CHALMERS

GÖTEBORGS UNIVERSITET

Optimering över finita diskreta mått

och dess tillämpning på bayesianska neurala nätverk

Optimization of Finite Discrete Measures

Kandidatarbete inom civilingenjörsutbildningen vid Chalmers Tekniska Högskola

Filip Berglund
Karl Egeland
Samuel Martinsson
Hampus Olsson
Joel Wall
Johan Ödesjö

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2023

Optimering över finita diskreta mått

och dess tillämpning på bayesianska neurala nätverk

Kandidatarbete i matematik inom civilingenjörsprogrammet Maskinteknik vid Chalmers
Hampus Olsson Karl Egeland

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers
Filip Berglund Samuel Martinsson
Joel Wall Johan Ödesjö

Handledare: Sergei Zuyev

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2023

Förord

Detta är ett kandidatarbete vid institutionen för Matematiska Vetenskaper vid Chalmers Tekniska Högskola under vårterminen 2023. Vi vill inledningsvis tacka handledare Sergei Zuyev för kontinuerligt stöd och vägledning under arbetet. Vi vill även tacka Hans Malmström för hans respons på ett tidigt utkast av rapporten och Marco Schirone för hans hjälp med källor och källhänvisning.

Arbetet har skrivits av sex författare och vad som skrivits av vem hittas i tabell (1). Det finns utöver detta en loggbok där det varje vecka antecknats vad alla medlemmar gjort under veckan och hur mycket tid de lagt. Denna loggbok är uppdelad i två delar, en del med hur mycket tid medlemmarna lagt med en kort förklaring av vad de gjort samt ett längre dokument där det står mer noggrant vad medlemmarna gjort. Samtliga författare är överens om att alla bidragit till projektet och att alla har varit delaktiga i utvecklandet av Python-koden. Vilka delar som varje författare kan tillskrivas framgår från tidigare nämnda loggbok.

Under arbetets gång har dessutom versionshanteringsystemet Git använts och alla implementationer i Python finns att tillgå i ett delat GitHub-projekt. Detta kan hämtas från https://github.com/Miralleyan/Kandidatarbete.

Tabell 1: Bidragsrapport - författare för avsnitt

Del i rapport	Författare	Kommentar
Populärvetenskaplig presentation	Karl, Samuel	Huvudförfattare: Karl
Abstract	Johan	
Sammandrag	Hampus	
Inledning	Karl, Samuel	
Syfte	Johan	
Avgränsningar	Karl	
Bakgrund		
Bayesiansk statistik	Samuel, Johan	
Neurala nätverk	Samuel, Johan, Joel	
Bayesianska neurala nätverk	Samuel	
Olika former av regression	Samuel	
PyTorch	Filip, Joel	
Automatisk differentiering	Johan, Samuel	
Mått	Joel, Filip	
Differentiering av mått	Filip	
Kärndensitetsuppskattning	Johan	
Metod	Johan	Texten innan sektion 3.1 börjar
Parametriska modeller	Hampus, Filip, Joel	3.1.2: Hampus
Icke-parametriska modeller	Joel, Filip, Karl, Johan	3.2.1: Filip, 3.2.3: Karl och Johan
Målfunktioner	Hampus, Joel, Samuel	3.3.1-3.3.2: Samuel, 3.3.3: Joel, 3.3.4: Hampus
Resultat		
Jämförelse mellan parametriska och icke-parametriska metoder	Samuel, Johan	Huvudförfattare: Samuel
Resultat från tester	Hampus	
Diskussion		
Jämförelse	Samuel	
Val för KDE	Johan	
Tidskomplexitet för större modeller med OFDM	Joel	
Vidare forskning	Samuel	
Samhälleliga och etiska aspekter	Karl	
Slutsats	Joel	
Appendix		
Resultattabeller	Johan, Samuel	
Degenererad summa av väntevärden	Filip	
Linjärkombination av stokastiska variabler	Filip	
Yttre straffunktion	Hampus	
Lagrangemultiplikator-metoden	Samuel	
Icke-parametriska diskreta sannolikhetsmått med softmax	Joel, Johan	
Poängfunktioner och poängregler	Filip	

Populärvetenskaplig presentation

Under de senaste åren har det blivit allt mer vanligt att använda sig av applikationer som bygger på artificiell intelligens. För att skapa artificiell intelligens används olika former av maskininlärning. Maskininlärning kan beskrivas som metoder för att få en dator att lära sig ett antal regler från en mängd data, utan explicit instruktion från användaren. Med hjälp av dessa regler ska exempelvis en uppgift kunna lösas eller förutsägelser ges utifrån ny indata.

Ett område där maskininlärningsprocesser har implementerats och som växt mycket de senaste åren är i rekrytering till företag vilket är en etisk aspekt som tas upp i texten. Här ges förutsägelser på hur väl en viss individ hade presterat i rollen den söker, utifrån data som beskriver hur väl individer med liknande egenskaper har presterat i en liknande roll tidigare. Det blir i detta fall tydligt hur verktyg som bygger på maskininlärning påverkar människor och vilka samhälleliga konsekvenser de kan ge. Det är i dessa tillämpningar, som kan påverka människor och samhället i stort, viktigt att verktygen kan ge goda förutsägelser för ny indata.

I maskininlärningsprocesserna användes ofta så kallade artificiella neurala nätverk. Traditionella artificiella neurala nätverk består av noder eller neuroner som sammankopplas med hjälp av länkar mellan de olika noderna. Detta är inspirerat av hur den mänskliga hjärnan fungerar. Noderna struktureras i ett antal lager där antalet noder i varje lager kan variera. Antalet lager kan också variera men i det enklaste fallet finns ett lager för indata, ett lager för utdata samt ett mellanliggande lager som kopplar indata till utdata. Varje länk går då från varje nod i ett lager till varje nod i ett närliggande lager. Länkarna har också en associerad vikt vilken beskriver hur stark en viss sammankoppling är mellan två noder. Dessa vikter kan sedan modifieras och det är just det som händer när det neurala nätverket lär sig regler för hur en viss indata bör ge en viss utdata.

Ett problem som vanligen uppstår när det neurala nätverket lär sig regler utifrån träningsdata är överanpassade nätverk. Överanpassning (eng. overfitting) innebär att det neurala nätverket är allt för väl anpassat på den data som datorn lärt sig regler utifrån. När det neurala nätverket sedan ska ge förutsägelser på ny indata blir förutsägelserna bristfälliga. Detta kan antingen bero på att den data som nätverket tränat på ej är representativ för alla möjliga värden på indata eller att mängden träningsdata helt enkelt är för liten. Det är i denna typ av fall som så kallade bayesianska neurala nätverk kan vara användbara.

Medan traditionella neurala nätverk har länkar bestående av numeriska vikter har istället bayesianska neurala nätverk länkar där ett tal slumpas varje gång enligt en given regel. Varje länk är alltså inte ett enstaka värde utan istället en slumpvariabel. Modeller som bygger på bayesianska neurala nätverk kallas stokastiska i motsats till de traditionella, deterministiska modellerna. Eftersom varje länk i nätverket har en sannolikhetsfördelning finns det en inbyggd variation i nätverket. Detta kan göra att nätverket inte blir överanpassat till träningsdata och genom att studera länkarnas sannolikhetsfördelningar ges en bild av hur säkert nätverket är, samt vilka andra möjliga värden modellens resultat kan anta.

I detta projekt utforskas bayesianska neurala nätverk, främst genom att undersöka metoder för att hitta den lämpligaste sannolikhetsfördelningen för en viss datamängd. Metoder som syftar till att hitta de lämpligaste lösningarna till ett specifikt problem kallas optimeringsalgoritmer. Vi studerar optimeringsalgoritmer ämnade att hitta en sannolikhetsfördelning som kan ha gett upphov till observerad data. Specifikt testas en metod som tidigare i stor utsträckning inte använts för just neurala nätverk. Metoden testas på olika typer av data och jämförs sedan med andra metoder med liknande syfte.

Abstract

Artificial intelligence and neural networks is currently a highly debated topic. In this paper, we will investigate some methods for optimization on measures which could be used to implement so called Bayesian neural networks. These can potentially be used to avoid overfitting and to model the uncertainty in the output of neural networks.

Using Python and PyTorch we have implemented three models designed for Bayesian regression: Steepest descent algorithm for fixed total mass problem, polynomial regression and linear regression. These are three distinct ways to model data, and we will investigate how effective and precise they are given the same data.

Linear combination of stochastic variables describes a measure as a sum of stochastic variables multiplied with an arbitrary function. By assuming gaussian distributions for the stochastic variables some assumptions can be made, which make calculations simpler. During testing, this model showed high precision but was the slowest to converge to an answer.

The polynomial neural network is similar to the linear combination in that the functions are terms in a polynomial. However, the method to solve it is different as it uses a neural network and never calculates the parameters of the stochastic variables explicitly. It returns functions that for input x returns the mean values or standard deviation in that x. The precision for this method is somewhat poorer, but it converges faster.

Steepest descent algorithm for fixed total mass problem has never been used together with neural networks and automatic differentiation. For this method, we make no assumptions on the distribution of the measure and instead represent it as a finite amount of discrete atoms. The mass of the atoms are adjusted to minimize a loss function, but the total mass is preserved. With this function, any function can be described and stochastic variables can be either dependent or independent. This method appears to need more data to train on for the results to be within acceptable precision, but is the fastest to converge.

Different loss functions are tried, however, we conclude that the logarithmic loss function is the most accurate for us. This loss function is used in all tests. For future papers, it could be valuable to investigate the method's ability as a part of larger networks, as this paper only cover simpler problems with few parameters. We conclude that further reaserch is needed before the steepest decent algorithm for fixed total mass is practical for use in large stochastic models like Bayesian neural networks.

Sammandrag

Artificiell intelligens och neurala nätverk är just nu ett mycket omdebatterat ämne. I denna rapport undersöks metoder för optimering av mått, vilka kan användas i implementationen av bayesianska neurala nätverk. Dessa nätverk kan potentiellt förhindra överanpassning till träningsdata, och möjliggör kvantifiering av osäkerheten för utdata.

Med hjälp av Python och PyTorch har tre modeller designade för bayesiansk regression implementerats: optimering över diskreta mått, polynomisk modell och linjärkombination av normalfördelade stokastiska variabler. Dessa har distinkta tillvägagångsätt att modellera data. I synnerhet undersöks hur effektivt och exakt de kan anpassas till given data.

Linjärkombination av stokastiska variabler beskriver ett mått som en summa av stokastiska variabler multiplicerat med en godtycklig funktion. Genom att anta att stokastiska variablerna är normalfördelade kan stora förenklingar göras vilket gör algoritmen relativt okomplicerad. Vid testning påvisar denna metod god förmåga att modellera data med hög precision, men konvergerar långsammast.

Det polynomiska neurala nätverket efterliknar linjärkombinationen vid fallet att funktionerna är termer i ett polynom. Lösningsgången är dock annorlunda då ett neuralt nätverk används och att stokastiska variablernas parametrar inte beräknas explicit. Istället matar nätverket ut en sammansatt funktion för medelvärde respektive standardavvikelse. Denna metod har något sämre precision men snabbare konvergenstid.

Optimeringsalgoritmen över finita diskreta mått är tidigare outforskad i samband med neurala nätverk och automatisk differentiering. Här görs inga antagelser om måttets form utan det representeras helt av ett finit antal diskreta atomer. Atomernas massa justeras för att minimera en målfunktion men måttet bibehåller dess totala massa. Med denna metod kan valfri funktion beskrivas och stokastiska variablerna kan antas vara beroende eller oberoende. Testning tyder på att denna metod kräver fler datapunkter att träna på för att erhålla en acceptabel precision men är den snabbaste att konvergera av de tre.

Olika målfunktioner undersöks, men det fastställs att log-likelihoodfunktionen är den bästa för stokastiska variabler. Denna används sedan för alla tester. Det kvarstår att undersöka metodernas effektivitet i större nätverk då denna rapport endast behandlar enklare problem med ett få antal parametrar. Slutsatsen är att vidare forskning för optimeringsalgoritmen över finita diskreta mått krävs innan metoden är tillämpbar i stora bayesianska neurala nätverk.

Innehållsförteckning

1 Inledning

Neurala nätverk har visat sig användbara inom maskininlärning, specifikt då de applicerats inom bild- och språkanalys. Ett stort problem med neurala nätverk är det fenomen som kallas överanpassning. Överanpassning innebär att en modell, i detta fall ett neuralt nätverk, anpassar sig för väl till träningsdata och därför inte kan beskriva ny data som den inte tidigare observerat. Det är intressant att försöka studera hur detta problem kan undvikas.

Bayesianska neurala nätverk (BNN) är en typ av neurala nätverk som utnyttjar stokastiska parametrar. De kvantifierar osäkerheten i en modells förutsägelser, vilket kan göra dem robustare vid tillämpning. Då parametrarna är stokastiska variabler kan även problemet med överanpassning undvikas, då modellen alltid kommer variera kring medelvärden.

Ett problem med BNN är att de är mer beräkningstunga än traditionella neurala nätverk (TNN). Detta beror på att parametrarna i ett BNN är sannolikhetsmått och träningen av dessa kräver mer beräkningskraft då varje sannolikhetsmått innehåller mer information än parametrar i ett TNN. Därav kan de bli opraktiska eftersom det tar lång tid att träna och utvärdera på en större datamängd. Det är därför av intresse att studera algoritmer som effektiviserar optimeringen över sannolikhetsmått.

En algoritm för optimering över mått som visat sig effektiv är *Steepest descent algorithm for fixed total mass* [21], vilken i denna rapport benämns ”optimeringsalgoritm över finita diskreta mått” (OFDM). Algoritmen påstås i rapporten vara en sann ekvivalent till ”sjunkgradient” (eng. gradient descent), som är en etablerad algoritm inom matematisk optimering [3]. Detta motiverar att den teoretiskt sett bör prestera väl vid optimering av funktioner över sannolikhetsmått.

Denna algoritm har studerats i syfte att implementeras som en del av ett BNN tidigare [12] och då påstods att algoritmen ej var praktisk för denna tillämpning, då den var svår att skala för högre dimensioner. Författarna beskriver hur det inte är uppenbart hur metoden ska kunna integreras med moderna tekniker för neurala nätverk. OFDM utvärderas dock ej explicit i rapporten. Vi anser att det är intressant att utvärdera metoden vidare i syfte att implementeras i neurala nätverk. Vi ämnar att jämföra OFDM med andra algoritmer och visa att algoritmen kan vara effektivare än dessa alternativ.

Den tidigare implementationen av algoritmen skedde i programmeringsspråket R. En implementering med Python-biblioteket PyTorch är ett framsteg då vi med detta kan beräkna gradienten för funktioner automatiskt. En sådan implementation skulle därför kunna vara beräkningsmässigt snabbare än tidigare och därför kunna appliceras på större nätverk.

1.1 Syfte

Syftet med arbetet är att studera olika typer av optimeringsalgoritmer för BNN. I synnerhet ska en algoritm [21] som tidigare inte utnyttjat automatisk differentiering implementeras med hjälp av Python-biblioteket PyTorch. Denna metod ska sedan jämföras med andra metoder, specifikt ”linjärkombination av stokastiska variabler” samt ”polynomisk modell med normalfördelade koefficienter”.

1.2 Avgränsningar

Projektet kommer enbart att implementeras i programmeringsspråket Python där det redan existerande biblioteket PyTorch utnyttjas. Detta för att upprätthålla en tillräcklig hög abstraktionsnivå med användning av redan implementerade funktioner för maskininlärning. En specifik funktion som utnyttjas är att gradienten automatiskt beräknas för tensorer i PyTorch.

Det finns många möjliga vägar för projektet, men på grund av begränsad tid har fokuserar rapporten på en jämförelse. På grund av en given tidsram undersöks och behandlas endast de algoritmer som anses mest relevanta. Någon omfattande jämförelse med traditionella neurala nätverk kommer heller inte genomföras.

2 Bakgrund

I detta avsnitt etableras nödvändig teori bakom bland annat bayesiansk statistik, neurala nätverk samt mått.

2.1 Bayesiansk statistik

Bayesiansk statistik grundar sig i Bayes lag, vilken kan definieras enligt följande:

Låt $A_1, ..., A_n$ vara n disjunkta händelser, där $\cup_{i=1}^{n} A_i = \Omega$ utgör rummet av alla möjliga händelser. Bayes lag säger då att sannolikheten, P, för händelse A_i givet en händelse B ges av

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{P(B)}. \tag{1}$$

Bayes lag används i en metod kallad bayesiansk inferens, vilket är en typ av statistisk inferens, som används för att bestämma egenskaper hos en okänd sannolikhetsfördelning.

Bayesiansk inferens utgår från att vi har en hypotes för fördelningen av en parameter. Denna hypotes kallas en a priori-fördelning och vi söker den riktiga fördelningen för parametern. För att beräkna den fördelningen behöver vi uppdatera vår a priori-fördelning med hjälp av data som vi observerar. Vi ställer upp en statistisk modell för hur sannolik vår observerade data är givet ett värde på parametern. Denna modell kallas för en likelihoodfunktion. En a posteriori-fördelning, d.v.s. vår nya hypotes om hur fördelning av parametern ser ut givet observerad data, kan då beräknas via

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}, \tag{2}$$

där H är hypotesen för parametern och E är data som observerats. $P(E|H)$ anger likelihoodfunktionen, $P(H)$ är a priori-fördelningen och $P(E)$ anger den så kallade marginella fördelningen, vilket är fördelningen för vår data oberoende av hypotesen. Denna metod kan användas för att uppdatera en initial hypotes gällande en parameters fördelning efter att data har observerats, vilket är användbart i situationer där det är svårt eller till och med omöjligt att bestämma en fördelning på förhand.

2.2 Neurala nätverk

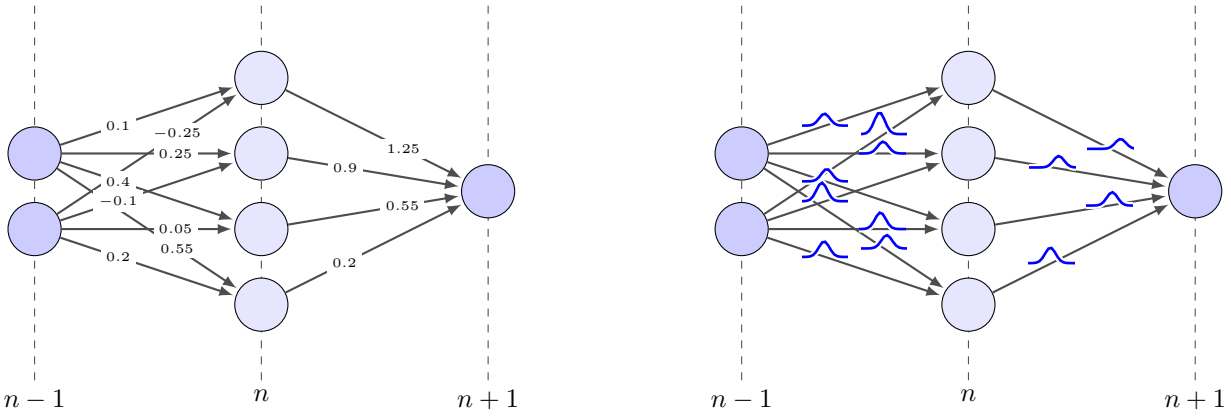
Neurala nätverk (NN) är strukturer som används inom maskininlärning för att skapa modeller som exempelvis klassificerar data eller implementerar regression till olika funktioner. Som namnet antyder imiterar dessa nätverk biologiska neurala nätverk, som exempelvis en mänsklig hjärna, och är på så sätt uppbyggda av noder och kopplingar mellan dessa noder. Dessa noder, eller neuroner, betraktas som funktioner som tar in en uppsättning variabler som indata och returnerar därefter ett värde.

Dessa noder delas in i lager vilka innehåller de noder som tar utdatan från föregående lager och vars utdata skickas till nästa lager av noder. Detta illustreras i figur 1a. Varje nod består av en linjär funktion plus en konstant, som sedan ges till en så kallad aktiveringsfunktion. Istället för en transformation för varje nod beskrivs ofta istället hela lager där ett lager beskrivs enligt

$$f(\boldsymbol{x}) = g\left(\mathbf{A}\boldsymbol{x} + \boldsymbol{b}\right),\ f : \mathbb{R}^n \mapsto \mathbb{R}^m. \tag{3}$$

$\boldsymbol{x} = [x_1, x_2, ..., x_n]^T$ är utdatan från det föregående lagret, $\mathbf{A} \in \mathbb{R}^{m \times n}$ innehåller nodernas vikter, $\boldsymbol{b} \in \mathbb{R}^m$ deras så kallade bias, och $g : \mathbb{R}^m \to \mathbb{R}^m$ är aktiveringsfunktionen[10]. Här representerar varje element i utdatan värdet av en nod i detta lager.

Att flera lager av dessa funktioner kan approximera okända funktioner visas av de så kallade universella approximationssatserna, vilka kommer i lite olika former beroende på vilka aktiveringsfunktioner som



(a) Traditionellt neuralt nätverk.

(b) Bayesianskt neuralt nätverk.

Figur 1: Schematiska bilder över olika neurala nätverk.

används, hur många noder ett lager får innehålla, samt hur många lager som används [17] [18]. Detta är den teoretiska bakgrunden till hur NN har lyckats göra allt från identifiering av objekt i bilder till röstigenkänning med mera.

2.3 Bayesianska neurala nätverk

Genom att sätta ihop teorin i de två sektionerna ovanför, 2.1, och 2.2, kan en ny typ av neurala nätverk skapas. Dessa kallas bayesianska neurala nätverk (BNN). Ett bayesianskt neuralt nätverk använder sig av stokastiska variabler som parametrar istället för att använda entydiga reella tal, se figur 1b. Dessa stokastiska variabler uppdateras i träningsskedet av nätverket genom att strategiskt använda sig av bayesiansk inferens. Det bayesianska nätverket är tänkt att kunna träna en mer robust modell, som undviker problemet där en modell blir överträngad på en viss datamängd och därav presterar dåligt på ny indata. BNN påstås även kunna ge en bättre bild av osäkerheten i en modells prediktion, då prediktionen är beroende av stokastiska variabler [28].

2.4 Olika former av regression

I neurala nätverk används regression av olika slag under träningsskedet för att uppdatera dess parametrar. I följande kapitel presenteras teorin för relevanta varianter av regression.

2.4.1 Linjär regression

Låt y vara en stokastisk variabel given av

$$y = ax + b + \epsilon, \tag{4}$$

där $a, b \in \mathbb{R}$ är konstanter och $\epsilon \sim N(0, \sigma^2)$ är en normalfördelad stokastisk variabel. Detta innebär att avvikelser från $y = ax + b$ är normalfördelade, det vill säga

$$y \sim N(a + bx, \sigma^2).$$

Detta kallas för en linjär regressionsmodell. Linjär regression innefattar anpassningen, eller regressionen, av denna modell y till data x, vilken görs genom maximering av sannolikheten att vår data x skulle genereras från modellen, vilket görs genom att skatta parametrarna a och b i modellen. Det existerar teoretiska metoder för att göra detta exakt [2], men regressionen kan även utföras via matematisk optimering, med en funktion för sannolikheten av vår data givet modellen som målfunktion.

2.4.2 Polynomiell regression

Vi kan även studera modeller som beskrivs av generella polynom. Vi antar då att y är en stokastisk variabel beskriven av

$$y = \sum_{k=0}^{K} a_k x^k + \epsilon, \tag{5}$$

där $a_k \in \mathbb{R}$, $\forall k \in \{0, 1, ..., K\}$ och $\epsilon \sim N(0, \sigma^2)$. Detta kallas för en polynomiell regressionsmodell, där polynomet i fråga har grad K. Denna generalisering av regressionsproblem från linjära till polynomiella gjordes först av Gergonne [15].

2.4.3 Stokastisk polynomiell regression

Parametrarna i dessa regressionsmodeller behöver inte vara bestämda värden, utan kan även antas vara stokastiska variabler [8]. Låt α_k, $k \in \{0, 1, ..., K\}$, för något $K \in \mathbb{N}$, vara stokastiska variabler med någon fördelning, det vill säga $\alpha_k \sim \mu_k$, där μ_k är en sannolikhetsfördelning. Då kan en stokastisk regressionsmodell ställas upp

$$y = \sum_{k=0}^{K} \alpha_k x^k. \tag{6}$$

En sådan modell kan bättre beskriva samband där de beskrivande parametrarna inte kan antas vara bestämda värden, utan bör antas vara stokastiska. Det är enbart denna typ av stokastiska regressionsmodeller som kommer studeras i denna rapport. En stokastisk regressionsmodell kan ses som det enklaste exemplet av ett BNN, där vi enbart studerar en nod med en funktion beskriven av regressionsmodellen.

2.5 Matematisk optimering

Det centrala problemet inom matematisk optimering handlar om att minimera (eller maximera) en viss funktion, kallad målfunktion, av variabler. Variablerna kan även ha likhets- eller olikhetskrav, som begränsar vilka värden de kan anta. Detta kan formuleras matematiskt som

$$\begin{aligned} \underset{x}{\text{minimera}} \quad & f(x) \\ \text{så att} \quad & x \geq g(x), \\ & x = h(x), \\ & x \in X. \end{aligned} \tag{7}$$

Här är f målfunktionen som ska minimeras, x variablerna, X definitionsmängd för variablerna, g och h begränsande funktioner för variablerna. Ett optimeringsproblem på denna formen kan sedan lösas med en optimeringsalgoritm. Exempelvis kan sjunkgradient användas för problem utan begränsande funktioner [3].

2.6 PyTorch

Projektet skrivs i Python och Python-biblioteket PyTorch används för att realisera neurala nätverk och för möjligheten att automatiskt differentiera tensorer. Informationen i följande avsnitt är tagen från PyTorch officiella dokumentation [14].

En tensor i PyTorch är ett objekt som representerar en multidimensionell matris. En instans av objektet kan vara endimensionell (vektor), tvådimensionell (matris) eller av högre dimension. PyTorch har även diverse inbyggda funktioner för vanliga transformationer av data. Ett exempel är en linjär

transformation (adderat med en konstant) för att modellera ett lager i ett neuralt nätverk. Detta implementeras med hjälp av multiplikation av indata med en matris innehållande vikter, och sedan vektoraddition för tillägg av en konstant.

Linjära funktioner tillsammans med så kallade aktiveringsfunktioner är de byggstenar som utgör majoriteten av neurala nätverk. En aktiveringsfunktion är en deriverbar olinjär funktion placerad mellan linjära funktioner. Utan dessa skulle neurala nätverk vara linjära funktioner och därmed högst begränsade i vad de kan användas till. Två exempel på vanliga aktiveringsfunktioner är ReLU (eng. rectified linear unit) vilken definieras enligt $f(x) = \max(0, x)$ och den så kallade sigmoid funktionen definierad som $f(x) = 1/\exp(-x)$.

I träningen av ett neuralt nätverk är det nödvändigt att beräkna derivatan av felet (ett mått på skillnaden mellan prediktion och faktiskt värde på träningsdata) med avseende på modellens parametrar. Detta görs i PyTorch med hjälp av funktionen `backward()`. Algoritmen bakom denna kallas automatisk differentiering och kommer förklaras i nästa sektion, 2.7. Denna funktions syfte är att genom små förändringar av parametervärdena kunna gå i den riktning (ändring av parametervärden) som minimerar felet snabbast.

2.7 Automatisk differentiering

När metoden `backward()` åberopas på en tensor i PyTorch beräknas gradienten för denna tensor. Det görs med hjälp av metoden automatisk differentiering [5] som varken är symbolisk eller numerisk, utan använder sig istället av duala tal [6]. För att beskriva duala tal introducerar vi det så kallade nilpotenta elementet ε, vilket är ett nollskilt tal som uppfyller att $\varepsilon^2 = 0$. Duala tal är då tal på formen $a + b\varepsilon$, där $a, b \in \mathbb{R}$. Låt oss betrakta Taylorutvecklingen av $f(a + b\varepsilon)$ kring punkten a,

$$f(a + b\varepsilon) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a) b^k \varepsilon^k}{k!}. \tag{8}$$

Genom definitionen av ε, följer det att

$$f(a + b\varepsilon) = f(a) + f'(a) b\varepsilon. \tag{9}$$

Både värdet av funktionen och dess derivata för ett reellt värde a kan därför beräknas samtidigt genom att utvärdera funktionen i det duala talet $a + b\varepsilon$, då vi observerar att koefficienten för $f'(a)$ kommer vara $b\varepsilon$. Detta fungerar även för sammansatta funktioner, eftersom

$$f(g(a + b\varepsilon)) = f(g(a) + g'(a) b\varepsilon) = f(g(a)) + f'(g(a)) g'(a) b\varepsilon. \tag{10}$$

Automatisk differentiering nyttjar kedjeregeln och beräknar de partiella derivatorna i varje delsteg av funktionsevalueringen. Evalueringen av en funktion delas därför upp i delsteg, där varje delsteg representeras av en elementär funktion. Exempelvis skulle funktionen

$$f(x_1, x_2) = \cos(x_1 + x_2) - \log(x_2) \tag{11}$$

kunna delas upp i delfunktionerna

$$\begin{aligned} &v_0(x_1, x_2) = x_1, && v_1(x_1, x_2) = x_2, \\ &v_2(x_1, x_2) = x_1 + x_2 = v_0 + v_1, && v_3(x_1, x_2) = \log(x_2) = \log(v_1), \\ &v_4(x_1, x_2) = \cos(x_1 + x_2) = \cos(v_2), && v_5(x_1, x_2) = f(x_1, x_2) = v_4 - v_3. \end{aligned} \tag{12}$$

På så sätt kan derivator för dessa delfunktioner beräknas och sedan multipliceras med redan beräknade derivator för att erhålla derivatan för varje variabel.

I vilken ordning derivatorna beräknas kan variera. Antingen kan derivatorna beräknas utgående från variablerna, vilket kallas framåt läge (eng. forward mode), eller kan derivatorna beräknas med utgångspunkt från sista beräkningen i funktionsevalueringen, så kallat omvänt läge (eng. reverse mode).

2.8 Mått

I träningskedet av BNN kommer optimering över sannolikhetsfördelningar behöva implementeras, vilket kräver en grundläggande etablering av teorin kring mått. De förenklingar som görs för att kunna implementera optimeringen innebär att de mått som kommer arbetas med är finita, diskreta mått.

Ett mått μ är en funktion $\mu : \mathcal{A} \to \mathbb{R}$, där $\mathcal{A}$ är en σ-algebra i domänen X[13]. Måttet μ har egenskaperna

$$\mu(\emptyset) = 0, \tag{13a}$$

$$\mu(A \cup B) = \mu(A) + \mu(B),\ A \cap B = \emptyset,\ \ A, B \in \mathcal{A}, \tag{13b}$$

$$\mu\left(\bigcap_{i=1}^{\infty} A_i\right) = \lim_{i \to \infty} \mu(A_i),\ \text{för följder}\ \ A_1 \supseteq A_2 \supseteq \dots \tag{13c}$$

Mått definieras som linjära och uppfyller därmed

$$(\mu + \eta)(A) = \mu(A) + \eta(A),\ \ \mu, \eta \in \mathbb{M},\ A \in \mathcal{A} \tag{14}$$

$$(t\mu)(A) = t\mu(A), \qquad \mu \in \mathbb{M},\ A \in \mathcal{A},\ t \in \mathbb{R}. \tag{15}$$

Ett mått kan delas upp i en positiv del och en negativ del. Detta kallas för Jordan-uppdelning och skrivs $\mu = \mu^+ - \mu^-$, där $\mu^+, \mu^- \geq 0$ och ortogonala[20]. Ortogonalitet betyder att måtten uppfyller

$$\begin{cases} \text{Om } \mu^+(A) > 0 \Rightarrow \mu^-(A) = 0,\ \forall A, \\ \text{Om } \mu^-(A) > 0 \Rightarrow \mu^+(A) = 0,\ \forall A. \end{cases} \tag{16}$$

Ett mått μ som uppfyller

$$\mu(A) \geq 0,\ \forall A \in \mathcal{A}, \tag{17}$$

kallas för ett positivt mått. Med Jordan-uppdelning kan därmed mått utryckas som skillnaden mellan två positiva mått. Totala variationen av ett mått definieras som

$$||\mu|| = \mu^+(X) + \mu^-(X). \tag{18}$$

Totala variationen utgör en norm för mått över X. Om totala variationen är finit över X är även $\mu(X)$ finit. Mängden av alla mått med finit norm $\mathbb{M}$ utgör därmed ett Banach rum över X[20]. Väntevärdet och variansen för ett mått definieras respektive som

$$\mathrm{E}[\mu] = \int x\mu(\mathrm{d}x), \tag{19}$$

$$\mathrm{Var}[\mu] = \int (x - \mathrm{E}[\mu])^2 \mu(\mathrm{d}x). \tag{20}$$

Sannolikhetsfördelningar är ett exempel på mått, vilka även har begränsningen att $\mu(X) = 1$. Antag att ett diskret finit mått μ har norm 1 och enbart har massa i punkter $x = (x_1, \dots, x_n)$ Variansen av μ över X beräknas enligt:

$$\mathrm{Var}_X(\mu) = \sum_{i=1}^{n} (x_i - \bar{x})^2 \mu(x_i), \tag{21}$$

där $\bar{x}$ är väntevärdet av μ över X[20].

2.9 Differentiering av mått

I följande underavsnitt är källan till teorin [20] *om inget annat anges.*

En funktion $\psi : \mathbb{M} \to \mathbb{R}$ är Frechét differentierbar om

$$\psi(\mu + \eta) - \psi(\mu) = D\psi(\mu)[\eta] + ||\eta||, \text{ när } ||\eta|| \to 0, \tag{22}$$

där $D\psi(\mu)[\eta]$ är en begränsad linjär kontinuerlig funktional av η. Låt $X = \{1, \ldots, n\}$. Finita mått på X är $\mu = (m_1, \ldots, m_n)$ vilket ger att $\mathbb{M} = \mathbb{R}^n$. $D\psi(\mu)$ är i det fallet en linjär avbildning vid punkten $\mu \in \mathbb{R}^n$, alltså finns det en vektor $d(x, \mu) = (d_1, \ldots, d_n),\ x \in X$, så att för en inkrementering $\eta(x) = (\eta_1, \ldots, \eta_n)$ är

$$D\psi(\mu)[\eta] = \sum_{x=1}^{n} d_x \eta_x = \int_X d(x, \mu)\eta(\mathrm{d}x). \tag{23}$$

I det fallet är $d(x, \eta)$ en gradient.

2.10 Kärndensitetsuppskattning

Då diskreta sannolikhetsmått arbetas med behövs ibland metoder för att approximera måttet med en kontinuerlig fördelning för att undvika problem. En möjlighet är kärndensitetsuppskattning (KDE från eng. Kernel Density Estimation) som är en välkänd metod för att approximera täthetsfunktionen p av ett okänt sannolikhetsmått P genom stickprov [11]. Via ett stickprov $X_1, X_2, ..., X_n \in \mathbb{R}^d$ från P som är oberoende och likafördelade kan vi approximera p som

$$\hat{p}_n(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right), \tag{24}$$

där $K : \mathbb{R}^d \mapsto \mathbb{R}$ är en glatt funktion som kallas kärnfunktionen och $h > 0$ är utjämningskonstanten som bestämmer hur mycket funktionen utjämnas.

Ett annat sätt att skriva ekvation (24) är

$$\hat{p}_m(x) = \frac{1}{mh^d} \sum_{i=1}^{m} w_i \cdot K\left(\frac{x - Z_i}{h}\right), \tag{25}$$

där $Z_1, Z_2, ..., Z_m$ är utfallsrummet av fördelningen P och w_i vikten av utfallet Z_i, det vill säga $P(Z_i) = w_i$ för $i = 1, 2, ...m$. För tillräckligt stora n, det vill säga stora stickprov, är ekvation (24) och (25) ekvivalenta enligt 'De stora talens lag' då frekvensen av $K\left(\frac{x - Z_i}{h}\right)$ kommer närma sig w_i [27].

Ett vanligt exempel på en kärnfunktion är

$$K(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}, \tag{26}$$

det vill säga en standardiserad normalfördelning, även kallad den gaussiska kärnfunktionen.

Av funktioner som fungerar som kärnfunktioner är det inte stor skillnad på deras effektivitet och därmed är inte valet av kärnfunktion särskilt viktigt. Däremot påverkar valet av utjämningskonstanten, h, vår approximation avsevärt [7]. En metod för att få ett värde på h är Silvermans tumregel,

$$h = 0.9 \cdot \min(\sigma, IQR/1.34) \cdot n^{-\frac{1}{5}} \tag{27}$$

där IQR är kvartilavståndet av fördelningen P, det vill säga avståndet mellan första och tredje kvartilen [7]. En nackdel med Silvermans tumregel är att den inte är lika användbar i fallet då fördelningen vars täthetsfunktion vi vill uppskatta inte är Gaussisk [7]. Ett annat sätt att välja utjämningskonstanten är Sheath och Jones metod vilket ger ett h som i många fall kommer vara bättre då metoden anpassar sig lättare efter situationen [24].

3 Modeller och metoder

Skattning av mått kan genomföras med två olika typer av modeller: parametrisk och icke-parametrisk. Med en parametrisk modell görs antagandet att fördelningen följer en tidigare känd fördelning som kan beskrivas fullkomligt av några få parametrar, exempelvis en normalfördelning. En icke-parametrisk modell däremot är generellt sett mer komplext då stora antaganden om hur fördelningen ser ut inte kan göras. Istället behöver värden skattas i en större mängd punkter för att beskriva fördelningen [22]. Nedan kommer teorin bakom ett urval av metoder för att optimera modellerna i båda kategorierna beskrivas som sedan kommer implementeras i Python. Under projektets gång har andra alternativ alternativ än modellerna nedan testats, exempelvis straffunktioner- och Lagrangemultiplikator-metoden, se appendix B.3-B.4. Dessa används dock inte i jämförelsen då de inte presterade tillräckligt bra under de givna förhållandena.

3.1 Parametrisk modell

När en modell antas vara parametrisk kan vi fullständigt beskriva den med några enstaka parametrar. Ett exempel på detta är normalfördelningen som har två parametrar: medelvärde och standardavvikelse. Två metoder som optimerar denna modellen testas.

3.1.1 Linjärkombination av stokastiska variabler

Följande teori är opublicerad men tillskrivs handledare Sergei Zuyev.

Låt $\eta(x)$ vara en funktion definierad enligt

$$\eta(x) = \sum_{k=1}^{K} \alpha_k h_k(x), \tag{28}$$

där $\alpha_k \sim \mu_k$ är oberoende stokastiska variabler och $h_k(x)$ antas vara kontinuerliga funktioner av x $\forall k$. I många fall kan det antas att en normalfördelning eller en sammansättning av normalfördelningar kan beskriva vår data. Det antagandet kommer göras här, men det generella fallet samt dess lösning finns beskrivet i appendix B.2. Om vi antar att α_k är oberoende och normalfördelade, det vill säga $\alpha_k \sim \mathcal{N}(m_k, \sigma_k^2)$ $\forall k$, kommer $\eta(x_i)$ också vara normalfördelad, eftersom det är en linjär kombination av normalfördelade stokastiska variabler med icke-stokastiska koefficienter $h_k(x)$ för alla x [19]. Fördelningen följer

$$\eta(x) \sim \mathcal{N}(m(x), \sigma^2(x)),\ \forall x, \tag{29}$$

där

$$m(x) = \sum_{k=1}^{K} m_k h_k(x), \tag{30}$$

$$\sigma^2(x) = \sum_{k=1}^{K} (\sigma_k h_k(x))^2. \tag{31}$$

Detta ger täthetsfunktionen för $\eta(x)$ som

$$f_{\eta(x)}(y) = \frac{1}{\sqrt{2\pi\sigma^2(x)}} \exp\left(-\frac{(y - m(x))^2}{2\sigma^2(x)}\right), \tag{32}$$

som kan kombineras med log-likelihoodfunktion i avsnitt 3.3.1 för att skatta parametrar.

3.1.2 Polynomisk modell med normalfördelade koefficienter

Om den tidigare nämnda linjärkombinationen antas vara ett polynom av grad K kan vi förenkla ytterligare till stokastisk polynomiell regression, se avsnitt 2.4,

$$\eta(x) = \sum_{k=0}^{K} \alpha^{(k)} x^k. \tag{33}$$

Istället för att explicit söka varje parametrar för varje stokastisk variabel i detta polynom kan ett kombinerat medelvärde med associerad varians hittas genom att separera dem till två separata polynom. Medelvärdet μ är ett polynom av grad K som det ursprungliga polynomet, där vi ersätter stokastiska variablerna med skalära parametrar a_μ,

$$\mu(x) = \sum_{k=0}^{K} a_\mu^{(k)} x^k. \tag{34}$$

Eftersom varians är kvadratiskt beroende av x enligt ekvation (21) kommer variansen σ^2 kräva ett polynom av graden $2K$,

$$\sigma^2(x) = \sum_{k=0}^{K} a_\sigma^{(k)} x^{2k}. \tag{35}$$

Ekvation (34) och (35) är båda polynom som kan lösas med generell polynomiell regression. I praktiken implementeras detta i PyTorch med hjälp av två parallella linjära lager utan bias. Ett lager för medelvärde och ett för variansen. En tensor med x för varje exponent matas in i respektive lager: $[1, x, x^2 ... \ x^K]$. Lösningen är en kurva för μ som visar medelvärdet i varje punkt x samt en kurva för σ^2 som visar totala variansen. Detta gör det enkelt att visualisera resultatet och om så önskas kan stokastiska variablernas parametrar härledas från de skalära parametrarna.

3.2 Icke-parametrisk modell

Nackdelen med en parametrisk modell är att antaganden om måttet måste göras. Användaren måste i förväg bestämma vilka parametrar modellen ska anpassa och hur dessa beskriver måttets form. En lösning är att istället diskretisera måttet. Ett mått med ett stöd på ett visst intervall kan modelleras med jämnt fördelade atomer (punkter) från detta intervall vilka har varsin vikt eller massa associerad med sig. I appendix B.5 illustreras hur fallet med diskreta sannolikhetsmått kan implementeras med den så kallade softmax-funktionen. Denna funktion är enkel att integrera i neurala nätverk, men har potentiellt långsam konvergens beroende på var i parameterrummet minimipunkten ligger. Nedan introduceras en alternativ metod som effektiviserar optimering av diskreta mått avsevärt.

3.2.1 Optimeringsalgoritm över finita diskreta mått (OFDM)

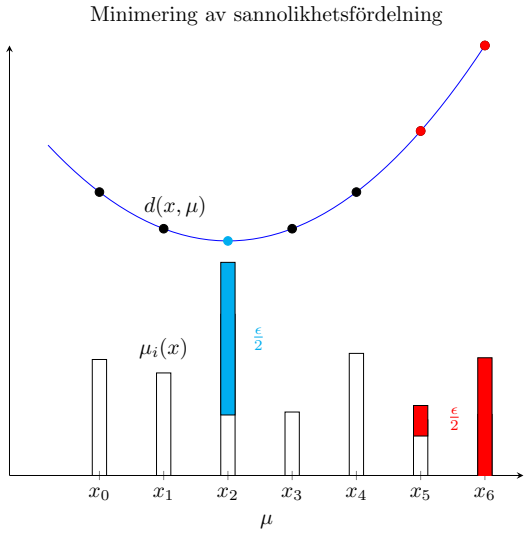
En metod för att minimera en funktion över ett diskret mått är att flytta en liten mängd massa i fördelningen [21, s. 119]. För att måttet ska fortsätta vara en sannolikhetsfördelning, alltså summera till 1, behöver all massa som subtraheras vid en punkt adderas vid en annan punkt. Låt μ vara en fördelning och F en funktion. Vi vill hitta ett η som minimerar funktionen F, alltså

$$\min_{\eta} \quad F(\mu + \eta) \tag{36a}$$

$$\text{så att} \quad 0 \leq \mu_i + \eta_i \leq 1, \ \forall i \in \{1, \ldots, n\}, \tag{36b}$$

$$||\mu|| = 1, \tag{36c}$$

$$\sum_{i=1}^{n} \eta_i = 0. \tag{36d}$$



Figur 2: För att minimera måttet över F adderas positiv massa (turkos stapel) till fördelningen μ där gradienten $d(x,\mu)$ är som minst (turkos punkt) och negativ massa (röda staplar) där gradienten $d(x,\mu)$ är som störst (röda punkter).

Eftersom η är konstant med avseende på μ kan minimering av (22) lösa minimering av (36a). Vi får då

$$F(\mu+\eta) = F(\mu) + DF(\mu)[\eta] + ||\eta||, \tag{37}$$

$$F(\mu+\eta) \approx F(\mu) + \sum_{i=1}^{n} d_i\eta_i, \tag{38}$$

enligt (22) och (23) och när $||\eta||$ är liten. Med hjälp av (22) kan alltså minimeringsproblemet approximeras genom att placera ut massorna av η_i på olika delar av gradienten $d(x,\mu)$. I figur 2 placerar vi ut den positiva massan där gradienten är som minst och negativ massa där gradienten är som störst för att minimera måttet över F (36a).

Det finns två utfall vid fördelning av den negativa massan. Antingen kan all massa placeras vid punkten där gradienten är störst eller bara en del av massan för att fortfarande uppfylla (36b). Efter att all möjlig negativ massa har placerats ut på en punkt itereras metoden på punkten med näst störst gradient med den återstående massan. Detta upprepas tills all massa har placerats.

3.2.2 Steg för implementering

Vi kan definera θ^+ och θ^- som den positiva respektive negativa massan som har distribuerats på η. Mer konkret är

$$\theta^+ = \sum_{i=1}^{n} \eta_i^+, \tag{39}$$

$$\theta^- = \sum_{i=1}^{n} \eta_i^-. \tag{40}$$

Nedan följer steg för hur massorna distribueras från given gradient $d(x,\mu)$.

1. Låt $\theta^+, \theta_0^- = \frac{\epsilon}{2}$ och $I = \emptyset$ och $k = 0$.

2. Låt i vara punkten där gradienten är minst, $i = \arg\min_{i \in \{1, \dots, n\}} d(x, \mu_i)$. Placera massan θ^+ på μ_i, vilket ger $\mu_i := \mu_i + \theta^+$.

3. Låt i vara punkten där gradienten är störst, $i = \arg\max_{i \in \{1, \dots, n\} \setminus I} d(x, \mu_i)$. Bestäm massa enligt $\eta_i^- = \min(\theta_k^-, \mu_i)$ och fördela $\mu_i := \mu_i - \theta^-$. Uppdatera $\theta_{k+1}^- = \theta_k^- - \eta_i^-$.

   Om $\theta_{k+1}^- = 0$: Stopp!

   Annars: Uppdatera $k := k + 1$, lägg till $I := I \cup \{i\}$ och **upprepa steg 3.**

Eftersom $\theta^+ = \theta^-$ har storleken av μ inte ändrats utan massan har enbart förflyttats för att minimera måttet över F. Dessa steg upprepas tills gradienten är konstant och minimal i alla atomer med nollskild massa. Måttet uppfyller då nödvändiga villkor för minimeringsproblemet.

3.2.3 Implementation i Python

Den huvudsakliga klassen `Measure`, med tillhörande metoder, implementerar ett diskret mått. Klassens metoder är dokumenterade och en instans av klassen beskrivs i huvudsak av dess vikter (parametern `self.weights`), samt en tillhörande plats (parametern `self.locations`) vilka tillsammans beskriver atomer. Koden hittas i appendix C.1. Funktionerna i denna klass uppfyller vanliga funktioner kring räkning med mått, så som att se om ett givet mått är ett sannolikhetsmått, vad ett mått har för total massa, att hitta stödet för ett mått, hur ett mått visualiseras, med mera.

En instans av klassen `Optimizer` skapas sedan med hjälp av en instans (eller lista av instanser) av klassen `Measure` samt en textsträng som specificerar vilken målfunktion som bör användas. Dess metod `minimize` används sedan för att optimera måtten. Koden hittas i appendix C.2. Kärnan av `minimize` funktionen är algoritmen som beskrivs ovan. Mellan varje iteration verifierar funktionen om resultatet uppfyller kraven för att avsluta som presenteras i sektion 3.2.2 och avlutar i så fall. Annars kollar den om resultatet faktiskt blev bättre, om det inte är fallet minskas steglängden och den senaste förändringen återställs; den kontrollerar även om steglängden har blivit mindre än ett givet tröskelvärde och i så fall anses optimeringen vara färdig. Om detta kriteriet ej uppnås avslutar algoritmen automatiskt efter ett givet antal iterationer.

Dessutom finns klassen `Check` som används för att undersöka hur likt ett anpassat mått är den ursprungliga fördelningens data. En instans av `Check` tar in en instans av `Optimizer`, modellen som används och data från den ursprungliga sannolikhetsfördelningen. Koden finns i appendix C.3.

3.3 Målfunktioner

Vid optimering av mått behövs en metod för att avgöra om den nya hypotesen är bättre än den förgående. Detta kan uppnås med en målfunktion som avbildar måttet på en tallinje vars värde indikerar hur bra måttet representerar given data. Ett värde närmare 0 indikerar att måttet är bättre anpassat till given data. Det finns flera relevanta målfunktioner att applicera på mått, deras relevans motiveras av poängfunktioner, se appendix B.6. Nedan presenteras de tre som undersöks.

3.3.1 Maximum likelihood-metoden

En vanlig målfunktion med en stark grund i matematisk statistik är maximum likelihood [2]. Låt $x_1, ..., x_n$ vara ett stickprov från en stokastisk variabel X med täthetsfunktion f_θ. Vi definierar den så kallade likelihoodfunktionen

$$L(\theta) = \prod_{i=1}^{n} f_\theta(x_i). \tag{41}$$

Det parametervärde $\hat{\theta}$ som maximerar likelihoodfunktionen kallas maximum likelihood-skattningen (ML-skattningen). Då det i många fall kan vara lättare att hantera en summa, kan vi definiera den logaritmska sannolikheten:

$$l(\theta) = \log\left(\prod_{i=1}^{n} f_\theta(x_i)\right) = \sum_{i=1}^{n} \log(f_\theta(x_i)) \tag{42}$$

Även denna funktion kommer maximeras av ML-skattningen. Båda dessa funktioner kan ses som ett mått på sannolikheten att ett stickprov producerades av en viss fördelning med parameter θ. Då det ofta är minimeringsproblem som studeras inom matematisk optimering, betraktas den additiva inversen av likelihoodfunktionen. Framöver benämns den som NLL-funktionen (negativ log-likelihood).

3.3.2 Chi-kvadrat-metoden

En annan målfunktion som är av intresse att testa är Chi-kvadrat (χ^2). Den så kallade Chi-kvadrat-metoden grundar sig i Chi-kvadrat-testet. Nedan följer en beskrivning av testet taget från [26].

Låt X vara en diskret stokastisk variabel, som kan anta värdena $x_i,\ i = 1, 2, ..., n$, och en täthetsfunktion enligt $P(X = x_i) = p_i$. Vi kan definiera

$$C = \sum_{i=1}^{n} \frac{(f_i - kp_i)^2}{kp_i} = \sum_{i=1}^{n} \frac{f_i^2}{kp_i} - k, \tag{43}$$

där f_i är frekvensen av värdet x_i från ett stickprov på k värden från fördelningen. Då k är stort approximerar C Chi-kvadrat-fördelningen med $n - 1$ frihetsgrader. Värdet kan därför användas som mått på hur nära stickprovet är den förväntade fördelningen. Om vi nu låter de relativa frekvenserna $\frac{f_i}{k}$ bestämmas av vikterna från ett förutspått sannolikhetsmått $\hat{Y} = (\hat{y}_1, ..., \hat{y_n})$ och $Y = (y_1, ..., y_n)$ vara fördelningen från observerad data. Då kan vi skriva C som $k\left(\sum_{i=1}^{n} \frac{\hat{y_i}^2}{y_i} - 1\right)$. För att $\hat{Y}$ ska bli så nära Y som möjligt vill vi då minimera $\sum_{i=1}^{n} \frac{\hat{y_i}^2}{y_i}$. Vi kan då definiera målfunktionen som kallas Chi-kvadrat förlust (eller bara Chi-kvadrat):

$$L = \sum_{i=1}^{n} \frac{\hat{y_i}^2}{y_i} \tag{44}$$

Denna funktion föreslogs först i *On the criterion that a given system of deviations...* [23] och analyserades vidare som en del av ett TNN i *Chi-square Loss for Softmax: an Echo of Neural Network Structure* [26].

Något som påpekas i den senare artikeln är att det kan förekomma frekvenser y_i i observerad data som ej är nollskilda. Detta blir problematiskt då dessa förekommer i nämnaren i Chi-kvadrat. För att lösa detta kan så kallad label smoothing användas. Denna teknik omfördelar vikterna så att alla vikter är nollskilda, utan att att påverka den generella formen av sannolikhetsmåttet. En exakt metod som kan användas för detta är att justera vikterna enligt

$$y_i^{LS} = y_i(1 - \alpha) + \frac{\alpha}{n}, \tag{45}$$

där α är ett litet positivt tal ($\alpha \in \mathbb{R}^+ :\ \alpha \approx 0$). Dessa justerade vikter kan sedan användas i funktionen.

3.3.3 ESSR-metoden

Väntevärdet av summan av kvadrerade rester eller ESSR (eng. Expected Sum of Square Residuals) innebär beräkning av

$$\arg\min_{\theta} \mathrm{E}_{\theta}\left[\sum_{i=1}^{n}(\hat{y}_i(\theta) - y_i)^2\right],$$

där y_i är värden tagna från den riktiga fördelningar och $\hat{y}_i$ är en uppskattning av y_i gjord av modelen. θ är fördelningen av parametrar för det diskreta måttet. I fallet linjär regression skulle värden från fördelningen θ kunna vara par av lutning och skärningspunkt på y-axel för en linje som skall anpassas till en datamängd. ESSR-metoden kan betraktas som den probabilistiska versionen av minstakvadratmetoden, vilken är standardmetoden för regression där parametrarna är vanliga skalärer.

3.3.4 Val av målfunktion

ESSR-metoden har egenskapen att lösningen degenererar till ML-skattningen av fördelningen, för mer infromation läs appendix B.1. Detta innebär att all vikt (eller vid parametriserade fördelningar, så mycket vikt som är möjligt) kommer att läggas på vad modellen tror är det mest sannolika värdet. Därmed reflekterar ej den resulterande fördelningen det faktum att vi fortfarande har viss osäkerhet kring parametervärdet. Eftersom syftet med det bayesianska tillvägagångsättet är att beskriva osäkerheten i resultatet är ESSR-metoden inte ett lämpligt val. Detta är ej fallet för Chi-kvadrat-metoden och log-likelihood. Fördelningen bibehåller en osäkerhet i form av en mängd olika möjliga världen. Efter vidare testning av dessa två dras slutsatsen att log-likelihood är det mer lämpliga valet då Chi-kvadrat vid flera tillfällen fastnade i lokala minimum. Eftersom ett diskret mått har ett begränsat antal vikter kommer $f_\theta(x_i)$ inte existera för alla x_i vid användning av log-likelihood. En lösning är att avrunda till närmaste x som existerar i det diskreta måttets utfallsrum. Detta kommer dock med sina nackdelar. Avrundningar minskar inte bara precisionen utan kan även skapa problem vid optimering då gradienten blir diskontinuerlig. Därför approximeras först måttets täthetsfunktion med hjälp av kärndensitetsuppskattning för att få mer exakta värden på $f_\theta(x_i)$.

4 Resultat

I detta avsnitt presenteras metodiken för tester som görs samt dess resultat. I syfte att förenkla återskapandet av resultaten visas här versioner av Python och andra relevanta bibliotek som använts:

- Python: Version 3.10.6
- PyTorch: Version 1.13.1
- Numpy: Version 1.24.1

4.1 Jämförelse mellan parametriska och icke-parametriska metoder

För att utvärdera OFDM och jämföra den med andra modeller som beskrivits i rapporten, ska fyra olika tester genomföras. Låt $\alpha_i, \beta_i, \gamma_i$ vara tre normalfördelade stokastiska variabler med parametrar μ_i och σ_i, $i = 1, 2...., 50$, som också är tagna från två normalfördelningar. Testerna består av anpassning till följande modeller:

1. En normalfördelad stokastisk variabel (α_i)
2. En linjär modell med en stokastisk variabel som koefficient ($\alpha_i \cdot x$)
3. En linjär modell med två stokastiska variabler som koefficienter ($\alpha_i + \beta_i \cdot x$)
4. En kvadratisk modell med tre stokastiska variabler som koefficienter ($\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$)

De tre metoderna som ska jämföras är de två parametriska metoderna Linjärkombination av stokastiska variabler och Polynomisk modell med normalfördelade koefficienter, samt den icke-parametriska OFDM. För varje test används metoden med 50 olika värden av parametrarna μ_i och σ_i och för varje sådant värde används tre olika storlekar på datamängder, 100, 500 respektive 1000 datapunkter vardera. Detta innebär 150 anpassningar för varje test och för varje metod. Alla dessa datamängder genereras via koden i appendix C.4.

I de olika testerna mäts tre olika metriker. **Prestanda** skattar sannolikheten att metoden kan anpassa en modell väl till godtycklig träningsdata. Alltså ett värde 0-1 där högre är bättre. Dessutom beräknas ett 95% konfidensintervall för prestandan. Se avsnitt 4.1.1 för en mer genomgående förklaring. Sedan mäts även det genomsnittliga antalet epoker som varje metod behöver för att konvergera, samt genomsnittlig körtid för metodernas konvergens. Dessa metriker kallas **konvergenstid (epoker)** och **konvergenstid (s)**. Resultatet av testerna kan ses i tabellerna i appendix A.1 och sammanfattas i avsnitt 4.2.

4.1.1 Prestanda

Implementationen av beräkningen av prestanda skiljer sig mellan OFDM och de parametriska metoderna, men idén är den samma. Först beräknas för varje invärde x ett stickprov av modell-värden $\hat{y}$ baserat på den anpassade modellen. Genom att bortse från de 2.5% lägsta, respektive högsta värdena i stickprovet erhålles ett approximativt konfidensintervall på 95%. Detta konfidensintervall kommer närma sig det teoretiska då antalet datapunkter växer. Sedan testas om den faktiska datapunkten y ligger i detta intervall och om inte räknas det som en "miss" (se figur 3). När detta gjorts för varje datapunkt x, beräknas ett teoretiskt konfidensintervall på 95% för hur många datapunkter y som bör missa. Detta görs genom att se antalet missar som en binomialfördelad stokastisk variabel, där antalet försök är lika med antalet datapunkter och sannolikheten för ett positivt utfall (en miss i detta fall) är 5% (se figur 4).

Metriken som beräknas i slutändan är andelen anpassningar där antalet missar ligger i konfidensintervallet. Denna parameter antas i sig vara binomialfördelad. Värdet kommer vara en skattning av det verkliga värdet på sannolikheten för att en modell producerad av metoden är välanpassad till

godtycklig träningsdata. En osäkerhet för denna metrik erhålles därför också, vilken ges av ett konfidensintervall på 95%. I beräkningarna antar vi i de flesta fall att metriken är normalfördelad, då detta är en bra approximation av binomialfördelningen när värdena på prestandan inte är nära 0 eller 1. Detta på grund av att normalfördelningen har svansartill skillnad från binomialfördelningen. I de fall prestandan var mycket nära eller lika med 0 eller 1 används därför ett 95% konfidensintervall för binomialfördelningen istället. För att se hur den faktiska implementationen gjordes för OFDM, se klassen `Check` i C.3, och för de parametriska metoderna, se funktionen `misses` i C.6 och C.7.

4.1.2 Konvergenstid för optimeringsalgoritmen över finita diskreta mått

Som det står ovanför undersöks konvergenstiden på två sätt, mätt i sekunder och mätt i antal epoker. Dessa mätningar sker på lite olika sätt för de olika metoderna och i detta avsnitt detaljeras hur detta görs för OFDM. När metoden `minimize` kör kan koden sluta av tre anledningar, den har kört max antal epoker som tillåts, den har nått ett optimum eller om steglängden är för liten. I fallen av att den nått ett optimum returnerar metoden den dåvarande tiden och epoken den är i. Om den däremot stannar av steglängden eller epokerna kommer den undersöka om målfunktionen inte har ändrat värde under de senaste fem eller fler epokerna. Om den inte ändrat under fem eller fler epoker returneras den första tiden och epoken där målfunktionen inte ändrades, annars returneras den nuvarande tiden och epoken.

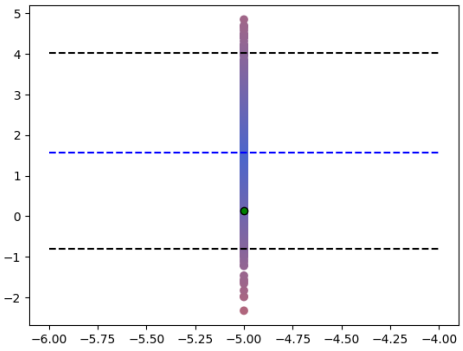
Värt att nämna är att metoden körs med ett max-antal epoker, vilket betyder att den inte kan köra för evigt. Den kan därför stanna på grund av att den når max-antalet innan den konvergerar och då räknas max-antalet som konvergenstiden. Max-antalet valdes till 4 000 epoker.

4.1.3 Konvergenstid för parametriska metoder

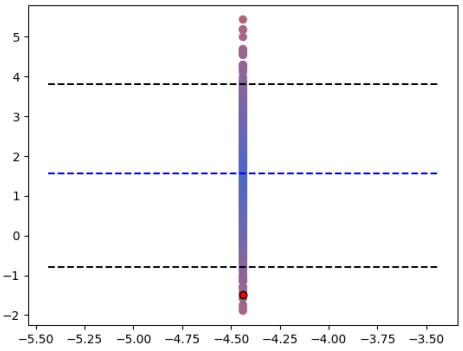
För de parametriska metoderna beräknas konvergenstid genom att i varje steg beräkna skillnaden i värde på målfunktionen mellan föregående och nuvarande steg. Denna skillnad jämförs sedan med en tolerans och om skillnaden är mindre än denna anses konvergens ha uppnåtts. Toleranserna väljs till 10^{-4} för parametriskt neuralt nätverk och 10^{-15} för metoden med linjärkombinationer av stokastiska variabler. Toleransen valdes till högre för parametriskt neuralt nätverk då denna metod observerades fluktuera runt en konvergenspunkt ibland, därför ansågs det att den bör anses konvergera för högre skillnader.

4.2 Resultat från tester

Det fullständiga resultatet finns i appendix A.1. Linjärkombination av normalfördelade stokastiska variabler får en ideal prestanda (1.0) på näst intill alla tester. Konvergenstiden verkar öka linjärt med antal datapunkter medan antal parametrar knappt påverkar tiden. Polynomisk modell med normalfördelade koefficienter påvisar sämre prestanda för fler datapunkter. Prestanda minskar även med antal datapunkter. OFDM har överlägset lägst prestanda i de flesta testerna men prestanda ökar med antalet datapunkter. För 1000 datapunkter uppnås en maximal prestanda av 0.92. Både OFDM och polynomisk modell har en betydligt lägre konvergenstid för större antal datapunkter än linjärkombination av stokastiska variabler.

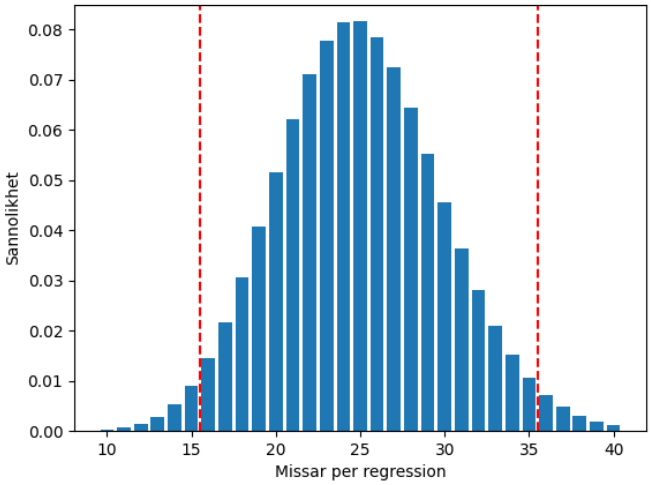


(a) Datapunkt innanför konfidensintervall



(b) Datapunkt utanför konfidensintervallet (räknas som en miss)

Figur 3: Stickprov av värden från normalfördelningar, med approximativt 95% konfidensintervall markerat med svart streckad linje och medelvärde markerat med blå streckad linje. I vänster figur är datapunkten innanför konfidensintervallet (grön punkt) och i höger figur är datapunkten utanför intervallet (röd punkt).



Figur 4: Täthetsfunktion för en binomialfördelad stokastisk variabel med sannolikhet för positivt utfall $p = 0.05$ och antal försök $n = 500$

5 Diskussion

I detta avsnitt görs jämförelser mellan OFDM-algoritmen och parametriska metoder och förslag på vidare forskning studeras. Sedan diskuteras kärdensitetsuppskattning och tidskomplexitet för större modeller samt samhälleliga och etiska aspekter utifrån arbetets syfte.

5.1 Jämförelse

I detta avsnitt hänvisas det till tabellerna i appendix A.1 om inget annat anges. Vi kan observera från resultaten att OFDM överlag har sämre prestanda än de parametriska metoderna för 100 och 500 datapunkter. Särskilt i fallet med modellen a, då den har prestanda 0.00 för alla mängder datapunkter. Detta kan bero på olika orsaker. En hypotes är att det beror på låg varians i vår data. I övriga modeller förstärks variansen då den multipliceras med x. För modellen a sker inte detta och denna mindre varians av vår data kan vara orsaken till låg varians i modellerna OFDM anpassar.

OFDM verkar ha relativt bra prestanda för högre antal datapunkter och presterar nära 0.95-1.00 då 1 000 datapunkter används. Den polynomiska modellen presterar bra för 100 och 500 datapunkter, men verkar prestera sämre för de flesta modellerna vid 1 000 datapunkter. Metoden med linjärkombination av stokastiska variabler verkar prestera inom det förväntade intervallet i alla tester.

Gällande prestandan för OFDM är det dock värt att nämna att den kan nå maxantalet epoker utan att konvergera. Prestandan skulle därför kunna öka om vi höjer detta maxantal, då den skulle kunna konvergera till bra anpassade modeller i fler fall. Detta påverkar såklart även konvergenstiden för OFDM, då denna metrik också skulle öka om maxantalet epoker ökas och många av regressionerna i nuläget inte konvergerar.

Om vi ser till konvergenstiden i sekunder är den som högst för linjärkombinations-metoden. Detta implicerar att denna är mer beräkningstung, då den verkar ha lägre antal epoker till konvergens jämfört med övriga. Detta kan innebära att den är mindre lämplig att applicera i ett neuralt nätverk, då träning av ett sådant kan ta längre tid än vad som är praktiskt genomförbart. Vi kan dock även observera att konvergenstiden tycks vara relativt oförändrad vid en ökning av antalet parametrar i modellen som skattas. Den skulle därför kunna vara att föredra vid modeller med många parametrar jämfört med OFDM och polynomiska modellen, vars beräkningstid tycks öka med antalet parametrar. För de modeller som studeras här verkar dock OFDM och polynomiska modellen vara att föredra rent tidsmässigt. Men som tidigare nämnts, påverkas den genomsnittliga konvergenstiden av maxantalet epoker om metoderna inte hinner konvergera.

Med avseende på antal epoker till konvergens är linjärkombinations-metoden mest effektiv. Den verkar som sagt inte heller påverkas i sin konvergenstid, även med avseende på antal epoker, av antalet parametrar. Detta skulle kunna förändras vid ett högre antal parametrar, men är intressant att observera. Polynomiska modellen verkar vara den metod som konvergerar efter högst antal epoker av metoderna i jämförelsen. Den verkar dock ha låg beräkningstid per epok, vilket gör antalet epoker mindre intressant i praktiken.

5.2 Val för KDE

När KDE i målfunktionen beräknas används varianten från ekvation (25) istället för ekvation (24). Anledningen till valet är att målfunktionen kräver en referens till w_i för att `backward()` ska kunna beräkna gradienten med avseende på vikterna. Om ekvation (24) används måste ett stickprov genereras vilket bryter kopplingen med vikten w_i. Genom att använda sannolikhtesmåttet direkt i uppskattningen undviks detta problem. Ekvationen uttrycks dock något annorlunda i koden då fördelningen X för oss följer en modell som generellt kan skrivas $y = \eta(x, \theta)$ där $\theta = (\alpha_1, ..., \alpha_m) \in \mathbb{R}^m$ är en fördelning med vikten w_i för ett utfall θ_i. Detta ger oss

$$\hat{p}_m(y) = \frac{1}{mh} \sum_{i=1}^{m} w_i \cdot K\left(\frac{y - \eta(x, \theta_i)}{h}\right). \tag{46}$$

I alla fall där KDE har använts i denna rapport har Silvermans tumregel utnyttjats för att välja lämplig bandbredd h. Detta fungerar bra då alla mått som använts har varit unimodala normalfördelningar. Försök gjordes även med Sheath and Jones som är en mer generell metod och fungerar i nästan alla exempel, men då erhölls problem vid implementation via biblioteket `KDEpy`. Paketets funktioner gav enbart värden på $\hat{p}_m(y)$ för jämnt fördelad data y, d.v.s. lika stort avstånd mellan alla datapunkter. I fall då $\hat{p}_m(y)$ sökes för data som kommer från en sannolikhetsfördelning är den inte jämnt fördelad. I och med detta problem och att en egen implementering av Sheath and Jones hade varit alltför tidskrävande gjordes valet att ändå använda Silvermans tumregel.

5.3 Tidskomplexitet för större modeller med OFDM

För OFDM metoden betraktas alla möjliga kombinationer av vikter från de olika måtten i beräkningen av felet. Vid regression med upp till tre koefficienter är detta ej ett stort problem men detta komplexitetsproblem lär bli signifikant när antalet mått ökar; antalet möjliga kombinationer av vikter från de olika måtten växer nämligen exponentiellt med antalet mått för nuvarande implementation.

Då neurala nätverk har många koefficienter inblandade innebär detta att motsvarande BNN skulle involvera många fördelningar. Därav är möjliga metoder för att hantera dessa problem en central del i de framtida efterforskningar som behövar göras om denna metod ska bli lämplig för användning i större BNN.

5.4 Vidare forskning

Fler kvantitativa tester behöver troligtvis utföras för att se hur OFDM presterar vid regression och för att avgöra dess lämplighet för träning av BNN. Dels undersöks inte algoritmen som en del av träningsprocessen för ett BNN i denna rapport, utan enbart småskalig optimering med ett få antal parametrar studeras. Det hade varit intressant att implementera ett BNN med denna algoritm och se hur nätverket presterar i en relevant tillämpning. Ytterligare vore det relevant att undersöka redan implementerade paket för BNN, exempelvis python biblioteket Pyro [9].

Denna jämförelse har heller inte tagit hänsyn till inte att metoden är beroende av antalet vikter som måtten består av. Det är av intresse att testa hur metoden presterar i de observerade metrikerna för olika mängder av vikter och då undersöka om det finns ett optimalt antal vikter. Något som också kan variera prestandan är intervallet för måtten som används. I utvecklingen av metoden kunde vi observera att prestandan varierade med längden och placeringen av intervallet relativt parametrarna i modellen som genererade data. Detta fenomen har dock ej kvantifierats i denna rapport och det skulle vara intressant att undersöka vidare. Detta kan troligtvis även samspela med antalet vikter. I testerna anpassades intervallet och antalet vikter för att förbättra anpassningen till testdata. För exakt utseende av denna anpassning, se appendix C.5.1-C.5.4.

Det hade även varit intressant att återskapa testerna med andra datamängder, för att se om detta påverkar prestandan. Möjligheten finns att dessa datamängder leder till resultat som än missvisande för metodernas allmänna prestanda. Att beräkna andra metriker för metoderna som kvantifierar deras prestanda kan också vara intressant. Exempelvis kan ett lågt värde i metriken som vi kallar prestanda bero på antingen en överanpassning, med för låg varians, eller en dålig anpassning. En annan metrik skulle möjligtvis kunna visa detta och då ge en bättre bild av metodernas prestanda.

Att även testa med data som genererats från andra fördelningar än normalfördelningar hade varit av intresse. Eftersom OFDM är icke-parametrisk och inte anpassad till specifikt normalfördelningar, som de parametriska i denna rapport, borde den prestera bättre för icke-normalfördelad data än de

andra metoderna. Även tester av algoritmen för data som ej genererats spexifikt för testerna hade varit intressant. Ett exempel skulle vara att nyttja OFDM för att anpassa en modell till någon form av ekonomisk data, så som aktievärden eller fastighetspriser.

5.5 Samhälleliga och etiska aspekter

Arbetets undersökning av optimeringen över mått görs med syftet att kunna göra träningen av bayesianska nätverken effektivare. Det är därför viktigt att också studera vilka samhälleliga konsekvenser ett bayesianskt neuralt nätverk kan ge upphov till när de tillämpas.

Neurala nätverk och deras tillämpningar kan i många fall lägga grunden för kraftfulla tekniska verktyg och idag finns det redan implementerade neurala nätverk som används av industri och näringsliv. Europeiska kommissionen har delat in områden, där artificiell intelligens kan tänkas implementeras, i olika riskklasser för att på så sätt kunna bestämma om vissa typer av tillämpningar bör regleras samt vilka krav på transparens som bör uppfyllas. Exempelvis är anställning samt personalhantering områden klassas som högriskområde där implementationer av maskininlärningsprocesser bör bedömas noggrant innan det släpps ut på marknaden [25].

När neurala nätverk implementeras är det dock av stor vikt att överväga konsekvenser som dessa tillämpningar kan ge upphov till, både på samhället i stort såväl som på individnivå. För att undvika att etiska problem uppstår i arbetet med implementeringen av bayesianska neurala nätverk har etiska aspekter som är generella för alla neurala nätverk beaktas under arbetets gång. Många av dessa finns beskrivna på EU-kommisionens hemsida [4].

De etiska aspekter som är generella för neurala nätverk är exempelvis partiskhet i träningsstadiet av de neurala nätverken, säkerhetsmässiga aspekter, integritet och transparens hos nätverken samt ansvar och gottgörelse vid negativ påverkan.

Neurala nätverk tränas med hjälp av stora mängder data och liknas ibland med en svart låda då det kan vara svårt att veta exakt vad som sker i träningsstadiet. Hur väl dessa nätverk sedan kan förutspå resultat på ny indata är helt beroende av vilken data nätverket tränats på. Det tränade nätverket kan i sig bli partiskt om den data som det tränats med är partisk på något vis. Risken för en partiskhet i det resulterande nätverket är viktig att vara medveten om i valet av träningsdata så att denna exempelvis inte innehåller mer data från vissa etniska grupper än andra samt att den är mångfaldig och representativ.

Konsekvensen av ett partiskt nätverk kan vara att orättvisa eller diskriminerande förutsägelser uppstår. Eftersom neurala nätverk och deras tillämpningar skulle kunna användas i många delar av samhället, som exempelvis domstol eller polisiär verksamhet, och därför påverka både individer och samhället i stort är det viktigt att säkerställa att förutsägelser är opartiska. Ett annat exempel på dessa tillämpningar är bankverksammhet där statistiska modeller ibland används för att uppskatta kreditvärdighet [25].

I tillämpningar såsom självkörande bilar eller medicinsk utrustning, där säkerheten är vital, är det viktigt att överväga hur pålitliga och säkra de neurala nätverken är samt att de inte utgör en risk för människor.

Neurala nätverk används för att hämta information från data. Om den data som ska analyseras är känslig, exempelvis finansiell eller hälsomässig information är det viktigt att den skyddas från obehöriga samt att GDPR [1] och andra förordningar följs.

Skulle en tillämpning av ett neuralt nätverk ändå få negativa konsekvenser är det viktigt att säkerställa vilka som bär ansvar för deras skapande och användning. Detta är viktigt för att kunna ge ersättning eller återupprättelse till någon som påverkats negativt.

Den största relevanta skillnaden mellan bayesianska neurala nätverk och traditionella neurala nätverk i detta sammanhang är att BNN låter oss få en bättre förståelse för osäkerheten i förutsägelserna. Detta låter oss potentiellt även få bättre insikt i hur nätverket fungerar och därav bättre bedöma dess lämplighet för olika tillämpningar.

5.6 Slutsats

Sammanfattningsvis lyckades vi implementera OFDM-algoritmen i PyTorch. Jämfört med regression med normalfördelade koefficienter har vi funnit att OFDM ger sämre förutsägelser, särskilt då modellen har tränats på mindre datamängder. Vi har observerat att modeller vilka använder OFDM går snabbare att anpassa jämfört med modeller baserade på normalfördelade mått, specifikt metoden med en linjärkombination av stokastiska variabler. Vi tror att OFDM underskattar den underliggande fördelningens varians, vilket skulle förklara varför den ofta har fler antal datapunkter utanför konfidensintervallet än förväntat, men att detta problem verkar minska i modeller där fler mått ingår. Detta är dock endast en hypotes och resultatet kan ej bekräfta denna hypotes.

Det återstår att se om den förutspådda variansen konvergerar mot det verkliga värdet eller kommer fortsätta öka vid ytterligare expansion av antal mått som ingår i modellen. Ytterligare efterforskningar krävs för att förstå mekanismerna bakom detta fenomen.

Problemet med tidskomplexiteten för beräkning av felet behöver även lösas innan OFDM-algoritmen praktiskt kan integreras i större neurala nätverk.

Överlag anser vi att fler tester behöver göras för att förstå hur OFDM anpassar modeller för olika typer av data och för olika val av parametrar i modellen, exempelvis definitionsintervallen för måtten, antal vikter per mått och initial steglängd. Med implementationen som gjorts i och med denna rapport finns goda möjligheter att genomföra ytterligare tester för att få en bättre insyn i metodens effektivitet och lämplighet i praktisk tillämpning.

Referenser

[1] "Allmän dataskyddsförordning", GDPR Info. (april 2016), URL: https://gdprinfo.eu/sv.

[2] M. Andersson och P. Olofsson, *Probability, Statistics and Stochastic Processes*, 2 utg. John Wiley & Sons, 2012, ISBN: 9780470889749.

[3] N. Andréasson, A. Evgrafov, M. Patriksson m. fl., *An Introduction to Continuous Optimization*, 3. utg. Lund, Sverige: Studentlitteratur, 2016, s. 374–375, ISBN: 9789144115290.

[4] "Artificiell intelligens: Möjligheter och risker: Nyheter: Europaparlamentet", EU-parlamentet. (maj 2022), URL: https://www.europarl.europa.eu/news/sv/headlines/society/20200918STO87404/artificiell-intelligens-mojligheter-och-risker.

[5] *Automatic differentiation package - torch.autograd*, 2022. URL: https://pytorch.org/docs/1.13/autograd.html.

[6] A. G. Baydin, B. A. Pearlmutter och A. A. Radul, "Automatic differentiation in machine learning: a survey", *CoRR*, årg. abs/1502.05767, 2015. DOI: 10.48550/arXiv.1502.05767.

[7] S. W. Bernard, *Density Estimation for Statistics and Data Analysis*. Bristol: J.W. Arrowsmith Ltd, 1986, ISBN: 9780412246203.

[8] S. Bharali och J. Hazarika, "Regression models with stochastic regressors: An expository note", *International Journal of Agricultural and Statistical Sciences*, årg. 15, nr 2, s. 873–880, 2019. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85077907409&partnerID=40&md5=1dd7db7534130bfda6b170156d83e876.

[9] E. Bingham, J. P. Chen, M. Jankowiak m. fl., "Pyro: Deep Universal Probabilistic Programming", *Journal of Machine Learning Research*, 2018. DOI: 10.48550/arXiv.1810.09538.

[10] C. M. Bishop, "Neural networks and their applications", *Review of Scientific Instruments*, årg. 65, nr 6, s. 1803–1832, juni 1994. DOI: 10.1063/1.1144830.

[11] Y.-C. Chen, "A tutorial on kernel density estimation and recent advances", *Biostatistics & Epidemiology*, årg. 1, s. 161–187, 2017. DOI: 10.1080/24709360.2017.1396742.

[12] C. Chu, J. Blanchet och P. Glynn, *Probability Functional Descent: A Unifying Perspective on GANs, Variational Inference, and Reinforcement Learning*, jan. 2019. DOI: 10.48550/arXiv.1901.10691.

[13] G. B. Folland, *Real Analysis: Modern Techniques and Their Applications*, 2 utg. John Wiley & Sons, 1999, s. 19–41, ISBN: 0471317160.

[14] T. L. Foundation, *Pytorch*, https://pytorch.org. (hämtad 2023-01-23).

[15] J. Gergonne, "The application of the method of least squares to the interpolation of sequences", *Historia Mathematica*, årg. 1, nr 4, s. 439–447, 1974. DOI: 10.1016/0315-0860(74)90034-2.

[16] T. Gneiting och A. E. Raftery, "Strictly Proper Scoring Rules, Prediction, and Estimation", *Journal of the American Statistical Association*, årg. 102, nr 477, s. 359–378, 2007. DOI: 10.1198/016214506000001437.

[17] B. Hanin och M. Sellke, *Approximating Continuous Functions by ReLU Nets of Minimal Width*, 2018. DOI: 10.48550/arXiv.1710.11278.

[18] K. Hornik, M. Stinchcombe och H. White, "Multilayer feedforward networks are universal approximators", *Neural Networks*, årg. 2, nr 5, s. 359–366, 1989. DOI: 10.1016/0893-6080(89)90020-8.

[19] D. S. Lemons, *An Introduction to Stochastic Processes in Physics*. The John Hopkins University Press, 2002, s. 34, ISBN: 0801868661.

[20] I. Molchanov och S. Zuyev, "Tangent Sets in the Space of Measures: With Applications to Variational Analysis", *Journal of Mathematical Analysis and Applications*, årg. 249, nr 2, s. 539–552, 2000. DOI: 10.1006/jmaa.2000.6906.

[21] I. Molchanov och S. Zuyev, "Steepest descent algorithms in a space of measures", *Statistics and computing*, årg. 12, nr 2, s. 115–123, 2002. DOI: 10.1023/A:1014878317736.

[22] "Parametric and non-parametric distributions", EPIX analytics. (), URL: https://modelassist.epixanalytics.com/display/EA/Parametric+and+non-parametric+distributions.

[23] K. Pearson och F.R.S., "X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, årg. 50, nr 302, s. 157–175, 1900. DOI: 10.1080/14786440009463897.

[24] S. J. Sheather och M. C. Jones, "A Reliable Data-Based Bandwidth Selection Method for Kernel Density Estimation", *Journal of the Royal Statistical Society. Series B (Methodological)*, årg. 53, nr 3, s. 683–690, 1991. URL: http://www.jstor.org/stable/2345597 (hämtad 2023-04-11).

[25] "Spetskompetens inom och förtroende för AI", EU-kommissionen. (2021), URL: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age/excellence-and-trust-artificial-intelligence_sv.

[26] Z. Wang och M. Wang, "Chi-square Loss for Softmax: an Echo of Neural Network Structure", *CoRR*, årg. abs/2108.13822, 2021. arXiv: 2108.13822. URL: https://arxiv.org/abs/2108.13822.

[27] E. Weisstein, *Weak Law of Large Numbers*, MathWorld A Wolfram Web Resource. URL: https://mathworld.wolfram.com/WeakLawofLargeNumbers.html.

[28] Y. Wong, *Why You Should Use Bayesian Neural Network*, okt. 2021. URL: https://towardsdatascience.com/why-you-should-use-bayesian-neural-network-aaf76732c150.

A Appendix 1 - Resultattabeller

A.1 Resultat av tester

I tabellerna nedan återfinns resultaten från jämförelsen beskriven i avsnitt 4.1. Tabellerna anger prestanda och 95% konfidensintervall för dessa värden, samt konvergenstid i sekunder och epoker, för samtliga metoder och regressionsmodeller. Konvergenstiderna är medelvärden för 50 tester.

Optimeringsalgoritm över finita diskreta mått (OFDM)

Antal datapunkter	Modell	Prestanda och 95% CI	Konvergenstid (s)	Konvergenstid (Epoker)
100	α_i	0.00 [0.00 0.06]	0.487	539.36
	$\alpha_i \cdot x$	0.52 [0.38 0.66]	3.195	2543.62
	$\alpha_i + \beta_i \cdot x$	0.48 [0.34 0.62]	2.219	1546.98
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	0.84 [0.74 0.94]	2.945	1205.00
500	α_i	0.00 [0.00 0.06]	3.251	3394.98
	$\alpha_i \cdot x$	0.58 [0.44 0.72]	4.908	3837.14
	$\alpha_i + \beta_i \cdot x$	0.66 [0.53 0.79]	5.624	3572.5
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	0.86 [0.76 0.96]	13.030	3467.70
1000	α_i	0.00 [0.00 0.06]	3.731	3817.72
	$\alpha_i \cdot x$	0.70 [0.57 0.83]	4.863	3645.60
	$\alpha_i + \beta_i \cdot x$	0.90 [0.82 0.98]	6.927	3961.00
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	0.92 [0.84 1.00]	18.652	3850.54

Polynomiell regression

Antal datapunkter	Modell	Prestanda och 95% CI	Konvergenstid (s)	Konvergenstid (Epoker)
100	α_i	1.00 [0.94 1.00]	4.358	5483.66
	$\alpha_i \cdot x$	0.98 [0.94 1.00]	8.323	16030.50
	$\alpha_i + \beta_i \cdot x$	1.00 [0.94 1.00]	9.500	11333.84
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	0.98 [0.94 1.00]	12.487	14667.54
500	α_i	0.98 [0.94 1.00]	5.530	6515.72
	$\alpha_i \cdot x$	0.88 [0.79 0.97]	8.095	10815.06
	$\alpha_i + \beta_i \cdot x$	0.92 [0.84 1.00]	10.834	12415.40
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	0.84 [0.74 0.94]	14.421	16834.62
1000	α_i	1.00 [0.94 1.00]	6.260	6947.02
	$\alpha_i \cdot x$	0.72 [0.60 0.84]	9.407	29693.18
	$\alpha_i + \beta_i \cdot x$	0.74 [0.62 0.86]	10.559	12516.06
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	0.70 [0.57 0.83]	16.696	20543.82

Linjärkombination av normalfördelade stokastiska variabler

Antal datapunkter	Modell	Prestanda och 95% CI	Konvergenstid (s)	Konvergenstid (Epoker)
100	α_i	1.00 [0.94 1.00]	4.376	308.14
	$\alpha_i \cdot x$	1.00 [0.94 1.00]	4.443	303.36
	$\alpha_i + \beta_i \cdot x$	1.00 [0.94 1.00]	4.317	300.12
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	1.00 [0.94 1.00]	4.765	336.6
500	α_i	1.00 [0.94 1.00]	25.156	311.86
	$\alpha_i \cdot x$	1.00 [0.94 1.00]	25.487	308.20
	$\alpha_i + \beta_i \cdot x$	1.00 [0.94 1.00]	41.128	521.64
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	1.00 [0.94 1.00]	27.529	344.16
1000	α_i	1.00 [0.94 1.00]	53.383	311.78
	$\alpha_i \cdot x$	1.00 [0.94 1.00]	54.570	313.70
	$\alpha_i + \beta_i \cdot x$	0.98 [0.94 1.00]	51.743	302.84
	$\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$	1.00 [0.94 1.00]	54.917	323.52

B Appendix 2 – Teori

B.1 Degenererad summa av väntevärden

Denna härledningen utarbetades tillsammans med handledaren. Antag att vi försöker hitta en fördelning θ så att $f(x, \theta)$ är nära y, alltså $|f(x, \theta) - y|$ är liten, där μ är den sanna fördelningen. Vi försöker alltså lösa

$$\min \quad \sum_i \mathrm{E}(f(x_i, \theta_i) - y_i)^2. \tag{47}$$

För ett generellt problem kan vi skriva $y_i = f(x_i, \theta_i)$ där $\theta_i \sim \mu$. Problemet kan formuleras om till

$$\min \quad \sum_i \mathrm{E}_\theta(f(x_i, \theta) - y_i)^2, \tag{48}$$

$$\min \quad \sum_i \int (f(x_i, \theta) - y_i)^2 \mathrm{d}\mu, \tag{49}$$

$$\min \quad < \underbrace{\sum_i (f(x_i, \cdot) - y_i)^2}_{g(\theta)}, \mu >, \tag{50}$$

där (50) är en linjär funktion med avseende på μ och $g(\theta)$ är gradienten. Vi får alltså nödvändiga och tillräckliga villkor för minimeringen som

$$\sum_i (f(x_i, \theta) - y_i)^2 = \begin{cases} = C, \mu, \\ \geq C, \forall \theta. \end{cases} \tag{51}$$

Lösningen till problemet är att distribuera μ över

$$\theta^* = \arg\min_\theta \sum_i (f(x_i, \theta) - y_i)^2. \tag{52}$$

För regression får vi

$$\min \quad \sum_i (\alpha + \beta x_i - y_i)^2, \tag{53}$$

och μ är en degenererad lösningen till regressionsproblemet i punkten (a, b).

B.2 Linjärkombination av stokastiska variabler

Följande teori är opublicerad men tillskrivs handledare Sergei Zuyev.

Låt $\eta(x)$ vara en funktion definierad enligt

$$\eta(x) = \sum_{k=1}^{K} \alpha^{(k)} h_k(x), \tag{54}$$

där $\alpha^{(k)} \sim \mu_k$ är oberoende stokastiska variabler och $h_k(x)$ är funktioner av x. Om vi antar att $\mu_k(dx) = f_k(x)dx$, där $f_k(x)$ är täthetsfunktioner och att h_k är kontinuerliga $\forall k$, då har $\eta(x)$ kumulativa täthetsfunktion

$$F_{\eta(x)}(y) = P(\eta(x) \leq y). \tag{55}$$

Med omskrivning blir den kumulativa täthetsfunktionen

$$F_{\eta(x)}(y) = P\left(\alpha^{(1)}h_1(x) \leq y - \sum_{k=2}^{K}\alpha^{(k)}h_k(x)\right), \tag{56}$$

$$F_{\eta(x)}(y) = E\left[F_1\left(\frac{y - \sum_{k=2}^{K}\alpha^{(k)}h_k(x)}{h_1(x)}\right)\middle|\alpha^{(2)}, \ldots, \alpha^{(k)}\right], \tag{57}$$

$$F_{\eta(x)}(y) = \int_{\mathbb{R}^{K-1}}\cdots\int F_1\left(\frac{y - \sum_{k=2}^{K}\alpha^{(k)}h_k(x)}{h_1(x)}\right) f_2(\alpha^{(2)})\ldots f_K(\alpha^{(K)})\mathrm{d}\alpha^{(2)}\ldots\mathrm{d}\alpha^{(K)}. \tag{58}$$

Täthetsfunktionen kan beskrivas som derivatan av den kumulativa täthetsfunktionen

$$f_{\eta(x)}(y) = \frac{\mathrm{d}}{\mathrm{d}y}F_{\eta(x)}(y), \tag{59}$$

$$\Rightarrow f_{\eta(x)}(y) = \int_{\mathbb{R}^{K-1}}\cdots\int f_1\left(\frac{y - \sum_{k=2}^{K}\alpha^{(k)}h_k(x)}{h_1(x)}\right) f_2(\alpha^{(2)})\ldots f_K(\alpha^{(K)})\mathrm{d}\alpha^{(2)}\ldots\mathrm{d}\alpha^{(K)}. \tag{60}$$

Sannolikheten för att observera en viss data $(x_1, y_1), \ldots, (x_N, y_N)$ givet en viss skattning $\bar{y}$ ges av

$$\mathrm{Lik}(\bar{y}) = \prod_{i=1}^{N} f_{\eta(x_i)}(y_i), \tag{61}$$

vilket beräknas enligt (60). För en bra skattning vill vi maximera sannolikheten att observera given data, alltså erhålls lösningen som

$$\max \mathrm{Lik}(\bar{y}) = \max \prod_{i=1}^{N} f_{\eta(x_i)}(y_i). \tag{62}$$

B.3 Yttre straffunktion

Metoden bygger på att bestraffa målfunktionen när ett kriterium ej uppfylls med hjälp av en straffunktion. Enligt [3] kan den önskade mängden av x formuleras som en kombination av likheter och olikheter enligt:

$$\begin{aligned} S := \{x \in \bar{\mathbb{R}}^n \,|\, g_i(x) \leq 0, \quad i = 1, ..., m \\ h_j(x) = 0, \quad j = 1, ...l\} \end{aligned} \tag{63}$$

Vilket sedan kan approximera den yttre straffunktionen [3]:

$$\chi s(x) \approx \nu\hat{\chi}(x) := \nu\left(\sum_{i=1}^{m}(\max\{0, g_i(x)\})^2 + \sum_{j=1}^{l}(h_j(x))^2\right) \tag{64}$$

Med denna metod kan begränsningen $0 \leq p \leq 1$ formuleras enligt följande:

$$\nu(\max(0, p-1)^2 + \max(0, -p)^2), \tag{65}$$

där ν är en skalfaktor för straffvärdet. Genom att addera straffunktionen till målfunktionen erhålls följande ekvation:

$$L = p(1-p) + \nu(\max(0, p-1)^2 + \max(0, -p)^2) \tag{66}$$

som kan styra optimeringen till att konvergera inom det önskade området eftersom max() termerna kommer vara nollskilda då $0 \leq p \leq 1$ ej uppfylls. Denna metod kan framgångsrikt implementeras för det enkla fallet med en parameter men problem uppstår då antalet parametrar ökar. Konvergensen är mycket känslig för straffaktorn ν och resultatet har en stor felmarginal. Detta fel skulle kunna minskas genom att dynamiskt justera steglängden då parametern närmar sig det korrekta värdet.

B.4 Lagrangemultiplikator-metoden

En metod som utvärderades i början av projektet var metoden med Lagrangemultiplikatorer som är vanlig inom matematisk optimering. Metoden konverterar ett begränsat problem till ett problem utan begränsningar, så att metoder för obegränsad optimering, så som gradient descent, kan användas. Metoden utgår från att först bilda den så kallade Lagrangefunktionen

$$\mathcal{L}(x, \lambda) = f(x) + \sum_k \lambda_k g_k(x), \tag{67}$$

där $f(x)$ är funktionen som vi vill minimera, $g_k(x)$ är en begränsning, λ_k är en så kallad Lagrangemultiplikator, $k = 1, ..., n$, där n är antalet begränsningar för problemet. Det kan visas att de stationära punkterna till Lagrangefunktionen kommer sammanfalla med de lokala extrempunkterna för $f(x)$.

Anledningen till att vi inte fortsatte arbetet med denna metoden var att den globala extrempunkten inte alltid är en stationärpunkt, exempelvis då extrempunkten ligger på randen av det begränsade området.

B.5 Icke-parametriska diskreta sannolikhetsmått med softmax

Softmax funktionen är definierad enligt

$$(\text{Softmax}(\mathbf{x}))_i = \frac{e^{x_i}}{\sum_j e^{x_j}},$$

där $\mathbf{x}$ är vektorn med element $x_1, x_2, \cdots$. Det är alltså en funktion som tar en vektor och ger tillbaka en vektor med lika många element, där varje element har transformerats enligt ovan. Detta skapar en vektor med element i intervallet $[0, 1]$ vilka har summan 1.

Ett exempel där detta kan appliceras är vid optimering av någonting som beror av en Bernoullifördelning. Bernoullifördelning är den fördelningen som ger värdet 1 med sannolikhet p och värdet 0 med sannolikhet $q = 1 - p$. Här måste båda världena vara mellan noll och ett, och summera till ett. Parameterrummet kan med andra ord betraktas som ett diskret sannolikhetsmått med två möjliga utfall. Om vi vill använda oss av vanlig sjunkgradient kan vi låta våra parametrar vara obegränsade reella tal och sedan applicera softmax på dem då vi vill finna p och q.

Det problem som uppstår är att det tar extremt många iterationer för optimerings-algoritmen att komma nära fallet där till exempel q är noll. Ett exempel där vi stöter på detta problem är minimeringen av Bernoullifördelningens varians; denna varians uttrycks algebraiskt pq, och kräver då att p eller q blir noll för att minimeras. För att uppnå detta behöver skillnaden mellan parametrarna (före applicering av softmax) vara oändlig. Därav kan det ta väldigt många itterationer för att uppnå en bra approximation av de optimala parametervärdena.

I detta fall kan vi få en bättre prestanda genom att sätta en högre steglängd (eng. Learning rate), men detta kan vara problematiskt vid mer komplicerade problem då målfunktionen kräver mer än att tvinga en parameter till noll för att finna minimum.

B.6 Poängfunktioner och poängregler

I följande underavsnitt är källan till teorin [16] *om inget annat anges.*

Vi har ett utfallsrum Ω, en σ-algebra $\mathcal{A}$ av delmängder på Ω och $\mathcal{P}$ en konvex klass av mått på $(\Omega, \mathcal{P})$. En funktion S är en poängregel (eng. scoring rule) om $S : \mathcal{P} \times \Omega \to \bar{\mathbb{R}}$, där $\bar{\mathbb{R}} = [-\infty, \infty]$ är den utökade realla tallinjen. Om ω är ett utfall i Ω ger $S(P, \omega)$ belöningen för vår förutsägelse P när ω

materialiserar. Den förväntade belöningen av förutsägelsen P när Q är den sanna fördelningen skriver vi som

$$S(P,Q) := \mathrm{E}_Q[S(P,\cdot)] = \int S(P,\omega)Q(\mathrm{d}\omega). \tag{68}$$

En poängregel är en funktion som sätter ett mått på hur väl en sannolikhetsfördelning förutsäger utfall. Exempelvis studeras de i syfte av att ge en rättvis evaluering av förutsägelser inom meteorologi.

Definition B.1. En poängregel sägs vara *äkta* (eng. proper) om

$$S(Q,Q) \geq S(P,Q), \quad \forall P,Q \in \mathcal{P}. \tag{69}$$

Ytterligare sägs en poängregel vara *strikt äkta* om och endast om likhet i (69) uppfylls då $P = Q$.

En äkta poängregel ger en ärlig belöning. Oavsett vad den sanna fördelningen Q är finns det ingen annan fördelning P som är bättre, och för en strikt äkta poängregel är det endast den sanna fördelningen som maximerar S.

Definition B.2. En poängregel $S : \mathcal{P} \times \Omega \to \bar{\mathbb{R}}$ sägs vara *reguljär* (eng. regular) med avseende på $\mathcal{P}$ om $S(P,Q)$ är reell värd för alla $P,Q \in \mathcal{P}$ förutom möjligen att $S(P,Q) = -\infty$ om $P \neq Q$.

Definition B.3. Om en poängregel S är äkta och reguljär kallar vi

$$d(P,Q) = S(Q,Q) - S(P,Q), \quad P,Q \in \mathcal{P}. \tag{70}$$

den associerade *divergensfunktionen* (eng. divergence function).

Divergensfunktionen är icke-negativ och då poängregeln är strikt äkta är divergensfunktionen strikt positiv förutom då $P = Q$. Vid optimering används avstånd för att mäta hur nära målet skattningen är den verkliga datan. Divergensfunktionen delar likheter med avstånd vilket motiverar teorin för poängregler. Då en divergensfunktion associerar med en äkta poängregel styrker det användningen denna som poängregel.

B.6.1 Exempel på poängregler

Ett exempel är Brier score. Om $\boldsymbol{p} = \{p_1, \dots p_n\}$ är en sannolikhetsfördelning är poängregeln

$$S(\boldsymbol{p}, i) = -\sum_{j=1}^{n}(\delta_{ij} - p_j)^2 = 2p_i - \sum_{j=1}^{n} p_j^2 - 1, \quad i \in 1 \dots n, \tag{71}$$

där $\delta_{ij} = 1$ då $i = j$ och $\delta_{ij} = 0$ då $i \neq j$. Divergensfunktionen för Brier score är

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sum_{j=1}^{n}(p_j - q_j)^2. \tag{72}$$

Ett annat exempel är logaritmisk poäng regeln. Poängregeln blir

$$S(\boldsymbol{p}, i) = \log p_i, \tag{73}$$

och divergensfunktionen blir

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sum_{j=1}^{n} q_j \log(q_j/p_j). \tag{74}$$

Notera att om förutsägelsen påstår att $p_i = 0$ men utfall i ändå uppstår blir $S(\boldsymbol{p}, i) = -\infty$.

C Appendix 3 – Källkod

För att se mer exakt hur filerna är uppbyggda och vilka bibliotek som importeras i de olika koderna se vårt GitHub-projekt: https://github.com/Miralleyan/Kandidatarbete

C.1 Måttklassen

```
class Measure:
    def __init__(self, locations: torch.tensor, weights: torch.tensor, device='cpu', optim_locations = False):
        self.locations = locations
        self.weights = torch.nn.parameter.Parameter(weights)
        self.device = device

    def __str__(self) -> str:
        """
        Returns the locations and weights of the measure as a string.
        :returns: str
        """
        out = "\033[4mLocations:\033[0m".ljust(28) + "\033[4mWeights:\033[0m \n"
        for i in range(len(self.weights)):
            out += f'{self.locations[i].item():<20.9f}{self.weights[i].item():<.9f}\n'
        return out

    def __repr__(self) -> str:
        """
        Returns the locations and weights of the measure as a string.
        :returns: str
        """
        return self.__str__()

    def is_probability(self, tol=1e-6):
        """
        Returns True if the measure is a probability measure.
        :param tol: Tolerance for summing to 1
        """
        if torch.any(self.weights < 0):
            return False
        if torch.abs(self.weights.sum() - 1) > tol:
            return False
        return True

    def total_mass(self) -> float:
        """
        Returns the sum of all weights in the measure: \sum_{i=1}^n w_i
        :returns: float
        """
        return sum(self.weights).item()

    def total_variation(self) -> float:
        """
        Returns the sum of the absolute value of all weights in the measure: \sum_{i=1}^n |w_i|
        :returns: float
        """
        return sum(abs(self.weights)).item()

    def support(self, tol=5e-3):
        """
        :param tol: proportion of total variation that can be un-accounted for by the support.
        :returns: all index where the weights are non-zero
        """
        sorted_idx = torch.argsort(self.weights.abs())
        accum_weight = torch.cumsum(self.weights[sorted_idx].abs(), dim=0)
        cutoff = tol * self.total_variation()
        return sorted_idx[cutoff < accum_weight]
```

```
58
59      def positive_part(self):
60          """
61          Returns the positive part of the Lebesgue decomposition of the measure
62          """
63          return Measure(self.locations, torch.max(self.weights, torch.zeros(len(self.weights))))
64
65      def negative_part(self):
66          """
67          Returns the negative part of the Lebesgue decomposition of the measure
68          """
69          return Measure(self.locations, -torch.min(self.weights, torch.zeros(len(self.weights))))
70
71      def sample(self, size):
72          """
73          Returns a sample of indeces from the locations of the measure
74          given by the distribution of the measures weights
75          :param size: Number of elements to sample
76          :returns: sample of indeces for random numbers based on measure
77          """
78          if torch.any(self.weights < 0):
79              assert ValueError("You can't have negative weights in a probability measure!")
80
81          sample_idx = torch.multinomial(self.weights, size, replacement=True)
82          sample = self.locations[sample_idx] + torch.randn(size)*(self.locations[1]-self.locations[0])/2
83          return sample
84
85      def copy(self):
86          return Measure(self.locations, self.weights)
87
88      def zero_grad(self):
89          self.weights.grad = None
90
91      def visualize(self):
92          """
93          Visualization of the weights
94          """
95          plt.bar(self.locations.detach(), self.weights.detach(), width=0.1, label="Measure")
96          plt.axhline(y=0, c="grey", linewidth=0.5)
97          plt.draw()
```

C.2 Optimeringsklassen

```
class Optimizer:

    def __init__(self, measures, loss : str, lr : float = 0.1):
        # Create list of measures
        if type(measures) == Measure:
            self.measures = [measures]
        elif type(measures) != list:
            Exception('Error: measures has to be of type Measure or list')
        else:
            self.measures = measures
        # Create list of lr's
        if type(lr) == float or type(lr) == int:
            self.lr = [lr]*len(self.measures)
        elif type(lr) != list:
            Exception('Error: lr has to be of type float or list')
        else:
            self.lr = lr

        loss_dict = {'essr':self.essr, 'nll':self.nll, 'KDEnll':self.KDEnll, 'chi_squared':self.chi_squared}
        self.loss = loss_dict[loss]
        self.state = {'measure':self.measures, 'lr':self.lr}
        self.is_optim = False

    def put_mass(self, meas_index, mass, location_index):
        """
        In current form, this method puts mass at a specified location, s.t. the location still
        has mass less at most 1 and returns how much mass is left to distribute.
        :param mass: Mass left to take
        :param location_index: Index of location to take mass from
        :returns: mass left to add to measure after adding at specified location
        """
        with torch.no_grad():
            self.measures[meas_index].weights[location_index] += mass

    def take_mass(self, meas_index, mass, location_index) -> float:
        """
        In current form, this method takes mass from a specified location, s.t. the location still
        has non-negative mass and returns how much mass is left to take.
        :param mass: Mass left to take
        :param location_index: Index of location to take mass from
        :returns: mass left to remove from measure after removing from specified location
        """
        with torch.no_grad():
            if mass > self.measures[meas_index].weights[location_index].item():
                mass_removed = self.measures[meas_index].weights[location_index].item()
                self.measures[meas_index].weights[location_index] = 0.
            else:
                self.measures[meas_index].weights[location_index] -= mass
                mass_removed = mass
        return mass_removed

    def stop_criterion(self, tol_supp=1e-6, tol_const=1e-2, adaptive = False):
        """
        Checks if the difference between the maximum and minimum gradient is
        within a certain range.
        :param tol_supp: lower bound for wieghts considered
        :param tol_const: stop value, when the maximum difference of gradients
        is smaller than this value the minimization should seize
        """
        if adaptive:
            return min([measure.weights.grad[measure.support(tol_supp)].max()
                        - measure.weights.grad.min() < tol_const*(measure.weights.grad.max() -
                                    measure.weights.grad.min()) for measure in self.measures])
        else:
```

```
65            return min([measure.weights.grad[measure.support(tol_supp)].max()
66                        - measure.weights.grad.min() < tol_const for measure in self.measures])
67
68    def step(self, meas_index):
69        """
70        Steepest decent with fixed total mass
71        """
72
73        # Sort gradient
74        grad_sorted = torch.argsort(self.measures[meas_index].weights.grad)
75
76        # Distribute positive mass
77        mass_pos = self.lr[meas_index]
78        self.put_mass(meas_index, mass_pos, grad_sorted[0].item())
79
80        # Distribute negative mass
81        mass_neg = self.lr[meas_index]
82        for i in torch.flip(grad_sorted, dims=[0]):
83            mass_neg -= self.take_mass(meas_index, mass_neg, i.item())
84            if mass_neg <= 0.:
85                break
86
87    def update_lr(self, index, fraction=0.7):
88        """
89        Updates learning rate for the optimizer
90
91        :param fraction: multiply lr with this value
92        """
93        self.lr[index]*=fraction
94
95    def state_dict(self):
96        """
97        Updates the state dictionary for the optimizer
98        """
99        print("\n".join("\t{}: {}".format(k, v) for k, v in self.state.items()))
100       return self.state
101
102   def load_state_dict(self, state_dict):
103       """
104       Overloads the current state dictionary
105
106       :param state_dict: state dictionary to load
107       """
108       self.state = state_dict
109
110   def lr_decrease_criterion(self, loss_fn, measure, old_measure):
111       """
112       Checks if learning rate should be decreased
113
114       :param loss_fn: loss function
115       :param measure: measure to compare current measure to
116       """
117       return loss_fn(old_measure) < loss_fn(measure)
118
119   def minimize(self, data, model, h = 0, alpha = 0.001, max_epochs=2000, smallest_lr=1e-6, verbose=False,
120                tol_supp=1e-6, tol_const=1e-2,  print_freq=100, adaptive=False,test=False):
121       """
122       :param data: list of tensors of data points. If x and y, then data should be on the form
123                    [x_tensor,y_tensor]. If only one series of data points, just this tensor is needed
124       :param model: model written as a function.
125       Should take a data input (x) and a set of parameters (params).
126       :param max_epochs: Max number of iterations
127       :param smallest_lr: Minimizer wil stop when lr is below this value
128       :param verbose: Print information about each epoch
129       :param print_freq: How frequently the minimizer should print information
```

```
        :param tol_supp:
        :param tol_const:
        :param adaptive:
        :return: Optimized measures
        """
        lr = self.lr
        loss_fn = self.loss

        if type(data) == torch.tensor:
            data = [data, data]

        perms, prep = self.prep_step(data, model, h, alpha)

        if test:
            tid=[]
            LossNotChanged=0
            t1=time.time()

        for epoch in range(max_epochs):
            # Backup current measures and reset grad
            old_measures = copy.deepcopy(self.measures)
            for m in self.measures:
                m.zero_grad()


            # Compute loss and grad
            loss_old = loss_fn(perms, *prep)
            loss_old.backward()

            # Stop criterion
            if self.stop_criterion(tol_supp, tol_const, adaptive):
                print(f'\nOptimum is attained. Loss: {loss_old}. Epochs: {epoch} epochs.')
                self.is_optim = True
                if test:
                    t2=time.time()
                    return self.measures,t2-t1,epoch
                else:
                    return self.measures

            if min(lr) < smallest_lr:
                print(f'The step size is too small: {lr}')
                if test:
                    t2=time.time()
                    if LossNotChanged<5:
                        return self.measures,t2-t1,epoch
                    else:
                        return self.measures, tid[0][0],tid[0][1]
                else:
                    return self.measures

            # Step
            maxima = []
            for meas_index in range(len(self.measures)):
                sup_index = self.measures[meas_index].support()
                grads = copy.deepcopy(self.measures[meas_index].weights.grad)
                #print(grads)
                maxima.append(torch.max(grads[sup_index]))
            max_index = maxima.index(sorted(maxima)[-1])
            self.step(max_index)

            loss_new = loss_fn(perms, *prep)
            loss_new.backward()

            # bad step
            if loss_old < loss_new:
```

```
                # Revert to the backup measure and decrease lr
                self.measures = copy.deepcopy(old_measures)
                self.update_lr(max_index, fraction=0.1)

                if verbose:
                    print(f'Epoch: {epoch:<10} Lr was reduced to: {lr}')
            elif loss_old == loss_new and verbose:
                if test and LossNotChanged<5:
                    LossNotChanged+=1
                    t2=time.time()
                    tid.append([t2-t1,epoch])
                print(f'Epoch: {epoch:<10} Loss did not change ({loss_new})')

            # successful step
            else:
                if test and LossNotChanged<5:
                    LossNotChanged=0
                    tid=[]
                if epoch % print_freq == 0:
                    if verbose:
                        print(f'Epoch: {epoch:<10} Loss: {loss_new:<10.9f} LR: {lr}')
                    else:
                        print('.')


        print('Max epochs reached')
        if test:
            t2=time.time()
            if LossNotChanged<5:
                return self.measures,t2-t1,epoch
            else:
                return self.measures, tid[0][0],tid[0][1]
        else:
            return self.measures

    def visualize(self):
        """
        Visualizes the measures of the optimizer. Currently requires that
        a gradient has been stored in the weights of the measures.
        """
        cols = int(torch.ceil(torch.sqrt(torch.tensor(len(self.measures)))).item())
        rows = int(torch.ceil(torch.tensor(len(self.measures)/cols)).item())
        fig, axs = plt.subplots(rows,cols)
        axs = axs.flatten()
        fig.suptitle('Optimizer Visualization')
        grads = [measure.weights.grad for measure in self.measures]
        for i, measure in enumerate(self.measures):
            M, m = grads[i].max(), grads[i].min()
            wm = measure.weights.max()
            scaled_weights = measure.weights * (M - m) / wm * 0.25 + m
            support = measure.support()
            with torch.no_grad():
                # Support locations
                axs[i%cols+(i//cols)*cols].plot(measure.locations[support], torch.zeros(len(measure.weights))[support]
                                + m, '.', c='red', label=' Measure Support')
                # Gradient
                axs[i%cols+(i//cols)*cols].plot(measure.locations, grads[i], c='green', label=' Gradient')
                # Measure weights where there is support
                axs[i%cols+(i//cols)*cols].vlines(measure.locations[support], torch.zeros(len(measure.weights))[support]
                            + 1.3 * m, scaled_weights[support], colors='blue', label=' Measure Weights')
                axs[i%cols+(i//cols)*cols].axhline(y=m, c="orange", linewidth=0.5)
                axs[i%cols+(i//cols)*cols].legend(loc='upper right')
                axs[i%cols+(i//cols)*cols].set_ylim([m, M])

    # Loss functions
```

```
    def essr(self, perms, errors):
        """
        Calculates the expected sum of square residuals loss function

        :param perms: list of the possible permutations of one location from each measure
        :param errors: tensor of the sum of errors, compared with true data, for each permutation of locations
        """
        probs = torch.cat([self.measures[i].weights[perms[:, i]].unsqueeze(1) for i in range(len(self.measures))],
                          1).prod(1)
        return errors.dot(probs)

    def nll(self, perms, loc_index):
        """
        Calculates the negative log-likelihood loss function

        :param perms: list of the possible permutations of one location from each measure
        :param loc_index: list of location permutation closest with the least absolute error for each data point
        """
        probs = torch.cat([self.measures[i].weights[perms[:, i]].unsqueeze(1) for i in range(len(self.measures))],
                          1).prod(1)
        return -sum(torch.log(probs[loc_index]))

    def KDEnll(self, perms, kde_mat, h):
        """
        Calculates the negative log-likelihood loss function, with a KDE

        :param perms: list of the possible permutations of one location from each measure
        :param kde_mat: matrix of kernels for each location permutation
        :param h: bandwidth for the KDE
        """
        probs = torch.cat([self.measures[i].weights[perms[:, i]].unsqueeze(1) for i in range(len(self.measures))],
                          1).prod(1)
        return -(torch.matmul(kde_mat, probs) / h).log().sum()

    def chi_squared(self, perms, bins_freq):
        """
        Calculates the chi-squared loss function

        :param perms: list of the possible permutations of one location from each measure
        :param bins_freq: frequencies of data points closest to each location permutation
        """
        probs = torch.cat([self.measures[i].weights[perms[:, i]].unsqueeze(1) for i in range(len(self.measures))],
                          1).prod(1)
        return (probs**2/bins_freq).sum()

    # Preparational step for loss functions
    def prep_step(self, data, model, h = 0, alpha = 0.001):
        """
        Does a preparatory step and returns the prepared data needed for the optimizers loss function

        :param data: data to fit the model to
        :param model: model that should be fit to data, should be a function taking the data (x) and
                      locations as parameters
        :param h: bandwidth parameter for KDE
        :param aplha: label smoothing parameter for chi-squared loss function
        """
        # permutations of all measures
        perms = torch.tensor([item for item in itertools.product(*[range(measure.weights.size(dim=0))
                                for measure in self.measures])])
        locs = torch.cat([self.measures[i].locations[perms[:, i]].unsqueeze(1) for i in range(len(self.measures))], 1)
        prep = []
        if self.loss == self.essr:
            prep.append(torch.tensor([(model(data[0], locs[i]) - data[1]).pow(2).sum() for i in range(len(perms))]))
        elif self.loss == self.nll:
            loc_idx = []
```

```
            for i in range(len(data[0])):
                ab = torch.abs(model(data[0][i], [locs[:,i] for i in range(locs.size(dim=1))]) - data[1][i])
                loc_idx.append(torch.argmin(torch.tensor(ab)))
            prep.append(torch.tensor(loc_idx))
        elif self.loss == self.KDEnll:
            if h == 0:
                sigma=torch.std(data[0])
                A=min(sigma,(torch.quantile(data[0],0.75)-torch.quantile(data[0],0.25))/1.34)
                h=0.9*A*len(data[0])**(-1/5)
            kde_mat = 1/np.sqrt(2*np.pi)*np.exp(-((data[1].view(-1,1) - model(data[0].view(-1,1), locs.transpose(0,1)))
                                                  / h)**2/2)
            prep.append(kde_mat)
            prep.append(h)
        elif self.loss == self.chi_squared:
            loc_idx = []
            for i in range(len(data[0])):
                ab = torch.abs(model(data[0][i], [locs[:,i] for i in range(locs.size(dim=1))]) - data[1][i])
                loc_idx.append(torch.argmin(ab))
            bins = torch.tensor(loc_idx)
            bins_freq = torch.bincount(bins, minlength=np.cumprod([self.measures[i].weights.size(dim=0)
                                       for i in range(len(self.measures))])[-1])/len(data[0])**2
            bins_freq = bins_freq*(1-alpha)+alpha / len(bins_freq)
            prep.append(bins_freq)
        return perms, prep
```

C.3 Checkklassen

```
class Check():
    def __init__(self, opt: Optimizer, model,
                 x: torch.tensor,y: torch.tensor, alpha=0.05,
                 normal=False, Return=False,sample_size=2000):
        """
        A class that will check how close a fitted measure is to the orginal by creating a confidence
        intervall at each x-value and checking to see if the corresponding y-value is insiede the
        confidence interval.

        :param opt: An instance of the class Optimizer
        :param model: A function with input: a list of x-values and a list of lists containing weights from measures
        :param x: The x-values used in the model
        :param y: The y-values from the original distribution that we use to fitt the measure
        :param alpha: The amount of confidence we want for our donfidence intervals, standard is 0.05
                      which corresponds to a 95% CI
        :param normal: Set to true if you know that the distribution is normal
        :param Return: Set to true if you wish the class to return the amount of misses and the bounds for the 95% CI
        """
        self.opt=opt
        self.model=model
        self.data=[x,y]
        self.N=len(x)
        self.alpha=alpha
        self.normal=normal
        self.Return=Return
        self.sample_size=sample_size


    def check(self):
        '''
        Calculates the amount of the original data (y) that is outside
        the boundaries of a 95% confidence intervall (if no value is given to
        the variable alpha) and then calculates the probability of
        that amount of misses

        '''
        bounds=[]
        for x in self.data[0]:
            input=[]
            for meas in self.opt.measures:
                input.append(meas.sample(self.sample_size))
            bounds.append(self.CI(self.model(x,input)))
        miss=self.misses(self.data[1],bounds)


        # b_lower = [b[0].item() for b in bounds]
        # b_upper = [b[1].item() for b in bounds]
        # plt.scatter(self.data[0], self.data[1], sizes=[20]*len(self.data[0]))
        # plt.plot(self.data[0], b_lower, 'r--')
        # plt.plot(self.data[0], b_upper, 'r--')
        # plt.show()

        lci=scipy.stats.binom.ppf(self.alpha/2,self.N,self.alpha)
        hci=scipy.stats.binom.ppf(1-self.alpha/2,self.N,self.alpha)
        if (miss >= lci) and (miss <= hci):
            print(f'{miss} is inside the confidence interval ({lci}, {hci}):')
            print(f'No contradiction with the fitted model at {100*(1-self.alpha)}% confidence level')
        else:
            print(f'{miss} is outside the confidence interval ({lci}, {hci}):')
            print(f'Number of misses is significantly at {100*(1-self.alpha)}% confidence level different
                    from expected for the fitted model!')
        if self.Return==True:
            return lci,hci, miss

```

```
    def CI(self, data:list[float]):
        '''
        Calculates the bounds of an approximate 95% confidence intervall
        for the given data in output
        :param data: List of values that the confidence interval is calculated from
        '''
        if self.normal:
            mean=torch.mean(data)
            std=torch.std(data)
            q=scipy.stats.norm.ppf(self.alpha/2)
            cil=mean+q*std
            cih=mean-q*std
            bounds=[cil,cih]
        else:
            edge=int(self.alpha/2*self.sample_size)
            idx_sorted_cropped=torch.argsort(data)[edge:self.sample_size-edge]
            bounds=data[idx_sorted_cropped[[0,-1]]]
        return bounds


    def misses(self,y:list[float],bounds:list[list[float,float]]):
        '''
        Calculates the amount of values in y that are not within the
        corresponding boundary in bounds

        :param y: The data which we will se if it is in the corresponding confidence interval
        :param bounds: list containing the pairs of bounds for each x
        '''
        miss=0
        for i in range(len(y)):
            if y[i]>bounds[i][1] or y[i]<bounds[i][0]:
                miss+=1
        return miss
```

C.4 Generering av data

Kod för att generera data som användes för testerna beskrivna i avsnitt 4.1, vars resultat återfinns i tabellerna i avsnitt A.1.

```
import numpy as np
np.random.seed(0)
tests = 50

filename = 'data.npy'
means = np.array([])
stds = np.array([])

for i in range(tests):
    mean = np.random.normal(0,1,3)
    std = np.abs(np.random.normal(1,0.1,3))
    means = np.append(means, mean)
    stds = np.append(stds, std)
    for N in [100, 500, 1000]:
        a = np.random.normal(mean[0], std[0], N)
        b = np.random.normal(mean[1], std[1], N)
        c = np.random.normal(mean[2], std[2], N)

        x = np.linspace(-5, 5, N)
        y = a
        y_lin = a + x * b
        y_sqr = a + x * b + x**2 * c
        y_nonNorm = np.concatenate((a[0:N//2], b[0:N//2]))
        y_ax = a * x

        data = {
            'y': y,
            'y_lin': y_lin,
            'y_sqr': y_sqr,
            'y_nonNorm': y_nonNorm,
            'y_ax': y_ax
        }

        for k, v in data.items():
            file, ext = filename.split('.', 1)
            name = file + '_' + str(N) + '_' + k + '_' + str(i) + '.' + ext

            with open(name, 'wb') as f:
                np.save(f, np.array(v))

params = np.array([means,stds])
# Saves means and stds in npy file in format: [[means],[stds]]
np.save('params.npy', params)
```

C.5 Tester för OFDM

C.5.1 Fallet α_i

```

import torch
import pytorch_measure as pm
import numpy as np
from tqdm import tqdm
import json
import math

# Regression model
def regression_model(x,list):
    return list[0]

#Load data of parameters from the file where you ran data_generator.py
param=np.load(f'../Finalized/test_data/params.npy')

for length in [100,500,1000]:
    success=[]
    tid=[]
    epoch=[]
    measures=[]
    for i in tqdm(range(50)):
        data=np.load(f'../Finalized/test_data/data_{length}_y_{i}.npy')
        y=torch.from_numpy(data)
        x = torch.linspace(-5, 5, length)

        M=length #Amount of datapoints

        s=2 #Decides width of the measures
        aU=math.ceil(param[0][3*i]+s*param[1][3*i])
        aL=math.floor(param[0][3*i]-s*param[1][3*i])
        N=2*(aU-aL)+1

        #Creates the measure
        measure = pm.Measure(torch.linspace(aL, aU, N), torch.ones(N).double() / N)

        # Instance of optimizer
        opt = pm.Optimizer([measure],"KDEnll" ,lr=0.1)

        # Call to minimizer
        new_mes,time,iteration=opt.minimize([x,y], regression_model,verbose=False,adaptive=False,
                                            max_epochs=4000,test=True)

        # Visualize measures and gradient
        new_mes[0].visualize()
        #plt.show()


        #Run tests and save the results
        check=pm.Check(opt,regression_model,x,y,normal=False,Return=True)
        l,u,miss=check.check()

        success.append(l<miss and miss<u)
        tid.append(time)
        epoch.append(iteration)
        measures.append([new_mes[0].locations.tolist(),new_mes[0].weights.tolist()])


    data=[measures,sum(tid)/len(tid),sum(epoch)/(len(epoch)),sum(success)/len(success)]
    with open(f"resultat/Sergey1_{M}.json", "w") as outfile:
        outfile.write(json.dumps(data))
```

C.5.2 Fallet $\alpha_i \cdot x$

```
import torch
import pytorch_measure as pm
import numpy as np
from tqdm import tqdm
import json
import math

# Linear regression model
def regression_model(x,list):
    return list[0]*x

#Load data of parameters from the file where you ran data_generator.py
param=np.load(f'../Finalized/test_data/params.npy')

for length in [100,500,1000]:
    success=[]
    tid=[]
    epoch=[]
    measures=[]
    for i in tqdm(range(50)):
        data=np.load(f'../Finalized/test_data/data_{length}_y_ax_{i}.npy')
        y=torch.from_numpy(data)
        x = torch.linspace(-5, 5, length)

        M=length #Amount of datapoints

        s=2 #Decides width of the measures
        aU=math.ceil(param[0][3*i]+s*param[1][3*i])
        aL=math.floor(param[0][3*i]-s*param[1][3*i])
        N=2*(aU-aL)+10

        #Creates the measure
        measure = pm.Measure(torch.linspace(aL, aU, N), torch.ones(N).double() / N)

        # Instance of optimizer
        opt = pm.Optimizer([measure],"KDEnll" ,lr=0.1)

        # Call to minimizer
        new_mes,time,iteration=opt.minimize([x,y], regression_model,verbose=False,adaptive=False,
                                            max_epochs=4000,test=True)

        # Visualize measures and gradient
        new_mes[0].visualize()
        #plt.show()

        #Run tests and save the results
        check=pm.Check(opt,regression_model,x,y,normal=False,Return=True)
        l,u,miss=check.check()

        success.append(l<miss and miss<u)
        tid.append(time)
        epoch.append(iteration)
        measures.append([new_mes[0].locations.tolist(),new_mes[0].weights.tolist()])

    data=[measures,sum(tid)/len(tid),sum(epoch)/(len(epoch)),sum(success)/len(success)]
    with open(f"resultat/Sergey1a_{M}.json", "w") as outfile:
        outfile.write(json.dumps(data))
```

C.5.3 Fallet $\alpha_i + \beta_i \cdot x$

```
import torch
import pytorch_measure as pm
import numpy as np
from tqdm import tqdm
import json
import math

# Linear regression model
def regression_model(x,list):
     return list[0]*x+list[1]

#Load data of parameters from the file where you ran data_generator.py
param=np.load(f'../Finalized/test_data/params.npy')

for length in [100,500,1000]:
     success=[]
     tid=[]
     epoch=[]
     measures=[]
     for i in tqdm(range(50)):
          data=np.load(f'../Finalized/test_data/data_{length}_y_lin_{i}.npy')
          y=torch.from_numpy(data)
          x = torch.linspace(-5, 5, length)

          M=length #Amount of datapoints

          s=2 #Decides width of the measures
          aU=math.ceil(param[0][3*i+1]+s*param[1][3*i+1])
          aL=math.floor(param[0][3*i+1]-s*param[1][3*i+1])
          bU=math.ceil(param[0][3*i]+s*param[1][3*i])
          bL=math.floor(param[0][3*i]-s*param[1][3*i])
          Nb=2*(bU-bL)+1
          Na=2*(aU-aL)+1

          #Creates the measures
          a = pm.Measure(torch.linspace(aL, aU, Na), torch.ones(Na).double() / Na)
          b = pm.Measure(torch.linspace(bL, bU, Nb), torch.ones(Nb).double() / Nb)
          measure= [a,b]

          # Instance of optimizer
          opt = pm.Optimizer(measure, "KDEnll", lr = 0.1)

          # Call to minimizer
          new_mes,time,iteration=opt.minimize([x,y],regression_model,max_epochs=4000,verbose = False,
                                              print_freq=100, smallest_lr=1e-10,test=True)
          # Visualize measures and gradient
          new_mes[0].visualize()
          #plt.show()
          new_mes[1].visualize()
          #plt.show()

          #Run tests and save the results
          check=pm.Check(opt,regression_model,x,y,normal=False,Return=True)
          l,u,miss=check.check()

          success.append(l<=miss and miss<=u)
          tid.append(time)
          epoch.append(iteration)
          for i in range(len(new_mes)):
               measures.append([new_mes[i].locations.tolist(),new_mes[i].weights.tolist()])

     data=[measures,sum(tid)/len(tid),sum(epoch)/len(epoch),sum(success)/len(success)]
     with open(f"resultat_samuel/Sergey2M_{M}.json", "w") as outfile:
          outfile.write(json.dumps(data))
```

C.5.4 Fallet $\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$

```
import torch
import pytorch_measure as pm
import numpy as np
from tqdm import tqdm
import json
import math

# Linear regression model
def regression_model(x,list):
     return list[0]*x**2+list[1]*x+list[2]

#Load data of parameters from the file where you ran data_generator.py
param=np.load('../Finalized/test_data/params.npy')

for length in [100,500,1000]:
    success=[]
    tid=[]
    epoch=[]
    measures=[]
    for i in tqdm(range(50)):
        data=np.load(f'../Finalized/test_data/data_{length}_y_sqr_{i}.npy')
        y=torch.from_numpy(data)
        x = torch.linspace(-5, 5, length)

        #plt.scatter(x,y)
        #plt.show()
        M=length #Amount of datapoints

        s=2 #Decides width of the measures
        aU=math.ceil(param[0][3*i+2]+s*param[1][3*i+2])
        aL=math.floor(param[0][3*i+2]-s*param[1][3*i+2])
        bU=math.ceil(param[0][3*i+1]+s*param[1][3*i+1])
        bL=math.floor(param[0][3*i+1]-s*param[1][3*i+1])
        cU=math.ceil(param[0][3*i]+s*param[1][3*i])
        cL=math.floor(param[0][3*i]-s*param[1][3*i])
        Nc=2*(cU-cL)+1
        Nb=2*(bU-bL)+1
        Na=2*(aU-aL)+1


        # Measure for slope (a) and intercept (b) of linear model
        a = pm.Measure(torch.linspace(aU, aL, Na), torch.ones(Na).double() / Na)
        b = pm.Measure(torch.linspace(bU, bL, Nb), torch.ones(Nb).double() / Nb)
        c= pm.Measure(torch.linspace(cU, cL, Nc), torch.ones(Nc).double() / Nc)
        measure = [a,b,c]

        # Instance of optimizer
        opt = pm.Optimizer(measure, "KDEnll", lr = 0.1)

        # Call to minimizer
        new_mes,time,iteration=opt.minimize([x,y],regression_model,max_epochs=4000,verbose = False,
                                            print_freq=100, smallest_lr=1e-10,test=True)

        # Visualize measures and gradient
        new_mes[0].visualize()
        #plt.show()
        new_mes[1].visualize()
        #plt.show()
        new_mes[2].visualize()
        #plt.show()

        #Run tests and save the results
        check=pm.Check(opt,regression_model,x,y,normal=False,Return=True)
        l,u,miss=check.check()
```

```
65
66          success.append(l<=miss and miss<=u)
67          tid.append(time)
68          epoch.append(iteration)
69          for i in range(len(new_mes)):
70              measures.append([new_mes[i].locations.tolist(),new_mes[i].weights.tolist()])
71
72      data=[measures,sum(tid)/len(tid),sum(epoch)/len(epoch),sum(success)/len(success)]
73      with open(f"resultat/Sergey3_{M}.json", "w") as outfile:
74          outfile.write(json.dumps(data))
```

C.6 Tester för linjärkombination av stokastiska variabler

C.6.1 Fallet α_i

```
import torch
import numpy as np
import scipy as sp
import linear_combination_optimizer as lco
import json

# Calculate confidence intervals and returns amount of misses
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        # Generate sample from model
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# For each data size
for length in [100, 500, 1000]:
    # Lists for storing results
    success=[]
    time=[]
    epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_{i}.npy')
        opt = lco.Optimizer(x, y, order=1)
        mu, sigma, conv_epoch, conv_time = opt.optimize(epochs = 3000, test = True)

        l, u, miss = misses(x,torch.tensor(y),mu,sigma)
        success.append(l<=miss and miss<=u)
        time.append(conv_time)
        epoch.append(conv_epoch)
        means.append(mu)
        std.append(sigma)

    # Save results in a json file
    results = [means, std, sum(time)/len(time), sum(epoch)/(len(epoch)),
                float(100*sum(success)/len(success))]
    with open(f"Lin_comb_results/lin_comb_results_{length}_y.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.6.2 Fallet $\alpha_i \cdot x$

```python
import torch
import numpy as np
import scipy as sp
import linear_combination_optimizer as lco
import json

# Confidence intervals
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        # Generate sample from model
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# For each data size
for length in [100, 500, 1000]:
    # Lists for storing results
    success=[]
    time=[]
    epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_ax_{i}.npy')
        opt = lco.Optimizer(x, y, order=1, ax = True)
        mu, sigma, conv_epoch, conv_time = opt.optimize(epochs = 3000, test = True)

        l, u, miss = misses(x,torch.tensor(y),mu,sigma)
        success.append(l<=miss and miss<=u)
        time.append(conv_time)
        epoch.append(conv_epoch)
        means.append(mu)
        std.append(sigma)

    # Save results in a json file
    results = [means, std, sum(time)/len(time), sum(epoch)/(len(epoch)),
                float(100*sum(success)/len(success))]
    with open(f"Lin_comb_results/lin_comb_results_{length}_y_ax.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.6.3 Fallet $\alpha_i + \beta_i \cdot x$

```python
import torch
import numpy as np
import scipy as sp
import linear_combination_optimizer as lco
import json

# Calculate confidence intervals and returns amount of misses
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# For each data size
for length in [100, 500, 1000]:
    # Lists for storing results
    success=[]
    time=[]
    epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_lin_{i}.npy')
        opt = lco.Optimizer(x, y, order=2)
        mu, sigma, conv_epoch, conv_time = opt.optimize(epochs = 3000, test = True)

        l, u, miss = misses(x,torch.tensor(y),mu,sigma)
        success.append(l<=miss and miss<=u)
        epoch.append(conv_epoch)
        means.append(mu)
        std.append(sigma)

    # Save results in a json file
    results = [means, std, sum(time)/len(time), sum(epoch)/(len(epoch)),
               float(100*sum(success)/len(success))]
    with open(f"Lin_comb_results/lin_comb_results_{length}_y_lin.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.6.4 Fallet $\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$

```python
import torch
import numpy as np
import scipy as sp
import linear_combination_optimizer as lco
import json

# Calculate confidence intervals and returns amount of misses
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        # Generate sample from model
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# For each data size
for length in [100, 500, 1000]:
    # Lists for storing results
    success=[]
    time=[]
    epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_sqr_{i}.npy')
        opt = lco.Optimizer(x, y, order=3)
        mu, sigma, conv_epoch, conv_time = opt.optimize(epochs = 3000, test = True)

        l, u, miss = misses(x,torch.tensor(y),mu,sigma)
        success.append(l<=miss and miss<=u)
        time.append(conv_time)
        epoch.append(conv_epoch)
        means.append(mu)
        std.append(sigma)

    # Save results in a json file
    results = [means, std, sum(time)/len(time), sum(epoch)/(len(epoch)),
                float(100*sum(success)/len(success))]
    with open(f"Lin_comb_results/lin_comb_results_{length}_y_sqr.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.7 Tester för polynomiskt neuralt nätverk

C.7.1 Fallet α_i

```
import torch
import numpy as np
import scipy as sp
import time
import json

# Polynomial nn method
def eval_powers_of_x(x, n):
    return x.pow(torch.arange(n))
def eval_even_powers_of_x(x, n):
    return x.pow(2 * torch.arange(n))

# Calculate confidence intervals and returns amount of misses
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        # Generate sample from model
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# Polynomial nn method
class NormalPolynomialModel(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.mean = None
        self.std = None

        self.N_mean = 1 # constant, linear, quadratic, ... terms
        self.N_var = 1

        self.poly_multipliers_mean = torch.nn.parameter.Parameter(torch.rand(self.N_mean))
        self.poly_multipliers_var = torch.nn.parameter.Parameter(torch.rand(self.N_var))
        self.var_shift = torch.nn.parameter.Parameter(torch.tensor(0.1))

        self.mean_layer = torch.nn.Linear(self.N_mean, 1)
        self.var_layer = torch.nn.Sequential(
            torch.nn.Linear(self.N_var, 1),
            torch.nn.ReLU()
        )

    def forward(self, x):
        self.mean = self.mean_layer(self.poly_multipliers_mean * eval_powers_of_x(x, self.N_mean))
        self.var = self.var_layer(self.poly_multipliers_var * eval_even_powers_of_x(x, self.N_var)) +
                    self.var_shift ** 2
        return self.mean, self.var

# Loss function for nn
def log_k_with_var(mean, var, y):
    return -0.5*torch.log(var) - (y - mean)**2 / (2 * var)

# For each data size
for length in [100, 500, 1000]:
    # Lists for storing results
```

```
    success=[]
    tid=[]
    end_epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_{i}.npy')
        y = torch.tensor(y)
        x_unsq = x.view(-1, 1)
        y = y.view(-1, 1)
        model = NormalPolynomialModel()
        opt = torch.optim.Adam(model.parameters())

        max_epoch = 300000
        conv_epoch = max_epoch
        conv_time = float('inf')
        old_loss = float('inf')
        t1 = time.time()
        # Main loop of regression
        for epoch in range(max_epoch):
            opt.zero_grad()
            mean, var = model(x_unsq)
            log_likelyhood = log_k_with_var(mean, var, y).sum()
            loss = -log_likelyhood
            loss.backward()
            opt.step()
            if torch.abs(old_loss-loss) < 1e-4:
                t2 = time.time()
                conv_time = t2-t1
                conv_epoch = epoch
                break
            old_loss = loss

            if epoch % 1000 == 0:
                print(f'{epoch}:: Loss = {loss.item()}')

        m, var = model(x_unsq)
        s = var.sqrt()
        m = m.detach().numpy()
        s = s.detach().numpy()

        l, u, miss = misses(x,torch.tensor(y),m,s)
        success.append(l<=miss and miss<=u)
        if conv_time != float('inf'):
            tid.append(conv_time)
        end_epoch.append(conv_epoch)
        means.append(m.tolist())
        std.append(s.tolist())

    # Save results in a json file
    results = [means, std, sum(tid)/len(tid), sum(end_epoch)/(len(end_epoch)),
                float(100*sum(success)/len(success))]
    with open(f"Poly_results/poly_results_{length}_y.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.7.2 Fallet $\alpha_i \cdot x$

```
import torch
import pytorch_measure as pm
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
from scipy import stats
import time
import linear_combination_optimizer as lco
import json

# Polynomial nn method
def eval_powers_of_x(x, n):
    return x
def eval_even_powers_of_x(x, n):
    return x.pow(2 * torch.arange(n))

# Calculate confidence intervals and returns amount of misses
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        # Generate sample from model
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# Polynomial nn method
class NormalPolynomialModel(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.mean = None
        self.std = None

        self.N_mean = 1 # constant, linear, quadratic, ... terms
        self.N_var = 1

        self.poly_multipliers_mean = torch.nn.parameter.Parameter(torch.rand(self.N_mean))
        self.poly_multipliers_var = torch.nn.parameter.Parameter(torch.rand(self.N_var))
        self.var_shift = torch.nn.parameter.Parameter(torch.tensor(0.1))

        self.mean_layer = torch.nn.Linear(self.N_mean, 1)
        self.var_layer = torch.nn.Sequential(
            torch.nn.Linear(self.N_var, 1),
            torch.nn.ReLU()
        )

    def forward(self, x):
        self.mean = self.mean_layer(self.poly_multipliers_mean * x)
        self.var = self.var_layer(self.poly_multipliers_var * x**2) + self.var_shift ** 2
        return self.mean, self.var

# Loss function for nn
def log_k_with_var(mean, var, y):
    return -0.5*torch.log(var) - (y - mean)**2 / (2 * var)

# For each data size
for length in [100, 500, 1000]:
```

```
    # Lists for storing results
    success=[]
    tid=[]
    end_epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_ax_{i}.npy')
        y = torch.tensor(y)
        x_unsq = x.view(-1, 1)
        y = y.view(-1, 1)
        model = NormalPolynomialModel()
        opt = torch.optim.Adam(model.parameters())

        max_epoch = 300000
        conv_epoch = max_epoch
        conv_time = float('inf')
        old_loss = float('inf')
        t1 = time.time()
        # Main loop of regression
        for epoch in range(max_epoch):
            opt.zero_grad()
            mean, var = model(x_unsq)
            log_likelyhood = log_k_with_var(mean, var, y).sum()
            loss = -log_likelyhood
            loss.backward()
            opt.step()
            if torch.abs(old_loss-loss) < 1e-4:
                t2 = time.time()
                conv_time = t2-t1
                conv_epoch = epoch
                break
            old_loss = loss

            if epoch % 1000 == 0:
                print(f'{epoch}:: Loss = {loss.item()}')

        m, var = model(x_unsq)
        s = var.sqrt()
        m = m.detach().numpy()
        s = s.detach().numpy()

        if sum([np.isnan(m[i]) for i in range(len(m))]) == 0 and
           sum([np.isnan(s[i]) for i in range(len(s))]) == 0:
            l, u, miss = misses(x,torch.tensor(y),m,s)
            success.append(l<=miss and miss<=u)
        if conv_time != float('inf'):
            tid.append(conv_time)
        end_epoch.append(conv_epoch)
        means.append(m.tolist())
        std.append(s.tolist())

    # Save results in a json file
    results = [means, std, sum(tid)/len(tid), sum(end_epoch)/(len(end_epoch)),
                float(100*sum(success)/len(success))]
    with open(f"Poly_results/poly_results_{length}_y_ax.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.7.3 Fallet $\alpha_i + \beta_i \cdot x$

```
import torch
import numpy as np
import scipy as sp
import time
import linear_combination_optimizer as lco
import json

# Polynomial nn method
def eval_powers_of_x(x, n):
    return x.pow(torch.arange(n))
def eval_even_powers_of_x(x, n):
    return x.pow(2 * torch.arange(n))

# Calculate confidence intervals and returns amount of misses
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        # Generate sample from model
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# Polynomial nn method
class NormalPolynomialModel(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.mean = None
        self.std = None

        self.N_mean = 2 # constant, linear, quadratic, ... terms
        self.N_var = 2

        self.poly_multipliers_mean = torch.nn.parameter.Parameter(torch.rand(self.N_mean))
        self.poly_multipliers_var = torch.nn.parameter.Parameter(torch.rand(self.N_var))
        self.var_shift = torch.nn.parameter.Parameter(torch.tensor(0.1))

        self.mean_layer = torch.nn.Linear(self.N_mean, 1)
        self.var_layer = torch.nn.Sequential(
            torch.nn.Linear(self.N_var, 1),
            torch.nn.ReLU()
        )

    def forward(self, x):
        self.mean = self.mean_layer(self.poly_multipliers_mean * eval_powers_of_x(x, self.N_mean))
        self.var = self.var_layer(self.poly_multipliers_var * eval_even_powers_of_x(x, self.N_var)) +
                   self.var_shift ** 2
        return self.mean, self.var

# Loss function for nn
def log_k_with_var(mean, var, y):
    return -0.5*torch.log(var) - (y - mean)**2 / (2 * var)

# For each data size
for length in [100, 500, 1000]:
    success=[]
    tid=[]
```

```
    end_epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_lin_{i}.npy')
        y = torch.tensor(y)
        x_unsq = x.view(-1, 1)
        y = y.view(-1, 1)
        model = NormalPolynomialModel()
        opt = torch.optim.Adam(model.parameters())

        max_epoch = 300000
        conv_epoch = max_epoch
        conv_time = float('inf')
        old_loss = float('inf')
        t1 = time.time()
        # Main loop of regression
        for epoch in range(max_epoch):
            opt.zero_grad()
            mean, var = model(x_unsq)
            log_likelyhood = log_k_with_var(mean, var, y).sum()
            loss = -log_likelyhood
            loss.backward()
            opt.step()
            if torch.abs(old_loss-loss) < 1e-4:
                t2 = time.time()
                conv_time = t2-t1
                conv_epoch = epoch
                break
            old_loss = loss

            if epoch % 1000 == 0:
                print(f'{epoch}:: Loss = {loss.item()}')

        m, var = model(x_unsq)
        s = var.sqrt()
        m = m.detach().numpy()
        s = s.detach().numpy()

        l, u, miss = misses(x,torch.tensor(y),m,s)
        success.append(l<=miss and miss<=u)
        if conv_time != float('inf'):
            tid.append(conv_time)
        end_epoch.append(conv_epoch)
        means.append(m.tolist())
        std.append(s.tolist())

    # Save results in a json file
    results = [means, std, sum(tid)/len(tid), sum(end_epoch)/(len(end_epoch)),
                float(100*sum(success)/len(success))]
    with open(f"Poly_results/poly_results_{length}_y_lin.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.7.4 Fallet $\alpha_i + \beta_i \cdot x + \gamma_i \cdot x^2$

```
import torch
import pytorch_measure as pm
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
from scipy import stats
import time
import linear_combination_optimizer as lco
import json

# Polynomial nn method
def eval_powers_of_x(x, n):
    return x.pow(torch.arange(n))
def eval_even_powers_of_x(x, n):
    return x.pow(2 * torch.arange(n))

# Calculate confidence intervals and returns amount of misses
def misses(x, y, mu, sigma):
    miss = 0
    S = 1000
    for i in range(x.size(dim=0)):
        # Generate sample from model
        sample = torch.normal(mean = float(mu[i]), std = float(sigma[i]), size = (1,S)).squeeze(dim=0)
        sample = torch.sort(sample)[0]
        if y[i] < sample[int(np.floor(S*0.025))] or y[i] > sample[int(np.ceil(S*0.975))-1]:
            miss += 1
    # Calculate confidence interval
    c1 = sp.stats.binom.ppf(0.025,y.size(dim=0),0.05)
    c2 = sp.stats.binom.ppf(0.975,y.size(dim=0),0.05)
    print(f"CI: [{c1}, {c2}], Misses: {miss}, Within CI: {c1<=miss<=c2}")
    return c1, c2, miss

# Polynomial nn method
class NormalPolynomialModel(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.mean = None
        self.std = None

        self.N_mean = 3 # constant, linear, quadratic, ... terms
        self.N_var = 3

        self.poly_multipliers_mean = torch.nn.parameter.Parameter(torch.rand(self.N_mean))
        self.poly_multipliers_var = torch.nn.parameter.Parameter(torch.rand(self.N_var))
        self.var_shift = torch.nn.parameter.Parameter(torch.tensor(0.1))

        self.mean_layer = torch.nn.Linear(self.N_mean, 1)
        self.var_layer = torch.nn.Sequential(
            torch.nn.Linear(self.N_var, 1),
            torch.nn.ReLU()
        )

    def forward(self, x):
        self.mean = self.mean_layer(self.poly_multipliers_mean * eval_powers_of_x(x, self.N_mean))
        self.var = self.var_layer(self.poly_multipliers_var * eval_even_powers_of_x(x, self.N_var)) +
                                    self.var_shift ** 2
        return self.mean, self.var

# Loss function for nn
def log_k_with_var(mean, var, y):
    return -0.5*torch.log(var) - (y - mean)**2 / (2 * var)

# For each data size
```

```
for length in [100, 500, 1000]:
    # Lists for storing results
    success=[]
    tid=[]
    end_epoch=[]
    means=[]
    std=[]
    # Run regressions 50 times
    for i in range(50):
        x = torch.linspace(-5, 5, length)
        y = np.load(f'../Finalized/test_data/data_{length}_y_sqr_{i}.npy')
        y = torch.tensor(y)
        x_unsq = x.view(-1, 1)
        y = y.view(-1, 1)
        model = NormalPolynomialModel()
        opt = torch.optim.Adam(model.parameters())

        max_epoch = 300000
        conv_epoch = max_epoch
        conv_time = float('inf')
        old_loss = float('inf')
        t1 = time.time()
        # Main loop of regression
        for epoch in range(max_epoch):
            opt.zero_grad()
            mean, var = model(x_unsq)
            log_likelyhood = log_k_with_var(mean, var, y).sum()
            loss = -log_likelyhood
            loss.backward()
            opt.step()
            if torch.abs(old_loss-loss) < 1e-4:
                t2 = time.time()
                conv_time = t2-t1
                conv_epoch = epoch
                print(f'{epoch}:: Loss = {loss.item()}')
                break
            old_loss = loss

            if epoch % 1000 == 0:
                print(f'{epoch}:: Loss = {loss.item()}')

        m, var = model(x_unsq)
        s = var.sqrt()
        m = m.detach().numpy()
        s = s.detach().numpy()

        l, u, miss = misses(x,torch.tensor(y),m,s)
        success.append(l<=miss and miss<=u)
        if conv_time != float('inf'):
            tid.append(conv_time)
        end_epoch.append(conv_epoch)
        means.append(m.tolist())
        std.append(s.tolist())

    # Save results in a json file
    results = [means, std, sum(tid)/len(tid), sum(end_epoch)/(len(end_epoch)),
               float(100*sum(success)/len(success))]
    with open(f"Poly_results/poly_results_{length}_y_sqr.json", "w") as outfile:
        outfile.write(json.dumps(results))
```

C.8 Implementation av linjärkombination av stokastiska variabler

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import time

class Optimizer():
    def __init__(self, x, y, order=3, n=3, ax = False):
        self.order = order
        self.n = n
        self.x = x
        self.y = y
        if ax == False:
            self.h = [self.create_lambda(i) for i in range(self.order)]
        else:
            self.h = [self.create_lambda(1)]
        h_all_data = [h(x) for h in self.h]
        self.h_all = torch.transpose(torch.stack(h_all_data, 0), 0, 1)
        self.mu = torch.tensor([0. for _ in range(order)], dtype=float, requires_grad=True)
        self.sigma = torch.tensor([1. for _ in range(order)], dtype=float, requires_grad=True)
        self.beta = [self.mu, self.sigma]

    def create_lambda(self, i):
        if i == 0:
            return lambda u: u*0+1
        return lambda u:u**i

    def optimize(self, epochs=300, lr=0.1, print_frequency = 10, test = False):
        optimizer = torch.optim.Adam(self.beta,lr=lr, maximize=True)
        old_loss = float('inf')
        cur_epoch = epochs
        t1 = time.time()
        t2 = float('inf')     # In case optimization doesnt terminate before max epochs are reached
        for epoch in range(epochs):
            optimizer.zero_grad()
            loss = self.log_lik(self.y, self.beta, self.h_all)
            loss.backward()
            optimizer.step()
            if epoch%print_frequency==0:
                print(epoch, "mu:", self.mu.detach().numpy(), "sigma:", self.sigma.detach().numpy(), "loss:", loss)
            if torch.abs(loss-old_loss) < 1e-15:
                t2 = time.time()
                cur_epoch = epoch
                break
            old_loss = loss
        self.mu_optim = self.beta[0].detach().numpy()
        self.sigma_optim = self.beta[1].detach().numpy()
        print(epoch, "mu:", self.mu_optim, "sigma:", self.sigma_optim)
        if test == False:
            return [self.m(self.mu, self.h_all[i,:]).detach().numpy().tolist() for i in range(self.x.size(dim=0))],
                    [(self.sigma_2(self.sigma,
                                self.h_all[i,:])**0.5).detach().numpy().tolist() for i in range(self.x.size(dim=0))]
        else:
            if t2 != float('inf'):
                return [self.m(self.mu, self.h_all[i,:]).detach().numpy().tolist() for i in range(self.x.size(dim=0))],
                       [(self.sigma_2(self.sigma,
                                   self.h_all[i,:])**0.5).detach().numpy().tolist() for i in range(self.x.size(dim=0))],
                      cur_epoch, t2-t1
            else:
                return [self.m(self.mu, self.h_all[i,:]).detach().numpy().tolist() for i in range(self.x.size(dim=0))],
                        [(self.sigma_2(self.sigma,
                                   self.h_all[i,:])**0.5).detach().numpy().tolist() for i in range(self.x.size(dim=0))],
                       cur_epoch, None


```

```
    def m(self, mu, h_x):
        return (mu * h_x).sum()

    def sigma_2(self, sigma, h_x):
        return (sigma * h_x).pow(2).sum()

    def sigma_2_regular_exp(self, sigma, h_x):
        return ((sigma * h_x)**2).sum()

    def log_normal_pdf(self, y, mu, sigma_2):
        return -1/2*(torch.log(2*np.pi*(sigma_2)) + ((y-mu)**2/sigma_2))

    def log_lik(self, y, beta, h_all):
        return sum([self.log_normal_pdf(y_i, self.m(beta[0], h_all[i]),
                                        self.sigma_2(beta[1], h_all[i])) for i, y_i in enumerate(y)])

    def get_plot_CI(self, mu, sigma, n=3):
        y_new = []
        sigma_upper = []
        sigma_lower = []
        sample_x = []
        sample_y = []

        for i in range(len(self.x)):
            h_x = [h_i.item() for h_i in self.h_all[i]]
            y_new.append(self.m(mu, h_x))
            s = self.sigma_2_regular_exp(sigma, h_x)**0.5
            sigma_upper.append(y_new[-1] + 2*s)
            sigma_lower.append(y_new[-1] - 2*s)
            sample = torch.normal(float(y_new[-1]), float(s), size = (1, n))
            for j in range(n):
                sample_x.append(self.x[i].item())
                sample_y.append(sample[0][j].item())
        return y_new, sigma_upper, sigma_lower

    def visualize(self):
        y_new, sigma_upper, sigma_lower = self.get_plot_CI(self.mu_optim, self.sigma_optim)
        markersize = 20
        plt.scatter(self.x, self.y, sizes=[markersize]*len(self.x))
        plt.plot(self.x, y_new, 'r-')
        plt.plot(self.x, sigma_upper, 'r--')
        plt.plot(self.x, sigma_lower, 'r--')
        plt.show()
```