Evaluierung von Transferlernen mit Deep Direct Cascade Networks

Simon Tarras

July 8, 2025

Contents

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Chapter 1

Einleitung

1.1 Einführung

Selbst diejenigen, die nichts mit Informatik zu tun haben, kennen heute KI. Dahinter stecken neuronale Netze. Diese werden meistens in einem Zug komplett gebaut und trainiert. Das dauert lange, sodass das Kaskadierungsverfahren der Cascade Correlation [ML90] entwickelt wurde. Dabei wurde bemerkt, dass es in Ordnung funktioniert und das schon bei kleineren Netzen. Deshalb wurden darauf aufbauend weitere Netzwerke und Kaskadierungsverfahren entwickelt [LR92b], [TG99], [MHN18]. Es wurden bei den Kaskadennetzwerken ebenso festgestellt, dass es mit diesem Aufbau der Netze einfach ist, einen Wechsel zwischen unterschiedlichen Daten und Aufgaben durchzuführen [Mar19], [TS], [JY10]. In dem Bereich des Wechsels zwischen unterschiedlichen Daten ist diese Arbeit anzusiedeln, denn sie überprüft, wie gut die einzelnen Netzwerkvarianten sind und geht auf die Probleme von TF und Cascade ein.

1.2 Motivation

Es gibt Aufgaben, die gerne von einer KI übernommen werden sollen, für die es nur sehr kleine Datensätze gibt. Diese sind so klein, dass das neuronale Netzwerk hinter der KI nicht genügend Daten für das Training hat, um ein ausreichend gutes und belastbares Ergebnis zu liefern. Es ist dabei auch egal, wie lange trainiert wird; es bleibt schlecht.

Gleichzeitig gibt es den Punkt, dass es meistens ziemlich lange dauert ein Netzwerk komplett zu trainieren. Beides soll verbessert werden.

Die Situation mit den zu wenig Daten wird darüber versucht zu verbessern, indem etwas anderes, aber ähnliches, gelernt wird. Dies folgt dem Prinzip, wie die Menschen andere ähnliche Dinge lernen können, wenn sie das Eine bereits können. Ebenso verlaufen Eselsbrücken für Menschen einem ähnlichem Prinzip. Diese Art des Lernens, die über bereits Bekanntes etwas neues lernt, soll nun auch für KI angewandt werden.

Die Trainingsdaten werden so reduziert, dass das, was tatsächlich im selbem Augenblick trainiert wird, nur sehr wenig ist. Dies soll über die Kaskadierungsverfahren Deep Cascade und Direct Cascade gehen.

1.3 Related Work

Diese Arbeit baut auf den CasCor-Algorithmus [ML90], der wegen der langen Trainingsdauer entwickelt wurde, auf und nutzt diesen, um Direct Cascade [LR92b] durchzuführen, wie es in der Thesis von Marquez [Mar19] verwendet wurde. Zudem wird Deep Cascade verwendet als Vergleichsbasis [MHN18]. Dabei sind Deep Cascade Netzwerke solche, die iterativ aufgebaut werden [TG99]. Es wird nur Domain-Wechsel angewandt, während es noch den Taskwechsel [TS] gibt. Zudem gibt es bei Transfer Learning drei Probleme [JY10], die hier auch bearbeitet werden.

Im Folgenden ist die Literaturrecherche:

Neural Networks werden für eine Reihe an Tasks genutzt, die ein, dem menschlichen Gehirn ähnlichem, Rechenwerk benötigen. Es gibt neben den computer vision Tasks auch die control and planning Tasks, die für Spielecomputer für Strategiespiele wie Go dienen [CK19]. Genauso gibt es Netze, die eine Classification Task lösen [LeC+]. Eine weitere Aufgabe von neural Networks ist es eine Funktion zu approximieren. Dies wird mithilfe von Regression Networks gemacht [Spe91].

Dabei kommt es aber vor, dass es zu wenig Daten für den Task gibt [JY10]. Zudem gibt es noch das Problem, dass es viel Zeit kosten kann ein adäquates neural Network zu trainineren [ML90]. Was bei allen Netzen vorkommt ist, dass sie nicht zu hundert Prozent korrekt sind [RCN07].

Transfer Learning (TF) ist, wenn bereits gelerntes Wissen angewendet oder genutzt wird, um etwas anderes zu erlernen [TS]. Das neural Network lernt etwas. Wenn es das kann, soll es etwas anderes lernen und das bereits gelernte Wissen dafür nutzen. Dabei wird der Ursprung dieser Daten Source und das Ziel Target genannt [Mar19]. Das ähnlichste Prinzip dieser Lernmethode ist es, wenn ein Mensch sich Sachen über Eselsbrücken einprägt. Dort wird auch nicht nur direkt das gelernt, was eigentlich gelernt werden soll. Ein Netzwerk, welches TF macht, muss also einen Wechsel vollziehen. Es gibt dabei drei Research issues:

1. What to transfer
2. How to transfer
3. When to transfer

[JY10] Das Erste davon ist die Frage mit was für anderen Daten die Eselsbrücke gebaut wird. Die Lernalgorithmen entscheiden über den zweiten Punkt. Wann das TF gemacht wird, wird meistens über Fehlermetriken entschieden. Sobald das Netz einen Threshold erreicht, bei dem der Fehler gering genug ist, wird der

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Wechsel gemacht. Dies kann aber beliebig ausgetestet werden, um einen optimalen Zeitpunkt zu finden. TF besitzt aber eine Grenze bei der Verbesserung des Netzes. Des Weiteren wird das Netz durch TF nicht zwingend besser, sondern kann auch schlechter werden, was Negative Transfer ist [JY10].

Dabei gibt es zwei Bereiche, in denen ein solcher Wechsel stattfinden kann. Damit TF gemacht wird, muss es mindestens in einem dieser Bereiche einen Wechsel geben. Es gibt dort zum einen die Domain, was den Datensatz oder den Shape der Daten ausmacht und zum anderen der Task, der sowas wie Bilderkennung oder Picture Segmentation ausmacht [JY10]. Zusätzlich gibt es auch die Möglichkeit, dass der Wechsel nicht direkt auf die Zieldomain gemacht wird, sondern es zwischendurch auf eine Zwischendomain gewechselt wird und dann auf die Zieldomain [KCR17]. Dies wird bridge Transferlearning genannt und wird angewendet, wenn sich der Shape des Feature Spaces ändert oder es durch zu große Unterschiede zwischen Source und Target zu Negative Transfer kommt [JY10].

Deep Learning kann komplexere Aufgaben lösen als Shallow Learning, allerdings benötigt das Trainieren relativ viel Zeit. Wenn diese nicht vorhanden ist, wird Cascade Learning angewendet [ML90]. Für gewöhnlich wird bei TF ein Pre-trained Network genutzt und als Ausgangspunkt des Transfers der letzte Feature Vector, da angenommen wird, dass dieser die meisten Informationen enthält. Diese Art nennt man Feature-Representation-Transfer [JY10]. Das ist bei Deep Transfer Learning erst spät der Fall. Bei einem Cascade Learning wird in jeder Iteration der Feature Vector ins Transfer Learning gegeben. Dadurch wird herausgefunden, welcher Feature Vector tatsächlich die meisten Informationen hat. Das neural Network wird dabei effizienter und weniger komplex als es bei Deep Transfer Learning ist [Mar19].

Cascade Networks werden während des Trainings erst gebaut. Dazu werden Constructive Algorithms genutzt. Diese haben den Vorteil, dass es unwichtig ist, die genaue Größe des Netzes vorher zu wissen [TG99]. Damit wird das Netz groß genug sein, um eine gute Lösung zu liefern, aber auch nicht so groß, dass das Training von vornherein lange dauern muss. Gleichzeitig hat es den Vorteil, dass die Trainingszeit geringer ist [TG99], [ML90]. Ein Beispiel dafür ist Cascade Correlation (CasCor). Hierbei wird immer ein Perceptron hinzugefügt. Dieses wird dann auf die höchste Correlation trainiert, bis es keine Verbesserung mehr gibt. Dann werden die Weights vom Perceptron gefreezt, sodass diese sich nicht mehr verändern können. Danach gibt es eine Überprüfung, ob das neural Network gut genug ist und falls ja, dann ist das Training beendet. Falls nein, wird ein weiteres Perceptron hinzugefügt und wieder trainiert. Das wird solange wiederholt bis das Netzwerk einen so geringen Error hat, wie es erwünscht ist [ML90]. Es hat durch das Erschaffen der Perceptrons und dem Freeze von den Weights den Vorteil, dass es schnell lernt und es kein Backpropagation durch das ganze Netz geben muss. Das liegt daran, dass die einzigen Änderungen die Weights des neuen Perceptrons sind und dieses mit dem Output des Netzes direkt verbunden ist [ML90]. Ein weiteres Prinzip ist das Direct Cascade Learning. Hierbei gibt es zwischen den trainierten Perceptrons keinerlei Nachbearbeitung der Daten, sondern der Output des Vorherigen wird als

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Input des Nächsten genutzt, wie es bei Cascade QEF und Cascade LLM der Fall ist [LR92b], [LR92a].

TF wurde bereits mit traditionellen Lernmethoden durchgeführt und neben diesen eher selten mit Cascade Learning. Ein Beispiel, bei dem es durchgeführt wurde ist Cascade Transfer Learning (CTL) [Mar19], welches effektiv CasCor [ML90] mit TF verband. Dies hatte den Effekt, dass die Performance von CTL etwas schlechter als fine-tuning TF war, aber der große Unterschied darin bestand, dass CTL deutlich weniger Speicherplatz benötigte [Mar19]. Es gibt neben CasCor allerdings noch weitere Cascade Learning Algorithmen, die bisher noch nicht für Transfer Learning benutzt worden sind, wie die, die Direct Cascade Learning vollziehen.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Chapter 2

Methodik

2.1 Transferlernen

Transferlernen (TF) ist das Prinzip des Lernens über einer Eselsbrücke. Es gibt mehrere Varianten, wie TF verwendet werden kann: Dies sind Task-Wechsel und Domain-Wechsel. Nur wenn keine Option davon genutzt wird, wird nicht von TF gesprochen. Hier wird nur der Domain-Wechsel vorgestellt werden, da nur dieser benutzt wird. Ein Domain-Wechsel ist hier der Wechsel zwischen zwei verschiedenen Datensätzen, während diegleiche Netzart genutzt wird. Dies wird Transductive Transferlernen[JY10] genannt. Das Wissen vom ersten Datensatz wird auf den Zweiten übertragen. Der erste Datensatz ist dabei die Source, der Zweite das Target. Es gibt dabei drei Stellschrauben, bei denen nicht klar ist, was besser ist: What, How und When to Transfer [JY10]. Da es sowohl Klassifikation- als auch Regressionnetze ausgetestet werden, werden jeweils zwei Source- und Targetdatensätze benötigt. Für Klassifikation wird die Source der Modified National Institute of Standards and Technology [LeC+] (MNIST) Datensatzes und der Street View House Numbers (SVHN) [Net+11] der Targetdatensatz sein. Beide müssen für das TF ein wenig verändert werden. Der MNIST wird von 28x28 Pixel auf 32x32 erweitert, während der SVHN von farbig auf schwarz-weiß verändert wird. Dies ist notwendig, da beide Datensätze als Input denselben Shape, also diegleichen Dimensionalitäten vorweisen, haben müssen. Bei der Regression ist der Sourcedatensatz der Boston Housing Prices (Bost) [Har+97] und der Targetdatensatz der California Housing Prices (Cali) [Nug18].

Beide Datensätze müssen stark reduziert werden. Von den acht beziehungsweise dreizehn Spalten bleiben nur drei übrig. Dies hat den Grund, dass nur Spalten als sinnvoll geachtet werden, die ein passendes Gegenüber haben. Der Bost-Datensatz hat allerdings ein ethnisches Problem, da dieser eine Spalte enthält, die diskriminierend ist. Diese wird entfernt. Die einzigen Spalten des Bost-Datensatzes, die übrig bleiben, sind: RM, AGE und LSTAT. RM ist die durchschnittliche Zimmeranzahl pro Wohnung, AGE ist die Anzahl der

Häuser, die vor 1940 bewohnt wurden und LSTAT ist der prozentuale Anteil der Bevölkerung mit niedrigerem Status. Der Datensatz Cali behält nur die Spalten MedInc und HouseAge. MedInc ist das durchschnittliche Einkommen des Häuserblocks und HouseAge das durchschnittliche Alter. Aus den Spalten AveRooms und Households wird die durchschnittliche Anzahl von Zimmer pro Haushalt berechnet. AveRooms ist dabei die durchschnittliche Anzahl an Räumen innerhalb eines Häuserblocks, während Households die Anzahl der Haushalte innerhalb des Häuserblocks ist. Dadurch ist die berechnete Spalte zu der RM-Spalte von Bost passend. LSTAT und MedInc sind wahrscheinlich abhängig, da es vermutet wird, dass diejenigen Menschen, die einen niedrigeren Status vorweisen, weniger Einnahmen haben. Deshalb dürfte es über diese beiden Spalten möglich sein TF zu nutzen. Allerdings sind sie zueinander antiproportional, weshalb die LSTAT Spalte invertiert wird, damit es zur Proportionalität kommt. Komplexer ist die Berechnung des Alters der Häuser, da AGE nur die Anzahl der Häuser, die vor 1940 gebaut wurden, beeinhaltet, aber HouseAge das durchschnittliche Alter des Häuserblocks ist. Die Maximalanzahl der betrachteten Häuser im Bost-Datensatz ist einhundert und das Alter der Häuser vor 1940 ist 85, wenn man auf 2025 rechnet. Dadurch kann AGE auf die Art von HouseAge mit folgender Formel umgerechnet werden:

$$\frac{AGE * 85}{Maximalanzahl} \tag{2.1}$$

Dadurch sind alle Source- und Targetdatensätze zueinander kompatibel. Damit ist ausreichend geklärt, mit was TF verwendet wird.

Die nächste Frage, die geklärt werden muss, ist das How to transfer. Dies wird jeweils ohne Veränderung der Weights der Netze gemacht. Es wird das neuronale Netz zuerst auf dem Sourcedatensatz trainiert und dann ohne irgendetwas zu tun auf den Targetdatensatz gewechselt, welcher auf demselbem Netz oder einem gleichen Netz wie zuvor ist. Wenn es dasselbe Netz ist, dann verändert sich nur aus welchem Datensatz der Input kommt, was bei Deep Cascade ist. Hingegen wird bei dem gleichen Netz der Input immer vergrößert und das TF passiert über diese Vergrößerung, was bei Direct Cascade ist.

Wann TF sinnvoll ist, ist nicht klar, weshalb es mal mit früherem und späteren TF probiert wird.

2.2 Kaskadierung

Hier wird erklärt, was ein Kaskadennetzwerk ist und welche Besonderheiten es dabei gibt. Ein Kaskadennetzwerk ist ein Netzwerk, welches in Kaskaden, also Schrittweise, aufgebaut wird. Während bei einem klassischen Netz vorher festgelegt wird, wieviele und welche Layer und Unternetzwerke dieses haben wird, ist es bei einem Kaskadennetzwerk nicht so. Ein solches Netzwerk wird erst während dem Training aufgebaut und immer erweitert. Deshalb werden, im Gegensatz zu den klassischen Netzen, nur der aktuelle neue Teil trainiert, während der Rest nicht mehr verändert wird. Die vorher gelernten Layer

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

und Unternetzwerke werden nach dem Training gefreezt. Dadurch werden die Gewichte der gefreezten Layer und Unternetzwerke nicht mehr verändert. Die Kaskadennetzwerke lernen dadurch das, was zwischen den Layern und Unternetzwerken gelernt wird, nicht und sind deshalb etwas schlechter als die klassischen Netzwerke bei gleich vielen Daten. Aber, weil immer nur das Aktuelle gelernt wird, sollten Kaskadennetzwerke im Training sehr viel schneller sein. Dies liegt daran, dass die Gewichte der vorherigen Layer kein sich verändertes Bild von den nachfolgenden Gewichten in jeder Epoche haben und sich nicht aktualisieren müssen.

2.2.1 Deep Cascade

Hier wird die Variante des Deep Cascade vorgestellt. Die Deep Cascade Netze werden iterativ während dem Training aufgebaut. Es bleibt dabei ein einziges Netz. Es wird zuerst definiert, welcher Optimizer und welcher Loss in dem Netz genutzt wird.

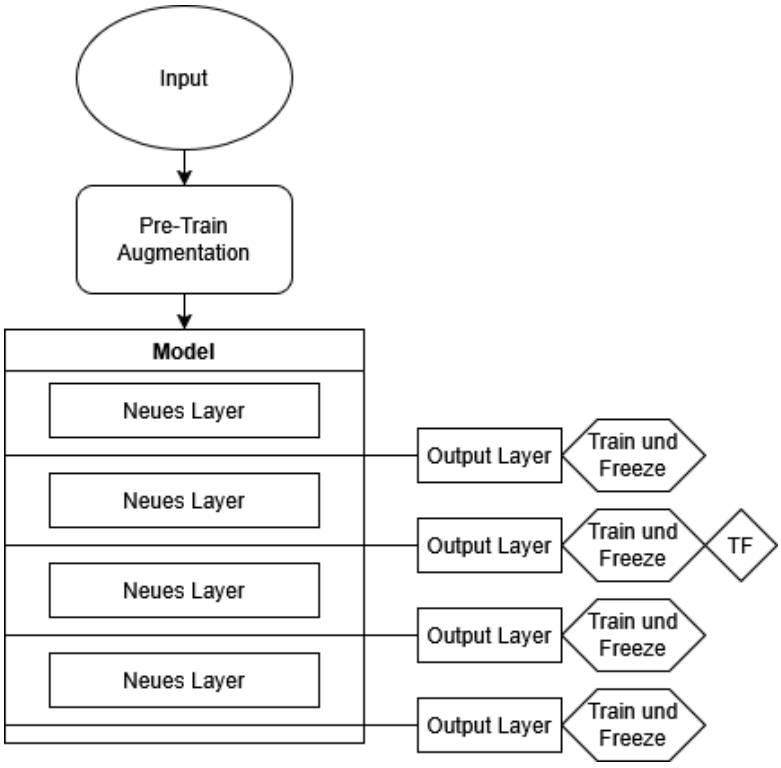

Figure 2.1: Hier ist zu sehen, wie Deep Cascade Netzwerke erstellt und trainiert werden. Das Modell selbst enthält mehrere Layer, die nacheinander trainiert werden. Außen sind mehrere Outputlayer, da sie nur für das aktuelle Training benötigt werden, aber innerhalb des Netzes stören würden.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Sobald dies beides gemacht wurde, wird im Netz das erste Layer definiert. Dieses wird ergänzt durch ein Output Layer und dann trainiert. Wenn das Training beendet wird, wird das Output Layer gelöscht und ein neues Layer hinzugefügt, wie es in Figure 2.1 gezeigt wird. Zudem wird das gerade trainierte Layer gefreezt, damit dieses keine weiteren Aktualisierungen mehr bekommt. Dann wiederholt sich das Training, das Löschen, das Freezing und weitere Hinzufügen von Layern. An einer beliebigen Stelle kann TF gemacht werden, indem, statt in der Trainingsphase den Sourcedatensatz zu nutzen, der Targetdatensatz genutzt wird.

2.2.2 Direct Cascade

Hier wird die Kaskadierungsvariante des Direct Cascade vorgestellt. Das Netzwerk ist hier vorher vollständig und besteht aus einem einzigen Hidden Layer und einem Output Layer. Es wird dasselbe Netz mehrfach trainiert und währenddessen wird das Wissen zwischen diesen Netzen weitergegeben.

Es beginnt, wie in Figure 2.2 gezeigt mit dem präpariertem Sourcedatensatz, der in die Instanz des Netzes hineingegeben wird. Damit wird diese Instanz trainiert und sobald dies beendet ist, wird einmal das fixe Netzwerk angewendet. Das Ergebnis davon ist die Prediction. Diese wird mit dem Input desselben Netzes verbunden und es entsteht der Augmented Vector. Darauf, wie dieser Augmented Vector genau entsteht, wird später nochmal eingegangen, da es bei jedem Direct Cascade Netzwerk ein wenig anders ist. Der Augmented Vector wird in die nächste Instanz von dem Netzwerk als input hineingegeben. Die Netzinstanzen, das Training, die Prediction und Berechnung der Augmented Vectors wird beliebig häufig wiederholt. Dabei lernt das Netzwerk über den Augmented Vector das Wissen der vorherigen Netzwerke mit, da dieses als Prediction dort mit vorkommt.

TF kann nun jederzeit im Trainingsschritt gemacht werden, indem dort der Targetdatensatz statt der Sourcedatensatz als Input genommen wird. Dabei können beliebig viele Netzwerke vor und nach TF genutzt werden. Der einzige Unterschied ist der, dass der Augmented Vector für jedes Netzwerk ein wenig größer wird, da dieser sowohl das Wissen von jedem bisherigen Netzwerk als auch die Ursprungsdaten enthält.

Dabei muss hier in der Implementation bereits von Anfang an sowohl der Sourcedatensatz als auch der Targetdatensatz in das feste Netz hineingegeben werden, um die Prediction auf dem Targetdatensatz von der Trainingsphase des Sourcedatensatzes im Augmented Vector zu integrieren, damit die Netzwerke, die auf dem Sourcedatensatz gelernt haben, während dem Training auf dem Targetdatensatz Berücksichtigung finden.

2.3 Setup

Alle Test wurden auf einem Erazer Gaming Notebook P15601 unter Windows 10 durchgeführt. Die Neuronalen Netze laufen dabei ausschließlich auf der CPU

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

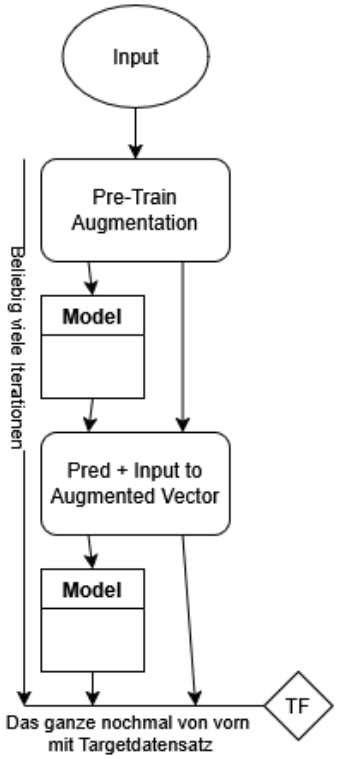

Figure 2.2: Hier zu sehen ist das Direct Cascade Verfahren. Dieses benutzt mehrere Netzwerke (hier Models), die alle nur sehr wenige Hidden Layer haben, meistens nur eines. Danach wird dieses einmal ohne Training angewandt und dessen Ergebnis mit dem Input verknüpft. Diese Verknüpfung ist der neue Input des nächsten Netzwerkes. Dadurch lernt das Verfahren zwischen den Netzwerken.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

und wurden nur trainiert, während der Rechner am Stromnetz angeschlossen war. Dieser Rechner hat einen intel Core i5 der neunten Generation mit 4 Kernen auf 8 logischen Prozessoren. Die Betriebsgeschwindigkeit liegt bei 2,4-5,1 GHz und die RAM-Größe liegt bei 15,8 GB bei einer Geschwindigkeit von 2667 MHz.

Es wurde mit PyCharm und der library Keras programmiert. Die Texte sind mit BibTex erstellt worden und die Plots mit der MatPlotLib library.

Dabei sind MNIST und Bost die Sourcedaten und SVHN und Cali die Targetdaten. Jeder Targetdatensatz wird händisch verkleinert, da es darum geht, nicht genügend Daten für sie allein zu haben und deshalb eine andere Methode genutzt werden muss.

Keiner dieser Datensätze wird in der normalisierten Form als Input verwendet.

2.4 Metrik

Es wurden drei Metriken erstellt. Die Accuracy- (ACCM), Loss- (LM) und MAE-Metrik (MAEM). MAE heißt dabei Mean Absolute Error. Alle drei Metriken sind für Early Stopping und entscheiden, wieviele Epochen genutzt werden. Die Accuracy-Metrik bricht immer dann ab, wenn die Validation-Accuracy mindestens um 10% schlechter ist als die Trainingsaccuracy, da dann in dem Netzwerk Overfitting herrscht.

Die Loss- und die MAE-Metrik brechen beide dann ab, wenn der Validation-Wert der aktuellen Epoche schlechter ist als in der Epoche davor. Dies hat zur Folge, dass die Netze in lokale Minima hineinlaufen und nicht wieder herauskommen. Dabei unterliegt die Anzahl der Netze für das Direct Cascade keiner Metrik.

2.5 Liste der Tests

Liste aller hier vorkommenden Netzen mit ihren Kürzeln:

1. ConvMaxPool (CMP)
2. 1DConv (1DC)
3. 2DConv (2DC)
4. ClassOneDense (COD)
5. RegressionTwo (Regr2)
6. OneLayer (1Lay)

Davon sind ConvMaxPool und RegressionTwo Deep Cascade Netzwerke, während alle anderen Direct Cascade Netzwerke sind. Ebenso sind nur RegressionTwo und OneLayer Regressionsnetze, während der Rest Klassifikationsnetze sind.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Alle Netze werden mit dem Adam-Optimizer mit der Lernrate 1e-3 gelernt. Klassifikationsnetze haben als Loss den CategoricalCrossEntropy und Softmax als Aktivierungsfunktion, während die Regressionsnetze MeanSquaredError und Linear als Aktivierungsfunktion vorweisen.

Mit allen Direct Cascade Netzwerken wurden zusätzlich Early Stopping Metriken durchgeführt mit MAEM, LM und ACCM.

Für alle Klassifikationsnetze gilt, dass sie mit fünf verschiedenen Größen des Targetdatensatzes trainiert wurden. Die Ausnahme ist das 2DC-Netzwerk, welches nur mit sehr wenigen Source- und Targetdaten trainiert werden kann, da es technisch auf derselben Hardware mit mehr Daten unmöglich ist.

Bei den Regressionsnetzen wird jeweils einmal mit vielen und wenigen Targetdaten trainiert.

Es wurde mit allen Netzwerken ein Vergleich sowohl zwischen mit TF und ohne als auch zwischen ohne TF und Kompletten angefertigt. Komplett heißt hier, dass es ein Netzwerk ohne TF und ohne Kaskadierung ist und dieses deshalb in einem komplett trainiert wird.

Mit allen Netzwerken wurde der Zeitpunkt für das TF frei ausgetestet.

Alle Direct Cascade Netzwerke haben jeweils nur ein Hidden Layer. In manchen Fällen sind sie noch mit einem Hilfslayer, um den Wechsel zwischen Filterlayern und Linearlayern zu bewerkstelligen.

Für alle Netzwerke wurde derselbe Seed für die Initialisierung der Weights genutzt.

In Tabelle 2.1 sind die Tests bezüglich Klassifikation und in Tabelle 2.2 die für die Regression. In beiden Tabellen sind die Tests, die sich mit der Zeitdauer befassen mit der Endung Time. Dabei gilt, dass CasTF Kaskadierung mit TF, Cas allein Kaskadierung ohne TF und Comp bedeutet, dass es weder TF noch Kaskadierung gab. ACCM, LM und MAEM sind die Tests bezüglich der Early-Stopping Metriken. Vor dem ersten Schrägstrich steht, wann TF gemacht wurde, welches mit TF im Eintrag gekennzeichnet ist. Wenn kein TF gemacht wurde, ist dieser erste Bereich nicht existent. Dahinter steht die Datenmenge des Targetdatensatzes und danach die Menge an Epochen pro Trainingsiteration. Wenn es noch etwas viertes gibt, dann zeigt dieses an, wieviele Epochen in Zehnern es insgesamt gab.

Eine Referenz zu einer dieser Listen ist CMP:TF0/732/10. Diese bedeutet, dass es um den Test mit der Kennung TF0/732/10 des CMP-Netzwerkes geht. Bei diesem gibt es die Besonderheit, dass TF nach dem ersten Layer gemacht wird, dieses Layer jedoch im Gegensatz zum restlichen Netzwerk nur mit einer Epoche trainiert wird. Selbiges gilt für Regr2:TF0/240/25.

Die Tests mit der Endung ts sind diejenigen, die einen explizit großen Testdatensatz haben.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

CMP	**COD**	**1DC**	**2DC**
TF0/732/10	CasTFTime	CasTFTime	CasTFTime
TF1/732/10	CasTime	CasTime	CasTime
TF2/732/10	CompTime	CompTime	CompTime
TF3/732/10	TF2/732/10	TF2/732/10	TF2/732/10
TF4/732/10	TF2/7k/10	TF2/7k/10	
TF5/732/10	TF2/21k/10	TF2/21k/10	
732/10	TF2/36k/10	TF2/36k/10	
CasTFTime	TF2/51k/10	TF2/51k/10	
CasTime	TF10/732/10/30	TF10/732/10/30	
CompTime	732/10/30	732/10/30	
TF2/7k/10	Comp/732//30	Comp/732//30	
TF2/21k/10	ACCM/732/10	ACCM/732/10	
TF2/36k/10	LM/732/10	LM/732/10	
TF2/51k/10			

Table 2.1: Liste aller Klassifikationstests

Regr2	**1Lay**
TF0/240/25	CasTFTime
TF1/240/25	CasTime
TF4/240/25	CompTime
CasTFTime	TF11/8k/10/21
CasTime	8k/10/11
CompTime	Comp/8k//8
TF4/8k/10/8	TF11/240/10/21
8k/10/8	240/10/11
Comp/8k//8	Comp/240//8
TF4/240/10/8	MAEM/240/10
240/10/8	LM/240/10
Comp/240//8	TF4/206/10/8/ts
TF4/206/10/8/ts	206/10/8/ts
206/10/8/ts	Comp/206//8/ts
Comp/206//8/ts	

Table 2.2: Liste aller Regressionstests

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Chapter 3

Allgemeine Resultate

3.1 Ploterklärung

In diesem Kapitel werden alle Arten der Plots einmal vorgestellt und auf alle Eigenheiten eingegangen, damit diese verstanden werden.

Es wird hier sowohl auf die Achsenbeschriftung als auch auf die Texte innerhalb der Plots eingegangen.

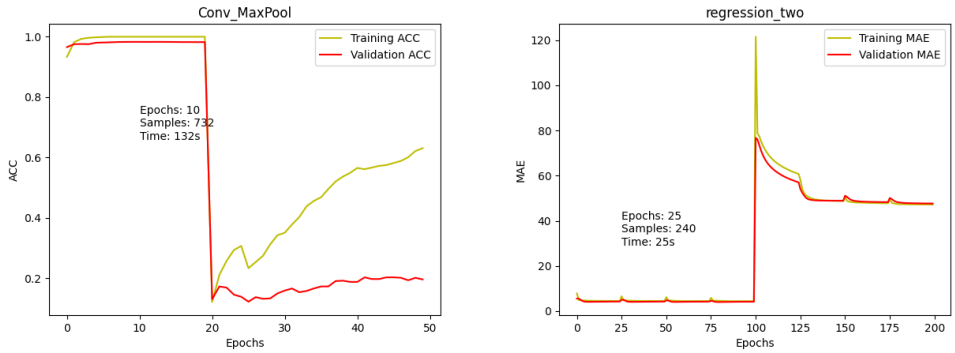

Figure 3.1: Es sind hier die Plots von zwei Deep Cascade Netzen. Links für die Klassifikation und rechts für die Regression. Zu sehen sind die Tests: CMP:TF2/732/10 und Regr2:TF4/240/25. Beides dient als Beispielplot dafür, wie diese mit TF für gewöhnlich aussehen.

Dazu wird Figure 3.1 betrachtet. In beiden Teilen stehen drei Zeilen Text auf die nun einzelnd eingegangen wird. Die Erste sagt aus, wieviele Epochen pro Layer oder Netzwerk trainiert worden sind. Die Zweite beschreibt wieviele Datensamples des Trainingssets des Targetdatensatzes im Training nach TF genutzt worden sind und die dritte Zeile zeigt die gesamte Trainingsdauer in Sekunden an.

Wenn es um die Accuracy geht, was bei Klassifikation der Fall ist, dann steht ACC auf der senkrechten Achse und beim Funktionsnamen dabei. Die senkrechte Achse ist dann bei 100%, wenn sie bei 1 ist. In Figure 3.1 ist links ein Beispielplot für diesen Fall.

Für die Regression, geht es um den MAE. Dies steht wiederum in den Namen der Funktionen und der senkrechten Achse. Diese Achse ist in 1000$ pro Einheit. Dabei ist es besser, je geringer der Wert ist.

3.2 ConvMaxPool

Anhand des ConvMaxPool-Netzwerks werden alle allgemeinen Resultate und Auffälligkeiten beschrieben. Dies ist ein Deep Cascade Classification Netzwerk und wird deshalb iterativ aufgebaut. Das Netz ist ein Convolution-Network mit Padding, sodass die Dimensionen während der Convolution-Layer nicht verringert werden. Es wird die Aktivierungsfunktion relu genutzt.

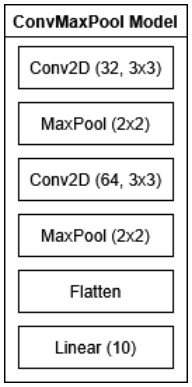

Figure 3.2: Diese Layer in genau der Reihenfolge, wie hier von oben nach unten, stecken hinter dem CMP-Netzwerk.

Alle Layer des ConvMaxPool Netzwerks sind in Figure 3.2 in korrekter Reihenfolge zu sehen. Dabei ist die erste Zahl eines Convolution Layer die Anzahl der genutzten Filter, während das folgende Tuple die Kerngröße beschreibt. Ebenfalls steht die Kerngröße bei den MaxPool Layern dort. Das Flatten- und das Linear-Layer sind der Output-Block. Das Linear-Layer benötigt zehn Nodes, da es zehn Klassen gibt. Jedes Hidden Layer wird mit zehn Epochen trainiert. Es gibt keine Early-Stopping Metrik und es wird derselbe Seed für alle Tests genutzt.

3.2.1 Veränderungen bei TF

Hier wird etwas sehr offensichtliches betrachtet. Dies passiert jedes Mal, wenn TF verwendet wird. Der Graph, der die Trainings- und Validationdaten nutzt,

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

hat immer einen Einbruch in der Performanz an der Stelle an der TF gemacht wird. Dies ist in der Figure 3.3 deutlich bei Epoche zwanzig zu sehen.

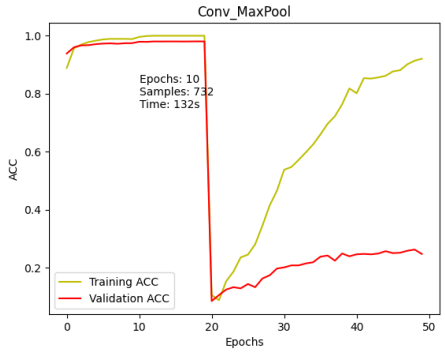

Figure 3.3: Hier zu sehen ist CMP:TF2/732/10. Der Hauptaugenmerk liegt hier bei Epoche 20, denn zu diesem Zeitpunkt wurde TF angewendet. Es kommt zum Einbruch der Performanz.

Dieser Einbruch passiert jedes Mal nach TF. Dies liegt daran, dass das Netz bisher die Targetdaten noch nie gesehen hat und bisher auf eine andere Domain mit dem Sourcedatensatz trainiert hat. Das Netz kennt nur das Wissen aus dem Sourcedatensatz und kann nur dieses anwenden. Wenn man aber das Testset, welches nur über die Targetdaten geht auf das ganze Netzwerk betrachtet, kommt Figure 3.4 heraus.

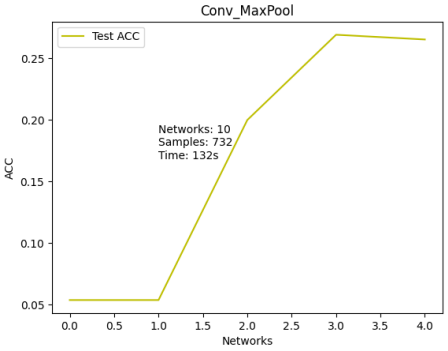

Figure 3.4: Dies ist der Testdatenplot von CMP:TF2/732/10. Dieser enthält die Targetdaten, die das Netz nicht während dem Training sieht. Immer nachdem ein Layer fertig trainiert wurde, werden einmal die Testdaten evaluiert. Deshalb ist der Punkt an dem TF gemacht wird, hier bei 2 Networks. Performanz wird zu dem Zeitpunkt besser.

Der Wechsel ist hierbei bei Netzwerk 2. Es ist eindeutig zu erkennen, dass

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

es nach TF besser wird. Dies hat den Grund, dass das Netzwerk ab diesem Zeitpunkt auf den Trainingsdaten trainiert, die zum Testdatenset passen, da dieses seine Daten nur aus dem Targetdatensatz bezieht.

3.2.2 Overfitting auf Sourcedatensatz

Wenn unterschiedlich lang auf dem Sourcedatensatz trainiert wird, fällt auf, dass das Netz unterschiedlich gut auf dem Targetdatensatz ist. Da es sowieso ausgestestet werden muss, wann TF genutzt wird, wird nun das ConvMaxPool-Netzwerk genommen und nach jedem Layer TF angewandt. Das Ergebnis davon ist in Figure 3.5 zu sehen.

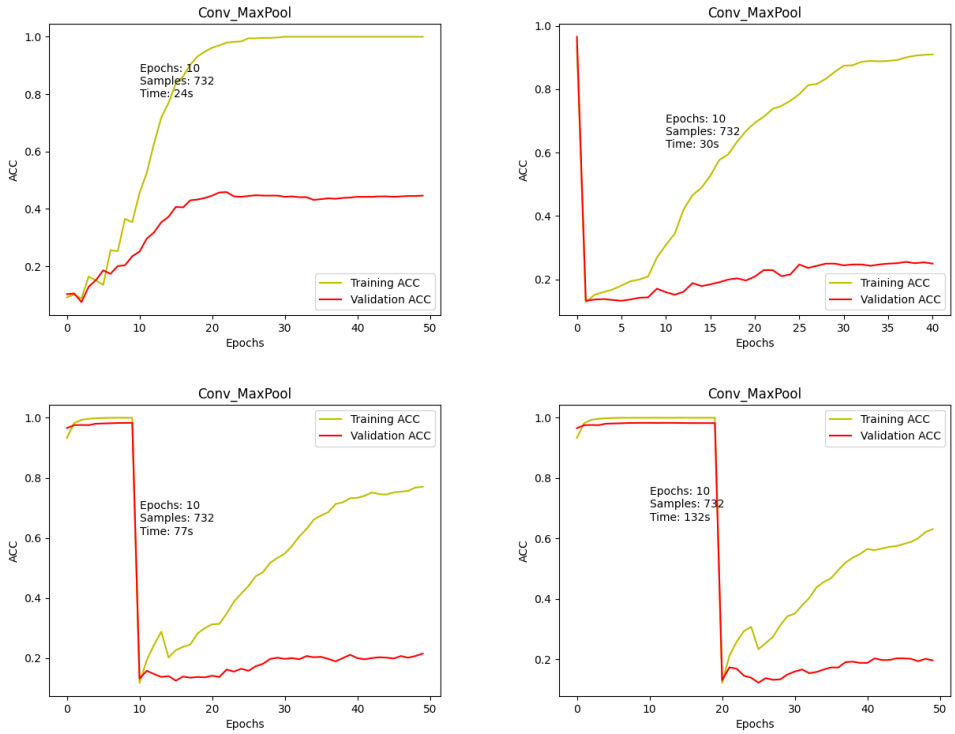

Figure 3.5: Hier sind von links oben nach rechts unten die Plots folgender Tests zu sehen: CMP:732/10, CMP:TF0/732/10, CMP:TF1/732/10, CMP:TF2/732/10. Alle Plots haben hier dasselbe Problem damit, dass der Unterschied zwischen der Trainings-Accuracy und der Validation-Accuracy immer größer wird.

Auffällig ist es, dass hier die beste Performanz ohne TF ist. Bereits nach nur einer Epoche im ersten Layer, welches auf dem Sourcedatensatz trainiert wird, bricht die Accuracy ein. Dies zeigt, dass TF bei Klassifikation und Deep Cascade Netzwerken sinnfrei ist. Das Trainingsset der Trainingsdaten kommt bei TF nie auch nur annähernd an den Bereich heran, in dem es ohne TF ist.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Daraus folgt, dass es bereits zu Overfitting auf dem Sourcedatensatz gekommen ist. Dadurch kann nicht mehr so gut auf dem Targetdatensatz gelernt werden. Dieses Overfitting passiert sogar bereits, wenn nur eine Epoche auf dem Sourcedatensatz gelernt wird, was die letzte Graphik von Figure 3.4 zeigt. Ebenso ist es offensichtlich, dass es bei jedem Graph zu Overfitting auf dem Trainingsset des Targetdatensatzes kam, da dieser um 60% höhere Accuracy als das Validationset und dem Testdatenset vorweist.

Bei einem Regressionsnetzwerk, wie dem Deep Cascade Netzwerk RegressionTwo kommt es, wie in Figure 3.6 zu sehen, nicht so schnell zu Overfitting. Es kommt dazu weder auf dem Sourcedatensatz noch auf dem Targetdatensatz.

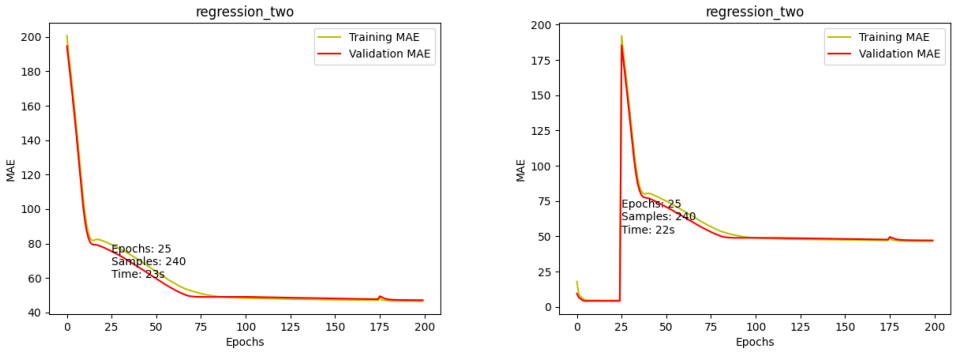

Figure 3.6: Hier ist links der Test Regr2:240/25 und rechts Regr2:TF1/240/25 zu sehen. Der MAE zwischen Trainingsset und Validationset hat kaum einen Unterschied. Hier ist kein Overfitting.

Dieses Overfitting-Problem hat nur die Klassifikation. Dies muss an der Loss-Function oder der Activation-Function, die für Klassifikation benutzt wird, liegen.

3.3 Zeitnahme

Hier werden alle Netze bearbeitet und überprüft wieviel Zeit sie für ihr Training benötigten. Diese wird in jedem Plot angezeigt. Sie ist generell stark abhängig davon, wieviele Datensamples genutzt werden und wieviele maximale Epochen erlaubt sind. Deshalb werden hier jeweils gleich viele Datensamples für die Klassifikationsnetze und Regressionsnetze verwendet, sowie eine gleiche Gesamtepochenanzahl.

Dazu wird jedes Mal der kleinste Targetdatensatz und der größte Sourcedatensatz genutzt, bis auf das 2DConv-Netzwerk, wenn TF verwendet wird. Solange nicht TF verwendet wird, wird nur der kleinste Targetdatensatz genommen. Jedes Klassifikationsnetz wird auf eine Gesamtanzahl von 40 Epochen trainiert. Wenn TF gemacht wird, dann nach 20 Epochen. Bei Regressionsnetzen wird auf 80 Epochen trainiert. TF wird nach 30 Epochen gemacht.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Tatsächliche Graphen werden hier nicht verwendet, da es zuviele sind und die meisten an anderen Stellen in ähnlicher Form bereits vorkommen. Um diese Graphen zu kontrollieren, schaut bitte auf das GitHub Repo unter `https://github.com/Lirras/BA_EvalTF_DDCN/tree/main/Plots/ba_plots/timing`.

Es wird hier keine Early-Stopping Metrik genutzt. Dabei werden alle Klassifikationsnetze mit denselben Inputdaten gespeist, ebenso wie bei allen Regressionsnetze, was zur besseren Vergleichbarkeit jeweils untereinander führt. Die jeweiligen Netzversionen zwischen Cascade TF, Cascade und Complete haben dieselben Layer, sowie diegleiche Anzahl.

Hier nun die Tabelle mit allen Zeiten der Netze in einer vergleichbaren Variante:

Netzwerk	**Cascade TF**	**Cascade**	**Complete**
ConvMaxPool	78	25	20
1DConv	207	34	30
2DConv	23	24	40
ClassOneDense	79	28	13
RegressionTwo	11	12	17
OneLayer	16	18	11

Table 3.1: Dies ist der Zeitdauervergleich zwischen den jeweiligen Netzen und ihren Varianten. Die Angaben sind in Sekunden.

In Table 3.1 ist die Spalte Cascade TF, die Kaskadennetzwerke mit TF beeinhaltet. Die Spalte Cascade ist diejenige, in der die Netze nur auf dem Targetdatensatz gelernt haben, aber Kaskadennetzwerke sind und die Spalte Complete sind die Lernzeiten der Netze, die weder TF machen noch Kaskadennetzwerke sind, sondern dessen Layer vor dem Training bereits feststanden und komplett in einem auf dem Targetdatensatz gelernt wurden.

Auffällig ist, dass die Regressionsnetze keine große Zeitveränderung haben. Mitunter benötigt mit TF weniger Zeit als ohne, was daran liegt, dass der Targetdatensatz der Regression ein wenig größer ist als der Sourcedatensatz. Beide sind allerdings mit etwas über 200 Datensamples sehr klein.

Ebenso brauchen alle Klassifikationsnetze länger mit TF als ohne, was daran liegt, dass sie zuerst auf dem Sourcedatensatz trainieren, welcher mit 48000 Trainingsdaten recht groß ausfällt, während die Anderen direkt auf dem kleinen Targetdatensatz, der mit 732 Datensamples klein ist, trainieren.

Da Cascade TF mit dem Sourcedatensatz ist und die anderen Netze nur auf dem Targetdatensatz arbeiten, sind letztere meist kürzer. Allerdings benötigen die Complete Netzwerke, die ohne Kaskadierung sind, noch weniger Zeit. Dies wird an der Implementierung der Netze liegen, da bei jedem Kaskadennetzwerk das Outputlayer in jeder Kaskade mit berechnet wird, während bei den Complete Netzwerken nur ein einziges Outputlayer existiert. Des Weiteren werden keine extra Predictions und keine Berechnung der Augmented Vectors gemacht.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Zudem sind die genutzten Netzwerke nur mäßig groß, sowie der betrachtete Targetdatensatz so klein, dass nicht das Training die meiste Zeit benötigt, sondern die Berechnungen drum herum.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Chapter 4

Klassifikation

Hier werden kurz die Direct Cascade Netze für die Klassifikation vorgestellt. Die Besonderheit eines solchen Netzwerkes ist es, dass immer nur ein einziges Hiddenlayer existiert und die Netze so iteriert werden, dass sie das Wissen der Vorherigen mitnehmen.

Name	**Hiddenlayer**	**N/F**	**Aktivierung**	**Inputdim**
COD	Linear	512	Relu	1
1DC	1DConv	32	Relu	1
2DC	2DConv	32	Relu	2

Table 4.1: Hier sind alle Direct Cascade Netzwerke für die Klassifikation mit den groben internen Daten. Hiddenlayer ist was für ein solches wurde genutzt. N/F ist Nodes/Filter bezüglich des Layers. Mit Aktivierung ist die Aktivierungsfunktion im Hiddenlayer gemeint und Inputdim ist in wievielen Dimensionen die Inputdaten vorliegen.

Dabei ist der Input die Dimension in der die Bilddaten vorliegen und nur in der ersten Zeile sind es Nodes, sonst sind es die Filter. Bei beiden Convolution-Netzen wird ein Kern der Größe 3 beziehungsweise 3x3 verwendet mit einem solchen Padding, dass die Größe der Daten dabei nicht verändert wird. Bei diesen Netzen wird als Batch jeweils 128 Datensamples genutzt.

4.1 Größe des Targetdatensatzes

Hier wird auf die Änderungen der Performanz der Netze eingegangen, die dadurch entstehen, wenn der Targetdatensatz unterschiedliche Größen hat. Die Vermutung ist, dass es schlechter wird, je weniger Daten vorhanden sind.

Um Vergleiche zu haben, die zueinander passen, werden wieder insgesamt 40 Epochen trainiert und nach zwanzig TF vollzogen. Dies wird mit den Netzwerken CMP, 1DC und COD durchgeführt. Hier wird auch auf den Testdatensatz eingegangen.

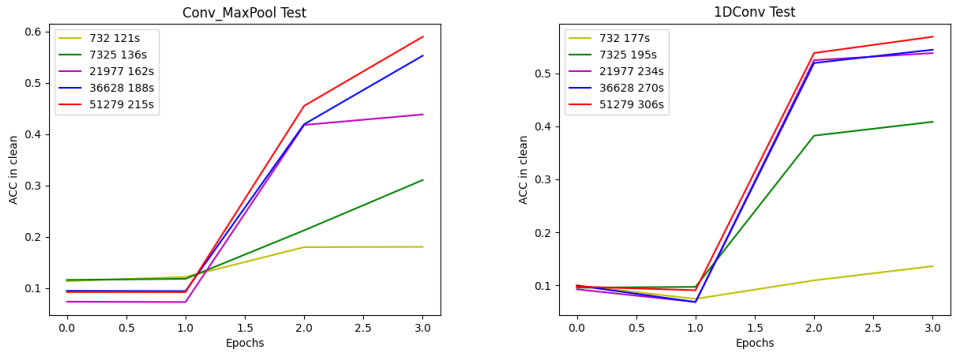

Figure 4.1: Hier ist die Veränderung der Accuracy bei Convolution Netzen zu sehen, wenn die Größe des Targetdatensatzes verändert wird. Es sind jeweils die Plots über die Testdaten des Targetdatensatzes, wobei links CMP und rechts 1DC ist. Dahinter sind folgende Tests: CMP: und 1DC:TF2/732/10, 1DC:TF2/7k/10, 1DC:TF2/21k/10, 1DC:TF2/36k/10, 1DC:TF2/51k/10 jeweils in den Farben gelb, grün, lila, blau und rot.

In Figure 4.1 sieht man die Testdatenläufe bezüglich der Menge der Trainingsdaten. Dabei ist die erste Zahl in der Legende die Menge und die Zweite die Dauer. Dabei bezieht die Menge sich nur auf den Targetdatensatz. Es wird deutlich, dass es länger dauert, je mehr Daten vorhanden sind. Ebenso bestätigt sich die Vermutung, dass es auch bei Kaskadennetzwerken mit TF besser wird, je mehr Daten vorhanden sind. Ebenfalls zeigt sich, dass Deep Cascade etwas besser ist als Direct Cascade. Dies dürfte daran liegen, dass Direct Cascade Netzwerke sind, die nur ein Hidden Layer haben, während es bei Deep Cascade mehrere sind.

In Figure 4.2 ist das Deep Cascade Netzwerk, welches als Hidden Layer ausschließlich Linearlayer hat. Auffällig ist, dass dieses bei egal wievielen Datensamples immer schlechter abschneidet als die anderen beiden Netzwerke, die Convolutionlayer besitzen. Dies liegt daran, dass dieses Netzwerk die wichtigen Bilderkennungsfeatures nicht ausreichend herausfiltern kann, weil die Daten zu komplex und nicht linear sind.

Generell ist es bei allen Testplots so, dass sie nie eine annehmbar hohe Accuracy besitzen und das auch dann nicht, wenn es genügend Daten gibt, damit auf dem Targetdatensatz direkt gelernt werden könnte. Wenn dies mit dem Deep Cascade Netzwerk gemacht wird, kommt dabei eine Accuracy von etwa 70% heraus, was deutlich besser als jedes TF Netzwerk ist.

4.2 Bilddimensionalität

Bei den beiden Convolution Direct Cascade Netzwerken ist der einzige Unterschied, dass sie die Bilder in ein- beziehungsweise in zweidimensionaler Form

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

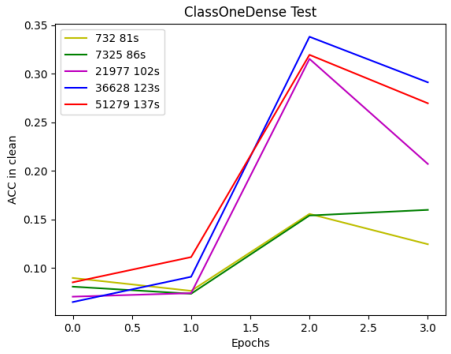

Figure 4.2: Hier ist die Veränderung der Test-Accuracy bei Linearen Netzwerk zu sehen, wenn die Größe des Targetdatensatzes verändert wird. Es sind die Tests COD:TF2/732/10, COD:TF2/7k/10, COD:TF2/21k/10, COD:TF2/36k/10 und COD:TF2/51k/10 in den Farben gelb, grün, lila, blau und rot.

sehen. Dabei fällt aber auf, dass es im zweidimensionalen Fall etwas besser ist. Dies liegt daran, dass das eindimensionale Netz in dem Filterlayer nur die Daten direkt rechts und links mit einbezieht. Das zweidimensionale Netzwerk hingegen nutzt bei der Operation jenes Layers nicht nur die direkt rechts und links, sondern auch die Daten, die oben und unten angrenzend sind, sowie die Daten, die in jede Richtung schräg vorkommen. Die Verbesserung ist aber nur minimal, wie in Figure 4.3 zu sehen.

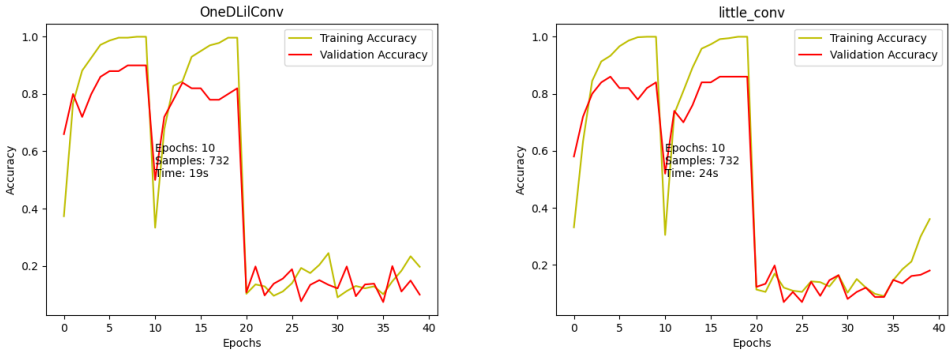

Figure 4.3: Hier ist links der Test 1DC:TF2/732/10 und rechts 2DC:TF2/732/10. Es geht hier darum einen möglichst geringen Unterschied zwischen den beiden zu haben, damit nur noch 1DC betrachtet werden muss.

Da das zweidimensionale Netzwerk mit nicht so vielen Daten genutzt werden kann, hat es hier eine sehr viel kürzere Zeit. Es kann deshalb nicht genutzt werden, da die Berechnung des Augmented Vectors zu Speicherplatzproblemen

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

im Arbeitsspeicher führt.

Weil diese Veränderung nur minimal ist, reicht es nur das eindimensionale Netz in den meisten Fällen zu betrachten, weshalb die technischen Probleme beim zweidimensionalen Netzwerk nicht so hinderlich für die Evaluierung von TF sind.

4.3 Augmentierung

Hier wird auf die Erstellung der Augmented Vectors der Direct Cascade Netzwerke für die Klassifikation eingegangen. Jedes der drei Netze hat eine eigene Berechnung davon. Es wird zuerst das COD-, dann das 1DC- und zum Schluss das 2DC-Netzwerk betrachtet.

Bei allen Netzwerken wird der Input des Netzwerkes und die Prediction gebraucht, um diesen Vector zu erstellen. Der Input ist entweder der Datensatz selbst oder der vorherige Augmented Vector. Nur beim ersten Mal ist es der Datensatz, da danach der Augmented Vector existiert. Der Augmented Vector wächst dabei bei jeder Iteration von einem neuen Netzwerk an, da darüber das Wissen aller vorherigen auf das neue übertragen wird. Die Prediction ist das Ergebnis, welches aus der Inferenz des fertig trainierten Netzwerkes kommt. Im folgenden bedeuten die Buchstaben N, W, H und C Datensamples, Bildbreite, Bildhöhe und Channel.

Das COD-Netzwerk hat als Input den Datensatz mit folgendem Shape: (N, W*H). Die Prediction hat immer den Shape (N, 10). Diese beiden Sachen werden in der zweiten Dimension konkateniert. Dies ergibt die Formel 4.1 und damit den Augmented Vector.

$$AugVec(Input(N, W * H), Prediction(N, 10)) = (N, (W * H).10) \tag{4.1}$$

Das 1DC-Netzwerk hat als Input hingegen den Datensatz in folgendem Shape: (N, W*H, C). Da der Channel nur eindimensional ist, wird dieser zuerst entfernt und dann die Berechnung nach der Formel 4.1 durchgeführt. Zum Schluss wird die Channeldimension wieder hinzugefügt. Beide bisher behandelten Netzwerke haben somit einen um N*10 Einträge linear wachsenden Augmented Vector.

Das 2DC-Netzwerk hat einen komplexeren Input mit dem Shape: (N, W, H, C). Dies muss verbunden werden mit der Prediction die in der Form (N, 10) vorliegt. Dafür wird für jedes N zehn Arrays gebaut, die alle die Form (W, H, C) haben. Diese haben von eins bis zehn den Inhalt der Prediction. Dies wird dann auf der Channeldimension konkateniert. Daraus resultiert die Formel 4.2.

$$\begin{aligned} AugVec(Input(N, W, H, C), Prediction(N, 10)) &= Input(N, W, H, C.ConVec) \\ ConVec(W, H, C)[0-9] &= Prediction(10)[0-9] \end{aligned} \tag{4.2}$$

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Dabei ist der ConVec der Vector in dem die Predictionwerte von eins bis zehn jeweils in der Form (W, H, C) enthalten sind. Dies skaliert bei den Netziterationen im Speicherplatz aber in der Form, wie es in Formel 4.3 zu sehen ist.

$$AugVecNew = N * W * H * C_{old} + N * W * H * 10 \tag{4.3}$$

Daraus ergibt sich, dass der Arbeitsspeicherplatz mit einer Steigung von dem Zehnfachen des Datensatzes zunimmt. Bei Daten, die in einem Sample bereits 8192 Bytes benötigen, ist klar, dass diese Variante des Bauens des Augmented Vectors nicht durchgeführt werden sollte, da dieser zu schnell zu groß wird. Aus diesem Grund wird das 2DC-Netzwerk im Folgenden nicht mehr verwendet.

4.4 Mit und Ohne

4.4.1 TF

Hier werden die Netze jeweils einmal mit und einmal ohne Transferlernen ausgetestet. Es werden nur die Direct Cascade Netzwerke betrachtet und sie werden mit deutlich mehr Netziterationen trainiert als bisher. Die Epochenanzahl pro Netzwerk bleibt aber gleich. Dabei werden jeweils nur wenig Targetdaten verwendet.

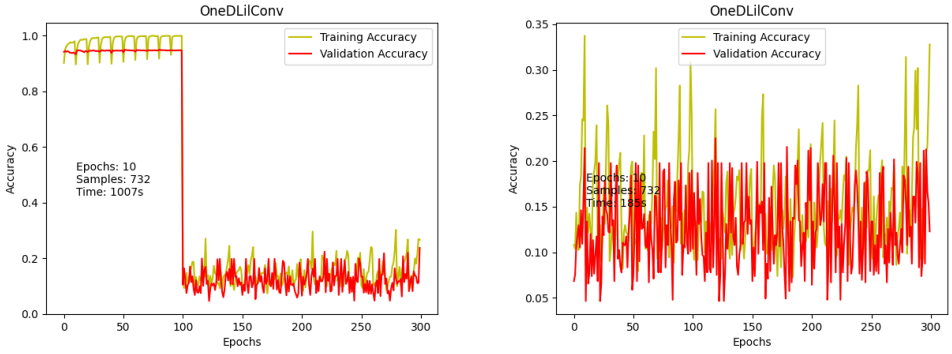

Figure 4.4: Hier ist links der Test 1DC:TF10/732/10/30 und rechts 1DC:732/10/30. Das eine ist mit TF, das andere ohne. Der mit TF dauert viel länger, da der Sourcedatensatz groß ist. Bei beiden ist gut zu sehen, dass die Performanz des Netzes sehr schlecht ist. Egal ob mit oder ohne TF.

Wie in Figure 4.4 zu sehen gibt es keinen Unterschied zwischen der Accuracy mit TF zu der ohne bei Convolutional Layern. In beiden Fällen ist diese extrem schlecht. Dies zeigt sich auch auf den Testdaten.

In Figure 4.5 zeigt sich dasselbe Bild nur auf Basis von Linear Layern. Dies kann zwei Gründe haben: Entweder funktioniert das Kaskadieren nicht oder es sind nicht genügend Targetdaten vorhanden. Letzteres wurde oben ausgetestet

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

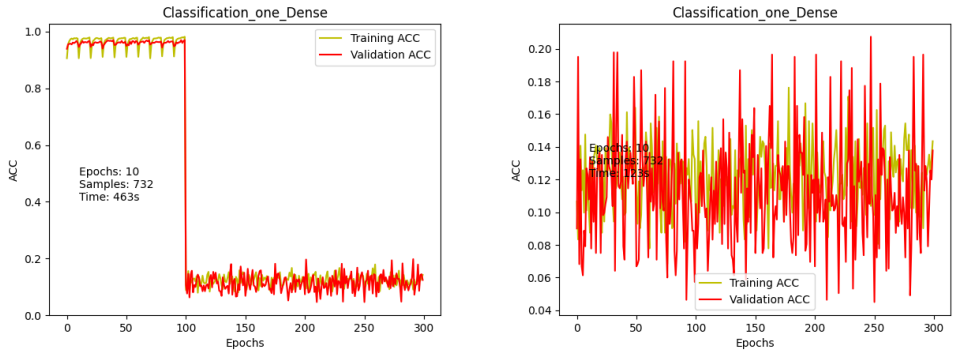

Figure 4.5: Hier ist links der Test COD:TF10/732/10/30, der mit TF ist und rechts COD:732/10/30, der ohne ist. Auch bei dem anderen Fall mit anderen Hidden Layern im Netzwerk ist es sowohl mit als auch ohne TF schlecht.

und lieferte zwar bessere, aber trotzdem nur mäßige Ergebnisse. Das Rauschen in den Plots kommt hier daher, dass alle zehn Epochen ein neues Netzwerk angefangen wird zu lernen. Dies hat zwar das Wissen aller vorherigen Netze im Input, aber nicht in der Art, dass die Gewichte direkt gleich gut sind.

Daraus folgt also, dass es Probleme beim Kaskadieren geben muss.

4.4.2 Kaskadierung

Also wird hier ausgetestet was passiert, wenn nicht kaskadiert wird. Da dies nur dann gut geht, wenn TF nicht verwendet wird, wird alles direkt auf dem Targetdatensatz gelernt. Die Netze werden so verändert, dass sie insgesamt gleich viele Hiddenlayer wie alle kleinen Netze, die im Direct Cascade Verfahren vorkommen, zusammen haben. Es werden dabei auch gleich viele Epochen insgesamt benutzt wie eben gerade.

Was in der Figure 4.6 auffällt ist, dass es bei den meisten Epochen zu keinem Lerneffekt kommt. Ebenso kommt Overfitting vor, was bei so wenigen Daten zu erwarten ist. Obwohl hier nicht TF angewendet wird, gibt es in der Mitte des einen Plot einen plötzlichen Anstieg der Accuracy. Der Start dieser Verbesserung kam davon, dass der Validationwert minimal schlechter wurde, während der Trainingswert minimal besser wurde. Beide Veränderungen waren im Bereich von Zehnteln der Prozente. Deswegen scheint es so, dass es zu einem lokalen Maximum während des Trainings gekommen ist. Bei dem anderen Netz blieb der Wert auf dem des lokalen Maximums stehen, denn es sind die exakt gleichen Werte. Trotzdem wird durch Figure 4.6 klar, dass bei so wenigen Daten eine maximale Accuracy von 40% zu erwarten ist. Dies ist das globale Maximum, da es den maximal möglichen Wert auf den Trainingsdaten vorweist. An diese kommt weder die Version des nur Kaskadierens noch die des Kaskadierens mit TF auch nur ansatzweise heran. Diese haben einen maximalen Wert von 20% und sind somit nur halb so gut.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

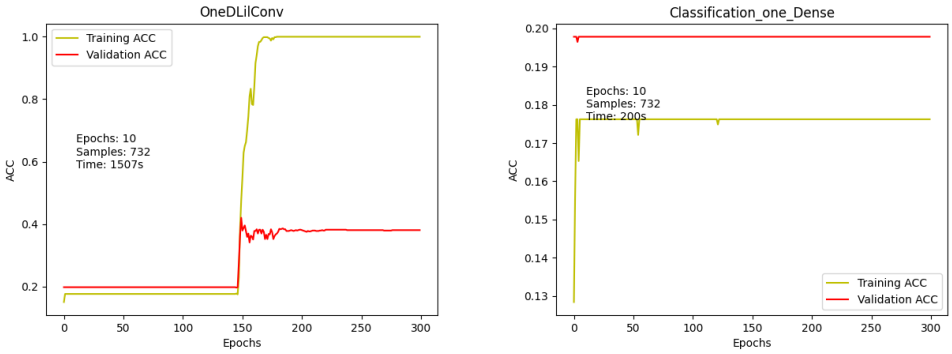

Figure 4.6: Hier sind Tests ohne Kaskadierung. Im genauen links: 1DC:Comp/732//30 und rechts: COD:Comp/732//30. Es ist zu sehen, wie der eine Plot in ein lokales Maximum geraten ist und welcher maximale Accuracywert mit so wenigen Daten erwartbar wäre. Dieser wird nur mit den kompletten hier aufgeführten Netzwerken erreicht.

Daraus folgt, dass es bereits an der Kaskadierung liegt, dass Klassifikation sinnlos mit TF in dem Direct Cascade Verfahren ist. Das kann dabei daran liegen, dass der CategoricalCrossEntropy sich selbst behindert, wenn dieser mehrfach genutzt wird. Sowie es auch an der Softmax-Aktivierungsfunktion oder die Art und Weise des Kaskadierens liegen könnte.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Chapter 5

Regression

Hier werden die beiden Regressionsnetze vorgestellt. Beide haben als Input Tabellen mit drei Spalten. Welche das genau sind, wurde oben bereits erklärt. Sie haben ebenfalls beide den Adam Optimizer mit der Mean Squared Error-Lossfunction. Als Outputlayer wird für Regression typisch ein einzelnes Linear Layer mit einer Node und der Linear Activation Function genutzt.

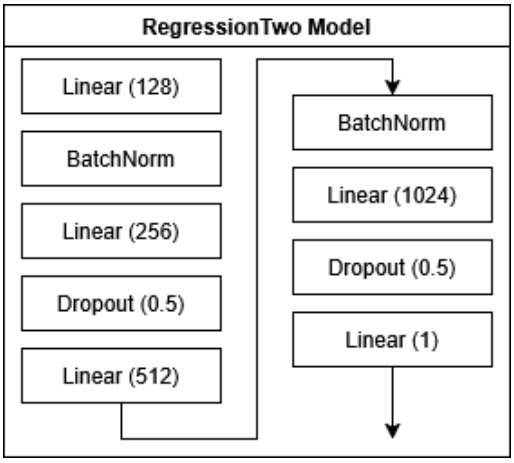

Figure 5.1: Hier ist das Regr2-Netzwerk im Detail zu sehen. Die Layerreihenfolge ist von oben nach unten und folgt dann dem Pfeil.

In Figure 5.1 ist das Regr2-Netzwerk mit allen seinen Layern. Dies ist ein Deep Cascade Netzwerk. Es wird also Layer für Layer trainiert. Dabei ist die Zahl hinter Linear die Anzahl der Nodes und die Zahl hinter Dropout die Prozente bezüglich dem Wert eins, die während des Trainings pro Epoche wegfallen.

Das 1Lay-Netzwerk ist das Direct Cascade Regressionsnetz. Dieses hat nur ein Hiddenlayer mit einem Linearlayer mit 128 Nodes. Die Aktivierungsfunktion

in diesem Layer ist Relu. Es wird iterativ genutzt und zwischen den Netzen Wissen mittels eines Augmented Vectors als neuen Input übertragen. Dieser wird mit der Prediction des vorherigen Netzes berechnet, indem diese als neue Spalte in der Inputtabelle des bisherigen Inputs hinzugefügt wird. Dies ist der Augmented Vector, der als neuer Input für das nächste Netz dient.

5.1 Datenaugmentation

Der Sourcedatensatz Bost hat nur 506 Datensamples insgesamt und ist somit sehr klein. Davon werden im folgenden 51 Samples als Testset, 91 als Validationset und 364 als Trainingsset genutzt.

Der Targetdatensatz Cali ist hingegen mit etwa 26 Tausend Samples sehr groß und wird nach Bedarf verkleinert. Es werden jeweils als Batch 16 Samples genutzt.

5.1.1 Viele Daten

Hier werden alle Vergleiche bei vollem Targetdatensatz verwendet. Dadurch umfassen die Trainingsdaten etwa 8000 Samples und die Testdaten etwa 4000. Diese Vergleiche beinhalten Komplette, TF Cascade und Cascade Netzwerke. Dabei wird als komplettes Netzwerk ein Netzwerk verstanden, welches alle Layer vorher definiert hat und dieses eine Netzwerk mit einem einzigen Trainingsaufruf alles trainiert, was der normale Fall eines neuronalen Netzwerks ist.

Es wird sowohl der Vergleich zwischen Deep Cascade, Direct Cascade und dem Kompletten Netzwerk gemacht.

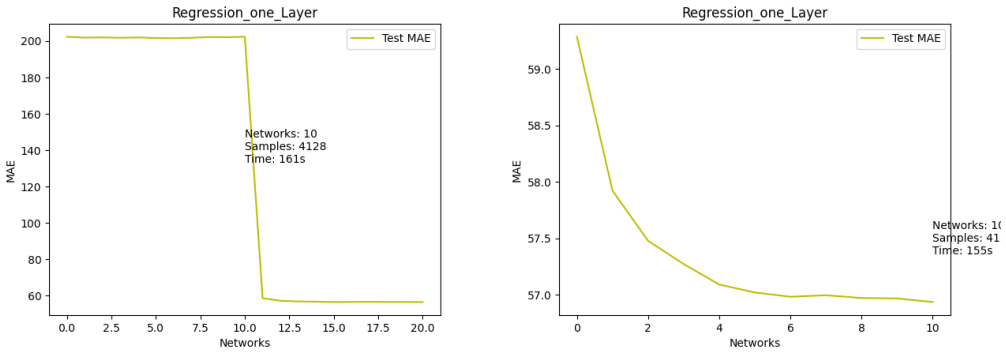

Figure 5.2: Hier sind die Tests 1Lay:TF11/8k/10/21 und 1Lay:8k/10/11 zu sehen. Beide sind am Ende in etwa gleich gut, benötigen nur unterschiedlich viel Zeit, um dahin zu kommen.

In Figure 5.2 ist das Ergebnis des Direct Cascade. Auf der linken Seite ist die Versioin mit TF. Dabei fällt auf, dass es bei vielen Targetdaten besser ist auf

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

dem Targetdatensatz direkt zu lernen, denn das Wissen, welches vom Sourcedatensatz übertragen wird, ist nicht so passend, wie das von den Targetdaten. Dies passiert aber nur, wenn es genügend Targetdaten gibt. Bei diesem Vergleich kommen auch beim Deep Cascade ähnliche Werte heraus. Ein komplettes Netzwerk, wie es für Figure 5.3 genutzt wurde, hat etwa dieselbe Performanz, wie die beiden Kaskadenversionen.

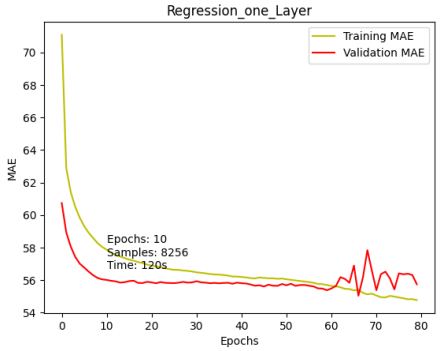

Figure 5.3: Dies hier ist der Test 1Lay:Comp/8k//8. Es ist mit vielen Targetdaten direkt durchgeführt worden, weshalb es bei den folgenden Betrachtungen als bestmöglicher Wert angesehen wird.

Dies liegt daran, dass die lineare Aktivierungsfunktion und der Mean Squared Error Loss für die Kaskadierung nicht störend sind. Also funktioniert Regression deutlich besser mit Kaskadennetzwerken als die Klassifikation, da sie an das Ergebnis des kompletten Netzes herankommt, was bei Klassifikation nie passiert ist.

5.1.2 Wenig Daten

In diesem Unterkapitel wird der Targetdatensatz deutlich verkleinert und hat dann nur noch 240 Datensamples. Es werden dieselben Tests wie im vorherigen Unterkapitel durchgeführt.

In Figure 5.4 sind die Ergebnisse des Deep Cascade Netzwerks. Es ist ohne TF tatsächlich besser als mit. Dies liegt daran, dass die Gewichte der ersten Hälfte des Netzes nur auf dem Sourcedatensatz passend gelernt haben.

Es gibt aber deutliche Unterschiede zu Direct Cascade, weshalb beide Netze hier gezeigt werden.

Figure 5.5 bezieht sich auf das Direct Cascade Netzwerk. Es wird deutlich, dass in dieser Kaskadierungsvariante das Netz deutlich schlechter ohne TF ist als mit. Daran wird erkannt, dass hier positive TF vorliegt. Die Predictions, die mithilfe des Sourcedatensatzes über die zu dem Zeitpunkt nicht betrachteten Targetdaten, verbessern das Gesamtergebnis deutlich. Dadurch gibt es mehr Daten pro Sample, wenn auf dem Targetdatensatz das Training begonnen wird.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

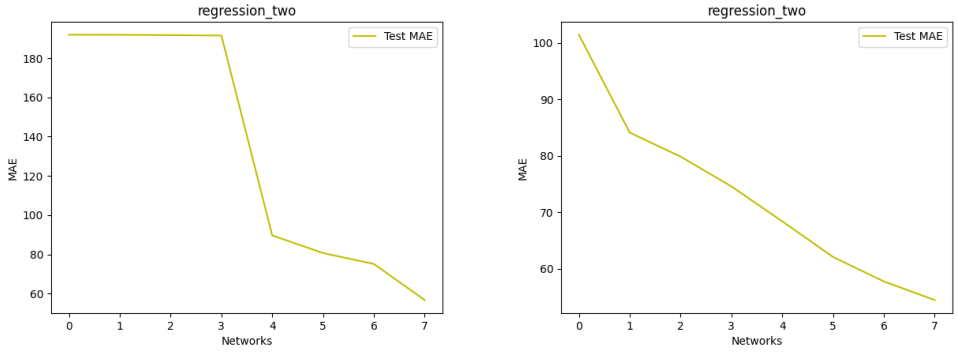

Figure 5.4: Hier ist der Vergleich zwischen mit TF und ohne mit wenig Targetdaten auf dem Regr2-Netzwerk zu sehen. Die Tests sind: links Regr2:TF4/240/10/8 und rechts Regr2:240/10/8. Dabei ist zu sehen, dass dieses Netzwerk ohne TF besser läuft als mit.

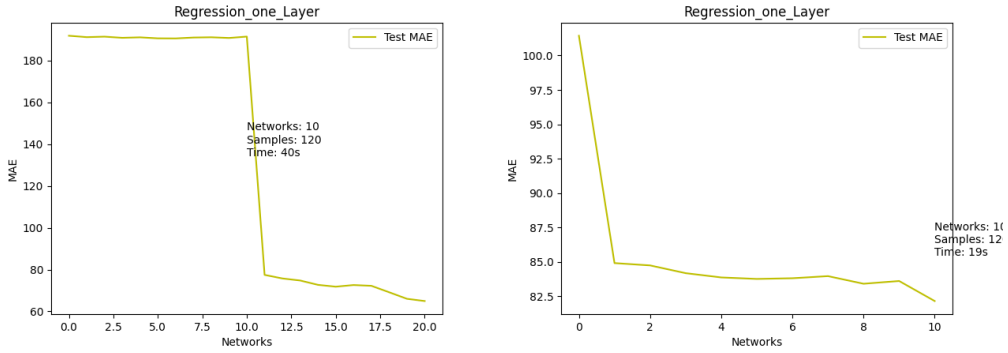

Figure 5.5: Dies ist der Vergleich im 1Lay-Netzwerk im Fall der wenigen Targetdaten zwischen TF und ohne. Die dahinter liegenden Tests sind: links 1Lay:TF11/240/10/21 und rechts 1Lay:240/10/11. Dabei auffällig ist, dass dieses Netzwekr mit TF besser ist als ohne.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

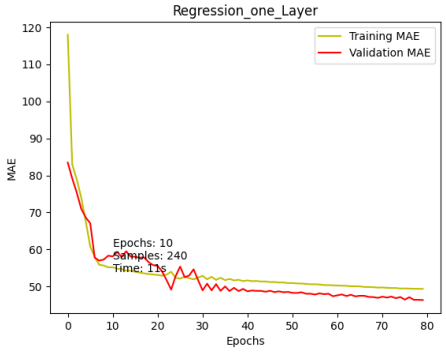

Figure 5.6: Hinter diesem Plot ist der Test 1Lay:Comp/240//8. Dieser ist trotz der wenigen Daten erfolgreicher als alle anderen Tests, obwohl dieser weder Kaskadierung noch TF besitzt.

Diese Daten sind dabei nicht störend für die Performanz, weshalb es dann besser mit TF ist als ohne.

Allerdings ist das Deep Cascade Netzwerk sowohl mit als auch ohne TF minimal besser. Dies kann aber auch an dem dahinter liegenden Netz liegen, da sie nicht nur die exakt gleichen Layer haben. Noch besser ist die Variante, die weder Kaskadierung noch TF nutzt, sondern ein komplettes Netzwerk ist, wie in Figure 5.6 gezeigt.

Der MAE-Wert der Testdaten beläuft sich hier auf 53 Tausend Dollar, während dieser sonst bei so wenig Datensamples bei 60 bis 80 liegt.

Also ist ein Netz gänzlich ohne Kaskadierung, aber mit vielen Hidden Layern und nur einem Trainingsaufruf bei 80 Epochen, auch bei wenigen Daten besser als diejenigen mit Kaskadierung und kommt sogar an den Wert heran, der mit vielen Daten erreicht wird. Dieses Netz hat im Gegensatz zu den Anderen die Möglichkeit zwischen den Layern zu lernen und mit den noch nicht fertig verarbeiteten Daten des Inputs weiterzurechnen. Das dürfte dazu beitragen, dass es eine bessere Performanz gibt.

Wenn diese Tests mit einem explizit großen Testdatensubset angewendet werden, dann ändert sich nur der tatsächliche MAE-Wert dahingehend, dass dieser etwas schlechter wird. Die Verhältnisse zwischen den Netzen hingegen nicht.

5.2 Early Stopping

Hier werden die Regressionsnetze mit Early Stopping verwendet und auch erklärt, warum das bei Klassifikation sinnlos ist. Dabei werden nur die Direct Cascade Netze betrachtet.

Sowohl für die Regressionsnetze als auch für die Klassifikationsnetze wurde LM verwendet. Zudem für Regression noch MAEM und für Klassifikation

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

ACCM.

Bei der Klassifikation kommt es nur manchmal zu einem Abbruch der Epochen über das ACCM, aber es wird dadurch nicht besser. Mit LM kommt dieser Abbruch öfter vor und das Training geht somit zwar schneller, jedoch bleibt Klassifikation mit Kaskadierung so schlecht, dass es nicht genutzt werden kann. Dass weder LM noch ACCM funktioniert sieht man deutlich in Figure 5.7. ACCM ist die einzige der hier vorkommenden Metriken, dessen Ziel ein Maximum ist.

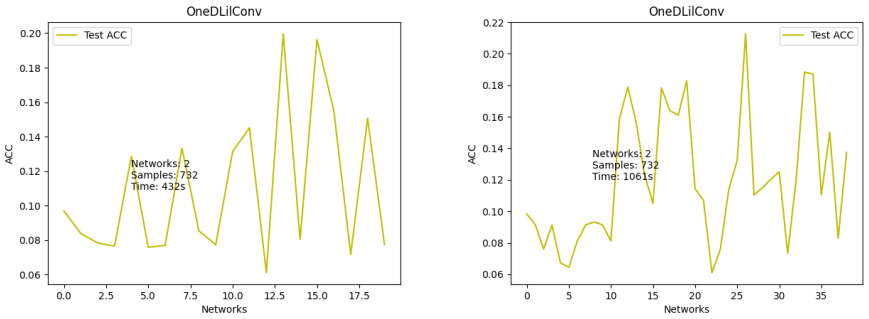

Figure 5.7: LM und ACCM mit 1DConv

Deshalb wird sich hier eingehender mit dem Regressionsnetz OneLayer befasst. Die Metriken LM und MAEM suchen dabei ein Minimum.

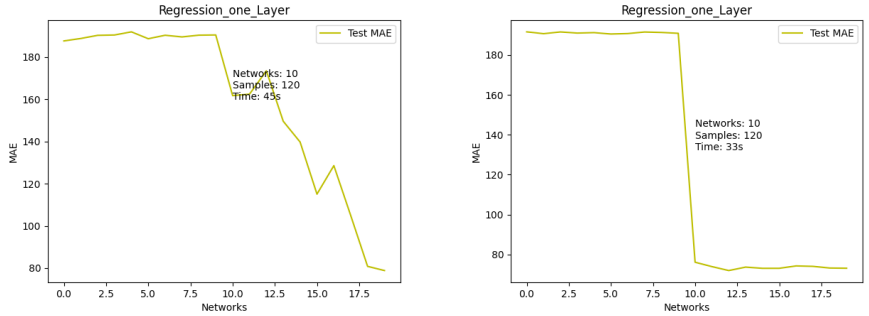

Figure 5.8: LM und MAEM mit OneLayer

Dieses liefert mit den beiden Early-Stopping Metriken LM und MAEM halbwegs brauchbare Ergebnisse, jedoch sind diese deutlich schlechter als ein Training ohne diese, wie an den Werten von Figure 5.8 abgelesen werden kann.

Diese Werte sind so schlecht als hätte man das OneLayer Netzwerk mit wenigen Targetdaten direkt auf diesen Datensatz lernen lassen. Das diese Early-Stopping Metriken so schleht sind, liegt daran, dass sie keine Verschlechterung im Validationset des Datensatzes dulden und ab der ersten das Netz der aktuellen Netziteration beenden. Dadurch ist selten das tatsächliche Minimum

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

das, was über den Augmented Vector weitergegeben wird, sondern nur ein leicht abweichendes. Dazu kommt, dass diese Metriken nicht das globale Minimum finden können, wenn sie auf ein Lokales treffen, denn sie werden versuchen in diesem zu verbleiben.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Chapter 6

Weiterführendes

6.1 Ausblick

Klassifikation sollte nicht weiter in einem Deep oder Direct Cascade Verfahren ausgetestet werden, da es bei egal wie vielen Daten immer schlechter ist als, wenn mit einem kompletten Netzwerk gelernt wird. Wenn, dann nur mit einem anderen Loss als CategoricalCrossEntropy oder einer anderen Art, wie die Augmented Vectors zwischen den Netzwerken gebaut werden. Was sonst noch ginge, wäre ein anderes Kaskadierungsverfahren, aber es ist unwahrscheinlich, dass das besser klappt.

Für die Regressionsnetze kann ein anderer Loss als MeanSquaredError verwendet werden.

Bei beiden Varianten können noch andere Early-Stopping Metriken genutzt werden, die auch eine minimale Verschlechterung dulden. Zudem auch welche, die die Anzahl der Netziterationen anhand einer solchen Metrik bestimmen. Genauso eine, die keine Maxmialanzahl an Epochen besitzt, aber dort kann es passieren, dass das Training nie endet.

Es kann auch noch ein anderer Optimizer als Adam genutzt werden und andere Datensätze als Source und Target. Sowie die aktuellen Datensätze in einer anderen Art für die Netze vorbereitet werden können.

Ebenso kann noch ausgetestet werden, wie sich TF bei einem Taskwechsel verhält, genauso wie bei Task- und Domainwechsel zugleich.

6.2 Fazit

Klassifikation ist mit Direct Cascade mit TF mit den hier genutzten Augmented Vectors nicht machbar. Deep Cascade läuft bei Klassifikation besser, ist aber auch nicht sinnvoll verwendbar.

Regression ist mit TF in der Direct Cascade Version sinnvoll, da es entweder gleich oder besser performt. Die einzige Möglichkeit, dass dies nicht der Fall ist, ist bei sehr wenigen Daten verglichen mit der kompletten Version des Netzes.

Es funktioniert ebenso mit Deep Cascade und das auch in dem eben genannten Sonderfall bei einer mithaltenden Performanz.

Die kompletten Netzwerke sind mit insgesamt gleich vielen Hidden Layer und bei sehr wenigen Daten schneller als Deep Cascade und diese als Direct Cascade. Dadurch ist es sinnvoller ab bereits sehr wenigen Daten ein komplettes Netzwerk zu nutzen als eines mit Kaskadierung.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.

Bibliography

[CK19] Radoslaw M. Cichy and Daniel Kaiser. "Deep Neural Networks as Scientific Models". In: *Trends in Cognitive Sciences* 23.4 (2019).

[Har+97] David Harrison et al. *Corrected Version of Boston Housing Data*. StatLib Library from Carnegie Mellon University. 1997.

[JY10] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on knowledge and data engineering* 22.10 (2010).

[KCR17] Hak Gu Kim, Yeoreum Choi, and Yong Man Ro. "Modality-bridge Transfer Learning for Medical Image Classification". In: *10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics*. 2017.

[LeC+] Yann LeCun et al. *Learning Algorithms for Classification: A Comparison on handwritten digit recognition*.

[LR92a] Enno Littmann and Helge Ritter. "Cascade LLM Networks". In: *Artificial Neural Networks 2*. Ed. by I. Aleksander and J. Taylor. 1992.

[LR92b] Enno Littmann and Helge Ritter. "Cascade Network Architectures". In: *Intern. Joint Conference On Neural Networks*. 1992.

[Mar19] Enrique S. Marquez. "Deep Cascade Learning". PhD thesis. Faculty of Engineering, Physical Sciences Electronics, and Computer Science, 2019.

[MHN18] Enrique S. Marquez, Jonathon S. Hare, and Mahesan Niranjan. "Deep Cascade Learning". In: *IEEE Transactions on neural networks and learning systems* 29.11 (2018).

[ML90] Enrique S. Marquez and Christian Lebiere. *The Cascade-Correlation Learning Architecture*. Tech. rep. School of Computer Science, Carnegie-Mellon University, 1990.

[Net+11] Yuval Netzer et al. "Reading Digits in Natural Images with Unsupervised FEature Learning". In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. Google Inc., Mountain View, CA and Stanford University, Stanford, CA, 2011.

[Nug18] Cam Nugent. *California Housing Prices*. online at www.kaggle.com. 2018.

[RCN07] Miguel Rocha, Paulo Cortez, and José Neves. "Evolution of neural networks for classification and regression". In: *Neurocomputing* 70.16-18 (2007).

[Spe91] Donald F. Specht. "A General Regression Neural Network". In: *IEEE Transactions on neural networks* 2.6 (1991).

[TG99] Nicholas K. Treadgold and Tamáis D. Gedeon. "Exploring Constructive Cascade Networks". In: *IEEE Transactions on neural networks* 10.6 (1999).

[TS] Lisa Torrey and jude Shavlik. "Transfer Learning". In: chap. 11, pp. 242–244.

Der hier hinterliegende Code, die Plots und die Textausarbeitung ist zu finden unter: `https://github.com/Lirras/BA_EvalTF_DDCN`.