YRKESHÖGSKOLAN NOVIA

Igenkänning av trädslag med djupinlärning

Isak Rönn

Examensarbete för ingenjörsexamen

Utbildningsprogrammet för el- och automationsteknik

Vasa 2021

EXAMENSARBETE

Författare: Isak Rönn

Utbildning och ort: El- och automationsteknik, Vasa

Inriktningsalternativ: Automation

Handledare: Jan Berglund

Titel: Igenkänning av trädslag med djupinlärning

Datum: 31.3.2021 Sidantal: 20

Abstrakt

På senare tid har maskininlärning och djupinlärning blivit alltmer populärt och används inom flera olika industrier för att automatisera arbeten som skulle vara oerhört tidskrävande eller omöjliga för en människa att utföra.

Målet med examensarbetet var att kunna känna igen olika trädslag med en hög träffsäkerhet genom att visa bilder till en djupinlärningsmodell.

Med hjälp av ramverket Fast.ai skapades en modell i Python, som kunde känna igen trädslagen för enskilda träd i bilder. Den färdiglärda modellen hade en felprocent på 3,33 procent. Det goda resultatet berodde på bilder av bra kvalitet samt en lämplig inlärningshastighet.

På grund av att Fast.ai inte i detta skede stödde objektdetektering kunde inte målet att identifiera flera träd på samma bild uppnås. Denna funktionalitet förväntas komma före 2022. När funktionalitet finns kan den ursprungliga idén om skoginventering vidare utforskas.

Språk: svenska Nyckelord: maskininlärning, djupinlärning, ai, träd

---

OPINNÄYTETYÖ

Tekijä: Isak Rönn
Koulutus ja paikkakunta: Sähkö- ja automaatiotekniikka, Vaasa
Suuntautumisvaihtoehto: Automaatio
Ohjaaja: Jan Berglund

Nimike: Puulajien tunnistaminen syvällisen oppimisen avulla

Päivämäärä: 31.3.2021 Sivumäärä:20

Tiivistelmä

Viime aikoina koneoppiminen ja syvällinen oppiminen ovat tulleet yhä suositummiksi, ja näitä käytetään useilla eri toimialoilla automatisoimaan työtä, joka olisi äärimmäisen aikaa vievää tai mahdotonta ihmisen suorittamiseksi.

Opinnäytetyön tavoitteena oli pystyä tunnistamaan eri puulaeja erittäin tarkasti näyttämällä kuvia syvälle oppimismallille.

Fast.ai-kehyksen avulla luotiin Pythonissa malli, joka pystyi tunnistamaan yksittäisten puiden puulajit kuvista. Valmistuneen mallin virhetaso oli 3,33 prosenttia. Hyvä tulos johtui laadukkaista kuvista ja sopivasta oppimisnopeudesta.

Koska Fast.ai ei tue objektien havaitsemista tässä vaiheessa, tavoitetta tunnistaa useita puita samasta kuvasta ei voitu saavuttaa. Tämän toiminnon odotetaan saapuvan ennen vuotta 2022. Kun toiminto on käytettävissä, alkuperäistä ideaa metsäluettelosta voidaan tutkia edelleen.

Kieli: ruotsi Avainsanat: koneoppiminen, syvä oppiminen, tekoäly, puu

---

BACHELOR'S THESIS

Author: Isak Rönn
Degree Programme: Electricity and automation technology
Specialization: Automation
Supervisor(s): Jan Berglund

Title: Tree species recognition with deep learning

Date: 31.3.2021 Number of pages: 20

Abstract

In recent times, machine learning and deep learning have become increasingly popular and are used in several different industries to automate the work that would be extremely time consuming or impossible for a human to perform.

The aim of the degree project was to be able to recognize different tree species with a high degree of accuracy by showing pictures to the deep learning model.

Using the Fast.ai framework, a model was created in Python, which could recognize the tree species for individual trees in images. The completed model had an error rate of 3.33 percent. The good result was due to images of good quality and a suitable learning speed.

Since Fast.ai did not support object detection at this stage, the goal of identifying multiple trees in the same image could not be achieved. This functionality is expected to arrive before 2022. When functionality is available, the original idea of a forest inventory can be further explored.

Language: swedish Key words: machine learning, deep learning, ai, tree

---

Innehållsförteckning

1 Inledning

Detta arbete går ut på att kunna automatisera igenkänning av trädarter och beräkning av trädart genom att träna en djupinlärningsmodell. Denna modell har som mål att ha en lägre felmarginal än en människa och utföra arbetet utan mänsklig insignal. Inom skogsindustrin används manuella metoder för att räkna antal träd och värdera skogsskiften, metoderna kan vara att räkna via flygfoton som är tagna på höjder mellan 1500 och 9000 meter eller drönarbilder som ofta är tagna på 100–150 meters höjd (Skogscentralen, 2020). Ofta räknas det manuellt genom att välja provytor på skogskiftet och räkna hur många stammar som finns inom en viss area, och sedan uppskatta genom att anta att hela skogskiftet ser ut på likadant sätt.

1.1 Bakgrund

Efter att ha stött på maskininlärning tyckte jag att inventering av skog kunde vara ett intressant tillämpningsområde. Innan detta arbete hade jag aldrig haft någonting att göra med djupinlärning eller maskininlärning över huvud taget, så det första steget var att undersöka om det fanns program och kurser som gjorde det möjligt att förskaffa grunderna inom ämnet. Efter att ha lärt mig dessa grunder utvecklade jag mina färdigheter så att med hjälp av olika ramverk och kodbibliotek kunde jag koda och lära en dator att känna igen trädslag, räkna trädstammar och räkna ut höjden och bredden för att få ut volymen på träden från en bild eller video.

Jag insåg efter några månader att det inte var möjligt att uppnå dessa mål på grund av att det ramverk som jag använde inte var optimerat för att kunna känna igen flera olika objekt på samma bild, ramverket klarade inte heller av att beräkna höjden eller bredden av träden. Modellen klarade dock av att känna igen ett träd och dess trädslag med en väldigt hög träffsäkerhet. Ett annat problem var den ekonomiska biten. Det finns andra verktyg och program som kan göra arbetet, men de skulle ta mycket längre tid att lära sig eller så kostar programmet för mycket för ett examensarbete. På grund av dessa anledningar avgränsade jag arbetet till att få datorn att med hög träffsäkerhet känna igen vilket trädslag som trädet hör och undersöka vilka omständigheter som leder till felidentifiering.

2 Teorin bakom djupinlärning

I detta avsnitt kommer grunderna för hur djupinlärning fungerar att förklaras, men också matematiken bakom djupinlärning och vilka matematiska funktioner som används i bakgrunden för att modellen skall kunna känna igen skillnaden på olika trädsorter. Alternativ på olika ramverk och algoritmer kommer att presenteras samt deras styrkor och svagheter.

2.1 Grunderna av djupinlärning

Inspirationen för djupinlärning är människans hjärna. I grunden är djupinlärning en maskininlärningsteknik som lär en dator att bearbeta insignaler genom att använda sig av lager för att lära sig att förutse och att klassificera information. Informationen kan vara bilder, text eller ljud. Hur kan detta kopplas ihop med människans hjärna? I hjärnan finns det hundra miljarder neuroner och varje neuron är kopplad till ungefär hundra tusen av sina grannar. Men neuroner är oanvändbara om de är ensamma därför behövs många som jobbar tillsammans för att kunna ta flera insignaler och ge nyttiga utsignaler. Man tar insignaler från observationer och skapar ett lager, det lagret skapar i sin tur en utsignal som bearbetas och blir till nästa lager och fortsätter så tills det ger ett svar. Så neuronerna eller noderna får signaler eller ingångsvärden som går genom neuronen som i sin tur ger en utsignal. Den ursprungliga insignalens lager är hos en människa till exempel sinnen som lukt, syn, känsel. För en dator kan dessa insignaler vara en kamera, en temperaturmätare eller en våg. I stället för att skicka insignalerna till neuroner skickar vi dem till datorns motsvarighet: neurala nätverk.

2.1.1 Neurala nätverk

Det finns två olika sätt för att få ett program att göra vad man vill. Det första är att hårdkoda exakt vad programmet skall göra. Det andra sättet är att använda neurala nätverk där man ger nätverket insignaler, inputs, och vilka utsignaler, outputs, dessa leder till, så att den kan lära sig själv. Genom att låta nätverket lära sig själv är det inte nödvändigt att ge en massa regler, vilket måste göras om man hårdkodar. För ett neuralt nätverk skapas arkitekturen och sen lär den sig själv. När nätverket är tränat klarar modellen av att få en insignal, input, med data som den aldrig har sett förut och ge korrekt utsignal. Det finns olika neurala nätverk och de klassificeras oftast som framkoppling, feedforward, eller återkoppling, feedback, nätverk.

2.2 Djupinlärningsalgoritmer

Det finns fyra olika djupinlärningsalgoritmer och alla har olika krav på hur datan skall användas. Alla algoritmer har också olika metoder och man måste välja vilken metod,

med andra ord hur algoritmen används, som passar bäst för det man vill få ut av algoritmen. De fyra algoritmerna är:

- Övervakad inlärning
- Semiövervakad inlärning
- Oövervakad inlärning
- Förstärkningsinlärning

2.2.1 Övervakad inlärning, Supervised learning

Som man kan gissa från namnet finns det en ”övervakare” som har huvuduppgiften att säga till när datorn gör rätt eller fel. Algoritmen används om det finns historiska data eller så kallad märkt data som till exempel berättar att bilden av en gran motsvarar ordet gran. Alltså vet den vad som kommer in och vad som skall komma ut, datan som kommer in delar den upp i två olika delar, en del där modellen vet vad som finns och en som den testar sin inlärning mot.

Baserat på den analyserade träningsdatan produceras en slutsats som kan användas för att kartlägga nya data. Detta kräver att inlärningsalgoritmen generaliserar från träningsdatan till osedda situationer på ett rimligt sätt (Bonaccorso, 2018).

Övervakad lärning använder i huvudsak tre metoder: klassifikation, linjär regression och logistikregression. Ett praktiskt exempel är om man skall lära ett barn hur olika färger ser ut måste man först visa hur färgen ser ut och sen säga vad färgen heter, samt upprepa ett antal gånger och sedan testa om barnet kan svara rätt. Om barnet har lärt sig färgen och dess namn har målet uppnåtts och om inte måste man ändra inlärningssättet till man hittat något som fungerar.

2.2.2 Semiövervakad inlärning, Semi-supervised learning

Semiövervakad inlärning är en algoritm där lite märkt data används och mycket omärkt data används under träning av modellen, kan också förklaras som övervakad inlärning med mera information eller oövervakad inlärning med begränsningar. På grund av att det finns mycket omärkt data som används med lite märkt data så går det att få väldigt bra inlärningsnoggrannhet. Algoritmen används ofta om problem uppstår med att få mycket märkt data, men det finns tillgång till omärkt data. Den används ofta inom organisering av information som finns på olika webbsidor. Semi-övervakad inlärning är också teoretiskt intressant för att den liknar mycket hur en människa lär sig (Bonaccorso, 2018). Semiövervakad inlärning kombinerar den information den har för att överträffa klassificeringsprestanda som kan erhållas, antingen genom att kassera omärkta uppgifter och genomföra övervakad inlärning eller genom att kassera etiketterna och genomföra undervisat utan tillsyn. Metoderna som används är generativa modeller, separation med låg densitet, grafbaserade metoder och heuristiska tillvägagångssätt.

2.2.3 Oövervakad inlärning, Unsupervised learning

Med oövervakad lärning finns inte färdiga namn på datakategorierna så istället används andra metoder för att hitta samband som inte har specificerats, med så liten mänsklig övervakning som möjligt. I jämförelse med övervakad inlärning där man använder data som en människa har märkt så möjliggör oövervakad inlärning, också kallad självorganiserad inlärning, modellering av sannolikhetstäthet över indatan (Bonaccorso, 2018). De metoder som används för oövervakad inlärning är kluster analys som används för att gruppera eller segmentera datasatser med delade attribut för att extrapolera algoritmiska relationer, och huvudkomponent analys som tar punkter i två eller flera olika dimensioners utrymme och passar in en linje så nära alla punkter som möjligt.

2.2.4 Förstärkningsinlärning, Reinforcement learning

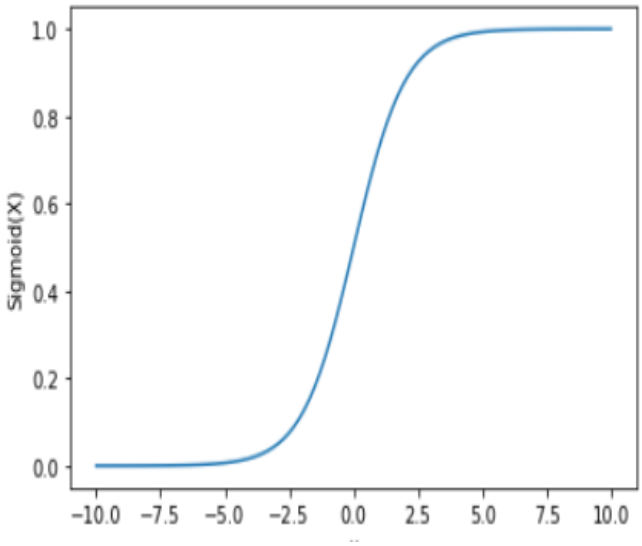
Algoritmen lär sig från noll, behöver ingen indata eller utdata utan istället fokuserar den på att hitta balansen mellan utforskning av okänt territorium och användning av nuvarande kunskap. Miljön den skapar åt sig baserar sig ofta på Markov decision process, MDP, eftersom många förstärkningsinlärningsalgoritmer för detta sammanhang använder dynamiska programmeringstekniker. Den största skillnaden mellan en klassisk dynamisk programmeringsmetod och förstärknings-inlärningsalgoritmerna är dock att den senare nämnda inte antar att den har lärt sig något eller har kunskap om en exakt MDP-modell utan den riktar sig till stora MDP:er där det blir omöjligt att fastlägga en exakt metod (Bonaccorso, 2018). Här används bara en metod som kallas för reinforcement.

Ett enkelt exempel på förstärkningsinlärning är att försöka lära en hund ett nytt trick. På grund av att hunden inte förstår vad den skall göra så belönas hunden enbart då den gör rätt. Så när hunden är i en liknande situation så vet hunden att om den gör rätt sätt så får den en belöning.

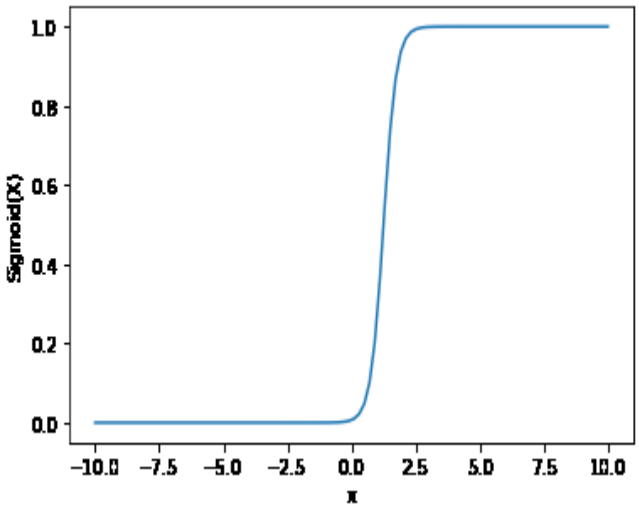
2.3 Matematiken bakom djupinlärning

Den matematiska grundfunktionen som ofta används inom djupinlärning men speciellt i klassificerings modeller är Sigmoid-funktionen. Funktionen används för att förstärka svar som är rätt och dämpa svar som är fel. Den liknar väldigt mycket en steg funktion men skillnaden är att den är kontinuerlig och differentiell. Skalan är oftast mellan 0 och 1, men kan också vara mellan -1 och 1. Om man ger ett negativt tal så går funktionen mot 0 medan om man ger ett positivt värde går den mot 1 (Han & Morag, 1995). Formelns matematiska uttryck är $\sigma(x)\frac{1}{1+e^{-x}}$ (Han & Morag, 1995) och dess graf syns i **figur 1**. I denna figur kan man se att i jämförelse med en linjärt växande funktion dämpar Sigmoid-funktionen den nedre delen av den negativa skalan, medan den lägre positiva delen förstärks. Höjningen eller sänkningen som syns på figuren sker hela tiden i neurala nätverk och det kallas för viktning och att

ingjuta partiskhet. Men sänkningen och stigningen av Sigmoid-funktionen kan i vissa fall vara för långsam så att den inte är hjälpsam, men det går att ändra på med att ändra på argumentet till $\sigma(ax + b)$ (Yadav, 2020) som exempel: $\sigma(4x - 5) = \frac{1}{1+e^{-(4x-5)}}$ (Yadav, 2020) och då ändrar grafen så att den ser ut som i **figur 2**. I den nya uträkningen står **x** för input och **a** för vikt och **b** för partisk och för att skapa en effektiv neuralt nätverk modell är det väldigt viktigt att man får bra värden på variablerna för situationen. Detta händer för en input-struktur vilket betyder att x är en skalär.



Figur 1 Sigmoid-funktion. (Yadav, 2020)



Figur 2 Sigmoid-funktionen med ändrat argument.

(Yadav, 2020)

Eftersom det handlar om flera insignaler kommer x variabeln inte att räcka till, med linjär algebra är det möjligt att skala om metoden för vektorer. Sigmoid-funktionen med vektorer X definieras som: $\sigma: R^m \to R^m\ \sigma(X) = [\sigma(x1), \dots, \sigma(xm)]$ *där* $X = [x1, \dots, xm]$. Definitionen tar komponenter i X vektorn och kartlägger komponentvis med Sigmoid-funktionen. Vikten ersätts av matrisen W och partiska blir vektorn B,

det skalade argumentet blir från $\sigma(ax + b)\ till\ \sigma(WX + B)$. Viktmatrisen (W) är av ordningen m x m, input-vektorn (X) har höjden m, partiska vektorn (B) är av höjden m. När den definierade Sigmoid-funktionen används bli resultatet lager, input och output alltså neuroner. (Yadav, 2020)

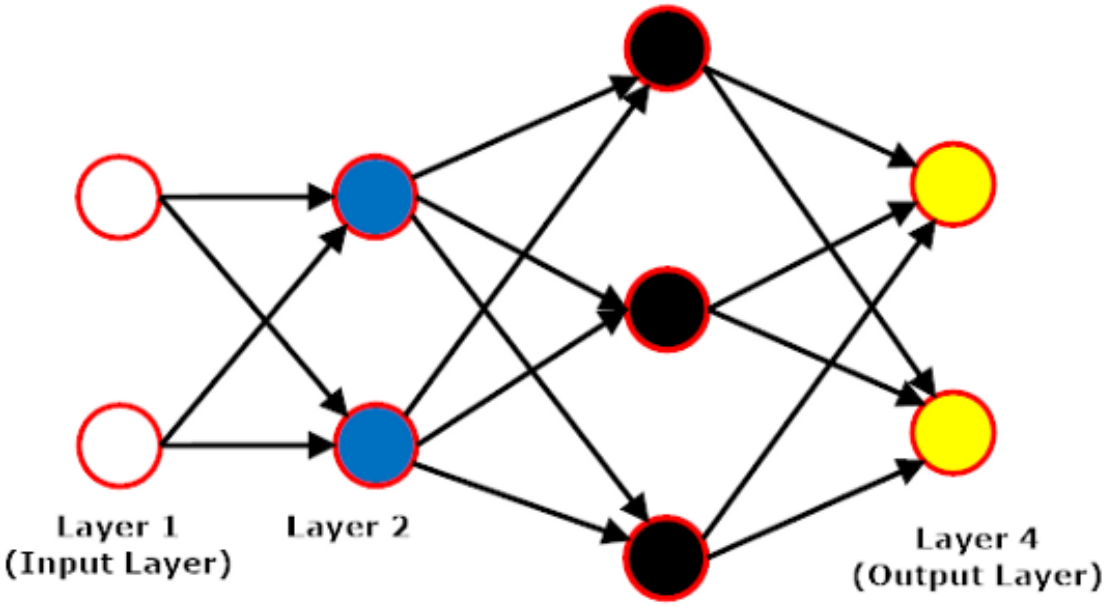
Exempel: input vektorn har höjden 2 så X=[x1, x2], partiska vektorn är B=[b1, b2] och viktmatrisen W=[w11, w12; w21, w22]. I detta fall är X input lagret som fungerar som:

$$WX + B = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$ (Yadav, 2020)

Efter ekvationen blir svaret ett nytt lager: $\begin{matrix} w_{11}x_1 + w_{12}x_2 + b1 \\ w_{21}x_1 + w_{22}x_2 + b2 \end{matrix}$ (Yadav, 2020).

Både x1 och x2 finns i båda delarna av svaret, denna procedur behövs för att skapa nya neuroner från insignalens data så ett nätverk kan skapas. Metoden kan skalas om till valfritt begränsat antal insignaler enligt variabeln m, Sigmoid-funktionen kan då skrivas som $\sum_j w_{ij}x_j + b_i$ (Yadav, 2020). För att få mera neuroner om det behövs skapas ett nytt lager med samma Sigmoid-funktion som förra gången, skillnaden är dock storleken på viktmatrisen och höjden på partiska vektorn. Som exempel om viktmatrisen ändrar så att den har höjden tre och längden två samt att den partiska vektorn har höjden tre så skapas tre nya neuroner till nästa lager. På grund av flexibiliteten i ordningen för vikt-matrisen är det lätt att få den önskade mängden neuroner till nästa lager. De lager som är mellan insignalens lager och utsignalens lager kallas för dolda lager, hidden layers, och antalet gånger som processen rekursivt upprepas blir antalet lager. Så om processen upprepas fyra gånger blir det fyra lager i ett neuralt nätverk och den matematiska funktionen blir: $F(X) = \sigma(W_3\sigma(W_2\sigma(WX + b) + b_2) + b_3)$ (Yadav, 2020)

Detta är den matematiska funktionen passar som en klassifikationsmodell. Mängden dolda lager och neuroner som måste skapas beror på användarens data. Naturligtvis kommer högre mängd dolda lager och neuroner att returnera komplexare funktioner(F(X)).



Figur 3 Resultat av ekvationen för fyra lager. (**Fung**, **2017**)

2.4 Val av ramverk för djupinlärning

Ett djupinlärnings ramverk behövs för de flesta personer som försöker skapa djupinlärningsmodeller, eftersom de inte har tid för att skapa eller implementera ett neuralt nätverk från grunden. Det kan dessutom ta flera veckor för att träna modellen om nätverket inte är bra optimerat. Ramverk är ett bibliotek som bygger modeller snabbt och enkelt med hjälp av underliggande algoritmer (Sciforce, 2020). I stället för att behöva skriva långa koder används färdiga funktioner och moduler som kallas på i korta kodkommandon för att skapa modellerna. De olika ramverken ser ofta lika ut på ytan men är ofta optimerade för olika ändamål, så som bildklassificering eller ljudigenkänning o.s.v. Det finns inget allmänt accepterat sätt för att använda ramverkens funktionalitet via deras gränssnitt för applikationsprogrammering (API, Application Programming Interface).

2.4.1 Tensorflow

Tensorflow är ett av de populäraste och mest använda ramverken någonsin. Det skapades 2015 av en av Googles avdelningar för artificiell intelligens Google Brain. Anledningen till att det är så populärt är att ramverket har många olika verktyg och bibliotek som samarbetar. Tensorflow har mycket färdigskriven kod som kan användas för många olika sorters modeller (Tensorflow, 2020). Ramverket är öppen källkod, så alla kan använda det. Det körs bäst i Python, men går att använda i andra kodprogram. Det kan dock vara problematiskt att läsa kod i kodprogram som C++.

De mest populära användningsområdena för ramverket är naturlig språkbehandling (NLP, Natural Language Processing) applikationer, bildigenkänning, ljudigenkänning,

analys av tidsserier och videoanalys (Sciforce, 2020). Tensorflow är ett väldigt bra djupinlärningsverktyg, men det är inte ett ramverk för nybörjare på grund av dess otydliga sätt att visa beräkningar och grafer, vilket gör inlärningskurvan lång i de flesta fall.

2.4.2 Caffe

Ursprungligen släpptes Caffe 2014 från University of California, senare har det dock skapats flera versioner. Bland dessa ramverk har Caffe2 och NVCaffe inkluderats som är en del av PyTorch. Ramverket är mest fokuserat på bildhantering. Det bästa med Caffe är att det finns många färdigt tränade modeller, nätverk eller vikter beroende på vad som behövs. Alla färdigt tränade modeller får användas fritt under BAIR modellens licens, vilket är en bra start för nybörjare. Populära användningsområden är lätt regression, storskalig visuellklassifikation, robotapplikationer samt siamesiska nätverk för att jämföra tal och bilder (Sciforce, 2020). Ramverkets svagheter är dock en komplicerad installationsprocess och dess krav på många olika program för att fungera felfritt.

2.4.3 PyTorch

PyTorch släpptes 2017 av Facebook och är en efterträdare till Torch som släpptes 2011. Skillnaden är att PyTorch har allting som Torch har, men ramverket kan köras i Python och C++. PyTorch har på grund av dess flexibilitet och mängd användningsområden blivit ett av de populäraste ramverken. Inom områdena tensorberäkning och djupa neurala nät (DNN, Deep Neural Network) konkurrerar PyTorch med Tensorflow. Övriga användningsområden inkluderar förstärkningsinlärning, NLP relaterade uppgifter och bildklassificering (Sciforce, 2020). PyTorch används i detta arbete som en del av Fast.ai. Detta på grund av att PyTorch använder en API på en mindre komplicerad nivå och har väldigt bra färdiga funktioner som är lättanvända, samt kan visa, ändra och skapa grafer eller diagram.

2.4.4 Keras

François Chollet skapade ramverket 2014 för att underlätta snabb experimentering och utveckling genom att använda en gemensam och abstrakt API. Detta ramverk är motsatsen till PyTorch, för att det använder en API med mer komplicerade kommandon och hanterar lager på annat sätt än tidigare nämnda ramverk. Keras kan också köras på andra ramverk som Tensorflow, MXNet etcetera med hjälp av Python API. Ramverket används mest för bild och NLP relaterade modeller (Sciforce, 2020). Det försöker vara mest lämpat för nybörjare och personer som vill få snabba resultat som är pålitliga.

Det finns två olika modeller som Keras skapar, dessa är sekventiella API och funktionella API. Sekventiell API under träning har alla lager definierade i rätt

ordning från början och implementerar dem en och en. Den funktionella API:n är mer flexibel, kan skapa mera komplexa modeller och använder sig av icke-linjär topologi, det vill säga delade lager eller många in och utsignaler (Keras, 2020).

2.4.5 Fast.ai

Ramverket använder PyTorch som grund, men 2018 när ramverket skapades använde det Keras. Fast.ai kan implementera optimerade algoritmer på endast 4–5 kodrader och är ett GPU optimerat datorseendebibliotek. Det använder sig av en ny data block API och ett tvåvägs tillbakakallande system, vilket när som helst kan ändra eller optimera vilken del av datan eller modellen som helst (fastai, 2020).

Huvudsakliga användningsområden är bildrelaterade. Dessutom har skaparna av fast.ai gjort flera kurser om ramverket och kodbiblioteket, vilket gör att nybörjare lätt kan lära sig steg för steg processen för att skapa en fungerande och exakt modell. Ramverkets svagheter är att det är tidskrävande att hitta rätt ställe för bilderna i Jupiter Notebook, som används som ett utbildningsverktyg, om inget färdigt dataset används. Jag fann det tidskrävande att få den första koden för ramverket att fungera så att träningen av modellen kunde börja.

2.5 Resnet

Resnet uppfanns 2015 och är en förkortning av residual network, vilket på svenska översätts till kvarvarande nätverk. En revolutionerade implementation av djupinlärning gör det möjligt för resnet att träna upp till flera hundratal olika lager och fortfarande få bra prestanda.

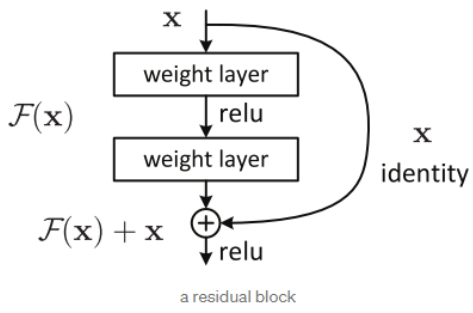
Ett vanligt framkopplande nätverk med ett lager är tillräckligt för att representera vilken funktion som helst. Problemet är att det ensamma lagret kan bli väldigt stort och då är det stor risk för överanpassning. På grund av detta försöker man anpassa nätverkets arkitektur för att arbeta på ett djupare plan. Som exempel kan man ha ett lager som lär sig att känna igen kanter, andra lager känner igen strukturer och tredje känner igen objekt på en bild. Det finns många andra djupa nätverksversioner, så som neurologiska (VGG) nätverk som använder sig av sexton till nitton lager. GoogleNet använder sig av hela tjugotvå lager. Men att öka antalet lager för mycket leder till nya problem.

Det är svårt att träna djupa nätverk på grund av det försvinnande gradient problemet, som uppstår då för många lager används för användningsändamålet. Detta på grund av upprepad bearbetning av data som leder till degradering av informationen i vikten inuti lagren. Prestandan slutar då bli bättre och kan i vissa fall snabbt bli sämre.

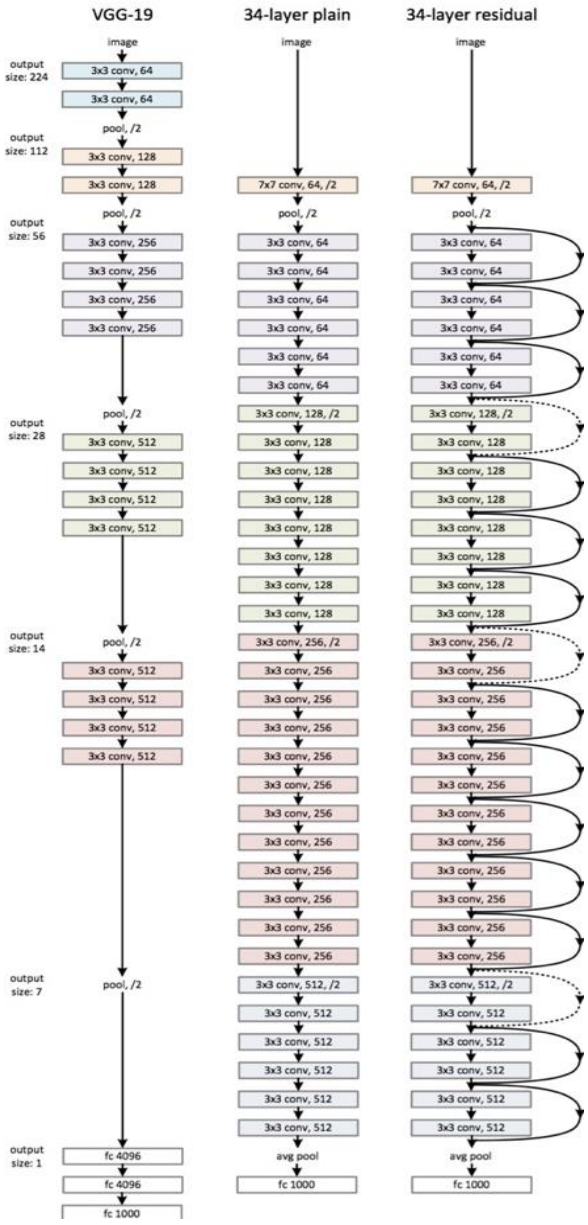
För att undvika detta hoppar resnet över lager genom att återanvända aktiveringar från ett föregående lager tills det intilliggande lagret lär sig det föregående lagrets vikter. Under träning, när resnet används, anpassar vikterna sig mot insignalernas

lager och på samma gång förstärks de tidigare överhoppade lagren. Detta fungerar bäst när ett olinjärt lager hoppas över, eller när de lager som hoppats över är linjära. Annars bör en viktmatris läras för de överhoppade anslutningen. Att hoppa över lager förenklar nätverket effektivt. Detta påskyndar inlärningen genom att minska på försvinnande gradient problemet, eftersom det finns mindre lager att sprida igenom. Nätverket återställer sedan de undangömda lagren när det lär sig funktionsutrymmet. Mot slutet av träningen när alla lager har återställts närmar sig nätverket rätt inlärt beteende. Ett nätverk utan kvarvarande delar utforskar ofta för mycket av funktionsutrymmet vilket leder störningar vilket i sin tur leder till att mera data krävs. (He, Zhang, Ren, & Sun, 2020)

Kvarvarande nätverk har oftast mindre lager än ett VGG-nätverk men nätverket infogar en genvägsanslutning som gör så att kvarvarande nätverket motsvarar andra nätverk med flera lager, på figur 4 är genvägen märkt med x identity. Identitetsgenvägarna F (x {W} + x) kan användas direkt när ingången och utgången är av samma dimensioner. När dimensionerna ökar så överväger nätverket två alternativ. Antingen så utför genvägen identitetskartläggning, med extra nollposter vadderade för ökande dimensioner. Detta alternativ introducerar ingen ytterligare parameter. Det andra alternativet är att genvägen som skapas av funktionen F (x {W} + x) används för att matcha dimensionerna vilket görs med 1 x 1 upplösning. För båda av alternativen, om genvägarna går över funktionskartor av två storlekar, utfördes de med ett steg av två. Varje resnet block är oftast två lager djupt och då används resnet 18 eller 34 som är mindre eller så är blocket tre lager djupt och då används resnet 50, 101, 152.



Figur 4 Resnet-lager med genväg. (Fung, 2017)



Figur 5 Resnet34 mot Enkelt nätverk med 34 lager. (Ruiz, 2020)

3 Praktiska processen

I detta avsnitt förklaras hur den praktiska processen för att skapa en modell går till och vilka problem som kan uppstå när man skapar en modell. Jag går även igenom provkörningen av en modell och hur man skall utvärdera hur bra modellen egentligen är. Till sist presenteras resultatet och skillnaden mellan en bra och dålig modell.

3.1 Inlärningsprocessen i praktiken

Innan träningen kan påbörjas av en modell måste bilder laddas ner och sätta in i rätt mappar. Det första steget är att installera PyTorch (1.) och fastai biblioteket (2.) i Jupyter Notebook, den version av Python som används i detta arbete. Egna bilder kan användas och laddas upp en mapp till Jupyter Notebook. För denna modell samlades bildlänkar in från Google bilder. Sökorden är till exempel *Fir*(gran) och sen skrivs det in en JavaScript kodrad i konsolfliken för webbsidan som samlar alla länkar i en fil som en lista(3.). Men innan listan kan användas måste filen omvandlas till ett csv format så att Python enkelt kan ladda ner bilder från länkarna. Sedan specificeras vilken fil som den skall ta datan från och vilken mapp den skall spara data till(4.) och efter det vägen till mappen som Python skall använda(5.). Sedan laddas bilderna ner från filen till mappen (6.). Steg tre till sex upprepas för varje objekt som modellen skall kunna klassificera.

Nu verifieras och normaliseras bilderna. Benämner först de olika klasserna (7.). När all data finns inuti mappar i Jupyter Notebook måste det kontrolleras att alla olika kategorier av data som behövs finns till exempel gran, tall, björk, al etcetera (8.). Efter tas det fram ett slumpmässigt dataurval som är tagen från den data som finns inuti datamapparna och normaliserar dem(9.). Dessa bilder kommer användas för att lära modellen. I nästa steg visas några bilder(10.), kontrollera att bilderna är av bra kvalitet. Om bilderna är av dålig kvalité gör steg nio och tio på nytt. Nu börjar själva inlärningen. En elev(learner) skapas, som sedan används på de bilder som finns i vårt dataurval(11.). I learner kommandot specificeras vårt dataurval som skall användas, vilken modell av resnet den skall använda och till sist vilken sorts mätvärde den skall använda för att mäta hur bra eller dåligt modellen lär sig vilket i detta fall är felprocenten, alltså hur mycket den har fel.

Att hitta rätt inlärningshastighet är ett av om inte det viktigaste steget i hela processen för att modellen skall kunna lära sig och bli så träffsäker som möjligt. Nu instrueras eleven hur länge, antal cykler även kallade epoker, och med vilken inlärningshastighet den skall lära sig med(12.). Inlärningshastigheten är ett mått på tid som skrivs som en siffra upphöjt i ett negativt tal, till exempel 3e-3. Det syns i detta skede om det är nödvändigt att köra igenom det på nytt med samma inlärningshastighet eller om det behöver prövas med andra inlärningshastigheter. Det skall köras igen med samma inlärningshastighet om felprocenten går neråt kraftigt. Men om det går sakta neråt eller går neråt kraftigt och sedan upp tillbaka så måste inlärningshastigheten bytas. I detta skede testas det om datan är bra. Helst

skall felprocenten vara under 10 procent innan man fortsätter vidare. Efter det sparas eleven så om något går fel kan den laddas om från detta skede(13.).

I denna del hittas intervallet för den bästa inlärningshastigheten och undersöker om det finns eventuella problem med vår modell. Fast.ai letar upp den optimala inlärningshastigheten via inlärningshastighetssökaren(14.). När den har kört klart ritas en graf upp(15.). I grafen där kurvan börjar sjunka och just före den börjar gå uppåt tillbaka finns den bästa inlärningshastigheten. Med dessa värden körs eleven igen för att lära sig i en ny cykel(16.), här behövs det mindre epoker på grund av att inlärningshastigheten borde vara så pass bra att modellen inte behöver köra genom datan lika många gånger. Så länge som felprocenten är under 10 procent är det skäl att spara i detta skede. Om resultaten är bra kan två tolkningskommandon skrivas in (17.) som ritar upp en förvirringsmatris(18.), denna visar hur många bilder av en viss kategori modellen fick rätt. Samt hur många bilder av samma kategori den trodde att var någonting annat.

Om resultaten från föregående steg inte gav bra felprocent(>10%) skall det granskas om problemet finns bland bilderna. Börjar med att skapa ett nytt dataurval (19.) och elev (20.). Inuti den nya eleven laddas kopian in från det andra stadiet (stage-2) (21.). Detta görs så att inte en helt ny elev behöver läras upp, utan startar från en punkt som fungerade någorlunda, men inte så bra som önskades. För att få denna elev skall bli bättre än den äldre så måste datan städas, genom att använda ett kommando som skapar en dataurvalsformaterare (22.). Den tar fram alla bilder som modellen har väsentliga problem med att välja rätt kategori för(top losses). Bildrensaren (ImageCleaner) visar bilderna, så att de kan markeras och sedan tas bort om det behövs(23.). Efteråt skapas en ny fil som heter ”cleaned.csv”, i vilken endast de godkända bilderna finns med. Efter det körs alla de kodraderna från steg fyra på nytt med cleaned.csv, detta borde resultera i bättre resultat så att modellen kan testas.

Verifierar att modellen är bra mot alla bilder och sedan exporteras en fungerande modell som är redo att användas. Testandet görs genom att exportera eleven (24.) och välja om den skall köras med cpun eller gpun (25.). Väljer nu ut en bild som modellen kan testas mot (26.), efter det laddas eleven in (27.). Om allting har gått rätt borde modellen identifiera trädet rätt på testbilden (28.).

Kodrader:

1. conda install -c pytorch -c fastai fastai
2. from fastai import *
   from fastai.vision import *
3. urls=Array.from(document.querySelectorAll(‘.rg_i')).map(el=>
   el.hasAttribute('data-src')?el.getAttribute('data-src'):el.getAttribute('data-iurl'));window.open('data:text/csv;charset=utf-8,' + escape(urls.join('\n')));
4. folder = 'treeai'
   file = 'tree.csv' #’fir.csv’
5. path = Path('data/treeai')
   dest = path/folder #path/fir

```
    dest.mkdir(parents=True, exist_ok=True)
6. download_images(path/file, dest, max_pics=200)
7. classes = ['Birch','Fir','Pine']
8. for c in classes:
     print(c)
     verify_images(path/c,delete=True, max_workers=8)
9. np.random.seed(42)
   data = ImageDataBunch.from_folder(path, train=".", valid_pct=0.2,
        ds_tfms=get_transforms(), size=224,
   num_workers=4).normalize(imagenet_stats)
10. data.show_batch(row=3,figsize=(7,8))
11. learner = create_cnn(data, models.resnet34, metrics=error_rate)
12. learner.fit_one_cycle(4, 3e-3)
13. learner.save('stage-1')
14. learner.lr_find()
15. learner.recorder.plot()
16. learner.fit_one_cycle(2, max_lr=slice(3e-5, 3e-3))
17. interp = ClassificationInterpretation.from_learner(learner)
18. interp.plot_confusion_matrix()
19. db = (ImageList.from_folder(path)
                .split_none()
                .label_from_folder()
                .transform(get_transforms(), size=224)
                .databunch())
20. learn_cln = cnn_learner(db, models.resnet34, metrics=error_rate)
21. learn_cln.load('stage-2');
22. ds, idxs = DatasetFormatter().from_toplosses(learn_cln)
23. ImageCleaner(ds, idxs, path)
24. learner.export()
25. defaults.device = torch.device('cpu')
26. img = open_image(path/'Fir'/'test.jpg')
27. learner = load_learner(path)
28. pred_class,pred_idx,outputs = learner.predict(img),pred_class.obj
```

3.2 Problem som kan uppstå

De största problemen som kan uppstå när en modell skall tränas är att felprocenten är för hög eller träffsäkerheten är för låg för att kunna använda modellen. Den största anledningen till dessa problem är antingen att inlärningshastigheten är för hög eller låg. Ett annat problem kan vara att mängden epoker är för många eller för få, antalet epoker är hur många gånger datorn kollar igenom datan och lär sig materialet. Något av dessa värden behöver ändras om träningsförlusten (train_loss)är högre än vad den giltiga förlusten (valid_loss) är, vilket aldrig borde ske. Modellen kan bli relativt bra

med en högre träningsförlust, men den kommer aldrig att bli så bra som den skulle kunna vara.

- Med för hög inlärningshastighet kommer giltig förlust att bli alldeles för hög på grund av att datorn lär sig för snabbt så att modellen inte tar lärdom av bilden den granskar.
- Med för låg inlärningshastighet kommer den att lära sig men det kommer att ta alldeles för länge för att det skall vara lönsamt, när man med en högre inlärningshastighet får ett mycket snabbare och möjligtvis bättre resultat.
- Om inlärningstillfället har för många epoker kommer den att köra igenom dataurvalet för många gånger och den kommer inte att lära sig till exempel hur en björk ser ut, utan den kommer att lära sig hur just de björkar som finns på bilderna ser ut. Detta fenomen kallas överanpassning, Overfitting, överanpassning leder till att felprocenten blir bättre en stund men sen blir den snabbt sämre igen.
- Med för lite epoker kan man få bra resultat, men träningsförlusten blir högre än den giltiga förlusten, för att modellen har kört igenom dataurvalet för få gånger.
- Inte lika troligt men det är ändå möjligt att det finns för lite data tillgängligt. Huvudproblemet med för lite data är att fastän modellen har en bra inlärningshastighet sjunker inte felprocenten som den borde. Vidare ger den dåliga resultat utanför sitt dataurval.

Problem kan även uppstå med datan som används. De största anledningarna till att problem eller dåliga resultat uppstår med bild data är att bilderna ser för lika ut på grund av vinklar, ljus, bildkvalitet och/eller miljö som bilden har tagits i.

3.2.1 Problem under examensarbetets gång

De flesta av problemen som uppstod under arbetets gång var på grund av tiden som det tar att lära sig något djupt från grunden. Inlärningen av kodningsprogrammet, kodbiblioteket och djupinlärningens grunder tog lång tid. Största problemet som uppstod i början av slutarbetets gång var förknippade med Jupyter Notebook och dess molnbaserade lagringssystem. Kursen jag följde var inte tillräckligt tydlig när det gällde Jupyters Notebooks säregenskaper. Under inlärningen av modellen var det utmanande att få en bra felprocent och att sänka träningsförlusten så att den var lägre än den giltiga förlusten. Största anledningen till detta var att den ursprungliga datan som användes var av låg kvalitet och att inlärningshastigheten var opasslig. Kombinationen av dessa problem gjorde det svårt att undersöka modellen.

3.2.2 Tester

Modellen har tränats på ungefär 300 bilder och testats på 120. Av dessa 120 fick den 116 rätt vilket betyder att modellen hade en felprocent på 3,33 procent (96,67 procent rätt). De fyra bilder som den fick fel hade för mycket av andra trädslag i

samma bild eller dåligt ljus så att trädet kunde uppfattas som ett annat trädslag. Här är exempel på bilder som modellen har testats med.

```
img = open_image(path/'Testbilder'/'birch_1.jpg')
img
```

```
learn = load_learner(path)
```

```
pred_class,pred_idx,outputs = learn.predict(img)
pred_class.obj
```

```
'Birch'
```

Figur 6 Testbild av en björk.

```
img = open_image(path/'Testbilder'/'fir_1.jpg')
img
```

```
learn = load_learner(path)
```

```
pred_class,pred_idx,outputs = learn.predict(img)
pred_class.obj
```

Figur 7 Testbild av en gran.

```
img = open_image(path/'Testbilder'/'pine_1.jpg')
img
```

```
learn = load_learner(path)
```

```
pred_class,pred_idx,outputs = learn.predict(img)
pred_class.obj
```

```
'Pine'
```

Figur 8 Testbild av en tall.

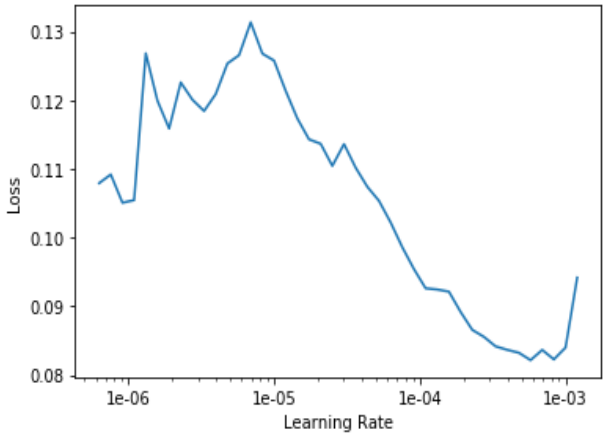
3.2.3 Resultat

Dessa figurer visar resultaten från modellen när träningsförlusten var högre än giltiga förlusten. Felprocenten gick inte lägre än 0.23 vilket betyder att modellen skulle ha fel 23% av gångerna, alltså nästan en fjärdedel av alla fall.

epoch	train_loss	valid_loss	error_rate	time
0	1.599636	0.918684	0.375000	00:32
1	1.266243	0.830664	0.233333	00:28
2	1.033394	0.776137	0.241667	00:27
3	0.900891	0.739480	0.250000	00:27

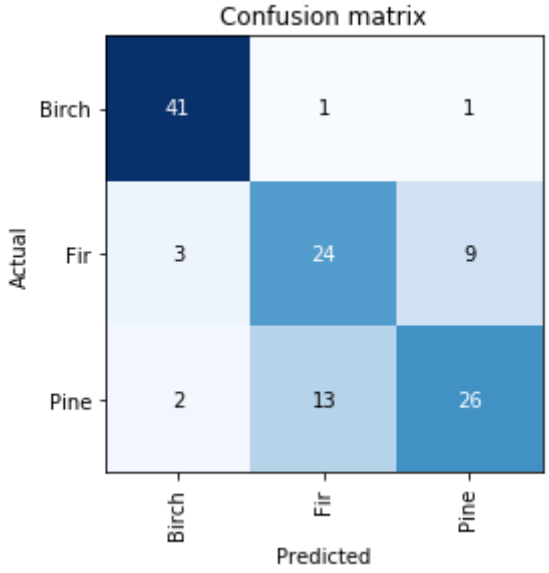
Figur 9 Felprocenter från tidig träning.

Figuren undertill visar inlärningskurvan från inlärningshastighetssökaren för en modell med dåliga data. Modellen hade svårt att lära sig i början, vilket gav upphov till toppar i grafens begynnelse. När modellen väl började lära sig var inlärningshastighetens förbättring för snabb, vilket gör så att den inte lär sig tillräckligt bra.



Figur 10 Graf av en dålig inlärningshastighetskurva.

Följande figur visar resultatet av en modell med dåliga data och dålig inlärningshastighet. Modellen har identifierat fel på över hälften av bilderna mellan tall och gran, vilket gör denna modell helt oanvändbar.



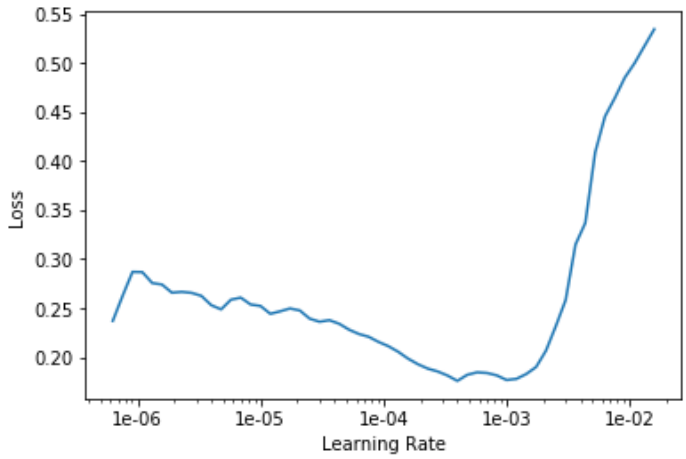
Figur 11 En dålig förvirringsmatris.

De figurer som kommer näst är resultatet av då städad data och rätt inlärningshastighet används. Dessa värden kommer efter att den bästa inlärningshastigheten från inlärningshastighetssökaren använts. Datan är städad så att det inte finns bilder som modellen blir förvirrad av. Resultatet av detta är att felprocenten endast är 0.046, vilket betyder att modellen endast har fel 4,6 procent av gångerna, vilket är acceptabelt.

epoch	train_loss	valid_loss	error_rate	time
0	0.064684	0.269035	0.076923	00:21
1	0.047539	0.191319	0.046154	00:21

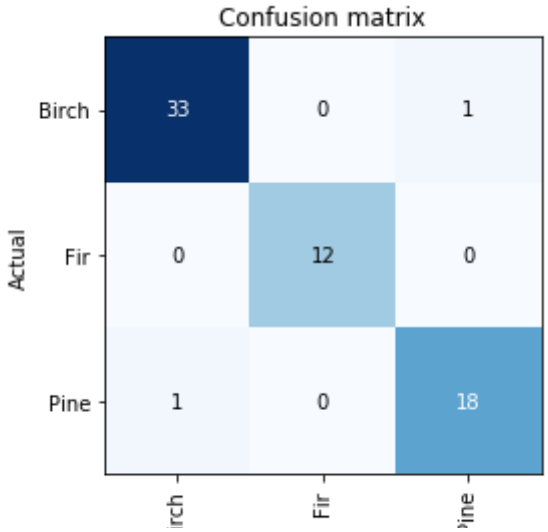
Figur 12 Låga felprocenter p.g.a. bra bilder.

Anledningen till att inlärningshastighetskurvan från inlärningshastighetssökaren är bra är för att kurvan är jämn, alltså utan hackiga toppar. Den går även sakta neråt vilket betyder att modellen lär sig i en lugn och bra takt.



Figur 13 En inlärningskurva utan toppar och bra lutning.

I figuren nedan syns hur en bra förvirringsmatris skall se ut. Modellen kommer alltid att få någon bild fel, men så länge som det endast är någon enstaka bild är det ändå en bra modell. Jag tror att anledningen till att det uppstår problem mellan just björk och tall i detta fall är ljuset i bilderna och formen på trädens kronor.



Figur 14 Förvirringsmatris med få felidentifieringar.

4 Diskussion

Fast.ai har planer på att släppa en version av ramverket som klarar av objekt detektering under 2021. Om det händer skulle modellen kunna vidareutvecklas så att modellen kan känna igen flera objekt på samma bild. När detta fungerar med en låg felprocent kommer utvecklingen gå mot hur alla objekt som har identifierats från bilderna skulle räknas ihop och visas i ett lättläst format. Ett annat alternativ är att använda ett ramverk som redan kan göra objekt detektering så som You Only Look Once (YOLO) v3 och v4.

Inom ramarna för detta slutarbete undersöktes flera olika ramverk som till exempel Fast.ai, Tensorflow och YOLOv3: Tensorflow hade allt för komplicerad installationsprocess och ramverket har allt för lång inlärningstid, YOLO är mera komplicerat och fanns inga gratis resurser för att lära sig språket. Fast.ai var med facit i hand inte det bästa ramverket men anledningen till att det valdes var för att ramverket var lättlärt och det fanns goda resurser för att lära sig på en djup nivå. Fast.ai var också bra som introduktion till djupinlärning och ger goda möjligheter för vidare specialisering.

Klassificeringsmodellen skulle kunna bli ännu bättre genom att gå ut i skogen och ta alla foton själv med rätta vinklar och ljus för ens egna ändamål, vilket skulle ge en lägre felprocent. Vidare kunde modellen bli mer allmän genom att lägga till mera trädslag så som al, asp eller rönn. Detta skulle användas för att se procentuellt hur mycket av de träden som är intressanta för industrin det finns på skiften, så som tall, gran, björk. När denna information finns kan det tas ett beslut om det behövs en skogsmaskin till skiftet i fråga, eller om det räcker med manuellt arbete.

5 Referenser

Bonaccorso, G. (2018). *Machine learning algorithms.* Packt.

fastai. (2020). *https://www.fast.ai/2020/02/13/fastai-A-Layered-API-for-Deep-Learning/*. Hämtat från fastai.

Fung, V. (den 15 Juli 2017). *https://towardsdatascience.com/*.

Gugger, J. H. (2020). *Deep learning for coders with fastai & PyTorch.* O'reilly Media.

Han, J., & Morag, C. (1995). The influence of the sigmoid function parameters on the speed of backpropagation learning. i *From natural to artificial neural computation : International Workshop on Artificial Neural Networks, Malaga-Torremolinos, Spain, June 7-9, 1995 : proceedings* (ss. 195-201). Torremolinos: Berlin ; New York : Springer-Verlag.

He, K., Zhang, X., Ren, S., & Sun, J. (den 18 8 2020). Deep Residual Learning for Image Recognition .

Higham, C. F., & Higham, D. J. (2019). Deep Learning: An Introduction for Applied Mathematicians. SIAM Review.

Keras. (den 25 11 2020). *Keras*. Hämtat från https://keras.io/

Ruiz, P. (den 15 12 2020). *towardsdatascience.com*. Hämtat från Towards data science: https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

Sciforce. (den 28 11 2020). *How to Find a Perfect Deep Learning Framework*. Hämtat från https://medium.com/sciforce/how-to-find-a-perfect-deep-learning-framework-3cc00a56334d

Skogscentralen. (den 21 Februari 2020). *https://www.metsakeskus.fi/sites/default/files/maastotarkastusohje-sv.pdf.*

Tensorflow. (den 25 11 2020). *Tensorflow*. Hämtat från https://www.tensorflow.org/

Yadav, A. (den 7 Juni 2020). *https://medium.com/artifical-mind/simple-mathematics-behind-deep-learning-c38152c8b534*.