Introduction to High-Performance Machine Learning

Lecture 3 02/08/24

Dr. Kaoutar El Maghraoui
Dr. Parijat Dube

Python and PyTorch performance

Performance Factors

Algorithms Performance

- Algorithm choice

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- **Python performance & PyTorch performance**

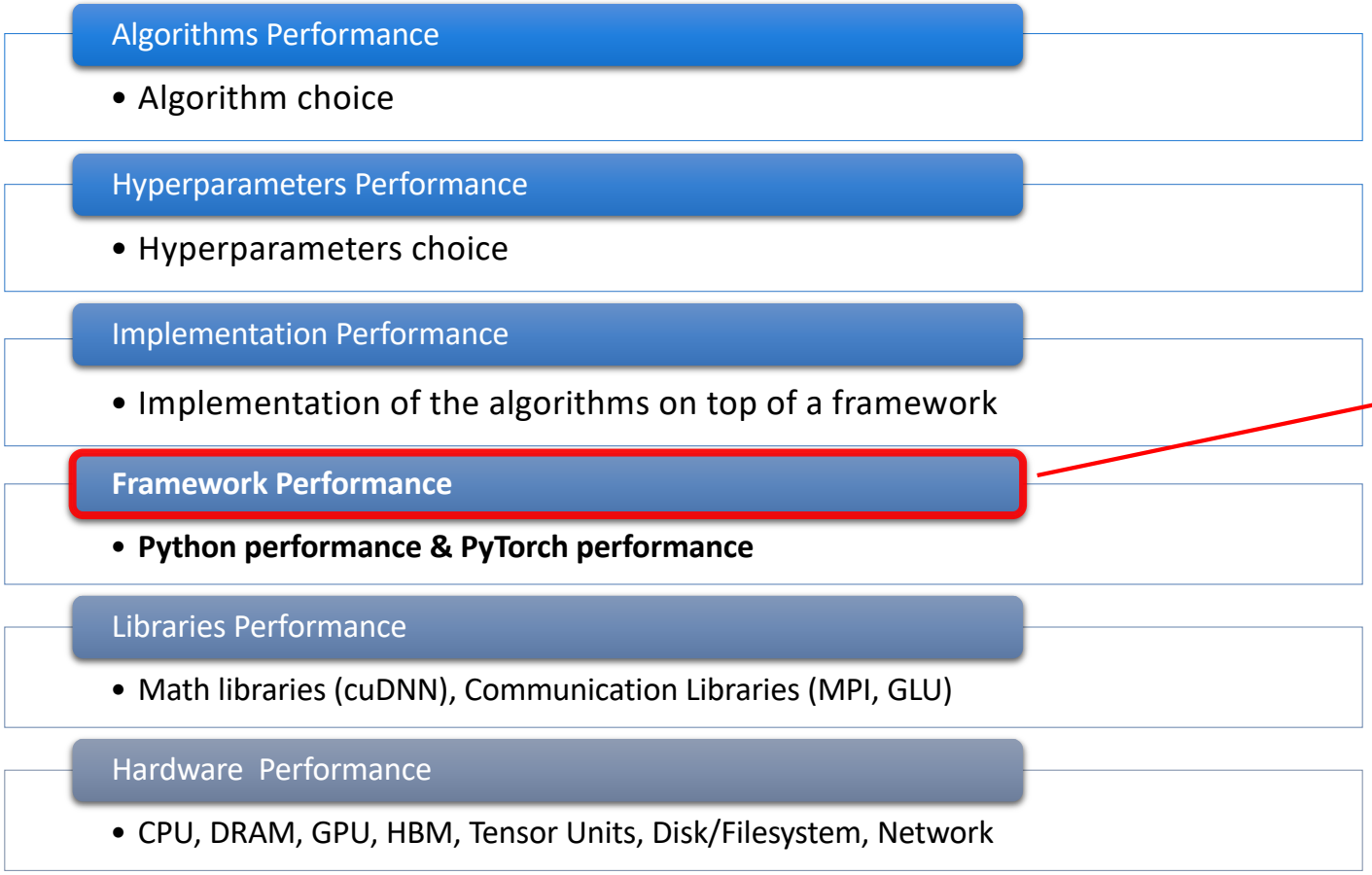
Libraries Performance

- Math libraries (cuDNN), Communication Libraries (MPI, GLU)

Hardware Performance

- CPU, DRAM, GPU, HBM, Tensor Units, Disk/Filesystem, Network

Performance Factors



Outline

- Python performance
- PyTorch performance
  - Computation Graph Approach
  - Just In Time Compilation
  - Profiling
  - Benchmarking

Python performance

Python

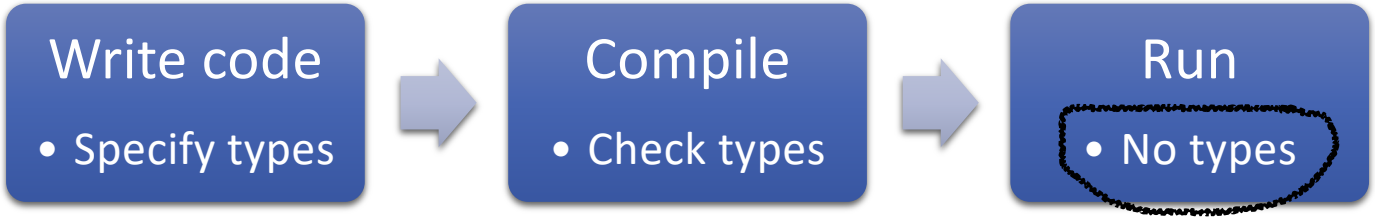
- Created in 1991 by Guido Van Rossum
- **Productivity-oriented** language
- Focuses on **Code readability:**
  - Fewer lines of code
  - Enormous library support
  - Many libraries in Python have underlying C++ code
- Performance relevant features:
  - **Dynamic typing** type of a variable is interpreted at runtime,
  - **Memory management**

Static Typing - C/C++

- The programmer must specify the types of each variable
- The compiler checks types at **compile time**
- Implications:
  - All types are known before execution like how much memory integer needs.
  - Variables in memory do not need to contain types, only values

as memory allocation and other things are already done at compile time.

```
/* C code */
int a = 1;
int b = 2;
int c = a + b;
```

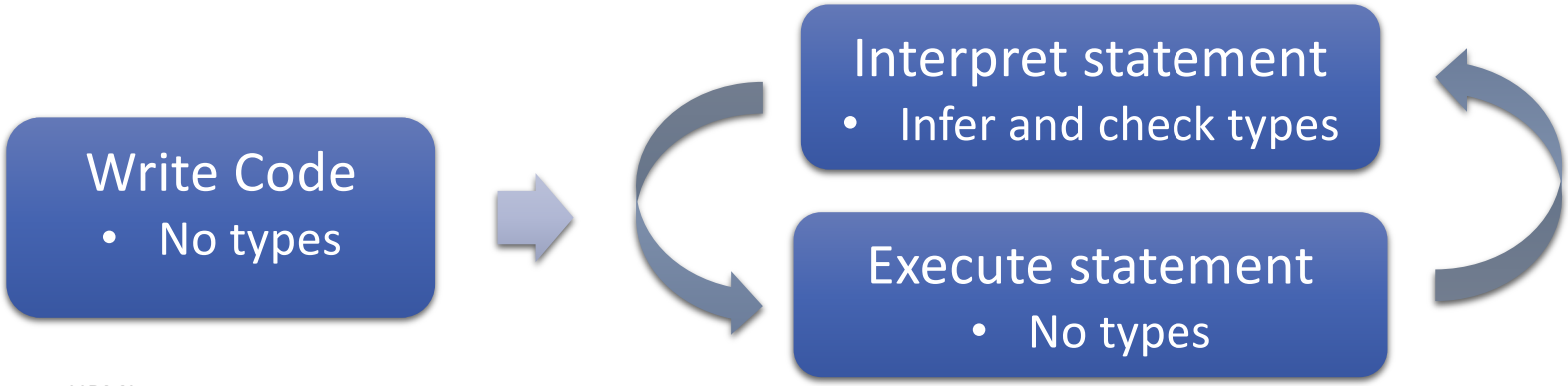


Dynamic Typing – Python

- Dynamic Typing (Python language):
  - Programmer does not specify variables types
  - Interpreter checks types at **run-time**
  - **Types are known only during execution**

```
# python code
a = 1
b = 2
c = a + b
```

- Duck typing: **"If it walks like a duck and it quacks like a duck, then it must be a duck"**
- An object is of a given type if it has all methods and properties required by that type



Example

- The ***ide*** in code method can have different types.
- What is key is that the object passed has an execute() method
- The type is dynamic

```
class MyEditor:

    def execute(self):
        print("Spell Check")
        print("Convention Check")
        print("Compiling")
        print("Running")

class Laptop:

    def code(self,ide):
        ide.execute()


ide = PyCharm()

lap1 = Laptop()
lap1.code(ide)
```

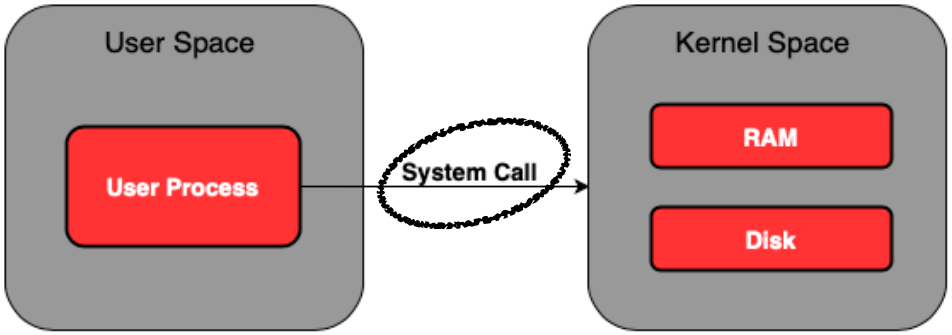
Static vs. Dynamic Types

- Static:
  - The compiler can catch common errors and mistakes early on (+)
  - You can fix things before deployment (+)
  - More code to write (-)
  - Need to know all the types beforehand (-)

- Dynamic
  - More expressive and flexible (+)
  - Write code faster (+)
  - Code more error-prone and brittle (-)

Memory Management Concepts

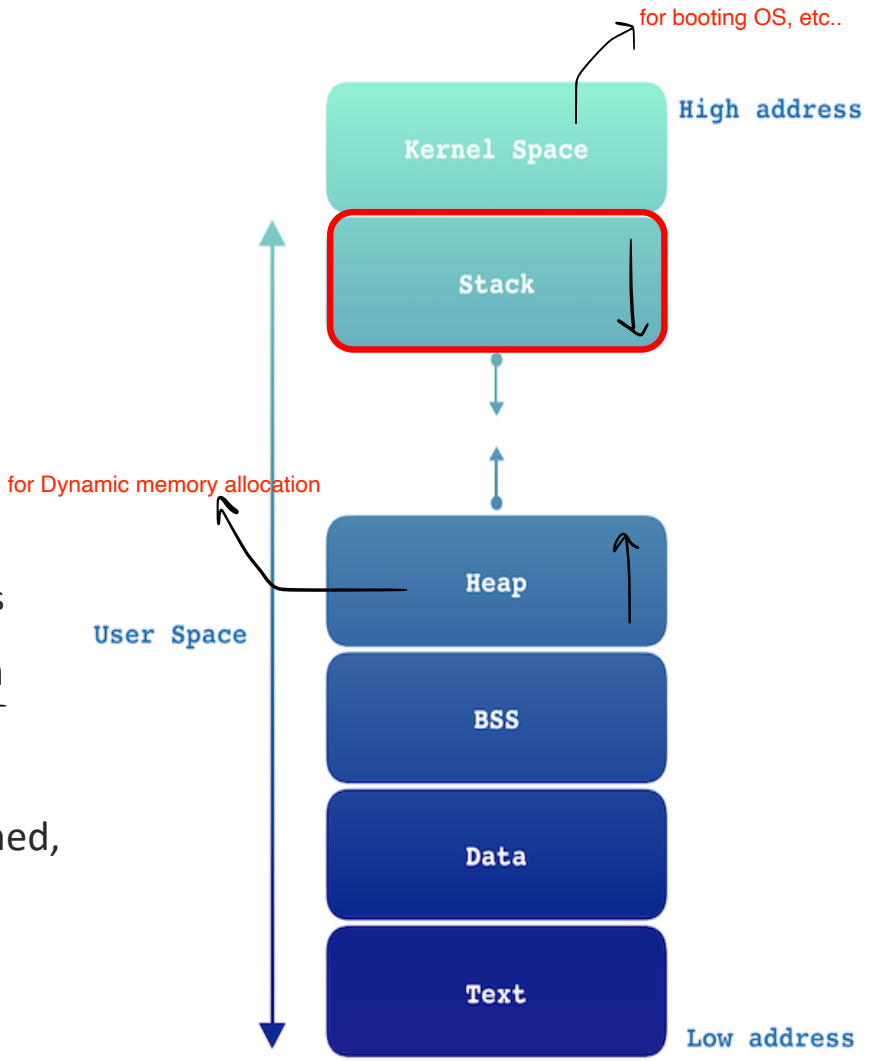
- **Kernel Space**
  - The kernel space can be accessed by user processes only using system calls that are requests in a Unix-like operating system such as input/output (I/O) or process creation.
- **User Space**
  - The user space is a computational resource allocated to a user, and it is a resource that the executing program can directly access. This space can be categorized into some segments.



kernel space and user space refer to distinct regions of a computer's memory that are utilized by the operating system and user applications, respectively.

Memory Layout in a Program

- **Stack**
  - Located just under the OS kernel space
  - Grows downwards to lower addresses
  - LIFO (last-in-first-out) data structure
  - Two principal operations:
    - **Push** adds an element to the collection, and
    - **Pop** removes the most recently added element that was not yet removed.
  - Devoted to storing all the data needed by a function call in a program.
  - Calling a function is the same as pushing the called function execution onto the top of the stack
  - Once that function completes, the results are returned, popping the function off the stack.
  - Memory allocated for stack is fixed during program execution.
  - **Speedy execution**



memory used is contiguous

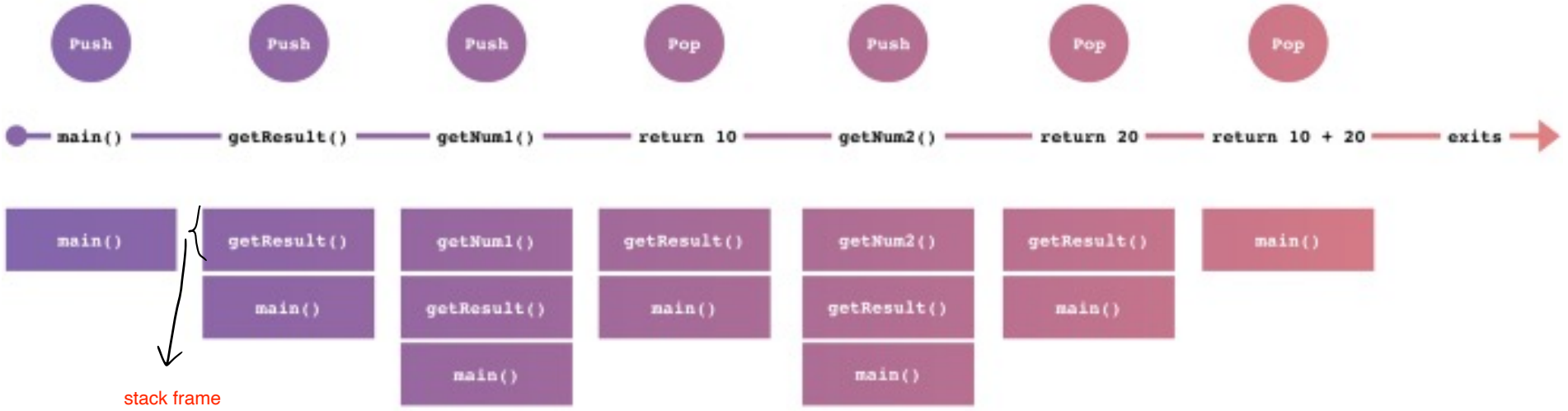
Example

The data pushed for a function call is named a **stack frame**, and it contains the following data.

- The arguments (parameter values) passed to the routine
- The return address back to the routine's caller
- Space for the local variables of the routine

- When the function is called, the stack frame is pushed to the top of stack. Then the process is executed, and the function goes out of scope, the stack frame pops from the top.

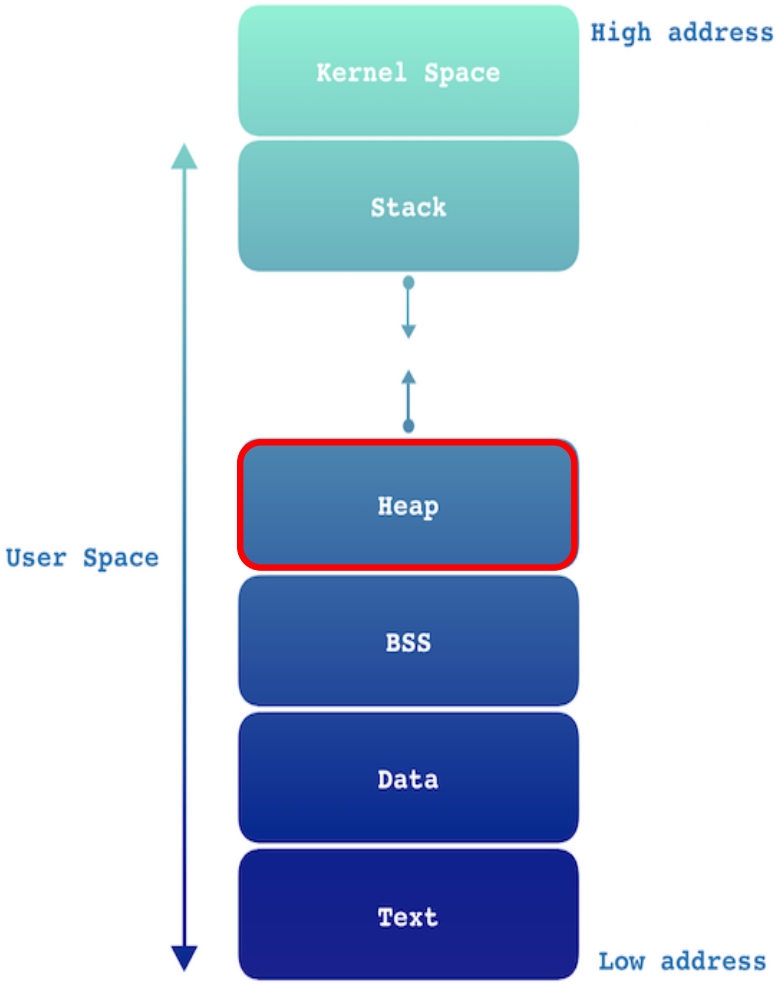
```
int main() {
  int result = getResult();
}

int getResult() {
  int num1 = getNum1();
  int num2 = getNum2();
  return num1 + num2;
}

int getNum1() {
  return 10;
}

int getNum2() {
  return 20;
}
```



why heap is used for memory allocation??

Memory Layout in a Program

- **Heap**
  - Segment where dynamic memory allocation usually takes place
  - In C, it's managed by malloc / new, free / delete
  - The allocation to the heap area occurs in the following cases:
    - Memory size is dynamically allocated at run-time
    - Scope is not limited. (i.e., variables referenced from several places)
  - Memory size is very large
  - The objects on the heap lead to memory leaks if they are not freed Memory Overflow error
  - In garbage-collected languages, the garbage collector frees memory on the heap and prevents memory leaks.
  - Application heap is not fixed
  - **Performance is relatively low**.

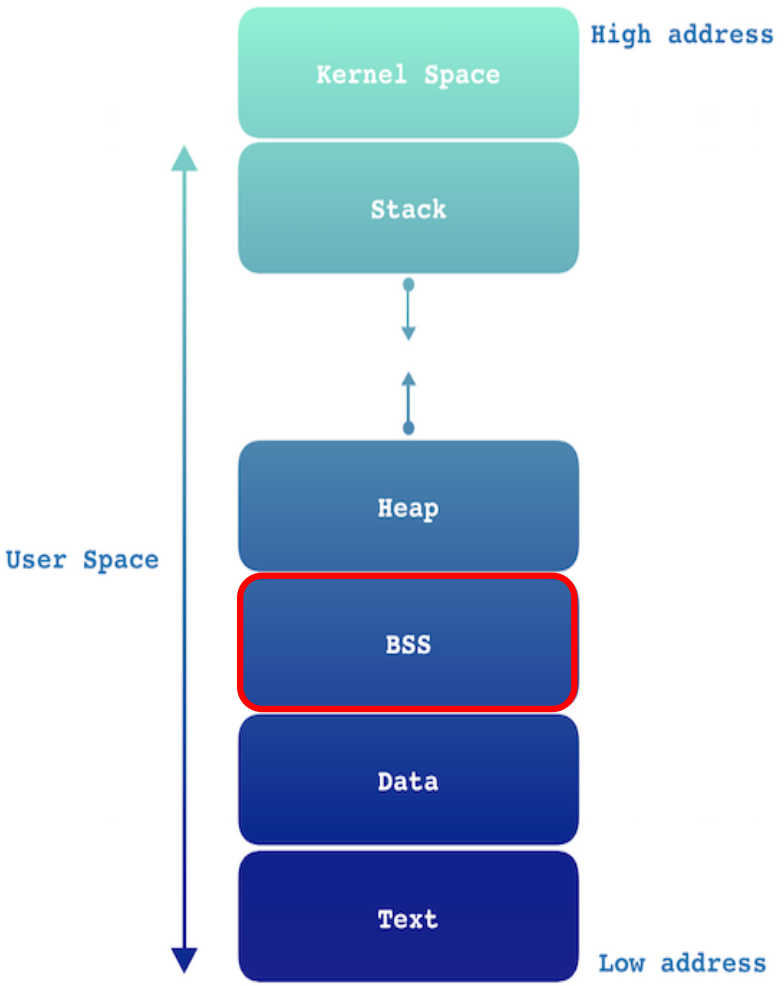


As it is not contiguous. can lead to fragmented memory. (Internal and external fragmentation)

why its not contiguous - coz of prefetcher

Memory Layout in a Program

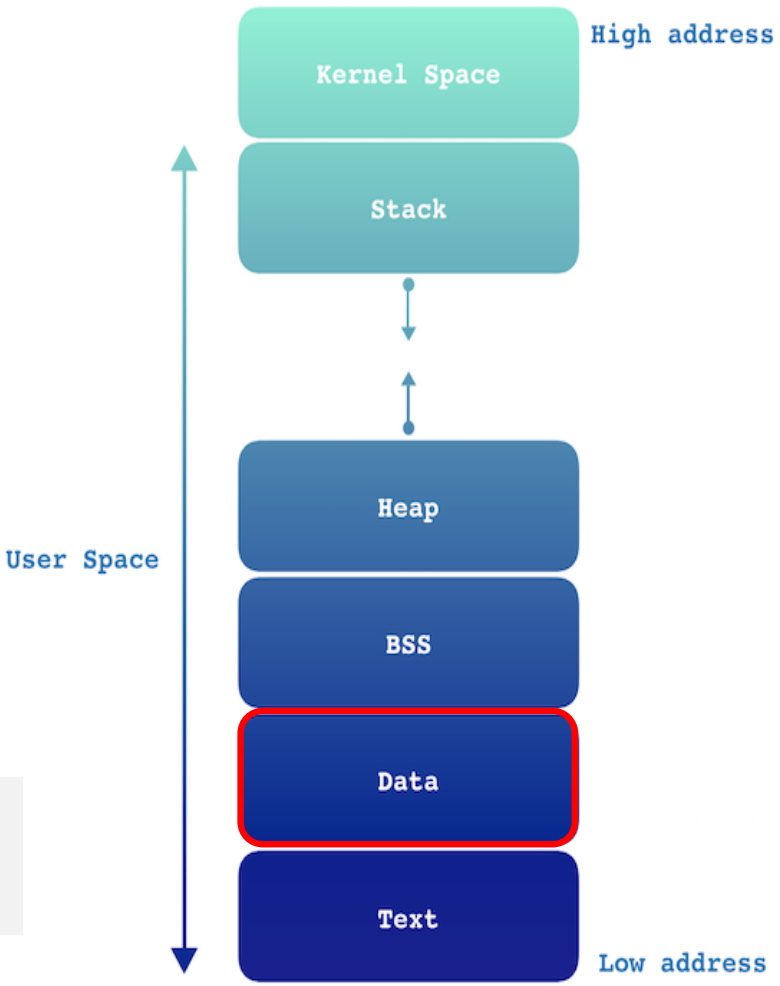
- **BSS (Block Started by Symbol)**
  - The uninitialized data segment is often called the BSS segment.
  - Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.
  - Example: a variable declared as static int i; would be allocated to the BSS segment.
  - Both BSS and Data usually refer to RAM objects.



Memory Layout in a Program

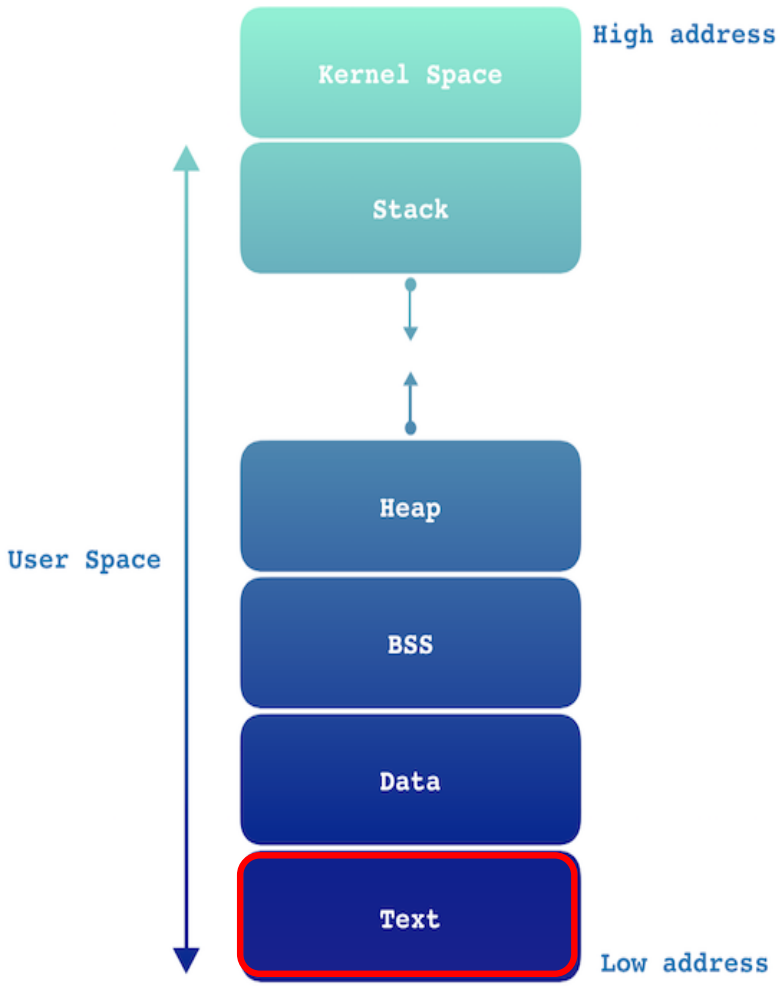
- **Data**
  - The data segment contains initialized global and static variables which have a pre-defined value and can be modified.
  - Divided into a read-only and a read-write space.
  - For example, the following C program outside the main

```
int val = 3;
char string[] = "Hello World";
```



Memory Layout in a Program

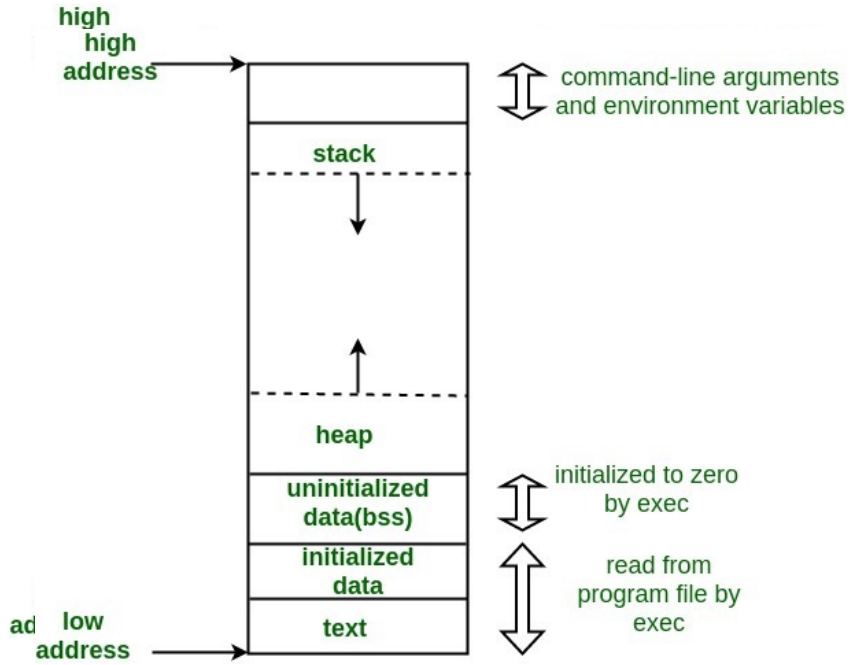
- **Text** for source code.
  - A segment in which a machine language instruction is stored.
  - This segment is a read-only space.



Memory Management

- Process' **stack**:
  - Automatic memory management by OS and Compiler
- Process' **heap**:
  - **Manual Memory management (ex. C)**
  - **Automatic Memory Management (ex. Python)**

- Thread's **stack:**
  - Resides in parent process' **heap**
  - Automatic mem. management by the thread library
- Thread's **heap**:
  - Manual Mem. Management (ex. C)
  - Automatic Mem. Management (ex. Python)



Process' memory layout

From: https://cdncontribute.geeksforgeeks.org/wp-content/uploads/memoryLayoutC.jpg

https://www.geeksforgeeks.org/memory-layout-of-c-program/

Manual Memory Management

- Programmer allocates and deallocates buffers in the **HEAP**
  - C language: *malloc() free()*
  - C++ language: *new* and *delete*

- Pros:
  - Higher **performance**
  - Deeper understanding of the program by the programmer
- Cons:
  - Code **complexity**
  - Higher **risk of bugs**
  - Lower programmer's **productivity**

- Languages:
  - Algol; **C; C++;** COBOL; Fortran; Pascal
- Tools for Memory Management Profiling: **Valgrind, GDB**

- http://www.memorymanagement.org/mmref/begin.html#manual-memory-management

Automatic Memory Management

- Runtime system is in charge to allocate and deallocate buffers in the **HEAP:**
  - Allocation embedded in the language, ex. object creation
  - Recycling techniques – Garbage collection:
    - Keep track of all references to objects
    - Free objects that are not needed anymore
- Pros:
  - Higher programmer's **productivity**
  - Code **simplicity**
  - Lower risk of **bugs**
- Cons:
  - Lower **performance**
  - Lower memory management **time and space efficiency**
- Languages:
  - BASIC, Dylan, Erlang, Haskell, Java, JavaScript, Lisp, ML, Modula-3, Perl, PostScript, Prolog, **Python**, Scheme, Smalltalk, etc.

C,C++ -> creates assembly code (machine dependent) - can directly interact with hardware

Python/Java -> create byte code (machine independent) -> converted to machine readable format

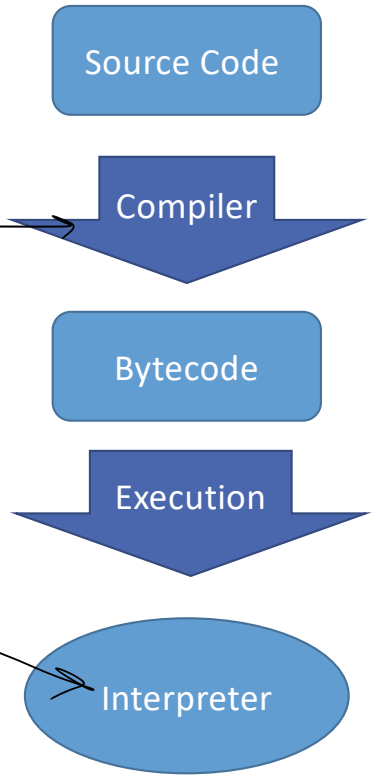
Python implementations

- Python Implementation
  - *a program or environment (runtime) which provides support for the execution of programs written in the Python language*
- **CPython** is the de-facto Python reference implementation written in C
- Alternative implementations
  - *Brython, CLPython, HotPy, IronPython, Jython, pyjs, PyMite, PyPy, pyvm, etc*.
- Compilers: compiling Python to C code
  - ***CPython**, 2c-python, GCC Python Front-end, etc*.
- Numerical Accelerators/Frameworks: offer accelerated numerical libraries (often c optimized libraries with Python APIs)
  - **PyTorch,** Numpy, Numba, Copperhead,

- https://wiki.python.org/moin/PythonImplementations
- https://medium.com/@elhayefrat/python-cpython-e88e975e80cd

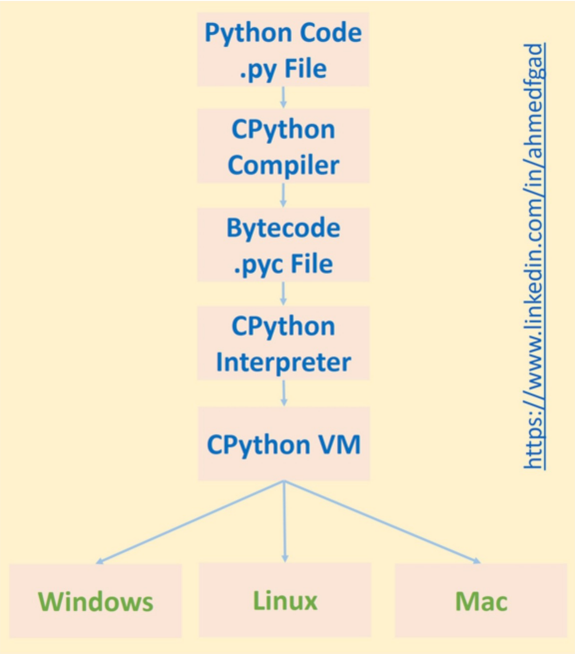
Python Execution Stages

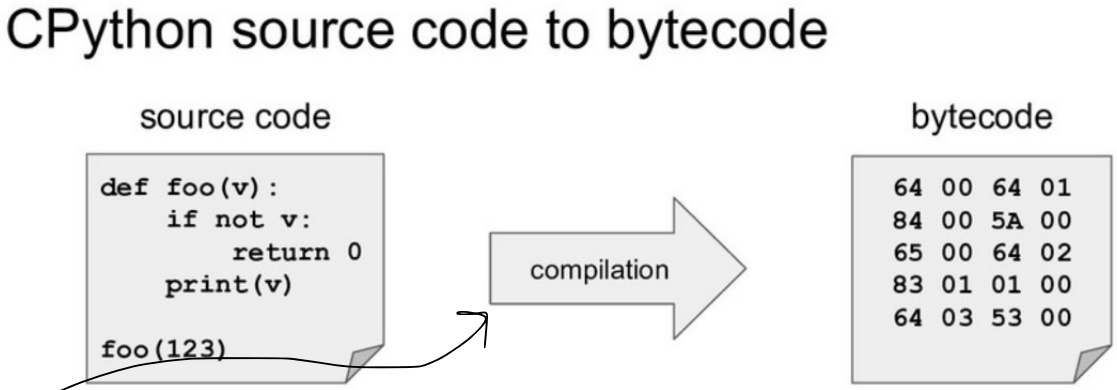
- CPython compiler:
  - Uses several stages to produce **bytecode**
  - Checks basic syntax and grammatical correctness
  - Bytecode can be saved in a .pyc file
  - Do not confuse with ***Cython***: superset of Python language to call C functions that is used to generate C code
- Cpython interpreter (Virtual Machine):
  - Executes the program described by the bytecode
- https://devguide.python.org/compiler/#



Cython tutorial: http://conference.scipy.org/proceedings/SciPy2009/paper_1/

CPython Flow





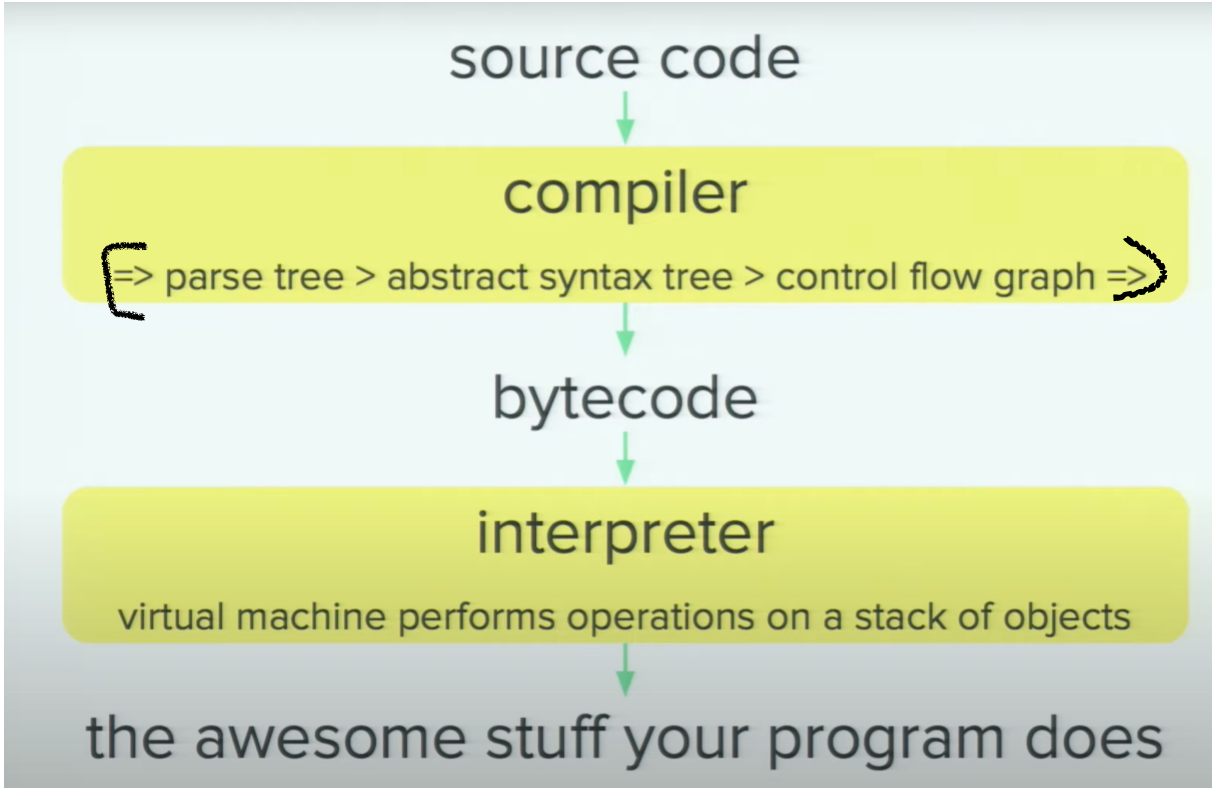
1) Parse source code into a parse tree (Parser/pgen.c)
2) Transform parse tree, Concrete Syntax Tree (CST) to an Abstract Syntax Tree (AST) (Python/ast.c)
3) Transform AST into a Control Flow Graph (CFG) (Python/compile.c)
4) Emit the bytecode based on the CFG (Python/compile.c)
5) Optimize the bytecode with peephole optimizations (Python/peephole.c)

Peephole Optimizations

- Peephole optimization is a technique used in Python and other programming languages to improve performance by making small optimizations at the bytecode level.
- This process involves looking at small sequences of bytecode (the "peephole") and finding ways to replace or remove certain operations with more efficient ones.
- Some common examples of peephole optimizations in Python:

  - **Constant Expressions**: Evaluating constant expressions at compile time rather than at runtime. For example, an expression like 2 + 2 will be replaced with 4.
  - **Constant Folding**: Combining constant operations into a single operation. For example, 4 * 25 would be optimized to 100.
  - **Removing Unused Imports**: If an import statement is declared but not used, it can be removed.
  - **Optimizing Short Sequences**: Replacing short sequences of instructions with more efficient ones. For example, replacing a sequence of LOAD_CONST instructions with a single BUILD_TUPLE instruction.
  - **Variable Lookup Optimization**: Optimizing the lookup of variables by replacing global variable lookups with faster local variable lookups when possible.

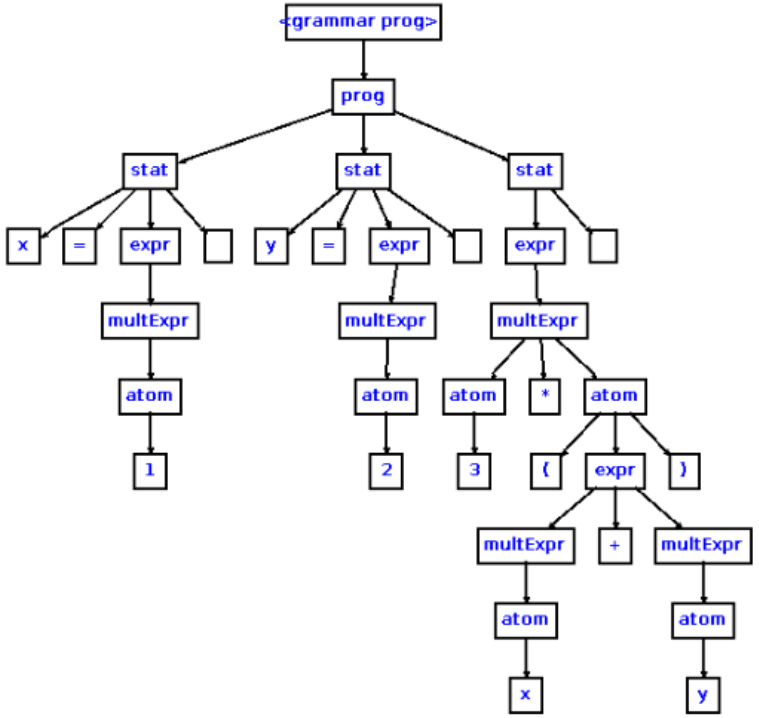
Summary of Python Execution Steps



Concerete Syntax Tree

Example: CST

- x=1
- y=2
- 3*(x+y)



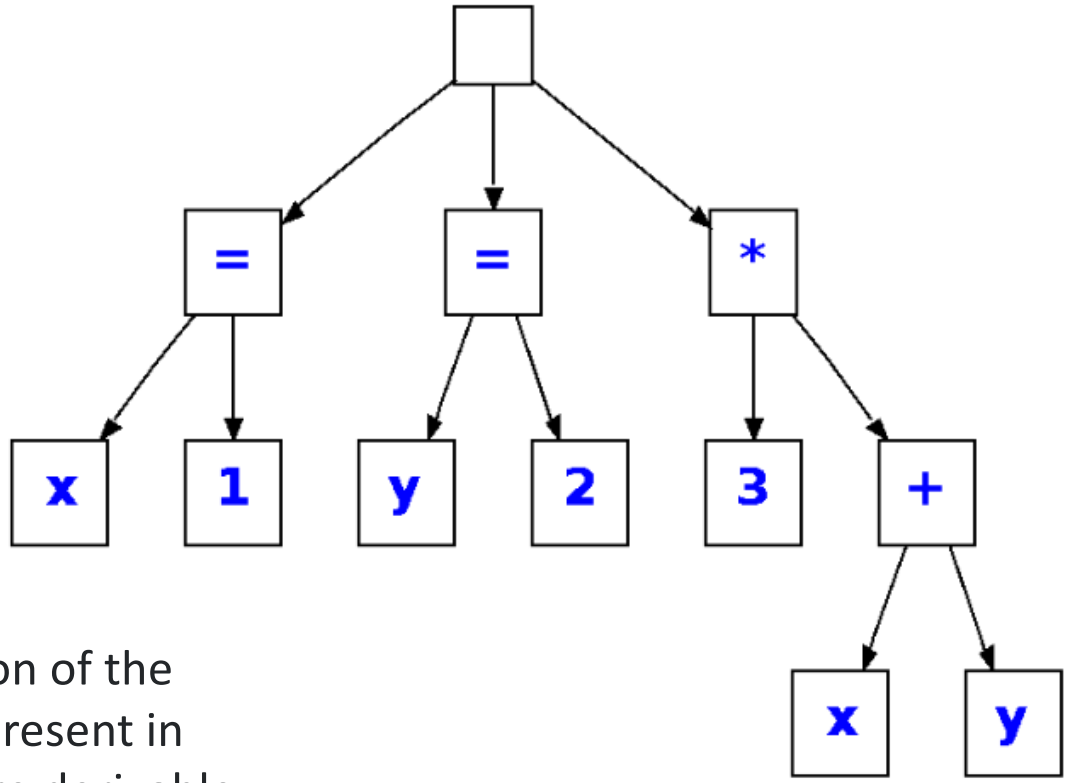
The parse tree is a concrete representation of the input. The parse tree retains all the information of the input. The empty boxes represent whitespace, i.e., end of line.

can be used to catch syntax error, doing some visualization, etc.

Abstract Syntax Tree

Example: AST

- x=1
- y=2
- 3*(x+y)



The AST is an abstract representation of the input. Notice that parents are not present in the AST because the associations are derivable from the tree structure.

represents logical structure

used in the later stage of compiler

Python Bytecode

- Bytecode looks like a simplified assembly code for a **stack machine**
- Caches in .pyc file for faster execution the second time the same file is executed
- Run on a VM that executes the machine code corresponding to each byte code
- Core of CPython is an eval loop that implements a simple stack-based virtual machine.
- The bytecode generated for any user function (or code in general) can be inspected using the dis module in human readable form

```
>>> import torch
>>> import dis
>>> def f():
...     x = torch.randn(2,2)
...     return x.mm(x)
...
>>> dis.dis(f)
  2           0 LOAD_GLOBAL              0 (torch)
              2 LOAD_ATTR                1 (randn)
              4 LOAD_CONST               1 (2)
              6 LOAD_CONST               1 (2)
              8 CALL_FUNCTION            2
             10 STORE_FAST               0 (x)

  3          12 LOAD_FAST                0 (x)
             14 LOAD_ATTR                2 (mm)
             16 LOAD_FAST                0 (x)
             18 CALL_FUNCTION            1
             20 RETURN_VALUE
>>>
```

dis: bytecode disassembler

https://docs.python.org/library/dis.html

```
>>> def hello():
...     return "Kaixo!"
...
>>> import dis
>>> dis.dis(hello)
  2           0 LOAD_CONST               1 ('Kaixo!')
              3 RETURN_VALUE
```

```
1  >>> from dis import dis
2  >>> def square(x):
3  ...     return x*x
4  ...
5
6  >>> dis(square)
7  2           0 LOAD_FAST                0 (x)
8              2 LOAD_FAST                0 (x)
9              4 BINARY_MULTIPLY
10             6 RETURN_VALUE
```

- The ./Include/opcodes.h file contains a listing of the Python Virtual Machine's bytecode instructions

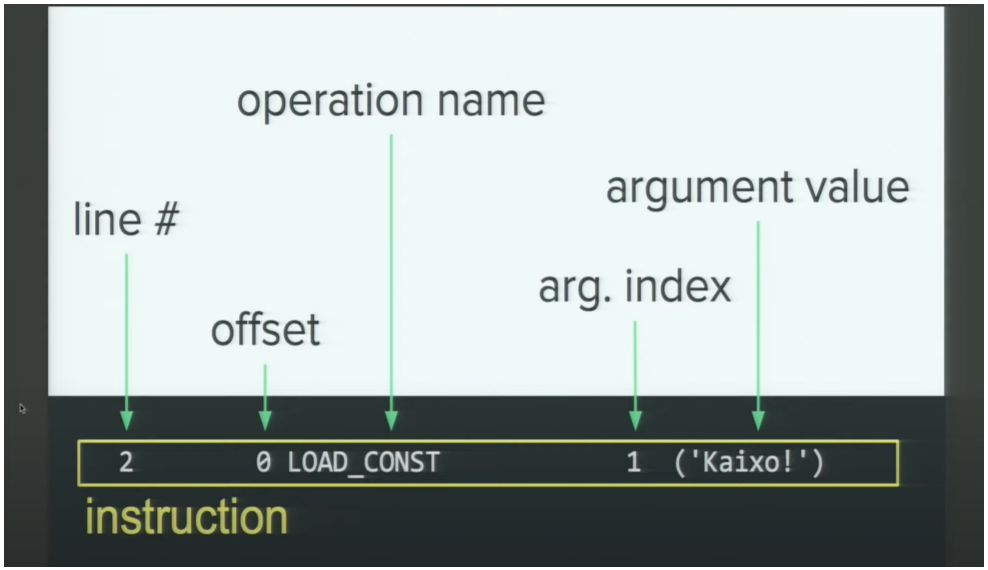
Bytecode Instructions

sample operations

https://docs.python.org/library/dis.html#python-bytecode-instructions

`LOAD_CONST(c)`	pushes *c* onto top of stack (TOS)
`BINARY_ADD`	pops & adds top 2 items, result becomes TOS
`CALL_FUNCTION(a)`	calls function with arguments from stack *a* indicates # of positional & keyword args

```
>>> dis.opmap['BINARY_ADD']    # => 23
>>> dis.opname[23]             # => 'BINARY_ADD'
```



https://www.youtube.com/watch?v=GNPKBICTF2w

Python interpreter

- Always running in a basic main thread
- Can do context-switching among its threads
- Based on a **Stack Machine with push and pop**
- Interpreter loop:
  1. Read next instruction in bytecode
  2. Evaluate the 16 bits bytecode: ***oparg*** and ***opcode***
  3. Switch/case: Call the corresponding C function (macro) that executes the instruction

operation code

CPython interpreter - Stack Machine Example

- Evaluate expression:

```
d = a + b * c
```

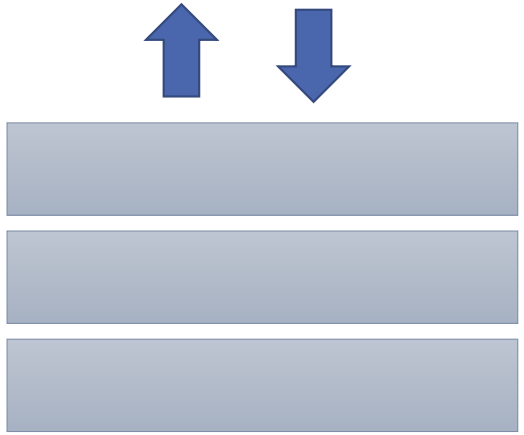
- Values array: [a, b, c, d]
- Compiled Bytecode:

```
LOAD_FAST 0
LOAD_FAST 1
LOAD_FAST 2
BINARY_MULTIPLY
BINARY_ADD
STORE_FAST 3
```

1 & 2

Index in the values array

- Stack state:



CPython interpreter - Stack Machine Example

- Evaluate expression:

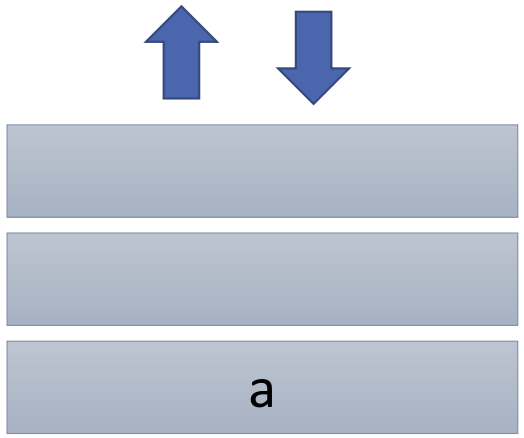
```
d = a + b * c
```

- Values array: [a, b, c, d]
- Compiled Bytecode:

IP

```
LOAD_FAST 0
LOAD_FAST 1
LOAD_FAST 2
BINARY_MULTIPLY
BINARY_ADD
STORE_FAST 3
```

- Stack state:



CPython interpreter - Stack Machine Example

- Evaluate expression:

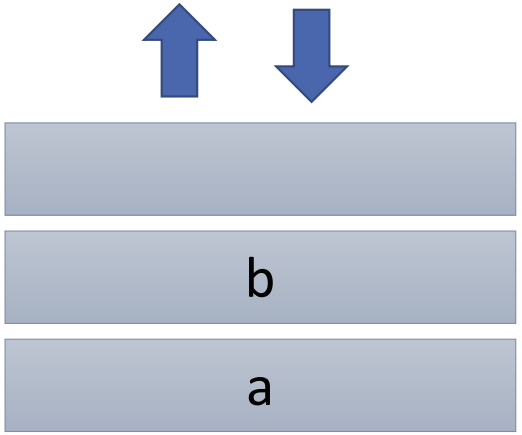
```
d = a + b * c
```

- Values array : [a, b, c, d]
- Compiled Bytecode:

IP

```
LOAD_FAST 0
LOAD_FAST 1
LOAD_FAST 2
BINARY_MULTIPLY
BINARY_ADD
STORE_FAST 3
```

- Stack state:



CPython interpreter - Stack Machine Example

- Evaluate expression:

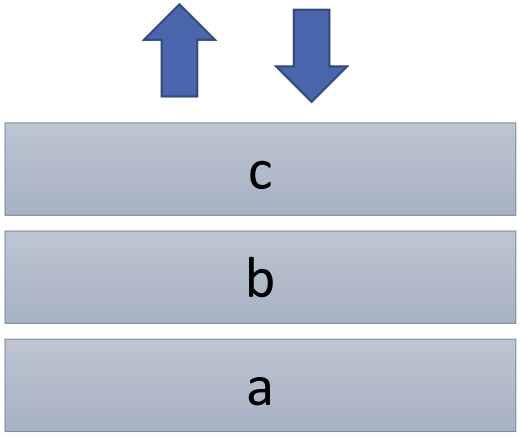
```
d = a + b * c
```

- Values array: [a, b, c, d]
- Compiled Bytecode:

IP

```
LOAD_FAST 0
LOAD_FAST 1
LOAD_FAST 2
BINARY_MULTIPLY
BINARY_ADD
STORE_FAST 3
```

- Stack state:



CPython interpreter - Stack Machine Example

- Evaluate expression:

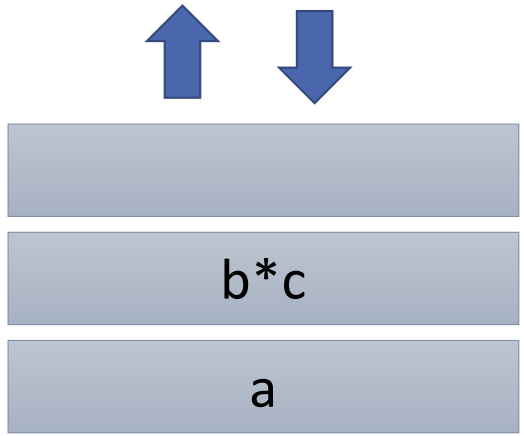
```
d = a + b * c
```

- Values array: [a, b, c, d]
- Compiled Bytecode:

```
LOAD_FAST 0
LOAD_FAST 1
LOAD_FAST 2
BINARY_MULTIPLY
BINARY_ADD
STORE_FAST 3
```

IP

- Stack state:



CPython interpreter - Stack Machine Example

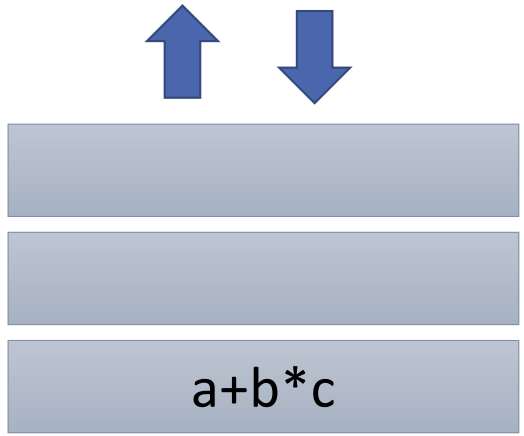
- Evaluate expression:

```
d = a + b * c
```

- Values array: [a, b, c, d]
- Compiled Bytecode:

```
IP  LOAD_FAST 0
    LOAD_FAST 1
    LOAD_FAST 2
    BINARY_MULTIPLY
    BINARY_ADD
    STORE_FAST 3
```

- Stack state:



CPython interpreter - Stack Machine Example

- Evaluate expression:

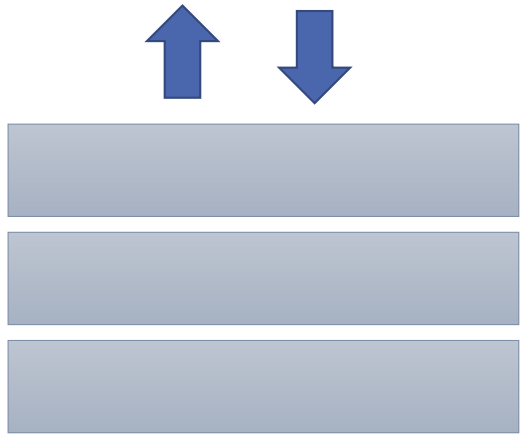
```
d = a + b * c
```

- Values array: [a, b, c, **d**]
- Compiled Bytecode:

```
LOAD_FAST 0
LOAD_FAST 1
LOAD_FAST 2
BINARY_MULTIPLY
BINARY_ADD
STORE_FAST 3
```

IP

- Stack state:

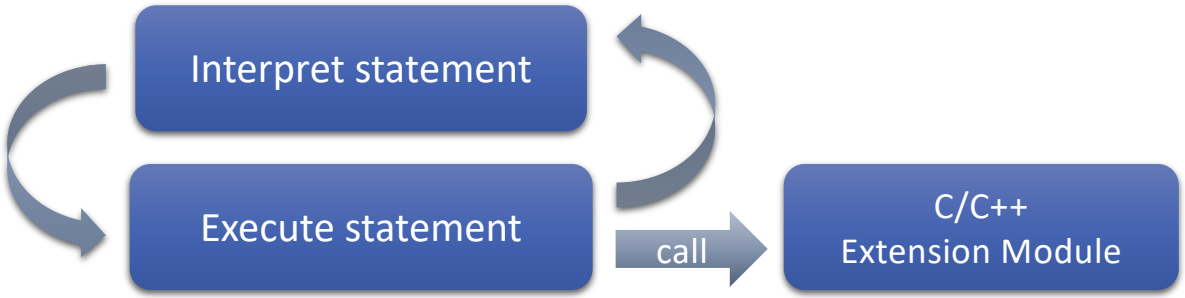


C/C++ -> Assembly code (machine dependent)
Python -> Byte code (machine independent)

CPython Extension Modules

enable the seamless incorporation of C code into Python programs

- Commonly used for **performance critical** codes
- Advantages:
  - Create a Python Object that has **a C/C++ data structure** inside, instead of using Python data-structures and objects
  - Allows the CPython interpreter **to directly call C/C++ compiled functions** and system-calls
  - Extension modules are **compiled** and **linked** (usually as .so) to the CPython binary
- Using Cython one can write extension modules



- See https://docs.python.org/3.7/extending/index.html
- Example: https://realpython.com/build-python-c-extension-module/

CPython Extension Module Example

- Method definition
- Add Method to Module
- Module definition
- Module initialization

```
static PyObject* hello_module_print_hello_world(PyObject
*self, PyObject *args) {
   printf("Hello World\n");
   Py_RETURN_NONE;
}
static PyMethodDef hello_module_methods[] = {
   { "print_hello_world",
      hello_module_print_hello_world,
      METH_NOARGS,
      "Print 'hello world' from a method defined in a C extension."
   },
   {NULL, NULL, 0, NULL}
};
static struct PyModuleDef hello_module_definition = {
   PyModuleDef_HEAD_INIT,
   "hello_module",
   "A Python module that prints 'hello world' from C code.",
   -1,
   hello_module_methods };
PyMODINIT_FUNC PyInit_hello_module(void) {
   Py_Initialize();
   return PyModule_Create(&hello_module_definition);
}
```

https://realpython.com/build-python-c-extension-module/

PyTorch Performance - Computational Graph

PyTorch

A Python-based scientific computing package targeted at two sets of audiences:

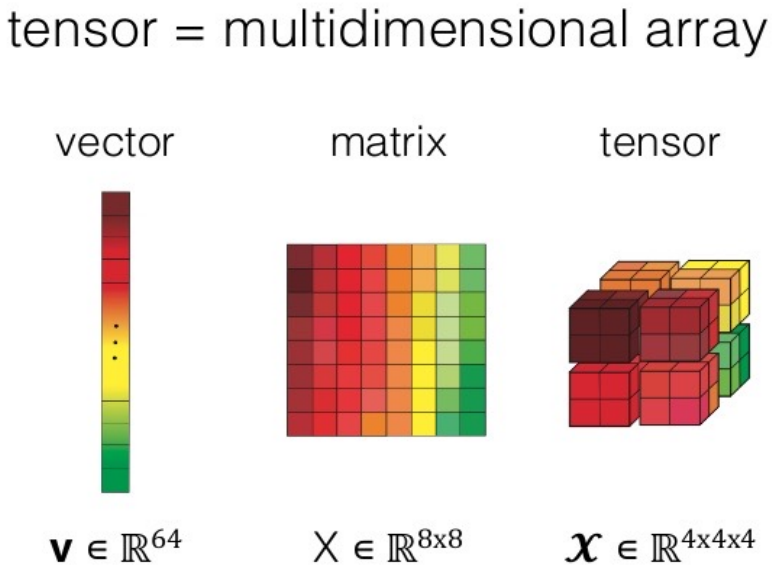
- *A replacement for numpy to use the power of GPUs*
- *A deep-learning research platform that provides maximum flexibility and speed*

[This lesson uses material from http://pytorch.org/tutorials/ *throughout.]*

- To install: https://github.com/pytorch/pytorch#installation

Tensors

- Tensors are matrix-like data structures which are essential components in deep learning libraries and efficient computation.
- GPUs are especially effective at calculating operations between tensors



From: https://www.slideshare.net/BertonEarnshaw/a-brief-survey-of-tensors

- Tensor operations:
  - ones, zeros, add, dot, etc.
- PyTorch tensors can live on
  - CPU
  - GPU (speedup!)
  - Or other accelerators

PyTorch Tensors

- Import Torch:

```
from __future__ import print_function
import torch
```

- Construct a 2x3 matrix, uninitialized:

```
x = torch.Tensor(2, 3)
print(x)
 0.00e+00  0.00e+00  1.15e-24
 1.58e-29  1.67e-37  2.97e-41
[torch.FloatTensor of size 2x3]
```

- Use tensor in CUDA

```
device = torch.device("cuda")
y = torch.ones_like(x, device=device) # directly create a tensor on GPU
x = x.to(device) # or just use .to("cuda")
```

- Initialize zeros or ones tensors

```
x = torch.zeros(2,3)
x = torch.ones(2,3)
```

- Convert a numpy array

```
b = torch.from_numpy(a)
```

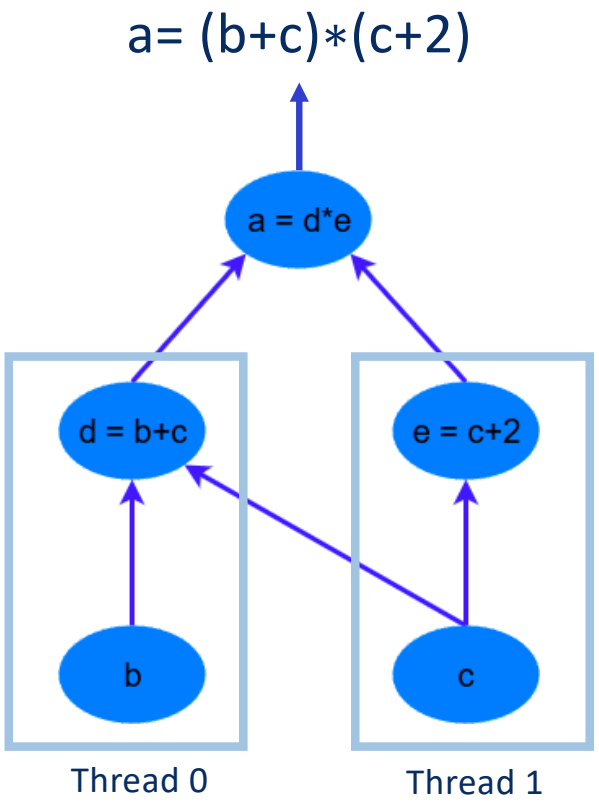
- the slice functionality is available like in numpy

```
x = torch.rand(2,3) # Initialize a tensor randomly
print x[:,1] #second column
0.6297 0.1196
[torch.FloatTensor of size 2]
print x[0,:] #first row
0.9749 0.6297 0.3045
[torch.FloatTensor of size 3]
```

a DAG

Computation Graphs

- A computational graph represents a function in a directed acyclic graph of its component functions
- **Performance optimizations:** Computation Graph exposes parallelism!
- In PyTorch the graph construction is **dynamic:** the graph is built at run-time
  - Easier debugging
  - Better for some algorithms (RNNs)
- In TensorFlow graph construction is **static:** meaning the graph is "compiled" and then run
  - Compiler adds latency but can also apply optimizations



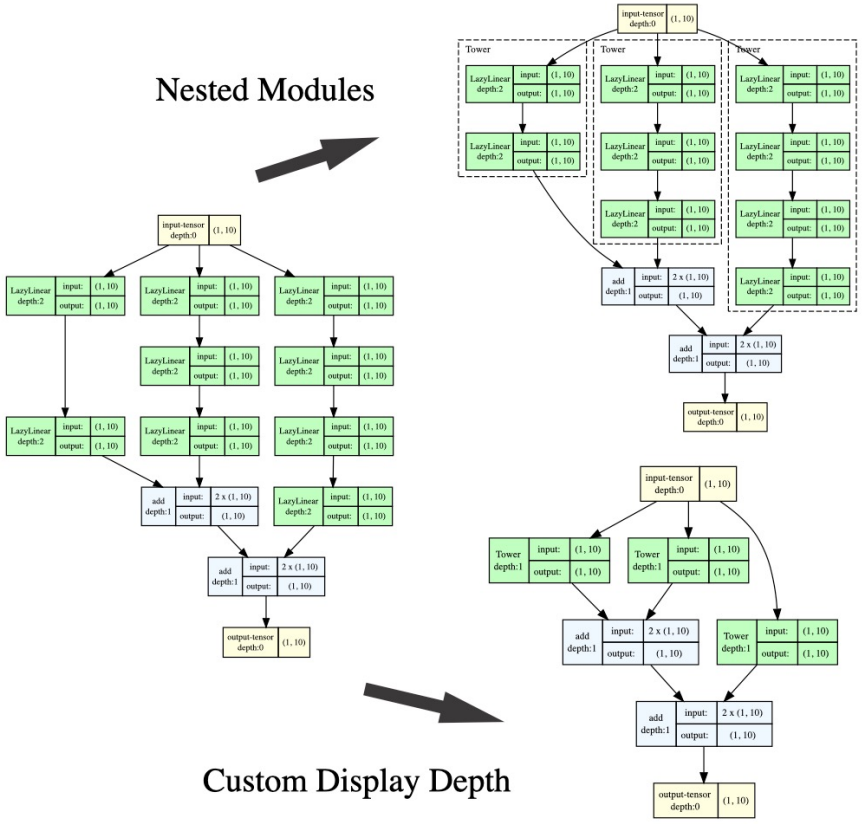
performance can be improved - thus used for deployment

Visualizing PyTorch Computational Graphs

- Example: Torchview provides visualization of PyTorch models in the form of visual graphs. Visualization includes tensors, modules, torch.functions, and info such as input/output shapes.

- Link: https://github.com/mert-kurttutan/torchview

- Example Notebooks
  - Introduction Notebook
  - Computer Vision Models Notebook
  - NLP Models Notebook



Autograd in PyTorch

- **Autograd** builds the Computation Graph **Dynamically**
- The **Tensor** class is the main component of this autograd system in PyTorch (from PyTorch 0.4 version, the *Variable* class is deprecated)
- If you set its attribute *.requires_grad* as *True*, it starts to track all operations on it
- The gradient for this tensor will be accumulated into *.grad* attribute
- Tensors allow automatic gradient computation when the *.backward()* function is called
- Based on the graph, *<variable>.backward()* computes the **gradient** and writes it in ***grad***
  - Example: **b.backward()** computes $\frac{d(y)}{dx}$

```
x = torch.randn(5, 5) # requires_grad=False by default
y = torch.randn(5, 5) # requires_grad=False by default
z = torch.randn((5, 5), requires_grad=True)
a = x + y
a.requires_grad
            False
b = a + z
b.requires_grad
            True
b.backward
```

PyTorch Tensors, Functions and Gradients

- Create a tensor

```
x = torch.tensor(torch.ones(2, 2) * 2,
requires_grad=True)
```

- Do a simple math equation:

```
z = 2 * (x * x) + 5 * x
```

- To get the gradient of this operation with respect to *x* i.e. *dz/dx* we can analytically calculate this.

- If all elements of *x* are 2, then we should expect the gradient *dz/dx* to be a (2, 2) shaped tensor with 13-values.
- However, first we have to run the .backwards() operation to compute these gradients.
- To compute gradients, we need to compute them with respect to something.
- In this case, we can supply a (2,2) tensor of 1-values to be what we compute the gradients against – so the calculation simply becomes *d/dx*:

```
z.backward(torch.ones(2, 2)*2)
print(x.grad)
        tensor([[ 13.,  13.],
                [ 13.,  13.]])
```

PyTorch Neural Network

- *torch.nn.module* is used to define a neural network
- Example: NN with 3 fully connected layers
  - Using RELU activation for 1st and 2nd layer
  - Input to 1st FC layer: 256 features
  - Input to 2nd FC layer: 120 values
  - Input to 3rd FC layer: 84 values
- Define only forward prop. : backward prop is automatically derived from it

```
import torch
import torch.nn as nn
import torch.nn.functional as F

#Inherit from class nn.Module
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # y = Wx + b
        self.fc1 = nn.Linear(256, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

PyTorch Loss Function

- Loss function:
  - How "far" from the **target** is the **output** of forward propagation (prediction)
- NN provides various loss functions with syntax:
  - *loss = <loss-function>(output, target)*
  - *loss, output* and *target* are Tensors

```
#net is the network previously defined
#input is the input data of the network
net = Net()
input = torch.randn(256) # a dummy input
output = net(input)
#criterion is a Mean-Squared Error loss function
criterion = nn.MSELoss()

target = torch.arange(1, 11)  # a dummy target
#target comes from the labelled dataset
loss = criterion(output, target)

print(loss)
```

PyTorch Backpropagation and weights update

- First reset gradients of the network
- Compute backward prop.
  - It uses *autograd* inside
- Update the weights with SGD:

  *weight = weight - learning_rate * gradient*
- Different optimization algorithms are in *torch.optim*

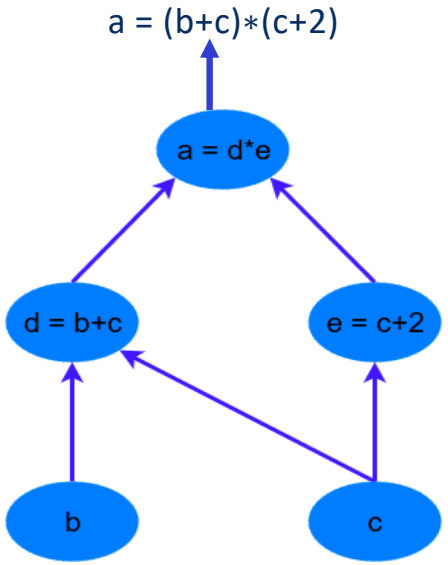
```
# Zeroes the gradient buffer of all parameters
net.zero_grad()

#Backpropagation step
loss.backward()

#Stocastic Gradient Descent weights update
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

Computation graph evaluation approaches

- Computation Graph
  - Forward propagation (use it as it is)
  - Backward propagation (compute gradients)

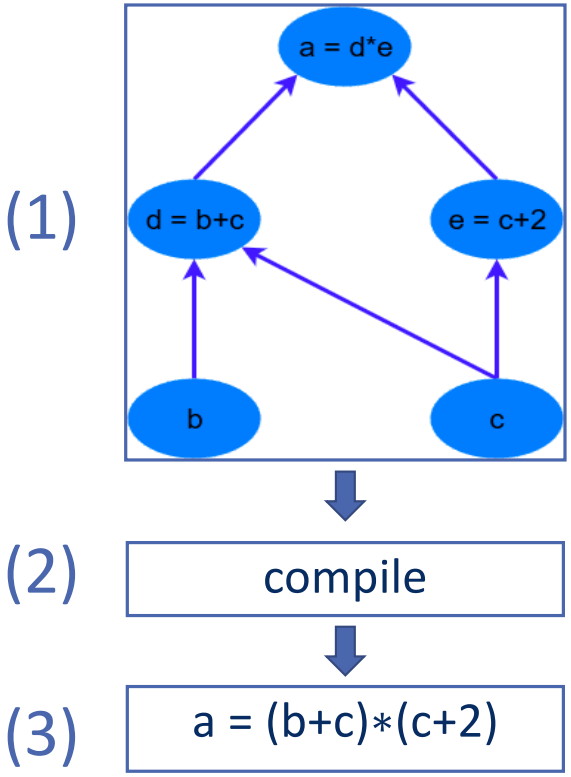
- Two evaluation approaches:
  - **Declarative**: declare all at once, compile, compute
    - TensorFlow, Caffe, Theano
  - **Imperative**: declare and compute each element at runtime
    - PyTorch, Chainer

- Deep Learning Programming Paradigm



Declarative/Symbolic approach

- Declarative approach:
  1. **Declare** the full computation graph in a high-level language
     - (ex. Python operators)
  2. **Compile** it and **optimize** based on full knowledge of the computation
     - Memory management opt, Operations fusion, etc.
     - Compiled computation graph can be run on environments without Python Interpreter like edge devices, mobile, backend devices
  3. **Compute** it on the **computing engine**
     - Separate compute engine can be highly optimized for performance



Declarative framework example: TensorFlow

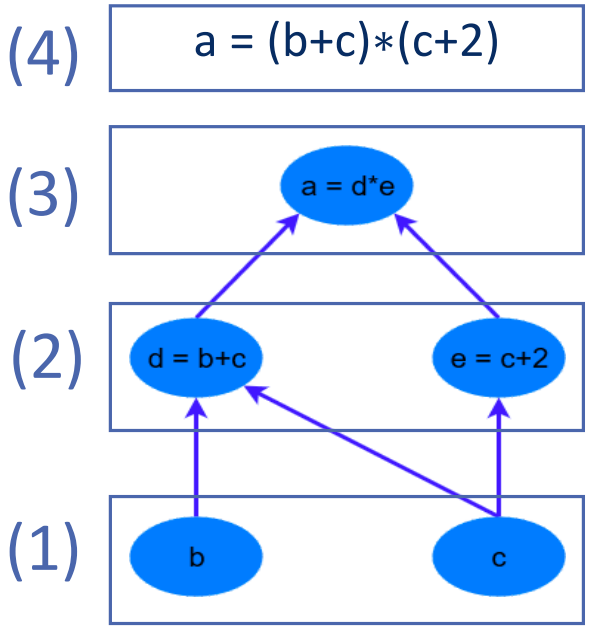
1. Declare:
   - Constants
   - Variables
   - Operators
2. Create session
   - Engine start/end
- Execute engine

```
from __future__ import print_function
import tensorflow as tf

# Basic constant operations (a and b represent the output)
a = tf.constant(2)
b = tf.constant(3)

with tf.Session() as sess:
    print("a=2, b=3")
    print("Addition with constants: %i" % sess.run(a+b))
    print("Multiplication with constants: %i" % sess.run(a*b))

# Basic Operations with variable as graph input
# The value returned by the constructor represents the output
# of the Variable op. (define as input when running session)
# tf Graph input
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)

# Define some operations
add = tf.add(a, b)
mul = tf.multiply(a, b)

# Launch the default graph.
with tf.Session() as sess:
    # Run every operation with variable input
    print("Addition with variables: %i" % sess.run(add, feed_dict={a: 2, b: 3}))
    print("Multiplication with variables: %i" % sess.run(mul, feed_dict={a: 2, b: 3}))
```

Imperative approach

1. **Start** declaring computation graph
2. Execute each single component of the graph: **do not wait** for full graph declaration
3. If more components are added keep computing

- Graph is built on the fly while the program is executed



Eager mode of execution in DL

- **Eager execution** (or eager evaluation): computational graph (the set of steps needed to perform forward or backwards propagation through the network) is build at runtime by the framework, e.g., Pytorch
  - Code that is executing the mathematical operations involved is ultimately a C++ or CUDA kernel
  - Result of each individual operation is immediately transferred to (and accessible from) the Python process as Python process manages the computation graph
  - Debugging is much easier `import pdb; pdb.set_trace()`
  - Good developer experience
- Tensorflow used **graph mode execution** by default in version 1, switched to using eager execution by default in TensorFlow 2

Graph mode of execution in DL

- Computation graph of DL is first defined and compiled
- Pushes the management of the computational graph down to the kernel level (e.g., to a C++ process)
- The intermediate state is not surfaced back to the Python process until after execution is complete; poor developer experience; debugging requires working C++ debugger
- Graph execution is faster than eager
- Graph execution is preferable in production environments due to its better performance
  - Portability; does not require Python interpreter to execute

Declarative vs. Imperative approach comparison

	Declarative	**Imperative**
Productivity		
Debugging		
Static analysis/optimization		

Declarative vs. Imperative approach comparison

	compile time	runtime
	Declarative	**Imperative**
Productivity	-	+
Debugging	-	+
Static analysis/optimization	+	-

Co-evolution of execution modes in DL frameworks

- TensorFlow, which started as a graph framework, now supports eager.
- PyTorch, which started as an eager framework, now supports graph—`Torchscript`

PyTorch Performance – Just In Time Compilation

Java script uses JIT

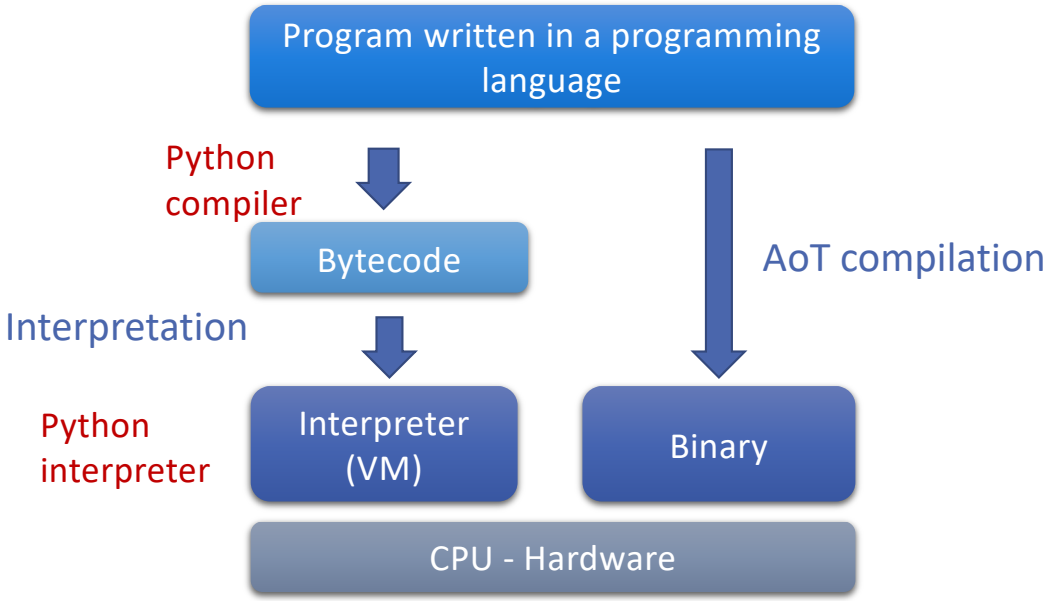
Some Definitions

- Assembly code is processor architecture specific
  - Assembly language is a low-level programming language that must be converted into machine code using software called an assembler

- Machine code is either a set of instructions in machine language or binary format which can be directly executed by the CPU

From language to binary execution

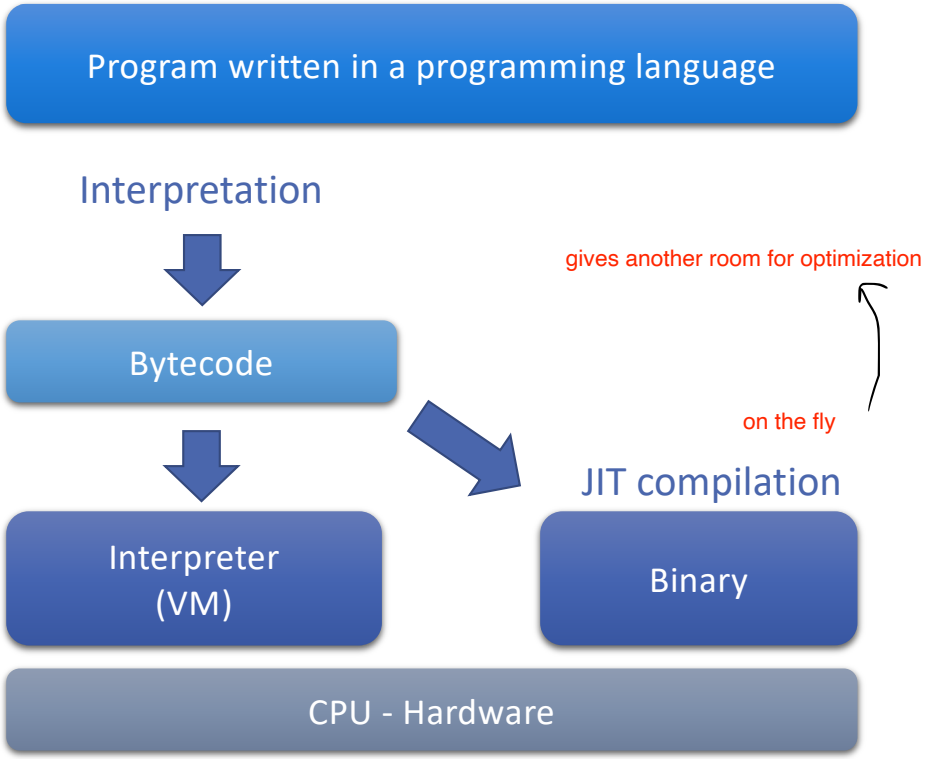
- **Interpretation:**
  1. Compile to bytecode
     - Bytecodes are platform-independent
     - Non-runnable code, needs a virtual machine to run
     - Python: collection of opcode and oparg
  2. Interpret in a virtual machine
     - Ex. Python, Java, Javascript
     - Virtual machine takes care of the differences between bytecodes for different platforms
- **Ahead of Time** compilation:
  1. Compile to binary code
  2. Execute in hardware
     - Ex. C, C++, Fortran
- Is there a third approach?



Just in time compilation

- JIT compilation
  1. Compile to bytecode
  2. Two options:
     1. Default: Execute in VM
     2. **JIT:** Compile to binary on the fly and execute on hardware

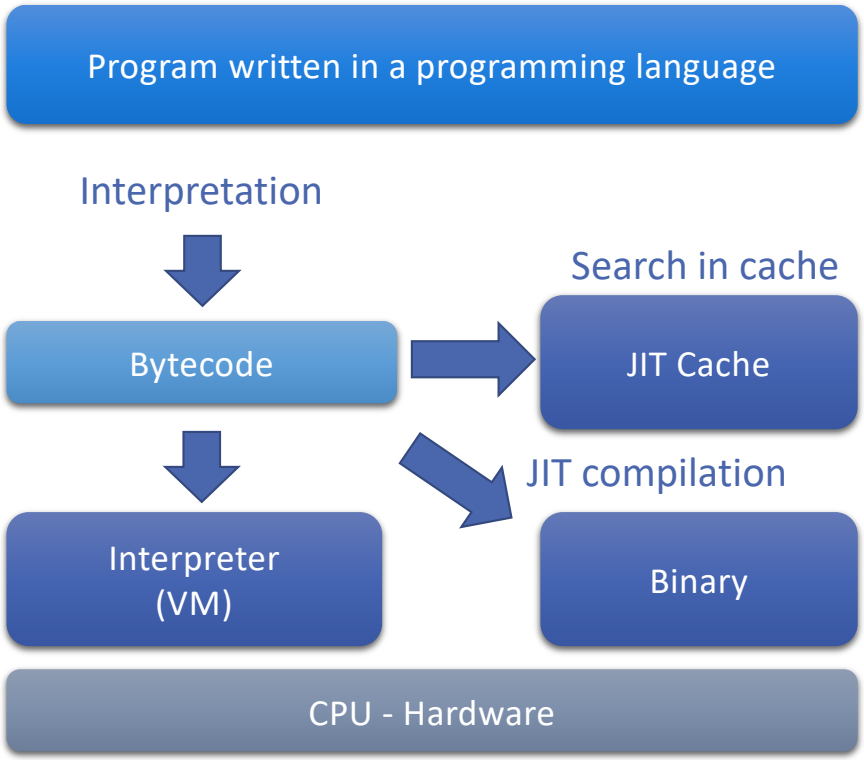
- When is it convenient to do JIT?
  - For functions that are going to be executed many times (ex. Chrome browser V8 engine Javascript JIT)



JIT Binary Caching

- We can keep a cache of already JIT compiled functions

1. Compile to bytecode
2. Search in JIT cache:
   1. If found use the binary
   2. If not found compile to binary
3. Execute



Python JIT Compilation - Numba

- Not supported by CPython interpreter but supported on other projects
- **Numba:**
  - Generates optimized code using the LLVM (low level virtual machine) compiler
  - Example: use the @jit decorator to compile at 1st execution
  - Can compile at:
    - Import time
    - Runtime
    - Statically
    - http://numba.pydata.org/numba-doc/0.37.0/user/jit.html
- What is the difference between Numba @jit and a CPython extension?

```
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when
# function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

https://numba.readthedocs.io/en/stable/user/5minguide.html

Numba vs Cython: How to Choose

- Numba uses LLVM to power Just-In-Time compilation of array-oriented Python code.
- Using Numba is usually about as simple as adding a decorator to your functions:

```
from numba import jit

@jit
def numba_mean(x):
    total = 0
    for xi in x:
        total += xi
    return total / len(x)
```

- Cython is a general-purpose tool, not just for array-oriented computing, that compiles Python into C extensions.
- To see impressive speedups, you need to manually add types:

```
def cython_mean(double[:] x):
    cdef double total = 0
    for i in range(len(x)):
        total += x[i]
    return total / len(x)
```

Cython is easier to distribute than Numba, which makes it a better option for user facing libraries.

PyTorch JIT Compilation

- Can be used to bring compilation advantages to imperative frameworks:
  - Static analysis
  - Optimization
- Lazy evaluation
  - Compile only when graph needs to be **evaluated**

Building the graph only

JIT compilation and evaluation

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))

prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())

next_h = i2h + h2h
next_h = next_h.tanh()

print(next_h)
```

PyTorch Performance – Just In Time Compilation

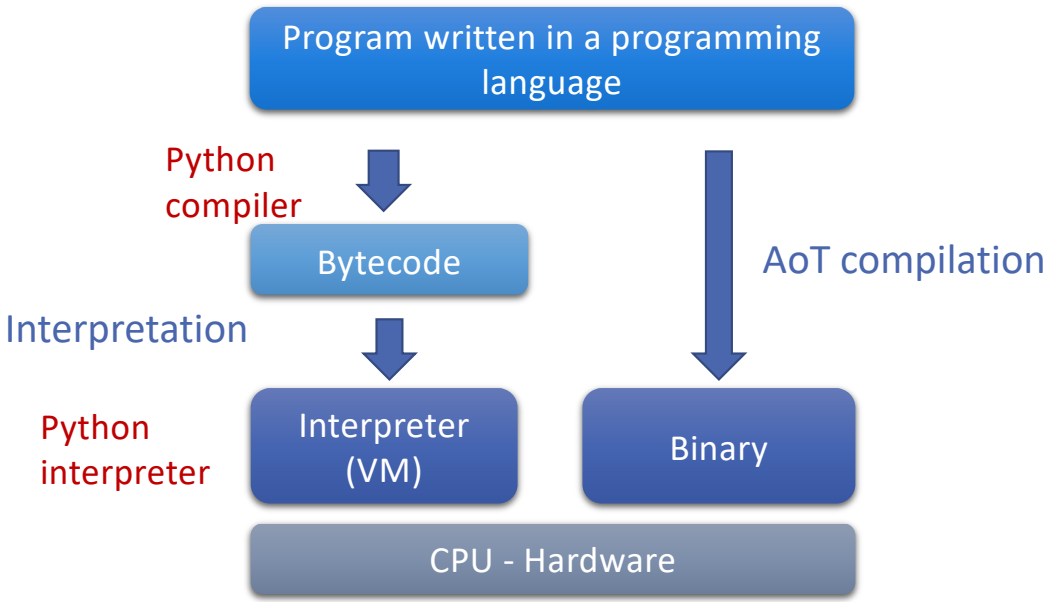
Some Definitions

- Assembly code is processor architecture specific
  - Assembly language is a low-level programming language that must be converted into machine code using software called an assembler

- Machine code is either a set of instructions in machine language or binary format which can be directly executed by the CPU

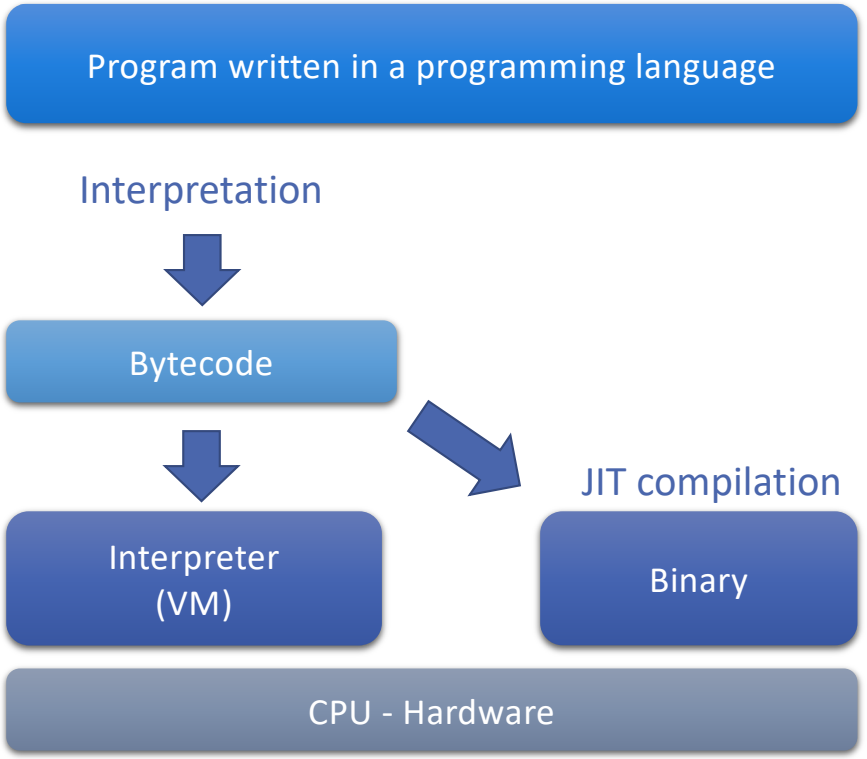
From language to binary execution

- **Interpretation:**
  1. Compile to bytecode
     - Bytecodes are platform-independent
     - Non-runnable code, needs a virtual machine to run
     - Python: collection of opcode and oparg
  2. Interpret in a virtual machine
     - Ex. Python, Java, Javascript
     - Virtual machine takes care of the differences between bytecodes for different platforms
- **Ahead of Time** compilation:
  1. Compile to binary code
  2. Execute in hardware
     - Ex. C, C++, Fortran
- Is there a third approach?



Just in time compilation

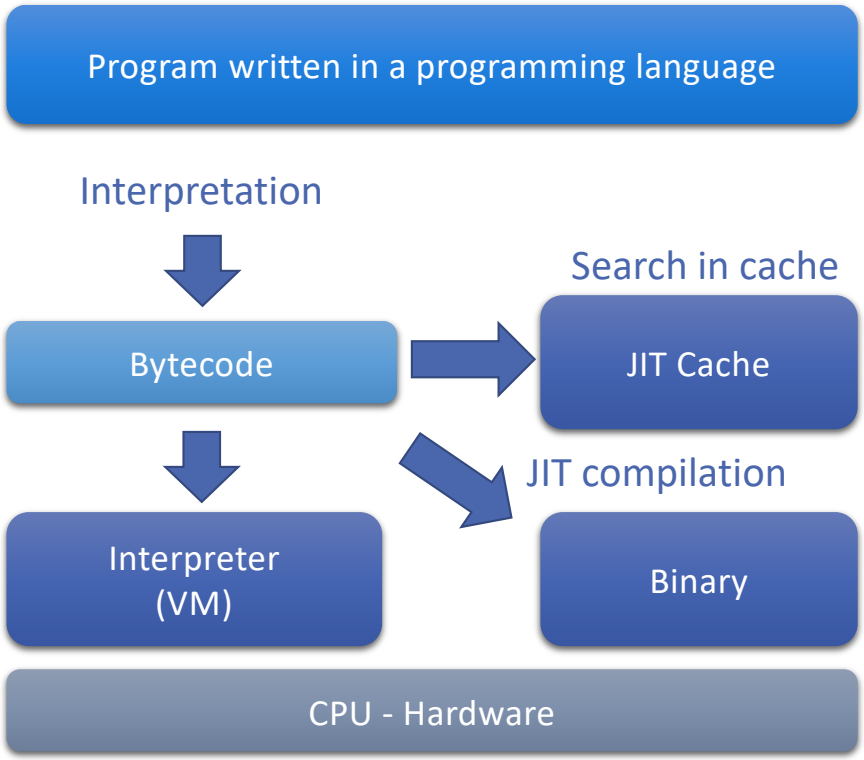
- JIT compilation
  1. Compile to bytecode
  2. Two options:
     1. Default: Execute in VM
     2. **JIT:** Compile to binary on the fly and execute on hardware
- When is it convenient to do JIT?
  - For functions that are going to be executed many times (ex. Chrome browser V8 engine Javascript JIT)
  - JIT allows observing the program's behavior while running and optimizing it on the fly (dynamic analysis and then conversion to machine code)



JIT Binary Caching

- We can keep a cache of already JIT compiled functions

1. Compile to bytecode
2. Search in JIT cache:
    1. If found use the binary
    2. If not found compile to binary
3. Execute



Python JIT Implementation

- Not supported by CPython, the standard Python implementation
- Supported by other projects such as:
  - Pypy
  - Numba
- Cons:
  - Overhead of the compilation process
  - The degree of performance improvements varies depending on the code and usage patterns

What is PyPY?

- PyPY is:
  - An implementation of Python in Python
  - A very flexible compiler framework (with some features that are especially useful for implementing interpreters)
- PyPY grew out of a desire to modify and extend the implementation of Python
  - Increase performance (JIT compilation and better garbage collectors)
  - Ease of porting to new platforms like the JVM or to low-memory situations
  - Add expressiveness (stackless-style routine, logic programming)
- Python also has other tools and libraries like **Numba** and **Cython** that can help optimize Python code by **compiling selected parts** of it into machine code

How does JIT work in the context of PyPy

- **Parsing and Analysis:** PyPy parses Python code, creates an AST, and performs initial analysis.
- **JIT Compilation:** PyPy's JIT compiler dynamically generates machine code based on runtime execution patterns.
- **Optimizations:** The JIT compiler optimizes machine code by inlining functions, optimizing loops, and eliminating branches.
- **Execution:** CPU executes generated machine code, offering a significant performance boost.
- **Profiling and Deoptimization:** JIT compiler monitors program behavior, deoptimizing when necessary for efficiency.

- You can turn on or off JIT when using PyPy
  - **pypy –jit off**

Python JIT Compilation - Numba

- Not supported by CPython interpreter but supported on other projects
- **Numba:**
  - Generates optimized code using the LLVM (low level virtual machine) compiler
  - Example: use the @jit decorator to compile at 1st execution
  - Can compile at:
    - Import time
    - Runtime
    - Statically
    - http://numba.pydata.org/numba-doc/0.37.0/user/jit.html
- What is the difference between Numba @jit and a CPython extension?

```
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when
# function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

https://numba.readthedocs.io/en/stable/user/5minguide.html

Numba vs Cython: How to Choose

- Numba uses LLVM to power Just-In-Time compilation of array-oriented Python code.
- Using Numba is usually about as simple as adding a decorator to your functions:

```
from numba import jit

@jit
def numba_mean(x):
    total = 0
    for xi in x:
        total += xi
    return total / len(x)
```

- Cython is a general-purpose tool, not just for array-oriented computing, that compiles Python into C extensions.
- To see impressive speedups, you need to manually add types:

```
def cython_mean(double[:] x):
    cdef double total = 0
    for i in range(len(x)):
        total += x[i]
    return total / len(x)
```

Cython is easier to distribute than Numba, which makes it a better option for user-facing libraries.

PyTorch JIT Compilation

- Can be used to bring compilation advantages to imperative frameworks:
  - Static analysis
  - Optimization
- Lazy evaluation
  - Compile only when graph needs to be **evaluated**

Building the graph only

JIT compilation and evaluation

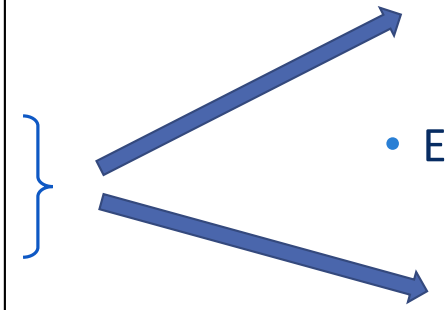
```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))

prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())

next_h = i2h + h2h
next_h = next_h.tanh()

print(next_h)
```

JIT Compilation optimization: Fusion

- **Fusion** can significantly improve performance reducing the number of operations

- PyTorch code:

```
x = Variable(torch.randn(1, 10))
y = Variable(torch.randn(1,10))

xy = torch.mm(x.t(), y)
xy = xy * 100
xy = xy + 10

# Apply fusion then execute
print(xy)
```



- Example C implementation:

```
for (i = 0; i < x_rows; ++i)
  for (j = 0; j < y_cols; ++j)
    for (k = 0; k < y_rows; ++k)
      xy[i][j] += x[i][k] * y[k][j];

for (i = 0; i < x_rows; ++i)
  for (j = 0; j < y_cols; ++j)
    xy[i][j] = xy[i][j] * 100;

for (i = 0; i < x_rows; ++i)
  for (j = 0; j < y_cols; ++j)
    xy[i][j] = xy[i][j] + 10;
```

- Example C implementation with **Fusion:**

```
for (i = 0; i < x_rows; ++i)
  for (j = 0; j < y_cols; ++j) {
    for (k = 0; k < y_rows; ++k)
        xy[i][j] += x[i][k] * y[k][j];
    xy[i][j] = xy[i][j] * 100 + 10;
  }
```

JIT Compilation optimization: OOO and work scheduling

- **Out of order execution** and automatic **work scheduling**
- PyTorch code:

```
from torch.autograd import Variable

x = Variable(torch.randn(1000,1000))
y = Variable(torch.randn(1000,1000))
z = Variable(torch.randn(1000,1000))

xy = torch.mm(x, y)
xz = torch.mm(x, z)

xy = xy + 100
xz = xz + 1000

# Reorder, fuse, and execute on
different devices (GPUs or cores)
print(xy + xz)
```

- Reorder, fuse, and schedule on different devices

```
for (i = 0; i < x_rows; ++i)
  for (j = 0; j < y_cols; ++j) {
    for (k = 0; k < y_rows; ++k)
        xy[i][j] += x[i][k] * y[k][j];
    xy[i][j] = xy[i][j] + 100;
  }
```

```
for (i = 0; i < x_rows; ++i)
  for (j = 0; j < z_cols; ++j) {
    for (k = 0; k < z_rows; ++k)
        xz[i][j] += x[i][k] * z[k][j];
    xz[i][j] = xz[i][j] + 1000;
  }
```

Using the PyTorch JIT compiler

- Work in progress… try only on latest master branch
- Uses an IR (JIT trace)
- Also uses caching:
  - stores previously compiled functions

- https://github.com/pytorch/pytorch/blob/master/torch/csrc/jit/README.md

- PyTorch JIT use example

```
import torch

@torch.jit.trace
def foo(x,y):
  xy = x + y
  return xy

x = torch.randn(1000,1000)
y = torch.randn(1000,1000)

xy = foo(x,y)

print(xy)
```

PyTorch Performance – Profiling

A Different Mental Model Required

GPU PERFORMANCE TUNING

CPU

Optimized for single thread performance
- Majority of chip area is control logic & caches

Complex and deep out-of-order pipelines
- Extract instruction level parallelism

The brain
- Job is to keep the accelerator busy

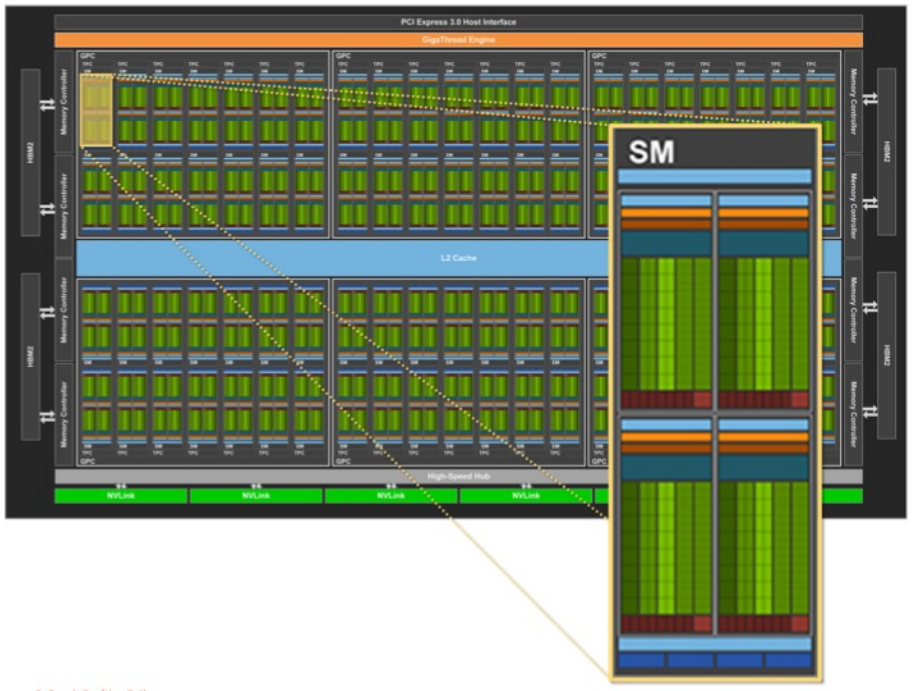
GPU

Optimized for throughput of data-parallel problems
- Majority of chip area is functional units

Simple, relatively slow in-order pipelines
- Achieves much higher total throughput

Accelerator attached via PCIe
- Order of magnitude faster but off to the side

NVIDIA Volta GPU Design



NVIDIA Volta V100 GPU

Composed of Streaming Multiprocessors (SMs)

Volta V100: 80x SMs
Ampere A100: 108 SMs

DGX A100 with 8 GPUs:
864 SMs vs 128 CPU cores

Streaming Multiprocessor Design

Streaming Multiprocessor

64x FP32 units

64x INT, 32x FP64, 32x LD/ST

8x Tensor Cores

5120 (6920 ON A100) FP32 EXECUTION UNITS PER GPU

Common Pitfalls

- Excessive CPU/GPU interactions – e.g., for loop launching GPU operations
  - Dominated by launch overheads
- Short GPU kernel durations – e.g., small inputs
  - Need enough data to feed 10s of thousands of threads
- CPU overheads and I/O bottlenecks are starving the GPU
  - Small operations on the CPU can quickly become dominant
- Framework inefficiencies
  - E.g., unnecessary copies and hidden CPU-side overheads

Visibility is key to understand what is going on and optimize your code

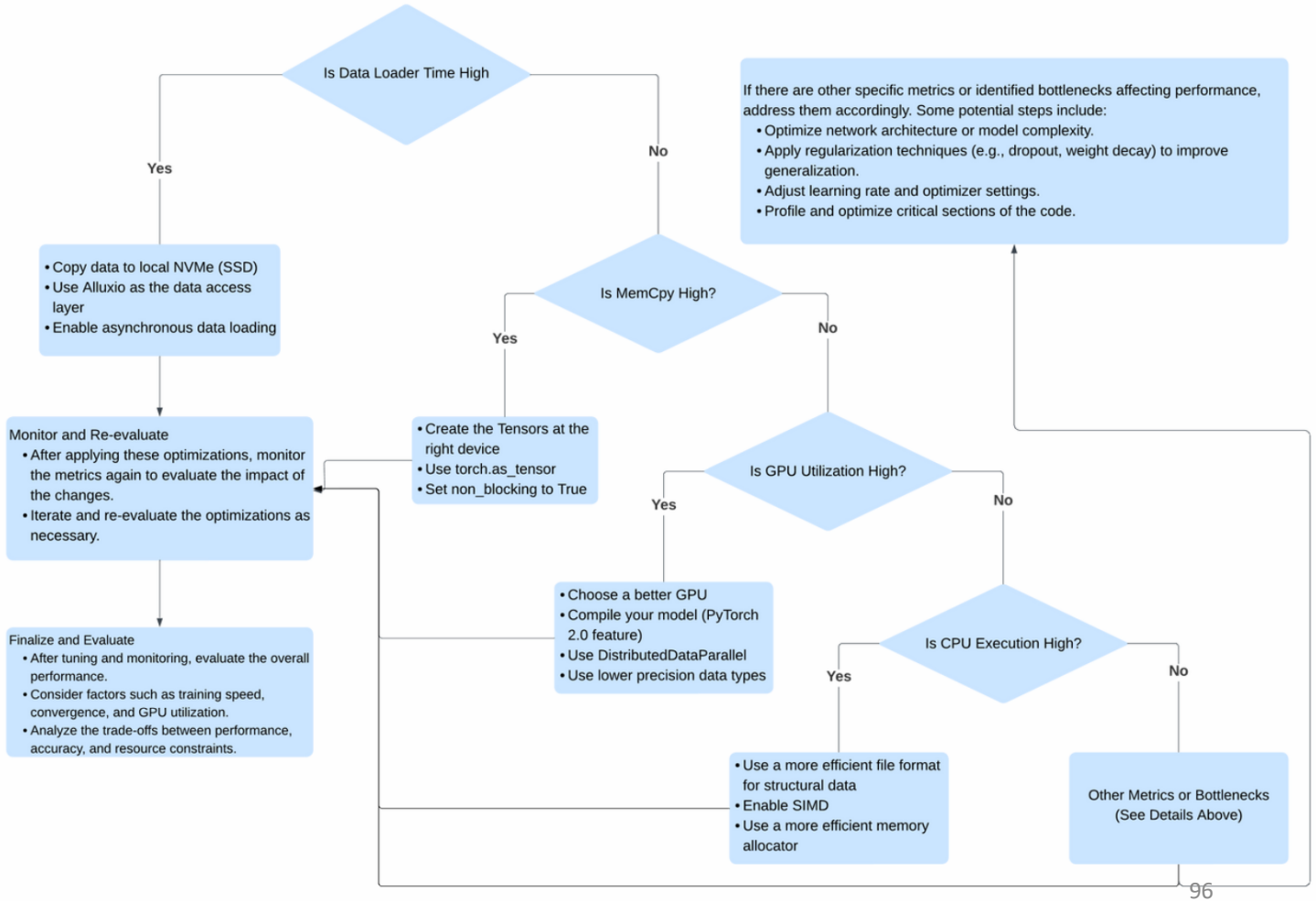
Profiling objectives

- **Measure** and identify critical sections and resource bottlenecks
  - CPU, GPU, Memory, Network, I/O (disk)
- Then**, Tune** the model based on the measured and profiled data

- **What can be profiled?** Almost everything with the right tools…
  - Hardware activity: performance counters
  - Operating system (perf)
  - Memory operations (perf, *valgrind*)
  - Libraries
  - Applications
  - Parallel Distributed Applications (MPI profilers…)

A General Performance Tuning Guide

A decision tree that shows a structured process for tuning PyTorch training based on key metrics such as GPU utilization, CPU execution, data loader time, and memory copy (memcpy).



Reference: PyTorch Model Training Performance Tuning Ebook by Alluxio

Profiling and Tracing Techniques Review

- **Counting** (Deterministic):
  - Count every time a hardware/software event happens (ex. memory load, function call)
  - Report a table of events count
- **Sampling** (Indeterministic: statistical effect):
  - Interrupt the application at **regular intervals** (sampling frequency) and increment a counter associated with the instruction that was interrupted
  - Compute a histogram associating samples to lines of code
  - Can be used to statistically **infer** the relative time in each part of the code
- **Tracing** (Deterministic):
  - Record every time a hardware/software event happens and also the time at which it happens (timestamp)
  - Report a table of relative time spent in each event

Profiling/Tracing techniques Overhead comparison

Counting:	Sampling:	Tracing:

• Mem. footprint: a counter for each (software/hardware) event
  • **low**
	• Mem. footprint: state of the program (instruction counter minimum) at each interval
	  • **medium** (depends on sampling frequency and state size)
		• Mem. footprint: event type + timestamp at each (software/hardware) event
		  • **high**

Identifying Bottlenecks

- You need to identify bottlenecks in the system before optimizing it.
  - Helps focus the optimization efforts on areas with the biggest impact on performance
- Bottlenecks can vary depending on several factors:
  - Size of the dataset
  - Complexity of the model
  - Hardware being used

- Examples:
  - Large dataset -> data loading step could be a bottleneck
  - Model is complex -> training steps could be a bottleneck
  - Code not using GPU -> CPU could be a bottleneck
  - Code using GPU -> bottleneck could be GPU memory or bandwidth between CPU and GPU

How to identify bottlenecks

- Traditional Command Line Tools
  - Helpful in monitoring PyTorch training and identifying bottlenecks
  - Easy to use
  - They can be accessed from any terminal
  - Can monitor various metrics: CPU usage, GPU usage, memory usage, and I/O traffic
- Visualization and tracing tools
  - Examples: Tensorboard, PyTorch Profiler Chrome tracing visualization, weights and biases, Flame Graph visualization, etc.
- Examples:
  - Large dataset -> data loading step could be a bottleneck
  - Model is complex -> training steps could be a bottleneck
  - Code not using GPU -> CPU could be a bottleneck
  - Code using GPU -> bottleneck could be GPU memory or bandwidth between CPU and GPU

Common Profiling Command Line tools

- Most used command-line tools for monitoring resource usage:

Tool	Description
nvidia-smi	This tool provides information about GPU utilization, memory usage, and other metrics related to the NVIDIA GPU
htop	It is a command-line tool that hierarchically displays system processes and provides insights into CPU and memory usage
iotop	With this tool, you can monitor I/O usage by displaying I/O statistics of processes running on your system
gpustat	It is a Python-based user-friendly command-line tool for monitoring NVIDIA GPU status.
nvtop	Similar to nvidia-smi, nvtop displays real-time GPU usage and other metrics in a user-friendly interface.
py-spy	It is a sampling profiler for Python that helps identify performance bottlenecks in your code.
strace	This tool allows you to trace system calls made by a program, providing insights into its behavior and resource usage.

Python profiling tools

- **PyTorch Profiler**
- ***profile:*** pure python module: higher overhead
- ***cProfile*****:** CPython extension modules that traces the execution of Python programs, collecting information on the functions and primitives used:
  - Number of calls
  - Total time (time spent in the function/primitive, excluding nested calls)
  - Cumulative time (time including nested calls)
  - Call graph

  cProfile is the C implementation of the profile interface
- ***pstats***: a module that provides analysis methods for the data collected by the profilers

PyTorch Profiler

- Slides adopted from:

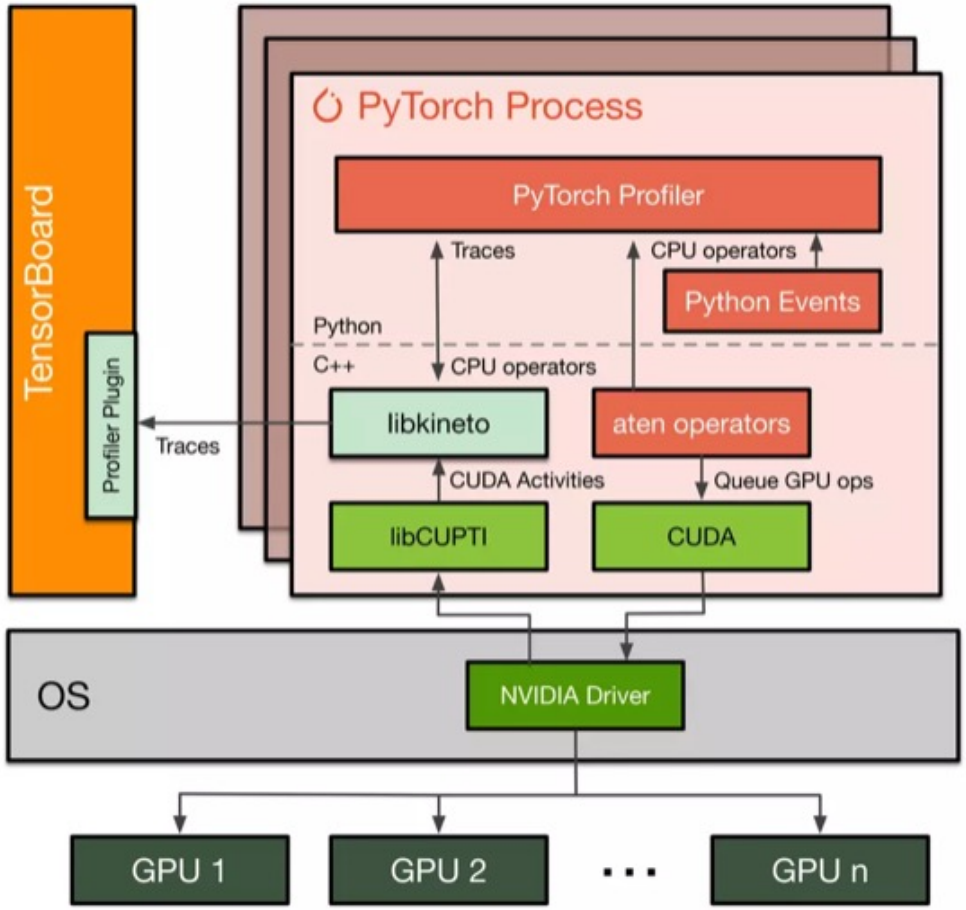
  GEETA CHAUHAN

  PYTORCH PARTNER ENGINEERING

  META AI

PyTorch Profiler

- Contributed by Microsoft & Facebook
  - PyTorch and GPU-level information
  - Automatic bottleneck detection
  - Actionable performance recommendations
  - Data scientist-friendly lifecycle and tools
  - TensorBoard Plugin – Chrome traces visualization
  - Kineto library- built on CUPTI (NVIDIA CUDA Profiling Tools Interface)
  - Easy-to-use Python API
  - VS Code integration



CUPTI: CUDA Profiling Tools Interfaces
ATEN: Short of "A Tensor Library", is a library on top all other Python and C++ interfaces in PyTorch are built
Libkineto: an in-process profiling library integrated with the PyTorch Profile

Profiling API: Basic Usage

```
import torch.profiler

with profiler.profile(..) as prof:
  <code to profile>

# Print results on console
print(prof.key_averages().table(..))
```

https://pytorch.org/tutorials/recipes/recipes/profiler.html

```
import torch
import torchvision.models as models
import torch.profiler as profiler

model = models.resnet18()
inputs = torch.randn(5, 3, 224, 224)

with profiler.profile(record_shapes=True) as prof:
   with profiler.record_function("model_inference"):
      model(inputs)

print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

```
# ------------------------  ------------  ----------  ------------  ---------
# Name                      Self CPU total   CPU total  CPU time avg  # Calls
# ------------------------  ------------  ----------  ------------  ---------
# model_inference           3.541ms         69.571ms    69.571ms      1
# conv2d                    69.122us        40.556ms    2.028ms       20
# convolution               79.100us        40.487ms    2.024ms       20
# _convolution              349.533us       40.408ms    2.020ms       20
# mkldnn_convolution        39.822ms        39.988ms    1.999ms       20
# batch_norm                105.559us       15.523ms    776.134us     20
# _batch_norm_impl_index    103.697us       15.417ms    770.856us     20
# native_batch_norm         9.387ms         15.249ms    762.471us     20
# max_pool2d                29.400us        7.200ms     7.200ms       1
# max_pool2d_with_indices   7.154ms         7.170ms     7.170ms       1
# ------------------------  ------------  ----------  ------------  ---------
```

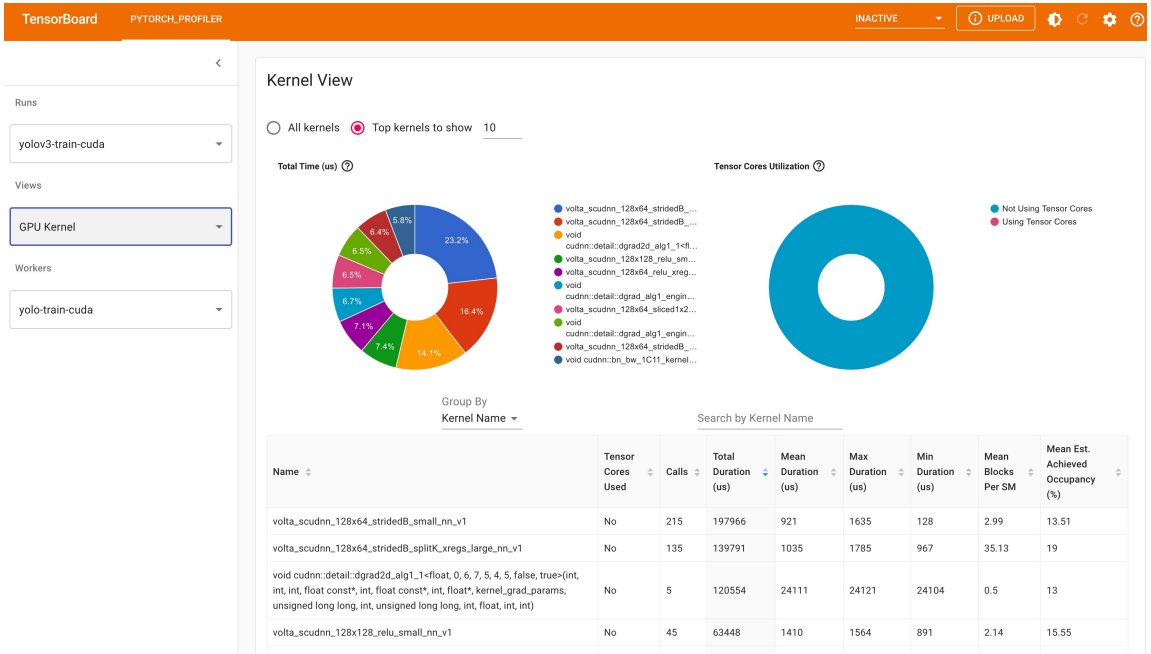
Profiling API: Tensorboard Plugin

```
$PIP INSTALL TORCH-TB-PROFIILER

import torch.profiler

# Export to tensorboard using
# on_trace_handler option
handler=
 tensorboard_trace_handler(..)
with profiler.profile(
 on_trace_ready=handler
) as prof:
 ...
```

```
import torch
import torchvision.models as models
import torch.profiler as profiler

model = models.resnet18()
inputs = torch.randn(5, 3, 224, 224)

with profiler.profile(
  record_shapes=True,
  on_trace_ready=torch.profiler.tensorboard_
trace_handler('results')
) as prof:
   model(inputs)

print(prof.key_averages().table(sort_by=
"cpu_time_total", row_limit=10))
```

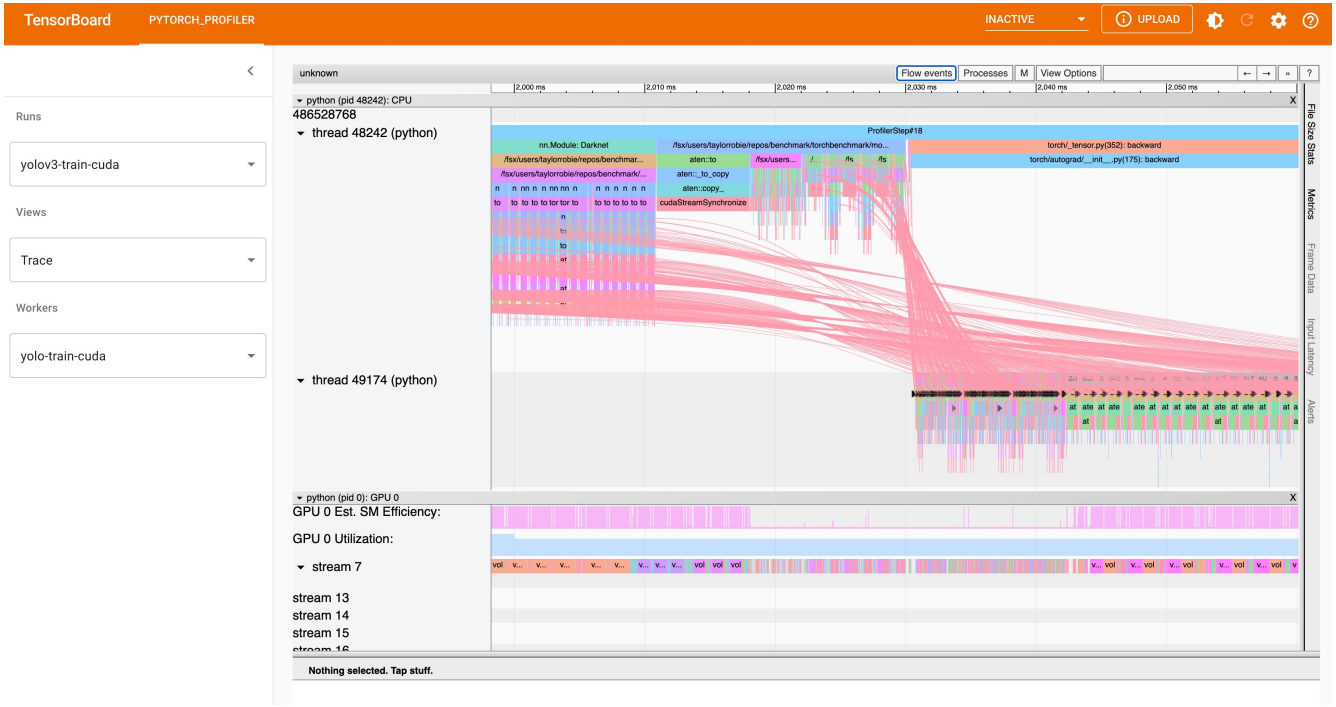
Profiling API: Tensorboard Plugin

```
$PIP INSTALL TORCH-TB-PROFIILER

import torch.profiler

# Export to tensorboard using
# on_trace_handler option
handler=
  tensorboard_trace_handler(..)
with profiler.profile(
  on_trace_ready=handler
) as prof:
  . . .
```



Profiling API: Tensorboard Plugin

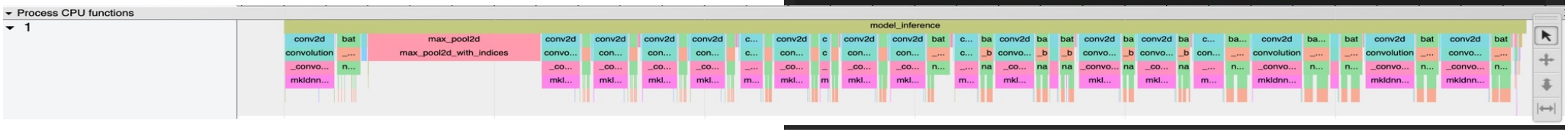
```
$PIP INSTALL TORCH-TB-PROFIILER

import torch.profiler

# Export to tensorboard using
# on_trace_handler option
handler=
  tensorboard_trace_handler(..)
with profiler.profile(
  on_trace_ready=handler
) as prof:
  . . .
```

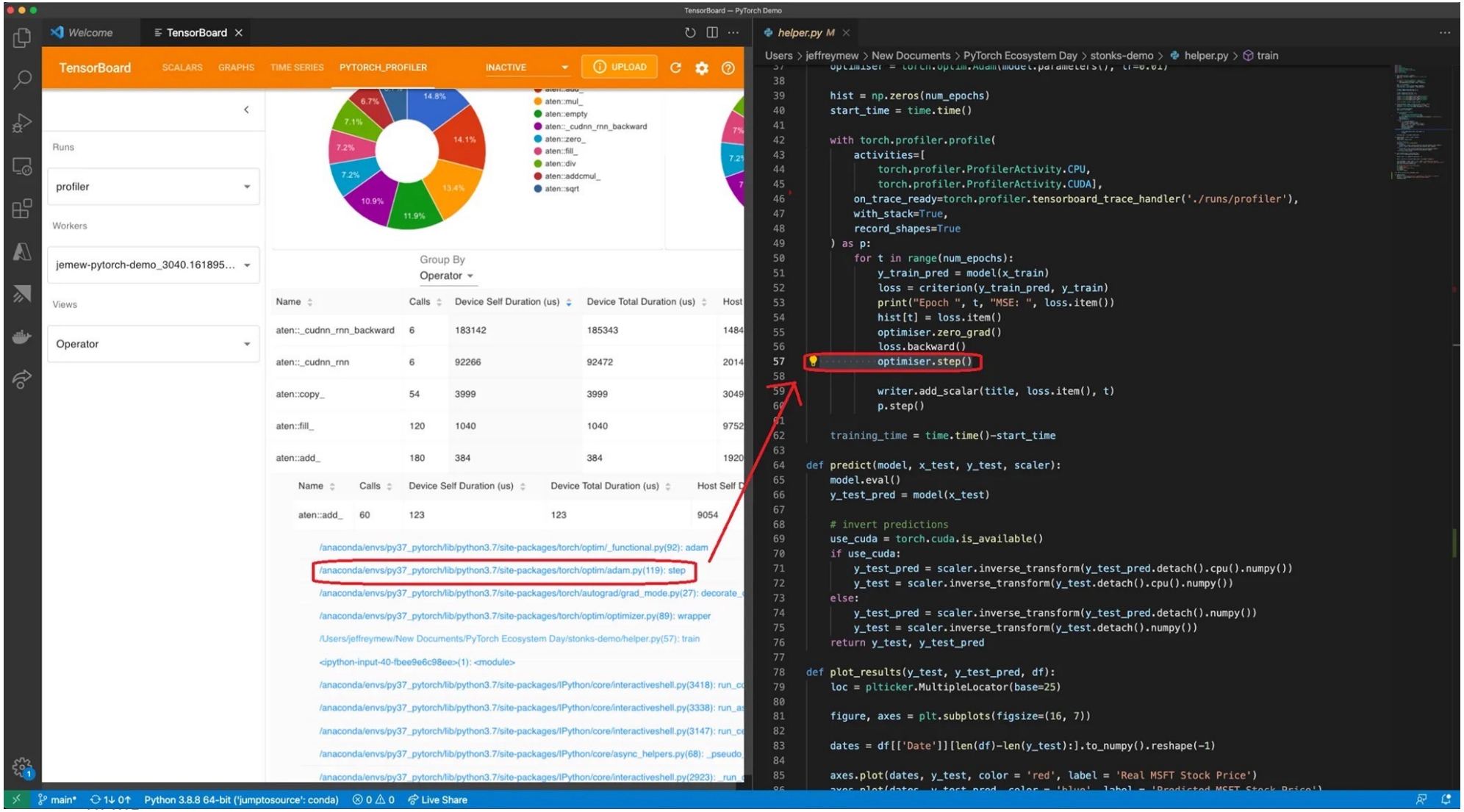


Advanced Options

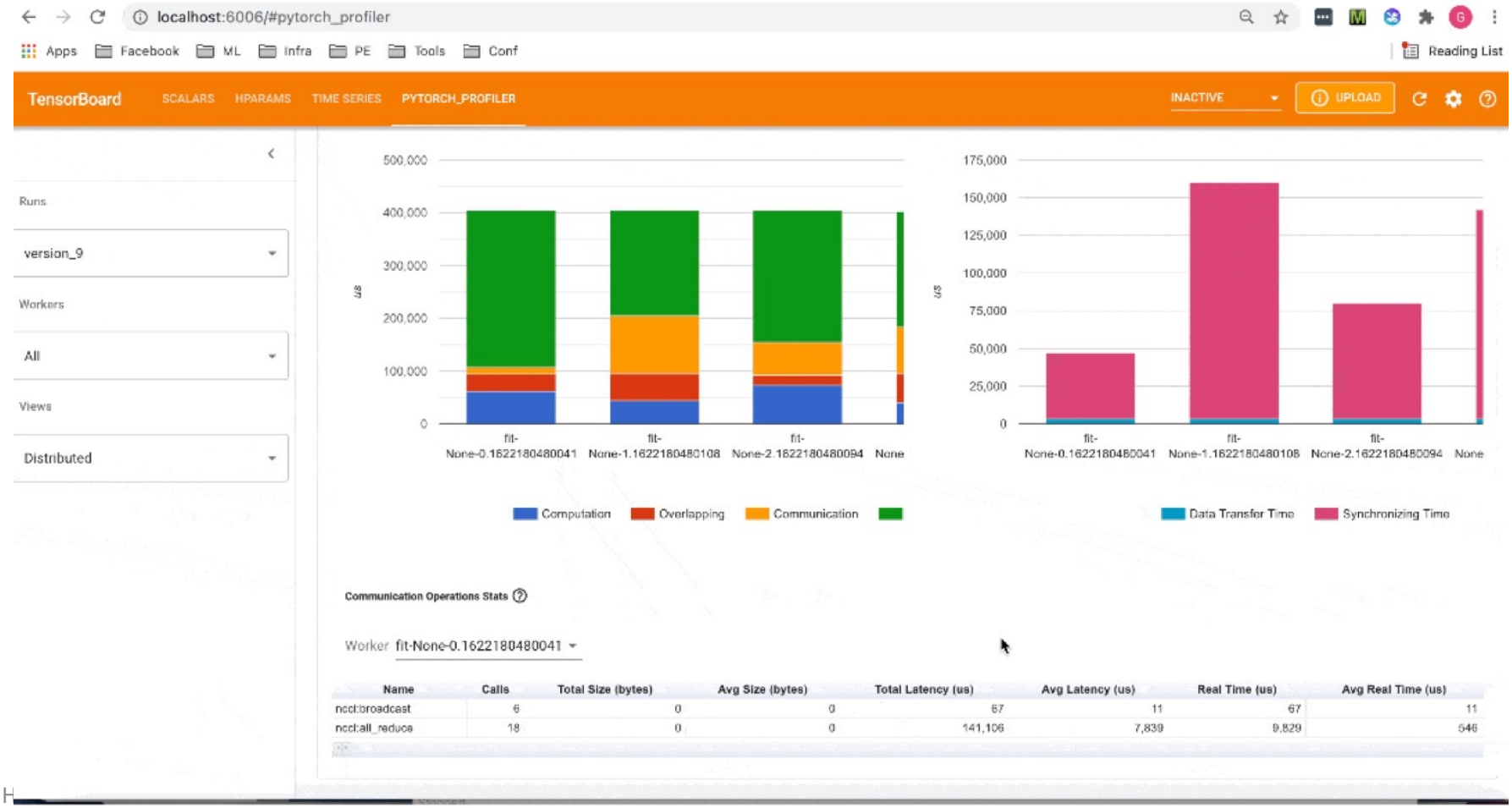
- When to trigger
- How many steps to profile
- Which activities to profile
- Callable handler to save the results
- Extra metadata, eg., shapes, stacks, memory
- Output options: eg.,, Chrome tracing, TensorBoard

```
import torch.profiler as profiler

with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA],
    schedule=torch.profiler.schedule(
        wait=2,
        warmup=3,
        active=6),
    on_trace_ready=torch.profiler.tensorboard_trace_handler('./result'),
    record_shapes=True,
) as p:
    for step, data in enumerate(trainloader, 0):

        inputs, labels = data[0].to(device=device), data[1].to(device=device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if step + 1 >= 11:
            break
        p.step()

    p.export_chrome_trace(path)
    print(p.key_averages().table(
            sort_by="self_cuda_time_total", row_limit=-1))
```

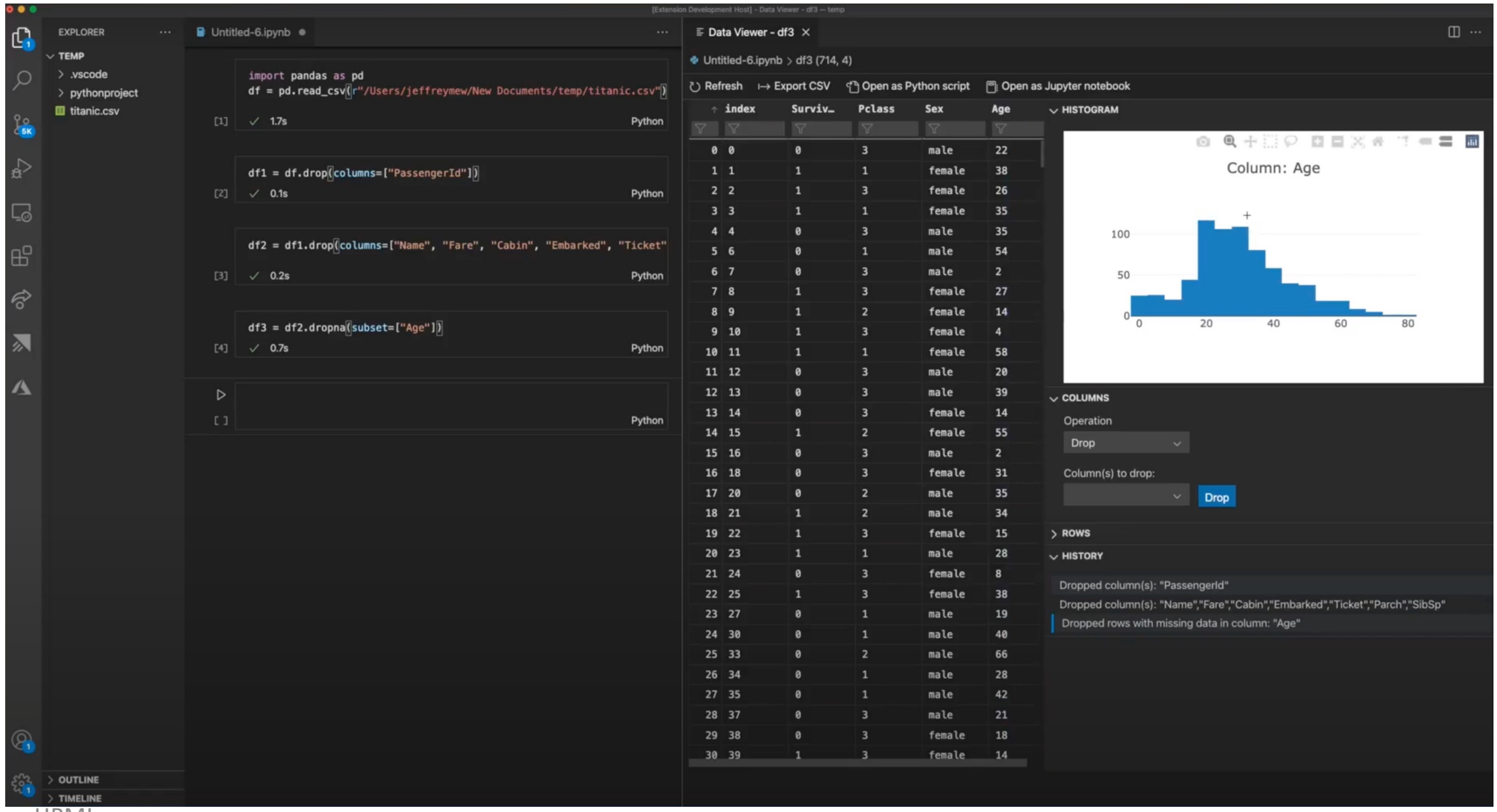



TensorBoard — PyTorch Demo
Welcome
TensorBoard
TensorBoard
SCALARS
GRAPHS
TIME SERIES
PYTORCH_PROFILER
INACTIVE
UPLOAD
Runs
profiler
Workers
jemew-pytorch-demo_3040.161895...
Views
Operator
14.8%
14.1%
13.4%
11.9%
10.9%
7.2%
7.2%
7.1%
6.7%
aten::mul_
aten::empty
aten::_cudnn_rnn_backward
aten::zero_
aten::fill_
aten::div
aten::addcmul_
aten::sqrt
Group By
Operator
Name
Calls
Device Self Duration (us)
Device Total Duration (us)
aten::_cudnn_rnn_backward 6 183142 185343
aten::_cudnn_rnn 6 92266 92472
aten::copy_ 54 3999 3999
aten::fill_ 120 1040 1040
aten::add_ 180 384 384
aten::add_ 60 123 123 9054
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/optim/_functional.py(92): adam
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/optim/adam.py(119): step
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/autograd/grad_mode.py(27): decorate_
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/torch/optim/optimizer.py(89): wrapper
/Users/jeffreymew/New Documents/PyTorch Ecosystem Day/stonks-demo/helper.py(57): train
<ipython-input-40-fbee9e6c98ee>(1): <module>
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(3418): run_co
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(3338): run_as
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(3147): run_co
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/async_helpers.py(68): _pseudo_
/anaconda/envs/py37_pytorch/lib/python3.7/site-packages/IPython/core/interactiveshell.py(2923): _run_
helper.py M
Users > jeffreymew > New Documents > PyTorch Ecosystem Day > stonks-demo > helper.py > train
hist = np.zeros(num_epochs)
start_time = time.time()
with torch.profiler.profile(
activities=[
torch.profiler.ProfilerActivity.CPU,
torch.profiler.ProfilerActivity.CUDA],
on_trace_ready=torch.profiler.tensorboard_trace_handler('./runs/profiler'),
with_stack=True,
record_shapes=True
) as p:
for t in range(num_epochs):
y_train_pred = model(x_train)
loss = criterion(y_train_pred, y_train)
print("Epoch ", t, "MSE: ", loss.item())
hist[t] = loss.item()
optimiser.zero_grad()
loss.backward()
optimiser.step()
writer.add_scalar(title, loss.item(), t)
p.step()
training_time = time.time()-start_time
def predict(model, x_test, y_test, scaler):
model.eval()
y_test_pred = model(x_test)
# invert predictions
use_cuda = torch.cuda.is_available()
if use_cuda:
y_test_pred = scaler.inverse_transform(y_test_pred.detach().cpu().numpy())
y_test = scaler.inverse_transform(y_test.detach().cpu().numpy())
else:
y_test_pred = scaler.inverse_transform(y_test_pred.detach().numpy())
y_test = scaler.inverse_transform(y_test.detach().numpy())
return y_test, y_test_pred
def plot_results(y_test, y_test_pred, df):
loc = plticker.MultipleLocator(base=25)
figure, axes = plt.subplots(figsize=(16, 7))
dates = df[['Date']][len(df)-len(y_test):].to_numpy().reshape(-1)
axes.plot(dates, y_test, color = 'red', label = 'Real MSFT Stock Price')
main*
Python 3.8.8 64-bit ('jumptosource': conda)
Live Share

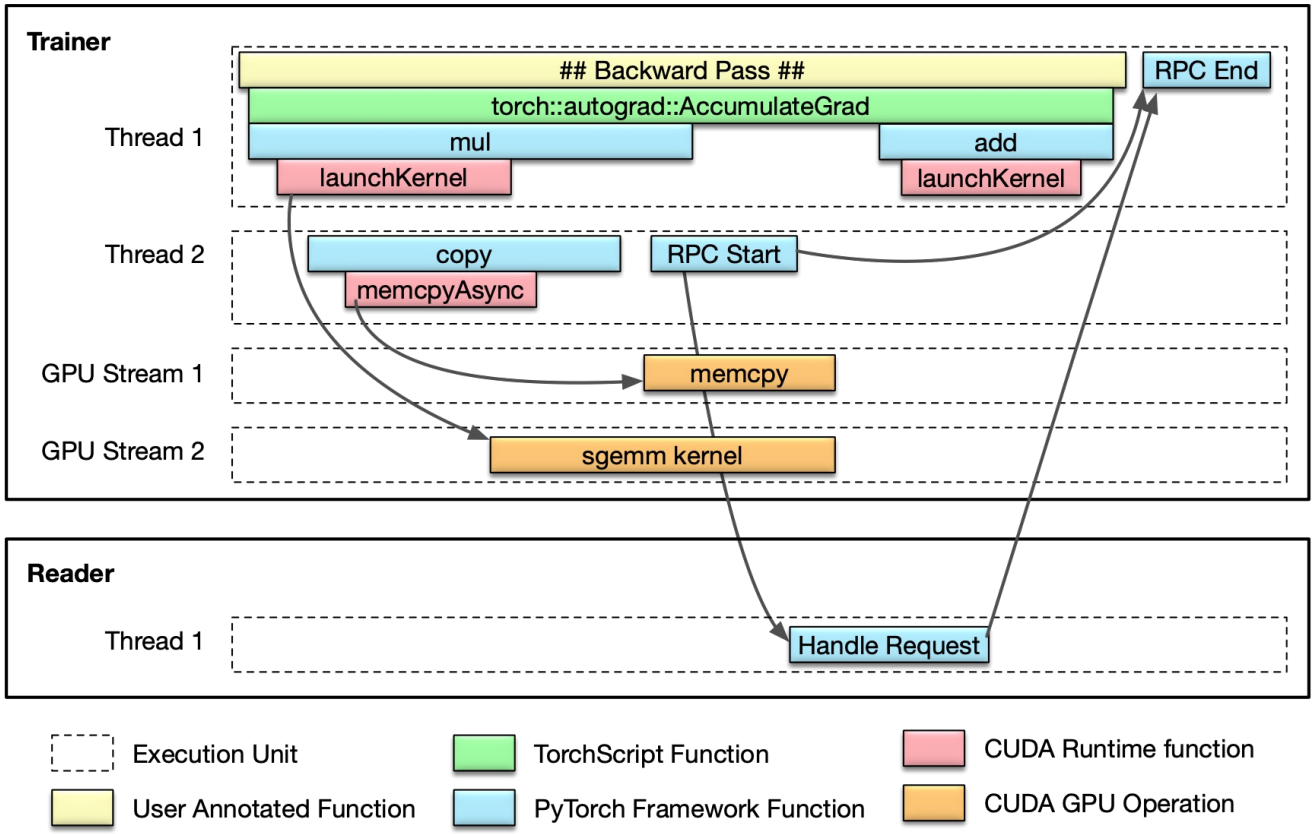
Distributed Training View



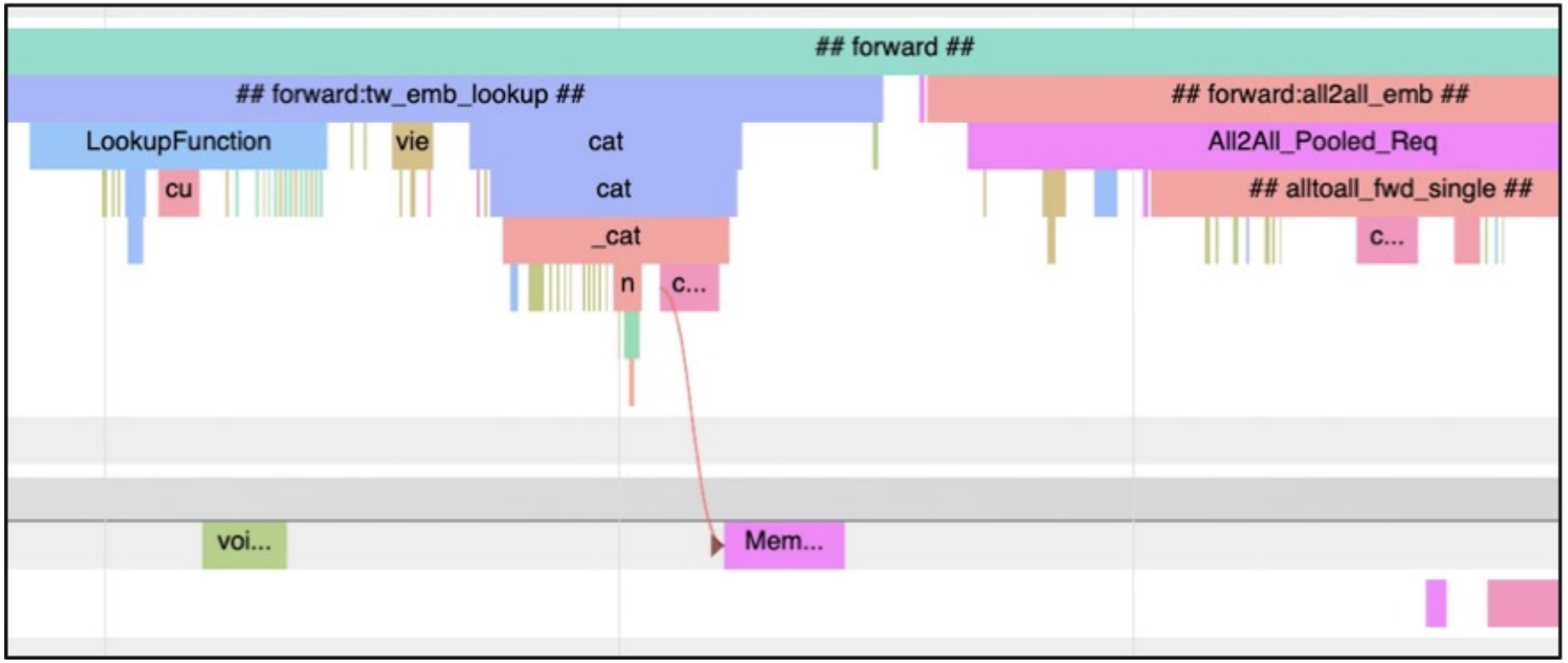
Vscode Integration: Data Wrangler



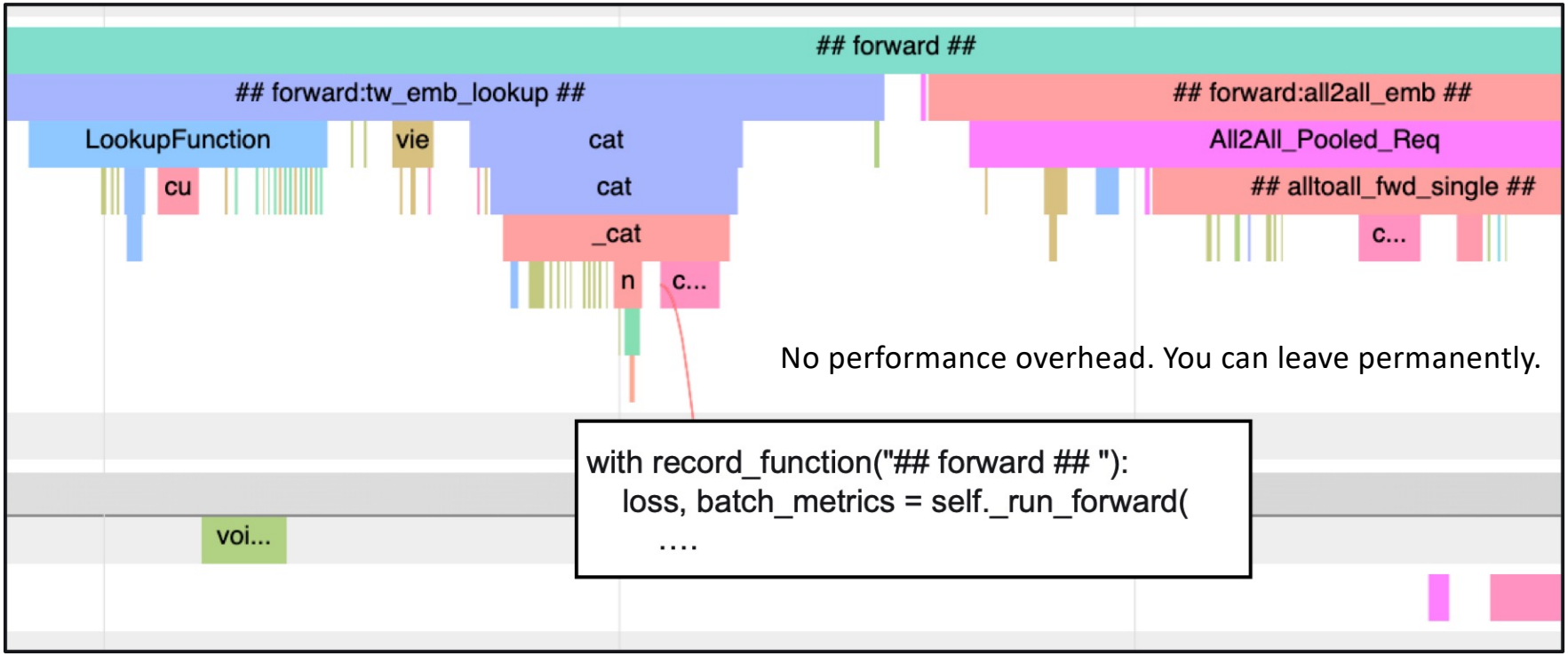
Timeline tracing: CPU and GPU Activities



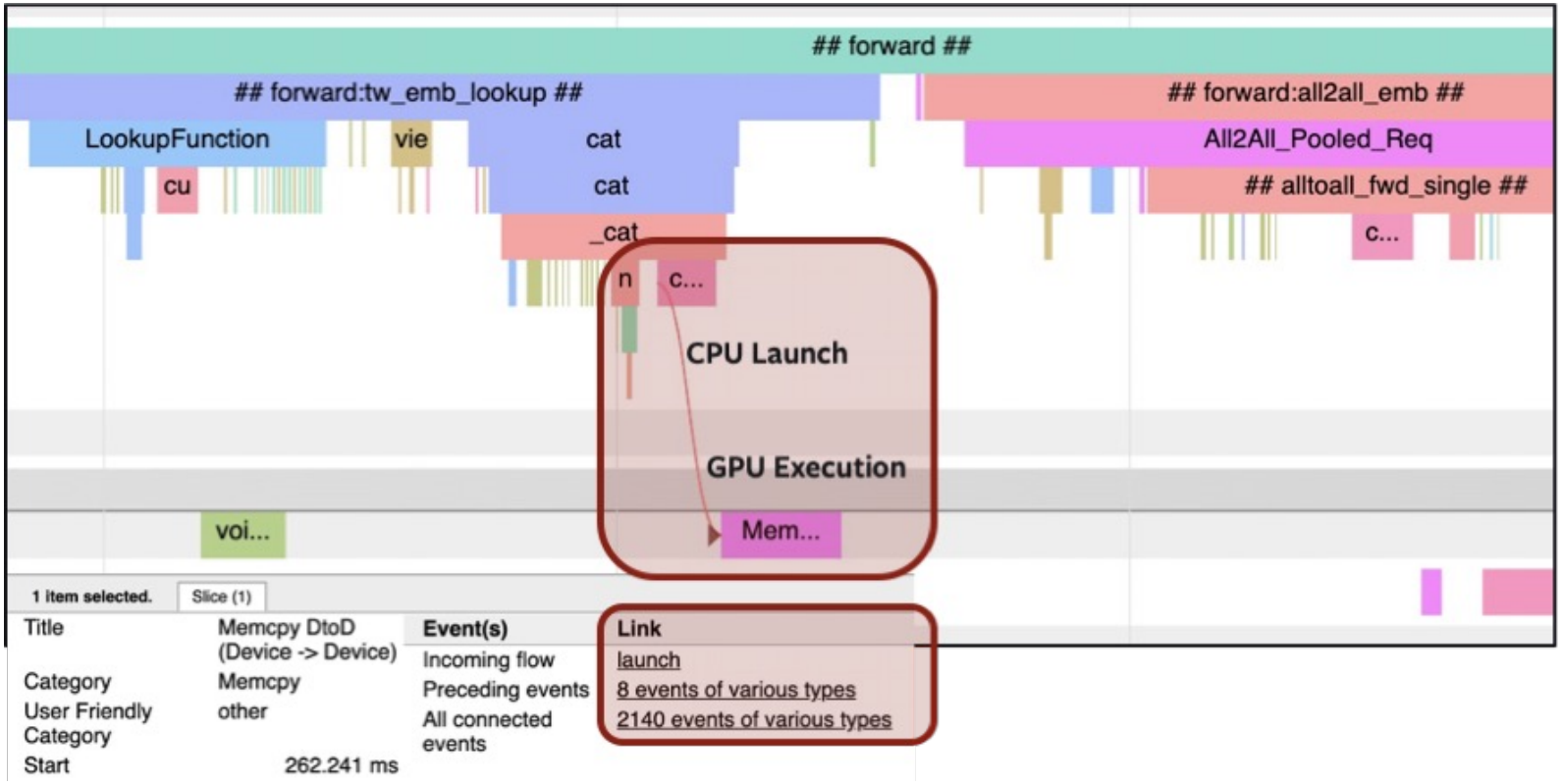
Timeline Tracing:
Chrome Trace Viewer- CPU and GPU timelines



Timeline Tracing:



You can see how CPU and GPU ops are connected

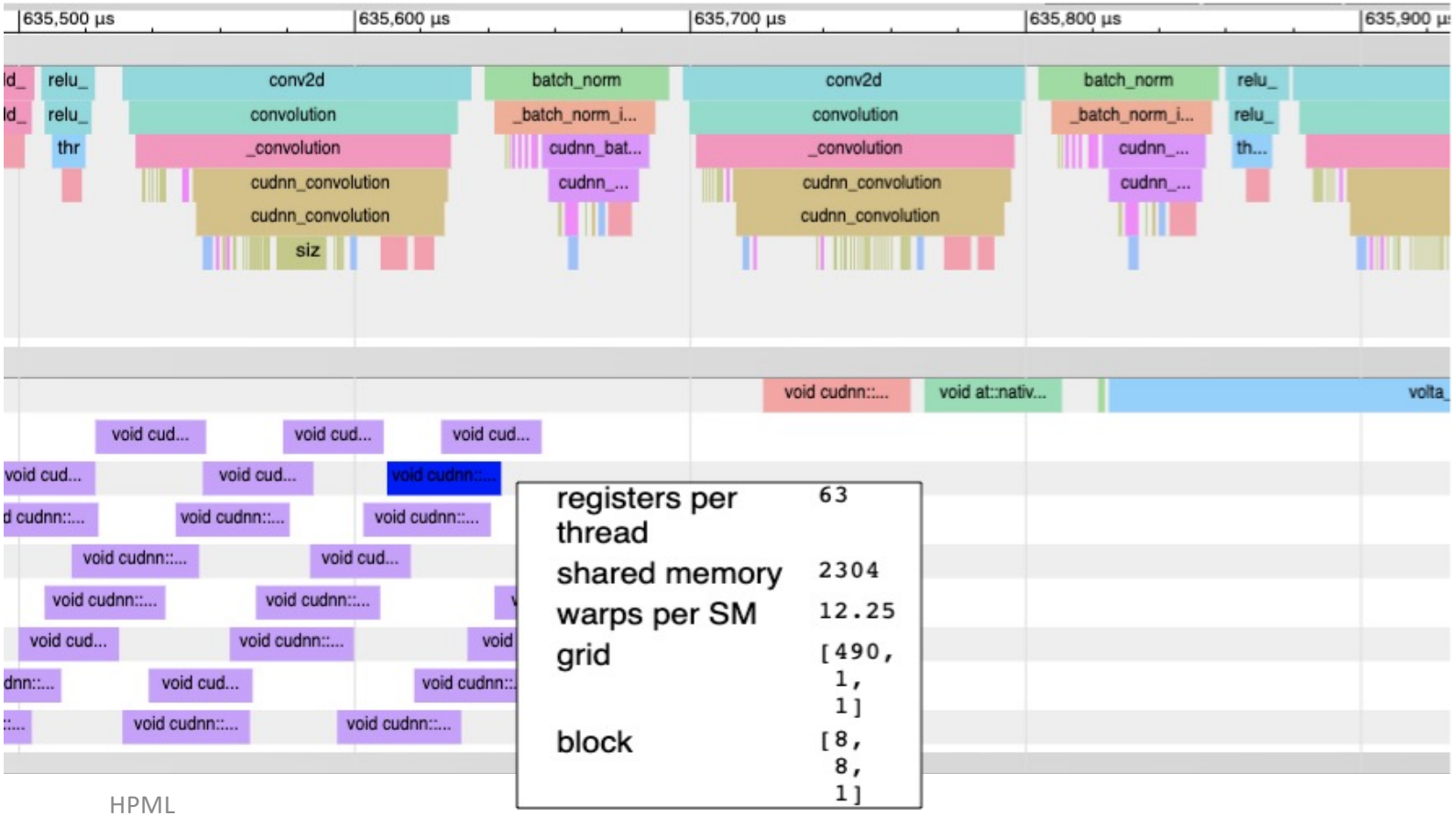


Timeline Tracing:
Inspect stats to individual activities

Nvidia-smi shows 86% utilization

But.. only a fraction of Streaming Multiprocessor are actually used by these kernels!

Timeline Tracing:
Inspect stats to individual activities



Much better utilization after increasing input sizes

Trace Analysis Examples with PyTorch Profiler

Thanks to Facebook Engineers for examples:
Lei Tian, Natalia Gimelshein, Lingyi Liu, Feng Shi & Zhicheng Yan

Anti-pattern: Long GPU idle time

Issue:

1. Large periods of GPU inactivity
2. Trace does not show why

Solution:

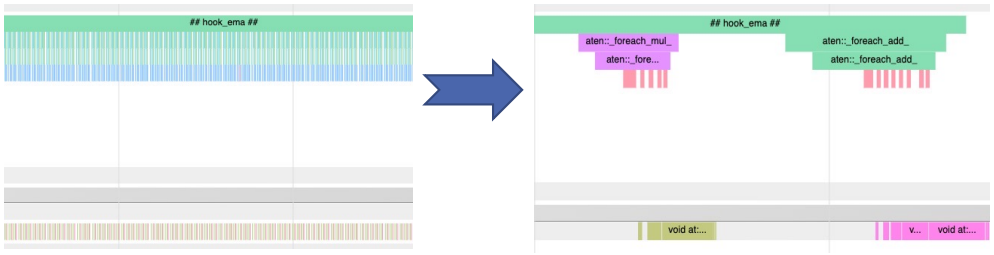
1. Use record_function to reveal bottlenecks on CPU
2. Parallelize CPU operations
3. Overlap CPU and GPU operations

```
temp = ""
num_substr = len(emb[k])

with record_function("## join_string {} ##".format(num_substr)):
    temp = ",".join(str(x) for x in emb[k]) # string concatenation

with record_function("## append_record_in_else ##"):
    records.append(f"{input_df.id[i + k]}\t{temp}\n") # list append
```

Anti-pattern: Excessive CPU/GPU Interactions

First issue:

- Exponential moving avg hook function has a loop – CPU bottleneck
- Can rewrite using torch._foreach ops – loop now on GPU




```
BEFORE
def on_step(self, task) -> None:
   ...
   with torch.no_grad():
      it = model_state_iterator(task.base_model)
      # iterate on every name & param
      for name, param in it:
         s = self.state.ema_model_state
         s[name] = self.decay * s[name] +
            (1 - self.decay) *
            param.to(device= self.device)


AFTER
def on_step(self, task) -> None:
   ...
   with torch.no_grad():
      torch._foreach_mul_(
         self.ema_model_state_list, self.decay)
      torch._foreach_add_(
         self.ema_model_state_list,
         self.param_list,
         alpha=(1 - self.decay))
```

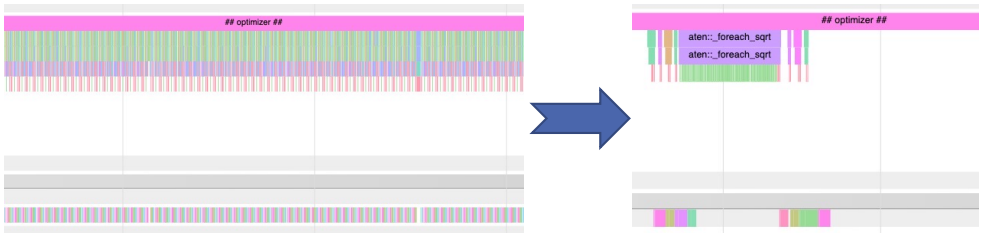
EMA HOOK 100X FASTER

ITERATION TIME: 860MS -> 770MS

Anti-pattern: Long GPU idle time

Second issue:

- Optimizer step uses a naïve implementation of RMSProp
- PyTorch provides an optimized multi-tensor version – using torch._foreach
- Switch to optimized version!




```
BEFORE
def prepare(self, param_groups):
   self.optimizer = RMSpropTFV2Optimizer(
      param_groups,
      …


AFTER
import torch.optim._multi_tensor as optim_mt
def prepare(self, param_groups):
   self.optimizer = optim_mt.RMSprop(
      param_groups,
      …
```

OPTIMIZER 12X FASTER

ITERATION TIME: 770MS -> 600MS

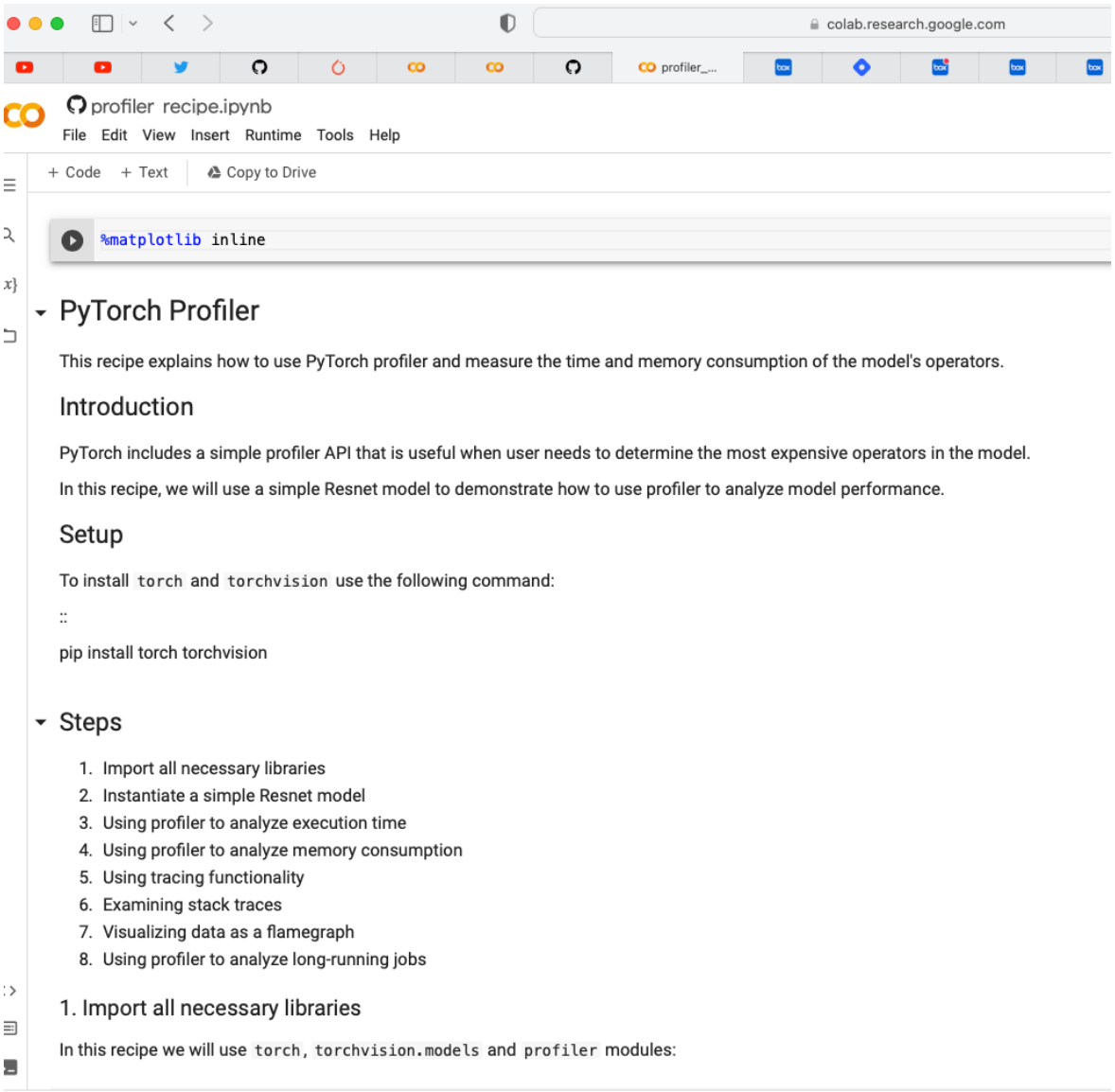
BERT Performance Optimization Case Study

- From 2.4 req/s to 1,400+ req/s
- CPU Inference
  - torch.set_num_threads(1)
  - Inel IPEX
  - Quantization
- GPU Inference on 1 T4 GPU
  - model.half()
  - DistiBERT
  - Increase batch size
  - Do not overpad
  - Faster Transformer

	Throughput	P99
BERT unoptimized bs=1	70.67 seq/s	20.44ms
BERT model.half() bs=8	359 seq/s	23.58ms
DistilBERT model.half() bs=16	689 seq/s	22.8ms
BERT Faster Transformer	885 seq/s	19.83ms
DistilBERT no padding model.half() bs=32	1423 seq/s	19.7ms

Example from PyTorch Documentation

https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html



Another Example from PyTorch Documentation with Tensorboard integration

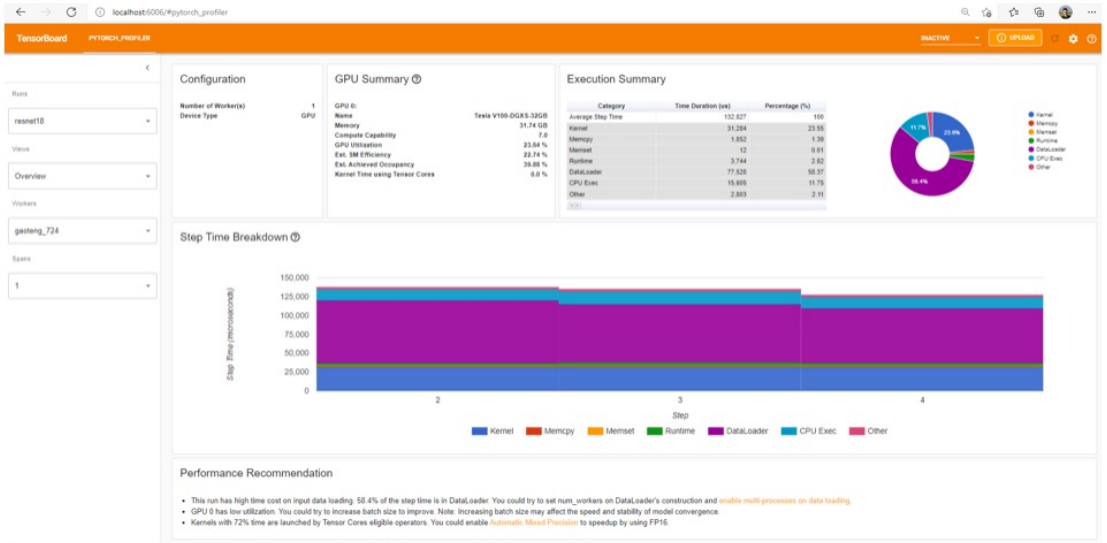
https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html

Tutorials > PyTorch Profiler With TensorBoard

```
http://localhost:6006/#pytorch_profiler
```

You could see Profiler plugin page as shown below.

- Overview



The overview shows a high-level summary of model performance.

The “GPU Summary” panel shows the GPU configuration, GPU usage and Tensor Cores usage. In this example, the GPU Utilization is low. The details of these metrics are here.

The “Step Time Breakdown” shows distribution of time spent in each step over different categories of execution. In this example, you can see the `DataLoader` overhead is significant.

The bottom “Performance Recommendation” uses the profiling data to automatically highlight likely bottlenecks, and gives you

Other Profiling Tools

Using *profile/cProfile*

- From your program:
  - Example profiling a regular expression

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

- To profile a script:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

- By default a summary is provided, using *pstats*
- By specifying an *output_file* the profile can be processed afterwards
- https://docs.python.org/3/library/profile.html

Profiling a PyTorch neural network

- Consider this NN example: a two-layers network with ReLU activation

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

loss_fn = torch.nn.MSELoss(size_average=False)
learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)

    loss = loss_fn(y_pred, y)
    print(t, loss.data[0])

    model.zero_grad()

    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
def main():
    x, y, model, loss_fn = setup(N, D_in, H, D_out)
    learn(x, y, model, loss_fn)

if __name__ == "__main__":
    main()
```

Profiling a PyTorch neural network

- Profile (partial) collected with:

```
python -m cProfile -s time nn.py
```

- Output:

```
326044 function calls (313968 primitive calls) in 1.073 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      500    0.249    0.000    0.249    0.000 {method 'run_backward' of 'torch._C._EngineBase' objects}
     1000    0.189    0.000    0.189    0.000 {built-in method torch._C.addmm}
    27/26    0.081    0.003    0.084    0.003 {built-in method _imp.create_dynamic}
      261    0.054    0.000    0.057    0.000 <frozen importlib._bootstrap_external>:830(get_data)
     1386    0.037    0.000    0.037    0.000 {built-in method posix.stat}
        3    0.026    0.009    0.063    0.021 utils.py:61(parse_header)
      261    0.025    0.000    0.025    0.000 {built-in method marshal.loads}
12000/5000   0.024    0.000    0.039    0.000 module.py:513(named_parameters)
     2000    0.023    0.000    0.023    0.000 {method 'mul' of 'torch._C.FloatTensorBase' objects}
  801/798    0.021    0.000    0.033    0.000 {built-in method builtins.__build_class__}
        1    0.021    0.021    1.073    1.073 nn.py:1(<module>)
```

Snakeviz

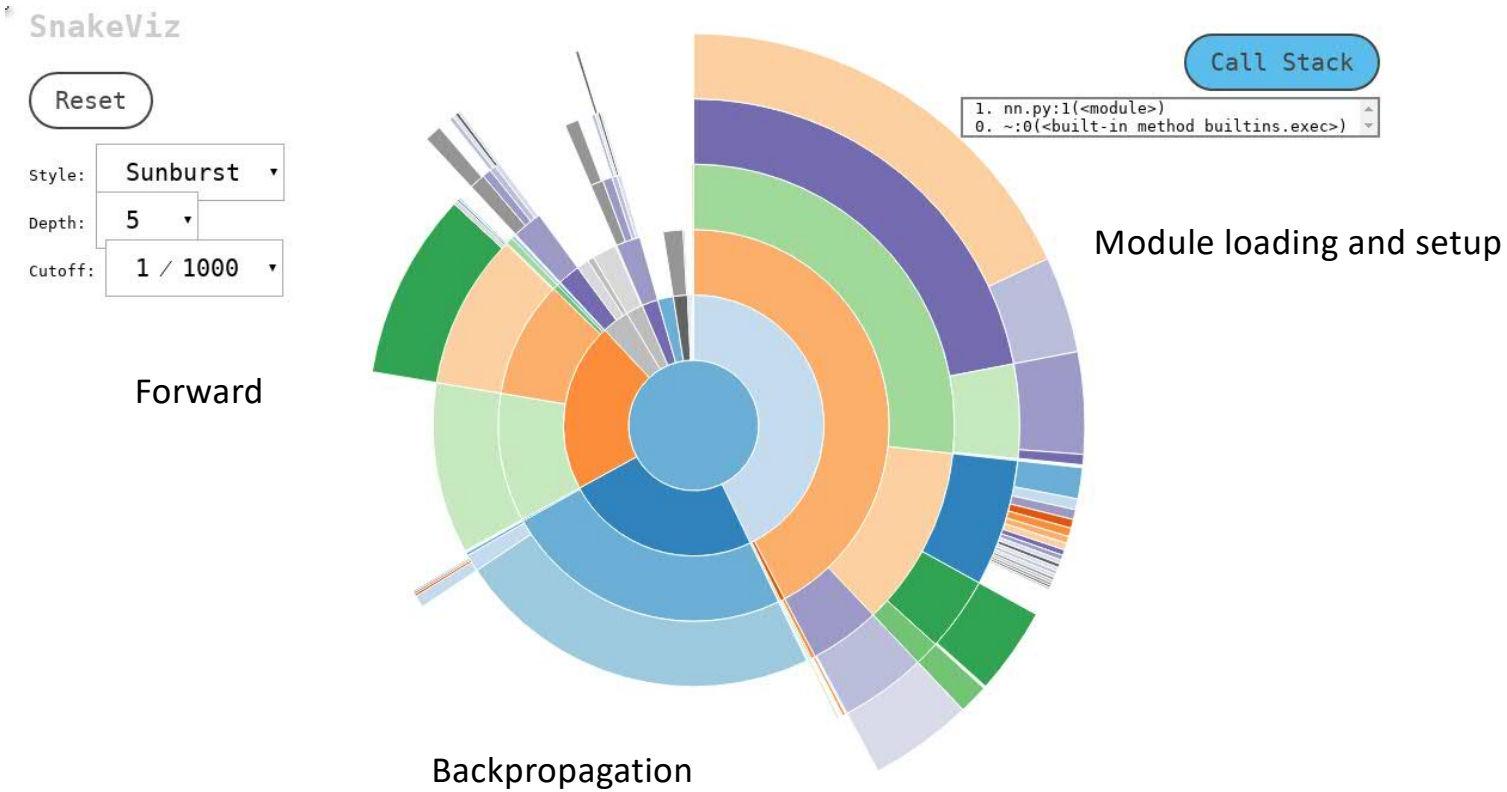
- Graphical tool to visualize content of profile created with cProfile
- Need to use the profile output file on your laptop
- Usage:

```
$ snakeviz nn.profile --server
snakeviz web server started on 127.0.0.1:8080; enter Ctrl-C to exit
http://127.0.0.1:8080/snakeviz/%2Fcygdrive%2Fc%2FUsers%2FAlessandroMORARI%2FBox+Sync%2Fwork%2Fcygwin%2FAlessandroMORARI%2Fnn.profile
```

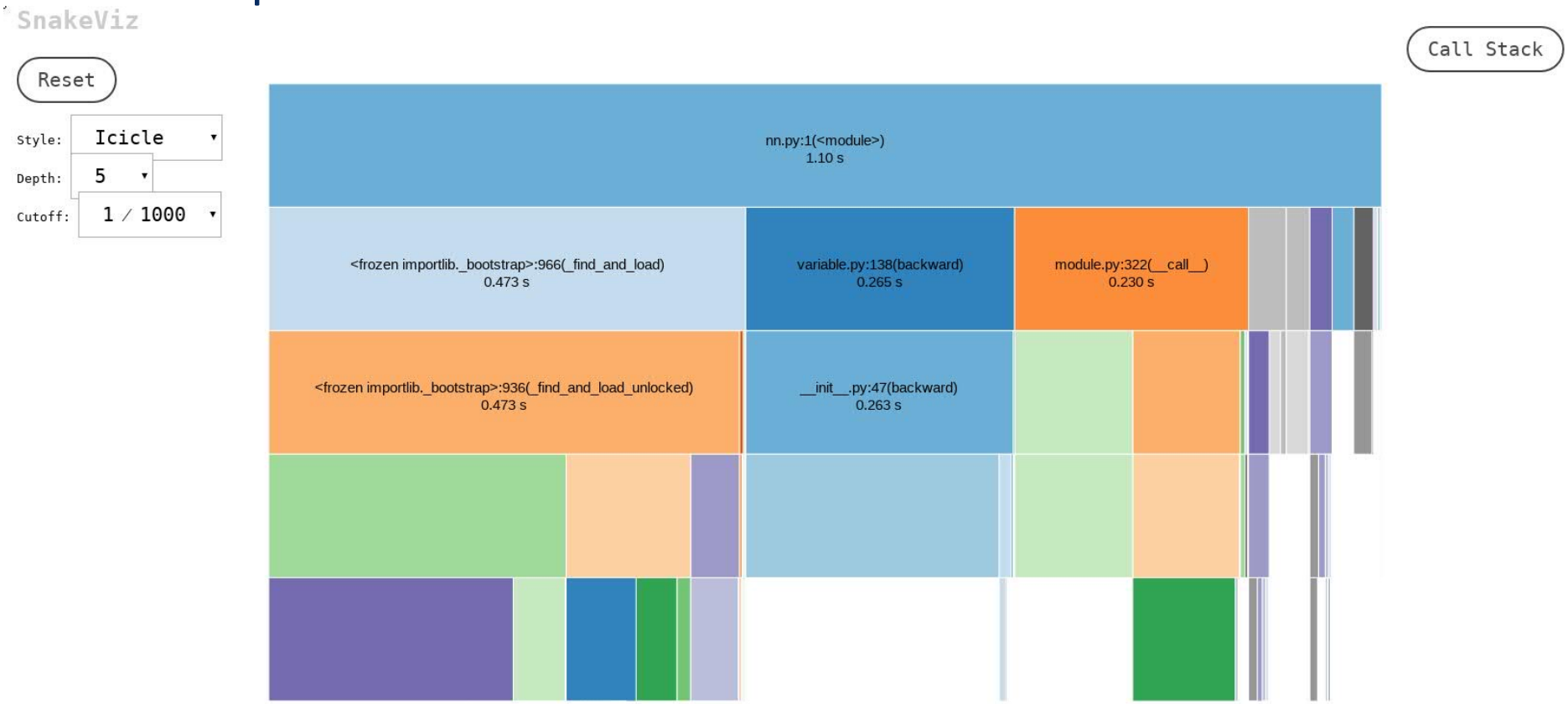
- Then open the browser and connect to URL provided….

- https://jiffyclub.github.io/snakeviz/

SnakeViz and sunburst plots



Icicle plots



Profiling memory usage

- Important to determine whether:
  - Allocations can be avoided in the critical paths (e.g. by reusing allocated memory)
  - The overall memory usage is too high and limits the problem size that can be solved
- https://pypi.python.org/pypi/memory_profiler
- Usage:
  - `python –m memory_profiler nn.py`

Example

- In the following example, we create a simple function `my_func` that allocates lists `a`, `b` and then deletes `b`:

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

```
$ python -m memory_profiler example.py
```

Example

- Execute the code passing the option `-m memory_profiler` to the python interpreter to load the memory_profiler module and print to stdout the line-by-line analysis. If the file name was example.py

```
$ python -m memory_profiler example.py
```

- this would result in:

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     3   38.816 MiB   38.816 MiB           1   @profile
     4                                         def my_func():
     5   46.492 MiB    7.676 MiB           1       a = [1] * (10 ** 6)
     6  199.117 MiB  152.625 MiB           1       b = [2] * (2 * 10 ** 7)
     7   46.629 MiB -152.488 MiB           1       del b
     8   46.629 MiB    0.000 MiB           1       return a
```

1st column: line number of the code that has been profiled

2nd column: memory usage of the Python interpreter after that line has been executed

3rd column: difference in memory of the current line with respect to the last one

Another Example: Output of *memory_profiler*

```
Filename: nn.py

Line #    Mem usage    Increment   Line Contents
================================================
    20   85.031 MiB   85.031 MiB   @profile
    21                             def learn(x, y, model, loss_fn):
    22   85.031 MiB    0.000 MiB       learning_rate = 1e-4
    23   91.242 MiB    0.000 MiB       for t in range(500):
    24   91.242 MiB    2.184 MiB           y_pred = model(x)
    25
    26   91.242 MiB    0.047 MiB           loss = loss_fn(y_pred, y)
    27   91.242 MiB    0.164 MiB           print(t, loss.data[0])
    28
    29   91.242 MiB    0.000 MiB           model.zero_grad()
    30
    31   91.242 MiB    3.242 MiB           loss.backward()
    32
    33   91.242 MiB    0.000 MiB           for param in model.parameters():
    34   91.242 MiB    0.574 MiB               param.data -= learning_rate * param.grad.data
```

PyTorch Performance - Benchmarking

Profiling vs. benchmarking

- In benchmarking we're interested in assessing the **absolute speed** of a piece of code
- Profiling results are not reliable for absolute values
  - Profiling introduces **overhead**
  - C++ and Python sections of the application are affected by profiling in a different way, depending on the profiling tools being used
- When benchmarking, variability has to be taken into account:
  - Dependencies on the input
  - Dependencies on temporary conditions
    - Always collect stats on multiple executions

Benchmarking: the *timeit* module

- The *timeit* module deals with many of the requirements of benchmarking
- Execute the code in a loop, and take the best of multiple runs
- Using from the command line
  - example (timing a matrix multiply in numpy, 5 runs of 20 iterations each):

```
% python3 -m timeit -v -n 20 -r 5 -s "import numpy; x=numpy.random.rand(1000, 1000)" "x=x.dot(x)"

raw times: 210 msec, 217 msec, 184 msec, 199 msec, 211 msec

20 loops, best of 5: 9.19 msec per loop
```

The *timeit* module

- From Python code:

```
import timeit

t = timeit.Timer("x = x.dot(x)",
                 setup = "import numpy; x = numpy.random.rand(1000, 1000)").repeat(number=20, repeat=10)

print("raw times: {0}\nbest of 5: {1:.0f} ms".format(" ".join([ "%.3f" % x for x in t ]), min(t)/20*1000))
```

  - Output:

```
% python3 test_timit.py
  raw times: 0.209 0.228 0.183 0.170 0.179 0.174 0.211 0.191 0.170 0.169
  best of 10: 8 ms
```

Lesson Key Points

- Python performance:
  - Interpreter inner workings
  - Memory Management
  - Typing
- PyTorch performance
  - Computation Graph Approach
  - Just In Time Compilation
  - Profiling
  - Benchmarking

PyTorch Examples

Autograd Example (1)

```
Import torch

dtype = torch.float
device = torch.device("cpu") # Use the CPU as device
# Alternatively, use device = torch.device("cuda:0") to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs.
# Setting requires_grad=False indicates that we do not need to compute gradients
# with respect to these Tensors during the backward pass.
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Create random Tensors for weights.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)
```

From:
http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

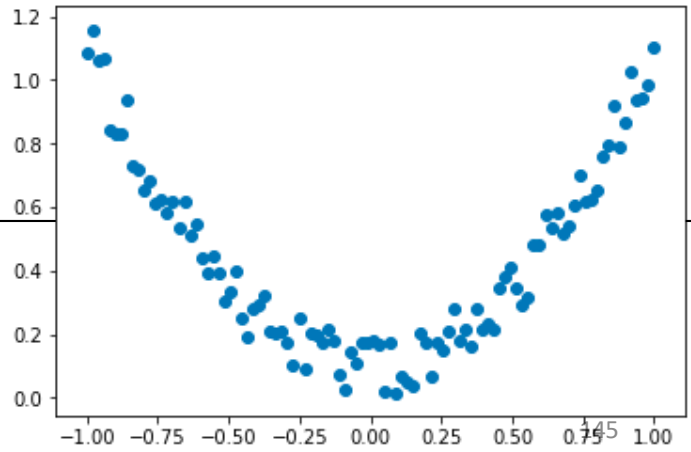
Autograd Example (2)

```
learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y using operations on Variables;
    # mm: matrix multiply; clamp: clamp in [min;max]
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    # Compute and print loss. Now loss is a Tensor of shape (1,)
    # loss.item() is a scalar value holding the loss.
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())
    # Use autograd to compute the backward pass.
    loss.backward()
    # Update weights using gradient descent; w1.data and w2.data are Tensors,
    # w1.grad and w2.grad are Variables and w1.grad.data and w2.grad.data are Tensors.
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        # Manually zero the gradients after updating weights
        w1.grad.zero_()
        w2.grad.zero_()
```

From: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Linear Regression Example (1)

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)  # x data (tensor), shape=(100, 1)
#unsqueeze: reshape tensor
#linspace: return a one-dimensional vector of 100 points between -1 and 1
y = x.pow(2) + 0.2*torch.rand(x.size())                 # noisy y data (tensor), shape=(100, 1)

plt.scatter(x.numpy(), y.numpy())
plt.show()
```



https://github.com/MorvanZhou/PyTorch-Tutorial

Linear Regression Example (2)

```
class Net(torch.nn.Module):
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden)   # hidden layer
        self.predict = torch.nn.Linear(n_hidden, n_output)   # output layer

    # For the forward() method, we supply the input data x as the primary argument.
    # Then we pass through the two layers of our simple network
    def forward(self, x):
        x = F.relu(self.hidden(x))      # activation function for hidden layer
        x = self.predict(x)             # linear output
        return x
```

https://github.com/MorvanZhou/PyTorch-Tutorial

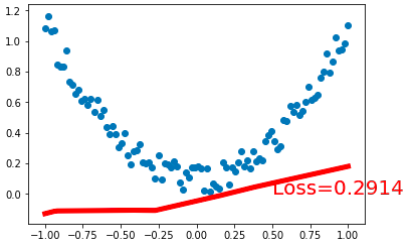
Linear Regression Example (3)

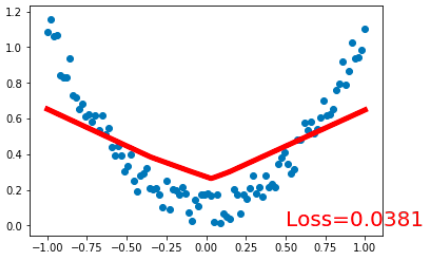
```
# create the network
net = Net(n_feature=1, n_hidden=10, n_output=1)     # define the network
print(net)  # net architecture

# create a stochastic gradient descent optimizer
# the method net.parameters() (from the base nn.Module class that we inherit in the Net class), contains all the parameters of our network
optimizer = torch.optim.SGD(net.parameters(), lr=0.2)
# define the loss, we use the regression mean squared loss
loss_func = torch.nn.MSELoss()

plt.ion()   # turn on interactive mode
for t in range(200):
    prediction = net(x)     # input x and predict based on x
    loss = loss_func(prediction, y)     # must be (1. nn output, 2. target)
    # With optimizer.zero_grad() we resets all the gradients in the model
    # so that it is ready to go for the next back propagation pass.
    optimizer.zero_grad()   # clear gradients for next train
    loss.backward()         # backpropagation, compute gradients
    # runs a back-propagation operation from the loss Tensor backwards
    # through the network and execute a gradient descent step based on
    # the gradients calculated during the .backward() operation.
    optimizer.step()        # apply gradients
```

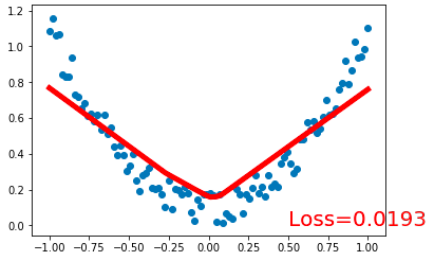
https://github.com/MorvanZhou/PyTorch-Tutorial

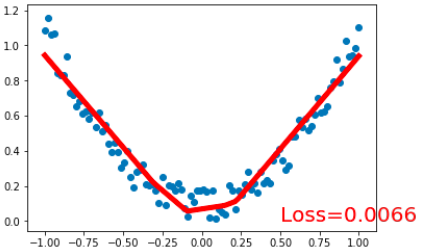
Linear Regression Example (4)

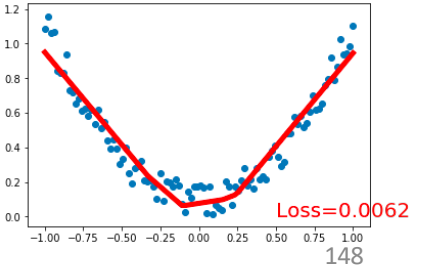
```
        # plot every 5 epochs the learning process
        if t % 5 == 0:
            # plot and show learning process
            plt.cla()
            plt.scatter(x.data.numpy(), y.data.numpy())
            plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=5)
            plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size': 20, 'color':  'red'})
            plt.pause(0.1)
plt.ioff()
plt.show()
```











https://github.com/MorvanZhou/PyTorch-Tutorial

…….

NN Example (1)

```
import torch

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(reduction='sum')
```

From: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

NN Example (2)

From: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

```
learning_rate = 1e-4
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(x)
    # Compute and print loss. We pass Tensors containing the predicted and true
    # values of y, and the loss function returns a Tensor containing the loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.item())
    # Zero the gradients before running the backward pass.
    model.zero_grad()
    # Backward pass: compute gradient of the loss with respect to all the learnable
    # parameters of the model. Internally, the parameters of each Module are stored
    # in Tensors with requires_grad=True, so this call will compute gradients for
    # all learnable parameters in the model.
    loss.backward()
    # Update the weights using gradient descent. Each parameter is a Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```