OKIEM EKSPERTA

Python

Uczenie maszynowe w przykładach

Najlepsze praktyki w realnych zastosowaniach

Wydanie IV

Yuxi (Hayden) Liu

Helion

packt

Tytuł oryginału: Python Machine Learning By Example: Unlock machine learning best practices with real-world use cases, 4th Edition

Tłumaczenie: Piotr Rajca, z wykorzystaniem fragmentów poprzedniego wydania w przekładzie Andrzeja Watraka

ISBN: 978-83-289-3171-8

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/pytuc4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 11

Kategoryzacja zdjęć odzieży przy użyciu konwolucyjnej sieci neuronowej

W poprzednim rozdziale zostały opisane dobre praktyki ogólnego, tradycyjnego uczenia maszynowego. Począwszy od tego rozdziału zajmiemy się bardziej zaawansowanymi zagadnieniami: uczeniem głębokim i uczeniem przez wzmacnianie.

Klasyfikacja obrazów zwykle polega na ich spłaszczaniu, generowaniu wektorów pikseli i przesyłaniu ich do sieci neuronowej lub innego modelu. W ten sposób można wprawdzie rozwiązać problem, ale traci się cenne informacje przestrzenne. W tym rozdziale będziemy wykorzystywać konwolucyjną sieć neuronową do wyodrębniania z obrazów bogatych, różnorodnych informacji. Dowiesz się, jak sieć rozpoznaje, że obraz cyfry 9 to cyfra 9, 4 to 4, kot to kot, a pies to pies.

Zaczniemy od przedstawienia bloków konstrukcyjnych sieci konwolucyjnej. Następnie zbudujemy przy użyciu pakietu TensorFlow sieć klasyfikującą zdjęcia odzieży. W ten sposób zdemistyfikujemy mechanizm działania sieci konwolucyjnej. Na koniec zajmiemy się uzupełnianiem danych i zwiększaniem skuteczności modeli sieci.

W tym rozdziale są opisane następujące zagadnienia:

- prezentacja bloków konstrukcyjnych konwolucyjnej sieci neuronowej,
- budowanie sieci konwolucyjnej do celów klasyfikacji,
- poznawanie zbioru danych ze zdęciami odzieży,
- klasyfikacja zdjęć odzieży przy użyciu sieci konwolucyjnej,
- wzmacnianie klasyfikatora przy wykorzystaniu uzupełniania danych,
- rozwijanie klasyfikatora z wykorzystaniem uczenia transferowego.

Prezentacja bloków konstrukcyjnych konwolucyjnej sieci neuronowej

Zwykłe, w pełni połączone warstwy ukryte, którymi zajmowaliśmy się do tej pory, do pewnego stopnia radzą sobie z wyodrębnianiem cech danych. Jednak uzyskane wyniki nie zawsze pozwalają rozróżniać klasy obrazów. Natomiast za pomocą **konwolucyjnej**

sieci neuronowej (ang. *Convolutional Neural Network*, CNN) można wydobywać bogate i różnorodne informacje, na przykład rozpoznawać samochody, samoloty, ręcznie napisane litery. Sieć konwolucyjna to rodzaj sieci neuronowej, o budowie inspirowanej ludzką korą wzrokową. Aby zdemistyfikować tego rodzaju sieć, zacznijmy od zapoznania się z jej komponentami — warstwami: konwolucyjną, nieliniową i redukującą.

Warstwa konwolucyjna

Pierwszą lub kilka pierwszych warstw sieci konwolucyjnej stanowią **warstwy konwolucyjne**. Danymi wejściowymi są obrazy lub macierze.

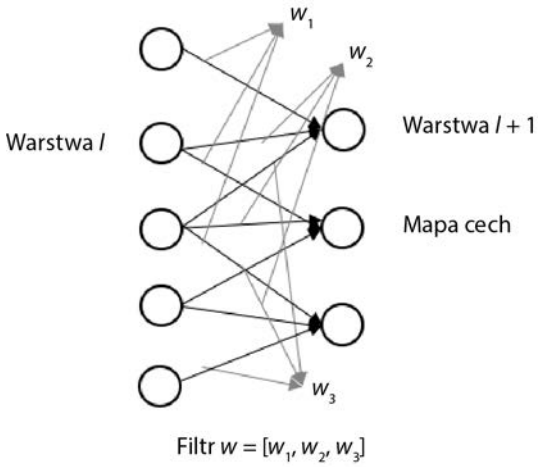
W praktyce konwolucyjna sieć neuronowa, a konkretnie jej warstwy konwolucyjne, funkcjonują podobnie jak komórki wzrokowe:

- Kora wzrokowa człowieka składa się z komórek nerwowych, posiadających wrażliwe podobszary pola widzenia, nazywane **polami recepcyjnymi**. Komórki reagują silniej lub słabiej na linie znajdujące się w polu widzenia w określonym położeniu. Na przykład niektóre komórki reagują tylko na pionowe linie proste, a inne na poziome. Wszystkie komórki razem wzięte tworzą system wzrokowy, przy czym każda komórka jest wyspecjalizowanym komponentem. Warstwa konwolucyjna w sieci zawiera zbiór filtrów funkcjonujących podobnie jak komórki w korze wzrokowej.
- Prosta komórka reaguje tylko wtedy, gdy w jej podobszarze recepcyjnym znajdzie się określona linia. Bardziej złożona komórka jest wrażliwa na większe podobszary i reaguje na linie proste znajdujące się w całym polu widzenia. Stos warstw konwolucyjnych jest zbiorem złożonych komórek, które wspólnie mogą wykrywać wzorce w większym zakresie.

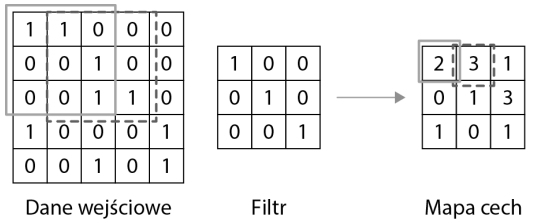
Warstwa konwolucyjna przetwarza przesyłane obrazy lub macierze i funkcjonuje podobnie jak komórki nerwowe, reagujące na sygnały odbierane za pomocą pól recepcyjnych. Przetwarza dane wejściowe, wykonując na nich operacje konwolucyjne, czyli mówiąc językiem matematycznym, wylicza iloczyn skalarny wartości zawartych w swoich węzłach i w poszczególnych małych obszarach warstwy wejściowej. Mały obszar jest polem recepcyjnym, natomiast węzeł można uznać za część filtru. W miarę przemieszczania się filtru przez warstwę wejściową wyliczany jest iloczyn skalarny wartości zawartych w tym filtrze i w bieżącym polu recepcyjnym (podobszarze). Gdy filtr przejdzie przez wszystkie podobszary, tworzona jest nowa warstwa, tzw. **mapa cech**. Rysunek 11.1 ilustruje prosty przykład.

W tym przykładzie pierwsza warstwa składa się z pięciu, a filtr z trzech węzłów [w_1, w_2, w_3]. Najpierw jest wyliczany iloczyn skalarny wartości zawartych w filtrze i w pierwszych trzech węzłach pierwszej warstwy. W ten sposób jest tworzony pierwszy węzeł w wyjściowej mapie cech. Następnie jest wyliczany iloczyn skalarny wartości zawartych w filtrze i w trzech środkowych węzłach. Tak powstaje drugi węzeł w mapie cech. Na koniec, na bazie trzech ostatnich węzłów pierwszej warstwy, jest tworzony trzeci węzeł warstwy l — mapy cech.

Teraz przyjrzyjmy się dokładniej procesowi konwolucji, pokazanemu na rysunku 11.2.



Rysunek 11.1. Proces tworzenia mapy cech



Rysunek 11.2. Proces konwolucji

W tym przykładzie filtr o wymiarach 3×3 przemieszcza się po macierzy o wymiarach 5×5 począwszy od lewego górnego podobszaru, w kierunku prawego dolnego. Dla każdego podobszaru jest przy użyciu filtru wyliczany iloczyn skalarny. Na przykład w lewym górnym obszarze (pomarańczowy prostokąt — linia ciągła) jest wykonywane działanie $1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 = 2$. W efekcie lewy górny węzeł w mapie cech uzyskuje wartość 2. Dla kolejnego podobszaru (niebieski prostokąt — linia przerywana) iloczyn jest równy $1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 3$ i taką wartość uzyskuje następny węzeł (wewnątrz tego prostokąta) w wynikowej mapie cech. Ostatecznie powstaje mapa cech o wymiarach 3×3.

Do czego są wykorzystywane warstwy konwolucyjne? Do wyodrębniania takich cech, jak linie proste i krzywe. Jeżeli pole recepcyjne zawiera linię prostą lub krzywą, rozpoznaną przez filtr, wówczas piksel w wynikowej mapie cech uzyskuje wysoką wartość. W powyższym przykładzie filtr definiuje odwrotny ukośnik (\), a w polu recepcyjnym wewnątrz niebieskiego prostokąta znajduje się podobna linia. W efekcie tworzony jest piksel zawierający najwyższą wartość 3. Natomiast pole recepcyjne w prawym górnym rogu nie zawiera ukośnika, więc piksel w wynikowej mapie cech ma wartość 0. Zatem filtr konwolucyjny działa jak detektor krzywych lub kształtów.

Warstwa konwolucyjna zazwyczaj zawiera wiele filtrów wykrywających różne linie krzywe i kształty. W powyższym prostym przykładzie został użyty jeden filtr i utworzona została jedna mapa cech określająca, jak wiernie obraz wejściowy oddaje krzywą

zapisaną w filtrze. Aby wykrywać w danych wejściowych więcej wzorców, stosuje się dodatkowe filtry, definiujące na przykład linie poziomą, pionową, nachyloną pod kątem 30° i prostopadłe do siebie.

Aby uzyskać reprezentacje wyższego poziomu, na przykład podstawowe kształty lub kontury, stosuje się kilka warstw konwolucyjnych. Łącząc większe liczby warstw, uzyskuje się większe pola recepcyjne, umożliwiające rozpoznawanie kolejnych ogólnych kształtów.

Często za każdą warstwą konwolucyjną znajduje się warstwa nieliniowa.

Warstwa nieliniowa

Warstwa nieliniowa jest w zasadzie warstwą aktywacji, którą poznałeś w rozdziale 6., „Prognozowanie cen akcji za pomocą sieci neuronowych". Oczywiście jej zadaniem jest wprowadzanie nieliniowości. Jak już wiesz, w warstwie konwolucyjnej są wykonywane tylko operacje liniowe, na przykład mnożenie lub dodawanie. Sieć neuronowa zawierająca wyłącznie liniowe warstwy ukryte niezależnie od ich liczby funkcjonuje jak perceptron jednowarstwowy. Dlatego zaraz po warstwie konwolucyjnej niezbędne jest wprowadzanie nieliniowych aktywacji. Jak zawsze najlepszą potencjalną funkcją nieliniową w głębokiej sieci neuronowej jest ReLU.

Warstwa redukująca

Zazwyczaj jedna lub kilka warstw konwolucyjnych (wraz z aktywacją nieliniową) generuje cechy, które można bezpośrednio wykorzystywać w klasyfikacji. Na przykład na potrzeby klasyfikacji wieloklasowej można zastosować funkcję softmax. Wykonajmy jednak najpierw kilka obliczeń matematycznych.

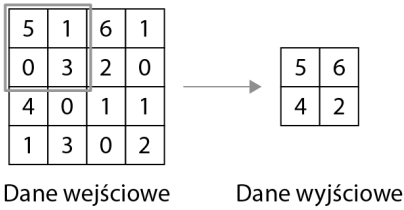
Załóżmy, że obraz wejściowy ma wymiary 28×28, a pierwsza warstwa konwolucyjna zawiera 20 filtrów, każdy o wymiarach 5×5. Uzyskujemy wtedy 20 map cech, każda o wielkości $(28 - 5 + 1) \cdot (28 - 5 + 1) = 24 \cdot 24 = 576$. Oznacza to, że liczba cech stanowiących dane wejściowe dla następnej warstwy wzrasta z $28 \cdot 28 = 784$ do $20 \cdot 576 = 11\,520$. Jeżeli w drugiej warstwie konwolucyjnej będzie 20 filtrów, każdy o wymiarach 5×5, to liczba cech wyjściowych wzrasta do $50 \cdot 20 \cdot (24 - 5 + 1) \cdot (24 - 5 + 1) = 400\,000$, czyli znacznie więcej od początkowej liczby 784. Wymiarowość danych gwałtownie rośnie wraz z każdą warstwą konwolucyjną aż do końcowej warstwy softmax. Jest to niekorzystny efekt, ponieważ model może łatwo nadmiernie się dopasować do danych, nie mówiąc już o kosztach jego treningu przy tak dużej liczbie wag.

Aby rozwiązać ten problem, często po warstwach konwolucyjnej i nieliniowej wprowadza się **warstwę redukującą** (ang. *pooling layer*), nazywaną również **warstwą próbkującą w dół** (ang. *downsampling layer*). Jak sugeruje nazwa, warstwa ta zmniejsza wymiarowość mapy cech. Agreguje w tym celu informacje o cechach znajdujących się w poszczególnych podobszarach. Typowe techniki redukcji są następujące:

- maksymalizacja, polegająca na wybieraniu maksymalnych wartości zawartych w nienakładających się podobszarach;

- uśrednianie, polegające na uśrednianiu wartości zawartych w nienakładających się podobszarach.

W przykładzie pokazanym na rysunku 11.3 zastosowany jest filtr maksymalizujący o wymiarach 2×2, przekształcający mapę o wymiarach 4×4 na wynikową mapę 2×2.

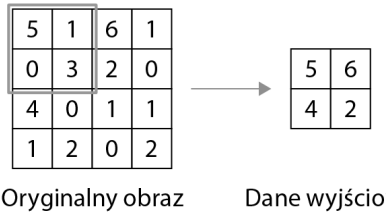


Rysunek 11.3. Zasada funkcjonowania warstwy redukującej

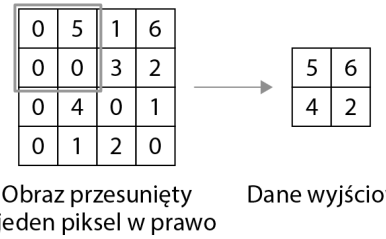
Dodatkową zaletą warstwy redukującej, oprócz redukcji wymiarowości, jest niezmienność translacji. Oznacza to, że wyjście nie zmienia się nawet wtedy, gdy macierz wejściowa ulega niewielkim przekształceniom. Na przykład jeśli obraz wejściowy zostanie przesunięty o kilka pikseli w lewo lub w prawo, dane wyjściowe maksymalizującej warstwy redukującej pozostaną niezmienne, o ile wartości pikseli w poszczególnych podobszarach będą takie same. Innymi słowy, prognozy są wtedy mniej wrażliwe na położenie obrazów.

Rysunek 11.4 ilustruje niezmienność translacji w warstwie redukującej. Przedstawia obraz o wymiarach 4×4 i wyjście warstwy redukującej o wymiarach 2×2.

Rysunek 11.5 pokazuje wynik uzyskany po przesunięciu obrazu o jeden piksel w prawo.



Rysunek 11.4. Przesunięty oryginalny obraz i wyjście maksymalizującej warstwy redukującej



Rysunek 11.5. Przesunięty obraz i uzyskany wynik

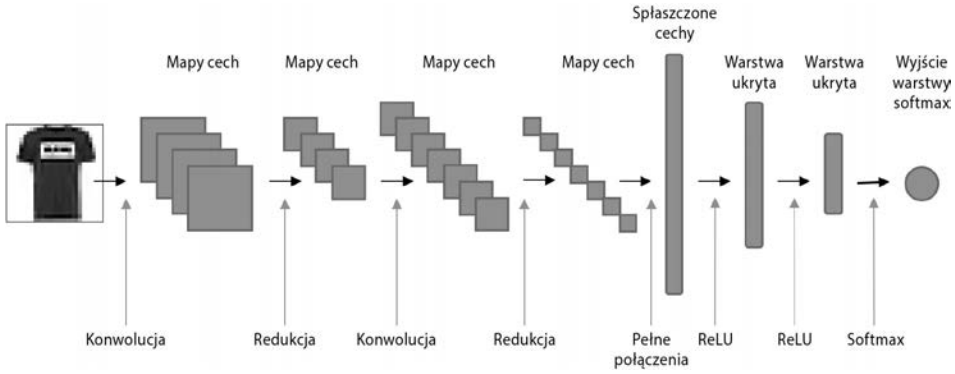
Wynik jest taki sam, mimo że obraz został przesunięty w poziomie. W ten sposób warstwa redukująca poprawia stabilność translacji obrazów.

Poznałeś bloki konstrukcyjne konwolucyjnej sieci neuronowej. Są prostsze, niż przypuszczałeś, prawda? Teraz dowiedz się, jak buduje się z nich sieć konwolucyjną.

Budowanie konwolucyjnej sieci neuronowej na potrzeby klasyfikacji

Wykorzystując trzy opisane wcześniej typy warstw oraz warstwy w pełni połączone, można zbudować strukturę modelu klasyfikacyjnego, opartego na konwolucyjnej sieci neuronowej. Ilustruje to rysunek 11.6.

W tym przykładzie obrazy wejściowe są wprowadzane do złożonej z kilku filtrów warstwy konwolucyjnej, wykorzystującej aktywację ReLU. Współczynniki filtrów mogą być modyfikowane podczas treningu. Dobrze wytrenowana pierwsza warstwa konwolucyjna dostarcza dobrych niskopoziomowych reprezentacji obrazów wejściowych, co ma krytyczne znaczenie dla kolejnych warstw konwolucyjnych (jeżeli są), jak również dalszych zadań klasyfikacyjnych. Każda wynikowa mapa cech jest następnie próbkowana w dół przez warstwę redukującą.



Rysunek 11.6. Architektura konwolucyjnej sieci neuronowej

Zagregowane mapy cech są wprowadzane do drugiej warstwy konwolucyjnej. Podobnie jak poprzednio, druga warstwa redukująca zmniejsza rozmiar wyjściowej mapy cech. Można tworzyć dowolnie długi łańcuch par zawierających warstwy konwolucyjną i redukującą. Druga warstwa konwolucyjna (i następne, jeżeli są) próbuje generować na podstawie serii niskopoziomowych reprezentacji uzyskanych z poprzednich warstw wysokopoziomowe reprezentacje, takie jak kształty lub kontury.

Mapy cech są macierzami. Dlatego przed klasyfikacją trzeba je spłaszczyć do wektorów. Spłaszczone cechy są danymi wejściowymi dla jednej lub kilku w pełni połączonych warstw ukrytych. Konwolucyjną sieć neuronową można traktować jako hierarchicznego ekstraktora cech, opartego na zwykłej sieci neuronowej. Sieć konwolucyjna jest dobrze przystosowana do identyfikowania silnych, unikatowych cech wyróżniających obrazy.

Na końcu sieci znajduje się funkcja logistyczna (w przypadku klasyfikacji binarnej), funkcja softmax (w przypadku klasyfikacji wieloklasowej) lub zbiór funkcji logistycznych (w przypadku wielu etykiet).

W tym momencie powinieneś już dobrze rozumieć funkcjonowanie konwolucyjnej sieci neuronowej i być gotowy do rozwiązania problemu klasyfikacji zdjęć odzieży. Zacznijmy od zbadania zbioru danych.

Badanie zbioru zdjęć odzieży

Zbiór Fashion-MNIST (*https://github.com/zalandoresearch/fashion-mnist*) zawiera zdjęcia produktów marki Zalando — największego w Europie internetowego sklepu odzieżowego. Zbiór składa się z 60 000 próbek treningowych i 10 000 testowych. Każda próbka jest obrazem o wymiarach 28×28 pikseli w odcieniach szarości. Obrazowi jest przypisana jedna z następujących 10 klas odzieży:

- 0: t-shirt/top,
- 1: spodnie,
- 2: sweter,
- 3: sukienka,
- 4: płaszcz,
- 5: sandały,
- 6: koszula,
- 7: tenisówki,
- 8: torebka,
- 9: buty.

Firma Zalando nazwała swój zbiór Fashion-MNIST, aby stał się równie popularny jak wykorzystywany do analiz porównawczych zbiór MNIST odręcznie napisanych cyfr.

Zbiór ten możesz łatwo pobrać, klikając odnośnik *Download* na stronie serwisu GitHub lub korzystając z modułu PyTorch, który oprócz zbioru zawiera odpowiedni interfejs API. W poniższym przykładzie wykorzystana jest druga możliwość.

```
>>> import torch, torchvision
>>> from torchvision import transforms
>>> image_path = './'
>>> transform = transforms.Compose([transforms.ToTensor()])
>>> train_dataset = torchvision.datasets.FashionMNIST(root=image_path,
...                                                   train=True,
...                                                   transform=transform,
...                                                   download=True)
>>> test_dataset = torchvision.datasets.FashionMNIST(root=image_path,
...                                                  train=False,
...                                                  transform=transform,
...                                                  download=False)
```

W powyższym przykładzie zaimportowałeś pakiet `torchvision`, część biblioteki PyTorch zapewniającą dostęp do zbiorów danych, architektur modeli oraz przeróżnych narzędzi

do przekształcania obrazów służących do wykonywania zadań związanych z wizją komputerową.

Uwaga

Biblioteka `torchvision` zawiera następujące kluczowe elementy:

- **Zbiory danych i narzędzia do ładowania danych**: Pakiet `torchvision.datasets` umożliwia dostęp do standardowych zbiorów danych wykorzystywanych w zadaniach takich jak klasyfikacja obrazów, detekcja obiektów, segmentacja semantyczna. Przykładowe zbiory danych to: MNIST, CIFAR-10, ImageNet, FashionMNIST. Klasa `torch.utils.data.DataLoader` ułatwia tworzenie obiektów wczytujących dane, które wydajnie ładują i wstępnie przetwarzają partie danych ze zbiorów danych.
- **Przekształcenia**: Moduł `torchvision.transforms` oferuje szeroki wybór narzędzi do przekształcania obrazów, służących do wzbogacania danych, normalizacji i wstępnego przetwarzania. Do najczęściej używanych przekształceń należą zmiana rozmiaru, przycinanie i normalizacja.
- **Architektury modeli**: Moduł `torchvision.models` udostępnia wstępnie wytrenowane architektury modeli przeznaczone do różnych zadań z zakresu komputerowego rozpoznawania obrazów.
- **Narzędzia pomocnicze**: Moduł `torchvision.utils` zawiera funkcje pomocnicze, które umożliwiają wizualizację obrazów, konwersję obrazów na różne formaty i wykonywanie innych operacji.

Zbiór danych Fashion-MNIST, który właśnie wczytałeś, zawiera gotowy sposób podziału danych na zbiory treningowy i testowy. Zbiór treningowy jest przechowywany w miejscu określonym przez zmienną `image_path`. Kolejnym krokiem będzie przekształcenie tych zbiorów do formatu Tensor. Wyniki zwracane przez te dwa obiekty zbiorów danych zawierają dodatkowe informacje na ich temat:

```
>>> print(train_dataset)
Dataset FashionMNIST
    Number of datapoints: 60000
    Root location: ./
    Split: Train
    StandardTransform
Transform: Compose(
    ToTensor()
   )
>>> print(test_dataset)
Dataset FashionMNIST
    Number of datapoints: 10000
    Root location: ./
    Split: Test
    StandardTransform
Transform: Compose(
               ToTensor()
           )
```

Jak widać, składają się one z 60 000 próbek treningowych i 10 000 próbek testowych.

Teraz wczytaj zbiór testowy we fragmentach zawierających po 64 próbki:

```
>>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(42)
>>> train_dl = DataLoader(train_dataset, batch_size, shuffle=True)
```

W bibliotece PyTorch `DataLoader` to klasa narzędziowa, która umożliwia efektywne ładowanie i wstępne przetwarzanie danych z zestawu danych podczas treningu lub ewaluacji modeli uczenia maszynowego. W praktyce `DataLoader` stanowi opakowanie dla zestawu danych i udostępnia metody pozwalające na iterowanie po partiach danych. Jest to szczególnie przydatne podczas pracy z dużymi zbiorami danych, które nie mieszczą się w całości w pamięci operacyjnej.

Uwaga

Kluczowe cechy klasy `DataLoader`:

- **Podział na partie**: Potrafi automatycznie dzielić zestaw danych na partie o określonej wielkości, pozwalając na ich stosowanie w procesie trenowania modeli.
- **Losowa zmiana kolejności**: Przypisanie parametrowi `shuffle` wartości `True` zapewnia możliwość losowej zmiany kolejności danych przed rozpoczęciem każdego treningu, co pozwala zredukować obciążenie i poprawić konwergencję

Jeśli chcesz, sprawdź próbki obrazów oraz ich etykiety z pierwszej partii danych:

```
>>> data_iter = iter(train_dl)
>>> images, labels = next(data_iter)
>>> print(labels)
tensor([5, 7, 4, 7, 3, 8, 9, 5, 3, 1, 2, 3, 2, 3, 3, 7, 9, 9, 3, 2, 4, 6, 3, 5, 5,
3, 2, 0, 0, 8, 4, 2, 8, 5, 9, 2, 4, 9, 4, 4, 3, 4, 9, 7, 2, 0, 4, 5, 4, 8, 2, 6,
7, 0, 2, 0, 6, 3, 3, 5, 6, 0, 0, 8])
```

Etykiety nie są nazwami klas. Zdefiniuj je więc jak niżej. Wykorzystasz je później do wyświetlenia wyników:

```
>>> class_names = ['T-shirt/top', 'Spodnie', 'Sweter', 'Sukienka', 'Płaszcz',
'Sandały', 'Koszula', 'Tenisówki', 'Torebka', 'Buty']
```

Sprawdź format obrazów:

```
>>> print(images[0].shape)
torch.Size([1, 28, 28])
>>> print(torch.max(images), torch.min(images))
tensor(1.) tensor(0.)
```

Każda próbka jest obrazem o wymiarach 28×28 pikseli.

Wyświetl próbkę treningową za pomocą poniższego kodu:

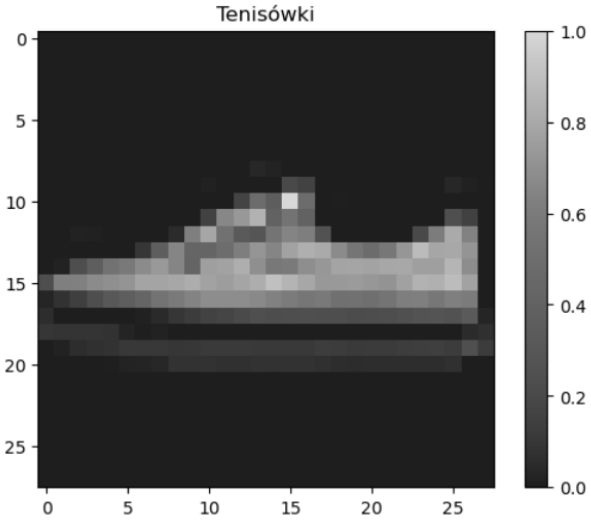
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> npimg = images[1].numpy()
>>> plt.imshow(np.transpose(npimg, (1, 2, 0)))
```

```
>>> plt.colorbar()
>>> plt.title(class_names[labels[1]])
>>> plt.show()
```

Uwaga

W bibliotece PyTorch podczas wyświetlania obrazów przy użyciu `matplotlib` należy zastosować wywołanie: `np.transpose(npimg, (1, 2, 0))`. Krotka `(1, 2, 0)` określa nową kolejność wymiarów. W PyTorch obrazy są reprezentowane w formacie: kanały, wysokość, szerokość. Wywołanie `np.transpose(npimg, (1, 2, 0))` powoduje zmianę kolejności wymiarów tablicy obrazu, tak by odpowiadały one wymaganiom biblioteki `matplotlib`.

Rysunek 11.7 przedstawia uzyskany wynik.



Rysunek 11.7. Próbka treningowa ze zbioru Fashion-MNIST

W podobny sposób możesz wyświetlić pierwszych 16 próbek treningowych:

```
>>> for i in range(16):
...     plt.subplot(4, 4, i + 1)
...     plt.subplots_adjust(hspace=.3)
...     plt.xticks([])
...     plt.yticks([])
...     npimg = images[i].numpy()
...     plt.imshow(np.transpose(npimg, (1, 2, 0)), cmap="Greys")
...     plt.title(class_names[labels[i]])
>>> plt.show()
```

Rysunek 11.8 przedstawia uzyskany wynik.

W następnym podrozdziale zbudujesz model sieci konwolucyjnej klasyfikującej zdjęcia odzieży.

Rysunek 11.8. Wynik wstępnego przetworzenia zbioru treningowego

Klasyfikowanie zdjęć odzieży za pomocą konwolucyjnej sieci neuronowej

Jak wspomniałem wcześniej, model konwolucyjnej sieci neuronowej składa się z dwóch komponentów: ekstraktora cech zbudowanego z warstw konwolucyjnych i redukujących oraz klasyfikatora podobnego do zwykłej sieci neuronowej.

Zacznijmy ten projekt od przygotowania modelu konwolucyjnej sieci neuronowej.

Tworzenie sieci

Zaimportuj niezbędny moduł i zainicjuj model `Sequential`:

```
>>> import torch.nn as nn
>>> model = nn.Sequential()
```

Ekstraktor cech będzie się składał z trzech warstw konwolucyjnych. Pierwsza niech zawiera 32 niewielkie filtry, każdy o wymiarach 3×3. Zaimplementuj ją za pomocą poniższego kodu:

```
>>> model.add_module('conv1',
...                  nn.Conv2d(in_channels=1,
...                            out_channels=32,
...                            kernel_size=3)
...                 )
>>> model.add_module('relu1', nn.ReLU())
```

Zwróć uwagę na użytą funkcję aktywacji ReLU.

Po warstwie konwolucyjnej będzie się znajdować maksymalizująca warstwa redukująca, zawierająca filtr o wymiarach 2×2:

```
>>> model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
```

Teraz dodaj drugą warstwę konwolucyjną, zawierającą 64 filtry, każdy o wymiarach 3×3 oraz — jak poprzednio — funkcję aktywacji ReLU:

```
>>> model.add_module('conv2',
...                  nn.Conv2d(in_channels=32,
...                  out_channels=64,
...                  kernel_size=3)
...                 )
>>> model.add_module('relu2', nn.ReLU())
```

Po drugiej warstwie konwolucyjnej umieść następną maksymalizującą warstwę redukującą, zawierającą filtr o wymiarach 2×2:

```
>>> model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
```

Następnie dodaj trzecią warstwę konwolucyjną, tym razem zawierającą 128 filtrów, każdy o wymiarach 3×3:

```
>>> model.add_module('conv3',
...                  nn.Conv2d(in_channels=64,
...                  out_channels=128,
...                  kernel_size=3)
...                 )
>>> model.add_module('relu3', nn.ReLU())
```

Zróbmy sobie w tym miejscu krótką przerwę, by sprawdzić, jak wyglądają uzyskane mapy filtrów. W tym celu wczytaj do modelu grupę 64 losowo wybranych próbek:

```
>>> x = torch.rand((64, 1, 28, 28))
>>> print(model(x).shape)
torch.Size([64, 128, 3, 3])
```

W tym przypadku przekazałeś dane wejściowe o kształcie określonym jako `(64, 1, 28, 28)`, czyli jedną grupę 64 obrazów o wymiarach 28×28, i uzyskałeś wyniki o kształcie `(64, 128, 3, 3)`, czyli mapy cech zawierające 128 kanałów o wymiarach 3×3.

Następnie musisz spłaszczyć uzyskaną mapę cech, aby wprowadzić je do klasyfikatora, który znajdzie się w dalszej części sieci:

```
>>> model.add_module('flatten', nn.Flatten())
```

W efekcie spłaszczyłeś do wymiarów (64, 1152), jak widać poniżej:

```
>>> print(model(x).shape)
torch.Size([64, 1152])
```

Jako klasyfikatora użyj ukrytej warstwy składającej się z 64 węzłów:

```
>>> model.add_module('fc1', nn.Linear(1152, 64))
>>> model.add_module('relu4', nn.ReLU())
```

Warstwa ukryta jest zwykłą, w pełni połączoną warstwą zagęszczoną, wykorzystującą funkcję aktywacji ReLU.

Ostatnia warstwa, generująca wyniki, niech składa się z 10 węzłów reprezentujących 10 klas. Funkcją aktywacji niech będzie softmax:

```
>>> model.add_module('fc2', nn.Linear(64, 10))
>>> model.add_module('output', nn.Softmax(dim = 1))
```

Przyjrzyjmy się architekturze utworzonego modelu:

```
>>> print(model)
Sequential(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (relu3): ReLU()
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=1152, out_features=64, bias=True)
  (relu4): ReLU()
  (fc2): Linear(in_features=64, out_features=10, bias=True)
  (output): Softmax(dim=1)
)
```

Jeśli będziesz chciał wyświetlić szczegółowe informacje na temat każdej z warstw tego modelu, w tym kształt danych wyjściowych każdej z warstw i liczbę ich parametrów, które można trenować, możesz to zrobić za pomocą biblioteki `torchsummary`. Zainstaluj ją, używając `pip`, jak pokazano poniżej:

```
>>> pip install torchsummary
>>> from torchsummary import summary
>>> summary(model, input_size=(1, 28, 28), batch_size=-1, device="cpu")
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 26, 26]             320
              ReLU-2           [-1, 32, 26, 26]               0
         MaxPool2d-3           [-1, 32, 13, 13]               0
            Conv2d-4           [-1, 64, 11, 11]          18,496
              ReLU-5           [-1, 64, 11, 11]               0
         MaxPool2d-6             [-1, 64, 5, 5]               0
            Conv2d-7            [-1, 128, 3, 3]          73,856
```

```
              ReLU-8             [-1, 128, 3, 3]               0
            Flatten-9                  [-1, 1152]               0
           Linear-10                    [-1, 64]          73,792
             ReLU-11                    [-1, 64]               0
           Linear-12                    [-1, 10]             650
          Softmax-13                    [-1, 10]               0
================================================================
Total params: 167,114
Trainable params: 167,114
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.53
Params size (MB): 0.64
Estimated Total Size (MB): 1.17
----------------------------------------------------------------
```

Powyższy wynik zawiera informacje o poszczególnych warstwach modelu, kształtach danych wyjściowych każdej z nich oraz o liczbach trenowanych parametrów. Z pewnością zauważyłeś, że wyjście warstwy konwolucyjnej ma trzy wymiary. Pierwsze dwa są wymiarami mapy cech, a trzecim jest liczba użytych filtrów. Wielkość wyjścia (pierwsze dwa wymiary) maksymalizującej warstwy redukującej jest równa połowie wielkości wejściowej mapy cech. Mapy cech są próbkowane w dół za pomocą warstwy redukującej. Zapewne chciałbyś wiedzieć, ile parametrów musiałbyś trenować, gdyby nie było warstw redukujących. Liczba ta wyniosłaby 4 058 314! Jak widać, korzyści ze stosowania warstw redukujących są oczywiste: zapobiega się nadmiernemu dopasowaniu modelu i zmniejsza koszt jego treningu.

Być może zastanawiasz się, dlaczego kolejne warstwy konwolucyjne zawierają coraz więcej filtrów. Przypomnę, że każda warstwa usiłuje identyfikować hierarchiczne wzorce danych. W pierwszej są to linie proste, kropki i linie krzywe. Każda kolejna warstwa łączy wzorce uzyskane z warstwy poprzedniej i tworzy wzorce wyższego poziomu, na przykład kształty i kontury. W większości przypadków w kolejnych warstwach kombinacji wzorców jest coraz więcej. Dlatego liczba filtrów też musi rosnąć (lub w ostateczności nie może się zmniejszać).

Trening sieci

Czas wytrenować zbudowany model sieci.

Najpierw skompiluj model, używając optymalizatora Adam, entropii krzyżowej jako funkcji straty i dokładności klasyfikacji jako metryki:

```
>>> device = torch.device("cuda:0")
# device = torch.device("cpu")
>>> model = model.to(device)
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

W tym przypadku do trenowania zastosowaliśmy procesor graficzny (GPU) — z tego względu użyte zostało wywołanie `torch.device("cuda:0")` (które odwołuje się do pierwszego urządzenia, o indeksie 0) — i na nim zapisaliśmy tensor. Trenowanie

modelu na procesorze głównym (CPU) jest alternatywnym rozwiązaniem — zadziała, lecz będzie wolniejsze.

Następnie wytrenuj model, używając do tego celu następującej funkcji:

```
>>> def train(model, optimizer, num_epochs, train_dl):
...     for epoch in range(num_epochs):
...         loss_train = 0
...         accuracy_train = 0
...         for x_batch, y_batch in train_dl:
...             x_batch = x_batch.to(device)
...             y_batch = y_batch.to(device)
...             pred = model(x_batch)
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_train += loss.item() * y_batch.size(0)
...             is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
...             accuracy_train += is_correct.sum().cpu()
...
...         loss_train /= len(train_dl.dataset)
...         accuracy_train /= len(train_dl.dataset)
...
...         print(f'Epoka {epoch+1} - strata: {loss_train:.4f} - dokładność:
{accuracy_train:.4f}')
```

Wytrenuj model w 30 iteracjach, monitorując przy tym postępy procesu uczenia

```
>>> num_epochs = 30
>>> train(model, optimizer, num_epochs, train_dl)
Epoka 1 - strata: 1.7253 - dokładność: 0.7385
Epoka 2 - strata: 1.6333 - dokładność: 0.8287
…
Epoka 10 - strata: 1.5572 - dokładność: 0.9041
…
Epoka 20 - strata: 1.5344 - dokładność: 0.9270
...
Epoka 29 - strata: 1.5249 - dokładność: 0.9362
Epoka 30 - strata: 1.5249 - dokładność: 0.9363
```

Dokładność na zbiorze treningowym wyniosła 94%. Kolejny przykład pokazuje, jak można sprawdzić dokładność uzyskiwaną na zbiorze testowym:

```
>>> test_dl = DataLoader(test_dataset, batch_size, shuffle=False)
>>> def evaluate_model(model, test_dl):
...     accuracy_test = 0
...     with torch.no_grad():
...        for x_batch, y_batch in test_dl:
...            pred = model.cpu()(x_batch)
...            is_correct = torch.argmax(pred, dim=1) == y_batch
...            accuracy_test += is_correct.float().sum().item()
...
...     print(f'Dokładność na zbiorze testowym: {100 * accuracy_test / 10000} %')
>>> evaluate_model(model, test_dl)
Dokładność na zbiorze testowym: 90.25 %
```

Na zbiorze testowym model osiągnął dokładność 90%. Zwróć uwagę, że ten wynik może się zmieniać w zależności od takich czynników jak różnice w inicjalizacji warstw ukrytych lub niedeterministyczny sposób działania procesora graficznego.

Najlepsze praktyki

Operacje, które lepiej nadają się do wykonywania na procesorze graficznym niż na procesorze głównym, to zazwyczaj zadania, które można realizować równolegle i które korzystają z ogromnej równoległości oraz mocy obliczeniowej oferowanej przez architekturę GPU. Oto kilka przykładów:

- Operacje macierzowe oraz konwolucyjne.
- Przetwarzanie dużych partii danych jednocześnie. Zadania obejmujące przetwarzanie wsadowe, takie jak uczenie i wnioskowanie na fragmentach danych w modelach uczenia maszynowego, korzystają z równoległego przetwarzania, które można realizować na procesorach graficznych.

Propagacja w przód i wstecz w sieciach neuronowych, która zwykle, dzięki akceleracji sprzętowej, przebiega szybciej na procesorach graficznych.

Najlepsze praktyki

Operacje, które lepiej wykonywać na procesorze głównym niż na procesorze graficznym, to zazwyczaj zadania trudne do zrównoleglenia, wymagające przetwarzania sekwencyjnego lub dotyczące niewielkich ilości danych. Oto kilka przykładów:

- Wstępne przetwarzanie, takie jak ładowanie danych, ekstrakcja cech oraz uzupełnianie danych.
- Wnioskowanie na małych modelach. W przypadku małych modeli lub zadań inferencyjnych o niskich wymaganiach obliczeniowych wykonywanie operacji na procesorze głównym może być bardziej opłacalne.
- Operacje związane z przepływem sterowania. Operacje obejmujące instrukcje warunkowe lub pętle są zazwyczaj bardziej efektywne na procesorze głównym ze względu na jego sekwencyjny charakter przetwarzania.

Wiesz już, jak działa wytrenowany model, i zapewne zastanawiasz się, jak wyglądają jego filtry konwolucyjne. Dowiesz się tego w następnym punkcie.

Wizualizacja filtrów konwolucyjnych

Aby wyodrębnić filtry konwolucyjne z wytrenowanego modelu i zwizualizować je, wykonaj poniższe kroki:

1. Na podstawie podsumowania modelu wiemy, że warstwami konwolucyjnymi naszego modelu są warstwy: `conv1`, `conv2` i `conv3`. Wyodrębnij filtry na przykład z trzeciej warstwy konwolucyjnej:

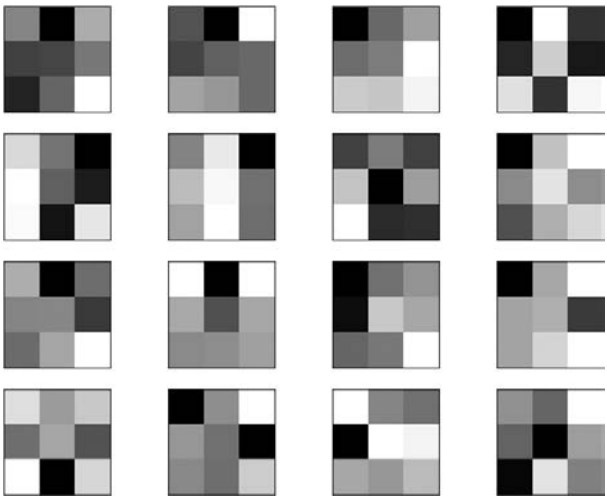
```
>>> conv3_weight = model.conv3.weight.data
>>> print(conv3_weigth.shape)
torch.Size([128, 64, 3, 3])
```

   Wyraźnie widać, że istnieje 128 filtrów, a każdy z nich ma wymiary 3×3 i zawiera 64 kanały.

2. Dla uproszczenia zwizualizuj jedynie pierwszy kanał pierwszych 16 filtrów, przedstawiając je w układzie czterech wierszy i czterech kolumn:

```
>>> n_filters = 16
>>> for i in range(n_filters):
...     weight = conv3_weight[i].cpu().numpy()
...     plt.subplot(4, 4, i+1)
...     plt.xticks([])
...     plt.yticks([])
...     plt.imshow(weight[0], cmap='gray')
... plt.show()
```

   Rysunek 11.9 przedstawia uzyskany wynik.



Rysunek 11.9. Wytrenowane filtry konwolucyjne

Ciemniejsze i jaśniejsze odcienie reprezentują, odpowiednio, mniejsze i większe wagi. Można się domyślić, że filtr w drugim wierszu i drugiej kolumnie identyfikuje pionowe linie znajdujące się w polu recepcyjnym, natomiast trzeci filtr w pierwszym wierszu identyfikuje przejścia koloru z jasnego w prawym dolnym rogu do ciemnego w lewym górnym.

Ten przykład polegał na wytrenowaniu klasyfikatora zdjęć odzieży na 60 000 próbek opatrzonych etykietami. W rzeczywistości uzyskanie tak dużego, oznaczonego zbioru nie jest łatwe. Szczególnie opatrywanie próbek etykietami jest kosztowne i czasochłonne. Jak można trenować klasyfikator na ograniczonej liczbie próbek? Jednym z rozwiązań jest uzupełnianie danych.

Wzmacnianie konwolucyjnej sieci neuronowej poprzez uzupełnianie danych

Uzupełnianie danych (ang. *data augmentation*) oznacza powiększanie istniejącego treningowego zbioru danych w celu zwiększenia skuteczności modelu. Unika się w ten sposób kosztów gromadzenia dodatkowych danych i opatrywania ich etykietami. W bibliotece PyTorch uzupełnianie obrazów w czasie rzeczywistym można implementować przy użyciu modułu `torchvision.transforms`.

Obracanie obrazów w poziomie i pionie

Istnieje wiele sposobów uzupełniania danych obrazów. Najprostszy polega na obracaniu ich w poziomie lub pionie. Na przykład uzyskamy nowy obrazek poprzez utworzenie odbicia istniejącego obrazka w poziomie. Aby stworzyć takie odbicie obrazka w poziomie, wystarczy użyć funkcji `transforms.functional.hflip`, jak pokazano poniżej:

```
>>> image = images[1]
>>> img_flipped = transforms.functional.hflip(image)
```

A oto jak można wyświetlić ten nowy obraz:

```
>>> def display_image_greys(image):
...     npimg = image.numpy()
...     plt.imshow(np.transpose(npimg, (1, 2, 0)), cmap="Greys")
...     plt.xticks([])
...     plt.yticks([])
...
>>> plt.figure(figsize=(8, 8))
>>> plt.subplot(1, 2, 1)
>>> display_image_greys(image)
>>> plt.subplot(1, 2, 2)
>>> display_image_greys(img_flipped)
>>> plt.show()
```

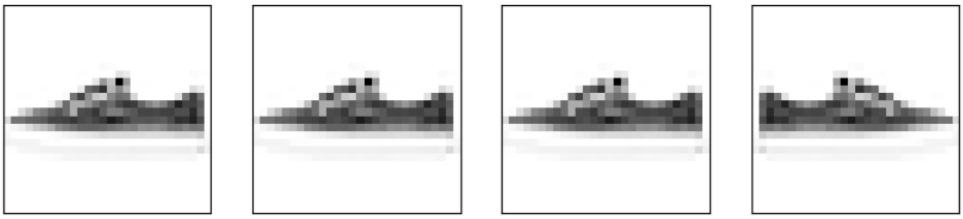
Rysunek 11.10 przedstawia uzyskany wynik.

Rysunek 11.10. Uzupełnianie obrazów poprzez obrócenie w poziomie

Podczas trenowania modeli z wykorzystaniem uzupełniania danych będziemy tworzyć zmodyfikowane obrazy, używając do tego celu generatora losowego. W przypadku obrotów w poziomie skorzystamy z funkcji `transforms.RandomHorizontalFlip`, która losowo obraca obrazek w poziomie z prawdopodobieństwem 50%, co pozwala na efektywne uzupełnienie zbioru danych. Przyjrzyjmy się trzem przykładom:

```
>>> torch.manual_seed(42)
>>> flip_transform = transforms.Compose([transforms.RandomHorizontalFlip()])
>>> plt.figure(figsize=(10, 10))
>>> plt.subplot(1, 4, 1)
>>> display_image_greys(image)
>>> for i in range(3):
...     plt.subplot(1, 4, i+2)
...     img_flip = flip_transform(image)
...     display_image_greys(img_flip)
```

Wyniki powyższego kodu zostały przedstawione na rysunku 11.11.



Rysunek 11.11. Losowe obracanie obrazka w poziomie w celu uzupełniania danych

Jak widać, wygenerowane obrazy są odwrócone lub nie są.

Zazwyczaj obrazy odwrócone w poziomie zawierają te same informacje, co obrazy oryginalne. Odwracanie w pionie stosuje się rzadko, choć można to robić za pomocą klasy `transforms.RandomVerticalFlip`. Należy pamiętać, że odwracanie jest pomocne tylko w modelach niewrażliwych na orientację, na przykład klasyfikujących zdjęcia kotów i psów lub rozpoznających części samochodów. Natomiast jest bardzo niebezpieczne w sytuacjach, gdy orientacja jest ważna, na przykład w klasyfikowaniu znaków skrętu w lewo lub w prawo.

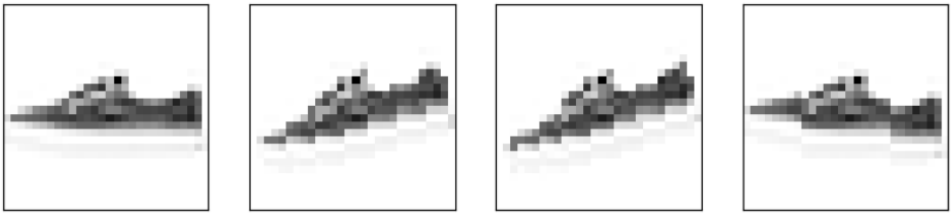
Obracanie obrazów

Obrazy można uzupełniać, nie tylko odwracając je w poziomie i pionie, ale również obracając o niewielki kąt. Wypróbuj to na przykładzie losowego obracania o losowy kąt przy użyciu modułu `transforms`:

```
>>> torch.manual_seed(42)
>>> rotate_transform =
...           transforms.Compose([transforms. RandomRotation(20)])
>>> plt.figure(figsize=(10, 10))
>>> plt.subplot(1, 4, 1)
>>> display_image_greys(image)
```

```
>>> for i in range(3):
...     plt.subplot(1, 4, i+2)
...     img_flip = rotate_transform(image)
...     display_image_greys(img_flip)
```

Rysunek 11.12 przedstawia uzyskany wynik.



Rysunek 11.12. Uzupełnianie obrazów poprzez obracanie

W tym przykładzie oryginalny obraz był obracany o kąty z przedziału od –20° do 20° (odpowiednio, przeciwnie i zgodnie z ruchem wskazówek zegara).

Przycinanie obrazów

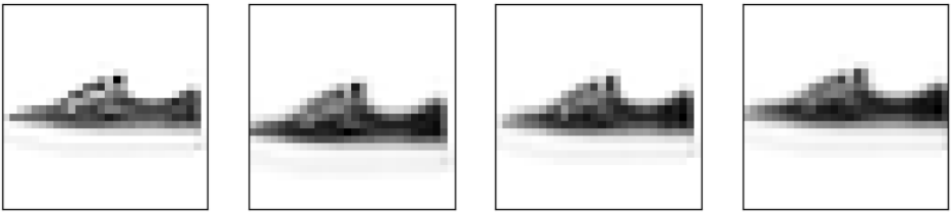
Przycinanie obrazów jest kolejną często stosowaną techniką uzupełniania danych. Pozwala ona generować nowe obrazy poprzez wybieranie fragmentów oryginalnego obrazu. Zazwyczaj proces ten jest uzupełniany o przeskalowanie przyciętego obrazka do z góry określonych wymiarów wyjściowych w celu zagwarantowania jednolitych wymiarów obrazów w całym zbiorze danych.

Zobaczmy teraz, w jaki sposób można zastosować tę technikę, korzystając z klasy `transforms.RandomResizedCrop`, aby losowo wybierać proporcje wycinanego fragmentu obrazka, a następnie przeskalować go ponownie do wymiarów oryginału:

```
>>> torch.manual_seed(42)
>>> crop_transform = transforms.Compose([
...     transforms.RandomResizedCrop(size=(28, 28), scale=(0.7, 1))])
>>> plt.figure(figsize=(10, 10))
>>> plt.subplot(1, 4, 1)
>>> display_image_greys(image)
>>> for i in range(3):
...     plt.subplot(1, 4, i+2)
...     img_crop = crop_transform(image)
...     display_image_greys(img_crop)
```

W tym przypadku parametr `size` określa wielkość wynikowego obrazka po przycięciu i przeskalowaniu, a parametr `scale` definiuje zakres skalowania stosowanego do przycinania. W razie użycia wartości (`min_scale, max_scale`) wielkość obszaru przycinania będzie wybierana losowo i będzie mieścić się w zakresie od `min_scale` do `max_scale` wymiarów początkowego obrazka.

Wyniki generowane przez powyższy kod zostały przedstawione na rysunku 11.13.



Rysunek 11.13. Obrazki przycięte na potrzeby uzupełniania danych

Jak widać w tym przykładzie, parametr `scale=(0.7, 1.0)` oznacza, że wielkość obszaru przycinania będzie się wahać w zakresie od 70% do 100% wielkości obrazka wyjściowego.

Usprawnianie klasyfikatora obrazów poprzez uzupełnianie danych

Wykorzystaj poznane techniki uzupełniania obrazów do wytrenowania klasyfikatora na małym zbiorze. W tym celu wykonaj poniższe kroki:

1. Zacznij od przygotowania funkcji przekształcającej, która wykorzysta wszystkie techniki uzupełniania danych opisane we wcześniejszych punktach rozdziału:

```
>>> torch.manual_seed(42)
>>> transform_train = transforms.Compose([
...     transforms.RandomHorizontalFlip(),
...     transforms.RandomRotation(10),
...     transforms.RandomResizedCrop(size=(28, 28), scale=(0.9, 1)),
...     transforms.ToTensor(),
... ])
```

   Zastosowaliśmy tu odwracanie w poziomie, obrót o kąt 10° oraz przycinanie w zakresie od 90% do 100% wielkości wyjściowego obrazka.

2. Ponownie wczytaj zbiór treningowy, używając powyższej funkcji przekształcającej, przy czym do uczenia użyj jedynie 500 próbek:

```
>>> train_dataset_aug =
...             torchvision.datasets.FashionMNIST(root=image_path,
...                                               train=True,
...                                               transform=transform_train,
...                                               download=False)
>>> from torch.utils.data import Subset
>>> train_dataset_aug_small = Subset(train_dataset_aug,
...                                  torch.arange(500))
```

   Teraz przekonajmy się jak uzupełnianie danych poprawi zdolność uogólniania oraz wydajność modelu, nawet jeśli dysponujemy jedynie bardzo małym zbiorem danych.

3. Załaduj mały, ale uzupełniony zbiór danych w partiach po 64 próbki, tak jak zrobiłeś to wcześniej:

```
>>> train_dl_aug_small = DataLoader(train_dataset_aug_small,
...                                 batch_size,
...                                 shuffle=True)
```

Zwróć uwagę, że w przypadku zastosowania tego sposobu ładowania danych dla jednego obrazka do trenowania modelu użytych zostanie kilka obrazków uzupełnionych, które mogą powstać poprzez odwrócenie, obrót lub przycięcie.

4. Zainicjuj model konwolucyjny, używając do tego celu takiej samej architektury i optymalizatora jak wcześniej:

```
>>> model = nn.Sequential()
>>> # tu pomijamy powtarzający się kod
>>> model = model.to(device)
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

5. Wytrenuj model na uzupełnionych danych w 1000 iteracji:

```
>>> train(model, optimizer, 1000, train_dl_aug_small)
Epoka 1 - strata: 2.3013 - dokładność: 0.1400
...
Epoka 301 - strata: 1.6817 - dokładność: 0.7760
...
Epoka 601 - strata: 1.5006 - dokładność: 0.9620
...
Epoka 1000 - strata: 1.4904 - dokładność: 0.9720
```

6. Sprawdź działanie modelu na zbiorze testowym:

```
>>> evaluate_model(model, test_dl)
Accuracy on test set: 79.24%
```

Model wytrenowany na uzupełnionych danych uzyskał na zbiorze testowym dokładność 79,24%. Zwróć uwagę, że ten wynik może się zmieniać.

Przeprowadziliśmy również eksperymenty z trenowaniem modelu bez uzupełniania danych, co zapewniło dokładność na zbiorze testowym na poziomie około 76%. Po zastosowaniu uzupełniania dokładność wzrosła do 79%. Jak zawsze możesz swobodnie dostrajać hiperparametry, tak jak robiliśmy to w rozdziale 6., „Prognozowanie cen akcji za pomocą sieci neuronowych", i sprawdzić, czy uzyskasz jeszcze większą dokładność klasyfikacji.

Kolejnym sposobem poprawiania wydajności klasyfikatorów jest technika określana jako uczenie transferowe. Przyjrzymy się jej bliżej w następnym podrozdziale.

Rozwijanie klasyfikatora z wykorzystaniem uczenia transferowego

Uczenie transferowe (ang. *transfer learning*) to technika uczenia maszynowego, w której model wytrenowany na jednym zadaniu jest adaptowany lub dostrajany do drugiego, powiązanego zadania. W uczeniu transferowym wiedza zdobyta podczas trenowania pierwszego zadania (tak zwanego zadania źródłowego) jest wykorzystywana do poprawy procesu uczenia drugiego zadania (zadania docelowego). Rozwiązanie to jest szczególnie przydatne, gdy dysponujemy ograniczoną ilością danych dla zadania docelowego, ponieważ umożliwia przeniesienie wiedzy z większego lub bardziej zróżnicowanego zbioru danych.

Typowy przebieg pracy z uczeniem transferowym obejmuje:

1. **Model wstępnie wytrenowany**: Rozpoczynamy od modelu wstępnie wytrenowanego, który został już nauczony na dużym i odpowiednim zbiorze danych dla innego, lecz powiązanego zadania. Taki model często jest głęboką siecią neuronową, na przykład siecią konwolucyjną wykorzystywaną do zadań związanych z obrazami.
2. **Ekstrakcja cech**: Wykorzystujemy model wstępnie wytrenowany jako ekstraktor cech. W takim przypadku należy z niego usunąć ostatnie warstwy klasyfikacyjne (jeśli takie są) i wykorzystać wyjście jednej z warstw pośrednich jako reprezentację cech danych. Takie cechy potrafią uchwycić wzorce wysokiego poziomu oraz informacje z zadania źródłowego.
3. **Dostrajanie**: Do ekstraktora cech dodajemy nowe warstwy. Te nowe warstwy są specyficzne dla zadania docelowego i zazwyczaj są inicjalizowane losowo. Następnie trenujemy cały model, łącznie z ekstraktorem cech i nowymi warstwami, na docelowym zbiorze danych. Dostrajanie umożliwia modelowi dostosowanie się do specyfiki zadania docelowego.

Zanim przejdziemy do implementacji uczenia transferowego w naszym zadaniu klasyfikacji obrazów odzieży, przyjrzyjmy się ewolucji architektur sieci konwolucyjnych oraz modeli wstępnie wytrenowanych. Nawet najwcześniejsze architektury sieci konwolucyjnych wciąż są aktywnie używane! Kluczową kwestią jest to, że wszystkie te architektury cały czas stanowią wartościowe narzędzia we współczesnym zestawie narzędzi do pracy z uczeniem głębokim, zwłaszcza gdy są stosowane w zadaniach uczenia transferowego.

Rozwój architektur sieci konwolucyjnych oraz wstępnie wytrenowanych modeli

Koncepcja wykorzystania sieci konwolucyjnych do przetwarzania obrazów sięga lat 90. XX wieku. Wczesne architektury, takie jak **LeNet-5** (z 1998 r.), pokazały potencjał głębokich sieci neuronowych w zadaniach klasyfikacji obrazów. LeNet-5 składa się z dwóch zestawów warstw konwolucyjnych, po których umieszczone są dwie warstwy w pełni połączone oraz jedna warstwa wyjściowa. Każda warstwa konwolucyjna wykorzystuje jądra o rozmiarze 5×5. Architektura LeNet-5 odegrała istotną rolę w udowodnieniu skuteczności uczenia głębokiego w zadaniach związanych z klasyfikacją obrazów. Model ten osiągnął wysoką dokładność na zbiorze danych MNIST, który jest powszechnie używanym zestawem testowym do rozpoznawania ręcznie pisanych cyfr.

Osiągnięcia architektury LeNet-5 utorowały drogę do powstania bardziej złożonych architektur, takich jak **AlexNet** (z 2012 r.). Architektura ta składa się z ośmiu warstw — pięciu zestawów warstw konwolucyjnych, po których następują trzy warstwy w pełni połączone. W architekturze AlexNet po raz pierwszy w głębokiej sieci konwolucyjnej zastosowano funkcję aktywacji ReLU oraz technikę dropout w warstwach w pełni połączonych w celu zapobiegania przeuczeniu. Do poprawy zdolności uogólniania modelu wykorzystano techniki uzupełniania danych, takie jak losowe przycinanie oraz odwracanie w poziomie. Sukces architektury AlexNet przyczynił się do ponownego

wzrostu zainteresowania sieciami neuronowymi i zapoczątkował rozwój jeszcze głębszych i bardziej złożonych architektur, takich jak VGGNet, GoogLeNet i ResNet, które stały się podstawą współczesnej wizji komputerowej.

Architektura **VGGNet** została zaproponowana przez Visual Geometry Group z Uniwersytetu Oksfordzkiego w 2014 r. Charakteryzuje się prostotą i jednolitością. Składa się z szeregu warstw konwolucyjnych, po których następują warstwy maksymalizujące, a na końcu znajduje się stos warstw w pełni połączonych. W tej architekturze wykorzystywane są głównie filtry konwolucyjne o rozmiarze 3×3, co pozwala sieci na wychwytywanie szczegółowych informacji przestrzennych. Najczęściej wykorzystywanymi wersjami tej architektury są VGG16 i VGG19, zawierające, odpowiednio, 16 i 19 warstw. Modele o tej architekturze są często używane jako punkt wyjścia do uczenia transferowego w różnych zadaniach z zakresu wizji komputerowej.

W tym samym roku firma Google opracowała architekturę **GoogLeNet**, znaną również jako **Inception**. Jej cechą charakterystyczną jest moduł *incepcji* (ang. *inception*). Zamiast stosowania pojedynczej warstwy konwolucyjnej o stałym rozmiarze filtra moduły incepcji wykorzystują równoległe filtry o różnych rozmiarach (1×1, 3×3, 5×5). Te równoległe operacje pozwalają wychwytywać cechy w różnych skalach i zapewniają bogatszą reprezentację danych. Podobnie jak VGGNet, wstępnie wytrenowane modele GoogLeNet występują w różnych wariantach, takich jak InceptionV1, InceptionV2, InceptionV3 i InceptionV4, z których każdy charakteryzuje się pewnymi modyfikacjami i ulepszeniami architektury.

Architektura **ResNet**, czyli **sieć rezydualna** (ang. *residual network*), została wprowadzona przez Kaiminga He i współpracowników w 2015 r. w celu rozwiązania problemu zanikających gradientów – zjawiska, w którym gradienty funkcji straty w konwolucyjnych sieciach neuronowych stają się bardzo małe. Kluczową innowacją tej architektury jest zastosowanie połączeń rezydualnych. Bloki rezydualne pozwalają sieci pomijać wybrane warstwy podczas procesu uczenia. Zamiast bezpośrednio uczyć się odwzorowania wejścia na wyjście, bloki rezydualne uczą się tzw. odwzorowania rezydualnego, które następnie dodawane jest do początkowego wejścia. Dzięki temu możliwe stało się budowanie głębszych sieci. Wstępnie wytrenowane modele ResNet dostępne są w różnych wersjach, takich jak ResNet-18, ResNet-34, ResNet-50, ResNet-101 czy niezwykle głęboki ResNet-152. Liczby te oznaczają głębokość sieci, czyli liczbę warstw.

Rozwój architektur konwolucyjnych sieci neuronowych oraz wstępnie wytrenowanych modeli trwa, czego przykładem są innowacje takie jak EfficientNet, MobileNet czy architektury projektowane do specyficznych zastosowań. Na przykład modele **MobileNet** zostały zaprojektowane z myślą o wysokiej wydajności pod względem zużycia zasobów obliczeniowych i pamięci. Są one przeznaczone do wdrożeń na urządzeniach o ograniczonych możliwościach sprzętowych, takich jak smartfony, urządzenia IoT oraz urządzenia brzegowe.

Uwaga

Lista wszystkich wstępnie wytrenowanych modeli dostępnych w bibliotece PyTorch znajduje się na stronie*https://pytorch.org/vision/stable/models.html#classification*.

Ewolucja architektur sieci konwolucyjnych oraz dostępność wstępnie wytrenowanych modeli zrewolucjonizowały zadania związane z komputerowym rozpoznawaniem obrazów. Znacząco poprawiły one wyniki osiągane w klasyfikacji obrazów, detekcji obiektów, segmentacji oraz wielu innych zastosowaniach.

Przyjrzyjmy się teraz, jak wykorzystać model wstępnie wytrenowany, aby ulepszyć nasz klasyfikator obrazów odzieży.

Poprawa klasyfikatora obrazów odzieży poprzez dostrajanie sieci ResNet

W tym przykładzie do przedstawienia wykorzystania uczenia transferowego użyjemy modelu o architekturze ResNet, a konkretnie modelu ResNet-18:

1. Zacznij od zaimportowania wstępnie wytrenowanego modelu ResNet-18 z pakietu `torchvision`:

   ```
   >>> from torchvision.models import resnet18
   >>> my_resnet = resnet18(weights='IMAGENET1K_V1')
   ```

   Wartość `IMAGENET1K` odwołuje się do wstępnie wytrenowanego modelu, który był trenowany na zbiorze danych ImageNet-1K (*https://www.image-net.org/download.php*), natomiast `V1` określa pierwszą wersję tego wstępnie wytrenowanego modelu.

   W ten sposób dysponujesz już wstępnie wytrenowanym modelem.

2. Zbiór danych ImageNet-1K zawiera obrazy RGB, dlatego też pierwsza warstwa konwolucyjna pierwotnego modelu ResNet została zaprojektowana pod kątem trójwymiarowych danych wejściowych. Jednak używany we wcześniejszych przykładach zbiór danych FashionMNIST zawiera obrazy w skali szarości, zatem musisz zmodyfikować model tak, by akceptował dane jednowymiarowe:

   ```
   >>> my_resnet.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2,
   ...                             padding=3, bias=False)
   ```

   W tym przykładzie w wywołaniu tworzącym pierwszą warstwę konwolucyjną zmieniliśmy pierwszy argument, wymiar danych wejściowych, z 3 na 1. Jej pierwotna definicja ma następującą postać:

   ```
   self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
   bias=False)
   ```

3. Zmień warstwę wynikową, tak by zwracała 100 z 1000 klas:

   ```
   >>> num_ftrs = my_resnet.fc.in_features
   >>> my_resnet.fc = nn.Linear(num_ftrs, 10)
   ```

   Zmieniliśmy jedynie wielkość wyników warstwy wyjściowej.

   Kroki 2. i 3. stanowią przygotowanie do procesu **dostrajania** modelu.

4. Dostrój zaadaptowany wstępnie wytrenowany model na pełnym treningowym zbiorze danych:

   ```
   >>> my_resnet = my_resnet.to(device)
   >>> optimizer = torch.optim.Adam(my_resnet.parameters(), lr=0.001)
   >>> train(my_resnet, optimizer, 10, train_dl)
   ```

```
Epoka 1 - strata: 0.4845 - dokładność: 0.8257
Epoka 2 - strata: 0.3305 - dokładność: 0.8795
Epoka 3 - strata: 0.2865 - dokładność: 0.8968
Epoka 4 - strata: 0.2679 - dokładność: 0.9037
Epoka 5 - strata: 0.2361 - dokładność: 0.9140
Epoka 6 - strata: 0.2183 - dokładność: 0.9198
Epoka 7 - strata: 0.1916 - dokładność: 0.9301
Epoka 8 - strata: 0.1884 - dokładność: 0.9306
Epoka 9 - strata: 0.1736 - dokładność: 0.9363
Epoka 10 - strata: 0.1426 - dokładność: 0.9476
```

Jak widać, w przypadku użycia dostrojonego modelu ResNet po 10 epokach uzyskaliśmy dokładność wynoszącą 94%.

5. Sprawdź wyniki uzyskiwane na zbiorze testowym:

```
>>> evaluate_model(my_resnet, test_dl)
Dokładność na zbiorze testowym: 91.27 %
```

Dokładność na zbiorze testowym udało się nam poprawić z 90% na 91% i to przy zastosowaniu jedynie 10 iteracji treningu.

Uczenie transferowe z wykorzystaniem konwolucyjnych sieci neuronowych jest potężną techniką pozwalającą na korzystanie ze wstępnie wytrenowanych modeli oraz adaptowanie ich do specyficznych zadań związanych z klasyfikowaniem obrazów.

Podsumowanie

W tym rozdziale została opisana klasyfikacja zdjęć odzieży, oparta na konwolucyjnej sieci neuronowej. Najpierw szczegółowo zapoznałeś się z elementami sieci i jej działaniem inspirowanym komórkami układu wzrokowego człowieka. Następnie zbudowałeś sieć konwolucyjną klasyfikującą zdjęcia odzieży marki Zalando, zawarte w zbiorze Fashion-MNIST. Dowiedziałeś się także, czym jest uzupełnianie danych, i poznałeś kilka popularnych metod uzupełniania obrazów. I w końcu, po opisaniu ewolucji konwolucyjnych sieci neuronowych, poznałeś przykład uczenia transferowego z wykorzystaniem modelu o architekturze ResNet.

W następnym rozdziale poznasz inny typ sieci uczenia głębokiego: rekurencyjną sieć neuronową. Sieci konwolucyjna i rekurencyjna to dwie najpotężniejsze odmiany sieci neuronowych, dzięki którym uczenie głębokie jest dzisiaj tak popularne.

Ćwiczenia

1. Jak wspomniałem wcześniej, poeksperymentuj z parametrami konwolucyjnej sieci neuronowej i sprawdź, czy uzyskasz jeszcze lepszy wynik.
2. Spróbuj zastosować techniki dropout w celu poprawy wyników modelu konwolucyjnej sieci neuronowej.
3. Spróbuj poeksperymentować z wykorzystaniem wstępnie wytrenowanego modelu Vision Transformer: *https://huggingface.co/google/vit-base-patch16-224.*

Skorowidz

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Z

Najlepsze praktyki uczenia maszynowego? Tylko z Pythonem!

Python wraz ze swoimi bibliotekami umożliwia tworzenie coraz bardziej wyrafinowanych implementacji algorytmów uczenia maszynowego. Dzięki temu systemy przetwarzania języka naturalnego i obrazów wkraczają do naszego życia na szeroką skalę. Aby jednak uzyskiwać najlepsze wyniki w tej dziedzinie, potrzebna jest znajomość dobrych praktyk.

Oto czwarte wydanie popularnego podręcznika, z którym nauczysz się stosować zaawansowane techniki uczenia maszynowego. Zawiera dwa nowe rozdziały poświęcone architekturze Transformer oraz modelom takim jak BERT i GPT, jak również multimodalnym modelom komputerowego rozpoznawania obrazów implementowanym z wykorzystaniem PyTorch i Hugging Face. Znajdziesz tu solidną dawkę teorii połączonej z przykładami jej praktycznego zastosowania. Dzięki lekturze poszerzysz wiedzę z zakresu uczenia głębokiego, odkryjesz pełny potencjał zaawansowanych technik uczenia maszynowego i łatwiej sprostasz codziennym wyzwaniom.

W książce między innymi:

- najlepsze praktyki uczenia maszynowego
- budowa i ulepszanie klasyfikatorów obrazów
- tworzenie i strojenie sieci neuronowych z wykorzystaniem TensorFlow i PyTorch
- rekurencyjne sieci neuronowe, transformery i model CLIP
- maszyna wektorów nośnych i poprawa ich wydajności
- regularyzacja, wybór cech i wiele innych przydatnych technik

Yuxi (Hayden) Liu jest autorem cenionych książek poświęconych uczeniu maszynowemu. Wcześniej pracował w Google i stosował techniki uczenia maszynowego w takich dziedzinach jak analiza danych i cyberbezpieczeństwo. Entuzjasta edukacji.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-3171-8
Cena: 129,00 zł
packt