Architektura transformatorów – głębsze omówienie

Ta część jest drugą z dwóch poświęconych architekturze transformatorów, która stanowi podstawę dla wielu modeli językowych, takich jak BERT i jego warianty , a następnie rodzina GPT-x i jej konkurenci. Pierwsza część omawia kodowanie pozycyjne, które obejmuje funkcje sinus() i cosinus() dla architektury transformatorów. W tej sekcji poznasz również trzy typy kodowania pozycyjnego. Druga część omawia dodatkowe szczegóły dotyczące komponentu kodera w architekturze transformatorów, a także przegląd komponentu dekodera. Ta sekcja zawiera również informacje o tym, jak określić, czy użyć kodera, dekodera, czy obu jednocześnie w LLM. Trzecia część rozdziału omawia klasy Auto, takie jak Auto-Model, AutoConfig i AutoTokenizer. Dowiesz się o modelach LLM, które oferują ulepszenia w stosunku do oryginalnej architektury transformatorów, takich jak Reformer, Longformer i transformator Switch, który składa się z biliona parametrów. Ostatnia część tego rozdziału zawiera przykład ładowania japońskiego zbioru danych, a następnie przykład analizy sentymentu. Teraz, gdy zapoznałeś się z przykładami kodu transformatora w Pythonie, zagłębmy się w jego wewnętrzną strukturę.

PRZEGLĄD KODERA

Jak się dowiedziałeś, istnieją LLM-y, które są tylko koderem, LLM-y, które są koderem i dekoderem, oraz LLM-y, które są tylko dekoderem. Ta sekcja dotyczy pierwszych dwóch typów LLM-ów, a kolejna omawia LLM-y zawierające dekoder. Po utworzeniu osadzeń słów z tokenów wejściowych, dla każdego tokena tworzony jest wektor kodowania pozycyjnego. Ten temat zostanie omówiony w kolejnej sekcji, która wyjaśnia, jak używać funkcji sinus() i cosinus() do generowania wektora pozycyjnego dla każdego tokena. Następny krok dotyczy mechanizmu uwagi, który jest kluczowym elementem architektury transformatora: koder bez mechanizmu uwagi udostępnia dekoderowi tylko stan końcowy kodera. W konsekwencji poprzednie stany kodera nie są dostępne dla dekodera, co powoduje wąskie gardło informacyjne, w którym wszystko dotyczące wejścia musi zostać zakodowane w jednym wyjściu kodera. Powyższy scenariusz dotyczy zarówno sieci RNN, jak i LSTM: obie te architektury dostarczają dane wyjściowe ostatniej warstwy bieżącego komponentu do następnego komponentu (takiego jak inny RNN lub inny komponent LSTM). Natomiast koder wykorzystujący mechanizm uwagi umożliwia dekoderowi dostęp do wszystkich pozostałych ukrytych stanów kodera, unikając w ten sposób problemu wąskiego gardła informacyjnego.

Liczba warstw kodera

Chociaż diagram architektury transformatora („Wystarczy uwaga”) przedstawia tylko jedną warstwę kodera (i tylko jedną warstwę dekodera), komponent kodera dla architektury transformatora zazwyczaj ma sześć lub dwanaście połączonych komponentów kodera. Każdy komponent kodera składa się z elementów, które mają dwie warstwy: dane wyjściowe pierwszej warstwy stają się danymi wejściowymi dla drugiej warstwy (jak miniaturowy potok). Dane wyjściowe szóstego (lub w niektórych przypadkach dwunastego) komponentu kodera są następnie przekazywane do każdego elementu dekodera w komponencie dekodera. Podobnie, komponent dekodera również ma sześć (czasem więcej) połączonych komponentów dekodera, dzięki czemu wyjście jednego elementu (w jednym komponencie) staje się wejściem dla następnego elementu. Należy jednak zauważyć, że każdy element kodera ma dwa podelementy, podczas gdy każdy element dekodera składa się z trzech podelementów, z których jeden jest wyjściem z kodera. Modele LLM zawierające koder i dekoder obejmują blok kodera składający się z wielu identycznych komponentów, a także blok dekodera z wieloma identycznymi komponentami. Każdy blok w koderze i dekoderze jest luźno analogiczny do elementów filtrujących w splotowej sieci neuronowej (CNN).

Jak działa koder?

Jak wspomniano w poprzedniej sekcji, komponent koderatransformatora jest w rzeczywistości pionowym stosem bloków kodera. Oryginalny artykuł dotyczący mechanizmu uwagi określa jeden koder z sześcioma komponentami, ale możliwe jest określenie większej liczby bloków. Każdy koder przetwarza dane wejściowe i generuje zakodowane dane wyjściowe, które są przekazywane w górę do kolejnego komponentu w bieżącym bloku kodera. Ostatni (najwyższy) koder w stosie bloków kodera zawiera reprezentację początkowego zdania wejściowego. Zakodowany wynik jest przekazywany do każdego komponentu bloku dekodera w celu przeprowadzenia dalszych kroków przetwarzania. Teraz, gdy omówiliśmy przepływ osadzania słów w komponencie kodera, możemy zagłębić się w proces konwersji tokenów w zdaniu wejściowym na osadzenia słów, co jest tematem następnej sekcji.

Początkowe osadzanie słów dla kodera

Modele uczenia maszynowego nie mogą przetwarzać tokenów wejściowych opartych na ciągach znaków. Zamiast tego tokeny korpusu muszą zostać przekonwertowane na zestaw osadzeń słów. Różne algorytmy wykorzystują różne techniki generowania osadzeń słów. Na przykład word2vec wykonuje to zadanie za pośrednictwem płytkiej sieci neuronowej (tj. pojedynczej warstwy ukrytej). Architektura transformatora obejmuje jednak kilka kroków. Pierwszy krok tworzy „kodowanie tokena" dla każdego tokena, które składa się z wektorów one-hot (poszukaj w internecie, jeśli potrzebujesz więcej informacji na temat kodowań one-hot). Rozmiar kodowań tokenów jest różny, ponieważ zależy od rozmiaru słownika tokenizatora: ten ostatni może mieć od 20 000 unikalnych tokenów do 200 000 unikalnych tokenów. Drugi krok polega na rzutowaniu wektorów one-hot na przestrzeń o niższym wymiarze. W tym momencie są one gotowe do przetworzenia przez koder warstwy początkowej architektury transformatora. W konsekwencji warstwa kodera generuje stan ukryty lub „zamaskowany" dla tokenów wejściowych, po czym kolejna warstwa przewiduje poprawną wartość dla tokena zamaskowanego. (Upewnij się, że rozumiesz różnicę między dostrajaniem a wstępnym trenowaniem modelu). W tym momencie możemy postępować w dwóch kierunkach, w zależności od konkretnego zadania. Jeśli celem jest wstępne trenowanie, stany ukryte wygenerowane z początkowych warstw kodera są dalej przetwarzane przez kolejne warstwy w celu tworzenia prognoz dotyczących zamaskowanych tokenów, a nie samych „stanów ukrytych". Z drugiej strony, jeśli zadaniem jest klasyfikacja, wykorzystywana jest warstwa klasyfikacji, zastępując warstwę modelowania języka używaną we wstępnym trenowaniu. W obu scenariuszach przetwarzanie jest kontynuowane przez pozostałe bloki kodera, które stanowią komponent kodera architektury transformatora. Powtórzmy ważny szczegół: osadzanie słów w architekturach transformatorowych wymaga mechanizmu uwagi, w którym każde osadzanie słów opiera się na wszystkich osadzaniach słów wszystkich tokenów w danym zdaniu. W związku z tym istnieje zasadnicza różnica między osadzaniami słów generowanymi przez algorytm word2vec a osadzaniami generowanymi za pomocą mechanizmu uwagi: ten drugi generuje różne osadzania dla tego samego słowa, które pojawia się w wielu zdaniach, podczas gdy word2vec generuje pojedyncze osadzanie słowa dla danego słowa, niezależnie od kontekstu tego słowa. Zatem, biorąc pod uwagę zdanie z n tokenami, konstrukcja każdego osadzenia słowa obejmuje pozostałe (n-1) słów. Zatem mechanizm oparty na uwadze ma rząd O(N^2), gdzie N to liczba unikalnych tokenów w korpusie.

CZYM SĄ KODOWANIA POZYCYJNE?

Kodowania pozycyjne są dodawane do osadzeń tokenów w sekwencji, aby dostarczyć modelowi informacji o pozycjach tokenów w sekwencji. Ponieważ modele transformatorowe, oparte na mechanizmach samouwagi, z natury nie mają poczucia kolejności ani pozycji, kodowania pozycyjne są kluczowe, aby umożliwić modelowi uwzględnienie kolejności tokenów podczas tworzenia prognoz.

Kodowania pozycyjne wprowadzają informacje o względnej lub bezwzględnej pozycji tokenów w sekwencji. Informacje te są niezbędne do zadań związanych z przechodzeniem z sekwencji do sekwencji, takich jak tłumaczenie, podsumowywanie itp., gdzie kolejność słów ma kluczowe znaczenie. Kodowania pozycyjne mają ten sam wymiar co osadzenia słów, co pozwala nam dodawać je do osadzeń słów. Co więcej, kodowania pozycyjne są dodawane do osadzeń słów przed przekazaniem ich do modelu transformatorowego. Ponadto kodowanie pozycyjne pozwala modelowi pracować z sekwencjami o zmiennej długości, a także umożliwia równoległe przetwarzanie sekwencji, co stanowi znaczącą zaletę modeli transformatorowych w porównaniu z rekurencyjnymi sieciami neuronowymi (RNN). Faktyczna konstrukcja kodowań pozycyjnych opiera się na połączeniu funkcji trygonometrycznych sinus() i cosinus(), co zostanie opisane bardziej szczegółowo w następnej sekcji.

Funkcje sinus() i cosinus() dla kodowań pozycyjnych

Kodowania pozycyjne oblicza się za pomocą trygonometrycznych funkcji sinus() i cosinus(), a następnie dodaje się je do każdego osadzenia słowa. Dla uproszczenia rozważmy pionowo wyrównany zestaw wykresów dla następującego zestawu funkcji sinus(), których częstości należą do zbioru {1, 2, 3, . . . , k}, gdzie k jest dowolną dużą dodatnią liczbą całkowitą, a każdy wykres ma własną pionową oś y z zakresu od -1 do 1:

{sinus(x), sinus(2x), ..., sinus(kx)}

Teraz narysuj pionową linię w dowolnym miejscu na poziomej osi x: ta linia przetnie k funkcji sinus() w k różnych pozycjach z zakresu od -1 do 1. Niech S będzie równe wartości przecięcia pionowej linii i funkcji sinus(). Następnie możemy skonstruować wektor

S = [s1, s2, . . . , sk], który zawiera wartości parami różne.

W analogiczny sposób rozważmy pionowo wyrównany zbiór wykresów

dla następującego zbioru funkcji cosinus():

{cosinus(x), cosinus(2x), ..., cosinus(kx) }

Jeśli narysujesz pionową linię przecinającą poziomą oś x w dowolnym miejscu, linia ta przetnie k funkcji cosinus() w k pozycjach. Ponieważ każda funkcja cosinus() jest rysowana z własną pionową osią y, która mieści się w zakresie od -1 do 1, współrzędna y każdego punktu przecięcia jest również wartością z zakresu od -1 do 1, która składa się ze zbioru wartości C = [c1, c2, . . . , ck]. Wartości na powyższej liście są wartościami unikatowymi, ponieważ k funkcji cosinus() ma różne częstości. Powyższe zestawy wartości S i C stanowią podstawę kodowania pozycyjnego, w którym wektor jest konstruowany z wartości sinusoidalnych dla pozycji parzystych i wartości cosinusoidalnych dla pozycji nieparzystych. Konkretnie, wektor kodowania pozycyjnego ma następującą postać:

P = [s1, c1, s2, c2, . . . , sk, ck]

Jednak wartości si i ci są obliczane za pomocą następujących wzorów (które różnią się od wartości sinusoidalnych i cosinusoidalnych obliczonych wcześniej dla S i C):

si = sin(pos/[10000]**di]

ci = sin(pos/[10000]**di]

Wartość di jest równa (2*i)/dk, gdzie i jest bieżącą wartością indeksu w wektorze kodowania pozycyjnego, a dk jest wymiarowością wejściową. Należy zauważyć, że indeks i zaczyna się od 0, co oznacza, że wyrazy sinusoidalne znajdują się na pozycjach o indeksie parzystym, podczas gdy wyrazy

cosinusoidalne znajdują się na pozycjach o indeksie nieparzystym. Dla przypomnienia, skonstruowany wektor ma ten sam kształt co osadzenie słowa, dlatego możemy dodać wektory kodowania pozycyjnego do osadzania słowa, aby uzyskać nowe wektory o tym samym kształcie co osadzenia słowa. Nowe wektory (tj. suma wektorów osadzania i kodowania pozycyjnego) są przekazywane do pierwszego bloku kodera w komponencie kodera. W rezultacie możemy zidentyfikować pierwotną lokalizację każdego tokena w osadzaniach. Możemy teraz podsumować kodowania pozycyjne jako posiadające następujące właściwości:

• Są one oparte na pozycji tokenów.

• Zawierają unikalne wartości.

• Są one ograniczone powyżej przez 1.

Suma kontekstu słowa i jego kodowania pozycyjnego, oparta na wektorach, jest równa osadzeniu z kontekstem: Osadzenie z kontekstem = kontekst słowa + kodowanie pozycyjne

Ponadto model języka zazwyczaj nie wymaga etykiet do swojego wstępnego treningu.

Trzy typy kodowania pozycyjnego

Wcześniej omówiono kodowanie pozycyjne, które jest jednym z typów kodowania. Istnieją jednak co najmniej trzy różne techniki, które generują informacje związane z pozycją, jak pokazano tutaj:

• osadzanie pozycji

• kodowanie pozycji

• pozycje względne

Modele języka oparte na transformatorach mogą wykorzystywać różne techniki kodowania. Na przykład BERT używa osadzania pozycji zamiast kodowania pozycji.

INNE SZCZEGÓŁY DOTYCZĄCE KODERÓW

W poprzednich częściach opisano początkowe części kodera, a także cel kodowania pozycyjnego. Przyglądając się koderowi w architekturze transformatora, zauważysz, że w każdej warstwie znajduje się blok uwagi wielogłowicowej i sieć sprzężenia zwrotnego, a także krok dodawania i normalizacji.

Blok uwagi wielogłowicowej w koderze

Blok uwagi wielogłowicowej w koderze modelu transformatora spełnia kilka kluczowych celów, zwiększając zdolność modelu do przetwarzania i analizowania sekwencji danych:

• Równoległe mechanizmy uwagi

• Ulepszone uczenie się reprezentacji

• Lepsze skupienie na różnych pozycjach

• Skalowalność i wydajność

Blok uwagi wielogłowicowej obejmuje kilka równoległych głowic uwagi, z których każda pracuje niezależnie, aby obliczać wyniki uwagi i dane wyjściowe. Ponadto obecność wielu głowic umożliwia modelowi jednoczesne uczenie się różnych reprezentacji danych, rejestrując różne aspekty relacji między tokenami.

Sieć sprzężenia zwrotnego w koderze

Sieć neuronowa sprzężenia zwrotnego (FFNN) w koderze modelu transformatora odgrywa kluczową rolę w przetwarzaniu informacji uzyskanych z poprzednich warstw i mapowaniu ich na potencjalnie bardziej złożoną przestrzeń w celu uchwycenia skomplikowanych wzorców w danych. W szczególności sieć FFNN wykorzystuje nieliniowe funkcje aktywacji ReLU (Rectified Linear Unit) lub GELU (Gaussian Error Linear Unit). W rzeczywistości sieć FFNN zazwyczaj składa się z jednej lub więcej warstw liniowych, po których następują nieliniowe funkcje aktywacji. W przeciwieństwie do warstw uwagi, które uwzględniają relacje między różnymi tokenami, sieć FFNN przetwarza każdy token niezależnie. Dzięki temu sieć FFNN może skupić się na ekstrakcji cech z poszczególnych tokenów, bez wpływu innych tokenów w sekwencji. Techniki regularyzacji, takie jak dropout, można zastosować w sieci FFNN, aby zapobiec nadmiernemu dopasowaniu i ułatwić uczenie się solidnych reprezentacji. Mówiąc w skrócie, sieć FFNN w koderze modelu transformatora służy do wprowadzania transformacji nieliniowych, ekstrakcji cech z pojedynczych tokenów, dodawania głębi modelowi i usprawniania uczenia się złożonych, hierarchicznych reprezentacji danych. Działa ona w tandemie z wielogłowicową warstwą uwagi, aby efektywnie przetwarzać i analizować sekwencje, odgrywając kluczową rolę w sukcesie modeli transformatorów w szerokim zakresie zadań przetwarzania języka naturalnego (NLP).

Czym jest dodawanie i normalizacja?

Jak opisano wcześniej, krok dodawania i normalizacji łączy dane wejściowe do komponentu z danymi wyjściowymi z komponentu, a następnie ta kombinacja podlega normalizacji. Ten krok jest wykonywany zarówno w wielogłowicowej warstwie uwagi, jak i w warstwie sprzężenia zwrotnego. Cel kroku dodawania jest w pewnym sensie analogiczny do działania repeatera w sieci, który zasadniczo „wzmacnia" (osłabiony) sygnał. Celem etapu normalizacji jest zapewnienie, że wartości wynikowe mieszczą się w tym samym zakresie (tj. bez wartości odstających lub innych nietypowo dużych wartości), co z kolei może skutkować krótszym czasem obliczeń. Możemy teraz przedstawić elementy komponentu kodera architektury transformatora w następujący sposób (każdy ENC jest blokiem kodera):

Osadzanie wejściowe > kodowanie pozycyjne > ENC1 > ENC2 > . . . ENCn

Liczba bloków kodera n wynosi zazwyczaj 6 lub 12, a czasami 24 (i potencjalnie może być większa). Sygnał wejściowy INP do pierwszego kodera ENC1 jest sumą osadzenia wejściowego i kodowania pozycyjnego, a zawartość każdego bloku kodera ENC jest pokazana tutaj:

MHA(out)+INP > ADD+Norm > FF > ADD+Norm > następny blok kodera

Teraz, gdy rozumiesz, jak tworzone są początkowe osadzenia słów dla tokenów z danego korpusu, przyjrzyjmy się bliżej mechanizmowi self-attention w koderze.

PRZEGLĄD DEKODERA

Komponent dekodera jest bardzo podobny do komponentu kodera i składa się z następujących elementów:

• Wiele bloków dekodera

• Kodowanie pozycyjne

• Osadzanie wyjściowe

Osadzenie wyjściowe to najniższy element każdego bloku dekodera (który zazwyczaj obejmuje 6 lub 12 takich bloków), a jego odpowiednikiem jest osadzenie wejściowe w komponencie kodera. Powyżej

tego elementu znajduje się kodowanie pozycyjne, które jest odpowiednikiem kodowania pozycyjnego w komponencie kodera. Powyżej kodowania pozycyjnego znajduje się zbiór jednego lub więcej bloków dekodera. Komponent dekodera można traktować jako nadzbiór komponentu kodera: podczas gdy bloki kodera zawierają dwie podwarstwy, bloki dekodera zawierają te same podwarstwy, a także trzecią podwarstwę, nazywaną maskowaną warstwą uwagi wielogłowicowej. Warstwa maskowanej uwagi wielogłowicowej (MMHA) zapewnia, że można skupić tylko uwagę „patrzącą wstecz" (od prawej do lewej). Model BERT jest trenowany jako model dwukierunkowy, co oznacza, że może widzieć zarówno poprzedni token, jak i następny token dowolnego danego tokena w sekwencji tokenów. W związku z tym BERT nie musi przewidywać następnego tokena w sekwencji, ponieważ w rzeczywistości zna następny token. Warstwa MMHA skutecznie uniemożliwia każdemu LLM sprawdzanie następnego tokena, co z kolei oznacza, że kolejny token musi zostać przewidziany. Nawiasem mówiąc, modele LLM, które mogą wykonywać jedynie analizę wsteczną, nazywane są modelami autoregresyjnymi.

Jak działa komponent dekodera?

Komponent dekodera działa w podobny sposób jak komponent kodera, wraz z dodatkiem MMHA opisanym w poprzedniej sekcji. Komponent dekodera zawiera bloki dekodera, gdzie pierwszy blok dekodera dekoduje dane wejściowe dostarczane z komponentu kodera. Dane wyjściowe z pierwszego bloku dekodera (i kolejnych bloków dekodera) są przekazywane do następnego bloku dekodera (w sposób oddolny) w komponencie dekodera. Ponadto dane wyjściowe z komponentu kodera są dostępne dla każdego bloku dekodera w komponencie dekodera. Każdy blok dekodera w komponencie dekodera otrzymuje dane wyjściowe z poprzedniego bloku dekodera, a także dane wyjściowe z komponentu kodera (przesłane przez wielogłowicową warstwę uwagi) jako dane wejściowe.

Autoatencja w dekoderze

Zarówno blok kodera, jak i blok dekodera zawierają wielogłowicowy element uwagi. Dodatkowo, blok dekodera zawiera komponent z zamaskowanym wielogłowicowym elementem uwagi.

Zamaskowany element wielogłowicowy uniemożliwia dekoderowi sprawdzenie kolejnego tokena w sekwencji, „wymuszając" w ten sposób wygenerowanie kolejnego słowa. Czynność ta jest realizowana poprzez ustawienie wartości w macierzy w prawym górnym rogu na ujemną nieskończoność, po czym funkcja softmax() zastępuje te wartości wartością 0, zapewniając w ten sposób, że kolejny token jest niedostępny.

Generowanie sekwencji wyjściowej

Ten krok jest wykonywany sekwencyjnie, co obejmuje sekwencję operacji przypominającą pętlę:

1. Rozpocznij od tokena startowego <SOS> jako początkowego wejścia do dekodera.

2. Przepuść ten token przez warstwę osadzania wyjściowego, aby uzyskać odpowiednie osadzenie.

3. Użyj tego osadzenia wraz z wyjściem kodera (lub wektorem kontekstu) jako wejścia do komponentu dekodera, aby wygenerować stan ukryty.

4. Stan ukryty służy do wygenerowania rozkładu prawdopodobieństwa w słowniku dla kolejnego tokena wyjściowego.

5. Wypróbuj lub wybierz najbardziej prawdopodobny token z dystrybucji jako token wyjściowy.

6. Dołącz ten token wyjściowy do wcześniej wygenerowanej listy tokenów wyjściowych.

7. Użyj listy wygenerowanych dotychczas tokenów wyjściowych (w tym najnowszego tokenu wyjściowego) jako nowego wejścia do dekodera w następnym kroku, wraz z wyjściem kodera.

8. Powtarzaj kroki od 2 do 7, aż zostanie wygenerowany token końcowy <EOS> lub osiągnięta zostanie maksymalna długość sekwencji.

Na przykład rozważ zdanie S, które jest równe „Uwielbiam pizzę z Chicago”. Zdanie wejściowe: „Uwielbiam pizzę z Chicago”.

Wyjście kodera: (Reprezentacja zdania wejściowego w przestrzeni wielowymiarowej, uzyskana po przetworzeniu zdania wejściowego przez koder).

Sekwencja kroków dekodera jest pokazana poniżej:

1. Rozpocznij od tokena początkowego <SOS> jako początkowego wejścia do dekodera.

Bieżący token wejściowy: <SOS>

2. Przepuść ten token przez warstwę osadzania wyjściowego, aby uzyskać odpowiednie osadzanie.

Osadzanie <SOS>: [0,1, 0,3, ..., 0,5] (hipotetyczny wektor osadzania)

3. Użyj tego osadzania wraz z wyjściem kodera jako danych wejściowych dla komponentu dekodera, aby wygenerować stan ukryty.

Stan ukryty: [0,4, 0,2, ..., 0,9] (hipotetyczny wektor stanu ukrytego)

4. Stan ukryty służy do generowania rozkładu prawdopodobieństwa w słownictwie dla kolejnego tokena wyjściowego.

Rozkład prawdopodobieństwa: {„J’”: 0,7, „aime”: 0,1, „Chicago”: 0,05, „pizza”: 0,05, ..., „<EOS>”: 0,01} (hipotetyczny rozkład w słownictwie francuskim)

5. Wylosuj lub wybierz najbardziej prawdopodobny token z rozkładu jako token wyjściowy.

Wybrany token wyjściowy: „J’” (hipotetycznie najbardziej prawdopodobny token zgodnie z rozkładem)

6. Dołącz ten token wyjściowy do wcześniej wygenerowanej listy tokenów wyjściowych.

Tokeny wyjściowe do tej pory: [<SOS>, „J’”]

7. Użyj listy tokenów wyjściowych wygenerowanych do tej pory (w tym najnowszego tokenu wyjściowego) jako nowego wejścia do dekodera w następnym kroku, wraz z wyjściem kodera.

Nowe dane wejściowe do dekodera: [<SOS>, „J’”]

8. Powtarzaj kroki od 2 do 7, aż zostanie wygenerowany token końcowy <EOS> lub osiągnięta zostanie maksymalna długość sekwencji.

Powtórz proces, a w kolejnych iteracjach możemy wygenerować tokeny „aime”, „la”, „pizza”, „de”, „Chicago” i na końcu „<EOS>”, co da w rezultacie zdanie po francusku: „J’aime la pizza de Chicago”.

KODER, DEKODER CZY OBA: JAK WYBRAĆ?

Istnieją trzy główne implementacje architektury transformatora:

• tylko koder (do klasyfikacji)

• tylko dekoder (do modelowania języka)

• koder-dekoder (do tłumaczenia maszynowego)

Modele oparte wyłącznie na koderze dobrze nadają się do zadań wymagających zrozumienia danych wejściowych, takich jak klasyfikacja zdań i rozpoznawanie jednostek nazwanych. Przykładami modeli opartych wyłącznie na koderze są BERT, alBERT, DistilBERT, RoBERTa i ELECTRA.

Kodery dobrze nadają się do zadań takich jak klasyfikacja tokenów.

Innym zadaniem jest ekstrakcyjna analiza pytań i odpowiedzi, w której przewidujemy początkowy i końcowy token fragmentu tekstu z dokumentu, który najlepiej odpowiada na pytanie. Blok kodera dobrze nadaje się do tych zadań, ponieważ koder mapuje sekwencję wejściową na sekwencję wyjściową o tej samej wymiarowości. Koder kontekstualizuje wektory osadzania tokenów dostarczane do systemu, a nie w celu generowania nowych sekwencji o innej długości. Modele oparte wyłącznie na dekoderze to modele autoregresyjne, które dobrze nadają się do zadań generatywnych, takich jak generowanie tekstu. Przykładami modeli opartych wyłącznie na koderze są GPT, GPT-2 i TransformerXL. Dla danego słowa warstwy uwagi mają dostęp tylko do słów umieszczonych przed nim w zdaniu. Co ciekawe, modele oparte wyłącznie na dekoderze są czasami preferowane w stosunku do modeli koder-dekoder w zakresie generowania języka. Modele koder-dekoder (zwane również modelami sekwencja-do-sekwencji) dobrze nadają się do zadań generatywnych wymagających danych wejściowych, takich jak tłumaczenie maszynowe, gdzie kodujemy sekwencje z jednego języka źródłowego, a następnie dekoder generuje sekwencję o potencjalnie innej długości w języku docelowym. Innym typem architektury koder-dekoder jest architektura multimodalnego transformatora (na przykład wizja-do-tekstu). Modele koder-dekoder są często określane jako seq2seq i są w stanie generować sekwencję wyjściową o innej długości niż sekwencja wejściowa. Przykładem modelu koder-dekoder jest BART, co jest akronimem od Bidirectional and Autoregressive Transformer (Transformator dwukierunkowy i autoregresyjny). Komponent kodera w BART działa tak samo, jak komponent kodera w BERT: przetwarza i kontekstualizuje dane wejściowe. Jednakże komponent kodujący BART dostarcza również wektory klucza i wartości z ostatniej warstwy uwagi do warstw uwagi dekodera. Komponent dekodera generuje dane wyjściowe w sposób autoregresyjny. Istnieją modele oparte na transformatorach, które zawierają oba komponenty architektury transformatora, a także wykonują zadania związane z przetwarzaniem języka naturalnego, które wymagają tylko jednego komponentu transformatora. Dla wyjaśnienia, architektura T5 wykonuje zadania klasyfikacyjne, które wymagają jedynie kodera, a mimo to T5 wykorzystuje koder i dekoder architektury transformatora. Ponadto, przypomnijmy sobie architekturę transformatora, że uwaga przejawia się na kilka sposobów:

• samouwaga w koderze

• samouwaga w dekoderze

• uwaga krzyżowa w układzie koder-dekoder

Mechanizm uwagi krzyżowej umożliwia dekoderowi pobieranie informacji z kodera. W rezultacie mechanizm samouwagi może się różnić w tych architekturach, jak opisano w następnym rozdziale.

Samouwaga przyczynowa

Samouwaga przyczynowa oznacza, że tokeny mogą uzyskać dostęp tylko do poprzednich tokenów, które również należą do tego samego ciągu tekstowego (zdania). Jest to mechanizm, który uniemożliwia dekoderom „zobaczenie" wszystkich tokenów w sekwencji wejściowej (więcej szczegółów poniżej). W niektórych architekturach transformatorowych samouwaga musi być przyczynowo zależna od poprzednich tokenów, podczas gdy inne architektury transformatorowe nie narzucają tego ograniczenia. W szczególności, samouwaga w architekturach opartych na koderze nie

musi być przyczynowo zależna od bieżących i przeszłych tokenów. Natomiast architektury oparte na koderze i dekoderze wymagają samouważności przyczynowej w dekoderze, ponieważ każdy krok dekodowania autoregresyjnego musi zależeć wyłącznie od poprzednich tokenów. Wreszcie, w architekturach opartych na dekoderze, samouważność również musi być przyczynowa. Technika wymuszania uwagi przyczynowej lub „maskowanej" polega na ustawieniu górnych wartości trójkątnych macierzy uwagi nxn równych -oo, po czym funkcja softmax() zastosowana do tej macierzy ustawia wszystkie wartości w górnej części trójkątnej równe ujemnej nieskończoności, której wartość softmax() wynosi 0. W rezultacie tokeny mogą uzyskać dostęp tylko do poprzednich tokenów.

Osadzenia jednokierunkowe i dwukierunkowe

Kontekst jednokierunkowy tokena analizuje token w jednym kierunku, aby utworzyć osadzenia, które zazwyczaj obejmują poprzednie tokeny. Ponieważ tekst piszemy w jednym kierunku (często od lewej do prawej, a czasami od prawej do lewej), generowanie tekstu dobrze nadaje się do osadzania jednokierunkowego. Dwukierunkowy kontekst tokenu analizuje poprzednie tokeny i kolejne tokeny z danego tokenu w celu wygenerowania osadzania. Osadzenia dwukierunkowe są przydatne w zadaniach na poziomie zdań, takich jak przepisywanie lub streszczanie. Należy pamiętać o następującej kwestii: architektura transformatorowa obsługuje dwukierunkowe bloki kodera i jednokierunkowe bloki dekodera. Ponadto modele autoregresyjne działają z osadzaniem jednokierunkowym.

ZANURZAJĄC SIĘ W ARCHITEKTURĘ TRANSFORMATORA

Wcześniej poznałeś kilka aspektów architektury transformatora, takich jak przepływ danych przez komponent kodera i komponent dekodera. Ta sekcja pokrótce opisuje dodatkowe szczegóły dotyczące transformatora, z których niektóre wymieniono tutaj:

- Architektura kodera-dekodera
- Macierz wag i uwaga wielogłowicowa
- Klucz, zapytanie i wartości nie są hiperparametrami
- Współczynnik skalowania dk
- Długość sekwencji wejściowej
- Przejście do przodu
- Połączenia resztkowe
- Normalizacja warstw
- Współczynnik odpadania
- Transformatory i backprop
- Tłumaczenie językowe i kolejność słów
- Osadzanie segmentów
- Transformator kompresujący

Po pierwsze, uwaga wielogłowicowa, będąca warstwą w komponencie dekodera, jest implementowana za pomocą pojedynczej macierzy wag. Ponadto wartości Q, K i V są danymi wejściowymi dla uwagi wielogłowicowej, a ich wartości początkowe są identyczne. Co ciekawe, wymiary wektorów klucza, zapytania i wartości tworzonych przez mechanizm uwagi nie mogą być

modyfikowane programowo, co oznacza, że ich wymiary nie są hiperparametrami. Chociaż współczynnik skalowania d w wyrażeniu Q*Kt/d może wydawać się nietypowy, wyrażenie to zostało określone empirycznie. Jego celem jest zapewnienie, że iloczyn skalarny wygeneruje „rozsądne" wartości. Architektura transformatorowa, sieci RNN i LSTM mają ograniczenia co do długości sekwencji wejściowej. Jeśli sekwencja tekstu jest zbyt długa, można zastosować standardowe techniki (np. obcinanie i dzielenie na fragmenty) do obsługi takich sekwencji tekstu za pomocą modeli opartych na transformatorach. Ciekawa różnica dotyczy przejścia do przodu w fazie uczenia, jak również w fazie wnioskowania: w pierwszej fazie wykonywane jest jedno przejście do przodu dla całej sekwencji, podczas gdy w drugiej dla każdego tokena wykonywany jest przebieg do przodu dekodera.

Połączenia resztkowe

Połączenie resztkowe (znane również jako połączenie pomijające) zostało wprowadzone w modelu ResNet (Sieć Resztkowa) dla systemów wizyjnych, a następnie zaimplementowane w AlphaZero, AlphaFold i w architekturze transformatorowej. Uzasadnieniem dla połączenia resztkowego jest to, że wydajność sieci neuronowej może spadać wraz z dodawaniem kolejnych warstw. Połączenie resztkowe polega na dodaniu początkowego sygnału wejściowego do każdej innej warstwy sieci neuronowej, co efektywnie zwiększa przetworzony sygnał wyjściowy. Połączenie resztkowe jest w pewnym sensie analogiczne do wzmacniacza, który wzmacnia stłumiony sygnał w sieci. Poniższe szczegóły mogą wyjaśnić, jak wykorzystać połączenie resztkowe w sieci neuronowej. Załóżmy, że x jest sygnałem wejściowym dla sieci neuronowej, a F jest funkcją przetwarzającą x podczas przechodzenia przez pierwszą warstwę, więc sygnał wyjściowy wynosi x1 = F(x). Kolejna warstwa również używa F do przetwarzania x1, a jej sygnał wyjściowy wynosi x2 = F(x1). Kontynuując w ten sposób, mamy sekwencję wartości {x, x1, x2, ..., xn}. Połączenie resztkowe po prostu dodaje wejście x do kolejnych wyjść, co tworzy sekwencję {x, x+F(x), x + F(x1), x + F(x2), ..., x + F(xn)}. Należy zauważyć, że istnieją warianty, w których naprzemienne wartości są aktualizowane, aby utworzyć następującą sekwencję: {x, F(x), x+F(x1), F(x2), x+F(x3), ..., x+F(xn)}

Normalizacja warstwy

Normalizacja warstwy (LN) pojawia się bezpośrednio po warstwie uwagi wielogłowicowej, a także bezpośrednio po sieci sprzężenia zwrotnego (omówionej później). Zatem sekwencja wykonania od lewej do prawej wygląda następująco: Multi-Head Attention > LN > FeedForward > LN

Normalizacja warstw jest w pewnym stopniu podobna do normalizacji wsadowej: ta druga normalizuje próbki w wsadzie, podczas gdy pierwsza normalizuje cechy. Dokładniej, normalizacja odejmuje średnią, a następnie dzieli przez odchylenie standardowe dla wszystkich wartości cech. Należy zauważyć, że ta transformacja jest nazywana standaryzacją w innych sytuacjach poza normalizacją warstw (tak, jest to mylące). Oto pełny wzór:

h = g * (x-średnia)/sigma + b (g = wzmocnienie adaptacyjne i b = odchylenie)

Współczynnik odrzuceń

Koncepcja współczynnika odrzuceń w architekturze transformatorowej jest taka sama jak koncepcja współczynnika odrzuceń dla sieci neuronowych (CNN), która została pierwotnie zaproponowana przez Geoffreya Hintona. Najwyraźniej Hinton wpadł na ten pomysł, stojąc w kolejce w banku; uważał to za sposób na zapobieganie (lub przynajmniej ograniczanie) możliwości zmowy między klientami a kasjerami. Idea współczynnika odrzuceń w sieciach neuronowych (CNN) jest prosta, sprytna i zaskakująco skuteczna: współczynnik odrzuceń to liczba z zakresu od 0 do 1 i jest to odsetek liczby neuronów ignorowanych podczas każdego przejścia przez sieć neuronową. Na przykład, jeśli

współczynnik odrzuceń wynosi 0,20, 20% neuronów w warstwie ukrytej jest ignorowanych podczas propagacji w przód (i są one ignorowane również podczas propagacji w tle). Za każdym razem, gdy sieć neuronowa przetwarza nowy punkt danych (tj. wektor liczb), losowo wybrany zestaw neuronów składający się z 20% danej warstwy jest ignorowany. Ta koncepcja zmniejszyła nadmierne dopasowanie, mając na celu uczynienie modelu bardziej uogólnialnym. W przypadku architektury transformatorowej, współczynnik porzucania jest stosowany do warstw uwagi wielogłowicowej (samodzielnej/krzyżowej), a także do sieci sprzężenia zwrotnego w przód (feedforward) zależnego od pozycji.

Transformatory i wsteczna propagacja błędów

Modele oparte na transformatorach podlegają „wstecznej propagacji błędów" (backprop), podobnie jak inne modele głębokiego uczenia (takie jak sieci neuronowe CNN). Interesującym aspektem jest to, że wsteczna propagacja jest wykonywana w całym modelu:

• podwarstwy FF (feedforward)

• macierze wag Q, K i V

• wagi kodera

• wagi dekodera

• osadzenia

Oprócz samouwagi, dekoder zawiera również warstwy uwagi krzyżowej, które obsługują wyjścia kodera. Wagi w tych warstwach są również aktualizowane podczas propagacji wstecznej. Powyższa lista dotyczy modeli tylko koder, tylko dekoder i modeli koder-dekoder.

Tłumaczenie językowe i kolejność wyrazów

Tłumacz Google to narzędzie online, które wykorzystuje transformatory do tłumaczenia języka, a jego dokładność jest imponująca, podobnie jak zdolność do generowania zdań z gramatycznie poprawną kolejnością wyrazów. Jeśli tłumaczyłeś między językami o innej kolejności wyrazów (np. angielskim i niemieckim), możesz się zastanawiać, jak to się właściwie odbywa. Na przykład, oto zdanie w języku angielskim i jego tłumaczenie na język niemiecki:

Przeczytałem książkę.

Ich habe das Buch gelesen.

Złożony angielski czasownik „have read" jest tłumaczony w języku niemieckim jako „habe gelesen", a imiesłów bierny „gelesen" pojawia się po niemieckim rzeczowniku „Buch".

Osadzanie segmentów

Osadzanie segmentów to technika łączenia wielu zdań, której przykład przedstawiono poniżej:

„Uwielbiam pizzę"

„Pizza z Chicago jest moją ulubioną"

Te dwa zdania są łączone w następujący sposób:

tokeny = [[CLS], uwielbiam, pizzę, [SEP], Chicago,

pizza jest, moją, ulubioną, [SEP]]

Jak widać, token [CLS] jest pierwszym tokenem, a token [SEP] służy do oddzielania zdań, a także tokenu końcowego. Teraz możesz zrozumieć, jak skonstruować sekwencję przekazywaną do kodera:

INPUT = osadzenia tokenów + osadzenia sekwencji + osadzenia pozycyjne.

Transformator Kompresyjny

Firma DeepMind stworzyła Transformator Kompresyjny jako rozszerzenie Transformatora, które zapewnia ulepszoną architekturę pamięci dalekiego zasięgu, wraz z nowym zestawem danych do wnioskowania pamięciowego w modelach językowych. Transformator Kompresyjny przeprowadza redukcję: mapuje przeszłe ukryte aktywacje (pamięci) na mniejszy zestaw skompresowanych reprezentacji (pamięci skompresowane). Transformator Kompresyjny wykorzystuje ten sam mechanizm uwagi w odniesieniu do swojego zestawu pamięci i pamięci skompresowanych, ucząc się przeszukiwać pamięć krótkoterminową granularną i długoterminową zgrubną. W istocie Transformator Kompresyjny przechowuje drobnoziarnistą pamięć przeszłych aktywacji, która jest następnie kompresowana do pamięci zgrubnej.

TRANSFORMATORY AUTOKODUJĄCE

Istnieją dwa główne typy modeli opartych na transformatorach: modele autokodujące i modele autoregresyjne. Głównym czynnikiem różnicującym modele autokodujące i autoregresyjne nie jest ich architektura: różnica opiera się na wykonywanych zadaniach i treningu. W szczególności modele autokodera uczą się kodowania, podczas gdy modele autoregresyjne mogą być wykorzystywane do zadań precyzyjnego dostrajania. Prawdopodobnie najbardziej znanym modelem autokodowania jest BERT, który wykonuje dwa zadania wstępnego trenowania: MLM (modelowanie warstw maskowanych) i NSP (przewidywanie następnego zdania). MLM zastępuje około 15% tokenów wejściowych tokenem [MASK], a następnie próbuje przewidzieć oryginalne dane wejściowe. Zadanie NSP polega na ustaleniu, czy para zdań występuje w korpusie w tej właśnie kolejności. BERT uczy się kodowania, które może być przetwarzane przez zadania takie jak udzielanie odpowiedzi na pytania. Należy pamiętać, że modele w stylu BERT przeprowadzają wstępne trenowanie na nienadzorowanych zbiorach danych. Co ciekawe, komponent dekodera oryginalnego transformatora może być wykorzystywany zarówno do autokodowania, jak i zadań autoregresyjnych; podobnie jest w przypadku komponentu kodera i zadań autoregresyjnych. Poniższa lista opisuje niektóre funkcje modeli autokodowania, autoregresji i sekwencji sekwencyjnej (SEQ2SEQ):

• modele autokodowania kodują dane wejściowe na podstawie elementów maskowanych

• modele autoregresji wykorzystują wszystkie poprzednie przewidywania do przewidywań przyszłych

• modele seq2SEQ przekształcają sekwencje bez zmiany semantyki

KLASY „AUTO”

Autoklasy Hugging Face to wygodne klasy, które pobierają odpowiedni model z nazwy i ścieżki określonej jako wartość parametru. Najprostszą klasą z tego zestawu jest klasa AutoModel, która jest zasadniczo klasą bazową. W rzeczywistości pakiet transformers udostępnia różne klasy z prefiksem Auto, które należą głównie do następujących grup:

• AutoModel

• AutoConfig

• AutoTokenizer

Pakiet transformers od Hugging Face udostępnia klasy AutoModel, które można zaimportować do kodu za pomocą następujących fragmentów kodu:

• from transformers import AutoModel

• from transformers import AutoModelForSequenceClassification

• from transformers import AutoModelForTokenClassification

Na przykład, poniższy fragment kodu tworzy instancję

BertModel za pośrednictwem klasy AutoModel:

model = AutoModel.from_pretrained('bert-base-cased')

Klasa AutoModelForSequenceClassification rozszerza funkcjonalność klasy AutoModel, dodając kolejną warstwę klasyfikacji na podstawie wyników wstępnie wytrenowanego modelu. Klasa AutoModelForTokenClassification rozszerza funkcjonalność klasy AutoModel, zapewniając warstwę klasyfikacji na podstawie istniejącego modelu. Ponadto ta klasa jest wymagana do wykonania zadania NER (rozpoznawania nazwanych jednostek). Poniższe trzy klasy są odpowiednikami trzech klas wymienionych na początku tej sekcji, a prefiks TF wskazuje, że są one przeznaczone dla TensorFlow:

• from transformers import TFAutoModel

• from transformers import TFAutoModelForSequence-Classification

• from transformers import TFAutoModelForTokenClassification

Instancja AutoConfig tworzy instancję klasy odpowiedniej architektury (tj. do konfiguracji), a instancja AutoTokenizer tworzy instancję klasy odpowiedniej architektury.

ULEPSZONE ARCHITEKTURY

Architektury w tej sekcji oferują ulepszenia w stosunku do oryginalnej architektury transformatorowej:

• Reformer

• Longformer

• Switch

• ELECTRA

Reformer

Architektura Reformera jest bardziej wydajna niż oryginalna architektura transformatorowa i zapewnia większe okna kontekstowe. Architektura Reformera osiąga swoją wydajność głównie dzięki dwóm technikom:

• Hashowanie wrażliwe na lokalizację (LSH)

• Odwracalne warstwy resztkowe

LSH zmniejsza złożoność przetwarzania długich sekwencji, podczas gdy odwracalne warstwy resztkowe są bardziej wydajne pod względem wykorzystania pamięci.

Longformer

Model Longformera to modyfikacja architektury transformatorowej, która dobrze nadaje się do przetwarzania długich dokumentów. Obliczanie wartości uwagi ma złożoność O(N^2) (gdzie N to długość sekwencji) w oryginalnej architekturze transformatorowej. W przeciwieństwie do tego, model Longformer zmniejszył złożoność obliczania uwagi do O(N).

Transformator przełącznika: bilion parametrów

Naukowcy z Google stworzyli model Switch LLM, który zawiera bilion parametrów, czyli prawie sześć razy więcej niż GPT-3 (175 miliardów parametrów). Model ten jest jednym z największych modeli, jakie kiedykolwiek stworzono, i aż cztery razy szybszym niż T5-XXL (poprzedni model LLM firmy Google). Naukowcy połączyli prostą architekturę z dużymi zbiorami danych i liczbą parametrów. Ponieważ trenowanie na dużą skalę jest intensywne obliczeniowo, zastosowali „transformator przełącznika", technikę wykorzystującą tylko podzbiór parametrów modelu. Oprócz rzadkości modelu, transformator przełącznika zręcznie wykorzystuje procesory GPU i TPU do intensywnych operacji mnożenia macierzy.

ELECTRA

ELECTRA to akronim od Efficiently Learning an Encoder that Classifies Token Replacements Accurately (Efektywne Uczenie się Kodera, który Dokładnie Klasyfikuje Zamienniki Tokenów); wykorzystuje on uczenie samonadzorowane. ELECTRA może wstępnie trenować sieci transformatorów w sposób wydajny obliczeniowo. ELECTRA wykorzystuje wstępne trenowanie koderów tekstu jako dyskryminatorów, a nie generatorów.

POTOKI HUGGING FACE I JAK DZIAŁAJĄ

Istnieją trzy główne kroki wymagane do wykonania różnych zadań związanych z NLP za pomocą Hugging Face:

• Zaimportuj funkcję potoku Hugging Face.

• Załaduj wstępnie wytrenowany model z Hugging Face.

• Określ zdanie (lub ciąg znaków) do wstępnie wytrenowanego modelu w celu przetworzenia.

Funkcja potoku Hugging Face (nazwa-zadania), określona jako pierwszy krok na powyższej liście, zapewnia intuicyjny mechanizm wykorzystania modelu NLP do wnioskowania. Potok może abstrahować większość kodu z biblioteki i zapewnić dedykowane API dla różnych zadań, z których niektóre przedstawiono tutaj:

• Potok automatycznego rozpoznawania mowy

• Potok pytań i odpowiedzi

• Potok tłumaczenia

Ponadto funkcja potoku Hugging Face to abstrakcja wysokiego poziomu, która została zaprojektowana do wykonywania następujących zadań:

• tokenizacja tekstu w korpusie

• konwersja tekstu na liczby całkowite

• przekazywanie tych liczb całkowitych przez model

Jak widać z powyższej listy, zadania, które mogłeś wykonać programowo, są wykonywane przez funkcję potoku w Hugging Face. Natomiast TensorFlow 2 wymaga jawnego wykonania niektórych z

powyższych zadań za pomocą odpowiedniego kodu. Oprócz wstępnie wytrenowanego modelu, obiekt potoku może określać tokenizator, ekstraktor cech i podstawową infrastrukturę. Pierwszy krok obejmuje instrukcję importu dla biblioteki transformatorów, jak pokazano tutaj:

```
from transformers import pipeline
```

Drugi krok obejmuje określenie ciągu znaków wskazującego zadanie, które chcesz wykonać. Na przykład, poniższy fragment kodu dotyczy analizy sentymentu:

```
nlp = pipeline('sentiment-analysis')
```

Poprzedni fragment kodu 1) wybiera wstępnie wytrenowany model, który 2) został dostrojony do analizy sentymentu i 3) jest w języku angielskim. Jeśli chcesz zobaczyć prosty i kompletny przykład kodu, który pokazuje, jak uzyskać sentyment (pozytywny lub negatywny), zapoznaj się z przykładem kodu w rozdziale 8. Pamiętaj, że wynik jest liczbą z zakresu od 0 do 1, zarówno dla sentymentu pozytywnego, jak i negatywnego, i możesz również przetworzyć tablicę zdań zamiast używać pętli do iteracji każdego zdania. Po utworzeniu instancji obiektu klasyfikatora, który w poprzednim fragmencie kodu nazywa się nlp, model jest pobierany i przechowywany w lokalizacji, z której można uzyskać do niego dostęp w celu kolejnych wywołań skryptów Pythona odwołujących się do niego (tj. wykonywane jest jednorazowe pobranie). Jeśli jednak zmodyfikujesz model, oryginalny model zostanie pobrany ponownie. Przekazanie tekstu do potoku składa się z trzech głównych kroków:

- Tekst jest wstępnie przetwarzany do formatu zrozumiałego dla modelu.
- Wstępnie przetworzone dane wejściowe są przekazywane do modelu.
- Prognozy modelu są przetwarzane później, aby można je było zrozumieć.

Dostępne potoki danych z Hugging Face

Wcześniej przedstawiliśmy listę zadań NLP, które mogą wykonywać transformatory, i są one tutaj przytoczone dla wygody:

- ekstrakcja cech (pobieranie wektorowej reprezentacji tekstu)
- maska wypełnienia
- NER
- analiza pytań i odpowiedzi
- analiza sentymentu
- podsumowanie
- generowanie tekstu
- tłumaczenie
- klasyfikacja zero-shot

ZBIORY DANYCH HUGGING FACE

Ta sekcja zawiera przykłady kodu Pythona, które pokazują, jak wyświetlić listę wszystkich dostępnych zbiorów danych Hugging Face i jak załadować konkretny zbiór danych. Wyświetlana jest zawartość pliku hf_datasets.py, który pokazuje, jak wyświetlić listę dostępnych zbiorów danych Hugging Face

```
from datasets import list_datasets, load_dataset
from pprint import pprint

datasets_list = list_datasets()
print(f"HuggingFace provides {len(datasets_list)}
datasets")
print()

print("-> FIRST FIVE DATASETS:")
print(f"{datasets_list[:5]}")
print()

"""
print("-> LIST OF DATASETS:")
pprint(datasets_list,compact=True)
print()

print("-> DETAILS OF EACH DATASET:")
# this list contains details of each dataset:
datasets_list = list_datasets(with_details=True)
pprint(datasets_list)
"""
```

Funkcja Listing importuje dwie klasy z zestawów danych, a następnie inicjuje zmienną datasets_list. Powoduje to wywołanie API list_data-sets(), a następnie wyświetlenie zawartości zestawów danych w tej zmiennej. Uruchom kod z Listingu 4.1 w Pythonie 3.7, a zobaczysz częściową listę ponad 600 zestawów danych w Hugging Face.

```
HuggingFace provides 3753 datasets

-> FIRST FIVE DATASETS:
['acronym_identification', 'ade_corpus_v2',
'adversarial_qa', 'aeslc', 'afrikaans_ner_corpus']

-> DETAILS OF EACH DATASET:
[datasets.ObjectInfo(
        id='acronym_identification',
        description='Acronym identification
training and development sets for the acronym
identification task at SDU@AAAI-21.',
        files=None
),
 datasets.ObjectInfo(
        id='ade_corpus_v2',
        description=' ADE-Corpus-V2  Dataset:
Adverse Drug Reaction Data. This is a dataset for
Classification if a sentence is ADE-related (True)
or not (False) and Relation Extraction between
Adverse Drug Event and Drug. DRUG-AE.rel provides
relations between drugs and adverse effects. DRUG-
DOSE.rel provides relations between drugs and
dosages. ADE-NEG.txt provides all sentences in the
ADE corpus that DO NOT contain any drug-related
adverse effects.',
        files=None
),
// details omitted for brevity
```

```
 datasets.ObjectInfo(
        id='ncoop57/csnc_human_judgement',
        description='This new dataset is designed
to solve this great NLP task and is crafted with a
lot of care.',
        files=None
)]
```

Ładowanie zestawu danych Iris

List pokazuje, jak załadować zestaw danych cifar10 do Hugging Face i wyświetlić część jego zawartości.

```
from datasets import load_dataset
train_ds, test_ds = load_dataset('cifar10',
split=['train[:4000]', 'test[:1000]'])

#display one column:
#print("=> train_ds['label']:")
#print(train_ds['label'])
#print()
"""
partial output:
[6, 9, 9, 4, 1, 1, 2, 7, 8, 3, 4, 7, 7, 2, 9, 9, 9,
3, 2, 6, 4, 3, 6, 6, 2, 6, 3, 5, 4, 0, 0, 9, 1, 3,
4, 0, 3, 7, 3, 3, 5, 2, 2, 7, 1, 1, 1, 2, 2, 0, 9,
5, 7, 9, 2, 2, 5, 2, 4, 3, 1, 1, 8, 2, 1, 1, 4
...
7, 3, 0, 8, 6, 7, 7, 0, 0, 1, 4, 1, 1, 6, 6, 6, 1,
3, 6, 0, 1, 3, 8, 5, 0, 1, 6, 5, 5, 4, 3, 9, 8, 6,
0, 5, 4, 9, 2, 8, 4, 8, 8, 2, 1, 4, 8, 6, 7, 3, 1,
3, 4, 9, 4, 8, 4, 5, 0, 9, 1, 3, 8, 7, 5, 4, 6]

"""

#display one row:
print("=> train_ds[7]['img'][0][0]:")
#print(train_ds[7]['img'][0][0])
# => output: [28, 35, 39]


print("train_ds[7]['img'][0]:")
print(train_ds[7]['img'][0])
# => output:
#[[28, 35, 39], [30, 34, 44], [33, 44, 47], [62,
83, 72], [63, 84, 72], [31, 49, 46], [29, 47, 50],
[42, 62, 63], [55, 77, 71], [67, 83, 74], [92, 108,
98], [76, 96, 85], [57, 80, 68], [75, 99, 88], [69,
82, 79], [57, 59, 60], [74, 72, 73], [98, 103,
100], [86, 99, 92], [71, 80, 73], [59, 67, 62],
[62, 67, 66], [57, 60, 63], [42, 46, 48], [51, 63,
57], [46, 60, 52], [41, 55, 47], [38, 53, 44], [37,
51, 41], [43, 56, 45], [52, 64, 53], [46, 58, 47]]

# approximately 46K output:
#print(train_ds[7])

# approximately 46K output:
#print(train_ds[7]['img'])

#display a batch of rows:
#print("=> train_ds[3:6]:")
#print(train_ds[3:6])
```

Listing rozpoczyna się od instrukcji import, a następnie inicjuje zmienne train_ds i test_ds wynikiem wywołania API load_datasets(). Uruchom kod z Listingu 4.2, a zobaczysz następujący wynik, zawierający szczegóły pobierania po pierwszym uruchomieniu tego kodu:

```
Downloading: 3.77kB [00:00, 1.44MB/s]
Downloading: 1.50kB [00:00, 401kB/s]
Downloading and preparing dataset cifar10/plain_
text (download: 162.60 MiB, generated: 418.17 MiB,
post-processed: Unknown size, total: 580.77 MiB) to
/Users/oswaldcampesato/.cache/huggingface/datasets/
cifar10/plain_text/1.0.0/951e017399fa3b30edfca5646
8a9
1ac6a03c9167c0b3f79367e777d1f9a0cf7e...
Downloading:  10%|                    | 17.6M/170M
Dataset cifar10 downloaded and prepared to /Users/
oswaldcampesato/.cache/huggingface/datasets/
cifar10/plain_text/1.0.0/951e017399fa3b30edfca56468
a91ac6a03c9167c0b3f79367e777d1f9a0cf7e.

Subsequent calls will reuse this data.
[00:14<05:30, 462kB/s]
```

Ładowanie japońskiego zestawu danych

Listing pokazuje, jak załadować jeden z japońskich zestawów danych w Hugging Face i wyświetlić część jego zawartości.

```
"""
Datasets involving japanese:
 'covid_qa_ucsd', 'covid_tweets_japanese',
'covost2', 'cppe-5',
 'snli', 'snow_simplified_japanese_corpus',
'so_stacksample',
 'Atsushi/fungi_diagnostic_chars_comparison_
japanese',
 'Atsushi/fungi_indexed_mycological_papers_
japanese',
"""

from datasets import load_dataset
snli = load_dataset('snli')
print("-> snli shape:",snli.shape)
print()

print("-> snli:")
print(snli)
print()

# split into train and test data:
train_ds, test_ds = load_dataset('snli',
split=['train[:4000]', 'test[:1000]'])

print("-> train_ds.column_names:")
print(train_ds.column_names)
print()
```

```
print("=> train_ds.features:")
print(train_ds.features)
print()

for idx in range(0,5):
  print("row",idx,"premise:   ",train_ds[idx]
['premise'])
  print("row",idx,"hypothesis:",train_ds[idx]
['hypothesis'])
  print("row",idx,"label:     ",train_ds[idx]
['label'])
  print()
print()

"""
sample output:
{'premise': 'A person on a horse jumps over a
broken down airplane.', 'hypothesis': 'A person is
training his horse for a competition.', 'label': 1}
"""
```

Listing rozpoczyna się od instrukcji importu, a następnie inicjuje zmienną snli jako instancję zbioru danych snli. Kolejny blok kodu wyświetla kształt i niektóre metadane zbioru danych snli. Kolejna część Listingu 4.3 wykonuje podział danych na potrzeby testów, co jest standardowym krokiem w modelach uczenia maszynowego. Następnie wyświetlane są kolumny zbioru train_ds (część szkoleniowa zbioru danych), po których wyświetlane są cechy zbioru train_ds. Ostatnia część Listingu 4.3 zawiera pętlę, która wyświetla wartości przesłanki, hipotezy i etykiety dla pierwszych pięciu wierszy zbioru train_ds. Uruchom kod z Listingu , a zobaczysz następujący wynik:

```
=> snli shape: {'test': (10000, 3), 'train':
(550152, 3), 'validation': (10000, 3)}

=> snli:
DatasetDict({
    test: Dataset({
        features: ['premise', 'hypothesis',
'label'],
```

```
        num_rows: 10000
    })
    train: Dataset({
        features: ['premise', 'hypothesis',
'label'],
        num_rows: 550152
    })
    validation: Dataset({
        features: ['premise', 'hypothesis',
'label'],
        num_rows: 10000
    })
})

-> train_ds.column_names:
['premise', 'hypothesis', 'label']

-> train_ds.features:
{'premise': Value(dtype='string', id=None),
'hypothesis': Value(dtype='string',
id=None), 'label': ClassLabel(num_classes=3,
names=['entailment', 'neutral', 'contradiction'],
names_file=None, id=None)}

row 0 premise:    A person on a horse jumps over a
broken down airplane.
row 0 hypothesis: A person is training his horse
for a competition.
row 0 label:      1

row 1 premise:    A person on a horse jumps over a
broken down airplane.
row 1 hypothesis: A person is at a diner, ordering
an omelette.
row 1 label:      2

row 2 premise:    A person on a horse jumps over a
broken down airplane.
row 2 hypothesis: A person is outdoors, on a horse.
row 2 label:      0

row 3 premise:    Children smiling and waving at
camera
row 3 hypothesis: They are smiling at their parents
row 3 label:      1

row 4 premise:    Children smiling and waving at
camera
row 4 hypothesis: There are children present
row 4 label:      0
```

TRANSFORMERSY I ANALIZA NASTRÓJU

Analiza nastrojów może być wykorzystana do określenia nastrojów klienta wobec konkretnego produktu lub wydarzenia. Rozważmy poniższą parę zdań, gdzie sentyment pierwszego zdania jest wyraźnie pozytywny, a sentyment drugiego zdania jest wyraźnie negatywny:

1. „Bardzo mi smakowała pizza”.

2. „Nie lubię jeździć samochodem codziennie”.

Teraz rozważmy poniższe zdanie, które składa się z nastrojów pozytywnych, po których następuje sentyment negatywny:

3. „Film był świetny, ale ceny biletów były za wysokie”.

Jeśli przypiszemy -1, 0 i +1 frazom lub zdaniom, które są odpowiednio negatywne, neutralne lub pozytywne, może to sugerować przypisanie wartości +1, 0 i -1 trzem poprzednim zdaniom. Jednak przypisanie wartości 0 do zdania 3) nie ma sensu, ponieważ suma +1 i -1 „znosi się” i daje wartość 0.

KOD ŹRÓDŁOWY DLA MODELI OPARTYCH NA TRANSFORMATORACH

Hugging Face udostępnia następujące repozytorium Github, które zawiera otwarty kod źródłowy dla wielu modeli opartych na transformatorach:

https://github.com/huggingface/transformers/blob/main/src/transformers/models/

Częściowa lista modeli opartych na transformatorach, z których niektóre omówione są w dalszych rozdziałach, znajduje się tutaj:

• albert

• bart

• bert

• bert_generation

• bert_japanese

• bertweet

• big_bird

• deberta

• distilbert

• flaubert

• gpt2

• gpt_neo

• longformer

• megatron_bert

• megatron_gpt2

• pegasus

• perceptionr

• reformer

• roberta

• squeezebert

• t5

- visual_bert
- xlm
- xml_roberta

STRESZCZENIE

Ta część rozpoczyna się od opisu kodowania pozycyjnego w architekturze Transformer, które obejmuje funkcje sine() i cosine(), a także trzy typy kodowania pozycyjnego. Następnie uzyskałeś dodatkowe informacje na temat komponentu enkodera w architekturze transformatora, a następnie omówiłeś komponent dekodera. Ponadto nauczyłeś się, jak zdecydować, czy użyć enkodera, dekodera, czy obu jednocześnie w programie LLM. Dodatkowo poznałeś transformatory Reformer, Longformer i Switch, które zapewniają udoskonalenia w stosunku do oryginalnej architektury transformatorów. Na koniec zobaczyłeś, jak pracować z zestawami danych Hugging Face, a także jak załadować japoński zestaw danych.