Computational Neuroscience, Digital Intelligence Laboratory

ASOCJACYJNA INDUKCJA GRAMATYKI REGULARNEJ

Waldemar Wietrzykowski

Instytut Inteligencji Stosowanej, 12 marzec 2023

Summary *this paper was created using the AI system (OpenAI Assistant), which utilizes the latest tools and technologies, and can be considered an interdisciplinary work. The paper presents a novel approach to inducing regular grammar using associative algorithms. The algorithm for creating a formal grammar describing the input language is described. The algorithm involves extracting the sequence of subsequent elements from the input string and generalizing them into the form of a regular grammar. The basis of the algorithm is the consolidation of two associations, which are combined if their sets of predecessors and successors are not empty. This allows gradually generalizing the associations until a formal grammar is obtained. The paper includes a description of the algorithm and Python functions implementing a single step of the associative machine, checking the consistency of input data with the grammar description stored in associations, analysis of the obtained results, and their interpretation. The paper presents examples of using associative induction for learning, recognizing, and performing basic arithmetic operations such as addition, and logical operations such as AND, OR, XOR, which may have applications in technologies other than microprocessors, such as biological neural networks.*

Translated into English by OpenAI Assistant.

Streszczenie *niniejsza praca została wykonana przy pomocy systemu AI (Asystent OpenAI), który wykorzystuje najnowsze narzędzia i technologie, i może być uważana za pracę interdyscyplinarną. W pracy tej przedstawiono nowatorskie podejście do indukcji gramatyki regularnej z użyciem algorytmów asocjacyjnych. Opisano algorytm tworzenia gramatyki formalnej opisującej język danych wejściowych. Algorytm polega na wyodrębnieniu relacji następstwa kolejnych elementów z ciągu wejściowego i uogólnieniu ich do postaci gramatyki regularnej. Podstawą algorytmu jest konsolidacja dwóch asocjacji, które są łączone, jeśli ich zbiory poprzedników i następników nie są puste. W ten sposób można stopniowo uogólniać asocjacje, aż do uzyskania gramatyki formalnej. Zamieszczono opis algorytmu oraz funkcje w języku Python realizujące pojedynczy krok maszyny asocjacyjnej, sprawdzenie zgodności danych wejściowych z opisem gramatyki zapisanej w asocjacjach oraz dokonano analizę uzyskanych wyników i przeprowadzono ich interpretację. W pracy przedstawiono przykłady wykorzystania asocjacyjnej indukcji regularnej do nauki, rozpoznawania i wykonywania podstawowych operacji arytmetycznych takich jak dodawanie oraz operacji logicznych takich jak AND, OR, XOR, co może mieć zastosowanie w technologiach innych niż mikroprocesory na przykład biologicznych sieciach neuronowych.*

Wprowadzenie

Algorytm indukcji gramatyki regularnej jest używany do tworzenia gramatyki formalnej opisującej język danych wejściowych. Może być on wykorzystywany w aplikacjach związanych z przetwarzaniem języka naturalnego, takich jak analiza składniowa, parsowanie i generowanie tekstu.

Poniżej opisano przykład takiego algorytmu. Opiera się on na wyodrębnieniu relacji następstwa kolejnych elementów z danych wejściowych, a następnie ich uogólnianiu do postaci gramatyki regularnej. Odbywa się to w ten sposób, że relacje następstwa są ekstrahowane (wyodrębnianie) z ciągu wejściowego do ciągu asocjacji jednostkowych a następnie kolejno wyodrębnione asocjacje są konsolidowane z asocjacjami już wyodrębnionymi.

Opis algorytmu

Niech przykładowo ciąg wejściowy: data = 1, 2, 3, 4, 5, 6. Wyodrębnieniem relacji następstwa kolejnych elementów tego ciągu będzie: 1 2, 2 3, 3 4, 4 5, 5 6, gdzie 1 2, 2 3, itd. będą nazywać się asocjacjami jednostkowymi będącymi parami elementów, w którym pierwszy element nazywa się poprzednik asocjacji a drugi następnik asocjacji. W języku Python taką asocjację jednostkową można przedstawić w postaci listy dwuelementowej, np. association_unit = [{1},{2}]. W asocjacją mamy do czynienia wtedy, kiedy z zbiorze typu set może wystąpić nie tylko jeden elementem ale kilka, np. association = [{1,3},{2}]. Stąd asocjacja jednostkowa jest też asocjacją z pojedynczymi elementami z zbiorze poprzednika i zbiorze następnika.

Podstawowym elementem tego algorytmu jest konsolidacja dwóch asocjacji. Przypuśćmy, że mamy dwie asocjacje P1 N1 oraz P2 N2, gdzie P1, P2 są poprzednikami a N1, N2 są następnikami tych asocjacji i jeżeli są one nierozłączne, to można połączyć je w jedną asocjację P N, gdzie poprzednik asocjacji P jest sumą zbiorów P1 i P2, a następnik asocjacji N jest sumą zbiorów N1 i N2.

Konsolidację można zapisać następująco:

Jeżeli

$(P1 \cap P2 \neq \emptyset) \vee (N1 \cap N2 \neq \emptyset)$

to

$P1\ N1, P2\ N2 \Rightarrow (P1 \cup P2)\ (N1 \cup N2)$

gdzie:

$\cap$ - iloczyn zbiorów

$\cup$ - suma zbiorów

$\emptyset$ - zbiór pusty

Jeżeli asocjację P1 N1 oznaczymy przez assoc1, gdzie P1 = assoc1[0] a N1 = assoc1[1], a asocjację P2 N2 oznaczymy przez assoc2, gdzie P2 = assoc2[0] a N2 = assoc[1], to konsolidację tą w języku Python można zapisać następująco:

```
def cons(assoc1, assoc2):

  if assoc1[0] & assoc2[0] or assoc1[1] & assoc2[1]:
        assoc1[0] = assoc1[0] | assoc2[0]
        assoc1[1] = assoc1[1] | assoc2[1]
        return True                          # asocjacje są łączne to skonsoliduj je, a wynik jest w "assoc1"
  else:
        return False                         # asocjacje są rozłączne więc nie wykonuj żadnych działań
```

Kolejno wyodrębniona asocjacja z ciągu wejściowego jest konsolidowana z asocjacjami już wyodrębnionymi. Realizuje to funkcja „list_cons”:

```
def list_cons(list_assoc, unit_assoc):
# "list_cons" - konsolidacja listy asocjacji "list_assoc" z asocjacją jednostkową "unit_assoc"
#                       Wynik konsolidacji w "list_assoc"
#                       Uwaga! Parametry "list_assoc" i "unit_assoc" przekazane są przez referencję

    list_join = []                  # lista zapamiętanych asocjacji połączonych z asocjacją "unit_assoc"

    for assoc in list_assoc:        # przegląd kolejnych elementów "assoc" listy asocjacji "list_assoc"
                                        # Uwaga! Asocjacja "assoc" jest połączona przez referencję z elementem listy "list_assoc"

        if cons(assoc, unit_assoc): # konsolidacja elementu "assoc" listy "list_assoc" z asocjacją jednostkową "unit_assoc"
            list_join += [assoc]    # zapamiętaj, jeżeli asocjacja "assoc" była połączona z asocjacją jednostkową "unit_assoc"

    if list_join == []:             # w przypadku braku asocjacji połączonych z asocjacją jednostkową "unit_assoc"
            list_assoc += [unit_assoc]  # inicjowanie listy asocjacji "list_assoc" asocjacją jednostkową "unit_assoc"
    elif len(list_join) == 2:       # jeżeli dwie asocjacje z listy "list_assoc" zostały połączone z "unit_assoc" to
            cons(list_join[0],list_join[1]) # należy je połączyć i zwrócić do listy "list_asoc" połączone
            list_assoc.remove(list_join[1]) # Zbędny element na liście "list_assoc" należy usunąć.
                                            # list_join[1] jest połączony z tym elementem przez referencję
```

Funkcja „list_cons” jest zastosowana do wszystkich wyodrębnionych asocjacji z ciągu wejściowego przy pomocy funkcji „data_cons”

```
def data_cons(list_assoc, data):
# "data_cons" - konsolidacja ciągu danych "data" w listę skonsolidowanych asocjacji "list_assoc"
# "Uwaga! Parametr list_assoc przekazany przez referencję
    list_assoc = [list_cons(list_assoc, [{data[i]}, {data[i+1]}]) for i in range(len(data)-1)]
```

Cały algorytm do wyodrębnienia relacji następstwa kolejnych elementów z danych wejściowych, a następnie ich uogólnieniu do postaci gramatyki regularnej wygląda następująco:

```
def cons(assoc1, assoc2):
# "cons" - konsolidacja dwóch asocjacji "assoc1" i "assoc2" jeżeli są nierozłączne
#           Wynik konsolidacji w "assoc1"
#           Uwaga! Parametry "assoc1" i "assoc2" przekazane są przez referencję
#                   ponieważ asocjacje są mutowalne (listy)

    if assoc1[0] & assoc2[0] or assoc1[1] & assoc2[1]:
        assoc1[0] = assoc1[0] | assoc2[0]
        assoc1[1] = assoc1[1] | assoc2[1]
        return True                         # asocjacje są łączne to skonsoliduj je, a wynik jest w "assoc1"
    else:
```

```
            return False                        # asocjacje są rozłączne więc nie wykonuj żadnych działań

def list_unit(list_assoc, unit_assoc):
# "list_unit" - konsolidacja listy asocjacji "list_assoc" z asocjacją jednostkową "unit_assoc"
#                       Wynik konsolidacji w "list_assoc"
#                       Uwaga! Parametry "list_assoc" i "unit_assoc" przekazane są przez referencję

    list_join = []                      # lista zapamiętanych asocjacji połączonych z asocjacją "unit_assoc"

    for assoc in list_assoc:        # przegląd kolejnych elementów "assoc" listy asocjacji "list_assoc"
                                            # Uwaga! Asocjacja "assoc" jest połączona przez referencję z elementem listy
"list_assoc"

        if cons(assoc, unit_assoc): # konsolidacja elementu "assoc" listy "list_assoc" z asocjacją jednostkową
"unit_assoc"
            list_join += [assoc]      # zapamiętaj, jeżeli asocjacja "assoc" była połączona z asocjacją jednostkową
"unit_assoc"

    if list_join == []:             # w przypadku braku asocjacji połączonych z asocjacją jednostkową "unit_assoc"
            list_assoc += [unit_assoc]    # inicjowanie listy asocjacji "list_assoc" asocjacją jednostkową "unit_assoc"
    elif len(list_join) == 2:     # jeżeli dwie asocjacje z listy "list_assoc" zostały połączone z "unit_assoc" to
            cons(list_join[0],list_join[1]) # należy je połączyć i zwrócić do listy "list_asoc" połączone
            list_assoc.remove(list_join[1]) # Zbędny element na liście "list_assoc" należy usunąć.
                                                    # list_join[1] jest połączony z tym elementem przez referencję

def data_list(list_assoc, data):
# "data_list" - konsolidacja ciągu danych "data" w listę skonsolidowanych asocjacji "list_assoc"
# "Uwaga! Parametr list_assoc przekazany przez referencję
    list_assoc = [list_unit(list_assoc, [{data[i]}, {data[i+1]}]) for i in range(len(data)-1)]

list_assoc = []                         # Inicjacja listy asocjacji

data = [3,5,0,6,4,2,1,0,3,6,4,7,1,0,6,2,1,0,6,1,0,6,7,2,1,5,3,3,6,1,0] # ciąg danych
data_list(list_assoc, data) # Zamiana ciągu danych "data" w listę skonsolidowanych asocjacji "list_assoc"
print(list_assoc)
```

Zastosowany algorytm do ciągu wejściowego:

data = [3,5,0,6,4,2,1,0,3,6,4,7,1,0,6,2,1,0,6,1,0,6,7,2,1,5,3,3,6,1,0]

daje gramatykę regularną w postaci listy asocjacji:

list_assoc = [[{0, 1, 3, 5}, {0, 3, 5, 6}], [{2, 4, 6, 7}, {1, 2, 4, 7}]]

Uniwersalność algorytmu

Kolejną sprawą jest sprawdzenie, że podany w tej pracy algorytm jest uniwersalny, w sensie przetwarzania różnych ciągów wejściowych, nie tylko ciągów złożonych z liczb.

Dla ciągu wejściowego:

data = ['p','q','r','s','q','s','r','t','u','w']

uzyskany ciąg asocjacji jest równy

list_assoc = [[{'p', 'r', 'q', 's'}, {'q', 's', 't', 'r'}], [{'t'}, {'u'}], [{'u'}, {'w'}]]

Jeżeli ciąg wejściowy będzie równy:

data = [(128, 0, 128), (128, 0, 128), (128, 128, 64), (128, 128, 64), (128, 0, 128), (128, 128, 64), (128, 128, 64), (128, 0, 128), (128, 0, 0), (0, 0, 0), (128, 0, 128), (128, 128, 64), (128, 128, 64), (128, 0, 128), (255, 255, 255), (255, 255, 255), (128, 0, 128), (128, 0, 128), (128, 0, 128), (128, 0, 128), (0, 0, 0), (128, 128, 64), (128, 0, 128), (0, 0, 0), (255, 255, 255), (128, 0, 128), (128, 128, 64), (128, 0, 128), (255, 255, 255), (128, 0, 128), (128, 0, 128), (128, 0, 128), (128, 128, 64)]

uzyskamy ciąg asocjacji równy:

list_assoc = [[{(255, 255, 255), (128, 128, 64), (128, 0, 0), (0, 0, 0), (128, 0, 128)}, {(255, 255, 255), (128, 128, 64), (128, 0, 0), (0, 0, 0), (128, 0, 128)}]]

Z powyższego wynika, że algorytm jest uniwersalny i potrafi przetwarzać ciągi wejściowe o różnej strukturze pod warunkiem, że asocjacje jednostkowe wyodrębnione z ciągu wejściowego są liczbami, stringami i tuple.

Zalety algorytmu

1. Największą zaletą algorytmu jest jego duża prostota i mała złożoność funkcjonowania, co stanowi wygodną podstawę do opracowania bardziej złożonych modeli.

2. Jest on oparty na języku Python, powszechnie stosowanym w przedsięwzięciach badawczych.

Szczegółowa analiza i interpretacja generowanej listy asocjacji

Opisany w niniejszej pracy algorytm jest algorytmem indukcyjnym opartym na automacie skończonym deterministycznym lub też samo-uczącą się jednokierunkową maszyną Turinga. Jedną z możliwości maszyny jest sprawdzanie, czy ciąg danych wejściowych jest zgodny z gramatyką zapisaną na liście asocjacji.

Krok maszyny asocjacyjnej składa się z następujących etapów: wczytanie aktualnej danej z ciągu danych wejściowych. Sprawdzenie, czy dana ta jest już zapisana w zbiorze danych następnika aktualnej asocjacji. Jeżeli jest taki zapis to poszukiwana jest następna asocjacja, w której w zbiorze poprzednika znajduje się ta dana. Jeżeli odczytana dana nie jest zapisana już w zbiorze następnika aktualnej asocjacji lub nie ma takiej asocjacji, w której w zbiorze poprzednika zapisana jest ta dana, jest ona niezgodna z gramatyką napisaną na liście asocjacji i maszyna kończy pracę z flagą niezgodności. Jak z tego wynika kolejny stan maszyny (w postaci asocjacji) jest zależny od poprzedniego stanu (poprzedniej asocjacji) oraz wczytanej danej wejściowej, a to jest właściwość automatu regularnego. Jeżeli ponadto każda wczytana dana może należeć do zbioru następnika przynajmniej jednej asocjacji to sama maszyna będzie deterministyczną. Stąd asocjacyjna maszyna Turinga posiadająca ten algorytm odpowiada automatowi skończonemu deterministycznemu.

Asocjacyjna indukcja gramatyki regularnej polega na przetwarzaniu ciągu danych wejściowych na listę asocjacji, której zadaniem jest możliwość sprawdzenia, czy i inne ciągi wejściowe są zgodne z gramatyką regularną opisaną tą listą asocjacji.

Podstawowym elementem sprawdzenia zgodności ciągu wejściowego z gramatyką określoną listą asocjacji jest sprawdzenie zgodności dwóch asocjacji P1 N1 oraz P2 N2, gdzie P1, P2 są poprzednikami a N1, N2 są następnikami tych asocjacji. Jeżeli są one zgodne to:

$$(P1 \cap P2 \neq ø) \wedge (N1 \cap N2 \neq ø)$$

gdzie:

∩ - iloczyn zbiorów

ø - zbiór pusty

Jeżeli asocjację P1 N1 oznaczymy przez assoc1, gdzie P1 = assoc1[0] a N1 = assoc1[1], a asocjację P2 N2 oznaczymy przez assoc2, gdzie P2 = assoc2[0] a N2 = assoc[1], to zgodność tą w języku Python można zapisać następująco:

```
def comp(assoc1, assoc2):
# "comp" - zgodność dwóch asocjacji "assoc1" i "assoc2"

    if assoc1[0] & assoc2[0] and assoc1[1] & assoc2[1]:
        return True      # asocjacje są zgodne
    else:
        return False    # asocjacje są niezgodne
```

W pierwszym etapie szukamy na liście „list_assoc” takiej asocjacji, która jest zgodna z asocjacją jednostkową „unit_assoc” wyodrębnioną z danych wejściowych. Realizuje to funkcja „list_comp”:

```
def list_comp(list_assoc, unit_assoc):
# "list_comp" - wyszukiwanie asocjacji na liście    "list_assoc" zgodnej z asocjacją jednostkową "unit_assoc"

    for assoc in list_assoc:       # przegląd kolejnych elementów "assoc" listy asocjacji "list_assoc"

        if comp(assoc, unit_assoc): # sprawdzenie zgodności "assoc" listy "list_assoc" z asocjacją jednostkową
"unit_assoc"
            return True    # asocjacje "assoc" i "unit_assoc" są zgodne, zakończ przeszukiwanie
        else:
            pass    # asocjacje "assoc" z "unit_assoc" są niezgodne, szukaj dalej

    return False    # wszystkie asocjacje na liście "list_assoc" są niezgodne z "unit_assoc"
```

Sprawdzanie zgodności całego ciągu wejściowego z listą asocjacji wygląda następująco:

```
def data_comp(list_assoc, data):
# "data_comp" - zgodność ciągu danych "data" z listą asocjacji "list_assoc"
```

```
    for i in range(len(data)-1):
        if list_comp(list_assoc,[{data[i]}, {data[i+1]}]): # sprawdzenie zgodności listy asocjacji z asocjacją jednostkową
            pass
        else:
            return False # lista asocjacji nie jest zgodna z asocjacją jednostkową, zakończ dalsze sprawadzanie
zgodności
    return True # lista asocjacji jest zgodna z ciągiem danych "data"
```

Cały algorytm do sprawdzania zgodności ciągu wejściowego z listą asocjacji wygląda następująco:

```
def comp(assoc1, assoc2):
# "comp" - zgodność dwóch asocjacji "assoc1" i "assoc2"

    if assoc1[0] & assoc2[0] and assoc1[1] & assoc2[1]:
        return True      # asocjacje są zgodne
    else:
        return False    # asocjacje są niezgodne

def list_comp(list_assoc, unit_assoc):
# "list_comp" - wyszukiwanie asocjacji na liście    "list_assoc" zgodnej z asocjacją jednostkową "unit_assoc"

    for assoc in list_assoc:       # przegląd kolejnych elementów "assoc" listy asocjacji "list_assoc"

        if comp(assoc, unit_assoc): # sprawdzenie zgodności "assoc" listy "list_assoc" z asocjacją jednostkową
"unit_assoc"
            return True    # asocjacje "assoc" i "unit_assoc" są zgodne, zakończ przeszukiwanie
        else:
            pass    # asocjacje "assoc" z "unit_assoc" są niezgodne, szukaj dalej

    return False    # wszystkie asocjacje na liście "list_assoc" są niezgodne z "unit_assoc"

def data_comp(list_assoc, data):
# "data_comp" - zgodność ciągu danych "data" z listą asocjacji "list_assoc"

    for i in range(len(data)-1):
        if list_comp(list_assoc,[{data[i]}, {data[i+1]}]): # sprawdzenie zgodności listy asocjacji z asocjacją jednostkową
            pass
        else:
            return False # lista asocjacji nie jest zgodna z asocjacją jednostkową, zakończ dalsze sprawadzanie
zgodności
    return True # lista asocjacji jest zgodna z ciągiem danych "data"

# list_assoc = []   # Inicjacja listy asocjacji
# data = [3,5,0,6,4,2,1,0,3,6,4,7,1,0,6,2,1,0,6,1,0,6,7,2,1,5,3,3,6,1,0] # ciąg danych
print(data_comp(list_assoc, data)) # Sprawadzenie zgodności "data" z listą asocjacji "list_assoc"
```

Zastosowany algorytm do sprawdzenia zgodności w przypadku ciągu wejściowego:

data = [3,5,0,6,4,2,1,0,3,6,4,7,1,0,6,2,1,0,6,1,0,6,7,2,1,5,3,3,6,1,0]

oraz gramatyką regularną określoną listą asocjacji:

list_assoc = [[{0, 1, 3, 5}, {0, 3, 5, 6}], [{2, 4, 6, 7}, {1, 2, 4, 7}]]

daje wynik „True”, co oznacza, że ciąg wejściowy jest zgodny z podaną listą asocjacji. Wynika z tego, że utworzona lista asocjacji z podanego ciągu wejściowego jest zgodna z tym ciągiem wejściowym.

Rozpoznawanie operacji matematycznych

Podstawowe operacje matematyczne jak: suma arytmetyczna (+), iloczyn logiczny (AND), suma logiczna (OR), alternatywa rozłączna (XOR), itd. są realizowane elektronicznie przy pomocy jednostki arytmetyczno logicznej mikroprocesora danego komputera. Jak się okazuje operacje te mogą być również wykonywane przez maszynę asocjacyjną i odpowiadają one gramatykom zapisanym na liście asocjacji. W odróżnieniu od mikroprocesora maszyna asocjacyjna sama może nauczyć się wykonywać te operacyjne po zaledwie jednym uczeniu. Liczby, na których wykonywane są operacje muszą być wstępnie przygotowane do tego. Komórki taśmy maszyny posiadają trzy bity: jeden bit należy do pierwszej liczby, drugi bit należy do drugiej liczby, a trzeci bit należy do wyniku operacji. W celu przygotowania danych wejściowych do maszyny asocjacyjnej na podstawie liczb: X1, X2, Y, zamieniamy liczby X1, X2, Y na postać binarną przy pomocy funkcji „bin()”. Następnie zapisy liczb binarnych, mające na początku znaki „0b”, okrajamy przy pomocy operacji okrojenia „[2:]”. Potem znajdujemy maksymalną długość z liczb X1, X2, Y, w następujący sposób: „len_max = max(len(X1),len(X2), len(Y))”. Na koniec dopełniamy zerami liczby X1, X2, Y do maksymalnej długości „X1 = X1.zfill(len_max)”, „X2 = X2.zfill(len_max)”, „Y = Y.zfill(len_max)” i łączymy liczby X1, X2, Y w jeden ciąg danych wejściowych.

```
def get_data(x1_int, x2_int, y_int):
    x1 = bin(x1_int)[2:]      # x1_int w zapisie binarnym
    x2 = bin(x2_int)[2:]      # x2_int w zapisie binarnym
    y = bin(y_int)[2:]        # y_int   w zapisie binarnym
    len_max = max(len(x1),len(x2),len(y)) # długość najdłuższej liczby binarnej
    x1 = x1.zfill(len_max)   # dopełnienie zerami do jednej długości liczb binarnych
    x2 = x2.zfill(len_max)
    y = y.zfill(len_max)

    return [
        int(y[i]+x2[i]+x1[i],2) # zmiana trzech bitów w pozycji i na liczbę typu int
        # for i in range(len_max-1,-1,-1) # pętla w przeciwnym kierunku od len_max-1 do 0
        for i in range(len_max-1,-1,-1) # pętla w przeciwnym kierunku od len_max do 0
            ]      # od najmniej znaczącego do najbardziej znaczącego bitu
```

Przeprowadzono test działania algorytmu rozpoznawania operacji matematycznych na podstawie tworzonych asocjacji. Dla poszczególnych operacji matematycznych: sumy arytmetycznej (+), iloczynu logicznego (AND), sumy logicznej (OR), alternatywy rozłącznej (XOR) wykonano działania na dwóch liczbach całkowitych, wygenerowano ciąg danych dla maszyny asocjacyjnej, a następnie uzyskano asocjacje będących gramatyką operacji matematycznych. Zauważono, że dla małych liczb, konsolidacja asocjacji nie była wystarczająca do uzyskania pełnej gramatyki. Każdej

operacji matematycznej, niezależnie od wyboru liczb, gramatyka operacji była ta sama. Wyniki przedstawiono w poniższej tabeli.

Lp	x1	x2	operacja	y	lista asocjacji
1	793319747	484756265	+	1278076012	[[{1, 2, 3, 7}, {1, 2, 4, 7}], [{0, 4, 5, 6}, {0, 3, 5, 6}]]
2	384562749	27369478	+	411932227	[[{0, 4, 5, 6}, {0, 3, 5, 6}], [{1, 2, 3, 7}, {1, 2, 4, 7}]]
3	2400456	16005647	+	18406103	[[{0, 4, 5, 6}, {0, 3, 5, 6}], [{2, 3}, {2, 4}]]
4	8399566721	1826389346	+	10225956067	[[{0, 4, 5, 6}, {0, 3, 5, 6}], [{1, 2, 3, 7}, {1, 2, 4, 7}]]
5	234	835	+	1069	[[{5, 6}, {0, 3}], [{1, 2, 3}, {1, 2, 4}], [{0, 4}, {5}]]
6	793319747	484756265	AND (&)	205523201	[[{0, 1, 2, 7}, {0, 1, 2, 7}]]
7	384562749	27369478	AND (&)	10592260	[[{0, 1, 2, 7}, {0, 1, 2, 7}]]
8	2400456	16005647	AND (&)	2367496	[[{0, 1, 2, 7}, {0, 1, 2, 7}]]
9	8399566721	1826389346	AND (&)	1686385920	[[{0, 1, 2, 7}, {0, 1, 2, 7}]]
10	234	835	AND (&)	66	[[{0, 1, 2, 7}, {0, 1, 2, 7}]]
11	793319747	484756265	OR (|)	1072552811	[[{0, 5, 6, 7}, {0, 5, 6, 7}]]
12	384562749	27369478	OR (|)	401339967	[[{0, 5, 6, 7}, {0, 5, 6, 7}]]
13	2400456	16005647	OR (|)	16038607	[[{0, 5, 6, 7}, {0, 5, 6, 7}]]
14	8399566721	1826389346	OR (|)	8539570147	[[{0, 5, 6, 7}, {0, 5, 6, 7}]]
15	234	835	OR (|)	1003	[[{0, 5, 6, 7}, {0, 5, 6, 7}]]
16	793319747	484756265	XOR (^)	867029610	[[{0, 3, 5, 6}, {0, 3, 5, 6}]]
17	384562749	27369478	XOR (^)	390747707	[[{0, 3, 5, 6}, {0, 3, 5, 6}]]
18	2400456	16005647	XOR (^)	13671111	[[{0, 3, 5, 6}, {0, 3, 5, 6}]]
19	8399566721	1826389346	XOR (^)	6853184227	[[{0, 3, 5, 6}, {0, 3, 5, 6}]]
20	234	835	XOR (^)	937	[[{0, 3, 5, 6}, {0, 3, 5, 6}]]

Dokonywanie operacji matematycznych na podstawie listy asocjacji

Jeżeli operacje matematyczne mogą być wyrażone przez maszynę asocjacyjną w postaci gramatyki zapisanej na liście asocjacji, to dysponując konkretną gramatyką można wykonywać operacje matematycznie przez nią wyrażoną. Na dwóch liczbach całkowitych wykonano różne operacje matematycznie w zależności od wskazanych asocjacji. Operacje te wykonano przy użyciu maszyny asocjacyjnej. Przed wykonaniem operacji z dwóch liczb wygenerowano ciąg danych wejściowych. Sposób generacji podobny do opisanego wcześniej z tym, że wynik operacji ma zostać wygenerowany przez maszynę asocjacyjną. Przy znajdowaniu maksymalnej długości liczb binarnych zwiększamy ją o 1, w przypadku możliwego przeniesienia przy operacji dodawania. Ostatecznie każda komórka taśmy z danymi przechowuje dwa bity x1, x2 należące do liczb X1, X2. Zadaniem maszyny jest wygenerowanie w każdej komórce trzeciego bitu y należącego do poszukiwanej liczby Y.

Mechanizm działania maszyny asocjacyjnej jest następujący: maszyna startuje od asocjacji początkowej, którą standardowo jest pierwsza asocjacja na liście asocjacji, i pobiera zawartość pierwszej komórki zawierającej pierwsze najmniej znaczące bity liczb X1 i X2, a następnie szuka, czy w następniku asocjacji początkowej nie znajduje się element posiadający te same bity x1x2 co w komórce taśmy. Z samego założenia, że maszyna jest deterministyczna wynika, że dla każdej asocjacji

istnieje co najwyżej jeden taki element o tym samym x1x2. Po znalezieniu tego elementu w postaci x1x2y w następnym ruchu maszyna wybiera taką asocjację, która w poprzedniku zawiera x1x2y i przechodząc na następną komórkę, rozpoczyna omówioną sekwencję po raz kolejny.

Kolejne ruchy maszyny przedstawiają pętla, w której danymi wejściowymi są bity x1x2. Następnie tworzone są elementy x1x2,0 lub x1x2,1, które szukane są w następniku asocjacji bieżącej. Znalezienie tych elementów oznacza wynik 0 lub 1. Do zamiany x1x2 w x1x2,0 lub x1x2,1 służy funkcja "x1x2y(x1x2, y)". Pętla z każdym kroku zwraca znaleziony w następniku asocjacji element x1x2y, który służy do rozpoczęcia kolejnego kroku od asocjacji mającej w poprzedniku x1x2y.
Kolejno znalezione x1x2y są odkładane na taśmie maszyny. Poniższej zamieszczono kod w języku Python wykonywania operacji matematycznych na podstawie listy asocjacji.

```
def get_data(x1_int, x2_int):
    x1 = bin(x1_int)[2:]
    x2 = bin(x2_int)[2:]
    len_max = max(len(x1),len(x2))+ 1 # pozycja dla przeniesienia
    x1 = x1.zfill(len_max)
    x2 = x2.zfill(len_max)
    return [
        int(x2[i]+x1[i],2) # zmiana trzech bitów w pozycji i na liczbę typu int
        for i in range(len_max-1,-1,-1) # pętla w przeciwnym kierunku od len_max-1 do 0
            ]     # od najmniej znaczącego do najbardziej znaczącego bitu

def get_assoc_next(assoc, x2x1_int):
    x2x1 = bin(x2x1_int)[2:]       # bity x1x2 bez wiodącego '0b'
    x2x1 = x2x1.zfill(2)           # dopełnienie zerami do dwóch pozycji

    yx2x1_int = int('0'+x2x1,2) # zaproponowanie bitów x1x2y z bitem y = 0
    if yx2x1_int in assoc[1]:     # sprawdzenie czy ta propozycja występuje w następniku asocjacji bieżącej
        return yx2x1_int             # jeżeli występuje to zwróć tą wartość
                                     # a jezeli nie występuje
    yx2x1_int = int('1'+x2x1,2) # zaproponowanie bitów x1x2y z bitem y = 1
    if yx2x1_int in assoc[1]:     # sprawdzenie czy ta propozycja występuje w następniku asocjacji bieżącej
        return yx2x1_int             # jeżeli występuje to zwróć tą wartość

    else:
        return None                  # zwróć "None" jeżeli w następniku asocjacji bieżącej nie ma elementu z x1x2

def get_data_y(list_assoc, data):
# data_y - na podstawie ciągu bitów x1x2 generuje ciąg bitów x1x2y
    stan = 0
    assoc1 = list_assoc[0]   # asocjacja bieżąca
    for i in range(len(data)):
        yx2x1_int = get_assoc_next(assoc1,data[i]) # na podstawie bitów x1x2 i asocjacji bieżącej generuje bity x1x2y
        print(stan,': ',data[i],' -> ',yx2x1_int)
        if yx2x1_int != None:    # w następniku asocjacji bieżącej znaleziono element z x1x2
            data[i] = yx2x1_int # jeżeli w następniku asocjacji bieżącej jest element x1x2 zamień x1x2 na x1x2y
        else:
            return False # w następniku asocjacji bieżącej nie znaleziono elementu x1x2

        # poszukiwanie następnej asocjacji na podstawie elementu x1x2y w poprzedniku
        stan = 0;
        for assoc in list_assoc:       # przegląd kolejnych elementów "assoc" listy asocjacji "list_assoc"
            if yx2x1_int in assoc[0]:
```

```
                    assoc1 = assoc          # ustaw asocjację bieżącą
                    break                   # przerwij szukanie
                else:
                    stan += 1
                    pass                    # szukaj dalej
            else:     # sekcja "else" zostanie wykonana, jeżeli pętla "for" nie zostanie zatrzymana przez instrukcję "break"
                return False
        return True # jeżeli pętla zostaje zatrzymana przez "break"

def get_y(data):
# generacja liczby y z ciągu wyjściowego data
        y = '';
        for i in range(len(data)-1,-1,-1):
                y += str((data[i] & 4) >> 2)

        return int(y,2)

x1, x2 =    793319747, 484756265
list_assoc = [[{0, 4, 5, 6}, {0, 3, 5, 6}], [{1, 2, 3, 7}, {1, 2, 4, 7}]]

data = get_data(x1, x2)
print(data,' LSB -> MSB')
if get_data_y(list_assoc, data):
        print(data)
else:
        print(data,' error!')

print(get_y(data)) # wynik operacji na liczbach x1, x2
```

W poniższej tabeli zamieszczono wyniki operacji matematycznych na podstawie asocjacji odpowiadającym sumie arytmetycznej (+), iloczynowi logicznemu (AND), sumie logicznej (OR), alternatywie rozłącznej (XOR). Uzyskane wyniki potwierdzają, że operacje matematyczne realizowane przez drogą elektroniczną przy pomocy jednostki arytmetyczno logicznej mikroprocesora danego komputera mogą również wykonywać maszyny asocjacyjna, ucząc się wcześniej tych operacji.

Lp	x1	x2	lista asocjacji	wynik operacji
1	793319747	484756265	[[{1, 2, 3, 7}, {1, 2, 4, 7}], [{0, 4, 5, 6}, {0, 3, 5, 6}]] równoważne operacji +	1278076012
2	384562749	27369478		411932227
3	2400456	16005647		18406103
4	8399566721	1826389346		10225956067
5	234	835		1069
6	793319747	484756265	[[{0, 1, 2, 7}, {0, 1, 2, 7}]] równoważne operacji AND (&)	205523201
7	384562749	27369478		10592260
8	2400456	16005647		2367496
9	8399566721	1826389346		1686385920
10	234	835		66
11	793319747	484756265	[[{0, 5, 6, 7}, {0, 5, 6, 7}]] równoważne operacji OR (|)	1072552811
12	384562749	27369478		401339967
13	2400456	16005647		16038607
14	8399566721	1826389346		8539570147

15	234	835		1003
16	793319747	484756265	[[{0, 3, 5, 6}, {0, 3, 5, 6}]] równoważne operacji XOR (^)	867029610
17	384562749	27369478		390747707
18	2400456	16005647		13671111
19	8399566721	1826389346		6853184227
20	234	835		937

Znaczenie maszyny asocjacyjnej

Wykonywanie operacji arytmetyczno logicznych może być realizowane w technologii krzemowej przy pomocy mikroprocesora. Maszyna asocjacyjna pokazuje, że operacje te mogą być wykonywane w innych technologiach, w których da się zaimplementować mechanizm konsolidacji asocjacji. Jednym z reprezentacji takich technologii są biologiczne sieci neuronowe lub nawet sieci oparte na włókienkach *neurofibrylowych* w jednokomórkowcach. Asocjacje regularne wyznaczają natywny model biologicznych sieci neuronowych. Ogromna sieć powiązanych ze sobą neuronów i nieprzeliczona ilość impulsów między nimi, trudna do analizy, może być rozpatrywania w sposób kompleksowy jako asocjacje między grupami neuronów. Ponadto asocjacje są dobrym kandydatem do wyjaśnienie mechanizmu przeskoku z poziomu zdarzeń fizycznych (neuronalnych) do poziomu zdarzeń umysłowych, czyli są sposobem na rozwiązanie trudnego problemu świadomości. Dodatkowo, znaczenie asocjacji umacnia dobry model ich działania, od tworzenia do funkcjonowania w postaci samo-uczącej się maszyny asocjacyjnej. Ponadto, według teorii Noama Chomsky'ego gramatyka regularna lub skończenie stanowa zawiera się w gramatyce wyższego rzędu w tym gramatyce języka naturalnego, tak więc zajmując się aspektami gramatyki regularnej częściowo rozstrzygamy też pewne wycinki języka naturalnego.

Model neuronalny maszyny asocjacyjnej

W modelu tym asocjację elementarną reprezentuje pojedynczy neuron, w którym jedna z synaps jest synapsą nieswoistą (niespecyficzną) i pobudza ona neuron podprogowo. Połączona jest ona z askonem innego lub tego samego neuronu. Synapsa ta spełnia rolę poprzednika asocjacji elementarnej. Pozostałe synapsy neuronu są synapsami swoistymi (specyficznymi) i pełnią rolę części warunkowej w następniku asocjacji elementarnej. Synapsy te otrzymują pobudzenia z dróg czuciowych, a wagi połączeń tych synaps reprezentują wzorzec elementu informacji.

Załóżmy, że w pewnym momencie jeden z neuronów został uaktywniony. W jego aksonie pojawia się impuls nerwowy, który przechodzi drogą aksonalną do drugiego neuronu (kolejnej asocjacji) i pobudza jego synapsę nieswoistą podprogowo. Pobudzenie to nie jest jeszcze wystarczające do uaktywnienia tego neuronu. Dopiero dodatkowe skuteczne pobudzenie synaps swoistych (zgodne z wagami tych synaps) powoduje uaktywnienie drugiego neuronu. Oznacza to, że warunek zawarty w następniku asocjacji został spełniony. Należy zauważyć, że bez podprogowego pobudzenia synapsy nieswoistnej niemożliwe jest pobudzenie neuronu, nawet jeżeli pobudzenie synaps swoistych jest zgodne z wagami tych synaps. Warunkiem uaktywnienia neuronu jest równoczesne pobudzenie synapsy nieswoistej jak i skutecznie pobudzenie synaps swoistych.

Analizując konsolidację wykonywaną przez samą asocjacyjną maszynę Turinga (omówioną w tej pracy) można powiedzieć, że w uczeniu sieci neuronowej ważne są dwa momenty. Pierwszy to ten, w którym impuls z uaktywnionego neuronu, drogą aksonalną, nie może znaleźć kolejnego neuronu. Wówczas wypustka aksonu wydłużając się łączy się z synapsą nieswoistą kolejnego neuronu. Drugi moment to ten, w którym neuron, pobudzony podprogowo przez synapsę nieswoistą, zapamiętuje

informację czuciową w wagach synaptycznych synaps swoistych. Odpowiada to uczeniu według pierwszego etapu konsolidacji (omówionego w tej pracy). Uczenie to polega na tworzeniu asocjacji, w której następniku mamy wiele neuronów, pobudzanych przez ich synapsy nieswoiste wspólnym sygnałem nieswoistym. Drugi etap konsolidacji, wolniejszy, polega na konsolidacji asocjacji nierozłącznych powstałych podczas pierwszego etapu. Łączą się połączenia nieswoiste przy pomocy drzewka aksonalnego, jak również odłączanie są z konsolidowanej asocjacji neurony reagujące na tą samą informację swoistą.

Najistotniejszymi elementami asocjacji w sieci neuronowej są połączenia: akson (lub drzewko aksonalne) uaktywnionego neuronu z synapsą nieswoistą lub synapsami nieswoistymi innych neuronów (nie wyłączając też tego samego neuronu), a także połączenia synaps swoistych, jednego lub wielu neuronów, z drogami wiodącymi informację czuciową. Drzewka aksonalne neuronów są wyrazem konsolidacji asocjacji.

Zanik połączenia z synapsą swoistą powoduje utratę szczegółu zapamiętanej informacji, natomiast zanik połączenia z synapsą nieswoistą powoduje niedostępność zapamiętanej informacji, choć jest ona zawarta w sieci asocjacji.

W danym momencie, te neurony mogą być uaktywnione, które są wstępnie pobudzone nieswoiście. Wśród neuronów pobudzonych nieswoiście ten neuron zostaje uaktywniony, którego wagi synaps swoistych, powstałe przy utworzeniu wzorca zapamiętanej informacji, są zgodne z aktualną informacją pochodzącą od zmysłów. Z kolei uaktywniony neuron pobudza w sposób nieswoisty, poprzez presynaptyczne zakończenia aksonu lub drzewka aksonalnego, inne neurony, które z kolei poprzez synapsy swoiste oczekują na informację zgodną ze wzorcami zapamiętanymi w tych neuronach. Jeżeli pojawi się informacja wynikająca z kontekstu wskazanego przez sieć, odpowiedni neuron zostaje uaktywniony. W przypadku gdy pobudzone nieswoiście neurony nie odpowiadają zgodnością z taką informacją, występuje odrzucenie takiej informacji lub jej nabycie.

Analizując bieg wybranego impulsu po sieci, zauważa się wstępne pobudzenie przez ten impuls alternatywnych neuronów, wśród których znajduje się ten neuron, który może być zgodny z nie odebraną jeszcze informacją. Ciąg pobudzeń kolejnych neuronów (nie wyłączając poprzednich neuronów) jest wyznaczony wyborem poprzedniego neuronu, a także kontekstem wyznaczonym przez sieć.

Ważną rolę spełniają rozłączne asocjacje, które reprezentują grupy neuronów, które są pobudzane nieswoiście w sposób niezależny. Jeżeli pobudzenie nieswoiste jednej grupy wywołuje pobudzenie nieswoiste przynajmniej jednego neuronu innej grupy, to te grupy są asocjacjami nierozłącznymi. Asocjacje nierozłączne łączą się w jedną asocjację w procesie konsolidacji asocjacji. Rozłączność asocjacji jest uwidaczniana jako izolacja grup neuronów. Pojęcie szeregu asocjacji rozłącznych ma bardzo duże znaczenie w topograficznej organizacji asocjacji (spotykanej w pierwotnej korze mózgowej) lub w budowie modularnej korzy mózgowej (obecność mikrokolumn).

Bibliografia

1. Waldemar Wietrzykowski, *Kontekstowe asocjacje regularne*, IIS 2022
2. Waldemar Wietrzykowski, *Rastrowe asocjacje regularne*, IIS 2022
3. Waldemar Wietrzykowski, *Rozpoznanie w asocjacjach ponadregularnych*, IIS 2021
4. Waldemar Wietrzykowski, *Asocjacje ponadregularne*, IIS 2021
5. Waldemar Wietrzykowski, *Implementacja asocjacyjnej maszyny Turinga*, IIS 2021
6. Waldemar Wietrzykowski, *Maszyna asocjacyjna*, IIS 2021
7. Waldemar Wietrzykowski, *Asocjacje regularne*, IIS 2021

8. Waldemar Wietrzykowski, *Subiektywne badanie pamięci*, IIS 2020
9. Waldemar Wietrzykowski, *Konsolidacja elementarna*, IIS 2019
10. Waldemar Wietrzykowski, *Naturalny System Operacyjny*, IIS 2019
11. Waldemar Wietrzykowski, *Dwie maszyny*, IIS 2018
12. Waldemar Wietrzykowski, *Uczenie języka naturalnego*, IIS 2018
13. Waldemar Wietrzykowski, *Implementacja rachunku sekwentowego*, DIL 2018
14. Waldemar Wietrzykowski, *Wystarczalność rachunku sekwentowego. Rachunek tautologiczny*, DIL 2018
15. Waldemar Wietrzykowski, *Teoria wiedzy*, DIL 2018
16. Waldemar Wietrzykowski, *Teoria wiedzy teoretycznej*, DIL 2018
17. Waldemar Wietrzykowski, *Teoria wiedzy praktycznej*, DIL 2018
18. Waldemar Wietrzykowski, *Teoria świadomej maszyny*, DIL 2018
19. Waldemar Wietrzykowski, *Założenia wstępne do projektu świadomej maszyny*, DIL 2017
20. Waldemar Wietrzykowski, *Świadoma maszyna*, DIL 2017
21. Waldemar Wietrzykowski, *Świadoma reakcja maszyn*, DIL 2017
22. Waldemar Wietrzykowski, *Samoucząca się maszyna motoryczna*, DIL 2017
23. Waldemar Wietrzykowski, *Strumień świadomości*, DIL 2017
24. Waldemar Wietrzykowski, S*calanie interpretacji maszyn*, DIL 2017
25. Waldemar Wietrzykowski, *Obraz rzeczywistości z poziomu maszyny*, DIL 2017
26. Waldemar Wietrzykowski, *Świadomość wielu maszyn*, DIL 2017
27. Waldemar Wietrzykowski, *Wiele maszyn*, DIL 2017
28. Waldemar Wietrzykowski, *Znaczenie interpretacji maszyn*, DIL 2017
29. Waldemar Wietrzykowski, *Interpretacje maszyn*, DIL 2016
30. Waldemar Wietrzykowski, *Jednofunkcyjne maszyny*, DIL 2016
31. Waldemar Wietrzykowski, *Samoucząca się maszyna*, DIL 2016
32. Waldemar Wietrzykowski, *Procesor samouczący się*, DIL 2016
33. Waldemar Wietrzykowski, *Processor self-learning*, DIL 2016
