Python 3 Clone of J. Walker's Neuron Network Associative Memory for Commodore 64

Seminar Paper

Patrik Reiske[1]

Humboldt-Universität zu Berlin
Kultur-, Sozial- und Bildungswissenschaftliche Fakultät
Institut für Musikwissenschaft und Medienwissenschaft

Submitted on March 16, 2022 to Dr. Dr. Stephan Höltgen[2]

---

[1] Technische Universität Berlin, Matriculation Number 498 142

[2] Humboldt-Universität zu Berlin

Contents

1 Introduction

Humans have dreamed of creating artificial intelligent machines ever since ancient times, aiming to overcome the limits biological living beings are constrained by (Cave & Dihal, 2018). Throughout history, the advances towards achieving this goal inspired some but also scared others, fearing a machine superiority dystopia — and for some it even did both (Kubrick & Clarke, 1968; The Wachowskis, 1999). Today, with digital computer technology unleashing the fast-paced evolution of artificial intelligent machines, the call for regulations of the field is getting louder and the split between supporters and opposers becomes deeper (Toews, 2020).

Just like in any other (macro social) discussion, the question of whether and how to regulate artificial intelligence (AI), and others can probably best be answered once the issue is understood — not necessarily in its depth but in its entirety of effects — by all discussants. However, developing the technical aspect of such an understanding of AI can be a difficult task for anyone without a proper background in mathematics.

Learning — no matter what subject — works best when starting off with an easy example (MIT-TLL, no date). This is especially true for cutting edge knowledge like AI. Hence, preparing a learning resource that is referencing to a simple AI approach is a reasonable thing to do. And even though one could argue that a multitude of such resources already exists (Agora, 2022), it's the diversity of resources that promises the best learning success for any learner.

The implementation of a (by todays means) simple neural network by Walker (1987) appears to be a suitable basis for such a learning resource. It consists of a fully connected[1] single layer neural network, as figure 1 shows, as well as a command-line user interface, as seen in figure 2, and can be used to learn a pattern recognition for digits and uppercase letters; it is discussed in detail in section 2.1.

In case of the present seminar paper, Python 3 is the programming language of choice for two reasons: 1) it is an easy to read and learn language, and 2) it is the most used language in the field of AI. Even though it offers — compared to BASIC — a wide variety of features that would allow a lean reimplementation, this is not an appropriate approach when preparing such a learning resource. Under the assumption that this resource will be used to ease access to AI, and for people studying the evolution of the field, recognizability of the source code is of importance. That is why the code discussed in section 2.2 is more of a literal translation than a modern day reimplementation.

[1] Neural layers and networks are considered fully connected if every neuron of a previous layer is connected to any neuron of a subsequent layer.

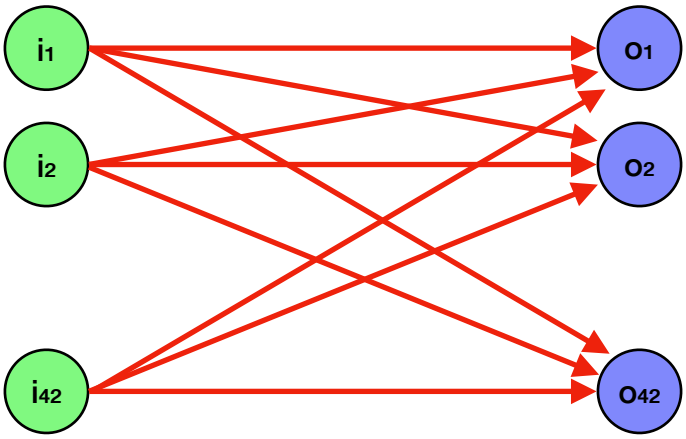

Figure 1: Simplified schematic of the fully connected single layer neural network by Walker (1987), where input neurons are green, weights interconnecting the neurons are red, and output neurons are blue.

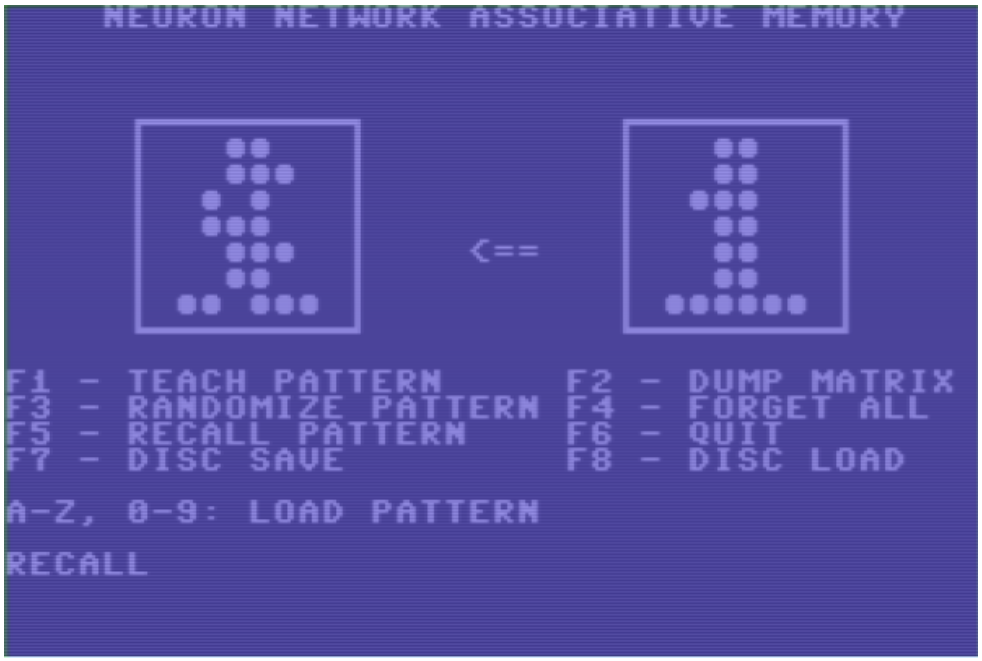

Figure 2: Command-line user interface by Walker (1987).

Also, to make using the created learning resource more playful and to honor the original implementation, the user interface seen in figure 2 is widely recreated and waiting times during execution are added — creating a somewhat nostalgic user experience.

2 Implementation

This section splits in two parts: analysis of the original software, and documentation of the reimplementation. In doing so, the main differences between the two, and difficulties encountered are highlighted.

2.1 Analysis of the J. Walker's Source Code

In this section, the entire original source code by Walker (1987) is analyzed before discussing the reimplementation in Python in the next section.

Here, it is to note that referencing to Walker (1987) is omitted in the rest of this subsection, as it should be obvious to any reader that anything references to it is the original project itself. Also, for reference on BASIC commands and any other C64 related information, visit the C64-Wiki website[1].

The operating system running on the Commodore 64 allowed text output to the display only. So it's little surprising that the screen's appearance is configured first.

```
10      rem screen configuration
20      poke 53280,13
30      poke 53281,6
40      print chr$(147);
50      open 15,8,15
```

Here, the foreground (text) color is set to be light green, and the background color to be blue. Also, the screen is cleared and a file handle is opened for later use.

```
60      rem variable declarations
70      dim f1%(42),f2%(42),m%(42,42)
80      dim v%,j,i
```

Yet before outputting the user interface for the first time and awaiting the user's input, the variables representing the neural network as well as others used during runtime are declared.

The variables f1 and f2 are arrays of 42 values each, representing the neural network's output and input layer, whereas the 42×42 shaped array m contains the weights connecting the two. The other variables, v, j, and i, are used as helpers, e.g. loop variables.

[1] https://www.c64-wiki.com

```
90      rem initialise screen
100     print chr$(147);
110     print "    neuron network associative memory"
120     print
130     for i=1 to 12: print chr$(17);: next i
140     print "f1 - teach pattern     ";
150     print "f2 - dump matrix"
160     print "f3 - randomize pattern ";
170     print "f4 - forget all"
180     print "f5 - recall pattern    ";
190     print "f6 - quit"
200     print "f7 - disc save         ";
210     print "f8 - disc load"
220     print
230     print "a-z, 0-9: load pattern"
240     r1 = 4 : c1 = 5 : gosub 600
250     r1 = 4 : c1 = 25 : gosub 600
260     gosub 750
270     gosub 860
280     gosub 970:print " ready    "
290     get a$ : if a$="" goto 290
```

Then, after clearing the display, the user interface is printed for the first time: first the title, then empty lines where the contents of variables f1, and f2 will be displayed, the list of available commands and user inputs, the patterns stored in the aforesaid variables and borders around them, and the status information.

Finally, the program waits for some (not empty) user input.

```
300     gosub 970:print "          "
310     k=asc(a$)
320     ifa$>="0"anda$<="9"thenk=k+64:goto340
330     if a$ < "a" or a$ > "z" then 500
340     gosub 970:print "fetch ";a$
350     l%=0
360     k=(k-64)*8+53248
370     poke56333,127:poke 1,peek(1)and251
380     fori=0to6:poke49408+i,peek(k+i):next
390     poke 1,peek(1) or 4:poke 56333,129
400     for i = 0 to 6
410     j% = peek(49408+i)/2
420     for k=1 to 6
430     l%=l%+1
440     f1%(l%) = -1 + (2 * (j% and 1))
450     j%=j%/2
460     next k
470     next i
480     gosub 750 : gosub 860 : goto 280
```

In case the user input was a digit or a letter, its representation as 7×6 element pattern is loaded from the ROM containing symbol representations into variable f1 and updates the information displayed on the screen. Then, the program goes back to awaiting user input.

```
490     rem dispatch function key commands
500     j%=asc(a$)-132
510     if j%=1 then gosub 1000:goto 280
520     if j%=5 then gosub 1080:goto 90
530     if j%=2 then gosub 1210:goto 280
540     if j%=6 then gosub 1680:goto 280
550     if j%=3 then gosub 1290:goto 280
560     if j%=7 then print chr$(147);:close15:end
570     if j%=4 then gosub 1800:goto 90
580     if j%=8 then gosub 1990:goto 90
590     go to 280
```

User commands other than loading patterns are handled by calling corresponding functions.

```
600     rem draw borders for fields
610     for i=0 to 1
620     v=1024+40*(r1+(i*8))+c1
630     poke v,112+(-3*i)
640     for j=1 to 8
650     poke v+j,67
660     next j
670     poke v+9,110+(15*i)
680     next i
690     for i=1 to 7
700     v=1024+40*(r1+i)+c1
710     poke v,93
720     poke v+9,93
730     next i
740     return
```

The function for drawing the borders surrounding the displays of the patterns stored in variables f1 and f2 are drawn by printing symbols to specific positions on the screen by looping their coordinates.

```
750     rem update field f2% on screen
760     l%=0
770     for i=0 to 6
780     v% = 1024+40*(i+5)+6
790     for j=2 to 7
800     l%=l%+1
810     iff1%(l%)=1thenpokev%+(8-j),81:goto830
820     poke v%+(8-j),32
830     next j
840     next i
850     return

860     rem update field f1% on screen
870     l%=0
880     for i = 0 to 6
890     v%=1024+40*(i+5)+26
900     for j=2 to 7
910     l%=l%+1
920     if f2%(l%)=1 then poke v%+(8-j),81:goto 940
930     poke v%+(8-j),32
940     next j
950     next i
960     return
```

The functions for displaying the patterns stored in variables f1 and f2 both work the same way: iterating the 42 values in the variables in sequences

of 6 (width of the pattern) values and printing whitespaces or asterisks to corresponding positions.

```
970     rem position to status area
980     print chr$(19);
982     for i=1 to 21: print chr$(17);: next i
990     return
```

Here, the status bar is cleared so that the functions handling the commands can print information there.

```
1000    rem train on pattern in f1%
1010    gosub 970:print "training"
1020    for i = 1 to 42
1030    for j = 1 to 42
1040    m%(i,j)=m%(i,j)+f1%(i)*f1%(j)
1050    next j
1060    next i
1070    return
```

The weight matrix m is trained as

$$m_{i,j} = m_{i,j} + f_{1,i} \cdot f_{1,j}$$

where i and j are indexes addressing the variables' elements that are looped such that all elements in m are updated. This essentially implements a trivial similarity measure of pairs of elements in variable f1 for a set of patterns.

```
1080    rem print part of matrix
1090    print chr$(147);
1100    for i=1 to 24
1110    for j=1 to 39
1120    ifm%(i,j)<0thenprint chr$(150);:goto1140
1130    print chr$(154);
1140    print chr$(asc("0")+abs(m%(i,j)));
1150    next j
1160    print
1170    next i
1180    print chr$(154);"press any key to continue:";
1190    get a$ : if a$="" goto 1190
1200    return
```

The weight matrix m is printed partially (as there's not enough space on the display) as colored symbols, where each symbol's PETSCII value equals the single weights' values, while the color indicates the signs — positives are blue, negatives are red.

```
1210    rem randomise 10 percent of f1%
1220    gosub 970:print "random"
1230    for i=1 to 42
1240    if rnd(0) > 0.1 then 1260
1250    f1%(i)=-f1%(i)
1260    next i
1270    gosub 750
1280    return
```

The randomize command toggles the sign of random 10% of the elements in variable f1 in order to introduce noise to the pattern.

The recall command consists of three steps:

```
1290    rem recall from pattern
1300    gosub 970:print "recall"
1310    p%=1024+40*9+19
1320    rem initially copy f1 to f2
1330    poke p%+1,asc("=")
1340    for i=1 to 42
1350    f2%(i)=f1%(i)
1360    next i
1370    gosub 860
```

1) copying the contents of variable f1 — the pattern loaded from ROM — to variable f2,

```
1380    rem f1 to f2 pass
1390    poke p%,asc("=")
1400    poke p%+2,asc(">")
1410    for j=1 to 42
1420    v%=0
1430    for i=1 to 42
1440    v%=v%+f1%(i)*m%(i,j)
1450    next i
1460    v%=sgn(v%)
1470    if v%<>0 then f2%(j)=v%
1480    next j
1490    gosub 860
```

2) calculate variable v as

$$v_j = \operatorname{sgn} \sum_{i=1}^{42} f_{1,i} m_{i,j}$$

and update the elements of variable f2 as

$$f_{2,j} = \begin{cases} v_j & \text{if } v \neq 0 \\ f_{2,j} & \text{otherwise,} \end{cases}$$

```
1500    rem f2 to f1 pass
1510    c%=0
1520    poke p%,asc("<")
1530    poke p%+2,asc("=")
1540    for i=1 to 42
1550    v%=0
1560    for j=1 to 42
1570    v%=v%+f2%(j)*m%(i,j)
1580    next j
1590    v%=sgn(v%)
1600    ifv%<>0andv%<>f1%(i)thenf1%(i)=v%:c%=1
1610    next i
1620    gosub 750
1630    if c%<>0 goto 1380
1640    poke p%,asc(" ")
1650    poke p%+1,asc(" ")
1660    poke p%+2,asc(" ")
1670    return
```

and 3) calculate

$$v_i = \operatorname{sgn} \sum_{j=1}^{42} f_{2,j} m_{i,j}$$

and update the elements of variable f1 as

$$f_{1,i} = \begin{cases} v_i & \text{if } v_i \neq 0 \text{ and } v_i \neq f_{1,i} \\ f_{1,i} & \text{otherwise.} \end{cases}$$

Also, after each of the steps, the user interface is updated.

```
1680    rem forget all - clear memory
1690    gosub 970:print "forget"
1700    for i=1 to 42
1710    f1%(i)=0
1720    f2%(i)=0
1730    for j=1 to 42
1740    m%(i,j)=0
1750    next j
1760    next i
1770    gosub 750
1780    gosub 860
1790    return
```

The forget command fills variables f1, f2, and m with zeros and updates the information displayed on the user interface.

```
1800    rem save state to disc file
1810    gosub 970:print "save"
1820    print "";
1830    input "file name: ";a$
1840    a$="@0:"+a$+",s,w"
1850    open 5,8,5,a$
1860    for i=1 to 42:print#5,f1%(i):next
1870    gosub 2240
1880    for i=1 to 42:print#5,f2%(i):next
1890    gosub 2240
1900    for i=1 to 42
1910    for j=1 to 42
1920    print#5,m%(i,j)
1930    next j
1940    gosub 2240
1950    next i
1960    close 5
1970    print "";
1980    return
```

To save the variables f1, f2, and m to disk, the user first needs to specify the file they should be stored in. This file is then opened and the variables are written to it.

```
1990    rem restore state from disc file
2000    gosub 970:print "restore"
2010    print "";
2020    input "file name: ";a$
2030    a$="@0:"+a$+",s,r"
2040    p%=asc("m")
2050    gosub 2240
2060    open 5,8,5,a$
2070    for i=1 to 42
2080    input#5,f1%(i)
2090    next i
2100    gosub 2240
2110    for i=1 to 42
2120    input#5,f2%(i)
2130    next i
2140    gosub 2240
2150    for i=1 to 42
2160    for j=1 to 42
2170    input#5,m%(i,j)
2180    next j
2190    gosub 2240
2200    next i
2210    close 5
2220    return
```

Similarly, the user needs to specify the file to load the variables f1, f2, and m from if the load command is invoked. Then, the file is opened and aforesaid variables' values are assigned to the file's contents.

```
2230    rem disc error check
2240    input#15,en,em$,et,es
2250    if en>0then print en,em$,et,es:stop
2260    return
```

This last function handles disc errors that might occur when reading or writing files storing the weight matrix.

2.2 Reimplementation in Python 3

Before discussing the reimplementation, it is to note that non-essential parts of the implementation, e.g. comments, are omitted in this section. The full source code can be found at the end of the document.

```
from pattern import GROUND_TRUTH_PATTERN
```

Since nowadays operating systems no longer have symbols' patterns easily accessible stored in ROM, the reimplementation uses a handcrafted variable containing the 7×6 element patterns of the learnable digits and characters.

Today, altering what a computers' screen displays is not as straight forward as it used to be in the days of the C64. Therefore, a function for printing the user interface to the command-line terminal is implemented.

```
def update_display(left: List[int], right: List[int], status: str,
                   middle: str = '   ', sleep: bool = True) -> None:
    os.system(CLEAR_COMMAND)
    print('\033[95;45m', end='')
```

First, whenever the user interface is updated, the command-line terminal is cleared and the foreground and background colors are configured.

```
    print('    NEURON NETWORK ASSOCIATIVE MEMORY   ')

    for _ in range(3):
        print(40*' ')

    print('     ┌──────┐          ┌──────┐         ')

    for row in range(7):
        print('     │', end='')

        for col in range(6):
            print('*' if left[6 * row + col] == 1 else ' ', end='')


        if row == 4:
            print(f'│    {middle}   │', end='')
        else:
            print('│          │', end='')

        for col in range(6):
            print('*' if right[6 * row + col] == 1 else ' ', end='')

        print('│         ')

    print('     └──────┘          └──────┘         ')
```

Then, after printing the title and some empty lines, the patterns and surrounding borders are printed.

```
    print(40*' ')

    print('F1 - TEACH PATTERN     F2 - DUMP MATRIX ')
    print('F3 - RANDOMIZE PATTERN F4 - FORGET ALL  ')
    print('F5 - RECALL PATTERN    F6 - QUIT        ')
    print('F7 - DISC SAVE         F8 - DISC LOAD   ')

    print(40*' ')

    print('A-Z, 0-9: LOAD PATTERN                  ')

    print(40*' ')

    print(status, (39-len(status))*' ')

    print(40*' ')
```

Followed by the listing of commands the user can invoke as well as the patterns of digits and characters that can be loaded.

```
    print(39*' ', '\033[00m', '\r', end='')
```

Then, the foreground and background color configuration is set back to default. This is necessary so that the user interface appears to be rectangular, even though it just covers a portion of the actual screen.

```
    if sleep:
        time.sleep(1)
```

In a last step, it pauses the program's execution for 1 second to make using the created learning resource more playful and to honor the original implementation.

The main program mainly implements a loop waiting and handling user input. Even though Python's paradigms would allow (and encourage) the implementation of individual functions for the individual commands, the reimplementation takes over the original source code's program flow to make it more recognizable.

```
def main() -> None:
    f1 = [0 for _ in range(42)]
    f2 = [0 for _ in range(42)]
    m = [[0 for _ in range(42)] for _ in range(42)]
```

Since Python does not allow easy low level memory access, the variables f1, f2, and m are not declared as arrays but lists.

```
    while True:
        update_display(f2, f1, 'READY', sleep=False)

        cmd = input('').strip().upper()
```

In the beginning of every loop cycle, the user interface is updated and the program waits for user input. Here, any user input must be written out as text input, whereas the original implementation handled key strokes directly. This reimplementation omits this functionality — even though easy to implement — to make the source code more easy to understand for readers with little programming experience.

```
        if len(cmd) == 0:
           continue
```

Empty user inputs (only pressing enter) do not invoke any function, not even an error message.

```
        elif len(cmd) == 1:  # LOAD PATTERN
            if cmd in GROUND_TRUTH_PATTERN:
                update_display(f2, f1, f'FETCH {cmd}')

                f1 = GROUND_TRUTH_PATTERN[cmd].copy()
            else:
               update_display(f2, f1, 'INVALID INPUT')
```

Inputting any digit 0–9 or character A–Z loads the symbol's pattern from the previously introduced variable that replaces the ROM. For any other standalone symbol, an error message is displayed.

```
    elif len(cmd) == 2:
        if cmd[0] == 'F':
            if cmd[1] == '1':  # TEACH PATTERN
                update_display(f2, f1, 'TRAINING')

                for i in range(42):
                    for j in range(42):
                        m[i][j] += f1[i] * f1[j]
```

Inputting F1 invokes the training process for the pattern that is stored in variable f1, implementing the equation just as the original source code does.

```
            elif cmd[1] == '2':  # DUMP MATRIX
                os.system(CLEAR_COMMAND)

                for i in range(23):
                    for j in range(40):
                        if m[i][j] < 0:
                            print('\033[31;45m', end='')
                        else:
                            print('\033[34;45m', end='')

                        print(chr(ord('0') + abs(m[i][j])), end='')

                    print()

                print('\033[95;45mPRESS ENTER TO CONTINUE:',
                      '               \033[00m\r', end='')

                input('')
```

Inputting F2 will output the same subset of the weight matrix m as the original source code does, with the same coding as used in the original source code.

```
            elif cmd[1] == '3':  # RANDOMIZE PATTERN
                update_display(f2, f1, 'RANDOM')

                for i in random.sample(list(range(42)), 4):
                    f1[i] *= -1
```

Inputting F3 randomly toggles the sign of random 10% of variable f1's elements.

```
            elif cmd[1] == '4':  # FORGET ALL
                update_display(f2, f1, 'FORGET')

                for i in range(42):
                    f1[i] = 0
                    f2[i] = 0

                    for j in range(42):
                        m[i][j] = 0
```

Inputting F4 iterates all elements in variables f1, f2, and m and sets their values to zero.

```
elif cmd[1] == '5':  # RECALL PATTERN
    update_display(f2, f1, 'RECALL', ' = ')

    for i in range(42):
        f2[i] = f1[i]

    update_display(f2, f1, 'RECALL', '==>')

    for j in range(42):
        v = 0

        for i in range(42):
            v += f1[i] * f2[j]

        v = sgn(v)

        if v != 0:
            f2[j] = v

    update_display(f2, f1, 'RECALL', '<==')

    c = 0

    for i in range(42):
        v = 0

        for j in range(42):
            v += f2[j] * m[i][j]

        v = sgn(v)

        if v != 0 and v != f1[i]:
            f1[i] = v
            c = 1
```

Inputting F5 implements the recall of patterns just as the original source code does, updating the user interface for all intermediate steps.

```
elif cmd[1] == '6':  # QUIT
    update_display(f2, f1, 'QUIT', ' = ')

    os.system(CLEAR_COMMAND)

    return
```

Inputting F6 exits the program.

```
elif cmd[1] == '7':  # DISC SAVE
    update_display(f2, f1, 'SAVE TO DISK: NEURON64.TXT')

    with open('neuron64.txt', 'wb', 1_024) as file:
        pickle.dump(m, file, 0)
```

Inputting F7 stores the contents of variables f1, f2, and m to a preset file on disk. Here, other than in the original implementation, the file cannot be specified by the user — it is not a core functionality and omitting it keeps the code leaner and easier to understand. Also, a Python package[2] for storing variables to files is used. This simplification was made as storing and loading variables is not an essential part of the program.

---

[2] Python packages provide functionality without the need of implementing those for every software project anew. Nowadays this is a common thing to do but it was not at the time of Walker's implementation.

```
            elif cmd[1] == '8':  # DISC LOAD
                update_display(f2, f1, 'LOAD FROM DISK: NEURON64.TXT')

                if os.path.isfile('neuron64.txt'):
                    with open('neuron64.txt', 'rb', 1_024) as file:
                        [m, f1, f2] = pickle.load(file)
                else:
                    update_display(f2, f1, 'FILE NEURON64.TXT MISSING')
```

Inputting F8 loads the contents of variables f1, f2, and m from the same files as stored to when inputting command F7. This is done using the same Python package (for the same reason) as discussed before.

```
            else:
               update_display(f2, f1, 'INVALID INPUT')
        else:
           update_display(f2, f1, 'INVALID INPUT')
    else:
       update_display(f2, f1, 'INVALID INPUT’)
```

In case anything else is inputted, an error message is displayed.

3 Usage

The usage of the program — both original and reimplementation — can exemplarily be described as follows:

After starting the program, the user can load patterns into the user interface's right field by inputting[1] digits or letters. Once a pattern is loaded, by inputting F1, the neural network can be trained to recognize (recall) the pattern. This may be repeated for several patterns — the learning capability lies below 5 patterns, though.

After the neural network is trained or a trained network is loaded — by inputting F8 — from disk, the network can be used to recall the pattern loaded into the user interface's right field by inputting F5. Here, the quality of the network's capability to recognize patterns becomes evident. Additionally, noise can be introduced by inputting F3 to test the robustness of the trained network.

Also, at any time during the use of the program, the network's weight matrix can be outputted by inputting F2 or stored to disk by inputting F7. Inputting F6 will quit the program.

[1] Any input in the reimplementation must be typed out and confirmed by pressing the return key.

4 Conclusion

The present seminar paper documents the reimplementation of the neural network implementation by Walker (1987), widely preserving the program flow and recreating the user interface, as can be seen in figure 3. It offers the same functionality and, in this, is limited by the same constraints in terms of learning capabilities.

The reimplementation can easily be executed on any modern computer that has Python 3 installed. The source code is lean, easy to understand, and widely preserves the original implementation's program flow. Hence, it should be a suitable learning resource for getting to know the field of AI,or a useful reference for studying the evolution of the field. Also, the skeuomorphic approach of the implementation — imitating the user experience of the original implementation — makes using the reimplementation playful and honors the original implementation.

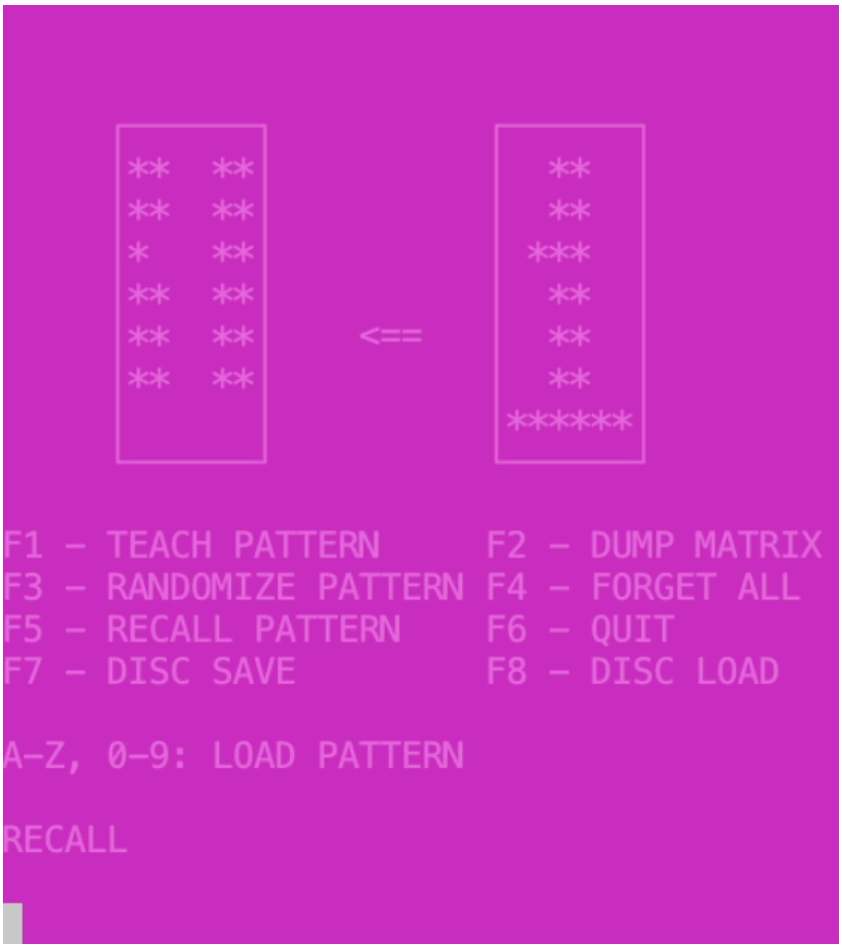

Figure 3: Recreated user interface of the reimplementation.

5 Discussion

The present seminar paper is limited in two different ways, which are briefly discussed in the following:

1. Due to the extend of such a seminar project, the used programming languages cannot be discussed other than as sketchily as done here. This might be a barrier for readers that have no or just little experience in programming.
2. Since modern day operating systems do not provide information on symbols' patterns that is as easy to access as it was at the times of the original implementation, a handcrafted replacement needed to be implemented. This deviation from the original implementation should not be hindering in serving the purpose of the project discussed in section 1, as it's not an essential part of the implemented AI approach.

Other than that, the reimplementation did not bear any bigger challenges — thanks to J. Walker's clean implementation style and good documentation of it.

Bibliography

Arora, S. K. (2022, February 23). 20 Best Machine Learning Books for Beginner & Experts in 2022. hackr.io. Retrieved on March 10, 2022, from https://hackr.io/blog/best-machine-learning-books

Cave, S., & Dihal, K. (2018). Ancient dreams of intelligent machines: 3,000 years of robots. Nature, 559, 473–475. https://doi.org/10.1038/d41586-018-05773-y

Kubrick, S., & Clarke, A. C. (1968). 2001: A Space Odyssey [Movie]. Metro-Goldwyn-Mayer.

MIT Teaching + Learning Lab [MIT-TLL]. (no date). Teaching an Interdisciplinary Subject. Retrieved on March 10, 2022, from https://tll.mit.edu/teaching-resources/how-to-teach/teaching-an-interdisciplinary-subject/

Toews, R. (2020, June 28). Here Is How The United States Should Regulate Artificial Intelligence. Forbes. https://www.forbes.com/sites/robtoews/2020/06/28/here-is-how-the-united-states-should-regulate-artificial-intelligence/?sh=3d1db60e7821

Wachowski, L., & Wachowski, L. [The Wachowskis]. (1999). The Matrix [Movie]. Warner Bros.

Walker, J. (1987, September 4). Neural Network on a Commodore 64. Fourmilab Switzerland. Retrieved on March 10, 2022, from https://www.fourmilab.ch/documents/commodore/BrainSim/

Source Code Listings

pattern.py

```
"""Neural Network on a Commodore 64 --- Python 3 Clone

This software program is a Python 3 clone of the software J. Walker
released on September 4, 1987. It is available on the website

    https://www.fourmilab.ch/documents/commodore/BrainSim/

and implements a "complete neural network associative memory pattern
recogniser".

This program was written by P. Reiske (TU Berlin) and submitted to
S. Höltgen (HU Berlin) on March 16, 2022.
"""


GROUND_TRUTH_PATTERN = {
    '0': [
        -1,  1,  1,  1,  1, -1,
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1,  1,  1,
         1, -1,  1,  1, -1,  1,
         1,  1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
        -1,  1,  1,  1,  1, -1,
    ],
    '1': [
        -1, -1,  1,  1, -1, -1,
        -1, -1,  1,  1, -1, -1,
        -1,  1,  1,  1, -1, -1,
        -1, -1,  1,  1, -1, -1,
        -1, -1,  1,  1, -1, -1,
        -1, -1,  1,  1, -1, -1,
         1,  1,  1,  1,  1,  1,
    ],
    '2': [
        -1,  1,  1,  1,  1, -1,
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1,  1,  1,
        -1, -1, -1,  1,  1, -1,
        -1, -1,  1,  1, -1, -1,
        -1,  1,  1, -1, -1, -1,
         1,  1,  1,  1,  1,  1,
    ],
    '3': [
        -1,  1,  1,  1,  1, -1,
         1, -1, -1, -1, -1,  1,
        -1, -1, -1, -1, -1,  1,
        -1, -1,  1,  1,  1, -1,
        -1, -1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
        -1,  1,  1,  1,  1, -1,
    ],
    '4': [
        -1, -1,  1,  1,  1, -1,
        -1,  1, -1, -1,  1, -1,
```

```
     1, -1, -1, -1,  1, -1,
     1,  1,  1, -1,  1,  1,
    -1, -1, -1, -1,  1, -1,
    -1, -1, -1, -1,  1, -1,
     1,  1,  1,  1,  1,  1,
],
'5': [
     1,  1,  1,  1,  1,  1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1,  1,  1,  1,  1, -1,
    -1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'6': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1, -1,
     1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'7': [
     1,  1,  1,  1,  1,  1,
    -1, -1, -1, -1, -1,  1,
    -1, -1, -1, -1,  1, -1,
    -1, -1, -1,  1, -1, -1,
    -1, -1,  1, -1, -1, -1,
    -1,  1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
],
'8': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'9': [
   -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1,  1,
    -1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'A': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1,  1,  1,  1,  1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
],
'B': [
```

```
     1,  1,  1,  1, -1, -1,
     1, -1, -1, -1,  1, -1,
     1, -1, -1, -1,  1, -1,
     1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1,  1,  1,  1,  1, -1,
],
'C': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'D': [
     1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1,  1,  1,  1,  1, -1,
],
'E': [
     1,  1,  1,  1,  1,  1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1,  1,  1,  1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1,  1,  1,  1,  1,  1,
],
'F': [
     1,  1,  1,  1,  1,  1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1,  1,  1,  1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
],
'G': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1,  1,  1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'H': [
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1,  1,  1,  1,  1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
```

```
],
'I': [
     1,  1,  1,  1,  1,  1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
     1,  1,  1,  1,  1,  1,
],
'J': [
     1,  1,  1,  1,  1,  1,
    -1, -1, -1, -1, -1,  1,
    -1, -1, -1, -1, -1,  1,
    -1, -1, -1, -1, -1,  1,
    -1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'K': [
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1,  1, -1,
     1, -1, -1,  1, -1, -1,
     1, -1,  1, -1, -1, -1,
     1,  1,  1, -1, -1, -1,
     1, -1, -1,  1, -1, -1,
     1, -1, -1, -1,  1, -1,
],
'L': [
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1,  1,  1,  1,  1,  1,
],
'M': [
     1, -1, -1, -1, -1,  1,
     1,  1, -1, -1,  1,  1,
     1, -1,  1,  1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
],
'N': [
     1, -1, -1, -1, -1,  1,
     1,  1, -1, -1, -1,  1,
     1, -1,  1, -1, -1,  1,
     1, -1,  1,  1, -1,  1,
     1, -1, -1,  1, -1,  1,
     1, -1, -1, -1,  1,  1,
     1, -1, -1, -1, -1,  1,
],
'O': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
```

```
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'P': [
     1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
     1, -1, -1, -1, -1, -1,
],
'Q': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1,  1, -1,  1,
     1, -1, -1, -1,  1,  1,
    -1,  1,  1,  1,  1, -1,
],
'R': [
     1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1,  1,  1,  1,  1, -1,
     1, -1, -1,  1, -1, -1,
     1, -1, -1, -1,  1, -1,
     1, -1, -1, -1, -1,  1,
],
'S': [
    -1,  1,  1,  1,  1, -1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1, -1,
    -1,  1,  1,  1,  1, -1,
    -1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'T': [
     1,  1,  1,  1,  1,  1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
    -1, -1,  1,  1, -1, -1,
],
'U': [
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
    -1,  1,  1,  1,  1, -1,
],
'V': [
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
     1, -1, -1, -1, -1,  1,
```

```
         1, -1, -1, -1, -1,  1,
        -1,  1, -1, -1,  1, -1,
        -1,  1, -1, -1,  1, -1,
        -1, -1,  1,  1, -1, -1,
    ],
    'W': [
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
         1, -1,  1,  1, -1,  1,
         1, -1,  1,  1, -1,  1,
        -1,  1, -1, -1,  1, -1,
    ],
    'X': [
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
        -1,  1, -1, -1,  1, -1,
        -1, -1,  1,  1, -1, -1,
        -1,  1, -1, -1,  1, -1,
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
    ],
    'Y': [
         1, -1, -1, -1, -1,  1,
         1, -1, -1, -1, -1,  1,
        -1,  1, -1, -1,  1, -1,
        -1, -1,  1,  1, -1, -1,
        -1, -1,  1,  1, -1, -1,
        -1, -1,  1,  1, -1, -1,
        -1, -1,  1,  1, -1, -1,
    ],
    'Z': [
         1,  1,  1,  1,  1,  1,
        -1, -1, -1, -1,  1, -1,
        -1, -1, -1,  1,  1, -1,
        -1, -1,  1,  1, -1, -1,
        -1,  1,  1, -1, -1, -1,
        -1,  1, -1, -1, -1, -1,
         1,  1,  1,  1,  1,  1,
    ]
}
```

main.py

```
#!/usr/bin/env python3

"""Neural Network on a Commodore 64 --- Python 3 Clone

This software program is a Python 3 clone of the software J. Walker
released on September 4, 1987. It is available on the website

    https://www.fourmilab.ch/documents/commodore/BrainSim/

and implements a "complete neural network associative memory pattern
recogniser".

This program was written by P. Reiske (TU Berlin) and submitted to
S. Höltgen (HU Berlin) on March 16, 2022.
```

```
"""


import pickle
import platform
import random
import time
import os

# NOTE Since nowadays computers do no longer have easily accessible
# read-only memory fields to get characters' patterns from, the variabel
# `GROUND_TRUTH_PATTERN` is used to replace it.

from pattern import GROUND_TRUTH_PATTERN
from typing import List


CLEAR_COMMAND = 'clear' if platform.system() != 'Windows' else 'cls'


def sgn(x: int) -> int:
    """Signum function

    The signum function returns -1 if argument `x` has an value less than
    zero, 1 if argument `x` is greater than zero, and 0 if argument `x`
    equals zero.
    """

    if type(x) is not int:
        raise TypeError('Argument x must be of type int.')

    if x < 0:
        return -1

    if x == 0:
        return 0

    return 1


def update_display(left: List[int], right: List[int], status: str,
                   middle: str = '   ', sleep: bool = True) -> None:
    """Update user interface on command-line

    Keyword arguments:

    left : List[int] -- Pattern to be displayed in the left field.

    right : List[int] -- Pattern to be displayed in the right field.

    status : str -- Status information to be displayed in the bottom line.

    middle : str -- Up to three characters to be displayed in between the
    pattern fields.

    sleep : bool -- En-/ or disable 1 second sleep after updating the
    interface.
    """

    # Clear command-line interface
    os.system(CLEAR_COMMAND)

    # Configure command-line colors
    print('\033[95;45m', end='')

    # Print title line
    print('    NEURON NETWORK ASSOCIATIVE MEMORY   ')
```

```
    # Print empty lines
    for _ in range(3):
        print(40*' ')

    # Print top border of pattern fields
    print('     ┌──────┐          ┌──────┐         ')

    # Print pattern (and the field's left and right borders)
    for row in range(7):
        # Print left border of the left pattern fields
        print('     │', end='')

        # Print symbols in left pattern field
        for col in range(6):
            print('*' if left[6 * row + col] == 1 else ' ', end='')


        # Print right border of the left pattern field, left border of the
        # right pattern field, and (optional) the characters to be
        # displayed in between those two fields.
        if row == 4:
            print(f'│    {middle}   │', end='')
        else:
            print('│          │', end='')

        # Print symbols in right pattern field
        for col in range(6):
            print('*' if right[6 * row + col] == 1 else ' ', end='')

        # Print right border of left pattern field
        print('│         ')

    # Print bottom border of the pattern fields
    print('     └──────┘          └──────┘         ')

    # Print empty line
    print(40*' ')

    # Print list of available commands
    print('F1 - TEACH PATTERN     F2 - DUMP MATRIX ')
    print('F3 - RANDOMIZE PATTERN F4 - FORGET ALL  ')
    print('F5 - RECALL PATTERN    F6 - QUIT        ')
    print('F7 - DISC SAVE         F8 - DISC LOAD   ')

    # Print empty line
    print(40*' ')

    # Print available patterns
    print('A-Z, 0-9: LOAD PATTERN                  ')

    # Print empty line
    print(40*' ')

    # Print status line
    print(status, (39-len(status))*' ')

    # Print empty line
    print(40*' ')

    # Reset command-line colors
    print(39*' ', '\033[00m', '\r', end='')

    # Sleep for 1 second (optional)
    if sleep:
        time.sleep(1)


def main() -> None:
```

```
f1 = [0 for _ in range(42)]
f2 = [0 for _ in range(42)]
m = [[0 for _ in range(42)] for _ in range(42)]

while True:
    update_display(f2, f1, 'READY', sleep=False)

    # Wait for user command
    cmd = input('').strip().upper()

    if len(cmd) == 0:
        # Skip empty command
        continue
    elif len(cmd) == 1:  # LOAD PATTERN
        if cmd in GROUND_TRUTH_PATTERN:
            update_display(f2, f1, f'FETCH {cmd}')

            f1 = GROUND_TRUTH_PATTERN[cmd].copy()
        else:
            # Invalid user input, print command-line user interface
            # with error message.
            update_display(f2, f1, 'INVALID INPUT')
    elif len(cmd) == 2:
        if cmd[0] == 'F':
            if cmd[1] == '1':  # TEACH PATTERN
                update_display(f2, f1, 'TRAINING')

                for i in range(42):
                    for j in range(42):
                        m[i][j] += f1[i] * f1[j]
            elif cmd[1] == '2':  # DUMP MATRIX
                os.system(CLEAR_COMMAND)

                for i in range(23):
                    for j in range(40):
                        if m[i][j] < 0:
                            print('\033[31;45m', end='')
                        else:
                            print('\033[34;45m', end='')

                        print(chr(ord('0') + abs(m[i][j])), end='')

                    print()

                print('\033[95;45mPRESS ENTER TO CONTINUE:',
                      '               \033[00m\r', end='')

                input('')
            elif cmd[1] == '3':  # RANDOMIZE PATTERN
                update_display(f2, f1, 'RANDOM')

                for i in random.sample(list(range(42)), 4):
                    f1[i] *= -1
            elif cmd[1] == '4':  # FORGET ALL
                update_display(f2, f1, 'FORGET')

                for i in range(42):
                    f1[i] = 0
                    f2[i] = 0

                    for j in range(42):
                        m[i][j] = 0
            elif cmd[1] == '5':  # RECALL PATTERN
                update_display(f2, f1, 'RECALL', ' = ')

                for i in range(42):
                    f2[i] = f1[i]
```

```
                    update_display(f2, f1, 'RECALL', '==>')

                    for j in range(42):
                        v = 0

                        for i in range(42):
                            v += f1[i] * f2[j]

                        v = sgn(v)

                        if v != 0:
                            f2[j] = v

                    update_display(f2, f1, 'RECALL', '<==')

                    c = 0

                    for i in range(42):
                        v = 0

                        for j in range(42):
                            v += f2[j] * m[i][j]

                        v = sgn(v)

                        if v != 0 and v != f1[i]:
                            f1[i] = v
                            c = 1
                elif cmd[1] == '6':  # QUIT
                    update_display(f2, f1, 'QUIT', ' = ')

                    os.system(CLEAR_COMMAND)

                    return
                elif cmd[1] == '7':  # DISC SAVE
                    update_display(f2, f1, 'SAVE TO DISK: NEURON64.TXT')

                    with open('neuron64.txt', 'wb', 1_024) as file:
                        pickle.dump(m, file, 0)
                elif cmd[1] == '8':  # DISC LOAD
                    update_display(f2, f1, 'LOAD FROM DISK: NEURON64.TXT')

                    if os.path.isfile('neuron64.txt'):
                        with open('neuron64.txt', 'rb', 1_024) as file:
                            [m, f1, f2] = pickle.load(file)
                    else:
                        update_display(f2, f1, 'FILE NEURON64.TXT MISSING')

                else:
                    # Invalid user input, print command-line user interface
                    # with error message.
                    update_display(f2, f1, 'INVALID INPUT')
            else:
                # Invalid user input, print command-line user interface
                # with error message.
                update_display(f2, f1, 'INVALID INPUT')
        else:
            # Invalid user input, print command-line user interface
            # with error message.
            update_display(f2, f1, 'INVALID INPUT')


if __name__ == '__main__':
    main()
```

Affidavit

I hereby declare that I have prepared the present seminar paper independently and without any other but the documented and authorized help and resources. Every passage in which the thoughts of others have been adopted (literal or in their sense) are made recognizable as such. All references are listed without any exception.

The present seminar paper has not been submitted to any other person or institution for examination before.

Patrik Reiske on March 16, 2022 in Berlin.