Обучение многослойных нейросетей
(на `numpy`)

Андрей Стоцкий, Влад Шахуро

Введение

В этом задании вы:

- изучите методы стохастического градиентного спуска и обратного распространения ошибки
- реализуете прямой и обратный проходы многослойной нейросети
- обучите полносвязанную нейросеть для классификации рукописных цифр (`MNIST`)

Критерии оценки

Максимальная оценка за задание — 5 баллов. 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`	1.0
2.1.3	`Dense`	1.0
2.2.1	`CrossEntropy`	0.5
2.3.1	`SGD`	0.5
2.3.2	`SGDMomentum`	0.5
2.4	`MNIST`	1.0

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.

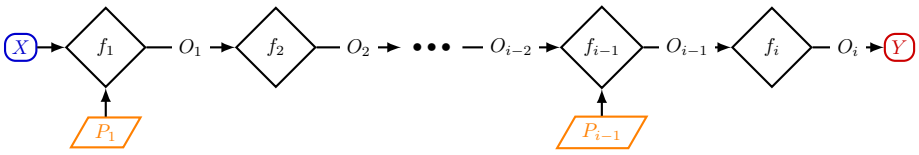

Рис. 1: Вычислительный граф

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. То есть,

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

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.

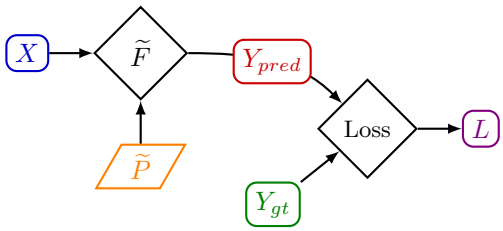

Рис. 2: Функция потерь

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.

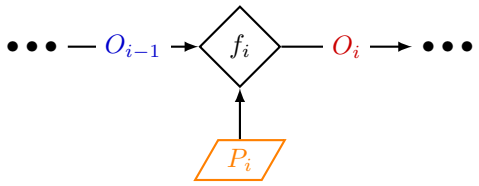

Рис. 3: Прямой проход

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

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

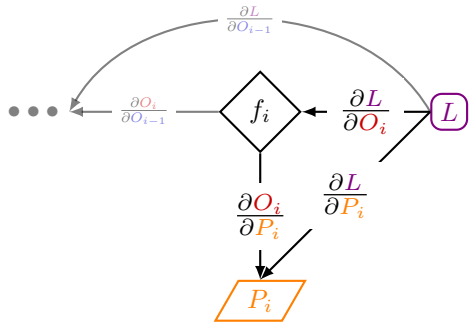

Рис. 4: Обратный проход (Параметры)

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

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

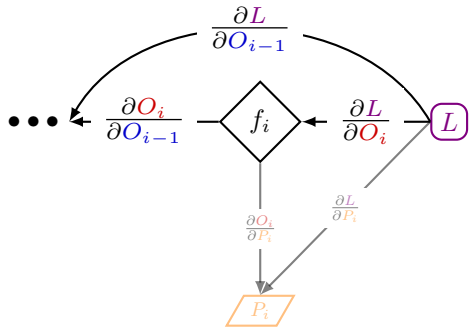

Рис. 5: Обратный проход (Входы)

Также, для того чтобы сохранить предположение индукции для $i-1$го слоя, вычислим

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

При этом, стоит обратить 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 про-

ходы 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, 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). При этом, все слои оперируют сразу над целыми батчами значений, так что 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`

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

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

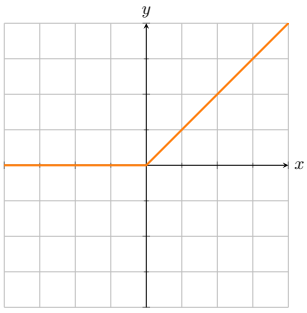

Рис. 6: График ReLU для одного элемента

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`

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

$$Y = \mathrm{Softmax}(X) = \left\{ y_i = \frac{e^{x_i}}{\sum_{j=0}^{d-1} e^{x_j}} \right\}_{i=0..d-1}$$

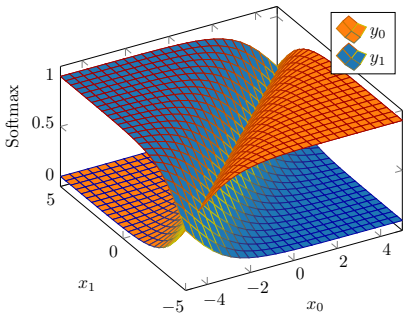

Рис. 7: График Softmax для $d = 2$

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 $0 \leq y_i \leq 1$ и $\sum_{i=0}^{d-1} y_i = 1$.

Обратите внимание, что каждый выход y_i зависит от всех входов x_j, даже для $j \neq i$, а значит и каждая производная $\frac{\partial L}{\partial x_j}$ будет зависеть от всех $\frac{\partial L}{\partial y_i}$. Если данный факт вам не кажется очевидным, обратите внимание на правила цепного дифференцирования сложной функции в многомерном случае для частных производных.

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`

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

$$Y = W \cdot X + B = \left\{ y_i = \sum_{j=0}^{d-1} (w_{ij} x_j) + b_i \right\}_{i=0..c-1}$$

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

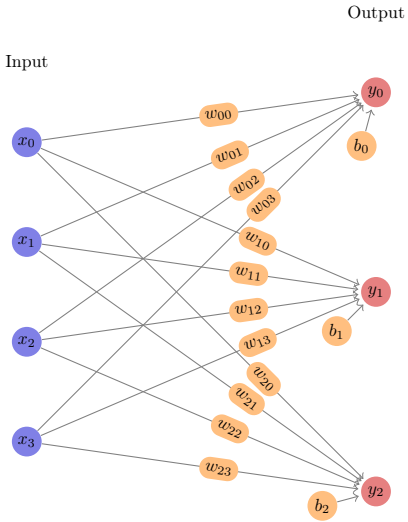

Рис. 8: Схема Dense слоя

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`.

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 $\mu = 0$, $\sigma = \sqrt{\frac{2}{d}}$. 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`

$$L \in \mathbb{R}^+$$
$$Y_{pred},\ Y_{gt} \in [0,\ 1]^d$$

$$L = \text{CrossEntropy}\left(Y_{gt},\ Y_{pred}\right) = -\sum_{i=0}^{d-1} y_i^{gt} \cdot \ln\left(y_i^{pred}\right)$$

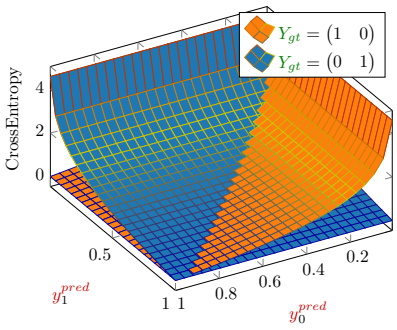

Рис. 9: Графики CrossEntropy для d=2

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.

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 (`MNIST`)

Рис. 10: Примеры данных из датасета `MNIST`

В этой части вам необходимо обучить вашу 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
```