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 — 15 баллов. Еще 5 бонусных баллов можно получить за выполнение задания на семинаре.

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

Список разделов и максимальных баллов

Раздел	Название	Баллы
2.1.1	`ReLU`	0.5
2.1.2	`Softmax`	1.5
2.1.3	`Dense`	1.5
2.2.1	`CrossEntropy`	0.5
2.3.1	`SGD`	0.5
2.3.2	`SGDMomentum`	0.5
2.4	`MNIST`	1.0
3.1	`Conv2D`	1.5
3.2	`MaxPooling2D`	1.0
3.3	`GlobalAveragePooling`	1.0
3.4	`BatchNormalization`	2.0
3.5	`Dropout`	1.0
3.6	`Flatten`	0.5
3.7	`CIFAR-10`	2.0

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/
    └── ...
```

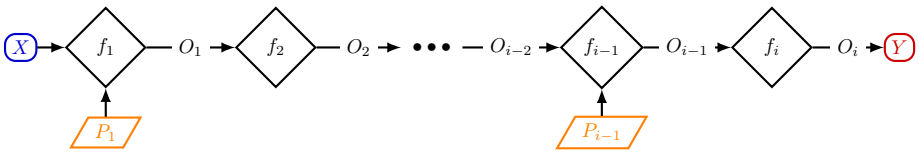
1 Принципы работы многослойных нейросетей

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.

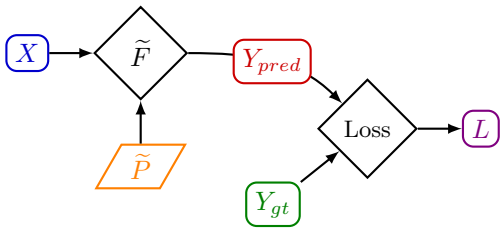
1.2 Обучение с учителем

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.

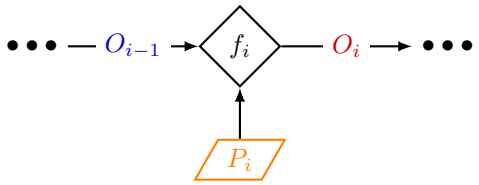
Если в батче больше одной пары, частная производная $\dfrac{\partial L}{\partial \widetilde{P}}$ просто усредняется по всем элементам батча. Это усреднение позволяет сгладить случайную природу выбора эталонных пар из всех данных для формирования батчей. Последовательно выбирая разные батчи и повторяя данную операцию для *всех* эталонных пар, мы сможем приблизительно найти локальный минимум функции потерь.

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

1.4 Обратное распространение ошибки

Суть метода обратного распространения ошибки заключается в последовательном применении цепного правила дифференцирования сложной функции. Для каждого слоя реализуются “прямой” и

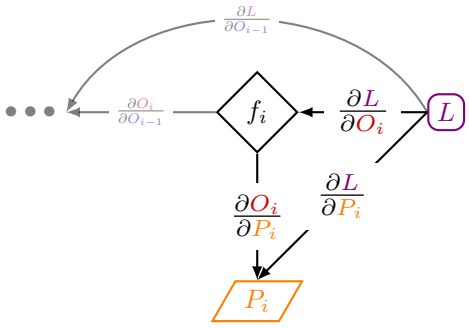
“обратный” вычислительные проходы. Рассмотрим один слой из нейросети.



Прямой проход

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

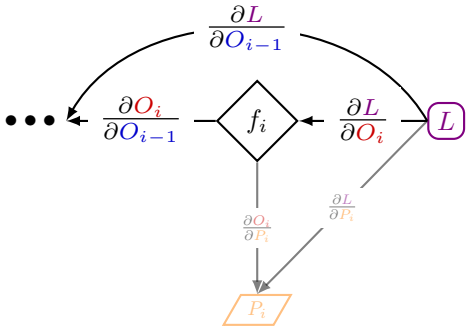
При обратном проходе индуктивно предполагается, что уже известна производная $\frac{\partial L}{\partial O_i}$.



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 Реализация полносвязанной многослойной нейросети

2.1 Реализация основных слоев

В этой части задания вам будет необходимо реализовать прямые и обратные проходы для некоторых слоев, часто используемых в многослойных полносвязанных нейросетях. От вас требуется заполнить в файле `solution.py` фрагменты кода, помеченные `# Your code here`. В файле `interface.py` находятся шаблоны абстрактных классов 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.

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` и `backward`.

Функция `backward` принимает на вход $\frac{\partial L}{\partial O_i}$ и возвращает $\frac{\partial L}{\partial O_{i-1}}$. Если у данного слоя есть обучаемые параметры, то функция `backward` 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`

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

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

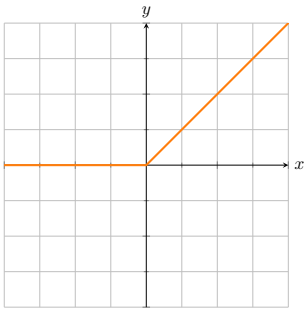


График ReLU для одного элемента

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`

$$x \in \mathbb{R}^d$$

$$y \in [0,1]^d$$

$$y = \text{Softmax}\,(x) = \left\{ y_i = \frac{e^{x_i}}{\sum\limits_{j=1}^{d} e^{x_j}} \right\}_{i=1..d}$$

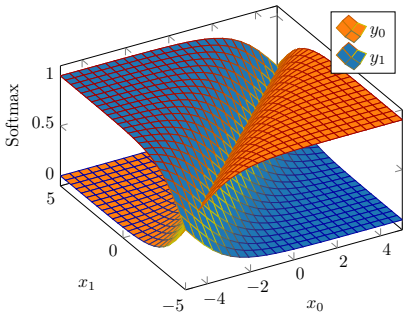


График Softmax для $d = 2$

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 $\sum_{i=1}^{d} y_i = 1$.

Выведем формулы подсчета производной слоя при обратном проходе.

$$\frac{\partial y_i}{x_k} = \frac{\partial \left(\dfrac{e_i^x}{\sum_{j=1}^{d} e_j^x} \right)}{\partial x_k}$$

При $i = k$:

$$\frac{\partial y_i}{x_k} = \frac{e_i^x \left(\sum_{j=1}^{d} e_j^x \right) - e_i^x e_k^x}{\left(\sum_{j=1}^{d} e_j^x \right)^2} = \frac{e_i^x}{\sum_{j=1}^{d} e_j^x} \frac{\sum_{j=1}^{d} e_j^x - e_k^x}{\sum_{j=1}^{d} e_j^x} = y_i(1 - y_k).$$

При $i \neq k$:

$$\frac{\partial y_i}{x_k} = \frac{-e_i^x e_k^x}{\left(\sum_{j=1}^{d} e_j^x \right)^2} = -y_i y_k.$$

Итого $\frac{\partial y_i}{x_k} = y_i(\delta_{ik} - y_k)$.

Теперь выведем полную формулу для обратного прохода, используя правило дифференцирования сложной функции в многомерном случае.

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} y_i(\delta_{ij} - y_j) = \frac{\partial L}{\partial y_j} y_j - \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} y_i y_j.$$

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`

$$x \in \mathbb{R}^c$$
$$y \in \mathbb{R}^d$$

$$y = Wx + b = \left\{ y_i = \sum_{j=1}^{c} w_{ij} x_j + b_i \right\}_{i=1..d}$$

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

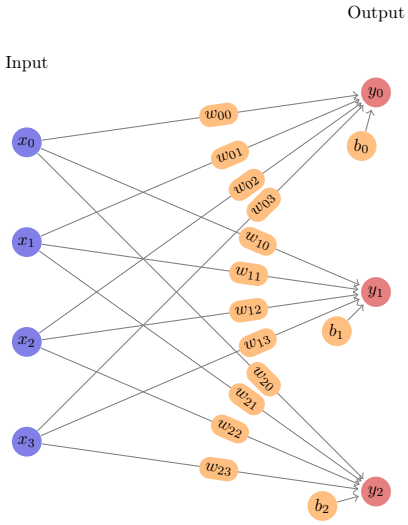


Схема Dense слоя для $c = 4$ и $d = 3$

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

Формулы необходимых частных производных:

$$\frac{\partial y_i}{\partial b_j} = \delta_{ij}, \quad \frac{\partial y_i}{\partial x_j} = w_{ij}, \quad \frac{\partial y_i}{\partial w_{jk}} = \frac{\partial \left(\sum_{l=1}^{c} w_{il} x_l \right)}{\partial w_{jk}} = \delta_{ij} x_k.$$

Используя их, выведем формулы обратного прохода:

$$\frac{\partial L}{\partial b_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial b_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \delta_{ij} = \frac{\partial L}{\partial y_j},$$

$$\frac{\partial L}{\partial w_{jk}} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial w_{jk}} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \delta_{ij} x_k = \frac{\partial L}{\partial y_j} x_k,$$

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^{d} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_{i=0}^{d} \frac{\partial L}{\partial y_i} w_{ij}.$$

Используя эти выражения, запишем формулы обратного прохода в векторном виде:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}, \quad \frac{\partial L}{\partial W} = \frac{\partial L}{\partial y} x^T, \quad \frac{\partial L}{\partial x} = W^T \frac{\partial L}{\partial y}.$$

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`.

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

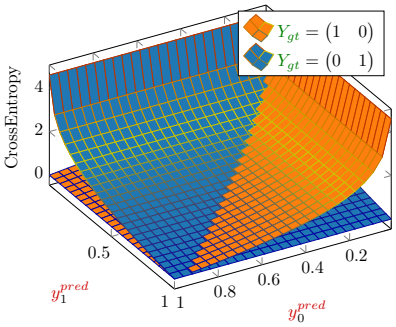
Вам дан интерфейс класса `Loss`, вам нужно реализовать вычисление значения функции (`__call__`) и ее производной (`gradient`).

2.2.1 `CrossEntropy`

$$L \in \mathbb{R}^+$$

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

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



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.

Производная кросс-энтропии:

$$\frac{\partial L}{\partial y_i} = -\frac{y_i^{gt}}{y_i^{pred}}.$$

Проверьте реализацию с помощью юнит-теста:

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

2.3 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.

2.3.1 `SGD`

Реализуйте стратегию обновления параметров стохастического градиентного спуска. (см. раздел 1.3)

Проверьте реализацию с помощью юнит-теста:

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

2.3.2 `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
```

2.4 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
```

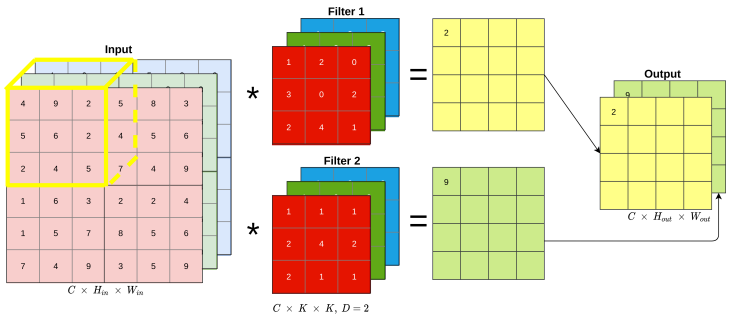
3 Реализация сверточной многослойной нейросети

В этой части задания вам будет необходимо реализовать прямые и обратные проходы для основных слоев, используемых в сверточных нейронных сетях, а также обучить сверточную нейронную сеть на датасете `CIFAR-10`.

3.1 `Conv2D`

3.1.1 Описание слоя

Conv2D, Convolutional layer, сверточный слой. Слой, в основе которого лежит операция свертки. Сверточные слои также состоят из нейронов, которые состоят из весов ядра свертки и весов сдвига.



Пример работы операции свертки

Сверточный слой на вход принимает тензор размером $C \times W_{in} \times H_{in}$. Здесь C — число входных карт признаков, W_{in} и H_{in} — ширина и высота входных карт признаков. Слой задается следующими параметрами:

1. K — размер ядра свертки. Для простоты рассматриваем только квадратные свертки, хотя на практике могут применяться и прямоугольные.
2. D — число сверток в слое.
3. S — шаг свертки (stride). Будем считать, что $S_x = S_y = S$, хотя на практике могут использоваться разные размеры шага вдоль разных осей.
4. P — величина дополнения входного тензора по краям (padding). Тензор, как правило, дополняется нулями. Это делается для эффективности реализации. Для подсчета величины P в библиотеках часто используются режимы:

   `same` — входной тензор дополняется так, чтобы размеры входного и выходного тензора были равны,

   `valid` — дополнение входного тензора не делается, $P = 0$.

На выходе из сверточного слоя получается тензор размера $D \times W_{out} \times H_{out}$. Ширину и высоту карт признаков на выходе можно вычислить по формулам

$$W_{out} = \left\lfloor \frac{W_{in} - K + 2P}{S} \right\rfloor + 1 \qquad H_{out} = \left\lfloor \frac{H_{in} - K + 2P}{S} \right\rfloor + 1.$$

Число параметров сверточного слоя задается формулой

$$D \cdot (C \cdot K \cdot K + 1).$$

У каждой свертки есть C ядер размером $K \times K$ (по одному ядру для каждого слоя) и одно смещение (bias). Всего сверток в слое — D штук.

Формально рассматриваемая нами операция называется кросс-корреляция, а не свертка. При выполнении операции свертки фильтр должен быть предварительно повернут на 180°. При обучении нейронных сетей веса в фильтрах настраиваются автоматически, поэтому выбор операции значения не имеет.

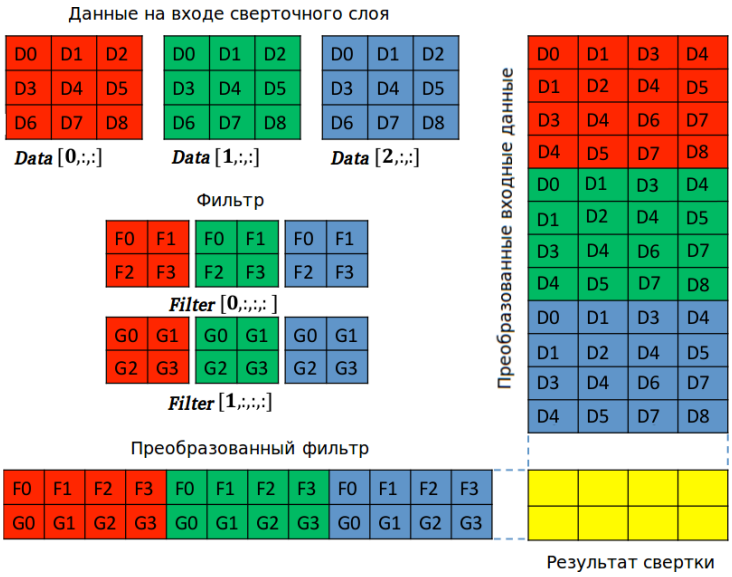
3.1.2 Прямой проход

Самый простой способ реализовать операцию свертки — умножать каждый фрагмент, извлеченный из изображения (со всеми каналами), на фильтр внутри двух вложенных циклов (по высоте и ширине). Для более эффективной реализации на графических ускорителях от операцию свертки реализуют с помощью матричного умножения. Алгоритм выглядит следующим образом:

1. **Преобразование входных данных**. Входные тензоры дополняются P элементами по высоте и ширине. Затем из них вырезаются фрагменты размером $K \times K$ с шагом S. Каждый фрагмент вытягивается в вектор-столбец. Вектор-столбы, соответствующие разным каналам одного фрагмента, конкатенируются по вертикали. Вектор-столбы, соответствующие последовательно извлеченным фрагментам, конкатенируются горизонтально. Длина вектора для одного фрагмента всех каналов — CK^2. Всего таких фрагментов $N = W_{out} \cdot H_{out}$. В итоге получим матрицу X_{col} размером $CK^2 \times N$. Поскольку фрагменты входного тензора пересекаются друг с другом, то элементы матрицы X_{col} будут повторяться. Описанная операция называется `im2col`.
2. **Преобразование весов сверток**. Ядра сверток вытягиваются в вектор-строки. Ядра, соответствующие разным каналам одной свертки, конкатенируются горизонтально. Получившиеся вектор-строки конкатерируются вертикально. В результате получается матрица W_{row} размером $D \times CK^2$.
3. **Матричное умножение**. Перемножив матрицы W_{row} и X_{col}, получим матрицу размером $D \times N$. Остается преобразовать матрицу в трехмерный тензор размером $D \times W_{out} \times H_{out}$. Полученный тензор и будет являться результатом свертки.

Минусом данного алгоритма свертки являются накладные расходы на хранение повторяющихся фрагментов исходного тензора в матрице X_{col}. Но ускорение свертки, получаемое за счет использования эффективной реализации перемножения матриц, перевешивает повышенные требования к оперативной памяти.

На рисунке показан пример преобразования данных с помощью операции `im2col` для сверточного слоя с параметрами $K = 2,\ D = 2,\ S = 1,\ P = 0$. Входной тензор имеет размеры $3 \times 3 \times 3$.



3.1.3 Обратный проход

Использование матричного умножения в прямом проходе напоминает работу полносвязного слоя. Можно считать, что одна итерация свертки (взвешенное суммирование элементов в одной области входного тензора) — это вычисление выхода одного нейрона. Сверточный слой по сравнению с полносвязным слоем имеет две особенности:

1. Для подсчета выхода нейрона используются не все входные значения, а только небольшая их часть, сосредоточенная в небольшом окне. Можно считать, что веса при остальных входных значениях равны 0.

2. Веса разных нейронов общие.

Таким образом, обратный проход сверточного слоя можно считать по формулам для полносвязного слоя. При этом после подсчета градиента $\frac{\partial L}{\partial x}$ его нужно преобразовать в тензор с помощью операции `col2im`, которая является обратной к операции `im2col`.

3.1.4 Реализация

Вам даны заготовки следующих функций:

`im2col` — преобразование тензора в колоночный вид,
`get_im2col_indices` — подсчет индексов элементов для колоночного преобразования,
`col2im` — преобразование тензора из колоночного вида,

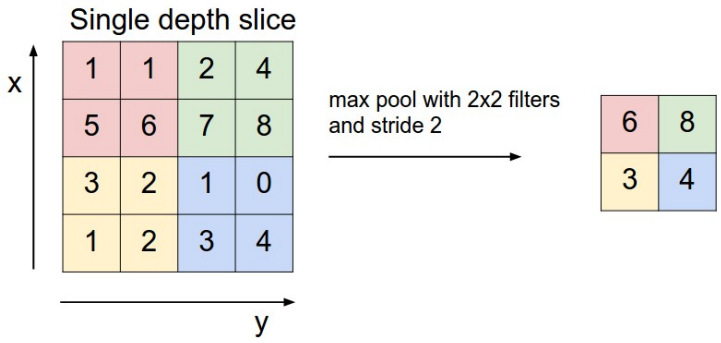
а также вспомогательная функция `determine_padding`, определяющая размер дополнения тензора P в зависимости от режима (`same` или `valid`). Реализуйте функции по их заготовкам. Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest im2col
```

Теперь от вас требуется реализовать прямой и обратный проход сверточного слоя. При этом предполагается, что ядро квадратное и шаг свертки (stride) может быть равен только 1. Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest conv
```

3.2 `MaxPooling2D`



MaxPooling2D, Maximum Pooling layer, слой подвыборки — помогает снизить размеры входных в него тензоров. На вход слой принимает тензор размера $C \times W_{in} \times H_{in}$. Параметры слоя:

1. K — размер окна. Для простоты рассматриваем только квадратные окна.
2. S — шаг окна (stride). Будем считать, что шаг окна по каждой оси одинаков: $S_x = S_y = S$.

Для каждого положения окна и для каждого слоя входного тензора применяется операция максимума. Результат работы слоя — тензор размера $C \times W_{out} \times H_{out}$, где

$$W_{out} = \left\lfloor \frac{W_{in} - K}{S} \right\rfloor + 1 \qquad H_{out} = \left\lfloor \frac{H_{in} - K}{S} \right\rfloor + 1.$$

При обратном проходе градиент учитывается только в тех элементах, в которых был взят максимум. При прямом проходе необходимо сохранять индексы максимальных элементов.

Вам необходимо реализовать слой MaxPooling2D только для $K = 2$ и $S = 2$. Этот слой можно реализовать с помощью циклов или операции `im2col`. Проверьте реализацию с помощью юнит-теста:

```
$ ./run.py unittest maxpool
```

3.3 GlobalAveragePooling

GlobalAveragePooling — слой подвыборки, который выполняет усреднение по каждому каналу тензора. На выход слой получает тензор размера $C \times W \times H$ и выдает тензор размера C.

Реализацию слоя можно проверить с помощью юнит-теста:

```
$ ./run.py unittest avgpool
```

3.4 BatchNormalization

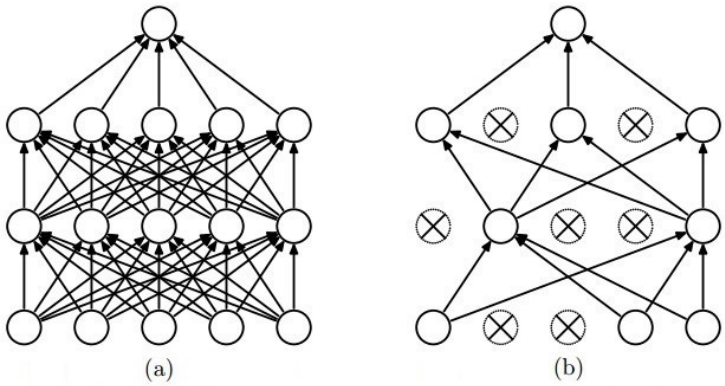
Batch Normalization, пакетная нормализация — слой, который позволяет повысить точность и стабилизировать процесс обучения нейронных сетей. Подробное описание и формулы прямого и обратного проходов слоя можно найти в статье авторов.

Обратите внимание, что на этапе тестирования не нужно обновлять значения `running_mean` и `running_var`, которые являются накопленными средними и дисперсиями только для тренировочной выборки.

Проверьте реализацию с помощью юнит-теста:

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

3.5 Dropout



Dropout — метод регуляризации нейронных сетей. Слой принимает на вход тензор и в процессе обучения с заданной вероятностью p обнуляет значения элементов тензора. При обратном проходе градиент по обнуленным элементам также равняется нулю. Слой дропаут используется после слоев с большим количеством параметров (в первую очередь полносвязных) для уменьшения склонности нейросети к переобучению.

На этапе тестирования элементы не обнуляются, а домножаются на величину $(1 - p)$. Это делается для того, чтобы сумма активаций нейронов при тестировании была в среднем такой же, как и при обучении.

Проверьте реализацию с помощью юнит-теста. При проверке раскомментируйте две строчки кода в методе `build`, фиксирующие значение `seed`.

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

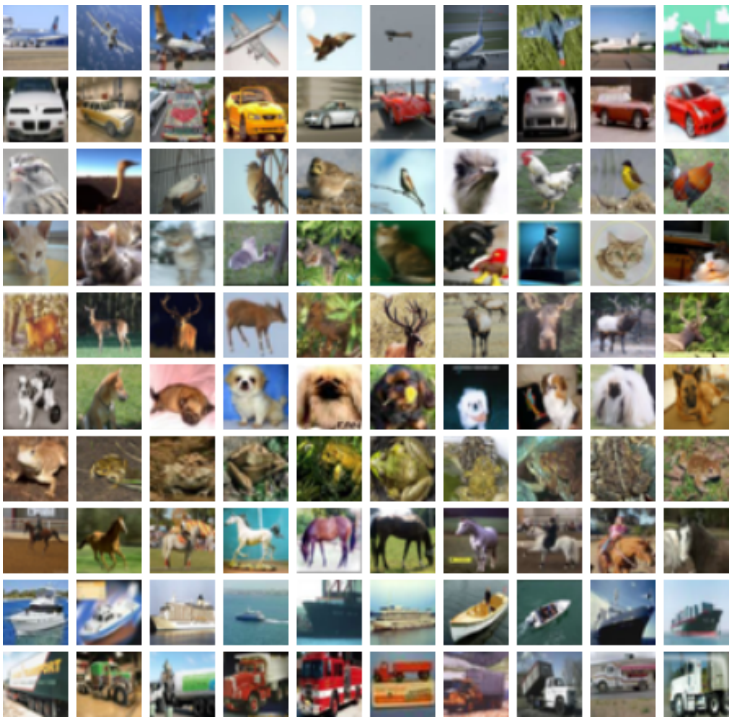
3.6 `Flatten`

Flatten — вспомогательный слой, который преобразует тензор произвольного размера $D_1 \times \cdots \times D_n$ в тензор размера $D_1 D_2 \cdots D_n$. Слой используется перехода от сверточных слоев к полносвязным.

Проверьте реализацию с помощью юнит-теста:

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

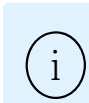
3.7 Обучение сверточной нейросети на `CIFAR-10`



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 cifar
```

При обучении глубокой сети у вас могут возникать ошибки, связанные с переполнением в функции `forward` слоя Softmax. Чтобы их избежать можно, например, перед подсчетом экспонент вычитать максимальное значение тензора из всех его элементов. Значения Softmax при этом не поменяются.