Evaluierung von Transferlernen mit Deep Direct Cascade Networks

Simon Tarras

June 23, 2025

Contents

Chapter 1

Einleitung

1.1 Einführung

1.2 Motivation

1.3 Related Work

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 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 diegleichen Netzarten 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, When to Transfer [JY10]. Da es sowohl eine Klassifikation als auch eine Regression ausgetestet wird, 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 Transfer 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, 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 Zimmers 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. Da LSTAT und MedInc wahrscheinlich abhängig sind, da es vermutet wird, dass diejenigen Menschen, die einen niedrigeren Status vorweisen, weniger Einnahmen haben. Deshalb dürfte es über diese beiden Spalten möglich zu 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. Während bei dem gleichen Netz der Input immer vergrößert wird und das TF über diese Vergrößerung passiert, 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 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 werden nach dem Training gefreezt. Dadurch werden die Gewichte mehr der gefreezten Layer nicht mehr verändert.

Die Kaskadennetzwerke lernen dadurch das, was zwischen den Layern gelernt wird, nicht und sind deshalb etwas schlechter als die klassischen Netzwerke bei gleich vielen Daten. Aber, weil immer nur das aktuelle gelernt wird, sind Kaskadennetzwerke im Training sehr viel schneller. 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.

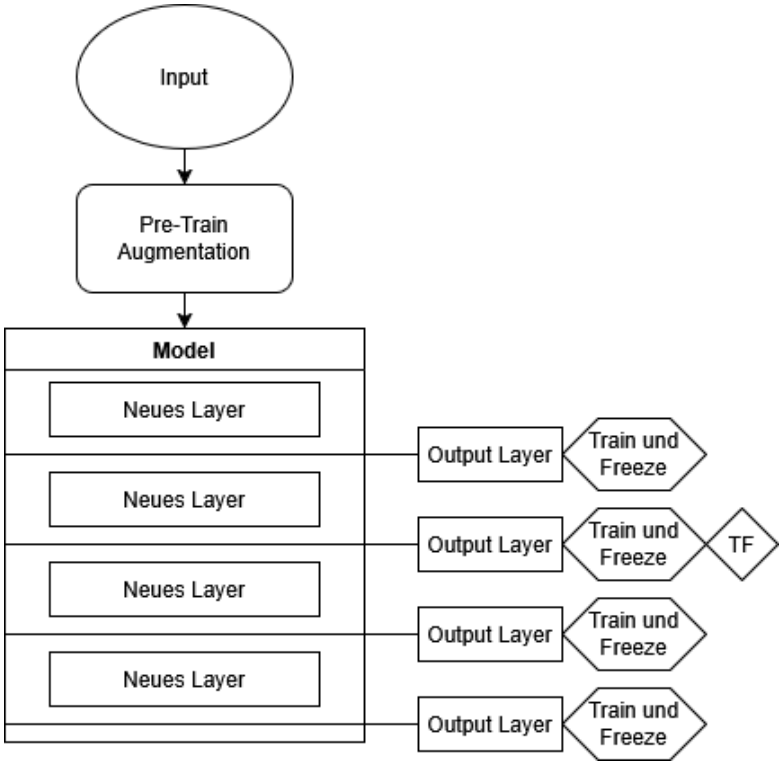

Figure 2.1: Vorstellung Deep Cascade Aufbau

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.

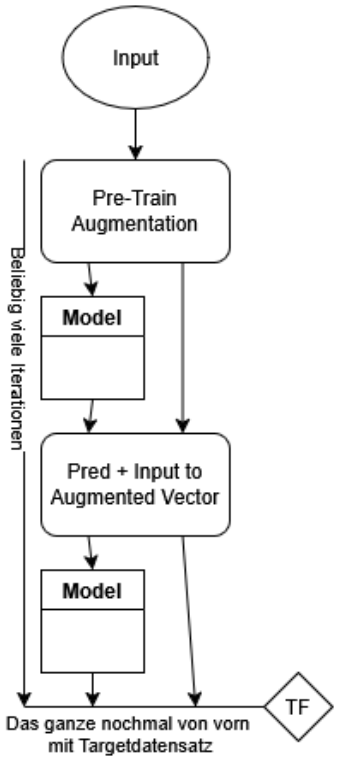

Figure 2.2: Vorstellung Direct Cascade Aufbau

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, 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 und wurden nur trainiert, wenn 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.

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 Netze:

1. ConvMaxPool
2. 1DConv
3. 2DConv
4. ClassOneDense
5. RegressionTwo
6. OneLayer

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.

Alle Netze werden mit dem Adam-Optimizer mit der learningrate 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, ACCM.

Für fast alle Netze gilt, dass sie mit vielen, mittleren und wenigen Source- und Targetdaten durchgeführt wurden. Die einzige Ausnahme ist das 2DConv-Netzwerk, welches nur mit wenigen Daten durchgeführt wurde, da mit mehr Daten es technisch nicht mehr mit derselben Hardware möglich war.

Es wurde mit allen Deep Cascade Netzen ein Vergleich zwischen mit TF und ohne angefertigt.

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

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

Für alle Direct Cascade Networks wurde derselbe Seed für die Initialisierung der Weights genutzt. Zudem wurde ein KFOLD verwendet und der Mean aller Läufe als Wert für den Plot genutzt.

Chapter 3

Allgemeine Resultate

3.1 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 sich nicht verringert. Es wird die Aktivierungsfunktion relu genutzt.

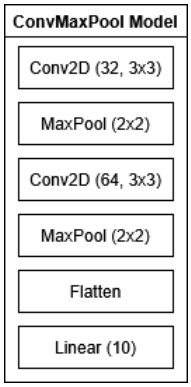

Figure 3.1: Vorstellung des ConvMaxPool Netzwerks

Alle Layer des ConvMaxPool Netzwerks sind in Figure 3.1 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 die Kerngröße steht 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.

3.1.1 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, hat immer einen Einbruch in der Performanz an der Stelle an der TF gemacht wird. Dies ist in der Figure 3.2 deutlich zu sehen, bei Epoche zwanzig.

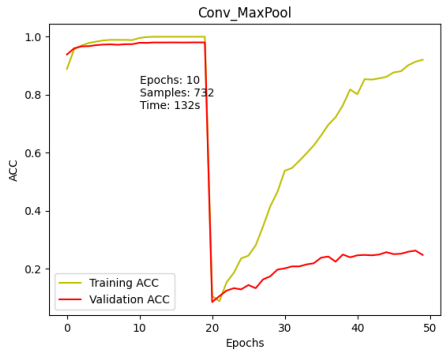

Figure 3.2: Einbruch bei TF

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 3.3 heraus.

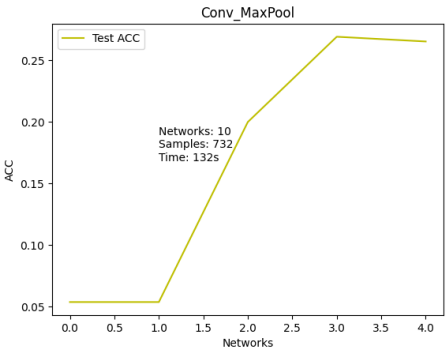

Figure 3.3: Verbesserung auf Testdaten

Der Wechsel ist hierbei bei Netzwerk 2. Es ist eindeutig zu erkennen, dass es nach TF besser wird. Dies hat den Grund, dass das Netzwerk ab diesem Zeitpunkt auf den Trainingsdaten trainiert, die zum Testdatenset passen.

3.1.2 Overfitting auf Sourcedatensatz

Wenn es 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 3.4 zu sehen.

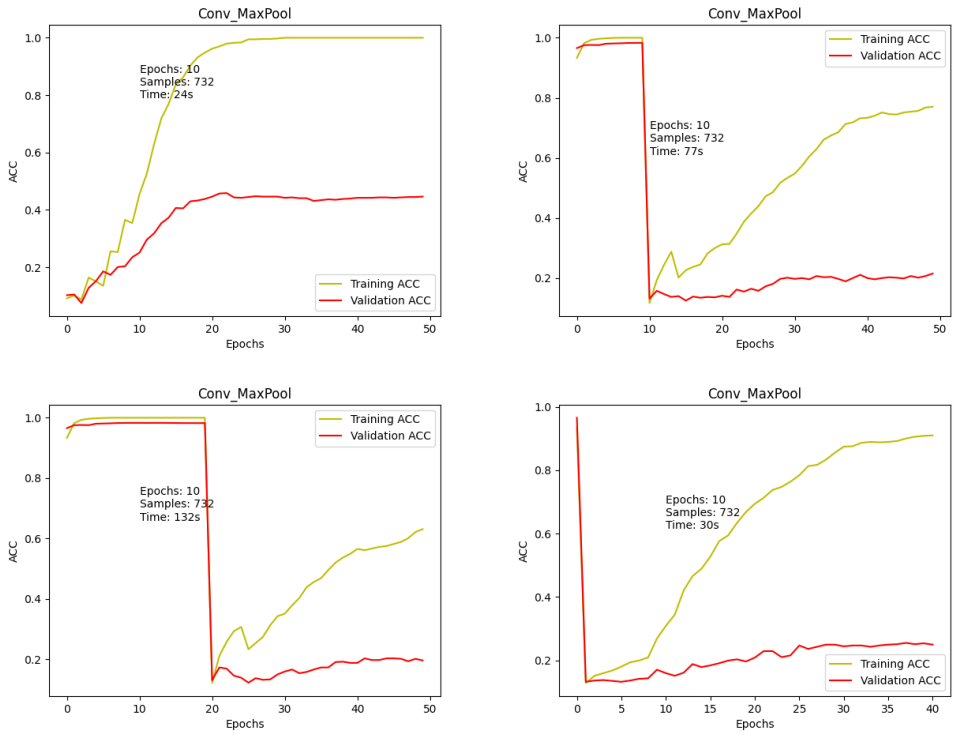

Figure 3.4: TF bei unterschiedlichen Layern

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 ist bei TF nie auch nur annähernd an den Bereich kommt, in dem es bei ohne TF ist. 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 Regressionnetzwerk, wie dem Deep Cascade Netzwerk RegressionTwo kommt es, wie in Figure 3.5 zu sehen, nicht so schnell zu Overfitting. Weder auf dem Sourcedatensatz noch auf dem Targetdatensatz.

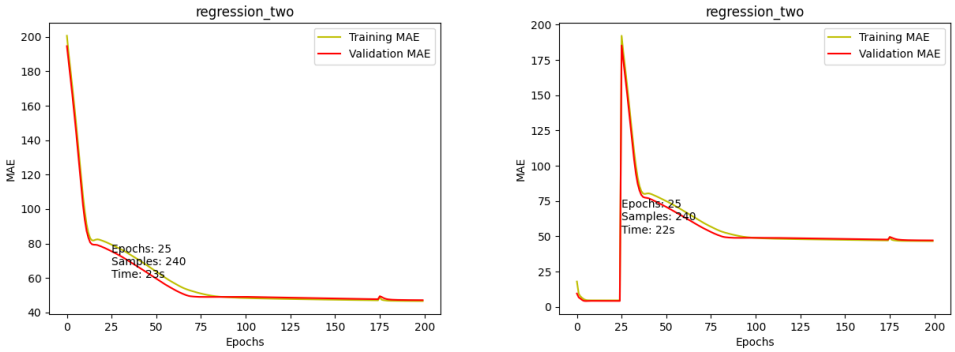

Figure 3.5: TF bei Regression

Dieses Overfitting-Problem hat nur die Klassifikation.

3.2 Zeitnahme

Chapter 4

Klassifikation

4.1 Größe des Targetdatensatzes

4.2 Bilddimensionalität

4.3 Augmentierung

4.4 Mit und Ohne

4.4.1 TF

4.4.2 Kaskadierung

Chapter 5

Regression

5.1 Datenaugmentation

5.1.1 Wenig Daten

5.1.2 Viele Daten

5.2 Early Stopping

Chapter 6

Diskussion

6.1 Erkenntnisse

Allgemeines:

Das ist alles ein Plot eines DeepCascade Networks, welches vor vorgestellt wird: Es kommt relativ schnell zu Overfitting auf dem Sourcedatensatz.

Bei der Stelle des TFs bricht die Performanz des Netzes ein, nur bei deep Cascade erholt sich das ein wenig.

Die Direct Cascade Netze sind erheblich schneller als die Deep Cascade und diese sind wiederum schneller als die normalen Netze.

Inverses Deep Cascade ist völliger Blödsinn.

Klassifikation:

Bei der Klassifikation ist der endgültige Accuracy-Wert hauptsächlich von der Datenmenge des Targetdatensatzes abhängig. Je weniger Daten, desto schlechter ist der Wert.

Klassifikation läuft so schlecht, dass man es nicht tun sollte, da es nichts bringt.

Bei TF ist die Performanz des Netzes etwas schlechter als ohne. Bei Cascade Networks ist es ebenfalls schlechter.

Die Erstellung bei mehrdimensionalen Augmented Vectors kann zur Arbeitsspeicherplatzproblemen führen.

Eindimensionale Klassifikation ist schlechter als zweidimensionale.

Regression:

Die Regression ist mit TF bei wenigen Daten besser als, wenn sie auf den Targetdatensatz von Scratch lernt.

In der Regression ist der Abfall der Performanz bei weitem weniger groß.

6.2 Ausblick

6.3 Fazit

Bibliography

[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).

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

[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.