EE-559 – Deep learning

4.3. PyTorch modules and batch processing

François Fleuret
https://fleuret.org/ee559/
Dec 23, 2019

Elements from `torch.nn.functional` are autograd-compliant functions which compute a result from provided arguments alone.

It is usually imported as `F`, and `torch.nn` as `nn`.

Subclasses of `nn.Module` are losses and network components. The latter embed parameters to be optimized during training.

Parameters are of the type `nn.Parameter` which is a `Tensor` with `requires_grad` to `True`, and known to be a model parameter by various utility functions, in particular `nn.Module.parameters()`.

⚠ Functions and modules from `nn` process **batches** of inputs stored in a tensor whose first dimension indexes them, and produce a corresponding tensor with the same additional dimension.

E.g. a fully connected layer $\mathbb{R}^C \rightarrow \mathbb{R}^D$ expects as input a tensor of size $N \times C$ and computes a tensor of size $N \times D$, where N is the number of samples and can vary from a call to another.


```
F.relu(input, inplace=False)
```

takes a tensor of any size as input, applies ReLU on each value to produce a result tensor of same size.

```
>>> x
tensor([[ 0.8008, -0.2586,  0.5019, -0.2002, -0.7416],
        [ 0.0557,  0.6046,  0.0864, -0.5929,  1.2606]])
>>> F.relu(x)
tensor([[ 0.8008,  0.0000,  0.5019,  0.0000,  0.0000],
        [ 0.0557,  0.6046,  0.0864,  0.0000,  1.2606]])
```

`inplace` indicates if the operation should modify the argument itself. This may be desirable to reduce the memory footprint of the processing.

The module

```
nn.Linear(in_features, out_features, bias=True)
```

implements a $\mathbb{R}^C \to \mathbb{R}^D$ fully-connected layer. It takes as input a tensor of size $N \times C$ and produce a tensor of size $N \times D$.

```
>>> f = nn.Linear(in_features = 10, out_features = 4)
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([4, 10])
bias torch.Size([4])
>>> x = torch.empty(523, 10).normal_()
>>> y = f(x)
>>> y.size()
torch.Size([523, 4])
```

⚠ The weights and biases are automatically randomized at creation. We will come back to that later.

The module

```
nn.MSELoss()
```

implements the Mean Square Error loss: the sum of the component-wise squared difference, **divided by the total number of components in the tensors**.

```
>>> f = nn.MSELoss()
>>> x = torch.tensor([[ 3. ]])
>>> y = torch.tensor([[ 0. ]])
>>> f(x, y)
tensor(9.)
>>> x = torch.tensor([[ 3., 0., 0., 0. ]])
>>> y = torch.tensor([[ 0., 0., 0., 0. ]])
>>> f(x, y)
tensor(2.2500)
```

The first parameter of a loss is traditionally called the **input** and the second the **target**. These two quantities may be of different dimensions or even types for some losses (*e.g.* for classification).

Criteria do not accept a tensor with `requires_grad` to `True` for target.

```
>>> import torch
>>> f = nn.MSELoss()
>>> x = torch.tensor([ 3., 2. ]).requires_grad_()
>>> y = torch.tensor([ 0., -2. ]).requires_grad_()
>>> f(x, y)
Traceback (most recent call last):
/.../
AssertionError: nn criterions don't compute the gradient w.r.t.
targets - please mark these tensors as not requiring gradients
```


Batch processing

Functions and modules from `nn` process samples by batches. This is motivated by the computational speed-up it induces.

Training a large network on CIFAR10:

Batch size	Time per epoch
1	4h22min
64	4min50s

speed up of $\times 54$.

To evaluate a module on a sample, both the module's parameters and the sample have to be first copied into **cache memory**, which is fast but small.

For any model of reasonable size, only a fraction of its parameters can be kept in cache, so a module's parameters have to be copied there every time it is used.

Memory transfers are slower than the computation. Batch processing cuts down to one copy of the parameters to the cache per batch.

It also cuts down the use of Python loops, which are awfully slow.

Consider a model composed of three modules

$$f = f_3 \circ f_2 \circ f_1,$$

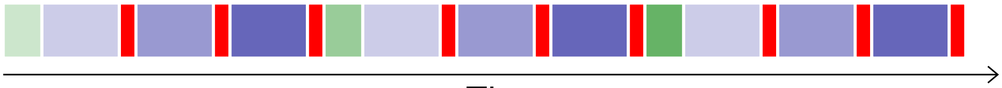
and we want to compute $f(x_1), f(x_2), f(x_3)$.

Copying the x_ns to cache memory

Copying the f_ds' parameters to cache memory

Computing a $f_d(.)$

Processing samples one by one:



Time

Batch processing:

Time

With

```
def timing(x, w, batch = False, nb = 101):
    t = torch.zeros(nb)

    for u in range(nb):
        t0 = time.perf_counter()
        if batch:
            y = x.mm(w.t())
        else:
            y = torch.empty(x.size(0), w.size(0))
            for k in range(y.size(0)): y[k] = w.mv(x[k])
        y.is_cuda and torch.cuda.synchronize()
        t[u] = time.perf_counter() - t0

    return t.median().item()
```

```
x = torch.empty(2500, 1000).normal_()
w = torch.empty(1500, 1000).normal_()
print('Batch-processing speed-up on CPU %.1f' %
      (timing(x, w, batch = False) / timing(x, w, batch = True)))

x, w = x.to('cuda'), w.to('cuda')
print('Batch-processing speed-up on GPU %.1f' %
      (timing(x, w, batch = False) / timing(x, w, batch = True)))
```

prints

```
Batch-processing speed-up on CPU 4.6
Batch-processing speed-up on GPU 144.4
```


Formally, we have to revisit a bit some expressions we saw previously for fully connected layers. We had

$$\forall l, n,\ w^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}},\ x_n^{(l-1)} \in \mathbb{R}^{d_{l-1}},\quad s_n^{(l)} = w^{(l)}\, x_n^{(l-1)}.$$

From now on, we will use row vectors, so that we can represent a series of samples as a 2d array with the first index being the sample's index.

$$x = \begin{pmatrix} x_{1,1} & \dots & x_{1,D} \\ \vdots & \ddots & \vdots \\ x_{N,1} & \dots & x_{N,D} \end{pmatrix} = \begin{pmatrix} (x_1)^T \\ \vdots \\ (x_N)^T \end{pmatrix},$$

which is an element of $\mathbb{R}^{N \times D}$.

To make all sample row vectors and apply a linear operator, we want

$$\forall n,\ s_n^{(l)} = \left(w^{(l)} \left(x_n^{(l-1)}\right)^T\right)^T = x_n^{(l-1)} \left(w^{(l)}\right)^T$$

which gives a tensorial expression for the full batch

$$s^{(l)} = x^{(l-1)} \left(w^{(l)}\right)^T.$$

And in torch/nn/functional.py

```
def linear(input, weight, bias=None):
    if input.dim() == 2 and bias is not None:
        # fused op is marginally faster
        return torch.addmm(bias, input, weight.t())

    output = input.matmul(weight.t())
    if bias is not None:
        output += bias
    return output
```


Similarly for the backward pass of a linear layer we get

$$\left[\!\left[\frac{\partial \mathscr{L}}{\partial w^{(l)}}\right]\!\right] = \left[\!\left[\frac{\partial \mathscr{L}}{\partial s^{(l)}}\right]\!\right]^T x^{(l-1)},$$

and

$$\left[\!\left[\frac{\partial \mathscr{L}}{\partial x^{(l)}}\right]\!\right] = \left[\!\left[\frac{\partial \ell}{\partial s^{(l+1)}}\right]\!\right] w^{(l+1)}.$$