CSE 291A/DSC 291: Deep Learning Systems, Spring 2026

2: Basics: Modern DL, computational graph, frameworks

Lecturer: Hao Zhang *Scribe: Yiming Zhao, Xinle Yu, Ohm Rishabh Venkatachalam, Chenrui Liu*

1 Introduction to Deep Learning Systems

Today's lecture will be light as we review machine learning in 30 minutes. I assume you have already taken some machine learning courses and know most of the basics, but we are going to review these models, especially popular ones, from a compute perspective. Once we finish that part, we will discuss the compute representation for machine learning models, known as a data flow graph, along with a few different machine learning frameworks.

The basic idea of deep learning is that you have many layers composed into a multi-layer deep neural network. You perform computation through this network; for example, in image classification, you represent input images as tensors or matrices and forward them through the layers. A loss function at the end predicts the probability that the image belongs to a certain class, which is then converted into a final label. This process is called forward propagation.

Before we can use the network for prediction, we must train the parameters through backward propagation. In this stage, you take labeled data x and y and minimize the loss against that data. This can be written as the "master equation" of most models:

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

Let's take a deeper look of this equation, which is very important for this course. In this equation, D represents the data and θ represents the model parameters. The loss function L calculates how well the model is performing and can be instantiated as L2 loss, hinge loss, softmax loss, or ranking loss. Softmax is highly popular in language models today. The weight update rule defines how updates (often gradients) are applied to parameters to reach the next iteration until convergence. Popular rules include the Adam optimizer and the more recent Muon optimizer used for Mixture of Experts (MOE). You can also do some fancier stuff like second-order methods, but they are not quite scalable.

2 The Three Pillars of Deep Learning Systems

By following the master equation previously discussed, we can represent almost all popular machine learning models used today. Before diving into specific models, we will reiterate this equation from a compute perspective, focusing on three essential components: data, the model, and compute.

2.1 Data

The first component is data (D). The nature of this data depends on the task; for example, training a language model requires a large volume of tokens, while an image classifier uses pairs of images and labels. For

diffusion models, the data often consists of a significant amount of video content. Modern data modalities can include text, images, audio, and video. Currently, researchers in frontier labs are working on "omni-models," which are designed to take all of these modalities as input and generate them as output using a single, unified model.

2.2 Model

The second component is the model (θ). Several popular model architectures exist today, ranging from slightly older ones, such as Convolutional Neural Networks (CNNs), to highly popular current architectures like Large Language Models (LLMs) and Mixture of Experts (MOE). Additionally, we have recently seen the emergence of new models like diffusion models.

2.3 Compute

The third component is compute, which refers to the process of fitting data into a model to perform training or inference. This computation occurs on various hardware devices and processors. While this work primarily happened on CPUs twenty years ago, it now mostly takes place on accelerators.

A Graphics Processing Unit (GPU) is the most common type of accelerator, but other types exist, such as the Tensor Processing Unit (TPU) and the more recent Language Processing Unit (LPU).

We also utilize heterogeneous hardware processors, such as the M1, M2, and M3 chips found in MacBooks. These chips are considered heterogeneous because they feature a unified memory architecture that can be accessed by both CPU and GPU cores, which are built onto a single integrated plate. While other devices like Field Programmable Gate Arrays (FPGAs) are also used in this field, GPUs currently hold the vast majority of the market share. These three core topics—data, models, and compute—form the foundation of the material we will cover in this course.

3 Core Workloads and Optimization Algorithms

To understand our workloads in deep learning, we must first instantiate the master equation. While history has produced many great models, building a system is significantly harder than building a model. Therefore, this course focuses on the "80-90%" rule: doubling down on the most important workloads that solve the majority of machine learning problems.

3.1 Top Three Model Architectures

We will focus on three primary model architectures:

1. **Convolutional Neural Networks (CNN):** Although their use for basic image classification has evolved, CNNs remain highly relevant. They are a core component of Variational Auto-Encoders (VAEs), which are essential building blocks in diffusion models for generating image and video representations.
2. **Transformers:** This is arguably the most important component in modern AI, serving as the backbone for most current language and vision models.
3. **Mixture of Experts (MOE):** This architecture is used to significantly increase the capacity of language models without a proportional increase in compute cost.

3.2 Optimization Algorithms

While optimization can be categorized into first-order and second-order methods, second-order methods are currently not scalable given the modern hardware landscape. Consequently, we will focus on first-order, gradient-based optimization.

- **Adam Optimizer:** This is the standard optimizer used for almost every modern neural network.
- **Muon Optimizer:** A recent and effective emerging optimizer invented by a UCSD alumnus (now at OpenAI). It has proven particularly effective for training MOE models.

4 Distinguishing Models, Architectures, and Building Blocks

It is important to clarify terminology that is often used interchangeably but has distinct meanings in a systems context.

- **Model vs. Building Block:** A Large Language Model (LLM) is a *family* of models (e.g., Llama, GPT, Claude) composed of stacked layers. In contrast, a **Transformer** is a *building block* that can be used in various ways to stack a model, whether it be a language model or a diffusion model.
- **Diffusion:** Technically, diffusion defines a *learning process*. A "Diffusion Model" refers to the model family, but its building blocks can vary. For instance, the original Stable Diffusion used **U-Net** (based on convolutions) as its building block, whereas the latest versions, such as Diffusion Transformers (DiT), use **Transformers** as the building block. Diffusion model refers to a family of models, and different diffusion models can have different building blocks.

By understanding the compute characteristics of these building blocks, we can reason about the performance and requirements of any model composed of them, regardless of whether it is used for language, images, or video.

In this course, we focus on three primary building blocks: convolutional layers, transformers, and Mixture of Experts (MoE). To define the necessary system building abstractions, we must identify the most important compute units within these blocks. If you find these concepts challenging, it is highly recommended to review foundational deep learning material.

5 Review of CNNs

5.1 Introduction to Convolutional Neural Networks (CNN)

CNNs were first introduced to address computer vision tasks such as like classification, retrieval, detection, segmentation, self-driving, and image generation. At their core, CNNs utilize filters (or kernels) containing learnable parameters. These filters can be 1D, 2D, or 3D.

The convolution process involves sliding a small filter across the input image—moving from left to right and top to bottom—performing a computation at each step to produce a representation of the entire image. As multiple layers are stacked, the initial filters learn low-level features, while deeper layers capture high-level semantic features, eventually enabling tasks like object detection or classification.

5.2 Key CNN Architectures

In the history of CNNs, three models stand out as particularly influential:

- **AlexNet:** This landmark paper by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton is widely considered the starting point of the deep learning era.
- **ResNet:** Developed by Kaiming He, ResNet is arguably the most adopted architecture in computer vision today.
- **U-Net:** This architecture served as the key enabler for the first generation of diffusion models used in image generation.

5.3 Compute Components in CNNs

From a systems perspective, we must identify the specific operators that dominate CNN computation. If we want to optimize CNN performance, these are the areas that require the most attention:

- **Convolution Operator (CONV):** The primary operator that slides a filter across an input. Variations include Conv1D, Conv2D, and Conv3D.
- **Matrix Multiplication (GEMM/MatMul):** Many architectures include linear layers to transform convolved features into different levels of representation. These layers are the primary way parameters (capacity) are introduced into the model.
- **Softmax:** Used at the final stage (typically with a cross-entropy loss) to classify images.
- **Element-wise Operations:** These occur between major layers and include:
  - Addition and Subtraction (residual connections).
  - Non-linear activation functions (e.g., ReLU).
  - Pooling and Normalization (e.g., Batch Normalization).

6 Review of RNNs

Before we transition to Transformers, we must discuss the architecture that dominated the field for years and received significant investment in both compute and research: the Recurrent Neural Network (RNN).

6.1 Concepts and Task Alignment

RNNs are primarily designed to model sequences. In a language modeling context, a sentence is treated as a sequence where each token represents a specific position. The core strength of an RNN is its ability to handle an arbitrary number of inputs and predict an arbitrary number of outputs. This allows for several configurations:

- **One-to-One:** Standard classification.
- **One-to-Many:** For example, image captioning.

- **Many-to-One:** Sentiment analysis.
- **Many-to-Many:** Machine translation or video labeling.

The fundamental idea behind an RNN is the maintenance of an internal hidden state that is updated as each element of the sequence is processed. If we "unroll" the network across timestamps (representing the sequence length), we can visualize the compute flow: at each step, the model takes a new input, updates its internal state based on the previous state, and makes a prediction.

6.2 Top Three RNN Models

While many variants exist, three models have historically been the most influential:

1. **Bidirectional RNN:** This model processes the sequence from both left-to-right and right-to-left, updating internal states to capture context from both directions.
2. **Long Short-Term Memory (LSTM):** A highly famous architecture that saw a massive surge in research between 2015 and 2018. It was specifically designed to handle long-term dependencies.
3. **Gated Recurrent Unit (GRU):** A more recent and computationally efficient variant of the LSTM that was also heavily researched during the same period.

6.3 A Note on the History of LSTM

As a bit of trivia, the invention of the LSTM is often a topic of discussion in the community.

6.4 The Transition to Transformers

Although RNNs are not yet obsolete and are occasionally revisited today for specific low-compute layers in language models, they have largely been replaced by Transformers. This transition occurred because RNNs possess severe structural problems that limit their effectiveness for modern large-scale tasks.

7 Review of Transformers

The primary reason ChatGPT was not built using Recurrent Neural Networks (RNNs) is due to two major drawbacks of the RNN architecture. The first issue is the "forgetting" problem, or vanishing gradient. In an RNN, the hidden state is updated through a series of linear and non-linear operations. Mathematically, this is equivalent to an equation where the hidden state is multiplied by a value smaller than one at each step. As the sequence length increases—reaching up to a million tokens in modern context windows like Claude—the signal eventually diminishes to zero, making it extremely difficult to train on long sequences.

The second problem is that RNNs are not scalable because they are difficult to parallelize. Since the next state update depends entirely on the result of the previous state, the computation must be performed sequentially.

In 2017 and 2018, Google researchers addressed these problems with the paper "Attention Is All You Need," introducing the Transformer. The core idea is the attention mechanism, which treats each position's representation as a query to access and incorporate information from a set of key-values. This mechanism is capable of modeling long sequences without the forgetting issue and is easily parallelized during training since the entire sequence is visible at once.

7.1 Transformer Variants and Top Models

Transformers were originally introduced with two variants: the encoder and the decoder. These correspond to different popular models in Natural Language Processing (NLP). BERT is composed of many encoders used for tasks like feature extraction and prediction, while GPT and most recent Large Language Models (LLMs) are composed of decoders for language modeling.

The top three model families using the Transformer as a building block are:

1. **BERT:** Highly popular between 2017 and 2020 for various linguistic tasks.
2. **GPT/LLMs:** Composed of transformer decoders.
3. **DiTs (Diffusion Transformers):** The most powerful modern diffusion models, which use transformers as their building block rather than the traditional UNet. In most current image and video diffusion models, the architecture is typically encoder-based.

The Transformer became the dominant architecture because, as Ilya Sutskever noted in early talks, if you have a large dataset and a very large neural network, success is virtually guaranteed. The Transformer's friendliness to parallelization makes it worth investing massive amounts of compute, leading to the emerging capabilities we see today.

7.2 Compute Components in Transformers

To make Transformers run fast, we need to break them down into their primary compute components. A Transformer block is essentially composed of Attention, a Multi-Layer Perceptron (MLP), and additional layers.

7.2.1 Attention Compute

From a systems perspective, Attention consists of three core operations:

- **Matrix Multiplication (MatMul):** Required for QKV (Query, Key, Value) projections and output projections.
- **Softmax:** Applied to calculate the attention scores.
- **Normalization:** Typically implemented as LayerNorm between operations.

7.2.2 MLP Compute

The MLP (Multi-Layer Perceptron) is simpler and consists of:

- **MatMul:** To project the representations.
- **Non-linear Activation Functions:** Such as ReLU, GeLU, or SiLU.

Additional components include residual connections and various non-linear functions (e.g., GeLU, ReLU, LayerNorm) depending on the specific Transformer variant. The main theme of this course will be optimizing these specific operators—MatMul and Softmax—to minimize execution time and make the entire network run as fast as possible.

8 Review of Mixture of Experts (MoE)

The Mixture of Experts (MoE) architecture was introduced to Large Language Models (LLMs) to significantly increase model capacity. The core idea is simple: instead of using a single "expert" for every computation, the model "votes" or selects from many experts to achieve a result that is superior to using just one.

Most state-of-the-art models today—including GPT-4, Grok, and many open-source models released by companies like DeepSeek—are literally all MoEs. The primary difference between an MoE and a dense model lies in the MLP (Multi-Layer Perceptron) layer. In an MoE, the standard MLP is replaced by a set of experts, where each expert is itself an MLP. The number of experts can be quite high; for example, in DeepSeek-V3, there are as many as 256 experts.

8.1 The "Free Lunch" of MoE

Despite having hundreds of experts, during any single forward pass or prediction, the model only chooses a small subset—typically one or two experts—to execute. This selection is handled by a learnable router that predicts which expert is most suitable for a specific token.

The advantages of this approach include:

- **Increased Parameters:** You introduce a massive number of parameters into the model, making it significantly more powerful if trained correctly.
- **Constant Compute Cost:** Because only a fixed number of experts are activated per token, the inference and training compute costs remain similar to those of a much smaller dense model.

This is essentially a "free lunch" in scaling: you obtain a much more powerful model with more parameters while keeping the active compute cost almost the same. That's how people basically continue to scale the capability of models.

8.2 System and Engineering Challenges

From a pure compute perspective, MoE does not introduce many novel components. The experts are standard MLPs, so nothing changes. The only differenceis you have a router, and the router is implemented using a Softmax function (which is essentially Matrix Multiplication plus a non-linear operation). However, MoE introduces significant engineering and system challenges during real-world deployment.

The difficulty arises from the dynamic nature of model execution. In a dense model, the execution path is predictable. In an MoE, the specific experts activated for each token are only known at runtime. This unpredictability makes it very hard to optimize system performance, load balancing, and memory management.

9 Course Objective and the Global Picture of MLSys

To summarize our mini machine learning course, we have analyzed several model families by breaking them down into their primary compute components. By looking at these building blocks, we can see a clear pattern in the operations required:

- **CNNs:** Convolution (Conv), Matrix Multiplication (MatMul), Softmax, ReLU, and Batch Normalization.
- **RNNs:** MatMul, Sigmoid, and Tanh (simple non-linear operations).
- **Transformers:** MatMul, Softmax, GeLU, and LayerNorm.
- **MoE:** MatMul and Softmax (used in the router).

If we count the frequency of these operators across all major architectures, Matrix Multiplication (MatMul) appears in every single one, while Softmax appears in three out of the four. Most other operations appear only once or twice.

9.1 MatMul is All You Need

This observation defines the core mission of this course. Because MatMul is the most pervasive and computationally expensive unit, we will spend approximately half of the quarter optimizing it. We will dive deep into how to implement MatMul across different devices, how to make it faster, and how to execute it efficiently using different precisions.

The remaining half of the course will focus on the second most critical operator: Softmax. This involves discussing GPU implementations like FlashAttention and other advanced attention algorithms. In my opinion, Machine Learning Systems (MLSys) essentially equals "MatMul optimization." If you can master these fundamental primitives, you can make an entire neural network fast.

9.2 The Global Picture of ML Systems

Let's put these components together to see the global picture of the course:

1. **Data:** Regardless of the modality (text, video, audio), all data is eventually represented as tensors.
2. **Model:** We have identified the primitive operators—primarily MatMul and Softmax—that compose our building blocks.
3. **Compute:** To build a functional system, we need a way to express the model computation by connecting these primitives. This is our next topic: developing a representation that can map these primitives into a coherent compute flow.

Finally, we combine the data, the connected primitives, and their optimized implementations onto hardware like GPUs to maximize performance.

Naturally, we are now going to dive deeper into the representation layer. Since we know the primitives that compose our blocks, we need a canonical orientation to connect them. In the next section, we will introduce the primary representation used in machine learning systems: the Data Flow Graph.

10 Computational Dataflow Graph

As SQL provides a declarative way to express queries that a query planner then optimizes and executes on a relational database, ML frameworks need a similar abstraction. A computational dataflow graph is a way of representing an ML flow computation.

We can view it as a directed graph with two components:

- Nodes represent both the operations being performed (like matmul, ReLU, softmax) and the input/output tensors of those operations. Some nodes are simply input constants or parameters (like weights) rather than compute operations.
- Edges represent data dependencies — they show which operation's output flows as input into which other operation. The direction of an edge indicates the dependency order.

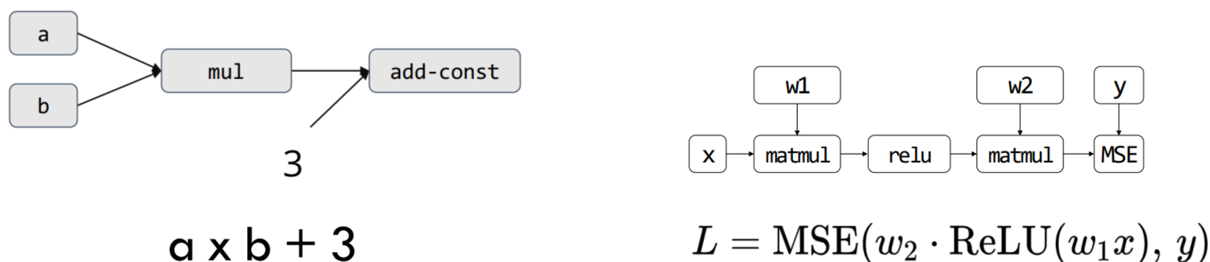

Figure 1: Dataflow Graph Example

Looking at Figure 1 as a simple example, we notice that Nodes x, y are fixed input data or labels, Nodes w_1, w_2 act as weights, Nodes *relu, matmul, MSE* are operations.

Therefore, we can easily model a neural network as a CDG by making layers as nodes and the flow of data as edges.

10.1 TensorFlow Case Study

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
```

Forward Computation
Declaration

Figure 2: Tensorflow Case Study

In TensorFlow, forward computation(variables/weights), loss function, autodiff, and SGD optimization rule are all declared before the real execution.

```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

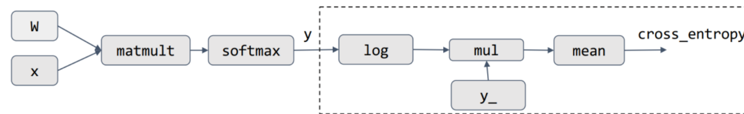

Figure 3: Tensorflow Building Graph

The above figure declares a cross entropy loss: you add nodes to the graph that compute the losses. These nodes connect to the output of the forward pass.

When you write TensorFlow code like **tf.matmul(x, w)**, you're creating a node in a graph object. Each line adds more nodes and edges. The session runtime then takes this complete graph, applies optimizations, allocates memory, schedules operations, and executes everything.

We call this as **Static Dataflow Graph**.

10.2 Benefits and Drawback

Because the framework can "see" the whole computation at once, it can perform some graph-level optimizations and enabling more parallelization(since the scheduler knows which operators are independent).

However Static Dataflow Graph is difficult in modeling dynamic computations like variable-length sequences and data-dependent computation(like tree-structured models), hard to debug(since the code you write is intermediate graph representation).

PyTorch uses a define-and-run approach — each line of Python actually executes the computation right away. A dataflow graph(used for tracking autodiff) is built dynamically as operations execute rather than being declared upfront.

A graph is created on the fly

W_h h W_x x

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

Figure 4: PyTorch Example

This **Imperative** style is more widely used with the cost of increased computation overhead.

11 Introduction to Symbolic and Imperative Programming

In machine learning systems, one important distinction is between symbolic and imperative programming. This distinction is closely related to the difference between define-then-run and define-and-run execution styles. The symbolic approach emphasizes building a computation graph before execution, while the imperative approach emphasizes executing operations immediately and observing results step by step.

12 Symbolic Programming

Symbolic programming is a programming paradigm in which a program first declares symbols and builds them into a larger computation before carrying out execution. In this style, the programmer specifies the structure of the computation in advance, and the system later executes the resulting graph. Some examples include C++, Tensorflow etc.

12.1 Key Idea

The central idea is to define the program through symbolic components first, then execute the whole computation afterward. In machine learning, this often means constructing a static computational dataflow graph that represents operators, tensors, and dependencies before the actual run begins.

12.2 Advantages

- Easier to optimize for distributed execution, batching, parallelization etc.
- Well-suited for deployment scenarios where the computation graph is fixed in advance.

12.3 Disadvantages

- Can feel counterintuitive because the user must define symbols before doing the actual work.
- Harder to debug, since execution is separated from declaration.
- Less flexible when the program contains dynamic behavior.

12.4 Example: Symbolic Style

The following TensorFlow-style snippet illustrates the symbolic approach, where the model structure is defined first:

Listing 1: Symbolic programming example

```
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
```

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

13 Imperative Programming

Imperative programming is a paradigm in which the code directly specifies the steps needed to solve a problem. Each line is executed in sequence, and intermediate values can be observed immediately.

13.1 Key Idea

The core idea is to define and execute operations as the program runs. In machine learning systems, this makes it natural to inspect tensors, print intermediate outputs, and debug interactively. Python and PyTorch are examples of imperative frameworks.

13.2 Advantages

- Very flexible and natural for experimentation.
- Easier to program and debug because each line can be evaluated directly.

13.3 Disadvantages

- Usually less efficient than symbolic execution.
- Harder for the framework to optimize aggressively.

13.4 Example: Imperative Style

The following PyTorch-style snippet illustrates imperative execution:

Listing 2: Imperative programming example

```
x = torch.Tensor([3])
y = torch.Tensor([2])

z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

14 Define-Then-Run vs. Define-And-Run

The symbolic and imperative styles are often summarized through two execution models:

- **Define-then-run:** first construct the computation graph, then execute it. This aligns with symbolic frameworks.

- **Define-and-run:** define operations and execute them immediately. This aligns with imperative frameworks.

Symbolic systems are attractive when optimization and large-scale execution matter most, while imperative systems are attractive when usability, experimentation, and debugging matter most.

15 Just-in-Time (JIT) Compilation

Just-in-time (JIT) compilation aims to combine the flexibility of imperative, define-and-run programming during development with the efficiency of symbolic, define-then-run execution during deployment. The core idea is to let researchers and developers write code in a dynamic and debuggable style first, then convert that code into a more static and optimized representation once the model behavior is finalized.

15.1 Development vs. Deployment

There is a natural tradeoff between imperative and symbolic execution styles. During development, we usually prefer define-and-run execution because it is flexible, easy to debug, and convenient for experimentation. This makes it easier to try different model architectures and algorithms, inspect intermediate values with print statements or interactive tools, and quickly modify code as ideas evolve.

During deployment, however, the priorities change. Once the model is finalized, we usually prefer define-then-run execution because it enables stronger optimization and often improves runtime performance. In this setting, the code no longer needs to be modified interactively, and the system can benefit from graph-level compilation and execution optimizations.

JIT compilation attempts to get the best of both worlds by preserving an imperative programming interface during development while compiling the resulting computation into a static graph when possible. In PyTorch, this idea is illustrated by `torch.compile()`: a developer writes ordinary dynamic PyTorch code first, and once the code is stable, `torch.compile()` can be applied so that the framework converts the dynamic computation into a more static, optimized graph for execution.

15.2 How JIT Relates to Symbolic and Imperative Programming

JIT compilation reflects the broader convergence of symbolic and imperative frameworks. Symbolic systems have become more flexible, while imperative systems have adopted more compilation and graph-based optimization techniques. In this sense, JIT serves as a bridge between the two programming styles.

Conceptually:

- imperative programming provides usability and flexibility,
- symbolic execution provides optimization opportunities,
- JIT compilation attempts to translate imperative programs into symbolic-style execution graphs when conditions allow.

15.3 Benefits of JIT Compilation

The main benefits of JIT compilation are:

- retaining an intuitive and pythonic development workflow,
- improving performance during deployment,
- enabling graph-level optimizations once the computation becomes static enough,
- reducing the gap between research code and production code.

This makes JIT especially attractive in machine learning systems, where developers often want both fast iteration and efficient large-scale execution.

Scribe Responsibilities

- **Yiming Zhao:** Section 1 - Section 9
- **Xinle Yu:** Section 10
- **Ohm Rishabh Venkatachalam:** Section 11 - Section 15
- **Chenrui Liu:** Proofreading