Klara Västra Teoretiska Gymnasium

Gymnasiearbete

John Möller

Neurala nätverk och 3-SAT

Handledare:
Gunnar Karlsson

Medbedömare
Petter Restadh

23 Januari 2020

Abstract

Many important problems such as protein folding, circuit design and breaking RSA encryption all fall under the complexity class NP. For many of these problems, the finding of a polynomial time algorithm would greatly be appreciated. Better circuit design would improve computing speed and faster protein folding could lead to a cure for cancer. But for problems such breaking RSA encryption, a polynomial time algorithm would not be as much appreciated as it would cause online banking to easily be hacked. If a polynomial time algorithm would be found to the 3-SAT problem it would also apply to all problems in the complexity class NP. A polynomial time algorithm to 3-SAT, or simply a proof of the existence or non-existence of one is yet to be found. Recently, advances in machine self-learning with the use of neural networks has lately shown much succes. Self taught AIs can now outperform humans in many advanced games such as DOTA and Go. In this paper, neural networks are trained to solve 3-SAT problems with fixed variable length in order to see what the results sugests regarding the finding of a polynomial time algorithm.

Innehåll

1 Inledning

Många problem som datorer försöker lösa kan delas upp i två klasser: P och NP. P kan anses som problem som kan lösas 'snabbt' ('snabbt' defineras i teori). Problem som faller under denna kategori är exempelvis multiplikation och att sortera namn i alfabetsordning (Cook 1971). Om man definerar P som problem som kan lösas 'snabbt' så är NP problem som man kan kontrolera om ett svar är korrrekt 'snabbt'. Ett exempel på detta är att primtals faktorisera. Om jag skulle fråga dig, läsaren att hitta delare till talet 2231 med endast penna och papper så skulle det nog tag ganska lång tid. Den enda metod du skulle ha förutom att använda delbarhetstester är att kolla varenda nummer lägre än 2231 och testa att dela. Men om jag istället bad dig att kolla om svaret är 23 och 97 så skulle det bli enklare. Då kan du helt enkelt ställa upp 23 gånger 97 och se om det blir 2231. Problemet anses vara i NP för att svaret kunde kontroleras 'snabbt'.

I klassen NP så finns det en delmängd problem som kallas NP-fullständiga. NP-fullständiga problem är problem vars lösningsalgoritm kan användas för varenda problem i hela klassen NP. D.v.s det räcker att du hittar en snabblösning för ett problem som är NP-komplett och då kan du lösa varenda problem i NP snabbt (ibid.). Ett problem som faller under klassen NP-fullständiga problem är 3-SAT, som är enkel att formulera matematisk då det inte primärt grundar sig på ett problem i verkligheten.

1.1 Syfte och Frågeställning

I detta arbete beskrivs skapandet av data som innehåller lösta 3-SAT problem som kan användas för träningen av ett neuralt nätverk. Sedan beskrivs konstruktionen av modellen för det neurala nätverket samt träningen av det nätverket med hjälp av den skapade datan. Syftet med denna uppgift är att utforska frågan om hur bra neurala nätverk kan lösa 3-SAT problem för att nyansera frågan om en algoritm som löser 3-SAT i polynomisk tid är möjlig. Texten har också i syfte att undervisa om hur python fungerar i samband av matematiska undersökningar.

2 Teori

2.1 Polynomisk tid

I inledningen så förklarades P som problem som kan lösas 'snabbt'. Det som menades med snabbt var att algoritmen löser den i polynomisk tid. Polynomisk tid innebär att antalet steg som krävs för att lösa problemet är en polynomisk funktion av storleken. T.ex om storleken på ett visst problem betecknas med variabeln x och algoritmen som löser den utför det i $x^2 + 5x + 3$ steg så innebär det att algoritmen är i P och därmed är problemet dessutom i P. NP är därmed de problem där ett svar kan kontroleras i polynomisk tid. Vad som räknas som steg och vad som bör mätas i storleken kommer ej behandlas i denna text, men en rigorös definition kan läsas i boken 'A first Course in Logic' av Shawn **hedmanlogic**.

2.2 Propositionell logik

I gymnasial matematik behandlar man främst oändliga mängder såsom de naturliga, reella och komplexa talen. Men mycket kan beskrivas med de minsta mängderna . Dock så vill man forfarande att det ska vara användbart. Den tomma mängden och en mängd med ett element är sannerligen små med de är inte särskillt användbara; en variabel kan ej ingå i en tom mängd och en variabel i en mängd med ett element är ju redan löst. Så vi kommer betrakta matematik som behandlar variabler som ingår i en mängd med två element. Sådan matematik kallas Propositionell logik.

Definition 2.1. *Ett element i mängden* $\mathbb{B} = \{0, 1\}$ *kallas för ett booleskt värde.*

Likt de tidigare mängderna, kan 0 och 1 symbolisera kvantiteter, men styrkan med booleska variabler är att 0 och 1 kan symbolisera saker som är i motsats till varandra. Detta kan vara falskt och sant, av och på, ner och upp o.s.v. När propositionell logik diskuteras i fortsättningen av denna text kommer orden 'falskt' och 'sant' användas syonymt med siffrorna 0 respektive 1 .

Med mängderna som de naturliga, reella och komplexa talen fanns det en tydlig betydelse av operationerna plus,minus,gånger och division. Detta är dock inte lika tydligt med booleska värden. Istället defineras följande två operationer.

Definition 2.2. *Om* $A, B \in \mathbb{B}$ *så är* $A \wedge B$ *lika med* 1 *om och endast om* A *och* B *är båda lika med* 1*, annars är det lika med* 0*.* $A \wedge B$ *uttalas 'A och B'.*

Definition 2.3. *Om* $A, B \in \mathbb{B}$ *så är* $A \vee B$ *lika med* 0 *om och endast om både* A *och* B *är lika med* 0*, annars är det lika med* 1*.* $A \vee B$ *uttalas 'A eller B'.*

Exempel 2.4. *Låt påståendet 'Himlen är ibland blå' betecknas med variabeln* A*, låt påståendet 'Himlen är ibland svart' betecknas med variabeln* B *och låt påståendet 'Himlen är ibland grön' betecknas med variabeln* C*. Om vi instämmer på att himlen aldrig är grön så har vi:*

$$A = 1, B = 1, C = 0.$$

'Himlen är ibland blå och himlen är ibland svart' är sant. Detta kan uttryckas som:

$$A \wedge B = 1.$$

'Himlen är ibland blå och himlen är ibland grön' är falskt. Detta kan uttryckas som:

$$A \wedge C = 0.$$

'Himlen är ibland blå eller himlen är ibland grön' är sant. Detta kan uttryckas som:

$$A \vee C = 1.$$

Som man ser i exempel 2.4 så är definitionen för 'och' ganska synonymt till hur man använder ordet i vardagligt språk. Sista påståendet som använde 'eller' översätts inte lika tydligt i vissa vardagliga uttryck, då det kan uppfattas som en fråga. Men i matematiska sammanhang är det alltid ett påstående.

Definition 2.5. *Negationen av en boolesk variabel A kan skrivas som $\overline{A}$. $\overline{A} = 0$ då $A = 1$ och $\overline{A} = 1$ då $A = 0$.*

Det finns många fler operationer som kan defineras, men alla dessa kan defineras att kombinera operationerna 'och', 'eller' och negation.

Definition 2.6. *En formel som endast innehåller operationen 'eller' samt negation d.v.s på formen $a_1 \vee a_2 \vee a_3 \ldots \overline{a_n} \vee \overline{a_{n+1}} \vee \overline{a_{n+2}} \ldots$ upp till a_n, kallas för en disjunktion.*

Exempel 2.7. *Följande formel är ett exempel på en disjunktion:*

$$a_1 \vee \overline{a_2} \vee \overline{a_3} \vee \overline{a_1}.$$

Definition 2.8. *En formel som endast innehåller operationen 'och' samt negation d.v.s på formen $a_1 \vee a_2 \vee a_3 \ldots a_n + a_{n+1}$ upp till a_n kallas för en konjuktion.*

2.3 3-SAT

Definition 2.9. *3-SAT är problemet om det finns en sann lösning till en formel som består av konjuktioner av disjuktioner, där varje disjunktion innehåller högst tre variabler.*

Exempel 2.10. *Formeln*

$$(a_1 \vee a_2 \vee a_3) \wedge (\overline{a_1} \vee a_2 \vee \overline{a_3}).$$

är ett exempel på ett 3-SAT problem som är satisferbar, då den har lösningen $a_1, a_2, a_3 = 1$.

2.4 Linjär algebra

2.4.1 Vektorer

En vektor är en lista av olika värden. Dimensionen av en vektor syftar på antalet element den har. Följande exempel visar en tredimensionell vektor.

Exempel 2.11. *En tredimensionell vektor*

$$\begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

Addition av vektorer defineras endast när båda vektorerna har samma dimension, och sker på följande vis.

Definition 2.12. *Addition av vektorer*

$$\begin{bmatrix} a_1 \\ a_2 \\ \ldots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \ldots \\ a_n + b_n \end{bmatrix}$$

Exempel 2.13.

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$$

Vektorer kan också multipliceras med ett reellt tal. Då multipliceras varje element i vektorn. Detta kallas skalärmultiplikation.

Definition 2.14. *Skalärmultiplikation*

$$k \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{bmatrix} = \begin{bmatrix} ka_1 \\ ka_2 \\ \dots \ ka_n \end{bmatrix}$$

Exempel 2.15.

$$3 \begin{bmatrix} 4 \\ 5 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \cdot 4 \\ 3 \cdot 5 \\ 3 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 \\ 15 \\ 6 \end{bmatrix}$$

2.4.2 Matriser

I denna sektion så kommer endast kvadratiska matriser behandlas samt defintionen för addition och multipliktation kommer förklaras på ett icke rigoröst sätt med exempel. Se 'Linjär Algebra' av Skjelnes (2015) för exakta definitioner.

Matris addition defineras som hur man förväntar sig. Likt vektoraddition så adderas motsvarande position i matrisen med varandra.

Definition 2.16. *Matris addition*

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ g_1 & h_1 & i_1 \end{bmatrix} + \begin{bmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ g_2 & h_2 & i_2 \end{bmatrix} = \begin{bmatrix} a_1 + a_2 & b_1 + b_2 & c_1 + c_2 \\ d_1 + d_2 & e_1 + e_2 & f_1 + f_2 \\ g_1 + g_2 & h_1 + h_2 & i_1 + i_2 \end{bmatrix}.$$

Exempel 2.17.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}.$$

Vektorer kan också multipliceras med matriser. Vid matrismultiplikation så delas matrisen upp i två separata vektorer. Betrakta exempel 2.18.

Exempel 2.18. *Matrisen* $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ *kan ses som vektorn* $\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ *och* $\begin{bmatrix} 2 \\ 4 \end{bmatrix}$.

När man multiplicerar en vekto med en matris så är det viktigt att matrisens bredd är lika med dimensionen av vektorn. Man kan se det som att första elementet i vektorn skalärmultiplicerar den första vektorn av matrisen, och det andra elementet skalärmultiplicerar den andra vektorn av matrisen o.s.v. Betrakta exempel 2.19.

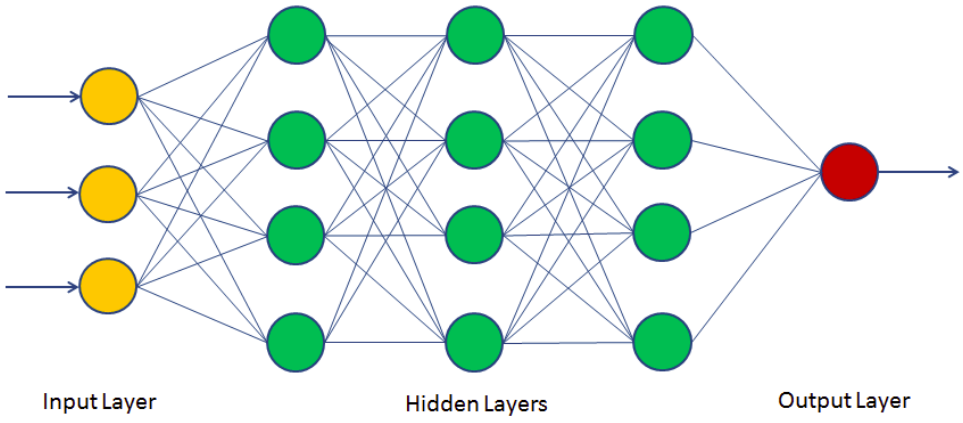
Exempel 2.19.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = 5 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 6 \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \end{bmatrix} + \begin{bmatrix} 12 \\ 24 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$$

2.5 Neurala nätverk

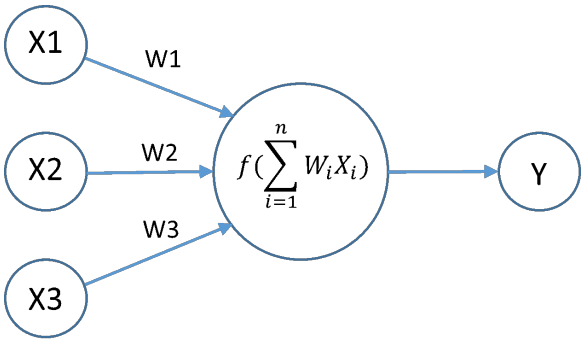
2.5.1 Bildlig förklaring

Ett neuralt nätverk är trots dess komplexa struktur en vanlig funktion. Det tar in data och och ger ut ett svar. Vad som gör det speciellt är att man har efterliknat modellen till hur hjärnan fungerar. Det lär sig dessutom på samma sätt männsikor lär sig:genom träning med exempel. Nätverket består av noder som är ordnade i lager. Varenda nod i det vänstra lagret är kopplad till varenda nod i nästa lager till höger, som i sig förhåller sig likadant till lagret till höger om sig (se figur 1).

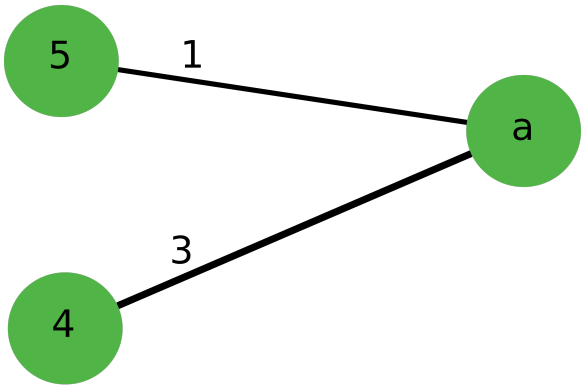


Figur 1: Nätverk med totalt fem lager

För att få en bättre uppfattning så betraktar vi en enstaka nod (se figur 2). För varje koppling så finns det ett assosierat värde, en så kallad vikt(variablerna $W1, W2, W3$). Den högra nodens värde bestämms genom summan av varje nod till vänster gånger kopplingens vikt. Sedan används en så kallad aktiveringsfunktion på hela summan. Än så länge kan betydlesen av aktiveringsfunktionen ignoreras då det kommer tas upp senare.



Figur 2: Allmän formel för värdet av en nod

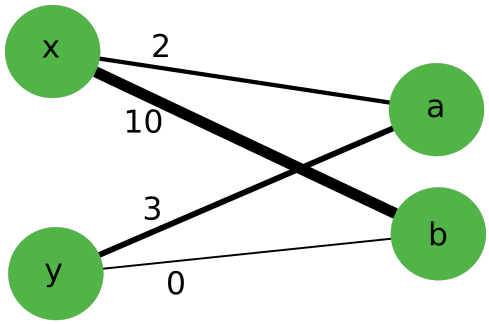


Figur 3: Exempelvikter till en nod

Exempel 2.20. *För att räkna ut värdet för noden a räknar man på följande vis:* $a = 5 \cdot 1 + 4 \cdot 3 = 17$.

2.6 Formell definition

Från att gå från ett lager till ett annat lager kan sybolisers med att ta en matris som består av vikterna gånger den vänstra lagret. Betrakta figur 2.6.



Figur 4: Ett nätverk som kan uttryckas med en matris

Detta nätverk kan uttryckas som

$$\begin{bmatrix} 2 & 3 \\ 10 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}.$$

Läsaren kan övertygas om detta genom att testa olika värden på x och y och se att a och b blir lika oavsett om man räknar genom figuren eller matrisen.

Dock som det nämndes så används en aktiveringsfunktion på den totala summan. En klassisk aktiveringsfunktion i neurala nätverk är sigmoid funktionen. Den är definerad som följande.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Figur 5: Definitionen av sigmoid funktionen

Sigmoids definitionsmängd är alla de reella talen och värdemängden är $0 < \sigma(x) < 1$.

Så om W symboliserar matrisen av vikterna och $\vec{x}$ symboliserar vänstra lagret som en vektor, så kan formeln för att räkna ut det högra lagrets vektor $\vec{y}$ skrivas som:

$$\vec{y} = \sigma(W\vec{x}),$$

givet att sigmoid är den valda aktiveringsfunktionen.

2.7 Grunderna till python och programering

2.7.1 Programmering

Programering är ett sätt att skriva instruktioner för vad en dator ska göra. Programmering utförs genom att skriva på ett låtsasspråk i en textfil. Sedan låter man datorn översätta textfilen direkt till ett program eller så låter man ett tolkprogram tolka instruktionerna i textfilen och utföra de direkt. Låtsasspråken kallas mer formellt som programspråk. De programspråk som kan översättas från en textfilen till ett program kallas kompilerade programspråk; De programsspråk där intruktionerna i textfilen tolkas direkt kallas interpreterade programspråk (Gustafsson och Holmberg 2011).

2.7.2 Python

Python är ett interpreterat programspråk. Detta är i fördel vid vetenskaplig programmering då man kan snabbt få resultat och inte behöva vänta på att koden kompileras. Instruktioner i python (och allmänt i programmering) ges genom att skriva funktioner. Funktioner i programmering är väldigt lika till funktioner i matematik, men inte helt ekvivalenta. Funktioner i python kan likt matematiska funktioner ta in ett tal och utföra någon operation och sedan mata ut det nya talet. Men den stora skillnaden är att en funktion i programmering kan också utföra ett moment. Detta kan vara att spara något i minnet, skicka ett medelande eller starta ett program. Tillskillnad från matematik, där funktioner betecknas med enstaka bokstäver, så betecknas funktioner i programmering med ord eller mindre meningar där varje ord är separerad med understreck. Den mest fundementala och viktigaste funktionen vid lärning av grunderna i python är funktionen *print*. Funktionen talar om för tolkprogrammet att visa dess argument. Betrakta följande kod:

```
1 print(5)
```

En femma visas då i tolkprogrammet som konsekvens. Python kod utförs uppifrån nedåt i ordning rad för rad. Så om man skriver

```
1 print(5)
2 print(4)
```

så kommer det först stå en femma och sedan stå en fyra. Python räknar ut och förenklar automatiskt matematiska uttryck.

```
1 print (9+10)
```

Resultatet av ovanstående kod blir då 19 och inte ”9+10”. Om man vill bokstavligt skriva ut symbolerna nio, plustecken, etta och nolla så omringar man uttrycket med citattecken.

```
1 print ("9+10")
```

Resultatet av ovanstående kod blir då ’9+10’ tillskillnad från koden innan. Värden kan sparas i variabler.

```
1 number = 2+2
2 text = "10+20+30"
3
4 print(number)
5 print(text)
```

Resultatet av ovanstående kod blir då att ’4’ visas först på skärmen och sedan ’10+20+30’. Alltså inte ’60’. Eftersom python kan krångla om man använder de svenska bokstäverna ’åäö’, så kommer kod i fortsättningen skrivas på engelska. Till varje variabel finns det en associerad typ. Variabeln *number* i det tidigare exemplet har typen *int*, förkortning för ordet integer, d.v.s. typen för heltal. Variabeln text har typen *str*, förkortning för string, för att syfta på att det är en sträng symboler som tolkas bokstavligt. Det är viktigt om man ska t.ex. plusa ihop två variabler, att de är av samma typ.

```
1 number = 10
2 text = "10"
3
4 print(text + text)
```

Ovanstående kod ger ett felmedelande då python ser text variabeln som en sammling symboler och vet inte hur man lägger ihop dem med ett nummer. Om man väl har samma typ så har operationen plus olika betydelse.

```
number = 10
text = "10"

print (number + number)
print (text + text)
```

Koden resulterar i att ’20’ först visas på skärmen och sedn ’1010’ Som man ser så utförs den matematiska operationen plus då variablerna är av typen *int*. Om de är av typen *str* så läggs symbolerna ihop. Det är dessutom viktigt att förstå att fast det är ett likhetstecken så ska man tolka det mer som att det som är på högersida läggs in på vänster sida.

```
x = 5
x = x + 10
print (x)
```

Den andra raden ser ut som ett felaktigt matematisk uttryck. Men det som sker är att första raden läses och python lägger in värdet 5 i variabeln x. Sedan läses andra raden och x + 10 räknas ut och sedan läggs det nya värdet in i x. D.v.s. ’15’ visas tillslut på skärmen.

2.7.3 Funktioner och loops

Som sagt så kan funktioner i python fungera som vanliga matematiska funktioner. Om man vill programmera funktionen

$$f(x) = 3x^2 + 4x + 5$$

så skriver man

```
def f(x):
  return 3 * x**2  +  4 * x  + 5

value = f(5)
print (value)
```

En stjärna betyder då multiplikation och två stjärnor betyder upphöjt. Men som det har tidigare nämnts så kan funktioner i python dessutom utföra moment. Om man vill visa texten ’Hello world’ och sedan resultatet av 2+2 på skärmen, så kan man definera funktionen

```
def f():
  print ("Hello world")
  print (2+2)
```

Som man ser i ovanstående kod så behöver funktionen inte ta några variabler som argument tillskillnad från riktiga funktioner. Man behövde dessutom inte skriva return då denna funktion enbart utförde något och inte gav tillbaka ett värde. En annan viktig observation är att alla instruktioner som tillhör funktionen är indenterade.

Som det har tidigare nämnts så är det mer korrekt enligt stilguiden att namge funktioner och variabler som ord eller meningar separerade med understreck. Så funktionen f skulle kunna namnges som

```
1 def do_something():
2   print ("Hello world")
3   print (2+2)
```

vilket är mer korrekt, även om de gör exakt samma sak. Dock om man skulle köra denna kod så skulle inget ske. Detta är p.g.a att koden endast förklarar vad funktionen do_something gör men utför aldrig den. För att utföra den så måste man skriva ut funktionen på en rad.

```
1 def do_something():
2   print ("Hello world")
3   print (2+2)
4
5 do_something()
```

För att repetera någonting skriver man som följande.

```
1 for number in range(0,4):
2   print(number)
```

Observera att variabeln number börjar på 0 går sedan igenom alla heltal lägre än fyra, men aldrig fyra i sig själv.

2.7.4 Moduler och något om objekt

Många användbara funktioner som är svåra att programmera har redan blivit kodade. Många av dessa finns tillgängliga genom moduler. En modul är en sammling funktioner och variabler man kan enkelt importera i sin egen kod. Antag att det finns en modul som heter Volumecontrols som har funktionen mute_volume. Givet att man har instalerat modulen på sitt system så kan man importera modulen och använda funktionen genom att skriva:

```
1 import Volumecontrols
2
3 Volumecontrols.mute_volume()
```

Man skriver alltså modulens namn och ett punkttecken och sedan namnet på funktionen. Men det kan bli jobbigt att jämnt skriva modulnamnet 'Volumecontrols' jämnt så då kan man importera modulen med ett smeknamn genom:

```
1 import Volumecontrols as vc
2
3 vc.mute_volume()
```

Många moduler används med objekt orienterad programmering. Objekt orienterad programmering är användingen av objekt. Ett objekt är en variabel som har associerade funktioner och värden. Dessa funktioner, och värden kan nås (likt moduler) genom att skriva namnet på objektet sedan ett punkttecken, och sist namnet på värdet eller funktionen. Ett objekt skapas genom att skriva namnet man vill ge på objektet på vänster sida av ett likhetstecken och sedan en funktion som skapar objekt på höger sida.

```
1 name_of_object = object_creating_method()
```

Så om man fortsätter på exemplet med låtsas modulen Volumecontrols så kan man tänka sig att Volumecontrols kan skapa objekt för olika enheter som kan spela ljud i hemmet, t.ex sin tv eller dator. Och om man har objeketet my_tv så kan man t.ex öka ljudet på sin tv med funktionen my_tv.increase_volume(), och så kan man se vilken nivå på ljudet med my_tv.volume. Så om man vill skriva ett program som visar volymen på sin tv och sedan ökar den, så kan det se ut som följande.

```
1 import Volumecontrols as vc
2 my_tv = vc.create_object(96896)
3 print (my_tv.volume)
4 my_tv.increase.volume()
```

Så den första raden är importeringen av modulen. I den andra raden skapas objektet. I detta exempel så tar funktionen som skapar objektet ett nummer. Detta symbolisera någon sorts ID på sin tv, så my_tv blir just den TV man syftar på. På tredje raden visas volymen my_tv har och på fjärde raden ökas volymen på my_tv.

Ett exempel på en typ som har objekt funktioner är list. En list är vad som det låter, en lista. En lista omringas av hakparanteser och elementen seprareras av kommatecken. Så om man vill ha en lista där första elementet är siffran 9, andra är siffran 8, och tredje 7 så skriver man som följande

```
1 my_list = [9,8,7]
```

Om man vill lägga till ett element kan man använda objektmetoden append.

```
1 my_list = [9,8,7]
2 my_list.append(5)
3 print(my_list)
```

resultatet av koden blir då att '[9,8,7,5]' visas på skärmen.

3 Metod

3.1 Skapandet av datan

3.1.1 Krav på datan

Oavsett hur man bygger upp det neurala nätverket så behövs det data med exempel som är redan lösta. I detta arbete är syftet att se om ett neuralt nätverk kan avgöra om ett 3-SAT problem är satisferbart eller inte. Alltså ska inputen vara ett 3-SAT problem och outputen vara en indikation på om det är satisferbart eller inte. För konstruktion av de neurala nätverk kommer modulen Tensorflow användas. Med den modulen krävs det att man har två listor: en för inputen och en för outputen. Alltså kommer första exemplets formel vara första elementet i inputlistan och svaret kommer vara första elementet i outputlistan. I detta arbete valdes det 60 000 träningsexempel och 10 000 utvärderingsexempel. Träningsexempelerna är de exempel det neurala nätverket kommer analysera och anpassa sig efter. Utvärderingsexemplaren är de som kommer användas i betygsättning av hur bra nätverket fungerar. Anledningen varför dessa måste vara separata är att det är väldigt enkelt för neurala nätverk att endast memorera exemplaren som användes vid träningen. Därför krävs det att exempelerna är unika så att inget exempel i träningsdatan finns dessutom i utvärderingsdatan.

Ett annat krav är att inputen och outputen är i form av en lista av reella värden. Detta är inte ett problem för outputen då man kan ha en en dimensionell lista där första värdet är 1 eller noll En lika tydlig lösning för inputen finns dock inte. Ett förslag är att associera ett värde till varje tecken och sedan låta antalet tecken beteckna dimensionen på listan (se exempel 3.1). Som man dock ser i exempel 3.1 så är dimensionen av vektorn onödigt stor då vissa tecken har alltid samma värde. De enda tecknen som kan vara olika är variablerna. I denna text kommer detta faktum utnyttjas, då antalet variabler i uttrycket kommer motsvara dimensionen av listan.

Exempel 3.1. *Antag att man har 3-SAT formeln:*

$$(a \vee b \vee c) \wedge (a \vee c \vee d).$$

Då kan man t.ex assosiera följande kod till varje tecken.

(	1
)	2
$\vee$	3
$\wedge$	4
a	5
b	6
c	7
d	8

Och då kan formeln översättas till följande lista.

$$[1, 5, 3, 6, 3, 7, 2, 4, 1, 5, 3, 7, 3, 8, 2].$$

3.1.2 Omvandling av 3-SAT formler

Låt variablerna i en formel ordnas i en viss order, exempelvis alfabetsordning och sedan stigande i nummer index. Den första variabeln kommer motsvara värdet 0, den andra 1, den tredje 2 o.s.v. Sedan när man har kodat för den sista variabeln med värdet n så låter man $n+1$ symbolisera negationen av den första variabeln och $n+2$ negationen för den andra variabeln o.s.v. Sedan låtes det variabeltecknet längst till vänster i formeln koda för första värdet i vektorn och den andra längst till vänster det andra värdet o.s.v. .

I denna text kommer det dessutom begränsas till max 5 olika variabler i varje formel. Därmed så kan varje heltal (om man tar hänsyn till ledande nollor) symbolisera en unik formel (se 3.2).

Exempel 3.2. *Talet* 013456025 *blir då formeln:*

$$(x_0 \vee x_1 \vee x_3) \wedge (x_4 \vee \overline{x_0} \vee \overline{x_1}) \wedge (x_0 \vee x_2 \vee \overline{x_0}).$$

Detta löser nästan problemet om att skapa 60 000 träningsexempel och 10 000 utvärderingsexempel som är alla unika. Eftersom varje heltal motsvarar en unik formel så kan man låta de talen från 0 till och med 59 999 koda för träningsexemplaren och talen 60 000 till 69 999 koda för utvärderingsexemplaren. Dock, som sagt löser detta nästan problemet. Första problemet är att om man ska ha fem stycken värdesiffror så kommer formelerna vara fem variabler långa. Men ett 3-SAT problem är på formen av en konjuktion av disjuktioner av längden tre variabler. Därför måste antalet variabler vara lika med en multipel av tre. Det andra problemet är att formeln är kort och när en formel är kortare så är det mindre chans att det blir en falsk formel (om man antar att antalet olika variabler håller sig konstant). Anledningen är att fler konjuktioner begränsar antalet möjliga lösningar (om antalet variabler håller sig konstant). Ett exempel på detta ges i 3.3.

Exempel 3.3. *Om en lösning existerar för formeln:*

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_3} \vee x_4 \vee \overline{x_1})$$

så innebär det att den lösningen får båda paranteserna att bli sanna, eftersom båda sidor måste vara sanna enligt definitionen av $\wedge$*. Det innebär att man kan plocka bort en parantes från formeln, och lösningen kommer fortfarande fungera på den nya formeln. Det samma kan dock inte sägas när man lägger till konjuktioner.*

Lösningen på problemet är att införa slumpmässiga siffror efter. På detta vis kommer alla exempel vara unika, och längden kan anpassas till en multipel av tre. I denna text kommer längden vara 72 variabler d.v.s. fem bestämda siffror och 67 slumpmässiga.

3.1.3 Simple-Sat

Simple-sat är en 3-SAT lösare kodad av Sahand Saba, och används i detta arbete under MIT lisens. Den tar in en input av typen *str* där varje konjuktion representeras med ett komma och disjuktioner av mellanslag. Så en variabel kan vara vilken kombination av bokstäver och siffror så länge det inte innehåller ett mellanslag. Det innebär att till skillnad från programspråk som python så kan en enkel siffra vara en unik variabel. Detta är i fördel till omvandlingen i denna metod eftersom varje siffra kodar en viss variabel. För att beteckna negationen används tecknet tilde (˜) framför variabeln utan mellanslag.

Problemet med detta program är att det kan endast bli tillkallad i en terminal. En terminal är ett helt textbaserat program som man kan skriva in kommandon till för att starta olika program. De främsta operativsystemen är egentligen en terminal som kör i bakgrunden och det grafiska användargränsnitt man har tillgänglig är ett program som körs genom terminalen. Om man använder windows så heter terminalprogrammet ”CMD”, på Ubuntu heter (den som ingår) bash. 3-SAT programmet är gjort för terminaler under operativsystem Linux. Så två problem måste lösas. Det första problemet är att omvandla en string av siffror till en formel som 3-SAT programet kan tolka. Det andra problemet är hur man ska kunna intargera med detta program genom python, då programet är gjord för att användas i en terminal i linux.

3.1.4 Omvandling till formel

För att omvandla en string av siffror till en string i det format 3-SAT programmet tar emot definerar vi funktionen generate_formula. Generate_formula tar argumentet **seed** som är stringen av siffror. Funktionen inleds med att definera två variabler.

```
1 digits = seed
2 formula = []
3 disjunct_counter  = 0
```

Seed döps om av den enkla anledningen att det kommer hjälpa med tydlighet i en loop som kommer senare. Formula variabeln är där alla tecken till formeln kommer läggas till. Vid slutet av programmet kommer elementen i formula sättas ihop till formel stringen, men eftersom man har mer kontroll med listor, defineras den som en tom lista i början. Variabeln disjunct_counter räknar antalet variabler d.v.s siffror i stringen. Detta behövs eftersom vid varje tredje variabel läggs det till en konjuktion, eller i fallet av denna 3-SAT lösare ett komma.

Nästa del är att gå igenom varje siffra i digits och utföra ett av två fall och eventuellt ett specialfall. Specialfallet är när disjunct counter är multipel av tre. Då vill man lägga till ett komma och ett mellanslag. Så den första if klasulen är:

```
1 if disjunct_counter % 3 == 0:
2     formula.append(",")
```

```
3     formula.append(" ")
```

Det finns dock ett problem med denna kod. Noll är ju också en multipel av tre och det innebär att ett komma och ett mellanslag kommer läggas till i början av loopen då disjunct_counter börjar på noll. Så därför lägger vi till kravet att disjunct_counter är dessutom inte lika med noll.

```
1 if disjunct_counter % 3 == 0 and disjunct_counter !=0:
2     formula.append(",")
3     formula.append(" ")
```

Sedan så finns det ett av två fall som alltid kommer ske exklusivt. Det ena är att siffran är lägre än fem och den andra är att siffran är högre eller lika med fem. Om siffran är lägre än fem så vill man lägga till siffran och ett mellanslag till formula listan. Om siffran är högre så vill man lägga till tilde och en nolla om det är fem, tilde och en etta om det är sex o.s.v.. Alltså vill man därmed lägga till tilde och siffran minus 5, samt ett mellanslag. Dessa två fall kan skrivas som följande:

```
1 if int(digit)<5 :
2     formula.append(str(digit))
3     formula.append(" " )
4 else:
5     formula.append("~")
6     formula.append(str(int(digit)-5))
7     formula.append(" ")
```

Anledningen varför man lägger till str(digit) och inte bara digit är eftersom digit är av typen Int. För att kunna sätta ihop listan formula så behövs det att alla element är av typen String. Funktionen str tar siffran och ger tillbaka samma siffra fast med typen String. För att sedan få funktionen att ge tillbaka formeln fast som en string skriver man:

```
1 return "".join(formula)
```

Funktionen .join kanske ser märkligt ut först men det är helt enkelt en funktion som tar en lista och ger tillbaka en string där varje element är ihopsatt. Anledningen för den märkliga syntaxen är att det som är emellan citattecknena är det som kopplar ihop elementen, betrakta exempelA. Eftersom det inte ska vara något emellan elementen i listan formula så skrivs det ingenting mellan citattecknena och därmed skrivs det .join.

Så funktionen i helhet ser ut som följande.

```
1 def generate_formula(seed):
2     """Generates a formula that can be checked with the check function.
3     """
4     digits = seed
5     formula = []
```

```
6     disjunct_counter  = 0
7
8     for digit in digits:
9         if disjunct_counter % 3 == 0 and disjunct_counter != 0:
10            formula.append(",")
11            formula.append(" ")
12        if int(digit)<5 :
13            formula.append(str(digit))
14            formula.append(" " )
15            if debug:
16                print ("borde vara under 5: " + str(digit))
17        else:
18            formula.append("~")
19            formula.append(str(int(digit)-5))
20            formula.append(" ")
21            if debug:
22                print ("borde vara over 5: "+ str(digit))
23        disjunction_counter += 1
24
25    return "".join(formula)
```

3.1.5 Satisferbarhet av formeln

För att få tillgång till att skicka kommandon till terminalen via python använder man modulen Subprocess. Funktionen subprocess.check_output tar in som argument ett terminal kommando och ger tillbaka resultatet av vad som skrevs i terminalen. Dock så tar funktionen inte en string som input fast istället en lista som sedan läggs ihop till en string där varje element är separerad med mellanslag. För att tillkalla programmet i terminalen skriver man:

```
1 python simple-sat/src/sat.py -ti "formula"
```

Så om variabeln string innehåller formeln och om man vill spara formelns validitet i en variabel result i python, så skriver man:

```
1 result = subprocess.check_output(["python",
2                                   "simple-sat/src/sat.py",
3                                   "-ti", string])
```

Programmet 3-SAT ger tillbaka en string med info om vad den satisferbara lösning är, om formeln är satisferbar. Om den inte är satisferbar ges en tom string tillbaka. Eftersom informationen om det är satisferbar eller inte är det enda som är relevant i skapandet av datan så räcker det att kolla om

outputen är tom eller inte. På grund av hur python hanterar strings så finns det två sätt att skriva en tom string : ” och b”. Funktionen ser då ut i sin helhet som följande:

```
def check(string):
    """Checks if a formula generated by the generate_formula method,
    can be satisfied or not.
    """
 result = subprocess.check_output(["python",
                                   "simple-sat/src/sat.py",
                                   "-ti",
                                   string])
 if result == "" or result == b'':
    return 0
 else:
    return 1
```

3.2 Algoritmen för skapandet av datan på en allmän nivå

1. Skapa en tom lista för all träningsdata
2. Skapa en tom lista för all utvärderingsdata
3. Låt n = 0
4. Omvandla n till en lista med 5 värden
5. Skapa en lista med 67 slumpmässiga värden
6. Sätt ihop listorna från 2 och 3 till en ny lista
7. Lägg till listan från 8 i input träningsdata listan 1
8. Omvandla listan från 8 till en formel
9. Kolla om formeln från 10 är satisferbar
10. Om den är satiferbar lägg till en etta i output träningsdatan
11. Om den inte är det lägg till en nolla i output träningsdatan
12. Repetera steg 3 till 13 där n ökar med ett varje gång, tills n=69999
13. Exportera datan

3.3 Algoritmen i python kod

3.3.1 Numpy arrays

För att översätta den allmänna koden till python så skulle det tekniskt gå att ha vanliga listor. Men p.g.a att det är så stora värden samt att vi vill senare kunna exportera dessa listor, så är det mer praktiskt att använda en modul som heter Numpy. Listor i numpy heter numpy arrays". Numpy arrays fungerar i omfattningen av denna text i princip som vanliga lists. Enda skillnaden är att det inte går att direkt modifera dem. Om man vill ändra en variabel som innehåller en numpy array så måste man skapa en ny array baserad på den gamla och sedan spara den nya i variabeln . För att skapa en numpy array använder man funktionen numpy.array. Som argument sätter man i en vanlig list som numpy arrayen kommer efterlikna.

```
1 import numpy
2 numpy_array = numpy.array([4,2,0])
```

En sak man måste ta i hänsyn med numpy är att man måste specifiera vid skapandet av en array vad för typ alla element kan vara. Man kan därmed inte ha olika typer. Detta i kontrast till vanliga listor som man kan ha flera olika typer. Betrakta exempel6 där listan av sig själv fungerar men om man omvandlar till en numpy array skapas det ett felmedelande. Man kan dock skapa en tom arrray och lägga till värden, så detta är inte ett problem för ouputdatan. Men för input datan så har man en sammling av listor av storleken 72. Om man skulle skapa en tom numpy array och en numpy array med 72 element och sedan försöka använda numpy.concatenate, visas ett felmedelande. Man är då tvungen att vid skapandet av numpy arrayen för träningsdata redan ha ett element som är en numpy array på storleken 72. Ett enkelt sätt att göra detta är med funktionen numpy.zeros. Numpy.zeros tar in en storlek som argument och skapar en numpy array fylld med nollor med den specifierade storleken. För att skapa en array med 5 nollor skriver man numpy.zeros(5). För att skapa en array fylld med 3 arrays där varje array har 2 nollor skriver man numpy.zeros((3,2)) .

Eftersom träningsdatan kommer vara en array där varje element är en array på 72 element, så skriver man numpy.zeros((1,72)). Ett problem som uppstår på grund av detta är att första träningsexemplet kommer vara på plats två och det andra på plats 3 o.s.v (se figur1). Detta löses dock enkelt genom att ta bort det första elementet i slutet av algoritmen. Man kan göra detta med funktionen numpy.delete. En sista detalj är att koden så betecknas input med x och output med y. Notationen är inte för någon speciell anledning, för att det ska bli enklare att skriva variabeln. Så steg 1 och två kan översättas till följande kod:

```
1 import numpy as np
2
3 x_train = np.zeros((1,72))
4 y_train = np.array([])
```

3.4 Loopen steg 3 till 7

Nästa del i koden är att påbörja loopen från steg 3 till 12. Första steget i loopen är att omvandla n till en lista med fem värdesiffror. För det första så är det mer korrekt enlight pythons rekomederade stilformat att ha ord istället för bokstäver som variabler, så n kommer reffereras som num i python koden. Omvandlingen av num delades upp i fyra delar. Först omvandlades num till en String. Sedan lades det eventuellt på ledande nollor för att få Stringen att ha 5 värdesiffror. Sedan omvandlades den till en vanlig lista, och tillsist en numpy array. Steg 1 och 2 blev en rad och steg 3 och 4 blev också en rad. Koden såg ut som följande:

```
1 num_string = str(num).zfill(5)
2 head_array = np.array(int_list(num_string))
```

Steg 5 skrevs på följande vis:

```
1 tail_array = np.random.randint(10, size=67)
```

Första argumentet anger då att alla heltal under det givna nummret får slumpmässigt skapas, och size parametern talar om för storleken av arrayen som skapas.

För att sedan lägga ihop de två arraysen används np.concatenate.

```
1 array_to_append = np.concatenate((head_array,tail_array))
```

Sedan för att lägga till denna array till träningsdatan använder np.concatenate för att updatera x_{train}.

```
1 x_train = np.concatenate((x_train, [array_to_append]))
```

3.5 Loopen steg 8 till 11

För att se om formeln är satisferbar eller inte används en 3-SAT lösare programerad av Sahand Saba under MIT licens. Sabas program körs egentligen via terminalen, men den är tillgänglig med modulen satsolver.py som finns beskriven i appendix I. Modulen innehåller två funktioner: satsolver.generate_formula och satsolver.check. Satsolver.generate_formula tar en numpy array och omvandlar den enligt metoden beskriven i omvandling av 3-sat, till en formel satsolver.check kan ta som argument. Satsolver.check ger då värdet noll om formeln är ej satisferbar och värdet ett om den är satisferbar. Så koden blir som följande:

```
1 formula = sat.generate_formula(array_to_append)
2 value_to_append = sat.check(formula)
3 y_train = np.append(y_train, value_to_append)
```

Ett steg som inte står beskrivet i den allmänna algortimen, är att radera det elementet i x_train. Detta var som sagt p.g.a vid skapandet av x_train behövdes ett element fylld av nollor som specifierade storleken på arrayen. Så för att radera det första elementet i x_train används följande kod:

```
x_train = np.delete(x_train, 0, 0)
```

3.6 Export av datan

För att exportera numpy arrays används funktionen np.save. Funktionen tar in namnet på en filplats och numpy arrayen man vill spara. I detta projekt så sparades filerna i mapp kallad 'data'. Så exporterings koden såg ut som följande:

```
np.save('data/x_train', x_train)
np.save('data/y_train', y_train)
```

3.7 Anpassning för utvärderingsdatan

Koden för utvärderingsdata är i princip likadan. Enda skillnaden är att num fick variera mellan 60 000 och 69 999 och datan sparades som x_test och y_test för input datan respektive output datan.

3.8 Skapandet av modellen

3.8.1 Representation av modellen

Modellen består alltså av 10 lager med 128 noder var, där Rectifier är aktiveringsfunktionen (en aktiveringsfunktion som inte är begränsad tillskillnad från sigmoid). Det finns dessutom ett till lager i slutet som består av endast en nod och har Sigmoid aktiveringsfunktionen .

3.8.2 Tensorflow och Keras

För att skapa den här modellen i python används modulerna Tensorflow och Keras. Tensorflow är en modul som tillåter en att bygga neurala nätverk från grunden. Keras är en modul som som förenklar byggandet och träningen av neurala nätverk. I koden kommer dessa moduler och delar modulerna importeras:

```
import numpy as np
from tensorflow import keras
from tensorflow.keras.layers import Activation, Dense
```

För att skapa en tom modell som man sedan kan senare tillägga fler lager till skriver man:

```
model = keras.models.Sequential()
```

Variabeln model blir då ett objekt som symboliserar det neurala nätverket. För tillägg av lager samt träning och utvärdering av modellen används assosierade objekt metoder. För att lägga till ett lager (till höger om man tänker som fig2 där input går in till vänster och output går ut från höger) så

använder man objektmetoden add. Som argument tar den vad för sorts lager. Den sortens lager som har hittils gått igenom i texten kallas i keras för 'Dense'. Det är alltså ett lager som kan symboliseras med en matris, eller noder på rad, samt en associerad aktivationsfunktion. Så för att lägga till ett lager med 128 noder, och Rectifier som aktiveringsfunktionen skriver man:

```
1 model.add(Dense(128, activation='relu'))
```

relu symbolisera då Recitifier funktionen. Den här raden av kod repeteras då 10 gånger så att det blir totalt tio lager. Sista lagret är två noder där första noden är sannolikheten att det är noll och andra är sannolikheten att det är 1. Detta görs med aktiveringsfunktionen softmax". Följande kod motsvarar detta lager:

```
1 model.add(Dense(2, activation='softmax'))
```

3.9 Träning av nätverket

För träning av nätverket används binary cross entropysom loss funktion, 'accuracy' som metric och stokastisk gradient nedgång som optimizer. 'Binary crossentropy' och 'accuracy' förklaras ej i detta arbete men dokumentaion om dessa funktioner finns på . För att träna nätverket och sedan utvärdera det skrivs följande kod:

```
1 model.compile(optimizer = 'SGD',
2             loss='binary_crossentropy',
3             metrics=['accuracy'])
4 model.fit(x_train, y_train, epochs=5)
5 model.evaluate(x_test, y_test, verbose=2)
```

Epochs är hur många gånger den går igenom all data. Så epoch 1 är första gången den går igenom datan epoch 2 andra gången o.s.v. Verbose variabeln specifierar bara hur mycket information som printas ut.

4 Resultat

Epoch	tid	resultat
1	8 sekunder	50 %
2	7 sekunder	50 %
3	7 sekunder	50 %
4	7 sekunder	50 %
5	7 sekunder	50 %

Och för träningsdatan var resultatet 50 %.

5 Diskussion och slutsats

Som man ser så var resultatet på samtliga delar av utvärderingsprocessen 50 %. Detta tyder med säkerhet att nätverket inte har hittat något mönster då det är lika med chansen för en gissning. Detta stödjer kontentan bland forskare att P är skiljt från NP (Sipser 1992). Eftersom om det finns en tydligt mönster som neurala nätverk kan se så skulle det öka chansen att det finns en enklare algortim för 3-SAT problem av allmänn storlek, som är tillräckligt enkel så att den är i polynomisk tid. Detta utesluter såklart dock inte att neurala nätverk kan tränas att lösa 3-SAT problem. På grund av att koden behövde köras på en persondator, så var antalet lager, noder, epochs, och träningsexempel begränsade till processorkraften och minnet tillgängligt.

Referenser

Cook, Stephen A. (1971). "The Complexity of Theorem-Proving Procedures". I: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, s. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

Gustafsson, Fredrik och Viktor Holmberg (2011). "Gradvis typat Python med Cython". I:

Sipser, Michael (1992). "The history and status of the P versus NP question". I: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, s. 603–618.

Skjelnes, Roy (2015). "Linjär algebra". I: URL: https://www.kth.se/polopoly_fs/1.591293.1550157526!/LinearAlgebra-2015.pdf.

5.1 Github

Alla program samt diagram som användes i samband med detta arbete finns tillgängliga i följande github repo: