

**WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ**

Kacper Urban

Zastosowanie sieci transformer do tłumaczenia języka polskiego
na sekwencje znaków migowych

Projekt inżynierski

Opiekun pracy:
dr hab. inż. Tomasz Kapuściński, prof. PRz

Rzeszów, 2023

Spis treści

1. Wstęp	6
2. Przegląd literatury	8
2.1. Transfer learning	8
2.2. Tłumaczenie maszynowe	8
2.3. Mechanizm uwagi	9
2.4. Architektura transfomera	9
2.4.1. Wstęp	9
2.4.2. Osadzenia	10
2.4.3. Kodowanie pozycyjne	11
2.4.4. Wielogłowicowa uwaga	11
2.4.5. Zamaskowana wielogłowicowa uwaga	11
2.4.6. Warstwa liniowa i softmax	12
2.5. Metryki oceny	12
2.5.1. Wstęp	12
2.5.2. BLEU	13
3. Opis wykorzystanych narzędzi	15
3.1. Tensorflow	15
3.2. Transfomers i HuggingFace	15
3.3. Docker	16
3.4. Neptune	16
4. Opis rozwiązania	17
4.1. Stworzenie środowiska	17
4.2. Format danych	18
4.3. Wczytanie modelu	19
4.4. Przygotowanie danych	20
4.4.1. Wczytanie danych	20
4.4.2. Dostosowywanie formatu danych	21
4.4.3. Podział danych na zbiór treningowy i testowy	23
4.4.4. Tokenizowanie	23
4.5. Trenowanie modelu	24
4.5.1. Przygotowanie modelu	24

4.5.2. Śledzenie parametrów modelu	25
4.5.3. Faza treningu	26
4.5.4. Oszacowanie dokładności modelu	27
5. Eksperymenty	30
5.1. Założenia	30
5.2. Współczynnik uczenia	30
6. Podsumowanie	35
Dodatek A Odtworzenie środowiska uruchomieniowego	37
Windows	37
Linux	37
Literatura	39

1. Wstęp

Celem projektu było wybranie wcześniej wytrenowanego modelu typu transformer umożliwiającego tłumaczenie, a następnie dotrenowanie tego modelu do zadania tłumaczenia języka polskiego na sekwencję znaków migowych oraz ocenienie dokładności modelu.

Zgodnie z aktualnym stanem wiedzy, nie znaleziono podobnego rozwiązania. Jedynie udało się znaleźć publikacje naukowe w języku angielskim, natomiast język migowy nie jest językiem uniwersalnym. Dlatego wykorzystanie tych modeli może być trudne do zastosowania dla języka polskiego migowego. Warto zaznaczyć, że zbudowanie modelu do tłumaczenia języka polskiego na język migowy pomogłoby z wyeliminowaniem wykluczenia osób głuchych. Dzięki temu rozwiązaniu jest możliwe np. tłumaczenie informacji podawanych na stacjach kolejowych na sekwencje znaków migowych. Osoby niesłyszące dzięki takiemu rozwiązaniu mogą być informowane o ewentualnych opóźnieniach na trasie, co może wpływać na większą pewność w wybieraniu komunikacji publicznej. Zakres projektu obejmował:

- a) przygotowanie środowiska,
- b) wyszukanie odpowiedniego modelu,
- c) przygotowanie i przetworzenie danych,
- d) przygotowanie i dotrenowanie modelu,
- e) oszacowanie dokładności modelu,
- f) przeprowadzenie eksperymentów.

Motywacją do napisania projektu była chęć pomocy osobom głuchym oraz zgłębianie wiedzy na temat dużych modeli językowych, które zyskują na popularności. Aktualnie na rynku dostępnych jest wiele rozwiązań usprawniających codzienne funkcjonowanie osób z niepełnosprawnościami. Zaczynając od aplikacji opisujących zdjęcia dla osób niewidomych, kończąc na aplikacjach umożliwiających rozmowę z innymi osobami, dla osób z wadami mowy. Natomiast osoby niesłyszące nie doczekały się propozycji nowego rozwiązania.

Struktura projektu obejmowała:

- a) przegląd literatury - krótki opis wykorzystanych metod oraz architektur,
- b) opis wykorzystanych narzędzi - krótkie przedstawienie wykorzystanych narzędzi w projekcie,
- c) opis rozwiązania - przedstawienie w jaki sposób został wykonany projekt, przykłady kodu,
- d) eksperymenty - przedstawienie przeprowadzonych eksperymentów,
- e) podsumowanie - opis wniosków końcowych.

2. Przegląd literatury

2.1. Transfer learning

Transfer learning to technika uczenia maszynowego, w której model wytrenowany w jednym zadaniu jest ponownie wykorzystywany w drugim powiązanym zadaniu. Wykorzystując tę metodę można otrzymać lepsze wyniki niż trenując model od podstaw, ponieważ przekazujemy wiedzę z podobnego zadania, które zostało już opanowane[1].

Przeprowadzenie transfer learningu obejmuje:

- 1) wybranie spośród dostępnych modeli wstępnie wytrenowanego modelu,
- 2) model można następnie wykorzystać jako punkt wyjścia dla modelu w drugim zadaniu będącym przedmiotem zainteresowania,
- 3) opcjonalnie, model może wymagać dostosowania lub dopracowania parametrów na podstawie danych.

Rozwiążanie oparte o transfer learning eliminuje problem z niedostateczną ilością danych. Dzięki użyciu takiej metody jednostki o mniejszych zasobach mogą konkurować z dużymi instytucjami. Dodatkowo dzięki nieustannie rozwijającym się społeczeństwom np. HuggingFace jest coraz więcej modeli, które można używać.

2.2. Tłumaczenie maszynowe

Tłumaczenie maszynowe to proces wykorzystywania sztucznej inteligencji do automatycznego tłumaczenia tekstu z jednego języka na inny bez udziału człowieka. Nowoczesne tłumaczenie maszynowe wykracza poza proste tłumaczenie słowo w słowo, aby przekazać pełne znaczenie oryginalnego tekstu w języku docelowym. Analizuje wszystkie elementy tekstu i rozpoznaje, w jaki sposób słowa wpływają na siebie nawzajem[2].

Można wyróżnić kilka typów tłumaczenia maszynowego:

- oparte na regułach - wykorzystuje reguły gramatyczne i językowe opracowane przez ekspertów językowych oraz słowniki, które można dostosować do konkretnego tematu lub branży,
- statystyczne - nie opiera się na regułach językowych i słowach. Model uczy się, jak tłumaczyć, analizując dużą liczbę istniejących ludzkich tłumaczeń,

- neuronowe - model uczy się, jak tłumaczyć za pomocą dużej sieci neuronowej. Metoda ta staje się coraz bardziej popularna, ponieważ zapewnia lepsze wyniki w przypadku par językowych[3].

2.3. Mechanizm uwagi

Człowiek podczas minuty rozmowy wypowiada średnio od 125 do 150 słów[4]. Nie jesteśmy w stanie zapamiętać wszystkich słów, pomimo tego odpowiadamy drugiej osobie na zadane pytanie. Dzieje się tak dlatego, że zapamiętujemy kluczowe słowa i formułujemy odpowiedź. W podobny sposób działa mechanizm uwagi. Zamiast skupiać się na każdym słowie, mechanizm uwagi dostarcza modelowi informację na których słowach powinien się skupić w danym momencie.

Funkcję uwagi można opisać jako mapowanie zapytania i zestawu par klucz-wartość na wynik, gdzie zapytanie, klucze, wartości i dane wyjściowe są wektorami. Przedstawiany mechanizm uwagi można nazwać jako „Skalowana uwaga iloczynu punktowego”. Wejście składa się z zapytania oraz klucza o wymiarze d_k i wartości o wymiarze d_v . Obliczane są iloczyny skalarne pomiędzy zapytaniem a kluczami i dzielone przez $\sqrt{d_k}$, a następnie jest zastosowana funkcja softmax. Te kroki zostały zastosowane, aby otrzymać wagi dla poszczególnych wartości[5].

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2.1)$$

gdzie Q – macierz zapytań, K^T – transponowana macierz kluczy, V – macierz wartości, $\sqrt{d_k}$ – pierwiastek z wymiaru kluczy.

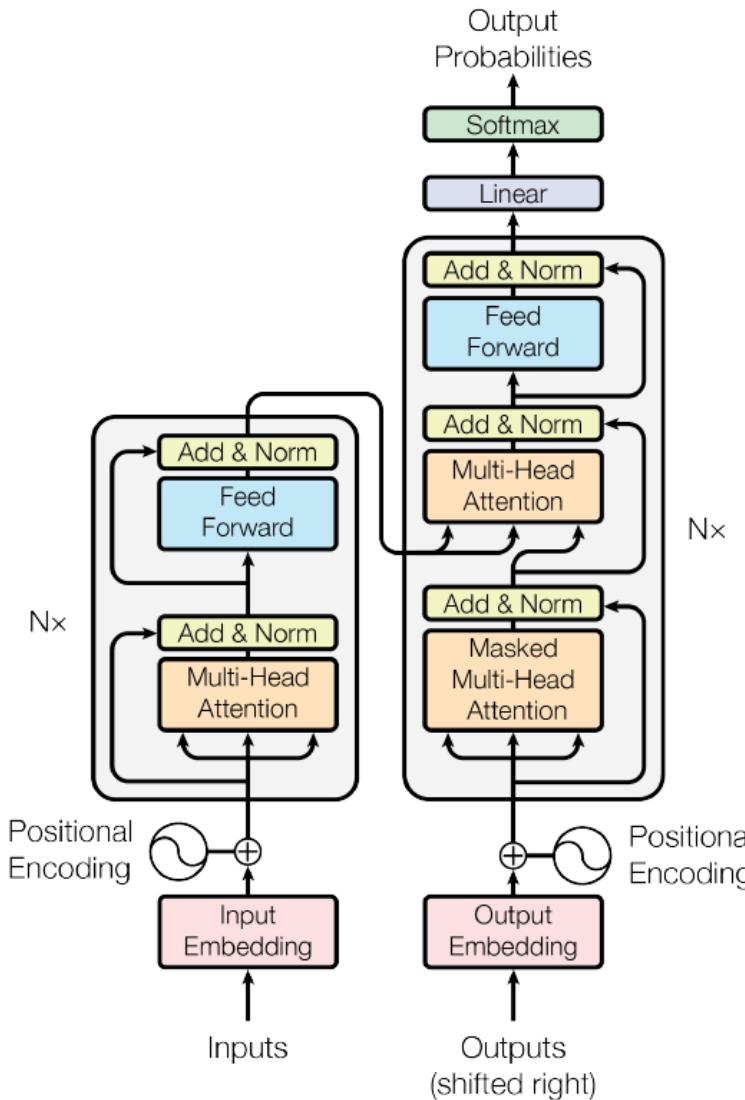
Mechanizm uwagi jest aktualnie wykorzystywany we wszystkich najnowszych architekturach i modelach. Jego popularność motywowana jest małą złożonością obliczeniową, możliwością wykonywania wielu operacji w jednym czasie oraz umiejętności przetwarzania zależności w długich sekwencjach.

2.4. Architektura transfomera

2.4.1. Wstęp

Model składa się z dwóch części. Pierwsza to enkoder, a druga to dekoder. Enkoder mapuje wejściowe sekwencji reprezentacji symboli (x_1, \dots, x_n) na sekwencję reprezentacji ciągłych $z = (z_1, \dots, z_n)$. Następnie dekoder generuje sekwencje wyjściową symboli (y_1, \dots, y_m) , jeden element na raz. Na każdym etapie model wykorzystuje wcze-

śniej wygenerowane symbole jako dodatkowe wejście przy generowaniu kolejnych [5]. Na rysunku 2.1 jest przedstawiona ogólna architektura transfomera.



Rysunek 2.1: Architektura transfomera[5]

2.4.2. Osadzenia

Osadzanie słów lub wektorów słów to podejście, za pomocą którego można reprezentować dokumenty i słowa. Są definiowane jako numeryczny wektor wejściowy, który pozwala słowom o podobnym znaczeniu mieć tę samą reprezentację. Może przybliżać znaczenie i reprezentować słowo w przestrzeni o niższym wymiarze. Rozwiązuje one problem z odpowiednią reprezentacją słów[6]. Dzięki możliwości dostosowywania wymiarów, mogą one reprezentować zawiłe zależności pomiędzy słowami. Przez to model może w bardziej efektywny sposób przetwarzać słowa i wyciągać wnioski na temat

zależności pomiędzy nimi.

2.4.3. Kodowanie pozycyjne

W modelu transfomer nie występują połączenia rekurencyjne, dlatego należy przekazać informacje na temat kolejności występowania tokenów w sekwencji. Zostało to zaimplementowane za pomocą kodowania pozycyjnego. Kodowanie pozycyjne jest dodawane do osadzenia wejściowego, aby przekazać modelowi informacje na temat kolejności tokenów. Wektor kodowania pozycyjnego ma takie same wymiary jak wektor wejściowy, dzięki czemu można w prosty sposób dodać je do siebie. Wzór 2.2 przedstawia kodowanie pozycyjne[5].

$$\begin{aligned} PE_{(pos,2i)} &= \sin(pos/10000^{\frac{2i}{d_{model}}}) \\ PE_{(pos,2i+1)} &= \cos(pos/10000^{\frac{2i}{d_{model}}}) \end{aligned} \quad (2.2)$$

gdzie pos - pozycja tokenu, i - wymiar, d_{model} - wymiar wektora.

Funkcja kodowania położenia zazwyczaj wykorzystuje kombinację funkcji sinus i cosinus o różnych częstotliwościach do kodowania informacji o położeniu. Powodem stosowania zarówno funkcji sinusoidalnych, jak i cosinusoidalnych jest umożliwienie modelowi przechwytywania różnych wzorców i unikania efektów zniekształcania sygnału[7].

2.4.4. Wielogłowicowa uwaga

Mechanizm uwagi został przedstawiony w podrozdziale 2.3. W architekturze transfomera, zamiast pojedynczego mechanizmu uwagi została zastosowana wielogłowicowa uwaga. Dzięki tej metodzie, jest możliwe wykrywanie przez pojedyncze mechanizmy uwagi różnych związków między tokenami. Po obliczeniu pojedynczych uwag, są one dodawane w jeden wynik. Wzór 2.3 przedstawia sposób obliczania wielogłowicowej uwagi[5].

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O \quad (2.3)$$

gdzie $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$.

2.4.5. Zamaskowana wielogłowicowa uwaga

Można zauważyć w architekturze transfomera w dekoderze znajduje się zamaskowana wielogłowicowa uwaga. Dokonuje ona tych samych obliczeń co klasyczny mechanizm uwagi, natomiast z małą modyfikacją. Ze względu na to, że podczas fazy uczenia do transfomera, są dostarczane wszystkie słowa, trzeba zabezpieczyć to, aby

transfomer nie podglądał kolejnych słów. Dlatego w przypadku tłumaczenia słowa 3 w danym ciągu, transfomer powinien "widzieć" tylko pierwsze dwa słowa przykładu referencyjnego i na ich podstawie dokonać tłumaczenia.

2.4.6. Warstwa liniowa i softmax

Warstwa liniowa rzutuje wektor dekodera na wyniki słów, z wartością wyniku dla każdego unikalnego słowa w słowniku docelowym, w każdej pozycji w zdaniu. Na przykład, jeśli końcowe zdanie wyjściowe ma 7 słów, a docelowe słownictwo ma 10000 unikalnych słów, generowane jest 10000 wartości punktowych dla każdego z tych 7 słów. Wartości punktowe wskazują prawdopodobieństwo wystąpienia każdego słowa w słowniku w danej pozycji zdania.

Następnie warstwa softmax przekształca te wyniki w prawdopodobieństwa (które sumują się do 1,0). W każdej pozycji znajduje indeks dla słowa o najwyższym prawdopodobieństwie, a następnie mapuje ten indeks na odpowiadające mu słowo w słowniku. Słowa te tworzą następnie sekwencję wyjściową transfomera[8]. Wzór 2.4 przedstawia sposób obliczania funkcji softmax.

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} \quad (2.4)$$

gdzie x_i - wartość pojedynczego tokenu, $\sum_{j=1}^n e^{x_j}$ - suma wartości wszystkich słów z wykorzystaniem funkcji eksponencjalnej.

2.5. Metryki oceny

2.5.1. Wstęp

Metryki mierzą jakość wyników systemu tłumaczenia maszynowego oraz są używane do porównywania różnych systemów tłumaczenia maszynowego. Metryki oceny są podzielone na wskaźniki oparte na ciągach znaków i wskaźniki oparte na uczeniu maszynowym. Metryki oparte na ciągach znaków zazwyczaj mierzą odległość słowa lub znaku między zdaniem docelowym a tłumaczeniem referencyjnym. Ten typ metryk jest często używany w pracach badawczych, ponieważ są one wytłumaczalne, a także mogą obsługiwać dowolną parę językową. Wyniki generalnie nie korelują dobrze z wynikami oceny ludzkiej, gdy jakość tłumaczenia jest wysoka[10]. Poniżej są przedstawione przykładowe metryki:

- BLEU,

- METEOR,
- NIST,
- TER.

Metryki oparte na uczeniu maszynowym wykorzystują osadzanie zdań do obliczania różnicy między wygenerowanym zdaniem docelowym a tłumaczeniem referencyjnym. Metryki oparte na uczeniu maszynowym wymagają modelu, który został wytrenowany na danych w języku źródłowym i docelowym. Wynik może dobrze korelować z wynikami oceny dokonywanej przez człowieka[10]. Poniżej są przedstawione przykładowe metryki:

- COMET,
- YiSi,
- BERTscore.

2.5.2. BLEU

Wynik BLEU mierzy podobieństwo między tekstem przetłumaczonym maszynowo a tłumaczeniami referencyjnymi przy użyciu n-gramów, które są ciągłymi sekwencjami n słów. Najczęściej używanymi n-gramami są unigramy (pojedyncze słowa), bigramy (sekwencje dwóch słów), trigramy (sekwencje trzech słów) itd. Metryka składa się z dwóch składowych. Pierwsza to obliczanie precyzji dopasowania słów pomiędzy zdaniem przetłumaczonym oraz referencjami. Druga to obliczanie kary za zwięzłość. Ze względu na to, że modele translacyjne w łatwy sposób mogłyby próbować zakłamywać swoje predykcje (np. poprzez generowanie jednego słowa kilkukrotnie) wprowadzony został zmodyfikowany sposób zliczania dopasowań. Wzór 2.5 przedstawia sposób zapobiegania przed ww. sytuacją.

$$Count_{clip} = \min(Count, Max_Ref_Count) \quad (2.5)$$

gdzie $Count$ – zliczanie wystąpień słowa w zdaniach referencyjnych, Max_Ref_Count – maksymalna ilość wystąpień słowa w jednym ze zdań referencyjnych.

Precyzję oblicza się po przez zliczenie słów oraz zastosowanie zmodyfikowanej wersji sposobu zliczania, a następnie podzielenie przez liczbę słów w przykładzie wygenerowanym przez model. Wzór 2.6 przedstawia uogólnioną wersję sposobu obliczania

dla kilku przykładów testowych.

$$p_n = \frac{\sum_{C \in Candidates} \sum_{n-gram \in C} Count_{clip}(n-gram)}{\sum_{C' \in Candidates} \sum_{n-gram' \in C'} Count(n-gram')} \quad (2.6)$$

Kolejną częścią metryki BLEU jest kara za zwięzłość. Karze ona teksty, które są krótsze od tekstów referencyjnych. Nie trzeba karać modelu za generowanie słów dłuższych niż tekst referencyjny. Jest to zaimplementowane w zmodyfikowanej funkcji zliczania. Wzór 2.7 przedstawia sposób obliczania kary za zwięzłość.

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{dla } c \leq r \end{cases} \quad (2.7)$$

gdzie c – długość frazy przetłumaczonej, r – długość frazy referencyjnej.

Ostateczna wersja metryki BLEU bierze pod uwagę kilka n-gramów i wylicza z nich średnią geometryczną. Wzór 2.8 przedstawia finalną wersję metryki BLEU[10].

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^N w_n \log p_n\right) \quad (2.8)$$

gdzie BP - funkcja obliczająca karę za zwięzłość, w_n - wagi dla każdego z n-gramów, p_n - zmodyfikowana funkcja zliczająca.

3. Opis wykorzystanych narzędzi

3.1. Tensorflow

TensorFlow to darmowa biblioteka oprogramowania do uczenia maszynowego stworzona przez Google. TensorFlow wywodzi swoją nazwę od wielowymiarowych tablic znanych jako tensory, które są wykorzystywane przez sieci neuronowe do różnych operacji. Został stworzony głównie do głębokich badań sieci neuronowych i do ułatwienia uczenia maszynowego, chociaż TensorFlow był również stosowany w wielu innych obszarach[12].

TensorFlow przedstawia obliczenia za pomocą grafów. Wierzchołki grafów odpowiadają za przedstawienie działań matematycznych, natomiast krawędzie grafów to tensory, które łączą wierzchołki. TensorFlow przyjmuje dane na wejściu jako wielowymiarową tablice o nazwie Tensor, następnie te dane są przetwarzane przez TensorFlow[13].

3.2. Transfomers i HuggingFace

Hugging Face to społeczność i platforma nauki o danych, która zapewnia:

- narzędzia, które umożliwiają użytkownikom budowanie, trenowanie i wdrażanie modeli ML w oparciu o kod i technologie open source,
- miejsce, w którym szeroka społeczność naukowców zajmujących się danymi, badaczy i inżynierów ML może spotykać się i dzielić pomysłami, uzyskiwać wsparcie i wnosić wkład w projekty open source[14].

Transformers zapewnia interfejsy API i narzędzia do łatwego pobierania i trenowania najnowocześniejszych wstępnie wytrenowanych modeli. Korzystanie ze wstępnie wytrenowanych modeli może zmniejszyć koszty obliczeniowe, ślad węglowy oraz oszczędzić czas i zasoby wymagane do wytrenowania modelu od podstaw. Modele te obsługują typowe zadania w różnych modalnościach, takich jak:

- Przetwarzanie języka naturalnego: klasyfikacja tekstu, rozpoznawanie nazwanych encji, odpowiadanie na pytania, modelowanie języka, podsumowywanie, tłumaczenie, wielokrotny wybór i generowanie tekstu.
- Wizja komputerowa: klasyfikacja obrazów, wykrywanie obiektów i segmentacja.

- Audio: automatyczne rozpoznawanie mowy i klasyfikacja audio.
- Multimodalne: odpowiadanie na pytania w tabelach, optyczne rozpoznawanie znaków, wydobywanie informacji z zeskanowanych dokumentów, klasyfikacja wideo i wizualne odpowiadanie na pytania[15].

3.3. Docker

Docker to otwarta platforma do tworzenia, dostarczania i uruchamiania aplikacji. Docker umożliwia oddzielenie aplikacji od infrastruktury, dzięki czemu można szybko dostarczać oprogramowanie. Docker zapewnia możliwość pakowania i uruchamiania aplikacji w izolowanym środowisku zwany kontenerem. Izolacja i bezpieczeństwo pozwalają uruchamiać wiele kontenerów jednocześnie na danymホście. Kontenery są lekkie i zawierają wszystko, co potrzebne do uruchomienia aplikacji, więc nie trzeba polegać na tym, co jest zainstalowane naホście[16]. Dodatkowo Docker Hub dostarcza gotowe obrazy, które można wykorzystać podczas budowania własnego środowiska. Ułatwia to pracę z Dockerem, ponieważ można pobrać dowolny obraz i zainstalować tylko potrzebne biblioteki.

3.4. Neptune

Neptune to narzędzie do śledzenia eksperymentów. Oferuje możliwość rejestracji, porównywania, przechowywania i współpracy nad eksperymentami i modelami[17]. Dodatkowo narzędzie umożliwia tworzenie własnych wykresów lub tabel, które mogą ułatwić analizę parametrów. Dzięki zaimplementowanym klasom można w prosty sposób przechwytywać parametry modelu podczas uczenia. Neptune jest kompatybilny z PyTorchem i TensorFlow, dodatkowo dostarcza wywołania zwrotne, dzięki którym parametry są przechwytywane automatycznie.

4. Opis rozwiązania

4.1. Stworzenie środowiska

Przed przystąpieniem do implementacji rozwiązania zostało stworzone i skonfigurowane środowisko. Aby zapewnić wyodrębnienie projektu oraz odtwarzalność, jako biblioteki do implementacji środowiska zostały wybrane Docker oraz Docker-Compose. Pierwszym krokiem było stworzenie lub wybranie obrazu bazowego. Jako obraz bazowy, który był nadbudowywany odpowiednimi bibliotekami został wybrany jupyter/minimal-notebook. Został on wybrany ze względu na mały rozmiar oraz przez dostarczenie wszystkich funkcjonalności, które były potrzebne do realizacji projektu. Stworzenie środowiska obejmowało poniższe kroki:

- zdefiniowanie pliku Dockerfile,
- opracowanie pliku z bibliotekami do zainstalowania,
- stworzenie pliku docker-compose.yaml.

Plik Dockerfile jest odpowiedzialny za zbudowanie odpowiedniego obrazu wraz z zainstalowaniem bibliotek. Listing 1 przedstawia zimportowanie odpowiedniego obrazu, następnie stworzenie folderu głównego. Następnie zostały skopowane wszystkie potrzebne biblioteki do tego folderu oraz za pomocą biblioteki pip zostały zainstalowane wszystkie potrzebne biblioteki do realizacji projektu. Ostatnia linia kodu odpowiada za ustalanie tokenu do jupyter notebook. Bez ustalania tego tokenu przy ponownym uruchomieniu projektu należy podać token, który jest generowany za każdym razem. Plik req.txt zawiera nazwy oraz wersje bibliotek, które są instalowane.

```
1 FROM jupyter/minimal-notebook
2 WORKDIR /code
3 COPY req.txt /code/
4 RUN pip install -r req.txt
5 COPY . /code/
6 ENTRYPOINT ["start.sh", "jupyter", "lab", "--LabApp.token='']
```

Listing 1: Plik Dockerfile

Ostatnim krokiem jest zdefiniowanie pliku docker-compose.yaml. Dzięki niemu można używać paru serwisów naraz oraz odpowiada za automatyzację procesu ponownego włączania i budowania obrazów. W prosty sposób można dobudować do aktualnego

rozwiązań prosty interfejs i połączyć te dwa rozwiązania. Listing 2 przedstawia implementacje serwisu, który obsługuje projekt. W tym pliku zdefiniowano odpowiedni folder przy metodzie build, aby była możliwość budowania obrazu. Określa wolumen, gdzie są składowane dane oraz port na którym powinno zostać uruchamiane środowisko.

```
1 version: "3.3"
2
3 services:
4   jupyter:
5     build: ./ai
6     volumes:
7       - ./ai:/code
8     ports:
9       - "8888:8888"
```

Listing 2: Plik docker-compose.yaml

4.2. Format danych

Dane były składowane w formacie .odt. Każde zdanie do przetłumaczenia było umieszczone w jednej linii od myślnika. Tłumaczenia były umieszczone pod zdaniem rozpoczynając i kończąc się od nawiasów kwadratowych, dzięki czemu przy kilku tłumaczeniach w prosty sposób można było pobrać dane. Poniżej został zaprezentowany format danych.

- zdanie,
- [,
- tłumaczenie 1,
- tłumaczenie 2,
- ...,
- tłumaczenie n,
-],

Format danych, który przyjmował model różnił się od formatu danych początkowych. Model przyjmował strukturę danych Dataset z biblioteki Transfomer. Dane były uporządkowane w liście słowników, gdzie każdy słownik zawierał jeszcze jeden słownik. Poniżej został zaprezentowany przykład, w jaki sposób dane zostały uporządkowane.
{'translation' : {'pl' : zdanie, 'mig' : tłumaczenie}}

4.3. Wczytanie modelu

Model użyty w projekcie był pobierany z platformy HuggingFace. Aby zoptymalizować proces używania tego modelu (nie pobierać modelu co każde włączenie środowiska), został napisany krótki skrypt, w którym model jest pobierany i zapisywany lokalnie. Od tego momentu można włączać model lokalnie, gdzie są pobrane wszystkie wagi oraz informację przekazywaną do modelu. Na listingu 3 jest przedstawiony proces wczytywania oraz zapisywania modelu. Aby pobrać model oraz tokenizera z platformy HuggingFace, zostały użyty dwie klasy AutoTokenizer oraz TFAutoModelForSeq2SeqLM, do których została przekazana nazwa odpowiedniego modelu. Następnie aby zapisać model oraz tokenizera lokalnie, została wykorzystana metoda save_pretrained oraz zostały wskazane odpowiednie foldery.

```
1 from transformers import AutoTokenizer, TFAutoModelForSeq2SeqLM
2
3 tokenizer = AutoTokenizer.from_pretrained("Helsinki-NLP/opus-mt-
4     pl-en")
5 model = TFAutoModelForSeq2SeqLM.from_pretrained("Helsinki-NLP/
6     opus-mt-pl-en")
7
8 tokenizer.save_pretrained("./model/tokenizer/")
9 model.save_pretrained("./model/model.h5")
```

Listing 3: Wczytanie i zapisanie modelu

Po wczytaniu oraz zapisaniu modelu, należało sprawdzić czy ten model działa poprawnie (czy wagi oraz tokenizator zostało poprawnie pobrany). Listing 4 przedstawia próbę translacji pomiędzy językiem polskim, a angielskim. Pierwszym krokiem było zaimportowanie obiektu pipeline, który umożliwia prostą implementację translacji. Następnie z lokalnych folderów zostały pobrane model oraz tokenizator. Ostatnim krokiem było przekazanie do pipeline modelu oraz tokenizera i poinformowanie o rodzaju zadania (w tym przypadku jest to tłumaczenie). W 7 linii kodu zostało przeprowadzone tłumaczenie zdania z języka polskiego na angielski.

```
1 from transformers import pipeline
2
3 model = TFAutoModelForSeq2SeqLM.from_pretrained("./model/model.
4     h5")
5 tokenizer = AutoTokenizer.from_pretrained("./model/tokenizer")
6
7 translation = pipeline("translation", model=model, tokenizer=
8     tokenizer)
9 translation("Lubie jesc jablka!")
```

Listing 4: Sprawdzenie działania modelu

4.4. Przygotowanie danych

4.4.1. Wczytanie danych

Dane były składowane formacie .odt. Dla tego formatu nie ma standardowych funkcji w bibliotece Pandas, dlatego została użyta biblioteka odf. Listing 5 przedstawia zainportowanie odpowiednich metod, stworzenie funkcji przetwarzającej wczytywanie danych oraz usuwanie z listy pustych ciągów znaków. Funkcja load_data przyjmuje jeden argument, który jest ścieżką do pliku. Następnie przetwarza plik i zwraca listę, w którym są wczytywane wiersze z dokumentu.

```
1 from odf.opendocument import load
2 from odf import text, teletype
3
4 def load_data(filepath):
5     raw_data = []
6     text_doc = load(filepath)
7     all_params = text_doc.getElementsByType(text.P)
8     for line in all_params:
9         raw_data.append(teletype.extractText(line))
10    return raw_data
11
12 raw_data = load_data('dataset/data.odt')
13 while '' in raw_data:
14     raw_data.remove('')
```

Listing 5: Wczytanie danych

Rysunek 4.2 przedstawia dane po wczytaniu z pliku. Są one początkowo składowane w liście. Struktura danych została dokładniej wytłumaczona w podrozdziale 4.2. Dwie pierwsze wiersze w liście przedstawiają nagłówki, które zostały pominięte. Nie wnoсиły one żadnych informacji do danych. Ze względu na regularną strukturę ułożenia danych, późniejsze przetwarzanie danych było możliwe do zautomatyzowania.

```
[ 'Wypowiedzi OG',
  'Przywitanie',
  '- Dzień dobry.',
  '[',
  'dzień dobry',
  'witać',
  ']',
  'Wyjaśnienie powodu wizyty',
  '- Chcę złożyć wniosek o wydanie dowodu osobistego.',
  '[',
  'ja wniosek dowód1 mieć',
  'ja wniosek dowód2 mieć',
  ']',
  '- Czy mogę odebrać dowód?',
  '[',
  'czy już nowy dowód1',
  'czy już nowy dowód2',
  'być mój nowy dowód1',
  'być mój nowy dowód2',
  ']',
```

Rysunek 4.2: Przykład wczytanych danych

4.4.2. Dostosowywanie formatu danych

W celu poprawnego zbudowania struktury danych wejściowych dla modelu, pierwszym krokiem było stworzenie funkcji do podzielenia danych na przykłady. Problem jaki trzeba było rozwiązać to, każdy przykład tłumaczenia należało podzielić na oddzielny przykład wejściowy, aby stworzyć pary zdanie-tłumaczenie. Listing 6 przedstawia stworzenie funkcji, zastosowania funkcji do podziału danych oraz stworzenie struktury DataFrame do zaprezentowania danych. Funkcja `split_data_from_list` przyjmuje jako argument wejściowy wczytany plik, a następnie iteruje po wierszach tego pliku. Przeszukiwanie pliku ma na celu znalezienie pierwszego nawiasu kwadratowego oraz dodaniu tłumaczeń do listy dopóki nie znajdzie nawiasu zamykającego. W 18 linii kodu została zastosowana ww. funkcja i zostały stworzone dwie listy z zdania oraz tłumaczeniami. Ostatnim etapem było stworzenie struktury DataFrame w której wierszami są pary zdanie-tłumaczenie, a kolumnami odpowiednie oznaczenia. Dla języka polskiego pl, natomiast dla języka migowego mig. Dane zostały zaprezentowane w tej strukturze, aby ułatwić późniejsze przetwarzanie oraz prezentacje danych.

```
1 def split_data_from_list(raw_data):
2     pl_sentence = []
3     sentence = []
4     i=0
5     while i < len(raw_data):
6         if raw_data[i+1] == '[':
```

```

7     value = raw_data[i]
8     i += 2
9     while i < len(raw_data):
10        if raw_data[i] == ']':
11            break
12        pl_sentence.append(value[1:])
13        sentence.append(raw_data[i])
14        i += 1
15    i += 1
16    return pl_sentence, sentence
17
18 s1, s2 = split_data_from_list(raw_data)
19
20 data = pd.DataFrame({'pl':s1, 'mig':s2})
21 print(data[0:10])

```

Listing 6: Podzielenie danych na pary przykładów

Na rysunku 4.3 została przedstawiona struktura danych (dokładnie par zdanie-tłumaczenie) po przetworzeniu oraz zapisaniu w obiekcie DataFrame. Można zauważyć, że dzięki takiemu zapisowi w łatwy sposób można zweryfikować czy dane są kompletne. Wykorzystany zbiór danych zawierał jedną pustą linie, która dzięki zweryfikowaniu danych została usunięta. Mogło to wpływać negatywnie na działanie modelu.

	pl	mig
	Dzień dobry.	dzień dobry
	Dzień dobry.	witać
Chce złożyć wniosek o wydanie dowodu osobistego.	ja wniosek dowód1 mieć	
Chce złożyć wniosek o wydanie dowodu osobistego.	ja wniosek dowód2 mieć	
	Czy mogę odebrać dowód?	czy już nowy dowód1
	Czy mogę odebrać dowód?	czy już nowy dowód2
	Czy mogę odebrać dowód?	być mój nowy dowód1
	Czy mogę odebrać dowód?	być mój nowy dowód2
Chce zgłosić utratę dowodu osobistego.	ja dowód1 zgubić	
Chce zgłosić utratę dowodu osobistego.	ja dowód2 zgubić	

Rysunek 4.3: Przykład par zdanie-tłumaczenie

Ostatnim etapem dostosowania formatu danych było przekształcenie danych ze struktury DataFrame na strukturę Dataset z biblioteki Transformer. Jest to struktura, którą model może przyjąć jako dane wejściowe. Ze względu na wykorzystanie wcześniej trenowanego modelu, dane muszą być przekazywane do modelu w takim samym formacie, jak podczas pierwszego treningu. Listing 7 przedstawia transformacje danych z DataFrame na Dataset.

```

1 raw_dataset_list = []
2 for i in range(0, len(data)):

```

```

3     raw_dataset_list.append({'translation' : {'pl' : data['pl'][i],
4                               'mig' : data['mig'][i]}})
5 raw_dataset = Dataset.from_list(raw_dataset_list)

```

Listing 7: Tworzenie struktury Dataset

4.4.3. Podział danych na zbiór treningowy i testowy

Przed przystąpieniem do tokenizowania i indeksowania danych, należało podzielić je na zbiór treningowy i testowy. W projekcie nie został wydzielony zbiór walidacyjny, ze względu na małą ilość danych. Przy ilości 123 przykładów zbiór walidacyjny nie spełniał by swoich założeń. Mogło by to jedynie wpłynąć negatywnie na fazę treningu, ponieważ model miałby mniej danych jako przykłady uczące. Do podzielenia danych została wykorzystana funkcja z biblioteki Transfomer. Do funkcji zostały przekazane dwa parametry. Pierwszy to wielkość zbioru testowego, która została ustawiona na 20%. Taka wartość parametru może wyodrębnić wystarczający zbiór testowy, aby był reprezentatywny. Drugi parametr to ziarno losowości, które umożliwia otrzymanie tego samego podziału danych przy każdym wywołaniu kodu. Listing 8 przedstawia podział danych na zbiór treningowy i testowy oraz stworzenie z tych dwóch zbiorów struktury DatasetDict.

```

1 train_test = raw_dataset.train_test_split(test_size=0.2, seed
2                               =42)
3
3 train_test_dataset = DatasetDict({
4     'train': train_test['train'],
5     'test': train_test['test']})

```

Listing 8: Podział danych

4.4.4. Tokenizowanie

Po podziale na zbiory treningowy i testowy należało przeprowadzić tokenizację i indeksowanie. Aby to wykonać została stworzona prosta funkcja. Do wybrania odpowiednich zdań z podziałem na język polski i migowy, zostało stworzone proste wyrażenie listowe, które iteruje po każdym zdaniu i wybiera odpowiednich język dla danych wejściowych i wyjściowych. Maksymalna długość sekwencji została ograniczona do 32 tokenów, ze względu na krótkie przykłady uczące. Model dopuszcza sekwencje o maksymalnej długości 512 tokenów. Listing 9 przedstawia funkcję, która tokenizuje i indeksuje dane, następnie w 7 linii kodu jest ona mapowana na zbiór danych w celu przeprowadzenia transformacji. Aby przyśpieszyć działanie funkcji, parametr batched

został ustawiony na wartość True, dzięki temu funkcja nie transformuje pojedynczo przykładów tylko w grupach.

```
1 train_test = raw_dataset.train_test_split(test_size=0.2, seed  
    =42)  
2  
3 train_test_dataset = DatasetDict({  
    'train': train_test['train'],  
    'test': train_test['test']})  
4  
5 tokenized_dataset = train_test_dataset.map(preprocess_function,  
    batched=True)
```

Listing 9: Tokenizowanie

Ostatnim krokiem w przygotowaniu danych jest wyrównanie długości wszystkich przykładów, przetasowanie ich oraz pogrupowanie. Listing 10 przedstawia finalne przetworzenie danych, które umożliwi skuteczne dotrenowanie modelu. Klasa DataCollatorForSeq2Seq jest odpowiedzialna za dynamiczne dodawanie tokenów, aby wszystkie sekwencje miały ten sam rozmiar. Wielkość każdej z grup została określona na 8 ze względu na małą ilość przykładów uczących.

```
1 batch_size = 8  
2  
3 data_collator = DataCollatorForSeq2Seq(tokenizer, model=model,  
    return_tensors="tf")  
4 train_dataset = model.prepare_tf_dataset(  
    tokenized_dataset["train"],  
    batch_size=batch_size,  
    shuffle=True,  
    collate_fn=data_collator,  
)  
5  
10 test_dataset = model.prepare_tf_dataset(  
    tokenized_dataset["test"],  
    batch_size=batch_size,  
    shuffle=False,  
    collate_fn=data_collator,  
)
```

Listing 10: Przygotowanie danych

4.5. Trenowanie modelu

4.5.1. Przygotowanie modelu

Przed przystąpieniem do fazy treningu została przeprowadzona inspekcja modelu. Aby móc efektywnie przeprowadzić trening należy zapoznać się ze strukturą modelu oraz z jego parametrami. Na rysunku 4.4 została przedstawiona ilość parametrów

z podziałem na możliwe do wytrenowania oraz parametry stałe.

```
Model: "tf_marian_mt_model_2"
=====
Layer (type)          Output Shape         Param #
=====
model (TFMarianMainLayer)    multiple      77138944
final_logits_bias (BiasLayer) multiple      63430
=====
Total params: 77,202,374
Trainable params: 77,138,944
Non-trainable params: 63,430
```

Rysunek 4.4: Struktura modelu

Poniżej zostały przedstawiona przykładowe parametry modelu:

- liczba warstw w dekoderze - 6,
- liczba warstw w enkoderze - 6,
- liczba głowic uwagi w dekoderze - 8,
- liczba głowic uwagi w enkoderze - 8,
- wielkość słownika - 63420.

4.5.2. Śledzenie parametrów modelu

W celu przeprowadzenie analizy poprawności działania modelu do śledzenia parametrów została wykorzystana platforma Neptune. Aby skorzystać z niej należy zainicjować połączenie przez API ze stworzonym projektem. Listing 11 przedstawia zainicjowanie połączenie z platformą oraz stworzenie wywołania zwrotnego. Wymaganymi argumentami są nazwa projektu oraz token API. Parametr tag jest opcjonalny, natomiast może pomóc w późniejszej analizie poszczególnych treningów. Dzięki temu parametrowi można oznaczyć konkretny trening.

```
1 run = neptune.init_run(
2     project=nazwa_projektu,
3     api_token=api_token,
4     tags=opcjonalnie,
5 )
6
7 neptune_callback = NeptuneCallback(run=run)
```

Listing 11: Inicjowanie połączenia z platformą Neptune

4.5.3. Faza treningu

Przed rozpoczęciem treningu został ustawiony optymalizator oraz metryka do sprawdzania dokładności modelu. Została użyta funkcja optymalizująca Adam oraz dokładność jako metryka oceny modelu. Aby zapobiec przetrenowaniu modelu zostało wykorzystane wywołanie zwrotne, które zatrzymuje treningu modelu w przypadku braku zmniejszania się straty przez 5 epok. W ostatniej linii kodu znajduje się wywołanie funkcji, która zatrzymanie śledzenia parametrów przez platformę Neptune. Alternatywnym rozwiązaniem było by zatrzymanie bezpośrednio z platformy, natomiast jest to bardziej problematyczne rozwiązanie. Wprowadzenie zatrzymania działania wywołania zwrotnego w kodzie daje większą kontrolę nad śledzeniem parametrów. Listing 12 przedstawia implementację fazy treningu.

```
1 optimizer = keras.optimizers.Adam(learning_rate=0.00005)
2 model.compile(optimizer=optimizer, metrics=["accuracy"])
3
4 early_stopping_callback = keras.callbacks.EarlyStopping(monitor=
5   'loss', patience=5)
6 model.fit(train_dataset, epochs=100, callbacks=[
7   early_stopping_callback, neptune_callback])
8 run.stop()
```

Listing 12: Faza treningu

Rysunek 4.5 przedstawia fazę uczenia na przykładzie kilkunastu epok. Można zauważyć poprawne działanie wywołania zwrotnego wczesnego zatrzymania. W ostatnich 5 epokach strata się nie zmniejszała, więc trening modelu został przerwany. Wywołanie zwrotne oprócz samego zatrzymania zapamiętuje ostatnią najlepszą wersję modelu, dzięki temu można ją odtworzyć. Tensorflow również dostarcza informacji na temat ilości grup oraz czasu przetworzenia epoki oraz poszczególnych kroków.

```

Epoch 24/100
12/12 [=====] - 10s 814ms/step - loss: 0.1709 - accuracy: 0.5562
Epoch 25/100
12/12 [=====] - 9s 797ms/step - loss: 0.1683 - accuracy: 0.6231
Epoch 26/100
12/12 [=====] - 10s 806ms/step - loss: 0.1673 - accuracy: 0.5636
Epoch 27/100
12/12 [=====] - 11s 883ms/step - loss: 0.1659 - accuracy: 0.5411
Epoch 28/100
12/12 [=====] - 10s 866ms/step - loss: 0.1472 - accuracy: 0.5878
Epoch 29/100
12/12 [=====] - 10s 848ms/step - loss: 0.1453 - accuracy: 0.5682
Epoch 30/100
12/12 [=====] - 10s 843ms/step - loss: 0.1615 - accuracy: 0.5564
Epoch 31/100
12/12 [=====] - 11s 903ms/step - loss: 0.1535 - accuracy: 0.5718
Epoch 32/100
12/12 [=====] - 11s 903ms/step - loss: 0.1617 - accuracy: 0.5478
Epoch 33/100
12/12 [=====] - 11s 907ms/step - loss: 0.1546 - accuracy: 0.5538
Epoch 34/100
12/12 [=====] - 11s 940ms/step - loss: 0.1596 - accuracy: 0.5526

```

Rysunek 4.5: Faza uczenia

4.5.4. Oszacowanie dokładności modelu

Ostatnim etapem była ocena dokładności modelu. Zostało to wykonane za pomocą obliczanie metryki BLEU oraz za pomocą egzaminowania poszczególnych tłumaczeń. Dzięki temu można mieć szerszy pogląd na popełniane przez model błędy. Jest to szczególnie ważne, aby w przyszłości zwiększyć dokładność modelu. Do wyliczenia metryki BLEU została wykorzystana gotowa funkcja z biblioteki NLTK. Przed przystąpieniem do obliczanie metryki BLEU, należało przygotować odpowiednio dane. Ze względu na to, że każde zdanie ma kilka tłumaczeń i na początku zostały one rozłożone na oddzielne przykłady uczące trzeba było znów je połączyć dla każdego tłumaczenia. Dodatkowo każde tłumaczenie należało przyporządkować do odpowiedniego tłumaczenia wygenerowanego przez model.

```

1 def prepare_label_to_bleu(raw_data):
2     hashmap = {}
3     i=0
4     while i < len(raw_data):
5         if raw_data[i+1] == '[':
6             value = raw_data[i]
7             temp_list = []
8             i += 2
9             while i < len(raw_data):
10                 if raw_data[i] == ']':
11                     hashmap[value[1:]] = temp_list
12                     break
13                 temp_list.append(raw_data[i])
14                 i += 1
15             i += 1

```

```
16     return hashmap
```

Listing 13: Przygotowanie danych dla BLEU

Listing 13 przedstawia funkcję, która jest odpowiedzialna za przygotowanie etykiet do obliczania metryki BLEU. Etykiety są przechowywane w słowniku. Słownik składa się z klucza, w którym jest zapisane zdanie do przetłumaczenia, natomiast wartością jest lista z tłumaczeniami. Użycie słownika optymalizuje to rozwiązanie, ze względu na to, że nie trzeba iterować i szukać tłumaczeń, wystarczy jako klucz przekazać zdanie do przetłumaczenia. Funkcja została podobnie zaimplementowana jak funkcja do tworzenia par przykładów zdanie-tłumaczanie. Natomiast została ona zmodyfikowane o dodawanie do listy tłumaczeń, zamiast rozkładaniu tłumaczeń na pojedyncze przykłady.

Ze względu na chęć sprawdzenia poszczególnych tłumaczeń modelu i porównania z tłumaczeniami referencyjnymi została stworzona dodatkowa funkcja. Funkcja dla każdego przykładu testowego generuje tłumaczenie, a następnie zapisuje je w liście, która później będzie przekazywana do funkcji corpus_bleu oraz do struktury DataFrame. Struktura ta będzie zwracana wraz z wynikiem metryki w celu szczegółowej analizy wyników.

```
1 def bleu_score():
2     ref_sent = prepare_label_to_bleu(raw_data)
3     translation_corpus = []
4     reference_corpus = []
5     for i in range(0, len(train_test_dataset["test"]["translation"])):
6         value = ref_sent[train_test_dataset["test"]["translation"]
7                           ][i][ "pl" ]
8         translation = translator(train_test_dataset["test"]["translation"]
9                                   )[i][ "pl" ]
10        translation_corpus.append(translation[0][ "translation_text" ].split())
11        if len(value) > 1:
12            reference_corpus.append([j.split() for j in value])
13        else:
14            reference_corpus.append(value[0].split())
15    b_score = corpus_bleu(reference_corpus, translation_corpus)
16    compare_ref_trans = pd.DataFrame({ "reference" :
17                                         reference_corpus, "translation" :
18                                         translation_corpus})
19    return b_score, compare_ref_trans
20
21 score, df_ref_trans = bleu_score()
22 print(f"BLEU score: {np.around(score, 2)}")
```

Listing 14: Obliczenie metryki BLEU

Listing 14 przedstawia implementacje funkcji do obliczanie metryki BLEU oraz zwracania struktury DataFrame z tłumaczeniami modelu oraz tłumaczeniami referencyjnymi. Najlepszy wynik metryki BLEU wyniósł 0.33.

	reference	translation
0	[[co, pisać, mieć], [jak, pisać, mieć]]	[jak, pisać, mieć]
1	[przyjść, papier, wniosek, mieć]	[tu, zmiana, dowód2]
2	[[ile, płacić, dowód1], [ile, płacić, dowód2]]	[nowy, dowód2, płacić, ile]
3	[[mój, dowód1, pękać], [mój, dowód2, pękać], [...	[mój, dowód2, psuć]
4	[[ile, płacić, dowód1], [ile, płacić, dowód2]]	[nowy, dowód2, płacić, ile]
5	[[dowód1, gotowy], [dowód2, gotowy]]	[być, mój, nowy, dowód2]

Rysunek 4.6: Faza uczenia

Rysunek 4.6 przedstawia dane zwrócone z funkcji obliczającej metrykę BLEU. Tłumaczenia referencyjne są umieszczone w lewej kolumnie, natomiast w prawej są umieszczone tłumaczenia zwrócone przez model. Taki sposób prezentacji danych może ułatwić weryfikację z jakimi frazami model sobie nie radzi.

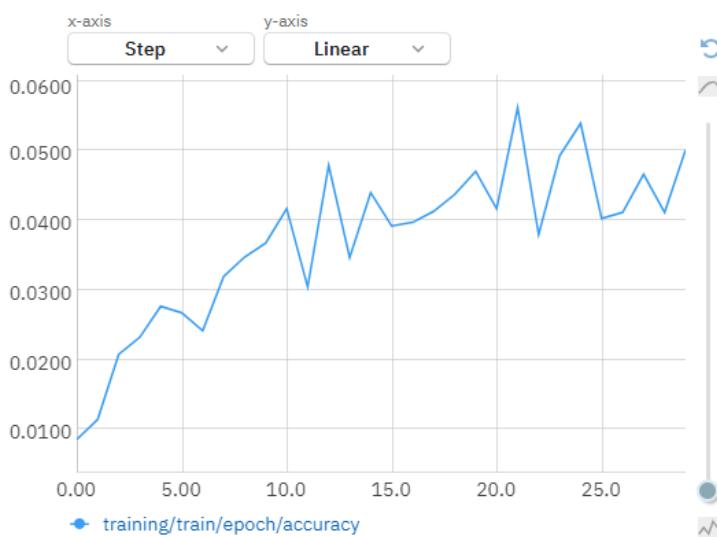
5. Eksperymenty

5.1. Założenia

Eksperymenty na hiperparametrach modelu zostały wykonane bez specjalistycznych metod takich jak przeszukiwanie siatki itp. Zostały dobierane intuicyjnie, natomiast ten sposób również wpływał na poprawę tłumaczenia modelu. Prawdopodobnie użycie konkretnych metod mogłoby zwiększyć skuteczność modelu, ze względu na ograniczenia w postaci małej liczby przykładów uczących nie zostało to zaimplementowane. Przetestowano 5 różnych wartości współczynnika uczenia. Wszystkie dane z eksperymentów zostały składowane na platformie Neptune.

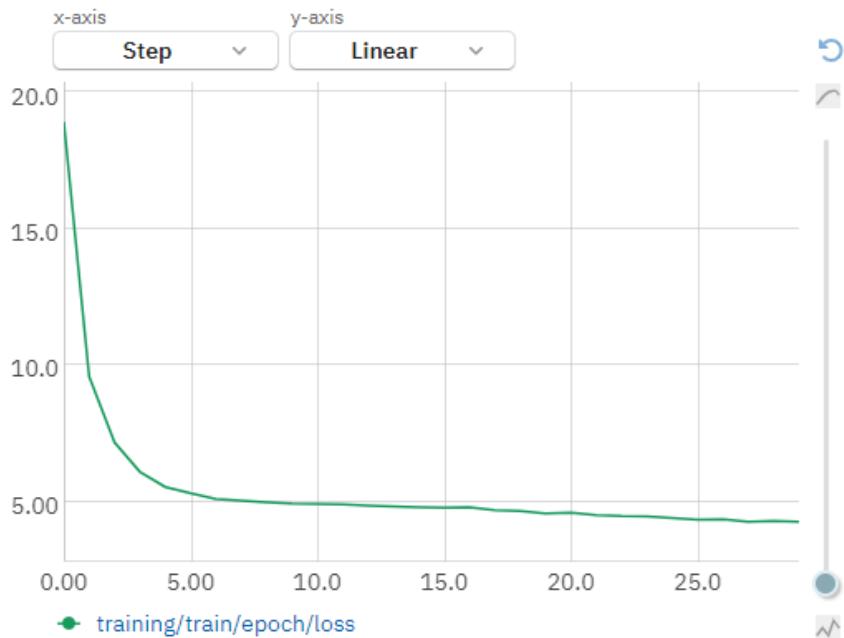
5.2. Współczynnik uczenia

Pierwsza wartość współczynnika uczenia jaka została sprawdzona wynosi 0.01. Została ona dobrana bez większego zagłębiania się w idee uczenia transferowego. Natomiast należy pamiętać o ustalaniu względnie małych wartości przy tego typu uczeniu, ponieważ celem tej metody jest tylko zmodyfikowanie aktualnych wag. Rysunek 5.7 przedstawia w jaki sposób zmieniała się dokładność modelu dla współczynnika uczenia 0.01, podczas treningu. Ze względu na brak zadowalających wyników ilość epok została ustawniona na 30 i nie była zwiększana. Oś y przedstawia dokładność modelu, a osł x przedstawia liczbę epok.



Rysunek 5.7: Dokładność modelu podczas treningu

Rysunek 5.8 przedstawia zmiany funkcji straty podczas treningu. Na osi y jest umieszczona wartość funkcji straty natomiast na osi x jest umieszczona liczba epok.



Rysunek 5.8: Strata modelu podczas treningu

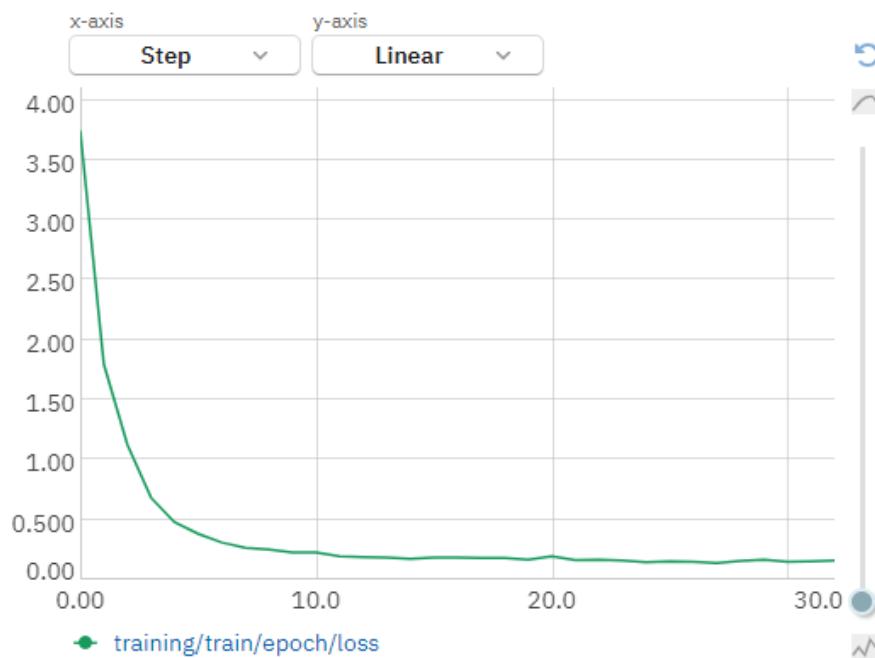
Można zauważyć, że jest to zdecydowanie za duża wartość współczynnika uczenia. Ma to wpływ na dużą wartość funkcji straty oraz bardzo małą dokładność. Jest to spowodowane tym, że wagi nie są modyfikowane tylko zmieniane, przez co model nie korzysta z uczenia transferowego. Średnia dokładność wyniosła około 0.04 oraz średnia funkcja straty wyniosła około 4.30.

Ze względu na poprzedni eksperyment wartość współczynnika uczenia została zdecydowanie zmniejszona. Kolejny eksperyment został przeprowadzony na wartości 0.0001. Rysunek 5.9 przedstawia dokładność modelu podczas treningu. Osie x i y zostaną tak samo oznaczone jak w poprzednim eksperymencie. Można zauważyć znaczną poprawę działania modelu. Jest to spowodowane ustalenie małej wartości współczynnika uczenia, dzięki czemu model może skorzystać z uczenia transferowego i tylko zmodyfikować swoje wagi.



Rysunek 5.9: Dokładność modelu podczas treningu

Rysunek 5.10 przedstawia zmiany funkcji straty podczas treningu modelu. Można zauważać zdecydowaną poprawę w porównaniu do wcześniejszego eksperymentu. Osie x i y zostały tak samo oznaczone jak w poprzednim eksperymencie.



Rysunek 5.10: Strata modelu podczas treningu

Można zauważyc, że wartość współczynnika w tym przypadku została dobrana poprawnie. Wskazuje na to umiarkowana dokładność oraz mała funkcja straty. Średnia dokładność dla tego parametru wyniosła około 0.53, natomiast średnia strata wyniosła około 0.15.

Kolejne eksperymenty zostały przeprowadzone na wartościach współczynnika uczenia, które wynosiły:

- 0.00005,
- 0.0002,
- 0.00015.

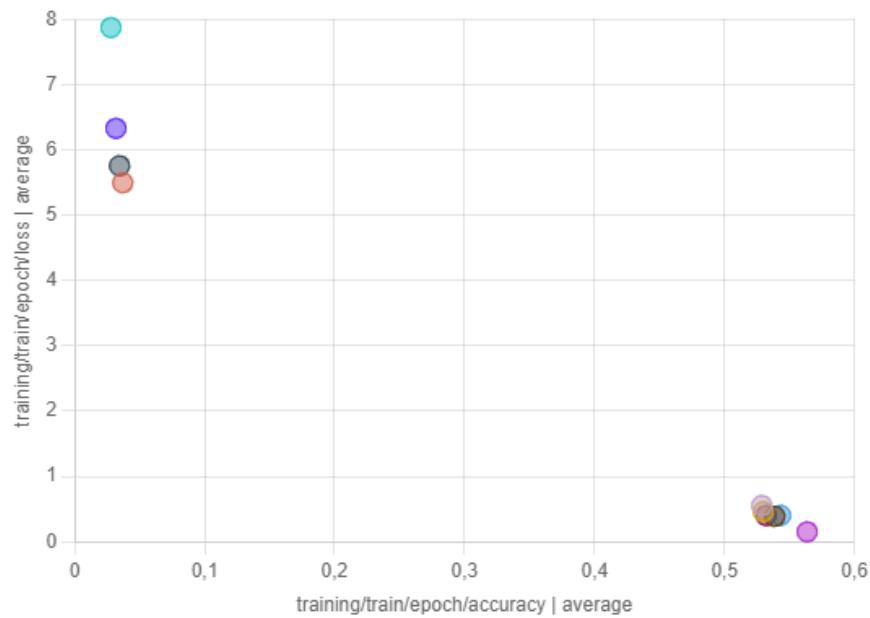
Dla tych wartości nie będą prezentowane dokładne przebiegi dokładności oraz funkcji straty podczas treningu, ze względu na małe zmiany tych metryk. Rysunek 5.11 przedstawia podsumowanie wartości jakie zostały uzyskane dla poszczególnych eksperymentów. Najwyższa dokładność została osiągnięta dla współczynnika uczenia równego 0.00015. Liczba epok w tabeli wskazuje na ustaloną liczbę epok. Platforma Neptune nie przechwytuje rzeczywistej liczby epok. Natomiast maksymalna liczba epok wynosiła około 34. Model po tej liczbie epok nie był w stanie nic więcej się nauczyć. Był to spowodowane małą liczbą przykładów uczących.

# .../learning_rate	Average ...epoch/accuracy	Last training/train/epoch/loss	# ...ms/epochs
0.00005	0.528765	0.159849	100
0.0002	0.529852	0.208474	100
0.00015	0.543207	0.207005	100
0.0001	0.532113	0.158342	100
0.01	0.0371078	4.276695	30

Rysunek 5.11: Podsumowanie eksperymentów

Rysunek 5.12 przedstawia wykres rozrzutu straty w funkcji dokładności. Każdy punkt na wykresie przedstawia pojedynczy trening. Pierwsze cztery punkty w lewym górnym rogu, przedstawiają trening dla wartości współczynnika uczenia równego 0.01.

Tak jak zostało to przedstawione we wcześniejszym eksperymencie były to wyniki nie-optimalne. W prawym dolnym rogu są przedstawione punkty dla innych współczynników uczenia. Widać, że poszczególne treningi różnią się od siebie małymi wartościami. Są to wyniki umiarkowanie dobre, natomiast głównym ograniczeniem w projekcie był bardzo mały zbiór danych.



Rysunek 5.12: Dokładność vs strata

6. Podsumowanie

W pierwszym rozdziale została przedstawiona struktura pracy oraz cel projektu, który udało się spełnić. W rozdziale drugim zostały przedstawiona najważniejsze metody, które zostały wykorzystane w projekcie, np. transfer learning. W kolejnym rozdziale zostały przedstawione i scharakteryzowane technologie, które pomogły w sprawnym stworzeniu projektu. Rozdział czwarty obejmował opis rozwiązania. Został w nim przedstawiony szczegółowy opis implementacji całego projektu od stworzenie środowiska do oszacowania dokładności działania modelu. Ostatni rozdział przedstawał przeprowadzenie eksperymentów. Eksperymenty obejmowały sprawdzenie odpowiednich wartości współczynnika uczenia, a następnie wybranie najlepszego hiperparametru dla modelu. Po przeprowadzeniu eksperymentów można wyciągnąć pewne wnioski na temat dobierania tego hiperparametru. Podczas używania modeli, które były wcześniej trenowane warto wybrać bardzo małą wartość współczynnika uczenia, aby modyfikować wagi, a nie je zmieniać. Jest to bardzo ważne, ponieważ chcąc przekazać wiedzę z jednej dziedziny do drugiej, należy stopniowo modyfikować parametry modelu.

Projekt ma dużą perspektywę rozwoju. Podobnego rozwiązania nie ma na polskim rynku, a warto zwrócić uwagę na to, że język migowy dla każdego języka jest inny. Nie jest to język uniwersalny. Dlatego nie jest prostym zadaniem zmodyfikowanie modelu w innym języku, aby tłumaczył tekst na sekwencje znaków migowych w języku polskim. Motywacją do dalszego rozwoju projektu jest chęć pomocy ludziom głuchym, którzy w pewnych sytuacjach mogą czuć się wykluczeni. Aby zobrazować wykluczenie tych osób, można przytoczyć sytuację z nadawaniem ważnych komunikatów w związku z opóźnieniem pociągów na stacjach kolejowych. Są one zazwyczaj nadawane tylko w formie werbalnej. Osoby, które nie słyszą mogą się czuć wykluczone i niechętnie korzystać z komunikacji publicznej. Po pewnych modyfikacjach projekt ten mógłby pomóc tym ludziom i umożliwić nadawanie komunikatów w języku migowym. Poniżej został przedstawiony dalszy plan rozwoju projektu:

- przeprowadzenie treningu na większym zbiorze danych,
- przeprowadzenie porównania kilku modeli, w celu znalezienia optymalnego rozwiązania,
- dostosowanie większej ilości hiperparametrów z użyciem zaawansowanych tech-

nik doboru parametrów,

- zmiana środowiska na Google Colab, aby zwiększyć dostępność kodu i usunąć ograniczenia wynikające z instalacji środowiska,
- zmiana używanej biblioteki na PyTorch, aby otrzymać dostęp do większej ilości modeli.

Autor za wkład własny uważa:

- przygotowanie środowiska,
- dobranie odpowiedniego modelu,
- wczytanie i przygotowanie danych,
- analiza i trening modelu,
- przeprowadzenie oceny dokładności modelu z wykorzystaniem metryki BLEU,
- przeprowadzenie eksperymentów.

Podsumowując projekt zakończył się powodzeniem. Cel projektu oraz wszystkie założenia stawiane na początku zostały zrealizowane. Wyniki modelu nie są zadowalające, natomiast ograniczenia związane z małą ilością danych nie pozwoliły na osiągnięcie lepszych wyników.

Dodatek A Odtworzenie środowiska uruchomieniowego

Windows

W celu odtworzenie środowiska uruchomieniowego w systemie Windows, należy pobrać Docker Desktop[18]. Warto pamiętać o wcześniejszej instalacji WSL. Jest to funkcja systemu Windows, która umożliwia użycie systemu Linux bez wirtualnego środowiska. Należy to wykonać, ponieważ Docker został stworzony na bazie systemu Linux. Kolejnym krokiem jest pobranie repozytorium z Github. Listing 15 przedstawia kod umożliwiający pobranie repozytorium.

```
1 git clone https://github.com/KacperUrban/Text_to_sign.git
```

Listing 15: Pobieranie projektu z repozytorium

Ostatnim krokiem jest otwarcie projektu w zintegrowanym środowisku programistycznym np. w PyCharm oraz zbudowanie obrazu. Warto pamiętać o tym, że jeśli nie mamy dostępu do wersji profesjonalnej, nie będzie możliwe bezpośrednie korzystanie z PyCharm. Wtedy należy włączyć projekt w przeglądarce, po przez naciśnięcie linku, który się wygeneruje po zbudowaniu obrazu oraz kontenera. Listing 16 przedstawia kod, który buduje obraz (jeśli nie był wcześniej zbudowany oraz uruchamia kontener).

```
1 docker-compose up
```

Listing 16: Zbudowanie obrazu oraz uruchomienie kontenera

Linux

Procedura odtwarzanie środowiska uruchomieniowego dla systemu Linux różni się od tej dla systemu Windows. Główną różnicą jest to, że nie trzeba instalować funkcji WSL. Przed przystąpieniem do pobierania należy uaktualnić wszystkie biblioteki. Następnym krokiem jest pobranie Dockera. Listing 17 przedstawia aktualizację bibliotek oraz pobieranie Dockera. Aktualnie dla systemu Linux jest dostępna wersja desktopowa, która sama w sobie zawiera Docker-compose, dlatego nie ma potrzeby instalacji tego narzędzia.

```
1 sudo apt-get update
2 sudo apt-get install ./docker-desktop-<version>-<arch>.deb
```

Listing 17: Aktualizacja bibliotek oraz instalacji Dockera

Pozostałe kroki, są takie same jak dla systemu Windows. Dlatego można skorzystać z instrukcji dla tego systemu. Warto pamiętać o tym, że niektóre środowiska programistyczne mogą wymagać dodatkowej konfiguracji w ustawieniach środowiska. Problem może być rozwiązyany, po przez ustawienie odpowiedniego linku do strony na której wyświetla się jupyter notebook. Również czasami środowisko może wymagać przekazania informacji jaki plik jest Dockerfile lub docker-compose.yaml.

Literatura

- [1] <https://machinelearningmastery.com/transfer-learning-for-deep-learning/>. Dostęp 15.11.2023.
- [2] <https://aws.amazon.com/what-is/machine-translation/>. Dostęp 15.11.2023.
- [3] <https://www.memoq.com/tools/what-is-machine-translation>. Dostęp 17.11.2023.
- [4] https://pl.wikipedia.org/wiki/Tempo_wypowiedzi. Dostęp: 29.11.2023.
- [5] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A., Kaiser Ł., Polosukhin I.: Attention Is All You Need. 2017.
- [6] <https://www.turing.com/kb/guide-on-word-embeddings-in-nlp>. Dostęp: 15.12.2023.
- [7] <https://medium.com/@prudhviraju.srivatsavaya/what-is-positional-encoding-32bcd5b569b4#:~:text=The%20positional%20encoding%20function%20typically,patterns%20and%20so%20on> 15.12.2023.
- [8] <https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34>. Dostęp: 15.12.2023.
- [9] <https://developer.apple.com/documentation/accelerate/bnnsactivationfunction/2915301-softmax>. Dostęp: 15.12.2023.
- [10] <https://machinetranslate.org/metrics>. Dostęp: 7.12.2023.
- [11] Papineni K., Roukos S., Ward T., Zhu W. BLEU: a Method for Automatic Evaluation of Machine Translation. 2002.
- [12] <https://pl.theastrologypage.com/tensorflow>. Dostęp 17.11.2023.
- [13] <https://boringowl.io/tag/tensorflow>. Dostęp 17.11.2023.
- [14] <https://towardsdatascience.com/whats-hugging-face-122f4e7eb11a>. Dostęp 17.11.2023.
- [15] <https://huggingface.co/docs/transformers/index>. Dostęp 17.11.2023.
- [16] <https://docs.docker.com/get-started/overview/>. Dostęp 17.11.2023.

[17] <https://docs.neptune.ai/>. Dostęp 17.11.2023.

[18] <https://docs.docker.com/desktop/install/windows-install/>. Dostęp: 15.12.2023.