CHALMERS

GÖTEBORGS UNIVERSITET

En jämförande studie av regulariserade neurala nätverk med tillämpning på bildklassificering

A comparative study of regularized neural networks with application to image classification

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

Eric Johansson
Björn Krook Willén
Aladdin Persson
Marcus Sajland

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2020

En jämförande studie av regulariserade neurala nätverk med tillämpning på bildklassificering

Eric Johansson Björn Krook Willén
Aladdin Persson Marcus Sajland

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Aladdin Persson

Kandidatarbete i matematik inom civilingenjörsprogrammet Kemiteknik med fysik vid Chalmers
Björn Krook Willén Marcus Sajland

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers
Eric Johansson

Handledare: Larisa Beilina

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2020

Förord

Denna rapport är en kandidatuppsats skriven på Chalmers tekniska högskola och Göteborgs universitet våren 2020. Detta är en speciell vår i och med den coronapandemi som i detta nu sveper över jorden. Pandemin har framförallt påverkat miljontals människor vilka i detta nu testat positivt för sjukdomen världen över och alla de som dött eller råkat ut för en närståendes bortgång, men också har den påverkat alla studenter som på grund av stängda skolor och universitet fått jobba hemifrån. Så har fallet även varit för oss i denna grupp som står bakom detta arbete. Ungefär halva tiden, från mitten av mars till vårterminen slut har arbetet förlagts just hemifrån. Trots de svårigheter som uppstått i kölvattnet av coronaviruset har arbetet kunnat utföras på ett nöjaktigt maner, men såklart skulle vissa delar kunnat flyta på bättre, och dessutom varit roligare att utföra, om vi kunnat samarbeta på plats. Vi som författat artikeln heter Eric Johansson, Björn Krook Willén, Aladdin Persson och Marcus Sajland. De individuella prestationerna finns dokumenterade i gruppens dagbok samt individuella loggbok och sammanfattas här.

Eric har skrivit avsnitten om struktur och notation, framåtpropagering, aktiveringsfunktioner, kostnadsfunktioner, initiering av vikter, uppdelning av träningsdata, normalisering och batchnormalisering. Eric har också samförfattat inledningen tillsammans med Björn och samförfattat diskussion och slutsatser tillsammans med Aladdin.

Björn har skrivit avsnitten om syfte, avgränsningar och etiska aspekter. Han har också skrivit delavsnittet om balansprincipen samt resultat för data separerad i två klasser och flera klasser. Björn har också samförfattat inledningen tillsammans med Eric samt den populärvetenskapliga texten tillsammans med Marcus.

Marcus har skrivit avsnitten om bakåtpropagering nedstigningsmetoder. Marcus skrev också sammanfattning och abstract. Han har också samförfattat den populärvetenskapliga texten tillsammans med Björn, och har också varit delaktig i skrivandet av diskussion.

Aladdin har skrivit avsnitten om konvolutionella neurala nätverk, inlärningsöverföring, regularisering, metoder för sökning av hyperparametrar, randomiserad logspace, metod beskrivning, beskrivning av data, dataset med bilder, implementation, resultat för bilddata, och samförfattat diskussion och slutsatser med Eric. Aladdin har varit ansvarig för träningen av neurala nätverk och alla numeriska resultat för MNIST, MNIST-Fashion, CIFAR10 och ISIC-dataset. Även varit huvudansvarig för kodandet i MATLAB och allt kodande i Pytorch ramverket.

Stora delar av skrivandet har skett med kontinuerlig återkoppling på varandras delar. Sista tiden innan inlämning har mycket tid lagts på att korrekturläsa och revidera rapporten både språkligt och matematiskt. Eric har varit den som spenderat mest tid på detta och sett till att alla delar på arbetet förbättrats. Aladdin och Marcus har också varit delaktig i att gå igenom rapporten på flera delar tillsammans med Eric.

Även fast många delar har haft en huvudförfattare har också mycket gjorts tillsammans i gruppen. En stor del av kodandet i MATLAB har gjorts gemensamt i gruppen, speciellt för att koda och förstå neurala nätverk samt för det linjära- och ickelinjära-data.

Slutligen vill vi tacka vår handledare Larisa för all hjälp under arbetets gång, fackspråk för hjälp med det språkliga samt Andrew Ng för hans arbete med att underlätta en oinsatt att snabbt förstå grunderna i djup maskininlärningsfältet.

Populärvetenskaplig presentation

Du har nog märkt att det ofta är mycket lättare att bara memorera saker istället för att lära sig att förstå dem. Detta är inte bara något som uppstår i människans biologiska hjärna utan även i artificiella hjärnor, eller artificiella neurala nätverk, som är en typ av Artificiell Intelligens (AI). Detta kan så klart leda till dålig prestation och därför är det viktigt att motverka memorering hos artificiell intelligens, vilket kan göras genom en teknik som kallas för regularisering. Det är detta som vi undersökte genom vårt kandidatarbete.

Ett problem som är väldigt vanligt inom artificiell inteligens är att datorprogrammet memorerar vad det ska göra och hur det ska göra det istället för att 'tänka' själv. Ta exempelvis en chattbot. Denna skulle mycket väl kunna memorera hur den ska svara på olika frågor, men när det då kommer en fråga som den aldrig har sett så kommer den inte kunna komma med ett vettigt svar. Regularisering, tekniken som motverkar memorering, kan implementeras genom en rad olika metoder, och det var dessa metoder som vi undersökte. Studien gick ut på att applicera olika regulariseringsmetoder på olika typer av data för att undersöka hur stor påverkan dessa metoder kan ha på resultaten.

Men vad är ett artificiellt neuralt nätverk egentligen, och vad har det att göra med en hjärna? Jo, artificiella neurala nätverk är en typ av djup maskininlärning som är inspirerad av den biologiska hjärnan och går ut på att matematiskt modellera den. Modellen består av lager av neuroner, eller mer specifikt, aktiveringsfunktioner, som behandlar olika signaler. En biologisk hjärna är extremt komplex och den är fortfarande ett mysterium men en väldigt förenklad bild över hur hjärnan behandlar information är ungefär så här: en elektrisk signal skapas och sänds till en neuron i form av en impuls där denna signal behandlas och sen åker vidare till andra neuroner som igen behandlar signalen och skickar den vidare. Denna process sker kontinuerligt i hjärnan och tycks ge upphov till alla våra kognitiva funktioner.

För att skapa en modell av detta krävs att man gör förenklingar och det visar sig att även en mycket simpel modell kan ge goda resultat. De första aktivieringsfunktionerna tar helt enkelt alla insignaler, till exempel färgstyrkan hos pixlar i en bild, summerar dem och skickar dem vidare till nästa lager precis som i en hjärna. När all data, det kan vara färgstyrkan för alla pixlar i bilden, har nått det sista lagret ska nätverket kunna göra någon form av bedömning. Detta kan till exempel vara vad bilden visade, och om nätverket är uppbyggt på ett tillräckligt bra sätt så kommer det ofta göra rätt bedömning.

Maskininlärning är ett väldigt brett område och omfattar allt från ansiktsigenkänning på telefonen till att få en bil till att köra av sig själv. Därför är uppgiften att undersöka regularisering oerhört komplex och vi valde därför att endast fokusera på bilder. De artificiella neurala närverken som beaktats är alltså gjorda för att identifiera vad som visas på en bild och kunna klassificera detta. Till att börja med så laddade vi ner flera tiotusentals bilder på handskrivna siffror och byggde upp ett nätverk med mål att kunna mata in dessa bilder för att nätverket sedan skulle avgöra vilken siffra som stod på varje bild. Detta låter kanske enkelt men tänk då på hur många år det tar för barn att lära sig att läsa. Och här förväntas datorn att göra detta på bara några minuter. Dessa bilder delades upp i två kategorier, en träningskategori och en testkategori. Det neurala nätverket fick sedan öva på att klassificera träningsbilderna, och när det hade tränat på detta flera hundratals gånger så fick den ett försök på sig att avgöra vilka siffror som stod på testbilderna. Det är precis här som regularisering kan ha en stor påverkan. Om nätverket har memorerat träningsbilderna så kommer det kunna klassificera dem väldigt väl men när det då visas testbilder så kommer det ha mycket svårt för att klassificera dem. Med en teknik som kallas för L_2-regularisering så erhölls en noggrannhet på 98.3% vilket betyder att om nätverket visas tusen bilder av handskrivna siffror som det aldrig har sett så kommer det kunna säga med korrekthet vad det står på över 983 av dem. Utan regularisering skulle nätverket generellt sätt klassificera fem färre bilder korrekt

Samma princip användes på några andra dataset, till exempel på klassificering av olika klädesplagg och även för ett dataset som bestod av både olika typer av djur och fordon. För båda dessa dataset visade det sig igen att regularisering ger bättre resultat och för det andra av dem så blev klassifice-

ringsnoggranheten högre med över 8%. En intressant upptäckt är att för alla dataset så var det en regulariseringsmetod som kallas för dropout som gav bäst resultat. Denna metod är relativt enkel att förstå då den går ut på att man slumpmässigt stänger av kopplingar mellan neuroner och på det sättet skapar många mindre nätverk i det stora nätverket. Detta gör att resultaten inte kan bero för mycket på enstaka kopplingar mellan neuroner. Ett sista dataset undersöktes också, och denna data var mer än tjugotusen bilder på hudcancer-fläckar som antingen var godartade eller maligna. Exakt samma princip användes här igen och med hjälp av regularisering producerade nätverket resultat i klass med läkares. Det vill säga att det artificiella neurala nätverket kunde ofta med högre noggrannhet än läkare bedöma om hudcancern var godartad eller malign.

Det finns flera detaljer som måste anpassas för att få artificiella neurala nätverk att fungera optimalt och om detta skulle bli för svårt skulle det inte vara första gången som förhoppningen om AI dör ut. Men när datorprogram kan avgöra om en cancer är godartad eller malign bättre och snabbare än en läkare så känns det som att där är stor motivation för att fortsätta framåt.

Sammanfattning

Denna rapport fokuserar på jämförelsen mellan olika regulariseringstekniker av artificiella neurala nätverk applicerade på klassificering av bilddata. Regulariseringsmetoderna som använts är L_2-regularisering och dropout, och dessa har jämförts med icke-regulariserande neurala nätverk. Ett neuralt nätverk programmerades från grunden i MATLAB som initialt användes, men för effektivare träning av större nätverk användes Pytorch ramverket. Dataseten som undersöks är MNIST, MNIST-Fashion, CIFAR10 och ett hudcancer-dataset från ISIC. Två simulerade dataset i 2D användes också för att få en visuell idé om hur regularisering påverkar nätverket. Resultaten visar att regularisering ger bättre generalisering, men också att nätverksarkitekturen kan ha stor påverkan och en regulariserande effekt. Med tillämpning på hudcancer-data ser vi att dropout ger bäst generalisering i fallet av konvolutionella och feedforward neurala nätverk samt noterar att modellens prestation är nära toppmodern och erhåller resultat i linje med dermatologer och läkare i klassificiering av hudcancer.

Abstract

This report compares different types of regularization techniques for artificial neural networks when classifying image data. The regularization techniques used are L_2-regularization and dropout, and these are also compared with unregularized networks. An artificial neural network was built from scratch in MATLAB and was used initially, but to allow for faster training of a large network a PyTorch framework was used. The datasets that are used are MNIST, MNIST-Fashion, CIFAR10, and an ISIC skin cancer dataset. To gain a visual understanding of how regularization impacts the performance of the network two 2D datasets were also created in MATLAB. Results show that regularized networks generalize better than their nonregularized counterparts, but it is also evident that network architecture can have a regularizing effect. For the skin cancer dataset the dropout technique showed the best performance, and furthermore, the network's performance is in line with dermatologists and medical professionls when it comes to classifying whether the cancer is benign or malignant.

Innehåll

1 Inledning

År 1950 formulerade matematikern Alan Turing sitt berömda Turingtest med avsikt att ge ett slags mått på en dators intelligens [1]. Detta kan ses som ett fundament till det konceptuella system rörande datorers intelligens och medvetande. Ett halvt decennium senare samlade John McCharty dessa begrepp under samlingsnamnet artificiell intelligens och definierade detta som ”kunskapen och ingenjörskapet att skapa intelligenta maskiner” [2]. Sedan dess har vetenskapen om artificiell intelligens behövt ta sig igenom två stycken så kallade AI-vintrar, perioder av lågt intresse från industri och akademi [3]. Den första inträffade under 1970-talet efter det att en artikel beställd av det brittiska vetenskapsrådet påvisat mycket pessimistiska framtidsutsikter för området vilket ledde till kraftigt minskad finansiering [4]. Den andra inträffade i slutet av 1980-talet efter en allmänt minskad tilltro till artificiell intelligens, bland annat efter det att forskare inom fältet gett allt för storslagna löften om framgångar vilka inte infriades och att den tidens artificiellt intelligenta maskiner konkurrerades ut av billiga persondatorer [3]. Sedan början av 2000-talet blomstrar intresset för Artificiell Intelligens än en gång, detta till stor del på grund av framgångar inom fältet maskininlärning.

Maskininlärning kan definieras som det område vilket ger datorer möjligheten att lära sig utan att vara explicit programmerade [5]. Vanligtvis är datorer programmerade genom att steg för steg beskriva hur en uppgift ska utföras, maskininlärningsalgoritmer använder istället data för att göra förutsägelser vilket kan liknas med att datorerna ”programmerar sig själva”. Ett fält inom maskininlärning som under senaste tiden gett flera revolutionerande resultat är deep learning.

Inom deep learning utnyttjar man så kallade artificiella neurala nätverk (ANN) vilka är datorprogram löst inspirerade från de nätverk av neuroner som finns i biologiska hjärnor. Namnet ”deep learning” kommer från att ANN konceptuellt kan beskrivas som att de är byggda i lager där inlärningsprocessen sker på en högre abstraktions-nivå desto djupare ned i lagren data bearbetas [6]. Fördelen med ANN är ett de är extremt flexibla med avseende på att den data som skickas in i nätverken inte behöver någon avancerad förbehandling [7] och att de kan användas till att lösa ett nästan oändligt antal olika problem, men de kan också vara instabila och svåra att träna [8] [9].

Ett problem med träningen av neurala nätverk är att de kan lära sig ”för bra”, vilket innebär att nätverket lärt sig all data det utsätts för men kan inte generalisera denna kunskap till nya, tidigare osedda exempel [6]. För att motverka detta används en metod kallad regularisering med vilken man vill motverka denna överanpassning samtidigt som nätverket måste lära sig från sedd data för att kunna generalisera till ny, liknande data. I denna rapport ska vi se närmare på olika tekniker för regularisering och effekten av dessa på olika typer av data.

1.1 Syfte

Det översiktliga syftet med detta arbete är att undersöka olika regulariseringstekniker på neurala nätverk som implementerats för att klassificera bilddata. Ett delsyfte har varit att bygga ett artificiellt neuralt nätverk från grunden och att simulera data för att kunna skapa en visuelisering av regulariseringens effekt. Vidare ska studien undersöka om det finns en regulariseringsteknik som presterar bättre på en majoritet av dataseten, samt om nätverksarkitekturen påverkar vilken teknik som presterar bäst. Slutligen undersöks vilken regulariseringsteknik som ger bäst resultat för ett mer avancerat set av bilddata, närmare bestämt för klassificiering av hudcancer.

1.2 Avgränsningar

Det finns flera olika regulariseringstekniker, men denna rapport behandlar endast de två vanliga teknikerna L_2-regularisering och dropout. Vi begränsar oss ytterligare till att bara undersöka ett fully connected och ett konvolutionellt nätverk per dataset. Vidare undersöks endast neurala nätverk med en bestämd aktiveringsfunktion, kostnadsfunktion, minibatch storlek, initialisering av vikter och nedstigningsmetod.

1.3 Etiska aspekter

Artificiell intelligens, vilket inkluderar neurala nätverk, är ett ämne med enorm potential och stora risker om tekniken hamnar i fel händer [10]. Med detta kommer så klart många moraliska och etiska frågor som måste besvaras innan tekniken kan appliceras på verkliga problem. Detta arbete behandlar en specifik matematisk teknik inom neurala nätverk som ämnar åt att förbättra prestanda. Vi anser inte att arbetets resultat i sig innebär etiska dilemman, men är medvetna om att det kan vara så då det sätts i ett större sammanhang. Arbetet använder data hämtad från offentliga databaser vilka inte innehåller några personuppgifter eller annan information som kan anses vara känslig. För vidare applikationer krävs rådgivning för moral och etik.

2 Teori

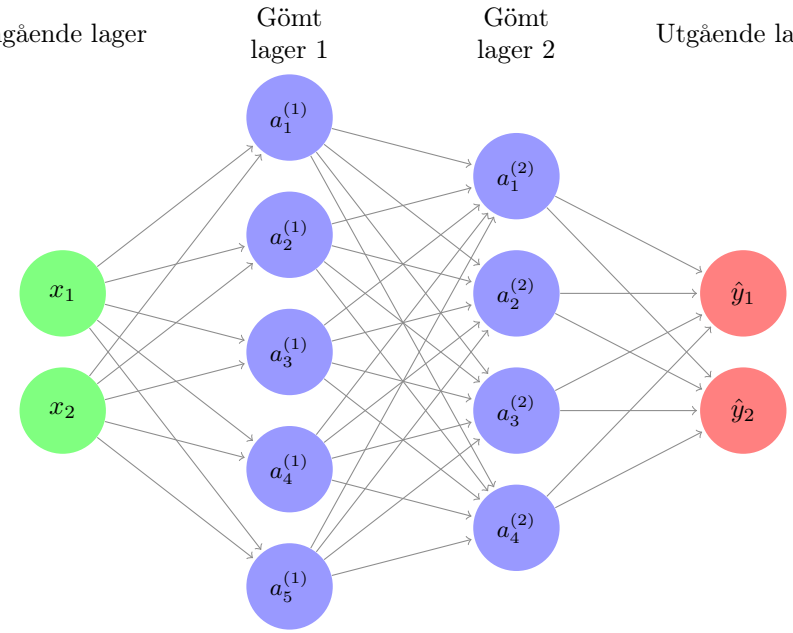
I detta avsnitt presenteras teori med avsikt att ge läsaren en grundläggande förståelse av de typer av nätverk som använts i rapporten. I sektion (2.1) presenteras neurala nätverk av typen fully connected [11], i sektion (2.2) presenteras konvolutionella neurala nätverk [12] och till sist i sektion (2.3) ges mer detaljerad förklaring till funktioner, metoder och initieringar hörande till de båda nätverkstyperna. Vidare beskrivs ett urval av optimeringsmetoder vilka används för att underlätta och effektivisera träningen.

2.1 Beskrivning av fully connected neurala nätverk

Ett neuralt nätverk är en algoritm som kan beskrivas genom att den dynamiskt uppdateras eller tränas för att lära sig att att utföra en uppgift, exempelvis att klassificera bilder, från given indata. För att nå tillfredsställande resultat utsätts det neurala nätverket för en mängd redan korrekt klassificerade träningsexempel för vilka utfallet jämförs med den korrekta klassificeringen. Med hjälp av snillrika metoder uppdateras det neurala nätverket stegvis med målsättning att extrahera en funktion som ger en generell lösning till den givna uppgiften [11].

2.1.1 Struktur och notation

I figur (1) presenteras en schematisk bild av ett simpelt fully connected neuralt nätverk.



Figur 1: Ett artificiellt neuralt nätverk är en samling noder vilka alla är kopplade med samtliga noder i föregående lager via vikter. I varje nod görs en beräkning som sedan skickas vidare till noderna i nästa lager.

Ett neuralt nätverk av typen fully connected [11] är uppbyggt av artificiella neuroner betecknade a_k^l där $l \in (0, \ldots, L)$ representerar vilket lager neuronen tillhör och k dess index i lagret. Varje neuron är sammankopplad med samtliga neuroner i det närmast påföljande lager via vikter. Vikterna som kopplar samman lager $l-1$ och l representeras av matrisen

$$\mathbf{W}^l = \{w_{ij}^l\}, \;\; i = 1, 2, \ldots, n_l, \;\; j = 1, 2, \ldots, n_{l-1}, \tag{1}$$

där i är indexet för den neuron vikten kopplar till i lager l, j är index för den neuron vikten kopplar till i lager $l-1$ och n_l är antalet noder i lager l. Dessa vikter uppdateras löpande i takt med att nätverket lär sig vilket beskrivs närmare i bakåtpropagering, i sektion (2.1.3). Ett specifikt träningsexempel, som skickas in i nätverket betecknas $\mathbf{x}^{(i)}$ där längden på $\mathbf{x}^{(i)}$ är densamma som antalet dimensioner i datan. Beteckningen i hänvisar till ett specifikt träningsexempel ur m möjliga och mer specifikt att $i \in \{1, 2, \ldots, m\}$. Med $\mathbf{x}^{(i)}$ följer exemplets klassificering eller målvärde $\mathbf{y}^{(i)}$ som exempelvis kan vara en vektor av samma längd som antalet klasser i datan med 0 i alla index förutom det index som markerar korrekt klassificering vilket markeras med 1. $\mathbf{y}^{(i)}$ kan också vara en skalär där värdet representerar klassificeringen.

Antag ett neuralt nätverk med L lager och samma strukturella uppbyggnad som det i figur (1). Det första lagret, med $l = 0$, är då ekvivalent med träningsexemplet $\mathbf{x}^{(i)}$ som skickas in i nätverket och kallas för ingående lager. Lagren med index $l = 1, \ldots, L-1$ kallas för gömda lager och består utav ett godtyckligt antal neuroner anpassat efter hur komplex den eftertraktade funktionsytan är. Det sista lagret, med index $l = L$ kallas för utgående lager och består av det antal neuroner som krävs för att lösa den aktuella uppgiften. Exempelvis om uppgiften är att klassificera data bland $n > 2$ klasser krävs oftast n neuroner i utgående lagret. Om datan består av två klasser kan det räcka med att det utgående lagret består av endast en neuron vilken klassificerar datan till klass 0 eller 1 beroende på om ett tröskelvärde överstigs eller inte.

Neuronernas värde för lagren $l = 1, \ldots, L$ bestäms av en aktiveringsfunktion f, vilken beskrivs närmare i sektion (2.3.1), som tar in neuroner i föregående lagers värde multiplicerade med vikterna kopplade till aktuell neuron plus ett bias. Datan flödar genom nätverket från ingående data och modifieras via neuroner i gömda lager för att till slut resultera i nätverkets utfall givet av aktiveringsfunktionerna i det utgående lagret. Denna beräkningsgång förklaras närmare i följande kapitel om framåtpropagering [11] som är den metod med vilken det neurala nätverket genererar utdata givet en viss indata.

2.1.2 Framåtpropagering

Då vi multiplicerar en viktmatris $\mathbf{W}^l$ med vektorn innehållande värden hörande till föregående lagers noder $\mathbf{a}^{l-1}$ (eller $\mathbf{x}^{(i)}$ om $l = 1$) erhålles

$$\mathbf{z}^l = W^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{2}$$

där $\mathbf{z}_k^l$ är värdet från lager $l-1$ in i neuron med index k i lager l [11]. $\mathbf{b}^l$ är en vektor med biaser hörande till varje neuron i lager l. Värdet i neuronen a_k^l erhålles sedan av

$$a_k^l = f(z_k^l) \tag{3}$$

där f är aktiveringsfunktionen. För att klargöra demonstrerar följande ekvation framåtpropagering för det artificiella nätverket i figur (1) givet ett träningsexempel $\mathbf{x}^{(i)}$:

$$\hat{\mathbf{y}}^{(i)} = f(W^3 \underbrace{f(W^2 \underbrace{f(W^1 \mathbf{x}^{(i)} + \mathbf{b}^1)}_{\mathbf{a}^{(1)}} + \mathbf{b}^2)}_{\mathbf{a}^{(2)}} + \mathbf{b}^3). \tag{4}$$

Eftersom att vikter och biaser ej uppdateras under framåtpropagering kan vi se framåtpropagering som en deterministisk algoritm där samma grundinställningar på parametrar tillsammans med en bestämd indata alltid kommer resultera i samma utdata. Utdatan definieras av värdet på neuronerna i det sista lagret och kan till exempel tolkas som att given indata tillhör den klass som

representeras av den neuron med högst aktivering.

Framåtpropageringen avslutas med att jämföra det av nätverket beräknade resultatet, $\hat{\mathbf{y}}^{(i)}$, med det sökta resultatet $\mathbf{y}^{(i)}$. Detta görs genom att mata in de båda resultaten i en så kallad kostnadsfuntion:

$$C(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) \tag{5}$$

I praktiken beräknar kostnadsfunktionen hur långt det beräknade värdet är ifrån det sökta värdet och ger möjlighet att analytiskt undersöka hur vikterna och biaser ska ändras för att nätverket ska ge ett bättre resultat. I träningen av det neurala nätverket vill man därför minimera kostnadsfunktionen vilket beskrivs närmare i avsnittet om bakåtpropagering (2.1.3). En mer detaljerad beskrivning av kostnadsfunktioner ges i avsnitt (2.3.2).

Vi avslutar kapitlet om framåtpropagering med en beskrivning av algoritmen i pseudokod:

Algorithm 1: Framåtpropagering($\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$)

Initiera vikter $\mathbf{W}^l$ och biaser $\mathbf{b}^l$ för lager $l = 1, 2, \ldots, L$
$\mathbf{a}^0 = \mathbf{x}^{(i)}$
for *l=1:L* **do**
  $\mathbf{z}^l = W^l \mathbf{a}^{l-1} + \mathbf{b}^l$
  $\mathbf{a}^l = f_l(\mathbf{z}^l)$
end
$\hat{\mathbf{y}}$=$\mathbf{a}^L$
cost = $\mathbf{C}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$

2.1.3 Bakåtpropagering

Efter att framåtpropagering har utförts för att göra en initial klassificering av data så används något som kallas för bakåtpropagering för att uppdatera nätverkets vikter och biaser. Kärnan i bakåtpropagering ligger i att minimera den kostnadsfunktion som definierats för nätverket och som beräknats i slutet av framåtpropageringen. För att möjliggöra detta beräknas kostnadsfunktionens gradienter med avseende på nätverkets vikter och biaser. Dessa gradienter ger information om hur vikter och biaser ska uppdateras för att minska kostnadsfunktionens värde. På detta vis kan framåt- och bakåtpropagering urföras i en iterativ process och erhålla allt bättre resultat.

Det slutliga målet är därför att beräkna $\frac{dC}{dw}$ och $\frac{dC}{db}$ för alla neuroner i alla lager och genom dessa uppdatera vikter och biaser [13]. För att kunna beräkna de önskade derivatorna så måste först derivatan av kostnadsfunktionen med avseende på $\mathbf{z}$ beräknas. Denna kallas för δ och definieras som

$$\delta_j^l = \frac{dC}{dz_j^l}, \tag{6}$$

och fungerar som en indikator på hur mycket en förändring i $\mathbf{z}$ i den j:e neuronen påverkar den slutliga kostnadsfunktionen. För neuronerna i det sista lagret i nätverket så kan (6) skrivas som

$$\delta_j^L = \frac{dC}{da_j^L} f^{'}(z_j^L), \tag{7}$$

där uttrycket har expanderats med hjälp av kedjeregeln och $f^{'}(z_j^L)$ är derivatan av aktiveringsfunktionen vid z_j^L. För att förenkla beräkningarna så görs allt på vektorform, och (7) kan då skrivas som följande:

$$\delta^L = \nabla_{\mathbf{a}^L} C \odot f^{'}(\mathbf{z}^L). \tag{8}$$

Här är $\odot$ den så kallade Hadamard produkten, som utför elementvis multiplikation, och $\nabla_{a^L} C$ är gradienten av kostandsfunktionen med avseende på a^L. Man utnyttjar sedan att det finns en

koppling mellan δ-värdena i det sista lagret och δ-värdena i det tidigare lagret. Denna typ koppling fortsätter sedan genom hela nätverket då man stegar genom det baklänges. Dessa kopplingar beskrivs på följande sätt

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot f^{'}(z^l), \tag{9}$$

där multiplikation av δ-värdet för det *l+1*:e lagret med transponatet av dess viktmatris flyttar tillbaka felet till lagret före och ger δ^l [13]. Då denna relation är beräknad så kan den stegvis appliceras på alla lager i det neurala nätverket.

Gradienterna kan nu användas för att uppdatera vikter och biaser enligt följande samband

$$\frac{dC}{db^l_j} = \delta^l_j, \tag{10}$$

$$\frac{dC}{dw^l_{jk}} = a^{l-1}_k \delta^l_j \tag{11}$$

vilket fås genom att kombinera ekvation (7) och (9). Med dessa relationer klargjorda så kan uppdatering av vikter och biaser utföras. Detta görs med någon nedstigningsmetod, som förklaras mer i (2.3.3), där vikterna och biaser förändras åt motsatt riktning än kostnadsfunktionens gradient. I den enklaste nedstigningsmetoden, kallad gradientnedstigning, uppdateras vikter och biaser enligt

$$W^{'} = W - \alpha \frac{dC}{dW}, \tag{12}$$

$$\mathbf{b}^{'} = \mathbf{b} - \alpha \frac{dC}{d\mathbf{b}} \tag{13}$$

där $W^{'}$ och $\mathbf{b}^{'}$ representerar de uppdaterade värdena. α kallas för nätverkets inlärningstakt och är ett mått på hur fort vikterna och biaserna uppdateras. Detta värdet gissas oftast först och optimeras sedan beroende på hur nätverket presterar med ett givet dataset.

Förloppet att utföra framåt och bakåtpropagering över alla träningsexempel kallas för en epok. Dock behöver inte framåtpropageringen göras över samtliga träningsexempel innan vikterna uppdateras utan man kan dela in exemplen i så kallade minibatches, vilket beskrivs närmare i (2.3.4), för att få en högre frekvens på inlärningen.

Vi avslutar kapitlet om bakåtpropagering med en beskrivning av algoritmen i pseudokod:

Algorithm 2: Bakåtpropagering

```
// Beräkna gradienter
for l = L:1 do
    Beräkna δ^l
    dC/db^l_j = δ^l_j
    dC/dw^l_jk = a^{l-1}_k δ^l_j
end
// Uppdatera vikter och biaser
for l = 1:L do
    for j = 1 : n_l do
        for k = 1 : n_{l-1} do
            w^l_jk = w^l_jk − α dC/dw^l_jk
        end
        b^l_j = b^l_j − α dC/db^l_j
    end
end
```

2.2 Översiktlig beskrivning av konvolutionella neurala nätverk

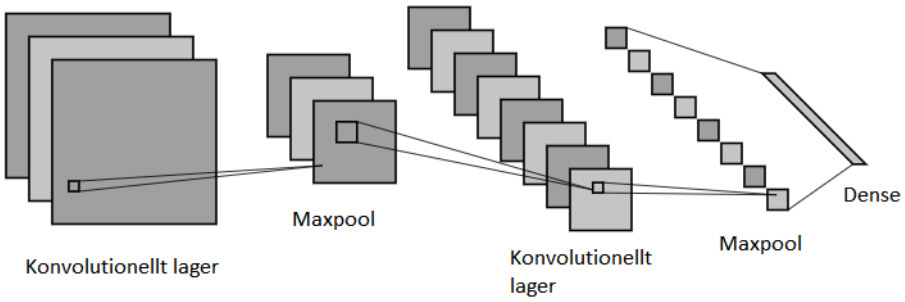
Konvolutionella neurala nätverk (CNNs) är en klass av artificiella neurala nätverk som är specialiserad för att fungera på data vars struktur är lokalt varierande och av den anledningen används CNNs i hög grad till bilder [14]. Från experiment under 1960-talet fick man djupare förståelse för hur synintryck fungerar genom att lysa olika ljus på katters näthinna. Resultaten visade att neuronaktivering i hjärnan var lokalt kopplad till ögats receptorfält [15]. CNNs bygger på dessa idéer genom att ha lokala receptorfält och återanvändning av vikter. Genom att ha dessa lokala receptorfält kan CNNs extrahera visuella karaktärsdrag som hörn, skuggor etc. även om de uppkommer på flera olika platser i bilderna [14].

Som lokalt receptorfält används en tensor med vikter, kallad filter, som matchar dimensionerna på ingående data. Exempelvis för en tvådimensionell bild kommer filtret också vara tvådimensionellt däremot är filterstorleken oftast betydligt mindre är bildens antal pixlar. Intuitivt låter vi ett filter svepa över bilden med specifik steglängd och beräkna konvolution med den del av bilden som täcks av filtren. Oftast används flera filter med olika vikter, för att kunna detektera olika karaktärsdrag i bilden och resultatet av konvolutionerna med dessa filter kallas feature maps [11].

En konvolution (*) för bilden I med ett visst filter K för en specifik (i,j) pixel beräknas som

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{14}$$

där det antas att funktionerna I och K är 0 utanför det område där pixelvärdet respektive vikterna befinner sig [11]. En annan typ av operation som ofta används i konvolutionella nätverk är ett pooling-lager som består av ett filter utan vikter som tar maximum av elementen i filtrets räckvidd. Om man exempelvis använder en maxpool av storlek 2x2 kommer detta pooling-lager reducera varje påträffad 2x2 i feature map till ett värde som är maxvärdet av 2x2-matrisen. En strukturell tolkning av ett CNNs arkitektur ges i figur (2).



Figur 2: Figur som beskriver uppdelning av CNN lager med maxpooling lager samt där termen dense används för ett fully connected nätverk.

Ett lager av ett konvolutionellt nätverk går igenom tre steg varav det första steget är att beräkna konvolutioner med alla filter och skapa ett antal feature maps [11]. Därefter används en ickelinjär aktiveringsfunktion elementvis på feature maps, exempelvis ReLU, och slutligen i tredje steget används många gånger ett pooling-lager som ofta är maxpool. Antalet av dessa lager som används i ett CNN beskrivs som djupet på nätverket. Eftersom konvolutionerna och pooling-lagren går med ett visst steg, ofta kallat ”stride”, kan storleken på feature maps beräknas med följande formel

$$n_{ut} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1. \tag{15}$$

Här antas att input är en kvadratisk matris där n_{in} är antalet dimensioner på matrisen och n_{ut} är dimensionerna efter en konvolution eller maxpool lager med stride s och dimensionerna på filtret är k. Det går också att utnyttja paddning vilket innebär att man utökar bilden med en tom ram av storlek p. Paddning kan användas för att anpassa storleken på bilden ut från ett lager vilket kan ses i ekvation (15) [12]. För fördjupad information om konvolutioner hänvisas till [12] och [11].

2.3 Träning av neurala nätverk

Följande avsnitt ger en mer detaljerad beskrivning av några utav de metoder och funktioner som ingår i de neurala nätverkens struktur. Dessutom beskrivs vanliga optimeringsmetoder för att få neurala nätverk att träna effektivare.

2.3.1 Aktiveringsfunktioner

Det finns flera olika typer av aktiveringsfunktioner som tillämpas på neurala nätverk. Dessa har olika egenskaper gällande exempelvis värdemängd och kontinuitet. En vanligt förekommande aktiveringsfunktion är den så kallade sigmoidfunktionen:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

Sigmoidfunktionen har värdemängd i intervallet $(0, 1)$, är överallt kontinuerligt deriverbar med derivata

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)), \tag{16}$$

och

$$\lim_{x \to \pm\infty} \sigma'(x) = 0. \tag{17}$$

Användandet av Sigmoidfunktionen var speciellt utbrett fram tills dess att ”rectifier linear unit function”- (ReLU) [16] studien publicerades 2010 och som sedan dess varit den främst använda aktiveringsfunktionen [17]. ReLU-funktionen definieras som den positiva delen av sitt argument enligt

$$\text{ReLU(x)} = max(0, x) \tag{18}$$

och har gradienten

$$\frac{d}{dx}\text{ReLU(x)} = \begin{cases} 0 & \text{om} \quad x \leq 0 \\ 1 & \text{om} \quad x > 0 \end{cases} \tag{19}$$

ReLU löser problem gällande för små och för stora gradienter (se kapitel 2.1.3) som sigmoidfunktionen kan medföra när den tillämpas på neurala nätverk med stort antal gömda lager. Dessutom har ReLU i allmänhet visat sig fungera bättre än sigmoid på vitt skilda typer av problem man vill lösa med hjälp av neurala nätverk [17].

Det neurala nätverk som implementerats i denna studie använder ReLU som aktiveringsfunktion i samtliga gömda lagers neuroner, men i utgående lager används en generalisering av sigmoidfunktionen kallad softmax [18]:

$$\text{Softmax}_i(\mathbf{x}) = \frac{\mathrm{e}^{x_i}}{\sum_j \mathrm{e}^{x_j}} \tag{20}$$

med gradient

$$\frac{d}{dx_j}\text{Softmax}_i(\mathbf{x}) = \text{Softmax}_i(\mathbf{x})(I_{ij} - \text{Softmax}_j(\mathbf{x})) \tag{21}$$

där i betecknar den i:te output-neuronen och I är enhetsmatrisen med dimension $n \times n$ där n är antalet noder i utgående lagret.

Två egenskaper hos softmax är

$$\text{Softmax}_i(\mathbf{x}) \quad \in [0, 1] \ \ \forall \ \ i \tag{22}$$

och

$$\sum_{i}^{n} \text{Softmax}_i(\mathbf{x}) = 1 \tag{23}$$

där n är antalet noder i utgående lagret. Detta gör att vi kan tolka Softmax($\mathbf{x}$) som en sannolikhetsfördelning med avseende på värdena $\mathbf{z}^L$ in i utgående lagret där det aktiverade värdet $\text{Softmax}_i(\mathbf{x})$ tolkas som sannolikheten att det givna exemplet tillhör klass i. I i Appendix (A) visas graferna för sigmoid- och ReLU-funktionerna (se figurerna (6) och (7)).

2.3.2 Kostnadsfunktioner

En kostnadsfunktion

$$C(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x})) \tag{24}$$

är ett mått på skillnaden mellan målvärdet $\mathbf{y}(\mathbf{x})$ och det av nätverket beräknade resultatet $\hat{\mathbf{y}}(\mathbf{x})$ [19]. Valet av kostnadsfunktion beror på uppgiften. Exempelvis för medicinsk diagnos är en kostnadsfunktion som ger ett högre värde om en sjuk patient klassificeras som frisk än vice versa att föredra. Som vi beskriver i avsnittet om bakåtpropagering (2.1.3) bygger nätverkets lärande på att bestämma hur vikter och biaser ska förändras för att minska felet i $\hat{\mathbf{y}}(\mathbf{x})$. Kostnadsfunktionen bör därför vara överallt differentierbar med avseende på nätverkets vikter W och biaser B för att möjliggöra en beräkning av gradienterna $\frac{dC}{dW}$ och $\frac{dC}{dB}$ över hela kostnadsfunktionens yta. Genom beräkningen av gradienterna erhålles information om hur W och B ska förändras för att få kostnadsfunktionen att konvergera mot ett minimum.

En vanlig kostnadsfunktion är medelvärdet av felet i kvadrat ”mean squared error” (MSE) [19]:

$$MSE(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x})) = \frac{1}{2} \sum_{i=1}^{K} (\hat{y}_i(\mathbf{x}) - y_i(\mathbf{x}))^2 \tag{25}$$

där det är lätt att se att $MSE \to 0$ då $\hat{\mathbf{y}} - \mathbf{y} \to \mathbf{0}$. (25) beskriver fallet för ett klassifieringsproblem med K klasser och endast ett träningsexempel.

En kostnadsfunktion som visat sig ge bättre resultat både för generalisering och tidsåtgång än MSE då den appliceras på klassificeringsproblem är cross entropy-funktionen [19][11]. Cross entropy-funktionen finns i flera varianter. Vilken man ska använda beror på uppgiften. Vid situationer då uppgiften är att klassificera data i en av $K > 2$ klasser, exempelvid vid ett tärningskast, används fördelaktigen den så kallade ”Categorial Cross Entropy loss”- eller ”Softmax loss”-funktionen. Den definieras enligt följande:

$$C(\mathbf{y}^{(i)}(\mathbf{x}), \mathbf{y}^{(i)}(\mathbf{x})) = -\sum_{j=1}^{K} (y_j \ln(\hat{y}_j)). \tag{26}$$

där $\hat{\mathbf{y}}$ beräknas av ett nätverk med Softmax som aktiveringsfunktion i utgående lagret. Notera att $\mathbf{y}$ är en vektor som består av uteslutande 0:or förutom en 1:a i det index som representerar $\mathbf{x}$ klasstillhörighet. I fallet då nätverket ger ett utfall nära 0 för den korrekta klasstillhörigheten kan vi se att kostnaden blir mycket stor ty $ln(x) \to -\infty$ då $x \to 0^+$. I fallet då nätverket ger ett utfall nära 1 för den korrekta klasstilhörigheten ser vi istället att kostnaden blir mycket liten eftersom $ln(x) \to 0$ då $x \to 1$. Därav ”vill” kostnadsfunktionen att nätverket ska lägga all sin vikt på korrekt klassificering för att den ska minimeras.

2.3.3 Nedstigningsmetoder

När vikter och biaser i nätverket skall uppdateras så görs detta med avseende på kostnadsfunktionen, och detta görs lättast med någon form av nedstigning metod. Den enklaste av dessa metoder är stokastisk gradientnedstigning [20], vilket innebär att värdena på vikter och biaser rör sig i riktningen motsatt till kostnadsfunktionens gradient, det vill säga åt den riktning där kostnadsfunktionen minskar som snabbast. För fallet med en vikt och en bias kan detta visualiseras som att vikten representeras som x-axlen och biasen representeras som y-axeln. Då är kostnadsfunktionen

en tre dimensionell yta med ett globalt minumum någonstans på x-y planet. Så klart så är riktiga nätverk mycket mer komplexa, men de baseras på samma idé. Med gradientnedstignings-metoden kan uppdateringen av vikter och biaser skrivas i generella termer som

$$w^{'} = w - \alpha \frac{dC}{dw}, \tag{27}$$

$$b^{'} = b - \alpha \frac{dC}{db} \tag{28}$$

där α är en parameter som kallas för inlärningstakt som avgör hur mycket värdena kan förändras med varje uppdatering. Här indikerar $'$ att vikten eller biasen har uppdaterats.

I vissa fall är gradientnedstigning tillräckligt effektiv då varje uppdatering av vikter och biaser tar kostnaden närmre ett minimum utan att variera för mycket åt andra riktningar, men många gånger är detta inte fallet. Till exempel, om ett lokalt minimum ligger i riktning med den positiva x-axeln så kan det hända att kostnaden rör sig åt det hållet men oscillerar i y-riktningen. Då det är tydligt att trenden är att kostnaden rör sig åt den positiva x-riktningen kan den så kallade "gradientnedstigning med momentum"-metoden införas. För att applicera denna metod uppdateras vikter och biaser genom att ta hänsyn till ett exponentiellt viktat medelvärde av tidigare gradienter. Detta innebär att oscillationer kring den optimala gradientnedstigning riktningen försvagas, samtidigt som att påverkan av en tidigare beräknad gradient blir mindre och mindre för varje steg som tas. I denna metod så multipliceras α inte med gradienterna, utan istället med de exponentiellt viktatde medelvärdena V_{dw} eller V_{db}, som kan definieras som

$$V_{dw}^{'} = \beta_1 V_{dw} + (1 - \beta_1)\frac{dC}{dw}, \tag{29}$$

$$V_{db}^{'} = \beta_1 V_{db} + (1 - \beta_1)\frac{dC}{db}. \tag{30}$$

Här är $\beta \in [0, 1]$ en vikt som avgör hur starkt tidigare värden på V_{dw} eller V_{db} påverkar nästa värde. dw och db är fortfarande gradienter och beräknas enligt gradientnedstignings-metoden. För varje batch (se 2.3.4) av data räknas gradienter ut. Dessa gör det möjligt att räkna ut V_{dw} och V_{db}, och slutligen kan vikter och biaser uppdateras genom

$$w_k^{'} = w_k - \alpha V_{dw}, \tag{31}$$

$$b_k^{'} = b_k - \alpha V_{db}. \tag{32}$$

Anledningen till att V_{dw} och V_{db} kallas exponentiellt viktatade medlevärden är för att en ny beräkning av $V_{dw,n}$ beror på $V_{dw,n-1}$, $V_{dw,n-2}$, och så vidare. Motsvarande gäller för biaserna.

En tredje nedstigningsmetod är RMSprop. Skillnaden mellan RMSprop och gradientnedstigning med momentum är att istället för V_{dw} och V_{db} så används S_{dw} och S_{db}. Dessa termer är definierade enligt

$$S_{dw}^{'} = \beta_2 S_{dw} + (1 - \beta_2)(\frac{dC}{dw})^2, \tag{33}$$

$$S_{db}^{'} = \beta_2 S_{db} + (1 - \beta_2)(\frac{dC}{db})^2, \tag{34}$$

med β_2 definierat på liknande sätt som β_1. Vikter och biaser uppdateras enligt

$$w_k^{'} = w_k - \alpha \frac{\frac{dC}{dw}}{\sqrt{S_{dw}} + \epsilon}, \tag{35}$$

$$b_k^{'} = b_k - \alpha \frac{\frac{dC}{db}}{\sqrt{S_{db}} + \epsilon}. \tag{36}$$

där ϵ är en konstant som är med är för att undvika att inlärningstermen går mot oändligheten för små S_{db}. På grund av att α divideras uttrycket i nämnaren kan inlärningen anpassas så att

den tar ut oscillationer [21]. Till exempel, om där skulle vara stora oscillationer i b-riktningen så skulle S_{db} vara stor för de första stegen, vilket skulle göra så att förändringen i $b_k^{'}$ inte skjuter över. .

Gradientnedstigning med momentum och RMSprop kan kombineras för att skapa Adam, eller ”Adaptive Moment Estimation”, som ofta ger bättre resultat [22]. När denna metod appliceras används oftast justerade värden för V_{dw}, V_{db}, S_{dw} och S_{db} som skrivs enligt

$$V_{dw}^{'} = \frac{V_{dw}}{1-\beta_1^t}, \tag{37}$$

$$V_{db}^{'} = \frac{V_{db}}{1-\beta_1^t}, \tag{38}$$

$$S_{dw}^{'} = \frac{S_{dw}}{1-\beta_2^t}, \tag{39}$$

$$S_{db}^{'} = \frac{S_{db}}{1-\beta_2^t}. \tag{40}$$

Detta kallas för ”bias justering” och är ett sätt att undvika att de första värdena ligger för nära noll. Adam-metoden innebär då att vikter och biaser uppdateras med följande ekvationer:

$$w_k^{'} = w_k - \alpha \frac{V_{dw}^{'}}{\sqrt{S_{dw}^{'}} + \epsilon}, \tag{41}$$

$$b_k^{'} = b_k - \alpha \frac{V_{db}^{'}}{\sqrt{S_{db}^{'}} + \epsilon}. \tag{42}$$

2.3.4 Uppdelning av träningsdata i minibatches

En förutsättning för en lyckad träning av ett neuralt nätverk är att utsätta nätverket för en mycket stor mängd exempel. En tumregel är att ju fler träningsexempel desto bättre, men det kan medföra en tidskrävande träningsprocess. Ett sätt att snabba upp träningsprocessen är att dela in mängden träningsexempel i ett antal submängder, så kallade minibatches [23]. Nätverket får då under en epok utföra träningsprocessen på varje sådan minibatch. Med en epok menas här att nätverket har sett hela mängden av träningsexempel. Det gör att uppdateringen av vikter och biaser, och således stegningen mot ett minimum i kostnadsfunktionen, kommer ske mer frekvent än om alla träningsexempel använts för varje uppdatering. För att undvika att nätverket ser samma minibatch under flera epoker och för att säkertsälla att en minibatch innehåller varierande klasser kan man randomisera ordningen på exemplen innan man delar upp i minibatches [24].

2.3.5 Normalisering av ingående data

Genom att normalisera sin data kan man uppnå ett antal fördelar, däribland att omotiverade skillnader i parametrars inverkan på framåtpropageringen minskar och i bästa fall uppnås en snabbare träningsprocess. Det finns flera metoder för normalisering av data, gemensamt är att de ofta bygger på att ändra parametrarnas intervall och medelvärde [25]. Ett exempel är Z-score normalisering som beräknas enligt:

$$\hat{x}_i = \frac{x_i - \mu_i}{\sigma_i} \tag{43}$$

där x_i är vektorn innehållande den i:te parametern för samtliga exempel i aktuellt dataset, μ_i är vektorns medelvärde och σ_i är dess standardavvikelse.

2.4 Regulariseringsmetoder för neurala nätverk

När modeller tränas är ett mål att kunna klassificera och skapa en bra representation av träningsdata. Det översiktliga målet är sedan att modellen ska kunna använda den representationen och generalisera till okänd data modellen tidigare inte sett, så kallad testdata [11]. Vi definierar regularisering som den klass av metoder som kan förbättra modellens förmåga att generalisera till okänd data utan hänsyn till exaktheten på träningsdatan. Det går att uppnå mycket hög träningsexaktheter med tillräckligt avancerade modeller men som ändå inte ger tillfredsställande resultat på testdata. När en modell presterar bra på träningsdata men inte lyckas generalisera benämns detta överanpassning. Detta kan tolkas som att modellen enbart memorerat alla exempel snarare än skapat en representation som känner igen de generella strukturerna [6].

I detta delkapitel kommer vi beskriva några av de mest populära regulariseringsteknikerna vilka visat sig vara mest effektiva i de flesta applikationer [11]. Eftersom det inte finns teoretiska bevis för om och när dessa tekniker fungerar bättre i det ickelinjära samband man använder neurala nätverk är det i praktiken något som man undersöker empiriskt, och ofta specifikt till det dataset man arbetar med. Detta är också anledningen till varför det finns en mängd olika tekniker och vissa kan prestera bättre i olika användningsområden, däremot noterar vi att de mest förekommande är dropout och L_2-regularisering som också är vad denna rapport använder sig av i de numeriska studierna.

2.4.1 L_2 regularisering

En av de vanligaste och enklaste regulariseringsteknikerna är L_2-regularisering också känt som weight decay, ridge regression eller Tikhonov regularisering [11]. Vi definierar kostnadsfunktionen $C(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x}))$ som exempelvis kan vara cross-entropy eller mean squared error vilka beskrivits i mer detalj i (2.3.2). Med L_2-regularisering modifierar vi kostnadsfunktionen på följande sätt

$$C'(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x}), \mathbf{W}) = C(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x})) + \frac{\lambda}{2} \sum_{l=1}^{L} \sum_{i=1}^{n_{l-1}} \sum_{j=1}^{n_l} (w_{ij}^l)^2 \tag{44}$$

där λ är regulariseringsparametern och n_l är antalet noder i lager l. Derivatan för en specifik vikt beräknas då som

$$\frac{\partial}{\partial w_{ij}^l} C'(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x}), \mathbf{W}) = \frac{\partial}{\partial w_{ij}^l} C(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x})) + \lambda(w_{ij}^l) \tag{45}$$

där $\hat{y}(x)$ beror på vikterna $\mathbf{W}$, mer om detta kan läsas i avsnittet om framåtpropagering i sektion (2.1). Ett sätt att se (44) är att modellen kommer vilja minimera $C(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x}))$ samt minimera summan av vikterna i kvadrat. Intuitivt kan vi notera att ett nätverk kommer vilja skapa en representation med vikter av lägre magnitud vilket resulterar i en lägre kostnad. En följd av detta är att modellerna oftast kommer bli enklare och därmed undviker att modellen överanpassar till specifik träningsdata [6]. För fördjupad information om L_2-regularisering hänvisar vi till [26].

2.4.2 Dropout

En populär regulariseringsteknik som är mycket beräkningseffektiv och ofta visat sig ge bra resultat är dropout [27]. Den grundläggande idén är att en kombination av flera olika modeller som tränats för samma uppgift nästan alltid ger en bättre prestation än om man endast förlitar sig på en ensam modell. Detta är dock en beräkningsmässigt kostsam operation eftersom träning av flera modeller tar lång tid. Dropout är en teknik som bygger på denna idé genom att den under varje iteration med en viss sannolikhet p sätter en nods aktivering till noll vilket förs vidare genom dess kopplingar till andra noder. På detta sätt tränas olika nätverksarkiteturer varje iteration.

Konkret tränar nätverk som använder dropout en stor mängd av delnätverk som skapas genom att en andel noder stängs av slumpmässigt med sannolikhet p [11]. På grund av denna randomisering i vilka noder som stängs av kan man också visa att dropout möjliggör att träna exponentiellt många olika arkitekturer. En intuitiv effekt av dropout är att nätverket inte kan förlita sig på att en

specifik nod ska ansvara för att upptäcka ett viss mönster i datan. Denna egenskap måste fördelas över ett större antal noder i fall att den nämnda noden skulle stängas ned under en iteration.

2.4.3 Early Stoppage

Ytterligare ett sätt att få en modell att generalisera bättre är den effektiva metoden early stoppage som ofta används för sin simplicitet [11]. Metoden är självförklarande på sättet att det enda vi gör är att avsluta modellens träning i ett tidigt skede. Genom att dela upp data i förväg i träningsdata och så kallad valideringsdata kontrollerar vi under varje träningsepok exaktheten på valideringsdata och när denna inte längre förbättras avslutas träningen. På detta sätt tillåter vi inte modellen att komma till den punkt att den börjat memorera alla träningsexempel. Denna metod att avsluta träningen tidigt är ett sätt att kontrollera nätverkskomplexiteten och att undvika överanpassning på träningsdata. När träningen inte längre ger generaliserbar förbättring till valideringsdata har vi påbörjat denna överanpassning till träningsdata och bör därför avsluta modellens träning [18].

2.4.4 Dataaugmentering

Regularisering definierades som metoder som kan förbättra exaktheten i klassificering av okänd data och att det kan tolkas som modellen enbart memorera alla exempel när den överanpassat till träningsdatat [6]. Med denna bild av överanpassning och regulariseringsmetoders mål att undvika denna memorering är en följdidé att försvåra möjligheten för nätverk att kunna memorera [11]. Därmed är det naturligt att utöka antalet exempel som nätverket får se och på så vis öka svårigheten att memorera exemplen.

Oftast uppstår ett problem i att det i praktiken är dyrt eller svårt att samla in fler unika exempel [11]. För att komma runt detta problem kan man använda dataaugementering. Man skapar då fler exempel genom att transformera den nuvarande datan och lägga till dessa modifierade exempel till träningsdatan. Exempelvis för bildklassificiering kan detta vara rotation och ändrad ljusstyrka på bilderna.

Dataaugmentering är en metod som speciellt inom bildklassificiering nästan alltid förbättrar modellens prestation, däremot finns fall där man bör vara försiktig innan användning av vissa transformationer. Exempelvis kan man tänka sig fallet att klassificera handskrivna siffror och att transformationer som att horisontellt eller vertikalt spegla bilden faktiskt skulle försämra modellens prestation eftersom detta innebär att bildens representation skulle kunna förändras. Att vertikalt och horisontellt spegla siffran sex ändrar också den korrekta klassificeringen för den nya bilden eftersom att den nu blivit siffran nio.

2.5 Metoder för sökning av hyperparametrar

L_2-regularisering kräver att vi hittar ett bra värde på λ för att metoden ska prestera optimalt. I denna sektion beskrivs några av de vanligaste och mest populära metoder metoder man kan använda för att hitta denna parameter. Vissa av dessa metoder generaliserar även för att söka andra hyperparametrar som exempelvis inlärningstakt, dropout-parameter p, batch-storlek m.m.

I fallet då vi vill optimera valet av regulariseringparameter λ vill vi experimentellt testa $\lambda \in \{\lambda_1, ..., \lambda_n\}$ för att avgöra vilket värde som presterar bäst på det specifika nätverket [11]. Om sökning ska göras för flera hyperparametrar där n är antalet olika värden och m antalet hyperparametrar blir tidskomplexiteten $\mathcal{O}(n^m)$, d.v.s det ökar exponentiellt med antalet värden på hyperparametrarna. Att ha effektiva metoder för att kunna söka efter dessa hyperparametrar underlättar därmed träningen av neurala nätverk och att spara på dessa beräkningar är mycket användbart i praktiken.

Randomiserad logspace En av de vanligaste metoderna för att söka hyperparametrar är att göra en rutnät-sökning genom att simulera värden $\{\lambda_1, ..., \lambda_n\}$ från en logaritmiskt uniform fördelning [28].

Studier har funnit både experimentellt samt teoretiskt stöd för att simulera dessa parametrar slumpmässigt från en logaritmiskt uniform fördelning och att detta presterar bättre än en linjär sökning [28]. Intuitivt kan det förklaras med att det ofta uppstår situationer där en parameter är mycket viktigare för nätverksprestation än andra och genom att simulera slumpmässigt uppstår fler instanser av denna viktiga parameter än om dessa skulle simuleras på ett linjärt rutnät. Detta beskrivs mer i detalj i den ursprungliga studien, som vi hänvisar till för djupare insikt.

Linjär sökning över intervall Med linjär sökning över intervall menas att värdet på parametern λ väljs från ett intervall $I = (a, b)$ med en steglängd m där $(b-a) \equiv 0 (\text{mod } m)$ så att $\lambda_n = a + nm$, $n \in \{0, k\}$ och $a + km = b$. Intuitivt görs alltså en systematisk genomsökning över ett intervall för att finna det värde på λ som passar bäst [28].

Balansmetoden Balansmetoden är en metod för att beräkna regulariseringsparametern λ, med hjälp av Tikhonovfunktionalen $J_\lambda(x)$ som är definierad på följande vis:

$$J_\lambda(x) = \frac{1}{2}\|\hat{\mathbf{y}}(\mathbf{x}) - \mathbf{y}(\mathbf{x})\|^2 + \lambda\psi(x) = \varphi(x) + \lambda\psi(x) \tag{46}$$

I fallet då vi använder oss av L_2-regularisering är $\psi(x) = \|W\|_{L_2}$. W är nätverkets vikter, $\mathbf{y}(\mathbf{x})$ är target-values och $\hat{\mathbf{y}}(\mathbf{x})$ är nätverkets beräknade output. Balansmetoden går ut på att man först gissar ett värde på λ för att sedan beräkna ett nytt värde efter varje iteration av träning enligt:

$$\lambda_{k+1} = \frac{\varphi(x_{\lambda_k})}{\gamma\psi(x_{\lambda_k})} \tag{47}$$

För enkelhetens skull har vi satt $\gamma = 1$ och detta kallas för ”Zero Crossing Method”. Genom att utföra denna sökning av hyperparametrar randomiserad har man kunnat visa att den resulterar i bättre prestation med samma beräkningskapacitet som om man utfört vanlig linjär sökning [29].

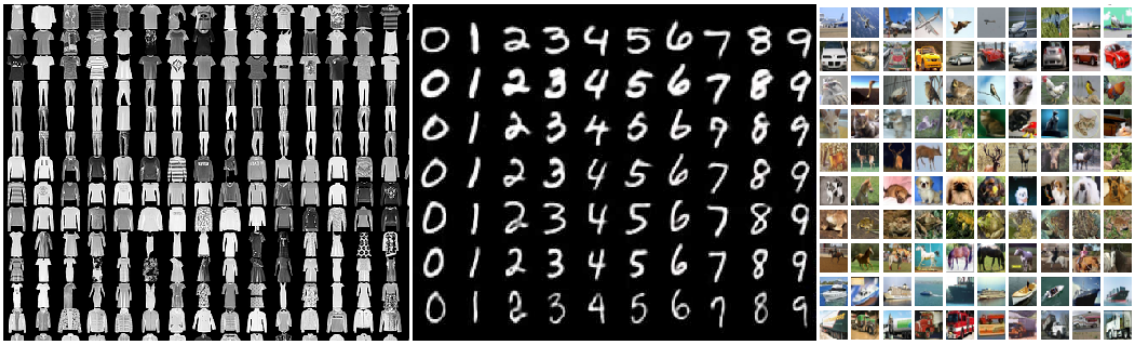
3 Metod

För att kunna jämföra olika regulariseringstekniker av neurala nätverk väljer vi att simulera egen data i två dimensioner som bidrar med att få en intuitiv förståelse för hur regularisering påverkar nätverkets beslutsgräns. Sedan undersöks regulariseringsmetoderna i mer detalj genom att se resultat på flera olika dataset, och undersöka hur dessa presterar i jämförelse med ett icke-regulariserat neuralt nätverk. I detta avsnitt beskrivs också nätverksarkitekturerna och översiktligt hur dessa implementerats, detta kan ses mer detaljerat i appendix eller [30].

3.1 Beskrivning av data för denna studie

En kort beskrivning av dataseten följer här i punktform och ett urval av bilderna visas i figur (3):

- **MNIST-Fashion** Ett standard dataset med bilder på olika klädesplagg [31]
- **MNIST** Ett standard dataset med bilder på handskrivna siffror [32]
- **CIFAR10** Ett dataset med bilder på olika djur och olika föremål indelade i 10 kategorier [33]
- **ISIC** Bilder på hudcancer som är separerat i bland annat godartad och elakartad cancer [34]



Figur 3: Från vänster: Ett urval av bilder från dataseten *MNIST-Fashion*, *MNIST* och *CIFAR10*

3.1.1 Simulerad linjär data separerad i två klasser

För den numeriska undersökningen av regulariseringens effekt genererades datapunkter slumpmässigt och uniformt på intervallet $[-1, 1]$ i både x- och y-led. De klassificeras sedan till en av två klasser enligt en förutbestämd beslutsgräns som utgörs av funktionen $f(x) = -x$ där punkter ovan funktionsytan ges en klassificering och punkter under ytan ges en annan. Efter att ha klassificerats flyttas varje datapunkt slumpmässigt enligt $y' = y(1 + \delta\beta)$ där δ är ett slumpmässigt genererat tal, och β är en brus-parameter. Vi ser exempel på sådan simulerad data i figurer (4).

3.1.2 Simulerad ickelinjär data separerad i fyra klasser

För den numeriska undersökningen av regulariseringens effekt genererades datapunkter separerade i fyra olika klasser i ett ickelinjärt mönster följande till enhetscirkeln med radien successivt ökande vilket skapar spiraler i datan. Punkter i en arm tilldelas samma klass. Efter generering används bruset $y' = y(1 + \delta\beta)$ med β som brus-parameter och δ som slumptal mellan $[-1, 1]$.

Inför skapandet av en visualiserbar yta för regulariseringens effekt valdes dessutom 20% av punkterna slumpmässigt ut och bytte plats med varandra för att ge ytterligare brus med punkter av olika klasstillhörighet i samma spiralarm. Vi ser exempel på sådan simulerad data i figurer (5).

3.1.3 Dataset med bilder

För dataseten MNIST, MNIST-Fashion och CIFAR-10 är alla bilder i samma storlek (28x28 för MNIST respektive 32x32 pixlar för CIFAR10 och MNIST-Fashion). För ISIC-dataset behövde vi göra en storleksändring och valde att skala om samtliga bilder till 224x224 pixlar. I tabell (1) ges en mer detaljerat beskrivning av dataseten som använts i denna rapport

Tabell 1: Beskrivning av dataset.

Dataset	Domän	Dimension	Träningsdata	Testdata
MNIST	Vision	784(28 ×28 svartvit)	60K	10K
MNIST-Fashion	Vision	3072(32 ×32 färg)	60K	10K
CIFAR10	Vision	3072(32 ×32 färg)	50K	10K
ISIC	Vision	150528 (224 ×224 färg)	18K	6K

För skeva dataset som exempelvis hudcancer-data där en övervägande majoritet är godartad är inte längre träningsaccuracy ett bra mått. Istället används precision, känslighet (*recall*) och specificitet [35]. I definitionerna används begreppen fp = ”falska positiva”, fn = ”falska negativa”, tp = ”sanna positiva”, och tn = ”sanna negativa”:

$$\begin{aligned} \text{Precision} &= \frac{tp}{tp+fp} \\ \text{Känslighet} &= \frac{tp}{tp+fn} \\ \text{Specificitet} &= \frac{tn}{tn+fp}. \end{aligned} \tag{48}$$

Formlerna kan tolkas som att precision beskriver andelen av alla exempel som klassificerades som positiva som faktiskt var positiva. Känsligheten beskriver andelen av alla positiva exempel som modellen klassificerade korrekt. Specificitet beskriver andelen av alla negativa exempel som modellen klassificerade korrekt. I praktiken kombineras ofta precision och känslighet genom att ta ett harmoniskt medelvärde till vad som kallas ett F1-score för att få ett enda mått att bedöma modellen utefter [36]

$$F = \frac{2}{\frac{1}{P} + \frac{1}{R}} \tag{49}$$

där P är precision och R känslighet eller recall.

3.2 Implementation

I detta arbete har vi implementerat ett generellt fully connected neuralt nätverk i MATLAB. Med en generell implementation menas att användaren enkelt kan ändra antalet lager och antalet noder i varje lager utan att behöva ändra något i framåtpropagerings- eller bakåtpropageringskod. Implementationen är därmed inte specifik för ett visst antal lager.

För träning av simulerad data i få dimensioner har kod skriven i MATLAB använts, men på de större dataset blev träningstid en begränsande faktor. De flesta dataset har över 50000 bilder som tar mycket lång tid att träna med ett tillräckligt stort nätverk med implementation i MATLAB. För träning på de större dataseten använde vi därför Pytorch som är ett ramverk för djup maskininlärning vilket kan köras på grafikkortet [37]. Pytorch underlättade också implementationen för konvolutionella neurala nätverk vilket gjorde det möjligt att jämföra regulariseringstekniker för både konvolutionella och fully connected neurala nätverk.

Arkitekturen för fully connected har varit på formen Nx5000x1000x500x75xC där N är storleken på indata, C antalet klasser och siffrorna därimellan betecknar storleken på de gömda lagren. I arkitekturen för de konvolutionella nätverken har ett modifierat VGG16 nätverk använts [38] för alla dataset förutom ISIC där ett modifierat ResNext nätverk använts [39]. ResNext modellen som använts är mer specifikt *ResNext-101 32x8d* som har blivit förtränat på 940 miljoner bilder, mer information om detta hittas på [40] och mer om inlärningsöverföring kan läsas i B.1. Vid träning på hudcancer-data har vi valt att modifiera kostnadsfunktionen till att bekosta falska negativa klassificeringar tio gånger så mycket som falska positiva för att öka känsligheten av modellen.

För att underlätta läsning av kod och tydliggöra dess struktur har vi skapat en Github samling för all kod som använts i denna studie. Koden för detta projekt kan ses på [30].

4 Resultat

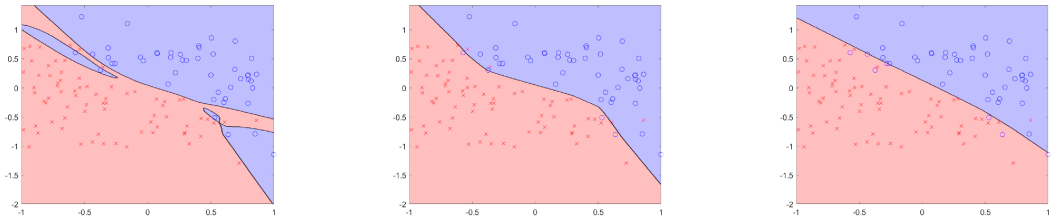
I detta avsnitt presenteras resultaten för klassificeringen av de olika dataseten. Samtliga resultat presenteras i tabeller där nätverkets exakthet med avseende på korrekt klassificering av data anges. För resultaten på genererad data presenteras också klassificeringsytorna visuellt för att ge en intuitiv bild över regulariseringens effekt.

4.1 Data separerad i två klasser

I tabell (2) presenteras resultaten för klassificering av datapunkter simulerade i två klasser kring funktionen $f(x) = -x$. Resultaten är givna för nätverket utan regularisering (med regulariseringsparameter $\lambda = 0$) samt för de bästa värdena av λ, med avseende på klassificering av testdata, genererade enligt de olika metoderna för val av regulariseringsparameter (se 2.5). Ett klassificeringsvärde av 1 innebär att alla exempel från testdatan klassificerats korrekt, och ett värde av 0 innebär att inget exempel klassificerats korrekt.

Tabell 2: Resultat för klassificering av data i 2 klasser med olika metoder att välja regulariseringsparameter. Resultaten anger andelen korrekt klassificerade exempel för tränings- och testdata.

λ (metod)	Klassificering träningsdata	Klassificering testdata
$\lambda = 0$ (icke-regularisering)	0.998	0.906
$\lambda = 0.032$ (logspace)	0.945	0.949
$\lambda =$ dynamiskt (balansprincipen)	0.987	0.919
$\lambda = 0.1$ (Linjär sökning)	0.943	0.946



(a) $\lambda = 0$ **(b)** $\lambda = 0.032$ **(c)** $\lambda = 0.551$

Figur 4: Beslutsgränser för olika värden på regulariseringsparametern λ

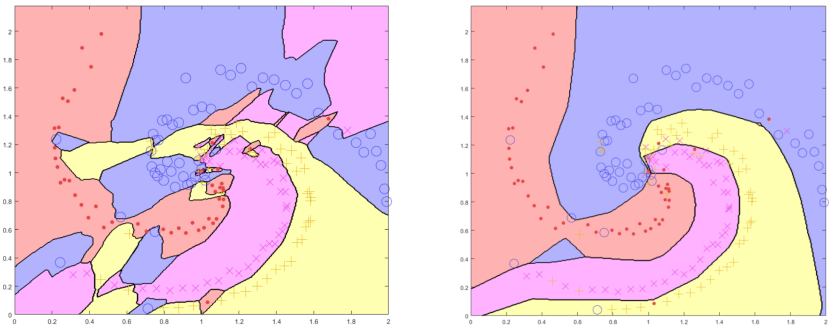
Figurerna (4) åskådliggör det neurala nätverkets klassificeringsyta för olika värden på lambda. Den vänstra figuren i (4) visar klassificeringsytan för ett icke-regulariserat neuralt nätverk plottat tillsammans med träningsdata. Den mellersta figuren i (4) visar klassificeringsytan tillsammans med träningsdata för $\lambda = 0.032$ genererat av ”logspace search” vilket gav högst andel korrekt klassificerade exempel från testdata. Vi väljer även att visa klassificeringsytan för $\lambda = 0.551$, vilket kan ses i den högre figuren i (4), genererat av logspace search vilken resulterar i en något lägre andel korrekt klassificerade testexempel (0.937) men som tydligt generaliserar till funktionen $f(x) = -x$ kring vilken de klassificerade punkterna är simulerade.

4.2 Data separerade i fler klasser

Enligt samma procedur som i (4.1) följer i tabell (3) resultaten för klassificering av ickelinjär data uppdelad i 4 klasser (se 3.1.2). Skillnaden i andel korrekt klassificerade testexempel är inte lika stor för den ickelinjära datan som för den linjära, detta beskrivs mer i diskussionen.

Tabell 3: Resultat för klassificering av data i 4 klasser med olika metoder att välja regulariseringsparameter. Resultaten anger andelen korrekt klassificerade exempel för tränings- och testdata.

λ (metod)	Klassificering träningsdata	Klassificering testdata
$\lambda = 0$ (icke-regularisering)	0.996	0.934
$\lambda = 0.0009$ (logspace)	0.990	0.942
$\lambda =$ dynamiskt (balansprincipen)	0.982	0.936
$\lambda = 0.02$ (linjär sökning)	0.936	0.928



(a) Icke-regulariserad beslutsgräns **(b)** Regulariserad beslutsgräns

Figur 5: Ickelinjärt data

Figurerna (5) åskådliggör effekten av regularisering på det neurala nätverket. För att så tydligt som möjligt illustrera hur detta fungerar har klassen slumpmässigt ändrats på 20 % av datapunkterna. Eftersom datan då innehåller flera punkter med udda klassificering för den arm de tillhör bildas utan regularisering ”öar” i beslutsytan vilket syns tydligt i figuren till vänster i (5). En person som betraktar testdatan kan tydligt se att den är strukturerad i en spiral, där varje spiralarm är en egen klass.

4.3 Data bilder

Följande är resultat efter träning med fully connected neuralt nätverk och konvolutionellt neuralt nätverk på samtliga dataset med bilder. I tabell (4) presenteras resultaten efter träning på dataseten MNIST, MNIST-Fashion och CIFAR10. Resultaten redogörs för tränings- och testdata som mängden korrekt klassificerade exempet där 1 innebär att samtliga exempel klassificerats korrekt.

Tabell 4: Resultat efter träning på dataseten MNIST, MNIST-fashion och CIFAR10. Resultaten presenteras som korrekt andel korrekt klassificerade träningsexempel.

	Fully Connected		**CNN**	
	Träningsdata	Testdata	Träningsdata	Testdata
	MNIST			
Icke regulariserad	1.00	0.978	0.999	0.994
L_2-regularisering	0.998	0.9826	0.999	0.995
Dropout	0.998	0.9827	0.999	0.995
	MNIST-Fashion			
Icke-regulariserad	0.9984	0.8968	1.00	0.930
L_2-regularisering	0.9907	0.9007	0.997	0.935
Dropout	0.994	0.9029	0999	0.934
	CIFAR10			
Icke-regulariserad	1.00	0.5315	1.00	0.9201
L_2-regularisering	0.998	0.5462	0.999	0.921
Dropout	0.994	0.5747	0.999	0.927

I tabeller (5) och (6) presenteras resultat efter träning på ISIC dataset. Termerna i vilka resultaten är beskrivna redogörs närmare för i sektion (3.1.3).

Tabell 5: Resultat fully connected på ISIC

	ISIC			
Metod	Precision	Känslighet	Specificity	F1-score
Icke-regulariserad	0.383	0.510	0.899	0.438
L^2 regularisering	0.384	0.609	0.889	0.472
Dropout	0.394	0.620	0.898	0.482

Tabell 6: Resultat CNN på ISIC

	ISIC			
Metod	Precision	Känslighet	Specificity	F1-score
Icke-regulariserad	0.704	0.597	0.967	0.646
L^2 regularisering	0.677	0.675	0.961	0.676
Dropout	0.744	0.671	0.972	0.706

5 Diskussion

Diskussionen behandlar först resultaten hörande till klassificeringen av de olika typerna av data. Därefter tas val av metoder och avgränsningar upp.

5.1 Simulerad data

För många metoder med neurala nätverk är det svårt att skapa en intuitiv bild av hur de påverkar nätverket, och neurala nätverk ses därför ofta som en *black box*, vilket innebär att man inte kan se exakt hur och var nätverkets beräkningar påverkar resultatet. Figurerna (4) ger en visuell bild av utgående klassificeringsytor från nätverket då det tränats på simulerad data. Från dessa ges en intuitiv bild om hur regularisering påverkar dessa klassificeringsytor.

Att överanpassning till data skulle motsvara att modellen memorerat exempel snarare än lärt sig den generella strukturen blir också tydligt i jämförelse av dessa figurer. Det blir tydligt att regularisering kan ses som ett sätt att försvåra processen att enbart memorera träningsexempel. Med ett högre värde på regulariseringsparametern verkar modellen föredra en visuellt simplare modell i både det linjära och ickelinjära fallet vilket ses i figurer (4) och (5).

I tabell (2) kan vi se valen av regulariseringsparameter som ger högst andel korrekt klassificerade tränings- och testexempel, och det är även tydligt att alla regulariseringsmetoder som använts ger bättre resultat än den icke-regulariserade. Då vi betraktar mellersta och högra figurerna i (4) kan vi se att den klassificeringsyta som ger högst resultat inte är den som bäst överensstämmer med den underliggande fördelningen som vi vet är simulerad kring funktionen $f(x) = -x$. Vi vet från stora talens lag att den klassificeringsyta som med högst sannolikhet klassificerar en punkt genererad av modellen är den som visualiseras i den högra figuren i (4) då den är mer lik den underliggande fördelningen. Vidare kan vi dra slutsatsen att ett mycket högt antal träningsexempel skulle resultera i en klassificeringsyta mer lik den högra figuren i 4. Detta pekar på betydelsen av att ha en stor mängd data att träna och testa neurala nätverk på.

Ovan noterade vi att regularisering verkar skapa en simplare klassificieringsyta, men att detta skulle få en modell att generalisera bättre är något som är svårare att argumentera för och ännu svårare att matematiska bevisa. Djup maskininlärning refererar ofta till empiriska studier på grund av denna svårighet att framställa matematiska bevis, och i fallet av L_2 regularisering och regularisering i allmänhet blir argumenten mer filosofiska, regularisering kan förklaras utifrån ”Ockhams rakkniv”-principen, snarare än baserad på teori [8]. För val av regulariseringsparameter så gav logspace bäst resultat för båda dataseten men för att kunna avgöra mer konkret vilken metod som

är bäst och när den är bäst så skulle metoderna behöva appliceras på mer data och i fler dimensioner. Att logspace genererar bättre resultat än linjär sökning kan förklaras av att sökintervallet består av ett bredare spann och att möjligheten att hitta en optimal regulariseringsparameter är större.

5.2 CIFAR10, MNIST och MNIST-Fashion

Från de bilddataset vi valt att fokusera på observeras att konvolutionella neurala nätverk är bättre anpassade till dataset med bilder än vad fully connected är. Vi ser också att regularisering har hjälpt modellen att generalisera bättre i alla de fall som undersökts, både för konvolutionella och fully connected neurala nätverk. Regulariseringen tycks ha en mer drastisk effekt på data i fler dimensioner. Detta kan ses på resultaten då regularisering har störst effekt på ISIC och CIFAR10 där bilderna är i färg och större i storlek.

På CIFAR10 observeras att regularisering har störst effekt på resultatet då den tränats med ett fully connected neuralt nätverk. Från resultaten verkar det också som att dropout är den metod som bidragit mest i jämförelse med L_2-regularisering, som presterat marginellt bättre i enstaka fall.

För att underbygga argumentet att konvolutionella neurala nätverk är bättre anpassade till bilder ser vi närmare på resultaten från tabell (4). Vi ser då att specifikt för CIFAR10 ger dropout flera procentenheters förbättring, däremot är intervallet mellan träning- och testdata fortfarande mycket stor med över 40 procentenheters skillnad.

I dessa fall när skillnaden mellan träning- och testdata är mycket stor verkar det som hjälper modellen att generalisera bäst vara ett byte av arkitektur till ett konvolutionellt neuralt nätverk vilket verkar ha en regulariserande effekt. Vi noterar att ett byte till ett konvolutionellt neuralt nätverk förbättrade testaccuracy med över 35 procentenheter på CIFAR10 både med och utan regularisering jämfört med resultaten från fully connected. Här uppmärksammas också att dropout är den metod som gett störst effekt. På CIFAR10 ger dropout flera procentenheters förbättring jämfört med resultat då L_2 regularisering använts. Att dropout presterar bäst är i stöd med resultaten från den ursprungliga dropout-studien [27].

5.3 ISIC

Inom medicinsk klassificering finns en avvägning mellan precision och känslighet vilket gör jämförandet av våra resultat med dermatologer och läkares mer komplicerat. På hudcancer-data är denna avvägning mellan att göra en högre andel falska positiva eller falska negativa klassificeringar. I vårt fall har vi prioriterat mot känslighet och anser att det är acceptabelt att modellen har fler falska positiva för att minska chansen att bedöma en cancer som godartad när den faktiskt är malign. För att uppnå detta har vi valt att ge en falsk negativ klassificering en 10 gånger större kostnad än vad vi ger en falskt positiv klassificering. För en eventuellt tillämpning är denna kostnad viktig att analysera närmare för att nå ett verktyg som kan ge hög andel korrekta klassificeringar och minska trycket på vården men inte heller friskförklara sjuka patienter.

Då vi bedömmer utfallen från vårt arbete, vilka kan ses i tabeller (5-6), med resultat presenterade i en jämförande studie [41] finner vi att vår modell är i linje med de flesta dermatologer, läkare och andra neurala nätverk. Detta även fast vi haft tillgång till betydligt mindre data. En anledning till att vår modell ger så pass bra resultat kan vara den stora mängd förträning vår ResNext-modell gjort. Det bör tilläggas att undersökningarna är gjorda på olika data vilket kan bidra till svårigheten att jämföra resultaten.

Regularisering gav en större effekt på hudcancer-data jämfört med dess effekt på övriga bilddataset. Dropout gav bäst resultat i jämförelse med L_2-regularisering för både fully connected och konvolutionellt neurala nätverk. Att regulariseringen har en så stor effekt på just hudcancer-data kan tänkas bero på att bildernas storlek är större och att det därmed finns mer information som

modellen potentiellt skulle kunna memorera. Det kan också bero på att storleken på datasetet är cirka en tredjedel av resterande dataset, och vi vet att data augmentering kan ha en regulariserande effekt, vilket skulle kunna minska L_2- och dropout-regulariseringens effekt.

5.4 Förslag till framtida forskning

I denna rapport studerades dropout och L_2 regularisering för sig, det hade det varit intressant att kombinera dessa och se om detta bidrar till bättre regulariseringseffekt. Detta undersöks inte i ursprungliga dropout studien.

Genom att undersöka fler regulariseringstekniker hade bättre slutsatser kunnat dras, i detta fall har endast två av de mest populära undersökts. Det finns många fler tekniker som är vanliga och används som hade varit intressant att se hur de presterar för att kunna ge ännu bättre riktlinjer om vilka man borde använda. En stor begränsning för detta har varit att modellerna som krävs för att prestera bra behöver vara oerhört stora. Det tar över två dagar att träna modellerna på hudcancerdata med det grafikkort som använts för denna studie, med ett nyare och kraftfullare grafikkort hade detta gett mer tid att simulera fler resultat med andra regulariseringstekniker.

I denna studie tog det mycket tid att implementera ett generellt fully connected nätverk i MATLAB, men om man hade börjat med att använda ett ramverk som PyTorch hade mycket tid ha kunnat sparats. Denna tid hade istället kunnat lagts på att simulera fler resultat med fler regulariseringstekniker, dataset och nätverksarkitekturer.

För att nå bättre klassificering på hudcancer-data bör fler nätverksarkitekturer prövas. Det finns också ett flertal etiska överväganden, bland annat över hur kostnadsfunktionen ska utformas och hur verktygen kan integreras i medicinska instrument, som med fördel skulle kunna diskuteras av en grupp med bred bakgrund. Denna grupp skulle förslagsvis bestå av sjukhuspersonal, datavetare, filosofer m. fl. för att en så stor andel viktiga faktorer som möjligt ska belysas i den eventuella utvecklingen av medicinska hjälpmedel.

6 Slutsatser

De huvudsakliga slutsatserna från studien är att regularisering har en positiv effekt på resultaten gällande klassificering av alla typer av undersökt data. Visualiseringen av resultaten på simulerad data visar att en effekt av regularisering är att det neurala nätverket genererar en mindre komplicerad klassificeringsyta än utan regularisering. Detta stärker den intuitiva bilden av regulariseringens generaliserande effekt. Resultaten på klassificering av bilddata är samstämmiga i att konvolutionella neurala nätverk lämpar sig bättre för bildigenkänning än fully connected neurala nätverk, vilket stämmer överens med tidigare studier. Vi ser också att dropout konsekvent presterar bättre än L_2 regularisering vilket är i linje med resultat från ursprungliga dropout studien [27]. Detta resultat generaliserar också när modellerna tränas på hudcancer data som inte undersökts i tidigare studier.

Resultaten på klassificering av huddata är i klass med dermatologer- och läkares. Vi drar slutsatsen att det finns stor potential för artificiell intelligens, och specifikt neurala nätverk, att integreras i vården. Här finns stora möjligheter att effektivisera och underlätta arbetet för läkare, sjuksköterskor och övrig vårdpersonal.

Referenser

[1] A. M. Turing, "Computing machinery and intelligence," in *Parsing the Turing Test*. Springer, 2009, pp. 23–65.

[2] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon, "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955," *AI magazine*, vol. 27, no. 4, pp. 12–12, 2006.

[3] J. Hendler, "Avoiding another ai winter," *IEEE Intelligent Systems*, no. 2, pp. 2–4, 2008.

[4] J. Lighthill *et al.*, "Artificial intelligence: a paper symposium," *Science Research Council, London*, 1973.

[5] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 44, no. 1.2, pp. 206–226, 2000.

[6] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015, http://www.neuralnetworksanddeeplearning.com.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] J. Sietsma and R. J. Dow, "Creating artificial neural networks that generalize," *Neural networks*, vol. 4, no. 1, pp. 67–79, 1991.

[9] I. A. Basheer and M. Hajmeer, "Artificial neural networks: fundamentals, computing, design, and application," *Journal of microbiological methods*, vol. 43, no. 1, pp. 3–31, 2000.

[10] I. Saftić *et al.*, "Max tegmark, life 3.0: Being human in the age of artificial intelligence," *Croatian Journal of Philosophy*, vol. 18, no. 54, pp. 512–516, 2018.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[12] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2016.

[13] T. Parr and J. Howard, "The matrix calculus you need for deep learning," *arXiv preprint arXiv:1802.01528*, 2018.

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[15] D. H. Hubel and T. N. Wiesel, "Receptive fields of single neurons in the cat's striate cortex," *Journal of Physiology*, vol. 148, pp. 574–591, 1959.

[16] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[17] D. Pedamonti, "Comparison of non-linear activation functions for deep neural networks on mnist classification task," *arXiv preprint arXiv:1804.02763*, 2018.

[18] C. M.Bishop, *Neural Networks for Pattern Recognition*. Calendon Press, Oxford, 1996.

[19] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[21] T. Kurbiel and S. Khaleghian, "Training of deep neural networks based on distance measures using rmsprop," *arXiv preprint arXiv:1708.01911*, 2017.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[23] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," 04 2018.

[24] P. Zhao and T. Zhang, "Accelerating minibatch stochastic gradient descent using stratified sampling," 05 2014.

[25] T. Jayalakshmi and S. A., "Statistical normalization and back propagation for classification," *International Journal Computer Theory Engineering (IJCTE)*, vol. 3, pp. 89–93, 01 2011.

[26] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds. Morgan-Kaufmann, 1992, pp. 950–957. [Online]. Available: http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf

[27] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[28] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/jmlr/jmlr13.html#BergstraB12

[29] L. Beilina, "Numerical analysis of least squares and perceptron learning for classification problems," *arXiv preprint arXiv:2004.01138*, 2020.

[30] E. Johansson, B. Krook Willén, A. Persson, and M. Sajland, *Kod för detta arbete*, 2020, https://github.com/Kandidatarbete-MVEX01-20-07/Kandidatarbete.

[31] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[32] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[33] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, vol. 55, 2014.

[34] "Isic-archive," https://www.isic-archive.com/.

[35] D. L. Olson and D. Delen, *Advanced Data Mining Techniques*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[36] Y. Sasaki, "The truth of the f-measure," *Teach Tutor Mater*, 01 2007.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[38] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[39] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," 2016.

[40] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten, "Exploring the limits of weakly supervised pretraining," 2018.

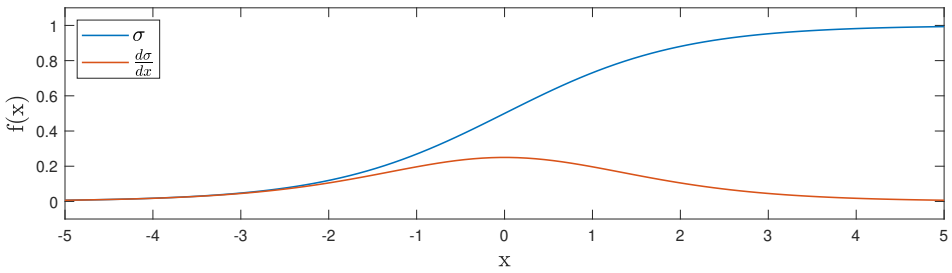
[41] M. Goyal, T. Knackstedt, S. Yan, A. Oakley, and S. Hassanpour, "Artificial intelligence-based image classification for diagnosis of skin cancer: Challenges and opportunities," 2019.

[42] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *ArXiv*, vol. abs/1502.03167, 2015.

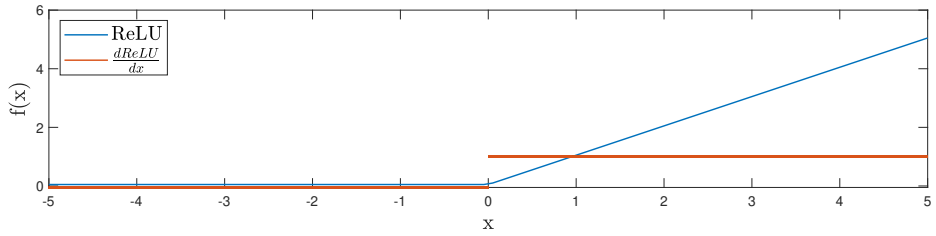
[43] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

Appendix

A Figurer



Figur 6: Figuren visar sigmoidfunktionen och dess derivata i intervallet $x \in (-5, 5)$



Figur 7: Figuren visar ReLU-funktionen och dess derivata i intervallet $x \in (-5, 5)$

B Metoder

B.1 Inlärningsöverföring

Neurala nätverk presterar betydligt bättre med enorma datamängder. För att få nätverk att prestera bra på fall där det inte finns tillgång till mycket data finns en metod som heter inlärningsöverföring [11]. Denna metod utgår från att ett nätverk tränats i ett utfall, exempelvis att känna igen olika bilar från bilder. Detta nätverk och dess vikter överförs sedan till en annan miljö genom att använda dessa vikter som initialisering till en annan uppgift, exempelvis att lära känna igen olika lastbilar. Det nätverk som använder dessa förtränade vikter som initialisering till den nya uppgiften kommer mest troligt prestera bättre då dessa två uppgifter är lika varandra. Som regel kommer inlärningsöverföring prestera bättre desto mer lika uppgifterna är, men troligtvis kommer denna metod också hjälpa i de fall där uppgifterna skiljer sig mer än för det givna exemplet. Specifikt i bildklassificiering behöver nätverket lära sig många delar som är lika varandra, exempelvis att extrahera kanter, former, och hur dessa förändras i olika ljusförhållanden mm. Eftersom att de flesta typer av bildklassificeringar innehåller många gemensamma moment kan vikter som tränats på ett brett urval av bildklassificerings-uppgifter med fördel användas som initiering.

B.2 Binary cross entropy

Vi betraktar i detta delavsnitt fallet att ge data en av två klassificeringar vilket kan tolkas som att avgöra om ett exempel tillhör en klass eller inte. Det generella fallet är att ha K stycken oberoende binära klassificeringsproblem, exempelvis att avgöra om en bild innehåller några av K stycken olika objekt. Vi kan då använda oss utav ett neuralt nätverk med K stycken noder i utgående

lagret och tilldela varje nod en sigmoid-aktiveringsfunktion. Sannolikhetsfunktionen för den givna klassifieringsvektorn $\mathbf{y}$ givet exemplet $\mathbf{x}$ och nätverkets vikter W och biaser B kan då skrivas

$$p(\mathbf{y}|\mathbf{x}, W, B) = \prod_{i=1}^{K} \hat{y}_i^{y_i}(1-\hat{y}_i)^{(1-y_i)} \tag{50}$$

där $y_i \in \{0, 1\}\ \forall\ i$ och $\hat{y}_i$ är den i:te utgående nodens aktiverade värde. Genom att ta negativa logaritmen av (50) erhålles följande kostnadsfunktion kallad ”binary cross entropy loss”

$$C(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y}(\mathbf{x})) = -\sum_{i=1}^{K}(y_i \ln(\hat{y}_i) + (1-y_i)\ln(1-\hat{y}_i)). \tag{51}$$

Från (51) kan vi se att för ett träningsexempel som klassificeras väldigt fel, exempelvis $y_i(x) = 1$ och $\hat{y}_i(x) = 0.001$, så blir kostnaden mycket stor eftersom $ln(0) = -\infty$. En jämförelse med MSE för $y_i(x) = 1$ och $\hat{y}_i(x) = 0.001$ visar att MSE genererar en mycket lägre kostnad för ett kraftigt felklassificerat exempel vilket motiverar användningen av binary cross entropy loss i klassificeringsproblem. Anledningen till att ta negativa logaritmen av (50) är att (51) då fås på en form som gör det ekvivalent att minimera (51) som att maximera sannolikheten i (50). Detta är åtråvärt då det är standard att minimera kostnadsfunktioner.

B.3 Batch normalisering för konvolutionella neurala nätverk

Istället för att bara dra fördel av att normalisera indatan i en lärprocess kan man utföra upprepade normaliseringar genom hela nätverket. Denna metod kallas Batch Normalisering [42] och fungerar genom att man normerar värdena som går in i neuronerna i varje lager enligt:

$$\tilde{z}_i = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{52}$$

med medelvärde

$$\mu = \frac{1}{N}\sum z_i \tag{53}$$

och varians

$$\sigma^2 = \frac{1}{N}\sum (z_i + \mu)^2 \tag{54}$$

där z_i är värdena in ett godtyckligt lager med N stycken neuroner. ϵ är en godtycklig term (liten) som adderas till σ^2 för att undvika division med 0. Dessutom medför ϵ en regulariserande effekt till nätverket genom att addera brus liknande funktionen av regulariseringstekniken dropout (se 2.4.2). För att undvika att information försvinner i normaliseringen modifieras $\tilde{z}$ så att

$$y = \gamma\tilde{z} + \beta \tag{55}$$

är det värde som går in i neuronens aktiveringsfunktion där γ och β är parametrar som nätverket lär sig i bakåtpropageringen. γ kan ses som en skalningsparameter och β som en förskjutningsparameter. Genom att använda batch normalisering kan man bygga djupare nätverk och man åstadkommer en snabbare läroprocess med möjlighet att välja ett högre värde på inlärningsparametern [42].

B.4 Initiering av nätverkets vikter och biaser

Processen att träna ett neuralt nätverk är som vi sagt tidigare att uppdatera dess vikter och biaser på ett sådant sätt att nätverket löser givna uppgifter med ökande tillförlitlighet. En naturlig frågeställning som uppstår är till vilket värde vikter och bias ska initieras inför träningsprocessen. För enkelhetens skulle väljer vi att initiera alla biaser till 1. Låt oss nu pröva idén att sätta alla vikter till samma värde, exempelvis

$$w_{ij}^{l} = 1\ \forall\ i, j, l. \tag{56}$$

Om vi följer strukturen i Figur 1 ser vi att varje nod är kopplad till samtliga noder i nästkommande lager. I fallet 56 innebär det att samtliga noder tillhörande ett lager kommer ge samma aktiveringsvärde. Genom att följa ekvation 6 till 13 inses att

$$\frac{dC}{dw_{ij}^{l}} = \frac{dC}{dw_{mn}^{l}} \ \forall \ i, j, m, n \tag{57}$$

och

$$\frac{dC}{db_{i}^{l}} = \frac{dC}{db_{j}^{l}} \ \forall \ i, j \tag{58}$$

vilket innebär att alla vikter i lager l kommer uppdateras till samma värde och att alla biaser i lager l kommer uppdateras till samma värde. Uppdateringen kan dock skilja sig mellan olika lager, men slutsatsen blir att alla neuroner tillhörande samma lager alltid kommer ge samma aktivering, oavsett hur länge vi tränar nätverket. Detta motsvarar ett neuralt nätverk med l gömda lager, men med endast en neuron i varje lager och alltså är idén att initiera alla vikter till samma värde att förkasta.

Ett standardsvar på frågan om initiering av vikter är att välja vikterna enligt normalfördelning med medelvärde $\mu = 0$ och standardavvikelse $\sigma = 1$. Denna metod åtgärdar problemet med att samtliga noder tillhörande ett och samma lager alltid resulterar i samma aktivering samt effektiviserar träningen genom att randomisera valet av vart i hyperrymden vi börjar leta efter ett minimum till kostnadsfunktionen.

Däremot kan andra problem uppstå beroende på valet av standardavvikelse. Från ekvation (9) ser vi att i beräkningen av δ^l sker en matrismultiplikation av viktmatriserna $(W^{l'})^T$ hörande till samtliga lager där $l' > l$. Om viktmatriserna innehåller en stor andel mycket små värden kan matrismultiplikationerna resultera i ett mycket litet värde vilket i sin tur leder till att uppdateringarna av vikter och bias i ekvation (13) blir försvinnande liten. Detta fenomen kallas för just försvinnande gradient och får till följd att vikter och bias upphör att förändras mellan uppdateringar och att lärandeprocessen således klingar av. Motsatsen infaller om viktmatriserna innehåller en hög andel stora vikter ($w > 1$) vilket då skulle innebära att δ^l i ekvation (9) enligt motsvarande resonemang divergerar och att uppdateringarna beskrivna i ekvation (13) blir mycket stora. Detta kallas för exploderande gradient och har som följd att lärandeprocessen divergerar. Besvär med försvinnande- och exploderande gradient sker främst i mycket djupa neurala nätverk.

I en artikel från 2015 publicerades Kaiming-initiering [43] där vikter $w_{ij}^{l} \ \forall \ i, j$ initieras enligt normalfördelning med $\mu = 0$ och $\sigma = \sqrt{2/n}$ där $n =$ antalet aktiveringar i lager l. Kaiming-initiering används i kombination med ReLU som aktiveringsfunktion och visar sig vara en god metod för att kringå gradientproblemen beskrivna ovan.

C Matlabkod

Här visas MATLAB-koden som skrivits i projektet. All kod finns även tillgänglig för nedladdning på Github [30].

Inehåll

1 Main

```
%% Main file, training of network and all hyperparameters etc decided here
clear all; close all; clc;

global total_layers; % make the amount of layers global, convenient.

%%%% ---------- Add all subfolders ----------%%%%
addpath('import_data/') % Functions to import the data, simulate. Makes code cleaner.
addpath('simulatedata/') % Functions to simulate data
addpath('neuralnet/') % All necessary functions for forward, backward, etc.
addpath('optimizers/') % Adam Optimizer, GD with momentum, vanilla GD
addpath('utils/') % Nice utilization functions, like get_accuracy etc.
addpath('activations/') % Nice utilization functions, like get_accuracy etc.

%%%% ---------- Import data ----------%%%%
% Below is import for simulation of non-linear 2D
% for linear dataset, import with function: generate_linear_2d
points_each_class = 50; D = 2; K = 4; noise_rate = 0.1; plot_data = true;
[X_train, y_train, X_test, y_test] = import_nonlinear_data(points_each_class, D, K, noise_rate, plot_data);

%%
%%%% ---------- Run Normalization ----------%%%%
[X_train,X_test] = normalization(X_train, X_test);

y_train=y_train';
y_test=y_test';

disp("Data loaded")

%% ---------- NN Hyperparameters & Training of NN ----------
epochs = 10000;
learning_rate = 1e-4; % (most ?) important hyperparameter
```

```
mini_batch_size = 128;
lambda = 0.0;
nodes = [2, 250, 100, 50, 4]; % Layer node setup
total_layers = length(nodes) - 1;
activation = "relu";
reg_method = "";
beta1 = 0.9; % Adam hyperparam, keep beta1 = 0.9
beta2 = 0.99; % Adam hyperparam, keep beta2 = 0.99
adam_iterations = 1;

%%%% ---------- Run weight initialization ----------%%%%
parameters = initialize_weights(nodes);

%%%% ---------- Training ----------%%%%
m_train = size(X_train, 1);

for epoch = 1:epochs
    indices = randperm(m_train);
    X_train = X_train(indices, :);
    y_train = y_train(1, indices);

    if mod(epoch, 1) == 0
        [accuracy, ~] = check_accuracy(X_train, y_train,
            parameters, lambda, activation);
        disp('Train Accuracy is: ' + string(accuracy))
    end

    for batch = 0:floor(m_train/mini_batch_size)
        start_position = 1 + batch*mini_batch_size;
        end_position = min(start_position +
            mini_batch_size, m_train) - 1;

        batch_X = X_train(start_position:end_position, :)
            ;
        batch_y = y_train(1, start_position:end_position)
            ;
        m_batch = size(batch_X, 1);

        [~, loss, cache] = forward_propogation(batch_X,
            batch_y, parameters, m_batch, lambda,
            activation);
        gradients = back_propogation(cache, batch_y,
            parameters, size(batch_X, 1), lambda,
            activation);
        parameters = gradient_descent_adam(gradients,
            parameters, learning_rate, beta1, beta2,
```

```
            adam_iterations);
        adam_iterations = adam_iterations + 1;
    end
end

[accuracy, ~] = check_accuracy(X_train, y_train,
    parameters, lambda, activation);
disp('Train Accuracy is: ' + string(accuracy))

[accuracy, ~] = check_accuracy(X_test, y_test, parameters
    , lambda, activation);
disp('Test Accuracy is: ' + string(accuracy))

%% Plot Decision Boundary: Only possible for 2D data
clc;
decision_boundary(parameters, lambda, activation, y_train
    , X_train)
decision_boundary(parameters, lambda, activation, y_test,
     X_test)

[accuracy, ~] = check_accuracy(X_train, y_train,
    parameters, lambda, activation);
disp('Train Accuracy is: ' + string(accuracy))

[accuracy, ~] = check_accuracy(X_test, y_test, parameters
    , lambda, activation);
disp('Test Accuracy is: ' + string(accuracy))
```

2 Huvudfunktioner

2.1 Initializering av vikter

```
function [parameters] = initialize_weights(node_vec)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

keySet = {};
valueSet = {};

for i = 1:(length(node_vec) - 1)
    l0 = node_vec(i);
    l1 = node_vec(i+1);

    W_i = strcat('W', num2str(i));
    b_i = strcat('b', num2str(i));

    keySet{end + 1} = W_i;
    valueSet{end + 1} = randn(l0,l1).*sqrt(2 ./ l0);

    keySet{end + 1} = b_i;
    valueSet{end + 1} = zeros(1,l1);

    v_dw = strcat('v_dW', num2str(i));
    v_db = strcat('v_db', num2str(i));

    keySet{end + 1} = v_dw;
    valueSet{end + 1} = zeros(l0,l1);

    keySet{end+1} = v_db;
    valueSet{end + 1} = zeros(1,l1);

    s_dw = strcat('s_dW', num2str(i));
    s_db = strcat('s_db', num2str(i));

    keySet{end + 1} = s_dw;
    valueSet{end + 1} = zeros(l0,l1);

    keySet{end+1} = s_db;
    valueSet{end + 1} = zeros(1,l1);
end
disp("Weight Initialization complete")
parameters = containers.Map(keySet,valueSet);
```

2.2 Forward propagation

```
function [probs, loss, cache] = forward_propogation(h0, t
    , parameters, m, ...
                                                    lambda
                                                        ,
                                                        activation
                                                        )
%forward propogation

global total_layers;
h_l = h0;
cache_keys = {'h0'};
cache_values = {h0};
regularization = 0;

for l = 1:total_layers
    W_l = parameters(strcat('W',num2str(l)));
    b_l = parameters(strcat('b',num2str(l)));

    regularization = regularization + sum(W_l .* W_l, '
        all');
    z_l = h_l*W_l + b_l;

    if activation == 'relu'
        h_l = relu(z_l);
    elseif activation == 'sigmoid'
        h_l = sigmoid(z_l);
    end

    cache_keys{end+1} = strcat('h', num2str(l));
    cache_values{end+1} = h_l;
end

probs = exp(z_l) ./ (sum(exp(z_l), 2));
cache_keys{end+1} = 'probs';
cache_values{end+1} = probs;

indices = sub2ind(size(probs), 1:m, t+1);
noreg_loss = sum(-log(probs(indices)), 2) ./ m;
reg_loss = 0.5 .* lambda .* regularization;
loss = noreg_loss + reg_loss;

% loss= noreg_loss;
% data for balancing principle L2 regularization
psi = regularization;
```

```
40 cache_keys{end+1} = 'psi';
41 cache_values{end+1} = psi;
42 cache_keys{end+1} = 'noreg_loss';
43 cache_values{end+1} = noreg_loss;
44
45 cache = containers.Map(cache_keys, cache_values);
46 end
```

2.3 Backward propagation

```
function [gradients] = back_propogation(cache, t,
    parameters, m, lambda, ...
                                        activation)
%backward propogation
gradient_keys = {};
gradient_values = {};
global total_layers;
L = double(parameters.Count) / 4; % number of layers
probs = cache('probs');
dz_l = probs;
indices = sub2ind(size(dz_l), 1:m, t+1);
dz_l(indices) = (dz_l(indices) - 1);
dz_l = dz_l ./ m;

% Ignore this: dz_l = -(t-y) .* y .* (1- y);

for l = total_layers:-1:2
    h_prev = cache(strcat('h',num2str(l-1)));
    W_l = parameters(strcat('W',num2str(l)));

    dW_l = h_prev' * dz_l + lambda * W_l;
    db_l = sum(dz_l,1);

    dh_prev = dz_l * W_l';

    if activation == 'relu'
        dz_l = dh_prev .* (h_prev > 0);
    elseif activation == 'sigmoid'
        dz_l = dh_prev .* h_prev .* (1- h_prev);
    end

    gradient_keys{end+1} = strcat('dW', num2str(l));
    gradient_values{end+1} = dW_l;

    gradient_keys{end+1} = strcat('db', num2str(l));
    gradient_values{end+1} = db_l;
end

h0 = cache('h0');
dW_1 = h0' * dz_l;
db_1 = sum(dz_l,1);

gradient_keys{end+1} = 'dW1';
gradient_values{end+1} = dW_1;
```

```
44
45  gradient_keys{end+1} = 'db1';
46  gradient_values{end+1} = db_1;
47
48  gradients = containers.Map(gradient_keys, gradient_values
        );
49  end
```

3 Aktiveringsfunktioner

3.1 Sigmoid

```
function g = sigmoid(z)
g = 1 ./ (1 + exp(-z));
end
```

3.2 ReLu

```
function g = relu(z)
g = max(0,z);
end
```

4 Datagenerering

4.1 Generering av 2D-data efter en rät linje

```
function [states,labels] = generate_linear_2d(points, a,
    b, c, noise_rate)

states = rand([points,2])*2 - 1; % to get between [-1,1]
labels = zeros(1,points);

for idx = 1:length(states)
    if [a,b] * states(idx, :)' + c > 0
        labels(1, idx) = 1;
    else
        labels(1, idx) = 0;
    end
end

alpha = 1 .* (rand([points,1])*2 - 1); % [-1,1]
% alpha = normrnd(0,3,[points,1]);

states(:,2) = states(:,2) + states(:,2) .* alpha .*
    noise_rate;
end
```

4.2 Generering av 2D-data i form av en spiral

```
function [X,y] = generate_nonlinear_2d(N,D,K, noise_rate)
% This function generates nonlinear data in spiral form,
    with K classes,
% N points per class and in D dimensions

clc;

X = zeros(N*K,D);
y = zeros(N*K,1);

for j = 0:(K-1)
    indices = (N*(j) + 1: N*(j+1)); % generate N points
        for this specific class
    size(indices)
    r = linspace(0, 1, N); % radius of circle
    t = linspace(j*4, (j+1)*4, N); % + randn([1,N])*0.2
    X(indices,:) = [r.*sin(t); r.*cos(t)]';
    y(indices) = j;
end

% (X: exempel, dim), y: (dim, exempel)
X(:,2) = X(:,2) + 1;
X(:,1) = X(:,1) + 1;

alpha = 1 .* (rand([N*(K),1])*2 - 1); % [-1,1]
X(:,2) = X(:,2) + X(:,2) .* alpha .* noise_rate;

% additional noise: switches addnoise% of datapoints to
    random class

addnoise=0.2;

for i = 1:length(y)
    if addnoise>=rand
        y(i)=randi(K,1)-1;
    end
end
end
```

5 Dataimportering

5.1 Importera linjär 2D-data

```
function [X_train, y_train, X_test, y_test] =
    import_lineardata(points, a, b, c, plot_data,
    noise_rate)
%  Define linear model to separate data: ax + by + c = 0

[X_train, y_train] = generate_linear_2d(points, a, b, c,
    noise_rate);
[X_test, y_test] = generate_linear_2d(points, a, b, c,
    noise_rate);
% [X_validation, y_validation] = generate_data_2d(0.1*
    points, a, b, c, noise_rate);

if plot_data == true
    figure(1)
    indices_blue = find(y_train==1);
    indices_red = find(y_train==0);
    plot(X_train(indices_red,1), X_train(indices_red,2),
        'x', 'color', 'red')
    hold on;
    plot(X_train(indices_blue,1), X_train(indices_blue,2)
        , 'o', 'color', 'blue')
    title('Training data')

    indices_blue = find(y_test==1);
    indices_red = find(y_test==0);
    figure(2)
    plot(X_test(indices_red, 1), X_test(indices_red,2), '
        x', 'color', 'red')
    hold on;
    plot(X_test(indices_blue,1), X_test(indices_blue,2),
        'o', 'color', 'blue')
    title('Test data')
end
disp("Loaded data")
end
```

5.2 Importera linjär 2D-data

```
function [X_train, y_train, X_test, y_test] =
    import_lineardata(points, a, b, c, plot_data,
    noise_rate)
% Define linear model to separate data: ax + by + c = 0

[X_train, y_train] = generate_linear_2d(points, a, b, c,
    noise_rate);
[X_test, y_test] = generate_linear_2d(points, a, b, c,
    noise_rate);
% [X_validation, y_validation] = generate_data_2d(0.1*
    points, a, b, c, noise_rate);

if plot_data == true
    figure(1)
    indices_blue = find(y_train==1);
    indices_red = find(y_train==0);
    plot(X_train(indices_red,1), X_train(indices_red,2),
        'x', 'color', 'red')
    hold on;
    plot(X_train(indices_blue,1), X_train(indices_blue,2)
        , 'o', 'color', 'blue')
    title('Training data')

    indices_blue = find(y_test==1);
    indices_red = find(y_test==0);
    figure(2)
    plot(X_test(indices_red, 1), X_test(indices_red,2), '
        x', 'color', 'red')
    hold on;
    plot(X_test(indices_blue,1), X_test(indices_blue,2),
        'o', 'color', 'blue')
    title('Test data')
end
disp("Loaded data")
end
```

5.3 Importera MNIST-data

```
function [X_train, y_train] = mnist_data(all_examples,
    number_examples)

% load data mnist
if all_examples == true
     data = csvread('mnist_train.csv');
     X_train = data(1:end, 2:end);
     y_train = data(1:end, 1)';
     disp("Loaded data")

else
     data = csvread('mnist_train.csv');
     X_train = data(1:number_examples, 2:end);
     y_train = data(1:number_examples, 1)';
     disp("Loaded data")
end
end
```

5.4 Importera ISIC-data

```
% This function reads all images in folder Data/resized_images and
% their corresponding labels. Stores every image and label into a matrix
% and if save_as_csv is set to True then will write this matrix to csv file
% cancer_data.csv

function data = import_ISIC(save_as_csv)
    myDir_images = "Data/new_resized/";
    myDir_descriptions = "Data/Descriptions/";

    myFiles = dir(fullfile(myDir_images, "*.jpeg"));
    files_vector = {myFiles.name};
    data = [];
    count = 1;
    not_loaded = 0;

    for file = files_vector
        disp('Currently loaded: ' + string(count) + ' images');

        pathImage = strcat(myDir_images, string(file));
        desription_file = strrep(string(file), ".jpeg", "");
        pathDescription = strcat(myDir_descriptions, desription_file);
        data_one_image = ISIC_read_single_image(pathImage, pathDescription);

        if length(data_one_image) == 0
            disp('Tried to load ' + string(file));
            not_loaded = not_loaded + 1
            disp('Unsucessfully loaded: ' + string(not_loaded));
        else
            data(:,count) = data_one_image;
            count = count+1;
        end

    end

    if save_as_csv % if True then write else skip
        disp("Writing all to csv file ...")
```

```
37            writematrix(data,'cancer_data.csv')
38            disp("Successfully wrote to csv file")
39        end
40    end
```

5.5 Läs in ISIC-bild

```
function data = ISIC_read_single_image(pathImage,
    pathDescription)

% Read image
image = imread(pathImage); %..\Data\Images\" +

% Unroll image
unrolledImage = image(:);

% Get image information
imageinfo = readcell(pathDescription);
%..\Data\Descriptions\" +

% Find row that contains information whether benign or
    malignant
index1 = find(strcmp(imageinfo, "benign_malignant:"), 1);
index2 = find(strcmp(imageinfo, "benign_malignant"), 1);

if length(index1) == 0 & length(index2) ~= 0
    index = index2;
elseif length(index1) ~= 0 & length(index2) == 0
    index = index1;
else
    index = [];
end

if index
    % Extract whether benign or malignant, remove comma
    classification = string({imageinfo(index,2)});
    classification = strrep(classification,",","");

    %target = 0 if benign, 1 if malignant
    if classification == "malignant"
        target=1;
    else
        target=0;
    end

    data=[target; unrolledImage];
else
%     disp("Benign/Malignant information was not found
    for this image");
    data = [];
end
```

6 Optimerare till gradient descent

6.1 Vanilla

```
function [parameters] = gradient_descent_vanilla(
    gradients, parameters, learning_rate)
global total_layers;

for l = 1:total_layers
    W_l = parameters(strcat('W',num2str(l)));
    b_l = parameters(strcat('b',num2str(l)));

    dW_l = gradients(strcat('dW',num2str(l)));
    db_l = gradients(strcat('db',num2str(l)));

    W_l = W_l - learning_rate .* dW_l;
    b_l = b_l - learning_rate .* db_l;

    parameters(strcat('W', num2str(l))) = W_l;
    parameters(strcat('b', num2str(l))) = b_l;
end
end
```

6.2 Momentum

```
function [parameters] = gradient_descent_momentum(
    gradients, parameters, learning_rate, beta)
global total_layers

for l = 1:total_layers
    W_l = parameters(strcat('W',num2str(l)));
    b_l = parameters(strcat('b',num2str(l)));

    dW_l = gradients(strcat('dW',num2str(l)));
    db_l = gradients(strcat('db',num2str(l)));

    v_dwl = parameters(strcat('v_dW', num2str(l)));
    v_dbl = parameters(strcat('v_db', num2str(l)));

    v_dwl = beta.*v_dwl + (1-beta).*dW_l;
    v_dbl = beta.*v_dbl + (1-beta).*db_l;

    W_l = W_l - learning_rate .* v_dwl;
    b_l = b_l - learning_rate .* v_dbl;

    parameters(strcat('v_dW', num2str(l))) = v_dwl;
    parameters(strcat('v_db', num2str(l))) = v_dbl;
    parameters(strcat('W', num2str(l))) = W_l;
    parameters(strcat('b', num2str(l))) = b_l;
end
end
```

6.3 Adam

```
function [parameters] = gradient_descent_adam(gradients,
    parameters, learning_rate, beta1, beta2, t)
epsilon = 1e-8; % in case of numerical errors
global total_layers;

for l = 1:total_layers
    W_l = parameters(strcat('W',num2str(l)));
    b_l = parameters(strcat('b',num2str(l)));

    dW_l = gradients(strcat('dW',num2str(l)));
    db_l = gradients(strcat('db',num2str(l)));

    v_dwl = parameters(strcat('v_dW', num2str(l)));
    v_dbl = parameters(strcat('v_db', num2str(l)));

    s_dwl = parameters(strcat('s_dW', num2str(l)));
    s_dbl = parameters(strcat('s_db', num2str(l)));

    % below is momentum equations
    v_dwl = beta1.*v_dwl + (1-beta1).*dW_l;
    v_dbl = beta1.*v_dbl + (1-beta1).*db_l;

    % below is RMSprop equations
    s_dwl = beta2.*s_dwl + (1-beta2).*dW_l.^2;
    s_dbl = beta2.*s_dbl + (1-beta2).*db_l.^2;

    % used for debugging
%     disp('Norm vdwl: ' + string(norm(v_dwl)))
%     disp('Norm vdbl: ' + string(norm(v_dbl)))
%     disp('Norm sdwl: ' + string(norm(s_dwl)))
%     disp('Norm sdbl: ' + string(norm(s_dbl)))

    % Adam calculations
    v_dwl_corrected = v_dwl ./ (1 - (beta1.^t));
    v_dbl_corrected = v_dbl ./ (1 - (beta1.^t));

    s_dwl_corrected = s_dwl ./ (1- (beta2.^t));
    s_dbl_corrected = s_dbl ./ (1- (beta2.^t));

    W_l = W_l - learning_rate .* (v_dwl_corrected ./ (
        sqrt(s_dwl_corrected) + epsilon));
    b_l = b_l - learning_rate .* (v_dbl_corrected ./ (
        sqrt(s_dbl_corrected) + epsilon));

```

```
42      parameters(strcat('v_dW', num2str(l))) = v_dwl;
43      parameters(strcat('v_db', num2str(l))) = v_dbl;
44      parameters(strcat('s_dW', num2str(l))) = s_dwl;
45      parameters(strcat('s_db', num2str(l))) = s_dbl;
46      parameters(strcat('W', num2str(l))) = W_l;
47      parameters(strcat('b', num2str(l))) = b_l;
48  end
49  end
```

7 Övrigt

7.1 Kolla noggrannhet

```
function [accuracy,predictions] = check_accuracy(X, y,
    parameters, lambda, activation)
m = size(X,1);
[y_hat, ~, ~] = forward_propogation(X, y, parameters, m,
    lambda, activation);
[~, predictions] = max(y_hat,[], 2);

% MatLab is 1 indexed and y labels start from 0 so we
    have to subtract 1
predictions = predictions - 1;
accuracy = sum(y==predictions')/m;
end
```

7.2 Kolla gradienter

```
function [dtheta_approx, dtheta] = gradient_checking(parameters, gradients, X_train, y_train, m_train, lambda, activation)
% This function is ugly coded: it was done quickly to check
% if the backpropagation was implemented correctly. Only used once.
% Should probably remove this function, but kept just incase.

h=1e-7;
global total_layers

for param_string=parameters.keys()
    param_string=string(param_string);
    dtheta_approx = [];
    dparam = gradients('d'+param_string);
    [r,c] = size(dparam);
    dtheta = reshape(dparam, [1,r*c]);
    for i = 1:r*c
        param = parameters(param_string);
        original_param = param;
        [r,c]=size(param);
        theta = reshape(param, [r*c,1]);
        int_var = theta;
        theta(i) = theta(i) + h;
        param = reshape(theta, [r,c]);
        parameters(param_string) = param;

        [~, loss1, ~] = forward_propogation(X_train, y_train, parameters, m_train, lambda, activation);

        int_var(i) = int_var(i) - h;
        param = reshape(int_var, [r,c]);
        parameters(param_string) = param;
        [~, loss2, ~] = forward_propogation(X_train, y_train, parameters, m_train, lambda, activation);

        dtheta_approx_i = (loss1-loss2)./(2*h);
        dtheta_approx(end+1)=dtheta_approx_i;
        parameters(param_string) = original_param;
    end
```

```
36      value_gradcheck = norm(dtheta_approx - dtheta, 2)./(
            norm(dtheta_approx)+norm(dtheta));
37      disp("Gradient check: " + string(value_gradcheck))
38  end
39
40  disp("done")
41  % norm(dtheta_approx(1:10) - dtheta(1:10))
42  norm(dtheta_approx - dtheta, 2);
43  % dtheta_approx(1:10)
44  end
```

7.3 Plotta beslutsgränsen

```
function [] = decision_boundary(parameters, lambda,
    activation, y, X)
min_x1 = floor(min(X(:,1)));
max_x1 = ceil(max(X(:,1)));
min_x2 = floor(min(X(:,2)));
max_x2 = max(X(:,2)) + 0.2;

[xx,yy]= meshgrid(min_x1:0.005:max_x1, min_x2:0.005:
    max_x2);
points = [xx(:), yy(:)];

gen_y = zeros(1, length(points));
[~, predictions] = check_accuracy(points, gen_y,
    parameters, lambda, activation);
predictions = reshape(predictions, size(xx));

figure(5)
contourf(xx, yy, predictions)

purple = [1 0.7 1];
blue = [0.7 0.7 1];
red = [1 0.7 0.7];
yellow = [1 1 0.7];
purple2 = [1 0.3 1];
blue2 = [0.3 0.3 1];
red2 = [1 0.3 0.3];
yellow2 = [1 0.8 0.2];

map = [purple; blue; red; yellow];
%map = [0.7 0.3 0.3; 0.3 0.3 0.7];
colormap(map)

indices_purple = find(y==0);
indices_blue = find(y==1);
indices_red = find(y==2);
indices_yellow = find(y==3);
hold on;
color_reduction = 0.9;
plot(X(indices_purple,1), X(indices_purple, 2), 'x', '
    color', purple2*color_reduction, 'MarkerSize', 15)
plot(X(indices_blue,1), X(indices_blue, 2), 'o', 'color',
     blue2*color_reduction, 'MarkerSize', 15)
plot(X(indices_red,1), X(indices_red, 2), '.', 'color',
    red2*color_reduction, 'MarkerSize', 20)
```

```
39 plot(X(indices_yellow,1), X(indices_yellow, 2), '+', '
       color', yellow2*color_reduction, 'MarkerSize', 15)
40
41 % hold on;
42
43  %indices_blue = find(y==1);
44  %indices_red = find(y==0);
45
46  %plot(X(indices_red,1), X(indices_red,2), 'x', 'color',
       'red')
47  %plot(X(indices_blue,1), X(indices_blue,2), 'o', 'color
       ', 'blue')
48 end
```

7.4 Learning rate decay

```
function [learning_rate] = exponential_decay(
    learning_rate, epoch_num)
learning_rate = 0.99999^(epoch_num) * learning_rate;
%decay_rate = 1;
%learning_rate = 1/(1+decay_rate*epoch_num) *
    learning_rate;

end
```

7.5 Normalizering av data

```
% Runs normalization on X, computes
% X <-- (X - mu) / sigma

function [X_train, X_test] = normalization(X_train,
    X_test)
mu = mean(X_train,1);
size(mu)
X_train = X_train - mu;
X_test = X_test - mu;
sigma_squared = sum(X_train.^2) ./ size(X_train,1);
size(sigma_squared)
X_train = X_train ./ (sqrt(sigma_squared) + 1e-6);
X_test = X_test ./ (sqrt(sigma_squared) + 1e-6);
disp("Data normalized")
end
```

7.6 Byt storlek på en bild

```
% This small script is to go thrpugh each image in Data/
    Images folder,
% and for each output a resized image into folder Data/
    rezied_images
% with size. This script should just be run once, after
    setting the dataset
% in the correct folders.

clear all; close all; clc;

size = [64,64];
path = "../Data/Images/";
newpath = "../Data/new_resized/";
myFiles = dir(fullfile(path, "*.jpeg")); % Finds all
    jpegs in folder
files_in_folder = {myFiles.name}; % Make all jpegs file
    into a vector

% Found a way to utilize parallel computing toolbox to
    make resizing
% the 24k images go a little bit faster. Using parfor
    instead of for.
% Takes about ~20 minutes even using parallel computing
    toolbox.

parfor (idx = 1:length(myFiles))
    file = files_in_folder(idx);
    idx
    image = imread(path + string(file)); % Read the jpeg
        file into tensor
    resizedImage = imresize(image,size); % Resize image
    imwrite(resizedImage, newpath + string(file)) %
        Output resized image
end

% If you don't have parallel computing toolbox, this
    works as well.
% Just a bit slower.

% total_resized = 0;
% for file = files_in_folder
%     image = imread(path + string(file)); % Read the
    jpeg file into tensor
%     resizedImage = imresize(image,size); % Resize image
```

```
33 %     imwrite(resizedImage, newpath + string(file)) %
       Output resized image
34 %     total_resized = total_resized + 1;
35 %     disp('Total resized: ' + string(total_resized))
36 % end
```

7.7 Spara parametrar

```
% Unfinished function to save parameters during training
    etc.

% function saveParameters(parameters, epoch)
% parameters_epoch
%
% save('..\saved_paramateres','parameters_epoch');
% end
```

7.8 Gör vektorer i logspace

```
1 function points = stlogspace(min, max, N)
2 points =sort(10.^((log10(max)-log10(min)).*rand([1,N])+
      log10(min)));
3 end
```

D Pytorch

Här visas Pytorch-koden som använts i projektet. All kod finns även tillgänglig för nedladdning på Github [30].

Inehåll

1 CIFAR10

1.1 Main

```
'''
This code is for training either modified VGG16
or a fully connected neural network on the CIFAR10
dataset. Included help functions check accuracy,
load model, save model, etc. Depending on what
regularization technique you want to use, set
dropout rate and weight_decay for l2 regularization

'''

import torchvision.models as models
import torch.nn as nn
import torch
import torchvision
import torchvision.transforms as transforms
import sys
from torch.utils.data import DataLoader
from simple_fullynet import fullyNet

# Train CIFAR10 with a CNN or Fully Connected network
train_CNN = True
train_FC = False
assert (train_CNN or train_FC) == 1 # must train on
    either FC or CNN

class CNN_CIFAR10(object):
    def __init__(self):
        self.learning_rate = 0.001
        self.drop_rate = 0.0
        self.weight_decay = 0.0
        self.num_epochs = 100000
        self.batch_size = 64
        self.num_workers = 0
        self.device = 'cuda' if torch.cuda.is_available()
            else 'cpu'
        self.dtype =  torch.float32
        self.save_model = False
        self.shuffle = True
        self.pin_memory = True
        self.checkpoint_file = 'checkpoint/CIFAR10_VGG16'

    def setup_model(self):
```

```
41          if train_CNN:
42              # Initialize modified VGG16
43              model = models.vgg16(pretrained=True)
44              model.features[4] = nn.Identity()
45              model.features[16] = nn.Identity()
46              model.features[23] = nn.Identity()
47              model.classifier[6] = nn.Linear(in_features
                    =4096, out_features=10, bias=True)
48              model.classifier[2] = nn.Dropout(p=0.0)
49              model.classifier[5] = nn.Dropout(p=0.0)
50              model.cuda()
51
52          elif train_FC:
53              # Initialize model
54              model = fullyNet(input_size=32*32*3,
                    drop_rate=0.0, init_weights=True)
55              model.cuda()
56
57          return model
58
59      def load_data(self):
60          self.transform_train, self.transform_test = self.
                transformations()
61          train_data, validation_data = torch.utils.data.
                random_split(torchvision.datasets.CIFAR10('./
                CIFAR10', train=True, transform=self.
                transform_train), [40000, 10000])
62          test_data = torchvision.datasets.CIFAR10('./
                CIFAR10', train=False, transform=self.
                transform_train)
63
64          train_loader = DataLoader(dataset = train_data,
                batch_size = self.batch_size, num_workers =
                self.num_workers)
65          validation_loader = DataLoader(dataset =
                validation_data, batch_size = self.batch_size,
                 num_workers =  self.num_workers)
66          test_loader =  DataLoader(dataset = test_data,
                batch_size = self.batch_size, num_workers =
                self.num_workers)
67
68          return train_loader, validation_loader,
                test_loader
69
70      # Mean, std values previously computed from dataset
71      def transformations(self):
```

```
72          transform_train = transforms.Compose([
73                  transforms.ToTensor(),
74                  transforms.Normalize(mean=(0.4914,
                        0.4822, 0.4465), std=(0.247, 0.243,
                        0.261)),
75              ])
76
77          transform_test = transforms.Compose([
78                  transforms.ToTensor(),
79                  transforms.Normalize(mean=(0.4914,
                        0.4822, 0.4465), std=(0.247, 0.243,
                        0.261)),
80              ])
81
82          return transform_train, transform_test
83
84      def check_accuracy(self, loader, model):
85          num_correct = 0
86          num_samples = 0
87          model.eval()  # set model to evaluation mode
88
89          with torch.no_grad():
90              for x, y in loader:
91                  x = x.to(device=self.device, dtype=self.
                        dtype)  # move to device, e.g. GPU
92                  y = y.to(device=self.device, dtype=torch.
                        long)
93
94                  if train_FC:
95                      x = x.reshape(x.shape[0], -1)
96
97                  scores = model(x)
98                  _, preds = scores.max(1)
99                  num_correct += (preds == y).sum()
100                 num_samples += preds.size(0)
101             acc = (float(num_correct) / num_samples) *
                    100.0
102
103             print('Got %d / %d correct (%.2f)' % (
                    num_correct, num_samples, acc))
104
105             model.train() # set model back to training
                    mode
106             return acc
107
108     def save_checkpoint(self, filename, model, optimizer,
```

```
            epoch):
109         save_state = {
110             'state_dict' : model.state_dict(),
111             'epoch' : epoch + 1,
112             'optimizer' : optimizer.state_dict(),
113                     }
114         print()
115         print('=> Saving current parameters')
116         torch.save(save_state, filename)
117
118     def load_model(self, model, optimizer,
            checkpoint_file):
119         checkpoint = torch.load(checkpoint_file)
120         model.load_state_dict(checkpoint['state_dict'])
121         optimizer.load_state_dict(checkpoint['optimizer'
                ])
122
123         #Update lr rate and weight decay when loaded
                model
124         for param_group in optimizer.param_groups:
125             param_group['lr'] = self.learning_rate
126             param_group['weight_decay'] = self.
                    weight_decay
127
128         print("=> loaded checkpoint")
129
130     def main(self):
131         model = self.setup_model()
132         criterion = nn.CrossEntropyLoss()
133         optimizer = torch.optim.SGD(model.parameters(),
                lr=self.learning_rate, weight_decay=self.
                weight_decay)
134         train_loader, validation_loader, test_loader =
                self.load_data()
135
136         # Uncomment if load model
137         #self.load_model(model, optimizer, self.
                checkpoint_file)
138
139         for epoch in range(self.num_epochs):
140             # Initialize for measuring running training
                    accuracy
141             num_correct, total_checked = 0, 0
142             losses = []
143
144             for batch_idx, (data, target) in enumerate(
```

```
                train_loader):
145             data = data.to(device=self.device, dtype=
                    self.dtype)
146             target = target.to(device=self.device,
                    dtype=torch.long)
147
148             # If train FullyConnected reshape to make
                     shapes match
149             if train_FC:
150                 data = data.reshape(data.shape[0],
                        -1)
151
152             #forward prop
153             scores = model(data)
154             loss = criterion(scores, target)
155             losses.append(loss.item())
156
157             #backward pass
158             optimizer.zero_grad() # Zero gradients
                    from prev. batch
159             loss.backward() # Backpropogation
160             optimizer.step() # GD step
161
162             # For running training accuracy accuracy,
                     NOTE:
163             # Running training accuracy is not
                    accurate (and especially not)
164             # after a single epoch, but saves on
                    compute
165             _, preds = scores.max(1)
166             num_correct += (preds == target).sum()
167             total_checked += preds.size(0)
168
169
170         if self.save_model:
171             self.save_checkpoint(self.checkpoint_file
                    , model, optimizer, epoch)
172
173         # Print metrics after 1 training epoch
174         print(f'Mean loss this epoch: {sum(losses)/
                len(losses):.4f}')
175         print('VALIDATION:')
176         self.check_accuracy(validation_loader, model)
177         print(f'Accuracy Training: {float(num_correct
                )/float(total_checked):.4f}')
178         print('\n')
```

```
179
180 training = CNN_CIFAR10()
181 training.main()
```

1.2 Simpelt fully connected nätverk

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class fullyNet(nn.Module):
    def __init__(self, input_size, drop_rate,
        init_weights = True):
        super().__init__()
        p = drop_rate
        self.fc1 = nn.Linear(input_size, 5000)
        self.drop1 = nn.Dropout2d(p)
        self.fc2 = nn.Linear(5000 , 1000)
        self.drop2 = nn.Dropout2d(p)
        self.fc3 = nn.Linear(1000, 500)
        self.drop3 = nn.Dropout2d(p)
        self.fc4 = nn.Linear(500 , 75)
        self.drop4 = nn.Dropout2d(p)
        self.fc5 = nn.Linear(75, 10)


        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        z1 = self.fc1(x)
        a1 = F.relu(z1)
        a1 = self.drop1(a1)

        z2 = self.fc2(a1)
        a2 = F.relu(z2)
        a2 = self.drop2(a2)

        z3 = self.fc3(a2)
        a3 = F.relu(z3)
        a3 = self.drop3(a3)

        z4 = self.fc4(a3)
        a4 = F.relu(z4)
        a4 = self.drop4(a4)

        z5 = self.fc5(a4)
        return z5


```

```
44      def _initialize_weights(self):
45          for m in self.modules():
46              if isinstance(m, nn.Linear):
47                  nn.init.kaiming_normal_(m.weight, mode='
                        fan_in', nonlinearity='relu')
48                  nn.init.constant_(m.bias, 0)
49
50  def test_fullyNet():
51      input_size = 100
52      net = fullyNet(input_size, drop_rate=0.0)
53      x = torch.randn(64,input_size)
54      y = net(x)
55      print(y.size())
56
57  if __name__ == '__main__':
58      test_fullyNet()
```

2 MNIST Fashion

2.1 Main

```
'''
This code is for training either modified VGG16
or a fully connected neural network on the FashionMNIST
dataset. Included help functions check accuracy,
load model, save model, etc. Depending on what
regularization technique you want to use, set
dropout rate and weight_decay for l2 regularization

'''

import torchvision.models as models
import torch.nn as nn
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from simple_fullynet import fullyNet

# Train CIFAR10 with a CNN or Fully Connected network
train_CNN = False
train_FC = True
assert (train_CNN or train_FC) == 1 and (train_CNN and
    train_FC) == 0 # must train on either FC or CNN

class CNN_FashionMNIST(object):
    def __init__(self):
        self.learning_rate = 0.001
        self.dropout = 0.0
        self.weight_decay = 0.0
        self.num_epochs = 100000
        self.batch_size = 64
        self.num_workers = 0
        self.device = 'cuda' if torch.cuda.is_available()
            else 'cpu'
        self.dtype =  torch.float32
        self.save_model = False
        self.shuffle = True
        self.pin_memory = True
        self.checkpoint_file = 'checkpoint/
           MNISTFashion_VGG16'

    def setup_model(self, drop_rate):
```

```
40          if train_CNN:
41              # Initialize modified VGG16
42              model = models.vgg16(pretrained=True)
43              model.features[0] = nn.Conv2d(in_channels=1,
                    out_channels=64,kernel_size=3,stride=1,
                    padding=1)
44              model.features[4] = nn.Identity()
45              model.features[16] = nn.Identity()
46              model.features[23] = nn.Identity()
47              model.classifier[6] = nn.Linear(in_features
                    =4096, out_features=10, bias=True)
48              model.classifier[2] = nn.Dropout(p=drop_rate)
49              model.classifier[5] = nn.Dropout(p=drop_rate)
50
51          elif train_FC:
52              # Initialize Fully Connected
53              model = fullyNet(input_size=28*28*1,
                    drop_rate=drop_rate, init_weights=True)
54
55          return model
56
57      def load_data(self):
58          self.transform_train, self.transform_test = self.
                transformations()
59          train_data, validation_data = torch.utils.data.
                random_split(torchvision.datasets.FashionMNIST
                ('./fashionMNIST', train=True, transform=self.
                transform_train), [50000, 10000])
60          test_data = torchvision.datasets.FashionMNIST('./
                fashionMNIST', train=False, transform=self.
                transform_train)
61
62          train_loader = DataLoader(dataset = train_data,
                batch_size = self.batch_size, num_workers =
                self.num_workers)
63          validation_loader = DataLoader(dataset =
                validation_data, batch_size = self.batch_size,
                 num_workers =  self.num_workers)
64          test_loader =  DataLoader(dataset = test_data,
                batch_size = self.batch_size, num_workers =
                self.num_workers)
65
66          return train_loader, validation_loader,
                test_loader
67
68      # Mean, std values previously computed from dataset
```

```
69      def transformations(self):
70          transform_train = transforms.Compose([
71                  transforms.ToTensor(),
72                  transforms.Normalize((0.1307,),
                        (0.3081,)),
73              ])
74
75          transform_test = transforms.Compose([
76                  transforms.ToTensor(),
77                  transforms.Normalize((0.1307,),
                        (0.3081,)),
78              ])
79
80          return transform_train, transform_test
81
82      def check_accuracy(self, loader, model):
83          num_correct = 0
84          num_samples = 0
85          model.eval()  # set model to evaluation mode
86
87          with torch.no_grad():
88              for x, y in loader:
89                  x = x.to(device=self.device, dtype=self.
                        dtype)  # move to device, e.g. GPU
90                  y = y.to(device=self.device, dtype=torch.
                        long)
91
92                  if train_FC:
93                      x = x.reshape(x.shape[0], -1)
94
95                  scores = model(x)
96                  _, preds = scores.max(1)
97                  num_correct += (preds == y).sum()
98                  num_samples += preds.size(0)
99              acc = (float(num_correct) / num_samples) *
                    100.0
100
101             print('Got %d / %d correct (%.2f)' % (
                    num_correct, num_samples, acc))
102
103         model.train() # set model to training mode again
104         return acc
105
106     def save_checkpoint(self, filename, model, optimizer,
            epoch):
107         save_state = {
```

```
108             'state_dict' : model.state_dict(),
109             'epoch' : epoch + 1,
110             'optimizer' : optimizer.state_dict(),
111                     }
112         print()
113         print('Saving current parameters')
114         print('
            ------------------------------------------------------------
            ')
115
116         torch.save(save_state, filename)
117
118     def load_model(self, model, optimizer,
            checkpoint_file):
119         checkpoint = torch.load(checkpoint_file)
120         model.load_state_dict(checkpoint['state_dict'])
121         optimizer.load_state_dict(checkpoint['optimizer'
                ])
122
123         #Update lr rate and weight decay when loaded
                model
124         for param_group in optimizer.param_groups:
125             param_group['lr'] = self.learning_rate
126             param_group['weight_decay'] = self.
                    weight_decay
127
128         print("=> loaded checkpoint")
129
130     def main(self):
131         model = self.setup_model(self.dropout).to(self.
                device)
132         criterion = nn.CrossEntropyLoss()
133         optimizer = torch.optim.SGD(model.parameters(),
                lr=self.learning_rate, weight_decay=self.
                weight_decay)
134         train_loader, validation_loader, test_loader =
                self.load_data()
135
136         ## Uncomment if load model
137         #self.load_model(model, optimizer, self.
                checkpoint_file)
138
139         for epoch in range(self.num_epochs):
140             num_correct, total_checked = 0, 0
141             losses = []
142
```

```
143             for batch_idx, (data,target) in enumerate(
                    train_loader):
144                 data = data.to(device=self.device, dtype=
                        self.dtype)
145                 target = target.to(device=self.device,
                        dtype=torch.long)
146 
147                 if train_FC:
148                     data = data.reshape(data.shape[0],
                            -1)
149 
150                 #forward prop
151                 scores = model(data)
152                 loss = criterion(scores, target)
153                 losses.append(loss.item()) # add to keep
                        track of loss
154 
155                 #backward pass
156                 optimizer.zero_grad() # Zero gradients
                        from prev. batch
157                 loss.backward() # Backpropogation
158                 optimizer.step() # GD step
159 
160                 # For running training accuracy accuracy,
                        NOTE:
161                 # Running training accuracy is not
                        accurate (and especially not)
162                 # after a single epoch, but saves on
                        compute
163                 _, preds = scores.max(1)
164                 num_correct += (preds == target).sum()
165                 total_checked += preds.size(0)
166 
167             if self.save_model:
168                 self.save_checkpoint(self.checkpoint_file
                        , model, optimizer, epoch)
169 
170             # Print metrics after 1 training epoch
171             print(f'Mean loss this epoch: {sum(losses)/
                    len(losses):.4f}')
172             print('VALIDATION:')
173             self.check_accuracy(validation_loader, model)
174             print(f'Accuracy Training: {float(num_correct
                    )/float(total_checked):.4f}')
175             print('\n')
176 
```

```
177
178 training = CNN_FashionMNIST()
179 training.main()
```

2.2 Simpelt fully connected nätverk

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class fullyNet(nn.Module):
    def __init__(self, input_size, drop_rate,
        init_weights = True):
        super(fullyNet, self).__init__()
        p = drop_rate
        self.fc1 = nn.Linear(input_size, 500)
        self.drop1 = nn.Dropout2d(p)
        self.fc2 = nn.Linear(500, 100)
        self.drop2 = nn.Dropout2d(p)
        self.fc3 = nn.Linear(100, 50)
        self.drop3 = nn.Dropout2d(p)
        self.fc4 = nn.Linear(50, 25)
        self.drop4 = nn.Dropout2d(p)
        self.fc5 = nn.Linear(25, 10)


        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        z1 = self.fc1(x)
        a1 = F.relu(z1)
        a1 = self.drop1(a1)

        z2 = self.fc2(a1)
        a2 = F.relu(z2)
        a2 = self.drop2(a2)

        z3 = self.fc3(a2)
        a3 = F.relu(z3)
        a3 = self.drop3(a3)

        z4 = self.fc4(a3)
        a4 = F.relu(z4)
        a4 = self.drop4(a4)

        z5 = self.fc5(a4)
        return z5


```

```
44      def _initialize_weights(self):
45          for m in self.modules():
46              if isinstance(m, nn.Linear):
47                  nn.init.kaiming_normal_(m.weight, mode='
                        fan_in', nonlinearity='relu')
48                  nn.init.constant_(m.bias, 0)
49
50  def test_fullyNet():
51      input_size = 100
52      net = fullyNet(input_size, drop_rate=0.0)
53      x = torch.randn(64,input_size)
54      y = net(x)
55      print(y.size())
56
57  if __name__ == '__main__':
58      test_fullyNet()
```

3 ISIC (cancerbilder)

3.1 Main

```
'''
This code is for training pretrained ResNext-101 network
modified to train on ISIC-dataset. Included help
    functions
check accuracy, load model, save model, etc. Depending on
what regularization technique you want to use, set
    dropout
rate and weight_decay for l2 regularization.

'''

import torch
import torch.optim as optim
import torch.nn as nn
import sys
import torch.nn.functional as F
from torch.utils.data import DataLoader
from ISIC_dataset import ISIC_dataset
from utils.import_utils import *
from model import FullyConnected
from simple_fullynet import fullyNet

# Train CIFAR10 with a CNN or Fully Connected network
train_CNN = True
train_FC = False

# must train on either FC or CNN and cant train both at
    same time
assert (train_CNN or train_FC) == 1 and (train_CNN and
    train_FC) == 0

class ISIC_CNN(object):
    def __init__(self):
        self.root_dir = 'Data/'
        self.train_csv_file = 'ISIC_train.csv'
        self.test_csv_file = 'ISIC_test.csv'
        self.validation_csv_file = 'ISIC_validation.csv'
        self.checkpoint_file = 'checkpoint/ISIC_CNN.pth.
            tar'

        self.device = 'cuda' if torch.cuda.is_available()
            else 'cpu'
```

```
37          self.learning_rate = 1e-3
38          self.weight_decay = 0.0
39          self.dropout = 0.0
40          self.num_epochs = 100
41          self.batch_size = 2
42          self.num_workers = 0
43
44          self.dtype =  torch.float32
45          self.shuffle = True
46          self.pin_memory = True
47          self.save_model = False
48
49          self.transform_train, self.transform_val =
                prepare_transformations()
50
51          self.train_loader = DataLoader(
52              ISIC_dataset(self.root_dir, self.
                    train_csv_file, self.transform_train),
53              batch_size = self.batch_size,
54              shuffle = self.shuffle,
55              num_workers = self.num_workers,
56              pin_memory = self.pin_memory)
57
58          self.test_loader = DataLoader(
59              ISIC_dataset(self.root_dir, self.
                    test_csv_file, self.transform_val),
60              batch_size = self.batch_size,
61              shuffle = self.shuffle,
62              num_workers = self.num_workers,
63              pin_memory = self.pin_memory)
64
65          self.validation_loader = DataLoader(
66              ISIC_dataset(self.root_dir, self.
                    validation_csv_file, self.transform_val),
67              batch_size = self.batch_size,
68              shuffle = self.shuffle,
69              num_workers = self.num_workers,
70              pin_memory = self.pin_memory)
71
72      def setup_model(self, drop_rate):
73          if train_CNN:
74              # Loading pretrained ResNext101 model
75              model = torch.hub.load('facebookresearch/WSL-
                    Images', 'resnext101_32x16d_wsl')
76
77              # Setting last fully connected layers to our
```

```
                defined FullyConnected
78              model.fc = FullyConnected(drop_rate=drop_rate
                    )
79
80          elif train_FC:
81              # Initialize model
82              model = fullyNet(input_size=224*224*3,
                    drop_rate=drop_rate, num_classes=2)
83
84          return model
85
86      def check_accuracy(self, loader, model):
87          num_correct = 0
88          num_samples = 0
89          model.eval()  # set model to evaluation mode
90
91          with torch.no_grad():
92              for x, y in loader:
93                  x = x.to(device=self.device, dtype=self.
                        dtype)  # move to device, e.g. GPU
94                  y = y.to(device=self.device, dtype=torch.
                        long)
95
96                  if train_FC:
97                      x = x.reshape(x.shape[0], -1)
98
99                  scores = model(x)
100                 _, preds = scores.max(1)
101                 num_correct += (preds == y).sum()
102                 num_samples += preds.size(0)
103             acc = (float(num_correct) / num_samples) *
                    100.0
104
105         model.train()
106         print('Got %d / %d correct (%.2f)' % (num_correct
                , num_samples, acc))
107         return acc
108
109     def main(self):
110         model = self.setup_model(self.dropout).to(self.
                device)
111         weight = torch.FloatTensor([1, 10]).cuda()
112         criterion = nn.CrossEntropyLoss(weight=weight)
113         optimizer = optim.SGD(model.parameters(), lr=self
                .learning_rate, weight_decay=self.weight_decay
                )
```

```
114
115         # Make sure loading from correct checkpoint
116         #load_model(model, optimizer, self.
                checkpoint_file)
117
118         # If loaded model then we need to update learning
                rate and weight decay parameters
119         for param_group in optimizer.param_groups:
120             param_group['lr'] = self.learning_rate
121             param_group['weight_decay'] = self.
                    weight_decay
122
123         for epoch in range(self.num_epochs):
124             num_correct, total_checked = 0, 0
125             losses = []
126
127             # Check scores on validation each epoch (
                    after at least training 1 epoch)
128             if epoch >= 1:
129                 print("Checking scores on validation...")
130                 f_score = check_precision_and_recall(self
                        .validation_loader, model)
131
132             for batch_idx, (data, targets) in enumerate(
                    self.train_loader):
133                 data = data.to(device=self.device, dtype=
                        self.dtype)
134                 targets = targets.to(device=self.device,
                        dtype=torch.long)
135
136                 if train_FC:
137                     data = data.reshape(data.shape[0],
                            -1)
138
139                 #forward prop
140                 scores = model(data)
141                 loss = criterion(scores, targets)
142                 losses.append(loss.item())
143
144                 #backward pass
145                 optimizer.zero_grad() # Zero gradients
                        from prev. batch
146                 loss.backward() # Backpropogation
147                 optimizer.step() # GD step
148
149                 # For running training accuracy accuracy,
```

```
                    NOTE:
150                 # Running training accuracy is not
                        accurate (and especially not)
151                 # after a single epoch, but saves on
                        compute
152                 _, preds = scores.max(1)
153                 num_correct += (preds == targets).sum()
154                 total_checked += preds.size(0)
155
156
157             if self.save_model:
158                 save_checkpoint(self.checkpoint_file,
                        model, optimizer, epoch)
159
160             # Print metrics after 1 training epoch
161             print(f'Mean loss this epoch: {sum(losses)/
                    len(losses):.4f}')
162             print('VALIDATION:')
163             self.check_accuracy(validation_loader, model)
164             print(f'Accuracy Training: {float(num_correct
                    )/float(total_checked):.4f}')
165             print('\n')
166
167
168 if __name__ == '__main__':
169     net = ISIC_CNN()
170     net.main()
```

3.2 Simpelt fully connected nätverk

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class fullyNet(nn.Module):
    def __init__(self, input_size, drop_rate,
        init_weights = True, num_classes=10):
        super(fullyNet, self).__init__()
        p = drop_rate
        self.fc1 = nn.Linear(input_size, 500)
        self.drop1 = nn.Dropout2d(p)
        self.fc2 = nn.Linear(500, 100)
        self.drop2 = nn.Dropout2d(p)
        self.fc3 = nn.Linear(100, 50)
        self.drop3 = nn.Dropout2d(p)
        self.fc4 = nn.Linear(50, 25)
        self.drop4 = nn.Dropout2d(p)
        self.fc5 = nn.Linear(25, num_classes)


        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        z1 = self.fc1(x)
        a1 = F.relu(z1)
        a1 = self.drop1(a1)

        z2 = self.fc2(a1)
        a2 = F.relu(z2)
        a2 = self.drop2(a2)

        z3 = self.fc3(a2)
        a3 = F.relu(z3)
        a3 = self.drop3(a3)

        z4 = self.fc4(a3)
        a4 = F.relu(z4)
        a4 = self.drop4(a4)

        z5 = self.fc5(a4)
        return z5


```

```
44      def _initialize_weights(self):
45          for m in self.modules():
46              if isinstance(m, nn.Linear):
47                  nn.init.kaiming_normal_(m.weight, mode='
                        fan_in', nonlinearity='relu')
48                  nn.init.constant_(m.bias, 0)
49
50  def test_fullyNet():
51      input_size = 100
52      net = fullyNet(input_size, drop_rate=0.0)
53      x = torch.randn(64,input_size)
54      y = net(x)
55      print(y.size())
56
57  if __name__ == '__main__':
58      test_fullyNet()
```

3.3 Modell

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class FullyConnected(nn.Module):
    def __init__(self, drop_rate):
        super().__init__()
        self.drop_rate = drop_rate

        self.fc1 = nn.Linear(2048, 1024)
        self.drop1 = nn.Dropout2d(self.drop_rate)
        self.fc2 = nn.Linear(1024 , 512)
        self.drop2 = nn.Dropout2d(self.drop_rate)
        self.fc3 = nn.Linear(512, 2)

    def forward(self, x):
        z1 = self.fc1(x)
        a1 = F.relu(z1)
        a1 = self.drop1(a1)

        z2 = self.fc2(a1)
        a2 = F.relu(z2)
        a2 = self.drop2(a2)

        z3 = self.fc3(a2)
        return z3
```

3.4 Dataset

```
from torch.utils.data import Dataset
import pandas as pd
import os
import numpy as np
import torch
from skimage import io

class ISIC_dataset(Dataset):
    def __init__(self, root_dir, csv_file, transform=None):
        self.root_dir = root_dir
        self.annotations = pd.read_csv(self.root_dir + csv_file)
        self.transform = transform

    def __len__(self):
        return len(self.annotations)

    def __getitem__(self, index):
        img_name = os.path.join(self.root_dir + '/Images_resized', self.annotations.iloc[index, 0])
        image = io.imread(img_name)
        y_label = int(self.annotations.iloc[index, 1])

        if self.transform:
            image = self.transform(image)

        return image, y_label
```

3.5 Övrigt

3.5.1 Utils

```
import torch
import torchvision.transforms as transforms

device = 'cuda' if torch.cuda.is_available() else 'cpu'
dtype =  torch.float32

def prepare_transformations():
    train_transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.RandomHorizontalFlip(),
        transforms.RandomAffine(degrees=15),
        transforms.ToTensor(),
        transforms.Normalize([0.7195, 0.5627, 0.5254],
            [0.0607, 0.0522, 0.0508])
    ])

    validation_transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.ToTensor(),
        transforms.Normalize([0.7195, 0.5627, 0.5254],
            [0.0607, 0.0522, 0.0508])
    ])

    return train_transform, validation_transform


def save_checkpoint(filename, model, optimizer, epoch):
    model.eval()
    save_state = {
        'state_dict' : model.state_dict(),
        'epoch' : epoch + 1,
        'optimizer' : optimizer.state_dict(),
                }
    print()
    print('Saving current parameters')
    print('
        ------------------------------------------------------------
        ')

    torch.save(save_state, filename)


```

```
39 def check_precision_and_recall(loader, model):
40     # If we say it's cancer: how good of a prediction is
       it <-- Precision
41     # Of all that had cancer, how many did we say have
       cancer? <-- Recall
42     true_positives = 0
43     predicted_positives = 0
44     actual_positives = 0
45     actual_negs = 0
46     true_negatives = 0
47
48     model.eval()  # set model to evaluation mode
49     with torch.no_grad():
50         for x, y in loader:
51             x = x.to(device=device, dtype=dtype)  # move
               to device, e.g. GPU
52             y = y.to(device=device, dtype=torch.long)
53             scores = model(x)
54             _, preds = scores.max(1)
55             true_positives += sum([1 if (x1 == 1) and (y1
                   == 1) else 0 for x1, y1 in zip(preds, y)
               ])
56             true_negatives += sum([1 if (x1 == 0) and (y1
                   == 0) else 0 for x1, y1 in zip(preds, y)
               ])
57             predicted_positives += sum(preds==1)
58             actual_positives += sum(y == 1)
59             actual_negs += sum(y == 0)
60
61
62         precision = (float(true_positives) / float(
               predicted_positives))
63         recall = (float(true_positives) / float(
               actual_positives))
64         specificity = (float(true_negatives) / float(
               actual_negs))
65         F_score = (2 / (1/precision + 1/recall))
66
67         print('PRECISION: Got %d / %d correct (%.2f)' % (
               true_positives, predicted_positives, precision
               *100))
68         print('RECALL: Got %d / %d correct (%.2f)' % (
               true_positives, actual_positives, recall*100))
69         print('SPECIFICITY: Got %d / %d correct (%.2f)' %
                (true_negatives, actual_negs, specificity
               *100))
```

```
70          print('F_score: ' + str(F_score)) # Harmonic Mean
71
72
73      model.train() # Set back to model.train
74      return F_score
75
76  def load_model(model, optimizer, checkpoint_file):
77      checkpoint = torch.load(checkpoint_file, map_location
            ='cuda:0')
78      model.load_state_dict(checkpoint['state_dict'])
79      optimizer.load_state_dict(checkpoint['optimizer'])
80
81      print("=> loaded checkpoint")
```

3.5.2 Importera utils

```
1 from utils.utils import prepare_transformations,
      save_checkpoint, check_accuracy,
      check_precision_and_recall, load_model
```

4 MNIST

4.1 Main

```
'''
This code is for training the VGG16 network modified
to train on MNIST. Included help functions check
accuracy, load model, save model, etc. Depending on
what regularization technique you want to use, set
dropout rate and weight_decay for l2 regularization

'''
import torchvision.models as models
import torch.nn as nn
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from simple_fullynet import fullyNet

# Train CIFAR10 with a CNN or Fully Connected network
train_CNN = False
train_FC = True

# must train on either FC or CNN and cant train both at
    same time
assert (train_CNN or train_FC) == 1 and (train_CNN and
    train_FC) == 0

class CNN_MNIST(object):
    def __init__(self):
        self.learning_rate = 0.001
        self.weight_decay = 0.0
        self.dropout = 0.0
        self.num_epochs = 100000
        self.batch_size = 64
        self.num_workers = 0
        self.device = 'cuda' if torch.cuda.is_available()
            else 'cpu'
        self.dtype =  torch.float32
        self.save_model = False
        self.shuffle = True
        self.pin_memory = True
        self.checkpoint_file = 'checkpoint/MNIST_VGG16'

    def setup_model(self, drop_rate):
```

```
40          if train_CNN:
41              # Initialize model
42              model = models.vgg16(pretrained=True)
43              model.features[0] = nn.Conv2d(in_channels=1,
                    out_channels=64, kernel_size = (3,3),
                    stride = (1,1), padding = (1,1))
44              model.features[4] = nn.Identity()
45              model.features[16] = nn.Identity()
46              model.features[23] = nn.Identity()
47              model.classifier[6] = nn.Linear(in_features
                    =4096, out_features=10, bias=True)
48              model.classifier[2] = nn.Dropout(p=drop_rate)
49              model.classifier[5] = nn.Dropout(p=drop_rate)
50
51          elif train_FC:
52              # Initialize Fully Connected
53              model = fullyNet(input_size=28*28*1,
                    drop_rate=drop_rate, init_weights=True)
54
55          return model
56
57      def load_data(self):
58          self.transform_train, self.transform_test = self.
                transformations()
59          train_data, validation_data = torch.utils.data.
                random_split(torchvision.datasets.MNIST('./
                mnist', train=True, transform=self.
                transform_train), [50000, 10000])
60          test_data = torchvision.datasets.MNIST('./mnist',
                 train=False, transform=self.transform_train)
61
62          train_loader = DataLoader(dataset = train_data,
                batch_size = self.batch_size, num_workers =
                self.num_workers)
63          validation_loader = DataLoader(dataset =
                validation_data, batch_size = self.batch_size,
                 num_workers =  self.num_workers)
64          test_loader =  DataLoader(dataset = test_data,
                batch_size = self.batch_size, num_workers =
                self.num_workers)
65
66          return train_loader, validation_loader,
                test_loader
67
68      # Mean, std values previously computed from dataset
69      def transformations(self):
```

```
70          transform_train = transforms.Compose([
71                  transforms.ToTensor(),
72                  transforms.Normalize((0.1307,),
                        (0.3081,)),
73              ])
74
75          transform_test = transforms.Compose([
76                  transforms.ToTensor(),
77                  transforms.Normalize((0.1307,),
                        (0.3081,)),
78              ])
79
80          return transform_train, transform_test
81
82      def check_accuracy(self, loader, model):
83          num_correct = 0
84          num_samples = 0
85          model.eval()  # set model to evaluation mode
86
87          with torch.no_grad():
88              for x, y in loader:
89                  x = x.to(device=self.device, dtype=self.
                        dtype)  # move to device, e.g. GPU
90                  y = y.to(device=self.device, dtype=torch.
                        long)
91
92                  if train_FC:
93                      x = x.reshape(x.shape[0], -1)
94
95                  scores = model(x)
96                  _, preds = scores.max(1)
97                  num_correct += (preds == y).sum()
98                  num_samples += preds.size(0)
99              acc = (float(num_correct) / num_samples) *
                    100.0
100
101             print('Got %d / %d correct (%.2f)' % (
                    num_correct, num_samples, acc))
102
103             model.train() # set model to training mode
                    again
104             return acc
105
106     def save_checkpoint(self, filename, model, optimizer,
            epoch):
107         save_state = {
```

```
108             'state_dict' : model.state_dict(),
109             'epoch' : epoch + 1,
110             'optimizer' : optimizer.state_dict(),
111                     }
112         print('=> Saving current parameters')
113 
114         torch.save(save_state, filename)
115 
116     def load_model(self, model, optimizer,
        checkpoint_file):
117         checkpoint = torch.load(checkpoint_file)
118         model.load_state_dict(checkpoint['state_dict'])
119         optimizer.load_state_dict(checkpoint['optimizer'
            ])
120 
121         #Update lr rate and weight decay when loaded
            model
122         for param_group in optimizer.param_groups:
123             param_group['lr'] = self.learning_rate
124             param_group['weight_decay'] = self.
                weight_decay
125 
126 
127         print("=> loaded checkpoint")
128 
129     def main(self):
130         model = self.setup_model(self.dropout).to(self.
            device)
131         criterion = nn.CrossEntropyLoss()
132         optimizer = torch.optim.SGD(model.parameters(),
            lr=self.learning_rate, weight_decay=self.
            weight_decay)
133         train_loader, validation_loader, test_loader =
            self.load_data()
134 
135         #self.load_model(model, optimizer, self.
            checkpoint_file)
136 
137         for epoch in range(self.num_epochs):
138             num_correct, total_checked = 0, 0
139             losses = []
140 
141             for batch_idx, (data,target) in enumerate(
                train_loader):
142                 data = data.to(device=self.device, dtype=
                    self.dtype)
```

```
143                     target = target.to(device=self.device,
                            dtype=torch.long)
144
145                     if train_FC:
146                         data = data.reshape(data.shape[0],
                                -1)
147
148                     #forward prop
149                     scores = model(data)
150                     loss = criterion(scores, target)
151                     losses.append(loss.item())
152
153                     #backward prop
154                     optimizer.zero_grad() # Zero gradients
                            from prev. batch
155                     loss.backward() # Backpropogation
156                     optimizer.step() # GD step
157
158                     # For running training accuracy accuracy,
                            NOTE:
159                     # Running training accuracy is not
                            accurate (and especially not)
160                     # after a single epoch, but saves on
                            compute
161                     _, preds = scores.max(1)
162                     num_correct += (preds == target).sum()
163                     total_checked += preds.size(0)
164
165                 if self.save_model:
166                     self.save_checkpoint(self.checkpoint_file
                            , model, optimizer, epoch)
167
168                 # Print metrics after 1 training epoch
169                 print(f'Mean loss this epoch: {sum(losses)/
                        len(losses):.4f}')
170                 print('VALIDATION:')
171                 self.check_accuracy(validation_loader, model)
172                 print(f'Accuracy Training: {float(num_correct
                        )/float(total_checked):.4f}')
173                 print('\n')
174
175 net = CNN_MNIST()
176 net.main()
```

4.2 Simpelt fully connected nätverk

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class fullyNet(nn.Module):
    def __init__(self, input_size, drop_rate,
        init_weights = True, num_classes=10):
        super(fullyNet, self).__init__()
        p = drop_rate
        self.fc1 = nn.Linear(input_size, 500)
        self.drop1 = nn.Dropout2d(p)
        self.fc2 = nn.Linear(500, 100)
        self.drop2 = nn.Dropout2d(p)
        self.fc3 = nn.Linear(100, 50)
        self.drop3 = nn.Dropout2d(p)
        self.fc4 = nn.Linear(50, 25)
        self.drop4 = nn.Dropout2d(p)
        self.fc5 = nn.Linear(25, num_classes)


        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        z1 = self.fc1(x)
        a1 = F.relu(z1)
        a1 = self.drop1(a1)

        z2 = self.fc2(a1)
        a2 = F.relu(z2)
        a2 = self.drop2(a2)

        z3 = self.fc3(a2)
        a3 = F.relu(z3)
        a3 = self.drop3(a3)

        z4 = self.fc4(a3)
        a4 = F.relu(z4)
        a4 = self.drop4(a4)

        z5 = self.fc5(a4)
        return z5


```

```
44      def _initialize_weights(self):
45          for m in self.modules():
46              if isinstance(m, nn.Linear):
47                  nn.init.kaiming_normal_(m.weight, mode='
                        fan_in', nonlinearity='relu')
48                  nn.init.constant_(m.bias, 0)
49
50  def test_fullyNet():
51      input_size = 100
52      net = fullyNet(input_size, drop_rate=0.0)
53      x = torch.randn(64,input_size)
54      y = net(x)
55      print(y.size())
56
57  if __name__ == '__main__':
58      test_fullyNet()
```