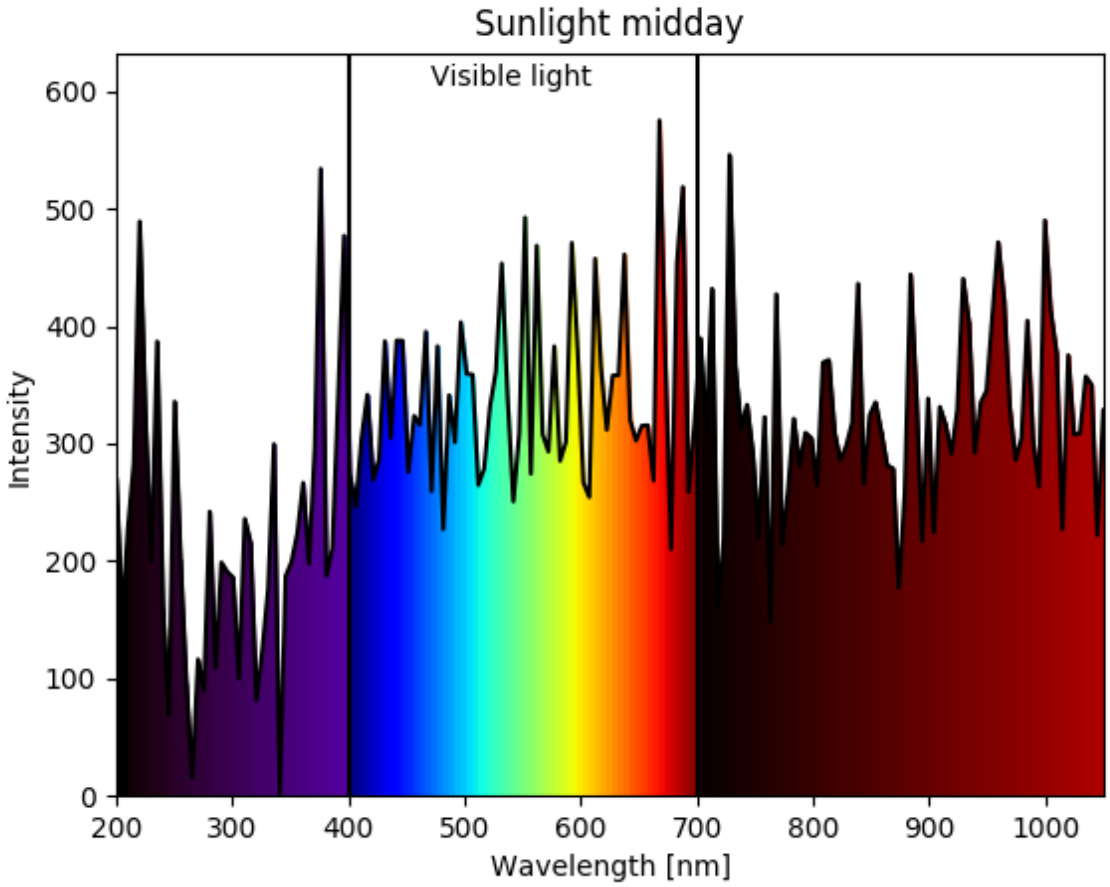
CHALMERS



Karakterisering av ljus med sensorfusion

Spektrumanpassning med Metropolis Monte Carlo och klassificering med artificiella neurala nätverk

Examensarbete inom Mekatronikprogrammet

Emily Giotas
Anesa Hodzic

Institutionen för Signaler och system
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2017

EXAMENSARBETE 2017

Karakterisering av ljus med sensorfusion

Spektrumanpassning med Metropolis Monte Carlo
och klassificering med artificiella neurala nätverk

Emily Giotas & Anesa Hodzic

CHALMERS

Karakterisering av ljus med sensorfusion
Spektrumanpassning med Metropolis Monte Carlo och klassificering med artificiella neurala nätverk

Författare: Emily Giotas & Anesa Hodzic

Chalmers Tekniska högskola AB
Hörselgången 4
412 96 Göteborg
Telefon: 031-772 27 21

Framsida: Spektrumanpassning för solljusmätserie gjord klockan 12.00, April 2017, Göteborg.

Förord

Det här examensarbetet utfördes av två studenter på mekatronikingenjörsprogrammet Chalmers tekniska högskola. Examensarbetet utfördes på instutitionen Signaler och system och omfattar 15 högskolepoäng.

Projektet utfördes på uppdrag av Broccoli Engineering AB som arbetar med konsultuppdrag inom mjukvaruutveckling och elektronik i Göteborg.

Vi skulle vilja ge ett gigantiskt tack till Rasmus Andersson för snabbkursen i Metropolis Monte Carlo-algoritmen. Stort tack till Christian Ågren för det korta men insiktsfulla förklaringen till de olika artificiella neurala nätverkens för- och nackdelar.

Vi skulle även vilja tacka vår handledare Björn Bergholm, vår examinator Bertil Thomas samt Göran Hult och Sakib Sistek för alla deras idéer och stöd.

Emily Giotas & Anesa Hodzic

Sammanfattning

Karakterisering av ljus sker idag med hjälp av kostsamma spektrometrar. Den huvudsakliga frågeställningen i detta arbete är om det går att enklare och billigare identifiera en ljuskällas spektrum och om det går att särskilja olika tider på dygnet genom att undersöka den relativa intensiteten hos specifika våglängder. Arbetet går ut på att med enkla och billiga fotosensorer ta fram en prototyp som kan ge information om ljuset i omgivningen.

Sensorerna som används är en fotodiod, en RGB-sensor samt en UV-sensor. Dessa tre sensorer, som sammanlagt ger fem sensorvärden, kopplas till en Arduino Uno Rev. 3 och värderna skickas via en CAN-buss till en dator.

Loggfilerna med sensorvärdern analyseras på två olika sätt, först genom en spektrumanpassning med Metropolis Monte Carlo-metod. Metropolismetoden är en stokastisk optimeringsalgoritm som här tillämpas genom att baserat på sensorernas känslighetskurvor förutspå deras utsignal för ett givet våglängsspektrum. Spektrat varieras slumpvis i varje steg och om de förutspådda sensorsignalerna kommer närmare de verkliga signalerna behålls förändringen, annars återställs spektrat. Algoritmen går ut på att upprepa dessa steg tills spektrat konvergerat till en stabil lösning.

Det andra sättet mätdatan analyseras är genom att ljuset klassificeras med ett djupt artificiellt neuralt nätverk skrivet i TensorFlow.

Resultatet från Metropolis Monte Carlo-programmet visar att det finns en potential för ett bra substitut till en spektrometer. För större noggranhet behövs fler och bättre sensorer, till exempel en CMOS bildsensor med större optisk area och/eller fotodioder vars känslighetskurvor har andra våglängdstoppar.

I ett av de undersökta fallen ger det artificiella neurala nätverket ger en noggrannhet på 96% vilket innebär att nätverket klassificerar rätt på ljusförhållandet 96 gånger av 100. Resultatet kan förbättras med bättre och fler mätningar för träningsdatan.

Abstract

Characterization of light is currently done using costly spectrometers. The main questions treated here are whether there are easier and less expensive ways to identify the spectrum of a light source and whether sunlight from different times of the day can be distinguished by the measured intensity at specific wavelengths. Only simple and cheap photo sensors are used to develop a prototype that can provide information about the light in the environment.

The sensors used are a photo diode, an RGB sensor and a UV sensor. These three sensors, which together provide five sensor values, are connected to an Arduino Rev. 3 and the values are sent to a computer via a CAN bus.

The log files with sensor values are analyzed in two different ways, first by a spectrum fitting using the Metropolis Monte Carlo algorithm. The metropolis method is a stochastic optimization algorithm that is used here to predict sensor output based on the sensitivity curves of the sensors, assuming a particular spectrum. The candidate spectrum is varied at each step, and whenvever the predicted sensor signals come closer to the real sensor outputs the changes are retained, otherwise they are discarded. The algorithm is based on repeating these steps until the spectrum has converged to a stable solution.

The second way the measurement data is analyzed is by classifying the light with a deep artificial neural network in TensorFlow.

The result of the Metropolis Monte Carlo program shows that there is a potential for a good substitute spectrometer. For greater accuracy, more and better sensors are required, such as a CMOS image sensor with larger optical area and/or photo-diodes whose sensitivity curves have other wavelength maxima.

In one of the cases investigated, the artificial neural network provides an accuracy of 96% which means the network classifies correctly on the light condition 96 times of 100. The result can be improved with better and more measurements for the training data.

Innehåll

Beteckningar

AD = Analog/digital
ANN = Artificiellt neuralt nätverk
CAN = Controller Area Network
CDF = Cumulative Distribution Function, kumulativ fördelningsfunktion
CPU = Central Processing Unit (centralprocessor)
GND = ground, jord; nollpunkt i en krets
GPU = Graphics Processing Unit (grafikbehandlingsenhet)
ID = Identifikation
IR = Infraröd
OP = Operationsförstärkare
PDF = Probability Density Function, sannolikhetdensitetsfunktion
PWM = Pulsbreddsmodulering
ReLU = Rectified Linear Unit
RGB = Red Blue Green (röd blå grön)
USB = Universial serial bus
UV = Ultraviolett

Kapitel 1

Inledning

Här presenteras en undersökning av ordinära sensorer och vilken information de ger om ljuset i omgivningen. Sensorerna består av en RGB-sensor, en fotodiod och en UV-sensor. Sensorerna kopplas samman med en Arduino och informationen skickas över en CAN-buss.

Mätvärderna från sensorerna analyseras på två sätt, en karakterisering med hjälp av Metropolis Monte Carlo och en klassificering med artificiellt neuralt nätverk i TensorFlow.

1.1 Bakgrund

Idag forskars det mycket på hur ljus påverkar både människor och växter. Till exempel visar forskning på hur vår exponering för blått ljus från mobiler, surfplattor, datorer och tv-apparater påverkar melatoninnivån och gör personer alerta och vakna. Gult och rött ljus har motsatt effekt och gör människor avslappnade och lugna. Växternas tillväxtfaser påverkas av olika våglängder. Det går att påverka deras växtfaser som till exempel tillväxt, blomning och vilofas.

Idag spenderar vi betydligt mer tid inomhus än vad vi gjorde förr och då är det särskilt viktigt att vi kompenserar de våglängder som saknas. Studier har visat samband mellan närsynhet hos barn och för mycket inomhusvistelse. Med inomhusvistelse menas huvudsakligen avsaknad av dagsljus [1].

Karakterisering av ljus sker i dagsläget med hjälp av optiska spektrometrar. Spektrometer är ett vetenskapligt instrument som används till att dela upp ljus i ett spektrum av olika färger. De är uppbyggda av prismor och gitter

som separerar våglängderna och visar intensiteten på ljuset som en funktion av våglängderna alternativt frekvenserna.

Spektrometrar används till exempel i astronomin för att analysera strålningen från astronomiska objekt och inom kemin för att härleda ett materials kemiska sammansättning. Instrumenten som finns på marknaden är för dyra för att kunna säljas till privatpersoner och används nästan uteslutande i labbmiljöer eller företag som ägnar sig åt ljusanalys. Priset för en optisk spektrometer ligger från 10000 SEK och uppåt och möjligheten att sänka kostnaden är stor [2].

1.2 Syfte

Frågeställningen är om det finns något enklare och billigare sätt att producera en optisk spektrometer. Spektrometern ska kunna identifiera våglängderna som en ljuskälla består av och skilja på solljus under olika tider på dygnet. Uppdraget går ut på att med enkla, billiga sensorer samla in mätdata och identifiera information om ljusets våglängder samt dess intensitet.

1.3 Mål

Målet är att undersöka vad ett litet antal billiga sensorer kan ge för information om ljusets spektrum och utifrån informationen bestämma ljusets våglängdsspektrum samt särskilja olika ljuskällor. Med hjälp av informationen om ljuset från sensorerna ska en billig optisk spektrometer framställas.

1.4 Avgränsningar

Ljusets våglängder som undersöks avgränsas från 200 nm till 1100 nm. All datakommunikation sker via CAN-buss och samtliga sensorer som används kostar omkring ett hundra svenska kronor. Ingen färdig produkt för försäljning byggs. Tidsramen för projektet är 10 veckor.

1.5 Specificering av frågeställningen

1. Går det att med hjälp av färgsensor och fotodioder fastställa vilken typ av ljuskälla som undersöks?

2. Går det att se skillnad på morgon- och kvällsljus samt utomhusljus och inomhusbelysning via sensorvärderna?

3. Finns det någon möjlighet att få fram ett våglängdsspektrum med informationen från sensorerna?

Kapitel 2

Teoretisk och teknisk bakgrund

I det här avsnittet ges en kort beskrivning av de olika komponenter som används, en beskrivning av den utvecklade mjukvaran och en kortfattad introduktion till teorin bakom de olika beräkningsalgoritmerna.

2.1 Ljus

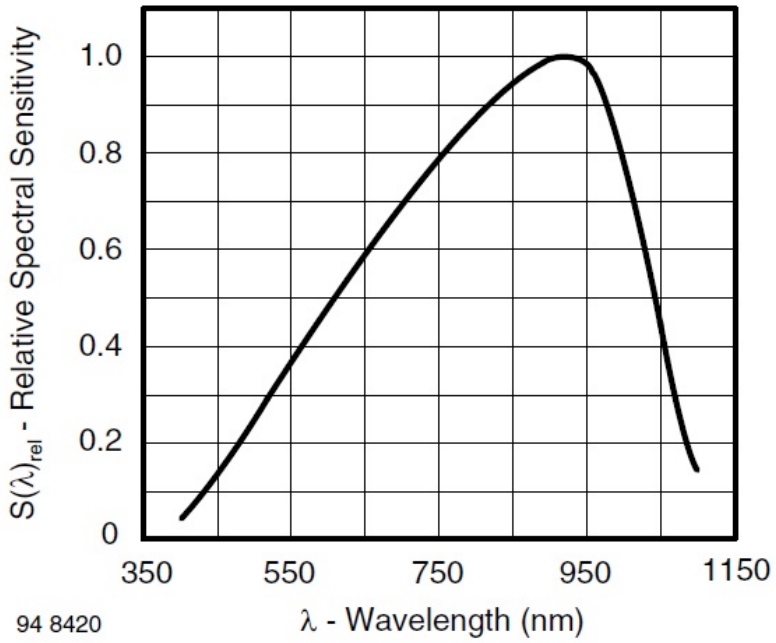
Ljus kan ses som både vågor och partiklar bestående av fotoner [3]. Vitt ljus består av våglängder inom intervallet $400-800$ nm vilket även motsvarar det synliga ljuset. UV- och IR-strålning ligger utanför intervallet, UV har kortare våglängder än 400 nm och IR har våglängder längre än 800 nm. Solljus har ett kontinuerligt våglängdsspektrum, medan artificiellt ljus oftast består av endast vissa våglängder och ger ett diskret spektrum.

Solljus består av alla våglängder, det vill säga UV-strålning, vitt ljus och IR-strålning. Vid soluppgång och solnedgång har ljuset en varmare färg. Det beror på att de blå våglängderna reflekteras bort i atmosfären. Ljuset skiljer sig även mellan soluppgång och solnedgång eftersom själva atmosfären har olika egenskaper på morgonen och på kvällen. Under dagen värms jorden upp vilket bidrar till ökad mängd partiklar i atmosfären. Även vinden och mänsklig aktivitet bidrar till ökningen av partiklarna. Det ger en högre absorption av det blåa ljuset. Därav ser solnedgången rödare ut än vad soluppgången gör.

UV-strålning delas upp i UVA, UVB och UVC. UVC kommer aldrig fram till jordens yta på grund av att de filtreras bort i atmosfären. UVB absorberas av glas så mätningar som görs genom en glasruta registrerar endast UVA. IR-strålning är också känt som värmestrålning.

2.2 Fotodiod, VBP104S

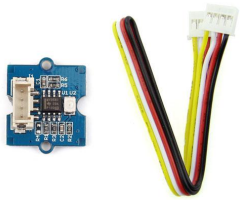
Fotodioden Vishay VBP104S IR+ Visible Light Si Photodiode är en silicon-baserad ytmonterad fotodiod med hög respons [4]. Den ljuskänsliga arean är 4.4 mm^2 och har en halvkänslighetsvinkel på $\pm 65^o$. Strömmen IF = 50 mA. Mätområdet för dioden är 430 nm till 1100 nm och den ger högsta möjliga ström vid 950 nm (se figur 2.1).



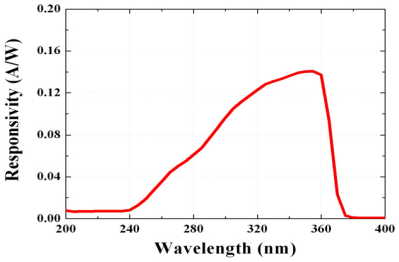
Figur 2.1: Fotodiodens (VBP104S) mätområde för ljusets våglängder [nm] och dess relativa ljuskänslighet [4].

2.3 Grove UV-sensor, GUVA-S12D

UV-sensorn är av sorten GUVA-S12D med ett spektralområde $200 - 400$ nm. Dioden är av Schottky-konstruktion , vilket innebär hög UV-känslighet [5]. UV-sensorns känslighetsgraf visas i figur 2.2b.



(a) Grove UV sensor [6].



(b) Känslighetskurva för UV-sensorn.

Figur 2.2: (a) Grove UV sensor. (b) Graf över UV-sensorn som visar förhållande mellan våglängder [nm] och responsivitet [A/W] där maximala responsiviteten ligger vid våglängden 360 nm.

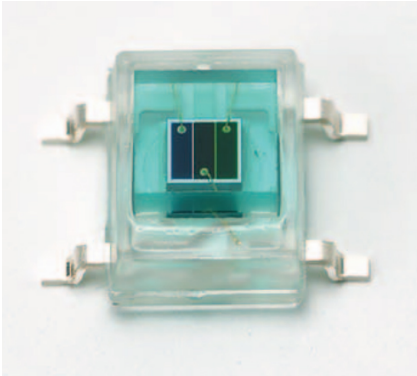
2.4 Arduino

Arduino Uno Rev. 3 (se figur 2.3) är ett utvecklingskort till för att förenkla elektronikanvändsning. Hårdvaran består av en mikrokontroller, 14 digitala in- och utgångar, varav 6 är PWM-styrda, samt 6 analoga ingångar. Arduino programmeras via USB med Arduino Software som liknar programmeringsspråket C++. Arduino drivs antingen via USB eller med en 5 V spänningskälla.

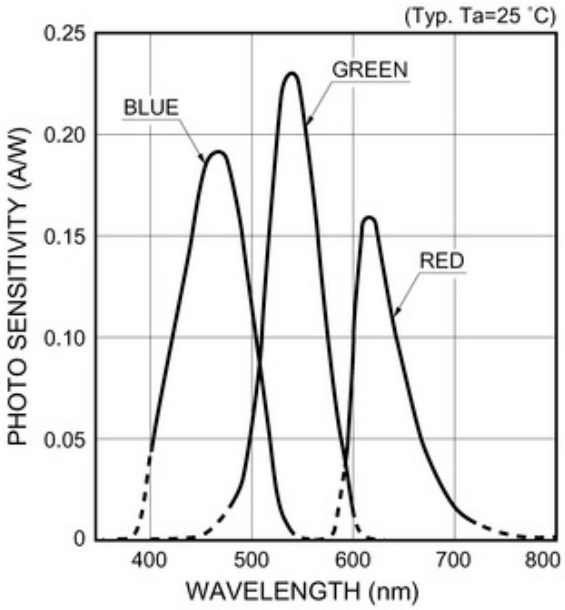
Figur 2.3: Arduino Uno Rev. 3 [7].

2.5 RGB-sensor, S9702

RGB-sensorn som används är av typen S9702, Hamamatsu (se figur 2.4). Den är uppbygd av tre fotodioder täckta av tre grundfärgsfilter; röd, grön och blå. De tre filtrerna släpper in våglängder som motsvarar dessa färger [7]. Sensorn ger maximalt utslag för blått ljus vid 460 nm, grönt vid 540 nm och rött vid 620 nm vilket kan utläsas ur figur 2.5.



Figur 2.4: RGB-sensor S9702, Hamamatsu. [7].



Figur 2.5: Grafen över RGB-sensor med förhållandet mellan ljussensitivitet och våglängd, datablad, S9702, Hamamatsu. [7].

2.6 Sensorfusion

Sensorfusion (även känt som multisensor eller givarfusion) innebär att data från flera sensorer eller annan data kombineras [8]. Det betyder att informationen får en mindre osäkerhet än om datan kommer från individuella källor. Osäkerhetsreduktionen kan betyda en mer exakt, mer fullständig eller mer pålitlig information.

Sensorfusion används här genom att kombinera sensordata från olika sensorer för att upplösa ljusets intensitet för olika våglängder.

2.7 CAN-bussen

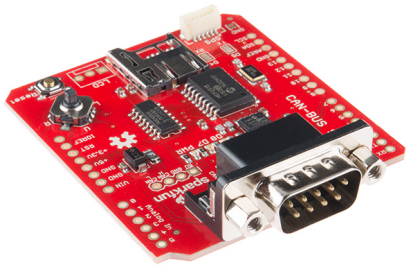
CAN-bussen är en buss utformad så att mikroprocessorer och andra enheter kan kommunicera med varandra utan en värddator [9]. CAN används främst i fordonsindustrin där sensorer ska samarbeta med varandra för en större säkerhet i form av till exempel autobromsar och autoparkering.
CAN är en multiserial buss som kräver två eller flera noder i CAN-nätverket. Nodernas komplexitet kan vara allt från binärt till sofistikerad mjukvara. Noden, om den används som en port, tillåter en dator att kommunicera över till exempel USB med andra enheter i nätverket.

Alla noder är anslutna med varandra via två trådsbussar som är nominellt partvinnade. Varje nod kan sända och ta emot meddelanden men aldrig samtidigt. Ett multimasterprincip förekommer i komunikationen mellan noderna och detta innebär att om en nod sänder meddelandet, så är de andra beredda att ta emot. Prioriteten i sändarens identitet avgör vilka som får sända först. Ett meddelande består av ett ID som representerar prioriteten hos meddelandet och utöver det finns 8 byte övrig minnesplats. De meddelanden som har det lägsta binära värdet på ID har högre prioritet.

2.8 CAN-shield

CAN-shield ger Arduino möjlighet att kommunicera via CAN-buss (se figur 2.6). Den används oftast för att till exempel få åtkomst till en fordonsdator och få fram information om kylvätsketemperatur, gasreglage och fordonshastighet. CAN-shield har två mikroprocessorer, MCP2515 CAN controller och MCP2551 CAN transceiver. De används för att skicka och ta emot meddelanden. Meddelandena består i det här fallet av de uppmätta sensorvärderna. För störningsdämpning i sladden vid överföringen av datan används ett 120 Ω

motstånd [10]. Det krävs vid längre kommunikationsbussar där överföringen kan påverkas av reflektion i sladden samt elektriska fält.



Figur 2.6: CAN-BUS Shield använder sig av Microchip MCP2515 CAN controller med MCP2551 CAN transceiver. [11].

2.9 Metropolis Monte Carlo metoden

Monte Carlo är benämningen för en familj av algoritmer för simulering med hjälp av slumptal. Dessa metoder har sitt ursprung på Los Alamos National Laboratory i USA där de testades på världens första elektroniska dator ENIAC [12].

Monte Carlo-metoder används i allmänhet för att genom slumpbaserade simuleringar generera fördelningar som är svåra att beräkna med traditionella lösningar av differentialekvationer. Istället för att lösa ett matematiskt problem som ofta i praktiken är olösbart simulerar man ett förlopp där resultatet går mot den fördelning man söker.

Metropolis-metoden går ut på att optimera en fördelning, exempelvis fördelningen av molekyler i en vätska, så att en kostnadsfunktion, till exempel interaktionsenergin mellan molekylerna, minimeras. Detta skulle i princip kunna skrivas som ett kopplat system av ett stort antal partiella differentialekvationer. För ett system med stort antal molekyler blir det mycket svårt att lösa. Metropolis-metoden går bakvägen genom att starta med en slumpmässig fördelning av molekyler och sedan vid varje tidssteg pröva att flytta på en slumpvis utvald molekyl. Vid varje steg beräknas den totala energin. Om den minskar till följd av en förflyttning behålls förändringen. Om

energin ökar flyttas molekylen tillbaka till sin tidigare position. Detta görs i ett stort antal tidssteg tills fördelningen konvergerar mot ett energiminimum.

Ett problem med Metropolis-metoden är att man till skillnad från vid lösning av differentialekvationerna inte kan vara säker på att det minimum man hittar är ett globalt minimum. För att öka sannolikheten för att man konvergerar mot ett globalt minimum istället för att fastna i ett lokalt minimum kastar man inte bort alla förändringar som leder till en ökning i kostnadsfunktionen. Istället behåller man dem med en viss sannolikhet som beror på ökningens storlek enligt en negativ exponentialfunktion:

$$P_{\text{behåll}} = e^{-\beta \Delta E} \tag{2.1}$$

där ΔE är förändringen i kostnadsfunktionen och β är en koefficient som bestämmer hur snabbt sannolikheten går mot noll som funktion av ΔE.

Här har Metropolis-metoden använts för att beräkna ett våglängsspektrum baserat på mätdata från tre optiska sensorer i olika våglängdsintervall. Varje sensor har en känd känslighetskurva som funktion av vågländ hos inkommande ljus. De tre sensorerna har också olika absolut känslighet (höjd på känslighetskurvorna). Givet sensorernas känslighetskurvor och ett givet spektrum kan sensorernas utsignaler förutsägas. En kostnadsfunktion kan definieras som skillnaden mellan förutsagda och uppmätta utsignaler. Optimeringen går ut på att under många iterationer modifiera spektrats intensitet vid en slumpvis utvald våglängd, beräkna kostnadsfunktionen och behålla förändringen om felet (kostnadsfunktionen) minskar. På så vis går simuleringen mot ett spektrum som minimerar skillnaden mellan uppmätta och förutsagda sensorvärden.

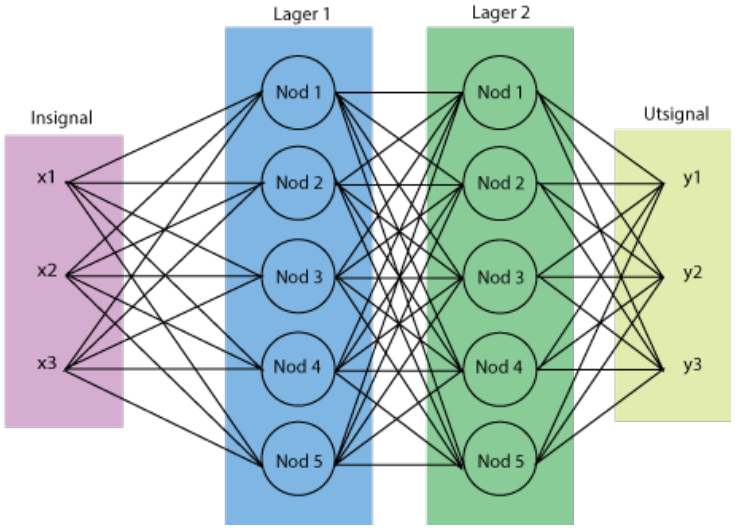
2.10 Artificiella neurala nätverk

Artificiella neurala nätverk (ANN) är ett samlingsnamn på ett antal självlärande algoritmer för informationsbehandling. Algoritmerna tillämpas i bland annat mönsterigenkänning, signalbehandling, reglerteknik och optimeringsproblem.

För att underlätta programmering av ANN finns det färdiga bibliotek att tillgå. Ett av dem är TensorFlow som är anpassat till Python och C++. Biblioteket är ursprungligen utvecklat av ingenjörer och forskare i Google Brain

team inom Googles forskningsorganisation [13] och gör det möjligt att distribuera beräkning till en eller flera CPU och GPU. Anledningen till användning av GPU är att grafikbehandlingsenheten kan hantera beräkningar snabbare på grund av sin parallella bearbetningsarkitektur [14].

Ett ANN består av insignaler, ett antal dolda lager samt utsignaler (se figur 2.7). Insignalen kan bestå av allt från handskrivna siffror, kamerasekvenser, fotografier på ansikten till mer fysikaliska parametrar som avstånd, nivåer och molekylkluster. Utsignalen är en klassificering av insignalen. Om till exempel insignalen till nätverket består av handskrivna siffror så kan utsignalen vara den digitala motsvarigheten.



Figur 2.7: Här visas schematiskt uppbyggnaden av ett ANN. Insignalen x, två lager med fem noder och utsignalen y.

På varje förgrening från insignalerna och noderna finns en tillhörande vikt som ändras vid träningen av nätverket. Träningen består av att nätverket matas med kända insignaler och utsignaler. Vikterna justeras så given insignal ska ge korrekt utsignal. Träningsdatamängden ska vara tillräckligt stor för nätverket och kan vid behov itereras flera gånger. Det är viktigt att träningsdatamängden är blandad, om all data är ordnad efter utsignalen kommer en överdriven viktning ske.

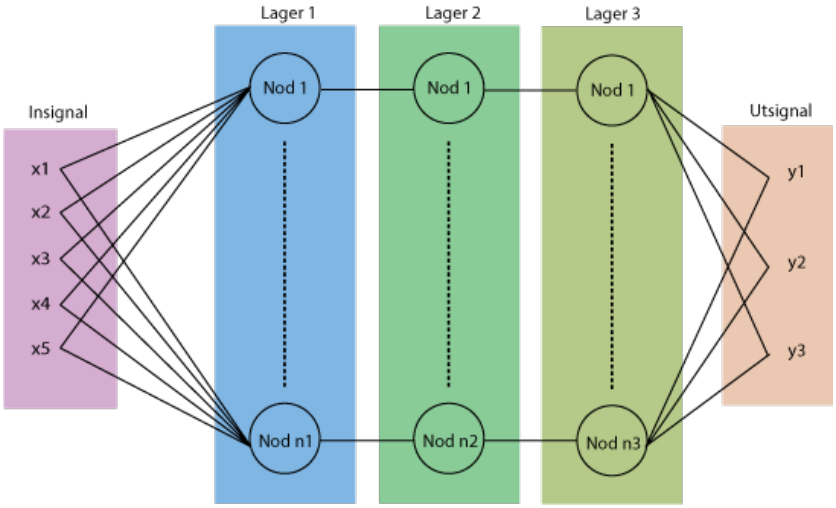
När nätverket tränas används en träningsalgoritm, i det här fallet backpropagationsalgoritmen. Algoritmen beräknar de enskilda vikternas fel med en förlust-funktion. Felvärderna passerar sedan nätverket baklänges från utgången tills varje nod har ett associerat felvärde som grovt representerar sitt

bidrag till det ursprungliga felet. Algoritmen använder sedan dessa felvärden för att uppdatera värdet på vikterna.

Antalet lager ett nätverk ska ha beror på hur problemet ser ut. Väljs för många lager kan systemet bli överbestämt och väljs för få kan de få en sämre noggranhet. I varje lager finns det ett antal noder och antalet kan variera mellan lagrerna.

I noderna finns en aktiveringsfunktion som skickar vidare värdet från noden som utsignal till nästa lager under förutsättningen att funktionens villkor är uppfyllda. Till exempel har en stegfunktion två villkor, 1 och 0, där noden aktiveras vid 1 och vilar vid 0.

Det artificiella neurala nätverket som klassificerar ljuset har en insignal, tre dolda lager och en utsignal (se figur 2.8). Insignalerna består av fem sensorvärdern och utsignalen av klassificeringen.



Figur 2.8: Här visas schematiskt uppbyggnaden ANN som ska klassificera ljuset. De fem insignalerna x_1 till x_5, tre lager med n_1, n_2 och n_3 antal noder samt utsignalen y med tre klasser.

Träningsdatamängden är uppbyggd av mätningar utförda vid olika tidpunkter under dagen och olika ljuskällor som delas upp i klasser.En mätningssession varar i ca 15 minuter med 2 ms mellan sampel.

Nätverket tränas med AdamOptimizer, en algoritm för första ordningens

gradientbaserad optimering av stokastiska funktioner [15]. Argumentet till TensorFlows AdamOptimizer är inlärningshastigheten som förhindrar att inlärningsförloppet blir instabilt. Hastigheten bestämmer hur snabbt det neurala nätverket ska förändra variablerna.

Aktiveringsfunktionen i lagrern är Rectified Linear Unit (ReLU) som är en rampfunktion och är defineras enligt

$$f(x) = max(0, x)$$

där x är insignalen till den specifika noden som aktiveras om x är positiv.

ReLU används ofta i ANN och är effektivare än sigmoidfunktionen (s-formad) samt den hyperboliska tangent-funktionen. ReLU är sedan 2015 den absolut vanligaste aktiveringsfunktionen för deep machine learning då den undviket problem med försvinnande gradienter som gör att inlärningen avstannar [16].

2.11 Sannolikhetsteori

Sannolikhetsteori är en avdelning inom matematiken och uppkom under 1600-talet där den tillämpades för tärningsspel [17]. Sannolikhetsdensitetsfunktionen (PDF) och den kumulativa fördelnings-funktionen (CDF) används till kurvanpassningen för sensorernas intensitetskurvor från databladen.

En stokastisk variabel är ett matematiskt uttryck för en variabel som beror av slumpen [18]. Det finns diskreta och kontinuerliga stokastiska variabler. De diskreta kan vara de två utfallen av ett mynt som singlas (krona eller klave) medan de kontinuerliga har en oändlig oupräknelig mängd tal.

PDF är en funktion som, vid varje stokastiskt värde i samplingsintervallet, ger en sannolikhetstäthet. Funktionen defineras enligt följande:

- Definerad på intervallet $[a, b]$ där $b > a$.
- Alltid positiv för alla $x \in [a, b]$
- Normaliserad, $\int_a^b f(x)dx = 1$

För en kontinuerlig stokastisk variabel specificerar PDF kvantiteten av sannolikheten för att en slumpmässig variabel x_i kommer att anta ett värde inom dx.

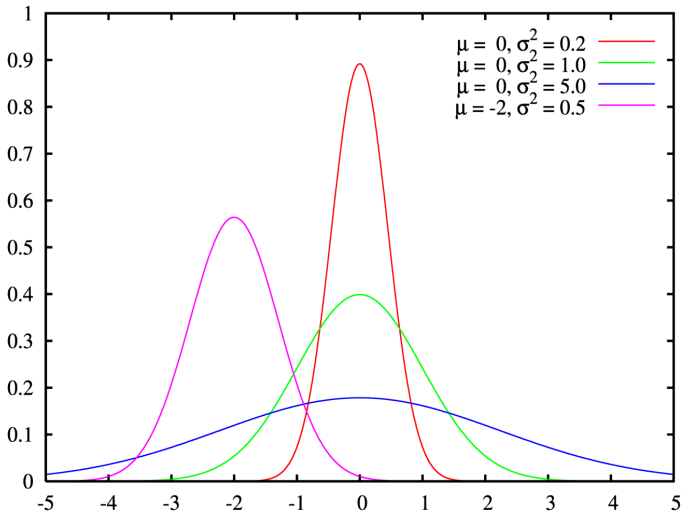
CDF beskriver en sannolikhetsfördelning för en slumpvariabel $F_X(x)$. Funktionen är monotont växande och beskriver sannolikheten att X är ett värde mindre eller lika med x.

Om den stokastiska variabeln X är kontinuerlig blir även $F_X(x)$ kontinuerlig vilket innebär att CDF kan uttryckas som arean under PDF till vänster om x.

$$F_X(x) = \int_{-\infty}^{x} f(t)dt$$

Gaussfördelningen, även kallas normalfördelningen (se figur 2.9), kan uttryckas med en PDF enligt ekvationen 2.2 där μ är medelvärdet av fördelningen, σ är standardavvikelsen och σ^2 är variansen.

$$\varphi(x) = \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.2}$$



Figur 2.9: Normalfördelningar med olika värden på σ och μ.

CDF av gaussfördelningen defineras enligt ekvation 2.3. Här integreras $\varphi(x')$ enligt definitionen för CDF med kontinuerliga stokastiska variabler.

$$\Phi(x) = \int_{-\infty}^{x} \varphi(x')dx' = \frac{1}{\sigma \cdot \sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{(x'-\mu)^2}{2\sigma^2}} dx' \tag{2.3}$$

För att få fram en skjuvad gaussfördelning används ekvation 2.4 där α ligger i intervallet $-\infty < \alpha < \infty$.

$$f(x) = 2\varphi(x)\Phi(\alpha x) \tag{2.4}$$

Parametern α styr åt vilket håll normalfördelningen är skjuvad och hur mycket. Om $\alpha < 0$ skjuvas fördelningen åt höger och är $\alpha > 0$ skjuvas den åt vänster, då $\alpha = 0$ är normalfördelningen oförändrad.

Kapitel 3

Metod

Här beskrivs tillvägagångsättet hur arbetet har lagts upp.

1. Kopplat sensorer till arduino
2. Skapat en CAN-buss med två noder bestående av arduino med CAN-shield och datorn med CanKing.
3. Spektrumanpassat med Metropolis Monte Carlo
4. Klassificerat med artificiella neurala nätverk

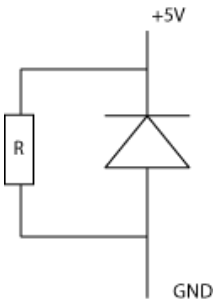
Det som inte framgår är att testning av hårdvara och mjukvara sker löpande under uppbyggnaden.

Mätningar görs med den färdiga kopplingen. De utförs i olika ljusförhållanden och under olika tider på dygnet. En mätning pågår mellan $1 - 20$ minuter med en samplingstid på 2 ms. Mätningarna av specifik ljuskälla görs i en låda för att stänga ute övrigt ljus. Solljuset mäts utomhus för att få med all UV-strålning. Ungefär 50 st mätningar har utförts kontinuerligt från mars till maj 2017 i Göteborg.

3.1 Hårdvarukonstruktion

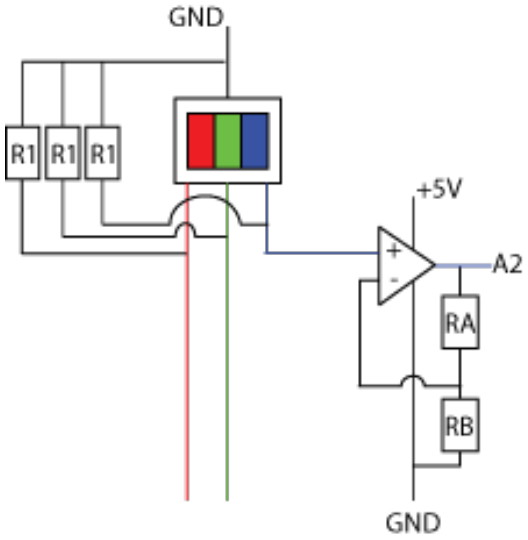
Hårdvarukonstruktionen består av kopplingen för sensorerna via Arduino med beräknade resistanser för önskad ström och CAN-Shielden. All matningsspänning som är 5 Volt kommer från Arduino.

För fotodiodens önskade ström på $I_{RA} = 100\mu$ A krävs en resistans $R = U/I = 56$ kΩ [4]. Motståndet kopplas mellan fotodiodens anod och GND (se figur 3.1).



Figur 3.1: Kopplingsschema för fotodioden.

RGB-sensorns fyra anoder parallellkopplas med resistans $R1$ enligt figur 3.2. Från databladet för RGB-sensorn fås sensorytans area $A = 0.94/3 \cdot 10^{-6} m^2$ [7]. Därefter beräknas effekten $P = I \cdot A$ där I för solljus är 1000 W/m^2. Effekten blir då P=3.13 $\cdot 10^{-4}$ W.



Figur 3.2: Kopplingsschema för RGB-sensorn.

Från diagrammet 2.5 fås A/W, där det största värdet 0.23A/W väljs. Med den utträknade effekten P fås strömmen över dioderna till $i = 72.067\mu$ A och därmed resistansen $R1 = 69380\ \Omega$ som enligt E-12 standard ger ett motstånd på 68 kΩ.

RGB-sensorns utsignal förstärks för lättare avläsning av värden, vilket görs med en ”rail-to-rail” MCP 6273 operationsförstärkare (OP). RGB-sensorns anoder (se figur 3.2) kopplas till en icke inverterande OP-koppling med resistanser RA $3.3k\Omega$ och RB $4.7k\Omega$ som ger en förstärkning $G = 1.7$.

Utgången från RGB-sensorns anoder genom OP-kopplingen ansluts till de analoga ingångarna A2, A3 och A4 på Arduinon.

UV-sensorn kopplas direkt till matningsspänning och GND [6] från Arduinon och till den analoga ingången A5.

CAN-shield kopplas på Arduinokortet och sensorvärderna skickas över CAN-bussen till datorn där de sparas i loggfiler.

3.2 Mjukvarukonstruktion

Här beskrivs hur mjukvaran är konstruerad. Mjukvaran består av programkod för Arduino med CAN-shield, Metropolis-programmet och det artificiella neurala nätverket.

3.2.1 Arduino och CAN-buss

I Arduinons programkod defineras sensorerna. Sensorernas värden kommer antingen digitalt eller analogt in på olika pins (ingångar) på Arduinokortet. Läsning från dessa sker med anrop enligt nedanstående exempel.

```
#define pinRed 3
red=analogRead(pinRed);
```

Här defineras pin 3 som är en analog ingång på arduinokortet. Därefter anropas analogRead-funktionen som läser av pin A3.

Värdet från sensorerna är ett värde mellan $0 - 1023$, 10 bitar. Eftersom ett meddelande i CAN-bussen består av 8 stycken 8 bitar delas sensorvärdet upp till två tal, de mest signifikanta i ett 8 bitars tal och de minst signifikanta i ett 2 bitars tal. Dessa skickas sedan i två meddelanden efter varandra, det första med längd 8 och det andra med längd 3.

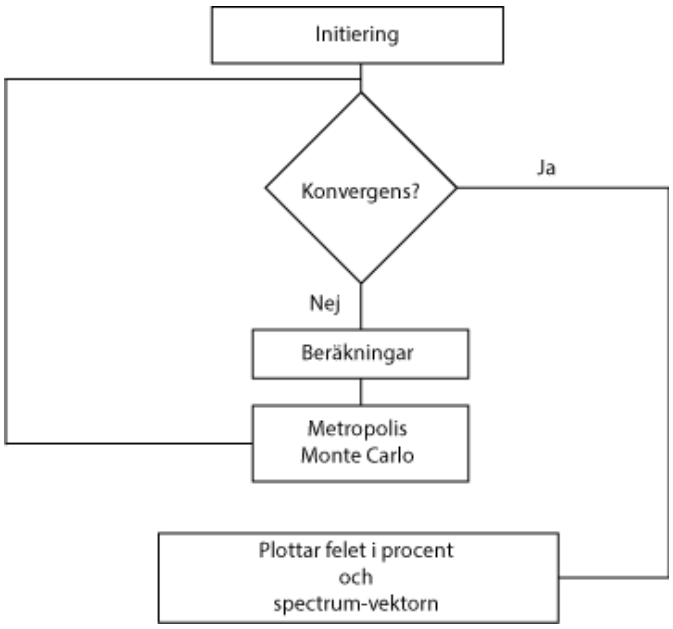
Med hjälp av biblioteken för CAN-bussen

```
#include <Canbus.h>
#include <defaults.h>
#include <global.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>
```

skickas två meddelanden. De två meddelandena skickas efter varandra med delay-funktion som fördröjer varje par meddelande med 2 ms. Programmet CanKing används för att ta emot meddelanden. Meddelandena loggas och sparas som textfiler i datorn. Hela programkoden för både Arduino och CAN-bussen återfinns i bilagan A.

3.2.2 Metropolis Monte Carlo

Efter att textfilen från CanKing lästs in och rätt värden placerats i rätt vektorer körs Metropolis Monte Carlo programmet. Innan exekveringen av programmet beräknas intensitetkurvornas vikter fram. Det görs med ett liknande program som, istället för att beräkna våglängdsspektrum, beräknar fram höjden för intensitetskurvorna genom en given solspektrumsvektor [19] (se programmet i bilaga B).



Figur 3.3: Flödesschema över Metropolis Monte Carlo programmet

Flödesschemat i figur 3.3 beskriver hur Metropolis-programmet är uppbyggt. Först initieras alla variabler. I while-loopen sker alla beräkningar tills avvikelsen konvergerat mot noll och därefter plottas spektrumanpassningen.

Känslighetskurvorna initieras som normalfördelningar. Kurvorna för blå och grön sensor ser ut som vanliga normalfördelningar och beräknas fram med hjälp av ekvationerna 2.2 och 2.3. Röd-, fotodiod- och UV-intensitetskurvorna ser däremot ut att vara skjuvade normalfördelningar och fås fram genom ekvation 2.2, 2.3 och 2.4. Programkoden för den röda sensorns intensitetskurva ser ut enligt följande:

```
redLambda = 620
width=10e3
alpha = 4

yr = np.exp((-(x - redLambda) ** 2) / width * 2)
pdfRed = np.exp(-((alpha * (x - redLambda)) ** 2) / width * 2)
redCDF = np.zeros(len(pdfRed))
redCDF[0] = pdfRed[0]

for i in range(1, len(yr)):
    redCDF[i] = redCDF[i - 1] + pdfRed[i]

redCDF /= redCDF[-1]
redCDF = 0.001 + redCDF

skewRed = 2 * yr * redCDF
skewRed /= np.max(skewRed)
```

där *redLambda* bestämmer maxima för normalfördelningen, *width* är bredden på basen och *alpha* är hur mycket den ska skjuvas. Värderna testas fram tills intensitetskurvan stämde överens med den ursprungliga kurvan. Med de tidigare framräknade vikterna justeras höjden på normalfördelningarna.

Sedan initieras en vektor som ska representera våglängder. Detta görs genom att fylla den med slumptal. Variabeln *OldDeviation* initieras till det största maximala felet för att säkerhetsställa att första avvikelsen alltid godkänns.

Därefter startas while-loopen med beräkningarna. En variabel *deviationStep* initieras med ett slumptal från en normalfördelning enligt

```
mu, sigma = 0,15
deviationStep = np.random.normal(mu, sigma, num_lambda)
```

Normalfördelningen har sitt centrum i origo $mu = 0$ och höjden $sigma = 15$ bestämmer hur stora slumptalen kan bli. Eftersom centrum ligger i origo kan både positiva och negativa slumptal placeras i *deviationStep*. Variabeln används sedan för att öka eller minska en våglängd i vektorn *spectrum*.

När den slumpade våglängden i *spectrum* förändrats beräknas konstnadsfunktionen *deviation*.

```
deviation = ((outRed-sensRed)**2 + (outBlue-sensBlue)**2 + (
    outGreen-sensGreen)**2 + (outInt-sensInt)**2 + (outUv-sensUv)
    **2)/(sensRed**2+sensBlue**2+sensGreen**2+sensInt**2+sensUv
    **2)
```

där *outRed*, *outBlue* och så vidare är summeringar av det nuvvarande vektorn *spektrum* multiplicerad med de tillhörande intensitetsgraferna. De andra variablerna *sensRed*, *sensBlue*, *sensGreen*, *sensInt* och *sensUV* är medelvärdet från mätningen för respektive sensor. Avståndet mellan *out*-värdet och *sens*-värdet beräknas. Beräkningen kan ses som den totala kostnaden till följd av förflyttningen i *spectrum*

Om *deviation* är större än *oldDeviation* (gamla värdet på kostnadsfunktionen) jämförs *deviation* med ett slumptal från den negativa exponentialfördelningen (se ekvation 2.1). Är *deviation* mindre än det slumpade värdet godkäns förändringen av *spectrum* och är felet större tas förändringen av *spectrum* tillbaka.

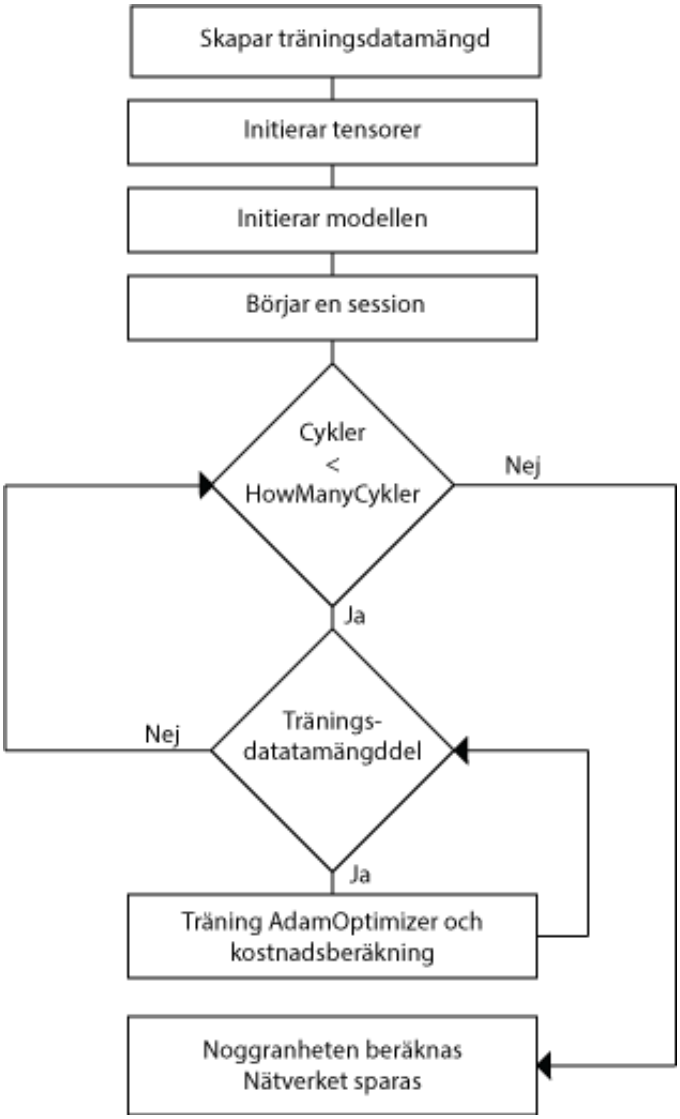
När kostnadsfunktionen konvergerat mot noll anses spektrumanpassningen vara färdig och våglängdsspektrumet plottas. Hela programmet återfinns i bilaga C.

3.2.3 Artificiella neurala nätverkens uppbyggnad

Tre artificiella neurala nätverk har konstruerats. Nätverken skiljer sig åt genom att utsignalsvektorn ger olika klassificeringar. Programmeringskoden för de tre nätverken börjar med biblioteksimportering av TensorFlow och bibliotekets funktioner anropas med *tf*. När programmet sedan kompilerar existrerar inte nätverken utanför sessionen *sess*. Alla variabler och modellen initieras innan men skapas och existerar endast under sessionens existens.

Programmet är indelat i två delar, den första initierar nätverken och tränar dem. Efter träningen sparas alla variabler.

I figur 3.4 beskrivs hur programmet skapar träningsdata, initierar nätverken och tränar dem. Först skapar programmet träningsdatamängden genom inläsning av datafiler innehållande mätdatan. Här väljs även klassificeringen beroende på vilken mätningsdata som används.



Figur 3.4: Flödesschema över programkoden för det artificiella neurala nätverket.

Sedan skapas tensorer, de är TensorFlows vektorer. Vektorerna består av insignalen x och utsignalen y och initieras genom att skapa *placeholders* enligt

```
x=tf.placeholder("float32",[None,inputSize])
y=tf.placeholder("float32")
```

där *placeholder* är en tom tensor och argumentet *inputSize* för x *placeholder* representerar antalet element i tensorn som här motsvarar antalet sensorvärden. Tensorn för utsignalen y har lämnats utan bestämd storlek eftersom utsignalsvektorn kommer variera för de olika nätverken.

I initieringen bestäms även antalet noder och lager. Antalet noder testas fram för att få en så hög noggrannhet (*accuracy*) som möjligt. Noggrannheten är ett mått på hur effektivt nätverket klassificerar rätt.

När träningen körs skrivs antalet cykler och vilken cykel som är gjord tillsammans med kostnaden. Kostnaden är ett mått på hur väl nätverket konvergerar och bör minska mellan cyklerna. Utskriften kan se ut enligt följande.

```
Cycle  0  completed out of  10 Loss:  59345.9013953
Cycle  1  completed out of  10 Loss:  732.618157417
Cycle  2  completed out of  10 Loss:  721.24370569
Cycle  3  completed out of  10 Loss:  716.481703103
Cycle  4  completed out of  10 Loss:  714.291466683
Cycle  5  completed out of  10 Loss:  710.151163578
Cycle  6  completed out of  10 Loss:  699.413550645
Cycle  7  completed out of  10 Loss:  695.827997997
Cycle  8  completed out of  10 Loss:  695.292057484
Cycle  9  completed out of  10 Loss:  694.659807473
Accuracy:  0.908839
```

I den andra delen av programmet anropas variablerna för testning och körning. På det här sättet slipper man träna nätverket varje gång då det kan ta väldigt lång tid. Filen öppnas i en ny session och ges insignalsvärden från en mätning. Därefter skriver nätverket ut sin klassificering. Eftersom nätverken endast existerar inom en session kan flera nätverk enkelt öppnas och köras efter varandra för att få utförligare information om ljusmätningen. Hela programkoden för nätverket återfinns i bilaga D.

Kapitel 4

Resultat

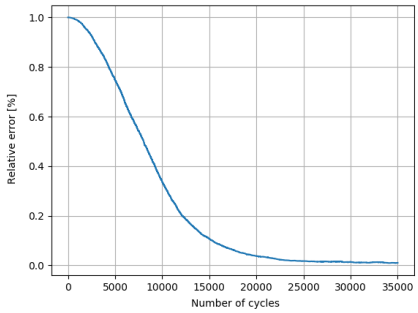
Här presenteras resultatet från Metropolis Monte Carlo programmet samt det artificiella neurala nätverket.

4.1 Metropolis Monte Carlo

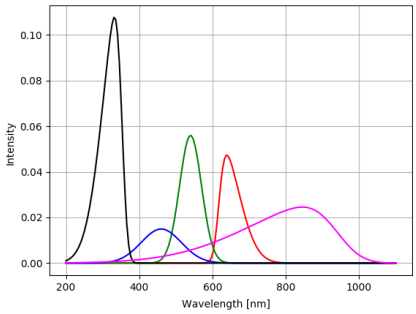
Kalibreringen av höjderna på intensitetskurvorna som programmet optimerar mot beräknades fram till

```
[0.68814022  0.40765524  0.83726203  1.55842677  1.50851575]
```

som motsvarar vikten för röd, grön, blå, intensiteten och UV. Se figur 4.1a för den konvergerande avvikelsen och figur 4.1b för de kalibrerade intensitetskurvorna.



(a) Avvikelsen i % som konvergerar mot 0 över antalet cykler.

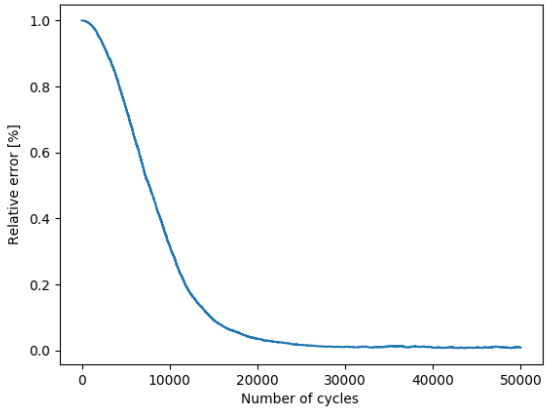


(b) Svart: UV, blå, grön, röd: RGB-sensorn. Magenta: fotodioden.

Figur 4.1: (a) Konvergerande avvikelsen vid beräkningen för vikterna. (b) De viktade intensitetskurvorna.

De kalibrerade intensitetskurvorna skiljer sig från databladens intensitetskurvor (jämför figur 4.1b med figurerna 2.1, 2.2b och 2.5). Det beror antingen på att kalibreringen kan förbättras eller att databladens intensitetskurvor är missvisande.

Vid körning av programmet är det viktigt att avvikelsekurvan konvergerar mot 0%. För de olika mätserierna som gjordes varierar det slutliga felet mellan $0 - 10\%$. Om kurvorna konvergerar mot något annat än 0% skapas en osäkerhet för hur väl spektrumanpassningen överrensstämmer med det verkliga spektrumet för ljuset enligt intensitetskurvorna.

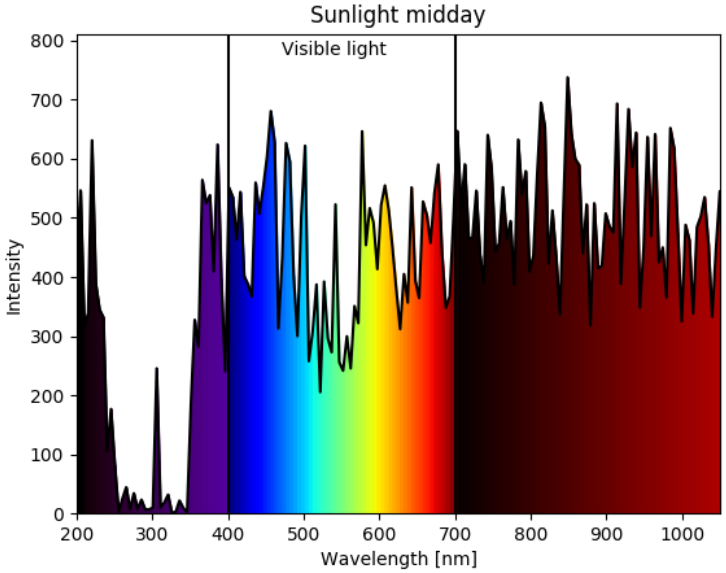


Figur 4.2: Kurvan för hur felet konvergerar. Y-axeln är i procent och x-axeln visar på antalet iterationer.

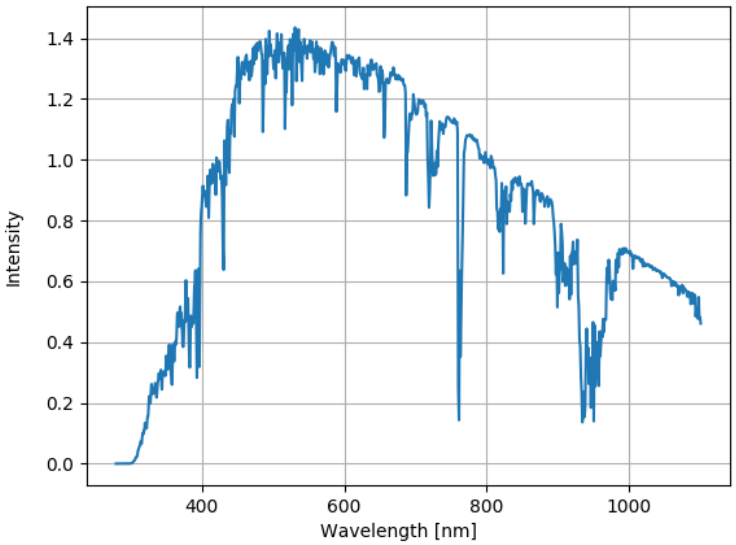
Vid starkt solljus mitt på dagen konvergerar felet (se figur 4.2) till 0%. Spektrumanpassningen plottades upp (se figur 4.3) och vid jämförelse med ett verkligt solspektrum (se figur 4.4) från ASTM [19] finns det både likheter och olikheter.

UV-ljuset stämmer ganska väl överens från våglängd 300 nm. Innan ökar UV-ljuset i spektrumanpassningen vilket tyder på att sensorn och dess intensitetskurva inte ger tillräcklig information om våglängder mindre än 300 nm.

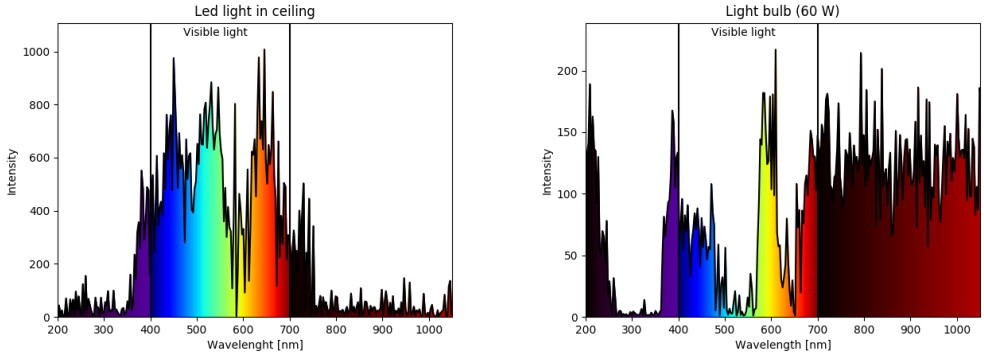
IR-strålningen i spektrumanpassningen stämmer inte överens med den verkliga IR-strålningen som finns i solljus. Det kan bero på, precis som för UV-strålningen, att sensorerna ger för lite information om ljuset efter 750 nm.



Figur 4.3: Plott över det beräknade våglängdspektrumet. Mätningen gjort klockan 12.00 i direkt solljus den 8 maj 2017.



Figur 4.4: Plott på datan från ASTM [19] som visar solljusspektrum mitt på dagen.



(a) Spektrumet för ledlampan.

(b) Spektrumet för glödlampan

Figur 4.5: Våglängdsspektrum för en kallvit ledlampa och en 60 Watt glödlampa.

För en mätning på en ledlampa såg våglängdsspektrumet ut enligt figur 4.5a. Där syns det tydligt att ledlampan inte har någon IR-strålning vilket är rimligt. Ledlampor blir inte varma till skillnad från glödlampor (se figur 4.5b). Det är betydligt mer IR-strålning från glödlampan och färre våglängder. Det är det gula och röda ljuset som dominerar i spektrumet vilket stämmer överens med verkligheten. Glödlampan hade ett varmare sken i jämförelse med ledlampan.

4.2 Artificiella neurala nätverk

Ett antal artificiella neurala nätverk skapades för de olika klassificeringarna. Detta för att kunna återanvända mätningar som gjorts. Till exempel används en mätning i direkt solljus gjord på förmiddagen för klassificeringen mellan inomhusbelysning och utomhusljus samt för klassificeringen morgonljus och eftermiddagssljus.

Det artificiella neurala nätverket byggdes upp med tre lager. Lagrens noder varierade för de olika nätverken. Dess antal prövades fram tills noggrannheten hos nätverket verkade nå sitt maximum. Antalet för de tre nätverken hamnade mellan $50 - 250$ st noder. Inlärningshastigheten justerades så att nätverket inte blev instabilt. Värdet på hastighetsparametern till AdamOptimizer låg mellan $0.01 - 0.0001$ beroende på nätverket som tränades. Ett sätt att förbättra träningen och minska tiden att träna ett nätverk hade varit att börja med en hög inlärningshastighet för att sedan minska den kontinuerligt under träningens gång. AdamOptimizer har en konstant inlärningshastighet

vilket innebär att det för svårare klassificeringsfall, tar otroligt lång tid att träna ett nätverk.

Antalet cykler som kördes under träningen anpassades också efter träningen. Under träningen plottades kostnadsförminskningen ut och vilken cykel av totala antalet. Nedan ses ett exempel på programutskriften under träningen.

```
Cycle  0  completed out of  15 Loss:  7260.21477067
Cycle  1  completed out of  15 Loss:  43.8826878071
Cycle  2  completed out of  15 Loss:  33.8420089781
            .
            .
            .
Cycle  13  completed out of  15 Loss:  25.3979098797
Cycle  14  completed out of  15 Loss:  25.4753010571
Accuracy:  0.907647
```

Fördelen med att skriva ut lite data under träningen är helt för sin egen skull. Det tar lång tid att träna nätverken och för att veta ungefär var i träningen nätverket befinner sig för tillfället underlättar. Utskriften av kostnadsförminskningen ("Loss") sker för att avgöra om nätverket blivit överbestämt. Ökar "Loss" har nätverket blivit överbestämt och kommer därmed inte nå sin maximala noggrannhet. I programutskriften kan man se att nätverket blev överbestämt mellan cykel 13 och 14 där "Loss" ökade med 0.0773911774.

De klassificeringar vars mätvärden till träningsdatamängden redan från början såg olika ut, till exempel mätning mellan inomhusbelysning och utomhusbelysning, behövde färre noder i lagrerna och färre cykler i träningen. Däremot var klassificeringen mellan olika solljus under dagen en krävande utmaning för nätverket. Där krävdes många noder och fler cykler för att få en noggrannhet över 90% på klassificeringen.

För klassificering av ljuset fungerade det artificiella neurala nätverket bra. Det kan skilja på inomhusbelysning och utomhusljus med 99.3% säkerhet. Nätverket skiljer även på förmiddagsljus och eftermiddagsljus med 90.8% säkerhet och kan se skillnaden på varm och kall ledbelysning med 96% säkerhet.

Kapitel 5

Slutsats och Diskussion

Problemet med spektrumanpassningen är att den fungerar bra till kontinuerligt ljus. Vid diskontinuerligt ljus som till exempel laser kommer Metropolisprogrammet ge ett felaktigt spektrum.

Spektrumanpassningen kan förbättras genom fler sensorer. Sensorerna kan fortfarande vara billiga men bör ha intensitetsmaximum på andra våglängder.

Exempel på en dioder skulle kunna vara IR-dioden SMD, PD70-01B/TR10 som har ett känslighetsmaximum på 940 nm och kostar omkring $8-9$ kronor.

Eventuellt om RGB sensor består av en större optisk area med fler pixlar, kan man få fram noggrannare mätningar av våglängderna inom det synliga ljusets spektra.

Klassificeringen med det artificiella neurala nätverket fungerar bra. Den har relativt hög noggrannhet som kan ökas ytterligare om mer tid läggs ned på mätningar till träningsdatamängden. Då kan fler nätverk skapas med andra klassificeringar som till exempel billjus och gatulampor eller solljus på våren, sommaren, hösten och vintern.

Att samla in träningsdata och träna nätverken är väldigt tidskrävande. En träning kan ta några minuter till flera timmar och insamlandet av träningsdata tar flera dagar. Med det i åtanke lyckades de artificiella neurala nätverken över förväntan.

5.1 Framtida utvecklingar

Den billiga sensorbaserade spektrometern fungerar för privatpersoner och består framförallt av vanliga billiga sensorer, sensorer som mycket väl kan återvinnas från gamla uttjänta elektroniska produkter. Elektroniskt avfall återvinns inte i dagsläget till 100 % utan det mals ned för extrahering av de dyrbaraste metallerna. Genom att plocka ut kamerasensorerna och fotodioder från bland annat gamla mobiler minskar elavfallet vilket i sin tur bidrar till en hållbar utveckling.

Det byggs allt fler växthus i vårt urbaniserade samhälle för att uttnytja olika ytor i städer till odling. Dessa växthus har sollampor för att förse växterna med det livsnödvändiga ljuset. För att spara på energi kan man enkelt, med spektrometern, anpassa lamporna som belyser växterna med de våglängder omgivningsljuset saknar och som plantorna behöver [20].

De stora korallreven i världshaven spelar en stor roll i ekosystemet. De visar en positiv tillväxt om de utsätts för blått och rött ljus. Genom att belysa korallerna med våglängderna som fattas kan en större tillväxt ske vilket förbättrar hela havens ekosystem.

Litteraturförteckning

[1] I. G. Morgan K. A. Rose. "Outdoor activity reduces the prevalence of myopia in children". *Ophtalomology*, 115:1279–1285, 2008. [Online]. Tillgänglig: http://www.sciencedirect.com/science/article/pii/S0161642007013644. Hämtad: 2017-04-11.

[2] Grafisk-Handel. "gl spectis 1.0 mini-spectrometer". [Online]. Tillgänglig: http://www.grafisk-handel.se/shop/gl-spectis-1-5989p.html?gclid=CKvA7tHw99MCFUW1GAodgywDdA. Hämtad: 2017-05-17.

[3] K. Nilson och J. Pålsgård, G. Kvist. *Ergo Fysik 2*. Liber AB, 2012.

[4] *Silicon PIN Photodiode, VBP104S*, Malvern, Pennsylvania, USA: Vishay Semiconductors, 2011. [Online]. Tillgänglig: http://www.vishay.com/docs/81170/vbp104sr.pdf. Hämtad: 2017-01-18.

[5] *Si Photodiodes Lineup of Si Photodiodes for UV to near IR, radiation*, Hamamatsu, 2017. [Online]. Tillgänglig: https://www.hamamatsu.com/resources/pdf/ssd/si_pd_kspd0001e.pdf. Hämtad: 2017-01-18.

[6] *Seeed studio Grove UV sensor*,Seeed, 2017. [Online]. Tillgänglig: http://www.mouser.com/catalog/specsheets/Seeed_101020043.pdf. Hämtad: 2017-01-19.

[7] *Si photodiode: S9702 RGB color sensor, Japan: Hamamatsu, 2016.* Hämtad: 2017-02-07.

[8] H. B. Mitchell. *Multi-sensor data fusion: an introduction.* Berlin, Tyskland: Springer Science & Business Media, 2007. [Online]. Tillgänglig: https://books.google.se/books?hl=sv&lr=&id=2hwcFSxQ1CAC&oi=fnd&pg=PA3&dq=Multi-sensor+fusion:+fundamentals+and+applications+with+software&ots=eM1My7dp2P&sig=ToecmQvOvSR1_

BlCSyZi9bk7K4Q&redir_esc=y#v=onepage&q&f=false.
Hämtad: 2017-05-15.

[9] W. Buchanan. *Computer Busses*. Uppl. 1, San Diego, USA: Butterworth-Heinemann, 2000. [Online]. Tillgänglig: http://www.sciencedirect.com.proxy.lib.chalmers.se/science/book/9780340740767. Hämtad: 2017-05-15.

[10] Can bus automatic line termination av R. E. Dozier, J. S. Beam och E. Topp, (3 december 2013). US Patent 8,597,054 [Online]. Tillgänglig: https://www.google.ch/patents/US20120231663 Hämtad: 2017-04-18.

[11] *MCP2515*, Microchip Technology, 2005. [Online]. Tillgänglig: http://ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf. Hämtad: 2017-01-19.

[12] C. Robert & G. Casella. "a short history of markov chain monte carlo: subjective recollections from incomplete data". *Statistical Science*, 2011. [Online]. Tillgänglig: http://projecteuclid.org.proxy.lib.chalmers.se/download/pdfview_1/euclid.ss/1307626568. Hämtad: 2017-03-02.

[13] Google Brain team. "Tensorflow", 2017. [Online]. Tillgänglig: https://www.tensorflow.org/. Hämtad: 2017-04-15.

[14] S. Music. *"Grafikkort till parallella beräkningar"*. examensarbete för kandidatexamen, Institution för Datavetenskap, Malmö högskola, Malmö, Sverige, 2012. [PDF]. Tillgänglig: https://dspace.mah.se/handle/2043/16799. Hämtad: 2017-02-28.

[15] J. Ba D. P. Kingma. *"Adam: A Method for Stochastic Optimization"*. Conference paper, 3rd International Conference for Learning Representation, San Diego, USA, 2015. [Online]. Tillgänglig: https://arxiv.org/abs/1412.6980. Hämtad: 2017-03-27.

[16] Yoshua Bengio & Geoffrey Hinton Yann LeCun. "deep learning". *Nature*, 2015. [Online]. Tillgänglig: http://search.proquest.com.proxy.lib.chalmers.se/docview/1685003444/fulltextPDF/98F775AB6F994146PQ/1?accountid=10041. Hämtad: 2017-05-21.

[17] A. A. Borovkov. *Probability Theory*. London, England: Springer London, 2013. [Online]. Tillgänglig: https://link-springer-com.proxy.lib.chalmers.se/book/10.1007%2F978-1-4471-5201-9. Hämtad: 2017-05-07.

[18] J. K. Shultis W. L. Dunn. *Exploring Monte Carlo methods*. Elsevier/Academic Press, 2012;2011;. [Online]. Tillgänglig: http://www.sciencedirect.com.proxy.lib.chalmers.se/science/book/9780444515759. Hämtad: 2017-03-27.

[19] "Solar spectrum american society for testing and materials (astm) terrestrial reference spectra for photovoltaic performance evaluation". [Online]. Tillgänglig: http://rredc.nrel.gov/solar/spectra/am1.5/. Hämtad: 2017-04-18.

[20] Karl-Johan Bergstrand H. K. Schüssler. "LED-teknik för assimilationsbelysning: Energibesparing och växtstyrning", 2009. [Online]. Tillgänglig: http://vaxthusljus.se/Slutrapport%20SLF%20LED%202009%202010.pdf. Hämtad: 2017-05-20.

Bilaga A

Arduino och CAN-buss

```
/***********
Librarys
***********/
#include <Canbus.h>
#include <defaults.h>
#include <global.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>
/***************
 Definitioner
***************/
tCAN message;

#define IRPin 1
#define photoRPin 0
#define pinRed 3
#define pinBlue 2
#define pinGreen 4
#define pinUV 5

unsigned int rawLight;
unsigned int red;
unsigned int blue;
unsigned int green;
unsigned char light1;
unsigned char light2;
unsigned char red1;
unsigned char red2;
unsigned char blue1;
unsigned char blue2;
unsigned char green1;
unsigned char green2;
unsigned char uv;
```

```
unsigned char IR;

bool canBusSend(tCAN);
/*************
 Set up loop
 *************/
void setup() {
 Serial.begin(9600);

 //Setup canbus baud rate
  if(Canbus.init(CANSPEED_500))   //MCP2515 CAN speed
    Serial.println("CAN Init ok");
  else
    Serial.println("Can't init CAN");

  delay(1000);
}

/**************
 Main loop
 **************/
void loop(){

//Values from sensors
 rawLight=analogRead(photoRPin);
 red=analogRead(pinRed);
 blue=analogRead(pinBlue);
 green=analogRead(pinGreen);
 uv=analogRead(pinUV);
 IR=analogRead(IRPin);

 light1=rawLight>>2;
 light2=rawLight&0x03;

 red1=red>>2;
 red2=red&0x03;

  green1=green>>2;
 green2=green&0x03;

 blue1=blue>>2;
 blue2=blue&0x03;

 //Send the results to Serial port (printing in seriel monitor)
 Serial.print(String(" ")+int(red)+String(" ")+ int(green)+
    String(" ")+ int(blue)+ String(" ")+ rawLight +String(" ")+
    int(uv) +"\n");

/*********************
```

```
 CAN WRITE
 ********************/

  message.id = 0x01;
  message.header.rtr = 0;
  message.header.length = 8;

  message.data[0] = red1;
  message.data[1] = red2;
  message.data[2] = green1;
  message.data[3] = green2;
  message.data[4] = blue1;
  message.data[5] = blue2;
  message.data[6] = light1;
  message.data[7] = light2;
  message.data[8] = 0x0;
  message.data[9] = 0x0;
  message.data[10] = 0x0;

  canBusSend(message);

    message.id = 0x01;
    message.header.rtr = 0;
    message.header.length = 3;
    message.data[0] = uv;
    message.data[1] = IR;
    message.data[2] = 0x0;
    canBusSend(message);
/*Serial.println("Returkod:");
Serial.println(messSend);
*/
/********************
 CAN READ
 ********************

if (mcp2515_check_message())
  {
    if (mcp2515_get_message(&message))
  {
        //if(message.id == 0x620 and message.data[2] == 0xFF)
           //uncomment when you want to filter
             //{

               Serial.print("ID: ");
               Serial.print(message.id,HEX);
               Serial.print(", ");
               Serial.print("Data: ");
               Serial.print(message.header.length,DEC);
               for(int i=0;i<message.header.length;i++)
```

```
                {
                  Serial.print(message.data[i],HEX);
                  Serial.print(" ");
                }
               Serial.println("");
             //}
           }}
 */
 //slow down the transmission for effective Serial communication
     .
 delay(1);
}

bool canBusSend(tCAN message){
  mcp2515_bit_modify(CANCTRL, (1<<REQOP2)|(1<<REQOP1)|(1<<REQOP0
      ), 0);
  int messSend=mcp2515_send_message(&message);
  if(messSend){
    return true;
  }else{
    return false;
    }
}
```

sensorljus.ino

Bilaga B

Beräkning av vikterna för intensitetskurvorna, Pythonprogramkod

```
""" ----------------- LIBRARIES ------------------- """

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.colors as colors
import matplotlib.cm as cmx

""" -----------------DEFINITIONS------------------- """

red = []
green = []
blue = []
intensity = []
uvLight = []
IRLight = []

num_sensor=5
maxValueSensor=1024
num_lambda = 300

# Create initial weights (heights) for the sensitivity functions
weights = np.ones(num_sensor)
# weight order: red, green, blue, intensity, uv


#Wavelenghts (x-axis)
x = np.linspace(200, 1100, num_lambda)
```

```
""" ---------------- FUNCTIONS --------------------"""


# readFile takes the string-argument ant opens the file,
# reads column by column, line by line and puts the
# values in right list (array).
#
# Input argument: string, name of textfile.
# Return: None
def readFile(filename):
    f = open(filename, "r")  # Opens the datafile
    firstLine = f.readline()  # First line in the textfile is
       rubbish. TOSS IT AWAAAY
    whichLine = False
    for line in f:  # Loops through the text file line by line
        i = 0
        whichLine = not whichLine
        if whichLine == True:
            for word in line.split():  # Splits the line word by
                word and puts the values in the right array
                if i == 2 and int(word) == 3:
                    whichLine = False
                    i = 0
                    break
                if i == 3:
                    temp = int(word) << 2
                elif i == 4:
                    red.append(temp + int(word))
                elif i == 5:
                    temp = int(word) << 2
                elif i == 6:
                    green.append(temp + int(word))
                elif i == 7:
                    temp = int(word) << 2
                elif i == 8:
                    blue.append(temp + int(word))
                elif i == 9:
                    temp = int(word) << 2
                elif i == 10:
                    intensity.append(temp + int(word))  # Concat
                        the 10 bit value of the light intensity
                       to one int
                i += 1
        if whichLine == False:
            for word in line.split():
                if i == 2 and int(word) == 8:
                    whichLine = True
                    break
```

```
                if i == 3:
                    uvLight.append(int(word))
                elif i == 4:
                    IRLight.append(int(word))
                i += 1
    f.close()

def uvGraf(x):
    global uv
    uvPeak = 355
    alpha = 4
    width = 5e3
    pdf = np.exp((-(x - uvPeak) ** 2) / width)
    pdfAlpha = np.exp(-((alpha * (x - uvPeak)) ** 2) / width)

    yCDF = np.zeros(len(pdfAlpha))
    yCDF[0] = pdfAlpha[0]

    for i in range(1, len(pdf)):
        yCDF[i] = yCDF[i - 1] + pdfAlpha[i]

    yCDF /= yCDF[-1]
    yCDF = 1 - yCDF
    uv = 2 * pdf * yCDF
    uv *= weights[4]/np.max(uv)
    return uv

def redGreenBlue(x):
    redLambda = 620
    blueLambda = 460
    greenLambda = 540
    width=10e3
    alpha=4
    global yr
    global yg
    global yb
    yred = weights[0]*np.exp((-(x - redLambda) ** 2) / (width *
        2))
    pdfRed = np.exp(-((alpha * (x - redLambda)) ** 2) / width *
        2)
    redCDF = np.zeros(len(pdfRed))
    redCDF[0] = pdfRed[0]

    for i in range(1, len(yred)):
        redCDF[i] = redCDF[i - 1] + pdfRed[i]

    redCDF /= redCDF[-1]
    redCDF = 0.001 + redCDF
```

```
    yr = 2 * yred * redCDF
    yr /= np.max(yr)

    yb = weights[1]*np.exp((-(x - blueLambda) ** 2) / (width
        *0.6))

    yg = weights[2]*np.exp((-(x - greenLambda) ** 2) / (width
        *0.18))

def intensityGraf(x):
    global skewPdf
    intensityPeak = 940
    alpha = 4
    width = 10e4

    yi = np.exp((-(x - intensityPeak) ** 2) / width)
    pdfAlpha = np.exp(-((alpha * (x - intensityPeak)) ** 2) /
        width)
    yCDF = np.zeros(len(pdfAlpha))
    yCDF[0] = pdfAlpha[0]

    for i in range(1, len(yi)):
        yCDF[i] = yCDF[i - 1] + pdfAlpha[i]

    yCDF /= yCDF[-1]
    yCDF = 1 - yCDF

    skewPdf = 2 * yi * yCDF
    skewPdf *= weights[3]/np.max(skewPdf)
    return skewPdf

def datasheetfunctions(x):
    global yr
    global yg
    global yb
    uv=uvGraf(x)
    redGreenBlue(x)
    skewPdf=intensityGraf(x)

    plt.plot(x, yr, color="red")
    plt.plot(x, yg, color="green")
    plt.plot(x, yb, color="blue")
    plt.plot(x, uv, color="black")
    plt.plot(x, skewPdf, color="magenta")
    plt.grid()
    plt.show()

def spectrumToOutput(spectrum):
    global spred
```

```
    spred = np.sum(spectrum * yr * weights[0])
    global spblue
    spblue = np.sum(spectrum * yb * weights[1])
    global spgreen
    spgreen = np.sum(spectrum * yg * weights[2])
    global spint
    spint = np.sum(spectrum * skewPdf * weights[3])
    global spuv
    spuv = np.sum(spectrum * uv * weights[4])

def plot_curves():

    light=plt.plot(red[0:],'r')
    light=plt.plot(green[0:],'g')
    light=plt.plot(blue[0:],'b')
    light=plt.plot(intensity[0:],'m')
    light=plt.plot(uvLight[0:],'k')
    light=plt.plot(IRLight[0:],'c')
    plt.show()


"""----------------CALL ON MEEEE----------------"""

def main():
    readFile("1145till12soligdag8maj1.txt")

    sensRed = np.sum(red) / len(red)
    sensGreen = np.sum(green) / len(green)
    sensBlue = np.sum(blue) / len(blue)
    sensInt = np.sum(intensity) / len(intensity)
    sensUv = np.sum(uvLight) / len(uvLight)

    sensorSum = (np.sum(red) + np.sum(blue) + np.sum(green) + np
        .sum(intensity) + np.sum(uvLight))/len(red)

    # Read in x and y values from solar spectrum
    x_sun_list = []
    y_sun_list = []
    f = open('solar_spectrum','r')
    for line in f.readlines():
        entries = line.split()
        x_sun_list.append(float(entries[0]))
        y_sun_list.append(float(entries[1]))
    x_sun = np.array(x_sun_list)
    y_sun = np.array(y_sun_list)

    plt.plot(x_sun,y_sun)
    plt.grid()
    plt.show()
```

```
# Create sensitivity functions using x vector from solar
    spectrum
datasheetfunctions(x_sun)

# Start Monte Carlo loop to find optimal sensor weights
global weights
# Initiates a large value to the old deviation to guarantee
    first trial is accepted
oldDeviation = (num_sensor*maxValueSensor)**2
converge = False
num_trials = 0
# List for storing deviations
epsilon0 = []
while not converge:
    num_trials += 1
    # Select a random weight to make a trial change to
    w_i = np.random.randint(num_sensor)
    # Draw trial step from normal distribution
    mu, sigma = 0, 0.01
    step = np.random.normal(mu, sigma)
    # Perform trial step
    weights[w_i] += step
    # Check that weight is still positive
    while weights[w_i] < 0:
        # reverse the step
        weights[w_i] -= step
        # pick a new random step
        step = np.random.normal(mu, sigma)
        # Perform new trial step
        weights[w_i] += step
    # Predict sensor output given the trial change
    spectrumToOutput(y_sun)

    # Calculate new deviation
    outputSum = spred + spblue + spgreen + spuv + spint
    scaleValue = outputSum / sensorSum

    outRed = spred * scaleValue
    outBlue = spblue * scaleValue
    outGreen = spgreen * scaleValue
    outInt = spint * scaleValue
    outUv = spuv * scaleValue

    deviation = ((outRed-sensRed)**2 + (outBlue-sensBlue)**2
        + (outGreen-sensGreen)**2 + (outInt-sensInt)**2 + (
       outUv-sensUv)**2)/(sensRed**2+sensBlue**2+sensGreen
       **2+sensInt**2+sensUv**2)

    # Decide whether to keep the change
```

```
            if deviation > oldDeviation:
                exp_parameter = 0.001
                if deviation - oldDeviation > np.random.exponential(
                    exp_parameter):
                    # Reject the trial change
                    weights[w_i] -= step

            oldDeviation = deviation

            # Record current deviation
            epsilon0.append(deviation)

            if num_trials > 1e5:
                converge = True

        # Rescale weights to stay on the order of 1
        weights /= np.mean(weights)
        print(weights)

        plt.plot(epsilon0)
        plt.show()


        import sys
        sys.exit()

main()
```

intensityWeight.py

Bilaga C

Metropolis Monte Carlo, Pythonprogramkod

```
""" ----------------- LIBRARIES ------------------- """

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.colors as colors
import matplotlib.cm as cmx

""" -----------------DEFINITIONS------------------- """

red = []
green = []
blue = []
intensity = []
uvLight = []
IRLight = []

num_sensor=5
maxValueSensor=1024
num_lambda = 300

#Wavelenghts (x-axis)
x = np.linspace(200, 1100, num_lambda)

""" ----------------- FUNCTIONS -------------------- """


# readFile takes the string-argument ant opens the file,
# reads column by column, line by line and puts the
# values in right list (array).
#
```

```
# Input argument: string, name of textfile.
# Return: None
def readFile(filename):
    f = open(filename, "r")  # Opens the datafile
    firstLine = f.readline()  # First line in the textfile is
       rubbish. TOSS IT AWAAAY
    whichLine = False
    for line in f:  # Loops through the textfile line by line
        i = 0
        whichLine = not whichLine
        if whichLine == True:
            for word in line.split():  # Splits the line word by
                word and puts the values in the right array
                if i == 2 and int(word) == 3:
                    whichLine = False
                    i = 0
                    break
                if i == 3:
                    temp = int(word) << 2
                elif i == 4:
                    red.append(temp + int(word))
                elif i == 5:
                    temp = int(word) << 2
                elif i == 6:
                    green.append(temp + int(word))
                elif i == 7:
                    temp = int(word) << 2
                elif i == 8:
                    blue.append(temp + int(word))
                elif i == 9:
                    temp = int(word) << 2
                elif i == 10:
                    intensity.append(temp + int(word))  # Concat
                        the 10 bit value of the light intensity
                       to one int
                i += 1
        if whichLine == False:
            for word in line.split():
                if i == 2 and int(word) == 8:
                    whichLine = True
                    break
                if i == 3:
                    uvLight.append(int(word))
                elif i == 4:
                    IRLight.append(int(word))
                i += 1
    f.close()

# uvGraf makes a skewed normal distribution to
```

```
# optimize against.
#
# Argument: None
# Return: Function for uv
def uvGraf():
    global uv
    uvPeak = 355
    alpha = 4
    width = 5e3

    #Make a normal distribution that skews with variable alpha
    pdf = np.exp((-(x - uvPeak) ** 2) / width)
    pdfAlpha = np.exp(-((alpha * (x - uvPeak)) ** 2) / width)

    yCDF = np.zeros(len(pdfAlpha))
    yCDF[0] = pdfAlpha[0]

    for i in range(1, len(pdf)):
        yCDF[i] = yCDF[i - 1] + pdfAlpha[i]

    yCDF /= yCDF[-1]
    yCDF = 1 - yCDF
    uv = 2 * pdf * yCDF
    uv /= np.max(uv)
    return uv

# redGreenBlue makes normal distributions for
# the red, green and blue plots for the optimizations.
#
# Argument: None
# Return: None.
def redGreenBlue():
    redLambda = 620
    blueLambda = 460
    greenLambda = 540
    width=10e3
    alpha = 4
    global yr
    global yg
    global yb
    global skewRed

    yr = np.exp((-(x - redLambda) ** 2) / width * 2)
    pdfRed = np.exp(-((alpha * (x - redLambda)) ** 2) / width *
        2)
    redCDF = np.zeros(len(pdfRed))
    redCDF[0] = pdfRed[0]

    for i in range(1, len(yr)):
```

```
            redCDF[i] = redCDF[i - 1] + pdfRed[i]

        redCDF /= redCDF[-1]
        redCDF = 0.001 + redCDF

        skewRed = 2 * yr * redCDF
        skewRed /= np.max(skewRed)

        yb = np.exp((-(x - blueLambda) ** 2) / (width*0.6))

        yg = np.exp((-(x - greenLambda) ** 2) / (width*0.18))

# intensityGraf is plotting a skewed normalized distribution
# for the optimization.
#
# Argument: None
# Return: Function for intensity
def intensityGraf():
    global skewPdf
    intensityPeak = 940
    alpha = 4
    width = 10e4

    yi = np.exp((-(x - intensityPeak) ** 2) / width)
    pdfAlpha = np.exp(-((alpha * (x - intensityPeak)) ** 2) /
        width)
    yCDF = np.zeros(len(pdfAlpha))
    yCDF[0] = pdfAlpha[0]

    for i in range(1, len(yi)):
        yCDF[i] = yCDF[i - 1] + pdfAlpha[i]

    yCDF /= yCDF[-1]
    yCDF = 1 - yCDF

    skewPdf = 2 * yi * yCDF
    skewPdf /= np.max(skewPdf)
    return skewPdf

# datasheetfunctions makes the red, green, blue, intensity
# and uv curves by calling their definitions and plotting
# them.
#
# Argument: None
# Return: None
def datasheetfunctions():
    global skewRed
    global yg
    global yb
```

```
    uv=uvGraf()
    redGreenBlue()
    skewPdf=intensityGraf()

    # Normalize the curves and multiplication between them and
        their weight.
    # Weights: [0.68814022  0.40765524  0.83726203  1.55842677
        1.50851575]
    skewRed /= np.sum(skewRed)
    skewRed *= 0.68814022
    yg /= np.sum(yg)
    yg *= 0.83726203
    yb /= np.sum(yb)
    yb *= 0.40765524
    uv /= np.sum(uv)
    uv *= 1.50851575
    skewPdf /= np.sum(skewPdf)
    skewPdf *= 1.55842677

    # Plots the curves that the program is optimizing
    # the sensor values against.
    plt.plot(x, skewRed, color="red")
    plt.plot(x, yg, color="green")
    plt.plot(x, yb, color="blue")
    plt.plot(x, uv, color="black")
    plt.plot(x, skewPdf, color="magenta")
    plt.grid()
    plt.show()


# spectrumToOutput takes a vector and sums
# the vectors element with the normal distrubutions
# for the sensors.
#
# Argument: vector named spectrum
# Return: None
def spectrumToOutput(spectrum):
    global spred
    spred = np.sum(spectrum * skewRed)
    global spblue
    spblue = np.sum(spectrum * yb)
    global spgreen
    spgreen = np.sum(spectrum * yg)
    global spint
    spint = np.sum(spectrum * skewPdf)
    global spuv
    spuv = np.sum(spectrum * uv)
```

```
# plot_curves() plots the sensor values
def plot_curves():

    light=plt.plot(red[0:],'r')
    light=plt.plot(green[0:],'g')
    light=plt.plot(blue[0:],'b')
    light=plt.plot(intensity[0:],'m')
    light=plt.plot(uvLight[0:],'k')
    light=plt.plot(IRLight[0:],'c')
    plt.show()


def main():

    # reads file with sensor values
    # name is for the plot later on
    readFile("1048soligdag8maj.txt")
    name="Morning sunlight"

    # makes normal distributions
    datasheetfunctions()

    # Initiates the spectrum vector
    spectrum=[]
    for _ in range(0,num_lambda):
        spectrum.append(np.random.randint(1))

    # Initiates a large value to the old deviation to guarantee
        first trial is accepted
    oldDeviation = (num_sensor*maxValueSensor)**2
    converge = False
    c=0
    epsilon = []
    exp_parameter=0.05

    # calculates the average value of the sensor values
    sensRed = np.sum(red) / len(red)
    sensGreen = np.sum(green) / len(green)
    sensBlue = np.sum(blue) / len(blue)
    sensInt = np.sum(intensity) / len(intensity)
    sensUv = np.sum(uvLight) / len(uvLight)

    while not converge:
        # mean and standard deviation for the normal
            distribution
        mu, sigma = 0,15
        # Random value from the normal distribution
        deviationStep = np.random.normal(mu, sigma)
```

```
        llambda = np.random.randint(num_lambda)
        randExp = np.random.exponential(exp_parameter)
        spectrum[llambda] += deviationStep

        spectrumToOutput(spectrum)

        outputSum = spred + spblue + spgreen + spuv + spint
        sensorSum = (np.sum(red) + np.sum(blue) + np.sum(green)
            + np.sum(intensity) + np.sum(uvLight))/len(red)

        scaleValue = outputSum / sensorSum

        outRed = spred * scaleValue
        outBlue = spblue * scaleValue
        outGreen = spgreen * scaleValue
        outInt = spint * scaleValue
        outUv = spuv * scaleValue

        deviation = ((outRed-sensRed)**2 + (outBlue-sensBlue)**2
             + (outGreen-sensGreen)**2 + (outInt-sensInt)**2 + (
            outUv-sensUv)**2)/(sensRed**2+sensBlue**2+sensGreen
            **2+sensInt**2+sensUv**2)

        if spectrum[llambda] < 1:
             spectrum[llambda] -= deviationStep

        elif deviation<oldDeviation:
             oldDeviation = deviation

        elif deviation>oldDeviation:
             if randExp>deviation:
                  oldDeviation = deviation
             else:
                  spectrum[llambda] -= deviationStep

        epsilon.append(deviation)

        c+=1
        if c>35000:
             converge=True

# Plots the descending deviation to make sure
# it converge to zero.
plt.plot(epsilon)
plt.grid()
plt.show()

# Visible light colormap
num_colors_visible_light=300
```

```
cmapVisible = mpl.cm.jet
zVisible = np.linspace(0, num_colors_visible_light,
    num_lambda * num_colors_visible_light / (x.max() - x.min
    ()))
normalizeVi = mpl.colors.Normalize(vmin=zVisible.min(), vmax
    =zVisible.max())

# Ultraviolett light colormap
num_colors_uv_light = 1800
cmapUV = mpl.cm.gnuplot
zUV = np.linspace(0, num_colors_uv_light, num_lambda *
    num_colors_uv_light / (x.max() - x.min()))
normalizeUv = mpl.colors.Normalize(vmin=zUV.min(), vmax=zUV.
    max())

# Infrared light colormap
num_colors_IR_light = 1400
cmapIR = mpl.cm.hot
zIR = np.linspace(0, num_colors_IR_light, num_lambda *
    num_colors_IR_light / (x.max() - x.min()))
normalizeIr = mpl.colors.Normalize(vmin=zIR.min(), vmax=zIR.
    max())

# Plots the outline of the spectrum.
plt.plot(x, spectrum, color="black")

cVI=0
cUV=0
cIR=0

# Plots the different colormaps under the spectrum curve and
    x-axis.
for i in range(num_lambda - 1):
    if x[i]<400:
        plt.fill_between([x[i], x[i + 1]], [spectrum[i],
            spectrum[i + 1]], color=cmapUV(normalizeUv(zUV[
            cUV])))
        cUV+=1
    if x[i]<701 and x[i]>399:
        plt.fill_between([x[i], x[i + 1]], [spectrum[i],
            spectrum[i + 1]], color=cmapVisible(normalizeVi(
            zVisible[cVI])))
        cVI+=1
    if x[i]>700:
        plt.fill_between([x[i], x[i + 1]], [spectrum[i],
            spectrum[i + 1]], color=cmapIR(normalizeIr(zIR[
            cIR])))
        cIR+=1
```

```
    plt.axis([200, 1050, 0, max(spectrum)+max(spectrum)*0.1])
    plt.title(name)
    plt.axvline(400,color='black')
    plt.axvline(700,color='black')
    plt.annotate('Visible light', xy=(470, (max(spectrum)+max(
        spectrum)*0.05)))
    plt.xlabel('Wavelength [nm]')
    plt.ylabel('Intensity')
    plt.show()

main()
```

metropolismontecarlo.py

Bilaga D

Artificiella neurala nätverket i Tensorflow

```
import numpy as np
import tensorflow as tf
from sklearn.utils import shuffle


# readFile takes the string-argument ant opens the file,
# reads column by column, line by line and puts the
# values in right list (array).
#
# Input argument: string, name of textfile.
# Return: arrays red, blue, green, intensity and UVLight
def readFile(filename):
    red = []
    green = []
    blue = []
    intensity = []
    uvLight = []
    IRLight = []
    f = open(filename, "r")  # Opens the datafile
    scaleSensValues=8
    firstLine = f.readline()  # First line in the textfile is
       rubbish. TOSS IT AWAAAY
    whichLine = False
    for line in f:  # Loops through the textfile line by line
        i = 0
        whichLine = not whichLine
        if whichLine == True:
            for word in line.split():  # Splits the line word by
                word and puts the values in the right array
                if i == 2 and int(word) == 3:
                    whichLine = False
```

```
                    i = 0
                    break
                if i == 3:
                    temp = int(word) << 2
                elif i == 4:
                    red.append((temp + int(word))*
                        scaleSensValues)
                elif i == 5:
                    temp = int(word) << 2
                elif i == 6:
                    green.append((temp + int(word))*
                        scaleSensValues)
                elif i == 7:
                    temp = int(word) << 2
                elif i == 8:
                    blue.append((temp + int(word))*
                        scaleSensValues)
                elif i == 9:
                    temp = int(word) << 2
                elif i == 10:
                    intensity.append(temp + int(word))  # Concat
                        the 10 bit value of the light intensity
                        to one int
                i += 1
        if whichLine == False:
            for word in line.split():
                if i == 2 and int(word) == 8:
                    whichLine = True
                    break
                if i == 3:
                    uvLight.append(int(word))
                elif i == 4:
                    IRLight.append(int(word))
                i += 1
    f.close()
    return red,green,blue,intensity,uvLight

# normedValues normalize an array to 0-1
#
# Argument: Array that shall be normalized
# Return: Normalized array
def normedValues(RGBarray):
    array=[]
    ymax=np.amax(RGBarray)

    for rgb in RGBarray:
        array.append(np.float32(rgb)/ymax)
    return array
```

```
# data_handeling makes small trainingsets from files.
#
# Argument: Filename.txt as string and
# the classification as an array
# Return: array label and lexicon.
def data_handeling(filename,classification):
    red,green,blue,intensity,uv=readFile(filename)
    red=normedValues(red)
    green=normedValues(green)
    blue=normedValues(blue)
    intensity=normedValues(intensity)
    label=[]
    lexicon=[]
    for i in range(len(red)):
        label.append(classification)
        lexicon.append([red[i],green[i],blue[i],intensity[i]])
    return label, lexicon

# Training_set makes big trainingset for the network
# from the small ones made by daata_handeling.
#
# Argument: None
# Return: Two arrays, one with the labels and
# one with input data.
def training_set():
    # [sunlight, kall led, varm led]
    label1,lexicon1=data_handeling("1019soligdag8maj.txt",[1,0])
    label2, lexicon2=data_handeling("1035soligdag8maj.txt"
        ,[1,0])
    label3, lexicon3=data_handeling("1048soligdag8maj.txt"
        ,[1,0])
    label4, lexicon4=data_handeling("1115soligdag8maj.txt"
        ,[1,0])
    label5, lexicon5=data_handeling("1145till12soligdag8maj.txt"
        ,[1,0])
    label6, lexicon6 = data_handeling("1529
       maj19soligdag0procentmoln.txt", [0,1])
    label7, lexicon7 = data_handeling("1549
       maj19soligdag0procentmoln.txt", [0,1])
    label8, lexicon8 = data_handeling("1559
       maj19soligdag0procentmoln.txt", [0,1])
    label9, lexicon9 = data_handeling("1614
       maj19soligdag0procentmoln.txt", [0,1])
    label10, lexicon10 = data_handeling("1622
       maj19soligdag0procentmoln.txt", [0,1])
    label11, lexicon11 = data_handeling("1633
       maj19soligdag0procentmoln.txt", [0,1])
    label12, lexicon12 = data_handeling("1644
       maj19soligdag0procentmoln.txt", [0,1])
```

```
    labelfirst=np.vstack((label1,label2))
    labelsecond=np.vstack((labelfirst,label3))
    labelthird=np.vstack((labelsecond,label4))
    labelforth=np.vstack((labelthird,label5))
    labelfive = np.vstack((labelforth, label6))
    labelsix = np.vstack((labelfive, label7))
    labelseven = np.vstack((labelsix, label8))
    labeleight = np.vstack((labelseven, label9))
    labelnine = np.vstack((labeleight, label10))
    labelten = np.vstack((labelnine, label11))
    labeleleven = np.vstack((labelten, label12))

    lexiconfirst = np.vstack((lexicon1, lexicon2))
    lexiconsecond = np.vstack((lexiconfirst, lexicon3))
    lexiconthird = np.vstack((lexiconsecond, lexicon4))
    lexiconforth = np.vstack((lexiconthird, lexicon5))
    lexiconfive = np.vstack((lexiconforth, lexicon6))
    lexiconsix = np.vstack((lexiconfive, lexicon7))
    lexiconseven = np.vstack((lexiconsix, lexicon8))
    lexiconeight = np.vstack((lexiconseven, lexicon9))
    lexiconnine = np.vstack((lexiconeight, lexicon10))
    lexiconten = np.vstack((lexiconnine, lexicon11))
    lexiconeleven = np.vstack((lexiconten, lexicon12))

    x, y = shuffle(labeleleven, lexiconeleven, random_state=0)
    return x,y

# How many input data there is to the network
# In this case it is 5, red, green, blue, uv and
# intensity which correspond to the value 4.
inputSize=4

# Number of nodes that the hidden layers have
nodesL1=200
nodesL2=200
nodesL3=200

# Number of outputs
classes=2

training_y, training_x = training_set()
numSets = len(training_x)

# Makes placeholders,
# not yet initialize variables
x=tf.placeholder("float32",[None,inputSize])
y=tf.placeholder("float32")
```

```
# neural_network_model makes a model for the neural
# network. Its a frame of the hidden layera and their
# activation functions.
#
# Argument: array of input data
# Return: array of output data
def neural_network_model(data):

    # Makes three hidden layer for the network with random
        weights and biases
    hiddenL1= {"weights":tf.Variable(tf.random_normal([inputSize
        ,nodesL1])),
               "biases":tf.Variable(tf.random_normal([nodesL1]))
                   }
    hiddenL2 = {"weights": tf.Variable(tf.random_normal([nodesL1
        ,nodesL2])),
                "biases": tf.Variable(tf.random_normal([nodesL2
                    ]))}
    hiddenL3 = {"weights": tf.Variable(tf.random_normal([nodesL2
        , nodesL3])),
                "biases": tf.Variable(tf.random_normal([nodesL3
                    ]))}
    outputLayer = {"weights": tf.Variable(tf.random_normal([
        nodesL3, classes])),
                "biases": tf.Variable(tf.random_normal([classes
                    ]))}

    # Matrix multiplication between the input data and the
        weights.
    # Adds the biases and runs the activation function relu (
        Rectified Linear Function)
    l1=tf.add(tf.matmul(data,hiddenL1["weights"]),hiddenL1["
        biases"])
    l1=tf.nn.relu(l1)

    l2=tf.add(tf.matmul(l1,hiddenL2["weights"]),hiddenL2["biases
        "])
    l2=tf.nn.relu(l2)

    l3=tf.add(tf.matmul(l2,hiddenL3["weights"]),hiddenL3["biases
        "])
    l3=tf.nn.relu(l3)

    output=tf.matmul(l3,outputLayer["weights"])+outputLayer["
        biases"]

    return output
```

```
def train_neural_network(inputData):
    # Initiate the model
    prediction = neural_network_model(inputData)
    # Initiate the cost function that compute the mean of
        elements across dimensions of a tensor,
    # in this case; softmax cross entropy between logits and
        labels
    cost=tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits
        (logits=prediction, labels=y))

    # algorithm for first - order gradient - based optimization
    learningRate=0.0001
    optimizer = tf.train.AdamOptimizer(learningRate).minimize(
        cost)
    # algorithm for first - order gradient - based optimization

    HowManyCycles = 15

    # Starts a session
    with tf.Session() as sess:
        # Initialize all the variables and the saver.
        # They only exist in the session enviroment.
        sess.run(tf.global_variables_initializer())
        saver = tf.train.Saver(tf.global_variables())

        for cycle in range(HowManyCycles):
            cycle_loss=0
            # batchsize is for how many trainingsets the
            # network is going to train in one cycle
            batchSize=100
            i=0
            # Trains the network in cycles and in batches.
            # This is to reduce calculations on the same
            # trainingsets.
            while i<batchSize:
                x_ = training_x
                y_ = training_y
                _, c =sess.run([optimizer,cost],feed_dict={x:x_,
                    y:y_})
                cycle_loss+=c
                i+=1
            batchSize+=batchSize
            # Print number of cycle and the cycle_loss every
                loop.
            # The loss is for checking if the network is going
                to be overdetermined.
            print("Cycle ",cycle, " completed out of ",
                HowManyCycles, "Loss: ", cycle_loss)
```

```
        # Calculates the accuracy, printing that and saving the
            model.
        correct=tf.equal(tf.argmax(prediction,1), tf.argmax(y,1)
            )
        accuracy=tf.reduce_mean(tf.cast(correct,"float"))
        print("Accuracy: ", accuracy.eval({x:training_x,y:
            training_y}))
        saved_path=saver.save(sess, "C:\games\solfmem.ckpt")
        print("Saved at path: %s", saved_path)


# gogogo takes input data and runs the restored network.
# After the calculations it prints what the estimation is.
#
# Argument: Array input data
# Return: None
def gogogo(input_data):
    prediction=neural_network_model(x)
    with tf.Session() as sess:
    # Restore variables from disk.
        saver = tf.train.Saver()
        saver.restore(sess, "C:\games\99accur3output.ckpt")
        print("Model restored.")

    # Do some work with the mode
        result=sess.run(tf.argmax(prediction.eval(feed_dict={x:
            input_data}),1))
        #print(prediction.eval(feed_dict={x:input_data}))
        #print(result)
        if result==0:
            print("Sunlight")
        elif result==1:
            print("Cold light")
        elif result==2:
            print("Warm light")
        else:
            print("Something is wrooong")

# testing takes one set of data from a file.
#This is for testing tha already calculated network.
#
# Argument: filename as a string, i as an integer
# Return: input data
def testing(filename,i):
    red,green,blue,intensity,uv=readFile(filename)
    red=normedValues(red)
    green=normedValues(green)
    blue=normedValues(blue)
    intensity=normedValues(intensity)
```

```
    lexicon=[]
    lexicon.append([red[i],green[i],blue[i],intensity[i]])
    return lexicon

train_neural_network(x)


#test_data=testing("taklampaarbetsrumlada.txt",100)
#gogogo(test_data)
```

tensors.py