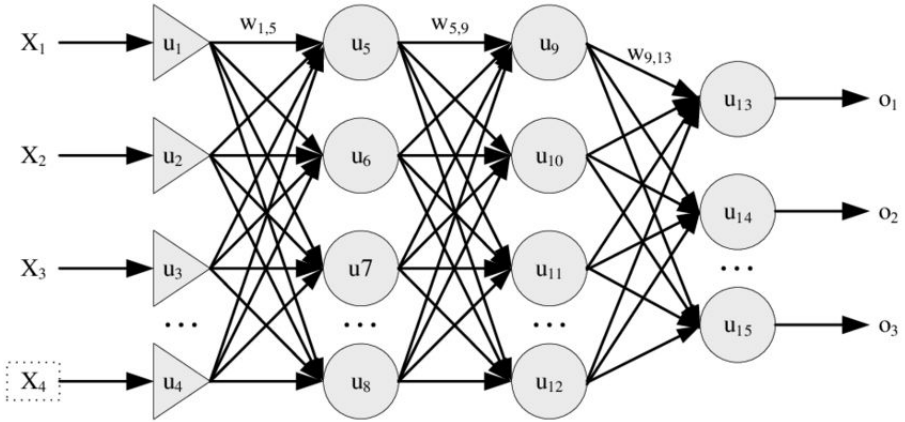
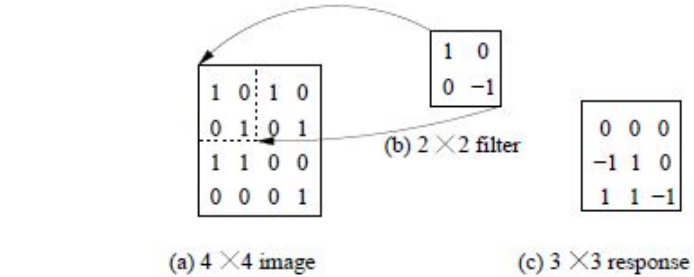
Large Scale Machine Learning With Multiple GPUs

Topics to be discussed

- Neural Network Architecture and Training
- CPU and GPU
- Multi-GPU Training - Ways to split the problem
- Data Parallelism
- Data Synchronisation
- Data Distribution
- Implementation

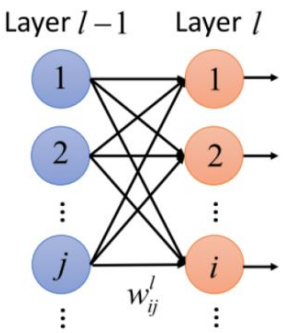
Neural Network Architecture and Training





Note that here *C* refers to the Loss function

$$\frac{\partial C^r}{\partial w_{ij}^l} = \frac{\partial z_i^l}{\partial w_{ij}^l} \frac{\partial C^r}{\partial z_i^l}$$



$$\begin{cases} a_j^{l-1} & l > 1 \\ x_j^r & l = 1 \end{cases}$$

δ_i^l

Forward Pass

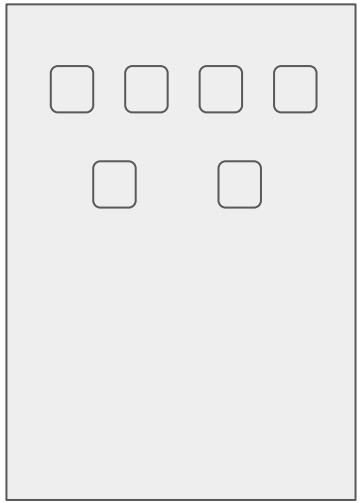
$$z^1 = W^1 x^r + b^1$$
$$a^1 = \sigma(z^1)$$
$$\cdots\cdots$$
$$z^{l-1} = W^{l-1} a^{l-2} + b^{l-1}$$
$$a^{l-1} = \sigma(z^{l-1})$$

Backward Pass

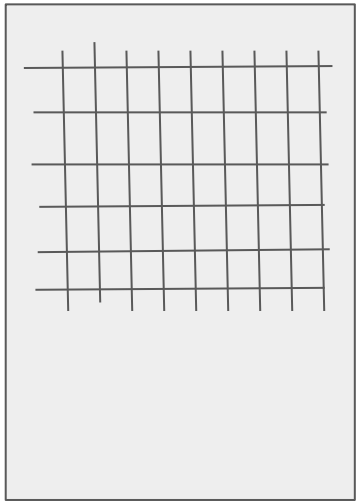
$$\delta^L = \sigma'(z^L) \bullet \nabla C^r(y^r)$$
$$\delta^{L-1} = \sigma'(z^{L-1}) \bullet (W^L)^T \delta^L$$
$$\cdots\cdots$$
$$\delta^l = \sigma'(z^l) \bullet (W^{l+1})^T \delta^{l+1}$$
$$\cdots\cdots$$

Slide Credit: Hung-yi Lee

CPU vs GPU

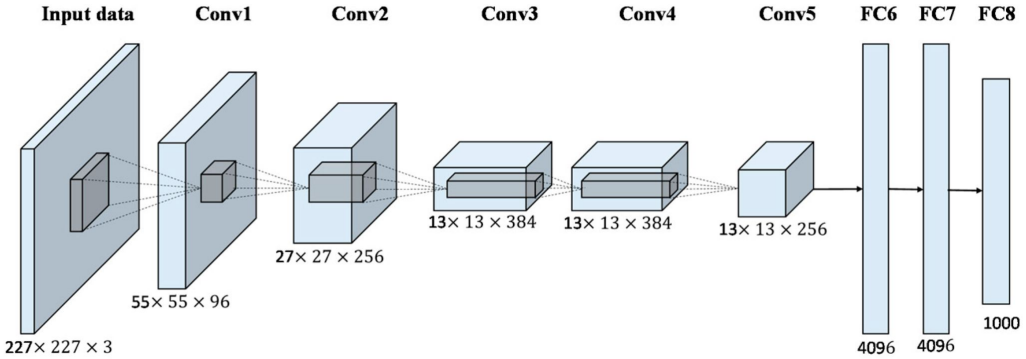


- Several cores
- Serial computation
- Low Computation power



- Thousands of cores
- Parallel computation
- High Computation power

GPU training of Neural networks



AlexNet architecture

- Parameters : 62.3 million

Number of parameters for some popular architectures:

Resnet50	23.5M
Densenet169	12.8M
VGG16	134.7M
GPT3 (m)	350M

Dataset Sizes

ImageNet - 14M images
CelebFaces - 2M images

Multi-GPU Training - Ways to split the problem

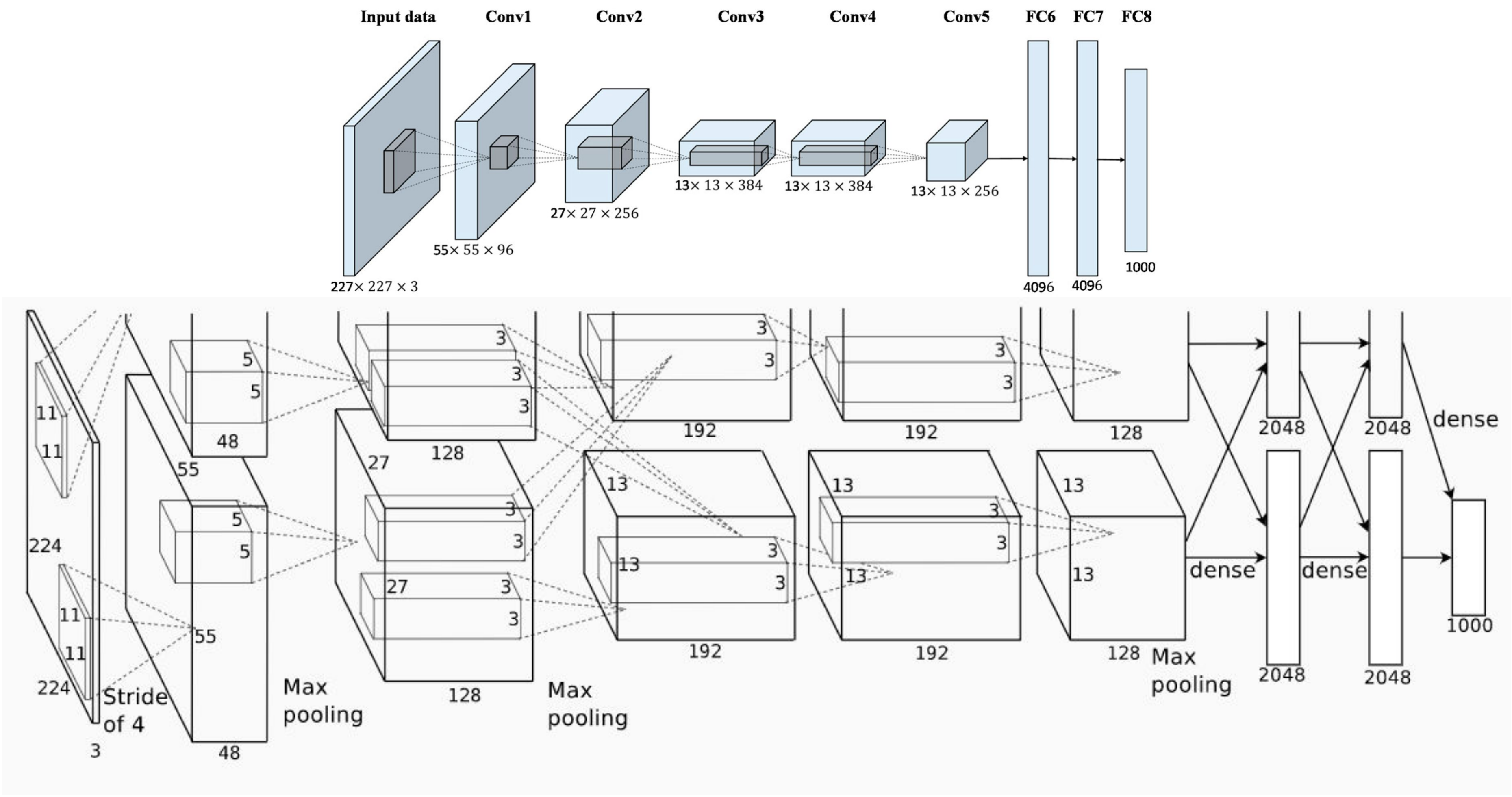
Split Network across Multiple GPUs

- GPU takes as input the data flowing into a particular layer
- Processes for a few layers and passes to next GPU
- Allows very large networks
- Memory footprint per GPU can be well controlled
- Requires tight synchronization.
- Makes the process sequential.
- Large amounts of data transfer.
- Tricky to manage when computational workloads are not properly matched between layers.

Multi-GPU Training - Ways to split the problem

Split Layers across Multiple GPUs

- Multiple GPUs corresponding to fraction of channels. (64 channel in a single GPU -> 4 GPUs with 16 channels each)
- Allows large networks
- Small memory footprint per GPU
- Good scaling.
- Somewhat parallel process (dependent on each other.)
- Requires large amount synchronization.
- At each layer computation depends on other GPUs
- Large amounts of data transfer.



Input data
Conv1
Conv2
Conv3
Conv4
Conv5
FC6
FC7
FC8
227× 227 × 3
55× 55 × 96
27× 27 × 256
13× 13 × 384
13× 13 × 384
13× 13 × 256
4096
4096
1000
224
224
3
11
11
11
11
Stride of 4
55
55
48
48
5
5
5
5
Max pooling
27
27
128
128
3
3
3
3
3
3
Max pooling
13
13
192
192
13
13
192
192
13
13
128
128
Max pooling
dense
dense
dense
2048
2048
2048
2048
1000

Multi-GPU Training - Ways to split the problem

Split Data across Multiple GPUs

- Multiple GPUs corresponding to whole architecture but a fraction of data.
- After each iteration (minibatch), the gradients are aggregated across GPU during training.
- Fully parallel process.
- Requires very little synchronisation
- Very little dependency on other GPUs
- Easy to implement.
- Good scaling.
- Does not allow large models - Architecture is restricted.
- Needs GPUs with large memory.

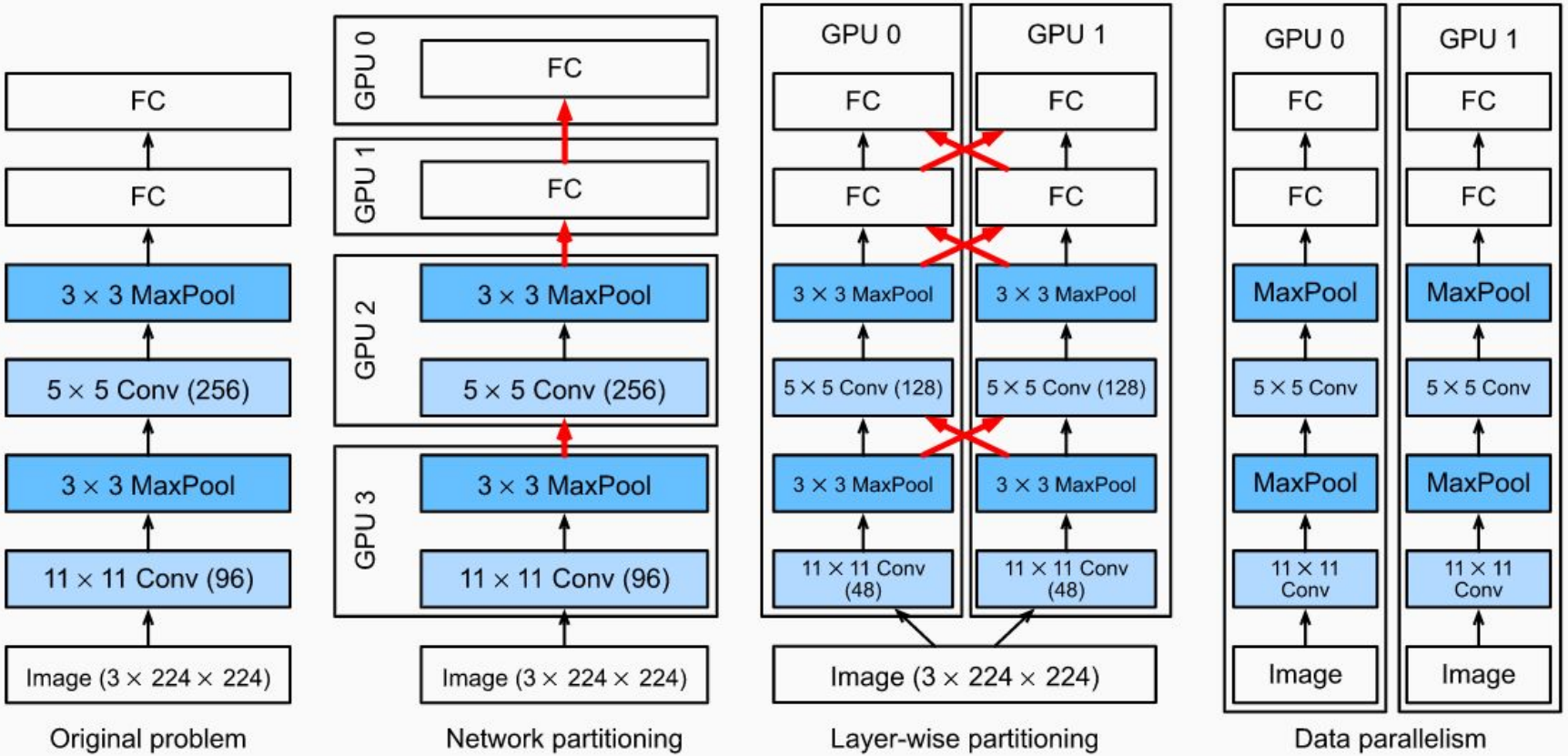
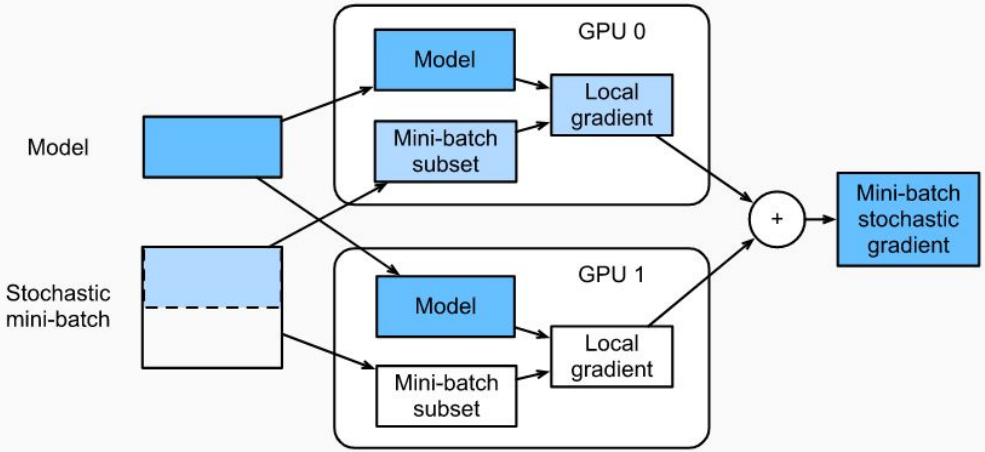


Fig. 12.5.2 Parallelization on multiple GPUs. From left to right: original problem, network partitioning, layerwise partitioning, data parallelism.

Data Parallelism



In general, the training proceeds as follows:

- In any iteration of training, given a random minibatch, we split the examples in the batch into k portions and distribute them evenly across the GPUs.
- Each GPU calculates loss and gradient of the model parameters based on the minibatch subset it was assigned.
- The local gradients of each of the k GPUs are aggregated to obtain the current minibatch stochastic gradient.
- The aggregate gradient is re-distributed to each GPU.
- Each GPU uses this minibatch stochastic gradient to update the complete set of model parameters that it maintains.

Data Parallelism

Points to remember:

- Increase the minibatch size k-fold when training on k GPUs
- Increase learning rate
- Adjust Batch normalisation

Data Synchronisation

We need to:

1. Distribute a list of parameters to multiple devices and attach gradients to them
2. Sum parameters across multiple devices

```
def get_params(params, device):
    new_params = [p.to(device) for p in params]
    for p in new_params:
        p.requires_grad_()
    return new_params
```

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
b1 weight: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       device='cuda:0', requires_grad=True)
b1 grad: None
```

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].to(data[0].device)
    for i in range(1, len(data)):
        data[i][:] = data[0].to(data[i].device)
```

```
data = [torch.ones((1, 2), device=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('before allreduce:\n', data[0], '\n', data[1])
allreduce(data)
print('after allreduce:\n', data[0], '\n', data[1])
```

```
before allreduce:
 tensor([[1., 1.]], device='cuda:0')
 tensor([[2., 2.]], device='cuda:1')
after allreduce:
 tensor([[3., 3.]], device='cuda:0')
 tensor([[3., 3.]], device='cuda:1')
```

Data Distribution

MXNET PYTORCH

```
data = torch.arange(20).reshape(4, 5)
devices = [torch.device('cuda:0'), torch.device('cuda:1')]
split = nn.parallel.scatter(data, devices)
print('input :', data)
print('load into', devices)
print('output:', split)
```

```
input : tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]])
load into [device(type='cuda', index=0), device(type='cuda', index=1)]
output: (tensor([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]], device='cuda:0'), tensor([[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]], device='cuda:1'))
```

Implementation

- Available framework in Pytorch, Tensorflow, Mxnet etc.
- Steps :
  - Initialize network in a single training loop
  - Training with multiple nodes:

As before, the training code needs to perform several basic functions for efficient parallelism:

- Network parameters need to be initialized across all devices.
- While iterating over the dataset minibatches are to be divided across all devices.
- We compute the loss and its gradient in parallel across devices.
- Gradients are aggregated and parameters are updated accordingly.

Implementation

- Pytorch :
  - DistributedDataParallel and DataParallel methods : Implements data parallelism.
  - DataParallel is single-process, multi-thread, and only works on a single machine
  - DistributedDataParallel is multi-process and works for both single- and multi- machine training.
  - DistributedDataParallel is more complex but faster, recommended from pytorch, although in the example shown uses DataParallel.

References:

1. Mining of massive datasets.
2. https://d2l.ai/chapter_computational-performance/multiple-gpus.html
3. https://www.youtube.com/watch?v=LfdK-v0SbGI