CHALMERS | GÖTEBORGS UNIVERSITET

Kolmogorov-Arnold Nätverk, Framtidens AI?

Kolmogorov-Arnold Networks: the Future of AI?

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Kandidatarbete inom civilingenjörsutbildningen vid Chalmers

Oscar Eliasson
Jolie Larsen
Alexander Malmquist
Måns Redin

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2025

Kolmogorov-Arnold Nätverk, Framtidens AI?

Kolmogorov-Arnold Networks: the Future of AI?

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk Matematik vid Chalmers

Oscar Eliasson
Alexander Malmquist

Kandidatarbete i matematik inom Matematikprogrammet vid Göteborgs Universitet

Jolie Larsen

Kandidatarbete i matematik inom civilingenjörsprogrammet Kemiteknik med Fysik vid Chalmers

Måns Redin

Handledare: Sergei Zuyev Matematiska vetenskaper

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2025

Förord

Tack till Sergei Zuyev, vår handledare för den teoretiska vägledningen, diskussionerna och en positiv attityd. Vi vill även rikta ett tack till handledarna på Chalmers bibliotek som har bistått med goda råd till denna rapport. Slutligen vill vi tacka institutionen för Data- och Informationsteknik vid Chalmers Tekniska Högskola för tillgång till deras kraftfulla datorer.

Under arbetet har en loggbok förts för att hålla koll på arbetet som utförts varje vecka, samt de individuella prestationerna.

Bidragsrapport

Avsnitt	Huvudförfattare	Övrig författare
Förord	Måns	
Populärvetenskaplig Presentation	Jolie	
Sammandrag	Alex	
Abstract	Alex	
Inledning	Alex	
Syfte och hypotes	Alex	Jolie
Teori(inledande stycke)	Alex	Jolie
Neurala nätverk som generella funktionsapproximerare	Alex	Jolie
MLP-noden	Alex	Jolie
KAN-noden	Alex	Jolie
Bevisidé av KAT	Jolie och Oscar	
Sprechers modifiering av KAN	Jolie och Oscar	
Tidigare resultat Sprecherrepresentation	Oscar	
Faltningsnätverk	Måns	Jolie
Bakåtpropagering	Måns	Jolie
Metod (inledande stycke)	Måns	
Implementation (inlednande text)	Oscar	
Implementation (KAN)	Måns	
Implementation (SKAN)	Oscar	
Datahantering och val av strukturer	Måns	
Bakåtpropagering	Måns	
Metod experiment 1: Funktionsanalys	Måns	
Metod experiment 2: Bildregression och rekonstruktion	Måns	
Metod experiment 3: Bildklassificering	Måns	Jolie
Resultat (inledande stycke)	Måns och Jolie	
Resultat experiment 1: Funktionsanalys	Måns och Jolie	
Resultat experiment 2: Bildregression och rekonstruktion	Måns och Oscar	Jolie
Resultat experiment 3: Bildklassificering	Måns och Oscar	Jolie
Diskussion (inledande stycke)	Måns	
Diskussion KAN	Måns	Jolie
Diskussion SKAN	Oscar	Jolie
Diskussion MLP och faltningsnätverk	Måns	
Samhälleliga och etiska aspekter	Måns	
Slutsats	Måns	
Framtidsutsikter	Måns och Oscar	

Tabell 1: Bidragsrapport för arbetet.

Övriga insatser

	Övriga insatser
Alex	• Skrivit all kod för det underliggande KAN-bibloteket. Måns har därefter lagt till kod för att köra experiment och producera resultat. • Läst igenom tidigare forskning för att utvärdera eventuella nya forskningsfrågor. Realiserat blev detta endast ett stycke i teori under bas-splines om att tidigare forskning etablerat att man kan approximera bas-splines genom radiella bas funktioner för bättre prestanda. • Producerat figurer för MLP och KAN-noden.
Jolie	• Haft ansvar för att se till att texten är enhetlig genom att omformulera stycken och meningar, korta ner text, korrekturläsa osv. • Hjälpt till med- och testat kod för sprecher för att se till att den ger bästa möjliga prestanda • Producerat resultat för Sprecher till funktionsanalysen genom att beräkna nätverk av önskat parameterantal samt använt koden • Kodat graferna till resultat • Undersökt beviset till Kolmogorv-arnolds sats, samt tagit fram bevisidé • Producerat bild till struktur på SKAN
Måns	• Skrivit all kod för analys av MLP, CNN och KAN. I KAN-skriptet har jag skrivit funktionerna train_model, batch_train_model, get_individual_splines, plot_splines och test_model. Skrivit skriptet för analys av KAN, analys av MLP, klassificering av KAN och CNN. • Kommenterat koden för KAN-, CNN- och MLP-skripten. • Producerat resultaten för KAN, MLP och CNNs. • Deltagit på tre fackspråkliga föreläsningar.
Oscar	• Undersökt beviset till Kolmogorv-arnolds sats, samt tagit fram bevisé • Undersökt Sprecherrepresentationer och forskning, kommit på och tagit fram nya representationer som använts som grund för SKAN. • Programmerat SKAN modellerna, producerat modeller utifrån Sprecherrepresentationerna och nya ideerna och korregeringarna som fungerar väl. • Testat, tagit fram resultat och analyserat resultaten och SKAN.
Alla	• Byggt upp kunskap kring ämnet genom att läsa källor • Varit dagboksansvarig var fjärde vecka

Populärvetenskaplig presentation

Har du använt AI någon gång? Artificiell intelligens, eller AI som det också kallas, växer i popularitet i dagens samhälle i en otrolig fart och används av allt från programmering och redigering till chattbotar och filter. Det har växt i en sådan fart att AIs relevans och uppkomst har liknats med när internet och mobiltelefoner uppfanns, uppmärksammades och normaliserades i samhället. I stort sett kan alla gruppers arbete förenklas på något sätt med denna nya teknik. Dagens artificiella intelligens bygger på klassiska artificiella neurala nätverk, men finns det inga andra sätt den kan konstrueras på? Behövs det fler sätt?

Först och främst är ett neuralt nätverk ekvationer som är sammanbundna på ett sätt sådant att du kan stoppa in en siffra, bild eller ekvation i ena änden, och så får du ut en annan siffra, en återskapning av bilden, eller ett svar på om det till exempel finns en hund på bilden eller om du stoppat in en sjua. För att se hur bra ett sådant nätverk fungerar brukar felet mätas och jämföras, så om den gissar rätt på att du har lagt in en sjua nio av tio gånger finns det ett fel på 10 procent. Det som skiljer olika uppbyggnader av nätverk åt är vilka ekvationer som används och hur de är strukturerade.

Under 1960-talet kom två ryska matematiker, Andrej Kolmogorov och Vladimir Arnold, fram till en sats som på den tiden inte verkade applicerbar på särskilt många användningsområden. Den kom att kallas Kolmogorov-Arnolds representationssats, och dess betydelse uppmärksammades först förra året när en rad forskare släppte en rapport där de visar hur satsen kan användas till att skapa AI. Anledningen till att det är intressant att undersöka andra uppbyggnader är för att de kan fungera bättre i vissa användningsområden, och självklart vill man uppnå den bästa möjliga artificiella intelligensen. Dessutom är detta ett område med stort utrymme för utveckling och justeringar då det är ett relativt nytt verktyg i samhället.

Detta projekt undersöker därför om så kallade Kolmogorov-Arnold nätverk kan konkurrera med de klassiska nätverken. Undersökningen bygger på att jämföra vilken uppbyggnad som presterar bäst, samt vilka som presterar bättre i olika avseenden. För att göra detta programmerar vi ett flertal nätverk som bygger på Kolmogorov-Arnolds implementation, samt klassiska nätverk. Med hjälp av dessa går det dessutom att jämföra vilken implementering av Kolmogorov-Arnolds sats som ger bäst prestanda. Detta görs genom att testa nätverken på både en funktion och bilder, och nätverket med lägst fel är det som presterar bäst. Vi låter också nätverken göra förstoringar av bilden för att se om den kan uppskatta vilka värden de extra pixlarna borde anta.

Samtidigt finns det många kontroversiella aspekter kring artificiell intelligens, som kan få en att undra om fortsatt forskning och utveckling kring ämnet verkligen kommer att gynna samhället? Trots att det finns många motståndare till denna sorts utveckling går det redan att hitta AI överallt runt omkring oss, till och med i de flesta appar i våra mobiler. Eftersom många redan förlitar sig på denna typ av verktyg, är det till vår egen fördel om de också presterar så bra som möjligt. Samtidigt handlar inte frågan om att den vanliga strukturen på artificiell intelligens inte presterar tillräckligt bra, utan mer om det är möjligt att få lika bra resultat med ett mindre antal parametrar. Parametrar är de tal i ekvationerna som skiljer uppbyggnaderna åt, och ett lägre antal parametrar innebär att mindre energi och tid behöver användas till nätverken, då det är färre beräkningar som krävs. Därmed kan en sådan egenskap faktiskt ses som något positivt ur ett miljöperspektiv.

Sammanfattningsvis är artificiell intelligens något vi behöver acceptera då det är ett användbart verktyg som redan omger oss dagligen. Arbetet med att förbättra dessa tekniker är viktigare än någonsin då vi behöver göra det bästa vi kan med energin som används, för att AI inte bara ska vara något vi kan använda nu, utan också långt framöver.

Sammandrag

Artificiell intelligens integreras idag i alltfler tekniska system. Trots det, är det fortfarande svårt att i praktiken implementera modeller från grunden på grund av höga resurskrav i form av data och beräkningskraft. Detta motiverar sökandet efter effektivare modeller.

I denna artikel undersöks *Kolmogorov Arnold Networks* (KANs), en alternativ arkitektur av neurala nätverk och jämförs med traditionella arkitekturer som *multilayer perceptrons* (MLPs) och *faltningsnätverk*. KANs är intressanta då de potentiellt erbjuder fördelar som högre noggrannhet, är lättare att tolka och innehar teoretiskt goda skalningslagar [1].

Vi undersöker KANs och särskilda modiferingar av dessa (Sprecher-KANs) relativt MLPs. Jämförelsen görs genom att analysera utfallet från tre experiment. Specifikt analyseras modellernas förmågor för kurvanpassning, bildrekonstruktion och bildklassificering. I resultaten uppvisar Sprecher-KANs högre prestanda än KAN och i sin tur att KAN överpresterar MLPs. Faltningsnätverken överträffar KANs i prestanda, medan Sprecher-KANs presterar likvärdigt med faltningsnätverken.

?abstractname?

Artificial intelligence is being integrated into more and more technical systems. Nevertheless, in practice, it is still difficult to implement models from scratch due to high resource requirements in terms of data and computational power. This motivates the search for more efficient models.

This article investigates *Kolmogorov Arnold Networks* (KANs), an alternative model to traditional *multilayer perceptrons* (MLPs) and convolutional neural networks (CNNs). KANs are interesting because they potentially have performance advantages such as higher accuracy, better interpretability and theoretically better scaling laws [1].

We investigate KANs and special modifications of these (Sprecher-KANs) relative to MLPs and CNNs. The comparison is done by analyzing the outcome of three experiments. The models are tested for curve fitting, image reconstruction and image classification. The results show that
Sprecher-KANs perform the best in terms of accuracy for curve fitting and image reconstruction, subsequently KANs and finally MLPs. For image classification CNNs outperform KANs, while Sprecher-KANs perform at the same level as CNNs.

Innehåll

Begreppslista

FFNN - Feed-Forward Neural Networks, synonymt med MLP
GAM - Generalized Additive Model.
KAN - Kolmogorov Arnold Networks.
KAT - Kolmogorov Arnold Theorem.
MLP - Multi-Layer Perceptron.
SKAN - Sprecher Kolmogorv Arnold Networks.
MSE - Medelkvadratfel.
MAE - Medelabsolutfel.
Epok - En runda där hela träningsdatan propagerats till nätverket.
Seed - Startvärde som används för att initiera en konsekvent slumptalsgenerator.

1 Inledning

Artificiell intelligens är mer relevant än någonsin. Framstegen inom fältet har från grunden ändrat vad som är tekniskt möjligt och medfört att tekniken i allt större utsträckning integreras i system. Trots det är det ofta svårt att integrera modeller i praktiken. En betydande anledning är att de är resurskrävande; ofta behövs mycket beräkningskraft och data. Detta leder till försämrad tillgänglighet då många användare inte har resurserna som krävs för ändamålet. Detta motiverar sökandet efter effektivare modeller.

Under våren 2024 lyfte forskare upp *Kolmogorov Arnold Networks* (KANs) [1], en alternativ underliggande modell till den mer traditionella modellen *multilayer perceptron* (MLP). KANs fungerar fundamentalt annorlunda i att *aktiveringsfunktionerna* i nätverket ej är förutbestämda, istället lär de sig lämpliga aktiveringar. KANs utlovar fördelar i vissa miljöer gentemot MLPs, bland annat högre noggrannhet, tolkningsförmåga och teoretiskt bättre skalningslagar [1]. KANs förenklar genom att bryta ned problemet i beståndsdelar i form av endimensionella funktioner.

I detta arbete jämför vi övergripande tre olika modeller; traditionella MLPs, KANs och en särskild typ av modifierade KANs, så kallade Sprecher-KANs. Modellerna jämförs genom analys av utfallet för tre olika experiment. I det första experimentet analyseras modellernas förmåga att kurvanpassa sig till funktionen $4xy^2$, och i det andra ska modellerna genomföra bildrekonstruktion. Sist jämförs modellernas förmåga att klassificera olika typer av bilder från *Pneumonia MNIST* datasetet.

Resultaten från experimenten blev att olika typer av Sprecher-nätverk presterat bäst utifrån ett noggrannhetsperspektiv för både kurvanpassning och bildrekonstruktion. I det tredje experimentet blev resultatet tvetydigt.

1.1 Syfte och hypotes

Syftet är att undersöka KANs och Sprecher-modifierade KANs, samt faltningsnätverk, för att se om det finns sammanhang då modellerna presterar bättre än traditionella MLPs utifrån aspekter som noggrannhet, minne och konvergens. Det som är särskilt intressant är att analysera nätverken när de tillämpas på bildanalys, då en av hypoteserna är att KANs kan nyttja symmetrier i bilder som MLPs inte kan. Den andra hypotesen är att KANs noggrannhet bör vara högre eftersom vid samma mängd parametrar har de fler frihetsgrader än MLPs.

2 Teori

För att möjliggöra jämförelsen och utvecklingen av neurala nätverk presenteras det här den grundläggande teorin. Delkapitlen är strukturerade på ett sätt så att informationen går från övergripande till detaljerad. Först diskuteras funktioner och hur de kan approximeras av neurala nätverk. Sedan beskrivs MLP- och KAN-noderna innan Sprechers modifiering av KAN tas upp, samt tidigare forskning som understödjer detta. Vidare förklaras faltningsnätverk, då de lämpar sig för bildanalys. Slutligen redogörs den mer allmänna tekniken för hur vikter kan uppdateras baklänges i ett nätverk kallat *bakåtpropagering*.

2.1 Neurala nätverk som funktionsapproximerare

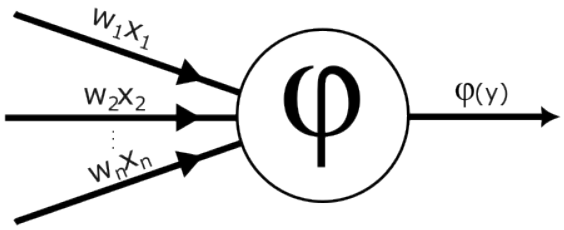
Funktioner kan vara svåra att explicit uttrycka, och approximering leder ofta till enklare hantering och bättre resultat. Neurala nätverk är universella funktionsapproximerare för en relativt allmän klass funktioner. Detta är ett centralt resultat för teorin inom universella approximationssatser. Både MLPs och KANs har ett motsvarande resultat var inom denna klass. MLPs bygger på den universella approximationssatsen medan KANs på Kolmogorov-Arnold representationssats. Explicit anger satsen för KANs att alla kontinuerliga, multivariata funktioner av typen $f : [0,1]^n \to R$ kan ekvivalent uttryckas som en summa av endimensionella funktioner. Om $\phi_{q,p} : [0,1] \to \mathbb{R}$ och $\Phi_q : \mathbb{R} \to \mathbb{R}$ gäller specifikt

$$f(\mathbf{x}) = f(x_1, \ldots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right). \tag{1}$$

Universella approximationssatsen kan ekvivalent tolkas inom KAN, då satsen till dessa anger att alla funktioner av tidigare nämnd klass kan uttryckas av ett KAN bestående av två lager. Enligt (1) är $\phi_{q,p}$ de inre funktionerna medan Φ_q är de yttre i nätverket. Vidare behövs n inre funktioner och $2n+1$ yttre funktioner, vilket garanterar existensen av ett nätverk som approximerar den bundna funktionen. Nätverkets funktion blir en sammansättning av flera approximerade funktioner. I praktiken generaliseras nätverken så att de har bredare och djupare lager. Artikel ([2]) motiverar varför det är möjligt.

2.2 MLP-noden

MLP-noden, eller perceptron, modellerar tills viss del den mänskliga hjärnans neuroner. Detta sker genom att reproducera två av dess mekanismer; att ta emot och skicka vidare signaler. Modellen beskrivs matematiskt med en riktad graf bestående av en nod med flera ingående kanter [3]. Se figur (1).



Figur 1: MLP-noden. I noden finns även en tröskelparameter b men som är utelämnad från bilden.

Noden tar emot signaler längs med grafens kanter och utgörs av linjära funktioner på formen $y_k = w_k x_k$ för kant k. Insignalerna förs sedan in i noden genom summationen $\sum_{k=1}^{n} w_k x_k - b$ till aktiveringsfunktionen φ, som bestämmer värdet hos utsignalen. Aktiveringsfunktionen är en förutbestämd vald funktion, ofta tanh, ReLU eller sigmoid, som beror av insignalerna och en konstant parameter b som agerar som ett tröskelvärde. Om summan av insignalerna är större än tröskelvärdet så är φ positiv, och noden säges då vara aktiverad. Annars är $\varphi < 0$ och motsvarar då en dämpning.

Ett viktigt krav i ett nätverk bestående av MLP-noder är att en av aktiveringsfunktionerna i nätverket är icke-linjär. Annars kan nätverket endast uttrycka linjära samband [4], vilket de flesta inte är.

2.2.1 Antalet modellparametrar i nod och nätverk för MLPs

Modellparametrar är variabler som anpassar modellens funktionskurva och erhålls genom estimering av data, vilket inom maskininlärning kallas för att man ”tränar” modellen. Desto fler parametrar en modell innehar, desto mer resurser såsom minne och beräkningar krävs för att lagra, använda och träna nätverket. Antalet modellparametrar är olika för olika modeller, även när de delar samma grafmässiga struktur, det vill säga lika många noder och kanter. För att göra rättvisa jämförelser behöver därför både den grafmässiga strukturen och antalet parametrar hos nätverken tas i åtanke. Antalet parametrar i MLP-noden ges av följande samband

$$P_{nod} = |V_{nod}| + 1. \tag{2}$$

Där P_{nod} betecknar antalet parametrar för noden och V_{nod} för antalet kanter in i noden. Sambandet motiveras av att vardera ingående kant bidrar med en var, samt att noden bidrar med en, konstanten b. Antalet parametrar i en MLP ges av sambandet

$$P_{MLP} = \sum_i P_i = \sum_i N_i P_{nod} = \sum_i N_i(|V_i| + 1). \tag{3}$$

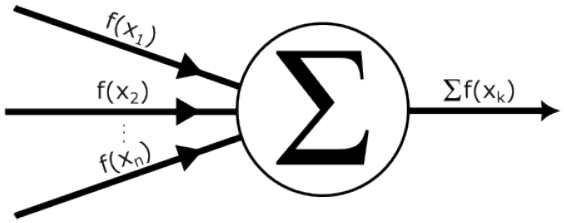
I ekvationen betecknar P_{MLP} antalet parametrar för en MLP, P_i antalet för lager i. N_i anger antalet noder för lager i, V_i antalet ingående kanter per nod för lager i. Sambanden motiveras från det att antalet parametrar i ett nätverk ges av summan av antalet från vardera lager. Antalet i ett lager ges från vardera nod i lagret. Eftersom det finns N_i noder så multipliceras antalet med antalet parametrar per nod.

2.3 KAN-noden

KAN-noden skiljer sig konceptuellt från MLP-noden i att aktiveringsfunktionen för noden **inte** är förutbestämd, utan hittas i stället genom träning. I modellen defineras den som summan av kantfunktionerna och betecknas endast med en summationssymbol i figur (2).

Nodens ingående kantfunktioner $f(x_k)$ uttrycks som linjärkombinationer av en angiven, tidigare bestämd bas $\{e_i\}_{i=1}^{n}$ med tillhörande koefficenter $\{c_j\}_{j=1}^{n}$. Likt [1] avgränsas studien till att använda bas-splines (se delkapitel 2.3.1), men valet påverkar bland annat minne, beräkningskomplexitet och träningskonvergens. Tidigare genomförd forskning har visat att bas-splines kan fördelaktigt approximeras och ersättas av radiella basfunktioner för att erhålla en modell som i snitt tränas tre

gånger så fort utan avsevärda förluster i noggrannhet [5]. Målet vid träning är att hitta konstanter $\{c_j\}_{j=1}^{n}$ sådana att önskat beteende uppnås genom nodaktivering.



Figur 2: KAN-noden.

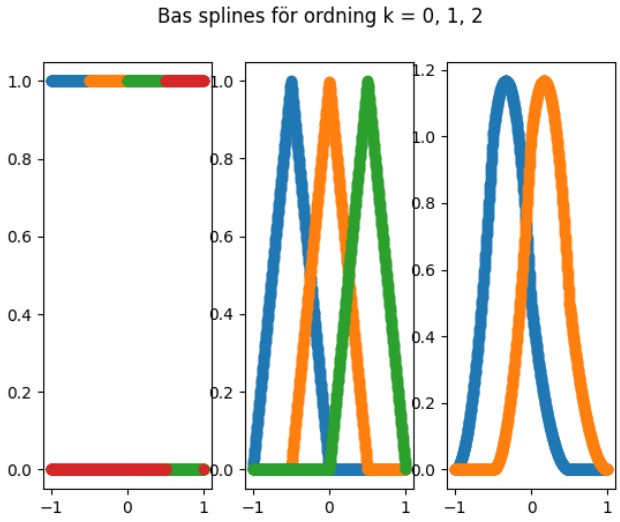
2.3.1 Bas-splines

I denna del tas teori för bas-splines upp då de valts som bas för KAN-nätverken. En spline är en funktion som utgörs av styckvisa polynom. Bas-splines är en spline som utgör en bas till polynomrummet P. Specifikt är de konstruerade så att de har minimalt överlapp (stöd) över vardera delintervall, och så att de är kontinuerligt deriverbara upp till ordning n-1. Bas-splines kan konstrueras genom Cox-De-Boors rekursionsformel, se ekvationer (4) och (5) som behandlar basfallet respektive det allmänna fallet. Härledning av Cox-de-Boors rekursionsformel återges i [6].

$$B_{i,0}(x) := \begin{cases} 1 & \text{om} \quad t_i \leq x < t_{i+1} \\ 0 & \text{annars} \end{cases} \tag{4}$$

$$B_{i,p}(x) := \frac{x - t_i}{t_{i+p} - t_i} B_{i,p-1}(x) + \frac{t_{i+p+1} - x}{t_{i+p+1} - t_{i+1}} B_{i+1,p-1}(x). \tag{5}$$

Figur (3) visar bas-splines definierade över ett intervall $\mathbf{I}$ för bas-splines av olika ordning.



Figur 3: Bas-splines för ordning k = 0, 1, 2 över intervallet $\mathbf{t} \in [-1, 1]$ där antalet knutpunkter är $|t| = 5$. Från vänster till höger.

Antalet bas-splines som erhålls över ett intervall $\mathbf{I}$ med $|\mathbf{t}| = t$ knutpunkter och polynom av grad k ges av följande samband

$$|e| = t - (k + 1). \tag{6}$$

Där $|e|$ betecknar antalet bas-splines, t antalet knutpunkter och k graden av polynomen hos bas-splinesen. Motiveringen till ekvation (6) är att vardera styckvist polynom kräver $k + 1$ knutpunkter för att vara kontinuerligt definierade. Dessa måste sedan definieras över $t - (n + 1)$ intervall.

För att i praktiken uppnå en högre noggrannhet behövs det fler bas-splines än de som erhålls ur Cox-de-Boors rekursionsformel, vilket kan uppnås genom en teknik som kallas för rutnätsutvidgning [1]. Ekvation (7) anger antalet parametrar efter att rutnätsutvidgning tillämpats.

$$|e| = t - (k+1) + 2k = t + k - 1 \tag{7}$$

2.3.2 Antalet parametrar i ett KAN

Enligt samma motivering som för MLPs är antalet parametrar hos ett KAN av intresse att erhålla. Antalet i en enskild nod ges av

$$P_{nod} = |e||V| = (t + k - 1)|V| \tag{8}$$

Där $|e|$ betecknar antalet basfunktioner och ges av ekvation (7), och i praktiken tillämpas rutnätsutvidgning. $|V|$ anger antalet kanter in i noden, t är antalet knutpunkter och k är graden hos basfunktionerna. Ett lager består av flera noder och har

$$P_{lager} = N_{lager} P_{nod} = N_{lager}|V|(t + k - 1) \tag{9}$$

antal parametrar. Ett KAN består av flera lager och har därmed följande antal parametrar;

$$P_{KAN} = \sum_{lager=1}^{n} P_{lager} = \sum_{lager=1}^{n} N_{lager}(t + k - 1)|V|. \tag{10}$$

2.4 Beviside av KAT

Beviset till KAT består av två delar, ett för den inre summan och ett för den yttre. Idén till den första halvan bygger på att dela upp enhetslinjen E^1 i segment beroende av n som betecknas $A^q_{k,i}$, och definieras enligt följande;

$$A^q_{k,i} = [\frac{1}{(9n)^k}(i - 1 - \frac{q}{3n}), \frac{1}{(9n)^k}(i - \frac{1}{3n} - \frac{q}{3n})]. \tag{11}$$

Där n är dimensionen och följande index uppfyller $1 \leq p \leq n$, $1 \leq q \leq 2n + 1$, $k = 1, 2, ...$ och $1 \leq i \leq (9n)^k + 1$. Sedan definieras kuber $S^q_{k,i_{1,\ldots,n}}$ på E^n genom att använda $A^q_{k,i}$ på följande sätt

$$S^q_{k,i_{1,\ldots,n}} = \Pi_p A^q_{k,i_p}. \tag{12}$$

Både segmenten och kuberna har små mellanrum mellan sig, men tillsammans över alla q täcker de varje punkt i E^n minst $n + 1$ gånger. Beviset fortsätter med att visa att det går att välja ut konstanter $\lambda^{p,q}_{k,i}$ på intervallen $A^q_{k,i}$ med skillnad mindre än $\frac{1}{2^k}$ till nästa $\lambda^{p,q}_{k,i+1}$, vilket gör att det går att definiera en unik kontinuerlig funktion mellan konstanterna.

Den andra delen av beviset påvisar existensen av de yttre funktionerna Φ^q genom konstruktion som gränsfunktion i en funktionsserie, där $\lim_{r\to\infty} \Phi^q_r$, $\Phi^0_r = 0$ och Φ^q_r bestäms med induktion på r ($r > 0$) och heltalen k_r. Låt $f_r(x_1, ..x_n) = \sum_{q=0}^{2n} \Phi^q_r \left(\sum_{p=1}^{n} \phi_{q,p}(x_p)\right)$. Genom att definiera $M_r = \sup_{E^n} |f - f_r|$ blir basfallet i induktionen $f_0 \equiv 0$, $M_0 = \sup_{E^r} |f|$ trivialt. Antag nu att Φ^q_{r-1}, k_{r-1} redan har bestämts, och därmed också f_{r-1}. Eftersom diametern på kuberna $S^q_{k,i_{1,\ldots,n}} \to 0$ då $k \to 0$ kan ett k_r väljas så att $|f - f_{r-1}| \leq \frac{1}{2n+2} M_{r-1}$ på godtycklig kub. För konstanter λ^{pq}_{k,i_p} existerar segmenten $\Delta^q_{k,i_{1,\ldots,n}} = [\sum_p \lambda^{pq}_{k,i_p}; \sum_p \lambda^{pq}_{k,i_p} + n\epsilon_k]$, som definierar Φ^q_r;

$$\Phi^q_r(y) = \Phi^q_{r-1} + \frac{1}{n+1}|f - f_{r-1}| \to |\Phi^q_r(y) - \Phi^q_{r-1}(y)| \leq \frac{1}{n+1} M_{r-1}. \tag{13}$$

Utanför dessa segment definieras Φ^q_r godtyckligt, så de uppfyller olikheten ovan. $f - f_r$ evalueras på en godtycklig punkt $x \in E^n$, vilket ger

$$f(x) - f_r(x) = f(x) - f_{r-1}(x) - \sum_q [\Phi^q_r(\sum_p \phi^{p,q}(x_p)) - \Phi^q_{r-1}(\sum_p \phi^{p,q}(x_p))]. \tag{14}$$

Oavsett om x ingår i $S^q_{k,i_{1,\ldots,n}}$ för varje term i summan över q eller inte, kan då summan begränsas av $\frac{1}{2n+2} M_{r-1} + \frac{1}{n+1} M_{r-1} = \frac{2n+1}{2n+2} M_{r-1}$. I sin tur gäller $\sup_{E^n} |f - f_r| = M_r \leq (\frac{2n+1}{2n+2})^r M_0$ och $\lim_{r\to\infty} M_r \to 0$. Därmed kommer Φ^q_r konvergera uniformt mot Φ^q då $r \to \infty$. Det fullständiga beviset till KAT finns i [2].

2.5 Sprechers modifiering av KAN

David Sprecher föreslog 1965 en alternativ formulering av Kolmogorov-Arnolds representationssats, som skrevs ner på följande sätt i [7].

Sprechers formulering av Kolmogorv-Arnolds representatinonssats
För varje tal $N \geq 2$ existerar det en reell, monoton växande funktion $\psi(x) \in Lip[\ln 2/\ln(2N+2)], \psi\xi = \xi$, beroende av N, som har den följande egenskapen:

För varje förutbestämt nummer $\delta > 0$ existerar ett rationellt nummer $\epsilon, 0 < \epsilon \leqq \delta$, sådant att för $2 \leqq n \leqq N$ gäller det att för varje reell kontinuerlig funktion med n variabler $f(x)$, definierad på ξ^n, har följande representation

$$f(\mathbf{x}) = f(x_1, \ldots, x_n) = \sum_{0 \leq q \leq 2n} \chi \left[\sum_{1 \leq p \leq n} \lambda_p \psi(x_p + \epsilon q) + q \right]. \tag{15}$$

Den största skillnaden mellan denna formulering och originalet är att i Sprechers notation har endast en funktion i varje summa, där den inre skalas och transformeras. Sprecher framställde ytterligare en representation år 1996, som visar att den yttre funktionen kan bytas ut mot $2n$ funktioner som är anpassade för varje q. Den yttre q-termen behövs därmed inte i den inre summan.

Sats (Sprecher, 1996)
Varje kontinuerlig funktion $f : I^n \to \mathbb{R}$ kan representeras som en summa av kontinuerliga reellvärda funktioner:

$$f(x_1, \ldots, x_n) = \sum_{q=0}^{2n} \chi_q(y_q). \tag{16}$$

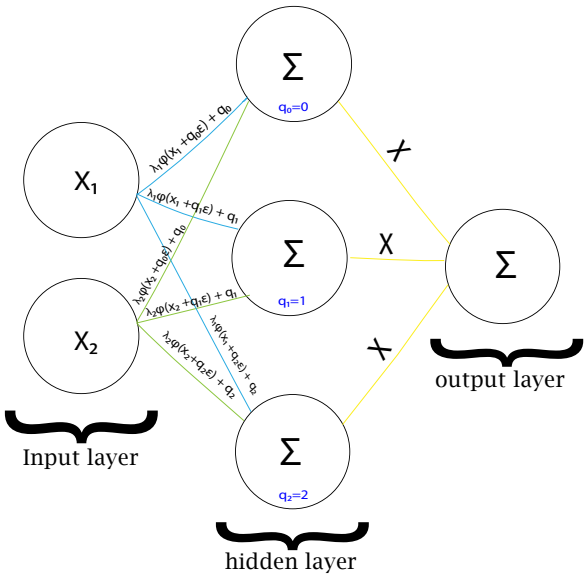
I denna representation är $x_1, \ldots, x_n$ parametrar för en inbäddning av I_n i $\mathbb{R}^{2n+1}$:

$$y_q = \eta_q(x) = \sum_{p=1}^{n} \lambda_p \psi(x_p + q\epsilon), \tag{17}$$

där ψ är en kontinuerlig reellvärd funktion och lämpliga konstanter λ_p och ϵ. Denna inbäddning är oberoende av f, satsen återfinns i [8].

En inbäddning är en avbildning från diskreta objekt till ett kontinuerligt flerdimensionellt vektorrum. I ett neuralt nätverk appliceras inbäddningar mellan lager, när utdatan transformeras till indatan i ett annat lager. Inbäddningar till högre dimensioner, alltså fler neuroner, kan ge nästa lager mer uttrycksfulla eller abstrakta egenskaper att arbeta med [9].

Enligt ovan, har Srechernätverk endast en lärbar aktiveringsfunktion mellan varje gömt lager. Denna skalas sedan med en vikt λ_p, för varje inmatningsnod, och transformerar indata med termen ϵq för varje neuron i nästa lager. Antalet λ- parametrar är lika många som summan av alla neuroner i alla lager borträknat de två sista, det finns lika många ϵ- parametrar som gömda lager minus ett. Till följd av detta faktum har SKAN betydligt färre parametrar jämfört med KAN. Sista inbäddningen görs antingen med en aktiveringsfunktion eller med en funktion per neuron i sista gömda lagret, vilket innebär fler parametrar. Nedan är ett exempel på ett nätverk baserat på ekvation (15).



Figur 4: Struktur på ett SKAN-nätverk.

2.6 Tidigare resultat Sprecherrepresentation

Forskningen och framgången av SKAN har varit begränsad, främst på grund av den komplexa naturen av den inre funktionen ψ. David Sprecher, matematikern bakom satsen, gav år 1997 en algoritm för beräkningen av den inre funktionen. Denna algoritm var däremot felaktig eftersom de inre funktionerna ψ inte var kontinuerliga. Detta åtgärdade Mario Köppen med en ny algoritm, återgiven i [10]. Dessa funktioner, även kallade Köppenfunktioner, är pseudofraktala och konstanta nästan överallt [10]. Den yttre funktionen fungerar då som en uppslagsmetod, då $f(x)$ ger värdet av χ under avbildningen av transformationen av ψ. Dessa beräkningar som skulle behövas för ett neuralt nätverk är däremot svåra och ineffektiva [11]. I [10] ger Sprecher en uppdaterad, mer parallelliserad algoritm för ett neuralt nätverk för funktioner av endast två variabler. I [11] används en universell Köppen-inbäddning oberoende av f och för yttre lagret används en Generalized Additive Model (GAM). De visar att denna modell kallad K-GAM möjligen kan vara ett alternativ till transformer-nätverk, med fördel av färre parametrar.

2.7 Faltningsnätverk

Ett *faltningsnätverk* är en variant av neurala nätverk som typiskt lämpar sig väl för applikationen bildanalys [3]. Detta nätverk består av en *faltningskomponent* och en MLP-komponent. Syftet med att implementera denna teknik är att extrahera egenskaper ur lokala regioner i bilden, istället för att träna ett nätverk på bildens färgvärden i pixlarna som motsvarande klassificering av en MLP gör. Tekniken använder filter, vilket är mindre matriser med kvadratisk struktur. Dessa appliceras stegvis, pixel för pixel, och beräknar en skalärprodukt mellan filtret och den lokala regionen i bilden [3]. Detta definieras för ett kvadratiskt filter med p pixlar i vardera dimension

$$u^{l+1}_{m,n,k} = \sum_{k=1}^{p_{l-1}} \sum_{i=1}^{p_l} \sum_{j=1}^{p_l} u^l_{m+i,n+j,k} w^l_{i,j,k}, \tag{18}$$

där u är värdet för faltningen i dess *interna* lager k i lager $l+1$ och spatiell position (m,n). I ekvationen ovan är p_l värdet av filtret i lager l och $w_{i,j,k}$ är värdet i kanal k av filtret i spatiell position (i,j). Filtret sveper rad för rad, utan att överskrida bildens gränser i någon dimension, där samtliga möjliga steg ger indatan till nästa faltningslager i nätverket. När flera filter används svarar dessa mot olika egenskaper i bilden, där egenskapernas komplexitet stiger i takt med lagrens ordning. Datan till första lagret av ett faltningsnätverk är av tvådimensionell representation, men när filter appliceras skapas lika många interna lager i datan till kommande lager, vilket ger ett internt djup. I det sista lagret av faltningskomponenten konverteras datan till endimensionell, som sedan förs till en MLP som tränas på den [3].

2.8 Bakåtpropagering

Parametrarna i nätverken tränas för att finna en *optimal* uppsättning som minimerar förlusten mellan data och nätverkets prediktioner. Detta görs med avseende på ett förutbestämt optimeringskriterium, där nätverket tränas genom en algoritm som kallas *bakåtpropagering*. Algoritmen är en iterativ process som uppdaterar parametrarna individuellt med avseende på en inlärningshastighet samt förändringen i nätverkets förlust med avseende på respektive parameter. Denna förändring bestäms med hjälp av kedjeregeln för partiella derivator applicerad på sammansättningar av funktioner i nätverket [3].

2.8.1 Vägar genom nätverket

Varje väg genom nätverket utgör en sammansättning av funktioner som beskriver hur data transformeras genom nätverket. För att bestämma hur en enskild parameter påverkar nätverkets förlustfunktion används en rekursiv algoritm som lager för lager, summerar bidraget från alla vägar mellan förlusten och parametern [3]. Eftersom varje nod är kopplad till flera andra noder uppstår många funktionella vägar. En parameter påverkar nätverkets förlust genom flera sådana vägar, och dess uppdatering beräknas som en summa av produkter av partiella derivator av aktiveringsfunktioner längs respektive väg.

2.8.2 Partiella derivator för sammansättning

Samtliga parametrars påverkan uttrycks genom partiella derivator som faktoriseras. Den första termen beskriver hur förlusten beror på nätverkets utvärde $\frac{\partial L}{\partial o}$, som bestäms av förlustfunktionen. För att beräkna förlustfunktionens beroende parameterförändring definieras följande samband för alla bakåtgående vägar $v_1, v_2, \ldots, v_n$;

$$\frac{\partial L}{\partial c_{v_{l-1},v_l}} = \frac{\partial L}{\partial o} \underbrace{\left[\sum_{[v_l, v_{l+1}, \ldots, v_k, o] \in \mathcal{V}} \frac{\partial o}{\partial v_k} \prod_{i=l}^{k-1} \frac{\partial v_{i+1}}{\partial v_i} \right]}_{\text{Summan av alla vägar genom nätverket}} \frac{\partial v_l}{\partial c_{v_{l-1},v_l}}, \tag{19}$$

där L är förlustfunktionen, och o är det sista lagret som motsvarar nätverkets genererade värden. $c_{l-1,l}$ är en parameter i nätverket mellan lager $l-1$ och l och v_k motsvarar de steg som kan tas för samtliga vägar för noderna i lagret l där $\mathcal{V}$ är mängden av alla vägar mellan v_l och o [3].

2.8.3 Uppdatering av parametrar

I *bakåtpropagerings*-algoritmen är stegets magnitud en produkt av inlärningshastigheten och den partiella derivatan av förlustfunktionen med avseende på den individuella parametern alternativt en adaptiv modifiering av den. Detta beskrivs av

$$c_{v_{l-1},v_l} \leftarrow c_{v_{l-1},v_l} - \eta \frac{\partial L}{\partial c_{v_{l-1},v_l}}, \tag{20}$$

där η beskriver inlärningshastigheten för nätverket [3]. Termerna i ekvation (19) bestäms annorlunda beroende på val av nätverk samt optimeringsalgoritm.

2.8.4 Uppdatering för MLP

För klassiska MLPs kan termerna i ekvation (19) faktoriseras i flera separata bidrag, vilka definieras nedan.

- $\frac{\partial L}{\partial o}$: Partiell derivata av förlustfunktionen med avseende på utvärdena från nätverket. Denna bestäms av vilken förlustfunktion som väljs.
- $\frac{\partial o}{\partial v_k}$: Den partiella derivatan av utvärdet med avseende på lagret innan. Den partiella derivatan för varje sammanslutning kan faktoriseras i två termer. Den ena termen utgörs av parametern som är kopplad till sammanslutningen, medan den andra representerar den partiella derivatan av aktiveringsfunktionen som används i noden för utvärdet.
- $\frac{\partial v_i}{\partial v_{i+1}}$: Denna term liknar den föregående, men för sammanslutningar mellan noder i tidigare lager än det sista. Även denna kan faktoriseras i två termer likt ovan.

- $\frac{\partial v_l}{\partial c_{v_{l-1},v_l}}$: Denna term beskriver enbart parameterns inverkan på förändring i efterföljande lager. För ett neuralt nätverk motsvarar detta derivatan av aktiveringsfunktionen multiplicerat med indatan till parametrarna c_{v_{l-1},v_l} i nod v_{l-1}.

2.8.5 Uppdatering för KANs

Likt för MLPs kan KANs faktoriseras enligt ekvation (19). Bidragen definieras nedan.

- $\frac{\partial L}{\partial o}$: Likt MLPs, bestäms denna term av vilken förlustfunktion som väljs.
- $\frac{\partial o}{\partial v_k}$: För KANs är förändringen i sammanslutningen beroende av justerbara parametrar i linjärkombinationer med bestämda basfunktioner. Linjärkombinationen definierar sammanslutningens aktiveringsfunktion.
- $\frac{\partial v_i}{\partial v_{i+1}}$: Likt föregående beskrivs denna term av derivatan av linjärkombinationen som evalueras med indatan för sammanslutningen.
- $\frac{\partial v_l}{\partial c_{v_{l-1},v_l}}$: Denna term beskriver hur förändringen i kommande lager beror på parametrarna. Detta motsvarar basfunktionerna, i och med att varje term i linjärkombinationen är linjär som funktion av dess parameter. Därmed blir denna parameterspecifik, med dess motsvarande basfunktion evaluerat med indatan.

2.8.6 Optimeringsalgoritm

För att träna nätverken används en *optimerare*. Denna optimerare uppdaterar parametrarna iterativt. I detta arbete används primärt optimeringsalgoritmen *AdamW*, en utvecklad version av *Adam*-algoritmen. *Adam* beskriver en uppdateringsprocess där hänsyn tas till ett exponentiellt glidande medelvärde av gradienten samt den kvadrerade gradienten [12]. Stegets storlek är adaptivt och anpassas utifrån tidigare gradientinformation för att undvika för stora hopp. Steget definieras

$$c_{v_{l-1},v_l} \leftarrow c_{v_{l-1},v_l} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{21}$$

där $\hat{m}_t$ benämns som första moment, vilket är ett exponentiellt glidande medelvärde av tidigare gradienter. $\hat{v}_t$ benämns det andra momentet och är ett exponentiellt glidande medelvärde av gradienternas kvadrater. Parametern ϵ är ett mycket litet positivt tal för att undvika division med noll. Uppdateringen balanseras i riktning av det första momentet och i storlek av det andra momentet [12]. Optimeringsalgoritmen *AdamW* liknar *Adam* men använder sig även av en regulariseringsteknik; *viktregularisering* [13]. Denna teknik beskrivs i kommande avsnitt.

2.8.7 L2-regularisering och viktregularisering

Ett fenomen som är frekvent inom träningen av komplexa neurala nätverk är att modellen övertränas på datan. Modellen skapar så starka relationer till den tillgängliga datan att den presterar sämre på data från samma uppsättning som modellen inte har exponerats för under träningen. Parametrarnas magnituder är inte uniformt distribuerade över nätverket. En teknik för att undvika detta är *L2-regularisering*, som bygger på att vid *bakåtpropagering* bestraffa stora vikter genom att addera en term till deras förlustfunktion i ett lager. Förlustfunktion i lager l vid uppdatering av parametrar definieras då

$$L_w = L + k \sum_{i=0}^{d} c_{i,l}^2, \tag{22}$$

med parameter i i lager l, där k är en liten konstant. När parametrarna kvadreras bestraffas stora parametrar eftersom det leder till en större förlust [3]. För somliga optimeringsalgoritmer är denna ekvivalent med *viktregularisering*, men i fallet när *Adam*-algoritmen tillämpas skiljer den sig [13]. I stället för att tillämpa regulariseringstermen på förlustfunktionen, tillämpas den på steget. Denna term är inte involverad i beräknandet av de exponentiellt glidande medelvärdena. Detta är fördelaktigt eftersom regulariseringen inte påverkar den skalning som *Adam*-algoritmen utför, utan tillämpas separat direkt på vikterna i uppdateringssteget [13]. *Viktregularisering* tillämpas till *AdamW*-algoritmen genom parametern *weight_ decay*.

3 Metod

Metoden för detta projekt bygger primärt på att utnyttja teorin ovan för att konstruera nätverk av typen KAN och SKAN. Dessa kommer att jämföras för olika strukturer i *tre* experiment. Det första experimentet undersöker en funktionsregression av en simpel funktion, $f(x) = 4xy^2$, som approximeras med hjälp av tre olika sorters nätverk. I det andra experimentet presenteras en komplexare funktion; en bild vars gråskalevärden är en funktion av koordinaterna. Det tredje experimentet undersöker nätverkens prestation vid bildklassificering, där de jämförs med faltningsnätverk. Initialt beskrivs implementationen, för att sedan mynna ut i en beskrivning av de experiment och de krav som ställs på dem. Implementationen av KANs samt analys återfinns i appendix C.1, och likartat för SKANs återfinns i appendix C.2.

3.1 Implementation

Nätverken i experimenten byggdes i Python med biblioteket *PyTorch*. *AdamW*-algoritmen och förlustfunktionen MSE användes för bakåtpropagering. I vissa fall användes även en schematisk successivt minskande inlärningshastighet för stabilare träning. Nätverken tränades i ett bestämt antal epoker för att fastställa bra resultat och ta hänsyn till att nätverken kan behöva längre träning. Aktiveringsfunktionerna är konstruerade som en viktad summa av en bas-spline-parametriseringen och en Silu-funktion:

$$\phi(x) = w_1 \cdot \text{B-spline}(x) + w_2 \cdot \text{Silu}(x).$$

Bas-spline-parametrarna initierades slumpmässigt, normalfördelade kring 0 med standardavvikelse 0.1. Vikterna initieras som 1 för B-spline-parametriseringen och enligt Xavier-initialisering för Silu-funktionen, tillvägagångssättet följer det i [1]. Knutintervallen är förutbestämda och bestäms utifrån maximala absolutvärdet av aktiveringarna och är symmetriska kring 0 med längd marginellt större än två gånger den maximala aktiveringen.

3.1.1 KAN

Utgångspunkten för KAN är Kolmogorov-Arnolds representationssats. Parametrarna mellan två lager för linjärkombinationerna lagrades i en lista av tensorer. En linjärkombination av basfunktionerna och parametrarna utgör aktiveringsfunktionerna mellan samtliga par av noder två lager sinsemellan. Detta resulterar i att de tre dimensionerna i ordning motsvarar vilken nod i det första lagret det är, de parametrar som den aktiveringsfunktionen har och vilken nod i det andra lagret det är. Samtliga funktionsvärden av de variabler som förts till lagret beräknades sedan och lagrats i en matris. Denna matris satsmultiplicerades parametertensorn, vilket frambringade utvärdena för den funktionen. Koden som beräknar basfunktioner återfinns i appendix C.1.2. Algoritmen som valdes vid optimering var *AdamW*, vilket finns implementerat i *PyTorch*-biblioteket och beskrivs utförligt i avsnitt 2.8.

Enligt Kolmogorov-Arnolds representationssats är sambandet ett KAN som beskriver en sammansatt funktion. Eftersom att funktionerna inte är monotona och kan skilja sig avsevärt mellan två närliggande punkter, kan två variabelvärden nära varandra ge betydligt annorlunda funktionsvärden. Denna avvikelse ackumuleras för varje samband som appliceras på variabelvärdet. För att motverka potentiellt kaotiskt beteende som modellen kan uppvisa vid träning, användes två tekniker; att använda samma uppsättning basfunktioner för samtliga aktiveringsfunktioner och att träna nätverken försiktigt med låg inlärningshastighet på små satser av data. Detta motverkar stora gradienter och minimerar risken för kaotiskt beteende under träningen.

3.1.2 SKAN

Utöver KANs har ett flertal neurala nätverk baserat på Sprechers satser byggts och utformats. Till följd av undermålig prestanda har en del nätverk korrigerats med nya termer och parametrar för att förbättra nätverkens prestation och därmed divergerat en aning från de ursprungliga satsrepresentationerna. Modellerna utgår från att den inre funktionen är optimerbar för funktionen i fråga och inte en global inbäddning som skall fungera för alla funktioner, samma utgångspunkt som i [1].

Dessa nätverk implementeras genom att en tensor med förra lagrets utdata skapas och till den adderas translationen ϵq till varje kolumn utifrån vilket kolumnindex q i tensorn som motsvarar noden q i lagret. Lagrets aktiveringsfunktion appliceras elementvis på tensorn, sedan multipliceras $\lambda_{p,q}$ med matchande element på index p, q och efter adderas eventuell q-term, utifrån kolumnindex. Till sist adderas tensorn längst raderna och resulterande tensor är det lagrets utdata. Tensorns kolumner representerar noderna i lagret och raderna det tidigare lagrets noder, tensorns djup står för satsstorleken och innehåller alla sådana prov. Nätverken som baserats på ekvation (16)

fungerar som ovan förutom sista lagret som fungerar som ett vanligt KAN-lager, med en aktiveringsfunktion per nod. Där appliceras varje sådan elementvis per kolumn i tensorn. Parametrarna för aktiveringsfunktionerna, samt variablerna , ϵ sparas som optimerbara tensorer i modellerna.

Nätverken baseras på att de klassiska representationerna var bristfälliga. Det föranledde idén att implementera individuella $\lambda_{p,q}$ för varje anslutning av noder, likt vikter i ett FFNN. För att öka nätverkets flexibilitet ersattes också i vissa fall termen $\epsilon \cdot q$ med enskilt ϵ_q för varje nod istället. Nedan följer modellernas utformning utefter representation.

Modell	**Representation**	**Kommentar**
SprecherOrg	$\sum_q \chi \left[\sum_p \lambda^p \psi(x_p + \epsilon q) + q\right]$	Ursprunglig representation från [7], med endast ett λ.
SprecherMulti	$\sum_q \chi \left[\sum_p \lambda_p \psi(x_p + \epsilon q) + q\right]$	Representation enligt Sprechers modifiering.
SprecherBrown	$\sum_q \chi_q \left[\sum_p \lambda_p \psi(x_p + \epsilon q)\right]$	$\epsilon = \frac{1}{(2n+1)(2n+2)}$ från Browns modifieringar återgivet i [11].
SprecherSingle	$\sum_q \chi \left[\sum_p \lambda^{p \cdot q} \psi(x_p + \epsilon q)\right]$	Representation från [7], endast ett λ.
Sprecher96Pro	$\sum_q \chi_q \left[\sum_p \lambda_p \psi(x_p + \epsilon_q)\right]$	Representation utifrån 16 med generaliserat ϵ.
SprecherSpecial	$\sum_q \chi \left[\sum_p \lambda_p \psi(x_p + \epsilon_q) + q \cdot r\right]$	Sprechermulti med generaliserat ϵ och optimerbart r.
SprecherLambda	$\sum_q \chi \left[\sum_p \lambda_{p,q} \psi(x_p + \epsilon_q)\right]$	SprecherMulti med generaliserade λ och ϵ.
SprecherComb	$\sum_q \chi_q \left[\sum_p \lambda_{p,q} \psi(x_p + \epsilon_q)\right]$	Sprecher96Pro med generaliserade λ.

3.2 Datahantering och val av strukturer

För att jämföra KAN och SKAN med klassiska neurala nätverk krävs ett tydligt och enhetligt experiment, samt att samma data används för att träna samtliga nätverk. För att skapa ett resultat som inte är missvisande delades all data upp i två delar. Den ena delen användes till att träna modellen och den andra till att testa modellens prestation. För att maximera modellens generaliseringsförmåga valdes majoriteten av all data till träningsdelen, med restriktionen att testmängden var av storlek att kunna möjliggöra statistiskt tillförlitliga resultat. Till regressionsexperimentet användes 80 % av all data till att träna modellerna och 20 % till att testa modellerna. All data för funktionen genererades slumpmässigt för ett antal *seeds*.

För att skapa rättvisa resultat är nätverksstrukturen en viktig aspekt. Målet med konstruktion av nätverken var att vara så konsistent som möjligt för att möjliggöra en enhetlig analys, vilket leder till att det huvudsakliga målet är att nätverken har jämförbara parameterantal. Sekundärt prioriterades djupet, antalet lager som nätverket har, där det var eftersträvat att nätverken hade lika djup. Utöver detta var det viktigt att antalet noder i de gömda lagren, mellan det första och det sista lagret, var jämnt fördelade. Detta sprider ut noderna, vilket är viktigt för att inte alla betydande sammanslutningar ska koncentreras i ett lager.

3.3 Bakåtpropagering

För att optimera parametrarna krävs en algoritm som beskriver processen. I detta projekt implementerades den automatiskt differentierande *PyTorch*-funktionen. Bibliotekets algoritm baseras sig på implementationen i avsnitt 2.8. Den automatiska *PyTorch*-funktionen bygger en beräkningsgraf av tensorer. Beräkningsgrafen i sin helhet motsvarar samtliga operationer som verkar på tensorerna i nätverket. Modulen beräknar gradienter för samtliga steg, där dessa sparas i tensorernas datastruktur. För detta är det viktigt att inkludera argumentet *requires_grad = True* för samtliga tensorer som förändras i beräkningsgrafen, samt listan för modellens parametrar. När dessa gradienter sedan är beräknade görs en uppdatering enligt ekvation (20).

3.4 Experiment 1: Funktionsanalys

Målet med det första experimentet var att så rättvist som möjligt jämföra MLP med KAN och SKAN. I detta experiment användes slumpmässigt genererad data från funktionen i ekvation (23).

$$f(x) = 4xy^2, \quad x, y \in (-2, 2) \tag{23}$$

För att vara konsekvent med vilken data som användes samma data för samtliga nätverk. Denna data genererades för ett specifikt antal *seeds*, för *seed* noll till nio, där ett medelvärde av medelabsolutfel beräknades. Detta gjordes eftersom optimeringsalgoritmen riskerar att fastna i ett lokalt minimum av förlustfunktionen, vilket förhindrar fullständig optimering. Experimentet baserades på 10000 datapunkter, där nätverken hade olika strukturer, framtagna för ett specifikt parameterantal. Strukturerna som nätverken benämns med motsvarar antalet noder i samtliga lager av nätverken.

Det primära jämförelseobjektet var modellernas medelabsolutfel på all testdata. Felen jämfördes inom en typ av nätverk, samt mellan typerna i förhållande till antalet parametrar som nätverken har, då den relativa prestationen var den intressanta jämförelsen.

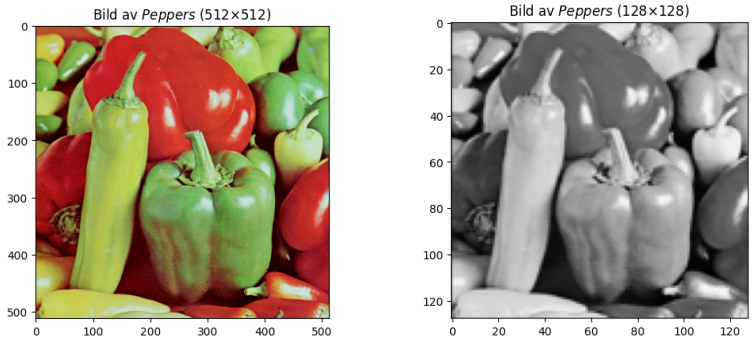
För att ställa upp en konsekvent analys av nätverken och minimera antalet felkällor som kan bidra till resultatet krävdes ett antal villkor. Villkoren definieras nedan.

- På grund av olika strukturer med olika konvergenshastighet tränas de över olika antal epoker.
- AdamW används som optimeringsalgoritm med $weight\_decay = 0.01$ och en schematisk inlärningshastighet som successivt minskades efter nätverket propagerat träningsdatan ett antal gånger.
- Nätverk av parameterantal omkring 50, 100, 200, 500 och 1000 parametrar jämförs. Exakt antal ska avvika så lite som möjligt.
- Tio *seeds*, samtliga mellan noll och nio, används till beräkning av MAE.
- Strukturer med två indata och en utdata används.

3.5 Experiment 2: Bildregression och rekonstruktion

Målet med det andra experimentet var att se hur väl de olika nätverken kan lära sig en mer komplex funktion genom bildregression; en gråskalebild. För att jämföra detta användes en klassisk maskininlärningsbild för att se hur väl de olika nätverken kan lära sig gråskalevärdena som en funktion av pixelkoordinater. För experimentet normaliserades både gråskalevärden och koordinater vid träning av modeller. Bilden som användes heter *Peppers* och är en klassisk testbild inom bildbehandling. Bilden kommer från databasen *USC-SIPI* [14]. Denna bild visas i figur 5. Till följd av att datamängden ökar kraftigt när färger används valdes bilden att hanteras i gråskala. Även granulariteten reduceras i syftet att krympa datamängden.

Den andra delen av experimentet är bildrekonstruktion, där en modell först tränades på bilden i figur 5b, vilken sedan användes till att generera bilder med 32, 64, 128, 256 och 512 pixlar per dimension, med syftet att undersöka modellens interpolering. Rekonstruktion från 64 pixlar per dimension till 512 utfördes av den bästa modellen. Samtliga rekonstruktioner till 512 pixlar per dimension jämfördes med resultatet från Adobe Photoshops bikubiska rekonstruktionsalgoritm. Samtliga pixlar normaliserades för att utföra detta experiment, med objektet att undersöka hur granulariteten i bilden kan förbättras utan att exponera bilden av den dimension för modellen.



(a) *Peppers* i full storlek, 512 pixlar i respektive dimension.

(b) *Peppers* nedskalad till 128 pixlar i respektive dimension i gråskala.

Figur 5: Orginal- och nedskalad bild av *Peppers*.

Jämförelseobjektet för detta experiment var huvudsakligen förlusten, samt de visuella resultaten. Med anledning av mer komplex funktion, valdes mer komplexa nätverk för att jämföra hur väl de kan lära sig funktionen som bilden svarar mot. Villkoren för experimentet definieras nedan.

- Nätverket tränas ett antal epoker, till den punkt att det inte uppnår en lägre förlust.
- AdamW används som optimeringsalgoritm med $weight\_decay = 0.01$ och en schematisk inlärningshastighet som minskas successivt efter antalet tränade epoker.
- Nätverken tränas på parameterantal nära 500, 1000, 2000, 5000 och 10000 med anledning av funktionens höga komplexitetsgrad. Parameterantal ska avvika minimalt.
- Endast en *seed* används för att träna nätverken, eftersom en bild ska genereras.

3.6 Experiment 3: Bildklassificering

Syftet med det tredje experimentet var att jämföra hur KAN presterar på bildklassificering, samt att jämföra dess prestanda med faltningsnätverk, som är speciellt konstruerade för att extrahera egenskaper ur lokala regioner i bilder. Det stöd som finns för att KANs kan vara effektiva är att varje aktiveringsfunktion kan tränas för att skapa ett samband som beskriver flera indata i en lokal region. Dessa behöver inte vara monotont växande likt för MLPs, vilket i kombination med interpoleringar resulterar i en liknande detektion av lokala egenskaper.

Det dataset som användes heter *Pneumonia MNIST* och består av röntgenbilder för patienter med lunginflammation. Bilderna är i gråskala med 28 pixlar per dimension, med objekt att prediktera om patienten har lunginflammation eller inte. Till varje bild hör en etikett som indikerar detta. Tränings- och testdel av dataset består sammanlagt av 5332 bilder, varav 4708 är träningsfall och 624 är testfall. Datasetet består av 1640 friska fall och 4216 drabbade fall, vilket indikerar att det är obalanserat. Detta kompenserades för genom att generera en större förlustgradient för den underrepresenterade klassen av datasetet. Därmed gjordes träningen opartisk oavsett klassernas storlek. En uppsättning av fyra bilder ur dataset, med deras respektive etikett, visas i figur 6.

Figur 6: Uppsättning av fyra bilder ut datasetet *Pneumonia MNIST* med deras respektive etikett. I andra och tredje bilden visas drabbade patienter, medan friska patienter demonstreras i första och fjärde bilden.

I detta experiment användes sju parametrar per basfunktion av anledningen att detta lämpar sig väl för att få ett parameterantal likt dem som önskas att jämföras. I detta experiment är antal parametrar per basfunktion endast det som justerades, eftersom att lägga till ett gömt lager inte kommer bidra till ett rättvist resultat. Detta är eftersom att ett fåtal aktiveringsfunktioner måste lära sig all information på ett fåtal parametrar, vilket resulterar i att träffsäkerheten blir avsevärt sämre. Därav valdes det istället att använda fler basfunktioner per aktiveringsfunktion och att bara ha ett lager av sammanslutningar för att optimera verkningsgraden av parametrarna.

4 Resultat

Resultatet från applicering av metoden var att SprecherComb presterade bäst när en enkel funktion testades. Under bildregressionen hade SprecherLambda bäst prestanda. Specifikt för regressioner har SKAN-varianterna visat att de har god potential att lära sig mycket komplexa samband med ett sparsamt antal parametrar. Under klassificeringen presterade faltningsnätverken bäst.

4.1 Experiment 1: Funktionsanalys

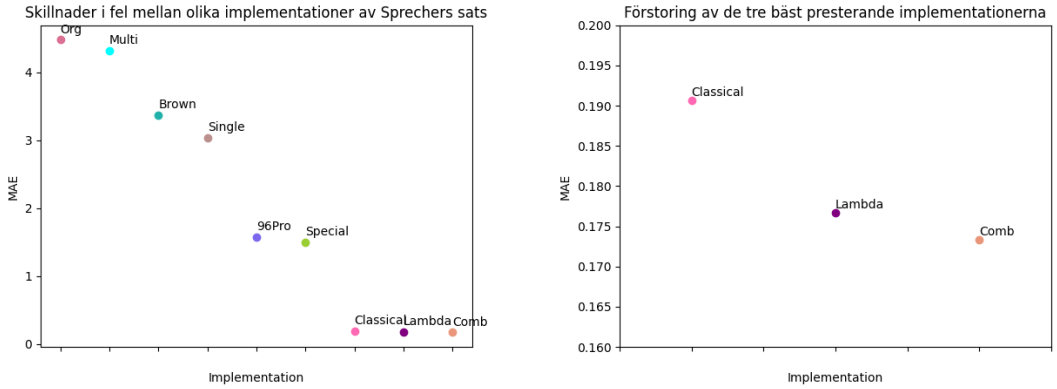
Nedan följer resultaten som producerats under test av nätverken med data av funktionen i ekvation (23). Strukturen som presenteras i tabellen indikerar hur nätverket är konstruerat, med antalet noder i respektive lager. Medelabsolutfelet avtar med antalet parametrar i nätverket.

Resultatet för KAN som tränats 500 epoker med satsstorlek 250 presenteras i tabell 2.

Tabell 2: MAE för KAN för funktionen $f(x) = 4xy^2$.

Nätverk	Struktur	Parametrar per aktiveringsfunktion	Antal parametrar	MAE
1	2-2-1	8	48	0.1779
2	2-4-1	8	96	0.1016
3	2-5-1	8	120	0.0892
4	2-3-5-1	8	208	0.0938
5	2-5-9-1	8	512	0.0677
6	2-10-10-1	8	968	**0.0586**
7	2-7-6-1	16	992	0.0749

Nedan syns skillnaderna i resultat för olika implementationer av SKAN-nätverk, där alla modeller har testats på ett $2-5-5-1$ nätverk med 16 parametrar per basfunktion. Figur 7 visar tydligare skillnaden på de tre bäst presterande modellerna, där "Classiscal"är ett KAN-nätverk.



(a) Skillnader i prestanda mellan de olika implementationerna av Sprechers sats.

Förtydligande av skillnaderna mellan de tre bäst presterande modellerna.

Figur 7: Översikt över modellprestanda.

Eftersom Lambda och Comb är de två modellerna som presterar bäst, är det dessa två som testas i tabellerna nedan. Varje struktur tränas över 200 epoker.

Tabell 3: MAE för SprecherLambda för funktionen $f(x) = 4xy^2$.

Nätverk	Struktur	Parametrar per aktiveringsfunktion	Antal parametrar	MAE
1	2-5-1	6	31	0.1113
2	2-9-1	6	49	0.0950
3	2-7-3-1	16	99	0.1879
4	2-11-9-1	16	195	0.0524
5	2-15-12-11-1	26	492	0.0289
6	2-15-20-24-1	36	1021	0.0304
7	2-25-20-15-1	26	1022	**0.0261**

Tabell 4: MAE för SprecherComb för funktionen $f(x) = 4xy^2$.

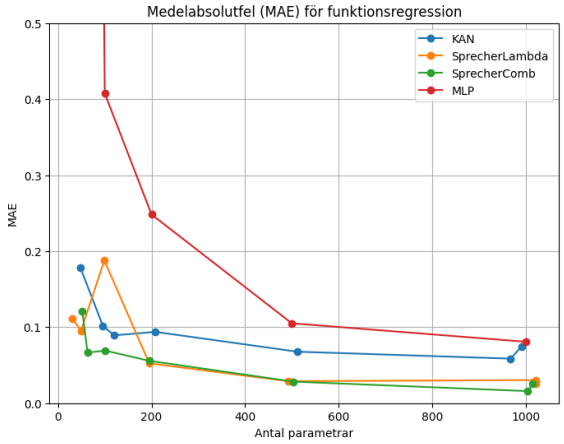
Nätverk	Struktur	Parametrar per aktiveringsfunktion	Antal parametrar	MAE
1	2-4-1	6	52	0.1206
2	2-5-1	6	63	0.0666
3	2-7-4-1	6	101	0.0691
4	2-5-6-1	16	195	0.0557
5	2-12-9-7-1	26	503	0.0284
6	2-13-14-16-1	26	1005	**0.0159**
7	2-10-11-15-1	36	1015	0.0253

Tabell 5 presenterar resultatet för MLPs.

Tabell 5: MAE för MLP som tränats över 500 epoker med satsstorlek 250 för funktionen $f(x) = 4xy^2$.

Nätverk	Struktur	Antal parametrar	MAE
1	2-3-4-4-1	50	3.6397
2	2-5-6-6-1	100	0.4074
3	2-9-10-6-1	200	0.2482
4	2-15-16-11-1	500	0.1051
5	2-15-27-18-1	1000	**0.0808**

Sammantaget bildar de resulterande medelabsolutfelen i tabellerna jämförelsen i figur 8.



Figur 8: Graf över medelabsolutfelen för alla modeller som testats på funktionen $4xy^2$.

4.2 Experiment 2: Bildregression och rekonstruktion

I detta experiment presenteras en mer komplex funktion, bilden *Peppers* i figur 5b, vars pixelns gråskaleväde beskrivs som en funktion av pixels koordinat.

Bildregression Tabell 6 visar KANs prestation på bildregressionen. Dessa har tränats över 500 epoker med satsstorlek 250.

Tabell 6: MAE för KAN för bilden *Peppers*.

Nätverk	Struktur	Antal parametrar	MAE
1	2-5-4-6-1	480	0.0867
2	2-8-8-5-1	1000	0.0652
3	2-10-11-10-1	2000	0.0514
4	2-10-20-13-10-1	4960	0.0411
5	2-20-20-20-20-1	9948	**0.0346**

Tabell 7 visar resultatet för bildregressionen med olika SprecherLambda modeller. Alla modellerna har 16 parametrar per basfunktion.

Tabell 7: MAE för SprecherLambda för bilden *Peppers*.

Nätverk	Struktur	Antal parametrar	MAE
1	2-19-20-1	511	0.0585
2	2-29-29-1	1011	0.0498
3	2-30-30-30-1	2022	0.0370
4	2-40-40-40-40-1	5090	0.0328
5	2-50-50-50-50-40-1	9948	**0.0196**

Tabell 8 visar resultatet för bildregressionen med olika SprecherComb modeller. Alla modellerna har 16 parametrar per basfunktion.

Tabell 8: MAE för SprecherComb för bilden *Peppers*.

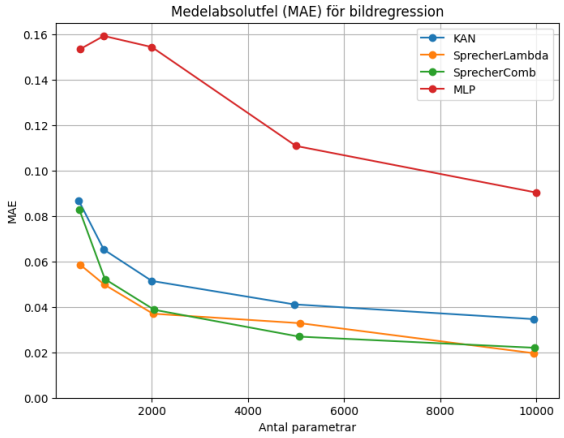
Nätverk	Struktur	Antal parametrar	MAE
1	2-13-13-1	491	0.0828
2	2-17-17-17-1	1024	0.0522
3	2-22-22-22-22-1	2052	0.0387
4	2-37-37-37-37-1	5067	0.0269
5	2-50-50-50-50-30-1	9960	**0.0220**

Resultaten för bildregression för MLP demonstreras i tabell 9. Dessa nätverk har tränats över 500 epoker med satsstorlek 250.

Tabell 9: MAE för MLP för bilden *Peppers*.

Nätverk	Struktur	Antal parametrar	MAE
1	2-9-10-6-1	500	0.1533
2	2-15-16-11-1	1000	0.1592
3	2-15-27-18-1	2000	0.1543
4	2-25-36-18-1	5000	0.1108
5	2-56-83-60-1	10000	**0.0903**

Slutligen visar figur 9 skillnaden i fel mellan de olika testade modellerna. De indikerar ett övervägande lågt medelabsolutfel för SKAN-varianterna, medan KANs har aningen högre, följt av MLPs som har nästan dubbelt så stort medelabsolutfel.



Figur 9: Jämförelse av medelabsolutfel för bildregression av bilden *Peppers*. Figuren speglar en avtagande förlust när antalet parametrar ökar.

Rekonstruktion För rekonstruktionen finns resultat från Adobe Photoshop med bikubisk rekonstruktionsalgoritm, och jämförelse med de andra modellerna finns i figur 17 och 18. Samtliga figurer återfinns i appendix A.2.

- KAN-modellen har struktur 2-41-50-43-20-1, 7 parametrar per basfunktion och består av 36134 parametrar totalt, se resultat i figur 14. Det slutliga MAE vid rekonstruktionen var 0.0442.
- SprecherLambdas resultat för en 2-200-100-80-60-40-1 modell med 36188 parametrar totalt finns i figur 15 och 18, med slutligt MAE på 0.0182. Alla modellerna har 16 parametrar per basfunktion.
- SprecherCombs resultat på en 2-200-100-80-60-30-1 modell med 36100 parametrar totalt finns i figur 16, med slutligt MAE på 0.0203. Alla modellerna har 16 parametrar per basfunktion.

4.3 Experiment 3: Bildklassificering

På grund av kraftigt varierande resultat över epokerna som nätverken tränats på har två olika resultat presenterats. Modellen testades under körningen, där resultaten oscillerade kraftigt mot slutet. Därav har det bästa testresultatet samt den avslutande klassificeringsträffsäkerheten noterats. Samtliga nätverk har tränats över 100 epoker med satsstorlek 250. Antal basfunktioner är angivet i hakparenteser efter strukturen i samtliga tabeller.

Resultaten för klassificering av KAN-nätverken presenteras i tabell 10. KAN-nätverken konvergerade snabbt på endast ett fåtal epoker. Se även figurer 19, 20 och 21 i appendix.

Tabell 10: Träffsäkerhet för klassificering av *Pneumonia MNIST*. I tabellen har nätverken samma struktur, men olika antal basfunktioner per aktiveringsfunktion.

Nätverk	Struktur	Parametrar	Bästa träffsäkerhet	Sista träffsäkerhet
1	784-1 [7]	5488	**85.90%**	85.10%
2	784-1 [13]	10192	82.85%	82.37%
3	784-1 [19]	14896	83.49%	82.85%

Resultaten för klassificering av SprecherLambda visas i tabell 11. Se även figurer 25, 26 och 27 i appendix.

Tabell 11: Träffsäkerhet av SKAN för klassificering av *Pneumonia MNIST*.

Nätverk	Struktur	Parametrar	Bästa träffsäkerhet	Sista träffsäkerhet
1	784-7-1 [16]	5531	87.18%	87.18%
2	784-10-50-50 -1 [16]	11022	88.78%	88.46%
3	784-15-60-60-1 [16]	16457	**89.58%**	89.58%

Resultaten för klassificering av SprecherComb visas i tabell 12. Se även figurer 28, 29 och 30 i appendix.

Tabell 12: Träffsäkerhet av SKAN för klassificering av *Pneumonia MNIST*.

Nätverk	Struktur	Parametrar	Bästa träffsäkerhet	Sista träffsäkerhet
1	784-7-1 [16]	5639	86.54%	85.10%
2	784-10-40-40 -1 [16]	11604	87.66%	86.70%
3	784-15-60-50 -1 [16]	16739	**90.06%**	87.66%

Resultaten för klassificeringen med faltningsnätverk presenteras i tabell 13. Se även figurer 22, 23 och 24 i appendix.

Tabell 13: Träffsäkerhet av faltningsnätverk för klassificering av *Pneumonia MNIST*. Nätverksstrukturerna benämner faltningslagerna som C med storleken på deras filter intill. Efter lagren beskrivs antal filter som appliceras på det lager. Dessa är slutligen sammanslutna med en MLP som tränas på egenskaperna som extraherats av faltningsdelen. Detta MLP går bara till en nod som predikterar etiketten för bilden. MLP-delen består endast av ett lager.

Nätverk	Struktur	Parametrar	Bästa träffsäkerhet	Sista träffsäkerhet
1	C5–32-C3–16–MLP	5857	87.50%	86.38%
2	C5–32–C3–32–MLP	10881	88.46%	86.38%
3	C5–48–C3–32–MLP	15905	**88.94%**	85.58%

5 Diskussion

Både KAN- och SKAN-nätverk god prestanda i experimenten, primärt i regressionsuppgifter. De har möjlighet att lära sig komplexa funktioner med låga fel på bekostnad av högre beräkningskapacitet. De utgör ett alternativ till klassiska MLPs. Gällande klassificering har KAN underpresterat gentemot faltningsnätverk, däremot presterar SKAN likvärdigt. En fördel för faltningsnätverkens över Kolmogorov-nätverken är att de har kortare träningstid.

5.1 KAN

Resultatet för det första experimentet presenteras i tabell 2. I det första experimentet tränades KAN-nätverken till ett lägre medelabsolutfel för testdata än vad MLP-nätverken lyckades producera. Det bästa medelabsolutfelet som presenteras är 0.0586, vilket kan jämföras med MLP-nätverkets motsvarade 0.0808 samt SKAN-varianterna som presenterar medelabsolutfel av storlek 0.0261 respektive 0.0159. De regressioner som skapas, beskrivna i avsnitt 2.3, är mer robusta än de MLP-nätverken lyckas producera. Detta är primärt för att KAN-nätverken har en intern frihetsgrad mellan noderna. Flera icke-linjära funktioner appliceras på variabelvärden för att sedan summera upp resultatet, vilket skiljer sig från MLP-nätverken vars variabelvärden summeras med dess vikter, för att sedan passera genom en aktiveringsfunktion. När olika funktioner appliceras finns det en större frihet i respons, samt att små förändringar i variabelvärde kan leda till stora variationer i funktionsvärdet. Detta demonstrerar även att approximationer av envariabelfunktioner är lättare att konstruera än de för flervariabelfunktioner, vilket presenteras i avsnitt 1 som den potential som finns för KAN-nätverk. Därav kan KAN-nätverken skapa precisa samband som beskriver funktionen på en större domän till ett sparsamt antal parametrar. KAN-nätverken underpresterar dock gentemot SKAN-varianterna, vilket indikerar den goda potential som finns i SKAN-nätverken och demonstrerar deras höga användningsgrad av dess parametrar. De två sista nätverken visar att under rådande omständigheter, ger åtta parametrar per aktiveringsfunktion bättre resultat än sexton. I sin tur indikerar detta att en mer spridd fördelning av parametrar över fler aktiveringsfunktioner

kan vara fördelaktig, då båda nätverk har omkring 1000 parametrar. Till följd av teorin i avsnitt 2.1 presenteras resultatet att fler funktioner med färre parametrar bättre kan approximera en funktion. Det här är primärt en konsekvens av att aktiveringsfunktionerna innehåller mycket information runt knutpunkterna mellan basfunktioner, där ett bättre icke-linjärt regressionssamband skapas. Det är även en konsekvens av att fler aktiveringsfunktioner kan skapa en större värdemängd av en begränsad definitionsmängd.

Resultaten för bildregressionen av KAN-nätverken presenteras i tabell 6, där samtliga överpresterade motsvarande MLP. Det bästa medelabsolutfel som presenteras för bilden är 0.0346, varpå SKAN-varianterna av motsvarande parameterantal presenterar ett lägre medelabsolutfel som är 0.0196 respektive 0.0220. Detta indikerar liknande resultat till vad det tidigare experimentet presenterar. Resultatet demonstrerar, likt tidigare experiment, att SKAN-varianterna har en större potential relativt parameterantalet när det gäller funktionsregressioner, men att KAN-nätverk fortfarande överpresterar gentemot MLP-nätverk. Den bild som nätverken tränas på är en mer komplex funktion, vilket resulterar i att större nätverk behöver tränas för att medelabsolutfelet ska bli lågt. För detta experiment blir dock skillnaden mellan KAN och motsvarande MLP större, vilket åter igen påvisar resultatet att mer precisa samband kan skapas när flera funktioner av en variabel summeras i flera led jämfört med ett MLP som försöker approximera en funktion av flera variabler. Detta blir specifikt tydligt när komplexiteten på den analyserade funktionen ökar. Modellen kan approximera en större värdemängd, beskriven av ett mer komplext mönster än en simpel funktion, med ett lägre fel. Detta knyter tillbaka till syftet i avsnitt 1.1, där en tydlig potential kan påvisas i dess förmåga att approximera komplexa mönster.

Bildrekonstruktion som presenteras i figur 14 samt 17 indikerar att KAN-nätverken har en viss potential, men inte bättre än SKAN-varianterna. Detta stämmer överens med att de inte tränats till lika låga medelabsolutfel. De interpolerar inte lika väl och misslyckas att skapa fina konturer i bilden, vilket slutligen presenterar en sämre rekonstruktion än vad den bikubiska rekonstruktionsalgoritmen i Adobe Photoshop gör. De har potential, men med komplikationen att dessa nätverk kräver stor beräkningskapacitet i jämförelse med Photoshop, resulterar i att de inte är fördelaktiga för detta ändamål.

Resultatet av klassificeringen är undermåligt i jämförelse med det som presenteras för faltningsnätverken. Detta indikerar, likt beskrivet i avsnitt 2.1, att KAN-nätverken huvudsakligen skapar regressiva samband, i detta fall av gråskalevärden och inte av de egenskaper som finns i bilden likt faltningsnätverk gör. Egenskaper i lokala regioner kan inte extraheras likt motiveringen i avsnitt 3.6. Som presenteras i avsnitt 1.1 var ett av huvudsyftena att undersöka nätverkens prestation för bildanalys, där detta experiment indikerar en underprestation relativt faltningsnätverk. KAN-nätverkens bästa träffsäkerhet var 85.90%, vilket är kan jämföras med faltningsnätverkets motsvarande 88.94%. Ett intressant resultat är att KAN-nätverken har det högsta resultatet för det nätverket med minst parametrar och att resultaten oscillerar, vilket är en stark indikation på att nätverken inte blir bättre på att modellera lokala egenskaper när antalet parametrar ökar, utan snarare överträner på bildernas pixlar. En fördel med de KAN-nätverk som konstruerats är att dem konvergerar fort, vilket är en konsekvens av att parametrarna endast är utspridda över ett lager av sammanslutningar. Inga kaotiska träningsbeteenden har möjlighet uppstå i samma utsträckning eftersom att fel inte ackumuleras över flera lager.

En markant skillnad för KAN-nätverk är träningen, där den primära orsaken är att små initiala fel ackumuleras över aktiveringsfunktioner när data propageras genom nätverket. Nätverken agerar oförutsägbart i den mån att de uppvisar ett kaotisk beteende och är känsliga mot små förändringar i intialvärden, vilket är en egenskap som MLP-nätverk inte uppvisar. Detta stämmer överrens med avsnitt 2.1, där aktiveringsfunktioner i flera led kan skapa en större värdemängd i sin helhet. Detta är positivt för dess möjlighet att kunna beskriva komplexa samband, men negativt då dess träningstid är längre. MLP-nätverk använder istället en monoton aktiveringsfunktionen som tillämpas på en summa av produkter av variabelvärden och parametrar, vilket undviker detta kaotiska fenomen. För KAN-nätverken medför detta fenomen att en liten inlärningshastighet behöver tillämpas, primärt mot slutet av träningen. Därav är det lämpligt att använda en schematisk avtagande inlärningshastighet för att reducera träningstiden. En åtgärd som togs för denna instabilitet var att fastställa domänen som basfunktionerna skapas på och hålla basfunktionerna konsekventa genom hela träningen, till skillnad från att göra dem adaptiva baserat på indata. Detta reducerar det kaotiska beteendet under träningen och ger nätverket en möjlighet att kunna konvergera. För tillfället är algoritmen ineffektiv i den mån att den hämtar basfunktionerna, som rekursivt beräknas, evaluerat i indata från ett annat skript för varje aktiveringsfunktion, vilket i kombination med att låg inlärningshastighet behöver tillämpas ger en långsam träning. Detta är dock snarare en fråga om algoritmeffektivisering.

5.2 SKAN

Resultatanalys
Likt KAN, visar Sprechernätverken hög prestanda i det första experimentet. SKAN överträffar de andra två nätverken med fel mindre än hälften så stora än KAN, se figur 8. En möjlig förklaring är att

nätverken generaliserar enklare och bättre med endast en lärbar aktiveringsfunktion och att många aktiveringsfunktioner kan göra nätverken för specifika som medför försämrad prediktion. Prestandan varierar något, det inte är säkert att ett nätverk med fler parametrar ger lägre fel. En anledning till variationen kan vara att felen redan är mycket små och modellerna bra, så slumpmässighet får en större inverkan på resultaten. Att modellerna varierar i struktur och antal basfunktioner kan också variera prestandan. Resultaten visar däremot generellt att prestandan ökar med fler parametrar, se figur 8. SprecherComb presterar aningen bättre än SprecherLambda i funktionsregressionen, vilket visar att det är av större vikt att modellen har individuella aktiveringsfunktioner i sista lagret än bredare och djupare lager. Resultaten tyder också på att fler parametrar per basfunktion inte är bättre efter en viss gräns, vilket också genomgående noterats genom arbetets gång. Nätverken får antagligen det svårt att generalisera när funktionen har många parametrar.

Vidare, visar resultaten från bildregressionen att SKAN klarar av att representera ytterst komplexa funktioner. Bilderna från båda modellerna blir bättre med fler parametrar och bilderna blir tydliga, med liten förlust av detaljer. Det visar att transformationer genom inre translation och yttre skalning av samma aktiveringsfunktion är tillräckligt även för att fånga svårare samband. Platta och avtagande strukturer har använts och analyserat därför att de uppvisat högre prestanda. Att de strukturerna är bättre kan vara till följd av att ett lagers λ- vikter kan ge mer information till nästkommande lager. Om det finns fler λ-vikter per nod till nästa lager till följd av avtagande struktur, kan det leda till att varje lager enklare kan tyda mer komplexa samband. SprecherLambda presterar bättre än SprecherComb i bildregressionen och interpoleringen, vilket indikerar att nätverket klarar sig bättre utan extra aktiveringsfunktioner på slutet i denna typ av uppgifter.

Även interpoleringen av SprecherLambda är god då bilden behåller struktur, former och får en mjukare visuell framtoning. Med lågt fel har modellen lärt sig bilden väl och med aktiveringsfunktioner för varje indata interpolerar nätverket enklare. Det sker eftersom intermediära värden följer aktiveringsfunktionernas värden, vilka beskriver sambandet i bilden. Jämförelsevis är denna interpolering likvärdig med den som Adobe Photoshop producerat då SprecherLambda har mjukare former och är mindre grynig, medan Adobe producerar bättre detaljer, speciellt vid paprikornas toppar, se figur 17. I figur 18 där en interpolering med faktor 8 demonstreras, överpresterar SprecherLambda Adobe Photoshop. Anledningen till att SprecherLambda överträffar Adobe Photoshop är sannolikt för att modellen lärt sig bildens struktur, medan Adobe Photoshop endast rekonstruerar från närliggande pixlar och beaktar endast den lokala strukturen på bilden [15]. Vid interpolering med faktor åtta krävs ännu mer kunskap kring hela strukturen på bilden, eftersom gränsande pixlar inte ger tillräckligt med information och därför lyckas SprecherLambda bättre. SprecherLambda misslyckas däremot med formen på den främre paprikan och det indikerar att nätverket generaliserar mycket och missar detaljer. SprecherLambda tar också längre tid att träna och sedan rekonstruera.

Likaså, visar SKAN-modellerna hög kvalitet i klassifikationsuppgiften. Resultaten från klassificeringen fastslår att SprecherComb generellt presterar på en något lägre nivå än SprecherLambda. Det tyder på att det är en begränsning att ha fler aktiveringsfunktioner i sista lagret, även om dessa utgör en liten del av antalet parametrar. Möjligtvis minskar det nätverkens generaliseringsförmåga på grund av att för många funktioner och parametrar är mellan de sista lagren. I första experimentet var funktionen enkel och behovet av generalisering lägre. SKAN-modellerna visar hög kvalitet även i jämförelse med faltningsnätverk, trots att de inte lär sig någon spatial information, vilket framhäver nätverkens styrka. Resultaten indikerar också att SKAN överträffar KAN, vilket sannolikt beror på att SKAN kan ha större strukturer per parameterantal och därmed bättre inlärningsförmåga. Djupare strukturer är antagligen fördelaktigt eftersom det underlättar nätverket kan lära sig de komplexa sambanden och de svårare testfallen.

De goda resultaten, både i jämförelse med KAN och MLPs underbygger tesen att det finns sammanhang, många och varierande, då (S)KAN-nätverk överträffar konventionella nätverksstrukturer. SKAN kan urskilja och representera bilder effektivt och har högre noggrannhet än både KAN och MLPs per parameterantal. En osäkerhet är valet av knutintervall i metoden, det finns en liten risk att aktiveringar hamnar utanför intervallen, vilket kan försämra träningen och ge sämre resultat. Framtida forskning på normaliseringsmetoder är därför motiverat.

Nätverksstruktur och representation

SKAN kan ses som en hybridversion av FFNN och KAN. SKAN har anpassningsbara aktiveringsfunktioner som i KAN, samt vikter och translationer likt det i ett konventionellt FNN. Skillnaden i SKAN är att vikterna appliceras efter aktiveringsfunktionen och fungerar därmed inte som vikt på tidigare lagers aktivering, utan som en skalning på aktiveringsfunktionen. En individuell lärbar aktiveringsfunktion ger större frihet och dynamik för nätverket att lära sig. Denna kombination av båda nätverk har visat sig framgångsrik, då den utnyttjar båda nätverkens styrkor på ett parameter-effektivt sätt. SKAN har lägre intern frihetsgrad än KAN eftersom aktiveringsfunktionen mellan ett par av noder endast kan skalas och tranlatera indatan. Det kan innebära att nätverket interpolerar eller skalar med parametrar värre jämfört med KAN. Däremot visar inte resultaten det, se exempelvis figur 9, men något som framtida studier möjligen kan analysera. En nackdel med SKAN är att en del av tolkningsförmågan från vanliga KAN förloras i och med introduktionen av vikterna.

SKANs sammansättning visar att det är bättre att aktiveringsfunktionerna är olika transfor-

mationer av samma funktion istället för helt olika. Det använder mycket färre parametrar och bevisligen räcker det för att fånga komplexa samband. Det innebär att skalning till större nätverk är mer genomförbart. Resultaten tyder på att SKAN kan kompensera uttrycksfriheten av fria funktioner med fler anslutningar och noder, vilket enligt resultat också överväger den fördelen.

Fortsättningsvis, konstateras att utan individuella λ lyckas inte nätverken med funktionsuppskattning, se figur 7. Det kan bero på att de har en fraktal natur enligt teorin, som inte går att representera eller optimera för i den metod som använts, eftersom de är konstanta med derivata noll överallt. Om λ däremot görs individuell för varje anslutning bryts denna relation när också satsens representation bryts. Generaliseringen i metoden av den inre translationen ϵq till ϵ_q gör också att varje nod blir oberoende av de andra. Det gör att varje nod kan få en aktiveringsfunktion som varierar mer än vad som annars varit möjligt, vilket kan förklara större tolkningsförmåga och bättre resultat.

Följaktligen, följer frågan om varför nätverken med individuella λ fungerar, och om och varför de bryter den potentiella fraktala naturen av de inre funktionerna. Anledningen till att nätverken med individuella $\lambda_{p,q}$ fungerar till skillnad från de strikt enligt satsen, är antagligen att introduktionen av fler parametrar $\lambda_{p,q}$ gör att aktiveringsfunktionen ψ inte fullständigt behöver beskriva alla möjliga indata och attribut. Sannolikt behöver ψ vara pseudo-fraktal för att ha förmågan att innefatta all information som behövs för att representera en funktion. Med introduktionen av nya parametrar tas en del av den bördan bort och därmed kan den fraktala strukturen brytas och det blir möjligt för ψ att vara en snäll kontinuerlig funktion som nätverken kan uppskatta med bas-splines.

Med prestandan i åtanke, uppstår frågan om de modifierade representationerna är giltiga. De modifieringar som gjorts har endast varit avslappnande, vilket medför att det inte begränsar parametrarna till att vara identiska med de i satserna. Det är möjligt att $\epsilon_q = \epsilon \cdot q \, \forall q$. På samma sätt kan $\lambda_{p,q} = \lambda_p \, \forall q$. Därmed kan de modifierade representationerna garanterat representera alla kontinuerliga funktioner på en begränsad mängd, men sannolikt också på fler sätt. Det säkerställer modellernas tillämpbarhet på diverse problem. SprecherComb bygger på avslappningen av Sprecher 1996 och SprecherLambda på Sprechers formulering av KAT utan yttre q-termer, se nedan.

$$\sum_{0 \leq q \leq 2n} \chi \left[\sum_{1 \leq p \leq n} \lambda_{p,q} \psi(x_p + \epsilon_q) \right] \tag{24}$$

$$\sum_{0 \leq q \leq 2n} \chi_q \left[\sum_{1 \leq p \leq n} \lambda_{p,q} \psi(x_p + \epsilon_q) \right] \tag{25}$$

5.3 MLP och faltningsnätverk

I experimenten har MLP- och faltningsnätverk använts som utgångspunkt för analys. Överlag har MLP-nätverken ett undermåligt resultat i regressionsexperimenten, vilket är av anledningen att KAN-nätverken kan skapa jämnare interpoleringar mellan datapunkter. Aktiveringsfunktionerna är funktioner av flera variabler, där parametrarna används för att modifiera dem i motsats till klassiska MLP-nätverk som använder parametrarna för skalning av variabelvärdet. Därmed blir regressionerna mindre precisa. MLP-nätverken har inte interna frihetsgrader i aktiveringsfunktionerna likt KAN-nätverken och SKAN-varianterna. Medelabsolutfelet som MLP-nätverken lyckas producera är som bäst för funktionsregressionen 0.0808 och för bildregressionen 0.0903, vilket är avsevärt högre än motsvarande fel för KAN och SKAN-varianterna.

Faltningsnätverk har ett bättre resultat än KAN-varianterna vid klassificering. Primärt intressant är den höga träffsäkerheten som det faltningsnätverk med 15905 parametrar har på 88.94%. Det var inte den bästa träffsäkerheten, men överlag har dem högre träffsäkerhet än KAN- och SKAN-varianter av motsvarande parameterantal. Dess goda prestanda härstammar från att definitionsmängden som neurala nätverket tränas på består av bildens egenskaper i lokala regioner istället för gråskalevärden. Egenskaperna är extraherade genom faltningslagrena, likt beskrivet i avsnitt 2.7 En huvudkomponent i klassificering är att kunna extrahera egenskaper ur bilder i lokala områden av bilden, vilket KAN-nätverken inte specifikt tränas på. Därav klassificerar faltningsnätverk med högre träffsäkerhet. Dessa är även relativt snabba att träna, vilket gör dem fördelaktiga modeller att använda och skala upp.

Faltningsnätverken som valts är konstruerade för att koncentrera majoriteten av parametrarna i faltningskomponenten, vilket är en felkälla som kan generera missvisande resultat. Med objektet att maximera prestandan är detta inte det mest optimala och struktur hade kunnat modifieras för att uppnå högre träffsäkerhet. Detta är en möjlig anledning till att SKAN-nätverket uppvisar en högre träffsäkerhet för det största nätverket.

6 Samhälleliga och etiska aspekter

Med konstruktionen av neurala nätverk uppstår etiska risker, primärt hur modellerna används vid generering av bilder och text. Huvudaspekten att ta hänsyn till är att inget material som kan anses kränkande, känsligt eller oetiskt får genereras eller användas för att träna modellerna. De huvudsakliga åtgärderna som tagits för detta arbete är att använda bilder som inte omfattas av någon av kriterierna ovan. Bilden *Peppers* och datasetet *Pneumonia MNIST* är de primära datakällorna som används för att träna modeller i detta projekt. Dessa har valts med noggrannhet för att varken exponera känslig information eller producera kränkande material.

Artificiell intelligens är för närvarande ett mycket aktuellt och omdiskuterat ämne. Detta arbete syftar till att undersöka alternativa tekniker till klassiska neurala nätverk. Med utveckling kan bättre modeller bli mer integrerat i samhället, vilket kommer med både för- och nackdelar. Nya teknologier som förenklar människors liv kan utvecklas, men ibland på bekostnad av att människor förlorar sina jobb.

7 Slutsats

Sammanfattningsvis kan det sägas att KAN och SKAN-varianter har ett övervägande gott resultat, där de är särskilt kraftfulla vid regressioner. Att utnyttja möjligheten att skapa träningsbara aktiveringsfunktioner fungerar mycket väl för att skapa samband som interpolerar och kompletterar saknade data. En stark potential uppvisades av SKAN-varianterna för rekonstruktion av bilder vid förfining av granularitet, där dessa för fallet av åtta gångers granularitetsförfining överpresterade gentemot ett klassiskt verktyg; Adobe Photoshop. Detta lovande resultat är dock på bekostnad av högre beräkningskapacitet. För klassificering presterar dessa typer av nätverk generellt sämre än faltningsnätverk, med något enskilt undantag. Faltningsnätverk som är specifikt konstruerade för att extrahera egenskaper ur lokala regioner i bilder, vilket KAN-nätverk och SKAN-varianter inte är.

8 Framtidsutsikter

Detta arbete utgör en grund för vidare studier inom ämnet. Arbetet visar på potential hos Kolmogorov-Arnold Nätverk för regressioner, vilket kan och bör undersökas vidare. Primärt intressant är att studera hur olika delar av nätverken påverkar prediktioner. Något som undersökningen även borde utvecklas i är vilka tekniker som kan användas för att träna dessa nätverk effektivare, då det för tillfället är den största begränsningen. För att denna typ av nätverk ska börja användas frekvent måste deras träning effektiviseras. Slutligen är det även intressant att undersöka hur olika funktionsapproximationer kan användas lokalt i noder. Exempel på detta är att modellera dem med små neurala nätverk, istället för att ha en uppsättning av basfunktioner med motsvarande parametrar. En annan aspekt att undersöka är användandet av funktioner av fler variabler som inre funktioner, och att ansluta dessa typer av nätverk till slutet av ett faltningsnätverk. Detta implementeras istället för ett klassiskt neuralt nätverk. Trots att ämnet endast berörs översiktligt i detta arbete, visar projektet på en lovande potential för fortsatt forskning.

Med grund i SKANs höga prestanda motiveras ytterligare undersökning av konvergensanalys i hur nätverken förbättras med antal parametrar. Skalar SKAN lika väl som KAN med stora mängder parametrar är det ett lovande nätverk som bör utvecklas vidare. Integrering i andra nätverksstrukturer, exempelvis kombinera faltningsnätverk med SKAN är tilltalande, eftersom båda nätverken har hög prestanda inom klassificering. Det kan vara betydelsefullt att undersöka normalisering av data mellan lager, vilket möjliggör ett konsekvent knutintervall för alla.

Referenser

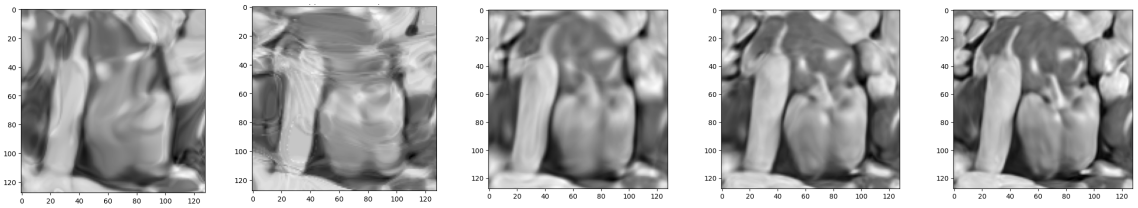
1. Liu Z, Wang Y, Vaidya S, Ruehle F, Halverson J, Soljačić M, Hou TY och Tegmark M. KAN. arXiv:2404.19756 2024. DOI: https://doi.org/10.48550/arXiv.2404.19756
2. Kolmogorov A. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. American Mathematical Society Translations 1963; 28:55–9
3. Aggarwal CC. Neural Networks and Deep Learning. Springer, 2018. DOI: 10.1007/978-3-319-94463-0. Available from: https://doi.org/10.1007/978-3-319-94463-0
4. Goodfellow I, Bengio Y och Courville A. Deep Learning. MIT Press, 2016. Available from: http://www.deeplearningbook.org
5. Li Z. Kolmogorov-Arnold Networks are Radial Basis Function Networks. 2024. DOI: https://doi.org/10.48550/arXiv.2405.06721. arXiv: 2405.06721
6. Boor C de. A Practical Guide to Splines. JSTOR, 1980. DOI: 10.2307/2006241. Available from: https://doi.org/10.2307/2006241
7. Sprecher D. On the Structure of Continuous Functions of Several Variables. Transactions of the American Mathematical Society 1965; 115:340–55
8. Köppen M och Yoshida K. Universal Representation of Image Functions by the Sprecher Construction. *Soft Computing as Transdisciplinary Science and Technology*. Utg. av Abraham A, Dote Y, Furuhashi T, Köppen M, Ohuchi A och Ohsawa Y. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005 :202–10
9. Goldberg Y. Neural Network Methods in Natural Language Processing. Vol. 10. Synthesis Lectures on Human Language Technologies 1. Morgan & Claypool Publishers, 2017
10. Demb R och Sprecher D. A note on computing with Kolmogorov Superpositions without iterations. Neural Networks 2021; 144:438–42. DOI: https://doi.org/10.1016/j.neunet.2021.07.006. Available from: https://www.sciencedirect.com/science/article/pii/S0893608021002690
11. Polson S och Sokolov V. Kolmogorov GAM Networks are all you need! 2025. arXiv: 2501.00704 [cs.LG]. Available from: https://arxiv.org/abs/2501.00704 [Accessed on: 2025 Jun 1]
12. Kingma DP och Ba J. Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980 2015. Available from: https://arxiv.org/abs/1412.6980
13. Loshchilov I och Hutter F. Decoupled Weight Decay Regularization. arXiv preprint arXiv:1711.05101 2019. Available from: https://arxiv.org/abs/1711.05101
14. The USC-SIPI Image Database. Available from: http://sipi.usc.edu/database/ [Accessed on: 2025 May 8]
15. Adobe Inc. Image size and resolution in Photoshop. [Internet]. San Jose (CA): Adobe; [updated 2025; cited 2025 May 30]. 2024. Available from: https://helpx.adobe.com/photoshop/using/image-size-resolution.html

Användande av artificiell intelligens

I denna artikel har AI i form av OpenAIs ChatGPT använts i mycket liten grad. Det har använts för att bygga förståelse om *Pytorch*-biblioteket, mer specifikt *AdamW*-algoritmen, inbyggda funktioner samt tensorer. Samtliga områden hade kunnat utforskas genom att använda en sökmotor, men denna åtgärd har tagits för att spara tid. Ingen kod eller annat material till rapporten har genererats med dessa verktyg.

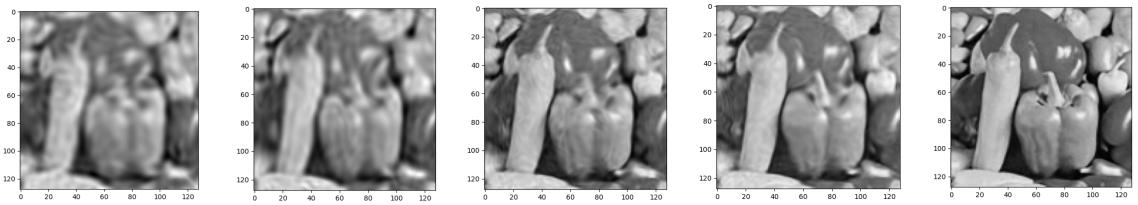
A Bildregressioner och bildrekonstruktioner

A.1 Bildregressioner



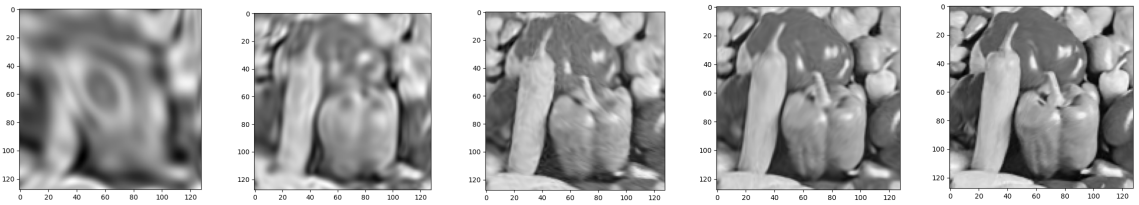
480 1000 2000 4960 9948

Figur 10: Bildregression för KAN på bilden *Peppers*. Samtliga nätverk är tränade på bilden med dimension 128x128. Bilderna benämns med antalet parametrar som nätverken haft.



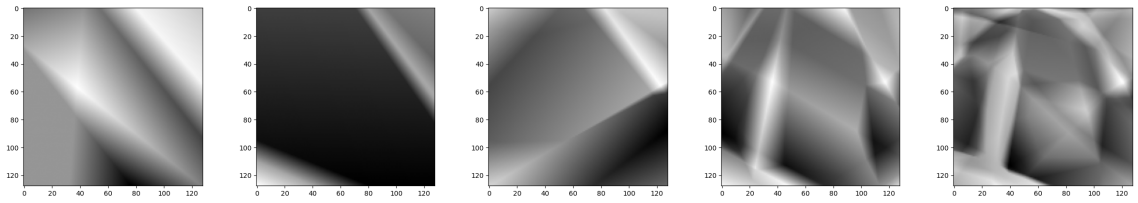
511 1011 2022 5090 9948

Figur 11: Bildregression för SprecherLambda på bilden *Peppers*. Samtliga nätverk är tränade på bilden med dimension 128x128. Bilderna benämns med antalet parametrar som nätverken haft.



491 1024 2052 5067 9960

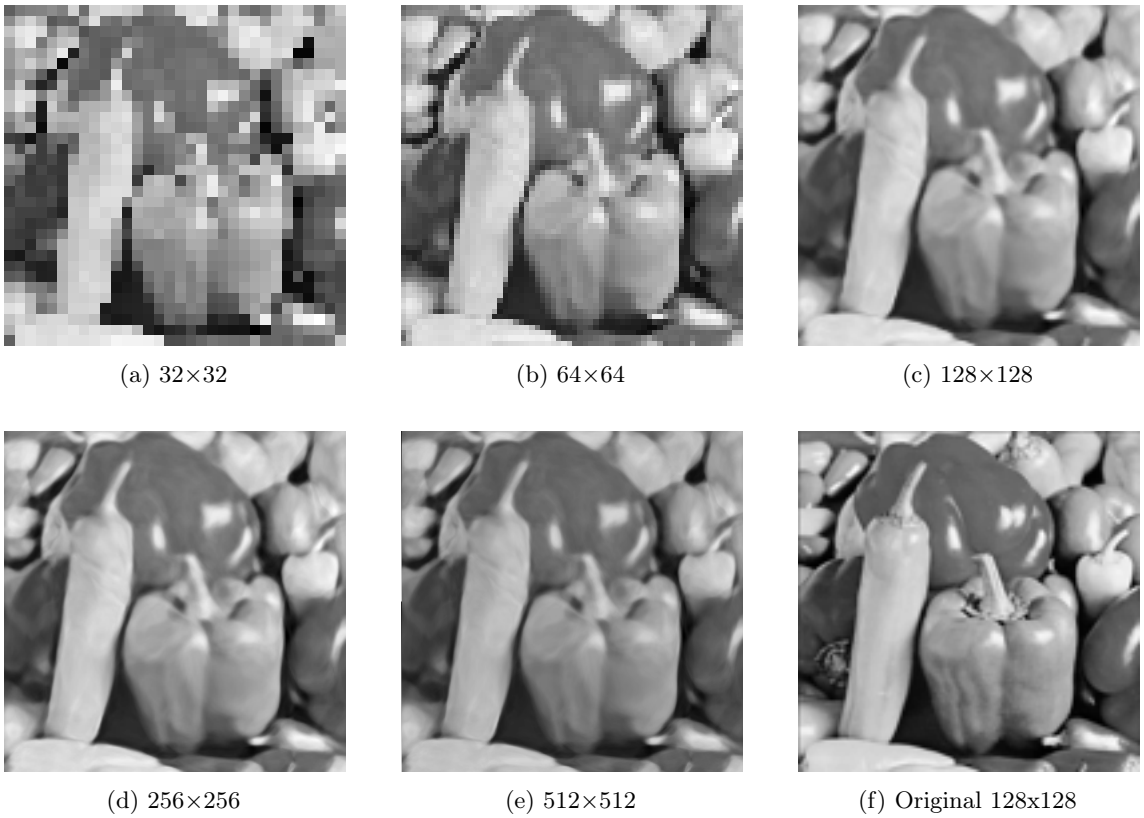
Figur 12: Bildregression för SprecherComb på bilden *Peppers*. Samtliga nätverk är tränade på bilden med dimension 128x128. Bilderna benämns med antalet parametrar som nätverken haft.



500 1000 2000 5000 10000

Figur 13: Bildregression för ett MLP på bilden *Peppers*. Samtliga nätverk är tränade på bilden med dimension 128x128. Bilderna benämns med antalet parametrar som nätverken haft.

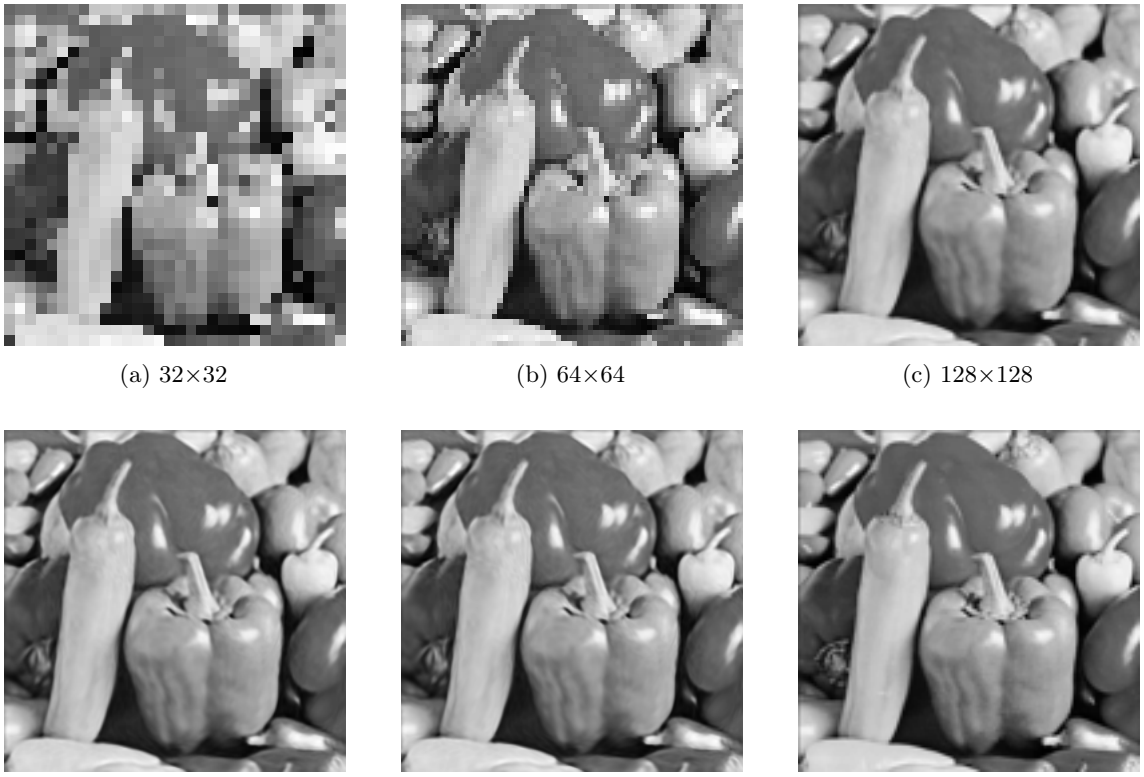
A.2 Bildrekonstruktioner



(a) 32×32 (b) 64×64 (c) 128×128

(d) 256×256 (e) 512×512 (f) Original 128x128

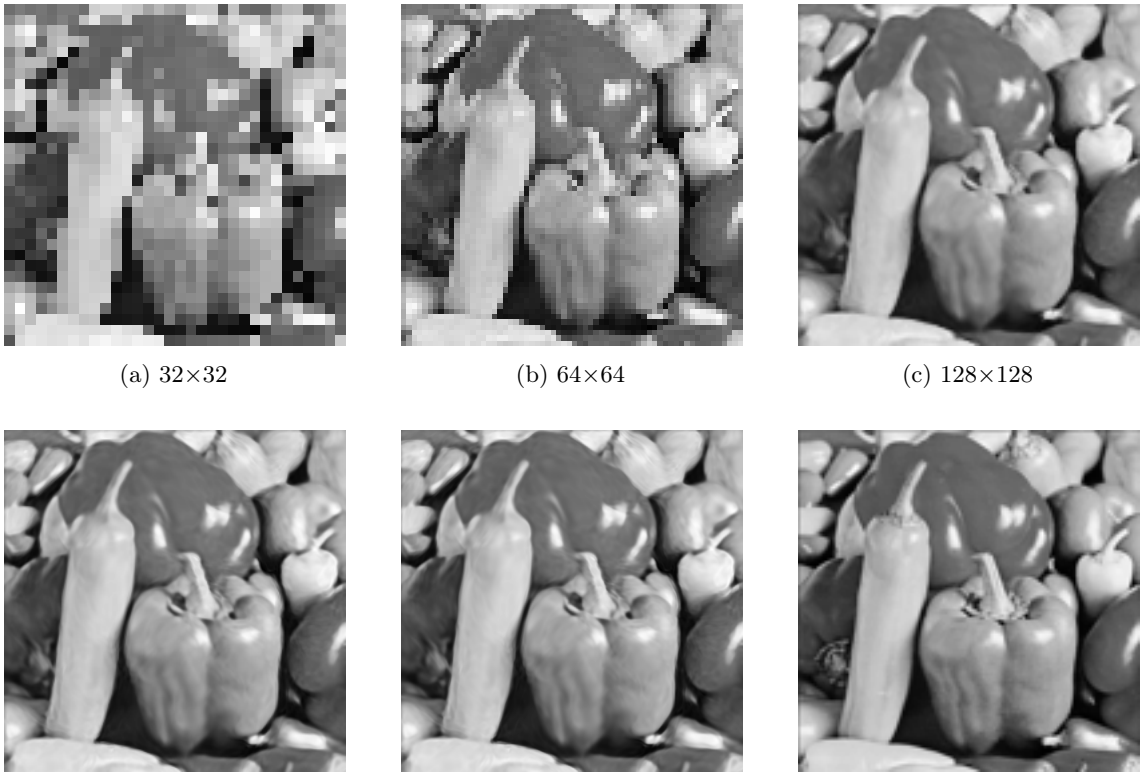
Figur 14: Bildrekonstruktion för KAN av bilden *Peppers*. Nätverket har tränats på bilden med dimension 128x128.



(a) 32×32 (b) 64×64 (c) 128×128

(d) 256×256 (e) 512×512 (f) Original 128x128

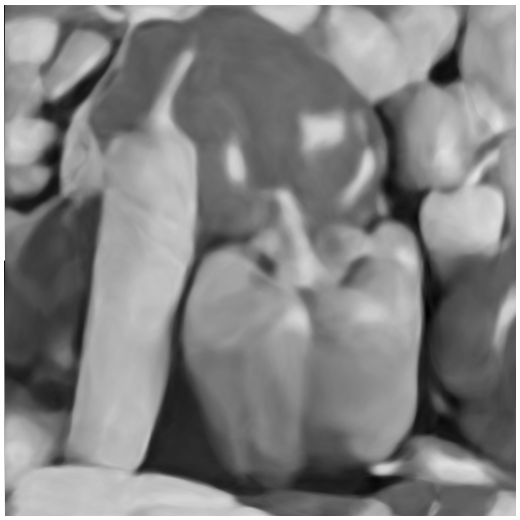
Figur 15: Bildrekonstruktion för SprecherLambda på bilden *Peppers*. Nätverket har tränats på bilden med dimension 128x128.



(a) 32×32 (b) 64×64 (c) 128×128

(d) 256×256 (e) 512×512 (f) Original 128x128

Figur 16: Bildrekonstruktion för SprecherComb på bilden *Peppers*. Nätverket har tränats på bilden med dimension 128x128.



(a) 512x512 interpolation av KAN.

(b) 512x512 interpolation av SprecherLambda.

(c) 512x512 interpolation av SprecherComb.

(d) 512×512 av Adobe Photoshop.

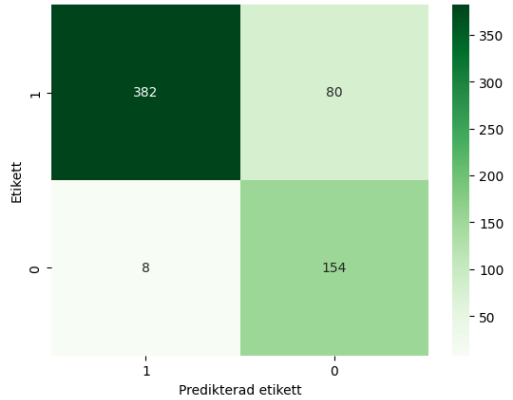
Figur 17: Jämförelse av rekonstruktion av 512x512 från en bild av dimension 128x128. SprecherLambda och SprecherComb skapar finare konturer i interpolationen än vad Adobe Photoshop och KAN gör.

(a) 512x512 interpolation av SprecherLambda

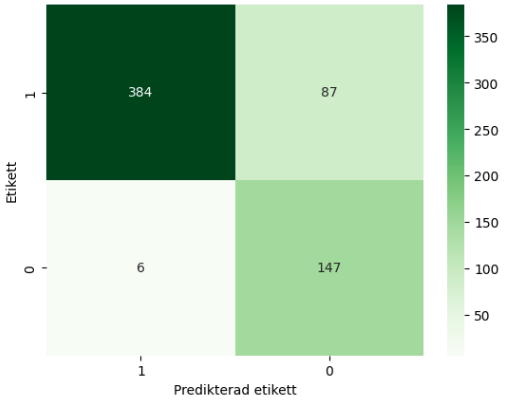
(b) 512x512 interpolation av Adobe Photoshop.

Figur 18: Jämförelse mellan SprecherLambda och Adobe Photoshop av Interpolation från 64x64 pixlar till 512x512.

B Förväxlingsmatris

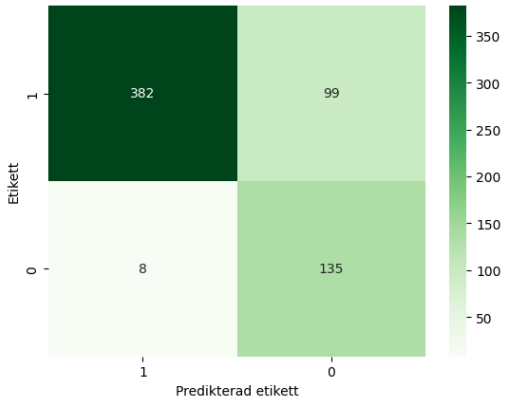


(a) Bästa prediktionen för klassificering av KAN med 5488 parametrar. Antalet korrekt klassificerade är 85.90 %.

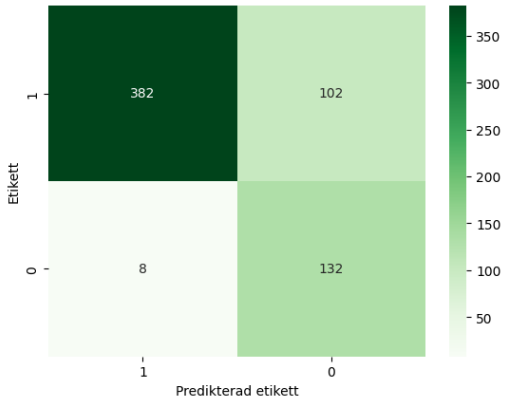


(b) Sista prediktionen för klassificering av KAN med 5488 parametrar. Antalet korrekt klassificerade är 85.10 %.

Figur 19: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av KAN med 5488 parametrar.

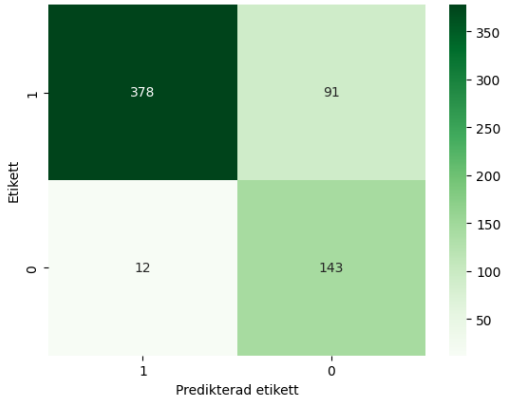


(a) Bästa prediktionen för klassificering av KAN med 10192 parametrar. Antalet korrekt klassificerade är 82.85 %.

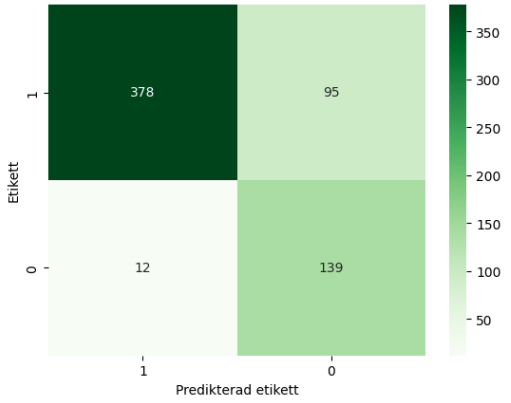


(b) Sista prediktionen för klassificering av KAN med 10192 parametrar. Antalet korrekt klassificerade är 82.37 %.

Figur 20: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av KAN med 10192 parametrar.

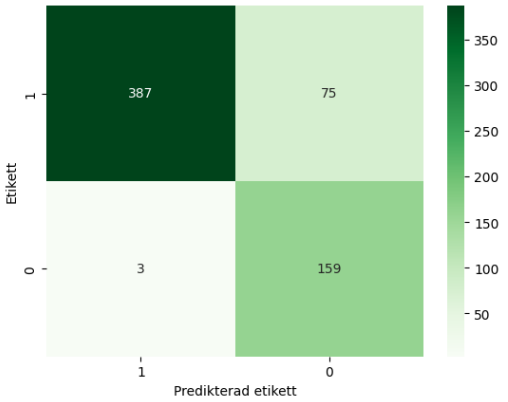


(a) Bästa prediktionen för klassificering av KAN med 14896 parametrar. Antalet korrekt klassificerade är 83.49 %.

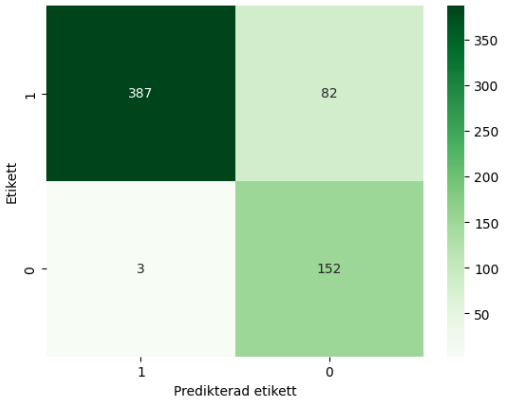


(b) Sista prediktionen för klassificering av KAN med 14896 parametrar. Antalet korrekt klassificerade är 82.85 %.

Figur 21: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av KAN med 14896 parametrar.

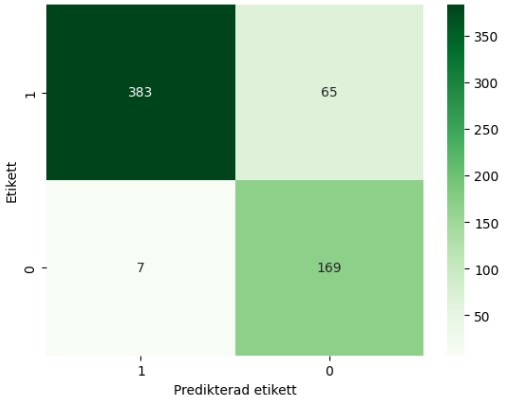


(a) Bästa prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 87.50 %.

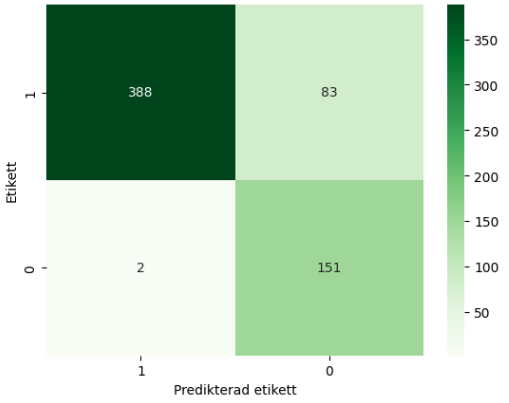


(b) Sista prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 86.38 %.

Figur 22: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av faltningsnätverk med 5488 parametrar.

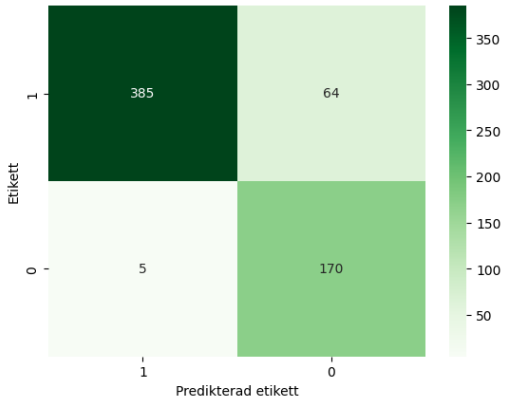


(a) Bästa prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 88.46 %.

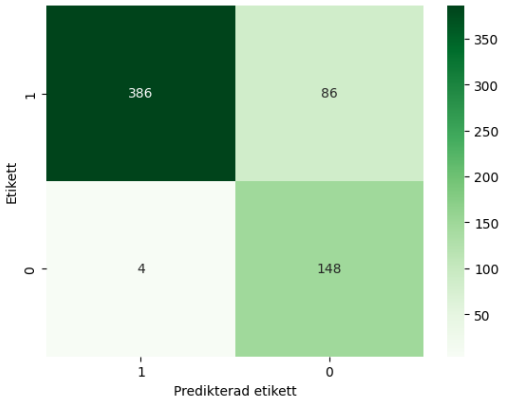


(b) Sista prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 86.38 %.

Figur 23: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av faltningsnätverk med 10192 parametrar.

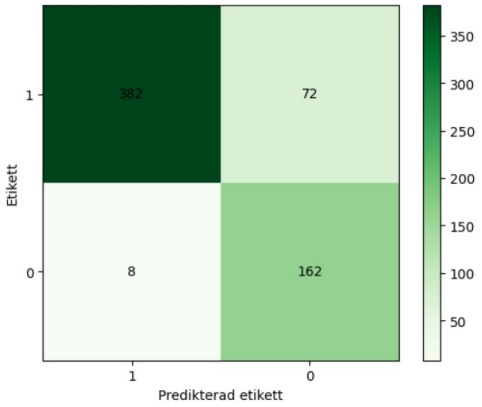


(a) Bästa prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 88.94 %.



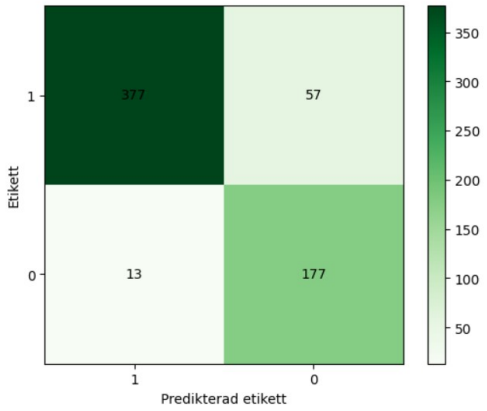
(b) Sista prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 85.58 %.

Figur 24: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av faltningsnätverk med 14896 parametrar.

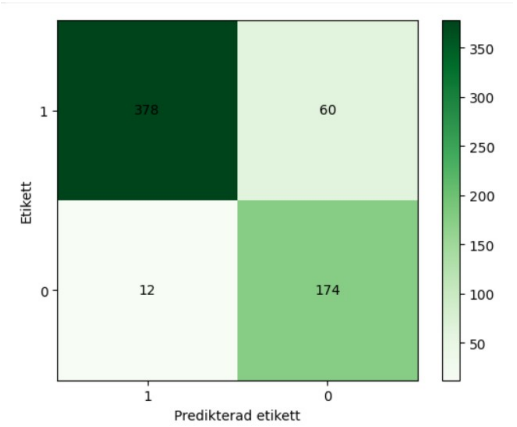


(a) Bästa och prediktionen för klassificering av SprecherLambda. Antalet korrekt klassificerade är 87.18 %.

Figur 25: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av SprecherLambda med 5531 parametrar.

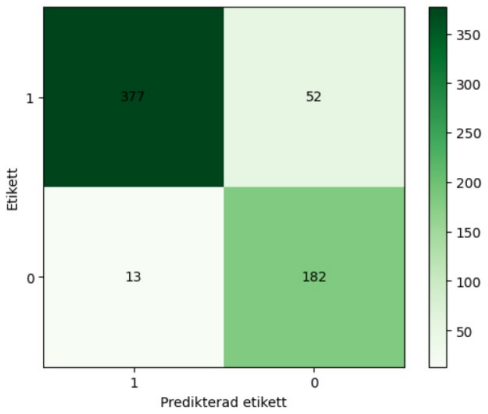


(a) Bästa prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 88.78 %.



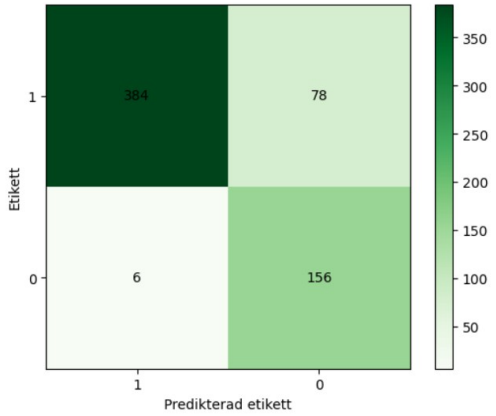
(b) Sista prediktionen för klassificering av SprecherLambda. Antalet korrekt klassificerade är 88.46 %.

Figur 26: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av SprecherLambda med 11022 parametrar.

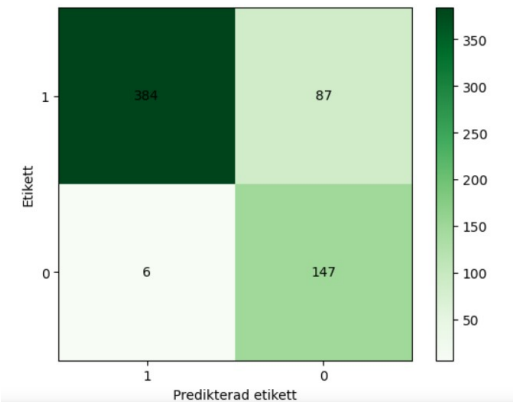


(a) Bästa och prediktionen för klassificering av SprecherLambda. Antalet korrekt klassificerade är 89.58 %.

Figur 27: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av SprecherLambda med 16457 parametrar.

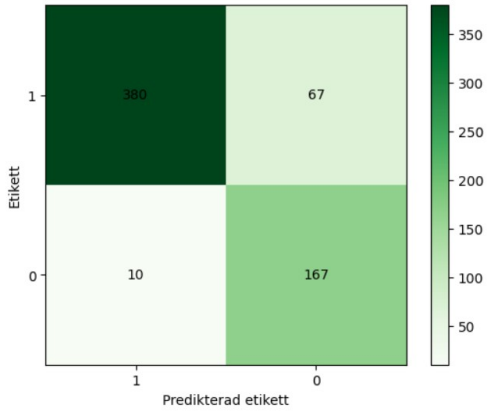


(a) Bästa prediktionen för klassificering av SprecherComb. Antalet korrekt klassificerade är 86.54 %.

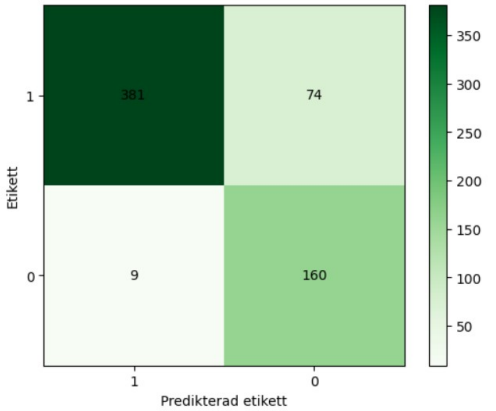


(b) Sista prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 85.10 %.

Figur 28: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av SprecherComb med 5639 parametrar.

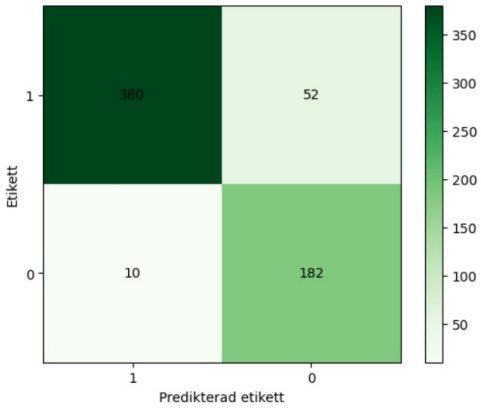


(a) Bästa prediktionen för klassificering av SprecherComb. Antalet korrekt klassificerade är 87.66 %.

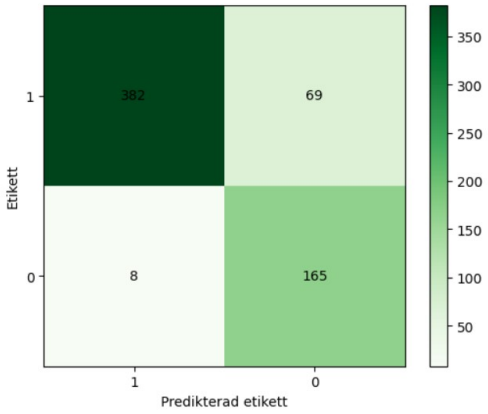


(b) Sista prediktionen för klassificering av faltningsnätverk. Antalet korrekt klassificerade är 86.70 %.

Figur 29: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av SprecherComb med 11604 parametrar.



(a) Bästa prediktionen för klassificering av SprecherComb. Antalet korrekt klassificerade är 90.06 %.



(b) Sista prediktionen för klassificering av SprecherComb. Antalet korrekt klassificerade är 87.66 %.

Figur 30: Jämförelse av prediktioner och faktiska etiketter för klassificeringar av SprecherComb med 16739 parametrar.

C Appendix 2 – källkod

C.1 KAN och MLP

C.1.1 Skript 1: KAN-nätverken

```
"""
 DOC INFO for modules.py

    Contains classes functional (1) and sequential (2).

    (1) functional represents a KAN layer
    (2) sequential builds a model sequentially, taking many KAN layers as inputs

"""
# Import necessary packages
import cox_de_boor
from torch import normal, empty, from_numpy, transpose, float64, sum as torch_sum, linspace
from torch.nn import LogSoftmax, Parameter
import torch
from scipy.interpolate import BSpline
import seaborn as sns
from torch.nn.functional import mse_loss
import numpy as np
from torch import linspace, tensor, stack, cat, zeros, randn, nn, randperm
from torch.utils.data.dataset import random_split
import matplotlib.pyplot as plt
from math import ceil
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class functional:
    """ Models one layer in a classical KAN using functional units (KAN nodes).

        Future potential upgrades:
        - "variant" argument to specify the variant of the node, i.e classical or
            David Sprechers representation
    """
    # Give instructions on how to initiate the model
    def __init__(self, nr_in, nr_out, k, t):

        # Readability variables
        self.nr_of_basis = len(t)-k-1 + 2*k # 2*k term comes from grid extension

        # Network hyperparameters init
        self.nr_in = nr_in
        self.nr_out = nr_out

        # Spline initialization
        self.t = t # control points
        self.k = k # degree
```

```
44          self.C = Parameter(normal(0, 0.1, size=(self.nr_in, self.nr_of_basis, self.
                nr_out), requires_grad=True, dtype=torch.float64, device=device))
45
46      # Define a standard __call__ function
47      def __call__(self, x):
48          return self.forward(x)
49
50      # Define how the data should be propagated
51      def forward(self, x):
52          """
53          A forward pass using input x. Outputs
54
55          Parameters
56          x : Input data tensor of size (nr of data, nr of inputs).
57          """
58
59          # Initializing basis tensor
60          nr_data = x.shape[0]
61          all_basis = empty(size=(self.nr_in, nr_data, self.nr_of_basis), dtype=
                float64, device=device).to(device)
62
63          # Propagated differently depending on whether there is one or multiple rows
                in the tensor
64          x_dimension = len(x.shape)
65          if x_dimension == 1:
66
67              all_basis[0] = cox_de_boor.basis_functions(x, t=self.t, k=self.k)
68
69          elif x_dimension == 2:
70              # Fill basis tensor before matrix multiplication
71              for i in range(x.shape[1]):
72                  # t change --> number of basis change --> number of parameters c
                        change (the formula len(t)-k-1+2*k still holds)
73                  all_basis[i] = cox_de_boor.basis_functions(x[:, i], t=self.t, k=
                        self.k)
74          else:
75              raise Exception("Input tensor dimension cannot exceed 2")
76
77          # Sum all basis splines
78          result = torch.sum(all_basis @ self.C, dim=0, dtype=float64).to(device)
79          return result
80
81      def parameters(self):
82          # getter method
83          return self.C
84
85
86  class sequential:
87      """Builds models sequentially like in PyTorch"""
88      def __init__(self, *layers):
89          self.layers = layers
90
91      def __call__(self, x):
92          return self.forward(x)
93
94      def forward(self, x):
95          # propagate though the network
96          for layer in self.layers:
97              x = layer(x)
98          return x
99
100     def parameters(self):
101         # provides an iterable containing torch.nn objects
102         parameters = []
103         for layer in self.layers:
104             parameters.append(layer.parameters())
105         return parameters
106
107     def train_model(self, x, targets, optimizer, loss_criterion, nr_epochs,
            print_loss_each=1000, classification_task = False):
108         """
109         x : input data for forward pass
110         target : the target values. NOTE: MAKE SURE IT IS A [1000, 1] TENSOR NOT
                JUST [1000]
111         optimizer : Chosen PyTorch optimzier
112         loss_criterion : a PyTorch defined loss function
113         nr_epochs : number of epochs
114         print_loss_each : print loss each j number of epochs
115         """
116         loss_list = []
117         for epoch in range(nr_epochs+1):
118
119             # Reset gradients
120             optimizer.zero_grad()
```

```
121
122             # Propagate the input through the network in order to obtain
                    predictions
123             predictions = self.forward(x)
124
125             # Compute the loss
126             loss = loss_criterion(predictions, targets)
127
128             # Compute gradients and optimize weights
129             loss.backward()
130             optimizer.step()
131
132             # print loss every print_loss_each steps
133             if epoch % print_loss_each == 0:
134                 print("Epoch", epoch, "loss: ", loss.item())
135
136             # Save the loss for visualization purposes
137             loss_list.append(loss.item())
138
139         # Define for what number of epochs the training loss were the lowest
140         best_amount_of_epochs = np.argmin(loss_list)
141
142         print(f"The best amount of epochs is {best_amount_of_epochs}")
143
144         # Plot the training loss
145         plt.figure()
146         plt.plot([i + 1 for i in range(nr_epochs + 1)], loss_list)
147         plt.xlabel("Antal epoker")
148         plt.ylabel("Medelkvadratf rlust f r tr ning")
149         plt.title("MSE f r tr ning")
150         plt.ylim(0, 0.01)
151         plt.grid()
152         plt.show()
153
154         # Return where the training loss was at its lowest
155         return best_amount_of_epochs
156
157     def batch_train_model(self, x, targets, optimizer, loss_criterion, nr_epochs,
        print_loss_each=1000, batch_size = 250, plot_loss = True, mse_plot_limit =
        0.2, classification_task = False, testing_data = torch.tensor(0).to(device)
        , testing_targets = torch.tensor(0).to(device)):
158         """
159         Arguments:
160         x : input data for forward pass
161         target : the target values. NOTE: MAKE SURE IT IS A [1000, 1] TENSOR NOT
                JUST [1000]
162         optimizer : Chosen PyTorch optimzier
163         loss_criterion : a PyTorch defined loss function
164         nr_epochs : number of epochs
165         print_loss_each : print loss each j number of epochs
166         """
167         # Number of batches
168         nr_data = x.size()[0]
169         nr_batches = ceil(nr_data / batch_size)
170
171         # Define a list to save the losses
172         loss_list = []
173
174         # Loop through all the epochs
175         for epoch in range(nr_epochs+1):
176
177             # Shuffle the indices - in order to batch-train properly
178             shuffled_indices = randperm(nr_data).to(device)
179
180             # Storing the epoch loss
181             epoch_loss = 0
182
183             # Loop through every batch
184             for batch_idx in range(nr_batches):
185
186                 # Extract the batch indices and check whether the batch does not
                        threspass the length of the list
187                 start_idx = batch_idx * batch_size
188                 end_idx = min(start_idx + batch_size, nr_data)
189
190                 # Define the indices of the batch
191                 batch_indices = shuffled_indices[int(start_idx) : int(end_idx)]
192
193                 # Dividing the data into batches
194                 batch_x, batch_targets = x[batch_indices], targets[batch_indices]
195
196                 # Reset gradients
197                 optimizer.zero_grad()
198
```

```
                # Forward the batch
                batch_predictions = self.forward(batch_x)

                # Compute loss
                batch_loss = loss_criterion(batch_predictions, batch_targets)
                epoch_loss += batch_loss.item() * (end_idx - start_idx) / nr_data

                # Compute gradients and optimize weights
                batch_loss.backward()
                optimizer.step()

            # Append to the loss list
            loss_list.append(epoch_loss)

            # Print epochs loss
            if epoch % (print_loss_each) == 0:
                print("Epoch", epoch, "loss: ", epoch_loss)

            # Test the model every epoch if it is a classification task
            if classification_task:
              _ = self.test_model(x = testing_data, targets = testing_targets,
                  classification_task = True)

        # The user can specify whether the training loss should be plotted or not
        if plot_loss:
          plt.title("MSE f r  tr ning")
          plt.plot([epoch for epoch in range(nr_epochs + 1)], loss_list)
          plt.ylim(0, mse_plot_limit)
          plt.show()

    # Get the tensors of the individual spline, if one wants to plot them for
        instance
    def get_individual_splines(self, nr_in = 2):
        """
        This function returns all the individual splines when using a dynamic
            interval.
        """

        # Setting spline domain
        nr_data = 1000
        spline_domain = stack([linspace(-2, 2, nr_data) for _ in range(nr_in)], dim
            = 0)
        spline_points = [spline_domain.clone()]
        all_splines = [] # nr_in x nr_data x nr_out

        # Loop through all layers
        for layer in self.layers:
            layer_splines = []
            layer_output = zeros([nr_data, layer.nr_out])

            # Loop through all the previous saved domains
            for mu, element in enumerate(spline_domain):

                # Extract the basis functions
                basis_functions = cox_de_boor.basis_functions(element.T, layer.t,
                    layer.k) # (input_data x nr_of_basis_functions), different
                    domains of each

                # Batch tensor-matrix multiplication to compute the splines
                spline = (basis_functions.float() @ layer.C[mu].float()).to(device)
                layer_output += spline

                # Append the splines to a list to make them easy accessible
                layer_splines.append(spline)

            # Re-define the spline domain for the upcoming layer
            spline_domain = layer_output.T

            # Save the spline domains in a gathered list
            spline_points.append(spline_domain.clone())

            # Reconstruct the layer_splines list to a tensor
            layer_splines = stack(layer_splines, dim = 0).to(device)

            # Append all the splines in the specific layer to a list that gathers
                all the splines
            all_splines.append(layer_splines)

        # Return the domains
        return spline_points, all_splines

    def plot_splines(self, spline_domain, all_splines):

        # Get the splines
```

```
276             spline_domain, all_splines = self.get_individual_splines()
277
278             # Loop through all layers in the sequential
279             for mu, layer in enumerate(self.layers):
280
281                 # Create a figure to plot on
282                 plt.figure(figsize=(4 * layer.nr_out, 3 * layer.nr_in))
283
284                 # Loop through all input neurons in the layer
285                 for input_neuron in range(layer.nr_in):
286
287                     # Loop through all input neurons in the layer
288                     for output_neuron in range(layer.nr_out):
289
290                         # Subplot all the splines in the layer
291                         plt.subplot(layer.nr_in, layer.nr_out, input_neuron * layer.
                                nr_out + output_neuron + 1)
292                         plt.title(f"Spline_{mu + 1}_{input_neuron + 1}_{output_neuron +
                                1}")
293
294                         # Needs to be sorted?
295                         plt.plot(spline_domain[mu][input_neuron].detach().numpy(),
                                all_splines[mu][input_neuron].T[output_neuron].detach().
                                numpy())
296
297                 # Set an overall title
298                 plt.suptitle(f"Spline_layer_inputneuron_outputneuron")
299                 plt.show()
300
301         def test_model(self, x, targets, loss_criterion = nn.L1Loss(), plot_splines =
            False, classification_task = False):
302
303             # Check whether it is a regression or a classification task
304             if classification_task:
305               """
306               predictions = torch.sigmoid(self.forward(x))
307               total_correct = ((predictions - targets).abs() < 0.5).sum().item()
308               total_samples = targets.size(0)
309
310               # Verify that total_samples is not zero, to avoid division by zero
311               if total_samples != 0:
312                 model_accuracy = total_correct / total_samples
313                 print(f"The model accuracy of test images is {model_accuracy * 100 :.4f
                        }%")
314               """
315               # Define that we do not want to update weights when evaluating
316               with torch.no_grad():
317
318                 # Sigmoid, which BCELossWithLogits, the classification optimizer used,
                        does internally
319                 predictions = torch.sigmoid(self.forward(x))
320
321                 # Separate which are correctely predicted, the labels are either 0 or
                        1, all above 0.5 is a 1 prediction
322                 predicted_labels = (predictions >= 0.5).float()
323                 targets = targets.view(-1, 1).float()
324
325                 # Count the total correct predicted
326                 total_correct = (predicted_labels == targets).sum().item()
327                 total_samples = targets.size(0)
328
329                 # Calculate the accuracy
330                 if total_samples != 0:
331                     model_accuracy = total_correct / total_samples
332                     print(f"The model accuracy of test images is {model_accuracy *
                            100:.2f}%")
333
334                 # Extract the predicted values to the cpu, needs to constructed to
                        numpy array
335                 preds = predicted_labels.cpu()
336                 truth = targets.cpu()
337
338                 # Count True Positive, True Negative, False Positive and False Negative
339                 TP = ((preds == 1) & (truth == 1)).sum().item()
340                 TN = ((preds == 0) & (truth == 0)).sum().item()
341                 FP = ((preds == 1) & (truth == 0)).sum().item()
342                 FN = ((preds == 0) & (truth == 1)).sum().item()
343
344                 # Construct a confusion matrix, a heatmap
345                 conf_matrix = np.array([[TP, FP], [FN, TN]])
346
347                 # Define the labels for the axises
348                 labels = ['1', '0']
349
```

```
            # Construct a heatmap
            plt.figure()
            sns.heatmap(conf_matrix, annot = True, fmt = "d", cmap = "Greens",
                xticklabels = labels, yticklabels = labels)
            plt.xlabel("Predikterad etikett")
            plt.ylabel("Etikett")
            plt.show()


        # If not a classification task
        else:
          predictions = self.forward(x)

        # Calculate the loss
        loss = loss_criterion(predictions, targets)

        # The user can specify if he wants to plot the splines
        if plot_splines:
            spline_domain, all_splines = self.get_individual_splines()
            self.plot_splines(spline_domain, all_splines)

        # Return the loss value
        return loss
    def sigmoid(self, x):
      return 1 / (1 + torch.exp(x))

class KAN:

    def __init__(self, *structure, t, k):
        """ Arguments
        structure : structure of network given as n-tuple. EX. fast_init(2, 5, 1, t
            , k) yields 2-5-1 KAN.
        """
        self.layers = []

        for i in range(len(structure)-1):
            layer = functional(nr_in=structure[i], nr_out=structure[i+1], k=k, t=t)
            self.layers.append(layer)

    def __call__(self, x):
        return self.forward(x)

    def parameters(self):
        # provides an iterable containing torch.nn objects
        parameters = []
        for layer in self.layers:
            parameters.append(layer.parameters())
        return parameters

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def train_model(self, x, targets, optimizer, loss_criterion, nr_epochs,
        print_loss_each=1000):
        """
        x : input data for forward pass
        targets : the target values. NOTE: MAKE SURE IT IS A [1000, 1] TENSOR NOT
            JUST [1000]
        optimizer : Chosen PyTorch optimzier
        loss_criterion : a PyTorch defined loss function
        nr_epochs : number of epochs
        print_loss_each : print loss each j number of epochs
        """
        for epoch in range(nr_epochs+1):

            # Reset gradients
            optimizer.zero_grad()

            # Compute loss
            predictions = self.forward(x)
            print("predictions shape", predictions.shape)
            print("targets shape", targets.shape)
            loss = loss_criterion(predictions, targets)

            # Compute gradients and optimize weights
            loss.backward()
            optimizer.step()

            # print loss every print_loss_each steps
            if epoch % print_loss_each == 0:
                print("Epoch", epoch, "loss: ", loss.item())
```

C.1.2 Skript 2: Cox-de-Boor's rekursionsformel

```
# Import necessary packages
from torch import cat
import torch
# Define that we want to use a gpu if it is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def basis_functions(x, t, k, extended=True):
    # Implements Cox-De Boor formula to obtain Basis functions for BSplines
    # This function is a re-written form of the Github user pg2455 who has provided
        his/her code under the MIT license. I have just simplified it.

    # Returns :

    # x: input data tensor
    # t: control points in 1D
    # k: degree of piecewise polyonomials

    if extended:
        t = extend_grid(t, k)

    # Reshape to perform comparisons.
    t_ = t.unsqueeze(dim=0)
    x_ = x.unsqueeze(dim=1)

    # Cox-de-Boor recursion formula
    for j in range(k+1):
        # Base case: j = 0
        if j == 0:
            N_j = ((x_ >= t_[:,:-1]) * ((x_ < t_[:, 1:])) * 1.0).to(device)
        else:
            left = (x_ - t_[:, :-(j+1)]) / (t_[:, 1:-j] - t_[:, :-(j+1)]) * N_j[:,
                :-1]
            right = (t_[:, (j+1):] - x_) / (t_[:, (j+1):] - t_[:, 1:-j]) * N_j[:,
                1:]
            N_j = left+right
    return N_j

def extend_grid(grid, k):

    """
    Extends the grid by 2*k points, k points per side

    Args:
        grid: number of splines x number of control points
        k: spline order

    Returns:
        new_grid: number of splines x (number of control points + 2 * k)
    """
    n_intervals = grid.shape[-1] - 1
    h = (grid[-1] - grid[0]) / n_intervals # step length to determine t_(i+1) = h*
        t_i

    for i in range(k):
        grid = torch.cat([grid[:1] - h, grid], dim=-1).to(device) # logic
            translated : t_(-1) := t_0 - h --> append t_(-1) to grid
        grid = torch.cat([grid, grid[-1:] + h], dim=-1).to(device) # t_(n+1) := t_n
             + h --> append t_(n+1) to grid

    return grid
```

C.1.3 Skript 3: Analys av KAN

```
# Importing libraries
import matplotlib.pyplot as plt
import torch
import numpy as np
import seaborn as sns

from torch.utils.data.dataset import random_split
import importlib
import alexKAN
importlib.reload(alexKAN)
from torch.optim.lr_scheduler import StepLR
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Train_test_split function
def train_test_split(input, target, test_fraction = 0.2):
    """
```

```
        Description:
        Splits the data with a fraction of training and test samples. The model keeps
            track that the correct target data are used for the correct input data.

        Arguments:
        input : The data that should be used.
        target : The target of prediction.
        test_fraction : Defines the fraction that should be in the test data sample.
            Standard: 0.2


        Returns:
        train_input : The splitted training input.
        train_target : The splitted training target.
        test_input : The splitted testing input.
        test_target : The splitted testing target.

        """
            # Metrics for the data
            data_length = input.size()[0]
            train_size = int((1 - test_fraction) * data_length) # Generate the train
                size

            # Random permutation of the tensors when merged to keep track of which
                inputs correspond to the same outputs
            perm = torch.randperm(nr_of_data).to(device)
            input, target = input[perm, :].to(device), target[perm, :].to(device)

            # Divide permutation based on training size
            train_input, train_target = input[:train_size, :].to(device), target[:
                train_size, :].to(device)
            test_input, test_target = input[train_size:, :].to(device), target[
                train_size:, :].to(device)

            # Return the train and test tensors
            return train_input, train_target, test_input, test_target

    """
    The following is for the function regression analysis.
    """

    # Defining the domain
    a = -2
    b = 2
    nr_of_data = 10000

    # Model parameters
    nr_of_knots = 6

    # t + k - 1 parameters per basis function, k = 3
    t_6 = torch.linspace(a, b, nr_of_knots).to(device)
    t_14 = torch.linspace(a, b, 14).to(device)
    k = 3 # degree
    weight_decay = 1e-2 # Default weight_decay - l2 regularization
    learning_rate = 0.01
    epochs = 500
    batch_size = 250


    def train_and_test(nr_of_seeds = 1, nr_of_data = 10000, batch_size = 250): # should
         take the model size as well
        """
        Description:
        Trains and test the specified KAN models stored in model_factiories for a
            certain amount of seeds.

        Arguments:
        nr_of_seeds : The amount of seeds the model should be trained and tested over
        batch_size : The batch size used for training the model

        Returns:
        mean_loss : The mean loss over all the seeds for all the models.

        """
        # Takes the model as an input
        mean_loss = torch.zeros(5).to(device)
        seed_loss = torch.zeros(nr_of_seeds).to(device)
        best_epochs_512 = [6051, 9861, 3215, 9736, 7815, 7641, 8063, 9979, 9024, 9213]
            # 512 parameters

        # Model optimizer
        loss_criterion = torch.nn.MSELoss()

        #best_nr_of_epochs_for_each_seed = []
```

```
92
93      for seed in range(nr_of_seeds):
94          # Reset the model
95          print(f"Seed {seed}")
96
97          # Input data
98          torch.manual_seed(seed) # setting the seed
99          x_data = (a - b) * torch.rand(nr_of_data).to(device) + b
100         y_data = (a - b) * torch.rand(nr_of_data).to(device) + b
101
102         # Targets
103         z_data = torch.empty(size = (nr_of_data, 1), dtype = torch.float64).to(
                device)
104
105         f = lambda x, y: 4 * x * y ** 2
106         z_data[:, 0] = f(x_data, y_data).to(device) # Wrong input
107
108         # Storing input data in tensor
109         input_tensor = torch.empty(size=(nr_of_data, 2)).to(device)
110         input_tensor[:, 0] = x_data; input_tensor[:, 1] = y_data
111
112         # Split the data into training and testing
113         train_input, train_target, test_input, test_target = train_test_split(
                input_tensor, z_data)
114
115         # Loop through all models and train them for this seed
116         for model_idx, model_factory in enumerate(model_factories):
117             model = model_factory()
118             optimizer = torch.optim.AdamW(model.parameters(), lr = learning_rate,
                        weight_decay = weight_decay)
119             # Train the model
120             model.batch_train_model(x = train_input, targets = train_target,
                        optimizer = optimizer, loss_criterion = loss_criterion, nr_epochs =
                         epochs, print_loss_each = (epochs // 2), batch_size = batch_size,
                        plot_loss = True, mse_plot_limit = 0.2)
121             # Test the model
122             running_loss = model.test_model(test_input, test_target, loss_criterion
                         = torch.nn.L1Loss())
123             seed_loss[seed] += running_loss.item()
124             mean_loss[model_idx] += running_loss.item() / nr_of_seeds
125
126     # Printing and returning the loss
127     print(f"The mean test loss for {nr_of_seeds} seeds is {mean_loss}")
128     print(f"The seed loss is {seed_loss}")
129     return mean_loss
130
131
132 # Executing the analysis, saving all the models
133 model_factories = [
134          lambda: alexKAN.sequential(
135         alexKAN.functional(nr_in=2, nr_out=2, k=3, t=t_6),
136         alexKAN.functional(nr_in=2, nr_out=1, k=3, t=t_6)
137     ),
138     lambda: alexKAN.sequential(
139         alexKAN.functional(nr_in=2, nr_out=4, k=3, t=t_6),
140         alexKAN.functional(nr_in=4, nr_out=1, k=3, t=t_6)
141     ),
142     lambda: alexKAN.sequential( # The iconic 2-5-1 structure - 8 basis functions
143         alexKAN.functional(nr_in=2, nr_out=5, k=3, t=t_6),
144         alexKAN.functional(nr_in=5, nr_out=1, k=3, t=t_6)
145     ),
146     lambda: alexKAN.sequential(
147         alexKAN.functional(nr_in=2, nr_out=3, k=3, t=t_6),
148         alexKAN.functional(nr_in=3, nr_out=5, k=3, t=t_6),
149         alexKAN.functional(nr_in=5, nr_out=1, k=3, t=t_6)
150     ),
151     lambda: alexKAN.sequential(
152         alexKAN.functional(nr_in=2, nr_out=5, k=3, t=t_6),
153         alexKAN.functional(nr_in=5, nr_out=9, k=3, t=t_6),
154         alexKAN.functional(nr_in=9, nr_out=1, k=3, t=t_6)
155     ),
156     lambda: alexKAN.sequential(
157         alexKAN.functional(nr_in=2, nr_out=10, k=3, t=t_6),
158         alexKAN.functional(nr_in=10, nr_out=10, k=3, t=t_6),
159         alexKAN.functional(nr_in=10, nr_out=1, k=3, t=t_6)
160     ),
161     lambda: alexKAN.sequential( # more parameters per basis function
162         alexKAN.functional(nr_in=2, nr_out=7, k=3, t=t_14),
163         alexKAN.functional(nr_in=7, nr_out=6, k=3, t=t_14),
164         alexKAN.functional(nr_in=6, nr_out=1, k=3, t=t_14)
165     ),
166 ]
167
168 # Define main function
```

```
def main(nr_of_seeds = 10, parameters = [48, 96, 208, 512, 968], nr_of_data = 10000, batch_size = 250):
    """
    Description:
    Runs the training and test, and plotting the losses for all the networks.

    Arguments:
    nr_of_seeds : The amount of seeds the model should be trained and tested over
    batch_size : The batch size used for training the model
    parameters : Number of parameters for all the models trained.
    nr_of_data : Number of data for generation.

    Returns:

    """
    mean_loss = train_and_test(nr_of_seeds = nr_of_seeds, nr_of_data = nr_of_data)
    plt.figure()
    plt.title(f"Medelabsolutfel (MAE) f r {nr_of_seeds} $\mathit{{seeds}}$")
    plt.plot(parameters, mean_loss.detach().cpu().numpy(), marker="*", label="MAE som funktion av antal parametrar")
    plt.xlim([0, 1.1 * max(parameters)])
    plt.ylim([0, 1.1 * max(mean_loss.cpu())])
    plt.legend()
    plt.grid()
    plt.xlabel("Antal parametrar")
    plt.ylabel("MAE")
    plt.show()

main(nr_of_seeds = 10, nr_of_data = nr_of_data, batch_size = batch_size)


"""
The following code is for image regression analysis.
"""

# Import necessary packages
from PIL import Image, ImageOps
import numpy as np
import torch
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
import torch
import math
import pandas as pd
from torch.utils.data.dataset import random_split
import importlib
import alexKAN

# Reload, when modifying it, it needs to be reloaded every time
importlib.reload(alexKAN)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Read the picture
nr_pixels = 128
im = Image.open("/content/pepper.png") # Change the path of the file
im = im.resize((nr_pixels, nr_pixels))
im = ImageOps.grayscale(im)
pix = im.load()

# Make a numpy array and a torch tensor out of the image, normalized grayscale values
im_array = np.array(im) / 255 # Normalize the greyscale value
im_tensor = torch.tensor(im_array).to(device)
print(im_tensor)

# Show the image
plt.imshow(im, cmap = "grey")
plt.axis('off')
plt.show()

# Construct the data to the form we want it in
im_height, im_width = im_tensor.size()[0], im_tensor.size()[1]
im_input = torch.empty([im_height * im_width, 2], dtype = torch.float64).to(device)
im_target = torch.empty([im_height * im_width, 1], dtype = torch.float64).to(device)

# Remake it to a tensor suitable for the KAN library
for y, row in enumerate(im_tensor):
    for x, pix in enumerate(row):

        # Setting all the x, y and z values in the correct spot
        im_input[y * im_width + x, 0] = x / im_width # x values
        im_input[y * im_width + x, 1] = y / im_height # y values
```

```
            im_target[y * im_width + x, 0] = pix.item()


    # Running the analysis - create an analysis python script

    # Defining the domain
    a = -2
    b = 2

    # Model parameters
    nr_of_knots = 6
    t = torch.linspace(a, b, nr_of_knots).to(device)
    t_14 = torch.linspace(a, b, 14).to(device)

    k = 3 # degree
    weight_decay = 1e-2 # Default weight_decay - l2 regularization
    learning_rate = 0.001
    epochs = 500
    batch_size = 250


    def train_and_test_image(nr_of_seeds = 1, parameters = [480, 1000, 2000, 4960,
        10080]): # should take the model size as well
        """
        Description:
        Trains and test the image regression model, for some specified KAN models.

        Arguments:
        nr_of_seeds : The number of seeds used to train the model
        parameters : Number of parameters for all models trained.

        Returns:

        """

        # Model optimizer
        loss_criterion = torch.nn.MSELoss()
        mean_loss = torch.zeros(5).to(device)

        # Do this over a number of seeds, for the image experiment it is only done over
            one
        for seed in range(nr_of_seeds):

            # Reset the model
            print(f"Seed {seed}")
            torch.manual_seed(seed) # setting the seed
            best_nr_epcohs_list = []

            # Reset the model, train on the optimal amount and test it
            for model_idx, model_factory in enumerate(model_factories):
                model = model_factory()
                optimizer = torch.optim.AdamW(model.parameters(), lr = learning_rate,
                    weight_decay = weight_decay)

                # Batch-train the model
                model.batch_train_model(x = im_input, targets = im_target, optimizer =
                    optimizer, loss_criterion = loss_criterion, nr_epochs = epochs,
                    print_loss_each = (epochs // 2),batch_size = batch_size, plot_loss
                    = True, mse_plot_limit = 0.2)
                # Test the model - sort the pixels back into a 2d array
                pred_image = torch.zeros([nr_pixels, nr_pixels]).to(device)
                pred = model.forward(x = im_input)
                for pix_idx, pix in enumerate(pred):
                    pix_x, pix_y = int(im_input[pix_idx, 0] * nr_pixels), int(im_input[
                        pix_idx, 1] * nr_pixels)
                    pred_image[pix_y, pix_x] = pix * 255

                # Test the loss for the model
                mae_loss = model.test_model(x = im_input, targets = im_target)
                mean_loss[model_idx] = mae_loss
                print(f"The MAE for the image is {mae_loss} after {epochs} epochs with
                    {parameters[model_idx]} parameters")

                # Show the predicted image
                print(f"Image for model {parameters[model_idx]} parameters\n")
                pred_image_np = pred_image.detach().cpu().numpy()
                plt.imshow(pred_image_np, cmap = 'grey')
                plt.show()

    # Defining all the models used
    model_factories = [
        lambda: alexKAN.sequential(
            alexKAN.functional(nr_in=2, nr_out=5, k=3, t=t),
            alexKAN.functional(nr_in=5, nr_out=4, k=3, t=t),
```

```
            alexKAN.functional(nr_in=4, nr_out=6, k=3, t=t),
            alexKAN.functional(nr_in=6, nr_out=1, k=3, t=t)
        ),
        lambda: alexKAN.sequential(
            alexKAN.functional(nr_in=2, nr_out=8, k=3, t=t),
            alexKAN.functional(nr_in=8, nr_out=8, k=3, t=t),
            alexKAN.functional(nr_in=8, nr_out=5, k=3, t=t),
            alexKAN.functional(nr_in=5, nr_out=1, k=3, t=t)
        ),
        lambda: alexKAN.sequential(
            alexKAN.functional(nr_in=2, nr_out=10, k=3, t=t),
            alexKAN.functional(nr_in=10, nr_out=11, k=3, t=t),
            alexKAN.functional(nr_in=11, nr_out=10, k=3, t=t),
            alexKAN.functional(nr_in=10, nr_out=1, k=3, t=t)
        ),
        lambda: alexKAN.sequential(
            alexKAN.functional(nr_in=2, nr_out=10, k=3, t=t),
            alexKAN.functional(nr_in=10, nr_out=20, k=3, t=t),
            alexKAN.functional(nr_in=20, nr_out=13, k=3, t=t),
            alexKAN.functional(nr_in=13, nr_out=10, k=3, t=t),
            alexKAN.functional(nr_in=10, nr_out=1, k=3, t=t)
        ),
        lambda: alexKAN.sequential(
            alexKAN.functional(nr_in=2, nr_out=20, k=3, t=t),
            alexKAN.functional(nr_in=20, nr_out=20, k=3, t=t),
            alexKAN.functional(nr_in=20, nr_out=20, k=3, t=t),
            alexKAN.functional(nr_in=20, nr_out=20, k=3, t=t),
            alexKAN.functional(nr_in=20, nr_out=1, k=3, t=t)
        ),
    ]

    # Define main function
    def main(nr_of_seeds = 1, parameters = [480, 1000, 2000, 4960, 10080]):
        train_and_test_image(nr_of_seeds = nr_of_seeds)
        """
        Description:
        Runs the image regression analysis for all KAN models.

        Arguments:
        nr_of_seeds : The number of seeds used to train the model
        parameters : Number of parameters for all models trained.

        Returns:
        """

    main(nr_of_seeds = 1)


    # The following code is for reconstruction of Peppers in different dimensions

    # The largest image has to be broken down because the memory of the GPU cannot
        handle that large of a tensor
    # hence the 512x512 image will be broken into 4 pieces, where the following
        function splits the image
    # It also selects one of the for to input to the model

    def image_selector(image_nr_x, image_nr_y, nr_of_subimages_per_dimension = 2):
        """
        Description:
        The function splits a large image into sub-parts. The number of sub-parts is 4.
             The purpose of this function is to divide the 512x512 image into four
            subparts

        Arguments:
        image_nr_x : which of the sub-images in x (width) direction
        image_nr_y : which of the sub-images in y (height) direction
        nr_of_sub_images_per_dimension : The number of images per dimension.

        Returns:
        test_im_input : Returns the sub-image of the specified location.


        """

      nr_of_images = nr_of_subimages_per_dimension ** 2
      nr_pixels = 512
      pixel_split = nr_pixels // nr_of_subimages_per_dimension
      test_im_input = torch.empty([pixel_split * pixel_split, 2], dtype = torch.float64
          ).to(device)
      for y in range(pixel_split * (image_nr_y), pixel_split * (image_nr_y + 1)):
        for x in range(pixel_split * (image_nr_x), pixel_split * (image_nr_x + 1)):
          local_x, local_y = x - pixel_split * image_nr_x, y - pixel_split * image_nr_y
               # Move all indiced to first quadrant
          idx = local_y * pixel_split + local_x
```

```
        # Setting all the x, y and z values in the correct spot
        test_im_input[idx, 0] = x / nr_pixels # x values
        test_im_input[idx, 1] = y / nr_pixels # y values
    print(test_im_input)
    return test_im_input


# The following reconstructs all the images for a 36100 parameter KAN and saves the image in a csv file for the 512x512 image

# Defining the domain
a = -2
b = 2

# Model parameters
nr_of_knots = 5
t = torch.linspace(a, b, nr_of_knots).to(device)

k = 3 # degree
weight_decay = 1e-2 # Default weight_decay - l2 regularization
learning_rate = 0.001
epochs = 500
batch_size = 250


def reconstruction_test(nr_of_seeds = 1, image_reconstruction_dimensions = [32, 64, 128, 256, 512]): # should take the model size as well
    """
    Description:
    This function reconstructs the image for the specified dimensions, trained on a specified KAN model.

    Arguments:
    nr_of_seeds : Number of seeds the training should be carried out on.
    image_reconstruction_dimensions : Defines the dimensions for which the image should be reconstructed from.

    Returns:

    """
    # Model optimizer
    loss_criterion = torch.nn.MSELoss()

    for seed in range(nr_of_seeds):

        # Reset the model
        print(f"Seed {seed}")
        torch.manual_seed(seed) # setting the seed
        best_nr_epcohs_list = []

        # Reset the model, train on the optimal amount and test it
        for model_idx, model_factory in enumerate(model_factories):
            model = model_factory()
            optimizer = torch.optim.AdamW(model.parameters(), lr = learning_rate, weight_decay = weight_decay)

            # Batch-train the model
            model.batch_train_model(x = im_input, targets = im_target, optimizer = optimizer, loss_criterion = loss_criterion, nr_epochs = epochs, print_loss_each = 10,batch_size = batch_size, plot_loss = True, mse_plot_limit = 0.2, testing_data = im_input, testing_targets = im_target)

            # Test the model - construct multiple images
            for dimension_idx, im_dimension in enumerate(
                image_reconstruction_dimensions):
              if im_dimension == 512:
              # Split the image into 4 sub-images
                nr_splits = 2
                nr_pixel_per_split = im_dimension // nr_splits

                image_x = 1
                image_y = 1
                test_im_input = image_selector(image_x, image_y, nr_of_subimages_per_dimension = nr_splits)
              else: # if not
                test_im_input = torch.empty([im_dimension * im_dimension, 2], dtype = torch.float64).to(device)
                for y in range(im_dimension):
                  for x in range(im_dimension):
                    # Setting all the x, y and z values in the correct spot
                    test_im_input[y * im_dimension + x, 0] = x / im_dimension # x values
                    test_im_input[y * im_dimension + x, 1] = y / im_dimension # y
```

```
                    values
                if im_dimension == 512:
                # Fix indexing - the current tensor, and re-organize it to fit the
                    local indices
                  test_im_input_local = test_im_input.clone()
                  test_im_input_local[:, 0] = (test_im_input[:, 0] * nr_splits) % 1
                  test_im_input_local[:, 1] = (test_im_input[:, 1] * nr_splits) % 1

                  # Define the image number
                  image_nr = image_x + image_y * nr_splits

                  pred_image_test = torch.zeros([nr_pixel_per_split,
                      nr_pixel_per_split]).to(device)
                else:
                  pred_image_test = torch.zeros([im_dimension, im_dimension]).to(
                      device)
                pred = model.forward(x = test_im_input)

                # Loop through all the pixels and replace them in their original
                    location
                for pix_idx, pix in enumerate(pred):
                  pix_x = int(test_im_input[pix_idx, 0] * im_dimension)
                  pix_y = int(test_im_input[pix_idx, 1] * im_dimension)
                  pred_image_test[pix_y, pix_x] = pix * 255
                pred_image_np = pred_image_test.detach().cpu().numpy()
                df = pd.DataFrame(pred_image_np)
                if im_dimension == 512: # Save the largest image in order to merge it
                  df.to_csv(f"image_{dimension_idx}.csv", index=False)
                  plt.imsave(f"image_{dimension_idx}_interpolated.png", pred_image_np
                      , cmap = "gray")

                # Show the predicted image
                print(f"Image for model {im_dimension} parameters\n")
                plt.imshow(pred_image_np, cmap = 'grey')
                plt.show()


# Define the only model
model_factories = [
    lambda: alexKAN.sequential(
        alexKAN.functional(nr_in=2, nr_out=41, k=3, t=t),
        alexKAN.functional(nr_in=41, nr_out=50, k=3, t=t),
        alexKAN.functional(nr_in=50, nr_out=43, k=3, t=t),
        alexKAN.functional(nr_in=43, nr_out=20, k=3, t=t),
        alexKAN.functional(nr_in=20, nr_out=1, k=3, t=t)

    ),
]

# Define main function
def main(nr_of_seeds = 1):
    reconstruction_test(nr_of_seeds = nr_of_seeds, split = False)

main(nr_of_seeds = 1)

# The following script merges all the sub-images that are stored in a dataframe

sub_image_0 = pd.read_csv("sub_image_0.csv")
sub_image_1 = pd.read_csv("sub_image_1.csv")
sub_image_2 = pd.read_csv("sub_image_2.csv")
sub_image_3 = pd.read_csv("sub_image_3.csv")
top_image = pd.concat([sub_image_0, sub_image_1], axis = 1)
bottom_image = pd.concat([sub_image_2, sub_image_3], axis = 1)
reconstructed_image = pd.concat([top_image, bottom_image], axis = 0)
# reconstructed_image.to_csv("restored_512_image.csv")
image_array = reconstructed_image.to_numpy()
import cv2
import numpy as np

array = reconstructed_image.to_numpy()
plt.figure()
plt.imshow(array, cmap = "grey")
plt.axis("off")
plt.show()
```

C.1.4 Skript 4: Analys av MLP

```
# Define the MLP structures that are wanted
function_networks_matrix = [
    [2, 3, 4, 4, 1],
    [2, 5, 6, 6, 1],
    [2, 9, 10, 6, 1],
```

```
    [2, 15, 16, 11, 1],
    [2, 15, 27, 18, 1]
]

image_regression_networks_matrix = [
    [2, 15, 16, 11, 1],
    [2, 15, 27, 18, 1],
    [2, 25, 36, 26, 1],
    [2, 45, 57, 38, 1],
    [2, 56, 83, 60, 1]
]

# Re-define them as tensors
function_networks = torch.tensor(function_networks_matrix).to(device)
image_regression_networks = torch.tensor(image_regression_networks_matrix).to(
    device)

# Building a function which trains and test the neural network
def train_and_test(train_input, train_target, test_input, test_target, model,
    nr_epochs = 10, batch_size = 250, print_test = False, print_test_every_epoch =
    5, classification_task = False, loss_criterion = torch.nn.MSELoss(),
    best_testing_accuracy = 0):

    """
    Description:
    Trains and test the neural network models, whos structures are user-specified.
        It also plots heatmaps if it is a classification task.

    Arguments:
    train_input : Training data input.
    train_target : Training target input.
    test_input : Testing data input.
    test_target : Testing target input.
    model : The model that should be used for training.
    nr_epochs : The number of epochs the model should be trained for.
    batch_size : The batch size the model should be trained for
    print_test : True / False statement if the test results should be printed.
        print_test_every_epoch : Define when the loss should be printed.
        classification_task : Defines if it is a classification task or not.
    loss_criterion : Defines what loss function should be used for optimization.
        best_testing_accuracy : Defines the best model accuracy during training,
        for evaluation of network.

    Returns:
    test_loss : Returns the test loss for the model when evaluated.
    """

    # Define which optimizer that should be used
    optimizer = torch.optim.AdamW(model.parameters(), lr = 0.001, weight_decay =
        0.01)

    # Define which loss criterion should be used for testing the model
    test_loss_criterion = torch.nn.L1Loss()

    # Define the number of data and the number of batches
    nr_data = train_input.size()[0]
    nr_batches = int(math.ceil(nr_data / batch_size))
    for epoch in range(nr_epochs):

        # Shuffle the indices
        shuffled_indices = torch.randperm(nr_data).to(device)

        for batch_idx in range(nr_batches):

            # Define the batch indices
            start_index = batch_idx * batch_size
            end_index = min((batch_idx + 1) * batch_size, nr_data)
            current_batch_indices = shuffled_indices[start_index : end_index].to(
                device)
            current_batch_input, current_batch_target = train_input[
                current_batch_indices].to(device), train_target[
                current_batch_indices].to(device)

            # Propagation through the network
            current_batch_predictions = model(current_batch_input)
            loss = loss_criterion(current_batch_predictions, current_batch_target)

            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()


        # Test the model
```

```
77          test_predictions = model(test_input)
78          test_loss = test_loss_criterion(test_predictions, test_target)
79
80          # Fix a heatmap for the classification
81          if classification_task:
82
83            # Define that we do not want to update weights when evaluating
84            with torch.no_grad():
85              # Sigmoid, which BCELossWithLogits does internally
86              predictions = torch.sigmoid(model.forward(test_input))
87
88              # Separate which are correctely predicted
89              predicted_labels = (predictions >= 0.5).float()
90              targets = test_target.view(-1, 1).float()
91
92              # Count the total correct predicted
93              total_correct = (predicted_labels == targets).sum().item()
94              total_samples = targets.size(0)
95
96              # Calculate the accuracy
97              if total_samples != 0:
98                  model_accuracy = total_correct / total_samples
99                  print(f"The model accuracy of test images is {model_accuracy *
                        100:.2f}%")
100
101             # Only want to show the best testing accuracy
102             if model_accuracy > best_testing_accuracy or epoch + 1 == nr_epochs:
103                 best_testing_accuracy = max(model_accuracy, best_testing_accuracy)
104
105                 # Extract the predicted values to the cpu, needs to constructed to
                        numpy array
106                 preds = predicted_labels.cpu()
107                 truth = targets.cpu()
108
109                 # Count True Positive, True Negative, False Positive and False
                        Negative
110                 TP = ((preds == 1) & (truth == 1)).sum().item()
111                 TN = ((preds == 0) & (truth == 0)).sum().item()
112                 FP = ((preds == 1) & (truth == 0)).sum().item()
113                 FN = ((preds == 0) & (truth == 1)).sum().item()
114
115                 # Construct a confusion matrix
116                 conf_matrix = np.array([[TP, FP], [FN, TN]])
117
118                 # Define the labels for the axises
119                 labels = ['1', '0']
120
121                 # Construct a heatmap
122                 plt.figure()
123                 sns.heatmap(conf_matrix, annot = True, fmt = "d", cmap = "Greens",
                        xticklabels = labels, yticklabels = labels)
124                 plt.xlabel("Predikterad etikett")
125                 plt.ylabel("Etikett")
126                 plt.show()
127
128
129
130         if (print_test) and ((epoch + 1) % print_test_every_epoch == 0):
131             print(f"The test MAE after {epoch + 1} epochs is {test_loss}")
132
133     # Return the final loss
134     return test_loss
135
136 # Construct the data that is used for the model
137
138
139 # Metrics - define for where the data should be gathered from for the function
        regression
140 seed = 0 # loop over seeds
141 a = -2
142 b = 2
143 nr_of_data = 10000
144
145 def get_data(seed, a, b, nr_of_data):
146     """
147     Description:
148     Generates the data for function regression, with a specific seed. The function
            for which analysis is carried out for is defined in the get_data function.
149
150     Arguments:
151     seed : Defines the seed for which the data should be randomly generated for.
152     a : Defines the lower limit for which the data should be randomly generated for
            .
153     b : Defines the upper limit for which the data should be randomly generated for
```

```
        .
154     nr_of_data : Defines the number of data that should be generated.
155
156     Returns :
157     train_input : The splitted training input data.
158     train_target : The splitted training target data.
159     test_input : The splitted testing input data.
160     test_target : The splitted testing target data.
161     """
162     torch.manual_seed(seed) # setting the seed
163
164     # Get the x and y data randomly over the interval
165     x_data = (a - b) * torch.rand(nr_of_data).to(device) + b
166     y_data = (a - b) * torch.rand(nr_of_data).to(device) + b
167
168     # Targets
169     z_data = torch.empty(size = (nr_of_data, 1), dtype = torch.float64).to(device)
170     f = lambda x, y: 4 * x * y ** 2
171     z_data[:, 0] = f(x_data, y_data).to(device) # Wrong input
172
173     # Storing input data in tensor
174     input_tensor = torch.empty(size=(nr_of_data, 2)).to(device)
175     input_tensor[:, 0] = x_data; input_tensor[:, 1] = y_data
176
177     # Split the data into training and testing
178     train_input, train_target, test_input, test_target = train_test_split(
            input_tensor, z_data)
179     return train_input, train_target, test_input, test_target
180
181 train_input, train_target, test_input, test_target = get_data(seed = seed, a = a, b
        = b, nr_of_data = nr_of_data)
182
183 # Define the main loop for the model - the training loop
184
185 def main(train_input_dataset, train_target_dataset, test_input_dataset,
        test_target_dataset, nr_epochs = 10, parameter_list = [], avg_test_losses =
        torch.zeros(len(function_networks)).to(device), analysis = "NaN", networks =
        function_networks, nr_of_seeds = 1):
186     # Loop over seeds
187     """
188     Description:
189     The function is the main training loop for the function regression analysis.
190
191     Arguments:
192     train_input_dataset : The train input dataset.
193     train_target_dataset : The train target dataset.
194     test_input_dataset : The test input dataset.
195     test_target_dataset : The test target dataset.
196     nr_epochs : Defines how many epochs the network should be trained over.
197     parameter_list : Defines the amount of parameters that the models have, used
            for plotting purposes.
198     avg_test_losses : stores all the losses of the functions
199     analysis : Defines which analysis the model is running, for printing purposes.
200     networks : defines the array of models that are used for this experiment.
201     nr_of_seeds : Defines how many seeds the models should be trained on for
            comparison.
202
203     Return:
204     avg_test_losses : The test losses for the functions.
205
206     """
207     for seed in range(nr_of_seeds):
208         torch.manual_seed(seed = seed)
209         print(f"Seed {seed}")
210         # Creating an image
211
212         for network_idx, network in enumerate(function_networks):
213
214             # Constructing the torch model
215             layers = []
216
217             for i in range(len(network) - 2):
218                 nr_layer_input_neurons, nr_layer_output_neurons = int(network[i]),
                        int(network[i + 1])
219                 layers.append(torch.nn.Linear(nr_layer_input_neurons,
                        nr_layer_output_neurons))
220                 layers.append(torch.nn.ReLU())
221
222             layers.append(torch.nn.Linear(int(network[-2]), int(network[-1]))) #
                    Final layer without activation
223
224             # Establishing the model for the network
225             model = torch.nn.Sequential(*layers).to(device)
226
```

```
            # Training and testing all models
            test_loss = train_and_test(train_input = train_input_dataset.float(),
                train_target = train_target_dataset.float(), test_input =
                test_input_dataset.float(), test_target = test_target_dataset.float
                (), model = model, nr_epochs = nr_epochs, batch_size = 250,
                print_test = False, print_test_every_epoch = 5)
            avg_test_losses[network_idx] += test_loss / nr_of_seeds

            # Showing a predicted image
            if analysis == "image regression":
                print(f"The image for {parameter_list[network_idx]} parameters")
                plt.figure()
                forwarded_image = model(test_input_dataset.float())
                predicted_image = np.zeros([nr_pixels, nr_pixels])
                for pix_idx, pix in enumerate(forwarded_image):
                        pix_x, pix_y = int(im_input[pix_idx, 0] * nr_pixels), int(
                            im_input[pix_idx, 1] * nr_pixels)
                        predicted_image[pix_y, pix_x] = pix * 255
                plt.imshow(predicted_image, cmap = 'grey')
                plt.show()



    # Printing the results
    print(f"The {analysis} analysis test losses are {avg_test_losses}")

    # Plotting the results
    if len(parameter_list) == len(avg_test_losses):
        plt.figure()
        plt.plot(parameter_list, avg_test_losses.cpu().detach(), label = f"Average
            test losses for {nr_of_seeds} seeds ")
        plt.legend()
        plt.grid()
        plt.show()

    # Return the average loss
    return avg_test_losses

# Metrics
function_parameter_list = [50, 100, 200, 500, 1000]
image_regression_parameter_list = [500, 1000, 2000, 5000, 10000]
nr_epochs = 500
nr_of_seeds = 10

function_test_losses = main(train_input_dataset = train_input, train_target_dataset
     = train_target, test_input_dataset = test_input, test_target_dataset =
    test_target, nr_epochs = nr_epochs, parameter_list = function_parameter_list,
    analysis = "function regression", networks = function_networks, nr_of_seeds =
    nr_of_seeds)
```

C.1.5 Skript 5: Klassificering av KAN och MLP

```

# Install the Pneumonia MNIST dataset
!pip install -q medmnist

# Importing libraries
import medmnist
from medmnist import PneumoniaMNIST
from medmnist import INFO
import importlib
import alexKAN
importlib.reload(alexKAN)

# PyTorch imports
import torch
from torchvision import transforms
from torch.utils.data import DataLoader, ConcatDataset
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Choose task and download
data_flag = 'pneumoniamnist'
download = True

# Define the information for the dataset, and load it with a data class
info = INFO[data_flag]
DataClass = getattr(medmnist, info['python_class'])

# Define transforms
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[.5], std=[.5])  # Since grayscale
```

```
])

# Load the different dataset
train_dataset = DataClass(split = 'train', transform = transform, download = download)
val_dataset   = DataClass(split = 'val', transform = transform, download = download)
test_dataset  = DataClass(split = 'test', transform = transform, download = download)
dataset = ConcatDataset([train_dataset, val_dataset, test_dataset])

# For exploratory purposes
print(f"Number of training samples: {len(train_dataset)}")
print(f"Number of validation samples: {len(val_dataset)}")
print(f"Number of testing samples: {len(test_dataset)}")

# Reconstruct the data in the form we want it in - suitable for the KAN networks
image_length = len(train_dataset[0][0][0].flatten())

# Split the images and their labels
pneumonia_data_set = torch.empty([len(dataset), image_length], dtype = torch.float64).to(device)
pneumonia_labels = torch.empty([len(dataset), 1], dtype = torch.float64).to(device)
for image in range(len(dataset)):
    pneumonia_data_set[image, :] = dataset[image][0][0].flatten()
    pneumonia_labels[image, 0] = torch.tensor(dataset[image][1], dtype = pneumonia_labels.dtype, device = pneumonia_labels.device)

# Train and test splut of the dataset
pneumonia_data_set_train = torch.empty([len(train_dataset), image_length], dtype = torch.float64).to(device)
pneumonia_labels_train = torch.empty([len(train_dataset), 1], dtype = torch.float64).to(device)
pneumonia_data_set_test = torch.empty([len(test_dataset), image_length], dtype = torch.float64).to(device)
pneumonia_labels_test = torch.empty([len(test_dataset), 1], dtype = torch.float64).to(device)

# Construct it to the form required for the KAN model
for image in range(len(train_dataset)):
    pneumonia_data_set_train[image, :] = train_dataset[image][0][0].flatten()
    pneumonia_labels_train[image, 0] = torch.tensor(train_dataset[image][1], dtype = pneumonia_labels_train.dtype, device = pneumonia_labels_train.device)
for image in range(len(test_dataset)):
    pneumonia_data_set_test[image, :] = test_dataset[image][0][0].flatten()
    pneumonia_labels_test[image, 0] = torch.tensor(test_dataset[image][1], dtype = pneumonia_labels_test.dtype, device = pneumonia_labels_test.device)

# Defining the domain
a = 0
b = 1

# Model parameters - define the different knot intervals
t_5 = torch.linspace(a, b, 5).to(device)
t_11 = torch.linspace(a, b, 11).to(device)
t_17 = torch.linspace(a, b, 17).to(device)

k = 3 # degree
weight_decay = 1e-2 # Default weight_decay - l2 regularization
learning_rate = 0.001
epochs = 100
batch_size = 250


def pneumonia_mnist_classification(nr_of_seeds = 1): # should take the model size as well
    """
    Description:
    The defined model is trained on the Pneumonia MNIST data set. This function loops through all KAN models and trains them on the data set.

    Arguments:
    nr_of_seeds : Defines how many seeds the models should be trained on for comparison.

    Return:
    None
    """
    # Define the classification loss function
    loss_criterion = torch.nn.BCEWithLogitsLoss()

    # Loop through all the seeds
    for seed in range(nr_of_seeds):
        # Reset the model
```

```
        print(f"Seed {seed}")
        torch.manual_seed(seed) # setting the seed

        # Reset the model, train on the optimal amount and test it
        for model_idx, model_factory in enumerate(model_factories):
            print("============================")
            print(f"Model {model_idx + 1}")
            model = model_factory()
            optimizer = torch.optim.AdamW(model.parameters(), lr = learning_rate,
                weight_decay = weight_decay)

            # Batch-train the model
            model.batch_train_model(x = pneumonia_data_set_train, targets =
                pneumonia_labels_train, optimizer = optimizer, loss_criterion =
                loss_criterion, nr_epochs = epochs, print_loss_each = 1, batch_size
                 = batch_size, plot_loss = True, mse_plot_limit = 0.7,
                classification_task = True, testing_data = pneumonia_data_set_test,
                 testing_targets = pneumonia_labels_test)

            # Test the model - sort the pixels back into a 2d array
            mae_loss = model.test_model(x = pneumonia_data_set_test, targets =
                pneumonia_labels_test, classification_task = True)
            print(f"The MAE for the image is {mae_loss} after {epochs} epochs with
                {nr_parameters} parameters")

# Defines the used models
model_factories = [
    lambda: alexKAN.sequential(
        alexKAN.functional(nr_in = 28 * 28, nr_out=1, k=3, t=t_5),
    ),
    lambda: alexKAN.sequential(
        alexKAN.functional(nr_in = 28 * 28, nr_out=1, k=3, t=t_11),
    ),
    lambda: alexKAN.sequential(
        alexKAN.functional(nr_in = 28 * 28, nr_out=1, k=3, t=t_17),
    ),
]

# Define main function
def main(nr_of_seeds = 1):
    """
    Description:
    The defined model is trained on the Pneumonia MNIST data set and runs the
        analysis.

    Arguments:
    nr_of_seeds : Defines how many seeds the models should be trained on for
        comparison.

    Return:
    """
    pneumonia_mnist_classification(nr_of_seeds = nr_of_seeds)

main(nr_of_seeds = 1)

# This runs the analysis for a CNN
import torch.nn as nn
import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from torch.utils.data import DataLoader

# Reload the data
train_loader = DataLoader(train_dataset, batch_size=len(train_dataset))
test_loader = DataLoader(test_dataset, batch_size=len(test_dataset))

# Load all data in one batch
train_data_iter = iter(train_loader)
test_data_iter = iter(test_loader)

# Split the data to their respective parts
images_train, labels_train = next(train_data_iter)
images_test, labels_test = next(test_data_iter)

# Convert to float - threw an error previously
images_train = images_train.float().to(device)
images_test = images_test.float().to(device)
labels_train = labels_train.float().to(device)
labels_test = labels_test.float().to(device)

# Define all the CNNs
CNN_5857 = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=0),
```

```
    nn.ReLU(),
    nn.MaxPool2d(2, 2),

    nn.Conv2d(32, 16, kernel_size=3, stride=1, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),

    nn.Flatten(),
    nn.Linear(400, 1)
).to(device)

CNN_10881 = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Flatten(),
    nn.Linear(800, 1)
).to(device)


CNN_15905 = nn.Sequential(
    nn.Conv2d(1, 48, kernel_size=5, stride=1, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),
    nn.Conv2d(48, 32, kernel_size=3, stride=1, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),

    nn.Flatten(),
    nn.Linear(800, 1)
).to(device)

# Define the hyperparameters for the model and specify the loss function. Weighted
    penalization due to being an imbalanced dataset.
num_parameters = sum(p.numel() for p in CNN_5857.parameters() if p.requires_grad)
nr_epochs = 400
pos_weight = torch.tensor([3439 / 1245]).to(device)
loss_criterion = torch.nn.BCEWithLogitsLoss(pos_weight = pos_weight)
best_5857_accuracy = 0
best_10881_accuracy = 0
best_15905_accuracy = 0

# Run the train_and_test function for all the CNNs
print(f"CNN 5857 ====================================================")
test_loss_5857 = train_and_test(train_input = images_train, train_target =
    labels_train, test_input = images_test, test_target = labels_test, model =
    CNN_5857, nr_epochs = nr_epochs, batch_size = 250, print_test = False,
    print_test_every_epoch = 5, classification_task = True, loss_criterion =
    loss_criterion, best_testing_accuracy = 0)
print(f"CNN 10881 ===================================================")
test_loss_10881 = train_and_test(train_input = images_train, train_target =
    labels_train, test_input = images_test, test_target = labels_test, model =
    CNN_10881, nr_epochs = nr_epochs, batch_size = 250, print_test = False,
    print_test_every_epoch = 5, classification_task = True, loss_criterion =
    loss_criterion, best_testing_accuracy = 0)
print(f"CNN 10881 ====================================================")
test_loss_15905 = train_and_test(train_input = images_train, train_target =
    labels_train, test_input = images_test, test_target = labels_test, model =
    CNN_15905, nr_epochs = nr_epochs, batch_size = 250, print_test = False,
    print_test_every_epoch = 5, classification_task = True, loss_criterion =
    loss_criterion, best_testing_accuracy = 0)
```

C.1.6 Skript 6: plottning av grafer

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

#values for plot between sprcher
#values = [4.47525, 4.31478, 3.36936, 3.03245, 1.57732, 1.49841, 0.19069, 0.17672,
    0.17338]

#Number of parameters
valuesx = [[48, 96, 120, 208, 512, 968, 992],
           [31, 49, 99, 195, 492, 1021, 1022],
           [52, 63, 101, 195, 503, 1005, 1015],
           [50, 100, 200, 500, 1000]]

```

```
13
14  #values for the number of parameters
15  valuesy = [[0.4312, 0.1935, 0.1338, 0.3445, 0.2055, 0.0586, 0.0749],
16             [0.11127, 0.09501, 0.18789, 0.05243, 0.02892, 0.0304, 0.02613],
17             [0.12059, 0.0666, 0.06913, 0.05574, 0.02836, 0.01585, 0.02525],
18             [3.4583, 0.8944, 0.1867, 0.0979, 0.0735] ]
19  colors = np.array(["palevioletred","cyan","lightseagreen","rosybrown","
        mediumslateblue","yellowgreen","hotpink","purple","darksalmon"])
20  names = ["Org", "Multi", "Brown", "Single", "96Pro", "Special", "Classical", "
        Lambda", "Comb"]
21
22  #Uncomment the following code to plot all 9 implementations.
23  """
24  for i in range(9):
25      plt.scatter(x=[i+1], y=[values[i]], c=colors[i], label=f'Plot {i+1}')
26      plt.text(x=i+1, y=values[i]+0.1, s=names[i])
27      plt.xticks(color='w')
28
29
30  plt.title("Skillnader i fel mellan olika implementationer av Sprechers sats")
31  plt.xlabel("Implementation")
32  plt.ylabel("MAE")
33  """
34  """
35  #Uncomment the following code to plot the three implementations that gave the best
        results zoomed in.
36
37  plt.subplot(xlim=(0.5, 3.5), ylim=(0.16, 0.2))
38  for i in range(3):
39      plt.scatter(x=[i+1], y=[values[i+6]], c=colors[i+6], label=f'Plot {i+6}')
40      plt.text(x=i+1, y=values[i+6]+0.0005, s=names[i+6])
41      plt.xticks(color='w')
42
43  plt.title("F rstoring av de tre b st presterande implementationerna")
44  plt.xlabel("Implementation")
45  plt.ylabel("MAE")
46  plt.show()
47  """
48  #Uncomment the following code to print MAE
49  #for i in range(7):
50  #    plt.scatter(x=[i+1], y=[values[i]], c=colors[i], label=f'Plot {i+1}')
51  #    plt.text(x=i+1, y=values[i]+0.1, s=names[i])
52
53
54  #plt.title("Olika implementationer av Sprechers sats")
55  #plt.xlabel("Implementation")
56  #plt.ylabel("MAE")
57
58  titel =["KAN", "SprecherLambda", "SprecherComb", "MLP"]
59
60
61  plt.figure()
62  plt.title(f"Medelabsolutfel (MAE) f r bildregressionen")
63  for i in range (4):
64   plt.plot(valuesx[i], valuesy[i], marker = "o", label = titel[i])
65  plt.xlim([0, 1.1 * 1000])
66  plt.ylim([0, 1.1 * 0.45])
67  plt.legend()
68  plt.grid()
69  plt.xlabel("Antal parametrar")
70  plt.ylabel("MAE")
71  plt.show()
72
73
74  """
75      plt.figure()
76      plt.title(f"Medelabsolutfel (MAE) f r {nr_of_seeds} $\mathit{{seeds}}$")
77      plt.plot(parameters, mean_loss.detach().numpy(), marker = "*", label = "MAE som
            funktion av antal parametrar")
78      plt.xlim([0, 1.1 * max(parameters)])
79      plt.ylim([0, 1.1 * max(mean_loss)])
80      plt.legend()
81      plt.grid()
82      plt.xlabel("Antal parametrar")
83      plt.ylabel("MAE")
84      plt.show()
85
86  main(nr_of_seeds = 1)
87  """
```

C.2 SKAN

C.2.1 Skript 1: Funktionsregression SKAN

```
#imports:
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import rand
import pandas as pd
from functools import partial
import random
torch.cuda.empty_cache()
#-----------------------------------------------------------------
# Setting up data:

def train_test_split(input, target, test_fraction = 0.2):

    """
    Train test split function:
    Arguments:
    input : The input tensor data.
    target : The target tensor data.
    test_fraction : The fraction of the dataset in the test data.

    Return:
    train_input : The training input to the network.
    train_target : The corresponding training target.
    test_input : The testing input to the network.
    test_target : The corresponding testing target.

    """

    # Metrics for the data
    data_length = input.size()[0]
    train_size = int((1 - test_fraction) * data_length) # Generate the train size

    # Random permutation of the tensors when merged to keep track of which inputs
        correspond to the same outputs
    perm = torch.randperm(data_length)
    input, target = input[perm, :], target[perm, :]

    # Divide permutation based on training size
    train_input, train_target = input[:train_size, :], target[:train_size, :]
    test_input, test_target = input[train_size:, :], target[train_size:, :]

    # Return the train and test tensors
    return train_input, train_target, test_input, test_target
seed=0
a = -2
b = 2
nr_of_data = 10000

def get_data(seed, a, b, nr_of_data):
    torch.manual_seed(seed) # setting the seed
    x_data = (a - b) * torch.rand(nr_of_data) + b
    y_data = (a - b) * torch.rand(nr_of_data) + b

    # Targets
    z_data = torch.empty(size = (nr_of_data, 1), dtype = torch.float64)
    f = lambda x, y: 4 * x * y ** 2
    z_data[:, 0] = f(x_data, y_data) # Wrong input

    # Storing input data in tensor
    input_tensor = torch.empty(size=(nr_of_data, 2))
    input_tensor[:, 0] = x_data; input_tensor[:, 1] = y_data

    # Split the data into training and testing
    train_input, train_target, test_input, test_target = train_test_split(
        input_tensor, z_data)
    return train_input, train_target, test_input, test_target

train_input, train_target, test_input, test_target = get_data(seed = seed, a = a, b
     = b, nr_of_data = nr_of_data)

# Setting seed manually:
def set_seed(seed):
    torch.manual_seed(seed)  # Set the seed for CPU operations
    np.random.seed(seed)  # Set the seed for numpy
    random.seed(seed)  # Set the seed for Python's random module
```

```
77
78      if torch.cuda.is_available():
79          torch.cuda.manual_seed(seed)  # Set seed for CUDA
80          torch.cuda.manual_seed_all(seed)  # Set seed for all GPUs
81          torch.backends.cudnn.deterministic = True  # Ensures deterministic behavior
82          torch.backends.cudnn.benchmark = False  # Disables optimization that can
                introduce randomness
83
84  # Example usage
85  set_seed(0)
86
87  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
88  print(device)
89  #------------------------------------------------------------------
90  #defining models:
91  class SprecherMulti(nn.Module):
92      def __init__(self,structure,batch,degree,num_knots):
93          super().__init__()
94          self.degree = degree
95          self.batch = batch
96          self.structure = structure
97          self.knots = torch.arange(-5, 5, step = 10/num_knots,dtype=torch.float32)
98          self.num_c = self.knots.size(0)-degree-1 #number of paramaters per
                acitivisionfunction
99          self.std = 0.1
100         self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
                structure)-1,self.num_c))) # paramters for each funciton
101         self.lambdas = nn.Parameter(torch.normal(mean=0, std=self.std, size=(sum(
                structure[:len(structure)-2]),))) #Lambda values for inner forumla
102         self.eps =nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
                structure)-2,))) #epsilon values outer formula
103
104
105     def layer_pass(self,x,layer):
106         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+( self
                .eps[layer] * torch.tensor([[i for i in range(self.structure[layer+1])
                ]]).float()).repeat(self.batch, self.structure[layer], 1) # init tensor
                 and adding input and translating with epislon times q
107         functions = self.apply_BSpline(functions,self.cs[layer]) * self.lambdas[sum
                (self.structure[:layer]):sum(self.structure[:layer+1])].view(self.
                structure[layer], 1) # applying function and multiplying with lambda
108         result = (torch.sum(functions,dim=1, keepdim=True)+ torch.tensor([[i for i
                in range(self.structure[layer+1])]]).float()).squeeze(1) # adding outer
                 q
109         return result
110     #Layerpass takes input and creates tensor for the outputs, by filling tensor
            with inputs and paramaters acc to formula and then apply the learnble
            activtion function
111
112     def final_pass(self,x):
113         result = torch.sum(self.apply_BSpline(x,self.cs[len(self.structure)-2]),dim
                =1)
114         return result
115     #apply the final function on second to last layer output
116
117     def forward(self,x):  #apply layerpasses for first n-2 such and the last apply
            general function pass acc to formula.
118         input = x
119         for layer in range(len(self.structure)-2):
120             input = self.layer_pass(input,layer)
121         result = self.final_pass(input).view(self.batch,1)
122         return result
123
124     def b_spline(self, knot_index, degree, t, knots):
125         """
126         Takes in a knot_index for the spline, starting, loop through the knots -
                degree - 1
127         """
128         if degree == 0:
129             # Convert boolean result to float (0 or 1)
130             return ((knots[knot_index] <= t) & (t < knots[knot_index + 1])).float()
131
132         else:
133             # Handling denominators for recursive B-spline calculation
134             denom1 = knots[knot_index + degree] - knots[knot_index] if (knot_index
                    + degree) < len(knots) else 0
135             denom2 = knots[knot_index + degree + 1] - knots[knot_index + 1] if (
                    knot_index + degree + 1) < len(knots) else 0
136
137             term1 = torch.zeros_like(t)  # Initialize term1 tensor
138             term2 = torch.zeros_like(t)  # Initialize term2 tensor
139
140             if denom1 != 0:
```

```
                term1 = ((t - knots[knot_index]) / denom1) * self.b_spline(
                    knot_index, degree - 1, t, knots)
            if denom2 != 0:
                term2 = ((knots[knot_index + degree + 1] - t) / denom2) * self.
                    b_spline(knot_index + 1, degree - 1, t, knots)

            return term1 + term2
    def BSpline(self,c,knot_interval,degree,t):
        sum = 0
        for i in range(len(c)):
            sum = sum + c[i]*self.b_spline(i,degree,t,knot_interval)  #is this
                correct? no?!!! let it be for now!
        return sum

    def apply_BSpline(self,tensor,c):
        return self.BSpline(c, self.knots, self.degree, tensor)

class SprecherSpecial(nn.Module):
    def __init__(self,structure,batch,degree,num_knots):
        super().__init__()
        self.degree = degree
        self.batch = batch
        self.structure = structure
        self.knots = torch.arange(-5,5, step = 10/num_knots,dtype=torch.float32)
        self.num_c = self.knots.size(0)-degree-1
        self.std = 0.1
        self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
            structure)-1,self.num_c)))
        self.lambdas = nn.Parameter(torch.normal(mean=0, std=1, size=(sum(structure
            [:len(structure)-2]),)))
        self.eps = nn.Parameter(torch.normal(mean=0, std=1, size=(len(structure)
            -2,)))
        #for Xavier init:
        sum_in_out_nodes =[structure[i]+structure[i+1] for i in range(len(structure
            )-1)]
        self.std_list = torch.tensor([self.xavier(x) for x in sum_in_out_nodes],
            dtype=torch.float32)
        xavier = torch.normal(0,self.std_list)
        ones = torch.ones(len(structure)-1)
        weights = torch.stack([ones,xavier],dim=1)
        self.weights = nn.Parameter(weights)
        self.layer = None
        self.dodo = nn.Parameter(torch.ones(len(structure)-2)) #optimizable factor
            scaling the outer q-term in layer function
        self.epz = nn.Parameter(torch.abs(torch.normal(mean=0, std=0.1, size=(sum(
            structure[1:len(structure)-1]),),))) # positive init f r peronalized
            epsilon, one per node in hidden layers
        # for easy calculation of indexing when using epz in layer funciton:
        o = structure.copy()
        del o[0]
        self.structure2 = o

    def xavier(self,x): #for Xavier init
        std = np.sqrt(2/(x))
        return std

    def layer_pass(self,x,layer):
        functions = torch.zeros((self.batch,self.structure[layer],self.structure[
            layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+(self.
            epz[sum(self.structure2[:layer]):sum(self.structure2[:layer+1])]).
            repeat(self.batch, self.structure[layer], 1)
        functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
            lambdas[sum(self.structure[:layer]):sum(self.structure[:layer+1])].view
            (self.structure[layer], 1) # is this ok broadcasting
            ?-----------------------------------
        result = (torch.sum(functions,dim=1, keepdim=True)+ self.dodo[layer]*torch.
            tensor([[i for i in range(self.structure[layer+1])]]).float()).squeeze
            (1)
        self.layer = result #saving last layer data
        return result

    def final_pass(self,x):
        result = torch.sum(self.phi(x,self.cs[len(self.structure)-2],self.weights[
            len(self.structure)-2]),dim=1)
        return result

    def forward(self,x):
        input = x
        for layer in range(len(self.structure)-2):
            input = self.layer_pass(input,layer)
        result = self.final_pass(input).view(self.batch,1)
        return result
```

```
205     def b_spline(self, knot_index, degree, t, knots):
206         """
207         Takes in a knot_index for the spline, starting, loop through the knots -
                degree - 1
208         """
209         if degree == 0:
210             # Convert boolean result to float (0 or 1)
211             return ((knots[knot_index] <= t) & (t < knots[knot_index + 1])).float()
212 
213         else:
214             # Handling denominators for recursive B-spline calculation
215             denom1 = knots[knot_index + degree] - knots[knot_index] if (knot_index
                    + degree) < len(knots) else 0
216             denom2 = knots[knot_index + degree + 1] - knots[knot_index + 1] if (
                    knot_index + degree + 1) < len(knots) else 0
217 
218             term1 = torch.zeros_like(t)  # Initialize term1 tensor
219             term2 = torch.zeros_like(t)  # Initialize term2 tensor
220 
221             if denom1 != 0:
222                 term1 = ((t - knots[knot_index]) / denom1) * self.b_spline(
                        knot_index, degree - 1, t, knots)
223             if denom2 != 0:
224                 term2 = ((knots[knot_index + degree + 1] - t) / denom2) * self.
                        b_spline(knot_index + 1, degree - 1, t, knots)
225 
226             return term1 + term2
227 
228 
229 
230     def BSpline(self,c,knot_interval,degree,t):
231         sum = 0
232         for i in range(len(c)):
233             sum = sum + c[i]*self.b_spline(i,degree,t,knot_interval)
234         return sum
235 
236     def phi(self,tensor,c,weight): # weighted activision function
237         return weight[0]*self.BSpline(c, self.knots, self.degree, tensor) + weight
                [1]*nn.functional.silu(tensor)
238 
239     def silu(self,x):
240         return x * (1 / (1 + np.exp(-x)))
241 
242     def plot_layer_function(self,layer,x): # for plotting inner functions
243         y = self.phi(x,self.cs[layer],self.weights[layer])
244         x = x.tolist()
245         y = y.tolist()
246         plt.figure()
247         plt.plot(x,y)
248         plt.show()
249 
250     def plot_final_function(self,layer,x):
251         y = self.phi(x,self)
252 
253 class SprecherClassical(nn.Module): #classical 2-5-1 KAN using Sprecher
        infrastructure
254     def __init__(self,structure,batch,degree,num_knots):
255         super().__init__()
256         self.degree = degree
257         self.batch = batch #batch_size
258         self.structure = structure
259         self.knots = torch.arange(-5, 5, step = 10/num_knots,dtype=torch.float32)
260         self.num_c = self.knots.size(0)-degree-1
261         self.std = 0.1
262         self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(10,self.
                num_c)))
263         self.lambdas = nn.Parameter(torch.normal(mean=0, std=self.std, size=(sum(
                structure[:len(structure)-2]),)))
264         self.eps =nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
                structure)-2,)))
265         self.final_cs = nn.Parameter(torch.normal(mean=0,std=self.std,size = (
                structure[len(structure)-2],self.num_c)))
266 
267 
268     def layer_pass(self,x,layer):
269         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])
270         s = 0
271         for i in range(2):
272             for k in range(5):
273                 functions[:,i,k] = self.apply_BSpline(functions[:,i,k],self.cs[s])
274                 s= s+1
275         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
276         return result
```

```
277
278         #Layerpass takes input and creates tensor for the outputs, by filling tensor
                with inputs and paramaters acc to formula and then apply the learnble
                activtion function
279
280         def final_pass(self,x):
281             result = torch.zeros(size=(self.batch,self.structure[len(self.structure)
                    -2]))
282             for i in range(self.structure[len(self.structure)-2]):
283                 res = self.apply_BSpline(x[:,i] ,self.final_cs[i])
284                 result[:,i] = res.view(self.batch)
285             return torch.sum(result,dim=1)
286
287
288         def forward(self,x):  #apply layerpasses for first n-2 such and the last apply
                general function pass acc to formula.
289             input = x
290             for layer in range(len(self.structure)-2):
291                 input = self.layer_pass(input,layer)
292             result = self.final_pass(input).view(self.batch,1)
293             return result
294
295         def b_spline(self, knot_index, degree, t, knots): #Whats t?
                ----------------------------
296             """
297             Takes in a knot_index for the spline, starting, loop through the knots -
                    degree - 1
298             """
299             if degree == 0:
300                 # Convert boolean result to float (0 or 1)
301                 return ((knots[knot_index] <= t) & (t < knots[knot_index + 1])).float()
302
303             else:
304                 # Handling denominators for recursive B-spline calculation
305                 denom1 = knots[knot_index + degree] - knots[knot_index] if (knot_index
                        + degree) < len(knots) else 0
306                 denom2 = knots[knot_index + degree + 1] - knots[knot_index + 1] if (
                        knot_index + degree + 1) < len(knots) else 0
307
308                 term1 = torch.zeros_like(t)  # Initialize term1 tensor
309                 term2 = torch.zeros_like(t)  # Initialize term2 tensor
310
311                 if denom1 != 0:
312                     term1 = ((t - knots[knot_index]) / denom1) * self.b_spline(
                            knot_index, degree - 1, t, knots)
313                 if denom2 != 0:
314                     term2 = ((knots[knot_index + degree + 1] - t) / denom2) * self.
                            b_spline(knot_index + 1, degree - 1, t, knots)
315
316                 return term1 + term2
317         def BSpline(self,c,knot_interval,degree,t):
318             sum = 0
319             for i in range(len(c)):
320                 sum = sum + c[i]*self.b_spline(i,degree,t,knot_interval)
321             return sum
322
323         def apply_BSpline(self,tensor,c):
324             return(self.BSpline(c, self.knots, self.degree, tensor))
325
326     class SprecherLambda(SprecherSpecial):
327         def __init__(self, structure, batch, degree, num_knots):
328             super().__init__(structure, batch, degree, num_knots)
329             self.lambda_matrices = nn.ParameterList([nn.Parameter(torch.normal(mean=0,
                    std=self.std, size=(structure[i],structure[i+1]))) for i in range(len(
                    structure)-2)])
330
331         def layer_pass(self,x,layer):
332             functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                    layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+(self.
                    epz[sum(self.structure2[:layer]):sum(self.structure2[:layer+1])]).
                    repeat(self.batch, self.structure[layer], 1)
333             functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                    lambda_matrices[layer].unsqueeze(0)
334             result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
335             self.layer = result
336             return result
337
338     class Sprecher96Pro(SprecherSpecial): # layer pass redone and new parameters added
            for the extra activisionfunctions in the last layer
339         def __init__(self, structure, batch, degree, num_knots):
340             super().__init__(structure, batch, degree, num_knots)
341             self.final_cs = nn.Parameter(torch.normal(mean=0,std=self.std,size = (
                    structure[len(structure)-2],self.num_c)))
342             self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
```

```
            structure)-2,self.num_c)))
343         ones = torch.ones(structure[len(structure)-2]-1)
344         print(self.std_list[-1])
345         std_list = self.std_list[-1].expand(ones.size(0))
346         xavier = torch.normal(0,std_list)
347         weights = torch.stack([ones,xavier],dim=1)
348         self.final_weights = nn.Parameter(weights)
349
350     def layer_pass(self,x,layer):
351         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+(self.
                epz[sum(self.structure2[:layer]):sum(self.structure2[:layer+1])]).
                repeat(self.batch, self.structure[layer], 1)
352         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambdas[sum(self.structure[:layer]):sum(self.structure[:layer+1])].view
                (self.structure[layer], 1) # is this ok broadcasting
                ?-----------------------------------
353         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
354         self.layer = result
355         return result
356
357     def final_pass(self,x):
358         result = torch.zeros(size=(self.batch,self.structure[len(self.structure)
                -2]))
359         result[:,0] =  self.phi(x[:,0] ,self.final_cs[0],self.weights[-1])
360         for i in range(self.structure[len(self.structure)-2]-1):
361             res = self.phi(x[:,i] ,self.final_cs[i],self.final_weights[i])
362             result[:,i+1] = res.view(self.batch)
363         return torch.sum(result,dim=1)
364
365 class SprecherComb(Sprecher96Pro):
366     def __init__(self, structure, batch, degree, num_knots):
367         super().__init__(structure, batch, degree, num_knots)
368         self.lambda_matrices = nn.ParameterList([nn.Parameter(torch.normal(mean=0,
                std=self.std, size=(structure[i],structure[i+1]))) for i in range(len(
                structure)-2)])
369
370     def layer_pass(self,x,layer):
371         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+(self.
                epz[sum(self.structure2[:layer]):sum(self.structure2[:layer+1])]).
                repeat(self.batch, self.structure[layer], 1)
372         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambda_matrices[layer].unsqueeze(0)
373         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
374         self.layer = result
375         return result
376
377 class SprecherSingle(SprecherSpecial): # model with one single lambda for each
        hidden layer
378     def __init__(self, structure, batch, degree, num_knots):
379         super().__init__(structure, batch, degree, num_knots)
380         self.lambdas = nn.Parameter(torch.normal(mean=0,std=self.std,size=(len(
                structure)-2,)))
381
382     def layer_pass(self,x,layer):
383         lambdas =  torch.full((self.structure[layer],self.structure[layer+1]),self.
                lambdas[layer].item())
384         for row in range(lambdas.size(0)):
385             for col in range(lambdas.size(1)):
386                 lambdas[row,col] = lambdas[row,col]**((row+1)*col)
387         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+( self
                .eps[layer] * torch.tensor([[i for i in range(self.structure[layer+1])
                ]]).float()).repeat(self.batch, self.structure[layer], 1)
388         functions = self.phi(functions,self.cs[layer],self.weights[layer]) *
                lambdas.unsqueeze(0)
389         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
390         self.layer = result
391         return result
392
393 class SprecherOrg(SprecherSpecial): # model based on orginal Sprecher theorem, one
        lmbda per layer, with p as exponent
394     def __init__(self, structure, batch, degree, num_knots):
395         super().__init__(structure, batch, degree, num_knots)
396         self.lambdas = nn.Parameter(torch.normal(mean=0,std=self.std,size=(len(
                structure)-2,)))
397
398     def layer_pass(self,x,layer):
399         lambdas =  torch.full((self.structure[layer],self.structure[layer+1]),self.
                lambdas[layer].item())
400         for row in range(lambdas.size(0)):
401             lambdas[row,:] = lambdas[row,:]**((row+1))
402         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
```

```
            layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+( self
            .eps[layer] * torch.tensor([[i for i in range(self.structure[layer+1])
            ]]).float()).repeat(self.batch, self.structure[layer], 1)
        functions = self.phi(functions,self.cs[layer],self.weights[layer]) *
            lambdas.unsqueeze(0)
        result = (torch.sum(functions,dim=1, keepdim=True)+ torch.tensor([[i for i
            in range(self.structure[layer+1])]]).float()).squeeze(1)
        self.layer = result
        return result

class SprecherBrown(Sprecher96Pro):
    def __init__(self, structure, batch, degree, num_knots):
        super().__init__(structure, batch, degree, num_knots)
        self.lambdas = nn.Parameter(torch.normal(mean=0,std=self.std,size=(len(
            structure)-2,)))

    def layer_pass(self,x,layer):
        n = self.structure[layer]
        eps = 1/((2*n+1)*(2*n+2))
        lambdas =  torch.full((self.structure[layer],self.structure[layer+1]),self.
            lambdas[layer].item())
        for row in range(lambdas.size(0)):
            lambdas[row,:] = lambdas[row,:]**((row+1))
        functions = torch.zeros((self.batch,self.structure[layer],self.structure[
            layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+(eps *
             torch.tensor([[i for i in range(self.structure[layer+1])]]).float()).
            repeat(self.batch, self.structure[layer], 1)
        functions = self.phi(functions,self.cs[layer],self.weights[layer]) *
            lambdas.unsqueeze(0)
        result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
        self.layer = result
        return result
#-----------------------------------------------------------------
# Training and testing the model:

# set up:
from torch.optim.lr_scheduler import StepLR
batch_size = 250

model =  SprecherLambda([2,25,20,15,1],batch_size,3,30) #choose model, strucutre,
    batchsize, degree and number of knots of the knotvector
model = model.to(device)

train_input, train_target, test_input, test_target = get_data(seed = 9, a = a, b =
    b, nr_of_data = nr_of_data)
dataset_train = TensorDataset(train_input, train_target)
dataset_test = TensorDataset(test_input,test_target)


train_loader = DataLoader(dataset_train, batch_size=batch_size, shuffle=False,
    drop_last=True)
test_loader = DataLoader(dataset_test,batch_size=batch_size, shuffle=False,
    drop_last=True)

test_fraction = 0.2


dataset_length = nr_of_data
train_size = int(dataset_length * (1 - test_fraction))
test_size = dataset_length - train_size  # Ensures the sum matches dataset length
loss_function = nn.MSELoss()
optimizer = optim.AdamW(model.parameters(),lr = 0.01)
scheduler = StepLR(optimizer, step_size=25, gamma=0.75)
num_epochs = 200
test_losses = []

# epoch train and test loop
for epoch in range(num_epochs):
    model.train()
    running_loss = 0
    for data,target in train_loader:
        optimizer.zero_grad() # reset gradients for new training epoch
        prediction = model(data) # model prediction
        loss = loss_function(prediction,target) # loss of prediction
        loss.backward() # backpropagation
        optimizer.step() # updating paramaters in model based on backpropagation
        running_loss += loss.item()
    print('running training loss: ', running_loss)
    scheduler.step()
    model.eval() # evalueate model mode
    test_loss = 0
    with torch.no_grad(): #test loss i MAE
        for data, target in test_loader:
            data,target = data.to(device), target.to(device)
```

```
            prediction = model(data)
            loss = torch.sum(torch.abs(prediction-target)) # mae loss
            test_loss += loss.item() # counting mae loss
    test_losses.append(test_loss)

    print('avg_test_loss: ', test_loss/((test_size//batch_size)*batch_size))
    print('epoch ',epoch+1 , 'done :)')
```

C.2.2 Skript 2: Bildregression och rekonstruktion SKAN

```

#imports:
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
from torch.cuda.amp import autocast, GradScaler

import os
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import rand
import pandas as pd
#from sklearn.model_selection import train_test_split
from functools import partial
import random
torch.cuda.empty_cache()

def set_seed(seed):
    torch.manual_seed(seed)  # Set the seed for CPU operations
    np.random.seed(seed)  # Set the seed for numpy
    random.seed(seed)  # Set the seed for Python's random module

    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)  # Set seed for CUDA
        torch.cuda.manual_seed_all(seed)  # Set seed for all GPUs
        torch.backends.cudnn.deterministic = True  # Ensures deterministic behavior
        torch.backends.cudnn.benchmark = False  # Disables optimization that can
            introduce randomness

# Example usage
set_seed(0)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
#
    ----------------------------------------------------------------------------------------------------------------

# defining models
class SprecherSpecial(nn.Module):
    def __init__(self,structure,batch,degree,num_knots):
        super().__init__()
        self.degree = degree
        self.batch = batch
        self.structure = structure
        self.knots = torch.arange(-2,2, step = 4/num_knots,dtype=torch.float32,
            device=device)
        self.num_c = self.knots.size(0)-degree-1
        self.std = 0.1
        self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
            structure)-1,self.num_c)))
        self.lambdas = nn.Parameter(torch.normal(mean=0, std=1, size=(sum(structure
            [:len(structure)-2]),)))
        self.eps = nn.Parameter(torch.normal(mean=0, std=1, size=(len(structure)
            -2,)))
        #for Xavier init
        sum_in_out_nodes =[structure[i]+structure[i+1] for i in range(len(structure
            )-1)]
        self.std_list = torch.tensor([self.xavier(x) for x in sum_in_out_nodes],
            dtype=torch.float32,device=device)
        xavier = torch.normal(0,self.std_list)
        ones = torch.ones(len(structure)-1,device = device)
        weights = torch.stack([ones,xavier],dim=1)
        self.weights = nn.Parameter(weights)
        self.layer = None
        self.dodo = nn.Parameter(torch.ones(len(structure)-2))
        self.epz = nn.Parameter(torch.abs(torch.normal(mean=0, std=0.1, size=(sum(
            structure[1:len(structure)-1]),),))) # epsilon values for each hidden
            node
        # for simpler indexing in layer pass function:
        o = structure.copy()
```

```
            del o[0]
            self.structure2 = o

        def xavier(self,x): #for Xavier init
            std = np.sqrt(2/(x))
            return std

        def layer_pass(self,x,layer):
            functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1]),device=device) + x.unsqueeze(2).repeat(1,1, self.structure[
                layer+1])+(self.epz[sum(self.structure2[:layer]):sum(self.structure2[:
                layer+1])]).repeat(self.batch, self.structure[layer], 1)
            functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambdas[sum(self.structure[:layer]):sum(self.structure[:layer+1])].view
                (self.structure[layer], 1)
            result = (torch.sum(functions,dim=1, keepdim=True)+ self.dodo[layer]*torch.
                tensor([[i for i in range(self.structure[layer+1])]],evice=device).
                float()).squeeze(1)
            self.layer = result
            return result

        def final_pass(self,x):
            result = torch.sum(self.phi(x,self.cs[len(self.structure)-2],self.weights[
                len(self.structure)-2]),dim=1)
            return result

        def forward(self,x):
            input = x
            for layer in range(len(self.structure)-2):
                input = self.layer_pass(input,layer)
            result = self.final_pass(input).view(self.batch,1)
            return result


        def basis_function(self, degree, knots, t): #non recursive faster basis
            function calculations
          """
          Compute all non-zero B-spline basis functions of a given degree for input
              tensor `t`.

          Returns: (t.shape, num_basis_functions) tensor of basis evaluations.
          """
          num_basis = len(knots) - degree - 1
          t = t.unsqueeze(-1)  # Shape: [*, 1] to [*, num_basis]

          # Initialize degree 0 basis functions (step functions)
          B = ((knots[:-1] <= t) & (t < knots[1:])).float()  # Shape: [..., num_basis +
               degree]

          for d in range(1, degree + 1):
              left = (t - knots[:-d-1]) / (knots[d:-1] - knots[:-d-1] )
              right = (knots[d+1:] - t) / (knots[d+1:] - knots[1:-d] )

              left_term = left * B[..., :-1]
              right_term = right * B[..., 1:]
              B = left_term + right_term

          return B  # Shape: [..., num_basis]


        def BSpline(self, c, knot_interval, degree, t):
          """
          Evaluate spline at t using basis function matrix and coefficients c.

          Args:
              c: (num_basis,) coefficient tensor
              knot_interval: knot vector (tensor)
              degree: degree of spline
              t: input tensor, shape [...].

          Returns:
              Spline values at t, shape: [...]
          """
          B = self.basis_function(degree, knot_interval, t)  # Shape [..., num_basis]
          return torch.sum(B * c, dim=-1)  # Weighted sum over basis


        def phi(self, tensor, c, weight):
          """
          Final phi function, compatible with existing code.
          """
          spline_part = self.BSpline(c, self.knots, self.degree, tensor)
          silu_part = nn.functional.silu(tensor)
          return weight[0] * spline_part + weight[1] * silu_part
```

```
135
136
137     def plot_layer_function(self,layer,x):
138         y = self.phi(x,self.cs[layer],self.weights[layer])
139         x = x.tolist()
140         y = y.tolist()
141         plt.figure()
142         plt.plot(x,y)
143         plt.show()
144
145
146 #   ------------------------------------------------------------------------------------------------------------
147
148 class Sprecher96Pro(SprecherSpecial):
149     def __init__(self, structure, batch, degree, num_knots):
150         super().__init__(structure, batch, degree, num_knots)
151         self.final_cs = nn.Parameter(torch.normal(mean=0,std=self.std,size = (
                structure[len(structure)-2],self.num_c)))
152         self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
                structure)-2,self.num_c)))
153         ones = torch.ones(structure[len(structure)-2]-1,device = device)
154         std_list = self.std_list[-1].expand(ones.size(0))
155
156         xavier = torch.normal(0,std_list)
157         weights = torch.stack([ones,xavier],dim=1)
158         self.final_weights = nn.Parameter(weights)
159
160     def layer_pass(self,x,layer):
161         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+(self.
                epz[sum(self.structure2[:layer]):sum(self.structure2[:layer+1])]).
                repeat(self.batch, self.structure[layer], 1)
162         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambdas[sum(self.structure[:layer]):sum(self.structure[:layer+1])].view
                (self.structure[layer], 1) # is this ok broadcasting
                ?-----------------------------------
163         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
164         self.layer = result
165         return result
166
167     def final_pass(self,x):
168         result = torch.zeros(size=(self.batch,self.structure[len(self.structure)
                -2]),device=device)
169         result[:,0] =  self.phi(x[:,0] ,self.final_cs[0],self.weights[-1])
170         for i in range(self.structure[len(self.structure)-2]-1):
171             res = self.phi(x[:,i] ,self.final_cs[i],self.final_weights[i])
172             result[:,i+1] = res.view(self.batch)
173         return torch.sum(result,dim=1)
174
175 #   ------------------------------------------------------------------------------------------------------------
176 class SprecherLambda(SprecherSpecial):
177     def __init__(self, structure, batch, degree, num_knots):
178         super().__init__(structure, batch, degree, num_knots)
179         self.lambda_matrices = nn.ParameterList([nn.Parameter(torch.normal(mean=0,
                std=self.std, size=(structure[i],structure[i+1]))) for i in range(len(
                structure)-2)])
180
181     def layer_pass(self,x,layer):
182         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1]),device = device) + x.unsqueeze(2).repeat(1,1, self.structure[
                layer+1])+(self.epz[sum(self.structure2[:layer]):sum(self.structure2[:
                layer+1])]).repeat(self.batch, self.structure[layer], 1)
183         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambda_matrices[layer].unsqueeze(0)
184         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
185         self.layer = result
186         return result
187
188 class SprecherComb(Sprecher96Pro):
189     def __init__(self, structure, batch, degree, num_knots):
190         super().__init__(structure, batch, degree, num_knots)
191         self.lambda_matrices = nn.ParameterList([nn.Parameter(torch.normal(mean=0,
                std=self.std, size=(structure[i],structure[i+1]))) for i in range(len(
                structure)-2)])
192         self.knots = torch.arange(-5,5, step = 10/num_knots,dtype=torch.float32,
                device=device)
193
194
195     def layer_pass(self,x,layer):
196         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
```

```
                layer+1]),device = device) + x.unsqueeze(2).repeat(1,1, self.structure[
                layer+1])+(self.epz[sum(self.structure2[:layer]):sum(self.structure2[:
                layer+1])]).repeat(self.batch, self.structure[layer], 1)
197         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambda_matrices[layer].unsqueeze(0)
198         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
199         self.layer = result
200         return result
201
202     def final_pass(self,x):
203         result = torch.zeros(size=(self.batch,self.structure[len(self.structure)
                -2]),device=device)
204         result[:,0] =  self.phi(x[:,0] ,self.final_cs[0],self.weights[-1])
205         for i in range(self.structure[len(self.structure)-2]-1):
206             res = self.phi(x[:,i] ,self.final_cs[i],self.final_weights[i])
207             result[:,i+1] = res.view(self.batch)
208         return torch.sum(result,dim=1)
209
210 #
        ----------------------------------------------------------------------------------------------------

211 #imorting and setting up image data for training:
212 from PIL import Image, ImageOps
213 nr_pixels = 64
214 im = Image.open("pepper.png")
215 im = im.resize((nr_pixels, nr_pixels))
216 im = ImageOps.grayscale(im)
217 pix = im.load()
218
219 im_array = np.array(im)
220 im_tensor = torch.tensor(im_array,dtype=torch.float32, device=device) / 255 #
        normalize
221
222 #
        ----------------------------------------------------------------------------------------------------

223 x,y = torch.meshgrid(torch.arange(nr_pixels)/nr_pixels,torch.arange(nr_pixels)/
        nr_pixels,indexing= 'ij') # kan vara d ligt med linspace
224
225 x = x.flatten()
226 y = y.flatten()
227
228 pixel_coords = torch.stack((x,y),dim=1)
229
230
231 df = pd.DataFrame({'x':x.ravel(),'y':y.ravel(),'z' : (im_array/255).ravel()})
232 input_cols = ["x", "y"]
233 target_cols = ["z"]
234
235 x = x.to(device)
236 y = y.to(device)
237 pixel_coords = torch.stack((x,y),dim=1)
238
239
240 def dataframe_to_tensor(df, input_cols, target_cols):
241     inputs = torch.tensor(df[input_cols].values, dtype=torch.float32,device =
            device)
242     targets = torch.tensor(df[target_cols].values, dtype=torch.float32,device =
            device)
243     return inputs, targets
244
245 inputs, targets = dataframe_to_tensor(df, input_cols, target_cols)
246 dataset = TensorDataset(inputs, targets)
247
248 pixel_vals = im_tensor.flatten().unsqueeze(1)
249
250 #
        ----------------------------------------------------------------------------------------------------

251 # setting up training and testing
252 from torch.optim.lr_scheduler import StepLR
253 batch_size = 500
254
255 model =  SprecherLambda([2,200,100,80,80,80,60,40,1],batch_size,3,20) #choose model
        , strucutre,, spline degree, batchsize and number of knots per knotvector
256
257
258 model = model.to(device)
259
260 test_fraction = 0.2
261
262 dataset_length = len(dataset)
263 train_size = int(dataset_length * (1 - test_fraction))
264 test_size = dataset_length - train_size  # Ensures the sum matches dataset length
```

```
train, test = random_split(dataset, [train_size, test_size])
train_loader = DataLoader(train, batch_size=batch_size, shuffle=True, drop_last=True, pin_memory = False)
test_loader = DataLoader(test, batch_size=batch_size, drop_last=True, pin_memory=False)
loss_function = nn.MSELoss()
optimizer = optim.AdamW(model.parameters(),lr = 0.01,weight_decay = 0.03)
scheduler = StepLR(optimizer, step_size=20, gamma=0.82)
num_epochs = 200
test_losses = []


for epoch in range(num_epochs):
    model.train()
    running_loss = 0
    for (data,target) in train_loader:
        optimizer.zero_grad()
        prediction = model(data)
        loss = loss_function(prediction,target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print('running training loss: ', running_loss)
    scheduler.step()
    model.eval()
    test_loss = 0
    with torch.no_grad(): #test loss in MAE
        for data, target in test_loader:
            data,target = data.to(device), target.to(device)
            prediction = model(data)
            loss = torch.sum(torch.abs(prediction-target))
            test_loss += loss.item()
    test_losses.append(test_loss)

    print('avg_test_loss: ', test_loss/((test_size//batch_size)*batch_size))
    print('epoch ',epoch+1 , 'done :)')


#
    ------------------------------------------------------------------------------------------------------------------------

# print model parameters
for name, param in model.named_parameters():
    print(f"Parameter: {name}, Shape: {param.shape}")
    print(param)
print('layer\n',model.layer)
#
    ------------------------------------------------------------------------------------------------------------------------

#save produced image by doing partwise prediction caluclations and reassebling the image in order to save Vram usage
matplotlib.use('Agg')  # Use non-interactive backend

# Create output directory if it doesn't exist
output_dir = "saved_images"
os.makedirs(output_dir, exist_ok=True)
batch_size = 1024
pixels = [256,512] # pixel dimensions for reconstruciton

for p in pixels:
    nr_pixels = p
    model.batch = batch_size
    x,y = torch.meshgrid(torch.arange(nr_pixels)/nr_pixels,torch.arange(nr_pixels)/nr_pixels,indexing= 'ij')
    x = x.flatten().to(device)
    y = y.flatten().to(device)

    pixel_coords = torch.stack((x,y),dim=1)

    output_chunks = []
    with torch.no_grad():
        for i in range(0, len(pixel_coords), batch_size):
            batch = pixel_coords[i:i+batch_size]
            output = model(batch)
            output_chunks.append(output)


    image = torch.cat(output_chunks, dim=0).view(nr_pixels, nr_pixels) # reassemble


# generate random filename without affecting global seed
    r = random.SystemRandom().randint(0, 10000)  # Uses system entropy, unaffected by torch or numpy seeds
    name = f"Lambda {r} {p}.png"
```

```
340
341 # save to output directory
342     save_path = os.path.join(output_dir, name)
343
344     array = image.detach().cpu().numpy()
345
346     # save image
347     plt.imsave(save_path, array, cmap='gray')
348
349     print(f"Image saved to: {save_path}")
350
351
352 #
    ------------------------------------------------------------------------------------#
```

C.2.3 Skript 3: Bildklassifikation SKAN

```
1  # import and set seed:
2  #pip install medmnist tqdm -- needed
3  import torch
4  import torch.nn as nn
5  import torch.optim as optim
6  from torch.utils.data import DataLoader, TensorDataset, random_split
7  from torch.cuda.amp import autocast, GradScaler
8
9  import os
10 import matplotlib
11 import numpy as np
12 ##from scipy.interpolate import BSpline
13 #from scipy.interpolate import UnivariateSpline
14 import matplotlib.pyplot as plt
15 from numpy.random import rand
16 import pandas as pd
17 #from sklearn.model_selection import train_test_split
18 from functools import partial
19 import random
20 torch.cuda.empty_cache()
21
22 def set_seed(seed):
23     torch.manual_seed(seed)  # Set the seed for CPU operations
24     np.random.seed(seed)  # Set the seed for numpy
25     random.seed(seed)  # Set the seed for Python's random module
26
27     if torch.cuda.is_available():
28         torch.cuda.manual_seed(seed)  # Set seed for CUDA
29         torch.cuda.manual_seed_all(seed)  # Set seed for all GPUs
30         torch.backends.cudnn.deterministic = True  # Ensures deterministic behavior
31         torch.backends.cudnn.benchmark = False  # Disables optimization that can
               introduce randomness
32
33 # Example usage
34 set_seed(1919)
35
36 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
37 print(device)
38
39 # set up data:
40
41 from medmnist import PneumoniaMNIST, INFO
42 from torchvision import transforms
43 from torch.utils.data import DataLoader
44 import torch
45 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
46 info = INFO['pneumoniamnist']
47 info['res'] = 64
48 # define transforms
49 transform = transforms.Compose([
50     transforms.ToTensor(),
51     transforms.Lambda(lambda x: x.view(-1))
52 ])
53 batch_size = 64
54 info = INFO
55 print(info["pneumoniamnist"])
56
57 # load the datasets
58 train_dataset = PneumoniaMNIST(split='train', transform=transform, download=True,
      as_rgb=False)
59 test_dataset = PneumoniaMNIST(split='test', transform=transform, download=True,
      as_rgb=False)
60
61 image, label = train_dataset[0]  # Get the first sample
```

```
print(image.shape)

#-------------------------------------------------------------------------------
#defining models
class SprecherSpecial(nn.Module):
    def __init__(self,structure,batch,degree,num_knots):
        super().__init__()
        self.degree = degree
        self.batch = batch
        self.structure = structure
        self.knots = torch.arange(-2,2, step = 4/num_knots,dtype=torch.float32,
            device=device)
        self.num_c = self.knots.size(0)-degree-1
        self.std = 0.1
        self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
            structure)-1,self.num_c)))
        self.lambdas = nn.Parameter(torch.normal(mean=0, std=1, size=(sum(structure
            [:len(structure)-2]),)))
        self.eps = nn.Parameter(torch.normal(mean=0, std=1, size=(len(structure)
            -2,)))
        #for Xavier init
        sum_in_out_nodes =[structure[i]+structure[i+1] for i in range(len(structure
            )-1)]
        self.std_list = torch.tensor([self.xavier(x) for x in sum_in_out_nodes],
            dtype=torch.float32,device=device)
        xavier = torch.normal(0,self.std_list)
        ones = torch.ones(len(structure)-1,device = device)
        weights = torch.stack([ones,xavier],dim=1)
        self.weights = nn.Parameter(weights)
        self.layer = None
        self.dodo = nn.Parameter(torch.ones(len(structure)-2))
        self.epz = nn.Parameter(torch.abs(torch.normal(mean=0, std=0.1, size=(sum(
            structure[1:len(structure)-1]),),))) # epsilon values for each hidden
            node
        # for simpler indexing in layer pass function:
        o = structure.copy()
        del o[0]
        self.structure2 = o

    def xavier(self,x): #for Xavier init
        std = np.sqrt(2/(x))
        return std

    def layer_pass(self,x,layer):
        functions = torch.zeros((self.batch,self.structure[layer],self.structure[
            layer+1]),device=device) + x.unsqueeze(2).repeat(1,1, self.structure[
            layer+1])+(self.epz[sum(self.structure2[:layer]):sum(self.structure2[:
            layer+1])]).repeat(self.batch, self.structure[layer], 1)
        functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
            lambdas[sum(self.structure[:layer]):sum(self.structure[:layer+1])].view
            (self.structure[layer], 1)
        result = (torch.sum(functions,dim=1, keepdim=True)+ self.dodo[layer]*torch.
            tensor([[i for i in range(self.structure[layer+1])]],evice=device).
            float()).squeeze(1)
        self.layer = result
        return result

    def final_pass(self,x):
        result = torch.sum(self.phi(x,self.cs[len(self.structure)-2],self.weights[
            len(self.structure)-2]),dim=1)
        return result

    def forward(self,x):
        input = x
        for layer in range(len(self.structure)-2):
            input = self.layer_pass(input,layer)
        result = self.final_pass(input).view(self.batch,1)
        return result


    def basis_function(self, degree, knots, t): #non recursive faster basis
        function calculations
      """
      Compute all non-zero B-spline basis functions of a given degree for input
          tensor `t`.

      Returns: (t.shape, num_basis_functions) tensor of basis evaluations.
      """
      num_basis = len(knots) - degree - 1
      t = t.unsqueeze(-1)  # Shape: [*, 1] to [*, num_basis]

      # Initialize degree 0 basis functions (step functions)
      B = ((knots[:-1] <= t) & (t < knots[1:])).float()  # Shape: [..., num_basis +
          degree]
```

```
127
128         for d in range(1, degree + 1):
129             left = (t - knots[:-d-1]) / (knots[d:-1] - knots[:-d-1] )
130             right = (knots[d+1:] - t) / (knots[d+1:] - knots[1:-d] )
131
132             left_term = left * B[..., :-1]
133             right_term = right * B[..., 1:]
134             B = left_term + right_term
135
136         return B  # Shape: [..., num_basis]
137
138
139     def BSpline(self, c, knot_interval, degree, t):
140         """
141         Evaluate spline at t using basis function matrix and coefficients c.
142
143         Args:
144             c: (num_basis,) coefficient tensor
145             knot_interval: knot vector (tensor)
146             degree: degree of spline
147             t: input tensor, shape [...].
148
149         Returns:
150             Spline values at t, shape: [...]
151         """
152         B = self.basis_function(degree, knot_interval, t)  # Shape [..., num_basis]
153         return torch.sum(B * c, dim=-1)  # Weighted sum over basis
154
155
156     def phi(self, tensor, c, weight):
157         """
158         Final phi function, compatible with existing code.
159         """
160         spline_part = self.BSpline(c, self.knots, self.degree, tensor)
161         silu_part = nn.functional.silu(tensor)
162         return weight[0] * spline_part + weight[1] * silu_part
163
164
165     def plot_layer_function(self,layer,x):
166         y = self.phi(x,self.cs[layer],self.weights[layer])
167         x = x.tolist()
168         y = y.tolist()
169         plt.figure()
170         plt.plot(x,y)
171         plt.show()
172
173
174 #   ------------------------------------------------------------------------------------------------------------------
175
176 class Sprecher96Pro(SprecherSpecial):
177     def __init__(self, structure, batch, degree, num_knots):
178         super().__init__(structure, batch, degree, num_knots)
179         self.final_cs = nn.Parameter(torch.normal(mean=0,std=self.std,size = (
                structure[len(structure)-2],self.num_c)))
180         self.cs = nn.Parameter(torch.normal(mean=0, std=self.std, size=(len(
                structure)-2,self.num_c)))
181         ones = torch.ones(structure[len(structure)-2]-1,device = device)
182         std_list = self.std_list[-1].expand(ones.size(0))
183
184         xavier = torch.normal(0,std_list)
185         weights = torch.stack([ones,xavier],dim=1)
186         self.final_weights = nn.Parameter(weights)
187
188     def layer_pass(self,x,layer):
189         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1])) + x.unsqueeze(2).repeat(1,1, self.structure[layer+1])+(self.
                epz[sum(self.structure2[:layer]):sum(self.structure2[:layer+1])]).
                repeat(self.batch, self.structure[layer], 1)
190         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambdas[sum(self.structure[:layer]):sum(self.structure[:layer+1])].view
                (self.structure[layer], 1) # is this ok broadcasting
                ?-----------------------------------
191         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
192         self.layer = result
193         return result
194
195     def final_pass(self,x):
196         result = torch.zeros(size=(self.batch,self.structure[len(self.structure)
                -2]),device=device)
197         result[:,0] =  self.phi(x[:,0] ,self.final_cs[0],self.weights[-1])
198         for i in range(self.structure[len(self.structure)-2]-1):
199             res = self.phi(x[:,i] ,self.final_cs[i],self.final_weights[i])
```

```
200             result[:,i+1] = res.view(self.batch)
201         return torch.sum(result,dim=1)
202
203 #
    ---------------------------------------------------------------------------------------------------------
204 class SprecherLambda(SprecherSpecial):
205     def __init__(self, structure, batch, degree, num_knots):
206         super().__init__(structure, batch, degree, num_knots)
207         self.lambda_matrices = nn.ParameterList([nn.Parameter(torch.normal(mean=0,
                std=self.std, size=(structure[i],structure[i+1]))) for i in range(len(
                structure)-2)])
208
209     def layer_pass(self,x,layer):
210         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1]),device = device) + x.unsqueeze(2).repeat(1,1, self.structure[
                layer+1])+(self.epz[sum(self.structure2[:layer]):sum(self.structure2[:
                layer+1])]).repeat(self.batch, self.structure[layer], 1)
211         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambda_matrices[layer].unsqueeze(0)
212         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
213         self.layer = result
214         return result
215
216 class SprecherComb(Sprecher96Pro):
217     def __init__(self, structure, batch, degree, num_knots):
218         super().__init__(structure, batch, degree, num_knots)
219         self.lambda_matrices = nn.ParameterList([nn.Parameter(torch.normal(mean=0,
                std=self.std, size=(structure[i],structure[i+1]))) for i in range(len(
                structure)-2)])
220         self.knots = torch.arange(-5,5, step = 10/num_knots,dtype=torch.float32,
                device=device)
221
222
223     def layer_pass(self,x,layer):
224         functions = torch.zeros((self.batch,self.structure[layer],self.structure[
                layer+1]),device = device) + x.unsqueeze(2).repeat(1,1, self.structure[
                layer+1])+(self.epz[sum(self.structure2[:layer]):sum(self.structure2[:
                layer+1])]).repeat(self.batch, self.structure[layer], 1)
225         functions = self.phi(functions,self.cs[layer],self.weights[layer]) * self.
                lambda_matrices[layer].unsqueeze(0)
226         result = (torch.sum(functions,dim=1, keepdim=True)).squeeze(1)
227         self.layer = result
228         return result
229
230     def final_pass(self,x):
231         result = torch.zeros(size=(self.batch,self.structure[len(self.structure)
                -2]),device=device)
232         result[:,0] =  self.phi(x[:,0] ,self.final_cs[0],self.weights[-1])
233         for i in range(self.structure[len(self.structure)-2]-1):
234             res = self.phi(x[:,i] ,self.final_cs[i],self.final_weights[i])
235             result[:,i+1] = res.view(self.batch)
236         return torch.sum(result,dim=1)
237
238 #
    ---------------------------------------------------------------------------------------------------------
239 # setting up training and testing
240 from torch.optim.lr_scheduler import StepLR
241
242 torch.cuda.empty_cache() # clearing cashe for gpu
243 torch.cuda.ipc_collect()
244
245 batch_size = 250
246
247 set_seed(0)
248 train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=False,
        drop_last=False)
249 test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,
        drop_last=False)
250
251
252 model =  SprecherLambda([28*28,13,1],batch_size,3,20) #choose model, strucutre,
        batchsize and number of knots
253
254 model = model.to(device)
255
256 loss_function = nn.BCEWithLogitsLoss()
257 optimizer = optim.AdamW(model.parameters(),lr = 0.001, weight_decay=0.01)
258
259 num_epochs = 100
260 acc_list= []
261
262 def confusion(pred,target): # funciton for extracting confusion cases saving them
```

```
  TP = 0
  TN = 0
  FP = 0
  FN = 0
  for i in range(pred.size(0)):
    if pred[i] == 1 and target[i] ==1:
      TP +=1
    elif pred[i] == 0 and target[i] ==1:
      FN +=1
    elif  pred[i] == 1 and target[i] ==0:
      FP +=1
    else:
      TN +=1
  return TP,TN,FP,FN

cm_list = []
# train and test

for epoch in range(num_epochs):
    model.train()
    running_loss = 0
    train_batches = list(train_loader)
    for i, (data, target) in enumerate(train_batches):
        if i == len(train_batches) - 1:
          model.batch = data.size(0)
        data, target = data.to(device), target.to(device)
        target = target.float()
        optimizer.zero_grad()
        prediction = model(data)

        loss = loss_function(prediction,target)
        loss.backward()

        optimizer.step()

        running_loss += loss.item()
        model.batch = batch_size
    print('running training loss: ', running_loss)

    model.eval()
    correct = 0
    total = 0
    val_loss = 0
    total_TP, total_TN, total_FP, total_FN = 0, 0, 0, 0

    with torch.no_grad():
      test_batches = list(test_loader)
      for i, (data, target) in enumerate(test_batches):
          if i == len(test_batches) - 1:
            model.batch = data.size(0)

          data, target = data.to(device), target.to(device)
          target = target.float()
          prediction = torch.sigmoid(model(data))
          loss = loss_function(prediction,target)
          val_loss += loss
          prediction = (prediction > 0.5).float()
          correct += (prediction == target).sum().item()
          total += target.size(0)
          TP,TN,FP,FN = confusion(prediction,target)
          total_TP += TP
          total_TN += TN
          total_FP += FP
          total_FN += FN
          model.batch = batch_size

    cm_list.append((total_TP, total_TN, total_FP, total_FN))

    print(f'Test acc: {100*(correct / total):.2f}%')
    acc_list.append(correct/total)
    print('epoch ',epoch+1 , 'done :)')


print(acc_list.max())


# plot confusion matrix :
cm = np.array([[total_TP,total_FP],[total_FP,total_TN]])
plt.imshow(cm,cmap='Greens')
for i in range(2):
    for j in range(2):
        plt.text(j, i, str(cm[i, j]), ha='center', va='center', color='black')
plt.xticks([0, 1], ['1', '0'])
plt.yticks([0, 1], ['1', '0'])
```

```
347 plt.xlabel('Predikterad etikett')
348 plt.ylabel('Etikett')
349 plt.colorbar()
350 plt.show()
```