.)
  - Any PyTorch loss function
- Epoch metrics
  - Non-decomposable metrics (e.g. F1, AUC ROC, etc.)
  - Provide a scikit-learn wrapper

```
from poutyne import Model

model = Model(network, 'sgd', 'cross_entropy',
                batch_metrics=['accuracy'], epoch_metrics=['f1'])
model.to(device)

model.fit_generator(train_loader, valid_loader,
                    epochs=num_epochs, callbacks=callbacks)

test_loss, (test_acc, test_f1) = model.evaluate_generator(test_loader)
print(f'Test: Loss: {test_loss}, Accuracy: {test_acc}, F1: {test_f1}')
```

# Poutyne Flow



# Callbacks

```python
class Callback:
    def on_train_begin(self, logs: dict): ...
    def on_train_end(self, logs: dict): ...

    def on_epoch_begin(self, epoch_number: int, logs: dict): ...
    def on_epoch_end(self, epoch_number: int, logs: dict): ...

    def on_train_batch_begin(self, batch_number: int, logs: dict): ...
    def on_train_batch_end(self, batch_number: int, logs: dict): ...

    def on_backward_end(self, batch_number: int): ...

    def on_test_batch_begin(self, batch_number: int, logs: dict): ...
    def on_test_batch_end(self, batch_number: int, logs: dict): ...

    def on_test_begin(self, logs: dict): ...
    def on_test_end(self, logs: dict): ...

    self.params = {...} # Contains 'epochs' and 'steps_per_epoch'
    self.model = ... # Poutyne Model
```

# Callbacks

```python
from poutyne import Model, ModelCheckpoint, CSVLogger

callbacks = [
    ModelCheckpoint('last_epoch.ckpt'),
    ModelCheckpoint('best_epoch.ckpt', save_best_only=True,
                    monitor='val_acc', mode='max'),
    CSVLogger('log.csv'),
]

model = Model(network, 'sgd', 'cross_entropy',
              batch_metrics=['accuracy'], epoch_metrics=['f1'])
model.to(device)

model.fit_generator(train_loader, valid_loader,
                    epochs=num_epochs, callbacks=callbacks)

test_loss, (test_acc, test_f1) = model.evaluate_generator(test_loader)
print(f'Test: Loss: {test_loss}, Accuracy: {test_acc}, F1: {test_f1}')
```

# Checkpointing

- ModelCheckpoint
- OptimizerCheckpoint
- LRSchedulerCheckpoint

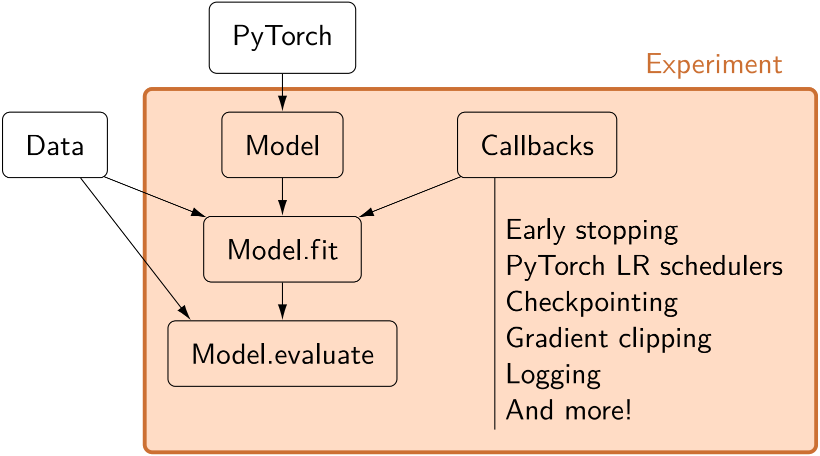
# Early Stopping and LR Scheduling

- EarlyStopping
- Any PyTorch LR scheduler
- FastAI-like learning rate policies

# Playing With Gradient

- Use PyTorch's respective functions
  - ClipNorm
  - ClipGrad
- TensorBoardGradientTracker

# Poutyne Flow



# Experiment Class

- Allows stopping and resuming optimization any time
- Saves and logs everything in a single directory
- Integration with Tensorboard

```
from poutyne import Experiment

# Instead of `task`, you can provide your own loss function and metrics.
expt = Experiment('my_directory', network,
                  task='classifier', optimizer='sgd')
expt.train(train_loader, valid_loader,
           epochs=epochs,
           callbacks=callbacks,
           seed=42)
expt.test(test_loader)
```

# Experiment Class

- Saves the last checkpoint and every "best" checkpoint (ModelCheckpoint).
- Saves the last states of the optimizer and LR schedulers (OptimizerCheckpoint, LRSchedulerCheckpoint).
- Logs training and validation loss and metrics into CSV and Tensorboard (CSVLogger, TensorBoardLogger).

# Related Works

**PyTorch Lightning:**

- Flexible
- Couples network with training
  - Requires inherting from a special class
  - Everything training related is inside the network
- Add boilerplate where it should not (e.g. adding LR schedulers seems awkward)

# Related Works

**PyTorch Lightning:**

- Flexible
- Couples network with training
  - Requires inherting from a special class
  - Everything training related is inside the network
- Add boilerplate where it should not (e.g. adding LR schedulers seems awkward)

**FastAI:**

- Lots of features
- API not as intuitive

# Related Works

**PyTorch Lightning:**

- Flexible
- Couples network with training
  - Requires inherting from a special class
  - Everything training related is inside the network
- Add boilerplate where it should not (e.g. adding LR schedulers seems awkward)

**FastAI:**

- Lots of features
- API not as intuitive

**AllenNLP:**

- Specialized for NLP
- Experiment framework
- Trainer not flexible enough (everything in `__init__`)

# Demo Time

Fine-tuning with dataset:
`http://www.vision.caltech.edu/visipedia/CUB-200.html`

# Future Works

- Add tqdm and colors for progression.
- Add tutorial pages to website.
- Integrate multi-GPU.
- Add a simpler way to add regularizer to the loss function.
- Add utilities to simplify parameters initialization.
- Integrate an experiment library such as MLFlow.

# Obtain Poutyne

Install via pip

`pip install poutyne`

Documentation and examples available!

`https://poutyne.org`

GRAAL-Research/poutyne

# Poutyne in the PyTorch Ecosystem

https://pytorch.org/ecosystem/

PyTorch

Get Started | Ecosystem | Mobile | Blog | Tutorials | Docs | Resources | GitHub

**ParlAI**

ParlAI is a unified platform for sharing, training, and evaluating dialog models across many tasks.

**PennyLane**

PennyLane is a library for quantum ML, automatic differentiation, and optimization of hybrid quantum-classical computations.

**Poutyne**

Poutyne is a Keras-like framework for PyTorch and handles much of the boilerplating code needed to train neural networks.

**Pyro**

Pyro is a universal probabilistic programming language (PPL) written in Python and supported by PyTorch on the backend.

**PySyft**

PySyft is a Python library for encrypted, privacy preserving deep learning.

**PyTorch Geometric**

PyTorch Geometric is a library for deep learning on irregular input data such as graphs, point clouds, and manifolds.

The end.

Questions?