

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAI
Facultatea de Informatica din Iasi

LUCRARE DE LICENTA

CREARE RETELELOR DE SORTARE FOLOSIND
DEEPLARNING

DANILA MARIUS CRISTIAN

Data: *Iulie, 2018*

Coordonator
Lect. Dr. Frasinaru Cristian

Abstract

Scopul acestei lucrari este de a construi un algoritm de deep learning care sa genereze compartorii dintr-o retea de marime N . N reprezinta numarul de fire din retea. Desi exista abordari neimperative functionale, care genereaza rezultate intr-un timp multumitor, nu s-a mai incercat pana acum generarea de retele de sortare prin aceasta abordare. Rezultatul obtinut este un model antrenat care produce rezultate pentru retele de dimensiuni mici.

Cuprins

1	Introducere	3
2	Descrierea solutiei	4
2.1	Arhitectura proiectului	4
2.2	Arhitectura modelului	7
2.3	Sursa modelului	7
2.4	Antrenarea modelului	8
2.5	Prezentarea interfetei	10
3	Retele de sortare	10
3.1	Prezentare	11
3.2	Principiul 0-1	11
3.2.1	Enunt	11
3.2.2	Demonstratie	13
3.3	Generarea retelelor de sortare folosind algoritmi evolutivi	13
3.3.1	Reprezentarea functiilor booleane	13
3.3.2	Simetrii in retele de sortare	15
3.3.3	Minimizarea numarului de comparatori necesari	15
3.3.4	Evoluarea retelelor minimale	15
3.4	Retelele optimale	16
3.4.1	Prezentare	16
3.5	Metode imperative de generare a retelelor de sortare .	17
3.6	Aplicatii	18
4	Retele neuronale, prezentare	19
4.1	Arhitecturi de retele neuronale	19
4.2	Neural Turing Machine (NTM)	20
4.2.1	Descriere	20
4.2.2	Operatia de citire	21

4.2.3	Operatia de scriere	22
4.2.4	Mecanismul de adresare	23
4.2.5	Aplicatii	23
5	Tehnologii folosite	23
5.1	NTMLasagne	24
5.2	Lasagne	24
5.3	Theano	24
5.4	Python	24
5.5	Java	25
5.6	Spring Boot	25
5.7	RaphaelJS	26
5.8	Twitter Bootstrap	26
6	Concluzii	27

1 Introducere

O retea de sortare reprezinta un model abstract din informatica, compus din fire si comparatori. Scopul comparatorilor este de a interschimba 2 numere de pe firele pe care sunt plasati asa in cat la finalul parcurgerii seriei de comparatori, outputul retelei se va afla in ordine sortata. Desi sunt modele simple, retelele de sortare au aplicatii numeroase in calculul paralel si sunt folosite la sortarea seturilor de numere pe placile grafice.

In cadrul acestei lucrari mi-am propus sa dezvolt un model capabil sa genereze, pe baza unui numar N de fire primit ca input, lista de comparatori necesari pentru o retea de sortare cu N fire. In cadrul proiectului am dezvoltat si o interfata web ce permite vizualizarea cat si simularea unei sortari pe reteaua obtinuta de model. Cu aceasta ocazie scoatem in lumina si elementul de inovatie al acestei lucrari, si anume ca pana acum nu s-a mai incercat generarea de retele de sortare prin aceasta abordare. In spate, modelul foloseste o arhitectura de retele neuronale introdusa recent (in *20 Oct 2014*), optimizata pentru invatarea de algoritmi imperativi. Modelul obtinut produce rezultate pentru retele de dimensiuni mici, de 2, 3 sau 4 fire. Nu produce comparatori pentru retele care sa sorteze complet siruri de numere mai mari de 4 elemente, dar acest lucru lasa loc de extindere a proiectului pe viitor.

Motivatia din spatele alegerii proiectului de fata este data de viteza de generare ca timp a retelelor de sortare ce este semnificativ mai buna pe retele de dimensiuni mari fata de celelalte abordari.

Proiectul inglobeaza 2 modele antrenate pe seturi de antrenament diferite, bazate pe rezultatele a 2 algoritmi imperativi. Acesti 2 algoritmi imperativi au la baza modul de generare a interschimarilor din algoritmi de sortare *Bubble Sort* si *Insertion Sort*. Fiecare model incearca sa produca rezultatele similare cu unul dintre cei 2 algoritmi imperativi folositi. Cele doua modele sunt rulate de o a doua componenta a proiectului, *model manager*, intr-un *thread*¹ separat fiecare model. Scopul acestei componente este sa centralizeze rezultatele de la cele doua modele si sa comunice cu componenta de *backend* din spatele interfetei web. Intr-un final, ultima componenta a proiectului este cea de interfata web care afiseaza retelele produse de cele doua modele.

¹S-a recurs la aceasta abordare din motive de performanta

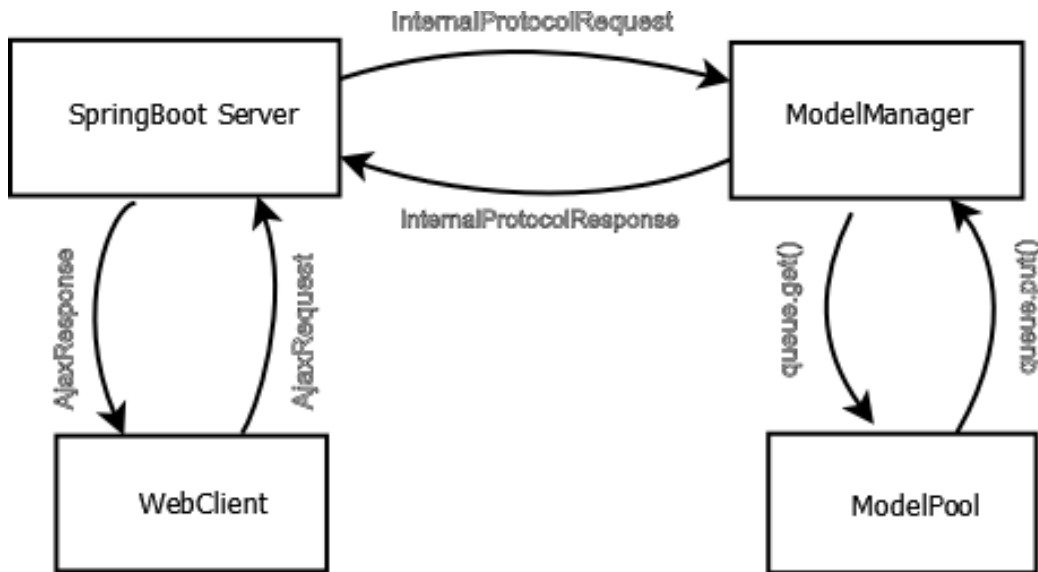
2 Descrierea solutiei

Solutia problemei consta intr-un model antrenat pe un set de date produs de un algoritm imperativ de generare a retelelor de tip bubble sorting network si insertion sorting network. Peste modelele generate avem o interfata web ce permite vizualizarea in browser a retelei generate.

2.1 Arhitectura proiectului

Proiectul complet este compus din 4 componente:

- Un server web, bazat pe SpringBoot. Rolul serverului web este sa-i livreze utilizatorului codul interfetei web a aplicatiei. De asemenea mediaza requesturile dintre requesturile unui utilizator de a primi comparatorii pentru un anumit numar de fire si *model manager*.
- O interfata in browser. Rolul interfetei in browser este acela de a se putea vizualiza efectiv reseaua generata de un modelul selectat pe un anumit numar de fire. Utilizatorul alege nuamrul de fire si modelul pe baza caruia se genereaza outputul.
- O componenta numita model manager. Managerul este compus dintr-un server TCP/IP, prin care care primeste requesturi de la serverul web. O clasa care proceseaza requestul primit de la serverul web si in functie de parametrii din request cere output de la thread pool-ul care ruleaza modelele.
- Un thread pool, fiecare model antrenat ruleaza intr-un thread separat din acest pool. Aceasta componenta incarca modelul antrenat de pe disc si ruleaza intr-un thread separat functia de predictie. Motivatia acestei abordari este legata de directiile de extindere pe viitor a proiectului.



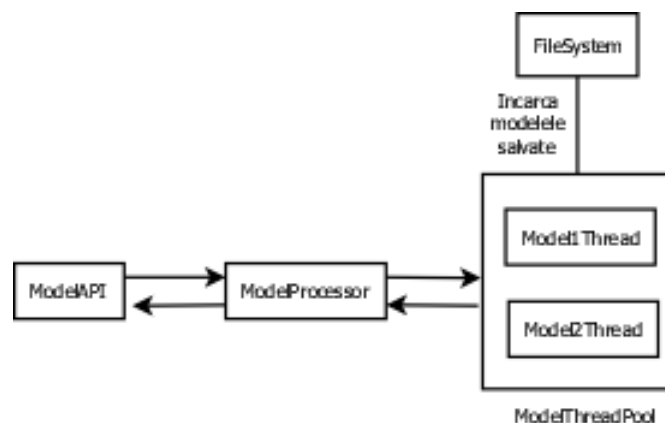
Arhitectura generara a proiectului

In diagrama de mai sus observam schematic, abstractizat, principalele componente ale proiectului si relatiile dintre ele. In continuare descriem in detaliu aceste relatii:

- **SpringBoot Server - WebUI**, etapa principala din ciclul de viata al proiectului este livrarea codului interfetei clientului, apoi, de interes este interactiunea dintre *WebUI* si server. De obicei fluxul normal este:
 1. Utilizatorul selecteaza numarul de fire, modelul antrenat folosit si asteapta producerea rezultatului. In spate, *WebUI* impacheteaza optiunile utilizatorului intr-un pachet *JSON* si trimite prin *AJAX* un request catre server.
 2. *Serverul* primeste requestul de la *WebUI* printr-un endpoint. In functie de optiunile alese, se creeaza un apel catre clasa *ManagerClient* care are rolul de a fi un adaptor intre server si *ModelManager*.
 3. *Server-ul* impacheteaza intr-un *JSON* lista de comparatori returnata de *ManagerClient* si il trimite *WebUI*-ului.
 4. *WebUI-ul* primeste *response*-ul *AJAX* cu lista de comparatori si initiaza un apel spre modulul de desenare, care creeaza reprezentarea grafica a retelei produse.

- **SpringBoot Server - ModelManager**, comunicarea dintre aceste componente este, de regula, initializata de *ManagerClient* si cronologic este structurata astfel:
 1. *ManagerClient*-ul initiaza o conexiune TCP/IP cu *ModelManager*-ul. Ii trimite acestuia un pachet codificat intr-un protocol propriu ce contine numarul de fire si modelul pe baza caruia sa se genereze comparatorii.
 2. *ModelManager*-ul accepta conexiunea de la *ManagerClient*, primeste pachetul, il decodifica si realizeaza un apel catre *ModelThreadPool*, apoi impacheteaza lista de comparatorii generata si o trimite inapoi la *ManagerClient*.
- **ModelManager - ModelPool**. De la initializare *ModelPool*-ul incarca de pe disc fiecare dintre modele si porneste un thread separat ce le ruleaza.
 1. In functie de modelul ales *ModelManager*-ul realizeaza un apel la metoda respectiva din *ModelPool*.
 2. Prin intermediul unei cozi *thread-safe*, *ModelPool*, semnalizeaza threadul cu modelul respectiv ca trebuie evaluat un nou rezultat. Rezultatul modelului este transmis inapoi prin acelasi mecanism.

In continuare dorim sa analizam mai atent structura si modul de functionare al *ModelManager*-ului.



Arhitectura interna a model managerului

În diagrama de mai sus sunt expuse principalele submodule ale managerului. Expunem succint scopul fiecărui submodule:

- *ModelApi*, reprezintă un server TCP/IP ce rulează pe portul 2018. Comunicarea cu componenta de *ModelManager* se face prin trimiterea de TCP/IP pe acest port.
- *ModelProcessor*, decodifică pachetele primite de eventualii utilizatori apelează metoda potrivită din urmaorul submodule. *ModelProcessor*-ul are și rolul de a aproxima probabilitățile din lista de comparatori generată de model.
- *ModelThreadPool*, încarcă de pe disc fiecare dintre modelele salvate și apoi creează un thread în care rulează fiecare model. Comunicarea dintre threadul principal se face prin intermediul a două cozi *thread-safe*. Threadul principal pune în coadă numărul de fire dorit, iar threadul ce rulează modelul respectiv pune în a doua coadă lista de comparatori generată.

2.2 Arhitectura modelului

Modelul este un "Neural Turing Machine". Pentru construcția modelului am folosit în spate un framework numit *ntm-lasage*, optimizat pentru crearea de modele de acest fel. Se abstractizează în acest fel detalii de implementare.

Modelul conține o memorie cu un layout de 1024x160 celule. Ca funcție de *loss* am folosit *crossentropy* binară, pentru că intern numerele ce vin ca input în model sunt reprezentate ca siruri binare.

Procesul de *backpropagation* este abstractizat de model. Actualizarea *weighturilor* din rețea este optimizată cu *rmsprop*. Datele de antrenament sunt procesate de rețea în batchuri de 1.

Structura inputului rețelei este de un vector de 9 elemente ce reprezintă codificarea în binar a numărului de fire din rețea. Outputul rețelei este un vector tridimensional cu $2 * \text{numar_comparatori}$ elemente. Elementele din vector sunt semantic grupate 2 câte 2, codificate ca un sir de biti, și reprezintă pozițiile comparatorilor din rețea.

2.3 Sursa modelului

Mai jos introducem componenta principală din cadrul proiectului:

```

def model(input_var, batch_size=1, size=8, num_units=100, memory_shape=(1024, 160)):

    l_input = InputLayer((batch_size, None, size + 1), input_var=input_var)
    _, seqlen, _ = l_input.input_var.shape

    memory = Memory(memory_shape, name='memory',
                     memory_init=lasagne.init.Constant(1e-6), learn_init=False)
    controller = GRUController(l_input, memory_shape=memory_shape,
                               num_units=num_units, num_reads=1,
                               nonlinearity=lasagne.nonlinearities.rectify,
                               name='controller')
    heads = [
        WriteHead(controller, num_shifts=3, memory_shape=memory_shape,
                  name='write', learn_init=False,
                  nonlinearity_key=lasagne.nonlinearities.rectify,
                  nonlinearity_add=lasagne.nonlinearities.rectify),
        ReadHead(controller, num_shifts=3, memory_shape=memory_shape,
                 name='read', learn_init=False,
                 nonlinearity_key=lasagne.nonlinearities.rectify)
    ]
    l_ntm = NTMLayer(l_input, memory=memory, controller=controller, heads=heads)

    l_output_reshape = ReshapeLayer(l_ntm, (-1, num_units))
    l_output_dense = DenseLayer(l_output_reshape, num_units=size + 1,
                                nonlinearity=lasagne.nonlinearities.sigmoid, \
                                name='dense')
    l_output = ReshapeLayer(l_output_dense, (batch_size, seqlen, size + 1))

    return l_output, l_ntm

```

Modelul este construit in intregime avand ca baza frameworkul *Lasagne*.

2.4 Antrenarea modelului

Modelul ce genereaza retele bubble sorter a fost antrenat pe un set de input 1000 de elemente, pe numere mai mari de atat timpul necesar finalizarii procesului de antrenament devine foarte mare. Datele din setul de antrenament au fost generate conform algoritmului:

```

def bubble_sorter(n):
    if n <= 1:
        return []

    ret_val = []
    for i in range(0, n - 1):
        ret_val.append((i, i + 1))
    ret_val += self.dummy_sorter(n - 1)
    return ret_val

```

si arata astfel pentru un set de antrenament de 5 elemente:

```

[2, [(0, 1)]]
[3, [(0, 1), (1, 2), (0, 1)]]
[4, [(0, 1), (1, 2), (2, 3), (0, 1),
      (1, 2), (0, 1)]]
[5, [(0, 1), (1, 2), (2, 3), (3, 4),
      (0, 1), (1, 2), (2, 3), (0, 1), (1, 2), (0, 1)]]
[6, [(0, 1), (1, 2), (2, 3), (3, 4),
      (4, 5), (0, 1), (1, 2), (2, 3),
      (3, 4), (0, 1), (1, 2), (2, 3), (0, 1), (1, 2), (0, 1)]]

```

Insa date de mai sus reprezinta, la nivel abstract, inputul si outputul dorit pentru retea. In proiect datele sunt reprezentate in format binar ca un sir de 9 biti, iar interschimbarile nu sunt reprezentate explicit ca tuple. Reteaua va produce, practic, o lista de siruri biti ce vor fi grupate, in ordine, 2 cate 2.

Mai precis:

1. **Setul de antrenament** este de forma (x, Y) unde x este un numar intreg, iar $Y = ((a_1, b_1), (a_2, b_2), \dots, (a_n, b_n))$, unde a_i, b_i sunt numere intregi.
2. **Outputul retelei** este de forma:

$$Out = (\underbrace{a_1, b_1}_{C_1}, \underbrace{a_2, b_2}_{C_2}, \dots, \underbrace{a_n, b_n}_{C_n})$$

. Unde a_i reprezinta firul superior pe care va fi amplasat comparatorul C_i , iar b_i reprezinta firul inferior.

Spre exemplu, pentru urmatorul input:

```
numpy.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0]])
```

reprezinta dimensiunea 3, adica retea va trebui sa genereze setul de comparatori pentru o retea de 3 elemente. Inputul de mai sus a fost testat pe o retea antrenata pe un set de date 100 de elemente, modelul este *Bubble Sorter*. Outputul produs este urmatorul:

```

[[[0.000751546365971, 0.370771723772,
0.000769892576671, 0.465849417004],
0.000637261131599,
0.000600783421602, [9.00920625725e-05,
0.12773718494, 7.82020981282e-05,
0.239040904467, 6.5095684253e-05,
0.214272843324, 6.73165195701e-05,

```

0.180481531645,	0.525120693702,
0.244833481652,	0.510196603956],
0.273710496834,	
0.454686659751,	[3.10455175941e-05,
0.496889833317],	2.33229850784e-05,
	1.51453999616e-05,
[4.46837405637e-05,	2.18735040572e-05,
3.44817119736e-05,	0.376419535208,
2.6885035585e-05,	0.237146807326,
3.21667374046e-05,	0.429571992622,
0.262205621903,	0.537611938861,
0.240104608732,	0.511490463268],
0.341346314965,	
0.501759117299,	[2.89867797738e-05,
0.507289515972],	2.23753419482e-05,
	1.35824510377e-05,
[3.48919682303e-05,	2.04069513415e-05,
2.59198402876e-05,	0.406359755172,
1.83589819237e-05,	0.238643669264,
2.46684454266e-05,	0.452255805808,
0.330040528556,	0.54490706061,
0.237285575094,	0.513243797725]]]
0.394284580217,	

– extinde in sectiune cu ‘Rezultate Obtinute’ – adauga schema retelelor obtinute, perechile input-output pentru 3 sample-uri de outputuri

Semnificatia fiecarui double din lista este probabilitatea ca bitul respectiv sa fie 1. Acuratetea probabilitatilor se sporeste cu antrenarea sporita a modelului. Intr-un final, se grupeaza 2 cate 2 elementele din lista si se decodifica din binar in decimal.

2.5 Prezentarea interfetei

– pune 1-2-3 screenshoturi – vorbeste foarte succint despre fiecare optiune pe care o are utilizatorul

3 Retele de sortare

In cadrul acestui capitol prezentam acestui capitol prezentam fundamentele teoretice necesare pentru intelegerea conceptului de *retea de sortare* cat si aplicatiile acestora in practica

3.1 Prezentare

O retea de sortare cu n inputuri este o secventa fixa de comparatii si inter-schimbari (comparatori) ce sorteaza toate inputurile de marime n . Deoarece aceeasi secventa de comparatori sorteaza orice secventa de marime n , reprezinta un algoritm de sortare independent de date. Asta inseamna ca secventa de comparatii ce se executa nu depinde de inputul primit. Datorita structurii pe care o are o retea de sortare, sunt de preferat in implementarile paralele ale algoritmilor de sortare, precum cele de pe placile grafice.

Motivati de aceste aplicatii in practice, retelele de sortare sunt un subiect important de cercetare inca din 1950. Un interes deosebit il au retelele de sortare optimale, care folosesc numarul minim posibil de comparatori. Generarea retelelor de sortare optimale reprezinta o problema grea de optimizare, investigata pentru prima data de O'Connor si Nelson pentri $4 \leq n \leq 8$. Retelele gasite de cei 2 aveau numarul minim de comparatori pentru 4, 5, 6 si 8 inputuri, dara veau nevoie de 2 comparatori in plus pentru 7 inputuri. Acest rezultata a fost imbunatatit in 1968 de Batcher care a gasit retele minimale pentru $n \leq 8$.

Pana in 2013, se cunosteau retele de sortare minimale doar pentru inputuri $n \leq 8$ (vezi Tabelul 1). In 2013, Valsalam a introdus un articol in care prezinta modalitati de generare a retelelor optimale prin algoritmi evolutivi si reuseste sa obtina noi rezultate pentru n de marime 17, 18, 19, 20, 21 si 22. In continuare, gasirea de retele optimale pentru inputuri $n > 8$ ramane o problema deschisa. Un caz particualr de interes este reprezentat de retelele descoperite de Green.

3.2 Principiul 0-1

3.2.1 Enunt

Principiul 0-1 este o modalitate de verificare a corectitudinii unei retele de sortare. Conform acestuia, o retea de sortare care sorteaza corect orice de secventa de 0 si 1, va sorta corect orice secventa de numere.

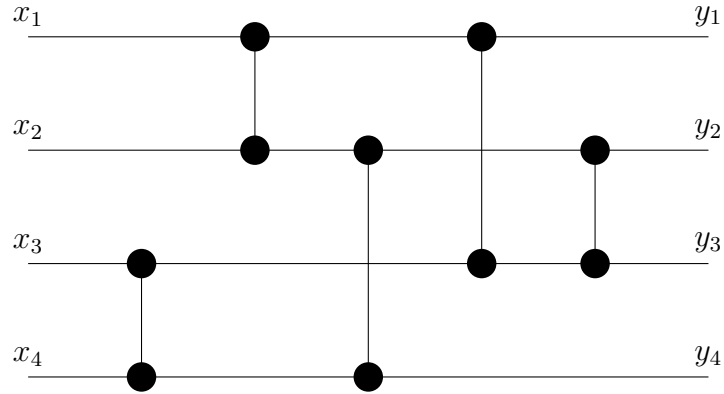


Fig. 1: O retea de sortare cu 4 inputuri. Valorile de input x_1, x_2, x_3, x_4 din partea stanga a liniilor orizontale trec printr-o secventa de operatii de comparatii si interschimbari, reprezentate prin linii verticale ce conecteaza perechi de linii orizontale. Fiecare comparator de acest fel isi sorteaza cele doua valori de input, rezultand intr-un final in liniile orizontale ce contin valorile sortate de output $y_1 \leq y_2 \leq y_3 \leq y_4$, Aceasta este o retea minimala din punct de vedere a numarului de comparatori folositi.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60

Fig. 2: Numarul minim de comparatori cunoscut pentru retele de sortare de dimensiune $n \leq 16$. Aceste retele au fost studiate intens, dar s-a dovedit ca doar rezultatele pentru $n \leq 8$ sunt minimale.

3.2.2 Demonstratie

Lemma: Presupunem f o functie monotona, crescatoare. Atunci, daca o retea mapeaza x_1, x_2, \dots, x_n la y_1, y_2, \dots, y_n , va mapa $f(x_1), f(x_2), \dots, f(x_n)$ la $f(y_1), f(y_2), \dots, f(y_n)$.

Demonstratie: Prin inductie la numarul de comparatori din retea folosind:

$$\begin{aligned}f(\min(a, b)) &= \min(f(a), f(b)) \\f(\max(a, b)) &= \max(f(a), f(b))\end{aligned}$$

Sa presupunem ca o retea nu este retea de sortare. Atunci, va mapa valorile x_1, x_2, \dots, x_n arbitrare, la y_1, y_2, \dots, y_n . Unde $y_i > y_{i+1}$ pentru $1 \leq i < n$. Presupunem $f(x) = 1$ daca si numai daca $x \geq y_i$, 0 altfel. Reteaua va mapa $f(x_1), f(x_2), \dots, f(x_n)$ la $f(y_1), \dots, f(y_i) = 1, f(y_{i+1}) = 0, \dots, f(y_n)$. Astfel, retea nu va sorta toate inputurile de tipul 0 – 1.

3.3 Generarea retelelor de sortare folosind algoritmi evolutivi

In acest subcapitol descriem succint modalitatea de generare a retelelor de sortare optimale folosind algoritmi evolutivi. Motivul pentru care am ales sa aducem in lumina si aceasta abordare este ca, problema fiind una dificila de rezolvat prin metode imperative, s-a incercat abordarea acesteia prin metode evolutive, iar rezultatele sunt surprinzatoare.

In principal, vom descrie metoda introdusa de *Valsalam* si *Miikkulainen*, (SENSO[15]).

3.3.1 Reprezentarea functiilor booleane

Folosind principiul 0-1 descris mai sus, putem exprima inputul unei retele de sortare ca variabile booleane, iar outputul retelei ca functii de aceste variabile. Astfel, problema gasirii comparatorilor pentru o retea de sortare se reduce la a numara firele ce au ca input valoarea 1 si a seta outputul firelor de output *de mai jos* la valoarea 1, iar outputul firelor ramase la 0. Deoarece aceste functii sunt implementate de comparatorii din retea, problema gasirii unei retele de sortare se poate reduce la problema gasirii unei secvente de comparatori care genereaza functiile de output ale retelei.

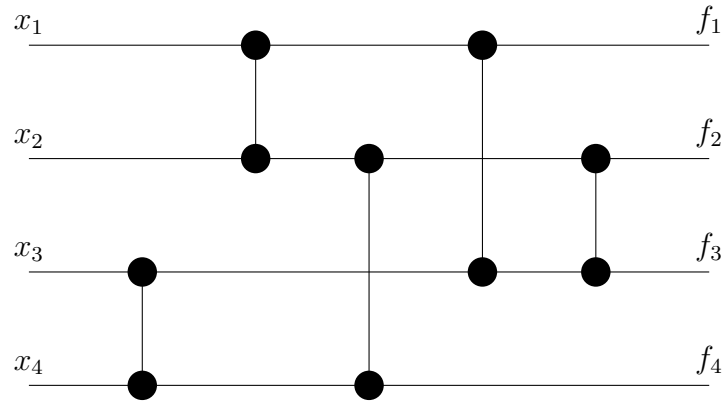


Fig. 3: Potrivit documentului introdus de Valsalam, outputurile rețelei pot fi exprimate astfel:

$$\begin{aligned}
 f_1 &= x_1 \wedge x_2 \wedge x_3 \wedge x_4, \\
 f_2 &= (x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee \dots \\
 f_3 &= (x_1 \wedge x_3) \vee (x_2 \wedge x_3) \vee \dots \\
 f_4 &= x_1 \vee x_2 \vee x_3 \vee x_4
 \end{aligned}$$

Fiecare comparator produce o conjunctie a inputurilor sale pe firul superior si o disjunctie a inputurilor sale pe firul inferior. In consecinta, functiile de output ale rețelei sunt combinatii de disjunctii si conjunctii ale inputurilor sale. Prin urmare, o rețea de sortare poate fi privita ca o secventa de comparatori care genereaza functiile de output pe baza inputurilor sale. Valsalam a folosit aceasta observatie in proiectarea algoritmului evolutiv descris mai jos.

3.3.2 Simetrii in retele de sortare

O simetrie intr-o retea de sortare este o operatie pe setul de functii de output al retelei care nu afecteaza outputul retelei. Valsalam defineste formal operatie de simetrie σ_i pentru $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$ care opereaza pe setul de functii de output din retea prin interschimbarea functiei f_i si dualei f_{n+1-i} , interschimband conjunctia si disjunctia dintre ele. Compozitia simetriilor este tot o simetrie, fiind inchisa la compozitie.

3.3.3 Minimizarea numarului de comparatori necesari

Minimizarea numarului de comparatori presupune gasirea comparatorilor ce pot fi impartiti pe mai multe fire.

3.3.4 Evoluarea retelelor minimale

Valsalam initializeaza evolutia cu o populatie de solutii produse de algoritmul greedy. Valoarea de *fitness* a fiecarei solutii este $-1 \cdot \text{numarul de comparatori}$, pentru ca imbunatatirea *fitness*-ului sa rezulte intr-un numar mai mic de comparatori generati. La fiecare noua generatie, selectia se face prin metoda *turneu* pe baza valorii de *fitness* pentru a selecta indivizii din populatie pentru reproducere. La reproducere se muteaza reteaua parinte si se creeaza o retea de *offspring* in doi pasi:

1. Un comparator este ales in mod aleator din retea si apoi se trunchiaza reteaua dupa el, eliminand comparatorii ce urmeaza dupa el.
2. Algoritmul greedy este folosit pentru a adauga noi comparatori construind o noua retea. Pentru ca algoritmul greedy alege un comparator cu cea mai mare utilitate in mod aleator, aceasta mutatie exploreaza o noua combinatie de comparatori ce ar putea fi mult mai folositori decat comparatorul parinte.

Aceasta abordare restrange spatiul de cautare la comparatorii generati de algoritmul greedy. SENSO[15] a fost rulat pe o populatie de marime 200 pentru 500 de generatii ca sa evolueze de retele de sortare optimale. La fiecare generatie, au selectate cele 100 de retele cu numarul minimal de comparatori pentru a estima modelul. Acelasi set de retele a fost copiat la urmatoarea generatie, fara modificari,

n	12	13	14	15	16	17	18	19	20	21	22	23
C	39	45	51	56	60	73	79	88	93	103	110	118
C_2	39	45	51	56	60	71	78	86	92	102	108	118

Fig. 4: Rezultate obtinute de Valsalam. Rezultatele sunt extrase din [15]. C reprezinta numarul minim de comparatori cunoscut pana atunci pentru o retea cu n inputuri, iar C_2 reprezinta numarul de comparatori obtinuti prin abordarea evolutiva. Au fost marcate in bold, retelele pentru care au fost scosi un numar de comparatori mai bun.

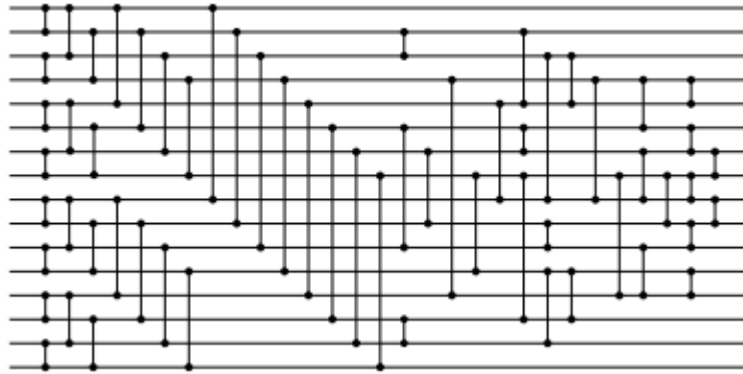


Fig. 5: Reteaua de comparatori a lui green pentru un input $n = 16$

Experimentul de mai sus a fost repetat de 20 de ori pentru fiecare varianta a algoritmului greedy si pentru fiecare input $n \leq 23$

Numarul de comparatori gasit de SENSO in corespondenta cu marimea reteli este reprezentat in figura de mai jos.

3.4 Retelele optimale

3.4.1 Prezentare

In aceasta sectiune a lucrarii dorim sa amintim despre *retelele lui Green*, intrucat folosind metoda lui *Green* de construire a retelelor obtinem numarul minimal de comparatori.

Mai jos afisam rezultatul metodei lui *Green* pentru o retea de marime 16: Acesta retea are 60 de comparatori, care reprezinta cea mai mica valoare

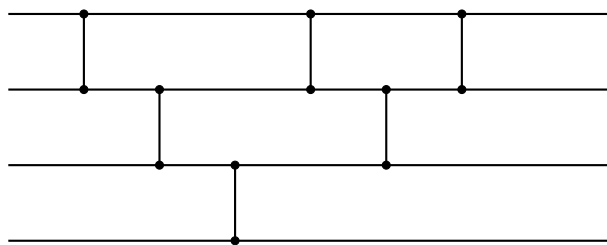


Fig. 6: Retea de sortata generata pe baza algoritmului *Bubble Sort* cu $n = 4$ inputuri.

cunoscuta pentru o retea cu 16 inputuri[16][17]. Comparatorii dintr-o astfel de retea sunt simetric aranjati de la axa orizontala pana in mijlocul retelei. Acest tip de retele au reprezentat pentru unii cercetatorii obiectiv de studiu pentru dezvoltarea algoritmilor evolutivi care genereaza retele de sortare.

3.5 Metode imperative de generare a retelelor de sortare

Exista mai multe metode imperative pentru a genera o retea de sortare. Cele mai usoare modalitati sunt de a genera o retea bazata pe algoritmii de sortare bubble sort si insertion sort. In cazul primei optiuni se adauga un comparator intre firele 0 si 1, 1 si 2, ..., $n - 1$ si n . Se reia procesul de creare a comparatorilor inserand un comparator intre 0 si 1, ..., $n - 2$ si $n - 1$. Se repeta aceasta procedura de creare a comparatorilor pana la pasul in care inseram un singur comparator intre firele 0 si 1.

In cazul unei insertion sorting network, o retea bazata pe insertion sort procesul este similar, singura diferenta este ca bucele de creare a comparatorilor sunt in ordine inversa fata de metoda precedenta. Astfel, prima data se creeaza un comparator intre firul 0 si 1. Se creeaza un comparator intre 0 si 1, 2. Pana cand se ajunge la pasul in care se genereaza un comparator intre 0 si 1, ..., $n - 1$ si n .

Un exemplu de bubble sorting network:

Un exemplu de insertion sorting network:

Algoritmul imperativ pe care l-am folosit pentru a genera un bubble sorting network este:

```
def bubble_sorter(n):
    if n <= 1:
        return []
```

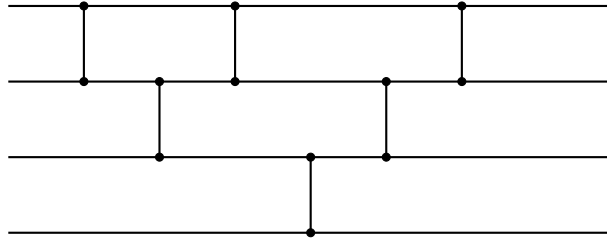


Fig. 7: Retea de sortare generata pe baza algoritmului *Insertion Sort* pentru $n = 4$ inputuri.

```
ret_val = []
for i in range(0, n - 1):
    ret_val.append((i, i + 1))
ret_val += self.dummy_sorter(n - 1)
return ret_val
```

Pentru un numar de fire dat, n , generam lista de comparatori, adica conexiunile dintre firele din retea.

Aceste arhitecturi de retele de sortare nu sunt optimale si vor sorta un sir de numere ca input in timp $O(n^2)$. Retelele optimale si cele mai des folosite in practica sunt bazate pe algoritmul bitonic sort, acestea ating complexitate de $O(n \log^2(n))$. Insa motivul pentru care am prezentat bubble sorting network si insertion sorting network este ca bazandu-ne pe acesti algoritmi am generat datele de antrenament pentru modelul dezvoltat. Motivatia ar fi ca, cel putin din experimentele realizate in cadrul acestui proiect, este mai usor de antrenat un model care sa produca date relevante atunci cand primeste date de antrenament generate pe baza acestor algoritmi.

3.6 Aplicatii

Cea mai la indemana aplicatie in practica a retelelor de sortare o reprezinta sortarea pe placile grafice. Placile grafice dispozitive orientate puternic spre paralelizarea instructiunilor, pe arhitectura *single-instruction, multiple data* (SIMD). Si avand in vedere si ca placile grafice depasesc in performanta CPU-ul cand vine vorba de algoritmi cu limita de memorie si timp de calcul, retelele de sortare devin o optiune eficienta.

4 Retele neuronale, prezentare

Retelele neuronale reprezinta o ramura a inteligentei artificiale. Sunt modele abstracte ce incearca sa simuleze creierul uman in modul de interactionare cu natura. O retea neuronală este compusa din neuroni artificiali. Neuronii artificiali din retea sunt dispusi pe mai multe straturi. Din exterior le putem vedea ca pe un black box conectat la un strat de neuroni de input, responsabil de preluarea inputului si la un strat de neuroni de output, responsabil pentru a genera outputul. In functie de neuronii din stratul de output care se vor activa se decodifica rezultatul produs de retea.

4.1 Arhitecturi de retele neuronale

Retelele neuronale convolutionale sunt modele simple, proiectate sa primeasca un set de input fix si sa produca un set output de marime fixa. In practica, acest tip de retele sunt ideale pentru computer vision. Sunt mult mai eficiente din punct de vedere al resurselor consumate la sarcini precum recunoasterea de obiecte in imagini.

Retelele neuronale recurente se deosebesc de cele convolutionale prin posibilitatea de a primi un set de input variabil si a produce un set de output variabil. In practica se folosesc la procesarea limbajului natural. Acest tip de retele neuronale sunt ideale si pentru rezolvarea problemei de generare a retelelor de sortare, intrucat putem codifica numarul de fire din retea ca o secventa de lungime variabila, iar rezultatul, respectiv setul de comparatori pentru numarul de fire dat poate varia. Pe langa aceste considerente, acest tip de retele contin si o memorie atasata.

Un caz particular de retele neuronale recurente sunt *Neural Turing Machines*. Acestea fiind optimizate in a invata algoritmi imperativi simpli. Un exemplu ar fi problema sortarii, rezolvata in mod traditional cu un algoritm imperativ poate fi rezolvata folosind acest tip de retele. Pentru aceasta, retea va primi ca input tuple de forma (X, Y) . Unde X reprezinta un sir de numere aflate in ordine aleatoare, iar Y reprezinta varianta sortata a sirului X cu un algoritm imperativ de sortare. Odata antrenata pe un set suficient de mare de date de acest fel, modelul va invata practic sa aplice algoritmul de sortare folosit pe componenta Y .

Motivul prezentarii scenariului de mai sus este ca hiperparametrii si structura straturilor de neuroni dintr-o retea neuronală folosita la sortarea de numere sunt similari in proportie foarte mare cu cei din retea proiectata sa

genereze rețele de sortare.

4.2 Neural Turing Machine (NTM)

Retelele NTM fac partea din clasa rețelelor neuronale recurente și sunt introduse relativ recent. Au fost introduse într-un articol publicat de Alex Graves în 2014. Acest model de rețea neuronală combină capacitatea de pattern matching a rețelelor neuronale cu capacitatea algoritmică a calculatoarelor programabile. Mai exact, putem folosi acest model de rețea pentru a învăța algoritmi simpli. În paperul introdus de Graves sunt aduse rezultate pentru algoritmi de copiere a unui sir sau sortare. Aceste aspecte ne inspiră o probabilitate mare că o rețea bazată pe acest model poate produce rezultate multumitoare pentru problema abordată.

4.2.1 Descriere

Neural Turing Machines sunt inspirate din două lumi: Machine Learning și Teoria Computabilității. Prima le permite calculatoarelor să realizeze taskuri care, deși pentru oameni sunt ușoare, se credeau grele pentru calculatoare, precum computer vision. A doua definește formal lucrurile de care este capabil un calculator.

Legătura dintre inteligența artificială și teoria computației în 1940. Modelul introdus de el, Mașina Turing, e un model computațional clasic care operează pe o bandă de memorie infinită și un cap care poate să scrie sau să citească. Pe baza acestor rezultate se inspiră și Neural Turing Machines.

Un exemplu de NTM desfasurat în timp:

Componenta de controller din cadrul unui Neural Turing Machine este o rețea neuronală care pune la dispoziție reprezentarea internă a inputului care este folosit de capetele de scriere și citire ca să poată interacționa cu memoria. Reprezentarea aceasta nu este identică cu cea stocată în memorie. Tipul de controller pe care îl alegem reprezintă cea mai semnificativă alegere arhitecturală. Controllerul poate fi fie o rețea feed-forward, fie o rețea recurentă.

Capetele de scriere și citire sunt singurele componente din cadrul unui Neural Turing Machine care interacționează în mod direct cu memoria. Într-un, comportamentul fiecărui dintre capete este controlat de vectorul de

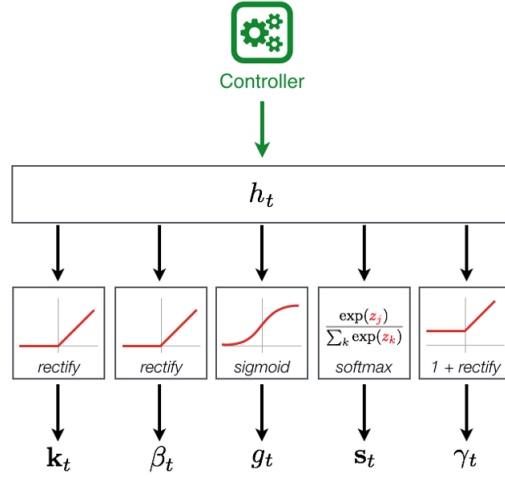


Fig. 9: Arhitectura capului de scriere/citire a unui *Neural Turing Machine*. Sursa: <https://medium.com/snips-ai/ntm-lasagne-a-library-for-neural-turing-machines-in-lasagne-2cdce6837315>

$$r_t \leftarrow \sum_t w_t(i) M_t(i)$$

4.2.3 Operatia de scriere

Fiecare operatie de scriere din aceasta arhitecra este impartita in doua parti:

1. o operatie de stergere (*erase*)
2. o operatie de adaugare (*add*)

Fiind dat un vector de weighturi w_t generat de capul de scriere la timpul t , si un vector de stergere (*erase vector*) e_t cu elemente in $(0, 1)$, vectorii de memorie $M_{t-1}(i)$ sunt modificati astfel:

$$\tilde{M}_t(i) \leftarrow M_{t-1}(i)[1 - w_t(i)e_t]$$

unde 1 reprezinta un vector linie cu elemente 1. Elementele de la o locatie de memorie sunt setate la 0 daca atat *weight*-urile de la acea locatie, cat si

elementul de trebuie sters sunt 1. Altfel, daca oricare dintre *weight*-uri sau elementul ce trebuie sters sunt 0, memoria este lasata neschimbata.

Capul de citire si un vector a_t de lungime M ce este adaugat in memorie dupa pasul de stergere:

$$M_t(i) \leftarrow \tilde{M}_t(i) + w_t(i)a_t$$

4.2.4 Mecanismul de adresare

In articolul introdus de *Graves*, adresarea se realizeaza prin doua modalitati:

1. **In functie de continut.**
2. **In functie de locatie.**

Vom descrie doar prima metoda

In *adresarea pe baza de continut* fiecare cap de citire sau scriere va produce un vector k_t de lungime M care este comparat cu fiecare vector $M_t(i)$ printr-o masura de similaritate $K[\cdot, \cdot]$. Sistemul de adresare pe baza continutului produce un vector normalizat w_t^e :

$$w_t^e = \frac{\exp(\beta_t K[k_t, M_t(i)])}{\sum_j \exp(\beta_t K[k_t, M_t(j)])}$$

In articolul lui *Graves* K este definit:

$$K[u, v] = \frac{u \cdot v}{||u|| \cdot ||v||}$$

4.2.5 Aplicatii

Neural Turing Machines inca se afla sub cercetare activa. Utilitatea acestei arhitecturi de retele neuronale este data posibilitatea unui model sa invete un *program*.

5 Tehnologii folosite

In cadrul proiectului am facut uz de o serie de frameworkuri si biblioteci pentru a fi scutiti de unele detalii de implementare. In aceasta sectiune vom prezenta pe scurt tehnologiile principale folosite, motivatia alegerii unei tehnologii, dar si unde a fost utilizata.

5.1 NTMLasagne

NTMLasagne este frameworkul pe care l-am folosit in proiect ca sa creem modelul. Pe scurt, este un wrapper peste Lasagne ce ne permite sa creem "Neural Turing Machines" intr-un mod simplificat, abstractizand detalii de implementare. Ne sunt puse module pentru stratul NTM (en. NTMLayer) unde toate componentele precum capetele de scriere, controllerul si memoria pot fi personalizate. In cadrul acestui proiect am ales sa nu personalizam niciuna dintre componente. In schimb, le-am folosit pe cele default oferite de framework.

Motivatia din spatele alegerii acestui framework este, chiar daca documentatia este relativ putina, comunitatea puternica din spatele proiectului si faptul ca procesul de dezvoltare este activ.

5.2 Lasagne

Lasagne este o biblioteca minimalista folosita pentru construirea de retele neuronale. Permite construire de retele feed-forward, cat si de retele convolutive, ofera implementari pentru metode de optimizare folosite des (i.e. ADAM si RMSProp). De asemenea, modele construite in Lasagne pot fi antrenate fie pe CPU sau GPU pentru ca biblioteca foloseste Theano in spate.

5.3 Theano

Theano este o biblioteca de python ce permite definirea si optimizarea de expresii matematice ce contin array-uri multidimensionale intr-un mod eficient. Chiar daca in momentul de fata Tensorflow este alegerea facuta de majoritatea proiectelor, theano este folosit de cele 2 biblioteci specificate anterior si este in general mai flexibil.

5.4 Python

Python este un limbaj de scripting, multi paradigma. Motivul pentru care am ales sa implementam modelul in python este sintaxa usoara, suportul mare pentru biblioteci si frameworkuri de *deeplearning*, atat ca documentatie, cat si ca implementari efective.

De asemenea componenta de *model manager* care ruleaza intr-un thread fiecare model este scrisa in tot *python*.

In cadrul proiectului am folosit mai multe module pe care le mentionam succint in aceasta sectiune:

1. **numpy** pentru reprezentarea inputului unui model, cat si pentru dependintele de mai sus (*Lasagne*, *NTM-Lasagne*)
2. **cPickle** pentru a serializa un model antrenat.
3. **json** pentru a impacheta mesajele trimise la serverul Java.

5.5 Java

Java este un limbaj de programare orientat-obiect, puternic tipizat, conceput de catre James Gosling la Sun Microsystems (acum filial Oracle) la inceputul anilor 90, fiind lansat in 1995. Cele mai multe aplicatii distribuite sunt scrise in Java, iar noile evoluii tehnologice permit utilizarea sa si pe dispozitive mobile gen telefon, agenda electronic, palmtop etc. In felul acesta se creeaza o platforma unica, la nivelul programatorului, deasupra unui mediu eterogen extrem de diversificat. Acesta este utilizat in prezent cu succes si pentru programarea aplicatiilor destinate intranet-urilor

In proiect, Java a fost folosit la crearea *back-endului* pentru interfata web. Motivatia alegerii este experienta puternica anterioara a autorului cu acest limbaj de programare.

5.6 Spring Boot

In cadrul proiectului am pus la dispozitie si o interfata web in care se afiseaza reprezentarea vizuala a retelei generate de modelul ales de utilizator. Back-endul acestei interfete este scris in Java si are in spate Spring Boot.

Pe scurt Spring Boot este frameworkul web pe care l-am folosit pentru a scrie serverul proiectului². Ni se pune la dispozitie posibilitatea de a scrie relativ usor o aplicatie web pe arhitectura *MVC*³.

Motivul pentru care s-a ales Spring Boot in locul unui alt framework web de Java este documentatia vasta si experienta anterioara a autorului cu aceasta tehnologie.

²Serverul ruleaza pe *tomcat 8*

³Design Patternul Model-View-Controller

5.7 RaphaelJS

RaphaelJS reprezinta biblioteca de JavaScript folosita pentru desenarea in browser a retelei generate de modelul ales de utilizator. RaphaelJS este un wrapper peste WebGL cu un API relativ usor de folosit.

5.8 Twitter Bootstrap

Twitter Bootstrap este un framework web pentru componenta de frontend. Se pun la dispozitie mai multe clase CSS pentru crearea de componente vizuale precum butoane, bare de meniu, cat si un mecanism specific de de pozitionare a continutului.

Motivul pentru care am introdus si aceasta dependinta in stiva de tehnologii folosita in proiect si nu am creat toata interfata in RaphaelJS este performanta. Desenarea intregului design si implementarea si mecanismilor din spate (precum gestionarea apasarilor de buton) ar fi fost remarcabil mai lenta. Ar fi fost o scadere redundanta in performanta.

6 Concluzii

În momentul de față, cele două modele antrenate, nu produc rezultate multumitoare, nu sunt capabile să genereze comparatori pentru rețele de dimensiuni mari. Optimizarea acestor modele pentru inputuri $N > 4$ poate reprezenta o direcție de viitor pentru proiect. Acest lucru ar putea fi realizat printr-o metodă introdusă relativ recent (*Mai 2018*)[18].

Readucem în atenție că această lucrare conține un element de noutate, întrucât această abordare de a genera rețele de sortare folosind *deeplearning* nu a mai fost atinsă până în acest moment.

Bibliografie

- [1] <https://github.com/primaryobjects/nnsorting>
- [2] [https : //github.com/drforester/Sequence_to_Sequence_Sorting](https://github.com/drforester/Sequence_to_Sequence_Sorting)
- [3] <https://github.com/aditya-prasad/dnnet>
- [4] <https://adventuresinevolutionblog.wordpress.com/2016/09/17/minimal-sorting-networks/>
- [5] <http://aclweb.org/anthology/I17-3017>
- [6] <http://psycnet.apa.org/record/1999-02657-007> – Seven times seven is about fifty
- [7] <https://www.sciencedirect.com/science/article/pii/S0020025512007670>
- [8] <https://github.com/apache/incubator-mxnet/tree/master/example/bi-lstm-sort> – sort numbers using lstm architecture
- [9] [https : //rylanschaeffer.github.io/content/research/neural_turing_machine/main.html](https://rylanschaeffer.github.io/content/research/neural_turing_machine/main.html)
- [10] <https://github.com/carpedm20/NTM-tensorflow>
- [11] <http://www.robots.ox.ac.uk/~tvlg/publications/talks/NeuralTuringMachines.pdf>
- [12] <https://medium.com/snips-ai/ntm-lasagne-a-library-for-neural-turing-machines-in-lasagne-2cdce6837315>
- [13] <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm>
- [14] <http://www.cs.tau.ac.il/~zwick/Adv-Alg-2015/Sorting-Networks.pdf>
- [15] Valsalam, Vinod K. and Miikkulainen, Risto Using Symmetry and Evolutionary Search to Minimize Sorting Networks *J. Mach. Learn. Res.*, 14(1): 303–331, feb 2013
- [16] D. E. Knuth. Art of Computer Programming: Sorting and Searching, volumul 3, capitolul 5, paginile 219229. Addison-Wesley Professional, 2 edition, April 1998

- [17] M. W. Green. Some improvements in non-adaptive sorting algorithms. In Proceedings of the Sixth Annual Princeton Conference on Information Sciences and Systems, paginile 387391, 1972.
- [18] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, Pierre Baldi Solving the Rubik's Cube Without Human Knowledge, 18 Mai 2018
- [19] Alex Graves, Greg Wayne, Ivo Danihelka Neural Turing Machines, 20 Oct 2014
- [20] Peter Kipfer, Rdiger Westermann Improved GPU Sorting, *https : //developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter46.html*, Aprilie 2005
- [21] Alex Graves, Greg Wayne, Ivo Danihelka Can neural nets learn a program? <http://www.robots.ox.ac.uk/tvg/publications/talks/Neural-TuringMachines.pdf>