Understanding Forward-Mode Auto Differentiation

- Mani Kishan Ghantasala
- Srimathi Mahalingam

Team: Mafia

Differentiation

Differentiation
- Manual
- Numeric
- Symbolic
- Automatic

Differentiation Types

```python
def f(x):
    return np.exp(2*x) - x**3

def f_prime(x):
    return 2*np.exp(2*x) - 3*x**2
```

Manual

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h}$$

Numerical

$$h(x) = f(x)g(x)$$

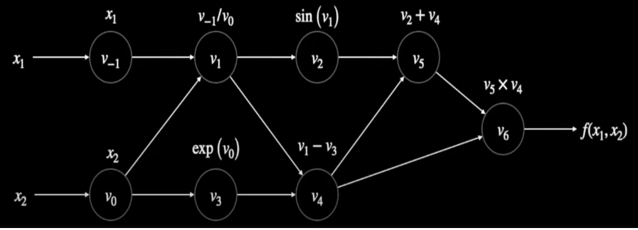
$$h'(x) = f'(x)g(x) + f(x)g'(x)$$

$$f(x) = u(x)v(x)$$

$$h'(x) = \big(u'(x)v(x) + u(x)v'(x)\big)g(x) + u(x)v(x)g'(x)$$

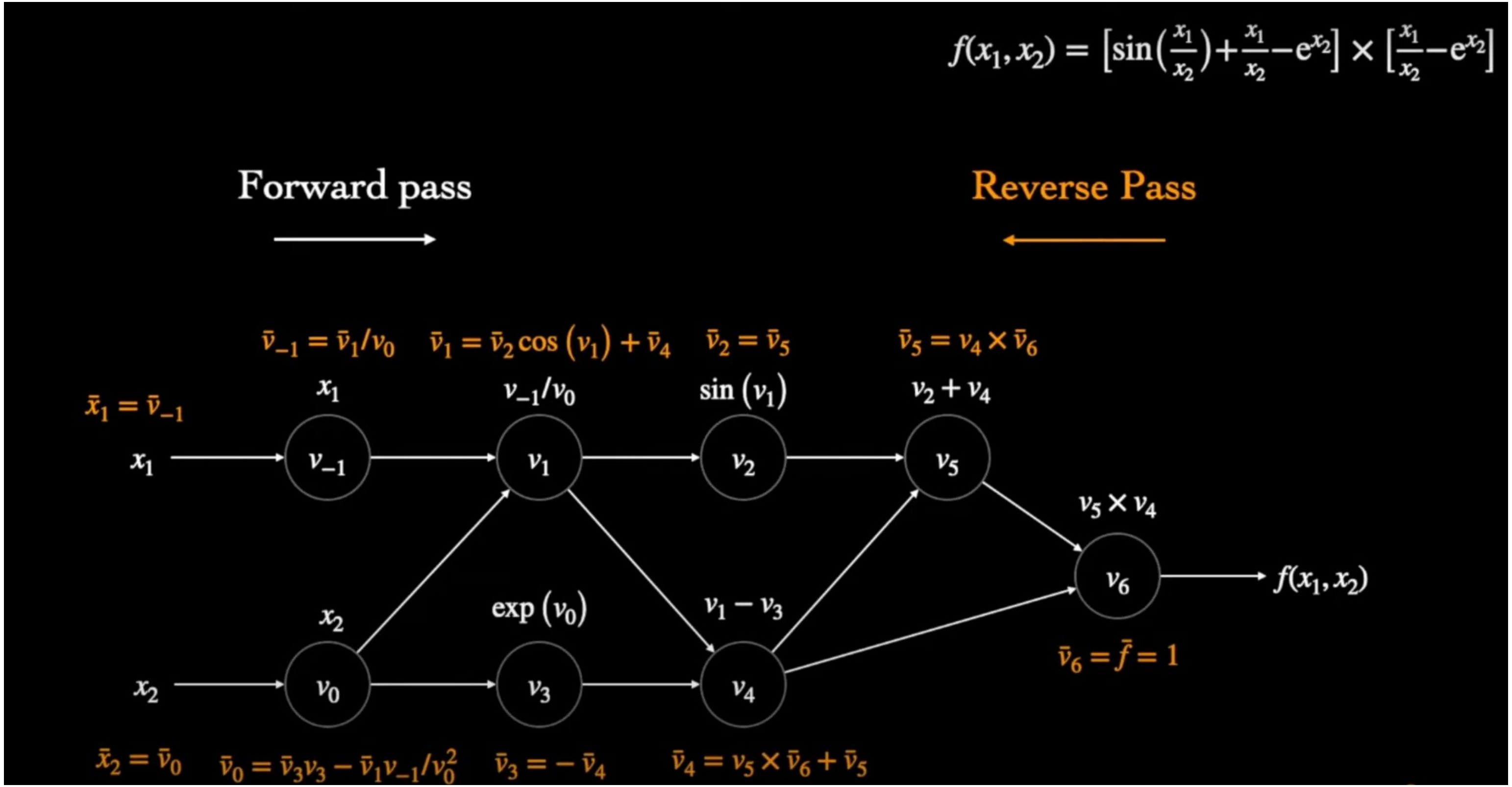
Symbolic

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$



Automatic

Forward Mode & Backward Mode AD



Jacobian Matrix & Jacobian Vector Product

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Expected Results

Task	Inference (and MeZO)	Backpropagation	Forward Auto-Differentiation
Excess Memory (MB)	327.50	24156.23	830.66

Current Results

```
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 20.00 MiB. GPU 0 has a total capacty
of 11.92 GiB of which 8.81 MiB is free. Including non-PyTorch memory, this process has 11.91 GiB memory
in use. Of the allocated memory 11.78 GiB is allocated by PyTorch, and 6.55 MiB is reserved by PyTorch b
ut unallocated. If reserved but unallocated memory is large try setting max_split_size_mb to avoid fragm
entation.  See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
Fatal Python error: none_dealloc: deallocating None: bug likely caused by a refcount error in a C extens
ion
Python runtime state: finalizing (tstate=0x00000000008a72b8)

Current thread 0x00007f6d375f9180 (most recent call first):
  Garbage-collecting
  <no Python frame>
Aborted
(/work/pi_huiguan_umass_edu/kunjal/latest_torch_venv) srimathimaha_umass_edu@gypsum-gpu087:/work/pi_huig
```

Using NVIDIA-smi -i

backprop.py (Backward prop)

```
===========|
|     0    N/A  N/A      49093      C    python
  2463MiB |
+------------------------------------------------------------
```

auto_diff.py (Forward AD)

```
===========|
|     0    N/A  N/A      48978      C    python
 12193MiB |
+------------------------------------------------------------
----------+
```

Goals

- Explore memory allocation challenges during JVP operator use in PyTorch profiler's forward auto mode differentiation.
- Determine if memory issues originate from JVP implementation or PyTorch profiler's memory allocation.
- Investigate memory profiling to identify the root cause of out-of-memory errors.

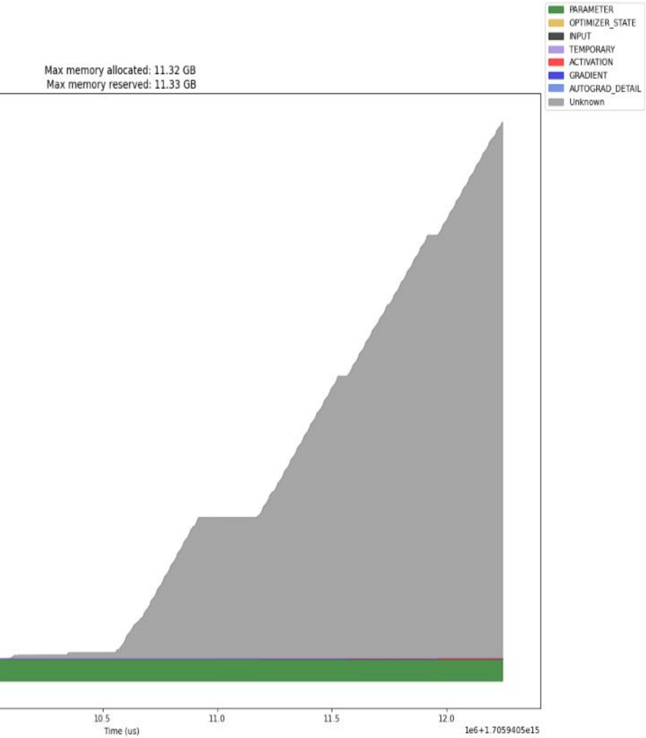
Environment Set Up

- CPU: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- GPU: NVIDIA GeForce GTX TITAN X - 16GB VRAM
- OS: Ubuntu 20.04.6 LTS
- RAM: 251GiB

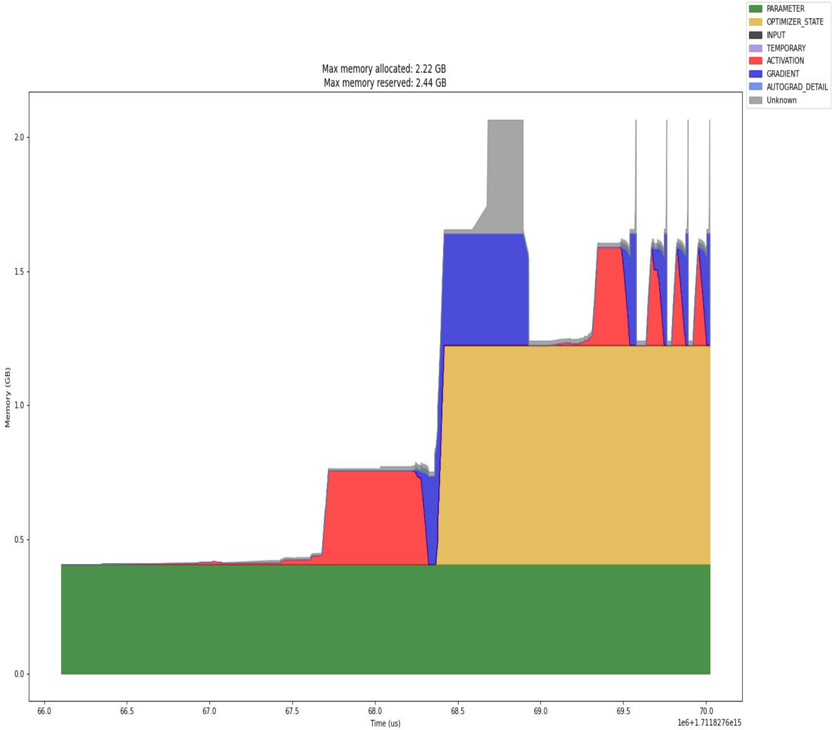
Approach 1: Try different Profiling methods

- Scalene
  - Challenge: Scalene profiler is incompatible with multithreaded code.
- Memory Timeline
- Eventlist Traces
- Memory Snapshot

Memory Timeline



Forward Mode (OOM Error)



Backward Mode

Memory usage

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

Forcing garbage collection didn't work

Name	Self CPU %	CUDA Mem	Self CUDA Mem	# of Calls
aten::empty_strided	0.76%	4.09 Gb	4.09 Gb	3189
aten::mul	3.21%	2.81 Gb	2.81 Gb	1710
aten::add	1.23%	2.69 Gb	2.19 Gb	1376
aten::mm	1.17%	1.25 Gb	1.25 Gb	564
aten::sub	1.38%	958.51 Mb	888.01 Mb	628
aten::addmm	8.97%	3.85 Gb	724.70 Mb	573
aten::empty	0.36%	715.92 Mb	715.92 Mb	1220
aten::bmm	0.80%	995.00 Mb	426.50 Mb	368
aten::gelu	0.34%	840.00 Mb	288.00 Mb	93
aten::gelu_backward	0.07%	276.00 Mb	276.00 Mb	46

torch.empty_strided

torch.empty_strided(*size*, *stride*, *, *dtype=None*, *layout=None*, *device=None*, *requires_grad=False*, *pin_memory=False*) → Tensor

Creates a tensor with the specified `size` and `stride` and filled with undefined data.

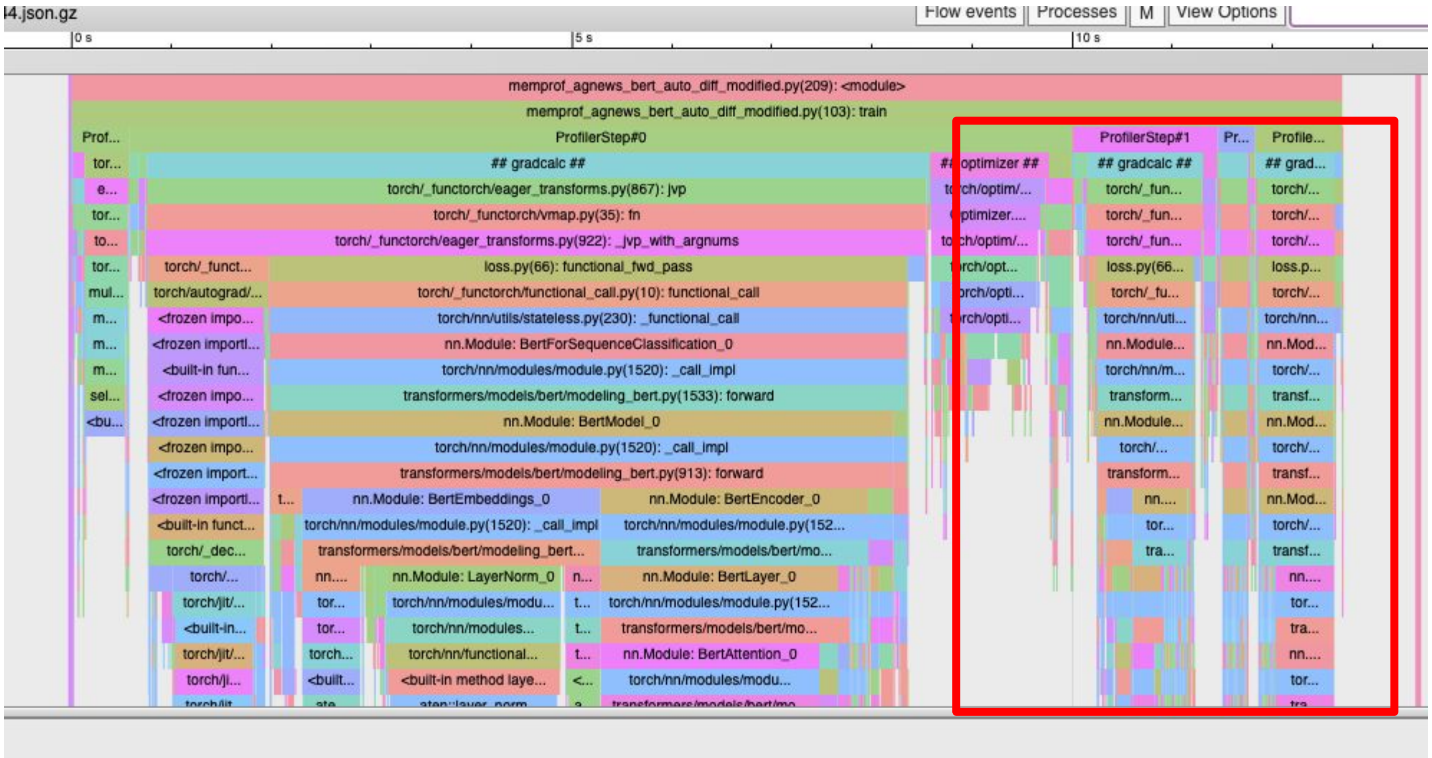
WARNING

If the constructed tensor is "overlapped" (with multiple indices referring to the same element in memory) its behavior is undefined.

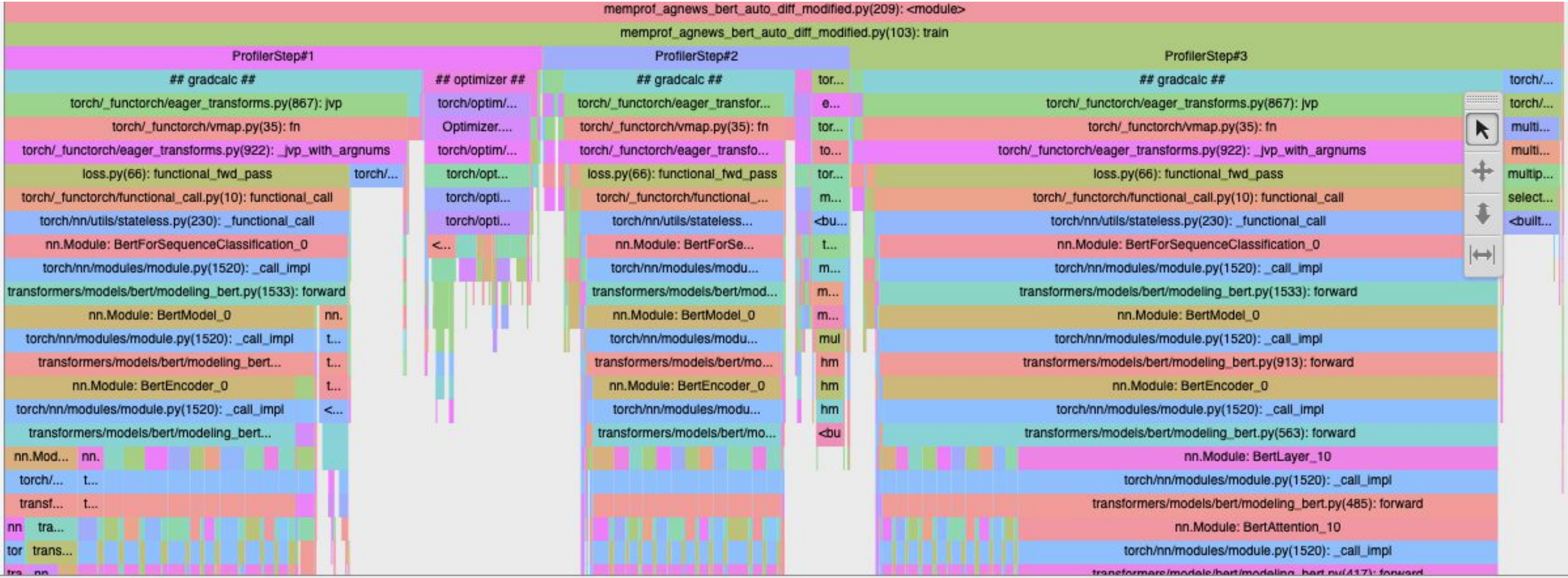
NOTE

If `torch.use_deterministic_algorithms()` and `torch.utils.deterministic.fill_uninitialized_memory` are both set to `True`, the output tensor is initialized to prevent any possible nondeterministic behavior from using the data as an input to an operation. Floating point and complex tensors are filled with NaN, and integer tensors are filled with the maximum value.

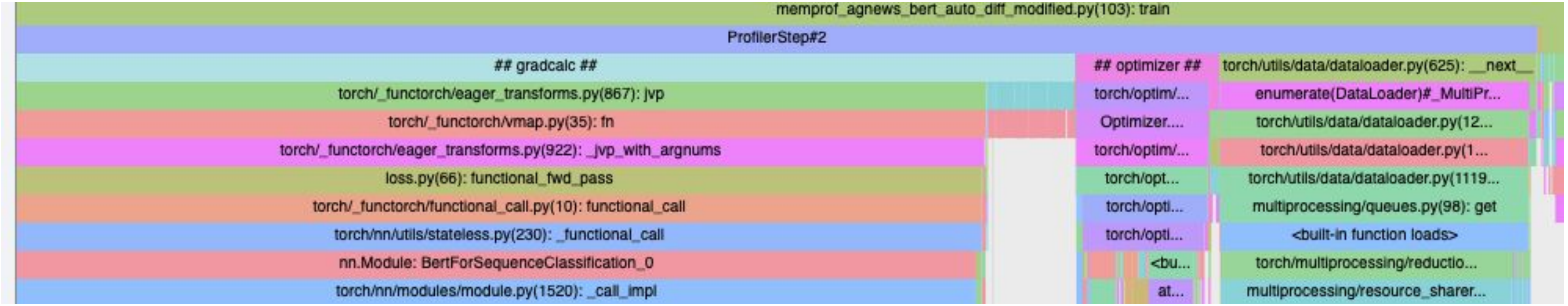
Exploring EventList with chrome traces

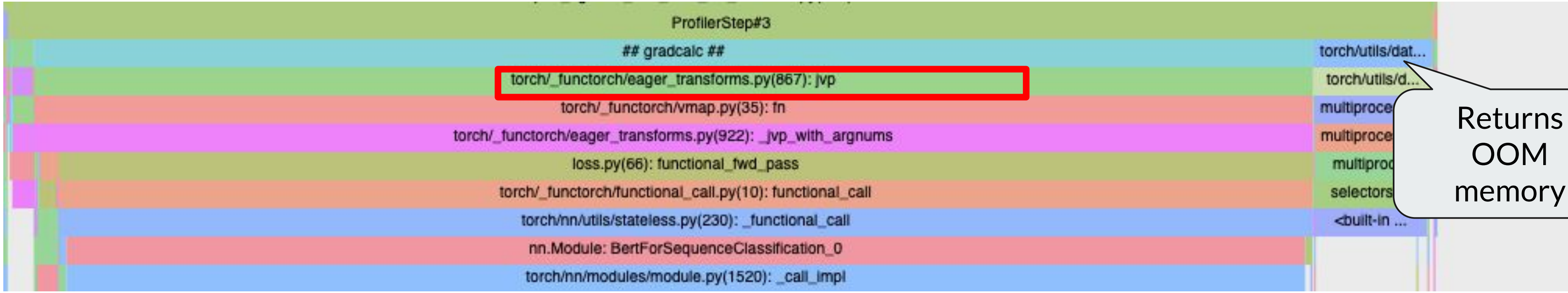


Exploring EventList with chrome traces

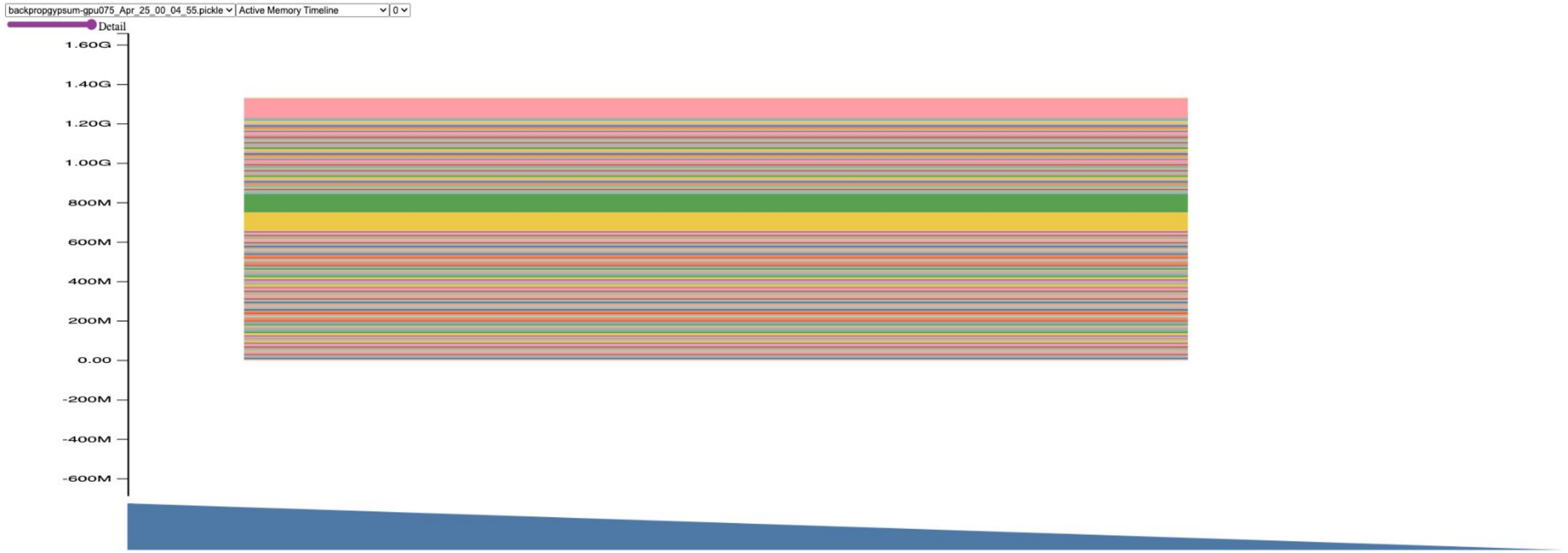


Exploring EventList with chrome traces



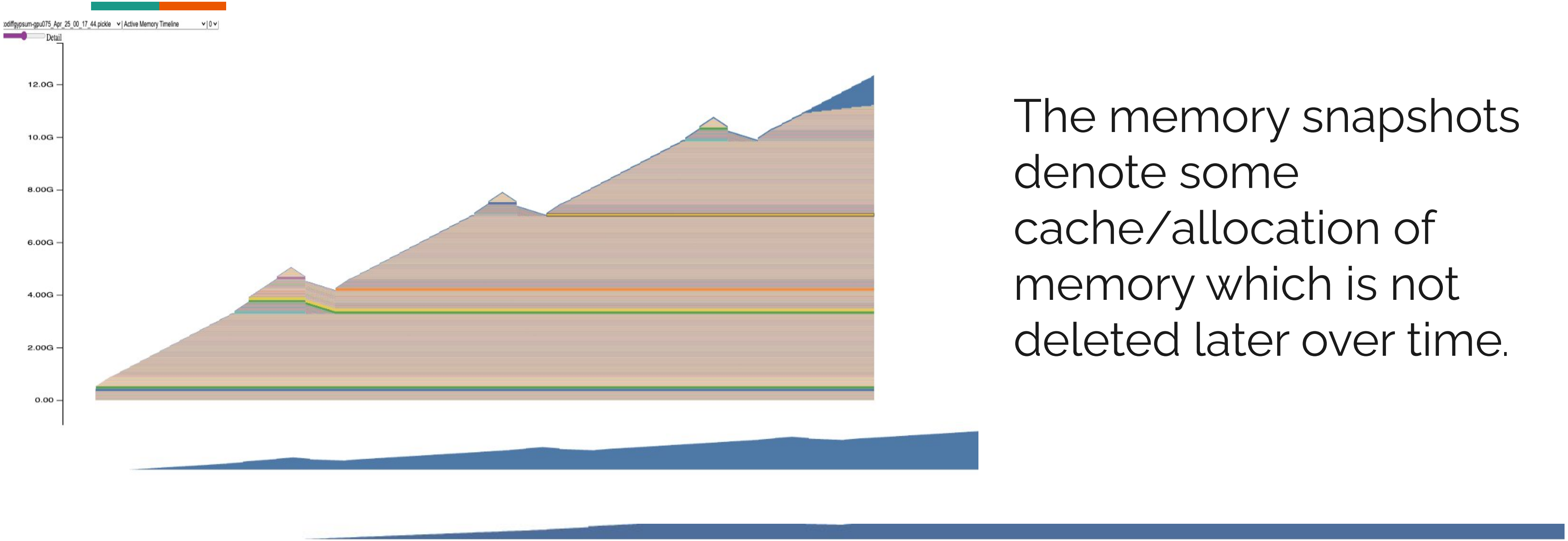


Memory Snapshotting Backward Mode AD



https://pytorch.org/memory_viz

Memory snapshotting Forward Mode AD



The memory snapshots denote some cache/allocation of memory which is not deleted later over time.

```
5004 Addr: b43c0b00000_1, Size: 9.0MiB (9437184 bytes) allocation, Total memory used after allocation: 6.7GiB (7215659056 bytes)
CUDACachingAllocator.cpp:0:c10::cuda::CUDACachingAllocator::Native::DeviceCachingAllocator::malloc(int, unsigned long, CUstream_st*)
:0:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::malloc(void**, int, unsigned long, CUstream_st*)
:0:c10::cuda::CUDACachingAllocator::Native::NativeCachingAllocator::allocate(unsigned long) const
:0:at::TensorBase at::detail::_empty_strided_generic<c10::ArrayRef<long> >(c10::ArrayRef<long>, c10::ArrayRef<long>, c10::Allocator*, c10::DispatchKeySet, c10::ScalarType)
??:0:at::detail::empty_strided_generic(c10::ArrayRef<long>, c10::ArrayRef<long>, c10::Allocator*, c10::DispatchKeySet, c10::ScalarType)
??:0:at::detail::empty_strided_cuda(c10::ArrayRef<long>, c10::ArrayRef<long>, c10::ScalarType, c10::optional<c10::Device>)
??:0:at::detail::empty_strided_cuda(c10::ArrayRef<long>, c10::ArrayRef<long>, c10::optional<c10::ScalarType>, c10::optional<c10::Layout>, c10::optional<c10::Device>, c10::optional<bool>)
??:0:at::native::empty_strided_cuda(c10::ArrayRef<long>, c10::ArrayRef<long>, c10::optional<c10::ScalarType>, c10::optional<c10::Layout>, c10::optional<c10::Device>, c10::optional<bool>)
```

Approach 2: Dual Numbers implementation -pytorch

```
import torch.nn as nn

model = nn.Linear(5, 5)
input = torch.randn(16, 5)

params = {name: p for name, p in model.named_parameters()}
tangents = {name: torch.rand_like(p) for name, p in params.items()}

with fwAD.dual_level():
    for name, p in params.items():
        delattr(model, name)
        setattr(model, name, fwAD.make_dual(p, tangents[name]))

    out = model(input)
    jvp = fwAD.unpack_dual(out).tangent
```

3.1.1 Dual Numbers

Mathematically, forward mode AD (represented by the left- and right-hand sides in Table 2) can be viewed as evaluating a function using dual numbers,[10] which can be defined as truncated Taylor series of the form

$$v + \dot{v}\epsilon \,,$$

where $v, \dot{v} \in \mathbb{R}$ and ϵ is a nilpotent number such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. Observe, for example, that

$$\begin{aligned} (v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) &= (v + u) + (\dot{v} + \dot{u})\epsilon \\ (v + \dot{v}\epsilon)(u + \dot{u}\epsilon) &= (vu) + (v\dot{u} + \dot{v}u)\epsilon \,, \end{aligned}$$

in which the coefficients of ϵ conveniently mirror symbolic differentiation rules (e.g., Eq. 3). We can utilize this by setting up a regime where

$$f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon \tag{5}$$

and using dual numbers as data structures for carrying the tangent value together with the primal.[11] The chain rule works as expected on this representation: two applications of Eq. 5 give

$$\begin{aligned} f(g(v + \dot{v}\epsilon)) &= f(g(v) + g'(v)\dot{v}\epsilon) \\ &= f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon \,. \end{aligned}$$

The coefficient of ϵ on the right-hand side is exactly the derivative of the composition of f

Atlim et.al, Symbolic Computation (cs.SC) 2018

Beta implementation -pytorch - Results

Implementation Variant	GPU Memory Utilization
Dual Number Implementation (Beta)	1,896 MiB
JVP Implementation - without profiling	4,152 MiB
JVP Implementation - with profiling	12,193 MiB
Backpropagation - with profiling	2,463 MiB

- OOM error with profiling
- Dual Number implementation consumes lesser memory than Backpropagation.

Potential Problems

1. Pytorch JVP() might have bad memory management.
   a. Emptying cache, forcing garbage collection didn’t work.
2. OOM occurs only while profiling JVP implementation ⇒ Profiler Memory leak
3. The allocated memory in JVP implementation is not being cleaned up later.

Approach 3: Use JAX

Challenges:

- PyTorch and JAX libraries are not meant to be used together.
- We cannot directly use the AutoModelForSequenceClassification from the Hugging Face Transformers library in JAX.
- Less documentation or examples available

Solution?

Implement a model in both Pytorch JVP and Jax JVP and compare results

Implementation

- Model: BertForSequenceClassification, FlaxBertForSequenceClassification from 🤗 Transformers
- Model checkpoint - bert-base-uncased
- Training methods: backprop, Forward mode AD (with JVP)

Pytorch JVP

- Examples per batch 8
- Agnews - classification

- Epochs 5
- Training samples 1000
- Testing samples 100

	Pytorch - backprop	Pytorch - JVP	JAX - backprop	JAX- JVP
Peak memory	2458MiB	4488MiB		
time	69.35 s	89.67 s		

Future Work

- Explore jax.jacfwd implementation.
- Dwelling into the implementation of JVP in Pytorch and Jax to address memory issues.
- Check if Pytorch Profiler has a memory leak.

References

1. Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research, 18*(1), 1-43. Link

2. Brady, N. W., Mees, M., Vereecken, P. M., & Safari, M. (2021). Implementation of Dual Number Automatic Differentiation with John Newman's BAND Algorithm. Journal of The Electrochemical Society, 168(11), 113501. Link


Any Questions?