Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki
Instytut Informatyki Stosowanej

PRACA DYPLOMOWA MAGISTERSKA

Automatyczna generacja kodu źródłowego w języku C przy użyciu sieci rekurencyjnych

Automatic C language source code generation using recurrent neural networks

Autor Filip Czerwiński
Numer albumu: 239515

Promotor / Promotor pomocniczy pracy:
prof. dr hab. inż. Anna Fabijańska

Łódź, luty, 2023

Streszczenie

Praca przedstawia model dwukierunkowej sieci rekurencyjnej (RNN) złożonej z bramek pamięci długotrwałej (LSTM), której celem jest generacja kodu źródłowego na podstawie tekstowych poleceń. Początkową część pracy stanowi opis wymagań projektowych oraz obowiązujące limity sieci. Dalej wskazano stosowane technologie wymagane do stworzenia modelu oraz przetwarzania danych. W części praktycznej opisany został dobór warstw modelu i jego trening. W ostatniej części pojawia się dyskusja dotycząca osiągniętych wyników.

Słowa kluczowe:

Generacja kodu źródłowego, C, Dwukierunkowe, RNN, LSTM

Abstract

Thesis presents the implementation of a bidirectional Recurrent Neural Network (RNN) network composed of Long Short-Term Memory (LSTM) gates, the purpose of which is to generate the source code on the basis of text commands. The initial part of the paper describes the requirements and limits of the network. The next one contains the technologies used to create the model and data processing. The practical part describes the selection of model layers and its training. In the last part there is a discussion of the achieved results.

Keywords:

Source code generation, C, Bidirectional, RNN, LSTM

Spis Treści

1 Wstęp

1.1 Motywacja

Obecnie temat generacji kodu bardzo szybko zyskuje na popularności. Ma to miejsce za sprawą popularnych rozwiązań sztucznej inteligencji wbudowanych w zintegrowane środowiska programistyczne, które daje użytkownikom wiele różnych narzędzi do generowania, optymalizacji oraz poprawiania fragmentów kodu. Narzędzia te są przydatne programistom w różnych sytuacjach. Ponieważ generowany kod jest zwykle optymalny oraz zostały w nim zastosowane dobre praktyki dla danego języka, jest to dobry przykład dla osób dopiero uczących się programować. W przypadku pracy w zespołach, kiedy wielu programistów pracuje nad jednym projektem, styl ich kodu może się znacząco różnić. Jeśli jednak spójność kodu jest kluczowa, użycie wygenerowanego kodu może znacznie ułatwić to zadanie, jak również pozwala zaoszczędzić czas na refaktoryzację. Generacja kodu zabiera zdecydowanie mniej czasu niż własnoręczne napisanie kodu, jest pozbawiona błędów, co znacząco ułatwia pracę każdego programisty.

1.2 Sieci neuronowe

Sztuczne sieci neuronowe wykonują czynności podobne do procesów zachodzących w mózgu człowieka. Ludzki mózg i procesy w nim zachodzące stanowią inspirację do konstrukcji modeli sztucznych sieci neuronowych. Przez wejście dane są przekazywane do kolejnych warstw złożonych z wielu neuronów, a po przetworzeniu są transmitowane jako dane wyjściowe. Tego rodzaju sieci pomagają realizować zadania, takie jak klasyfikacja danych i rozpoznawanie wzorca.

Skuteczność takich sieci w rozwiązywaniu problemów, które nie zostały jeszcze sformalizowane, jak również ich niezwykła zdolność generalizowania zdobytych informacji sprawiły, że w ostatnim czasie są one przedmiotem szczególnego zainteresowania [1]. Kolejnym czynnikiem, który znacząco wpłynął na popularność sieci neuronowych, są rosnące ilości danych, z których należy wydobyć tylko istotne dla rozstrzygnięcia określonego problemu informacje. Jednak mimo wielu zalet charakteryzujących tego rodzaju sieci mają ograniczenia, które pojawiają się w złożonych problemach, takich jak klasyfikacja obrazu czy też przetwarzanie sekwencji danych. Tego rodzaju problemy daje się rozwiązać, tworząc kolejne

odmiany sieci, w rodzaju splotowych bądź rekurencyjnych sieci neuronowych. Sieci rekurencyjne są pomocne w antycypacji kolejnych fragmentów tekstu. Zapamiętane wcześniej słowa są podstawą do podjęcia decyzji nie tylko w oparciu o konkretne słowo, ale również w kontekście pozostałych słów.

1.3 Cel

Celem tej pracy jest stworzenie modelu sieci rekurencyjnej, a następnie w oparciu o przykłady z bazy danych CodeNet zawierającej opisy problemów programistycznych oraz odpowiadający im kod źródłowy w języku C, wykonanie przetwarzania wstępnego danych oraz wytrenowanie sieci. Następnie na nauczonym modelu sprawdzić, czy wygenerowane przez sieć kody źródłowe dla problemów z bazy danych się kompilują, oraz w jakim stopniu przypominają kod referencyjny dla danego problemu.

1.4 Zakres pracy

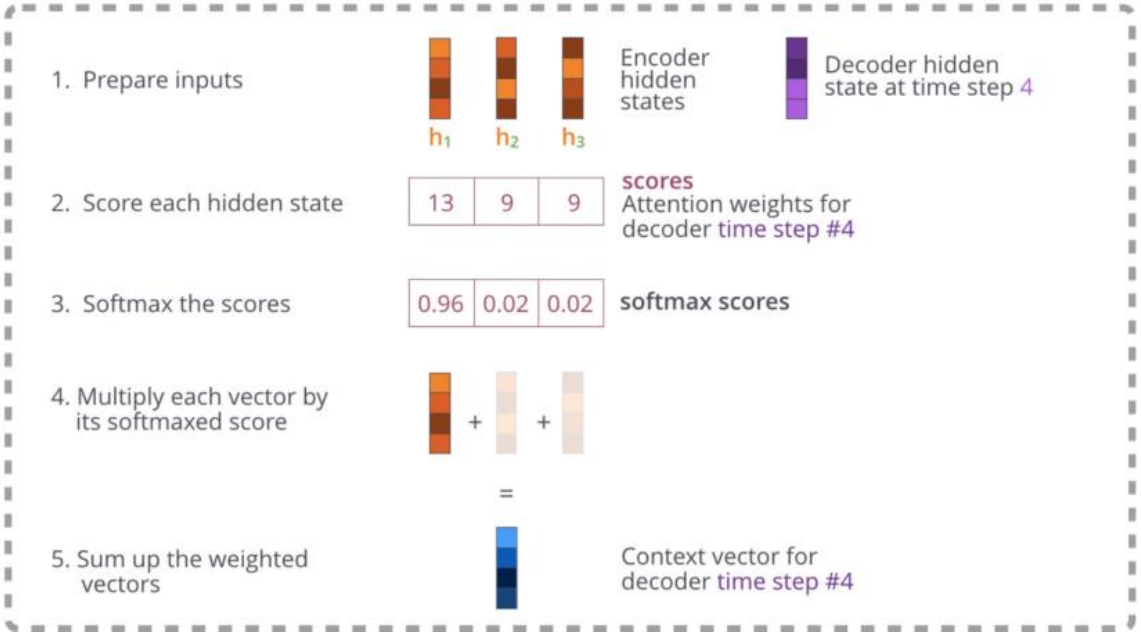
W zakresie niniejszej pracy jest napisanie sieci do generowania fragmentów kodu w języku C. Kod powinien zawierać elementy składni standardowej dla danego języka oraz przypominać kod referencyjny. Jednocześnie sieć powinna nabyć umiejętność generalizacji w celu dostosowania generowanego kodu w przypadku zmiany opisu problemu. Kryterium oceny jest możliwość kompilacji wygenerowanego kodu oraz poprawność otrzymanego wyniku.

2 Aktualny stan wiedzy

W tym rozdziale zostały zaprezentowane istniejące rozwiązania w zakresie generacji kodu źródłowego na podstawie poleceń użytkownika. Ponieważ uczenie sieci neuronowych, mających na celu przetwarzanie języka naturalnego, jest z reguły procesem o ogromnej złożoności obliczeniowej, stworzono modele posiadające od kilku milionów do kilkudziesięciu bilionów parametrów, które następnie zostały trenowane na dużych korpusach tekstu z różnych serwisów internetowych, takich jak reddit lub wikipedia. Modele są w stanie posłużyć do rozwiązania podobnych problemów, do których zostały stworzone. Przykładami omawianych modeli są wielozadaniowe dwukierunkowe reprezentacje enkodera z transformatorów (BERT) i generatywnie wstępnie wytrenowanego transformatora (GPT) [38].

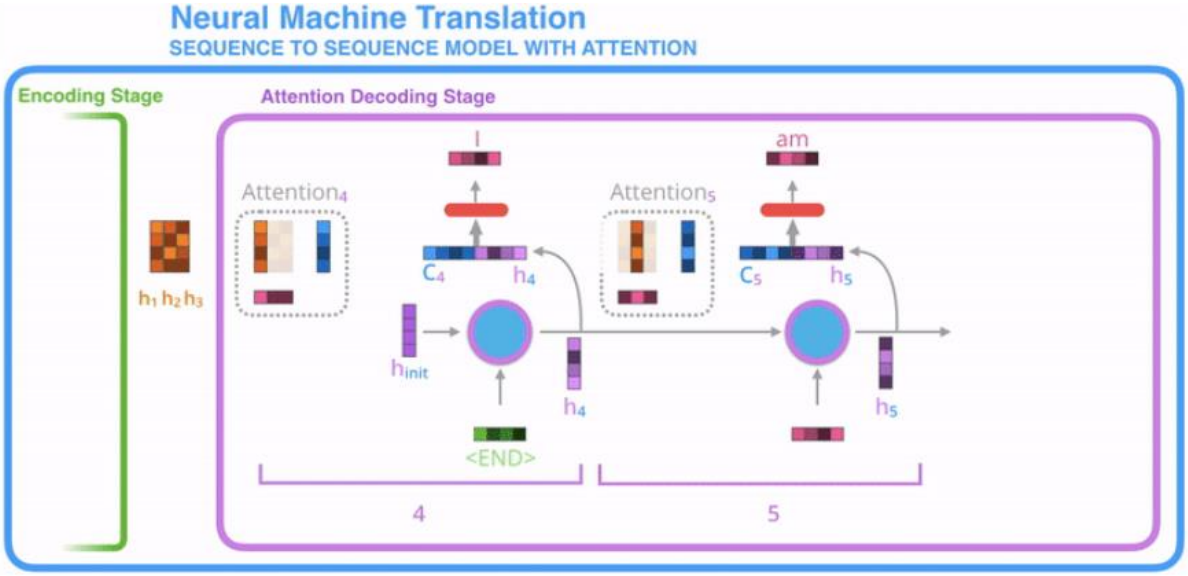
2.1 Architektura transformer

Pierwowzorem architektury transformera były sieci rekurencyjne, których komórki przetwarzają kolejną informację w sekwencji, jak również aktualny stan. Z uwagi na pojawiający się problem zanikających gradientów, który zaobserwowano przy treningu sieci rekurencyjnych na długich sekwencjach, jak również trudność w śledzeniu dalekich zależności danych oraz długi czas obliczeń, większość aktualnych rozwiązań opartych jest o architekturę zaproponowaną w artykule “Attention Is All You Need”, gdzie Transformer pojawił się po raz pierwszy [4]. Mechanizm uwagi, który został w nim opisany, stał się kluczowy dla współczesnych rozwiązań. Dzięki niemu sieci potrafią dokonywać selekcji stanów ukrytych, które są istotne do predykcji danych wyjściowych. Sposób ten jest tożsamy z tym, w jaki ludzie śledzą informację w sekwencji, czyli analiza na podstawie wyłącznie najbardziej istotnych dla nas informacji. Enkoder w transformerze w przeciwieństwie do sieci rekurencyjnych przekazuje wszystkie stany ukryte sieci zamiast wyłącznie wyjściowych. Dekoder sprawdza, czy stany ukryte przekazane z enkodera są powiązane z określonym słowem, a następnie wystawia ocenę mnożoną przez funkcję softmax, co przedstawione zostało na rysunku 1 [2].



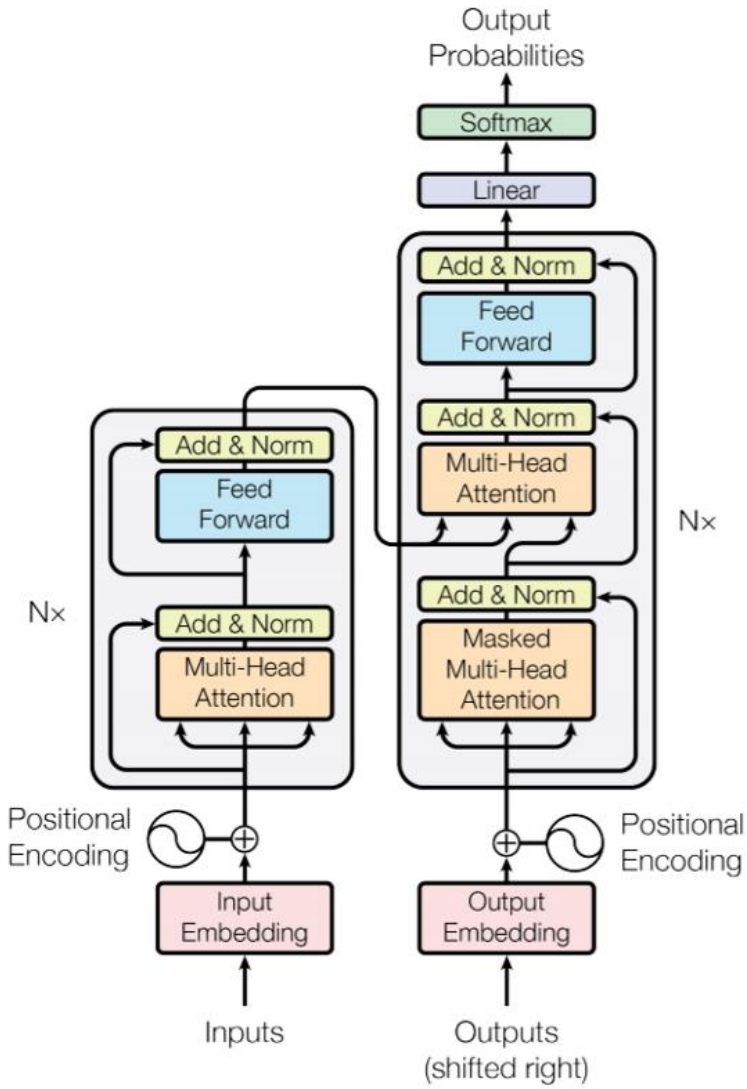
Rysunek 1 Wizualizacja utworzenia wektora kontekstu ze stanów ukrytych [2]

Tego rodzaju proces wykonywany jest dla każdego kroku czasowego. Warstwa dekodera przyjmuje poprzedni token oraz stan ukryty, a następnie tworzy nowy stan ukryty. Kolejne ukryte stany enkodera, jak również następny ukryty stan dekodera użyte są, aby utworzyć wektor kontekstu dla danego kroku czasowego. Wektor kontekstu połączony z nowym ukrytym stanem dekodera trafiają do jednokierunkowych sieci feed-forward, które wskazuje słowo wyjściowe dla danego kroku czasowego. Sygnały przechodzą z warstwy wejściowej do warstwy ukrytej, o ile takie występują, by dotrzeć jednokierunkowo do warstwy wyjściowej. Rolą neuronów wejściowych jest wstępna obróbka sygnału, polegająca m.in. na normalizacji czy też kodowaniu. Przetwarzanie decyzyjne jest realizowane w neuronach ukrytych i w warstwie wyjściowej. Taka operacja powtarzana jest, dla wszystkich kroków czasowych dekodera co obrazuje rysunek 2 [2]. Transformer jest siecią typu auto enkoder, która w zależności od rodzaju może składać się wyłącznie z warstwy enkodera bądź dekodera.



Rysunek 2 Wizualizacja dekodowania w modelu seq2seq z attention [2]

Warstwa embedding zwykle składa się z modeli typu pretrained word embeddings, które trenowane są na dużych zbiorach danych, gdzie uczą się znajdowania zależności między danymi. Najbardziej znane to Word2Vec [3] oraz GloVe [3] które uczone są w sposób nienadzorowany. Oba używają architektury Continuous Bag of Words [3] lub Skip-gramu [3]. Różnica między nimi pojawia się przy treningu, gdzie współwystępowanie w przypadku Word2Vec odnosi się do kontekstu lokalnego (sąsiadujące słowa), w przypadku GloVe do kontekstu globalnego. Kolejnym modelem podobnym do przywoływanych jest FastText [3], który rozwiązał problem enkodowania słów pojawiających się w treningu poprzez ich podział na podsłowa. Ostatnim dość znanym jest ELMo, który w odróżnieniu od wcześniej wskazywanych, wykonuje osadzenie na całych zdaniach. Za sprawą przetwarzania pojedynczych znaków, zamiast wyrazów może przetwarzać nieznane słowa [3]. Pełną architekturę transformera przedstawia rysunek 3.



Rysunek 3 Architektura modelu Transformera [4]

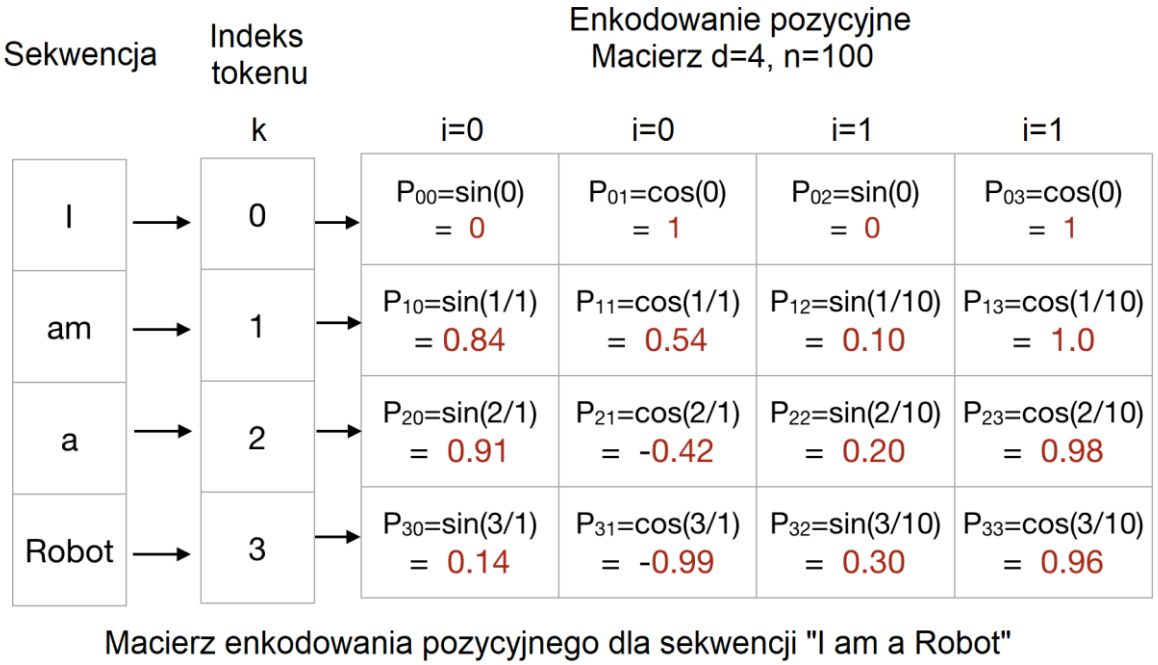
Ponieważ transformery nie posiadają żadnych wbudowanych mechanizmów, takich jak rekurencja czy konwolucja, które umożliwiłyby im śledzenie kolejności danych w sekwencji, muszą traktować dane niezależnie od siebie. To wymusza konieczność zamiany słów na liczby za sprawą tzw. osadzenia (ang. embeddings), polegającego na umieszczeniu słów w przestrzeni wielowymiarowej, zamianie słowa na wektor. Dlatego, aby zachować informację o kolejności słów, zaczęto stosować warstwę enkoderów pozycjonujących (ang. positional embedding), czyli wektorów dających kontekst w odniesieniu do pozycji słowa w zdaniu czy też wybranej sekwencji słów. Powszechnie stosowanym rozwiązaniem jest użycie funkcji sinus i cosinus o różnej częstotliwości. Funkcje tę opisują wzory 1 oraz 2, gdzie “poz” to pozycja, a “i” to

wymiar. Oznacza to, że każdy wymiar kodowania pozycyjnego odpowiada sinusoidzie. Długości fal tworzą postęp geometryczny od 2π do 10000 · 2π [4].

$$Pe_{(pos,2i)} = sin(pos/1000^{2i/d_{model}}) \quad (1)$$

$$Pe_{(pos,2i+1)} = cos(pos/1000^{2i/d_{model}}) \quad (2)$$

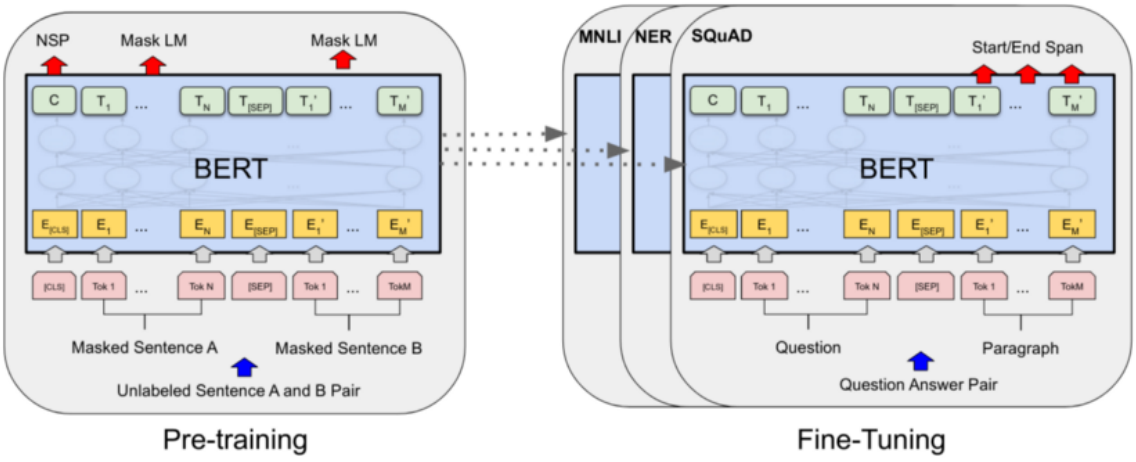
Funkcje te mają na celu doprowadzić do przesunięcia słowa w przestrzeni wielowymiarowej, tak by słowa o tym samym indeksie w sekwencji były bardziej zbliżone. Główną zaletą stosowania funkcji sinusa i cosinusa dla realizacji tego celu jest to, że ich zakres wartości jest zawarty w przedziale (1; -1), dzięki czemu wartości nawet w bardzo długiej sekwencji nie będą wychodziły poza wspomniany zakres wartości. Przykład enkodowania pozycyjnego przedstawiony został na rysunku 4 [5].



Rysunek 4 Przykład enkodowania pozycyjnego [5]

2.2 Wstępnie wyszkolone transformatory

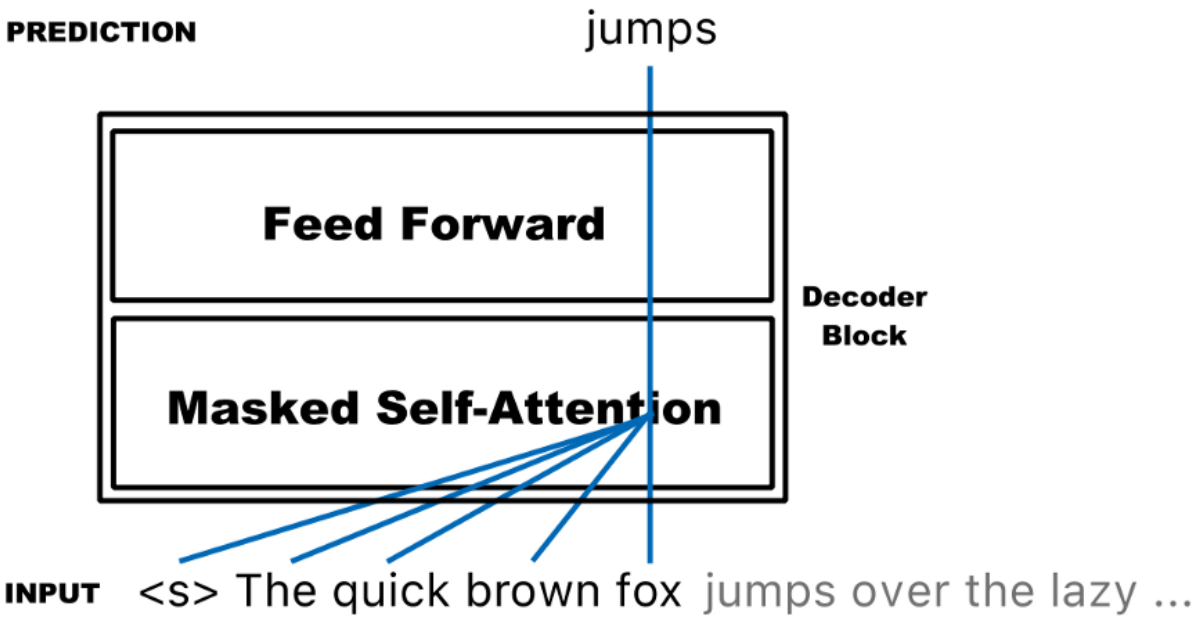
Trend w stosowaniu wstępnie wyszkolonych transformatorów pojawił się ze względu na ich elastyczność i niezależność od wykonywanych zadań. Wcześniej sieci musiały posiadać architekturę stworzoną specjalnie dla powierzonego im zadania. Obecnie pre-trained transformers są trenowane w sposób nienadzorowany na dużych, zróżnicowanych zestawach danych, a następnie wykonuje się tak zwany fine-tuning, który wyeliminował potrzebę dostosowywania architektury do zadania, co doprowadziło do znacznego postępu w różnych zadaniach związanych z przetwarzaniem języka naturalnego, takich jak czytanie ze zrozumieniem, odpowiedzi na pytania i wiele innych. Fine-tuning polega na dostrojeniu sieci w sposób nadzorowany na mniejszym zestawie danych specyficznym dla danego zadania. Ograniczeniami tego rodzaju modeli jest fakt, że ostatecznie muszą zostać douczone w sposób nadzorowany, a co za tym idzie, nadal pojawia się konieczność przygotowywania danych, jak również ich umiejętność generalizowania jest ograniczona do dziedziny, do której zostały dostrojone [6]. Za przykład można podać transformer Bidirectional Encoder Representations from Transformers (BERT), którego procedura dostrajania przedstawiona została na rysunku 5.



Rysunek 5 BERT trening wstępny i zadania dostrajające [7]

2.3 GPT

Większość współczesnych zadań analizy, podsumowania, uzupełniania oraz generacji tekstu, w tym również generacji kodu, opiera się o generatywne modele pre-trained transformer stworzone przez OpenAI, które aktualnie występują w trzech wersjach. Określenie generatywne jest używane dlatego, iż takie modele potrafią tworzyć długie sekwencje tekstu [8]. Architektura tych modeli oparta jest na bloku dekodera transformera. Część wzajemnej uwagi nie występuje w tych modelach, ze względu na brak bloku enkodera. W modelach zastosowane zostało swobodne modelowanie języka, który polega na wprowadzeniu wszystkich tokenów wejściowych do modelu i przewidywaniu kolejnych tokenów w każdym kroku czasowym. Następnie proces jest powtarzany wraz nowo wygenerowanymi tokenami, które zostają podane do sieci, aby wykonać kolejną predykcję dla następnego kroku czasowego. Część zamaskowanej własnej uwagi będzie zapobiegać analizy przyszłych tokenów przez ich zamaskowanie [9]. Przykład maskowania bloku dekodera obrazuje rysunek 6.



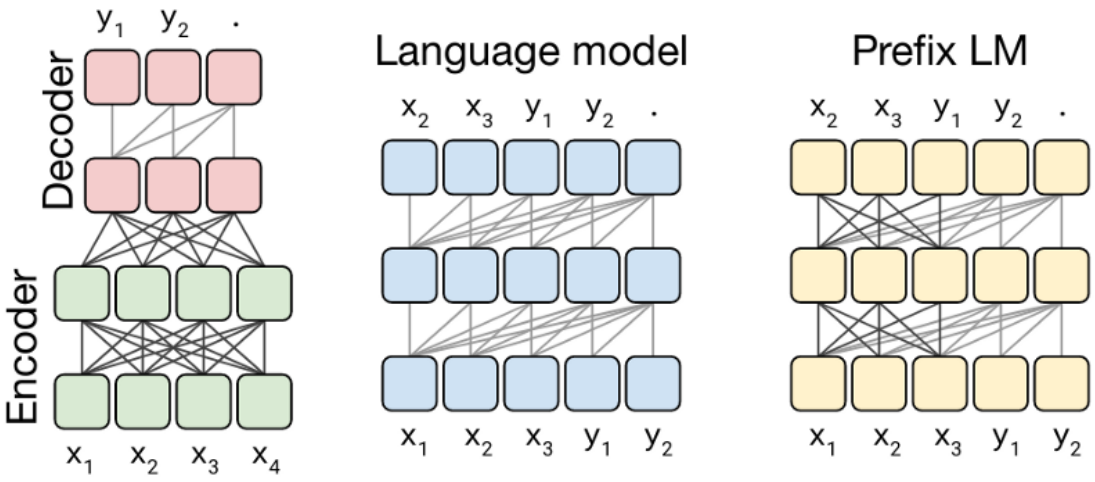
Rysunek 6 Schemat bloku dekodera z zamaskowanymi połączeniami własnej uwagi w danym kroku czasowym [9]

W GPT zastosowana została nauka jednorazowego uczenia się (ang. few-shots-learning), która zastąpiła dostrajanie. Jest to wydajny sposób ze względu na możliwość

ograniczenia wielkości zbioru, na którym należy dostroić sieć. Dokonuje się tego na podstawie podobieństwa danych ze zbiorem pomocniczym podczas predykcji. Zwykle dostrojenie sieci do nowego zestawu danych tą metodą wymaga zaledwie kilka przykładów [8]. Modele generatywne są w stanie wykonywać wiele różnych zadań. Do niektórych danych dodawane są specjalne tokeny reprezentujące zadanie zrozumiałe dla sieci, a następnie predykcje zostają dostosowane względem danego zadania.

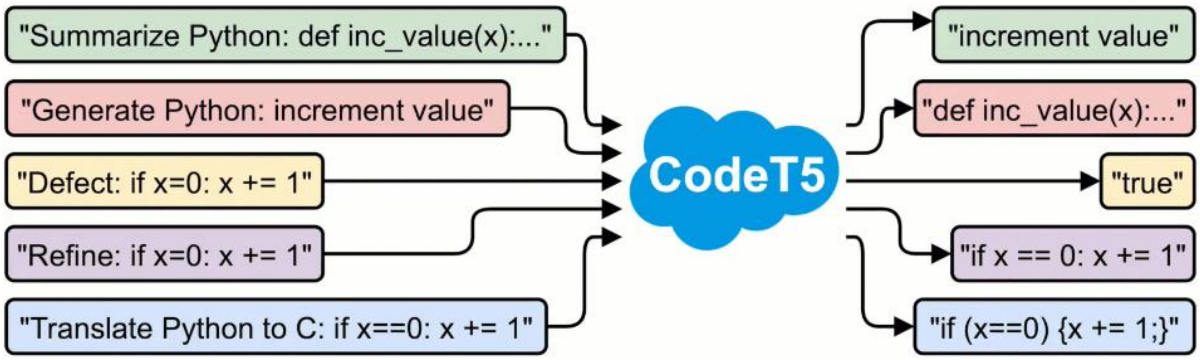
2.4 Modele do generacji kodu

Najbardziej znanym na tę chwilę modelem generowania kodu jest OpenAI Codex, który został stworzony przez OpenAI i zasila Github copilota. Oparty jest na modelu GPT-3 i jest w stanie pisać kod w kilkunastu językach programowania. Do treningu wykorzystane zostały miliardy wierszy kodu z domen publicznych, jak również repozytoriów GitHuba [10]. Sam model jest w stanie tworzyć kolejne linie kodu na podstawie zarówno polecenia, jak i kodu wcześniej napisanego. Narzędzie to jest również w stanie tworzyć podsumowanie kodu, przeprowadzić konwersję z jednego języka na inny, jak również refaktoryzować kod. Kolejnym modelem do generacji kodu jest stworzony przez SalesForce CodeT5. Model posiada architekturę podobną do znanego modelu BERT, który polega na maskowaniu kolejnych słów w sekwencji, a następnie próbie ich przewidzenia. CodeT5 posiada jednak zarówno warstwę enkodera oraz dekodera co sprawia, że model ten lepiej radzi sobie lepiej w przypadku zadań takich jak podsumowywanie kodu [11]. Głównym czynnikiem wyróżniającym różne architektury jest „maska” używana przez różne mechanizmy uwagi w modelu. Na poziomie architektonicznym istnieje kilka opcji przedstawionych na rysunku 7. Oprócz standardowej architektury typu enkoder-dekoder można również użyć modelu językowego, który jest metodą modelowania autoregresyjnego z mechanizmem przyczynowej uwagi oraz Prefix LM, który łączy podejścia w stylu BERT i modelu językowego [12].



Rysunek 7 Przykłady architektury transformera przetwarzającego tekst na tekst [11]

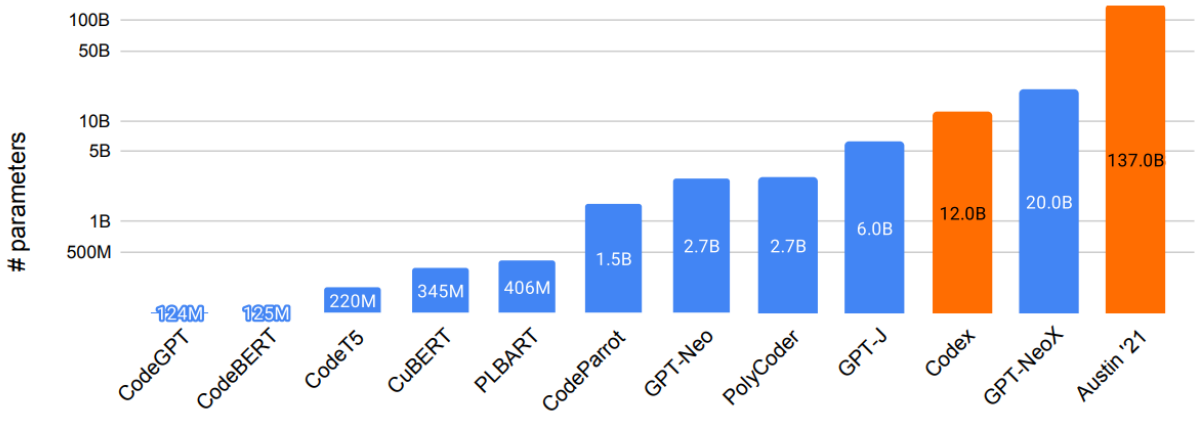
Model CodeT5, podobnie jak Github copilot, trenowany był na kodzie z publicznych repozytoriów GitHuba, jak również na zbiorze CodeSearchNet. Jest on również w stanie wykonać podobne zadania związane z przetwarzaniem kodu co Github copilot. Przykładowe zadania, które mogą zostać zrealizowane przy pomocy tego modelu, przedstawione zostały na rysunku 8.



Rysunek 8 Możliwe zadania związane z przetwarzaniem kodu przez CodeT5 [12]

Ostatnim interesującym modelem do generacji kodu źródłowego jest Polycoder, który został opracowany na podstawie modelu GPT-2 stworzonego przez OpenAI. Jego główną zaletą jest fakt, że pomimo posiadania znacząco mniejszej ilości parametrów niż OpenAI Codex autorzy Polycodera zapewniają, że potrafi pisać kod w języku C z większą dokładnością niż jakikolwiek inny model. Model ten jest jednym z pierwszych

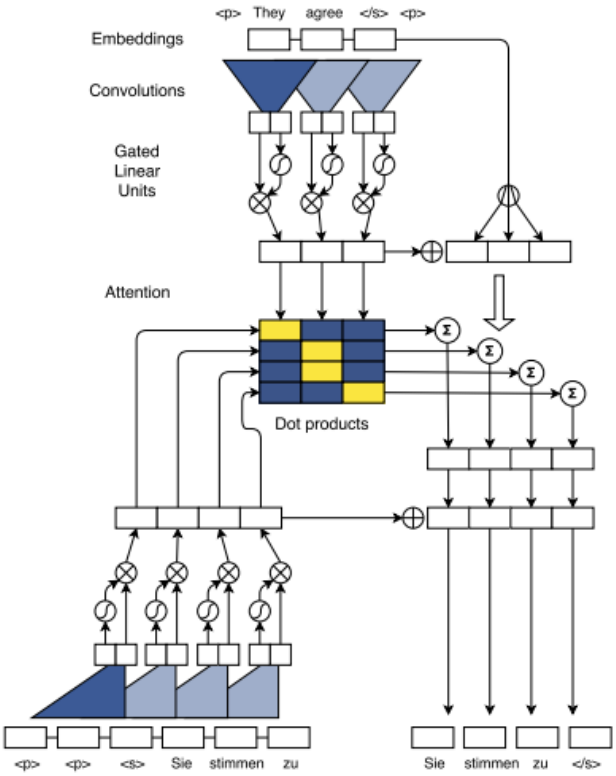
modeli generowania kodu open source, który według naukowców przewyższa podobne pod względem rozmiaru modele open source w językach takich jak JavaScript, Rust, Scala i TypeScript [13]. Istniejące modele językowe do generacji kodu względem ich rozmiaru przedstawione zostały na rysunku 9 [15]. Modele powszechnie dostępne (ang. open-source) oznaczone zostały na niebiesko, w przeciwnym wypadku na pomarańczowo.



Rysunek 9 Istniejące modele do generacji kodu względem ich rozmiaru [15]

2.5 Fairseq

Ciekawą alternatywą dla poprzednio omówionych rozwiązań jest architektura Fairseq prezentowana na rysunku 10, która została zaproponowana przez pracowników grupy Facebook AI Research służąca do modelowania sekwencji przy użyciu sieci splotowych. Wyróżnia ją znacząco wyższa prędkość przetwarzania tekstu niż w sieciach rekurencyjnych przy równoczesnym zachowaniu wydajności [16]. Na obecną chwilę model ten stosowany jest głównie do zadań związanych z tłumaczeniem tekstu, jednakże może stanowić ciekawą alternatywę w innych obszarach, między innymi generacji kodu.

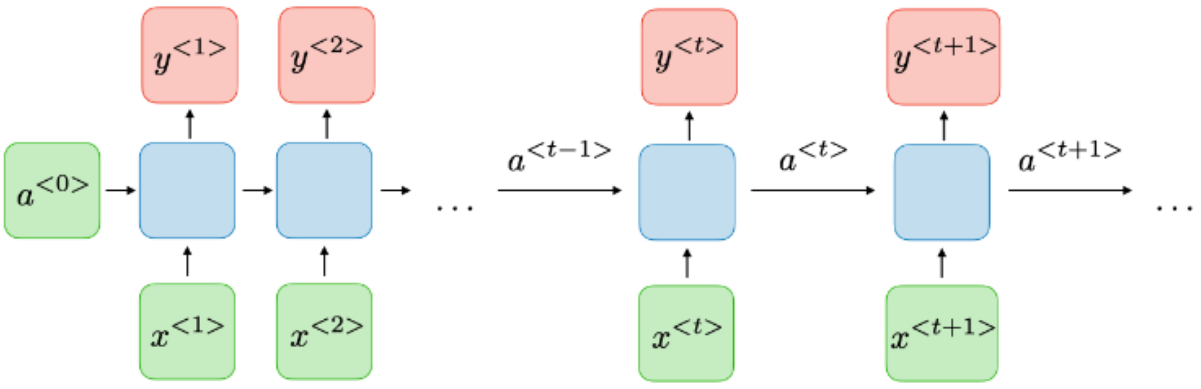


Rysunek 10 Architektura Fairseq [16]

3 Rekurencyjne sieci neuronowe

3.1 Działanie

Rekurencyjna sieć neuronowa jest rodzajem sztucznej sieci neuronowej, która wykorzystuje dane sekwencyjne lub szeregi czasowe. Te algorytmy głębokiego uczenia są powszechnie stosowane w przypadku problemów porządkowych lub czasowych, takich jak tłumaczenie języka, przetwarzanie języka naturalnego, rozpoznawanie mowy i podpisy obrazów. Podobnie jak sieci jednokierunkowe i konwolucyjne sieci neuronowe, rekurencyjne sieci neuronowe wykorzystują dane szkoleniowe do uczenia się. Wyróżniają się swoją pamięcią, ponieważ pobierają informacje z wcześniejszych danych wejściowych, aby wpływać na bieżące wejście i wyjście. Podczas gdy tradycyjne głębokie sieci neuronowe zakładają, że dane wejściowe i wyjściowe są od siebie niezależne, dane wyjściowe powtarzających się sieci neuronowych zależą od wcześniejszych elementów w sekwencji. Podczas gdy przyszłe zdarzenia byłyby również pomocne w określaniu wyniku danej sekwencji, jednokierunkowe sieci neuronowe nie mogą uwzględniać tych zdarzeń w swoich przewidywaniach. Sieci RNN również uczą się uwzględniać pozycję każdego słowa w zdaniu i wykorzystują tą informacje do przewidywania następnego słowa w sekwencji. Inną charakterystyczną cechą sieci rekurencyjnych jest to, że współdzielą one parametry w każdej warstwie sieci. Podczas gdy sieci jednokierunkowe mają różne wagi w każdym węźle, sieci rekurencyjne mają ten sam parametr wagi w każdej warstwie sieci. Wagi te są nadal dostosowywane w procesach propagacji wstecznej, aby ułatwić uczenie się przez wzmacnianie [17]. Budowę przykładowej sieci RNN przedstawia rysunek 11.



Rysunek 11 Przykładowa sieć RNN [17]

Budowa tego rodzaju sieci różni się w zależności, do czego jest stosowana. Możemy tu wyróżnić kilka jej rodzajów:

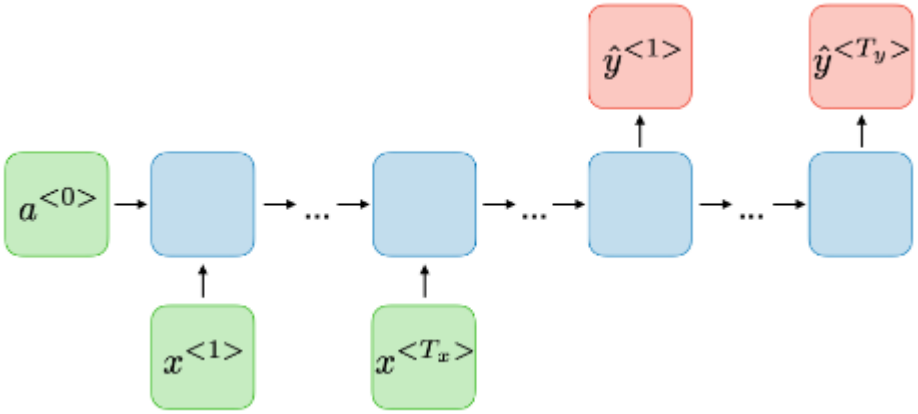
- jeden do wielu,

- wiele do jednego,

- wiele do wielu ze stałą długością wejść i wyjść,

- wiele do wielu ze zmienną długością wejść i wyjść.

Sieci wiele do wiele można podzielić ze względu na stałą lub zmienną wielkość wejść oraz wyjść sieci. Natomiast w przypadku gdy z małej ilości informacji na wejściu należy uzyskać dużą ilość na wyjściu, można zastosować sieć typu jeden do wielu, a w odwrotnej sytuacji sieć typu wiele do jednego. Do generacji kodu wymagana jest budowa sieci o zmiennej długości, której przykład znajduje się na rysunku 12 [17].



Rysunek 12 Sieć rekurencyjna typu wiele do wiele o zmiennej długości wejść i wyjść [17]

3.2 Funkcja straty

W sieciach rekurencyjnych stosuje się tę samą funkcję straty co w zwykłych sieciach. Jednak wartość funkcji straty zależy od gradientu w aktualnym kroku, jak również gradientu we wszystkich poprzednich, ponieważ wagi neuronów są współdzielone pomiędzy wszystkimi krokami w czasie. Ostatecznie wartość wszystkich gradientów zostaje zsumowana [17]. Omówioną funkcję straty przedstawia wzór 3 gdzie L to wartość funkcji straty, ŷ to wartość predykcji, y to wartość prawdziwa, t jest aktualnym krokiem czasowym, T jest ilością kroków czasowych w sekwencji.

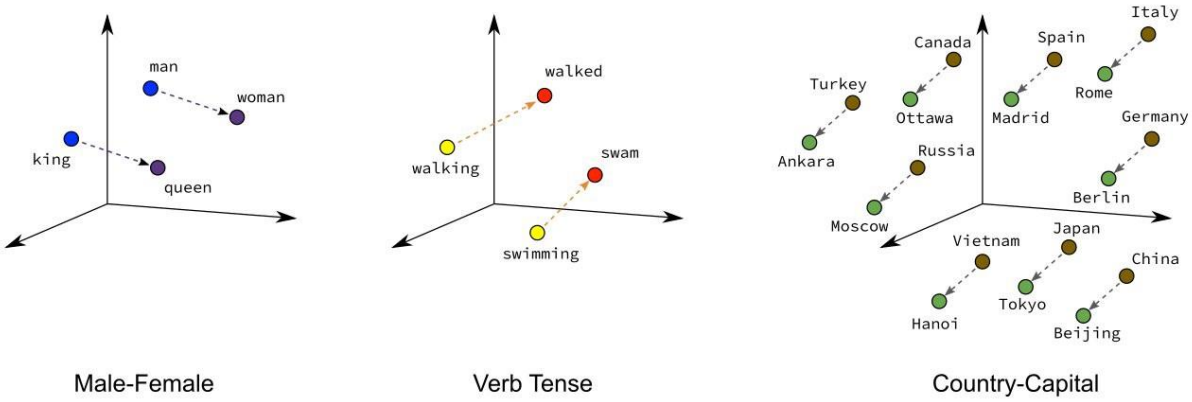
$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L(\hat{y}^{<t>}, y^{<t>}) \quad (3)$$

Obliczanie wartości propagacji wstecznej w czasie opisane jest we wzorze 4 gdzie L to wartość funkcji straty, W to macierz wag, t jest aktualnym krokiem czasowym, T jest ilością kroków czasowych w sekwencji.

$$\frac{\partial L^{(T)}}{\partial W} = \sum_{t=1}^{T} \frac{\partial L^{(T)}}{\partial W}\Big|_{(T)} \quad (4)$$

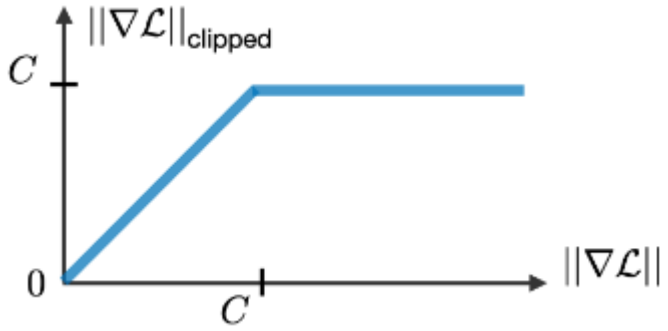
3.3 Osadzenie

Podobnie jak w przypadku innych rodzajów, sieć RNN ma również swoje ograniczenia. Pierwsze z nich związane jest z przetwarzaniem tekstu. Tekst musi zostać zamieniony na reprezentację liczbową, aby sieć była go w stanie przetworzyć. Można to zrobić poprzez przypisanie każdemu słowu albo literze unikalnej liczby, która będzie je reprezentować. Jednakże liczba ta nie daje żadnej istotnej informacji o tym, jak bardzo dane słowa różnią się od pozostałych. Pomocną w tej sytuacji jest warstwa osadzenia, mająca możliwość poszukiwania zależności między kolejnymi słowami, a następnie grupowanie ich w przestrzeni wielowymiarowej przez co do sieci dostarczane jest więcej informacji. Przykład wspomnianego grupowania odzwierciedla rysunek 13.



Rysunek 13 Przykład działania osadzenia [18]

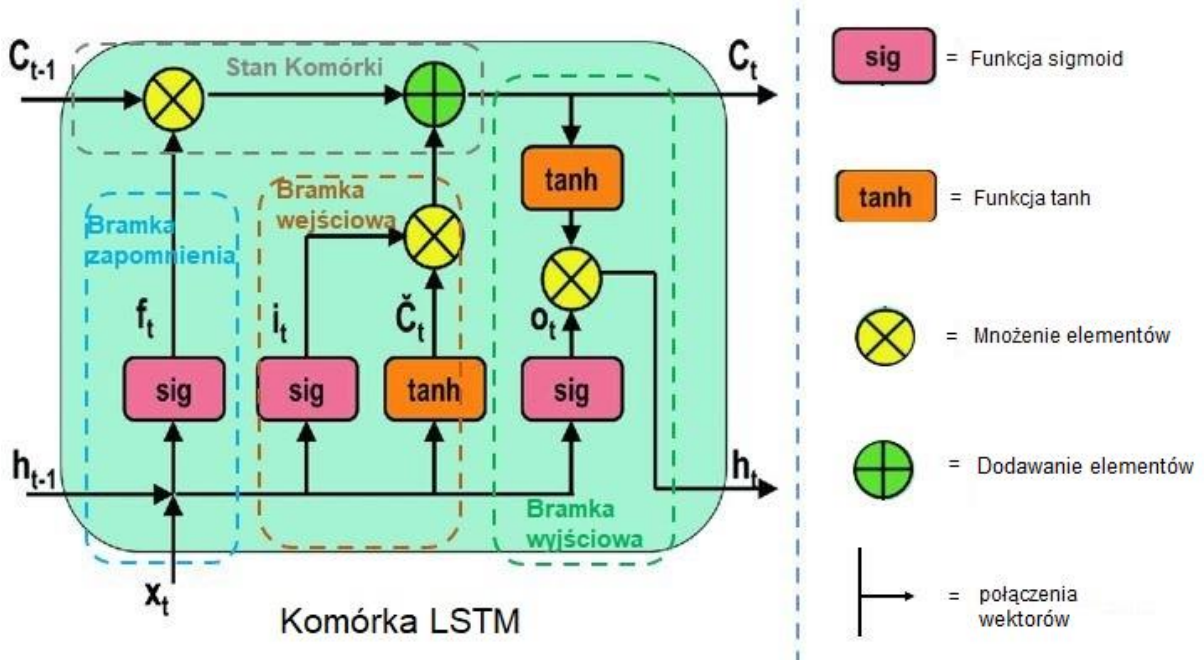
Kolejną niebezpieczną sytuacją są skrajnie duże lub małe wartości gradientu, które mogą się pojawić podczas uczenia tego rodzaju sieci. Aby poradzić sobie z tym problemem, początkowo stosowano gradient clipping w propagacji wstecznej, aby kontrolować maksymalne wartości gradientu czego przykład ukazuje rysunek 14 [17]. Po utworzeniu nowych bramek mających na celu ograniczenie zjawiska eksplodujących gradientów a przede wszystkim po zastosowaniu mechanizmu uwagi ograniczenie wartości gradientów spadło na znaczeniu.



Rysunek 14 Przykład ograniczenia wartości gradientu w propagacji wstecznej [17]

3.4 Long Short Term Memory

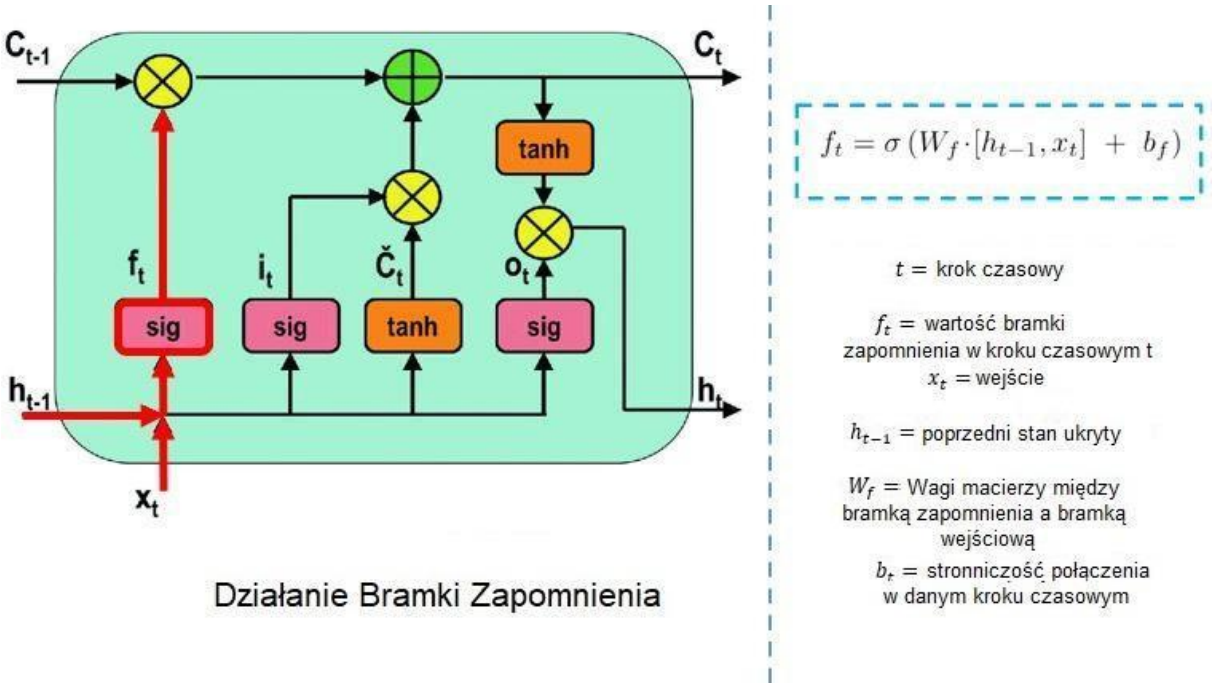
Mimo wysokiej skuteczności sieci rekurencyjnych prawdziwa rewolucja nadeszła wraz ze stworzeniem warstw LSTM (long short term memory) i GRU (grant recurrent unit), które znacznie usprawniły działanie sieci sekwencyjnych dzięki swoim bramkom [19]. Budowa sieci LSTM została zaprezentowana na rysunku 15.



Rysunek 15 Budowa bramki lstm [19]

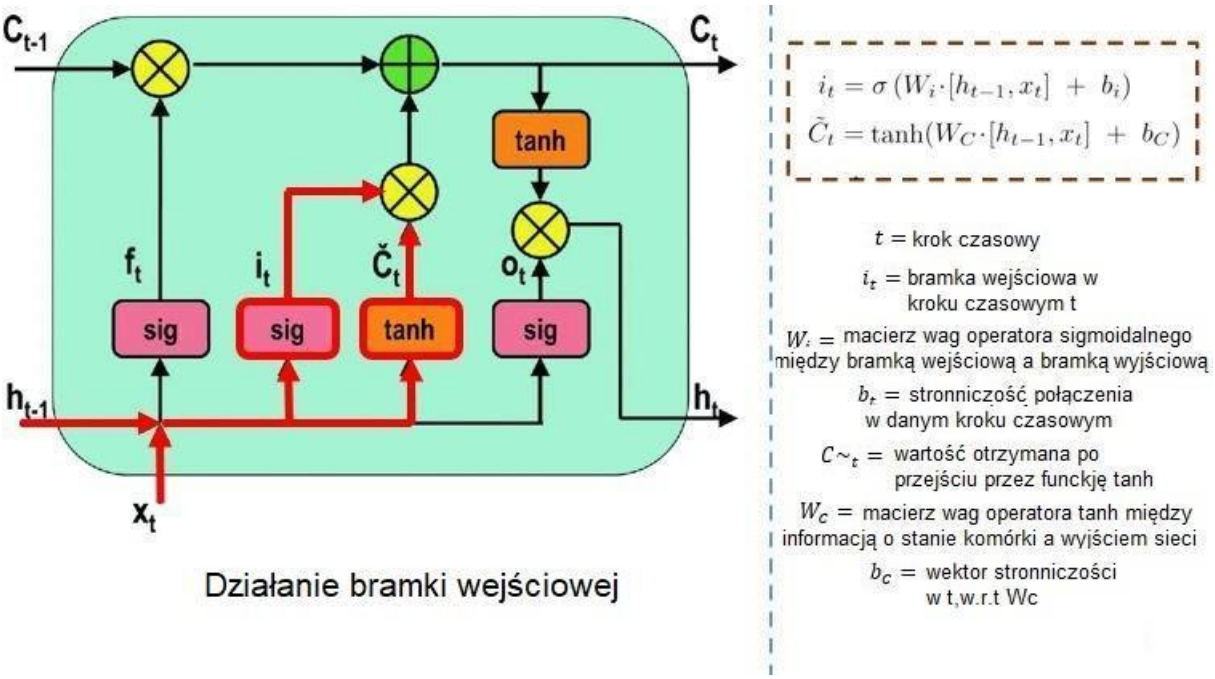
Wyróżnia się kilka rodzajów takich bramek:

bramka zapomnienia - przedstawiona na rysunku 16 decyduje, które informacje wymagają uwagi, a które można zignorować, Wartości z wejścia i ukrytego stanu są przekazywane do funkcji sigmoid [19].



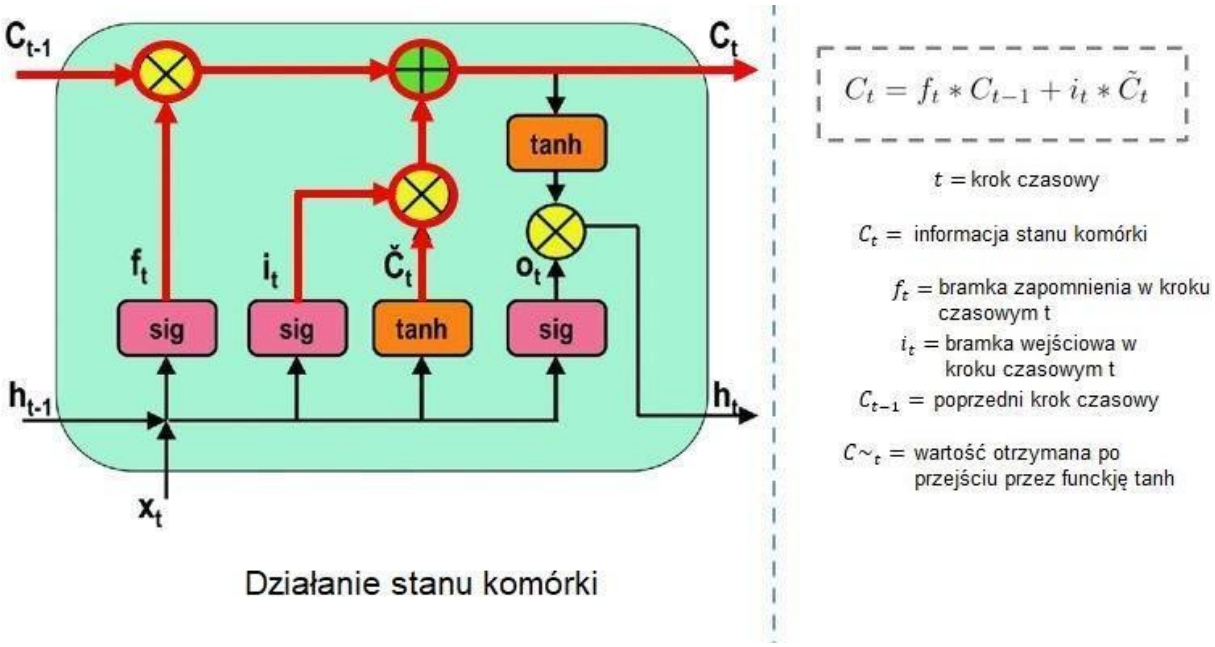
Rysunek 16 Schemat bramki zapomnienia [19]

bramka wejściowa - przedstawiona na rysunku 17 podejmuje decyzje, które wartości będą aktualizowane poprzez przepuszczenie wartości na wejściu, jak również ostatniego ukrytego stanu przez kolejną funkcję sigmoid. Funkcja tangensa hiperbolicznego umożliwia podjęcie decyzji w sprawie dodania do stanu określonych wartości [19].



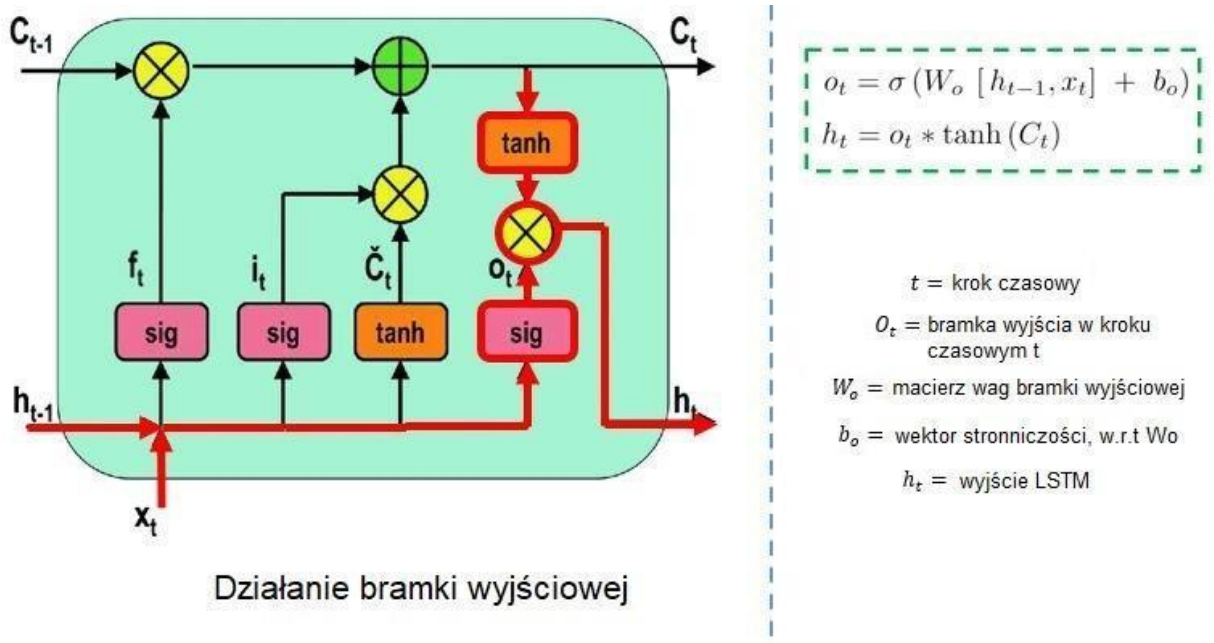
Rysunek 17 Schemat bramki wejściowej [19]

stan komórki - Ponieważ sieć posiada wystarczającą ilość informacji z bramki zapominania i bramki wejściowej, pozwala to na podjęcie decyzji i przechowywanie informacji z nowego stanu w stanie komórki przedstawionej na rysunku 18. Poprzedni stan komórki jest mnożony przez wektor zapominania. W przypadku wyniku zerowego wartości zostają usunięte ze stanu komórki. Dalej sieć pobiera wartość wyjściową wektora wejściowego, dodając punkt po punkcie, co aktualizuje stan komórki, nadając sieci nowy stan komórki [19].



Rysunek 18 Schemat stanu komórki [19]

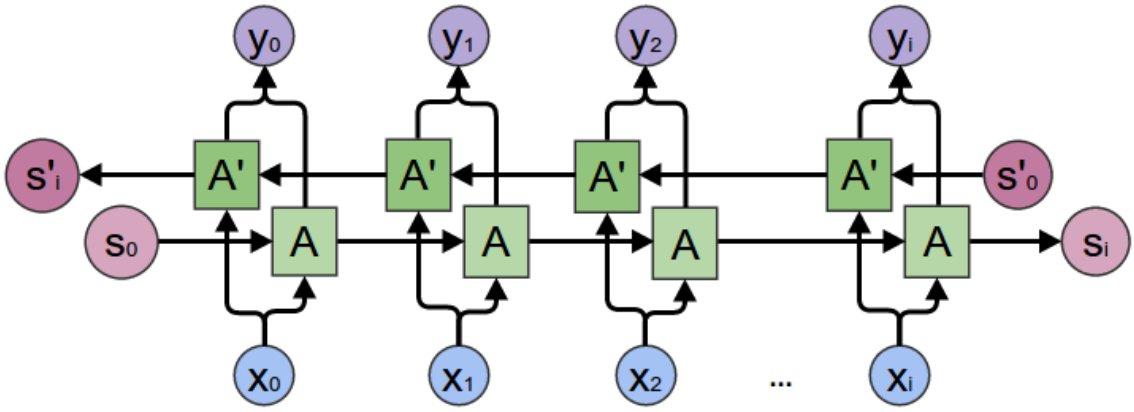
bramka wyjściowa - Wartości bieżącego stanu i poprzedniego stanu ukrytego są przekazywane do trzeciej funkcji sigmoidalnej. Następnie nowy stan komórki wygenerowany ze stanu komórki przechodzi przez funkcję tanh. Oba te wyjścia są mnożone punkt po punkcie. Na podstawie końcowej wartości sieć decyduje, jakie informacje powinien nieść stan ukryty. Ten ukryty stan służy do przewidywania. Finalnie nowy stan komórki i nowy stan ukryty są przenoszone do następnego kroku czasowego [19]. Schemat bramki wyjściowej przedstawiony został na rysunku 19.



Rysunek 19 Schemat bramki wyjściowej [19]

3.5 Warstwa dwukierunkowa

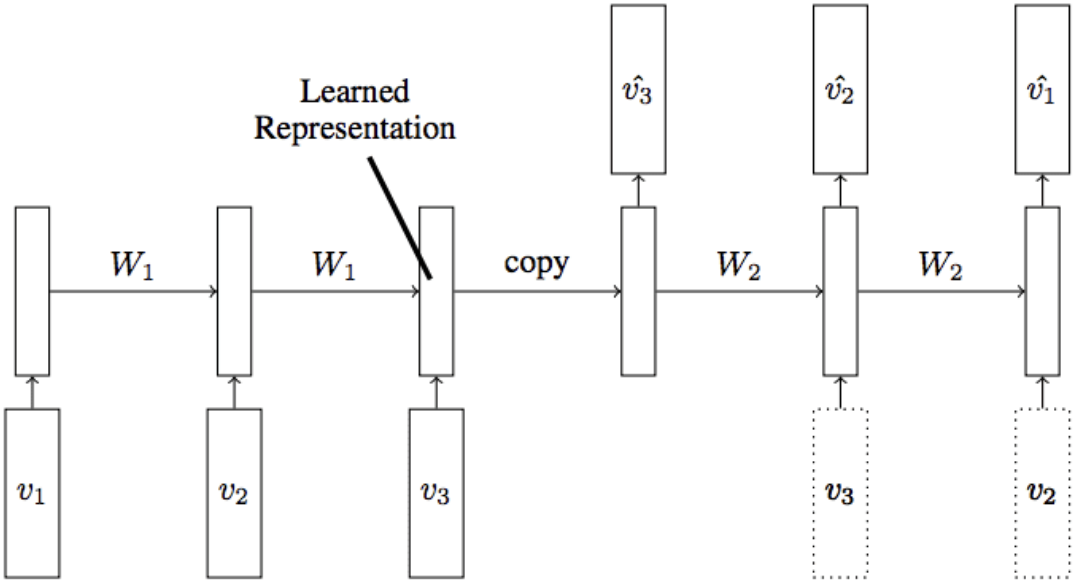
Zwykłe warstwy rekurencyjne pobierają wyłącznie poprzednie wartości, jak również aktualną, jednak czasami należy zwrócić uwagę na wartości, które w tekście mogą dopiero się pojawić. W tym celu dopuszczalne jest użycie warstw dwukierunkowych, aby umożliwić analizę danych wejściowych od przodu jak również w kierunku przeciwnym. BRNN przestawiona na rysunku 20 jest kombinacją dwóch RNN. Jedna sieć RNN porusza się do przodu, zaczynając od początku sekwencji danych, a drugi porusza się wstecz, zaczynając od końca sekwencji danych. Bloki sieciowe w BRNN mogą być prostymi RNN, GRU lub LSTM [20].



Rysunek 20 Warstwa dwukierunkowa [20]

3.6 Struktura Auto-Enkoder

Auto-Enkoder to nienadzorowana sztuczna sieć neuronowa, która uczy się, jak skutecznie kompresować i kodować dane, a następnie uczy się, jak zrekonstruować dane ze zredukowanej zakodowanej reprezentacji do reprezentacji, która jest jak najbardziej zbliżona do oryginalnego sygnału wejściowego. Auto Enkoder z założenia redukuje wymiary danych, ucząc się ignorowania szumu w danych [21]. Strukturę wspomnianej sieci prezentuje rysunek 21.



Rysunek 21 Struktura LSTM Auto Enkoder [21]

4 Biblioteki

Własna implementacja oraz testowanie działania warstw sieci może zająć dużą ilość czasu, oraz być obarczona wieloma błędami wpływającymi na jakość działania sieci. Dlatego współcześnie do budowy sieci używa się już gotowych oraz przetestowanych rozwiązań zawartych w bibliotekach dedykowanych uczeniu maszynowemu. Zawierają one również istotne funkcje do przetwarzania danych oraz analizy jakości modelu.

4.1 Tensorflow

Tensorflow to otwartoźródłowa biblioteka programistyczna napisana przez Google Brain Team i wydana 9 listopada 2015 roku, wykorzystywana jest w uczeniu maszynowym i głębokich sieciach neuronowych. Biblioteka składa się z kilku modułów. W jej najniższej warstwie znajduje się rozproszony silnik wykonawczy, który w celu podniesienia wydajności został zaimplementowany w języku programowania C++. Nad nią znajdują się nakładki napisane w kilku językach programowania m.in. w Pythonie. Jedną z największych zalet Tensorflow jest możliwość pracy na kartach graficznych, procesorach, jak również wyspecjalizowanych mikroprocesorach nazywanych akceleratorami AI. Możliwość obliczeń na kartach graficznych w szczególności na architekturze obliczeniowej CUDA znacząco przyspiesza uczenie [22]. Biblioteka tensorflow użyta została do budowy modelu. Zawiera ona wszystkie warstwy konieczne do zbudowania sieci rekurencyjnej.

4.2 Keras

Keras to interfejs API sieci neuronowej wysokiego poziomu umożliwiający uruchamianie innych popularnych platform sieci DNN w celu uproszczenia programowania [23]. Zaprojektowany, aby umożliwić szybkie eksperymentowanie z głębokimi sieciami neuronowymi, skupia się na byciu przyjaznym dla użytkownika, modułowym i rozszerzalnym. Kiedyś biblioteka obsługiwała wiele backendów, jednak aktualnie wspierany jest wyłącznie Tensorflow. Znacząco ułatwia on pracę ze wskazaną biblioteką, jednak jednocześnie ogranicza jej wydajność [24].

4.3 Scikit-Learn

Scikit-learn, czasami nazywana sklearn, to biblioteka open source uczenia maszynowego dla języka programowania Python. Zawiera ona różne algorytmy

klasyfikacji, regresji i klastrowania. Dostarcza ona narzędzi, przy pomocy których można wykonać wstępne przetwarzanie danych, dokonać wyboru modelu oraz ocenić jego jakość. Biblioteka ta zapewnia dziesiątki wbudowanych algorytmów i modeli uczenia maszynowego, zwanych estymatorami. Każdy estymator może być dopasowany do pewnych danych przy użyciu jego metody dopasowania. Metoda dopasowania przyjmuje dwa wejścia: Macierz próbek X oraz Wartość docelowa Y [25]. Biblioteka została wykorzystana do podziału danych na dane testowe i treningowe oraz do ewaluacji jakości modelu.

4.4 Google colaboratory

Google Colaboratory to środowisko Jupyter dostarczane i obsługiwane przez Google z możliwością pracy z procesorami, układami GPU, a nawet TPU. Działa niczym każdy inny Jupyter notebook. Wirtualna maszyna, udostępniana przez google colabolatory posiada system Linux, kartę graficzną NVidia Tesla K80 posiadającą 12GB pamięci RAM, jak również kilka ważnych funkcji takich jak integracja z github, niezależność platformy, jak również dostęp do darmowych zasobów [26]. Środowisko google colabolatory posłużyło głównie do treningu modelu dzięki udostępnianym kartą graficznym z dużą ilością pamięci. Wspomniane środowisko również znacząco ułatwia pracę z przetwarzaniem danych dzięki dużej ilości domyślnie zainstalowanych bibliotek.

5 Przetwarzanie wstępne danych

5.1 Zbiór danych

Dane pozyskane do treningu pochodzą z bazy danych CodeNet, która posiada 14 milionów próbek kodu, z których każda jest zamierzonym rozwiązaniem jednego z 4000 problemów związanych z kodowaniem. Próbki kodu są pobierane z dwóch witryn internetowych: AIZU Online Judge i AtCoder zawierających wiele różnych problemów programistycznych głównie na potrzeby konkursów programistycznych. Próbki kodu są napisane w ponad 50 językach programowania [35]. Ponieważ część problemów została napisana w innym języku niż angielski, za pomocą biblioteki “polyglot” został wykryty oryginalny język zadania, a następnie przy użyciu biblioteki “deep-translator” opisy problemów programistycznych zostały przetłumaczone na język angielski. Do danych został dołączony plik csv z informacjami dotyczącymi między innymi statusu akceptacji zgłoszonych zadań, języka programowania, w jakim zgłoszenie zostało wysłane, jak również informacje na temat wydajności kodu pod względem prędkości oraz wykorzystanej pamięci. Na podstawie tych danych wyselekcjonowane zostały zaakceptowane zgłoszenia w języku C. Wyrazy w języku naturalnym można łatwo podzielić na tokeny ze względu na odstęp między słowami. Ponieważ w kodzie źródłowym odstęp rozdzielający różne znaki nie zawsze występuje, do tokenizacji użyta została biblioteka “sctokenizer”. Następnie zostały utworzone 2 zbiory. Pierwszy, w którym wszystkie symbole leksykalne (ang. identifier) oprócz nazw funkcji biblioteki standardowej zostały zamienione na nazwy typu id: numer gdzie wartość numeru przyjmuje różne wartości. Drugi zbiór został pozostawiony bez zmian. Zmiana symboli w pierwszym zbiorze ma na celu zredukować wielkość słownika, jak również sprawdzić, czy działanie sieci się polepszy w momencie, gdy wszystkie zmienne oraz nazwy funkcji będą podobne. Przykłady danych treningowych zbioru CodeNet zostały zamieszczone na rysunku 22.

Przykłady opisów problemów w zbiorze CodeNet
You are given two positive integers A and B Compare the magnitudes of these numbers
Compute A + B
Write a program which calculates the cube of a given integer x
Print the circumference of a circle of radius R
Write a program which converts uppercaselowercase letters to lowercaseuppercase for a given string

Przykład kodu w języku C odpowiadającemu problemowi w zbiorze CodeNet
main ( ) { double a , b ; scanf (" %lf%lf " , & a , & b) ; puts (a - b ? a > b ? " GREATER " : " LESS " : " EQUAL ") ; }
main (a , b) { for (; ~ scanf (" %d%d " , & a , & b) ;) printf (" %d\n " , a + b) ; }
a ; main ( ) { scanf (" %d " , & a) ; a = ! printf (" %d\n " , a * a * a) ; }
main (r) { scanf (" %d " , & r) ; printf (" %f " , r * 6.28) ; }
main ( ) { char s ; for (; ~ scanf (" %c " , & s) ; putchar (64 < s && s < 91 ? s + 32 : 96 < s && s < 123 ? s - 32 : s)) ; exit (0) ; }

Przykład kodu w języku C odpowiadającemu problemowi w zbiorze CodeNet po zamianie symboli leksykalnych
id1 ( ) { double id0 , id2 ; scanf (" %lf%lf " , & id0 , & id2) ; puts (id0 - id2 ? id0 > id2 ? " GREATER " : " LESS " : " EQUAL ") ; }
id1 (id0 , id2) { for (; ~ scanf (" %d%d " , & id0 , & id2) ;) printf (" %d\n " , id0 + id2) ; }
id0 ; id1 ( ) { scanf (" %d " , & id0) ; id0 = ! printf (" %d\n " , id0 * id0 * id0) ; }
id1 (id0) { scanf (" %d " , & id0) ; printf (" %f " , id0 * 6.28) ; }
id1 ( ) { char id0 ; for (; ~ scanf (" %c " , & id0) ; putchar (64 < id0 && id0 < 91 ? id0 + 32 : 96 < id0 && id0 < 123 ? id0 - 32 : id0)) ; exit (0) ; }

Rysunek 22 Przykłady danych treningowych w zbiorze CodeNet

Dane dla modeli porównawczych dla języka python oraz java pochodzą z bazy danych CodeXGLUE (General Language Understanding Evaluation benchmark for CODE) stworzonej przez Microsoft. Zadania są typu kod-kod (wykrywanie klonów, wykrywanie defektów, test luk, refaktoryzacja kodu, udoskonalanie kodu i tłumaczenie kodu na kod), tekst-kod (wyszukiwanie kodu w języku naturalnym, generacja kodu przy użyciu

języka naturalnego), kod-tekst (podsumowanie kodu) i tekst-tekst (tłumaczenie dokumentacji) [27]. Ze względu na rodzaj zadania trening został przeprowadzony na dwóch zestawach danych. Pierwszy pochodzi ze zbioru CodeSearchNet drugi zaś ze zbioru CONCODE. Zawierają one kolejno 251 tysięcy oraz 100 tysięcy opisów różnych problemów programistycznych wraz z odpowiadającym im kodem źródłowym. Język kodu różni się w zależności od zbioru. W zbiorze CodeSearchNet kod napisany został w języku python, a dla CONCODE w języku java. Pomimo że oba zbiory posiadają opis problemu w języku naturalnym dla danych wejściowych oraz odpowiadający problemowi kod jako dane wyjściowe, ze względu na charakter zadania opisy problemów w zbiorach znacząco się różnią. W obu zbiorach opisy pochodzą z dokumentacji (docstrings, JavaDoc i inne) projektów typu open-source na platformie github, jednak w przypadku zbioru CONCODE opisy problemów są długie a odpowiadający im kod krótki, dla zbioru CodeSearchNet jest zupełnie przeciwnie, co zostało zobrazowane na rysunku 25. Przykłady omawianych danych zostały zamieszczone na rysunkach 23 oraz 24 jednak w celu zachowania przejrzystości niektóre z nich zostały skrócone.

Przykład opisu w zbiorze CONCODE	Przykład kodu odpowiadającemu opisowi w zbiorze CONCODE
retrieves the table object on which this object is currently reporting .	Table function ( ) { return this . table ; }
return the current time associated with this receiver .	Time function ( ) { IOPort loc0 = getContainer ( ) ; Actor loc1 = (Actor) loc0 . getContainer ( ) ; Director loc2 = loc1 . getDirector ( ) ; return loc2 . getModelTime ( ) ; }

adds the given callback to the task . when the api call completes , via the get method , the api sends the data to the callback . note : you can add multiple callbacks .	GetMessagesTask function (
	BriefMessengerCallback < List < Message >> arg0) {
	if (arg0 != null) { mCallbacks . add (arg0) ; } return this ; }

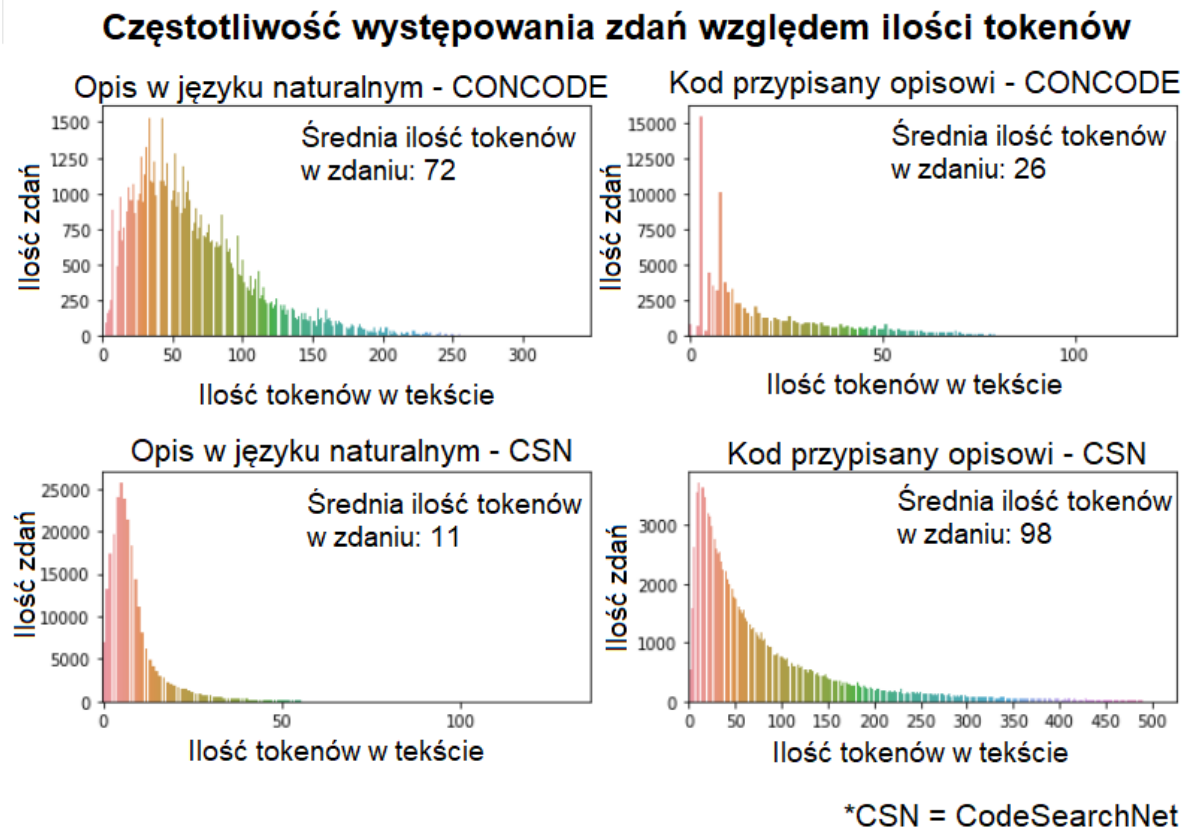
appends the specified value to the end of this list .	void function (double arg0) { ensureCapacity (size + 1) ; data [size ++] = arg0 ; }
specify the types of node which should be merged .	void function (Set < Integer > arg0) { this . nodeTopoToMerge . addAll (arg0) ; }

Rysunek 23 Przykłady danych treningowych w zbiorze CONCODE

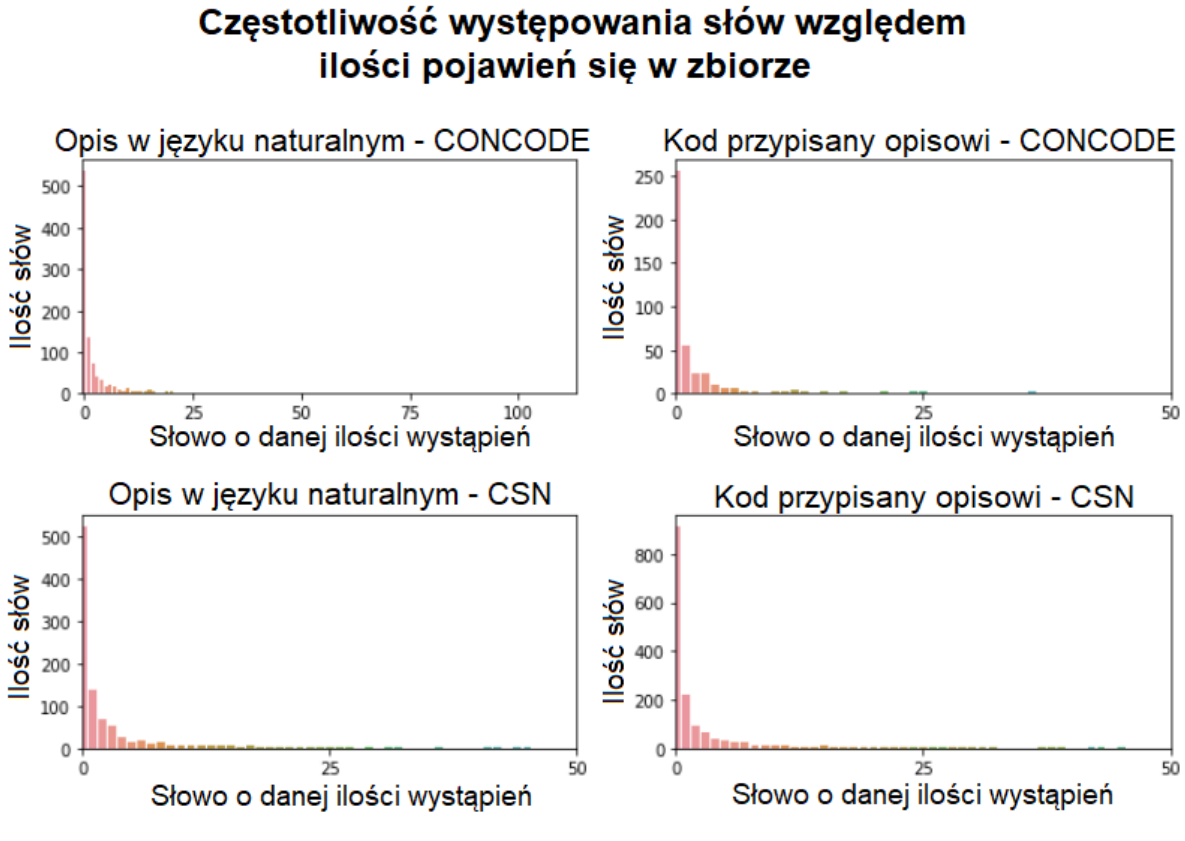
Przykład opisu w zbiorze CodeSearchNet	Przykład kodu odpowiadającemu opisowi w zbiorze CSN
Change the clients username .	def switch_user (self , username , password) : self . _username = username self . _password = password
Move Rectangle to x y coordinates	def move (self , x , y) : self . x = x self . y = y
Log out revoking the access tokens and forgetting the login details if they were given .	def logout (self) : self . revoke_refresh_token ( ) self . revoke_access_token ( ) self . _username ,
Takes either a file path or an open file handle checks validity and returns an open file handle or raises an appropriate Exception .	def file_handle (fnh , mode = "rU") : handle = None if isinstance (fnh , file) : if fnh . closed : raise ValueError ...
Return either the full or truncated version of a QIIME - formatted taxonomy string .	def split_phylogeny (p , level = "s") : level = level + "__" result = p . split (level)

Rysunek 24 Przykłady danych treningowych w zbiorze CodeSearchNet

Istotna w przypadku treningu sieci jest liczba unikalnych słów. Z jednej strony, jeśli będzie ich zbyt mało, sieć nie będzie w stanie nauczyć się generalizować pewnych rzeczy, jednak w przypadku predykcji zwiększa się liczba możliwych do przewidzenia słów, co przekłada się na wielkość modelu jak również zwiększa ryzyko błędów. Analiza obu zbiorów przedstawiona na rysunku 26 wykazała, że większość słów pojawia się wyłącznie raz lub niewielką liczbę razy. Dotyczy to zarówno kodu źródłowego, jak również opisu w języku naturalnym. Ponieważ słowa rzadko się powtarzają, jest to dodatkowe wyzwanie dla sieci, która ma wysokie prawdopodobieństwo na nauczenie się słów w zbiorze treningowym, które ponownie nie wystąpią w przypadku innych przykładów, jak również istnieje duże ryzyko pojawienia się słów w zbiorze testowym, z którymi sieć jeszcze nie miała styczności.



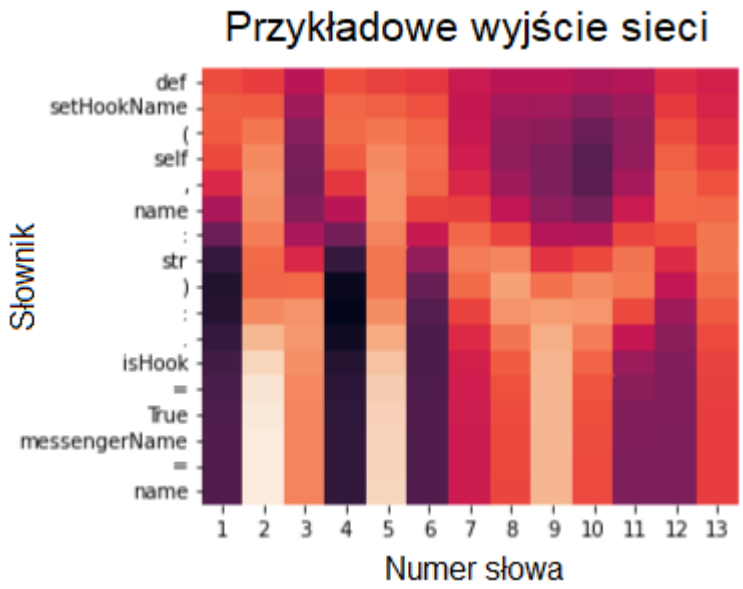
Rysunek 25 Wykres częstotliwość występowania zdań względem ilości tokenów



Rysunek 26 Wykres częstotliwość występowania słów względem ilości pojawień w zbiorze

5.2 Tokenizacja danych

Ponieważ sieć nie jest w stanie przetwarzać tekstu, należy stworzyć jego numeryczną reprezentację. Można to osiągnąć na wiele sposobów, jednak ze względu na przebieg treningu warto zwrócić uwagę, jak sposób tokenizacji wpływa na liczbę elementów w zbiorze uczącym, jak również na zasób słownika. Aby lepiej zobrazować problem na rysunku 27, przedstawione zostało przykładowe wyjście sieci, gdzie dla każdego słowa na wyjściu w kolejności od 1 do 13 sieć przewiduje token ze słownika. Prawdopodobieństwo wyboru tokenu zostało odzwierciedlone w jasności koloru. Im jaśniejszy kolor, tym większe prawdopodobieństwo wystąpienia tokenu na danej pozycji.

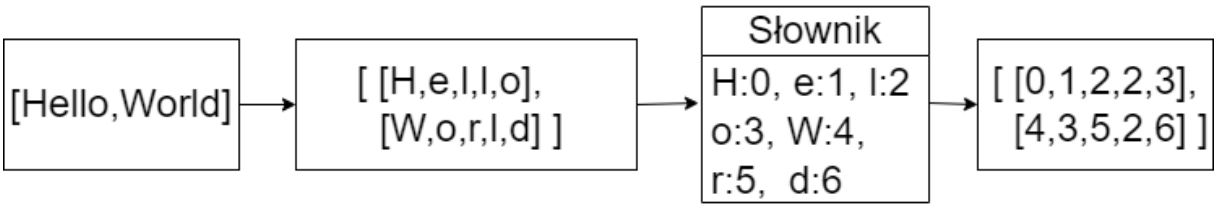


Rysunek 27 Przykładowe wyjście sieci

Pierwszym ze sposobów, który został zastosowany, polegał na stworzeniu słownika pojawiających się w zbiorze słów i przypisaniu im unikalnego identyfikatora. Przykład tokenizacji opartej na słowach przedstawia rysunek 28.

Rysunek 28 Przykład tokenizacji metodą opartej na słowach

Sposób ten działa w przypadku małej liczby danych, w których zasób słów jest ograniczony, jednak problemy pojawiają się w przypadku dużych zbiorów danych. Pomimo że liczba słów po zamianie w tokeny pozostaje niezmienna, zasób słów słownika jest w stanie urosnąć do poziomu, w którym sieć będzie nieskuteczna. Szczególnie problemy tej metody można zauważyć w przypadku zamiany kodu źródłowego na tokeny gdzie nazwy zmiennych lub funkcji mogą składać się z kilku słów połączonych znakiem podkreślenia np. “my_custom_variable”. W przypadku tej metody zostaną one potraktowane, jako nowe słowa a zasób słownika może dążyć do nieskończoności. Kolejną metodą, która rozwiązuje wspomniany problem, jest tokenizacja oparta na znakach. Metoda ta działa podobnie do poprzedniej, jednak wszystkie wyrazy zostają podzielone na pojedyncze znaki. Przykład tokenizacji pojedynczych znaków zawiera rysunek 29.



Rysunek 29 Przykład tokenizacji metodą opartej na znakach

Pomimo że w prezentowanym przykładzie liczba elementów w słowniku zwiększyła się, w przypadku zbioru z dużą liczbą słów zasób słownika będzie ograniczony ze względu na stałą liczbę możliwych do użycia znaków. Jednak już na przedstawionym przykładzie widać, że metoda ta wiąże się ze znacznym wzrostem liczby utworzonych tokenów. Nawet w przypadku użycia bramek LSTM sieci mają problem w radzeniu sobie z bardzo długimi sekwencjami. Ostatnia użyta metoda stanowi hybrydę między dwoma poprzednimi, dając najlepsze wyniki. Polega na podziale często używanych słów na złożone z nich podsłowa. Można tu wyróżnić kilka znanych algorytmów [36]:

- Byte-Pair Encoding [36]
- Byte-level BPE [36]
- WordPiece [36]
- Unigram [36]
- SentencePiece [36]

Wymienione algorytmy są do siebie podobne. Byte-Pair Encoding polega na doborze najczęściej występującej pary symboli. Byte-level BPE działa analogicznie jednak

podstawowy słownik jest ograniczony do 256 symboli. WordPiece dobiera pary według specjalnego wzoru, który ma na celu łączyć symbole rzadko występujące w słowniku. Unigram w przeciwieństwie do poprzednio omówionych zaczyna na dużej ilości słownictwa, a następnie iteracyjnie je zmniejsza. SentencePiece powstał głównie z myślą o danych, które nie są separowane spacją [36].

Dla wybranych do treningu zestawów został zastosowany WordPiece tokenizer, którego przykład użycia obrazuje rysunek 30.

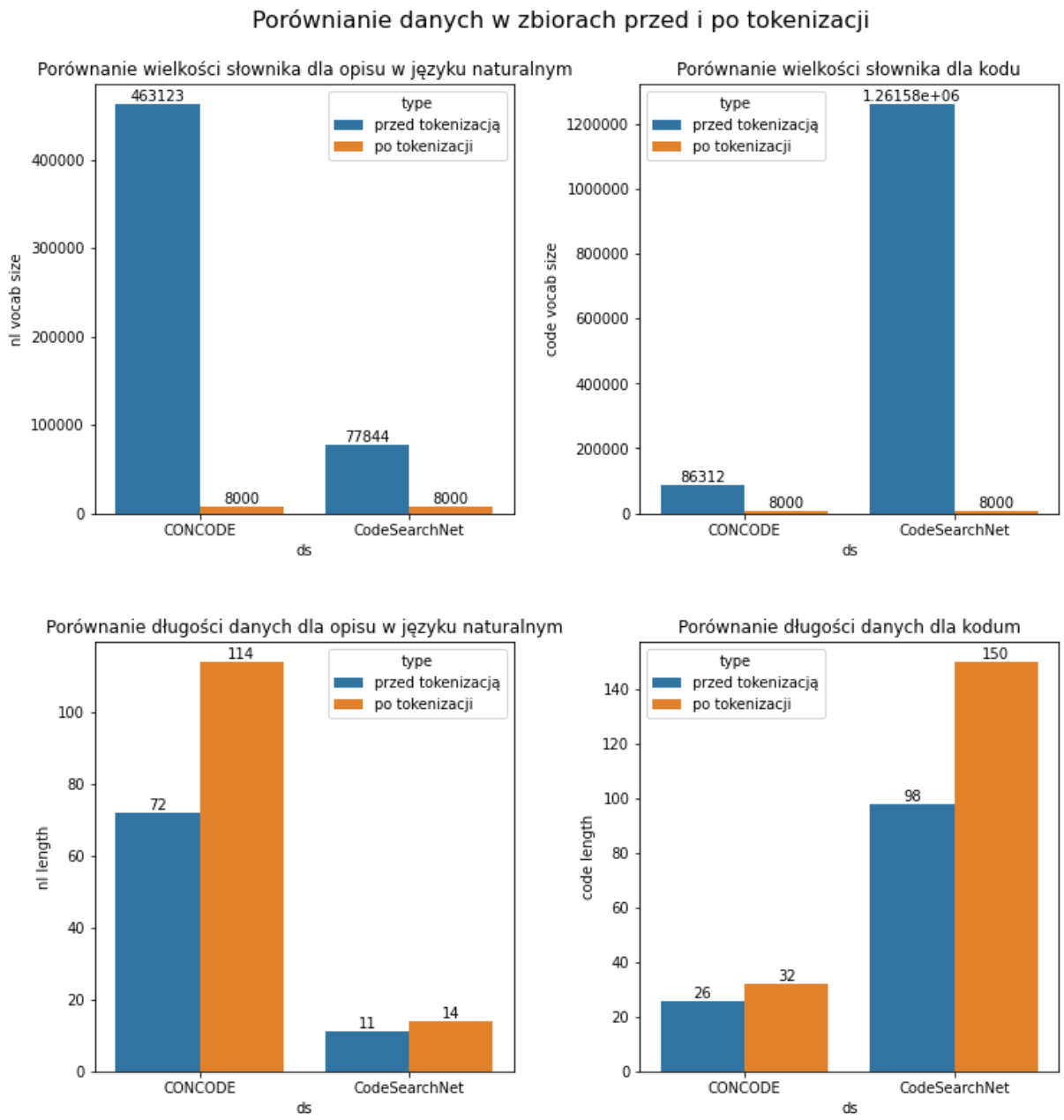
Word	Token(s)
surf	['surf']
surfing	['surf', '##ing']
surfboarding	['surf', '##board', '##ing']
surfboard	['surf', '##board']

Rysunek 30 Przykład podziału słów metodą WordPiece [28]

WordPiece dobiera symbole poprzez przypisanie im punktów według wzoru 5 [30], który oblicza liczbę wystąpień (Fr) pary symboli we wszystkich wyrazach w zbiorze, a następnie dzieli je przez iloczyn liczby wystąpień pierwszego oraz drugiego symbolu w zbiorze. Następnie wybiera parę z największą liczbą punktów, która zostaje dodana do słownika. Ma to na celu łączyć symbole rzadko występujące w słowniku [29]. Początkowo alfabet zawiera wszystkie znaki obecne na początku słowa oraz znaki obecne w słowie poprzedzone specjalnym przedrostkiem. Następnie WordPiece uczy się reguł scalania.

$$Wynik = \frac{Fr(pary)}{Fr(pierwszego\ elementu) * Fr(drugiego\ elementu)} \quad (5)$$

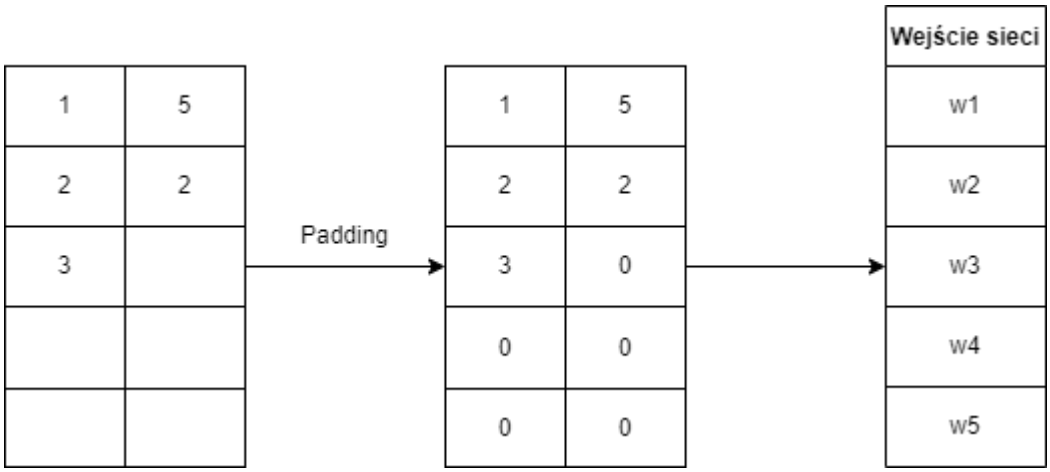
Tokenizacja zbioru została przeprowadzona przy użyciu biblioteki tokenizers. Zasób słownika dostosowany został manualnie w taki sposób, aby znacząco zmniejszyć zasób słów w słowniku jednocześnie ograniczając zbyt duży wzrost liczby danych. Rysunek 31 przedstawia porównanie wielkości słowników, jak również średniej ilości tokenów w zdaniu przed i po tokenizacji metodą word-level i WordPiece, z którego wynika znaczna redukcja wielkości słownika kosztem niewielkiej zmiany średniej liczby tokenów.



Rysunek 31 Porównania zbiorów przed i po tokenizacji

Ostatnią część przetwarzania wstępnego stanowi ujednolicenie wektorów podanych do sieci. Ze względu na fakt przyjmowania przez sieci danych o stałej długości, zdefiniowanej podczas budowy modelu, należy dostosować dane w taki sposób, aby były z nią zgodne. Można w tym celu zastosować kilka różnych sposobów, jednak najbardziej znanymi są kubełkowanie (ang. bucketing) oraz wypełnienie (ang. padding). Bucketing polegający na podziale danych ze względu na ich długość, a następnie trenowanie ich oddzielnie, jest skuteczną metodą, jednak jest bardzo trudną w treningu. Dlatego do treningu modelu zastosowana została druga metoda, która polega na dodaniu wypełnienia prawostronnego mającego na celu zwiększenie

długości wektora o określoną liczbę zer reprezentujących brak słowa aż do momentu ujednolicenia długości wszystkich wektorów na wejściu. Przykład działania wypełnienia znajduje się na rysunku 32.

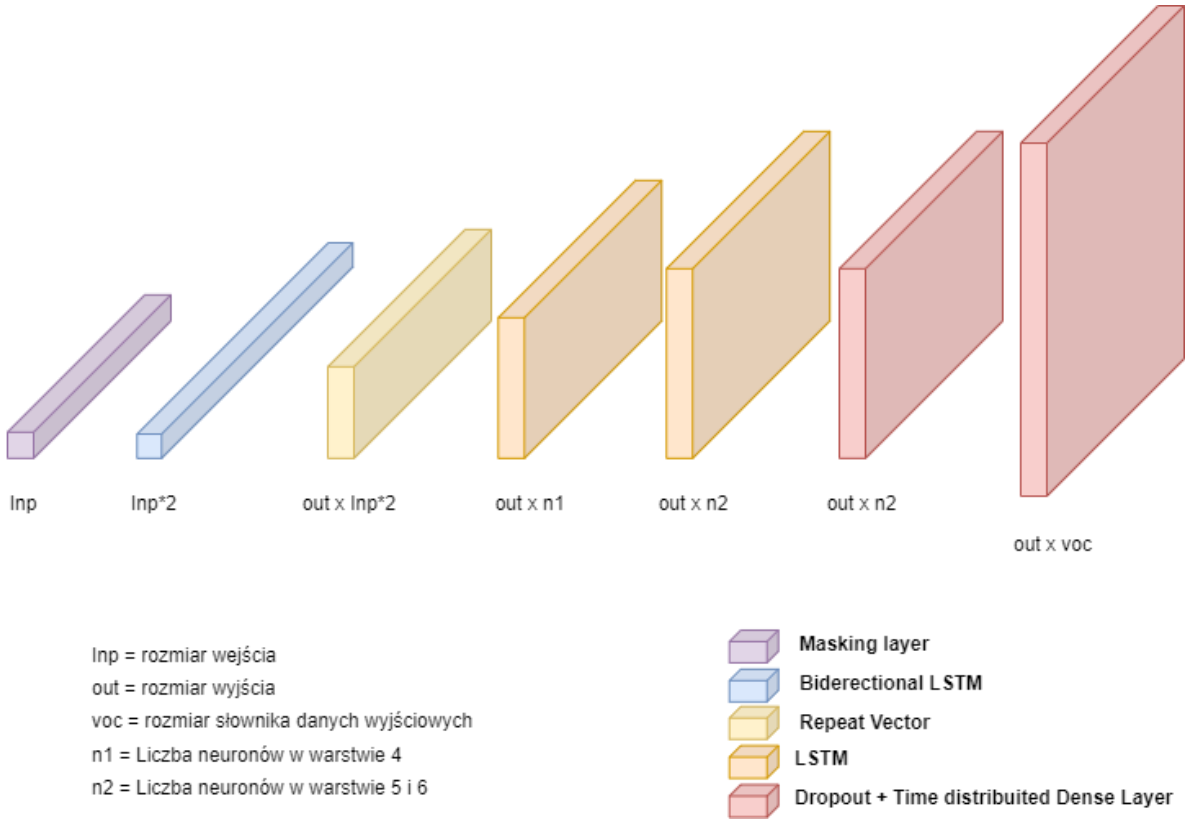


Rysunek 32 Przykład działania wypełnienia

6 Trening i generacja kodu

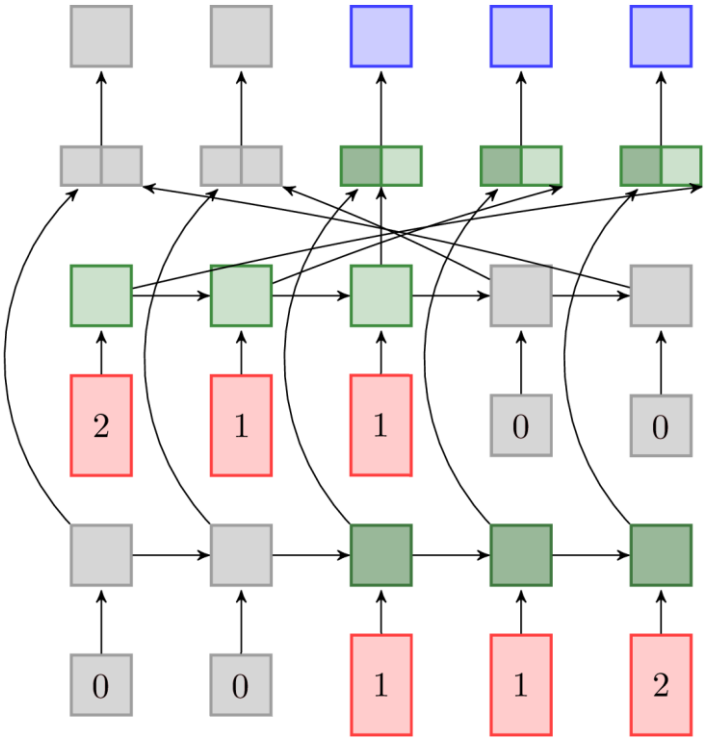
6.1 Model

Stworzenie odpowiedniej architektury stanowi podstawę dla odpowiedniego działania sieci. Podczas budowy sieci sekwencyjnych pojawia się kilka istotnych problemów takich jak zmienna długość wektorów wejściowych oraz wyjściowych w zależności od długości zdania, negatywny wpływ długich sekwencji na przetwarzanie przez sieć informacji, jak również duża ilość zasobów wymaganych szczególnie w przypadku dużych zbiorów danych. Sieć utworzono przy użyciu biblioteki tensorflow, która zawiera implementacje wszystkich warstw użytych w architekturze modelu, która przedstawiona została na Rysunku 33. Rozmiar warstwy wejściowej jest zależny od długości zdań przekazywanych na wejście, czyli po uwzględnieniu wypełnienia będzie to długość najdłuższego opisu problemu w zbiorze. Ponieważ pierwsza warstwa jest maskująca, kolejna warstwa musi również przyjąć dane o długości danych na wejściu. Wyjścia warstwy dwukierunkowej po połączeniu tworzą wektor o podwójnej długości wejścia. Ponieważ długość wyjścia różni się od długości wejścia, warstwa repeat vector ponawia przyjęte dane w celu dostosowania danych do długości danych wyjściowych. W celu skutecznego przetwarzania informacji przez sieć często stosuje się przynajmniej 3 warstwy, których wielkość dobiera się względem wejścia, wyjścia oraz wielkości zbioru treningowego. Jednak z uwagi na ograniczone zasoby pamięci oraz długi czas treningu zastosowano dwie warstwy LSTM o liczbie neuronów kolejno 512, oraz 1024. Warstwa wyjściowa analogicznie do warstwy wejściowej posiada długość najdłuższego kodu w zbiorze. Na każdy słowo na wyjściu przypada liczba słów w słowniku utworzonym na zbiorze kodów źródłowych.



Rysunek 33 Architektura utworzonego modelu

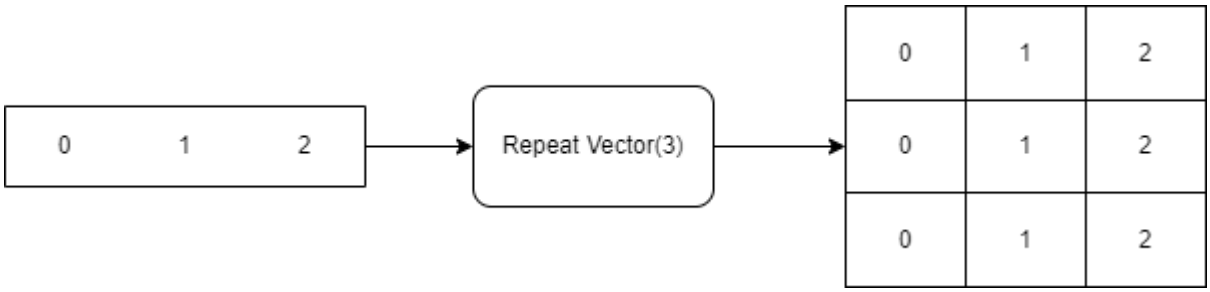
Ponieważ do ujednolicenia długości danych wejściowych, jak również wyjściowych został zastosowany padding, w sieci powstała duża liczba danych, które pozbawione są znaczenia, jednak nadal zostają uwzględnione w treningu sieci. Aby poradzić sobie z tym zjawiskiem można użyć warstwy maskującej, warstwy Embedding lub stworzyć własną warstwę, która mogłaby tę funkcję obsługiwać. Maskowanie we wspomnianych warstwach polega na rozpoznaniu i zignorowaniu wartości odpowiadającej padding-owi, czyli najczęściej zeru. Następnie stworzony zostaje wektor lub macierz w zależności od kształtu danych wejściowych zawierający wartość logiczną reprezentującą czy dany token jak zamaskowany, czy nie. Informacje te można propagować do kolejnych warstw, dzięki czemu również w nich maska wpływa na przetwarzanie informacji. W przypadku biblioteki tensorflow wszystkie warstwy użyte w architekturze wspierają maskowanie. Jeśli jednak jest to niemożliwe, można stworzyć własną warstwę uwzględniającą tę informację. Przykład maskowania przedstawiony został na rysunku 34. Prezentuje on przejście danych przez warstwę dwukierunkową. Dane na wejściu o wartości 0 reprezentujące padding zostały zamaskowane, co ukazuje szary kolor przy kolejnych blokach, co jest równoważne z pominięciem informacji przekazywanej przez te neurony.



Rysunek 34 Przykład działania maskowania [32]

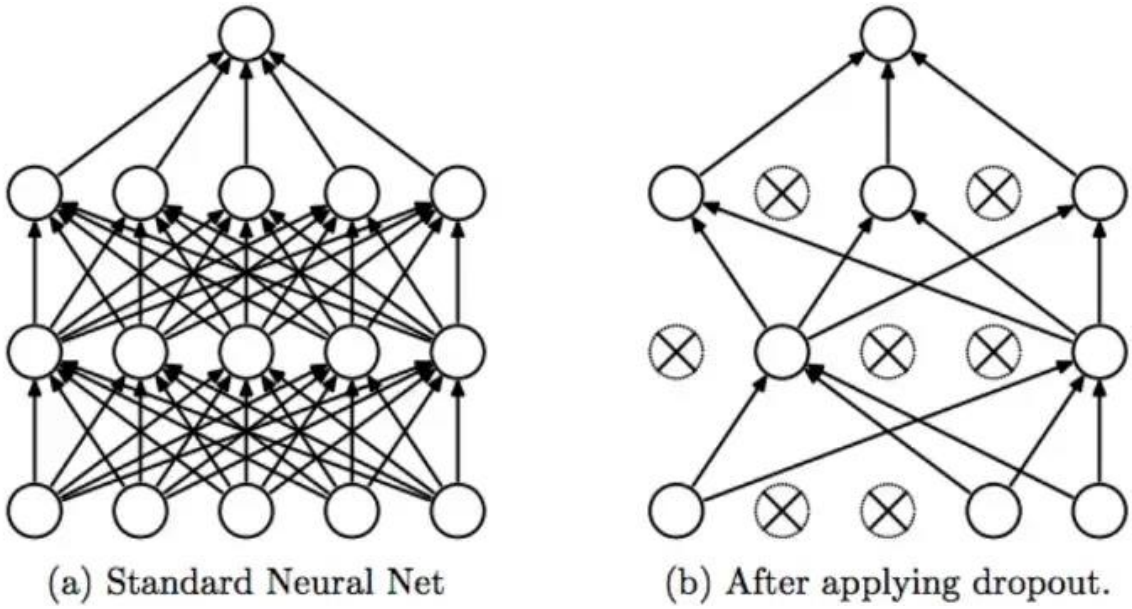
Początkowo zamiast warstwy maskującej użyta została warstwa Embedding, która daje sieci więcej informacji na temat występujących na wejściu słów, jednak wiąże się to również ze wzrostem liczby danych. Istotną kwestią jest również działanie warstwy Embedding, jeśli dane zostały tokenizowane algorytmem WordPiece. W takim wypadku tworzenie reprezentacji w przestrzeni wielowymiarowej dla podsłów wydaje się nieodpowiednie z uwagi na fakt, że ich znaczenie będzie zależne od kontekstu. Zastosowanie WordPiece wyklucza również użycie word2vec oraz glove. Ponieważ w zbiorach pojawiają się dość długie dane wejściowe, aby upewnić się, że sieć będzie je w pełni rozumieć, użyta zostaje dwukierunkowa warstwa LSTM z zastosowanym parametrem return_sequences ustawionym na stan False, dzięki czemu na wyjściu tego bloku otrzymany został wektor reprezentujący cechy z wejścia. Wyjściem wspomnianej warstwy jest wektor łączący stany ukryte z obu kierunków. W przypadku problemów z pamięcią stany te można zsumować. Ponieważ wyjście sieci jest macierzą, której rozmiar definiuje długość, jak również wielkość słownika danych wyjściowych, należy przekształcić rozmiar danych w taki sposób, aby odpowiadał rozmiarowi wyjścia sieci. W tym celu użyta zostaje warstwa powtórzeń wektora (ang. repeat vector) używana w sieciach many to many w celu radzenia sobie z tym problemem. Warstwa ta powtarza dane z wejścia odpowiednią liczbę razy, którą można zdefiniować. Zwiększa to jednocześnie wymiarowość sieci, co pozwala

zamienić wektor wejściowy w macierz z odpowiadającą wyjściu długość. Przykład działania omówionej warstwy znajduje się na rysunku 35.



Rysunek 35 Przykład działania warstwy powtórzeń wektora

Następnie użyte są dwie warstwy LSTM z parametrem return_sequences przyjmującym wartość True, dzięki czemu sieć zwraca sekwencję danych. Na koniec w celu utworzenia macierzy na wyjściu użyte zostają warstwy time distributed dense, które stosują warstwę dense do każdego kroku czasowego, dzięki czemu rozmiar sieci pozostaje odpowiedni, jak również nie zostają przekazywane informację między różnymi krokami czasowymi [33]. Dodatkowo zostały one poprzedzone warstwami dropout, które poprzez ignorowanie pewnych neuronów podczas propagacji do przodu, jak również wstecz ograniczają rozwój współzależności między neuronami, co redukuje zjawisko przetrenowania modelu [34] co obrazuje rysunek 36.

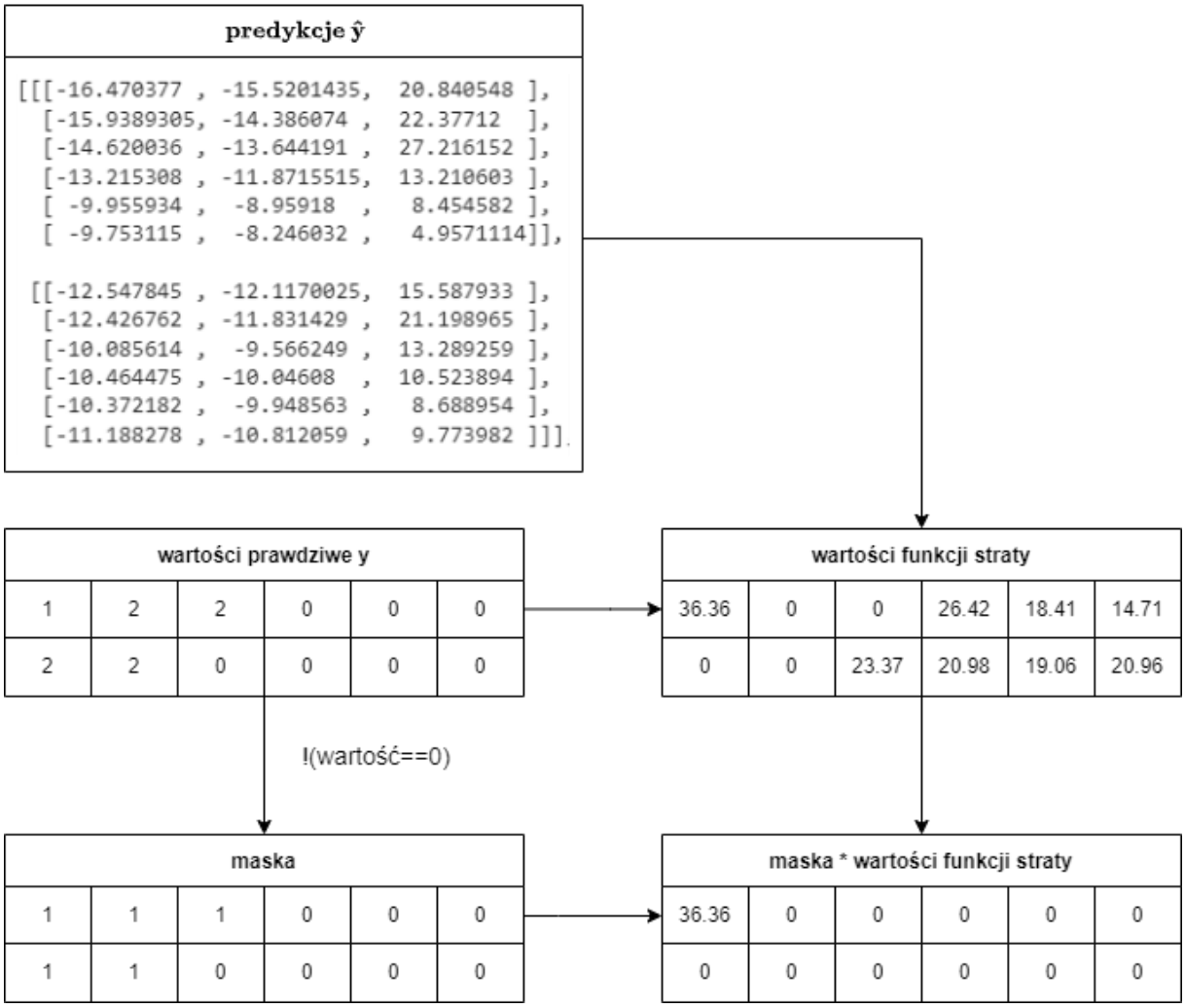


Rysunek 36 Przykład działania warstwy dropout [34]

6.2 Trening sieci

Do treningu użyte zostały trzy modele o podobnej strukturze, jedyne różnice występują w rozmiarze wejścia, jak również wyjścia. Każdy z modeli trenowany został na osobnym zestawie danych. Z uwagi na podobieństwo problemów można w takich sytuacjach próbować stosować fine tuning, jednak ze względu na odmienność słownictwa w zbiorach, jak również długości danych w zestawach, opcja ta jest niemożliwa. Dlatego do każdego ze zbiorów powstały dwa tokenizatory z oddzielnymi słownikami dla opisów w języku naturalnym i kodu. W przypadku zbioru z CodeSearchNeta powstały problemy przy treningu dlatego został użyty pre-trained BERT tokenizer z biblioteki transformers. Zasób słów dla całego zbioru w obu przypadkach wahał się między około 8000-10000 słów. Problem jednak pojawił się przy długości danych. Długość danych po tokenizacji oraz dodaniu paddingu sięgała około 1400 tokenów, co przy 251000 przykładów powodowało problemy z pamięcią. Dlatego ze zbiorów konieczne było usunięcie zbyt długich próbek, aby trening był możliwy. Dlatego został zastosowany filtr mający odrzucić dane, których długość wynosiła powyżej 150 tokenów w przypadku tekstu naturalnego dla zbioru z CONCODE, jak również to samo ograniczenie powstało dla przykładów kodu z CodeSearchNet. Liczba danych po wspomnianych modyfikacjach nadal pozostała wystarczająca, aby przeprowadzić trening sieci. Tak dostosowane dane zostały podzielone na dane treningowe, testowe, walidacyjne w proporcji 8:1:1. Następnie dane treningowe zostały wymieszane oraz podzielone na batche. Początkowo do treningu jako funkcja straty przyjęta została Sparse Categorical Cross Entropy, jednak pomimo słabych wyników sieci wartość straty była bardzo niska. Dzięki warstwie maskującej sieć była w stanie przetwarzać informację o paddingu w danych wejściowych, lecz problem pojawił się w przypadku paddingu w danych wyjściowych. Padding stanowił większość tokenów w danych wyjściowych, co sprawiało, że sieć klasyfikowała poprawnie mimo słabych rezultatów, dlatego została nałożona dodatkowa maska na funkcję straty, która ma na celu wyzerować wartości dla wszystkich tokenów reprezentujących padding. Na koniec wartość funkcji została uśredniona z całego batcha. Omówioną czynność przedstawia rysunek 37 gdzie na podstawie predykcji $\hat{y}$ oraz wartości prawdziwych y obliczona została wartość funkcji straty. Poprzez przekazanie wartości prawdziwych y do funkcji porównania gdzie każda wartość zostaje zestawiona z zerem, a następnie do funkcji negacji otrzymana

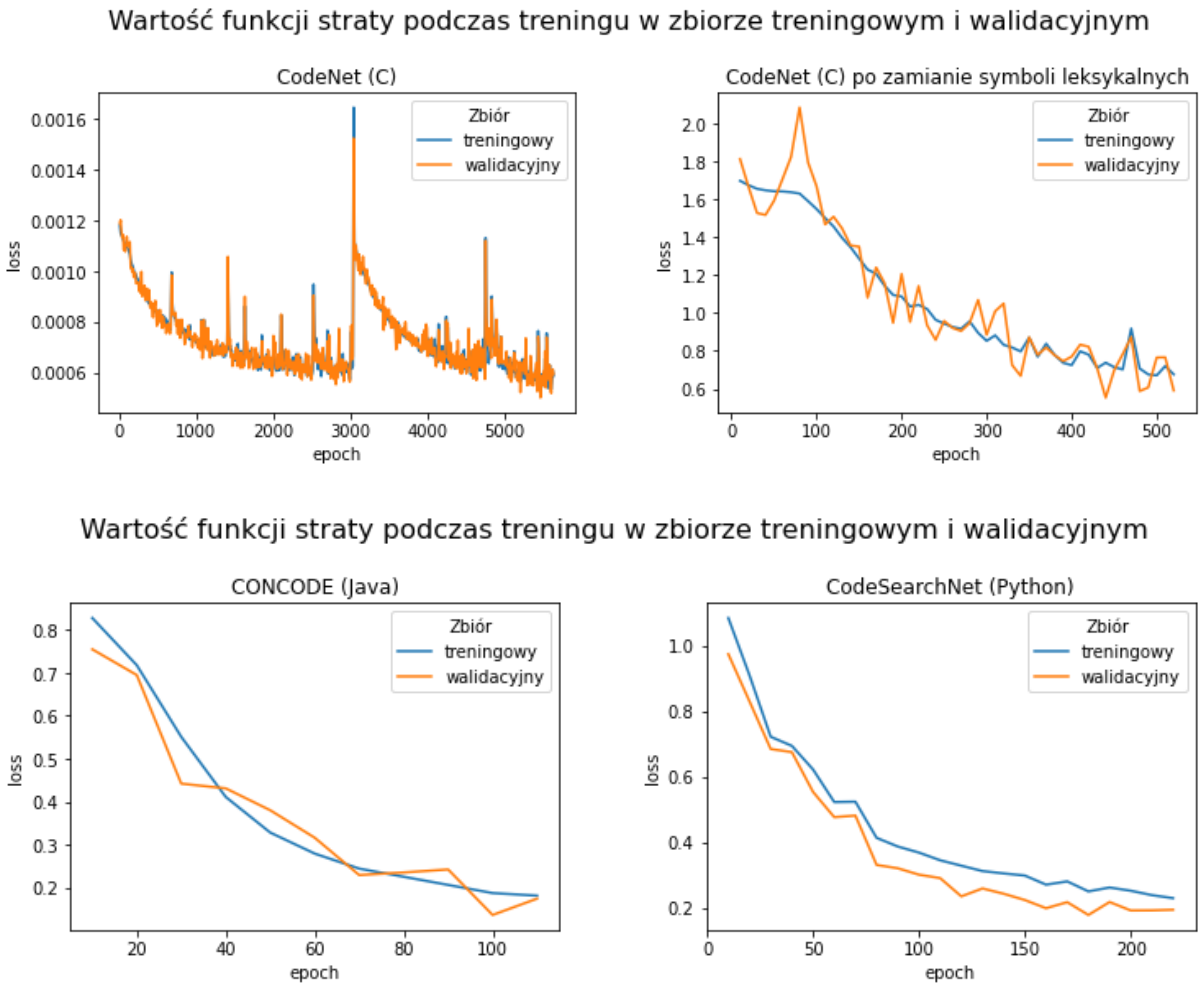
została maska. Poprzez pomnożenie jej przez funkcję straty wyzerowane są wszystkie wartości, które wpłynęłyby na korektę wag dla wartości pustych reprezentowanych jako padding.



Rysunek 37 Przykład działania maskowania

Do treningu użyty został optymalizator Adam (adaptive learning rate optimization) z domyślnym wskaźnikiem uczenia się ustawionym na 0,001. Spadek funkcji straty weryfikowany był co 10 epok. Trening trwał do momentu, w którym funkcja straty nie zmieniła się w znaczny sposób przez 30 epok lub gdy wartość funkcji straty dla zbioru walidacyjnego rosła względem zbioru treningowego co wskazywało na przetrenowanie modelu. Wykres spadku funkcji straty dla obu zbiorów został zamieszczony na rysunku 38. Z wykresów można odczytać, że wartość funkcji straty dla zbioru walidacyjnego nie odbiega znacząco od tej dla zbioru treningowego. Zjawisko to utrzymuje się również przez całość trwania treningu. Oznacza to, że nie dochodzi do tak zwanego przeuczenia, które jest nadmiernym dopasowaniem modelu do zbioru uczącego.

Przeuczenie zachodzi najczęściej w trzech przypadkach: długiego treningu, zbyt małej liczby przykładów, korzystaniu z modelu na tyle dużego, aby zapamiętać cały zbiór. Ponieważ wartości funkcji straty dla obu zbiorów były podobne, a danych było dużo, brak wystąpienia przeuczenia wskazuje na zbyt małą liczbę parametrów w modelu w stosunku do ilości danych w zbiorze.

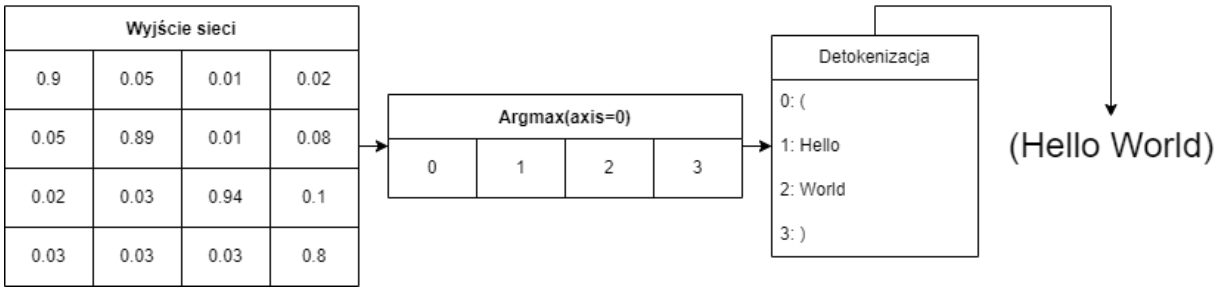


Rysunek 38 Wartość funkcji straty podczas treningu

6.2 Wyniki

Ostatnia część pracy przedstawia otrzymane wyniki. Dodatkowo omówione zostały sposoby weryfikacji ich jakości oraz zaprezentowano najważniejsze napotkane problemy. Na koniec przedstawione zostały wnioski oraz sugestie dotyczące możliwych uproszczeń, które mogłyby zostać wykonane w celu poprawy predykcji. Przed przystąpieniem do oceny wyników dane musiały zostać przekonwertowane z powrotem do postaci tekstu w sposób przedstawiony na rysunku 39. Wyjście sieci zwraca prawdopodobieństwa występowania wyrazów, które następnie trafiają do

funkcji argmax. Ma ona na celu wybrać wyłącznie największe wartości wzdłuż odpowiedniej osi, a następnie zwrócić ich indeks. W tym przypadku będą to tokeny z największym prawdopodobieństwem pojawienia się w danym miejscu. Następnie tokeny należy ponownie zamienić do domyślnej postaci za pomocą wcześniej zdefiniowanego słownika.



Rysunek 39 Przykład konwersji predykcji

Do problemu generacji kodu przystąpiono w sposób tożsamy z tłumaczeniem jednego języka naturalnego na drugi, jednak w przypadku weryfikacji jakości należy rozpatrzyć kilka kwestii. Po pierwsze istotnym czynnikiem jest fakt czy kod jest w stanie się skompilować, a drugim czy logika kodu odpowiada opisowi. Ponieważ twórcy zbioru CodeXGlue deklarowali, że fragmenty kodów przygotowane zostały w taki sposób, aby do uruchomienia nie wymagane były żadne moduły zewnętrzne, kod wygenerowany przez sieć powinien być w stanie się skompilować. Po wytrenowaniu sieci dostrzeżono pewien problem, którym była duża liczba nieodpowiednich znaków generowanych w dalszych częściach sekwencji, co w większości przypadków uniemożliwia kompilację. W celu lepszego zobrazowania wyników przykłady predykcji w języku C oraz porównawczo w językach Java i Python zostały przedstawione na rysunkach 40,41 oraz 42, na których zamieszczone zostały opisy problemów wraz z kodem źródłowym utworzonym do rozwiązania danego problemu, jak również kodem wygenerowanym przez sieć. Ponieważ weryfikacja pod względem możliwości kompilacji kodu okazała się niemożliwa, za kryterium jakości kodu przyjęte zostało podobieństwo do kodu referencyjnego dla danego problemu. Sprawdzono również wpływ długości opisu problemu na podobieństwo między predykcją sieci a kodem referencyjnym, lecz nie dostrzeżono różnic. Dodatkowo ze względu na podobieństwo wyników predykcji sieci na zbiorze CodeNet przedstawione zostały wyniki sieci trenowanej na kodach źródłowych bez zmienionych symboli leksykalnych. Na

omawianych rysunkach predykcje sieci zbliżone do kodu referencyjnego zostały oznaczone na zielono a w przeciwnym wypadku na czerwono.

<table>
<tr><th colspan="2">CodeNet (C)</th></tr>
<tr><td colspan="2">Opis problemu: find the intersection of two sets a = a0 a1 an-1 and b = b-1 b1 bm-1</td></tr>
<tr><td>Kod referencyjny: # include < stdio.h > int a [200010], b [200010] ; int main ( ) { int n, m ; scanf (" %d ", & n) ; for (int i = 0 ; i < n ; i ++) scanf (" %d ", a + i) ; scanf (" %d ", & m) ; for (int i = 0 ; i < m ; i ++) scanf (" %d ", b + i) ; a [n] = b [m] = 2e9 ; for (int i = 0, j = 0 ; i!= n || j!= m ;) { if (a [i] < b [j]) i ++ ; else if (a [i] > b [j]) j ++ ; else printf (" %d\n ", a [i]), i ++, j ++ ; } }</td><td>Predykcja: # include < stdio.h > int a [200010], b [200010] ; int main ( ) { int (int m ; scanf (" %d ", & n) ; for (int i = 0 ; i < n ; i ++) scanf (" %d ", a + i ; ; scanf (" %d ", & m) ; for (int i = 0 ; i < m ; i ++) scanf (" %d ", b + i) ; a [a] =] [[ ] = 2e9 for ((int = = 0 0 ; ; ;!=!=!=!=!=!=)))) [[[[[[[ ]]]) if if (printf " %d\n %d\n %d\n a a a [[[else else if [[[[[[[[[[[ ] printf } }</td></tr>
<tr><td colspan="2">Opis problemu: compute a + b</td></tr>
<tr><td>Kod referencyjny: main (a, b) { for (; ~ scanf (" %d%d ", & a, & b) ;) printf (" %d\n ", a + b) ; }</td><td>Predykcja: main (a, b) { for (; ~ scanf (" %d%d ", & a, & b) ;) printf (" %d\n ", a + b) ; }) ; } } } } } } }</td></tr>
<tr><td colspan="2">Opis problemu: write a program which reverseo a given string str</td></tr>
<tr><td>Kod referencyjny: main ( ) { puts (" xy63276440.3084middle ") ; }</td><td>Predykcja: main ( ) { puts (" xy63276440.3084middle ") ; } } " " " " ") ; } } * + +) } : " : } " : " }) ; } ; } " " } ") ; } } } } } } }</td></tr>
<tr><td colspan="2">Opis problemu: write a program which calculates the area and perimeter of a given rectangle</td></tr>
<tr><td>Kod referencyjny: a ; main (b) { scanf (" %d%d " , & a , & b) ; printf (" %d %d\\n " , a * b , a + a + b + b) ; }</td><td>Predykcja: a ; main (b) { scanf (" %d%d ", & a, & b) ; printf (" %d %d\n ", a * b, a + a + b + b) ; } " " } } ; }) ; })) }) } }) }) ; } }</td></tr>
<tr><td colspan="2">Opis problemu: you are given a 4-character string s consisting of uppercase english letters determine if s consists of exactly two kinds of characters which both appear twice in s</td></tr>
<tr><td>Kod referencyjny: # include < stdio.h > # include < stdlib.h > # include < string.h > int main ( ) { char a [5] ; scanf (" %s ", a) ; qsort (a, 4, sizeof (a [0]), strcmp) ; puts (a [0] == a [1] && a [2] == a [3] && a [0]!= a [2]? " yes " : " no ") ; return 0 ; }</td><td>Predykcja: # n < 9 [[, [[ ]] int r [invader_count] ; int (} } } a a a) { char a (a < 4 ; i ((a a a a r { r ; a ;) a if { a [a [{ a [a a a) t (t t t t t t t a a a a a a a a a a a a a a a a a a a a a r r r r r r r r r r r r r r r r r r r r r r r while r r r r r r r r r r while while while while while while while while while</td></tr>
<tr><td colspan="2">Opis problemu: there is an nansar train you are given an integer i find the value of j such that the following statement is true the i-th car from the front of the train is the j-th car from the back</td></tr>
<tr><td>Kod referencyjny: i ; main (n) { scanf (" %d%d ", & n, & i) ; printf (" %d ", n - i + 1) ; }</td><td>Predykcja: main (<, a s [ ]],, b b b b main n n, (scanf (" %d %d, & s) s for (i %d ; ; ; n n i))) scanf (" %d ", & a, & & += s) (2 n 1 ; n + 1) n - -))) } })) } } } } } }) } } } } } } } } " } } } }))))))))) } } } } } } } } } } } } } } } } } ; " "</td></tr>
<tr><td colspan="2">Opis problemu: write a program that reads the sales unit price and sales quantity and outputs the total sales price and the average sales quantity</td></tr>
<tr><td>Kod referencyjny: n, t, n, i ; main (u) { for (; ~ scanf (" hoge ", & u, & n)? t += u * n :! printf (" %d\n%d\n ", t, nd) ; i ++) n -= ~ n ; }</td><td>Predykcja: c ; 26] ; main (c) { for (; ~ (c ", i, & c)) if c))) (++ ++) ((800) ; (800 800 b ; }) 0 ; 97 ; 0 ; ((0 = 0 ; c : 26 ; ; ++) printf (" %c ") ") ; exit (} ; } ; } ; exit ((; ; ((( ) ; ((( ) (((( ) (((( ) (</td></tr>
<tr><td colspan="2">Opis problemu: you are given three integers a b and c determine whether c is not less than a and not greater than b</td></tr>
<tr><td>Kod referencyjny: main (a, b, c) { scanf (" %d%d%d ", & a, & b, & c) ; puts (c < a | b < c? " no " : " yes ") ; }</td><td>Predykcja: typedef define a ll b ll a, b, c, & ; g g c & ; ll ll %)) if (% y) g ((y (% (2, (% (! " (") " " " ", a, & a, & a) ;) -- (-- -- a) a) a) a - c - c - c - c * 2 - c * 2) c * d) d) d) d) d))))) }</td></tr>
</table>

Rysunek 40 Przykład predykcji oraz danych testowych ze zbioru CodeNet

CONCODE (Java)	
Opis problemu: returns true if there's another tlp mode to try.	
Kod referencyjny: boolean function ( ) { return nexttlsmode!= tls_mode_null ; }	**Predykcja:** boolean function ( ) { return nexttlsmode!= tls_mode_null ;}) ; }. }))!) ; } ( )))) ; } } } ; } ; } } }) ; }) ; } ; } ; } } ; } ;
Opis problemu: returns the scalar used for determining the rotation y value.	
Kod referencyjny: double function ( ) { return this. rotatei ; }	**Predykcja:** double function ( ) { return this. rotatei ; } ; } loc0 = integer. integer = ; } }. ;) ; }.names3 /) ; } }
Opis problemu: returns true if there's another tlp mode to try.	
Kod referencyjny: boolean function ( ) { return nexttlsmode!= tls_mode_null ; }	**Predykcja:** boolean function ( ) { return nexttlsmode!= tls_mode_null ; }) ; }. }))!) ; } ( )))) ; } } } ; } ; } } }) ; }) ; } ; } ; } } ; } ;
Opis problemu: auto generated method comment.	
Kod referencyjny: void function (sessionfactory arg0) { this. sessionfactory = arg0 ; }	**Predykcja:** void function (sessionfactory arg0) { this. sessionfactory = arg0 ; } ; ;n (n ; } ; ; ;.. (( ) ; ; ; ; ; ;...))) ; ;... ; ; ; ; ;....... ; ; } } } } } } } } } } } } } } } } } }
Opis problemu: calculates the order price.	
Kod referencyjny: bigdecimal function (order arg0) { double loc0 = 0 ; if (arg0. getgoodsinordern ( )!= null) for (goodsinorder loc1 : arg0. getgoodsinordern ( )) { loc0 += (loc1. getprice ( ). doublevalue ( ) * loc1. getquantity ( )) ; } return new bigdecimal (loc0) ; }	**Predykcja:** void function (string arg0, arg0, { try {sh. dialog (dialog ; onc onc (( ).... ("srini_string".....)...)))))))))))))))) ;
Opis problemu: play this streamsound..	
Kod referencyjny: void function ( ) { this. play (1.0) ; }	**Predykcja:** void function (file { system (lockhandler ( ) ; listeners) { listeners. length. length)).. lastindexof. lastindexof))) ; } } loc0)) ; ; ; ; ; ; ; ; } } } }
Opis problemu: returns a list of ownerp with the given owner code.	
Kod referencyjny: list < ownerpatch > function (string arg0) { return this. repository. findbyownercodeandpatchtype (arg0, documenttype. owner) ; }	**Predykcja:** int [[ ] [function arg0 (false) supportedfunction. objects objects } } } } loc0. loc0))) return } } return loc0 loc0 loc0 loc0 loc0 ; ; ; ; ; } } }
Opis problemu: function that converts a solution between matrix representation and array representation.	
Kod referencyjny: int [ ] function (int [ ] [ ] arg0) { int [ ] loc0 = new int [nb + nz - 1] ; for (int loc1 = 0, loc2 = 0 ; loc1 < nz ; loc1 ++) { for (int loc3 = 1 ; loc3 < arg0 [loc1]. length ; loc3 ++, loc2 ++) loc0 [loc2] = arg0 [loc1] [loc3] - 1 ; if (loc1!= nz - 1) { loc0 [loc2] = nb + loc1 ; loc2 ++ ; } } return loc0 ; }	**Predykcja:** void function (string arg0) { } =) new remove ( ) ; 0 ((( ) }

Rysunek 41 Przykład predykcji oraz danych testowych ze zbioru CONCODE

CodeSearchNet (Python)	
Opis problemu: setter for the error handler function.	
Kod referencyjny: def error _ handler (self, handler) : if not self. opened ( ) : handler = handler or util. noop self. _ error _ handler = enums. jlinkfunctions. log _ prototype (handler) self. _ dll. jlinkarm _ seterrorouthandler (self. _ error _ handler)	**Predykcja:** def warning _ handler (self, handler) : if not self. opened ( ) : handler = handler. handleril.il. self. self. self. prototype. prototype. prototype. prototype. prototype. prototype. prototype.arm.arm.arm.arm.arm.arm.arm.arm.arm.arm. self. self
Opis problemu: search the manga series list by name case - insensitive.	
Kod referencyjny: def search _ manga _ series (self, query _ string) : result = self. _ manga _ api. list _ series ( ) return [series for series in result if series ['locale'] ['enus'] ['name']. lower ( ). startswith (query _ string. lower ( ))]	**Predykcja:** def get (resources _ name) : filter = = = self. : from = self. get _ get _ per _ _ per) per per per per. per. per.... _ _ _))))
Opis problemu: set fields used to group query results.	
Kod referencyjny: def distinct _ on (self, value) : if isinstance (value, str) : value = [value] self. _ distinct _ on [:] = value	**Predykcja:** def distinct _ to (self, value) : if isinstance (value) : value ( ) return value (value (value (value
Opis problemu: cleans up the context and the hooks.	
Kod referencyjny: def deallocate (self) : self. logger. debug (" deallocating environment. ") self. _ deallocate ( ) self. logger. debug (" environment successfully deallocated. ")	**Predykcja:** def deallocate (self) : self. logger _ loglog ("cating.cating.. self. self......... " " " " " " " " " " " " " "....
Opis problemu: gets the long algebraic notation of the given move in the context of the current position.	
Kod referencyjny: def lan (self, move : move) - > str : return self. _ algebraic (move, long = true)	**Predykcja:** def visit _ table _ for _ x, :) : x = x, = 1) : if = self. get _ _ (( ) _) _) _) _) _) _ _) _)))))))))
Opis problemu: convenience function for packing.	
Kod referencyjny: def pack (self) : block = bytearray (self. size) self. pack _ into (block) return block	**Predykcja:** def save (self, :,,, : usd :) : : : : : : :) :) '...........).)
Opis problemu: unloads all modules that weren t loaded when save _ modules was called.	
Kod referencyjny: def restore (self) : sys = set (self. _ sys _ modules. keys ( )) for mod _ name in sys. difference (self. _ saved _ modules) : del self. _ sys _ modules [mod _ name]	**Predykcja:** def play _ cache (funn _ _able _ none, : _ none) self. self _ key _
Opis problemu: create a thread.	
Kod referencyjny: def _ spawn _ thread (self, target) : t = thread (target = target) t. daemon = true t. start ( ) return t	**Predykcja:** def create (self (self,,,,,,, :) : :.... _. _.......... _))))))))))

Rysunek 42 Przykład predykcji oraz danych testowych ze zbioru CodeSearchNet

Podobieństwo w przedstawionych przykładach zostało zmierzone za pomocą algorytmu do ewaluacji jakości tłumaczenia automatycznego z jednego języka naturalnego na inny (ang. BLEU). Początkowo stosowana była jedno segmentowa wersja algorytmu, reprezentowana za pomocą wzoru 6 gdzie porównywana jest liczba pasujących słów predykcji względem zdania referencyjnego, w zależności od liczby słów w predykcji. Wartość podobieństwa mieści się w zakresie od 0 do 1 [37].

$$Podobieństwo = \frac{liczba\ \ pasujących\ słów}{liczba\ słów\ w\ predykcji} (6)$$

Wzór ten uwzględnia powtarzające się słowa w predykcji oraz dowolne zmiany w ich kolejności dlatego BLEU rozszerzono o podział tekstu na segmenty najczęściej 4 wyrazowe w przypadku predykcji, oraz zdania referencyjnego. Porównywanie niewielkich fragmentów tekstu wpływa bezpośrednio na zachowanie kolejności słów. Podobieństwo jest obliczone poprzez podzielenie liczby poprawnie przewidzianych segmentów przez całkowitą liczbę segmentów. Dodatkowo w celu zapobiegania powtarzającym się wyrazom w predykcji ich wartość zostaje ograniczona do liczby powtórzeń danego słowa w zdaniu referencyjnym. Wspomniane modyfikacje zostały uwzględnione we wzorze 7, gdzie Clip jest ograniczeniem liczby słów w segmencie względem liczby słów w zdaniu referencyjnym [37].

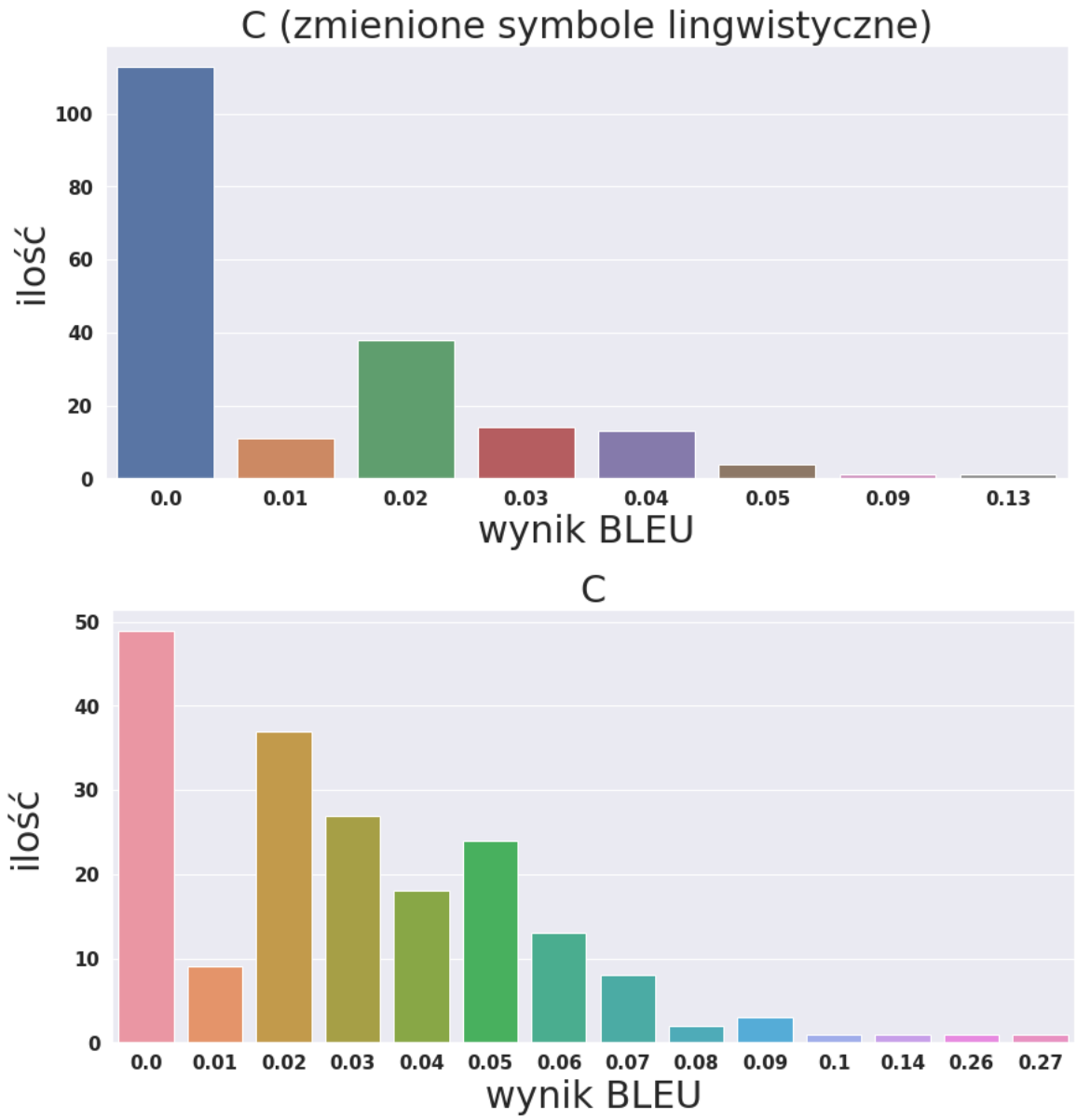
$$Podobieństwo = \frac{Clip(liczba\ poprawnie\ przewidzianych\ segmentów)}{liczba\ \ wszystkich\ segmentów} (7)$$

Na rysunku 43 przedstawiona została liczba danych treningowych, testowych oraz walidacyjnych wraz z uśrednioną wartością wskaźnika podobieństwa BLEU dla każdego ze zbiorów. Dodatkowo na rysunkach 44 oraz 45 przedstawiono rozkład wartości wspomnianego wskaźnika w zależności od zbioru. Ze względu na większą liczbę danych w zbiorach CONCODE (Java) oraz CodeSearchNet (Python) wartości zostały przedstawione dla odpowiednich zakresów.

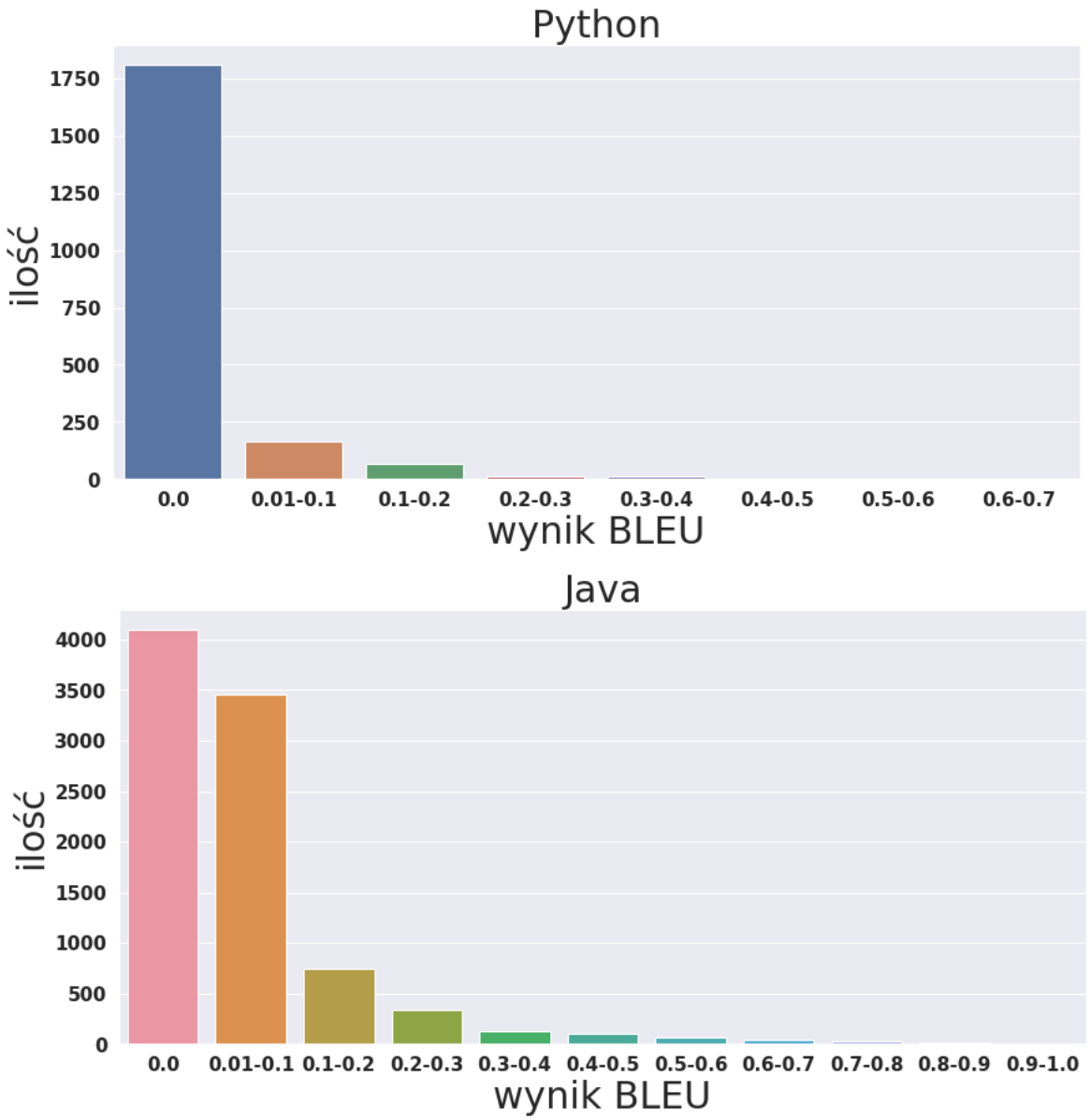
C	C (zmienione symbole)
Dane treningowe: 1556	Dane treningowe: 1557
Dane testowe: 194	Dane testowe: 195
Dane walidacyjne: 195	Dane walidacyjne: 195
Uśredniona wartość wskaźnika podobieństwa BLUE: 0.03	Uśredniona wartość wskaźnika podobieństwa BLUE: 0.01

Python	Java
Dane treningowe: 74876	Dane treningowe: 60410
Dane testowe: 1970	Dane testowe: 7551
Dane walidacyjne: 1970	Dane walidacyjne: 7552
Uśredniona wartość wskaźnika podobieństwa BLUE: 0.01	Uśredniona wartość wskaźnika podobieństwa BLUE: 0.06

Rysunek 43 Tabela liczby danych wraz oraz uśrednionej wartości wskaźnika podobieństwa dla każdego ze zbiorów



Rysunek 44 Rozkład wartości wskaźnika BLEU dla zbioru CodeNet



Rysunek 45 Rozkład wartości wskaźnika BLEU dla zbioru CONCODE oraz CodeSearchNet

7 Dyskusja

Pomimo pewnych odstępstw od kodu referencyjnego, w przypadku wszystkich 3 modeli generujących kod dla osobnych języków programowania, można znaleźć dużą liczbę podobieństw. Oceniając budowę generowanego kodu, zawiera ona elementy charakterystyczne dla gramatyki nauczanego języka, jak również znajduję korelację między pewnymi nazwami funkcji lub zmiennych a opisem w języku naturalnym. Podczas treningu widoczne było zjawisko generowania zadań o zbliżonej treści, nawet w przypadku pojawienia się kilku słów znacząco zmieniających kontekst. Predykcje najbardziej zbliżone do kodu referencyjnego pojawiły się w przypadku modelu trenowanego na zbiorze CONCODE. W modelu trenowanym na zbiorze CodeSearchNet wyniki były słabsze niż w przypadku pozostałych modeli. Różnica w jakości predykcji na wspomnianych zbiorach może wynikać z różnicy długości kodów źródłowych w danych treningowych, które dla tych zbiorów były największe. Może to wskazywać na zbyt ograniczoną architekturę modelu, która nie jest w stanie przewidywać długich sekwencji na wyjściu. Kolejnym czynnikiem, który mógł mieć wpływ na uczenie była jakość danych uczących. W przypadku zarówno języka naturalnego, jak i kodu źródłowego pojawia się duża liczba zbędnych znaków, które ciężko wyeliminować. Ponieważ opis w języku naturalnym jest połączony z kodem źródłowym pewną logiką, ciężko jest modyfikować zawartość jednego z nich, nie wpływając bezpośrednio na drugie. Mogło się to przełożyć w szczególności na zbiór CodeNet, gdzie część poleceń została przetłumaczona z innych języków niż angielski, co mogło zaburzyć logikę tekstu. Dodatkowo duża liczba danych przekłada się bezpośrednio na ilość wymaganych zasobów oraz czasu do treningu sieci. W celu poprawy wyników możliwe jest użycie architektury sieci typu transformer, rozbudowa sieci o kolejne warstwy oraz dobranie problemów programistycznych zbliżonych do siebie pod względem treści.

8 Podsumowanie

Celem niniejszej pracy było wykazanie, że generacja kodu źródłowego w języku C przy użyciu sieci rekurencyjnych jest możliwa. Jest to szczególnie ważne zagadnienie ze względu na szybko rosnące zainteresowanie narzędziami wspomagającymi programistów podczas pracy. W teoretycznej części pracy omówiono istniejące rozwiązania oraz biblioteki użyte do stworzenia modelu, przedstawiono działanie sieci rekurencyjnych oraz wyjaśniono sposób przetwarzania wstępnego danych. W części praktycznej ukazana została budowa modelu oraz jego trening. Podczas treningu modelu napotkane zostały problemy związane głównie z przetwarzaniem i tokenizacją danych, które przełożyły się bezpośrednio na wyniki. Wspomniane problemy udało się ostatecznie rozwiązać i udowodnić, że generacja kodu na podstawie opisów problemów programistycznych w języku C, przy użyciu sieci rekurencyjnych jest możliwa.

9 Bibliografia

1 Ryszard Tadeusiewicz, Maciej Szaleniec: Leksykon sieci neuronowych

https://www.researchgate.net/publication/294578763_LEKSYKON_SIECI_NEURONOWYCH_Lexicon_on_Neural_Networks (dostęp : 01.08.2022)

2 Utkarsh Ankit: Transformer Neural Network: Step-By-Step Breakdown of the Beast

https://towardsdatascience.com/transformer-neural-network-step-by-step-breakdown-of-the-beast-b3e096dc857f (dostęp : 01.08.2022)

3 Kashyap Kathrani: All about Embeddings

https://medium.com/@kashyapkathrani/all-about-embeddings-829c8ff0bf5b

(dostęp : 01.08.2022)

4 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin: Attention Is All You Need

https://arxiv.org/abs/1706.03762 (dostęp : 01.08.2022)

5 Mehreen Saeed: A Gentle Introduction to Positional Encoding In Transformer Models, Part 1

https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/ (dostęp : 01.08.2022)

6 Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei: Language Models are Few-Shot Learners

https://arxiv.org/pdf/2005.14165.pdf (dostęp : 01.08.2022)

7 Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova:

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

https://arxiv.org/abs/1810.04805 (dostęp : 01.08.2022)

8 Sciforce: What is GPT-3, How Does It Work, and What Does It Actually Do?

https://medium.com/sciforce/what-is-gpt-3-how-does-it-work-and-what-does-it-actually-do-9f721d69e5c1 (dostęp : 01.08.2022)

9 NLPiation: What are the differences in Pre-Trained Transformer-base models like BERT, DistilBERT, XLNet, GPT, XLNet, …

https://medium.com/mlearning-ai/what-are-the-differences-in-pre-trained-transformer-base-models-like-bert-distilbert-xlnet-gpt-4b3ea30ef3d7 (dostęp : 01.08.2022)

10 OpenAI Codex

https://openai.com/blog/openai-codex/ (dostęp : 01.08.2022)

11 Siwei Causevic: Self-supervised Transformer Language Models — BERT, GPT3, MUM and PaML

https://towardsdatascience.com/self-supervised-transformer-models-bert-gpt3-mum-and-paml-2b5e29ea0c26 (dostęp : 01.08.2022)

12 Rohan Jagtap: T5: Text-To-Text Transfer Transformer

https://towardsdatascience.com/t5-text-to-text-transfer-transformer-643f89e8905e (dostęp : 01.08.2022)

13 Salesforce: CodeT5

https://github.com/salesforce/CodeT5

14 Sherry J. Basler: CMU Researchers Open Source 'PolyCoder': A Machine Learning-Based Code Generator with 2.7 Billion Parameters

https://learningcomputation.com/cmu-researchers-open-source-polycoder-a-machine-learning-based-code-generator-with-2-7-billion-parameters (dostęp : 01.08.2022)

15 Frank F. Xu, Uri Alon, Graham Neubig, Vincent J. Hellendoorn: Systematyczna ocena dużych modeli językowych kodu

https://arxiv.org/pdf/2202.13169.pdf (dostęp : 01.08.2022)

16 Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, Yann N. Dauphin: Convolutional Sequence to Sequence Learning

https://arxiv.org/abs/1705.03122 (dostęp : 01.08.2022)

17 Afshine Amidi, Shervine Amidi: Recurrent Neural Networks

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks (dostęp : 01.08.2022)

18 Embeddings: Translating to a Lower-Dimensional Space

https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space (dostęp : 01.08.2022)

19 Gaurav Singhal: Introduction to LSTM Units in RNN

https://www.pluralsight.com/guides/introduction-to-lstm-units-in-rnn

(dostęp : 01.08.2022)

20 Samhita Alla: Advanced Recurrent Neural Networks: Bidirectional RNNs

https://blog.paperspace.com/bidirectional-rnn-keras/ (dostęp : 01.08.2022)

21 Jason Brownlee: A Gentle Introduction to LSTM Autoencoders

https://machinelearningmastery.com/lstm-autoencoders/ (dostęp : 01.08.2022)

22 Amy Unruh: What is the TensorFlow machine intelligence platform?

https://opensource.com/article/17/11/intro-tensorflow (dostęp : 01.08.2022)

23 Trenowanie modeli Keras na dużą skalę przy użyciu usługi Azure Machine Learning

https://docs.microsoft.com/pl-pl/azure/machine-learning/how-to-train-keras

(dostęp : 01.08.2022)

24 Keras dokumentacja

https://keras.io/#why-this-name-keras (dostęp : 01.08.2022)

25 Mateusz Kuba: Scikit-learn biblioteka do uczenia maszynowego w Pythonie

https://boringowl.io/tag/scikit-learn/ (dostęp : 01.08.2022)

26 Karan Bhanot: Google Colab - pythonowy obszar roboczy w chmurze

https://bulldogjob.pl/readme/google-colab-pythonowy-obszar-roboczy-w-chmurze (dostęp : 01.08.2022)

27 CodexGLUE dokumentacja

https://microsoft.github.io/CodeXGLUE/ (dostęp : 08.12.2022)

28 James Briggs: How to Build a WordPiece Tokenizer For BERT

https://towardsdatascience.com/how-to-build-a-wordpiece-tokenizer-for-bert-f505d97dddbb (dostęp : 08.12.2022)

29 Chetna Khanna: WordPiece: Subword-based tokenization algorithm

https://towardsdatascience.com/wordpiece-subword-based-tokenization-algorithm-1fbd14394ed7 (dostęp : 08.12.2022)

30 Hugging Face: WordPiece tokenization

https://huggingface.co/course/chapter6/6?fw=pt (dostęp : 08.12.2022)

31 Kenneth Leung: How to Easily Draw Neural Network Architecture Diagrams

https://towardsdatascience.com/how-to-easily-draw-neural-network-architecture-diagrams-a6b6138ed875 (dostęp : 08.12.2022)

32 Dirko Coetse: Masked Bidirectional LSTMS with keras

https://dirko.github.io/Bidirectional-LSTMs-with-Keras/ (dostęp : 08.12.2022)

33 Rizky Luthfianto: The difference between Dense and TimeDistributedDense of Keras

https://datascience.stackexchange.com/questions/10836/the-difference-between-dense-and-timedistributeddense-of-keras (dostęp : 08.12.2022)

34 Amar Budhiraja: Dropout in (Deep) Machine learning

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

35 IBM Research: Project CodeNet

https://developer.ibm.com/exchanges/data/all/project-codenet/ (dostęp : 08.12.2022)

(dostęp : 08.12.2022)

36 Hugging Face: Summary of the tokenizers

https://huggingface.co/docs/transformers/tokenizer_summary (dostęp : 08.12.2022)

37 Foundations of NLP Explained — Bleu Score and WER Metrics

https://towardsdatascience.com/foundations-of-nlp-explained-bleu-score-and-wer-metrics-1a5ba06d812b (dostęp : 26.01.2023)

38 Inez Okulska: Sieci neuronowe typu GAN i GTP-2, słowa zużyte i kreatywność, czyli literacki secon-hand, „Forum Poetyki", nr 18/2019

39 Stanisław Osowsk:i Sieci - neuronowe w ujęciu algorytmicznym

40 Tadeusiewicz R: Sieci neuronowe, Akademicka Oficyna Wydawnicza, Warszawa 1993 (wyd. I i II tego samego roku)

41 Korbicz J., Obuchowicz A., Uciński D: Sztuczne sieci neuronowe: podstawy i zastosowania, Akademicka Oficyna Wydawnicza, Warszawa, 1994.

42 Herz J., Krogh A., Palmer R.G.: Wstęp do teorii obliczeń neuronowych, Wydawnictwa Naukowo-Techniczne, Warszawa 1995.

43 Jędruch W.: Sztuczne sieci neuronowe, Wydawnictwo Naukowe PWN, Warszawa 1996.

44 Żurada J., Barski M., Jędruch W.: Sztuczne sieci neuronowe. Podstawy teorii i zastosowania, Wydawnictwo Naukowe PWN, Warszawa1996.

45 Master T.: Sieci neuronowe w praktyce, Wydawnictwa Naukowo-Techniczne, Warszawa 1996.

46 Rutkowska D., Piliński M., Rutkowski L.: Sieci neuronowe, algorytmy genetyczne i systemy rozmyte, Wydawnictwo Naukowe PWN, Warszawa 1997.

47 Denis Rothman: Transformers for natural language processing

48 Nils Reimers, Iryna Gurevych: Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks

https://arxiv.org/abs/1908.10084 (dostęp : 27.01.2022)

49 Minh-Thang Luong, Hieu Pham, Christopher D. Manning: Effective Approaches to Attention-based Neural Machine Translation

https://arxiv.org/abs/1508.04025 (dostęp : 27.01.2022)

50 Jianpeng Cheng, Li Dong, Mirella Lapata: Long Short-Term Memory-Networks for Machine Reading

https://arxiv.org/abs/1601.06733 (dostęp : 27.01.2022)

51 Alex Graves: Generating Sequences With Recurrent Neural Networks

https://arxiv.org/abs/1308.0850 (dostęp : 27.01.2022)

52 Łukasz Kaiser, Ilya Sutskever: Neural GPUs Learn Algorithms

https://arxiv.org/abs/1511.08228 (dostęp : 27.01.2022)

53 Łukasz Wordliczek: Wykorzystanie sztucznych sieci neuronowych

54 Juri Fedjaev: Decoding EEG Brain Signals using Recurrent Neural Networks

55 Sanjay Kumar Boddhu: Towards Building a Neuromorphic Computer: A Reconfigurable Hardware Continuous Time Recurrent Neural Network

56 Gerardus Blokdyk: Recurrent neural network

57 Seyedali Mirjalili, Hossam Faris, Ibrahim Aljarah: Evolutionary Machine Learning Techniques

58 Yoav Goldberg, Graeme Hirst: Neural Network Methods in Natural Language Processing

59 Joish Bosco, Fateh Khan: Stock Market Prediction and Efficiency Analysis using Recurrent Neural Network

60 Tarik Rashid: Recurrent Neural Network Model

61 Filippo Maria Bianchi, Enrico Maiorino, Michael C. Kampffmeyer, Antonello Rizzi: Recurrent Neural Networks for Short-Term Load Forecasting

62 Conrad Tiflin: LSTM Recurrent Neural Networks for Signature Verification

10 Spis Rysunków

11 Wykaz Symboli

RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
GPT	Generative Pre-trained Transformer
BERT	Bidirectional Encoder Representation From Transformer
CNN	Convolutional Neural Network
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
BLEU	Bilingual Evaluation Understudy

12 Załączniki

W skład pracy wchodzi kopia tej pracy jak również wytrenowane modele wraz z danymi uczącymi po podziale na zbiory treningowe, testowe oraz walidacyjne.

10.1 Notatnik i Model

W tym folderze znajduje się notatniki w których zawarty został proces uczenia sieci jak również zapisy modeli które zostały nauczone na danych treningowych. Dodatkowo plik zawiera notatniki z kodem do testowania modeli.

10.2 Dane

Katalog z danymi zawiera plik c_train, py_train oraz java_train gdzie znajdują się wszystkie dane do uczenia, walidacji i testowania modelu. Wszystkie zawarte tam dane są już po tokenizacji oraz filtrowaniu długich sekwencji.

10.3 Wymagane oprogramowanie

Wykorzystane biblioteki do uczenia wraz z użytą wersją znajdują się w pliku tekstowym lib_info.txt.