

Общие методы для более быстрого обучения и более эффективных моделей

Егор Швецов

tg: @dalime e-mail: e.shvetsov@skoltech.ru

План на сегодня:

- Данные
 - Дата loaders
 - Выбор данных и активное обучение
- Про ускорение работающих моделей
 - Torch JIT, Tensor RT и ONNX, Torch Dynamo

>>>

Данные

Размер данных: датасеты часто превышают емкость локального дискового хранилища, что требует распределенных систем хранения и эффективного доступа к сети.

Скорость передачи данных: ограниченная скорость I/O.

Аугментация и перемешивание: данные для обучения необходимо перемешивать и аугментировать.

Масштабируемость: пользователи часто хотят разрабатывать и тестировать небольшие наборы данных, а затем быстро масштабировать их до больших наборов данных.

- WebDataset
- TFRecord
- MXNet RecordIO
- FFCV

- Разобьем датасет на шарды
- Соберем все в один файл

Возможность поиска/индексируемость.

Хороший формат данных должен также изначально поддерживать быстрый доступ только к определенному подмножеству данных.

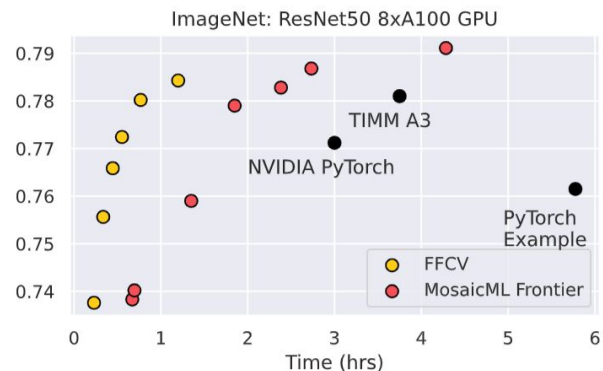
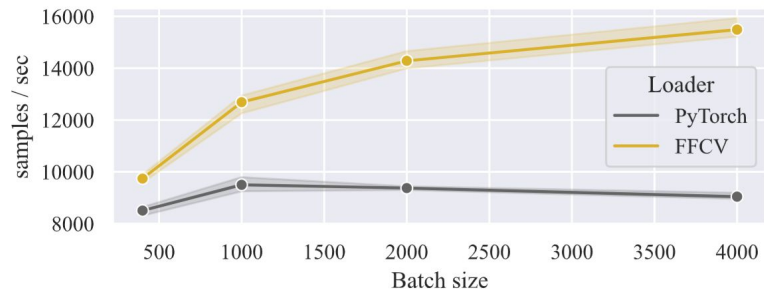
FFCV — библиотека для простого и быстрого обучения моделей машинного обучения.

FFCV ускоряет обучение модели за счет устранения (часто незаметных) узких мест в загрузке данных из процесса обучения.

В частности:

- Эффективный формат хранения файлов
- Кэширование
- Предварительная загрузка
- Асинхронная передача данных
- JIT компиляция для загрузки данных

ImageNet ResNet-50 75% за 20 минут на одной машине.



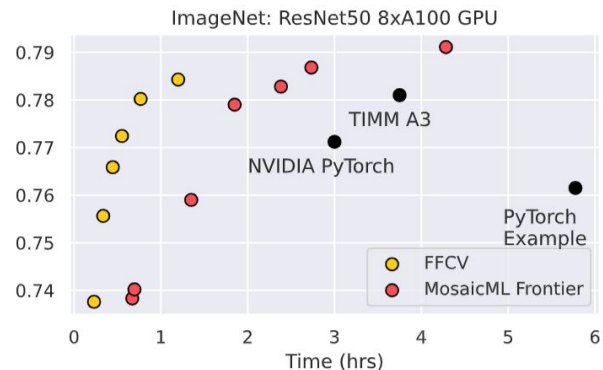
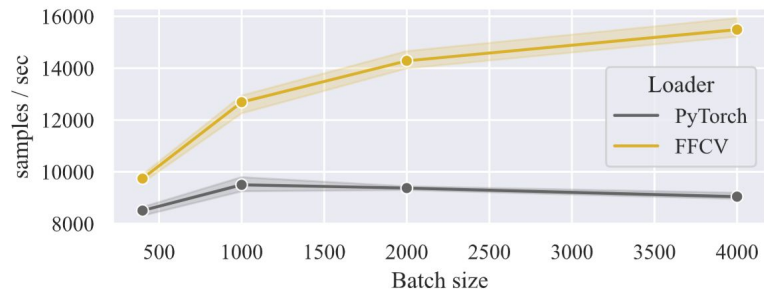
FFCV — библиотека для простого и быстрого обучения моделей машинного обучения.

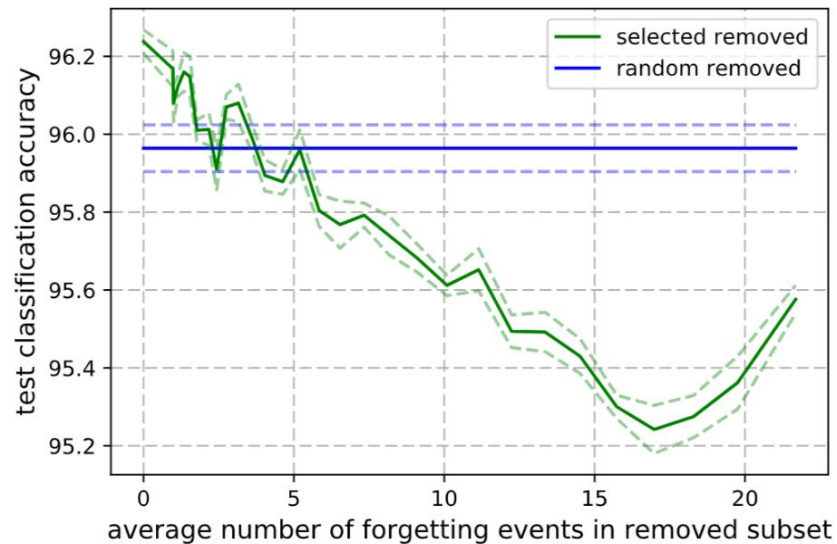
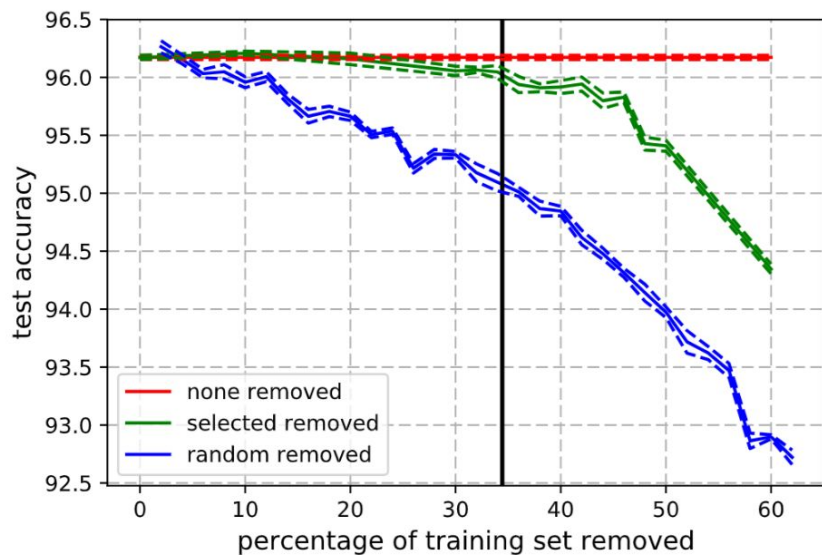
FFCV ускоряет обучение модели за счет устранения (часто незаметных) узких мест в загрузке данных из процесса обучения.

В частности:

- Эффективный формат хранения файлов
- Кэширование
- Предварительная загрузка
- Асинхронная передача данных
- JIT компиляция для загрузки данных

ImageNet ResNet-50 75% за 20 минут на одной машине.





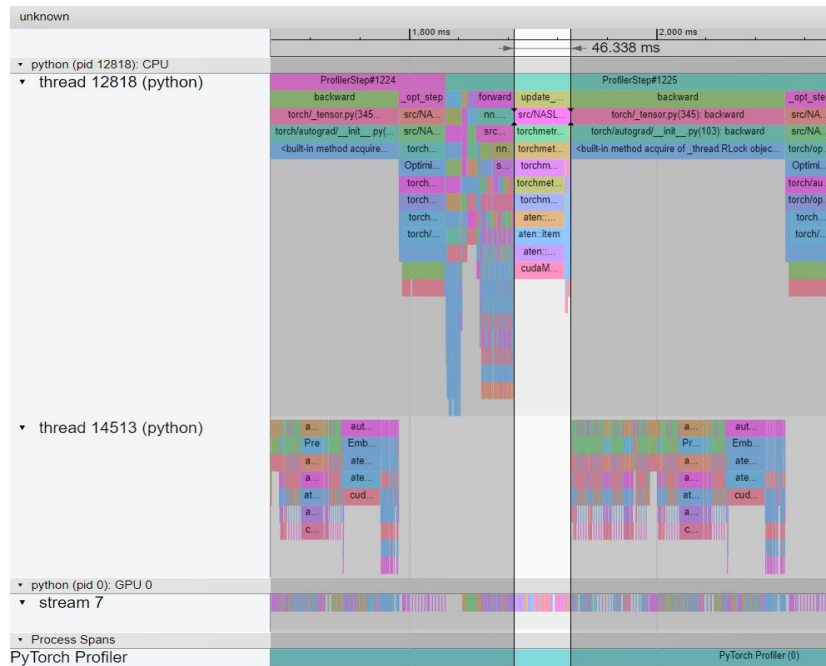
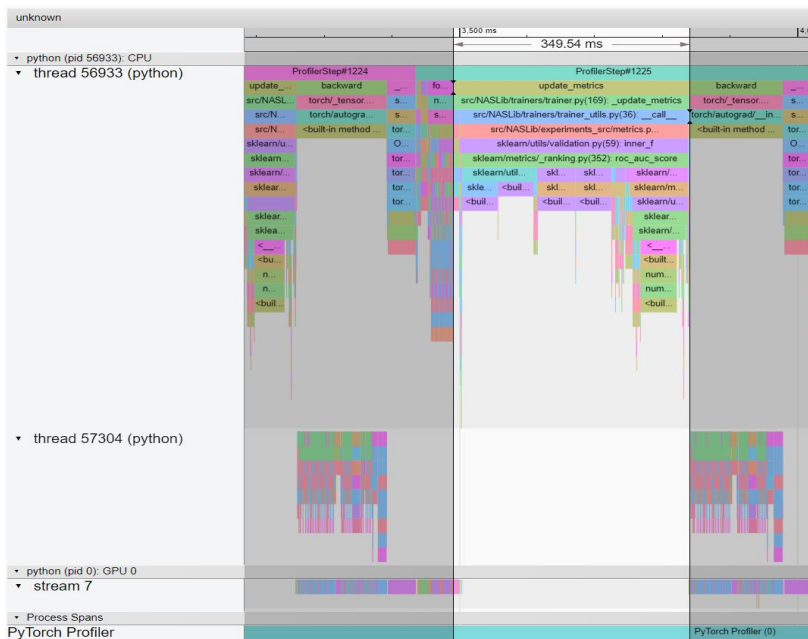
AN EMPIRICAL STUDY OF EXAMPLE FORGETTING DURING DEEP NEURAL NETWORK LEARNING

>>>

Case Study

Профилировка:

Для определения медленных участков кода была проведена профилировка. Были обнаружены 2 слабых места: sklearn метрики и загрузка данных. Первая проблема ощущалась наиболее остро, т.к. сильно замедляла обучение со временем (рисунки до/после оптимизации ниже, ускорение с ~350 до ~46 ms). Решением первой проблемы был переход к torchmetrics, второй - webdataset. Переход к torchmetrics обусловлен также тем, что данная библиотека поддерживает ddp, mixed precision и работу на GPU.



Для ускорения обучения были имплементированы multi-gpu и ddp методы.

Multi-gpu подразумевает обучение одной архитектуры на одной видеокарте, ddp (distribute data parallel) - обучение одной модели на нескольких видеокартах путем распределения батча между видеокартами. Второй метод используется в тех случаях, когда использование первого невозможно, например в методе Diff NAS.

Также для ускорения были заменены torch.nn.LayerNorm на apex FusedLayerNorm от Nvidia и оптимизаторы SGD и ADAM на FusedSGD и FusedADAM.

Поддерживаемые методы ускорения поиска оптимальной архитектуры

Для ускорения обучения были имплементированы multi-gpu и ddp методы.

Multi-gpu подразумевает обучение одной архитектуры на одной видеокарте, ddp (distribute data parallel) - обучение одной модели на нескольких видеокартах путем распределения батча между видеокартами. Второй метод используется в тех случаях, когда использование первого невозможно, например в методе Diff NAS.

Также для ускорения были заменены torch nn.LayerNorm на apex FusedLayerNorm от Nvidia и оптимизаторы SGD и ADAM на FusedSGD и FusedADAM.

Поддерживаемые методы ускорения поиска оптимальной архитектуры

Оптимизация по памяти с mixed precision и FusedLayerNorm на примере датасета amex и трансформера.


Dataset	Model	Mixed Precision	FusionLayerNorm	Batch	GPU mem	Saved mem
amex	EncoderDecoderModel	False	False	512	4.0 GB	0 GB
amex	EncoderDecoderModel	True	False	512	4.4 GB	- 0.4 GB
amex	EncoderDecoderModel	True	True	512	2.9 GB	1.1 GB
amex	EncoderDecoderModel	False	False	1024	6.9 GB	0 GB
amex	EncoderDecoderModel	True	False	1024	7.8 GB	- 0.9 GB
amex	EncoderDecoderModel	True	True	1024	4.7 GB	2.3 GB

Вывод: переход от ADAM к FusedADAM дает ускорение до 30%, а замена LayerNorm сильно уменьшает потребляемую видеокартой память (из-за особенности реализации архитектуры трансформера) для mixed precision. На следующих слайдах будут приведены результаты ускорения.

Оптимизация по времени с использованием FusedAdam на примере датасета amex и трансформера.

Dataset	Model	Mixed Precision	FusedAdam	Batch	time/epoch	Speedup
amex	EncoderDecoderModel	False	False	512	498 sec	0 %
amex	EncoderDecoderModel	True	False	512	387 sec	22 %
amex	EncoderDecoderModel	True	True	512	342 sec	<u>31 %</u>
amex	EncoderDecoderModel	False	False	1024	380 sec	0 %
amex	EncoderDecoderModel	True	False	1024	320 sec	16 %
amex	EncoderDecoderModel	True	True	1024	280 sec	26 %


Вывод: переход от ADAM к FusedADAM дает ускорение до **30%**, а замена **LayerNorm** сильно уменьшает потребляемую видеокартой память (из-за особенности реализации архитектуры трансформера) для mixed precision. На следующих слайдах будут приведены результаты ускорения.

 Apex


0.1

Search docs

AMP: AUTOMATIC MIXED PRECISION

 apex.amp

DISTRIBUTED TRAINING

 apex.parallel

FUSED OPTIMIZERS

apex.optimizers

FUSED LAYER NORM

apex.normalization.fused_layer_norm

[Docs](#) » Apex (A PyTorch Extension)[Edit on GitHub](#)

Apex (A PyTorch Extension)

This site contains the API documentation for Apex (<https://github.com/nvidia/apex>), a Pytorch extension with NVIDIA-maintained utilities to streamline mixed precision and distributed training. Some of the code here will be included in upstream Pytorch eventually. The intention of Apex is to make up-to-date utilities available to users as quickly as possible.

Installation instructions can be found here: <https://github.com/NVIDIA/apex#quick-start>.

Some other useful material, including GTC 2019 and Pytorch DevCon 2019 Slides, can be found here: https://github.com/mcarilli/mixed_precision_references.

AMP: Automatic Mixed Precision

- [apex.amp](#)

Distributed Training

- [apex.parallel](#)

Fused Optimizers

- [apex.optimizers](#)

Fused Layer Norm

- [apex.normalization.fused_layer_norm](#)

Indices and tables

- [Index](#)
- [Module Index](#)

[Next](#) ➞<https://nvidia.github.io/apex/>

```
>>> JIT - trace - compile  
& repeat
```


Compiler	Interpreter
Компилятор работает со всем кодом сразу	Работает с кодом по мере того как читает его
Компилятор генерирует машинный код	Не создает промежуточно представления в виде машинного кода
Компилятор подходит для продакшена	Хорошо подходит для быстрой разработки

	Programming Languages	Compilers and Interpreters version
Compiled	C, C++ Go Rust	gcc version 6.3.1 20161221- go version go1.7.5 rustc version 1.18.0
Semi-Compiled	VB.NET C# Java	mono version 4.4.2.0 (vbnc) mono version 4.4.2.0 (mics) javac version 1.8.0_131
Interpreted	JavaScript Perl PHP Python R Ruby Swift	node version 6.10.3 perl version 5.24.1 php version 7.0.19 python version 2.7.13 Rscript version 3.3.3 ruby version 2.3.3p222 swift version 3.0.2

Compiler

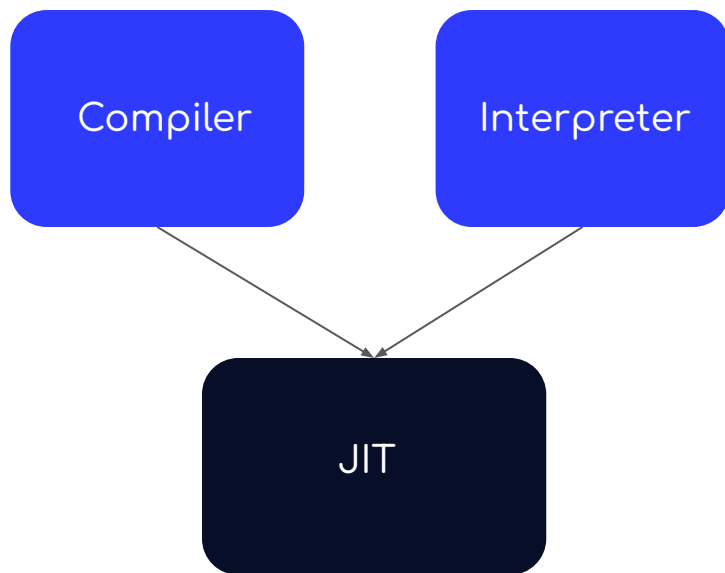
Компиляторы преобразуют код языка высокого уровня в машинный код за один сеанс. (1) Компиляторам может потребоваться некоторое время, поскольку им приходится сразу транслировать код высокого уровня на машинный язык более низкого уровня, а затем сохранять исполняемый объектный код в памяти. (2) Компилятор создает машинный код, который выполняется на процессоре с определенной архитектурой набора инструкций (ISA), которая зависит от процессора. Например, вы не сможете скомпилировать код для x86 и запустить его на архитектуре MIPS без специального компилятора. (3) Компиляторы также зависят от платформы. То есть компилятор может преобразовать, например, C++ в машинный код, предназначенный для платформы, на которой работает ОС Linux. Однако кросс-компилятор может генерировать код для платформы, отличной от той, на которой он работает сам.

Интерпретаторы

Существует несколько типов интерпретаторов:

- Синтаксически-управляемый интерпретатор (т. е. интерпретатор абстрактного синтаксического дерева (AST))
- Интерпретатор байт-кода и многопоточный интерпретатор (не путать с потоками параллельной обработки)
- JIT-интерпретатор (разновидность гибридного интерпретатора/компилятора) и некоторые другие.

Примерами языков программирования, использующих интерпретаторы, являются Python, Ruby, Perl и PHP.



Just-in-time - компиляция «точно в срок» — это метод повышения производительности интерпретируемых программ. Во время выполнения программа может быть скомпилирована в нативный код для повышения производительности..

Статическая компиляция преобразует код в язык для конкретной платформы. **Интерпретатор непосредственно выполняет исходный код.**

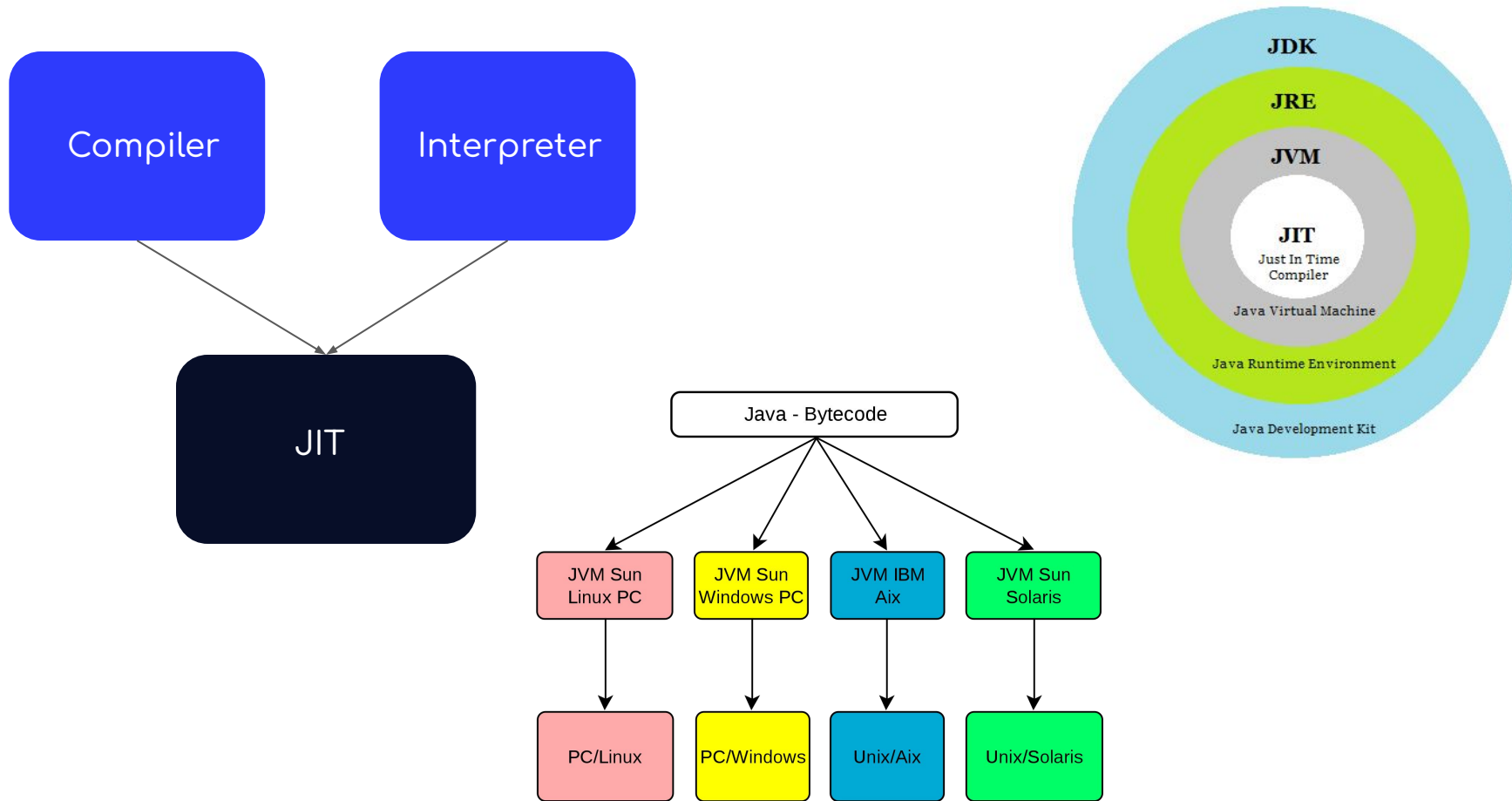
JIT-компиляция пытается использовать преимущества обоих. Пока интерпретируемая программа выполняется, JIT-компилятор определяет наиболее часто используемый код и компилирует его в машинный код.

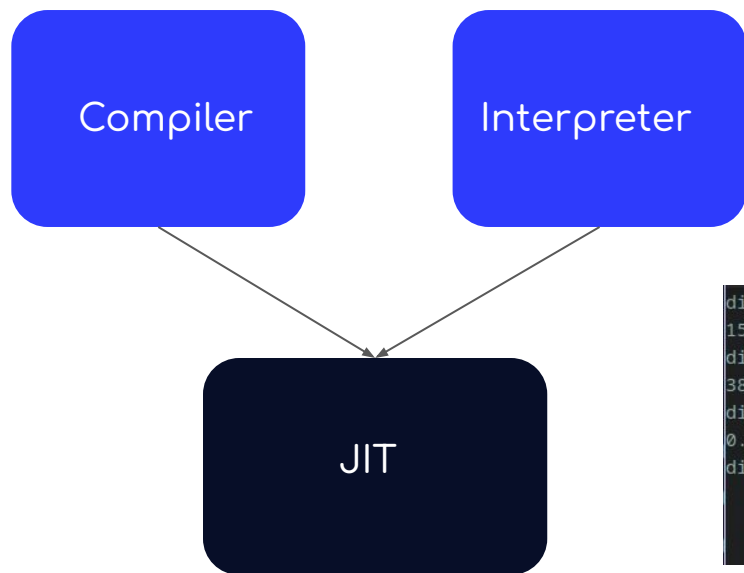
В зависимости от компилятора это преобразование может выполняться для небольшого участка кода или для всего.

Что важно понимать о JIT-компиляции, так это то, что она скомпилирует байт-код в инструкции машинного кода работающей машины. Это означает, что полученный машинный код оптимизирован для архитектуры ЦП работающей машины.

Динамический анализ делает JIT очень эффективным:

- Например. он запускает код и видит, что код использует только целые числа, поэтому мы можем оптимизировать эту часть.
- Он содержит больше информации, чем компилятор.





```
diff7@droid: time python3 add.py 8
15.234
diff7@droid: time pypy3 --jit off add.py 8
38.172
diff7@droid: time pypy3 add.py 8
0.225
diff7@droid:
```

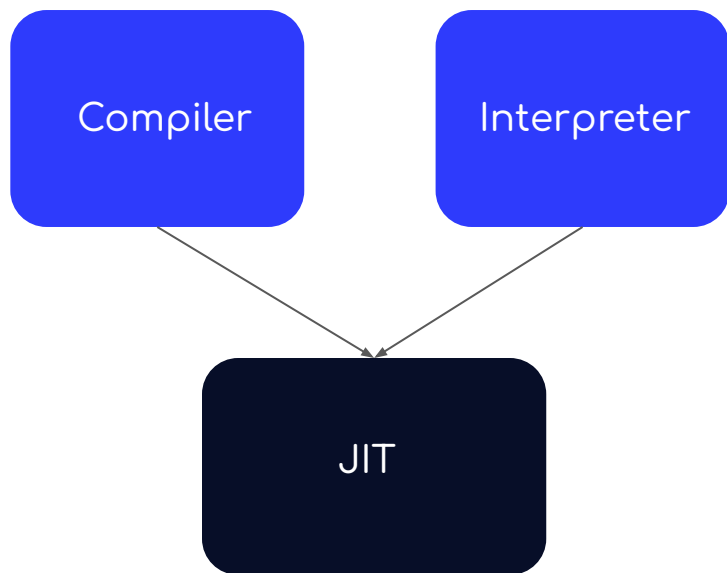
```
import sys
s = int(sys.argv[1])

def my_add(x,y):
    return x+y

a = 0
for _ in range(int(10**s)):
    a+=my_add(1,10)
```

Dynamic Analysis makes JIT very effective:

- E.g. it runs code and sees that it uses only integers, so we can optimize that part.
- It has more information than a compiler



```
diff7@droid: time python3 add.py 8
15.234
diff7@droid: time pypy3 --jit off add.py 8
38.172
diff7@droid: time pypy3 add.py 8
0.225
diff7@droid:
```

```
import sys
s = int(sys.argv[1])

def my_add(x,y):
    return x+y

a = 0
for _ in range(int(10**s)):
    a+=my_add(1,10)
```

```
diff7@droid: time python3 add_random.py 5
0.044
diff7@droid: time pypy3 add_random.py 5
0.897
diff7@droid: █
```

```
import sys
import random

s = int(sys.argv[1])

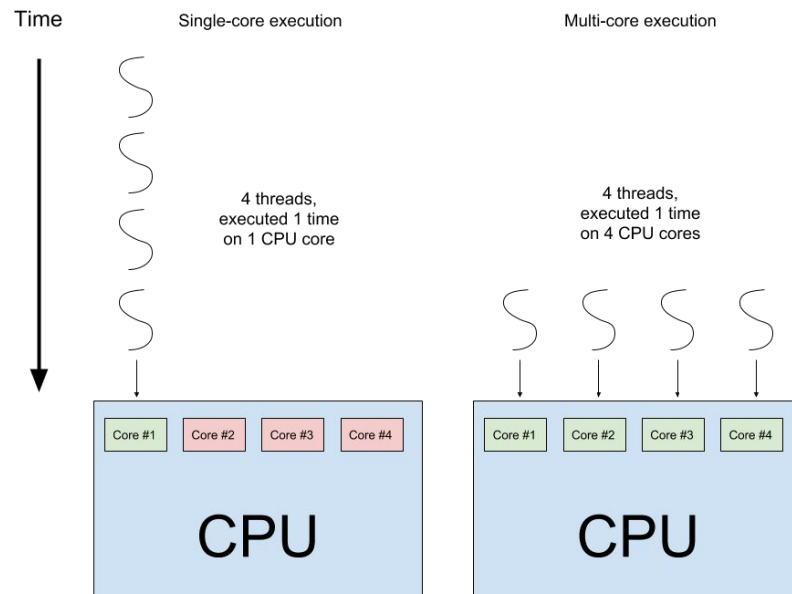
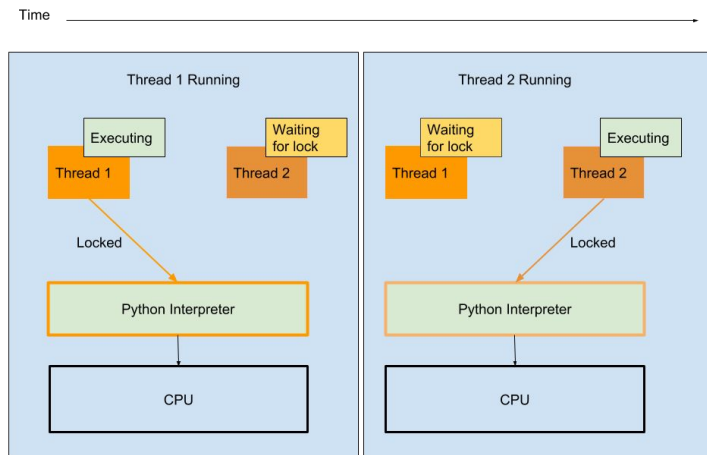
def my_add(x,y):
    return x+y

a = 0
w = ''
for _ in range(int(10**s)):
    if random.random() > 0.5:
        w+=my_add('a','b')
    else:
        a+=my_add(1,1)
```

Dynamic Analysis makes JIT very effective:

- E.g. it runs code and sees that it uses only integers, so we can optimize that part.
- It has more information than a compiler

Python Global Interpreter Lock (GIL)



К чему это все?

К чему это все?

- Хотим уметь перенести код в другую среду выполнения (любую)

К чему это все?

- Хотим уметь перенести код в другую среду выполнения (любую)
- Хотим адаптировать код под текущее железо (любое)

К чему это все?

- Хотим уметь перенести код в другую среду выполнения (любую)
- Хотим адаптировать код под текущее железо (любое)
- Хотим адаптировать код в принципе (чуть позже о том как)

К чему это все?

- Хотим уметь перенести код в другую среду выполнения (любую)
- Хотим адаптировать код под текущее железо (любое)
- Хотим адаптировать код в принципе (чуть позже о том как)
- Хотим избавиться от Python GIL

К чему это все?

- Хотим уметь перенести код в другую среду выполнения (любую)
- Хотим адаптировать код под текущее железо (любое)
- Хотим адаптировать код в принципе (чуть позже о том как)
- Хотим избавиться от Python GIL
- Хотим чтобы это было удобно

Dynamic control flow	<p>Когда выполнение зависит от данных</p> <pre>if x[0] == 4: x += 1</pre>
Tracing	Метод экспорта. Он запускает модель с определенными входными данными и «отслеживает/записывает» все выполняемые операции в граф.
Scripting	Еще один способ экспорта. Он анализирует исходный код модели на Python и компилирует код в граф.
TorchScript	TorchScript дает нам представление, в котором мы можем оптимизировать код компилятором, чтобы обеспечить более эффективное выполнение.
<code>torch.jit.trace</code>	При использовании <code>torch.jit.trace</code> вы предоставляете свою модель и образец входных данных в качестве аргументов. Входные данные будут передаваться через модель, как при обычном запуске, выполненные операции будут отслеживаться и записываться в TorchScript. Логическая структура будет заморожена в пути, выбранном во время выполнения.
<code>torch.jit.script</code>	При использовании <code>torch.jit.script</code> вы просто указываете свою модель в качестве аргумента. TorchScript будет сгенерирован в результате статической проверки содержимого <code>nn.Module</code> (рекурсивно).
<code>torch.fx.symbolic_trace</code> (другой зверь, но тоже рядом)	<p><code>torch.fx</code> это платформа для Python-to-Python преобразований кода PyTorch. TorchScript, с другой стороны, больше ориентирован на перемещение программ PyTorch за пределы Python для целей развертывания.</p> <p>В этом смысле FX и TorchScript ортогональны друг другу и даже могут быть составлены друг из друга (например, преобразовывать программы PyTorch с помощью FX, а затем экспортировать их в TorchScript для развертывания).</p> <p>Одно из применений <code>torch.fx</code> это создание графа вычислений и манипуляция этим графом.</p>



```

graph(%0 : Float(3, 10), %1 : Float(3, 20), %2 : Float(3, 20), %3 : Float(80, 10)
      %4 : Float(80, 20), %5 : Float(80), %6 : Float(80)) {
  %7 : Float(10!, 80!) = aten::t(%3)
  %10 : int[] = prim::ListConstruct(3, 80)
  %12 : Float(3!, 80) = aten::expand(%5, %10, 1)
  %15 : Float(3, 80) = aten::addmm(%12, %0, %7, 1, 1)
  %16 : Float(20!, 80!) = aten::t(%4)
  %19 : int[] = prim::ListConstruct(3, 80)
  %21 : Float(3!, 80) = aten::expand(%6, %19, True)
  %24 : Float(3, 80) = aten::addmm(%21, %1, %16, 1, 1)
  %26 : Float(3, 80) = aten::add(%15, %24, 1)
  %29 : Dynamic[] = aten::chunk(%26, 4, 1)
  %30 : Float(3!, 20), %31 : Float(3!, 20), %32 : Float(3!, 20), %33 : Float(3!, 20) = prim::ListUnpack(%29)
  %34 : Float(3, 20) = aten::sigmoid(%30)
  %35 : Float(3, 20) = aten::sigmoid(%31)
  %36 : Float(3, 20) = aten::tanh(%32)
  %37 : Float(3, 20) = aten::sigmoid(%33)
  %38 : Float(3, 20) = aten::mul(%35, %32)
  %39 : Float(3, 20) = aten::mul(%34, %36)
  %41 : Float(3, 20) = aten::add(%38, %39, 1)
  %42 : Float(3, 20) = aten::tanh(%41)
  %43 : Float(3, 20) = aten::mul(%37, %42)
  return (%43, %41);
}
  
```

А в чем всетаки разница?

`torch.jit.trace`

`torch.jit.script`



А когда нам лучше
“поймать” граф
вычислений?

`torch.jit.script` фиксирует как операции, так и полную условную логику вашей модели.

Но только в случае если ваша модель не использует не [Pytorch функционал](#) и ее логика ограничена [поддерживаемым подмножеством функций и синтаксиса Python](#).

Если что-то не поддерживается или не стандартно то нам надо переписывать код, это работает в большинстве случаев.

Поддерживает только статическую типизацию.

`torch.jit.trace` не учитывает динамически меняющуюся структуру, которая зависит от входных данных. Может сгенерировать код, только по одному графу вычислений и не выдать ошибок.

```
def f1(x, y):  
    if x.sum() < 0:  
        return -y  
    return our_lib.squeeze(y)
```

`torch.jit.trace` запомнит только один путь

`torch.jit.script` не будет работать с какой-то непонятной библиотекой если она не на чистом Python или Pytorch. Например часть `Scipy` на `C++`.

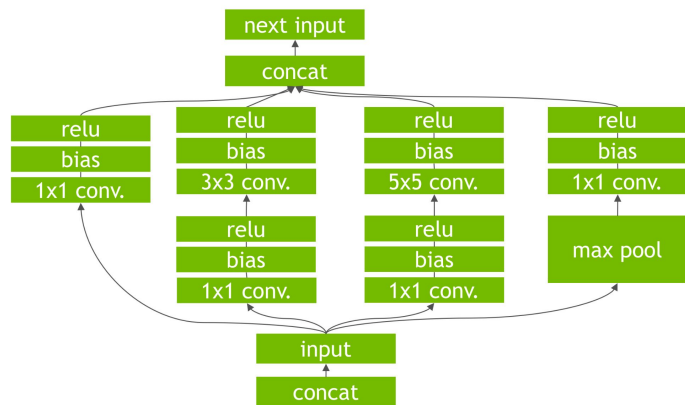


Pytorch 2.0 - Torch Dynamo

Разобьет граф
вычислений на кусочки,
будет работать только
с тем с чем может.



Как происходит оптимизация?

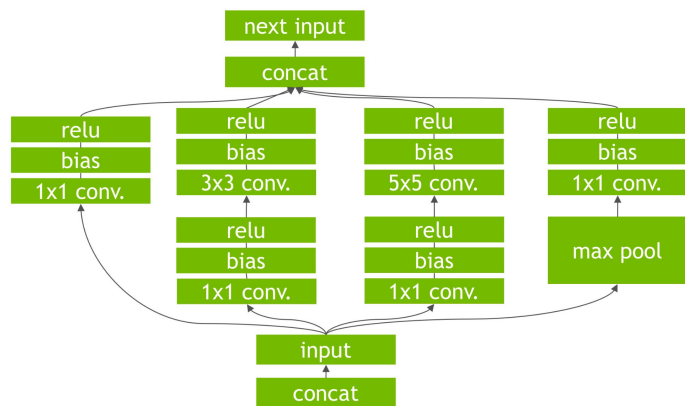


Vertical Fusion

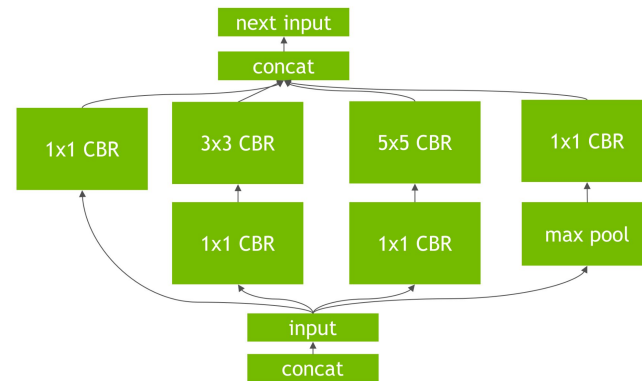


Horizontal Fusion



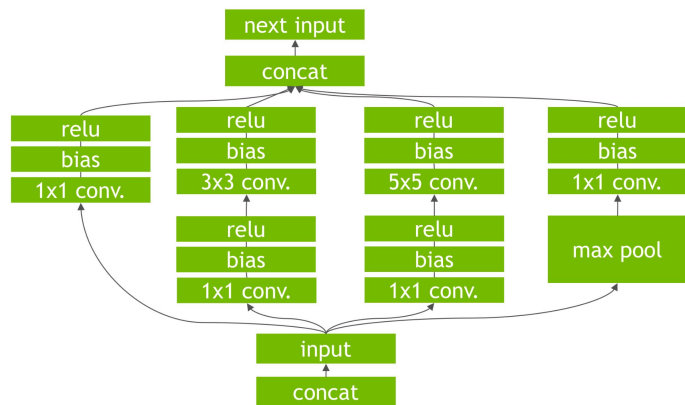


Vertical Fusion

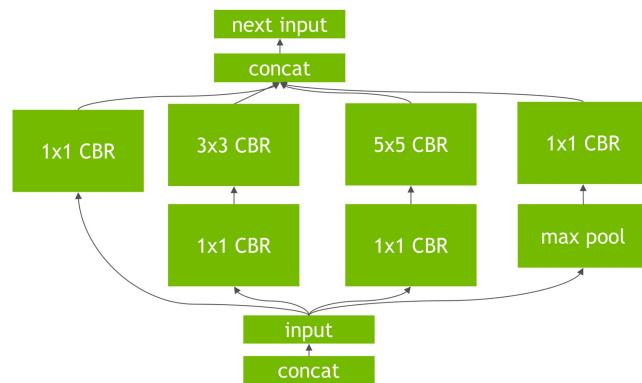


Horizontal Fusion

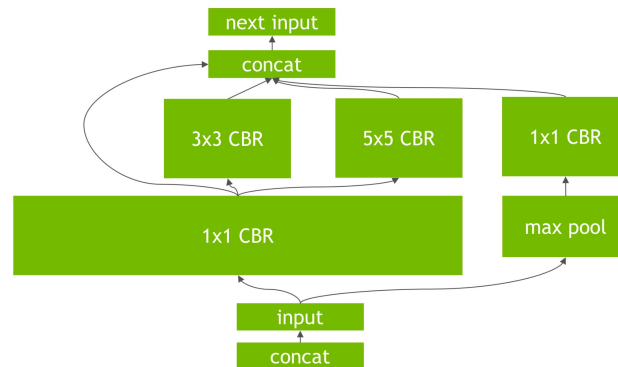




Vertical Fusion



Horizontal Fusion



- Fusion
- Algebraic rewriting
- Loop unrolling
- Automatic work placement
- Out of order execution
- Allow Multithreading
- TorchScript automatically optimizes common patterns

ResNet 18

```

1 import torchvision
2 import torch
3 from time import perf_counter
4 import numpy as np
5
6 def timer(f,*args):
7     start = perf_counter()
8     f(*args)
9     return (1000 * (perf_counter() - start))
10
11 # Example 1.1 Pytorch cpu version
12
13 model_ft = torchvision.models.resnet18(pretrained=True)
14 model_ft.eval()
15 x_ft = torch.rand(1,3, 224,224)
16 np.mean([timer(model_ft,x_ft) for _ in range(10)])
17
18 # Example 1.2 Pytorch gpu version
19
20 model_ft_gpu = torchvision.models.resnet18(pretrained=True).cuda()
21 x_ft_gpu = x_ft.cuda()
22 model_ft_gpu.eval()
23 np.mean([timer(model_ft_gpu,x_ft_gpu) for _ in range(10)])
24
25 # Example 2.1 torch.jit.script cpu version
26
27 script_cell = torch.jit.script(model_ft, (x_ft))
28 np.mean([timer(script_cell,x_ft) for _ in range(10)])
29
30 # Example 2.2 torch.jit.script gpu version
31
32 script_cell_gpu = torch.jit.script(model_ft_gpu, (x_ft_gpu))
33 np.mean([timer(script_cell_gpu,x_ft.cuda()) for _ in range(100)])

```

ResNet-truncated.py hosted with ❤ by GitHub

[view raw](#)

	Latency on CPU (ms)	Latency on GPU(ms)
PyTorch	25.96	4.02
TorchScript	23.01	2.41

ResNet example for PyTorch and Script

BERT

```

1 from transformers import BertTokenizer, BertModel
2 import numpy as np
3 import torch
4 from time import perf_counter
5
6 def timer(f,*args):
7
8     start = perf_counter()
9     f(*args)
10    return (1000 * (perf_counter() - start))
11
12 script_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', torchscript=True)
13 script_model = BertModel.from_pretrained("bert-base-uncased", torchscript=True)
14
15
16 # Tokenizing input text
17 text = "[CLS] Who was Jim Henson ? [SEP] Jim Henson was a puppeteer [SEP]"
18 tokenized_text = script_tokenizer.tokenize(text)
19
20 # Masking one of the input tokens
21 masked_index = 8
22
23 tokenized_text[masked_index] = '[MASK]'
24
25 indexed_tokens = script_tokenizer.convert_tokens_to_ids(tokenized_text)
26
27 segments_ids = [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
28
29 # Creating a dummy input
30 tokens_tensor = torch.tensor([indexed_tokens])
31 segments_tensors = torch.tensor([segments_ids])

```

TorchScript-BERT-Example1.1.py hosted with ❤ by GitHub

[view raw](#)

```

1 # Example 1.1 BERT on CPU
2
3 native_model = BertModel.from_pretrained("bert-base-uncased")
4 np.mean([timer(native_model,tokens_tensor,segments_tensors) for _ in range(100)])
5
6 # Example 1.2 BERT on GPU
7 # Both sample data model need be on the GPU device for the inference to take place
8 native_gpu = native_model.cuda()
9 tokens_tensor_gpu = tokens_tensor.cuda()
10 segments_tensors_gpu = segments_tensors.cuda()
11 np.mean([timer(native_gpu,tokens_tensor_gpu,segments_tensors_gpu) for _ in range(100)])

```

TorchScript-BERT-PyTorch-Example1-2.py hosted with ❤ by GitHub

[view raw](#)

```

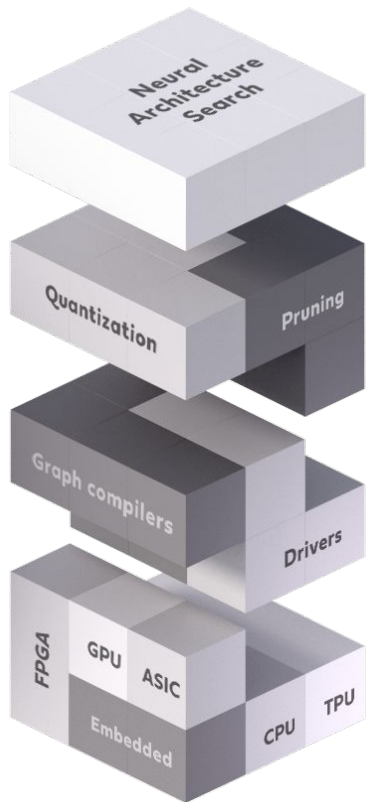
1 # Example 2.1 torch.jit.trace on CPU
2
3 traced_model = torch.jit.trace(script_model, [tokens_tensor, segments_tensors])
4
5 np.mean([timer(traced_model,tokens_tensor,segments_tensors) for _ in range(100)])
6
7 # Example 2.2 torch.jit.trace on GPU
8
9 traced_model_gpu = torch.jit.trace(script_model.cuda(), [tokens_tensor.cuda(), segments_
10
11 np.mean([timer(traced_model_gpu,tokens_tensor.cuda(),segments_tensors.cuda()) for _ in r

```

TorchScript-Script-Example-1.3.py hosted with ❤ by GitHub

[view raw](#)

	Latency on CPU (ms)	Latency on GPU(ms)
PyTorch	86.23	16.49
TorchScript	81.57	10.54



HARDWARE

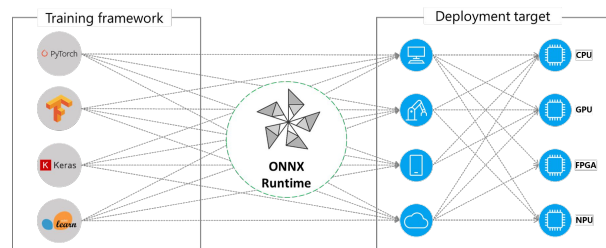
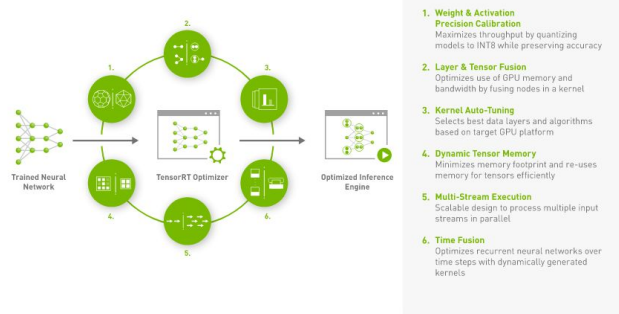
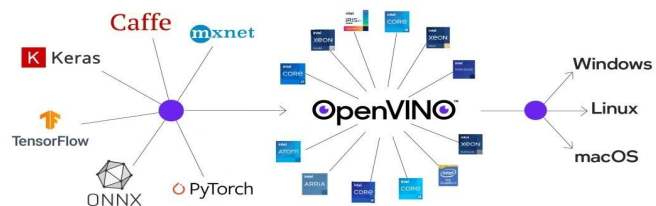
- GPU
- TPU
- CPU
- FPGA
- Ascend
- etc.

SOFTWARE

- CUDA
- Graph Compilers
- MKL - DNN
- etc.

ALGORITHMIC

- PRUNING
- QUANTIZATION
- NAS
- etc.



ONNX: Этот инструмент предоставляет стандартизированный формат для моделей глубокого обучения, что позволяет легко использовать их в различных средах и платформах. Часто ONNX используется для обеспечения плавного перехода моделей между различными средами.

ONNX Runtime	Открытый исходный код, первоначально созданный Microsoft и Facebook.	Может использоваться как интерфейс высокого уровня для TensorRT и OpenVino.
TensorRT	Nvidia	Этот инструмент специально разработан для графических процессоров NVIDIA и ориентирован на максимизацию пропускной способности и эффективности. Он оптимизирует модели нейронных сетей путем объединения слоев, выбора наиболее эффективных форматов данных и использования арифметики с пониженной точностью (FP16), где это возможно.
OpenVino	Intel	OpenVINO, разработанный Intel, специализируется на оптимизации моделей глубокого обучения для оборудования Intel, особенно процессоров. Это важнейший инструмент для повышения производительности моделей, в которых ресурсы графического процессора недоступны или ограничены.

Нам нужны промежуточные представления
что бы импортировать модель в другую
среду выполнения:

ONNX, Tensor RT, OpenVino и т.д.

И один из вариантов это сделать как раз
через `torch.jit.trace`

Memog	Ускорение инференс
<code>torch.compile</code> (Torch Dynamo)	~ 1.5 x (CPU и GPU)
ONNX Runtime + NVIDIA Triton server	~ 2 - 4 x (CPU и GPU)
Nvidia TensorRT + NVIDIA Triton server	~ 5 - 10 x (только GPU)

Memog	Ускорение на обучении
<code>torch.compile</code> (Torch Dynamo)	~1.25
TorchScript	~1.25
ONNX Runtime	~ 1.5
TensorRT	Только инференс

	TensorRT	ONNX
Лицензия	Apache 2.0 (optimization engine is closed source)	MIT
Легкость использования	сложна	терпимо
Документация	разбросано по кусочкам	становится лучше
Перформанс	Отличный	Хорошее

<https://github.com/ELS-RD/transformer-deploy/tree/main>