CSE-234: Data Systems for Machine Learning, Winter 2025

2: Basics: Modern DL, computational graph, autodiff, frameworks

Lecturer: Hao Zhang

Scribe: Nicholas King, Chun-Mao (Michael) Lai, Jaedyn Chang, Vincent Hsieh, Keren Gutman, Zhengan Cheng, Zerlina Lai, Ramsundar Tanikella, Akanksha Pandey, Luke Mitbo

1 Background: Deep Learning Computing

When we are building systems, we need to first understand our workload, which primarily consists of machine learning and deep learning computations. Deep learning involves stacking many neural network layers to compose a large and effective model. For example, in image classification, we perform the following steps:

1. Define the layers of the neural network
2. Perform specific computations on each layer
3. Forward the images through the network
4. Obtain the prediction for the given image

A neural network will not work unless it is trained; to train it, first predict with forward progagation, compute the gradients of the loss, and finally perform backward propagation to update the parameters based on the computed gradients.

Equation:

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

- $\theta^{(t)}$: Parameters of the model at iteration t
- $\mathcal{L}(\theta, D)$: Loss function, which depends on the problem, e.g., L2 loss, softmax loss
- $\nabla_{\mathcal{L}}(\theta^{(t)}, D^{(t)})$: Gradient of the loss function $\mathcal{L}$ with respect to the parameters $\theta^{(t)}$ given the training data $D^{(t)}$
- $D^{(t)}$: Training data (e.g., images, text, video, audio)
- T: Total number of iterations until convergence

In deep learning, the objective is to optimize the set of parameters θ that will minimize the loss, which is achieved by continuously feeding data into the neural network, calculating gradients, and applying parameter updates until the parameters converges with gradients as close to 0 as possible. The optimization method is a procedure used to find a set of parameters that minimize the loss. Common optimization algorithms include SGD, Adam, Newton, etc.

Three Most Important Components

- **Data:** Images, text, audio, tables, etc.
- **Model:** CNNs, RNNs, Transformers, Mixture of Experts (MoEs), etc.
- **Compute:** CPUs, GPUs/TPUs/LPUs, M1/M2/M3/M4, FPGA, etc.

2 Understand our Workloads: Deep Learning

In system building, it is often impractical to support all possible models. Instead, we focus on identifying the most important workloads that should solve about 80% of the problem. System building is the process of uncovering the most critical factors that define these workloads.

Most Important Models: CNNs (Convolutional Neural Networks), RNNs (Recurrent Neural Networks), Transformers, MoEs (Mixture-of-Experts).

Most Important Algorithms: SGD (Stochastic Gradient Descent) and its variants, such as Adam.

The key to system building is identifying the most important X in Y ($X \subset Y$) to define the system's abstractions, such as $X =$ ResNet and $Y =$ CNNs.

2.1 Convolutional Neural Networks (CNN)

2.1.1 Applications

CNNs enable several interesting applications:

1. **Classification**. One example is categorizing an image as "human" or "bicycle."
2. **Retrieval**. For example, retrieving images similar to a query image.
3. **Detection**. This application is for both detecting an object and its' position.
4. **Segmentation**. For instance, assigning labels to regions of an image and segmenting it.
5. **Self-driving**. This uses detection and segmentation to recognize objects and elements on and off the road.
6. **Synthesis**, as in synthesizing and generating visual data.

As you can see, CNNs are used mainly in computer vision problems. The reason for that is because their design is inherently good at finding local spatial relationships in images as a result of their focus on small patches of an image. In other words, they are good at detecting relationships in a neighborhood of pixels due to the sliding window technique they integrate. See design details below for more.

2.1.2 What are the basics of a CNN's design?

At the basis of a CNN is the convolution layer, which takes a 2D input image and produces a slightly smaller 2D image as the output[1]. This transformation occurs through the use of convolution filters, often very small

[1]The output is smaller because the filter cannot process pixels that are near the edges without exceeding the boundaries of the image, so the outermost rows and columns are excluded.

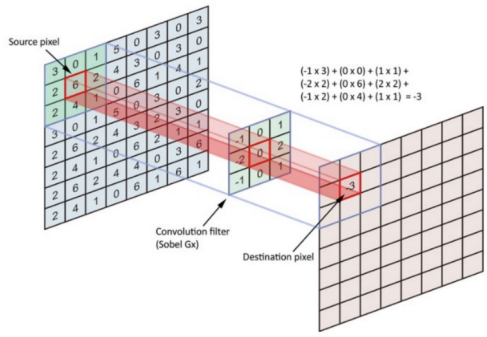

Figure 1: Sliding example from lecture

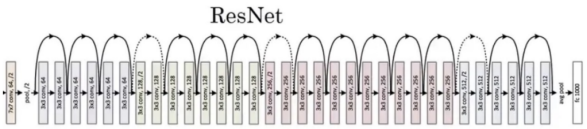

Figure 2: ResNet skip connections

(3x3 or 4x4), which slide over the input image from left to right and top to bottom. Each filter uses the same set of weights over the entire image, which expose specific patterns like edges.

Each movement involves element-wise multiplications and summing the results. The outputs are then written into the output image to represent learned characteristics.

The magic of CNNs is that if we stack their convolution layers many times, we start to learn more complex features. At the very lowest level, we learn edge and color. As we stack, we begin to get higher-level features such as texture, shapes, and eventually high-level semantic representations like distinguishing between "human" and "bicycle."

2.1.3 What are the top 3 models for CNNs?

1. **AlexNet**. This was the breakthrough model for CNNs.
2. **ResNet**. This model brought CNNs up to scale by introducing skip connections, which address the vanishing gradient problem.
3. **U-Net**. This was the backbone for stable diffusion. Aside from that, it's widely adopted in medical image analysis for segmentation.

2.1.4 What are the most important components for CNNs? What do we care about for system building?

1. **Convolutional operations.** Specifically, Conv1d (for sequential data such as audio signals), Conv2d (for images), Conv3d (for videos), and especially, Conv2d with a 3x3 filter.
2. **Matrix multiplication.** This is included because skip connections, which are similarly structured to multilayer perceptrons, use them, and skip connections are extremely important for scaling CNNs.

3. **Softmax.** This is applied at the output layer to predict the label with probabilities.
4. **Element-wise operations.** Examples include ReLU, addition, subtraction, pooling (to reduce dimensions), and batch normalization.

2.2 Recurrent Neural Networks (RNN)

2.3 Introduction

RNNs possess a unique ability to model both one-to-many and many-to-one relationships, such as mapping a sequence of inputs to a single output label. They use sequential data and this data can be dynamic, allowing for an arbitrary number of inputs and outputs. Additionally, any neural network can be made recurrent by maintaining its internal state and passing it through both the inputs and outputs.

2.3.1 Applications

RNNs enable several interesting applications:

1. **Natural Language Processing**. RNNs can predict the next word in a sentence or determine the sentiment of input text.
2. **Time-Series Analysis**. RNNs can analyze time-based data, such as weather or stock market trends.
3. **Speech and Audio Processing**. RNNs can generate speech or convert speech to text.
4. **Anomaly Detection**. RNNs can identify unusual patterns in data which is useful for fraud detection and systems monitoring.

2.3.2 Idea behind RNNs

The key idea behind RNNs is that they contain an internal state in a which gets updated as input sequences are processed. The internal state is said to act like memory, allowing the model to remember previous data points within a sequence so that it can make informed predictions. This architecture makes RNNs thrive in settings in which the data is sequential or time-based.

2.3.3 Unrolling the Computation

Theoretically, any neural network could be made into a RNN. Figure 3 shows how the building blocks of the model could be embedded with by any kind of neural network, such as a CNN or MLP.

2.3.4 Top Three Models

1. **Bidirectional RNN**. Makes computations in both directions, left-to-right and right-to-left.
2. **LSTM**. Has been widely adopted in time-series analysis due to its ability to remember or forget information as required.
3. **GRU**. A powerful, simplified version of LSTM.

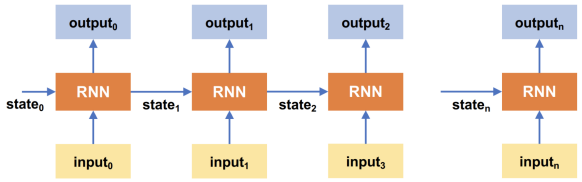

Figure 3: Unrolling the RNN Architecture

2.3.5 What are the most important components in RNNs?

1. **Matrix multiplication**. Required for updating the internal states and for forward propagation.
2. **Element-wise nonlinear activations**. ReLU, tanh, and sigmoid functions are frequently used in RNNS.

2.3.6 Why was ChatGPT not built using RNNs?

RNNs have two major limitations: they tend to forget information over time and suffer from a lack of parallelizability.

RNNs tend to forget information within long input sequences. This causes them to struggle with long sequences of text since they fail to recall previous information if it appeared far before. In many cases, data points at the beginning of a sequence remain important across the entirety of the input, and RNNs simply cannot maintain this importance.

Furthermore, the computations in RNNs cannot be parallelized due to their sequential nature, where a state relies on all previous states to be computed before computing the next step. This makes training RNNs very time-consuming. This limitation of being non-parallelizable is overcome in attention and transformer models.

2.4 Attention & Transformer

Attention

Researchers at Google proposed the influential paper *Attention Is All You Need* to fix the problems in RNNs.

Key Idea

Attention treats each position's representation as a query to access and incorporate information from a set of values. Since each position operates independently, there are no sequential dependencies, allowing for parallel processing.

In fig. 4, each hidden position in a given layer is computed by treating its current representation as a query, which then attends to the keys and values from all positions in the previous layer. In this way, all the positions with label "1" could be computed in parallel from layer 0.

Because of its sequential independency, the attention mechanism is perfect for GPUs to parallelize. In other words, Attention is **massively parallelizable**:

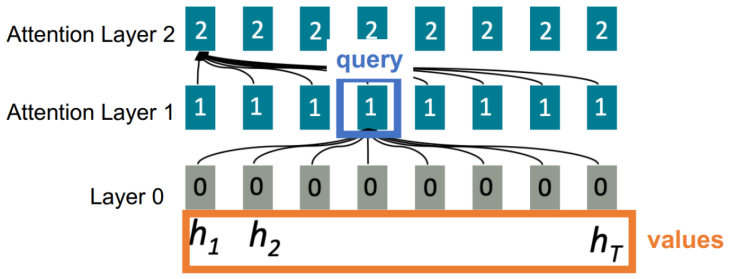

Figure 4: Attention Mechanism

- The number of unparallelizable operations does not increase sequence length. Hence, attention can handle arbitrarily large sequences without suffering from sequential dependency.
- In RNNs, you have to compute the hidden states sequentially, which is the reason why Attention becomes dominant and beats RNNs.

We will explore attention and transformers in more detail in upcoming lectures, covering topics such as self-attention, masked attention, and multi-head attention.

Transformer

With the invention of attention mechanisms, we can construct the transformer model.

As shown in fig. 5, the inputs go through attentions, LayerNorms - normalization operation element-wise, MLPs, and finally output some value.

The transformer model has two types:

- **Encoder**: The most famous example of an encoder transformer is BERT.
- **Decoder**: The most famous example of a decoder transformer is GPT.

Top 3 Models for Transformers

- BERT
- GPT/LLm's
- DiT (Diffusion). (DiT is the reason why U-Net is no longer used in diffusion models.)

The Chosen One

Today, everyone uses Transformer models as the default backbone for all types of tasks.

Ilya Sutskever wrote the *paper Sequence to Sequence Learning with Neural Networks* a decade ago, which is the foundation and first attempt at training modern GPT-like models. He also has a very famous slide with 3 sentences in a talk:

- If you have a large big dataset

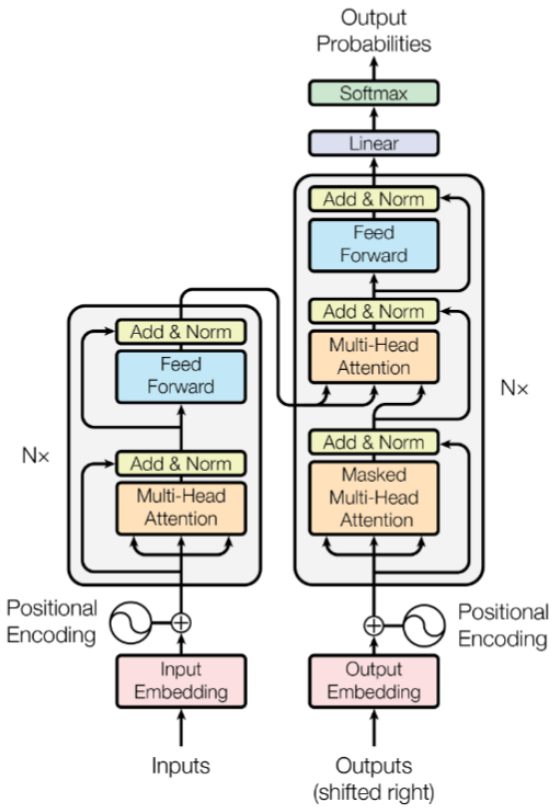

Figure 5: The Transformer Architecture

- And you train a very big neural network
- Then success is guaranteed!

Most Important Components of Transformer Models

$$\textbf{Transformers} = \textbf{Attention} + \textbf{MLP} + \textbf{Something Else}$$

- **Attention**: Matmul, Softmax, Normalization
- **MLP**: Matmul
- **Something Else**: LayerNorm, GELU, etc.

2.5 Mixture of Experts (MoE)

MoE is typically used used together with one of the previous 3 types of models. MoE can be integrated with any one of them.

Key Idea

MoE replicates certain components of a single model into multiple experts. The idea is to combine the predictions from these experts, similar to a voting ensemble, with the hope that the collective decision of many experts will outperform a single expert.

In fact, many of the latest LLMs are built on the MoE architecture, including Grok, Mixtral, GPT-4 (unconfirmed), and DeepSeek-v3.

Novel Component

The only novel component in MoE is something called a **router**. Because the model is replicated across multiple experts, you need to embed a router to determine which expert will process each input.

The router is constituted by the following operations:

- Matmul
- Softmax

Given an input, a router classifies it into the number of classes corresponding to certain experts, then takes the top 2 values, and outputs it.

After-class Question: Why does the router make it hard? (This will be addresssed in later lectures).

2.6 Summary

We can summarize the 4 models from a computation perspective in table 1.

Model	Matmul	Softmax	Others
CNNs	√	√	Conv, ReLU, BatchNorm
RNNs	√	-	Sigmoid, Tanh
Transformers	√	√	GeLU, LayerNorm
MoE	√	√	-

Table 1: Rundown from a compute perspective

Or we can count the frequency of each operator's appearance:

- Matmul: 4 times
- Softmax: 3 times
- Others...

In a nutshell, the true name of this class is Matmul (plus Softmax) are all you need.

MLSys ≈ Matmul Sys

We are going to scale these two operations from CPU to GPU, and from single devices to clusters. Most of our focus will be on finding ways to speed up Matmul and Softmax, or how to parallelize them across multiple GPUs.

The high-level picture is as follows:

- We have **data** x_i for $i = 1$ to n, represented as tensors in memory.
- We have our **model**, which consists of mathematical primitives (primarily `matmul`). The model represents the computation through these primitives.
- We have **compute**, which ensures the programs run on clusters of different types of hardware.

3 Dataflow Graph Representation

How can we develop a representation that expresses the model?

A computational dataflow graph (fig. 6) is a directed graph that represents the flow of computation in an ML pipeline. It represents how data flows between different operations and shows dependencies.

Dataflow graphs are useful because they provide a clear and structured way of modeling computations and system design. They provide a means of abstraction to represent complex ML pipelines and algorithms.

Although there are countless different ML models and architectures, dataflow graphs present a common vocabulary of mathematical primitives, basic operations like addition, multiplication, activation functions, etc., that can be used to represent the majority of these architectures.

What does a model consist of in terms of mathematical primitives?

It includes the model architecture (composed of mathematical operations), the objective function, the optimizer, and the data used for training.

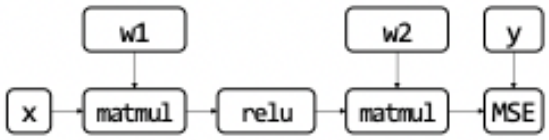

Figure 6: A simple dataflow graph example. Note: in this example, the MSE calculation has been represented as a single node. In actuality, MSE contains multiple operations that should be represented as separate nodes.

3.1 Computational Dataflow Graph

The key components of a computational dataflow graph are:

1. **Nodes:** Nodes represent an operator, which is a computational operation or task. Nodes can also represent the output tensor of the operator. This means the computation not only defines a mathematical operation but also produces an output result (a tensor). Alternatively, a node can also represent a constant input tensor. These are fixed tensor values such as hyperparameters or input data.

   For example, the `matmul` node in fig. 6 represents both the operation of matrix multiplication and the output of the operation. That is why there is an arrow from `matmul` to `relu`.

2. **Edges:** Edges represent data dependencies, which is the way data flows between nodes, showing how the output of one operation becomes the next operation's input. The direction of the edge indicates the dependency order.

Figure 7

Let's look at a simple example:

Figure 7 is a CDG representing the set of mathematical operations to multiply matrices a and b then add 3. The nodes `a` and `b` represent fixed input data. The edges leading from these nodes to `mul` indicates that `a` and `b` are inputs to the `mul` operation. The edge leading out of `mul` indicates that the operation produces an output, which is then fed into the `add-const` operation along with a constant value of 3.

Applying these concepts on a larger scale: in a neural network, layers can be modeled as nodes, and the flow of data through the layers is represented as the edges. A feedforward neural network can be expressed as a series of connected primitive operations including `matmul`, `relu`, and `softmax`.

Case study: TensorFlow In TensorFlow, the forward computation, loss function, autodiff, and SGD update rule are first declared in code. The real execution is triggered as the last step. Behind the scenes, TF is building a static CDG as it declares each component, which the execution then follows strictly.

The benefits of building a dataflow graph using this method are that abstracting a pipeline or series of computations as a CDG can allow us to capitalize on opportunities for parallelism and optimization, since different branches of the graph can be run in parallel.

However, there are also some drawbacks. It is difficult to model dynamic computations with static dataflow graphs. In static CDGs, the graph is fully defined before execution (as TF does), representing a rigid structure that strictly follows the predefined computations and data flow.

Case study: Pytorch Pytorch implements a solution to the above problem. Instead of pre-defining the CDG before execution, it creates the graph as the forward operations are executed. It then uses the resultant graph during backpropagation. However, this flexibility comes at the cost of increased computational overhead because the graph is recreated every forward pass.

3.2 Symbolic vs. Imperative

3.2.1 Symbolic Programming

Symbolic programming is a programming paradigm in which the program manipulates its functions and components by constructing more complex processes through the combination of simpler logical units and functionalities. In essence, it involves building intricate program components step by step, starting with less complex ones. This approach often requires declaring and defining all symbols before executing any tasks. Figure 8 illustrates an example of symbolic programming, where all symbols are defined prior to performing any operations.

Advantages of Symbolic Programming:

- Symbolic frameworks are often easier to optimize, offering features like distributed processing, paral-

```
# 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]))
```

Figure 8: Symbolic programming

lelization, and batching.

- Symbolic frameworks are typically the most efficient way to write a program, often achieving up to 10 times greater efficiency.

Disadvantages of Symbolic Programming:

- Symbolic programming can be counterintuitive, making debugging more challenging since the code is written by defining all symbols first and gradually developing the program's logic.
- Symbols need to be defined first before proceeding with the actual task making it harder to code and debug, as well as being less flexible

Examples of Symbolic Languages: C, C++, SQL.

3.2.2 Imperative Programming

Imperative programming is a programming paradigm where the code explicitly defines the steps required to solve a problem. In other words, this means the code is defined in a way that justifies each action needed to solve the problem. Figure 9 shows an example of imperative programming style.

```
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

Figure 9: Imperative programming

Advantages of Imperative Programming:

- Imperative programming is very flexible. Each line of code written can be evaluated and checked to ensure it performs the correct action.

- Imperative languages are very easy to program and debug, and promote user readability.

Disadvantages of Imperative Programming:

- Imperative programming languages are less efficient than symbolic programming languages.
- Imperative programming languages are harder to optimize than symbolic programming languages.

Examples of Imperative Languages: Python

Python is a define-then-run programming language, and since TensorFlow is built using Python, it follows the same define-then-run paradigm. However, TensorFlow is considered a symbolic framework, while Python is an imperative language.

How did this happen?

TensorFlow uses Python as its primary interface, but it relies on a domain-specific language (DSL) built on top of Python. In contrast, PyTorch's DSL is more Pythonic than TensorFlow's. Today, these frameworks are converging, with symbolic frameworks becoming more imperative and imperative frameworks incorporating symbolic elements, largely due to Just-In-Time (JIT) compilation, which allows for define-then-run operations during deployment.

Popular pythonic symbolic frameworks include Tensorflow, Caffe2, theano and Caffe. Popular pythonic imperative frameworks include Pytorch, DyNet (written in C++ with Python bindings) and Chainer. We see that Python supports popular ML frameworks in both symbolic and imperative classes.

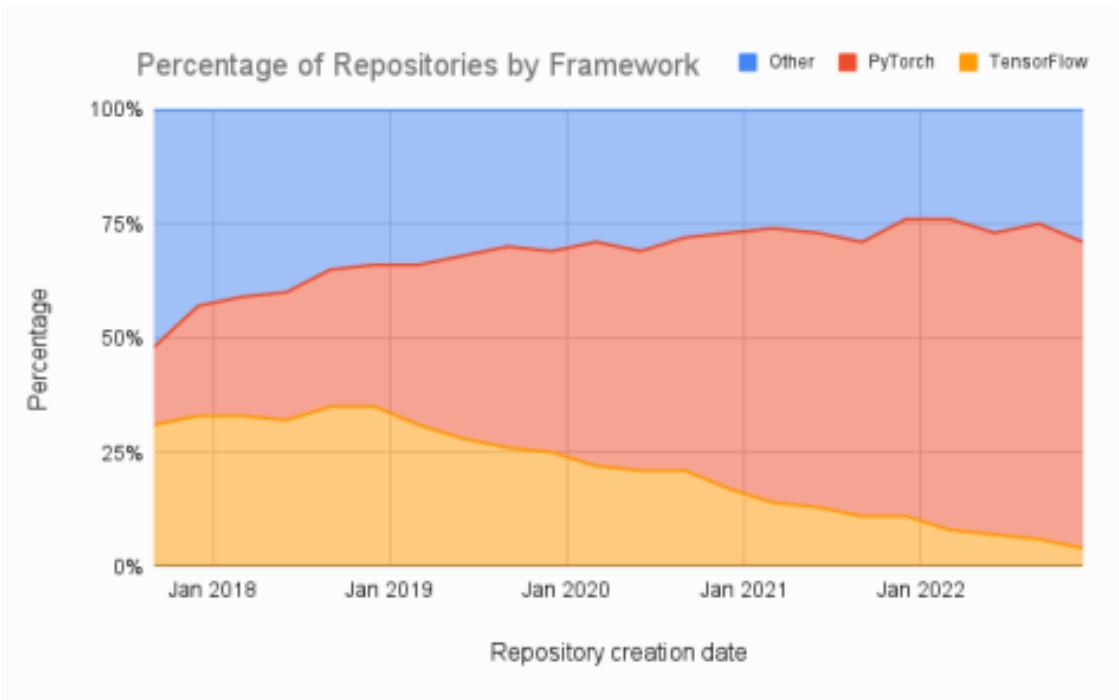

Figure 10: Framework statistics

As an interesting observation in this discussion on frameworks, it is noteworthy to notice that even though Pytorch is one of the later ML frameworks to come up, it wins the "research market" in terms of number of

Pytorch repositories (refer figure 10) and number of pytorch models on huggingface (refer figure 11). Tensorflow on the other hand originated earlier than Pytorch but is lagging as the de-facto choice of framework in research articles behind Pytorch. This observation can be mainly attributed to the fact that Pytorch is imperative making it easier to understand and code than Tensorflow which is generally symbolic. (As a heads up, significant changes in TF1 API to TF2 API may have also accelerated this change.) Though for industry usage, Tensorflow is still generally the most popular framework.

Another interesting observation is even though Tensorflow is a *define then run* framework (making it ideal for deployment of models, thus more industry usage!), it uses Python as its primary interface langauge, which is infact a *define and run* programming langauge (ideal for model developers, thus its popularity!).

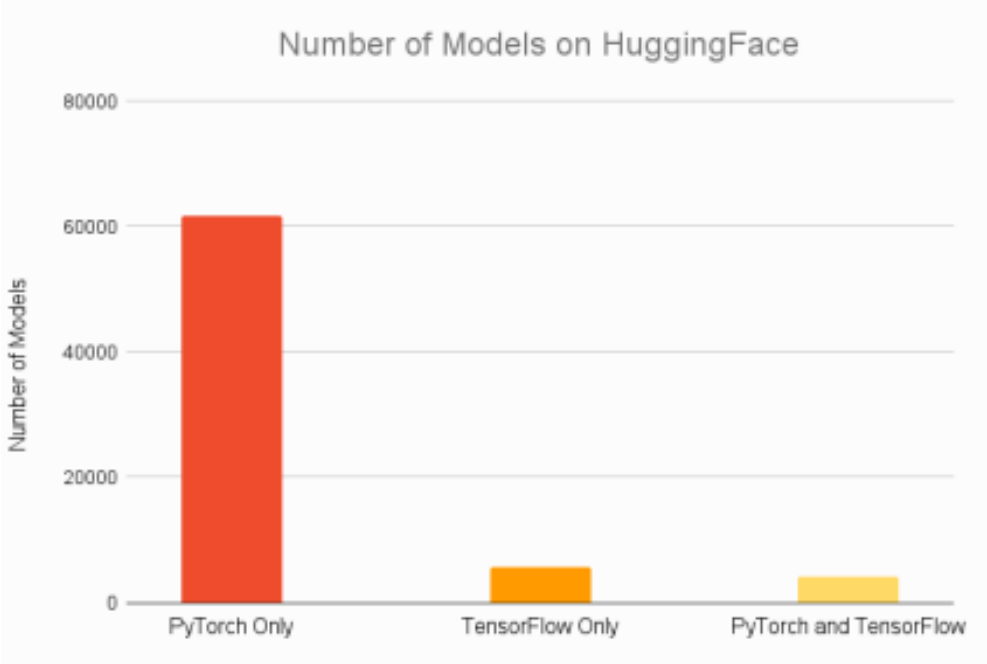

Figure 11: Framework statistics - huggingface

3.3 Just-in-time (JIT) Compilation

JIT compilation combines the flexibility of define-and-run during development with the efficiency of define-then-run during deployment. The challenge with JIT is finding the right balance between these two approaches to achieve optimal performance.

3.3.1 Development Mode: Define-and-Run

During development, we want a flexible environment to:

- Experiment with models and algorithms.
- Easily debug using print statements and interactive tools.

3.3.2 Deployment Mode: Define-then-Run

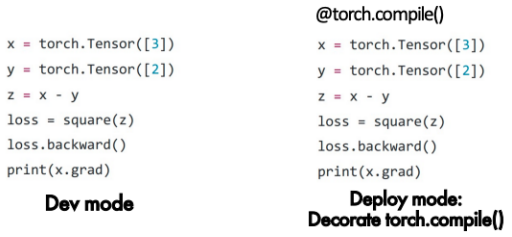

Figure 12: The conversion of PyTorch code between development mode and deploy mode using `torch.compile()`.

Once the model is finalized, we prioritize performance:

- The code should run efficiently without interactive modifications.
- The decorator `@torch.compile()` converts the dynamic PyTorch code into a static, optimized graph.
- Static graph resembles TensorFlow's execution model.

3.3.3 Using `torch.compile()`

- In development, you write your code in PyTorch.
- Once the code is locked down, add `torch.compile()` as a decorator to optimize the model.
  - The optimized graph improves performance but can no longer be altered with tools like print statements.

3.3.4 Problem with JIT Compilation

JIT compilation can only be used on a **static execution graph**, which poses limitations:

- It struggles with **very dynamic programs**, i.e. those with frequent `if/else` branching.
- Static graphs prevent dynamic debugging and reduce flexibility.

3.3.5 Summary

- **Development:** Use PyTorch for dynamic, flexible execution.
- **Deployment:** Apply `torch.compile()` for optimized performance.

JIT compilation is best suited for models that can be locked down into a static graph. Highly dynamic behavior may reduce its effectiveness.

3.4 Static Models vs. Dynamic Models

3.4.1 Static Models & Static Dataflow Graphs

Static Models are models where the dataflow graph remains unchanged regardless of the input data. This requires that both the input and output data have fixed shapes. For example, a Convolutional Neural Network (CNN) is a static model. When input data with the same shape is fed into the model, the dataflow graph remains consistent, as illustrated in fig. 13.

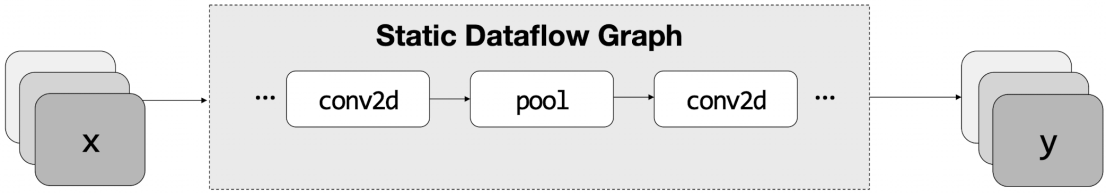

Figure 13: Static Models

In the case of static dataflow graphs, the graph is defined and optimized once. After this definition, the graph can be executed multiple times, ensuring that computations strictly adhere to the predefined structure.

3.4.2 Dynamic Models & Dynamic Dataflow Graphs

Dynamic Models are models where the dataflow graph changes based on the input data. This flexibility allows the model to handle input and output with varying shapes or structures. For example, in Semantic Parsing Trees for NLP, the structure of the parsing tree differs for each input sentence. If we use a Recurrent Neural Network (RNN) to compute over the tree structure, the dataflow graph must adapt dynamically to accommodate these variations (see fig. 14).

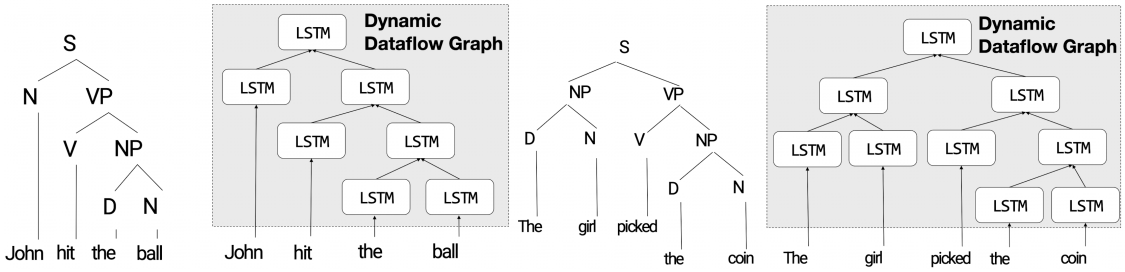

Figure 14: Dynamic Models

Dynamic dataflow graphs introduce challenges due to their complexity. Expressing complex flow-control logic becomes difficult since the graph structure varies with the input. Debugging is also more challenging, as errors are often tied to specific input cases. Furthermore, just-in-time (JIT) compilation is nearly impossible for programs relying on dynamic graphs.

3.4.3 How to Handle Dynamic Models?

Dynamic models are commonly used, particularly in natural language processing, where language is inherently dynamic—each sentence typically has a different number of tokens, and the computation depends on the input's length.

If performance is not a concern, one could simply adopt a define-and-run approach, disregarding just-in-time (JIT) compilation and optimization. However, to achieve better performance, we still hope to have some way to express the dynamic model in a relatively static way so that we can optimize it. Two common methods for handling this are introducing control flow operations and using piecewise compilation with guards.

Control Flow Operations

Control flow operations are a fundamental concept in programming languages. Examples of control flow operations include `if...then...`, `for`, `while`, and more.

Examples of control flow operations in dataflow graphs include the *Switch* and *Merge* primitives (fig. 15), introduced by the TensorFlow team. The *Switch* operation determines whether to pass the input data or output a dead tensor based on a boolean predicate. Conversely, the *Merge* operation takes two inputs and outputs the one that is not a dead tensor.

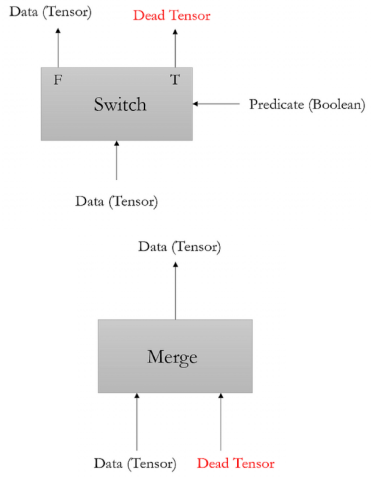

Figure 15: *Switch* and *Merge* Primitives

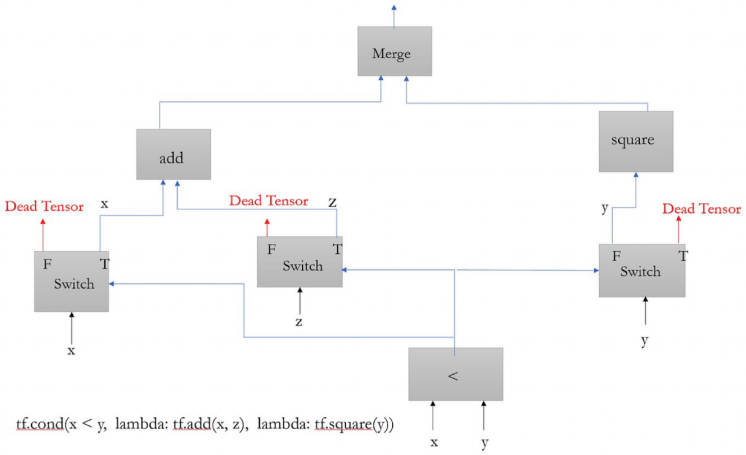

Figure 16: Control Flow Graph Example

By combining *Switch*, *Merge*, and other mathematical operators, it becomes possible to express dynamic programs as part of a static dataflow graph, as illustrated in fig. 16.

However, incorporating control flow operations into dataflow graphs, especially in machine learning, presents certain challenges:

- Deriving gradients for control flow operations (ie. switch, fork) can be challenging, although it is feasible.
- Adding control flow increases the complexity of the dataflow graph.

Piecewise Compilation

Case 1: How can we apply static JIT compilation to a graph that accepts input shapes of `[x, c1, c2]` where `c1` and `c2` are constants and `x` is a variable?

Possible piecewise compilation solutions:

1. Compile for All Possible Input Dimensions:
   In this case, we pre-compile multiple graphs, each corresponding to a specific value of `x`, and select the graph that matches the input dimensions at runtime.
2. Bucketing with Power of 2:
   We compile graphs for input shapes where `x` aligns with powers of 2 (e.g., `x = 1, 2, 4, 8, ...`). At runtime, the input dimensions are mapped to the closest ceiling power of 2 bucket. This reduces the number of pre-compiled graphs compared to the first approach, offering a balance between flexibility and resource utilization.

Case 2: How can we apply static JIT compilation to a graph that is static first, dynamic next, and then static?

This can be addressed by introducing **guards** at the boundaries of the static and dynamic sections. Specifically:

1. Insert guards at the end of the first static section and the beginning of the second static section.
2. Compile the static sections into binary files.
3. Execute the first static binary, pass its result into the dynamic section (which runs in pure Python), and feed the dynamic output into the second static binary for further processing.

4 Scribe Contributions

- Vincent Hsieh: section 1, section 2 background
- Keren Gutman: section 2.1
- Luke Mitbo: section 2.2
- Zhengan Cheng: section 2.4, section 2.5, section 2.6
- Zerlina Lai: section 3.1, section 3 background
- Ramsundar Tanikella : section 3.2
- Akanksha Pandey: section 3.3
- Michael Lai: Overall layout & section 3.4
- Jaedyn Chang: Peer review, editing, and formatting
- Nicholas King: Overleaf layout, peer review, and editing