ECE-GY 9143
Introduction to High Performance Machine Learning

Lecture 7

Parijat Dube

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

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
  - More white space
- Performance relevant features:
  - **Dynamic typing**
  - **Memory management**

Static Typing - C/C++

- Programmer has to specify types of each variable
- Compiler checks types at **compile time**
- Implications:
  - All types are known before execution
  - Variables in memory do not need to contain types, only values

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

Write code
- Specify types

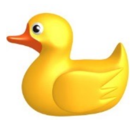
Compile
- Check types

Run
- No types

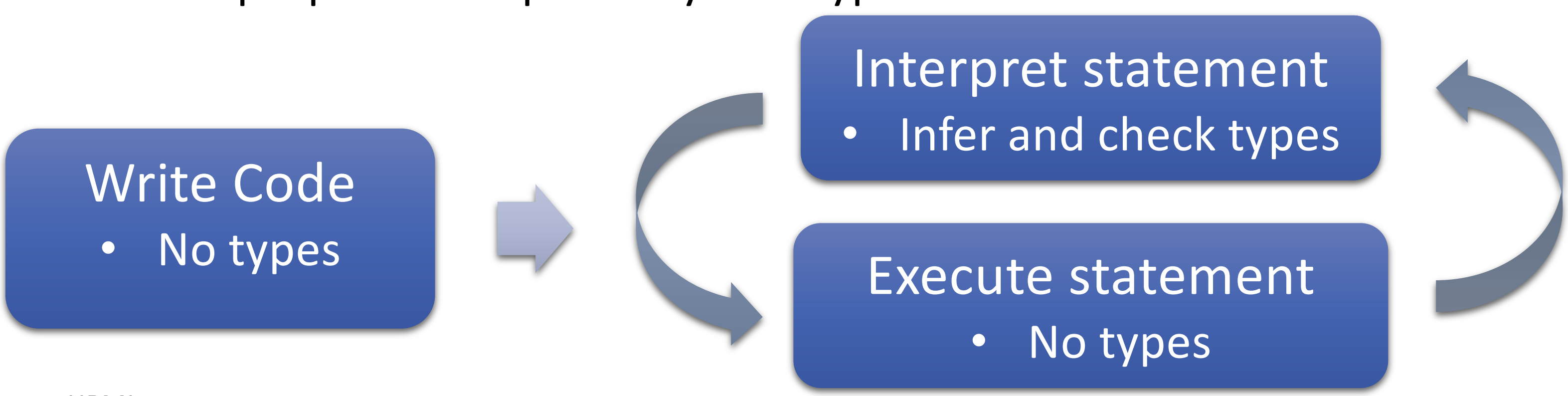
Dynamic Typing – Python

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

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



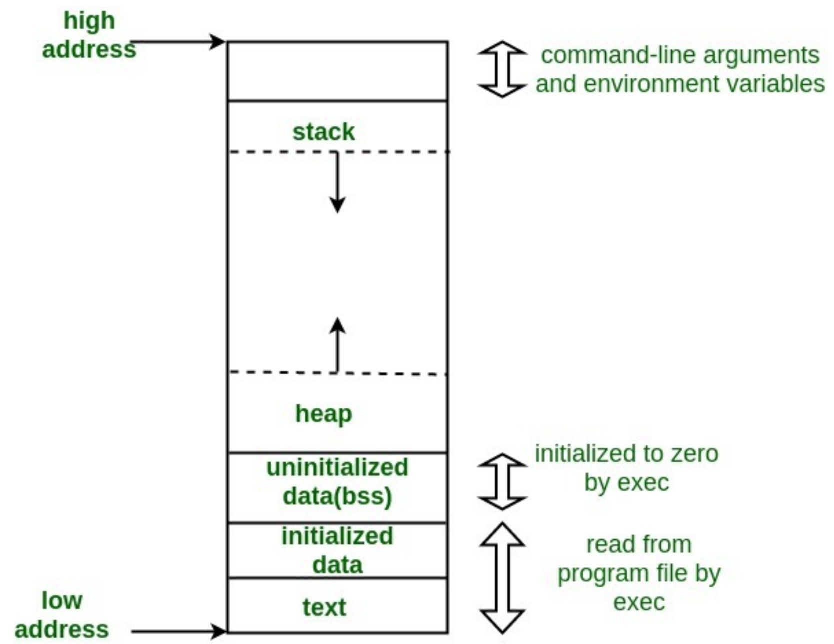
- Duck typing: **“If it walks like a duck and it quacks like a duck, then it must be a duck”**
- An object is of a given type if it has all methods and properties required by that type



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 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.

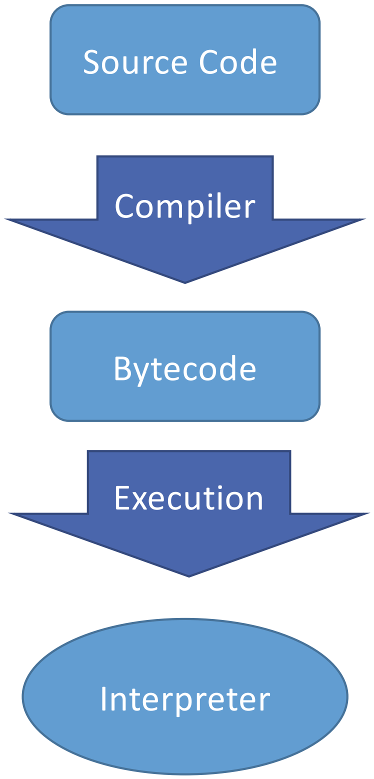
Python implementations

- 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, Nuitka, etc*.
- Numerical Accelerators/Frameworks: offer accelerated numerical libraries
  - **PyTorch,** Numpy, Numba, Copperhead,

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

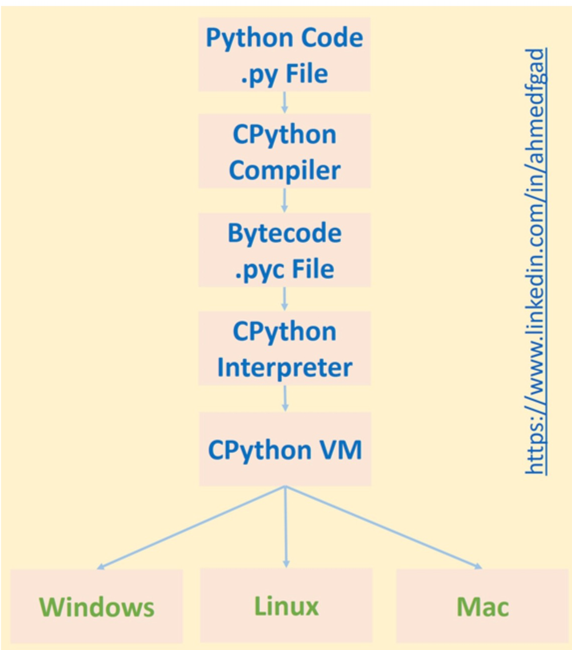
Python Execution Stages

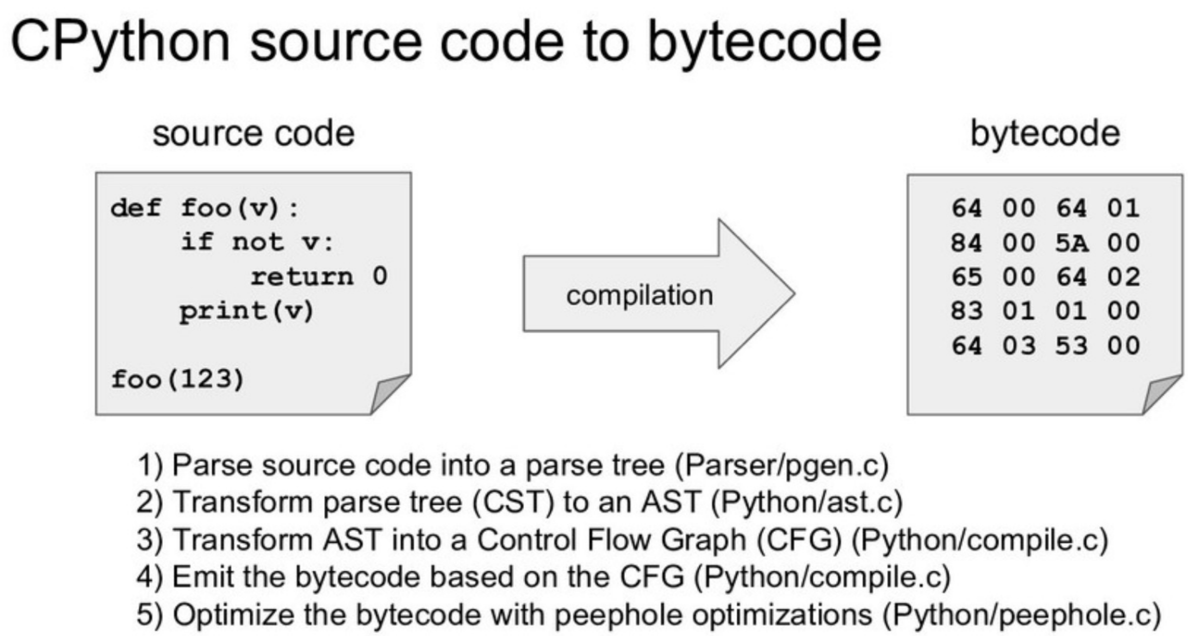
- Cpython is both an implementation of Python and a compiler
- CPython compiler:
  - Uses several stages to produce **bytecode**
  - Checks basic syntax and grammatical correctness
  - Bytecode can be saved in a .pyc file
  - Do not confuse with ***Cython***: superset of Python language to call C functions 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





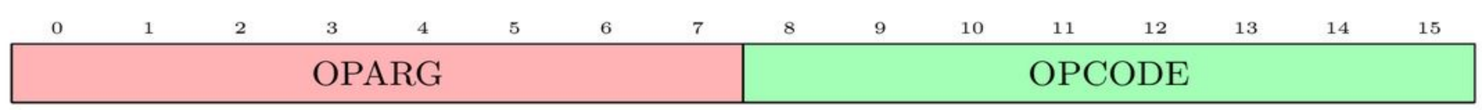
Python Bytecode

- 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
>>>
```

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



CPython interpreter - Stack Machine Example

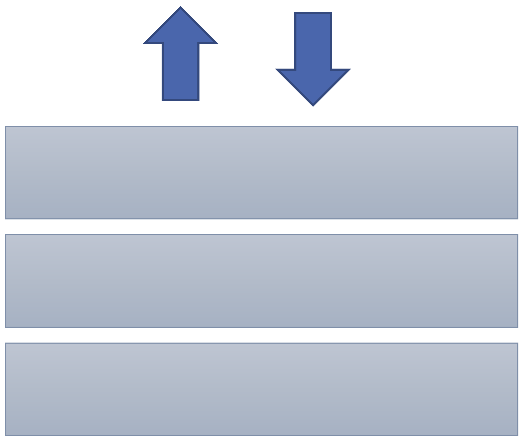
- Evaluate expression:

  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
```

- Stack state:



CPython interpreter - Stack Machine Example

- Evaluate expression:

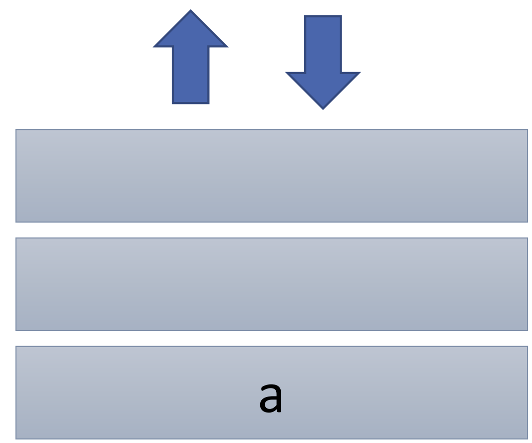
  $\boxed{d = a + b * c}$

- 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:

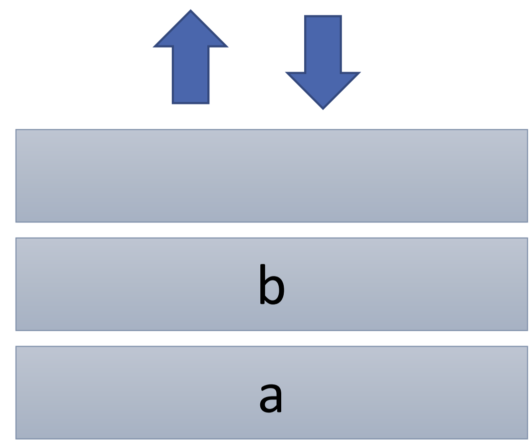
  d = a + b * c

- 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

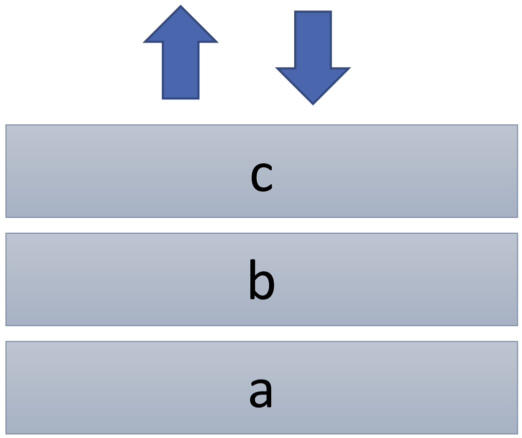
- Evaluate expression:

  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

- Stack state:



CPython interpreter - Stack Machine Example

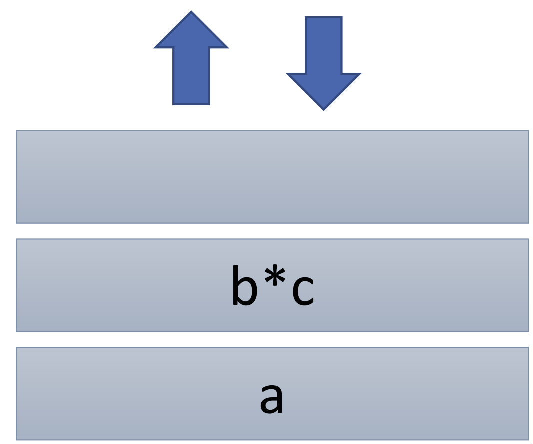
- Evaluate expression:

  d = a + b * c

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

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

- Stack state:



CPython interpreter - Stack Machine Example

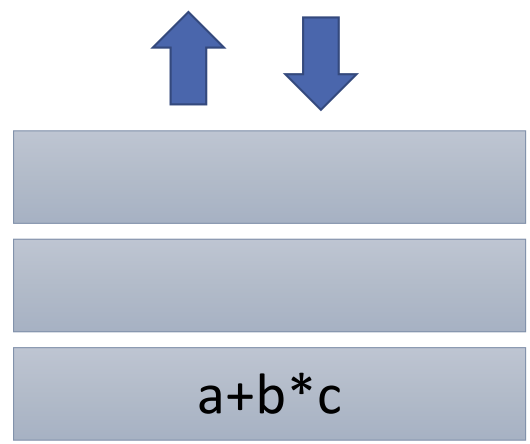
- Evaluate expression:

  d = a + b * c

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

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

- Stack state:



CPython interpreter - Stack Machine Example

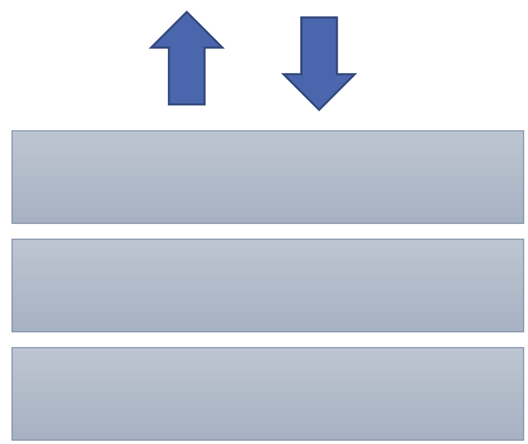
- Evaluate expression:

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
IP  STORE_FAST 3
```

- Stack state:



CPython instruction and value representation

- Instruction:

```
struct instr {
  unsigned i_jabs : 1;
  unsigned i_jrel : 1;
  unsigned char i_opcode;
  int i_oparg;
  struct basicblock_ *i_target;
  int i_lineno;
};
```

- *i_jabs, i_jrel* contain addresses for jumps
- *i_target* points to the basic block
- *i_lineno* contains the line number
- https://leanpub.com/insidethepythonvirtualmachine/read#leanpub-auto-the-interpreter-state

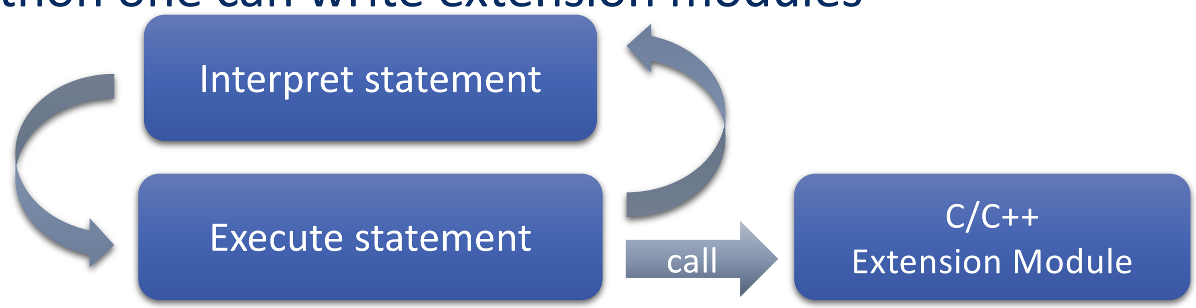
- Value object (base struct):

```
typedef struct _object {
      _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
  } PyObject;
```

- Every value is a **PyObject**
- *_PyObject_HEAD_EXTRA* linked list of objects
- *ob_refcnt* counts the references for recycling
- *ob_type* points to the object type

CPython Extension Modules

- 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

CPython Extension Module Example

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

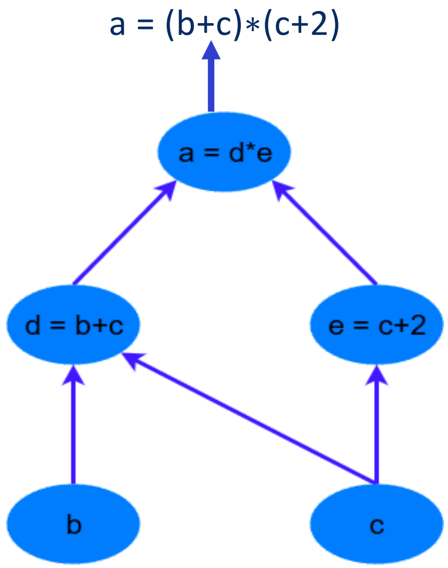
```
static PyObject* hello_module_print_hello_world(PyObject
*self, PyObject *args) {
   printf("Hello World\n");
   Py_RETURN_NONE;
}
static PyMethodDef hello_module_methods[] = {
   { "print_hello_world",
      hello_module_print_hello_world,
      METH_NOARGS,
      "Print 'hello world' from a method 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

Computation graph evaluation approaches

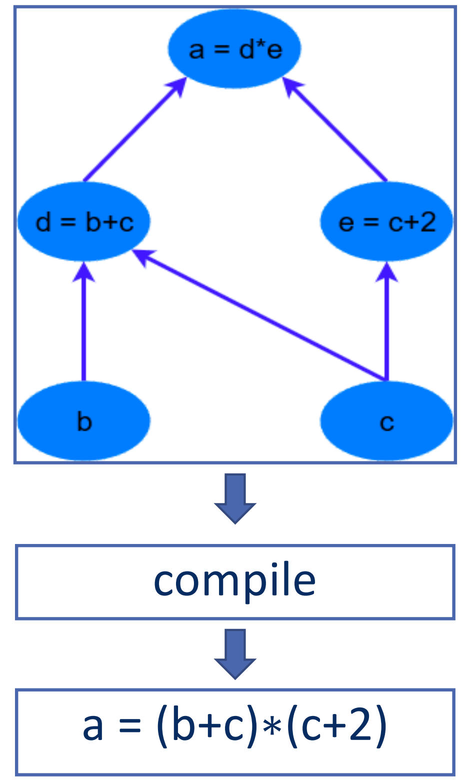
- Computation Graph
  - Forward propagation (use it as it is)
  - Backward propagation (compute gradients)
- Two evaluation approaches:
  - **Declarative**: declare all at once, compile, compute
    - TensorFlow, Caffe, Theano
  - **Imperative**: declare and compute each element at runtime
    - PyTorch, 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

(1)



(2)

(3)

Declarative framework example: TensorFlow

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

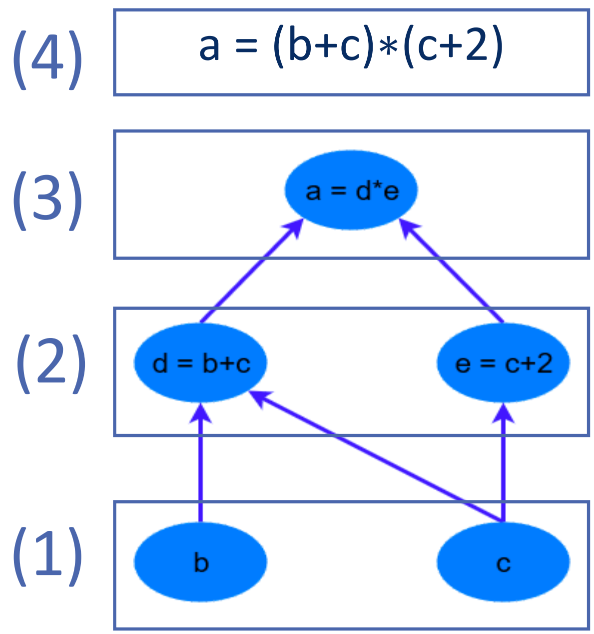
- Execute engine

```
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's actually 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 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

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

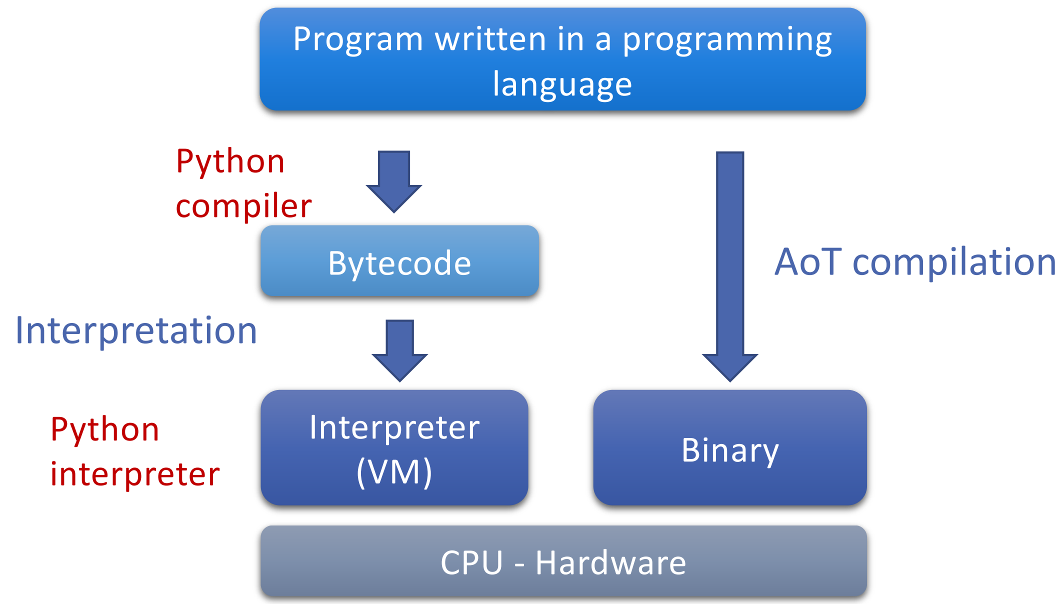
Co-evolution of execution modes in DL frameworks

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

PyTorch Performance – Just In Time Compilation

From language to binary execution

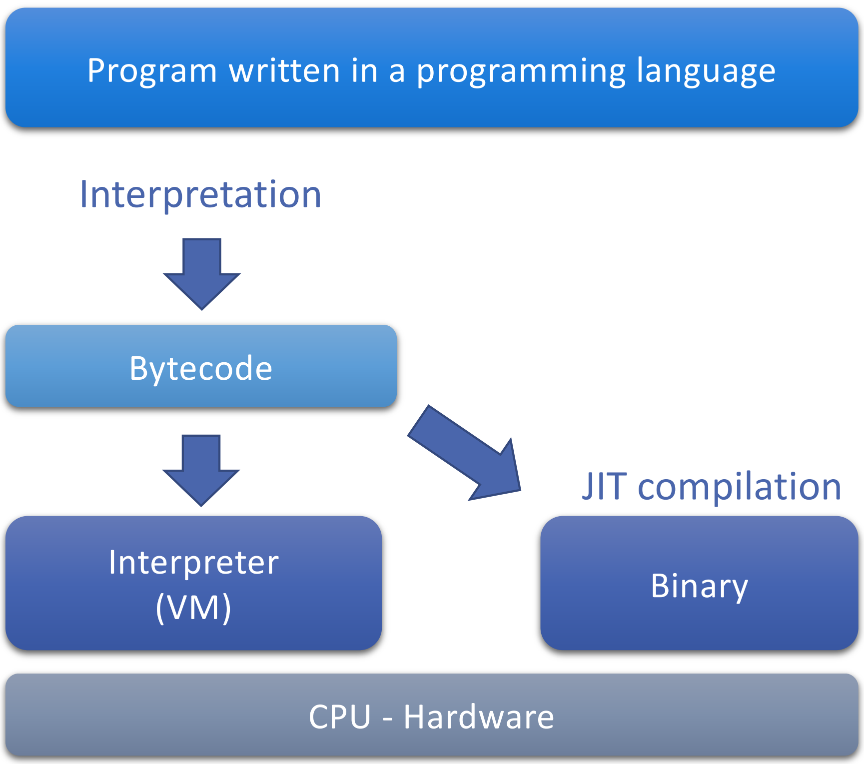
- **Interpretation:**
  1. Compile to bytecode
     - Bytecodes are platform-independent
     - Non-runnable code, needs a virtual machine to run
     - Python: collection of opcode and oparg
  2. Interpret in a virtual machine
     - Ex. Python, Java, Javascript
     - Virtual machine takes care of 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

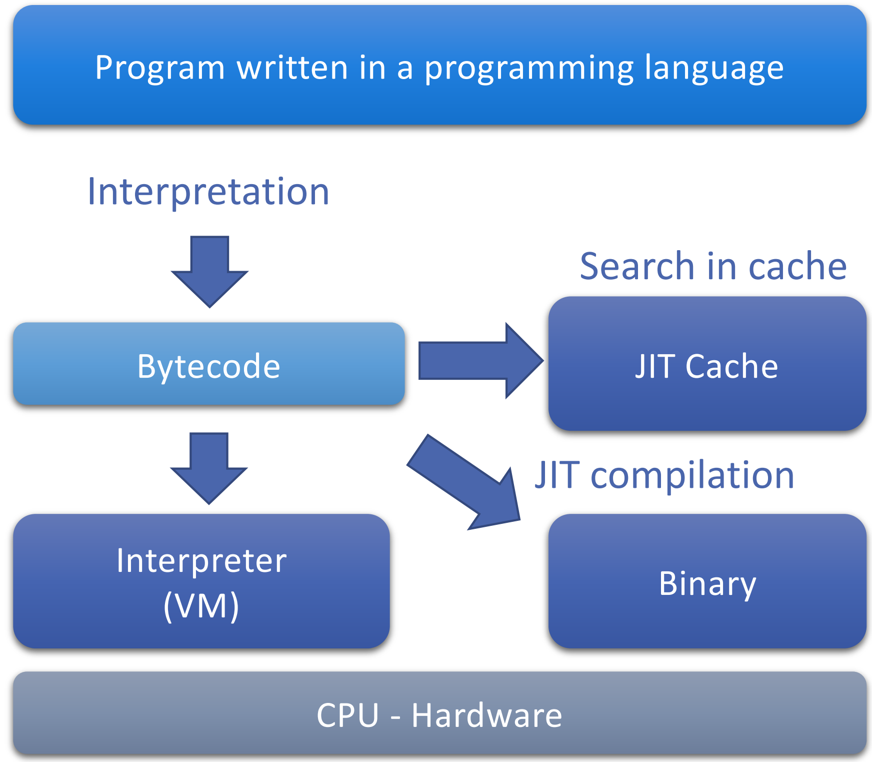
- When is it convenient to do JIT?
  - For functions that are going to be executed many times



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

- Numba is JIT compiler for Python
- https://numba.pydata.org
- Works best on code that uses NumPy arrays and functions, and loops
- **Numba:**
  - Generates optimized machine code at runtime for Python functions using the LLVM 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

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

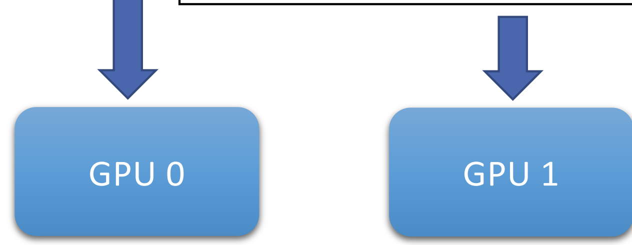
- **Out of order execution** and automatic **work scheduling**
- PyTorch code:

```python
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

```c
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;
  }
```

```c
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;
  }
```



PyTorch Performance – Profiling

Profiling objectives

- Identify critical section and resource bottlenecks
  - CPU, GPU, Memory, Network, I/O (disk)

- 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…)

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

Python profiling tools

- ***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
- ***profile:*** pure python module: higher overhead
- *pstats*: a module that provides analysis methods for the data collected by the profilers

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

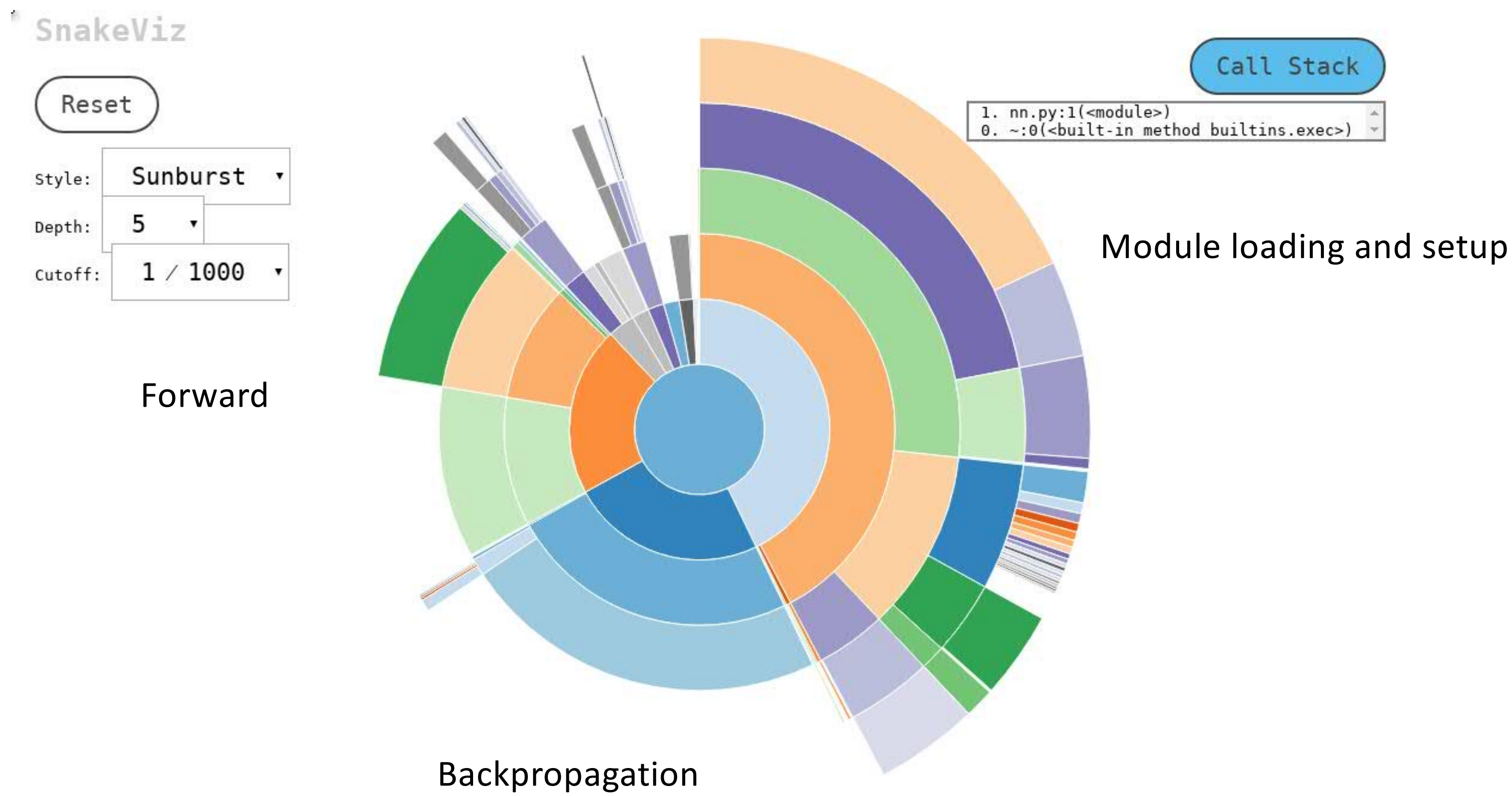
- Graphical tool to visualize content of profile created with cProfile
- Need to use the profile output file on your laptop
- Usage:

```
$ 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%2FAlessandroMOR
ARI%2FBox+Sync%2Fwork%2Fcygwin%2FAlessandroMORARI%2Fnn.profile
```

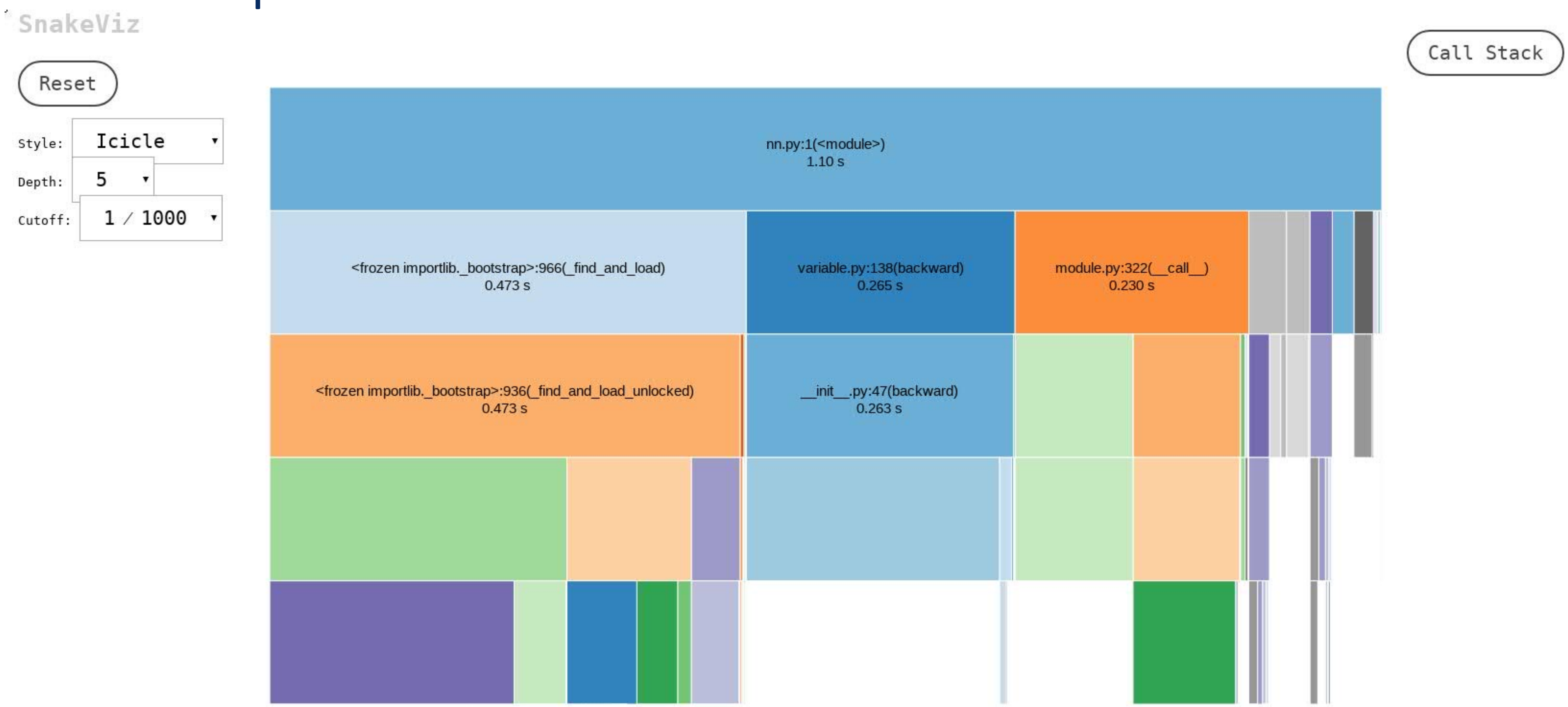
- Then open the browser and connect to URL provided….

- 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`**

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