DSC-204A: Scalable Data Systems, Winter 2024

ML System - 2

Lecturer: Hao Zhang *Scribe: Nandita Sanjivi, Dinesh Karthikeyan and Prathish Murugan*

1 ML System history

Machine learning systems are coped with the unification of machine learning. The first and famous ML system is the primary server as it was discovered that ML models can be trained with iterative-convergent algorithms. As more and more ML components like models and optimization algorithms are unified, the whole ML systems evolve. This lecture dives into the second unification of Neural Nets and different types of models.

1.1 Recap of Primary Server

Pros of Primary Server:

- In general it is an abstract of the iterative-convergent algorithm
- Relaxed Consistency: It only requires stale synchronous unlike traditional data processing
- Nice and similar style of Map-Reduce, with the flexibility to implement various functions

Cons of Primary Server:

- Primarily for CPU's so questionable scope of extension to GPU's
- It assumes the network is small, so it is prone to communication bottleneck

Communication bandwidth between one node of a GPU is pretty high (100x) compared to cross node, but the primary server is suitable for only one node, making it redundant.

2 The Second Unification: Neural Networks

Talks about the paper "ImageNet Classification with Deep Convolutional Neural Networks" as the paper surpassed the existing state-of-the-art method by 10 points in ImageNet and beat the incumbent graphic models. It also can scale with the size of data and the model itself is simple as it is optimized by 1st-order method SGD. Training with GPU made the model faster than the usual CPU methods.

2.1 Characteristics of deep learning

- It is Iterative-convergent because of SGD
- It is less diverse than all of ML but still has various architectures.
- Models are becoming compute-intensive and larger with transformer architecture
- Existing data systems aren't very suitable for NN as MapReduce isn't for iterative-convergence and Spark is very coarse-grained and not for NN operations.
  Primary server but we still need to implement the model in a push-and-pull interface.

2.2 Background of DL Computation

In a Neural Network, we need to train the model to obtain the weights, which require forward and backward propagation. Forward propagation calculates the loss and backward propagation calculates the gradients and uses that gradients to update the parameters. We iteratively do this till the loss is converged.

$$\theta^{(t+1)} = f(\theta^{(t)}, \nabla_{\theta^{(t)}} L(\theta^{(t)}, D^{(t)})) \tag{1}$$

Here θ is the parameter, D is the data, f is the optimizer function which updates the weight and $\nabla_{\theta^{(t)}}$ is the loss function and gradient.

2.3 Computational Layer in DL: Forward propagation

We represent an NN with a data flow graph and a layer($f_i(x)$) in a neural network is composed of a few finer computational operations. Let's take an example,

$$z = f_i(x); \quad y = Wx + b; \quad z = \text{ReLU}(y) \tag{2}$$

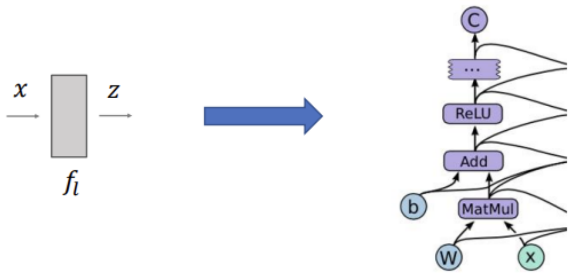

Figure 1: A layer in forward propagation

Here the node represents computational operators like Add and edges represent the data flow direction. Here the data flows through MatMul, Add, and then ReLU. Following this we denote the backward propagation through the chain rule which derives the gradients.

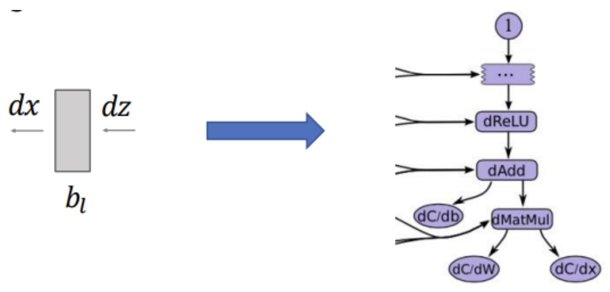

Figure 2: Backward pass

The backward pass-through layer l is denoted by b_l and computes gradients of input dx and parameters W, b from output gradient dz. The term dx is the gradient input for the previous layer $l-1$. This represents a reverse data flow where gradients propagate backward through the network.

2.4 A Layer as a Data Flow Graph

After getting the forward and backward values, we get the output which is also a tensor, which describes the gradient derived. When the forward and backward parts are put together, this gives rise to the layer as a data flow graph. The gradients are computed by performing auto differentiation. Data flow graphs are the most common representation of neural networks. The graph includes the computation made by the optimizer used as well.

2.5 Practice

From the neural network provided in Figure 3,

$$\theta^{(t+1)} = f\left(\theta^{(t)}, \nabla_L\left(\theta^{(t)}, D^{(t)}\right)\right)$$

$$L = \text{MSE}(w_2 \cdot \text{ReLU}(w_1 x), y) \quad \theta = \{w_1, w_2\}, D = \{(x, y)\}$$

$$f(\theta, \nabla_L) = \theta - \nabla_L$$

Figure 3: Neural Network Description

- We can observe that the neural network has two weights w_1 and w_2. Further, the input variable is x and the output variable is y. Further, we make use of a simple optimizer which is the difference between θ and the gradient. Now, to represent this as a data flow graph, we have three steps - Forward (obtain loss), Backward (derive gradient), and Update the weight.
- The MSE block is further broken down into primitive operations. During the backward stage, we focus on obtaining gradients only with respect to w_1 and w_2. The MSE' depicted in the Backward graph is the gradient of MSE. The same notion is followed for the ReLU blocks as well. In reality, the entire data flow graph is taken care of by the Deep Learning library.

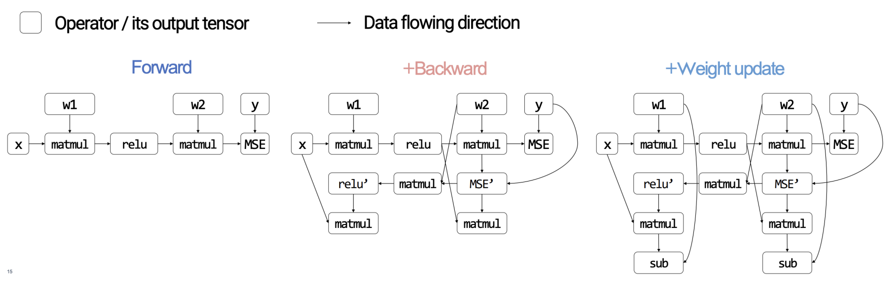

Figure 4: Data Flow Graph of Neural Network

2.6 How are data flow graphs programmed in reality?

- Firstly, the neural network is defined by defining the operations and layers. A few examples of this would be mentioning if the layers are fully connected or to perform convolution or making it recurrent.
- After defining the neural networks by the operations provided by the Deep Learning library, the data I/O is specified. Essentially, mentioning where the input data is read from and where the output data is written. This is similar to MapReduce or Spark, where a pointer is provided to the location of the training data.
- Next, the loss function to be utilized is to be provided. A few instances of this would be the L2 loss or the Softmax or ranking loss.,
- Further, the optimization algorithm is given. For example, this could be SGD, Adam, AdaGrad, Momentum SGD, etc.
- The system will convert the provided neural network definition into a data flow graph, and perform training and updates.

2.7 Discussion: Comparing this model vs. Spark, parameter server, MapReduce

- The parameter server and MapReduce are similar in this context. Data flow programming is quite restrictive in the sense that you can choose only from whatever is offered.
- To perform new operations or algorithms not provided in the framework, on needs to take on the role of a system builder rather than a user (but these libraries have improved a lot over the years to include a variety of operations and algorithms). Therefore, the parameter server and MapReduce have more flexibility.
- The programming process is made easy in the case of data flow graphs since you build it in a confined sense.
- This goes for Spark as well, where operations and auto differentiations are just added for the same task to be performed. The Deep Learning library inherits the spirit of Spark.
- The evolution can be seen as - MapReduce, Spark, parameter server, and Data flow programming (most evolved).

2.8 Systems side of auto-differentiable libraries

- The first few libraries were: torch, theano. These frameworks are not very famous, the work was performed on CPU and therefore using small neural networks. The power of neural networks hadn't been fully unleashed yet.
- The next library: Caffe. This transformed the entire academia space. This is also attributed to well-written code and is built using C++. However, programming in this library requires expertise.
- In 2015, TensorFlow was launched by Google. Offered a plethora of operations, functionalities, and features; making it versatile and powerful.

2.9 DMLC MXNet:

DMLC MXNet, an open-source initiative spearheaded by the renowned Tianqi Chen, who is also credited with the development of TVM, represents a pioneering effort in the domain of neural network frameworks. Despite its foundational contributions to the field, MXNet has struggled to maintain relevance and achieve widespread adoption, a situation attributed to its reliance on what has become outdated technology in the fast-paced evolution of neural network methodologies.

2.10 DyNet and Chainer

: In the span of 2017 to 2018, the field witnessed a paradigm shift towards models capable of processing natural language in a more literal and dynamic manner, challenging the static nature of conventional convolutional neural networks (CNNs). This shift underscored the need for frameworks that could adapt to the inherently dynamic nature of natural language processing. Within this context, dynamic neural networks emerged as a critical development, with Professor Hao Zhang making notable contributions to the coding and development of DyNet. The integration of DyNet with Chainer marked a significant milestone, creating a composite framework that effectively harnessed the dynamic capabilities essential for modern neural network applications. Despite this innovative approach, the widespread acceptance and success of these models were impeded by engineering limitations, until Meta's intervention, which significantly enhanced the efficiency and application of dynamic neural networks.

2.11 PyTorch

: In response to the evolving demands of the field, the team at Meta developed PyTorch, a framework that has since emerged as a cornerstone in the landscape of neural network research and application. PyTorch, building on the legacy of the original Torch framework, which was primarily based on Lua and faced criticisms for its inefficiencies, has been celebrated for its robustness and adaptability, cementing its status as a leading framework in the domain.

3 Overview of Deep Learning Toolkits

: The ecosystem of neural network frameworks is characterized by a rich diversity, with each framework tailored to specific computational domains, ranging from vision-centric applications, exemplified by CNNs, to complex natural language processing tasks, facilitated by RNNs and Transformers. Recent advancements

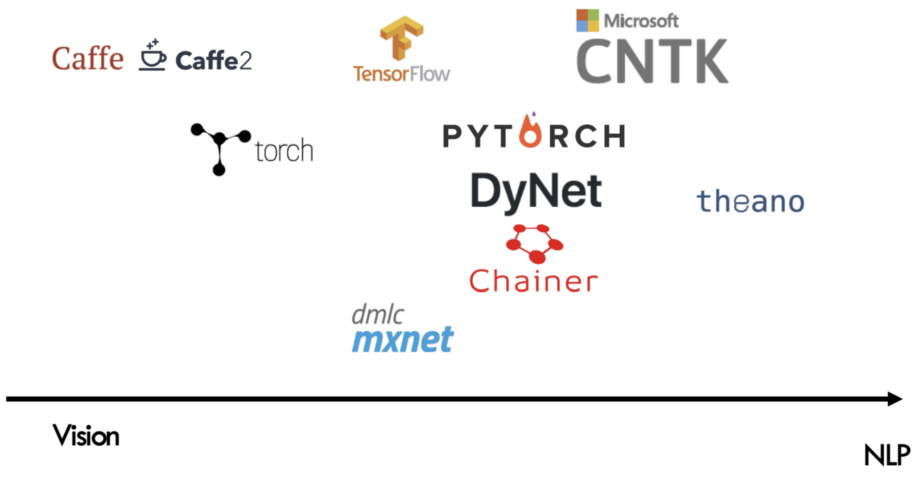

Figure 5: Vision Vs NLP

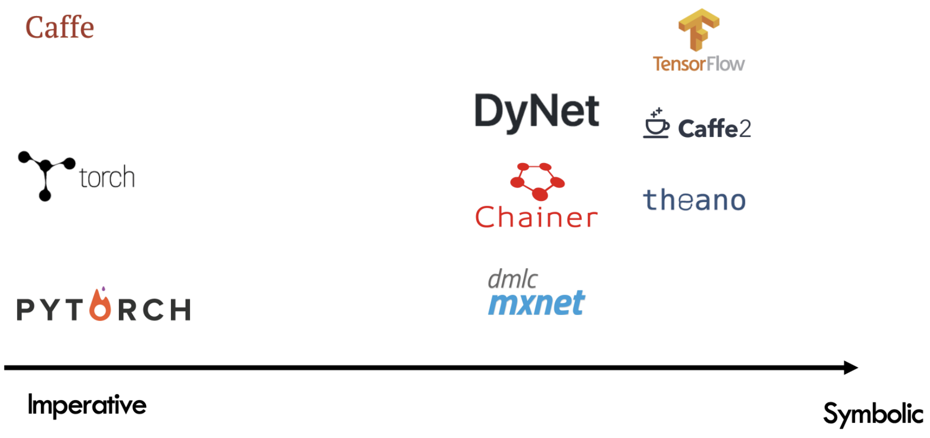

Figure 6: Symbolic Vs Imperative

have blurred these boundaries, with CNNs evolving to incorporate functionalities traditionally associated with Transformers, which remain unparalleled in their processing capabilities.

This segmentation of frameworks, from vision-focused tools like Caffe2, early versions of Torch, and MXNet, to NLP-oriented frameworks like DyNet and Theano, highlights the breadth of the field.

A fundamental distinction within this ecosystem is the division between imperative and symbolic programming paradigms.

3.1 Advantages of Symbolic Programming

: Symbolic programming significantly enhances the optimization process for frameworks like Spark by providing a comprehensive overview of the operations to be performed. This holistic view allows for preemptive optimization of the graph, leading to greater efficiency throughout execution.

3.2 Drawbacks of Symbolic Programming

: The programming approach in symbolic systems is often seen as non-intuitive, demanding a broad understanding of the graph's final structure from the onset. Debugging becomes a complex task; identifying errors after execution is challenging due to the inability to inspect individual outcomes directly. Furthermore, this paradigm lacks flexibility, necessitating the pre-definition of symbols before execution, a limitation that persists even in smaller-scale models.

3.3 Benefits of Imperative Programming

: Imperative programming is hailed for its intuitive nature, simplifying the development process significantly. It allows for immediate execution, debugging, and the insertion of breakpoints, streamlining the development process.

3.4 Limitations of Imperative Programming

: However, imperative programming does not provide a comprehensive understanding of the neural network's architecture to the system, which impedes its ability to optimize effectively. This method is capable of optimizing individual lines of code but may not achieve the most efficient overall model performance.

3.5 Conclusion for Symbolic vs Imperative

: TensorFlow and PyTorch exemplify the symbolic and imperative programming paradigms, respectively. TensorFlow offers a highly efficient platform suitable for deploying across various hardware configurations, favored for its optimization capabilities. However, its programming complexity makes it less accessible for development purposes. In contrast, PyTorch's imperative nature makes it more user-friendly for programming and debugging, ideal for development phases. This distinction suggests TensorFlow's suitability for deployment stages, where efficiency and hardware compatibility are paramount, once the model and its parameters have been firmly established.

Frameworks such as TensorFlow epitomize the symbolic paradigm, offering highly efficient, deployment-ready solutions. In contrast, PyTorch is emblematic of the imperative approach, preferred for its flexibility in research and developmental contexts. PyTorch distinguishes itself through its ability to shift towards a more efficient mode of operation through the use of decorators like 'torch.script' or 'torch.compile', a feature that has contributed significantly to its success.

3.6 Limitation of Dataflow Graphs

: The fundamental inquiry revolves around whether every model can be encapsulated within a graph structure. It's possible to architect your model in advance of execution, implying the dataflow graph remains static, and unaffected by varying inputs. This pre-construction of the dataflow graph does not entail an analysis of the inputs. However, as the machine learning (ML) field has advanced, there emerged models that dynamically adjust based on the input. This topic will be explored further in the subsequent class.

The models in question, such as CNNs, Transformers, and BERT, exhibit indifference to the variability of input space, demonstrating the capacity to process any form of input.

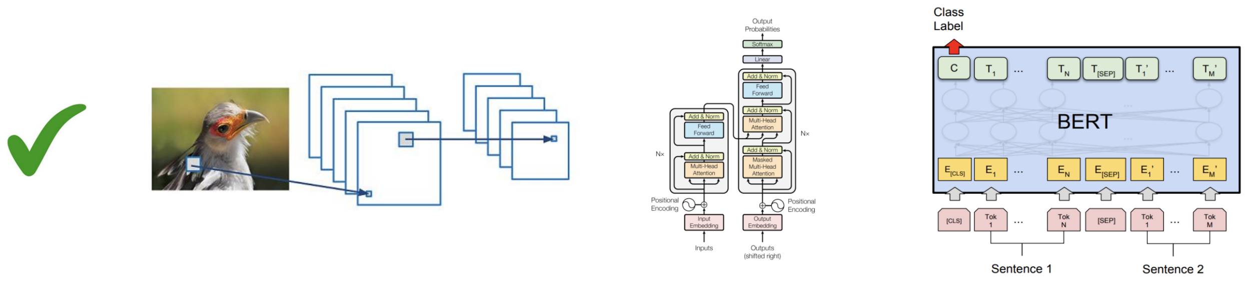

Figure 7: Models compatible with input variance

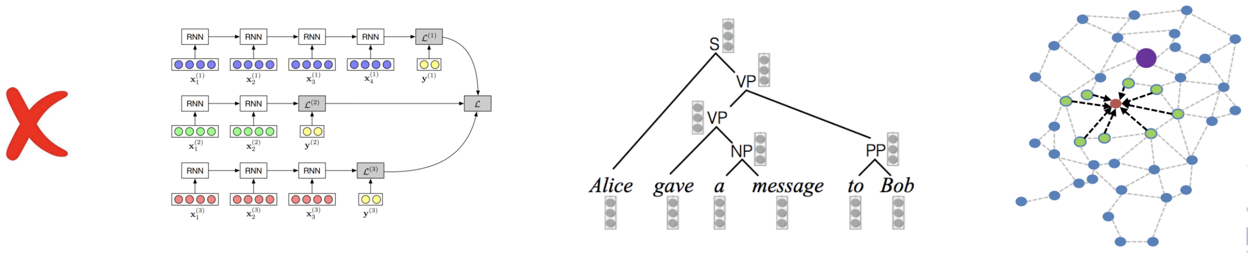

Figure 8: Models depending on input size

Conversely, in the context of Recurrent Neural Networks (RNN), the model faces challenges with novel inputs, executing a variable number of sequences corresponding to the number of inputs provided. For instance, should five inputs be introduced, the model will undertake five computational steps, with the sequence count fluctuating in response to input quantity. This principle also applies to tree-structured neural networks, where the structure of the tree varies based on the input. In these instances, the neural network adheres to a grammatical tree structure, with the computational process evolving as the grammatical structure shifts.

This adaptability underscores the success of PyTorch, which employs an imperative programming approach. There's no requirement to pre-construct a data flow graph; instead, following a successful implementation, a data flow graph can be developed through sequential execution, enhancing the framework's flexibility in handling dynamic models.