EE-559 – Deep learning

10.4. Model persistence and checkpoints

François Fleuret

https://fleuret.org/ee559/

Wed Mar 6 06:07:21 UTC 2019

Saving and loading models is key to use models trained previously.

It also allows to implement **checkpoints** which keep track of the state during training and allow to either restart after an expected interruption, or modulate meta-parameters manually.

Saving and loading models is key to use models trained previously.

It also allows to implement **checkpoints** which keep track of the state during training and allow to either restart after an expected interruption, or modulate meta-parameters manually.

The underlying operation is **serialization**, that is the transcription of an arbitrary object into a sequence of bytes saved on disk.

The main PyTorch methods for serializing are `torch.save(obj, filename)` and `torch.load(filename)`.

```
>>> x = 34
>>> torch.save(x, 'x.pth')
>>> y = torch.load('x.pth')
>>> y
34
```

The main PyTorch methods for serializing are `torch.save(obj, filename)` and `torch.load(filename)`.

```
>>> x = 34
>>> torch.save(x, 'x.pth')
>>> y = torch.load('x.pth')
>>> y
34

>>> z = { 'a': torch.LongTensor(2, 3).random_(10), 'b': nn.Linear(10, 20) }
>>> torch.save(z, 'z.pth')
>>> w = torch.load('z.pth')
>>> w
{'a': tensor([[4, 2, 9],
        [7, 2, 7]]), 'b': Linear(in_features=10, out_features=20, bias=True)}
```

One can save directly a full model like this, including arbitrary fields

```
>>> x = nn.Sequential(nn.Linear(3, 10), nn.ReLU(), nn.Linear(10, 1))
>>> x.blah = 14
>>> torch.save(x, 'model.pth')
>>>
>>> z = torch.load('model.pth')
>>> z(torch.empty(2, 3).normal_())
tensor([[ 0.0665],
        [ 0.2116]])
>>> z.blah
14
```

Saving a full model with `torch.save()` bounds the saved quantities to the specific class implementation, and may break after changes in the code.

Saving a full model with `torch.save()` bounds the saved quantities to the specific class implementation, and may break after changes in the code.

The suggested policy is to save the **state dictionary** alone, as provided by `Module.state_dict()`, which encompasses `Parameters` and **buffers** such as batchnorm running estimates, etc.

Saving a full model with `torch.save()` bounds the saved quantities to the specific class implementation, and may break after changes in the code.

The suggested policy is to save the **state dictionary** alone, as provided by `Module.state_dict()`, which encompasses `Parameters` and **buffers** such as batchnorm running estimates, etc.

Additionally

- Tensors are saved with their locations (CPU, or GPU), and will be loaded in the same configuration,
- in your `Module`s, buffers have to be identified with `register_buffer`,
- loaded models are in train mode by default,
- optimizers have a state too (momentum, Adam).

A checkpoint is a persistent object that keeps the global state of the training: model and optimizer.

A checkpoint is a persistent object that keeps the global state of the training: model and optimizer. In the following example (1) we load it when we start if it exists, and (2) we save it at every epoch.

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

checkpoint_name = 'checkpoint.pth'

try:
    checkpoint = torch.load(checkpoint_name)
    nb_epochs_finished = checkpoint['nb_epochs_finished']
    model.load_state_dict(checkpoint['model_state'])
    optimizer.load_state_dict(checkpoint['optimizer_state'])
    print('Checkpoint loaded with %d epochs finished.' % nb_epochs_finished)

except FileNotFoundError:
    print('Starting from scratch.')

except:
    print('Error when loading the checkpoint.')
    exit(1)
```

```
for k in range(nb_epochs_finished, nb_epochs):
    acc_loss = 0

    for input, targets in zip(train_input.split(batch_size),
                              train_targets.split(batch_size)):
        output = model(input)
        loss = criterion(output, targets)
        acc_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(k, acc_loss)

    checkpoint = {
        'nb_epochs_finished': k + 1,
        'model_state': model.state_dict(),
        'optimizer_state': optimizer.state_dict()
    }
    torch.save(checkpoint, checkpoint_name)
```

If we `killall python` during training

```
fleuret@elk:/tmp/ ./tinywithcheckpoint.py
Starting from scratch.
0 161.2404215920251
1 35.50377965264488
2 24.43254833246465
3 18.57419647696952
4 14.582882737944601
Killed
```

If we `killall python` during training

```
fleuret@elk:/tmp/ ./tinywithcheckpoint.py
Starting from scratch.
0 161.2404215920251
1 35.50377965264488
2 24.43254833246465
3 18.57419647696952
4 14.582882737944601
Killed
```

and re-start

```
fleuret@elk:/tmp/ ./tinywithcheckpoint.py
Checkpoint loaded with 5 epochs finished.
5 11.396404800716482
6 8.944935847055604
7 7.116929043420896
8 5.463898817846712
9 4.41012461569494
test_error 1.01% (101/10000)
```

Since a model is saved with information about the CPU/GPUs where each `Storage` is located there may be issues if the model is loaded on a different hardware configuration.

For instance, if we save a model located on a GPU:

```
>>> x = torch.nn.Linear(10, 4)
>>> x.to('cuda')
Linear(in_features=10, out_features=4, bias=True)
>>> torch.save(x, 'x.pth')
```

For instance, if we save a model located on a GPU:

```
>>> x = torch.nn.Linear(10, 4)
>>> x.to('cuda')
Linear(in_features=10, out_features=4, bias=True)
>>> torch.save(x, 'x.pth')
```

And load it on a machine without GPU:

```
>>> x = torch.load('x.pth')
Traceback (most recent call last):
/.../
RuntimeError: cuda runtime error (35) : CUDA driver version is insufficient for
CUDA runtime version at torch/csrc/cuda/Module.cpp:51
```

For instance, if we save a model located on a GPU:

```
>>> x = torch.nn.Linear(10, 4)
>>> x.to('cuda')
Linear(in_features=10, out_features=4, bias=True)
>>> torch.save(x, 'x.pth')
```

And load it on a machine without GPU:

```
>>> x = torch.load('x.pth')
Traceback (most recent call last):
/.../
RuntimeError: cuda runtime error (35) : CUDA driver version is insufficient for
CUDA runtime version at torch/csrc/cuda/Module.cpp:51
```

This can be fixed by specifying at load time how to relocate storages:

```
>>> x = torch.load('x.pth', map_location = lambda storage, loc: storage)
```

The end