Дата последней сборки документа:
13 марта 2024 г. в 01:47

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].

Чтобы преобразовать изображение PIL в тензор PyTorch, можно воспользоваться модулем `torchvision.transforms`. Есть преобразование `ToTensor`, превращающее массивы NumPy и изображения PIL в тензоры. Оно также располагает измерения выходного тензора в порядке $C \times H \times W$ (каналы, высота, ширина).

```
from torchvision import transforms, datasets

cifar10_train = datasets.CIFAR10(data_path, train=True, download=True)
```

```
img, label = ciraf10_train[99]

to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
img_t.shape  # torch.Size([3, 32, 32])
```

Изображение `img` превратилось в тензор формы $3 \times 32 \times 32$, то есть в изображение размером 32×32 с тремя цветовыми каналами (RGB).

Это преобразование можно передать непосредственно в виде аргумента `datasets.CIFAR10`

```
tensor_cifar10 = datasets.CIFAR10(
    data_path,
    train=True,
    download=False,
    transform=transforms.ToTensor(),
)
```

Рекомендуемая практика – нормализовать набор данных до нулевого среднего значения и единичного стандартного отклонения по каждому из каналов.

При выборе функций активации, линейных около нуля (± 1 или ± 2), ограничение данных тем же диапазоном повышает вероятность *ненулевых градиентов нейронов*, а значит, и *ускоряет обучение*. Кроме того, нормализация каналов к одинаковому распределению гарантирует смешение и обновление информации из разных каналов (посредством градиентного спуска) с одинаковой скоростью обучения.

Чтобы обеспечить нулевое среднее и единичное стандартное отклонение по каждому из каналов, необходимо вычислить среднее значение и стандартное отклонение каждого из каналов набора данных и применить следующее преобразование `v_n[c] = (v[c] - mean[c]) / stdev[c]`.

Поскольку набор данных CIFAR-10 невелик, можно работать с ним полностью в оперативной памяти. Разместим все возвращаемые объектом `Dataset` тензоры последовательно в дополнительном измерении

```
imgs = torch.stack([img_t for img_t, _ in tensor_cifar_10], dim=3)
imgs.shape  # torch.Size([3, 32, 32, 50 000])
```

Теперь можно легко вычислить поканальные средние значения

```
imgs.view(3, -1)  # torch.Size([3, 51200000])
imgs.view(3, -1).mean(dim=1)
imgs.view(3, -1).std(dim=1)
```

Теперь

```
transformed_cifar10 = datasets.CIFAR10(
    data_path,
    train=True,
    download=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforsm.Normalize(
            mean=imgs.view(3, -1).mean(dim=1),
            std=imgs.view(3, -1).std(dim=1),
        )
    ])
)
```

Сколько признаков содержит каждый пример данных? Так, $3 \times 32 \times 32$ равняется 3072 входных признака на каждый пример. Получаем новую модель `nn.Linear` с 3072 входными признаками и некоторым количеством скрытых принаков, за которым следует функция активации, а затем еще один `nn.Linear`, сокращающий модель до соответствующего количества выходных признаков (в данном случае 2)

```
import torch.nn as nn

n_out = 2
model = nn.Sequential(
    nn.Linear(
        3072,  # входные признаки
        512,  # размер скрытого слоя
    ),
    nn.Tanh(),
    nn.Linear(
        512,  # размер скрытого слоя
        n_out,  # выходные признаки
    )
)
```

Нейронной сети требуется по крайней мере один скрытый слой (активации, поэтому два модуля) с *нелинейностью* между слоями, чтобы сеть *могла усваивать произвольные функции*, в противном случае модуль будет просто *линейной*.

Здесь необходимо понять, что выходной сигнал носит категориальный характер: птица или самолет. Для представления категориальной величины следует воспользоваться унитарным кодированием, например, `[1, 0]` для самолета и `[0, 1]` для птицы (порядок выбран произвольно). Такая схема подходи и в случае 10 классов, как в полном наборе данных CIFAR-10; просто вектор будет длиной 10.

В нашем частном случае бинарной классификации (птица или самолет; по сути птица или не птица) два значения – избыточно, поскольку одно всегда равно 1 минус второе. И действительно, PyTorch позволяет выдавать на выходе одно значение *вероятности*, получая вероятность путем использования в конце модели функции активации `nn.Sigmoid` и функции потерь на основе *бинарной перекрестной энтропии* `nn.BCELoss`. Существует также `nn.BCELossWithLogits`, объединяющая эти два шага [1, стр. 229].

В идеальном случае сеть должна выдавать на выходе `torch.tensor([1.0, 0.0])` для самолета и `torch.tensor([0.0, 1.0])` – для птицы. На практике же, поскольку наш класификатор не будет идеален, следует ожидать от сети неких промежуточных значений. Главное в этом случае, что мы можем интерпретировать выходные сигналы как вероятности: первая запись – вероятность класса `"airplane"`, а вторая – `"bird"`.

Чтобы развернуть изображение формы $3 \times 32 \times 32$, а затем преобразовать его в вектор-строку делаем так

```
img_batch = img.view(-1).unsqueeze(0)
img_batch.shape  # torch.Size([1, 3072]); 3 * 32 * 32 = 3072
```

Что касается функции потерь для задач классификации, то здесь необходимо накладывать штраф на ошибки классификации, а не кропотливо штрафовать все, что не равно в точности 0.0 или 1.0, поэтому в задачах классификации плохо работает среднеквадратическая функция потерь.

В этом случае *необходимо максимизировать вероятность*, соответсвующую истинному классу.

Другими словами, нам нужна функция потерь, принимающая очень высокие значения, когда правдоподобие (истинность параметров нашей модели при имеющихся данных) низко: настолько низко, что вероятности альтернативных вариантов выше. И наоборот, потери должны быть низкими, когда правдоподобие данного варианта выше, чем у альтернатив, и мы не хотим зацикливаться на доведении вероятности до 1 [1, стр. 234].

Действующая подобным образом функция потерь существует и называется *отрицательной логарифмической функцией правдоподобия* (negative log likelihood, NNL).

Для каждого примера данных в батче мы делаем следующее:

1. Производим прямой проход и получаем выходные значения из последнего (линейного) слоя.
2. Вычисляем для них многомерную логистическую функцию и получаем вероятности.
3. Извлекаем предсказанную вероятность для истинного класса (правдоподобие параметров). Отметим, что истинный класс известен, поскольку обучение производится с учителем, – это наши эталнные данные.
4. Вычисляем ее логарифм, ставим перед ним знак «минус» и прибавляем к потерям.

Функция `nn.LogSoftmax()` обеспечивает численную устойчивость вычислений

```
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1),
)
```

Среднеквадратическая функция потерь (MSE) насыщается намного раньше и – что принципиально – для совершенно неправильных предсказаний. Основная причина состоит в том, что уклон MSE слишком мал, чтобы компенсировать пологость многомерной логистической функции активации для неправильных предстказаний. Поэтому MSE для вероятностей плохо подходит для задач классификации [1, стр. 237].

```
import torch
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1),
)

learning_rate = 1e-2

optimizer = optim.SGD(model.parameters(), lr=learning_rate)

n_epochs = 100

for epoch in n_epochs:
    for img, label in cifar2:
        out = model(img.view(-1).unsqueeze(0))
        loss = loss_fn(out, torch.tensor([label]))

        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()

    print(...)
```

Мы поняли, что обработка всех 10 000 изображений одним батчем – это перебор, так что решили создать внутренний цикл, чтобы *обрабатывать по одному примеру данных за раз* и производить *обратное распространение ошибки по этому одному примеру.*

Перетасовывая примеры данных *на каждой эпохе* и вычисляя градиент по одному или (что желательно из соображений устойчивости) нескольким примерам данных за раз, мы фактически вносим элемент случайности в алгоритм градиентного спуска. Оказывается, что следование *градиентам, вычисленным по мини-батчам*, которые представляют собой лишь слабые аппроксимации градиентов, вычисленных по всему набору данных, *улучшает сходимость* и *предотвращает «застревание»* процесса оптимизации во встреченных по пути локальных минимумах [1, стр. 238].

Обычно размер мини-батча представляет собой константу, задаваему до обучения, аналогично скорости обучения.

Модуль `torch.utils.data` включает класс, помогающий с перетасовкой и организацией данных по мини-батчам: `Dataloader`. Задача загрузчика данных состоит в выборе мини-батчей из набора данных с гибкими возможностями использования различных стратегий выборки. Одна из самых распространенных стратегий: равномерная выборка после *перетасовки* данных *в каждой эпохе.*

```
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=True)
```

Пример

```
import torch
import torch.nn as nn

train_loader = torch.utils.data.DataLoader(
    cifar2, batch_size=64,
    shuffle=True,
)

model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2)
    nn.LogSoftmax(dim=1)
)

learning_rate = 1e-2

optimiizer = optim.SGD(model.parameters(), lr=learning_rate)

# отрицательный логарифм правдоподобия
loss_fn = nn.NLLLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
```

```
        loss = loss_fn(outputs, labels)

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

На каждой итерации внутреннго цикла `imgs` представляет собой мини-батч 64 RGB-изображений (размером 32×32), а `labels` – тензор размером 64 с индексами меток.

Наша цель в том, чтобы правильно присвоить изображениям метки классов, причем желательно на независимом наборе данных

```
val_loader = torch.utils.data.DataLoader(
    cifar2_val,
    batch_size=64,
    shuffle=False,  # NB
)

correct = 0
total = 0

with torch.no_grad():  # NB!
    for imgs, labels in val_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        _, predicted = torch.max(outputs, dim=1)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())

print(correct / total)
```

Достаточно часто последний слой `nn.LogSoftmax` не включают в сеть, используя в качестве *функции потерь* `nn.CrossEntoryLoss`.

Полносвязная сеть не является *инвариантной относительно сдвига*. Это значит, что сеть, обученная распознавать «Спитфайер» (самолет), начинающийся с позиции 4,4, не сможет распознать *тот же самый* «Спитфайен», начинающийся с позиции 8,8. Нам пришлось бы дополнять (augment) набор данных, то есть применять случайные сдвиги к изображениям во время обучения, чтобы сеть могла заметить «Спитфайер» в любом месте изображения, причем это пришлось бы делать для всех изображений в наборе [1, стр. 246].

Учтите, что *весовые коэффициенты ядра заранее не известны*, а инициализируются *случайным* образом и обновляются посредством обратного распространения ошибки. Отметим также, что для всего изображения используется одно и то же ядро, а это значит, что и весовые коэффициенты ядра [1, стр. 252]. В производную функции потерь по свертрочным весам вности свой вклад все изображение.

Свертка эквивалентна нескольким линейным операциям, весовые коэффициенты которых равны нулю практически везде, кроме окрестностей отдельных пикселей, и получают одинаковые обновления во время обучения.

Очень часто применяют ядра, размеры которых одинаковы по всем измерениям, поэтому в PyTorch есть сокращенная форма записи для них: `kernel_size=3` для двумерной свертки означает форму 3×3, для трехмерной свертки – форму $3 \times 3 \times 3$.

```
conv = nn.Conv2d(3, 16, kernel_size=3)
# или так
conv = nn.Conv2d(3, 16, kernel_size=(3, 3))
```

Здесь 3 признака на один пиксель (3 канала; RGB) и 16 выходных каналов.

В результате прохода двумерной свертки получается двумерное изображение, пиксели которого представляют собой взвешенную сумму значений по локальным окрестностям входного изображения.

Как начальные значения весовых коэффициентов ядра `conv.weight`, так и смещения задаются случайным образом, так что выходное изображение особого смысла не имеет.

Лучше предерживаться ядер нечентных размеров; ядра четного размера встречаются редко [1, стр. 256].

Задача сверточной сети состоит в оценке ядра набора фильтров в последовательных слоях, преобразующих многоканальное изображение в другое многоканальное изображение, в котором различные каналы соответствуют разным признакам (например, один канал – для среднего значения, другой – для вертикальных краев и т.д.).

Первый набор ядер работает с маленькими окрестностями низкоуровневых признаков первого порядка, а второй набор фактически работает с более широкими окрестностями, *генерируя признаки, представляющие собой композицию предыдущих признаков*. Благодаря этому замечательному механизму сверточные нейронные сети способны анализировать очень сложные кадры.

```
# свертка + нелинейность + пулинг
model = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1),  # свертка
    nn.Tanh(),  # нелинейность
    nn.MaxPool2d(2),  # пулинг
    nn.Conv2d(16, 8, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    ...
)
```

Первая операция свертки превращает три канала RGB в 16, благодаря чему у сети появляется возможность генерировать *16 независимых признаков* (16 каналов). Далее мы применяем функцию активации `Tanh`. Полученное 16-канальное изображение 32×32 субдискритизируется первым слоем `nn.MaxPool2d` до 16-канального изображения 16×16.

Теперь субдискретизированное изображение подвергается еще одной операции свертки, выдающей на выходе 8-канальный выходной сигнал 16×16. Если повезет, это выходное изображение будет состоять из высокоуровневых признаков. И опять же мы применяем функцию активации `Tanh`, после чего производим субдискретизацию до 8-канального выходного изображения 8×8.

После уменьшения входного изображения до набора 8×8 признаков можно надеяться вернуть из сети значения вероятностей, подходящих для подачи на вход отрицательной логарифмической функции правдоподобия. Нужно преобразовать 8-канальное изображение 8×8 в одномерный вектор и завершить нашу сеть набором полносвязаных слоев

```
model = nn.Sequential(
    nn,Conv2d(3, 16, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    nn.Conv2d(16, 8, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    ...  # Пропущен важный момент
    nn.Linear(8 * 8 * 8, 32),
    nn.Tanh(),
    nn.Linear(32, 2),
```

```
)
```

Для повышения *разрешающих возможностей модели* можно *увеличить количество выходных каналов* сверточных слоев (то есть число *признаков*, генерируемых каждым из сверточных слоев), в результате чего увеличится и размер линейного слоя.

PyTorch позволяет производить в модели любые вычисления путем создания подклассов `nn.Module`.

Готовые или пользовательские свертки как подмодули обычно включаются в программу посредством описания в конструкторе `__init__` и присваивания их `self` для использования в функции `forward`. Их параметры в то же время храняться в них на протяжении всего жизненного цикла нашего модуля. Обратите внимание, что перед этим необходимо вызывать `super().__init__()`.

3. Обобщения с помощью сверток

3.1. Сеть как подкласс `nn.Module`

Задача классификационных сетей обычно заключается в сжатии информации в том смысле, что мы начинаем с изображения, содержащего значительное количество пикселей, и сжимаем его в вектор вероятностей классов.

Для создания подкласса `nn.Module` как минимум необходимо описать функцию `forward`, принимающую входные сигналы модуля и возвращающую выходной.

Подмодули должны быть атрибутами верхнего уровня, а не быть «закопаны» внутри экземпляров list или dict! В противном случае оптимизатор не сможет их (а значит, и их параметры) найти. На случай, если модели потребуется список или ассоциативный массив подмодулей, в PyTorch есть классы `nn.ModuleList` и `nn.ModuleDict` [1, стр. 269].

3.2. Функциональные API

В PyTorch есть *функциональные* аналоги для всех модулей `nn`. Под функциональными здесь подразумевается «без внутреннего состояния» – другими словами, «выходное значение которых целиком и полностью определяется значениями входных аргументов».

```
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.fc2 = nn.Linear(32, 2)

   def forward(self, x):
       out = F.max_pool2d(  # пулинг
           torch.tanh(  # нелинейность
               self.conv1(x)  # свертка
           ), 2)
       out = F.max_pool2d(
           torch.tanh(
               self.conv2(x)
           ), 2)
       out = out.view(-1, 8 * 8 * 8)
```

```
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)

        return out
```

3.3. Обучение модели

В основе сверточной сети лежат два вложенных цикла: внешний – по *эпохам*, а внутренний – на основе объекта `DataLoader`, генерирующего батчи из объекта `Dataset`. На каждой итерации цикла необходимо [1, стр. 273]:

- Пропустить входные сигналы через модель (прямой проход).
- Вычислить функцию потерь (также часть прямого прохода).
- Обнулить все старые градиенты.
- Вызвать `loss.backward()` для вычисления градиентов функции потерь относительно каждого из параметров (обратный проход).
- Оптимизировать в сторону уменьшения потерь.

```
import datetime

def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            # избавляемся от градиентов с предыдущих итераций
            optimizer.zero_grad()
            # выполняем обратный проход; то есть вычисляем градиенты
            # по всем обучаемым параметрам сети
            loss.backward()
            # обновляем модель
            optimizer.step()
            # суммируем потери за эпоху
            loss_train += loss.item()

     if epoch == 1 or epoch % 10 == 0:
         # получаем средние потери на батч
         print("{} Epoch {}, Training loss {}".format(
             datetime.datetime.now(), epoch,
             loss_train / len(train_loader))
         )
```

Обучение в течение 100 эпох

```
# Объект DataLoader организует примеры данных из нашего набора по батчам.
# Перетасовка обеспечивает случайный порядок примеров данных из набора
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=False)

model = Net()
optimizer = optim.SGD(model.parameters(), lr=1e-02)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=100,
    optimizer=optimizer,
```

```
    model=model,
    loss_fn=loss_fn,
    train_loader=train_loader,
)
```

Сохранить модель можно так

```
# сохраняютс только веса
torch.save(model.state_dict(), data_path + "birds_vs_airplanes.pt")
```

Файл `birds_vs_airplanes.pt` теперь содержит все параметры объекта `model`: весовые коэффициенты и смещения для двух модулей свертки и двух линейных модулей. Никакой структуры, только весовые коэффициенты. Это значит, что при развертывании модели в реальных условиях нам понадобится описание класса `model`

```
loaded_model = Net()
loaded_model.load_state_dict(torch.load(data_path + "birds_vs_airplanes.pt))
```

В `nn.Module` есть реализована функция `.to`, перемещающая все параметры в GPU (или приводящая тип данных, если передать ей аргумент `dtype`).

Между `Module.to` и `Tensor.to` существует тонкое различие. `Module.to` производит операции с заменой на месте, то есть изменяет экземпляр модуля. А `Tensor.to` – нет, возвращая *новый тензор*.

Рекомендуемой практикой является создание экземпляра `Optimizer` *после* перемещения всех параметров на нужное устройство [1, стр. 275].

Перенос вычислений на GPU при его наличии считается хорошим стилем программирования. Неплохим паттерном программирования будет установка значения переменной `device` в зависимости от `torch.cuda.is_avaible`:

```
device = (
    torch.device("cuda") if torch.cuda.is_avaible() else torch.device("cpu")
)
```

```
def training_loop(
    n_epochs, optimizer,
    model, loss_fn, train_loader
):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            imgs = imgs.to(device=device)  # NB
            labels = labels.to(device=device)  # NB
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)
```

Можно создать экземпляр модели и перенести ее на `device`

```
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=True)

# перенести модель (все ее параметры) на GPU
model = Net().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-02)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=100,
    optimizer=optimizer,
```

```
    model=model,
    loss_fn=loss_fn,
    train_loader=train_loader,
)
```

Если забыть перенести саму модель или входные данные на GPU, вы получите сообщения об ошибках, указывающие, что тензоры располагаются на различных устройствах, поскольку операторы PyTorch не поддерживают смеси входных данных на GPU и CPU.

PyTorch попытается загрузить веса на то же устройство, с которого они были сохранены, то есть весовые коэффициенты с GPU будут восстановлены на GPU. Лаконичным вариантом будет потребовать от PyTorch переопределить информацию об устройстве при загрузке весовых коэффициентов

```
loaded_model = Net().to(device=device)
loaded_model.load_state_dict(
    torch.load(
        data_path + "birds_vs_airplanes.pt",
        map_location=device
    )
)
```

3.4. Ширина сети

Ширина сети это количество нейронов в слое или каналов на каждую операцию свертки. Расширить модель в PyTorch очень легко. Необходимо просто указать большее количество выходных каналов в первой свертке и увеличивать следующие слои соответствующим образом, не забывая менять функцию `forward` так, чтобы отразить увеличивщуюся длину вектора при переходе на полносвязные слои.

Чем больше разрешающих возможностей модели, тем с большей степенью изменчивости входных сигналов сможет справиться модель, но в то же время тем выше вероятность переобучения, поскольку модель сможет воспользоваться дополнительными параметрами для запоминания несущественных аспектов входных данных.

3.5. Регуляризация

Основные способы регуляризации:

- штрафы на весовые коэффициенты (L_1, L_2-норма),
- дропаут,
- нормализация по батчам (альтернатива дропауту).

Первый способ достижения устойчивости обобщения: добавления члена регуляризации в формулу потерь. Этот дополнительный член ограничивает рост весовых коэффициентов модели в процессе обучения: он устроен так, что они стремятся оставаться маленькими. Другими словами, он налагает штраф на большие значения весов. В результате форма функции потерь становится более гладкой, и для модели нет особого смысла подстраиваться под отдельные примеры данных.

Наиболее популярные виды членов регуляризации: L_2-регуляризация и L_1-регуляризация. L_2-регуляризацию также называют *затуханием весов* (weight decay). Прибавление к функции потерь члена L_2-регуляризации эквивалентно уменьшению каждого весового коэффициента пропорционально его текущему значению во время шага оптимизации (отсюда и название «затухание

веса»). Обратите внимание, что затухание веса относится ко всем параметрам сети, в том числе и к смещениям.

```
def training_loop_l2reg(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            imgs = imgs.to(device)
            labels = labels.to(device=device)
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            l2_lambda = 0.001
            ls_norm = sum(p.pow(2.0).sum() for p in model.parameters())
            loss = loss + l2_lambda * l2_norm

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            loss_train += loss.item()
```

Впрочем, в оптимизаторе SGD в PyTorch уже есть параметр `weight_decay`, соответствующий `2 * lambda`, который напрямую осуществляет затухание весов во время их обновления. Он полностью эквивалентен прибавлению L_2-нормы весовых коэффициентов к функции потерь без необходимости накопления в функции потерь и вовлечения автоматического вычисления градиентов.

Идея дропаута действительно проста: обнуляем случайную часть выходных сигналов нейронов по сети, причем этот случайный выбор производится на каждой итерации обучения.

Фактически в результате этой процедуры на каждой итерации формируются слегка отличающиеся модели с различными топологиями нейронов, уменьшая шансы нейронов модели скоординироваться в процессе запоминания, что происходит при переобучении.

В PyTorch можно реализовать дропаут в модели с помощью добавления модуля `nn.Dropout` между нелинейной функцией активации и линейным или сверточным модулем последующего слоя. В качестве аргумента необходимо указать вероятность, с которой будут обнуляться входные сигналы.

```
class NetDropout(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv1_dropout = nn.Dropout2d(p=0.4)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)
        self.conv2_dropout = nn.Dropout2d(p=0.4)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = self.conv1_dropout(out)  # Dropout
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = self.conv2_dropout(out)  # Dropout
        out = self.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
```

```
        return out
```

Дропаут обычно происходит *во время обучения*, в то время как *во время использования* обученной модели в реальных условиях *модуль дропаута обходят* или, что эквивалентно, присваивают равную нулю вероятность. Этот процесс контролируется свойством `train` модуля `Dropout` [1, стр. 283].

PyTorch позволяет переключаться между двумя режимами, вызывая `model.train()` или `model.eval()` для любого подкласса `nn.Model`.

В статье Сергея Йоффе «Нормализация по батчам: ускорение обучения нейронных сетей путем сокращения внутреннего ковариантного сдвига» (Batch Normalization: Accelerating Deep Network Trainging by Reducing Internal Covariate Shift) описывается методика, которая позволяет *повысить скорость обучения* и *снизить зависимость обучения от начальных значений*, а также играет роль *регуляризатора*, тем самым представляя альтернативу дропауту [1, стр. 283].

Основная идея *нормализации по батчам* состоит в нормализации входных сигналов функций активации сети так, чтобы получить определенное желательное распределение для мини-батчей. Если вспомнить внутренние механизмы обучения и роль нелинейных функций активации, становится ясно, что это помогает *избежать чрезмерного углубления входных сигналов функций активации в область насыщения*, что гасит градиенты и замедляет обучение [1, стр. 283].

На практике нормализация по батчам сдвигает и масштабирует промежуточные входные сигналы на основе среднего значения и стандартного отклонения, вычисленых в этой промежуточной точке по примерам данных мини-батча. Эффект от регуляризации основан на том, что отдельные примеры данных и следующие далее по конвейеру фунцкии активации всегда рассматриваются моделью как сдвинутые и нормализованные, в зависимости от статистических показателей выделенного случайным образом мини-батча. Авторы статьи полагают, что нормализация по батчам исключает или по крайней мере сокращает необходимость в дропауте.

Поскольку цель нормализации по батчам – масштабировать входные сигналы функции активации, логично будет производить ее после линейного преобразования (свертки а данном случае), то есть перед функцией активации

```
class NetBatchNorm(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv1_batchnorm = nn.BatchNorm2d(num_features=n_chans1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)
        self.conv2_bachnorm = nn.BatchNorm2d(num_features=n_chans1 // 2)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = self.conv1_batchnorm(self.conv1(x))  # batch norm
        out = F.max_pool2d(torch.tanh(out))
        out = self.conv2_batchnorm(self.conv2(out))  # batch norm
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)

        return out
```

Как и дропаут, нормализация по батчам должна вести себя по-разному во время обучения и во время выполнения вывода. На самом деле *во время выполнения вывода* желательно, чтобы выходной сигнал для конкретного входного сигнала не зависел от прочих входных сигналов, подаваемых на вход модели. А это значит, что необходим способ нормализовать данные, но при этом *раз и навсегда зафиксировать параметры нормализации*.

При обработке мини-батчей, помимо оценки среднего значения и стандартного отклонения для текущего мини-батча, PyTorch также обновляет в качестве приближения *скользящие оценки среднего значения и стандартного отклонения*, репрезентативные для всего набора данных.

Таким образом, если пользователь указывает `model.eval()` и модель содержит модуль нормализации по батчам, скользящие оценки фиксируются и используются для нормализации. Для возврата к использованию статистических показателей мини-батчей мы вызываем `model.train()` точно так же, как для дропаута.

Глубина обеспечивает возможность работы сети с *иерархической информацией*, когда для анализа какого-либо входного сигнала необходимо понимать *конекст* [1, стр. 285].

Углубление сети обычно ухудшает сходимость. Производные функции потерь относительно параметров, особенно в первых слоях, приходится умножать на множество других чисел, генерируемых цепочкой операций между функцией потерь и параметром. Эти множители могут быть маленькими, приводя в результате к еще меньшим числам, или большими, поглощая маленькие числа из-за приближенности операций с плавающей запятой. В сухом остатке мы получаем, что в результате длинной цепочки операций умножение *вклад отдельного параметра в градиент исчезает*, и это ведет к *неэффективному обучению* данного слоя, поскольку ни этот, ни другие параметры не будут обновляться должным образом.

Добавление в модель обходной связи (skip connection) наподобие ResNet сводится к прибавлению выходного сигнала какого-то слоя к входному сигналу другого слоя

```
class ResNet(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(n_chans1 // 2, n_chans1 // 2, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(4 * 4 n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        out = F.max_pool2d(torch.relu(self.conv2(out)), 2)
        out1 = out
        out = F.max_pool2d(torch.relu(self.conv3(out)) + out1, 2)  # skip connection
        out = out.view(-1, 4 * 4 * self.n_chans1 // 2)
        out = self.fc2(out)

        return out
```

Обходная связь создает прямой путь от расположенных глубоко параметров к функции потерь, благодаря чему они вносят более непосредственный вклад в градиент функции потерь, ведь частные производные функции потерь по этим параметрам теперь получают шанс не умножаться на коэффициент в длинной цепочке прочих операций.

Отмечается, что обходные связи благотворно влияют на сходимость, особенно на начальных этапах обучения. Кроме того, поверхность функции потерь глубоких остаточных сетей намного глаже, чем у сетей прямого распространения той же глубины и ширины.

Применение обходных связей в ResNet дало возможность успешно обучать модели глубиной более 100 слоев.

`nn.Sequential` гарантирует, что выходной сигнал одного блока будет использован как входной сигнал следующего, а также что все параметры блока видимы `Net`.

```
class ResBlock(nn.Module):
    def __init__(self, n_chans):
        super().__init__()
        # Слой BatchNorm свел бы на нет эффект смещения, так что смещение обычно опускают
        self.conv = nn.Conv2d(n_chans, n_chans, kernel_size=3, padding=1, bias=False)
        self.batch_norm = nn.BatchNorm(n_num_features=n_chans)
        torch.nn.init.kaiming_normal_(self.conv.weight, nonlinearity="relu")
        torch.nn.init.constant_(self.batch_norm.weight, 0.5)
        torch.nn.init.zeros_(self.batch_norm.bias)

    def forward(self, x):
        out = self.conv(x)
        out = self.batch_norm(out)
        out = torch.relu(out)

        return out + x

class NetResDeep(nn.Module):
    def __init__(self, n_chans1=32, n_blocks=10):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.resblocks = nn.Sequential(
            *(n_blocks * [ResBlock(n_chans=n_chans1)]))
        self.fc1 = nn.Linear(8 * 8 * n_chans1, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        out = self.resblocks(out)
        out = F.max_pool2d(out, 2)
        out = out.view(-1, 8 * 8 * self.n_chans1)
        out = torch.relu(self.fc1(out))
        out = self.fc2(out)

        return out
```

У регуляризации весов и дропаута статистическая интерпретация в качестве методов регуляризации более строгая, чем нормализация по батчам. Нормализация по батчам предназначена скорее для *улучшения сходимости* [1, стр. 292].

4. Применение PyTorch в борьбе с раком

Компьютерная томография – это, по сути, трехмерные рентгеновские снимки, представленные в виде трехмерного массива одноканальных данных изображений.

Воксель – трехмерный эквивалент привычного двумерного пикселя. Он занимает некий объем пространства, а не плоскую область и обычно размещается в трехмерной сетке.

Каждый воксель КТ имеет числовое значение, которое примерно соответствует *средней массовой плотности вещества*, содержащегося в этой точке. На большинстве визуализаций подобных данных вещества высокой плотности, такие как кости и металлические импланты, отображаются белым, воздух и легочная ткань с малой плотностью – черной, а жир и ткани – различными оттенками серого.

Основное различие между компьютерной томографией и рентгеновскими снимками заключается в том, что ренгеновский снимок представляет собой проекцию трехмерной интенсивности (в данном случае плотности ткани и костей) на двумерную плоскость, а компьютерная томография сохраняет данные в третьем измерении.

Компьютерная томография фактически измеряет радиоплотность, которая является функцией как массовой плотности, так и атомного номера исследуемого вещества.

4.1. Файлы необработанных данных КТ

Данные КТ у нас представлены в двух видах файлов: файлах `.mhd`, содержащих метаданные заголовков, и файлах `.raw`, содержащих необработанные байты, в виде трехмерных массивов. Имя каждого файла начинается с уникального идентификатора, называемого UID (название происходит от номенклатуры цифровых изображений и коммуникаций в медицине – Digital Imaging and Communecations in Medicine, DICOM) для компьютерной томографии. Например, UID 1.2.3 соответствуют два файла: `1.2.3.mhd` и `1.2.3.raw`.

4.2. Обучающие и проверочные данные

Для любой стандартной задачи обучения с учителем данные деляться на обучающие и проверочные наборы. Оба должны быть *репрезентативны* для диапазона реальных входных данных, которые мы ожидаем получить и хотим обрабатывать. Если какой-либо из наборов существенно отличается от реальных вариантов использования, то вполне вероятно, что наша модель будет вести себя не так, как мы ожидаем, поскольку модель, обученная на далеких от реальности данных, не сможет работать нормально в полевых условиях!

4.3. Единицы Хаунсфилда

Воксели КТ выражены в *единицах Хаунсфилда* (HU), где воздух имеет значение -1000 HU, вода составляет 0 HU, а кость – не менее +1000 HU. Некоторые компьютерные томографы используют значения HU, соответствующие отрицательной плотности, указывая с их помощью на воксели, находящиеся за пределами поля зрения компьютерного томографа. Для наших целей все, что находится за пределами пациента, должно считаться воздухом, поэтому мы отбрасываем информацию о поле зрения, устанавливая нижнюю границу значений на уровне - 1000 HU. Аналогично точная плотность костей, металлических имплантов и так далее нас тоже не интересует, так что мы ограничиваем ее значением примерно 2 г/см^3 (1000 HU), хотя в большинстве случаев это биологически неточно.

Следует удалить из наших данных все выбивающиеся значения. Они не нужны для нашей цели, а их наличие может усложнить работу модели. Это усложнение может произойти по-разному, но чаще всего возникает ситуация, когда при пакетной нормализации выбивающейся значения искажают данные.

К сожалению, все данные центра кандидата, выражены в милимитрах, а не в вокселях! Нам нужно преобразовать наши координаты из миллимитровой системы координат (X, Y, Z), в которой они выражены, в систему координат, основанную на *адресах вокселей* (I, R, C), используемую для получения срезов массива из данных компьютерной томографии.

В *системе координат пациента* положительное значение X определяется как направление к левой стороне тела пациента (влево), положительное значение Y – как направление к спине (назад) и положительное значение Z – как направление к голове пациента (вверх). Эту систему иногда сокращенно называют LPS – left-posterior-superior, или «влево – назад – вверх» [1, стр. 333].

Система координат пациента измеряется в миллиметрах, а начало координат в ней расположено произвольно и не соответствует началу координат массива воксеелей.

Многие компьютерные томографы часто отличаются друг от друга размером вокселей, которые обычно не кубической формы. Они могут иметь размеры, например, $1.125 \times 1.125 \times 2.5$ мм.

КТ-сканы обычно имеют размерность 512 строк на 512 столбцов, а по оси индексов обычно бывает от 100 полных срезов до, возможно, 250 срезов (250 срезов по 2.5 мм обычно достаточно, чтобы охватить интересующую область).

Заставить модель исследовать такие огромные массивы данных в поисках намеков на нужные нам узелки – это все равно что попросить вас найти одно слово с ошибкой в сборнике романов, написанных на незнакомом вам языке.

Вместо этого выделим область вокруг каждого кандидата и заставим модель рассматривать кандидатов по одному.

PyTorch API требует, чтобы любые подклассы `Dataset`, которые мы хотим реализовать, должны иметь две функции:

- `__len__`, которая после инициализации должна возвращать постоянное значение,
- `__getitem__`, которая принимает индекс и возвращает кортеж с демонстрационными данными.

Чтобы `LunaDataset` работал достаточно эффективно, нужно кэшировать результаты на диске.

```
@functools.lru_cache(1, typed=True)
def getCt(series_uid):
    return Ct(series_uid)

@raw_cache.memoize(typed=True)
def getCtRawCandidate(series_uid, center_xyz, width_irc):
    ct = getCt(series_uid)
    ct_chunk, center_irc = ct.getRawCandidate(center_xyz, width_irc)

    return ct_chunk, center_irc
```

Здесь мы используем несколько различных методов кэширования. Прежде всего мы кэшируем возвращаемое значение `getCt` в памяти, чтобы можно было многократно запрашивать один и тот же экземпляр `Ct`, не загружая заново все данные с диска. Это даст огромный прирост скорости в случае повторяющихся запросов, но мы сохраняем в памяти только один КТ, поэтому промахи кэша будут частыми, если мы не будем следить за порядком доступа [1, стр. 342].

Однако функция `getCtRawCandidate`, которая вызвает `getCt`, также кэширует свои выходные данные; поэтому после того, как наш кэш будет заполнен, функция `getCt` вызываться не будет. Эти значения *кэшируются на диск* с помощью библиотеки `diskcache` https://grantjenks.com/docs/diskcache/.

Библиотека DiskCache использует дисковое пространство для кэширования! Объект `Cache` потокобезопасный и потому его можно разделять между несколькими потоками. Два `Cache` объекта могут ссылаться на одну и ту же директорию из разных потоков или процессов. Все операции с кэшем в отличие от файлов являются *атомарными*.

```
from diskcache import Cache

cache = Cache()

with Cache(cache.directory) as reference:
    reference.set("key", "value")
```

Закрытые `Cache`-объекты откроются снова при доступе к ним. Но открытие `Cache`-объекта занимает относительно много времени, поэтому можно кэш не закрывать.

Работа с кэшем выглядит как с обычным словарем Python

```
cache["key"] = "value"
cache["key"]  # 'value'
"key" in cache  # True
del cache["key"]
```

Или так

```
from io import BytesIO
cache.set("key", BytesIO(b"value"), expire=5, read=True, tag="data")
```

Здесь ключ перестает быть доступным через 5 секунд, значение представляется как файловый объект и назначается тэг.

Прочитать значение по ключу можно так

```
cache.get("key", read=True, expire_time=True, tag=True)
```

Метод `.touch()` используется для обновления значения параметра `expire`

```
cache.touch("key", expire=None)
```

Метод `.add()` может использоваться для вставки пары в кэш. Пара вставляется только если ключ не существовал раньше.

```
cache.add(b"test", 123)  # True
cache[b"test"]
cache.add(b"test", 456)  # False
cache[b"test"]  # 123
```

Значения ключей можно изменять через инкремент / декремент (`.incr()`, `.decr()`).

`FanoutCache` автоматически сегментирует базу данных.

```
from diskcache import FanoutCache

cache = FanoutCache(shards=4, timeout=1)
```

Здесь создается кэш во временной директории с 4 сегментами и 1 секундой ожидания. Есть еще декоратор `memoize`

```
from diskcache import FanoutCache
cache = FanoutCache()
@cache.memoize(typed=True, expire=1, tag='fib')
def fibonacci(number):
  if number == 0:
```

```
    return 0
  elif number == 1:
    return 1
  else:
    return fibonacci(number - 1) + fibonacci(number - 2)
print(sum(fibonacci(value) for value in range(100)))  # 573147844013817084100
```

Размер кэша на диске задается с помощью параметра `size_limit`

```
cache = Cache(size_limit=int(4e9))  # 4 Gb
```

Считывать с диска 2^{15} значения типа `float32` намного быстрее, чем читать 2^{25} значений типа `int16`, преобразовывать их в `float32`, а затем выбирать из них 2^{15} значений. Начиная со второго прохода данных, время ввода-вывода для ввода должно сократиться до незначительного значения.

Если в системе установлено более одного графического процессора, то мы задействуем класс `nn.DataParallel` для распределения работы между всеми графическими процессорами в системе, затем собираем и повторно синхронизируем обновления параметров и т.д.

```
...
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
    model = model.to(self.device)
...
```

Вызов `model.to(self.device)` перемещает параметры модели в графический процессор, настраивая свертки и другие вычисления с целью использовать графический процессор для тяжелой вычислительной работы.

Важно сделать это перед созданием оптимизатора, поскольку в противном случае оптимизатору придется работать с объектами в центральном процессоре, а не с объектами, скопированными в графический процессор [1, стр. 355].

Здесь рассматривается случай использования нескольких графических процессоров с помощью класса `DataParallel`. Им легко обернуть уже имеющиеся модели. Но в целом этот способ применения нескольких графических процессоров не является самым эффективным и ограничен работой с оборудованием, имеющимся на одной машине.

В PyTorch также есть класс `DistributedDataParallel`, который рекомендуется использовать в случаях, когда вам нужно распределить работу между *несколькими* графическими процессорами или машинами. Правильно выполнить его настройку довольно непросто.

SGD довольно часто используется в качестве первого оптимизатора. В некоторых задачах SGD может работать плохо, но такие задачи относительно редки. Аналогично скорость обучения 0.001 и импульс 0.9 – достаточно безопасные стартовые значения. Опыт свидетельствует, что SGD с этими значениями хорошо показал себя в довольно широком круге проектов.

Нам не нужно реализовывать пакетную обработку, поскольку класс PyTorch `DataLoader` умеет это делать. Мы уже построили преобразование из КТ-сканов в тензоры PyTorch с помощью класса `LunaDataset`, поэтому осталось лишь подключить наш набор данных к загрузчику

```
def initTrainDl(self):
    train_ds = LunaDataset(  # пользовательский набор данных
        val_stride=10,
        isValSet_bool=False,
    )
```

```
    batch_size = self.cli_args.batch_size
    if self.use_cuda:
        batch_size *= torch.cuda.device_count()

    trail_dl = DataLoader(
        train_ds,
        batch_size=batch_size,  # Разбиение на пакеты выполняется автоматически
        num_workers=self.cli_args.num_workers,
        pin_memory=self.use_cuda,  # область памяти перемещается в GPU
    )
```

В дополнение к пакетной обработке отдельных образцов загрузчики данных (`DataLoader`) также могут обеспечивать *параллельную* загрузку данных с помощью отдельных процессов и общей памяти. Все, что нам нужно сделать, – это указать `num_workers=` при создании экземпляра загрузчика данных [1, стр. 358].

4.4. Первый сквозной дизайн нейронной сети

В каждом блоке одинаковый (или по крайней мере схожий) набор слоев, хотя часто размер входных данных и количество фильтров у блоков различаются.

```
class LunaBlock(nn.Module):
    def __init__(self, in_channels, conv_channels):
        super().__init__()
        self.conv1 = nn.Conv3d(
            in_channels, conv_channels, kernel_size=3,
            padding=1, bias=True,
        )

        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv3d(
            conv_channels, conv_channels,
            kernel_size=3, padding=1, bias=True
        )
        self.relu2 = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool3d(2, 2)

    def forward(self, input_batch):
        block_out = self.conv1(input_batch)
        block_out = self.relu1(block_out)
        block_out = self.conv2(block_out)
        block_out = self.relu2(block_out)

        return self.maxpool(block_out)
```

Если выходной воксель подается в другое ядро $3\times3\times3$ в качестве одного из краевых вокселей, то часть входных данных первого слоя будет находится за пределами области $3 \times 3 \times 3$ второго рода. Конечный результат двух сложенных слоев имеет эффективное рецептивное поле $5 \times 5 \times 5$. Это означает, что взятые вместе слои действуют так же, как один сверточный слой большего размера [1, стр. 362].

Иными словами, каждый сверточный слой $3\times3\times3$ добавляет дополнительный одновоксельный слой по краю рецептивного поля.

Два сложенных слоя $3 \times 3 \times 3$ используют меньше параметров, чем полная свертка $5 \times 5 \times 5$ (и, следовательно, вычисление также выполняется быстрее).

Чтобы добиться хорошей производительности модели, веса, смещения и другие параметры сети должны обладать определенными свойствами. Представим себе вырожденный случай, когда все веса сети больше 1 (и у нас нет остаточный связей). В такой случае повторное умножение на эти веса приведет к тому, что выходные данные слоя будут расти по мере прохождения данных через слои сети. Аналогично, вес меньшей 1 приведет к уменьшению и исчезновению выходных данных всех слоев. Такие же соображения применимы к градиентам в обратном проходе [1, стр. 365].

Обеспечить правильное поведение выходных данных слоя можно с помощью множества методов нормализации. Один из самых простых – убедиться, что веса сети инициализированы таким образом, чтобы промежуточные значения и градиенты не становились ни неоправданно малыми, ни неоправданно большими.

```
def _init_weights(self):
    for m in self.modules():
        if type(m) in {nn.Linear, nn.Conv3d}:
            nn.init.kaiming_normal_(
                m.weight.data, a=0, mode="fan_out", nonlinearity="relu"
            )
            if m.bias is not None:
                fan_in, fan_out = nn.init._calculate_fan_in_and_fan_out(m.weight.data)
                bound = 1 / math.sqrt(fan_out)
                nn.init.normal_(m.bias, - bound, bound)
```

Метод `.detach()`, то есть «отсоединение» используется тогда, когда объект не должен удерживать градиенты [1, стр. 369].

Цикл проверки работает аналогично. Кроме того, код работает быстрее благодаря использованию контекстного менеджера `with torch.no_grad()`, явно информирующего PyTorch о том, что *не нужно вычислять градиенты*.

Простой способ определить наличие узкого места в загрузке данных или вычислениях – подождать несколько секунд, а затем вызвать `top` или `nvidia-smi`:

- если 8 рабочих процессов Python загружают более 80% ЦП, то, вероятно, вам необходимо подготовить кэш,
- если `nvidia-smi` сообщает, что значение `GPU-Util` более 80%, ГП загружен работой.

Наша цель – насытить ГП работой, то есть мы хотим использовать как можно больше его вычислительной мощности для быстрого завершения эпох.

Для визуализации метрик удобно использовать `TensorBoard` (требуется установить пакет `tensorflow`). Можно установить стандартный пакет только для ЦП.

Рекомендуется разделить ваши данные по отдельным папкам, так как `TensorBoard` несколько загромождается, если вы проведете более 10 или 20 экспериментов.

```
$ tensorboard --logdir ./runs
```

Точность и отклик (полнота) – это ценные показатели, которые можно отслеживать во время обучения. Если любой из них упадет до нуля, то стоит предположить, что модель начала вести себя вырожденным образом [1, стр. 401].

В данном случае соотношение положетиельных точек к отрицательным составляет 400:1. Это *катастрофически* большой дисбаланс! Неудевительно, что настоящие узелки попросту теряются в общей массе!

В качестве функции потерь будем использовать `nn.CrossEntropyLoss`, которая технически оперирует необработанными логитами, а не вероятностями класса.

Прогнозы, численно близкие к правильной метке, не приводят к значительному изменению весов сети, а прогнозы, значительно отличающиеся от правильного ответа, влияют на веса гораздо сильнее.

Наш обучающий набор нужно будет отредактировать так, чтобы в нем *чередовались* положительные и отрицательные элементы.

Мы *не будем проводить балансировку проверочного* (тестового) набора. Наша модель должна хорошо работать в реальном мире, а он несбалансирован (в конце концов, именно из реального мира мы получили необработанные данные!) [1, стр. 414].

Мы *дополняем* набор данных, применяя синтетические изменения к отдельным элементам, в результате чего получается новый набор данных, превышающий по размеру исходный. Обычно цель дополнения – получить синтетический набор, который остается репрезентативным для того же общего класса, что и исходный, но который нельзя тривиально запомнить вместе с оригиналом. При правильном выполнении это увеличение может увеличить тренировочный набор настолько, что модель не сможет его запомнить, и тогда она будет вынуждена больше полагаться на обобщение.

Методы дополнения (аугментации) данных:

- зеркальное отражение изображения вверх-вниз, влево-вправо и/или вперед-назад,
- сдвиг изображения на несколько вокселей,
- масштабирование изображения вверх или вниз,
- вращение изображения вокруг оси «голова – нога»,
- добавление шума.

Для каджого метода нужно убедиться, что выбранный подход сохраняет репрезентативный характер измененного обучающего элемента, но в то же время отличается достаточно, чтобы этот элемент был полезен для обучения.

NB: Важно структурировать конвейер данных таким образом, чтобы этапы кэширования выполнялись до дополнения данных! В противном случае ваши данные дополнятся, а затем будут сохранены в таком состоянии, а это противоречит цели.

Классификация говорит, есть ли кот на картинке, а сегментация показывает где именно. Модели классификации дают ответ в форме «Да, где-то в этой огромной куче пикселей есть кот» или «Нет, здесь нет котов» [1, стр. 440].

4.5. Архитектура U-Net

Архитектура U-Net – это предназначенный для сегментации дизайн нейронной сети, который может производить попиксельные выходные данные. Данные проходят сверху слева к центру через серию сверток и масштабирования. Затем мы снова поднимаемся вверх, используя *развертку*, чтобы вернуться к полному разрешению.

Однако в более ранних проектах у таких сетей были проблемы со сходимостью, скорее всего, из-за потери пространственной информации во время субдискритизации. Чтобы решить эту проблему, авторы U-Net добавили *пропущенные соединения*. В U-Net пропущенные соединения связывают входы на пути *понижения дискретизации* с соответствующими слоями на пути *повышения дискретизации*. Эти слои получают в качестве входных данных как результаты повышения частоты дискретизации слоев широкого рецептивного поля из нижних слоев U, так и выходные данные более ранних слоев с мелкими деталями через мостовые соединения вида «копировать и обрезать». Это ключевое нововведение U-Net. В результате окончательные слои детализации

берут лучшее из обоих источников данных. У них есть информация о более широком контексте, окружающем непосредственную область, а также подробные данные из первого набора слоев с полным разрешением. Слой *conv* 1×1 в крайнем правом углу сети изменяет количество каналов с 64 на 2 [1, стр. 445].

Вместо того чтобы просто выводить бинарную классификацию, которая дает вывод в виде «истина» или «ложь», мы реализуем U-Net, чтобы получить модель, способную выводить значение вероятности для каждого пикселя, то есть выполнять сегментацию.

4.5.1. Адаптация готовой модели

Мы будем передавать входные данные через пакетную нормализацию. Тогда нам не придется самостоятельно нормализовать данные в наборе данных; и что более важно, мы получим статистику нормализации (среднее значение и стандартное отклонение), расчитанную по отдельным пакетам.

Случайный выбор элементов данных в пакетах в каждую эпоху сводит к минимуму вероятность того, что скучный элемент окажется в полностью скучном пакете, и, следовательно, такие элементы будут рассматриваться чрезмерно внимательно.

Во-вторых, поскольку выходные значения не ограничены, мы должны пропустить выходные данные через слой `nn.Sigmoid`, чтобы ограничить их диапазоном $[0, 1]$. В-третьих, мы уменьшим общую глубину и количество фильтров, которые модель будет применять. Это значит, что мы вряд ли найдем предварительно обученную модель, которая будет точно соответствовать нашим потребностям. Выходные данные получаются одноканальными, где каждый пиксель вывода содержит оценку вероятности того, что он является частью узелка.

4.5.2. Особые требования U-Net к размеру входных данных

Первая проблема заключается в том, что размеры входных и выходных фрагментов данных в U-Net очень специфичны. В документе U-Net использовались фрагменты изображений размером 572×572, что дает выходные карты размером 388×388. Входные изображения должны быть больше, чем наши срезы 512×512, а выходные данные оказываются чуть меньше! Это означало бы, что узелки вблизи края среза КТ вообще не будут сегментированы.

4.5.3. Компромиссы U-Net при работе с 3D- и 2D-данными

Вторая проблема заключается в том, что наши 3D-данные не совсем совпадают с ожидаемыми U-Net двумерными входными данными. Просто взять наше изображение размером $512 \times 512 \times 128$ и передать его в преобразованный в 3D класс U-Net не получиться, поскольку на это хватит памяти графического процессора. Каждое изображение имеет размеры $2^9 \times 2 \times^9 \times 2^7$, по 2 байта на каждый воксель. Первый уровень U-Net – это 64 канала, или 2^6. Это показатель степени $9+9+7+2+6 = 33$, или 8 Гб, только для первого сверточного слоя. У нас 2 сверточных слоя (16 Гб), а затем каждое понижение разрешения уменьшает разрешение вдвое, но удваивает каналы, то есть еще 2 Гб для каждого слоя после первого понижения разрешения. В итоге мы дошли до 20 Гб еще до того, как достигли второго понижения разрешения.

Каждый срез будем рассматривать как задачу двумерной сегментации, а в качестве контекста третьего измерения станем передвать соседние срезы как отдельные каналы. Вместо традиционных красных, зеленых и синих каналов, используемых в фотографиях, нашими каналами будут

«на два среза выше», «на один срез выше», «сегментируемый срез», «на один срез ниже» и т.д. [1, стр. 451]

Однако, мы теряем прямую пространственную связь между срезами, когда они передаются в виде каналов, поскольку все каналы будут линейно объединены ядрами свертки без указания того, с какой стороны и на каком расстоянии находятся срезы. Еще один момент, который следует учитывать и в текущем, и трехмерном подходе, заключается в том, что теперь мы игнорируем точную толщину среза.

Для ввода в нашу модель классификации мы рассматривали эти срезы как трехмерный массив данных и использовали трехмерные свертки для обработки каждого элемента. В модели сегментации мы будем рассматривать каждый срез как один канал и создавать многоканальное 2D-изображение. Каждый срез компьютерной томографии это как бы цветовой канал изображения RGB. Входные срезы КТ будут сложены вместе и использоваться так же, как и любое другое 2D-изображение. Каналы совмещенного КТ-изображения не будут соответствовать цветам, но в 2D-свертках и не требуется, чтобы входные каналы были именно цветами.

Вместо полных срезов КТ мы будем выполнять обучение на фрагментах 64×64 вокруг положительных кандидатов (настоящих узелков). В качестве дополнительных «каналов» для 2D-сегментации мы также включим по три фрагмента сверху и снизу.

Этот подход позволяет сделать обучение более стабильным и ускорить сходимость. Мы считаем, что обучение с помощью всего фрагмента оказалось нестабильным из-за проблемы с *балансировкой классов*. Поскольку каждый узелок весьма мал по сравнению со всем срезом КТ, мы снова оказались в ситуации «иголка в стоге сена», когда положительные элементы данных были завалены отрицательными. В данном случае мы говорим о пикселях, а не об узелках, но суть та же. Выполняя обучение на фрагментах, мы сохраняем то же количество положительных пикселей (относящихся к узелкам), но уменьшаем количество отрицательных пикселей на несколько порядков [1, стр. 464].

Поскольку наша модель сегментации является попиксельной и принимает изображения произвольного размера, мы можем обойтись без обучения и проверки на выборках с разными размерами.

Для этого подхода важно отметить, что, поскольку, проверочный набор содержит на несколько порядков больше отрицательных пикселей, во время проверки модель будет выдавать огромный процент ложноположительных результатов.

Необходимо устранять узкие места в конвейере обучения. Хитрость заключается в том, чтобы убедиться, что узкое место находится в самом дорогом и трудном для обновления ресурсе и использование этого ресурса не является расточительным.

Часто узкие места возникают в следующих процессах [1, стр. 466]:

- в конвейере загрузки данных во время работы с необработанным вводом-выводом или при распаковке данных, когда они находятся в ОЗУ. Можно решить эту проблему с помощью библиотеки `diskcache`,
- при предварительной обработке загружаемых данных в процессоре. Обычно это *нормализация* или *дополнение данных*,
- в цикле обучения на ГП. Обычно нам лучше иметь узкое место именно здесь, поскольку общие затраты на операции глубокого обучения у графических процессоров обычно выше, чем у хранилища или ЦП,

- реже узким местом становится пропускная способность памяти между ЦП и ГП. Это означает, что ГП выполняет не так уж много работы по сравнению с объемом перемещаемых данных.

Поскольку графические процессоры при выполнении определенного класса задач могут быть в 50 раз быстрее, чем центральные, часто имеет смысл переносить эти задачи на ГП с ЦП, если последний слишком сильно нагружается. Это особенно верно в случаях, когда данные дополняются (аугментируются) во время этой обработки, так как при перемещении меньшего объема входных данных в графический процессор дополнительные данные остаются для него локальными, поэтому приходится перемещать меньше данных.

Список литературы

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