EE613: Lab on CNN with PyTorch

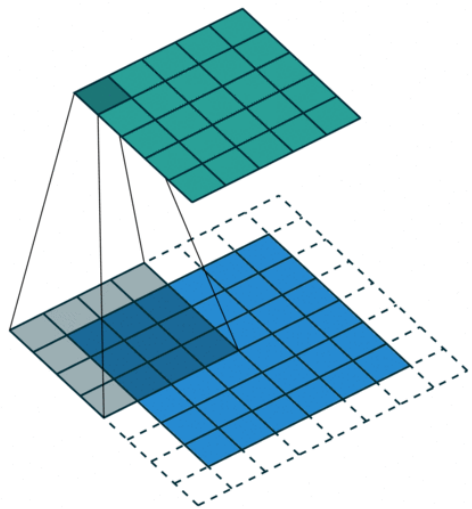
Olivier Canévet

December 22, 2023

Convolutional Neural Networks

Convolutional Neural Network

A convolution consists in computing sequentially a dot product between a signal of size n and a filter of size f. The input signal is padded on both sides with p elements. The filter is moved along the signal with a stride of s.



The convolved signal has the following number of elements:

$$\left\lfloor \frac{n + 2p - f}{s} \right\rfloor + 1$$

where $\lfloor\ \rfloor$ denote the "floor" operation: $\lfloor 2.3 \rfloor = \lfloor 2.8 \rfloor = 2$.

---

https://github.com/vdumoulin/conv_arithmetic

Activation map size in LeNet5

```
class LeNet5(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 10, 5, 1, 0)
        self.conv2 = nn.Conv2d(10, 20, 5, 1, 0)
        self.fc1 = nn.Linear(20*4*4, 100)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2, 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2, 2)
        x = x.view(-1, 4*4*self.n2)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = F.log_softmax(x, dim=1)
        return x
```

Block name	Input size	Output size
Convolution by 10 filters of size 5×5	$1 \times 28 \times 28$	$10 \times 24 \times 24$
ReLU (component wise operation)	$10 \times 24 \times 24$	$10 \times 24 \times 24$
2×2 Max pooling with a stride of 2	$10 \times 24 \times 24$	$10 \times 12 \times 12$
Convolution by 20 filters of size 5×5	$10 \times 12 \times 12$	$20 \times 8 \times 8$
ReLU (component wise operation)	$20 \times 8 \times 8$	$20 \times 8 \times 8$
2×2 Max pooling with a stride of 2	$20 \times 8 \times 8$	$20 \times 4 \times 4$

A CNN on CIFAR10

CIFAR10 is a dataset of RGB images of size 32×32. Here is a proposed network with 3 convolutional layers and 2 hidden layers for the MLP classifier:

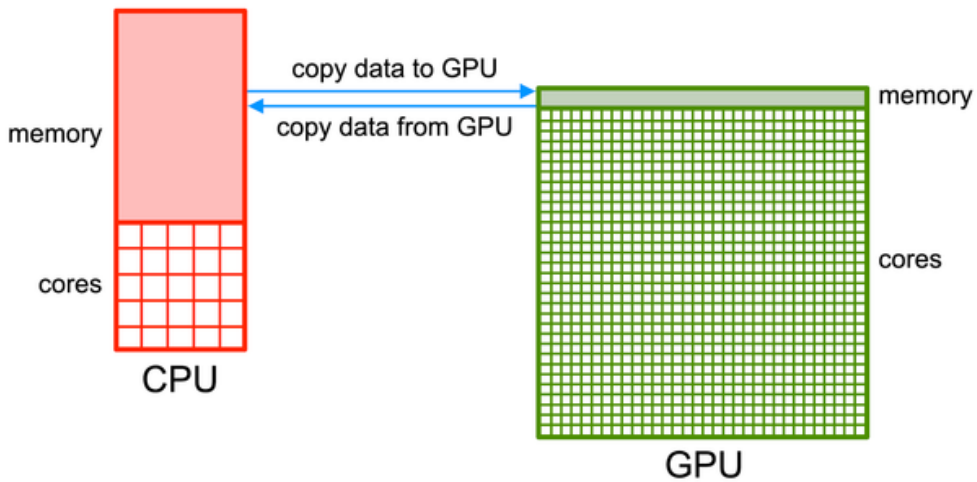
```
class DeepLeNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 5, 1, 0) # 32 -> 28
        self.conv2 = nn.Conv2d(32, 64, 3, 1, 0) # 14 -> 12
        self.conv3 = nn.Conv2d(64, 128, 3, 1, 0) # 6 -> 4
        self.fc1 = nn.Linear(2*2*128, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc3 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2, 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2, 2)
        x = F.max_pool2d(F.relu(self.conv3(x)), 2, 2)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = F.log_softmax(x, dim=1)
        return x
```

Hardware

Hardware

Graphics processing units (GPU) are very efficient at performing linear algebra operations such as matrix-matrix multiplications, 3d rendering, video encoding and decoding.

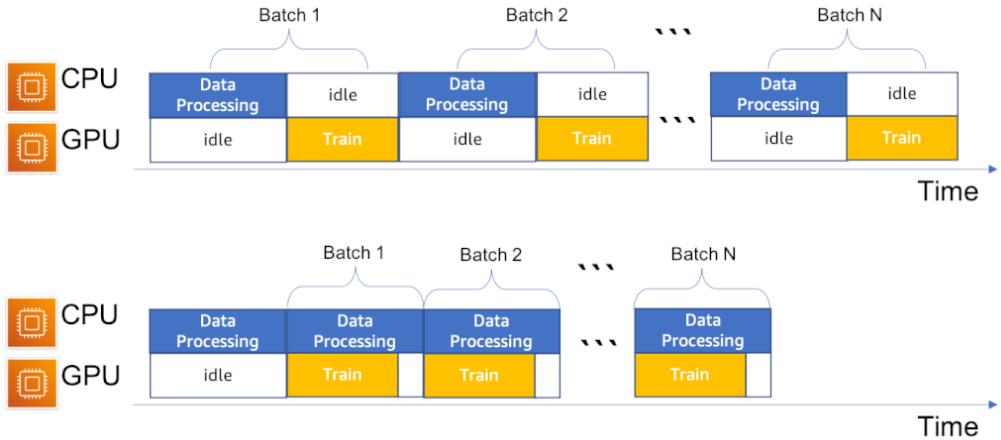
GPUs consist of a large number of computation units. The computation is distributed among all the GPU cores which makes it faster than a CPU. However, a GPU have a larger memory (RAM).



https://researchcomputing.princeton.edu/support/knowledge-base/gpu-computing

Efficient data transfert between CPU and GPU

To prevent the CPU to be idle when the GPU is processing the minibatch, and the GPU to wait for the CPU to prepare the next minibatch, frameworks make the CPU prepare minibatch $N + 1$ when the GPU is processing minibatch N:



In PyTorch, this is handled automatically with the following:

```
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=100, shuffle=True,
    num_workers=8, pin_memory=True
)
```

https://aws.amazon.com/blogs/machine-learning/best-practices-for-tensorflow-1-x-acceleration-training-on-amazon-sagemaker/

Using the GPU with PyTorch

By default, the tensors are created on the CPU.

```
x = torch.rand(10, 3, 32, 32)
```

The tensor can be moved to the (NVIDIA) GPU with:

```
x = x.to("cuda")
```

Or if you have an Apple M1 chip (with unified memory):

```
x = x.to("mps")
```

Even better is to create a PyTorch device to avoid hard coding:

```
device = torch.device(
    "cuda" if torch.cuda.is_available() else "cpu"
)
x = x.to(device)
```

The tensor can be moved back to the CPU with

```
x = x.to("cpu")
```

Models (inheriting from `nn.Module`) are moved in-place:

```
model.to(device)
```