Projekt kompetencyjny

Sieci neuronowe Seq2Seq

	Autor	**Album**
	Kateryna Tsarova	226451

	Prowadzący
	Piotr Duch, dr inż.

Ocena prowadzącego:

Łódź, 08.09.2021 r.

SPIS TREŚCI

1. WSTĘP

1.1 Cel projektu

Celem mojego projektu jest napisanie sieci neuronowej, która będzie w stanie znaleźć i poprawić błędy programistyczne w programie użytkownika, napisanej w języku C.

1.2 Czym są sieci neuronowe Seq2Seq

Modele zamieniające pewną sekwencję wejściową w jednej domenie na sekwencję wyjściową w innej domenie są nazywane SEQ2SEQ. Najprostszy model SEQ2SEQ składa się z 2 komponentów – enkodera i dekodera RNN.

Schemat działania SEQ2SEQ jest następujący:

- Zadaniem enkodera wejściowego (złożonego z warstwy powtarzających się jednostek, np. RNN) jest utworzenie wewnętrznej reprezentacji wektorowej tekstu wejściowego. Owy stan wewnętrzny enkodera będzie służył do warunkowania dekodera. Każda jednostka RNN otrzymuje informacje o pojedynczym elemencie, przetwarza informacje i przekazuje je dalej.

- Dekoder (zbiór powtarzających się jednostek, np. RNN, z których każda przewiduje wynik y_t w kroku czasowym t), otrzymawszy ową sekwencję, generuje sekwencję wyjściową. Jest w ten sposób warunkowany kontekstem na wyjściu enkodera. Dekoder jest uczony przewidywania następnych znaków sekwencji docelowej, biorąc pod uwagę poprzednie znaki sekwencji i generuje słowa, dopóki nie zostanie utworzony specjalny znacznik końca zdania.

Istnieje wiele modeli Seq2Seq – np. powstałych z użyciem sieci RNN, LSTM, GRU lub konwolucyjnych 1D. W bardziej skomplikowanych wykorzystuje się np. większą liczbę wektorów do kodowania sekwencji wejściowej, co pozwala uchwycić więcej istotnych w przetwarzaniu tekstu informacji. [1].

1.3 Poznane techniki i narzędzia

Do napisania sieci neuronowej użyłam języka Python, przy napisaniu korzystałam z bibliotek Tensorflow, Keras.

1.4 Przebieg pracy

Projekt był podzielony na dwie części:

1) zaimplementowanie rekurencyjnej sieci neuronowej do generowania kodu w języku C,

2) zaimplementowanie sieci neuronowej Seq2Seq do poprawienia błędów syntaktycznych w programie, napisanej w języku C.

2. DZIAŁANIE SIECI NEURONOWYCH

2.1 Przygotowanie danych wejściowych

Do wytrenowania sieci neuronowych była wykorzystana baza danych DeepFix, która składa się z 53 478 kodów w języku C [2].

Programy, podawane na wejście sieci, są traktowane jako sekwencje tokenów. Proces tokenizacji wygląda następująco:

1) Z wczytanego zestawu danych jest tworzony słownik, obejmujący takie tokeny w języku C jak:
- typy (np. int, float, char),
- słowa kluczowe (np. break, case, const, continue),
- znaki specjalne (np. średnik, przecinek, dwukropek),
- dyrektywy (np. #include, #define),
- biblioteki (np. <stdio.h>, <stdlib.h>),
- funkcje biblioteki standardowej (np. scanf, printf).

2) Także do słownika były dodane specjalne tokeny, reprezentujące początek i koniec sekwencji:
- <sof>,
- <eof>.

3) Dokładna wartość literałów nie jest znacząca dla uczenia sieci, więc literały są mapowane jako specjalne tokeny w oparciu o typ:
- _<number>_ (dla liczb),
- _<string>_ (dla ciągów znaków).

4) Nazwy zmiennych i funkcji nie są znaczące dla uczenia sieci, więc w słowniku jest definiowana pula nazw o stałym rozmiarze, która jest wystarczająco duża, aby umożliwić mapowanie dla dowolnego programu w zestawie danych. Dla każdego programu podczas tokenizacji jest tworzona oddzielna mapa kodowania, mapująca każdy odrębny identyfikator (nazwę zmiennej lub funkcji) w programie na unikalną nazwę, wziątą z puli nazw. Przykład definiowania puli nazw i mapowania zmiennych dla dwóch różnych programów:

Pula nazw: 1@, 2@, 3@, 4@, 5@

Pierwszy program (do tokenizacji): int num = 5;

Pierwszy program (po tokenizacji): int @1 = <number>;

Drugi program (do tokenizacji): int a, b, c;

Drugi program (po tokenizacji): int @1, @2, @3;

5) Do słownika był dodany specjalny token (~), reprezentujący nową linię w kodzie.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i,siz1,siz2;
    scanf("%d",siz1);
    scanf("%d",siz2);
    char ch1[siz1];
    char ch2[siz2];
    for(i=0;i<siz1;i++)
        ch1[i]= getchar();
    for(i=0;i<siz1;i++)
        putchar(ch1[i]);
    return 0;
}
```

Rys. 1: kod w języku C do tokenizacji

```
<sof> ~ #include <stdio.h> ~ #include <stdlib.h> ~ int main ( ) { ~ int 1@ , 2@ , 3@ ;
~ scanf ( _<string>_ , 2@ ) ; ~ scanf ( _<string>_ , 3@ ) ; ~ char 4@ [ 2@ ] ;
~ char 5@ [ 3@ ] ; ~ for ( 1@ = _<number>_#  ; 1@ < 2@ ; 1@ ++ ) ~ 4@ [ 1@ ] = getchar ( ) ;
~ for ( 1@ = _<number>_#  ; 1@ < 2@ ; 1@ ++ ) ~ 6@ ( 4@ [ 1@ ] ) ; ~ return _<number>_#  ;
~ } ~ <eof>
```

Rys. 2: kod w języku C po tokenizacji

```
<sof>
~ #include <stdio.h>
~ #include <stdlib.h>
~ int main ( ) {
~       int 1@ , 2@ , 3@ ;
~       scanf ( _<string>_ , 2@ ) ;
~       scanf ( _<string>_ , 3@ ) ;
~       char 4@ [ 2@ ] ;
~       char 5@ [ 3@ ] ;
~       for ( 1@ = _<number>_#  ; 1@ < 2@ ; 1@ ++ )
~           4@ [ 1@ ] = getchar ( ) ;
~       for ( 1@ = _<number>_#  ; 1@ < 2@ ; 1@ ++ )
~           6@ ( 4@ [ 1@ ] ) ; ~ return _<number>_#  ;
~ }
~ <eof>
```

Rys. 3: kod w języku C po tokenizacji
I formatowaniu

2.2 Sieć do generowania kodu

2.2.1 Model LSTM

Model rekurencyjnej sieci neuronowej był stworzony w *Tensorflow* (open-source'owy framework stworzony przez Google'a do obliczeń numerycznych). Oferuje on zestaw narzędzi służących do projektowania, trenowania oraz uczenia sieci neuronowych [3].

Każda warstwa modelu ma dokładnie jeden tensor wejściowy i jeden tensor wyjściowy - jest to model sekwencyjny (rys. 4). Sieć neuronowa składa się z kilku rodzajów warstw, oferowanych przez *Tensorflow*:

- *LSTM* – długa pamięć krótkotrwała,
- *Bidirectional LSTM* – dwukierunkowy LSTM,
- *Dense* – warstwa, dla której wszystkie jednostki poprzedniej warstwy są połączone ze wszystkimi w następnej,
- *Dropout* – przepuszczających tylko fragment danych (aby zapobiec przeuczeniu sieci).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_sequences (InputLayer) [(None, 15, 204)]         0
_________________________________________________________________
bidirectional (Bidirectional (None, 15, 512)           944128
_________________________________________________________________
dropout_bidirectional_lstm ( (None, 15, 512)           0
_________________________________________________________________
lstm (LSTM)                  (None, 64)                147712
_________________________________________________________________
drop_out_lstm (Dropout)      (None, 64)                0
_________________________________________________________________
first_dense (Dense)          (None, 3060)              198900
_________________________________________________________________
drop_out_first_dense (Dropou (None, 3060)              0
_________________________________________________________________
second_dense (Dense)         (None, 1020)              3122220
_________________________________________________________________
drop_out_second_dense (Dropo (None, 1020)              0
_________________________________________________________________
last_dense (Dense)           (None, 204)               208284
_________________________________________________________________
activation (Activation)      (None, 204)               0
=================================================================
Total params: 4,621,244
Trainable params: 4,621,244
Non-trainable params: 0
```

Rys. 5: Architektura sieci neuronowej

Jako dane wejściowe warstwa wejściowa przyjmuje sekwencję tokenów w postaci liczbowej o rozmiarze *seq_length* (dla tej sieci neuronowej długość sekwencji jest równa 15). Na wyjściu sieć generuje wektor one-hot.

Lizba neuronów w dwóch warstwach LSTM jest równa odpowiednio 512 i 64, liczba neuronów w trzech warstwach *Dense* - 15 * len(słownik), 5 * len(słownik), len(słownik). Parametr warstwy *Dropout* jest równy 0.1. Warstwa wyjściowa korzysta z funkcji optymalizacji *softmax*, dzięki czemu można otrzymać na wyjściu tablicę składającą się z len(słownik) wartości sumujących się do 1.

Aby ocenić wydajność sieci neuronowej, do wytrenerowania modelu była wykorzystana kategoryczna entropia krzyżowa (ang. *categorical crossentropy*). Był także wykorzystany algorytm optymalizacji ADAM (ang. *adaptive moment estimation*).

2.2.2 Wyniki nauczania

Uczenie modelu było dokonywane dwuetapowo. Do trenowania modelu były wybrane takie parametry:

1 etap:
- *epoch* – 15
- *batch_size* – 128

2 etap:
- *epoch* – 5
- *batch_size* – 2048

Epoch odpowiada za liczbę epok użytych do trenowania modelu – jedna epoka oznacza przejście całego zbioru przez sieć. *Batch_size* odpowiada za zdefiniowanie ile rekordów (obserwacji) przechodzi na raz podczas pojedynczego przebiegu zanim nastąpi pierwsza aktualizacja wag parametrów.

Końcowa dokładność sieci neuronowej jest równa 68.4%. Czas treningu - 6.5 godz.

2.2.3 Wyniki działania sieci

Wyniki działania sieci można zaobserwować na rys. 6, 7 i 8.

```
<sof> ~ #include <stdio.h> ~ #include <stdlib.h> ~ int main ( ) {
~ int 1@ , 2@ , 3@ , 4@ , 5@ = _<number>_# ; ~ scanf ( _<string>_ , & 2@ ) ;
~ int 6@ [ 2@ ] ; ~ for ( 3@ = _<number>_#  ; 3@ < = 2@ ; 3@ ++ )
~ { scanf ( _<string>_ , & 4@ ) ; ~ if ( 2@ == 3@ ) ~ {
~ if ( 4@ ! = _<number>_#  ) ~ { ~ 4@ = 3@ / _<number>_#  ;
~ printf ( _<string>_ , 5@ ) ; ~ } ~ } ~ return _<number>_#  ;
~ } <eof>
```

Rys. 6: Wygenerowany kod

```
~ #include <stdio.h>
~ #include <stdlib.h>
~ int main ( ) {
~       int 1@ , 2@ , 3@ , 4@ , 5@ = _<number>_# ;
~       scanf( _<string>_ , & 2@ );
~       int 6@ [ 2@ ] ;
~       for( 3@ = _<number>_#  ; 3@ < = 2@ ; 3@ ++ ){
~           scanf ( _<string>_ , & 4@ );
~           if ( 2@ == 3@ ){
~               if ( 4@ ! = _<number>_#  ){
~                   4@ = 3@ / _<number>_#  ;
~                   printf ( _<string>_ , 5@ ) ;
~               }
~           }
~   return _<number>_#  ;
~ }
```

Rys. 7: Wygenerowany kod po formatowaniu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int a, b, c, d, e = 2;
6      scanf("%d", &b);
7      int f[b] ;
8      for(c = 0; c <= b ; c++){
9          scanf("%d", &d);
10         if (b == c){
11             if (d != 3){
12                 d = c / 2;
13                 printf("%d", e);
14             }
15         }
16     }
17
18     return 0;
19 }
```

```
> clang-7 -pthread -lm -o main main.c
> ./main
2
5
7
7
2>
```

Rys. 8: Kompilacja wygenerowanego kodu (online kompilator - Repl.it [4])

Jest widoczne, że wytrenowana sieć neuronowa spełnia swoje zadanie i jest w stanie generować programy w języku C. W przeważającej większości wypadków wygenerowany kod kompiluje się bez żadnych błędów i ostrzeżeń; przykład działania wygenerowanego kodu można zobaczyć na rys. 8 (wszystkie tokeny reprezentujące wartości literałów i nazwy zmiennych były zamienione na odpowiadające losowe liczby i poprawne ze względu na składnię języka C nazwy zmiennych).

2.3 Sieć do poprawienia błędów

2.3.1 Model Seq2Seq

Modele enkodera i dekodera są rekurencyjnymi sieciami neuronowymi, które składają się z kilku rodzajów warstw, oferowanych przez *Tensorflow*:

- *LSTM* – długa pamięć krótkotrwała,
- *Dense* – warstwa, dla której wszystkie jednostki poprzedniej warstwy są połączone ze wszystkimi w następnej,
- *Embedding* - przekształca dodatnie liczby całkowite w gęste wektory o stałym rozmiarze.

```
Model: "encoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        multiple                  28160
_________________________________________________________________
lstm (LSTM)                  multiple                  1574912
=================================================================
Total params: 1,603,072
Trainable params: 1,603,072
Non-trainable params: 0
```

Rys. 9: Architektura enkodera

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      multiple                  28160
_________________________________________________________________
dense (Dense)                multiple                  56430
_________________________________________________________________
lstm_cell_1 (LSTMCell)       multiple                  2623488
_________________________________________________________________
LuongAttention (LuongAttenti multiple                  262144
_________________________________________________________________
attention_wrapper (Attention multiple                  3409920
_________________________________________________________________
basic_decoder (BasicDecoder) multiple                  3466350
=================================================================
Total params: 3,494,510
Trainable params: 3,494,510
Non-trainable params: 0
_________________________________________________________________
Decoder Outputs Shape:  (16, 370, 110)
```

Rys. 10: Architektura dekodera

Jako dane wejściowe warstwa wejściowa przyjmuje sekwencję tokenów w postaci liczbowej, reprezentującą jeden wiersz kodu. Na wyjściu sieć generuje sekwencję tokenów, reprezentującą poprawiony kod.

Lizba neuronów w warstwie LSTM enkodera jest równa 512, liczba neuronów w warstwie *Embedding* - 256. Rozmiar kontekst wektora jest równy (*batch_size, units*) - (16, 512).

Lizba neuronów w warstwie LSTM dekodera jest równa 512, liczba neuronów w warstwie *Embedding* - 256. Jako mechanizm uwagi był wybrany *LuongAttention*. Jako dekoder był wybrany *BasicDecoder*.

Aby ocenić wydajność sieci neuronowej, do wytrenerowania modelu była wykorzystana kategoryczna entropia krzyżowa (ang. *categorical crossentropy*). Był także wykorzystany algorytm optymalizacji ADAM (ang. *adaptive moment estimation*).

2.3.2 Wyniki nauczania

Do trenowania modelu były wybrane takie parametry:

- *epoch* – 20
- *batch_size* – 16

Epoch odpowiada za liczbę epok użytych do trenowania modelu – jedna epoka oznacza przejście całego zbioru przez sieć. *Batch_size* odpowiada za zdefiniowanie ile rekordów (obserwacji) przechodzi na raz podczas pojedynczego przebiegu zanim nastąpi pierwsza aktualizacja wag parametrów.

Końcowa strata sieci neuronowej jest równa 0.029%. Czas treningu - 19 godz.

2.3.3 Wyniki działania sieci

Wyniki działania sieci można zaobserwować na poniższych rysunkach.

```
Input: int ch1[siz1]];
Output: ['~ int 1@ [ 2@ ] ; <eof>']
```

Rys. 11: Sieć wykryła błąd («]]»)

```
Input: for((i=0;i<siz1;i++)
Output: ['~ for ( 1@ = _<number>_# ; 1@ < 2@ ; 1@ ++ ) <eof>']
```

Rys. 12: Sieć wykryła błąd («((»)

```
Input: for(i=0;i<siz1;i++
Output: ['~ for ( 1@ = _<number>_# ; 1@ < 2@ ; 1@ ++ ) <eof>']
```

Rys. 13: Sieć wykryła błąd (brak «)»)

```
Input: int a[b;
Output: ['~ int 1@ [ 2@ ] ; <eof>']
```

Rys. 14: Sieć wykryła błąd (brak «]»)

```
Input: int a = 1
Output: ['~ int 1@ = _<number>_# ; <eof>']
```

Rys. 15: Sieć wykryła błąd (brak «;»)

```
Input: int a, b,, c;
Output: ['~ int 1@ , 2@ , 3@ ; <eof>']
```

Rys. 16: Sieć wykryła błąd («,,»)

Jest widoczne, że wytrenowana sieć neuronowa spełnia swoje zadanie i jest w stanie wykrywać i poprawiać błędy syntaktyczne (np. brak średników, zła ilość nawiasów). Także sieć skutecznie działa w przypadku występowania kilku błędów w jednej linii kodu, co można zaobserwować na rys. 17. Poza tym sieć może rozpoznać poprawny ze względu na składnię języka C kod - w takim przypadku sieć niczego nie zmienia w podanym na wejście kodzie (rys. 18).

```
Input: for(i=0 i<siz1;i++
Output: ['~ for ( 1@ = _<number>_# ; 1@ < 2@ ; 1@ ++ ) <eof>']
```

Rys. 17: Sieć wykryła dwa błędy (brak «;» i brak «)»)

```
Input: for(i=0;i<siz1;i++)
Output: ['~ for ( 1@ = _<number>_# ; 1@ < 2@ ; 1@ ++ ) <eof>']
```

Rys. 18: Sieć nie wykryła błędów w kodzie

W przeważającej większości wypadków wytrenowana sieć neuronowa może poprawnie wykrywać błędy syntaktyczne (albo brak błędów), ale w szczególnych wypadkach sieć może sama tworzyć błędy, co jest widoczne na rys. 19. Ten problem można wyeliminować zwiększeniem liczby epok lub liczby neuronów w enkoderze i dekoderze.

```
Input: scanf("%d",,siz1);
Output: ['~ scanf ( _<string>_ , 1@ ; <eof>']
```

Rys. 19: Sieć poprawiła błąd w linii kodu («,,»), ale na wyjściu wygenerowała nowy błąd (brak «)»)

Istotną wadą danego podejścia do klasyfikacji błędów jest to, że na wejście sieci jest podawana tylko jedna linia kodu, więc sieć może wykryć błędy tylko w kontekście podanego na wejściu kodu i nie jest w stanie wykryć błędy w znacznie większym kontekście (np. brak «}» na końcu funkcji lub pętli). Mimo to sieć spełnia swoje zadanie w wyszukiwaniu błędów w lokalnym kontekście, co jest pokazano na przykładzie analizy kodu na rys. 20, 21 i 22. Sieć skutecznie rozpoznała i poprawiła każdy z zaznaczonych na rys. 20 błędów syntaktycznych, i wygenerowany na wyjściu poprawny kod kompiluje się bez żadnych błędów i ostrzeżeń (rys. 22).

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i,siz1 = 5,,siz2;
    for(i=0;;i<siz1;;i++)
        printf("*";
    return 0
}
```

Rys. 20: Oryginalny kod (błędy są zaznaczone na czerwono)

```
~ #include <stdio.h> <eof>
~ #include <stdlib.h> <eof>
~ int main ( ) { <eof>
~       int 1@ , 2@ = _<number>_# , 3@ ; <eof>
~       for ( 1@ = _<number>_# ; 1@ < 2@ ; 1@ ++ ) <eof>
~           printf ( _<string>_ ) ; <eof>
~       return _<number>_# ; <eof>
~ } <eof>
```

Rys. 21: Wynik działania sieci

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main ( ) {
4      int a, b = 5, c;
5      for (a = 0; a < b; a++)
6          printf("*");
7      return 0;
8  }
```

```
> clang-7 -pthread -lm -o main main.c
> ./main
*****>
```

Rys. 22: Kompilacja poprawionego kodu (online kompilator - Repl.it [4])

3. PODSUMOWANIE

Dzięki realizacji tego projektu zdobyłam szeroki zakres wiedzy i umiejętności w dziedzinach takich jak:

- Pisanie w języku Python,
- zapoznanie z uczeniem maszynowym,
- zapoznanie z uczeniem głębokim,
- zapoznanie z działaniem rekurencyjnych sieci neuronowych,
- zapoznanie z działaniem sieci neuronowych Seq2Seq,
- rozwiązywanie problemów z dziedziny informatyki,
- organizacja pracy.

5. ŹRÓDŁA

[1] https://www.petrospsyllos.com/pl/prezentacje/blog/719-modele-seq2seq-oraz-transformery-sztuczna-inteligencja
[2] https://paperswithcode.com/dataset/deepfix
[3] https://geek.justjoin.it/zalety-wady-korzystania-tensorflow-srodowisku-produkcyjnym
[4] https://replit.com/