rotor tutorial

Lionel Eyraud-Dubois, Olivier Beaumont, Alena Shilova, Rémi Duclos

TOPAL Working Group

June 17, 2021

Presentation of **rotor**

Objectives

- Limit the memory used while training Pytorch models
- Drop some intermediate results, recompute them when needed
- **Optimal** selection of results to drop and when to recompute
- Transparent usage

Available as a Python library

`https://gitlab.inria.fr/hiepacs/rotor`

Contents

Recap of normal Pytorch usage

Simple usage of **rotor**

How it works

Advanced usage

Create a model

A model is a subclass of `torch.nn.Module`. You just need to implement a `forward()` function which describes the computation done in the model.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MyModel(nn.Module):
    def __init__(self, hidden1=100, hidden2=100):
        super().__init__()
        self.hidden1 = nn.Linear(784, hidden1)
        self.hidden2 = nn.Linear(hidden1, hidden2)
        self.hidden3 = nn.Linear(hidden2, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = self.hidden1(x)
        x = F.relu(x)
        x = self.hidden2(x)
        x = F.relu(x)
        x = self.hidden3(x)
        x = F.softmax(x, dim=0)
        return x
```

Simpler implementation: the Sequential container

```
def myModel(hidden1=100, hidden2=100):
    list = [
        nn.Flatten(),
        nn.Linear(784, hidden1),
        nn.ReLU(),
        nn.Linear(hidden1, hidden2),
        nn.ReLU(),
        nn.Linear(hidden2, 10),
        nn.Softmax(dim=0)
    ]
    return nn.Sequential(list)
```

Or alternatively:

```
class MyModel(nn.Sequential):
    def __init__(self, hidden1=100, hidden2=100):
        super().__init__()
        self.add_module("flatten", nn.Flatten())
        self.add_module("hidden1", nn.Linear(784, hidden1))
        self.add_module("relu1", nn.ReLU())
        self.add_module("hidden2", nn.Linear(hidden1, hidden2))
        self.add_module("relu2", nn.ReLU())
        self.add_module("hidden3", nn.Linear(hidden2, 10))
        self.add_module("softmax", nn.Softmax(dim=0))
```

Read the dataset

```
from torchvision import datasets
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader

data = datasets.MNIST(root="data", train=True, download=True, transform=ToTensor())
loader = DataLoader(training_data, batch_size=64)
```

Prepare the model and optimization setting

```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model = MyModel().to(device)
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
epochs = 10
```

Training loop

For all batches in the dataset

- send data to the GPU
- compute the prediction with the model
- compute the loss by comparing with the target
- use `backward` to produce all gradients
- use the optimizer to update the weights given the gradients
- *(optional)* test the current model on the test dataset after each epoch

```
for epoch in range(epochs):
    for (input, target) in loader:
        input, target = input.to(device), target.to(device)
        pred = model(input)
        loss_value = loss(pred, target)
        optimizer.zero_grad()
        loss_value.backward()
        optimizer.step()
```

Simple usage of **rotor**

- Just replace your pytorch `model` by `rotor.Checkpointable(model)`.
- The rest of the training process is unchanged.

```
import rotor

model = myModel().to(device)
model = rotor.Checkpointable(model)
```

- **rotor** automatically limits the memory usage of your model to what is available on the CUDA device when it is first executed.
- Specify a memory limit (eg 10GB) with `Checkpointable(model, mem_limit=10*2**30)`
- As of now, this limit only includes the memory used by the activations.

Important limitation

- The model given to **rotor** needs to be a `torch.nn.Sequential` model.
- This allows **rotor** to know which computations happen in the `forward` function of the user model.
- Not possible to directly use the models from `torchvision` in **rotor**.

Adapted implementations

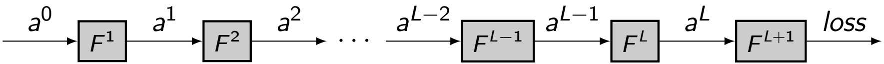
rotor contains adapted (equivalent) implementations of the `torchvision` models

```
model = rotor.models.resnet101().to(device)
model = rotor.Checkpointable(model)
```

In most cases, making an implementation based on `Sequential` is not difficult. We will discuss it in more details later.

How it works: dependency graph

- Forward computation: a list of layers F_i, the input of F_{i+1} is the output of F_i.



Computing a^2 from a^1:

```
with torch.no_grad():
    a2 = F2(a1)
```

How it works: dependency graph

- Forward computation: a list of layers F_i, the input of F_{i+1} is the output of F_i.
- Backward computation in reverse: B_i requires the output of F_i and F_{i-1}.

$$\xrightarrow{a^0} \boxed{F^1} \xrightarrow{a^1} \boxed{F^2} \xrightarrow{a^2} \cdots \xrightarrow{a^{L-2}} \boxed{F^{L-1}} \xrightarrow{a^{L-1}} \boxed{F^L} \xrightarrow{a^L} \boxed{F^{L+1}} \xrightarrow{loss}$$

Computing a^2 from a^1:

```
with torch.no_grad():
    a2 = F2(a1)
```

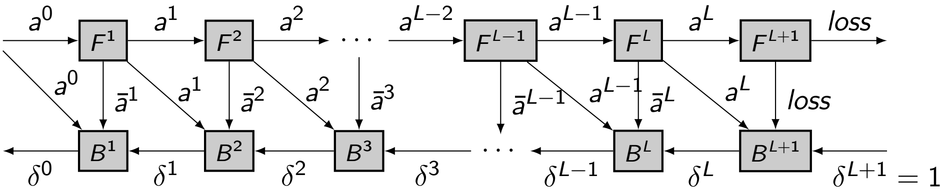
Computing B^2:

```
with torch.enable_grad():
    a2 = F2(a1)

a2.backward(delta2)
delta1 = a1.grad
```

How it works: dependency graph

- Forward computation: a list of layers F_i, the input of F_{i+1} is the output of F_i.
- Backward computation in reverse: B_i requires the output of F_i and F_{i-1}.



Computing a^2 from a^1:

```
with torch.no_grad():
    a2 = F2(a1)
```

Computing B^2:

```
with torch.enable_grad():
    a2 = F2(a1)

a2.backward(delta2)
delta1 = a1.grad
```

First step: Measuring

Before executing the model, **rotor** measures all layers F_i, using the first batch. Values measured are:

- execution time of forward and backward
- memory usage of the outputs (a^i and $\bar{a}^i$)
- memory peak during the forward and backward (usage of temporary data)

This can be triggered independently with

```
model.measure(sample_input)
```

Second step: Optimization

rotor describes the computation by a `Sequence` of operations, among:

- `Fng(i)` computes the output of F_i, and forgets the input. Equivalent to:

```
with torch.no_grad():
    x = F[i](x)
```

- `Fck(i)` computes the output of F_i, and keeps the input. Equivalent to:

```
with torch.no_grad():
    y = F[i](x)
```

- `Fe(i)` computes the output of F_i, enabling gradient computation. Equivalent to:

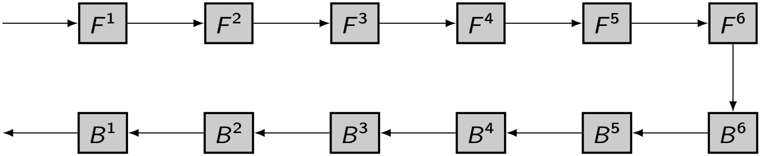
```
with torch.enable_grad():
    y = F[i](x)
```

- `B(i)` computes the backward of layer i. Equivalent to:

```
y.backward(g)
g = x.grad
```

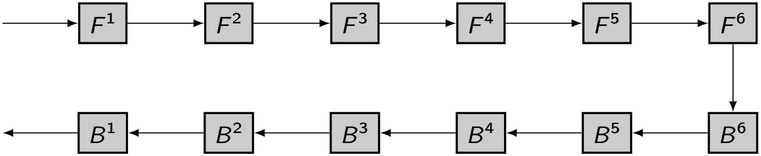
Second step: Optimization

- Dynamic Programming used to compute an optimal sequence
- Optimal: **minimal overhead given a memory limit**



Second step: Optimization

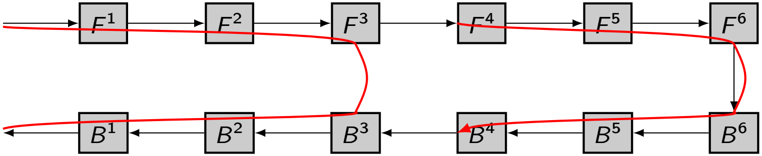
- Dynamic Programming used to compute an optimal sequence
- Optimal: **minimal overhead given a memory limit**



Divide and Conquer: half the memory for each half of the model

Second step: Optimization

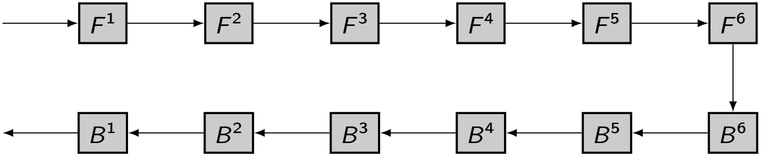
- Dynamic Programming used to compute an optimal sequence
- Optimal: **minimal overhead given a memory limit**



Divide and Conquer: half the memory for each half of the model
Wasteful: the backward of the first half could use all the memory!

Second step: Optimization

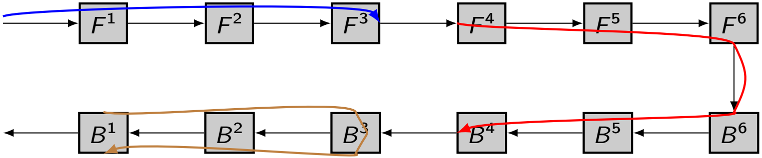
- Dynamic Programming used to compute an optimal sequence
- Optimal: **minimal overhead given a memory limit**



Dynamic Programming: recursive computation of optimal sequence $\mathrm{opt}(i, j)$ constrained to storing the input of F^i

Second step: Optimization

- Dynamic Programming used to compute an optimal sequence
- Optimal: **minimal overhead given a memory limit**

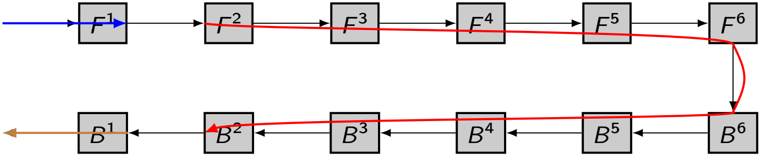


Dynamic Programming: recursive computation of optimal sequence $\mathrm{opt}(i, j)$ constrained to storing the input of F^i

- If we decide to store the input of k:
  $\mathrm{Fck}(i)\ \mathrm{Fng}(i+1)\ \ldots\ \mathrm{Fng}(k-1)\ \mathrm{opt}(k, j)\ \mathrm{opt}(i, k-1)$

Second step: Optimization

- Dynamic Programming used to compute an optimal sequence
- Optimal: **minimal overhead given a memory limit**

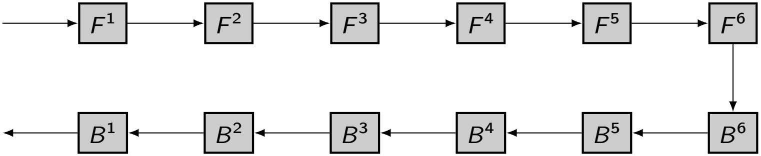


Dynamic Programming: recursive computation of optimal sequence $\mathrm{opt}(i, j)$ constrained to storing the input of F^i

- If we decide to store the input of k:
  $\mathrm{Fck}(i)\ \mathrm{Fng}(i+1)\ \ldots\ \mathrm{Fng}(k-1)\ \mathrm{opt}(k, j)\ \mathrm{opt}(i, k-1)$
- If we decide not to recompute F^i:
  $\mathrm{Fe}(i)\ \mathrm{opt}(i+1, j)\ \mathrm{B}(i)$

Second step: Optimization

- Dynamic Programming used to compute an optimal sequence
- Optimal: **minimal overhead given a memory limit**



Dynamic Programming: recursive computation of optimal sequence $\mathrm{opt}(i, j)$ constrained to storing the input of F^i

- If we decide to store the input of k:
  $\mathrm{Fck}(i)\ \mathrm{Fng}(i+1)\ \ldots\ \mathrm{Fng}(k-1)\ \mathrm{opt}(k, j)\ \mathrm{opt}(i, k-1)$
- If we decide not to recompute F^i:
  $\mathrm{Fe}(i)\ \mathrm{opt}(i+1, j)\ \mathrm{B}(i)$

Can be triggered with `model.compute_sequence(mem_limit)`

Third step: Execution

Internally, **rotor** defines a custom Pytorch `Function`, which provides specific `forward` and `backward` methods. **rotor** calls the `forward` method for each application of the model on a Tensor (if the model is in training mode), with the sequence computed as above. The `backward` method is then automatically called by Pytorch's autograd mechanism.

From the user's perspective, all this is transparent. It is enough to perform the usual call:

```
pred = model(input)
pred.backward(input_gradient)
```

Sequentialization

- **rotor** requires a `Sequential` model as an input
- For most Deep Learning models, this is conceptually not a constraint, but it may require in practice to change the implementation
- Example: the `forward` function of the ResNet model from `torchvision`:

```
def forward(self, x: Tensor) -> Tensor:
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)

    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)

    return x
```

- Can be very easily converted to a `Sequential` implementation
- Available in `rotor.models`

Recursive Sequential containers

- Layers which are themselves `Sequential` are explored recursively
- From the point of view of **rotor**, the two following models are equivalent

```
model = nn.Sequential([
    nn.Flatten(),
    nn.Linear(784, hidden1),
    nn.ReLU(),
    nn.Linear(hidden1, hidden2),
    nn.ReLU(),
    nn.Linear(hidden2, 10),
    nn.Softmax(dim=0)
  ])
```

```
def linear_and_relu(dim1, dim2):
    return nn.Sequential(nn.Linear(dim1, dim2), nn.ReLU())

model = nn.Sequential([
    nn.Flatten(),
    linear_and_relu(784, hidden1),
    linear_and_relu(hidden1, hidden2),
    nn.Linear(hidden2, 10),
    nn.Softmax(dim=0)
  ])
```

- More flexibility in the implementation (here, code re-use)
- In the ResNet example, `self.layer1` to `self.layer4` are actually `Sequential`

In-place operations

- Some in-place operations allowed by Pytorch (`ReLU` for example)
- Very beneficial in terms of memory

- In **rotor**, the first computation in any layer F_i can **not** be in-place
- In-place operations need to be fused with the previous operation
- **rotor** provides a `rotor.models.utils.ReLUAtEnd` to help using in-place `ReLU`

In-place operations

- Some in-place operations allowed by Pytorch (`ReLU` for example)
- Very beneficial in terms of memory

- In **rotor**, the first computation in any layer F_i can **not** be in-place
- In-place operations need to be fused with the previous operation
- **rotor** provides a `rotor.models.utils.ReLUAtEnd` to help using in-place `ReLU`

```
model = nn.Sequential([
    nn.Flatten(),
    nn.Linear(784, hidden1),
    nn.ReLU(inplace=True),
    nn.Linear(hidden1, hidden2),
    nn.ReLU(inplace=True),
    nn.Linear(hidden2, 10),
    nn.Softmax(dim=0)
  ])
```

In-place operations

- Some in-place operations allowed by Pytorch (`ReLU` for example)
- Very beneficial in terms of memory

- In **rotor**, the first computation in any layer F_i can **not** be in-place
- In-place operations need to be fused with the previous operation
- **rotor** provides a `rotor.models.utils.ReLUAtEnd` to help using in-place `ReLU`

```
from rotor.models.utils import ReLUAtEnd

model = nn.Sequential([
    nn.Flatten(),
    ReLUAtEnd(nn.Linear(784, hidden1)),
    ReLUAtEnd(nn.Linear(hidden1, hidden2)),
    nn.Linear(hidden2, 10),
    nn.Softmax(dim=0)
  ])
```

That's all folks!

Thank you for your attention

Questions?