WSTĘP DO ARCHITEKTURY TRANSFORMATORÓW

Ta część jest pierwszym z dwóch, które omawiają architekturę transformatorów, będącą podstawą wielu modeli językowych, takich jak BERT i jego warianty , a także rodzinę GPT-x z OpenAI . Opisano główne komponenty oryginalnego transformatora, a także przykłady kodu transformatora w języku Python dla różnych zadań przetwarzania języka naturalnego (NLP), takich jak NER, QnA i zadania wypełniania maski. Pierwsza część rozdziału przedstawia architekturę transformatorów opracowaną przez Google i wydaną pod koniec 2017 roku. W tej sekcji omówiono również niektóre modele oparte na transformatorach. Druga część rozdziału omawia bibliotekę transformatorów firmy Hugging Face, która udostępnia repozytorium ponad 20 000 modeli opartych na transformatorach. Ta sekcja zawiera również przykłady kodu transformatora w Pythonie, które realizują różne zadania przetwarzania języka naturalnego (NLP), takie jak NER, QnA i wypełnianie maski. Trzecia część tego rozdziału zawiera więcej szczegółów dotyczących komponentu kodera, wraz z dodatkowymi informacjami dotyczącymi mechanizmu uwagi, a także komponentu dekodera, który jest drugim głównym komponentem architektury transformatora. Przed przeczytaniem tego rozdziału należy pamiętać, że istnieją przypadki „odwoływania się do przodu", omówione w kolejnych rozdziałach, które zapewniają kontekst dla danego pojęcia. Oczywiście, jeśli nie znasz pojęcia omówionego w kolejnym rozdziale, pełne zrozumienie omawianego tematu również będzie trudne. W rezultacie prawdopodobnie będziesz musiał przeczytać niektóre tematy więcej niż raz, po czym pojęcia staną się łatwiejsze do zrozumienia, gdy zrozumiesz bieżący kontekst i kontekst materiału, do którego się odwołujesz w kolejnych rozdziałach (na szczęście ten proces czytania jest skończony).

MODELE SEKWENCYJNE

Terminy „seq2seq" i „koder-dekoder" są ze sobą ściśle powiązane i często są używane zamiennie w kontekście architektur sieci neuronowych dla zadań przetwarzania sekwencji na sekwencję. W związku z tym można znaleźć artykuły online, w których używane są oba te terminy, a czasami nie ma między nimi rozróżnienia. Krótko mówiąc, model sekwencyjny (nazywany również Seq2Seq lub seq2seq) przetwarza sekwencję wejściową i generuje sekwencję (potencjalnie innego typu). W przypadku architektury transformatorowej proces jednoczesnego trenowania kodera i dekodera tworzy model sekwencyjny na sekwencję. Jeśli uczyłeś się języka obcego, z pewnością wiesz, że rzadko istnieje dosłowne tłumaczenie zdania na inny język. Ponadto, czasami sekwencja wielu słów w jednym języku może mieć tłumaczenie jednosłowne (i odwrotnie). W konsekwencji długość zdania w języku źródłowym może mieć inną długość w języku docelowym. W rezultacie, w 2014 roku Richard Sutskever (i inni) opracowali architekturę sekwencja-do-sekwencji lub Seq2Seq, której głównymi komponentami są koder i dekoder. Ta architektura seq2seq opiera się na dwóch wielowarstwowych LSTM: jeden LSTM mapuje sekwencję wejściową na wektor o stałym rozmiarze (wektor kontekstu), a drugi LSTM dekoduje wektor o stałym rozmiarze, aby wygenerować sekwencję w języku docelowym (jedno słowo na raz). To podejście okazało się na tyle skuteczne, że w 2016 roku Tłumacz Google przeszedł na model seq2seq. Model seq2seq został uogólniony w tym sensie, że może składać się z wielu LSTM, co pozwala również na komponenty Gated Recurrent Unit (GRU). Jednak podstawowa struktura składająca się z komponentu kodera i komponentu dekodera pozostaje nienaruszona. Warto ponownie wspomnieć, że komponent kodera podsumowuje informacje i generuje wektor o stałej długości, który komponent dekodera wykorzystuje do wygenerowania zdania w języku docelowym. Należy pamiętać, że modele seq2seq mogą być używane do przetwarzania mowy, wideo, tekstu, tłumaczenia maszynowego i odpowiedzi na pytania. Co więcej, modele seq2seq doskonale nadają się do generowania tekstu: mogą obsługiwać jeden typ sekwencji jako dane wejściowe, a następnie generować inny typ sekwencji jako dane wyjściowe. Jeśli wytrenowany zostanie tylko jeden komponent (tj. koder lub dekoder), rezultatem jest odpowiednio model autokodujący lub model autoregresyjny (więcej szczegółów na ten

temat znajduje się później). Ponadto długość sekwencji wejściowej może różnić się od długości sekwencji wyjściowej.

Rodzaje modeli seq2seq

Istnieje kilka rodzajów architektur seq2seq, z których niektóre są wymienione tutaj:

• Seq2seq oparte na rekurencyjnej sieci neuronowej (RNN)

• Seq2seq oparte na pamięci długoterminowej (LSTM)

• Seq2seq oparte na uwadze

Przypomnijmy, że RNN wykorzystuje funkcję aktywacji tanh() do przetwarzania sekwencji wejściowej, a także do łączenia wejścia w kroku t z wyjściem z kroku (t-1) w celu wygenerowania wyjścia, które jest wykorzystywane jako wejście dla kroku (t+1). Komórka LSTM to wyspecjalizowany typ sieci neuronowej RNN, który charakteryzuje się następującymi cechami:

• funkcje aktywacji sigmoid() i tanh()

• bramka „zapomnij", bramka wejściowa i bramka wyjściowa

• długoterminowy stan komórki, który jest aktualizowany za pomocą poprzednich bramek

Zauważ, że głębokie sieci neuronowe (DNN) nie znajdują się na poprzedniej liście: wynika to z faktu, że DNN nie mogą być używane do mapowania sekwencji na sekwencje. Można spotkać się z opisem seq2seq jako specyficznego typu sieci neuronowej RNN, często implementowanego przez architekturę koder-dekoder. Jak zobaczysz później, architektura transformatorowa to architektura koder-dekoder, która pierwotnie opierała się na sieciach RNN. Jednak architektura transformatorowa zrezygnowała z sieci RNN i opiera się na mechanizmie uwagi (omówionym później). Innymi słowy, zarówno architektura seq2seq, jak i architektura koder-dekoder obejmują sekwencje wejściowe i wyjściowe. Różnica polega na tym, że wczesne architektury seq2seq opierały się na sieciach neuronowych (RNN), podczas gdy nowsze architektury koder-dekoder preferują uwagę zamiast RNN. Pierwotny transformator to model seq2seq, który składa się z komponentu kodera i komponentu dekodera. Komponent kodera zawiera N warstw, z których każda koduje dane wejściowe do wektora 1x512. Chociaż dekoder również zawiera N warstw dekodera, nie jest on po prostu lustrzanym odbiciem kodera. W rzeczywistości każdy blok dekodera w komponencie dekodera przetwarza dane wejściowe z następujących źródeł:

• dane wyjściowe z całego komponentu kodera

• dane wyjściowe z poprzedniego bloku dekodera

Problemy z przewidywaniem sekwencji

Przewidywanie sekwencji często obejmuje albo 1) prognozowanie następnej wartości w sekwencji o wartościach rzeczywistych, albo 2) wyprowadzanie etykiety klasy dla sekwencji wejściowej. W związku z tym istnieje kilka typów przewidywań sekwencji:

• jeden krok czasowy wejścia do jednego kroku czasowego wyjścia (jeden do jednego)

• wiele kroków czasowych wejścia do jednego kroku czasowego wyjścia (wiele do jednego)

• sekwencja jako wejście i przewidywanie sekwencji jako wyjście

Są to tzw. problemy przewidywania sekwencji-do-sekwencji, w skrócie seq2seq. Należy pamiętać, że długość sekwencji wejściowych i wyjściowych może się różnić. W rzeczywistości, gdy występuje wiele kroków czasowych wejścia i wiele kroków czasowych wyjścia, nazywa się to problemem przewidywania sekwencji typu wiele-do-wielu. Niestety, modele seq2seq mogą być nieefektywne pod względem próbkowania i słabo wypadać w różnych testach porównawczych.

PRZYKŁADY MODELI SEQ2SEQ

Modele sekwencja-do-sekwencji (seq2seq) były wykorzystywane w wielu zastosowaniach w przetwarzaniu języka naturalnego (NLP) i innych dziedzinach. Poniżej przedstawiono kilka przykładów zadań i konkretnych implementacji modeli, które wykorzystują paradygmat seq2seq.

Tłumaczenie maszynowe

Przykład: System tłumaczenia maszynowego neuronowego (GNMT) firmy Google. GNMT to model seq2seq używany przez Tłumacza Google. Wykorzystuje on wiele warstw LSTM do kodowania i dekodowania. Model ten integruje również mechanizm uwagi, aby skupić się na odpowiednich fragmentach zdania źródłowego podczas generowania zdania docelowego.

Podsumowanie tekstu

Przykład: Sieci generatorów wskaźników

W przypadku abstrakcyjnego podsumowania tekstu, gdzie podsumowanie nie jest jedynie wyodrębniane, ale może być nowym tekstem, sieć generatora wskaźników łączy zalety podsumowania ekstrakcyjnego i abstrakcyjnego. Może kopiować słowa z tekstu źródłowego poprzez wskazywanie, a także generować nowe słowa.

Agenci konwersacyjni (chatboty)

Przykład: Początkowe wersje GPT-2 firmy OpenAI do generowania dialogów. Dzięki trenowaniu na zbiorach danych dialogowych, modele seq2seq mogą być używane do generowania odpowiedzi konwersacyjnych.

Opisywanie obrazów

Przykład: Model „Pokaż i opowiedz”

Ten model wykorzystuje splotową sieć neuronową (CNN) jako koder do przetwarzania obrazu oraz RNN jako dekoder do generowania opisowego podpisu. Chociaż koder nie przetwarza sekwencji, dekoder ją generuje, model nadal podąża za paradygmatem seq2seq. Rozpoznawanie mowy

Przykład: Słuchaj, Uważnie i Przeliteruj (LAS)

LAS wykorzystuje model seq2seq do transkrypcji języka mówionego na tekst. Koder przetwarza klatki audio, a dekoder, z mechanizmem uwagi, generuje odpowiednią transkrypcję tekstową.

Napisy wideo

Przykład: Modele wideo-tekst

Podobne do napisów do obrazów, ale koder przetwarza sekwencję klatek (wideo), aby wygenerować opis tekstowy lub napisy dla każdej klatki lub segmentu.

Projektowanie molekularne

Przykład: Modele do odkrywania leków

Modele seq2seq mogą być używane do tłumaczenia między różnymi reprezentacjami cząsteczek lub przewidywania właściwości molekularnych, co wspomaga odkrywanie i projektowanie leków.

Prognozowanie szeregów czasowych

Przykład: Prognozowanie cen akcji

Wykorzystanie historycznych cen akcji (sekwencji cen) do przewidywania cen w przyszłości.

Tłumaczenie programów

Przykład: Modele translacji kodu

Tłumaczenie kodu z jednego języka programowania na inny.

Generowanie muzyki

Przykład: Modele generowania melodii

Na podstawie sekwencji nut, wygeneruj kontynuację lub sekwencję harmonizującą. Powyższe przykłady pokazują wszechstronność architektury seq2seq, czyniąc ją fundamentalnym narzędziem w wielu obszarach uczenia maszynowego i sztucznej inteligencji.

CO Z RNN I LSTM?

Dane wejściowe dla RNN mają tę samą długość: krótsze zdania są uzupełniane (po lewej lub prawej stronie) do określonej długości, a zdania są obcinane, jeśli są dłuższe niż ta określona długość. Jednak modele kodera/dekodera generują kodowanie danych wejściowych przed przekazaniem go do dekodera w celu przetworzenia. W rezultacie nie ma ograniczeń co do długości zdania wejściowego: model kodera/dekodera może przetwarzać zdania wejściowe o zmiennej i różnej długości, a także generować zdania o różnej długości jako dane wyjściowe. Architektura Encoder-Decoder LSTM składa się z dwóch modeli: jeden model odczytuje sekwencję wejściową i koduje ją do wektora o stałej długości, a następnie drugi model dekoduje wektor o stałej długości i również generuje przewidywaną sekwencję. Encoder-Decoder LSTM został opracowany z myślą o problemach z przetwarzaniem języka naturalnego, gdzie osiągnął wydajność SOTA w tłumaczeniu tekstu. Analogicznie, architektura kodera-dekodera RNN składa się z dwóch sieci RNN, gdzie koder odwzorowuje sekwencję źródłową o zmiennej długości na wektor o stałej długości, a dekoder odwzorowuje reprezentację wektorową z powrotem na sekwencję wyjściową o zmiennej długości.

LSTM i ich warstwa końcowa

Model oparty na LSTM składa się z wielu komórek LSTM, gdzie dane wyjściowe jednej komórki LSTM są traktowane jako „wektor kontekstu", który staje się danymi wejściowymi dla kolejnej komórki LSTM. Jednakże dane wyjściowe pośrednich komórek ukrytych LSTM są odrzucane: do kolejnej komórki LSTM przekazywane jest tylko dane wyjściowe komórki końcowej (czyli „stan komórki") w danej sieci LSTM. Dlaczego LSTM odrzuca te inne pośrednie dane wyjściowe? Jednym z powodów może być to, że końcowy stan komórki zawiera skumulowane informacje zgromadzone w wyniku przetwarzania każdej z poprzednich warstw ukrytych, a zatem zawartość końcowego stanu komórki jest zasadniczo nadzbiór wszystkich poprzednich komórek. Innymi słowy, wszystko jest „zwinięte" w skrajnie prawej warstwie końcowej, więc poprzednie warstwy pośrednie nie dostarczają żadnych dodatkowych informacji. Chociaż powyższa logika jest prawdopodobna, w rzeczywistości nie jest prawdziwa, co wyraźnie pokazuje mechanizm uwagi, który uwzględnia dane wyjściowe ze wszystkich warstw ukrytych.

Przetwarzanie w LSTM jest sekwencyjne i nie może być wykonywane równolegle ze względu na zależności między sąsiednimi warstwami ukrytymi. Natomiast mechanizm uwagi można zrównoleglić, ponieważ obliczane jest podobieństwo między każdym słowem w zdaniu a każdym słowem w zdaniu, a zatem nie występują zależności sekwencyjne.

Stan ukryty w sieciach LSTM a architektura transformatorowa

W sieciach RNN i LSTM stan ukryty jest generowany jako dane wyjściowe w każdej warstwie sieci RNN i LSTM. Co więcej, tylko ostatni (tj. skrajnie prawy) stan ukryty jest przekazywany do następnego komponentu (czy to sieci RNN, czy LSTM) w sekwencji komponentów. Poprzednia technika ignoruje dane wyjściowe wszystkich warstw poza ostatnią w sieci RNN lub LSTM. Chociaż można by pomyśleć, że ta technika ignoruje ważne informacje, można ją uznać za rozsądną decyzję projektową z następującego powodu: ostatnia warstwa każdego komponentu zawiera informacje z poprzednich warstw „zwinięte" w warstwę końcową. Tak czy inaczej, okazuje się, że dane wyjściowe dla wszystkich stanów ukrytych są przydatne, a ich przechwytywanie jest kluczowym aspektem implementacji mechanizmu uwagi. Nawiasem mówiąc, możesz być zaskoczony, odkrywając, że wczesne wersje TensorFlow 1.x obsługiwały mechanizm uwagi, który poprzedzał popularność architektury transformatorowej. Jednak zanim ludzie dowiedzieli się o takim wsparciu, komponenty RNN i LSTM architektury transformatorowej nie były już uważane za niezbędne.

MODELE KODERA/DEKODERA

Modele koder-dekoder zostały zaprojektowane do rozwiązywania zadań sekwencyjnych (sekwencja-sekwencja). Takie modele stanowią podstawę dla innych modeli i architektur, takich jak modele uwagi, transformatory, modele BERT i GPT. Co ciekawe, BERT to architektura oparta wyłącznie na koderze. Struktura koder-dekoder to paradygmat projektowania sieci neuronowych używany w zadaniach sekwencyjno-sekwencyjnych, gdzie dane wejściowe są sekwencją, a dane wyjściowe inną sekwencją. Architektura ta składa się z dwóch głównych komponentów: kodera i dekodera. Koder przetwarza sekwencję wejściową i kompresuje jej informacje do wektora kontekstu o stałym rozmiarze, często nazywanego „stanem ukrytym" lub „kontekstem". W wielu modelach kontekst ten reprezentuje ostatni ukryty stan kodera. Zazwyczaj w tym celu wykorzystuje się sieci neuronowe RNN lub ich warianty, takie jak LSTM i GRU. Dekoder pobiera wektor kontekstu wygenerowany przez koder i generuje z niego sekwencję wyjściową. Zazwyczaj jest to sieć RNN, LSTM lub GRU, która generuje sekwencję wyjściową krok po kroku, często w oparciu o wynik z poprzedniego kroku i kontekst z kodera. Dekoder jest często wyposażony w mechanizmy uwagi, które pozwalają mu skupić się na różnych częściach sekwencji wejściowej na każdym etapie generowania danych wyjściowych, co poprawia wydajność, szczególnie w przypadku dłuższych sekwencji.

Relacja między seq2seq a koderem/dekoderem

Model seq2seq jest zasadniczo implementacją architektury koder-dekoder. Struktura seq2seq opiera się na paradygmacie koder-dekoder, w którym koder przetwarza sekwencję źródłową, a dekoder generuje sekwencję docelową. Kiedy mowa o modelach seq2seq, często mamy na myśli modele wykorzystujące koder do przetwarzania sekwencji źródłowej i dekoder do generowania sekwencji docelowej. Termin „seq2seq" zazwyczaj odnosi się do modeli, które podążają za tą strukturą koder-dekoder. Poniższe stwierdzenie podsumowuje relację między modelami seq2seq a modelami koder-dekoder: wszystkie modele seq2seq są modelami koder-dekoder, ale nie wszystkie modele koder-dekoder są koniecznie wykorzystywane w zadaniach seq2seq (chociaż większość tak). Chociaż wszystkie modele seq2seq opierają się na architekturze koder-dekoder, struktura koder-dekoder może być wykorzystywana w zadaniach, które nie są ściśle sekwencyjne. Na przykład, podpisy do obrazów wykorzystują obraz jako dane wejściowe (zakodowane przez CNN) i sekwencję słów jako dane

wyjściowe (generowane przez RNN lub LSTM), co pasuje do schematu koder-dekoder, ale nie do typowego paradygmatu seq2seq, w którym zarówno dane wejściowe, jak i wyjściowe są sekwencjami.

PRZYKŁADY MODELI KODERA/DEKODERA

Architektura kodera-dekodera to fundamentalna koncepcja leżąca u podstaw wielu modeli, szczególnie w zadaniach przetwarzania sekwencji z jednej na drugą. Oto kilka przykładów zadań i konkretnych implementacji wykorzystujących strukturę kodera-dekodera.

Tłumaczenie maszynowe

Przykład: Przetwarzanie sekwencji z uwzględnieniem uwagi

Jest to jeden z pionierskich modeli w neuronowym tłumaczeniu maszynowym. Koder przetwarza język źródłowy, a dekoder generuje język docelowy. Mechanizm uwagi pozwala dekoderowi skupić się na różnych częściach zdania źródłowego podczas tłumaczenia.

Podpisy do obrazów

Przykład: Pokaż, Uczestnicz i Powiedz

Koder, zazwyczaj CNN, przetwarza obraz. Dekoder, zazwyczaj RNN lub LSTM, generuje podpis. Mechanizm uwagi pozwala dekoderowi skupić się na różnych obszarach obrazu podczas generowania podpisu.

Rozpoznawanie mowy

Przykład: DeepSpeech 2 firmy Baidu

Koder przetwarza sygnał audio, a dekoder generuje odpowiednią transkrypcję tekstową.

Podsumowanie tekstu

Przykład: Modele podsumowania abstrakcyjnego

Koder przetwarza oryginalny, długi tekst, a dekoder generuje zwięzłe podsumowanie. Wiele z tych modeli integruje równieżmechanizmy uwagi.

Napisy wideo

Przykład: Wideo na tekst z sekwencji do sekwencji

Koder przetwarza sekwencje klatek wideo, a dekoder generuje opis lub podpis dla każdej klatki lub segmentu.

Rozpoznawanie jednostek nazwanych (z wykorzystaniem seq2seq)

Przykład: Modele traktujące NER jako zadanie seq2seq

Koder przetwarza sekwencję słów, a dekoder generuje sekwencję znaczników encji.

Tłumaczenie obrazu na obraz

Przykład: Pix2Pix

Ten model wykorzystuje architekturę koder-dekoder do zadań takich jak konwersja zdjęć satelitarnych na mapy lub czarno-białych zdjęć na kolorowe.

Tłumaczenie muzyki

Przykład: Modele tłumaczące brzmienie jednego instrumentu na inny

Wykrywanie anomalii w szeregach czasowych

Przykład: Modele oparte na autokoderze do wykrywania anomalii

Koder przetwarza sekwencję punktów danych, aby utworzyć skompresowaną reprezentację, a dekoder rekonstruuje tę sekwencję. Duże błędy rekonstrukcji mogą wskazywać na anomalie.

Segmentacja semantyczna w obrazach

Przykład: Architektura U-Net

Koder, używany do segmentacji obrazów biomedycznych, przechwytuje kontekst obrazu, a dekoder następnie wykorzystuje ten kontekst do wygenerowania mapy segmentacyjnej o wysokiej rozdzielczości.Powyższe przykłady podkreślają elastyczność i szerokie zastosowanie architektury koder-dekoder w różnych dziedzinach uczenia maszynowego i sztucznej inteligencji. Architektura koder-dekoder to fundamentalna koncepcja leżąca u podstaw wielu modeli, zwłaszcza w zadaniach sekwencyjnych. W kolejnych sekcjach przedstawiono przykłady zadań i konkretnych implementacji wykorzystujących architekturę koder-dekoder.

MODELE AUTOREGRESYWNE

Modele autoregresyjne oparte na architekturze transformatora to architektury wyłącznie dekoderowe. Modele autoregresyjne wykorzystują poprzednie prognozy do generowania nowych. W związku z tym ich trenowanie wiąże się z zadaniem modelowania języka: modele muszą nauczyć się języka i współzależności między słowami i frazami, w tym semantyki. Modele autoregresyjne przewidują wartość dla następnego kroku czasowego w sekwencji na podstawie wyników z poprzednich kroków czasowych.

Modele autoregresyjne są wstępnie trenowane z istotnym ograniczeniem: widzą tylko tokeny poprzedzające bieżący token i nie widzą kolejnych tokenów. Ten mechanizm „wymusza” na modelach autoregresyjnych przewidywanie kolejnego tokena bez faktycznego jego sprawdzenia. To ograniczenie osiąga się poprzez ustawienie wartości w górnej trójkątnej części macierzy uwagi na 0. Autoregresyjne (AR) modele LLM dobrze nadają się do generatywnych zadań przetwarzania języka naturalnego (NLP). Chociaż modele AR LLM mogą pracować zarówno w kontekście do przodu, jak i wstecz, nie mogą jednocześnie używać kontekstu do przodu i wstecz. Jak zobaczysz w rozdziale 6, autoregresyjny model XLNet wykorzystuje permutacyjne modelowanie języka w fazie wstępnego treningu, co pozwala XLNet zbierać informacje z obu stron każdego tokena w sekwencji. Jednak ten typ modelu ma dwa istotne ograniczenia. Pierwsze ograniczenie dotyczy predykcji słów w danym kroku czasowym: model autoregresyjny opiera się na słowie poprzedzającym i następującym po słowie bieżącym. Drugim ograniczeniem jest fakt, że dokładność przewidywanego słowa zależy od dokładności przewidywania słów poprzednich w sekwencji.

Transformatory autoregresyjne

„Normalny” dekoder transformatorowy jest autoregresyjny w momencie wnioskowania i nieautoregresyjny w momencie uczenia. W konsekwencji model transformatorowy jest autoregresyjny, ponieważ jego dekoder jest autoregresyjny. Istnieją jednak nieautoregresywne warianty transformatora: stanowią one raczej przedmiot badań niż gotowe rozwiązania.

Jak przyspieszyć transformatory autoregresyjne

W poniższym artykule autorzy twierdzą, że mogą sprawić, że modele oparte na transformatorach autoregresyjnych będą dwa razy szybsze niż odpowiadające im modele na Hugging Face:

https://www.reddit.com/r/MachineLearning/comments/uwkpmt/p_what_we_learned_by_making_t5large_2x_faster

Autorzy opisują trzy techniki, które podwoiły wydajność modelu T5:

1. Przechowywanie 2 grafów obliczeniowych w jednym pliku Onnx

2. Wykonywanie kopii zerowej w celu pobrania danych wyjściowych ze środowiska wykonawczego Onnx

3. Uniwersalne narzędzie do konwersji dowolnego modelu (niezależnie od architektury) do FP16

Należy pamiętać o poniższych komentarzach i zaleceniach, parafrazując poprzedni artykuł na Reddicie:

1. Technika podwójnego wykresu ma zastosowanie tylko w modelach autoregresyjnych.

2. Technika FP16 nie jest przydatna w modelach typu BERT.

3. Użyj tensorrt w modelach typu BERT i zastosuj kwantyzację int8, co może skutkować 5- do 10-krotnym wzrostem wydajności odpowiednio dla sekwencji umiarkowanie długich i bardzo krótkich.

Przypomnijmy, że T5 opiera się zarówno na komponencie kodera, jak i dekodera architektury transformatora, co oznacza, że T5 jest również modelem autoregresyjnym.

Autorzy udostępniają również notatnik (tylko dla środowiska uruchomieniowego Onnx):

https://github.com/ELS-RD/transformer-deploy/blob/main/demo/generative-model/t5.ipynb

Istnieje repozytorium Github zawierające poprzedni notatnik:

https://github.com/ELS-RD/transformer-deploy

Następująca strona internetowa zawiera link do tego samego notatnika w nbviewer:

https://nbviewer.jupyter.org/url/github.com/ELS-RD/transformer-

deploy/blob/main/demo/generative-model/t5.ipynb

Jeśli chcesz wykonać kod z notatnika, poniższy link do bindera uruchomi aplikację Jupyter Serwer:

https://mybinder.org/v2/gh/ELS-RD/transformer-deploy/main?filepath=demo%2Fgenerative-model%2Ft5.ipynb

MODELE AUTOENKODOWANIA

Modele autoregresyjne (omówione wcześniej w tym rozdziale) są bardzo dobre, gdy celem jest generowanie języka naturalnego (NLG). Jednak zadania związane z rozumieniem języka naturalnego (NLU) nie są dobrze dostosowane do modeli autoregresyjnych ani do modeli seq2seq; te ostatnie nadają się do tłumaczenia języka. Lepszym wyborem dla NLU jest model autokodowania. Autoenkodery uczą się reprezentacji (tj. kodowania) zbioru danych, najczęściej w celu redukcji wymiarowości, tak aby warstwa wejściowa i warstwa końcowa były takie same. Mówiąc w uproszczeniu, modele autoenkodowania są wstępnie trenowane poprzez zniekształcenie tokenów wejściowych w określony sposób, a następnie próbę rekonstrukcji zdania oryginalnego. Na przykład model BERT „koryguje” sekwencję wejściową, zastępując podzbiór tokenów tokenem [MASK]. Do

pewnego stopnia modele autoenkodowania próbują pokonać ograniczenia modeli autoregresyjnych. Modele autoenkodowania opierają się na oryginalnym transformatorze i badają dwustronne zależności tokenów w zdaniu, aby uzyskać kontekst dwukierunkowy. Ograniczenie modeli autoenkoderów wynika z kroku MLM wykonywanego na etapie wstępnego trenowania. Ponieważ MLM próbuje przewidzieć zamaskowany token podczas MLM, ograniczenie pojawia się, gdy wiele zamaskowanych tokenów jest od siebie zależnych na tym etapie predykcji. Drugie ograniczenie pojawia się w przypadku modeli (takich jak BERT), które przetwarzają sekwencje o stałej długości i maksymalnej dopuszczalnej długości (która dla BERT wynosi 512).

Autoenkodery

Autoenkoder to w pełni połączona prosta sieć neuronowa, będąca jednym z typów modeli koder-dekoder, w których warstwa wejściowa i wyjściowa są takie same, a zazwyczaj między warstwą wejściową a wyjściową znajduje się jedna warstwa ukryta, mniejsza niż warstwy wejściowa i wyjściowa. Celem warstwy ukrytej jest redukcja wymiarowości: po wytrenowaniu modelu zawartość warstwy ukrytej jest zachowywana dla innych zadań (takich jak QnA), podczas gdy warstwy wejściowa i wyjściowa są pomijane. Autoenkodery to typ sieci neuronowej wykorzystywany głównie do zadań uczenia bez nadzoru, w szczególności redukcji wymiarowości i uczenia cech. Autoenkoder składa się z dwóch głównych części: kodera i dekodera. Koder kompresuje dane wejściowe do reprezentacji przestrzeni ukrytej, a dekoder rekonstruuje dane wejściowe na podstawie tej reprezentacji. Głównym celem autoenkodera jest minimalizacja różnic między oryginalnym danymi wejściowymi a ich rekonstrukcją, co pozwala na uzyskanie zwartej i wydajnej reprezentacji danych. Istnieją odmiany tej architektury, które zawierają dodatkowe, ukryte warstwy, z których niektóre są bardziej złożone niż warstwy wejściowa i wyjściowa. Wyszukaj w internecie hasło „image-autoencoders”, a znajdziesz mnóstwo interesujących typów autoenkoderów.

Transformatory autokodujące

Transformator autokodujący to sieć neuronowa, która łączy możliwości uczenia się bez nadzoru autoenkoderów z potężnymi możliwościami przetwarzania sekwencji transformatorów. Został zaprojektowany do kodowania sekwencji wejściowych do ukrytych reprezentacji za pomocą mechanizmów transformatorów, a następnie ich dekodowania z powrotem, dążąc do minimalizacji błędu rekonstrukcji. Ta architektura może uczyć się zwartych i sensownych reprezentacji danych sekwencyjnych, co czyni ją użyteczną w wielu zadaniach w przetwarzaniu języka naturalnego i nie tylko. Transformator autokodujący łączy zasady działania autoenkoderów i transformatorów. Wykorzystuje architekturę transformatorów do kodowania danych wejściowych do ukrytej reprezentacji, a następnie dekodowania ich z powrotem do pierwotnej postaci. Transformator autokodujący może być używany do różnych zadań, takich jak:

• Uczenie się reprezentacji: Aby nauczyć się sensownych reprezentacji danych wejściowych

• Odszumianie: Aby zrekonstruować czyste dane z zaszumionych danych wejściowych

• Wykrywanie anomalii: Duże błędy rekonstrukcji mogą wskazywać na anomalie lub wartości odstające w danych

Na przykład model BERT (Bidirectional Encoder Representations from Transformers) można postrzegać jako rodzaj transformatora autokodującego. BERT jest trenowany w celu przewidywania zamaskowanych fragmentów tekstu, kodując w ten sposób otaczający kontekst w swoich osadzeniach. Chociaż nie rekonstruuje całego wejścia jak tradycyjne autokodery, jego zasada uczenia jest podobna, ponieważ uczy się reprezentacji na podstawie rekonstrukcji części wejścia. Oto ważna uwaga: wybór

między modelami autoregresyjnymi a modelami autokodowania zależy od zadania i treningu, a nie od architektury. Co więcej, możliwe jest użycie dekodera do autokodowania i możliwe jest użycie kodera do zadań autoregresyjnych.

ARCHITEKTURA TRANSFORMATORA: WSTĘP

Architektura transformatora zadebiutowała w 2017 roku i została zaprojektowana jako architektura kodera-dekodera, będąca modelem sekwencyjnym (SEQ2SEQ). Transformer wywarł ogromny wpływ na modelowanie języka, częściowo dzięki mechanizmowi samouwagi, który śledzi wzajemne powiązania tokenów. Architektura transformatorowa wykonuje zadania związane z przetwarzaniem języka naturalnego, które zasadniczo różnią się od innych architektur. W rzeczywistości architektura transformatorowa umożliwiła rozwój wielu znanych modeli dużego języka (LLM), takich jak BERT, T5, GPT-2 i GPT-3.

Oryginalna architektura transformatora

Możesz być zaskoczony, odkrywając, że oryginalna architektura transformatora była początkowo modelem seq2seq, który obejmował LSTM. Ktoś odkrył, że architektura transformatora działała równie dobrze bez LSTM dzięki mechanizmowi uwagi, co z kolei doprowadziło do powstania przełomowego artykułu „Uwaga to wszystko, czego potrzebujesz". Interesującym rozwinięciem architektury transformatora jest to, że istnieje kilka jej typów:

• Architektura tylko z koderem (np. modele oparte na BERT)

• Architektura tylko z dekoderem (np. GPT-x)

• Zarówno architektura kodera, jak i dekodera (np. T5 i BART)

Dlatego, poznając modele LLM oparte na transformatorach, warto zadać sobie pytanie, jaki typ architektury transformatora jest używany w modelu LLM. Przykładami modeli obejmujących zarówno koder, jak i dekoder są modele seq2seq, BART i model T5 (Google), które można precyzyjnie dostroić do wykonywania wielu zadań. Słowo „koder" jest przeciążone: koder może odnosić się do całego komponentu kodera lub do jednego z 6 (lub 12) koderów znajdujących się w całym komponencie kodera (podobne uwagi dotyczą dekodera). W związku z tym w tej książce używane są następujące terminy w celu ujednoznacznienia słów „koder" i „dekoder":

• Komponent to pełny koder lub dekoder oryginalnej architektury transformatora.

• Blok zawiera warstwy, które realizują funkcję uwagi, funkcję przekazywania dalej itd.

• Warstwa to element (w bloku), który realizuje funkcję uwagi, funkcję przekazywania dalej itd.

Architektury oparte na transformatorach zazwyczaj mają albo 6 bloków kodera i 6 bloków dekodera odpowiednio dla komponentu kodera i komponentu dekodera, albo 12 bloków kodera dla komponentu kodera i 12 bloków dekodera dla komponentu dekodera (możliwe są również inne wartości). Należy pamiętać, że blok w komponencie kodera jest po prostu synonimem samodzielnego, mniejszego kodera; podobnie blok w komponencie dekodera jest w rzeczywistości samodzielnym, mniejszym dekoderem. Na razie zwróć uwagę, że każdy blok w komponencie kodera zawiera dwie warstwy, podczas gdy jeden blok w komponencie dekodera zawiera trzy warstwy. Jak zobaczysz w rozdziale 6, BERT wykorzystuje tylko komponent kodera z transformatora, podczas gdy GPT-2 obejmuje tylko komponent dekodera z transformatora, a T5 wykorzystuje zarówno koder, jak i dekoder.

Kolejność operacji

Jeśli model oparty na transformatorach składa się z jednego komponentu, wszystkie operacje są wykonywane w tym komponencie w sposób oddolny. Jeśli model oparty na transformatorze obejmuje zarówno komponent enkodera, jak i dekodera, operacje są wykonywane w sposób oddolny w komponencie enkodera, a następnie jego dane wyjściowe są przekazywane do komponentu dekodera, gdzie kolejne operacje są wykonywane w sposób oddolny.

Niezależność od sieci RNN/LSTM

Co ciekawe, początkowa implementacja architektury transformatora obejmowała sieci RNN i LSTM. W pewnym momencie ktoś odkrył, że porównywalna dokładność jest możliwa bez sieci RNN i LSTM. To odkrycie doprowadziło do sformułowania „Uwaga to wszystko, czego potrzebujesz" w przełomowym artykule poświęconym architekturze transformatora. W szczególności architektura transformatora różni się od innych architektur w następujących istotnych kwestiach:

• Opiera się na mechanizmie uwagi.

• Uczenie modelu może być zrównoleglone.

• Nie są wymagane sieci CNN/RNN/LSTM.

Należy zauważyć, że model koder-dekoder różni się od modelu seq2seq, który często zawiera sieci RNN lub LSTM. Niezależność od sieci RNN i LSTM upraszcza architekturę transformatora, która opiera się na samouwadze, co oznacza, że każde słowo w zdaniu jest porównywane z każdym innym słowem w tym samym zdaniu. To „globalne" podejście do przetwarzania dokumentów wykazało lepsze wyniki niż inne techniki przetwarzające „lokalne" fragmenty dokumentu. Co więcej, architektura transformatorowa jest lepiej przygotowana do przetwarzania zależności dalekiego zasięgu w korpusie. Sieci neuronowe (RNN) okazały się przydatne w rozwiązywaniu różnorodnych zadań, w tym zadań opartych na przetwarzaniu języka naturalnego (NLP). W szczególności systemy tłumaczenia maszynowego, które mapują sekwencje słów z języka źródłowego na język docelowy, opierały się na RNN. W rzeczywistości zadania tłumaczeniowe mogą być obsługiwane przez architekturę koder-dekoder, a także architekturę seq2seq (sekwencja-do-sekwencji). Architektura transformatorowa została pierwotnie zaprojektowana dla modeli tłumaczenia maszynowego, dlatego została ustrukturyzowana jako architektura koder/dekoder. W rzeczywistości architektura transformatorowa okazała się niezwykle skuteczna w przypadku modeli językowych. Po opublikowaniu początkowej architektury transformatorowej, badacze opracowali architektury, które są bardziej wydajne pod względem obliczeń, a także zużycia pamięci, co jest szczególnie ważne w przypadku małych urządzeń (takich jak telefony komórkowe) i przetwarzania obrazów binarnych. Rzeczywiście, transformatory wymagają znacznej złożoności obliczeniowej, przekraczającej możliwości urządzeń takich jak smartfony.

Bloki komponentów transformatora

Ogólnie rzecz biorąc, używamy litery T do wskazania liczby tokenów przekazywanych do początkowej warstwy osadzania. Wyjściem początkowej warstwy osadzania jest macierz wartości Txd, gdzie d jest szerokością wejściową modelu. Wartość d w artykule dotyczącym transformatorów wynosi 512, a w innych modelach d może wynosić 1024 lub 2048. Bloki transformatorów wykonują różne operacje bez zmiany kształtu danych (tj. Txd). Oto ważna kwestia: kodowania pozycyjne i dane wyjściowe z bloków transformatora w komponencie kodera lub dekodera mają kształt Txd (T i d są zdefiniowane w poprzednim akapicie).

Główne komponenty transformatora

Jak dowiedziałeś się w poprzedniej sekcji, kompletna architektura transformatora obejmuje mechanizm uwagi (omówiony w rozdziale 1), kodowania pozycyjne (omówione później), koder i dekoder w celu obsługi zadań sekwencyjnych. Mechanizm uwagi może składać się z następujących typów:

• Uwaga w koderze

• Zamaskowana uwaga w dekoderze

• Uwaga między koderem a dekoderem

Pierwszy krok polega na osadzeniu danych wejściowych, które obejmują macierz (zwaną macierzą osadzania pozycji). Zatem dane wejściowe składają się z wektora z wieloma tokenami. Kolejnym interesującym szczegółem dotyczącym modeli opartych na transformatorach jest to, że niektóre z nich wykorzystują tylko koder (BERT), inne tylko dekoder (GPT-3), a jeszcze inne zarówno koder, jak i dekoder (T5). Każda z tych trzech architektur nadaje się do różnych typów zadań przetwarzania języka naturalnego. Poniżej znajduje się podsumowanie trzech ważnych aspektów architektury transformatorów, z których dwa zostaną omówione bardziej szczegółowo w dalszej części tego rozdziału: Mechanizm uwagi to operacja sekwencyjna, która przyjmuje serię wektorów jako dane wejściowe i generuje inną sekwencję wektorów. Niektóre obliczenia uwagi to po prostu średnia ważona wszystkich wektorów wejściowych. Komponent kodera zawiera sześć bloków, z których każdy zawiera kilka warstw, a te z kolei zawierają dwie podwarstwy: pierwsza podwarstwa zawiera mechanizm samouwagi, a druga podwarstwa składa się z sieci sprzężenia zwrotnego. Komponent dekodera składa się z sześciu bloków, z których każdy zawiera kilka warstw, a te z kolei trzy podwarstwy: dwie z tych podwarstw są takie same jak ich odpowiedniki w koderze, a trzecia warstwa to wielogłowicowa warstwa uwagi do transformacji sygnału wejściowego kodera. Należy zwrócić uwagę na fakt, że chociaż koder i dekoder są bardzo podobnymi komponentami, blok dekodera zawiera dodatkową podwarstwę składającą się z wielogłowicowej warstwy uwagi.

TRANSFORMER JEST MODELEM KODER/DEKODER

Architektura transformatora ma dwie istotne zalety w porównaniu z innymi architekturami:

• niższa złożoność obliczeniowa

• wyższa łączność, co jest szczególnie korzystne w sekwencjach obejmujących zależności długoterminowe

Ponadto koder i dekoder są trenowane w celu minimalizacji warunkowego logarytmu prawdopodobieństwa. Jak zobaczysz , komponent kodera w oryginalnym artykule o transformatorze (dostępnym w arxiv) ma 6 identycznych warstw. Co więcej, niektóre LLM-y składają się z 12 identycznych warstw zamiast 6. Każda warstwa składa się z następujących elementów:

• Uwaga wielogłowicowa

• Dodawanie i normalizacja

• Sieć feed-forward

• Dodawanie i normalizacja

Omawiając koder lub dekoder, pomocne jest zwizualizowanie kodera jako pionowego stosu elementów kodera, umieszczonego po lewej stronie komponentu dekodera na schemacie architektury transformatora, który z kolei również składa się z pionowego stosu elementów dekodera. Komponent

dekodera przypomina rozszerzenie komponentu kodera, które zawiera trzecią warstwę dla „uwagi maskowanej”, jak wymieniono tutaj:

• Uwagę maskowaną wielogłowicową

• Dodawanie i normalizacja

• Uwagę wielogłowicową

• Dodawanie i normalizacja

• Sieć feed-forward

• Dodawanie i normalizacja

Uwagę maskowaną wielogłowicową w komponencie dekodera również wykonuje maskowanie, aby zapobiec „wybieganiu w przyszłość” w kierunku przyszłych tokenów. Komponent dekodera może również składać się z 6 identycznych warstw lub 12 identycznych warstw, które odpowiadają liczbie elementów w komponencie kodera. Przypomnijmy, że w przypadku sekwencji elementów LSTM, ostatnia ukryta warstwa danego LSTM jest przekazywana tylko do jednego LSTM, który jest kolejnym LSTM w sekwencji. Zatem istnieje relacja jeden do jednego między wyjściem jednego LSTM a wejściem dla kolejnego LSTM w sekwencji. W tym tkwi istotna różnica dotycząca sposobu przekazywania wyjścia z jednego elementu do następnego w sekwencji elementów. W przeciwieństwie do sekwencji elementów opartej na LSTM, wyjście komponentu kodera jest przekazywane do każdej zamaskowanej warstwy uwagi w każdym elemencie dekodera. Co więcej, ta trzecia warstwa dekodera akceptuje wejście nie tylko z wyjścia komponentu kodera, ale także z niższej warstwy dekodera w pionowym stosie elementów w komponencie dekodera. Zatem, jeśli komponenty kodera i dekodera składają się z N elementów, to wyjście kodera jest przekazywane do każdego z N elementów w dekoderze. W pewnym sensie, wyjście z komponentu kodera do komponentu dekodera jest relacją jeden do wielu.

PRZEPŁYW TRANSFORMATORA I JEGO WARIANTY

Dane wejściowe do transformatora składają się z zestawu tokenów, których maksymalna długość wynosi zazwyczaj 512 lub 1024, a długość powiązanych wektorów wejściowych nazywana jest rozmiarem kontekstu. Zdania dłuższe niż maksymalna długość muszą zostać skrócone, co może prowadzić do utraty dokładności, jeśli skrócone fragmenty zawierają ważne informacje. Co ważne, transformator koduje te wektory równolegle za pomocą mechanizmu uwagi wielogłowicowej (omówionego później). Natomiast sieci neuronowe RNN i LSTM przetwarzają tokeny sekwencyjnie. Po zakodowaniu tokenów jako wektorów jednowymiarowych i przetworzeniu ich przez komponent kodera, wynik jest wysyłany do dekodera w celu wygenerowania sekwencji wyjściowej. Należy zauważyć, że dekoder przetwarza dane wejściowe sekwencyjnie, a nie równolegle. Później dowiesz się o modelach wykorzystujących tylko koder (np. BERT), innych modelach wykorzystujących tylko dekoder (np. GPT-3) oraz niektórych modelach wykorzystujących zarówno koder, jak i dekoder (np. BART i T5). Oto podsumowanie dotychczasowej wiedzy na temat architektury transformatorowej:

• Opiera się na mechanizmie uwagi.

• Uczenie modelu może być zrównoleglone.

• Nie są wymagane żadne CNN/RNN/LSTM.

• Niektóre modele wykorzystują tylko koder (np. BERT).

• Niektóre modele wykorzystują tylko dekoder (np. GPT-3).

• Niektóre modele wykorzystują zarówno koder, jak i dekoder (np. BART i T5).

• Maksymalna długość danych wejściowych to 512.

Architektura transformatorowa była wykorzystywana w zadaniach z innych dziedzin niezwiązanych z przetwarzaniem języka naturalnego, takich jak chemia, widzenie komputerowe, klasyfikacja obrazów i wideo (między innymi). Modele LLM oparte na transformatorach doprowadziły do opracowania wstępnie wytrenowanych modeli, które są trenowane na dużym i nieoznakowanym korpusie. Co więcej, takie modele można dostroić do wyspecjalizowanych zadań „z dołu" poprzez dołączenie do modeli warstwy specyficznej dla danego zadania, a następnie przeprowadzenie dodatkowego treningu w oparciu o stosunkowo niewielką liczbę oznaczonych przykładów.

Trzy warianty architektury transformatorowej

W poprzedniej sekcji wskazano, że architektura transformatorowa obejmuje połączenie autoattention, enkodera i dekodera. Co ciekawe, istnieją modele, które implementują różne podzbiory (komponenty) architektury transformatorowej. Chociaż wszystkie z nich oferują mechanizm auto-uwagi, różnią się one w następujący sposób:

• Tylko koder

• Tylko dekoder

• Koder i dekoder (oryginalna architektura)

Jak dowiedzieliście się wcześniej w tym rozdziale, GPT wykorzystuje tylko komponent dekodera architektury transformatorowej, podczas gdy BERT korzysta tylko z komponentu kodera. BART i T5 to dwa modele, które wykorzystują zarówno koder, jak i dekoder architektury transformatorowej. Dodatkowe informacje na temat trzech różnych implementacji architektury transformatorowej znajdują się w dalszej części tego rozdziału.

Modele NLP oparte na transformatorach

Oprócz posiadania różnych kombinacji komponentów najwyższego poziomu architektury transformatorowej, modele NLP oparte na transformatorach można podzielić na trzy główne typy, których przykłady podano w nawiasach:

• Seq2seq (BART/T5)

• Autokodowanie (BERT)

• Autoregresja (GPT)

Poprzednie modele transformatorowe zostały wytrenowane jako modele językowe na dużych ilościach tekstu w sposób samonadzorowany. Ponadto XGLM to rodzina wielkoskalowych, wielojęzycznych, autoregresyjnych modeli językowych, które osiągają wyniki SOTA w wielojęzycznym uczeniu się z niewielką liczbą powtórzeń. XGLM od MetaAI jest teraz dostępny w transformatorach Hugging Face:

https://huggingface.co/spaces/valhalla/XGLM-zero-shot-COPA

Akcelerator Transformera od NVidia

Teraz, gdy modele oparte na transformatorach przekroczyły bilion parametrów, prędkość obliczeniowa stała się coraz ważniejsza dla tych modeli, a także dla przyszłych modeli, które będą jeszcze większe. Chociaż wczesne spekulacje sugerowały, że GPT-4 może osiągnąć 100 bilionów parametrów, GPT-4 wydaje się mieć 1,76 biliona parametrów, w oparciu o 8 „komponentów", z których każdy ma 220

miliardów parametrów. Obecnie NVidia dostarcza układ GPU A100, a następcą GPU A100 jest GPU H100, który może zapewnić znacznie większą moc obliczeniową niż A100. W zależności od zastosowania, H100 może zapewnić od 8 do 30 razy szybsze przetwarzanie niż A100. „Hopper" to nazwa układu scalonego H100, który zawiera silnik transformatorowy zapewniający lepszą wydajność. H100 obsługuje inne techniki zwiększające wydajność.

Architektura oparta na transformatorach dla grafów

Chociaż oryginalna architektura transformatorowa dobrze nadaje się do zadań związanych z przetwarzaniem języka naturalnego (NLP), dostępna jest również ulepszona architektura transformatorowa do przetwarzania dowolnych grafów. W szczególności ta ulepszona architektura modyfikuje trzy istniejące aspekty transformatora, a także dodaje nową właściwość:

• mechanizm uwagi

• kodowanie pozycyjne

• normalizacja warstwy

• reprezentacja cech krawędzi

Więcej informacji na temat tej architektury można znaleźć online:

https://arxiv.org/pdf/2012.09699.pdf

BIBLIOTEKA TRANSFORMERSÓW OD HUGGING FACE

Hugging Face stworzył bibliotekę transformatorów opartą na Pythonie oraz repozytorium open source do tworzenia modeli opartych na architekturze transformatorów, dostępne pod adresem https://github.com/huggingface/transformers. Biblioteka transformatorów open source Hugging Face składa się z trzech głównych komponentów:

• Klasy modeli

• Klasy konfiguracji

• Klasy tokenizatora

Klasy modeli udostępniają ponad dwadzieścia wstępnie wytrenowanych modeli transformatorów. Klasy konfiguracji udostępniają różne klasy z parametrami niezbędnymi do utworzenia instancji modelu. Klasy tokenizera to komponenty, które wyposażają każdy model w słownik i metody kodowania/dekodowania ciągów znaków w celu zarządzania osadzeniem tokenów i indeksami. W dalszej części tego rozdziału znajdziesz przykłady kodu w Pythonie, które pokazują, jak używać klasy tokenizera z Hugging Face.

Biblioteka Transformers i zadania NLP

Biblioteka Transformers udostępnia wstępnie wytrenowane modele dla NLU i NLG. W rzeczywistości Hugging Face udostępnia ponad 30 wstępnie wytrenowanych modeli dla ponad 100 języków, a także zapewnia kompatybilność między TensorFlow 2 i PyTorch. Ponadto Hugging Face obsługuje modele związane z BERT, GPT-2/GPT-3, XLNet i inne. Oto lista niektórych modeli obsługiwanych przez Hugging Face:

• BART (od Facebooka)

- BERT (od Google)
- Blenderbot (od Facebooka)
- CamemBERT (od Inria/Facebook/Sorbonne)
- CTRL (od Salesforce)
- DeBERTa (od Microsoft Research)
- DistilBERT (od Hugging Face)
- ELECTRA (od Google Research/Uniwersytet Stanforda)
- FlauBERT (od CNRS)
- GPT-2 (od OpenAI)
- Longformer (od AllenAI)
- LXMERT (od UNC Chapel Hill)
- Pegasus (od Google)
- Reformer (od Google Research)
- RoBERTa (od Facebooka)
- SqueezeBert
- T5 (od Google AI)
- Transformer-XL (od Google/CMU)
- XLM-RoBERTa (od Facebook AI)
- XLNet (od Google/CMU)

Sprawdź Więcej informacji na temat modeli wymienionych na powyższej liście można znaleźć w dokumentacji online. Te modele językowe łącznie obsługują NLU i NLG, umożliwiając wykonywanie zadań takich jak klasyfikacja tekstu, ekstrakcja informacji i generowanie tekstu.

Klasa potoku Transformer

Biblioteka Transformers zapewnia standardowy interfejs dla wielu modeli, co upraszcza integrację modeli językowych z aplikacjami niestandardowymi. W szczególności biblioteka Transformers obsługuje JAX, PyTorch i TensorFlow.Biblioteka transformatorów udostępnia również klasę pipeline(), której można używać w mechanizmie opartym na potokach. Poniższy blok kodu ilustruje, jak utworzyć instancję potoku transformatora dla różnych zadań przetwarzania języka naturalnego (NLP) określonych w cudzysłowie (pełne przykłady kodu zostaną omówione później):

```
from transformers import pipeline

classifier1 = pipeline("ner")

classifier2 = pipeline("klasyfikacja-tokenów")

classifier3 = pipeline("pytanie-odpowiedź")

classifier4 = pipeline("analiza-nastroju")
```

```
classifier5 = pipeline("generacja-tekstu")

classifier6 = pipeline("ekstrakcja-cech")

classifier7 = pipeline("podsumowanie")

classifier8 = pipeline("klasyfikacja-nastroju")
```

Jak widać w poprzednim bloku kodu, klasa pipeline() obsługuje NER, klasyfikację tokenów, analizę sentymentu i tak dalej. Ponadto biblioteka transformers obsługuje powyższe zadania przetwarzania języka naturalnego w ponad 100 językach. Hugging Face Hub zawiera liczne modele:

https://huggingface.co/models

ZŁOŻONOŚĆ ARCHITEKTURY TRANSFORMATORA

Złożoność architektury transformatora jest sumą złożoności mechanizmu uwagi i złożoności różnych komponentów architektury transformatora. Mnożenie macierzy wymaga złożoności O(N^2) dla każdego produktu, aby obliczyć podobieństwo parami między każdą parą tokenów. Poniższa lista zawiera złożoność kilku komponentów architektury transformatora:

• O(N^2*D): złożoność samouwagi

• O(N*D^2): złożoność warstw zapytania/klucza/wartości

• O(N*D^2): złożoność warstwy sprzężenia zwrotnego

W większości przypadków wartość D jest znacznie większa niż N. Na przykład wartość D w GPT-3 wynosi 12288, a N jest znacznie mniejsze niż D. Kolejnym szczegółem, który należy wziąć pod uwagę, jest to, że przepustowość jest zazwyczaj wąskim gardłem dla LLM, co oznacza, że zmniejszenie czasu obliczeniowego i zasobów pamięci w połączeniu z tą samą przepustowością nie poprawia ogólnej wydajności. Zmniejszanie złożoności uwagi. Mechanizm uwagi w architekturze transformatora ma złożoność O(N^2), gdzie N równa się liczbie tokenów. Na szczęście architektura Performer oblicza uwagę ze złożonością O(N), a zatem może obsługiwać okna o znacznie szerszym kontekście. Na przykład Flamingo LLM wykorzystuje model Perceiver (nieomawiany w tej książce) ze względu na rozmiar przetwarzanych obrazów.

PRZYKŁADY KODU TRANSFORMERA HUGGING FACE

Zanim zagłębimy się w dodatkowe szczegóły dotyczące architektury transformatora, przyjrzyjmy się kilku przykładom kodu w Pythonie, które ilustrują, jak używać transformatorów Hugging Face do wykonywania zadań związanych z przetwarzaniem języka naturalnego (NLP). Chociaż te przykłady kodu są bardzo proste, pokazują, jak przeprowadzać analizę NER, QnA i analizę sentymentu za pomocą transformatora Hugging Face. Ponadto przykłady kodu stanowią solidną podstawę, na której można poszerzyć wiedzę praktyczną na temat transformatorów i modeli językowych opartych na transformatorach.

Zadania dotyczące transformatorów i NER

Listing przedstawia zawartość pliku hf_transformer_ner.py, który ilustruje sposób wykonania zadania NER z transformatorem Hugging Face.

Listing: hf_transformer_ner.py

```
from transformers import pipeline
```

```
import pandas as pd

nlp = pipeline('ner')

result = nlp("I am a UCSC instructor and my name is

Oswald")

print("result:",result)

print()

df = pd.DataFrame(result)

print("df:",df)

print()

nlp = pipeline('ner')

result = nlp("I Love Chicago Pizza and I Really

Like New York Bagels")

print("=> result:",result)

print()

df = pd.DataFrame(result)

print("df:",df)
```

Listing rozpoczyna się od instrukcji import, a następnie inicjalizuje zmienną nlp jako instancję funkcji potoku, z parametrem ner. Następnie zmienna nlp jest wywoływana za pomocą zakodowanego na stałe przykładowego zdania. Dane wyjściowe są przypisywane do zmiennej result, której zawartość jest następnie wyświetlana. Uruchom kod z Listingu za pomocą następującego polecenia:

```
python3 hf_transformer_ner.py
```

Poprzednie polecenie uruchamia wersję Pythona zainstalowaną na Twoim komputerze. Jak dowiedziałeś się we wstępie do tego rozdziału, Hugging Face obecnie obsługuje Pythona 3.7, ale w przyszłości prawdopodobnie będą obsługiwane również późniejsze wersje Pythona. Powyższe polecenie wyświetli następujące dane wyjściowe:

```
=> result: [{'word': 'UC', 'score':
0.9993938207626343, 'entity': 'I-ORG', 'index':
4}, {'word': '##SC', 'score': 0.9974051713943481,
'entity': 'I-ORG', 'index': 5}, {'word': 'Oswald',
'score': 0.9988114833831787, 'entity': 'I-PER',
'index': 11}]

df:   word     score  entity  index  start  end
0       UC  0.999394   I-ORG      4      7    9
1     ##SC  0.997405   I-ORG      5      9   11
2   Oswald  0.998811   I-PER     11     38   44

=> result: [{'word': 'Chicago', 'score':
0.962387204170227, 'entity': 'I-ORG', 'index': 3,
```

```
'start': 7, 'end': 14}, {'word': 'Pizza', 'score':
0.958823561668396, 'entity': 'I-ORG', 'index': 4,
'start': 15, 'end': 20}, {'word': 'New', 'score':
0.6438480615615845, 'entity': 'I-ORG', 'index': 9,
'start': 39, 'end': 42}, {'word': 'York', 'score':
0.8661817908287048, 'entity': 'I-ORG', 'index': 10,
'start': 43, 'end': 47}, {'word': 'Ba', 'score':
0.5761372447013855, 'entity': 'I-MISC', 'index':
11, 'start': 48, 'end': 50}, {'word': '##gel',
'score': 0.4892355501651764, 'entity': 'I-MISC',
'index': 12, 'start': 50, 'end': 53}]

df:    word     score  entity  index  start  end
0   Chicago  0.962387   I-ORG      3      7   14
1     Pizza  0.958824   I-ORG      4     15   20
2       New  0.643848   I-ORG      9     39   42
3      York  0.866182   I-ORG     10     43   47
4        Ba  0.576137   I-MISC    11     48   50
5     ##gel  0.489236  I-MISC     12     50   53
```

Zadania Transformer i QnA

Listing 2 przedstawia zawartość pliku hf_transformer_qa.py, który ilustruje sposób wykonania zadania typu „pytanie-odpowiedź” z wykorzystaniem transformatora Hugging Face.

Listing 2: hf_transformer_qa.py

```
from transformers import pipeline

nlp = pipeline('question-answering')

result = nlp({

'question': "Do you know my name?",

'context': "My name is Oswald"

})

print("result:",result)
```

Listing 2 rozpoczyna się od instrukcji import, a następnie inicjuje zmienną nlp jako instancję funkcji potoku, z parametrem pytanie-odpowiedź. Następnie wywoływana jest zmienna nlp z parą pytanie/kontekst. Dane wyjściowe są przypisywane do zmiennej wynik, której zawartość jest następnie wyświetlana. Uruchom kod z Listingu 2, a zobaczysz następujący wynik:

```
result: [{'word': 'UC', 'score':
0.9993938207626343, 'entity': 'I-ORG', 'index':
4}, {'word': '##SC', 'score': 0.9974051713943481,
'entity': 'I-ORG', 'index': 5}, {'word': 'Oswald',
'score': 0.9988114833831787, 'entity': 'I-PER',
'index': 11}]
```

Transformator i podsumowanie tekstu

Listing 3 przedstawia zawartość pliku hf_text_summarization.py, który ilustruje sposób przeprowadzania podsumowania tekstu za pomocą transformatora Hugging Face. Należy zauważyć, że blok tekstu w tym przykładzie kodu jest identyczny z tekstem we wstępie.

Listing 3: hf_text_summarization.py

```
from transformers import pipeline

summarizer = pipeline("summarization")

text = """
This chapter is the first of two chapters that
discusses the transformer architecture, which is
the foundation for a plethora of language models,
such as BERT and its variants (discussed in
Chapters 4 and 5), as well as the GPT-x family from
OpenAI, various other LLMs (discussed in Chapters 6
and 7).
This chapter describes the main components of the
original transformer, along with Python-based
transformer code samples for various NLP tasks,
such as NER, QnA, and mask filling tasks. Please
read chapter one that discusses some topics that
are relevant to the material in this chapter.
The first part of this chapter introduces the
transformer architecture that was developed by
Google and released in late 2017. This section also
discusses some NLP Transformer models.


The second part of this chapter discusses the
transformers library from Hugging Face, which is
a company that provides a repository of more than
20,000 transformer-based models. This section also
contains Python-based transformer code samples that
perform various NLP tasks, such as NER, QnA, and
mask filling tasks.
The third part of this chapter provides more
details regarding the encoder component, along
with additional details regarding the attention
mechanism, as well as the decoder component, which
is the other main component of the transformer
architecture.
Before you read this chapter, keep in mind that
there are cases of "forward referencing" concepts
that are discussed in later chapters, which
provide context for a given concept. Of course,
if you are unfamiliar with a concept discussed in
a later chapter, then it will also be challenging
to fully understand the topic under discussion.
As a result, you will probably need to read some
topics more than once, after which concepts become
easier to grasp when you understand the current
context and the context of referenced material in
later chapters (fortunately, this reading process
is finite).
"""

result = summarizer(text)
#print("summary text:",result)
print("summary text:",result[0]['summary_text'])
```

Listing 3 rozpoczyna się od instrukcji import, a następnie inicjuje zmienną summaryr wynikiem wywołania klasy potoku z zadaniem podsumowania. Kolejna część Listingu 3 inicjuje zmienną tekstową text treścią ze wstępu do tego rozdziału. Ostatni blok kodu inicjuje zmienną result wynikiem z summaryr(text), a następnie wyświetla zawartość summary_text. Uruchom kod z Listingu 3, a po pierwszym uruchomieniu zobaczysz następujący wynik:

```
Downloading: 100%|
            | 1.80k/1.80k [00:00<00:00, 283kB/s]
Downloading: 100%|
              | 1.22G/1.22G [00:27<00:00, 44.2MB/s]
Downloading: 100%|
            | 899k/899k [00:00<00:00, 1.55MB/s]
Downloading: 100%|
            | 456k/456k [00:00<00:00, 2.02MB/s]
Downloading: 100%|
            | 26.0/26.0 [00:00<00:00, 6.87kB/s]

summary text:  This chapter is the first of two
that discusses the transformer architecture, which
is the foundation for a plethora of language
models. This chapter describes the main components
of the original transformer, along with Python-
based transformer code samples for various NLP
tasks. The second part of this chapter discusses
the transformers library from HuggingFace,
which provides a repository of more than 20,000
transformer-based models.
```

Podczas kolejnych wywołań kodu Pythona sekcja pobierania zostanie pominięta, ponieważ LLM jest już dostępny na Twoim komputerze. Porównaj wygenerowane dane wyjściowe z zawartością zmiennej text, aby sprawdzić, czy kod zapewnia dokładne podsumowanie.

Transformator i tłumaczenie tekstu

Listing 4 przedstawia zawartość pliku hf_text_translation.py, który pokazuje, jak przetłumaczyć tekst z angielskiego na włoski za pomocą transformatora Hugging Face.

Listing 4: hf_text_translation.py

```
from transformers import pipeline

translator = pipeline("translation_en_to_
it",model="Helsinki-NLP/opus-mt-en-it")

input = "Today the weather was really nice and
sunny"
input = "I love Chicago deep dish pizza and a
bottle of stout"


results = translator(input, clean_up_tokenization_
spaces=True, min_length=50)
print("results[0]['translation_text']:")
print(results[0]['translation_text'])
```

Listing 4 rozpoczyna się od instrukcji import, a następnie inicjuje zmienną translator jako instancję funkcji potoku wraz z parametrem translation_en_to_it (tj. z angielskiego na włoski). Następnie wywoływana jest zmienna nlp z parą pytanie/kontekst. Dane wyjściowe są przypisywane do zmiennej result, której zawartość jest następnie wyświetlana. Zauważ, że sufiks en-it (pogrubiony) dla nazwy modelu to technika skrócona, która określa tłumaczenie z angielskiego na włoski. Uruchom kod z Listingu 4, a zobaczysz następujący wynik:

```
Adoro la pizza a base di piatti profondi di Chicago
e una bottiglia di robusto... e un po' di roba...
e... e... un po' di roba...
```

ZADANIA ZWIĄZANE Z TRANSFORMATOREM I MASKĄ

W tej sekcji dowiesz się, jak wykonywać zadania wypełniania maski za pomocą różnych zdań. Zobaczysz również przykład tego, co się stanie, gdy w zdaniu określisz więcej niż jeden element <mask> (odpowiedź: to nie zadziała). Przed uruchomieniem przykładów kodu upewnij się, że masz połączenie z internetem, w przeciwnym razie pojawi się następujący komunikat o błędzie:

```
raise ValueError(
ValueError: Connection error, and we cannot find
the requested files in the cached path. Please try
again or make sure your Internet connection is on.
```

Listing 5 przedstawia zawartość pliku hf_fill_mask1.py, który ilustruje sposób wypełniania maski za pomocą transformatora Hugging Face.

Listing 5: hf_fill_mask1.py

```
from transformers import pipeline

mlm = pipeline("fill-mask")
prompt = "The dog <mask> over the fence"


result = mlm(prompt)
print("prompt:",prompt)
print("result:",result)
print("---------------")

mlm = pipeline("fill-mask")
prompt = "The dog <mask> over the pool"
result = mlm(prompt)
print("prompt:",prompt)
print("result:",result)
print("---------------")

mlm = pipeline("fill-mask")
prompt = "The horse <mask> over the wall"
result = mlm(prompt)
print("prompt:",prompt)
print("result:",result)
```

Listing 5 rozpoczyna się od instrukcji import, a następnie inicjuje zmienną nlp jako instancję funkcji potoku, która przyjmuje parametr fill-mask. Następnie zmienna nlp jest wywoływana z zakodowanym na stałe ciągiem tekstowym. Dane wyjściowe są przypisywane do zmiennej result, której zawartość jest następnie wyświetlana. Uruchom kod z Listingu 5, a zobaczysz następujący wynik:

```
prompt: The dog <mask> over the fence
result: [{'sequence': '<s>The dog jumps over the
fence</s>', 'score': 0.27710869908332825, 'token':
13855, 'token_str': 'Ġjumps'}, {'sequence':
'<s>The dog leaps over the fence</s>', 'score':
0.1510920524597168, 'token': 32564, 'token_str':
'Ġleaps'}, {'sequence': '<s>The dog jumped over the
fence</s>', 'score': 0.06851337850093842, 'token':
4262, 'token_str': 'Ġjumped'}, {'sequence':
'<s>The dog climbs over the fence</s>', 'score':
0.04117066413164139, 'token': 30318, 'token_str':
'Ġclimbs'}, {'sequence': '<s>The dog walks over the
fence</s>', 'score': 0.03938476741313934, 'token':
5792, 'token_str': 'Ġwalks'}]
---------------
```

```
prompt: The dog <mask> over the pool
result: [{'sequence': '<s>The dog jumps over the
pool</s>', 'score': 0.1273069977760315, 'token':
13855, 'token_str': 'Ġjumps'}, {'sequence':
'<s>The dog walks over the pool</s>', 'score':
0.106533944606781, 'token': 5792, 'token_str':
'Ġwalks'}, {'sequence': '<s>The dog runs over
the pool</s>', 'score': 0.06851048767566681,
'token': 1237, 'token_str': 'Ġruns'}, {'sequence':
'<s>The dog leaps over the pool</s>', 'score':
0.05455796793103218, 'token': 32564, 'token_str':
'Ġleaps'}, {'sequence': '<s>The dog leaping over
the pool</s>', 'score': 0.03309628739953041,
'token': 33189, 'token_str': 'Ġleaping'}]
---------------
prompt: The horse <mask> over the wall
result: [{'sequence': '<s>The horse jumps over the
wall</s>', 'score': 0.05724393203854561, 'token':
13855, 'token_str': 'Ġjumps'}, {'sequence':
'<s>The horse rode over the wall</s>', 'score':
0.056807991117239, 'token': 12783, 'token_str':
'Ġrode'}, {'sequence': '<s>The horse leaps over the
wall</s>', 'score': 0.04696424677968025, 'token':
32564, 'token_str': 'Ġleaps'}, {'sequence':
'<s>The horse climbs over the wall</s>', 'score':
0.04176592454314232, 'token': 30318, 'token_str':
'Ġclimbs'}, {'sequence': '<s>The horse rides over
the wall</s>', 'score': 0.04105382785201073,
'token': 9668, 'token_str': 'Ġrides'}]
```

Listing 6 przedstawia zawartość pliku hf_fill_two_masks.py, pokazującą, co się stanie, gdy w jednym zdaniu określisz więcej niż jeden element <mask>.

Listing 3.: hf_fill_two_masks.py

```
from transformers import pipeline

mlm = pipeline("fill-mask")
prompt = "The horse <mask> over the fence and the
<mask> ate the pizza"
result = mlm(prompt)
print("prompt:",prompt)
print("result:",result)
```

Uruchom kod z Listingu 6, a zobaczysz następujący wynik:

```
raise PipelineException(
transformers.pipelines.base.PipelineException: More
than one mask_token (<mask>) is not supported
```

Zadanie z podpowiedzią i wypełnieniem

Listing 7 przedstawia zawartość pliku hf_prompt_fill.py, który ilustruje, jak określić podpowiedź, a następnie wykonać zadanie z maską wypełnienia za pomocą transformatora Hugging Face.

Listing 7: hf_prompt_fill.py

```
from transformers import pipeline

pipe = pipeline("fill-mask", model="bert-base-
uncased")

# => part 1:
movie_desc = "the main characters of the movie
matrix are neo, morpheus, and trinity."
prompt = "The movie is about [MASK]."
output = pipe(movie_desc + prompt)

print("desc:  ",movie_desc)
print("prompt:",prompt)
for element in output:
  print(f"Token {element['token_str']}:
  \t{element['score']:.3f}%")
print("-------------\n\n")

# => part 2:
movie_desc = "In the movie transformers aliens can
morph into a wide range of vehicles."
prompt = "The movie is about [MASK]."
output = pipe(movie_desc + prompt,
targets=["animals","cars"])


print("desc:  ",movie_desc)
print("prompt:",prompt)
for element in output:
  print(f"Token {element['token_str']}:
  \t{element['score']:.3f}%")
```

Listing 7 rozpoczyna się od instrukcji import, a następnie inicjuje zmienną nlp jako instancję funkcji potoku, która przyjmuje parametr fill-mask. Następnie zmienna nlp jest wywoływana z zakodowanym na stałe ciągiem tekstowym. Dane wyjściowe są przypisywane do zmiennej result, której zawartość jest następnie wyświetlana. Uruchom kod z Listingu 7, a zobaczysz następujący wynik:

```
desc:   the main characters of the movie matrix are
neo, morpheus, and trinity.
prompt: The movie is about [MASK].
Token trinity:  0.058%
Token neo:      0.053%
Token love:     0.050%
Token dreams:   0.028%
Token fate:     0.024%
-------------

desc:   In the movie transformers aliens can morph
into a wide range of vehicles.
prompt: The movie is about [MASK].
Token cars:     0.139%
Token animals:  0.006%
```

PODSUMOWANIE

Ta część rozpoczęła się od wprowadzenia do architektury transformatora, a następnie opisano trzy warianty architektury transformatora: tylko koder (BERT), tylko dekoder (GPT) oraz zarówno koder, jak i dekoder (BART i T5). Następnie zapoznano się z biblioteką transformatorów firmy Hugging Face, która udostępnia ogromne repozytorium modeli opartych na transformatorach. Następnie przedstawiono przykłady kodu wykorzystujące modele oparte na transformatorach do wykonywania różnych zadań przetwarzania języka naturalnego (NLP), takich jak NER, QnA i wypełnianie masek. Ponadto zapoznano

się ze szczegółami komponentu kodera i komponentu dekodera, które są dwoma głównymi komponentami architektury transformatora. Zobaczono również, jak podobne są struktury kodera i dekodera, z dodatkową, maskowaną warstwą samouwagi w dekoderze.