O'REILLY®

Uczenie głębokie od zera

Podstawy implementacji w Pythonie

Helion

Seth Weidman

Tytuł oryginału: Deep Learning from Scratch: Building with Python from First Principles

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-6597-1

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/uczgle
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 3.

Deep learning od podstaw

Możliwe, że o tym nie wiesz, ale masz już wszystkie matematyczne i konceptualne podstawy, aby poznać odpowiedź na postawione na początku książki najważniejsze pytania o modele z dziedziny deep learning. Wiesz już, *jak* działają sieci neuronowe (znasz obliczenia związane z mnożeniem macierzy, wartość straty i pochodne cząstkowe względem wartości straty) i *dlaczego* używane obliczenia są poprawne (dzięki regule łańcuchowej z analizy matematycznej). Dowiedziałeś się tego, budując sieci neuronowe z użyciem podstawowych zasad. Sieci reprezentowane były jako seria elementów w postaci pojedynczych funkcji matematycznych. W tym rozdziale nauczysz się reprezentować te elementy w formie abstrakcyjnych klas Pythona. Dalej wykorzystasz te klasy do budowania modeli z dziedziny deep learning. Do czasu zakończenia tego rozdziału utworzysz model typu „deep learning od podstaw".

Ponadto przekształcimy opisy sieci neuronowych oparte na elementach w bardziej standardowe opisy modeli z dziedziny deep learning, o których może już słyszałeś. Po lekturze tego rozdziału będziesz na przykład wiedzieć, co oznacza, że model z dziedziny deep learning ma „wiele warstw ukrytych". Na tym polega prawdziwe zrozumienie tematu — na umiejętności przechodzenia między wysokopoziomowymi opisami a niskopoziomowymi szczegółami operacji. Zacznijmy teraz drogę do tego przechodzenia. Do tej pory modele były tu opisywane wyłącznie w kategoriach niskopoziomowych operacji. W pierwszej części tego rozdziału taki opis modeli zostanie zastąpiony popularnymi wysokopoziomowymi pojęciami takimi jak warstwy. Te pojęcia później pozwolą na łatwiejsze opisywanie bardziej skomplikowanych modeli.

Definicja procesu deep learning — pierwszy krok

Czym *jest* model z dziedziny „deep learning"? W poprzednim rozdziale model został zdefiniowany jako funkcja matematyczna reprezentowana przez graf obliczeniowy. Przeznaczeniem takiego modelu jest odwzorowywanie danych wejściowych pochodzących ze zbioru danych o wspólnych aspektach (na przykład pojedyncze dane wejściowe reprezentujące różne cechy domów) na dane wyjściowe z powiązanego zbioru (na przykład na ceny tych domów). Odkryliśmy, że jeśli zdefiniujemy model jako funkcję przyjmującą na wejściu *parametry*, można dopasować model, by optymalnie opisywał dane. Służy do tego następująca procedura:

1. Wielokrotne przekazywanie obserwacji do modelu i zapisywanie wartości obliczanych w kroku w przód.
2. Obliczanie *wartości straty*, reprezentującej, jak odległe są predykcje modelu od oczekiwanych danych wyjściowych (*zmiennych wyjściowych*).
3. Używanie wartości obliczonych w kroku w przód i opartych na regule łańcuchowej obliczeń ustalonych w rozdziale 1., aby obliczyć, w jakim stopniu każdy z wejściowych *parametrów* wpływa na wartość straty.
4. Modyfikowanie wartości parametrów, tak aby wartość straty zmalała po przekazaniu do modelu następnego zbioru obserwacji.

Zaczęliśmy od modelu obejmującego serię operacji liniowych przekształcających cechy w odpowiedzi (okazało się to odpowiednikiem tradycyjnego modelu opartego na regresji liniowej). Oczekiwanym ograniczeniem tego podejścia jest to, że nawet po „optymalnym" dopasowaniu model może reprezentować tylko liniowe relacje między cechami a odpowiedziami.

Następnie zdefiniowaliśmy strukturę funkcji, w której najpierw wykonywane są operacje liniowe, potem operacja *nie*liniowa (funkcja `sigmoid`), a następnie końcowy zestaw operacji liniowych. Okazało się, że po tej modyfikacji model *potrafił* nauczyć się zależności bardziej zbliżonych do rzeczywistości (czyli nieliniowych relacji między danymi wejściowymi i wyjściowymi), a dodatkową korzyścią była możliwość nauczenia modelu relacji między *kombinacjami* cech wejściowych a odpowiedziami.

Jak łączą się modele tego rodzaju z modelami z dziedziny deep learning? Zaczniemy od nieco niezręcznej próby podania definicji: modele z dziedziny deep learning są reprezentowane w formie serii operacji, w której występują *przynajmniej dwie nienastępujące po sobie* funkcje nieliniowe.

Zaraz wyjaśnię, skąd wzięła się ta definicja. Najpierw jednak warto zauważyć, że ponieważ modele z dziedziny deep learning to serie operacji, proces uczenia ich jest *identyczny* jak dla omawianych wcześniej prostszych modeli. W końcu tym, dzięki czemu proces uczenia działa, jest różniczkowalność modelu względem danych wejściowych. W rozdziale 1. wspomniałem, że funkcja złożona obejmująca funkcje różniczkowalne też jest różniczkowalna. Dlatego jeśli poszczególne operacje tworzące funkcję są różniczkowalne, cała funkcja też taka jest i można uczyć model za pomocą opisanej wcześniej czteroetapowej procedury.

Jednak do tej pory uczenie modeli polegało na ręcznym obliczaniu pochodnych za pomocą ręcznego pisania kodu kroków w przód i wstecz oraz mnożenia odpowiednich wartości. W prostym modelu opartym na sieci neuronowej z rozdziału 2. wymagało to 17 kroków. Ponieważ modele do tej pory są opisywane na tak szczegółowym poziomie, nie jest jasne, jak zwiększyć złożoność modelu (i co dokładnie mogłoby to oznaczać), a nawet jak wprowadzić prostą zmianę, na przykład zastąpić funkcję sigmoidalną inną funkcją nieliniową. Aby móc budować dowolnie „głębokie" i „złożone" modele z dziedziny deep learning, trzeba zastanowić się nad tym, gdzie w tych 17 krokach można utworzyć komponenty wielokrotnego użytku i przyjąć poziom bardziej ogólny niż poszczególnych operacji, co pozwali zastępować te komponenty w celu tworzenia innych modeli. By zacząć tworzyć odpowiednie abstrakcje, spróbuję odwzorować używane operacje na tradycyjne opisy sieci neuronowych obejmujące „warstwy", „neurony" itd.

W pierwszym kroku spróbujemy utworzyć abstrakcję reprezentującą poszczególne operacje, które stosowaliśmy do tej pory. Dzięki temu nie trzeba będzie wielokrotnie pisać kodu do mnożenia macierzy i dodawania wyrazu wolnego.

Elementy sieci neuronowych — operacje

Klasa `Operation` będzie reprezentować jedną z funkcji składowych w sieci neuronowej. Wiemy (na podstawie sposobu, w jaki korzystaliśmy z funkcji w modelach), że na ogólnym poziomie powinna ona zawierać metody `forward` i `backward`. Obie te metody powinny przyjmować tablice `ndarray` i zwracać tablice `ndarray`. Niektóre operacje, na przykład mnożenie macierzy, mają *inny* specjalny rodzaj danych wejściowych: parametry; także one są zapisane w tablicy `ndarray`. W naszej klasie `Operation` (lub w innej klasie, pochodnej od `Operation`) dla parametrów utworzymy dodatkową zmienną instancji `params`.

Następne spostrzeżenie dotyczy tego, że są dwa rodzaje operacji. Niektóre, na przykład mnożenie macierzy, zwracają jako dane wyjściowe tablicę `ndarray` o kształcie innym niż tablica `ndarray` otrzymana jako dane wejściowe. Z kolei niektóre operacje, na przykład funkcja `sigmoid`, oznaczają zastosowanie jakiejś funkcji do każdego elementu wejściowej tablicy `ndarray`. Jak więc wygląda ogólna reguła dotycząca kształtów tablic `ndarray` przekazywanych między operacjami? Zastanów się nad takimi tablicami. Każda operacja przesyła dane wyjściowe do przodu w kroku w przód, a w kroku wstecz otrzymuje „gradient dla danych wyjściowych", który reprezentuje pochodną cząstkową funkcji straty względem każdego elementu z danych wyjściowych określonej operacji (w obliczeniach biorą udział inne operacje tworzące daną sieć). W kroku wstecz każda operacja przesyła wstecz „gradient dla danych wejściowych", reprezentujący pochodną cząstkową funkcji straty względem każdego elementu z danych wejściowych.

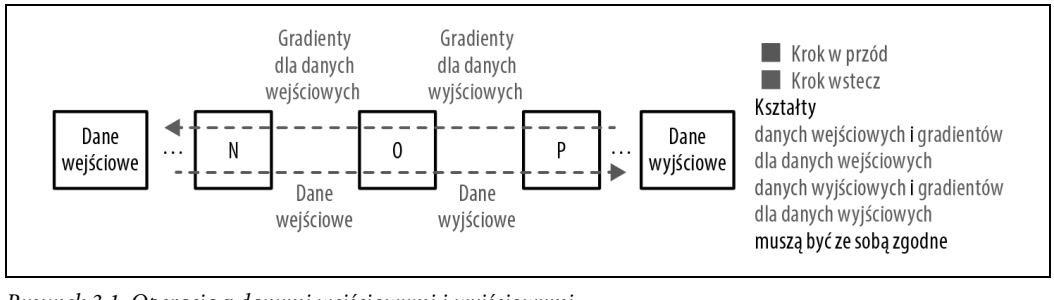
Te stwierdzenia nakładają na działanie operacji kilka ważnych ograniczeń, które pomogą zagwarantować, że gradienty są obliczane w poprawny sposób:

- Kształt tablicy `ndarray` z *gradientami dla danych wejściowych* musi pasować do kształtu *danych wyjściowych*.
- Kształt tablicy *gradientów dla danych wejściowych* przekazywanych przez operację do tyłu w kroku wstecz musi pasować do kształtu *danych wejściowych* tej operacji.

Stanie się to bardziej zrozumiałe, gdy zobaczysz rysunek. Przyjrzyj mu się teraz.

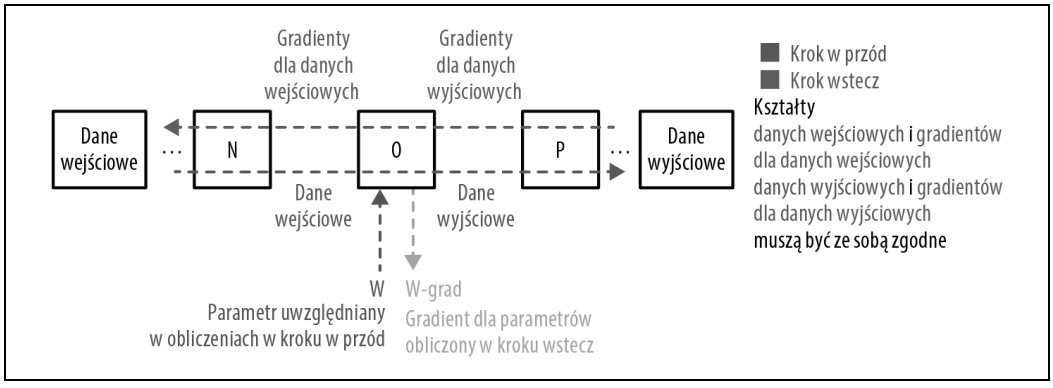
Diagram

Wszystkie te rozważania zostały podsumowane na rysunku 3.1 dla operacji `O` przyjmującej dane wejściowe od operacji `N` i przekazującej dane wyjściowe do innej operacji, `P`.



Rysunek 3.1. Operacja z danymi wejściowymi i wyjściowymi

Na rysunku 3.2 zilustrowany jest scenariusz dla operacji z parametrami.



Rysunek 3.2. Operacja z danymi wejściowymi, danymi wyjściowymi i parametrami (klasa ParamOperation)

Kod

Po tym omówieniu można napisać kod podstawowego elementu sieci neuronowej — klasy `Operation`:

```
class Operation(object):
    '''
    Klasa bazowa dla operacji w sieci neuronowej.
    '''
    def __init__(self):
        pass

    def forward(self, input_: ndarray):
        '''
        Zapisuje dane wejściowe w zmiennej instancji self._input.
        Wywołuje funkcję self._output().
        '''
        self.input_ = input_

        self.output = self._output()

        return self.output

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Wywołuje funkcję self._input_grad().
        Sprawdza, czy odpowiednie kształty pasują do siebie.
        '''
        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        return self.input_grad

    def _output(self) -> ndarray:
        '''
        Dla każdej operacji trzeba zdefiniować metodę _output.
        '''
```

```
        raise NotImplementedError()

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Dla każdej operacji trzeba zdefiniować metodę _input_grad.
        '''
        raise NotImplementedError()
```

Dla każdej definiowanej operacji trzeba zaimplementować funkcje `_output` i `_input_grad` (te nazwy odpowiadają wartościom obliczanym przez te funkcje).

Klasy bazowe tego rodzaju definiuję tu głównie w celach edukacyjnych. Ważne jest, aby zbudować model umysłowy wyjaśniający, że *wszystkie* operacje, jakie napotkasz w dziedzinie deep learning, przesyłają dane wejściowe w przód i gradienty wstecz, a także że kształty otrzymywane w kroku w przód pasują do kształtów przesyłanych do tyłu w kroku wstecz (i na odwrót).

Dalej w rozdziale zdefiniowane są konkretne operacje używane do tej pory — mnożenie macierzy itd. Najpierw jednak warto zdefiniować inną klasę, pochodną od klasy `Operation`. Ta nowa klasa jest przeznaczona dla operacji z użyciem parametrów:

```
class ParamOperation(Operation):
    '''
    Operacja z parametrami.
    '''

    def __init__(self, param: ndarray) -> ndarray:
        '''
        Metoda ParamOperation.
        '''
        super().__init__()
        self.param = param

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Wywołuje metody self._input_grad i self._param_grad.
        Sprawdza odpowiednie kształty.
        '''

        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)
        self.param_grad = self._param_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        assert_same_shape(self.param, self.param_grad)

        return self.input_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Każda klasa pochodna od ParamOperation musi zawierać implementację metody _param_grad.
        '''
        raise NotImplementedError()
```

Podobnie jak klasa bazowa `Operation` klasa `ParamOperation` musi zawierać definicje funkcji `_output` i `_input_grad`; dodatkowo musi też obejmować definicję funkcji `_param_grad`.

Elementy sieci neuronowej używane do tego miejsca w modelach zostały już formalnie opisane. Moglibyśmy przejść teraz do definiowania sieci neuronowych bezpośrednio z użyciem takich operacji, ale najpierw trzeba zdefiniować klasę pomocniczą, wokół której krążymy od półtora rozdziału — klasę `Layer`.

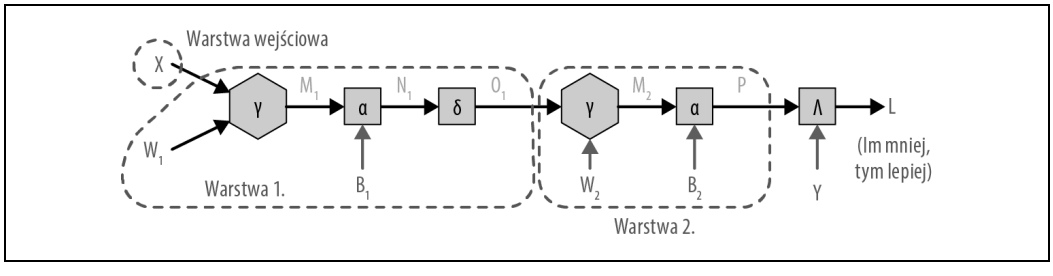
Elementy sieci neuronowych — warstwy

W kategoriach operacji warstwy można opisać jako serie operacji liniowych, po których następuje operacja liniowa. Na przykład sieć neuronowa z poprzedniego rozdziału ma pięć ogólnych operacji: dwie operacje liniowe (mnożenie wag i dodawanie wyrazu wolnego), po których wykonywana jest funkcja `sigmoid`, a następnie dwie kolejne operacje liniowe. W tym scenariuszu trzy pierwsze operacje (włącznie z nieliniową) tworzą pierwszą warstwę, a ostatnie dwie operacje znajdują się w drugiej warstwie. Ponadto same dane wejściowe tworzą specjalny rodzaj warstwy, warstwę *wejściową* (nie jest ona uwzględniana w numerowaniu, dlatego można o niej myśleć jak o warstwie zerowej). Podobnie ostatnia warstwa jest nazywana *wyjściową*. Warstwa środkowa, pierwsza w naszej numeracji, także ma ważną nazwę. Jest określana mianem warstwy *ukrytej*, ponieważ jest jedyną warstwą, której wartości nie są zwykle bezpośrednio widoczne w trakcie uczenia.

Warstwa wyjściowa jest ważnym wyjątkiem w tej definicji warstw, ponieważ *nie wymaga* stosowania operacji nieliniowej. Wynika to z tego, że często chcemy, aby wartości z tej warstwy znajdowały się w przedziale od minus do plus nieskończoności (a przynajmniej od 0 do nieskończoności), a funkcje nieliniowe zazwyczaj kompresują dane wejściowe do podzbioru tego przedziału adekwatnego do rozwiązywanego problemu. Na przykład funkcja `sigmoid` kompresuje dane wejściowe do przedziału od 0 do 1.

Diagramy

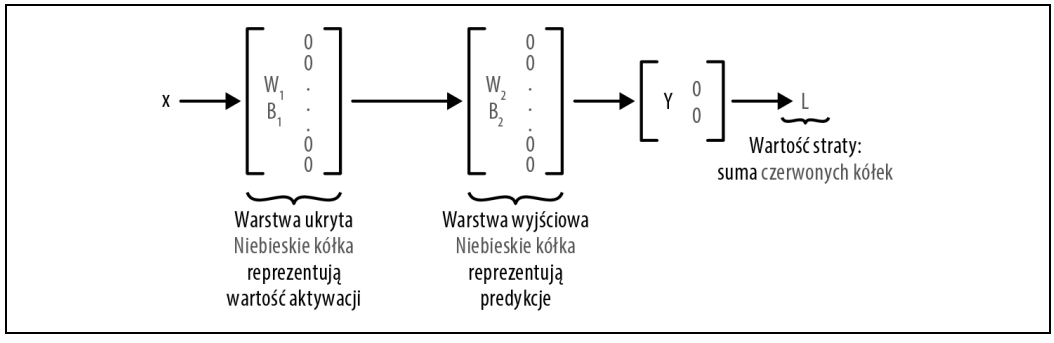
Aby powiązania były oczywiste, na rysunku 3.3 pokazany jest diagram sieci neuronowej z poprzedniego rozdziału. Poszczególne operacje są tu pogrupowane w warstwy.



Rysunek 3.3. Sieć neuronowa z poprzedniego rozdziału z operacjami pogrupowanymi w warstwy

Widać tu, że dane wejściowe są reprezentowane jako warstwa wejściowa, trzy następne operacje (kończące się funkcją `sigmoid`) reprezentują następną warstwę, a dwie ostatnie operacje to ostatnia warstwa.

Jest to oczywiście dość niewygodne. To nie przypadek — reprezentowanie sieci neuronowych w formie poszczególnych operacji wprawdzie dobrze pokazuje, jak działają takie sieci i jak ich uczyć, jest to jednak zbyt niskopoziomowa reprezentacja dla sieci neuronowych bardziej skomplikowanych niż sieci dwuwarstwowe. To dlatego częściej stosowana jest reprezentacja w formie warstw, pokazana na rysunku 3.4.



Rysunek 3.4. Sieć neuronowa z poprzedniego rozdziału przedstawiona w formie warstw

Analogie do mózgu

Pora przedstawić ostatnie powiązanie między tym, co zostało opisane do tego miejsca, a zagadnieniem, z którym zapewne się już zetknąłeś. Każda warstwa ma określoną liczbę *neuronów*, równą *liczbie wymiarów wektora reprezentującego każdą obserwację w danych wyjściowych tej warstwy*. Dlatego można stwierdzić, że sieć neuronowa z poprzedniego przykładu ma 13 neuronów w warstwie wejściowej, następnie ponownie 13 neuronów w warstwie ukrytej i jeden neuron w warstwie wyjściowej.

Neurony w mózgu przyjmują dane wejściowe od wielu innych neuronów, a następnie „odpalają" i przesyłają sygnał dalej tylko wtedy, jeśli otrzymane sygnały wspólnie dają określoną energię aktywacji. Neurony w sieciach neuronowych działają dość podobnie — rzeczywiście przesyłają sygnały dalej na podstawie danych wejściowych, przy czym te dane są przetwarzane na dane wyjściowe za pomocą funkcji nieliniowej. Ta funkcja nieliniowa jest tu nazywana *funkcją aktywacji*, a wartości wyjściowe tej funkcji to *aktywacje* dla danej warstwy[1].

Po zdefiniowaniu warstw można przedstawić bardziej klasyczną definicję deep learning: *modele w dziedzinie deep learning to sieci neuronowe mające więcej niż jedną warstwę ukrytą*.

Widać, że jest to odpowiednik wcześniejszej definicji (wyrażonej za pomocą samych operacji), ponieważ warstwa to seria operacji z nieliniową operacją na końcu.

Po zdefiniowaniu klasy bazowej dla operacji pora pokazać, że może ona stanowić podstawowy element modeli opisanych w poprzednim rozdziale.

[1] Spośród wszystkich funkcji aktywacji funkcja `sigmoid` (odwzorowująca wartości wejściowe na przedział od 0 do 1) najściślej odzwierciedla rzeczywistą aktywację neuronów w mózgu. Jednak ogólnie funkcją aktywacji może być dowolna monotoniczna funkcja nieliniowa.

Elementy z elementów

Jakie konkretne operacje trzeba zaimplementować, aby modele z poprzedniego rozdziału mogły działać? Dzięki doświadczeniu w implementowaniu sieci neuronowej krok po kroku wiemy, że potrzebne są trzy rodzaje operacji:

- mnożenie macierzy — danych wejściowych przez macierz parametrów,
- dodawanie wyrazu wolnego,
- funkcja aktywacji `sigmoid`.

Zacznijmy od operacji `WeightMultiply`:

```
class WeightMultiply(ParamOperation):
    '''
    Operacja mnożenia wag w sieci neuronowej.
    '''

    def __init__(self, W: ndarray):
        '''
        Inicjowanie operacji wartością self.param = W.
        '''
        super().__init__(W)

    def _output(self) -> ndarray:
        '''
        Obliczanie danych wyjściowych.
        '''
        return np.dot(self.input_, self.param)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Obliczanie gradientu dla danych wejściowych.
        '''
        return np.dot(output_grad, np.transpose(self.param, (1, 0)))

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Obliczanie gradientu dla parametrów.
        '''
        return np.dot(np.transpose(self.input_, (1, 0)), output_grad)
```

Jest to prosty kod mnożenia macierzy w kroku w przód oraz reguł przesyłania w tył gradientów dla danych wejściowych i parametrów w kroku wstecz. Używane są tu reguły wywnioskowane w końcowej części rozdziału 1. Dalej zobaczysz, że te operacje można wykorzystać jako *elementy* dodawane w łatwy sposób do warstw.

Teraz zajmijmy się operacją dodawania. Nazwiemy ją `BiasAdd`:

```
class BiasAdd(ParamOperation):
    '''
    Dodawanie wyrazu wolnego.
    '''

    def __init__(self,
```

```
                B: ndarray):
        '''
        Inicjowanie operacji wartością self.param = B.
        Sprawdzanie kształtu.
        '''
        assert B.shape[0] == 1

        super().__init__(B)

    def _output(self) -> ndarray:
        '''
        Obliczanie danych wyjściowych.
        '''
        return self.input_ + self.param

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Obliczanie gradientu dla danych wyjściowych.
        '''
        return np.ones_like(self.input_) * output_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Obliczanie gradientu dla parametrów.
        '''
        param_grad = np.ones_like(self.param) * output_grad
        return np.sum(param_grad, axis=0).reshape(1, param_grad.shape[1])
```

Teraz przejdźmy do funkcji `sigmoid`:

```
class Sigmoid(Operation):
    '''
    Funkcja aktywacji Sigmoid.
    '''

    def __init__(self) -> None:
        ''' Przekazywanie do typu bazowego. '''
        super().__init__()

    def _output(self) -> ndarray:
        '''
        Obliczanie danych wyjściowych.
        '''
        return 1.0/(1.0+np.exp(-1.0 * self.input_))

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Obliczenie gradientu dla danych wejściowych.
        '''
        sigmoid_backward = self.output * (1.0 - self.output)
        input_grad = sigmoid_backward * output_grad
        return input_grad
```

Jest to prosta implementacja obliczeń matematycznych opisanych w poprzednim rozdziale.

W obu klasach, `sigmoid` i `ParamOperation`, krok w kroku wstecz, gdzie wykonywane są obliczenia:

```
input_grad = <coś> * o rutput_grad
```

jest krokiem, gdzie stosowana jest reguła łańcuchowa. Analogiczna reguła w klasie `WeightMultiply` to:

```
np.dot(output_grad, np.transpose(self.param, (1, 0)))
```

Jest to, jak opisano w rozdziale 1., reguła analogiczna do reguły łańcuchowej w sytuacji, gdy funkcją jest mnożenie macierzy.

Po precyzyjnym zdefiniowaniu operacji można korzystać *z nich* jako elementów do definiowania klasy `Layer`.

Wzorzec warstwy

Dzięki temu, jak napisane zostały operacje, utworzenie klasy `Layer` jest łatwe:

- Metody `forward` i `backward` przesyłają dane wejściowe w przód w serii operacji — dokładnie tak, jak jest to pokazane na diagramach. Jest to najważniejsza informacja związana z działaniem warstw. Reszta kodu to nakładka na ten proces przeznaczona głównie do wykonywania zadań porządkujących. Oto te zadania:
  - Definiowanie odpowiedniej serii operacji w funkcji `_setup_layer` oraz inicjalizowania i zapisywania parametrów w tych operacjach (co też będzie wykonywane w funkcji `_setup_layer`).
  - Zapisywanie odpowiednich wartości w polach `self.input_` i `self.output` w metodzie `forward`.
  - Sprawdzanie asercji w metodzie `backward`.
- Funkcje `_params` i `_param_grads` pobierają parametry i ich gradienty (względem wartości straty) z operacji sparametryzowanych z warstwy.

Oto cały ten kod:

```
class Layer(object):
    '''
    Warstwa neuronów w sieci neuronowej.
    '''

    def __init__(self,
                 neurons: int):
        '''
        Liczba neuronów w przybliżeniu odpowiada szerokości
        warstwy.
        '''
        self.neurons = neurons
        self.first = True
        self.params: List[ndarray] = []
        self.param_grads: List[ndarray] = []
        self.operations: List[Operation] = []
```

```
def _setup_layer(self, num_in: int) -> None:
    '''
    W każdej warstwie trzeba zaimplementować funkcję _setup_layer.
    '''
    raise NotImplementedError()

def forward(self, input_: ndarray) -> ndarray:
    '''
    Przekazuje dane wejściowe w przód w serii operacji.
    '''
    if self.first:
        self._setup_layer(input_)
        self.first = False

    self.input_ = input_

    for operation in self.operations:

        input_ = operation.forward(input_)

    self.output = input_

    return self.output

def backward(self, output_grad: ndarray) -> ndarray:
    '''
    Przekazuje output_grad wstecz w serii operacji.
    Sprawdza kształty danych.
    '''
    assert_same_shape(self.output, output_grad)

    for operation in reversed(self.operations):
        output_grad = operation.backward(output_grad)

    input_grad = output_grad

    self._param_grads()

    return input_grad

def _param_grads(self) -> ndarray:
    '''
    Pobiera _param_grads z operacji z warstwy.
    '''

    self.param_grads = []
    for operation in self.operations:
        if issubclass(operation.__class__, ParamOperation):
            self.param_grads.append(operation.param_grad)

def _params(self) -> ndarray:
    '''
    Pobiera _params z operacji z warstwy.
    '''

    self.params = []
    for operation in self.operations:
        if issubclass(operation.__class__, ParamOperation):
            self.params.append(operation.param)
```

Podobnie jak przeszliśmy od abstrakcyjnej definicji operacji do implementacji konkretnych operacji na potrzeby sieci neuronowej z rozdziału 2., teraz zaimplementujemy także warstwę z tej sieci.

Warstwa gęsta

Używane operacje nazwaliśmy `WeightMultiply`, `BiasAdd` itd. Jak nazwać warstwę, której używaliśmy do tej pory? Warstwą liniowo-nieliniową (`LinearNonLinear`).

Cechą definiującą tę warstwę jest to, że *każdy neuron wyjściowy jest funkcją od wszystkich neuronów wejściowych*. Tak właśnie działa mnożenie macierzy. Jeśli macierz ma n_{in} wierszy i n_{out} kolumn, mnożenie oblicza n_{out} nowych cech, z których każda jest ważoną liniową kombinacją *wszystkich* n_{in} cech wejściowych[2]. Takie warstwy są często nazywane *każdy z każdym* (ang. *fully connected*). Od niedawna w popularnej bibliotece `Keras` są one nazywane warstwami `Dense` czy gęstymi. Jest to bardziej zwięzła nazwa, która oznacza to samo.

Teraz, gdy wiesz już, jak nazywa się omawiana warstwa i dlaczego, pora utworzyć warstwę `Dense` w kategoriach już zdefiniowanych operacji. Zobaczysz, że z powodu formy definicji klasy bazowej `Layer` wystarczy dodać zdefiniowane w poprzednim punkcie operacje jako listę w funkcji `_setup_layer`:

```
class Dense(Layer):
    '''
    Warstwa gęsta dziedzicząca po klasie Layer.
    '''
    def __init__(self,
                 neurons: int,
                 activation: Operation = Sigmoid()) -> None:
        '''
        Inicjalizacja wymaga określenia funkcji aktywacji.
        '''
        super().__init__(neurons)
        self.activation = activation

    def _setup_layer(self, input_: ndarray) -> None:
        '''
        Definiuje operacje warstwy gęstej.
        '''
        if self.seed:
            np.random.seed(self.seed)

        self.params = []

        # Wagi.
        self.params.append(np.random.randn(input_.shape[1], self.neurons))

        # Wyraz wolny.
        self.params.append(np.random.randn(1, self.neurons))

        self.operations = [WeightMultiply(self.params[0]),
                           BiasAdd(self.params[1]),
                           self.activation]

        return None
```

[2] W rozdziale 5. zobaczysz, że nie dla wszystkich warstw jest to prawdą. Na przykład w warstwach *konwolucyjnych* każda cecha wyjściowa jest kombinacją *tylko niewielkiego podzbioru* cech wejściowych.

Domyślnie stosowana jest aktywacja liniowa, co tak naprawdę oznacza, że nie używamy aktywacji i stosujemy funkcję tożsamościową do danych wyjściowych warstwy.

Jakie elementy należy dodać na poziomie powyżej klas `Operation` i `Layer`? Wiadomo, że do uczenia modelu potrzebna będzie klasa `NeuralNetwork` obejmująca obiekty klasy `Layer` (podobnie jak klasa `Layer` obejmuje obiekty typu `Operation`). Nie jest jednak oczywiste, jakie jeszcze klasy będą potrzebne. Dlatego przejdźmy do budowania klasy `NeuralNetwork` i ustalmy, jakie jeszcze inne klasy będą potrzebne.

Klasa NeuralNetwork (i ewentualnie inne)

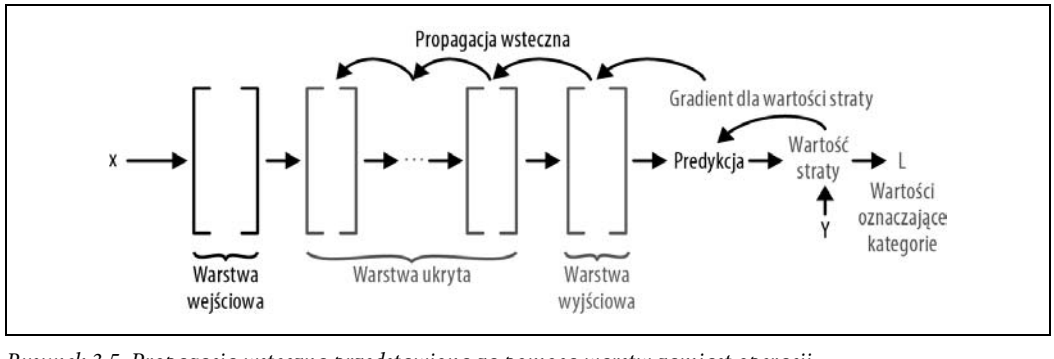
Jakie powinny być możliwości klasy `NeuralNetwork`? Na ogólnym poziomie ta klasa powinna móc *uczyć się na podstawie danych*, a bardziej precyzyjnie — przyjmować porcje danych reprezentujące obserwacje (`X`) i poprawne odpowiedzi (`y`) oraz uczyć się relacji między `X` i `y`, co oznacza ustalenie funkcji, która potrafi przekształcać `X` na predykcje `p` bardzo zbliżone do `y`.

Jak dokładnie przebiega to uczenie, jeśli wziąć pod uwagę właśnie zdefiniowane klasy `Layer` i `Operation`? Zgodnie z modelem z poprzedniego rozdziału zaimplementujmy proces uczenia tak:

1. Sieć neuronowa powinna przyjmować `X` i przekazywać je do wszystkich kolejnych warstw (warstwa ta jest tylko wygodną nakładką na przekazywanie danych między wieloma operacjami). Wynik reprezentuje predykcje.
2. Następnie predykcje należy porównać z wartością `y`, aby obliczyć wartość straty i wygenerować gradient dla wartości straty, czyli pochodną cząstkową wartości straty względem każdego elementu z ostatniej warstwy sieci (czyli z warstwy, która wygenerowała predykcję).
3. W ostatnim kroku gradient dla wartości straty jest przesyłany wstecz przez wszystkie warstwy. Obliczane są przy tym gradienty dla parametrów (pochodne cząstkowe wartości straty względem każdego parametru); te gradienty są zapisywane w odpowiednich operacjach.

Diagram

Na rysunku 3.5 przedstawiony jest opis sieci neuronowej w kategoriach warstw.



Rysunek 3.5. Propagacja wsteczna przedstawiona za pomocą warstw zamiast operacji

Kod

Jak zaimplementować ten proces? Przede wszystkim sieć neuronowa powinna używać warstw w taki sam sposób, jak warstwy używają operacji. Na przykład metoda `forward` powinna przyjmować `X` na wejściu i robić coś takiego:

```
for layer in self.layers:
    X = layer.forward(X)

return X
```

Podobnie metoda `backward` powinna przyjmować argument (nazwijmy go na razie `grad`) i wykonywać operację podobną do tej:

```
for layer in reversed(self.layers):
    grad = layer.backward(grad)
```

Skąd pochodzi argument `grad`? Musi pochodzić ze specjalnej *funkcji straty*, która przyjmuje argumenty `prediction` i `y` oraz:

- Oblicza wartość reprezentującą błąd sieci generującej daną predykcję.
- Przesyła wstecz gradient dla każdego elementu z argumentu `prediction` względem wartości straty. Ten gradient jest otrzymywany przez ostatnią warstwę sieci jako dane wejściowe funkcji `backward`.

W przykładzie z poprzedniego rozdziału funkcja straty wyznaczała kwadrat różnicy między predykcją a odpowiedzią. Na tej podstawie obliczany był gradient dla predykcji względem wartości straty.

Jak to zaimplementować? Wydaje się, że opisany proces jest na tyle ważny, że zasługuje na własną klasę. Tę klasę należy zaimplementować podobnie jak klasę `Layer`, przy czym metoda `forward` powinna zwracać jako wartość straty konkretną liczbę (typu `float`) zamiast tablicy `ndarray` przesyłanej w przód do następnej warstwy. Zapiszmy to w formalny sposób.

Klasa Loss

Klasa bazowa `Loss` jest podobna do klasy `Layer`. Metody `forward` i `backward` będą sprawdzać, czy kształty odpowiednich tablic `ndarray` są identyczne. Należy też zdefiniować dwie metody, `_output` i `_input_grad`, które powinna być zdefiniowane w każdej podklasie klasy `Loss`:

```
class Loss(object):
    '''
    Wartość straty w sieci neuronowej.
    '''

    def __init__(self):
        ''' Operacja pusta. '''
        pass

    def forward(self, prediction: ndarray, target: ndarray) -> float:
        '''
        Oblicza wartość straty.
        '''
        assert_same_shape(prediction, target)
```

```
        self.prediction = prediction
        self.target = target

        loss_value = self._output()

        return loss_value

    def backward(self) -> ndarray:
        '''
        Oblicza gradient dla wartości straty względem danych wejściowych
        funkcji straty.
        '''
        self.input_grad = self._input_grad()

        assert_same_shape(self.prediction, self.input_grad)

        return self.input_grad

    def _output(self) -> float:
        '''
        Funkcja _output musi być zaimplementowana w każdej klasie pochodnej od Loss.
        '''
        raise NotImplementedError()

    def _input_grad(self) -> ndarray:
        '''
        Funkcja _input_grad musi być zaimplementowana w każdej klasie pochodnej od Loss.
        '''
        raise NotImplementedError()
```

Podobnie jak w klasie `Operation` należy sprawdzać, czy gradient przesyłany wstecz ma ten sam kształt co argument `prediction` otrzymany jako dane wejściowe z ostatniej warstwy sieci:

```
class MeanSquaredError(Loss):

    def __init__(self)
        ''' Operacja pusta. '''
        super().__init__()

    def _output(self) -> float:
        '''
        Obliczanie wartości straty (jako błędu kwadratowego) dla obserwacji.
        '''
        loss =
            np.sum(np.power(self.prediction - self.target, 2)) /
            self.prediction.shape[0]

        return loss

    def _input_grad(self) -> ndarray:
        '''
        Obliczanie gradientu dla wartości straty względem danych wejściowych
        (używany jest tu błąd średniokwadratowy).
        '''

        return 2.0 * (self.prediction - self.target) / self.prediction.shape[0]
```

Zapisaliśmy tu reguły dla kroków w przód i wstecz z użyciem błędu średniokwadratowego jako wartości straty.

Jest to ostatni ważny element potrzebny do zbudowania modelu deep learning od podstaw. Pora zobaczyć, jak poszczególne elementy są powiązane ze sobą, a następnie przejść do budowania modelu.

Deep learning od podstaw

Ostatecznie chcemy utworzyć klasę `NeuralNetwork`, używając rysunku 3.5 jako punktu wyjścia. Ta klasa ma umożliwić definiowanie i uczenie modeli z dziedziny deep learning. Zanim zaczniemy pisać kod, warto precyzyjnie opisać, jak taka klasa powinna wyglądać i jak powinna komunikować się z już zdefiniowanymi klasami `Operation`, `Layer` i `Loss`:

1. Klasa `NeuralNetwork` powinna mieć atrybut w postaci listy warstw. Powinny to być warstwy takie jak zdefiniowano wcześniej, z metodami `forward` i `backward`. Te metody powinny przyjmować tablice `ndarray` i zwracać takie tablice.
2. W każdej warstwie zapisana jest lista operacji zapisywanych w atrybucie `operations` warstwy w funkcji `_setup_layer`.
3. Te operacje, podobnie jak sama warstwa, mają metody `forward` i `backward`, które przyjmują jako argumenty tablice `ndarray` i zwracają takie tablice jako dane wyjściowe.
4. W każdej operacji kształt argumentu `output_grad` otrzymanego przez metodę `backward` musi być taki sam jak kształt atrybutu `output` w warstwie. To samo dotyczy kształtów tablicy `input_grad` przekazywanej wstecz w metodzie `backward` i atrybutu `input_`.
5. Niektóre operacje mają parametry (zapisane w atrybucie `param`). Takie operacje dziedziczą po klasie `ParamOperation`. Opisane ograniczenia dotyczą także kształtów danych wejściowych i wyjściowych w warstwach oraz metod `forward` i `backward` warstw. Te metody przyjmują i zwracają tablice `ndarray`, a kształty atrybutów `input` i `output` oraz powiązanych gradientów muszą do siebie pasować.
6. Klasa `NeuralNetwork` zawiera też obiekt klasy `Loss`. Klasa `Loss` przyjmuje dane wyjściowe z ostatniej operacji z klasy `NeuralNetwork` i odpowiedź, sprawdza, czy kształty tych elementów są takie same, a następnie oblicza zarówno wartość straty (liczbę), jak i tablicę `ndarray` `loss_grad`, która zostanie przekazana do warstwy wyjściowej w celu rozpoczęcia propagacji wstecznej.

Implementowanie treningu na porcjach danych

Kilkakrotnie opisane zostały już wysokopoziomowe kroki uczenia modelu porcja po porcji. Te kroki są ważne i warto je powtórzyć:

1. Przekazywanie danych wejściowych do funkcji modelu (krok w przód) w celu uzyskania predykcji.
2. Obliczanie liczby reprezentującej wartość straty.
3. Obliczanie gradientu dla wartości straty względem parametrów z wykorzystaniem reguły łańcuchowej i wartości obliczonych w kroku w przód.
4. Modyfikowanie parametrów na podstawie gradientów.

Następnie przekazywana jest nowa porcja danych i kroki są powtarzane.

Te kroki można łatwo przekształcić na opisany właśnie model z klasą `NeuralNetwork`:

1. Przyjmowanie `X` i `y` jako danych wyjściowych. Oba te elementy to tablice `ndarray`.
2. Przekazywanie `X` w przód do wszystkich warstw.
3. Używanie klasy `Loss` do ustalenia wartości straty i przekazywanego wstecz gradientu dla wartości straty.
4. Używanie gradientu dla wartości straty jako danych wejściowych metody `backward` sieci. Metoda ta oblicza gradienty `param_grads` dla każdej warstwy w sieci.
5. Wywołanie funkcji `update_params` dla każdej warstwy. Funkcja ta używa ogólnego współczynnika uczenia dla danego obiektu klasy `NeuralNetwork`, a także nowo obliczonych gradientów `param_grads`.

Wreszcie uzyskaliśmy kompletną definicję sieci neuronowej z uwzględnieniem uczenia na podstawie porcji danych. Pora napisać dla niej kod.

Klasa NeuralNetwork — kod

Kod dla tej klasy jest całkiem prosty:

```
class NeuralNetwork(object):
    '''
    Klasa dla sieci neuronowej.
    '''
    def __init__(self, layers: List[Layer],
        loss: Loss,
        seed: float = 1)
        '''
        Sieci neuronowe wymagają warstw i wartości straty.
        '''
        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:
        '''
        Przekazywanie danych w przód serii warstw.
        '''
        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:
        '''
        Przekazywanie danych wstecz serii warstw.
        '''
```

```
        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:
        '''
        Przekazywanie danych w przód serii warstw.
        Obliczanie wartości straty.
        Przekazywanie danych wstecz serii warstw.
        '''

        predictions = self.forward(x_batch)

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

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

        return loss

    def params(self):
        '''
        Pobieranie parametrów sieci.
        '''
        for layer in self.layers:
            yield from layer.params

    def param_grads(self):
        '''
        Pobieranie gradientów dla wartości straty względem
        parametrów sieci.
        '''
        for layer in self.layers:
            yield from layer.param_grads
```

W klasie `NeuralNetwork` można zaimplementować modele z poprzedniego rozdziału w bardziej modułowy, elastyczny sposób, a także zdefiniować inne modele, reprezentujące złożone nieliniowe relacje między danymi wejściowymi i wyjściowymi. Na przykład poniżej pokazane jest, jak można w łatwy sposób utworzyć instancje dwóch modeli omówionych w poprzednim rozdziale — opartych na regresji liniowej i sieci neuronowej[3]:

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

neural_network = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
```

[3] Wartość współczynnika uczenia równa 0,01 nie jest specjalną liczbą. Wartość ta okazała się optymalna w trakcie eksperymentów związanych z poprzednim rozdziałem.

```
Dense(neurons=1,
      activation=Linear())],
loss = MeanSquaredError(),
learning_rate = 0.01
)
```

Kod jest już w zasadzie gotowy. Teraz wystarczy wielokrotnie przekazywać dane w sieci, aby ją wytrenować. Jednak aby ten proces był bardziej przejrzysty i łatwiejszy do rozszerzania na potrzeby bardziej skomplikowanych scenariuszy z dziedziny deep learning, z jakimi zetkniesz się w następnym rozdziale, warto zdefiniować dodatkowe klasy, która odpowiadają za uczenie, czyli modyfikowanie parametrów obiektu klasy `NeuralNetwork` na podstawie gradientów obliczonych w kroku wstecz. Zdefiniujmy szybko te dwie klasy.

Nauczyciel i optymalizator

Najpierw warto zwrócić uwagę na podobieństwa między tymi klasami a kodem używanym do uczenia sieci w rozdziale 2. Tam używaliśmy następującego kodu, aby zaimplementować cztery kroki opisane wcześniej na potrzeby uczenia modelu:

```
# Przekazywanie X_batch w przód i obliczanie wartości straty.
forward_info, loss = forward_loss(X_batch, y_batch, weights)

# Obliczanie gradientu dla wartości straty względem każdej z wag.
loss_grads = loss_gradients(forward_info, weights)

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

Ten kod znajdował się w pętli `for`, która wielokrotnie przekazywała dane do funkcji definiującej i modyfikującej sieć.

Gdy dostępne są już istniejące klasy, opisany proces można wykonać w funkcji `fit` w klasie `Trainer`, która jest przede wszystkim nakładką na funkcję `train` używaną w poprzednim rozdziale. Kompletny kod znajdziesz w arkuszu Jupyter Notebook dla tego rozdziału na stronie książki w serwisie GitHub (*https://oreil.ly/2MV0aZI*). Główna różnica polega na tym, że w nowej funkcji dwa pierwsze wiersze z poprzedniego bloku kodu zostaną zastąpione następującym wierszem:

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

Modyfikowanie parametrów wykonywane w dwóch kolejnych wierszach będzie odbywać się w odrębnej klasie `Optimizer`. Ponadto pętla `for`, która wcześniej obejmowała cały kod związany z uczeniem, znajdzie się w klasie `Trainer` zawierającej obiekty typów `NeuralNetwork` i `Optimizer`.

Teraz warto omówić, dlaczego potrzebna jest klasa `Optimizer` i jak powinna ona wyglądać.

Optymalizator

W modelu opisanym w poprzednim rozdziale każda warstwa zawiera prostą regułę modyfikowania wag na podstawie parametrów i gradientów. W następnym rozdziale wspomniane jest, że można stosować także wiele innych reguł, na przykład uwzględniających *historię* zmian gradientów,

a nie tylko modyfikacje gradientów na podstawie konkretnej porcji danych przekazanej w danej iteracji. Utworzenie odrębnej klasy `Optimizer` daje swobodę podmiany reguł modyfikowania parametrów. Temat ten jest szczegółowo omówiony w następnym rozdziale.

Opis i kod

Klasa bazowa `Optimizer` przyjmuje obiekt klasy `NeuralNetwork` i przy każdym wywołaniu funkcji `step` modyfikuje parametry sieci na podstawie ich bieżących wartości, gradientów i innych informacji zapisanych w klasie `Optimizer`:

```
class Optimizer(object):
    '''
    Klasa bazowa dla optymalizatora sieci neuronowej.
    '''

    def __init__(self,
                 lr: float = 0.01):
        '''
        Każdy optymalizator musi mieć początkowy współczynnik uczenia.
        '''
        self.lr = lr

    def step(self) -> None:
        '''
        Każdy optymalizator musi implementować funkcję step.
        '''
        pass
```

Tak kod wygląda dla prostej reguły modyfikowania parametrów używanej do tej pory. Ta reguła to *SGD* (ang. *stochastic gradient descent*):

```
class SGD(Optimizer):
    '''
    Optymalizator używający metody SGD.
    '''
    def __init__(self,
                 lr: float = 0.01) -> None:
        ''' Operacja pusta. '''
        super().__init__(lr)

    def step(self):
        '''
        Dla każdego parametru wprowadzana jest zmiana w odpowiednim kierunku.
        Wielkość zmiany jest zależna od współczynnika uczenia.
        '''
        for (param, param_grad) in zip(self.net.params(),
                                       self.net.param_grads()):

            param -= self.lr * param_grad
```

Warto zauważyć, że choć nasza klasa `NeuralNetwork` nie ma metody `_update_params`, korzystamy z metod `params()` i `param_grads()`, aby pobrać odpowiednie tablice `ndarray` na potrzeby optymalizacji.

To była prosta klasa `Optimizer`. Przejdźmy teraz do klasy `Trainer`.

Nauczyciel

Oprócz uczenia modelu w opisany wcześniej sposób klasa `Trainer` łączy też klasy `NeuralNetwork` i `Optimizer`, dbając o to, by ta druga poprawnie uczyła tę pierwszą. Może zauważyłeś w poprzednim punkcie, że w ramach inicjalizowania obiektu klasy `Optimizer` nie jest przekazywany do niego obiekt klasy `NeuralNetwork`. Zamiast tego obiekt klasy `NeuralNetwork` będzie ustawiany jako atrybut obiektu klasy `Optimizer` w ramach inicjalizowania obiektu klasy `Trainer`. Służy do tego następujący wiersz:

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

W następnym punkcie przedstawiam uproszczoną, ale działającą wersję klasy `Trainer`, która na razie zawiera tylko metodę `fit`. Ta metoda uczy model przez określoną liczbę *epok* i po każdej ustalonej liczbie epok wyświetla wartość straty. W każdej epoce kod:

1. Losowo przestawia dane na początku każdej epoki.
2. Przekazuje dane w sieci w porcjach, modyfikując parametry po przetworzeniu każdej porcji.

Epoka kończy się po przetworzeniu przez obiekt klasy `Trainer` całego zbioru treningowego.

Kod nauczyciela

Dalej pokazany jest kod prostej wersji klasy `Trainer`. Ukryte są tu dwie oczywiste metody pomocnicze używane w funkcji `fit`: `generate_batches` (generuje porcje danych na potrzeby uczenia na podstawie kolekcji `X_train` i `y_train`) i `mute_data` (losowo przestawia elementy kolekcji `X_train` i `y_train` na początku każdej epoki). W funkcji `train` znajduje się też argument `restart`. Jeśli jego wartość to `true` (jest to ustawienie domyślne), po wywołaniu funkcji `train` parametry modelu są ponownie inicjalizowane wartościami losowymi:

```
class Trainer(object):
    '''
    Uczy sieć neuronową.
    '''
    def __init__(self,
                 net: NeuralNetwork,
                 optim: Optimizer)
        '''
        Wymaga sieci neuronowej i optymalizatora, aby wykonać proces uczenia.
        Przypisuje sieć neuronową jako zmienną instancji do optymalizatora.
        '''
        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:
        '''
        Dopasowuje sieć neuronową do danych treningowych przez określoną liczbę epok.
        Co "eval_every" epok sprawdza sieć neuronową względem
        danych testowych.
```

```
        '''
        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"Kontrolna wartość straty po {e+1} epokach wynosi {loss:.3f}")
```

W pełnej wersji tej funkcji w repozytorium książki w serwisie GitHub (*https://oreil.ly/2MV0aZI*) zaimplementowane jest też *przerywanie uczenia* (ang. *early stopping*), które:

1. Zapisuje wartość straty co `eval_every` epok.
2. Sprawdza, czy kontrolna wartość straty jest niższa niż po ostatnim jej obliczeniu.
3. Jeśli kontrolna wartość straty *nie* jest niższa, używa modelu sprzed `eval_every` epok.

Wreszcie dostępne są wszystkie elementy potrzebne do uczenia modeli!

Łączenie wszystkich elementów

Oto kompletny kod do uczenia sieci z użyciem klas `Trainer` i `Optimizer` oraz dwóch zdefiniowanych wcześniej modeli: `linear_regression` i `neural_network`. Współczynnik uczenia jest ustawiony na `0.01`, a maksymalna liczba epok to `50`. Ocena modeli odbywa się co `10` epok:

```
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);

Kontrolna wartość straty po 10 epokach wynosi 30.295
Kontrolna wartość straty po 20 epokach wynosi 28.462
Kontrolna wartość straty po 30 epokach wynosi 26.299
Kontrolna wartość straty po 40 epokach wynosi 25.548
Kontrolna wartość straty po 50 epokach wynosi 25.092
```

Zastosowanie funkcji do oceny modelu z rozdziału 2. i umieszczenie ich w funkcji `eval_regre ↪ssion_model` pozwala uzyskać następujące wyniki:

```
eval_regression_model(linear_regression, X_test, y_test)
Średni błąd bezwzględny: 3.52
Pierwiastek błędu średniokwadratowego: 5.01
```

Są to wartości podobne do wyników dla regresji liniowej z poprzedniego rozdziału. Potwierdza to, że nowa platforma działa.

Uruchomienie tego samego kodu dla modelu `neural_network` z ukrytą warstwą z 13 neuronami daje następujące wyniki:

```
Kontrolna wartość straty po 10 epokach wynosi 27.434
Kontrolna wartość straty po 20 epokach wynosi 21.834
Kontrolna wartość straty po 30 epokach wynosi 18.915
Kontrolna wartość straty po 40 epokach wynosi 17.193
Kontrolna wartość straty po 50 epokach wynosi 16.214

eval_regression_model(neural_network, X_test, y_test)
Średni błąd bezwzględny: 2.60
Pierwiastek błędu średniokwadratowego: 4.03
```

Także te wyniki są podobne do danych z poprzedniego rozdziału i okazują się znacznie lepsze niż dla prostej regresji liniowej.

Pierwszy model z dziedziny deep learning (napisany od podstaw)

Teraz, po zakończeniu wszystkich przygotowań, zdefiniowanie pierwszego modelu z dziedziny deep learning będzie proste:

```
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
)
```

Nie będę nawet próbował kombinować z ustawieniami (na razie). Dodam tylko warstwę ukrytą o tych samych wymiarach co warstwa pierwsza, tak aby sieć miała teraz dwie warstwy ukryte (każda po 13 neuronów).

Uczenie tego modelu z tym samym współczynnikiem uczenia i harmonogramem ocen co dla poprzednich modeli daje następujące wyniki:

```
Kontrolna wartość straty po 10 epokach wynosi 44.134
Kontrolna wartość straty po 20 epokach wynosi 25.271
Kontrolna wartość straty po 30 epokach wynosi 22.341
Kontrolna wartość straty po 40 epokach wynosi 16.464
Kontrolna wartość straty po 50 epokach wynosi 14.604

eval_regression_model(neural_network, X_test, y_test)
Średni błąd bezwzględny: 2.45
Pierwiastek błędu średniokwadratowego: 3.82
```

Wreszcie dotarliśmy do zastosowania deep learning od podstaw. I rzeczywiście w tym praktycznym problemie, bez stosowania żadnych sztuczek (tylko z niewielką zmianą współczynnika uczenia), nasz model z dziedziny deep learning okazał się nieco lepszy niż sieć neuronowa z tylko jedną warstwą ukrytą.

Ważniejsze jest jednak to, że zbudowaliśmy platformę, którą można łatwo rozbudować. Moglibyśmy w prosty sposób zaimplementować operacje innego rodzaju, umieścić je w nowych warstwach i od razu zastosować — pod warunkiem, że zdefiniowane byłyby dla nich metody `_output` i `_input_grad`, a wymiary danych wejściowych, danych wyjściowych i parametry pasowałyby do wymiarów powiązanych gradientów. Łatwo moglibyśmy także zastosować inne funkcje aktywacji dla istniejących warstw i sprawdzić, czy skutkuje to zmniejszeniem błędów. Zachęcam, aby sklonować repozytorium książki z serwisu GitHub (*https://oreil.ly/deep-learning-github*) i spróbować wykonać opisane zadania.

Podsumowanie i dalsze kroki

W następnym rozdziale omawiam kilka sztuczek, które są nieodzowne do poprawnego uczenia modeli przeznaczonych dla trudniejszych problemów niż proste zadanie omawiane do tej pory[4]. Objaśniam przede wszystkim definiowanie klas `Loss` i `Optimizer`. Opisuję też dodatkowe sztuczki związane z dostosowywaniem współczynników uczenia i modyfikowaniem ich w jego trakcie. Pokazuję ponadto, jak zastosować te sztuczki w klasach `Optimizer` i `Trainer`. W końcowej części rozdziału przedstawiam klasę `Dropout`. Jest to nowy rodzaj operacji, który okazał się niezbędny do poprawy stabilności uczenia modeli z dziedziny deep learning. Do dzieła!

[4] Nawet w tym prostym problemie niewielka zmiana hiperparametrów może spowodować, że model z dziedziny deep learning nie będzie lepszy niż dwuwarstwowa sieć neuronowa. Sklonuj repozytorium z serwisu GitHub (*https://oreil.ly/deep-learning-github*) i sam się o tym przekonaj.

Uczenie głębokie: zrozum, zanim zaimplementujesz!

Uczenie głębokie (ang. *deep learning*) zyskuje ostatnio ogromną popularność. Jest to ściśle związane z coraz częstszym zastosowaniem sieci neuronowych w przeróżnych branżach i dziedzinach. W konsekwencji inżynierowie oprogramowania, specjaliści do spraw przetwarzania danych czy osoby w praktyce zajmujące się uczeniem maszynowym muszą zdobyć solidną wiedzę o tych zagadnieniach. Przede wszystkim trzeba dogłębnie zrozumieć podstawy uczenia głębokiego. Dopiero po uzyskaniu biegłości w posługiwaniu się poszczególnymi koncepcjami i modelami możliwe jest wykorzystanie w pełni potencjału tej dynamicznie rozwijającej się technologii.

Ten praktyczny podręcznik, poświęcony podstawom uczenia głębokiego, zrozumiale i wyczerpująco przedstawia zasady działania sieci neuronowych z trzech różnych poziomów: matematycznego, obliczeniowego i konceptualnego. Takie podejście wynika z faktu, że dogłębne zrozumienie sieci neuronowych wymaga nie jednego, ale kilku modeli umysłowych, z których każdy objaśnia inny aspekt działania tych sieci. Zaprezentowano tu również techniki implementacji poszczególnych elementów w języku Python, co pozwala utworzyć działające sieci neuronowe. Dzięki tej książce stanie się jasne, w jaki sposób należy tworzyć, uczyć i wykorzystywać wielowarstwowe, konwolucyjne i rekurencyjne sieci neuronowe w różnych praktycznych zastosowaniach.

W książce między innymi:

- matematyczne podstawy uczenia głębokiego
- tworzenie modeli do rozwiązywania praktycznych problemów
- standardowe i niestandardowe techniki treningu sieci neuronowych
- rozpoznawanie obrazów za pomocą konwolucyjnych sieci neuronowych
- rekurencyjne sieci neuronowe, ich działanie i implementacja
- praca z wykorzystaniem biblioteki PyTorch

Seth Weidman specjalizuje się w nauce o danych (ang. *data science*). Przez wiele lat prowadził szkolenia w zakresie uczenia maszynowego. Obecnie buduje modele uczenia maszynowego dla zespołu odpowiedzialnego za infrastrukturę w Facebooku. Pasjonuje go objaśnianie złożonych zagadnień w możliwie prosty sposób. Uważa, że po drugiej stronie złożoności znajduje się prostota.

ISBN 978-83-283-6597-1