Дата последней сборки документа:
20 февраля 2024 г. в 02:07

Приемы работы с библиотекой PyTorch

Содержание

1. Вводные замечания

Чаще всего цикл обучения модели реализуют в виде обычного цикла `for` Python. Оптимизатор, доступный в модуле `torch.optim` PyTorch, который будет отвечать за обновление параметров.

По умолчанию в PyTorch используется модель немедленного выполнения (eager mode). Как только интерпретатор Python выполняет инструкцию, связанную с PyTorch, базовая реализация C++ или CUDA сразу же производит соответствующую операцию.

PyTorch также предоставляет возможности предварительной компиляции моделей с помощью TorchScript. Используя TorchScript, PyTorch может преобразовать модель в набор инструкций, которые можнро независимо вызывать из Python, допустим, из программ на C++ или на мобильных устройствах. Это можно считать своего рода виртуальной машиной с ограниченным набором инструкций, предназначенным для опреаций с тензорами. Экспортировать модель можно либо в виде TorchScript для использования со средой выполнения Python, либо в стандартизированном формате ONNX (платформонезависимый формат описания моделей).

Сети среднего размера могут потребовать от нескольких часов до нескольких дней для обучения с нуля на больших реальных наборах данных на рабочих станциях с хорошим GPU [1, стр. 48]. Длительность обучения можно сократить за счет использования на одной машине нескольких GPU или даже еще сильнее – на кластере машин, оснащенных несколькими GPU.

Для примера создадим сеть AlexNet

```
# TorchVision включает несколько лучших нейросетевых архитектур для машинного зрения
from torchvision import models

alexnet = models.AlexNet()
```

Подав на вход `alexnet` данные четко определенного размера, мы выполним прямой проход (forward pass) по сети, при котором входной сигнал пройдет через первый набор нейронов, выходные сигналы которых будут поданы на вход следующего набора нейронов, и так до самого

итогового выходного сигнала. На практике это означает, что при наличии объекта `input` нужного типа можно произвести прямой проход с помощью оператора `ouput = alexnet(input)`.

Но если мы так поступим, то получим мусор. А все потому, что сеть не была инициализирована: ее веса, числа, с которыми складываются и на которые умножаются входные сигналы, не были обучены на чем-либо, сеть сама по себе – чистый (или, точнее, сказать случайный) лист. Необходимо либо обучить ее с нуля, либо загрузить веса, полученные в результате предыдущего обучения [1, стр. 58].

В `models` названия в верхнем регистре соответствуют классам, реализующим популярные архитектуры, предназначенные для машинного зрения. С другой стороны, названия в нижнем регистре соответствуют функциям, создающим экземпляры моделей с заранее определенным количеством слоев и нейронов, а также, возможно, скачивающие и загружающие в них предобученные веса.

Для того чтобы привести входные изображения к нужному размеру, а их значения (цвета) примерно в один числовой диапазон, можно воспользоваться преобразованиями модуля `torchvision`

```
from torchvision import transforms

# это функция
preprocess = transformers.Compose([
    transformers.Resize(256),
    transformers.CenterCrop(224),
    transformers.ToTensor(),
    transformers.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

Здесь описана функция `preprocess`, масштабирующую входное изображение до размера 256×256, обрезающую его до 224×224 по центру, преобразующую в тензор (многомерный массив PyTorch: в данном случае трехмерных массив, содержащий цвет, высоту и ширину) и нормализующую его компоненты RGB (красный, зеленый, синий) до заданных среднего значения и стандартного отклонения.

Если мы хотим получить от сети осмысленные ответы, все это должно соответствовать данным, полученным сетью во время обучения.

Процесс выполнения обученной модели на новых данных в сфере глубокого обучения называется *выводом* (inference). Для выполнения вывода необходимо перевести сеть в режим `eval`

```
resnet.eval()
```

Если забыть сделать это, некоторые предоубченные модели, например включающие нормализацию по мини-батчам и дропаут, не дадут никаких осмысленных результатов просто по причине их внутреннего устройства. Теперь, после установки режима `eval`, можно выполнять вывод

```
out = resnet(batch_t)
# получается что-то вроде степени уверенности модели в конкретном предсказании
percentage = torch.nn.functional.softmax(out, dim=1)[0] * 100
```

Успешность работы сети во многом зависит от наличия соответствующих объектов в обучающем наборе данных. Если подать нейронной сети нечто выходящее за рамки обучающего набора данных, вполне возможно, что она достаточно уверенно вернет неправильный ответ [1, стр. 64].

Сеть представляет собой всего лишь каркас, а вся суть – в весовых коэффициентах [1, стр. 69].

1.1. Torch Hub

Автору, чтобы опубликовать модель через механизм Torch Hub, необходимо всего лишь поместить файл `hubconf.py` в корневой каталог репозитория GitHub. Структура очень проста

```
# необязательный список модулей, от которых зависит данный код
dependencies = ["torch", "math"]

# одна или несколько функций, открываемых пользователям в качестве входных точек репозитория. Эт
    и функции должны инициализировать модели в соответствии с агрументами и возвращать их
def some_entry_fn(*args, **kwargs):
    model = build_some_model(*args, **kwargs)
    return model

def another_entry_fn(*args, **kwargs):
    model = build_another_model(*args, **kwargs)
    return model
```

Теперь интересные предобученные модели можно искать в репозиториях GitHub, содержащих файл `hubconf.py`, зная сразу же, что их можно будет загрузить с помощью модуля torch.hub.

```
import torch
from torch import hub

resnet18_model = hub.load(
    "pytorch/vision:main",  # название и ветка репозитория GitHub
    "resnet18",  # название точки входа
    pretrained=True  # ключевой аргумент
)
```

Приведенный код скачивает копию состояния ветки `main` репозитория `pytorch/vision`, вместе с весовыми коэффициентами в локальный каталог (по умолчанию `.torch/hub` в домашнем каталоге) и выполняет функцию точки входа `resnet18`, возвращающую созданный экземпляр модели.

2. Тензор

В контексте глубокого обучения тензоры связаны с обобщением векторов и матриц на произвольную размерность. Другими словами, речь идет о многомерных массивах.

По сравнению с массивами NumPy тензоры PyTorch обладают несколькими потрясающими способностями, например возвожностью чрезвычайно быстро выполнять операции на графических процессорах, умением распределять операции по нескольким устройствам или машинам, а также отслеживать породивший их граф вычислений.

Тензоры PyTorch и массивы NumPy это представления над (обычно) *непрерывными блоками памяти*, содержащими распакованные (unboxed) числовые типы данных Си, *а не объекты Python* [1, стр. 83].

Пример тензора

```
img_t = torch.randn(3, 5, 5)
batch_t = torch.tensor(2, 3, 5, 5)  # [батч, каналы, строки, столбцы]
```

Иногда каналы RGB размещаются в измерении 0, а иногда – в измерении 1. Но обобщение можно производить путем отсчета с конца: *каналы* всегда расположены в измерении -3, третьем с конца.

```
img_gray_naive = img_t.mean(-3)
batch_gray_naive = batch_t.mean(-3)
img_gray_naive.shape, batch_gray_naive.shape  # (torch.Size([5, 5]), torch.Size([2, 5, 5]))
```

PyTorch автоматически добавляет в начало измерение размером 1. Эта функция называется *транслированием* (broadcasting). `batch_t` формы $(2, 3, 5, 5)$ умножается на `unsqueeze_weights` формы $(3, 1, 1)$, в результате чего получается тензор формы $(2, 3, 5, 5)$, в котором затем можно сложить третье измерение с конца (три канала).

```
weights = torch.tensor([0.2126, 0.7152, 0.0722])  # torch.Size([3])
unsqueezed_weights = weights.unsqueeze(-1).unsqueeze_(-1)  # torch.Size([3, 1, 1])
img_weights = (img_t * unsqueezed_weights)
batch_weights = (batch_t * unsqueezed_weights)
```

В PyTorch 1.3 добавилась экспериментальная возможность *именновых тензоров*. У функций создания тензоров, например `tensor` и `rand`, есть аргумент `names`. В качестве аргумента `names` должна передаваться последовательность строковых значений

```
weights_named = torch.tensor([0.2126, 0.7152, 0.0722], names=["channels"])
```

При необходимости добавить названия в имеющийся тензор (не меняя существующие) можно вызвать его метод `refine_names`. Аналогично доступу по индексу с помощью многоточия можно пропускать любое количество измерений. С помощью родственного ему метода `rename` можно также переопределять или удалять (путем передачи `None`) уже существующие названия

```
img_named = img_t.refine_names(..., "channels", "rows", "columns")
```

Метод `align_as` возвращает тензор, в котором добавлены недостающие измерения, а уже существующие переставлены в нужном порядке [1, стр. 89]

```
weights_named.shape  # torch.Size([3])
img_named.shape  # torch.Size([3, 5, 5])
weights_aligned = weights_named.align_as(img_named)
weights_aligned.shape  # torch.Size([3, 1, 1])
```

Функции, принимающие на входе аргументы для измерений, также позволяют указывать поименованные измерения

```
(img_named * weights_aligned).sum("channels")
# то же самое
(img_named * weights_aligned).sum(0)
```

При попытке сочетать измерения с различными названиями выдается сообщение об ошибке

```
# img_named[..., :3] то же самое, что и img_named[:, :, :3]
gray_named = (img_named[..., :3] * weights_named).sum("channels")  # Ошибка, т.к. размерности не
    совпадают
# а так можно
(img_named[:, :, :3] * weights_named).sum("channels")
```

При необходимости использовать тензоры не только в функциях, работающих с поименованными тензорами, необходимо удалить названия, установив их в `None`

```
img_named.rename(None)
```

2.1. Типы элементов тензоров

Использовать стандартные типы данных Python не рекомендуется по нескольким причинам [1, стр. 90]:

- *Числовые значения в Python являются объектами*. В то время как число с плавающей запятой требует для представления в компьютере только 32 бита, Python преобразует его в *полноценный объект Python* с подсчетом ссылок и т.д. Эта операция, которая называется *упаковой* (boxing), не является проблемой при хранении небольшого количества числовых значений, но выделять память для миллионов таких объектов – совершенно нерационально.
- *Списки в Python предназнены для хранения последовательных наборов объектов*. В них нет операций для быстрого вычисления скалярного произведения двух векторов или их суммирования. Кроме того, *списки Python* не оптимизируют размещение своего содержимого в памяти, поскольку представляют собой *наборы указателей на объекты Python* (любые, не только числовые значения) с доступом по индексу. Наконец, списки Python одномерны, и, хотя можно создавать списки списков, это тоже нерационально.
- *Интерпретатор Python работает медленно по сравнению с оптимизированным, скомпилированным кодом.*

Вычисления в нейронных сетях обычно производятся над 32-битными значениями с плавающей запятой. Более высокая точность, например 64-битные значения, обычно не повышает безошибочность модели, но требует больше памяти и вычислительного времени. Нативная поддержка типа данных с половинной точностью – 16-битных значений с плавающей запятой – в стандартных CPU обычно отсутствует, зато предоставляется современными GPU. При необходимости можно перейти на *половинную точность* для снижения объема занимаемой памяти нейросетевой модели без особого влияния на степень безошибочности [1, стр. 92].

Привести результат функции создания тензора к нужному типу с помощью соответствующего метода приведения типов, можно так

```
torch.zeros(10, 2).double()
torch.ones(10, 2).short()
```

Или с помощью более удобного метода `.to()`

```
torch.zeros(10, 2).to(torch.double)
torch.ones(10, 2).to(dtype=torch.short)
```

Память под значения в тензорах выделяется непрерывными фрагментами памяти под управлением экземпляров `torch.Storage`. Хранилище представляет собой одномерный массив числовых данных, то есть непрерывный фрагмент памяти, содержащий числа заданного типа, например `float` (32-битные значения, выражающие числа с плавающей запятой) или `int64` (64-битные значения, выражающие целые числа). Экземпляр класса `Tensor` PyTorch – это представление подобного экземпляра `Storage` с возможностью доступа к хранилищу по индексу через указание сдвига и шага по каждому измерению.

Хранилище *всегда* представляет собой *одномерный* массив вне зависимости от размерности каких-либо ссылающихся на него тензоров.

Методы, имена которых заканчиваются на символ «_», как в `zero_`, указывают, что метод работает с заменой на месте (in place), изменяя входный данные вместо того, чтобы создавать новый выходной тензор и возвращать его. Метод `zero_` обнуляет все элементы входного тензора.

Все методы, в конце названия которых нет символа подчеркивания, оставляют исходный тензор неизменным и вместо этого возвращают новый.

Для транспонирования двумерных тензоров используется метод `.t()`. Но в PyTorch транспорировать можно не только матрицы. Можно транспонировать многомерный массив, и для этого достаточно указать два измерения, по которым нужно произвести транспонирование (зеркально отражая форму шага)

```
somt_t = torch.ones(3, 4, 5)
transpose_t = some_t.transpose(0, 2)
some_t.shape  # torch.Size([3, 4, 5])
transpose_t.shape  # torch.Size([5, 4, 3])
```

Любой из тензоров PyTorch можно перенести на (один из) GPU системы для массово-параллельных быстрых вычислений.

Помимо `dtype`, класс `Tensor` предоставляет атрибут `device`, который описывает, где на компьютере размещаются данные тензора.

```
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device="cuda")
```

Вместо этого можно скопировать созданный в CPU тензор на GPU с помощью метода `to`

```
points_gpu = points.to(device="cuda")
```

При этом возвращается новый тензор с теми же числовыми данными, но хранящийся в *памяти GPU*, а не в обычной *оперативной памяти системы* [1, стр. 105].

Если на нашей машине более одного GPU, можно также указать, на каком именно GPU размещать тензор, передав отсчитываемый с нуля целочисленный номер GPU на машине, вот так

```
points_gpu = points.to(device="cuda:0")

# умножение выполняется на CPU
points = 2 * points
# умножение выполняется на GPU
points_gpu = 2 * points.to(device="cuda")
```

Отметим, что тензор `points_gpu` не передается обратно в CPU после вычисления результата. Вот что происходит в этой строке:

- Тензор `points` копируется в GPU.
- Выделяется память в GPU под новый тензор, в котором будет храниться результат умножения.
- Возвращается обращение к этому GPU-тензору.

Следовательно, если мы прибавим к результату константу

```
points_gpu = points_gpu + 4
```

операция сложения будет по-прежнему производиться в GPU и никакой информации в CPU передаваться не будет (если мы не будем выводить полученный тензор на экран или обращаться к нему). Для переноса тензора обратно в CPU необходимо указать в методе `.to()` аргумент `cpu`

```
points_gpu.to(device="cpu")
```

Можно также для получения того же результата воспользоваться сокращенными методами `.cpu()` и `.cuda()` вместо метода

```
points_gpu = points.cuda()
points_gpu = points.cuda(0)
points_gpu.cpu()
```

Стоит упомянуть, что с помощью метода `.to()` можно менять тип данных и их место размещения одновременно, указав в качестве аргументов `device` и `dtype`.

Тензоры PyTorch можно очень эффективно преобразовать в массивы NumPy и наоборот. Благодаря этому можно воспользоваться огромными объемами функциональности экосистемы Python, основанной на типах массивов NumPy. Подобная совместимость с массивами NumPy, не требующая копирования кода, возможна благодаря работы системы хранения с буферным протоколом Python.

В разреженных тензорах хранятся только ненулевые значения, а также информация об индексах.

Для сериализации обхектов-тензоров PyTorch использует «за кулисами» `pickle`, а также специализированный код сериализации для хранилища. Вот как можно сохранить наш тензор `points` в файл `ourpoints.t`

```
torch.save(points, "./data/p1ch3/ourpoints.t)
```

Либо можно передать файловый дескриптор файла вместо названия

```
with open("./data/p1ch3/ourpoints.t", mode="wb") as f:
    torch.save(points, f)
```

Загрузка тензора `points` обратно также выполняется одной строкой кода

```
points = torch.load("./data/p1ch3/ourpoints.t")
```

что эквивалентно

```
with open("./data/p1ch3/ourpoints.t", mode="rb") as f:
    points = torch.load(f)
```

И хотя подобным образом можно быстро сохранять тензоры, если нужно загрузить их только в PyTorch, сам по себе формат файла не отличается совместимостью: прочитать тензор с помощью какого-либо еще ПО, помимо PyTorch, не получится.

HDF5 – переносимый, широко поддерживаемый формат представления сериализованных многомерных массивов, организованный в виде вложенного ассоциативного массива типа «ключ–значение». Python поддерживает формат HDF5 благодаря библиотеке `h5py`, принимающей и возвращающей данные в виде массивов NumPy.

```
import h5py

f = h5py.File("./ourpoints.hdf5", "w")
dset = f.create_dataset("coords", data=points.numpy())
f.close()
```

Здесь `"coords"` – это ключ для файла в формате HDF5. В HDF5 интересна возможность индексации набора данных на диске и обращения только к нужным нам элементам.

```
f = h5py.File("./ourpoints.hdf5", "r")
dset = f["coords"]
torch.from_numpy(dset[-2:])
f.close()
```

2.2. Представление реальных данных с помощью тензоров

2.2.1. Работа с изображениями

Изображения представляются в виде набора скалярных значений расположенных на равномерной сетке с высотой и шириной (в пикселях), например, по одному скалярному значению на каждую точку сетки (пиксель) для изображения в оттенках серого или несколько скалярных значений на каждую точку сетки для представления различных цветов.

Отражающие значения для различных пикселей скаляры обычно кодируются 8-битными целыми числами, как в бытовых фотоаппаратах. В медицинских, научных и промышленных приложениях нередко встречается более высокая точность, например 12- или 16-битная, для расширения диапазона или повышения чувствительности в случаях, когда пиксель отражает информацию о физическом свойстве, например о плотности костной ткани, температуре или глубине.

Загрузить изображение можно так

```
import imageio

img_arr = imageio.imread("./bobby.jpg")
img_arr.shape  # (720, 1280, 3)
```

Модули PyTorch, работающие с изображениями, требуют от тензоров измерений $C \times H \times W$ (каналы, высота и ширина).

Для получения нужной нам схемы расположения можно воспользоваться методом `permute` тензора, указав в качестве параметров старые измерения для каждого из новых.

```
img = torch.from_numpy(img_arr)
out = img.permute(2, 0, 1)
```

Эта операция не копирует данные тензора, вместо этого `out` *использует то же самое хранилище*, что и `img`, только меняя информацию о размере и шаге на уровне тензора.

Несколько более эффективная альтернатива использованию для создания тензора `stack` – выделить заранее память под тензор нужного размера, а затем заполнить его загруженными из каталога изображениями следующим образом

```
batch_size = 3
batch = torch.zeros(batch_size, 3, 256, 256, dtype=torch.uint8)
```

Батч будет состоять из трех RGB-изображений по 256 пикселей высотой и 256 пикселей шириной.

Нейронные сети демонстрируют наилучшее качество обучения, когда входные данные находятся в диапазоне примерно от 0 до 1 или от -1 до 1.

Никаких принципиальных различий между тензорами, содержащими объемные пространственные данные и данные изображения, нет. Просто появляется дополнительное измерение, глубина, вслед за измерением каналов, и получается 5-мерный тензор формы $N \times C \times D \times H \times W$.

Загрузим пример КТ-снимка с помощью функции `volread` из модуля `imageio`, принимающий в качестве аргумента каталог и собирающей все файлы в формате DICOM (Digital Imaging and Communications in Medicine) в трехмерный массив NumPy

```
import imageio

dir_path = ".../2-LUNG 3.0B70f-04083"
vol_arr = imageo.volread(dir_path, "DICOM")
vol_arr.shape  # (99, 512, 512)
```

```
vol = torch.from_numpy(vol_arr).float()
vol = torch.unsqueeze(vol, 0)
vol.shape  # torch.Size([1, 99, 512, 512]): каналы, глубина, высота, ширина
```

При вызове метода `.view()` на тензоре возвращает новый тензор с другими размерностью и шагами *без изменения хранилища*. Это позволяет перегруппировать тензор прктически без затрат, поскольку *никакие данные копировать не нужно*.

Нормализацию данных к отрезку $[0; 1]$ или $[-1; 1]$ желательно производить *для всех количественных величин*, таких как «температура» (это полезно для процесса обучения) [1, стр. 137].

По завершении обучения алгоритм способен генерировать правильные выходные сигналы при получении новых данных, *достаточно схожих* со входными данными, на которых он обучался. В случае глубокого обучения этот процесс работает даже тогда, когда входные данные и требуемые выходные сгиналы *далеки* друг от друга: когда они относятся к различным предметным областям.

У нас есть модель с неизвестными значениями параметров и нужно получить оценку этих параметров, которая бы минимизировала расхождение между предсказанными выходными сигналами и измеренными значениями (ошибка). Целью процесса оптимизации должен быть поиск таких параметров модели, которые минимизировали бы функцию потерь.

Отдельная итерация обучения, во время которой обновляются параметры для всех обучающих примеров данных, называется *эпохой*.

Градиенты по параметрам модели должны быть одного порядка [1, стр. 168].

Аргумент `requires_grad` указывает PyTorch отслеживать целое семейство тензоров. Если функции дифференцируемые (как большинство операций над тензорами PyTorch), величина производной будет автоматически занесена в атрибут `.grad`.

Количество тензоров с параметром `requires_grad`, установленным в `True` аргументом, и композиции функций может быть любым. В этом случае PyTorch вычисляет производные функции потерь по всей цепочке функций (графу вычислений) и накапливает их значения в атрибутах `.grad` этих тензоров (узлы этого графа).

При вызове `.backward()` производные *накапливаются* в узлах-листьях. Необходимо *явным образом обнулять градиенты* после обновления параметров на их основе. Так что, если `backward` вызывался ранее, потери оцениваются опять, `backward` вызывается снова, после чего накапливаются градиенты во всех листьях графа, то есть суммируются с вычисленными на предыдущей итерации, в результате чего неправильное значение градиента [1, стр. 173].

Чтобы предотвратить подобное, необходимо *явным образом обнулять градиенты* на каждой итерации.

```
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad. is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()  # выполяем обратный проход и вычисляем градиенты

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print(...)
```

При вызове `loss.backward()` PyTorch обходит граф в обратном порядке, вычисляя градиенты. Контекст `torch.no_grad()` означает, что механизм автоматического вычисления градиента игнорирует внутренности блока `with`: то есть не добавляет ребра в граф прямого прохода.

У каждого оптимизатора доступны два метода: `zero_grad` и `step`. Метод `zero_grad` обнуляет атрибут `grad` всех передаваемых оптимизатору параметров при его создании. А метод `step` обновляет значения параметров в соответствии с реализуемой конкретным оптимизатором стратегией оптимизации.

```
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

        # обнуляем градиент, потому как в противном случае
        # производные накапливались бы в узлах-листьях графа вычислений
        optimizer.zero_grad()
        loss.backward()  # обратный проход по графу; вычисляем градиенты
        optimizer.step()  # обновляем параметры модели

        if epoch % 500 == 0:
            print(...)

    return params
```

Очень гибкая модель с большим количеством параметров стремится к минимизации функции потерь в точках данных и нет никаких гарантий, что она будет себя вести нужным образом *вдали* или *между* точками данных [1, стр. 180].

Потери на обучающем наборе данных показывают, можно ли вообще подгнать нашу модель к этому обучающему набору данных - другими словами, достаточны ли *разрешающие возможности* (capacity) этой модели для обработки содержащейся в данных информации [1, стр. 182].

Глубокая нейронная сеть потенциально может аппроксимировать очень сложные функции при условии достаточно большого числа нейронов, а значит, и параметров. Чем меньше параметров, тем проще должна быть форма функции, чтобы наша сеть смогла ее аппроксимировать. Итак, правило 1: если потери на обучающем наборе данных не уменьшаются, вероятно, модель слишком проста *для имеющихся данных*.

Что ж, если вычисленная на проверочном наборе данных функция потерь не убывает вместе с обучающим набором, значит, наша модель обучается лучше аппроксимировать полученные во время обучения примеры данных, но не *обобщается* на примеры данных, которые не входят в этот конкретный набор. Правило 2: если потери на обучающем и проверочном наборах данных расходятся – модель переобучена.

С интуитвной точки зрения более простая модель может описывать обучающие данные не так хорошо, как более сложная, но зато, вероятно, будет вести себя более равномерым образом между точами данных.

Следовательно, процесс выбора правильного размера нейросетевой модели в смысле количества параметров основан на двух шагах:

- увеличение размера до тех пор, пока модель не будет хорошо подогнана к данным,
- а затем уменьшение, пока не будет устранено переобучение.

Разбиение набора данных. Перетасовка элементов тензора эквивалентна перестановке его индексов – как раз то, что делает функция `randperm`

```
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)

shuffled_indicies = torch.randperm(n_samples)
train_indicies = shuffled_indicies[:-n_val]
test_indicies = shuffled_indicies[-n_val:]

train_t_u = t_u[train_indices]
train_t_c = t_c[train_indices]

val_t_u = t_u[val_indicies]
val_t_c = t_c[val_indicies]
```

Осталось только дополнительно вычислять потери на проверочном наборе данных *на каждой эпохе*, чтобы заметить переобучение

```
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u, train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        val_t_p = model(val_t_u, *params)
        val_loss = loss_fn(val_t_p, val_t_c)

        optimizer.zero_grad()
        # здесь только train_loss.backward(), поскольку мы не хотим обучать
        # модель на проверочном наборе данных
        train_loss.backward()
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
            print(...)

    return params
```

Запуск

```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    params = params,
    train_t_u = train_t_un,
    val_t_u = val_t_un,
    train_t_c = train_t_c,
    val_t_c = val_t_c,
)
```

Наша цель – убедиться, что убывают как потери на обучающем наборе данных, *так* и потери на проверочном [1, стр. 186].

Вопрос: раз мы никогда не вызываем `backward()` для `val_loss`, зачем вообще формировать граф вычислений? Можно просто вызывать `model` и `loss_fn` как обычные функции, без отслеживания истории вычислений.

```
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u, train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
```

```
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        with torch.no_grad():
            val_t_p = model(val_t_u, *params)
            val_loss = loss_fn(val_t_p, val_t_c)
            # Убеждаемся, что для нашего вывода аргумент requires_grad == False
            assert val_loss.requires_grad == False

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
```

Нейрон – по сути представляет собой линейное преобразование входного сигнала (например, умножение входного сигнала на какое-либо число (*вес*) и прибавление к нему константы *смещения*) с последующим применением фиксированной нелинейной функции (*функции активации*).

Функция активации играет две выжные роли [1, стр. 195]:

- Во внутренних частях модели благодаря функции активации возможны различные наклоны графика выходного сигнала в разных значениях – нечто, по определению *недоступное для линейной функции*. Искусно сочетая эти по-разному наклоненные участки для различных выходных сигналов, нейронные сети могут аппроксимировать любые функции.
- На последнем слое сети она локализует выходные сигналы предыдущей линейной операции в заданном интервале.

Нейрон – это просто линейная функция с последующей функцией активации.

Функции активации по определению [1, стр. 199]:

- *нелинейны* – сколько ни применяй преобразование вида $w \cdot x + b$ без функции активации, все равно полчится функция той же самой (аффинной линейной) формы. *Нелинейность* позволяет сети в целом аппроксимировать более сложные функции,
- *дифференцируемы*, что дает возможность вычисления градиентов. Точечные разрывы (Hardtanh, ReLU etc.), допустимы.

В отсутствие этих характеристик сеть либо превратиться обратно в линейную модель, либо с трудом будет поддаваться обучению.

Для функций активации справедливо следующее:

- Имеется по крайней мере один диапазон *чувствительности*, внутри которого нетривиальные изменения входного сигнала приводят к соответствующим нетривиальным изменениям выходного. Необходимо для обучения.
- У многих из них есть также диапазон *нечувствительности* (*насыщения*), в котором изменения входного сигнала практически не приводят к изменениям выходного.

В целом получается механизм, обладающий большими возможностями: при получении на входе различных данных в сети, составленной из *линейных* и *активационных блоков*:

- различные нейроны могут возвращать для одних входных сигналов результаты, относящиеся к различным диапазонам,
- соответствующие этим входным сигналам ошибки в основном влияют на нейроны, работающие в диапазоне чувствительности, а на остальные блоки процесс обучения практически не влияет.

В результате объединения множества линейных и активационных блоков параллельно и последовательно получается математический объект, способный аппроксимировать сложные функции. Различные сочетания нейронов реагируют в различных диапазонах на входные сигналы,

причем параметры этих блоков можно довольно легко оптимизировать посредством градиентного спуска, поскольку процесс обучения напоминает обучение линейной функции, вплоть до момента насыщения выходного сигнала.

В PyTorch есть отдельный подмодуль, посвященный нейронным сетям, – `torch.nn`. Он включает «кирпичики», необходимые для создания всех видов нейросетевых архитектур. В терминологии PyTorch эти «кирпичики» называются *модулями* (в других фреймворках подобные стандартные блоки часто называются *слоями* (layers)). Модуль PyTorch – это класс Python, наследующий базовый класс `nn.Module`.

Подмодули должны быть атрибутами верхнего уровня, а не быть закопаны внутри экземпляров list или dict! В противном случае оптимизатор не сможет их найти (а значит, и их параметры). На случай, если модели потребуется список или ассоциативный массив подмодулей, в PyTorch есть классы `nn.ModuleList` и `nn.ModuleDict`. У `nn.Module` есть подкласс `nn.Linear`, применяющий ко входным сигналам аффинное преобразование.

У всех подклассов `nn.Module` в PyTorch есть метод `__call__`, позволяющий создавать экземпляры `nn.Linear` и вызывать их как функции следующим образом

```
import torch.nn as nn

linear_model = nn.Linear(1, 1)
linear_model(t_un_val)
```

Вызов экземпляра `nn.Module` с набором инструментов приводит к вызову метода `forward` с теми же аргументами, который реализует *прямой проход* вычислений, в то время как `__call__` выполняет другие немаловажные операции до и после вызова `forward`. Так что формально можно вызвать `forward` напрямую, и он вернет тот же результат, что и `__call__`, но делать это из пользовательского кода не рекомендуется

```
y = model(x)  # Правильно!
y = model.forward(x)  # НЕправильно! Так делать не надо!!!
```

Конструктор `nn.Linear` принимает три аргумента: число входных принаков, число выходных признаков и булево значение, указывающее, включает линейная модель смещение или нет

```
import torch.nn as nn

linear_model = mm.Linear(1, 1)
linear_model(t_un_val)
```

Все модули в `nn` ориентированы на генерацию выходных сигналов сразу для батча из нескольких входных сигналов. Следовательно, если нам нужно выполнить `nn.Linear` для десяти примеров данных, можно создать входной тензор размеров $B \times N_{\text{вх}}$, где B – размер батча, а $N_{\text{вх}}$ – число входных признаков, и пропустить его один раз через модель.

Причины для организации данных по батчам многогранны. Одна из них – желание полноценно загрузить вычислениями имеющиеся вычислительные ресурсы. В частности, GPU позволяют сильно распараллеливать вычисления, так что при одиночном входном сигнале для маленькой модели большинство вычислительных элементов будет простаивать.

Еще одно преимущество в том, что некоторые развитые модели способны использовать статистическую информацию по целому батчу, и эти статистические показатели будут точнее при большом размере батча.

```
linear_model = nn.Linear(1, 1)
```

```
optimizer = optim.SGD(
    linear_model.parameters(),
    lr=1e-02,
)
```

При вызове метода `training_loss.backward()` в листьях графа вычислений накапливаются градиенты. При вызове `opimizer.step()` программа проходит по всем объектам `Parameter` и меняет их на соответствующую содержимому атрибута `grad` долю.

```
def training_loop(
    n_epochs,
    optimizer,
    model,
    loss_fn,
    t_u_train,
    t_u_val,
    t_c_train,
    t_c_val,
):
    for epoch in range(1, n_epochs + 1):
        t_p_train = model(t_u_train)
        loss_train = loss_fn(t_p_train, t_c_train)

        t_p_val = model(t_u_val)
        loss_val = loss_fn(t_p_val, t_c_val)

        # требуется обязательно обнулять градиент на каждой итерации
        optimizer.zero_grad()
        # выполняем проход в обратном направлении и вычисляем градиенты
        loss_train.backward()
        # обновляем параметры модели
        optimizer.step()
```

Модуль `nn` включает несколько распространенных функций потерь, одна из которых – `nn.MSELoss`.

Модуль `nn` предоставляет удобный способ соединения модулей цепочкой с помощью контейнера `nn.Sequential`

```
seq_model = nn.Sequential(
    nn.Linear(1, 13),
    nn.Tanh(),
    nn.Linear(13, 1)
)
```

Модель переходит от одного входного признака до 13 скрытых признаков, пропускает их через функцию активации `Tanh` и, наконец, объединяет получившиеся 13 чисел в один выходной признак.

Названия модулей в `Sequential` представляют собой просто порядковые номера модулей в списке аргументов. Что любопытно, `Sequential` также принимает на входе `OrderDict`, в котором можно указать название каждого из передаваемых `Sequential` модулей

```
from collections import OrderDict

seq_model = nn.Sequential(OrderDict([
    ("hidden_linear", nn.Linear(1, 8)),
    ("hidden_activation", nn.Tanh()),
    ("output_linear", nn.Linear(8, 1))
]))
```

```
for name, param in seq_model.named_parameters():
    print(name, param.shape)
# output
hidden_linear.weight torch.Size([8, 1])
hidden_linear.bias torch.Size([8])
output_linear.weight torch.Size([1, 8])
output_linear.bias torch.Size([1])
```

Обращаться к конкретным объектам `Parameter` можно путем указания подмодулей в качестве атрибутов

```
seq_model.output_linear.bias
# output
Parameter containing:
tensor([0.1402], requires_grad=True)
```

Можно посмотреть *градиенты параметра* `weight` линейной части скрытого слоя. Запускаем цикл обучения для новой модели нейронной сети, после чего смотрим на получившиеся градиенты после последней эпохи

```
seq_model.hidden_linear.weight.grad
```

Функции активации, в дополнение к линейным преобразованиям, позволяют нейронным сетям аппроксимировать сильно нелинейные функции, оставляя их при этом достаточно простыми для оптимизации [1, стр. 215].

Список литературы

1. *Стивенс Э.* PyTorch. Освещая глубокое обучение. – СПб.: Питер, 2022. – 576 с.