Оглавление

Предисловие **8**

Для понимания нейронных сетей нужно несколько мысленных моделей 10

Структура книги 11

Условные обозначения 13

Использование примеров кода 14

Благодарности 14

От издательства 15

Глава 1. Математическая база 16

Функции 17

Производные 22

Вложенные функции 24

Цепное правило 26

Более длинная цепочка 30

Функции нескольких переменных 34

Производные функций нескольких переменных 36

Функции нескольких переменных с векторными аргументами 37

Создание новых признаков из уже существующих 38

Производные функции нескольких векторных переменных 41

Производные векторных функций: продолжение 43

Вычислительный граф для двух матриц 47

Самое интересное: обратный проход 51

Заключение 58

Глава 2. Основы глубокого обучения 59

Обучение с учителем 60

Алгоритмы обучения с учителем 62

Линейная регрессия 63
Обучение модели 69
Оценка точности модели 73
Код 74
Основы нейронных сетей 79
Обучение и оценка нейронной сети 86
Заключение 90

Глава 3. Основы глубокого обучения 91
Определение глубокого обучения: первый проход 91
Строительные блоки нейросети: операции 93
Строительные блоки нейросети: слои 97
Блочное строительство 100
Класс NeuralNetwork и, возможно, другие 107
Глубокое обучение с чистого листа 111
Trainer и Optimizer 115
Собираем все вместе 119
Заключение и следующие шаги 122

Глава 4. Расширения 123
Немного о понимании нейронных сетей 124
Многопеременная логистическая функция активации с перекрестно-энтропийными потерями 126
Эксперименты 135
Импульс 138
Скорость обучения 142
Инициализация весов 145
Исключение, или дропаут 149
Заключение 153

Глава 5. Сверточная нейронная сеть 155
Нейронные сети и обучение представлениям 155
Слои свертки 160

Реализация операции многоканальной свертки ... 167
Свертка: обратный проход ... 171
Использование операции для обучения CNN ... 184
Заключение ... 188

Глава 6. Рекуррентные нейронные сети ... 190
Ключевое ограничение: работа с ветвлениями ... 191
Автоматическое дифференцирование ... 194
Актуальность рекуррентных нейронных сетей ... 199
Введение в рекуррентные нейронные сети ... 201
RNN: код ... 209
Заключение ... 230

Глава 7. Библиотека PyTorch ... 231
Класс PyTorch Tensor ... 231
Глубокое обучение с PyTorch ... 233
Сверточные нейронные сети в PyTorch ... 242
P. S. Обучение без учителя через автокодировщик ... 251
Заключение ... 261

Приложение A. Глубокое погружение ... 262
Цепное правило ... 262
Градиент потерь с учетом смещения ... 266
Свертка с помощью умножения матриц ... 266

Об авторе ... 272

Об обложке ... 272

Глубокое обучение с чистого листа

В конечном итоге мы хотим создать класс `NeuralNetwork`, как на рис. 3.5, и использовать эту нейросеть для моделей глубокого обучения. Прежде чем мы начнем писать код, давайте точно опишем, каким будет этот класс и как он будет взаимодействовать с классами `Operation`, `Layer` и `Loss`, которые мы только что определили:

1. `NeuralNetwork` будет в качестве атрибута получать список экземпляров `Layer`. Слои будут такими, как было определено ранее — с прямым и обратным методами. Эти методы принимают объекты `ndarray` и возвращают объекты `ndarray`.
2. Каждый `Layer` будет иметь список операций `Operation`, сохраненный в атрибуте `operations` слоя функцией `_setup_layer`.
3. Эти операции, как и сам слой, имеют методы прямого и обратного преобразования, которые принимают в качестве аргументов объекты `ndarray` и возвращают объекты `ndarray` в качестве выходных данных.
4. В каждой операции форма `output_grad`, полученная в методе `backward`, должна совпадать с формой выходного атрибута `Layer`. То же самое верно для форм `input_grad`, передаваемых в обратном направлении методом `backward` и атрибутом `input_`.
5. Некоторые операции имеют параметры (которые хранятся в атрибуте `param`). Эти операции наследуют от класса `ParamOperation`. Те же самые ограничения на входные и выходные формы применяются к слоям и их методам `forward` и `backward` — они берут объекты `ndarray`, и формы входных и выходных атрибутов и их соответствующие градиенты должны совпадать.
6. У класса `NeuralNetwork` также будет класс `Loss`. Этот класс берет выходные данные последней операции из `NeuralNetwork` и цели, проверяет, что их формы одинаковы, и, вычисляя значение потерь (число) и `ndarray loss_grad`, которые будут переданы в выходной слой, начинает обратное распространение.

Реализация пакетного обучения

Мы уже говорили о шагах обучения модели по одной партии за раз. Повторим:

1. Подаем входные данные через функцию модели («прямой проход») для получения прогноза.
2. Рассчитываем потери.
3. Вычисляем градиенты потерь по параметрам с использованием цепного правила и значений, вычисленных во время прямого прохода.
4. Обновляем параметры на основе этих градиентов.

Затем мы передаем новый пакет данных и повторяем эти шаги.

Перенести эти шаги платформу `NeuralNetwork` очень просто:

1. Получаем объекты `ndarray` `X` и `y` в качестве входных данных.
2. Передаем `X` по слоям.
3. Используем `Loss` для получения значения потерь и градиента потерь для обратного прохода.
4. Используем градиент потерь в качестве входных данных для метода `backward`, вычисления `param_grads` для каждого слоя в сети.
5. Вызываем на каждом слое функцию `update_params`, которая будет брать скорость обучения для `NeuralNetwork`, а также только что рассчитанные `param_grads`.

Наконец, у нас есть полное определение нейронной сети, на которой можно выполнять пакетное обучение. Теперь напишем код.

Нейронная сеть: код

Код выглядит весьма просто:

```
class NeuralNetwork(object):
    '''
    Класс нейронной сети.
    '''
    def __init__(self, layers: List[Layer],
                 loss: Loss,
                 seed: float = 1)
        '''
        Нейросети нужны слои и потери.
        '''
```

```
        self.layers = layers
        self.loss = loss
        self.seed = seed
        if seed:
            for layer in self.layers:
                setattr(layer, "seed", self.seed)

    def forward(self, x_batch: ndarray) -> ndarray:
        '''
        Передача данных через последовательность слоев.
        '''
        x_out = x_batch
        for layer in self.layers:
            x_out = layer.forward(x_out)

        return x_out

    def backward(self, loss_grad: ndarray) -> None:
        '''
        Передача данных назад через последовательность слоев.
        '''
        grad = loss_grad
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

        return None

    def train_batch(self,
                    x_batch: ndarray,
                    y_batch: ndarray) -> float:
        '''
        Передача данных вперед через последовательность слоев.
        Вычисление потерь.
        Передача данных назад через последовательность слоев.
        '''

        predictions = self.forward(x_batch)

        loss = self.loss.forward(predictions, y_batch)

        self.backward(self.loss.backward())
```

```
        return loss

    def params(self):
        '''
        Получение параметров нейросети.
        for layer in self.layers:
            yield from layer.params

    def param_grads(self):
        '''
        Получение градиента потерь по отношению к параметрам нейросети.
        '''
        for layer in self.layers:
            yield from layer.param_grads
```

С помощью этого класса `NeuralNetwork` мы можем реализовать модели из предыдущей главы модульным, гибким способом и определить другие модели для представления сложных нелинейных отношений между входом и выходом. Например, ниже показано, как создать две модели, которые мы рассмотрели в предыдущей главе: линейную регрессию и нейронную сеть[1]:

```
linear_regression = NeuralNetwork(
   layers=[Dense(neurons = 1)],
           loss = MeanSquaredError(),
           learning_rate = 0.01
           )

neural_network = NeuralNetwork(
   layers=[Dense(neurons=13,
                 activation=Sigmoid()),
           Dense(neurons=1,
                 activation=Linear())],
           loss = MeanSquaredError(),
           learning_rate = 0.01
           )
```

Почти готово. Теперь мы просто многократно передаем данные через сеть, чтобы модель начала учиться. Но чтобы сделать этот процесс понятнее и проще в реализации для более сложных сценариев глубокого обучения,

[1] Скорость обучения 0.01 найдена оптимальной путем экспериментов.

надо определить другой класс, который будет выполнять обучение, а также дополнительный класс, который выполняет «обучение» или фактическое обновление параметров `NeuralNetwork` с учетом градиентов, вычисленных при обратном проходе. Давайте определим эти два класса.

Trainer и Optimizer

Есть сходство между этими классами и кодом, который мы использовали для обучения сети в главе 2. Там для реализации четырех шагов, описанных ранее для обучения модели, мы использовали следующий код:

```
# Передаем X_batch вперед и вычисляем потери
forward_info, loss = forward_loss(X_batch, y_batch, weights)

# Вычисляем градиент потерь по отношению к каждому весу
loss_grads = loss_gradients(forward_info, weights)

# обновляем веса
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

Этот код находился внутри цикла `for`, который неоднократно передавал данные через функцию, определяющую и обновляющую нашу сеть.

Теперь, когда у нас есть нужные классы, мы, в конечном счете, сделаем это внутри функции подгонки в классе `Trainer`, который в основном будет оберткой вокруг функции `train`, использованной в предыдущей главе. (Полный код для этой главы в можно найти на странице книги на GitHub.) Основное отличие состоит в том, что внутри этой новой функции первые две строки из предыдущего блока кода будут заменены этой строкой:

```
neural_network.train_batch (X_batch, y_batch)
```

Обновление параметров, которое происходит в следующих двух строках, будет происходить в отдельном классе `Optimizer`. И наконец, цикл `for`, который ранее охватывал все это, будет выполняться в классе `Trainer`, который оборачивает `NeuralNetwork` и `Optimizer`.

Далее обсудим, почему нам нужен класс `Optimizer` и как он должен выглядеть.

Optimizer

В модели, которую мы описали в предыдущей главе, у слоев есть простое правило для обновления весов на основе параметров и их градиентов. В следующей главе мы узнаем, что есть множество других правил обновления. Например, некоторые правила учитывают данные не только от текущего набора данных, но и от всех предыдущих. Создание отдельного класса `Optimizer` даст нам гибкость в замене одного правила обновления на другое, что мы более подробно рассмотрим в следующей главе.

Описание и код

Базовый класс `Optimizer` будет принимать `NeuralNetwork`, и каждый раз, когда вызывается пошаговая функция `step`, будет обновлять параметры сети на основе их текущих значений, их градиентов и любой другой информации, хранящейся в `Optimizer`:

```
class Optimizer(object):
    '''
    Базовый класс оптимизатора нейросети.
    '''
    def __init__(self,
        lr: float = 0.01):
        '''
        У оптимизатора должна быть начальная скорость обучения.
        '''
        self.lr = lr


    def step(self) -> None:
        '''
        У оптимизатора должна быть функция "step".
        '''
        pass
```

И вот как это выглядит с простым правилом обновления (*стохастический градиентный спуск*):

```
class SGD(Optimizer):
    '''
    Стохастический градиентный оптимизатор.
    '''
    def __init__(self,
```

```
                lr: float = 0.01) -> None:
        '''пока ничего'''
        super().__init__(lr)

    def step(self):
        '''
        Для каждого параметра настраивается направление, при этом
        амплитуда регулировки зависит от скорости обучения.
        '''
        for (param, param_grad) in zip(self.net.params(),
                                       self.net.param_grads()):

            param -= self.lr * param_grad
```

Обратите внимание, что хотя наш класс `NeuralNetwork` не имеет метода `_update_params`, мы используем методы `params()` и `param_grads()` для извлечения правильных `ndarrays` для оптимизации.

Класс `Optimizer` готов, теперь нужен `Trainer`.

Trainer

Помимо обучения модели класс `Trainer` также связывает `NeuralNetwork` с `Optimizer`, гарантируя правильность обучения. Вы, возможно, заметили в предыдущем разделе, что мы не передавали нейронную сеть при инициализации нашего оптимизатора. Вместо этого мы назначим `NeuralNetwork` атрибутом `Optimizer` при инициализации класса `Trainer`:

```
setattr(self.optim, 'net', self.net)
```

В следующем подразделе я покажу упрощенную, но рабочую версию класса `Trainer`, которая пока содержит только метод `fit`. Этот метод выполняет несколько эпох обучения и выводит значение потерь после некоторого заданного числа эпох. В каждую эпоху мы будем:

- перемешивать данные в начале эпохи;
- передавать данные через сеть в пакетном режиме, обновляя параметры.

Эпоха заканчивается, когда мы пропускаем весь обучающий набор через `Trainer`.

Код класса Trainer

Ниже приведен код простой версии класса `Trainer`. Мы скрываем два вспомогательных метода, использующихся во время выполнения функции `fit`: `generate_batches`, генерирующий пакеты данных из `X_train` и `y_train` для обучения, и `per mute_data`, перемешивающий `X_train` и `y_train` в начале каждой эпохи. Мы также включили аргумент `restart` в функцию `train`: если он имеет значение `True` (по умолчанию), то будет повторно инициализировать параметры модели в случайные значения при вызове функции `train`:

```
class Trainer(object):
    '''
    Обучение нейросети.
    '''
    def __init__(self,
                 net: NeuralNetwork,
                 optim: Optimizer)
        '''
        Для обучения нужны нейросеть и оптимизатор. Нейросеть
         назначается атрибутом экземпляра оптимизатора.
        '''
        self.net = net
        setattr(self.optim, 'net', self.net)

    def fit(self, X_train: ndarray, y_train: ndarray,
            X_test: ndarray, y_test: ndarray,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32,
            seed: int = 1,
            restart: bool = True) -> None:
        '''
        Подгонка нейросети под обучающие данные за некоторое число
        эпох. Через каждые eval_every эпох выполняется оценка.
        '''
        np.random.seed(seed)

        if restart:
            for layer in self.net.layers:
                layer.first = True
```

```
        for e in range(epochs):

            X_train, y_train = permute_data(X_train, y_train)

            batch_generator = self.generate_batches(X_train, y_train,
                                                    batch_size)

            for ii, (X_batch, y_batch) in enumerate(batch_generator):

                self.net.train_batch(X_batch, y_batch)

                self.optim.step()

            if (e+1) % eval_every == 0:

                test_preds = self.net.forward(X_test)

                loss = self.net.loss.forward(test_preds, y_test)

                print(f"Validation loss after {e+1} epochs is
                     {loss:.3f}")
```

В полной версии этой функции в хранилище GitHub книги (https://oreil.ly/2MV0aZI) мы также реализовали *раннюю остановку*, которая выполняет следующие действия:

1. Сохраняет значение потерь каждые `eval_every` эпох.
2. Проверяет, стали ли потери меньше после настройки.
3. Если потери не стали ниже, модель откатывается на шаг назад.

Теперь у нас есть все необходимое для обучения этих моделей!

Собираем все вместе

Ниже приведен код для обучения сети с использованием всех классов `Trainer` и `Optimizer` и двух моделей, определенных ранее, — `linear_regression` и `neural_network`. Мы установим скорость обучения равной 0.01, максимальное количество эпох — 50 и будем оценивать наши модели каждые 10 эпох:

```
optimizer = SGD(lr=0.01)
trainer = Trainer(linear_regression, optimizer)

trainer.fit(X_train, y_train, X_test, y_test,
            epochs = 50,
            eval_every = 10,
            seed=20190501);

Validation loss after 10 epochs is 30.295
Validation loss after 20 epochs is 28.462
Validation loss after 30 epochs is 26.299
Validation loss after 40 epochs is 25.548
Validation loss after 50 epochs is 25.092
```

Использование тех же функций оценки моделей из главы 2 и помещение их в функцию `eval_regression_model` дают нам следующие результаты:

```
eval_regression_model(linear_regression, X_test, y_test)

Mean absolute error: 3.52

Root mean squared error 5.01
```

Это похоже на результаты линейной регрессии, которую мы использовали в предыдущей главе, а это подтверждает, что наша структура работает.

Запустив тот же код с моделью `neural_network` со скрытым слоем с 13 нейронами, мы получим следующее:

```
Validation loss after 10 epochs is 27.434
Validation loss after 20 epochs is 21.834
Validation loss after 30 epochs is 18.915
Validation loss after 40 epochs is 17.193
Validation loss after 50 epochs is 16.214

eval_regression_model(neural_network, X_test, y_test)

Mean absolute error: 2.60

Root mean squared error 4.03
```

Опять же, эти результаты похожи на те, что мы видели в предыдущей главе, и они значительно лучше, чем линейная регрессия.

Наша первая модель глубокого обучения (с нуля)

С настройками покончено, запустим первую модель:

```
deep_neural_network = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=LinearAct())],
   loss=MeanSquaredError(),
   learning_rate=0.01
)
```

Пока не будем фантазировать и просто добавим скрытый слой с той же размерностью, что и первый слой, так что наша сеть теперь имеет два скрытых слоя, каждый из которых содержит 13 нейронов.

Обучение с использованием той же скорости обучения и периодичности оценки, что и в предыдущих моделях, дает следующий результат:

```
Validation loss after 10 epochs is 44.134
Validation loss after 20 epochs is 25.271
Validation loss after 30 epochs is 22.341
Validation loss after 40 epochs is 16.464
Validation loss after 50 epochs is 14.604

eval_regression_model(deep_neural_network, X_test, y_test)

Mean absolute error: 2.45

Root mean squared error 3.82
```

Наконец-то мы перешли непосредственно к глубокому обучению, но тут уже будут реальные задачи, и без фокусов и хитростей наша модель глубокого обучения будет работать не намного лучше, чем простая нейронная сеть с одним скрытым слоем.

Что еще более важно, полученная структура легко расширяема. Мы могли бы весьма просто реализовать другие виды `Operation`, обернуть их в новые слои и сразу вставить их, предполагая, что в них определены `_output` и `_input_grad` и что размерности данных совпадают с размерностями их

соответствующих градиентов. Аналогично мы могли бы попробовать другие функции активации и посмотреть, уменьшит ли это показатели ошибок. Призываю взять наш код с GitHub (https://oreil.ly/deep-learning-github) и попробовать!

Заключение и следующие шаги

В следующей главе я расскажу о нескольких приемах, которые будут необходимы для правильной тренировки наших моделей, чтобы решать более сложные задачи[1]. Мы попробуем другие функции потерь и оптимизаторы. Я также расскажу о дополнительных приемах настройки скоростей обучения и их изменения в процессе обучения, а также покажу, как реализовать это в классах `Optimizer` и `Trainer`. Наконец, мы рассмотрим прореживание (Dropout), узнаем, что это такое и зачем оно нужно для повышения устойчивости обучения. Вперед!

[1] Даже в этой простой задаче незначительное изменение гиперпараметров может привести к тому, что модель глубокого обучения не справится с двухслойной нейронной сетью. Возьмите код с GitHub и попробуйте сами!