Материалы кафедры ММП факультета ВМК МГУ. Введение в глубокое обучение.

Занятие 04. Введение в PyTorch

Занятие провел: Оганов Александр (@welmud)

Материалы составили: Феоктистов Дмитрий (@trandelik), Находнов Максим (nakhodnov17@gmail.com)

Москва, Весенний семестр 2026

Источники:

- Документация PyTorch
- Руководства PyTorch

О чём можно узнать из этого ноутбука:

- Почему PyTorch самая популярная библиотека для глубокого обучения и зачем уметь считать на GPU?
- Базовый синтаксис и понятия библиотеки PyTorch
- Как устроен граф вычислений
- Возможности автоматического дифференцирования и как посчитать Гессиан
- Как построить нейронную сеть

In [ ]:
```
import torch
import numpy as np
```

In [ ]:
```
seed = 42
torch.manual_seed(seed)
np.random.seed(seed)
```

Почему torch?

Какие библиотеки для машинного обучения мы уже знаем? Есть numpy/sklearn, но так ли они удобны для глубокого обучения? Несмотря на наличие модуля sklearn.neural_network, пользоваться им не стоит. Почему?

1. Вы уже могли заметить, что дифференцировать функции руками это больно, поэтому каждый раз для своей нейронки писать backward самостоятельно не хотелось бы.

2. Нейронные сети – это матричные произведения. Очень много матричных произведений. И очень мало всего остального.

Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

Матричные произведения – хорошо параллелизуемая операция, но CPU не очень хорошо масштабируется по FLOPS'ам чисто по физическим причинам (они греются так, что человечество не научилось достаточно быстро отводить тепло от них). Из-за чего так происходит? CPU – умная штука, она должна поддерживать много операций (вспомните курс ассемблера). А для матричного произведения нужна просто числодробилка, то есть довольно "тупая" вещь. Если вещь тупая, то она и масштабироваться будет лучше, а значит выполнять больше параллельных операций, а значит быстрее перемножать матрицы, а значит быстрее работать с нейронными сетями. Такой вещью является GPU/TPU/ASIC

Итак, все эти хотелки содержит в себе `PyTorch`. Ключевые отличия от `numpy/sklearn`:

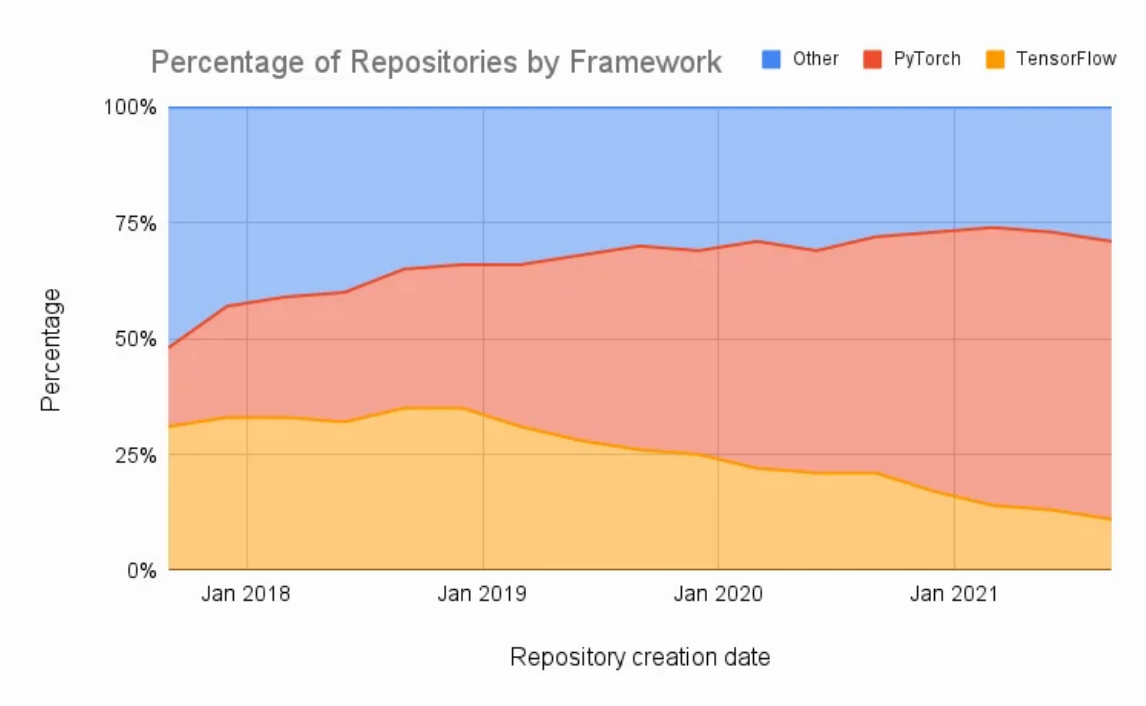
1. Поддерживает вычисления на **CPU/GPU/TPU**
2. **Автоматическое дифференцирование**

Основные методы обучения нейронных сетей являются методами оптимизации первого порядка. Они присутствуют, например, в пакете `scipy.optimize`, но либо считают градиенты численно, либо требуют внешних функций, предоставляющих им градиенты в требуемой точке. Написание таких функций вручную для каждой нейронной сети – задача решаемая, но достаточно бессмысленная, т.к. производные, даже аналитически, достаточно легко может считать и машина.

Концепция библиотеки `PyTorch` – расширить функционал `numpy`, добавив туда возможность автоматического расчёта градиентов произвольных функций и их композиций, стараясь максимально сохранить привычную семантику.

Особенности:

1. **Синтаксически повторяет `numpy` / `scipy`**
2. Большой набор готовых компонент для реализации и обучения DL моделей
3. Для решения домен-спецефичных задач есть свои библиотеки
   - **`torchvision`**
   - `torchaudio`
   - ~~`torchtext`~~
   - `TorchData`
   - `TorchRec`
   - `TorchServe`
4. В конкурентной борьбе победил конкурентов в исследованиях и образовании



Общий обзор

База:

- torch.Tensor
- Tensor Views
- Tensor Attributes
- torch — основные операции над тензорами
  - Tensors
    - Creation Ops
    - Indexing, Slicing, Joining, Mutating Ops
  - Generators
  - Random sampling
    - In-place random sampling
    - Quasi-random sampling
  - Serialization
  - Parallelism
  - Locally disabling gradient computation
  - Math operations
    - Pointwise Ops
    - Reduction Ops
    - Comparison Ops
    - Spectral Ops

        - Other Operations
        - BLAS and LAPACK Operations
    - Utilities
    - Operator Tags
- torch.nn — блоки-классы для DL моделей
    - Containers
    - Convolution Layers
    - Pooling layers
    - Padding Layers
    - Non-linear Activations (weighted sum, nonlinearity)
    - Non-linear Activations (other)
    - Normalization Layers
    - Recurrent Layers
    - Transformer Layers
    - Linear Layers
    - Dropout Layers
    - Sparse Layers
    - Distance Functions
    - Loss Functions
    - Vision Layers
    - Shuffle Layers
    - DataParallel Layers (multi-GPU, distributed)
    - Utilities
    - Quantized Functions
    - Lazy Modules Initialization
- torch.nn.functional — блоки-функции для DL моделей в
- torch.linalg — аналог `numpy.linalg`
- torch.special — аналог `scipy.special`
- torch.nn.init — методы для инициализации весов
- torch.optim — классы-оптимизаторы
- torch.random — случайное семплирование
- torch.autograd — продвинутое дифференцирование
- torch.hub — предобученные модели
- torch.fft — аналог `scipy.fft`
- torch.utils.tensorboard — логгирование в tensorboard

Популярные модули:

- torch.amp — mixed precision training
- Complex Numbers — комплексные числа
- Quantization — квантизация
- torch.distributions — классы для распределений с поддержкой дифференцирования (Reparameterization Trick/REINFORCE)

Вспомодательные модули:

- torch.cuda — параметры и свойства вычисления на GPU
- torch.backends — параметры устройства для вычисления
- torch.**config** — текущая конфигурация

Распределённые вычисления:

- torch.distributed
- torch.distributed.algorithms.join
- torch.distributed.elastic
- torch.distributed.fsdp
- torch.distributed.optim
- DDP Communication Hooks
- Pipeline Parallelism
- Distributed RPC Framework

Для разработки расширений:

- torch.library
- torch.utils.cpp_extension

Разное:

- torch.jit
- torch.futures
- torch.fx
- torch.monitor
- torch.overrides
- torch.package
- torch.profiler
- torch.onnx
- torch.masked
- torch.nested
- torch.sparse
- torch.Storage
- torch.testing
- torch.utils.benchmark
- torch.utils.bottleneck
- torch.utils.checkpoint

- torch.utils.jit
- torch.utils.dlpack
- torch.utils.mobile_optimizer
- torch.utils.model_zoo
- Type Info
- Named Tensors
- Named Tensors operator coverage

Базовые операции над тензорами

Основной объект в PyTorch — тензор (`torch.Tensor`), который является близким аналогом массива из numpy (`np.ndarray`).

In [ ]:
```
(
    torch.zeros([5, 7]),
    torch.ones([5, 7], dtype=torch.int32),
    torch.randn([5, 7])
)
```

Набор операций тоже очень схож:

In [ ]:
```
A, B = torch.randn(3, 5), torch.randn(5, 3)
A, A + B.T, A @ B, torch.sum(A), B.prod()
```

Преобразование из torch в numpy:

In [ ]:
```
w = torch.tensor([1, 2, 3.0])
w_np = w.numpy()
type(w_np), w_np, type(w), w
```

Преобразование из numpy в torch через копирование данных:

In [ ]:
```
v_np = np.arange(10)
v = torch.tensor(v_np)

v_np += 1

type(v_np), v_np, type(v), v
```

Преобразование из numpy в torch с общим буфером:

In [ ]:
```
u_np = np.full([2, 3], -1.0)
u = torch.from_numpy(u_np)

u_np += 10.0

type(u_np), u_np, type(u), u
```

Сравнение скорости:

In [ ]:
```
u_np = np.full([1000, 1000], 317.0)
```

In [ ]:
```
%timeit u = torch.from_numpy(u_np)
```

In [ ]:
```
%timeit u = torch.tensor(u_np)
```

Однако есть небольшие различия:

- `axis` -> `dim` : `np.sum(A, axis=0)` -> `torch.sum(A, dim=0)`
- `.reshape` -> `.view/.reshape`
- `.astype` -> `.to/.type`

Так же, есть возможность использовать inplace операции:

In [ ]:
```
A[1] += A[2]
A
```

In [ ]:
```
A.zero_()
A
```

In [ ]:
```
torch.exp(A)
A.exp_()
```

При работе с тензорами в PyTorch нужно быть аккуратными при работе с различными типами данных

In [ ]:
```
A = torch.zeros(10, dtype=torch.float32)
B = torch.ones(10, dtype=torch.float64)
A + B
```

In [ ]:
```
A @ B
```

In [ ]:
```
A.to(torch.float64) @ B
```

Полезные функции

- Сконкатенировать набор тензоров вдоль заданной размерности `torch.cat`
- Соединить тензоры одинаковой формы вдоль новой размерности `torch.stack`
- Добавить/убрать новую "единичную" размерность в тензор `torch.unsqueeze/torch.squeeze`
- Разбить тензор на заданное число блоков `torch.chunk`
- Переставить между собой две размерности `torch.transpose`
- Переставить местами все размерности `torch.permute`
- Повторить тензор `torch.tile/torch.repeat`
- Найти максимальный элемент (возвращает И положение, И значение) `torch.max`
- Поэлементный максимум между двумя тензорами `torch.maximum`

- "Выпрямить" тензор, объединив все размерности в одну `torch.ravel`
- Объединить только заданные размерности `torch.flatten`
- Вернуть k наибольших элементов `torch.topk`

Первое ключевое отличие – возможность перемещения на GPU

In [ ]:
```
a = torch.randn(3, 5)
a.dtype, a.device
```

Проверим, что GPU доступна:

In [ ]:
```
torch.cuda.is_available()
```

In [ ]:
```
device = (
    torch.device('cuda', 0) # эквивалентно: torch.device('cuda:0') / 'cuda'
    if torch.cuda.is_available()
    else torch.device('cpu')
)
device
```

In [ ]:
```
a.to(device)
```

Операции между тензорами на разных устройствах не возможны:

In [ ]:
```
a.cuda() + a
```

Обратите внимание, что при изменении device (с GPU на CPU и обратно происходит копирование):

In [ ]:
```
p_cpu = torch.rand(10)
p_gpu = p_cpu.to(device)

p_cpu -= 1
p_cpu, p_gpu
```

Сравним скорость работы типичного слоя нейронной сети – matmat + нелинейность на CPU и GPU

In [ ]:
```
def layer(A, B):
    C = A @ B
    C = C ** 2
    return C
```

In [ ]:
```
A = torch.randn([1000, 1000])
B = torch.randn([1000, 1000])
```

In [ ]:
```
%timeit _ = layer(A, B)
```

In [ ]:
```
A_device, B_device = A.to(device), B.to(device)
%timeit _ = layer(A_device, B_device)
```

Видим, что вычисления на GPU значительно быстрее вычислений на CPU, но мы не учли одну вещь – время копирования данных. Давайте сделаем более честный замер

In [ ]:
```
%timeit _ = layer(A.to(device), B.to(device))
```

Второе ключевое отличие — Autograd

Все тензоры содержат атрибут `.grad`, который может хранить градиент по этому тензору.

In [ ]:
```
a.grad
```

По умолчанию, тензоры, создаваемые в PyTorch, не будут требовать, чтобы для них посчитали градиент. Для этого надо добавить дополнительный аргумент `requires_grad=True`, либо вызвав метод `.requires_grad_()`.

In [ ]:
```
x = torch.tensor([1., 2., 3.], requires_grad=True)
y = torch.tensor([1., 2., 3.])
y.requires_grad_()
```

Производя операции с переменными, по которым нужно считать градиенты, мы конструируем граф вычислений:

In [ ]:
```
z = 3 * x**3 - y**2
z.requires_grad
```

В каждой переменной есть информация о том, как именно она была получена при проходе вперёд. Исходя из этой информации у тензоров в графе вычислений хранятся функции, которые должны быть вызваны на обратном проходе для расчёта градиента.

In [ ]:
```
z.grad_fn
```

Считать для каждого тензора якобиан целиком — сильно неоптимально. Вместо этого на всех промежуточных этапах autograd считает только произведения якобиан-вектор. В частности из-за этого, конечный тензор в графе всегда **должен быть скаляром**, что выполняется для всех функций потерь по определению.

Для примера сделаем из тензора `z` скаляр, сложив все его элементы, и посчитаем градиенты с помощью функции .backward()

In [ ]:
```
z.sum().backward()
```

In [ ]:
```
x.grad, y.grad
```

Сравним с посчитанным вручную градиентом:

In [ ]:
```
torch.allclose(x.grad, 9 * x**2)
```

Однако, для оптимизации вычислений, градиенты не вычисляются в явном виде для промежуточных вершин графа

In [ ]:
```
z.grad
```

Если такое всё же нужно, требуется указать это явно с помощью вызова `.retain_grad()`

In [ ]:
```
x = torch.tensor([1., 2., 3.], requires_grad=True)
y = torch.tensor([1., 2., 3.])

z = 3 * x**3 - y**2
z.retain_grad()

z.sum().backward()
z.grad
```

Визуализация графа вычислений

Для примера визуализируем, как бы выглядел граф вычислений для линейной регресси.

Заметим, что пакет визуализации `pytorchviz` предназначен в первую очередь для визуализации нейронных сетей, поэтому нам необходимо будет использовать класс `torch.nn.Parameter`, который является обёрткой над тензором и несколько расширяет возможности аргумента `requires_grad=True`

Графы визуализируются библиотекой `torchviz`, которая использует `graphviz` для визуализации графа вычислений.

In [ ]:
```
# !pip3 install torchviz
```

In [ ]:
```
from torchviz import make_dot
```

In [ ]:
```
X = torch.rand(5, 10)
y = torch.rand(5)

w = torch.nn.Parameter(torch.rand(10))
b = torch.nn.Parameter(torch.rand(1))

y_hat = X @ w + b

loss = torch.mean((y - y_hat)**2)
make_dot(loss, params={'Weight': w, 'Bias': b, 'Loss': loss})
```

Что можно сказать об этом графе вычислений?

- В листьях графа мы не видим тензора `X`, так как нам требуется расчёт градиента только по параметрам модели.
- `MvBackward0` соответствует матрично-векторному умножению (отсюда и первые буквы `Mv`) с матрицей `X`.
- `AddBackward0` соответствует добавлению смещения `b`, остальная часть графа — вычисление MSE.

Зачем тогда уметь писать backward руками?

На текущий момент не для всех функций поддерживается автоматическое дифференцирование. Раньше большой проблемой были комплексные числа, сейчас с их поддержкой стало лучше, однако, далеко не все функции до сих пор дифференцируемы. Просто убедимся, что такая проблема действительно есть, не вдаваясь в подробности, что это за функция

In [ ]:
```
module = torch.nn.Embedding(10, 10, dtype=torch.complex128)
x = torch.tensor([1, 2, 1, 3])
preds = module(x)
loss = torch.linalg.norm(preds)
loss.backward()
```

Рассмотрим интерфейс для реализации backpropagation в PyTorch. Отметим, что любая написанная руками функция встраивается в автоматический граф вычислений.

In [ ]:
```
class MySquare(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and ret
        a Tensor containing the output. ctx is a context object that can be
        to stash information for backward computation. You can cache arbitra
        objects for use in the backward pass using the ctx.save_for_backward
        """
        ctx.save_for_backward(input)
        return input ** 2

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of
        with respect to the output, and we need to compute the gradient of t
        with respect to the input.
```

```
        """
        input, = ctx.saved_tensors
        grad_input = 2 * input * grad_output.clone()
        return grad_input
```

In [ ]:
```
X = torch.rand(5, 10)
y = torch.rand(5)

w = torch.nn.Parameter(torch.rand(10))
b = torch.nn.Parameter(torch.rand(1))

y_hat = X @ w + b
y_hat = MySquare.apply(y_hat)

loss = torch.mean((y - y_hat)**2)
make_dot(loss, params={'Weight': w, 'Bias': b, 'Loss': loss})
```

Backward по отдельным параметрам

Если мы внимательно посмотрим на аргументы функции `backward`, то можем заметить параметр `inputs`. Указав конкретные значения, мы можем ограничить список параметров по которым считается `backward`, то есть ускорить его и использовать меньше памяти. Такой подход может помочь, когда у вас в модели много параметров, но вам необходимо посчитать градиент по маленькому подмножеству, например, вы оптимизируете смещения в линейных слоях

In [ ]:
```
a = torch.tensor(1., requires_grad=True)
a.backward?
```

Так как мы хотим показать ускорение прохода назад по графу вычислений, то для этого необходимо создать действительно большой граф вычислений!

In [ ]:
```
d = 500
A = torch.rand(d, d).to(device).requires_grad_(True)
B = torch.rand(d, d).to(device).requires_grad_(True)
b = torch.tensor(1.).to(device).requires_grad_(True)
q = 0.9

def zero_grad(A, B, b):
    """
    set gradient to zeros
    """
    A.grad = None
    B.grad = None
    b.grad = None

def calc_loss(A, B, q, b, num_iters=2_000):

    C = A + A @ B + B

    for _ in range(num_iters):
        C = q * C + A @ B + b
```

```
    return torch.mean(C)
```

In [ ]:
```
# На всякий случай очистим память
torch.cuda.empty_cache()

loss = calc_loss(A, B, q, b)
loss
```

Теперь самое главное, сделаем `backward`.

In [ ]:
```
%%time
loss.backward()
```

In [ ]:
```
torch.mean(A.grad), torch.mean(B.grad), b.grad
```

In [ ]:
```
zero_grad(A, B, b)
```

Работает действительно долго, но допустим нам не нужно знать градиенты по всем параметрам. Пусть мы сначала хотим оптимизировать по `A`, а уже потом по `B`

In [ ]:
```
# На всякий случай очистим память
torch.cuda.empty_cache()

loss = calc_loss(A, B, q, b)
loss
```

In [ ]:
```
%%time
loss.backward(inputs=[A])
```

In [ ]:
```
torch.mean(A.grad), B.grad, b.grad
```

In [ ]:
```
zero_grad(A, B, b)
```

Стало быстрее примерно в 2 раза, так как мы считаем градиенты только для матрицы `A`. В нашем лоссе есть два равнозначных параметра: `A`, `B`. Допустим наша цель обновить одно число `b`, в нашем примере при подсчете градиента по `b` нет сложных вычислений, проверим это

In [ ]:
```
loss = calc_loss(A, B, q, b)
loss
```

In [ ]:
```
%%time
loss.backward(inputs=[b])
```

In [ ]:
```
A.grad, B.grad, b.grad
```

Получилось еще быстрее! Если же мы укажем тензоры, которые не учавствовали в вычислении градиента, то обратный проход будет работать моментально

In [ ]:
```
loss = calc_loss(A, B, q, b)
```

In [ ]:
```
c = torch.rand(d, d).to(device).requires_grad_(True)
```

In [ ]:
```
%%time
loss.backward(inputs=[c])
```

Аккумулирование градиентов

Если не предпринимать никаких дополнительных действий, то множественный вызов `.backward()` будет **не перезаписывать градиенты** тензоров, а **складывать их** с уже существующим значением (сам граф вычислений каждый раз разрушается).

In [ ]:
```
x = torch.randn(3, 3, requires_grad=True)
y = torch.sum(x * x)
y.backward()

x.grad
```

In [ ]:
```
z = torch.sum(2 * x)
z.backward()

# Заметьте, что ко всем значениям прибавилось 2
x.grad
```

Чтобы считать градиент с нуля, достаточно удалить тензор.

In [ ]:
```
x.grad = None
```

А зачем такое может быть вообще надо? Рассмотрим задачу линейной регрессии. С большой матрицей, очень большой матрицей

In [ ]:
```
size = 100_000
dim = 4_000
X_master = torch.rand(size, dim)
y_master = torch.rand(size).to(device)

w = torch.nn.Parameter(torch.rand(dim)).to(device)
b = torch.nn.Parameter(torch.rand(1)).to(device)
```

In [ ]:
```
# На всякий случай очистим память
torch.cuda.empty_cache()

y_hat = X_master.to(device) @ w + b

loss = torch.mean((y_master - y_hat)**2)
loss.backward()
```

In [ ]:
```
!nvidia-smi
```

Из этого вывода мы можем понять: объем памяти, сколько памяти занято, нагрузку на

GPU, какие процессы используют GPU, версию драйвера (CUDA Version), что полезно при устновки PyTorch.

In [ ]:
```
# На всякий случай очистим память
torch.cuda.empty_cache()

X = X_master[:X_master.shape[0] // 2]
y = y_master[:y_master.shape[0] // 2]

y_hat =  X.to(device) @ w + b

loss = torch.mean((y - y_hat)**2)
loss.backward()
```

In [ ]:
```
# На всякий случай очистим память
torch.cuda.empty_cache()

X = X_master[:X_master.shape[0] // 2]
y = y_master[:y_master.shape[0] // 2:]

y_hat =  X.to(device) @ w + b

loss = torch.mean((y - y_hat)**2)
loss.backward()
```

In [ ]:
```
!nvidia-smi
```

Такое поведение может быть, например, нужно, чтобы посчитать градиент по батчу данных, который не влезает в память компьютера целиком, так как градиент модели аддитивен по входным данным. Также подобный подход используется в рекуррентных нейронных сетях, где к одному и тому же тензору весов нейросети происходит несколько обращений во время прямого прохода.

Inplace операции

По умолчанию, вызов `y = 2 * x` создаст новый тензор, в который скопирует значения `x`, умноженные на `2`. И есть большое желание провести данную операцию на месте, то есть без аллокации памяти. В numpy это не имело бы никаких дополнительных последствий, но в PyTorch нам надо помнить о графе вычислений, который должен быть без петель, а также может использовать тензоры, рассчитанные при прямом вычислении. В некоторых случаях библиотека может выполнить код и не ругнуться, но описание ситуаций, когда такое сработает, а когда нет, очень сложно, и потому сами разработчики не рекомендуют использовать in-place операции там, где необходим расчёт градиента.

In-place операции всегда имеют символ `_` на конце.

In [ ]:
```
x = torch.randn(3, 3, requires_grad=True)
y = 2 * x
```

```
z = y ** 2
# inplace operation!
y.exp_()
z.sum().backward()
```

Другой распространённый в numpy сценарий – маскированное изменение значений тензора. Это тоже является in-place операцией. В PyTorch для этого лучше использовать функцию `torch.where`:

In [ ]:
```
x = torch.rand(3, 3, requires_grad=True)
x
```

In [ ]:
```
x[x > 0.5] = 0
```

In [ ]:
```
torch.where(x > 0.5, 0, x)
```

Копирование тензоров

В numpy существует интуитивно понятная функция `.copy()`, но в PyTorch функции с таким названием нет! Это связано с тем, что тензоры в PyTorch привязаны к графу вычислений, который надо также учитывать при копировании.

`.clone()` копирует тензор и сохраняет его привязку к текущему дереву вычислений:

In [ ]:
```
x = torch.rand(3, requires_grad=True)
y = x.clone()
y.requires_grad
```

`.detach()` исходя из своего названия, копирует лишь значения элементов тензора, отвязывая его от текущего графа вычислений:

In [ ]:
```
y = x.detach()
y.requires_grad
```

При этом `.detach()` возвращает тензор на той же памяти, то есть изменение `y` приводит к изменению `x`.

Очень часто при копировании градиентов недостаточно сделать только `.clone`, так как мы не хотим сохранять связь с графом вычислений. Например, если мы применим `.clone` к выходу нейросети, то добавим новое ребро в граф вычислений, что повличет накладные расходы. Кроме того, можно получить очень неожиданный эффекты если забывать про `.detach`

In [ ]:
```
# На всякий случай очистим память
torch.cuda.empty_cache()
```

In [ ]:
```
d = 4
x = torch.nn.Parameter(torch.rand(d, d, device=device))
```

```
x.grad
```

In [ ]:
```
z = x.clone() # Считаем, что это копия тензора, которая не связана с тензорс

y = z * 10
loss = y.sum()

loss.backward()
```

In [ ]:
```
x.grad # Все же связаны...
```

In [ ]:
```
z = x.detach().clone()

y = z * 10
loss = y.sum()

loss.backward()
```

Продемонстрируем на графе вычислений эти как работает `.clone`, а как `.detach` операции:

In [ ]:
```
x = torch.nn.Parameter(torch.rand(3))
y = torch.nn.Parameter(torch.rand(3))

z = torch.sum(x + x.clone() + y.detach())

make_dot(z, params={'X': x, 'Y': y})
```

Что можно понять из данного графа?

- Видна операция `CloneBackward0`, которая клонирует тензор `x`. Благодаря ей в сумме участвует как исходный тензор, так и его экспонированная версия.
- Во второй сумме мы не видим параметра `y`, потому что он входит в граф вычислений только через `.detach()`, что убирает проход градиентов.

Продвинутое дифференцирование

В модуле `autorgrad` есть набор функций для более сложных операций по подсчёту градиентов:

In [ ]:
```
X = torch.randn(5, 10)
w = torch.randn(10, requires_grad=True)
```

Например, подсчёт градиентов скалярной функции:

In [ ]:
```
torch.autograd.grad(torch.mean(X @ w), [w]), X.mean(dim=0)
```

Вычисление произведения якобиана на вектор vJ, где
$J = \nabla_w L(w) \in \mathbb{R}^{n \times d}, w \in \mathbb{R}^d, L(w) \in \mathbb{R}^n$

In [ ]:
```
v = torch.randn(5)
torch.autograd.grad(X @ w, [w], grad_outputs=v), v @ X
```

Подсчёт градиентов по элементам в батче:

In [ ]:
```
# Экспериментальная фича

torch.autograd.grad(X @ w, [w], grad_outputs=(torch.eye(5), ), is_grads_batc
```

In [ ]:
```
# Медленная альтернатива:

y = X @ w
torch.stack(([torch.autograd.grad(y[idx], w, retain_graph=True)[0] for idx i
```

Обратите внимание, что в предыдущем примере проход через один и тот же граф вычислений делается несколько раз. По умолчанию, после первого прохода граф уничтожается. Чтобы этого избежать используйте `retain_graph=True`

Подсчёт вторых производных:

In [ ]:
```
S = w @ X.T @ X @ w

# Подсчёт в два шага. Для создания графа вычислений вторых производных испол

[grad] = torch.autograd.grad(S, w, create_graph=True)
print(torch.linalg.norm(grad - 2 * X.T @ X @ w))

[hessian] = torch.autograd.grad(grad, w, grad_outputs=(torch.eye(w.shape[0])
print(torch.linalg.norm(hessian - 2 * X.T @ X))

# Подсчёт в функциональном виде:

hessian_func = torch.autograd.functional.hessian(
    lambda w: w @ X.T @ X @ w, w
)
print(torch.linalg.norm(hessian - hessian_func))
```

Менеджеры контекста

Поведением градиента сразу группы тензоров можно управлять с помощью специальных функций, которые вызываются через стандартную семантику питона `with foo():`, где `foo` — одна из трёх функций ниже:

Default Mode

Стандартный режим работы PyTorch, в котором управление градиентами происходит через requires_grad. Явно его нужно вызывать только внутри других контекстных менеджеров, чтобы временно снова активировать расчёт градиентов (что случается крайне редко) вызовом `torch.enable_grad()`.

No grad mode

Данный режим используется когда блока кода нет необходимости вычислять градиенты, что занимает как вычислительные ресурсы, так и дополнительную память. Реализуется вызовом `torch.no_grad()`.

Inference mode

Аналогично No grad mode отключает расчёт градиентов, но кроме того проводит дополнительные оптимизации, что делает вычисления внутри блока кода ещё быстрее. Однако, тензоры, созданные в таком блоке будет невозможно использовать совместно с тензорами, для которых расчёт градиента необходим. Реализуется вызовом `torch.inference_mode()`.

In [ ]:
```
x = torch.rand(3, requires_grad=True)

with torch.no_grad():
    y = x + x

y.requires_grad
```

In [ ]:
```
@torch.no_grad()
def foo(x):
    return x + 2

y = foo(x)
y.requires_grad
```

.item()

Часто при подсчёте метрик возникают тензоры из одного элемента/скаляры, которые могут находится на GPU или быть частью графа вычислений.

In [ ]:
```
torch.cuda.empty_cache()

w = torch.randn(10000, requires_grad=True).to(device)

losses = []
for i in range(50):
    x = torch.randn((10000, 10000)).to(device)
    loss = torch.linalg.norm(x @ w)
    losses.append(loss)
    print(f"Сейчас занято {torch.cuda.memory_allocated() // 1024 // 1024} ME
```

Сохранение таких тензоров в массив приведёт к утечке памяти. Чтобы этого избежать существует возможность трансформирования таких тензоров в Python скаляр:

In [ ]:
```
torch.cuda.empty_cache()

w = torch.randn(10000, requires_grad=True).to(device)

losses = []
```

```
for i in range(50):
    x = torch.randn((10000, 10000)).to(device)
    loss = torch.linalg.norm(x @ w)
    losses.append(loss.item())
    print(f"Сейчас занято {torch.cuda.memory_allocated() // 1024 // 1024} ME
```

Построение нейронных сетей

Для обучения нейронной сети необходимо определить ее структуру (архитектуру). Чаще всего модели собираются из заранее реализованных блоков, как конструктор. В этой секции мы посмотрим на основные блоки и как из них собирать модели.

torch.nn.Module

Для того, чтобы реализовать свой блок, нужно, чтобы он был отнаследован от базового класса для всех нейронных сетей `torch.nn.Module`.

Затем, для удобства и в соответствии с общепринятой практикой, модуль должен реализовывать функцию `forward`. В этой функции обычно реализуют основную логику вычислений в модуле, так называемый *прямой проход* через слой. PyTorch использует функцию `forward` для переопределения магического метода класса `__call__`, что позволяет использовать следующий синтаксический сахар для вычисления прямого прохода:

```
# Два эквивалентных способа делать прямой проход через блок:
out = module(x)
out = module.forward(x)
```

Нужно отметить, что блоки являются лишь удобной обёрткой для вычислений и создания графа автоматического дифференцирования.

Рассмотрим простейший пример модуля, который реализует **Тождественное преобразование**:

In [ ]:
```
class Identity(torch.nn.Module):
    def __init__(self):
        '''
        Конструктор блока. Здесь обычно создают обучаемые параметры и сохран
            определяющие глобальное состояние слоя, а так же гиперпараметры
        Блок, может содержать в себе подблоки, которые также были отнаследов
        '''
        # Необходимо вызвать конструктор базового класса для корректной рабо
        super().__init__()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        '''
        Функция, реализующая прямой проход через слой. В процессе вычислений
            сквозь который выполняется автоматическое дифференцирование.
        '''
        return x
```

```
    def __repr__(self):
        '''
        Хорошей практикой является переопределение строкового представления
        Обычно, помимо названия класса также выводят гиперпараметры данного
        '''
        return 'Identity()'
```

In [ ]:
```
identity_layer = Identity()
x = torch.randn([3, 5])

# Обратите внимание, метод __call__ был автоматически переопределён через вы
print(x, identity_layer(x), sep='\n')
print(identity_layer, torch.equal(x, identity_layer(x)))
```

Основным преимуществом базового класса `torch.nn.Module` является наличие интерфейса работы с параметрами (обучаемые веса модели), буферами (необучаемые тензоры в модели) и подмодулями (то есть другие модули-наследники `torch.nn.Module`, которые в итоге создают иерархическую структуру модели).

Если параметры/буферы/подмодули зарегистрированы в модуле, то базовый класс позволяет удобно работать с этими сущностями. Рассмотрим основные методы:

- `parameters` — возвращает итератор по всем параметрам, зарегистрированным в данном модуле и всех его подмодулях
- `named_parameters` — возвращает итератор по всем параметрам и их именам, зарегистрированным в данном модуле и всех его подмодулях
- `buffers` / `named_buffers` — аналогичные функции для получения доступа к списку всех буферов
- `requires_grad_(requires_grad=True)` — in-place метод, включающий или выключающий подсчёт градиентов для параметров
- `to(device, dtype)` — изменение типа параметров и устройства на котором они располагаются
- `train` / `eval` — рекурсивное переключение режимов работы модулей. Обычно, перед обучением сети необходимо вызвать метод `train`, а перед тестированием (инференсом) — перевести в режим `eval`. Данные методы необходимы для корректной реализации таких методов регуляризации как BatchNorm и Dropout

Подробное описание работы этих и других доступных методов можно найти в документации.

Теперь рассмотрим процесс регистрации параметров/буферов/подмодулей. Существует два основных варианта регистрации:

1. **Неявная регистрация**, которая происходит в момент создания атрибута класса и присваивания ему класса-обёртки для параметров `torch.nn.Parameter` (для регистрации параметров) или наследника `torch.nn.Module` (для регистрации подмодулей). Обратите внимание, что **присваивание тензора не приводит к**

регистрации параметра. Также, стоит отметить, что нет неявного способа регистрации буферов
2. **Явная регистрация** с использованием методов `register_parameter, register_buffer, register_module`.

По умолчанию при написании кода стоит отдавать предпочтение **неявной регистрации**.

Отдельно выделим частую ситуацию, в которой требуется зарегистрировать переменное (или просто достаточно большое число) параметров/подмодулей. Действительно, в современных моделях может быть десятки и сотни слоёв и тысячи тензоров-параметров, которые логично хранить в модели не в виде отдельных атрибутов, а в виде списка или словаря. Однако, сохранив параметры/подмодули в такую структуру данных, **неявная регистрация автоматически не сработает**. Чтобы обойти этот недостаток в **PyTorch** реализован набор контейнеров, которые при добавлении в них параметров будут производить их регистрацию в текущем блоке. Так, например, доступен контейнер для списка параметров `torch.nn.ParameterList` и контейнер для словаря параметров `torch.nn.ParameterDict`. Аналогичные классы-контейнеры доступны и для модулей: `torch.nn.ModuleList`, `torch.nn.ModuleDict`. При реализации слоёв с фиксированным числом параметров (например, `Linear`) такие классы обычно не нужны.

Сборка нейронной сети

Для создания моделей из набора слоёв в **PyTorch** обычно используется следующий иерархический подход. Внутри каждого блока, отнаследованного от `torch.nn.Module`, могут содержаться другие слои, так же отнаследованные от `torch.nn.Module`. Как описывалось ранее, подмодули/параметры/буферы имеют два способа регистрации — неявный (через присваивание атрибута класса) и явный (в случае модулей, через метод `register_module`).

Для решения проблемы регистрации списков/словарей модулей **PyTorch** реализует контейнеры с поддержкой автоматической регистрации. Как и для параметров, существует несколько таких контейнеров:

- `torch.nn.ModuleList` — хранит список модулей,
- `torch.nn.ModuleDict` — хранит словарь модулей,
- `torch.nn.Sequential` — особый контейнер, который хранит список модулей и позволяет последовательно применить модули из списка к объекту, поданному ему на вход.

Вопрос: Как примерно оценить вычислительную сложность модели?

Ответ:

Для начала можем посмотреть на количество параметров, что в первом преближении поможет нам оценить сложность. Почему такая характеристика часто далека от правды?

In [ ]:
```python
def print_params_count(model):

    total_params = sum(p.numel() for p in model.parameters())
    total_params_grad = sum(p.numel() for p in model.parameters() if p.requi

    model_name = model.__class__.__name__
    print(f"Информация о числе параметров модели: {model_name}")
    print(f"Всего параметров: \t\t {total_params}")
    print(f"Всего обучаемых параметров: \t {total_params_grad}")
    print()
```

Моя первая нейронная сеть

In [ ]:
```python
from torch import nn
```

Для начала создадим свою первую нейросеть для классификации, используя уже готовые блоки из PyTorch.

In [ ]:
```python
class MyFirstNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes):
        # Необходимо вызвать конструктор базового класса для корректной рабо
        super().__init__()


        # Создание линейного слоя
        self.fc_in = nn.Linear(
            in_features=input_dim,
            out_features=hidden_dim,
            bias=True
        )

        # torch.nn.ReLU по умолчанию stateless модуль, реализующий stateless
        #   Поэтому его можно безопасно переиспользовать в нескольких местах
        #   Однако это верно не для всех модулей и слоёв, например,
        #   после квантизации сети ReLU становится statefull
        self.relu1 = nn.ReLU()

        # Создание еще одного линейного слоя
        self.fc_hidden = nn.Linear(hidden_dim, hidden_dim, bias=True)
        self.relu2 = nn.ReLU()

        # Создание линейного слоя для получения логитов
        self.fc_out = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        x = self.fc_in(x)
        x = self.relu1(x)
```

```
        x = self.fc_hidden(x)
        x = self.relu2(x)

        return self.fc_out(x)
```

Если вывести объект класса `torch.nn.Module` мы можем увидеть какие блоки в нем стоят и их параметры (если для них определен метод `__repr__`).

In [ ]:
```
input_dim   = 64
hidden_dim  = 32
num_classes = 2  # бинарная классификация

net = MyFirstNetwork(
    input_dim=input_dim,
    hidden_dim=hidden_dim,
    num_classes=num_classes
)
print_params_count(net)
net
```

In [ ]:
```
batch = torch.ones((4, input_dim))
batch, net(batch)
```

Вопрос: Действительно ли так пишут модели? Какие у вас предположения?

Ответ:

На практике так не пишут, так как сложно масштабировать архитектуру модели. Кроме того, при такой записи невозможно понять какие блоки за что отвечают. Мы часто будем пробовать похожие архитектуры но с разной глубиной, шириной гипер параметрами. Разумно, чтобы эти параметры задавались через конструктор модели, так будет проще ставить эксперименты.

Моя вторая нейронная сеть

В примере выше очень трудно понять к чему относится `relu1`, к чему относится `relu2`. Желательно объединять модули в логически блоки, например, как в примере ниже.

In [ ]:
```
class MySecondNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes):
        super().__init__()

        self.fc_in = nn.Sequential(
                nn.Linear(in_features=input_dim, out_features=hidden_dim, bi
                nn.ReLU()
            )

        self.fc_hidden = nn.Sequential(
                nn.Linear(hidden_dim, hidden_dim, bias=True),
```

```
                nn.ReLU()
            )

        self.fc_out = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):

        x = self.fc_in(x)
        x = self.fc_hidden(x)

        return self.fc_out(x)
```

In [ ]:
```
net = MySecondNetwork(
    input_dim=input_dim,
    hidden_dim=hidden_dim,
    num_classes=num_classes
)
print_params_count(net)
net
```

Так как `nn.Sequential` принимает на вход список, то мы можем достаточно легко масштабировать нейросеть и задать в ней произвольное число слоев.

In [ ]:
```
class DeepNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes, num_layers):
        super().__init__()

        self.fc_in = nn.Sequential(
                nn.Linear(in_features=input_dim, out_features=hidden_dim, bi
                nn.ReLU()
            )

        self.layers = nn.Sequential(
            *[
                nn.Sequential(nn.Linear(hidden_dim, hidden_dim), nn.ReLU())
                for i in range(num_layers)
             ],
        )

        self.fc_out = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):

        x = self.fc_in(x)
        x = self.layers(x)

        return self.fc_out(x)
```

In [ ]:
```
num_layers = 2

net = DeepNetwork(
    input_dim=input_dim,
    hidden_dim=hidden_dim,
    num_classes=num_classes,
    num_layers=num_layers
```

```
)
print_params_count(net)
net
```

In [ ]:
```
num_layers = 4

net = DeepNetwork(
    input_dim=input_dim,
    hidden_dim=hidden_dim,
    num_classes=num_classes,
    num_layers=num_layers
)
print_params_count(net)

net
```

Мы наконец-то научились создовать нейросети! Все очень просто и удобно, теперь мы готовы решать действительно трудные задачи! Или нет?...

Ниже пример стандартной архитектуры для классификации изображений, но что там внутри мы поговорим на следующей неделе!

In [ ]:
```
from torchvision.models import resnet101
```

In [ ]:
```
net = resnet101()
print_params_count(net)
```

In [ ]:
```
net
```

Заключение

Хорошей практикой для закрепления этого занятия будет установить PyTorch локально и проверить доступность вычисления на GPU командой `torch.cuda.is_available()`, которая должна вернуть `True`.

Важно отметить, что кроме PyTorch также используется jax. Jax позволяет обучать модели быстрее благодаря оптимизации и компиляции, но требует более строго кода и большого количества вычислительных ресурсов.