2 - Software & Hardware for Deep Learning

Marco Willi

Deep Learning Software

There are a variety of Deep Learning frameworks. These frameworks allow for easy configuration, training, and deploying of neural networks. They are often developed via Python API. Figure 1 shows some frameworks.

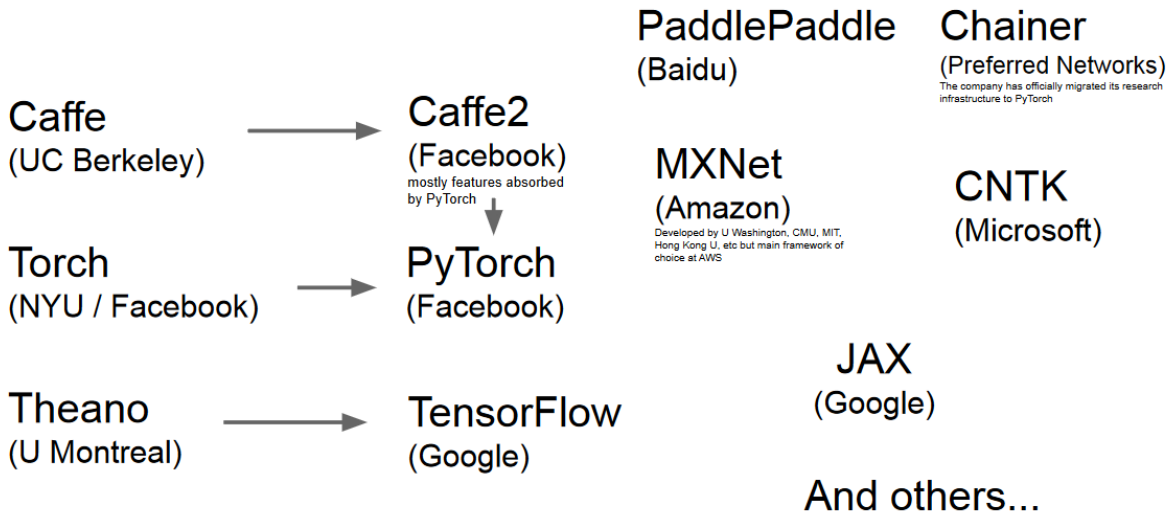

Figure 1: Frameworks (from Li (2022)).

Key features of such frameworks are:

- Fast development and testing of neural networks
- Automatic differentiation of operations
- Efficient execution on diverse hardware

Computational Graph & Autograd

At the core of neural networks is the *Computational Graph*. It automatically embeds dependent operations in a *directed acyclic graph (DAG)*. Gradients are tracked as needed, allowing variables to be efficiently updated/trained.

The following shows an example in Numpy where we define computations and manually calculate derivatives. The graph is shown in Figure 2.

$$f(\mathbf{A}, \mathbf{B}, \mathbf{C}) = \sum_{ij} \left((\mathbf{A} \odot \mathbf{B}) + \mathbf{C} \right)_{ij} \tag{1}$$

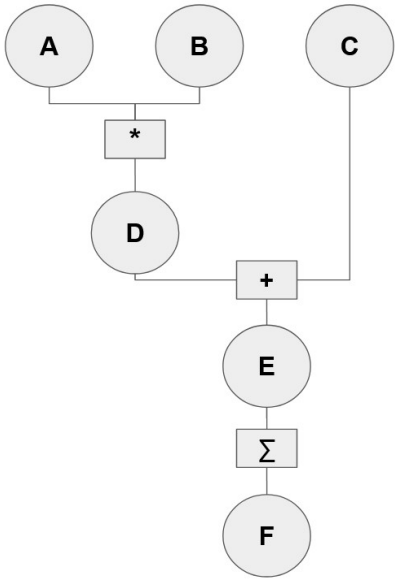

Figure 2: Computational Graph.

```
import numpy as np

np.random.seed(123)

H, W = 2, 3

a = np.random.random(size=(H, W))
b = np.random.random(size=(H, W))
c = np.random.random(size=(H, W))

d = a * b
```

```
e = d + c
f = e.sum()

df_de = 1.0
de_dd = 1.0
de_dc = c
dd_da = b

df_da = df_de * de_dd * dd_da

print(df_da)
```

```
[[0.9807642  0.68482974 0.4809319 ]
 [0.39211752 0.34317802 0.72904971]]
```

Here's the same example in PyTorch. Using `x.backward()`, gradients with respect to `x` are computed for variables connected to `x`.

```
import torch

np.random.seed(123)

H, W = 2, 3

a = torch.tensor(a, requires_grad=True)
b = torch.tensor(b, requires_grad=True)
c = torch.tensor(c, requires_grad=True)

d = a * b
e = d + c
f = e.sum()

f.backward()
print(a.grad)
```

```
tensor([[0.9808, 0.6848, 0.4809],
        [0.3921, 0.3432, 0.7290]], dtype=torch.float64)
```

Here are the nodes of the computational graph.

To perform the computation on a GPU, a simple instruction is enough:

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

a = a.to(device=device)
b = b.to(device=device)
c = c.to(device=device)
```

```
Using cpu device
```

PyTorch

In this class, we use PyTorch. PyTorch has gained enormous popularity in recent years and stands out for its high flexibility, a clean API, and many open-source resources.

Fundamental Concepts

- Tensor: N-dimensional array, similar to `numpy.array`
- Autograd: Functionality to create computational graphs and compute gradients.
- Module: Class to define components of neural networks

Tensors

`torch.Tensor` is the central data structure in PyTorch. Essentially very similar to `numpy.array`, it can be easily loaded onto GPUs.

Tensors can be created in various ways. For example, from lists:

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
print(x_data)
```

```
tensor([[1, 2],
        [3, 4]])
```

Or from numpy.ndarray:

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(x_np)
```

```
tensor([[1, 2],
        [3, 4]])
```

Or from other tensors:

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.5436, 0.0411],
        [0.4006, 0.7320]])
```

Or with randomly generated numbers or constants:

```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
 tensor([[0.5402, 0.1558, 0.5754],
        [0.6255, 0.2027, 0.8695]])

Ones Tensor:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])

Zeros Tensor:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

Tensor attributes:

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

There are over 100 operations that can be performed on a tensor. The full list is available here.

Indexing and Slicing:

```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[:, -1]}")
tensor[:,1] = 0
print(tensor)
```

```
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Joining tensors:

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

```
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

Arithmetic operations:

```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will have the same
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)


# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Autograd

To train neural networks, backpropagation is typically used. This calculates the gradient of the loss function with respect to the model parameters. To compute these gradients, PyTorch provides an *auto-diff* functionality: `torch.autograd`. This can automatically compute gradients for a *computational graph*.

The following is an example using a 1-layer neural network (see Figure 3):

Here is the definition of the network in PyTorch:

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5

, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

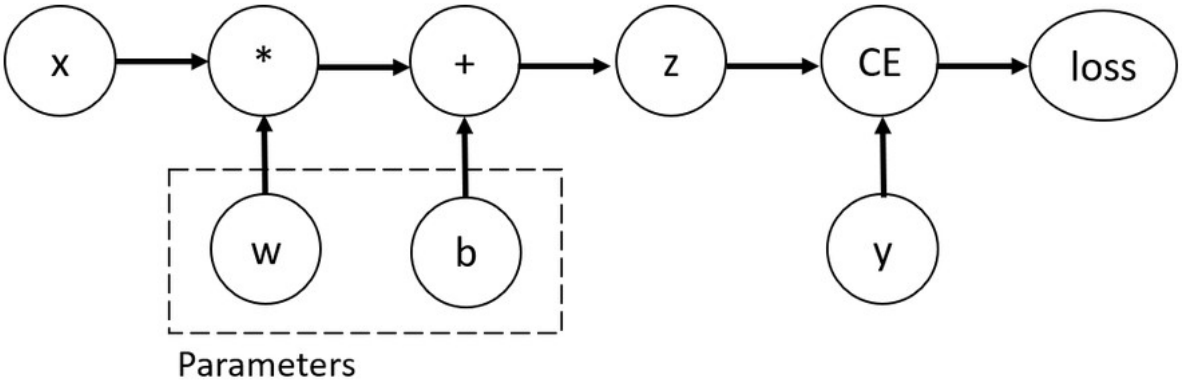

Figure 3: Source: PyTorch

We can now use Autograd to compute the gradient:

```
loss.backward()
print(w.grad)
print(b.grad)
```

```
tensor([[0.1406, 0.0568, 0.3045],
        [0.1406, 0.0568, 0.3045],
        [0.1406, 0.0568, 0.3045],
        [0.1406, 0.0568, 0.3045],
        [0.1406, 0.0568, 0.3045]])
tensor([0.1406, 0.0568, 0.3045])
```

torch.nn

PyTorch provides various building blocks for creating neural networks. These are available in `torch.nn`. Additionally, you can define any compositions of such building blocks that inherit from `torch.nn.Module`. A neural network is typically a `torch.nn.Module`. Each module implements the `forward()` method to define how data is processed.

Here is an example:

```
from torch import nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
```

```
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

You can also visualize the model:

```
model = NeuralNetwork()
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

To use the model, you can pass input data. This will execute the `forward()` method, along with background operations.

```
X = torch.rand(1, 28, 28)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

```
Predicted class: tensor([1])
```

The executed operations will look like this:

torch.optim

To optimize the parameters of a model, you need an optimization algorithm. `torch.optim` implements various algorithms, such as *Stochastic Gradient Descent* or the often used *Adam Optimizer*.

```
from torch import optim
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

You can then use the optimizer to adjust the parameters, you just need to define a loss function:

```
loss_fn = torch.nn.CrossEntropyLoss()
for i in range(0, 3):
    input, target = torch.rand(1, 28, 28), torch.randint(low=0, high=10, size=(1, ))
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

> **Note**
>
> Note `optimizer.zero_grad()` which resets the accumulated gradients of the variables to 0.

Training Loops

Typically, you put together a training loop to train a model. A training loop iterates over batches of data and optimizes the model parameters with each iteration.

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

> **Note**
>
> PyTorch-Lightning provides many functionalities to simplify managing training loops. It simplifies using PyTorch similar to how Keras does for TensorFlow.

Pre-trained models

Since training models can be time-consuming and expensive, pre-trained models are often used. They allow models to be adapted to a specific task more quickly and cost-effectively. In many

areas, particularly NLP and computer vision, using pre-trained models is standard. PyTorch provides `torchvision` for computer vision applications. `torchvision` provides functionalities useful for modeling image data. Pre-trained models can also be easily integrated, as shown in the following example:

```
from torchvision.models import resnet50, ResNet50_Weights

weights = ResNet50_Weights.IMAGENET1K_V2
model = resnet50(weights=weights)
```

Other Frameworks

Other important frameworks are listed below (not exhaustive).

TensorFlow

For a long time, PyTorch and TensorFlow have been the biggest deep learning frameworks. TensorFlow stands out with a clean high-level API with `Keras`, which allows for easy implementation of complex models. Traditionally, TensorFlow is well established in the industry, while PyTorch is widely used in academia.

Scikit-Learn

`Scikit-Learn` is THE machine learning framework in Python. However, Scikit-Learn never covered the area of neural networks and lacks auto-diff functionality. Therefore, Scikit-Learn is irrelevant when training neural networks. However, Scikit-Learn functionalities are often used to carry out the machine learning process, such as splitting datasets into train, validation, and test sets. Also, visualizations, such as the confusion matrix or calculating metrics, can be done via Scikit-Learn.

ONNX

ONNX is an open format to represent machine learning models. It allows models trained in one framework to be transferred to another. Trained models can also be deployed on various platforms.

Monitoring

When training models, monitoring the training process, debugging, and logging hyperparameters, metrics, etc., is very important. Various tools enable these functionalities. Well-known examples are `TensorBoard` and `Weights & Biases`.

Hardware

Tensor Operations

In neural networks, there are many tensor operations. Tensors are essentially multi-dimensional arrays, such as a scalar x, a vector $\mathbf{x}$, or a matrix $\mathbf{X}$.

Figure 4 illustrates a matrix multiplication, a typical representative of a tensor operation. As you can see, the calculations (entries of the matrix $\mathbf{AC}$) are independent of each other and can be fully parallelized.

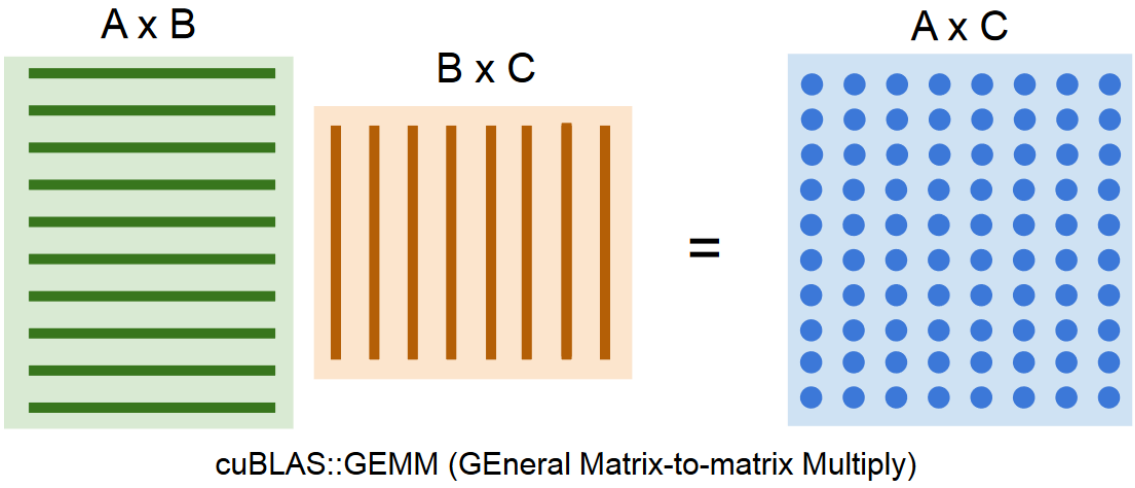

Figure 4: Matrix Multiplication (from Li (2022)).

Graphics Processing Units (GPUs)

GPUs have made deep learning possible in the first place. With their parallel structure, they can efficiently compute parallelizable tasks such as tensor operations.

CPUs have far fewer cores than GPUs, but they are faster and can handle more complex tasks. CPUs are therefore ideal for sequential tasks. GPUs have many more cores, which are less

complex and slower. Therefore, GPUs are excellent for parallel tasks. Figure 5 illustrates the differences.

	Cores	Clock Speed	Memory	Price	Speed (throughput)
CPU (Intel Core i9-7900k)	10	4.3 GHz	System RAM	$385	~640 GFLOPS FP32
GPU (NVIDIA RTX 3090)	10496	1.6 GHz	24 GB GDDR6X	$1499	~35.6 TFLOPS FP32

Figure 5: CPU vs GPU example (from Li (2022)).

CUDA & cuDNN

CUDA is an API by Nvidia to perform computations on the GPU. It allows parallelizable tasks to be implemented efficiently. cuDNN is a library that efficiently executes certain operations, such as convolutions, in neural networks on the GPU. cuDNN is based on CUDA and significantly accelerates the training of neural networks. Figure 6 illustrates speed differences when training various neural networks with CPU, GPU, and optimized cuDNN.

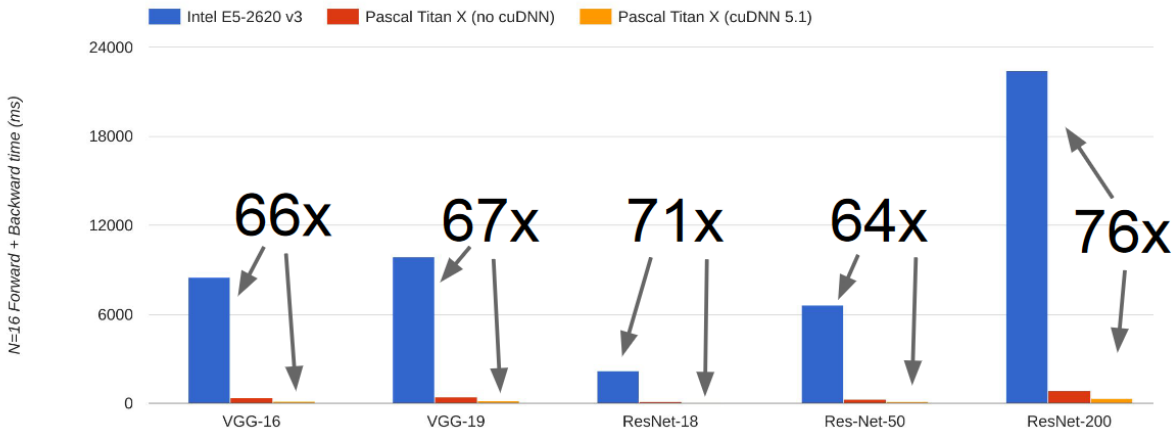

Figure 6: Speed comparison (from Li (2022), data from Link)

Data Loading

A crucial bottleneck in practice is the transfer of data (such as images) from the disk to the GPU. If this transfer is not fast enough, it is referred to as *GPU starvation*. There are several approaches to solve this problem:

- Read the data into RAM (not feasible for larger datasets)
- Use fast disks, such as SSDs
- Utilize multiple CPU threads to read data in parallel and keep it in RAM (*pre-fetching*)

Figure 7 shows the various components.

Figure 7: Source: Li (2022)

Deep learning frameworks like PyTorch implement special classes that allow data to be prepared in multiple threads. Sometimes a certain number of CPU cores is needed to supply a GPU with enough data. Figure 8 shows a starved GPU: You can clearly see that the utilization repeatedly drops to 0 because the GPU has to wait for data.

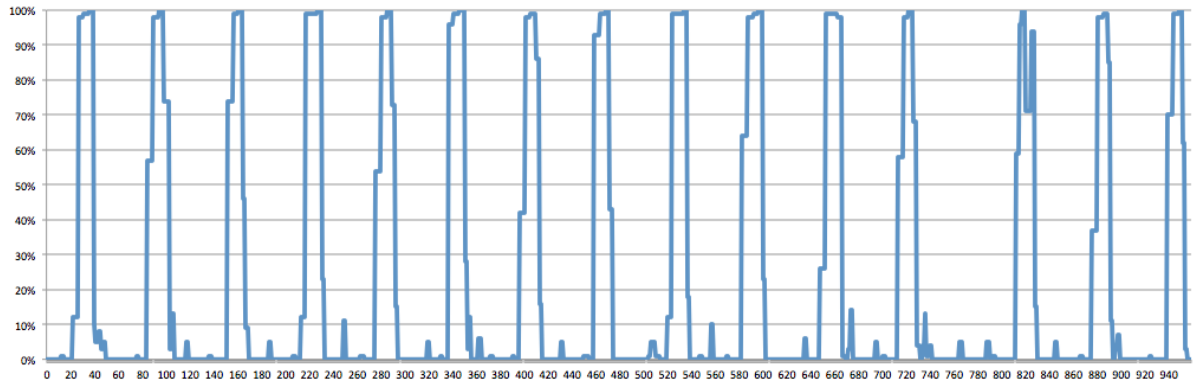

Figure 8: The Y-axis shows the GPU utilization in percentage, while the X-axis represents time. Source

GPU Parallelism

Models can also be trained on multiple GPUs. There are two main paradigms: *data parallelism* and *model parallelism* (see Figure 9). With *data parallelism*, each GPU has a copy of the model, and each GPU is trained on different data batches. With *model parallelism*, the model is split across multiple GPUs. Models can be trained on a server with multiple GPUs or even over the network (*distributed*). ML frameworks provide functionalities to handle these.

References

Li, Fei-Fei. 2022. "CS231n Convolutional Neural Networks for Visual Recognition." Lecture {Notes}. https://cs231n.github.io.

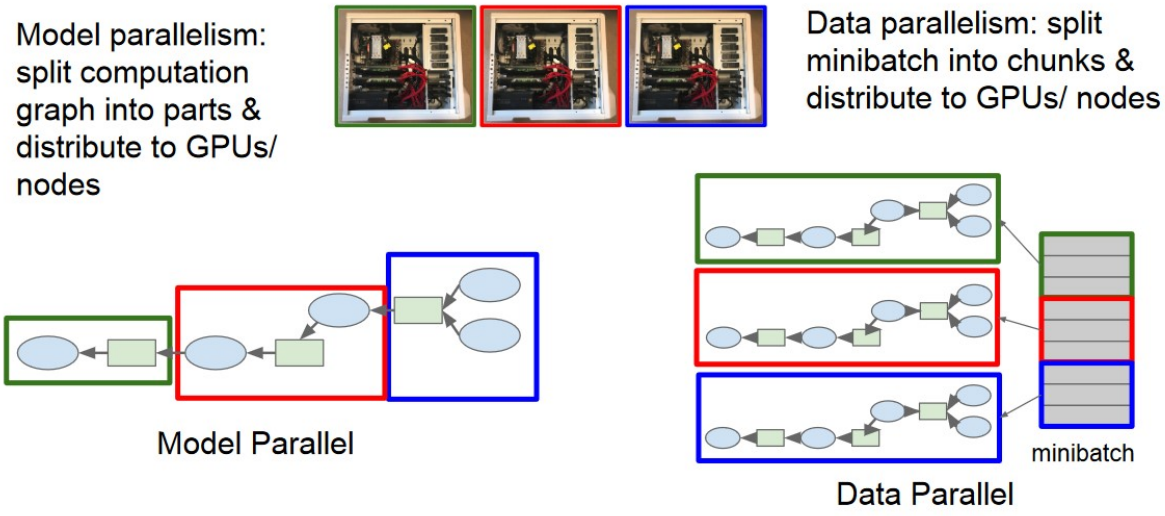

Figure 9: Data and Model Parallelism (from Li (2022)).