CS242 – Computing at Scale

Matheus S. Farias[*]

School of Engineering and Applied Sciences, Harvard University

Fall, 2021

Abstract

This document is composed by lecture notes of CS242 – Computing at Scale, taught by Professor H.T. Kung in Fall 2021. I am responsible to all mistakes here written.

Contents

§1 09/01 – Course Overview

A first and important discussion about the difference between three hyped areas. One can see a graphical perspective in Figure 1.

[*]E-mail address: matheusfarias@g.harvard.edu

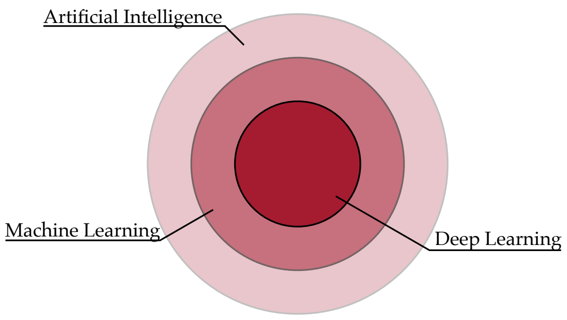

Figure 1. Deep learning is a subset of machine learning, that is a subset of artificial intelligence.

- **Artificial Intelligence** – a program that can sense, reason, act and adapt.
- **Machine Learning** – algorithms whose performance improve as they are exposed to more data over time.
- **Deep Learning** – subset of machine learning in which multilayered neural networks learn from vast amounts of data.

§1.1 The Golden Age

We are in a new golden age in the field of computer architecture. Previously if more computing resources were demanded, one could simply wait for new processors or CPUs that run faster and more efficiently. The economy also depends on the increase of computing performance to facilitate the innovation in various fields. However, the Dennard scaling and Moore's law have almost ended. The alternative architectures, such as domain-specific accelerators, can play a role to continue scaling of performance and efficiency.

In CS242, we are interested in **scaling computations** for AI-assisted systems focusing on acceleration strategies, both **speed** and **efficiency**.

§1.2 Convolutional Neural Networks

Convolutional neural network (CNN) (Lecun et al., 1998) is one kind of neural network architecture. It uses convolutions between layers, and more common used on image processing applications due to high dimension. In general, a neural network consists of two steps to work:

- **Inference** – A series of **inner products** (matrix multiplications), each of them is just a sequence of **multiplier-accumulator (MAC)** operations.
- **Training** – A more hardware demanding process, needs to take **gradients**.

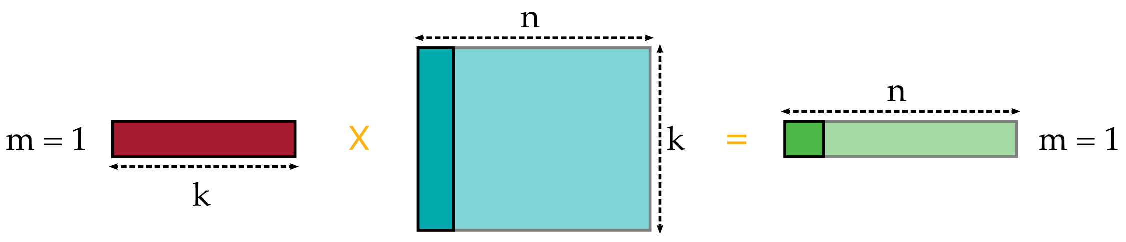

Figure 2. GEMM schematic for fully connected layers.

One of the greatest challenges in the area is the **energy consumption**. It is always a trade off choice, the higher the performance in a metric, the less it will be in another one.

To understand how CNN works, we must first learn the idea of General Matrix to Matrix Multiplication (GEMM).

§2 09/08 – Accelerators and Parallelization

§2.1 General Matrix to Matrix Multiplication

Part of the Basic Linear Algebra Subprograms (BLAS), developed in 1979, GEMM was created to tackle matrix multiplication in **high dimensions**. Almost all the running time of a CNN is spent on **fully connected** and **convolutional layers**, implemented using GEMM. Image processing applications usually work with very big matrixes (also considering RGB channels). Big matrixes lead to big number of operations, which can result in **billions of floating point operations per second (FLOPS) on a single frame!**

Let's see how GEMM works with fully connected and convolutional layers.

§2.1.1 Fully Connected Layers

We have an input vector with the red color (see Figure 2), where it has **k** input values. In this diagram we are considering a mini-batch size (**m**) of just 1 vector. Then we multiply the input vector to a matrix of weights (actually, it's a dot product) with **n** neurons. The output of this layer is a matrix (or vector) **m** x **n**.

Fully connected layers are more specific for classification.

§2.1.2 Convolutional Layers

In this case we are frequently talking about images, so our input is a three-dimensional array, and the depth is related to colors (RGB) and also opacity (RGBA). This process consists on taking a convolution kernel, which is also a three-dimensional array, and applying it at many portions of the input image.

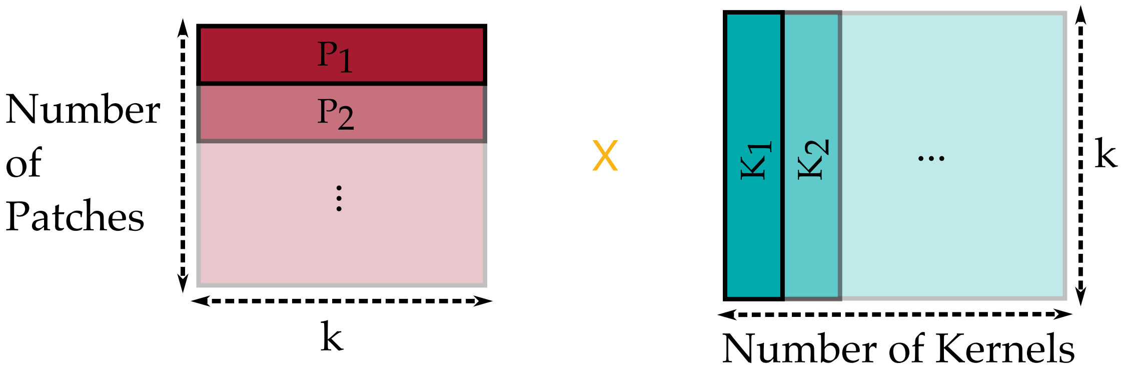

Figure 3. Final GEMM schematic for convolutional layers.

So the kernel is applied to a grid of points, where each point multiplies input values and weights, and then sum. It looks like an edge detector, once the kernel has a specific pattern, and when the input is similar to the kernel, the operation returns a high value (**inner products represent similarity**).

Convolutional layers are more specific for feature extraction.

There are some relevant variables that constitutes a CNN:

- **Kernel** – The size of the mask we are going to apply in the input image.
- **Padding** – The amount of new lines filled with zero that aims to reduce the loss of dimension in the output. Try to always use odd number in the kernel to facilitate the calculation here.
- **Stride** – The slide you make after every convolution step.
- **Pooling** – A kernel that do a specific operation, two common ones are max pooling and average pooling.

Padding and **stride** are used to administrate the output dimension. Assuming the input vector with dimensions (n_H, n_W), the kernel with dimensions (k_H, k_W), the padding with dimensions (p_H, p_W), and the stride with dimensions (s_H, s_W), the final output is calculated as show in Equation 1.

$$\left\lfloor\left(\frac{n_H + p_H - k_H + s_H}{s_H}\right)\right\rfloor \times \left\lfloor\left(\frac{n_W + p_W - k_W + s_W}{s_W}\right)\right\rfloor \tag{1}$$

After applying the kernel, our multiplication (without considering padding and stride) is represented in Figure 3. Note that if the stride is less than the number of kernels, we may have overlapping information between lines/columns. At a first glance, it sounds inefficient, but it's worth it.

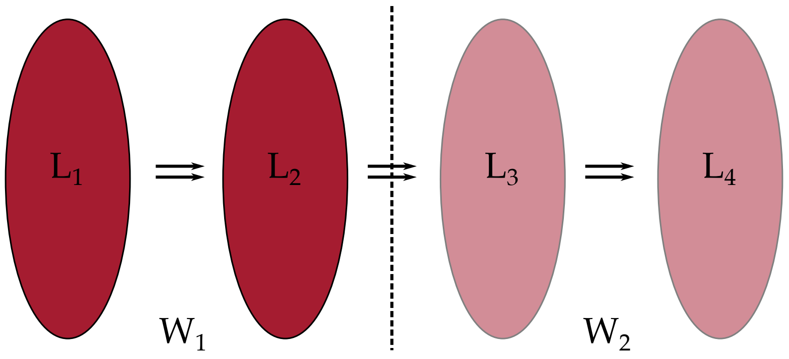

Figure 4. Diagram of model-parallel: interlayer method of parallelization.

§2.2 Parallelism

The idea of scaling is to use many computers to train and perform tasks. The backward pass is **two times** more demanding than the forward. Also, it requires the activations performed by the forwarding pass, thus it needs to **store the calculations** (can be a difficult task to high dimensions).

We have mainly three different ways to do parallelization in neural networks: **data parallel**, **model-parallel: interlayer**, and **model-parallel: intralayer**.

§2.2.1 Data-parallel

In this model, each worker has a copy of the entire model, and each of them gets a different mini-batch. This way, the **forward and back-propagations are the same**, but the problem is the **weight update**.

To deal with communication during weight update, we have two common ways:

- **Ring reduction** – each worker communicates with two other neighbors, thus it has $2(N-1)$ synchronizations total (one can think of communication with 4 neighbors, it's called toroidal and it's used in TPUs).
- **Fully connected** – each worker communicates with everyone ($N-1$ neighbors), each of them with $N-1$ substeps.

§2.2.2 Model-parallel: interlayer

Also called **pipeline parallel**, this model is challenging as it **needs communication on each stage** (forward and back). Also, the timing of passing the calculation from a previous worker to the next one can be a problem, once the next one can be **busy**. The diagram of this method is shown in Figure 4.

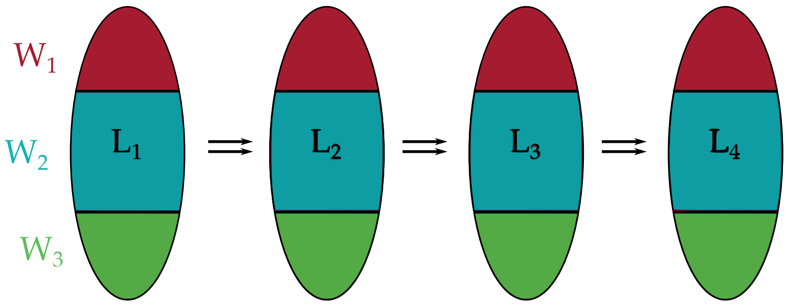

Figure 5. Diagram of model-parallel: intralayer method of parallelization.

§2.2.3 Model-parallel: intralayer

There are various tricks in intralayer to reduce communication, such as alternating between horizontal and vertical partitions, but this approach also suffers from it being difficult to overlap communication with computation. But when the model is too big for data parallelism you have to deal with it. The diagram of this method is shown in Figure 5.

§3 09/13 – SIMD and Systolic Arrays

References

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.