Обучение многослойных нейросетей на numpy

Андрей Стоцкий, Максим Красильников, Влад Шахуро

Graphics & Media Lab
Vision Group

В этом задании вы:

- изучите методы стохастического градиентного спуска и обратного распространения ошибки,
- реализуете прямой и обратный проходы различный нейросетевых слоев,
- обучите полносвязную нейросеть для классификации рукописных цифр `MNIST`,
- обучите сверточную нейросеть для классификации изображений из набора `CIFAR-10`.

Критерии оценки

Максимальная оценка за задание — 20 баллов.

Раздел считается выполненным, если в проверяющей системе пройдены все юнит-тесты из данного раздела.

Список разделов и максимальных баллов

Часть 1. Полносвязанные нейросети

Раздел	Название	Баллы
1.4.1	`SGD`	0.5
1.4.2	`SGDMomentum`	0.5
2.1.1	`ReLU`	1.0
2.1.2	`Softmax`	1.5
2.1.3	`Dense`	2.0
2.2.1	`CrossEntropy`	1.0
2.3	`MNIST`	2.0

Часть 2. Сверточные нейросети

Раздел	Название	Баллы
3.3.2	`convolve`	1.0
4.1.1	`Conv2D`	2.0
4.1.2	`Pooling2D`	1.5
4.1.3	`BatchNorm`	2.0
4.1.4	`Flatten`	1.0
4.1.5	`Dropout`	1.0
4.2	`CIFAR-10`	3.0

Файлы и разархивированные zip-архивы из проверяющей системы разместите следующим образом:

```
├── common.py
├── interface.py
├── run.py
├── solution.py
└── tests/
    ├── 00_unittest_relu_input/
    ├── 01_unittest_softmax_input/
    └── ...
```

Для решения задания нужно заполнить в файле `solution.py` фрагменты кода, помеченные `# Your code here`. В файле `interface.py` находятся абстрактные классы и уже написанный за вас код.

В проверяющую систему нужно загружать только файл `solution.py`.
Решения с изменённым кодом (кроме `# Your code here`) могут быть не зачтены.

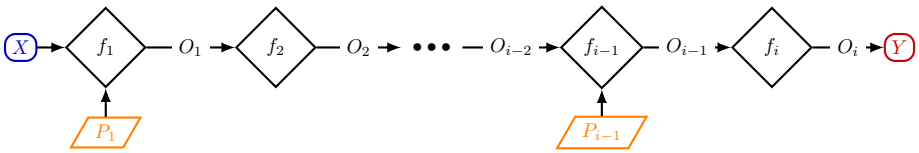
1 Принципы работы многослойных нейросетей

1.1 Вычислительные графы

В современной литературе и библиотеках для машинного обучения принято моделировать нейронные сети как направленные, ациклические вычислительные графы. Вершинами в таких графах являются данные и операции над этими данными, а ребра показывают зависимости (аргументы/результаты вычислений).

Обратите внимание, что данные в таких графах – многомерные массивы, а не числа. Такие массивы называются тензорами и являются обобщением векторов и матриц на произвольное количество индексов.

Рассмотрим пример вычислительного графа для последовательной многослойной нейросети.



Вычислительный граф

В данном графе, вершины X и Y соответствуют входу и выходу нейросети, вершины f_i – каким-то произвольным функциям, а P_i – внутренним параметрам нейросети.

На ребрах подписаны промежуточные результаты вычисления графа:

$$\begin{aligned} O_1 &= f_1(X, P_1) \\ O_2 &= f_2(O_1) \\ &\vdots \\ O_{i-1} &= f_{i-1}(O_{i-2}, P_{i-1}) \\ Y = O_i &= f_i(O_{i-1}) \end{aligned}$$

В последовательных многослойных нейросетях, каждая функция и все связанные с этой функцией параметры вместе называются слоем нейросети. Слои нейросети, у которых нет внутренних параметров, называются функциями активаций. Например, (f_1, P_1) – первый слой этой нейросети, а f_i – функция активации и последний слой.

В общем случае, все упомянутые далее методы применимы к произвольным вычислительным графам, однако в данном задании нами будут рассмотрены только последовательные вычислительные графы, в которых у каждого слоя только 1 вход и 1 выход.

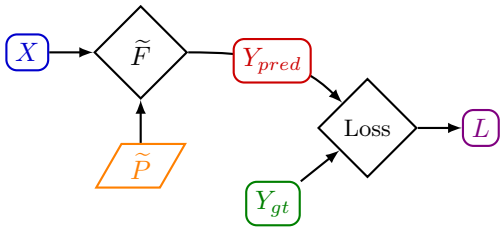
1.2 Обучение с учителем

Процесс обучения нейросети заключается в подборе параметров таким образом, чтобы приблизить выходы нейросети к желаемым. Такой подход к обучению называется обучением с учителем.

В данном задании, мы рассмотрим метод стохастического градиентного спуска, который является де-факто самым простым и распространённым методом оптимизации параметров.

Для обучения методом градиентного спуска выбирается специальная функция Loss, называющаяся функцией ошибок или функцией потерь. Данная функция должна оценивать, насколько точно выход

нейросети приближает интересующий нас результат.



Функция потерь

Пусть $\widetilde{F}$ и $\widetilde{P}$ – все слои и все параметры нейросети. Тогда, для заранее известной пары $(X,\ Y_{gt})$, где X – входные данные нейросети, а Y_{gt} – эталонный (правильный) результат для этого входа X, вычислим

$$\begin{aligned} Y_{pred} &= \widetilde{F}\left(X,\ \widetilde{P}\right) && \text{— предсказание (выход нейросети)} \\ L &= \text{Loss}\left(Y_{gt},\ Y_{pred}\right) && \text{— ошибка предсказания (скаляр/число)} \end{aligned}$$

Целью обучения с учителем является минимизация ошибки предсказания. Очевидно, что чем меньше значение функции потерь, тем лучше нейросеть предсказывает эталонные значения Y_{gt}. Формально, мы хотим найти набор параметров $\widetilde{P}$, минимизирующий значение L для *всех* известных пар $(X,\ Y_{gt})$.

1.3 Стохастический градиентный спуск

Метод стохастического градиентного спуска – одна из стратегий поиска данного $\widetilde{P}$. Рассмотрение *всех* эталонных пар – вычислительно сложная задача. Вместо рассмотрения *всех* эталонных значений, метод градиентного спуска предлагает последовательно улучшать предсказания $\widetilde{P}$ для разных *маленьких подмножеств* эталонных данных, называемых *батчами* (batch).

Рассмотрим, как работает метод стохастического градиентного спуска для *одной* пары эталонных значений. Шаг градиентного спуска предлагает обновлять параметры нейросети по следующему правилу:

$$\widetilde{P} \leftarrow \widetilde{P} - \alpha \cdot \frac{\partial L}{\partial \widetilde{P}}\bigg|_X$$

Данное правило изменяет каждый параметр в направлении, обратному производной L. При выборе достаточно маленького значения α, значение ошибки L для *данной* пары $(X,\ Y_{gt})$ будет уменьшаться.

Если в батче больше одной пары, то значение функции потерь Loss усредняется по всем элементам батча. То есть,

$$L = \sum_{(X,\ Y_{gt})\ \in\ \text{Batch}} \frac{\text{Loss}\left(Y_{gt},\ \widetilde{F}\left(X,\ \widetilde{P}\right)\right)}{|\text{Batch}|}$$

Это усреднение позволяет сгладить случайную природу выбора эталонных пар из всех данных для формирования батчей. Последовательно выбирая разные батчи и повторяя данную операцию для *всех* эталонных пар, мы сможем приблизительно найти локальный минимум функции потерь.

1.4 Реализация стохастического градиентного спуска

Вам дан интерфейс класса `Optimizer`. Для каждого обучаемого параметра (зарегистрированного через `add_parameter`) будет вызвана функция `get_parameter_updater`. Вам нужно реализовать функцию `updater`, которая вычисляет новое значение параметра на основании его текущего значения и последней частной производной.

1.4.1 `SGD`

Реализуйте стратегию обновления параметров стохастического градиентного спуска, описанную в предыдущем разделе.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest sgd
```

1.4.2 `SGDMomentum`

Может оказаться, что направление многомерной производной сильно меняется от шага к шагу. В такой ситуации, чтобы добиться более эффективной сходимости, можно усреднять производную с нескольких предыдущих шагов – в этом случае шум уменьшится, и усреднённая производная будет более стабильно указывать в сторону общего направления движения.

Введем тензор инерции $\widetilde{I}$, изначально равный 0. Тогда шаг градиентного спуска с инерцией будет:

$$\widetilde{I} \leftarrow \beta\widetilde{I} + \alpha\frac{\partial L}{\partial \widetilde{P}} \quad \text{— обновим тензор инерции на основании текущей производной}$$
$$\widetilde{P} \leftarrow \widetilde{P} - \widetilde{I} \quad \text{— изменим веса в направлении инерции}$$

Гиперпараметр β называется инерцией (momentum) градиентного спуска. При $\beta = 0$, данная стратегия эквивалентна обычному стохастическому градиентному спуску. Для $0 < \beta < 1$, вклад прошлых градиентов в текущий шаг уменьшается со скоростью β^k (где k – номер шага).

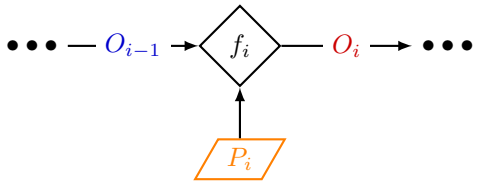
Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest momentum
```

1.5 Обратное распространение ошибки

На первый взгляд может быть не очевидно, как эффективно вычислить $\frac{\partial L}{\partial \widetilde{P}}$. Для вычисления частных производных в произвольных вычислительных графах можно использовать метод обратного распространения ошибки.

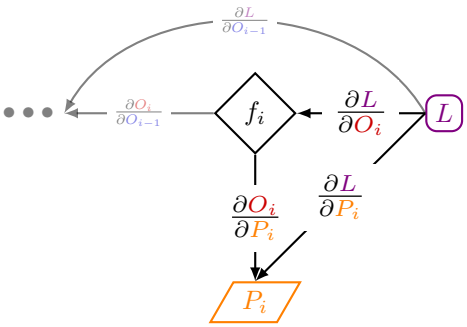
Суть метода обратного распространения ошибки заключается в последовательном применении цепного правила дифференцирования сложной функции. Для каждого слоя реализуются “прямой” и “обратный” вычислительные проходы. Рассмотрим один слой из нейросети.



Прямой проход

При прямом проходе просто вычисляется значение функции и передаётся в следующий слой.

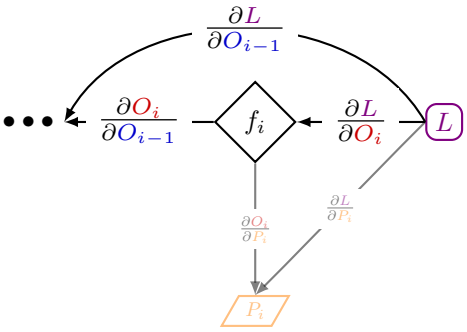
При обратном проходе индуктивно предполагается, что уже известна производная $\frac{\partial L}{\partial O_i}$.



Обратный проход (Параметры)

Тогда интересующая нас производная $\frac{\partial L}{\partial P_i}$ может быть вычислена как

$$\frac{\partial L}{\partial P_i} = \frac{\partial L}{\partial O_i} \cdot \frac{\partial O_i}{\partial P_i}$$



Обратный проход (Входы)

Чтобы сохранить предположение индукции для слоя $i-1$, вычислим

$$\frac{\partial L}{\partial O_{i-1}} = \frac{\partial L}{\partial O_i} \cdot \frac{\partial O_i}{\partial O_{i-1}}$$

Обратите внимание, что в данных формулах, $\frac{\partial O_i}{\partial P_i}$ и $\frac{\partial O_i}{\partial O_{i-1}}$ зависят только от выбора функции f_i, значений параметров P_i и входа O_{i-1}. Тогда, для каждого конкретного слоя (функции), прямой и обратный проходы можно задать аналитически.

Для самого последнего слоя $O_i = Y_{pred}$ и тогда значение $\frac{\partial L}{\partial Y_{pred}}$ зависит только от выбора функции Loss, самого предсказания Y_{pred} и эталонного значения Y_{gt}. Следовательно, прямой и обратный проходы для функции потерь также задаются аналитически и являются базой для нашего индуктивного предположения.

Особо внимательные читатели заметят, что производные в данных формулах – тоже тензоры. При этом символ ‘$\cdot$’ следует интерпретировать как тензорное произведение.

Формально, многомерные производные называются градиентами или матрицами Якоби. Цепное правило дифференцирования сложной функции верно для тензоров в силу линейности операций дифференцирования и тензорного умножения.

Более подробно о многомерных производных в нейросетях можно прочитать здесь.

2 Полносвязанные многослойные нейросети

2.1 Реализация слоев

В этой части задания вам будет необходимо реализовать прямые и обратные проходы для некоторых слоев, часто используемых в многослойных полносвязанных нейросетях.

Вам дан абстрактный класс `Layer`, который предоставляет интерфейс одного слоя нейросети. Вам необходимо аналитически вывести формулы, нужные для обратного прохода нейросети и реализовать функции `forward_impl` и `backward_impl`.

Функция `backward_impl` принимает $\frac{\partial L}{\partial O_i}$ и возвращает $\frac{\partial L}{\partial O_{i-1}}$. Если у данного слоя есть обучаемые параметры, то функция `backward_impl` также должна обновить значения их производных $\frac{\partial L}{\partial P}$.

Обратите внимание, что все вычисления необходимо делать тензорно (т.е. используя функции `numpy` и `numpy.ndarray`, а не циклы и списки). Слои оперируют сразу над целыми батчами значений, поэтому размерность всех тензоров начинается с `n` – размерности батча.

Если вам все ещё не понятны методы градиентного спуска и обратного распространения ошибки, рекомендуем просмотреть серию видео 3Blue1Brown про нейронные сети.

2.1.1 `ReLU`

$$X, Y \in \mathbb{R}^{(\cdots)}$$

$$y = \mathrm{ReLU}(x) = \begin{cases} x, & x \geqslant 0 \\ 0, & x < 0 \end{cases} = \max(x, 0)$$

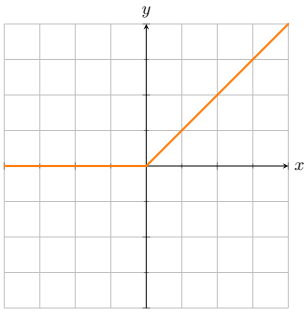


График ReLU для одного элемента

ReLU, Rectified Linear Unit, линейный выпрямитель или полулинейный элемент — это поэлементная функция активации. То есть функция ReLU независимо применяется к каждому элементу входного тензора.

Для обратного прохода вам могут понадобиться значения последних входов нейросети. Для вашего удобства, они автоматически сохраняются в `self.forward_inputs`.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest relu
```

2.1.2 `Softmax`

$$X \in \mathbb{R}^d$$
$$Y \in [0,1]^d$$

$$Y = \text{Softmax}(X) = \left\{ y_i = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}} \right\}_{i=1..d}$$

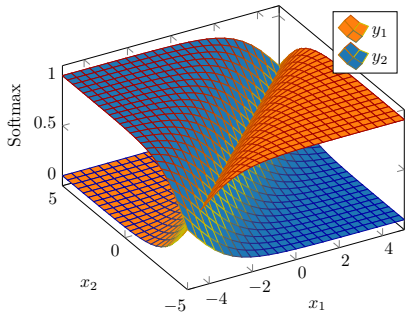


График Softmax для $d = 2$

Softmax, normalized exponential function, многомерная логистическая функция — функция активации.

Особенностью Softmax является то, что её выход — вектор, который можно интерпретировать, как вектор вероятностей. Действительно, ведь все значения $0 \leqslant y_i \leqslant 1$ и $\sum_{i=1}^{d} y_i = 1$.

Для обратного прохода вам могут понадобиться значения последних выходов нейросети. Для вашего удобства они автоматически сохраняются в `self.forward_outputs`.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest softmax
```

2.1.3 `Dense`

$$X \in \mathbb{R}^d$$
$$Y \in \mathbb{R}^c$$

$$Y = W \cdot X + B = \left\{ y_i = \sum_{j=1}^{d} w_{ij} x_j + b_i \right\}_{i=1..c}$$

$$W \in \mathbb{R}^{c \times d}$$
$$B \in \mathbb{R}^c$$

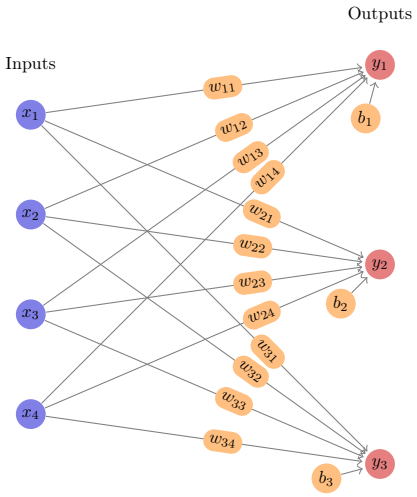


Схема Dense слоя для $d = 4$ и $c = 3$

Dense layer, Fully connected layer, полносвязанный слой, линейное преобразование. Вычисляет взвешенную линейную комбинацию входов и добавляет вектор сдвига.

Обратите внимание на переменные `self.weights`, `self.biases`, `self.weights_grad` и `self.biases_grad`. В них хранятся веса и вектор сдвигов и их частные производные. Не забудьте обновить значения частных производных в функции `backward_impl`.

Рассмотрите функцию `build`. В ней происходит инициализация обучаемых параметров.

Вообще говоря, существуют разные стратегии инициализации параметров. В данном задании мы инициализируем вектор сдвигов нулями, а веса – случайными значениями из нормального распределения с $\mu = 0, \sigma = \sqrt{2/d}$. Подробнее о данной стратегии инициализации вы можете прочитать здесь.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest dense
```

2.2 Реализация функции потерь

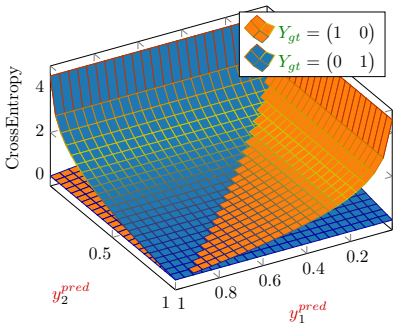
Вам дан интерфейс класса `Loss`, вам нужно реализовать вычисление значения функции (`value_impl`) и ее производной (`gradient_impl`).

2.2.1 `CrossEntropy`

$$\text{Loss} \in \mathbb{R}^+$$

$$Y_{pred},\ Y_{gt} \in [0,\ 1]^d$$

$$\text{Loss} = \text{CrossEntropy}\left(Y_{gt},\ Y_{pred}\right) = -\sum_{i=1}^{d} y_i^{gt} \ln\left(y_i^{pred}\right)$$



Графики CrossEntropy для d=2

Categorical Cross Entropy, категориальная кросс-энтропия, перекрёстная энтропия — функция потерь для сравнения векторов вероятностей.

Вспомним, что слой Softmax возвращает вектор Y_{pred}, который можно интерпретировать, как вектор вероятностей. Решая задачу классификации на d различных классов, будем рассматривать i-ый элемент этого вектора как вероятность принадлежности к i-ому классу.

Тогда, предполагая что для эталонных значений класс известен с 100% точностью, вектор Y_{gt} будет выглядеть как вектор, в котором во всех позициях находятся нули, кроме позиции i, где i – индекс эталонного класса. Такое представление называется one-hot encoding.

Обратите внимание, что в стохастическом градиентном спуске предполагается, что Loss усредняется по всем элементам батча. Особое внимание также уделите вычислительной стабильности связки Softmax + CrossEntropy.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest crossentropy
```

2.3 Обучение полносвязанной нейросети на MNIST

Примеры изображений из набора MNIST

В этой части вам необходимо обучить нейросеть для задачи классификации рукописных цифр MNIST. Вам дан класс Model, который реализует последовательную полносвязанную многослойную нейросеть.

От вас требуется реализовать функцию train_mnist_model, которая принимает на вход предобработанные данные из датасета MNIST для обучения и валидации и возвращает модель, обученную на этих данных.

2D (grayscale) изображения из датасета MNIST были предобработаны для вас. Значения были выпрямлены из тензора размера (28, 28) в вектор длины 784, а диапазон значений был приведен из целых чисел в интервале [0, 255] к числам с плавающей точкой с $\mu = 0$, $\sigma = 1$.

Преобразование формы тензора нужно для того, чтобы использовать данные в полносвязанных (Dense) слоях. Нормализация распределения входных значений обеспечивает базу индукции для предположения о нормальности распределения выходов нейросети в начале обучения и обеспечивает более стабильную сходимость.

Вам понадобятся функции add и fit класса Model, а также классы слоев, оптимизаторов и функции потерь, реализованные в предыдущих разделах. Обратите внимание, что для первого слоя нейросети необходимо явно указать размеры входных данных, используя параметр input_shape.

Архитектуру нейросети, гиперпараметры оптимизатора, шаг обучения, количество эпох и размер батча выберите на ваше усмотрение. Ваше решение должно обучаться не дольше 10 минут и иметь итоговую точность на тестовой выборке MNIST (Final test accuracy) более 90%.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest mnist
```

3 Принципы работы сверточных слоев

Выполняя задания из разделов про полносвязанные нейросети (разд. 2.1-2.3), вы могли обратить внимание, что получившаяся нейросетевая архитектура медленно обучается и имеет достаточно большое количество параметров.

При этом, датасет `MNIST` и сама задача распознавания рукописных цифр являются достаточно простыми даже по мерками тренировочных "игрушечных" наборов данных. Если же попробовать применить такую полносвязанную нейросеть к какому-либо более серьезному набору данных, то очень быстро станет понятно, что в лучшем случае алгоритм будет работать с плохой точностью, а возможно и совсем перестанет быть вычислительно доступным.

Основная причина такого плохого результата заключается именно в использовании полносвязанных слоев. Давайте рассмотрим примерную вычислительную сложность одного полносвязанного слоя.

Очевидно, что в формуле полносвязанного слоя самой сложной и медленной операцией является матрично-векторное умножение. Нетрудно показать, что при вычислении прямого и обратного проходов полносвязанного слоя будет выполнено $O(cd)$ умножений*.

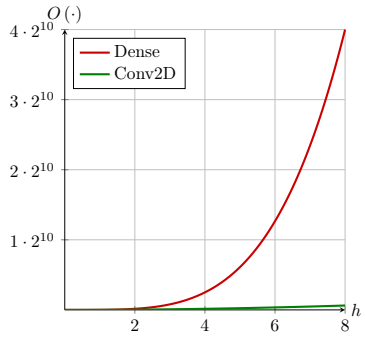
Эмпирические опыты показывают, что для достижения хорошей точности необходимо выбирать количество скрытых нейронов, примерно пропорциональное количеству входных нейронов $c \sim d$ (по крайней мере для ранних слоев нейросети). Отсюда получаем конечную оценку сложности $O\left(d^2\right)$.

Поскольку входной вектор был получен нами выпрямлением исходного изображения, как в разделе 2.3, то для изображения размера $d \times h \times h$ количество нейронов будет dh^2 и соответственно сложность вычисления такого полносвязанного слоя $O\left(d^2h^4\right)$.

В данной части задания рассмотривается математическая операция свертки и основанный на ней сверточный слой, сложность вычисления прямого и обратного прохода которого $O\left(d^2h^2k^2\right)$.

Эмпирические опыты показывают, что сверточные нейросети допускают $k \ll h$ и даже $k = \text{const}$ (очень часто используется $k = 3$). Отсюда получаем конечную оценку сложности $O\left(d^2h^2\right) \ll O\left(d^2h^4\right)$.

Например, даже для очень маленького изображения 32×32 пикселей, используя 32 канала/нейрона, полносвязанный слой будет иметь сложность $32^2 \cdot 32^4 = 2^{30} = 1\ 073\ 741\ 824$, а сверточный – всего $32^2 \cdot 32^2 = 2^{20} = 1\ 048\ 576$, в 1024 раза меньше.



Сравнение сложностей
для `Dense` и `Conv2D`

*Полученную оценку $O(cd)$ можно на самом теоретически улучшить, если использовать асимптотически оптимальный или параллельный алгоритм для матричного умножения, однако на практике подобные оптимизации либо не актуальны, либо равносильно применимы и для вычисления сверток.

3.1 Терминология

Во избежание путаницы, внимательно прочитайте данный раздел!

Термин “свертка” в литературе крайне перегружен. Достаточно часто словом “свертка” называются слегка разные по сути понятия. В данном задании рассматривается несколько объектов, которые все можно было бы назвать “свертка”. Чтобы вам было проще понять, о каком из объектов идет речь, мы будем использовать следующую терминологию

Математическая операция свертки.
Двухместная математическая операция из теорий функционального анализа и обработки сигналов. Обозначается символом $*$ и оперирует над одноканальными сигналами. Данная операция обсуждается в разделе 3.2.

Многоканальная свертка.
Функция `convolve(inputs, kernels)`, реализующая аналог матричного умножения для изображений. На вход принимает `inputs` – батч многоканальных изображений и `kernels` – “матрицу” с ядрами сверток. Вам предстоит реализовать данную функцию в разделе 3.3.

Сверточный слой.
Полноценный слой `Conv2D` с прямым и обратным проходами, вектором сдвига и обучаемыми параметрами. Использует ранее реализованную функцию `convolve`. Вам предстоит реализовать данный слой в разделе 4.1.

3.2 Математическая операция свертки

Из курсов математического/функционального анализа вам может быть знакома математическая операция свертки. Для пары скалярных функций $f(t)$ и $g(t)$, их свертка – это скалярная функция $o(t)$, задаваемая интегралом

$$f * g = o(t) = \int_{-\infty}^{+\infty} f(t-x)\, g(x)\, dx \qquad (f,\ g,\ o : \mathbb{R} \to \mathbb{R})$$

Операция свертки является в каком-то смысле аналогом операции умножения для функций. Так, например, операция свертки имеет свойства коммутативности $f * g = g * f$, ассоциативности $(f_1 * f_2) * f_3 = f_1 * (f_2 * f_3)$ и дистрибутивности с поточечным сложением $(f_1 + f_2) * g = (f_1 * g) + (f_2 * g)$.

Если предположить, что функции f, g и o задают значения каких-то величин, зависящих от времени (сигналов), то естественным образом можно вывести дискретный аналог операции свертки.

Для пары дискретизированных во времени сигналов F и G, их свертка – это дискретизированный во времени сигнал O, задаваемый суммой

$$F * G = O = \left\{ \sum_x F_{t-x} G_x \right\}_t \qquad (F,\ G,\ O \in \mathbb{R}^{\cdot})$$

Как вам известно, одноканальные изображения можно рассматривать как двухмерные сигналы. Из данной интерпретации вытекает определение математической операции свертки для одноканальных 2D изображений

$$F * G = O = \left\{ \sum_x \sum_y F_{t-x,s-y} G_{x,y} \right\}_{t,s} \qquad (F,\ G,\ O \in \mathbb{R}^{\cdot\times\cdot})$$

3.2.1 Граничные эффекты

Особо внимательные читатели могли заметить, что в дискретном определении операции свертки опущены границы суммирования и размеры операндов. Это было сделано намерено, так как данная деталь заслуживает отдельного раздела.

Дело в том, что в интегральной формулировке свертки, функции f, g и o определены на множестве всех действительных чисел. Однако на практике, в дискретном случае, мы бы хотели использовать операцию свертки для работы с конечными сигналами/изображениями.

Самый простой способ решения данной проблемы заключается в доопределении дискретизированных изображений нулями. При этом, элементы выхода, всегда тождественно равные нулю отбрасываются. Данный подход называется `full`.

Кроме метода `full` также часто используются методы `valid` и `same`. Метод `valid` предлагает брать на выходе алгоритма только те элементы O, которые не зависят от доопределеных элементов F. Метод `same` предлагает ограничить размер O, так, чтобы он совпал по размеру с входом F. Стоит заметить, что в отличие от метода `full`, методы `valid` и `same` не сохраняют свойства коммутативности и ассоциативности операции свертки.

Математические операции свертки одномерных сигналов и изображений фактически отличаются только количеством символов $\sum$ и индексов.

В целях улучшения читаемости и экономии места, в дальнейших теоретических выкладках, не теряя общности, будут рассматриваться свертки одномерных сигналов.

Рассмотрим более подробно практическую реализацию этих трех методов. Будем обозначать индексацию с учетом доопределения нулями квадратными скобками.

$$F_{[i]} = \begin{cases} F_i, & i \in [1,\ T_f] \\ 0, & \text{иначе} \end{cases}$$

Зафиксируем размеры наших аргументов. Тогда все три метода можно записать в обобщенном виде следующим образом

$$F \in \mathbb{R}^{T_f}, \quad G \in \mathbb{R}^{T_g}, \quad O \in \mathbb{R}^{T_o}$$

$$F \overset{(p)}{*} G = O = \left\{ \sum_{x=1}^{T_g} F_{[t-x]} G_x \right\}_{t=T_g-p+1\,..\,T_f+p+1}$$

В данной формулировке не очень удобно, что индексация O идет не от 1 до T_o, а от $T_g - p + 1$ до $T_f + p + 1$. Исправим это, сделав замену $t' = t - T_g + p$ и получим

$$F \overset{(p)}{*} G = O = \left\{ \sum_{x=1}^{T_g} F_{[t'+T_g-p-x]} G_x \right\}_{t'=1..T_o}$$

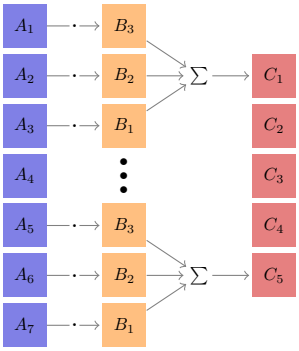
$$T_o = T_f - T_g + 1 + 2p$$

Где p задает размер области F, которую мы доопределили нулями.

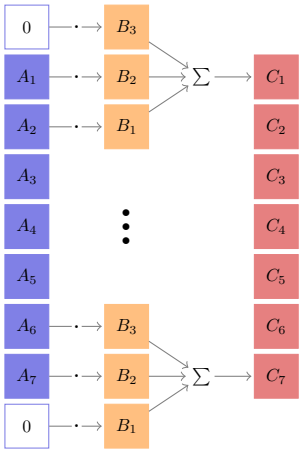
`valid`:	$p = 0$	$T_o = T_f - T_g + 1$
`same`:	$p = \dfrac{T_g - 1}{2}$	$T_o = T_f$
`full`:	$p = T_g - 1$	$T_o = T_f + T_g - 1$

В данном задании будем всегда считать, что при использовании метода `same` $T_g - 1$ делится на 2 нацело, а при использовании метода `valid` $T_f \geqslant T_g$.

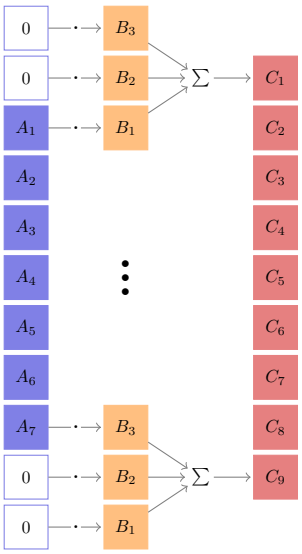
Визуализация разных методов сверток для $T_f = 7,\ T_g = 3$.
Показаны только суммы для крайних элементов выхода.



Свертка методом `valid`
$p = 0,\ T_o = 5$
(2D версия)



Свертка методом `same`
$p = 1,\ T_o = T_f = 7$
(2D версия)



Свертка методом `full`
$p = 2,\ T_o = 9$
(2D версия)

3.2.2 Другие виды сверток

Кроме выше рассмотренных методов дискретизации сверток, существует также множество других методов и вариаций. Они не будут использоваться в данном задании, однако имеет смысл их все же кратко рассмотреть.

Если предположить, что дополненная версия исходного изображения F должно иметь ту же структуру, что и исходное изображение F, то может иметь смысл дополнение его не нулями, а граничными значениями, отраженными копиями или нечетными/четными продолжениями F. Данные виды дополнения исходного изображения сравнительно редко используются в сверточных нейронных сетях.

Если предположить, что сигналы f, g и o были дискретизированны с разными частотами, то при переходе от интеграла к сумме получатся слегка другие индексы и границы суммирования.

Если частота дискретизации O в k раз меньше, то получится свертка с шагом (strided convolution)

$$F * G = \widetilde{O} = \sum_x F_{kt-x} G_x \qquad \left(\widetilde{T}_o = \frac{T_o}{k}\right)$$

Если частота дискретизации G в k раз меньше, то получится расширенная свертка (dilated/atrous convolution)

$$F * \widetilde{G} = O = \sum_x F_{t-kx} G_x \qquad \left(\widetilde{T}_g = kT_g\right)$$

Последний вид свертки – транспонированная свертка с шагом (transposed strided convolution) соответствует случаю, когда частота дискретизации F в k раз меньше. В отличие от расширенной

свертки и свертки с шагом, данный вид свертки проще всего представить как обычную свертку после замены

$$\widetilde{F}_i = \begin{cases} F_j, & \dfrac{i}{k} = j \in \mathbb{Z} \\ 0, & \dfrac{i}{k} \notin \mathbb{Z} \end{cases} \qquad \left(\widetilde{T}_f = kT_f\right)$$

Анимированную визуализацию всех трех видов сверток в 2D случае можно посмотреть тут: strided, dilated, transposed.

3.3 Многоканальная свертка

Разобранная нами операция свертки пока способна оперировать только одноканальными сигналами и одноканальными изображениями. Однако, нам бы хотелось помимо одноканальных изображений уметь оперировать и многоканальными.

При этом поддержка многоканальных изображений нам нужна не только для того, чтобы обрабатывать цветные (3-х канальные) изображения, но и для того, чтобы иметь возможность хранить в промежуточных представлениях нейросети информацию о нескольких видах структур.

Теоретически, мы бы могли просто рассмотреть каналы изображений, как просто еще одно измерение: сигнал – 1D свертка, одноканальное изображение – 2D свертка, многоканальное изображение – 3D свертка. Однако, данный подход на практике плохо работает, так как информация в разных каналах не имеет пространственных закономерностей.

Действительно, в многоканальном 2D изображении, значение (i, j)-го пикселя в k-ом канале связано со значениями соседних пикселей в том же канале абсолютно по другому, нежели оно связано со значениями того же пикселя в соседних каналах.

Вместо использования 3D сверток для обработки многоканальных 2D изображений, более правильным будет использовать специальную многоканальную свертку. Идею для того, как эта многоканальная свертка должна работать мы позаимствуем из полносвязанного Dense слоя.

3.3.1 Аналогия с матричным произведением

Как уже ранее упоминалось в разделе 3.2, операция свертки в определенном смысле похожа своими свойствами на операцию умножения. Более формально, рассмотрим одноканальные изображения, как самостоятельные объекты, образующие алгебраическое кольцо $(\mathbb{I}, +, *)$.

числа		изображения
$f,\ g \in \mathbb{R}$		$F,\ G \in \mathbb{I} = \mathbb{R}^{h \times w}$
умножение		операция свертки
$f \cdot g$	$\Longleftrightarrow$	$F * G$
сложение		поэлементное сложение
$f + g$		$F + G$

В такой интерпретации, многоканальные изображения – это векторы из одноканальных изображений.

векторы		многоканальные изображения
$X \in \mathbb{R}^d$		$X \in \mathbb{I}^d = \mathbb{R}^{d \times h \times w}$
матрицы	$\Longleftrightarrow$	многоканальные ядра
$W \in \mathbb{R}^{c \times d}$		$K \in \mathbb{I}^{c \times d} = \mathbb{R}^{c \times d \times h \times w}$

Соответственно, многоканальная свертка есть не что иное, как матричное произведение для векторов/матриц из элементов кольца $(\mathbb{I}, +, *)$.

$$\underset{\text{матричное произведение}}{W \cdot X =} = \left\{ \sum_{j=1}^{d} w_{ij} x_j \right\}_{i=1..c} \iff \underset{\text{многоканальная свертка}}{\text{convolve}\,(X,\ K) =} = \left\{ \sum_{j=1}^{d} X_j * K_{ij} \right\}_{i=1..c}$$

3.3.2 `convolve`

Рассмотрите функцию `convolve` в `solution.py`. Эта функция должна выполнять операцию многоканальной свертки, описанную в предыдущем разделе.

За вас написана реализация данной функции, которая перенаправляет вычисление в одну из функций `convolve_pytorch` или `convolve_numpy`. Вам нужно реализовать функцию `convolve_numpy`.

Тесты из данного раздела и раздела 4.1.1 будут использовать вашу реализацию `convolve_numpy`, а при обучении итоговой сверточной нейросети в разделе 4.2 будет автоматически использована более быстрая реализация `convolve_pytorch`, использующая библиотеку `pytorch`. Установите её, если у вас её нет. Обратите внимание, что `GPU` в `convolve_pytorch` не используется.

Чтобы упростить вам задачу, в качестве исключения, в функции `convolve_numpy` вам разрешается использовать не полностью векторизованную реализацию. В этой функции вы можете использовать до двух вложенных циклов `for` (например, по двум координатам ядра).

Обратите внимание, что первый аргумент имеет дополнительное измерение `n` – размер батча. Аргумент `padding` соответствует параметру p, описанному в разделе 3.2.1.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest convolve
```

Обратите внимание, что в некоторых библиотеках бывает, что функции с названиями вроде `convolve` на самом деле реализуют не свертку, а кросс-корреляцию. Кросс-корреляция отличается от свертки тем, что в кросс-корреляции второй аргумент не отражается перед суммированием.

Вообще говоря, при обучении нейронных сетей это отличие значения не имеет, так как веса настраиваются автоматически. Однако, в этом задании от вас требуется реализовать именно свертку, а не кросс-корреляцию.

4 Сверточные многослойные нейросети

4.1 Реализация слоев

В этой части задания вам будет необходимо реализовать прямые и обратные проходы для некоторых слоев, часто используемых в многослойных сверточных нейросетях.

Как и в разделе 2.1 вам необходимо аналитически вывести формулы, нужные для обратного прохода нейросети и реализовать функции `forward_impl` и `backward_impl`.

Обратите внимание, что все вычисления необходимо делать тензорно (т.е. используя функции `numpy` и `numpy.ndarray`, а не циклы и списки). Слои оперируют сразу над целыми батчами значений, поэтому размерность всех тензоров начинается с `n` – размерности батча.

4.1.1 `Conv2D`

$$X \in \mathbb{R}^{d \times h \times w} \qquad K \in \mathbb{R}^{c \times d \times p \times p}$$
$$Y \in \mathbb{R}^{c \times h \times w} \qquad B \in \mathbb{R}^{c}$$

$$Y = \sum_{j=1}^{d} X_j * K_{ij} + B = \text{convolve}\,(X,\ K) + B$$

Conv2D, Convolutional layer, сверточный слой. Слой, в основе которого лежит математическая операция свертки. Вычисляет многоканальную свертку входов с ядром параметров и добавляет к каждому каналу параметр сдвига.

Реализуйте прямой и обратный проходы. Для вычисления всех сверток в прямом и обратном проходах используйте функцию `convolve`. Параметры p выберите так, чтобы при прямом проходе не изменялись ширина и высота изображения (стратегия `same`). Ядро – квадратное, шаг свертки $k = 1$.

Обратите внимание на переменные `self.kernels`, `self.biases`, `self.kernels_grad` и `self.biases_grad`. В них хранятся веса и вектор сдвигов и их частные производные. Не забудьте обновить значения частных производных в функции `backward_impl`.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest conv2d
```

4.1.2 `Pooling2D`

$$X \in \mathbb{R}^{d \times ph \times pw}$$
$$Y \in \mathbb{R}^{d \times h \times w}$$
$$\text{reduce} \in \{\max,\ \text{mean}\}$$

$$Y = \underset{(t,s)\ \in\ \text{pool}_p}{\text{reduce}} X_{\cdot ts} = \left\{ \underset{s=1+pj}{\overset{t=1+pi}{\overset{p(j+1)}{\overset{p(i+1)}{\text{reduce}}}}} x_{k,t-p,s-p} \right\}_{\substack{k=1..d \\ i=1..h \\ j=1..w}}$$

Pooling2D, Local pooling layer, слой подвыборки. Позволяет уменьшить размеры тензора, при этом агрегируя локальную информацию.

Слой для каждого канала отдельно собирает пиксели в локальные группы размера $p \times p$ и вычисляет внутри каждой группы статистику reduce. Выбор функции reduce задается аргументом `pool_mode`.

Размер группы p – одинаковый по ширине и высоте, и делит ширину и высоту исходного изображения нацело. Смещение/шаг групп одинаковый по ширине и высоте и равен размеру группы (то есть группы не пересекаются).

Для случая reduce = max, можно запоминать индексы максимальных элементов для каждой группы в `forward_impl` и переиспользовать их в `backward_impl`.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest pooling2d
```

4.1.3 `BatchNorm`

$$X \in \mathbb{R}^{n \times d \times h \times w} \qquad \gamma \in \mathbb{R}^d \qquad \widetilde{\mu} \in \mathbb{R}^d$$
$$Y \in \mathbb{R}^{n \times d \times h \times w} \qquad \beta \in \mathbb{R}^d \qquad \widetilde{\nu} \in \mathbb{R}^d$$

$$Y = \text{BatchNorm}(X) = \gamma \cdot \widetilde{X} + \beta$$

$$\widetilde{X} = \text{normal}(X) :=$$

$$\begin{bmatrix} \underline{\text{Train}} \\ \mu = \underset{n,h,w}{\text{mean}}(X) \\ \nu = \underset{n,h,w}{\text{var}}(X) \\ \widetilde{X} = \dfrac{X - \mu}{\sqrt{\varepsilon + \nu}} \\ \widetilde{\mu} \leftarrow m\widetilde{\mu} + (1-m)\,\mu \\ \widetilde{\nu} \leftarrow m\widetilde{\nu} + (1-m)\,\nu \end{bmatrix} \quad \begin{bmatrix} \underline{\text{Inference}} \\ \widetilde{X} = \dfrac{X - \widetilde{\mu}}{\sqrt{\varepsilon + \widetilde{\nu}}} \end{bmatrix}$$

Batch Normalization, пакетная нормализация. Помогает стабилизировать процесс обучения нейронных сетей. Более подробное описание и формулы прямого и обратного проходов слоя можно найти в статье авторов.

Вообще говоря, полноценный слой `BatchNorm` способен работать как с изображениями, так и с "плотными" векторами, однако в данном задании от вас требуется реализовать только версию для работы с изображениями.

Обратите внимание, что γ = `gamma` и β = `beta` – это параметры, а $\widetilde{\mu}$ = `running_mean` и $\widetilde{\nu}$ = `running_var` – скользящие статистики*. Скользящие статистики отличаются от параметров тем, что для них не нужно вычислять градиенты, а обновление этих статистик производит слой `BatchNorm`, сразу в `forward_impl`, а не оптимизатор после обратного прохода. Гиперпараметр m = `momentum` задает скорость обновления скользящих статистик.

*Вообще говоря, авторы исходной статьи предлагают использовать несмещенную оценку при обновлении скользящей дисперсии, однако в данном задании мы для простоты игнорируем данную деталь.

На этапе тестирования не нужно вычислять μ или ν и не нужно обновлять $\widetilde{\mu}$ или $\widetilde{\nu}$. Вместо этого нужно просто использовать те значения $\widetilde{\mu}$ и $\widetilde{\nu}$, которые были получены в конце обучения.

Также учтите, что μ и ν, в отличие от $\widetilde{\mu}$ и $\widetilde{\nu}$ являются функциями от X и соответственно при расчете $\frac{\partial \tilde{X}}{\partial X}$ нужно не забыть учесть слагаемые $\frac{\partial \tilde{X}}{\partial \mu}\frac{\partial \mu}{\partial X} \neq 0$ и $\frac{\partial \tilde{X}}{\partial \nu}\frac{\partial \nu}{\partial X} \neq 0$.

Постарайтесь максимально экономить количество повторных вычислений, запоминая промежуточные результаты в `forward_impl` и используя их в `backward_impl`. Можете использовать в качестве ε константу `eps`, импортированную для вас из `interface`.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest batchnorm
```

4.1.4 `Flatten`

$$X \in \mathbb{R}^{d_1 \times \cdots \times d_n}$$
$$Y \in \mathbb{R}^{d}$$
$$d = d_1 \cdot \ldots \cdot d_n$$

$$Y = \text{Flatten}\,(X) =$$
$$= \{y_i = x_{i_1,..,i_n}\}$$

Flatten, спрямляющий слой. Данный слой нужен для того, чтобы перейти от сверточной части нейросети к вектору вероятностей на выходе.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest flatten
```

4.1.5 `Dropout`

$$X \in \mathbb{R}^{(\cdots)} \quad Y \in \mathbb{R}^{(\cdots)}$$

$$Y = \text{Dropout}\,(X) \coloneqq$$

$$\underline{\text{Train}} \qquad\qquad \underline{\text{Inference}}$$
$$\begin{bmatrix} M \sim \text{Bern}\,(1-p)^{(\cdots)} \\ Y = M \odot X \end{bmatrix} \quad \begin{bmatrix} \\ Y = (1-p)\,X \end{bmatrix}$$

Dropout. Метод регуляризации нейронных сетей. Используется после слоев с большим количеством параметров для уменьшения склонности нейросети к переобучению.

В процессе обучения с заданной вероятностью p обнуляет значения случайных элементов тензора. Для обратного прохода вам понадобится информация о том, какие элементы были обнулены, запомните эту информацию в функции `forward_impl`.

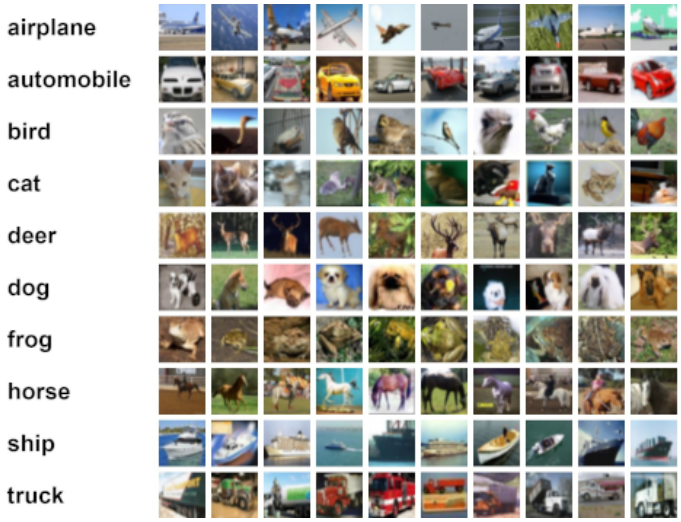
На этапе тестирования элементы не обнуляются, а домножаются на величину $(1-p)$. Это делается для того, чтобы усредненное значение активации нейронов при тестировании было примерно таким же, как и при обучении.

Для получения случайных величин используйте один вызов `np.random.uniform`. Не изменяйте самостоятельно `seed` генератора псевдослучайных чисел.

Проверьте реализацию с помощью юнит-теста.

```
$ ./run.py unittest dropout
```

4.2 Обучение сверточной нейросети на `CIFAR-10`



Примеры изображений из датасета `CIFAR-10`

В этой части вам необходимо обучить сверточную нейросеть для задачи классификации на датасете `CIFAR-10`. Вам дан класс `Model`, который реализует последовательную многослойную нейросеть.

От вас требуется реализовать функцию `train_cifar10_model`, которая принимает на вход предобработанные данные из датасета `CIFAR-10` для обучения и валидации и возвращает модель, обученную на этих данных.

Изображения представляют собой тензоры размера (3, 32, 32), а диапазон значений был приведен из целых чисел в интервале $[0, 255]$ к числам с плавающей точкой с $\mu = 0$, $\sigma = 1$.

Вам понадобятся функции `add` и `fit` класса `Model`, а также классы слоев, оптимизаторов и функции потерь, реализованные в предыдущих разделах. Обратите внимание, что для первого слоя нейросети необходимо явно указать размеры входных данных, используя параметр `input_shape`.

Архитектуру нейросети, гиперпараметры оптимизатора, шаг обучения, количество эпох и размер батча выберите на ваше усмотрение. Ваше решение должно обучаться не дольше 60 минут и иметь итоговую точность на тестовой выборке `CIFAR-10` (`Final test accuracy`) более 70%.

Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest cifar10
```

5 Подсказки

В данном разделе собраны аналитические выводы некоторых производных и обратных проходов, а также другие подсказки к заданию. Все необходимые для выполнения задания формулы и приемы вам уже известны из курсов математического анализа и дифференциальных уравнений. Перед тем как подглядывать решения из данного раздела, попробуйте вывести эти формулы сами.

Обратите внимание, что рассматриваемые здесь функции оперируют тензорами. В общем случае для таких функций, каждый выход y_i может зависеть от всех входов x_j, даже для $j \neq i$, а значит и каждая производная $\frac{\partial L}{\partial x_j}$ может зависеть от всех $\frac{\partial L}{\partial y_i}$. Для вычисления $\frac{\partial L}{\partial x_j}$ необходимо использовать правило дифференцирования сложной функции в многомерном случае:

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_j}$$

5.1 `ReLU`

$$\frac{\partial y}{\partial x} = \frac{\partial \max(x, 0)}{\partial x} = \begin{cases} 1, & x \geqslant 0 \\ 0, & x < 0 \end{cases} \qquad \text{(Частная производная } X\text{)}$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x} = \begin{cases} \dfrac{\partial L}{\partial y}, & x \geqslant 0 \\ 0, & x < 0 \end{cases} \qquad \text{(Обратный проход } X\text{)}$$

Итого (в тензорном виде)

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \odot [X \geqslant 0]$$

5.2 `Softmax`

$$\frac{\partial y_i}{\partial x_k} = \frac{\partial \left(\dfrac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}} \right)}{\partial x_k} = \frac{\dfrac{\partial e^{x_i}}{\partial x_k} \left(\sum_{j=1}^{d} e^{x_j} \right) - e^{x_i} \dfrac{\partial \left(\sum_{j=1}^{d} e^{x_j} \right)}{\partial x_k}}{\left(\sum_{j=1}^{d} e^{x_j} \right)^2} =$$

$$= \frac{e^{x_i} \left(\sum_{j=1}^{d} e^{x_j} \right) \delta_{ik} - e^{x_i} e^{x_k}}{\left(\sum_{j=1}^{d} e^{x_j} \right)^2} = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}} \frac{\left(\sum_{j=1}^{d} e^{x_j} \right) \delta_{ik} - e^{x_k}}{\sum_{j=1}^{d} e^{x_j}} = \qquad \text{(Частная производная } X\text{)}$$

$$= y_i(\delta_{ik} - y_k) = \begin{cases} y_i - y_i y_k, & i = k \\ -y_i y_k, & i \neq k \end{cases}$$

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} y_i(\delta_{ij} - y_j) = \frac{\partial L}{\partial y_j} y_j - \left(\sum_{i=1}^{d} \frac{\partial L}{\partial y_i} y_i \right) y_j \qquad \text{(Обратный проход } X\text{)}$$

Итого (в тензорном виде)

$$\frac{\partial L}{\partial X} = Y \odot \frac{\partial L}{\partial Y} - Y \cdot Y^T \cdot \frac{\partial L}{\partial Y}$$

5.3 `Dense`

$$\frac{\partial y_i}{\partial b_j} = \frac{\partial\left(\sum_{l=1}^{d} w_{il}x_l + b_i\right)}{\partial b_j} = \frac{\partial b_i}{\partial b_j} = \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \qquad \text{(Частная производная } B\text{)}$$

$$\frac{\partial L}{\partial b_j} = \sum_{i=1}^{c} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial b_j} = \sum_{i=1}^{c} \frac{\partial L}{\partial y_i}\delta_{ij} = \frac{\partial L}{\partial y_j} \qquad \text{(Обратный проход } B\text{)}$$

$$\frac{\partial y_i}{\partial w_{jk}} = \frac{\partial\left(\sum_{l=1}^{d} w_{il}x_l + b_i\right)}{\partial w_{jk}} = \frac{\partial(w_{ik}x_k)}{\partial w_{jk}} = \delta_{ij}x_k = \begin{cases} x_k, & i = j \\ 0, & i \neq j \end{cases} \qquad \text{(Частная производная } W\text{)}$$

$$\frac{\partial L}{\partial w_{jk}} = \sum_{i=1}^{c} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial w_{jk}} = \sum_{i=1}^{c} \frac{\partial L}{\partial y_i}\delta_{ij}x_k = \frac{\partial L}{\partial y_j}x_k \qquad \text{(Обратный проход } W\text{)}$$

$$\frac{\partial y_i}{\partial x_j} = \frac{\partial\left(\sum_{l=1}^{d} w_{il}x_l + b_i\right)}{\partial x_j} = \frac{\partial(w_{ij}x_j)}{\partial x_j} = w_{ij} \qquad \text{(Частная производная } X\text{)}$$

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^{c} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial x_j} = \sum_{i=1}^{c} \frac{\partial L}{\partial y_i}w_{ij} \qquad \text{(Обратный проход } X\text{)}$$

Итого (в тензорном виде)

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y} \qquad \frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} \cdot X^T \qquad \frac{\partial L}{\partial X} = W^T \cdot \frac{\partial L}{\partial Y}$$

5.4 `CrossEntropy`

Производная кросс-энтропии:

$$\frac{\partial L}{\partial y_i^{pred}} = \frac{\partial\left(-\sum_{j=1}^{d} y_j^{gt} \ln\left(y_j^{pred}\right)\right)}{\partial y_i^{pred}} = -y_i^{gt}\frac{\partial \ln\left(y_i^{pred}\right)}{\partial y_i^{pred}} = -\frac{y_i^{gt}}{y_i^{pred}} \qquad \text{(Обратный проход } y_i^{pred}\text{)}$$

5.5 Ошибки переполнения

При обучении сети у вас могут возникать ошибки, вызванные переполнением при вычислении экспонент в слое `Softmax`, логарифма и деления в функции потерь `CrossEntropy`. Это связано с тем, что функции e^x, $\ln(x)$ и $1/x$ могут выдавать большие по модулю значения при определенных x. Таким образом, при вычислениях могут получаться промежуточные результаты $\pm\infty$ или NaN.

В реальных библиотеках данная проблема обычно решается объединением слоев `Softmax` и `CrossEntropy` в вычислительном графе и использованием приема LogSumExp. Если вы столкнетесь с подобными ошибками в ходе решение данного задания, мы предлагаем вам использовать слегка упрощенную версию данного приема.

Обратите внимание, что

$$\forall c \in \mathbb{R}$$

$$\text{Softmax}(X + c) = \left\{ \frac{e^{x_i + c}}{\sum\limits_{j=1}^{d} e^{x_j + c}} = \frac{e^c e^{x_i}}{e^c \sum\limits_{j=1}^{d} e^{x_j}} = \frac{e^{x_i}}{\sum\limits_{j=1}^{d} e^{x_j}} \right\}_{i=1..d} = \text{Softmax}(X)$$

То есть если вычесть из всех элементов вектора X одно и то же число, то результат Softmax не изменится. Тогда вместо $\text{Softmax}(X)$ можно вычислять $\text{Softmax}\left(X - \max\limits_i x_i\right)$, который никогда не переполняется. Дополнительно, можно ограничить снизу значения Y_{pred} в прямом и обратном проходах функции `CrossEntropy` каким-нибудь маленьким числом ε.

$$Y_{pred} \leftarrow \max(\varepsilon, Y_{pred})$$

В коде можете использовать в качестве ε константу `eps`, импортированную для вас из `interface`.

5.6 Математическая операция свертки

Рассмотрим математическую операцию свертки в формулировке из раздела 3.2.1:

$$F \overset{(p)}{*} G = O = \left\{ \sum_{x=1}^{T_g} F_{[t' + T_g - p - x]} G_x \right\}_{t'=1..T_o}$$

$$T_o = T_f - T_g + 1 + 2p$$

Перед тем, как начинать вывод частных производных и обратного прохода, докажем следующее вспомогательное утверждение о псевдо-коммутативности операции $\overset{(p)}{*}$:

$$O = F \overset{(p)}{*} G = G \overset{(p')}{*} F = O'$$
$$p = p' + T_g - T_f$$

Для начала, покажем, что эти две свертки дают результат одинакового размера

$$\begin{aligned} T'_o &= T_g - T_f + 1 + 2p' && = \\ &= T_g - T_f + 1 + 2(T_f - T_g + p) && = \\ &= T_g - T_f + 2T_f - 2T_g + 1 + 2p && = \\ &= T_f - T_g + 1 + 2p && = T_o \end{aligned}$$

Учитывая данный факт, остается показать, что $O_i = {O'}_i$:

$$O_i = \sum_{x=1}^{T_g} F_{[i+T_g-p-x]} G_x = \sum_{x=-\infty}^{+\infty} F_{[i+T_g-p-x]} G_{[x]} =$$

$$= \left\{ \begin{array}{c} p = p' + T_g - T_f \\ x = i + T_f - p' - x' \end{array} \right\} =$$

$$= \sum_{x'=-\infty}^{+\infty} G_{[i+T_f-p'-x']} F_{[x']} = \sum_{x'=1}^{T_f} G_{[i+T_f-p'-x']} F_{x'} = {O'}_i$$

Утверждение о псевдо-коммутативности операции $\overset{(p)}{*}$ доказано.

Также, введем новое вспомогательное обозначение. Пусть символом $\widehat{\cdot}$ обозначается сигнал, проиндексированный в обратном порядке:

$$\widehat{F}_i = F_{(T_f+1)-i}$$

Для 2D изображений такая индексация соответствует отражению изображения по обеим осям.

Итак, приступим к аналитическому выводу формул частных производных и обратных проходов.

По левому аргументу

$$\frac{\partial \left\{ F \overset{(p)}{*} G \right\}_i}{\partial F_j} = \frac{\partial O_i}{\partial F_j} = \frac{\partial \left(\sum\limits_{x=1}^{T_g} F_{[i+T_g-p-x]} G_x \right)}{\partial F_j} =$$

$$= \left\{ \begin{array}{c} i + T_g - p - x = j \\ x = (T_g + 1) - (j + 1 + p - i) \end{array} \right\} = G_{[(T_g+1)-(j+1+p-i)]} = \widehat{G}_{[j+1+p-i]} = \quad \text{(Частная производная } F\text{)}$$

$$= \left\{ \begin{array}{l} 1 + p = T_o - \tilde{p} \\ \tilde{p} = T_o - p - 1 \end{array} \right\} = \widehat{G}_{[j+T_o-\tilde{p}-i]}$$

$$\frac{\partial L}{\partial F_j} = \sum_{i=1}^{T_o} \frac{\partial L}{\partial O_i} \frac{\partial O_i}{\partial F_j} = \sum_{i=1}^{T_o} \frac{\partial L}{\partial O_i} \widehat{G}_{[j+T_o-\tilde{p}-i]} =$$

$$= \sum_{i=1}^{T_o} \widehat{G}_{[j+T_o-\tilde{p}-i]} \frac{\partial L}{\partial O_i} = \widehat{G} \overset{(\tilde{p})}{*} \frac{\partial L}{\partial O} = \widehat{G} \overset{(T_o-p-1)}{*} \frac{\partial L}{\partial O} = \quad \text{(Обратный проход } F\text{)}$$

$$= \left\{ \begin{array}{c} \tilde{p} = T_o - p - 1 \\ p' = \tilde{p} + T_g - T_o = T_g - p - 1 \end{array} \right\} = \frac{\partial L}{\partial O} \overset{(p')}{*} \widehat{G} = \frac{\partial L}{\partial O} \overset{(T_g-p-1)}{*} \widehat{G}$$

	Прямой проход	$p' = T_g - p - 1$	Обратный проход	
`valid`:	$p = 0$	$\Longrightarrow$	$p' = T_g - 1$	`:full`
`same`:	$p = \frac{T_g - 1}{2}$	$\Longrightarrow$	$p' = \frac{T_g - 1}{2}$	`:same`
`full`:	$p = T_g - 1$	$\Longrightarrow$	$p' = 0$	`:valid`

По правому аргументу

$$\frac{\partial\left\{F \overset{(p)}{*} G\right\}_i}{\partial G_j} = \frac{\partial O_i}{\partial G_j} = \frac{\partial\left(\sum\limits_{x=1}^{T_g} F_{[i+T_g-p-x]} G_x\right)}{\partial G_j} = F_{[i+T_g-p-j]} =$$

$$= F_{[(T_f+1)-(j+p-T_g-i+T_f+1)]} = \widehat{F}_{[j+p-T_g-i+T_f+1]} =$$

$$= \left\{T_o = T_f - T_g + 1 + 2p\right\} = \widehat{F}_{[j+T_o-p-i]}$$

(Частная производная G)

$$\frac{\partial L}{\partial G_j} = \sum_{i=1}^{T_o} \frac{\partial L}{\partial O_i}\frac{\partial O_i}{\partial G_j} = \sum_{i=1}^{T_o} \frac{\partial L}{\partial O_i}\widehat{F}_{[j+T_o-p-i]} =$$

$$= \sum_{i=1}^{T_o} \widehat{F}_{[j+T_o-p-i]}\frac{\partial L}{\partial O_i} = \widehat{F} \overset{(p)}{*} \frac{\partial L}{\partial O}$$

(Обратный проход G)

Итого (в тензорном виде)

Обратный проход для операции свертки тоже реализуется, как операция свертки, но с одним перевернутым аргументом. Для левого аргумента при этом используется свертка с другим размером $p' = T_g - p - 1$.

$$\frac{\partial L}{\partial F} = \frac{\partial L}{\partial O} \overset{(p')}{*} \widehat{G} \qquad\qquad \frac{\partial L}{\partial G} = \widehat{F} \overset{(p)}{*} \frac{\partial L}{\partial O}$$

5.7 `convolve`/`Conv2D`

$$Y = \text{convolve}_p\left(X,\ K\right) + B = \left\{\sum_{j=1}^{d} C_{ij} + B_i\right\}_{i=1..c}$$

где $C_{ij} = X_j \overset{(p)}{*} K_{ij}$

$$\frac{\partial Y_i}{\partial B_j} = \frac{\partial(\text{convolve}_p\left(X,\ K\right)_i + B_i)}{\partial B_j} = \frac{\partial B_i}{\partial B_j} = \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

(Частная производная B)

$$\frac{\partial L}{\partial B_j} = \sum_{i=1}^{c}\sum_{t=1}^{h}\sum_{s=1}^{w} \frac{\partial L}{\partial Y_{its}}\frac{\partial Y_{its}}{\partial B_j} = \sum_{i=1}^{c}\sum_{t=1}^{h}\sum_{s=1}^{w} \frac{\partial L}{\partial Y_{its}}\delta_{ij} = \sum_{t=1}^{h}\sum_{s=1}^{w} \frac{\partial L}{\partial Y_{jts}}$$

(Обратный проход B)

$$\frac{\partial Y_i}{\partial K_{jk}} = \frac{\partial(\text{convolve}_p\,(X,\ K)_i + B_i)}{\partial K_{jk}} =$$

$$= \frac{\partial\left(\sum\limits_{r=1}^{d} C_{ir}\right)}{\partial K_{jk}} = \frac{\partial C_{ik}}{\partial K_{jk}} = \delta_{ij}\frac{\partial C_{jk}}{\partial K_{jk}}$$

(Частная производная K)

$$\frac{\partial L}{\partial K_{jk}} = \sum_{i=1}^{c}\frac{\partial L}{\partial Y_i}\frac{\partial Y_i}{\partial K_{jk}} = \sum_{i=1}^{c}\frac{\partial L}{\partial Y_i}\delta_{ij}\frac{\partial C_{jk}}{\partial K_{jk}} =$$

$$= \frac{\partial L}{\partial Y_j}\frac{\partial C_{jk}}{\partial K_{jk}} = \frac{\partial L}{\partial Y_j}\frac{\partial\left(X_k \overset{(p)}{*} K_{jk}\right)}{\partial K_{jk}} = \widehat{X}_k \overset{(p)}{*} \frac{\partial L}{\partial Y_j}$$

(Обратный проход K)

$$\frac{\partial Y_i}{\partial X_j} = \frac{\partial(\text{convolve}_p\,(X,\ K)_i + B_i)}{\partial X_j} =$$

$$= \frac{\partial\left(\sum\limits_{k=1}^{d} C_{ik}\right)}{\partial X_j} = \frac{\partial C_{ij}}{\partial X_j}$$

(Частная производная X)

$$\frac{\partial L}{\partial X_j} = \sum_{i=1}^{c}\frac{\partial L}{\partial Y_i}\frac{\partial Y_i}{\partial X_j} = \sum_{i=1}^{c}\frac{\partial L}{\partial Y_i}\frac{\partial C_{ij}}{\partial X_j} =$$

$$= \sum_{i=1}^{c}\frac{\partial L}{\partial Y_i}\frac{\partial\left(X_j \overset{(p)}{*} K_{ij}\right)}{\partial X_j} = \sum_{i=1}^{c}\frac{\partial L}{\partial Y_i} \overset{(p')}{*} \widehat{K}_{ij}$$

(Обратный проход X)

Итого (в тензорном виде)

$$\frac{\partial L}{\partial B} = \sum_{hw}\frac{\partial L}{\partial Y} \qquad \frac{\partial L}{\partial K} = \text{convolve}_p^{*}\left(\widehat{X}^T, \frac{\partial L}{\partial Y}^T\right)^T \qquad \frac{\partial L}{\partial X} = \text{convolve}_{p'}\left(\frac{\partial L}{\partial Y}, \widehat{K}^T\right)$$

*При вычислении $\frac{\partial L}{\partial K}$, функция `convolve` используется не совсем по назначению, так как вместо суммирования по каналам происходит суммирования по размерности батча.