Evaluierung von Transferlernen mit Deep Direct Cascade Networks

Simon Tarras

May 9, 2025

Contents

1 Einführung **2**
1.1 Cascade Networks 2
1.2 Transferlernen 2
1.3 Wieso keine Graphen 2

2 Klassifikation **3**
2.1 Quellen für Programmierung 3
2.2 Beschreibung der Tätigkeit 3
2.3 Ergebnisse PyTorch 3
2.3.1 Inverses Cascade TF 4
2.3.2 Größe des TargetNets 5
2.3.3 Stabilität des SourceNets 5
2.3.4 Weitere Beispiele 5
2.3.5 Schnelleres Training 6
2.4 Ergebnisse Keras 7
2.4.1 Keras Netzwerke 7

3 Regression **10**

4 Fazit **11**

Chapter 1

Einführung

1.1 Cascade Networks

Kaskadennetzwerke sind so aufgebaut, dass sie während sie im Training sind wachsen und nur die neu hinzugekommenen Sachen traininert werden. [ML90]

1.2 Transferlernen

Transferlernen ist, wenn ein Neuronales Netzwerk von der einen Sache vorlernt, um eine andere Sache besser zu bearbeiten.

1.3 Wieso keine Graphen

Die Daten sind für Klassifikation Bilder und somit sehr einfach strukturiert. Ebenso sind die Daten auch bei der Regression als große Tabellen noch verhältnismäßig einfach zu verarbeiten. Ein Netzwerk mithilfe von Graphen kann sehr viel und ist auch überall anwendbar, doch es ist nicht so einfach damit zu arbeiten als mit PyTorch. Da die Daten nicht so komplex vorliegen, dass Graphennetze zwingend benötigt werden, werden sie nicht genutzt. Zudem liegen einige Datensätze nicht in einer Variante vor, die von den Graphen genutzt werden können. Ein Beispiel hierfür ist der SVHN-Datensatz.

Chapter 2

Klassifikation

Eine Klassifikation ist dann, wenn ein NN eingegebene Daten zu eindeutigen Klassen zuordnen soll.

2.1 Quellen für Programmierung

keras.io, numpy.org, stackoverflow, discuss.pytorch.org, pytorch.org, kaggle

2.2 Beschreibung der Tätigkeit

Es soll TF mit Klassifikation durchgeführt werden. Genutzt wird hier ein Domainwechsel, da die Datensätze gewechselt werden. Die dafür vorgesehenen Datensätze sind Modified National Institute of Standards and Technology(MNIST) und Streetview House Numbers(SVHN). Es gibt die Frameworks PyTorch und Keras, die ebenfalls miteinander verglichen werden. Der MNIST-Datensatz ist dabei die Source und der SVHN das Target. Um einen Vergleichswert zu haben, wurden beide Netze auch allein gelöst.

2.3 Ergebnisse PyTorch

MNIST ist sehr leicht zu lösen. Es reicht ein einzelnes Linearlayer bei keiner expliziten Aktivierungsfunktion. Softmax wird automatisch von PyTorch hinzugefügt. Dies hat bei bereits einer Epoche eine Accuracy von etwa 80% und kann bis auf 90% hoch gehen. Der MNIST-Datensatz wird verändert und bekommt ein Resize von 28x28 auf 32x32 Pixel, sowie ein CenterCrop Layer und eine Normalization in dem Pre-Processing. Dies ist deshalb notwendig, da beide Datensätze in derselben Form vorliegen müssen. Sonst müsste sich die Anzahl der Weights des bisherigen Netzes, die zu dem Zeitpunkt bereits gefreezt sind, bei dem TF ändern. Es ist hier deshalb ein Upscaling der Daten, damit der andere Datensatz keine Informationen unnötig verliert.

SVHN hat als beste Accuracy nur knapp 40%, aber das Training dazu dauert sehr lang (2LayerNetwork: 1 Epoche: ca. 30min, also ca. 3std.). Der Datensatz bekommt als Pre-Processing ein GrayScaling, CenterCrop und Normalization. Die Vermutung, dass dieser Datensatz deshalb so schlecht funktioniert, weil es ein GrayScaling gab, ist falsch gewesen. Das hat keine Auswirkungen.

Ohne Normalization ist das Ergebnis sogar etwas schlechter.

Auffällig bei PyTorch ist, dass die Berechnungszeit sehr lange dauert. Ebenso ist es so, dass der Shape aller Tensoren passend auf den gesetzt wird, der als letztes in einer Epoche betrachtet wird, was zur Folge hat, dass sie ungewollt kleiner werden, wenn die Batchsize kein Teiler der Anzahl der Datansamples ist.

Die PyTorch Netze werden auf der GPU ausgeführt. Dem Modell kann nicht direkt Layer hinzugefügt werden, sondern es muss über eine Iterable-Variable gehen, die in der Forward-Methode durchgegangen wird. Hier ist dies eine Liste. Da nicht jedes Layer direkt in die Liste hinzugefügt werden kann, ergibt sich, dass diese nicht gefreezt werden können. Dies betrifft allerdings nur Layer, die keine Weights haben.

Die Update Regeln sind Stochastic Gradient Descent oder Adam mit ein Layer Backpropagation und freezing weights.

2.3.1 Inverses Cascade TF

Dieses Netz hat eine Accuracy nach TF, die bei 9.5% liegt und ist damit das Schlechteste Ergebnis überhaupt. MNIST liegt in (1, 28, 28) vor, während SVHN in (3, 32, 32) vorliegt. Der Gedanke war, zuerst MNIST zu lernen und dann vorne ein Pre-Processing Layer einbauen, welches SVHN auf den passenden Shape umbaut. Es ist wie folgt aufgebaut:

1. Identity → Conv2D 2 (3, 1, 5)
2. Reshape 1
3. Linearlayer 1 (784, 784)
4. Reshape 2
5. Conv2D 1 (1, 1, 3)
6. Reshape 3
7. Linearlayer 2 (784, 10)

Das TF ist hier also das hinzufügen von Conv2D 2. Davor besitzt dieses Netz eine Accuracy von 88.0% und stürzt dann auf 10% in der ersten Epoche ab und wird pro Epoche noch schlechter. Bei diesem Netz ist es auch so, dass es nach TF nur im Conv2D-Layer weight-updates gibt. Hier liegt Negative Transfer vor. Dieses Ergbenis hat mehrere Gründe. Erstens sind die Datensätze nicht aufeinander abgestimmt und es wird mithilfe eines Convolutionlayer versucht die 3-Kanäligen RGB-Daten zu 1-Kanäligen Daten zu machen, was das genaue Gegenteil ist, was Convolutionlayer tun sollten, nämlich mehr Kanäle

hinzufügen. Zweitens wird zuerst auf MNIST gelernt und die Weights, die am Output des Netzes anliegen, die dann gefreezt sind und auf MNIST laufen, sodass diese die SVHN-Daten nicht gut verarbeiten können. Dadurch ist auf MNIST Overfitting passiert. Und dann hat das Lernen der Gewichte im ersten Layer nur geringe Auswirkungen auf die Performanz des Netzes.

2.3.2 Größe des TargetNets

Wenn es unbekannt ist, wie das Target gelernt werden kann, dann sollte das Netz nach dem TF sehr klein sein, denn sonst verliert sich der Einfluss der Source und die Performanz gleicht sich dem an, die ohne TF gewesen wäre.

2.3.3 Stabilität des SourceNets

Das SourceNet kann bereits eine Stabilität aufweisen, wenn es größer als unbedingt benötigt wird, ist, was zur Folge hat, dass die Accuracy vom TargetNet den Wert nach dem ersten Layer nach dem TF nicht mehr unterschreitet. Aber, wenn das SourceNet zu groß ist, kommt es zu Overfitting. Dadurch wird nach dem TF das TargetNet nicht mehr besser, sondern bleibt nahezu konstant.

2.3.4 Weitere Beispiele

Das Lin4Conv1 Compose Database Network ist ähnlich zu dem Inversen, aber das erste Layer bleibt ein Identity-Layer. Also so:

1. Identity
2. Linearlayer 1 (784, 784)
3. Conv2D (1, 1, 3)
4. Linearlayer 2 (784, 784)
5. Linearlayer 3 (784, 10)

Hier werden die Daten bereits während ihres Ladens bereits so verarbeitet, dass sie zueinander passen. Der SVHN-Datensatz erhält ein downscaling auf die Shape des MNIST-Datensatzes. Zudem wird aus den RGB-Daten von SVHN Grayscale-Daten gemacht. Dieses Netz hat aufgrund des Convolutionlayer eine recht hohe Stabilität. TF wird hier zwischen Linearlayer 2 und 3 durchgeführt. Vor dem TF hat dieses Netz eine ACC von 87.6% und danach eine von 21.2%. Dies ist sehr schlecht, aber immer noch besser als ein solches Conv2D-Layer wie hier alleine mit SVHN. Dieses hat nur 3%. Aber SVHN mit nur einem Linearlayer allein hat bereits einen Wert von 22.7%. Diesen gilt es zu übertreffen. Ohne das Conv2D-Layer hat das Netz 19.6%, was nur Geschwindigkeit bringt. Wenn das Linearlayer mehr Outputfeatures erhält, wird dieses Netz noch schlechter. Das dieses Netz so schlecht ist, liegt daran, dass ein Convolutionlayer normalerweise Channels hinzufügt, was hier nicht passiert und somit die Daten nur reduziert werden.

Das 2LayerLinear ist ein zwei-Layer Netz mit nur Linearlayern. Das erste trainiert mit MNIST, das zweite mit SVHN bei jeweils fünf Epochen. Die Accuracy vor TF ist: 90.4%, während die nach TF: 22.4% ist. Hier wurde MNIST geupscaled, anstatt die Scale von SVHN zu verändern. Daraus folgt, dass es nur einen minimalen Unterschied gibt.

Das 3Conv3Linear ist das erste Netzwerk mit korrekten Convolutionlayern mit Max-Pooling nach jedem Convolutionlayer. Die Prozentwerte dahinter sind die Accuracy-Zahlen nach einer Epoche. Es ist wie folgt aufgebaut:

1. Conv2D 1 (1, 4, 7); 5.8%
2. Conv2D 2 (4, 16, 5); 26.2%
3. Conv2D 3 (16, 64, 3); 32.4%
4. Linear 1 (64, 64); 22.5%
5. Linear 2 (64, 64); 9.3%
6. Linear 3 (64, 64); 10.6%
7. TF
8. Linear 4 (64, 10); 19.6%

Wenn fünf Epochen genutzt werden, dann ist die Accuracy vor TF bei 41.3% und danach bei 20.2%. Bei einer Epoche gibt es zwar positive TF, aber das ist Zufall gewesen, da das Netz vorher so schlecht war.

Das Conv2DMidTF-Netzwerk ist dazu da zu überprüfen, ob ein TF im Convolution-Block sinnvoll ist, was nicht der Fall ist, da MNIST nicht gut mit Convolutionlayern ist. Das Ergebnis sind auch hier 19.5%. Dies dürfte aber hauptsächlich daran liegen, dass die Convolutionlayer im Verhältnis sehr wenige Channels aufmachen, so wie bei allen Netzen bei denen das vorkam.

Das LinNet with Dropout and ReLU ist auf PyTorch extrem schlecht. Es kommt nicht über 20% hinaus, wenn man es als komplettes Netzwerk lernt. Das liegt daran, dass die Features im ersten Layer erweitert werden und es Dropout gibt. Dropout kann die Accuracy um 20% verringern. Dieses Netz ist das Erste, welches mit dem Keras-Framework verglichen wurde und Keras hat eine erheblich höhere Accuracy bei deutlich schnellerer Trainingszeit. In diesem Fall hat Keras um die 75%, was aber an der Art des Ladens der Datensätze liegt.

2.3.5 Schnelleres Training

Obwohl es definitiv Cascade Networks sind und nur die Gewichte, der neuen Layer gelernt werden, ist PyTorch sehr langsam in der Berechnung und es sollte irgendeine Möglichkeit geben, diese zu Erhöhen. Die Idee ist, den verarbeiteten Datensatz zwischenzuspeichern und das Netzwerk wird von dem Zeitpunkt erst aufgerufen, ab dem es neue Layer hat. Der zwischengespeicherte Datensatz kommt von genau dem davorgehenden Zeitpunkt und dient als neuer Input. Das

funktioniert nicht, da der Datensatz nach einem Layer neu geschrieben werden muss. Dies ist bei einem einfachen Layer bereits circa 3 Sekunden langsamer als der normale Versuch. Zudem gibt es keine wirkliche Möglichkeit die weights zu speichern, sowie das Übertragen selbst geht nur als neuer Datensatz. Dies ist aber so umständlich und wurde nur über Numpy-Arrays versucht. Die Rechenzeiten der Transformationen, die dafür benötigten werden ist jedoch so lang, dass es sinnlos ist, dies zu versuchen.

2.4 Ergebnisse Keras

Keras wird deshalb genutzt, weil es sowohl schneller als auch besser ist als PyTorch. Dieses Framework nutzt als Backend Tensorflow. Es gibt zwar die Aussage, dass die PyTorch Dataloader Objekte als Input für ein Keras Modell genutzt werden kann, doch dies ist nicht möglich, da die für die Labels erwartete Shape bei den Dataloader Objekten nicht vorhanden ist. Es gibt eine leere Dimension die aber mit der Anzahl der Klassen gefüllt werden muss. Dies liegt auch daran, dass das Output-Layer bei Keras mit im Modell enthalten ist.

Die Daten der Dataloader Objekte können aber so umgebaut werden, dass sie als Input für Keras funktionieren, aber es läuft nicht so gut. Dann gibt es folgende Werte für die Netze: MNIST allein hat nur 10%, SVHN allein hat nur 19%, SVHN in RGB hat auch nur 19%. Das ist schlechter als PyTorch, aber es gibt noch die zu Keras gehörende Möglichkeit die Daten zu laden. Dies geht dann über numpy-Arrays, die zuerst ein wenig umgebaut werden. Die Channels müssen hinten stehen, die Bilder müssen auf 32x32 ein upscaling bekommen und die RGB-Daten müssen zu GrayScale. Wenn dies alles aber gemacht wurde, hat Keras folgende Werte: MNIST allein: 98% [Cho20] und SVHN allein: 91% [Rou20]. Zusätzlich ist Keras deutlich schneller.

Bei allen Keras-Netzen wird das Outputlayer selbst mit ins Netz hinzugefügt. Daraus resultiert, dass dieses Layer ständig in das Netz hinzu kommt und im nächsten Schritt wieder entfernt. Es handelt sich hierbei um ein Linearlayer mit Softmax-Aktivierungsfunktion.

2.4.1 Keras Netzwerke

1. Dense 1 (1024 Nodes)
2. Conv2D 1 (1, 32, 3)
3. Batch Normalization und TF
4. Conv2D 2 (32, 32, 3)
5. Dense 2 (10 Nodes, Outputlayer)

Dieses Netz hat eine Accuracy von 48.5%. Da es vor dem TF bei 98.2% war und direkt danach bei 22.6%, scheint es zu Negative TF gekommen zu sein, was nicht der Fall war. Es ist allerdings deshalb so gut, weil es sehr viele Channels durch

die Convolutionlayer gibt. Mit Keras geht das, weil es schneller berechnet wird. Ein PyTorch-Netz mit so vielen Channels benötigt pro Epoche in etwa eine halbe Stunde, während es bei Keras wenige Sekunden sind. Dieses Netz wurde zudem nur mit ein bis vier Epochen trainiert, was nur deshalb funktioniert, weil es als Kaskadennetzwerk trainiert wurde. Des Weiteren ist es das Netz, was eigentlich MNIST löst. Es hat in etwa 4 Millionen Parameter. Bei TF in einem beliebigen Layer passiert folgendes:

Table 2.1: ACC Vergleich

TF Layer	ACC Ende	ACC vorher	ACC nachher
Ohne (Dense 1)	41.4%	Kein Wert	24.3%
Conv2D 1	50.3%	97.9%	23.3%
Batch Norm	45.7%	98.2%	21.8%
Conv2D 2	47.5%	98.3%	23.1%
MaxPool	44.2%	98.2%	22.2%
Dense 2	39.3%	98.3%	25.4%

Da dieses Netz auf SVHN ohne TF nur bei 41.4% ist, ist alles mit einer höheren Accuracy positive TF, was bei allen bis auf Batch Norm und Dense 2 vorkommt. Dies liegt bei Dense 2 wahrscheinlich daran, dass es auf MNIST Overfitting ist und es zu wenig auf SVHN trainiert wird. Alle Features, die Layer übergreifend gelernt werden, können hier in der Mangelung von Layern nicht gelernt werden. Dies leigt daran, dass das Netz direkt nach dem TF immer sehr schlecht ist und dann erst besser wird. Diese Verbesserung ist allerdings schneller als ohne TF. Zudem sind die Werte leicht unterschiedlich zwischen den einzelnen Durchführungen, weil der Datensatz gemischt wird. In diesem Fall sollte nachdem mit MNIST bereits nach einem Layer eine Accuracy von über 90% erreicht wurde, direkt TF gemacht werden, damit das Netz die Features davon lernt und keinen großen Fokus auf die Source legt.

1. Conv2D (1, 32, 3)
2. MaxPool (2, 2)
3. Conv2D (32, 64, 3)
4. MaxPool (2, 2)
5. Dense (10 Nodes, Outputlayer) und TF

Dieses Netz läuft deutlich besser. Es hat eine Accuracy von 72.2%, wenn mit 4 Epochen im letzten Layer gelernt wird. Wenn mit nur einer Epoche gelernt wird, hat es einen Wert von 50%. Dieses Netz hat vor dem TF eine Accuracy von 95.1%. Es ist das Beste TF-Netz, obwohl es immer noch deutlich schlechter auf SVHN ist als ohne TF. Die Vermutung ist, dass auf MNIST zu viel trainiert wurde und dort Overfitting entstanden ist.

Table 2.2: ACC Vergleich

TF Layer	ACC Ende	ACC vorher	ACC nachher
ohne (Conv2D 1)	78.2%	Kein Wert	61.2%
MaxPool	55.7%	91.4%	41.6%
Conv2D 2	66.8%	93.4%	53.1%
MaxPool 2	76.5%	93.7%	55.2%
Dense	70.2%	94.5%	51.0%

Dieses Netz wird nur im letzten Layer für 4 Epochen trainiert, sonst immer nur mit einer. Hier ist es auffällig, dass die Performanz des Netzes genau dann mehr wird, je später im Netz TF gemacht wird, während es im vorherigen genau anders herum war.

Chapter 3

Regression

Bei einer Regression kommt es dazu, dass das NN sich mithilfe der Daten der korrekten Funktion annähert.

Chapter 4

Fazit

Bibliography

[Cho20] Francois Chollet. *Simple MNIST Convnet*. online at kers.io. 2020.

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

[Rou20] Dimitrios Roussis. *SVHN Classification with CNN*. https://www.kaggle.com/code/dimitriosroussi classification-with-cnn-keras-96-acc. 2020.