

Sebastian Raschka



STWÓRZ WŁASNE **AI**

Jak od podstaw zbudować
duży model językowy

Helion 

Tytuł oryginału: Build a Large Language Model (From Scratch)

Tłumaczenie: Radosław Meryk

ISBN: 978-83-289-2498-7

© Helion S.A. 2025.

Authorized translation of the English edition © 2025 Manning Publications.
This translation is published and sold by permission of Manning Publications,
the owner of all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any
form or by any means, electronic or mechanical, including photocopying, recording
or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości
lub fragmentu niniejszej publikacji w jakiekolwiek postaci jest zabronione.
Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie
książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie
praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi
bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje
były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich
wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych
lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności
za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/stwlai_ebook

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)

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

<i>Przedmowa</i>	7
<i>Podziękowania</i>	8
<i>O tej książce</i>	9
<i>O autorze</i>	13

1 *Czym są duże modele językowe?* 15

1.1.	Czym jest model LLM?	16
1.2.	Zastosowania modeli LLM	18
1.3.	Etapy tworzenia modeli LLM i korzystania z nich	20
1.4.	Wprowadzenie do architektury transformerów	22
1.5.	Wykorzystanie dużych zbiorów danych	24
1.6.	Szczegóły architektury modeli GPT	27
1.7.	Tworzenie dużego modelu językowego	29
	Podsumowanie	30

2 *Praca z danymi tekstowymi* 32

2.1.	Czym są osadzenia słów?	33
2.2.	Tokenizacja tekstu	36
2.3.	Konwersja tokenów na identyfikatory	39
2.4.	Dodawanie specjalnych tokenów kontekstowych	43
2.5.	Kodowanie par bajtów	47
2.6.	Próbkowanie danych z oknem przesuwnym	50
2.7.	Tworzenie osadzeń tokenów	56
2.8.	Kodowanie pozycji słów	58
	Podsumowanie	62

3

Kodowanie mechanizmów uwagi 64

- 3.1. Problem z modelowaniem długich sekwencji 65
 - 3.2. Przechwytywanie zależności między danymi za pomocą mechanizmów uwagi 67
 - 3.3. Zwracanie uwagi na różne części danych wejściowych przez mechanizm samouwagi 69
 - 3.3.1. *Prosty mechanizm samouwagi bez trenowań wag* 70
 - 3.3.2. *Obliczanie wag uwagi dla wszystkich tokenów wejściowych* 75
 - 3.4. Implementacja mechanizmu samouwagi z trenowańmi wagami 77
 - 3.4.1. *Obliczanie wag uwagi krok po kroku* 78
 - 3.4.2. *Implementacja kompaktowej klasy samouwagi w Pythonie* 82
 - 3.5. Ukrywanie przyszłych słów dzięki zastosowaniu uwagi przyczynowej 87
 - 3.5.1. *Wykorzystanie maski uwagi przyczynowej* 88
 - 3.5.2. *Maskowanie dodatkowych wag uwagi z użyciem dropoutu* 91
 - 3.5.3. *Implementacja zwięzzej klasy przyczynowej uwagi* 93
 - 3.6. Rozszerzenie uwagi jednogłowicowej na wielogłowicową 95
 - 3.6.1. *Utworzenie stosu wielu jednogłowicowych warstw uwagi* 95
 - 3.6.2. *Implementacja uwagi wielogłowicowej z podziałem wag* 98
- Podsumowanie 103

4

Implementacja od podstaw modelu GPT do generowania tekstu 105

- 4.1. Kodowanie architektury LLM 106
- 4.2. Normalizacja warstwowa aktywacji 112
- 4.3. Implementacja sieci ze sprzężeniem w przód z aktywacjami GELU 118
- 4.4. Dodawanie połączeń skrótowych 122
- 4.5. Łączanie warstw uwagi i warstw liniowych w bloku transformera 126

4.6.	Kodowanie modelu GPT	129
4.7.	Generowanie tekstu	134
	Podsumowanie	139

5

Wstępne szkolenie na nieoznakowanych danych 140

5.1.	Ocena generatywnych modeli tekstowych	141
5.1.1.	<i>Używanie modelu GPT do generowania tekstu</i>	142
5.1.2.	<i>Obliczanie strat związanych z generowaniem tekstu</i>	144
5.1.3.	<i>Obliczanie strat w zestawie szkoleniowym i walidacyjnym</i>	152
5.2.	Szkolenie modelu LLM	157
5.3.	Strategie dekodowania w celu zarządzania losowością	163
5.3.1.	<i>Skalowanie temperatury</i>	164
5.3.2.	<i>Próbkowanie top-k</i>	167
5.3.3.	<i>Modyfikacja funkcji generowania tekstu</i>	169
5.4.	Wczytywanie i zapisywanie wag modeli z użyciem frameworka PyTorch	171
5.5.	Ładowanie wstępnie przeszkolonych wag z modelu OpenAI	173
	Podsumowanie	179

6

Dostrajanie modelu LLM do zadań klasyfikacji 181

6.1.	Różne kategorie dostrajania	182
6.2.	Przygotowanie zbioru danych	183
6.3.	Tworzenie mechanizmów ładowających dane	188
6.4.	Inicjalizacja modelu z użyciem wag wstępnie przeszkolonego modelu	193
6.5.	Dodawanie nagłówka klasyfikacji	195
6.6.	Obliczanie straty i dokładności klasyfikacji	202
6.7.	Dostrajanie modelu na danych nadzorowanych	206
6.8.	Wykorzystanie modelu LLM jako klasyfikatora spamu	212
	Podsumowanie	214

7

Dostrajanie modelu LLM do zadań wykonywania instrukcji 215

- 7.1. Wprowadzenie do dostrajania do wykonywania instrukcji 216
- 7.2. Przygotowanie zbioru danych do nadzorowanego dostrajania pod kątem wykonywania instrukcji 218
- 7.3. Organizowanie danych w partie szkoleniowe 222
- 7.4. Tworzenie mechanizmów ładujących dane dla zbioru danych instrukcji 234
- 7.5. Ładowanie wstępnie przeszkolonego modelu LLM 237
- 7.6. Dostrajanie modeli LLM do zadań wykonywania instrukcji 241
- 7.7. Wyodrębnianie i zapisywanie odpowiedzi 245
- 7.8. Ocena dostrojonego modelu LLM 250
- 7.9. Wnioski 260
 - 7.9.1. *Co dalej?* 260
 - 7.9.2. *Bądź na bieżąco w szybko zmieniającej się dziedzinie* 261
 - 7.9.3. *Na koniec* 261
- Podsumowanie 261

Dodatek A Wprowadzenie w tematykę frameworka PyTorch 263

Dodatek B Bibliografia i lektura uzupełniająca 303

Dodatek C Rozwiązańa ćwiczeń 315

Dodatek D Usprawnianie pętli szkoleniowej 328

Dodatek E Skuteczne dostrajanie parametrów za pomocą LoRA 337

Przedmowa

Modele językowe zawsze mnie fascynowały. Ponad 10 lat temu rozpoczęłem własną podróż w kierunku sztucznej inteligencji od zajęć z klasyfikacji wzorców statystycznych. Wtedy to stworzyłem swój pierwszy niezależny projekt: opracowałem model i aplikację webową do wykrywania tonu piosenki na podstawie jej tekstu.

Po kilku latach, w 2022 roku, wraz z udostępnieniem ChatGPT, duże modele językowe (LLM) szturmem zdobyły świat i zrewolucjonizowały sposób pracy wielu z nas. Modele te są niezwykle wszechstronne. Pomagają ludziom w takich zadaniach jak sprawdzanie gramatyki, komponowanie wiadomości e-mail, streszczanie długich dokumentów i wielu innych. Wynika to ze zdolności modeli do parsowania i generowania tekstu podobnego do ludzkiego, co jest ważne w różnych dziedzinach, od obsługi klienta po tworzenie treści, a nawet w bardziej technicznych, takich jak kodowanie i analiza danych.

Jak wskazuje nazwa, cechą charakterystyczną LLM jest to, że są „duże” – bardzo duże – ponieważ obejmują od kilku milionów do kilku miliardów parametrów (dla porównania, w bardziej tradycyjnych metodach uczenia maszynowego lub metodach statystycznych stosuje się zbiór danych kwiatów Iris, który pozwala klasyfikować kwiaty z dokładnością ponad 90% z użyciem małego modelu zaledwie dwoma parametrami). Jednak pomimo dużego rozmiaru modeli LLM w porównaniu z bardziej tradycyjnymi metodami, nie muszą być one czarną skrzynką.

Z tej książki dowiesz się, jak krok po kroku zbudować model LLM. Po zakończeniu lektury będziesz wiedział, jak pod maską działa model LLM podobny do tego, którego użyto w systemie ChatGPT. Wierzę, że kluczowe znaczenie na drodze do sukcesu ma dokładne zrozumienie każdej części podstawowych pojęć i bazowego kodu. Nie tylko pomaga to w naprawianiu błędów i poprawianiu wydajności, ale także umożliwia eksperymentowanie z nowymi pomysłami.

Kilka lat temu, kiedy zacząłem pracować z modelami LLM, sam musiałem uczyć się ich implementowania. Aby zdobyć potrzebną wiedzę, musiałem przeszukiwać dziesiątki artykułów naukowych i niekompletnych repozytoriów kodu. Dzięki tej książce, w której udostępniam samouczek implementacji modeli LLM szczegółowo, krok po kroku, opisujący wszystkie główne komponenty i fazy rozwoju LLM, mam nadzieję uczynić modele LLM bardziej dostępnymi.

Mocno wierzę, że najlepszym sposobem na zrozumienie modeli LLM jest zakodowanie ich od podstaw. Przekonasz się przy tym, że to też może być świetna zabawa!

Milego czytania i kodowania!

Podziękowania

Napisanie książki to poważne przedsięwzięcie. Chciałbym wyrazić szczerą wdzięczność mojej żonie Lizie za jej cierpliwość i wsparcie podczas całego procesu. Jej bezwzględna miłość i ciągłe wsparcie były absolutnie niezbędne.

Jestem niezmiernie wdzięczny Danielowi Kleine'owi, którego nieocenione opinie na temat powstających rozdziałów i kodu wykraczaly poza wszelkie granice. Dzięki jego wnikliwemu spojrzeniu na szczegóły i cennym sugestiom niewątpliwie czyta się tę książkę płynniej i przyjemniej.

Chciałbym również podziękować wspaniałym pracownikom Manning Publications, w tym Michaelowi Stephensowi, za wiele owocnych dyskusji, które pomogły ukształtować tę książkę, oraz Dustinowi Archibaldowi, którego konstruktywne opinie i wskazówki dotyczące przestrzegania wytycznych Manninga były kluczowe w jej powstaniu. Doceniam także Waszą elastyczność w dostosowaniu się do unikatowych wymagań tego niekonwencjonalnego podejścia budowy modelu LLM od podstaw. Specjalne podziękowania kieruję do Aleksandara Dragosavljevića, Kari Lucke i Mike'a Beady'ego za ich pracę nad profesjonalnym układem oraz dla Susan Honeywell i jej zespołu za dopracowanie grafiki.

Pragnę wyrazić szczerą wdzięczność Robin Campbell i jej wybitnemu zespołowi marketingowemu za ich nieocenione wsparcie podczas całego procesu pisania.

Na koniec chciałbym podziękować recenzentom. Na podziękowanie zasługują następujące osoby: Anandaganesh Balakrishnan, Anto Aravindh, Ayush Bihani, Bassam Ismail, Benjamin Muskalla, Bruno Sonnino, Christian Prokopp, Daniel Kleine, David Curran, Dibyendu Roy Chowdhury, Gary Pass, Georg Sommer, Giovanni Alzetta, Guillermo Alcántara, Jonathan Reeves, Kunal Ghosh, Nicolas Modrzyk, Paul Silisteau, Raul Ciotescu, Scott Ling, Sriram Macharli, Sumit Pal, Vahidovi Mirjalili, Vaijanath Rao i Walter Reade. Wasze cenne spostrzeżenia i komentarze miały nieoceniony wpływ na jakość tej książki.

Dziękuję wszystkim, którzy towarzyszyli mi w tej podróży. Jestem Wam szczerze wdzięczny. Wasze wsparcie, wiedza i poświęcenie odegrały kluczową rolę w powstaniu tej książki. Jeszcze raz dziękuję!

O tej książce

Książka *Stwórz własne AI* pomoże Ci zrozumieć i tworzyć od podstaw własne duże modele językowe (LLM) podobne do GPT. Rozpoczyna się od omówienia podstawowych zasad pracy z danymi tekstowymi i kodowania mechanizmów uwagi, a następnie prowadzi krok po kroku przez implementację kompletnego modelu GPT. Następnie omawia mechanizm wstępnego szkolenia, a także dostrajanie do konkretnych zadań, takich jak klasyfikacja tekstu i wykonywanie instrukcji. Po zakończeniu lektury tej książki będziesz dogłębnie rozumiał działanie modeli LLM i umiał tworzyć własne. Chociaż Twoje modele będą działać w mniejszej skali niż duże modele podstawowe, będą się opierać na tych samych pojęciach i zapewniać bogate możliwości nauki, pozwalające zrozumieć podstawowe mechanizmy i techniki stosowane podczas budowania najnowocześniejszych modeli LLM.

Kto powinien przeczytać tę książkę?

Książka *Stwórz własne AI* jest przeznaczona dla entuzjastów uczenia maszynowego, inżynierów, badaczy, studentów i wszystkich tych, którzy chcą dogłębnie zrozumieć, jak działa LLM, i nauczyć się budować własne modele. Zarówno początkujący, jak i doświadczeni programiści będą mogli wykorzystać swoje dotychczasowe umiejętności i wiedzę do zrozumienia pojęć i technik stosowanych w tworzeniu LLM.

To, co wyróżnia tę książkę, to kompleksowe omówienie całego procesu tworzenia LLM, od pracy z zestawami danych po implementację architektury modelu, wstępne szkolenie na nieoznakowanych danych i dostrajanie do określonych zadań. W chwili gdy piszę te słowa, żadna inna pozycja nie zapewnia tak kompletnego i praktycznego podejścia do budowania LLM od podstaw.

Aby zrozumieć przykłady kodu w tej książce, powinieneś mieć solidną wiedzę na temat programowania w Pythonie. Chociaż pewna znajomość uczenia maszynowego, uczenia głębokiego i sztucznej inteligencji może się przydać, rozległe doświadczenie w tych obszarach nie jest wymagane. Modele LLM to unikatowy podzbior AI, więc nawet jeśli jesteś początkujący w tej dziedzinie, będziesz w stanie uczyć się z materiału zawartego w książce.

Jeśli masz jakieś doświadczenia z głębokimi sieciami neuronowymi, pewne pojęcia mogą wydać Ci się znajome, ponieważ modele LLM są oparte na tych architekturach. Dogłębna znajomość framework'a PyTorch nie jest jednak warunkiem koniecznym.

Dodatek A zawiera zwięzłe wprowadzenie do tego framework'a. Jego lektura dostarczy Ci wiedzy niezbędnej do zrozumienia przykładów kodu w całej książce.

Podczas odkrywania wewnętrznego działania modeli LLM przyda Ci się wiedza z zakresu matematyki na poziomie szkoły średniej, w szczególności o działaniach na wektorach i macierzach. Do zrozumienia kluczowych pojęć przedstawionych w tej książce nie jest jednak konieczna zaawansowana wiedza matematyczna.

Najważniejszym warunkiem wstępnyim są solidne podstawy programowania w Pythonie. Dzięki tej wiedzy będziesz dobrze przygotowany do odkrywania fascynującego świata LLM, zrozumienia zaprezentowanych w książce pojęć i przedstawionych w niej przykładów kodu.

Jak zorganizowana jest ta książka? Mapa drogowa

Ta książka jest przeznaczona do czytania rozdział po rozdziale, ponieważ każdy z nich opiera się na pojęciach i technikach wprowadzonych we wcześniejszych rozdziałach. Książka jest podzielona na siedem rozdziałów, które obejmują istotne aspekty modeli LLM i ich implementacji.

Rozdział 1. zawiera ogólne wprowadzenie do podstawowych pojęć dotyczących modeli LLM. Opisuje architekturę transformera, będącą podstawą dla modeli LLM podobnych do tych, których użyto na platformie ChatGPT.

Rozdział 2. przedstawia plan budowania modelu LLM od podstaw. Obejmuje on proces przygotowywania tekstu do szkolenia modelu LLM, w tym podział tekstu na tokeny wyrazów i podwyrazów, używanie kodowania par bajtów do zaawansowanej tokenizacji, próbkowanie przykładów szkoleniowych z użyciem okna przesuwnego oraz konwertowanie tokenów na wektory, niezbędne do działania modeli LLM.

Rozdział 3. koncentruje się na mechanizmach uwagi wykorzystywanych w modelach LLM. Wprowadzam w nim podstawowy framework samouwagi, a następnie przechodzę do ulepszzonego mechanizmu samouwagi. Rozdział opisuje również implementację modułu uwagi przyczynowej, umożliwiający modelowi LLM generowanie jednego tokenu na raz, maskowanie losowo wybranych wag uwagi za pomocą techniki dropoutu w celu zmniejszenia nadmiernego dopasowania oraz łączenie wielu modułów uwagi przyczynowej w moduł uwagi wielogłowicowej.

Rozdział 4. skupia się na kodowaniu podobnego do GPT modelu LLM, który można przeszkoić do generowania tekstu podobnego do ludzkiego. Obejmuje on takie techniki jak normalizacja aktywacji warstw w celu stabilizowania szkolenia sieci neuronowych, dodawanie połączeń skrótowych w głębokich sieciach neuronowych w celu wydajniejszego szkolenia modeli, implementację bloków transformera do tworzenia modeli GPT o różnych rozmiarach oraz obliczanie liczby parametrów i szacowanie wymagań pamięci trwałe dla modeli GPT.

Rozdział 5. opisuje implementację procesu wstępnego szkolenia modeli LLM. Opisałem w nim obliczanie strat zbiorów szkoleniowego i walidacyjnego w celu oceny jakości tekstu generowanego przez model LLM, implementację funkcji szkoleniowej, a także wstępne szkolenie modeli LLM, zapisywanie i ładowanie wag modelu w celu kontynuowania szkolenia oraz ładowanie wstępnie wyuczonych wag modeli OpenAI.

W rozdziale 6. przedstawiłem różne podejścia do dostrajania modeli LLM. Opisuję w nim przygotowanie zbioru danych do klasyfikacji tekstu, modyfikację wstępnie przeszkolonego modelu LLM w celu dostrajania, dostrajanie LLM w celu identyfikowania wiadomości spamowych oraz ocenę dokładności dostrojonego klasyfikatora LLM.

Rozdział 7. analizuje proces dostrajania modelu LLM pod kątem wykonywania instrukcji. Obejmuje on przygotowanie zbioru danych do nadzorowanego dostrajania pod kątem wykonania instrukcji, organizowanie danych instrukcji w partie szkoleniowe, ładowanie wstępnie przeszkolonego modelu LLM i dostrajanie go do wykonywania instrukcji użytkownika, wyodrębnianie generowanych przez LLM odpowiedzi na instrukcje w celu ich oceny oraz ocenę modelu LLM dostrojonego pod kątem wykonywania instrukcji.

Informacje na temat kodu

Aby maksymalnie ułatwić śledzenie przykładów kodu omawianych w tej książce, wszystkie udostępniono na stronie internetowej wydawnictwa Manning pod adresem <https://www.manning.com/books/build-a-large-language-model-from-scratch>, a także w formacie notatnika Jupyter w serwisie GitHub, pod adresem <https://github.com/rasbt/LLMs-from-scratch>. Nie martw się, że utkniesz – rozwiązania wszystkich ćwiczeń z kodem można znaleźć w Dodatku C.

Ta książka zawiera wiele przykładów kodu źródłowego, zarówno w postaci numerowanych listingów, jak i wewnątrz tekstu. W obu przypadkach kod źródłowy sformatowano czcionką o stałej szerokości, aby oddzielić go od zwykłego tekstu.

W wielu przypadkach oryginalny kod źródłowy został przeformatowany; dodaliśmy podziały wierszy i zmieniliśmy wcięcia, aby dostosować je do miejsca dostępnego na drukowanej stronie. Sporadycznie nawet to nie wystarczało, dlatego listingi zawierają znaczniki kontynuacji wiersza (\rightarrow). Dodatkowo, tam gdzie kod jest opisywany w tekście, często z kodu źródłowego usunięto komentarze. Wielu listingom towarzyszą adnotacje, które uwypuklają ważne pojęcia.

Jednym z kluczowych celów tej książki jest dostępność. Z tego względu przykłady kodu starannie zaprojektowano w taki sposób, aby działały wydajnie na zwykłym laptopie i nie wymagały od Czytelnika użycia specjalnego sprzętu. Jeśli jednak masz dostęp do układu GPU, w niektórych podrozdziałach znajdziesz pomocne wskazówki dotyczące skalowania zbiorów danych i modeli w celu wykorzystania tej dodatkowej mocy.

W całej książce będziemy używać frameworka PyTorch jako biblioteki działań na tensorach oraz technik uczenia głębokiego potrzebnych do implementowania modeli LLM od podstaw. Jeśli PyTorch jest dla Ciebie nowością, zachęcam Cię do zapoznania się z „Dodatkiem A”, zawierającym obszerne wprowadzenie do zagadnień związanych z tym frameworkiem wraz z zaleceniami dotyczącymi konfiguracji.

Forum dyskusyjne liveBook

Zakup książki *Stwórz własne AI* obejmuje bezpłatny dostęp do liveBook, anglojęzycznej internetowej platformy do czytania przygotowanej przez wydawnictwo Manning. Z wykorzystaniem wyjątkowych funkcji dyskusji liveBook można dodać komentarze do książki jako całości, a także do określonych podrozdziałów lub akapitów. Możesz z łatwością robić notatki, zadawać pytania techniczne i odpowiadać na nie, a także otrzymywać pomoc od autora i innych użytkowników. Aby uzyskać dostęp do forum, przejdź na stronę <https://livebook.manning.com/book/build-a-large-language-model-from-scratch/discussion>. O forach wydawnictwa Manning i zasadach postępowania na nich możesz również dowiedzieć się więcej na stronie <https://livebook.manning.com/discussion>.

Zobowiązaniem Manninga wobec Czytelników jest zapewnienie miejsca, w którym jest możliwy dialog między poszczególnymi Czytelnikami oraz między Czytelnikami a autorem. Nie jest to zobowiązanie do jakiegokolwiek konkretnego udziału ze strony autora, którego wkład w forum pozostaje dobrowolny (i nieodpłatny). Zachęcamy do zadania autorowi kilku trudnych pytań, tak aby nie stracił zainteresowania! Forum i archiwa poprzednich dyskusji będą dostępne na stronie wydawcy dopóty, dopóki książka będzie dostępna w druku.

Inne zasoby online

Interesują Cię najnowsze trendy w badaniach nad sztuczną inteligencją i modelami LLM?

- Zajrzyj na mój blog pod adresem <https://magazine.sebastianraschka.com>, gdzie regularnie omawiam najnowsze badania nad sztuczną inteligencją, z naciskiem na modele LLM.

Potrzebujesz pomocy w opanowaniu głębokiego uczenia i frameworka PyTorch?

- Na swojej stronie internetowej <https://sebastianraschka.com/teaching> oferuję kilka bezpłatnych kursów. Dzięki tym zasobom szybko zapoznasz się z najnowszymi technikami.

Szukasz dodatkowych materiałów związanych z książką?

- Odwiedź repozytorium GitHub książki pod adresem <https://github.com/rasbt/LLMs-from-scratch>. Znajdziesz tam dodatkowe zasoby i przykłady uzupełniające naukę.

O autorze

Dr SEBASTIAN RASCHKA od ponad dekady zajmuje się uczeniem maszynowym i sztuczną inteligencją. Oprócz tego, że jest badaczem, Sebastian ma prawdziwe zamiłowanie do edukacji. Jest znany ze swoich bestsellerowych książek na temat uczenia maszynowego w Pythonie oraz wkładu w rozwój oprogramowania open source.

Sebastian jest pracownikiem naukowym w Lightning AI, gdzie pracuje nad implementacją modeli LLM i ich szkoleniem. Zanim rozpoczął pracę w branży, był adiunktem w katedrze statystyki na Uniwersytecie Wisconsin-Madison, gdzie zajmował się m.in. badaniami nad uczeniem głębokim. Więcej informacji o Sebastianie znajdziesz na stronie <https://sebastianraschka.com>.

1

Czym są duże modele językowe?

W tym rozdziale:

- Ogólne objaśnienie podstawowych pojęć dotyczących dużych modeli językowych
- Architektura transformera, z którego wywodzą się modele LLM
- Plan budowania modelu LLM od podstaw

Duże modele językowe (ang. *Large Language Models* – LLM), takie jak te oferowane w systemie ChatGPT firmy OpenAI, to modele głębokich sieci neuronowych, które opracowano w ciągu ostatnich kilku lat. Modele LLM zapoczątkowały nową erę przetwarzania języka naturalnego (ang. *Natural Language Processing* – NLP). Zanim pojawiły się modele LLM, tradycyjne metody AI doskonale sprawdzały się w zadaniach kategoryzacji, takich jak klasyfikacja spamu e-mail, czy też proste zadania rozpoznawania wzorców, które można było opisać za pomocą ręcznie tworzonych reguł lub z użyciem prostszych modeli. Zazwyczaj jednak, zwłaszcza w zadaniach językowych, które wymagaly złożonych umiejętności rozumienia i tworzenia, takich jak parsowanie szczegółowych instrukcji, przeprowadzanie analizy kontekstowej czy tworzenie spójnego i odpowiedniego do kontekstu dokumentu, wyniki uzyskiwane za pomocą tradycyjnych metod były gorsze. Na przykład poprzednie generacje modeli językowych nie były w stanie wykonać tak trywialnego dla współczesnych modeli LLM zadania jak napisanie wiadomości e-mail na podstawie listy słów kluczowych.

Modele LLM mają niezwykle możliwości rozumienia, generowania i interpretowania ludzkiego języka. Warto jednak wyjaśnić, że kiedy mówimy, że modele językowe „rozumieją”, mamy na myśli ich zdolność przetwarzania i generowania tekstu w sposób, jaki wydaje się spójny i odpowiedni do kontekstu, a nie to, że mają ludzką świadomość lub wiedzę.

Dzięki postępu w uczeniu głębokim, które jest podzbiorem uczenia maszynowego i sztucznej inteligencji (AI) skoncentrowanym na sieciach neuronowych, modele LLM szkoli się na ogromnych ilościach danych tekstowych. To wielkoskalowe szkolenie pozwala modelom LLM na uchwycenie głębszych informacji kontekstowych i więcej subtelności ludzkiego języka w porównaniu z poprzednimi podejściami. W rezultacie wprowadzenie modeli LLM znacznie poprawiło wydajność w szerokim zakresie zadań NLP: w tłumaczeniach tekstu, analizie tonu, odpowiadaniu na pytania i wielu innych.

Inną ważną różnicą między współczesnymi modelami LLM a wcześniejszymi modelami NLP jest to, że wcześniejsze modele NLP zwykle projektowano do konkretnych zadań, takich jak kategoryzacja tekstu, tłumaczenie języka itp. Podczas gdy wspomniane wcześniejsze modele NLP wyróżniały się w wąskich zastosowaniach, modele LLM wykazują większą biegłość w szerokim zakresie zadań NLP.

Sukces modeli LLM można przypisać architekturze transformerów, która leży u podstaw wielu modeli LLM, oraz ogromnej ilości danych, na których LLM są szkolenie. Ta wielka ilość danych pozwala im uchwycić szeroką gamę niuansów językowych, kontekstów i wzorców, których ręczne zakodowanie byłoby wyzwaniem.

Zwrot w kierunku modeli opartych na architekturze transformerów i wykorzystania do szkolenia modeli LLM dużych zbiorów danych zasadniczo zmienił dziedzinę NLP — zapewnił bardziej wydajne narzędzia do rozumienia ludzkiego języka i wykonywania związanych z tym zadań.

Poniższe podrozdziały są podstawą do zrozumienia modeli LLM przez zaimplementowanie w kodzie krok po kroku opartego na architekturze transformera modelu LLM podobnego do ChatGPT.

1.1. Czym jest model LLM?

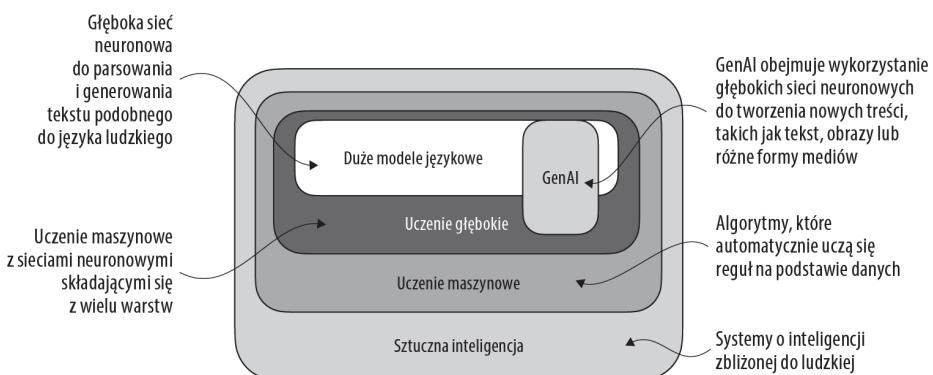
LLM to sieć neuronowa zaprojektowana do rozumienia ludzkiego języka, zdolna do generowania tekstu i reagowania na tekst w sposób podobny do tego, w jaki robią to ludzie. Modele LLM są głębokimi sieciami neuronowymi przeszkolonymi na ogromnych ilościach danych tekstowych, czasami obejmujących duże fragmenty całego tekstu publicznie dostępnego w internecie.

Słowo „duży” w nazwie „duży model językowy” odnosi się zarówno do rozmiaru modelu wyrażonego liczbą parametrów, jak i do ogromnego zbioru danych, na którym go przeszkołono. Takie modele często mają dziesiątki, a nawet setki miliardów parametrów. Są to dostrajalne wagie w sieci, optymalizowane podczas szkolenia w celu prognozowania następnego słowa w sekwencji. Prognozowanie następnego słowa ma sens, ponieważ wykorzystuje sekwencyjną naturę języka w kontekście szkolenia modeli w zakresie rozumienia kontekstu, struktury i relacji w tekście. Jest to jednak bardzo proste zadanie, dlatego wielu badaczy jest zaskoczonych tym, że w ten sposób można

stworzyć tak sprawne modele. W kolejnych rozdziałach omówię i zaimplementuję krok po kroku procedurę szkolenia w celu przewidywania następnego słowa.

Modele LLM wykorzystują architekturę określającą jako *transformer*, dzięki której podczas tworzenia prognoz mogą selektywnie zwracać uwagę na różne części danych wejściowych. Dzięki temu stają się szczególnie biegłe w radzeniu sobie z niuansami i złożonością ludzkiego języka.

Ponieważ modele LLM są zdolne do *generowania* tekstu, określa się je również jako formę *generatywnej sztucznej inteligencji*, nazywanej często w skrócie *GenAI*. Jak pokazano na rysunku 1.1, sztuczna inteligencja obejmuje szerszą dziedzinę maszyn zdolnych do tworzenia, które umieją wykonywać zadania wymagające inteligencji podobnej do ludzkiej, w tym rozumienia języka, rozpoznawania wzorców i podejmowania decyzji. Sztuczna inteligencja obejmuje takie poddziedziny jak uczenie maszynowe i uczenie głębokie.



Rysunek 1.1. Jak sugeruje to hierarchiczne przedstawienie relacji między różnymi dziedzinami AI, modele LLM reprezentują konkretne zastosowanie technik uczenia głębokiego. Wykorzystując ich podobną do ludzkiej zdolność do przetwarzania i generowania tekstu. Uczenie głębokie to wyspecjalizowana gałąź uczenia maszynowego, która koncentruje się na wykorzystaniu wielowarstwowych sieci neuronowych. Uczenie maszynowe i uczenie głębokie to dziedziny mające na celu implementację algorytmów umożliwiających komputerom uczenie się na podstawie danych i wykonywanie zadań, które zwykle wymagają ludzkiej inteligencji

Algorytmy wykorzystywane do implementowania sztucznej inteligencji są przedmiotem zainteresowania dziedziny uczenia maszynowego. W szczególności uczenie maszynowe obejmuje rozwijanie algorytmów, które potrafią uczyć się z danych i wykonywać prognozy lub podejmować decyzje na podstawie tych danych bez wyraźnego zaprogramowania logiki wnioskowania. Aby to zilustrować, jako praktyczne zastosowanie uczenia maszynowego wyobraźmy sobie filtr antyspamowy. Zamiast ręcznie pisać reguły identyfikacji spamu, wystarczy przekazać do algorytmu uczenia maszynowego przykłady wiadomości e-mail oznaczonych jako spam i wiadomości e-mail, które nie są spamem. Przez minimalizowanie błędu w swoich prognozach model uczy

się na zbiorze danych szkoleniowych rozpoznawać wzorce i cechy wskazujące na spam, co umożliwia klasyfikowanie nowych wiadomości e-mail jako spam bądź nie.

Jak pokazałem na rysunku 1.1, uczenie głębokie to podzbiór uczenia maszynowego, który koncentruje się na wykorzystaniu do modelowania w danych złożonych wzorców i abstrakcji sieci neuronowych obejmujących co najmniej trzy warstwy (znane również jako głębokie sieci neuronowe). W przeciwieństwie do uczenia głębokiego, tradycyjne uczenie maszynowe wymaga ręcznego wyodrębniania cech. Oznacza to, że zadanie zidentyfikowania i wybrania najbardziej odpowiednich cech dla modelu spoczywa na ludzkich ekspertach.

Chociaż dziedzina sztucznej inteligencji jest obecnie zdominowana przez uczenie maszynowe i uczenie głębokie, obejmuje również inne podejścia — na przykład wykorzystuje systemy oparte na regułach, algorytmy genetyczne, systemy eksperckie, logikę rozmytą oraz rozumowanie symboliczne.

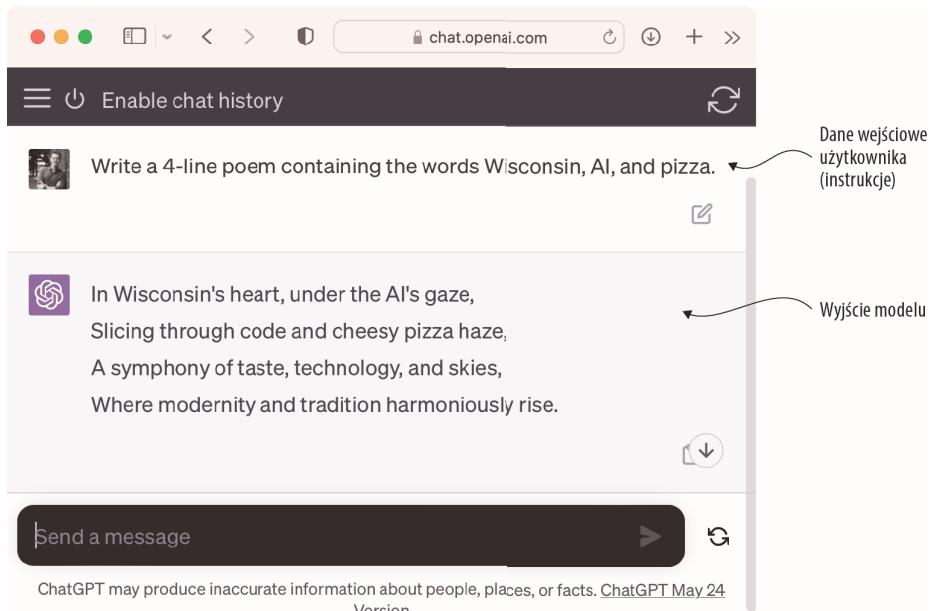
Wracając do przykładu klasyfikacji spamu, w tradycyjnym uczeniu maszynowym ludzcy eksperci mogą ręcznie wyodrębniać cechy z tekstu wiadomości e-mail, takie jak częstość niektórych słów wyzwalających (na przykład „nagroda”, „wygrałeś”, „za darmo”), liczba wykrzykników, użycie wszystkich wielkich liter lub obecność podejrzanych odnośników. Ten zbiór danych, utworzony na podstawie zdefiniowanych przez ekspertów cech, jest następnie wykorzystywany do przeszkoletnia modelu. W przeciwieństwie do tradycyjnego uczenia maszynowego, uczenie głębokie nie wymaga ręcznej ekstrakcji cech. Oznacza to, że ludzcy eksperci nie muszą identyfikować i wybierać najbardziej odpowiednich cech dla modelu (jednak zarówno tradycyjne techniki uczenia maszynowego, jak i techniki uczenia głębokiego stosowane do klasyfikacji spamu nadal wymagają gromadzenia etykiet, takich jak „spam” lub „nie spam”, które muszą być wybierane przez eksperta w dziedzinie lub użytkowników).

Przyjrzyjmy się wybranym problemom, które można dziś rozwiązać za pomocą modeli LLM, wyzwaniom, którym zastosowanie modeli LLM pozwoliło sprostać, oraz ogólnej architekturze modeli LLM, którą zaimplementujemy w dalszej części książki.

1.2. Zastosowania modeli LLM

Dzięki zaawansowanym możliwościom parsowania i rozumienia nieustrukturyzowanych danych tekstowych modele LLM mają szeroki zakres zastosowań w różnych dziedzinach. Obecnie modele LLM wykorzystuje się do tłumaczenia maszynowego, generowania nowych tekstów (patrz rysunek 1.2), analizy tonu, tworzenia streszczeń dokumentów i wielu innych zadań. Ostatnio używa się ich także do tworzenia treści, na przykład pisania beletryzki, artykułów, a nawet komputerowego kodu.

Modele LLM oferują również duże możliwości zastosowania w zaawansowanych chatbotach i wirtualnych asystentach, takich jak ChatGPT firmy OpenAI i Gemini firmy Google (dawniej znany jako Bard), które umieją odpowiadać na pytania użytkowników



Rysunek 1.2. Interfejsy modeli LLM umożliwiają komunikację między użytkownikami a systemami sztucznej inteligencji w języku naturalnym. Ten zrzut ekranu pokazuje ChatGPT piszący wiersz zgodnie ze specyfikacją użytkownika

i uzupełniać funkcjonalność tradycyjnych wyszukiwarek, takich jak Google Search i Microsoft Bing.

Co więcej, modele LLM można wykorzystywać do skutecznego wyszukiwania wiedzy z ogromnych ilości tekstu w specjalistycznych dziedzinach, takich jak medycyna czy prawo. Obejmuje to przeglądanie dokumentów, streszczenie długich fragmentów i odpowiadanie na techniczne pytania.

W skrócie – modele LLM są nieocenione w automatyzacji prawie każdego zadania związanego z analizą i generowaniem tekstu. Ich zastosowania są niemal nieograniczone, a dzięki ciągłym innowacjom i odkrywaniu nowych sposobów ich wykorzystania mają one potencjał, by na nowo określić naszą relację z techniką, czyniąc ją bardziej konwersacyjną, intuicyjną i dostępną.

W tej książce skoncentruję się na gruntownym zrozumieniu, jak działa LLM, oraz pokażę, jak zakodować model LLM, który potrafi generować teksty. Zapoznasz się także z technikami, które pozwalają modelom LLM wykonywać zapytania – od odpowiadania na pytania po tworzenie streszczeń tekstu, tłumaczenie tekstu na różne języki itp. Mówiąc inaczej, przy okazji budowania krok po kroku złożonego asystenta LLM, podobnego do ChatGPT, dowiesz się, jak działają takie systemy.

1.3. Etapy tworzenia modeli LLM i korzystania z nich

Po co mielibyśmy tworzyć własne modele LLM? Kodowanie modeli LLM od podstaw jest doskonałym ćwiczeniem pozwalającym zrozumieć ich mechanikę i ograniczenia. Ponadto pozwala zdobyć wiedzę niezbędną do wstępnego szkolenia istniejących architektur LLM typu open source bądź dostrajania ich do własnych zbiorów danych lub zadań specyficznych dla domeny.

UWAGA Większość współczesnych modeli LLM jest implementowana z użyciem biblioteki głębokiego uczenia PyTorch. W tej książce również skorzystamy z tej biblioteki. Obszerne wprowadzenie w tematykę biblioteki PyTorch znajdziesz w „Dodatku A”.

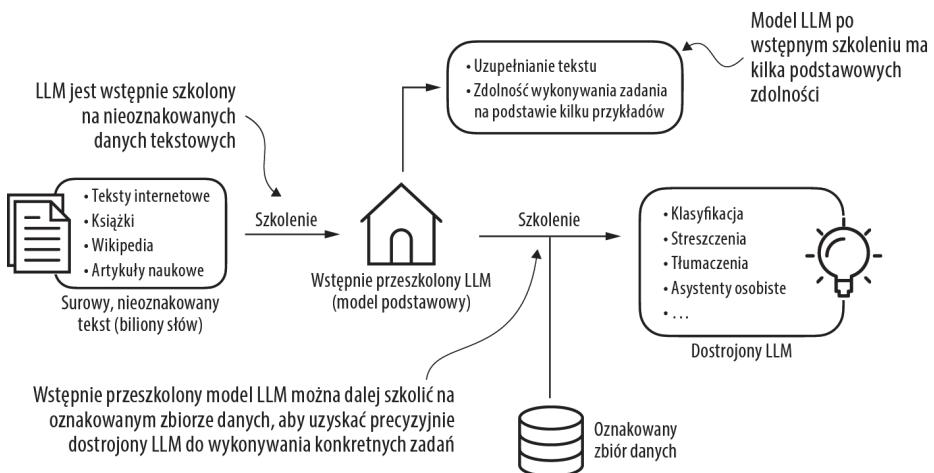
Z badań wynika, że niestandardowe modele LLM – te dostosowane do konkretnych zadań lub dziedzin – mogą przewyjszać wydajnością modele LLM ogólnego przeznaczenia, służące do wykonywania szerokiej gamy zadań, takie jak modele dostarczane przez system ChatGPT. Do przykładów takich programów można zaliczyć BloombergGPT (wyspecjalizowany w finansach) oraz modele LLM dostosowane pod kątem odpowiadania na pytania medyczne (więcej szczegółów w „Dodatku B”).

Korzystanie z niestandardowych rozwiązań LLM daje szereg korzyści, zwłaszcza w zakresie prywatności danych. Na przykład ze względu na obawy dotyczące poufności firmy mogą nie chcieć udostępniać wrażliwych danych zewnętrznym dostawcom LLM, takim jak OpenAI. Dodatkowo tworzenie mniejszych, niestandardowych modeli LLM pozwala na ich instalowanie bezpośrednio na urządzeniach klienckich, takich jak laptopy i smartfony. Nad takimi zastosowaniami pracuje obecnie wiele firm, na przykład Apple.

Lokalna implementacja pozwala znacznie zmniejszyć opóźnienia i obniżyć koszty związane z użytkowaniem serwera. Co więcej, niestandardowe modele LLM zapewniają programistom pełną autonomię, co pozwala im w razie potrzeby uzyskać kontrolę nad aktualizacjami i modyfikacjami modelu.

Ogólny proces tworzenia modelu LLM obejmuje szkolenie wstępne i dostrajanie. Słowo „wstępne” w pojęciu „wstępne szkolenie” odnosi się do początkowej fazy, w której model taki jak LLM jest szkolony na dużym, zróżnicowanym zbiorze danych. Celem tej fazy jest rozwinięcie w modelu ogólnego zrozumienia języka. Taki wstępnie przeszkolony model można następnie wykorzystać jako podstawowy zasób i udoskonalać go przez *dostrajanie*. Proces dostrajania polega na specjalistycznym szkoleniu modelu na węższym zestawie danych, który jest dobrany pod kątem określonych zadań lub dziedzin. To dwuetapowe podejście do szkolenia, składające się ze szkolenia wstępnego i dostrajania, jest przedstawione na rysunku 1.3.

Pierwszym krokiem w tworzeniu LLM jest przeszkołenie go na dużym korpusie danych tekstowych, czasami nazywanym *surowym tekstem*. W tym przypadku słowo „surowy” odnosi się do faktu, że dane są zwykłym tekstem bez żadnych informacji pełniących



Rysunek 1.3. Wstępne szkolenie modelu LLM obejmuje przewidywanie następnego słowa na podstawie obszernych zbiorów danych tekstowych. Wstępnie przeszkolony model LLM można następnie dostroić z użyciem mniejszego, oznakowanego zbioru danych

funkcję etykiet (można zastosować filtrowanie, na przykład usuwanie znaków formattowania lub dokumentów w nieznanych językach).

UWAGA Czytelnicy z doświadczeniem w uczeniu maszynowym być może zauważą, że w przypadku tradycyjnych modeli uczenia maszynowego i głębokich sieci neuronowych szkolonych za pomocą konwencjonalnego paradygmatu uczenia nadzorowanego etykiety zwykle są wymagane. Nie dotyczy to jednak etapu wstępnego szkolenia modeli LLM. W tej fazie modele LLM wykorzystują uczenie samonadzorowane, w którym model samodzielnie generuje etykiety na podstawie danych wejściowych.

Ten pierwszy etap szkolenia modelu LLM jest również znany jako *szkolenie wstępne* (ang. *pretraining*), w którego wyniku powstaje początkowy, wstępnie przeszkolony model LLM, często nazywany modelem bazowym lub podstawowym (ang. *Foundation model*). Typowym przykładem takiego modelu jest GPT-3 (prekursor oryginalnego modelu oferowanego w ChatGPT). Ten model jest zdolny do uzupełniania tekstu – czyli potrafi dokończyć dostarczone przez użytkownika w połowie napisane zdanie. Ma także ograniczone możliwości uczenia się na podstawie niewielu przykładów, co oznacza, że potrafi nauczyć się wykonywać nowe zadania na podstawie zaledwie kilku przykładów i nie wymaga korzystania z obszernych danych szkoleniowych.

Po uzyskaniu po szkoleniu na dużych zbiorach danych tekstowych modelu LLM wstępnie przeszkolonego pod kątem przewidywania następnego słowa w tekście można szkolić model LLM dalej, na danych oznaczonych. Ten etap szkolenia określa się też jako *dostrajanie*.

Dwie najpopularniejsze kategorie dostrajania modeli LLM to *dostrajanie instrukcji* i *dostrajanie klasyfikacji*. W przypadku dostrajania instrukcji oznaczony zbiór danych składa się z par instrukcji i odpowiedzi, takich jak zapytanie o przetłumaczenie

tekstu razem z poprawnie przetłumaczonym tekstem. W przypadku dostrajania klasyfikacji oznaczony zbiór danych składa się z tekstów i powiązanych etykiet klas — na przykład wiadomości e-mail powiązanych z etykietami „spam” i „nie spam”.

W dalszej części książki omówię implementację wstępnego szkolenia i dostrajania modeli LLM, a także zaprezentuję specyfikę dostrajania zarówno instrukcji, jak i klasyfikacji po wstępny przeszkołeniu podstawowego modelu LLM.

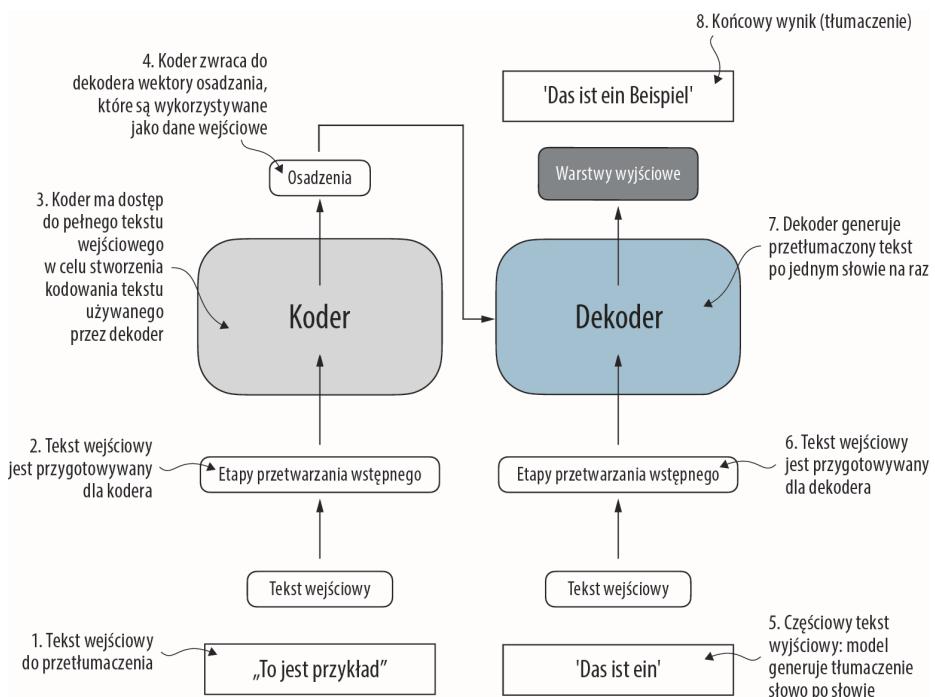
1.4. Wprowadzenie do architektury transformerów

Większość nowoczesnych modeli LLM opiera się na architekturze *Original Transformer*, czyli głębszej sieci neuronowej przedstawionej w 2017 roku w artykule *Attention Is All You Need* (<https://arxiv.org/abs/1706.03762>). Aby zrozumieć modele LLM, trzeba zapoznać się z tą architekturą, którą opracowano na potrzeby tłumaczenia maszynowego tekstów w języku angielskim na język niemiecki i francuski. Uproszczona wersja architektury transformera jest pokazana na rysunku 1.4.

Architektura transformera składa się z dwóch podmodułów: kodera i dekodera. Moduł kodera przetwarza tekst wejściowy i koduje go do postaci ciągu reprezentacji liczbowych lub wektorów opisujących kontekst danych wejściowych. Następnie moduł dekodera pobiera zakodowane wektory i generuje tekst wyjściowy. Na przykład w zadaniu tłumaczenia koder koduje tekst z języka źródłowego do postaci wektorowej, a dekoder dekoduje te wektory w celu wygenerowania tekstu w języku docelowym. Zarówno koder, jak i dekoder składają się z wielu warstw połączonych tak zwanym *mechanizmem samouwagi* (ang. *self-attention mechanism*). Na temat sposobu wstępnego przetwarzania i kodowania danych wejściowych można zadać wiele pytań. Odpowiem na nie krok po kroku w kolejnych rozdziałach.

Kluczowym komponentem transformerów i modeli LLM jest mechanizm samouwagi (na rysunku 1.4 go nie pokazano), który dostarcza modelowi wagi znaczenia różnych słów lub tokenów w sekwencji względem siebie. Dzięki temu mechanizmowi model może uchwycić w danych wejściowych odległe zależności i relacje kontekstowe, co zwiększa jego zdolności do generowania spójnych i kontekstowo istotnych danych wyjściowych. Ze względu na jego złożoność dokładniejszy opis jego działania odłożę do rozdziału 3., w którym go omówię i krok po kroku zaimplementuję.

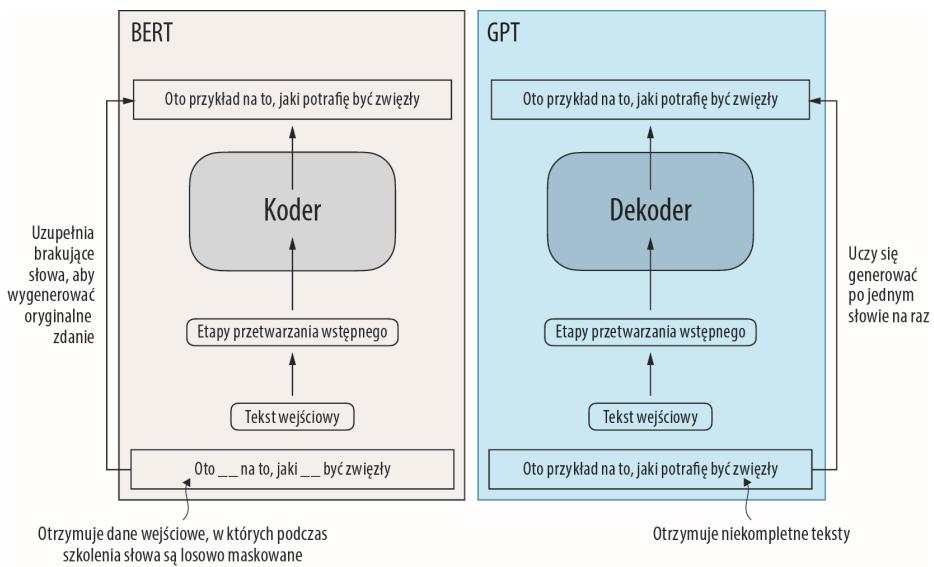
Na tym pojęciu opierały się późniejsze odmiany architektury transformerów, takie jak BERT (skrót od *bidirectional encoder representations from transformers* – dosłownie: dwukierunkowe reprezentacje kodera z transformerów) i różne modele GPT (skrót od *generative pretrained transformers* – dosłownie: generatywne, wstępnie przeszkołone transformery). Opracowanie tych transformerów pozwoliło dostosować architekturę transformera do wykonywania różnych zadań. Jeśli jesteś zainteresowany, zapoznaj się z „Dodatkiem B”, w którym znajdziesz sugestie dotyczące lektury uzupełniającej.



Rysunek 1.4. Uproszczona reprezentacja architektury Original Transformer, będącej modelem uczenia głębokiego do tłumaczeń językowych. Transformer składa się z dwóch części: (a) kodera, który przetwarza tekst wejściowy i tworzy reprezentację osadzeń tekstu (reprezentację numeryczną wielu różnych czynników w różnych wymiarach), którą (b) dekoder może wykorzystać w celu wygenerowania przetłumaczonego tekstu po jednym słowie na raz. Ten rysunek przedstawia końcowy etap procesu tłumaczenia, w którym dekoder, aby wykonać tłumaczenie oryginalnego tekstu wejściowego („This is an example”), musi wygenerować tylko słowo końcowe („Beispiel”) i częściowo przetłumaczone zdanie („Das ist ein”)

Model BERT, który jest zbudowany na podstawie podmodułu kodera architektury *Original Transformer*, różni się od modeli GPT podejściem do szkolenia. O ile GPT jest przeznaczony do zadań generatywnych, o tyle BERT i jego odmiany specjalizują się w przewidywaniu zamaskowanych słów (model przewiduje zamaskowane lub ukryte słowa w danym zdaniu – rysunek 1.5). Dzięki tej unikatowej strategii szkoleniowej model BERT sprawdza się w zadaniach klasyfikacji tekstu, w tym zadaniach oznaczania tonu i kategoryzacji dokumentów. W chwili gdy piszę te słowa, modelu BERT używa się do wykrywania toksycznych treści na platformie X (dawniej Twitter).

Z drugiej strony model GPT skupia się na części dekodera architektury *Original Transformer* i jest przeznaczony do zadań wymagających generowania tekstu. Obejmuje to tłumaczenie maszynowe, streszczanie tekstu, pisanie beletryztyki, pisanie kodu komputerowego i wiele innych.



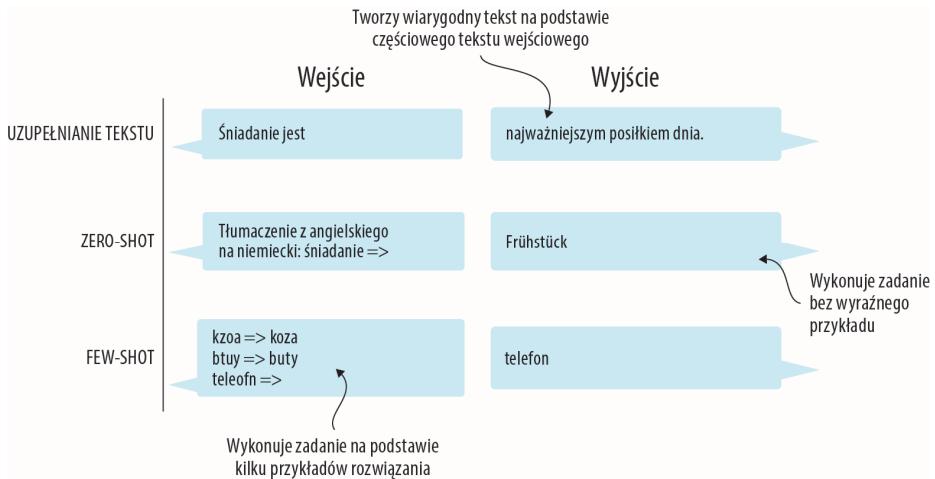
Rysunek 1.5. Wizualna reprezentacja podmodułów transformera: kodera i dekodera.

Po lewej: segment kodera jest przykładem modelu LLM podobnego do BERT, który koncentruje się na prognozowaniu zamaskowanych słów i jest używany głównie do takich zadań, jak klasyfikacja tekstu. Po prawej: segment dekodera prezentuje modele LLM podobne do GPT, zaprojektowane do zadań generatywnych i tworzenia spójnych sekwencji tekstowych

Modele GPT, zaprojektowane i przeszkolone głównie do wykonywania zadań uzupełniania tekstu, również wykazują niezwykłą wszechstronność. Modele te doskonale sprawdzają się w wykonywaniu zarówno zadań typu *zero-shot* (niewymagających podawania przykładów rozwiązań), jak i *few-shot* (wymagających podania zaledwie kilku przykładów rozwiązań). Pojęcie *zero-shot* odnosi się do zdolności uogólniania modelu na zadania, których model nigdy nie widział (tzn. nigdy wcześniej nie przekazano mu żadnych konkretnych przykładów rozwiązywania tego zadania). Z drugiej strony uczenie typu *few-shot* polega na uczeniu się na podstawie minimalnej liczby przykładów, które użytkownik podaje jako dane wejściowe (rysunek 1.6).

1.5. Wykorzystanie dużych zbiorów danych

Duże zbiorы danych szkoleniowych dla popularnych modeli GPT i BERT reprezentują różnorodne i obszerne korpusy tekstowe, obejmujące miliardy słów i dotyczące szerokiego zakresu tematów oraz języków naturalnych i komputerowych. W ramach konkretnego przykładu w tabeli 1.1 zestawiłem zbiorów danych wykorzystane do wstępniego szkolenia modelu GPT-3 – bazowego modelu dla pierwszej wersji ChatGPT.



Rysunek 1.6. Modele LLM podobne do GPT oprócz uzupełniania tekstu potrafią na podstawie danych wejściowych rozwiązywać różne zadania bez konieczności ponownego szkolenia, dostrajania ani zmian architektury modelu specyficznych dla zadania. Czasami w ramach danych wejściowych przydaje się podanie przykładów celu. Takie zadania określa się terminem few-shot. Jednak modele LLM podobne do GPT są również zdolne do wykonywania zadań bez konkretnego przykładu, co nazywa się konfiguracją zero-shot

Transformery kontra modele LLM

Współczesne modele LLM są oparte na architekturze transformera. Z tego względu pojęcia transformer i model LLM często są używane w literaturze jako synonimy. Należy jednak pamiętać, że nie wszystkie transformery są modelami LLM. Transformery można także wykorzystywać w zadaniach widzenia komputerowego. Ponadto nie wszystkie modele LLM są transformerami — istnieją modele LLM oparte na architekturach sieci rekurencyjnych i konwolucyjnych. Główną motywacją stojącą za tymi alternatywnymi podejściami jest poprawa wydajności obliczeniowej LLM. To, czy te alternatywne architektury LLM mogą konkurować z możliwościami modeli LLM opartych na transformerach i czy ostatecznie się przyjmą, dopiero się okaże. Dla uproszczenia terminu „LLM” będę używać w odniesieniu do podobnych do GPT modeli LLM opartych na transformerach (zainteresowani Czytelnicy mogą sięgnąć do „Dodatku B”, w którym zamieściłem odnośniki do zasobów opisujących te architektury).

W tabeli 1.1 zestawiono liczbę tokenów, przy czym token rozumie się jako jednostkę tekstu odczytywaną przez model, a liczba tokenów w zbiorze danych jest w przybliżeniu równoważna liczbie słów i znaków interpunkcyjnych w tekście. Proces tokenizacji, czyli przekształcania tekstu na tokeny, opisałem w rozdziale 2.

Dzięki ogromnej skali i różnorodności zestawu danych modele zyskują zdolność radzenia sobie z wieloma różnorodnymi wyzwaniami. Potrafią analizować składnię, rozumieją znaczenie i kontekst języka, a nawet potrafią sprostać zadaniom wymagającym szerszej wiedzy o świecie — to najważniejszy wniosek, jaki można wyciągnąć z danych zaprezentowanych w tej tabeli.

Tabela 1.1. Zbiór danych użyty do wstępnego szkolenia popularnego modelu LLM GPT-3

Nazwa zbioru danych	Opis zbioru danych	Liczba tokenów	Proporcja w danych szkoleniowych
CommonCrawl (filtrowany)	Dane indeksowania stron internetowych	410 miliardów	60%
WebText2	Dane indeksowania stron internetowych	19 miliardów	22%
Books1	Internetowy korpus książek	12 miliardów	8%
Books2	Internetowy korpus książek	55 miliardów	8%
Wikipedia	Tekst wysokiej jakości	3 miliardy	3%

Szczegóły zbioru danych modelu GPT-3

W tabeli 1.1 zaprezentowałem zbiór danych wykorzystany w modelu GPT-3. Kolumna proporcji w tabeli sumuje się do 100% próbkowanych danych (w granicach błędu wynikającego z zaokrągleń). Choć podzbiory w kolumnie **Liczba tokenów** obejmują łącznie 499 miliardów, model przeszkolony tylko na 300 miliardach tokenów. Autorzy artykułu poświęconego modelowi GPT-3 nie sprecyzowali, dlaczego model nie został przeszkolony na wszystkich 499 miliardach tokenów.

Aby zrozumieć kontekst, rozważmy rozmiar zbioru danych *CommonCrawl*, który sam w sobie składa się z 410 miliardów tokenów i wymaga około 570 GB pamięci. Dla porównania, w późniejszych iteracjach modeli podobnych do GPT-3, takich jak LLaMA firmy Meta, rozszerzono zakres szkolenia o dodatkowe źródła danych, na przykład o artykuły naukowe w serwisie Arxiv (92 GB), a także pytania i odpowiedzi związane z kodem z serwisu StackExchange (78 GB).

Autorzy artykułu poświęconego modelowi GPT-3 nie udostępnili szkoleniowego zbioru danych, ale porównywalnym zbiorem, który jest publicznie dostępny, jest zbiór udostępniony w pracy Soldaini et al. 2024, *Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research* (<https://arxiv.org/abs/2402.00159>). Ten zbiór może jednak zawierać utwory chronione prawem autorskim, a dokładne warunki korzystania mogą zależeć od zamierzonego przypadku użycia i kraju.

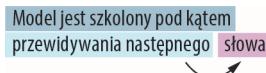
Modele, które wstępnie przeszkolono, są niezwykle wszechstronne i łatwo można je dostosować do różnych zadań. Z tego względu nazywa się je modelami bazowymi lub podstawowymi. Wstępne szkolenie modeli LLM wymaga dostępu do znacznych zasobów i jest bardzo kosztowne. Przykładowo koszt wstępnego szkolenia modelu GPT-3 z uwzględnieniem cen usług chmurowych oszacowano na 4,6 miliona dolarów (<https://mng.bz/VxEW>).

Na szczęście do pisania, wyodrębniania i edytowania tekstów, które nie były częścią danych szkoleniowych, można wykorzystać narzędzia ogólnego przeznaczenia oraz wiele wstępnie przeszkolonych modeli LLM typu *open source*. Ponadto modele LLM można precyzyjnie dostroić do określonych zadań z użyciem znaczco mniejszych zbiorów danych, co zmniejsza ilość potrzebnych zasobów obliczeniowych i poprawia wydajność.

W dalszej części książki zaprezentuję przeznaczony do wstępniego szkolenia modelu LLM kod, z którego skorzystamy w celach edukacyjnych. Wszystkie obliczenia można wykonać na sprzęcie konsumenckim. Po opracowaniu kodu wstępniego szkolenia nauczysz się wykorzystywać dostępne za darmo wagi modeli i ładować je do tworzonej architektury, co pozwoli pominąć kosztowny etap wstępniego szkolenia podczas dostrajania modelu LLM.

1.6. Szczegóły architektury modeli GPT

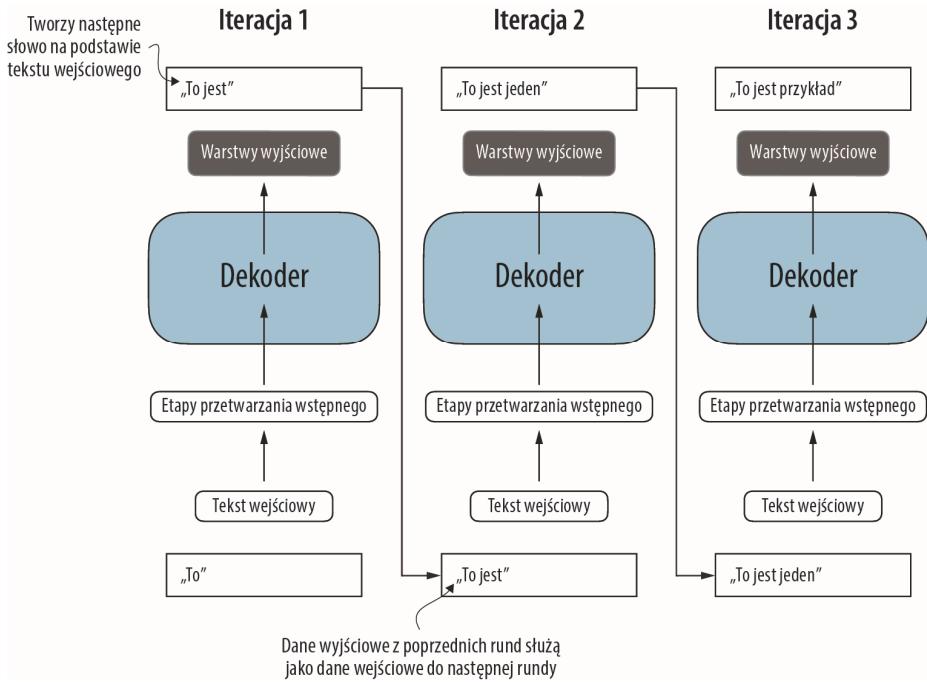
Model GPT po raz pierwszy przedstawiono w artykule *Improving Language Understanding by Generative Pre-Training* (<https://mng.bz/x2qg>), opracowanym przez zespół Radford et al. z firmy OpenAI. GPT-3 to skalowana wersja tego modelu, która ma więcej parametrów i została przeszkolona na większym zbiorze danych. Ponadto oryginalny model oferowany w systemie ChatGPT stworzono przez dostrojenie modelu GPT-3 na dużym zbiorze danych z użyciem metody opisanej w opublikowanym przez firmę OpenAI artykule *InstructGPT* (<https://arxiv.org/abs/2203.02155>). Jak pokazałem na rysunku 1.6, modele te są zdolne do wykonywania zadań uzupełniania tekstu, a także innych, takich jak korekta pisowni, klasyfikacja i tłumaczenia. To niezwykłe, zważywszy na to, że modele GPT wstępnie przeszkolono na stosunkowo prostym zadaniu przewidywania następnego słowa (co pokazałem na rysunku 1.7).



Rysunek 1.7. W zadaniu wstępniego szkolenia modeli GPT w przewidywaniu następnego słowa system uczy się przewidywać kolejne słowo w zdaniu na podstawie słów, które pojawiły się przed nim. Takie podejście pomaga modelowi zrozumieć, w jaki sposób w języku zazwyczaj pasują do siebie słowa i frazy. To jest podstawa, którą można zastosować do różnych innych zadań

Zadanie przewidywania następnego słowa jest formą uczenia samonadzorowanego, które jest rodzajem samodzielnego znakowania. Oznacza to, że nie ma potrzeby jawnego wybierania etykiet dla danych szkoleniowych, a do tego celu można wykorzystać strukturę samych danych. Następne słowo w zdaniu lub dokumencie może posłużyć jako etykieta, którą model powinien przewidzieć. Ponieważ zadanie przewidywania następnego słowa pozwala tworzyć etykiety „w locie”, do szkolenia modeli LLM można użyć ogromnej liczby nieoznakowanych zbiorów danych tekstowych.

W porównaniu z architekturą *Original Transformer*, którą omówiłem w podrozdziale 1.4, ogólna architektura modelu GPT jest stosunkowo prosta. W istocie jest to sam dekoder, bez kodera (rysunek 1.8). Ponieważ modele typu „sam dekoder”, takie jak GPT, generują tekst przez przewidywanie tekstu po jednym słowie na raz, uważa się je za rodzaj modelu *autoregresyjnego*. Modele autoregresyjne wykorzystują swoje



Rysunek 1.8. Architektura GPT wykorzystuje tylko część dekodera z architektury *Original Transformer*. Zaprojektowano ją do przetwarzania jednokierunkowego, od lewej do prawej, dzięki czemu dobrze nadaje się do generowania tekstu i zadań przewidywania następnego słowa w celu generowania tekstu w sposób iteracyjny, po jednym słowie na raz

poprzednie wyniki jako dane wejściowe do przyszłych prognoz. W rezultacie w modelu GPT każde nowe słowo jest wybierane na podstawie poprzedzającej je sekwencji, co poprawia spójność wynikowego tekstu.

Takie architektury jak GPT-3 są również znacznie obszerniejsze od modelu *Original Transformer*. Na przykład w architekturze *Original Transformer* bloki kodera i dekodera były powielone sześciokrotnie. Model GPT-3 ma 96 warstw transformerów i łącznie 175 miliardów parametrów.

Model GPT-3 wprowadzono w 2020 roku, co według standardów uczenia głębokiego i rozwoju modeli LLM uznaje się za odległą przeszłość. Jednak nowsze architektury, takie jak modele Llama firmy Meta, nadal opierają się na tych samych podstawowych pojęciach i wprowadzono w nich jedynie niewielkie modyfikacje. W związku z tym zrozumienie modeli GPT nadal jest bardzo istotne. Z tego powodu skupię się na zaimplementowaniu istotnej architektury modelu GPT, a jednocześnie wskażę konkretne poprawki zastosowane w alternatywnych modelach LLM.

Chociaż model *Original Transformer*, składający się z bloków kodera i dekodera, pierwotnie zaprojektowano do tłumaczenia języków, modele GPT — pomimo ich obszerniejszej, ale prostszej architektury „tylko dekoder”, której celem jest przewidywanie

następnego słowa — również są zdolne do wykonywania zadań tłumaczeniowych. Ta zdolność była początkowo zaskoczeniem dla badaczy, ponieważ wyłoniła się z modelu przeszkolonego głównie w przewidywaniu następnego słowa, czyli zadaniu, które nie było specjalnie ukierunkowane na tłumaczenie.

Zdolność modelu do wykonywania zadań, do których specjalnie go nie szkolono, nazywa się *zachowaniem emergentnym*. Zdolność ta nie jest wyraźnie nauczana podczas szkolenia, ale pojawia się jako naturalna konsekwencja ekspozycji modelu na ogromne ilości wielojęzycznych danych w różnych kontekstach. Fakt, że modele GPT mogą „uczyć się” wzorców tłumaczeń między językami i wykonywać zadania tłumaczeniowe, nawet jeśli nie zostały do tego specjalnie przeszkolone, pokazuje korzyści i możliwości tych wielkoskalowych, generatywnych modeli językowych. Za ich pomocą można wykonywać różne zadania bez konieczności używania różnych modeli do różnych typów zadań.

1.7. Tworzenie dużego modelu językowego

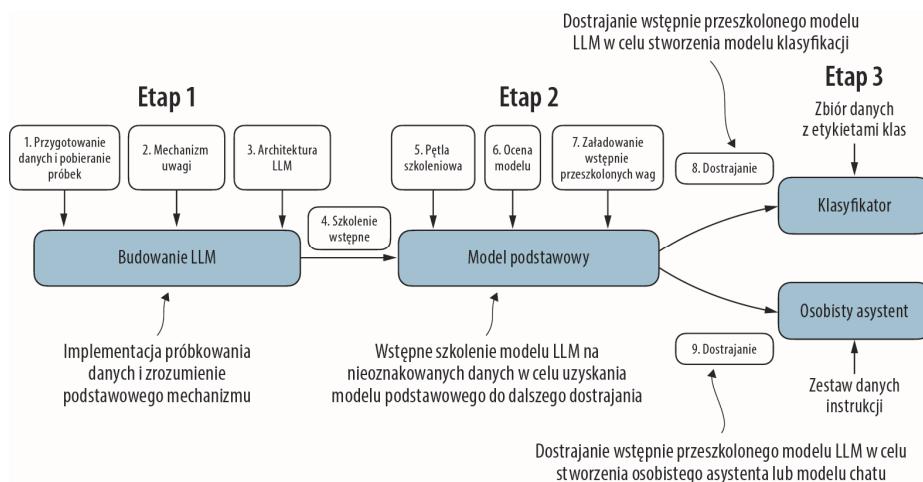
Po zaprezentowaniu podstaw dotyczących modeli LLM spróbujemy zakodować taki model od początku. Przyjmiemy za plan fundamentalną ideę modelu GPT i zrealizujemy implementację modelu w trzech etapach (rysunek 1.9).

Podczas etapu 1. zapoznam Czytelnika z podstawowymi krokami wstępnego przetwarzania danych i sposobem kodowania mechanizmu uwagi, będącego sercem każdego modelu LLM. Następnie, w etapie 2., pokażę, jak zakodować i wstępnie przeszkoślić model LLM podobny do GPT, zdolny do generowania nowych tekstów. Omówię również podstawy oceny modeli LLM, która jest niezbędna do tworzenia wydajnych systemów NLP.

Wstępne szkolenie modelu LLM od podstaw to znaczące przedsięwzięcie, z którym w przypadku modeli podobnych do GPT wiążą się koszty od kilku tysięcy do wielu milionów dolarów. Z tego powodu w etapie 2. skoncentruję się na implementacji szkolenia dla celów edukacyjnych, z użyciem niewielkiego zbioru danych. Ponadto udostępnę przykłady kodu umożliwiające załadowanie ogólnodostępnych wag modeli.

Na koniec, w trakcie etapu 3., pokażę, jak dostroić wstępnie przeszkolony model LLM do wykonywania takich instrukcji jak odpowiadanie na pytania lub klasyfikowanie tekstu — czyli najczęstszych zadań w wielu rzeczywistych aplikacjach i badaniach.

Mam nadzieję, że z niecierpliwością czekasz, aż wyruszmy w tę ekscytującą podróż!



Rysunek 1.9. Trzy główne etapy kodowania modelu LLM to implementacja architektury LLM i proces przygotowania danych (etap 1.), wstępne szkolenie modelu LLM w celu utworzenia modelu podstawowego (etap 2.) oraz dostrajanie modelu podstawowego, aby stał się osobistym asystentem lub klasyfikatorem tekstu (etap 3.).

Podsumowanie

- Modele LLM znacząco zmieniły dziedzinę przetwarzania języka naturalnego, w której wcześniej wykorzystywano głównie systemy oparte na jawnych regułach i prostszych metodach statystycznych. Wraz z modelami LLM pojawiły się nowe podejścia oparte na uczeniu głębokim, co pozwoliło osiągnąć postępy w zakresie rozumienia, generowania i tłumaczenia ludzkiego języka.
- Szkolenie współczesnych modeli LLM przebiega w dwóch głównych etapach:
 - Po pierwsze, wstępnie szkoli się je na dużym korpusie nieoznaczonych tekstu, z wykorzystaniem w roli etykiety prognozy następnego słowa w zdaniu.
 - Następnie dostraja się je na docelowym, mniejszym i oznakowanym, zbiorze danych pod kątem postępowania według instrukcji lub wykonywania zadań klasyfikacji.
- Modele LLM są oparte na architekturze transformera. Kluczowym komponentem architektury transformera jest mechanizm uwagi, który podczas generowania wyjścia po jednym słowie na raz daje modelowi LLM selektywny dostęp do całej sekwencji wejściowej.
- Architektura *Original Transformer* składa się z kodera do parsowania tekstu i dekodera do generowania tekstu.

- Modele LLM do generowania tekstu i wykonywania instrukcji, takie jak GPT-3 i ChatGPT, implementują tylko moduły dekodera, co upraszcza architekturę.
- Do wstępnego szkolenia modeli LLM niezbędne są duże zbiory danych, składające się z miliardów słów.
- Podczas gdy ogólnym zadaniem szkolenia wstępnego dla modeli podobnych do GPT jest przewidywanie następnego słowa w zdaniu, te modele LLM wykazują właściwości emergentne, takie jak zdolność do klasyfikowania, tłumaczenia lub tworzenia streszczeń dokumentów.
- W wyniku wstępnego przeszkolenia modelu LLM powstaje model podstawowy, który można bardziej efektywnie dostroić do różnych zadań końcowych.
- Modele LLM precyzyjnie dostrojone na niestandardowych zestawach danych mogą w określonych zadaniach przewyższać wydajnością uniwersalne modele LLM.

Praca z danymi tekstowymi

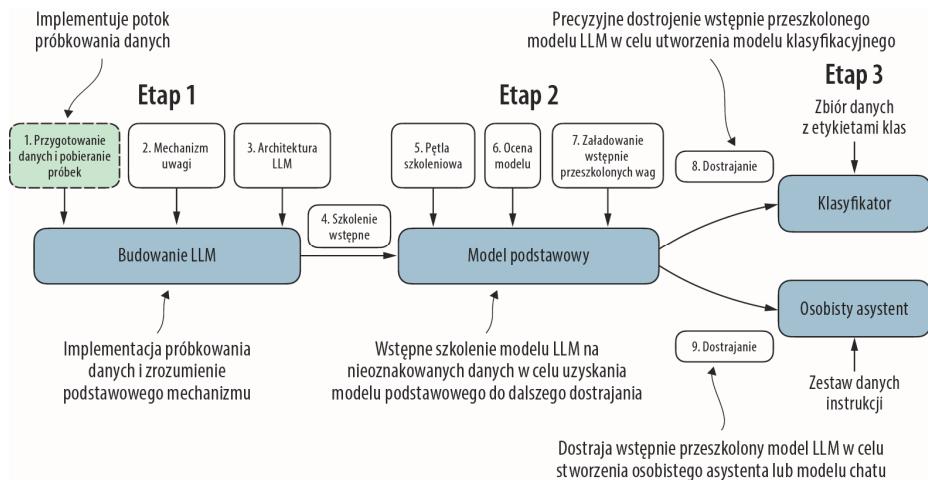
W tym rozdziale:

- Przygotowanie tekstu do szkolenia dużych modeli językowych
- Podział tekstu na tokeny słów i podłów
- Kodowanie par bajtów jako bardziej zaawansowany sposób tokenizacji tekstu
- Próbkowanie przykładów szkoleniowych za pomocą metody okna przesuwnego
- Konwersja tokenów na wektory w celu przekazania ich do modelu LLM

Dotychczas omówiłem ogólną strukturę dużych modeli językowych (LLM) i powiedziałem, że są one wstępnie szkolone na ogromnych ilościach tekstu. W szczególności skupiłem się na modelach LLM opartych wyłącznie na dekoderach i architekturze transformera, która leży u podstaw modeli używanych w ChatGPT i innych popularnych modeli LLM podobnych do GPT.

Na etapie szkolenia wstępnego model LLM przetwarza tekst po jednym słowie na raz. W wyniku szkolenia modeli LLM obejmujących od wielu milionów do wielu miliardów parametrów z użyciem zadania przewidywania następnego słowa uzyskujemy modele o imponujących możliwościach. Modele te można następnie dostroić w taki sposób, aby wykonywały ogólne instrukcje lub konkretne zadania końcowe. Jednak przed przystąpieniem do implementacji i szkolenia modelu LLM trzeba przygotować szkoleniowy zbiór danych, zgodnie z procedurą pokazaną na rysunku 2.1.

Z tego rozdziału dowiesz się, jak przygotować tekst wejściowy do szkolenia modelu LLM. Obejmuje to podział tekstu na pojedyncze tokeny słów i podłów, które następnie można zakodować w reprezentacjach wektorowych wykorzystywanych przez model LLM. Zapoznasz się również z zaawansowanymi schematami tokenizacji, takimi



Rysunek 2.1. Trzy główne etapy kodowania modelu LLM. Ten rozdział koncentruje się na kroku 1. etapu 1. — implementacji potoku próbkowania danych

jak kodowanie par bajtów wykorzystywane w popularnych programach LLM opartych na modelach GPT. Na koniec zaimplementujemy strategię próbkowania i ładowania danych w celu uzyskania par wejść i wyjść niezbędnych do szkolenia modelu LLM.

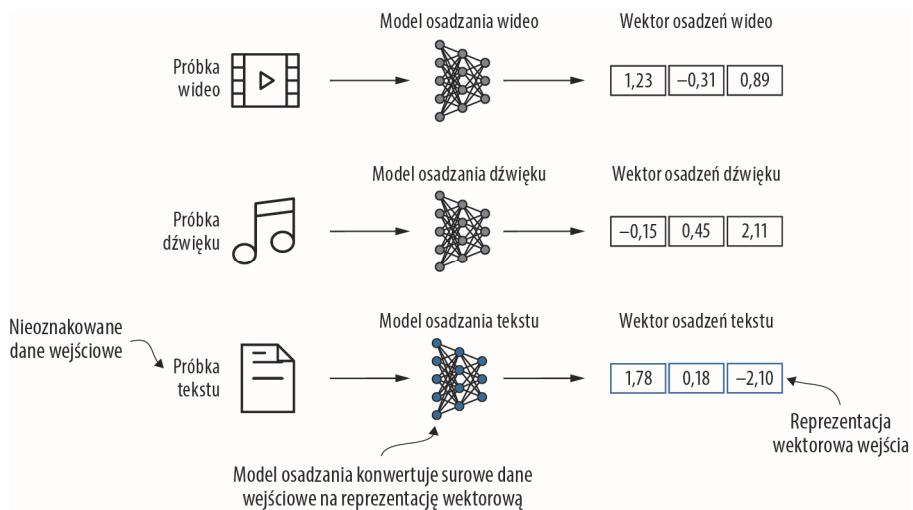
2.1. Czym są osadzenia słów?

Modele głębokich sieci neuronowych, włącznie z modelami LLM, nie potrafią bezpośrednio przetwarzać surowego tekstu. Ponieważ tekst jest skategoryzowany, nie nadaje się do wykonywania działań matematycznych wykorzystywanych w procesach implementacji i szkolenia sieci neuronowych. Dlatego trzeba znaleźć sposób na reprezentowanie słów w postaci wektorów zawierających ciągle wartości.

UWAGA Więcej informacji o wektorach i tensorach w kontekście obliczeniowym można znaleźć w „Dodatku A”, w punkcie A.2.2.

Konwersję danych na format wektorowy często określa się jako *osadzanie* (ang. *embedding*). Korzystając z określonej warstwy sieci neuronowej lub innego wstępnie przeszkolonego modelu sieci neuronowej, można osadzać różne typy danych — na przykład wideo, audio i tekst (rysunek 2.2). Należy jednak pamiętać, że różne formaty danych wymagają różnych modeli osadzeń. Na przykład model osadzania zaprojektowany pod kątem tekstu nie jest odpowiedni do osadzania danych audio lub wideo.

U podstaw osadzania leży mapowanie dyskretnych obiektów, takich jak słowa, obrazy, a nawet całe dokumenty, na punkty w ciągłej przestrzeni wektorowej — głównym celem osadzania jest konwersja danych nieliczbowych na format, który można przetwarzać w sieciach neuronowych.

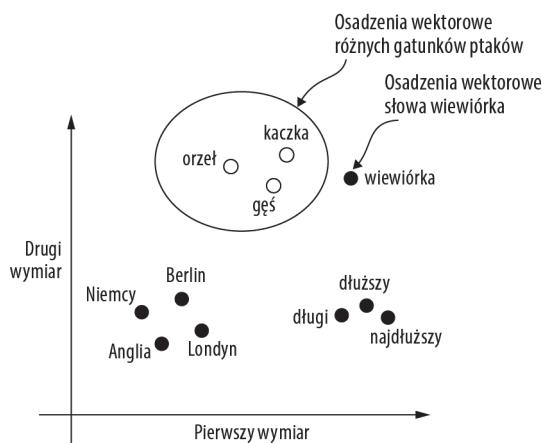


Rysunek 2.2. Modele uczenia głębokiego nie potrafią przetwarzać takich formatów danych jak wideo, audio i tekst w surowej postaci. Z tego powodu używamy modelu osadzania, którego zadaniem jest przekształcenie surowych danych w gęstą reprezentację wektorową, którą architektury uczenia głębokiego mogą łatwo zrozumieć i przetworzyć. W szczególności ten rysunek ilustruje proces przekształcania surowych danych w trójwymiarowy wektor liczbowy

Podczas gdy osadzanie słów jest najbardziej powszechną formą osadzania tekstu, istnieją również osadzenia zdań, akapitów lub całych dokumentów. Osadzanie zdań lub akapitów jest popularną opcją stosowaną w przypadku techniki RAG (ang. *retrieval augmented generation* – dosłownie: generowanie wspomagane wyszukiwaniem). Technika RAG łączy generowanie (np. tworzenie tekstu) z wyszukiwaniem (np. przeszukiwaniem zewnętrznej bazy wiedzy) w celu uzyskania odpowiednich informacji w trakcie generowania tekstu. Szczegółowy opis techniki RAG wykracza poza zakres tej książki. Ponieważ moim celem jest przeszkolenie modeli LLM podobnych do GPT, które uczą się generować tekst po jednym słowie na raz, skupię się na osadzeniach słów.

Do generowania osadzeń słów opracowano kilka algorytmów i frameworków. Jednym z wcześniejszych i najpopularniejszych przykładów jest podejście *Word2Vec*. *Word2Vec* wykorzystuje architekturę sieci neuronowej do generowania osadzeń słów przez przewidywanie kontekstu danego słowa na podstawie słowa docelowego lub odwrotnie (słowa docelowego na podstawie jego kontekstu). Zgodnie z głównym założeniem *Word2Vec* słowa pojawiające się w podobnych kontekstach zwykle mają zbliżone znaczenie. W rezultacie, jak pokazano na rysunku 2.3, po zrzutowaniu na dwuwymiarowe osadzenia słów do celów wizualizacji podobne terminy są pogrupowane.

Osadzenia słów mogą mieć różne wymiary – od jednego do wielu tysięcy. Wyższa wymiarowość pozwala uchwycić bardziej znuansowane relacje, ale kosztem wydajności obliczeniowej.



Rysunek 2.3. Jeśli osadzenia słów są dwuwymiarowe, można w celu wizualizacji wykreślić je na dwuwymiarowym wykresie punktowym tak, jak pokazano na rysunku. W przypadku korzystania z technik osadzania słów, takich jak Word2Vec, słowa odpowiadające podobnym pojęciom często pojawiają się w przestrzeni osadzania blisko siebie. Na przykład różne gatunki ptaków pojawiają się w przestrzeni osadzania bliżej siebie i w oddaleniu od krajów bądź miast

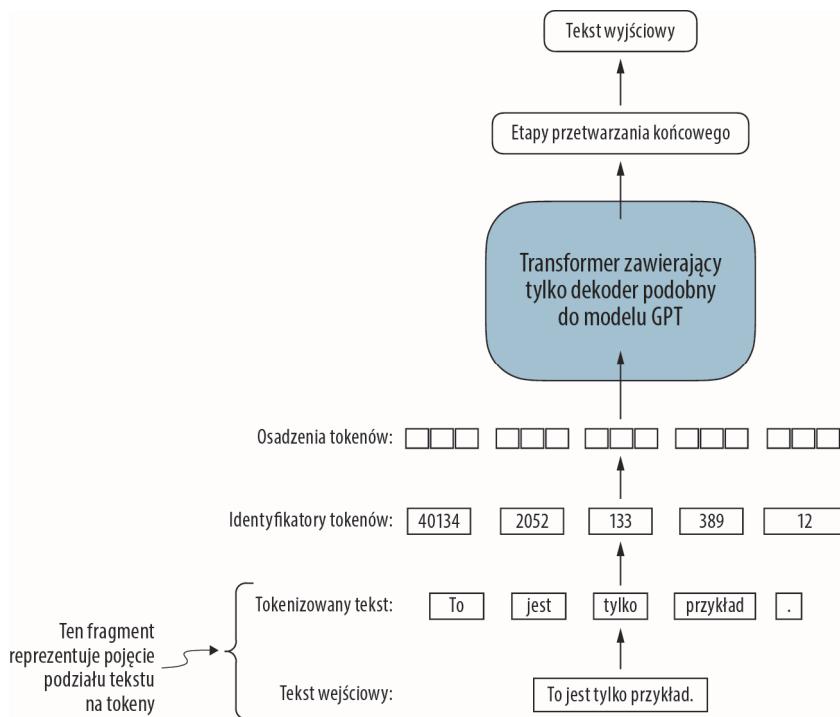
Chociaż do generowania osadzeń na potrzeby modeli uczenia maszynowego można użyć wstępnie przeszkołonych modeli, takich jak Word2Vec, modele LLM zazwyczaj tworzą własne osadzenia, które są częścią warstwy wejściowej i są aktualizowane podczas uczenia. Zaletą optymalizacji osadzeń w ramach szkolenia LLM zamiast korzystania z Word2Vec jest uzyskanie osadzeń zoptymalizowanych pod kątem konkretnego zadania i dostępnych danych. Takie warstwy osadzeń zaimplementuję w dalszej części tego rozdziału (modele LLM mogą również tworzyć kontekstowe osadzenia wyjściowe, o czym opowiem w rozdziale 3.).

Niestety, wielowymiarowe osadzenia są trudne do zwizualizowania, ponieważ ludzka percepcja zmysłowa i typowe reprezentacje w formie grafu z natury są ograniczone do trzech lub mniejszej liczby wymiarów. Z tego powodu na rysunku 2.3 przedstawiono dwuwymiarowe osadzenia na dwuwymiarowym wykresie punktowym. Podczas pracy z modelami LLM zazwyczaj jednak wykorzystuje się osadzenia o znacznie wyższej wymiarowości. Zarówno w przypadku GPT-2, jak i GPT-3 wymiar osadzeń (często określany jako wymiarowość ukrytych stanów modelu) różni się w zależności od konkretnego wariantu i rozmiaru modelu. Jest to kompromis między wydajnością a skutecznością. Dla przykładu, najmniejsze modele GPT-2 (117 mln i 125 mln parametrów) wykorzystują osadzenia o 768 wymiarach. Największy model GPT-3 (175 mld parametrów) wykorzystuje osadzenia o 12 288 wymiarach.

W kolejnym podrozdziale omówię czynności wymagane do przygotowania osadzeń używanych przez model LLM, co obejmuje podział tekstu na słowa, konwertowanie słów na tokeny i przekształcanie tokenów w wektory osadzeń.

2.2. Tokenizacja tekstu

Zastanówmy się, w jaki sposób wejściowy tekst jest dzielony na pojedyncze tokeny. Taki podział jest wymaganym krokiem przetwarzania wstępnego, potrzebnym do tworzenia osadzeń wykorzystywanych przez modele LLM. Tokeny są pojedynczymi słowami lub znakami specjalnymi, w tym znakami interpunkcyjnymi (rysunek 2.4).



Rysunek 2.4. Widok etapów przetwarzania tekstu w kontekście modeli LLM. W tym przypadku dzielimy tekst wejściowy na pojedyncze tokeny reprezentujące słowa lub znaki specjalne, takie jak znaki interpunkcyjne

Tekstem, który poddamy tokenizacji na potrzeby szkolenia LLM, jest opowiadanie *The Verdict* autorstwa Edith Wharton, które opublikowano w domenie publicznej i które w związku z tym można wykorzystywać w zadaniach szkoleniowych LLM. Tekst jest dostępny w serwisie Wikisource pod adresem https://en.wikisource.org/wiki/The_Verdict. Można go skopiować i wkleić do pliku tekstowego. Ja skopiowałem go do pliku tekstowego *the-verdict.txt*¹.

¹ W polskim tłumaczeniu książki użyłem w tym rozdziale tekstu po przetłumaczeniu maszynowym na język polski za pomocą wbudowanej funkcji programu MS Word — *przyp. tłum.*

Plik *the-verdict.txt* możesz również pobrać z repozytorium GitHub tej książki², dostępnego pod adresem <https://mng.bz/Adng>. Aby pobrać plik, możesz skorzystać z następującego kodu w Pythonie:

```
import urllib.request
url = ("https://raw.githubusercontent.com/rasbt/
        "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
        "the-verdict.txt")
file_path = "the-verdict.txt"
urllib.request.urlretrieve(url, file_path)
```

Następnie można załadować plik *the-verdict.txt* z użyciem standardowych narzędzi Pythona do czytania plików (listing 2.1).

Listing 2.1. Wczytywanie krótkiego opowiadania jako próbki tekstu do Pythona

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Łączna liczba znaków:", len(raw_text))
print(raw_text[:99])
```

W celach ilustracyjnych za pomocą polecenia `print` wyświetlamy całkowitą liczbę znaków, a następnie pierwsze 100 znaków tego pliku:

```
Łączna liczba znaków: 20891
Zawsze uważałem Jacka Gisburna za zwykłego geniusza – choć był dość dobrym człowiekiem
→ nie było w
```

Celem w tym przykładzie jest tokenizacja krótkiego opowiadania o objętości 20 891 znaków na pojedyncze słowa i znaki specjalne, które można następnie przekształcić w osadzenia dla celów szkolenia LLM.

UWAGA Podczas pracy z modelami LLM często przetwarzane są miliony artykułów i setki tysięcy książek — wiele gigabajtów tekstu. Jednak w celach edukacyjnych, aby zilustrować główne pojęcia związane z kolejnymi etapami przetwarzania tekstu i uruchomić proces w rozsądny czasie na sprzęcie konsumenckim, wystarczy pracować z mniejszymi próbками tekstu, na przykład z jedną książką.

Jak można najlepiej podzielić ten tekst, aby uzyskać listę tokenów? W tym celu wybierzemy się na małą wycieczkę i żeby zilustrować zagadnienie, wykorzystamy bibliotekę obsługi wyrażeń regularnych w Pythonie `re` (nie musisz uczyć się ani zapamiętywać składni wyrażeń regularnych, ponieważ później użyjemy gotowego tokenizera).

Aby podzielić tekst w miejscach występowania tak zwanych białych znaków w prostym, przykładowym tekście, można użyć polecenia `re.split` o składni zaprezentowanej poniżej:

² Dotyczy wersji oryginalnej — przyp. tłum.

```
import re
text = "Witaj, świecie. To jest test."
result = re.split(r'(\s)', text)
print(result)
```

W wyniku uzyskujemy listę złożoną z pojedynczych słów, białych znaków i znaków interpunkcyjnych:

```
['Witaj', ' ', 'świecie', ' ', 'To', ' ', 'jest', ' ', 'test.']}
```

Ten prosty schemat tokenizacji sprawdza się głównie przy rozdzielaniu przykładowego tekstu na pojedyncze słowa. Niektóre słowa, które powinny być raczej osobnymi pozycjami na liście, są jednak nadal połączone ze znakami interpunkcyjnymi. Powstrzymujemy się również od pisania całego tekstu małymi literami, ponieważ wielkie litery pomagają modelom LLM odróżniać rzeczowniki własne od pospolitych, zrozumieć strukturę zdań i nauczyć się generować tekst o prawidłowej wielkości liter.

Następnie zmodyfikujmy podziały wyrażeń regularnych na białe znaki (\s), przecinki i kropki ([.,.]):

```
result = re.split(r'([.,.]\s)', text)
print(result)
```

Można zauważyć, że słowa i znaki interpunkcyjne są teraz oddzielnymi pozycjami na liście, tak jak chcieliśmy:

```
['Witaj', ' ', ' ', ' ', 'świecie', ' ', ' ', ' ', 'To', ' ', 'jest', ' ', 'test', ' ', ' ']
```

Pozostał jeszcze niewielki problem: otóż lista nadal zawiera niedrukowalne, tak zwane białe znaki. Opcjonalnie można bezpiecznie usunąć te nadmiarowe znaki w następujący sposób:

```
result = [item for item in result if item.strip()]
print(result)
```

Końcowy wynik, bez białych znaków, wygląda następująco:

```
['Witaj', ' ', 'świecie', ' ', 'To', ' ', 'jest', ' ', 'test', ' ']
```

UWAGA To, czy przy tworzeniu prostego tokenizera należy kodować białe znaki jako oddzielne, czy po prostu je usuwać, zależy od aplikacji i jej wymagań. Usunięcie białych znaków zmniejsza zapotrzebowanie na pamięć i wymagania dotyczące mocy obliczeniowej. Zachowanie białych znaków może być jednak korzystne w przypadkach szkolenia modeli wrażliwych na dokładną strukturę tekstu (na przykład kod Pythona, w którym istotne znaczenie mają wcięcia i odstępy). W prezentowanym przykładzie usuwamy białe znaki dla uproszczenia i zwięzłości tokenizowanych wyników. W dalszej części przejdziemy na schemat tokenizacji uwzględniający białe znaki.

Opracowany tutaj schemat tokenizacji dobrze się sprawdza w przetwarzaniu przykładowego, prostego tekstu. Zmodyfikujmy go nieco bardziej, tak aby można było obsługiwać także inne znaki interpunkcyjne, takie jak znaki zapytania, cudzysłowy i podwójne myślniki, a także dodatkowe znaki specjalne, z którymi zetknęliśmy się wcześniej w pierwszych 100 znakach opowiadania Edith Wharton:

```

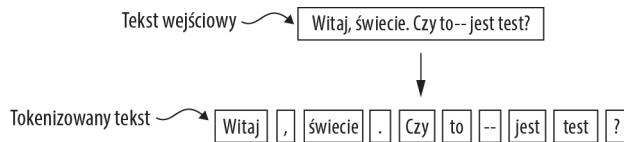
text = "Witaj, świecie. Czy to-- jest test?"
result = re.split(r'([.,.;?_!"]|--)|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)

```

Oto wynik działania tego kodu:

```
[ 'Witaj', ',', 'świecie', '.', 'Czy', 'to', '--', 'jest', 'test', '?' ]
```

Jak widać na podstawie wyników zestawionych na rysunku 2.5, schemat tokenizacji pozwala teraz z powodzeniem obsługiwać różne znaki specjalne w tekście.



Rysunek 2.5. Zaimplementowany do tej pory schemat tokenizacji umożliwia podział tekstu na pojedyncze słowa i znaki interpunkcyjne W tym konkretnym przykładzie tekst został podzielony na 10 osobnych tokenów

Po opracowaniu działającego podstawowego tokenizera zastosujmy go do całego opowiadania Edith Wharton:

```

preprocessed = re.split(r'([.,.;?_!"]|--)|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if
    item.strip()]
print(len(preprocessed))

```

Powyższa instrukcja print wyświetla 3997, co oznacza liczbę tokenów w przykładowym tekście (bez białych znaków). W celu szybkiego sprawdzenia wizualnego spróbujmy wyświetlić pierwsze 30 tokenów:

```
print(preprocessed[:30])
```

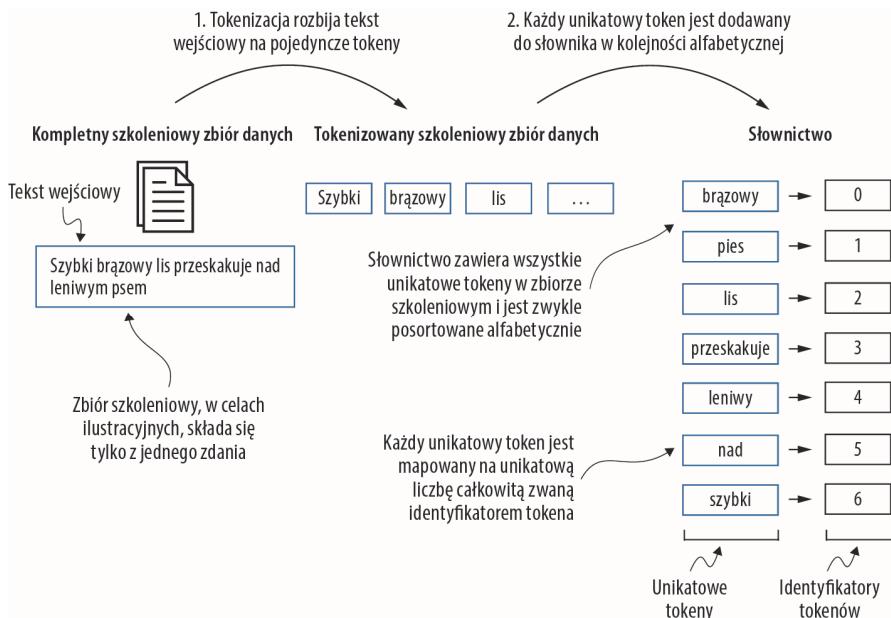
Wynik pokazuje, że wykorzystywany tokenizator dobrze sobie radzi z tekstem. Wszystkie słowa i znaki specjalne są starannie oddzielone:

```
['Zawsze', 'uważałem', 'Jacka', 'Gisburna', 'za', 'zwykłego', 'geniusza', '–', 'choć',
    'był', 'dość', 'dobrym', 'człowiekiem', '–', 'nie', 'było', 'więc', 'dla', 'mnie',
    'wielkim', 'zaskoczeniem', ',', 'gdy', 'usłyszałem', ',', 'że', 'u', 'szczytu',
    'swojej', 'chwałą']
```

2.3. Konwersja tokenów na identyfikatory

W kolejnym kroku dokonamy konwersji tokenów z tekstowej reprezentacji w Pythonie na identyfikatory w postaci liczb całkowitych. Wspomniana konwersja to krok pośredni poprzedzający przekształcenie identyfikatorów tokenów w wektory osadzeń.

Aby zmapować wcześniej wygenerowane tokeny na identyfikatory, trzeba najpierw zbudować słownik. Ten słownik określa sposób mapowania wszystkich słów i znaków specjalnych na unikatowe liczby całkowite (rysunek 2.6).



Rysunek 2.6. Tworzymy słownik przez tokenizację całego tekstu w zbiorze danych szkoleniowych na pojedyncze tokeny. Te pojedyncze tokeny są następnie sortowane alfabetycznie, a duplikaty tokenów są usuwane. Unikatowe tokeny są następnie agregowane do słownika, który definiuje mapowanie z każdego unikatowego tokenu na unikatową liczbę całkowitą. Zaprezentowane słownictwo celowo jest niezbyt obszerne i dla uproszczenia nie zawiera znaków interpunkcyjnych ani specjalnych

Po przeprowadzeniu tokenizacji opowiadania Edith Wharton i przypisaniu tokenów do zmiennej Pythona o nazwie `preprocessed` utworzymy listę wszystkich unikatowych tokenów i w celu określenia rozmiaru słownictwa posortujemy ją alfabetycznie:

```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)
print(vocab_size)
```

Po ustaleniu za pomocą tego kodu, że rozmiar słownika wynosi 1534, tworzymy słownik i w celach ilustracyjnych wyświetlamy jego pierwszych 51 pozycji (listing 2.2).

Listing 2.2. Tworzenie słownictwa

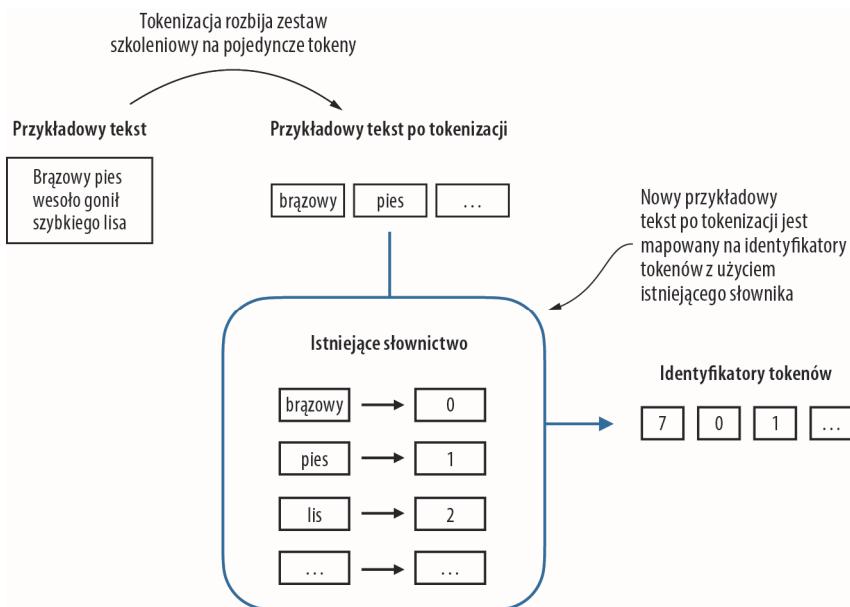
```
vocab = {token:integer for integer,token in enumerate(all_words)}
for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break
```

Oto wynik:

```
('!', 0)
('!', 1)
```

```
('"', 2)
...
('Gideon', 49)
('Gisburn', 50)
```

Jak można zauważyć, słownik zawiera pojedyncze tokeny powiązane z unikatowymi etykietami w postaci liczb całkowitych. Kolejnym celem jest zastosowanie tego słownika do konwersji nowego tekstu na identyfikatory (rysunek 2.7).



Rysunek 2.7. Począwszy od nowej próbki tekstu, tokenizujemy tekst i używamy słownictwa do konwersji tekstowych tokenów na liczbowe identyfikatory. Słownictwo jest budowane na podstawie całego zestawu szkoleniowego i można je zastosować do samego zestawu szkoleniowego oraz dowolnych nowych próbek tekstu. Zaprezentowane słownictwo dla uproszczenia nie zawiera znaków interpunkcyjnych ani specjalnych

Do zamiany danych wyjściowych modelu LLM z liczb na tekst potrzebny jest sposób na przekształcenie identyfikatorów tokenów na tekst. W tym celu można utworzyć odwrotną wersję słownika, która mapuje identyfikatory tokenów na odpowiadające im tokeny tekstowe.

Spróbujmy zaimplementować kompletną klasę tokenizera w Pythonie z metodą `encode`, która dzieli tekst na tokeny i w celu uzyskania identyfikatorów tokenów prowadzącą za pomocą słownika mapowanie ciągu znaków na liczbę całkowitą. Ponadto zaimplementujmy metodę `decode`, która prowadzącą odwrotną mapowanie liczb całkowitych na ciągi znaków w celu skonwertowania identyfikatorów tokenów z powrotem na tekst. Kod implementacji tokenizera zamieściłem na listingu 2.3.

Listing 2.3. Implementacja prostego tokenizera tekstu

```
Zapisuje słownik jako atrybut klasy do wykorzystania
w metodach encode i decode
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([.,.?_!"()\']|--|\s)', text)
        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        text = re.sub(r'\s+([.,?!"\'])', r'\1', text)
        return text
```

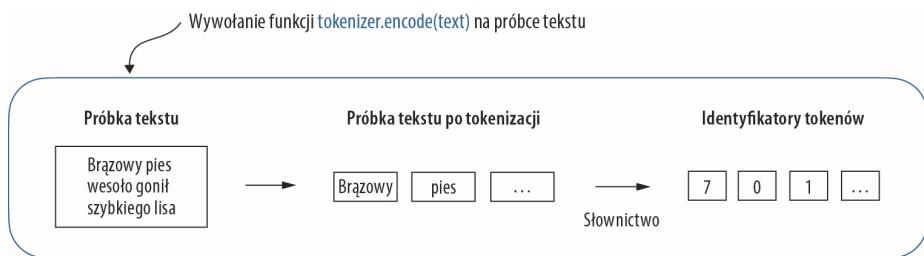
Tworzy odwrotny słownik, odwzorowujący identyfikatory tokenów na oryginalne tokeny tekstowe

Przetwarza tekst wejściowy na identyfikatory tokenów

Konwertuje identyfikatory tokenów na tekst

Usuwa spacje przed określonymi znakami interpunkcyjnymi

Korzystając z klasy Pythona SimpleTokenizerV1, można za pomocą istniejącego słownika utworzyć egzemplarze nowych obiektów tokenizera. Następnie można wykorzystać te obiekty do kodowania i dekodowania tekstu w sposób pokazany na rysunku 2.8.



Rysunek 2.8. Implementacje tokenizerów mają dwie wspólne metody: metodę encode i metodę decode. Metoda encode pobiera przykładowy tekst, dzieli go na pojedyncze tokeny i za pomocą słownika konwertuje tokeny na identyfikatory. Metoda decode pobiera identyfikatory tokenów, konwertuje je na tokeny tekstowe i łączy tokeny tekstowe w naturalny tekst

Utworzymy teraz egzemplarz nowego obiektu tokenizera klasy SimpleTokenizerV1 i aby wypróbować go w praktyce, wykonamy tokenizację fragmentu opowiadania Edith Wharton:

```
tokenizer = SimpleTokenizerV1(vocab)
text = """Wiesz, to ostatni, który namalował"""
       powiedziała pani Gisburn z wybaczalną dumą."""
ids = tokenizer.encode(text)
print(ids)
```

Powyższy kod wyświetla następujące identyfikatory tokenów:

```
[1, 169, 5, 1207, 798, 5, 544, 669, 1, 905, 812, 50, 1382, 1331, 385, 7]
```

Następnie sprawdźmy, czy za pomocą metody decode można przekształcić te identyfikatory tokenów z powrotem na tekst:

```
print(tokenizer.decode(ids))
```

Oto wynik:

```
" Wiesz, to ostatni, który namalował" powiedziała pani Gisburn z wybaczalną dumą.
```

Na podstawie tych danych wyjściowych można stwierdzić, że metoda decode pomyślnie przekonwertowała identyfikatory tokenów na oryginalny tekst.

Jak dotąd wszystko przebiega dobrze. Zaimplementowaliśmy tokenizer zdolny do tokenizacji i detokenizacji tekstu na podstawie fragmentu z zestawu szkoleniowego. Zastosujmy go teraz do nowej próbki tekstu, której nie ma w zbiorze szkoleniowym:

```
text = "Witaj, czy lubisz herbatę?"
print(tokenizer.encode(text))
```

Wykonanie tego kodu spowoduje wyświetlenie następującego błędu:

```
KeyError: 'Witaj'
```

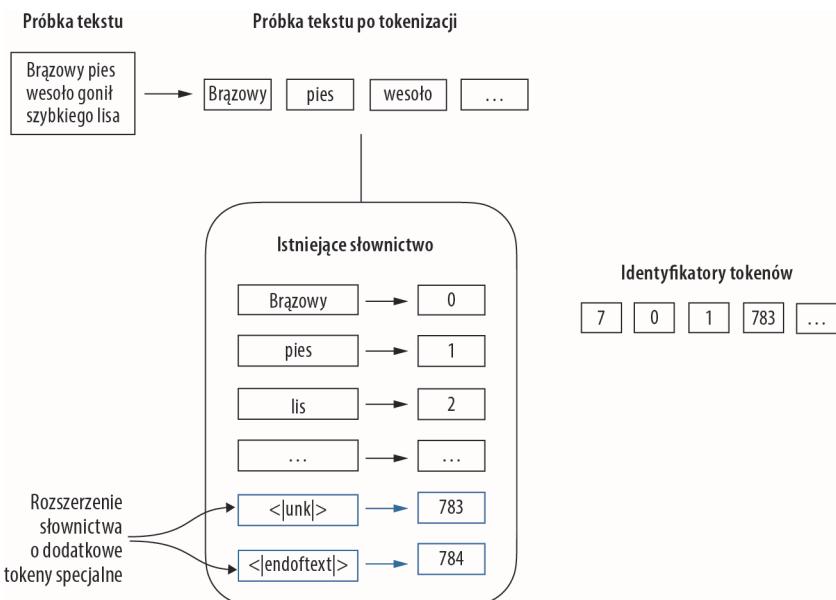
Problem polega na tym, że w przykładowym opowiadaniu nie ma słowa „Witaj”. Z tego powodu nie ma go w słowniku. Podkreśla to potrzebę uwzględnienia podczas pracy nad modelami LLM dużych i zróżnicowanych zestawów szkoleniowych w celu rozszerzenia słownictwa.

W kolejnym podrozdziale przetestujemy tokenizer na tekście zawierającym nieznane słowa. Przy tej okazji opiszę dodatkowe, specjalne tokeny, które można wykorzystać podczas szkolenia w celu dostarczenia dodatkowego kontekstu dla modelu LLM.

2.4. Dodawanie specjalnych tokenów kontekstowych

Trzeba zmodyfikować tokenizer, aby obsługiwał nieznane słowa. Trzeba również zająć się wykorzystaniem i dodaniem specjalnych tokenów kontekstowych, które mogą poprawić zrozumienie przez model kontekstu lub innych istotnych informacji

w tekście. Te specjalne tokeny mogą zawierać na przykład znaczniki nieznanych słów i granice dokumentów. W szczególności zmodyfikujemy słownik i tokenizer. W efekcie stworzymy tokenizer SimpleTokenizerV2, aby obsługiwał dwa nowe tokeny: `<|unk|>` i `<|endoftext|>`, tak jak pokazałem na rysunku 2.9.



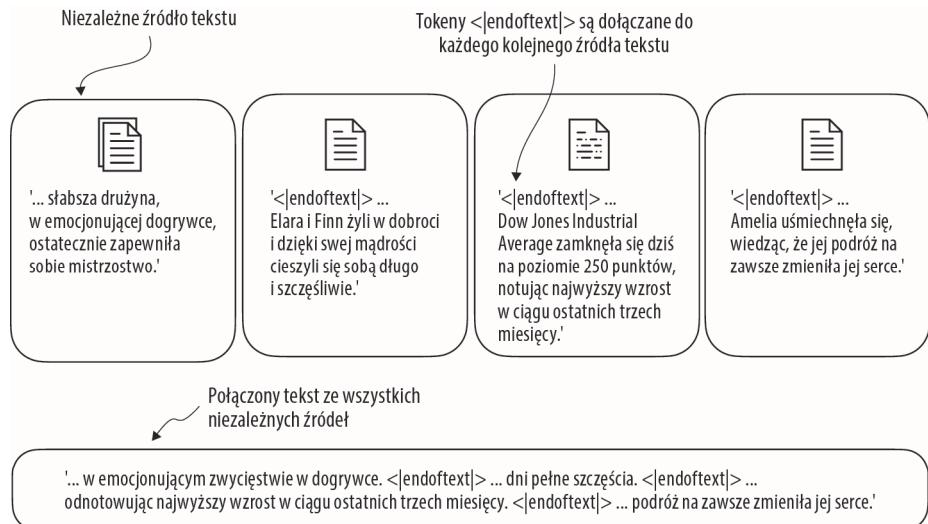
Rysunek 2.9. Dodajemy do słownika specjalne tokeny, aby obsłużyć określone konteksty. Na przykład dodajemy token `<|unk|>` w celu reprezentowania nowych i nieznanych słów, które nie były częścią danych szkoleniowych, a tym samym nie były częścią istniejącego słownictwa. Ponadto dodajemy token `<|endoftext|>`, którego można użyć do oddzielenia dwóch niepowiązanych ze sobą źródeł tekstu

Można zmodyfikować tokenizer w taki sposób, aby w razie napotkania słowa, które nie jest częścią słownika, używał tokenu `<|unk|>`. Ponadto dodajemy token między niepowiązanymi ze sobą tekstami. Na przykład podczas szkolenia modelu LLM podobnego do GPT na wielu niezależnych dokumentach lub książkach przed każdym dokumentem lub książką, które następują po poprzednim źródle tekstu, często wstawia się token (rysunek 2.10). Pomaga to modelowi LLM zrozumieć, że chociaż te źródła tekstu są połączone w celu szkolenia, to w gruncie rzeczy nie są ze sobą powiązane.

Zmodyfikujmy teraz słownictwo, aby uwzględnić te dwa specjalne tokeny: `<unk|>` i `<|endoftext|>`. W tym celu dodamy je do listy wszystkich unikatowych słów:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)}

print(len(vocab.items()))
```



Rysunek 2.10. Podczas pracy z wieloma niezależnymi źródłami tekstu pomiędzy tymi tekstami dodajemy tokeny <|endoftext|>. Tokeny <|endoftext|> pełnią funkcję znaczników sygnalizujących początek lub koniec danego segmentu, co pozwala modelowi LLM przetwarzać tekst efektywniej i lepiej go zrozumieć

Na podstawie danych wyjściowych tej instrukcji print nowy rozmiar słownika wynosi 1132 (poprzedni rozmiar słownika wynosił 1130).

W ramach dodatkowego szybkiego sprawdzenia spróbujmy wyświetlić pięć ostatnich wpisów zaktualizowanego słownika:

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

Oto wynik działania tego kodu:

```
('żądanie', 1531)
('–', 1532)
('–', 1533)
('<|endoftext|>', 1534)
('<|unk|>', 1535)
```

Na podstawie wyniku działania kodu można potwierdzić, że dwa nowe tokeny specjalne zostały pomyślnie włączone do słownika. Następnie odpowiednio dostosujemy tokener z listingu 2.3, jak pokazano na listingu 2.4.

Listing 2.4. Prosty tokenizer tekstu obsługujący nieznane słowa

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items() }
```

```

def encode(self, text):
    preprocessed = re.split(r'([,:;?!"]|--)|\s', text)
    preprocessed = [
        item.strip() for item in preprocessed if item.strip()
    ]
    preprocessed = [item if item in self.str_to_int      ← Zastępuje nieznane
                    else "<|unk|>" for item in preprocessed] ← słowa tokenami <|unk|>
    ids = [self.str_to_int[s] for s in preprocessed]
    return ids

def decode(self, ids):
    text = " ".join([self.int_to_str[i] for i in ids]) ← Usuwa spacje przed
    text = re.sub(r'\s+([,:;?!"]|--)', r'\1', text) ← określonymi znakami
    return text

```

W porównaniu z klasą SimpleTokenizerV1, której implementacja jest pokazana na listingu 2.3, nowa klasa SimpleTokenizerV2 zastępuje nieznane słowa tokenami <|unk|>.

Wypróbujmy teraz ten nowy tokenizer w praktyce. W tym celu użyjemy prostej próbki tekstu, którą scalimy z dwóch niezależnych i niepowiązanych ze sobą zdań:

```

text1 = "Witaj, czy lubisz herbatę?"
text2 = "Wyszedł na skąpany w słońcu taras pałacu"
text = " <|endoftext|> ".join((text1, text2))
print(text)

```

Oto wynik:

Witaj, czy lubisz herbatę? <|endoftext|> Wyszedł na skąpany w słońcu taras pałacu.

Następnie dokonamy tokenizacji przykładowego tekstu z użyciem klasy SimpleTokenizerV2 oraz słownika, który utworzyliśmy wcześniej za pomocą kodu pokazanego na listingu 2.2:

```

tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))

```

Spowoduje to wyświetlenie następujących identyfikatorów tokenów:

[1535, 5, 326, 572, 1535, 10, 1534, 1535, 653, 1091, 1279, 1176, 1193, 1535]

Można zauważyc, że lista identyfikatorów tokenów zawiera 1534 dla tokena separatora <|endoftext|>, a także kilka tokenów 1535, które są używane do słów nieznanych.

Aby sprawdzić poprawność działania tokenizera, spróbujmy zdetokenizować tekst:

```
print(tokenizer.decode(tokenizer.encode(text)))
```

Oto wynik:

<|unk|>, czy lubisz <|unk|>? <|endoftext|> <|unk|> na skąpany w słońcu taras <|unk|>

Na podstawie porównania tego zdetokenizowanego tekstu z tekstem wejściowym możemy stwierdzić, że szkoleniowy zbiór danych — opowiadanie Edith Wharton *The Verdict*, nie zawiera słów „Witaj”, „herbatę”, „Wyszedł” i „pałacu”.

W zależności od modelu LLM, niektórzy badacze zalecają również zastosowanie dodatkowych specjalnych tokenów, np:

- [BOS] (ang. *beginning of sequence* – dosłownie: początek sekwencji) – ten token oznacza początek tekstu. Jest to informacja dla modelu LLM, gdzie zaczyna się fragment treści.
- [EOS] (ang. *end of sequence* – koniec sekwencji) – ten token jest umieszczany na końcu tekstu i jest szczególnie przydatny podczas łączenia wielu niepowiązanych tekstów, podobnie do <| endoftext |>. Na przykład w przypadku łączenia dwóch różnych książek lub artykułów z Wikipedii token [EOS] wskazuje, gdzie kończy się jeden dokument, a gdzie zaczyna następny.
- [PAD] (ang. *padding* – uzupełnienie) – podczas szkolenia modelu LLM z rozmiarami partii większymi niż jeden partia może zawierać teksty o różnej długości. Aby zapewniona była równa długość wszystkich tekstów, krótsze teksty są przedłużane, czyli „uzupełniane” za pomocą tokena [PAD] do długości najdłuższego tekstu w partii.

Tokenizer używany w modelach GPT nie potrzebuje żadnego z tych tokenów; używa on, dla uproszczenia, jedynie tokena <| endoftext |>. Token <| endoftext |> jest analogiczny do tokena [EOS]. Token <| endoftext |> jest również używany do wypełniania. Jednak jak opowiem w kolejnych rozdziałach, podczas szkolenia na danych wejściowych przesyłanych partiami zwykle używa się masek. Z tego powodu nie ma potrzeby zwracania uwagi na tokeny uzupełnień. W związku z tym konkretny token wybrany do uzupełniania nie ma znaczenia.

Co więcej, tokenizer stosowany w modelach GPT nie używa również tokena <| unk |> w odniesieniu do słów spoza słownictwa. Zamiast tego modele GPT wykorzystują tokenizer *kodujący pary bajtów*, który rozbija słowa na jednostki podsłów, co omówię w następnym podrozdziale.

2.5. Kodowanie par bajtów

Przyjrzyjmy się bardziej zaawansowanemu schematowi tokenizacji, opartemu na pojęciu zwanym kodowaniem par bajtów (ang. *byte pair encoding* – BPE). Tokenizera BPE użyto do szkolenia takich modeli LLM jak GPT-2, GPT-3 i oryginalny model stosowany w systemie ChatGPT.

Ponieważ implementacja BPE może być stosunkowo złożona, wykorzystamy istniejącą bibliotekę open source Pythona o nazwie `tiktoken` (<https://github.com/openai/tiktoken>), która bardzo wydajnie implementuje algorytm BPE na podstawie źródła w języku Rust. Podobnie jak w przypadku innych bibliotek Pythona, bibliotekę `tiktoken` można zainstalować w terminalu za pomocą instalatora `pip` Pythona:

```
pip install tiktoken
```

Zaprezentowany poniżej kod korzysta z biblioteki `tiktoken` 0.8.0. Aby sprawdzić zainstalowaną wersję, można użyć następującej instrukcji:

```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

Po zainstalowaniu biblioteki egzemplarz tokenizera BPE z biblioteki `tiktoken` można utworzyć w następujący sposób:

```
tokenizer = tiktoken.get_encoding("gpt2")
```

Sposób użycia tego tokenizera przypomina użycie zaimplementowanego wcześniej tokenizera `SimpleTokenizerV2` — należy zastosować metodę `encode`:

```
text = (
    "Cześć, lubisz herbatę? <|endoftext|> Na skąpany w słońcu taras "
    "jakiegoś nieznanegoMiejsca."
)
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
```

Wykonanie powyższego kodu powoduje wyświetlenie następujących identyfikatorów tokenów:

```
[34, 2736, 129, 249, 38325, 11, 300, 46676, 89, 607, 8664, 128, 247, 30, 220, 50256,
→11013, 1341, 128, 227, 79, 1092, 266, 264, 41615, 78, 129, 226, 27399, 13422, 292,
→73, 461, 494, 2188, 129, 249, 299, 494, 47347, 1531, 2188, 44, 494, 73, 1416, 64, 13]
```

Następnie można przekształcić te identyfikatory tokenów na tekst za pomocą metody `decode` — podobnie do sposobu używanego z tokenizatorem `SimpleTokenizerV2`:

```
strings = tokenizer.decode(integers)
print(strings)
```

Oto wynik działania tego kodu:

```
Cześć, lubisz herbatę? <|endoftext|> Na skąpany w słońcu taras jakiegoś
→nieznanegoMiejsca.
```

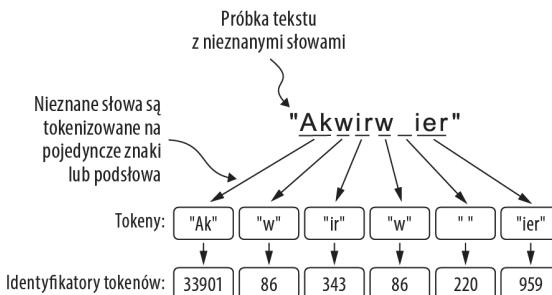
Na podstawie identyfikatorów tokenów i zdekodowanego tekstu można zaobserwować dwie rzeczy:

Po pierwsze, token `<|endoftext|>` ma przypisaną stosunkowo wysoką wartość identyfikatora tokena, mianowicie 50256. Tokenizer BPE, którego używano do szkolenia takich modeli jak GPT-2, GPT-3 i oryginalny model stosowany w systemie ChatGPT, ma całkowity rozmiar słownictwa 50 257, a token `<|endoftext|>` ma przypisany największy identyfikator.

Po drugie, tokenizer BPE poprawnie koduje i dekoduje nieznane słowa, takie jak `nieznanegoMiejsca`. Tokenizer BPE pozwala obsłużyć wszystkie nieznane słowa. Jak to jest możliwe bez użycia tokenów `<|unk|>`?

Algorytm, na którym opiera się BPE, rozbija słowa, których nie ma w jego predefiniowanym słowniku, na mniejsze jednostki — podsłowa lub nawet na pojedyncze

znaki. Dzięki temu możliwa jest obsługa słów spoza słownika. Tak więc dzięki algorytmowi BPE, jeśli podczas tokenizacji tokenizer napotka nieznane słowo, może zaprezentować je jako sekwencję tokenów lub znaków podsłowa (rysunek 2.11).



Rysunek 2.11. Tokenizery BPE dzielą nieznane słowa na podsłowa i pojedyncze znaki. Dzięki temu tokenizer BPE może sparsować dowolne słowo i nie trzeba zastępować nieznanych słów specjalnymi tokenami, takimi jak <|unk|>

Dzięki zdolności do dzielenia nieznanych słów na pojedyncze znaki tokenizer, a w konsekwencji model LLM, który jest szkolony z jego użyciem, może przetwarzać dowolny tekst, nawet wtedy, gdy zawiera on słowa, które nie występują w danych szkoleniowych.

Ćwiczenie 2.1. Kodowanie par bajtów nieznanych słów

Wypróbuje tokenizer BPE z biblioteki `tiktoken` na nieznanych słowach „Akwirw ier” i wyświetli identyfikatory poszczególnych tokenów. Następnie, aby odtworzyć mapowanie pokazane na rysunku 2.11, wywołaj funkcję `decode` na każdej z liczb całkowitych z tej listy. Na koniec, aby sprawdzić, czy można odtworzyć oryginalne dane wejściowe, czyli słowa „Akwirw ier”, wywołaj metodę `decode` na identyfikatorach tokenów.

Szczegółowy opis i implementacja tokenizera BPE wykraczają poza zakres tej książki. W skrócie — tokenizer BPE buduje słownictwo przez iteracyjne łączenie częstych znaków w podsłowa i częstych podłów w słowa. Na przykład tokenizer BPE zaczyna od dodania do swojego słownika wszystkich pojedynczych znaków ('a', 'b' itp.). W kolejnym etapie łączy kombinacje znaków, które często występują razem, w podwyrazy. Na przykład znaki „c” i „z” mogą zostać połączone w podwyraz „cz”, który często występuje w wielu polskich słowach, takich jak „czwartek”, „czysta”, „paczka” czy „uczyć”. Połączenia są ustalane na podstawie progu częstotliwości (ang. *frequency cutoff*).

2.6. Próbkowanie danych z oknem przesuwnym

Kolejny krok w tworzeniu osadzeń na potrzeby modelu LLM polega na wygenerowaniu wymaganych do szkolenia modelu LLM par wejście-cel. Jak wyglądają pary wejście-cel? Jak już się dowiedziałeś, modele LLM są wstępnie szkolone przez przewidywanie następnego słowa w tekście (rysunek 2.12).



Rysunek 2.12. Na podstawie próbki tekstu wyodrębniane są bloki wejściowe jako podpróbki pełniące funkcję danych wejściowych modelu LLM, a zadaniem predykcyjnym modelu LLM podczas szkolenia jest przewidywanie następnego słowa, które występuje po bloku wejściowym. Podczas szkolenia wszystkie słowa, które znajdują się za słowem docelowym, są maskowane. Należy zauważyć, że tekst pokazany na tym rysunku, zanim model LLM będzie mógł go przetworzyć, musi zostać poddany tokenizacji. Na tym rysunku dla zachowania czytelności etap tokenizacji pominięto

Spróbujmy zaimplementować komponent ładowający dane, który pobiera ze zbioru danych szkoleniowych przedstawione na rysunku 2.12 pary wejście-cel metodą okna przesuwnego. Na początek dokonamy tokenizacji całego opowiadania *The Verdict* z użyciem tokenizera BPE:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

Wykonanie tego kodu zwróci wartość 11 345 – całkowitą liczbę tokenów w zbiorze szkoleniowym po zastosowaniu tokenizera BPE.

Następnie w celach demonstracyjnych usuwamy ze zbioru danych pierwsze 50 tokenów, ponieważ dzięki temu na potrzeby kolejnych kroków uzyskamy nieco bardziej interesujący fragment tekstu:

```
enc_sample = enc_text[50:]
```

Jednym z najprostszych i najbardziej intuicyjnych sposobów tworzenia par wejście-cel dla zadania przewidywania następnego słowa jest utworzenie dwóch zmiennych, x i y , gdzie x zawiera tokeny wejściowe, a y zawiera cele, które reprezentują wejścia przesunięte o 1:

```
context_size = 4
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y: {y}")
```

Rozmiar kontekstu określa liczbę tokenów w wejściu.

Uruchomienie powyższego kodu spowoduje wyświetlenie następującego wyniku:

```
x: [78, 45967, 128, 247]
y: [45967, 128, 247, 66]
```

Przetwarzanie wejść razem z celami, czyli wejściami przesuniętymi o jedną pozycję, pozwala utworzyć zadania przewidywania następnego słowa (patrz rysunek 2.12) w następujący sposób:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

Oto wynik działania tego kodu:

```
[78] ----> 45967
[78, 45967] ----> 128
[78, 45967, 128] ----> 247
[78, 45967, 128, 247] ----> 66
```

Wszystko po lewej stronie strzałki (---->) odnosi się do danych wejściowych, które otrzymuje model LLM, natomiast identyfikator tokena po prawej stronie strzałki reprezentuje docelowy token, który model LLM ma przewidzieć. Powtórzmy poprzedni kod. Teraz jednak skonwertujemy identyfikatory tokenów na tekst:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

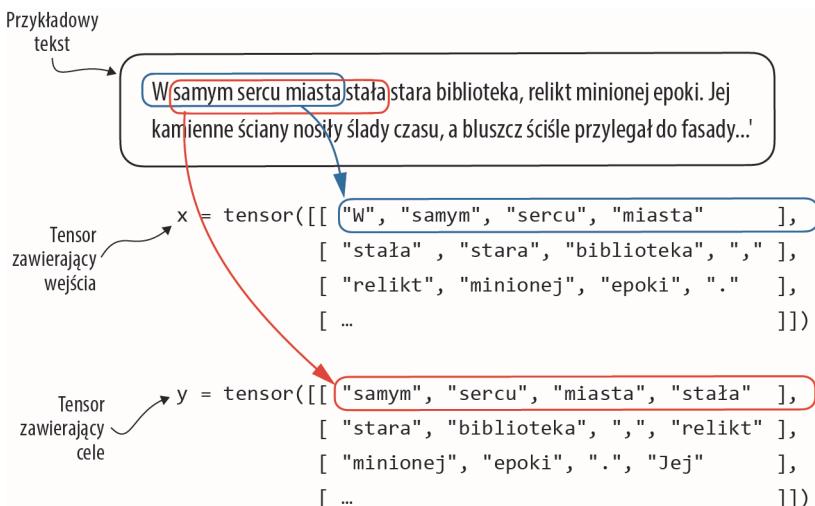
Poniższe wyniki pokazują, jak dane wejściowe i wyjściowe wyglądają w formacie tekstowym:

```
o ----> wi
o wi ---->
o wi ---->
o wię ----> c
```

Utworzyliśmy teraz pary wejście-cel, które można wykorzystać do szkolenia modelu LLM.

Przed przekształceniem tokenów w osadzenia pozostało jeszcze jedno zadanie: implementacja wydajnego komponentu ładującego dane, który iteruje po wejściowym zbiorze danych i zwraca dane wejściowe i docelowe jako tensory PyTorch, które można

traktować jako wielowymiarowe tablice. Celem jest uzyskanie dwóch tensorów: tensora wejściowego, reprezentującego tekst przetwarzany przez model LLM, i tensora docelowego, który zawiera wartości przewidywane przez model LLM (rysunek 2.13). Dla celów ilustracyjnych tokeny na rysunku przedstawiono w formacie tekstowym. W implementacji kodu operujemy jednak bezpośrednio na identyfikatorach tokenów. Wynika to z faktu, że metoda kodowania tokenizera BPE w jednym kroku wykonuje zarówno tokenizację, jak i konwersję tokenów na identyfikatory.



Rysunek 2.13. Aby zaimplementować wydajne komponenty ładujące dane, zbieramy dane wejściowe w tensorze x , w którym każdy wiersz reprezentuje jeden kontekst wejściowy. Drugi tensor, y , zawiera odpowiednie cele predykcji (następne słowa), tworzone przez przesunięcie danych wejściowych o jedną pozycję

UWAGA Do stworzenia wydajnej implementacji komponentu ładującego dane wykorzystamy wbudowane w PyTorch klasy Dataset i DataLoader. Dodatkowe informacje i wskaźniki dotyczące instalacji PyTorch można znaleźć w podpunkcie A.2.1.3, w „Dodatku A”.

Kod klasy dataset zamieściłem na poniższym listingu (listing 2.5).

Listing 2.5. Zbiór danych dla podzielonych na partie danych wejściowych i docelowych

```
import torch
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt) ← Tokenizuje cały tekst
```

```

for i in range(0, len(token_ids) - max_length, stride): ←
    input_chunk = token_ids[i:i + max_length]
    target_chunk = token_ids[i + 1: i + max_length + 1]
    self.input_ids.append(torch.tensor(input_chunk))
    self.target_ids.append(torch.tensor(target_chunk))

def __len__(self): ← Zwraca całkowitą liczbę wierszy w zbiorze danych
    return len(self.input_ids)

def __getitem__(self, idx): ← Zwraca pojedynczy wiersz
    return self.input_ids[idx], self.target_ids[idx]

```

Używa okna przesuwnego do podzielenia książki na nakładające się na siebie sekwencje o maksymalnej długości

Zwraca pojedynczy wiersz ze zbioru danych

Klasa GPTDatasetV1 opiera się na klasie Dataset z biblioteki PyTorch i określa sposób pobierania poszczególnych wierszy ze zbioru danych, przy czym każdy wiersz składa się ze zbioru identyfikatorów tokenów (na podstawie wartości `max_length`) przypisanych do tensora `input_chunk`. Tensor `target_chunk` zawiera cele odpowiadające wejściom. W dalszej części rozdziału pokażę, jak wyglądają dane zwrócone z tego zbioru danych po połączeniu zbioru danych z klasą DataLoader z biblioteki PyTorch — dzięki temu uzyskamy dodatkowy obraz i poprawimy czytelność przykładu.

UWAGA Czytelników nieznających zbyt dobrze struktury klas Dataset biblioteki PyTorch, takich jak te pokazane na listingu 2.5, zachęcam do zapoznania się z podrozdziałem A.6 w „Dodatku A”, gdzie wyjaśnitem ogólną strukturę i sposób użycia klas Dataset i DataLoader z biblioteki PyTorch.

W kodzie pokazanym na listingu 2.6 do załadowania danych wejściowych partiami za pomocą komponentu DataLoader framework PyTorch wykorzystałem klasę `GPTDatasetV1`.

Listing 2.6. Komponent ładujący dane do generowania partii par wejście-cele

```

def create_dataloader_v1(txt, batch_size=4, max_length=256,
                        stride=128, shuffle=True, drop_last=True,
                        num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2") ← Inicjalizuje tokenizer
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride) ← Tworzy zbiór danych
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last, ← True odrzuca ostatnią partię, jeśli jest krótsza niż określony rozmiar
                            partii. Ma to przeciwodzielić skokowej zmianie wartości funkcji straty
                            podczas szkolenia
        num_workers=num_workers ← Liczba procesów CPU używanych
                                do wstępnego przetwarzania
    )
    return dataloader

```

Aby uzyskać intuicyjny obraz sposobu, w jaki współpracują ze sobą klasa `GPTDatasetV1` z listingu 2.5 i funkcja `create_dataloader_v1` z listingu 2.6, przetestujmy obiekt ładujący dane dla partii o rozmiarze 1 dla modelu LLM, w którym rozmiar kontekstu wynosi 4:

```

with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False)
data_iter = iter(dataloader) ← Konwertuje obiekt dataloader na iterator Pythona, aby pobrać
first_batch = next(data_iter)   następny wpis za pomocą wbudowanej funkcji Pythona next()
print(first_batch)

```

Wykonanie powyższego kodu powoduje wyświetlenie następujących wyników:

```
[tensor([[ 57, 8356, 2736, 334]]), tensor([[ 8356, 2736, 334, 10247]])]
```

Zmienna `first_batch` zawiera dwa tensory: pierwszy przechowuje identyfikatory tokenów wejściowych, natomiast drugi przechowuje identyfikatory tokenów docelowych. Ponieważ zmienna `max_length` jest ustawiona na 4, każdy z dwóch tensorów zawiera cztery identyfikatory tokenów. Należy zauważać, że rozmiar wejścia wynoszący 4 jest dość mały i wybrano go jedynie dla uproszczenia. Podczas szkolenia modeli LLM rozmiar danych wejściowych wynosi co najmniej 256.

Aby zrozumieć znaczenie ustawienia `stride=1`, spróbujmy pobrać z tego zbioru danych kolejną partię:

```

second_batch = next(data_iter)
print(second_batch)

```

Druga partia ma następującą zawartość:

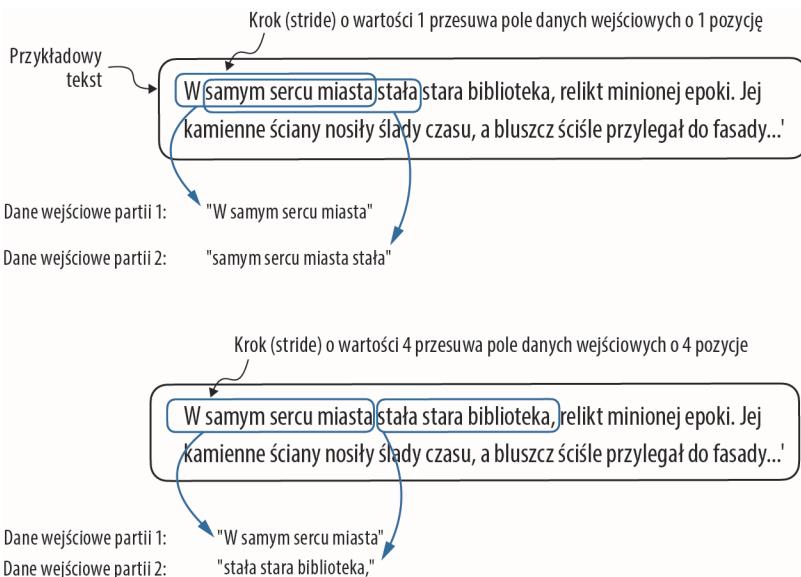
```
[tensor([[ 8356, 2736, 334, 10247]]), tensor([[ 2736, 334, 10247, 129]])]
```

Podczas porównywania pierwszej partii z drugą można zauważać, że identyfikatory tokenów drugiej partii są przesunięte o jedną pozycję (na przykład drugi identyfikator w danych wejściowych pierwszej partii, 8356, jest pierwszym identyfikatorem danych wejściowych drugiej partii). Ustawienie `stride` określa liczbę pozycji, o jaką przesuwają się dane wejściowe pomiędzy partiami. W ten sposób naśladujemy podejście okna przesuwnego (rysunek 2.14).

Ćwiczenie 2.2. Komponenty ładujące dane z różnymi krokami i rozmiarami kontekstu

Aby lepiej zrozumieć, jak działa komponent ładujący dane, spróbuj uruchomić go z różnymi ustawieniami, takimi jak `max_length=2` i `stride=2` oraz `max_length=8` i `stride=2`.

Rozmiary partii wynoszące 1, takie jak te, które stosowaliśmy do tej pory w komponencie ładującym dane, są przydatne do celów ilustracyjnych. Czytelnicy, którzy mają doświadczenie z uczeniem głębokim, wiedzą, że partie o małych rozmiarach wymagają podczas uczenia mniej pamięci, ale ich użycie sprawia, że wersje modelu są bardziej zaszumione. Podobnie jak w zwykłym uczeniu głębokim, rozmiar partii jest hiperparametrem, z którym można eksperymentować w trakcie uczenia modelu LLM.



Rysunek 2.14. Podczas tworzenia wielu partii ze zbioru danych wejściowych przesuwamy po tekście okno. Jeśli parametr stride jest ustawiony na 1, to przy tworzeniu następnej partii przesuwamy okno wejściowe o jedną pozycję. Dzięki ustawieniu parametru stride na wartość równą rozmiarowi okna wejściowego można zapobiec nakładaniu się partii na siebie

Przyjrzyjmy się, w jaki sposób można użyć komponentu ładującego dane do próbkiowania danych o rozmiarze partii większym niż 1:

```
data_loader = create_data_loader_v1(
    raw_text, batch_size=8, max_length=4, stride=4,
    shuffle=False
)

data_iter = iter(data_loader)
inputs, targets = next(data_iter)
print("Wejścia:\n", inputs)
print("\nCele:\n", targets)
```

Uruchomienie tego kodu spowoduje wyświetlenie następujących wyników:

```
Wejścia:
tensor([[ 57,  8356,  2736,   334],
       [10247,   129,   120,    64],
       [41615,   368,  3619,    64],
       [ 402,   271, 10899,    64],
       [ 1976,    64,  1976, 21768],
       [   74, 41615,  1533,    78],
       [15632,  4496,   784,  1727],
       [38325,    416, 41615,   466]])
```

```
Cele:
tensor([[ 8356,  2736,   334, 10247],
       [ 129,   120,    64, 41615],
```

```
[ 368, 3619,   64,   402],
[ 271, 10899,   64, 1976],
[ 64, 1976, 21768,   74],
[41615, 1533,   78, 15632],
[ 4496,   784, 1727, 38325],
[ 416, 41615,   466, 129]])
```

Zwróćmy uwagę, że aby w pełni wykorzystać zbiór danych, zwiększymy krok do 4 (nie pomijamy ani jednego słowa). W ten sposób można uniknąć nakładania się partii, ponieważ większe nakładanie się mogłyby prowadzić do zwiększonego ryzyka nadmiernego dopasowania.

2.7. Tworzenie osadzeń tokenów

Ostatni krok podczas przygotowywania tekstu wejściowego do uczenia modelu LLM polega na przekształceniu identyfikatorów tokenów w wektory osadzeń tak, jak pokazano na rysunku 2.15. W ramach wstępniego kroku trzeba zainicjować wagi osadzeń losowymi wartościami. Ta inicjalizacja jest punktem wyjścia dla procesu szkolenia modelu LLM. W rozdziale 5. pokażę, jak zoptymalizować wagi osadzeń w ramach szkolenia modelu LLM.

Ciągła reprezentacja wektorowa lub osadzanie są konieczne, ponieważ modele LLM podobne do GPT są głębokimi sieciami neuronowymi szkolonymi za pomocą algorytmu propagacji wstecznej.

UWAGA Jeśli nie wiesz, w jaki sposób szkoli się sieci neuronowe za pomocą propagacji wstecznej, przeczytaj podrozdział A.4 w „Dodatku A”.

Zobaczmy na praktycznym przykładzie, jak działa konwersja identyfikatorów tokenów na wektor osadzeń. Założymy, że mamy cztery tokeny wejściowe o identyfikatorach 2, 3, 5 i 1:

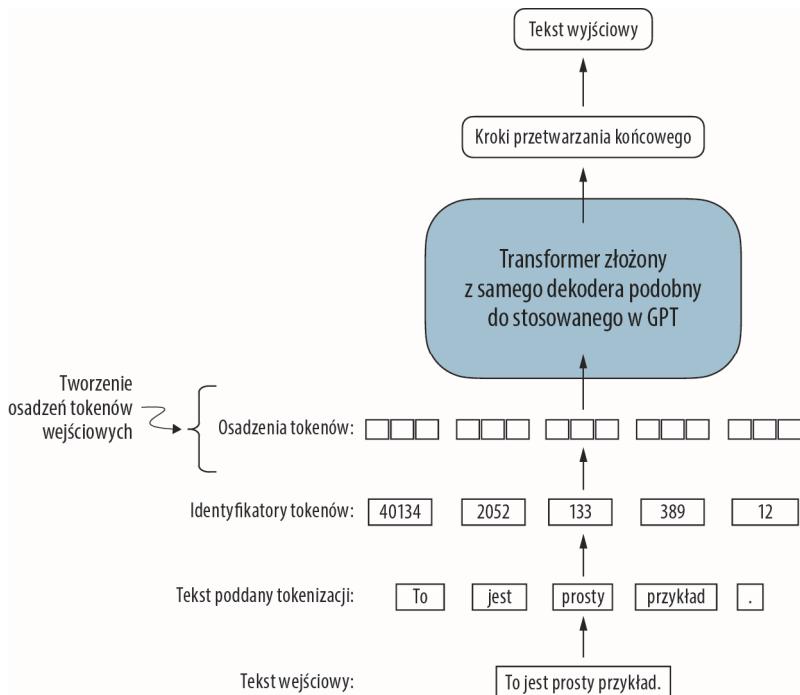
```
input_ids = torch.tensor([2, 3, 5, 1])
```

Dla uproszczenia założymy, że zbiór słownictwa jest mały i składa się z zaledwie 6 słów (zamiast słownictwa tokenizera BPE o rozmiarze 50 257 słów) i chcemy utworzyć osadzenia o rozmiarze 3 (w modelu GPT-3 rozmiar osadzeń ma 12 288 wymiarów):

```
vocab_size = 6
output_dim = 3
```

Korzystając ze zmiennych `vocab_size` i `output_dim`, można utworzyć egzemplarz warstwy osadzeń (z użyciem klasy `Embedding` z biblioteki PyTorch). W tym celu, aby uzyskać powtarzalność, można ustawić ziarno losowości na 123:

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```



Rysunek 2.15. Przygotowanie danych obejmuje tokenizację tekstu, konwersję tekstowych tokenów na identyfikatory oraz konwersję identyfikatorów tokenów na wektory osadzeń. W tym przypadku, aby utworzyć wektory osadzeń tokenów, uwzględniamy wcześniej utworzone identyfikatory

Instrukcja print wyświetla macierz wag warstwy osadzeń:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
       [ 0.9178,  1.5810,  1.3010],
       [ 1.2753, -0.2010, -0.1606],
       [-0.4015,  0.9666, -1.1481],
       [-1.1589,  0.3255, -0.6315],
       [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

Macierz wag warstwy osadzeń zawiera niskie, losowe wartości. Wartości te są optymalizowane podczas szkolenia LLM w ramach optymalizacji całego modelu LLM. Ponadto można zauważyć, że macierz wag ma sześć wierszy i trzy kolumny. Na każdy z sześciu możliwych tokenów w słowniku przypada jeden wiersz, a na każdy z trzech wymiarów osadzeń przypada jedna kolumna.

Teraz, aby uzyskać wektor osadzeń, zastosujmy go do identyfikatorów tokenów:

```
print(embedding_layer(torch.tensor([3])))
```

Oto otrzymany wektor osadzeń:

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

Jeśli porównasz wektor osadzeń dla tokena o identyfikatorze 3 z poprzednią macierzą osadzeń, zauważysz, że jest on identyczny z czwartym wierszem (listy w Pythonie zaczynają się od indeksu zerowego, więc jest to wiersz odpowiadający indeksowi 3). Mówiąc inaczej, warstwa osadzeń jest w istocie operacją wyszukiwania, która polega na pobraniu za pośrednictwem identyfikatora tokena wierszy z macierzy wag warstwy osadzeń.

UWAGA Osoby znające zasady kodowania one-hot powinny zauważyc, że opisane tutaj podejście warstwy osadzeń jest w istocie tylko bardziej wydajnym sposobem implementacji tego kodowania. Polega ono na zastosowaniu kodowania one-hot, po którym następuje mnożenie macierzy w warstwie w pełni połączonej. Ilustrację tego procesu można znaleźć w kodzie dostępnym w serwisie GitHub, pod adresem <https://mng.bz/ZEB5>. Ponieważ warstwa osadzeń jest wydajniejszą alternatywą dla kodowania one-hot i mnożenia macierzy, można ją traktować jako warstwę sieci neuronowej, możliwą do zoptymalizowania za pomocą propagacji wstecznej.

Wcześniej pokazałem, jak przekonwertować pojedynczy identyfikator tokena na trójwymiarowy wektor osadzeń. Zastosujmy teraz ten sposób do wszystkich czterech identyfikatorów wejściowych (`torch.tensor([2, 3, 5, 1])`):

```
print(embedding_layer(input_ids))
```

Z wyników instrukcji `print` widać, że uzyskujemy macierz 4×3 :

```
tensor([[ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-2.8400, -0.7849, -1.4096],
        [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0>)
```

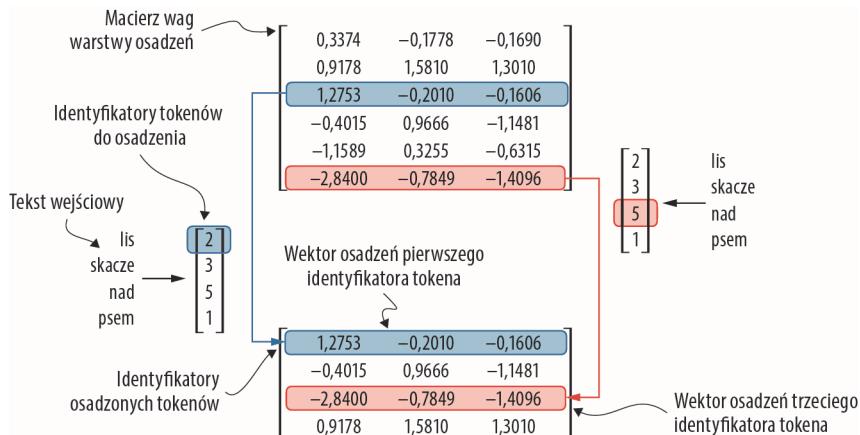
Każdy wiersz w tej wynikowej macierzy uzyskujemy w wyniku wyszukiwania w macierzy wag osadzeń, tak jak pokazałem na rysunku 2.16.

Po utworzeniu wektorów osadzeń na podstawie identyfikatorów tokenów wprowadzimy w tych wektorach niewielką modyfikację, tak by zakodować informacje o pozycji tokena w tekście.

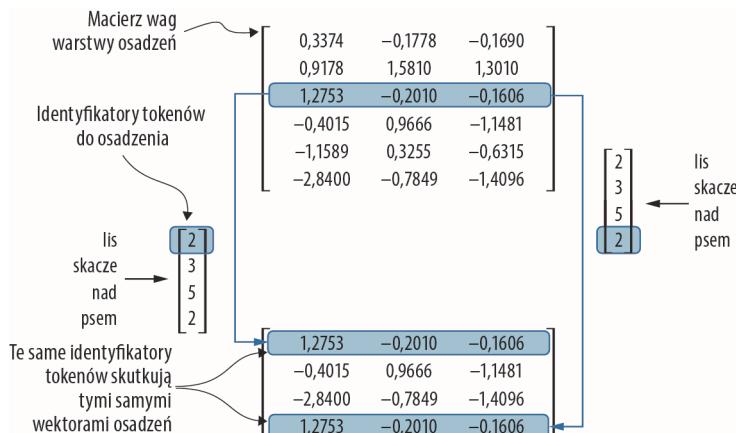
2.8. Kodowanie pozycji słów

Ogólnie rzeczą biorąc, osadzenia tokenów są odpowiednimi danymi wejściowymi dla modelu LLM. Niewielką wadą modeli LLM jest jednak to, że ich mechanizm samouwagi (patrz rozdział 3.) nie rozpoznaje pojęcia pozycji ani kolejności tokenów w sekwencji. Sposób działania wcześniej wprowadzonej warstwy osadzeń polega na tym, że ten sam identyfikator tokena jest zawsze mapowany na tę samą reprezentację wektorową, niezależnie od tego, gdzie identyfikator tokena znajduje się w sekwencji wejściowej, co pokazałem na rysunku 2.17.

Deterministyczne, niezależne od pozycji osadzenie identyfikatora tokena jest, ogólnie rzeczą biorąc, dobre z punktu widzenia możliwości odtwarzania. Ponieważ jednak



Rysunek 2.16. Warstwy osadzeń wykonują operację wyszukiwania: pobierają wektor osadzeń odpowiadający identyfikatorowi tokena z macierzy wag warstwy osadzeń. Na przykład wektor osadzeń tokenu o identyfikatorze 5 odpowiada szóstemu wierszowi macierzy wag warstwy osadzeń (jest to wiersz szósty, a nie piąty, ponieważ indeksy list Pythona zaczynają się od zera). Zakładamy, że identyfikatory tokenów zostały utworzone z użyciem niewielkiego słownika utworzonego w podrozdziale 2.3

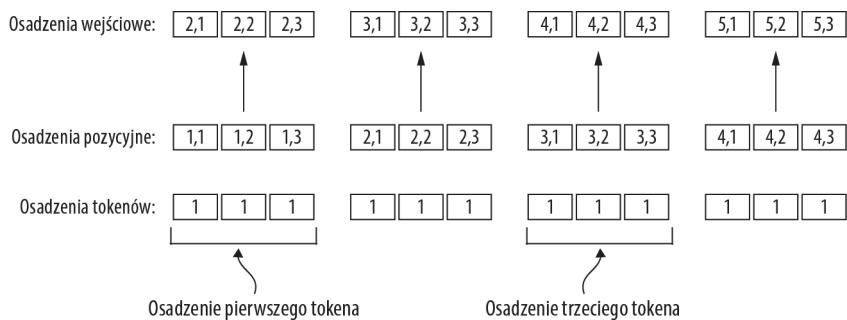


Rysunek 2.17. Warstwa osadzeń przekształca identyfikator tokena w tę samą reprezentację wektorową, niezależnie od tego, gdzie on się znajduje w sekwencji wejściowej. Na przykład token o identyfikatorze 5 będzie przekształcony na ten sam wektor osadzeń niezależnie od tego, czy znajduje się na pierwszej czy czwartej pozycji w wektorze wejściowym

Sam mechanizm samouwagi modelu LLM również działa niezależnie od pozycji, warto przekazać modelowi LLM dodatkowe informacje o pozycji.

Aby to osiągnąć, można wykorzystać dwie obszerne kategorie osadzeń z uwzględnieniem pozycji: względne osadzanie pozycyjne i bezwzględne osadzanie pozycyjne. Bezwzględne osadzenia pozycyjne są bezpośrednio powiązane z określonymi pozycjami w sekwencji. Dla każdej pozycji w sekwencji wejściowej do wektora osadzeń

tokena jest dodawane unikatowe osadzenie, które reprezentuje dokładną lokalizację tokena. Na przykład pierwszy token będzie miał określone osadzenie pozycyjne, drugi token inne osadzenie i tak dalej (rysunek 2.18).



Rysunek 2.18. W celu utworzenia wejściowych osadzeń dla modelu LLM do wektora osadzeń tokenów są dodawane osadzenia pozycyjne. Wektory pozycyjne mają ten sam wymiar, co oryginalne osadzenia tokenów. Dla uproszczenia osadzenia tokenów są wyświetlane z wartością 1

We względnym osadzaniu pozycyjnym bezwzględna pozycja tokena nie ma znaczenia. Zamiast tego ważna jest pozycja względna reprezentująca odległość między tokenami. Oznacza to, że model uczy się relacji w kategoriach „jak daleko od siebie”, a nie „w jakiej pozycji”. Dzięki temu podejściu model może lepiej generalizować sekwencje o różnej długości, nawet jeśli nie widział takich sekwencji podczas szkolenia.

Oba typy osadzania pozycyjnego mają na celu zwiększenie zdolności LLM do rozumienia kolejności tokenów i relacji między nimi, co zapewnia prognozy, które są dokładniejsze i bardziej świadome kontekstu. Wybór między nimi często zależy od konkretnej aplikacji i charakteru przetwarzanych danych.

Modele GPT firmy OpenAI wykorzystują bezwzględne osadzenia pozycyjne, które podczas procesu uczenia są optymalizowane, a nie są stałe lub predefiniowane, jak kodowanie pozycyjne w modelu *Original Transformer*. Ten proces optymalizacji jest częścią samego szkolenia modelu. Aby utworzyć dane wejściowe modelu LLM na potrzeby tego przykładu, utwórzmy początkowe osadzenia pozycyjne.

Wcześniej dla uproszczenia skupiliśmy się na osadzeniach o bardzo małych rozmiarach. Rozważmy teraz bardziej realistyczne i użyteczne rozmiary osadzenia i zakładmy wejściowe tokeny w 256-wymiarowej reprezentacji wektorowej, znacznie mniejszej od używanej pierwotnie w modelu GPT-3 (w GPT-3 rozmiar osadzenia wynosi 12 288 wymiarów), ale nadal rozsądnej z punktu widzenia eksperymentów. Ponadto założymy, że identyfikatory tokenów zostały utworzone przez zaimplementowany wcześniej tokenizer BPE, którego rozmiar słownictwa wynosi 50 257:

```
vocab_size = 50257
output_dim = 256
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

W przypadku wykorzystania utworzonej wcześniej warstwy `token_embedding_layer` próbkowanie danych z komponentu ładującego dane tworzyło osadzenie każdego tokena w każdej partii w wektorze o 256 wymiarach. Dla partii o rozmiarze 8, z czterema tokenami, w wyniku uzyskamy tensor o wymiarach $8 \times 4 \times 256$.

Najpierw tworzymy egzemplarz obiektu ładującego dane (patrz podrozdział 2.6):

```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length,
    stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Identyfikatory tokenów:\n", inputs)
print("\nKształt danych wejściowych:\n", inputs.shape)
```

Oto wynik działania tego kodu:

```
Identyfikatory tokenów:
tensor([[ 57,  8356, 2736,   334],
       [10247,   129,   120,    64],
       [41615,   368, 3619,    64],
       [ 402,   271, 10899,    64],
       [ 1976,    64, 1976, 21768],
       [  74, 41615, 1533,    78],
       [15632,  4496,   784, 1727],
       [38325,   416, 41615,   466]])
```

Kształt danych wejściowych:
`torch.Size([8, 4])`

Jak można zauważyć, tensor identyfikatorów tokenów ma wymiary 8×4 , co oznacza, że partia danych składa się z ośmiu próbek tekstu z czterema tokenami w każdej.

Teraz użyjemy warstwy osadzeń, aby osadzić identyfikatory tokenów w 256-wymiarowych wektorach:

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
```

Wywołanie funkcji `print` zwraca następujący wynik:

```
torch.Size([8, 4, 256])
```

Na podstawie tensora wyjściowego o wymiarach $8 \times 4 \times 256$ można zauważyć, że każdy identyfikator tokena jest osadzony w wektorze o 256 wymiarach.

W przypadku podejścia z bezwzględnym osadzaniem pozycji dla modelu GPT trzeba jedynie utworzyć kolejną warstwę osadzeń. Ma ona taki sam wymiar jak warstwa `token_embedding_layer`:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
print(pos_embeddings.shape)
```

Wejściem dla `pos_embeddings` jest zwykle wektor zastępczy `torch.arange(context_length)`, zawierający sekwencję liczb 0, 1, ..., aż do maksymalnej długości wejścia pomniejszonej o jeden. `context_length` to zmienna reprezentująca obsługiwany rozmiar wejścia modelu LLM. W tym przykładzie wybieramy ją w sposób podobny do maksymalnej długości tekstu wejściowego. W praktyce, jeśli tekst wejściowy jest dłuższy niż obsługiwana długość kontekstu, można go odpowiednio przyciąć.

Oto wynik działania instrukcji `print`:

```
torch.Size([4, 256])
```

Jak można zauważyc, pozycyjny tensor osadzeń składa się z czterech wektorów o 256 wymiarach. Te wektory można teraz bezpośrednio dodać do osadzeń tokenów. Biblioteka PyTorch automatycznie doda tensor `pos_embeddings` o wymiarach 4×256 do każdego tensora osadzeń tokenów o wymiarach 4×256 w każdej z ośmiu partii:

```
input_embeddings = token_embeddings + pos_embeddings  
print(input_embeddings.shape)
```

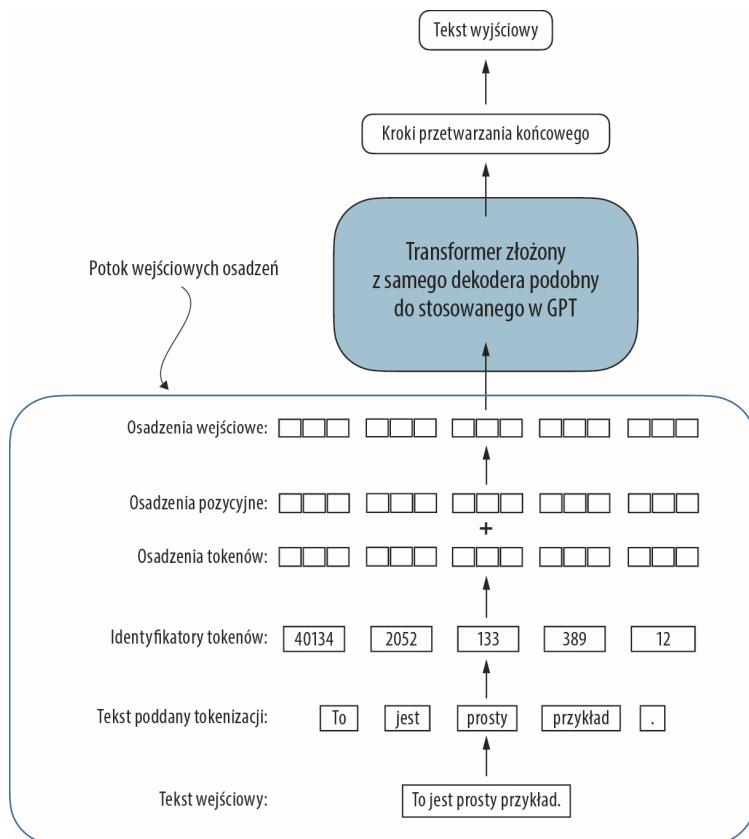
Oto wynik działania instrukcji `print`:

```
torch.Size([8, 4, 256])
```

Utworzone osadzenia `input_embeddings`, jak podsumowano na rysunku 2.19, są przykładami wejść, które mogą być przetwarzane przez główne moduły modelu LLM. Te główne moduły zaczniemy implementować w następnym rozdziale.

Podsumowanie

- Modele LLM nie potrafią przetwarzać surowego tekstu. Wymagają konwersji danych tekstowych na wektory liczbowe, znane jako osadzenia (ang. *embeddings*). Osadzenia transformują dyskretne dane (takie jak słowa lub obrazy) na ciągłe przestrzenie wektorowe, odpowiednie do wykonywania działań typowych dla sieci neuronowych.
- W pierwszym kroku surowy tekst zostaje podzielony na tokeny, którymi mogą być słowa bądź znaki. Następnie tokeny są konwertowane na reprezentacje liczb całkowitych, określane jako identyfikatory tokenów.
- Dla poprawy czytelności danych, dzięki czemu będą bardziej zrozumiałe dla modelu, a także w celu obsługi różnych kontekstów, na przykład nieznanych słów, albo po to, by oznaczyć granicę między niepowiązanymi tekstami, można dodawać specjalne tokeny, takie jak `<|unk|>` i `<|endoftext|>`.
- Tokenizer BPE oparty na kodowaniu par bajtów używany w takich modelach LLM jak GPT-2 i GPT-3 potrafi skutecznie obsługiwać nieznane słowa dzięki podzieleniu ich na jednostki — podwyrazy lub pojedyncze znaki.



Rysunek 2.19. W ramach potoku przetwarzania danych wejściowych tekst wejściowy najpierw jest dzielony na pojedyncze tokeny. Te tokeny są następnie konwertowane z użyciem słownika na identyfikatory. Identyfikatory tokenów są konwertowane na wektory osadzeń, do których dodawane są osadzenia pozycyjne o zbliżonym rozmiarze. W efekcie powstają osadzenia wejściowe wykorzystywane jako dane wejściowe dla głównych warstw modelu LLM

- W celu generowania par wejście-ceł na potrzeby szkolenia modelu LLM stosujemy dla tokenizowanych danych podejście okna przesuwnego.
- Warstwy osadzeń w bibliotece PyTorch działają podobnie jak operacja wyszukiwania — pobierają wektory odpowiadające identyfikatorom tokenów. Wynikowe wektory osadzeń dostarczają ciągłych reprezentacji tokenów, co ma kluczowe znaczenie w szkoleniu modeli uczenia głębokiego, takich jak modele LLM.
- Podczas gdy osadzenia tokenów zapewniają spójne reprezentacje wektorowe dla każdego tokena, nie zawierają informacji o pozycji tokena w sekwencji. Aby temu zaradzić, można zastosować dwa główne typy osadzania pozycyjnego: bezwzględne i względne. Modele GPT firmy OpenAI wykorzystują bezwzględne osadzenia pozycyjne, które są dodawane do wektorów osadzeń tokenów i są optymalizowane podczas uczenia modelu.

Kodowanie mechanizmów uwagi

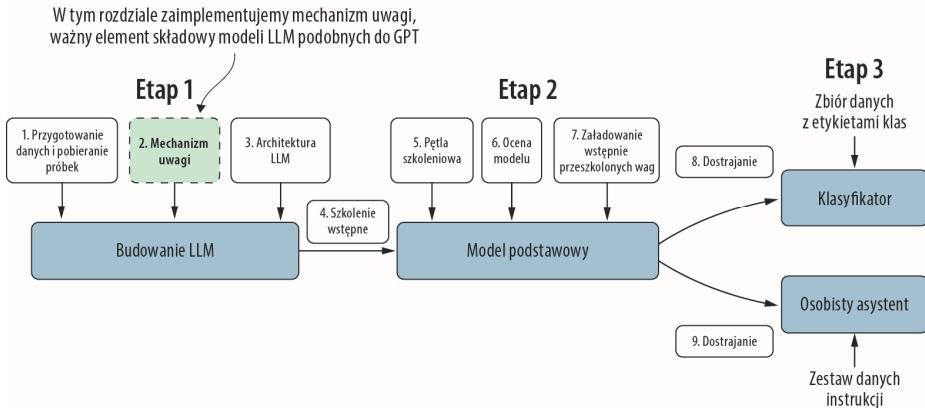
W tym rozdziale:

- Powody stosowania mechanizmów uwagi w sieciach neuronowych
- Prosty framework samouwagi i jego usprawnienie
- Moduł uwagi przyczynowej pozwalający modelowi LLM generować po jednym tokenie na raz
- Maskowanie losowo wybranych wag uwagi w celu ograniczenia nadmiernego dopasowania
- Łączenie wielu modułów uwagi przyczynowej w wielogłowicowy moduł uwagi

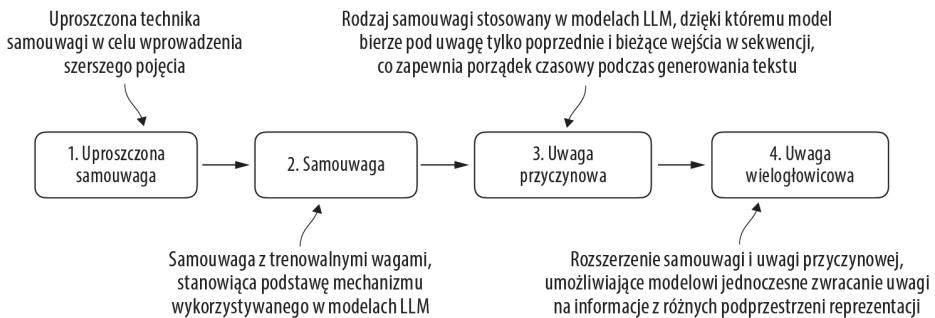
Z wcześniejszych rozdziałów dowiedziałeś się, jak przygotować tekst wejściowy do szkolenia modelu LLM przez podzielenie tekstu na pojedyncze tokeny reprezentujące wyrazy i podwyrazy, które na potrzeby modelu LLM można zakodować w reprezentacjach wektorowych, czyli osadzeniach.

W kolejnym punkcie przyjrzymy się integralnej części architektury LLM — mechanizmowi uwagi (rysunek 3.1). W tym rozdziale będziemy analizować mechanizmy uwagi głównie w oderwaniu od innych komponentów, koncentrując się na ich mechaniczne działania. Następnie zakodujemy pozostałe części modelu LLM otaczające mechanizm samouwagi, aby zaobserwować jego działanie w praktyce i stworzyć model do generowania tekstu.

Zaimplementujemy cztery różne warianty mechanizmów uwagi (rysunek 3.2). Te różne warianty uwagi opierają się na sobie nawzajem, a celem ich tworzenia jest użycanie zwartej i wydajnej implementacji wielogłowicowej uwagi. Następnie będzie można włączyć tę implementację do architektury modelu LLM, który zakodujemy w następnym rozdziale.



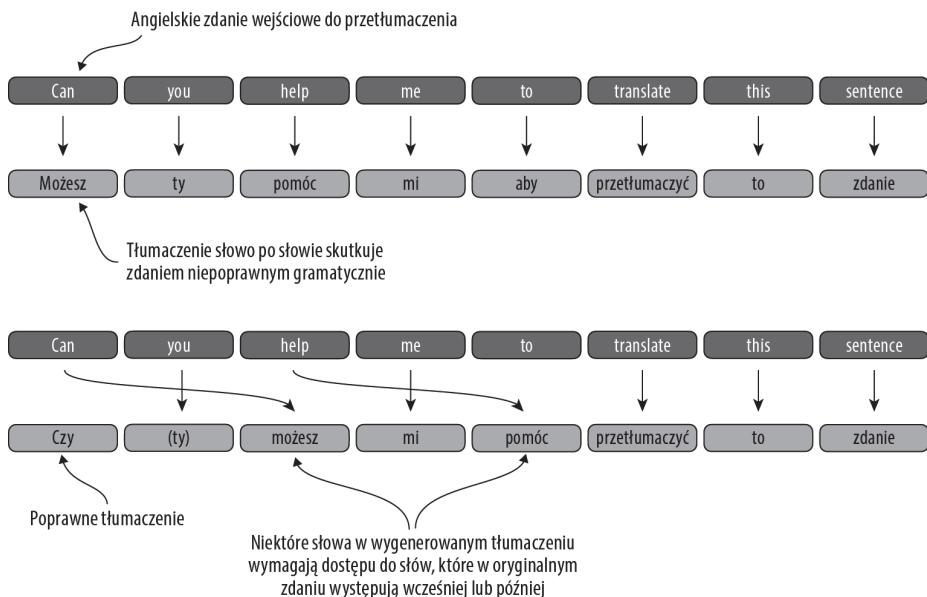
Rysunek 3.1. Trzy główne etapy kodowania modelu LLM. W tym rozdziale skupię się na kroku 2. etapu 1. — implementacji mechanizmów uwagi będących integralną częścią architektury LLM



Rysunek 3.2. Rysunek przedstawia różne mechanizmy uwagi, które będziemy kodować w tym rozdziale. Zaczniemy od uproszczonej wersji warstwy samouwagi, a następnie dodamy trenowalne wagi. Mechanizm uwagi przyczynowej dodaje do warstwy samouwagi maskę, która pozwala modelowi LLM generować po jednym słowie na raz. Na koniec zaimplementujemy uwagę wielogłowicową, w której mechanizm uwagi jest zorganizowany w formie wielu głów. Dzięki temu model może równolegle przetwarzać różne aspekty danych wejściowych

3.1. Problem z modelowaniem długich sekwencji

Zanim zagłębimy się w mechanizm *samouwagi*, będący sercem modeli LLM, rozważmy problem architektur poprzedzających modele LLM, które nie zawierały mechanizmów uwagi. Założymy, że chcemy opracować model, który tłumaczy tekst z jednego języka na inny. Jak pokazano na rysunku 3.3, ze względu na struktury gramatyczne w językach źródłowym i docelowym nie można po prostu przetłumaczyć tekstu słowo po słowie.



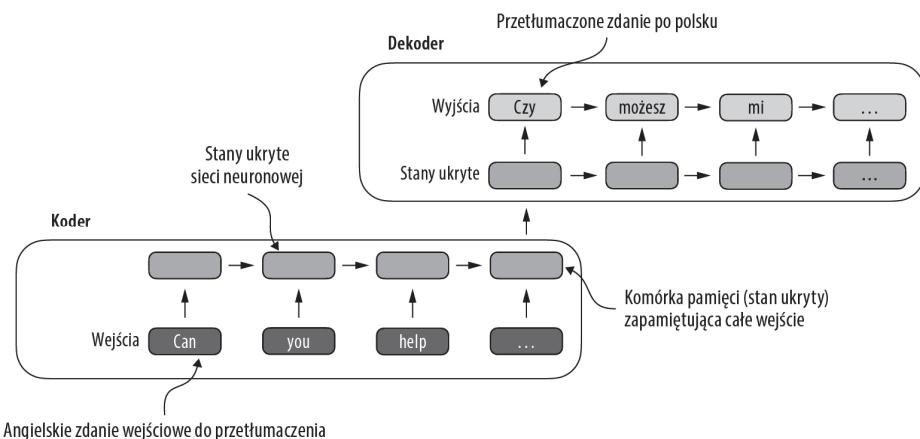
Rysunek 3.3. Podczas tłumaczenia tekstu z jednego języka na inny, na przykład z angielskiego na polski, nie można po prostu tłumaczyć słowa po słowie. Proces tłumaczenia wymaga zrozumienia kontekstu i dopasowania gramatycznego

Aby rozwiązać ten problem, często stosuje się głęboką sieć neuronową z dwoma podmodułami, *koderem* i *dekoderem*. Najpierw koder wczytuje i przetwarza cały tekst, a następnie dekoder tworzy przetłumaczony tekst.

Przed pojawiением się transformerów najpopularniejszą architekturą koder-dekoder wykorzystywaną do tłumaczenia z języka na inny język były *rekurencyjne sieci neuronowe* (RNN). RNN to rodzaj sieci neuronowej, w której dane wyjściowe z poprzednich etapów są przekazywane jako dane wejściowe do etapu bieżącego. Dzięki temu sieci RNN dobrze nadają się do sekwencyjnych danych, takich jak tekst. Jeśli nie znasz tematyki sieci RNN, nie martw się — aby śledzić tematy opisywane w tym rozdziale, nie musisz znać szczegółowego opisu sieci RNN. W tym rozdziale skupię się bardziej na ogólnym pojęciu konfiguracji koder-dekoder.

W sieci RNN koder-dekoder tekst wejściowy jest podawany do kodera, który przetwarza go w sposób sekwencyjny. W każdym kroku koder aktualizuje swój ukryty stan (wewnętrzne wartości w warstwach ukrytych) i próbuje uchwycić całe znaczenie zdania wejściowego w końcowym stanie ukrytym (rysunek 3.4). Następnie dekoder przyjmuje ten końcowy stan ukryty i słowo po słowie rozpoczyna generowanie przetłumaczonego zdania. W każdym kroku aktualizuje również swój ukryty stan, który ma obejmować kontekst niezbędny do przewidzenia następnego słowa.

Chociaż nie musisz znać szczegółów działania sieci RNN typu koder-dekoder, powinieneś zapamiętać, że część kodująca przetwarza cały tekst wejściowy do stanu ukrytego (komórki pamięci). Następnie dekoder przetwarza ten ukryty stan w celu



Rysunek 3.4. Przed pojawieniem się modeli transformerów popularnym wyborem do tłumaczenia maszynowego były sieci koder-dekoder typu RNN. Koder pobiera jako dane wejściowe sekwencję tokenów z języka źródłowego. Stan ukryty (pośrednia warstwa sieci neuronowej) kodera koduje skompresowaną reprezentację całej sekwencji wejściowej. Następnie dekoder wykorzystuje swój bieżący stan ukryty, aby rozpocząć tłumaczenie, token po tokenie

wygenerowania danych wyjściowych. Ukryty stan można porównać do wektora osadzeń, pojęcia, które omówiłem w rozdziale 2.

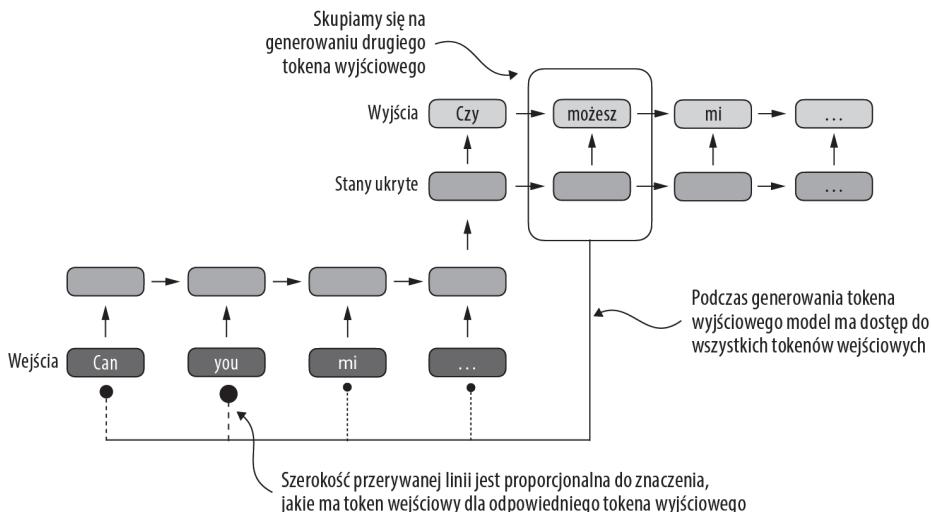
Dużym ograniczeniem sieci RNN typu koder-dekoder jest brak możliwości uzyskania podczas fazy dekodowania bezpośredniego dostępu do wcześniejszych stanów ukrytych kodera. W konsekwencji dekodowanie opiera się wyłącznie na bieżącym stanie ukrytym, który zawiera wszystkie istotne informacje. W złożonych zdaniach, w których zależności obejmują duże odległości, może to prowadzić do utraty kontekstu.

Na szczęście nie trzeba rozumieć szczegółów działania sieci RNN, żeby zbudować model LLM. Wystarczy zapamiętać, że sieć RNN typu koder-dekoder miała wadę, która stała się motywacją do zaprojektowania mechanizmów uwagi.

3.2. Przechwytywanie zależności między danymi za pomocą mechanizmów uwagi

Chociaż sieci RNN dobrze sprawdzają się przy tłumaczeniu krótkich zdań, nie działają dobrze w przypadku dłuższych tekstów. Nie mają one bowiem bezpośredniego dostępu do poprzednich słów w danych wejściowych. Jedną z głównych wad tego podejścia jest to, że sieć RNN przed przekazaniem danych do dekodera musi zapamiętać całe zakodowane wejście w pojedynczym stanie ukrytym (rysunek 3.4).

Z tego powodu w 2014 roku opracowano dla sieci RNN mechanizm *uwagi Bahdanaua* (nazwany na cześć pierwszego autora artykułu opisującego ten mechanizm; więcej informacji można znaleźć w „Dodatku B”). Ten mechanizm modyfikuje sieć RNN koder-dekoder w taki sposób, że dekoder może uzyskać selektywny dostęp do różnych części sekwencji wejściowej na każdym etapie dekodowania (rysunek 3.5).

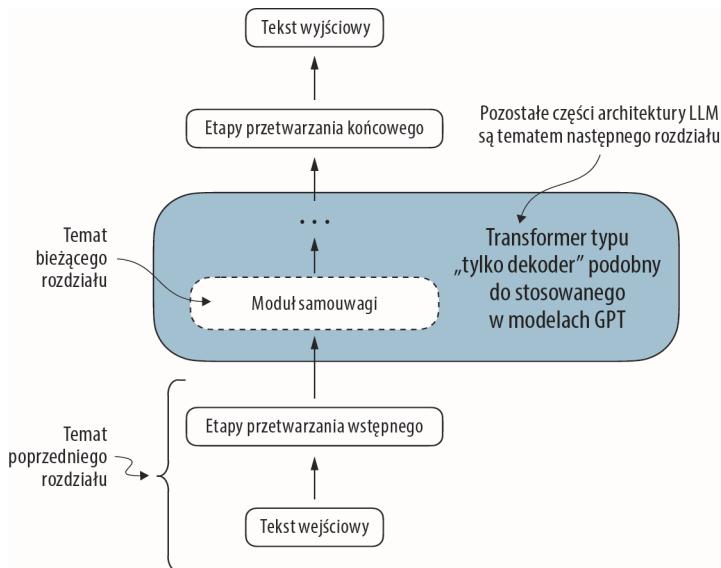


Rysunek 3.5. Dzięki mechanizmowi uwagi część dekodująca sieci odpowiedzialna za generowanie tekstu może uzyskać selektywny dostęp do wszystkich tokenów wejściowych. Oznacza to, że z perspektywy celu, jakim jest wygenerowanie określonego tokena wyjściowego, niektóre tokeny wejściowe są ważniejsze niż inne. Stopień ważności jest określany przez wagi warstwy uwagi, które obliczymy później. Należy zauważać, że na tym rysunku pokazano ogólną ideę warstwy uwagi. Nie przedstawia on dokładnej implementacji mechanizmu Bahdanaua — metody stosowanej w sieciach RNN, której opisanie wykracza poza zakres tej książki

Co ciekawe, zaledwie trzy lata później naukowcy odkryli, że do budowania głębokich sieci neuronowych na potrzeby przetwarzania języka naturalnego nie są konieczne architektury RNN, i zaproponowali architekturę *Original Transformer* (omówioną w rozdziale 1.), obejmującą inspirowany mechanizmem uwagi Bahdanaua mechanizm samouwagi.

Samouwaga jest mechanizmem, który podczas obliczania reprezentacji sekwencji pozwala uwzględnić istotność każdej pozycji w sekwencji wejściowej, czyli „zwrócić uwagę” na wszystkie inne pozycje w tej samej sekwencji. Samouwaga jest kluczowym elementem współczesnych modeli LLM opartych na architekturze transformera, takich jak seria modeli GPT.

Ten rozdział koncentruje się na kodowaniu i zrozumieniu mechanizmu samouwagi używanego w modelach podobnych do GPT (rysunek 3.6). W następnym rozdziale zakodujemy pozostałe części modelu LLM.



Rysunek 3.6. Samouwaga jest mechanizmem stosowanym w transformerach do wyznaczania bardziej wydajnych reprezentacji wejściowych. Umożliwia bowiem każdej pozycji w sekwencji interakcje i określanie wag znaczenia w odniesieniu do wszystkich innych pozycji w tej samej sekwencji. W tym rozdziale zakodujemy mechanizm samouwagi od podstaw, a następnie, w kolejnym rozdziale, przejdziemy do kodowania pozostałych części modelu LLM podobnego do GPT

3.3. Zwracanie uwagi na różne części danych wejściowych przez mechanizm samouwagi

Tematem tego podrozdziału jest wewnętrzne działanie mechanizmu samouwagi, który nauczysz się kodować od podstaw. Warstwa samouwagi stanowi fundament każdego modelu LLM opartego na architekturze transformera. Choć temat może wymagać dużego skupienia i uwagi (gra słów nie jest zamierzona), to po zrozumieniu podstaw poradzisz sobie z jednym z najtrudniejszych wyzwań omówionych w tej książce oraz związanych z implementowaniem modeli LLM w ogóle.

Ponieważ warstwa samouwagi może, zwłaszcza jeśli spotykasz się z nią po raz pierwszy, wydawać się złożona, zacznę od przeanalizowania jej uproszczonej wersji. Następnie zaimplementujemy mechanizm samouwagi z możliwymi do wyuczenia wagami używanymi w modelu LLM.

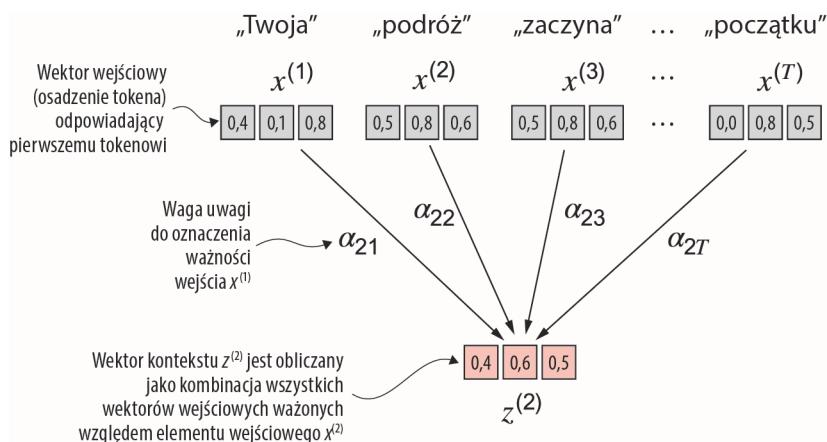
„Samo” w mechanizmie samouwagi

W mechanizmie samouwagi część „samo” odnosi się do zdolności mechanizmu do obliczania wag uwagi przez badanie związków pomiędzy tokenami na różnych pozycjach w ramach pojedynczej sekwencji wejściowej. Mechanizm samouwagi ocenia związki i zależności między różnymi częściami danych wejściowych, takimi jak słowa w zdaniu lub piksele na obrazie, i uczy się ich.

Jest to przeciwieństwo tradycyjnych mechanizmów uwagi, w których uwaga skupia się na związkach między elementami należącymi do różnych sekwencji. Takie mechanizmy występują w modelach sekwencja-sekwencja podobnych do przykładu pokazanego na rysunku 3.5.

3.3.1. Prosty mechanizm samouwagi bez trenowalnych wag

Zacznijmy od zaimplementowania uproszczonego wariantu mechanizmu samouwagi, bez trenowalnych wag (rysunek 3.7). Chodzi o to, by przed dodaniem trenowalnych wag zilustrować kilka kluczowych pojęć związanych z samouwagą.



Rysunek 3.7. Celem mechanizmu samouwagi jest obliczenie dla każdego elementu wejściowego wektora kontekstu, który łączy informacje ze wszystkich innych elementów wejściowych. W tym przykładzie obliczamy wektor kontekstu $z^{(2)}$. Znaczenie lub wkład każdego elementu wejścia do obliczenia $z^{(2)}$ są określone przez wagi uwagi α_{21} do α_{2T} . Podczas obliczania $z^{(2)}$ wagi uwagi są wyliczane w odniesieniu do elementu wejściowego $x^{(2)}$ i wszystkich innych wejść

Na rysunku 3.7 przedstawiono sekwencję wejściową, oznaczoną jako x , składającą się z T elementów reprezentowanych jako wektory od $x^{(1)}$ do $x^{(T)}$. Ta sekwencja zazwyczaj reprezentuje tekst (na przykład zdanie), który został wcześniej przekształcony w osadzenia tokenów.

Rozważmy na przykład tekst wejściowy typu „Twoja podróż zaczyna się od początku”. W tym przypadku każdy element sekwencji, taki jak $x^{(1)}$, odpowiada d -wymiarowemu wektorowi osadzeń reprezentującemu określony token, na przykład „Twoja”.

Na rysunku 3.7 przedstawiono te wektory wejściowe w postaci trójwymiarowych osadzeń.

Celem warstwy samouwagi jest obliczenie wektorów kontekstu $z^{(i)}$ dla każdego elementu $x^{(i)}$ w sekwencji wejściowej. Wektor kontekstu można zinterpretować jako wzbogacony wektor osadzeń.

Aby zilustrować to pojęcie, skupmy się na wektorze osadzeń drugiego elementu wejścia, $x^{(2)}$ (czyli wektorze odpowiadającym tokenowi „podróż”) i odpowiadającym mu wektorze kontekstu, $z^{(2)}$, pokazanym u dołu rysunku 3.7. Ten rozszerzony wektor kontekstu, $z^{(2)}$, jest osadzeniem, zawierającym informacje o $x^{(2)}$ i wszystkich innych elementach wejściowych — od $x^{(1)}$ do $x^{(T)}$.

Wektory kontekstu odgrywają w mechanizmie samouwagi kluczową rolę. Ich celem jest tworzenie wzbogaconych reprezentacji każdego elementu w sekwencji wejściowej (takiej jak zdanie) przezłączenie informacji ze wszystkich innych elementów w sekwencji (rysunek 3.7). W przypadku modeli LLM, które muszą rozumieć związki pomiędzy słowami i ich znaczenie w zdaniu, jest to niezbędne. W kolejnym etapie dodamy trenowalne wagi, które pomogą modelowi LLM nauczyć się konstruować wektory kontekstu w taki sposób, aby model LLM mógł je wykorzystać podczas generowania następnego tokena. Najpierw jednak, aby obliczyć te wagi i krok po kroku wyznaczyć wynikowy wektor kontekstu, zaimplementujemy uproszczony mechanizm samouwagi.

Rozważmy następujące zdanie wejściowe, które zostało już osadzone w trójwymiarowych wektorach (patrz rozdział 2.). Wybrałem osadzenie o niewielkich rozmiarach, tak aby zmieściło się na stronie bez podziałów wierszy:

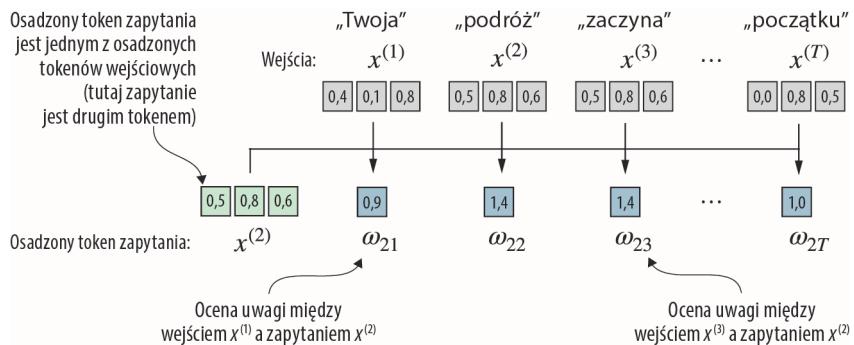
```
import torch
inputs = torch.tensor(
    [[0.43, 0.15, 0.89], # Twoja (x ^ 1)
     [0.55, 0.87, 0.66], # podróż (x ^ 2)
     [0.57, 0.85, 0.64], # zaczyna (x ^ 3)
     [0.22, 0.58, 0.33], # się (x ^ 4)
     [0.77, 0.25, 0.10], # od (x ^ 5)
     [0.05, 0.80, 0.55]] # początku (x ^ 6)
)
```

Pierwszy krok podczas implementacji mechanizmu samouwagi polega na obliczeniu wartości pośrednich ω , określanych jako oceny uwagi — ang. *attention scores* (rysunek 3.8). Ze względu na ograniczenia miejsca zaprezentowane na rysunku wartości tensora `inputs` zostały przycięte — na przykład wartość 0,87 przycięto do 0,8. W tej wersji osadzenia słów „podróż” i „zaczyna” przez przypadek mogą wydawać się podobne.

Na rysunku 3.8 zilustrowałem sposób, w jaki należy obliczyć pośrednie oceny uwagi między tokenem zapytania a pozostałymi tokenami wejściowymi. Oceny obliczamy jako iloczyn skalarny zapytania $x^{(2)}$ i pozostałych tokenów wejścia:

```
query = inputs[1]                                ←
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query) print(attn_scores_2)
```

Drugi token
wejściowy służy
jako zapytanie.



Rysunek 3.8. Ogólnym celem tego rysunku jest zilustrowanie obliczeń wektora kontekstu $z^{(2)}$ z użyciem drugiego elementu wejściowego, $x^{(2)}$, jako zapytania. Na tym rysunku przedstawiłem pierwszy etap pośredni — obliczenie ocen uwagi w jako iloczynu skalarnego wektora reprezentującego zapytanie $x^{(2)}$ przez wszystkie inne elementy wejścia (należy zapamiętać, że dla poprawy czytelności liczby przycięto do jednej cyfry po przecinku)

Obliczone oceny uwagi są następujące

`tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])`

Iloczyn skalarny

Iloczyn skalarny to zwięzły sposób mnożenia dwóch wektorów polegający na pomnożeniu elementów wektorów przez siebie, a następnie zsumowaniu wyników. Można to przedstawić w następujący sposób:

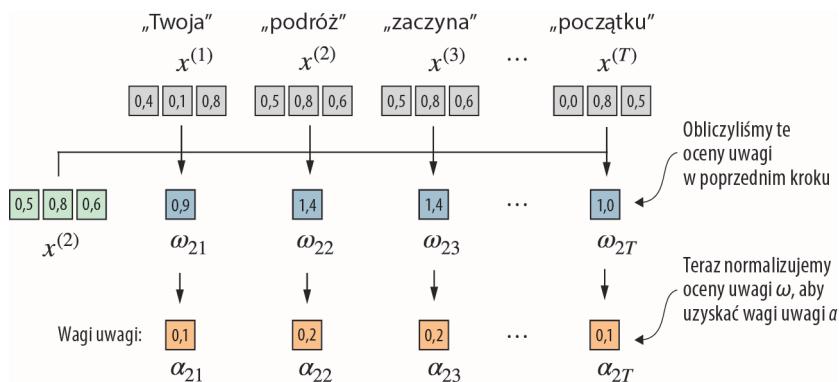
```
res = 0.
for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
print(torch.dot(inputs[0], query))
```

Wynik potwierdza, że suma mnożenia elementów daje takie same wyniki jak iloczyn skalarny:

```
tensor(0.9544)
tensor(0.9544)
```

Poza tym, że operację iloczynu skalarnego postrzega się jako matematyczne narzędzie, które łączy dwa wektory w celu uzyskania skalarnej wartości, jest on miarą podobieństwa, ponieważ określa ilościowo, jak bliskie siebie są dwa wektory: większa wartość iloczynu skalarnego wskazuje na większe wyrównanie, czyli większe podobieństwo między wektorami. W kontekście mechanizmów samouwagi iloczyn skalarny określa stopień, w jakim każdy element w sekwencji skupia się na innym elemencie lub „zwraca na niego uwagę”: im wyższy iloczyn skalarny, tym wyższe podobieństwo i ocena uwagi między dwoma elementami.

W następnym kroku, jak pokazałem na rysunku 3.9, normalizujemy wszystkie obliczone wcześniej oceny uwagi. Głównym celem normalizacji jest uzyskanie wag uwagi, które sumują się do 1. Normalizacja to konwencja przydatna do interpretowania wyników szkolenia modeli LLM i utrzymania stabilności tego szkolenia. Oto prosta metoda osiągnięcia tego etapu normalizacji:



Rysunek 3.9. Następnym krokiem po obliczeniu ocen uwagi od ω_{21} do ω_{2T} w odniesieniu do zapytania wejściowego $x^{(2)}$ jest obliczenie wag uwagi od α_{21} do α_{2T} , które uzyskujemy przez normalizację ocen uwagi

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Wagi uwagi:", attn_weights_2_tmp)
print("Suma:", attn_weights_2_tmp.sum())
```

Jak można zauważyć na podstawie wyniku, wagi uwagi sumują się do 1:

```
Wagi uwagi: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Suma: tensor(1.0000)
```

W praktyce bardziej powszechnie i zalecane jest stosowanie do normalizacji funkcji *softmax*. Podejście to sprawdza się lepiej w odniesieniu do wartości skrajnych i pozwala uzyskać podczas szkolenia lepsze właściwości gradientu. Oto podstawowa implementacja funkcji *softmax* na potrzeby normalizacji wyników uwagi:

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Wagi uwagi:", attn_weights_2_naive)
print("Suma:", attn_weights_2_naive.sum())
```

Jak widać w wyniku, funkcja *softmax* również się sprawdza i normalizuje wagi uwagi w taki sposób, aby sumowały się do 1:

```
Wagi uwagi: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Suma: tensor(1.)
```

Ponadto, za sprawą funkcji *softmax*, wagi uwagi są zawsze dodatnie. Dzięki temu wyniki można interpretować jako prawdopodobieństwa lub względową istotność, przy czym wyższe wartości wag wskazują na większą istotność.

Warto zauważyć, że z pokazaną naiwną implementacją funkcji *softmax* (*softmax_naive*) mogą się wiązać problemy z niestabilnością numeryczną, na przykład dla wysokich lub niskich wartości wejść mogą wystąpić przepełnienia i niedopełnienia.

Z tego powodu w praktyce zaleca się korzystanie z implementacji *softmax* z biblioteki PyTorch, którą gruntownie zoptymalizowano pod kątem wydajności:

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Wagi uwagi:", attn_weights_2)
print("Suma:", attn_weights_2.sum())
```

Tutaj zastosowanie funkcji daje takie same wyniki jak w przypadku poprzedniej funkcji, *softmax_naive*:

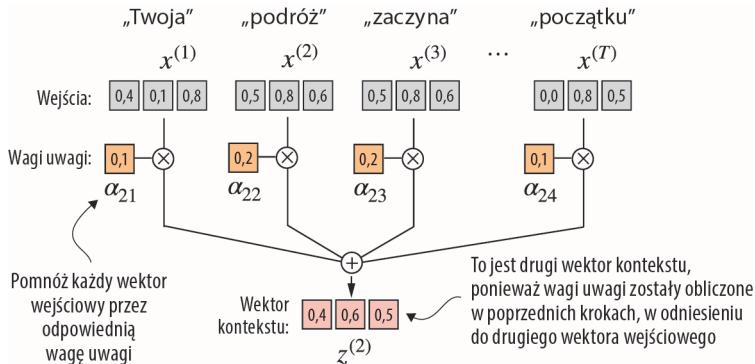
```
Wagi uwagi: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Suma: tensor(1.)
```

Po obliczeniu znormalizowanych wag uwagi można wykonać ostatni krok (rysunek 3.10): obliczenie wektora kontekstu $z^{(2)}$ przez pomnożenie osadzonych tokenów wejściowych, $x^{(i)}$, przez odpowiednie wagi uwagi, a następnie zsumowanie uzyskanych wektorów. Wektor kontekstu $z^{(2)}$ jest ważoną sumą wszystkich wektorów wejściowych, uzyskaną przez pomnożenie każdego wektora wejściowego przez odpowiadającą mu wagę uwagi:

```
query = inputs[1]                                     ← Drugi token wejściowy odgrywa rolę zapytania
context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

Wyniki obliczeń są następujące:

```
tensor([0.4419, 0.6515, 0.5683])
```



Rysunek 3.10. Ostatnim krokiem po obliczeniu i znormalizowaniu wyników uwagi jest wyliczenie wag uwagi dla zapytania $x^{(2)}$. Polega ono na obliczeniu wektora kontekstu $z^{(2)}$. Ten wektor kontekstu jest kombinacją wszystkich wektorów wejściowych od $x^{(1)}$ do $x^{(T)}$ z uwzględnieniem wag uwagi

W kolejnym kroku uogólnimy procedurę obliczania wektorów kontekstu, aby obliczyć wszystkie wektory kontekstu jednocześnie.

3.3.2. Obliczanie wag uwagi dla wszystkich tokenów wejściowych

Do tej pory obliczyliśmy wagi uwagi i wektor kontekstu dla wejścia 2 tak, jak pokazano w wyróżnionym wierszu na rysunku 3.11. Teraz rozszerzymy te obliczenia, aby obliczyć wagi uwagi i wektory kontekstu dla wszystkich wejść.

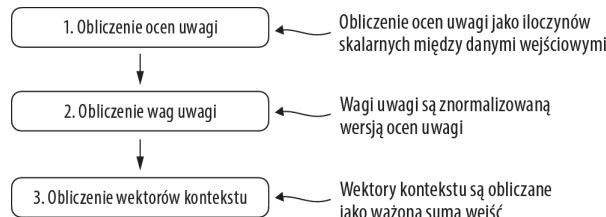
	Twoja	podróż	zaczyna	się	od	początku
Twoja	0,20	0,20	0,19	0,12	0,12	0,14
podróż	0,13	0,23	0,23	0,12	0,10	0,15
zaczyna	0,13	0,23	0,23	0,12	0,11	0,15
się	0,14	0,20	0,20	0,14	0,12	0,17
od	0,15	0,19	0,19	0,13	0,18	0,12
początku	0,13	0,21	0,21	0,14	0,09	0,18

Ten wiersz zawiera obliczone wcześniej wagi uwagi (znormalizowane oceny uwagi)

Rysunek 3.11. Zaznaczony wiersz pokazuje wagi uwagi dla drugiego elementu wejścia jako zapytania. W kolejnym kroku uogólnimy obliczenia, aby uzyskać pozostałe wagi uwagi (należy pamiętać, że dla większej czytelności liczby na tym rysunku przycięto do dwóch cyfr po przecinku. Wartości w każdym wierszu powinny sumować się do 1,0, czyli 100%)

Wykonujemy te same trzy kroki, co poprzednio (rysunek 3.12). Tym razem jednak, aby obliczyć wszystkie wektory kontekstu zamiast tylko drugiego, $z^{(2)}$, wprowadzimy w kodzie szereg modyfikacji:

```
attn_scores = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
print(attn_scores)
```



Rysunek 3.12. W kroku 1., aby obliczyć iloczyn skalarny dla wszystkich par wejść, wprowadzamy dodatkową pętlę for

Uzyskane oceny uwagi są następujące:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
       [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
```

```
[0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
[0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
[0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
[0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

Każdy element w tensorze reprezentuje ocenę uwagi między poszczególnymi parami wejść (rysunek 3.11). Należy zwrócić uwagę, że wartości na tym rysunku są znormalizowane, dlatego różnią się od nieznormalizowanych ocen uwagi w poprzednim tensorze. Normalizacją zajmiemy się później.

Podczas obliczania poprzedniego tensora ocen uwagi używaliśmy pętli `for` w Pythonie. Pętle `for`, ogólnie rzecz biorąc, są jednak wolne, a takie same wyniki można osiągnąć za pomocą mnożenia macierzy:

```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

Możemy wizualnie potwierdzić, że wyniki są takie same jak poprzednio:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
       [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
       [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
       [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
       [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
       [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

W kroku 2. pokazanym na rysunku 3.12 poddajemy normalizacji każdy wiersz w taki sposób, aby wartości w każdym wierszu sumowały się do 1:

```
attn_weights = torch.softmax(attn_scores, dim=-1)
print(attn_weights)
```

W efekcie uzyskujemy następujący tensor wag uwagi, odpowiadający wartościami pokazanym na rysunku 3.10:

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
       [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
       [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],
       [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],
       [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],
       [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]])
```

W kontekście biblioteki PyTorch parametr `dim` w takich funkcjach jak `torch.softmax` określa wymiar tensora wejściowego wykorzystywany do obliczeń funkcji. Ustawienie `dim=-1` instruuje funkcję `softmax`, aby zastosowała normalizację z użyciem ostatniego wymiaru tensora `attn_scores`. Jeśli `attn_scores` jest tensorem dwuwymiarowym (na przykład o kształcie [wiersze, kolumny]), zostanie znormalizowany w kolumnach, tak aby suma elementów w każdym wierszu (suma w wymiarze kolumny) wynosiła 1.

Aby sprawdzić, czy suma elementów wierszy rzeczywiście wynosi 1., można skorzystać z następującego kodu:

```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
print("Suma wiersza 2:", row_2_sum)
print("Suma wszystkich wierszy:", attn_weights.sum(dim=-1))
```

Oto wynik:

Suma wiersza 2: 1.0

Suma wszystkich wierszy: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])

W trzecim i ostatnim kroku pokazanym na rysunku 3.12 użyjemy wyznaczonych wag uwagi do obliczenia wszystkich wektorów kontekstu przez mnożenie macierzy:

```
all_context_vecs = attn_weights @ inputs  
print(all_context_vecs)
```

W wynikowym tensorze wyjściowym każdy wiersz zawiera trójwymiarowy wektor kontekstu:

```
tensor([[0.4421, 0.5931, 0.5790],  
       [0.4419, 0.6515, 0.5683],  
       [0.4431, 0.6496, 0.5671],  
       [0.4304, 0.6298, 0.5510],  
       [0.4671, 0.5910, 0.5266],  
       [0.4177, 0.6503, 0.5645]])
```

Aby się upewnić, że kod jest poprawny, można porównać drugi wiersz z wektorem kontekstu $z^{(2)}$, obliczonym w punkcie 3.3.1:

```
print("Poprzedni, 2. wektor kontekstu:", context_vec_2)
```

Na podstawie wyniku można zauważyc, że obliczony wcześniej context_vec_2 dokładnie odpowiada drugiemu wierszowi w poprzednim tensorze:

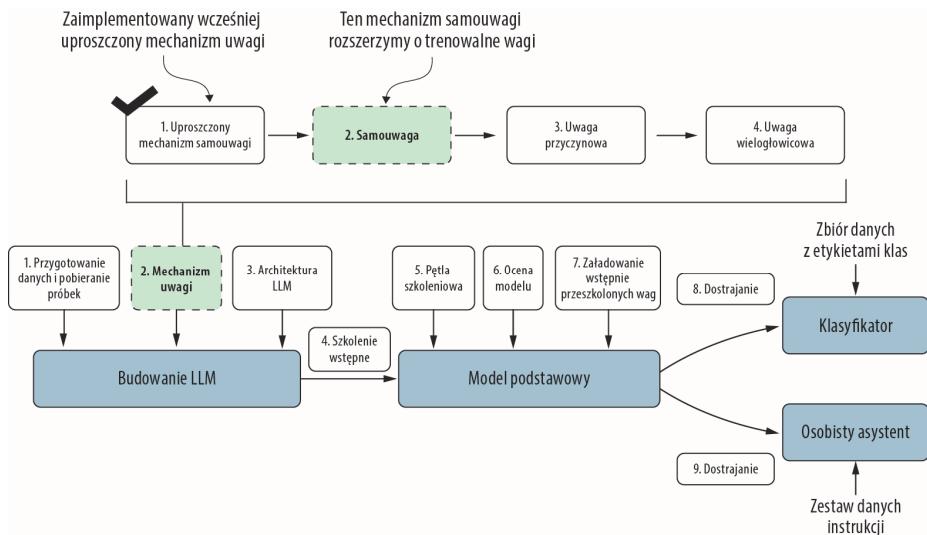
```
Poprzedni, 2. wektor kontekstu: tensor([0.4419, 0.6515, 0.5683])
```

Na tym kończy się przykład kodu prostego mechanizmu samouwagi. W kolejnym podrozdziale wprowadzimy trenowalne wagi, co umożliwi modelowi LLM uczenie się na podstawie danych i poprawę wydajności w określonych zadaniach.

3.4. Implementacja mechanizmu samouwagi z trenowalnymi wagami

W kolejnym kroku zaimplementujemy mechanizm samouwagi używany w architekturze *Original Transformer*, modelach GPT i większości innych popularnych modeli LLM. Taki mechanizm samouwagi jest również nazywany *skalowaną uwagą opartą na iloczynie skalarnym* (ang. *scaled dot-product attention*). Sposób wykorzystania takiego mechanizmu samouwagi w szerszym kontekście implementacji modelu LLM pokazano na rysunku 3.13.

Jak pokazałem na rysunku 3.13, mechanizm samouwagi z trenowalnymi wagami opiera się na poprzednich pojęciach: chcemy obliczać wektory kontekstu jako ważone



Rysunek 3.13. Aby zrozumieć podstawową mechanikę warstw uwagi, wcześniej zakodowaliśmy uproszczony mechanizm. Teraz do tego mechanizmu uwagi dodajemy trenowalne wagи. Później ten mechanizm samouwagi rozszerzymy przez dodanie maski przyczynowej i wprowadzenie wielu głów

sumy wektorów wejściowych specyficznych dla danego elementu wejścia. Jak można zauważać, różnice względem podstawowego mechanizmu samouwagi, który zakodowaliśmy wcześniej, są niewielkie.

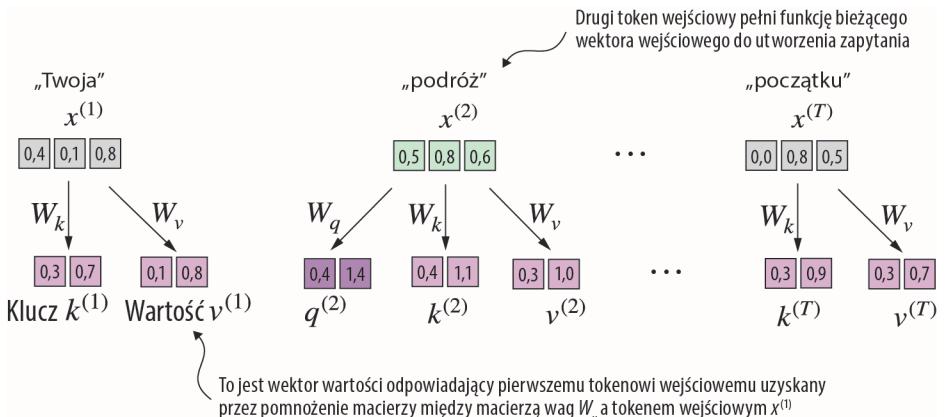
Najbardziej zauważalną różnicą jest wprowadzenie macierzy wag, które są aktualizowane podczas szkolenia modelu. Te trenowalne macierze wag są niezbędne, aby model (a konkretne: moduł uwagi wewnętrz modelu) mógł nauczyć się tworzenia „dobrych” wektorów kontekstu (szkoleniem modeli LLM zajmiemy się w rozdziale 5.).

Tym mechanizmem samouwagi zajmiemy się w dwóch punktach. Najpierw, tak jak wcześniej, zakodujemy go krok po kroku. Następnie zorganizujemy kod w zwięzlej klasie Pythona, którą można zimportować do architektury LLM.

3.4.1. Obliczanie wag uwagi krok po kroku

Zaimplementujmy mechanizm samouwagi krok po kroku przez wprowadzenie trzech trenowalnych macierzy wag: W_q , W_k , i W_v . Te trzy macierze są używane do rzutowania osadzonych tokenów wejściowych, $x^{(i)}$, odpowiednio na wektory zapytań, klucz i wartości (rysunek 3.14).

Drugi element wejścia $x^{(2)}$ zdefiniowaliśmy wcześniej jako zapytanie, gdy obliczaliśmy uproszczone wagи uwagi w celu obliczenia wektora kontekstu $z^{(2)}$. Następnie uogólniliśmy to pojęcie w celu obliczenia wszystkich wektorów kontekstu $z^{(1)} \dots z^{(T)}$ dla sześciowarzowego zdania wejściowego „Twoja podróż zaczyna się od początku”.



Rysunek 3.14. W pierwszym kroku implementacji mechanizmu samouwagi z trenowalnymi macierzami wag obliczamy wektory zapytania (q), klucza (k) i wartości (v) dla elementów wejściowych x . Podobnie jak w poprzednich podrozdziałach, drugie wejście, $x^{(2)}$, oznaczamy jako wejście zapytania. Wektor zapytań $q^{(2)}$ uzyskujemy przez pomnożenie macierzy wejść $x^{(2)}$ i macierzy wag W_q . W podobny sposób obliczamy wektory klucza i wartości przez mnożenie macierzy W_k i W_v .

W tym przypadku jest podobnie – w celach ilustracyjnych zaczynamy od obliczenia tylko jednego wektora kontekstu, $z^{(2)}$. Następnie zmodyfikujemy kod, aby obliczyć wszystkie wektory kontekstu.

Zaczniemy od zdefiniowania kilku zmiennych:

```

x_2 = inputs[1]
d_in = inputs.shape[1]
d_out = 2
    
```

Drugi element
wejścia

Rozmiar osadzenia wyjść, d_out=2

Należy pamiętać,
że indeksy, d=3

Należy zauważyć, że w modelach podobnych do GPT wymiary wejść i wyjść zwykle są takie same, ale aby lepiej śledzić obliczenia, w przykładzie użyjemy różnych wymiarów: dla wejść ($d_in=3$) i dla wyjść ($d_out=2$).

Następnie zainicjujemy trzy macierze wag: W_q , W_k , i W_v , pokazane na rysunku 3.14:

```

torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_key   = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
    
```

Aby ograniczyć w wynikach zbędne elementy, ustawiliśmy zmienną `requires_grad=False`, ale gdybyśmy chcieli użyć do szkolenia modelu macierzy wag, należałoby ustawić `requires_grad=True`, co pozwoliłoby na aktualizowanie tych macierzy podczas szkolenia modelu.

Następnie obliczamy wektory zapytania, klucza i wartości:

```

query_2 = x_2 @ W_query
key_2   = x_2 @ W_key
    
```

```
value_2 = x_2 @ W_value
print(query_2)
```

Wynikiem zapytania jest dwuwymiarowy wektor, ponieważ za pomocą zmiennej `d_out` ustawiliśmy liczbę kolumn odpowiedniej macierzy wag na 2:

```
tensor([0.4306, 1.4551])
```

Parametry wag a wagi uwagi

W macierzach wag W termin „waga” opisuje „parametry wag” — wartości sieci neuronowej, które są optymalizowane podczas szkolenia. Nie należy tego mylić z wagami uwagi. Jak widzeliśmy wcześniej, wagi uwagi określają zakres, w jakim wektor kontekstu zależy od różnych części danych wejściowych (tzn. w jakim stopniu sieć koncentruje się na różnych częściach wejścia).

Podsumowując, parametry wag są podstawowymi, wyuczonymi współczynnikami, które określają połączenia sieci, podczas gdy wagi uwagi są wartościami dynamicznymi, specyficznymi dla kontekstu.

Nawet jeśli tymczasowym celem jest obliczenie tylko jednego wektora kontekstu, $z^{(2)}$, nadal potrzebne są wektory klucza i wartości dla wszystkich elementów wejściowych, ponieważ są one wykorzystywane w obliczeniach wag uwagi w odniesieniu do zapytania $q^{(2)}$ (patrz rysunek 3.14).

Wszystkie klucze i wartości można uzyskać przez mnożenie macierzy:

```
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

Jak można stwierdzić na podstawie wyników, udało się zrzutować sześć tokenów wejściowych z trójwymiarowej przestrzeni osadzania na dwuwymiarową:

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

Drugi krok polega na obliczeniu ocen uwagi w sposób pokazany na rysunku 3.15. Najpierw obliczmy wynik uwagi ω_{22} :

```
keys_2 = keys[1]
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
```

Należy pamiętać, że indeksy tablic w Pythonie zaczynają się od 0

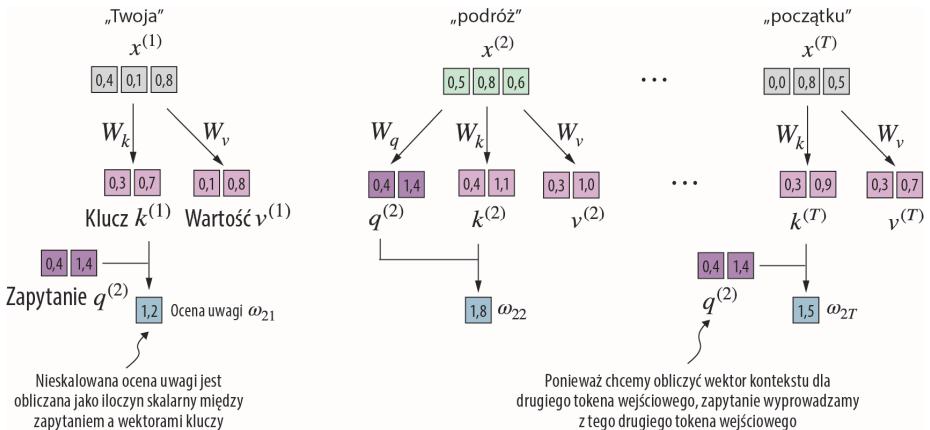
Wynik dla nieznormalizowanego wyniku uwagi jest następujący:

```
tensor(1.8524)
```

Tak jak pisalem wcześniej, mnożenie macierzy pozwala uogólnić obliczenia na wszystkie wyniki uwagi:

```
attn_scores_2 = query_2 @ keys.T
print(attn_scores_2)
```

Wszystkie wyniki uwagi dla zapytania



Rysunek 3.15. Obliczenie oceny uwagi sprawdza się do wyznaczenia iloczynu skalarnego. Podobny sposób zastosowaliśmy w uproszczonym mechanizmie samouwagi opisany w podrozdziale 3.3. Nowością jest brak bezpośredniego obliczania iloczynu skalarnego między elementami wejścia. Zamiast tego używamy zapytania i klucza uzyskanych przez przekształcenie wejść za pomocą odpowiednich macierzy wag

Jak można szybko sprawdzić, drugi element wyjścia pasuje do obliczonego wcześniej elementu `attn_score_22`:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Teraz chcemy przejść od ocen uwagi do wag uwagi (rysunek 3.16). Wagi uwagi obliczamy przez skalowanie ocen uwagi i użycie funkcji `softmax`. Teraz jednak skalujemy oceny uwagi przez podzielenie ich przez pierwiastek kwadratowy wymiaru osadzeń kluczów (pierwiastek kwadratowy jest matematycznie równoważny podniesieniu do potęgi 0,5):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

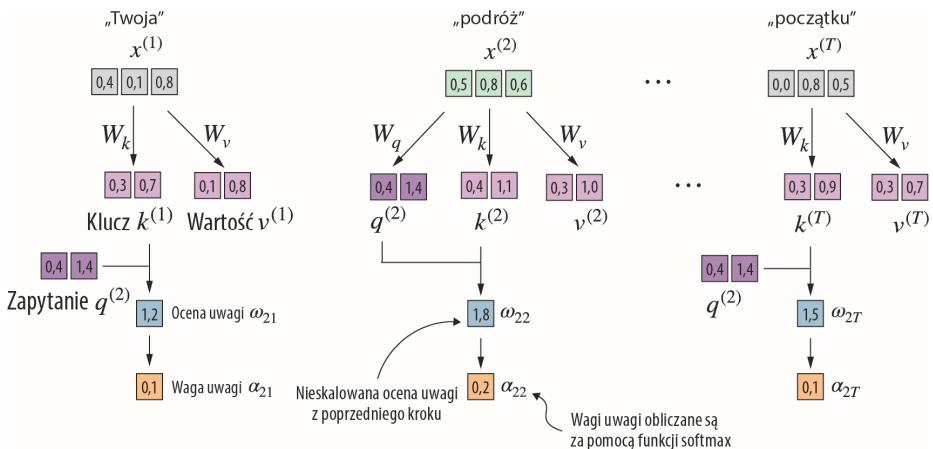
Oto uzyskane wagi uwagi:

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

Na koniec należy obliczyć wektory kontekstu (rysunek 3.17).

Podobnie jak wcześniej obliczyliśmy wektor kontekstu jako sumę ważoną wektorów wejściowych (patrz podrozdział 3.3), teraz obliczamy wektor kontekstu jako sumę ważoną wektorów wartości. W tym przypadku wagi uwagi służą jako czynnik opisujący znaczenie każdego wektora wartości. Tak jak poprzednio, aby uzyskać wynik w jednym kroku, można użyć mnożenia macierzy:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```



Rysunek 3.16. Następnym krokiem po obliczeniu wyników uwagi ω jest normalizacja wyników z użyciem funkcji softmax w celu uzyskania wag uwagi α

Skąd nazwa skalowana uwaga oparta na iloczynie skalarnym?

Powodem normalizacji według rozmiaru osadzeń jest eliminowanie niskich wartości gradientów, co wpływa na poprawę wydajności szkolenia. Na przykład przy skalowaniu w górę wymiaru osadzeń, który w modelach LLM podobnych do GPT jest zwykle większy niż 1000, wysokie wartości iloczynów skalarnych ze względu na zastosowaną do nich funkcję softmax mogą podczas propagacji wstecznej skutkować bardzo małymi gradientami. Wraz ze wzrostem wartości iloczynu skalarnego funkcja softmax zachowuje się bardziej jak funkcja progowa, co skutkuje gradientami bliskimi zera. Takie niewielkie gradienty mogą znaczco spowolnić naukę lub całkowicie zatrzymać proces szkolenia.

To właśnie skalowanie przez pierwiastek kwadratowy z wymiaru osadzeń dało nazwę mechanizmowi uwagi nazywanemu skalowaną uwagą opartą na iloczynie skalarnym (ang. *scaled-dot product*).

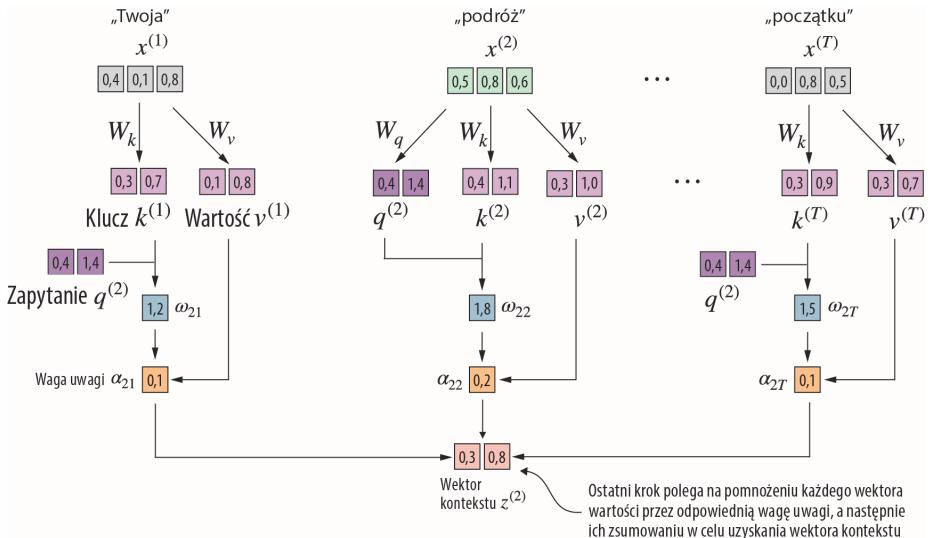
Zawartość wynikowego wektora jest następująca:

```
tensor([0.3061, 0.8210])
```

Do tej pory obliczyliśmy tylko jeden wektor kontekstu, $z^{(2)}$. Następnie uogólnimy kod, aby obliczyć wszystkie wektory kontekstu w sekwencji wejściowej, od $z^{(1)}$ do $z^{(T)}$.

3.4.2. Implementacja kompaktowej klasy samouwagi w Pythonie

Do tej pory omówiłem wiele czynności wymaganych do obliczenia wyjść samouwagi. Zrobiłem to głównie w celach ilustracyjnych, aby można było zrealizować zadanie krok po kroku. W praktyce, mając na uwadze implementację modelu LLM w następnym rozdziale, warto zorganizować ten kod w klasę Pythona, tak jak pokazałem na lisingu 3.1.



Rysunek 3.17. Na ostatnim etapie obliczeń samouwagi wyznaczamy wektor kontekstu przez połączenie wszystkich wektorów wartości za pomocą wag uwagi

Dlaczego zapytanie, klucz i wartość?

Terminy „klucz”, „zapytanie” i „wartość” w kontekście mechanizmów uwagi pochodzą z dziedziny wyszukiwania informacji i baz danych, w której podobnych określeń używa się w kontekście przechowywania, wyszukiwania i pobierania informacji.

Zapytanie (ang. *query*) jest analogiczne do zapytania służącego do wyszukiwania w bazie danych. Reprezentuje bieżący element (np. słowo lub token w zdaniu), na którym model się skupia lub który próbuje zrozumieć. Zapytanie jest używane do sprawdzania innych części sekwencji wejściowej w celu określenia, jaką część uwagi należy im poświęcić.

Klucz (ang. *key*) przypomina klucz w bazie danych używany do indeksowania i wyszukiwania. W mechanizmie uwagi z każdym elementem w sekwencji wejściowej (np. z każdym słowem w zdaniu) jest powiązany klucz. Te klucze są używane do dopasowania zapytania.

Wartość w tym kontekście przypomina wartość w parze klucz-wartość w bazie danych. Reprezentuje rzeczywistą zawartość lub reprezentację elementów wejściowych. Gdy model określi, które klucze (a tym samym które części wejścia) są najbardziej istotne dla zapytania (bieżącego elementu, na którym skupia się uwaga), pobiera odpowiednie wartości.

Listing 3.1. Kompaktowa klasa samouwagi

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))
```

```

def forward(self, x):
    keys = x @ self.W_key
    queries = x @ self.W_query
    values = x @ self.W_value
    attn_scores = queries @ keys.T # omega
    attn_weights = torch.softmax(
        attn_scores / keys.shape[-1]**0.5, dim=-1
    )
    context_vec = attn_weights @ values
    return context_vec

```

W powyższym kodzie, opartym na bibliotece PyTorch, `SelfAttention_v1` jest klasą pochodną klasy `nn.Module` — podstawowego bloku budulcowego modeli PyTorch, dostarczającego niezbędnych funkcji do tworzenia warstw modelu i zarządzania nimi.

Metoda `__init__` inicjuje trenowalne macierze wag (`W_query`, `W_key` i `W_value`) dla zapytań, kluczy i wartości. Każda z nich przekształca wejście o wymiarze `d_in` na wyjście o wymiarze `d_out`.

Podczas propagacji w przód za pomocą metody `forward` obliczamy wyniki uwagi (`attn_scores`) przez pomnożenie zapytań i kluczy, a następnie normalizację tych wyników z użyciem funkcji `softmax`. Na koniec tworzymy wektor kontekstu, który jest kombinacją wartości pomnożonych przez odpowiadające im znormalizowane oceny uwagi.

Zaimplementowaną powyżej klasę można wykorzystać w następujący sposób:

```

torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))

```

Ponieważ `inputs` zawiera sześć wektorów osadzeń, uzyskujemy macierz przechowującą sześć wektorów kontekstu:

```

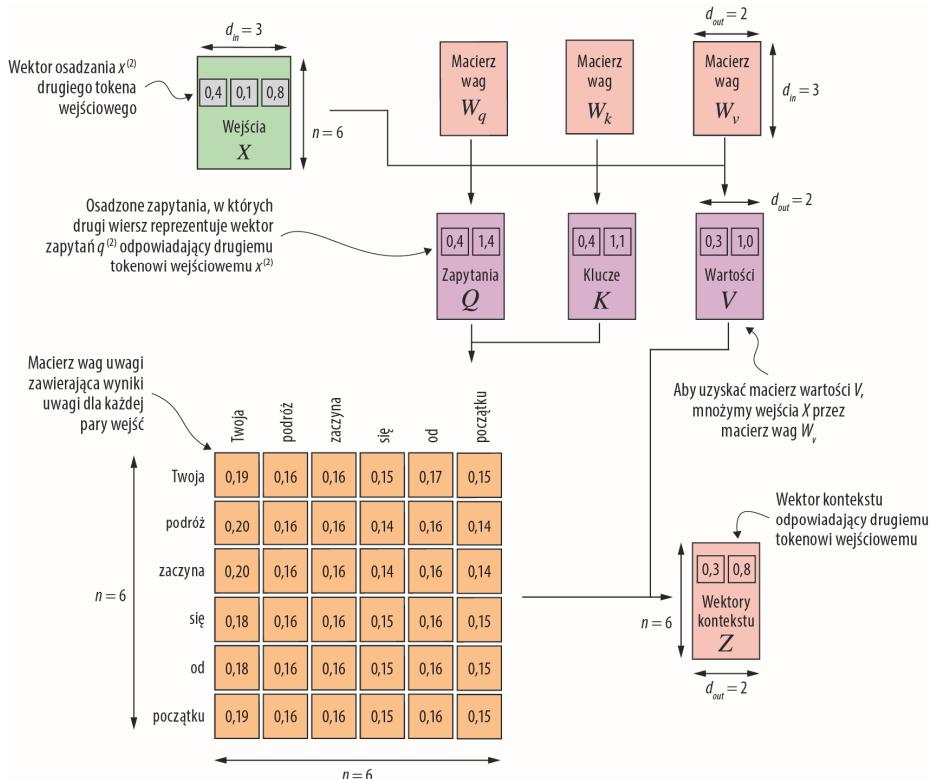
tensor([[0.2996,  0.8053],
       [0.3061,  0.8210],
       [0.3058,  0.8203],
       [0.2948,  0.7939],
       [0.2927,  0.7891],
       [0.2990,  0.8040]], grad_fn=<MmBackward0>)

```

W ramach szybkiego sprawdzenia zwróć uwagę, że drugi wiersz ([0.3061, 0.8210]) pasuje do zawartości `context_vec_2` z poprzedniego punktu. Zaimplementowany mechanizm samouwagi podsumowałem na rysunku 3.18.

Samouwaga obejmuje trenowalne macierze wag W_q , W_k , i W_v . Te macierze przekształcają dane wejściowe na, odpowiednio, zapytania, klucze i wartości — najważniejsze elementy mechanizmu uwagi. Gdy model podczas szkolenia korzysta z większej ilości danych, dostosowuje te trenowalne wagi. Przekonasz się o tym w kolejnych rozdziałach.

Implementację klasy `SelfAttention_v1` można ulepszyć dzięki wykorzystaniu warstwy biblioteki PyTorch `nn.Linear`, która skutecznie przeprowadza mnożenie macierzy, gdy elementy stronniczości (ang. *bias elements*) są wyłączone. Dodatkową, znaną zaletą korzystania z `nn.Linear` zamiast ręcznej implementacji `nn.Parameter`



Rysunek 3.18. W warstwie samouwagi przekształcamy wektory wejściowe w wejściowej macierzy X za pomocą trzech macierzy wag: W_q , W_k i W_v . Na ich podstawie obliczamy macierz wag uwagi, wykorzystując wynikowe zapytania (Q) i klucze (K). Następnie, używając wag uwagi i wartości (V), obliczamy wektory kontekstu (Z). Dla wizualnej przejrzystości skupiam się na pojedynczym tekście wejściowym z n tokenami, a nie na partii wielu tekstów. W związku z tym trójwymiarowy tensor wejściowy jest w tym kontekście uproszczony do dwuwymiarowej macierzy. Takie podejście umożliwia prostszą wizualizację i pozwala lepiej zrozumieć zachodzące procesy. W celu zachowania spójności z późniejszymi rysunkami wartości w macierzy uwagi nie przedstawiają rzeczywistych wag uwagi (aby uzyskać większą zwięzłość, liczby na tym rysunku przycięto do dwóch cyfr po przecinku. Wartości w każdym wierszu powinny sumować się do 1,0, czyli do 100%).

\hookrightarrow (`torch.rand(...)`) jest to, że `nn.Linear` ma zoptymalizowany schemat inicjalizacji wag, co przyczynia się do bardziej stabilnego i skutecznego uczenia modelu (listing 3.2).

Listing 3.2. Klasa samouwagi wykorzystująca warstwy liniowe biblioteki PyTorch

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
```

```

self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

def forward(self, x):
    keys = self.W_key(x)
    queries = self.W_query(x)
    values = self.W_value(x)
    attn_scores = queries @ keys.T
    attn_weights = torch.softmax(
        attn_scores / keys.shape[-1]**0.5, dim=-1
    )
    context_vec = attn_weights @ values
    return context_vec

```

Klasy `SelfAttention_v2` można używać podobnie jak klasy `SelfAttention_v1`:

```

torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))

```

Oto wynik działania tego kodu:

```

tensor([[-0.0739,  0.0713],
       [-0.0748,  0.0703],
       [-0.0749,  0.0702],
       [-0.0760,  0.0685],
       [-0.0763,  0.0679],
       [-0.0754,  0.0693]], grad_fn=<MmBackward0>)

```

Warto zwrócić uwagę, że klasy `SelfAttention_v1` i `SelfAttention_v2` dają różne wyniki. To dlatego, że używają różnych wag początkowych dla macierzy wag — `nn.Linear` stosuje bardziej złożony schemat inicjalizacji wag.

Ćwiczenie 3.1. Porównanie klas `SelfAttention_v1` z `SelfAttention_v2`

Zauważ, że `nn.Linear` w klasie `SelfAttention_v2` stosuje inny schemat inicjalizacji wag niż `nn.Parameter(torch.rand(d_in, d_out))` używany w `SelfAttention_v1`. Z tego powodu te dwa mechanizmy zwracają różne wyniki. Aby sprawdzić, czy obie implementacje, `SelfAttention_v1` i `SelfAttention_v2`, są podobne, można przenieść macierze wag z obiektu `SelfAttention_v2` do `SelfAttention_v1`, tak aby obydwa obiekty zwracały takie same wyniki.

Twoim zadaniem jest poprawne przypisanie wag z egzemplarza klasy `SelfAttention_v2` do egzemplarza klasy `SelfAttention_v1`. Aby to zrobić, trzeba zrozumieć związek między wagami w obu wersjach (wskazówka: `nn.Linear` przechowuje macierz wag w postaci transponowanej). Po wykonaniu zadania powinieneś zauważyć, że oba egzemplarze dają takie same wyniki.

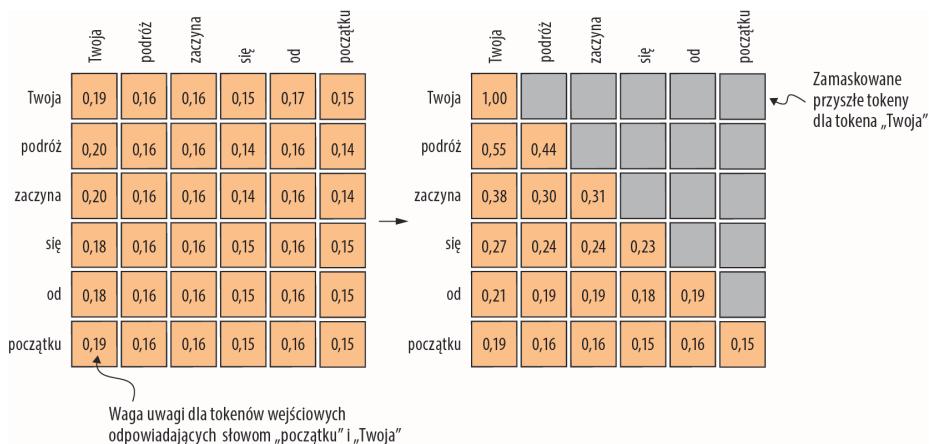
W kolejnym kroku wprowadzimy ulepszenia do mechanizmu samouwagi. W szczególności skoncentrujemy się na włączeniu elementów przyczynowo-skutkowych i wielu głowic. Aspekt przyczynowy obejmuje zmodyfikowanie mechanizmu uwagi tak, aby uniemożliwić modelowi dostęp do przyszłych informacji w sekwencji. Ma to kluczowe znaczenie w takich zadaniach jak modelowanie języka, w których każde słowo powinno zależeć tylko od poprzednich słów.

Zastosowanie komponentu wielogłowicowego polega na podzieleniu mechanizmu uwagi na wiele „głowic”. Każda głowica uczy się różnych aspektów danych, co pozwala modelowi jednocześnie zwracać uwagę na informacje pochodzące z różnych podprzestrzeni reprezentacji, w różnych pozycjach. Poprawia to wydajność modelu podczas rozwiązywania złożonych zadań.

3.5. *Ukrywanie przyszłych słów dzięki zastosowaniu uwagi przyczynowej*

W przypadku wielu zadań wykonywanych przez modele LLM mechanizm samouwagi w trakcie przewidywania następnego tokena w sekwencji uwzględnia tylko te tokeny, które występują przed bieżącą pozycją. Uwaga przyczynowa, nazywana również *uwagą maskowaną* (ang. *masked attention*), jest specjalną formą samouwagi. Ogranicza model tak, aby podczas przetwarzania dowolnego tokena przy obliczaniu wyników uwagi uwzględniał jedynie wcześniejsze i bieżące wejścia w sekwencji. Jest to podejście różne od standardowego mechanizmu samouwagi, który umożliwia dostęp do całej sekwencji wejściowej.

W kolejnym kroku zmodyfikujemy standardowy mechanizm samouwagi, aby stworzyć mechanizm *uwagi przyczynowej*, niezbędny do opracowania modeli LLM w kolejnych rozdziałach. Aby to uzyskać w modelach LLM podobnych do GPT, dla każdego przetwarzanego tokena tokeny przyszłe, czyli te, które występują w tekście wejściowym po tokenie bieżącym, należy zamaskować (rysunek 3.19).



Rysunek 3.19. W przypadku uwagi przyczynowej wagi uwagi są maskowane powyżej przekątnej. Dzięki temu model LLM podczas obliczania wektorów kontekstu z wykorzystaniem wag uwagi nie ma dostępu do przyszłych tokenów. Na przykład w przypadku słowa „podróż” w drugim wierszu uwzględniane są jedynie wagi uwagi dla słów występujących wcześniej („Twoja”) oraz dla słowa na bieżącej pozycji („podróż”)

Maskujemy wagi uwagi powyżej przekątnej i normalizujemy je bez maskowania tak, aby w każdym wierszu sumowały się do 1. W dalszej części książki zaimplementujemy tę procedurę maskowania i normalizacji w kodzie.

3.5.1. Wykorzystanie maski uwagi przyczynowej

Kolejny krok to zaimplementowanie w kodzie maski uwagi przyczynowej. Aby zaimplementować kroki mające na celu zastosowanie maski uwagi przyczynowej w celu uzyskania wag uwagi maskowanej, jak pokazałem na rysunku 3.20, skorzystamy z ocen i wag uwagi obliczonych w poprzednim punkcie i zakodujemy mechanizm uwagi przyczynowej.



Rysunek 3.20. Jednym ze sposobów uzyskania macierzy wag maskowanej uwagi w mechanizmie uwagi przyczynowej jest zastosowanie funkcji softmax do ocen uwagi, wyzerowanie elementów powyżej przekątnej i znormalizowanie wynikowej macierzy

W pierwszym kroku, tak jak wcześniej, obliczamy wagi uwagi za pomocą funkcji *softmax*:

```

queries = sa_v2.W_query(inputs) ←
keys = sa_v2.W_key(inputs)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
print(attn_weights)
  
```

Dla wygody ponownie wykorzystujemy macierze wag zapytań i kluczy obiektu *SelfAttention_v2* obliczone w poprzednim punkcie

Skutkuje to uzyskaniem następujących wag uwagi:

```

tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
       [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
       [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
       [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
       [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]], grad_fn=<SoftmaxBackward0>)
  
```

Drugi krok można zaimplementować za pomocą funkcji *tril* z biblioteki PyTorch, co pozwoli utworzyć maskę, w której wartości powyżej przekątnej mają wartość zero:

```

context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
  
```

Uzyskana maska to:

```
tensor([[1., 0., 0., 0., 0., 0.],
       [1., 1., 0., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.],
       [1., 1., 1., 1., 0., 0.],
       [1., 1., 1., 1., 1., 0.],
       [1., 1., 1., 1., 1., 1.]])
```

Teraz można pomnożyć tę maskę przez wagi uwagi, co spowoduje wyzerowanie wartości powyżej przekątnej:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

Jak można zauważyc, elementy powyżej przekątnej zostały wyzerowane:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
       [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
       [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
      grad_fn=<MulBackward0>)
```

Trzeci krok polega na ponownej normalizacji wag uwagi tak, aby w każdym wierszu znów sumowały się do 1. Można to osiągnąć przez podzielenie każdego elementu w każdym wierszu przez sumę w każdym wierszu:

```
row_sums = masked_simple.sum(dim=-1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

W wyniku uzyskujemy macierz wag uwagi, w której wagi uwagi powyżej przekątnej mają wartość zero, a suma elementów w każdym wierszu wynosi 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
       [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
       [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
      grad_fn=<DivBackward0>)
```

W tym momencie można by zakończyć implementację przyczynowej uwagi. Wciąż jednak można ją ulepszyć. Weźmy pod uwagę matematyczną właściwość funkcji *softmax* i spróbujmy stworzyć bardziej wydajną implementację obliczeń wag zamaskowanej uwagi, w mniejszej liczbie kroków (rysunek 3.21).

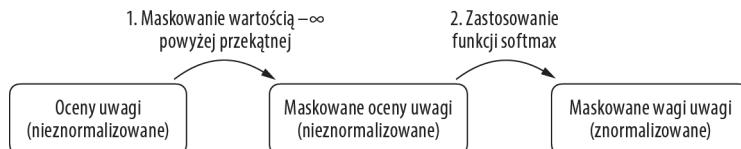
Funkcja *softmax* przekształca dane wejściowe w rozkład prawdopodobieństwa. Gdy w wierszu występują wartości oznaczone jako minus nieskończoność ($-\infty$), funkcja *softmax* interpretuje je jako prawdopodobieństwo zerowe (jest to uzasadnione matematycznie tym, że $e^{-\infty}$ zmierza do 0).

Wyciek informacji

Po zastosowaniu maski, a następnie ponownej normalizacji wag uwagi początkowo może się wydawać, że informacje pochodzące z przyszłych tokenów (tych, które zamierzamy zamaskować) nadal mogą wpływać na bieżący token — ich wartości są bowiem uwzględniane w obliczeniach softmax. Trzeba jednak pamiętać, że ponowna normalizacja wag uwagi po maskowaniu w istocie skutkuje ponownym obliczeniem funkcji softmax na mniejszym podzbiorze (ponieważ zamaskowane pozycje nie mają wpływu na wartość funkcji softmax).

Matematyczna elegancja funkcji softmax polega na tym, że pomimo początkowego uwzględnienia wszystkich pozycji w mianowniku efekt zamaskowanych pozycji w wyniku zastosowania maski i ponownej normalizacji zostaje zniwelowany — nie mają one znaczącego wpływu na wynik funkcji softmax.

Mówiąc prościej, rozkład wag uwagi po zastosowaniu maski i ponownej normalizacji jest taki, jakby był obliczany tylko z uwzględnieniem pozycji niezamaskowanych. W ten sposób zyskujemy gwarancję, że nie dojdzie do wycieku informacji z przyszłych (lub w inny sposób zamaskowanych) tokenów, co jest zgodne z zamierzeniami.



Rysunek 3.21. Wydajniejszy sposób uzyskania macierzy maskowanych wag uwagi w przypadku uwagi przyczynowej polega na zamaskowaniu ocen uwagi przed zastosowaniem funkcji softmax wartościami minus nieskończoność

Tę bardziej wydajną „sztuczkę” maskowania można zaimplementować przez utworzenie maski z jedynkami powyżej przekątnej, a następnie zastąpienie tych jedynek wartościami minus nieskończoność ($-\inf$):

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

W wyniku uzyskamy następującą maskę:

```
tensor([[0.2899, -inf, -inf, -inf, -inf, -inf],
        [0.4656, 0.1723, -inf, -inf, -inf, -inf],
        [0.4594, 0.1703, 0.1731, -inf, -inf, -inf],
        [0.2642, 0.1024, 0.1036, 0.0186, -inf, -inf],
        [0.2183, 0.0874, 0.0882, 0.0177, 0.0786, -inf],
        [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
      grad_fn=<MaskedFillBackward0>)
```

Teraz trzeba tylko zastosować do tych zamaskowanych wyników funkcję *softmax*:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

Jak można zauważyć na podstawie uzyskanych wyników, wartości w każdym wierszu sumują się do 1 i dalsza normalizacja nie jest konieczna:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
       [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
       [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
       [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
       [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],  
grad_fn=<SoftmaxBackward0>)
```

Można teraz użyć zmodyfikowanych wag uwagi do obliczenia wektorów kontekstu za pomocą instrukcji `context_vec = attn_weights @ values`, tak jak w podrozdziale 3.4. Najpierw jednak omówię inną drobną modyfikację mechanizmu uwagi przyczynowej, która jest przydatna w trakcie uczenia modelu LLM, ponieważ pozwala zmniejszyć ryzyko nadmiernego dopasowania.

3.5.2. Maskowanie dodatkowych wag uwagi z użyciem dropoutu

Dropout w uczeniu głębokim to technika, w której losowo wybrane jednostki ukrytej warstwy są podczas uczenia ignorowane, co sprowadza się do ich „odrzucania” (ang. *dropping*). Zastosowanie tej metody pomaga zapobiegać nadmierнемu dopasowaniu, ponieważ dzięki odrzuceniu wybranych elementów model nie stanie się nadmiernie zależny od określonego zbioru jednostek warstwy ukrytej. Ważne jest, aby podkreślić, że dropout jest używany tylko w trakcie szkolenia, a po jego zakończeniu jest wyłączany.

W architekturze transformera, włącznie z takimi modelami jak GPT, dropout w mechanizmie uwagi zwykle stosuje się w dwóch specyficznych momentach: po obliczeniu wag uwagi lub po zastosowaniu wag uwagi do wektorów wartości. W tym przykładzie zastosujemy maskę dropoutu po obliczeniu wag uwagi, tak jak pokazano na rysunku 3.22, ponieważ w praktyce ten wariant stosuje się częściej.

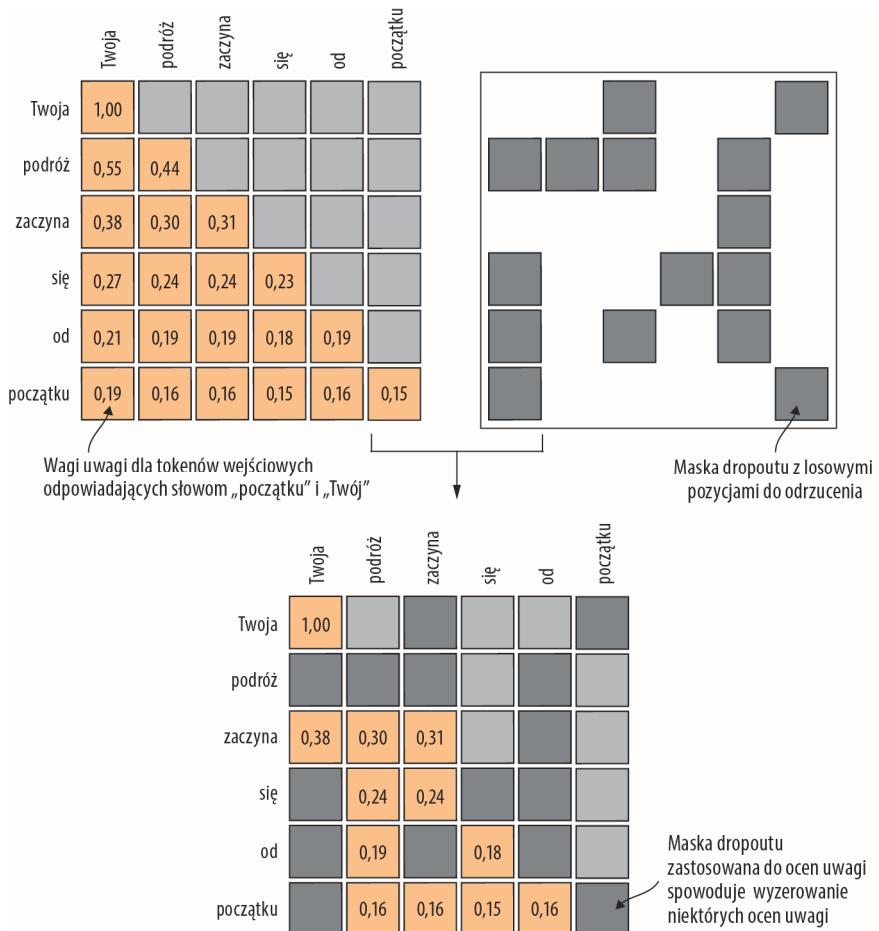
W poniższym przykładzie kodu zastosowano współczynnik dropoutu równy 50%, co oznacza maskowanie połowy wag uwagi (w dalszych rozdziałach w trakcie szkolenia modelu GPT użyjemy niższej wartości współczynnika dropoutu, np. 0,1 lub 0,2). Dla uproszczenia zastosujemy najpierw implementację dropoutu z biblioteki PyTorch do tensora 6×6 składającego się z samych jedynek:

```
torch.manual_seed(123)  
dropout = torch.nn.Dropout(0.5) ← Wybieramy współczynnik dropoutu  
example = torch.ones(6, 6) ← na poziomie 50%.  
print(dropout(example)) ← Tutaj tworzymy macierz składającą się z jedynek.
```

Jak można zauważyć, mniej więcej połowa wartości jest wyzerowana:

```
tensor([[2., 2., 0., 2., 2., 0.],  
       [0., 0., 0., 2., 0., 2.],  
       [2., 2., 2., 2., 0., 2.],  
       [0., 2., 2., 0., 0., 2.],  
       [0., 2., 0., 2., 0., 2.],  
       [0., 2., 2., 2., 0., 2.]])
```

Jeśli dropout stosuje się do macierzy wag uwagi ze współczynnikiem 50%, połowa elementów macierzy jest losowo ustawiana na zero. Aby zrekompensować redukcję



Rysunek 3.22. Za pomocą maski uwagi przyczynowej (lewy górnny róg) stosujemy dodatkową maskę dropoutu (prawy górnny róg), tak aby dzięki wyzerowaniu dodatkowych wag uwagi zminimalizować podczas szkolenia problemy nadmiernego dopasowania

aktywnych elementów, wartości pozostałych elementów w macierzy skaluje się w górę z użyciem współczynnika $1 / 0,5 = 2$. To skalowanie ma kluczowe znaczenie dla utrzymania ogólnej równowagi wag uwagi. Dzięki temu zyskujemy gwarancję spójności przeciętnego wpływu mechanizmu uwagi zarówno w fazie szkolenia, jak i podczas wnioskowania.

Teraz zastosujmy dropout do macierzy wag uwagi:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

Wynikowa macierz wag uwagi ma teraz wyzerowane dodatkowe elementy i przeskakowane pozostałe jedynki:

```
tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],  
       [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],  
       [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],  
       [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],  
       [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],  
       [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],  
      grad_fn=<MulBackward0>)
```

Warto zwrócić uwagę, że wynikowe wyjścia dropoutu mogą wyglądać inaczej w zależności od systemu operacyjnego; więcej o tej niespójności można przeczytać na stronie PyTorch issue tracker pod adresem <https://github.com/pytorch/pytorch/issues/121595>.

Po omówieniu uwagi przyczynowej i maskowania z użyciem dropoutu możemy opracować zwięzłą klasę Pythona. Celem tej klasy jest ułatwienie skutecznego stosowania tych dwóch technik.

3.5.3. Implementacja zwięzkiej klasy przyczynowej uwagi

W tym punkcie zastosujemy modyfikacje uwagi przyczynowej i dropoutu do klasy Pythona `SelfAttention`, którą opracowaliśmy w podrozdziale 3.4. Ta klasa posłuży następnie jako szablon do opracowania *uwagi wielogłowicowej* – ostatniej klasy uwagi, którą zaimplementujemy.

Wcześniej jednak sprawdzimy, czy kod może obsługiwać partie składające się z więcej niż jednego wejścia, tak aby klasa `CausalAttention` obsługiwała partie wyjść generowane przez program ładujący dane, zaimplementowany w rozdziale 2.

Dla uproszczenia, aby zasymulować partie wejścia, powielimy przykładowy tekst wejściowy:

```
batch = torch.stack((inputs, inputs), dim=0) ← Dwa wejścia z sześcioma tokenami każde;  
print(batch.shape)                            każdy token ma wymiar osadzeń równy 3.
```

W wyniku uzyskamy trójwymiarowy tensor składający się z dwóch tekstów wejściowych, o sześciu tokenach każdy, przy czym każdy token jest trójwymiarowym wektorem osadzeń:

```
torch.Size([2, 6, 3])
```

Poniższa klasa `CausalAttention` jest podobna do zaimplementowanej wcześniej klasy `SelfAttention`. Różnica polega na dodaniu komponentu dropout i maski przyczynowej (listing 3.3).

Listing 3.3. Kompaktowa klasa uwagi przyczynowej

```
class CausalAttention(nn.Module):  
    def __init__(self, d_in, d_out, context_length,  
                 dropout, qkv_bias=False):  
        super().__init__()  
        self.d_out = d_out  
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)  
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
```

```

self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias) ←
self.dropout = nn.Dropout(dropout)
self.register_buffer(
    'mask',
    torch.triu(torch.ones(context_length, context_length),
    diagonal=1) ← Dodaliśmy również wywołanie register_buffer
) ← (więcej informacji znajdziesz w dalszej części tekstu).

```

W porównaniu z poprzednią klasą SelfAttention_v1 dodaliśmy warstwę dropout.


```

def forward(self, x):
    b, num_tokens, d_in = x.shape ← Transponujemy wymiary 1 i 2,
    keys = self.W_key(x)           zachowując wymiar parti na pierwszej pozycji (0).
    queries = self.W_query(x)
    values = self.W_value(x)

    attn_scores = queries @ keys.transpose(1, 2) ←
    attn_scores.masked_fill_(← Działania z końcowym podkreśleniem
        self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
    attn_weights = torch.softmax(
        attn_scores / keys.shape[-1]**0.5, dim=-1
    )
    attn_weights = self.dropout(attn_weights)

    context_vec = attn_weights @ values
    return context_vec

```

w bibliotece PyTorch są wykonywane w miejscu, co pozwala uniknąć niepotrzebnego kopowania pamięci.

O ile wszystkie dodane wiersze kodu powinny być w tym momencie jasne, o tyle warto zwrócić uwagę na wywołanie `self.register_buffer()` w metodzie `__init__`. Użycie wywołania `register_buffer` w PyTorch nie jest bezwzględnie konieczne we wszystkich przypadkach użycia, ale pozwala uzyskać kilka korzyści. Na przykład, gdy w modelu LLM używamy klasy `CausalAttention`, wraz z modelem są automatycznie przenoszone bufory na odpowiednie urządzenie (CPU lub GPU). Jest to bardzo istotne podczas szkolenia modelu LLM. To oznacza, że nie ma potrzeby ręcznego sprawdzania, czy te tensory znajdują się na tym samym urządzeniu, co parametry modelu. Dzięki temu można uniknąć błędów wynikających z niedopasowania urządzeń.

Klasę `CausalAttention` można wykorzystać w podobny sposób jak użyta wcześniej klasę `SelfAttention`:

```

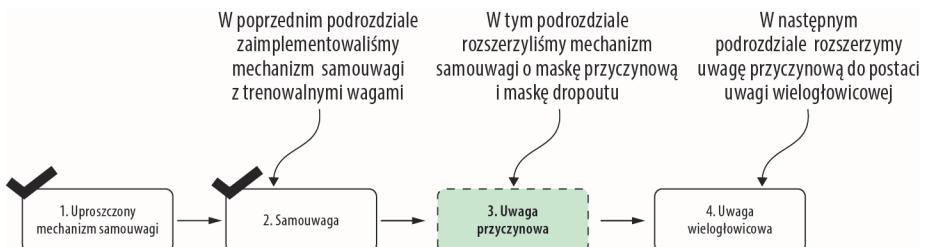
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)

```

Wynikowy wektor kontekstu jest trójwymiarowym tensorem, w którym każdy token jest reprezentowany przez dwuwymiarowe osadzenie:

```
context_vecs.shape: torch.Size([2, 6, 2])
```

Dotychczasowe dokonania podsumowano na rysunku 3.23. Skupiliśmy się na pojęciu uwagi przyczynowej w sieciach neuronowych i jej implementacji. Następnie rozwinimy to pojęcie i zakodujemy wielogłowicowy moduł uwagi, który równolegle zaimplementuje kilka mechanizmów uwagi przyczynowej.



Rysunek 3.23. Zaczeliśmy od uproszczonego mechanizmu uwagi, dodaliśmy trenowalne wagi, a następnie przyczynową maskę uwagi. W kolejnym podrozdziale rozszerzymy mechanizm uwagi przyczynowej i zakodujemy uwagę wielogłowicową, którą wykorzystamy w tworzonym modelu LLM

3.6. Rozszerzenie uwagi jednogłowicowej na wielogłowicową

Ostatni krok tworzenia mechanizmu uwagi polega na rozszerzeniu wcześniej zaimplementowanej klasy uwagi przyczynowej na wiele głowic. Taki mechanizm uwagi nazywa się również *uwagą wielogłowicową*.

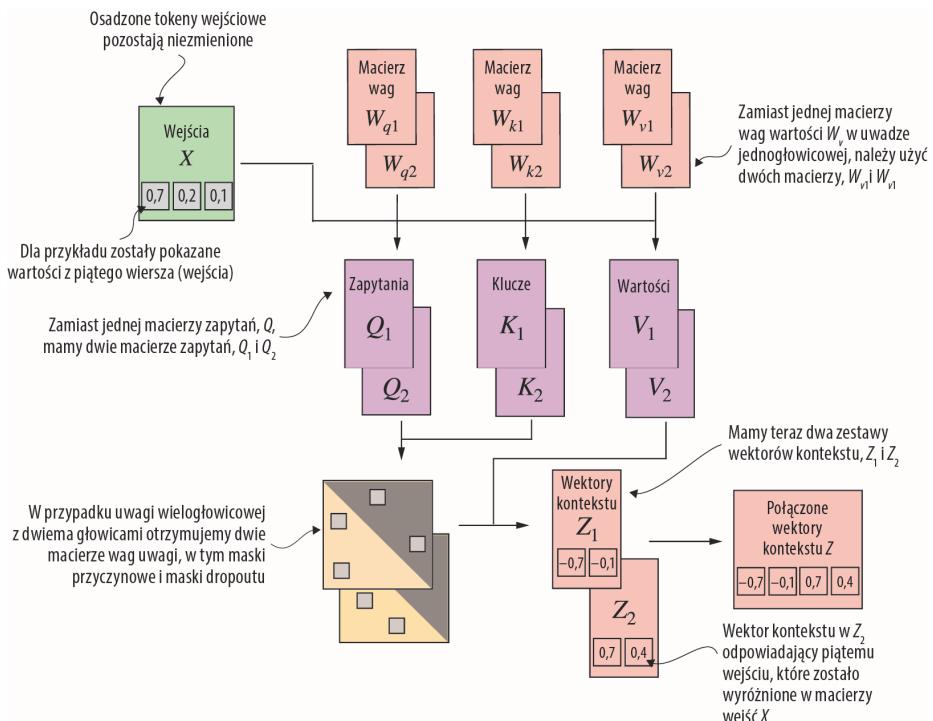
Termin „wielogłowicowa” odnosi się do podziału mechanizmu uwagi na wiele części (głowice), z których każda działa niezależnie. W tym kontekście pojedynczy moduł uwagi przyczynowej można uznać za uwagę jednogłowicową, w której do sekwencyjnego przetwarzania wejść wykorzystywany jest tylko jeden zbiór wag uwagi.

W tym podrozdziale zajmiemy się rozszerzeniem uwagi przyczynowej na uwagę wielogłowicową. Najpierw, przez połączenie wielu modułów CausalAttention, w intuicyjny sposób zbudujemy wielogłowicowy moduł uwagi. Następnie ten sam wielogłowicowy moduł uwagi zaimplementujemy w sposób bardziej złożony, ale też bardziej wydajny obliczeniowo.

3.6.1. Utworzenie stosu wielu jednogłowicowych warstw uwagi

W kategoriach praktycznych implementacja uwagi wielogłowicowej sprowadza się do utworzenia wielu egzemplarzy mechanizmu samouwagi (rysunek 3.18), z których każdy ma własne wagi, a następnie na połączeniu ich wyników. Korzystanie z wielu egzemplarzy mechanizmu samouwagi może oznaczać wykonywanie bardzo złożonych obliczeń, ale ma kluczowe znaczenie dla rodzaju złożonego rozpoznawania wzorców, z którego znane są modele LLM oparte na transformerach.

Strukturę wielogłowicowego modułu uwagi, składającego się z wielu ułożonych w stos jednogłowicowych modułów uwagi, przedstawionych wcześniej na rysunku 3.18, zilustrowano na rysunku 3.24.



Rysunek 3.24. Wielogłowicowy moduł uwagi obejmuje dwa jednogłowicowe moduły uwagi ułożone w stos. Tak więc zamiast używać do obliczania macierzy wartości pojedynczej macierzy, W_v , w wielogłowicowym module uwagi z dwiema głowicami wykorzystujemy dwie macierze wag wartości, W_{v1} i W_{v2} . To samo dotyczy innych macierzy wag, W_q i W_k . W ten sposób otrzymujemy dwa zbiory wektorów kontekstowych, Z_1 i Z_2 , które można połączyć w jedną macierz wektorów kontekstowych, Z

Jak wspomniano wcześniej, główną ideą, na której opiera się działanie uwagi wielogłowicowej, jest wielokrotne uruchamianie mechanizmu uwagi (równolegle) z różnymi wyuczonymi projekcjami liniowymi — wynikami mnożenia danych wejść (takich jak wektory zapytania, klucza i wartości w mechanizmach uwagi) przez macierz wag. W kodzie można to osiągnąć przez zaimplementowanie prostej klasy `MultiHeadAttentionWrapper`, która tworzy wiele egzemplarzy zaimplementowanego wcześniej modułu `CausalAttention` (listing 3.4).

Listing 3.4. Klasa opakowująca do implementacji wielogłowicowej uwagi

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(
                d_in, d_out, context_length, dropout, qkv_bias
```

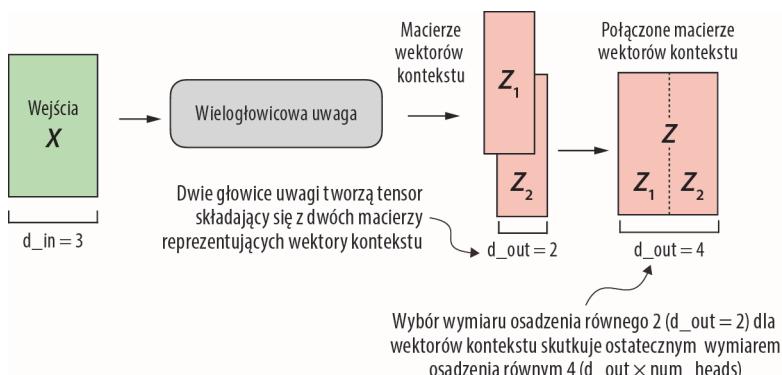
```

        )
    for _ in range(num_heads)]
)

def forward(self, x):
    return torch.cat([head(x) for head in self.heads], dim=-1)

```

Na przykład, jeśli użyjemy klasy `MultiHeadAttentionWrapper` z dwiema głowicami uwagi (dzięki ustawieniu `num_heads=2`) i wymiarem wyjściowym `CausalAttention d_out=2`, otrzymamy czterowymiarowy wektor kontekstu ($d_{out} \times num_heads = 4$), tak jak pokazano na rysunku 3.25.



Rysunek 3.25. Za pomocą klasy `MultiHeadAttentionWrapper` określiliśmy liczbę głowic uwagi (`num_heads`). Jeśli ustawimy `num_heads=2`, tak jak w tym przykładzie, otrzymamy tensor z dwoma zbiorami macierzy wektorów kontekstu. W każdej macierzy wektora kontekstu wiersze reprezentują wektory kontekstu odpowiadające tokenom, a kolumny odpowiadają wymiarowi osadzeń określonym przez `d_out=4`. Macierze wektorów kontekstu łączymy według wymiaru kolumny. Ponieważ mamy dwie głowice uwagi, a wymiar osadzeń wynosi 2, ostateczny wymiar osadzeń wynosi $2 \cdot 2 = 4$

Aby lepiej to zilustrować, na konkretnym przykładzie, możemy użyć klasy `MultiHeadAttentionWrapper` podobnej do wcześniejszej klasy `CausalAttention`:

```

torch.manual_seed(123)
context_length = batch.shape[1] # To jest liczba tokenów
d_in, d_out = 3, 2
mha = MultiHeadAttentionWrapper(
    d_in, d_out, context_length, 0.0, num_heads=2
)
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)

```

W wyniku działania tego kodu otrzymamy następujący tensor reprezentujący wektory kontekstu:

```

tensor([[-0.4519,  0.2216,  0.4772,  0.1063],
       [-0.5874,  0.0058,  0.5891,  0.3257],
       [-0.6300, -0.0632,  0.6202,  0.3860],
       [-0.5675, -0.0843,  0.5478,  0.3589],
       [-0.5526, -0.0981,  0.5321,  0.3428],
       [-0.5299, -0.1081,  0.5077,  0.3493]],

      [[-0.4519,  0.2216,  0.4772,  0.1063],
       [-0.5874,  0.0058,  0.5891,  0.3257],
       [-0.6300, -0.0632,  0.6202,  0.3860],
       [-0.5675, -0.0843,  0.5478,  0.3589],
       [-0.5526, -0.0981,  0.5321,  0.3428],
       [-0.5299, -0.1081,  0.5077,  0.3493]]], grad_fn=<CatBackward0>
→context_vecs.shape: torch.Size([2, 6, 4])

```

Pierwszy wymiar wynikowego tensora `context_vecs` wynosi 2, ponieważ mamy dwa teksty wejściowe (teksty wejściowe są powielone, dlatego odpowiadające im wektory kontekstu są dokładnie takie same). Drugi wymiar odnosi się do sześciu tokenów w każdym wejściu. Trzeci wymiar dotyczy czterowymiarowego osadzenia każdego tokena.

Ćwiczenie 3.2. Obliczenia dwuwymiarowych wektorów osadzeń

Zmień argumenty wejściowe dla wywołania `MultiHeadAttentionWrapper(..., num_heads=2)` tak, aby przy zachowaniu ustawienia `num_heads=2` wyjściowe wektory kontekstu były dwuwymiarowe zamiast czterowymiarowe. Wskazówka: nie musisz modyfikować implementacji klasy; wystarczy zmienić jeden z pozostałych argumentów wejściowych.

Dotychczas zaimplementowaliśmy klasę `MultiHeadAttentionWrapper`, która łączyła wiele jednogłowicowych modułów uwagi. Były one jednak przetwarzane sekwencyjnie, za pomocą instrukcji `[head(x) for head in self.heads]` w metodzie `forward`. Tę implementację można ulepszyć przez przetwarzanie głowic równolegle. Jednym ze sposobów osiągnięcia tego celu jest jednoczesne obliczanie wyników dla wszystkich głowic uwagi przez mnożenie macierzy.

3.6.2. Implementacja uwagi wielogłowicowej z podziałem wag

Wcześniej stworzyliśmy klasę `MultiHeadAttentionWrapper` jako implementację wielogłowicowej uwagi przez połączenie w stos wielu jednogłowicowych modułów uwagi. W tym celu utworzyliśmy kilka egzemplarzy obiektów `CausalAttention` i je połączyczyliśmy.

Zamiast utrzymywać dwie oddzielne klasy, `MultiHeadAttentionWrapper` i `CausalAttention`, można zaimplementować te pojedynczo w jedną klasę `MultiHeadAttention`. Ponadto oprócz połączenia klasy `MultiHeadAttentionWrapper` z klasą `CausalAttention` wprowadzimy kilka innych modyfikacji, których celem będzie poprawa wydajności implementacji wielogłowicowej uwagi.

W klasie `MultiHeadAttentionWrapper` zaimplementowaliśmy wiele głowic przez utworzenie listy obiektów `CausalAttention` (`self.heads`), z których każdy reprezentuje oddzielną głowicę uwagi. Klasa `CausalAttention` implementuje mechanizm uwagi niezależnie, a wyniki z poszczególnych głowic są ze sobą łączone. Dla odróżnienia – zamieszczona poniżej klasa `MultiHeadAttention` integruje funkcjonalność wielu głowic w ramach jednej klasy. Dzieli dane wejściowe na wiele głowic przez przekształcenie rzutowanych tensorów zapytań, kluczy i wartości, a następnie, po obliczeniu uwagi, łączy wyniki z tych głowic.

Przed dokładnym omówieniem implementacji klasy `MultiHeadAttention` przyjrzyjmy się jej kodowi (listing 3.5):

Listing 3.5. Wydajna wielogłowicowa klasa uwagi

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
        values = values.view(b, num_tokens, self.num_heads, self.head_dim)
        queries = queries.view(
            b, num_tokens, self.num_heads, self.head_dim
        )
        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        attn_scores = queries @ keys.transpose(2, 3)
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]
        attn_scores.masked_fill_(mask_bool, -torch.inf)
```

Dzięki dodaniu wymiaru `num_heads` niejawnie dzielimy macierz. Następnie rozwijamy ostatni wymiar: $(b, \text{num_tokens}, d_{\text{out}}) \rightarrow (b, \text{num_tokens}, \text{num_heads}, \text{head_dim})$.

Obliczenie iloczynu skalarnego dla każdej głowicy

Zmniejszenie wymiarów rzutowania w celu dopasowania do żądanego wymiaru wyjścia

Kształt tensora: $(b, \text{num_tokens}, d_{\text{out}})$

Wykorzystanie warstwy liniowej do połączenia wyjść głowic

Transpozycja z kształtu $(b, \text{num_tokens}, \text{num_heads}, \text{head_dim})$ na $(b, \text{num_heads}, \text{num_tokens}, \text{head_dim})$

Maski przyjęte do liczby tokenów

Wykorzystanie maski do wypełnienia ocen uwagi

```

attn_weights = torch.softmax(
    attn_scores / keys.shape[-1]**0.5, dim=-1)
attn_weights = self.dropout(attn_weights)

context_vec = (attn_weights @ values).transpose(1, 2) ←
Połączenie głowic,
przy czym self.d_out
= self.num_heads *
self.head_dim →
context_vec = context_vec.contiguous().view(
    b, num_tokens, self.d_out
)
context_vec = self.out_proj(context_vec) ← Dodaje opcjonalną projekcję liniową
return context_vec
    
```

Kształt tensora:
(b, num_tokens,
n_heads,
head_dim)

Mimo że zmiana kształtu (.view) i transpozycja (.transpose) tensorów wewnętrz klasy MultiHeadAttention wygląda na operacje bardzo złożone matematycznie, klasa MultiHeadAttention implementuje to samo pojęcie, co zaimplementowana wcześniej klasa MultiHeadAttentionWrapper.

Na poziomie ogólnym, w zaimplementowanej wcześniej klasie MultiHeadAttentionWrapper, utworzyliśmy stos złożony z wielu warstw jednogłowicowych uwag, które łącznie utworzyły wielogłowicową warstwę uwagi. Do klasy MultiHeadAttention zastosowaliśmy podejście zintegrowane. Punktrem wyjścia jest warstwa wielogłowicowa, która następnie wewnętrznie dzieli się na pojedyncze głowice uwagi (rysunek 3.26).

Podział tensorów zapytania, kluczy i wartości uzyskujemy przez zmianę kształtu tensora i operacje transpozycji za pomocą metod .view i .transpose z biblioteki PyTorch. Wejście jest najpierw poddawane transformacji (za pomocą liniowych warstw dla zapytań, kluczy i wartości), po czym wykonywana jest zmiana kształtu w celu utworzenia reprezentacji wielu głowic.

Najważniejsza operacja polega na podzieleniu wymiaru `d_out` na `num_heads` i `head_dim`, gdzie `head_dim = d_out / num_heads`. Ten podział osiągamy następnie za pomocą metody `.view`: tensor o wymiarach (b, num_tokens, d_out) jest przekształcany do wymiaru (b, num_tokens, num_heads, head_dim).

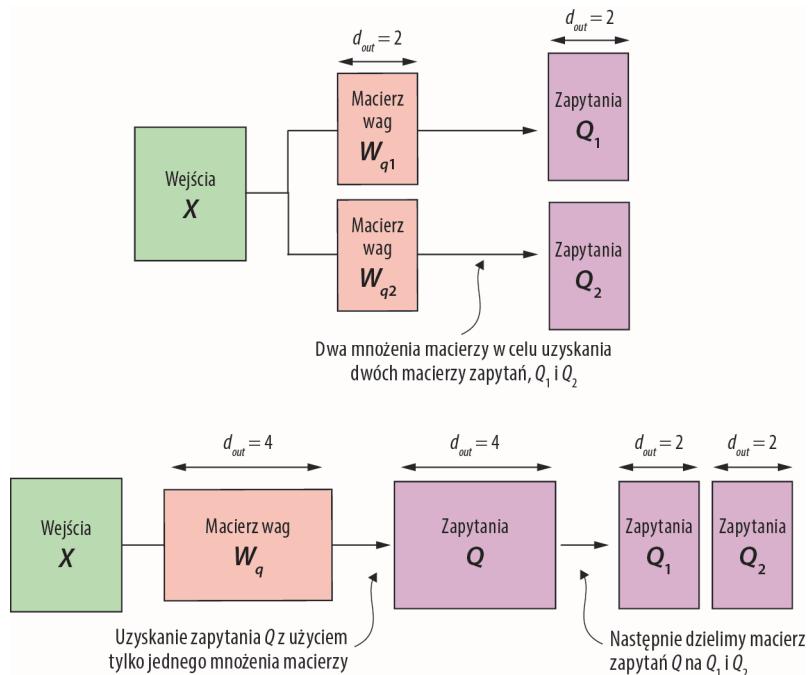
Następnie, w celu przeniesienia wymiaru `num_heads` przed wymiar `num_tokens`, wykonujemy transpozycję tensorów. W efekcie uzyskujemy kształt (b, num_heads, num_tokens, head_dim). Ta transpozycja ma kluczowe znaczenie dla właściwego wyrównania zapytań, kluczy i wartości w różnych głowicach oraz wydajności mnożenia macierzy parti.

Aby zilustrować grupowe mnożenie macierzy parti, założymy, że mamy następujący tensor:

```

a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573], ←
    [0.8993, 0.0390, 0.9268, 0.7388],
    [0.7179, 0.7058, 0.9156, 0.4340]],
    [[0.0772, 0.3565, 0.1479, 0.5331],
    [0.4066, 0.2318, 0.4545, 0.9737],
    [0.4606, 0.5159, 0.4220, 0.5786]]])
    
```

Kształt tego tensora to (b, num_heads,
num_tokens, head_dim) = (1, 2, 3, 4).



Rysunek 3.26. W klasie `MultiHeadAttentionWrapper` z dwiema głowicami uwagi zainicjalizowaliśmy dwie macierze wag, W_{q1} i W_{q2} , po czym obliczyliśmy dwie macierze zapytań, Q_1 i Q_2 (górną część rysunku). W klasie `MultiheadAttention` inicjujemy jedną większą macierz wag W_q , wykonujemy tylko jedno mnożenie macierzy z wejściami, aby uzyskać macierz zapytań Q , a następnie dzielimy macierz zapytań na Q_1 i Q_2 (dolna część rysunku). Te same zadania wykonujemy dla kluczy i wartości, które ze względu na czytelność rysunku nie są wyświetlane

Teraz wykonujemy mnożenie macierzy partiionowane pomiędzy samym tensorem a widokiem tensora, w którym dwa ostatnie wymiary, `num_tokens` i `head_dim`, zostały przetransponowane:

```
print(a @ a.transpose(2, 3))
```

Oto wynik:

```
tensor([[[[1.3208, 1.1631, 1.2879],
          [1.1631, 2.2150, 1.8424],
          [1.2879, 1.8424, 2.0402]],

         [[0.4391, 0.7003, 0.5903],
          [0.7003, 1.3737, 1.0620],
          [0.5903, 1.0620, 0.9912]]]])
```

W tym przypadku implementacja mnożenia macierzy w bibliotece PyTorch obsługuje czterowymiarowy tensor wejściowy w taki sposób, że mnożenie macierzy jest wykonywane między dwoma ostatnimi wymiarami (`num_tokens`, `head_dim`), a następnie powtarzane dla pojedynczych głowic.

Na przykład bardziej kompaktowy sposób obliczania mnożenia macierzy dla każdej głowicy osobno może zapewnić następujący kod:

```
first_head = a[0, 0, :, :]
first_res = first_head @ first_head.T
print("Pierwsza głowica:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nDruga głowica:\n", second_res)
```

Wyniki są dokładnie takie same jak te, które uzyskaliśmy przy mnożeniu macierzy partii. Oto wynik działania instrukcji `print(a @ a.transpose(2, 3))`:

```
Pierwsza głowica:
tensor([[1.3208, 1.1631, 1.2879],
       [1.1631, 2.2150, 1.8424],
       [1.2879, 1.8424, 2.0402]])

Druga głowica:
tensor([[0.4391, 0.7003, 0.5903],
       [0.7003, 1.3737, 1.0620],
       [0.5903, 1.0620, 0.9912]])
```

Oto dalszy opis klasy `MultiHeadAttention`: po obliczeniu wag uwagi i wektorów kontekstu wektory kontekstu ze wszystkich głowic są transponowane z powrotem do kształtu (`b, num_tokens, num_heads, head_dim`). Następnie te wektory są przekształcane (spłaszczone) do kształtu (`b, num_tokens, d_out`), co sprowadza się do połączenia wyjść ze wszystkich głowic.

Ponadto po połączeniu głowic dodaliśmy do klasy `MultiHeadAttention` wyjściową warstwę projekcji (`self.out_proj`), która nie była obecna w klasie `CausalAttention`. Ta wyjściowa warstwa projekcji nie jest bezwzględnie konieczna (więcej szczegółów znajdziesz w „Dodatku B”), ale stosuje się ją w wielu architekturach modeli LLM, dlatego dodałem ją w tej implementacji, tak aby opis był kompletny.

Mimo że ze względu na dodatkowe przekształcenia i transpozycję tensorów klasa `MultiHeadAttention` sprawia wrażenie bardziej złożonej niż klasa `MultiHeadAttentionWrapper`, jest ona bardziej wydajna. Wynika to stąd, że do obliczenia kluczy wystarczy jedno mnożenie macierzy, na przykład `keys = self.W_key(x)` (to samo dotyczy macierzy zapytań i wartości). W klasie `MultiHeadAttentionWrapper` trzeba było powtórzyć to mnożenie macierzy, a w przypadku każdej głowicy uwagi to mnożenie wiąże się z największym kosztem obliczeniowym.

Klasy `MultiHeadAttention` można używać podobnie do zaimplementowanych wcześniej klas `SelfAttention` i `CausalAttention`:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
```

```
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

Wyniki dowodzą, że na wymiar wyjściowy bezpośrednio wpływa argument `d_out`:

```
tensor([[0.3190, 0.4858],
       [0.2943, 0.3897],
       [0.2856, 0.3593],
       [0.2693, 0.3873],
       [0.2639, 0.3928],
       [0.2575, 0.4028]],
      [[0.3190, 0.4858],
       [0.2943, 0.3897],
       [0.2856, 0.3593],
       [0.2693, 0.3873],
       [0.2639, 0.3928],
       [0.2575, 0.4028]]], grad_fn=<ViewBackward0>) context_vecs.shape:
→torch.Size([2, 6, 2])
```

Zaimplementowaliśmy klasę `MultiHeadAttention`, której będziemy używać podczas implementowania i szkolenia modelu LLM. Należy pamiętać, że chociaż ten kod jest w pełni funkcjonalny, to w celu zapewnienia czytelności wyników użyłem stosunkowo niskich rozmiarów osadzeń i niewielkiej liczby głowic uwagi.

Dla porównania, najmniejszy model GPT-2 (117 milionów parametrów) ma 12 głowic uwagi, a rozmiar osadzeń wektora kontekstu wynosi 768. Największy model GPT-2 (1,5 miliarda parametrów) ma 25 głowic uwagi, a rozmiar osadzeń wektora kontekstu wynosi 1600. Rozmiary osadzeń tokenów wejściowych i osadzeń kontekstu są w modelach GPT takie same (`d_in = d_out`).

Ćwiczenie 3.3. Inicjalizacja modułów uwagi o rozmiarze GPT-2

Korzystając z klasy `MultiHeadAttention`, zainicjuj wielogłowicowy moduł uwagi, który ma taką samą liczbę głowic uwagi jak najmniejszy model GPT-2 (12 głowic uwagi). Upewnij się również, że używasz właściwych, podobnych do stosowanego w modelu GPT-2, rozmiarów osadzeń wejściowych i wyjściowych (768 wymiarów). Warto zwrócić uwagę, że najmniejszy model GPT-2 obsługuje kontekst o rozmiarze 1024 tokenów.

Podsumowanie

- Mechanizmy uwagi transformują elementy wejściowe w rozszerzone reprezentacje wektorów kontekstu, które zawierają informacje dotyczące wszystkich wejść.
- Mechanizm samouwagi oblicza reprezentację wektora kontekstu jako ważoną sumę wejść.
- W uproszczonym mechanizmie uwagi wagi są obliczane za pomocą iloczynów skalarnych.

- Iloczyn skalarny to zwięzły sposób mnożenia dwóch wektorów element po elemencie, a następnie sumowania iloczynów.
- Mnożenie macierzy, choć nie jest ściśle wymagane, dzięki zastąpieniu zagnieżdżonych pętli `for` pomaga zaimplementować obliczenia w sposób bardziej wydajny i zwięzły.
- W stosowanych w modelach LLM mechanizmach samouwagi, zwanych również skalowanymi mechanizmami uwagi opartymi na iloczynie skalarnym, w obliczeniach pośrednich transformacji wejść: zapytań, wartości i kluczy, uwzględniamy trenowalne macierze wag.
- Podczas pracy z modelami LLM, które czytają i generują tekst od lewej do prawej, aby uniemożliwić modelowi LLM dostęp do przyszłych tokenów, dodajemy maskę uwagi przyczynowej.
- Oprócz masek uwagi przyczynowej w celu wyzerowania wag uwagi można dodać maskę dropoutu, której zastosowanie pozwala zmniejszyć problemy nadmiernego dopasowania modelu LLM.
- Moduły uwagi w modelach LLM opartych na transformerach obejmują wiele egzemplarzy uwagi przyczynowej, co określamy jako uwagę wielogłowicową.
- Wielogłowicowy moduł uwagi można utworzyć przez utworzenie stosu złożonego z wielu egzemplarzy modułów uwagi przyczynowej.
- Bardziej wydajny sposób tworzenia wielogłowicowych modułów uwagi obejmuje mnożenie macierzy partiami.

Implementacja od podstaw modelu GPT do generowania tekstu

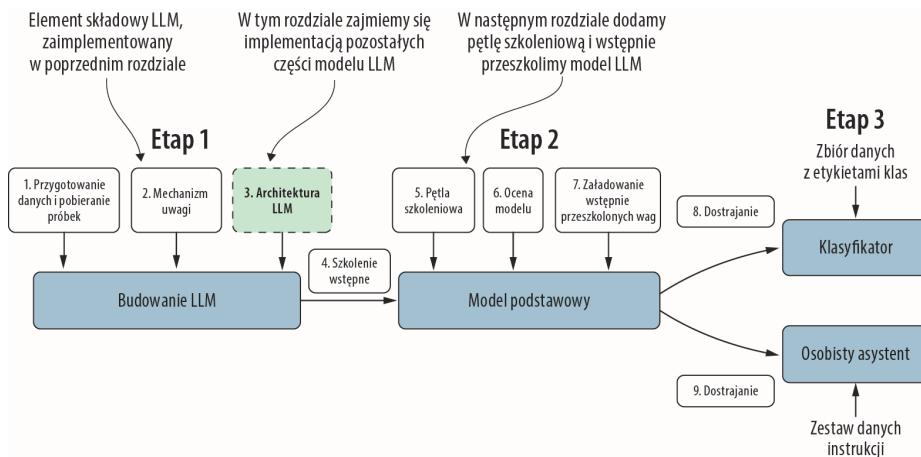
W tym rozdziale:

- Kodowanie dużego modelu językowego (LLM) podobnego do GPT, który można przeszkościć pod kątem generowania tekstu w języku podobnym do ludzkiego
- Normalizacja aktywacji warstw w celu ustabilizowania szkolenia sieci neuronowej
- Dodawanie połączeń skrótowych w głębokich sieciach neuronowych
- Implementacja bloków transformera w celu tworzenia modeli GPT o różnych rozmiarach
- Obliczanie liczby parametrów i wymagań dotyczących przechowywania modeli GPT

W poprzednich rozdziałach zapoznałeś się z jednym z podstawowych bloków budulcowych modelu LLM – *mechanizmem wielogłowicowej uwagi*, i dowiedziałeś się, jak go zakodować. W tym rozdziale pokażę Ci, jak zakodować inne elementy składowe modeli LLM. Za ich pomocą stworzysz model podobny do GPT, który będziesz szkolić w następnym rozdziale, pod kątem generowania tekstu w języku podobnym do ludzkiego¹.

¹ Ze względu na różnice w tokenizacji słów w językach angielskim i polskim w przykładach zachowano oryginalne teksty. Dzięki temu śledzenie procesu szkolenia modelu LLM będzie łatwiejsze – *przyp. tłum.*

Architektura LLM przedstawiona na rysunku 4.1 składa się z kilku bloków konstrukcyjnych. Najpierw opiszę widok architektury modelu góra-dół, a następnie przejdę do bardziej szczegółowego omówienia poszczególnych komponentów.



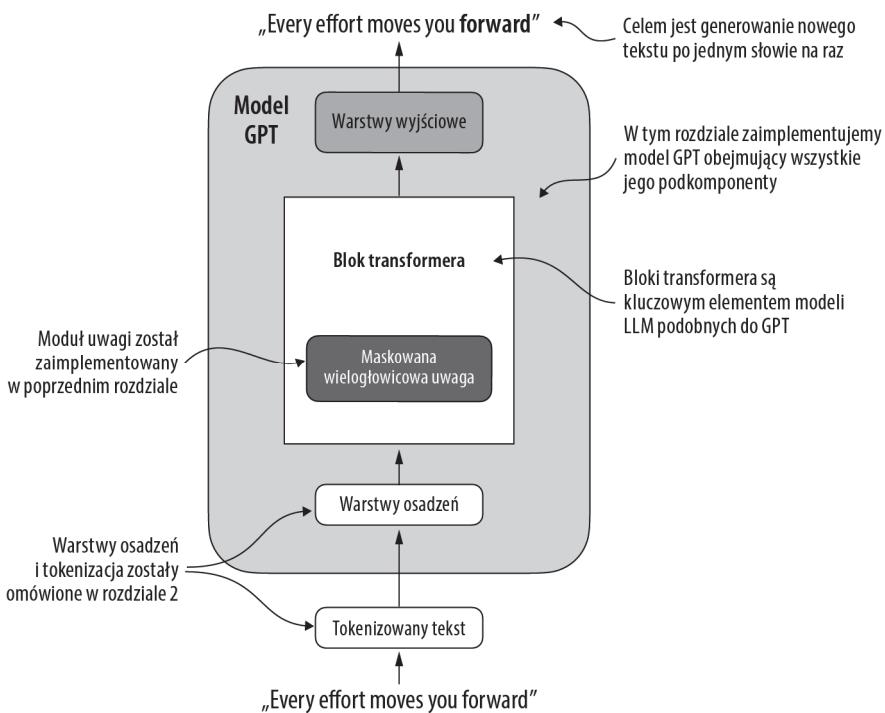
Rysunek 4.1. Trzy główne etapy kodowania modelu LLM. Ten rozdział koncentruje się na kroku 3. etapu 1. — implementacji architektury LLM

4.1. Kodowanie architektury LLM

Modele LLM podobne do GPT (ang. *generative pretrained transformer*) to rozbudowane architektury głębokich sieci neuronowych zaprojektowane do generowania nowego tekstu po jednym słowie (lub tokenie) na raz. Pomimo rozmiaru architektura modelu jest mniej skomplikowana, niż można by sądzić, ponieważ — jak przekonasz się później — wiele jej elementów się powtarza. Widok góra-dół architektury modelu LLM podobnego do GPT, z wyróżnionymi jej głównymi składowymi, przedstawiłem na rysunku 4.2.

Dotąd omówiłem kilka aspektów architektury LLM, takich jak tokenizacja i osadzanie danych wejściowych oraz maskowany moduł wielogłowicowej uwagi. W tym rozdziale zaimplementujemy od podstaw strukturę modelu GPT, w tym *bloki transformerów*, które później przeszkołymy pod kątem generowania tekstu w języku przypominającym ludzki.

Wcześniej, dla uproszczenia, używałem mniejszych wymiarów osadzeń, dzięki czemu ilustracje przykładów i pojęć można było wygodnie zmieścić na jednej stronie. Teraz przeskakuję komponenty „w górę”, do rozmiaru małego modelu GPT-2 — konkretnie do opisanej w pracy Radfora i współpracowników *Language Models Are Unsupervised Multitask Learner* jego najmniejszej (obejmującej 124 miliony parametrów) wersji — <https://mng.bz/yoBq>. Zwrót uwagę, że chociaż w raporcie pierwotnie wspomniano o 117 milionach parametrów, później to poprawiono. W rozdziale



Rysunek 4.2. Model GPT. Oprócz warstw osadzania obejmuje jeden lub większą liczbę bloków transformera, zawierających zaimplementowany wcześniej maskowany moduł wielogłowicowej uwagi

6. skupię się na załadowaniu do stworzonej implementacji wstępnie przeszkolonych wag i dostosowaniu ich do większych modeli GPT-2, zawierających, odpowiednio, 345, 762 i 1542 miliony parametrów.

W kategoriach uczenia głębokiego i modeli LLM, takich jak GPT, termin „parametry” odnosi się do trenowalnych wag modelu. W istocie te wagi są wewnętrznymi zmiennymi modelu, dostosowywanymi i optymalizowanymi podczas procesu uczenia w celu zminimalizowania określonej funkcji strat. Ta optymalizacja pozwala modelowi uczyć się na podstawie danych szkoleniowych.

Na przykład w warstwie sieci neuronowej reprezentowanej przez macierz (lub tensor) wag o wymiarach 2048×2048 parametrami są wszystkie elementy tej macierzy. Ponieważ istnieje 2048 wierszy i 2048 kolumn, całkowita liczba parametrów w tej warstwie wynosi 2048 razy 2048, co daje liczbę 4 194 304 parametrów.

Konfigurację małego modelu GPT-2 określmy za pomocą poniższego słownika Pythona, którego użyjemy w dalszych przykładach kodu:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,      # Rozmiar słownictwa
    "context_length": 1024,   # Rozmiar kontekstu
    "emb_dim": 768,          # Wymiar osadzenia
```

GPT-2 kontra GPT-3

Zwróć uwagę, że w tym rozdziale skupiam się na modelu GPT-2, ponieważ dla tego modelu OpenAI udostępniło publicznie wagę wstępnie przeszkolonego modelu. Załadujemy je do implementacji modelu LLM w rozdziale 6. Model GPT-3 w kategoriach architektury jest zasadniczo taki sam, z wyjątkiem tego, że przeskakowano go z 1,5 miliarda parametrów (GPT-2) do 175 miliardów parametrów (GPT-3) i przeskakowano na większej ilości danych. W chwili gdy pisałem te słowa, wagi dla modelu GPT-3 nie były publicznie dostępne. Model GPT-2 jest również lepszym wyborem do nauki implementowania modeli LLM, ponieważ można go uruchomić na zwykłym laptopie. W przeciwieństwie do niego model GPT-3 wymaga do uczenia i wnioskowania klastra GPU. Według Lambda Labs (<https://lambda-labs.com/>) przeszkolenie modelu GPT-3 na dostępnym w centrach danych pojedynczym układzie GPU V100 zajęłoby 355 lat, a na konsumenckim procesorze GPU RTX 8000 – 665 lat.

```

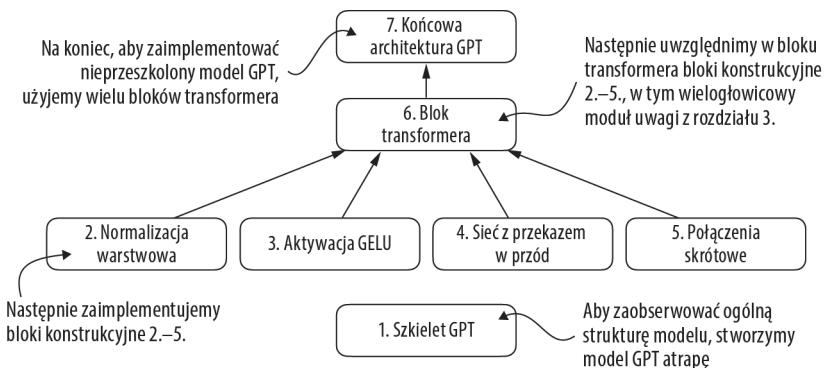
    "n_heads": 12,           # Liczba głowic uwagi
    "n_layers": 12,          # Liczba warstw
    "drop_rate": 0.1,         # Wskaźnik dropoutu
    "qkv_bias": False        # Stronniczość Zapytanie-Klucz-Wartość
}

```

W słowniku `GPT_CONFIG_124M` dla przejrzystości i w celu uniknięcia długich wierszy kodu zastosowałem zwięzłe nazwy zmiennych:

- `vocab_size` odnosi się do słownika zawierającego 50 257 słów, używanego przez tokenizator BPE (patrz rozdział 2.).
- `context_length` oznacza maksymalną liczbę tokenów wejściowych, które model może obsłużyć za pomocą osadzeń pozycyjnych (patrz rozdział 2.).
- `emb_dim` reprezentuje rozmiar osadzenia, pozwalający na transformację każdego tokena w wektor o 768 wymiarach.
- `n_heads` oznacza liczbę głowic uwagi w mechanizmie uwagi wielogłowicowej (patrz rozdział 3.).
- `n_layers` określa liczbę bloków transformera w modelu, czym zajmiemy się w dalszej części rozdziału.
- `drop_rate` wskazuje intensywność mechanizmu dropoutu (0,1 oznacza losowe odrzucanie 10% ukrytych jednostek), co ma zapobiec nadmiernemu dopasowaniu (patrz rozdział 3.).
- `qkv_bias` określa, czy w obliczeniach zapytań, kluczy i wartości w warstwach Linear uwagi wielogłowicowej uwzględnić wektor stronniczości. Początkowo, zgodnie z normami nowoczesnego LLM, wyłączymy tę funkcję, ale wróćmy do niej w rozdziale 6., gdy do implementowanego modelu będziemy ładować wstępnie przeskakowane wagi GPT-2 z modelu OpenAI.

Korzystając z tej konfiguracji, zaimplementujemy atrapę architektury GPT (DummyGPT → Model), którą pokazałem na rysunku 4.3. Dzięki analizie tego rysunku zyskasz ogólny



Rysunek 4.3. Kolejność kodowania architektury GPT. Zaczynamy od szkieletu GPT, atrapy architektury, po czym przechodzimy do poszczególnych elementów podstawowych i na koniec tworzymy z nich blok transformera do wykorzystania w ostatecznej architekturze GPT

obraz tego, jak poszczególne elementy do siebie pasują i jakie inne komponenty trzeba zakodować, aby stworzyć pełną architekturę modelu GPT.

Ponumerowane pola na rysunku 4.3 pokazują kolejność, w jakiej wykonamy poszczególne kroki niezbędne do zakodowania ostatecznej architektury GPT. Zaczniemy od kroku 1. — stworzenia szkieletu modelu GPT, któremu nadamy nazwę `DummyGPT` → `Model` (listing 4.1).

Listing 4.1. Klasa reprezentująca atrapę architektury modelu GPT

```
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
```

Używa atrapy klasy `TransformerBlock`

← Używa atrapy klasy `LayerNorm`

```

x = tok_embeds + pos_embeds
x = self.drop_emb(x)
x = self.trf_blocks(x)
x = self.final_norm(x)
logits = self.out_head(x)
return logits

class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()

    def forward(self, x):
        return x

```

prosta klasa atrapa, która będzie później zastąpiona rzeczywistą implementacją klasy TransformerBlock

Ten blok nic nie robi i po prostu zwraca przekazane dane wejściowe

prosta klasa-atrafa, która później będzie zastąpiona rzeczywistą implementacją klasy TransformerBlock

Parametry w tej funkcji mają jedynie naśladować interfejs LayerNorm

Klasa DummyGPTModel w tym kodzie definiuje uproszczoną wersję podobnego do GPT modelu z użyciem modułu sieci neuronowej PyTorch (nn.Module). Architektura modelu w klasie DummyGPTModel składa się z osadzeń tokenów i pozycji, dropoutu, ciągu bloków transformera (DummyTransformerBlock), końcowej normalizacji warstwowej (DummyLayerNorm) oraz liniowej warstwy wyjściowej (out_head). Konfiguracja jest przekazywana za pośrednictwem słownika Pythona, podobnego do pokazanego wcześniej słownika GPT_CONFIG_124M.

Metoda forward opisuje przepływ danych przez model: oblicza tokeny i osadzenia pozycyjne dla indeksów wejściowych, stosuje dropout, przetwarza dane przez bloki transformera, stosuje normalizację, a na koniec tworzy logity z liniową warstwą wyjściową.

Kod z listingu 4.1 już można wykorzystać praktycznie. Należy jednak pamiętać, że w przypadku bloku transformera i normalizacji warstwowej zastosowano atrapy (DummyTransformerBlock i DummyLayerNorm). Ich właściwe implementacje opracujemy później.

W kolejnym kroku przygotujemy dane wejściowe i zainicjujemy nowy model GPT, aby zilustrować sposób jego użycia. Opierając się na kodzie tokenizera (patrz rozdział 2.), przyjrzymy się teraz, jak dane wpływają do modelu GPT i jak z niego wypływają (rysunek 4.4).

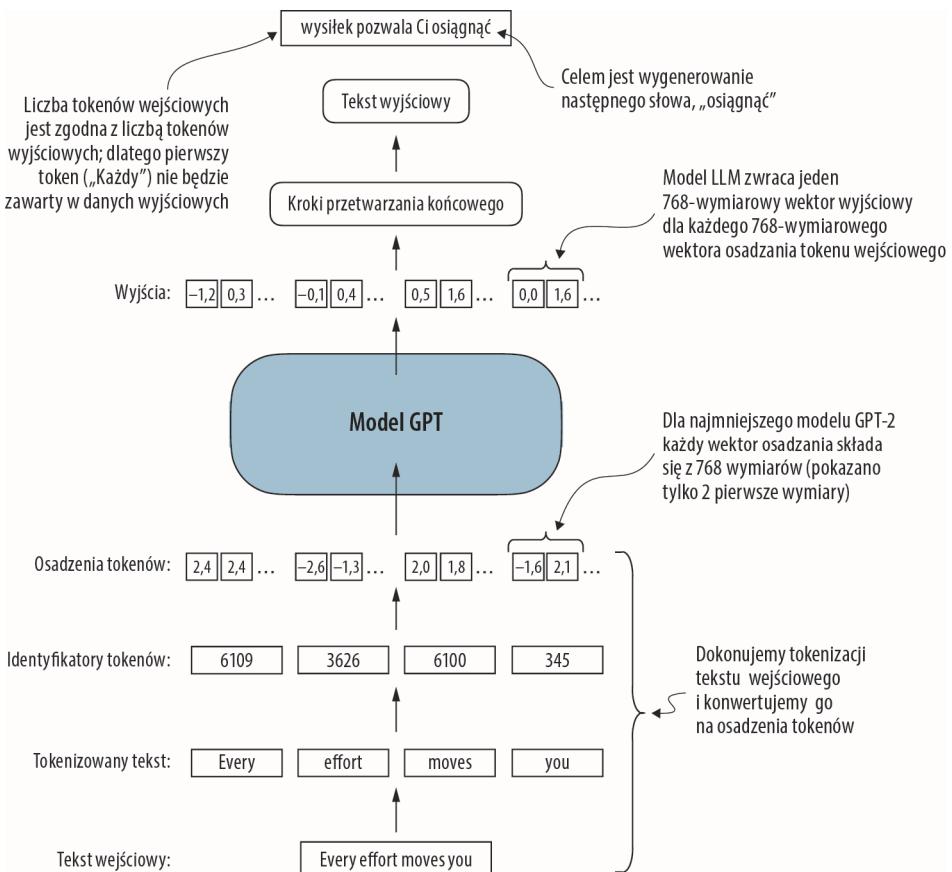
Aby zaimplementować te działania, przeprowadzimy tokenizację partii składającej się z dwóch tekstowych danych wejściowych dla modelu GPT z użyciem tokenizera tiktoken z rozdziału 2.:

```

import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []

```



Rysunek 4.4. Ogólny obraz pokazujący sposób tokenizacji, osadzania i przekazywania danych wejściowych do modelu GPT Zauważ, że w zakodowanej wcześniej klasie DummyGPTClass osadzanie tokenów jest obsługiwane wewnętrz modelu GPT. W modelach LLM wymiar osadzonego tokenu wejściowego zazwyczaj odpowiada wymiarowi wyjściowemu. Osadzenia wyjściowe reprezentują tutaj wektory kontekstu (patrz rozdział 3.)

```
txt1 = "Every effort moves you"
txt2 = "Every day holds a"
```

```
batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

Wynikowe identyfikatory tokenów dla dwóch tekstów są następujące:

```
tensor([[6109, 3626, 6100, 345],
       [6109, 1110, 6622, 257]])
```

Pierwszy wiersz odpowiada pierwszemu tekstowi, a drugi wiersz odpowiada drugiemu tekstowi

Następnie inicjujemy nowy egzemplarz klasy DummyGPTModel o 124-milionach parametrów i przekazujemy do niego poddaną tokenizacji partię batch:

```

torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Kształt wyjścia:",
logits.shape)
print(logits)

```

Dane wyjściowe modelu, które powszechnie określa się jako logity, są następujące:

```

Kształt wyjścia: torch.Size([2, 4, 50257])
tensor([[[[-0.9289,  0.2748, -0.7557,  ..., -1.6070,  0.2702, -0.5888],
          [-0.4476,  0.1726,  0.5354,  ..., -0.3932,  1.5285,  0.8557],
          [ 0.5680,  1.6053, -0.2155,  ...,  1.1624,  0.1380,  0.7425],
          [ 0.0447,  2.4787, -0.8843,  ...,  1.3219, -0.0864, -0.5856]],

         [[-1.5474, -0.0542, -1.0571,  ..., -1.8061, -0.4494, -0.6747],
          [-0.8422,  0.8243, -0.1098,  ..., -0.1434,  0.2079,  1.2046],
          [ 0.1355,  1.1858, -0.1453,  ...,  0.0869, -0.1590,  0.1552],
          [ 0.1666, -0.8138,  0.2307,  ...,  2.5035, -0.3055, -0.3083]]], grad_fn=<UnsafeViewBackward0>)

```

Tensor wyjściowy zawiera dwa wiersze odpowiadające dwóm próbkom tekstu. Każda próbka tekstu składa się z czterech tokenów; każdy token to 50 257-wymiarowy wektor, który odpowiada rozmiarowi słownictwa tokenizera.

Osadzenie ma 50 257 wymiarów, ponieważ każdy z tych wymiarów odnosi się do unikatowego tokena w słowniku. W implementacji kodu przetwarzania końcowego uwzględnimy konwersję tych złożonych z 50 257 wymiarów wektorów z powrotem na identyfikatory tokenów, które następnie będzie można zdekodować na słowa.

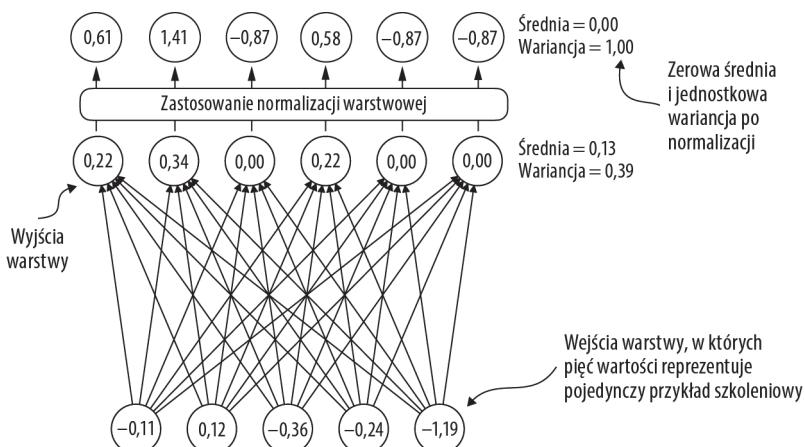
Po omówieniu architektury GPT oraz jej wejść i wyjść utwórzmy kod rzeczywistych klas, które zastąpią atrapy. Na początek opracujmy rzeczywistą implementację klasy normalizacji warstwowej, która zastąpi użytą we wcześniejszym kodzie klasę `dummy_layer_norm`.

4.2. Normalizacja warstwowa aktywacji

Szkolenie głębokich sieci neuronowych z wieloma warstwami ze względu na liczne problemy, takie jak znikające lub eksplodujące gradienty, czasami może okazać się trudne. Tego rodzaju problemy prowadzą do niestabilnej dynamiki uczenia i utrudniają sieci skuteczne dostrajanie wag. W takim przypadku w procesie uczenia sieci neuronowej trudno znaleźć zestaw parametrów (wag), przy których wartość funkcji straty jest najmniejsza. Mówiąc inaczej, sieć ma trudności z uczeniem się podstawowych wzorców w danych w stopniu, jaki pozwoliłby jej na dokładne prognozowanie lub podejmowanie decyzji.

UWAGA Jeśli dopiero zaczynasz przygodę ze szkoleniem sieci neuronowych i pojęcie gradientów nie jest Ci dobrze znane, sięgnij do krótkiego wprowadzenia do tych pojęć, zawartego w podrozdziale A.4 w „Dodataku A”. Szczegółowe zrozumienie gradientów na poziomie matematycznym nie jest jednak konieczne do śledzenia treści tej książki.

Zaimplementujmy teraz *normalizację warstwową*, której funkcja polega na poprawieniu stabilności i wydajności szkolenia sieci neuronowej. Głównym zadaniem normalizacji warstwowej jest dostosowanie aktywacji (wyjścia) warstwy sieci neuronowej tak, aby ich średnia wynosiła 0, a wariancja 1, co określa się również jako wariancję jednostkową. Taka regulacja przyspiesza zbieżność do skutecznych wag i zapewnia spójne, niezawodne szkolenie. W modelu GPT-2 i nowoczesnych architekturach transformerowych normalizację warstwową zwykle stosuje się przed modułem uwagi wielogłowicowej i po nim oraz, jak widzieliśmy w przypadku klasy atrapy `DummyLayerNorm`, przed końcową warstwą wyjściową. Wizualny przegląd funkcji normalizacji warstwowej przedstawiam na rysunku 4.5.



Rysunek 4.5. Ilustracja normalizacji warstwowej, w której sześć wyjść warstwy, zwanych również aktywacjami, jest znormalizowanych w taki sposób, że ich średnia wynosi 0, a wariancja 1

Przykład pokazany na rysunku 4.5 można odtworzyć za pomocą poniższego kodu. Jest to implementacja warstwy sieci neuronowej z pięcioma wejściami i sześcioma wyjściami, które będą zastosowane do dwóch przykładów wejściowych:

```
torch.manual_seed(123)
batch_example = torch.randn(2, 5)
layer = nn.Sequential(nn.Linear(5, 6), nn.ReLU())
out = layer(batch_example)
print(out)
```

Tworzy dwa przykłady szkoleniowe, z których każdy ma pięć wymiarów (cech)

Uruchomienie tego kodu spowoduje wyświetlenie tensora zamieszczonego poniżej. Jego pierwszy wiersz zawiera dane wyjściowe warstwy dla pierwszego wejścia, a drugi wiersz zawiera dane wyjściowe warstwy dla drugiego wiersza:

```
tensor([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000],
       [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]], grad_fn=<ReluBackward0>)
```

Zakodowana powyżej warstwa sieci neuronowej składa się z warstwy Linear, po której następuje nieliniowa funkcja aktywacji, ReLU (akronim od *rectified linear unit*), będąca standardową funkcją aktywacji w sieciach neuronowych. Dla wyjaśnienia: ReLU po prostu przekształca ujemne wartości wejściowe w zera, co sprawia, że warstwa generuje jedynie wartości dodatnie. Dzięki funkcji ReLU wynik tej warstwy nie zawiera żadnych wartości ujemnych. W dalszej części rozdziału użyję w modelu GPT innej, bardziej zaawansowanej funkcji aktywacji.

Przed zastosowaniem normalizacji warstwowej do tych wyników zbadajmy średnią i wariancję:

```
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Średnia:\n", mean)
print("Wariancja:\n", var)
```

Oto wynik działania tego kodu:

```
Średnia:
tensor([[0.1324],
       [0.2170]], grad_fn=<MeanBackward1>
)
Wariancja:
tensor([[0.0231],
       [0.0398]], grad_fn=<VarBackward0>)
```

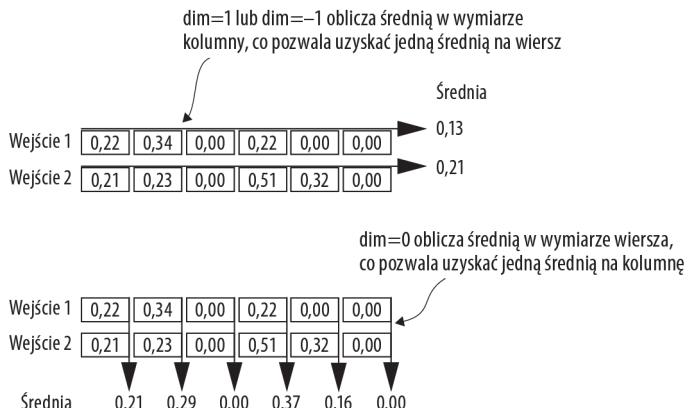
Pierwszy wiersz w tensorze średniej zawiera średnią wartość dla pierwszego wiersza wejściowego, a drugi wiersz wyjściowy zawiera średnią dla drugiego wiersza wejściowego.

Dzięki użyciu w takich operacjach jak obliczanie średniej lub wariancji instrukcji `keepdim=True` tensor wyjściowy zachowuje tę samą liczbę wymiarów, co tensor wejściowy, nawet jeśli operacja zmniejsza tensor wzduż wymiaru określonego przez `dim`. Na przykład bez instrukcji `keepdim=True` wynikowy tensor średniej byłby dwuwymiarowym wektorem `[0.1324, 0.2170]` zamiast macierzą 2×1 `[[0.1324], [0.2170]]`.

Parametr `dim` określa wymiar, wzduż którego powinno zostać wykonane obliczenie statystyki (tutaj średniej lub wariancji) w tensorze. Jak wyjaśniam na rysunku 4.6, dla dwuwymiarowego tensora (czyli macierzy) użycie `dim=-1` do takich operacji jak obliczanie średniej lub wariancji daje taki sam efekt jak użycie `dim=1`. To dlatego, że `-1` odnosi się do ostatniego wymiaru tensora, odpowiadającego kolumnom w dwuwymiarowym tensorze.

Po dodaniu do modelu GPT normalizacji warstwowej, która spowoduje utworzenie trójwymiarowych tensorów o kształcie `[batch_size, num_tokens, embedding_size]`, nadal można używać do normalizacji w ostatnim wymiarze ustawienia `dim=-1`, co pozwala uniknąć zmiany z `dim=1` na `dim=2`.

W kolejnym kroku zastosujmy normalizację warstwową do uzyskanych wcześniej wyników. Operacja polega na odjęciu średniej i podzieleniu przez pierwiastek kwadratowy wariancji (tak obliczoną wielkość określa się także jako odchylenie standardowe):



Rysunek 4.6. Ilustracja parametru dim podczas obliczania średniej tensora. Na przykład dla tensora dwuwymiarowego (macierzy) o wymiarach [wiersze, kolumny] użycie dim=0 spowoduje wykonanie działania w wierszach (w pionie — tak jak pokazałem na dole rysunku). To skutkuje wynikiem, który agreguje dane dla każdej kolumny. Użycie dim=1 lub dim=-1 będzie skutkować wykonaniem działania na kolumnach (w poziomie, tak jak pokazałem na górze rysunku), co spowoduje zagregowanie danych wyjściowych dla każdego wiersza

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Znormalizowane wyniki warstwy:\n", out_norm)
print("Średnia:\n", mean)
print("Wariancja:\n", var)
```

Jak można zauważyć na podstawie wyników, znormalizowane wyjście warstwy, teraz zawierające również wartości ujemne, mają średnią o i wariancję 1:

```
Znormalizowane wyniki warstwy:
tensor([[ 0.6159,  1.4126, -0.8719,  0.5872, -0.8719, -0.8719],
       [-0.0189,  0.1121, -1.0876,  1.5173,  0.5647, -1.0876]],
      grad_fn=<DivBackward0>)
Średnia:
tensor([[-9.9341e-09],
       [1.9868e-08]], grad_fn=<MeanBackward1>)
Wariancja:
tensor([[1.0000],
       [1.0000]], grad_fn=<VarBackward0>)
```

Zwrć uwagę, że wartość -9.9341×10^{-9} w tensorze wyjściowym jest zapisem naukowym dla -9.9341×10^{-9} , co w postaci dziesiętnej odpowiada wartości -0.000000099341 . Ta wartość jest bardzo bliska zera, ale ze względu na niewielkie błędy numeryczne, które z powodu skończonej dokładności reprezentowania liczb w komputerach mogą się kumulować, nie wynosi dokładnie 0.

Aby poprawić czytelność, można również wyłączyć wyświetlanie wartości tensora w notacji naukowej. By to zrobić, należy ustawić parametr `sci_mode` na `False`:

```
torch.set_printoptions(sci_mode=False)
print("Średnia:\n", mean)
print("Wariancja:\n", var)
```

Oto wynik działania tego kodu:

```
Średnia:
tensor([[ -0.0000],
       [ -0.0000]], grad_fn=<MeanBackward1>
Wariancja:
tensor([[1.0000],
       [1.0000]], grad_fn=<VarBackward0>)
```

Do tej pory kodowaliśmy normalizację warstwową i stosowaliśmy ją krok po kroku. Spróbujmy teraz zamknąć ten proces w module PyTorch, który będzie można później wykorzystać w modelu GPT (listing 4.2).

Listing 4.2. Klasa normalizacji warstwowej

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

Ta konkretna implementacja normalizacji warstwowej działa na ostatnim wymiarze wejściowego tensora x , który reprezentuje wymiar osadzenia (`emb_dim`). Zmienna `eps` (*epsilon*) to dodawana do wariancji stała o niewielkiej wartości, której celem jest zapobieżenie dzieleniu przez zero podczas normalizacji. Parametry `scale` i `shift` to dwa trenowalne parametry (o tym samym wymiarze, co wejście), które model LLM automatycznie dostosowuje, jeśli ze szkolenia wynika, że ich zmiana poprawia wydajność modelu. W ten sposób model uczy się odpowiedniego skalowania i przesuwania, które najlepiej pasują do przetwarzanych danych.

Wypróbujmy teraz moduł `LayerNorm` w praktyce i zastosujmy go do danych przekazywanych partiami:

```
ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
mean = out_ln.mean(dim=-1, keepdim=True)
var = out_ln.var(dim=-1, unbiased=False, keepdim=True)
print("Średnia:\n", mean)
print("Wariancja:\n", var)
```

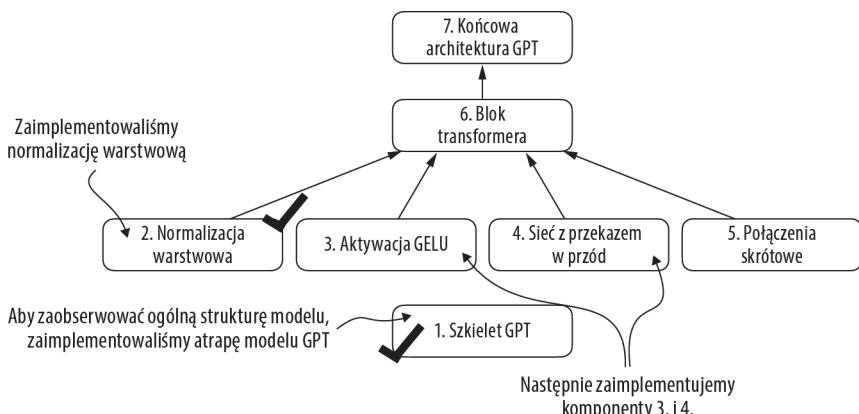
Stronnicza wariancja

W metodzie obliczania wariancji używa się szczegółu implementacji — ustawienia `unbiased=False`. W obliczeniach wariancji oznacza to podzielenie wartości przez liczbę wejść n . To podejście nie uwzględnia korekty Bessela, która w celu skorygowania stronniczości w oszacowaniach wariancji próby zazwyczaj polega na użyciu w mianowniku $n-1$ zamiast n . Ze względu na tę decyzję uzyskujemy tak zwane stronnicze oszacowanie wariancji. W przypadku modeli LLM, w których wymiar osadzenia n jest duży, różnica między użyciem n i $n-1$ jest praktycznie niewielka. Wybrałem to podejście, aby zapewnić kompatybilność z warstwami normalizacji modelu GPT-2 i ponieważ odzwierciedla ono domyślne zachowanie zaimplementowane w bibliotece TensorFlow, zastosowane do implementacji oryginalnego modelu GPT-2. Użycie podobnego ustawienia zapewnia zgodność zastosowanej metody ze wstępnie przeszkolonymi wagami, które załadujemy w rozdziale 6.

Wyniki pokazują, że kod normalizacji warstwy działa zgodnie z oczekiwaniami i normalizuje wartości każdego z dwóch wejść tak, aby miały średnią 0 i wariancję 1:

```
Średnia:  
tensor([[ -0.0000],  
       [ 0.0000]], grad_fn=<MeanBackward1>)  
Wariancja:  
tensor([[1.0000],  
       [1.0000]], grad_fn=<VarBackward0>)
```

Omówiliem dwa bloki konstrukcyjne, które trzeba zaimplementować w architekturze GPT (rysunek 4.7). W kolejnym kroku przyjrzymy się zastosowaniu funkcji aktywacji GELU, która jest jedną z funkcji aktywacji używanych w LLM, zamiast tradycyjnej funkcji ReLU, której użyliśmy wcześniej.



Rysunek 4.7. Elementy składowe niezbędne do zbudowania architektury GPT. Do tej pory zakończyliśmy implementację normalizacji warstwowej i szkielet modelu GPT. W dalszej części skupimy się na aktywacji GELU i sieci ze sprzężeniem w przód

Normalizacja warstwowa a normalizacja partiami

Osoby znające pojęcie normalizacji partiami (ang. *batch normalization*), powszechniej i tradycyjnej metody normalizacji stosowanej w sieciach neuronowych, mogą się zastanawiać, jak wypada ona w porównaniu z normalizacją warstwową. W przeciwieństwie do normalizacji partiami, która normalizuje wymiar partii, normalizacja warstwowa normalizuje wymiar cech. Modele LLM często wymagają znacznych zasobów obliczeniowych, a dostępny sprzęt lub konkretny przypadek użycia może dyktować wielkość partii potrzebną podczas szkolenia lub wnioskowania. Ponieważ normalizacja warstwowa normalizuje każde wejście niezależnie od wielkości partii, zapewnia w tych scenariuszach większą elastyczność i stabilność. Jest to szczególnie korzystne w przypadku szkoleń rozproszonych lub podczas wdrażania modeli w środowiskach, w których zasoby są ograniczone.

4.3. Implementacja sieci ze sprzężeniem w przód z aktywacjami GELU

W kolejnym kroku zaimplementujemy prosty podmoduł sieci neuronowej wykorzystywany w modelach LLM jako część bloku transformera. Pracę rozpoczynamy od zaimplementowania funkcji aktywacji GELU, która w tym podmodułie sieci neuronowej odgrywa kluczową rolę.

UWAGA Dodatkowe informacje na temat implementacji sieci neuronowych w bibliotece PyTorch można znaleźć w podrozdziale A.5 w „*Dodatku A*”.

W przeszłości funkcję aktywacji ReLU powszechnie stosowano w uczeniu głębokim, głównie ze względu na jej prostotę i skuteczność w różnych architekturach sieci neuronowych. Jednak w modelach LLM poza tradycyjną funkcją ReLU stosuje się kilka innych funkcji aktywacji. Dwa godne uwagi przykłady to GELU (ang. *Gaussian error linear unit*) i SwiGLU (ang. *Swish-gated linear unit*).

GELU i SwiGLU są bardziej złożonymi i gładkimi funkcjami aktywacji, zawierającymi, odpowiednio, jednostki liniowe Gaussa i jednostki sigmoidalne. W przeciwieństwie do prostszej funkcji ReLU zapewniają one lepszą wydajność w przypadku modeli uczenia głębokiego.

Funkcję aktywacji GELU można zaimplementować na kilka sposobów. Wersja dokładna jest zdefiniowana jako $\text{GELU}(x) = x \cdot \Phi(x)$, gdzie $\Phi(x)$ jest funkcją skumulowanego rozkładu standardowego rozkładu Gaussa. Jednak w praktyce powszechnie implementuje się tańsze obliczeniowo przybliżenie (oryginalny model GPT-2 również przeszkolono z przybliżeniem, które zostało wyznaczone poprzez dopasowanie krzywej):

$$\text{GELU}(x) \approx 0,5 \cdot x \cdot \left(1 + \tanh \left[\sqrt{\frac{2}{\pi}} \cdot (x + 0,044715 \cdot x^3) \right] \right)$$

W kodzie można zaimplementować tę funkcję jako moduł PyTorch (listing 4.3).

Listing 4.3. Implementacja funkcji aktywacji GELU

```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

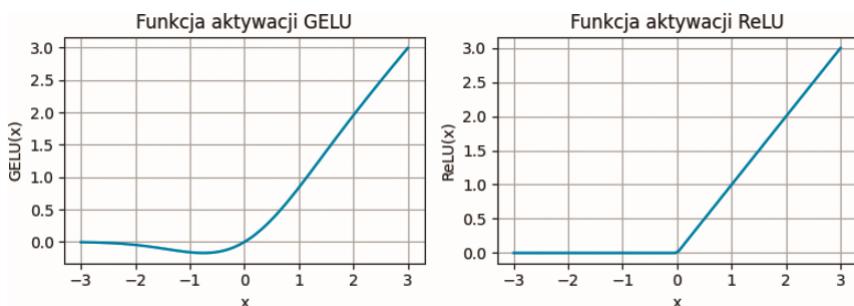
    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3)))
        ))
```

Następnie, aby zorientować się, jak wygląda funkcja GELU i jak wypada w porównaniu z funkcją ReLU, spróbujmy wykreślić te funkcje obok siebie:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100) ← Tworzy 100 przykładowych punktów
y_gelu, y_relu = gelu(x), relu(x) danych w zakresie od -3 do 3
plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"])):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"Funkcja aktywacji {label}")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

Jak widać na wynikowym wykresie na rysunku 4.8, ReLU (po prawej) jest funkcją liniową, która bezpośrednio zwraca dane wejściowe, jeśli są dodatnie; w przeciwnym razie zwraca zero. GELU (po lewej) to gładka, nieliniowa funkcja, która przybliża funkcję ReLU, ale z niezerowym gradientem dla prawie wszystkich wartości ujemnych (z wyjątkiem wartości $x \approx -0,75$).



Rysunek 4.8. Wykresy funkcji GELU i ReLU wykonane z użyciem biblioteki matplotlib. Oś x przedstawia dane wejściowe funkcji, a oś y dane wyjściowe funkcji

Gladkość funkcji GELU może prowadzić do lepszych właściwości optymalizacyjnych podczas szkolenia, ponieważ pozwala na bardziej zniuansowane dostrojenie parametrów modelu. Dla odróżnienia od GELU, funkcja ReLU ma ostry narożnik przy zerze (rysunek 4.18, po prawej), co czasami, szczególnie w sieciach, które są bardzo głębokie lub mają złożoną architekturę, może utrudniać optymalizację. Co więcej, w przeciwnieństwie do ReLU, która zwraca zero dla każdego ujemnego wejścia, GELU dla wartości ujemnych może zwracać niewielkie, niezerowe wyjście. Ze względu na tę cechę neurony, które podczas procesu uczenia otrzymują ujemne wejścia, mogą nadal, choć w mniejszym stopniu niż wejścia dodatnie, przyczyniać się do procesu uczenia.

W kolejnym kroku zastosujemy funkcję GELU w celu zaimplementowania prostego modułu sieci neuronowej FeedForward, który później wykorzystamy w bloku transformera modelu LLM (listing 4.4).

Listing 4.4. Moduł sieci neuronowej typu feed forward

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```

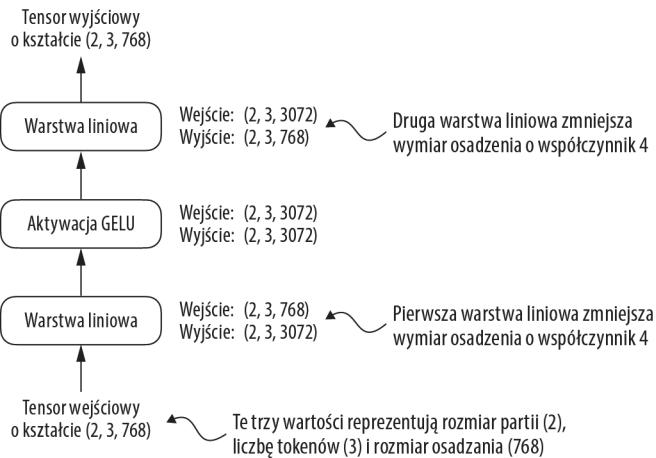
Jak można zauważyć, moduł FeedForward to mała sieć neuronowa składająca się z dwóch warstw Linear i funkcji aktywacji GELU. W 124-milionowym modelu GPT, zgodnie z konfiguracją określoną w słowniku GPT_CONFIG_124M, otrzymuje on partie wejściowe z tokenami o rozmiarze osadzeń 768 każdy (GPT_CONFIG_124M['emb_dim'] = 768). Sposób przetwarzania rozmiaru osadzania wewnętrz tej małej sieci neuronowej ze sprzężeniem w przód pokazałem na rysunku 4.9.

Podążając za przykładem przedstawionym na rysunku 4.9 zainicjujmy nowy moduł FeedForward z rozmiarem osadzania tokenów wynoszącym 768 i przekażmy do niego partię danych wejściowych z dwiema próbками i trzema tokenami każda:

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768)           ←
out = ffn(x)                      |
print(out.shape)                  | Tworzy próbkę danych wejściowych
                                    | z wymiarem partii 2
```

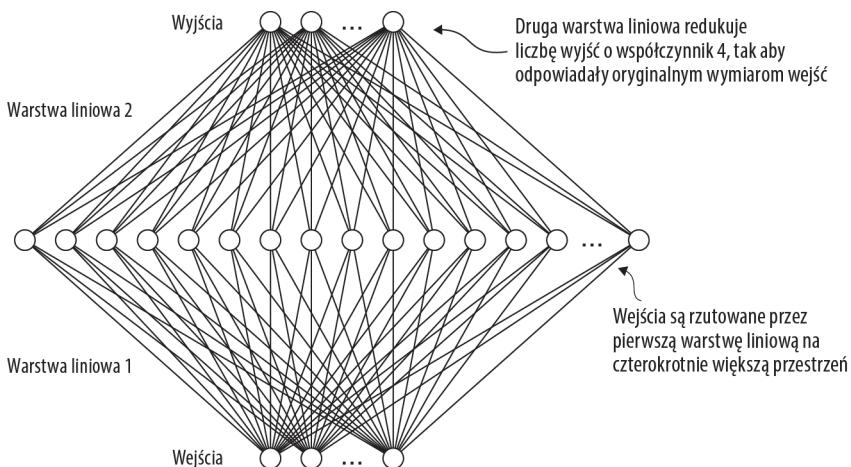
Jak można zauważyć, kształt tensora wyjściowego jest taki sam jak tensora wejściowego:
`torch.Size([2, 3, 768])`

Moduł FeedForward odgrywa kluczową rolę w zwiększaniu zdolności modelu do uczenia się na podstawie danych i ich uogólniania. Chociaż wymiary wejściowe i wyjściowe tego modułu są takie same, wewnętrznie moduł rozszerza wymiar osadzenia do



Rysunek 4.9. Przegląd połączeń między warstwami sieci neuronowej typu feed forward. Tę sieć neuronową można dostosować do różnych wielkości partii i liczby tokenów na wejściu. Rozmiar osadzeń dla każdego tokenu jest jednak określany i ustalany podczas inicjalizacji wag

przestrzeni o wyższym wymiarze poprzez pierwszą warstwę liniową (rysunek 4.10). Za tym rozszerzeniem występuje nieliniowa aktywacja GELU, a następnie ponowna redukcja, za pomocą drugiego transformera liniowego, do oryginalnego wymiaru. Taki projekt pozwala na eksplorację bogatszej przestrzeni reprezentacji.

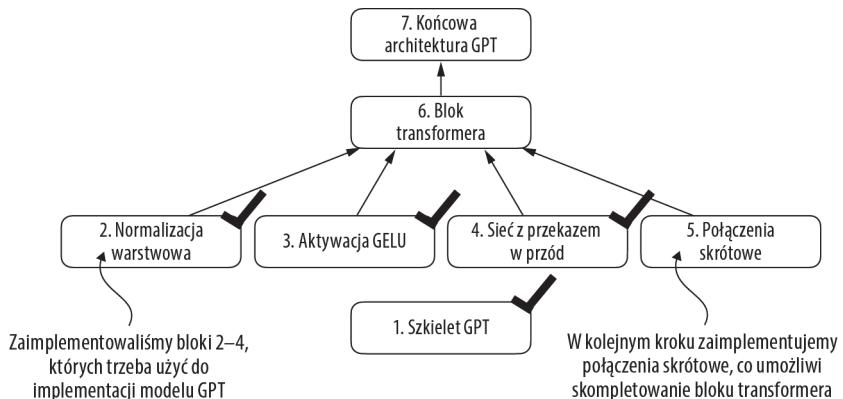


Rysunek 4.10. Ilustracja rozszerzania i redukcji wyjść warstw w sieci neuronowej typu feed forward. Wejścia najpierw zwiększają swoją liczbę 4-krotnie — z 768 do 3072 wartości. Następnie druga warstwa redukuje 3072 wartości z powrotem do 768-wymiarowej reprezentacji

Co więcej, jednolitość wymiarów wejściowych i wyjściowych upraszcza architekturę, co pozwala na układanie stosu złożonego z wielu warstw. Jak się przekonasz, dzięki

takiemu układowi nie ma konieczności dostosowywania wymiarów między warstwami, dzięki czemu model jest bardziej skalowalny.

Jak widać na rysunku 4.11, zaimplementowaliśmy już większość elementów składowych modelu LLM. W kolejnym kroku omówię pojęcie połączeń skrótowych, które wstawia się pomiędzy różnymi warstwami sieci neuronowej, a które mają istotne znaczenie dla poprawy wydajności uczenia w architekturach głębokich sieci neuronowych.



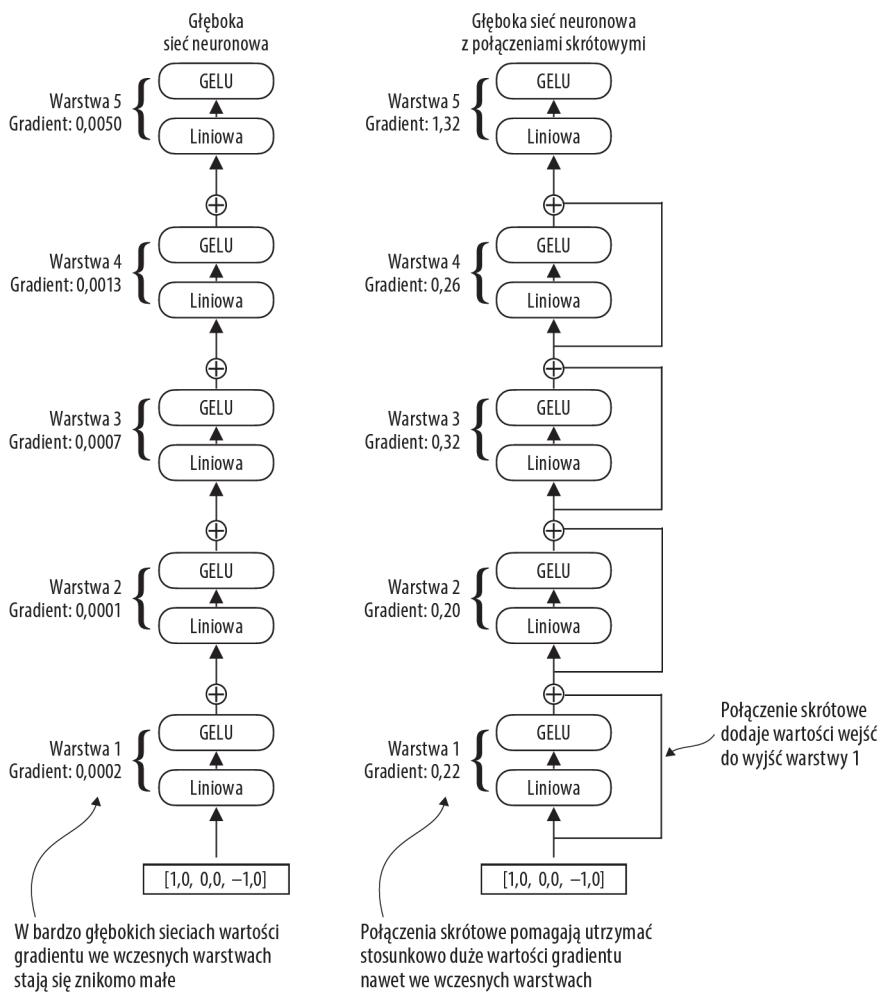
Rysunek 4.11. Elementy składowe niezbędne do zbudowania architektury GPT. Czarne znaczniki oznaczają te, które już zostały zaimplementowane

4.4. Dodawanie połączeń skrótowych

W tym podrozdziale opiszę pojęcie *połączeń skrótowych* (ang. *shortcut connections*), znanych również jako połączenia pomijane (ang. *skip connections*) lub połączenia szczegółowe (ang. *residual connections*). Połączenia skrótowe pierwotnie zaproponowano dla sieci głębokich, w zadaniach komputerowego widzenia (w szczególności w sieciach rezydualnych), w celu złagodzenia problemów związanych z zanikającymi gradientami. Problem zanikających gradientów odnosi się do kwestii, w której gradienty (wpływające na aktualizacje wag podczas szkolenia), gdy propagują się wstecz przez warstwy, stają się coraz mniejsze, co utrudnia skuteczne szkolenie wcześniejszych warstw.

Jak widać na rysunku 4.12, dzięki pominięciu warstwy lub większej liczby warstw, co osiąga się przez połączenie wyjścia jednej warstwy z wyjściem kolejnej, połączenie skrótowe tworzy alternatywną, krótszą ścieżkę dla przepływającego przez sieć gradientu. Z tego powodu połączenia skrótowe czasami określa się również jako połączenia pomijane. Odgrywają one kluczową rolę w zachowaniu przepływu gradientów w trakcie przebiegu wstecznego podczas szkolenia.

Aby zobaczyć, jak można dodać połączenia skrótowe w metodzie `forward`, na liscingu 4.5 zaimplementujemy sieć neuronową z rysunku 4.12.



Rysunek 4.12. Porównanie głębokiej sieci neuronowej składającej się z pięciu warstw bez połączeń skrótnowych (po lewej) i z nimi (po prawej). Połączenia skrótnowe obejmują dołączanie wejść warstwy do jej wyjścia, co skutecznie tworzy alternatywną ścieżkę, która pomija niektóre warstwy. Gradienty w każdej warstwie oznaczają średnią, bezwzględną wartość, której sposób obliczenia jest pokazany na listingu 4.5

Listing 4.5. Sieć neuronowa do zilustrowania połączeń skrótnowych

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]),
```

Implementacja
pięciu warstw

```

        GELU()),
nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]),
              GELU()),
nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]),
              GELU()),
nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]),
              GELU())
    ])

def forward(self, x):
    for layer in self.layers:
        layer_output = layer(x)
        if self.use_shortcut and x.shape == layer_output.shape: ←
            x = x + layer_output
        else:
            x = layer_output
    return x

```

The diagram shows the forward pass of a neural network. It highlights the use of GELU activation functions in each layer, the addition of skip connections (shortcuts) to the input, and the final output.

Kod implementuje głęboką sieć neuronową z pięcioma warstwami, z których każda składa się z warstwy Linear i funkcji aktywacji GELU. W przejściu do przodu iteracyjnie przekazujemy przez warstwy dane wejściowe i opcjonalnie — jeśli atrybut self `→.use_shortcut` jest ustawiony na True — dodajemy połączenia skrótu.

Użyjmy tego kodu do zainicjowania sieci neuronowej bez połączeń skrótoowych. Każda warstwa zostanie zainicjowana w taki sposób, aby akceptowała przykład z trzema wartościami wejściowymi i zwracała trzy wartości wyjściowe. Ostatnia warstwa zwraca pojedynczą wartość wyjściową:

```

layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]]) ←
torch.manual_seed(123) ←
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)

```

The diagram shows the initialization of a neural network. It highlights the definition of layer sizes, sample input, and manual seed, with annotations explaining the purpose of each step.

Następnie zaimplementujemy funkcję, która oblicza gradienty w przebiegu modelu wstecz:

```

def print_gradients(model, x):
    output = model(x) ← Przebieg w przód
    target = torch.tensor([[0.]]) ←

    loss = nn.MSELoss() ← Oblicza stratę na podstawie tego,
    loss = loss(output, target) ← jak blisko siebie znajdują się cel i bieżący wynik

    loss.backward() ←
    for name, param in model.named_parameters():
        if 'weight' in name:
            print(f"Średnia gradientu dla {name} wynosi
→{param.grad.abs().mean().item()}")

```

The diagram shows the backward pass for gradient calculation. It highlights the forward pass, loss calculation, backward pass, and printing of average gradients for weight parameters.

Ten kod określa funkcję straty, która oblicza, jak blisko siebie jest wynik modelu i określony przez użytkownika cel (tutaj, dla uproszczenia, wartość 0). Następnie wywoływana jest metoda `loss.backward()`, która w bibliotece PyTorch oblicza gradient strat dla każdej warstwy w modelu. Po parametrach wag można iterować za pomocą metody

`model.named_parameters()`. Założymy, że dla danej warstwy mamy macierz parametrów wagi 3×3 . W takim przypadku ta warstwa będzie miała gradienty o wymiarach 3×3 . Obliczamy ich średnią wartość bezwzględną, aby uzyskać pojedynczą wartość gradientu na warstwę, co ułatwia porównanie gradientów między warstwami.

Krótko mówiąc, `.backward()` jest dostępną w PyTorch wygodną metodą obliczającą gradienty strat wymagane podczas uczenia modelu. Dzięki niej nie trzeba implementować działań matematycznych w celu obliczania gradientu, przez co praca z głębokimi sieciami neuronowymi staje się znacznie łatwiejsza.

UWAGA Czytelników, którzy nie znają dobrze pojęcia gradientów i zagadnień związanych z uczeniem sieci neuronowych, zachęcam do przeczytania podrozdziałów A.4 i A.7 w „Dodatku A”.

Zastosujmy teraz funkcję `print_gradients` do modelu bez połączeń skrótowych:

```
print_gradients(model_without_shortcut, sample_input)
```

Uzyskamy następujący wynik:

```
Średnia gradientu dla layers.0.0.weight wynosi 0.00020173584925942123  
Średnia gradientu dla layers.1.0.weight wynosi 0.00012011159560643137  
Średnia gradientu dla layers.2.0.weight wynosi 0.0007152040489017963  
Średnia gradientu dla layers.3.0.weight wynosi 0.0013988736318424344  
Średnia gradientu dla layers.4.0.weight wynosi 0.005049645435065031
```

Wyniki działania funkcji `print_gradients` pokazują, że w miarę przechodzenia od ostatniej warstwy (`layers.4`) do pierwszej warstwy (`layers.0`) gradienty stają się mniejsze. To zjawisko określa się jako *problem zanikającego gradientu*.

Utwórzmy teraz egzemplarz modelu z połączeniami skrótowymi i zobaczymy, jak on wygląda w porównaniu z poprzednim:

```
torch.manual_seed(123)  
model_with_shortcut = ExampleDeepNeuralNetwork(  
    layer_sizes, use_shortcut=True  
)  
print_gradients(model_with_shortcut, sample_input)
```

Oto uzyskane wyniki:

```
Średnia gradientu dla layers.0.0.weight wynosi 0.22169791162014008  
Średnia gradientu dla layers.1.0.weight wynosi 0.20694105327129364  
Średnia gradientu dla layers.2.0.weight wynosi 0.32896995544433594  
Średnia gradientu dla layers.3.0.weight wynosi 0.2665732204914093  
Średnia gradientu dla layers.4.0.weight wynosi 1.3258540630340576
```

Ostatnia warstwa (`layers.4`) nadal ma większy gradient niż pozostałe. Wartość gradientu stabilizuje się jednak w miarę postępów w kierunku pierwszej warstwy (`layers.0`) i nie zmniejsza się do wartości znikomo małej.

Podsumowując, połączenia skrótowe są ważne w pokonywaniu ograniczeń związanych z problemem zanikającego gradientu w głębokich sieciach neuronowych. Są one podstawowym elementem składowym bardzo dużych modeli, takich jak LLM.

Dzięki temu, że podczas szkolenia modelu GPT zapewniają spójny przepływ gradientu, umożliwiają bardziej skuteczne szkolenie. Właściwość tę wykorzystamy w następnym rozdziale.

Na koniec połączmy wszystkie wcześniej omówione pojęcia (normalizacja warstwowa, aktywacja GELU, moduł sprzężenia zwrotnego i połączenia skrótowe) w bloku transformera — ostatnim bloku konstrukcyjnym, który trzeba zakodować w architekturze GPT.

4.5. Łączenie warstw uwagi i warstw liniowych w bloku transformera

Teraz zaimplementujmy *blok transformera*, podstawowy element modelu GPT i innych architektur LLM. Ten blok, w 124-milionowej architekturze GPT-2 powtarzany kilka-naście razy, łączy w sobie kilka omówionych wcześniej pojęć: wielogłowicową uwagę, normalizację warstwową, dropout, warstwy z przekazem w przód i aktywacje GELU. W dalszej części rozdziału połączymy ten blok transformera z pozostałymi częściami architektury GPT.

Blok transformera, który łączy w sobie kilka komponentów, przedstawiłem na rysunku 4.13. Wspomniany blok obejmuje maskowany moduł wielogłowicowej uwagi (patrz rozdział 3.) oraz zaimplementowany wcześniej moduł FeedForward (patrz podrozdział 4.3). Podczas przetwarzania sekwencji wejściowej bloku transformera każdemu elementowi sekwencji (na przykład słowi lub tokenowi reprezentującemu pod-słowo) odpowiada wektor o stałym rozmiarze (w tym przypadku 768 wymiarów). Operacje w bloku transformera, w tym w wielogłowicowej warstwie uwagi i warstwach ze sprzężeniem w przód, mają na celu przekształcenie tych wektorów w taki sposób, aby była zachowana wymiarowość.

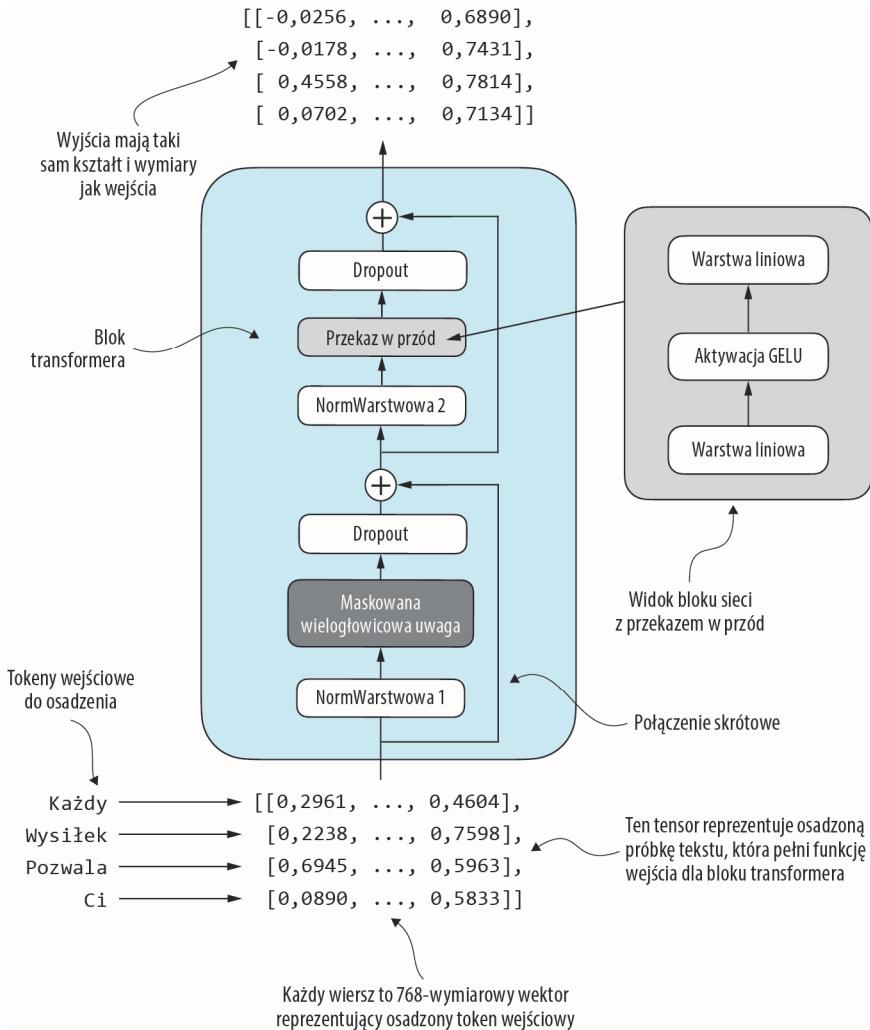
Chodzi o to, że mechanizm samouwagi w wielogłowicowym bloku uwagi identyfikuje i analizuje związki między elementami w sekwencji wejściowej. W przeciwieństwie do tego, sieć z przekazem w przód modyfikuje dane w każdej pozycji indywidualnie. To połączenie nie tylko umożliwia bardziej znuansowane zrozumienie i przetwarzanie danych wejściowych, ale także zwiększa ogólną zdolność modelu do obsługi złożonych wzorców danych.

Obiekt `TransformerBlock` można utworzyć w kodzie (listing 4.6).

Listing 4.6. Składnik bloku transformera modelu GPT

```
from chapter03 import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
```



Rysunek 4.13. Ilustracja bloku transformera. Tokeny wejściowe zostały osadzone w 768-wymiarowych wektorach. Każdy wiersz odpowiada wektorowej reprezentacji jednego tokena. Wyjściami bloku transformera są wektory o tym samym wymiarze, co wejście, które można następnie wprowadzić do kolejnych warstw modelu LLM

```

d_in=cfg["emb_dim"],
d_out=cfg["emb_dim"],
context_length=cfg["context_length"],
num_heads=cfg["n_heads"],
dropout=cfg["drop_rate"],
qkv_bias=cfg["qkv_bias"])
self.ff = FeedForward(cfg)
self.norm1 = LayerNorm(cfg["emb_dim"])
self.norm2 = LayerNorm(cfg["emb_dim"])
self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

```

```

def forward(self, x):
    shortcut = x                                ← Połączenie skrótowe dla bloku uwagi
    x = self.norm1(x)
    x = self.att(x)
    x = self.drop_shortcut(x)
    x = x + shortcut                            ← Dodanie oryginalnego wejścia

    shortcut = x      ← Połączenie skrótowe dla bloku sprzężenia zwrotnego
    x = self.norm2(x)
    x = self.ff(x)
    x = self.drop_shortcut(x)
    x = x + shortcut                            ← Dodanie oryginalnego wejścia
    return x

```

W tym kodzie zdefiniowałem opartą na bibliotece PyTorch klasę `TransformerBlock`, która zawiera wielogłowicowy mechanizm uwagi (`MultiHeadAttention`) i sieć z przekazem w przód (`FeedForward`). Obie zostały skonfigurowane na podstawie dostarczonego słownika konfiguracji (`cfg`), podobnego do pokazanego wcześniej słownika `GPT_CONFIG_124M`.

Przed każdym z tych dwóch składników zastosowano normalizację warstwową (`LayerNorm`). Dalej uwzględniono dropout, którego zadaniem jest regularyzacja modelu i zapobieżenie nadmiernemu dopasowaniu. Tę warstwę określa się również jako *Pre-LayerNorm*. W starszych architekturach, takich jak model *Original Transformer*, za warstwą samouwagi i przekazu w przód stosowano normalizację warstwową, znaną jako *Post-LayerNorm*, co często prowadziło do gorszej dynamiki szkolenia.

Klasa implementuje również sieć z przekazem w przód, w której po każdym komponencie występuje połączenie skrótowe wejścia bloku z jego wyjściem. Dzięki tej klużowej właściwości przez sieć podczas szkolenia mogą przepływać gradienty, co poprawia uczenie głębokich modeli (patrz podrozdział 4.4).

Korzystając ze zdefiniowanego wcześniej słownika `GPT_CONFIG_124M`, spróbujmy utworzyć egzemplarz bloku transformera i przekazać do niego przykładowe dane:

```

torch.manual_seed(123)
x = torch.rand(2, 4, 768)                         ← Utworzenie próbki wejściowej
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Kształt wejścia:", x.shape)
print("Kształt wyjścia:", output.shape)

```

Oto uzyskane wyniki:

```

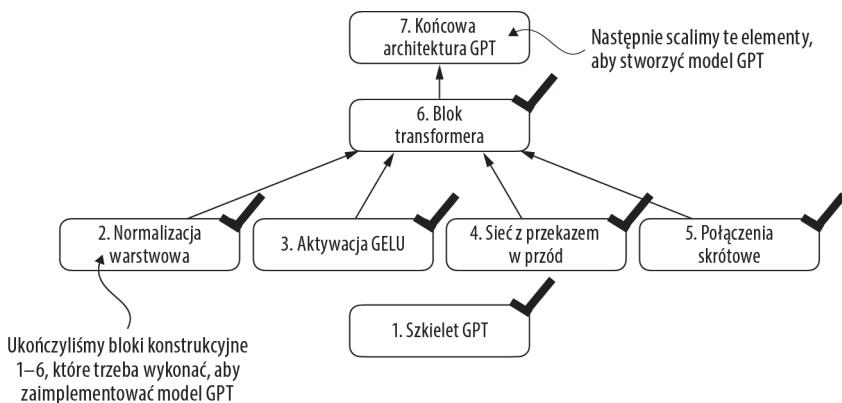
Kształt wejścia: torch.Size([2, 4, 768])
Kształt wyjścia: torch.Size([2, 4, 768])

```

Jak można zauważyć, blok transformera zachowuje na wyjściu wymiary wejścia, co pokazuje, że architektura transformera przetwarza sekwencje danych bez zmiany ich kształtu w całej sieci.

Zachowanie kształtu w całej architekturze bloku transformera nie jest przypadkowe. Stanowi kluczowy aspekt jego konstrukcji. Taka konstrukcja umożliwia skuteczne stosowanie modelu w szerokim zakresie zadań sekwencyjnych, gdzie każdy wektor wyjściowy bezpośrednio odpowiada wektorowi wejściowemu, co zachowuje relację jeden do jednego. Wyjście tworzy jednak wektor kontekstu, zawierający informacje z całej sekwencji wejściowej (patrz rozdział 3.). Oznacza to, że o ile fizyczne wymiary sekwencji (długość i rozmiar elementu) pozostają niezmienione, o tyle gdy przechodzi ona przez blok transformera, zawartość każdego wektora wyjściowego jest ponownie kodowana w celu zintegrowania informacji kontekstowych z całej sekwencji wejściowej.

Po zaimplementowaniu bloku transformera mamy do dyspozycji wszystkie bloki konstrukcyjne, które trzeba zaimplementować w architekturze GPT. Jak pokazałem na rysunku 4.14, blok transformera łączy normalizację warstwową, sieć z przekazem w przód, aktywację GELU i połączenia skrótowe. Jak się ostatecznie przekonamy, ten blok transformera będzie stanowił główny element architektury GPT.



Rysunek 4.14. Elementy składowe niezbędne do zbudowania architektury GPT. Czarne znaczniki wyboru oznaczają bloki, które zostały wykonane

4.6. Kodowanie modelu GPT

Rozdział rozpocząłem od ogólnego przeglądu architektury GPT, której nadalem nazwę DummyGPTModel. W implementacji kodu modelu DummyGPTModel pokazałem wejścia i wyjścia modelu GPT, ale jego bloki konstrukcyjne pozostały czarną skrzynką — użyłem atrap DummyTransformerBlock i DummyLayerNorm.

W tym podrozdziale zastąpimy zakodowane wcześniej atrapy DummyTransformerBlock i DummyLayerNorm rzeczywistymi implementacjami TransformerBlock i LayerNorm. W ten sposób uzyskamy w pełni działającą wersję oryginalnej wersji modelu

GPT-2 o 124 milionach parametrów. W rozdziale 5. poddamy model GPT-2 wstępemu szkoleniu, a w rozdziale 6. załadujemy wstępnie przeszkolone wagi z modelu OpenAI.

Przed przystąpieniem do implementacji modelu GPT-2 w kodzie spójrzmy na jego ogólną strukturę (rysunek 4.15), obejmującą wszystkie bloki, które zrealizowaliśmy dotychczas. Jak można zauważyc, w całej architekturze modelu GPT blok transformera jest wielokrotnie powtarzany. W przypadku 124-milionowego modelu GPT-2 ten blok jest powtarzany 12 razy, co w słowniku GPT_CONFIG_124M określono za pomocą pozycji n_layers. Blok transformera w największym modelu GPT-2, z 1542 milionami parametrów, powtarza się 48 razy.

Wyjścia z końcowego bloku transformera przechodzą następnie przez ostatni etap normalizacji warstwowej, po czym docierają do warstwy liniowego wyjścia. Ta warstwa mapuje wyjścia transformera na przestrzeń o wysokiej liczbie wymiarów (w tym przypadku 50 257, co odpowiada rozmiarowi słownictwa modelu), co pozwala przewidzieć następny token w sekwencji. Przystąpmy teraz do zakodowania architektury przedstawionej na rysunku 4.15 (listing 4.7).

Listing 4.7. Implementacja architektury modelu GPT

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

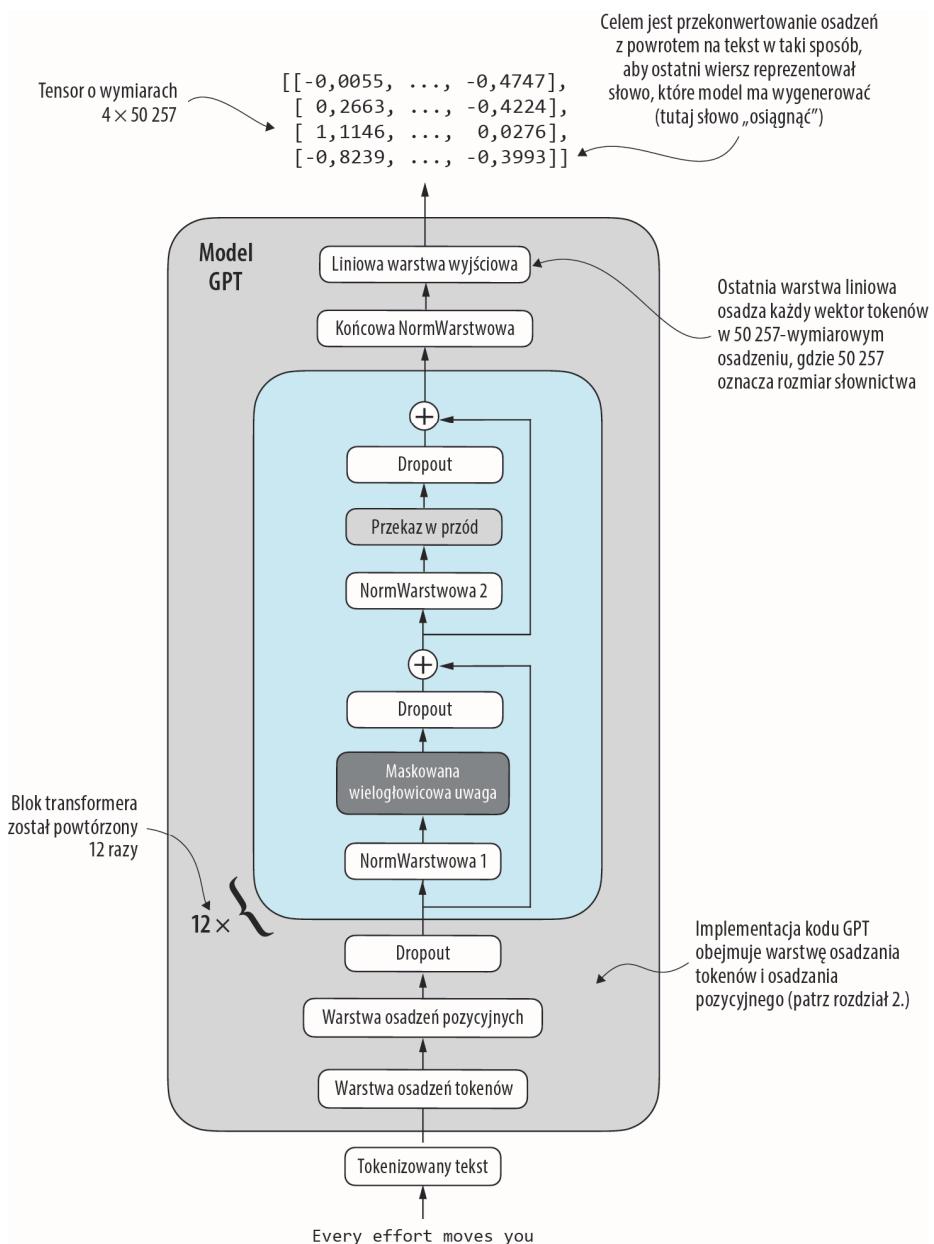
        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Ustawienie urządzenia pozwoli szkolić model na układach CPU lub GPU, w zależności od tego, na którym urządzeniu są dane wejściowe

Dzięki klasie TransformerBlock klasa GPTModel jest stosunkowo mała i kompaktowa.

Konstruktor __init__ klasy GPTModel inicjalizuje warstwy osadzeń tokenów i pozycji z użyciem konfiguracji przekazanych za pośrednictwem słownika Pythona, cfg.



Rysunek 4.15. Przegląd architektury modelu GPT pokazujący przepływ danych przez model GPT. Począwszy od dołu tokenizowany tekst jest najpierw konwertowany na osadzenia tokenów, które są następnie uzupełniane o osadzenia pozycyjne. Ta połączona informacja tworzy tensor, który przechodzi przez ciąg pokazanych w środku bloków transformera (każdy zawiera wielogłowicową uwagę i warstwy sieci neuronowej z przekazem w przód z dropoutem i normalizacją warstwową), ułożonych w stos i powtarzonych 12 razy

Te warstwy osadzeń są odpowiedzialne za przekształcanie indeksów tokenów wejściowych w gęste wektory i dodanie informacji o pozycji (patrz rozdział 2.).

Następnie metoda `__init__` tworzy sekwencyjny stos modułów `TransformerBlock` równy liczbie warstw określonej w `cfg`. Za blokami transformera stosowana jest warstwa `LayerNorm`, standaryzująca wyjścia z bloków transformera i pozwalająca ustabilizować proces uczenia. Na koniec definiujemy liniową głowicę wyjściową bez stronniczości, która rzutuje dane wyjściowe transformera na przestrzeń słownikową tokenizera w celu wygenerowania logitów dla każdego tokena w słowniku.

Metoda `forward` pobiera partię indeksów tokenów wejściowych, oblicza ich osadzenia, stosuje osadzenia pozycyjne, przekazuje sekwencję przez bloki transformera, normalizuje końcowe dane wyjściowe, a następnie oblicza logity reprezentujące nieznormalizowane prawdopodobieństwa następnego tokena. W kolejnym podrozdziale pokażę sposób konwersji tych logitów na tokeny i wyjścia tekstowe.

W kolejnym kroku zainicjujemy 124-milionowy model GPT. W tym celu przekazujemy za pomocą parametru `cfg` słownik `GPT_CONFIG_124M` i przekazujemy do niego utworzoną wcześniej partię tekstowych danych:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Partia wejściowa:\n", batch)
print("\nKształt wyjścia:", out.shape)
print(out)
```

Ten kod wypisuje zawartość partii wejściowej, za którą występuje wyjściowy tensor:

Partia wejściowa: <code>tensor([[6109, 3626, 6100, 345], [6109, 1110, 6622, 257]])</code>	
Kształt wyjścia: <code>torch.Size([2, 4, 50257])</code> <code>tensor([[[[0.1381, 0.0077, -0.1963, ..., -0.0222, -0.1060, 0.1717], [0.3865, -0.8408, -0.6564, ..., -0.5163, 0.2369, -0.3357], [0.6989, -0.1829, -0.1631, ..., 0.1472, -0.6504, -0.0056], [-0.4290, 0.1669, -0.1258, ..., 1.1579, 0.5303, -0.5549]], [[0.1094, -0.2894, -0.1467, ..., -0.0557, 0.2911, -0.2824], [0.0882, -0.3552, -0.3527, ..., 1.2930, 0.0053, 0.1898], [0.6091, 0.4702, -0.4094, ..., 0.7688, 0.3787, -0.1974], [-0.0612, -0.0737, 0.4751, ..., 1.2463, -0.3834, 0.0609]]])</code>	Identyfikatory tokenów tekstu nr 1 Identyfikatory tokenów tekstu nr 2

Jak widać, tensor wyjściowy ma kształt `[2, 4, 50257]`. Wynika to z przekazania dwóch tekstów wejściowych, po cztery tokeny każdy. Ostatni wymiar, 50257, odpowiada rozmiarowi słownictwa tokenizera. W dalszej części rozdziału pokażę, jak przekonwertować każdy z tych 50 257-wymiarowych wektorów wyjściowych z powrotem na tokeny.

Zanim przejdę do kodowania funkcji konwertującej dane wyjściowe modelu na tekst, poświęćmy nieco więcej czasu samej architekturze modelu i analizie jego rozmiaru.

Za pomocą metody `numel()` (skrót od *number of elements* – liczba elementów) można wyznaczyć łączną liczbę parametrów w tensorach parametrów modelu:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Całkowita liczba parametrów: {total_params:,}".replace(","," " ))
```

Wynik jest następujący:

```
Całkowita liczba parametrów: 163 009 536
```

Uważni Czytelnicy prawdopodobnie zauważą pewną rozbieżność. Wcześniej wspominałem o inicjalizacji modelu GPT o 124 milionach parametrów, więc dlaczego rzeczywista liczba parametrów wynosi 163 miliony?

Powodem jest tak zwane *współdzielenie wag* (ang. *weight tying*), które wykorzystano w oryginalnej architekturze GPT-2. Oznacza to, że w oryginalnej architekturze modelu GPT-2 w warstwie wyjścia ponownie zastosowano wagi z warstwy osadzeń tokenów. Aby zrozumieć to lepiej, przyjrzymy się kształtom warstwy osadzania tokenów i liniowej warstwy wyjściowej, zainicjowanych wcześniej w zmiennej `model` za pomocą klasy `GPTModel`:

```
print("Kształt warstwy osadzania tokena:", model.tok_emb.weight.shape)
print("Kształt warstwy wyjściowej:", model.out_head.weight.shape)
```

Jak można zauważyc na podstawie wyników, tensory wag dla obu tych warstw mają ten sam kształt:

```
Kształt warstwy osadzania tokena: torch.Size([50257, 768])
Kształt warstwy wyjściowej: torch.Size([50257, 768])
```

Ze względu na liczbę wierszy odpowiadających 50 257 pozycjom w słowniku tokenizera warstwy osadzania tokenów i warstwy wyjściowej są bardzo duże. Spróbujmy usunąć liczbę parametrów warstwy wyjściowej z całkowitej liczby modelu GPT-2 zgodnie z zasadą współdzielenia wag:

```
total_params_gpt2 =
    total_params - sum(p.numel()
        for p in model.out_head.parameters())
)
print(f'Liczba trenowańnych parametrów '
      f'z uwzględnieniem współdzielenia wag: {total_params_gpt2:,}'.replace(","," " ))
```

Oto uzyskany wynik:

```
Liczba trenowańnych parametrów z uwzględnieniem współdzielenia wag: 124 412 160
```

Jak można zauważyc, model ma teraz tylko 124 miliony parametrów, co odpowiada oryginalnemu rozmiarowi modelu GPT-2.

Współdzielenie wag zmniejsza ogólną ilość zajmowanej pamięci i złożoność obliczeniową modelu. Z moich doświadczeń wynika jednak, że stosowanie oddzielnych warstw osadzania tokenów i warstw wyjściowych skutkuje lepszym uczeniem i większą wydajnością modelu. Z tego względu w implementacji klasy `GPTModel` używamy osobnych warstw. To samo dotyczy nowoczesnych modeli LLM. Do pojęcia współdzielenia

wag powrócimy w rozdziale 6., przy okazji ładowania wstępnie przeszkolonych wag z OpenAI.

Ćwiczenie 4.1. Liczba parametrów w modułach ze sprzężeniem w przód i w module uwagi

Oblicz i porównaj liczby parametrów zawartych w module ze sprzężeniem w przód oraz tych, które są zawarte w module wielogłowicowej uwagi.

Na koniec obliczymy zapotrzebowanie na pamięć dla 163 milionów parametrów w obiekcie GPTModel:

```
total_size_bytes = total_params * 4           ← Oblicza całkowity rozmiar w bajtach (przy założeniu,
total_size_mb = total_size_bytes / (1024 * 1024) ← że każdy parametr typu float32 obejmuje 4 bajty)
print(f"Całkowity rozmiar modelu: {total_size_mb:.2f} MB")   ← Konwersja
                                                               na megabajty
```

Wynik jest następujący:

Całkowity rozmiar modelu: 621.83 MB

Podsumowując, z obliczeń zapotrzebowania na pamięć dla 163 milionów parametrów w obiekcie GPTModel — i przy założeniu, że każdy parametr jest 32-bitową liczbą zmiennoprzecinkową zajmującą 4 bajty — wynika, że całkowity rozmiar modelu wynosi 621,83 MB, co ilustruje stosunkowo dużą objętość pamięci wymaganą do pomieszczenia nawet stosunkowo małych modeli LLM.

Po zaimplementowaniu architektury GPTModel przekonaliśmy się, że jej wyjście zawiera tensorы numeryczne o kształcie [batch_size, num_tokens, vocab_size]. W kolejnym kroku stworzymy kod odpowiedzialny za konwersję tych tensorów wyjściowych na tekst.

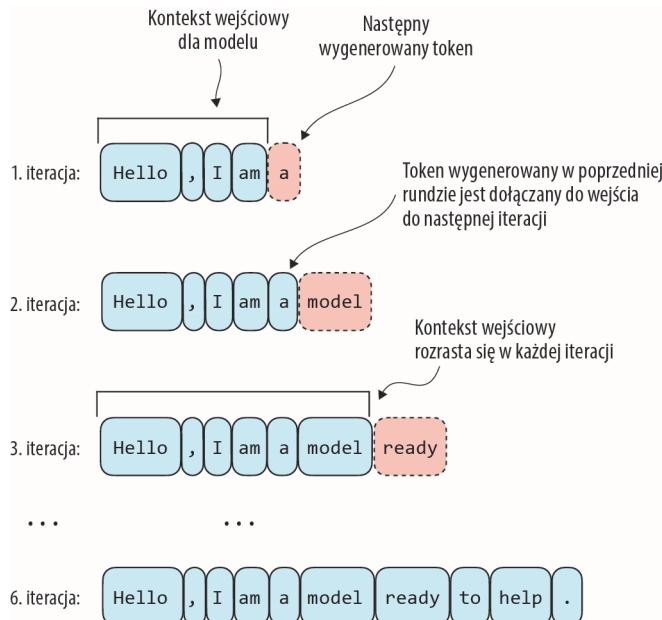
Ćwiczenie 4.2. Inicjalizacja większych modeli GPT

Właśnie zainicjowałeś model GPT o 124 milionach parametrów, znany pod nazwą „GPT-2 small”. Bez wprowadzania jakichkolwiek modyfikacji w kodzie poza aktualizacją pliku konfiguracyjnego użyj klasy GPTModel w celu zaimplementowania modelu „GPT-2 medium” (1024-wymiarowe osadzenia, 24 bloki transformera, 16 wielogłowicowych głowic uwagi), modelu „GPT-2 large” (1280-wymiarowe osadzenia, 36 bloków transformera, 20 wielogłowicowych głowic uwagi) i modelu „GPT-2 XL” (1600-wymiarowe osadzenia, 48 bloków transformera, 25 wielogłowicowych głowic uwagi). W ramach zadania dodatkowego oblicz całkowitą liczbę parametrów w każdym modelu GPT.

4.7. Generowanie tekstu

Teraz zaimplementujemy kod, który konwertuje wyjścia tensora modelu GPT z powrotem na tekst. Zanim zaczniemy, przyjrzyjmy się pokrótce, w jaki sposób model generatywny, taki jak LLM, generuje tekst po jednym słowie (lub tokenie) na raz.

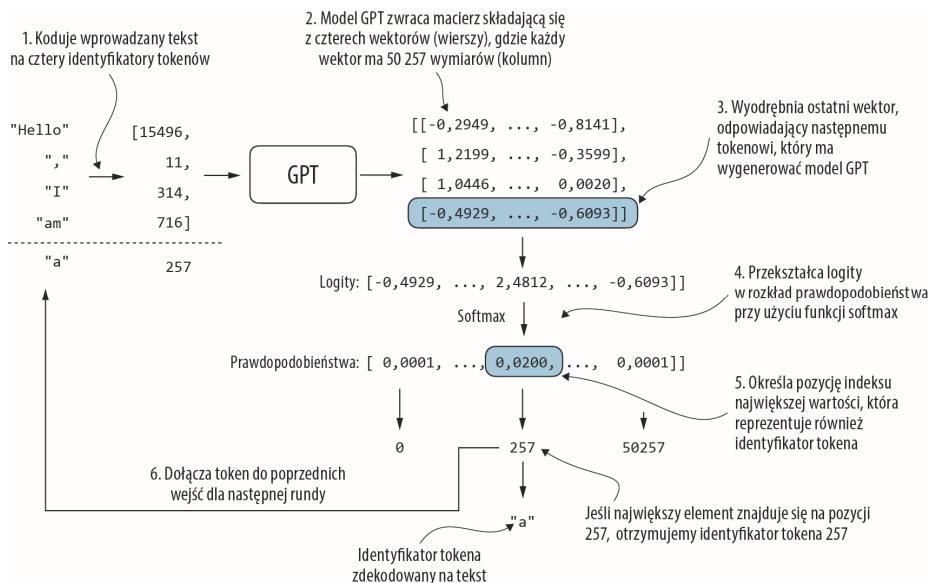
Po szczególne kroki procesu, w którym model GPT generuje tekst, z uwzględnieniem kontekstu wejściowego w postaci frazy „Hello, I am”, są przedstawione na rysunku 4.16. W każdej iteracji kontekst wejściowy rozszerza się, co umożliwia modelowi generowanie spójnego i odpowiedniego do kontekstu tekstu. W szóstej iteracji model skonstruował pełne zdanie: „Cześć, jestem modelem gotowym do pomocy”. Przekonałeś się, że obecna implementacja klasy `GPTModel` generuje tensory o kształcie `[batch_size, num_token, vocab_size]`. Teraz pytanie brzmi: W jaki sposób model GPT przechodzi od tych wyjściowych tensorów do wygenerowanego tekstu?



Rysunek 4.16. Proces generowania przez model LLM tekstu krok po kroku, po jednym tokenie na raz. Począwszy od startowego kontekstu wejściowego („Cześć, jestem”), model przewiduje podczas każdej iteracji kolejny token, po czym dołącza go do kontekstu wejściowego w następnej rundzie prognozowania. Jak widać, w pierwszej iteracji model dodaje słowo „a”, w drugim „model”, a w trzeciej „ready”. W ten sposób stopniowo buduje zdanie

Proces, w którym model GPT przechodzi od wyjściowych tensorów do wygenerowanego tekstu, obejmuje kilka kroków, które zilustrowałem na rysunku 4.17. Te kroki obejmują dekodowanie tensorów wyjściowych, wybieranie tokenów na podstawie rozkładu prawdopodobieństw i konwertowanie tych tokenów na tekst czytelny dla człowieka.

Proces generowania następnego tokena pokazany na rysunku 4.17 ilustruje pojedynczy krok, w którym model GPT na podstawie danych wejściowych generuje następny token. W każdym kroku model generuje macierz z wektorami reprezentującymi potencjalne następne tokeny. Wektor odpowiadający następnemu tokenowi jest wyodrębniany i przekształcany na rozkład prawdopodobieństwa za pomocą funkcji softmax.



Rysunek 4.17. Mechanika generowania tekstu w modelu GPT na podstawie pojedynczej iteracji w procesie generowania tokena. Proces rozpoczyna się od zakodowania tekstu wejściowego w identyfikatory tokenów, które są następnie wprowadzane do modelu GPT. Następnie wyjścia modelu są poddawane konwersji z powrotem na tekst i dołączane do oryginalnego tekstu wejściowego

W wektorze zawierającym wyniki prawdopodobieństwa znajduje się indeks największej wartości, który przekłada się na identyfikator tokena. Ten identyfikator tokena jest następnie dekodowany na tekst, co tworzy następny token w sekwencji. Na koniec ten token jest dołączany do wcześniejszych danych wejściowych, co tworzy nową sekwencję wejściową dla kolejnej iteracji. Ten proces umożliwia modelowi sekwencyjne generowanie tekstu, polegające na budowaniu spójnych fraz i zdań na podstawie początkowego kontekstu wejściowego.

W praktyce ten proces jest powtarzany przez wiele iteracji (rysunek 4.16), aż do osiągnięcia określonej przez użytkownika liczby wygenerowanych tokenów. Proces generowania tokenów można zaimplementować w kodzie w sposób pokazany na listingu 4.8.

Listing 4.8. Funkcja modelu GPT do generowania tekstu

```
def generate_text_simple(model, idx, max_new_tokens, context_size):  
    idx jest tablicą indeksów w bieżącym kontekście (partia, liczba_tokenów)  
    for _ in range(max_new_tokens):  
        idx_cond = idx[:, -context_size:] ← Przyjęcie bieżącego kontekstu, jeśli przekracza obsługiwany rozmiar. Na przykład, jeśli LLM obsługuje tylko 5 tokenów, a rozmiar kontekstu wynosi 10, to jako kontekst będzie używane tylko ostatnie 5 tokenów  
        with torch.no_grad():  
            logits = model(idx_cond)
```

```

idx_next ← Koncentrujemy się tylko na ostatnim kroku
ma kształt ← czasowym, więc tablica (partia, liczba_tokenów,
(partia, 1)   rozmiar_słownictwa) zostaje przekształcona
              na tablicę (partia, rozmiar_słownictwa)

logits = logits[:, -1, :] ← Tensor probas ma kształt
probas = torch.softmax(logits, dim=-1) ← (partia, rozmiar_słownictwa)
idx_next = torch.argmax(probas, dim=-1, keepdim=True) ←
idx = torch.cat((idx, idx_next), dim=1) ←

return idx ← Dodaje do działającej sekwencji próbkowany indeks,
              gdzie idx ma kształt (partia, liczba_tokenów+1)

```

Powyższy kod demonstruje prostą implementację generatywnej pętli dla modelu językowego z użyciem biblioteki PyTorch. Kod iteruje po określonej liczbie nowych tokenów do wygenerowania, przycina bieżący kontekst, aby dopasować go do maksymalnego rozmiaru kontekstu modelu, oblicza prognozy, a następnie, na podstawie prognozy o najwyższym prawdopodobieństwie, wybiera następny token.

Aby zakodować funkcję `generate_text_simple`, używamy funkcji `softmax` w celu przekształcenia logitów w rozkład prawdopodobieństwa, z którego za pomocą `torch.argmax` wyznaczamy pozycję o najwyższej wartości. Funkcja `softmax` jest monotoniczna, co oznacza, że przy przekształcaniu na wyjścia zachowuje kolejność wejść. W praktyce krok polegający na zastosowaniu funkcji `softmax` jest więc zbędny, ponieważ pozycja z najwyższą oceną w tensorze wyjściowym `softmax` odpowiada pozycji w tensorze logitów. Mówiąc inaczej, bezpośrednie zastosowanie funkcji `torch.argmax` do tensora `logits` pozwala uzyskać identyczne wyniki. Aby zilustrować kompletny proces transformacji logitów na prawdopodobieństwa, udostępniłem jednak kod do konwersji. Dzięki temu będziesz mógł lepiej zrozumieć sposób generowania przez model najbardziej prawdopodobnego następnego tokena. Ten proces jest znany jako *zachłanne dekodowanie* (ang. *greedy decoding*).

Podeczas implementacji kodu szkoleniowego GPT w następnym rozdziale, aby zmodyfikować wyjścia `softmax`, tak by model nie zawsze wybierał najbardziej prawdopodobny token, użyjemy dodatkowych technik próbkowania. Wprowadzi to zmienność i kreatywność do generowanego tekstu.

Proces generowania jednego identyfikatora tokena na raz i dołączania go do kontekstu za pomocą funkcji `generate_text_simple` jest dokładniej zilustrowany na rysunku 4.18 (proces generowania identyfikatora tokena dla każdej iteracji jest szczegółowo opisany na rysunku 4.17). Identyfikatory tokenów są generowane w sposób iteracyjny. Na przykład w iteracji 1 model otrzymuje tokeny odpowiadające frazie „Hello, I am”, przewiduje następny token (o identyfikatorze 257, czyli „a”) i dołącza go do danych wejściowych. Ten proces jest powtarzany do chwili, gdy po sześciu iteracjach model wygeneruje pełne zdanie „Hello, I am model ready to help”.

Wypróbujmy teraz funkcję `generate_text_simple` z kontekstem „Hello, I am” jako danymi wejściowymi modelu. Najpierw kodujemy kontekst wejściowy na identyfikatory tokenów:

```

start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
print("encoded:", encoded)

```

```
encoded_tensor = torch.tensor(encoded).unsqueeze(0) ← Dodaje wymiar parti
print("encoded_tensor.shape:", encoded_tensor.shape)
```

Oto zakodowane identyfikatory:

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

W kolejnym kroku przełączymy model w tryb `.eval()`. Powoduje to wyłączenie losowych komponentów używanych tylko podczas szkolenia, takich jak dropout, i użycie funkcji `generate_text_simple` na zakodowanym tensorze wejściowym:

```
model.eval() ← Wyłącza dropout, ponieważ nie szkolimy modelu
out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=5,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Wyjście:", out)
print("Rozmiar wyjścia:", len(out[0]))
```

Oto wynikowe identyfikatory tokenów wyjściowych:

```
Wyjście: tensor([[15496,     11,    314,    716, 27018, 24086, 47843, 30961, 42348, 7267]])
Rozmiar wyjścia: 10
```

Za pomocą metody `.decode` tokenizera można przekonwertować identyfikatory z powrotem na tekst:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

Oto wyjście modelu w formacie tekstowym:

```
Hello, I am Featureiman Byeswickattribute argue
```

Jak można zauważyć, model wygenerował bełkot, który w niczym nie przypomina spójnego tekstu „Hello, I am a model ready to help”. Co się stało? Otóż model nie jest w stanie wygenerować spójnego tekstu, ponieważ jeszcze nie został przeszkolony. Do tej pory zaimplementowaliśmy tylko architekturę GPT i zainicjalizowaliśmy egzemplarz modelu GPT z początkowymi, losowymi wagami. Szkolenie modeli to obszerny temat sam w sobie. Zajmę się nim w następnym rozdziale.

Ćwiczenie 4.3. Korzystanie z osobnych parametrów dropoutu

Na początku tego rozdziału zdefiniowaliśmy w słowniku `GPT_CONFIG_124M` globalne ustawienie `drop_rate`, którego celem było ustalenie szybkości odrzucania w różnych miejscach architektury `GPTModel`. Zmodyfikuj ten kod, aby określić oddzielną wartość dropoutu dla różnych warstw dropoutu w całej architekturze modelu (wskazówka: są trzy różne miejsca, w których użyto warstw dropoutu: warstwa osadzania, warstwa skrótów i wielogłowicowy moduł uwagi).

Podsumowanie

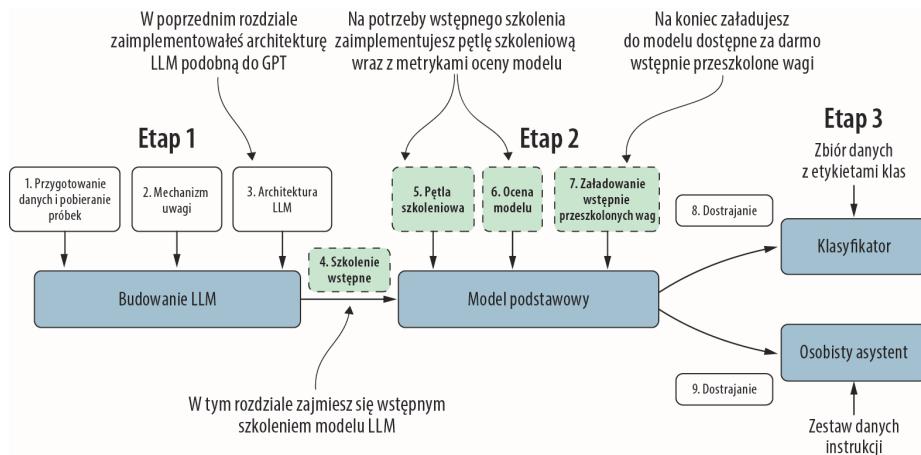
- Normalizacja warstwowa stabilizuje szkolenie, dzięki czemu wyjścia każdej warstwy mają spójną średnią i wariancję.
- Połączenia skrótowe pomijają warstwę lub większą liczbę warstw i przekazują wyjście jednej warstwy bezpośrednio do warstwy znajdującej się głębiej. Pomaga to złagodzić problem zanikającego gradientu podczas szkolenia głębokich sieci neuronowych, takich jak modele LLM.
- Podstawowym elementem strukturalnym modeli GPT są bloki transformera, które łączą maskowane wielogłowicowe moduły uwagi z w pełni połączonymi sieciami ze sprzężeniem w przód, wykorzystującymi funkcję aktywacji GELU.
- GPT to modele LLM z wieloma powtarzonymi blokami transformerów, które mają od kilku milionów do kilku miliardów parametrów.
- Modele GPT występują w różnych rozmiarach, na przykład 124, 345, 762 i 1542 milionów parametrów, które można zaimplementować za pomocą tej samej klasy Pythona GPTModel.
- Zdolność modelu LLM podobnego do GPT do generowania tekstu polega na dekodowaniu tensorów wyjściowych na tekst czytelny dla człowieka przez sekwenncyjne przewidywanie jednego tokena na raz na podstawie kontekstu wejściowego.
- Bez szkolenia model GPT generuje niespójny tekst, co podkreśla znaczenie szkolenia modelu.

Wstępne szkolenie na nieoznakowanych danych

W tym rozdziale:

- Obliczanie strat dla zbioru szkoleniowego i walidacyjnego w celu oceny jakości tekstu generowanego przez model LLM podczas szkolenia
- Implementacja funkcji szkoleniowej i wstępne szkolenie modelu LLM
- Zapisywanie i ładowanie wag modelu w celu kontynuowania szkolenia modelu LLM
- Ładowanie wstępnie przeszkołonych wag z OpenAI

Dotychczas zaimplementowałeś mechanizm próbkowania danych i uwagi oraz zakodowałeś architekturę LLM. Nadszedł czas na implementację funkcji szkoleniowej i wstępne szkolenie modelu LLM. Zapoznasz się z podstawowymi technikami oceny modelu, pozwalającymi zmierzyć jakość wygenerowanego tekstu, co jest warunkiem koniecznym optymalizacji modelu LLM podczas procesu szkolenia. Ponadto omówię sposób ładowania wstępnie przeszkołonych wag, który daje modelowi LLM solidny punkt wyjścia do dostrajania. Ogólny plan podkreślający zagadnienia, które omówię w tym rozdziale, przedstawiłem na rysunku 5.1.



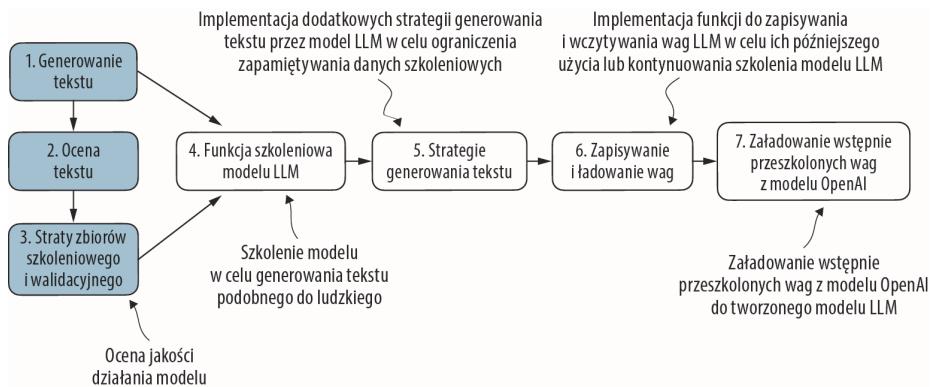
Rysunek 5.1. Trzy główne etapy kodowania modelu LLM. Ten rozdział koncentruje się na etapie 2. — wstępnym szkoleniu modelu LLM (krok 4.), obejmującym implementację kodu szkoleniowego (krok 5.), ocenę wydajności (krok 6.), a także zapisywanie wag modelu oraz ich ładowanie (krok 7.)

Parametry wag

W kontekście modeli LLM i innych modeli uczenia głębokiego *wagi* odnoszą się do możliwych do wyszkolenia parametrów, dostrajanych w procesie uczenia. Te wagi są również znane jako *parametry wag* lub po prostu *parametry*. W takich frameworkach jak PyTorch te wagi są przechowywane w warstwach liniowych. Skorzystaliśmy z nich do implementacji modułu uwagi wielogłowicowej w rozdziale 3. oraz klasy GPTModel w rozdziale 4. Po zainicjowaniu warstwy (`new_layer = torch.nn.Linear(...)`) można uzyskać dostęp do jej wag za pośrednictwem atrybutu `.weight`, `new_layer.weight`. Dodatkowo, dla wygody, PyTorch umożliwia bezpośredni dostęp za pomocą metody `model.parameters()` do wszystkich możliwych do przeszkolenia parametrów modelu, w tym wag i stronniczości. Z tej metody skorzystamy później podczas implementacji szkolenia modelu.

5.1. Ocena generatywnych modeli tekstowych

Po krótkim przypomnieniu zasad generowania tekstu z rozdziału 4. skonfigurujemy model LLM do generowania tekstu. Następnie omówię podstawowe sposoby oceny jakości wygenerowanego tekstu. Potem obliczymy straty zbiorów szkoleniowego i walidacyjnego. Tematy, którymi zajmę się w tym rozdziale, przedstawiłem, z wyróżnionymi trzema pierwszymi krokami, na rysunku 5.2.



Rysunek 5.2. Przegląd tematów omawianych w tym rozdziale. Zacznę od przypomnienia procesu generowania tekstu (krok 1.), po czym przejdę do opisu podstawowych technik oceny modelu (krok 2.) oraz strat szkoleniowych i walidacyjnych (krok 3.).

5.1.1. Używanie modelu GPT do generowania tekstu

Skonfigurujmy model LLM i krótko podsumujmy zaimplementowany w rozdziale 4. proces generowania tekstu. Zaczynamy od inicjalizacji modelu GPT z użyciem klasy `GPTModel` i słownika `GPT_CONFIG_124M` (patrz rozdział 4.). Tak skonfigurowany model będzie później poddany ocenie i szkoleniu:

```

import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,           ← Skróciliśmy długość kontekstu z 1024 do 256 tokenów
    "context_length": 256,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,             ← Możliwe i powszechnie stosowane jest ustawienie parametru dropout na 0
    "qkv_bias": False
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()

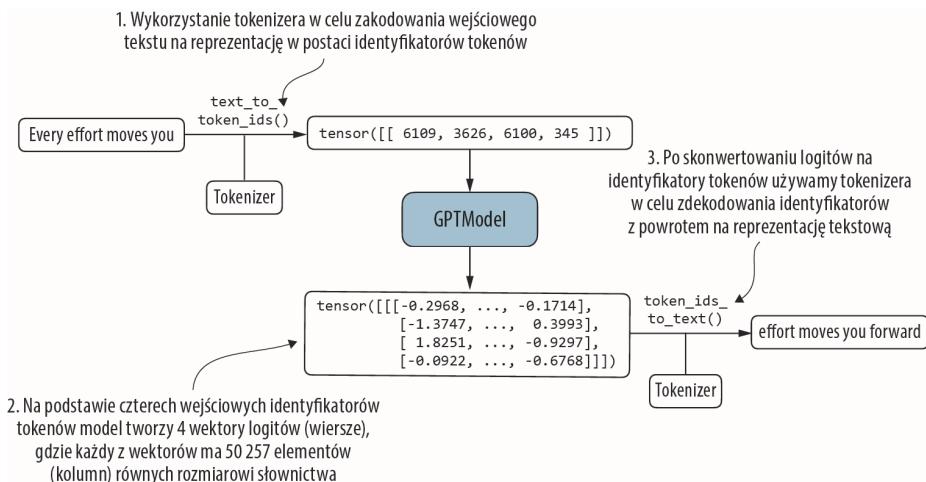
```

Patrząc na słownik `GPT_CONFIG_124M`, można zauważyc, że jedyną wprowadzoną zmianą w porównaniu z poprzednim rozdziałem jest zmniejszenie rozmiarów kontekstu (`context_length`) do 256 tokenów. Ta modyfikacja zmniejsza wymagania obliczeniowe związane ze szkoleniem modelu, co umożliwia przeprowadzenie szkolenia na standardowym laptopie.

Pierwotnie model GPT-2 obejmujący 124 miliony parametrów był skonfigurowany do obsługi do 1024 tokenów. Po zakończeniu procesu szkolenia zaktualizujemy ustawienie rozmiaru kontekstu i załadujemy wstępnie przeszkołone wagę do pracy z modelem skonfigurowanym dla kontekstu o długości wynoszącej 1024 tokeny.

Korzystając z egzemplarza klasy `GPTModel` jako punktu wyjścia, zaadaptujemy funkcję `generate_text_simple` z rozdziału 4. i wprowadzimy dwie przydatne funkcje: `text_to_token_ids` i `token_ids_to_text`. Te funkcje ułatwią konwersję między reprezentacją tekstu a tokenową, którą będziemy wykorzystywać w tym rozdziale.

Trzyetapowy proces generowania tekstu z użyciem modelu GPT zilustrowałem na rysunku 5.3. Najpierw tokenizer konwertuje tekst wejściowy na ciąg identyfikatorów tokenów (patrz rozdział 2.). Następnie model pobiera te identyfikatory tokenów i generuje odpowiednie logity, które są wektorami reprezentującymi rozkład prawdopodobieństw dla poszczególnych tokenów w słowniku (patrz rozdział 4.). W kolejnym kroku te logity są ponownie konwertowane na identyfikatory tokenów, które tokenizer dekoduje na tekst czytelny dla człowieka. Na tym kończy się cykl przekształcania tekstu wejściowego na tekstowe wyjście.



Rysunek 5.3. Generowanie tekstu obejmuje kodowanie tekstu do postaci identyfikatorów tokenów, które model LLM przetwarza na postać wektorów złożonych z logitów. Wektory logitów są następnie konwertowane z powrotem na identyfikatory tokenów, po czym są detokenizowane do reprezentacji tekstuowej

Proces generowania tekstu można zaimplementować w sposób pokazany na listingu 5.1.

Listing 5.1. Funkcje narzędziowe do konwersji tekstu na identyfikatory tokenów

```
import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    encoded_tensor = torch.tensor(encoded).unsqueeze(0)
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    tokens = tokenizer.decode(token_ids)
    return tokens
```

unsqueeze(0) dodaje wymiar partií

```

flat = token_ids.squeeze(0)           ← Usunięcie wymiaru partii
return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Tekst wynikowy:\n", token_ids_to_text(token_ids, tokenizer))

```

Z pomocą tego kodu model generuje następujący tekst:

Tekst wynikowy:

Every effort moves you rentingetic wasn? refres RexMeCHicular stren

Najwyraźniej model nie generuje jeszcze spójnego tekstu, ponieważ nie został poddany szkoleniu. Aby określić, co sprawia, że tekst jest „spójny” lub „wysokiej jakości”, trzeba zaimplementować numeryczną metodę oceny wygenerowanej treści. Takie podejście pozwala monitorować wydajność modelu przez cały proces szkolenia i poprawiać ją.

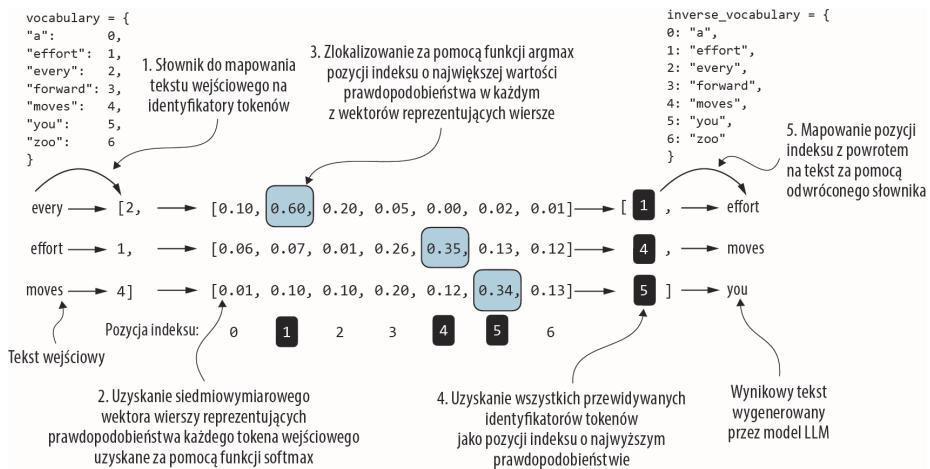
W kolejnym kroku obliczymy *wskaźnik strat* dla wygenerowanych wyników. Wielkość straty odgrywa rolę wskaźnika postępu i sukcesu postępu w szkoleniu. W późniejszych rozdziałach, przy okazji dostrajania modelu LLM, dokonam przeglądu dodatkowych metodyk oceny jakości modelu.

5.1.2. Obliczanie strat związanych z generowaniem tekstu

W tym punkcie poddam analizie techniki numerycznej oceny jakości tekstu generowanego podczas szkolenia przez obliczanie *straty generowania tekstu*. Temat omówię krok po kroku na praktycznym przykładzie tak, aby pojęcia były jasne i możliwe do zastosowania. Rozpoczну od krótkiego podsumowania sposobu ładowania danych i generowania tekstu za pomocą funkcji `generate_text_simple`.

Ogólny przepływ od tekstu wejściowego do tekstu wygenerowanego przez model LLM z użyciem pięcioetapowej procedury przedstawiłem na rysunku 5.4. Pokazany na rysunku proces generowania tekstu ilustruje sposób wewnętrznego działania funkcji `generate_text_simple`. Przed obliczeniem straty, za pomocą której w dalszej części tego podrozdziału zmierzymy jakość wygenerowanego tekstu, trzeba wykonać te same początkowe kroki.

Na rysunku 5.4 przedstawiłem proces generowania tekstu z użyciem słownika składającego się z siedmiu słów tak, aby ilustracja zmieściła się na jednej stronie. Model GPTModel działa jednak ze znacznie większym słownikiem, składającym się z 50 257 słów; dlatego identyfikatory tokenów w poniższym kodzie będą mieściły się w przedziale od 0 do 50 256, a nie od 0 do 6.



Rysunek 5.4. Dla każdego z trzech tokenów wejściowych pokazanych po lewej stronie rysunku obliczamy wektor zawierający wyniki prawdopodobieństwa odpowiadające poszczególnym tokenom w słowniku. Pozycja indeksu najwyższego wyniku prawdopodobieństwa w każdym wektorze reprezentuje najbardziej prawdopodobny identyfikator następnego tokena. Identyfikatory tokenów powiązane z najwyższymi wynikami prawdopodobieństwa są wybierane i mapowane z powrotem na tekst, który reprezentuje tekst wygenerowany przez model

Na rysunku 5.4 dla uproszczenia uwzględniono tylko jeden przykład tekstu („every effort moves”). W poniższym praktycznym przykładzie kodu, który implementuje kroki przedstawione na rysunku, będziemy pracować z dwoma przykładami wejściowymi dla modelu GPT („every effort moves” i „I really like”)¹.

Rozważmy te dwa przykłady wejściowego tekstu, które już zostały zmapowane na identyfikatory tokenów (rysunek 5.4, krok 1.):

```
inputs = torch.tensor([[16833, 3626, 6100], # "every effort moves",
                      [40, 1107, 588]]) # "I really like"])
```

Po dopasowaniu tych wejść zmenna targets zawiera identyfikatory tokenów, które mają być generowane przez model:

```
targets = torch.tensor([[3626, 6100, 345], # "effort moves you",
                       [1107, 588, 11311]]) # "really like chocolate"])
```

Zauważ, że cele (targets) odpowiadają tekstem wejściowym (inputs) przesuniętym o jedną pozycję do przodu. To pojęcie omawiałem w rozdziale 2., przy okazji implementacji mechanizmu ładującego dane. Strategia przesuwania ma kluczowe znaczenie dla procesu uczenia modelu przewidywania następnego tokena w sekwencji.

¹ „every effort moves” — „każdy wysiłek porusza”; „I really like” — „(ja) naprawdę lubię” — przyp. tłum.

² „effort moves you” — „wysiłek porusza Cię”; „really like chocolate” — „naprawdę lubię czekoladę” — przyp. tłum.

W kolejnym kroku przekazujemy do modelu dane wejściowe, aby dla dwóch przykładów wejścia, z których każdy zawiera po trzy tokeny, obliczyć wektory logitów. Następnie, aby przekształcić te logity w oceny prawdopodobieństwa (zapisane w zmiennej probas; rysunek 5.4, krok 2.), stosujemy funkcję softmax:

```
with torch.no_grad():
    logits = model(inputs)
    probas = torch.softmax(logits, dim=-1)
    print(probas.shape)
```

←
Prawdopodobieństwo odpowiadające
każdemu tokenowi w słowniku

Wyłącza śledzenie gradientu,
ponieważ jeszcze nie szkolimy
modelu

Wynikowy wymiar tensora ocen prawdopodobieństwa (probas) to:

```
torch.Size([2, 3, 50257])
```

Pierwsza liczba, 2, odpowiada dwóm przykładom (wierszom) w danych wejściowych. Tę wielkość określa się również jako rozmiar partii. Druga liczba, 3, odpowiada liczbie tokenów w każdym wejściu (wierszu). Wreszcie ostatnia liczba odpowiada wymiarowości osadzeń, która jest określona przez rozmiar słownictwa. Po konwersji logitów na prawdopodobieństwa za pomocą funkcji softmax funkcja generate_text_simple dokonuje konwersji wynikowych ocen prawdopodobieństwa z powrotem na tekst (rysunek 5.4, kroki 3. – 5.).

Kroki 3. i 4. można wykonać przez zastosowanie do ocen prawdopodobieństwa funkcji argmax. W ten sposób uzyskujemy właściwe identyfikatory tokenów:

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Identyfikatory tokenów:\n", token_ids)
```

Biorąc pod uwagę, że mamy dwie partie wejściowe, z których każda zawiera po trzy tokeny, po zastosowaniu funkcji argmax do ocen prawdopodobieństwa (rysunek 5.4, krok 3.) uzyskamy dwa zbiory wyników, z których każdy zawiera trzy prognozowane identyfikatory tokenów:

```
Identyfikatory tokenów:
tensor([[16657], ← Pierwsza partia
        [ 339],
        [42826]],
       [[49906], ← Druga partia
        [29669],
        [41751]])
```

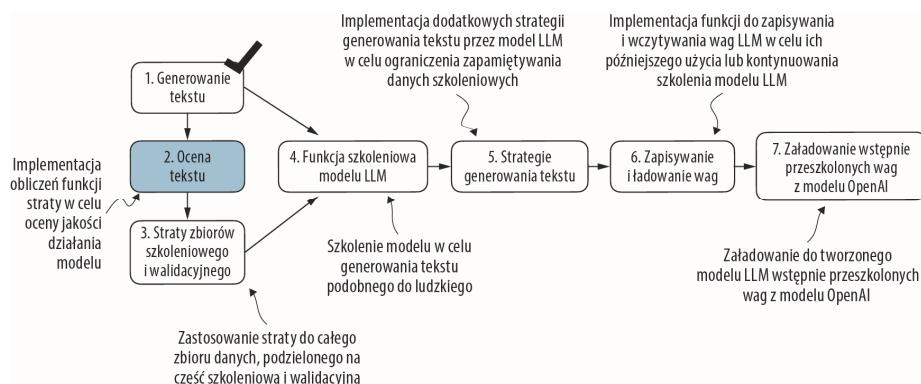
Na koniec, w kroku 5., dokonujemy konwersji identyfikatorów tokenów z powrotem na tekst:

```
print(f"Cel dla partii 1: {token_ids_to_text(targets[0], tokenizer)}")
print(f"Wynik dla partii 1: {token_ids_to_text(token_ids[0].flatten(), tokenizer)}")
f" {token_ids_to_text(token_ids[0].flatten(), tokenizer)}")
```

Po zdekodowaniu tych tokenów okazuje się, że uzyskane tokeny wyjściowe znacznie różnią się od tokenów oznaczonych jako cel, czyli tych, których wygenerowania oczekujemy od modelu:

Cel dla partii 1: effort moves you
 Wynik dla partii 1: Armed heNetflix

Model wygenerował losowy tekst, który różni się od tekstu docelowego. To dlatego, że nie został jeszcze przeszkolony. Chcemy teraz ocenić jakość wygenerowanego przez model tekstu numerycznie za pomocą funkcji straty (rysunek 5.5). Jest to nie tylko przydatne z perspektywy pomiaru jakości wygenerowanego tekstu, ale także stanowi element składowy wymagany do implementacji funkcji szkoleniowej. Ten mechanizm wykorzystamy do aktualizacji wag modelu, by poprawić generowany tekst.

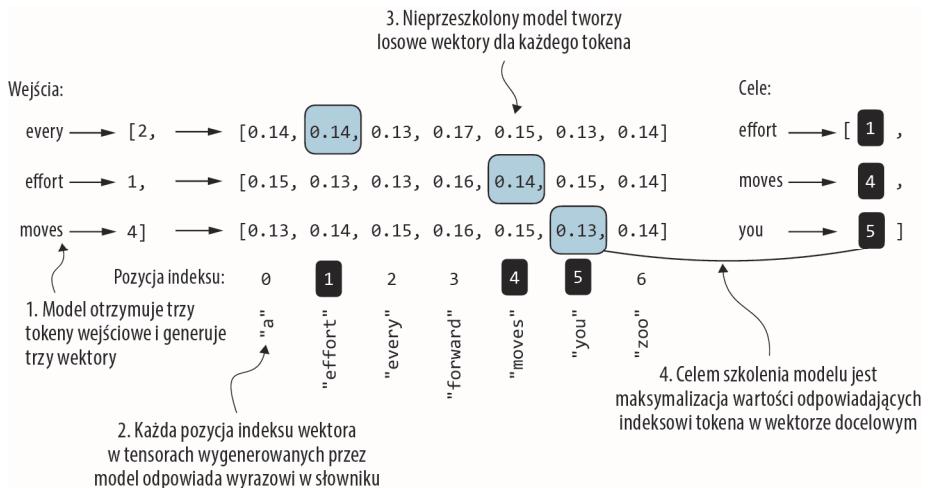


Rysunek 5.5. Przegląd tematów omówionych w tym rozdziale. Zakończyliśmy krok 1. Możemy teraz przejść do implementacji funkcji oceny tekstu (krok 2.).

Jednym z elementów procesu oceny tekstu, które zaimplementujemy, jest zmierzanie „odległości”, jaka dzieli wygenerowane tokeny od prawidłowych prognoz (celów) – patrz rysunek 5.5. Funkcja uczenia, którą zaimplementujemy później, wykorzysta te informacje do dostosowania wag modelu tak, aby model generował tekst, który będzie bardziej przypominał tekst docelowy (lub, w idealnym przypadku, będzie z nim zgodny).

Szkolenie modelu ma na celu zwiększenie prawdopodobieństwa *softmax* w pozycjach indeksu odpowiadających poprawnym identyfikatorom docelowych tokenów (rysunek 5.6). Prawdopodobieństwo *softmax* jest wykorzystywane również w metryce oceny, którą zaimplementujemy w następnej kolejności. Metryka ta pozwala ocenić za pomocą liczbowego wskaźnika jakość wygenerowanych wyników modelu: im wyższe prawdopodobieństwo we właściwych pozycjach, tym lepiej.

Zwróć uwagę, że na rysunku 5.6 przedstawiłem prawdopodobieństwa *softmax* dla słownika składającego się zaledwie siedmiu słów (aby zmieścić wszystko na jednej ilustracji). Z tego powodu początkowe, losowe wartości będą oscylować wokół wartości $\frac{1}{7}$, co odpowiada w przybliżeniu wartości 0,14. Słownictwo, którego używa model GPT-2, ma jednak 50 257 tokenów, zatem większość początkowych prawdopodobieństw będzie oscylować wokół wartości 0,00002 ($\frac{1}{50\,257}$).



Rysunek 5.6. Przed szkoleniem model tworzy losowe wektory prawdopodobieństw dla następnego słowa. Celem szkolenia modelu jest zyskanie pewności, że wartości prawdopodobieństw odpowiadających podświetlonym identyfikatorom docelowych tokenów są jak największe

Początkowe wartości prawdopodobieństw softmax, odpowiadające tokenom docelowym dla każdego z dwóch tekstów wejściowych, można wyświetlić za pomocą następującego kodu:

```
text_idx = 0
target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Tekst 1:", target_probas_1)

text_idx = 1
target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Tekst 2:", target_probas_2)
```

Trzy docelowe prawdopodobieństwa identyfikatorów docelowych tokenów dla każdej partii są następujące:

```
Tekst 1: tensor([7.4540e-05, 3.1061e-05, 1.1563e-05])
Tekst 2: tensor([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

Celem szkolenia modelu LLM jest maksymalizacja prawdopodobieństwa poprawnego tokena, co wiąże się ze zwiększeniem jego prawdopodobieństwa w stosunku do innych tokenów. W ten sposób zyskujemy pewność, że w procesie generowania model LLM będzie konsekwentnie wybierał token docelowy — w praktyce często będzie to następne słowo w zdaniu.

Następnie obliczymy wartości funkcji straty odpowiadające ocenom prawdopodobieństw dla dwóch przykładowych partii, `target_probas_1` i `target_probas_2`. Najważniejsze kroki tego procesu zilustrowałem na rysunku 5.7. Ponieważ w celu uzyskania

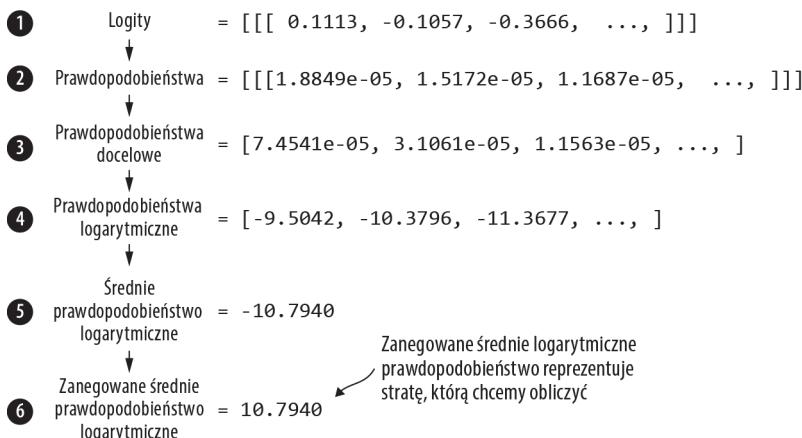
Propagacja wsteczna

Jak zmaksymalizować wartości prawdopodobieństwa *softmax* odpowiadające docelowym tokenom? Najważniejsza jest aktualizacja wag modelu, dzięki której model generuje wyższe wartości dla tych identyfikatorów tokenów, które chcemy wygenerować. Aktualizacja wag odbywa się z zastosowaniem procesu zwanego *propagacją wsteczną* (ang. *backpropagation*) — standardowej techniki uczenia głębszych sieci neuronowych (więcej szczegółów na temat propagacji wstecznej i uczenia modelu można znaleźć w podrozdziałach od A.3 do A.7 w „Dodatku A”).

Propagacja wsteczna potrzebuje funkcji straty, obliczającej różnicę między przewidywanym wynikiem modelu (tutaj prawdopodobieństwami odpowiadającymi identyfikatorom tokenów docelowych) a wynikiem pożądany. Wspomniana funkcja straty jest miarą odległości prognoz modelu od zgodnych z założeniami wartości docelowych.

wartości `target_probas_1` i `target_probas_2` zastosowaliśmy już kroki od 1. do 3., teraz przejdziemy do kroku 4., w którym do wyników prawdopodobieństwa zastosujemy funkcję *logarytmu*:

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2)))
print(log_probas)
```



Rysunek 5.7. Obliczanie funkcji straty obejmuje kilka kroków. W wykonanych wcześniej krokach od 1. do 3. obliczyliśmy prawdopodobieństwa tokenów odpowiadające docelowym tensorom. Następnie, w krokach od 4. do 6., te prawdopodobieństwa są przekształcane za pomocą funkcji logarytmicznej, a potem uśredniane

W wynikach uzyskujemy następujące wartości:

```
tensor([-9.5042, -10.3796, -11.3677, -11.4798, -9.7764, -12.2561])
```

Przetwarzanie logarytmów ocen prawdopodobieństwa w porównaniu z korzystaniem z dokładnych ocen prawdopodobieństwa pozwala na łatwiejszą optymalizację matematyczną. Dokładne omówienie tego tematu wykracza poza zakres tej książki.

Szczegółowo omówiłem tę tematykę w wykładzie, do którego odnośnik można znaleźć w „Dodatku B”.

W kolejnym kroku scalamy logarytmy prawdopodobieństw w pojedynczą ocenę. W tym celu wyliczamy średnią (krok 5. na rysunku 5.7):

```
avg_log_probas = torch.mean(log_probas)  
print(avg_log_probas)
```

Oto wynikowy średni logarytm prawdopodobieństwa:

```
tensor(-10.7940)
```

Celem obliczeń jest uzyskanie średniego prawdopodobieństwa logarytmicznego jak najbliższego zera. Ten cel można osiągnąć dzięki aktualizacji wag modelu, w ramach procesu uczenia. Powszechną praktyką w uczeniu głębokim nie jest jednak dążenie do uzy-skania zerowego średniego prawdopodobieństwa logarytmicznego. Prawdziwym celem jest zwiększenie do zera zanegowanego średniego prawdopodobieństwa logarytmicz-nego. Zanegowane średnie prawdopodobieństwo logarytmiczne to po prostu średnie prawdopodobieństwo logarytmiczne pomnożone przez -1 — obliczenie to pokazałem w kroku 6. na rysunku 5.7:

```
neg_avg_log_probas = avg_log_probas * -1  
print(neg_avg_log_probas)
```

Uruchomienie powyższego kodu spowoduje wyświetlenie wartości `tensor(10.7940)`. W kategoriach uczenia głębokiego wyraz zamieniający tę ujemną wartość, -10.7940 , na 10.7940 określa się jako stratę *entropii krzyżowej*. Do tego celu przydaje się PyTorch — biblioteka, która ma wbudowaną funkcję `cross_entropy`, obejmującą wszystkie sześć kroków przedstawionych na rysunku 5.7.

Strata entropii krzyżowej

Strata entropii krzyżowej jest popularną miarą w uczeniu maszynowym i uczeniu głębokim, mierzącą różnicę między dwoma rozkładami prawdopodobieństwa — zazwyczaj rzeczywistym rozkładem etykiet (tutaj: tokenów w zbiorze danych) i prognozowanym rozkładem uzyskanym z modelu (na przykład prawdopodobieństwami tokenów wygenerowanych przez model LLM).

W kontekście uczenia maszynowego, a w szczególności w takich frameworkach jak PyTorch, funkcja `cross_entropy` oblicza tę miarę dla dyskretnych wyników. Obliczenie to przypomina zanegowane średnie prawdopodobieństwa logarytmiczne docelowych tokenów, z uwzględnieniem prawdopodobieństwa tokenów wygenerowanych przez model. W związku z tym terminy „entropia krzyżowa” i „zanegowane średnie prawdopodobieństwo logarytmiczne” są ze sobą powiązane i w praktyce często używane zamiennie.

Przed zastosowaniem funkcji `cross_entropy` przyjrzyjmy się kształtu logitów i docelowych tensorów:

```
print("Kształt logitów:", logits.shape)  
print("Kształt docelowych tensorów:", targets.shape)
```

Oto wynikowe kształty:

```
Kształt logitów: torch.Size([2, 3, 50257])
Kształt docelowych tensorów: torch.Size([2, 3])
```

Jak można zauważyc, tensor `logits` ma trzy wymiary: rozmiar partii, liczbę tokenów i rozmiar słownictwa. Tensor `targets` ma dwa wymiary: rozmiar partii i liczbę tokenów.

W przypadku funkcji straty `cross_entropy` we frameworku PyTorch chcemy spłaszczyć te tensorы przez ich scalenie zgodnie z wymiarami partii:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Spłaszczone logity:", logits_flat.shape)
print("Spłaszczone tensorы docelowe:", targets_flat.shape)
```

Wynikowe wymiary tensora to:

```
Spłaszczone logity: torch.Size([6, 50257])
Spłaszczone tensorы docelowe: torch.Size([6])
```

Należy pamiętać, że zmienna `targets` zawiera identyfikatory tokenów, które model LLM ma wygenerować, a `logits` zawiera nieskalowane wyniki modelu, przed wprowadzeniem ich do funkcji `softmax` w celu uzyskania ocen prawdopodobieństwa.

Wcześniej zastosowaliśmy funkcję `softmax`, wybraliśmy oceny prawdopodobieństwa odpowiadające identyfikatorom tokenów docelowych i obliczyliśmy zanegowane średnie prawdopodobieństwa logarytmiczne. Wszystkie te kroki wykonuje funkcja `cross_entropy` z biblioteki PyTorch:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

Wynikowa strata jest równa tej, którą uzyskaliśmy wcześniej, gdy pojedynczo uruchamialiśmy kroki pokazane na rysunku 5.7:

```
tensor(10.7940)
```

Perpleksja

Perpleksja jest, obok straty entropii krzyżowej, często stosowaną miarą oceny wydajności modeli w takich zadaniach jak modelowanie języka naturalnego. Zapewnia ona łatwiejszy do interpretacji sposób zilustrowania niepewności modelu w przewidywaniu następnego tokena w sekwencji.

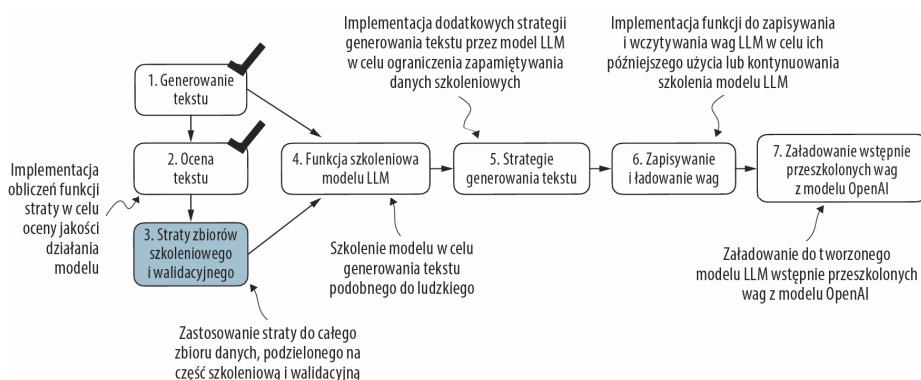
Perpleksja jest miarą dopasowania rozkładu prawdopodobieństwa prognozowanego przez model do rzeczywistego rozkładu słów w zbiorze danych. Podobnie jak w przypadku straty entropii krzyżowej, niższa perpleksja wskazuje, że przewidywania modelu są bliższe rzeczywistemu rozkładowi. Perpleksję można obliczyć za pomocą instrukcji `perplexity = torch.exp(loss)`, która po zastosowaniu do wcześniej obliczonej straty zwraca wartość `tensor(48725.8203)`.

Perpleksję często uważa się za łatwiejszą do interpretacji od nieprzetworzonej wartości straty, ponieważ oznacza ona skuteczny rozmiar słownictwa, co do którego model jest niepewny na każdym z etapów. W podanym przykładzie oznaczałoby to, że model nie jest pewien, który z 48 725 tokenów w słowniku ma wygenerować jako następny token w sekwencji.

W celach ilustracyjnych obliczyliśmy stratę dla dwóch niezbyt obszernych wejść tekstowych. W kolejnym punkcie zastosujemy obliczenia strat do całego zestawu szkoleniowego i walidacyjnego.

5.1.3. Obliczanie strat w zestawie szkoleniowym i walidacyjnym

Najpierw trzeba przygotować zbiory danych szkoleniowy i walidacyjny, których użyjemy do szkolenia modelu LLM. Następnie, jak pokazałem na rysunku 5.8, obliczymy entropię krzyżową dla tych zbiorów, co jest ważnym elementem procesu uczenia modelu.



Rysunek 5.8. Po wykonaniu kroków 1. i 2., w tym obliczeniu straty entropii krzyżowej, można zastosować obliczenia tej straty do całego zbioru danych tekstowych, który będzie wykorzystany do szkolenia modelu

Aby obliczyć stratę dla szkoleniowych i walidacyjnych zbiorów danych, użyjemy bardzo małego tekstuowego zbioru danych, opowiadania *The Verdict* Edith Wharton, z którym pracowaliśmy już w rozdziale 2. Dzięki wybraniu tekstu z domeny publicznej nie musimy się obawiać jakiegokolwiek naruszenia praw autorskich. Dodatkowo użycie tak małego zbioru danych pozwoli na uruchomienie przykładów kodu na standardowym laptopie w ciągu kilku minut, nawet bez wysokiej klasy układu GPU, co w procesie nauki jest szczególnie korzystne.

UWAGA Zainteresowani Czytelnicy mogą przygotować większy zbiór danych, składający się z ponad 60 tysięcy książek z domeny publicznej z Project Gutenberg, i za pomocą kodu uzupełniającego do tej książki przeprowadzić na nim szkolenie modelu LLM (szczególnie w „Dodatku D”).

Tekst opowiadania *The Verdict* jest ładowany przez poniższy kod:

```

file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()
  
```

Koszty wstępnego szkolenia modelu LLM

Aby spojrzeć na skalę projektu uczenia modelu LLM z odpowiedniej perspektywy, rozważmy szkolenie modelu Llama 2 o 7 miliardach parametrów, stosunkowo popularnego ogólnodostępnego modelu LLM. Ten model wymagał 184 320 godzin pracy drogich procesorów graficznych A100 i przetworzenia 2 bilionów tokenów. W chwili gdy pisałem te słowa, uruchomienie w chmurze AWS serwera 8xA100 kosztowało około 30 dolarów za godzinę. Według przybliżonych oszacowań całkowity koszt szkolenia takiego modelu LLM wynosi około 690 tysięcy dolarów (obliczony jako 184 320 godzin podzielone przez 8, a następnie pomnożone przez 30 dolarów).

Po załadowaniu zbioru danych można sprawdzić liczbę znaków i tokenów w zbiorze danych:

```
total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Liczba znaków:", total_characters)
print("Liczba tokenów:", total_tokens)
```

Oto uzyskany wynik:

```
Liczba znaków: 20479
Liczba tokenów: 5145
```

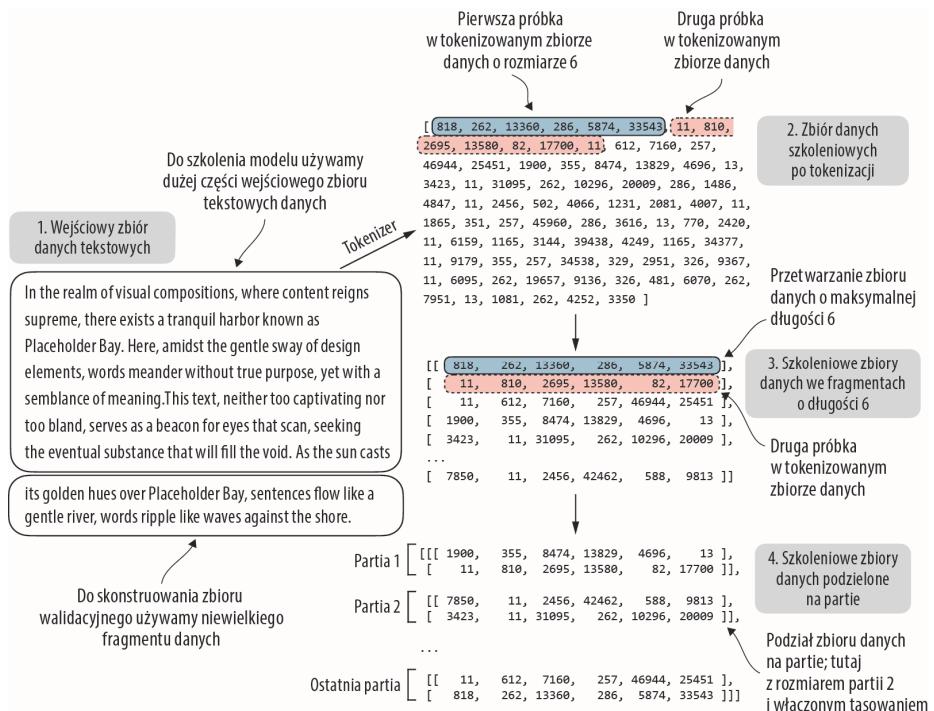
Tekst zawierający zaledwie 5145 tokenów może wydawać się zbyt mały do przeszko- lenia modelu LLM, ale jak wspomniałem, tworzony model jest przeznaczony do celów edukacyjnych, a dzięki temu, że nie jest zbyt obszerny, kod można uruchomić w ciągu kilku minut zamiast wielu tygodni. Dodatkowo później do kodu klasy GPTModel załadujemy wagi OpenAI ze wstępnego szkolenia.

W kolejnym kroku podzielimy zbiór danych na szkoleniowy i walidacyjny, a w celu przygotowania partii do szkolenia modelu LLM użyjemy znanych Ci już z rozdziału 2. mechanizmów ładujących dane. Ten proces przedstawiłem na rysunku 5.9. Ze względu na ograniczenia przestrzenne użyłem ustawienia `max_length=6`. Jednak na potrzeby rzeczywistych programów ładujących dane można ustawić wartość `max_length` równą obsługiwanej przez model LLM 256-tokenowej długości kontekstu. Dzięki temu model LLM będzie widział podczas szkolenia dłuższe teksty.

UWAGA Dla uproszczenia i w celu poprawy wydajności szkolimy model z użyciem danych szkoleniowych przekazywanych we fragmentach o podobnej wielkości. W praktyce model LLM można jednak szkolić z danymi wejściowymi o zmiennej długości. To pozwala modelowi LLM lepiej uogólnić różne rodzaje używanych danych wejściowych.

Aby zaimplementować podział danych i ich ładowanie, najpierw definiujemy wartość `train_ratio` tak, aby wykorzystać 90% danych do szkolenia, a pozostałe 10% – jako dane walidacyjne do oceny modelu podczas szkolenia:

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```



Rysunek 5.9. Podczas przygotowywania mechanizmów ładujących dane tekst wejściowy dzieli się na zbiór szkoleniowy i walidacyjny. Następnie tekst jest poddawany tokenizacji (dla uproszczenia na rysunku pokazano tokenizację tylko dla części zbioru szkoleniowego), a następnie dzielony na fragmenty o długości określonej przez użytkownika (tutaj 6). Na koniec są tasowane wiersze, a tekst jest dzielony na partie (tutaj rozmiar parti wynosi 2), które można wykorzystać do uczenia modelu

Korzystając z podzbiorów `train_data` i `val_data` oraz kodu `create_dataloader_v1` z rozdziału 2., możemy utworzyć odpowiedni mechanizm ładujący dane:

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
```

```
drop_last=False,
shuffle=False,
num_workers=0 )
```

Aby zmniejszyć zapotrzebowanie na zasoby obliczeniowe, użyliśmy stosunkowo małego rozmiaru partii, ponieważ pracujemy z bardzo małym zbiorem danych. W praktyce nie jest rzadkością szkolenie modeli LLM z partiami o wielkości 1024 lub większymi.

W ramach opcjonalnego sprawdzenia możemy iterować po mechanizmach ładujących dane, aby zweryfikować, czy zostały poprawnie utworzone:

```
print("Szkoleniowy mechanizm ładujący:")
for x, y in train_loader:
    print(x.shape, y.shape)

print("\nWalidacyjny mechanizm ładujący:")
for x, y in val_loader:
    print(x.shape, y.shape)
```

Powinieneś zobaczyć następujące wyniki:

```
Szkoleniowy mechanizm ładujący:
torch.Size([2, 256]) torch.Size([2, 256]) torch.Size([2, 256])
Walidacyjny mechanizm ładujący:
torch.Size([2, 256]) torch.Size([2, 256])
```

Na podstawie wyników działania poprzedniego kodu można stwierdzić, że w zbiorze szkoleniowym jest dziewięć partii, a każda z nich obejmuje dwie próbki po 256 tokenów. Ponieważ przydzieliśmy zaledwie 10% danych do walidacji, w zbiorze walidacyjnym jest tylko jedna partia składająca się z dwóch próbek wejściowych. Zgodnie z oczekiwaniami dane wejściowe (x) i dane docelowe (y) mają ten sam kształt (rozmiar partii razy liczba tokenów w każdej partii), ponieważ – jak pisałem w rozdziale 2. – dane docelowe są tworzone przez przesunięcie danych wejściowych o jedną pozycję.

W kolejnym kroku zaimplementujemy funkcję narzędziową służącą do obliczenia straty entropii krzyżowej określonej partii zwróconej przez szkoleniowy i walidacyjny mechanizm ładujący dane:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)           | Transfer do danego urządzenia
    target_batch = target_batch.to(device)         | pozwala przesyłać dane do układu GPU
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten())
)
return loss
```

Możemy teraz użyć funkcji `calc_loss_batch`, obliczającej stratę dla pojedynczej partii, aby zaimplementować funkcję `calc_loss_loader`, która oblicza stratę dla wszystkich partii próbkiowych przez określony mechanizm ładujący dane (listing 5.2).

Listing 5.2. Funkcja obliczająca stratę szkoleniową i walidacyjną

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader) ← W razie nieokreślenia stałej liczby partii
    else: ← należy wykonać iteracje po wszystkich partiach
        num_batches = min(num_batches, len(data_loader)) ← Jeżeli num_batches przekracza liczbę parti w mechanizmie ładującym dane, zmniejsza liczbę parti tak, aby dopasować je do całkowitej liczby parti w mechanizmie ładującym dane
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item() ← Suma strat dla każdej partii
        else:
            break
    return total_loss / num_batches ← Średnia strata dla wszystkich parti
```

Domyślnie funkcja `calc_loss_loader` iteruje po wszystkich partiach obsługiwanych przez określony mechanizm ładujący dane, akumuluje straty w zmiennej `total_loss`, a następnie oblicza i uśrednia straty dla całkowitej liczby parti. Aby przypieszyć ocenę podczas szkolenia modelu, można także określić mniejszą liczbę parti za pomocą ustawienia `num_batches`.

Zobaczmy teraz, jak funkcja `calc_loss_loader` działa w praktyce. W tym celu zastosujemy ją do zbiorów szkoleniowego i walidacyjnego:

Jeśli masz maszynę z układem GPU obsługującym CUDA, model LLM będzie szkolony na układzie GPU bez wprowadzania jakichkolwiek zmian w kodzie

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu") ← Wyłączenie śledzenia gradientu w celu poprawy wydajności, ponieważ jeszcze nie przeprowadzamy szkolenia
model.to(device)
with torch.no_grad(): ←
    train_loss = calc_loss_loader(train_loader, model, device) ←
    val_loss = calc_loss_loader(val_loader, model, device)
print("Strata zbioru szkoleniowego:", train_loss)
print("Strata zbioru walidacyjnego:", val_loss)
```

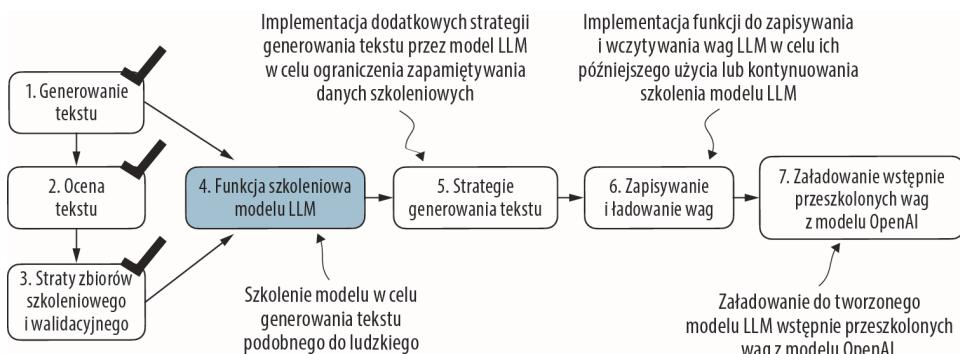
Za pomocą ustawienia „`device`” zapewniamy ładowanie danych do tego samego urządzenia, którego używał model LLM

Oto wynikowe wartości strat:

Strata zbioru szkoleniowego: 10.98758347829183
 Strata zbioru walidacyjnego: 10.98110580444336

Wartości strat są stosunkowo wysokie, ponieważ model nie został jeszcze przeszkołony. Dla porównania, jeśli model uczy się generować kolejne tokeny, kiedy pojawiają się w zbiorach szkoleniowych i walidacyjnych, strata zbliża się do zera.

Po opracowaniu sposobu mierzenia jakości wygenerowanego tekstu możemy przystąpić do szkolenia modelu LLM, mającego na celu zmniejszanie straty. W ten sposób model będzie stawał się coraz lepszy w generowaniu tekstu (rysunek 5.10).



Rysunek 5.10. Podsumowaliśmy proces generowania tekstu (krok 1.) i zaimplementowaliśmy podstawowe techniki oceny modelu (krok 2.) w celu obliczenia strat w zbiorze szkoleniowym i walidacyjnym (krok 3.). Następnie przejdziemy do funkcji szkoleniowych i wstępnie przeszkołimy model LLM (krok 4.).

W kolejnym kroku skupimy się na wstępny szkoleniu modelu LLM. Po przeszkołeniu modelu zaimplementujemy alternatywne strategie generowania tekstu oraz zapiszemy model i załadjemy wstępnie przeszkołone wagi.

5.2. Szkolenie modelu LLM

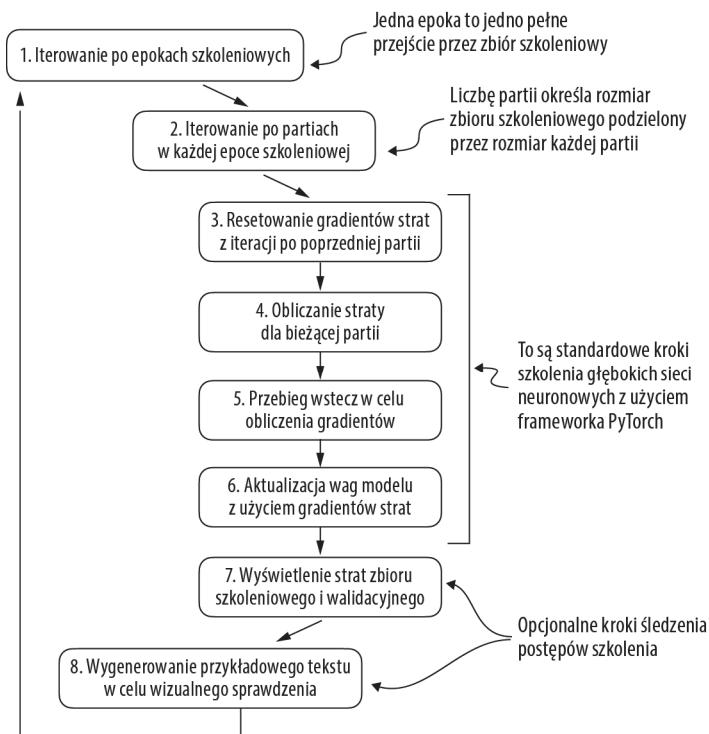
Nadszedł czas na implementację kodu do wstępnego szkolenia modelu LLM, czyli klasy GPTModel. W tym celu, aby kod był zwięzły i czytelny, skupię się na prostej pętli szkoleniowej.

UWAGA W „Dodatku D” opisałem bardziej zaawansowane techniki, między innymi *rozgrzewkę współczynnika uczenia* (ang. *learning rate warmup*), *kosinusowe wyżarzanie* (ang. *cosine annealing*) i *przycinanie gradientu* (ang. *gradient clipping*).

Schemat blokowy na rysunku 5.11 przedstawia typowy przepływ pracy szkolenia sieci neuronowej z użyciem frameworka PyTorch wykorzystywany do szkolenia modeli LLM. Przepływ pracy obejmuje osiem kroków — zaczyna się od iterowania po wszystkich epokach, przetwarzania partii, resetowania gradientów, obliczania strat i nowych gradientów oraz aktualizowania wag, a kończy na etapach związanych z monitorowaniem, takich jak wyświetlanie strat i generowanie próbek tekstu.

UWAGA Czytelników poczynających w szkoleniu głębszych sieci neuronowych z użyciem frameworka PyTorch, którym część kroków przedstawionych na rysunku 5.11 nie jest znana, zachęcam do przeczytania podrozdziałów od A.5 do A.8 w „Dodatku A”.

Zaprezentowany przepływ szkolenia można zaimplementować w kodzie za pomocą funkcji `train_model_simple` (listing 5.3).



Rysunek 5.11. Typowa pętla szkolenia głębokich sieci neuronowych z użyciem frameworka PyTorch składa się z wielu kroków, obejmujących iterowanie po partiach w zbiorze szkoleniowym przez kilka epok. W każdej iteracji pętli obliczana jest strata dla każdej partii zbioru szkoleniowego, co pozwala określić gradienty strat. Stosuje się je do aktualizacji wag modelu, tak aby zminimalizować stratę zbioru szkoleniowego

Listing 5.3. Główna funkcja implementująca wstępne szkolenie modeli LLM

```

def train_model_simple(model, train_loader, val_loader,
                      optimizer, device, num_epochs,
                      eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1
    for epoch in range(num_epochs):  ← Uruchomienie głównej pętli szkoleniowej
        model.train()  ← Inicjalizacja list do śledzenia strat i obserwowanych tokenów
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()  ← Zresetowanie gradientów strat z iteracji dla poprzedniej partii
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()  ← Obliczenie gradientów strat
            optimizer.step()  ← Aktualizacja wag modelu z użyciem gradientów strat
            tokens_seen += input_batch.numel()
            global_step += 1
  
```

```

if global_step % eval_freq == 0:           ← Opcjonalny
    train_loss, val_loss = evaluate_model(   krok oceny
        model, train_loader, val_loader, device, eval_iter)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    track_tokens_seen.append(tokens_seen)
    print(f"Epoka {epoch+1} (krok {global_step:06d}): "
          f"Strata zbioru szkoleniowego {train_loss:.3f}, "
          f"Strata zbioru walidacyjnego {val_loss:.3f}"
    )
)                                     ← Wyświetlenie próbki tekstu
generate_and_print_sample(            po każdej epoce
    model, tokenizer, device, start_context
)
return train_losses, val_losses, track_tokens_seen

```

Zauważ, że funkcja `train_model_simple`, którą właśnie stworzyliśmy, korzysta z dwóch funkcji, które jeszcze nie zostały zdefiniowane: `evaluate_model` i `generate_and_print_sample`.

Funkcja `evaluate_model` odpowiada kroku 7. z rysunku 5.11. Wyświetla straty zbioru szkoleniowego i walidacyjnego po każdej aktualizacji modelu, dzięki czemu można ocenić, czy szkolenie wpływa na poprawę modelu. Dokładniej mówiąc, funkcja `evaluate_model` oblicza stratę na zbiorach szkoleniowym i walidacyjnym. Jednocześnie podczas obliczania straty na zbiorach szkoleniowym i walidacyjnym dba o to, aby model był w trybie oceny, w którym śledzenie gradientu i dropout są wyłączone:

```

def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    → model.eval()           ← Wyłączenie śledzenia gradientu, które podczas oceny nie jest wymagane, w celu zmniejszenia narżenia obliczeniowego
    with torch.no_grad():   ← Podczas oceny dropout, w celu uzyskania stabilnych, powtarzalnych wyników, jest wyłączony.
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss

```

Podobnie jak `evaluate_model`, `generate_and_print_sample` jest funkcją narzędziową wykorzystywaną do śledzenia, czy model poprawia się podczas szkolenia. W szczególności funkcja `generate_and_print_sample` pobiera jako dane wejściowe fragment tekstu (`start_context`), dokonuje jej konwersji na identyfikatory tokenów i przekazuje do modelu LLM w celu wygenerowania próbki tekstu za pomocą wykorzystanej wcześniej funkcji `generate_text_simple`:

```

def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,

```

```

        max_new_tokens=50, context_size=context_size
    )
decoded_text = token_ids_to_text(token_ids, tokenizer)
print(decoded_text.replace("\n", " "))
model.train()

```

| Zwięzły format wyświetlania

O ile funkcja `evaluate_model` zwraca liczbową estymatę postępu szkolenia modelu, o tyle funkcja `generate_and_print_sample` dostarcza wygenerowanego przez model konkretnego przykładu tekstowego, pozwalającego ocenić możliwości modelu podczas szkolenia.

AdamW

Podczas szkolenia głębokich sieci neuronowych często wykorzystuje się optymalizatory Adam. W pętli szkoleniowej zastosowanej w przykładzie wybrałem jednak optymalizator `AdamW`. `AdamW` to wariant optymalizatora Adam, w którym udoskonalono metodę ograniczania wartości wag (ang. *weight decay*). Dzięki temu model staje się mniej złożony, a ograniczenie wysokich wartości wag pomaga zapobiegać nadmiernemu dopasowaniu. Dzięki temu dostrojeniu optymalizator `AdamW` osiąga bardziej skuteczną regularyzację i lepsze uogólnienie, co czyni go dobrym wyborem do szkolenia modeli LLM.

Zobaczmy, jak przedstawione elementy zachowują się w praktyce. W tym celu przeprowadzimy szkolenie egzemplarza modelu `GPTModel` przez 10 epok z użyciem optymalizatora `AdamW` i zdefiniowanej wcześniej funkcji `train_model_simple`:

```

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),           ← Metoda .parameters() zwraca wszystkie
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)

```

Uruchomienie funkcji `train_model_simple` rozpoczyna proces szkolenia. Jego ukończenie na MacBooku Air lub podobnym laptopie zajmuje około 5 minut. Oto wynik wyświetlanego podczas tego uruchomienia:

```

Epoka 1 (krok 000000): Strata zbioru szkoleniowego 9.818, Strata zbioru walidacyjnego
→9.928
Epoka 1 (krok 000005): Strata zbioru szkoleniowego 8.065, Strata zbioru walidacyjnego
→8.335
Every effort moves you,,,,,,,,,,,
Epoka 2 (krok 000010): Strata zbioru szkoleniowego 6.622, Strata zbioru walidacyjnego
→7.051
Epoka 2 (krok 000015): Strata zbioru szkoleniowego 6.047, Strata zbioru walidacyjnego
→6.600

```

Every effort moves you, and,, and,,, and,,, and,..
Epoka 3 (krok 000020): Strata zbioru szkoleniowego 5.586, Strata zbioru walidacyjnego
→6.477
Epoka 3 (krok 000025): Strata zbioru szkoleniowego 5.523, Strata zbioru walidacyjnego
→6.399
Every effort moves you, and,
→and, and, and, and
[...] ← Dla oszczędności miejsca usunięto wyniki pośrednie
Epoka 9 (krok 000075): Strata zbioru szkoleniowego 1.499, Strata zbioru walidacyjnego
→6.230
Epoka 9 (krok 000080): Strata zbioru szkoleniowego 1.174, Strata zbioru walidacyjnego
→6.250
Every effort moves you know," was not that my hostess was "interesting": on that point
→I could have given Miss Croft the fact, with a Mrs. Gisburn's open countenance.
→"It's his pictures with a "strongest," she was
Epoka 10 (krok 000085): Strata zbioru szkoleniowego 0.901, Strata zbioru walidacyjnego
→6.328
Every effort moves you?" "Yes--quite insensible to the irony. She wanted him vindica
→ted--and by me!" He placed them at my elbow and as I turned, and down the
→room, when I

Jak można zauważyć, strata zbioru szkoleniowego znacząco się poprawia. Początkowo ma wartość 9.818, by ostatecznie osiągnąć 0.901. Umiejętności językowe modelu wyraźnie się poprawiły. Na początku model był w stanie dodawać przecinki do początkowego kontekstu (Every effort moves you,,,,,,,) lub powtarzać słowo *and*. Po zakończeniu szkolenia potrafi wygenerować znacznie bardziej rozbudowany tekst.

Podobnie jak w przypadku straty dla zbioru szkoleniowego, strata zbioru walidacyjnego zaczyna się od wysokiej wartości (9.928) i podczas szkolenia maleje. Nigdy jednak nie staje się tak mała jak strata zestawu szkoleniowego i po 10. epoce pozostaje na poziomie 6.328.

Dokładniejszym omówieniem straty zbioru walidacyjnego zajmę się w dalszej części tego rozdziału. Wcześniej jednak utworzę prosty wykres, który pokazuje straty zbioru szkoleniowego i walidacyjnego obok siebie:

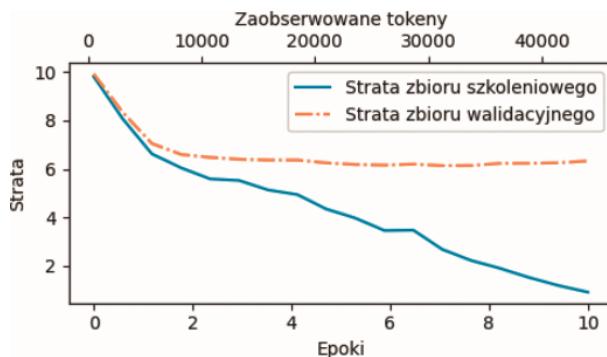
```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Strata zbioru szkoleniowego")
    ax1.plot(
        epochs_seen, val_losses, linestyle="--", label="Strata zbioru walidacyjnego"
    )
    ax1.set_xlabel("Epoki")
    ax1.set_ylabel("Strata")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax2 = ax1.twiny()
    ax2.plot(tokens_seen, train_losses, alpha=0)
    ax2.set_xlabel("Zaobserwowane tokeny")
    fig.tight_layout()
    plt.show()
```

Utworzenie drugiej osi x,
która współdzieli tę samą osią y

Niewidoczny wykres w celu
wyrownywania punktów

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

Wynikowy wykres strat zbiorów szkoleniowych i walidacyjnych pokazałem na rysunku 5.12. Jak można zauważyć, zarówno strata szkoleniowa, jak i walidacyjna zaczynają się poprawiać w pierwszej epoce. Jednak po drugiej epoce wielkości strat zaczynają się rozchodzić. Ta rozbieżność i fakt, że strata walidacyjna jest znacznie większa niż strata szkoleniowa, wskazują, że model jest nadmiernie dopasowany do danych szkoleniowych. Możemy potwierdzić, że model zapamiętuje dane szkoleniowe dosłownie. Wyszukuje w pliku tekstowym opowiadania *The Verdict* wygenerowane fragmenty tekstu, na przykład „quite insensible to the irony”³.



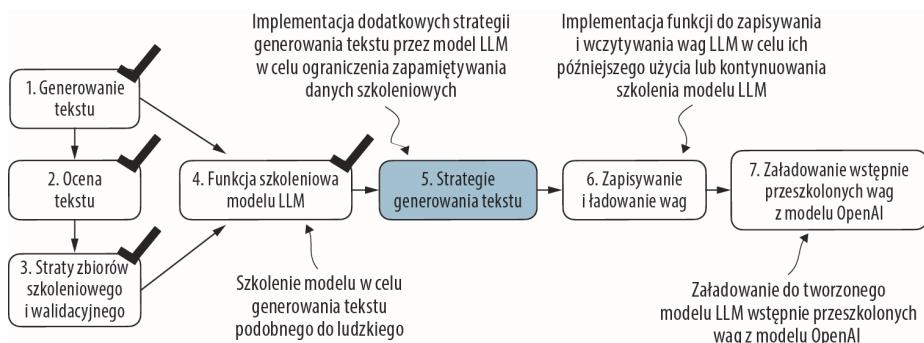
Rysunek 5.12. Na początku szkolenia zarówno straty w zbiorze szkoleniowym, jak i walidacyjnym gwałtownie spadają, co jest oznaką tego, że model się uczy. Jednak strata zbioru szkoleniowego po drugiej epoce nadal spada, podczas gdy strata zbioru walidacyjnego pozostaje na tym samym poziomie. Jest to znak, że model nadal się uczy, ale po epoce 2. nadmiernie dopasowuje się do zestawu szkoleniowego

Dosłowne zapamiętywanie fragmentów zbioru szkoleniowego jest oczekiwane, ponieważ pracujemy z bardzo małym zbiorem danych szkoleniowych i szkolimy model przez wiele epok. Zazwyczaj modele szkoli się na znacznie większym zbiorze danych i tylko przez jedną epokę.

UWAGA Jak wspomniałem, zainteresowani Czytelnicy mogą spróbować przeszkoić model na korpusie składającym się z ponad 60 tysięcy książek z domeny publicznej dostępnych w ramach projektu Gutenberg. Przy takim szkoleniu problem nadmiernego dopasowania nie występuje. Szczegółowe informacje znajdziesz w „Dodatku B”.

Jak widać na rysunku 5.13, osiągnęliśmy cztery cele, które wyznaczyłem dla tego rozdziału. W kolejnym podrozdziale omówię strategie generowania tekstu przez modele LLM w celu złagodzenia efektu zapamiętywania danych szkoleniowych i zwiększenia oryginalności tekstu generowanego przez LLM. Następnie przejdę do opisania

³ *quite insensible to the irony* – całkiem niewrażliwy na ironię – przyp. tłum.



Rysunek 5.13. Po zastosowaniu funkcji szkoleniowej model potrafi wygenerować spójny tekst. Często jednak dosłownie zapamiętuje fragmenty ze zbioru szkoleniowego. W dalszej części książki omówię strategie generowania bardziej zróżnicowanych tekstów wynikowych

problemu ładowania wag oraz zapisywania i ładowania wstępnie przeszkolonych wag z modelu GPT OpenAI.

5.3. Strategie dekodowania w celu zarządzania losowością

Przyjrzyjmy się strategiom generowania tekstu (zwanych również strategiami dekodowania) pozwalającym wygenerować bardziej oryginalny tekst. Najpierw krótko wróćmy do funkcji `generate_text_simple`, której używaliśmy wcześniej w funkcji `generate_and_print_sample`. Następnie omówię dwie techniki: *skalowanie temperatury* i *próbkowanie top-k*, mające na celu ulepszenie tej funkcji.

Zacznę od przeniesienia modelu z układu GPU do CPU, ponieważ wnioskowanie z użyciem względnie małego modelu nie wymaga układu GPU. Ponadto po szkoleniu, aby wyłączyć losowe komponenty, takie jak dropout, przełączę model w tryb oceny:

```
model.to("cpu")
model.eval()
```

Następnie w funkcji `generate_text_simple`, która używa modelu LLM do generowania jednego tokena na raz, zastosuję egzemplarz klasy `GPTModel` (zapisany w zmiennej `model`):

```
tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Wynikowy tekst:\n", token_ids_to_text(token_ids, tokenizer))
```

Wygenerowany tekst to:

Wynikowy tekst:
Every effort moves you?"

"Yes--quite insensible to the irony. She wanted him vindicated--and by me!"

Jak wyjaśniałem wcześniej, prognozowany token jest wybierany na każdym etapie generowania tekstu, na podstawie największego prawdopodobieństwa wśród wszystkich tokenów w słowniku. Oznacza to, że nawet jeśli wielokrotnie uruchomimy funkcję `generate_text_simple` z tym samym kontekstem początkowym (Every effort moves you), model LLM zawsze wygeneruje ten samy wynik.

5.3.1. Skalowanie temperaturą

Przyjrzyjmy się teraz skalowaniu temperaturą, technice, która do zadania generowania kolejnych tokenów wprowadza probabilistyczny proces wyboru. Wcześniej, wewnątrz funkcji `generate_text_simple`, zawsze jako następny token próbkovaliśmy token o najwyższym prawdopodobieństwie. Do tego celu wykorzystywaliśmy wywołanie `torch.argmax`. Taką technikę często nazywa się *zachłannym dekodowaniem* (ang. *greedy decoding*). Aby generować teksty o większej różnorodności, możemy zastąpić `argmax` funkcją próbującą z rozkładu prawdopodobieństwa (tutaj: ocen prawdopodobieństwa generowanych przez model LLM dla każdej pozycji w słowniku, na każdym etapie generowania tokena).

Aby zilustrować probabilistyczne próbkoowanie na konkretnym przykładzie, przyjrzyjmy się pokrótko procesowi generowania kolejnych tokenów. Dla celów ilustracyjnych skorzystamy ze słownika o bardzo małej objętości:

```
vocab = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}
inverse_vocab = {v: k for k, v in vocab.items()}
```

Następnie założymy, że model LLM otrzymuje kontekst początkowy „every effort moves you” i generuje takie oto logity następnego tokena:

```
next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)
```

Jak pisałem w rozdziale 4., wewnątrz funkcji `generate_text_simple` konwertujemy logity na prawdopodobieństwa za pomocą funkcji `softmax`. Następnie za pomocą funkcji `argmax` uzyskujemy identyfikator tokena odpowiadający wygenerowanemu tokenowi.

Ten token można następnie, za pomocą odwrotnego słownika, odwzorować z powrotem na tekst:

```
probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])
```

Ponieważ największa wartość logitu i, odpowiednio, największa ocena prawdopodobieństwa softmax odpowiadają czwartej pozycji (pozycja indeksu 3, ponieważ indeksy tablic w Pythonie zaczynają się od 0), model generuje słowo „forward”.

Aby zaimplementować probabilistyczny proces próbkowania, można zastąpić argmax dostępną we frameworku PyTorch funkcją `multinomial`:

```
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])
```

Model tak jak poprzednio wyświetla słowo „forward”. Co się stało? Funkcja `multinomial` próbuje następny token proporcjonalnie do jego oceny prawdopodobieństwa. Mówiąc inaczej, słowo „forward” nadal jest najbardziej prawdopodobnym tokenem i będzie wybierane przez funkcję `multinomial` przez większość czasu, ale nie przez cały czas. Aby to zilustrować, zaimplementujmy funkcję, która powtarza próbkowanie 1000 razy:

```
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item()
              for i in range(1_000)]
    sampled_ids = torch.bincount(torch.tensor(sample))
    for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[i]}")
```

```
print_sampled_tokens(probas)
```

Oto wynik próbkowania:

```
71 x closer
2 x every
0 x effort
544 x forward
2 x inches
1 x moves
0 x pizza
376 x toward
4 x you
```

Jak można zauważyć, słowo „forward” jest wybierane przez większość czasu (544 na 1000 razy), ale niekiedy są wybierane również inne tokeny, takie jak „closer”, „inches” i „toward”. Oznacza to, że gdybyśmy wewnątrz funkcji `generate_and_print_sample` zastąpili funkcję `argmax` funkcją `multinomial`, model LLM czasami generowałby takie

teksty jak „every effort moves you toward”⁴, „every effort moves you inches”⁵ i „every effort moves you closer”⁶ zamiast generowania za każdym razem „every effort moves you forward”⁷.

Proces dystrybucji i selekcji można dodatkowo kontrolować za pomocą mechanizmu zwanego *skalowaniem temperaturą*. Skalowanie temperaturą to po prostu specjalistyczny termin opisujący dzielenie logitów przez liczbę większą od 0:

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

Stosowanie temperatur większych niż 1 skutkuje bardziej równomiernie rozłożonymi prawdopodobieństwami tokenów, a temperatur mniejszych niż 1 – rozkładami bardziej stromymi (ostrzejszymi, z wyraźnymi szczytami). Zilustrujmy to na przykładzie wykresu oryginalnych prawdopodobieństw razem z prawdopodobieństwami skalowanymi z użyciem różnych wartości temperatur:

```
temperatures = [1, 0.1, 5]
scaled_probas = [softmax_with_temperature(next_token_logits, T)
                 for T in temperatures]
x = torch.arange(len(vocab))
bar_width = 0.15
fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i],
                   bar_width, label=f'Temperatura = {T}')
ax.set_ylabel('Prawdopodobieństwo')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()
plt.tight_layout()
plt.show()
```

Pierwotny, niższy i wyższy poziom ufności

Wynikowy wykres pokazałem na rysunku 5.14.

Użycie temperatury o wartości 1 powoduje podzielenie logitów przez 1 przed przekazaniem ich do funkcji softmax w celu obliczenia ocen prawdopodobieństwa. Mówiąc inaczej, używanie temperatury równej 1 jest równoważne z nieużywaniem żadnego skalowania temperaturą. W tym przypadku tokeny są wybierane z prawdopodobieństwem równym oryginalnym ocenom prawdopodobieństwa softmax uzyskanym za pomocą funkcji próbkowania `multinomial` z biblioteki PyTorch. Na przykład dla ustawienia temperatury 1 token odpowiadający słowu „forward” byłby wybierany przez mniej więcej 60% czasu (rysunek 5.14).

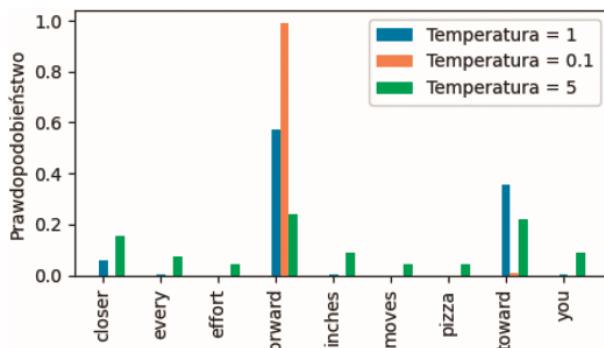
Jak widać na rysunku 5.14, zastosowanie bardzo małych temperatur, na przykład 0.1, skutkuje ostrzejszymi rozkładami. Przy takim ustawieniu funkcja `multinomial`

⁴ każdy wysiłek przesuwa cię w kierunku – przyp. tłum.

⁵ każdy wysiłek przesuwa cię o kilka cali – przyp. tłum.

⁶ każdy wysiłek przesuwa cię bliżej – przyp. tłum.

⁷ każdy wysiłek przesuwa cię do przodu – przyp. tłum.



Rysunek 5.14. Temperatura o wartości 1 odpowiada nieskalowanym ocenom prawdopodobieństwa dla wszystkich tokenów w słowniku. Zmniejszenie temperatury do 0,1 wywostra rozkład, więc najbardziej prawdopodobny token (tutaj „forward”) będzie miał jeszcze wyższą ocenę prawdopodobieństwa. Z kolei zwiększenie temperatury do 5 sprawi, że rozkład stanie się bardziej równomierny

wybiera najbardziej prawdopodobny token (tutaj „forward”) prawie w 100% przypadków, co zbliża ją do zachowania funkcji argmax. Z kolei temperatura 5 skutkuje bardziej równomiernym rozkładem, przy którym częściej są wybierane inne tokeny. Pozwala to bardziej urozmaicić generowane teksty, ale również skutkuje częstszym generowaniem bezsensownego tekstu. Na przykład użycie temperatury 5 skutkuje generowaniem w mniej więcej 4% prób takich tekstów jak „every effort moves you pizza”⁸.

Ćwiczenie 5.1

Użyj funkcji `print_sampled_tokens`, aby wyświetlić częstości próbkowania prawdopodobieństw softmax skalowanych z użyciem temperatur pokazanych na rysunku 5.14. Jak często w każdym przypadku jest próbkowane słowo „pizza”? Czy potrafisz znaleźć szybszy i dokładniejszy sposób na określenie częstości próbkowania słowa „pizza”?

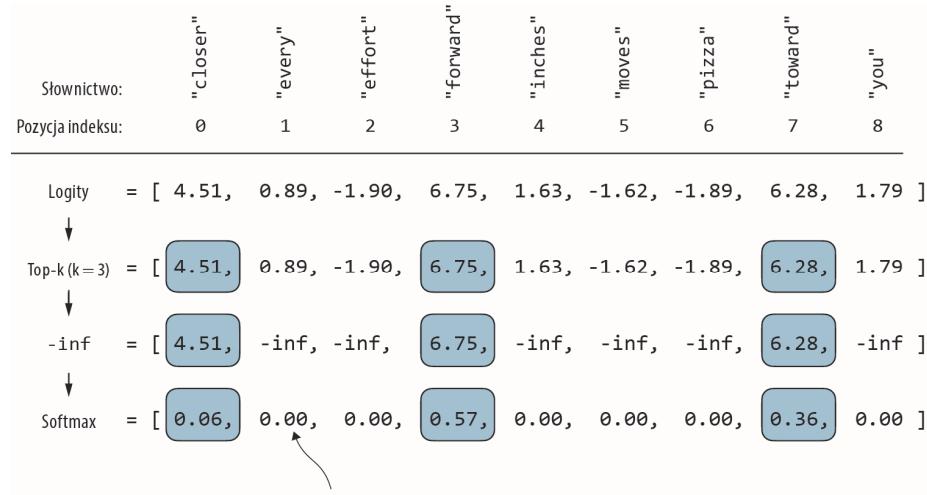
5.3.2. Próbkowanie top- k

W celu zwiększenia różnorodności wyników zaimplementowaliśmy probabilistyczne podejście do próbkowania w połączeniu ze skalowaniem temperaturą. Jak możesz zauważyć, większe wartości temperatury skutkują bardziej równomiernie rozłożonymi prawdopodobieństwami następnego tokena, co powoduje generowanie bardziej zróżnicowanych wyników. Im wyższa temperatura, tym mniejsze prawdopodobieństwo, że model wielokrotnie wybierze najbardziej prawdopodobny token. Pokazana metoda pozwala na odkrywanie mniej prawdopodobnych, ale potencjalnie bardziej interesujących i kreatywnych ścieżek w procesie generowania tekstu. Wadą tego podejścia

⁸ każdy wysiłek porusza cię pizzo – przyp. tłum.

jest jednak uzyskiwanie od czasu do czasu niepoprawnych gramatycznie lub całkowicie bezsensownych wyników, takich jak „every effort moves you pizza”.

Wyniki generowania tekstu można poprawić dzięki zastosowaniu *próbkowania top-k* w połączeniu z próbkowaniem probabilistycznym i skalowaniem temperaturą. Zastosowanie próbkowania top-k dzięki zamaskowaniu niektórych ocen prawdopodobieństwa pozwala ograniczyć próbkowane tokeny do najbardziej prawdopodobnych oraz wykluczyć z procesu selekcji wszystkie inne tokeny (rysunek 5.15).



Dzięki przypisaniu zerowego prawdopodobieństwa do pozycji innych niż top-k zyskujemy pewność, że następny token zawsze będzie próbowany z pozycji top-k

Rysunek 5.15. Dzięki zastosowaniu próbkowania top-k z k = 3 skupiamy się na trzech tokenach powiązanych z logitami o najwyższej wartości, a przed zastosowaniem funkcji softmax maskujemy wszystkie inne tokeny wartością minus nieskończoność (-inf). Skutkuje to rozkładem prawdopodobieństwa z wartością prawdopodobieństwa 0 przypisaną do wszystkich tokenów spoza zbioru top-k (w celu poprawy czytelności liczby na tym rysunku przyjęto do dwóch cyfr po przecinku; wartości wierszu „Softmax” powinny sumować się do jedności)

Po zastosowaniu próbkowania top-k wszystkie niewybrane logity są zastępowane wartością minus nieskończoność (-inf). Dzięki temu podczas obliczania wartości funkcji softmax oceny prawdopodobieństwa tokenów spoza zbioru top-k wynoszą 0, a pozostałe prawdopodobieństwa sumują się do 1 (uważni Czytelnicy pewnie pamiętają sztuczkę z maskowaniem z podrozdziału 3.5.1 w rozdziale 3., w którym opisywałem moduł przyczynowej uwagi).

Procedurę top-k z rysunku 5.15 można zaimplementować za pomocą poniższego kodu. Zaczynamy od wyboru tokenów o najwyższych wartościach logitów:

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
print("Najwyższe wartości logitów:", top_logits)
print("Pozycje najwyższych wartości:", top_pos)
```

Wartości logitów i identyfikatory tokenów trzech najlepszych tokenów w kolejności malejącej to:

```
Najwyższe wartości logitów: tensor([6.7500, 6.2800, 4.5100])
Pozycje najwyższych wartości: tensor([3, 7, 0])
```

Następnie, aby ustawić wartości logitów tokenów, które są poniżej najniższej wartości logitu w wybranej pierwszej trójce, na minus nieskończoność (-inf), stosujemy funkcję `where` z biblioteki PyTorch:

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1], ← Identyfikacja logitów o wartości mniejszej
    input=torch.tensor(float('-inf')), ← od minimalnej wartości w pierwszej trójce
    other=next_token_logits ← Przypisanie wartości -inf
)
print(new_logits) ← Zachowanie oryginalnych wartości
                    logitów dla wszystkich innych tokenów
```

Wynikowe logity dla następnego tokena w słowniku złożonym z dziewięciu tokenów to:

```
tensor([4.5100, -inf, -inf, 6.7500, -inf, -inf, -inf, 6.2800, -inf])
```

Na koniec, aby przekształcić je w prawdopodobieństwa kolejnych tokenów, stosujemy funkcję `softmax`:

```
topk_probs = torch.softmax(new_logits, dim=0) print(topk_probs)
```

Jak można zobaczyć, w wyniku zastosowania tego podejścia otrzymujemy trzy niezerowe wyniki prawdopodobieństwa:

```
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610,
       0.0000])
```

Teraz, aby wybrać następny token spośród trzech niezerowych ocen prawdopodobieństwa z użyciem próbkowania probabilistycznego, można zastosować skalowanie temperaturą i funkcję `multinomial`. Zrobimy to w kolejnym podpunkcie, dzięki odpowiedniemu zmodyfikowaniu funkcji generowania tekstu.

5.3.3. Modyfikacja funkcji generowania tekstu

W kolejnym kroku zastosujemy skalowanie temperaturą i próbkowanie top-k, aby zmodyfikować funkcję `generate_text_simple`, używaną wcześniej do generowania tekstu za pomocą modelu LLM, i stworzyć nową funkcję `generate`. Kod zmodyfikowanej funkcji zamieściłem na listingu 5.4.

Listing 5.4. Zmodyfikowana funkcja generowania tekstu o większej różnorodności

```
def generate(model, idx, max_new_tokens, context_size,
            temperature=0.0, top_k=None, eos_id=None):
    for _ in range(max_new_tokens): ← Pętla for jest taka sama jak poprzednio:
        idx_cond = idx[:, -context_size:] ← pobiera logity i skupia się tylko
        with torch.no_grad(): ← na ostatnim kroku czasowym
            logits = model(idx_cond)
```

```

logits = logits[:, -1, :]
if top_k is not None:
    top_logits, _ = torch.topk(logits, top_k)
    min_val = top_logits[:, -1]
    logits = torch.where(
        logits < min_val,
        torch.tensor(float('-inf')).to(logits.device),
        logits
    )
if temperature > 0.0:           ← Skalowanie temperaturą
    logits = logits / temperature
    probs = torch.softmax(logits, dim=-1)
    idx_next = torch.multinomial(probs, num_samples=1)
else:
    idx_next = torch.argmax(logits, dim=-1, keepdim=True)
    if idx_next == eos_id:          ← Zachowany wybór następnego tokena — tak jak poprzednio,
                                    gdy skalowanie temperaturą było wyłączone
        break
idx = torch.cat((idx, idx_next), dim=1)   ← Wcześniej zatrzymanie generowania po napotkaniu tokena oznaczającego koniec sekwencji
return idx

```

Przyjrzyjmy się teraz, jak działa nowa funkcja generate:

```

torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
print("Wynikowy tekst:\n", token_ids_to_text(token_ids, tokenizer))

```

Oto wynik działania tego kodu:

Wynikowy tekst:

Every effort moves you know began to happen a little wild--I was such a sketch
→of enough

Jak widać, wygenerowany tekst bardzo różni się od tego, który wcześniej wygenerowaliśmy za pomocą funkcji generate_simple w podrozdziale 5.3, będącego zapamiętany fragmentem ze zbioru szkoleniowego.

Ćwiczenie 5.2

Poeksperymentuj z różnymi temperaturami i ustawieniami top-k. Czy na podstawie własnych obserwacji potrafisz znaleźć zastosowania, w których są pożądane niższe wartości temperatury i parametru top-k? Czy potrafisz wskazać zastosowania, w których preferowane są wyższe wartości ustawień temperatury i parametru top-k? (Zachęcam również do ponownego wykonania tego ćwiczenia na zakończenie tego rozdziału, po załadowaniu wstępnie przeszkołonych wag z OpenAI).

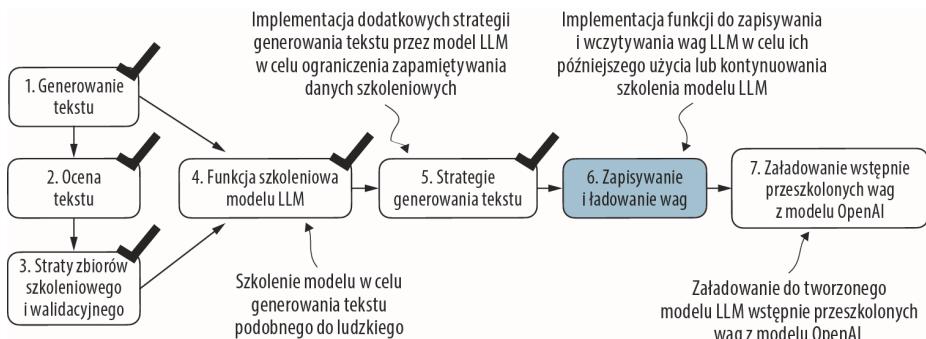
Ćwiczenie 5.3

Jakie są różne kombinacje ustawień funkcji generate pozwalające wymusić deterministyczne zachowanie, to znaczy wyeliminowanie losowego próbkowania, tak aby funkcja, podobnie jak funkcja generate_simple, zawsze generowała te same wyniki?

5.4. Wczytywanie i zapisywanie wag modeli z użyciem frameworka PyTorch

Dotychczas opisałem sposoby numerycznej oceny postępów szkolenia oraz wstępnego szkolenia modeli LLM od podstaw. Mimo że zarówno model LLM, jak i zbiór danych były stosunkowo małe, wykonanie ćwiczenia pokazało, że wstępne szkolenie modeli LLM jest kosztowne obliczeniowo. Z tego powodu istotne znaczenie ma możliwość zapisania modelu LLM tak, abyśmy nie musieli ponownie uruchamiać szkolenia za każdym razem, gdy chcemy użyć go w nowej sesji.

Przyjrzyjmy się, jak można zapisać i załadować wstępnie przeszkolony model, zgodnie z procedurą pokazaną na rysunku 5.16. Następnie załadujemy do egzemplarza klasy GPTModel wagi bardziej wydajnego, wstępnie przeszkolonego modelu GPT firmy OpenAI.



Rysunek 5.16. Po przeszkoleniu i sprawdzeniu modelu często warto go zapisać, tak aby można było go później użyć lub kontynuować szkolenie (krok 6.)

Na szczęście zapisanie modelu stworzonego za pomocą frameworka PyTorch jest stosunkowo proste. Zalecany sposób polega na zapisaniu za pomocą funkcji torch.save słownika state_dict modelu, mapującego każdą warstwę, na jej parametry:

```
torch.save(model.state_dict(), "model.pth")
```

model.pth to nazwa pliku, w którym będzie zapisany słownik `state_dict`. Rozszerzenie *.pth* jest zgodne z konwencją dla plików frameworka PyTorch, choć w zasadzie można użyć dowolnego rozszerzenia pliku.

Po zapisaniu wag modelu za pośrednictwem słownika `state_dict` wagi modelu można załadować do nowego egzemplarza obiektu `GPTModel`:

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

Jak opisywalem w rozdziale 4., dropout — technika polegająca na losowym „odrzuceniu” neuronów warstw podczas szkolenia — pomaga zapobiegać nadmiernemu dopasowaniu modelu do danych szkoleniowych. W trakcie wnioskowania nie chcemy jednak losowo odrzucać żadnych informacji, których sieć się nauczyła. Użycie funkcji `model.eval()` przełącza model do trybu oceny w celu wnioskowania. Zastosowanie tej funkcji wyłącza warstwy dropout modelu. Jeśli zamierzasz kontynuować wstępne szkolenie modelu później — na przykład za pomocą zdefiniowanej wcześniej funkcji `train_model_simple` — powinieneś także zapisać stan optymalizatora.

Optymalizatory adaptacyjne, takie jak `AdamW`, dla każdej wagi modelu przechowują dodatkowe parametry. Optymalizator `AdamW` do dynamicznego dostosowywania współczynników uczenia dla każdego parametru modelu wykorzystuje dane historyczne. Bez tego optymalizator resetuje się, a model może uczyć się nieoptymalnie lub nawet mieć problemy z właściwą zbieżnością, co jest równoznaczne z utratą zdolności do generowania spójnego tekstu. Za pomocą wywołania `torch.save` można zapisać zarówno zawartość słownika `state_dict` modelu, jak i optymalizatora:

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
},
"model_and_optimizer.pth")
```

Następnie można przywrócić stany modelu i optymalizatora, najpierw przez załadowanie zapisanych danych za pomocą metody `torch.load`, a następnie za pomocą metody `load_state_dict`:

```
checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

Ćwiczenie 5.4

Po zapisaniu wag załóż model i optymalizator w nowej sesji Pythona lub pliku notatnika Jupyter i kontynuuj wstępne szkolenie przez jeszcze jedną epokę za pomocą funkcji `train_model_simple`.

5.5. Ładowanie wstępnie przeszkołonych wag z modelu OpenAI

Wcześniej przeszkołiliśmy niewielki model GPT-2 z użyciem ograniczonego zbioru danych obejmującego krótkie opowiadanie. Takie podejście pozwoliło skupić się na podstawach i nie wymagało poświęcania dużej ilości czasu i zasobów obliczeniowych.

Na szczęście OpenAI udostępniło wagę swoich modeli GPT-2 za darmo. Dzięki temu nie trzeba inwestować dziesiątek, a nawet setek tysięcy dolarów w ponowne szkolenie modelu na dużym korpusie dokumentów. Spróbujmy więc załadować te wagę do egzemplarza klasy `GPTModel` i użyć modelu do generowania tekstu. W tym kontekście *wagi* odnoszą się do parametrów przechowywanych w atrybutach `.weight` warstw `Linear` i `Embedding` frameworka PyTorch. Dostęp do tych parametrów uzyskiwaliśmy wcześniej podczas szkolenia modelu za pośrednictwem funkcji `model.parameters()`. W rozdziale 6. ponownie wykorzystamy wstępnie przeszkołone wagę, aby dostroić model do zadania klasyfikacji tekstu, stosując podejście podobne do tego, które wykorzystuje ChatGPT.

Warto zwrócić uwagę, że firma OpenAI pierwotnie zapisywała wagę GPT-2 z użyciem biblioteki TensorFlow. Aby załadować wagę w Pythonie, trzeba ją zainstalować. W zaprezentowanym poniżej kodzie do śledzenia procesu pobierania wykorzystałem narzędzie paska postępu o nazwie `tqdm`. To narzędzie również trzeba zainstalować.

Aby zainstalować potrzebne biblioteki, uruchom w terminalu następujące polecenie:

```
pip install tensorflow>=2.15.0 tqdm>=4.66
```

Kod umożliwiający pobranie wag z internetu jest dość rozwiązkowy, w większości standarowy i niezbyt interesujący. Dlatego zamiast poświęcać cenne miejsce na omawianie kodu Pythona do pobierania plików z internetu, pobierzemy moduł Pythona `gpt_download.py` bezpośrednio z repozytorium online tego rozdziału:

```
import urllib.request
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

Następnie, po pobraniu pliku do lokalnego katalogu sesji Pythona, należy sprawdzić zawartość tego pliku, aby zyskać pewność, że został poprawnie zapisany i zawiera odpowiedni kod Pythona.

Możesz teraz zaimportować funkcję `download_and_load_gpt2` z pliku `gpt_download.py` w sposób pokazany poniżej. Spowoduje to załadowanie do sesji Pythona ustawień architektury GPT-2 (`settings`) i parametrów wag (`params`):

```
from gpt_download import download_and_load_gpt2
settings,
params = download_and_load_gpt2(
    model_size="124M", models_dir="gpt2"
)
```

Uruchomienie tego kodu spowoduje pobranie następujących siedmiu plików powiązanych z modelem GPT-2 o 124 milionach parametrów:

```
checkpoint: 100%|██████████| 77.0/77.0 [00:00<00:00,
63.9kiB/s]
encoder.json: 100%|██████████| 1.04M/1.04M [00:00<00:00,
2.20MiB/s]
hparams.json: 100%|██████████| 90.0/90.0 [00:00<00:00,
78.3kiB/s]
model.ckpt.data-00000-of-00001: 100%|██████████| 498M/498M [01:09<00:00,
7.16MiB/s]
model.ckpt.index: 100%|██████████| 5.21k/5.21k [00:00<00:00,
3.24MiB/s]
model.ckpt.meta: 100%|██████████| 471k/471k [00:00<00:00,
2.46MiB/s]
vocab.bpe: 100%|██████████| 456k/456k [00:00<00:00,
1.70MiB/s]
```

UWAGA Podczas pobierania wag mogą wystąpić problemy związane z przerwami w połączeniu internetowym z serwerem lub zmianami w sposobie udostępniania przez OpenAI darmowych wag modelu GPT-2. Jeśli napotkasz problemy, odwiedź internetowe repozytorium kodu tego rozdziału pod adresem <https://github.com/rasbt/LLMs-from-scratch>, gdzie znajdziesz aktualne instrukcje pobierania. Możesz także poszukać odpowiedzi za pośrednictwem forum wydawnictwa Manning.

Aby sprawdzić, czy uruchomienie poprzedniego kodu zakończyło się sukcesem, sprawdź zawartość zmiennych `settings` i `params`:

```
print("Ustawienia:", settings)
print("Klucze słownika parametrów:", params.keys())
```

Oto wynik działania tego kodu:

```
Ustawienia: {'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_head': 12,
'n_layer': 12}
Klucze słownika parametrów: dict_keys(['blocks', 'b', 'g', 'wpe', 'wte'])
```

Zarówno `settings`, jak i `params` są słownikami Pythona. Słownik `settings` przechowuje ustawienia architektury LLM podobne do tych, które przechowywaliśmy w ręcznie zdefiniowanym słowniku `GPT_CONFIG_124M`. Słownik `params` zawiera rzeczywiste tensoły wag. Zwróć uwagę, że wyświetliśmy tylko klucze słownika. Wyświetlenie wartości wag zajęłoby zbyt dużo miejsca na ekranie. Można jednak sprawdzić tensoły wag przez wyświetlenie całego słownika za pomocą instrukcji `print(params)`. Można też wybrać poszczególne tensoły za pomocą odpowiednich kluczów słownika, na przykład wag warstwy osadzania:

```
print(params["wte"])
print("Wymiary tensora wag osadzeń tokenów:", params["wte"].shape)
```

Wagi warstwy osadzeń tokenów są następujące:

```
[[-0.11010301 ... -0.1363697  0.01506208  0.04531523]
 [ 0.04034033 ...  0.08605453  0.00253983  0.04318958]
 [-0.12746179 ...  0.08991534 -0.12972379 -0.08785918]
 ...
 [-0.04453601 ...  0.10435229  0.09783269 -0.06952604]
 [ 0.1860082 ... -0.09625227  0.07847701 -0.02245961]
 [ 0.05135201 ...  0.00704835  0.15519823  0.12067825]]
```

Wymiary tensora wag osadzeń tokenów: (50257, 768)

Za pomocą ustawienia `download_and_load_gpt2(model_size='124M', ...)` pobraliśmy i wczytaliśmy wagi najmniejszego modelu GPT-2. Firma OpenAI udostępnia również wagi większych modeli: o rozmiarach 355, 774 i 1558 milionów parametrów. Ogólnie rzecz biorąc, architektura tych modeli GPT jest taka sama (rysunek 5.17), z wyjątkiem tego, że różne elementy architektury są powtórzone różną liczbą razy. Inny jest także rozmiar osadzeń.

Kod w dalszej części tego rozdziału jest kompatybilny również z tymi większymi modelami.

Po załadowaniu wag modelu GPT-2 do Pythona trzeba jeszcze przenieść je ze słowników `settings` i `params` do egzemplarza klasy `GPTModel`. Najpierw tworzymy słownik, w którym zestawione różnice pomiędzy różnymi rozmiarami modeli GPT z rysunku 5.17:

```
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
```

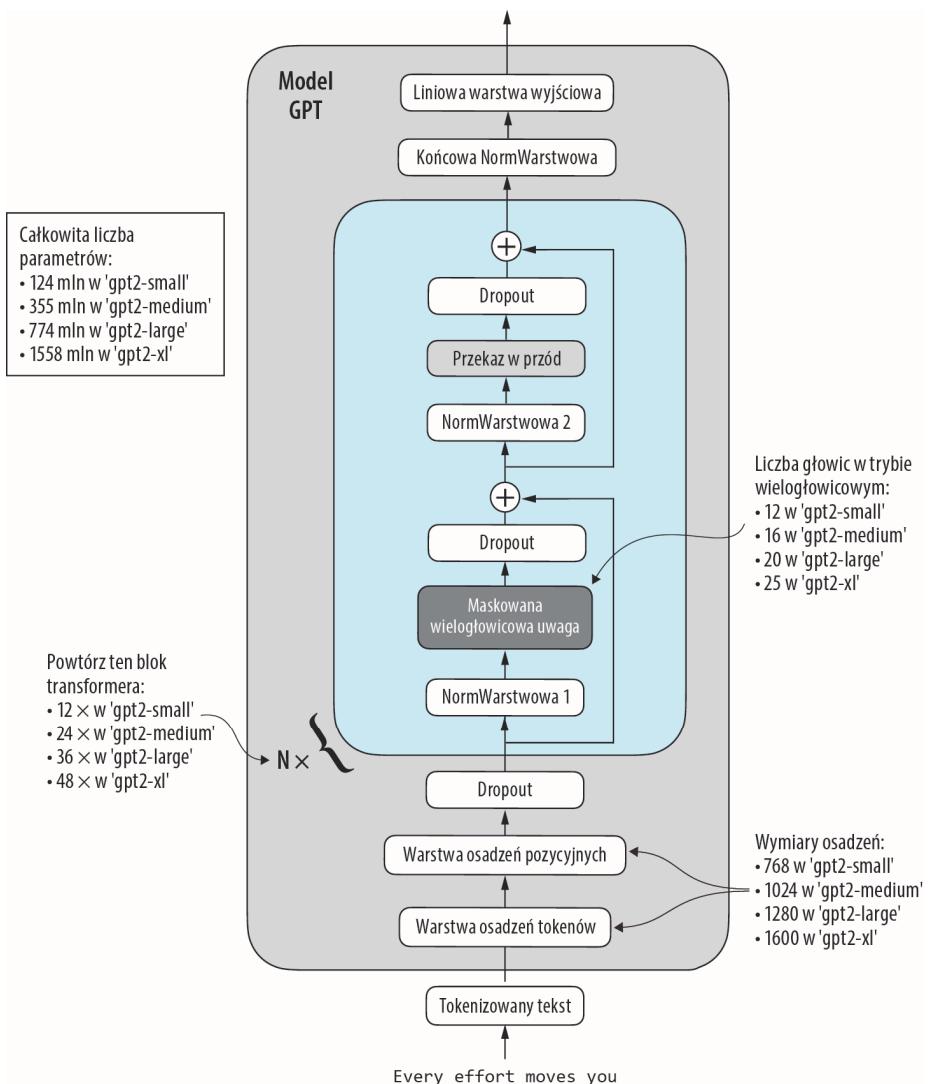
Załóżmy, że jesteśmy zainteresowani załadowaniem najmniejszego modelu, "gpt2-small (124M)". Aby zaktualizować pełnowymiarowy słownik `GPT_CONFIG_124M`, który zdefiniowaliśmy i wykorzystaliśmy wcześniej, możesz użyć odpowiednich ustawień z tabeli `model_configs`:

```
model_name = "gpt2-small (124M)"
NEW_CONFIG = GPT_CONFIG_124M.copy()
NEW_CONFIG.update(model_configs[model_name])
```

Uważni Czytelnicy z pewnością pamiętają, że wcześniej używaliśmy tokenów o długości 256. Oryginalne modele GPT-2 firmy OpenAI były jednak szkolone dla długości 1024-tokenów. Trzeba więc odpowiednio zaktualizować ustawienie `NEW_CONFIG`:

```
NEW_CONFIG.update({"context_length": 1024})
```

Ponadto firma OpenAI wykorzystała wektory stronniczości w warstwach liniowych modułu wielogłowicowej uwagi do implementowania obliczeń macierzy zapytań, kluczy i wartości. Wektorów stronniczości już nie stosuje się powszechnie w modelach LLM, nie poprawiają bowiem wydajności modelowania, a zatem nie są konieczne.



Rysunek 5.17. Modele LLM GPT-2 są dostępne w kilku różnych rozmiarach, od 124 milionów do 1558 milionów parametrów. Ogólna architektura jest taka sama, a jedyną różnicą są rozmiary osadzeń i liczba powtórzeń poszczególnych elementów, takich jak głowice uwagi i bloki transformera

Ponieważ jednak pracujemy ze wstępnie przeszkolonymi wagami, powinniśmy dopasować ustawienia pod kątem spójności i włączyć wektory stronniczości:

```
NEW_CONFIG.update({"qkv_bias": True})
```

Możesz teraz użyć zaktualizowanego słownika NEW_CONFIG w celu zainicjowania nowego egzemplarza klasy GPTModel:

```
gpt = GPTModel(NEW_CONFIG)
gpt.eval()
```

Na potrzeby wstępnego szkolenia egzemplarz `GPTModel` jest domyślnie inicjowany za pomocą losowych wag. Ostatnim krokiem wymaganym do tego, aby można było skorzystać z wag modelu OpenAI, jest zastąpienie losowych wag wagami wczytanymi do słownika `params`. W tym celu najpierw definiujemy prostą funkcję narzędziową `assign`. Funkcja sprawdza, czy dwa tensory, czyli tablice (`left` i `right`), mają te same wymiary i kształt, i zwraca prawy tensor jako trenowalne parametry PyTorch:

```
def assign(left, right):
    if left.shape != right.shape:
        raise ValueError(f"Niedopasowany kształt. Lewa: {left.shape}, "
                         "Prawa: {right.shape}")
    )
    return torch.nn.Parameter(torch.tensor(right))
```

Następnie definiujemy funkcję `load_weights_into_gpt`, która ładuje wagi ze słownika `params` do egzemplarza modelu `GPTModel` (listing 5.5).

Listing 5.5. Ładowanie wag OpenAI do kodu modelu GPT

```
import numpy as np.

def load_weights_into_gpt(gpt, params): ← Ustawienie wag pozycyjnych i wag osadzeń tokenów
    gpt.pos_emb.weight = assign(gpt.pos_emb.weight, params['wpe']) ← modelu na wartości określone w parametrach
    gpt.tok_emb.weight = assign(gpt.tok_emb.weight, params['wte'])

    for b in range(len(params["blocks"])): ← Iterowanie po wszystkich
        q_w, k_w, v_w = np.split( ← blokach transformera w modelu
            (params["blocks"][b]["attn"]["c_attn"])["w"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.weight = assign(
            gpt.trf_blocks[b].att.W_query.weight, q_w.T)
        gpt.trf_blocks[b].att.W_key.weight = assign(
            gpt.trf_blocks[b].att.W_key.weight, k_w.T)
        gpt.trf_blocks[b].att.W_value.weight = assign(
            gpt.trf_blocks[b].att.W_value.weight, v_w.T) ← Funkcja np.split służy
                                                               do dzielenia wag uwagi
                                                               i stronniczości
                                                               na trzy równe części,
                                                               odpowiadające
                                                               komponentom zapytań,
                                                               kluczy i wartości

        q_b, k_b, v_b = np.split(
            (params["blocks"][b]["attn"]["c_attn"])["b"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.bias = assign(
            gpt.trf_blocks[b].att.W_query.bias, q_b)
        gpt.trf_blocks[b].att.W_key.bias = assign(
            gpt.trf_blocks[b].att.W_key.bias, k_b)
        gpt.trf_blocks[b].att.W_value.bias = assign(
            gpt.trf_blocks[b].att.W_value.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"]["w"].T)
        gpt.trf_blocks[b].att.out_proj.bias = assign(
            gpt.trf_blocks[b].att.out_proj.bias,
            params["blocks"][b]["attn"]["c_proj"]["b"])
```

```

gpt.trf_blocks[b].ff.layers[0].weight = assign(
    gpt.trf_blocks[b].ff.layers[0].weight,
    params["blocks"][b]["mlp"]["c_fc"]["w"].T)
gpt.trf_blocks[b].ff.layers[0].bias = assign(
    gpt.trf_blocks[b].ff.layers[0].bias,
    params["blocks"][b]["mlp"]["c_fc"]["b"])
gpt.trf_blocks[b].ff.layers[2].weight = assign(
    gpt.trf_blocks[b].ff.layers[2].weight,
    params["blocks"][b]["mlp"]["c_proj"]["w"].T)
gpt.trf_blocks[b].ff.layers[2].bias = assign(
    gpt.trf_blocks[b].ff.layers[2].bias,
    params["blocks"][b]["mlp"]["c_proj"]["b"])

gpt.trf_blocks[b].norm1.scale = assign(
    gpt.trf_blocks[b].norm1.scale,
    params["blocks"][b]["ln_1"]["g"])
gpt.trf_blocks[b].norm1.shift = assign(
    gpt.trf_blocks[b].norm1.shift,
    params["blocks"][b]["ln_1"]["b"])
gpt.trf_blocks[b].norm2.scale = assign(
    gpt.trf_blocks[b].norm2.scale,
    params["blocks"][b]["ln_2"]["g"])
gpt.trf_blocks[b].norm2.shift = assign(
    gpt.trf_blocks[b].norm2.shift,
    params["blocks"][b]["ln_2"]["b"])

gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g"])
gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b"])
gpt.out_head.weight = assign(gpt.out_head.weight, params["wte"])

```

W oryginalnym modelu GPT-2
 firmy OpenAI, aby zmniejszyć
 całkowitą liczbę parametrów,
 w warstwie wyjściowej
 wykorzystywano wagie osadzeń
 tokenów; technikę tę określa
 się jako **współdzielenie wag**

W funkcji `load_weights_into_gpt` starannie dopasowujemy wagie z implementacji OpenAI do implementacji modelu `GPTModel`. Posłużymy się konkretnym przykładem: firma OpenAI zapisała tensor wagie dla wyjściowej warstwy rzutowania pierwszego bloku transformera jako `params['blocks'][0]['attn']['c_proj']['w']`. W prezentowanej implementacji ten tensor wag odpowiada właściwości `gpt.trf_blocks[b].att[0].out_proj.weight`, gdzie `gpt` jest egzemplarzem klasy `GPTModel`.

Zaprogramowanie funkcji `load_weights_into_gpt` wymagało dokładnej analizy kodu, dlatego że firma OpenAI użyła nieco innej konwencji nazewnictwa niż ta, którą zastosowaliśmy w prezentowanym przykładzie. Gdybyśmy jednak próbowały dopasować dwa tensory o różnych wymiarach, funkcja `assign` wygenerowałaby ostrzeżenie. Poza tym, gdybyśmy popełnili błąd w tej funkcji, zauważlibyśmy to, ponieważ wynikowy model GPT nie byłby w stanie wygenerować spójnego tekstu.

Wypróbujmy teraz funkcję `load_weights_into_gpt` w praktycznym działaniu. Spróbujmy załadować wagie modelu OpenAI do egzemplarza klasy `GPTModel` – zmieńmy `gpt`:

```
load_weights_into_gpt(gpt, params)
gpt.to(device)
```

Jeśli model został poprawnie załadowany, można go wykorzystać do wygenerowania nowego tekstu za pomocą zdefiniowanej wcześniej funkcji `generate`:

```
torch.manual_seed(3)
token_ids = generate(
    model=gpt,
    idx=text_to_token_ids("Every effort moves you", tokenizer).to(device),
    max_new_tokens=25,
    context_size=NEW_CONFIG["context_length"],
    top_k=50,
    temperature=1.5
)
print("Wynikowy tekst:\n", token_ids_to_text(token_ids, tokenizer))
```

Wynikowy tekst wygląda następująco:

Wynikowy tekst:
Every effort moves you along to solve more, bigger questions and make the day.
→It's a simple idea! With just a single touch we can⁹

Możemy być pewni, że wagie modelu zostały załadowane poprawnie, ponieważ model wygenerował spójny tekst. Drobny błąd w tym procesie spowodowałby błąd modelu. W kolejnych rozdziałach będziemy dalej pracować ze wstępnie przeszkolonym modelem z tego rozdziału i dostrajać go do zadań klasyfikowania tekstu i wykonywania instrukcji.

Ćwiczenie 5.5

Oblicz straty zbioru szkoleniowego i walidacyjnego modelu GPTModel z wagami modeli OpenAI wstępnie przeszkolonymi na zbiorze danych opowiadania *The Verdict*.

Ćwiczenie 5.6

Poeksperymentuj z modelami GPT-2 o różnych rozmiarach — na przykład z największym modelem o 1558 milionach parametrów. Następnie porównaj wygenerowany tekst z tym, który wygenerował model o 124 milionach parametrów.

Podsumowanie

- Model LLM generuje tekst po jednym tokenie na raz.
- Następny token jest domyślnie generowany przez skonwertowanie wyników modelu na oceny prawdopodobieństwa i wybranie ze słownika tokenu odpowiadającego najwyższej ocenie prawdopodobieństwa. Takie zachowanie określa się jako „zachłanne dekodowanie”.

⁹ Dosłownie: każdy wysiłek przybliża cię do rozwiązywania kolejnych, ważniejszych problemów i sprawia, że dzień staje się lepszy. To bardzo proste! Za pomocą jednego dotknięcia można... To zdanie nie brzmi idealnie ani w oryginale, ani w tłumaczeniu, ale przypomina tekst, który mógłby w pewnym kontekście być wypowiedziany przez człowieka — przyp. tłum.

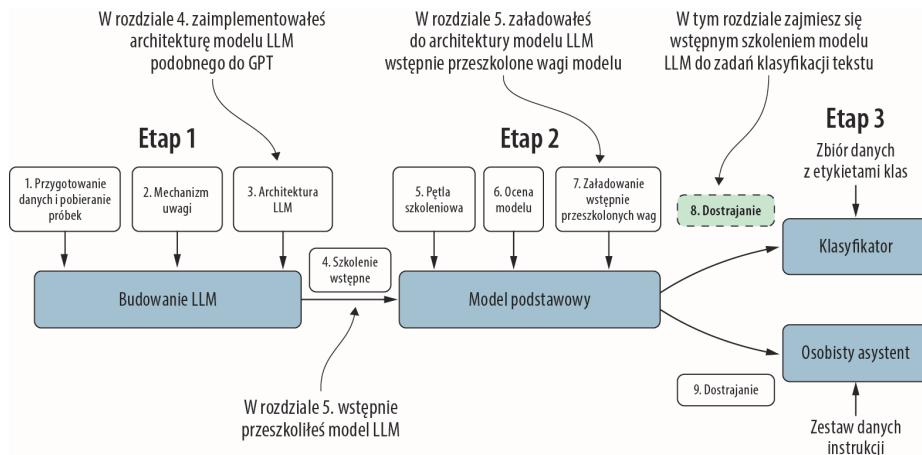
- Zastosowanie technik probabilistycznego próbkowania i skalowania temperaturą pozwala wpływać na różnorodność i spójność generowanego tekstu.
- Straty w zbiorze szkoleniowym i walidacyjnym można wykorzystać do oceny jakości tekstu generowanego przez model LLM podczas szkolenia.
- Wstępne szkolenie modelu LLM polega na modyfikacji jego wag w celu zminimalizowania strat związanych ze szkoleniem.
- Pętla szkoleniowa dla modelu LLM jest standardową procedurą w uczeniu głębokim, wykorzystującą konwencjonalną strategię entropii krzyżowej i optymalizator AdamW.
- Wstępne szkolenie modeli LLM na obszernym korpusie tekstowym jest czasochłonne i wymaga dużych zasobów obliczeniowych. Jako alternatywę dla samodzielnego wstępnego szkolenia modelu na dużym zbiorze danych można więc załadować wagi dostępne za darmo.

Dostrajanie modelu LLM do zadań klasyfikacji

W tym rozdziale:

- Wprowadzenie różnych podejść do dostrajania modeli LLM
- Przygotowanie zbioru danych do klasyfikowania tekstu
- Modyfikowanie wstępnie przeszkolonego modelu LLM w celu dostrajania
- Dostrajanie modelu LLM w celu identyfikacji wiadomości spamowych
- Ocena dokładności dostrojonego klasyfikatora LLM
- Używanie dostrojonego modelu LLM w celu klasyfikowania nowych danych

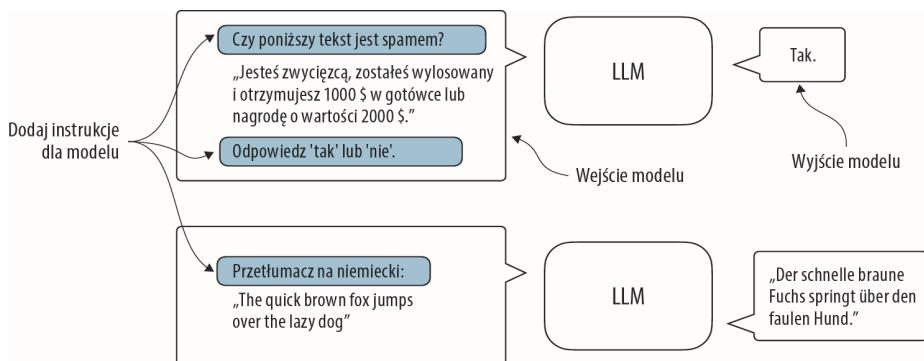
Do tej pory zakodowaleś architekturę LLM, wstępnie przeszkołileś model i nauczyleś się importować do tworzonego modelu wagę wstępnie przeszkolonego modelu z zewnętrznego źródła (na przykład modelu OpenAI). W tym rozdziale pokażę Ci, jak dostroić model LLM do konkretnego zadania końcowego, na przykład klasyfikacji tekstu. W ramach przykładu pokażę sposób dostrajania wiadomości w celu klasyfikowania wiadomości tekstowych jako „spam” lub „nie spam”. Dwa główne sposoby dostrajania modeli LLM: dostrajanie do zadań klasyfikacji (krok 8.) i dostrajanie do wykonywania instrukcji (krok 9.) przedstawiłem na [rysunku 6.1](#).



Rysunek 6.1. Trzy główne etapy kodowania modelu LLM. Ten rozdział skupia się na etapie 3. (krok 8.) — dostrajaniu wstępnie przeszkołonego modelu LLM jako klasyfikatora

6.1. Różne kategorie dostrajania

Do najczęstszych sposobów dostrajania modeli językowych należą *dostrajanie do wykonywania instrukcji* i *dostrajanie do zadań klasyfikacji*. Dostrajanie do wykonywania instrukcji obejmuje szkolenie modelu językowego na zbiorze zadań formułowanych za pomocą instrukcji. Jego celem jest poprawa zdolności modelu do rozumienia i wykonywania zadań opisanych w promptach formułowanych w języku naturalnym, co pokazałem na rysunku 6.2.

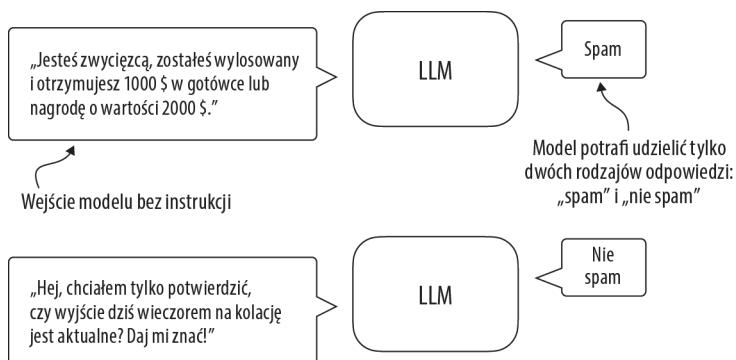


Rysunek 6.2. Dwa różne scenariusze dostrajania do wykonywania instrukcji. W górnej części rysunku model ma za zadanie określenie, czy podany tekst jest spamem. W dolnej części model otrzymuje zadanie przetłumaczenia angielskiego zdania na język niemiecki

W dostrajaniu do zadań klasyfikacji (to pojęcie może być znane Czytelnikom mającym doświadczenie w uczeniu maszynowym) model jest szkolony pod kątem rozpoznawania

określonego zbioru etykiet reprezentujących klasy, takie jak „spam” i „nie spam”. Przykłady zadań klasyfikacyjnych wykraczają poza filtrowanie odpowiedzi modeli LLM czy wiadomości e-mail: obejmują identyfikację różnych gatunków roślin na podstawie zdjęć, przyporządkowywanie artykułów informacyjnych do tematów takich jak sport, polityka czy technika, a nawet rozróżnianie pomiędzy guzami łagodnymi a złośliwymi w zadaniach analizy zdjęć medycznych.

Należy zwrócić uwagę, że model dostrojony do zadań klasyfikacji może przewidywać tylko te klasy, które napotkał podczas szkolenia. Na przykład może określić, czy wiadomość jest „spamem” lub „nie jest spamem” (patrz rysunek 6.3), ale nie może powiedzieć niczego więcej o wprowadzonym tekście.



Rysunek 6.3. Scenariusz klasyfikowania tekstu przez model LLM. Model dostrojony do zadań klasyfikacji spamu nie wymaga przekazywania z danymi wejściowymi dodatkowych instrukcji. W przeciwieństwie do modelu dostrojonego do wykonywania instrukcji, potrafi odpowiedzieć tylko „spam” lub „nie spam”

Model dostrojony do wykonywania instrukcji zazwyczaj potrafi wykonać szerszy zakres zadań niż model dostrojony do zadań klasyfikacji przedstawiony na rysunku 6.3. Model dostrojony do zadań klasyfikacji można postrzegać jako wysoce wyspecjalizowany. Zasadniczo model specjalistyczny łatwiej opracować od modelu ogólnego, który dobrze się sprawdza w różnych zadaniach.

6.2. Przygotowanie zbioru danych

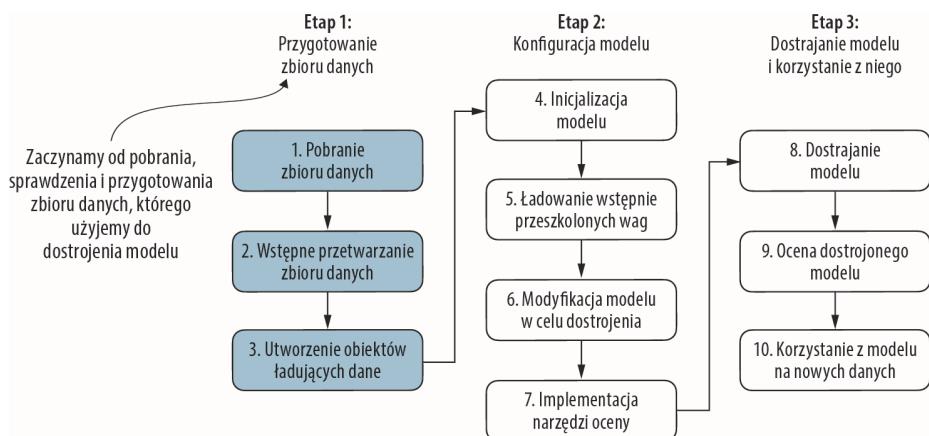
W tym rozdziale zmodyfikujesz model GPT, który wcześniej zaimplementowałeś i wstępnie przeszkoliłeś, oraz dostroisz go pod kątem zadań klasyfikacji. Na początek trzeba pobrać i przygotować zbiór danych, tak jak pokazałem na rysunku 6.4. Aby stworzyć intuicyjny i użyteczny przykład dostrajania do zadań klasyfikacji, będziemy pracować ze zbiorem danych wiadomości tekstowych złożonym z wiadomości będących spamem oraz niebędącymi spamem.

Proces zaczyna się od pobrania zbioru danych (listing 6.1).

Wybór właściwego podejścia

Dostrajanie do wykonywania instrukcji poprawia zdolności modelu do rozumienia konkretnych instrukcji przekazanych przez użytkownika i generowania na ich podstawie odpowiedzi. Dostrajanie do wykonywania instrukcji najlepiej sprawdza się w modelach, które są przeznaczone do realizacji różnych zadań według złożonych instrukcji użytkownika, co poprawia elastyczność i jakość interakcji. Dostrajanie do zadań klasyfikacji idealnie sprawdza się w projektach wymagających precyzyjnej kategoryzacji danych do predefiniowanych klas, takich jak zadania analizy tonu lub mechanizm detekcji spamu.

Chociaż dostrajanie do wykonywania instrukcji pozwala uzyskać bardziej wszechstronny model, utworzenie modelu sprawnie wykonującego różne zadania wymaga większych zbiorów danych i większych zasobów obliczeniowych. Natomiast dostrajanie do zadań klasyfikacji wymaga mniejszej ilości danych i mniejszej mocy obliczeniowej, ale zastosowania tak dostrojonych modeli są ograniczone do określonych klas, na których model przeszkołono.



Rysunek 6.4. Trzyetapowy proces dostrajania modelu LLM do zadań klasyfikacji. Etap 1. obejmuje przygotowanie zbioru danych. Etap 2. koncentruje się na konfiguracji modelu. Etap 3. obejmuje dostrajanie i ocenę modelu

UWAGA Wiadomości tekstowe zazwyczaj wysyła się za pośrednictwem telefonu, a nie poczty e-mail. Jednak te same kroki mają zastosowanie w klasyfikowaniu wiadomości e-mail. Zainteresowani Czytelnicy mogą znaleźć odnośniki do zbiorów danych klasyfikacji spamu w „Dodatku B”.

Listing 6.1. Pobieranie zbioru danych i rozpakowywanie go

```

import urllib.request
import zipfile
import os
from pathlib import Path

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
    
```

```

data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

def download_and_unzip_spam_data(
    url, zip_path, extracted_path, data_file_path):
    if data_file_path.exists():
        print(f"{data_file_path} istnieje. Pomijam pobieranie "
              "i wyodrębnianie.")
    )
    Return

    with urllib.request.urlopen(url) as response:           ← Pobranie pliku
        with open(zip_path, "wb") as out_file:
            out_file.write(response.read())

    with zipfile.ZipFile(zip_path, "r") as zip_ref:          ← Rozpakowanie pliku
        zip_ref.extractall(extracted_path)

    original_file_path = Path(extracted_path) / "SMSSpamCollection"
    os.rename(original_file_path, data_file_path)           ← Dodanie rozszerzenia
    print(f"Plik pobrany i zapisany jako {data_file_path}") | pliku .tsv

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

```

Uruchomienie powyższego kodu spowoduje zapisanie zbioru danych *SMSSpamCollection.tsv*, w formacie pliku tekstowego rozdzielonego tabulatorami, w folderze *sms_spam_collection*. Można go załadować do struktury DataFrame biblioteki pandas w następujący sposób:

```

import pandas as pd
df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Etykieta", "Tekst"]
) df           ← Wyświetla ramkę danych w notatniku Jupyter. Można też użyć instrukcji print(df)

```

Wynikową ramkę danych zbioru danych zawierających spam przedstawiłem na rysunku 6.5.

Przyjrzyjmy się rozkładowi etykiet klas:

```
print(df["Etykieta"].value_counts())
```

Dzięki uruchomieniu powyższego kodu można stwierdzić, że dane sklasyfikowane jako „ham” (tzn. nie spam) występują znacznie częściej niż „spam”:

Etykieta	Count
ham	4825
spam	747

Name: count, dtype: int64

Dla uproszczenia i ponieważ preferujemy niewielkie zbiory danych (dzięki czemu dostrojenie modelu LLM będzie szybsze), zdecydowaliśmy się na próbkowanie zbioru danych w taki sposób, aby uwzględnić po 747 egzemplarzy z każdej klasy.

Aby zmniejszyć próbę i utworzyć zrównoważony zbiór danych, można użyć kodu zamieszczonego na listingu 6.2.

Etykieta		Tekst
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...
5567	spam	This is the 2nd time we have tried 2 contact u...
5568	ham	Will ü b going to esplanade fr home?
5569	ham	Pity, * was in mood for that. So...any other s...
5570	ham	The guy did some bitching but I acted like i'd...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Rysunek 6.5. Podgląd zbioru danych SMSSpamCollection w strukturze DataFrame biblioteki pandas, pokazujący etykiety klas („ham” lub „spam”) i odpowiadające im wiadomości tekstowe. Zbiór danych składa się z 5572 wierszy (wiadomości tekstowe i etykiety)

UWAGA Dostępnych jest kilka innych metod obsługiwanego problemów związanych z nie równym rozłożeniem klas, ale ich opis wykracza poza zakres tej książki. Czytelnicy, którzy chcieliby się zapoznać z metodami postępowania w takich przypadkach, dodatkowe informacje znajdują w „Dodatku B”.

Listing 6.2. Tworzenie zrównoważonego zbioru danych¹

```
def create_balanced_dataset(df):
    num_spam = df[df["Etykieta"] == "spam"].shape[0]           ← Zliczanie egzemplarzy
    ham_subset = df[df["Etykieta"] == "ham"].sample(            sklasifikowanych jako „spam”
        num_spam, random_state=123)                                ← Losowe próbkowanie wiadomości
    balanced_df = pd.concat([                                     oznaczonych jako „ham” w celu dopasowania
        ham_subset, df[df["Etykieta"] == "spam"]])                liczby wiadomości oznaczonych jako „spam”
    ])                                                               ← Połączenie podzbiorów wiadomości
    return balanced_df                                            klasa „ham” z klasą „spam”
```

```
balanced_df = create_balanced_dataset(df)
print(balanced_df["Etykieta"].value_counts())
```

¹ W przykładowym zbiorze danych wiadomości niebędące spamem zostały oznaczone etykietą „ham” (czyli „szynka”), w przeciwieństwie do etykiety „spam” (czyli „mielonka”) – *przyp. tłum.*

Po wykonaniu poprzedniego kodu w celu zrównoważenia zbioru danych można zauważyc, że zbiór danych zawiera teraz taką samą liczbę wiadomości spamowych i nie-spamowych:

```
Etykieta
ham      747
spam     747
Name: count, dtype: int64
```

Następnie dokonamy konwersji tekstowych etykiet klas „ham” i „spam” na etykiety w postaci liczb całkowitych, odpowiednio o i 1:

```
balanced_df["Etykieta"] = balanced_df["Etykieta"].map({"ham": 0, "spam": 1})
```

Ten proces przypomina konwertowanie tekstu na identyfikatory tokenów. Jednak zamiast ze słownictwem GPT, które składa się z ponad 50 tysięcy słów, mamy do czynienia tylko z dwoma identyfikatorami tokenów: 0 i 1.

W kolejnym kroku, w celu podzielenia zbioru danych na trzy części: 70% na szkolenie, 10% na walidację i 20% na testowanie (takie współczynniki podziału powszechnie stosuje się w uczeniu maszynowym do szkolenia, dostosowywania i oceny modeli), stworzymy funkcję random_split (listing 6.3).

Listing 6.3. Podział zbioru danych

```
def random_split(df, train_frac, validation_frac):
    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True) ← Tasowanie całej zawartości ramki DataFrame
    train_end = int(len(df) * train_frac) ← Obliczenie indeksów podziału
    validation_end = train_end + int(len(df) * validation_frac)
    ← Podzielenie ramki DataFrame
    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df
train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1) ← Wielkość części testowej wynosi 0,2, co stanowi pozostałą część zbioru
```

Zapiszmy zbiór danych w formacie pliku CSV, tak aby można było z niego skorzystać później:

```
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Do tej pory pobrałeś zbiór danych, doprowadziłeś do jego zrównoważenia i podzieliłeś go na podzbiory szkoleniowy i ewaluacyjny. W kolejnym kroku skonfigurujesz mechanizmy ładowania danych frameworka PyTorch, które będą wykorzystywane do szkolenia modelu.

6.3. Tworzenie mechanizmów ładujących dane

W tym podrozdziale opracujesz mechanizmy ładowania danych frameworka PyTorch, które pod względem koncepcyjnym przypominają te, które zaimplementowałeś wcześniej podczas pracy z danymi tekstowymi. Do generowania fragmentów tekstu o jednakowym rozmiarze, które następnie pogrupowałeś w partie w celu bardziej wydajnego uczenia modelu, wykorzystałeś technikę okna przesuwnego. Każdy fragment działał jako pojedyncza jednostka szkoleniowa. Teraz jednak pracujesz ze zbiorom danych wiadomości spamowych, zawierającym wiadomości tekstowe o różnej długości. Aby utworzyć partie takich wiadomości, możesz użyć, podobnie jak w przypadku fragmentów tekstu, jednej z dwóch podstawowych opcji:

- obcięcie wszystkich wiadomości do długości najkrótszej wiadomości w zbiorze danych lub partii,
- wypełnienie wszystkich wiadomości do długości najkrótszej wiadomości w zbiorze danych lub partii.

Pierwsza opcja jest mniej kosztowna obliczeniowo, ale może powodować znaczącą utratę informacji w przypadku, gdy krótsze wiadomości są znacznie mniejsze niż średnie lub najdłuższe wiadomości, co może pogorszyć wydajność modelu. Wybieramy zatem drugą opcję, pozwalającą zachować całą treść wszystkich wiadomości.

Aby zaimplementować taki podział na partie, gdzie wszystkie wiadomości są uzupełniane do długości najdłuższej wiadomości w zbiorze danych, dodajemy do wszystkich krótkich wiadomości tokeny dopełniające. Do tego celu używamy tokena "`<|en>|doftext|>`".

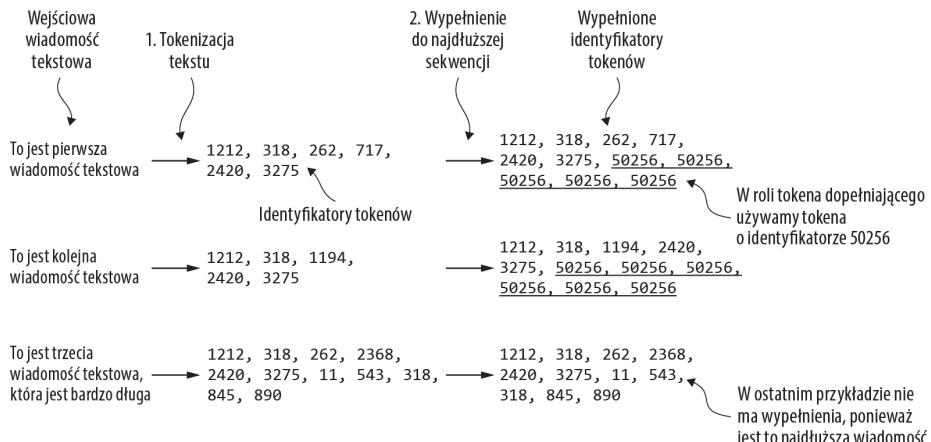
Jednak zamiast do każdej wiadomości tekstowej bezpośrednio dołączać ciąg "`<|endoftext|>`", do zakodowanych wiadomości tekstowych można dodać identyfikator odpowiadający tokenowi "`<|endoftext|>`" (rysunek 6.6). Identyfikator tokena dopełniającego "`<|endoftext|>`" to 50256. Aby się upewnić, że identyfikator tokena jest poprawny, można zakodować token "`<|endoftext|>`" za pomocą dostępnego w pakiecie `tiktoken` tokenizera *GPT-2*, którego używaliśmy wcześniej:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

Uruchomienie powyższego kodu zwraca wartość [50256].

Przed utworzeniem egzemplarzy obiektów ładowania danych trzeba zaimplementować obiekt `Dataset` frameworka PyTorch, określający sposób ładowania i przetwarzania danych. W tym celu zdefiniujemy klasę `SpamDataset` (listing 6.4), która implementuje pojęcia przedstawione na rysunku 6.6. Klasa `SpamDataset` obsługuje kilka kluczowych

zadań: identyfikuje najdłuższą sekwencję w szkoleniowym zbiorze danych, koduje wiadomości tekstowe i zapewnia dopasowanie długości najdłuższej sekwencji przez uzupełnienie sekwencji *tokenem wypełniającym*.



Rysunek 6.6. Proces przygotowania wejściowego tekstu. Najpierw każda wejściowa wiadomość tekstowa jest konwertowana na ciąg identyfikatorów tokenów. Następnie, aby zapewniona była jednolita długość sekwencji (długość była dopasowana do najdłuższej sekwencji), krótsze sekwencje są uzupełniane tokenem wypełniającym (w tym przypadku o identyfikatorze 50256)

Listing 6.4. Konfigurowanie klasy Dataset framework Pytorch

```
import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None,
                 pad_token_id=50256):
        self.data = pd.read_csv(csv_file)
        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length

        self.encoded_texts = [
            encoded_text[:self.max_length]
            for encoded_text in self.encoded_texts
        ]

        self.encoded_texts = [
            encoded_text + [pad_token_id] * (self.max_length - len(encoded_text))
            for encoded_text in self.encoded_texts
        ]
```

Annotations explaining the code steps:

- "Wstępna tokenizacja wiadomości tekstowych" (Initial tokenization of textual messages) points to the line: `self.encoded_texts = [tokenizer.encode(text) for text in self.data["Text"]]`
- "Przycięcie sekwencji dłuższych niż max_length" (Truncating sequences longer than max_length) points to the line: `self.encoded_texts = [encoded_text[:self.max_length] for encoded_text in self.encoded_texts]`
- "Wypełnienie sekwencji w celu dopasowania do najdłuższej" (Padding the sequence to match the longest) points to the line: `self.encoded_texts = [encoded_text + [pad_token_id] * (self.max_length - len(encoded_text)) for encoded_text in self.encoded_texts]`

```

        ]
def __getitem__(self, index):
    encoded = self.encoded_texts[index]
    label = self.data.iloc[index]["Label"]
    return (
        torch.tensor(encoded, dtype=torch.long),
        torch.tensor(label, dtype=torch.long)
    )
def __len__(self):
    return len(self.data)

def _longest_encoded_length(self):
    max_length = 0
    for encoded_text in self.encoded_texts:
        encoded_length = len(encoded_text)
        if encoded_length > max_length:
            max_length = encoded_length
    return max_length

```

Klasa SpamDataset ładuje dane z utworzonych wcześniej plików CSV, tokenizuje tekst za pomocą tokenizera GPT-2 z biblioteki tiktoken, a następnie *wypełnia* lub *przyrina* sekwencje do jednolitej długości określonej przez najdłuższą sekwencję lub predefiniowaną maksymalną długość. Dzięki temu wszystkie tensory wejściowe mają ten sam rozmiar. Jest to konieczne do tworzenia partii w mechanizmie ładującym dane szkoleniowe, który zaimplementujemy w następnej kolejności:

```

train_dataset = SpamDataset(
    csv_file="train.csv",
    max_length=None,
    tokenizer=tokenizer
)

```

Długość najdłuższej sekwencji jest zapisana w atrybucie `max_length` zbioru danych. Jeśli chcesz się dowiedzieć, jaka jest liczba tokenów w najdłuższej sekwencji, możesz użyć następującego kodu:

```
print(train_dataset.max_length)
```

Kod wyświetla wartość 120, co pokazuje, że najdłuższa sekwencja zawiera nie więcej niż 120 tokenów. Jest to typowa długość dla wiadomości tekstowych. Biorąc pod uwagę limit długości kontekstu, model może obsługiwać sekwencje do 1024 tokenów. Jeśli zbiór danych zawiera dłuższe teksty, możesz ustawić podczas tworzenia szkoleniowego zbioru danych wartość `max_length=1024`. W ten sposób zyskasz pewność, że dane wejściowe nie przekroczą obsługiwanej przez model długości (kontekstu).

Następnie, aby dopasować długość najdłuższej sekwencji szkoleniowej, uzupełnimy zbiory walidacyjny i testowy. Co ważne, próbki zbioru walidacyjnego i testowego przekraczające długość najdłuższej próbki szkoleniowej są przycinane w kodzie zdefiniowanej wcześniej klasy `SpamDataset` z użyciem instrukcji `encoded_text[:self->.max_length]`. To przycięcie jest opcjonalne; jeśli w zbiorach walidacyjnym i testowym nie ma sekwencji przekraczających 1024 tokeny, możesz ustawić dla nich wartość `max_length=None`:

```
val_dataset = SpamDataset(  
    csv_file="validation.csv",  
    max_length=train_dataset.max_length,  
    tokenizer=tokenizer  
)  
test_dataset = SpamDataset(  
    csv_file="test.csv",  
    max_length=train_dataset.max_length,  
    tokenizer=tokenizer  
)
```

Ćwiczenie 6.1. Zwiększenie długości kontekstu

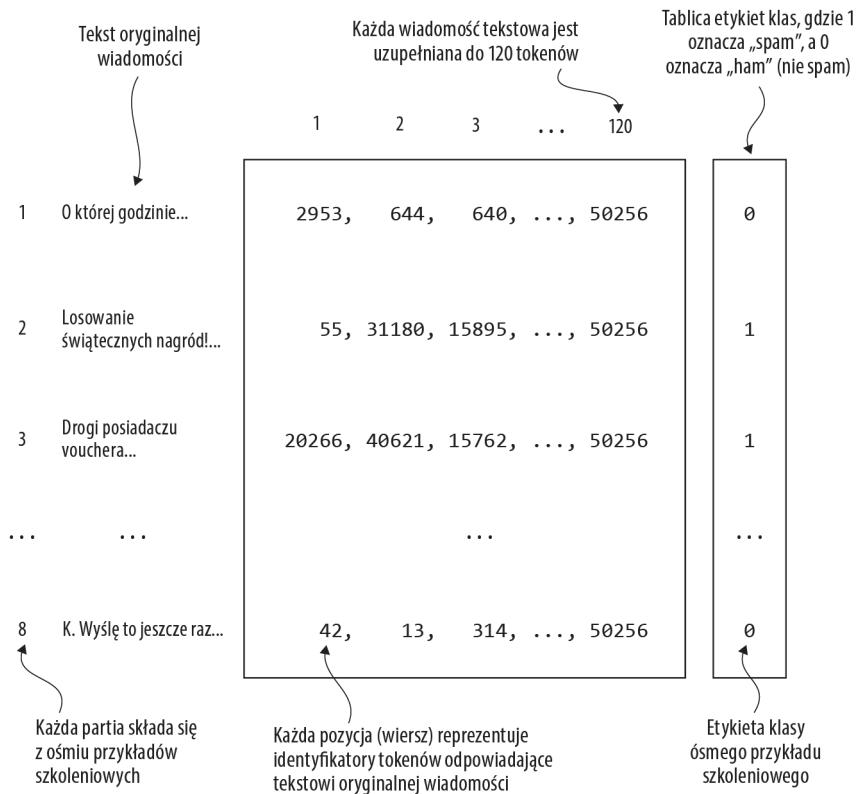
Uzupełnij dane wejściowe do maksymalnej liczby tokenów, jaką obsługuje model, i obserwuj, jaki to ma wpływ na wydajność prognozowania.

Używając zbiorów danych jako danych wejściowych, można, podobnie jak w pracy z danymi tekstowymi, utworzyć egzemplarz mechanizmu ładującego dane. Jednak w tym przypadku to cele reprezentują etykiety klas, a nie kolejne tokeny w tekście. Na przykład, jeśli wybierzesz rozmiar partii 8, każda partia będzie składać się z ośmiu przykładów szkoleniowych o długości 120 i odpowiednio etykiety klasy każdego przykłdu (rysunek 6.7).

Kod z listingu 6.5 tworzy obiekty ładujące dane zestawu szkoleniowego, walidacyjnego i testowego odpowiedzialne za ładowanie wiadomości tekstowych i etykiet w partiach o rozmiarze 8.

Listing 6.5. Tworzenie obiektów ładujących dane framework PyTorch

```
from torch.utils.data import DataLoader  
  
num_workers = 0      ←———— To ustawienie zapewnia zgodność z większością komputerów  
batch_size = 8  
torch.manual_seed(123)  
  
train_loader = DataLoader(  
    dataset=train_dataset,  
    batch_size=batch_size,  
    shuffle=True,  
    num_workers=num_workers,  
    drop_last=True,  
)  
val_loader = DataLoader(  
    dataset=val_dataset,  
    batch_size=batch_size,  
    num_workers=num_workers,  
    drop_last=False,  
)  
test_loader = DataLoader(  
    dataset=test_dataset,  
    batch_size=batch_size,  
    num_workers=num_workers,  
    drop_last=False,  
)
```



Rysunek 6.7. Pojedyncza partia szkoleniowa złożona z ośmiu wiadomości tekstowych reprezentowanych za pomocą identyfikatorów tokenów. Każda wiadomość tekstowa składa się ze 120 identyfikatorów tokenów. W tablicy etykiet klas jest zapisanych osiem etykiet klas odpowiadających wiadomościom tekstowym, które mogą mieć wartość 0 („nie spam”) lub 1 („spam”)

Aby się upewnić, że obiekty ładowające dane działają i faktycznie zwracają partie o oczekiwany rozmiarze, iterujemy po szkoleniowym obiekcie ładowającym dane, a następnie wyświetlamy wymiary tensora ostatniej partii:

```
for input_batch, target_batch in train_loader:
    pass
print("Wymiary partii wejściowej:", input_batch.shape)
print("Wymiary partii etykiet:", target_batch.shape)
```

Oto uzyskany wynik:

```
Wymiary partii wejściowej: torch.Size([8, 120])
Wymiary partii etykiet: torch.Size([8])
```

Jak można zauważyć, partie wejściowe, zgodnie z oczekiwaniemi, składają się z ośmiu przykładów szkoleniowych po 120 tokenów każdy. Tensor etykiet przechowuje etykiety klas odpowiadające ośmiu przykładom szkoleniowym.

Na koniec, aby uzyskać wyobrażenie o rozmiarze zbioru danych, wyświetlimy całkowitą liczbę partii w każdym zbiorze danych:

```
print(f"{len(train_loader)} partii szkoleniowych")
print(f"{len(val_loader)} partii walidacyjnych")
print(f"{len(test_loader)} partii testowych")
```

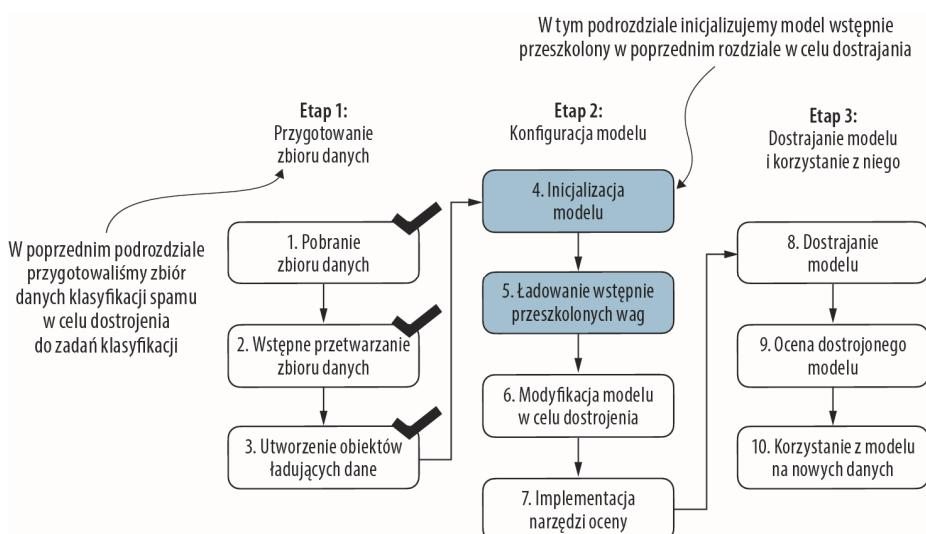
Liczby partii w każdym zbiorze wynoszą:

```
130 partii szkoleniowych
19 partii walidacyjnych
38 partii testowych
```

Po zakończeniu etapu przygotowania danych trzeba przygotować model do dostrajania.

6.4. Inicjalizacja modelu z użyciem wag wstępnie przeszkolonego modelu

Teraz trzeba przygotować model do dostrajania pod kątem klasyfikowania wiadomości spamowych. Zaczynamy od inicjalizacji wstępnie przeszkolonego modelu (rysunek 6.8).



Rysunek 6.8. Trzyetapowy proces dostrajania modelu LLM do zadań klasyfikacji. Po ukończeniu etapu 1., czyli przygotowaniu zbioru danych, trzeba zainicjować model LLM, który następnie dostroimy do zadań klasyfikacji wiadomości spamowych

Aby rozpocząć proces przygotowania modelu, stosujemy te same konfiguracje, których używaliśmy do wstępnego szkolenia nieoznaczonych danych:

```
CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"
```

```

BASE_CONFIG = {
    "vocab_size": 50257,           ← Rozmiar słownictwa
    "context_length": 1024,        ← Długość kontekstu
    "drop_rate": 0.0,             ← Wskaźnik dropout
    "qkv_bias": True             ← Stronniczość zapytanie klucz
}                                wartość (ang. query-key-value)
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

```

Następnie, aby załadować pobrane wagi do modelu GPT (listing 6.6), importujemy z modułu *gpt_download.py* funkcję `download_and_load_gpt2` i ponownie wykorzystujemy klasę `GPTModel` oraz funkcję `load_weights_into_gpt`, której używaliśmy do wstępniego szkolenia (patrz rozdział 5.).

Listing 6.6. Ładowanie wag wstępnie przeszkolonego modelu GPT

```

from gpt_download import download_and_load_gpt2
from chapter05 import GPTModel, load_weights_into_gpt

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()

```

Po załadowaniu wag modelu do obiektu klasy `GPTModel` ponownie wykorzystujemy funkcję narzędziową generowania tekstu znaną Ci już z rozdziałów 4. i 5., aby sprawdzić, czy model generuje spójny tekst:

```

from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"
token_ids = generate_text_simple(
    model=model,
    idxtext_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))

```

Poniższy wynik dowodzi, że model generuje spójny tekst, co wskazuje na to, że wagi modelu zostały załadowane poprawnie:

Every effort moves you forward.
The first step is to understand the importance of your work²

Przed rozpoczęciem dostrajania modelu jako klasyfikatora spamu sprawdźmy, czy model klasyfikuje wiadomości spamowe po przekazaniu mu wskazówek za pomocą promptu:

```
text_2 = (
    "Is the following text 'spam'? Answer with 'yes' or 'no':"
    " 'You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award.'"
)
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_2, tokenizer),
    max_new_tokens=23,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

Model odpowiada w następujący sposób:

```
Is the following text 'spam'? Answer with 'yes' or 'no': 'You are a winner you have
→been specially selected to receive $1000 cash or a $2000 award.'
The following text 'spam'? Answer with 'yes' or 'no': 'You are a winner'3
```

Na podstawie wyników można zauważyc, że model ma trudności z udzielaniem odpowiedzi według instrukcji. Można się było tego spodziewać, ponieważ został poddany jedynie wstępemu szkoleniu i brakuje mu dostrojenia do wykonywania instrukcji. Spróbujmy zatem przygotować model do dostrajania do zadań klasyfikacji.

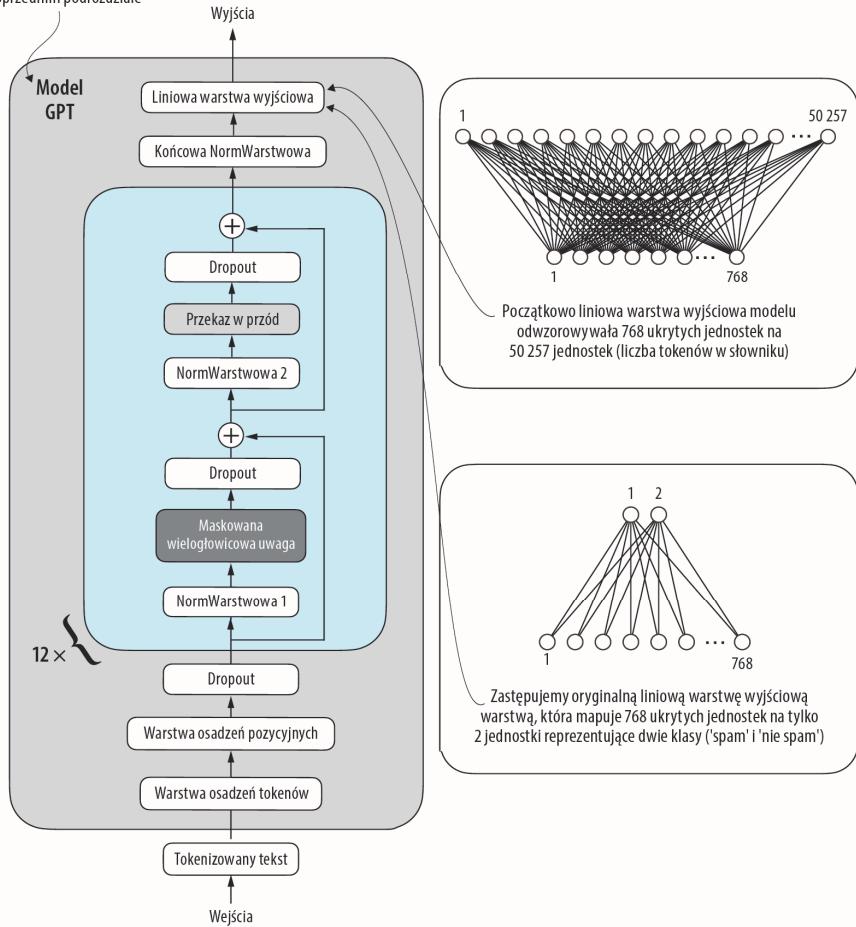
6.5. Dodawanie nagłówka klasyfikacji

Trzeba zmodyfikować wstępnie przeszkolony model LLM w celu przygotowania go do dostrajania do zadań klasyfikacji. Dlatego zastąpimy oryginalną warstwę wyjściową, która mapuje ukrytą reprezentację na słownictwo złożone z 50 257 tokenów, mniejszą warstwę wyjściową, która mapuje ją na dwie klasy: 0 („nie spam”) i 1 („spam”) – patrz rysunek 6.9. Użyjemy tego samego modelu, co wcześniej. Jedyna różnica to zastąpienie warstwy wyjściowej.

² Każdy wysiłek posuwa Cię do przodu. Pierwszym krokiem jest zrozumienie ważności Twojej pracy. — *przyp. tłum.*

³ Czy poniższy tekst jest spamem? Odpowiedz „tak” lub „nie”: „Zostałeś zwycięzcą... — *przyp. tłum.*

Model GPT, który zaimplementowałeś w rozdziale 5. i wczytałeś do pamięci w poprzednim podrozdziale



Rysunek 6.9. Dostosowanie modelu GPT do zadań klasyfikacji spamu przez zmodyfikowanie jego architektury. Początkowo liniowa warstwa wyjściowa modelu odwzorowywała 768 ukrytych jednostek na słownictwo składające się z 50 257 tokenów. Aby model wykrywać spam, zastępujemy tę warstwę nową warstwą wyjściową, która mapuje te same 768 ukrytych jednostek na dwie klasy, reprezentujące „spam” i „nie spam”

Węzły warstwy wyjściowej

Ponieważ mamy do czynienia z zadaniem klasyfikacji binarnej, moglibyśmy użyć pojedynczego węzła wyjściowego. Wymagałoby to jednak modyfikacji funkcji strat. Tematykę tę omówiłem w artykule *Losses Learned-Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch* (<https://mng.bz/NRZZ>). Z tego powodu wybrałem bardziej ogólne podejście, w którym liczba węzłów wyjściowych odpowiada liczbie klas. Na przykład w przypadku problemu klasyfikacji z trzema klasami, takiego jak klasyfikowanie artykułów informacyjnych jako „Technika”, „Sport” lub „Polityka”, należałoby użyć trzech węzłów wyjściowych.

Przed próbą wykonania modyfikacji pokazanej na rysunku 6.9 spróbujmy wyświetlić architekturę modelu za pomocą instrukcji `print(model)`:

```
GPTModel(  
    (tok_emb): Embedding(50257, 768)  
    (pos_emb): Embedding(1024, 768)  
    (drop_emb): Dropout(p=0.0, inplace=False)  
    (trf_blocks): Sequential(  
        ...  
        (11): TransformerBlock(  
            (att): MultiHeadAttention(  
                (W_query): Linear(in_features=768, out_features=768, bias=True)  
                (W_key): Linear(in_features=768, out_features=768, bias=True)  
                (W_value): Linear(in_features=768, out_features=768, bias=True)  
                (out_proj): Linear(in_features=768, out_features=768, bias=True)  
                (dropout): Dropout(p=0.0, inplace=False)  
            )  
            (ff): FeedForward(  
                (layers): Sequential(  
                    (0): Linear(in_features=768, out_features=3072, bias=True)  
                    (1): GELU()  
                    (2): Linear(in_features=3072, out_features=768, bias=True)  
                )  
            )  
            (norm1): LayerNorm()  
            (norm2): LayerNorm()  
            (drop_resid): Dropout(p=0.0, inplace=False)  
        )  
        (final_norm): LayerNorm()  
        (out_head): Linear(in_features=768, out_features=50257, bias=False)  
    )
```

Ta warstwa wyjściowa odpowiada architekturze, którą przedstawiłem w rozdziale 4. Jak wspomniałem wcześniej, klasa `GPTModel` składa się z warstw osadzeń, po których następuje 12 identycznych *bloków transformera* (dla związkowości na rysunku jest pokazany tylko ostatni blok), a potem końcowa warstwa `LayerNorm` i warstwa wyjściowa, `out_head`.

Następnie zastąpimy warstwę `out_head` nową warstwą wyjściową, którą będziemy dostrajać (rysunek 6.9).

Dostrajanie wybranych warstw a dostrajanie wszystkich warstw

Ponieważ zaczynamy od wstępnie przeszkolonego modelu, nie ma konieczności dostrajania wszystkich warstw modelu. W modelach językowych opartych na sieciach neuronowych niższe warstwy zwykle odpowiadają za podstawowe struktury językowe oraz semantykę mającą zastosowanie w szerokim zakresie zadań i zbiorów danych. Z tego powodu, aby dostosować model do nowych zadań, często wystarczy dostrojenie jedynie ostatnich warstw (tzn. warstw w pobliżu wyjścia), ponieważ odpowiadają one za bardziej złożone wzorce językowe i cechy specyficzne dla zadania. Dodatkową zaletą takiego podejścia jest większa wydajność obliczeniowa — dostrajanie tylko niewielkiej liczby warstw wymaga mniej zasobów. Czytelnicy zainteresowani szczegółami, w tym eksperymentami dotyczącymi wyboru warstw do dostrajania, znajdą więcej informacji w „Dodatku B”.

Aby przygotować model do dostrajania pod kątem zadań klasyfikacji, najpierw należy zamrozić model, co oznacza, że oznaczamy wszystkie warstwy jako nietrenowalne:

```
for param in model.parameters():
    param.requires_grad = False
```

Następnie zastępujemy warstwę wyjściową (`model.out_head`), która pierwotnie mapowała wejścia warstwy na 50 257 wymiarów, czyli rozmiar słownika (patrz rysunek 6.9) (listing 6.7).

Listing 6.7. Dodawanie warstwy klasyfikacji

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(
    in_features=BASE_CONFIG["emb_dim"],
    out_features=num_classes
)
```

Aby kod był bardziej uniwersalny, używamy ustawienia `BASE_CONFIG["emb_dim"]`, które w modelu "gpt2-small (124M)" ma wartość 768. Dzięki temu tego samego kodu można użyć także do pracy z większymi wariantami modelu GPT-2.

Teraz warstwa wyjściowa `model.out_head` ma domyślnie ustawiony atrybut `requires_grad` na `True`, co oznacza, że jest to jedyna warstwa w modelu, która będzie aktualizowana podczas szkolenia. Ogólnie rzecz biorąc, wystarczy szkolenie dodanej przed chwilą warstwy wyjściowej. Jak jednak odkryłem w trakcie eksperymentów, dostrojenie dodatkowych warstw zauważalnie poprawia wydajność predykcyjną modelu (więcej informacji można znaleźć w „Dodatku B”). Konfigurujemy również ostatni blok transformera i końcowy moduł `LayerNorm`, łączący ten blok z warstwą wyjściową, tak aby można było go szkolić (rysunek 6.10).

Aby końcowy blok `LayerNorm` i ostatni blok transformera były trenowalne, ustawiamy ich właściwości `require_grad` na `True`:

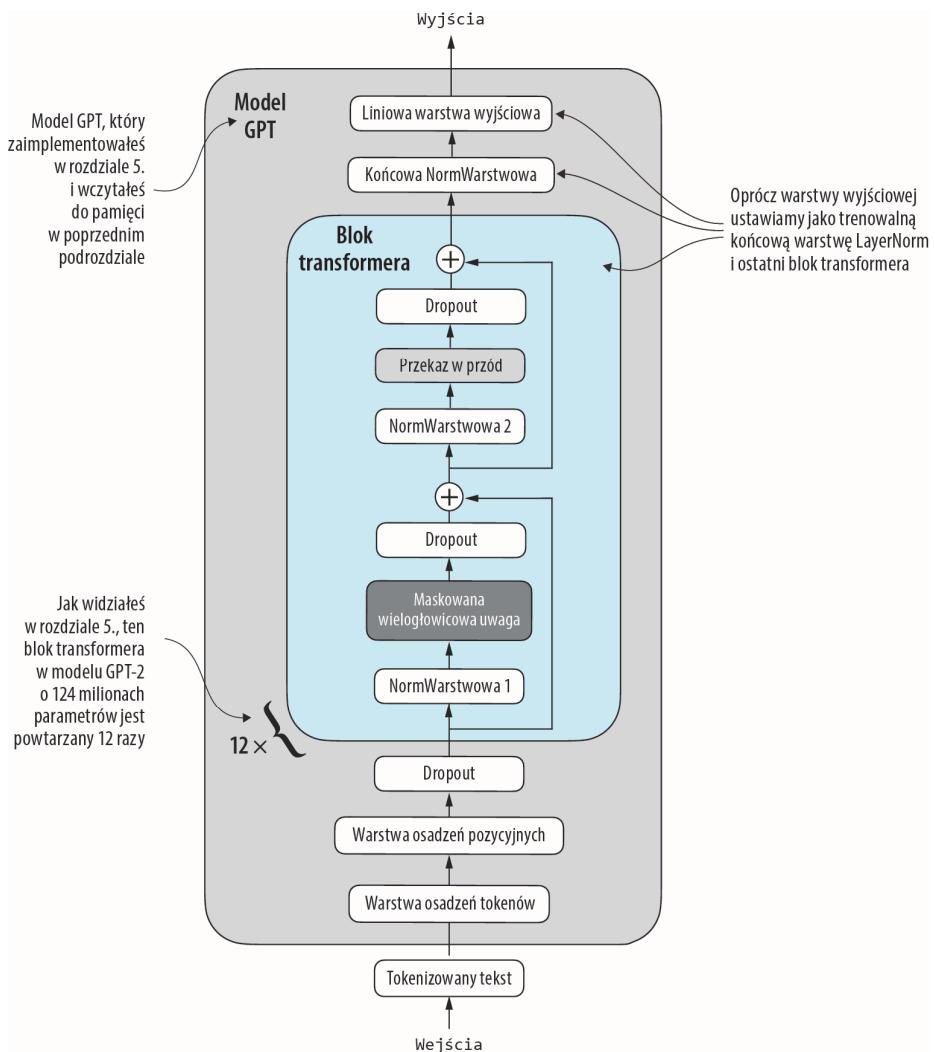
```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

Ćwiczenie 6.2. Dostrajanie całego modelu

Trzeba dostroić nie tylko końcowy blok transformera, ale cały model, a następnie ocenić wpływ tego dostrajania na wydajność predykcyjną.

Mimo że dodaliśmy nową warstwę wyjściową i oznaczyliśmy niektóre warstwy jako trenowalne bądź nietrenowalne, nadal możemy korzystać z tego modelu podobnie jak poprzednio.

Na przykład możemy przekazać modelowi przykładowy tekst identyczny z tekstem używanym wcześniej:



Rysunek 6.10. Model GPT zawiera 12 powtórzonych bloków transformera. Oprócz warstwy wyjściowej ustawiamy jako trenowalne końcową warstwę LayerNorm i ostatni blok transformera. Reszta z 11 bloków transformera i warstwy osadzania pozostaje nietrenowalna

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)
print("Wejścia:", inputs)
print("Wymiary wejśc:", inputs.shape) ← ksztalt (batch_size, num_tokens)
```

Na podstawie wyników działania tego kodu można stwierdzić, że powyższy kod koduje dane wejściowe do tensora składającego się z czterech tokenów wejściowych:

```
Wejścia: tensor([[5211, 345, 423, 640]])
Wymiary wejśc: torch.Size([1, 4])
```

Teraz możemy w zwykły sposób przekazać do modelu zakodowane identyfikatory tokenów:

```
with torch.no_grad():
    outputs = model(inputs)
print("Wyjścia:\n", outputs)
print("Wymiary wyjścia:", outputs.shape)
```

Tensor wyjściowy wygląda następująco:

```
Wyjścia:
tensor([[[[-1.5854,  0.9904],
          [-3.7235,  7.4548],
          [-2.2661,  6.6049],
          [-3.5983,  3.9902]]]])
Wymiary wyjścia: torch.Size([1, 4, 2])
```

Wcześniej podobne wejście spowodowałoby wygenerowanie tensora wyjściowego [1, 4, 50257], gdzie 50257 oznaczało rozmiar słownika. Liczba wierszy wyjściowych odpowiada liczbie tokenów wejściowych (w tym przypadku cztery). Ponieważ jednak zastąpiliśmy warstwę wyjściową modelu, wymiar osadzeń każdego wyjścia (liczba kolumn) wynosi teraz 2 zamiast 50 257.

Pamiętaj, że chcemy dostroić ten model w taki sposób, aby zwracał etykietę klasy decydującą o tym, czy wiadomość przekazana na wejściu modelu jest „spamem”, czy też to „nie jest spam”. Nie ma potrzeby dostrajania wszystkich czterech wierszy wyjściowych; wystarczy skupić się na jednym tokenie wyjściowym. W szczególności, jak pokazałem na rysunku 6.11, skupimy się na ostatnim wierszu, odpowiadającym ostatniemu tokenowi wyjściowemu.

Aby z tensora wyjściowego wyodrębnić ostatni token wyjściowy, można użyć następującego kodu:

```
print("Ostatni token wyjściowy:", outputs[:, -1, :])
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

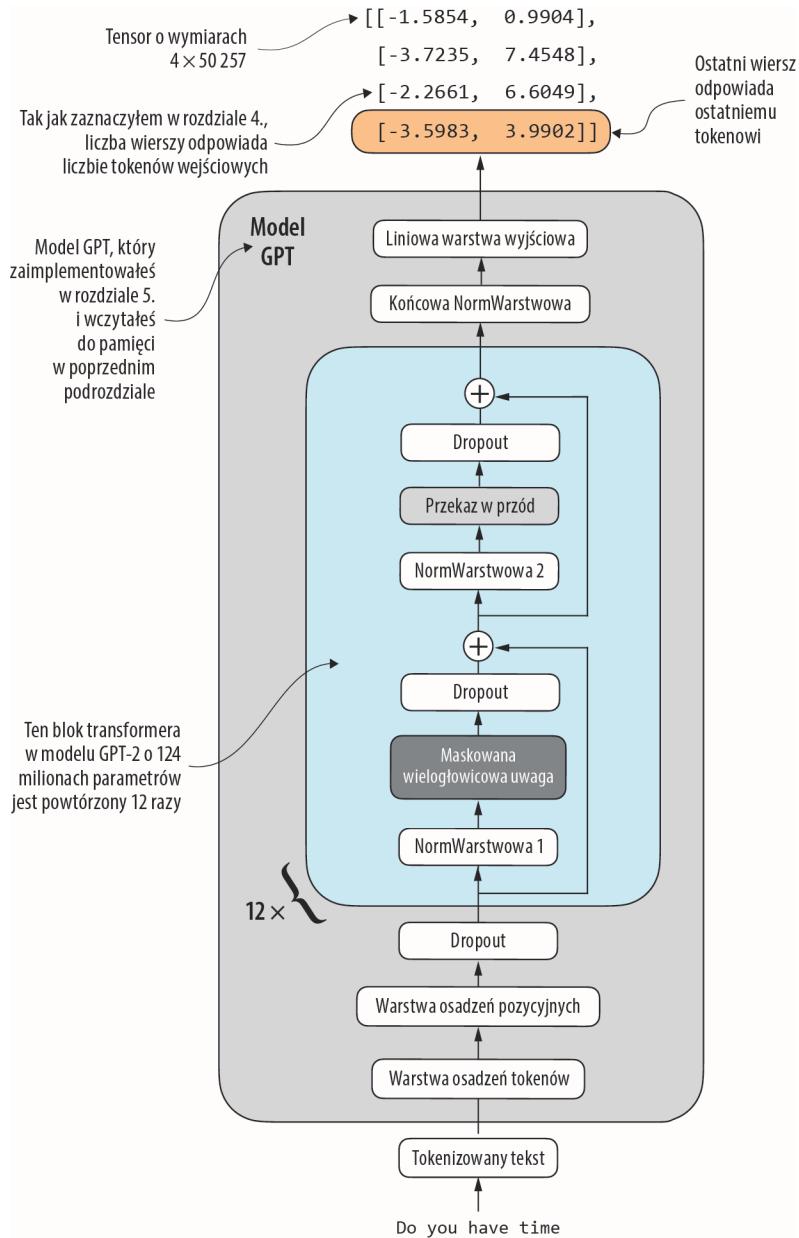
```
Ostatni token wyjściowy: tensor([-3.5983,  3.9902])
```

Trzeba jeszcze przekonwertować wartości na prognozy klas, czyli etykiety. Najpierw jednak spróbuję wyjaśnić, dlaczego interesuje nas tylko ostatni token wyjściowy.

Wcześniej omówiłem mechanizm uwagi, ustanawiający relację między każdym tokenem wejściowym a każdym innym tokenem wejściowym, oraz pojęcie *przyczynowej maski uwagi*, powszechnie stosowanej w modelach podobnych do GPT (patrz rozdział 3.).

Ta maska ogranicza fokus tokenu do jego bieżącej pozycji oraz tych, które są przed nim. Dzięki temu na każdy token może mieć wpływ tylko on sam oraz tokeny, które go poprzedzają (rysunek 6.12).

W przypadku maski przyczynowej o konfiguracji pokazanej na rysunku 6.12 ostatni token w sekwencji gromadzi najwięcej informacji, ponieważ jest to jedyny token mający dostęp do danych ze wszystkich poprzednich tokenów. Z tego powodu w przykładowym



Rysunek 6.11. Model GPT z przykładowym wejściem i wyjściem złożonymi z czterech tokenów. Ze względu na zmodyfikowaną warstwę wyjściową tensor wyjściowy składa się z dwóch kolumn. Podczas dostrajania modelu do klasyfikacji spamu interesuje nas tylko ostatni wiersz, odpowiadający ostatniemu tokenowi



Rysunek 6.12. Mechanizm uwagi przyczynowej, w którym oceny uwagi między tokenami wejściowymi są przedstawione w formie macierzy. Puste komórki wskazują pozycje zamaskowane przez maskę uwagi przyczynowej, która zapobiega uwzględnianiu przyszłych tokenów. Wartości w komórkach reprezentują współczynniki uwagi; ostatni token, „time”, jest jedynym, który oblicza współczynniki uwagi dla wszystkich wcześniejszych tokenów

zadaniu klasyfikacji spamu podczas procesu dostrajania skupiamy się na tym ostatnim tokenie.

Teraz jesteś gotowy do przekształcenia ostatniego tokena w prognozę etykiety klasy oraz obliczenia początkowej dokładności prognozy modelu. Następnie dostroimy model do zadania klasyfikacji spamu.

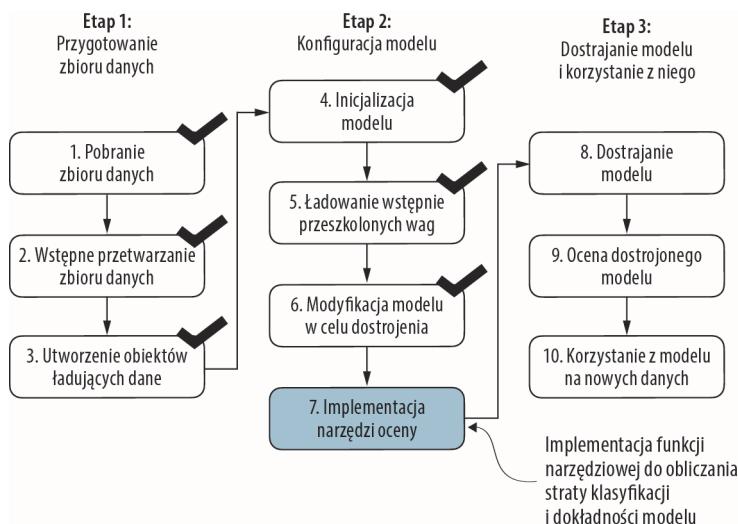
Ćwiczenie 6.3. Dostrajanie pierwszego i ostatniego tokena

Spróbuj dostroić pierwszy token wyjściowy. Zwróć uwagę na zmiany w wydajności predykcyjnej w porównaniu z dostrajaniem ostatniego tokena wyjściowego.

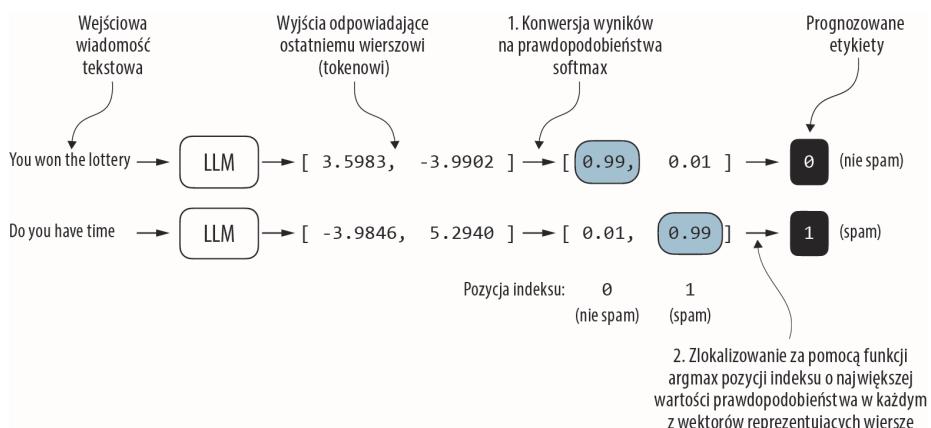
6.6. Obliczanie straty i dokładności klasyfikacji

Przed przystąpieniem do dostrajania modelu pozostaje jedno proste zadanie: trzeba zaimplementować funkcje oceny modelu używane podczas dostrajania (rysunek 6.13).

Zanim przystąpię do implementacji narzędzi oceny, spróbuję krótko opisać sposób konwersji wyjść modelu na prognozy etykiet klas. Identyfikator następnego tokena wygenerowanego przez LLM został obliczony wcześniej. W tym celu dokonaliśmy konwersji 50 257 współczynników na prawdopodobieństwa za pomocą funkcji softmax, a następnie — z użyciem funkcji argmax — zwróciliśmy pozycję najwyższej prawdopodobieństwa. To samo podejście zastosujemy w celu obliczenia, czy dla danego wejścia model generuje prognozę „spam” czy „nie spam” (rysunek 6.14). Jedyna różnica to wymiar wyjścia, który teraz wynosi 2, a nie 50 257.



Rysunek 6.13. Trzyetapowy proces dostrajania modelu LLM do zadań klasyfikacyjnych. Wykonaliśmy pierwsze sześć kroków. Możemy teraz wykonać ostatni krok etapu 2. — implementację funkcji do oceny wydajności modelu w celu klasyfikowania wiadomości spamowych przed dostrajaniem, w trakcie dostrajania i po nim



Rysunek 6.14. Dla każdego tekstu wejściowego wyjście modelu odpowiadające ostatniemu tokenowi są konwertowane na współczynniki prawdopodobieństwa. W celu znalezienia etykiet klas wyszukiwana jest pozycja indeksu o najwyższym współczynniku prawdopodobieństwa. Model prognozuje etykiety spamu niepoprawnie, ponieważ jeszcze nie został przeszkołony

Rozważmy ostatni token wyjściowy na konkretnym przykładzie:

```
print("Ostatni token wyjściowy:", outputs[:, -1, :])
```

Wartości tensora odpowiadającego ostatniemu tokenowi to:

```
Ostatni token wyjściowy: tensor([-3.5983,  3.9902])
```

Aby uzyskać etykietę klasy, można użyć następującego kodu:

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Etykieta klasy:", label.item())
```

W tym przypadku kod zwraca 1, co oznacza, że zgodnie z prognozą modelu wejściowy tekst to „spam”. Użycie funkcji softmax jest w tym przypadku opcjonalne, ponieważ największe wartości wyjść bezpośrednio odpowiadają najwyższym współczynnikom prawdopodobieństwa. Można więc uprościć kod bez korzystania z funkcji softmax:

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("Etykieta klasy:", label.item())
```

To pojęcie można wykorzystać do obliczenia dokładności klasyfikacji, która mierzy procent poprawnych prognoz w całym zbiorze danych.

Aby określić dokładność klasyfikacji, stosujemy do wszystkich przykładów w zbiorze danych kod predykcji oparty na funkcji argmax i obliczamy odsetek poprawnych prognoz. Do tego celu definiujemy funkcję calc_accuracy_loader (listing 6.8).

Listing 6.8. Obliczanie dokładności klasyfikacji

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)

            with torch.no_grad():
                logits = model(input_batch)[:, -1, :] ← Logity ostatniego tokena wyjściowego
                predicted_labels = torch.argmax(logits, dim=-1)

            num_examples += predicted_labels.shape[0]
            correct_predictions += (
                (predicted_labels == target_batch).sum().item()
            )
        else:
            break
    return correct_predictions / num_examples
```

Spróbujmy użyć tej funkcji do określenia dokładności klasyfikacji w różnych zbiorach danych oszacowanych, w celu poprawy wydajności, na podstawie 10 partii:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

torch.manual_seed(123)
```

```

train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Dokładność zbioru szkoleniowego: {train_accuracy*100:.2f}%")
print(f"Dokładność zbioru walidacyjnego: {val_accuracy*100:.2f}%")
print(f"Dokładność zbioru testowego: {test_accuracy*100:.2f}%")

```

Ustawienie `device` decyduje o układzie, na którym działa model. Jeśli jest dostępny układ GPU z obsługą Nvidia CUDA, model automatycznie działa na układzie GPU, a w przeciwnym razie działa na CPU. Oto uzyskany wynik:

```

Dokładność zbioru szkoleniowego: 46.25%
Dokładność zbioru walidacyjnego: 45.00%
Dokładność zbioru testowego: 48.75%

```

Jak można zauważyć, dokładność prognoz jest bliska predykcji losowej, która w tym przypadku wynosiłaby 50%. Aby poprawić dokładność prognozowania, trzeba dostroić model.

Przed przystąpieniem do dostrajania trzeba jednak zdefiniować funkcję `strat`, którą będziemy optymalizować podczas szkolenia. Celem jest maksymalizacja dokładności klasyfikowania spamu przez model. Dążymy do tego, aby zamieszczony powyżej kod generował poprawne etykiety klas: 0 dla wiadomości niebędącej spamem i 1 dla spamu.

Ponieważ dokładność klasyfikacji nie jest funkcją różniczkowalną, do maksymalizowania dokładności w roli zamiennika używamy straty entropii krzyżowej. W związku z tym funkcja `calc_loss_batch` pozostaje bez większych zmian poza jedną: skupiamy się na optymalizacji tylko ostatniego tokena, `model(input_batch)[:, -1, :]`, zamiast wszystkich tokenów, `model(input_batch)`:

```

def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)
    logits = model(input_batch)[:, -1, :]           ← Logity ostatniego
    loss = torch.nn.functional.cross_entropy(logits, target_batch) ← tokena wyjściowego
    return loss

```

Aby obliczyć stratę dla pojedynczej partii uzyskanej za pomocą zdefiniowanych wcześniej mechanizmów ładujących dane, używamy funkcji `calc_loss_batch`. Aby obliczyć stratę dla wszystkich partii generowanych przez mechanizm ładujący dane, tak jak poprzednio definiujemy funkcję `calc_loss_loader` (listing 6.9).

Listing 6.9. Obliczanie straty klasyfikacji

```

def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:

```

```

        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches

```

Sprawdzenie, czy liczba partii nie przekracza liczby partii uzyskanych przez komponent ładujący dane

Podobnie jak w przypadku obliczania dokładności szkolenia, teraz obliczamy początkową stratę dla każdego zbioru danych:

```

with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)
print(f"Strata zbioru szkoleniowego: {train_loss:.3f}")
print(f"Strata zbioru walidacyjnego: {val_loss:.3f}")
print(f"Strata zbioru testowego: {test_loss:.3f}")

```

Wyłączenie śledzenia gradientu w celu poprawy wydajności, ponieważ jeszcze nie szkolimy modelu

Oto początkowe wartości strat:

```

Strata zbioru szkoleniowego: 2.453
Strata zbioru walidacyjnego: 2.583
Strata zbioru testowego: 2.322

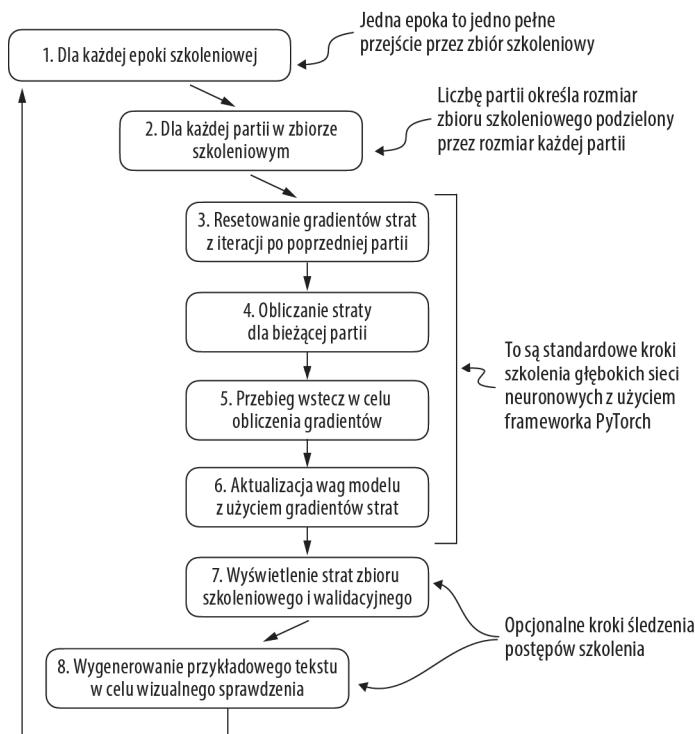
```

W kolejnym kroku w celu dostrojenia modelu, czyli zminimalizowania straty zbioru szkoleniowego, zaimplementujemy funkcję szkoleniową. Zminimalizowanie straty zbioru szkoleniowego pomoże zwiększyć dokładność klasyfikacji, co jest ogólnym celem.

6.7. Dostrajanie modelu na danych nadzorowanych

Trzeba zdefiniować funkcję szkoleniową i jej użyć, aby dostroić wstępnie przeszkolony model LLM i poprawić jego dokładność w klasyfikowaniu spamu. Pętla szkoleniowa, którą pokazałem na rysunku 6.15, jest taka sama jak ta, której używaliśmy do wstępniego szkolenia. Jedyna różnica polega na tym, że teraz, zamiast generować przykładowy tekst do oceny modelu, obliczamy dokładność klasyfikacji.

Funkcja szkoleniowa implementująca pojęcia pokazane na rysunku 6.15 równieżściśle odzwierciedla funkcję `train_model_simple`, której używaliśmy do wstępniego szkolenia modelu. Jedyne dwie różnice polegają na tym, że teraz śledzimy liczbę zaobser-



Rysunek 6.15. Typowa pętla szkoleniowa głębowych sieci neuronowych w PyTorch składa się z kilku kroków polegających na iterowaniu po partiach zbioru szkoleniowego przez kilka epok. W każdej pętli obliczamy straṭy dla każdej partii zbioru szkoleniowego, aby określić gradienty straṭ, które następnie są wykorzystywane do aktualizacji wag modelu w celu zminimalizowania straṭu zbioru szkoleniowego

wowanych przykładów szkoleniowych (`examples_seen`) zamiast liczby tokenów, a po każdej epoce dokładność, zamiast wyświetlać przykładowy tekst (listing 6.10).

Listing 6.10. Dostrajanie modelu do klasyfikacji spamu

```
def train_classifier_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs, eval_freq, eval_iter):
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    examples_seen, global_step = 0, -1

    for epoch in range(num_epochs):                                ← Główna pętla szkoleniowa
        model.train()                                              ← Ustawienie modelu w tryb szkolenia

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()                                     ← Zresetowanie gradientów straṭ
            loss = calc_loss_batch(                                    ← z iteracji po poprzedniej partii
                input_batch, target_batch, model, device
            )
```

Inicjalizacja list do śledzenia straṭ i zaobserwowanych przykładów

```

loss.backward()
optimizer.step()
examples_seen += input_batch.shape[0]
global_step += 1
    ← Obliczenie gradientów strat
    ← Aktualizacja wag modelu
    | z użyciem gradientów strat
    | Nowość: śledzenie przykładów
    | zamiast tokenów
→ Opcjonalny
→ krok oceny
if global_step % eval_freq == 0:
    train_loss, val_loss = evaluate_model(
        model, train_loader, val_loader, device, eval_iter)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    print(f"Epoka {epoch+1} (krok {global_step:06d}): "
        f"Strata zbioru szkoleniowego {train_loss:.3f}, "
        f"Strata zbioru walidacyjnego {val_loss:.3f}"
    )

train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=eval_iter
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=eval_iter
)
    ← Obliczanie dokładności po każdej epoce
print(f"Dokładność zbioru szkoleniowego: {train_accuracy*100:.2f}% | ", end="")
print(f"Dokładność zbioru walidacyjnego: {val_accuracy*100:.2f}%")
train_accs.append(train_accuracy)
val_accs.append(val_accuracy)

return train_losses, val_losses, train_accs, val_accs, examples_seen

```

Funkcja `evaluate_model` jest identyczna z tą, której używaliśmy do wstępnego szkolenia:

```

def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss

```

Następnie inicjalizujemy optymalizator, ustaviamy liczbę epok szkoleniowych i rozpoczynamy szkolenie za pomocą funkcji `train_classifier_simple`. Szkolenie na laptopie M3 MacBook Air trwa około 6 minut, a na komputerze z układem GPU V100 lub A100 mniej niż pół minuty:

```

import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)
num_epochs = 5

train_losses, val_losses, train_accs, val_accs, examples_seen = \

```

```
train_classifier_simple(  
    model, train_loader, val_loader, optimizer, device,  
    num_epochs=num_epochs, eval_freq=50,  
    eval_iter=5  
)  
  
end_time = time.time()  
execution_time_minutes = (end_time - start_time) / 60  
print(f"Szkolenie zakończono. Czas szkolenia wyniósł {execution_time_minutes:.2f} minut.")
```

Oto wyniki, które obserwujemy podczas szkolenia:

```
Epoka 1 (krok 000000): Strata zbioru szkoleniowego 2.153, Strata zbioru walidacyjnego  
→ 2.392  
Epoka 1 (krok 000050): Strata zbioru szkoleniowego 0.617, Strata zbioru walidacyjnego  
→ 0.637  
Epoka 1 (krok 000100): Strata zbioru szkoleniowego 0.523, Strata zbioru walidacyjnego  
→ 0.557  
Dokładność zbioru szkoleniowego: 70.00% | Dokładność zbioru walidacyjnego: 72.50%  
Epoka 2 (krok 000150): Strata zbioru szkoleniowego 0.561, Strata zbioru walidacyjnego  
→ 0.489  
Epoka 2 (krok 000200): Strata zbioru szkoleniowego 0.419, Strata zbioru walidacyjnego  
→ 0.397  
Epoka 2 (krok 000250): Strata zbioru szkoleniowego 0.409, Strata zbioru walidacyjnego  
→ 0.353  
Dokładność zbioru szkoleniowego: 82.50% | Dokładność zbioru walidacyjnego: 85.00%  
Epoka 3 (krok 000300): Strata zbioru szkoleniowego 0.333, Strata zbioru walidacyjnego  
→ 0.320  
Epoka 3 (krok 000350): Strata zbioru szkoleniowego 0.340, Strata zbioru walidacyjnego  
→ 0.306  
Dokładność zbioru szkoleniowego: 90.00% | Dokładność zbioru walidacyjnego: 90.00%  
Epoka 4 (krok 000400): Strata zbioru szkoleniowego 0.136, Strata zbioru walidacyjnego  
→ 0.200  
Epoka 4 (krok 000450): Strata zbioru szkoleniowego 0.153, Strata zbioru walidacyjnego  
→ 0.132  
Epoka 4 (krok 000500): Strata zbioru szkoleniowego 0.222, Strata zbioru walidacyjnego  
→ 0.137  
Dokładność zbioru szkoleniowego: 100.00% | Dokładność zbioru walidacyjnego: 97.50%  
Epoka 5 (krok 000550): Strata zbioru szkoleniowego 0.207, Strata zbioru walidacyjnego  
→ 0.143  
Epoka 5 (krok 000600): Strata zbioru szkoleniowego 0.083, Strata zbioru walidacyjnego  
→ 0.074  
Dokładność zbioru szkoleniowego: 100.00% | Dokładność zbioru walidacyjnego: 97.50%  
Szkolenie zakończono. Czas szkolenia wyniósł 5.52 minut.
```

Następnie możemy użyć biblioteki Matplotlib w celu wykreślenia funkcji strat dla zbiorów szkoleniowego i walidacyjnego (listing 6.11).

Listing 6.11. Wykres strat dla zadania klasyfikacji

```
import matplotlib.pyplot as plt  
  
def plot_values(  
    epochs_seen, examples_seen, train_values, val_values,  
    label="loss"):
```

```

fig, ax1 = plt.subplots(figsize=(5, 3))           Wykresy strat zbiorów szkoleniowego
                                                    i walidacyjnego w poszczególnych epokach
                                                    ←
ax1.plot(epochs_seen, train_values, label=f"Szkolenie {label}")
ax1.plot(
    epochs_seen, val_values, linestyle="--",
    label=f"Walidacja {label}"
)
ax1.set_xlabel("Epoki")
ax1.set_ylabel(label.capitalize())
ax1.legend()                                     ← Utworzenie drugiej osi x dla obserwowanych przykładów

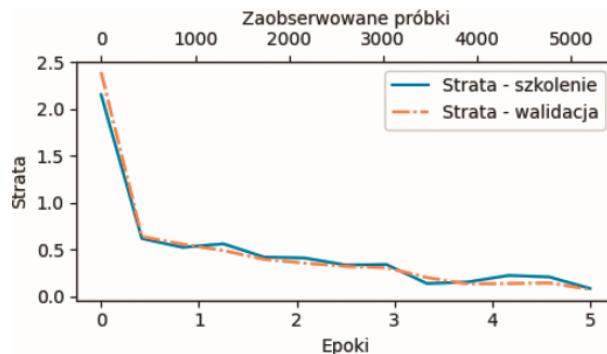
ax2 = ax1.twiny()                                ← Niewidoczny wykres w celu
ax2.plot(examples_seen, train_values, alpha=0)   wyrównywania punktów
ax2.set_xlabel("Zaobserwowane próbki")           ←

fig.tight_layout()                               ← Dostosowanie układu w celu zarządzania miejscem
plt.savefig(f"{label}-plot.pdf")                ←
plt.show()                                       ←

```

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))
plot_values(epochs_tensor, examples_seen_tensor, train_losses, val_losses)

Wynikowe krzywe strat zaprezentowałem na rysunku 6.16.



Rysunek 6.16. Straty zbiorów szkoleniowego i walidacyjnego modelu w pięciu epokach szkoleniowych. Zarówno strata zbioru szkoleniowego, reprezentowana przez linię ciągłą, jak i strata zbioru walidacyjnego, reprezentowana przez linię przerywaną, gwałtownie spadają w pierwszej epoce i stopniowo stabilizują się w piątej epoce. Ten wzorzec wskazuje na dobre postępy w nauce i sugeruje, że model nauczył się z danych szkoleniowych, a zarazem został dobrze uogólniony na niewidoczne dane walidacyjne

Jak widać na podstawie ostrego nachylenia wykresu w dół na rysunku 6.16, model dobrze uczy się na podstawie danych szkoleniowych i nie wykazuje prawie żadnych oznak nadmiernego dopasowania — oznacza to, że nie ma zauważalnej lukie między stratami zbioru szkoleniowego a stratami zbioru walidacyjnego.

Wybór liczby epok

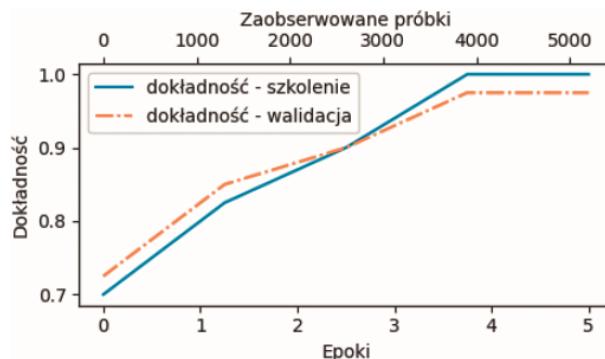
Kiedy rozpoczęliśmy szkolenie, ustawiliśmy liczbę epok na pięć. Liczba epok zależy od zbioru danych i trudności zadania. Nie ma tu uniwersalnego rozwiązania ani zalecenia, ale ustawienie liczby epok na pięć zwykle jest dobrym punktem wyjścia. Jeśli po kilku pierwszych epokach model nadmiernie się dopasowuje, co widać na wykresie zmian funkcji strat (patrz rysunek 6.16), być może trzeba zmniejszyć liczbę epok. Z kolei jeśli z wykresu trendu wynika, że strata walidacji można poprawić przez dalsze szkolenie, należy zwiększyć liczbę epok. W tym konkretnym przypadku pięć epok to rozsądna liczba, ponieważ model nie wykazuje oznak wcześniego nadmiernego dopasowania, a strata zbioru walidacyjnego jest bliska zera.

Za pomocą tej samej funkcji `plot_values` spróbujemy teraz wykreślić dokładność klasyfikacji:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(
    epochs_tensor, examples_seen_tensor, train_accs, val_accs,
    label="dokładność"
)
```

Wykres dokładności zamieściłem na rysunku 6.17. Po 4. i 5. epoce model osiąga stosunkowo wysoką dokładność zbioru szkoleniowego i walidacyjnego.



Rysunek 6.17. Zarówno dokładność zbioru szkoleniowego (linia ciągła), jak i dokładność zbioru walidacyjnego (linia przerywana) znacznie wzrastają we wczesnych epokach, a następnie osiągają niemal idealną dokładność 1.0. Bliskość tych dwóch linii w epokach sugeruje, że model nie jest nadmiernie dopasowany do danych szkoleniowych

Co ważne, korzystając wcześniej z funkcji `train_classifier_simple`, ustawiliśmy `eval_iter=5`. To oznacza, że oszacowania wydajności szkolenia i walidacji opierają się tylko na pięciu partiach.

Teraz trzeba obliczyć wskaźniki wydajności dla zbiorów szkoleniowego, walidacyjnego i testowego w całym zbiorze danych. W tym celu skorzystamy z poniższego kodu, tym razem bez definiowania wartości `eval_iter`:

```

train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Dokładność zbioru szkoleniowego: {train_accuracy*100:.2f}%)")
print(f"Dokładność zbioru walidacyjnego: {val_accuracy*100:.2f}%)")
print(f"Dokładność zbioru testowego: {test_accuracy*100:.2f}%)")

```

Oto uzyskane wartości dokładności:

```

Dokładność zbioru szkoleniowego: 97.21%
Dokładność zbioru walidacyjnego: 97.32%
Dokładność zbioru testowego: 95.67%

```

Wyniki dla zbioru szkoleniowego i testowego są niemal identyczne. Niewielka rozbieżność między dokładnością zbiorów szkoleniowego i testowego sugeruje minimalne niedopasowanie danych szkoleniowych. Dokładność zbioru walidacyjnego zazwyczaj jest nieco wyższa niż dokładność zbioru testowego. Wynika to stąd, że tworzenie modelu często obejmuje dostrajanie hiperparametrów pod kątem odpowiedniego działania na zbiorze walidacyjnym. Może to skutkować niedostatecznie dobrym uogólnieniem na zbiór testowy. Sytuacja ta jest powszechna, ale rozbieżność można potencjalnie zminimalizować przez dostosowanie w konfiguracji optymalizatora ustawień modelu, na przykład zwiększenie współczynnika dropoutu (`drop_rate`) lub parametru `weight_decay`.

6.8. Wykorzystanie modelu LLM jako klasyfikatora spamu

Po dostrojeniu i przeprowadzeniu oceny modelu możemy przystąpić do klasyfikowania wiadomości spamowych (rysunek 6.18). Spróbujmy skorzystać z dostrojonego modelu klasyfikacji spamu opartego na GPT. Poniższa funkcja `classify_review` realizuje etapy wstępnego przetwarzania danych przypominające te, których wcześniej użyliśmy w klasie `SpamDataset` (listing 6.12). Następnie, po przetworzeniu tekstu na identyfikatory tokenów, funkcja zastosuje model do prognozowania całkowitoliczbowej etykiety klasy, podobnej do tej, którą zaimplementowaliśmy w podrozdziale 6.6, a następnie zwróci właściwą nazwę klasy.

Listing 6.12. Wykorzystanie modelu do klasyfikowania nowych tekstów

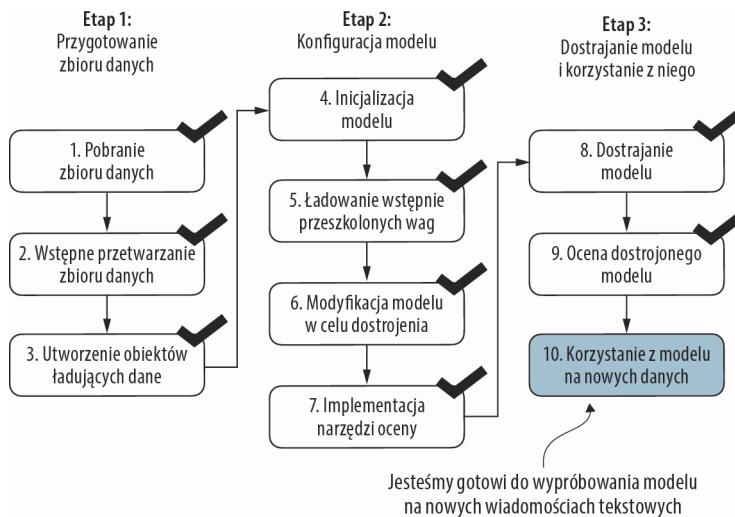
```

def classify_review(
    text, model, tokenizer, device, max_length=None,
    pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]

```





Rysunek 6.18. Trzyetapowy proces dostrajania modelu LLM do zadań klasyfikacji.

Krok 10. jest ostatnim krokiem etapu 3. — polega na wykorzystaniu dostrojonego modelu do klasyfikowania nowych wiadomości spamowych

```

input_ids = input_ids[:min(
    max_length, supported_context_length
)]
input_ids += [pad_token_id] * (max_length - len(input_ids)) ← Wypełnienie sekwencji
                                                               do najdłuższej

input_tensor = torch.tensor(
    input_ids, device=device
).unsqueeze(0) ← Dodanie wymiaru partii

with torch.no_grad():
    logits = model(input_tensor)[:, -1, :] ← Wnioskowanie modelu bez śledzenia gradientu
predicted_label = torch.argmax(logits, dim=-1).item() ← Logity ostatniego
                                                               tokena wyjściowego

return "spam" if predicted_label == 1 else "not spam" ← Zwrócenie wyniku klasyfikacji

```

Wypróbujmy funkcję `classify_review` na przykładowym tekście:

```

text_1 = (
    "You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award."
)

print(classify_review(
    text_1, model, tokenizer, device, max_length=train_dataset.max_length
))

```

Wynikowy model poprawnie wnioskuje "spam". Spróbujmy podać inny przykład:

```

text_2 = (
    "Hey, just wanted to check if we're still on"
    " for dinner tonight? Let me know!"
)

```

```
)  
  
print(classify_review(  
    text_2, model, tokenizer, device, max_length=train_dataset.max_length  
))
```

Model ponownie dokonuje poprawnej prognozy i zwraca etykietę „not spam”.

Na koniec zapisujemy model, na wypadek gdybyśmy chcieli użyć go później bez konieczności ponownego szkolenia. Do tego celu można użyć metody `torch.save`:

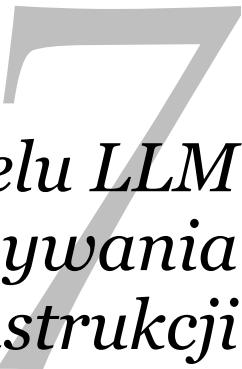
```
torch.save(model.state_dict(), "review_classifier.pth")
```

Po zapisaniu model można załadować:

```
model_state_dict = torch.load("review_classifier.pth", map_location=device)  
model.load_state_dict(model_state_dict)
```

Podsumowanie

- Istnieją różne strategie dostrajania modeli LLM, w tym dostrajanie do zadań klasyfikacji i dostrajanie do wykonywania instrukcji.
- Dostrajanie do zadań klasyfikacji polega na zastąpieniu warstwy wyjściowej modelu LLM niewielką warstwą klasyfikacji.
- W przypadku zadania klasyfikacji wiadomości tekstowych jako „spam” lub „nie spam” nowa warstwa klasyfikacji składa się tylko z dwóch węzłów wyjściowych. Wcześniej używaliśmy liczby węzłów wyjściowych równej liczbie unikatowych tokenów w słowniku (tzn. 50 256).
- Zamiast prognozy następnego tokena w tekście, jak w przypadku wstępnego szkolenia, dostrajanie do zadań klasyfikacji polega na szkoleniu modelu pod kątem generowania poprawnej etykiety klasy — na przykład „spam” lub „nie spam”.
- Wejście modelu do dostrajania to, podobnie jak w przypadku szkolenia wstępnego, tekst skonwertowany na identyfikatory tokenów.
- Przed dostrojeniem modelu LLM ładowamy wstępnie przeszkolony model jako model bazowy.
- Ocena modelu klasyfikacji obejmuje obliczenie dokładności klasyfikacji (ułamek lub procent poprawnych prognoz).
- W dostrajaniu modelu klasyfikacji wykorzystywana jest ta sama funkcja straty entropii krzyżowej, której używaliśmy podczas wstępnego szkolenia modelu LLM.



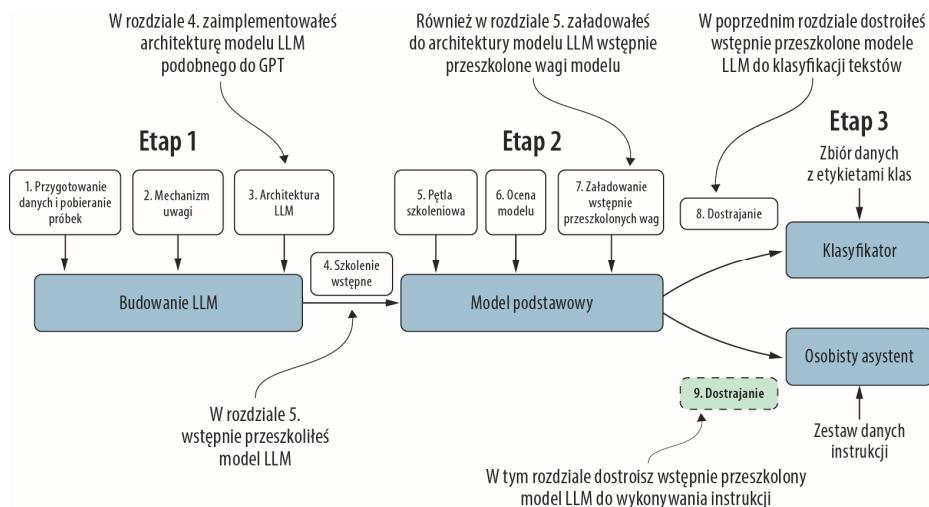
Dostrajanie modelu LLM do zadań wykonywania instrukcji

W tym rozdziale:

- Proces dostrajania modeli LLM do zadań wykonywania instrukcji
- Przygotowanie zbioru danych do nadzorowanego dostrajania pod kątem wykonywania instrukcji
- Organizowanie danych instrukcji w partie szkoleniowe
- Wczytywanie wstępnie przeszkolonego modelu LLM i dostrajanie go do wykonywania poleceń człowieka
- Wyodrębnianie odpowiedzi modelu LLM na instrukcje do oceny
- Ocena modelu LLM dostrojonego do wykonywania instrukcji

Wcześniej zaimplementowałeś architekturę LLM, przeprowadziłeś wstępne szkolenie i zimportowałeś do tworzonego modelu wstępnie przeszkolone wagę z zewnętrznych źródeł. Następnie skupiliśmy się na dostrojeniu modelu LLM do konkretnego zadania klasyfikacyjnego: rozróżniania spamu i wiadomości tekstowych niebędących spamem. W tym rozdziale zaimplementujesz proces dostrajania modelu LLM do wykonywania poleceń (rysunek 7.1). Dostrajanie do wykonywania instrukcji jest jedną z głównych technik realizowanych w ramach tworzenia modeli LLM wykorzystywanych w chatbotach, osobistych asystentach i innych zadaniach konwersacyjnych.

Na rysunku 7.1 pokazałem dwa główne sposoby dostrajania modeli LLM: dostrajanie do zadań klasyfikacji (krok 8.) i dostrajanie do wykonywania instrukcji (krok 9.). Krok 8. zaimplementowałeś w rozdziale 6. Teraz zajmiemy się dostrajaniem LLM z użyciem *zbioru danych instrukcji*.

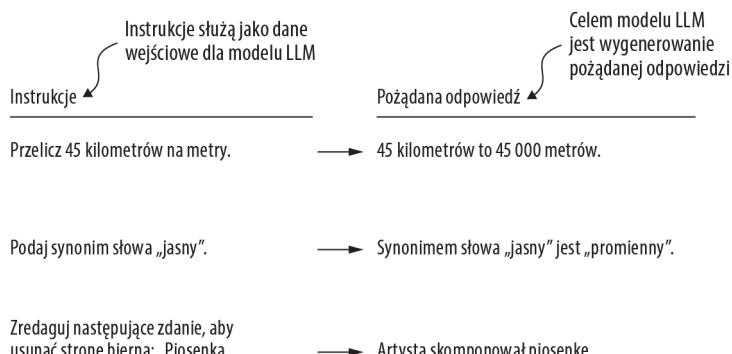


Rysunek 7.1. Trzy główne etapy kodowania LLM. Ten rozdział koncentruje się na kroku 9. etapu 3. — dostrajaniu wstępnie przeszkolonego modelu LLM do wykonywania instrukcji przekazywanych przez człowieka

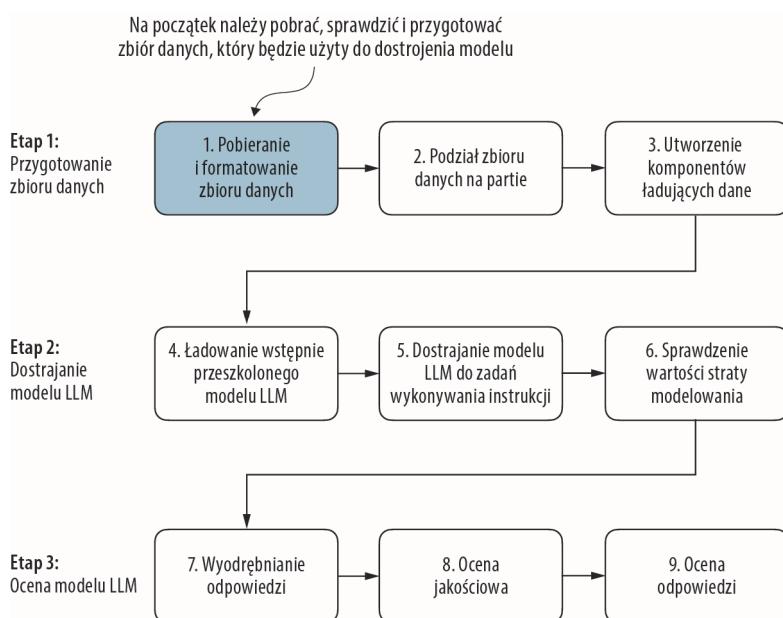
7.1. Wprowadzenie do dostrajania do wykonywania instrukcji

Jak już wiesz, wstępne szkolenie modeli LLM obejmuje procedurę, w której model uczy się generować jedno słowo na raz. Uzyskany w wyniku zastosowania tego procesu wstępnie przeszkolony model LLM umie uzupełniać tekst, co oznacza, że na podstawie fragmentów przekazanych jako dane wejściowe potrafi dokończyć zdanie lub napisać akapit tekstu. Wstępnie przeszkolone modele LLM często mają jednak trudności z konkretnymi instrukcjami, takimi jak „Popraw gramatykę w tym tekście” lub „Przekształć ten tekst na stronę bierną”. W dalszej części rozdziału przeanalizujemy konkretny przykład, w którym załadujemy wstępnie przeszkolone modele LLM jako podstawę do dostrajania pod kątem wykonywania instrukcji. Technikę tę określa się jako *nadzorowane dostrajanie pod kątem wykonywania instrukcji* (ang. *supervised instruction fine-tuning*).

Na początek skoncentrujmy się na poprawie zdolności modelu LLM do wykonywania takich instrukcji i generowania żądanych odpowiedzi (rysunek 7.2). Kluczowym aspektem dostrajania pod kątem wykonywania instrukcji jest przygotowanie zbioru danych. Następnie wykonamy wszystkie kroki trzech etapów procesu dostrajania do wykonywania instrukcji, począwszy od przygotowania zbioru danych (rysunek 7.3).



Rysunek 7.2. Przykłady instrukcji przetwarzanych przez model LLM w celu wygenerowania pożądanych odpowiedzi



Rysunek 7.3. Trzyetapowy proces dostrajania modelu LLM pod kątem wykonywania instrukcji. Etap 1. obejmuje przygotowanie zbioru danych, etap 2. koncentruje się na konfiguracji i dostrajaniu modelu, a etap 3. obejmuje ocenę modelu. Zaczniemy od kroku 1. etapu 1. – pobierania i formatowania zbioru danych

7.2. Przygotowanie zbioru danych do nadzorowanego dostrajania pod kątem wykonywania instrukcji

Pobierzmy i sformatujmy zbiór danych instrukcji do dostrajania wstępnie przeszkołonego modelu LLM. Zbiór danych składa się z 1100 par instrukcja-odpowiedź podobnych do tych na rysunku 7.2¹. Ten zbiór danych stworzyłem specjalnie do tej książki. Zainteresowani Czytelnicy mogą jednak siegnąć do „Dodatku B”, w którym można znaleźć alternatywne, publicznie dostępne zbiory danych instrukcji.

Do pobrania tego zbioru danych, który jest stosunkowo małym plikiem (tylko 204 kB) w formacie JSON, można użyć kodu przedstawionego na listingu 7.1. Format JSON, czyli *JavaScript Object Notation* (notacja obiektowa JavaScript), odzwierciedla strukturę słowników Pythona, co czyni z niego prosty i wygodny mechanizm wymiany danych, które są zarówno czytelne dla człowieka, jak i łatwe do interpretowania przez komputery.

Listing 7.1. Pobieranie zbioru danych

```
import json
import os
import urllib

def download_and_load_file(file_path, url):
    if not os.path.exists(file_path):
        with urllib.request.urlopen(url) as response:
            text_data = response.read().decode("utf-8")
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data)
    else:
        with open(file_path, "r", encoding="utf-8") as file:
            text_data = file.read()
    with open(file_path, "r") as file:
        data = json.load(file)
    return data

file_path = "instruction-data.json"
url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch"
    "/main/ch07/01_main-chapter-code/instruction-data.json"
)

data = download_and_load_file(file_path, url)
print("Liczba pozycji:", len(data))
```

← Pominiecie pobierania, jeśli plik został pobrany wcześniej

¹ W zbiorze danych stosowanym w przykładach kodu użyto par instrukcja-odpowiedź w języku angielskim — przyp. tłum.

Oto wynik uruchomienia powyższego kodu:

Liczba pozycji: 1100

Lista data, którą wczytaliśmy z pliku JSON, zawiera 1100 pozycji zbioru danych instrukcji. Spróbujmy wyświetlić jeden z tych wpisów, aby zobaczyć, jaką mają strukturę:

```
print("Przykładowa pozycja:\n", data[50])
```

Treść przykładowej pozycji to:

Przykładowa pozycja:

```
{'instruction': 'Identify the correct spelling of the following word.',  
 'input': 'Ocassion', 'output': "The correct spelling is 'Occasion.'"}  
Jak można zauważać, przykładowe pozycje są obiektami słowników Pythona zawierającymi klucze 'instruction', 'input' i 'output'. Przyjrzyjmy się innemu przykładowi:
```

```
print("Inna przykładowa pozycja:\n", data[999])
```

Na podstawie zawartości tej pozycji można zauważać, że pole 'input' może czasami być puste:

Inna przykładowa pozycja:

```
{'instruction': "What is an antonym of 'complicated'??",  
 'input': '',  
 'output': "An antonym of 'complicated' is 'simple'."}
```

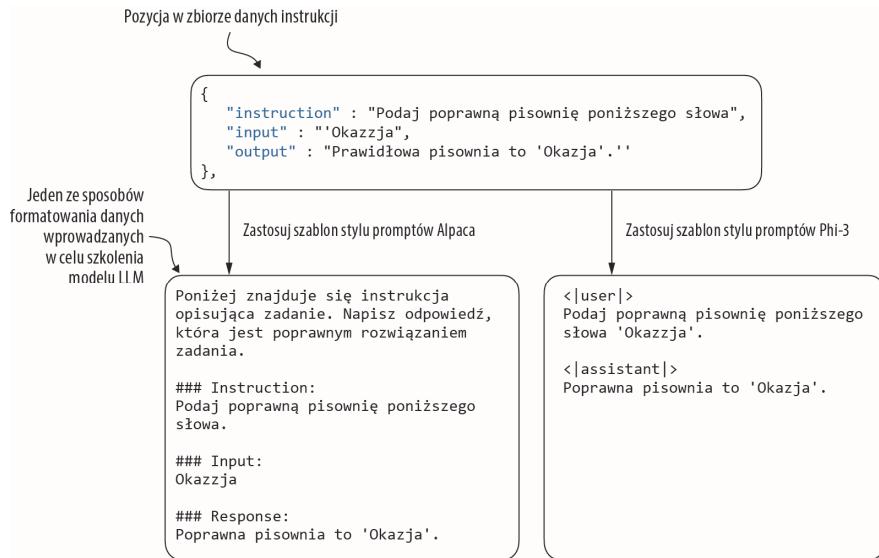
Dostrajanie pod kątem wykonywania instrukcji obejmuje szkolenie modelu na zbiorze danych, w którym wyraźnie dostarczamy pary wejście-wyjście, podobne do tych, które wyodrębniliśmy z pliku JSON. Istnieją różne metody formatowania tych pozycji na potrzeby modeli LLM. Dwa różne przykładowe formaty, często określane jako *style promptów* (ang. *prompt styles*), używane w szkoleniach znanych modeli LLM, takich jak Alpaca i Phi-3, zilustrowałem na rysunku 7.4.

Alpaca był jednym z pierwszych modeli LLM, dla których udostępniono publicznie proces dostrajania pod kątem wykonywania instrukcji. Opracowany przez Microsoft model Phi-3 uwzględniłem w celu zademonstrowania różnorodności stylów promptów. W pozostałej części tego rozdziału będziemy wykorzystywać styl promptów Alpaca. To jeden z najpopularniejszych stylów, głównie dlatego, że pomógł zdefiniować pierwotne podejście do dostrajania.

Ćwiczenie 7.1. Zmiana stylu promptów

Po dostrojeniu modelu za pomocą stylu promptów Alpaca wypróbuje zilustrowany na rysunku 7.4 styl promptów Phi-3 i zaobserwuj, czy zmiana wpłynęła na jakość odpowiedzi modelu.

Listing 7.2 zawiera definicję funkcji `format_input`, której można użyć do konwersji pozycji na liście `data` na format wejściowy w stylu Alpaca.



Rysunek 7.4. Porównanie stylów promptów w celu dostrajania modeli LLM pod kątem wykonywania instrukcji. Styl Alpaca (po lewej) wykorzystuje ustrukturyzowany format ze zdefiniowanymi sekcjami instrukcji, wejścia i odpowiedzi, podczas gdy styl Phi-3 (po prawej) wykorzystuje prostszy format z tokenami <|user|> i <|assistant|>

Listing 7.2. Implementacja funkcji formatowania promptów

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instrukcja:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Wejście:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

Zamieszczona na listingu funkcja `format_input` pobiera jako dane wejściowe pozycję w słowniku i konstruuje sformatowany ciąg znaków. Przetestujmy to na pozycji zbioru danych wejściowych `data[50]`, z której korzystaliśmy wcześniej:

```
model_input = format_input(data[50])
desired_response = f"\n\n### Odpowiedź:\n{data[50]['output']}"
print(model_input + desired_response)
```

Sformatowany wynik ma następującą postać:

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instrukcja:

Identify the correct spelling of the following word.

Wejście:
Ocassion

Odpowiedź:
The correct spelling is 'Occasion.'

Zwróć uwagę, że jeśli pole 'input' jest puste, funkcja `format_input` pominie opcjonalną część ### Input:. Aby to sprawdzić, można użyć funkcji `format_input` w odniesieniu do wykorzystanej wcześniej pozycji `data[999]`:

```
model_input = format_input(data[999])
desired_response = f"\n\n## Odpowiedź:\n{data[999]['output']}"
print(model_input + desired_response)
```

Na podstawie wyników widać, że pozycje z pustym polem 'input' nie zawierają w sformowanym wyniku fragmentu ### Wejście::

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instrukcja:
What is an antonym of 'complicated'?

Odpowiedź:
An antonym of 'complicated' is 'simple'.

Przed przystąpieniem do konfiguracji mechanizmów ładowania danych frameworka PyTorch, czym zajmiemy się w następnym podrozdziale, spróbujmy podzielić zbiór danych na zbiory szkoleniowy, walidacyjny i testowy w sposób analogiczny do tego, jaki zastosowaliśmy w poprzednim rozdziale w odniesieniu do zbioru danych do klasyfikacji spamu. Kod odpowiedzialny za podział poszczególnych porcji znajdziesz na listingu 7.3.

Listing 7.3. Podział zbioru danych

```
train_portion = int(len(data) * 0.85)           ← Wykorzystanie 85% danych do szkolenia
test_portion = int(len(data) * 0.1)                ← Wykorzystanie 10% danych do testowania
val_portion = len(data) - train_portion - test_portion ← Wykorzystanie pozostałych 5% danych do walidacji

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Rozmiar zbioru szkoleniowego:", len(train_data))
print("Rozmiar zbioru walidacyjnego:", len(val_data))
print("Rozmiar zbioru testowego:", len(test_data))
```

Podział wykonany za pomocą powyższego kodu skutkuje następującymi rozmiarami zbiorów danych:

Rozmiar zbioru szkoleniowego: 935

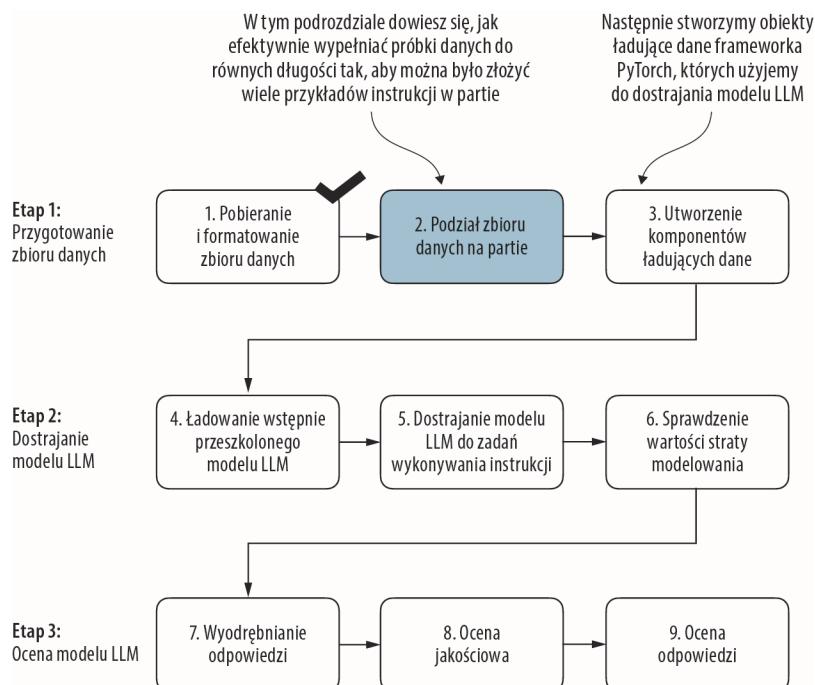
Rozmiar zbioru walidacyjnego: 55

Rozmiar zbioru testowego: 110

Po pomyślnym pobraniu zbioru danych i podzieleniu go oraz zapoznaniu się ze sposobami formatowania promptów dla zbioru danych możemy przystąpić do właściwej implementacji procesu dostrajania modelu pod kątem wykonywania instrukcji. W kolejnym podrozdziale zajmę się opracowaniem metody konstruowania partii szkoleniowych w celu dostrajania modelu LLM.

7.3. Organizowanie danych w partie szkoleniowe

Kolejny krok fazy implementacji procesu dostrajania pod kątem wykonywania instrukcji, zilustrowany na rysunku 7.5, koncentruje się na skutecznym konstruowaniu partii szkoleniowych. Obejmuje to zdefiniowanie metody, dzięki której model otrzyma podczas procesu dostrajania sformatowane dane szkoleniowe.



Rysunek 7.5. Trzyetapowy proces dostrajania modelu LLM. Nadszedł czas, by zająć się krokiem 2. etapu 1. — wyodrębnianiem partii szkoleniowych

Partie szkoleniowe w poprzednim rozdziale utworzyłeś automatycznie za pomocą klasy `DataLoader` z framework PyTorch. Do łączenia list próbek w partie klasa `DataLoader` wykorzystuje domyślną funkcję `collate`. Funkcja `collate` jest odpowiedzialna za pobranie listy pojedynczych próbek danych i połączenie ich w partię, którą model może skutecznie przetwarzać podczas szkolenia.

Proces grupowania próbek w celu dostrojenia modelu pod kątem wykonywania instrukcji jest jednak nieco bardziej skomplikowany i wymaga stworzenia własnej, niestandardowej funkcji `collate`, którą później należy podłączyć do obiektu `DataLoader`. Tę niestandardową funkcję `collate` zaimplementujemy w celu obsłużenia określonych wymagań i formatowania zbioru danych pod kątem dostrajania modelu do wykonywania instrukcji.

Zajmijmy się procesem podziału na partie obejmującym kilka kroków. Jednym z nich jest zakodowanie pokazanej na rysunku 7.6 niestandardowej funkcji `collate`. Najpierw, aby zaimplementować kroki 2.1 i 2.2, kodujemy klasę `InstructionDataset`, która stosuje funkcję `format_input` i dokonuje wstępnej tokenizacji wszystkich wejść w zbiorze danych, podobnie jak w przypadku klasy `SpamDataset` z rozdziału 6. Ten dwuetapowy proces, szczegółowo zilustrowany na rysunku 7.7, jest zaimplementowany w metodzie `__init__` — konstruktorze klasy `InstructionDataset` (listing 7.4).

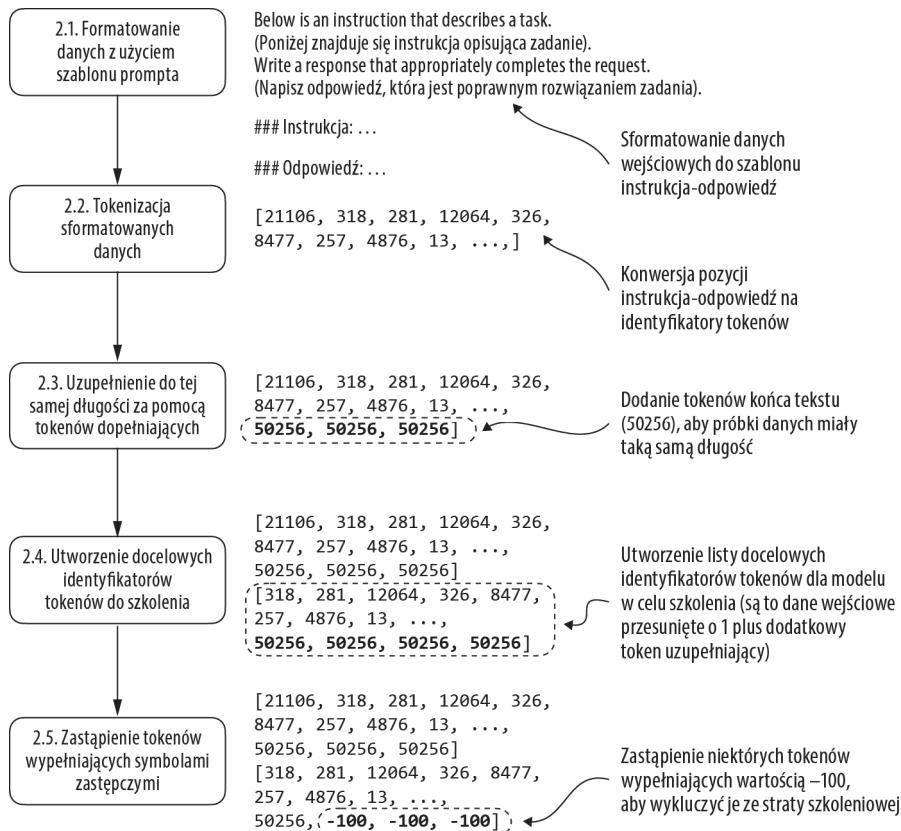
Tak jak w podejściu, które zastosowaliśmy podczas dostrajania modelu pod kątem zadań klasyfikacji, chcemy przyspieszyć szkolenie przez zebranie wielu przykładów szkoleniowych w partie. Wymaga to uzupełnienia wszystkich danych wejściowych tak, by miały taką samą długość. Podobnie jak w przypadku dostrajania do zadań klasyfikacji, w roli tokena wypełniającego stosujemy `<|endoftext|>`.

Zamiast dodawać tokeny `<|endoftext|>` do tekstowych danych wejściowych, można bezpośrednio dołączyć do stokenizowanych danych wejściowych identyfikator odpowiadający tokenowi `<|endoftext|>`. Aby sprawdzić, którego identyfikatora tokena powinniśmy użyć, możemy zastosować metodę `.encode` tokenizera w odniesieniu do tokena `<|endoftext|>`:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

Wynikowy identyfikator tokena to 50256.

W kroku 2.3 procesu (patrz rysunek 7.6) przyjmujemy bardziej wyrafinowane podejście. Polega ono na stworzeniu niestandardowej funkcji `collate`, którą można przekazać do komponentu ładującego dane. Niestandardowa funkcja `collate` jest odpowiedzialna za wypełnianie przykładów szkoleniowych w każdej partii do tej samej długości, przy czym różne partie mogą mieć różne długości (patrz rysunek 7.8). Zastosowanie tego podejścia pozwala zminimalizować niepotrzebne wypełnianie, ponieważ sekwencje są rozszerzane tylko tak, aby pasowały do najdłuższej w każdej partii, a nie do całego zbioru danych.



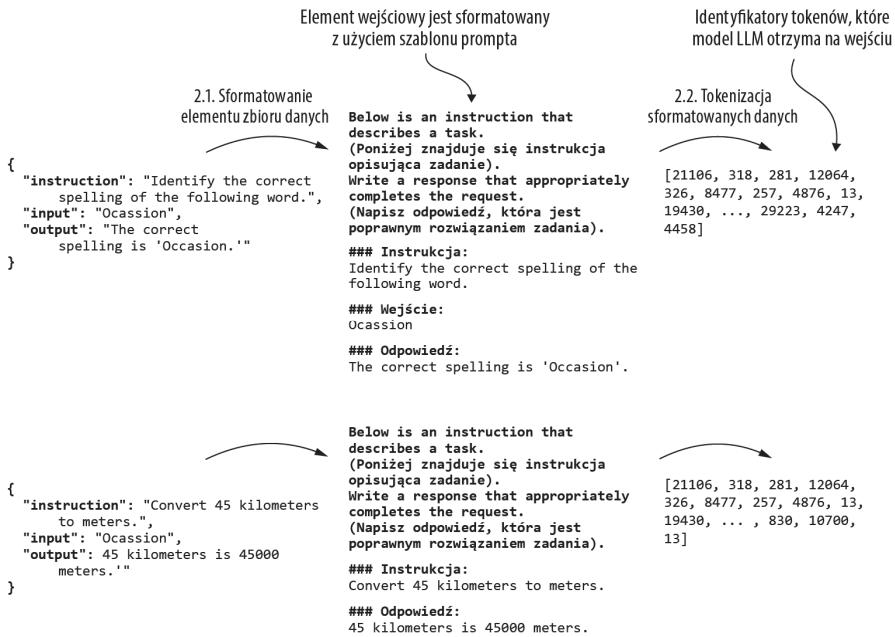
Rysunek 7.6. Pięć podetapów związanych z implementacją procesu podziału na partie: (2.1) zastosowanie szablonu promptów, (2.2) użycie tokenizacji znanej Ci z poprzednich rozdziałów, (2.3) dodanie tokenów wypełniających, (2.4) utworzenie identyfikatorów tokenów docelowych oraz (2.5) zastąpienie zastępczych tokenów -100 w celu zamaskowania tokenów uzupełniających w funkcji straty²

Listing 7.4. Implementacja klasy reprezentującej zbiór danych instrukcji

```
import torch
from torch.utils.data import Dataset

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.encoded_texts = []
        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Odpowiedź:\n{entry['output']}"
```

² Na rysunku w nawiasach obok promptów przesyłanych do modelu podano ich tłumaczenie na język polski – przyp. tłum.



Rysunek 7.7. Pięć podetapów związanych z implementacją procesu podziału na partie: wpisy są najpierw formatowane z użyciem określonego szablonu prompta (2.1), a następnie poddawane tokenizacji (2.2), co skutkuje utworzeniem sekwencji identyfikatorów tokenów, które model może przetwarzać³

```
full_text = instruction_plus_input + response_text
self.encoded_texts.append(
    tokenizer.encode(full_text)
)

def __getitem__(self, index):
    return self.encoded_texts[index]

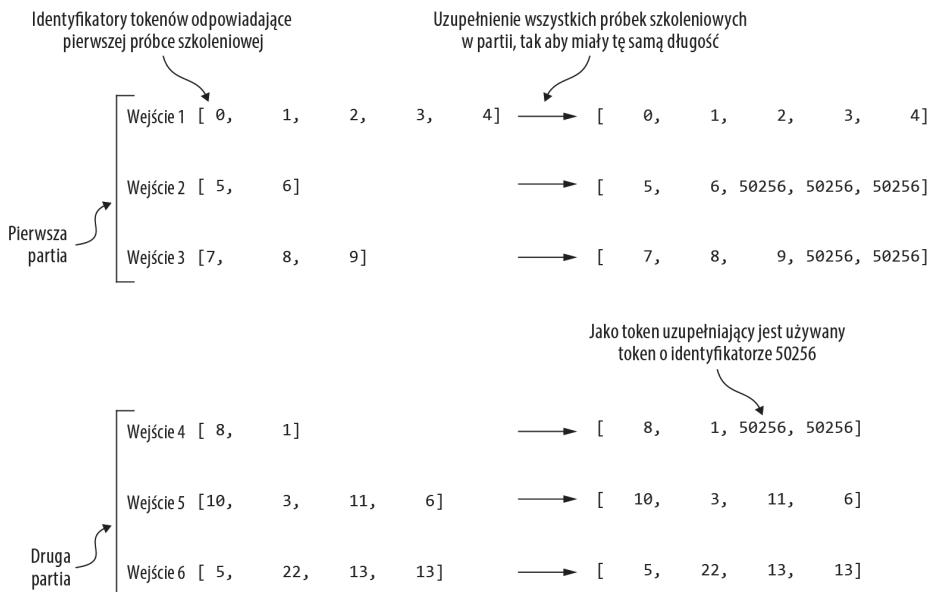
def __len__(self):
    return len(self.data)
```

Proces uzupełniania można zaimplementować za pomocą niestandardowej funkcji `collate`:

```
def custom_collate_draft_1(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst = []
```

← Szukanie najdłuższej sekwencji w partii

³ Na rysunku w nawiasach obok promptów przesyłanych do modelu podano ich tłumaczenie na język polski — *przyp. tłum.*



Rysunek 7.8. Wypełnianie przykładów szkoleniowych w partiach z użyciem tokena o identyfikatorze 50256 w celu zapewnienia jednolitej długości w każdej partii. Jak pokazałem na pierwszym i drugim wykresie, każda partia może mieć różną długość

```

for item in batch:           ← Wypełnianie i przygotowanie wejść
    new_item = item.copy()
    new_item += [pad_token_id]

    padded = (
        new_item + [pad_token_id] * 
        (batch_max_length - len(new_item))
    )
    inputs = torch.tensor(padded[:-1]) ← Usunięcie nadmiarowego
    inputs_lst.append(inputs)          tokena dodanego wcześniej
inputs_tensor = torch.stack(inputs_lst).to(device) ← Konwersja listy wejść na tensor
return inputs_tensor           i przesunięcie go do docelowego urządzenia

```

Zaimplementowaną funkcję `custom_collate_draft_1` zaprojektowano do integracji z klasą `DataLoader` frameworka PyTorch, ale można jej również używać jako samodzielnego narzędzia. W prezentowanym przykładzie stosujemy ją niezależnie, w celu przetestowania i zweryfikowania, czy działa zgodnie z przeznaczeniem. Wypróbujmy ją na trzech różnych danych wejściowych, z których chcemy utworzyć partię, przy założeniu, że każda próbka będzie uzupełniona do tej samej długości:

```

inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]
batch = (
    inputs_1,
    inputs_2,
)

```

```

    inputs_3
)
print(custom_collate_draft_1(batch))

```

Wynikowa partia ma następującą postać:

```

tensor([[ 0,  1,  2,  3,  4],
       [ 5,  6, 50256, 50256, 50256],
       [ 7,  8,  9, 50256, 50256]])

```

Na podstawie tego wyniku możemy stwierdzić, że wszystkie dane wejściowe zostały uzupełnione do długości najdłuższej listy wejściowej, `inputs_1`, zawierającej pięć identyfikatorów tokenów.

Właśnie zaimplementowałeś pierwszą niestandardową funkcję `collate` do tworzenia partii na podstawie listy próbek wejściowych. Jednak jak dowiedziałeś się wcześniej, trzeba również utworzyć partie z docelowymi identyfikatorami tokenów odpowiadającymi partii identyfikatorów wejściowych. Te docelowe identyfikatory, jak pokazałem na rysunku 7.9, mają kluczowe znaczenie, ponieważ reprezentują to, co chcemy, aby model wygenerował, i czego potrzebujemy podczas szkolenia do obliczenia straty dla aktualizacji wag. Oznacza to, że modyfikujemy niestandardową funkcję `collate` w taki sposób, aby oprócz identyfikatorów tokenów wejściowych zwracała docelowe identyfikatory tokenów.

Podobnie jak w procesie wstępnego uczenia modelu LLM, docelowe identyfikatory tokenów odpowiadają identyfikatorom tokenów wejściowych, ale są przesunięte o jedną pozycję w prawo. Pokazana konfiguracja, jak pokazałem na rysunku 7.10, pozwala modelowi LLM nauczyć się przewidywać następny token w sekwencji.

Zamieszczona poniżej zaktualizowana funkcja `collate` na podstawie identyfikatorów tokenów wejściowych generuje docelowe identyfikatory tokenów:

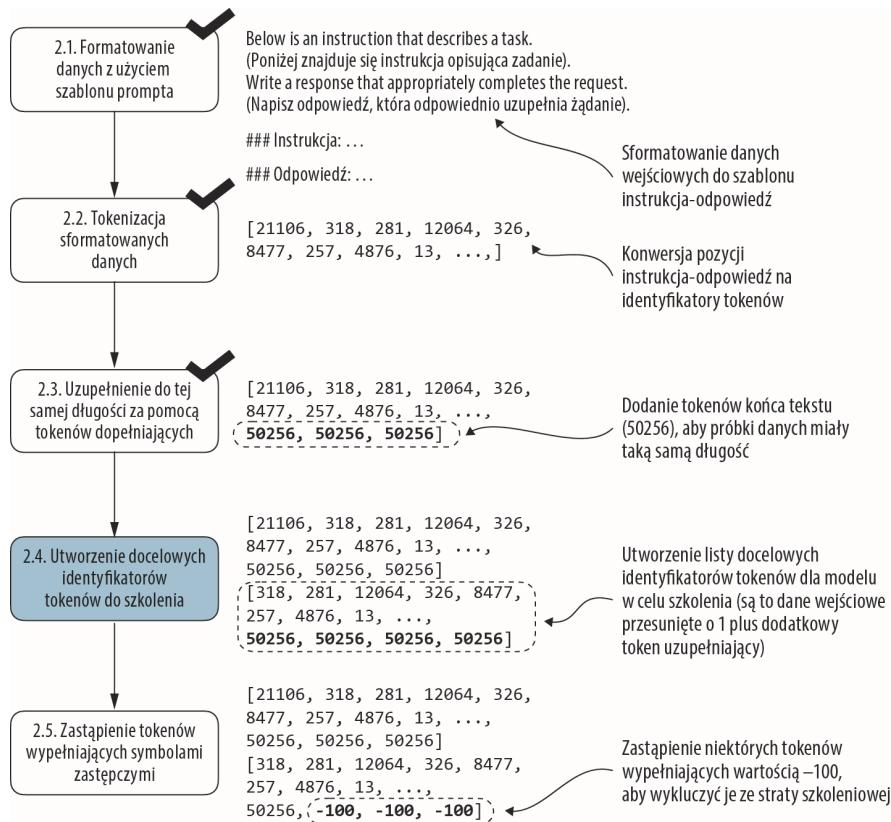
```

def custom_collate_draft_2(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]
        padded = (
            new_item + [pad_token_id] * 
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])           ← Obcięcie ostatniego tokena wejściowego
        targets = torch.tensor(padded[1:])           ← Przesunięcie o 1 w prawo w celu
                                                    uzyskania tensora docelowego
        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor

```



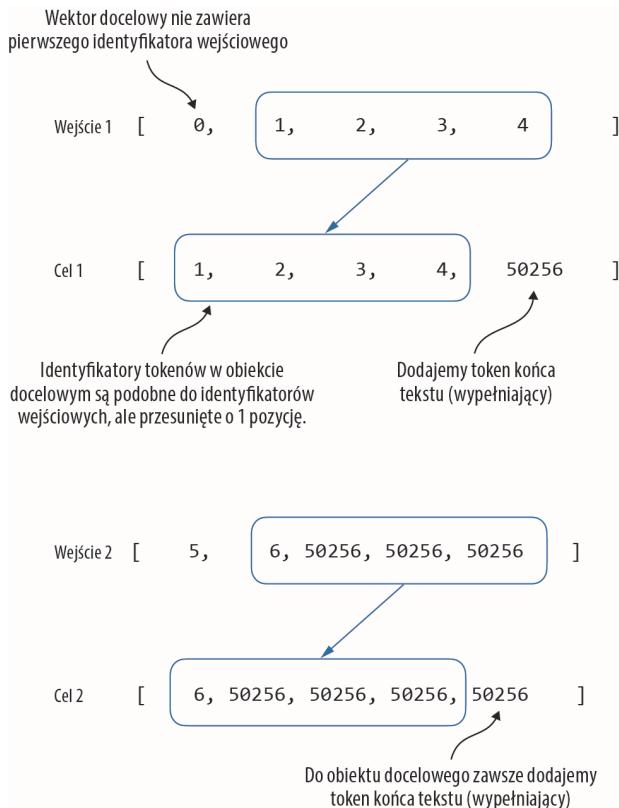
Rysunek 7.9. Pięć podetapów związanych z implementacją procesu podziału na partie. Teraz koncentrujemy się na kroku 2.4, czyli tworzeniu identyfikatorów tokenów docelowych. Ten krok jest niezbędny, ponieważ umożliwia modelowi uczenie się i przewidywanie tokenów, które ma wygenerować⁴

```
inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)
```

Nowa funkcja `custom_collate_draft_2` zastosowana do zdefiniowanych wcześniej trzech list wejściowych tworzących przykładową partię `batch`, zwraca partię wejściową i docelową:

```
tensor([[ 0,    1,    2,    3,    4], ←
       [ 5,    6, 50256, 50256, 50256], ← Pierwszy tensor reprezentuje dane wejściowe
       [ 7,    8,    9, 50256, 50256]]),
tensor([[ 1,    2,    3,    4, 50256], ←
       [ 6, 50256, 50256, 50256, 50256], ← Drugi tensor reprezentuje tokeny docelowe
       [ 8,    9, 50256, 50256, 50256]])
```

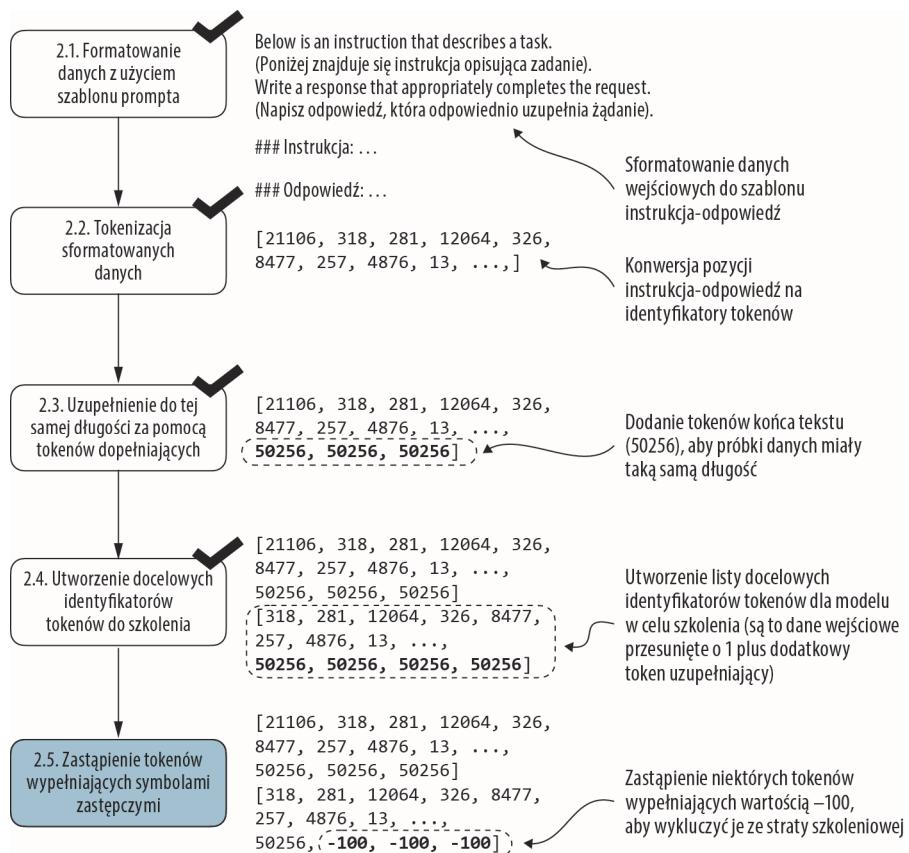
⁴ Na rysunku w nawiasach obok promptów przesyłanych do modelu podano ich tłumaczenie na język polski — *przyp. tłum.*



Rysunek 7.10. Wyrównanie tokena wejściowego i docelowego wykorzystywane w procesie dostrajania modelu LLM pod kątem wykonywania instrukcji. Dla każdej sekwencji wejściowej tworzona jest odpowiadająca jej sekwencja docelowa przez przesunięcie identyfikatorów tokenów o jedną pozycję w prawo, pominięcie pierwszego tokena wejściowego i dołączenie tokena końca tekstu

W kolejnym kroku, jak zaznaczyłem na rysunku 7.11, do wszystkich tokenów wypełniających przypisujemy wartość zastępczą -100 . Ta specjalna wartość pozwala wykluczyć tokeny wypełniające z obliczeń strat zbioru szkoleniowego. Dzięki temu zyskujemy pewność, że na uczenie modelu mają wpływ wyłącznie znaczące dane. Ten proces omówię bardziej szczegółowo po zaimplementowaniu tej modyfikacji (podczas dostrajania do zadań klasyfikacji nie trzeba się było tym martwić, ponieważ szkolenie modelu było realizowane wyłącznie na podstawie ostatniego tokena wyjściowego).

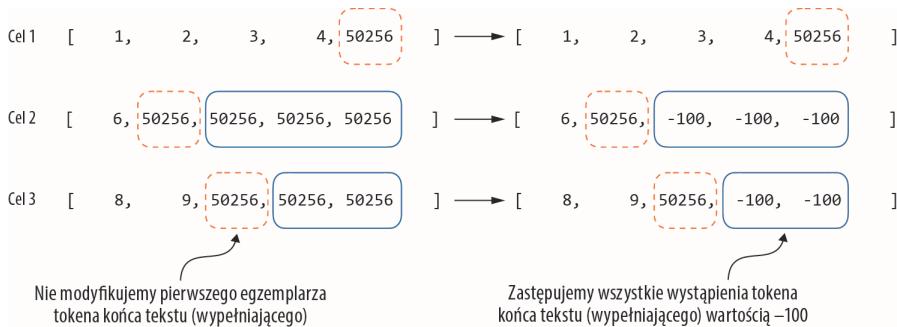
Należy jednak zauważyć, że na liście docelowej zachowujemy jeden token końca tekstu o identyfikatorze 50256 (patrz rysunek 7.12). Dzięki temu model LLM może się nauczyć, kiedy generować token końca tekstu w odpowiedzi na instrukcję. Wykorzystujemy to jako wskaźnik, że wygenerowana odpowiedź jest kompletna.



Rysunek 7.11. Pięć podetapów związanych z implementacją procesu podziału na partie. Po utworzeniu sekwencji docelowej przez przesunięcie identyfikatorów tokenów o jedną pozycję w prawo i dołączenie tokenu końca tekstu w kroku 2.5 zastępujemy tokeny wypełniające <endoftext> wartością zastępczą (-100)⁵

Na listingu 7.5 zamieściłem zmodyfikowaną wersję niestandardowej funkcji `collate`, w której tokeny o identyfikatorze 50256 zostały na listach docelowych zastąpione wartością -100. Dodatkowo wprowadzamy parametr `allowed_max_length`, który pozwala opcjonalnie ograniczyć długość próbek. Ta modyfikacja jest przydatna w sytuacjach, gdy planujesz korzystanie z własnych zbiorów danych, które przekraczają obsługiwany przez model GPT-2 rozmiar kontekstu, 1024 tokeny.

⁵ Na rysunku w nawiasach obok promptów przesyłanych do modelu podano ich tłumaczenie na język polski – *przyp. tłum.*



Rysunek 7.12. Krok 2.4 w procesie zastępowania tokenów w partii docelowej w ramach przygotowania danych szkoleniowych. Zastępujemy wszystkie tokeny końca tekstu z wyjątkiem pierwszego egzemplarza, którego używamy jako wypełnienia, wartością zastępczą -100. W każdej sekwencji docelowej zachowujemy początkowy token końca tekstu

Listing 7.5. Implementacja niestandardowej funkcji podziału na partie

```
def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (
            new_item + [pad_token_id] * 
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])

        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index | Zastąpienie w tensorze docelowym wszystkich tokenów wypełniających oprócz pierwszego wartością ignore_index

        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length] | Opcjonalne przycięcie do maksymalnej długości sekwencji

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor
```

Wypełnienie sekwencji do max_length
Przyjęcie ostatniego tokena dla wejść, przesunięcie o jedną pozycję w prawo w celu utworzenia tensora docelowego

Wypróbujmy funkcję `collate` na utworzonej wcześniej przykładowej partii, aby sprawdzić, czy działa zgodnie z zamierzeniami:

```
inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)
```

Oto uzyskane wyniki. Pierwszy tensor reprezentuje wejścia, a drugi reprezentuje cele:

```
tensor([[ 0,    1,    2,    3,    4],
       [ 5,    6, 50256, 50256, 50256],
       [ 7,    8,    9, 50256, 50256]])
tensor([[ 1,    2,    3,    4, 50256],
       [ 6, 50256, -100, -100, -100],
       [ 8,    9, 50256, -100, -100]])
```

Zaktualizowana funkcja `collate` działa zgodnie z oczekiwaniami: modyfikuje listę docelową przez wstawienie tokena o identyfikatorze `-100`. Jaka jest logika tej modyfikacji? Przyjrzyjmy się jej podstawowemu celowi.

Dla celów demonstracyjnych rozważmy następujący prosty przykład, w którym każdy wynikowy logit odpowiada potencjalnemu tokenowi ze słownika modelu. Oto jak można obliczyć stratę entropii krzyżowej (wprowadzoną w rozdziale 5.) podczas szkolenia w sytuacji, gdy model przewiduje sekwencję tokenów. To działanie przypomina to, co zrobiliśmy podczas wstępniego szkolenia modelu, gdy dostrajaliśmy go do zadań klasyfikacji:

```
logits_1 = torch.tensor(
    [[-1.0, 1.0],   ← prognozy dla 1. tokena
     [-0.5, 1.5]]  ← prognozy dla 2. tokena
)
targets_1 = torch.tensor([0, 1]) # Correct token indices to generate
loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)
```

Wartość straty obliczona za pomocą powyższego kodu wynosi `1.1269`:

```
tensor(1.1269)
```

Jak można oczekiwąć, wprowadzenie dodatkowego identyfikatora tokena wpływa na obliczenia straty:

```
logits_2 = torch.tensor(
    [[-1.0, 1.0],
     [-0.5, 1.5],
     [-0.5, 1.5]]  ← Nowa prognoza identyfikatora trzeciego tokena
)
targets_2 = torch.tensor([0, 1, 1])
loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
```

Po dodaniu trzeciego tokena wartość straty wynosi `0.7936`.

Do tej pory przeprowadziliśmy kilka mniej lub bardziej oczywistych przykładowych obliczeń z użyciem funkcji straty entropii krzyżowej framework PyTorch, tej samej funkcji straty, której używaliśmy w funkcjach szkoleniowych do wstępniego

szkolenia i dostrajania do zadań klasyfikacji. Przejdźmy teraz do bardziej interesującej części i zobaczymy, co się stanie, jeśli zastąpimy trzeci identyfikator docelowego tokena wartością -100:

```
targets_3 = torch.tensor([0, 1, -100])
loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
```

Oto uzyskany wynik:

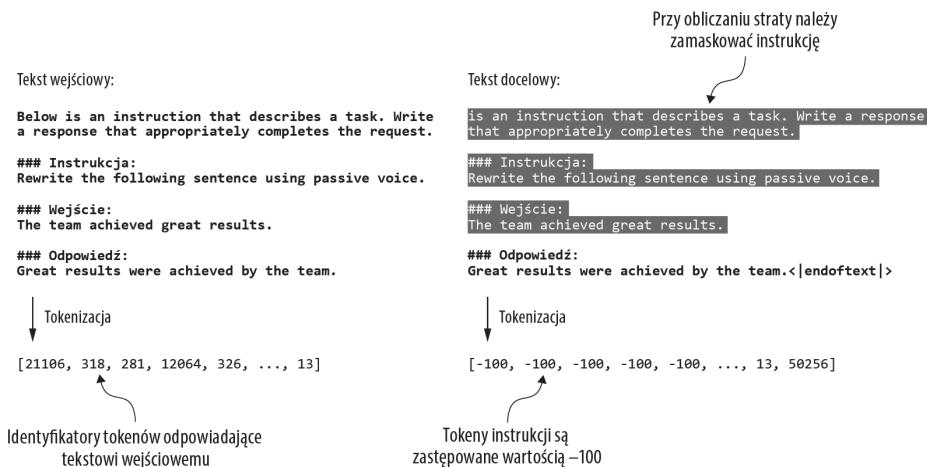
```
tensor(1.1269)
loss_1 == loss_3: tensor(True)
```

Wynikowa strata dla tych trzech przykładów próbek szkoleniowych jest identyczna ze stratą, którą obliczyliśmy na podstawie dwóch wcześniejszych próbek szkoleniowych. Mówiąc inaczej, funkcja straty entropii krzyżowej zignorowała trzecią pozycję w wektorze targets_3, identyfikator tokenu odpowiadający wartości -100 (zainteresowani Czytelnicy mogą spróbować zastąpić wartość -100 innym identyfikatorem tokenu, który nie jest równy 0 lub 1; taka próba spowoduje błąd).

Co zatem jest takiego szczególnego w wartości -100, że jest ona ignorowana w obliczeniach entropii krzyżowej? Domyslnym ustawieniem funkcji entropii krzyżowej w PyTorch jest cross_entropy(..., ignore_index=-100). Oznacza to, że funkcja ignoruje cele oznaczone wartością -100. Z ustawienia ignore_index korzystamy w celu zignorowania dodatkowych tokenów końca tekstu (wypełnienia), których użyliśmy do uzupełnienia przykładów szkoleniowych do takiej samej długości w każdej partii. Chcemy jednak zachować wśród wartości docelowych jeden token o identyfikatorze 50256 (koniec tekstu), ponieważ pomaga on modelowi LLM nauczyć się generowania tokenów końca tekstu, których można użyć jako wskaźników kompletności odpowiedzi.

Oprócz maskowania tokenów wypełniających powszechnie jest również maskowanie identyfikatorów tokenów docelowych, odpowiadających instrukcji (rysunek 7.13). Maskowanie docelowych identyfikatorów tokenów modeli LLM odpowiadających instrukcji umożliwia obliczanie entropii krzyżowej wyłącznie dla wygenerowanych docelowych identyfikatorów odpowiedzi. Dzięki temu model jest szkolony pod kątem koncentrowania się na generowaniu dokładnych odpowiedzi zamiast na zapamiętywaniu instrukcji, co może pomóc zmniejszyć efekt nadmiernego dopasowania.

W chwili gdy pisałem te słowa, wśród ekspertów nie było zgody, czy podczas dostrajania instrukcji maskowanie instrukcji jest korzystne. Na przykład w artykule Shi i współpracowników z 2024 roku, *Instruction Tuning With Loss Over Instructions* (<https://arxiv.org/abs/2405.14394>), wskazano, że brak maskowania instrukcji wpływa korzystnie na wydajność modelu LLM (więcej szczegółów znajdziesz w „Dodatku B”). W tym przykładzie nie zastosuję maskowania. Wypróbowanie tej techniki pozostawiem zainteresowanym Czytelnikom jako opcjonalne ćwiczenie.



Rysunek 7.13. Po lewej: sformatowany tekst wejściowy poddajemy tokenizacji, a następnie przekazujemy podczas szkolenia modelowi LLM. Po prawej: przygotowany dla modelu tekst docelowy LLM, w którym można opcjonalnie zamaskować sekcję instrukcji. Oznacza to zastąpienie identyfikatorów tokenów wartością **-100** (`ignore_index`)

Ćwiczenie 7.2. Maskowanie instrukcji i danych wejściowych

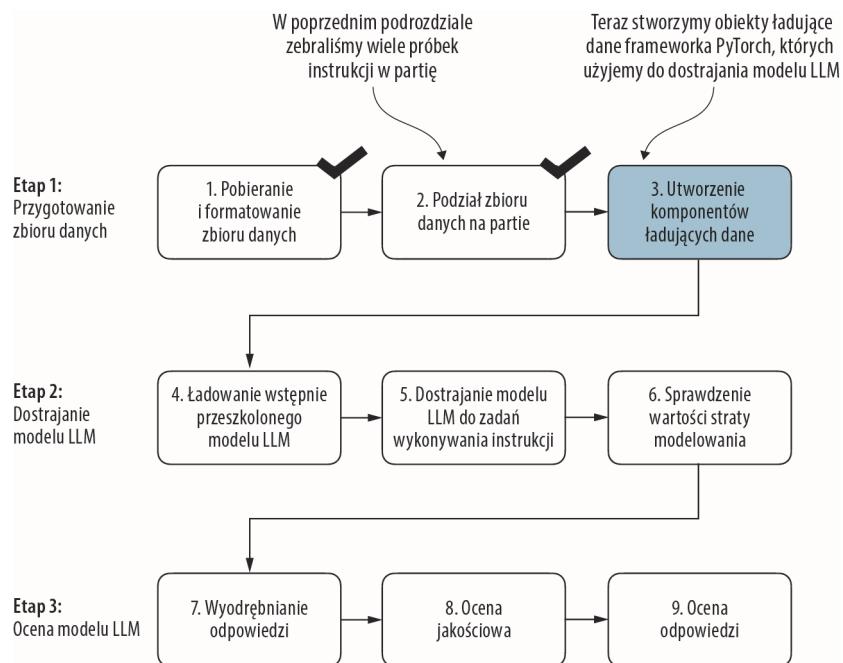
Po ukończeniu lektury tego rozdziału i dostrojeniu modelu z użyciem obiektu klasy `InstructionDataset`, aby użyć metody maskowania instrukcji zilustrowanej na rysunku 7.13, zastąp instrukcje i tokeny wejściowe maską **-100**. Następnie oceń, czy to działanie pozytywnie wpływa na wydajność modelu.

7.4. Tworzenie mechanizmów ładujących dane dla zbioru danych instrukcji

Ukończyliśmy kilka etapów implementacji klasy `InstructionDataset` i funkcji `custom_collate_fn` dla zbioru danych instrukcji. Jak pokazałem na rysunku 7.14, jesteś teraz gotowy do zebrania owoców tej pracy. Wystarczy, że podłączysz zarówno obiekt `InstructionDataset`, jak i funkcję `custom_collate_fn` do komponentów ładowania danych frameworka PyTorch.

Te komponenty ładujące automatycznie tasują i organizują partie wykorzystywane w procesie dostrajania modelu LLM do wykonywania instrukcji.

Przed przystąpieniem do implementacji kroku tworzenia komponentu ładującego dane warto krótko omówić ustawienie `device` w funkcji `custom_collate_fn`. Funkcja `custom_collate_fn` zawiera kod przenoszący tensory wejściowe i docelowe (np. `torch.stack(inputs_lst).to(device)`) do wskazanego urządzenia, którym może być "cpu"



Rysunek 7.14. Trzyetapowy proces dostrajania modelu LLM do zadań wykonywania instrukcji. Do tej pory przygotowałeś zbiór danych i zaimplementowałeś niestandardową funkcję `collate` w celu podzielenia zbioru danych instrukcji na partie. Teraz możesz utworzyć i zastosować obiekty ładowające dane do zbiorów szkoleniowego, walidacyjnego i testowych, potrzebnych w procesie dostrajania modelu LLM do zadań wykonywania instrukcji

lub "cuda" (dla procesorów graficznych firmy Nvidia) lub opcjonalnie "mps" dla komputerów Mac z układami Apple Silicon.

UWAGA Jeśli zdecydujesz się na użycie urządzenia "mps", możesz uzyskać wyniki obliczeń inne od tych, które przedstawiam w treści tego rozdziału, ponieważ obsługa Apple Silicon we frameworku PyTorch wciąż jest eksperymentalna.

Wcześniej w głównej pętli szkoleniowej przenosiliśmy dane na urządzenie docelowe (np. układ GPU w przypadku ustawienia `device="cuda"`). Wykonywanie tej operacji w ramach funkcji `collate` ma tę zaletę, że proces transferu układu docelowego odbywa się w tle, poza pętlą uczenia, co zapobiega blokowaniu układu GPU podczas szkolenia modelu.

Oto kod odpowiedzialny za inicjalizację urządzenia `device`:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# if torch.backends.mps.is_available():
#     device = torch.device('mps')
print("Układ:", device)
```

Usuń te dwa wiersze, aby użyć układu GPU na chipie Apple Silicon

Uruchomienie powyższego kodu spowoduje wyświetlenie komunikatu "Układ: cpu" lub "Układ: cuda", w zależności od konfiguracji sprzętowej.

Następnie, aby wykorzystać wybrane ustawienie device w funkcji `custom_collate_fn` po podłączeniu go do klasy `DataLoader` frameworka PyTorch, użyjemy funkcji częściowej ze standardowej biblioteki Pythona `functools` (co spowoduje utworzenie nowej wersji funkcji ze wstępnie wypełnionym argumentem `device`). Dodatkowo ustawiamy wartość `allowed_max_length` na 1024, co spowoduje przycięcie danych do maksymalnej długości kontekstu obsługiwanej przez model GPT-2 (która dostroimy później):

```
from functools import partial

customized_collate_fn = partial(
    custom_collate_fn,
    device=device,
    allowed_max_length=1024
)
```

Następnie można, tak jak wcześniej, skonfigurować obiekty ładujące dane. Tym razem jednak użyjemy w procesie podziału na partie niestandardowej funkcji `collate` (listing 7.6).

Listing 7.6. Inicjalizacja obiektów ładujących dane

```
from torch.utils.data import DataLoader

num_workers = 0 ←
batch_size = 8 | Jeśli Twój system operacyjny obsługuje równoległe procesy Pythona,
                możesz spróbować zwiększyć tę liczbę.

torch.manual_seed(123)

train_dataset = InstructionDataset(train_data, tokenizer)
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers
)

val_dataset = InstructionDataset(val_data, tokenizer)
val_loader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers
)

test_dataset = InstructionDataset(test_data, tokenizer)
test_loader = DataLoader(
    test_dataset,
```

```
batch_size=batch_size,  
collate_fn=customized_collate_fn,  
shuffle=False,  
drop_last=False,  
num_workers=num_workers  
)
```

Przyjrzyjmy się wymiarom partii wejściowej i docelowej generowanych przez obiekt ładujący dane:

```
print("Szkoleniowy mechanizm ładujący:")  
for inputs, targets in train_loader:  
    print(inputs.shape, targets.shape)
```

Oto wynik działania tego kodu (skrócony dla oszczędności miejsca):

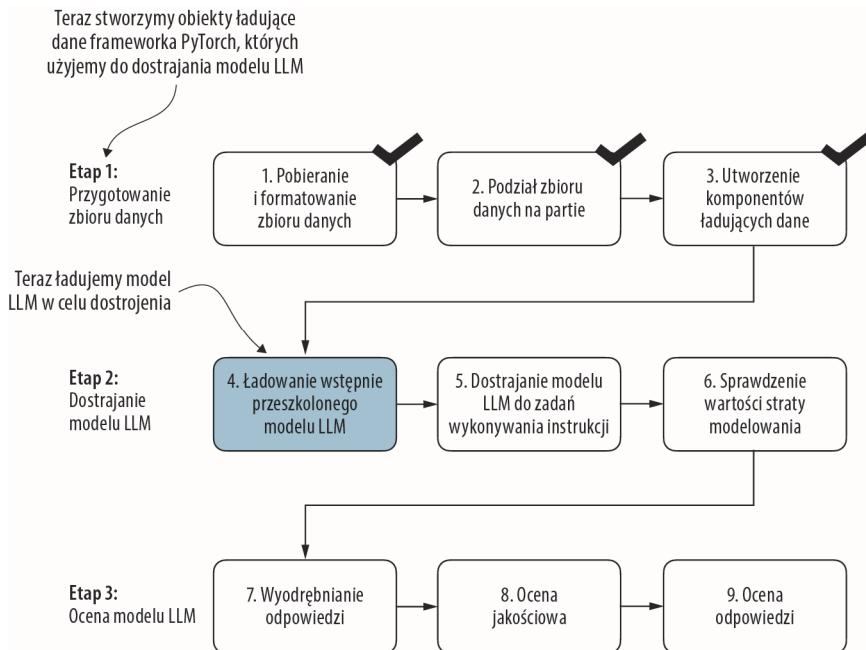
```
Szkoleniowy mechanizm ładujący:  
torch.Size([8, 61])  
torch.Size([8, 61])  
torch.Size([8, 76])  
torch.Size([8, 76])  
torch.Size([8, 76])  
torch.Size([8, 73])  
...  
torch.Size([8, 73])  
torch.Size([8, 74])  
torch.Size([8, 74])  
torch.Size([8, 69])
```

Powyższy wynik pokazuje, że pierwsza partia wejściowa i docelowa mają wymiary 8×61 , gdzie 8 reprezentuje rozmiar partii, a 61 to liczba tokenów w każdym przykładzie szkoleniowym w tej partii. Druga partia wejściowa i docelowa mają inną liczbę tokenów – w tym przykładzie 76. Dzięki niestandardowej funkcji `collate` obiekt ładujący dane potrafi tworzyć partie o różnej długości. W następnym podrozdziale załadowujemy wstępnie przeszkolony model LLM, który następnie będzie można dostroić z użyciem powyższego obiektu ładującego dane.

7.5. Ładowanie wstępnie przeszkolonego modelu LLM

Poświęciliśmy dużo czasu na przygotowanie zbioru danych do dostrajania modelu LLM do wykonywania instrukcji, co jest kluczowym elementem nadzorowanego procesu dostrajania. Wiele innych elementów jest takich samych jak w szkoleniu wstępny, co pozwala ponownie wykorzystać większość kodu z wcześniejszych rozdziałów.

Przed rozpoczęciem dostrajania do zadań wykonywania instrukcji trzeba załadować wstępnie przeszkolony model GPT, który chcemy dostroić (patrz rysunek 7.15). Ten proces przeprowadzaliśmy już wcześniej. Jednak teraz, zamiast używać, jak



Rysunek 7.15. Trzyetapowy proces dostrajania modelu LLM do zadań wykonywania instrukcji.
Po przygotowaniu zbioru danych proces dostrajania modelu LLM do zadań wykonywania instrukcji rozpoczyna się od załadowania wstępnie przeszkołonego LLM, który będzie służyć jako podstawa do późniejszego szkolenia

poprzednio, najmniejszego modelu, o 124 milionach parametrów, ładujemy średniej wielkości model, o 355 milionach parametrów.

Powodem tego wyboru jest to, że 124-milionowy model ma zbyt ograniczoną pojemność, aby dostrajanie do zadań wykonywania instrukcji pozwoliło osiągnąć zadawalające wyniki. W szczególności mniejszym modelom brakuje niezbędnych zdolności do uczenia się oraz zachowywania skomplikowanych wzorców i znuansowanych zachowań niezbędnych do realizacji wysokiej jakości zadań polegających na wykonywaniu instrukcji.

Załadowanie wstępnie przeszkołonych modeli wymaga tego samego kodu, co podczas wstępnego szkolenia danych (podrozdział 5.5) i dostrajania ich do zadań klasyfikacji (podrozdział 6.4), z wyjątkiem tego, że teraz zamiast modelu "gpt2-small (124M)" użyjemy modelu "gpt2-medium (355M)" (listing 7.7).

Listing 7.7. Ładowanie wstępnie przeszkołonego modelu

```

from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

BASE_CONFIG = {
    "model_type": "gpt2",
    "model_name": "gpt2-medium"
}

```

```

"vocab_size": 50257,      # Rozmiar słownictwa
"context_length": 1024,  # Rozmiar kontekstu
"drop_rate": 0.0,        # Współczynnik dropoutu
"qkv_bias": True         # Stronniczość zapytanie klucz-wartość
}
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

CHOOSE_MODEL = "gpt2-medium (355M)"
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")

settings, params = download_and_load_gpt2(
    model_size=model_size,
    models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval();

```

UWAGA Wykonanie tego kodu zainicjuje pobieranie średniej wielkości modelu GPT, co wymaga około 1,42 gigabajta miejsca na dysku. Jest to około trzech razy więcej od przestrzeni dyskowej wymaganej dla małego modelu.

Uruchomienie powyższego kodu spowoduje pobranie kilku plików:

```

checkpoint: 100%|██████████| 77.0/77.0 [00:00<00:00, 156kiB/s]
encoder.json: 100%|██████████| 1.04M/1.04M [00:02<00:00, 467kiB/s]
hparams.json: 100%|██████████| 91.0/91.0 [00:00<00:00, 198kiB/s]
model.ckpt.data-00000-of-00001: 100%|██████████| 1.42G/1.42G [05:50<00:00,
4.05MiB/s]
model.ckpt.index: 100%|██████████| 10.4k/10.4k [00:00<00:00, 18.1MiB/s]
model.ckpt.meta: 100%|██████████| 927k/927k [00:02<00:00, 454kiB/s]
vocab.bpe: 100%|██████████| 456k/456k [00:01<00:00, 283kiB/s]

```

Poświęćmy teraz chwilę na ocenę wydajności wstępnie przeszkolonego modelu LLM w jednym z zadań walidacyjnych przez porównanie uzyskanych wyników z oczekiwanaą odpowiedzią. Dzięki temu zyskasz podstawową wiedzę na temat tego, jak dobrze model radzi sobie z zadaniem wykonywania instrukcji „powyjęciu z pudełka”, jeszcze przed dostrojeniem, co pomoże Ci docenić efekt późniejszego dostrojenia. Do tej oceny wykorzystamy pierwszy przykład ze zbioru walidacyjnego:

```

torch.manual_seed(123)
input_text = format_input(val_data[0])
print(input_text)

```

Treść instrukcji jest następująca:

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instrukcja:

Convert the active sentence to passive: 'The chef cooks the meal every day.'

W kolejnym kroku generujemy odpowiedź modelu z użyciem tej samej funkcji gene →rate, której użyliśmy w rozdziale 5. do wstępnie szkolenia modelu:

```
from chapter05 import generate, text_to_token_ids, token_ids_to_text

token_ids = generate(
    model=model,
    idx=text_to_token_ids(input_text, tokenizer),
    max_new_tokens=35,
    context_size=BASE_CONFIG["context_length"],
    eos_id=50256,
)
generated_text = token_ids_to_text(token_ids, tokenizer)
```

Funkcja generate zwraca połączony tekst wejściowy i wynikowy. Takie zachowanie było wcześniej wygodne, ponieważ wstępnie przeszkolone modele LLM są przede wszystkim zaprojektowane jako modele uzupełniania tekstu, których zadaniem jest łączenie danych wejściowych z wynikami w celu utworzenia spójnego i czytelnego tekstu. Jednak gdy oceniamy wydajność modelu w konkretnym zadaniu, często chcemy się skupić wyłącznie na wygenerowanej przez niego odpowiedzi.

Aby wyodrębnić tekst odpowiedzi modelu, trzeba odjąć długość instrukcji wejściowej od początku wygenerowanego tekstu:

```
response_text = generated_text[len(input_text):].strip()
print(response_text)
```

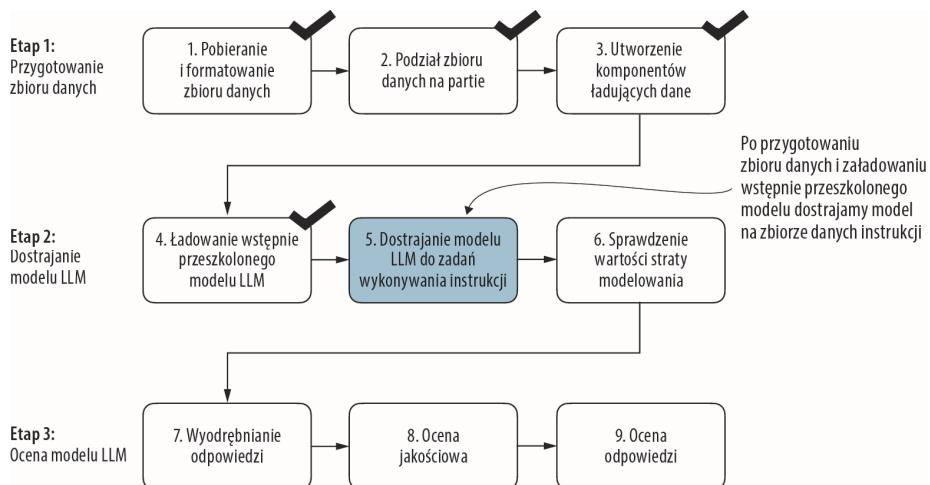
Powyższy kod usuwa tekst wejściowy z początku zmiennej generated_text i pozostawia tylko wygenerowaną odpowiedź modelu. Następnie, w celu usunięcia wszelkiego rodzaju początkowych lub końcowych białych znaków, stosujemy funkcję strip(). Oto uzyskany wynik:

```
### Odpowiedź:
The chef cooks the meal every day.
### Instrukcja:
Convert the active sentence to passive: 'The chef cooks the
```

Na podstawie tego wyniku widać, że wstępnie przeszkolony model jeszcze nie jest w stanie poprawnie wykonać przekazanej instrukcji. Chociaż tworzy sekcję odpowiedzi (Response), to jedynie powtarza oryginalne zdanie wejściowe i część instrukcji i nie konwertuje zdania w stronę czynnej na stronę bierną, jak nakazano mu w żądaniu. W kolejnym podrozdziale zaimplementujemy więc proces dostrajania, którego celem będzie poprawienie zdolności modelu do rozumienia podobnych żądań i odpowiedniego reagowania na nie.

7.6. Dostrajanie modeli LLM do zadań wykonywania instrukcji

Nadszedł czas, aby dostroić model LLM do zadań wykonywania instrukcji (rysunek 7.16). Skorzystamy z załadowanego w poprzednim podrozdziale wstępnie przeszkolonego modelu i będziemy kontynuować jego szkolenie z użyciem przygotowanego wcześniej w tym rozdziale zbioru danych instrukcji.



Rysunek 7.16. Trzyetapowy proces dostrajania modelu LLM do zadań wykonywania instrukcji. W kroku 5. szklimy załadowany uprzednio wstępnie przeszkolony model na przygotowanym wcześniej zbiorze danych instrukcji

Całą ciężką pracę wykonaliśmy na początku tego rozdziału, podczas implementacji przetwarzania zbioru danych instrukcji. We właściwym procesie dostrajania można ponownie wykorzystać funkcje obliczania strat i uczenia zaimplementowane w rozdziale 5.:

```
from chapter05 import (
    calc_loss_loader,
    train_model_simple
)
```

Przed przystąpieniem do szkolenia obliczamy początkową strategię dla zbiorów szkoleniowego i walidacyjnego:

```
model.to(device)
torch.manual_seed(123)

with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
```

```

    val_loss = calc_loss_loader(
        val_loader, model, device, num_batches=5
    )

    print("Strata zbioru szkoleniowego:", train_loss)
    print("Strata zbioru walidacyjnego:", val_loss)

```

Początkowe wartości strat zamieszczone poniżej. Tak jak poprzednio, naszym celem jest zminimalizowanie strat:

Strata zbioru szkoleniowego: 3.825908660888672
 Strata zbioru walidacyjnego: 3.7619335651397705

Radzenie sobie z ograniczeniami sprzętowymi

Korzystanie z większego modelu, takiego jak GPT-2 medium (355 milionów parametrów), i szkolenie go, jest bardziej wymagające obliczeniowo w porównaniu z mniejszym modelem GPT-2 (124 miliony parametrów). W razie napotkania problemów związanych z ograniczeniami sprzętowymi możesz przełączyć się na mniejszy model. W tym celu wystarczy zmienić ustawienie CHOOSE_MODEL – 'gpt2-medium (355M)' na CHOOSE_MODEL – 'gpt2-small (124M)' (patrz podrozdział 7.5). Aby przyspieszyć szkolenie modelu, możesz również przełączyć się na korzystanie z układu GPU. Listę kilku opcji wykorzystania układów GPU w chmurze zamieściłem w następującej sekcji uzupełniającej repozytorium kodu tej książki: <https://mng.bz/EOEq>.

W poniższej tabeli zamieszczone są referencyjną listę czasów wymaganych do szkolenia każdego z wymienionych modeli GPT-2 na różnych urządzeniach, w tym procesorach CPU i GPU. Uruchomienie prezentowanego kodu na kompatybilnym procesorze graficznym nie wymaga żadnych zmian w kodzie i może znacznie przyspieszyć szkolenie. Do uzyskania wyników przedstawionych w tym rozdziale wykorzystałem średni model GPT-2 i przeszkoliłem go na układzie GPU A100.

Nazwa modelu	Urządzenie	Czas działania dla dwóch epok
gpt2-medium (355 mln parametrów)	CPU (M3 MacBook Air)	15,78 minuty
gpt2-medium (355 mln parametrów)	GPU (Nvidia L4)	1,83 minuty
gpt2-medium (355 mln parametrów)		0,86 minuty
gpt2-small (124 mln parametrów)	CPU (M3 MacBook Air)	5,74 minuty
gpt2-small (124 mln parametrów)	GPU (Nvidia L4)	0,69 minuty
gpt2-small (124 mln parametrów)	GPU (Nvidia A100)	0,39 minuty

Po przygotowaniu modelu i obiektów ładujących dane można przystąpić do szkolenia modelu. Kod na listingu 7.8 konfiguruje proces uczenia, w tym inicjuje optymalizator, ustawia liczbę epok oraz określa częstotliwość oceny i kontekst początkowy do oceny wygenerowanych odpowiedzi modelu LLM podczas uczenia, na podstawie pierwszej instrukcji zbioru walidacyjnego (`val_data[0]`), której przyjrzaliśmy się w podrozdziale 7.5.

Listing 7.8. Dostrajanie modelu LLM do zadań wykonywania instrukcji

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(
    model.parameters(), lr=0.00005, weight_decay=0.1
)
num_epochs = 2

train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context=format_input(val_data[0]), tokenizer=tokenizer
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Szkolenie zakończono w ciągu {execution_time_minutes:.2f} minut.")
```

Poniższe wyniki przedstawiają postęp szkolenia w dwóch epokach. Spadek funkcji straty wskazuje na poprawę zdolności do wykonywania instrukcji i generowania poprawnych odpowiedzi:

```
Epoka 1 (Krok 000000): Strata zbioru szkoleniowego 2.727, Strata zbioru walidacyjnego  
→ 2.692
Epoka 1 (Krok 000005): Strata zbioru szkoleniowego 1.035, Strata zbioru walidacyjnego  
→ 0.981
Epoka 1 (Krok 000010): Strata zbioru szkoleniowego 0.759, Strata zbioru walidacyjnego  
→ 0.824
Epoka 1 (Krok 000015): Strata zbioru szkoleniowego 0.754, Strata zbioru walidacyjnego  
→ 0.789
Epoka 1 (Krok 000020): Strata zbioru szkoleniowego 0.674, Strata zbioru walidacyjnego  
→ 0.764
Epoka 1 (Krok 000025): Strata zbioru szkoleniowego 0.660, Strata zbioru walidacyjnego  
→ 0.740
...
Epoka 1 (Krok 000115): Strata zbioru szkoleniowego 0.436, Strata zbioru walidacyjnego  
→ 0.565
Below is an instruction that describes a task. Write a response that appropriately  
→ completes the request. ### Instrukcja: Convert the active sentence to passive:  
→ 'The chef cooks the meal every day.' ### Odpowiedź: The chef cooks the meal every  
→ day.<|endoftext|>The following is an instruction that describes a task. Write  
→ a response that appropriately completes the request. ### Wejście:  
Epoka 2 (Krok 000120): Strata zbioru szkoleniowego 0.372, Strata zbioru walidacyjnego  
→ 0.570
Epoka 2 (Krok 000125): Strata zbioru szkoleniowego 0.386, Strata zbioru walidacyjnego  
→ 0.587
Epoka 2 (Krok 000130): Strata zbioru szkoleniowego 0.382, Strata zbioru walidacyjnego  
→ 0.584
Epoka 2 (Krok 000135): Strata zbioru szkoleniowego 0.351, Strata zbioru walidacyjnego  
→ 0.582
Epoka 2 (Krok 000140): Strata zbioru szkoleniowego 0.354, Strata zbioru walidacyjnego  
→ 0.582
```

Epoka 2 (Krok 000145): Strata zbioru szkoleniowego 0.315, Strata zbioru walidacyjnego
 $\rightarrow 0.581$

...
 Epoka 2 (Krok 000230): Strata zbioru szkoleniowego 0.248, Strata zbioru walidacyjnego
 $\rightarrow 0.549$

Below is an instruction that describes a task. Write a response that appropriately
 \rightarrow completes the request. ### Instrukcja: Convert the active sentence to passive:
 \rightarrow 'The chef cooks the meal every day.' ### Odpowiedź: The meal is cooked every day
 \rightarrow by the chef.<|endoftext|>The following is an instruction that describes a task.
 \rightarrow Write a response that appropriately completes the request. ### Odpowiedź

Szkolenie zakończono w czasie 1.00 minut.

Wyniki szkolenia pokazują, że model uczy się skutecznie, co można stwierdzić na podstawie konsekwentnie malejących wartości funkcji straty zbiorów szkoleniowego i walidacyjnego na przestrzeni dwóch epok. Ten wynik sugeruje, że model stopniowo poprawia swoją zdolność rozumienia dostarczonych instrukcji i postępowania według nich (ponieważ model wykazał skuteczność uczenia się w ciągu tych dwóch epok, rozszerzenie szkolenia na trzecią epokę lub dalsze w istocie nie jest konieczne, a nawet może przynieść efekt przeciwny do zamierzonego, ponieważ może prowadzić do nasielenia się problemu nadmiernego dopasowania).

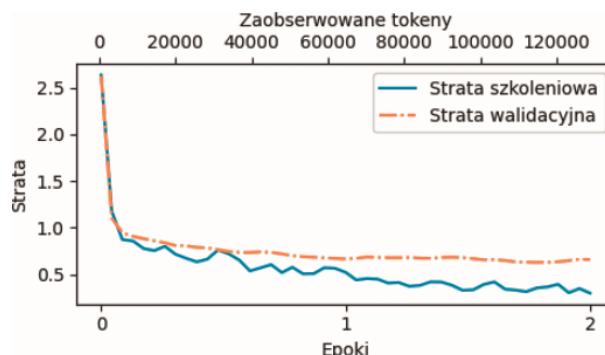
Co więcej, odpowiedzi wygenerowane na koniec każdej epoki pozwalają sprawdzić postępy modelu w poprawnym wykonywaniu danego zadania w przykładzie ze zbioru walidacyjnego. W tym przypadku model z powodzeniem przekształca zdanie w stronie czynnej „The chef cooks the meal every day” (szef kuchni codziennie gotuje posiłek) na jego odpowiednik w stronie biernej: „The meal is cooked every day by the chef” (posiłek jest przygotowywany każdego dnia przez szefa kuchni).

W dalszej części rozdziału ponownie przyjrzymy się odpowiedziom modelu i bardziej szczegółowo ocenimy ich jakość. Na razie, aby zyskać dodatkowy pogląd na proces uczenia się modelu, przeanalizujmy krzywe strat zbiorów szkoleniowego i walidacyjnego. W tym celu używamy tej samej funkcji `plot_losses`, której używaliśmy do wstępniego szkolenia:

```
from chapter05 import plot_losses
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

Na podstawie wykresu funkcji straty pokazanego na rysunku 7.17 można wywnioskować, że w trakcie szkolenia wydajność modelu, zarówno na zbiorach szkoleniowych, jak i walidacyjnych, znacznie się poprawia. Szybki spadek funkcji straty w początkowej fazie wskazuje, że model szybko uczy się z danych znaczących wzorców i reprezentacji. Następnie, w miarę postępu szkolenia do drugiej epoki, straty nadal maleją, ale w wolniejszym tempie, co sugeruje, że model dostraja swoje wyuczone reprezentacje i zbliża się do stabilnego rozwiązania.

Podczas gdy wykres funkcji straty z rysunku 7.17 wskazuje, że model skutecznie się uczy, najważniejszą cechą jest jego wydajność w kategoriach jakości i poprawności odpowiedzi. W związku z tym w następnym kroku wyodrębnimy odpowiedzi i zapiszemy je w formacie, który pozwoli ocenić jakość odpowiedzi i ująć je ilościowo.



Rysunek 7.17. Trendy funkcji straty zbiorów szkoleniowego i walidacyjnego na przestrzeni dwóch epok. Linia ciągła reprezentuje stratę zbioru szkoleniowego. Widać gwałtowny spadek, a następnie stabilizację, podczas gdy linia przerywana reprezentuje stratę zbioru walidacyjnego, która zmienia się zgodnie z podobnym wzorcem

Ćwiczenie 7.3. Dostrajanie oryginalnego zbioru danych Alpaca

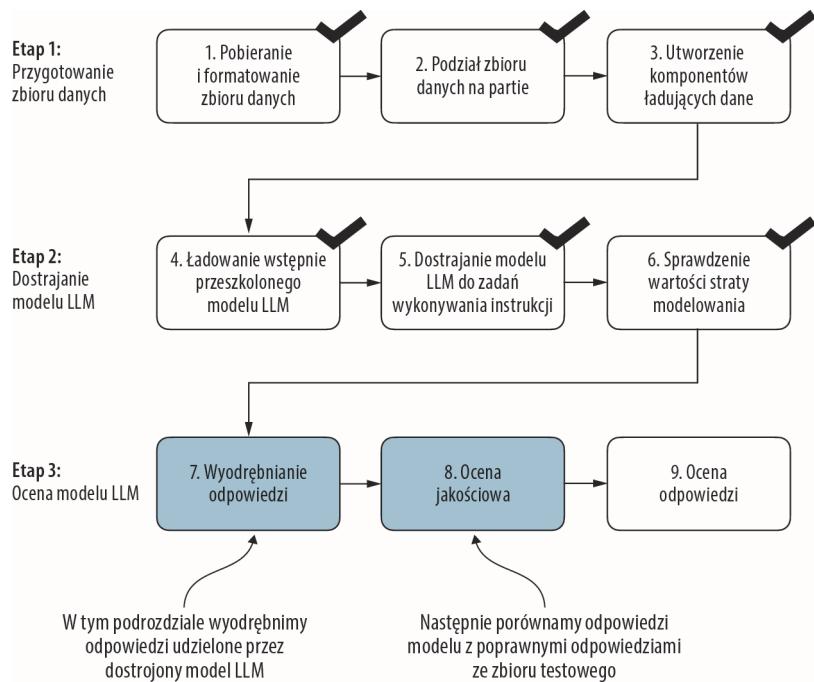
Zbiór danych Alpaca, opracowany przez naukowców z Uniwersytetu Stanforda, jest jednym z najwcześniejszych i najpopularniejszych zbiorów danych instrukcji, składającym się z 52 002 pozycji. Jako alternatywę dla pliku *instruction-data.json*, którego użyłem w prezentowanym przykładzie, warto rozważyć dostrojenie modelu LLM na zbiorze danych Alpaca. Zbiór danych jest dostępny pod adresem <https://mng.bz/NBnE>.

Zawiera on 52 002 pozycje, czyli około 50 razy więcej niż zbiór wykorzystany w przykładzie. Ponadto większość pozycji to przykłady bardziej rozbudowane. W związku z tym zdecydowanie zalecam wykorzystanie do przeprowadzenia szkolenia układu GPU, co powinno znacznie przyspieszyć proces dostrajania. Jeśli napotkasz błędy braku pamięci, rozważ zmniejszenie wartości parametru `batch_size` z 8 do 4, 2 lub nawet 1. Obniżenie wartości `allowed_max_length` z 1024 do 512 lub 256 również może pomóc w obsłudze problemów z pamięcią.

7.7. Wyodrębnianie i zapisywanie odpowiedzi

Po dostrojeniu modelu LLM na szkoleniowej części zbioru danych instrukcji można przystąpić do oceny jego wydajności na wyznaczonym zbiorze testowym. Najpierw wyodrębniamy wygenerowane przez model odpowiedzi dla każdego wejścia w testowym zbiorze danych i zbieramy je do ręcznej analizy, a następnie oceniamy model LLM w celu ilościowego określenia jakości odpowiedzi (rysunek 7.18).

Do realizacji kroku odpowiedzi na instrukcje używamy funkcji `generate`. Następnie wyświetlamy odpowiedzi modelu wraz z oczekiwaniymi odpowiedziami ze zbioru testowego dla pierwszych trzech pozycji zbioru testowego i prezentujemy je obok siebie w celu porównania:



Rysunek 7.18. Trzyetapowy proces dostrajania modelu LLM do wykonywania instrukcji. W pierwszych dwóch krokach etapu 3. wyodrębniamy i zbieramy odpowiedzi modelu na wyznaczonym zbiorze danych testowych do dalszej analizy, a następnie oceniamy model w celu ilościowego określenia wydajności modelu LLM dostrojonego do wykonywania instrukcji

```

torch.manual_seed(123)

for entry in test_data[:3]:
    input_text = format_input(entry)
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)

    response_text = (
        generated_text[len(input_text):]
        .replace("### Odpowiedź:", "")
        .strip()
    )
    print(input_text)
    print(f"\nPoprawna odpowiedź:\n>> {entry['output']}\"")
    print(f"\nOdpowiedź modelu:\n>> {response_text.strip()}\"")
    print("-----")

```

Iteracja po pierwszych trzech próbkach zbioru testowego

Wykorzystanie funkcji generate zaimportowanej w podrozdziale 7.5

Jak wspomniałem wcześniej, funkcja generate zwraca połączony tekst wejściowy i wyjściowy. Zatem aby wyodrębnić odpowiedź modelu, używamy mechanizmu wyznaczania wycinków tekstu i metody `.replace()` na zawartości zmiennej `generated ↪_text`. Instrukcje, po których następuje odpowiedź ze zbioru testowego i odpowiedź modelu, pokazalem poniżej.⁶

Below is an instruction that describes a task. Write a response that appropriately completes the request. (Poniżej znajdziesz instrukcję opisującą zadanie. Napisz odpowiedź, która je wykonuje).

Instrukcja:

Rewrite the sentence using a simile. (Przepisz zdanie, używając porównania).

Wejście:

The car is very fast. (Samochód jest bardzo szybki).

Poprawna odpowiedź:

>> The car is as fast as lightning. (Samochód jest szybki jak błyskawica).

Odpowiedź modelu:

>> The car is as fast as a bullet. (Samochód jest szybki jak pocisk).

Below is an instruction that describes a task. (Poniżej znajduje się instrukcja opisująca zadanie). Write a response that appropriately completes the request. (Napisz odpowiedź, która je wykonuje).

Instrukcja:

What type of cloud is typically associated with thunderstorms? (Jaki rodzaj chmur zwykle wiąże się z burzami?)

Poprawna odpowiedź:

>> The type of cloud typically associated with thunderstorms is cumulonimbus. (Rodzajem chmury typowo związany z burzami jest cumulonimbus).

Odpowiedź modelu:

>> The type of cloud associated with thunderstorms is a cumulus cloud. (Rodzajem chmury związanej z burzami jest chmura cumulus).

⁶ W nawiasach podano tłumaczenie instrukcji modelu oraz udzielonych odpowiedzi na język polski – *przyp. tłum.*

Below is an instruction that describes a task. (Poniżej znajduje się instrukcja opisująca zadanie). Write a response that appropriately completes the request. (Napisz odpowiedź, która je wykonuje).

Instrukcja:

Name the author of ‘Pride and Prejudice’. (Wymień autora „Dumy i uprzedzenia”).

Poprawna odpowiedź:

>> Jane Austen.

Odpowiedź modelu:

>> The author of ‘Pride and Prejudice’ is Jane Austen. (Autorką „Dumy i uprzedzenia” jest Jane Austen).

Jak można zauważyć na podstawie instrukcji w zbiorze testowym, poprawnych odpowiedzi i odpowiedzi modelu, model radzi sobie stosunkowo dobrze. Odpowiedzi na pierwsze i ostatnie pytanie wyraźnie są poprawne, podczas gdy druga odpowiedź jest bliska poprawnej, ale nie jest dokładna. Model odpowiedział „chmura cumulus” zamiast „cumulonimbus”, choć warto zauważyć, że chmury cumulus mogą się przekształcić w zdolne do wywołania burz chmury cumulonimbus.

Co najważniejsze, ocena modelu nie jest tak prosta jak w przypadku dostrajania pod kątem zadań klasyfikacji, w których aby uzyskać dokładność klasyfikacji, po prostu obliczamy procent poprawnych etykiet klas „spam” bądź „nie spam”. W praktyce istnieje wiele sposobów oceny modeli LLM dostrojonych do wykonywania instrukcji, takich jak chatboty:

- testy porównawcze krótkich odpowiedzi i odpowiedzi wielokrotnego wyboru, takie jak test MMLU (ang. *Measuring Massive Multitask Language Understanding* – <https://arxiv.org/abs/2009.03300>), które sprawdzają ogólną wiedzę o modelu,
- porównanie preferencji użytkowników w zestawieniu z innymi modelami LLM (np. na platformie LMSYS – <https://arena.lmsys.org>),
- zautomatyzowane testy konwersacyjne, w których do oceny odpowiedzi używany jest inny model LLM, taki jak GPT-4, na przykład AlpacaEval (https://tatsu-lab.github.io/alpaca_eval/).

W praktyce warto rozważyć wszystkie trzy rodzaje metod oceny: odpowiadanie na pytania wielokrotnego wyboru, ocenę przez człowieka i zautomatyzowane wskaźniki mierzące wydajność konwersacji. Ponieważ jednak jesteśmy przede wszystkim zainteresowani oceną wydajności konwersacji, a nie tylko umiejętnością odpowiadania na pytania wielokrotnego wyboru, najbardziej odpowiednie wydają się ocena dokonana przez ludzi oraz automatyczne wskaźniki jakości konwersacji.

Wydajność konwersacyjna

Wydajność konwersacji modeli LLM odnosi się do ich zdolności do komunikowania się w sposób podobny do ludzkiego w zakresie rozumienia kontekstu, istniejących niuansów i intencji. Obejmuje to takie umiejętności jak udzielanie trafnych i spójnych odpowiedzi, utrzymanie spójności oraz dostosowywanie się do różnych tematów i stylów interakcji.

Ocena dokonywana przez ludzi, choć dostarcza cennych informacji, może być stosunkowo pracochłonna i czasochłonna, zwłaszcza w przypadku dużej liczby odpowiedzi. Na przykład przeczytanie ze zbioru testowego wszystkich 1100 odpowiedzi i przypisanie do nich ocen wymagałoby znacznego wysiłku.

W związku z tym, biorąc pod uwagę skalę zadania, zaimplementujemy podejście przypominające zautomatyzowane wskaźniki jakości konwersacji, które obejmują automatyczną ocenę odpowiedzi z użyciem innego modelu LLM. Ta metoda umożliwia skutecną ocenę jakości odpowiedzi wygenerowanych przez model bez konieczności dużego zaangażowania człowieka, co pozwoli oszczędzić czas i zasoby, a zarazem uzyskać sensowne wskaźniki wydajności.

Spróbujmy zastosować podejście zainspirowane narzędziem AlpacaEval — wykorzystanie do oceny odpowiedzi dostrojonego modelu LLM innego modelu LLM. Zamiast jednak polegać na publicznie dostępnym zbiorze danych, użyjemy wyodrębnionego zbioru testowego. To dostrojenie pozwoli na bardziej ukierunkowaną i lepszą ocenę wydajności modelu w kontekście zamierzonych przypadków użycia, reprezentowanych w zastosowanym zbiorze danych instrukcji.

Aby przygotować odpowiedzi do procesu oceny, dołączymy wygenerowane odpowiedzi modelu do słownika `test_set` i zapiszemy zaktualizowane dane w pliku `instruction-data-with-response.json` do późniejszego wykorzystania. Dodatkowo dzięki zapisaniu tego pliku możemy później w razie potrzeby łatwo wczytać i przeanalizować odpowiedzi w osobnych sesjach Pythona.

Poniższy kod wykorzystuje metodę `generate` w taki sam sposób jak poprzednio, jednak teraz iteruje po całym zbiorze `test_set`. Ponadto, zamiast wyświetlać odpowiedzi modelu, dodajemy je do słownika `test_set` (listing 7.9).

Listing 7.9. Generowanie odpowiedzi dla zbioru testowego

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)

    token_ids = generate(
        model=model, SS
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
```

```

generated_text = token_ids_to_text(token_ids, tokenizer)

response_text = (
    generated_text[len(input_text):]
    .replace("### Odpowiedź:", "")
    .strip()
)
test_data[i]["model_response"] = response_text

with open("instruction-data-with-response.json", "w") as file: ← wcięcie w celu
    json.dump(test_data, file, indent=4) ← poprawy estetyki

```

Przetwarzanie zbioru danych na układzie GPU A100 zajmuje około 1 minuty, a na MacBooku Air M3 6 minut:

```
100%|██████████| 110/110 [01:05<00:00, 1.68it/s]
```

Sprawdźmy na podstawie analizy jednego z wpisów, czy odpowiedzi zostały poprawnie dodane do słownika test_set:

```
print(test_data[0])
```

Wyniki wskazują, że wartość model_response została dodana poprawnie:

```
{
    'instruction': 'Rewrite the sentence using a simile.',
    'input': 'The car is very fast.',
    'output': 'The car is as fast as lightning.',
    'model_response': 'The car is as fast as a bullet.'}
```

Na koniec, aby móc ponownie wykorzystać model w przyszłych projektach, zapiszemy go w pliku *gpt2-medium355M-sft.pth*.

```

import re

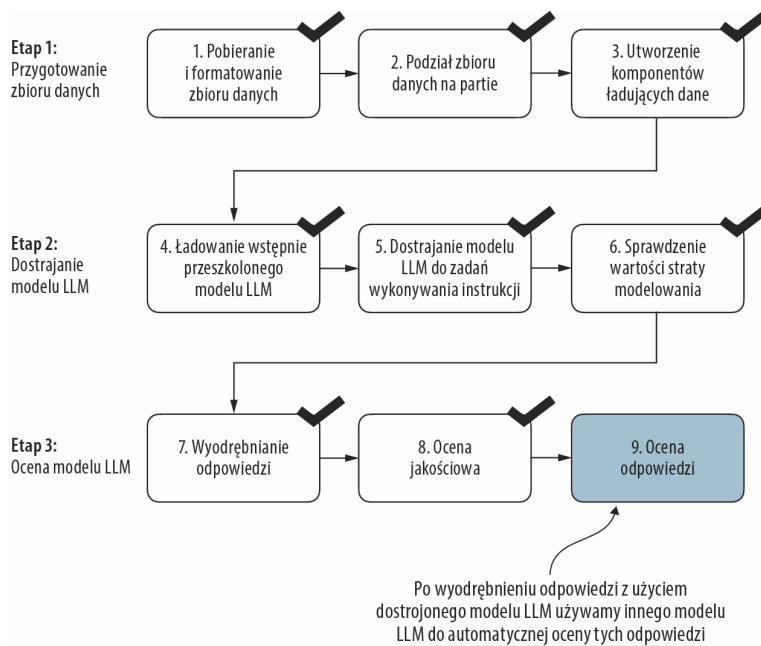
file_name = f"{re.sub(r'[ ]', '', CHOOSE_MODEL)}-sft.pth" ← Usunięcie z nazwy pliku
    torch.save(model.state_dict(), file_name) ← białych spacji i nawiasów
print(f"Model zapisany w pliku {file_name}")

```

Zapisany model można następnie załadować za pomocą instrukcji `model.load_state_dict(torch.load('gpt2-medium355M-sft.pth'))`.

7.8. Ocena dostrojonego modelu LLM

Wcześniej ocenialiśmy wydajność modelu dostrojonego do wykonywania instrukcji na podstawie analizy jego odpowiedzi dla trzech przykładów ze zbioru testowego. Chociaż zastosowanie tego podejścia daje przybliżone pojęcie o tym, jak dobrze działa model, metoda ta nie skaluje się dobrze do większej liczby odpowiedzi. W związku z tym zaimplementowaliśmy metodę automatyzacji oceny odpowiedzi dostrojonego modelu LLM z wykorzystaniem innego, większego modelu LLM, tak jak pokazałem na rysunku 7.19.



Rysunek 7.19. Trzyetapowy proces dostrajania modelu LLM do wykonywania instrukcji.
W ostatnim kroku potoku dostrajania do wykonywania instrukcji implementujemy metodę ilościowego określania wydajności dostrojonego modelu przez przypisanie oceny odpowiedzi wygenerowanej dla próbki testowej

Aby automatycznie ocenić odpowiedzi zbioru testowego, wykorzystujemy opracowany przez Meta AI model Llama 3 o 8 miliardach parametrów, dostrojony do wykonywania instrukcji. Model ten można uruchomić lokalnie za pomocą aplikacji open source Ollama (<https://ollama.com>).

UWAGA Ollama to wydajna aplikacja do uruchamiania modeli LLM na laptopie. Służy jako wrapper biblioteki open source *llama.cpp* (<https://github.com/ggerganov/llama.cpp>), która w celu zmaksymalizowania wydajności implementuje model LLM w czystym kodzie C/C++. Ollama jest jednak wyłącznie narzędziem do generowania tekstu z użyciem modeli LLM (wnioskowanie) i nie obsługuje szkolenia ani dostrajania modeli LLM.

Aby uruchomić poniższy kod, zainstaluj Ollama. W tym celu odwiedź stronę <https://ollama.com> i postępuj zgodnie z instrukcjami dla swojego systemu operacyjnego:

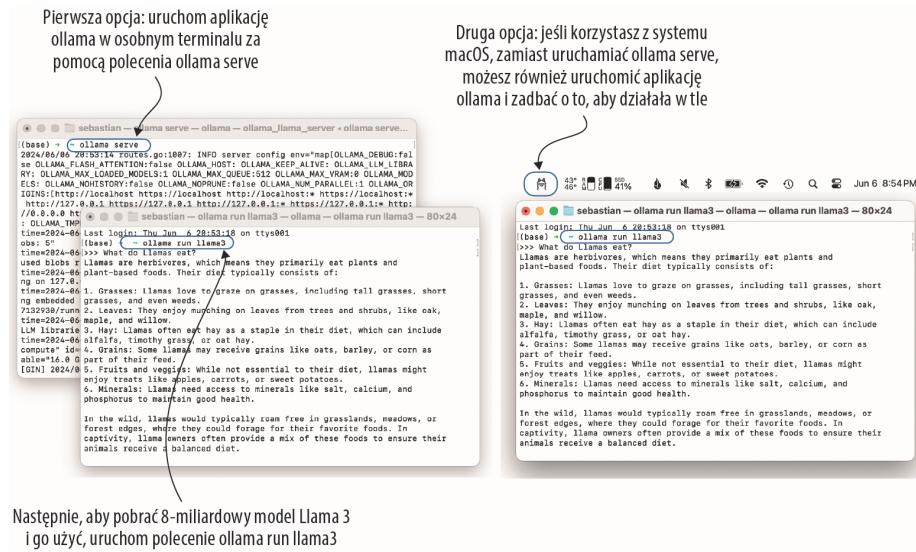
- *Użytkownicy systemów macOS i Windows* – otwórz pobraną aplikację Ollama. Po wyświetleniu monitu o zainstalowanie narzędzia wiersza poleceń wybierz Yes.
- *Użytkownicy systemu Linux* – użyj polecenia instalacji dostępnego na stronie internetowej Ollama.

Przed zaimplementowaniem kodu odpowiedzialnego za ocenę modelu najpierw pobierzemy model Llama 3 i sprawdzimy, czy aplikacja Ollama działa poprawnie

Korzystanie z większych modeli LLM za pośrednictwem webowych interfejsów API

Model Llama 3 o 8 miliardach parametrów to bardzo wydajny LLM działający lokalnie. Nie jest on jednak tak wydajny jak duże, zastrzeżone modele LLM, takie jak oferowany przez OpenAI GPT-4. Czytelnicy, którzy są zainteresowani zbadaniem wykorzystania modelu GPT-4 za pośrednictwem interfejsu API OpenAI do oceny wygenerowanych odpowiedzi, mogą skorzystać z notatnika z kodem dostępnego w materiałach uzupełniających dołączonych do tej książki, pod adresem <https://mng.bz/BgEv>.

z poziomu terminala. Aby skorzystać z Ollamy z wiersza poleceń, należy uruchomić aplikację Ollama lub uruchomić polecenie `ollama serve` w osobnym terminalu tak, jak pokazałem na rysunku 7.20.



Następnie, aby pobrać 8-miliardowy model Llama 3 i go użyć, uruchom polecenie `ollama run llama3`

Rysunek 7.20. Dwie opcje uruchamiania aplikacji Ollama. Lewy panel ilustruje uruchomienie Ollamy z użyciem polecenia `ollama serve`. Prawy panel pokazuje drugą opcję w systemie macOS, polegającą na uruchomieniu aplikacji Ollama w tle zamiast użycia polecenia do uruchomienia aplikacji `ollama serve`

Po uruchomieniu aplikacji Ollama lub zainicjowaniu instrukcji `ollama serve` w innym terminalu, aby wypróbować model Llama 3 o 8 miliardach parametrów, wykonaj w wierszu poleceń (nie w sesji Pythona) następujące polecenie:

```
ollama run llama3
```

Przy pierwszym wykonaniu tego polecenia zostanie automatycznie pobrany model. Do pobrania potrzeba 4,7 GB przestrzeni dyskowej. Wynik działania kodu wyglądają następująco:

```
pulling manifest
pulling 6a0746a1ec1a... 100% | [██████████] | 4.7 GB
pulling 4fa551d4f938... 100% | [██████████] | 12 KB
pulling 8ab4849b038c... 100% | [██████████] | 254 B
pulling 577073ffcc6c... 100% | [██████████] | 110 B
pulling 3f8eb4da87fa... 100% | [██████████] | 485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```

Alternatywa dla modeli Ollama

llama3 w poleceniu `ollama run llama3` odnosi się do modelu Llama 3 o 8 miliardach parametrów, dostrojonego do wykonywania instrukcji. Korzystanie z aplikacji Ollama z modelem llama3 wymaga około 16 GB pamięci RAM. Jeśli komputer nie ma wystarczającej ilości pamięci RAM, można spróbować użyć mniejszego modelu, na przykład phi3 o 3,8 miliarda parametrów, który wymaga tylko około 8 GB pamięci RAM. Aby go uruchomić, użyj polecenia `ollama run phi3`.

W przypadku mocniejszych komputerów można również użyć większego, 70-miliardowego modelu Llama 3. W tym celu należy zastąpić nazwę modelu `llama3` nazwą ze wskazaniem liczby parametrów: `llama3:70b`. Ten ostatni model wymaga jednak znacznie więcej zasobów obliczeniowych.

Po zakończeniu pobierania modelu wyświetlany jest interfejs wiersza poleceń, który umożliwia interakcję z modelem. Dla przykładu, spróbuj zapytać model: „Co jedzą lamy?”.

```
>>> What do llamas eat?
Llamas are ruminant animals, which means they have a four-chambered stomach and eat
→plants that are high in fiber. In the wild, llamas typically feed on:
1. Grasses: They love to graze on various types of grasses, including tall grasses,
wheat, oats, and barley.
```

W tłumaczeniu na język polski:

```
>>> Co jedzą lamy?
Lamy są przeżuwaczami, co oznacza, że mają czterokomorowy żołądek i jedzą rośliny
→bogate w błonnik. Na wolności lamy zazwyczaj żywią się:
1. Trawy: Uwielbiają wypasać się na różnego rodzaju trawach, w tym na wysokich
→trawach, pszenicy, owsie i jęczmieniu.
```

Należy pamiętać, że odpowiedź, której udzielił model, może się różnić od tej pokazanej powyżej, ponieważ w chwili, gdy pisałem te słowa, odpowiedzi aplikacji Ollama nie były deterministyczne.

Aby zakończyć sesję `ollama run llama3`, wpisz polecenie `/bye`. Aby jednak śledzić pozostałe przykłady w dalszej części tego rozdziału, polecenie `ollama serve` lub aplikacja Ollama powinny pozostać uruchomione.

Aby użyć Ollamy do oceny odpowiedzi zbioru testowego, można za pomocą poniższego kodu sprawdzić, czy sesja Ollama działa poprawnie:

```
import psutil

def check_if_running(process_name):
    running = False
    for proc in psutil.process_iter(["name"]):
        if process_name in proc.info["name"]:
            running = True
            break
    return running

ollama_running = check_if_running("ollama")

if not ollama_running:
    raise RuntimeError(
        "Ollama nie jest uruchomiona. Aby uruchamiać przykłady, zadbaj o jej uruchomienie."
)
print("Ollama uruchomiona:", check_if_running("ollama"))
```

Jeśli Ollama działa, powyższy kod powinien wyświetlić komunikat Ollama uruchomiona: True. Jeśli wyświetliły się inne wyniki, sprawdź, czy polecenie ollama serve lub aplikacja Ollama zostały uruchomione.

Uruchamianie kodu w nowej sesji Pythona

Jeśli zamknąłeś sesję Pythona lub jeśli wolisz uruchamiać pozostały kod z tego rozdziału w innej sesji Pythona, możesz skorzystać z poniższego kodu. Jego działanie polega na załadowaniu utworzonego wcześniej pliku danych instrukcji i odpowiedzi oraz zmianie wykorzystywanej wcześniej definicji funkcji format_input (biblioteka tqdm została zaimportowana po to, aby w dalszej części kodu można było skorzystać z paska postępu):

```
import json
from tqdm import tqdm

file_path = "instruction-data-with-response.json"
with open(file_path, "r") as file:
    test_data = json.load(file)

def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n## Instrukcja:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n## Wejście:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

Alternatywą dla polecenia `o1lama run` jest interakcja z modelem za pośrednictwem interfejsu API REST z użyciem skryptu w Pythonie. Sposób korzystania z API zadeemonstrowałem na listingu 7.10 w funkcji `query_model`.

Listing 7.10. Odpytywanie lokalnego modelu Ollama

```
import urllib.request

def query_model(
    prompt,
    model="llama3",
    url="http://localhost:11434/api/chat"
):
    data = {                                     ← Utworzenie ładunku danych w formacie słownika
        "model": model,
        "messages": [
            {"role": "user", "content": prompt}
        ],
        "options": {                                ← Ustawienia w celu uzyskiwania deterministycznych odpowiedzi
            "seed": 123,
            "temperature": 0,
            "num_ctx": 2048
        }
    }

    payload = json.dumps(data).encode("utf-8")      ← Konwersja słownika na ciąg znaków w formacie JSON i zakodowanie go do postaci bajtów
    request = urllib.request.Request(
        url,
        data=payload,
        method="POST"
    )

    request.add_header("Content-Type", "application/json")      ← Utworzenie obiektu żądania, ustawienie metody na POST i dodanie niezbędnych nagłówków

    response_data = ""
    with urllib.request.urlopen(request) as response:          ← Wysłanie żądania i odczytanie odpowiedzi
        while True:
            line = response.readline().decode("utf-8")
            if not line:
                break
            response_json = json.loads(line)
            response_data += response_json["message"]["content"]

    return response_data
```

Przed uruchomieniem kolejnych komórek kodu w tym notatniku upewnij się, że Ollama nadal jest uruchomiona. Poprzednie komórki kodu powinny wypisać komunikat "`O1lama uruchomiona": True`", co potwierdza, że model jest aktywny i gotowy do odbierania żądań.

Oto przykład użycia zaimplementowanej przed chwilą funkcji `query_model`:

```
model = "llama3"
result = query_model("What do Llamas eat?", model)
print(result)
```

A oto uzyskana odpowiedź:

Llamas are ruminant animals, which means they have a four-chambered stomach that
 ↪allows them to digest plant-based foods. Their diet typically consists of:
 1. Grasses: Llamas love to graze on grasses, including tall grasses, short grasses,
 ↪and even weeds.

...

W tłumaczeniu na język polski:

Lamy są przeżuwaczami, co oznacza, że mają czterokomorowy żołądek i jedzą rośliny
 ↪bogate w błonnik. Na wolności lamy zazwyczaj żywią się:

1. Trawy: Uwielbiają wypasać się na różnego rodzaju trawach, w tym na wysokich
 ↪trawach, pszenicy, owsie i jęczmieniu.

Za pomocą wykorzystanej wcześniej funkcji query_model można ocenić odpowiedzi wygenerowane przez dostrojony model. Przekazujemy do modelu Llama 3 prompt, w którym prosimy o ocenę odpowiedzi dostrojonego modelu w skali od 0 do 100 na podstawie przekazanej dla porównania odpowiedzi ze zbioru testowego.

Najpierw zastosujemy to podejście do analizowanych wcześniej pierwszych trzech przykładów ze zbioru testowego:

```
for entry in test_data[:3]:
    prompt = (
        f"Given the input `{format_input(entry)}` "
        f"and correct output `{entry['output']}`, "
        f"score the model response `{entry['model_response']}` "
        f"on a scale from 0 to 100, where 100 is the best score. "
    )
    print("\nOdpowiedź ze zbioru testowego:")
    print(">>", entry['output'])
    print("\nOdpowiedź modelu:")
    print(">>", entry["model_response"])
    print("\nOcena:")
    print(">>", query_model(prompt))
    print("\n-----")
```

Ten kod wyświetla wyniki podobne do poniższych (w chwili gdy pisalem te słowa, aplikacja Ollama nie była w pełni deterministyczna, więc odpowiedzi, które uzyskasz, mogą się od nich różnić)⁷:

Odpowiedź ze zbioru testowego:

>> The car is as fast as lightning. (Samochód jest szybki jak błyskawica).

Odpowiedź modelu:

>> The car is as fast as a bullet. (Samochód jest szybki jak pocisk).

⁷ Tak jak wcześniej, w nawiasach podano tłumaczenie na język polski instrukcji przekazanych modelowi oraz udzielonych przez niego odpowiedzi – *przyp. tłum.*

Ocena:

>> Odpowiedź modelu „The car is as fast as a bullet.” oceniłbym na 85 punktów na 100.

Oto dlaczego:

W odpowiedzi model poprawnie użył porównania: porównał prędkość samochodu do czegoś innego (w tym przypadku do pocisku).

Porównanie jest trafne i ma sens, ponieważ jak wiadomo, pociski są wystrzeliwane z dużą szybkością.

Wyrażenie „as fast as” („tak szybko jak”) poprawnie wprowadza porównanie.

Jedynym powodem, dla którego nie przyznaję modelowi idealnej oceny, jest to, że niektórzy ludzie mogą uznać to porównanie za mało sugestywne. Porównanie czegoś do błyskawicy (jak w oryginalnej odpowiedzi) może być odebrane jako bardziej dramatyczne i bardziej przyciągające uwagę. Jednak porównanie „as fast as a bullet” to nadal mocne i skuteczne porównanie, które dobrze oddaje ideę szybkości samochodu.

Ogólnie uważam, że model wykonał świetną robotę!

Odpowiedź ze zbioru testowego:

>> The type of cloud typically associated with thunderstorms is cumulonimbus. (Rodzajem chmury typowo związanym z burzami jest cumulonimbus).

Odpowiedź modelu:

>> The type of cloud associated with thunderstorms is a cumulus cloud. (Rodzajem chmury związanej z burzami jest chmura cumulus).

Ocena:

>> Oceniłbym tę odpowiedź modelu na 40 punktów na 100.

Oto dlaczego:

Model poprawnie identyfikuje, że burze są związane z chmurami (poprawnie identyfikuje rodzaj zjawiska).

Jednak rodzaj chmury związanej z burzami został wskazany niepoprawnie. Chmury cumulus zwykle nie są związane z burzami. Z burzami są związane chmury cumulonimbus.

W odpowiedzi brakuje dokładności.

Ogólnie rzecz biorąc, chociaż model próbuje odpowiedzieć na instrukcję, udziela niepoprawnej odpowiedzi, co jest znaczącym błędem.

Odpowiedź ze zbioru testowego:

>> Jane Austen.

Odpowiedź modelu:

>> The author of ‘Pride and Prejudice’ is Jane Austen. (Autorką „Dumy i uprzedzenia” jest Jane Austen).

Ocena:

>> Odpowiedź modelu oceniłbym na 95 na 100. Oto dlaczego:

Odpowiedź dokładnie odpowiada na pytanie. Model wymienia w niej autorkę „Dumy i uprzedzenia” jako Jane Austen.

Odpowiedź jest zwięzła i jasna, dzięki czemu łatwo ją zrozumieć.

Nie ma błędów gramatycznych ani niejasności, które mogłyby prowadzić do nieporozumień.

Jedynym powodem, dla którego nie przyznałbym modelowi idealnej oceny, jest to, że odpowiedź jest nieco zbyt rozwlekła — powtarzanie treści pytania w odpowiedzi nie jest konieczne. Bardziej zwięzłą odpowiedzią byłoby po prostu „Jane Austen”.

Na podstawie wygenerowanych odpowiedzi można stwierdzić, że model Llama 3 zapewnia rozsądne oceny i potrafi przyznawać punkty także wtedy, gdy odpowiedź modelu nie jest całkowicie poprawna. Na przykład, jeśli weźmiemy pod uwagę ocenę odpowiedzi „cumulus cloud”, zauważymy, że model uznaje częściową poprawność odpowiedzi.

W odpowiedzi na poprzedni prompt oprócz oceny punktowej uzyskaliśmy bardzo szczegółową ocenę. Prompt można zmodyfikować w taki sposób, aby model generował wyłącznie liczby całkowite w zakresie od 0 do 100, gdzie 100 oznacza najlepszy możliwy wynik. Dzięki tej modyfikacji możemy obliczyć średni wynik dla modelu, co można wykorzystać jako bardziej zwięzłą i ilościową ocenę jego wydajności. Funkcja `generate_model_scores` pokazana na listingu 7.11 korzysta ze zmodyfikowanego prompta informującego model, aby zwracał wyłącznie ocenę w postaci liczb całkowitych.

Listing 7.11. Ocena modelu LLM dostrojonego do zadań wykonywania instrukcji

```
def generate_model_scores(json_data, json_key, model="llama3"):
    scores = []
    for entry in tqdm(json_data, desc="Scoring entries"):
        prompt = (
            f"Given the input `~{format_input(entry)}~`"
            f"and correct output `~{entry['output']}~`,"
            f"score the model response `~{entry[json_key]}~`"
            f"on a scale from 0 to 100, where 100 is the best score."
            f"Respond with the integer number only." ←
        )
        score = query_model(prompt, model)
        try:
            scores.append(int(score))
        except ValueError:
            print(f"Could not convert score: {score}")
            continue

    return scores
```

Zmodyfikowany wiersz z instrukcją dla modelu, aby zwracał wyłącznie całkowitoliczbową ocenę

Spróbujmy teraz zastosować funkcję `generate_model_scores` do całego zbioru `test_data`, co na MacBooku Air M3 zajmuje około 1 minuty:

```
scores = generate_model_scores(test_data, "model_response")
print(f"Number of scores: {len(scores)} of {len(test_data)}")
print(f"Average score: {sum(scores)/len(scores):.2f}\n")
```

Wyniki są następujące:

```
Scoring entries: 100% [██████████] | 110/110
[01:10<00:00, 1.56it/s]
Number of scores: 110 of 110
Average score: 50.32
```

Na podstawie wyników oceny możemy się dowiedzieć, że dostrojony model osiąga średnią ocenę przekraczającą 50, co jest sensownym punktem odniesienia w celu porównania z innymi modelami lub eksperymentowania z różnymi konfiguracjami szkoleniowymi w celu poprawy wydajności modelu.

Warto zwrócić uwagę na to, że w chwili, gdy pisałem te słowa, aplikacja Ollama nie była w pełni deterministyczna we wszystkich systemach operacyjnych, co oznacza, że wyniki, które uzyskasz, mogą się nieznacznie różnić od moich. Aby uzyskać bardziej wiarygodne wyniki, możesz powtórzyć ocenę wiele razy, a rezultaty uśrednić.

Aby jeszcze bardziej poprawić wydajność modelu, można zbadać różne strategie, na przykład:

- dostosowanie podczas dostrajania hiperparametrów, takich jak szybkość uczenia, rozmiar partii lub liczba epok,
- zwiększenie rozmiaru szkoleniowego zbioru danych lub zróżnicowanie przykładów w celu uwzględnienia szerszego zakresu tematów i stylów,
- eksperymentowanie z różnymi promptami lub formatami instrukcji w celu skuteczniejszego zarządzania odpowiedziami modelu,
- korzystanie z większego wstępnie przeszkolonego modelu, potencjalnie o większych możliwościach rozpoznawania złożonych wzorców i generowania dokładniejszych odpowiedzi.

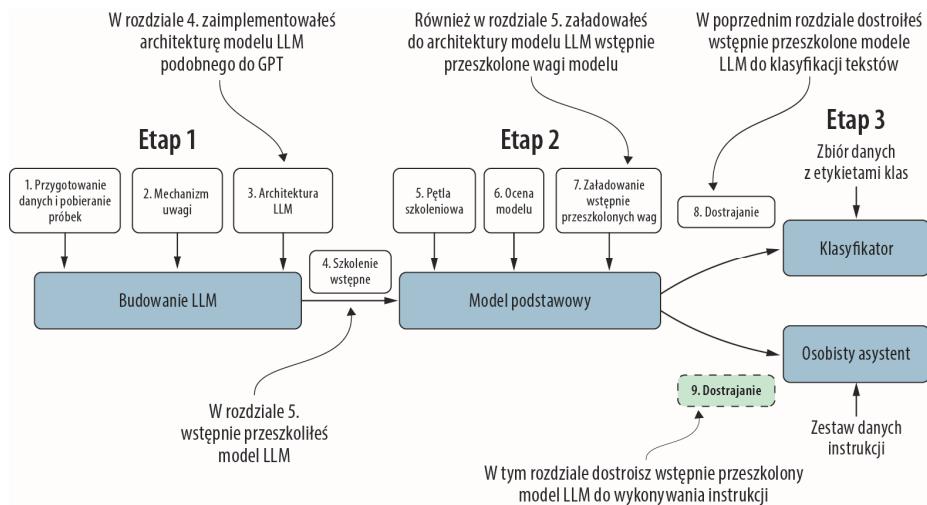
UWAGA Dla porównania, po zastosowaniu opisanej tutaj metodyki model podstawowy Llama 3 8B bez żadnego dostrajania osiągał dla wykorzystawanego zbioru testowego średni wynik 58,51. Model wyspecjalizowany do zadań wykonywania instrukcji Llama 3 8B, który dostrojono na ogólnym zbiorze danych do wykonywania instrukcji, osiąga imponujący średni wynik 82,6.

Ćwiczenie 7.4. Dostrajanie parametrów z użyciem metody LoRA

Aby skuteczniej dostroić model LLM, zmodyfikuj kod zaprezentowany w tym rozdziale w taki sposób, by korzystał z metody adaptacji LoRA (ang. *low-rank adaptation method*), opisanej w „Dodatku E”. Porównaj czas szkolenia i wydajność modelu przed wprowadzeniem modyfikacji i po jej wprowadzeniu.

7.9. Wnioski

Ten rozdział kończy Twoją podróż przez cykl tworzenia modelu LLM. Omówiłem wszystkie najważniejsze kroki związane z opracowywaniem modeli LLM, w tym implementację architektury LLM, wstępne szkolenie modelu i dostrojenie go do określonych zadań, tak jak pokazałem na rysunku 7.21. Przedyskutujmy kilka pomysłów, na które warto zwrócić uwagę w następnej kolejności.



Rysunek 7.21. Trzy główne etapy kodowania modelu LLM

Główna treść tej książki uzupełnia dostępny w repozytorium GitHub duży wybór dodatkowych materiałów, które mogą okazać się przydatne. Aby dowiedzieć się więcej o tych zasobach, zajrzyj do sekcji *Bonus Material* na stronie *README* repozytorium pod adresem <https://mng.bz/r12g>.

7.9.1. Co dalej?

Chociaż w tej książce omówiłem najważniejsze kroki tworzenia modeli LLM, jest dodatkowy krok, który można wykonać po dostrojeniu modelu do zadań wykonywania instrukcji: dostrojenie preferencji. Dostrajanie preferencji jest szczególnie przydatne z perspektywy dostosowywania modelu do konkretnych preferencji użytkownika. Jeśli chcesz dowiedzieć się więcej na ten temat, zajrzyj do folderu *o4_preference-tuningwith-dpo* w dodatkowym repozytorium GitHub przygotowanym dla tej książki, dostępnych pod adresem <https://mng.bz/dZwD>.

7.9.2. *Bądź na bieżąco w szybko zmieniającej się dziedzinie*

Dziedziny sztucznej inteligencji i modeli LLM rozwijają się w imponującym tempie. Jednym ze sposobów, aby być na bieżąco z najnowszymi osiągnięciami, jest zapoznawanie się z najnowszymi artykułami badawczymi w serwisie arXiv, pod adresem <https://arxiv.org/list/cs.LG/recent>. Ponadto wielu badaczy i ekspertów bardzo aktywnie dzieli się najnowszymi osiągnięciami – oraz je omawia – na platformach mediów społecznościowych, takich jak X (dawniej Twitter) i Reddit. W szczególności dobrym zasobem pozwalającym na kontakt ze społeczeństwem oraz wymianę informacji na temat najnowszych narzędzi i trendów jest podkanał serwisu Reddit *r/LocalLLaMA*. Ja własnymi spostrzeżeniami oraz informacjami dotyczącymi najnowszych badań związanych z modelami LLM dzielę się również na swoim blogu, dostępnym pod adresami <https://magazine.sebastianraschka.com> i <https://sebastianraschka.com/blog/>.

7.9.3. *Na koniec*

Mam nadzieję, że spodobała Ci się podróż, podczas której pokazałem Ci tajniki implementowania modeli LLM od podstaw oraz kodowania funkcji związanych z ich wstępny szkoleniem i dostrajaniem. Moim zdaniem zbudowanie LLM od podstaw jest najskuteczniejszym sposobem na dogłębne zrozumienie sposobu działania tych modeli. Mam nadzieję, że zaprezentowane praktyczne podejście dostarczyło Ci cennych informacji i zapewniło solidne podstawy do rozwijania modeli LLM.

Chociaż głównym celem tej książki jest edukacja, być może jesteś zainteresowany wykorzystaniem innych, bardziej wydajnych modeli LLM w praktycznych zastosowaniach. Jeśli tak jest, to zachęcam Cię do zapoznania się z popularnymi narzędziami takimi jak *Axolotl* (<https://github.com/OpenAccess-AI-Collective/axolotl>) i *LitGPT* (<https://github.com/Lightning-AI/litgpt>), w których rozwój jestem aktywnie zaangażowany.

Dziękuję Ci za to, że dołączyłeś do mnie w tej podróży edukacyjnej, i życzę wszystkiego najlepszego w przyszłych przedsięwzięciach w ekscytującej dziedzinie modeli LLM i AI!

Podsumowanie

- Proces dostrajania modelu do wykonywania instrukcji pozwala dostosować wstępnie przeszkolony model LLM do wykonywania instrukcji człowieka i generowania pożądanych odpowiedzi.
- Przygotowanie zbioru danych obejmuje pobranie zbioru danych zawierających pary instrukcja-odpowiedź, sformatowanie zapisanych w nim pozycji oraz podzielenie na zbiory szkoleniowy, walidacyjny i testowy.

- Partie szkoleniowe konstruuje się za pomocą niestandardowej funkcji `collate`, która uzupełnia sekwencje, tworzy docelowe identyfikatory tokenów i maskuje tokeny wypełniające.
- W rozdziale załadowaliśmy wstępnie przeszkolony model GPT-2 średniej wielkości, z 355 milionami parametrów, który służył jako punkt wyjścia do dostrajania modelu pod kątem wykonywania instrukcji.
- Następnie dostroiliśmy wstępnie przeszkolony model na zbiorze danych instrukcji z wykorzystaniem pętli szkoleniowej podobnej do tej, jaką zastosowaliśmy podczas wstępnego szkolenia.
- Ocena modelu polega na wyodrębnieniu jego odpowiedzi dla instrukcji ze zbioru testowego i ich ocenie (np. z użyciem innego modelu LLM).
- Do automatycznego oceniania odpowiedzi dostrojonego modelu na zbiorze testowym można wykorzystać aplikację Ollama z 8-miliardowym modelem Llama. W ten sposób można obliczyć średnią ocenę umożliwiającą ilościowe określenie wydajności modelu.

Dodatek A

Wprowadzenie w tematykę frameworka PyTorch

Celem tego dodatku jest dostarczenie Czytelnikom niezbędnych informacji umożliwiających zastosowanie uczenia głębokiego w praktyce i implementację dużych modeli językowych (LLM) od podstaw. Głównym narzędziem wykorzystywanym w tej książce jest PyTorch — popularna biblioteka uczenia głębokiego oparta na Pythonie. Przeprowadzę Cię przez proces konfiguracji środowiska uczenia głębokiego z obsługą biblioteki PyTorch i urządzenia GPU.

Następnie zapoznasz się z pojęciem tensorów i ich zastosowaniem we frameworku PyTorch. Zajmiemy się także automatycznym mechanizmem różniczkowania PyTorch, umożliwiającym wygodne i efektywne wykorzystanie propagacji wstecznej — kluczowego elementu szkolenia sieci neuronowych.

Ten dodatek pełni funkcję elementarza dla tych, którzy dopiero zaczynają zajmować się uczeniem głębokim i frameworkiem PyTorch. Chociaż staram się objaśnić framework PyTorch od podstaw, moim celem nie jest jego wyczerpujące omówienie. Zamiast tego skupię się na podstawach tego frameworka i komponentach, które będą niezbędne do implementacji modelu LLM. Jeśli znasz podstawy uczenia głębokiego, możesz pominąć ten dodatek i przejść bezpośrednio do rozdziału 2.

A.1. Czym jest PyTorch?

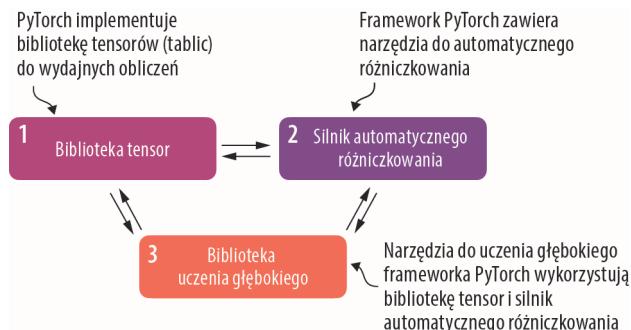
PyTorch (<https://pytorch.org/>) to oparta na języku Python biblioteka open source do obsługi uczenia głębokiego. Według *Papers With Code* (<https://paperswithcode.com/trends>), platformy, która śledzi i analizuje artykuły naukowe, od 2019 roku PyTorch jest zdecydowanie najczęściej używaną biblioteką w badaniach nad uczeniem głębokim. Z kolei według ankiety Kaggle Data Science and Machine Learning Survey 2022

(<https://www.kaggle.com/c/kaggle-survey-2022>) około 40% respondentów korzysta z PyTorch, a liczba ta rośnie z każdym rokiem.

Jednym z powodów, dla których PyTorch jest tak popularny, jest jego wydajność oraz wygodny interfejs. Jest nie tylko łatwy w obsłudze, ale także niezwykle elastyczny, dzięki czemu zaawansowani użytkownicy mogą modyfikować niskopoziomowe aspekty modeli w celu ich dostrojenia i optymalizacji. Krótko mówiąc, dla wielu użytkowników i ekspertów w dziedzinie AI framework PyTorch oferuje właściwą równowagę między użytecznością a zakresem udostępnianych funkcji.

A.1.1. Trzy podstawowe komponenty frameworka PyTorch

PyTorch jest dość rozbudowaną biblioteką. Jednym ze sposobów poznawania jej możliwości jest skupienie się na trzech głównych komponentach, które podsumowałem na rysunku A.1.



Rysunek A.1. Trzy główne komponenty frameworka PyTorch obejmują bibliotekę tensor, będącą podstawowym blokiem budulcowym do obliczeń, silnik automatycznego różniczkowania, ułatwiający optymalizację modeli, oraz funkcje narzędziowe uczenia głębokiego, ułatwiające implementację i szkolenie modeli opartych na głębokich sieciach neuronowych

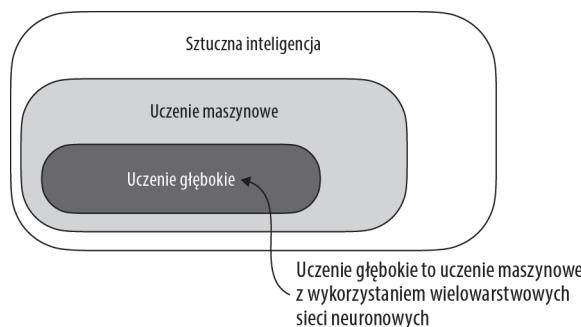
Po pierwsze, PyTorch to *biblioteka oparta na tensorach*, rozszerzająca funkcjonalność zorientowanej na tablice biblioteki programistycznej NumPy o dodatkową funkcję przyspieszającą obliczenia na układach GPU. Dzięki niej możliwe jest płynne przełączanie się między układami CPU i GPU. Po drugie, PyTorch to *silnik automatycznego różniczkowania*, znany również jako autograd, umożliwiający automatyczne obliczanie gradientów w działaniach na tensorach. Komponent ten upraszcza wsteczną propagację i optymalizację modelu. Wreszcie — PyTorch jest *biblioteką uczenia głębokiego*. Dostarcza modułowych, elastycznych i wydajnych elementów, takich jak wstępnie przeszkozone modele, funkcje strat i optymalizatory niezbędne do projektowania i szkolenia szerokiej gamy modeli uczenia głębokiego. Dzięki temu zaspokaja potrzeby zarówno badaczy, jak i programistów.

A.1.2. Definicja uczenia głębokiego

Modele LLM często określa się w publikacjach jako modele AI. Modele LLM są jednak również rodzajem głębokich sieci neuronowych, a PyTorch jest biblioteką uczenia głębokiego. Brzmi skomplikowanie? Zanim przejdziemy dalej, warto poświęcić krótką chwilę, by podsumować związek między tymi terminami.

Sztuczna inteligencja (ang. *artificial intelligence* – AI), ogólnie rzecz biorąc, polega na tworzeniu systemów komputerowych zdolnych do wykonywania zadań, które zwykle wymagają ludzkiej inteligencji. Zadania te obejmują rozumienie języka naturalnego, rozpoznawanie wzorców i podejmowanie decyzji (pomimo znaczących postępów sztuczna inteligencja wciąż jest daleka od osiągnięcia tego poziomu inteligencji ogólnej).

Uczenie maszynowe (ang. *machine learning*) – jak pokazałem na rysunku A.2, stanowi poddziedzinę sztucznej inteligencji, która koncentruje się na opracowywaniu i ulepszaniu algorytmów uczenia. Kluczowa idea uczenia maszynowego to umożliwienie komputerom uczenia się na podstawie danych i prognozowania lub podejmowania decyzji bez konieczności wyraźnego zaprogramowania do wykonania zadania. Wiąże się to z opracowywaniem algorytmów, które pozwalają identyfikować wzorce, uczyć się na podstawie danych historycznych oraz – dzięki coraz większej ilości danych i informacji zwrotnych – poprawiać z biegiem czasu swoją wydajność.



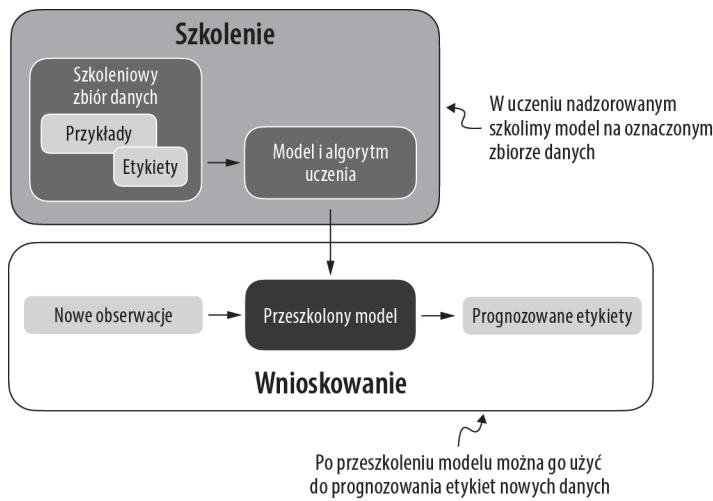
Rysunek A.2. Uczenie głębokie to podkategoria uczenia maszynowego, która skupia się na implementacji głębokich sieci neuronowych. Uczenie maszynowe to podkategoria sztucznej inteligencji, która dotyczy algorytmów uczących się na podstawie danych. Sztuczna inteligencja to szersze pojęcie, opisujące zdolność komputerów do wykonywania zadań, które zazwyczaj wymagają ludzkiej inteligencji

Uczenie maszynowe odegrało kluczową rolę w ewolucji sztucznej inteligencji; było podstawą rozwoju wielu modeli, które są dziś dostępne, włącznie z modelami LLM. Uczenie maszynowe jest również podstawą takich technologii jak systemy rekommendacji używane przez sprzedawców internetowych, usługi strumieniowego przesyłania treści, filtrowanie spamu w wiadomościach e-mail, rozpoznawanie głosu w wirtualnych asystentach, a nawet autonomiczne samochody. Wprowadzenie uczenia maszynowego i jego rozwój znacznie zwiększyły możliwości sztucznej inteligencji. Pozwoliły

wyjść poza ramy ścisłych systemów opartych na regułach i umożliwiły dostosowanie się modeli do nowych danych wejściowych lub zmieniających się środowisk.

Uczenie głębokie to podkategoria uczenia maszynowego, która skupia się na implementacji głębokich sieci neuronowych. Pierwotną inspiracją do stworzenia głębokich sieci neuronowych był sposób, w jaki działa ludzki mózg, a zwłaszcza wzajemne połączenia między wieloma neuronami. Słowo „głębokie” w nazwie uczenie głębokie odnosi się do wielu ukrytych warstw sztucznych neuronów lub węzłów, które pozwalają im modelować złożone, nieliniowe związki w danych. W przeciwieństwie do tradycyjnych technik uczenia maszynowego, które wyróżniają się prostym rozpoznawaniem wzorców, uczenie głębokie szczególnie dobrze sprawdza się w zadaniach obsługi niestrukturyzowanych danych, takich jak obrazy, dźwięk lub tekst, więc nadaje się zwłaszcza do zastosowań dotyczących modeli LLM.

Typowy przepływ pracy modelowania predykcyjnego (zwanego również *uczeniem nadzorowanym*) w uczeniu maszynowym i uczeniu głębokim podsumowałem na rysunku A.3.



Rysunek A.3. Proces uczenia nadzorowanego w modelowaniu predykcyjnym składa się z etapu szkolenia, w którym model jest szkolony na oznaczonych przykładach w zestawie danych szkoleniowych. Przeszkolony model można następnie wykorzystać do przewidywania etykiet dla nowych obserwacji

Model korzysta z algorytmu uczenia się i szkoli się na zbiorze danych złożonym z przykładów i odpowiadających im etykiet. Na przykład w przypadku klasyfikatora spamu w wiadomościach e-mail zbiór danych szkoleniowych składa się z wiadomości e-mail oraz etykiet „spam” i „nie spam”, przypisanych przez człowieka. Następnie przeszkolony model można wykorzystać w odniesieniu do nowych obserwacji (tzn. nowych wiadomości e-mail), tak aby model przewidział ich nieznaną etykietę („spam” lub „nie spam”). Oczywiście pomiędzy etapami uczenia i wnioskowania trzeba poddać model

ocenie, aby przed użyciem go w rzeczywistym zastosowaniu zyskać pewność, że spełnia kryteria wydajności.

Jeśli szkolisz model LLM pod kątem klasyfikowania tekstów, przepływ pracy związany ze szkoleniem i używaniem modelu LLM jest podobny do przedstawionego na rysunku A.3. Jeśli jesteś zainteresowany szkoleniem modelu LLM w celu generowania tekstów, co jest głównym celem w tej książce, to na rysunku A.3 również znajdziesz cenne informacje. W tym przypadku etykiety podczas wstępnego szkolenia można uzyskać na podstawie samego tekstu (zadanie przewidywania następnego słowa wprowadzone w rozdziale 1.). Podczas wnioskowania model LLM, na podstawie wprowadzonego prompta, wygeneruje całkowicie nowy tekst (zamiast przewidywać etykiety).

A.1.3. Instalacja frameworka PyTorch

PyTorch można zainstalować tak jak każdą inną bibliotekę lub pakiet Pythona. Ponieważ jednak PyTorch jest kompleksową biblioteką zawierającą kody kompatybilne z układami CPU i GPU, instalacja może wymagać dodatkowych wyjaśnień.

Wersja Pythona

Wiele bibliotek do obliczeń naukowych nie obsługuje najnowszej wersji Pythona. Z tego względu w przypadku instalacji frameworka PyTorch zalecam korzystanie z wersji Pythona starszej o jedno lub dwa wydania. Na przykład, jeśli najnowsza wersja Pythona to 3.13, lepiej użyć Pythona 3.11 lub 3.12.

Na przykład istnieją dwie wersje PyTorch: wersja zubożona, obsługująca wyłącznie obliczenia na układach CPU, i wersja pełna, która obsługuje zarówno obliczenia na układach CPU, jak i GPU. Jeśli Twój komputer posiada w układzie GPU zgodny z CUDA, który można wykorzystać do uczenia głębokiego (najlepiej Nvidia T4, RTX 2080 Ti lub nowszy), zachęcam do zainstalowania wersji GPU. Niezależnie od tego domyślnym poleciением instalacji PyTorch w terminalu jest:

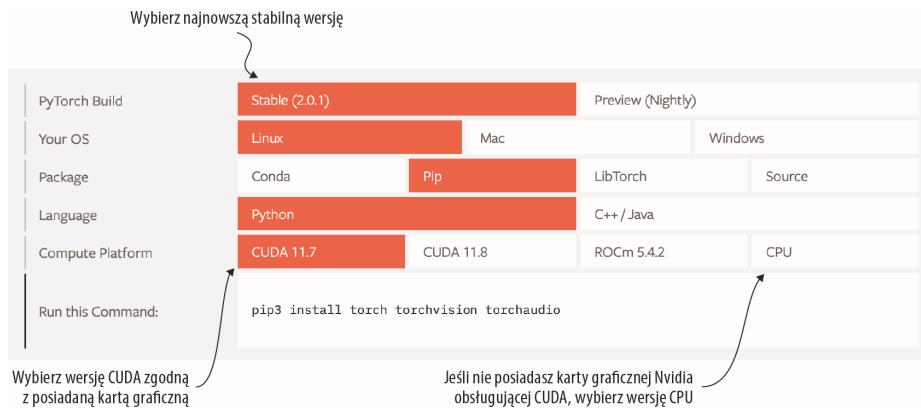
```
pip install torch
```

Załóżmy, że Twój komputer obsługuje układ graficzny zgodny z CUDA. W takim przypadku (przy założeniu, że środowisko Pythona, z którym pracujesz, ma zainstalowane niezbędne zależności — np. pip) zostanie automatycznie zainstalowana wersja PyTorch obsługująca akcelerację GPU za pośrednictwem CUDA.

UWAGA W czasie gdy pisałem tę książkę, do frameworka PyTorch dodano również eksperymentalną obsługę układów GPU AMD za pośrednictwem ROCm. Dodatkowe instrukcje można znaleźć na stronie <https://pytorch.org>.

Aby jawnie wskazać wersję PyTorch zgodną z CUDA, często lepiej jest określić tę wersję CUDA, z którą framework PyTorch ma być zgodny. Polecenia instalacji PyTorch z obsługą CUDA dla różnych systemów operacyjnych można znaleźć na oficjalnej

stronie PyTorch (<https://pytorch.org>). Polecenie, które zainstaluje framework PyTorch wraz z opcjonalnymi dla tej książki bibliotekami torchvision i torchaudio, pokazalem na rysunku A.4.



Rysunek A.4. Informacje o poleceniu instalacji dla Twojego systemu i zalecenia dotyczące instalacji framework PyTorch znajdziesz na stronie <https://pytorch.org>

W przykładach zaprezentowanych w książce używałem PyTorch w wersji 2.4.0, więc w celu zainstalowania dokładnie tej wersji, tak aby zagwarantować kompatybilność z przykładami w tej książce, zalecam użycie poniższego polecenia:

```
pip install torch==2.4.0
```

Jednak jak wspomniałem wcześniej, w zależności od systemu operacyjnego polecenie instalacji może się nieznacznie różnić od pokazanego powyżej. Dlatego aby wybrać polecenie instalacji dla swojego systemu operacyjnego, warto odwiedzić stronę <https://pytorch.org> i skorzystać z menu instalacji (patrz rysunek A.4). Pamiętaj, aby w poleceniu zastąpić nazwę torch nazwą ze wskazaniem wersji — torch==2.4.0.

Aby sprawdzić wersję PyTorch, możesz uruchomić następujący kod:

```
import torch
torch.__version__
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
'2.4.0'
```

Jeśli szukasz dodatkowych zaleceń i instrukcji dotyczących konfigurowania środowiska Python lub instalowania innych bibliotek używanych w tej książce, odwiedź repozytorium GitHub tej książki, dostępne pod adresem <https://github.com/rasbt/LLMs-from-scratch>.

Po zainstalowaniu frameworka PyTorch możesz sprawdzić, czy instalacja rozpoznałej wbudowany układ GPU Nvidia, uruchomisz poniższy kod w Pythonie:

```
import torch
torch.cuda.is_available()
```

PyTorch i Torch

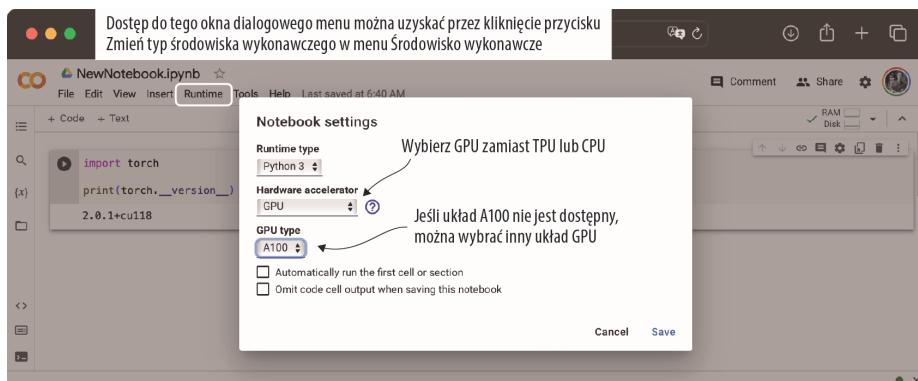
Biblioteka Pythona nosi nazwę PyTorch przede wszystkim dlatego, że jest kontynuacją biblioteki Torch, ale dostosowaną do Pythona (stąd „PyTorch”). „Torch” nawiązuje do korzeni biblioteki Torch — oferującego zaawansowaną obsługę algorytmów uczenia maszynowego framework obliczeń naukowych, który pierwotnie stworzono z wykorzystaniem języka programowania Lua.

Uruchomienie tego kodu zwraca następujący wynik:

True

Jeżeli polecenie zwróci True, będzie to oznaczać, że instalacja PyTorch przebiegła poprawnie. Jeśli zwróci False, komputer być może nie ma kompatybilnego układu GPU lub framework PyTorch go nie rozpoznaje. Chociaż układy GPU nie są konieczne do uruchamiania przykładów z początkowych rozdziałów tej książki, które koncentrują się na implementacji modeli LLM do celów edukacyjnych, można dzięki nim znacznie przyspieszyć obliczenia związane z uczeniem głębokim.

Jeśli nie masz dostępu do układów GPU, możesz skorzystać z usług kilku dostawców w chmurze, co pozwoli Ci na uruchamianie obliczeń na układach GPU za opłatą zgodną z faktycznym wykorzystaniem. Popularnym środowiskiem opartym na notatnikach Jupyter jest Google Colab (<https://colab.research.google.com>). W chwili gdy pisałem te słowa, ta platforma zapewniała ograniczony czasowo dostęp do układów GPU. Układ GPU możesz wybrać za pomocą menu **Środowisko wykonawcze/Zmień typ środowiska wykonawczego** (rysunek A.5).



Rysunek A.5. Wybierz urządzenie GPU w środowisku Google Colab, w menu Środowisko wykonawcze/Zmień typ środowiska wykonawczego

PyTorch na komputerach z układem Apple Silicon

Jeśli posiadasz komputer Apple Mac z chipem Apple Silicon (np. M1, M2, M3 lub nowszym), możesz wykorzystać jego możliwości do przyspieszenia uruchamiania kodu PyTorch. Aby użyć układu Apple Silicon z PyTorch, trzeba najpierw zainstalować framework PyTorch w zwykły sposób. Następnie, aby sprawdzić, czy komputer Mac obsługuje akcelerację PyTorch za pomocą układu Apple Silicon, możesz uruchomić prosty fragment kodu w Pythonie:

```
print(torch.backends.mps.is_available())
```

Jeśli powyższe polecenie zwróci True, to znaczy, że komputer Mac ma układ Apple Silicon, który można wykorzystać do akceleracji kodu PyTorch.

Ćwiczenie A.1

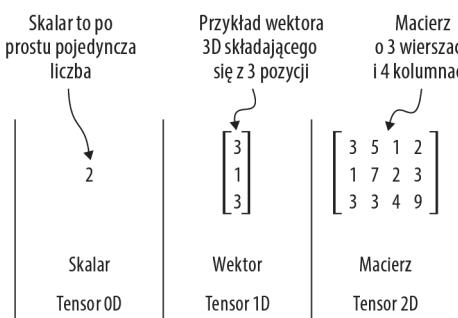
Zainstaluj framework PyTorch i skonfiguruj go na swoim komputerze.

Ćwiczenie A.2

Uruchom dodatkowy kod dostępny na stronie <https://mng.bz/o05v>, który sprawdza, czy środowisko jest poprawnie skonfigurowane.

A.2. *Tensory*

Tensory reprezentują pojęcie matematyczne, które uogólnia wektory i macierze do potencjalnie wyższych wymiarów. Mówiąc inaczej, tensory są obiektami matematycznymi, które można scharakteryzować za pomocą rzędu (lub rangi), określającego liczbę wymiarów. Na przykład, jak pokazałem na rysunku A.6, skalar (zwykła liczba) jest tensorem rangi 0, wektor jest tensorem rangi 1, a macierz jest tensorem rangi 2.



Rysunek A.6. Tensory o różnych rangach. Na tym rysunku 0D odpowiada randze 0, 1D — randze 1, a 2D — randze 2. Trójwymiarowy wektor, który składa się z trzech elementów, nadal jest tensorem rangi 1

Z perspektywy obliczeniowej tensory odgrywają rolę kontenerów danych. Na przykład przechowują wielowymiarowe dane, przy czym każdy z wymiarów reprezentuje inną cechę. Biblioteki obsługi tensorów, takie jak PyTorch, umożliwiają skuteczne tworzenie struktur tensorów, wykonywanie na nich przekształceń i oparte na nich obliczenia. W tym kontekście biblioteka obsługi tensorów pełni funkcję biblioteki obsługi tablic.

Tensory PyTorch są podobne do tablic biblioteki NumPy, ale mają kilka dodatkowych cech, które są ważne z perspektywy uczenia głębokiego. Na przykład w PyTorch wprowadzono silnik automatycznego różniczkowania, co upraszcza *obliczanie gradientów* (patrz podrozdział A.4). Tensory PyTorch obsługują również obliczenia na układach GPU, co pozwala przyspieszyć szkolenie głębokich sieci neuronowych (patrz podrozdział A.8).

PyTorch z interfejsem API podobnym do NumPy

We frameworku PyTorch na potrzeby działań na tensorach zaadaptowano większość wywołań API i składni biblioteki NumPy. Jeśli jesteś nowym użytkownikiem biblioteki NumPy, możesz zapoznać się z najważniejszymi pojęciami w moim artykule *Scientific Computing in Python: Introduction to NumPy and Matplotlib*, dostępnym na stronie <https://sebastianraschka.com/blog/2020/numpy-intro.html>.

A.2.1. Skalary, wektory, macierze i tensory

Jak wspomniałem wcześniej, tensory PyTorch są kontenerami danych dla struktur podobnych do tablic. Skalar to tensor o wymiarze zerowym (zwykła liczba), wektor to tensor jednowymiarowy, a macierz jest tensorem dwuwymiarowym. Nie istnieje żadna konkretna nazwa dla tensorów o wyższych wymiarach, więc zazwyczaj odnosimy się do tensora trójwymiarowego jako 3D, tensora czterowymiarowego – jako 4D i tak dalej. Obiekty klasy Tensor frameworka PyTorch można tworzyć za pomocą funkcji `torch.tensor`, tak jak pokazano na listingu A.1.

Listing A.1. Tworzenie tensorów frameworka PyTorch

```
import torch

tensor0d = torch.tensor(1)           ← Utworzenie zerowymiarowego tensora (skalara)
                                    na podstawie liczby całkowitej Pythona

tensor1d = torch.tensor([1, 2, 3])   ← Utworzenie jednowymiarowego tensora (wektora)
                                    na podstawie listy Pythona

tensor2d = torch.tensor([[1, 2],
                       [3, 4]])     ← Utworzenie dwuwymiarowego tensora
                                    na podstawie zagnieżdzonej listy Pythona

tensor3d = torch.tensor([[[1, 2], [3, 4]],
                       [[5, 6], [7, 8]]]) ← Utworzenie trójwymiarowego tensora
                                    na podstawie zagnieżdzonej listy Pythona
```

A.2.2. Typy danych tensorów

We frameworku PyTorch zaadaptowano z Pythona domyślny, 64-bitowy typ danych całkowitych. Dostęp do typu danych tensora można uzyskać za pośrednictwem atrybutu `.dtype` tensora:

```
tensorId = torch.tensor([1, 2, 3])
print(tensorId.dtype)
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
torch.int64
```

Jeśli utworzysz tensora na podstawie typu `float` w Pythonie, PyTorch domyślnie utworzy tensora z 32-bitową dokładnością:

```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

Oto uzyskany wynik:

```
torch.float32
```

Zastosowany wybór wynika przede wszystkim z dążenia do zachowania równowagi między dokładnością a wydajnością obliczeniową. Trzydziestodwubitowa liczba zmiennoprzecinkowa zapewnia wystarczającą dokładność dla większości zadań uczenia głębokiego, a zarazem wymaga mniej pamięci i zasobów obliczeniowych niż w przypadku zastosowania 64-bitowej liczby zmiennoprzecinkowej. Co więcej, architektury układów GPU są zoptymalizowane pod kątem obliczeń 32-bitowych, a korzystanie z tego typu danych może znacznie przyspieszyć uczenie modelu i usprawnić wnioskowanie.

Ponadto za pomocą metody `.to` tensora możliwa jest zmiana dokładności. Oto przykład kodu, który realizuje zmianę 64-bitowego tensora liczb całkowitych na 32-bitowy tensor liczb zmiennoprzecinkowych:

```
floatvec = tensorId.to(torch.float32)
print(floatvec.dtype)
```

Uruchomienie tego kodu zwraca następujący wynik:

```
torch.float32
```

Więcej informacji na temat różnych typów danych tensorowych dostępnych we frameworku PyTorch można znaleźć w oficjalnej dokumentacji frameworka, na stronie <https://pytorch.org/docs/stable/tensors.html>.

A.2.3. Typowe operacje na tensorach w PyTorch

Kompleksowe omówienie wszystkich działań na tensorach oraz związanych z nimi poleceń we frameworku PyTorch wykracza poza zakres tej książki. Będę jednak krótko opisywać potrzebne operacje, w miarę ich wprowadzania w treści książki.

Wcześniej wprowadziłem funkcję `torch.tensor()`, służącą do tworzenia nowych tensorów:

```
tensor2d = torch.tensor([[1, 2, 3],  
                      [4, 5, 6]])  
print(tensor2d)
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
tensor([[1, 2, 3],  
       [4, 5, 6]])
```

Aby odczytać kształt tensora, można skorzystać z atrybutu `.shape`:

```
print(tensor2d.shape)
```

Oto uzyskany wynik:

```
torch.Size([2, 3])
```

Jak widać, wywołanie `.shape` zwraca `[2, 3]`, co oznacza, że tensor ma dwa wiersze i trzy kolumny. Aby przekształcić tensor w tensor 3×2 , można użyć metody `.reshape`:

```
print(tensor2d.reshape(3, 2))
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
tensor([[1, 2],  
       [3, 4],  
       [5, 6]])
```

Należy jednak pamiętać, że bardziej popularnym poleceniem do przekształcania tensorów w PyTorch jest `.view()`:

```
print(tensor2d.view(3, 2))
```

Oto wynik:

```
tensor([[1, 2],  
       [3, 4],  
       [5, 6]])
```

Podobnie jak w przypadku atrybutów `.reshape` i `.view`, w kilku przypadkach framework PyTorch zapewnia wiele opcji składni do wykonywania tych samych obliczeń. PyTorch początkowo był zgodny z oryginalną konwencją składni Torch języka Lua, ale później, w odpowiedzi na powtarzające się życzenie, dodano elementy składni mające na celu zbliżenie go do konwencji biblioteki NumPy (subtelna różnica między `.view()` a `.reshape()` w PyTorch polega na obsłudze topologii pamięci: metoda `.view()` wymaga, aby oryginalne dane były ciągłe i jej wykonanie nie powiedzie się, jeśli tak nie jest, podczas gdy metoda `.reshape()` będzie działać niezależnie od topologii — jeśli to konieczne, to skopiuje dane, aby zapewnić pożądany kształt).

Można również użyć właściwości `.T`, służącej do transpozycji tensora, co oznacza odwrócenie go wzduż przekątnej. Należy zauważyć, że to działanie przypomina zmianę kształtu tensora. Można to zaobserwować na podstawie poniższego wyniku:

```
print(tensor2d.T)
```

Oto uzyskany wynik:

```
tensor([[1, 4],  
       [2, 5],  
       [3, 6]])
```

Wreszcie powszechnym sposobem mnożenia dwóch macierzy w PyTorch jest metoda `.matmul`:

```
print(tensor2d.matmul(tensor2d.T))
```

Wynik działania tego kodu to:

```
tensor([[14, 32],  
       [32, 77]])
```

Można jednak również zastosować operator `@`, który pozwala wykonać to samo działanie w bardziej zwięzły sposób:

```
print(tensor2d @ tensor2d.T)
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
tensor([[14, 32],  
       [32, 77]])
```

Jak wspomniałem wcześniej, tam, gdzie będzie taka potrzeba, będę wprowadzał dodatkowe operacje. Czytelnikom, którzy chcieliby przejrzeć wszystkie działania na tensorach dostępne w bibliotece PyTorch (większość z nich nie będzie potrzebna do uruchamiania przykładów w tej książce), zalecam zapoznanie się z oficjalną dokumentacją, dostępną na stronie <https://pytorch.org/docs/stable/tensors.html>.

A.3. *Postrzeganie modeli jako grafów obliczeniowych*

Przyjrzyjmy się teraz silnikowi automatycznego różniczkowania framework PyTorch, znanemu również jako `autograd`. System `autograd` framework PyTorch zapewnia funkcje do automatycznego obliczania gradientów w dynamicznych grafach obliczeniowych.

Graf obliczeniowy to graf skierowany, który pozwala nam reprezentować i wizualizować wyrażenia matematyczne. W kontekście uczenia głębokiego graf obliczeniowy określa sekwencję obliczeń potrzebnych do wyznaczenia wyjścia sieci neuronowej. Grafy obliczeniowe będą potrzebne do obliczenia wymaganych gradientów dla propagacji wstecznej, głównego algorytmu szkoleniowego dla sieci neuronowych.

Aby zilustrować pojęcie grafu obliczeniowego, przyjrzyjmy się konkretnemu przykładowi. Kod na poniższym listingu implementuje przejście w przód (krok prognozowania) prostego klasyfikatora regresji logistycznej, który można postrzegać jako

jednowarstwową sieć neuronową. Ten kod zwraca ocenę między 0 a 1, która podczas obliczania straty jest porównywana z rzeczywistą etykietą klasy (0 lub 1) (listing A.2).

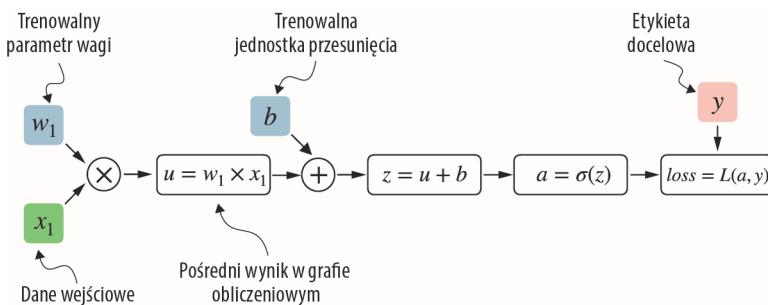
Listing A.2. Przebieg w przód regresji logistycznej

```
import torch.nn.functional as F
y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2])
b = torch.tensor([0.0])
z = x1 * w1 + b
a = torch.sigmoid(z)
loss = F.binary_cross_entropy(a, y)
```

Ta instrukcja import jest powszechną konwencją w PyTorch, mającą na celu zapobieżenie długim wierszom kodu.

Faktyczna etykieta
Cecha wejściowa
Parametr wagi
Jednostka przesunięcia
Wejście sieci
Aktywacja i wyjście

Jeśli nie wszystkie komponenty w powyższym kodzie mają dla Ciebie sens, nie martw się. Celem tego przykładu nie jest implementacja klasyfikatora regresji logistycznej, ale raczej zilustrowanie, w jaki sposób można myśleć o sekwencji obliczeń jako grafie obliczeniowym podobnym do pokazanego na rysunku A.7.



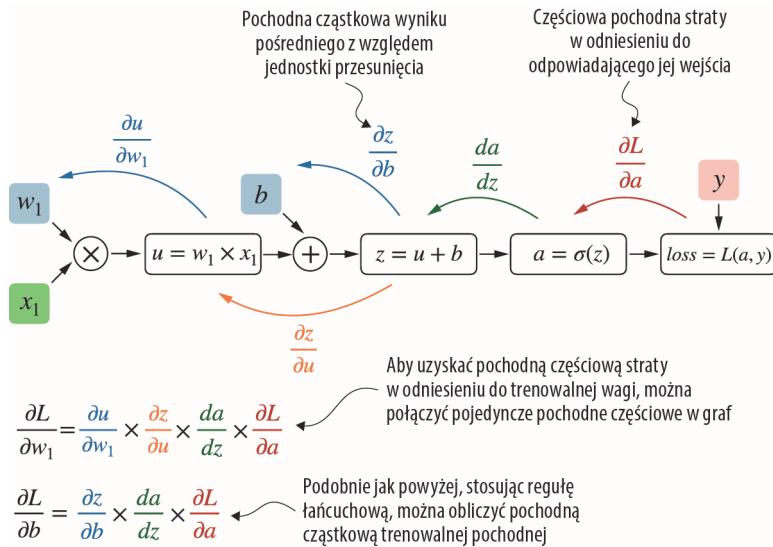
Rysunek A.7. Przebieg regresji logistycznej w przód jako graf obliczeniowy. Cecha wejściowa x_1 jest mnożona przez wagę modelu w_1 i po dodaniu przesunięcia (odchylenia) jest przekazywana do funkcji aktywacji σ . Strata jest obliczana przez porównanie wyniku modelu a z daną etykietą y .

W gruncie rzeczy PyTorch buduje taki graf obliczeniowy w tle i można go użyć do obliczenia gradientów funkcji straty w odniesieniu do parametrów modelu (tutaj w_1 i b) w celu przeskolenia modelu.

A.4. Łatwe automatyczne różniczkowanie

Gdy wykonujesz obliczenia w PyTorch, to jeśli jeden z węzłów końcowych ma atrybut `requires_grad` ustawiony na `True`, framework domyślnie buduje wewnętrznie graf obliczeniowy. Cechą ta przydaje się do obliczania gradientów. Gradienty są wymagane

podczas szkolenia sieci neuronowych z użyciem popularnego algorytmu propagacji wstecznej, który można uznać za implementację zilustrowanej na rysunku A.8 reguły łańcuchowej rachunku różniczkowego w odniesieniu do sieci neuronowych.



Rysunek A.8. Najczęstszym sposobem obliczania gradientów strat w grafie obliczeniowym jest zastosowanie reguły łańcuchowej od prawej do lewej, zwanej również automatycznym różniczkowaniem modelu w trybie wstecznym lub propagacją wsteczną. Zaczynamy od warstwy wyjściowej (lub wartości straty) i cofamy się przez sieć do warstwy wejściowej. Celem tego działania jest obliczenie w odniesieniu do każdego parametru (wag i odchyleń) w sieci gradientu straty, który informuje, w jaki sposób te parametry są aktualizowane podczas szkolenia

POCHODNE CZĄSTKOWE I GRADIENTY

Pochodne cząstkowe, mierzące szybkość, z jaką funkcja zmienia się w odniesieniu do jednej z jej zmiennych, przedstawiono na rysunku A.8. Gradient to wektor zawierający wszystkie pochodne cząstkowe funkcji wielu zmiennych, czyli funkcji z więcej niż jedną zmienną na wejściu.

Jeśli nie znasz lub nie pamiętasz zagadnień związanych z pochodnymi cząstkowymi, gradientami lub regułą łańcuchową rachunku różniczkowego, nie martw się. Wszystko, co trzeba wiedzieć na ogólnym poziomie, na potrzeby analizowania przykładów w tej książce, to to, że reguła łańcuchowa jest sposobem obliczania gradientów funkcji straty z uwzględnieniem parametrów modelu w grafie obliczeniowym. Zastosowanie reguły łańcuchowej pozwala uzyskać informacje potrzebne do aktualizacji każdego parametru w celu zminimalizowania funkcji straty, co służy jako miara wydajności modelu z użyciem takich metod jak spadek gradientowy. Do implementacji obliczeń w pętli szkoleniowej w PyTorch powróć w podrozdziale A.7.

Jak to wszystko ma się do silnika automatycznego różniczkowania (autograd), wspomnianego wcześniej drugiego komponentu biblioteki PyTorch? Silnik autograd framework PyTorch konstruuje w tle graf obliczeniowy, pozwalający śledzić każdą operację wykonywaną na tensorach. Następnie, przez wywołanie funkcji grad, można obliczyć gradient straty dotyczący parametru modelu $w1$, tak jak pokazałem na lisingu A.3.

Listing A.3. Obliczanie gradientów przez silnik autograd

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True) ←
grad_L_b = grad(loss, b, retain_graph=True)
```

Domyślnie po obliczeniu gradientów PyTorch niszczy graf obliczeniowy, aby zwolnić pamięć. Jednakże ponieważ wkrótce ponownie użyjemy tego grafu obliczeniowego, ustawiamy retain_graph=True, aby pozostać w pamięci

Uzyskane wartości strat z uwzględnieniem parametrów modelu są następujące:

```
print(grad_L_w1)
print(grad_L_b)
```

Uruchomienie tego kodu spowoduje wyświetlenie następujących wyników:

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

W tym przykładzie używaliśmy funkcji grad ręcznie. Taki sposób można wykorzystać do eksperymentowania z pojęciami, debugowania kodu i demonstrowania pojęć. Jednak w praktyce PyTorch zapewnia więcej wysokopoziomowych narzędzi do automatyzacji tego procesu. Na przykład można wywołać metodę backward w odniesieniu do wartości straty, a PyTorch obliczy gradienty wszystkich węzłów liści grafu, które będą przechowywane za pośrednictwem atrybutów .grad tensorów:

```
loss.backward()
print(w1.grad)
print(b.grad)
```

Oto uzyskane wyniki:

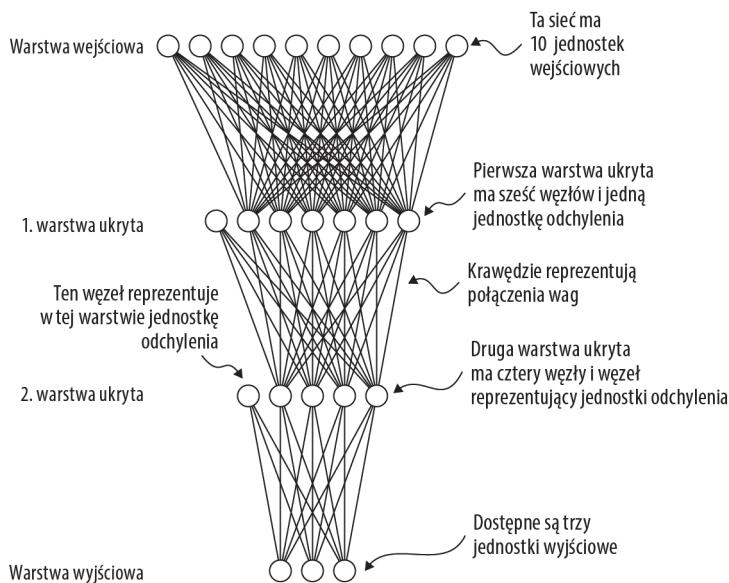
```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

Dostarczyłem Ci sporo informacji, a pojęcia związane z rachunkiem różniczkowym mogą Cię przytaczać, ale nie martw się. Chociaż żargon związany z rachunkiem

różniczkowym jest środkiem do wyjaśnienia komponentu autograd frameworka PyTorch, wszysktko, co musisz wiedzieć, to to, że PyTorch wykonuje za nas obliczenia różniczkowe za pomocą metody backward — nie musisz ręcznie obliczać żadnych pochodnych ani gradientów.

A.5. Implementacja wielowarstwowych sieci neuronowych

Następnie skupimy się na PyTorch jako bibliotece do implementacji głębokich sieci neuronowych. W ramach konkretnego przykładu spójrzmy na wielowarstwowy perceptron, w pełni połączoną sieć neuronową, którą pokazałem na rysunku A.9.



Rysunek A.9. Wielowarstwowy perceptron z dwiema warstwami ukrytymi. Każdy węzeł reprezentuje jednostkę w odpowiedniej warstwie. Dla celów ilustracyjnych każda warstwa ma bardzo małą liczbę węzłów.

Aby podczas implementacji sieci neuronowej w PyTorch zdefiniować własną architekturę sieci, można stworzyć pochodną klasy `torch.nn.Module`. Klasa bazowa `Module` dostarcza wielu funkcji ułatwiających budowanie modeli i ich szkolenie. Na przykład pozwala hermetyzować warstwy i działania oraz śledzić parametry modelu.

W utworzonej klasie potomnej definiujemy w konstruktorze `__init__` warstwy sieci, a w metodzie `forward` określamy sposób interakcji warstw. Metoda `forward` opisuje sposób, w jaki dane wejściowe są przekazywane przez sieć i łączą się w graf obliczeniowy.

W przeciwieństwie do tego, metoda backward, której zwykle nie trzeba implementować samodzielnie, jest używana podczas uczenia do obliczania gradientów funkcji straty przy danych parametrach modelu (patrz podrozdział A.7). Dla zilustrowania typowego użycia klasy `Module` kod z listingu A.4 implementuje klasyczny wielowarstwowy perceptron z dwiema ukrytymi warstwami.

Listing A.4. Wielowarstwowy perceptron z dwiema warstwami ukrytymi

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(
            # pierwsza warstwa ukryta
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),
            # druga warstwa ukryta
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),
            # warstwa wyjściowa
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

The diagram shows the code from Listing A.4 with several annotations:

- A callout points to the first two lines of the class definition, explaining that encoding the number of inputs and outputs as variables allows for reuse of the same code for datasets with different numbers of features and classes.
- An annotation points to the first two lines of the first layer definition, stating that the Linear layer takes the number of input and output nodes as arguments.
- An annotation points to the twoReLU layers, stating that between hidden layers, non-linear activation functions are applied.
- An annotation points to the second layer's definition, stating that the number of output nodes must match the number of input nodes of the next layer.
- A final annotation points to the last line of the forward method, stating that the outputs of the final layer are called logits.

Następnie można utworzyć nowy obiekt sieci neuronowej w następujący sposób:

```
model = NeuralNetwork(50, 3)
```

Aby przed użyciem tego nowego obiektu `model` zobaczyć podsumowanie jego struktury, możesz wywołać na modelu funkcję `print`:

```
print(model)
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Zauważ, że w implementacji klasy `NeuralNetwork` używamy klasy `Sequential`. Użycie klasy `Sequential` nie jest konieczne, ale jeśli, tak jak w tym przypadku, mamy zbiór warstw, które mają być uruchomione w określonej kolejności, możemy ułatwić sobie

życie. Dzięki temu po utworzeniu egzemplarza klasy `Sequential` za pomocą instrukcji `self.layers = Sequential(...)` w konstruktorze `__init__`, zamiast wywoływać każdą warstwę indywidualnie w metodzie `forward` sieci neuronowej, trzeba jedynie wywołać `self.layers`.

Następnie sprawdźmy całkowitą liczbę możliwych do wyuczenia parametrów tego modelu:

```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Łączna liczba możliwych do wyuczenia parametrów modelu:", num_params)
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
Łączna liczba możliwych do wyuczenia parametrów modelu: 2213
```

Każdy parametr, dla którego zastosowano ustawienie `requires_grad=True`, liczy się jako parametr możliwy do wyuczenia i podczas szkolenia będzie aktualizowany (patrz podrozdział A.7).

W przypadku modelu sieci neuronowej z dwiema pokazanymi wcześniej warstwami ukrytymi te możliwe do wyuczenia parametry są zawarte w warstwach `torch.nn.Linear`. Warstwa `Linear` mnoży dane wejściowe przez macierz wag i dodaje wektor odchylenia. Warstwę tę czasami określa się jako warstwę *sprzężenia zwrotnego w przód* (ang. *feedforward*) lub w pełni połączoną (ang. *fully connected*).

Na podstawie wyniku działania instrukcji `print(model)`, z której tutaj skorzystaliśmy, możemy zauważyć, że pierwsza warstwa `Linear` znajduje się na pozycji indeksu 0 w atrybucie `layers`. Dostęp do odpowiedniej macierzy parametrów wag można uzyskać w następujący sposób:

```
print(model.layers[0].weight)
```

Oto wynik działania tego kodu:

```
Parameter containing:
tensor([[ 0.1174, -0.1350, -0.1227, ...,  0.0275, -0.0520, -0.0192],
       [-0.0169,  0.1265,  0.0255, ..., -0.1247,  0.1191, -0.0698],
       [-0.0973, -0.0974, -0.0739, ..., -0.0068, -0.0892,  0.1070],
       ...,
       [-0.0681,  0.1058, -0.0315, ..., -0.1081, -0.0290, -0.1374],
       [-0.0159,  0.0587, -0.0916, ..., -0.1153,  0.0700,  0.0770],
       [-0.1019,  0.1345, -0.0176, ...,  0.0114, -0.0559, -0.0088]], requires_grad=True)
```

Ponieważ ta duża macierz nie jest wyświetlana w całości, użyjmy atrybutu `.shape`, aby pokazać jej wymiary:

```
print(model.layers[0].weight.shape)
```

Wynik jest następujący:

```
torch.Size([30, 50])
```

(W podobny sposób, za pomocą wywołania `model.layers[0].bias`, można uzyskać dostęp do wektora `bias`).

Macierz wag w tym przykładzie ma wymiary 30×50 i, jak można zauważyć, wartość `requires_grad` jest ustawiona na `True`, co oznacza, że model może się uczyć jej elementów — jest to domyślne ustawienie dla wag i odchyłeń w warstwie `torch.nn.Linear`.

Jeśli spróbujesz uruchomić powyższy kod na swoim komputerze, liczby w macierzy wag prawdopodobnie będą różnić się od tych pokazanych powyżej. Wagi modelu są inicjowane małymi liczbami losowymi, które różnią się za każdym razem, gdy tworzysz egzemplarz sieci. W uczeniu głębokim inicjalizacja wag modelu małymi liczbami losowymi jest pożądana, ponieważ pozwala wyeliminować symetrię podczas szkolenia. W przeciwnym razie w trakcie propagacji wstępnej węzły wykonywałyby te same działania i aktualizacje, przez co sieć nie byłaby w stanie uczyć się złożonych mapowań od wejść do wyjść.

O ile jednak nadal chcesz używać małych liczb losowych jako wartości początkowych dla wag warstw, możesz sprawić, aby inicjalizacja liczb losowych była powtarzalna. Aby tak się stało, wystarczy skorzystać z wywołania `manual_seed` i przekazać do generatora liczb losowych PyTorch ziarno losowości:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

Wynik działania tego kodu jest następujący:

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702,  ...,  0.0222,  0.1260,  0.0865],
       [ 0.0502,  0.0307,  0.0333,  ...,  0.0951,  0.1134, -0.0297],
       [ 0.1077, -0.1108,  0.0122,  ...,  0.0108, -0.1049, -0.1063],
       ...,
       [-0.0787,  0.1259,  0.0803,  ...,  0.1218,  0.1303, -0.1351],
       [ 0.1359,  0.0175, -0.0673,  ...,  0.0674,  0.0676,  0.1058],
       [ 0.0790,  0.1343, -0.0293,  ...,  0.0344, -0.0971, -0.0509]],

      requires_grad=True)
```

Po dokonaniu inspekcji egzemplarza `NeuralNetwork` sprawdźmy pokrótko, w jaki sposób jest on wykorzystywany podczas przebiegu w przód:

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

Wynik jest następujący:

```
tensor([[-0.1262,  0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

W powyższym kodzie wygenerowaliśmy jako przykład wejścia pojedynczą losową próbki szkoleniową `X` (należy pamiętać, że sieć w tym przykładzie oczekuje 50-wymiarowych wektorów cech) i przekazaliśmy ją do modelu, który zwrócił trzy wyniki. Gdy wywołujemy `model(x)`, następuje automatyczne uruchomienie przejścia modelu „w przód”.

Przejście w przód odnosi się do obliczania tensorów wyjściowych na podstawie tensorów wejściowych. Obejmuje to przekazanie danych wejściowych przez wszystkie

warstwy sieci neuronowej, od warstwy wejściowej poprzez warstwy ukryte aż do warstwy wyjściowej.

Te trzy zwrócone liczby odpowiadają ocenie przypisanej do każdego z trzech węzłów wyjściowych. Zauważ, że tensor wyjściowy zawiera również wartość `grad_fn`.

Tutaj `grad_fn=<AddmmBackward0>` reprezentuje funkcję ostatnio używaną do obliczenia zmiennej w grafie obliczeniowym. W szczególności `grad_fn=<AddmmBackward0>` oznacza, że tensor, który sprawdzamy, został utworzony za pomocą operacji mnożenia i dodawania macierzy. Framework PyTorch wykorzysta te informacje podczas obliczania gradientów na etapie propagacji wstecznej. Część `<AddmmBackward0>` parametru `grad_fn=<AddmmBackward0>` określa wykonywaną operację. W tym przypadku jest to operacja `Addmm`. `Addmm` oznacza mnożenie macierzy (`mm`), po którym następuje dodawanie (`Add`).

Jeśli chcesz użyć sieci bez szkolenia lub propagacji wstecznej — na przykład jeśli używasz jej do prognozowania po szkoleniu — konstruowanie grafu obliczeniowego na potrzeby propagacji wstecznej może być marnotrawstwem, ponieważ wymaga niepotrzebnych obliczeń i zużywa dodatkową pamięć. Z tego powodu, gdy używasz modelu do wnioskowania (np. tworzenia prognoz), a nie szkolenia, najlepszą praktyką jest użycie menedżera kontekstu `torch.no_grad()`. Jest to informacja dla frameworka PyTorch, że nie ma potrzeby śledzenia gradientów. Może to skutkować znacznymi oszczędnościami pamięci i mocy obliczeniowej:

```
with torch.no_grad():
    out = model(X)
print(out)
```

Wynik działania tego kodu jest następujący:

```
tensor([-0.1262,  0.1080, -0.1792])
```

W kodzie korzystającym z frameworka PyTorch powszechną praktyką jest kodowanie modeli w taki sposób, aby zwracały wyjścia ostatniej warstwy (logity) bez przekazywania ich do nieliniowej funkcji aktywacji. To dlatego, że funkcje strat powszechnie używane we frameworku PyTorch łączą w jednej klasie operację `softmax` (lub `sigmoid` w przypadku klasyfikacji binarnej) z ujemną stratą logarytmu prawdopodobieństwa. Chodzi o wydajność obliczeniową i stabilność. Jeśli więc chcesz obliczyć prawdopodobieństwo przynależności do klasy na potrzeby wykonywanych prognoz, musisz jawnie wywołać funkcję `softmax`:

```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

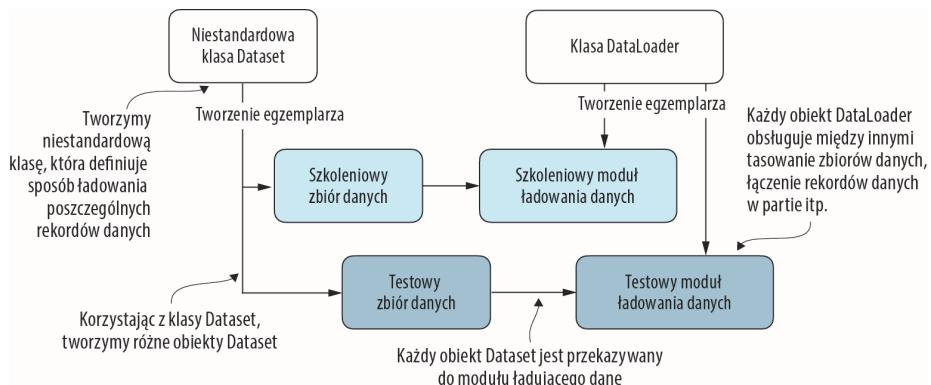
Oto wynik działania tego kodu:

```
tensor([0.3113, 0.3934, 0.2952]))
```

Wartości można teraz interpretować jako prawdopodobieństwa przynależności do klas, które sumują się do 1. Wartości dla tego losowego wejścia są w przybliżeniu równe, co w przypadku losowo zainicjowanego modelu, bez szkolenia, jest oczekiwane.

A.6. Konfigurowanie wydajnych mechanizmów ładujących dane

Przed przystąpieniem do szkolenia modelu trzeba pokrótkę omówić tworzenie wydajnych mechanizmów ładujących dane framework PyTorch, po których będziemy iterować podczas szkolenia. Ogólną koncepcję ładowania danych w PyTorch zilustrował na rysunku A.10.



Rysunek A.10. PyTorch implementuje klasy Dataset i DataLoader. Klasa Dataset służy do tworzenia egzemplarzy obiektów definiujących sposób ładowania każdego rekordu danych. Klasa DataLoader obsługuje tasowanie danych i łączenie ich w partie

Zgodnie z rysunkiem A.10 zaimplementujemy niestandardową klasę Dataset, której użyjemy do utworzenia szkoleniowego i testowego zbioru danych, by następnie wykorzystać ją do utworzenia mechanizmów ładujących dane. Zaczniemy od utworzenia prostego zbioru danych składającego się z pięciu próbek szkoleniowych, po dwie cechy każda. Razem z próbками szkoleniowymi tworzymy również tensor zawierający odpowiednie etykiety klas: trzy przykłady należą do klasy 0, a kolejne dwa do klasy 1. Ponadto utworzymy zestaw testowy składający się z dwóch elementów. Kod do utworzenia tego zbioru danych przedstawiłem na listingu A.5.

Listing A.5. Utworzenie prostego zbioru danych

```

X_train = torch.tensor([
    [-1.2, 3.1],
    [-0.9, 2.9],
    [-0.5, 2.6],
    [2.3, -1.1],
    [2.7, -1.5]
])
y_train = torch.tensor([0, 0, 0, 1, 1])
X_test = torch.tensor([
    [-0.8, 2.8],
    [0.5, -1.2]
])

```

```
[2.6, -1.6],  
])  
y_test = torch.tensor([0, 1])
```

UWAGA PyTorch wymaga, aby etykiety klas zaczynały się od 0. Z kolei największa wartość etykiety klasy nie powinna przekraczać liczby węzłów wyjściowych pomniejszonych o 1 (ponieważ indeksy tablic w Pythonie zaczynają się od zera). Jeśli więc mamy etykiety klas 0, 1, 2, 3 i 4, to warstwa wyjściowa sieci neuronowej powinna składać się z pięciu węzłów.

Następnie stworzymy niestandardową klasę zbioru danych ToyDataset, będącą klasą potomną klasy bazowej frameworka PyTorch Dataset. Jej implementację zamieściłem na listingu A.6.

Listing A.6. Definicja niestandardowej klasy Dataset

```
from torch.utils.data import Dataset  
  
class ToyDataset(Dataset):  
    def __init__(self, X, y):  
        self.features = X  
        self.labels = y  
  
    def __getitem__(self, index):  
        one_x = self.features[index]  
        one_y = self.labels[index]  
        return one_x, one_y  
  
    def __len__(self):  
        return self.labels.shape[0]  
  
train_ds = ToyDataset(X_train, y_train)  
test_ds = ToyDataset(X_test, y_test)
```

Instrukcje pobierania dokładnie jednego rekordu danych i odpowiadającej mu etykiety

Instrukcje dotyczące zwracania całkowitej długości zbioru danych

Celem tej niestandardowej klasy ToyDataset jest utworzenie egzemplarza klasy Data Loader frameworka PyTorch. Zanim jednak przejdziemy do kolejnego kroku, przyjrzyjmy się pokrótko ogólnej strukturze kodu klasy ToyDataset.

Trzy główne komponenty niestandardowej klasy Dataset we frameworku PyTorch to konstruktor `__init__`, metoda `__getitem__` i metoda `__len__` (patrz listing A.6). W metodzie `__init__` ustawiamy atrybuty, do których można później uzyskać dostęp w metodach `__getitem__` i `__len__`. Mogą to być ścieżki do plików, obiekty plików, mechanizmy konektorów baz danych itd. Ponieważ utworzyliśmy zbiór danych tensora, który znajduje się w pamięci, po prostu przypisujemy `X` i `y` do tych atrybutów, które są symbolami zastępczymi dla obiektów tensora.

W metodzie `__getitem__` definiujemy instrukcje zwracania dokładnie jednego elementu ze zbioru danych za pośrednictwem atrybutu `index`. Dotyczy to cech i etykiet klas odpowiadających pojedynczej próbce szkoleniowej lub egzemplarzowi testowemu (mechanizm lądujący dane dostarczy mechanizmu `index`, który wkrótce omówię).

Na koniec metoda `_len_` zawiera instrukcje pobierania długości zbioru danych. Tutaj, aby zwrócić liczbę wierszy w tablicy cech, używamy atrybutu `.shape` tensora. W przypadku szkoleniowego zbioru danych mamy pięć wierszy, które warto sprawdzić:

```
print(len(train_ds))
```

Wynik jest następujący:

5

Po zdefiniowaniu klasy `Dataset` framework PyTorch, której można użyć w odniesieniu do przykładowego zbioru danych, możemy skorzystać z klasy `DataLoader` w celu pobrania próbek ze zbioru danych w sposób pokazany na listingu A.7.

Listing A.7. Utworzenie komponentów ładowania danych

```
from torch.utils.data import DataLoader

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,           ← Utworzony wcześniej egzemplarz klasy ToyDataset służy
    batch_size=2,                ← jako dane wejściowe dla komponentu ładującego dane
    shuffle=True,               ← Czy należy przetasować dane
    num_workers=0               ← Liczba procesów w tle
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,              ← Tasowanie testowego zbioru danych nie jest konieczne
    num_workers=0
)
```

Po utworzeniu egzemplarza mechanizmu ładującego dane szkoleniowe można po nim iterować. Iteracja po klasie `test_loader` działa podobnie, ale pominąłem ją dla zwięzłości przykładu:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Partia {idx+1}:", x, y)
```

Wynik działania tego kodu jest następujący:

```
Partia 1: tensor([[-1.2000,  3.1000],
                   [-0.5000,  2.6000]]) tensor([0, 0])
Partia 2: tensor([[ 2.3000, -1.1000],
                   [-0.9000,  2.9000]]) tensor([1, 0])
Partia 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

Jak można zauważyć na podstawie powyższych danych wyjściowych, obiekt `train_loader` iteruje po szkoleniowym zbiorze danych — każda próbka szkoleniowa jest przetwarzana dokładnie raz. Jest to tak zwana epoka szkoleniowa. Ponieważ za pomocą instrukcji `torch.manual_seed(123)` zastosowaliśmy w przykładzie ziarno losowości

generatora liczb losowych, powinieneś otrzymać dokładnie taką samą kolejność tasowania przykładów szkoleniowych jak ta, którą uzyskałem powyżej. Jeśli jednak wykonasz iterację po zbiorze danych po raz drugi, zauważysz, że kolejność tasowania się zmieni. Ma to na celu zapobieżenie podczas szkolenia uwiecznieniu głębokich sieci neuronowych w powtarzających się cyklach aktualizacji.

W tym przykładzie określiliśmy rozmiar drugiej partii, ale trzecia zawiera tylko jedną próbkę. To dlatego, że mamy pięć przykładów szkoleniowych, a liczba 5 nie dzieli się przez 2 bez reszty. W praktyce wykorzystanie znacznie mniejszej partii jako ostatniej w epoce treningowej może zakłócić zbieżność w trakcie szkolenia. Aby temu zapobiec, należy ustawić `drop_last=True`. To spowoduje odrzucenie ostatniej partii w każdej epoce, co pokazałem na listingu A.8.

Listing A.8. Szkoleniowy mechanizm ładujący, który odrzuca ostatnią partię

```
train_loader = DataLoader(  
    dataset=train_ds,  
    batch_size=2,  
    shuffle=True,  
    num_workers=0,  
    drop_last=True  
)
```

Teraz podczas iterowania po szkoleniowym mechanizmie ładującym dane można zauważyc, że ostatnia partia została pominięta:

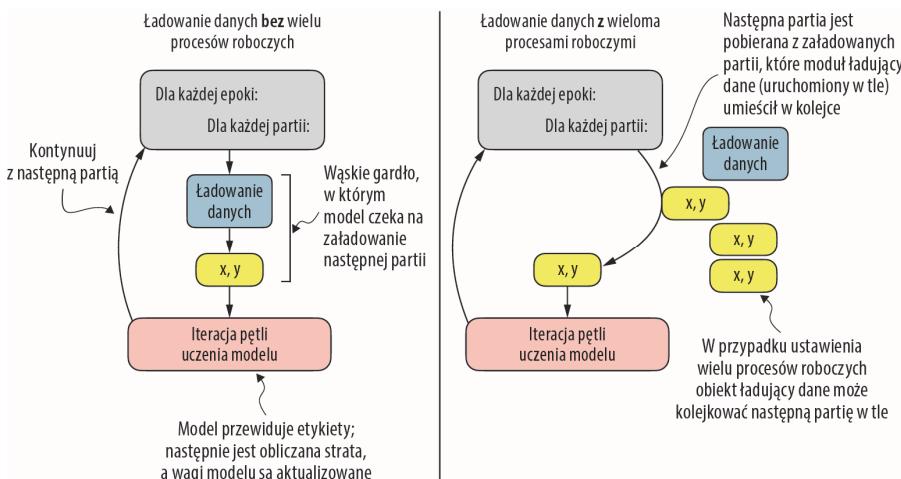
```
for idx, (x, y) in enumerate(train_loader):  
    print(f"Partia {idx+1}:", x, y)
```

Wynik działania tego kodu jest następujący:

```
Partia 1: tensor([[-0.9000,  2.9000],  
                  [ 2.3000, -1.1000]]) tensor([0, 1])  
Partia 2: tensor([[ 2.7000, -1.5000],  
                  [-0.5000,  2.6000]]) tensor([1, 0])
```

Na koniec omówię ustawienie `num_workers=0` w klasie `DataLoader`. Ten parametr w klasie `DataLoader` frameworka PyTorch ma kluczowe znaczenie dla zrównoleglenia ładowania i wstępnego przetwarzania danych. Gdy wartość `num_workers` jest ustaliona na 0, ładowanie danych będzie wykonywane w głównym procesie, a nie w oddzielnych procesach roboczych. Można by pomyśleć, że to nie problem, ale w przypadku szkolenia większych sieci na układach GPU łatwo w ten sposób doprowadzić do znacznych spowolnień w trakcie uczenia modeli. Zamiast skupiać się wyłącznie na przetwarzaniu modeli uczenia głębokiego, procesor musi również poświęcić czas na załadowanie danych i ich wstępne przetworzenie. W rezultacie układ GPU może pozostawać bezczynny w oczekiwaniu na zakończenie zadań wykonywanych przez CPU. W przeciwieństwie do tej sytuacji, gdy `num_workers` jest ustalone na liczbę większą niż 0, uruchamianych jest wiele procesów roboczych w celu równoległego ładowania danych. Dzięki

temu główny proces może się skupić na uczeniu modelu i lepszym wykorzystaniu zasobów systemu (rysunek A.11).



Rysunek A.11. Ładowanie danych bez wielu procesów roboczych (ustawienie num_workers=0) spowoduje powstanie wąskiego gardła ładowania danych, kiedy to model pozostanie bezczynny do momentu załadowania następnej partii (lewa strona rysunku). W przypadku ustawienia wielu procesów roboczych obiekt ładowujący dane może kolejkować następną partię w tle (prawa strona rysunku)

Jeśli jednak pracujesz z bardzo małymi zbiorami danych, ustawienie num_workers na 1 lub większą wartość może nie być konieczne, ponieważ całkowity czas szkolenia i tak zajmuje tylko ułamki sekundy. Jeśli zatem pracujesz z niewielkimi zbiorami danych lub korzystasz ze środowisk interaktywnych, takich jak notatniki Jupyter, zwiększenie wartości num_workers może nie zapewnić zauważalnego przyspieszenia. W gruncie rzeczy może prowadzić do pewnych niedogodności. Jednym z potencjalnych problemów jest narzut związany z uruchamianiem wielu procesów roboczych. W przypadku niewielkich zbiorów danych to uruchamianie może trwać dłużej niż faktyczne ładowanie danych.

Co więcej, w przypadku notatników Jupyter ustawienie num_workers na wartość większą niż o czasami może prowadzić do problemów związanych z dzieleniem zasobów między różne procesy, co skutkuje błędami lub zawieszaniem się notatnika. Z tego względu trzeba rozumieć ten kompromis i podjąć przemyślaną decyzję dotyczącą ustawienia parametru num_workers. Przemyślane ustawienie tego parametru można uznać za korzystne narzędzie, ale w celu uzyskania optymalnych wyników należy go dostosować do konkretnego rozmiaru zbioru danych i środowiska obliczeniowego.

Z mojego doświadczenia wynika, że w przypadku wielu rzeczywistych zbiorów danych ustawienie num_workers=4 zwykle prowadzi do optymalnej wydajności. Optymalne ustawienie zależy jednak od sprzętu i używanego do ładowania próbek szkoleniowych kodu zdefiniowanego w klasie Dataset.

A.7. Typowa pętla szkoleniowa

Spróbujmy teraz przeszkolić sieć neuronową na przykładowym zbiorze danych. Kod szkolenia przedstawiłem na listingu A.9.

Listing A.9. Szkolenie sieci neuronowej z użyciem frameworka PyTorch

```
import torch.nn.functional as F

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)           ← Zbiór danych
                                                               ma dwie cechy i dwie klasy
    optimizer = torch.optim.SGD(
        model.parameters(), lr=0.5
    )                                                               ← Optymalizator musi wiedzieć,
                                                               które parametry zoptymalizować

num_epochs = 3
for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):
        logits = model(features)

        loss = F.cross_entropy(logits, labels)           ← Ustawienie gradientów z poprzedniej rundy na 0,
                                                       aby zapobiec niezamierzonej akumulacji gradientu
        optimizer.zero_grad()                         ← Obliczenie gradientów strat
        loss.backward()                                ← z uwzględnieniem parametrów modelu
        optimizer.step()                             ← Optymalizator wykorzystuje gradienty
                                                       do aktualizacji parametrów modelu

    ### LOGOWANIE
    print(f"Epoka: {epoch+1:03d}/{num_epochs:03d}"
          f" | Partia {batch_idx:03d}/{len(train_loader):03d}"
          f" | Strata zbioru szkoleniowego: {loss:.2f}")

model.eval()                                                     ← # Wstaw opcjonalny kod oceny modelu
```

Uruchomienie powyższego kodu spowoduje wyświetlenie następującego wyniku:

```
Epoka: 001/003 | Partia 000/002 | Strata zbioru szkoleniowego: 0.75
Epoka: 001/003 | Batch 001/002 | Strata zbioru szkoleniowego 0.65
Epoka: 002/003 | Batch 000/002 | Strata zbioru szkoleniowego 0.44
Epoka: 002/003 | Batch 001/002 | Trainl Loss: 0.13
Epoka: 003/003 | Batch 000/002 | Strata zbioru szkoleniowego 0.03
Epoka: 003/003 | Batch 001/002 | Strata zbioru szkoleniowego 0.00
```

Jak można zauważyć, wartość straty po trzech epokach osiąga wartość 0, co oznacza, że dla tego zbioru szkoleniowego model uzyskał zbieżność. W tym przypadku inicjujemy model z dwoma wejściami i dwoma wyjściami, ponieważ przykładowy zbiór danych obejmuje dwie cechy wejściowe i dwie etykiety klas. Użyliśmy stochastycznego optymalizatora gradientowego (SGD) ze współczynnikiem uczenia (lr) wynoszącym 0.5. Szybkość uczenia się jest hiperparametrem, co oznacza, że jest to ustawienie, z którym trzeba eksperymentować na podstawie obserwacji strat. W idealnej sytuacji

powinniśmy wybrać taki współczynnik uczenia, aby strata zbiegała się po określonej liczbie epok — liczba epok jest kolejnym hiperparametrem do wyboru.

Ćwiczenie A.3

Ile parametrów ma sieć neuronowa przedstawiona na listingu A.9?

W praktyce, aby znaleźć optymalne ustawienia hiperparametrów, często używa się trzeciego zbioru danych, tak zwanego zbioru walidacyjnego. Walidacyjny zbiór danych jest podobny do zbioru testowego. Jednak o ile ze zbioru testowego korzystamy tylko raz, aby uniknąć stronniczości oceny, o tyle aby dostroić ustawienia modelu, ze zbioru walidacyjnego korzysta się wiele razy.

Użyliśmy także nowych metod — `model.train()` i `model.eval()`. Jak sugerują ich nazwy, te ustawienia są używane do przełączania modelu w tryb szkolenia i oceny. Jest to konieczne w przypadku komponentów, które zachowują się inaczej podczas uczenia i wnioskowania, takich jak *warstwa dropout* lub *warstwa normalizacji parti*. Ponieważ w klasie `NeuralNetwork` nie mamy dropoutu ani innych komponentów, na które wpływają te ustawienia, użycie metod `model.train()` i `model.eval()` w poprzednim kodzie jest zbędne. Ich uwzględnienie jest jednak dobrą praktyką, która pozwala uniknąć nieoczekiwanych zachowań w przypadku zmiany architektury modelu lub ponownie wykorzystać kod do szkolenia innego modelu.

Jak wspomniałem wcześniej, logity przekazujemy bezpośrednio do funkcji straty `cross_entropy`, która ze względu na wydajność i stabilność numeryczną stosuje wewnętrznie funkcję `softmax`. Następnie wywołanie `loss.backward()` oblicza gradienty w grafie obliczeniowym, skonstruowanym w tle przez framework PyTorch. Metoda `optimizer.step()` wykorzystuje gradienty do aktualizacji parametrów modelu w taki sposób, aby zminimalizować wartość straty. W przypadku optymalizatora SGD oznacza to pomnożenie gradientów przez współczynnik uczenia i dodanie do parametrów skalowanego ujemnego gradientu.

UWAGA Aby zapobiec niepożądanej akumulacji gradientów, należy zadbać o uwzględnienie w każdej rundzie aktualizacji wywołania `optimizer.zero_grad()`. Ma to na celu zresetowanie gradientów do 0. Bez tego wywołania gradienty będą się akumulować, co jest niepożądane.

Po przeszkołeniu modelu można wykorzystać go do tworzenia prognoz:

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

Wyniki są następujące:

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
```

```
[ 2.0944, -3.1820],
[-1.4814,  1.4816],
[-1.7176,  1.7342]])
```

Aby uzyskać prawdopodobieństwo przynależności do klasy, można użyć funkcji softmax z biblioteki PyTorch:

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

Oto wynik:

```
tensor([[ 0.9991,    0.0009],
       [ 0.9982,    0.0018],
       [ 0.9949,    0.0051],
       [ 0.0491,    0.9509],
       [ 0.0307,    0.9693]])
```

Rozważmy pierwszy wiersz w powyższym wyniku działania kodu. Pierwsza wartość (kolumna) oznacza, że próbka szkoleniowa ma prawdopodobieństwo przynależności do klasy o rzędu 99,91% i prawdopodobieństwo przynależności do klasy 1 na poziomie 0,09% (aby wyniki były bardziej czytelne, wykorzystano wywołanie set_print →options).

Aby przekonwertować te wartości na prognozy klas, można użyć funkcji argmax framework PyTorch, która przy ustawieniu dim=1 zwraca pozycję indeksu najwyższej wartości w każdym wierszu (ustawienie dim=0 zwróciłoby zamiast tego najwyższą wartość w każdej kolumnie):

```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

Uruchomienie tego kodu spowoduje wyświetlenie następujących wyników:

```
tensor([0, 0, 0, 1, 1])
```

Warto zauważyć, że obliczanie prawdopodobieństwa softmax w celu uzyskania etykiet klas nie jest konieczne. Funkcję argmax można również zastosować bezpośrednio do logitów (wyjść):

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

Oto wynik:

```
tensor([0, 0, 0, 1, 1])
```

W tym przykładzie obliczyliśmy prognozowane etykiety dla zbioru danych szkoleniowych. Ponieważ zbiór danych szkoleniowych nie jest zbyt obszerny, można go ręcznie porównać z rzeczywistymi etykietami szkoleniowymi. Łatwo zauważać, że model udzielił 100% poprawnych odpowiedzi. Można to sprawdzić za pomocą operatora porównania ==:

```
predictions == y_train
```

Wyniki są następujące:

```
tensor([True, True, True, True, True])
```

Aby policzyć liczbę poprawnych prognoz, można skorzystać z wywołania `torch.sum`:

```
torch.sum(predictions == y_train)
```

Oto uzyskany wynik:

```
5
```

Ponieważ zbiór danych składa się z pięciu próbek szkoleniowych, model udziela pięciu poprawnych prognoz na 5, co daje $5/5 \cdot 100\% = 100\%$ dokładności prognoz.

Aby uogólnić obliczanie dokładności prognoz, zaimplementujmy funkcję `compute_accuracy`, której kod pokazałem na listingu A.10.

Listing A.10. Funkcja obliczająca dokładność prognoz

```
def compute_accuracy(model, dataloader):
    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):
        with torch.no_grad():
            logits = model(features)
            predictions = torch.argmax(logits, dim=1)
            compare = labels == predictions
            correct += torch.sum(compare)
            total_examples += len(compare)

    return (correct / total_examples).item()
```

- Zwrotka tensor złożony z wartości True (False), w zależności od tego, czy etykiety są zgodne
- Operacja sumy zlicza liczbę wartości True
- Ułamek poprawnych prognoz, wartość między 0 a 1. Metoda item() zwraca wartość tensora jako liczbę float Pythona

Kod iteruje po mechanizmie ładowania danych oraz oblicza liczbę i ułamek poprawnych prognoz. Gdy pracujemy z dużymi zbiorami danych, ze względu na ograniczenia pamięci zazwyczaj możemy wywołać model tylko na niewielkiej części zbioru. Pokazana powyżej funkcja `compute_accuracy` jest ogólną metodą, która skaluje się do zbiorów danych o dowolnym rozmiarze, ponieważ w każdej iteracji fragment zbioru przekazywany do modelu ma taki sam rozmiar jak rozmiar partii widoczny podczas szkolenia. Wewnętrzne mechanizmy działania funkcji `compute_accuracy` przypominają te, z których korzystaliśmy wcześniej przy konwersji logitów na etykiety klas.

Następnie możemy zastosować tę funkcję do szkolenia:

```
print(compute_accuracy(model, train_loader))
```

Wynik jest następujący:

```
1.0
```

W podobny sposób można zastosować tę funkcję do zestawu testowego:

```
print(compute_accuracy(model, test_loader))
```

Uruchomienie tego kodu spowoduje wyświetlenie następującego wyniku:

```
1.0
```

A.8. Zapisywanie i wczytywanie modeli

Po przeszkołeniu modelu warto go zapisać tak, aby można go było później ponownie wykorzystać. Oto zalecany sposób, w jaki można zapisywać i ładować modele w PyTorch:

```
torch.save(model.state_dict(), "model.pth")
```

Atrybut `state_dict` modelu jest obiektem słownika Pythona, który mapuje każdą warstwę w modelu na jej trenowalne parametry (wagi i odchylenia). `"model.pth"` to dowolna nazwa pliku modelu zapisywanego na dysku. Można wykorzystać dowolną nazwę pliku i rozszerzenie, jednak najczęściej stosowanymi są `.pth` i `.pt`.

Po zapisaniu modelu można go przywrócić z dysku:

```
model = NeuralNetwork(2, 2)
model.load_state_dict(torch.load("model.pth"))
```

Funkcja `torch.load("model.pth")` odczytuje plik `model.pth` i rekonstruuje obiekt słownika Pythona zawierający parametry modelu, podczas gdy funkcja `model.load_state_dict()` stosuje te parametry do modelu, co powoduje przywrócenie jego wyuczonego stanu z chwili, w której model został zapisany.

Wiersz `model = NeuralNetwork(2, 2)` nie jest bezwzględnie konieczny, jeśli ten kod zostanie wykonany w tej samej sesji, w której zapisano model. Umieściłem go tutaj jednak, ponieważ chciałem pokazać, że aby model mógł zastosować zapisane parametry, egzemplarz modelu musi być zapisany w pamięci. W tym przypadku trzeba dokładnie dopasować architekturę `NeuralNetwork(2, 2)` do zapisanego modelu.

A.9. Optymalizacja wydajności szkolenia z użyciem układów GPU

Przyjrzyjmy się teraz, jak wykorzystać układy GPU, które w porównaniu ze zwykłymi układami CPU przyspieszają uczenie głębokich sieci neuronowych. Najpierw przyjrzymy się głównym pojęciom dotyczącym obliczeń na GPU we frameworku PyTorch. Następnie przeszkołimy model na pojedynczym układzie GPU. Na koniec przyjrzymy się rozproszonemu szkoleniu z użyciem wielu układów GPU.

A.9.1. Obliczenia PyTorch na urządzeniach GPU

Modyfikacja pętli szkolenia w celu opcjonalnego uruchomienia na układach GPU jest stosunkowo prosta i wymaga jedynie zmiany trzech wierszy kodu (patrz podrozdział A.7). Zanim dokonamy modyfikacji, warto przyjrzeć się głównym pojęciom związanym z obliczeniami na układach GPU z użyciem frameworka PyTorch. Układ obliczeniowy (ang. *device*) w PyTorch to procesor, który wykonuje obliczenia na danych. Przykładami układów obliczeniowych są CPU i GPU. W układzie obliczeniowym jest zapisany tensor frameworka PyTorch. W tym samym układzie są wykonywane działania na tym tensorze.

Zobaczmy, jak to działa w praktyce. Jeśli zainstalowałeś wersję frameworka PyTorch zgodną z GPU (patrz podrozdział A.1.3), to możesz za jego pomocą sprawdzić, czy Twój środowisko wykonawcze rzeczywiście obsługuje obliczenia na GPU. Aby to zrobić, możesz skorzystać z następującego kodu:

```
print(torch.cuda.is_available())
```

Jeśli w Twoim środowisku jest dostępny układ GPU, powinieneś uzyskać następujący wynik:

```
True
```

Załóżmy teraz, że mamy dwa tensory, które można do siebie dodać. Te obliczenie domyślnie będą wykonywane na urządzeniu CPU:

```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)
```

Wykonanie tego kodu spowoduje wyświetlenie następującego wyniku:

```
tensor([5., 7., 9.])
```

Teraz możesz użyć metody `.to()`. Ta metoda jest taka sama jak ta, którą zastosowaliśmy do zmiany typu danych tensora (patrz podpunkt 2.2.2), aby przenieść tensory na GPU i tam wykonać dodawanie:

```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)
```

Oto uzyskane wyniki:

```
tensor([5., 7., 9.], device='cuda:0')
```

Wynikowy tensor zawiera teraz informacje o urządzeniu, `device='cuda:0'`, co oznacza, że tensory znajdują się na pierwszym układzie GPU. Jeśli Twój komputer obsługuje wiele układów GPU, możesz wskazać układ GPU, do którego mają być przeniesione tensory. W tym celu należy w poleceniu transferu podać identyfikator urządzenia. Na przykład możesz skorzystać z wywołania `.to("cuda:0")`, `.to("cuda:1")` itd.

Wszystkie tensorы muszą jednak znajdować się na tym samym urządzeniu. W przeciwnym razie, gdy jeden tensor znajduje się na urządzeniu CPU, a drugi na GPU, obliczenia zakończą się niepowodzeniem:

```
tensor_1 = tensor_1.to("cpu")
print(tensor_1 + tensor_2)
```

Oto wynik działania tego kodu:

```
RuntimeError      Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but found at least two
→ devices, cuda:0 and cpu!
```

Podsumowując, powinieneś zadbać o przeniesienie tensorów na ten sam układ GPU. Kiedy to zrobisz, PyTorch zajmie się resztą.

A.9.2. Szkolenie na jednym układzie GPU

Gdy już wiesz, jak przenosić tensorы на układy GPU, możesz zmodyfikować pętlę szkoleniową w taki sposób, aby działała na GPU. Ten krok wymaga zmiany tylko trzech wierszy kodu tak, jak na poniższym listingu (listing A.11).

Listing A.11. Pętla szkoleniowa na GPU

```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
device = torch.device("cuda")           ← Definicja zmiennej device,
model = model.to(device)               ← Przeniesienie modelu do układu GPU

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):
        features, labels = features.to(device), labels.to(device)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Funkcja straty

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    ### LOGOWANIE
    print(f"Epoka: {epoch+1:03d}/{num_epochs:03d}"
          f" | Partia {batch_idx:03d}/{len(train_loader):03d}"
          f" | Strata Szkolenie/Walidacja: {loss:.2f}")
model.eval()                            ← Przeniesienie danych
# Wstaw opcjonalny kod oceny modelu
```

Uruchomienie powyższego kodu zwróci wynik podobny do wyników uzyskanych na układzie CPU (podrozdział A.7):

```
Epoka: 001/003 | Partia 000/002 | Strata Szkolenie/Walidacja: 0.75
Epoka: 001/003 | Partia 001/002 | Strata Szkolenie/Walidacja: 0.65
Epoka: 002/003 | Partia 000/002 | Strata Szkolenie/Walidacja: 0.44
Epoka: 002/003 | Partia 001/002 | Strata Szkolenie/Walidacja: 0.13
Epoka: 003/003 | Partia 000/002 | Strata Szkolenie/Walidacja: 0.03
Epoka: 003/003 | Partia 001/002 | Strata Szkolenie/Walidacja: 0.00
```

Zamiast wywołania `device = torch.device("cuda")` możesz użyć atrybutu `.to("cuda")`. Przeniesienie tensora do układu "cuda" zamiast `torch.device("cuda")` działa tak samo dobrze, a instrukcja jest krótsza (zobacz podrozdział A.9.1). Możesz również zmodyfikować instrukcję, co sprawi, że jeśli układ GPU nie będzie dostępny, ten sam kod będzie mógł być wykonywany na układzie CPU. Uważa się to za najlepszą praktykę podczas udostępniania kodu frameworka PyTorch:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Po zmodyfikowaniu tej przykładowej pętli szkoleniowej prawdopodobnie, ze względu na koszt transferu pamięci z CPU do GPU, nie zaobserwujemy przyspieszenia. Możemy jednak oczekiwać znacznego przyspieszenia w trakcie szkolenia głębokich sieci neuronowych, zwłaszcza modeli LLM.

PyTorch na macOS

Jeśli zamiast komputera z procesorem graficznym firmy Nvidia posługujesz się komputerem Apple Mac z układem Apple Silicon (takim jak M1, M2, M3 lub nowszy model), to aby skorzystać z układu `mps`, możesz zamienić instrukcję:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
na:
device = torch.device(
    "mps" if torch.backends.mps.is_available() else "cpu"
)
```

Ćwiczenie A.4

Porównaj czas mnożenia macierzy na układzie CPU i GPU. Przy jakim rozmiarze macierzy mnożenie macierzy na układzie GPU zaczyna być szybsze niż na układzie CPU? Wskazówka: aby porównać czasy działania kodu w środowisku Jupyter, użyj polecenia `%timeit`. Na przykład, jeśli korzystasz z macierzy `a` i `b`, uruchom w nowej komórce notatnika polecenie `%timeit a @ b`.

A.9.3. Szkolenie z użyciem wielu układów GPU

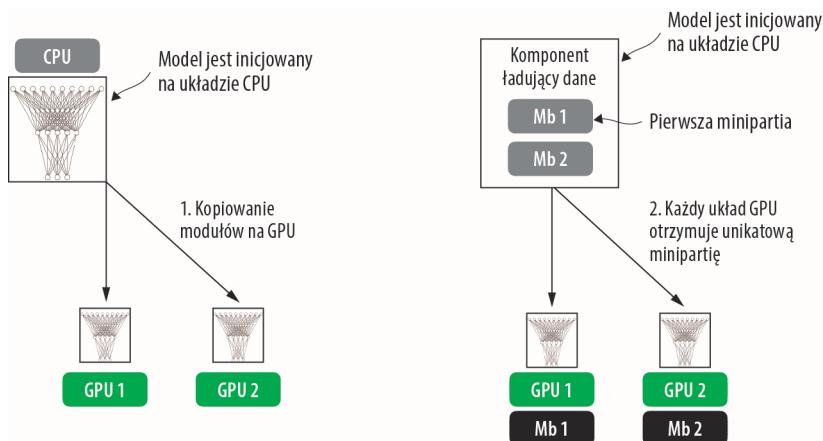
Szkolenie rozproszone to pojęcie polegające na rozdzieleniu szkolenia modelu na wiele układów GPU i maszyn. Dlaczego trzeba to robić? Chociaż przeszkolenie modelu na pojedynczym układzie GPU lub na jednej maszynie jest możliwe, proces ten może być niezwykle czasochłonny. Czas szkolenia można znacznie skrócić przez rozłożenie

procesu szkolenia na wiele maszyn, z których każda może być wyposażona w wiele układów GPU. Jest to szczególnie ważne na eksperymentalnych etapach rozwoju modelu, gdy w celu dostrojenia parametrów i architektury modelu może być potrzebnych wiele iteracji szkoleniowych.

UWAGA Do śledzenia przykładów w tej książce nie jest potrzebny dostęp do wielu układów GPU. Ten podrozdział jest przeznaczony dla Czytelników, którzy są zainteresowani sposobem wykonywania obliczeń z użyciem framework'a PyTorch na wielu układach GPU.

Zacznijmy od najbardziej podstawowego przypadku szkolenia rozproszonego w PyTorch: strategii `DistributedDataParallel` (DDP). Strategia DDP umożliwia uzyskanie współbieżności dzięki podziałowi danych wejściowych na dostępne urządzenia i jednocześnie przetwarzanie tych podzbiorów danych.

Jak to działa? PyTorch uruchamia na każdym układzie GPU oddzielny proces. Każdy proces otrzymuje osobną kopię modelu; podczas szkolenia te kopie są synchronizowane. Aby to zilustrować, założymy, że mamy dwa układy GPU, które chcemy wykorzystać do szkolenia sieci neuronowej, tak jak pokazałem na rysunku A.12.

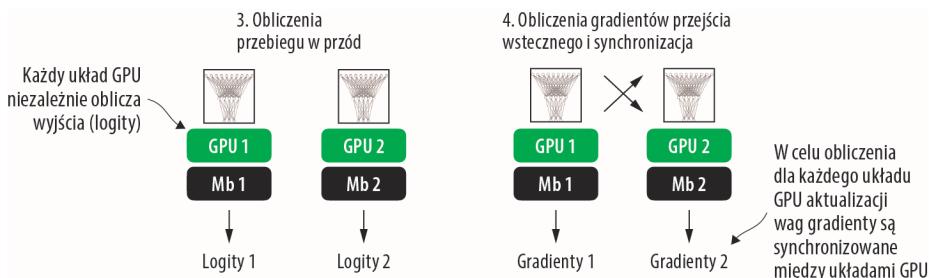


Rysunek A.12. Szkolenie modelu i transfer danych zgodne ze strategią DDP obejmują dwa główne kroki. Najpierw na każdym z układów GPU tworzymy kopię modelu. Następnie dzielimy dane wejściowe na unikatowe minipartie, które przekazujemy do każdej kopii modelu

Każdy z dwóch układów GPU otrzyma swoją kopię modelu. Następnie, w każdej iteracji szkolenia, każdy model otrzyma z mechanizmu ładującego dane minipartię (lub po prostu „partię”). Aby zyskać pewność, że w trakcie korzystania z DDP każdy układ GPU otrzyma inną partię, nienakładającą się na pozostałe, można użyć obiektu `DistributedSampler`.

Ponieważ każda kopia modelu widzi inną próbke danych szkoleniowych, kopie modelu podczas przebiegu wstecz zwierają różne logity i obliczają różne gradienty. Te gradienty są następnie uśredniane i synchronizowane w czasie szkolenia w celu

aktualizacji modeli. W ten sposób zyskujemy pewność, że modele nie staną się rozbieżne (rysunek A.13).



Rysunek A.13. Przebiegi w przód i wstecz w strategii DDP są wykonywane niezależnie na każdym GPU dla przydzielonego do niego podzbioru danych. Po zakończeniu przejęć w przód i wstecz gradienty z każdej repliki modelu (na każdym GPU) są synchronizowane na wszystkich GPU. Dzięki temu każda replika modelu dysponuje takimi samymi zaktualizowanymi wagami

Zaletą korzystania z DDP jest większa szybkość przetwarzania zbioru danych w porównaniu z przetwarzaniem go na pojedynczym układzie GPU. Jeśli pominać niewielki narzut komunikacyjny między urządzeniami, który pojawi się przy korzystaniu z DDP, dzięki użyciu dwóch układów GPU teoretycznie można przetworzyć epokę szkoleniową w czasie o połowę krótszym niż z wykorzystaniem tylko jednego układu GPU. Wydajność czasowa skaluje się wraz z liczbą GPU, zatem gdy dysponujemy ośmioma takimi układami, możemy przetworzyć epokę osiem razy szybciej — i tak dalej.

UWAGA DDP nie działa poprawnie w interaktywnych środowiskach Pythona, takich jak notatniki Jupyter, które nie obsługują wieloprocesowości w taki sam sposób jak samodzielny skrypt Pythona. Z tego powodu poniższy kod powinien być wykonywany jako skrypt na pojedynczej maszynie, a nie w interfejsie notatnika, takim jak Jupyter. DDP musi inicjować wiele procesów, z których każdy powinien dysponować własnym egzemplarzem interpretera Pythona.

Zobaczmy teraz, jak DDP działa w praktyce. Dla zachowania zwięzości skupię się na podstawowych częściach kodu, które trzeba dostosować na potrzeby szkolenia w trybie DDP. Czytelnicy, którzy chcą uruchomić kod na własnej maszynie z wieloma układami GPU lub na wybranym egzemplarzu w chmurze, powinni skorzystać z samodzielnego skryptu udostępnionego w repozytorium GitHub tej książki, pod adresem <https://github.com/rasbt/LLMs-from-scratch>.

Najpierw trzeba zimportować kilka podmodułów, klas i funkcji w celu obsługi rozproszonego szkolenia z użyciem frameworka PyTorch, tak jak pokazano na listingu A.12.

Zanim zagłębię się w szczegóły zmian mających na celu zapewnienie kompatybilności szkolenia z techniką DDP, omówię pokrótce uzasadnienie zastosowania tych nowo zimportowanych narzędzi razem z klasą `DistributedDataParallel`.

Listing A.12. Narzędzia frameworka PyTorch do szkolenia rozproszonego

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
```

Podmoduł `multiprocessing` frameworka PyTorch zawiera takie funkcje jak `multiprocessing.spawn`, której użyjemy do zainicjowania wielu procesów i zastosowania funkcji do równoległego przetwarzania wielu wejść. Zastosujemy ją do uruchomienia jednego procesu szkoleniowego na układzie GPU. Gdy uruchomisz na potrzeby szkolenia wiele procesów szkoleniowych, będziesz zmuszony znaleźć sposób na podzielenie zbioru danych pomiędzy te różne procesy. Do tego wykorzystamy klasę `DistributedSampler`.

Do inicjowania i zamknięcia rozproszonych modów szkoleniowych są używane metody, odpowiednio, `init_process_group` i `destroy_process_group`. Funkcja `init_process_group` powinna być wywołana na początku skryptu szkoleniowego, aby zainicjować grupę procesów dla każdego procesu w rozproszonej konfiguracji, natomiast metoda `destroy_process_group` powinna być wywołana na końcu skryptu szkoleniowego, aby zniszczyć daną grupę procesów i zwolnić wykorzystywane zasoby. Sposób wykorzystania tych nowych komponentów do implementacji szkolenia w trybie DDP dla zaimplementowanego wcześniej modelu sieci neuronowej (Neural Network) jest zilustrowany na listingu A.13.

Listing A.13. Szkolenie modelu z użyciem strategii DistributedDataParallel

```
def ddp_setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"           ← Adres głównego węzła
    os.environ["MASTER_PORT"] = "12345"                ← Dowolny wolny port na komputerze
    world_size to liczba
    wykorzystywanych
    układów GPU
    init_process_group(                                nccl to akronim pochodzący
                                                       od NVIDIA Collective Communication Library
                                                       rank=rank,                                rank odnosi się do indeksu układu GPU, którego chcemy użyć
                                                       world_size=world_size)
    torch.cuda.set_device(rank)                         ← Ustawienie bieżącego układu GPU, na którym
                                                       będą alokowane tensory i wykonywane operacje

def prepare_dataset():
    # wstaw kod odpowiedzialny za przygotowanie zbioru danych
    train_loader = DataLoader(
        Obiekt
        Distributed
        Sampler jest
        odpowiedzialny
        za tasowanie
        dataset=train_ds,
        batch_size=2,
        shuffle=False,
        pin_memory=True,
        drop_last=True,
        sampler=DistributedSampler(train_ds)          ← Włącza szybszy transfer pamięci
                                                       podczas szkolenia na układzie GPU
    )
    return train_loader, test_loader                  ← Podzielenie zbioru danych na odrębne,
                                                       nienakładające się podzbiory dla każdego
                                                       procesu (układu GPU)

def main(rank, world_size, num_epochs):
    ddp_setup(rank, world_size)                      ← Główna funkcja uruchamiająca
    train_loader, test_loader = prepare_dataset()     ← szkolenie modelu
```

```

model = NeuralNetwork(num_inputs=2, num_outputs=2)
model.to(rank)
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
model = DDP(model, device_ids=[rank])
for epoch in range(num_epochs):
    for features, labels in train_loader:
        features, labels = features.to(rank), labels.to(rank)
        # wstaw kod odpowiedzialny za prognozowanie i propagację wsteczną
        print(f"[GPU{rank}] Epoka: {epoch+1:03d}/{num_epochs:03d}"
              f" | Rozmiar partii {labels.shape[0]:03d}"
              f" | Strata Szkolenie/Walidacja: {loss:.2f}")

    model.eval()
    train_acc = compute_accuracy(model, train_loader, device=rank)
    print(f"[GPU{rank}] Dokładność zbioru szkoleniowego", train_acc)
    test_acc = compute_accuracy(model, test_loader, device=rank)
    print(f"[GPU{rank}] Dokładność zbioru testowego", test_acc)
    destroy_process_group()                                     ← Wyczyszczenie alokacji zasobów

if __name__ == "__main__":
    print("Liczba dostępnych układów GPU:", torch.cuda.device_count())
    torch.manual_seed(123)                                     Uruchomienie głównej funkcji z użyciem wielu procesów,
    num_epochs = 3                                            gdzie nprocs=world_size oznacza jeden proces na GPU
    world_size = torch.cuda.device_count()                   ←
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_size)

```

Zanim uruchomisz ten kod, podsumujmy, jak on działa, zgodnie z adnotacjami zamieszczonymi na listingu. Na końcu listingu jest instrukcja `if __name__ == '__main__'`, zawierająca kod wykonywany w przypadku uruchamiania kodu jako skryptu Pythona zamiast importowania go jako moduł. Ten kod najpierw wyświetla liczbę dostępnych układów GPU za pomocą instrukcji `torch.cuda.device_count()`, ustawiąc ziarno losowości w celu zapewnienia powtarzalności wyników, a następnie, za pomocą funkcji frameworka PyTorch `multiprocessing.spawn`, uruchamia nowe procesy. W tym przypadku funkcja `spawn`, do której przekazujemy ustawienie `nproces=world_size`, gdzie `world_size` oznacza liczbę dostępnych układów GPU, uruchamia jeden proces na GPU. Funkcja `spawn` uruchamia kod w funkcji `main`, którą definiujemy w tym samym skrypcie, z kilkoma dodatkowymi argumentami przekazanymi za pośrednictwem listy argumentów `args`. Warto zwrócić uwagę, że funkcja `main` ma argument `rank`, którego nie uwzględniamy w wywołaniu `mp.spawn()`. Dzieje się tak dlatego, że parametr `rank`, który odnosi się do identyfikatora procesu wykorzystywanego w roli identyfikatora GPU, jest przekazywany automatycznie.

Funkcja `main` konfiguruje środowisko rozproszone za pomocą innej zdefiniowanej wcześniej funkcji — `ddp_setup`, która ładuje zbiory szkoleniowy i testowy, konfiguruje model i przeprowadza szkolenie. W przeciwieństwie do szkolenia z jednym układem GPU (podrozdział A.9.2), teraz przesyłamy model i dane do urządzenia docelowego za pomocą metody `.to(rank)`, której używamy w celu odwołania się do identyfikatora układu GPU. Ponadto opakowujemy model za pomocą klasy `DDP`, która podczas szkolenia umożliwia synchronizację gradientów między różnymi układami GPU. Po

zakończeniu szkolenia i dokonaniu oceny modeli używamy funkcji `destroy_process_group()`, której zadanie polega na czystym zakończeniu szkolenia rozprozonego i zwolnieniu przydzielonych zasobów.

Wcześniej wspomniałem, że każdy układ GPU otrzymuje inną próbkę danych szkoleniowych. Aby to zapewnić, w komponencie ładującym dane skorzystaliśmy z ustawienia `sampler=DistributedSampler(train_ds)`.

Ostatnią funkcją do omówienia jest `ddp_setup`. Funkcja ta ustawia adres głównego węzła i port potrzebne do komunikacji między różnymi procesami, inicjuje grupę procesów za pomocą backendu NCCL (zaprojektowanego do komunikacji GPU-GPU) oraz ustawia parametry `rank` (identyfikator procesu) i `world_size` (całkowita liczba procesów). Na koniec określa urządzenie GPU odpowiadające bieżącej randze procesu szkolenia modelu.

WYBIERANIE DOSTĘPNYCH UKŁADÓW GPU NA KOMPUTERZE Z WIELOMA UKŁADAMI GPU

Najprostszym sposobem ograniczenia liczby układów GPU używanych do szkolenia na komputerze z wieloma układami GPU jest użycie zmiennej środowiskowej `CUDA_VISIBLE_DEVICES`. Aby to zilustrować, założymy, że Twój komputer ma wiele układów GPU, a Ty chcesz użyć tylko jednego — na przykład układu GPU o indeksie 0. Zamiast polecenia `python twój_skrypt.py` możesz uruchomić z terminala następujące polecenie:

```
CUDA_VISIBLE_DEVICES=0 python twój_skrypt.py
```

Jeśli Twój komputer jest wyposażony w cztery układy GPU, a chcesz używać tylko pierwszego i trzeciego, możesz użyć polecenia:

```
CUDA_VISIBLE_DEVICES=0,2 python twój_skrypt.py
```

Ustawienie zmiennej `CUDA_VISIBLE_DEVICES` w ten sposób jest prostą i skutecną metodą zarządzania alokacją GPU bez modyfikowania skryptów PyTorch.

Spróbujmy teraz uruchomić zamieszczony powyżej kod, aby zobaczyć, jak działa w praktyce. W tym celu uruchom w terminalu następujące polecenie:

```
python ch02-DDP-script.py
```

Należy pamiętać, że powinno to działać zarówno na komputerach z jednym, jak i wieloma układami GPU. Jeśli uruchomisz ten kod na pojedynczym układzie GPU, powinieneś zobaczyć następujące wyniki:

```
Wersja PyTorch: 2.2.1+cu117
CUDA dostępny: True
Liczba dostępnych układów GPU: 1
[GPU0] Epoka: 001/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.62
[GPU0] Epoka: 001/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.32
[GPU0] Epoka: 002/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.11
[GPU0] Epoka: 002/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.07
[GPU0] Epoka: 003/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.02
[GPU0] Epoka: 003/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.03
[GPU0] Dokładność zbioru szkoleniowego 1.0
[GPU0] Dokładność zbioru testowego 1.0
```

Wynik działania kodu wygląda podobnie do wyniku uzyskanego na pojedynczym układzie GPU (podrozdział A.9.2), co jest dobrym sprawdzianem poprawności.

Jeśli uruchomisz to samo polecenie i kod na komputerze z dwoma układami GPU, powinieneś uzyskać następujące wyniki:

```
Wersja PyTorch: 2.2.1+cu117
CUDA dostępny: True
Liczba dostępnych układów GPU: 2
[GPU1] Epoka: 001/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.60
[GPU0] Epoka: 001/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.59
[GPU0] Epoka: 002/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.16
[GPU1] Epoka: 002/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.17
[GPU0] Epoka: 003/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.05
[GPU1] Epoka: 003/003 | Rozmiar partii 002 | Strata Szkolenie/Walidacja: 0.05
[GPU1] Dokładność zbioru szkoleniowego 1.0
[GPU0] Dokładność zbioru szkoleniowego 1.0
[GPU1] Dokładność zbioru testowego 1.0
[GPU0] Dokładność zbioru testowego 1.0
```

Tak jak można się było spodziewać, niektóre partie są przetwarzane na pierwszym GPU (GPU0), a inne na drugim (GPU1). Jednak w wynikach podczas wyświetlania dokładności szkolenia i testu widać zdublowane wiersze. Każdy proces (mówiąc inaczej – każdy układ GPU) wyświetla dokładność testu niezależnie. Ponieważ technika DDP polega na replikacji modelu na każdym układzie GPU, a każdy proces działa niezależnie, to jeśli w pętli testowej znajduje się instrukcja `print`, zostanie uruchomiona przez każdy z procesów, co doprowadzi do powtarzających się wierszy wyjściowych. Jeśli Ci to przeszkadza, możesz użyć do zarządzania instrukcją `print` rangi każdego procesu:

```
if rank == 0:                                     ← Wyświetlanie tylko w pierwszym procesie
    print("Dokładność zbioru testowego: ", accuracy)
```

Tak w skrócie działa rozproszone szkolenie z użyciem DDP. Jeśli jesteś zainteresowany dodatkowymi szczegółami, zachęcam Cię, żebyś się zapoznał z oficjalną dokumentacją API, dostępną na stronie <https://mng.bz/9dPr>.

Zamienniki dla interfejsów API frameworka PyTorch na potrzeby szkolenia na wielu układach GPU

Jeśli wolisz prostszy sposób korzystania z wielu układów GPU w PyTorch, możesz rozważyć zastosowanie dodatkowych interfejsów API, takich jak biblioteka open source Fabric. Pisatem o tym w artykule *Accelerating PyTorch Model Training: Using Mixed-Precision and Fully Sharded Data Parallelism* (<https://mng.bz/jXle>).

Podsumowanie

- PyTorch to biblioteka typu open source z trzema podstawowymi komponentami: biblioteką tensorów, funkcjami automatycznego różniczkowania i narzędziami do uczenia głębokiego.

- Biblioteka frameworka PyTorch przeznaczona do obsługi tensorów jest podobna do bibliotek opartych na macierzach, takich jak NumPy.
- W kontekście frameworka PyTorch tensory są strukturami danych przypominającymi tablice, reprezentującymi skalary, wektory, macierze i tablice wyższych wymiarów.
- Tensory frameworka PyTorch można przetwarzać na układach CPU, ale jedną z głównych zalet formatu tensorowego we frameworku PyTorch jest obsługa GPU w celu przyspieszenia obliczeń.
- Funkcje automatycznego różniczkowania (autograd) w PyTorch pozwalają na wygodne szkolenie sieci neuronowych z użyciem propagacji wstecznej, bez ręcznego wyprowadzania gradientów.
- Narzędzia do uczenia głębokiego w PyTorch dostarczają bloków konstrukcyjnych do tworzenia niestandardowych głębokich sieci neuronowych.
- PyTorch zawiera klasy Dataset i DataLoader, umożliwiające konfigurowanie wydajnych potoków ładowania danych.
- Modele LLM najłatwiej jest szkolić na układzie CPU lub na pojedynczym układzie GPU.
- We frameworku PyTorch najprostszym sposobem na przyspieszenie szkolenia w przypadku, gdy jest dostępnych wiele układów GPU, jest skorzystanie z techniki DistributedDataParallel.

Dodatek B

Bibliografia

i lektura uzupełniająca

Rozdział 1.

Zespół firmy Bloomberg udowodnił na przykładzie wersji modelu GPT wstępnie przeszkolonego od podstaw na danych finansowych, że samodzielnie zbudowane modele LLM mogą przewyższać modele LLM ogólnego przeznaczenia. Niestandardowy model LLM przewyższył ChatGPT w zadaniach finansowych, a jednocześnie zachował dobrą wydajność w ogólnych testach porównawczych modeli LLM:

- Wu i współpracownicy (2023), *BloombergGPT: A Large Language Model for Finance*, <https://arxiv.org/abs/2303.17564>.

Istniejące modele LLM można dostosować i dostroić tak, aby przewyższały uniwersalne modele LLM. Udowodniły to zespoły z Google Research i Google DeepMind w kontekście medycznym:

- Singhal i współpracownicy (2023), *Towards Expert-Level Medical Question Answering with Large Language Models*, <https://arxiv.org/abs/2305.09617>.

Oryginalną architekturę transformera zaprezentowano w poniższym artykule:

- Vaswani i współpracownicy (2017), *Attention Is All You Need*, <https://arxiv.org/abs/1706.03762>.

Więcej informacji na temat transformera w stylu kodera o nazwie BERT można znaleźć w artykule:

- Devlin i współpracownicy (2018), *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, <https://arxiv.org/abs/1810.04805>.

Artykuł opisujący model GPT-3 w stylu dekodera, który stał się inspiracją dla nowoczesnych modeli LLM i który wykorzystano w roli szablonu do implementacji modelu LLM od podstaw w tej książce, to:

- Brown i współpracownicy (2020), *Language Models are Few-Shot Learner*, <https://arxiv.org/abs/2005.14165>.

Artykuł, w którym opisano oryginalny transformer wizyjny do klasyfikacji obrazów. Pokazano w nim, że architektury transformerów nie ograniczają się tylko do danych tekstowych:

- Dosovitskiy i współpracownicy (2020), *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, <https://arxiv.org/abs/2010.11929>.

Poniższe eksperymentalne (ale mniej popularne) architektury LLM mogą posłużyć jako dowody na to, że nie wszystkie modele LLM muszą być oparte na architekturze transformera:

- Peng i współpracownicy (2023), *RWKV: Reinventing RNNs for the Transformer Era*, <https://arxiv.org/abs/2305.13048>.
- Poli i współpracownicy (2023), *Hyena Hierarchy: Towards Larger Convolutional Language Models*, <https://arxiv.org/abs/2302.10866>.
- Gu i Dao (2023), *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*, <https://arxiv.org/abs/2312.00752>.

Model Meta AI jest popularną implementacją modelu podobnego do GPT, która w przeciwieństwie do modeli GPT-3 i ChatGPT jest darmowa:

- Touvron i współpracownicy (2023), *Llama 2: Open Foundation and Fine-Tuned Chat Models*, <https://arxiv.org/abs/2307.092881>.

Czytelników zainteresowanych dodatkowymi informacjami o zbiorach danych przedstawionych w podrozdziale 1.5 zachęcam do zapoznania się z dokumentem opisującym publicznie dostępny zbiór danych *The Pile*, utrzymywany przez Eleutheria AI:

- Gao i współpracownicy (2020), *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*, <https://arxiv.org/abs/2101.00027>.

Poniższy artykuł zawiera opis przeznaczonego do dostrajania GPT-3 InstructGPT, o którym wspominałem w podrozdziale 1.6 oraz który szczegółowo omówilem w rozdziale 7.:

- Ouyang i współpracownicy (2022), *Training Language Models to Follow Instructions with Human Feedback*, <https://arxiv.org/abs/2203.02155>.

Rozdział 2.

Czytelnicy zainteresowani opisem i porównaniem przestrzeni osadzania z przestrzeniami ukrytymi oraz ogólnym pojęciem reprezentacji wektorowych znajdą więcej informacji w pierwszym rozdziale mojej książki:

- Sebastian Raschka (2023), *Machine Learning Q and AI*, <https://leanpub.com/machine-learning-q-and-ai>.

Sposób kodowania par bajtów jako metody tokenizacji szczegółowo opisano w poniższym artykule:

- Sennrich i współpracownicy (2015), *Neural Machine Translation of Rare Words with Subword Units*, <https://arxiv.org/abs/1508.07909>.

Kod tokenizera opartego na kodowaniu par bajtów wykorzystywanego do szkolenia modelu GPT-2 udostępniła firma OpenAI:

- <https://github.com/openai/gpt-2/blob/master/src/encoder.py>.

OpenAI udostępnia interaktywny interfejs użytkownika pozwalający zilustrować działanie tokenizera par bajtów w modelach GPT:

- <https://platform.openai.com/tokenizer>.

Czytelnicy zainteresowani kodowaniem i szkoleniem tokenizera BPE od podstaw znajdą minimalną i czytelną implementację w repozytorium `minbpe` Andreja Karpathy'ego w serwisie GitHub:

- *A Minimal Implementation of a BPE Tokenizer*, <https://github.com/karpathy/minbpe>.

Czytelnicy zainteresowani zbadaniem alternatywnych schematów tokenizacji, używanych w innych popularnych modelach LLM, mogą znaleźć więcej informacji w dokumentach opisujących tokenizacje SentencePiece i WordPiece:

- Kudo i Richardson (2018), *SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing*, <https://aclanthology.org/D18-2012/>.
- Song i współpracownicy (2020), *Fast WordPiece Tokenization*, <https://arxiv.org/abs/2012.15524>.

Rozdział 3.

Czytelnicy zainteresowani warstwą uwagi Bahdanau dla sieci RNN oraz wykorzystaniem jej do tłumaczeń znajdą dodatkowe informacje w poniższym artykule:

- Bahdanau, Cho i Bengio (2014), *Neural Machine Translation by Jointly Learning to Align and Translate*, <https://arxiv.org/abs/1409.0473>.

Pojęcie samouwagi jako skalowanego iloczynu skalarnego wprowadzono w dokumencie opisującym architekturę *Original Transformer*:

- Vaswani i współpracownicy (2017), *Attention Is All You Need*, <https://arxiv.org/abs/1706.03762>.

FlashAttention to wysoce wydajna implementacja mechanizmu samouwagi, w której, dzięki optymalizacji wzorców dostępu do pamięci, przyspieszono proces obliczeniowy. FlashAttention z perspektywy obliczeń matematycznych jest identyczny ze standardowym mechanizmem samouwagi, ale obejmuje zoptymalizowany proces obliczeniowy, co wpływa na wydajność korzystających z niego modeli:

- Dao i współpracownicy (2022), *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*, <https://arxiv.org/abs/2205.14135>.
- Dao (2023), *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*, <https://arxiv.org/abs/2307.08691>

Framework PyTorch implementuje funkcję samouwagi i uwagi przyczynowej, która w celu poprawy wydajności obsługuje wykorzystanie warstwy FlashAttention. Ta funkcja jest w wersji beta i może się zmienić:

- Dokumentacja mechanizmu scaled_dot_product_attention: <https://mng.bz/NRJd>.

Framework PyTorch implementuje również wydajną klasę MultiHeadAttention opartą na funkcji scaled_dot_product:

- Dokumentacja klasy MultiHeadAttention: <https://mng.bz/DdJV>.

Dropout to technika regularyzacji polegająca na losowym usuwaniu jednostek (wraz z ich połączeniami) z sieci neuronowej podczas uczenia, stosowana w sieciach neuronowych w celu zapobiegania nadmiernemu dopasowaniu:

- Srivastava i współpracownicy (2014), *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, <https://jmlr.org/papers/v15/srivastava14a.html>.

O ile korzystanie z wielogłowicowej uwagi opartej na skalowanej warstwie uwagi korzystającej z iloczynu skalarnego w praktyce pozostaje najczęstszym wariantem samouwagi, autorzy poniższego artykułu odkryli, że osiągnięcie dobrej wydajności jest możliwe również bez macierzy wag wartości i warstwy rzutowania:

- He i Hofmann (2023), *Simplifying Transformer Blocks*, <https://arxiv.org/abs/2311.01906>.

Rozdział 4.

Poniższy artykuł opisuje technikę, która stabilizuje dynamikę ukrytego stanu sieci neuronowych przez normalizację zsumowanych wejść do neuronów w warstwie ukrytej, co w porównaniu z wcześniej opublikowanymi metodami znacznie skraca czas szkolenia:

- Ba, Kiros i Hinton (2016), *Layer Normalization*, <https://arxiv.org/abs/1607.06450>.

Mechanizm Post-LayerNorm, używany w modelu *Original Transformer* stosuje normalizację warstwową po warstwie samouwagi i sieci ze sprzężeniem w przód. W przeciwieństwie do tego schematu, mechanizm Pre-LayerNorm, stosowany w takich modelach jak GPT-2 i nowszych modelach LLM, stosuje normalizację warstwową przed tymi komponentami. Pozwala to uzyskać bardziej stabilną dynamikę szkolenia i jak wykazano, w niektórych przypadkach poprawia wydajność. Tematykę tę poruszono w następujących artykułach:

- Xiong i współpracownicy (2020), *On Layer Normalization in Transformer Architecture*, <https://arxiv.org/abs/2002.04745>.
- Tie i współpracownicy (2023), *ResiDual: Transformer with Dual Residual Connections*, <https://arxiv.org/abs/2304.14802>.

Popularnym wariantem normalizacji LayerNorm stosowanym w nowoczesnych modelach LLM, charakteryzującym się lepszą wydajnością obliczeniową, jest RMSNorm. Zastosowanie mechanizmu w tym wariantie upraszcza proces normalizacji, który polega na zastosowaniu w tym celu wyłącznie pierwiastka ze średniej kwadratowej wejść, bez odejmowania średniej przed podniesieniem do kwadratu. Oznacza to, że przed obliczeniem skali nie trzeba obliczać średniej. Normalizację RMSNorm opisano szczegółowo w artykule:

- Zhang and Sennrich (2019), *Root Mean Square Layer Normalization*, <https://arxiv.org/abs/1910.07467>.

Funkcja aktywacji GELU (ang. *Gaussian Error Linear Unit*) łączy w sobie właściwości zarówno klasycznej funkcji aktywacji ReLU, jak i funkcji skumulowanego rozkładu normalnego, pozwalających modelować wyjścia warstwy, co w modelach uczenia głębokiego umożliwia stochastyczną regularyzację i nielinowość:

- Hendricks i Gimpel (2016), *Gaussian Error Linear Units (GELUs)*, <https://arxiv.org/abs/1606.08415>.

W dokumencie opisującym model GPT-2 wprowadzono szereg modeli LLM opartych na transformerach o różnych rozmiarach – 124 miliony, 355 milionów, 774 miliony i 1,5 miliarda parametrów:

- Radford i współpracownicy (2019), *Language Models Are Unsupervised Multitask Learner*, <https://mng.bz/lMgo>.

W modelu GPT-3 firmy OpenAI, ogólnie rzecz biorąc, zastosowano tę samą architekturę co w modelu GPT-2, z tym wyjątkiem, że największa wersja (175 miliardów) jest 100 razy większa niż największy model GPT-2 i została przeszkolona na znacznie większej ilości danych. Zainteresowanych Czytelników zachęcam do zapoznania się z oficjalnym dokumentem GPT-3 przygotowanym przez OpenAI oraz z przeglądem technicznym Lambda Labs, w którym obliczono, że szkolenie modelu GPT-3 na pojedynczym konsumenckim układzie GPU RTX 8000 zajęłoby 665 lat:

- Brown i współpracownicy (2020), *Language Models are Few-Shot Learner*, <https://arxiv.org/abs/2005.14165>.
- *OpenAI's GPT-3 Language Model: A Technical Overview*, <https://lambdalabs.com/blog/demystifying-gpt-3>.

NanoGPT to repozytorium kodu z minimalistyczną, ale wydajną implementacją modelu GPT-2, podobnego do modelu zaimplementowanego w tej książce. Kod zaprezentowany w tej książce różni się jednak od nanoGPT. Pomimo to jego repozytorium zainspirowało zreorganizowanie dużej implementacji klasy bazowej GPT w mniejsze podmoduły:

- NanoGPT, repozytorium do szkolenia modeli GPT średniej wielkości, <https://github.com/karpathy/nanoGPT>.

Opublikowany na blogu pouczający wpis, który pokazuje, że większość obliczeń w modelach LLM, gdy rozmiar kontekstu jest mniejszy niż 32 000 tokenów, jest wykonywana w warstwach sprzężenia zwrotnego, a nie w warstwach uwagi:

- Harm de Vries, *In the Long (Context) Run*, <https://www.harmdevries.com/post/context-length/>.

Rozdział 5.

Szczegółowe informacje na temat określania funkcji straty i stosowania przekształceń logarytmicznego w celu ułatwienia optymalizacji matematycznej można znaleźć w filmie z mojego wykładu:

- *L8.2 Logistic Regression Loss Function*, <https://www.youtube.com/watch?v=GxJeODZvydM>.

Poniższy wykład i przykład kodu zaprezentowany przez autora wyjaśniają, jak „pod maską” działają funkcje entropii krzyżowej framework PyTorch:

- *L8.7.1 OneHot Encoding and Multi-category Cross Entropy*, <https://www.youtube.com/watch?v=4n71-tZ94yk>.
- *Understanding Onehot Encoding and Cross Entropy in PyTorch*, <https://mng.bz/o05v>.

Poniższe dwa artykuły szczegółowo opisują zagadnienia związane ze zbiorami danych, hiperparametrami oraz szczegółami architektury wykorzystywanej do wstępniego szkolenia modeli LLM:

- Biderman i współpracownicy (2023), *Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling*, <https://arxiv.org/abs/2304.01373>.
- Groeneveld i współpracownicy (2024), *OLMo: Accelerating the Science of Language Models*, <https://arxiv.org/abs/2402.00838>.

Repozytorium kodu towarzyszące tej książce zawiera instrukcje przygotowania ponad 60 tysięcy książek z domeny publicznej dostępnych w ramach Projektu Gutenberg do szkolenia modelu LLM:

- *Pretraining GPT on the Project Gutenberg Dataset*, <https://mng.bz/Bdw2>.

Wstępne szkolenie modeli LLM omówiłem w rozdziale 5., a w „Dodatku D” omówię bardziej zaawansowane funkcje szkoleniowe, takie jak *linear warmup* (rozgrzewka liniowa) oraz wyżarzanie kosinusowe. W poniższym artykule stwierdzono, że podobne techniki można z powodzeniem stosować do kontynuowania wstępnego szkolenia przeszkolonych modeli LLM. Można w nim również znaleźć dodatkowe wskazówki i spostrzeżenia:

- Ibrahim i współpracownicy (2024), *Simple and Scalable Strategies to Continually Pre-train Large Language Models*, <https://arxiv.org/abs/2403.08763>.

BloombergGPT to przykład modelu LLM specyficznego dla domeny, stworzonego w wyniku szkolenia zarówno na ogólnych, jak i specyficznych dla domeny korpusach tekstowych, szczególnie z dziedziny finansów:

- Wu i współpracownicy (2023), *BloombergGPT: A Large Language Model for Finance*, <https://arxiv.org/abs/2303.17564>.

GaLore to najnowszy projekt badawczy, którego celem jest zwiększenie wydajności wstępniego szkolenia modeli LLM. Wymagana modyfikacja kodu sprowadza się do zastąpienia w funkcji szkoleniowej optymalizatora AdamW framework PyTorch dostarczonym w pakiecie `galore-torch` Pythona optymalizatorem `GaLoreAdamW`:

- Zhao i współpracownicy (2024), *GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection*, <https://arxiv.org/abs/2403.03507>.
- Repozytorium kodu GaLore, <https://github.com/jiaweizzhao/GaLore>.

W poniższych dokumentach i zasobach można znaleźć listę dostępnych za darmo wielkoskalowych zbiorów danych, o objętości od kilkuset gigabajtów do wielu terabajtów danych tekstowych, które można wykorzystać do wstępniego szkolenia modeli LLM:

- Soldaini i współpracownicy (2024), *Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research*, <https://arxiv.org/abs/2402.00159>.

- Gao i współpracownicy (2020), *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*, <https://arxiv.org/abs/2101.00027>.
- Penedo i współpracownicy (2023), *The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only*, <https://arxiv.org/abs/2306.01116>.
- Together AI, *RedPajama*, <https://mng.bz/d6nw>.
- Zbiór danych FineWeb, zawierający ponad 15 bilionów tokenów oczyszczonych i zdeduplikowanych angielskich danych z internetu, których źródłem jest serwis CommonCrawl, <https://mng.bz/rVzy>.

Artykuł, w którym po raz pierwszy opisano próbkowanie top-k, to:

- Fan i współpracownicy (2018), *Hierarchical Neural Story Generation*, <https://arxiv.org/abs/1805.04833>.

Alternatywą dla próbkowania top-k jest próbkowanie top-p (nie omówiłem go w rozdziale 5.), polegające na wyborze z najmniejszego zbioru najlepszych tokenów, których skumulowane prawdopodobieństwo przekracza próg p . Dla odróżnienia próbkowania top-k polega na wyborze z k najlepszych tokenów według prawdopodobieństwa:

- *Top-p sampling*, https://en.wikipedia.org/wiki/Top-p_sampling.

Wyszukiwanie wiązką — ang. *beam search* (nie omówiłem go w rozdziale 5.) to alternatywny algorytm dekodowania polegający na generowaniu sekwencji wyjściowych przez zachowanie na każdym etapie — aby zrównoważyć wydajność i jakość — tylko najwyższej punktowanych sekwencji częściowych:

- Vijayakumar i współpracownicy (2016), *Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models*, <https://arxiv.org/abs/1610.02424>.

Rozdział 6.

Oto lista dodatkowych zasobów, w których opisano różne rodzaje dostrajania:

- *Using and Finetuning Pretrained Transformers*, <https://mng.bz/VxJG>.
- *Finetuning Large Language Models*, <https://mng.bz/x28X>.

Opis dodatkowych eksperymentów, w tym porównanie dostrajania pierwszego tokena wyjściowego z ostatnim tokenem wyjściowym, można znaleźć w materiałach uzupełniających w serwisie GitHub:

- *Additional spam classification experiments*, <https://mng.bz/AdJx>.

W zadaniu klasyfikacji binarnej, takim jak klasyfikacja spamu, z technicznego punktu widzenia możliwe jest użycie tylko jednego węzła wyjściowego zamiast dwóch węzłów wyjściowych, co omawiam w poniższym artykule:

- *Losses Learned—Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch*, <https://mng.bz/ZEJA>.

Opis dodatkowych eksperymentów dotyczących dostrajania różnych warstw modeli LLM można znaleźć w poniższym artykule, w którym pokazałem, że dostrojenie oprócz warstwy wyjściowej także ostatniego bloku transformera znacznie poprawia wydajność predykcyjną:

- *Finetuning Large Language Models*, <https://mng.bz/RZJv>.

Czytelnicy mogą znaleźć dodatkowe zasoby i informacje dotyczące postępowania z niezrównoważonymi zbiorami danych klasyfikacji w dokumentacji *imbalanced-learn*:

- *Imbalanced-Learn User Guide*, <https://mng.bz/2KNa>.

Czytelnicy zainteresowani klasyfikacją spamu w wiadomościach e-mail, a nie spamu w wiadomościach tekstowych mogą skorzystać z wymienionego poniżej obszernego zbioru danych do klasyfikacji spamu w wiadomościach e-mail w wygodnym formacie CSV, podobnym do formatu zbioru danych używanego w rozdziale 6.:

- Zbiór danych do klasyfikacji spamu e-mail, <https://mng.bz/1GEq>.

GPT-2 to model oparty na module dekodera architektury transformera, a jego głównym celem jest generowanie nowego tekstu. W zadaniach klasyfikacji można również wykorzystać modele oparte na koderach, takie jak BERT i RoBERTa:

- Devlin i współpracownicy (2018), *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, <https://arxiv.org/abs/1810.04805>.
- Liu i współpracownicy (2019), *RoBERTa: A Robustly Optimized BERT Pretraining Approach*, <https://arxiv.org/abs/1907.11692>.
- *Additional Experiments Classifying the Sentiment of 50k IMDB Movie Reviews*, <https://mng.bz/PZJR>.

Ostatnie prace dowodzą, że wydajność klasyfikacji można dodatkowo poprawić dzięki usunięciu maski przyczynowej podczas dostrajania do zadań klasyfikacji:

- Li i współpracownicy (2023), *Label Supervised LLaMA Finetuning*, <https://arxiv.org/abs/2310.01208>.
- BehnamGhader i współpracownicy (2024), *LLM2Vec: Large Language Models Are Secretly Powerful Text Encoders*, <https://arxiv.org/abs/2404.05961>.

Rozdział 7.

Zbiór danych Alpaca do dostrajania instrukcji zawiera 52 000 par instrukcja-odpowiedź. Jest jednym z pierwszych i najpopularniejszych publicznie dostępnych zbiorów danych do dostrajania modeli pod kątem wykonywania instrukcji:

- *Stanford Alpaca: An Instruction-Following Llama Model*, https://github.com/tatsu-lab/stanford_alpaca.

Spośród dodatkowych publicznie dostępnych zbiorów danych odpowiednich do dostrajania modeli pod kątem wykonywania instrukcji można wymienić następujące:

- LIMA, <https://huggingface.co/datasets/GAIR/lima>.
 - Więcej informacji można znaleźć w artykule Zhou i współpracowników, *LIMA: Less Is More for Alignment*, <https://arxiv.org/abs/2305.11206>.
- UltraChat, <https://huggingface.co/datasets/openchat/ultrachat-sharegpt>.
 - Wielkoskalowy zbiór danych składający się z 805 000 par instrukcja-odpowiedź; aby uzyskać więcej informacji, patrz Ding i współpracownicy, *Enhancing Chat Language Models by Scaling Highquality Instructional Conversations*, <https://arxiv.org/abs/2305.14233>.
- Alpaca GPT4, <https://mng.bz/Aaop>.
 - Zbiór danych podobny do Alpaca z 52 000 parami instrukcja-odpowiedź wygenerowanymi z użyciem modelu GPT-4 zamiast GPT-3.5.

Phi-3 to model o 3,8 miliardach parametrów z wariantem dostrojonym do wykonywania instrukcji, porównywalny ze znacznie większymi zastrzeżonymi modelami, takimi jak GPT-3.5:

- Abdin i współpracownicy (2024), *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*, <https://arxiv.org/abs/2404.14219>.

Eksperci zajmujący się modelami LLM zaproponowali metodę generowania syntetycznych danych modeli instrukcyjnych pozwalającą wygenerować z precyzyjnie dostrojonego modelu Llama3 300 000 wysokiej jakości par instrukcja-odpowiedź. Wstępnie przeszkolony model bazowy Llama 3 dostrojony do tych przykładów instrukcji działa porównywalnie do oryginalnego modelu Llama-3 dostrojonego do wykonywania instrukcji:

- Xu i współpracownicy (2024), *Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing*, <https://arxiv.org/abs/2406.08464>.

Badania wykazały, że brak maskowania instrukcji i wejść podczas dostrajania do wykonywania instrukcji skutecznie poprawia wydajność w różnych zadaniach NLP i testach porównawczych generowania, szczególnie w przypadku szkolenia na zbiorach danych

z długimi instrukcjami i krótkimi odpowiedziami lub w razie użycia niewielkiej liczby próbek szkoleniowych:

- Shi (2024), *Instruction Tuning with Loss Over Instructions*, <https://arxiv.org/abs/2405.14394>.

Prometheus i PHUDGE to darmowe modele LLM, które w zakresie oceny długich odpowiedzi z uwzględnieniem spersonalizowanych kryteriów dorównują modelowi GPT-4. Nie korzystałem z nich w tej książce, ponieważ w czasie, gdy ją pisałem, nie były obsługiwane przez aplikację Ollama, a zatem nie można ich było skutecznie uruchamiać na laptopie:

- Kim i współpracownicy (2023), *Prometheus: Inducing Finegrained Evaluation Capability in Language Models*, <https://arxiv.org/abs/2310.08491>.
- Deshwal i Chawla (2024), *PHUDGE: Phi-3 as Scalable Judge*, <https://arxiv.org/abs/2405.08029>.
- Kim i współpracownicy (2024), *Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models*, <https://arxiv.org/abs/2405.01535>.

Wyniki przedstawione w poniższym raporcie potwierdzają pogląd, że modele LLM przede wszystkim zdobywają wiedzę faktograficzną podczas wstępnego szkolenia, a dostrajanie głównie zwiększa ich skuteczność w korzystaniu z tej wiedzy. Co więcej, w tej analizie przedstawiono sposób, w jaki dostrajanie modeli LLM z użyciem nowych informacji wpływa na ich zdolność do wykorzystywania wcześniej istniejącej wiedzy. Ujawniono, że modele uczą się nowych faktów wolniej, a ich wprowadzanie w trakcie dostrajania zwiększa skłonność modelu do generowania niepoprawnych odpowiedzi:

- Gekhman (2024), *Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?*, <https://arxiv.org/abs/2405.05904>.

Dostrajanie do preferencji jest opcjonalnym krokiem po dostrajaniu do wykonywania instrukcji, mającym na celu ściślejsze dostosowanie modeli LLM do ludzkich preferencji. Więcej informacji na temat tego procesu zawierają dwa moje artykuły:

- *LLM Training: RLHF and Its Alternatives*, <https://mng.bz/ZVPm>.
- *Tips for LLM Pretraining and Evaluating Reward Models*, <https://mng.bz/RNXj>.

Dodatek A

O ile zapoznanie się z materiałami wymienionymi w „Dodatku A” powinno wystarczyć, aby zyskać niezbędną perspektywę, o tyle jeśli szukasz bardziej kompleksowego wprowadzenia do zagadnień uczenia głębokiego, polecam następujące książki:

- Sebastian Raschka, Hayden Liu, Vahida Mirjalili *Machine Learning with PyTorch and Scikit-Learn*, 2022, ISBN 978-1801819312.
- Eli Stevens, Luca Antiga, Thomas Viehmann, *Deep Learning with PyTorch*, 2021, ISBN 978-1617295263.

Bardziej szczegółowe wprowadzenie do pojęcia tensorów zapewni 15-minutowy samouczek wideo:

- Wykład 4.1: *Tensors in Deep Learning*, <https://www.youtube.com/watch?v=JXfDlgrfOBY>.

Jeśli chcesz dowiedzieć się więcej o ocenie modeli w uczeniu maszynowym, polecam swój artykuł:

- Sebastian Raschka (2018), *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*, <https://arxiv.org/abs/1811.12808>.

Czytelnicy zainteresowani odświeżeniem wiadomości z rachunku różniczkowego lub łagodnym wprowadzeniem w tę tematykę znajdą przydatne informacje w rozdziale poświęconym rachunkowi różniczkowemu, który jest dostępny bezpłatnie na mojej stronie internetowej:

- Sebastian Raschka, *Introduction to Calculus*, <https://mng.bz/WEyW>.

DLaczego PyTorch nie wywołuje automatycznie w tle funkcji `optimizer.zero_grad()`? W niektórych przypadkach kumulowanie gradientów może być pożądane, a framework PyTorch pozostawia taką możliwość. Jeśli chcesz dowiedzieć się więcej o akumulacji gradientu, zapoznaj się z poniższym artykułem:

- Sebastian Raschka, *Finetuning Large Language Models on a Single GPU Using Gradient Accumulation*, <https://mng.bz/8wPD>.

W tym dodatku omówiłem DDP, popularne podejście do szkolenia modeli uczenia głębokiego na wielu układach GPU. W bardziej zaawansowanych przypadkach używa, w których model nie mieści się na układzie GPU, można również rozważyć metodę frameworka PyTorch FSDP (ang. *Fully Sharded Data Parallel*), która wykorzystuje mechanizmy rozproszonej współprzecięźności danych i rozkłada duże warstwy na wiele układów GPU. Więcej informacji można znaleźć w poniższym artykule, zawierającym również dalsze odnośniki do dokumentacji API:

- *Introducing PyTorch Fully Sharded Data Parallel (FSDP) API*, <https://mng.bz/EZJR>.

Dodatek C

Rozwiązań ćwiczeń

Kompletne przykłady kodu odpowiedzi do ćwiczeń można znaleźć w repozytorium kodu tej książki, dostępnym w serwisie GitHub pod adresem <https://github.com/rasbt/LLMs-from-scratch>.

Rozdział 2.

Ćwiczenie 2.1.

Identyfikatory pojedynczych tokenów można uzyskać przez przekazywanie do tokenizera po jednym ciągu znaków na raz:

```
print(tokenizer.encode("Ak"))
print(tokenizer.encode("w"))
# ...
```

Uruchomienie tego kodu spowoduje wyświetlenie poniższego wyniku:

```
[33901]
[86]
# ...
```

Teraz można użyć następującego kodu do uzyskania oryginalnego ciągu znaków:

```
print(tokenizer.decode([33901, 86, 343, 86, 220, 959]))
```

Uruchomienie tego kodu zwraca następujący wynik:

```
'Akwirw ier'
```

Ćwiczenie 2.2

Kod komponentu ładującego dane z `max_length=2` i `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=2, stride=2
)
```

Ten komponent ładujący dane tworzy partie o następującym formacie:

```
tensor([[ 40,  367],
       [2885, 1464],
       [1807, 3619],
       [ 402,  271]])
```

Oto kod drugiego komponentu ładującego dane z `max_length=2` i `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=8, stride=2
)
```

Przykładowa partia wygląda tak:

```
tensor([[ 40,  367, 2885, 1464, 1807, 3619, 402, 271],
       [2885, 1464, 1807, 3619, 402, 271, 10899, 2138],
       [1807, 3619, 402, 271, 10899, 2138, 257, 7026],
       [ 402, 271, 10899, 2138, 257, 7026, 15632, 438]])
```

Rozdział 3.

Ćwiczenie 3.1

Poprawne przypisanie wag to:

```
sa_v1.W_query = torch.nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = torch.nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = torch.nn.Parameter(sa_v2.W_value.weight.T)
```

Ćwiczenie 3.2

Aby uzyskać wymiar wyjściowy równy 2, podobny do tego, jaki mieliśmy w przypadku uwagi jednogłowicowej, trzeba zmienić wymiar rzutowania `d_out` na 1.

```
d_out = 1
mha = MultiHeadAttentionWrapper(d_in, d_out, block_size, 0.0, num_heads=2)
```

Ćwiczenie 3.3

Oto inicjalizacja najmniejszego modelu GPT-2:

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

Rozdział 4.

Ćwiczenie 4.1

Liczبę parametrów w module ze sprzężeniem w przód i w module uwagi można obliczyć w następujący sposób:

```
block = TransformerBlock(GPT_CONFIG_124M)

total_params = sum(p.numel() for p in block.ff.parameters())
print(f"Całkowita liczba parametrów w module ze sprzężeniem w przód: {total_params:,}")

total_params = sum(p.numel() for p in block.att.parameters())
print(f"Całkowita liczba parametrów w module uwagi: {total_params:,}")
```

Jak można zauważyć, moduł ze sprzężeniem w przód zawiera około dwóch razy więcej parametrów niż moduł uwagi:

```
Całkowita liczba parametrów w module ze sprzężeniem w przód: 4,722,432
Całkowita liczba parametrów w module uwagi: 2,360,064
```

Ćwiczenie 4.2.

Aby utworzyć egzemplarze modelu GPT o innych rozmiarach, można zmodyfikować słownik konfiguracji w sposób pokazany poniżej (tutaj dla modelu GPT-2 XL):

```
GPT_CONFIG = GPT_CONFIG_124M.copy()
GPT_CONFIG["emb_dim"] = 1600
GPT_CONFIG["n_layers"] = 48
GPT_CONFIG["n_heads"] = 25
model = GPTModel(GPT_CONFIG)
```

Następnie do obliczenia liczby parametrów i zapotrzebowania na pamięć RAM możesz wykorzystać kod z podrozdziału 4.6. Oto wynik jego działania:

```
gpt2-xl:
Całkowita liczba parametrów: 1,637,792,000
Liczba trenowańnych parametrów z uwzględnieniem współdzielenia wag: 1,557,380,800
Całkowity rozmiar modelu: 6247.68 MB
```

Ćwiczenie 4.3.

W rozdziale 4. są trzy różne miejsca, w których użyliśmy warstw dropout: warstwa osadzeń, warstwa skrótów i moduł wielogłowicowej uwagi. Aby kontrolować współczynniki dropoutu dla każdej z warstw, możemy zakodować je osobno w pliku konfiguracyjnym, a następnie odpowiednio zmodyfikować implementację kodu.

Zmodyfikowana konfiguracja wygląda następująco:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,
    "emb_dim": 768,
```

```

    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1,           ← Dropout dla uwagi wielogłowicowej
    "drop_rate_shortcut": 0.1,       ← Dropout dla połączeń skrótywych
    "drop_rate_emb": 0.1,           ← Dropout dla warstwy osadzania
    "qkv_bias": False
}

```

Oto zmodyfikowane klasy TransformerBlock i GPTModel:

```

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate_attn"],   ← Dropout dla wielogłowicowej uwagi
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(
            cfg["drop_rate_shortcut"])
}

def forward(self, x):
    shortcut = x
    x = self.norm1(x)
    x = self.att(x)
    x = self.drop_shortcut(x)
    x = x + shortcut

    shortcut = x
    x = self.norm2(x)
    x = self.ff(x)
    x = self.drop_shortcut(x)
    x = x + shortcut
    return x

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate_emb"])
        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False)

```

```
)  
  
def forward(self, in_idx):  
    batch_size, seq_len = in_idx.shape  
    tok_embeds = self.tok_emb(in_idx)  
    pos_embeds = self.pos_emb(  
        torch.arange(seq_len, device=in_idx.device)  
)  
    x = tok_embeds + pos_embeds  
    x = self.drop_emb(x)  
    x = self.trf_blocks(x)  
    x = self.final_norm(x)  
    logits = self.out_head(x)  
    return logits
```

Rozdział 5.

Ćwiczenie 5.1.

Liczبę próbowania tokena (lub słowa) „pizza” można wyświetlić za pomocą funkcji `print_sampled_tokens`, którą zdefiniowaliśmy w tym podrozdziale. Zacznijmy od kodu, który zdefiniowaliśmy w podrozdziale 5.3.1.

Token „pizza” jest próbowany o razy, jeśli temperatura wynosi 0 lub 0.1, natomiast jest próbowany 32 razy, jeśli temperatura zostanie podniesiona do 5. Oszacowanie prawdopodobieństwa wynosi $\frac{32}{1000} \cdot 100\% = 3,2\%$.

Rzeczywiste prawdopodobieństwo wynosi 4,3% i jest zawarte w przeskalowanym tensorze prawdopodobieństwa softmax (`scaled_probas[2][6]`).

Ćwiczenie 5.2

Próbkowanie top-k i skalowanie temperatury to ustawienia, które należy dostosować w zależności od konkretnego modelu LLM oraz zgodnie z pożądanym stopniem różnorodności i losowości wyjść.

W przypadku używania stosunkowo małych wartości top-k (np. mniejszych niż 10) oraz gdy temperatura jest ustalona poniżej 1, wynik modelu staje się mniej losowy (jest zatem w większym stopniu deterministyczny). To ustawienie przydaje się w przypadkach, kiedy wygenerowany tekst ma być bardziej przewidywalny, spójny i — na podstawie danych szkoleniowych — bliższy najbardziej prawdopodobnych wyników.

Zastosowania dla tak niskich ustawień k i temperatury obejmują generowanie formalnych dokumentów lub raportów, w których najważniejsza jest przejrzystość i dokładność. Spośród innych przykładów zastosowań można wymienić analizę techniczną i zadania związane z generowaniem kodu, w których dokładność ma kluczowe znaczenie. Dokładnych wyników wymagają również zadania polegające na udzielaniu odpowiedzi na pytania oraz treści edukacyjne. W tych przypadkach warto zastosować temperaturę poniżej 1.

Z drugiej strony większe wartości top-k (np. wartości w zakresie od 20 do 40) i wartości temperatury powyżej 1 przydają się, gdy wykorzystujemy modele LLM do burzy mózgów lub generowania kreatywnych treści, na przykład treści fikcyjnych.

Ćwiczenie 5.3

Istnieje wiele sposobów użycia funkcji generate w celu wymuszenia deterministycznego zachowania:

1. Ustawienie `top_k=None` i brak skalowania temperatury.
2. Ustawienie `top_k=1`.

Ćwiczenie 5.4

Ogólnie rzecz biorąc, musimy załadować model i optymalizator, które zapisaliśmy wcześniej:

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
```

Następnie, aby przeszkościć model przez kolejną epokę, trzeba wywołać funkcję `train_simple_function` z parametrem `num_epochs=1`.

Ćwiczenie 5.5

Do obliczenia strat zbioru szkoleniowego i walidacyjnego modelu GPT można użyć następującego kodu:

```
train_loss = calc_loss_loader(train_loader, gpt, device)
val_loss = calc_loss_loader(val_loader, gpt, device)
```

Uzyskane wartości straty dla 124 milionów parametrów są następujące:

```
Strata zbioru szkoleniowego: 3.754748503367106
Strata zbioru walidacyjnego: 3.559617757797241
```

Najważniejszą obserwacją jest to, że wyniki zbioru szkoleniowego i walidacyjnego są na tym samym poziomie. Można to wyjaśnić na wiele sposobów:

1. Opowiadanie *The Verdict* nie było częścią zbioru danych wykorzystanego do wstępnego szkolenia w czasie, gdy firma OpenAI szkoliła model GPT-2. W związku z tym model nie jest wyraźnie nadmiernie dopasowany do zestawu szkoleniowego i działa porównywalnie na fragmentach zbioru szkoleniowego i walidacyjnego odpowiadających opowiadaniu *The Verdict* (strata zbioru walidacyjnego jest nieco mniejsza niż strata zbioru szkoleniowego, co w uczeniu głębokim jest nietypowe). Prawdopodobnie jest to jednak spowodowane losowym szumem, ponieważ zbiór danych jest stosunkowo mały. W praktyce,

jeśli nie ma nadmiernego dopasowania, oczekuje się, że wyniki straty zbioru szkoleniowego i walidacyjnego będą w przybliżeniu takie same.

2. Opowiadanie *The Verdict* było częścią zbioru danych szkoleniowych modelu GPT-2. W tym przypadku nie można stwierdzić, czy model jest nadmiernie dopasowany do danych szkoleniowych, ponieważ zbiór walidacyjny również wykorzystano do szkolenia. Aby ocenić stopień nadmiernego dopasowania, po zakończeniu szkolenia modelu GPT-2 przez OpenAI trzeba by wygenerować nowy zbiór danych. W ten sposób można sprawdzić, czy nie wykorzystywano go we wstępny szkoleniu.

Ćwiczenie 5.6

W treści rozdziału eksperymentowaliśmy z najmniejszym modelem GPT-2, który miał tylko 124 miliony parametrów. Chodziło o to, aby wymagania dotyczące zasobów były na jak najniższym poziomie. Wystarczą jednak niewielkie zmiany w kodzie, aby można było poeksperymentować z większymi modelami. Na przykład zamiast wczytywać 1558 milionów wag modelu w miejsce, jak w prezentowanym w rozdziale 5. przykładzie, 124 milionów, wystarczy zmodyfikować następujące dwa wiersze kodu:

```
hparams, params = download_and_load_gpt2(model_size="124M", models_dir="gpt2")
model_name = "gpt2-small (124M)"
```

Oto zaktualizowany kod:

```
hparams, params = download_and_load_gpt2(model_size="1558M", models_dir="gpt2")
model_name = "gpt2-xl (1558M)"
```

Rozdział 6.

Ćwiczenie 6.1

Dane wejściowe można uzupełnić do maksymalnej liczby obsługiwanych przez model tokenów przez ustawienie podczas inicjalizacji zbiorów danych maksymalnej długości na `max_length = 1024`:

```
train_dataset = SpamDataset(..., max_length=1024, ...)
val_dataset = SpamDataset(..., max_length=1024, ...)
test_dataset = SpamDataset(..., max_length=1024, ...)
```

Dodatkowe wypełnienie skutkuje jednak znacznie gorszą dokładnością testu, wynoszącą 78,33% (w porównaniu z 95,67% w przykładzie zaprezentowanym w rozdziale).

Ćwiczenie 6.2

Zamiast dostrajać tylko końcowy blok transformera, można dostroić cały model. W tym celu należy usunąć z kodu następujące wiersze:

```
for param in model.parameters():
    param.requires_grad = False
```

Ta modyfikacja skutkuje dokładnością testu poprawioną o 1%, wynoszącą 96,67% (w porównaniu z 95,67% w przykładzie zaprezentowanym w rozdziale).

Ćwiczenie 6.3

Zamiast dostrajać ostatni token wyjściowy, można dostroić pierwszy token wyjściowy. W tym celu należy zastąpić wszędzie w kodzie wywołanie `model(input_batch)[:, -1, :]` instrukcją `model(input_batch)[:, -1, -1]`.

Zgodnie z oczekiwaniemi, ponieważ pierwszy token zawiera mniej informacji niż ostatni, taka zmiana skutkuje znacznie gorszą dokładnością testu, wynoszącą 75,00% (w porównaniu z 95,67% w przykładzie zaprezentowanym w rozdziale).

Rozdział 7.

Ćwiczenie 7.1

Format promptu Phi-3, który pokazano na rysunku 7.4, dla podanego przykładowego wejścia wygląda następująco:

```
<user>
Identify the correct spelling of the following word: 'Occasion'

<assistant>
The correct spelling is 'Occasion'.
```

Aby użyć tego szablonu, można zmodyfikować funkcję `format_input` w następujący sposób:

```
def format_input(entry):
    instruction_text = (
        f"<|user|>\n{entry['instruction']}"
    )
    input_text = f"\n{entry['input']}" if entry["input"] else ""
    return instruction_text + input_text
```

Na koniec trzeba również zaktualizować sposób wyodrębniania wygenerowanej odpowiedzi podczas zbierania odpowiedzi ze zbioru testowego:

```
for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)
    tokenizer=tokenizer
    token_ids = generate(
        model=model,
```

```

        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
generated_text = token_ids_to_text(token_ids, tokenizer)
response_text = (
    generated_text[len(input_text):]           ← Nowe: zastąp ciąg ###Response ciągiem
    .replace("<|assistant|>:", "")           | <|assistant|>
    .strip()
)
test_data[i]["model_response"] = response_text

```

Dostrajanie modelu za pomocą szablonu Phi-3 jest o mniej więcej 17% szybsze, ponieważ skutkuje mniejszą objętością danych wejściowych modelu. Ocena jest bliska 50, czyli wynosi tyle samo, co w przypadku promptu w stylu Alpaca.

Ćwiczenie 7.2

Aby zamaskować instrukcje w sposób pokazany na rysunku 7.13, trzeba wprowadzić niewielkie modyfikacje do klasy `InstructionDataset` i funkcji `custom_collate_fn`. Klasę `InstructionDataset` można zmodyfikować w taki sposób, aby zbierała długość poszczególnych instrukcji, które podczas kodowania funkcji `collate` wykorzystamy do zlokalizowania pozycji zawartości instrukcji w obiektach docelowych:

```

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.instruction_lengths = []           ← Oddzielna lista do zapisywania długości instrukcji
        self.encoded_texts = []

        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Odpowiedź:\n{entry['output']}"
            full_text = instruction_plus_input + response_text

            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )
            instruction_length = (
                len(tokenizer.encode(instruction_plus_input))
            )
            self.instruction_lengths.append(instruction_length)   ← Zebranie
                                                               | długości instrukcji

    def __getitem__(self, index):           ← Zwraca długość instrukcji i jej tekst
        return self.instruction_lengths[index], self.encoded_texts[index]

    def __len__(self):
        return len(self.data)

```

Następnie zaktualizujemy funkcję `custom_collate_fn`, w której, ze względu na zmiany w klasie `InstructionDataset`, każda partia (argument `batch`) jest teraz krotką zawierającą dane (`instruction_length`, `item`) zamiast tylko `item`. Ponadto na liście docelowych identyfikatorów maskujemy odpowiednie tokeny instrukcji:

```

def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for instruction_length, item in batch)
    inputs_lst, targets_lst = [], [] ← argument batch jest teraz krotką
    for instruction_length, item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]
        padded = (
            new_item + [pad_token_id] * (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])
        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index
        targets[:instruction_length-1] = -100 ← Zamaskowanie wszystkich tokenów
                                                wejściowych i tokenów instrukcji
                                                na liście tokenów docelowych
        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]

    inputs_lst.append(inputs)
    targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)

    return inputs_tensor, targets_tensor

```

Podeczas oceny modelu dostrojonego z użyciem tej metody maskowania instrukcji można zauważyc, że osiąga on nieco gorsze wyniki (około 4 punktów z użyciem metody Ollama Llama 3 z rozdziału 7.). Jest to zgodne z obserwacjami poczynionymi w artykule *Instruction Tuning With Loss Over Instructions* (<https://arxiv.org/abs/2405.14394>).

Ćwiczenie 7.3

Aby dostroić model na oryginalnym zestawie danych Stanford Alpaca (https://github.com/tatsu-lab/stanford_alpaca), wystarczy zmienić adres URL pliku z

```
url = "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/main/ch07/01_main-
       -chapter-code/instruction-data.json"
```

na

```
url = "https://raw.githubusercontent.com/tatsu-lab/stanford_alpaca/main/alpaca_data.json"
```

Należy zauważyc, że zbiór danych zawiera 52 000 elementy (50 razy więcej niż w rozdziale 7.), które są dłuższe od tych, które wykorzystywaliśmy w rozdziale 7.

W związku z tym zdecydowanie zalecam uruchamianie szkolenia na układzie GPU.

W razie napotkania błędów braku pamięci należy rozważyć zmniejszenie rozmiaru partii z 8 do 4, 2 lub 1. Oprócz zmniejszenia rozmiaru partii warto również rozważyć zmniejszenie dozwolonej maksymalnej długości (`allowed_max_length`) z 1024 do 512 lub 256.

Poniżej znajduje się kilka przykładów ze zbioru danych Alpaca, w tym wygenerowane odpowiedzi modelu.

Ćwiczenie 7.4

Aby dostroić model z użyciem metody LoRA, należy użyć odpowiednich klas i funkcji z „Dodatku E”:

```
from appendix_E import LoRALayer, LinearWithLoRA, replace_linear_with_lora
```

Następnie poniżej kodu ładowania modelu, w podrozdziale 7.5, dodaj następujące wiersze kodu:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Całkowita liczba trenowańnych parametrów przed: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Całkowita liczba trenowańnych parametrów po: {total_params:,}")
replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Całkowita liczba trenowańnych parametrów LoRA: {total_params:,}")
model.to(device)
```

Należy zauważyc, że dostrojenie z użyciem metody LoRA na układzie GPU Nvidia L4 zajmuje 1,30 minuty. Uruchomienie oryginalnego kodu na tym samym układzie GPU zajmuje 1,80 minuty. Zatem metoda LoRA jest w przybliżeniu o 28% szybsza. Ocena metodą Ollama Llama 3 z rozdziału 7. wynosi około 50, czyli tyle samo, co dla oryginalnego modelu.

Dodatek A

Ćwiczenie A.1

Szczegółowe informacje dotyczące instalacji i konfiguracji frameworka PyTorch znajdziesz w podrozdziale A.1.3. Aby zainstalować PyTorch na swoim komputerze, postępuj zgodnie z nimi.

Ćwiczenie A.2

Pobierz notatnik ze strony <https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/Lo8/code/cross-entropy-pytorch.ipynb>. Aby sprawdzić, czy framework PyTorch jest poprawnie skonfigurowany, uruchom go na swoim komputerze.

Ćwiczenie A.3

Sieć ma dwa wejścia i dwa wyjścia. Ponadto istnieją dwie warstwy ukryte, zawierające, odpowiednio, 30 i 20 węzłów. Liczbę parametrów można programowo obliczyć w następujący sposób:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Całkowita liczba trenowalnych parametrów modelu:", num_params)
```

Uruchomienie tego kodu zwraca następujący wynik:

752

Można to również obliczyć ręcznie:

- *pierwsza warstwa ukryta* – 2 wejścia razy 30 jednostek ukrytych plus 30 jednostek stronniczości,
- *druga warstwa ukryta* – 30 jednostek wejściowych razy 20 węzłów plus 20 jednostek stronniczości,
- *warstwa wyjściowa* – 20 węzłów wejściowych razy 2 węzły wyjściowe plus 2 jednostki stronniczości.

Dodanie wszystkich parametrów w każdej z warstw daje wynik $2 \cdot 30 + 30 \cdot 20 + 20 \cdot 2 + 2 = 752$.

Ćwiczenie A.4

Dokładne czasy działania są specyficzne dla sprzętu użytego w tym eksperymencie. Gdy w swoich eksperymentach korzystałem z egzemplarza Google Colab podłączonego do układu GPU V100, zaobserwowałem znaczne przyspieszenie nawet w przypadku mnożenia małych macierzy, takich jak poniższa:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

Po użyciu układu CPU czasy były następujące:

$63.8 \mu\text{s} \pm 8.7 \mu\text{s}$ per loop

Po uruchomieniu na GPU:

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

Wynik był zaś taki:

$13.8 \mu\text{s} \pm 425 \text{ ns per loop}$

W tym przypadku układ V100 pozwolił wykonać obliczenia mniej więcej cztery razy szybciej.

Dodatek D

Usprawnianie pętli szkoleniowej

W tym dodatku usprawnimy funkcję uczenia dla procesów wstępnego uczenia i dostrajania omówionych w rozdziałach od 5. do 7. W szczególności omówię techniki *rozgrzewki tempa uczenia* (ang. *learning rate warmup*), *wygaszania kosinusowego* (ang. *cosine decay*) i *obcinania gradientu* (ang. *gradient clipping*). Następnie zastosujemy te techniki w funkcji szkoleniowej i przeprowadzimy wstępne szkolenie modelu LLM.

Aby kod był kompletny, należy ponownie zainicjować model, który przeszkoliliśmy w rozdziale 5.:

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,           ← Rozmiar słownictwa
    "context_length": 256          ← Skrócona długość kontekstu (oryginalnie: 1024)
    "emb_dim": 768,               ← Wymiar osadzeń
    "n_heads": 12,                ← Liczba głowic uwagi
    "n_layers": 12,               ← Liczba warstw
    "drop_rate": 0.1,              ← Współczynnik dropoutu
    "qkv_bias": Fałsz             ← Odchylenie zapytanie-klucz-wartość (ang. query-key-value)
}

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
model.eval()
```

Po zainicjowaniu modelu trzeba zainicjować komponenty ładujące dane. Najpierw ładujemy opowiadanie *The Verdict*:

```
import os
import urllib.request

file_path = 'the-verdict.txt'
```

```
url = (
    'https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/'
    'main/ch02/01_main-chapter-code/the-verdict.txt'
)

if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, 'w', encoding='utf-8') as file:
        file.write(text_data)
else:
    with open(file_path, 'w', encoding='utf-8') as file:
        text_data = file.read()
```

Następnie ładujemy `text_data` do obiektów ładujących dane:

```
from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M['context_length'],
    stride=GPT_CONFIG_124M['context_length'],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M['context_length'],
    stride=GPT_CONFIG_124M['context_length'],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

D.1. Rozgrzewka współczynnika uczenia

Implementacja rozgrzewki współczynnika uczenia pozwala ustabilizować szkolenie złożonych modeli, takich jak LLM. Proces ten polega na stopniowym zwiększeniu szybkości uczenia od bardzo niskiej wartości początkowej (`initial_lr`) do określonej przez użytkownika maksymalnej wartości (`peak_lr`). Rozpoczęcie szkolenia od mniejszych aktualizacji wag zmniejsza ryzyko napotkania przez model podczas fazy szkolenia dużych, destabilizujących aktualizacji.

Załóżmy, że planujemy szkolenie modelu LLM przez 15 epok, począwszy od początkowego wskaźnika uczenia 0,0001, który ma być zwiększany do maksymalnego wskaźnika uczenia 0,01:

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

Liczba kroków rozgrzewki zwykle ustawia się w przedziale od 0,1% do 20% całkowitej liczby kroków, co można obliczyć w następujący sposób:

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps) ← 20% rozgrzewki
print(warmup_steps)
```

Uruchomienie tego kodu wyświetla 27, co oznacza, że mamy 20 kroków rozgrzewki, które pozwalają zwiększyć początkowy współczynnik uczenia w pierwszych 27 krokach treningowych z 0,0001 do 0,01.

Następnie, aby zilustrować ten proces rozgrzewki, zaimplementujemy prosty szablon pętli szkoleniowej:

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps ← Ten przyrost jest określany przez poziom zwiększenia parametru initial_lr w każdym z 20 kroków rozgrzewki

global_step = -1
track_lrs = []

for epoch in range(n_epochs): ← Wykonanie typowej pętli szkoleniowej — iterowanie po partiach w obiekcie ładującym dane w każdej epoce
    dla input_batch, target_batch w train_loader:
        optimizer.zero_grad()
        global_step += -1

        if global_step < warmup_steps: ← Aktualizacja szybkości uczenia, o ile nadal trwa faza rozgrzewki
            lr = initial_lr + global_step * lr_increment
        inaczej:
            lr = peak_lr

        for param_group in optimizer.param_groups:
            param_group['lr'] = lr
        track_lrs.append(optimizer.param_groups[0]['lr']) ← W pełnej pętli szkoleniowej obliczane byłyby straty i aktualizacje modelu, które tutaj dla uproszczenia pominięto
```

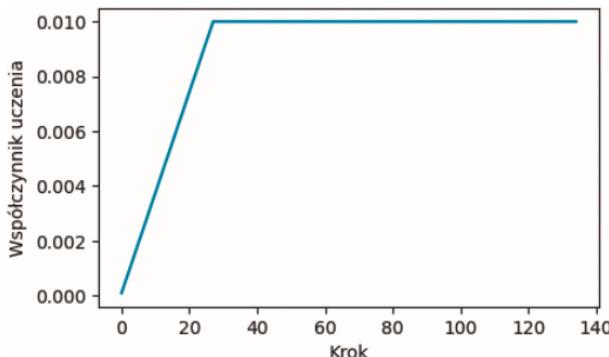
Zastosowanie do optymalizatora obliczonego współczynnika uczenia

Aby po uruchomieniu powyższego kodu sprawdzić, czy rozgrzewka szybkości uczenia działa zgodnie z zamierzeniem, możemy zwizualizować, w jaki sposób pętla szkoleniowa zmieniła szybkość uczenia modelu:

```
import matplotlib.pyplot as plt

plt.ylabel('Współczynnik uczenia')
plt.xlabel("Krok")
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs);
plt.show()
```

Uzyskany wykres pokazuje, że szybkość uczenia modelu zaczyna się od niskiej wartości i wzrasta przez 20 kroków, aż osiągnie maksymalną wartość (rysunek D.1).



Rysunek D.1. Rozgrzewka tempa uczenia zwiększa tempo uczenia przez pierwsze 20 kroków szkoleniowych. Po 20 krokach wskaźnik uczenia osiąga wartość szczytową 0,01 i przez resztę szkolenia pozostaje na stałym poziomie

Następnie będziemy dalej modyfikować szybkość uczenia się modelu, tak aby po osiągnięciu maksymalnej szybkości uczenia zmniejszała się, co dodatkowo pomaga poprawić szkolenie modelu.

D.2. Wygaszanie kosinusowe

Inną powszechnie stosowaną techniką uczenia złożonych głębokich sieci neuronowych i modeli LLM jest *wygaszanie kosinusowe*. Ta metoda polega na modulowaniu szybkości uczenia się w trakcie epok szkoleniowych, dzięki czemu po etapie rozgrzewki wykres szybkości szkolenia przypomina krzywą kosinusoidalną.

Technika wygaszania kosinusowego w swojej popularnej odmianie zmniejsza szybkość uczenia się modelu prawie do zera, co przypomina trajektorię połowy cyklu funkcji kosinus. Stopniowy spadek tempa uczenia się w technice wygaszania kosinusowego ma na celu spowolnienie tempa aktualizowania wag przez model. Jest to szczególnie ważne ze względu na zminimalizowanie ryzyka przekroczenia minimów strat podczas procesu szkolenia, co jest niezbędne do zapewnienia stabilności szkolenia w jego późniejszych fazach.

Dodanie techniki wygaszania kosinusowego pozwala zmodyfikować szablon pętli szkoleniowej:

```
import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -1

for epoch in range(n_epochs):
    dla input_batch, target_batch w train_loader:
        optimizer.zero_grad()
```

```

global_step += -1

if global_step < warmup_steps:           ← Zastosowanie liniowej rozgrzewki
    lr = initial_lr + global_step * lr_increment
inaczej:                                ← Wykorzystanie po rozgrzewce
    progress = ((global_step - warmup_steps) /      wyżarzania kosinusowego
                 (total_training_steps - warmup_steps))
    lr = min_lr + (peak_lr - min_lr) * 0.5 * (
        1 + math.cos(math.pi * postęp)
    )

for param_group in optimizer.param_groups:
    param_group['lr'] = lr
    track_lrs.append(optimizer.param_groups[0]['lr'])

```

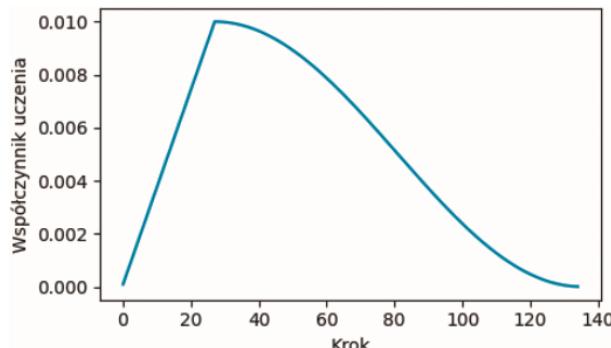
Aby sprawdzić, czy szybkość uczenia zmieniła się zgodnie z zamierzeniami, można stworzyć wykres szybkości uczenia się:

```

plt.ylabel('Współczynnik uczenia')
plt.xlabel("Krok")
plt.plot(range(total_training_steps), track_lrs)
plt.show()

```

Uzyskany wykres pokazuje, że szybkość uczenia modelu zaczyna się od fazy rozgrzewki liniowej, w której współczynnik szybkości uczenia wzrasta przez 20 kroków, aż osiągnie wartość maksymalną. Po 20 krokach liniowej rozgrzewki rozpoczyna się wygaszanie kosinusowe, które zmniejsza szybkość uczenia się modelu aż do osiągnięcia minimum (rysunek D.2).



Rysunek D.2. Po pierwszych 20 krokach liniowej rozgrzewki tempa uczenia następuje faza wygaszania kosinusowego, w której tempo uczenia zmniejsza się w cyklu półkosinusoidalnym, aż do osiągnięcia na końcu szkolenia wartości minimum

D.3. Przycinanie gradientu

Kolejną ważną techniką zwiększenia stabilności podczas szkolenia modeli LLM jest *przycinanie gradientu*. Metoda ta polega na ustawieniu progu, powyżej którego gradienty są skalowane w dół do określonej wartości maksymalnej. Zastosowanie tego procesu sprawia, że aktualizacje parametrów modelu w trakcie propagacji wstępnej pozostają w rozsądnym zakresie.

Na przykład dzięki ustawieniu `max_norm=1.0` w funkcji `clip_grad_norm` framework PyTorch zapewnia, że norma gradientów nie przekroczy wartości 1.0. W tym przypadku termin „norma” oznacza miarę długości lub wielkości wektora gradientu w przestrzeni parametrów modelu. W szczególności odnosi się do normy L2, znanej również jako norma euklidesowa.

W kategoriach matematycznych dla wektora v składającego się z elementów $\mathbf{v} = [v_1, v_2, \dots, v_n]$ normę L2 można określić w następujący sposób:

$$\|v\|_2 = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

Tę metodę obliczeniową stosuje się również do macierzy. Dla przykładu, rozważmy macierz gradientu określoną jako:

$$\mathbf{G} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Aby przyciąć te gradienty do `max_norm` równej 1, najpierw obliczamy normę L2 tych gradientów, która wynosi:

$$\|\mathbf{G}\|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

Ze względu na to, że wartość $\|\mathbf{G}\|_2 = 5$ przekracza wartość `max_norm` ustawioną na 1, skalujemy gradienty w dół, aby ich norma była równa dokładnie 1. Osiąga się to dzięki użyciu współczynnika skalowania, obliczanego jako $\text{max_norm}/\|\mathbf{G}\|_2 = 1/5$. W rezultacie skorygowana macierz gradientów \mathbf{G}' przyjmuje następującą postać:

$$\mathbf{G}' = \frac{1}{5} \times \mathbf{G} = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{2}{5} & \frac{4}{5} \end{bmatrix}$$

Aby zilustrować proces przycinania gradientu, zaczynamy, podobnie jak w przypadku standardowej pętli szkoleniowej, od zainicjowania nowego modelu i obliczenia dla niej straty:

```
from chapter05 import calc_loss_batch

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
```

```
loss = calc_loss_batch(input_batch, target_batch, model, urządzenie)
loss.backward()
```

Po wywołaniu metody `.backward()` framework PyTorch oblicza gradienty strat i dla każdego tensora wagi (parametru) modelu przechowuje je w atrybutie `.grad`.

Aby wyjaśnić tę kwestię, możesz zdefiniować pokazaną poniżej funkcję `find_highest_gradient`, która po wywołaniu funkcji `.backward()` identyfikuje w wyniku skanowania atrybutów `.grad` wszystkich tensorów wag modelu najwyższą wartość gradientu:

```
def find_highest_gradient(model):
    max_grad = Brak
    for param in model.parameters():
        jeśli param.grad nie jest None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            jeśli max_grad to None lub max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad
print(find_highest_gradient(model))
```

Najwyższa wartość gradientu zidentyfikowana przez powyższy kod to:

```
tensor(0.0411)
```

Zastosujmy teraz technikę przycinania gradientu, aby zaobserwować, w jaki sposób wpływa to na największą wartość gradientu:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
```

Najwyższa wartość gradientu po zastosowaniu przycinania gradientu z maksymalną normą 1 jest znacznie mniejsza niż wcześniej:

```
tensor(0.0185)
```

D.4. Zmodyfikowana funkcja szkoleniowa

Na koniec usprawnimy funkcję szkoleniową `train_model_simple` (patrz rozdział 5.) przez zastosowanie wprowadzonych w niniejszym dokumencie trzech technik: rozgrzewki liniowej, wygaszania kosinusowego i przycinania gradientu. Te trzy metody zastosowane łącznie pomagają ustabilizować szkolenie LLM.

Kod ze zmianami w porównaniu z funkcją `train_model_simple`, wraz z odpowiednimi adnotacjami, wygląda następująco:

```
from chapter05 import evaluate_model, generate_and_print_sample

def train_model(model, train_loader, val_loader, optimizer, device,
               n_epochs, eval_freq, eval_iter, start_context, tokenizer,
               warmup_steps, initial_lr=3e-05, min_lr=1e-06):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
```

```

tokens_seen, global_step = 0, -1
peak_lr = optimizer.param_groups[0]['lr']
total_training_steps = len(train_loader) * n_epochs
lr_increment = (peak_lr - initial_lr) / warmup_steps
for epoch in range(n_epochs):
    model.train()
    dla input_batch, target_batch w train_loader:
        optimizer.zero_grad()
        global_step += 1
        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        inaczej:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                1 + math.cos(math.pi * progress))
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr
        track_lrs.append(lr)
        loss = calc_loss_batch(input_batch, target_batch, model, urządzenie)
        loss.backward()

        if global_step > warmup_steps:
            torch.nn.utils.clip_grad_norm_(
                model.parameters(), max_norm=1.0
            )
    optimizer.step()
    tokens_seen += input_batch.numel()

    if global_step % eval_freq == 0:
        train_loss, val_loss = evaluate_model(
            model, train_loader, val_loader,
            urządzenie, eval_iter
        )
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        track_tokens_seen.append(tokens_seen)
        print(f'Epoka {epoch+1} (Iteracja {global_step:06d}): '
              f'Strata szkoleniowa {train_loss:.3f}, '
              f'Strata walidacyjna {val_loss:.3f}')
    )

    generate_and_print_sample(
        model, tokenizer, urządzenie, start_context
    )
return train_losses, val_losses, track_tokens_seen, track_lrs

```

Odczytanie z optymalizatora początkowego tempa uczenia, przy założeniu, że używamy go jako szczytowego tempa uczenia

Obliczenie całkowitej liczby iteracji w procesie szkolenia

Obliczenie przyrostu szybkości uczenia podczas fazy rozgrzewki

Dostosowanie szybkości uczenia na podstawie bieżącej fazy (rozgrzewka lub kosinusowe wyżarzanie)

Zastosowanie obliczonego współczynnika uczenia do optymalizatora

Zastosowanie po fazie rozgrzewki przycinania gradientu w celu uniknięcia eksplozji gradientów

zajdujący się poniżej kod pozostaje niezmieniony w porównaniu z użytą w rozdziale 5. funkcją train_model_simple.

Po zdefiniowaniu funkcji `train_model` można z niej skorzystać w sposób podobny do tego, w jaki do szkolenia wstępnego użyliśmy metody `train_model_simple`:

```

import tiktoken

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 5e-4
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
tokenizer = tiktoken.get_encoding('gpt2')

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=n_epochs,
    eval_freq=5, eval_iter=1, start_context="Every effort moves you",
    tokenizer=tokenizer, warmup_steps=warmup_steps,
    initial_lr=1e-5, min_lr=1e-5
)

```

Szkolenie na MacBooku Air lub podobnym laptopie zajmie około 5 minut. Uruchomienie kodu spowoduje wyświetlenie następujących wyników:

```

Epoka 1 (Iteracja 000000): Strata szkoleniowa 10.924, Strata walidacyjna 10.939
Epoka 1 (Iteracja 000005): Strata szkoleniowa 9.317, Strata walidacyjna 9.460
Every effort moves you,,,,,,,,,,,
Epoka 2 (Iteracja 000010): Strata szkoleniowa 7.971, Strata walidacyjna 8.179
Epoka 2 (Iteracja 000015): Strata szkoleniowa 6.589, Strata walidacyjna 6.852
Every effort moves you,,,,,,,,,,,
...
Epoka 15 (Iteracja 000130): Strata szkoleniowa 0.041, Strata walidacyjna 6.915
Every effort moves you?" "Yes--quite insensible to the irony. She wanted him
→vindicated--and by me!" He laughed again, and threw back his head to look up at
→the sketch of the donkey. "There were days when I

```

Podobnie jak w przypadku szkolenia wstępnego, po kilku epokach model zaczyna się nadmiernie dopasowywać, co wynika z wykorzystania bardzo małego zbioru danych, po którym wielokrotnie iterujemy. Niemniej jednak można zauważyć, że funkcja działa, ponieważ minimalizuje stratę w zbiorze szkoleniowym.

Zachęcam Czytelników do przeszkoletnia modelu na większym zbiorze danych tekstowych i porównania wyników uzyskanych za pomocą tej bardziej zaawansowanej funkcji szkoleniowej z wynikami, które uzyskaliśmy za pomocą funkcji `train_model_simple`.

Dodatek E

Skuteczne dostrajanie parametrów za pomocą LoRA

Adaptacja niskiego rzędu (ang. *Low-rank adaptation* – LoRA) jest jedną z najczęściej stosowanych technik dostrajania parametrów. Poniższy opis opiera się na przykładzie dostrajania klasyfikacji spamu podanym w rozdziale 6. Dostrajanie LoRA ma jednak również zastosowanie do nadzorowanego dostrajania instrukcji omówionego w rozdziale 7.

E.1. Wprowadzenie do LoRA

LoRA to technika, która dostosowuje wstępnie przeszkolony model, aby lepiej pasował do określonego, często mniejszego zbioru danych, przez dostosowanie tylko niewielkiego podzbioru parametrów wagowych modelu. Aspekt „niskiego rzędu” odnosi się do matematycznego pojęcia ograniczenia dostosowań modelu do przestrzeni – o mniejszej liczbie wymiarów – całkowitej przestrzeni parametrów wag. Pozwala to skutecznie uchwycić najbardziej wpływowe kierunki zmian parametrów wag podczas szkolenia. Metoda LoRA jest przydatna i popularna, ponieważ umożliwia wydajne dostrajanie dużych modeli na danych specyficznych dla zadania, co znacznie zmniejsza koszty obliczeniowe i wymagania co do zasobów niezbędnych do dostrajania.

Załóżmy, że duża macierz wag W jest powiązana z określona warstwą. Technikę LoRA można zastosować do wszystkich warstw liniowych w modelu LLM. W celach ilustracyjnych skupimy się jednak na pojedynczej warstwie.

Podczas szkolenia głębokich sieci neuronowych, w trakcie propagacji wstecznej, model uczy się macierzy ΔW , która zawiera informacje o tym, jak bardzo chcemy zaktualizować oryginalne parametry wag, aby zminimalizować funkcję straty podczas szkolenia. W dalszej części tego dodatku będę używał terminu „waga” jako krótszego określenia parametrów wag modelu.

W zwykłym szkoleniu i dostrajaniu aktualizacja wagi jest zdefiniowana w następujący sposób:

$$W_{zaktualizowane} = W + \Delta W$$

Metoda LoRA, zaproponowana przez Hu i współpracowników (<https://arxiv.org/abs/2106.09685>), oferuje wydajniejszą alternatywę dla obliczania aktualizacji wag ΔW dzięki uczeniu się jej przybliżenia:

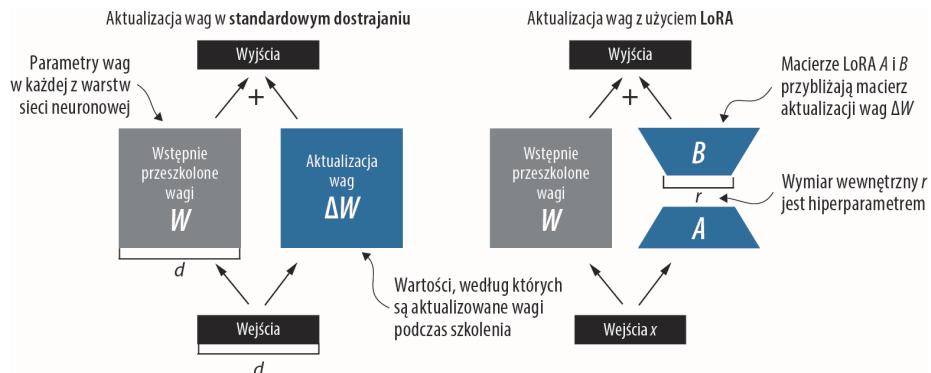
$$\Delta W \approx AB$$

gdzie A i B są dwiema macierzami znacznie mniejszymi niż W , a AB reprezentuje iloczyn macierzy A i B .

Dzięki technice LoRA można przeformułować aktualizację wag, którą zdefiniowaliśmy wcześniej:

$$W_{zaktualizowane} = W + AB$$

Formuły aktualizacji wag dla pełnego dostrajania oraz dostrajania techniką LoRA przedstawiłem obok siebie na rysunku E.1.



Rysunek E.1. Zwykłe dostrajanie obejmuje aktualizację wstępnie przeszkolonej macierzy wag W bezpośrednio za pomocą ΔW (lewa strona rysunku). W technice LoRA wykorzystuje się dwie mniejsze macierze A i B do przybliżenia ΔW . Do macierzy W dodawany jest iloczyn AB ; r oznacza wymiar wewnętrzny, dostrajalny hiperparametrum (prawa strona rysunku)

Uważni Czytelnicy z pewnością zauważą, że wizualne reprezentacje pełnego dostrajania i dostrajania techniką LoRA na rysunku E.1 różnią się nieco od wcześniej przedstawionych formuł. Ta zmiana wynika z prawa rozdzielności mnożenia macierzy, które pozwala oddzielić oryginalne i zaktualizowane wagę, zamiast je łączyć. Na przykład w przypadku zwykłego dostrajania z macierzą wejściową x obliczenia można przedstawić w następujący sposób:

$$x(W + \Delta W) = xW + x\Delta W$$

Z kolei dla LoRA można zapisać następujący wzór:

$$x(W + AB) = xW + xAB$$

Zastosowanie techniki LoRA nie tylko zmniejsza liczbę wag do aktualizacji podczas szkolenia, ale przede wszystkim pozwala oddzielić macierz wag LoRA od oryginalnych wag modelu. Dzięki temu technika LoRA jest tak przydatna w praktyce. Jej zastosowanie pozwala na pozostawienie wstępnie wyuczonych wag modelu bez zmian, a macierze LoRA są stosowane dynamicznie po szkoleniu, w fazie korzystania z modelu.

Oddzielenie wag LoRA jest bardzo przydatne, ponieważ pozwala na dostrajanie modelu LLM bez konieczności przechowywania jego wielu kompletnych wersji. Zmniejsza to wymagania dotyczące pamięci dyskowej i poprawia skalowalność, ponieważ dostrajanie LLM na potrzeby każdego konkretnego klienta lub aplikacji wymaga dostosowania i zapisania wyłącznie mniejszych macierzy LoRA.

Zobaczmy teraz, jak można wykorzystać LoRA w zadaniu dostrojenia LLM do klasyfikacji spamu, podobnym do przykładu dostrajania, który zastosowaliśmy w rozdziale 6.

E.2. Przygotowanie zbioru danych

Przed zastosowaniem LoRA w przykładzie klasyfikacji spamu musimy załadować zbiór danych i wstępnie przeszkolony model, z którym będziemy pracować. Kod zamieszczony w tym podrozdziale powtarza przygotowanie danych z rozdziału 6. (zamiast powtarzać kod, mógłbyś otworzyć i uruchomić notatnik z rozdziału 6. i wstawić tam kod LoRA z podrozdziału E.4). Najpierw pobieramy zbiór danych i zapisujemy go w formacie CSV (listing E.1).

Listing E.1. Pobieranie i przygotowywanie zbioru danych

```
from pathlib import
Path import pandas as pd
from ch06 import (
    download_and_unzip_spam_data,
    create_balanced_dataset,
    random_split
)
url = \
https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSpamCollection.tsv"

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
```

```

)
balanced_df = create_balanced_dataset(df)
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)
train_df.to_csv("train.csv", index=None)

validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)

```

Następnie tworzymy egzemplarze obiektów SpamDataset (listing E.2).

Listing E.2. Utworzenie egzemplarzy zbiorów danych PyTorch

```

import torch
from torch.utils.data import Dataset
import tiktoken
from chapter06 import SpamDataset

tokenizer = tiktoken.get_encoding("gpt2")
train_dataset = SpamDataset("train.csv", max_length=None,
    tokenizer=tokenizer)
val_dataset = SpamDataset("validation.csv",
    max_length=train_dataset.max_length, tokenizer=tokenizer)
test_dataset = SpamDataset(
    "test.csv", max_length=train_dataset.max_length, tokenizer=tokenizer)

```

Po utworzeniu obiektów reprezentujących zbiory danych PyTorch tworzymy egzemplarze obiektów odpowiedzialnych za ładowanie danych (listing E.3).

Listing E.3. Tworzenie obiektów ładujących dane frameworka PyTorch

```

from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)

val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
)

```

```
        drop_last=False,  
    )  
  
    test_loader = DataLoader(  
        dataset=test_dataset,  
        batch_size=batch_size,  
        num_workers=num_workers,  
        drop_last=False,  
    )
```

W celu weryfikacji iterujemy po obiektach ładujących dane i sprawdzamy, czy partie zawierają po 8 próbek szkoleniowych, z których każda składa się ze 120 tokenów:

```
print("Szkoleniowy mechanizm ładujący:")  
for input_batch, target_batch in train_loader:  
    pass  
  
print("Wymiary partii wejściowej:", input_batch.shape)  
print("Wymiary partii etykiet:", target_batch.shape)
```

Oto uzyskane wyniki:

```
Szkoleniowy mechanizm ładujący:  
Wymiary partii wejściowej: torch.Size([8, 120])  
Wymiary partii etykiet: torch.Size([8])
```

Na koniec wyświetlamy całkowitą liczbę partii w każdym zbiorze danych:

```
print(f"{len(train_loader)} partii szkoleniowych")  
print(f"{len(val_loader)} partii walidacyjnych")  
print(f"{len(test_loader)} partii testowych")
```

W tym przypadku w każdym zbiorze danych mamy następującą liczbę partii:

```
130 partii szkoleniowych  
19 partii walidacyjnych  
38 partii testowych
```

E.3. Inicjalizacja modelu

Aby załadować i przygotować wstępnie przeszkolony model GPT, powtarzamy kod z rozdziału 6. Zaczynamy od pobrania wag modelu i załadowania ich do klasy `GPTModel` (listing E.4).

Listing E.4. Ładowanie wstępnie przeszkolonego modelu GPT

```
from gpt_download import download_and_load_gpt2  
from chapter04 import GPTModel  
from chapter05 import load_weights_into_gpt  
  
CHOOSE_MODEL = "gpt2-small (124M)"  
INPUT_PROMPT = "Every effort moves"  
  
BASE_CONFIG = {
```

```

"vocab_size": 50257,           ← Rozmiar słownictwa
"context_length": 1024,        ← Długość kontekstu
"drop_rate": 0.0,             ← Wskaźnik dropoutu
"qkv_bias": True              ← Odchylenie zapytanie-klucz-wartość (ang. query-key-value)
}

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)
model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()

```

Aby się upewnić, że model został załadowany poprawnie, sprawdźmy, czy generuje spójny tekst:

```

from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))

```

Poniższy wynik dowodzi, że model generuje spójny tekst, co wskazuje na to, że wagi modelu zostały załadowane poprawnie:

```

Every effort moves you forward.
The first step is to understand the importance of your work

```

W tłumaczeniu na język polski:

Każdy wysiłek posuwa cię naprzód.
Pierwszy krok to zrozumienie znaczenia twojej pracy

Następnie przygotowujemy model do dostrajania pod kątem zadań klasyfikacji. W tym celu postępujemy podobnie jak w rozdziale 6., w którym zastąpiliśmy warstwę wyjściową:

```

torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(in_features=768, out_features=num_classes)

```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Na koniec obliczamy początkową dokładność klasyfikacji niedostrojonego modelu (oczekujemy, że wyniesie około 50%, co oznacza, że model jeszcze nie jest w stanie niezawodnie rozróżniać spamu i wiadomości niebędących spamem):

```
from chapter06 import calc_accuracy_loader

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Dokładność zbioru szkoleniowego: {train_accuracy*100:.2f}%")
print(f"Dokładność zbioru walidacyjnego: {val_accuracy*100:.2f}%")
print(f"Dokładność zbioru testowego: {test_accuracy*100:.2f}%")
```

Oto początkowe dokładności prognozowania:

```
Dokładność zbioru szkoleniowego: 46.25%
Dokładność zbioru walidacyjnego: 45.00%
Dokładność zbioru testowego: 48.75%
```

E.4. Dostrajanie PEFT z użyciem techniki LoRA

W kolejnym kroku modyfikujemy model LLM i dostrajamy go z użyciem techniki LoRA. Zaczynamy od zainicjowania warstwy LoRALayer, która tworzy macierze A i B , wraz ze współczynnikiem skalowania α i konfiguracją $\text{rank}(r)$. Jak pokazano na rysunku E.2, ta warstwa może przyjmować wejścia i obliczać odpowiadające im wyjścia.

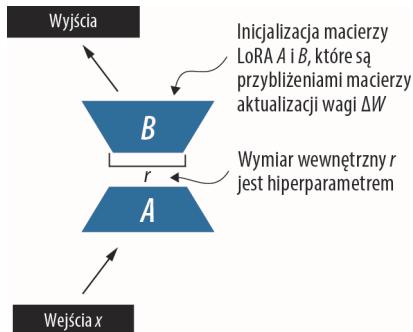
Tę warstwę LoRA można zaimplementować w kodzie w sposób pokazany na listingu E.5.

Listing E.5. Implementacja warstwy LoRA

```
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
```

Ta sama inicjalizacja
jest używana dla warstw
liniowych w PyTorch



Rysunek E.2. Macierze LoRA A i B są stosowane do wejść warstwy i uwzględniane podczas obliczania wyników modelu. Wewnętrzny wymiar r tych macierzy służy jako ustawienie, które przez zmianę rozmiarów macierzy A i B dostosowuje liczbę trenowalnych parametrów

```
self.alpha = alpha

def forward(self, x):
    x = self.alpha * (x @ self.A @ self.B)
    return x
```

Argument rank decyduje o wewnętrznym wymiarze macierzy A i B. W gruncie rzeczy to ustawienie określa liczbę dodatkowych parametrów wprowadzanych przez LoRA. W ten sposób, dzięki dostosowywaniu liczby używanych parametrów, możemy zachować równowagę między zdolnością modelu do adaptacji a jego wydajnością.

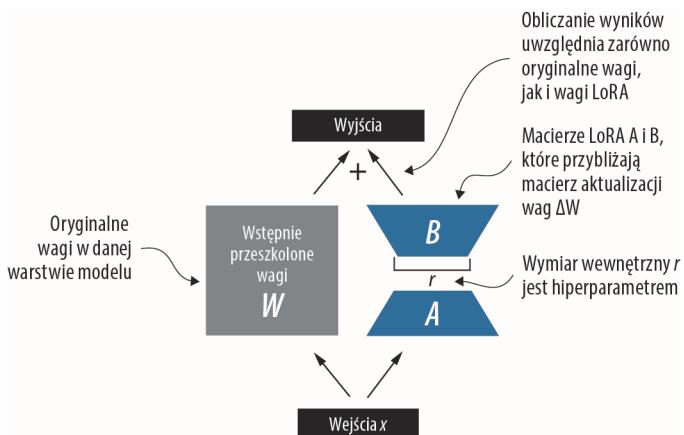
Inne ważne ustawienie, `alpha`, pełni funkcję współczynnika skalowania dla wyników adaptacji niskiego rzędu. Decyduje ono przede wszystkim o stopniu, w jakim wyniki dostrojonej warstwy mogą wpływać na wyniki warstwy oryginalnej. Ten współczynnik można postrzegać jako sposób na regulowanie wpływu adaptacji niskiej rangi na wyniki warstwy. Przekształcanie wejść warstwy umożliwia klasa `LoRALayer`, którą zaimplementowaliśmy wcześniej.

Celem techniki LoRA zwykle jest zastąpienie istniejących warstw liniowych, co umożliwia bezpośrednią aktualizację wcześniej przeszkołonych wag (rysunek E.3).

Aby zintegrować wagi oryginalnej warstwy `Linear`, utworzymy teraz warstwę `LinearWithLoRA`. Ta warstwa wykorzystuje wcześniej zaimplementowaną warstwę `LoRALayer`. Zaprojektowano ją w celu zastąpienia istniejących warstw liniowych w sieci neuronowej, takich jak moduły samouwagi lub moduły sprzężenia zwrotnego w GPT `Model` (listing E.6).

Listing E.6. Zastąpienie warstwy `LinearWithLora` warstwami `Linear`

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
```



Rysunek E.3. Integracja LoRA z warstwą modelu. Oryginalne wstępnie wyuczone wagi (W) warstwy są łączone z danymi wyjściowymi z macierzy LoRA (A i B), będącymi przybliżeniami macierzy aktualizacji wag (ΔW). Ostateczne wyjście jest obliczane przez dodanie wyjścia dostosowanej warstwy (z użyciem wag LoRA) do oryginalnego wyjścia

```

        linear.in_features, linear.out_features, rank, alpha
    )

def forward(self, x):
    return self.linear(x) + self.lora(x)

```

Powyższy kod łączy standardową warstwę Linear z warstwą LoRALayer. Metoda forward oblicza wynik przez dodanie wyników z oryginalnej warstwy liniowej i warstwy LoRA.

Ponieważ macierz wag B (`self.B` w klasie LoRALayer) jest inicjowana wartościami zerowymi, iloczyn macierzy A i B daje macierz zerową. Dzięki temu mnożenie nie zmienia oryginalnych wag, ponieważ dodanie zer ich nie zmienia.

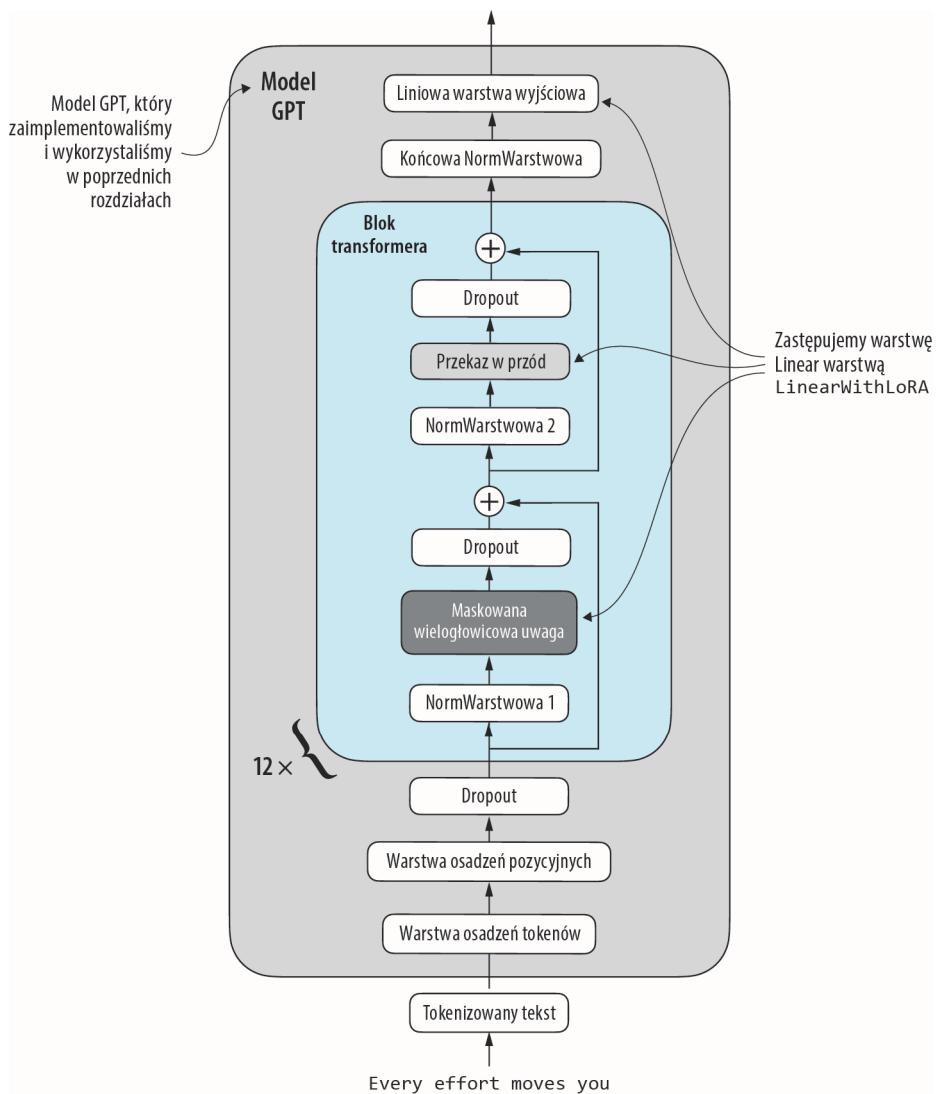
Aby zastosować technikę LoRA do wcześniejszej zdefiniowanego modelu GPTModel, wprowadzamy funkcję `replace_linear_with_lora`. Ta funkcja zamieni wszystkie istniejące warstwy Linear w modelu na nowo utworzone warstwy LinearWithLoRA:

```

def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):           Zastępuje warstwę Linear
            setattr(model, name, LinearWithLoRA(module, rank, alpha))   warstwą LinearWithLoRA
        else:
            replace_linear_with_lora(module, rank, alpha)           Rekurencyjnie stosuje tę samą
                                                               funkcję do modułów potomnych

```

W celu wydajnego dostrajania parametrów zaimplementowaliśmy cały niezbędny kod pozwalający zastąpić warstwy liniowe w GPTModel nowo opracowanymi warstwami LinearWithLoRA. W kolejnym kroku zastosujemy zaktualizowaną klasę LinearWithLORA do wszystkich warstw Linear znalezionych w wielogłowicowej uwadze, modułach sprzężenia zwrotnego i warstwie wyjściowej modelu GPTM (rysunek E.4).



Rysunek E.4. Architektura modelu GPT. Podkreśla te części modelu, w których — w celu skutecznego dostrajania parametrów — warstwy Linear zostały zaktualizowane do warstw LinearWithLoRA

Zanim zastosujemy ulepszenia warstwy LinearWithLoRA, najpierw zamrażamy oryginalne parametry modelu:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Całkowita liczba trenowalnych parametrów przed: {total_params},")
```

```
for param in model.parameters():
    param.requires_grad = False
```

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Całkowita liczba trenowalnych parametrów po: {total_params:,}")
```

Możemy teraz zauważyc, że żaden ze 124 milionów parametrów modelu nie jest trenowalny:

```
Całkowita liczba trenowalnych parametrów przed: 124,441,346
Całkowita liczba trenowalnych parametrów po: 0
```

Następnie w celu zastąpienia warstw Linear używamy funkcji `replace_linear_with_lora`:

```
replace_linear_with_lora(model, rank=16, alpha=16)
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Całkowita liczba trenowalnych parametrów LoRA: {total_params:,}")
```

Po dodaniu warstw LoRA liczba trenowalnych parametrów jest następująca:

```
Całkowita liczba trenowalnych parametrów LoRA: 2,666,528
```

Jak widać, dzięki zastosowaniu LoRA zmniejszyliśmy liczbę trenowalnych parametrów prawie 50-krotnie. Wartości rank i alpha równe 16 to dobre ustawienia domyślne. Często jednak zwiększa się również parametr rank, który z kolei zwiększa liczbę trenowalnych parametrów. Parametr alpha zwykle ustawia się na wartość równą połowie parametru rank, jego dwukrotności bądź też oba argumenty są równe.

Sprawdźmy, czy warstwy zostały zmodyfikowane zgodnie z zamierzeniem. W tym celu wyświetlimy architekturę modelu:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(model)
```

Oto uzyskane wyniki:

```
GPTModel(
    (tok_emb): Embedding(50257, 768)
    (pos_emb): Embedding(1024, 768)
    (drop_emb): Dropout(p=0.0, inplace=False)
    (trf_blocks): Sequential(
        ...
        (11): TransformerBlock(
            (att): MultiHeadAttention(
                (W_query): LinearWithLoRA(
                    (linear): Linear(in_features=768, out_features=768, bias=True)
                    (lora): LoRALayer()
                )
                (W_key): LinearWithLoRA(
                    (linear): Linear(in_features=768, out_features=768, bias=True)
                    (lora): LoRALayer()
                )
                (W_value): LinearWithLoRA(
                    (linear): Linear(in_features=768, out_features=768, bias=True)
                    (lora): LoRALayer()
                )
            (out_proj): LinearWithLoRA(
                (linear): Linear(in_features=768, out_features=768, bias=True)
                (lora): LoRALayer()
            )
        )
    )
)
```

```

        )
        (dropout): Dropout(p=0.0, inplace=False)
    )
    (ff): FeedForward(
        (layers): Sequential(
            (0): LinearWithLoRA(
                (linear): Linear(in_features=768, out_features=3072, bias=True)
                (lora): LoRALayer()
            )
            (1): GELU()
            (2): LinearWithLoRA(
                (linear): Linear(in_features=3072, out_features=768, bias=True)
                (lora): LoRALayer()
            )
        )
    )
    (norm1): LayerNorm()
    (norm2): LayerNorm()
    (drop_resid): Dropout(p=0.0, inplace=False)
)
)
(final_norm): LayerNorm()
(out_head): LinearWithLoRA(
    (linear): Linear(in_features=768, out_features=2, bias=True)
    (lora): LoRALayer()
)
)
)

```

Model zawiera teraz nowe warstwy `LinearWithLoRA`, które same składają się z oryginalnych warstw `Linear`, ustawionych jako nietrenowalne, oraz nowych warstw `LoRA`, które będziemy dostrajać.

Przed przystąpieniem do dostrajania modelu obliczamy początkową dokładność klasyfikacji:

```

torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Dokładność zbioru szkoleniowego: {train_accuracy*100:.2f}%")
print(f"Dokładność zbioru walidacyjnego: {val_accuracy*100:.2f}%")
print(f"Dokładność zbioru testowego: {test_accuracy*100:.2f}%")

```

Uzyskane wartości dokładności to:

```

Dokładność zbioru szkoleniowego: 46.25%
Dokładność zbioru walidacyjnego: 45.00%
Dokładność zbioru testowego: 48.75%

```

Uzyskane wartości dokładności są identyczne z wartościami z rozdziału 6. Taki wynik uzyskaliśmy dlatego, że macierz LoRA B została zainicjowana zerami. W konsekwencji iloczyn macierzy AB daje macierz zerową. Dzięki temu mnożenie nie zmienia oryginalnych wag, ponieważ dodanie zer ich nie zmienia.

Przejdzmy teraz do bardziej interesującej części — dostrajania modelu za pomocą funkcji szkoleniowej z rozdziału 6. Szkolenie na laptopie M3 MacBook Air zajmuje około 15 minut i mniej niż pół minuty na układzie GPU V100 lub A100 (listing E.7).

Listing E.7. Dostrajanie modelu z użyciem warstw LoRA

```
import time
from chapter06 import train_classifier_simple

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50, eval_iter=5,
        tokenizer=tokenizer
    )
end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Szkolenie zakończono w ciągu {execution_time_minutes:.2f} minut.")
```

Oto wyniki, które obserwujemy podczas szkolenia:

```
Epoka 1 (Krok 000000): Strata szkoleniowa 3.820, Strata walidacyjna 3.462
Epoka 1 (Krok 000050): Strata szkoleniowa 0.396, Strata walidacyjna 0.364
Epoka 1 (Krok 000100): Strata szkoleniowa 0.111, Strata walidacyjna 0.229
Dokładność zbioru szkoleniowego: 97.50% | Dokładność zbioru walidacyjnego: 95.00%
Epoka 2 (Krok 000150): Strata szkoleniowa 0.135, Strata walidacyjna 0.073
Epoka 2 (Krok 000200): Strata szkoleniowa 0.008, Strata walidacyjna 0.052
Epoka 2 (Krok 000250): Strata szkoleniowa 0.021, Strata walidacyjna 0.179
Dokładność zbioru szkoleniowego: 97.50% | Dokładność zbioru walidacyjnego: 97.50%
Epoka 3 (Krok 000300): Strata szkoleniowa 0.096, Strata walidacyjna 0.080
Epoka 3 (Krok 000350): Strata szkoleniowa 0.010, Strata walidacyjna 0.116
Dokładność zbioru szkoleniowego: 97.50% | Dokładność zbioru walidacyjnego: 95.00%
Epoka 4 (Krok 000400): Strata szkoleniowa 0.003, Strata walidacyjna 0.151
Epoka 4 (Krok 000450): Strata szkoleniowa 0.008, Strata walidacyjna 0.077
Epoka 4 (Krok 000500): Strata szkoleniowa 0.001, Strata walidacyjna 0.147
Dokładność zbioru szkoleniowego: 100.00% | Dokładność zbioru walidacyjnego: 97.50%
Epoka 5 (Krok 000550): Strata szkoleniowa 0.007, Strata walidacyjna 0.094
Epoka 5 (Krok 000600): Strata szkoleniowa 0.000, Strata walidacyjna 0.056
Dokładność zbioru szkoleniowego: 100.00% | Dokładność zbioru walidacyjnego: 97.50%
```

Szkolenie zakończone w 12.10 minut.

Szkolenie modelu z użyciem techniki LoRA trwało dłużej niż szkolenie bez LoRA (patrz rozdział 6.), ponieważ warstwy LoRA wprowadzają dodatkowe obliczenia podczas

przejścia do przodu. Jednak w przypadku większych modeli, kiedy to propagacja wsteczna staje się bardziej kosztowna, szkolenie modeli z użyciem techniki LoRA zazwyczaj przebiega szybciej niż bez jej zastosowania.

Jak można zauważać, dla modelu przeprowadzono doskonałe szkolenie, w którego wyniku uzyskano bardzo wysoką dokładność zbioru walidacyjnego.

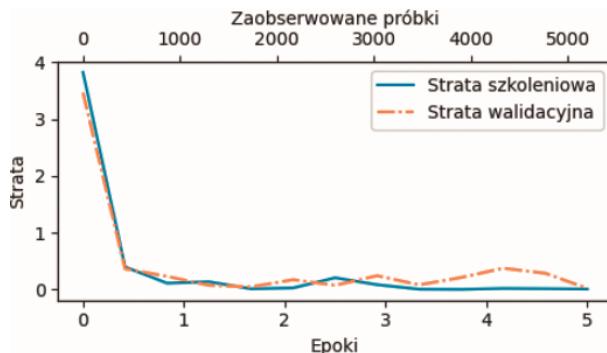
Aby lepiej zaobserwować, czy szkolenie było zbieżne, spróbujmy zwizualizować krzywe strat:

```
from chapter06 import plot_values

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(
    epochs_tensor, examples_seen_tensor,
    train_losses, val_losses, label="loss"
)
```

Wyniki przedstawiłem na rysunku E.5.



Rysunek E.5. Krzywe strat zbiorów szkoleniowego i walidacyjnego dla modelu uczenia maszynowego na przestrzeni sześciu epok. Początkowo straty zarówno zbioru szkoleniowego, jak i walidacyjnego gwałtownie spadają, a następnie wyrównują się, co dowodzi zbieżności modelu. Oznacza to również, że dalsze szkolenie nie przyniesie zauważalnej poprawy

Oprócz oceny modelu na podstawie krzywych strat obliczmy również dokładność na pełnych zbiorach szkoleniowym, walidacyjnym i testowym (podczas szkolenia, za pomocą ustawienia eval_iter = 5, przybliźliśmy dokładność zbioru szkoleniowego i walidacyjnego z pięciu partii):

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Dokładność zbioru szkoleniowego: {train_accuracy*100:.2f}%")
print(f"Dokładność zbioru walidacyjnego: {val_accuracy*100:.2f}%")
print(f"Dokładność zbioru testowego: {test_accuracy*100:.2f}%")
```

Uzyskane wartości dokładności to:

Dokładność zbioru szkoleniowego: 100.00%

Dokładność zbioru walidacyjnego: 96.64%

Dokładność zbioru testowego: 98.00%

Na podstawie tych wyników widać, że model działa dobrze na zbiorach danych szkoleniowym, walidacyjnym i testowym. Model doskonale nauczył się danych szkoleniowych, a dokładność zbioru szkoleniowego wyniosła 100%. Nieco niższa dokładność zbiorów walidacyjnego i testowego (odpowiednio 96,64% i 97,33%) sugeruje pewien stopień nadmiernego dopasowania, ponieważ w porównaniu ze zborem szkoleniowym model nie uogólnia się tak dobrze na niewidocznych danych. Ogólnie rzecz biorąc, uwzględniając to, że dostroiliśmy tylko stosunkowo małą liczbę wag modelu, możemy stwierdzić, że wyniki są imponujące (2,7 miliona wag LoRA zamiast oryginalnych 124 milionów wag modelu).

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
<http://program-partnerski.helion.pl>