

курс «Глубокое обучение»



PyTorch

Александр Дьяконов

05 сентября 2022 года

План

Часть 1 – тензоры

Часть 2 – предобработка данных, нейросети

Pytorch

**открытый фреймворк для построения и использования динамических графов
вычислений и глубокого обучения**

Есть альтернативы: TensorFlow, JAX, Caffe

**Изначально разрабатывался Facebook's AI Research Lab (FAIR).
Вместе с функционалом Python удобен для экспериментов и разработки
(минимум кода при максимуме возможностей)**

Наиболее важные для DL возможности

**автоматическое дифференцирование,
вычисления на базе многомерных матриц (тензоров) - очень похож на numpy,
поддержка динамических вычислительных графов (создаются при работе),
поддержка вычислений на GPU,
есть полезные модули (например, torchvision).**

Про установку см. на официальном сайте <https://pytorch.org/get-started/locally/>

Проверка версии питона, пайторча и доступности видеокарты

```
import torch # заметьте, что не import pytorch
from platform import python_version
print(python_version()) # 3.8.8
print(torch.__version__) # 1.9.0
print (torch.cuda.is_available()) # True
```

3.8.8

1.9.0

True

COLAB - прикручиваем свой гугл диск

```
from google.colab import drive

drive.mount("/content/gdrive", force_remount=True)

data_path = "/content/gdrive/My Drive/Colab Notebooks/name/"
train_ann_path = data_path + 'train.csv'

train_df = pd.read_csv(train_ann_path)
print(train_df.head())

# команды для bash пишутся с !
!ls /content/gdrive/My\ Drive/Colab\ Notebooks/name/
```

Часть 1 – Тензоры (`torch.Tensor`)

– аналоги многомерных массивов пакета numpy, только могут располагаться на GPU (или поддерживать вычисления на нескольких CPU), могут быть элементами вычислительного графа и поддерживать автоматическое дифференцирование (об этом позже).

Это фундаментальная структура данных в Pytorch (с помощью неё будут храниться и обрабатываться объекты: тексты, сигналы изображения и батчи - наборы объектов).

Могут в многомерном матричном виде хранить данные определённого типа.

Тензоры: создание (из списка)

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])

print(x, '\n',
      x.shape, '\n', # размер тензора
      x.dtype, '\n', # тип
      x.device, '\n', # где лежит
      x.type(), '\n', # тип
      x.dim(), '\n', # размерность
      x.size(), '\n', # размер; .shape и .size() одно и то же
      x.numel()) # тип тензора
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
torch.Size([2, 3])
torch.int64
cpu
torch.LongTensor
2
torch.Size([2, 3])
6
```


Тензоры: создание

```
x = torch.cuda.FloatTensor(2, 3) # те тензоры были на CPU
print (x)
```

Тензоры: приведение типов

```
# приведение типов

x = torch.IntTensor([1, 2]).float()
print (x)

x = torch.IntTensor([1, 2]).to(torch.float64)
print (x)

x = torch.IntTensor([1, 2]) + 0.0 #
print (x)
```

Тензоры: создание

Во всех функциях создания тензоров (ниже) есть параметры `dtype` - тип элементов тензора и `device` - где размещать тензор.

```
x = torch.empty(3, 5) # пустая матрица (тензор)
tensor([[8.9082e-39, 5.9694e-39, 8.9082e-39, 1.0194e-38, 9.1837e-39],
        [4.6837e-39, 9.9184e-39, 9.0000e-39, 1.0561e-38, 1.0653e-38],
        [4.1327e-39, 8.9082e-39, 9.8265e-39, 9.4592e-39, 1.0561e-38]])
```

```
x = torch.ones(3, 5) # матрица из 1
tensor([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]])
```

```
x = torch.full((3, 5), 3.14, dtype=torch.float) # матрица из 3.14
```

Тензоры: создание

```
# единичная матрица (с единицами на главной диагонали)
x = torch.eye(3, 5)

# случайная матрица с элементами равномерно распределёнными на [0, 1]
torch.manual_seed(123)
x = torch.rand(3, 5)

# случайная матрица с нормально распределёнными элементами
torch.manual_seed(123)
x = torch.randn(3, 5)

# случайная матрица с числами от 2 до 4 (не включая)
torch.manual_seed(123)
x = torch.randint(2, 4, (3, 5))
```

Тензоры: создание «равномерные» массивы

```
x = torch.arange(0, 10, 2) # аналог np.arange
tensor([0, 2, 4, 6, 8])
```

```
x = torch.linspace(0, 10, 3) # аналог np.linspace
tensor([ 0.,  5., 10.])
```

```
x = torch.logspace(0, 1, 3) # аналог np.logspace
tensor([ 1.0000,  3.1623, 10.0000])
```

сделать тензоры «по образцу» (использовать такой же тип и размеры)

```
torch.empty_like(x), torch.zeros_like(x), torch.ones_like(x)
(tensor([ 0.,  5., 10.]), tensor([0., 0., 0.]), tensor([1., 1., 1.]))
```

Тензоры: индексация

Индексация аналогичная принятой в питоне, в частности в numpy: [start:end:step]

```
x = torch.randint(0, 10, (2, 5))
```

```
x[0], x[0, :], x[[0], :], x[:1, :]
```

```
(tensor([6, 5, 3, 9, 4]),  
 tensor([6, 5, 3, 9, 4]),  
 tensor([[6, 5, 3, 9, 4]]),  
 tensor([[6, 5, 3, 9, 4]]))
```

```
x[:, [1]], x[:, 1], x[:, -4]
```

```
(tensor([[5],  
         [1]]),  
 tensor([5, 1]),  
 tensor([5, 1]))
```

Тензоры: индексация

```
print (x[0, 0]) # это тензор 1x1  
print (x[0, 0].item()) # а это уже отдельный элемент  
tensor(6)  
6
```

При сравнении возникают «логические тензоры»

Тензоры: как скопировать тензор

При использовании `clone()` копия остаётся в графе вычислений.

При использовании `copy_()` такого не происходит.

`detach` - убирает информацию связанную с вычислительным графом из объекта.

```
# копирование

a = torch.tensor([[1, 2], [3, 4]])

b = a.new_tensor(a)
b = a.clone().detach()
b = torch.empty_like(a).copy_(a)
b = torch.tensor(a)
b = a.detach().clone() # лучше так
```

Тензоры: операции

транспонируем – не происходит копирования, используется та же память

```
xt = x.t()  
x[0, 0] = 30  
print ('t() \n', x, '\n', xt)
```

```
t()  
tensor([[30,  2],  
        [ 3,  4]])  
tensor([[30,  3],  
        [ 2,  4]])
```


Тензоры: операции

```
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[2, 2], [2, 2]])

z = torch.stack((x, y)) # состыковка тензоров (по умолчанию dim=0)
print (z, '\n', z.shape)
tensor([[[1, 2],
          [3, 4]],
        [[2, 2],
          [2, 2]]])
torch.Size([2, 2, 2])

x, y = z.unbind(dim=0) # разстыковка тензоров
print (x, '\n', y)
tensor([[1, 2],
        [3, 4]])
tensor([[2, 2],
        [2, 2]])
```

Тензоры: конкатенация

```
# конкатенация по 0 и 1 размерностям
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[2, 2], [2, 2]])
torch.cat([x, y], axis=1), torch.cat([x, y], axis=0)
(tensor([[1, 2, 2, 2],
         [3, 4, 2, 2]]),
 tensor([[1, 2],
         [3, 4],
         [2, 2],
         [2, 2]]))
```

cat в отличие от **stack** использует существующие размерности

Тензоры: дополнительные размерности

```
# создание фиктивной размерности - в какую позицию вставлять фиктивную
x = torch.tensor([[1, 2], [3, 4]])
print (x.unsqueeze(dim=0).shape, x.unsqueeze(dim=1).shape,
        x.unsqueeze(dim=2).shape)
print (x[None, :, :].shape, x[:, None, :].shape, x[:, :, None].shape)
(torch.Size([1, 2, 2]), torch.Size([2, 1, 2]), torch.Size([2, 2, 1]))
(torch.Size([1, 2, 2]), torch.Size([2, 1, 2]), torch.Size([2, 2, 1]))
```

```
# удаляем единичные размеры
torch.empty(3, 1, 2, 1).squeeze().shape
torch.Size([3, 2])
```

«Смежность» тензора

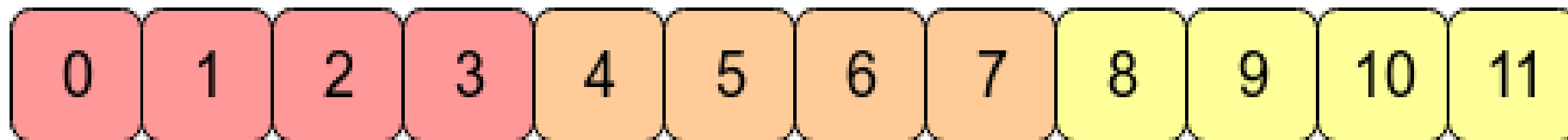
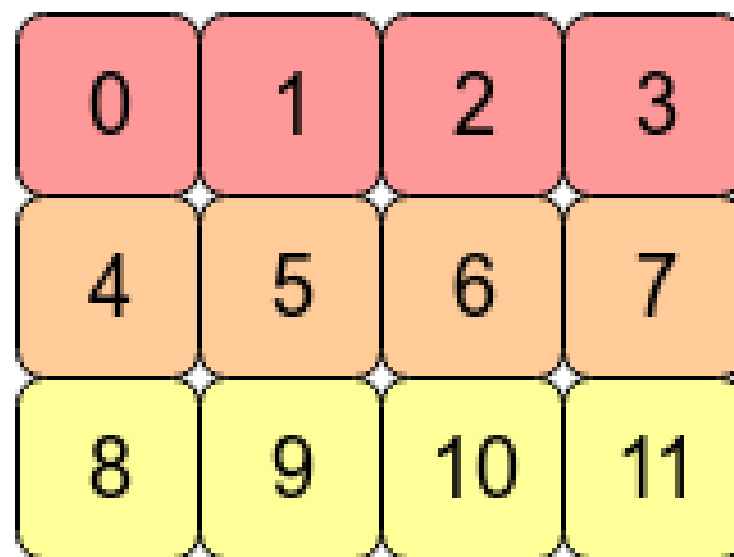
Разберёмся, как хранятся в памяти двумерные тензоры

```
import torch  
H = torch.rand((10000, 10000))
```

```
%%time  
s = torch.sum(H, axis=1) # суммы строк быстрее  
CPU times: user 36.8 ms, sys: 1.87 ms, total: 38.6 ms  
Wall time: 85.7 ms
```

```
%%time  
s = torch.sum(H, axis=0) # суммы столбцов медленнее  
CPU times: user 94.6 ms, sys: 0 ns, total: 94.6 ms  
Wall time: 106 ms
```

«Смежность» тензора



<https://stackoverflow.com/questions/26998223/what-is-the-difference-between-contiguous-and-non-contiguous-arrays/26999092#26999092>

«Смежность» тензора

Они по умолчанию лежат «по стокам» – такое размещение и называется «смежным». Однако, например, в случае транспонирования тензора, данные не копируются, Pytorch просто запоминает, что теперь размещение «несмежное».

```
x = torch.randn(3, 2)
print (x.is_contiguous()) # contiguous
y = torch.transpose(x, 0, 1)
print (x.is_contiguous()) # НЕ contiguous
x[0, 0] = 42 # но они делят память
print(y[0,0]) # 42
```

True

True

tensor(42.)

Тензоры: размеры

Можно менять «представление» тензора с помощью `view`, фактически это изменение размеров, но реально данные не перемещаются, `pytorch` просто запоминает, что тензор, заданный элементами, лежащими в определённой области памяти, имеет другой размер.

View работает только на `contiguous tensors`
(которые «правильно» последовательно лежат в памяти)

При использовании `view` может выдаваться сообщение об ошибке – если нельзя использовать эту область памяти (можно тогда сделать предварительно `.contiguous()`)

Reshape работает всегда
(старается выдать `view`, если не получается делает копию данных)
При `reshape()` тензор может копироваться!

Тензоры: view & reshape

```
x = torch.arange(4*10*2).view(4, 10, 2)
y = x.permute(2, 0, 1)
```

View

```
print('смежность', x.is_contiguous())
print('вытягиваем', x.view(-1))
```

смежность True

вытягиваем tensor([0, 1, ...

Reshape

```
print('смежность', y.is_contiguous())
print('вытягиваем', y.view(-1))
```

ошибка view size is not compatible with input tensor's size and stride

```
print('решейпим', y.reshape(-1))
print('делаем смежным и решейпим', y.contiguous().view(-1))
```

решейпим tensor([0, 2, 4, ...

делаем смежным и решейпим tensor([0, 2, 4, ...

Тензоры: размеры

```
x = torch.arange(8)
x.view(4, 2), x.view(2, -1)
(tensor([[0, 1],
         [2, 3],
         [4, 5],
         [6, 7]]),
 tensor([[0, 1, 2, 3],
         [4, 5, 6, 7]]))
```

Здесь сам тензор не поменялся, т.к. не было присваивания `x = x.view()`

**В Pytorch-е тензоры хранятся в формате `[channel, height, width]`,
в других системах чаще `[height, width, channel]`.**

Тензоры: размеры

```
# как хранятся данные, где следующий элемент по каждой из разметностей  
print (x, x.storage(), x.stride(), x.t().stride())
```

```
tensor([[0, 1],  
        [2, 3],  
        [4, 5],  
        [6, 7]]) 0
```

1

2

3

4

5

6

7

```
[torch.LongStorage of size 8] (2, 1) (1, 2)
```

Тензоры: транспонирование

```
z = torch.rand(1, 2, 3, 4)
z = z.permute(0, 3, 1, 2) # NxHxWxC -> NxCxHxW
z.shape
torch.Size([1, 4, 2, 3])
```

```
# другие способы транспонирования:
```

```
x.transpose(0, 1), x.t(), x.t_()
```

```
(tensor([[0, 2, 4, 6],
         [1, 3, 5, 7]]),
 tensor([[0, 2, 4, 6],
         [1, 3, 5, 7]]),
 tensor([[0, 2, 4, 6],
         [1, 3, 5, 7]]))
```

```
# векторизация
```

```
x.flatten() # ещё вариант .view(-1)
```

```
tensor([0, 2, 4, 6, 1, 3, 5, 7])
```

Тензоры: операции

аналогичны операциям в numpy, большинство операций выполняются поэлементно.

Поддерживаются операции линейной алгебры, многие из которых взяты из библиотек Basic Linear Algebra Subprograms (BLAS) и Linear Algebra Package (LAPACK).

Полный список операций линейной алгебры:

<https://pytorch.org/docs/stable/torch.html#blas-and-lapack-operations>

Тензоры: inplace-операции

**В inplace-операциях используется черта,
в этом случае операция выполняется на данном тензоре:**

```
x = torch.tensor([[1, 2], [3, 4]])  
  
# заполнение  
x.fill_(3) # черта - признак выполнения на данном тензоре  
# обнуление  
x.zero_()
```

**inplace-операции специально «запрятаны»,
так как при их использовании могут быть проблемы при распространении градиента**

Тензоры: операции

```
x = torch.tensor([[1, 2], [3, 4]])  
y = torch.tensor([[2, 2], [2, 2]])  
v = torch.tensor([1, 2])
```

сложение

```
print (x + y,  
       x.add(y))
```

черта означает inplace-операцию - меняется первый тензор:

```
x.add_(y)
```

поэлементное умножение

```
x * y, x.mul(y), torch.mul(x, y)
```

Тензоры: операции

Матричное умножение можно сделать по-разному:

torch.matmul – операция определена над тензорами,

можно указывать размерность для умножения (см)

torch.mm – обычное матричное умножение, но без приведения размеров (broadcasting)

torch.bmm – матричное умножение с поддержкой батчей: $(b \times n \times m) \cdot (b \times m \times p) = b \times n \times p$

```
# матричное умножение
x @ y, x.mm(y), x.matmul(y), torch.matmul(x, y)

(tensor([[ 6,  6],
         [14, 14]]),
 tensor([[ 6,  6],
         [14, 14]]),
 tensor([[ 6,  6],
         [14, 14]]),
 tensor([[ 6,  6],
         [14, 14]]))
```

Тензоры: операции

```
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[2, 2], [2, 2]])
v = torch.tensor([1, 2])

print (torch.dot(v, v), v.dot(v)) # скалярное умножение
torch.dot(x.view(-1), y.view(-1))

tensor(5) tensor(5)
tensor(20)

torch.mv(x, v), x.mv(v) # умножение на вектор
(tensor([ 5, 11]), tensor([ 5, 11]))
```


Тензоры: операции

Всё не будем описывать

Есть специальные операции типа $y + x*x$...

Определитель, с.з., ...

Решение уравнений, SVD, ...

```
x.type(torch.DoubleTensor).log()  
# приводим тип - иначе не работает log
```

```
x.pow(2), x**2  
(tensor([[ 1.,  4.],  
         [ 9., 16.]]),  
 tensor([[ 1.,  4.],  
         [ 9., 16.])))
```

Тензоры: статистики

Не забывать, `item` «выцепляет» элемент

```
x = torch.tensor([[1, 2], [3, 4]])  
y = torch.tensor([[2, 2], [2, 2]])
```

```
# sum  
torch.sum(x), x.sum(), x.sum().item(), x.sum(axis=0), x.sum(axis=1)  
(tensor(10), tensor(10), 10, tensor([4, 6]), tensor([3, 7]))
```

```
x.sum(axis=1, keepdim=True) # keepdim - тензор остаётся двумерным  
tensor([[3],  
        [7]])
```

Тензоры: статистики

max возвращает и индексы
но есть также **argmax**

```
torch.max(x), x.max(), x.max().item(), x.max(axis=0), x.max(axis=1)  
(tensor(4),  
 tensor(4),  
 4,  
 torch.return_types.max(  
  values=tensor([3, 4]),  
  indices=tensor([1, 1])),  
 torch.return_types.max(  
  values=tensor([2, 4]),  
  indices=tensor([1, 1])))
```

Тензоры: статистики

Есть стандартные: среднее, медиана, мода, std

```
x = torch.tensor([2,1,2,3,0,4,3])  
x.topk(k=2), x.kthvalue(k=2)
```

```
(torch.return_types.topk(  
    values=tensor([4, 3]),  
    indices=tensor([5, 3])),  
torch.return_types.kthvalue(  
    values=tensor(1),  
    indices=tensor(1)))
```

```
x = torch.tensor([1., 2., 3.])  
# сразу вычислить СКО / дисперсию и среднее  
torch.std_mean(x), torch.var_mean(x)  
((tensor(1.), tensor(2.)), (tensor(1.), tensor(2.)))
```

Тензоры: приведение размеров

- когда 2 размерности совпадают
- когда одна размерность равна 1

не надо вручную приводить размеры «размножая тензор»

```
x = torch.tensor([[1, 2], [3, 4]])  
y = torch.tensor([10, 20])  
  
x + y  
tensor([[11, 22],  
        [13, 24]])
```

Тензоры: приведение размеров

```
x = torch.tensor([[1, 2]])
y = torch.tensor([[0], [3]])
print (x, x.shape)
print (y, y.shape)
z = x + y
print (z, z.shape)
tensor([[1, 2]]) torch.Size([1, 2])
tensor([[0],
        [3]]) torch.Size([2, 1])
tensor([[1, 2],
        [4, 5]]) torch.Size([2, 2])
```

Тензоры: связь с Numpy

При переводе в numpy необходимо, чтобы тензор находился в CPU, а не на GPU.

```
x = torch.tensor([[1, 2], [3, 4]])
x.numpy()
y = x.cpu().detach().numpy() # правильное так
print (y)
array([[1, 2],
       [3, 4]], dtype=int64)
```

```
x[0, 0] = 10
print (y)
[[10  2]
 [ 3  4]]
```

Тензоры: связь с Numpy

```
# из numpy.array в pytorch-тензор
torch.from_numpy(np.ones((2, 2)))
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64)
```


Тензоры: сохранение и загрузка

```
# сохранение и загрузка тензоров
```

```
torch.save(x, 'x-file')
```

```
x2 = torch.load("x-file")
```

```
x2
```

```
mydict = {'x': x, 'y': y}
```

```
torch.save(mydict, 'mydict')
```

```
mydict2 = torch.load('mydict')
```

```
mydict2
```

```
{ 'x': tensor([1., 2., 3.]),
```

```
  'y': tensor([[ 3.,  0.],
```

```
            [ 3., 12.]], requires_grad=True) }
```

GPU

Тензоры и вычислительные графы (нейросети) могут находиться как на CPU, так и на GPU (ещё на TPU).

**Переменные и модели на разных устройствах не видят друг друга!
Поэтому их надо перенести на один вычислитель.**

На GPU вычисления производятся существенно быстрее из-за параллелизма.

Такие записи эквивалентны для обозначения устройства-GPU:

```
device="cuda"  
device=torch.device("cuda")  
device="cuda:0"  
device=torch.device("cuda:0")
```

```
x.cuda()  
x.cpu()  
x.is_cuda  
z.to("cpu", torch.double)
```

GPU

```
# самая популярная конструкция, определяющая доступное для хранения
# тензоров/нейросетей устройство
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
m.to(device) # перенос на доступное устройство
```

```
# информация о GPU
```

```
print('Using device:', device)
print()
```

```
if device.type == 'cuda':
    print(torch.cuda.get_device_name(0))
    print('Memory Usage:')
    print('Allocated:',
          round(torch.cuda.memory_allocated(0)/1024**3,1), 'GB')
    print('Cached:    ',
          round(torch.cuda.memory_reserved(0)/1024**3,1), 'GB')
```

GPU – очистка памяти

```
# чистка памяти

gc.collect() # Python thing

with torch.no_grad(): # работает с этой инструкцией
    torch.cuda.empty_cache()
```

GPU

```
x = torch.randn(5000, 5000)
```

```
# CPU
```

```
_ = torch.matmul(x, x)
```

```
CPU time: 0.77800s
```

```
# 2GPU
```

```
x = x.to("cuda:0")
```

```
CPU2GPU time: 0.09263s
```

```
# GPU
```

```
x = x + 0.0 # просто первая операция
```

```
_ = torch.matmul(x, x)
```

```
GPU time: 0.00803s
```

Инициализация генератора псевдослучайных чисел

```
def set_seed(seed):  
    np.random.seed(seed)  
    torch.manual_seed(seed)  
    if torch.cuda.is_available(): # для GPU отдельный seed  
        torch.cuda.manual_seed(seed)  
        torch.cuda.manual_seed_all(seed)  
  
set_seed(42)  
  
# есть стохастические операции на GPU  
# сделаем их детерминированными для воспроизводимости  
torch.backends.cudnn.deterministic = True  
torch.backends.cudnn.benchmark = False
```

Авто дифференцирование (AutoGrad)

**Сначала приведём пример автоматического дифференцирования,
мы зададим функцию $y = \sin(x) \cdot (\sin^2(x) + \cos^2(x))$,
а Pytorch автоматически вычислит её производную**

**backward – функция «обратного прохода»,
именно при её вызове автоматически считаются производные по всем переменным,
у которых `requires_grad=True`**

inplace-операции не работают в графе

Авто дифференцирование (AutoGrad)

```
from torch.autograd import Variable

x = torch.linspace(-2, 2, 101, dtype=torch.float32, requires_grad=True)
# x = Variable(x, requires_grad=True) # можно ли проще? - да!
y = torch.sin(x) * (torch.sin(x) ** 2 + torch.cos(x) ** 2)
# здесь это прямой проход
# дальше часто будем определять класс с методом y.forward()
y.sum().backward() # превращаем в число
                    # (только от таких функций берётся градиент)
g = x.grad # взятие производных в каждой точке
```

```
x = torch.linspace(-2, 2, 3, dtype=torch.float32)
# над целочисленными тензорами нельзя взять производную
x.requires_grad_() # как указать, что хотим вычислять производную
x
tensor([-2.,  0.,  2.], requires_grad=True)
```



```
import torch
from torch.autograd import Variable

#x = Variable(torch.Tensor([1]), requires_grad=True)
#y = Variable(torch.Tensor([2]), requires_grad=True)
#z = Variable(torch.Tensor([3]), requires_grad=True)

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

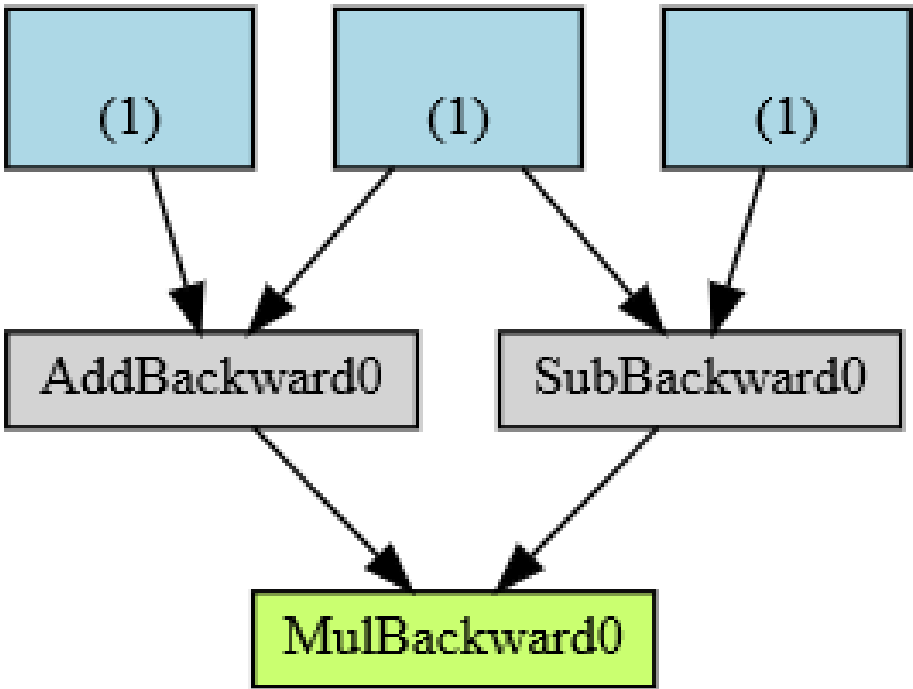
f = (x + y) * (y - z)

f.backward()
x.grad, y.grad, z.grad
```

```
from torchviz import make_dot
make_dot(f)
```

Авто дифференцирование (AutoGrad)

Граф вычислений



Авто дифференцирование (AutoGrad)

Если устраняем вычисления градиентов - считается быстрее

```
f = (x + y) * (y - z)
z = f.detach()
print (f.requires_grad, z.requires_grad)

with torch.no_grad(): # когда просто нужен прямой проход -
                      # и не надо считать градиенты
    f = (x + y) * (y - z)
print (f.requires_grad)
True False
False
```

Авто дифференцирование (AutoGrad)

```
x = torch.Tensor([1, 2, 3])
w = torch.tensor([1., 1, 1], requires_grad=True)
z = w @ x
z.backward()
print(x.grad, # по этому не указывали возможность взятия градиента
      w.grad, # должен выводиться x
      sep=' \n')
```

None

tensor([1., 2., 3.])

```
z = w @ x
z.backward()
print(x.grad,
      w.grad, # идёт накопление!!!
      sep=' \n')
```

None

tensor([2., 4., 6.])

Авто дифференцирование (AutoGrad)

```
with torch.no_grad(): # нет накопления
```

```
    z = w @ x
```

```
    # z.backward()
```

```
print(x.grad, w.grad, sep='\n')
```

```
w.grad.data.zero_() # а так - совсем обнулить
```

```
z = w @ x
```

```
z.backward()
```

```
print(x.grad, w.grad, sep='\n')
```

```
None
```

```
tensor([2., 4., 6.])
```

```
None
```

```
tensor([1., 2., 3.])
```

```
# w.numpy() - будет ошибка
```

```
w.detach().numpy() # создаётся копия, которую можно в пр -  
    у неё requires_grad=False
```

```
array([1., 1., 1.], dtype=float32)
```

Авто дифференцирование (AutoGrad)

```
# Иллюстрация взятия градиента - с detach
x = torch.tensor([2.], requires_grad=True)

y = x * x
y.detach_() # добавили
z = x * y
# а тут не работает
z.backward()

print(x.grad) # (2*2*x)' = 4
tensor([4.])
```

Авто дифференцирование (AutoGrad)

```
# ещё одна иллюстрация detach()  
import torch  
x = torch.tensor([2.], requires_grad=True)  
print(x)  
print(x.detach())  
tensor([2.], requires_grad=True)  
tensor([2.])
```

```
# динамический граф вычислений в цикле
x = torch.tensor([[1, 2], [3, 4]], requires_grad=True, dtype=torch.float32)
x0 = x
for _ in range(2):
    x = x * x

z = x.mean() # здесь будет 1/4 !!!
z.backward()
print(x, '\n', x.grad)
print(x0, '\n', x0.grad) # градиент лежит здесь!!!
# поскольку x превратился во внутреннюю вершину графа вычислений
tensor([[ 1., 16.],
        [ 81., 256.]], grad_fn=<MulBackward0>)
None
tensor([[1., 2.],
        [3., 4.]], requires_grad=True)
tensor([[ 1., 8.],
        [27., 64.]])
<ipython-input-123-7397ba33abd6>:11: UserWarning: The .grad attribute of a Tensor that
is not a leaf Tensor is being accessed. ...
```


Возможность вычислять отдельные производные

```
x = torch.tensor([[1., 2, 3, 4]], requires_grad=True)
# x = Variable(torch.FloatTensor([[1, 2, 3, 4]]), requires_grad=True)
z = x ** 2 / 2
loss = z.sum(dim=1)

# backward для первого элемента z
z.backward(torch.FloatTensor([[1, 0, 0, 0]]), retain_graph=True)
print(x.grad.data)
x.grad.data.zero_()
```

```
tensor([[1., 0., 0., 0.]])
```

Возможность вычислять отдельные производные

```
# backward для всех элементов z с равными  
весами z.backward(torch.FloatTensor([[1, 1, 1, 1]]), retain_graph=True)  
print(x.grad.data)  
x.grad.data.zero_()
```

```
# обычный backward для loss  
loss.backward() # ~ loss.backward(torch.FloatTensor([1.0]))  
print(x.grad.data)
```

```
tensor([[1., 2., 3., 4.]])  
tensor([[1., 2., 3., 4.]])
```

Два обратных прохода по графу

```
a = torch.rand(1, 4, requires_grad=True)
b = a**2
c = b*2
d = c.mean()
e = c.sum()

# если так
d.backward() #
e.backward() # тут будет ошибка
# RuntimeError: Trying to backward through the graph a second time ...

# надо так
d.backward(retain_graph=True) #
e.backward(retain_graph=True) #
```

Часть 2 – предобработка данных, нейросети

TensorDataset / DataLoader – организация подачи данных в модель

TensorDataset – для представления датасета.

Часто приходится определять свой датасет, наследуя этот класс.

DataLoader – подаёт батчами данные, позволяет итерироваться по датасету, автоматически формируя батчи (и делая некоторые сопутствующие действия).

В DataLoader может передаваться сэмплер.

```
from torch.utils.data import TensorDataset
import numpy as np
from torch.utils.data import DataLoader
```

```
x = torch.from_numpy(np.vstack([np.arange(10, dtype='float32'),  
                                np.ones(10, dtype='float32')]).T)  
y = torch.from_numpy(np.arange(10, dtype='float32')[:, np.newaxis] ** 2)  
  
train_ds = TensorDataset(x, y)  
train_dl = DataLoader(train_ds, batch_size=4, shuffle=True)  
  
for xb, yb in train_dl:  
    print(xb)  
    print(yb)
```

```
tensor([[3., 1.],  
        [9., 1.],  
        [7., 1.],  
        [5., 1.]])  
tensor([[ 9.],  
        [81.],  
        [49.],  
        [25.]])  
tensor([[1., 1.],  
        [8., 1.],  
        [6., 1.],  
        [4., 1.]])  
tensor([[ 1.],  
        [64.],  
        [36.],  
        [16.]])  
tensor([[2., 1.],  
        [0., 1.]])  
tensor([[4.],  
        [0.]])
```

TensorDataset

Есть много вспомогательных полезных функций.

Например, можно разделить датасет на части.

```
train_size = int(0.8 * len(ds))
test_size = len(ds) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(ds,
                                                             [train_size, test_size])
```

TensorDataset / DataLoader: как список передать в нейронку

```
import torch
import numpy as np
from torch.utils.data import TensorDataset, DataLoader

my_x = [np.array([[1.0,2],[3,4]]), np.array([[5.,6],[7,8]])] # a list of
numpy arrays
my_y = [np.array([4.]), np.array([2.])] # another list of numpy arrays
(targets)

tensor_x = torch.Tensor(my_x) # transform to torch tensor
tensor_y = torch.Tensor(my_y)

my_dataset = TensorDataset(tensor_x, tensor_y) # create your dataset
my_dataloader = DataLoader(my_dataset) # create your dataloader
```


Загрузка с трансформациями

```
from torchvision import datasets

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ColorJitter(brightness=(0.6, 1),
                                              contrast=(0.8, 1)),
                       transforms.RandomRotation((-15, 15)),
                       # перевод в тензор + нормировка на отрезок
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=64, shuffle=True, num_workers=8, pin_memory=True)
```

с num_workers лучше поэкспериментировать!

Это число подпроцессов для загрузки данных.

Чаще pin_memory=True если используем GPU. Предварительно копирует данные на GPU.

Загрузка с трансформациями

Трансформации можно и нужно делать разные для обучения и теста

И тест не перемешиваем

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,  
                                         download=True, transform=transform)  
testset = torchvision.datasets.CIFAR10(root='./data', train=False,  
                                         download=True, transform=transform)
```

Своя трансформация

```
class Noise():
    """Adds gaussian noise to a tensor.
    """
    def __init__(self, mean, stddev):
        self.mean = mean
        self.stddev = stddev

    def __call__(self, tensor):
        noise = torch.zeros_like(tensor).normal_(self.mean, self.stddev)
        return tensor.add_(noise)

    def __repr__(self):
        repr = f"{self.__class__.__name__} (mean={self.mean},
                                                    stddev={self.stddev}) "
        return repr
```

Создание своего датасета

```
class CustomTextDataset(Dataset):  
    '''  
    Simple Dataset initializes with X and y vectors  
    '''  
    def __init__(self, X, y=None):  
        self.data = list(zip(X, y))  
        # Sort by length of first element in tuple  
        self.data = sorted(self.data, key=lambda x: len(x[0]))  
  
    def __len__(self):  
        # raise NotImplementedError  
        return len(self.data)  
  
    def __getitem__(self, idx):  
        # raise NotImplementedError  
        return self.data[idx]
```

Модуль nn

Простейший линейный слой

```
from torch import nn
model = nn.Linear(in_features=2,
                  out_features=1,
                  bias=True)

print(model.weight)
print(model.bias)
list(model.parameters())
```

```
Parameter containing:
tensor([[ -0.0243, -0.1254]], requires_grad=True)
Parameter containing:
tensor([-0.4057], requires_grad=True)
[Parameter containing:
  tensor([[ -0.0243, -0.1254]], requires_grad=True),
 Parameter containing:
  tensor([-0.4057], requires_grad=True)]
```

Модуль nn: простейшее задание нейронной сети

```
# через nn.Sequential
net = nn.Sequential(nn.Linear(10, 5),
                    nn.ReLU(),
                    nn.Linear(5, 2))
```

```
# через nn.Sequential, но с удобными именами
from collections import OrderedDict
net1 = nn.Sequential(OrderedDict([('hidden_linear', nn.Linear(10, 5)),
                                  ('hidden_activation', nn.ReLU()),
                                  ('output', nn.Linear(5, 2))]))
```

```
X = torch.rand(3, 10)
net(X)
tensor([[[-0.2359,  0.1572],
         [-0.0946,  0.0312],
         [-0.2290,  0.0459]]], grad_fn=<AddmmBackward>)
```

Модуль nn: простейшее задание нейронной сети

```
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self):
        # инициализация параметров
        # обращение к инициализации родителя
        super().__init__() # super(MLP, self).__init__()
        self.hidden = nn.Linear(10, 5) # Hidden layer
        self.out = nn.Linear(5, 2) # Output layer

    def forward(self, X):
        # как обрабатываются данные и получается ответ
        return self.out(F.relu(self.hidden(X)))
```

наследуя от nn.Module и прописывая __init__ и forward

Модуль nn: простейшее задание нейронной сети

```
class NNN(torch.nn.Module):  
    def __init__(self):  
        super(NNN, self).__init__()  
  
        self.layers = torch.nn.Sequential()  
        self.layers.add_module('lin1', torch.nn.Linear(10, 5))  
        self.layers.add_module('relu1', torch.nn.ReLU())  
        self.layers.add_module('lin2', torch.nn.Linear(5, 2))  
  
    def forward(self, input):  
        return self.layers(input)
```

присоединение модулей с помощью .add_module

Модуль nn: простейшее задание нейронной сети

forward – главная ф-ия определяет прямой проход и, собственно, функционирование сети
(обратный проход не прописывается отдельным методом)

Там может быть что угодно

- **и на низком уровне**
- **и композиция других модулей**

**Но когда мы делаем прямой проход, то явно не вызываем метод forward,
а используем имя НС: net(X)**

(т.к. до и после вызова неявно ещё кое-что выполняется, ex: `_forward_pre_hooks`)

Модуль nn: простейшее задание нейронной сети

```
net
```

```
Sequential(  
  (0): Linear(in_features=10, out_features=5, bias=True)  
  (1): ReLU()  
  (2): Linear(in_features=5, out_features=2, bias=True)  
)
```

```
net2
```

```
MLP(  
  (hidden): Linear(in_features=10, out_features=5, bias=True)  
  (out): Linear(in_features=5, out_features=2, bias=True)  
)
```

```
net3
```

```
NNN(  
  (layers): Sequential(  
    (lin1): Linear(in_features=10, out_features=5, bias=True)  
    (relu1): ReLU()  
    (lin2): Linear(in_features=5, out_features=2, bias=True)  
  ) )
```

Модуль nn: доступ к параметрам НС

```
net[0], net2.hidden, net3.layers.relu1  
(Linear(in_features=10, out_features=5, bias=True),  
 Linear(in_features=10, out_features=5, bias=True),  
 ReLU())
```

```
net.state_dict() # все параметры сети
```

```
OrderedDict([('0.weight',  
            tensor([[ -0.1661, -0.1646, -0.148,  0.2944,  0.079,  0.178],  
                    [ 0.0149,  0.2513,  0.094, -0.3056, -0.097, -0.160],  
                    [-0.1832, -0.0631,  0.015, -0.0190, -0.276, -0.036],  
                    [ 0.1317, -0.0984,  0.307, -0.1473, -0.066,  0.109],  
                    [ 0.1706, -0.2755,  0.303, -0.1257, -0.241, -0.314]])),  
            ('0.bias', tensor([ 0.1352, -0.032,  0.1371])),  
            ('2.weight',  
            tensor([[ 0.0051,  0.3829, -0.1443, -0.0035, -0.2843],  
                    [ 0.2041, -0.2619,  0.3165,  0.3838, -0.1594]])),  
            ('2.bias', tensor([-0.2393,  0.0215]))])
```

Модуль nn: доступ к параметрам НС

```
net[0].bias.data, net2.hidden.bias.data  
(tensor([-0.1051, -0.1820,  0.1631,  0.2234, -0.3068]),  
 tensor([ 0.3139, -0.1659, -0.0243,  0.2580,  0.1043]))
```

```
net[0].bias.grad, net2.hidden.bias.grad  
# не было обучения (BP), поэтому None  
(None, None)
```

```
# считаем число параметров  
numel_list = [p.numel() for p in net.parameters() if p.requires_grad == True]  
sum(numel_list), numel_list  
(67, [50, 5, 10, 2])
```

Разделение параметров

```
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
```

Свои модули (слои), задание модулей

```
# центрирующий слой без параметров

class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

Явное прописывание весов

```
class MyNetworkWithParams(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyNetworkWithParams, self).__init__()
        # параметр автоматически регистрируется как параметр модуля
        self.layer1_weights = nn.Parameter(torch.randn(input_size,
                                                         hidden_size))
        self.layer1_bias = nn.Parameter(torch.randn(hidden_size))
        self.layer2_weights = nn.Parameter(torch.randn(hidden_size,
                                                         output_size))
        self.layer2_bias = nn.Parameter(torch.randn(output_size))

    def forward(self, x):
        h1 = torch.matmul(x, self.layer1_weights) + self.layer1_bias
        h1_act = torch.max(h1, torch.zeros(h1.size())) # ReLU
        output = torch.matmul(h1_act, self.layer2_weights) + self.layer2_bias
        return output

net = MyNetworkWithParams(32, 128, 10)
```

Списки слоёв

```
class MyNet(nn.Module):
    def __init__(self, n_hidden_layers):
        super(MyNet, self).__init__()
        self.n_hidden_layers = n_hidden_layers
        self.final_layer = nn.Linear(128, 10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128, 128))
        self.hidden = nn.ModuleList(self.hidden) # это важно!
    def forward(self, x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```

Инициализация

Можно применять и по отдельным модулям

```
def init_normal(m):  
    if type(m) == nn.Linear:  
        nn.init.normal_(m.weight, mean=0, std=0.01)  
        nn.init.zeros_(m.bias)  
  
def init_normal_v2(m):  
    if isinstance(m, nn.Linear):  
        torch.nn.init.xavier_uniform_(m.weight)  
        m.bias.data.fill_(0.01)  
  
net.apply(init_normal)  
print (net[0].weight.data[0], net[0].bias.data[0])  
  
net.apply(init_normal_v2)  
print (net[0].weight.data[0], net[0].bias.data[0])  
tensor([-0.0013,  0.0183]) tensor(0.)  
tensor([ 0.5608, -0.1452]) tensor(0.0100)
```


Инициализация

```
# способ, где копируются веса в нужное место
def init_custom(m):
    if type(m) == nn.Linear:
        rw = torch.randn(m.weight.data.size())
        m.weight.data.copy_(rw)
```

Модуль nn: как обучать сеть

```
def train_epoch(model, train_loader, criterion, optimizer):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad(set_to_none=True) # ~ model.zero_grad()
        # = 0 чтобы не накапливались
        data = data.to(device)
        target = target.to(device) # перенос на device

        outputs = model(data) # получили выход сетки
        loss = criterion(outputs, target) # посчитали для этого выхода лосс
        running_loss += loss.item()
        loss.backward() # вычислили градиенты loss по параметрам сети (w)
        optimizer.step() # сдalem шаг по антиградиенту - обновляем веса сети
    running_loss /= len(train_loader)

    return running_loss
```

Loss-функции

```
# [S] CrossEntropyLoss = Softmax + CrossEntropy
# тот же эффект - logSoftmax + NLLLoss
loss = nn.CrossEntropyLoss()
a = torch.tensor([[1.0, 2.0, 3.0]])
y = torch.tensor([1])

print (a, y, loss(a, y))
```

Своя Loss-функция

```
class CustomNLLLoss(nn.Module):  
    def __init__(self):  
        super().__init__()  
    def forward(self, x, y):  
        # x should be output from LogSoftmax Layer  
        log_prob = -1.0 * x  
        # Get log_prob based on y class_index as loss=-mean(ylogp)  
        loss = log_prob.gather(1, y.unsqueeze(1))  
        loss = loss.mean()  
        return loss  
  
criterion = CustomNLLLoss() # nn.NLLLoss()  
CustomNLLLossClass = criterion(preds, y)
```

Темп обучения, программы изменения темпов

```
# свой темп для каждого слоя

from torch import optim

optim.SGD([
    {'params': model.features.parameters()},
    {'params': model.classifier.parameters(), 'lr': 1e-3}
], lr=1e-2, momentum=0.9)
```

Темп обучения, программы изменения темпов

```
from torch.optim.lr_scheduler import StepLR,
                                ReduceLROnPlateau, CosineAnnealingLR

# сообщаем параметры оптимизатору!
optimizer = optim.SGD(net.parameters(), lr)
# сообщаем оптимизатор "шедьюлеру"
scheduler_1 = ReduceLROnPlateau(optimizer, factor=0.1,
                                patience=1, threshold=0.1)
# умножаем на gamma через каждые step_size шагов
scheduler_2 = StepLR(optimizer, step_size=1, gamma=0.1)
# ....
# шаг на эпоху должен быть один, сделаем его после валидации
if not is_train:
    scheduler_1.step(running_loss / (i + 1))
    scheduler_2.step()
```

Свой оптимизатор

```
class OptimizerTemplate:

    def __init__(self, params, lr):
        self.params = list(params)
        self.lr = lr

    def zero_grad(self):
        ## Set gradients of all parameters to zero
        for p in self.params:
            if p.grad is not None:
                p.grad.detach_() # For second-order optimizers important
                p.grad.zero_()
```

далее продолжение

```
@torch.no_grad()
def step(self):
    ## Apply update step to all parameters
    for p in self.params:
        if p.grad is None: # We skip parameters without any gradients
            continue
        self.update_param(p)

def update_param(self, p):
    # To be implemented in optimizer-specific classes
    raise NotImplementedError
```

```
class SGD(OptimizerTemplate):
    def __init__(self, params, lr):
        super().__init__(params, lr)
    def update_param(self, p):
        p_update = -self.lr * p.grad
        p.add_(p_update) # In-place => saves memory + doesn't create c. graph
```


Сохранение / загрузка сети

```
torch.save(model, "/tmp/model.pth")
model = torch.load("/tmp/model.pth")

torch.save(model.state_dict(), "/tmp/model.pth") # только параметры
model = MLP()
model.load_state_dict(torch.load("/tmp/model.pth")) # только параметры
model.to(device) # вот сейчас переносим
model.eval()
```

```
state = {
    'epoch': epoch + 1,
    'state_dict': net.state_dict(),
    'optimizer' : optimizer.state_dict()
}

torch.save(state, './my_checkpoint.pth')
```

Своя функция активации

```
@torch.jit.script
def fused_gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

```
class MySigmoid (nn.Module):

    def __init__(self):
        super().__init__()
        self.name = self.__class__.__name__
        self.config = {"name": self.name}

    def forward(self, x):
        return 1 / (1 + torch.exp(-x))
```

Использование нейросети

```
# валидация
model.eval() # переводим сеть в нужный режим -
              # меняет поведение forward() - влияет на DO, BN
with torch.no_grad(): # контекстный менеджер, в котором не вычисляются
                      # градиенты, даже у тензоров с requires_grad=True
    train, y_train = train_dataset.tensors
    # train, y_train = train.to(device), y_train.to(device)
    train_preds = model(train)
    train_loss = loss_fn(train_preds, y_train).item()
```

Будет в лекциях

свёртки

пулинг

падинг

dropout

нормализация

Transfer Learning

**В torchvision.models есть готовые модели,
см. <https://pytorch.org/vision/stable/models.html>**

```
# берём готовую модель
from torchvision import models
transfer_model = models.resnet34(pretrained=True)
transfer_model.eval() # чтобы нормально работали BN и DO
```

```
# заморозка слоёв
for name, param in model_cnn.named_parameters():
    if ("bn" not in name): # BN лучше не замораживать!
        param.requires_grad = False
```

Transfer Learning

```
from torchvision import models

resnet18 = models.resnet18(pretrained=True)

for parameter in resnet18.parameters(): # тут на самом деле веса,
                                         # но не как dict
    parameter.requires_grad = False

#заменяем слой
resnet18.fc = nn.Linear(512, 10)
```

**не забыть трансформировать объект также,
как он был трансформирован при обучении сети**

Transfer Learning

Удобное представление модели получается с помощью библиотеки `torchsummary`.

```
from torchvision import models
from torchsummary import summary

vgg = models.vgg16()
summary(vgg, (3, 224, 224))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 224, 224]	1,792
ReLU-2	[-1, 64, 224, 224]	0
Conv2d-3	[-1, 64, 224, 224]	36,928
ReLU-4	[-1, 64, 224, 224]	0
MaxPool2d-5	[-1, 64, 112, 112]	0
Conv2d-6	[-1, 128, 112, 112]	73,856
ReLU-7	[-1, 128, 112, 112]	0
Conv2d-8	[-1, 128, 112, 112]	147,584
ReLU-9	[-1, 128, 112, 112]	0

MaxPool2d-10	[-1, 128, 56, 56]	0
Conv2d-11	[-1, 256, 56, 56]	295,168
ReLU-12	[-1, 256, 56, 56]	0
Conv2d-13	[-1, 256, 56, 56]	590,080
ReLU-14	[-1, 256, 56, 56]	0
Conv2d-15	[-1, 256, 56, 56]	590,080
ReLU-16	[-1, 256, 56, 56]	0
.		
MaxPool2d-31	[-1, 512, 7, 7]	0
AdaptiveAvgPool2d-32	[-1, 512, 7, 7]	0
Linear-33	[-1, 4096]	102,764,544
ReLU-34	[-1, 4096]	0
Dropout-35	[-1, 4096]	0
Linear-36	[-1, 4096]	16,781,312
ReLU-37	[-1, 4096]	0
Dropout-38	[-1, 4096]	0
Linear-39	[-1, 1000]	4,097,000
=====		
Total params: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

Input size (MB): 0.57		
Forward/backward pass size (MB): 218.78		
Params size (MB): 527.79		
Estimated Total Size (MB): 747.15		

Transfer Learning

```
# ещё способ посмотреть на параметры модели
from prettytable import PrettyTable

def count_parameters(model):
    table = PrettyTable(["Modules", "Parameters"])
    total_params = 0
    for name, parameter in model.named_parameters():
        if not parameter.requires_grad: continue
        params = parameter.numel()
        table.add_row([name, params])
        total_params+=params
    print(table)
    print(f"Total Trainable Params: {total_params}")
    return total_params

count_parameters(vgg)
```

Transfer Learning

+-----+-----+	
Modules	Parameters
+-----+-----+	
features.0.weight	1728
features.0.bias	64
features.2.weight	36864
features.2.bias	64
features.5.weight	73728
features.5.bias	128
.	
features.28.weight	2359296
features.28.bias	512
classifier.0.weight	102760448
classifier.0.bias	4096
classifier.3.weight	16777216
classifier.3.bias	4096
classifier.6.weight	4096000
classifier.6.bias	1000
+-----+-----+	
Total Trainable Params: 138357544	
138357544	

Ссылки

- https://github.com/MLWhiz/data_science_blogs/blob/master/pytorch_guide/Pytorch%20Guide.ipynb
- <https://d2l.ai/>
- <https://atcold.github.io/pytorch-Deep-Learning/>
- семинары OzonMasters
- <https://habr.com/ru/post/334380/>
- <https://github.com/andriygav/MachineLearningSeminars/>
- <https://uvadlc-notebooks.readthedocs.io/en/latest/index.html>
- книга Joe Papa «PyTorch Pocket Reference» 2021, <https://github.com/joe-papa/pytorch-book>
- <https://github.com/vahidk/EffectivePyTorch>