Общие методы для более быстрого обучения и более эффективных моделей

Егор Швецов

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

>>> Кейсы

Размер данных: датасеты часто превышают емкость локального дискового хранилища, поэтому нужны распределенные системы хранения и эффективный доступ 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

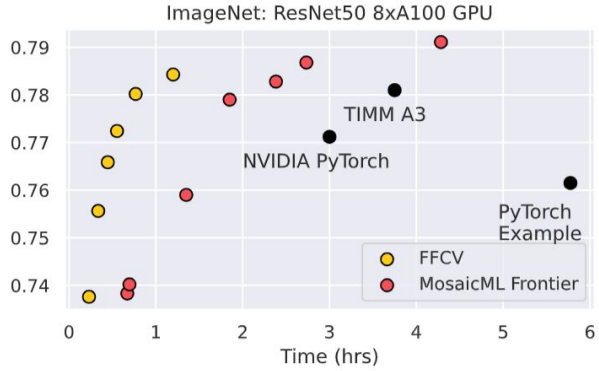
Хороший формат данных должен также изначально поддерживать быстрый доступ только к определенному подмножеству данных.

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

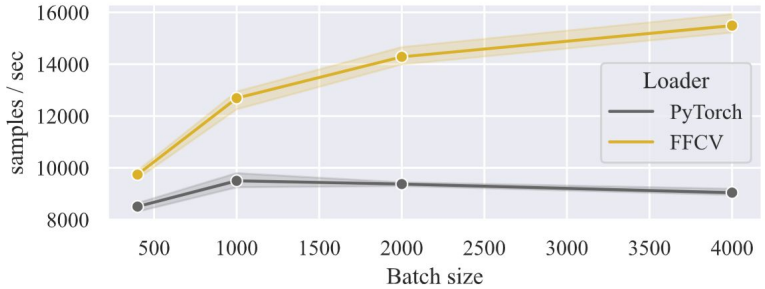
FFCV ускоряет обучение модели за счет устранения (часто незаметных) узких мест в загрузке данных из процесса обучения.

В частности:

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

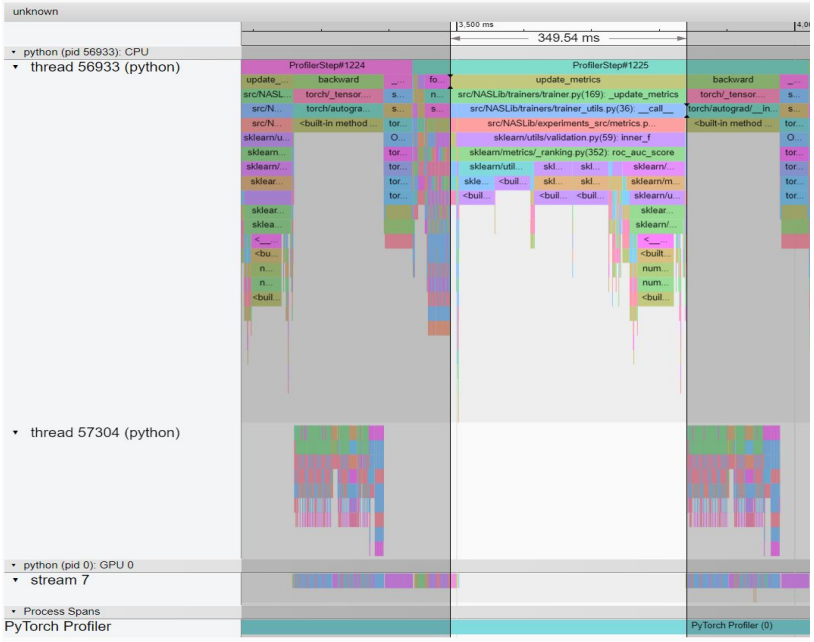


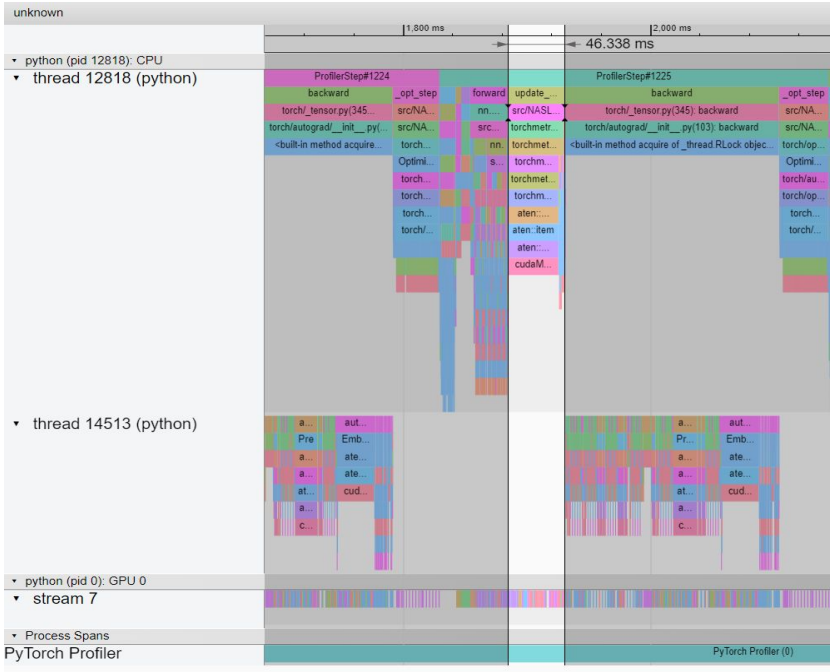
ImageNet ResNet-50 75% за 20 минут на одной машине.



Пример применения профилировки и дальше

Для определения медленных участков кода провели профилировку. Нашли два слабых места: sklearn-метрики и загрузка данных. Первая проблема ощущалась наиболее остро — 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 эта 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.

DDP нужен, когда multi-gpu нельзя использовать, например в методе Diff NAS.

Также для ускорения заменили torch nn.LayerNorm на арех FusedLayerNorm от Nvidia и оптимизаторы SGD и ADAM на FusedSGD и FusedADAM, которые быстрее.

ссылка

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

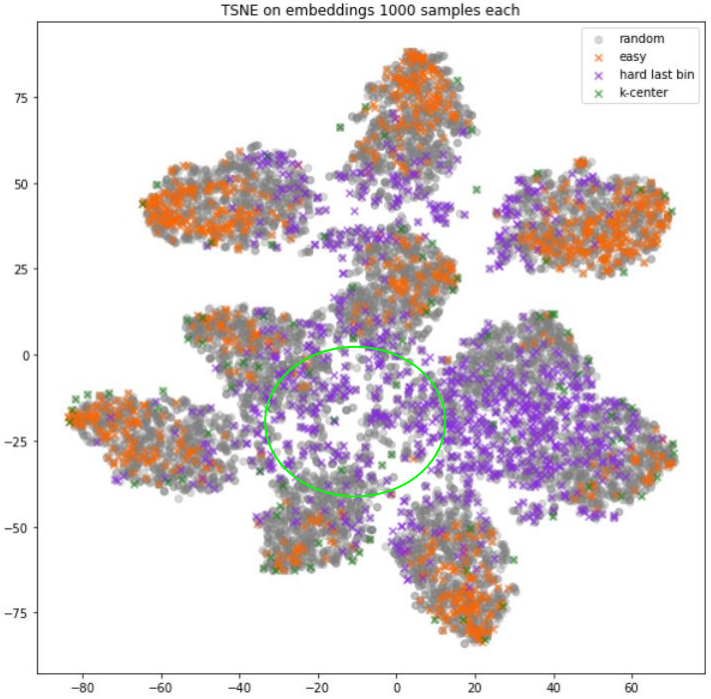
Вывод: переход от ADAM к FusedADAM дает ускорение до 30%, а замена **LayerNorm** сильно уменьшает потребляемую видеокартой память (из-за особенности реализации архитектуры трансформера) для mixed precision. На следующих слайдах будут результаты ускорения.

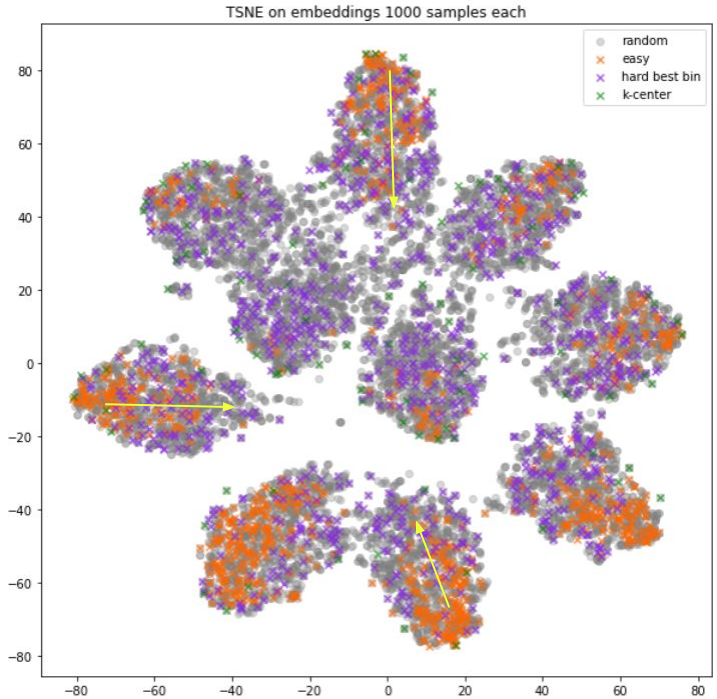
Оптимизация по времени с использованием FusedAdam на примере датасета amex и трансформера

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	31 %
amex	EncoderDecoderModel	False	False	1024	380 sec	0 %
amex	EncoderDecoderModel	True	False	1024	320 sec	16 %
amex	EncoderDecoderModel	True	True	1024	280 sec	26 %

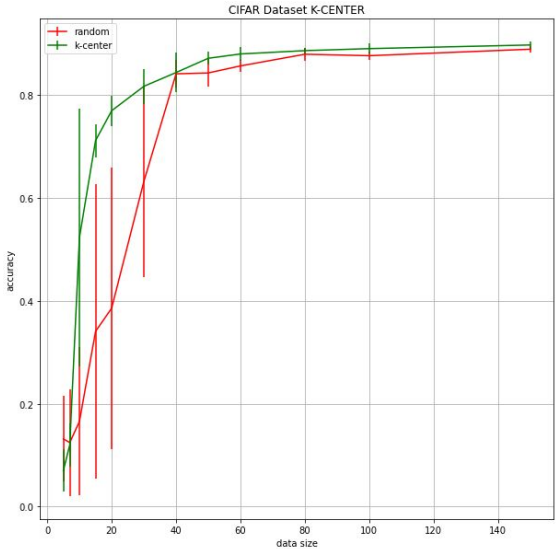
Вывод: переход от ADAM к FusedADAM дает ускорение до 30%, а замена **LayerNorm** сильно уменьшает потребляемую видеокартой память (из-за особенности реализации архитектуры трансформера) для mixed precision. На следующих слайдах будут результаты ускорения.

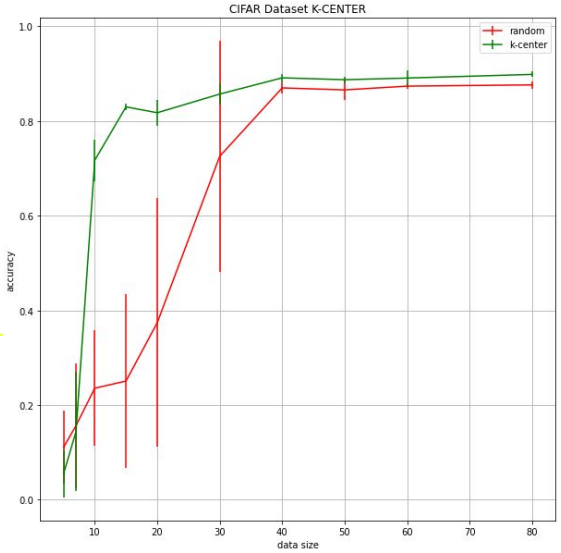
>>> Корсеты


TSNE on embeddings 1000 samples each
random
easy
hard last bin
k-center


TSNE on embeddings 1000 samples each
random
easy
hard best bin
k-center

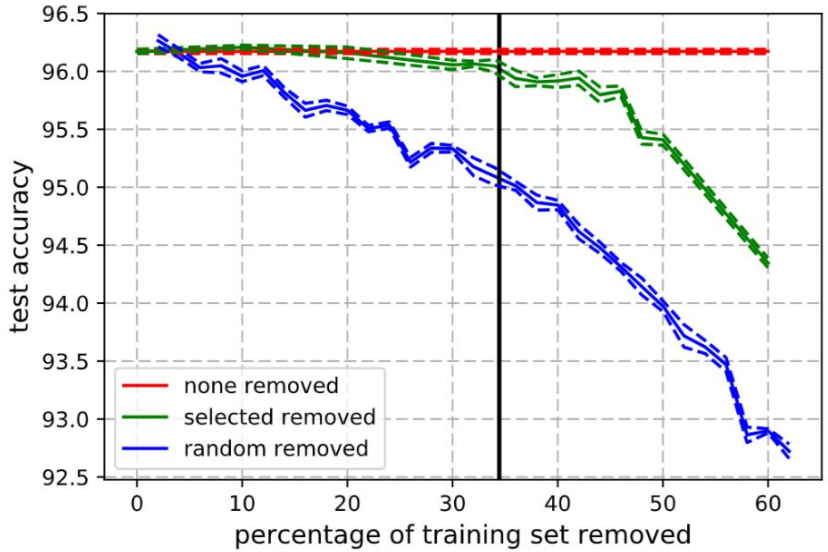
[Источник]


CIFAR Dataset K-CENTER
random
k-center
accuracy
data size
0.0
0.2
0.4
0.6
0.8
0
20
40
60
80
100
120
140


CIFAR Dataset K-CENTER
random
k-center
accuracy
data size
0.0
0.2
0.4
0.6
0.8
1.0
10
20
30
40
50
60
70
80

Источник

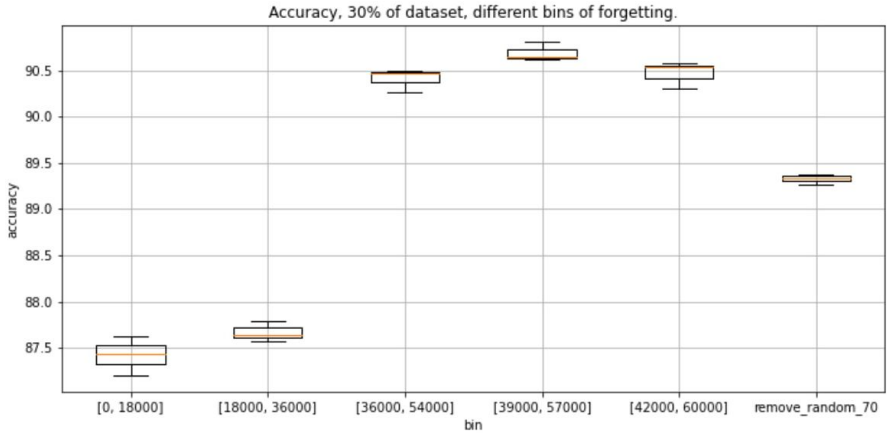
Точность на тесте



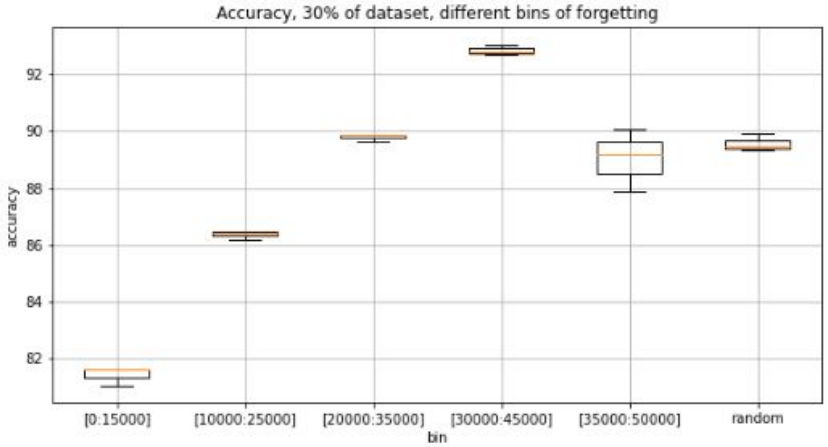
Процент удаленных данных

Источник

Fashion MNIST



Cifar 10



Источник

[1] Apricot: Submodular selection for data summarization in Python
[2] Estimating Example Difficulty Using Variance of Gradients
[3] Coresets for Data-efficient Training of Machine Learning Models
[4] Coresets for Robust Training of Deep Neural Networks against Noisy Labels
[5] Using Small Proxy Datasets to Accelerate Hyperparameter Search
[6] Rethinking Differentiable Search for Mixed-Precision Neural Networks
[7] ESTIMATING EXAMPLE DIFFICULTY USING VARIANCE OF GRADIENTS
[8] Selection via Proxy: Efficient Data Selection for Deep Learning
[9] Estimating Training Data Influence by Tracing Gradient Descent
[10] An Empirical Study of Example Forgetting during Deep Neural Network Learning
[11] Introduction to Core-sets: an Updated Survey
[12] Core-set Sampling for Efficient Neural Architecture Search
[13] The Implicit Bias of Gradient Descent on Separable Data

Источник

>>> JIT — trace — compile
& repeat

Compiler

vs

Interpreter

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

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

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

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

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

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

Компиляторы

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.

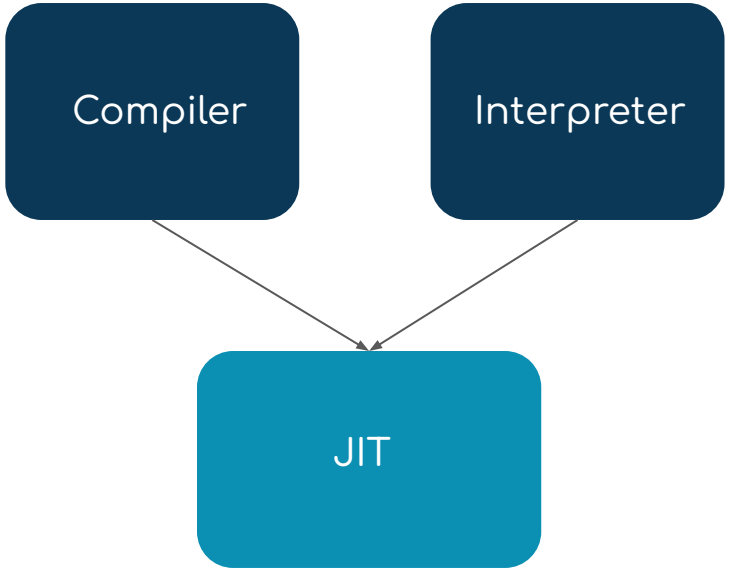
(3) **Компиляторы также зависят от платформы.** То есть компилятор может преобразовать, например, C++ в машинный код, предназначенный для платформы, на которой работает ОС Linux. Однако кросс-компилятор может генерировать код для платформы, отличной от той, на которой он работает сам.

Интерпретаторы

Существует несколько типов интерпретаторов:

- Синтаксически управляемый интерпретатор, то есть интерпретатор абстрактного синтаксического дерева (AST)
- Интерпретатор байт-кода и многопоточный интерпретатор.
- **JIT-интерпретатор (разновидность гибридного интерпретатора или компилятора) и некоторые другие**

  Примеры языков программирования, использующих интерпретаторы: 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 анализирует код и компилирует его в машинный код.

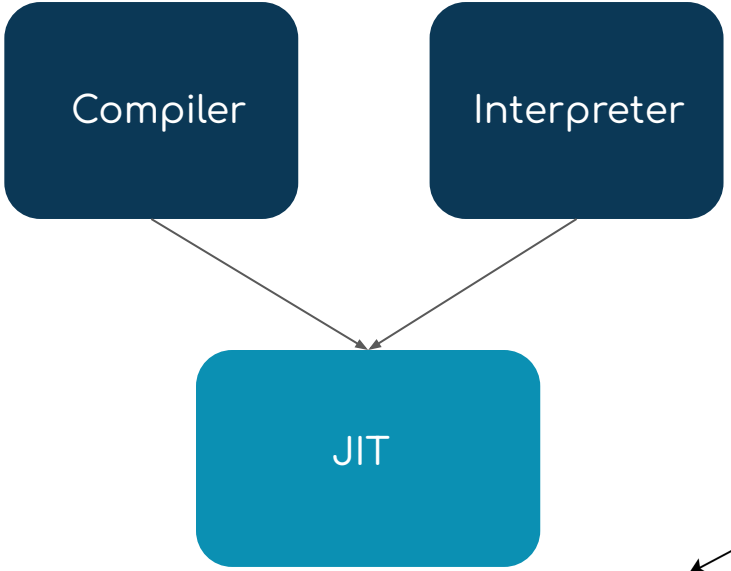
В зависимости от компилятора это преобразование может выполняться для небольшого участка кода или для всего кода.

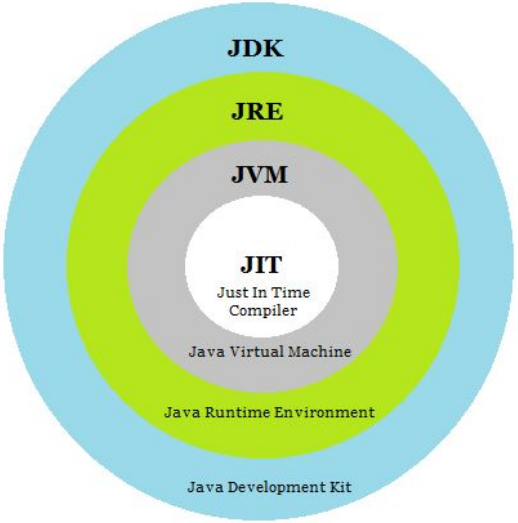
Что важно понимать о JIT-компиляции: она скомпилирует байт-код в инструкции машинного кода работающей машины. Это означает, что полученный машинный код оптимизирован для архитектуры ЦП работающей машины.

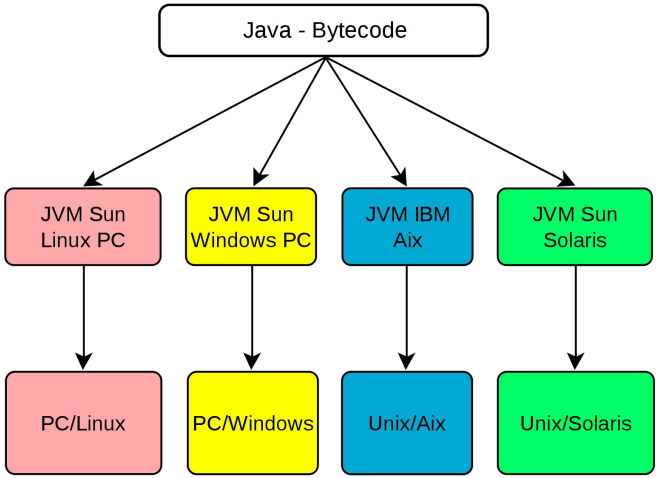
Динамический анализ делает JIT эффективным:

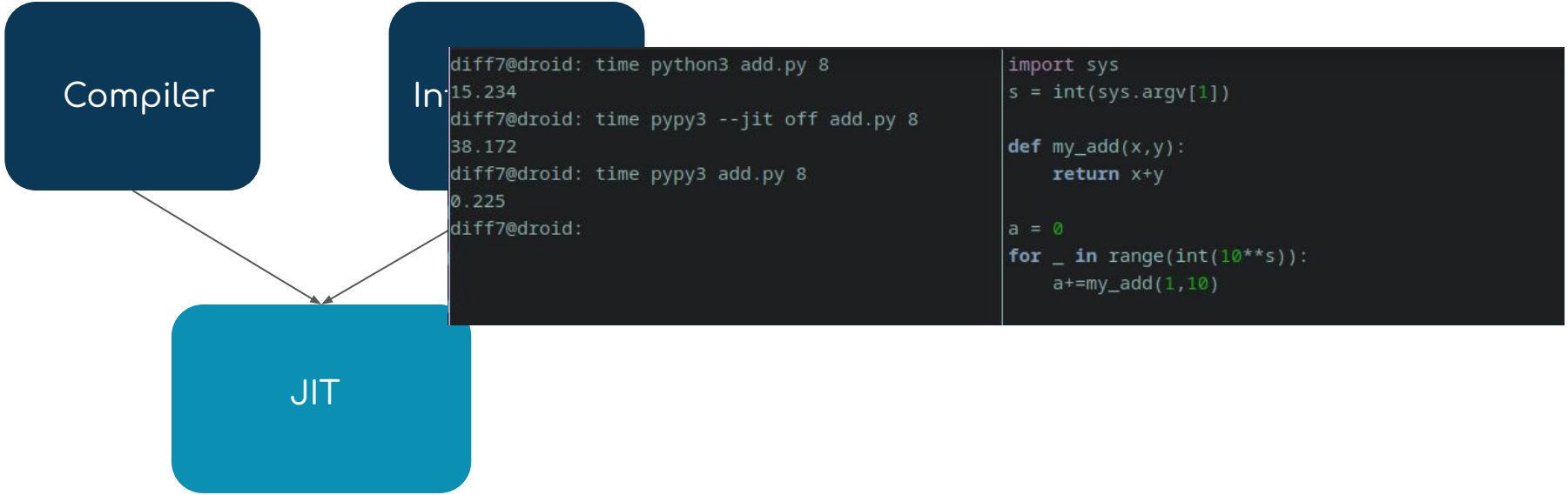
Например, он запускает код и видит, что он использует только целые числа, поэтому мы можем оптимизировать эту часть.

Получается, у него больше информации, чем у компилятора.


Compiler
Interpreter
JIT


JDK
JRE
JVM
JIT
Just In Time Compiler
Java Virtual Machine
Java Runtime Environment
Java Development Kit

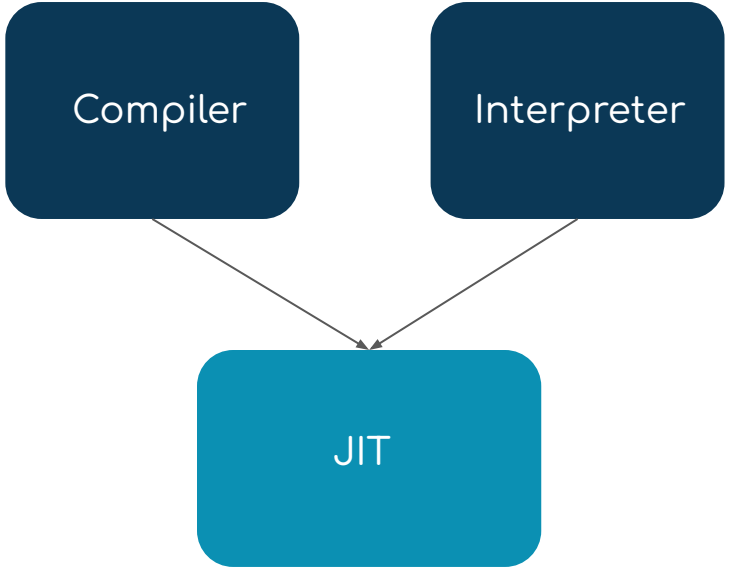

Java - Bytecode
JVM Sun Linux PC
JVM Sun Windows PC
JVM IBM Aix
JVM Sun Solaris
PC/Linux
PC/Windows
Unix/Aix
Unix/Solaris



Динамический анализ делает JIT эффективным:

Например, он запускает код и видит, что он использует только целые числа, поэтому мы можем оптимизировать эту часть.

Получается, у него больше информации, чем у компилятора.




```
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)
```

Динамический анализ делает JIT эффективным:

Например, он запускает код и видит, что он использует только целые числа, поэтому мы можем оптимизировать эту часть.

Получается, у него больше информации, чем у компилятора.

So, JIT is always better?

When compilation happens	At runtime, just before code is executed	Before runtime (e.g., during build/deployment)
Startup time	Slower initial startup (due to compilation overhead)	Faster startup (code is already compiled)
Peak performance	Can be very high (uses runtime profiling to optimize hot paths)	Good, but may miss runtime-specific optimizations
Memory usage	Higher (needs to store both bytecode and compiled native code)	Lower (only native code is needed)
Portability	High (distributes platform-independent bytecode)	Lower (must compile for each target platform)
Optimization opportunities	Can use actual runtime data (e.g., which branches are taken)	Limited to static analysis and assumptions

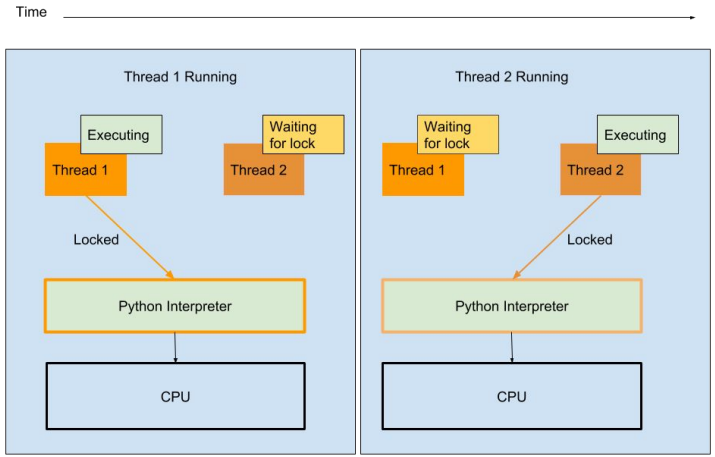
А что мы вообще хотим от наших компиляторов?

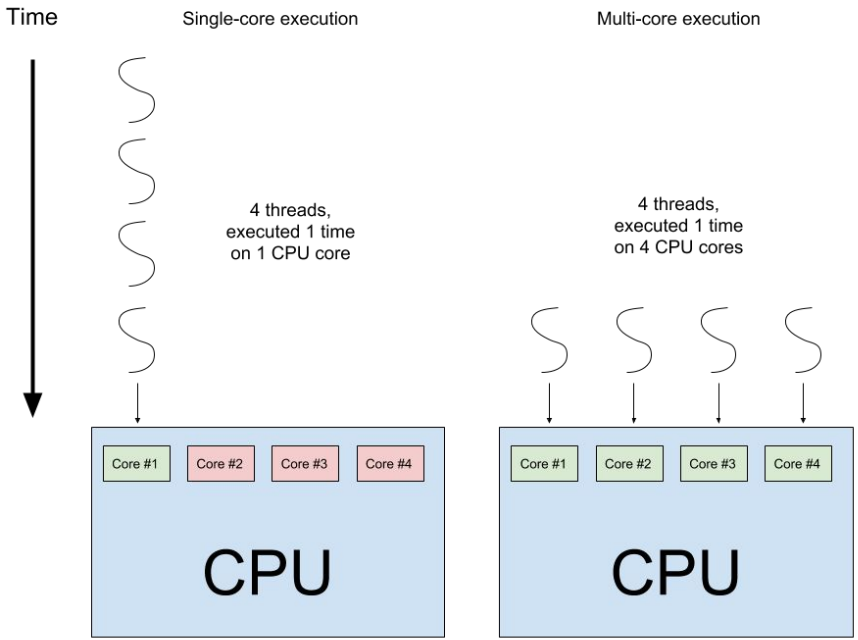
К чему это все?

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

В итоге, коллеги вдохновились тем что есть в других языках и решили сделать что-то похожее для торча.

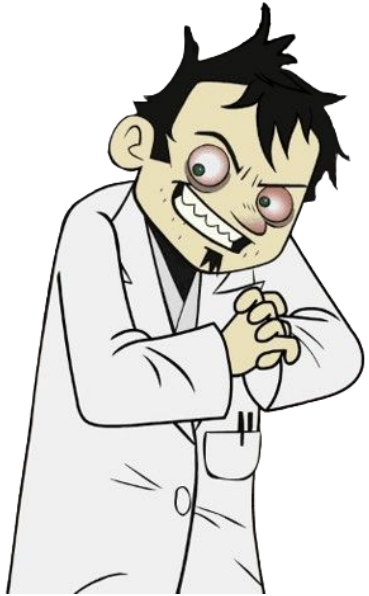
Глобальная блокировка интерпретатора Python (GIL)





Dynamic control flow	Когда выполнение зависит от данных if x[0] == 4: x += 1
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	Когда говорят TorchScript обычно имеют в виду graph-based intermediate representation (IR)
torch.jit.trace	При использовании torch.jit.trace 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 = Fails to capture dynamic control flow ==
torch.jit.script	При использовании torch.jit.script вы просто указываете свою модель в качестве аргумента. TorchScript - IR будет сгенерирован в результате статической проверки содержимого nn.Module (рекурсивно). == Not all Python features are supported ==
torch.fx.symbolic_trace (другой зверь, но тоже рядом)	torch.fx — 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. 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). Одно из применений torch.fx — это создание графа вычислений и манипуляция этим графом

Собственно и все



graph-based intermediate representation (IR)

Компилируем в граф

Оригинал

```
@torch.jit.script
def foo(len):
    # type: (int) ->
torch.Tensor
    rv = torch.zeros(3, 4)
    for i in range(len):
        if i < 10:
            rv = rv - 1.0
        else:
            rv = rv + 1.0
    return rv
print(foo.graph)
```

```
graph(%len.1 : int):
  %24 : int = prim::Constant[value=1]()
  %17 : bool = prim::Constant[value=1]() # test.py:10:5
  %12 : bool? = prim::Constant()
  %10 : Device? = prim::Constant()
  %6 : int? = prim::Constant()
  %1 : int = prim::Constant[value=3]() # test.py:9:22
  %2 : int = prim::Constant[value=4]() # test.py:9:25
  %20 : int = prim::Constant[value=10]() # test.py:11:16
  %23 : float = prim::Constant[value=1]() # test.py:12:23
  %4 : int[] = prim::ListConstruct(%1, %2)
  %rv.1 : Tensor = aten::zeros(%4, %6, %6, %10, %12) # test.py:9:10
  %rv : Tensor = prim::Loop(%len.1, %17, %rv.1) # test.py:10:5
    block0(%i.1 : int, %rv.14 : Tensor):
      %21 : bool = aten::lt(%i.1, %20) # test.py:11:12
      %rv.13 : Tensor = prim::If(%21) # test.py:11:9
        block0():
          %rv.3 : Tensor = aten::sub(%rv.14, %23, %24) # test.py:12:18
          -> (%rv.3)
        block1():
          %rv.6 : Tensor = aten::add(%rv.14, %23, %24) # test.py:14:18
          -> (%rv.6)
      -> (%17, %rv.13)
  return (%rv)
```

В чем все-таки 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 функционанал 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++.

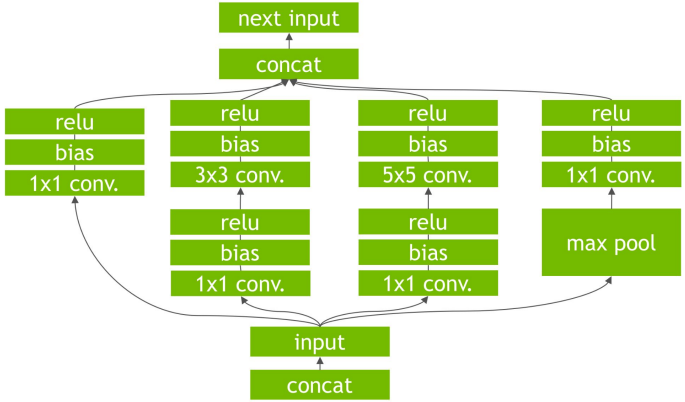
Ключевые различия

	torch.jit.script	torch.jit.trace	torch.compile
Цель	Экспорт моделей с динамической логикой	Экспорт статических моделей	Оптимизация скорости выполнения (во время работы)
Обрабатывает условные конструкции/циклы	Да (статический анализ кода)	Нет (записывает один путь выполнения)	Да (через PyTorch Dynamo)
Сериализуемый	Да (для деплоя)	Да (для деплоя)	Нет
Пример использования	Деплой вне Python (C++, мобильные устройства). Ускорение инференса.	Статические модели. Ускорение инференса.	Ускорение обучения/инференса
Представление	graph-based intermediate representation (IR)	graph-based intermediate representation (IR)	Torch FX

Pytorch 2.0 - Torch Dynamo

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

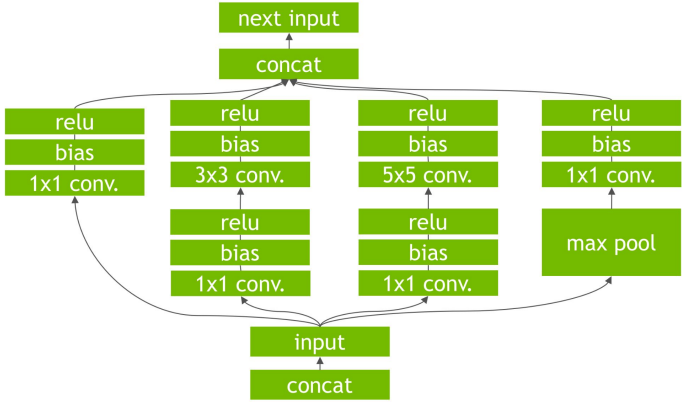
Как происходит оптимизация?



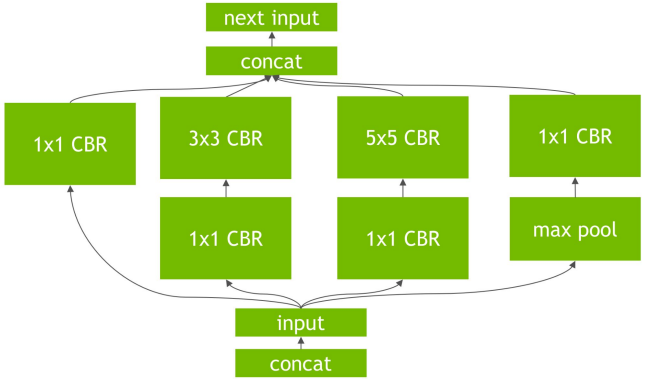
Vertical Fusion

Horizontal Fusion

[Источник]

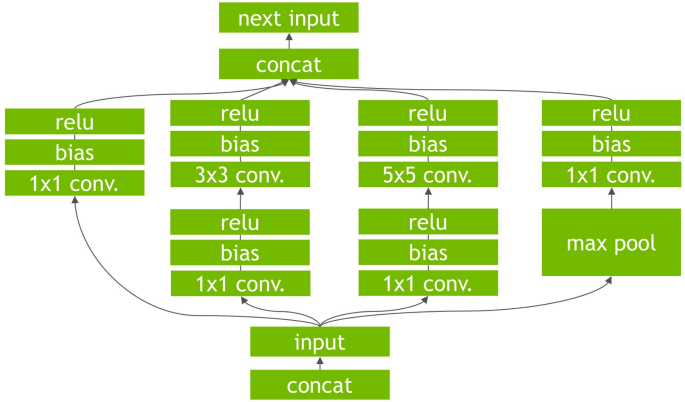


Vertical Fusion

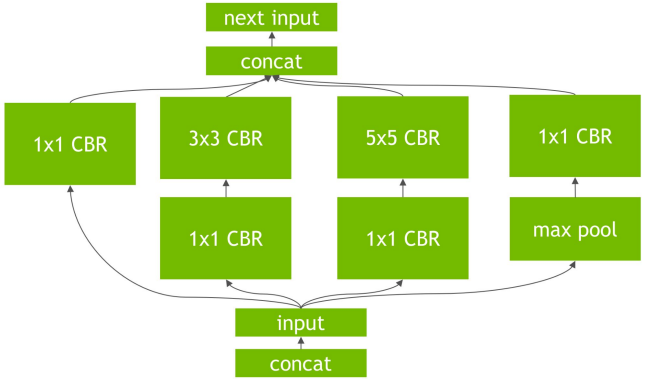


Horizontal Fusion

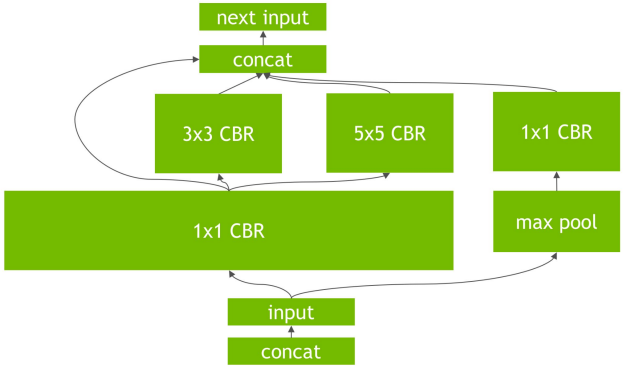
[Источник]



Vertical Fusion



Horizontal Fusion



Источник

- Fusion
- Оптимизация графа
- Оптимизация памяти (Buffer Reuse)
- Алгебраическое переписывание
- Развертывание цикла
- Автоматическое размещение работы
- Выполнение вне очереди
- Многопоточность

```python
from transformers import BertModel, BertTokenizer, BertConfig
import torch

enc = BertTokenizer.from_pretrained("google-bert/bert-base-uncased")

# Tokenizing input text
text = "[CLS] Who was Jim Henson ? [SEP] Jim Henson was a puppeteer [SEP]"
tokenized_text = enc.tokenize(text)

# Masking one of the input tokens
masked_index = 8
tokenized_text[masked_index] = "[MASK]"
indexed_tokens = enc.convert_tokens_to_ids(tokenized_text)
segments_ids = [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]

# Creating a dummy input
tokens_tensor = torch.tensor([indexed_tokens])
segments_tensors = torch.tensor([segments_ids])
dummy_input = [tokens_tensor, segments_tensors]

# Initializing the model with the torchscript flag
# Flag set to True even though it is not necessary as this model does not have an LM Head.
config = BertConfig(
    vocab_size_or_config_json_file=32000,
    hidden_size=768,
    num_hidden_layers=12,
    num_attention_heads=12,
    intermediate_size=3072,
    torchscript=True,
)

# Instantiating the model
model = BertModel(config)

# The model needs to be in evaluation mode
model.eval()

# If you are instantiating the model with *from_pretrained* you can also easily set the TorchScript flag
model = BertModel.from_pretrained("google-bert/bert-base-uncased", torchscript=True)

# Creating the trace
traced_model = torch.jit.trace(model, [tokens_tensor, segments_tensors])
torch.jit.save(traced_model, "traced_bert.pt")
```

	Скорость на CPU в mc	Скорость на GPU в mc
Pytorch	86.23	16.49
TorchScript	81.57	10.54

BERT

```python
from transformers import BertTokenizer, BertModel
import numpy as np
import torch
from time import perf_counter

def timer(f,*args):

    start = perf_counter()
    f(*args)
    return (1000 * (perf_counter() - start))

script_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', torchscript=True)
script_model = BertModel.from_pretrained("bert-base-uncased", torchscript=True)


# Tokenizing input text
text = "[CLS] Who was Jim Henson ? [SEP] Jim Henson was a puppeteer [SEP]"
tokenized_text = script_tokenizer.tokenize(text)

# Masking one of the input tokens
masked_index = 8

tokenized_text[masked_index] = '[MASK]'

indexed_tokens = script_tokenizer.convert_tokens_to_ids(tokenized_text)

segments_ids = [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]

# Creating a dummy input
tokens_tensor = torch.tensor([indexed_tokens])
segments_tensors = torch.tensor([segments_ids])
```

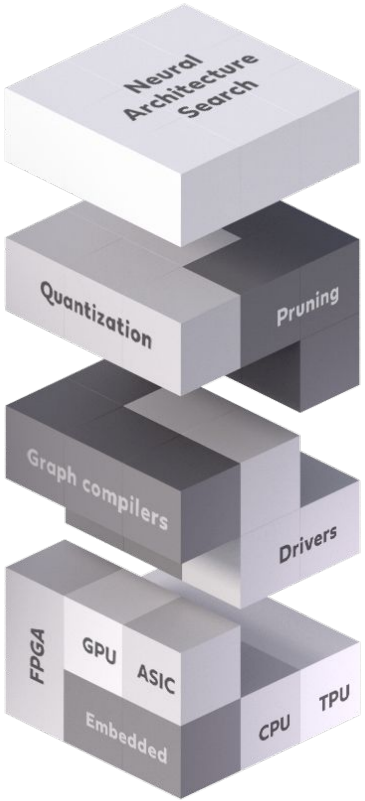
TorchScript-BERT-Example1.1.py hosted with ❤ by **GitHub** **view raw**

```python
# Example 1.1 BERT on CPU

native_model = BertModel.from_pretrained("bert-base-uncased")
np.mean([timer(native_model,tokens_tensor,segments_tensors) for _ in range(100)])

# Example 1.2 BERT on GPU
# Both sample data model need be on the GPU device for the inference to take place
native_gpu = native_model.cuda()
tokens_tensor_gpu = tokens_tensor.cuda()
segments_tensors_gpu = segments_tensors.cuda()
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**

```python
# Example 2.1 torch.jit.trace on CPU

traced_model = torch.jit.trace(script_model, [tokens_tensor, segments_tensors])

np.mean([timer(traced_model,tokens_tensor,segments_tensors) for _ in range(100)])

# Example 2.2 torch.jit.trace on GPU

traced_model_gpu = torch.jit.trace(script_model.cuda(), [tokens_tensor.cuda(), segments_

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**



ЖЕЛЕЗО

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

СОФТ

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

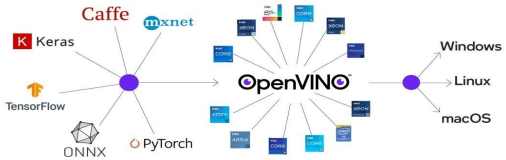
АЛГОРТИМЫ

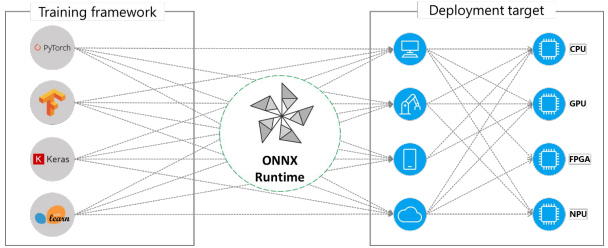
- PRUNING
- QUANTIZATION
- NAS
- etc.

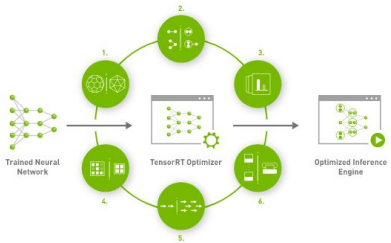
Источник картинки: deci.ai

ONNX предоставляет стандартизированный формат для моделей глубокого обучения. Это позволяет легко использовать их в различных средах и платформах. Часто ONNX применяют 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, разработанный для графических процессоров NVIDIA, максимизирует пропускную способность и эффективность. Он оптимизирует нейронные сети путем объединения слоев, выбора эффективных форматов данных и использования арифметики с пониженной точностью (FP16)
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 ограниченны







Нам нужны промежуточные представления, чтобы импортировать модель в другую среду выполнения:

ONNX, Tensor RT, OvenVino и так далее.

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

Метод	Ускорение инференс
torch.compile (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)

Метод	Ускорение на обучении
torch.compile (Torch Dynamo)	~1.25
TorchScript	~1.25
ONNX Runtime	~ 1.5
TensorRT	Только инференс

SOURCE

	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