Prädiktion Mit Hilfe von Neuronalen Netzen

Master Thesis

Mark Tamaev Mat.Nr: 516214
HS KOBLENZ | KONRAD-ZUSE-STRAßE 1,56075 KOBLENZ

I. Inhaltsverzeichnis

Einleitung 5
Gliederung 6
I. Künstliches neuronales Netz 7
1. Geschichte 7
2. Neuronen 9
1) Neuron bei einem Lebewesen 9
2) Die Künstlichen Neuronalen Netze 10
3) Aktivitätsfunktion 11
4) Gewicht trainieren 13
5) Bias-Units 15
3. Die binären Operationen und das XOR-Problem 15
4. Synapsen (Verbindungen) und der Netzaufbau 16
5. Backpropagation 17
6. Datenauswertung mit Hilfe von neuronale Netze 18
1) Visual-XSel 19
2) MemBrain 19
3) SPSS 20
4) Matlab 20
5) OpenNN 21
6) OpenCV 21
II. Training von Neuronalen Netzen 23
1. Modellierungsfälle 23
1) Dataklassifikation 23
2) Regressionsanalyse 23
3) Clustering 23
4) Zeitreihen 24

2. Input 24
1) Textbasierter Input 24
2) Auffüllen 25
3) Bilder 25
III. Videoanalyse mit Hilfe von Neuronalen Netzen 27
1. Bekannte Projekte 27
1) Vorhersage von Verhalten von Menschen 27
2) Video Qualität 28
2. Weitere mögliche Forschungsgebiete 29
1) Ziel vorhersage 29
2) Gefahrvorhersage 29
IV. Videovorhersage 30
1. Vorgehensweise 30
2. Anwendung des oben genannten Algorithmus auf das Video 33
1) Parameter und Vorhersage 34
2) Verbesserung der Vorhersage 37
3) Ergebnisse von dem verbessertem Verfahren 39
4) Experiment mit einem kleinen Bild 41
3. Quellkode Beschreibung 42
1) Parameter 51
2) Bilder aufschneiden (SplitImages) 52
V. Zusammenfassung 56
VI. Ausblick 58
VII. Quellverzeichnis 59
VIII. Anhang 60
1. Quellcode 42
2. Ordnerstruktur 60

Abbildungsverzeichnis

Abbildung 1 Homo erectus macht Feuer 5
Abbildung 2 Walter Pitts 7
Abbildung 3 Warren McCulloch 7
Abbildung 4 Donald O. Hebb 7
Abbildung 5 Marvin Minsky 8
Abbildung 6 Seymour Papert 8
Abbildung 7 Teuvo Kohonen 8
Abbildung 8 Neuronen und Synapsen beim Menschen 9
Abbildhung 9 Schematische Darstellung der Funktionsweise einer Unit 10
Abbildung 10 Und (And) 15
Abbildung 11 Oder (Or) 15
Abbildung 12 exklusives Oder (XOR) 16
Abbildung 13 Neuronales Netzwerk von links nach rechts (Input Layer, Hidden Layer, Output Layer) 16
Abbildung 14 Forward-Pass 17
Abbildung 15 Backward-Pass 17
Abbildung 16 Visual-XSel 19
Abbildung 17 MemBrain 19
Abbildung 18 SPSS 20
Abbildung 19 Matlab (nntool) 20
Abbildung 20 Verkleinerte Auflösung (von 500x500 auf 50x50) 26
Abbildung 21 Aufteilung auf mehrere Bilder 26
Abbildung 22 Schlägerei im Parlament 27
Abbildung 23 Dart 29
Abbildung 24 Auto Unfall 29
Abbildung 25 Zusammenfassung aus mehreren Bilder 30
Abbildung 26 Ablaufdiagramm 31
Abbildung 27 Bildaufteilung 32
Abbildung 28 Ausgang von 80x80 Neuronen und 40x40 Neuronen Input 34
Abbildung 29 30x30x3 Neuronen mit (Input+Output)/2 für Hiddenlayer 35

Abbildung 30 16x16x3 Neuronen mit (Input+Output)x4 für Hiddenlayer 35
Abbildung 31 16X16X3 Neuronen mit (INPUT+OUTPUT)X4 + (INPUT+OUTPUT) für den Hiddenlayer 36
Abbildung 32 8X8X3 Neuronen mit (INPUT+OUTPUT)/2 für den Hiddenlayer 36
Abbildung 33 Bildaufteilung bei einem Verbessertem Verfahren 38
Abbildung 34 Links Origenal rechts Vorhergesagte 39
Abbildung 35 Vergleich von dem Beweglichem Objekt 40
Abbildung 36 Differenzbild zwischen dem Original und Vorhergesagten Bild 40
Abbildung 37 Vergleich das Kopf im Vordergrund 40
Abbildung 38 Origenalbild links Vorhergesagte Rechts 41

Einleitung

Ab etwa 700.000 v. Chr. lernte der Mensch (damals Homo erectus) Feuer zu machen. Er sah, wie der Blitz in die Erde schlug und Bäume oder Gras entzündete. Der Homo erectus hatte erkannt, dass die Tiere sich vor dem Feuer fürchten und dass man es dadurch als Schutz vor Tiere benutzen kann. Durch viele Versuche das Feuer zu bändigen hat der damalige Mensch gelernt, dass man das Feuer außer für den Schutz auch für Essen benutzen kann.

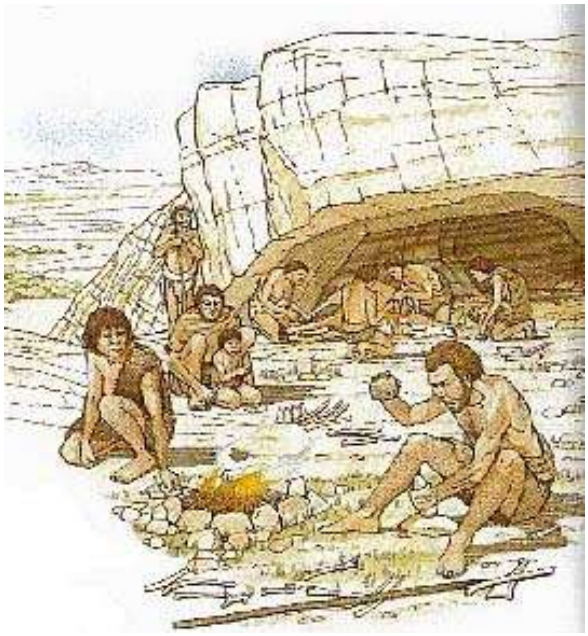


ABBILDUNG 1 HOMO ERECTUS MACHT FEUER

Seit dieser Zeit haben die Menschen sehr viel Neues gelernt und erfunden. Das Feuer ist für den modernen Menschen nichts Neues mehr. Mit dem modernen Wissen konnte der Mensch schon auf den Mond fliegen und in die Tiefen des Ozeans vordringen. Um das Leben zu vereinfachen erfand der Mensch die Rechenmaschine und benutzt sie als Hilfe bei der Arbeit oder auch für seine Freizeit.

Seit Erfindung des Computers versucht der Mensch dem Computer auch viele menschliche Eigenschaften, wie z.B. das Erkennen von Objekten oder das Verstehen von Sprachen oder das Analysieren von Situationen beizubringen. Dies soll dem Menschen helfen, Unfälle zu vermeiden, Objekte schneller zu finden oder die Kommunikation zwischen den Menschen zu verbessern. Die einzige Möglichkeit, dies zu erreichen, ist dem Computer das Verhalten der neuronalen Netze im Gehirn beizubringen. Ebenso, wie der Mensch die Bändigung des Feuers erlernte.

Gliederung

Im ersten Teil werden die Grundlagen von neuronalen Netzen beschrieben. In diesem Teil kann man etwas über die Geschichte der Entwicklung von neuronalen Netzen erfahren sowie über Fall und Aufstieg der Forschung. Hier steht auch der Aufbau des Neurons beschrieben und die Möglichkeit, es zu trainieren. Nach diesem Kapitel können wir ein eigenes Neuron programmieren und trainieren.

Das zweite Kapitel beschreibt die wichtigsten Punkte, die man bei der Auswahl von Datasets beachten muss. Hier werden die Möglichkeiten und Wege zum Import der Daten in das Neuronale Netz beschrieben.

Im drittem Kapitel wird die Videoanalyse mit Hilfe von neuronalen Netzen erläutert. Am Anfang werden die fertigen Projekte beschrieben, danach werden die für dieses Thema interessante Projekte aufgelistet, die entweder noch nicht in der Forschung wahren oder geheim gehalten werden.

Im vierten Kapitel wird die Videovorhersage mit Hilfe eines neuronalen Netzes beschrieben. Hier wird das eigentliche Thema behandelt. Es werden zwei Vorgehensweisen für die Vorhersage anhand mehrerer Bilder beschrieben. Diese Verfahren werden so angepasst, dass sie auch auf einem 32 Bit System arbeiten können.

I. Künstliches neuronales Netz

1. Geschichte

Die ersten einfachen neuronalen Netze wurden im Jahr 1943 von Warren McCulloch (Abbildung 3) und Walter Pitts (Abbildung 2) entwickelt. Sie zeigten, dass man mit Hilfe von neuronalen Netzen prinzipiell jede arithmetische oder logische Funktion berechnen kann. Die ersten neuronalen Netze besaßen aber noch nicht die Fähigkeit zur Selbstmodifikation bzw. dem damit verbundenen Lernen.[1] [2]

ABBILDUNG 2 WALTER PITTS

ABBILDUNG 3 WARREN MCCULLOCH

Das Lernen von neuronalen Netzen wurde erst im Jahr 1949 von Donald O. Hebb (Abbildung 4) in seinem Buch „The Organization of Behaviour“ beschrieben. Die Hebb'sche Lernregel wird heutzutage immer noch fast in allen neuronalen Lernverfahren verwendet.

Der erste bekannte Neurocomputer wurde von Marvin Minsky im 1951 entwickelt. Es war die *Snark*, die in der Lage war, ihre Gewichte automatisch einzustellen. Dieser Rechner wurde aber noch nie praktisch eingesetzt.

ABBILDUNG 4 DONALD O. HEBB

„Mark I perceptron“ war der erste erfolgreiche Neurocomputer (1957 - 1958), der von Frank Rosenblatt, Charles Wightman und Mitarbeitern am MIT entwickelt und für Mustererkennungsprobleme eingesetzt wurde. Dieser Rechner konnte mit einem 20x20 Pixel großen Bildsensor einfache Ziffern erkennen und funktionierte mit Hilfe von 512 motorgetriebenen Potentiometern.

ABBILDUNG 5 MARVIN MINSKY

Im Jahr 1969 wurde durch die genauere mathematiche Untersuchung von Marvin Minsky (Abbildung 5) und Seymour Papert (Abbildung 6) das XOR-Problem, das Parity-Problem und das Connectivity-Problem gefunden. Diese Probleme brachten ein research-dead-end und stoppten die Forschung an den neuronalen Netzen für weitere 15 Jahre.

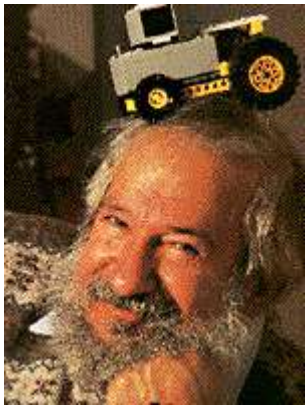


ABBILDUNG 6 SEYMOUR PAPERT

Derjenige der die Neuronale Netze wiederbelebt hatte war Teuvo Kohonen (Abbildung 7). Er stellte im Jahr 1972 in seiner Arbeit „Correlation matrix memories“ ein Modell des linearen Assoziierens, eines speziellen Assoziativspeichers, vor.

Das in modernen neuronalen Netzen meistbenutzte Backpropagation Model wurde erst im Jahr 1974 von Paul Werbos in seiner Desertation entwickelt.

ABBILDUNG 7 TEUVO KOHONEN

Das erste Modell, das zur Erkennung handgeschriebener Zeichen entwickelt wurde, war Neocognitron und wurde von Kunihiko Fukushima, S. Miyake und T. Ito entwickelt.

2. Neuronen

1) Neuron bei einem Lebewesen

Das Künstliche neuronale Netz wurde von einem Neurophysiologen entwickelt und kopiert somit das Verhalten von einem menschlichen und tierischen neuronalen Netzen. Das Menschliche Gehirn hat etwa 100 Milliarden Neuronen und etwa 100 Billionen Synapsen, wobei rein rechnerisch ein Neuron mit 100 anderen Neuronen verbunden ist [4] [3].

Ein **Neuron** (oder auch Nervenzelle) ist eine auf Erregungsleitung und Erregungsübertragung spezialisierte Zelle. In einem Neuron findet eine Gewichtung von Signalen in chemischer und elektrischer Form statt.

Synapsen sind die Verbindungen die Neuronen verknüpfen und somit die Information zwischen ihnen übertragen.

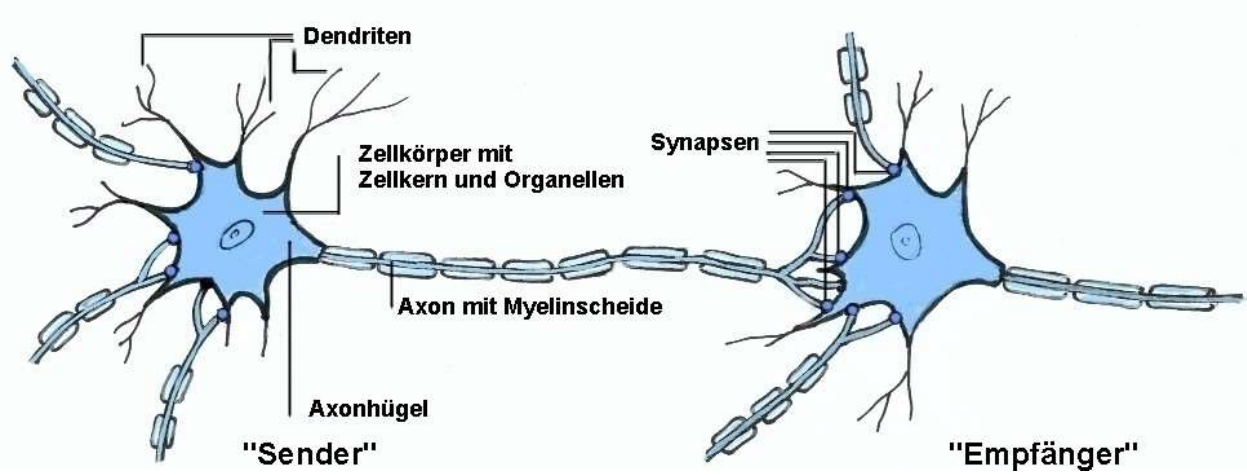
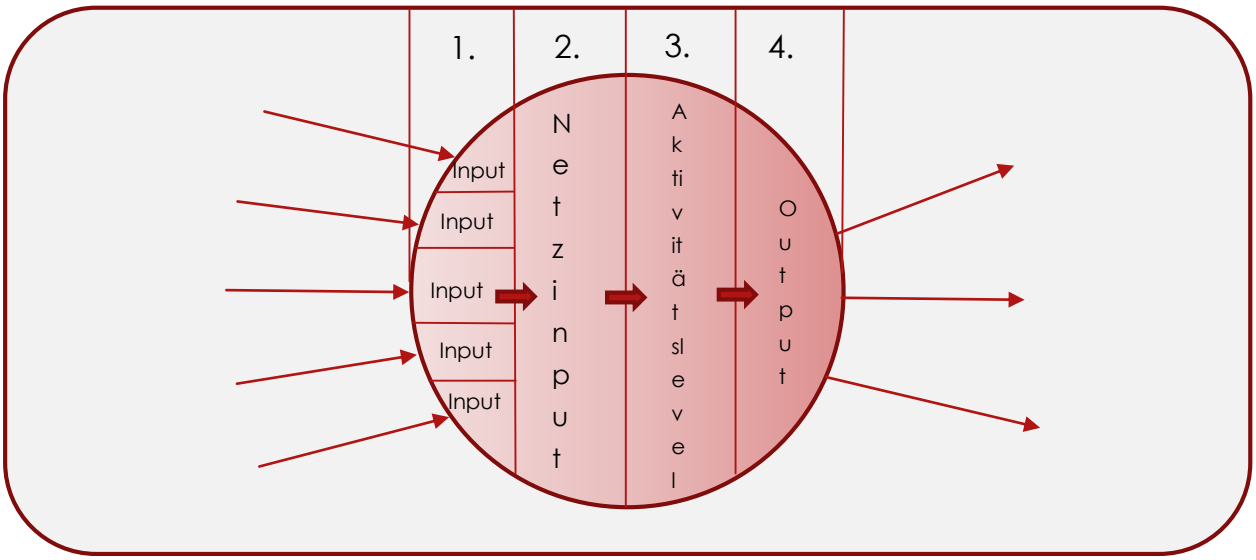


ABBILDUNG 8 NEURONEN UND SYNAPSEN BEIM MENSCHEN

2) Die Künstlichen Neuronalen Netze

Die künstlichen neuronalen Netze bestehen genauso wie menschliche neuronale Netze aus Neuronen und Synapsen. Neuronen werden in der Kybernetik auch als Unit, Einheit oder Knoten bezeichnet. Die Units dienen zur: [5]

1. Berechnung der einzelnen Inputwerte.
2. Bildung des Netzinputs mit Hilfe der einzelnen Inputwerte.
3. Zuordnung des Netzinputs zu einem Aktivitätslevel.
4. Erzeugung des Outputs bzw. der Ausgabe aus dem Aktivitätslevel.



ABBILDHUNG 9 SCHEMATISCHE DARSTELLUNG DER FUNKTIONSWEISE EINER UNIT

Jedem Input der Unit wird ein Gewicht „w“ zugewiesen. Dieses Gewicht wird mit dem Eingang Signal „a“ multipliziert und mit anderen Inputs zu einem Netzinput addiert:

$$input = a * w$$

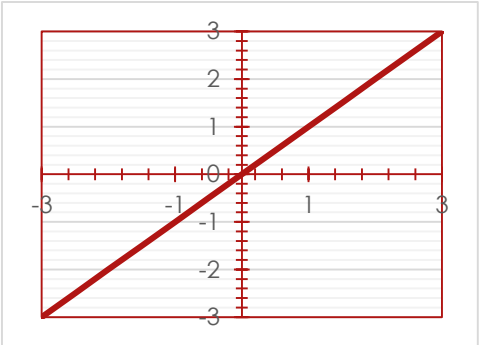
$$netinput = \sum_j input_j = \sum_j a_j w_j$$

3) Aktivitätsfunktion

Der Berechnung des Netinputs wird durch die Aktivitätsfunktion durchgeführt. Es existieren fünf Arten von Aktivitätsfunktionen:

Lineare Aktivitätsfunktion:

In dieser Aktivitätsfunktion wird der Netinput linear mit einer bestimmten Steigung, oder im speziellen Fall ohne Veränderung an den Output weitergeleitet.

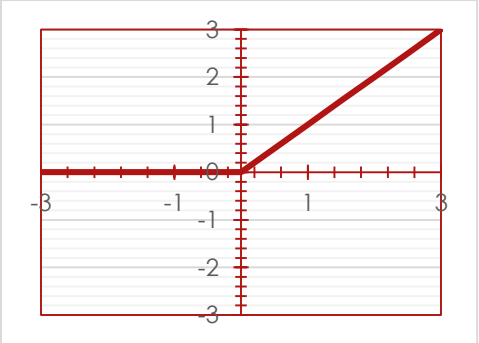


$$output = netinput$$

$$output = m * netinput + b$$

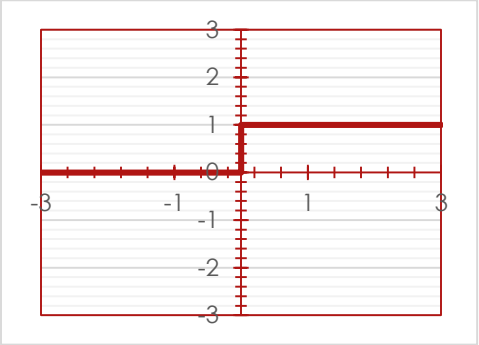
Lineare Aktivitätsfunktion mit Schwelle:

Diese Funktion liefert eine Null vor dem Schwellwert und nach dem Schwellwert funktioniert sie ähnlich wie die lineare Funktion.



$$output = \begin{cases} 0, & netinput < 0 \\ m * netinput + b, & netinput \geq 0 \end{cases}$$

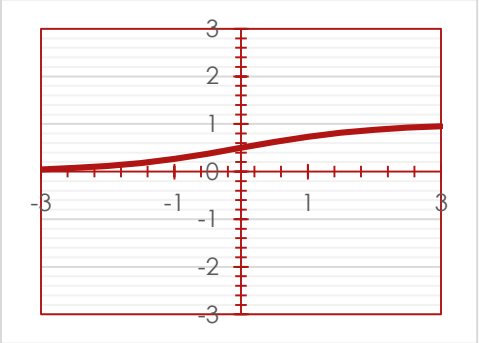
Binäre Aktivitätsfunktion:



Die binäre Funktion gibt eine 0 für Werte unter dem Schwellwert und eine 1 für Werte über den Schwellwert.

$$output = \begin{cases} 0, & netinput < 0 \\ 1, & netinput \geq 0 \end{cases}$$

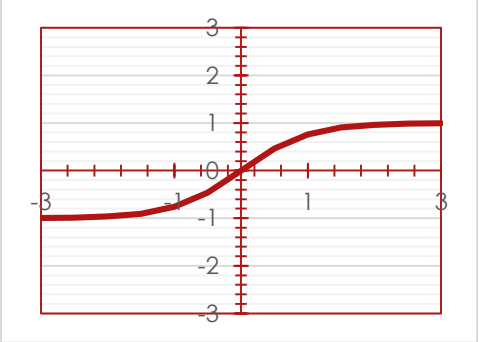
Sigmoide Aktivitätsfunktion:



- Logische Aktivitätsfunktion

Diese Funktion begrenzt den Netzinput auf 0 bis 1. Diese Funktion wird auch als Fermifunktion benannt.

$$output = \frac{1}{1 + e^{-netinput}}$$



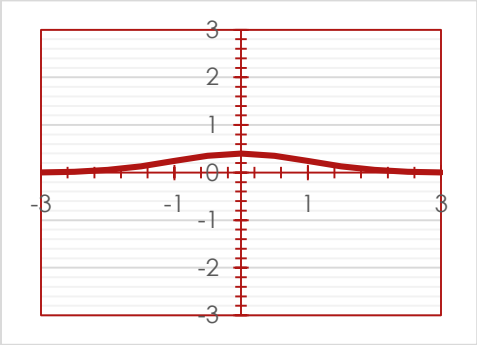
- Tanges Hyperbolicus Aktivitätsfunktion

Diese Funktion hat einen ähnlichen Verlauf wie die logische Aktivitätsfunktion, der Wertebereich liegt aber zwischen 1 und -1.

$$output = \tanh(netinput) = \frac{e^{netinput} - e^{-netinput}}{e^{netinput} + e^{-netinput}}$$

Normalverteilte Aktivitätsfunktion:

Diese Funktion wird sehr selten für neuronale Netze verwendet. Sie wird aber in manchen Programmen zur Auswahl gegeben.



$$output = \frac{1}{\sqrt{2\pi}} e^{\frac{-netinput^2}{2}}$$

4) Gewicht trainieren

Hebb'sche Lernregel

Die erste Trainingsfunktion in neuronalen Netzen war die **Hebb'sche Lernregel**. Die Regel lautet: „*What fires together, wires together*" oder kurz gesagt: das Gewicht wir nur dann geändert, wenn beide Units gleichzeitig aktiv sind.

$$\Delta w_{ij} = \varepsilon * a_i a_j$$

$$w_{ij\_new} = w_{ij\_old} + \Delta w_{ij}$$

Dieses Verfahren funktioniert aber nur, wenn beide Werte größer Null sind. Da in diesem Fall keine Absenkung des Wertes möglich ist, kommt es öfters zu einem Überlauf. Das zweite Problem ist eine geringere Mächtigkeit des Systems: Durch positive und negative Werte können Abweichungen von einem neutralem Wert Null sowohl in positiver als auch negativer Richtung codiert werden.

Delta-Regel

Die Delta-Regel ist eine Erweiterung der Hebb'schen Lernregel. Sie wird auch Widrow-Hoff-Regel oder auch LMS-Regel (Last Mean Square Regel) genannt. In diesem Verfahren wird der Berechnete Wert mit einem gewünschten Wert vergleichen.

$\delta_i = a_{i(soll)} - a_{i(ist)}$

$\Delta w_{ij} = \varepsilon \delta_i a_j$

$w_{ij\_new} = w_{ij\_old} + \Delta w_{ij}$

Dieses Verfahren führt aber nur bei einfachen Aufgaben zum Ziel. Um den richtigen Wert zu errechnen, muss man das Gradientenverfahren anwenden.

Gradienten-verfahren

Das Gradienten-verfahren ist eine Erweiterung der Delta-Regel. Sie benutzt die Regel als Teil des gesamten Verfahrens und wiederholt sie solange, bis der richtige Wert gefunden wurde.

Hier ist die Vorgehensweise:

1. Wahl eines (zufälligen) Startpunktes
2. Festsetzung eines Lernparameters
3. Festlegung eines Abbruchkriteriums
4. Berechnung des Gradienten (mit Hilfe der Delta-Regel)
5. Veränderung der Gewichte

Die beiden letzten Punkte werden solange wiederholt, bis mindestens ein

Abbruchkriterium auftritt.

5) Bias-Units

Außer dem Input kann ein Neuron auch eine Bias-Unit enthalten. Dieses Bias hat als Input immer eine 1. Wegen des Gewichts, das am Bias hängt, wird das Eingangssignal angehoben. In manchen Algorithmen wie z.B das „Online-Training“ wird der Bias sofort mit dem Gewicht zusammengerechnet. In diesem Fall wird aber die Fehlerdifferenz mit -1 multipliziert:

$$\theta = \theta - \varepsilon\delta_i$$

3. Die binären Operationen und das XOR-Problem

Zu Beginn wurde bei neuronalen Netzen viel mit binären Funktionen experimentiert:

Beim dem „AND“ Gatter (Abbildung 10) hebt das neuronale Netz die Ausgleichsgerade so, dass es nur dann „wahr“ annimmt, wenn beide Eingange auf 1 stehen.

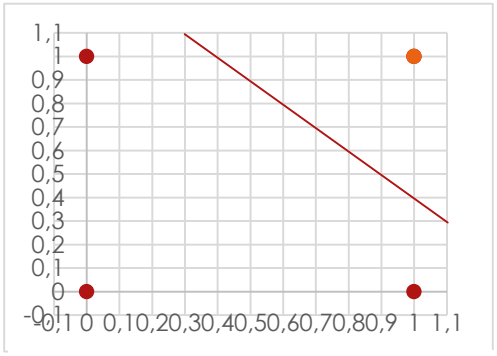


Abbildung 10 Und (And)

Das „OR“-Gatter (Abbildung 11) hat im Vergleich zum „AND“-Gatter ein umgekehrtes Verhalten: es nimmt eine 0 an, wenn beide Eingange auf 0 stehen, ansonsten 1. So stellt das neuronales Netz die Ausgleichsgerade so, dass das Ausgangssignal immer Wahr annimmt wenn irgendein wert gleich Eins ist.

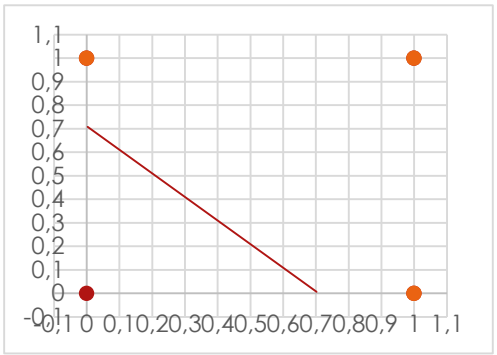


Abbildung 11 Oder (Or)

Das „XOR“-Gatter (Abbildung 12) wird nur dann wahr, wenn nur einer der beiden Ausgänge gleich 1 ist. Leider schafft es das einfache Neuron nicht, die Werte voneinander zu trennen und liefert immer ein falsches Ergebnis. Wegen dieses Verhaltens wurde die Forschung an neuronalen Netzen für 15 Jahre gestoppt. Die Lösung für solche Fälle ist der Aufbau eines größeren Netzes, das aus mehrere Neuronen besteht. Dieses Netz wird dann mit der „Backpropagation“-Methode gelöst. Außer der Backpropagation wurde in vielen Büchern auch Methoden vorgeschlagen wie die Hinzunahme von weiteren Input Units (Macho, 2002), eine normalverteilte Aktivitätsfunktion oder die Modifikation der Delta Regel (Valle-Lisboa, Reali, Anastasia, & Mizraji, 2005).

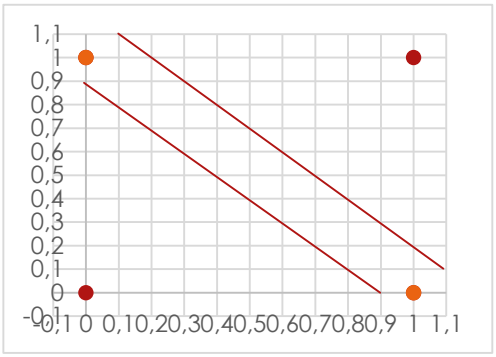


Abbildung 12 exklusives Oder (XOR)

4. Synapsen (Verbindungen) und der Netzaufbau

Um ein funktionierendes Netzwerk aufzubauen, braucht man mehrere Neuronen und Synapsen (Abbildung 13). In diesem Fall wird der Ausgang von einem Neuron auf die Eingänge der anderen Neuronen geklemmt. Deswegen wird in vielen Büchern der Eingang des Empfängerneurons als Ausgang des Senderneurons bezeichnet.

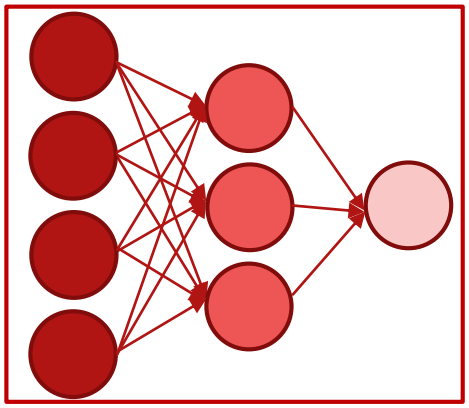


Abbildung 13 Neuronales Netzwerk von links nach rechts (Input Layer, Hidden Layer, Output Layer)

Das Netz besteht aus 3 Schichtarten (engl. Layer). Die erste Schicht wird Input Layer genannt, in die Neuronen aus dieser Schicht wird das Eingangssignal eingeführt. Die dritte Schicht wird Output Layer benannt. Hier kann man die Ausgangswerte darstellen. Die mittlere Schicht heißt Hidden Layer. Sie kann aus mehreren Schichten bestehen und bekommt ihre Signale nur von anderen Neuronen.

5. Backpropagation

Backpropagation ist das meistgenutzte Verfahren um ein neuronales Netz zu trainieren. Sie besteht aus 3 Schritten:

1. **Forward-pass**: Hier werden die Signale an den Input-Layer des Netzes gegeben. Das Signal wird dann vom Input-Layer durch die Hidden-Layer an den Output-Layer geführt und ausgegeben.

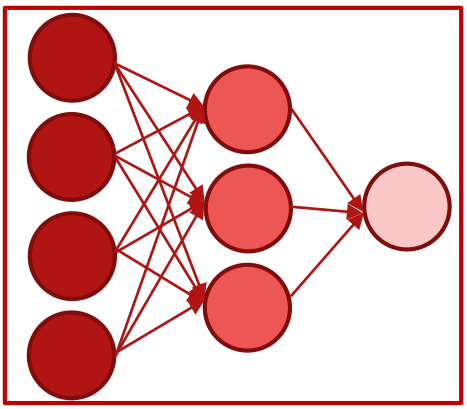


ABBILDUNG 14 FORWARD-PASS

2. **Fehlerbestimmung**: Nach dem man das Ausgangsignal aus dem Output-Layer bekommen hat, wird mit Hilfe der Delta-Regel der Fehler berechnet und aufsummiert. Wenn der Fehler den Schwellwert überschreiten, wird der dritte Schritt ausgeführt. Wenn er unter dem Schwellwert liegt, dann wird die Trainingsphase abgebrochen.

3. **Backward-pass**: Im dritten Schritt liegt die Hauptidee der Backpropagation Algorithmen. Hier geht man durch jede Schicht vom Output-Layer bis zum Input-Layer und berechnet die Gewichte mit Hilfe des Gradientenverfahrens. Wenn der Input-Layer erreicht wurde, fängt man wieder mit dem Forward-Pass an.

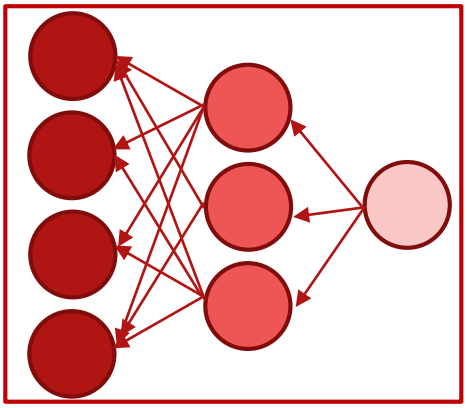


ABBILDUNG 15 BACKWARD-PASS

In dem Backpropagation-Algorithmus wird dieselbe Formel wie in der Deltaregel benutzt:

$$\Delta w_{ij} = \varepsilon \delta_i a_j$$

Wegen der Einführung der Hidden-Layer wird die Formel für den Delta-Wert modifiziert:

$$\delta_i = \begin{cases} f'_{Akt(Netzinput_i)} * (a_{i(soll)} - a_{i(ist)}) \mid i = Output-Unit \\ f'_{Akt(Netzinput_i)} * \sum_L (\delta_l * w_{li}) \mid i = Hidden-Unit \end{cases}$$

Dabei ist:

$f'_{Akt(Netzinput_i)}$: Die erste Ableitung der Aktivitätsfunktion an der Stelle des Netzinputs der empfangenden Einheit i

a_i: Aktivitätslevel der empfangenden Schicht i

L = Nachfolgeschicht nach der empfangenden Schicht i

l = Nachfolgende Einheit, die sich nach der empfangenden Einheit i befindet.

6. Datenauswertung mit Hilfe von neuronale Netze

In diesem Unterkapitel werden die gängige Programmen und Bibliotheken aufgelistet, die für die Analyse mit Hilfe von neuronale Netzen geeignet sind. Es werden Programme wie Visual-XSel, MemBrain, SPSS und Matlab als auch Bibliotheken wie OpenNN und OpenCV vorgestellt und beschrieben.

1) Visual-XSel

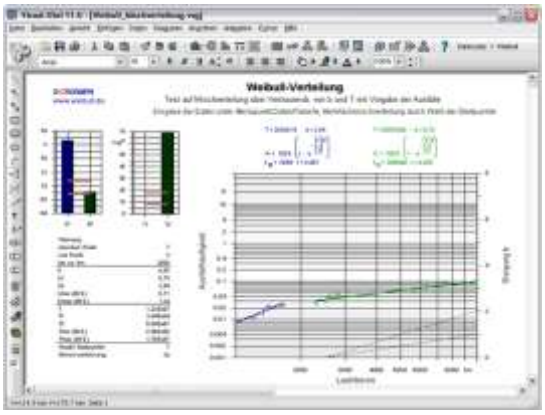


ABBILDUNG 16 VISUAL-XSEL

Visual-XSel ist eine Statistiksoftware, die es ermöglicht, Datensätzen mit Hilfe neuronaler Netze mit der Regressionsanalyse/ANOVA, dem Partial-Least-Square und der logischen Regression zu visualisieren. Dieses Programm stammt von Dipl.-Ing. (FH) Curt Ronniger und kann als Demoversion ohne vorherige Registrierung unter www.crgraph.de/WebDownload.htm heruntergeladen werden. Für Studierende und Mitarbeiter an Universitäten und Fachhochschulen ist die Software für wissenschaftliche Zwecke kostenlos. Diese Software hat in Vergleich zur Matlab und MemBrain sehr wenige Funktionen und ist mehr für die Leute mit wenig Erfahrung in Bereich neuronaler Netze geeignet.

2) MemBrain

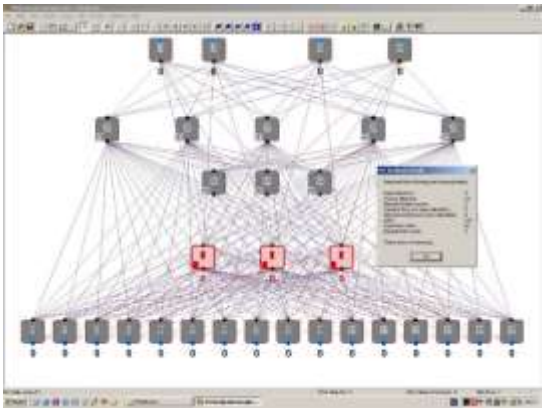


ABBILDUNG 17 MEMBRAIN

MemBrain ist ein Programm zu Visualisierung von neuronalen Netzen. Dieses Programm bietet außer Funktionen zur Datenauswertung (ähnlich wie Visual-XSel und SPSS) auch Funktionen zur Visualisierung der einzelnen Neuronen und zur Gestaltung von speziellen Netzen und ist somit gut geeignet für experimentelle Anwendungen. Diese Software wurde von Dipl. Ing. Thomas Jetter geschrieben und kann auf seiner Homepage (www.membrain-nn.de) für den privaten und nichtkommerziellen Einsatz kostenlos geladen werden.

3) SPSS

Das SPSS ist ein kostenpflichtiges Programm zur Datenauswertung, das innerhalb der Psychologie sehr verbreitet ist. Es hat ähnliche Funktionen wie das Programm Visual-XSel und ist somit nur für Anfänger geeignet ist.

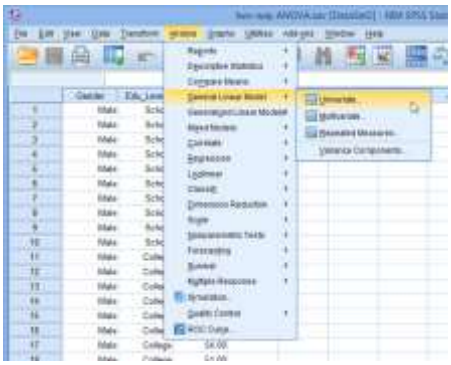


ABBILDUNG 18 SPSS

4) Matlab

Matlab ist ein sehr verbreitetes kostenpflichtiges Mathematikprogramm, das fast für jedes Gebiet der Mathematik und Simulation geeignet ist. In dieses Programm wurde ein Tool namens „nntool“ integriert, das das Trainieren und Auswerten von neuronalen Netz ermöglicht. Außerdem hat es viele Funktionen, die für den Bereich digitale Bildverarbeitung geeignet sind und es beherrscht auch eigene Programmiersprache, was bei der Konvertierung der Bildmatrix ins Dataset hilfreich ist. Diese und tausende andere Funktionen machen dieses Programm zu einem mächtigen Tool für die Analyse neuronaler Netze und es gibt viele Möglichkeiten zum Experimentieren mit Netzen. Wegen der vielen Funktionen ist dieses Programm mehr für erfahrene Benutzer geeignet.

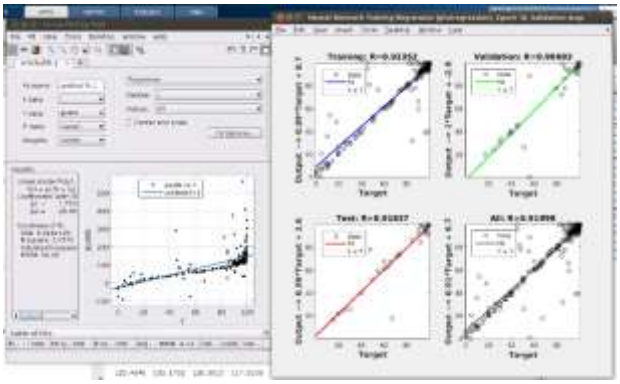


ABBILDUNG 19 MATLAB (NNTOOL)

5) OpenNN

OpenNN ist eine freie Bibliothek, die mit Hilfe der Programmiersprache C++ geschrieben wurde. Sie wird viel im Bereich Deep Learning benutzt. Diese Bibliothek arbeitet mit einer beliebigen Anzahl von Schichten und bietet hohe Performance. Man braucht allerdings viel Erfahrungen in C++ und neuronalen Netzen. Die Bibliothek kann man von der Seite (www.opennn.net) herunterladen.

6) OpenCV

OpenCV ist eine im Bereich Bildverarbeitung sehr verbreitete Bibliothek. Diese freie Programmbibliothek wurde mit für die Sprachen C und C++ geschrieben und dann für weiteren Sprachen wie Python und C# portiert. Außer den Funktionen für Bildverarbeitung bietet es auch Algorithmen für das Training und Auswerten von neuronale Netzen. Mit Hilfe der Funktionen für digitale Bildverarbeitung kann man Bilder in die Dataset Matrizen konvertieren, für die neuronalen Netze vorbereiten und nach der Auswertung zurück in das ursprüngliche Bildformat konvertieren.

OpenCV benutzt die „ml::ANN_MLP“ Klasse, um Funktionen für neuronale Netze abzubilden.

Am Anfang muss man den Pointer mit Hilfe der Create-Funktion erzeugen und somit das leere neuronale Netz herstellen:

```
Ptr<ml::ANN_MLP> nnPtr = ml::ANN_MLP::create();
```

Mit der Hilfe der Funktion „setLayerSize“ kann man die Ebenen anlegen:

```
vector<int> layerSizes = { 20, 15, 8, 10 };
nnPtr->setLayerSizes( layerSizes );
```

Die Funktion „setActivationFunktion“ setzt die Aktivierungsfunktion:

```
nnPtr->setActivationFunction(cv::ml::ANN_MLP::SIGMOID_SYM, 1, 1 );
```

Die Auswahl zwischen Backpropogation und RPROB sowie Parameter wie Schrittweite für das Training werden in der Funktion „setTrainMethod“ gesetzt:

```
nnPtr->setTrainMethod(ml::ANN_MLP::BACKPROP,0.000001,0.000001);
```

Die maximale Iteration sowie minimale Fehlerquote werden in der Funktion setTermCriteria gesetzt:

```
TermCriteria tc;
tc.epsilon=0.00001;
tc.maxCount=10000;
tc.type = TermCriteria::MAX_ITER + TermCriteria::EPS;
nnPtr->setTermCriteria(tc);
```

Mit Hilfe der „train“ Funktion wird das Training gestartet. Ans Input werden die Proben „samples“ gegeben und ans Output werden die Antworten „responses“ auf die Proben gegeben:

```
nnPtr->train( samples, ml::ROW_SAMPLE, responses)
```

Die „predict“ Funktion wendet das angelernte an das Eingang Signal:

```
nnPtr->predict( samples, output );
```

II. Training von Neuronalen Netzen

1. Modellierungsfälle

[6]

In neuronalen Netzen existieren 4 Lernprobleme, die man auf unterschiedliche Weise lösen muss.

1) Dataklassifikation

Bei der Dataklassifikation ist zu jedem Input ein Label gegeben. Das meistbenutzte Dataset ist das Iris-Blumen Dataset von Fisher (1936):

http://www.heatonresearch.com/wiki/Iris_Data_Set

TABELLE 1 FISHER'S IRIS DATASET BEISPIEL

Sepal length	Sepal width	Petal length	Petal width	Species
7.9	3.8	6.4	2.0	*I. virginica*
7.7	3.8	6.7	2.2	*I. virginica*
7.7	2.6	6.9	2.3	*I. virginica*

2) Regressionsanalyse

Bei der Regressionsanalyse ist es ähnlich, nur wird anstatt des Labels oder der ID eine Zahl gespeichert. Dazu gehören Daten wie z. B. Strompreisanstieg oder Kraftstoffverbrauch, die berechnet werden können.

3) Clustering

Clustering ist ähnlich der Dataklassifikation, hat aber keinen bekannten Ausgang. Der Programmierer gibt dem neuronalen Netz die Anzahl der Untergruppen vor und das neuronale Netz ordnet die Objekte anhand von Ähnlichkeitskriterien zu.

4) Zeitreihen

Normalerweise arbeiten neuronale Netze nur mit einer Sequenz und brauchen dafür keine Zeitreihe. Es gibt aber Fälle, in denen für den Ausgang die Daten von früheren Sequenzen benötigt werden, oder Videosequenzen, bei denen die Information aus mehreren Bilder extrahiert werden soll. Für diese Fälle muss man im neuronalen Netz einen Teil der Daten aus der letzten Sequenz in die neue Sequenz einzufügen.

2. Input

Das neuronale Netz arbeitet wie jede mathematische Funktion mit Zahlen und Variablen. Deswegen muss man jeden Input in eine mathematische Form konvertieren und dann in ein Array einfügen, dessen Größe gleich der Anzahl der Inputneuronen ist.

1) Textbasierter Input

Ein Text hat keine Zahlen, mit denen man Rechnen könnte. Deswegen wird in diesem Fall jedem Wort eine Zahl zugewiesen:

Wenn die Seele bereit ist, sind es die Dinge auch.

Wo Worte selten, haben sie Gewicht.

[1 2 3 4 5 6 7 8 2 9 10 11]

[12 13 14 6 15 16 17 11]

1	Wenn
2	die
3	Seele
4	bereit
5	ist
6	,
7	sind
8	es
…	…

2) Auffüllen

Das neuronale Netz hat eine feste Anzahl von Inputs und Outputs. Man kann die Anzahl während des Trainings nicht vorhersagen oder verändern. Deswegen muss man die fehlenden Stellen mit Nullen auffüllen.

1	2	3	4	5	6	7	8	2	9	10	11
12	13	14	6	15	16	17	11				

1	2	3	4	5	6	7	8	2	9	10	11
12	13	14	0	0	6	15	16	17	0	0	11

3) Bilder

Neuronale Netze werden häufig in der Bildbearbeitung benutzt. Die Bilder werden meistens in mehrdimensionalen Matrizen gespeichert, wobei die ersten 2 Dimensionen (X und Y) die Position darstellen und die restlichen Werte für die Farben stehen (zum Beispiel für RGB). Der Input des neuronalen Netzwerkes ist aber ein Array, deswegen muss man die mehrdimensionale Matrix in ein Array konvertieren.

R11 G11 B11	R12 G12 B12	R13 G13 B13	…
R21 G21 B21	R21 G21 B21	R13 G13 B13	…
R31 G31 B31	R32 G32 B32	R33 G33 B33	…
…	…	…	…

[R11 G11 B11 R12 G12 B12 R13 G13 B13 R21 G21 B21 R21 G21 B21 R13 G13 …]

Das zweite Problem im neuronalen Netz ist die Größe. Das Netz hat mehrere Schichten mit mehreren Neuronen. Wenn wir in das Netz zum Beispiel ein farbiges Bild mit einer FullHD Größe-mit insgesamt 6220800 (1920x1080x3)

Eingangsneuronen einspeisen, dann wird das ganze System wegen des fehlenden Speichers scheitern oder das Training wird durch die große Anzahl an Neuronen mehrere Jahren dauern. Dafür gibt es 2 Wege, um die Anzahl der Inputs zu verkleinern.

ABBILDUNG 20 VERKLEINERTE AUFLÖSUNG (VON 500X500 AUF 50X50)

1. Die Auflösung des Bildes verkleinern.

2. Wenn es keine Abhängigkeiten von Farbe gibt, dann kann man das Farbbild in ein Schwarzweiß- oder Graubild konvertieren.

3. Wenn die Pixel im Bild keinen Einfluss haben, kann man das große Bild auf mehrere kleinere Bilder zuschneiden, um dann mit den kleinen Teilen zu

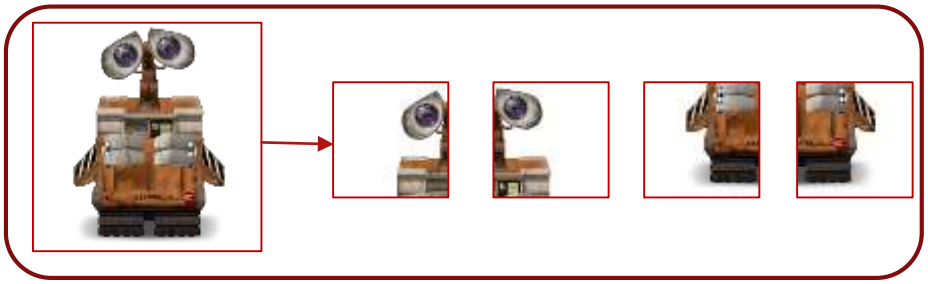


ABBILDUNG 21 AUFTEILUNG AUF MEHRERE BILDER

arbeiten.

III. Videoanalyse mit Hilfe von Neuronalen Netzen

1. Bekannte Projekte

Neuronale Netze werden oft für Bild und Video Analyse benutzt. Hier sind ein paar Projekte aufgelistet, die sich mit Video Analyse mit Hilfe von neuronalen Netzen befassen.

1) Vorhersage von Verhalten von Menschen

Um Konflikte zu vermeiden, muss der Computer das Verhalten von Menschen analysieren, um gewalttätige Momente so früh wie möglich vorhersagen zu können. Man kann zum Beispiel an Hand des Verhaltens von Menschen in einem Fußballspiel entscheiden, ob es zu einer Schlägerei kommt (Abbildung 22). Im Kino und in Büchern wurden schon vor langer Zeit Menschen oder Maschinen beschrieben, die eine Straftat vorhersagen können.

ABBILDUNG 22 SCHLÄGEREI IM PARLAMENT

Im Film „Minority Report“, der auf der gleichnamigen Kurzgeschichte basiert, wurden Menschen beschrieben, die einen Mord vorhersagen konnten. In der Serie „Person of Interest“ hatte ein Programmierer eine Maschine entwickelt, die die Menschen durch Kameras beobachtet und anhand ihres Verhaltens vorhersagen kann, ob eine Person jemanden tötet oder ob einer Person eine Gefahr droht.

Leider existiert diese Technologie heute noch nicht, es wird aber versucht, mit Hilfe von neuronalen Netze etwas Ähnliches zu aufzubauen.

Der MIT Doktorand und Projektforscher Carl Vondrick hat einen bekannten Versuch gemacht, um menschliches Verhalten vorherzusagen. Er hatte das neuronale Netz mit 600 Stunden Youtube Videos und Sitcoms gefüttert und dem Computer die Vorhersage von menschlicher sozialer Interaktionen, wie Umarmen, Küssen, High-fiving und Handshaking beigebracht. Das Netz konnte zu 43% erkennen, was in der nächsten Sekunde passiert, der Mensch in Vergleich antwortete zu 71% richtig. [7]

2) Video Qualität

Ein nicht komprimiertes Video kann mehrere Terabytes groß sein, deswegen wurden Codecs entwickelt, die Videos in ein kleineres Format codieren. Diese Verfahren verschlechtern aber die Videoqualität. Um die Qualität zu bewerten gibt es 3 gängige Verfahren: PSNR, SSIM und die subjektive Videoqualität. Die ersten beiden Verfahren basieren auf Mathematik und können leicht dem Rechner beigebracht werden. Das Ergebnis unterscheidet sich aber meistens stark von dem, was der Mensch sieht. Für das subjektive Verfahren braucht man einen Menschen, der die Qualität betrachtet und bewertet, deswegen kennt man immer die für den Betrachter richtige Antwort, man kann sie aber leider nicht programmieren.

Für solche Fälle hatte die Firma Netflix (**Ошибка! Источник ссылки не найден.**) dem Computer die subjektive Videoqualitätserkennung beigebracht. Durch Mischung mehrerer Verfahren ist das neue VMAF Model entstanden. [8]

2. Weitere mögliche Forschungsgebiete

Hier werden interessante Themen aufgelistet, für die man die neuronale Videoanalyse benutzen könnte.

1) Ziel vorhersage

Man könnte mit Hilfe eines neuronalen Netzes vorhersagen, ob zum Beispiel ein Basketball den Korb trifft oder ob ein Dart-Spieler das Bullseye trifft. Diese Funktion könnte man sowohl im Sport für wetten benutzen und als auch im Militär für die Umlenkung eines Schusses.

Abbildung 23 Dart

2) Gefahrvorhersage

Wenn man im Auto eine Kamera einbaut und das neuronale Netz Videos mit Unfällen oder anderem Gefahren anlernt, kann man durch die Beobachtung viele Unfälle vermeiden, in dem man automatisch bremst, wenn man sieht, dass das Gegnerauto nicht rechtzeitig bremsen kann. Man kann dem Netz auch Prioritäten beibringen, wie zum Beispiel gegen einen Baum fahren anstatt ein Kind zu verletzen.

Abbildung 24 Auto Unfall

IV. Videovorhersage

Die Aufgabe ist, mit Hilfe von neuronalen Netzen anhand eines oder mehrerer Frames den nächsten oder einen Frame nach einer gewissen Zeit vorherzusagen. Da es sich um eine Vorhersage handelt, müssen in diesem Fall, wie es im Unterkapitel Zeitreihen beschrieben wurde, die Bilder aus der Vergangenheit im Eingangssignal zusammengefasst werden.

Für das neuronale Netz wird die OpenCV Bibliothek benutzt. Da diese Bibliothek für die Bildverarbeitung geschrieben wurde, bringt sie auch weitere Vorteile mit wie zum Beispiel die Zusammenführung der Bilder oder auch die Konvertierung des Bildes in ein Dataset und zurück. Für das Training wird die Backpropagation Methode benutzt.

1. Vorgehensweise

Um RAM Speicher zu sparen werden die Bilder in kleinere Bilder geschnitten. Diese Bilder sollen Teile aus mehreren Frames enthalten.

ABBILDUNG 25 ZUSAMMENFASSUNG AUS MEHREREN BILDER

Für diese Aufgabe braucht man mehrere Frames für eine Vorhersage, deswegen gehören sie zu einer Zeitreihe, die aus mehreren zeitlich aufeinander folgenden Bildern besteht. In diesem Fall werden einzelne Frames in einer Kollage mit mehreren Bildern zusammengefasst.

Da man für eine Prädiktion keine Farbbilder braucht wird für diesen Fall nur der Luminanzkanal benutzt. Somit braucht man weniger Eingangsneuronen und auch wenig er Ausgangsneuronen.

Das Ausgangssignal ist das Zukunftsbild. Das kann der nächste Frame sein, aber auch ein Frame in 60 Sekunden.

Im Bild Ausschneiden wird ein Bild in kleinere Bilder aufgeteilt und dann aus mehreren Bildern zusammengefasst und in ein Arrayformat Pixel für Pixel gespeichert.

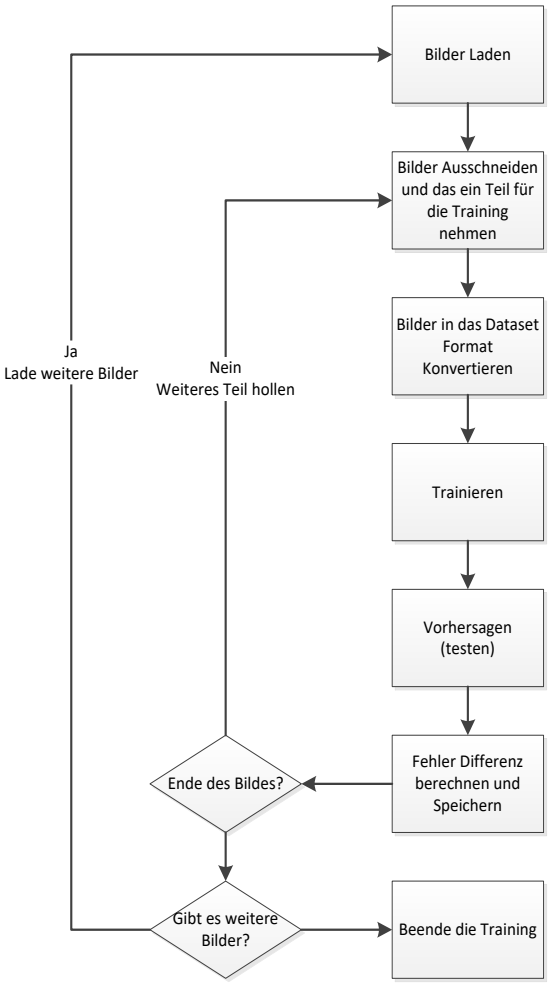


ABBILDUNG 26 ABLAUFDIAGRAMM

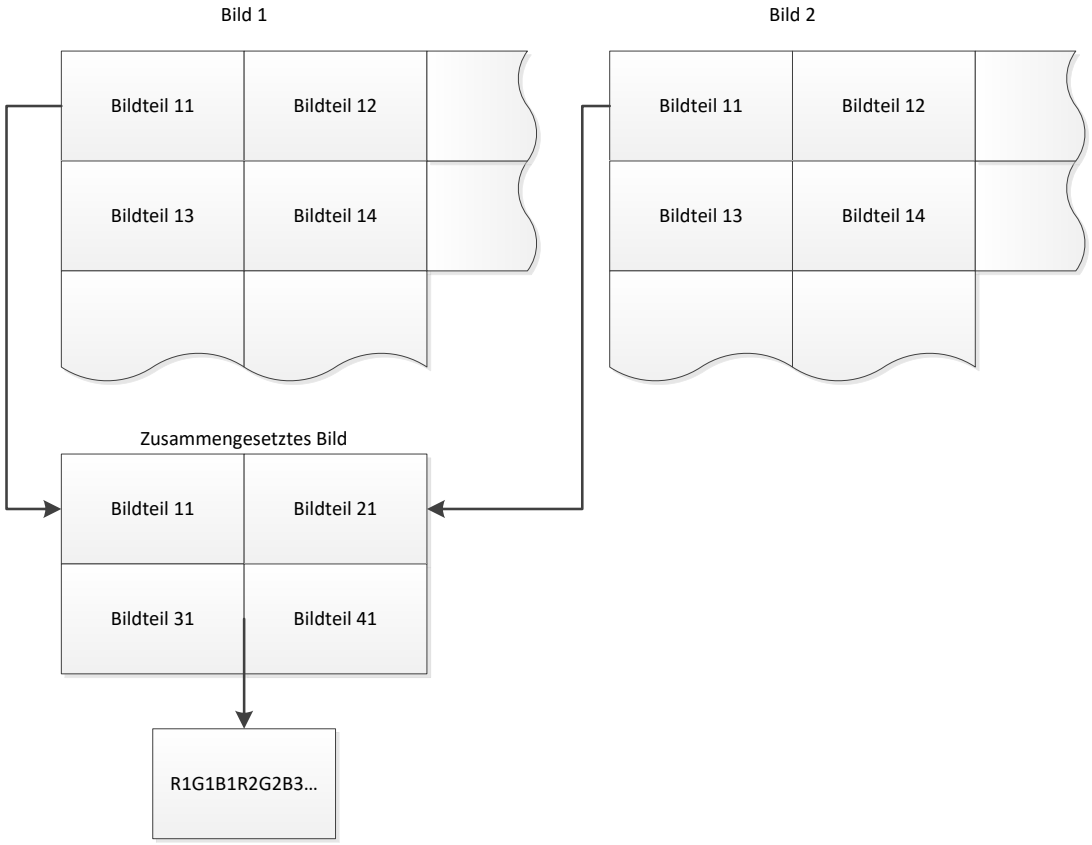


ABBILDUNG 27 BILDAUFTEILUNG

Danach wird das Array in die Matrix eingefügt, wo jede Zeile ist das Array von oben ist und enthält die Daten von allen Bildanteile die zum Training gehören.

2. Anwendung des oben genannten Algorithmus auf das Video

Zum Testen wurden 3 Vorhersagetests jeweils mit mehreren Einstellungen gemacht. Um das Verfahren schnell zu prüfen wurde der erster Test nicht mit dem Zukunftsframe getestet sondert mit einem Bild, das aus den 4 Inputbildern gewählt wurde. Dieser Test soll den Bereich zeigen, in dem die Vorhersage oder weitere Bildverarbeitung überhaupt funktioniert.

Der zweite Test wurde mit 5 Frames aus einem Video gemacht. Hier werden die vier ersten Frames als Input gegeben und das Letze fünfte Frame ans Output gegeben. Dieser Test soll zeigen, ob die Vorhersage in kleineren zeitlichen Abstand überhaupt funktioniert.

Der dritte Test basiert auf dem zweiten Test. Hier wird aber als Output nicht der nächste Frame gegeben, sondern die 10 nächsten Frames. So kann man die Maximale Vorhersagelänge für dieses Verfahren rausfinden.

1) Parameter und Vorhersage

Für die Prädiktion wurden mehrere Parameter und Varianten getestet, wie z.B. zwei versteckte Schichten oder viele Neuronen in einer versteckten Schicht. Auch wurde die Bildgrößen geändert um zu sehen, welche Größe am besten zur Vorhersage passt.

In diesem Abschnitt werden ein paar Parameter und deren Ausgaben aufgelistet:

Input 80x80 Output 40x40:

Um Ram Speicher zu sparen wurden für diesen Versuch nur die Graubilder benutzt. Leider kam für beide Auflösungen (40X40 & 80X80) kein korrektes Ergebnis raus:

Auflösung	Layer Anzahl	PSNR
80x80	6400-6400-6400	0.0125
80x80	6400 -6400-6400	9.34
40x40	1600-1600-1600	9.97

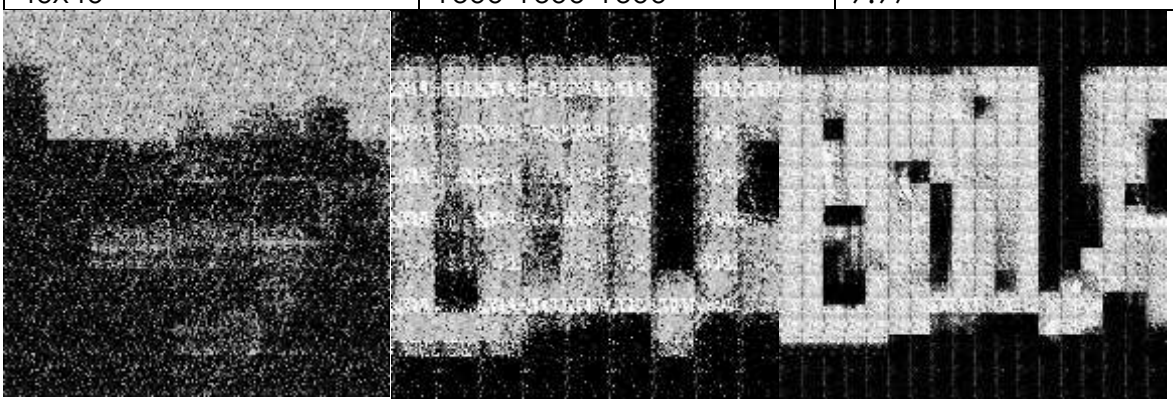


ABBILDUNG 28 AUSGANG VON 80X80 NEURONEN UND 40X40 NEURONEN INPUT

Input 32x32 Output 16x16:

Die 32x32 Abschnitte sind viel besser ausgefallen als 80x80 und 40x40. Für dieses Bereich wurde es mit mehrere Variationen von Neuronen Experimentiert. Die kleineren Abschnitte verkleinern den Vorhersagebereich und somit wird die Vorhersage verschlechtert.

Auflösung	Layer Anzahl	PSNR
30x30x3 (RGB)	2700-2700-2700	16.63
30x30x3 (RGB)	2700-2700-2700	17.97

ABBILDUNG 29 30X30X3 NEURONEN MIT (INPUT+OUTPUT)/2 FÜR HIDDENLAYER

Bei einer Vergrößerung der Anzahl der Neuronen wird der Ausgang schärfer und erhält einen höheren Kontrast. Um ein gutes Bild zu erzielen reicht eine Anzahl von Neuronen die zwischen (Input+Output) und (Input+Output)/2 liegt.

Auflösung	Layer Anzahl	PSNR
16x16x3 (RGB)	768-6144-768	18.99
16x16x3 (RGB)	768-6144-768	16.45

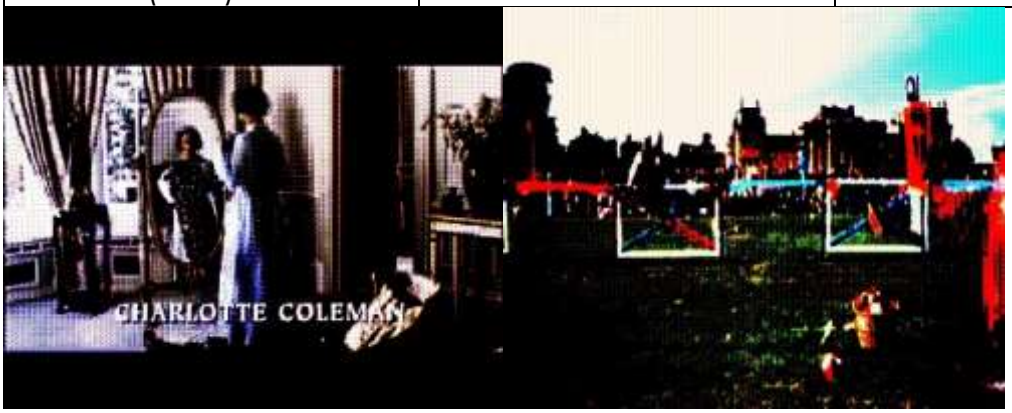


ABBILDUNG 30 16X16X3 NEURONEN MIT (INPUT+OUTPUT)X4 FÜR HIDDENLAYER

Beim Vergrößern der Anzahl der Hiddenlayer wird das Bild nicht mehr erkennbar und somit ungeeignet für die Vorhersage.

Auflösung	Layer Anzahl	PSNR
16x16x3 (RGB)	768-6144-1536-768	22.03
16x16x3 (RGB)	768-6144-1536-768	21.77

ABBILDUNG 31 16X16X3 NEURONEN MIT (INPUT+OUTPUT)X4 + (INPUT+OUTPUT) FÜR DEN HIDDENLAYER

Input 8x8 Output 4x4:

Bei einer Bildgröße die unter 8x8 liegt verbessert sich stark die Bildqualität. Die Vorhersage wird dadurch leider noch stärker verschlechtert.

Auflösung	Layer Anzahl	PSNR
8x8x3 (RGB)	192-192-192	39.5
8x8x3 (RGB)	192-192-192	31.52

ABBILDUNG 32 8X8X3 NEURONEN MIT (INPUT+OUTPUT)/2 FÜR DEN HIDDENLAYER

2) Verbesserung der Vorhersage

Um eine gute Bildqualität zu erziehen brauchen wir kleinere Systemantworten und um eine bessere Vorhersage zu erziehen, muss mehr Inputinformation in das System eingeführt werden. Also wird das im früheren Kapitel beschriebene Verfahren so abgeändert, dass es gleichzeitig mehr Eingangsworte und weniger Ausgangswerte bekommt.

Dafür werden in den Eingang die Teilbilder mit 40x40 oder mehr eingeführt. Das nächste Bild wird sich aber nur um ein paar Pixel von dem vorherigen Bild unterscheiden.

Für den Ausgang werden nur die oberen Pixel verwendet. Die sich wiederholenden Pixel werden entfernt. Dadurch haben wir den gleichen Anteil von Eingangs- und Ausgangsbilder, sie unterscheiden sich aber in der Größe.

Nach dem Zusammenbau haben wir dann ein Bild, das besser die Zukunft vorhersagt und dabei ein guterkennbares Bild ausgibt.

Durch die Verkleinerung der Schritte werden in das neuronale Netz mehr Objekte eingeführt als beim vorherige Verfahren mit 40x40. Durch die Mischung des 40x40 Eingangs mit dem 2x2 Ausgang wird der RAM schnell voll.

Um das zu vermeiden werden auch die Trainingsdaten in Gruppen aufgeteilt, so dass es zum Beispiel erstmal nur mit dem ersten 80x80 (2 mal 40) Bildteil trainiert, dann mit dem nächstem.

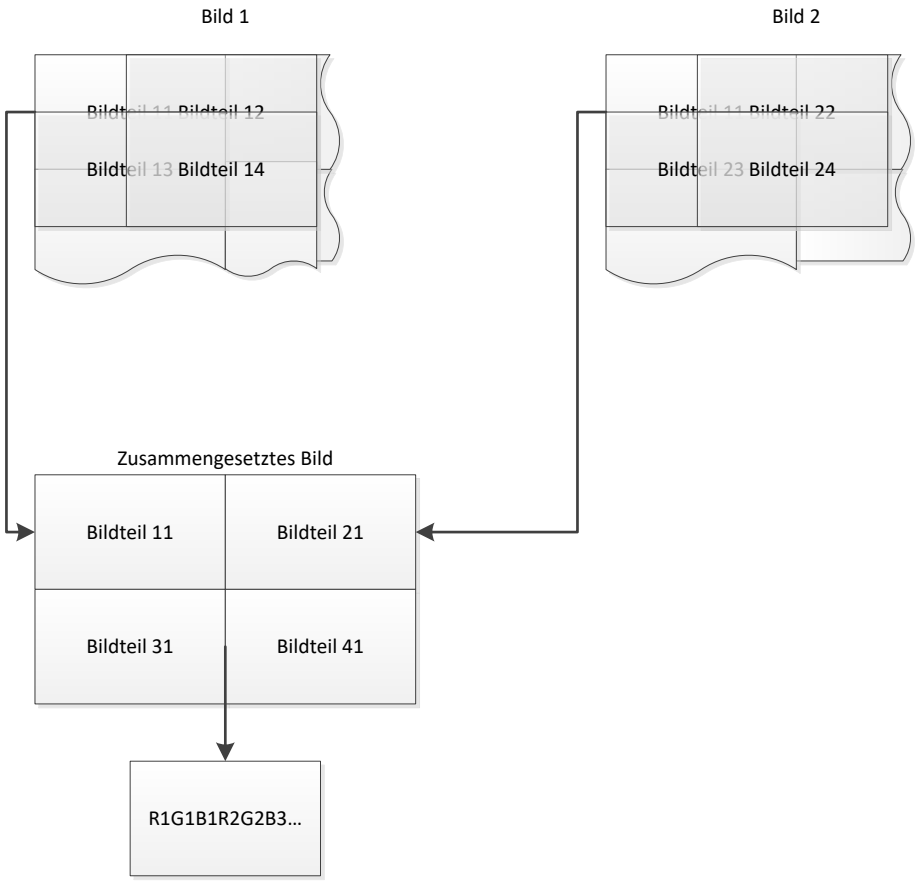


ABBILDUNG 33 BILDAUFTEILUNG BEI EINEM VERBESSERTEM VERFAHREN

3) Ergebnisse von dem verbessertem Verfahren

Für dieses Verfahren haben folgende Parameter gut gepasst:

Inputlayer: 40x40 (1600 Bildpunkte)

Hiddenlayer 1: Input+Output (16001 Bildpunkte)

Hiddenlayer 2: (Input + Output)/2 (800 Bildpunkte)

Outputlayer: 1 (1 Bildpunkt)

PSNR: 25.31 dB

Die Vergrößerung der Anzahl der Hiddenlayer verschlechtert das Bild an der Stelle wo sich Elemente sich bewegen.

Hier ist ein Beispiel mit der 1-Bild-Vorhersage:

Die stillstehenden Momente wurden noch gut erkannt. Das menschliche Auge erkennt fast keine Unterschiede an den vorhergesagten Stellen. Das Bild ist nur heller geworden und hat ein paar Artefakte. Die beweglichen Stellen sind aber verschwommen.

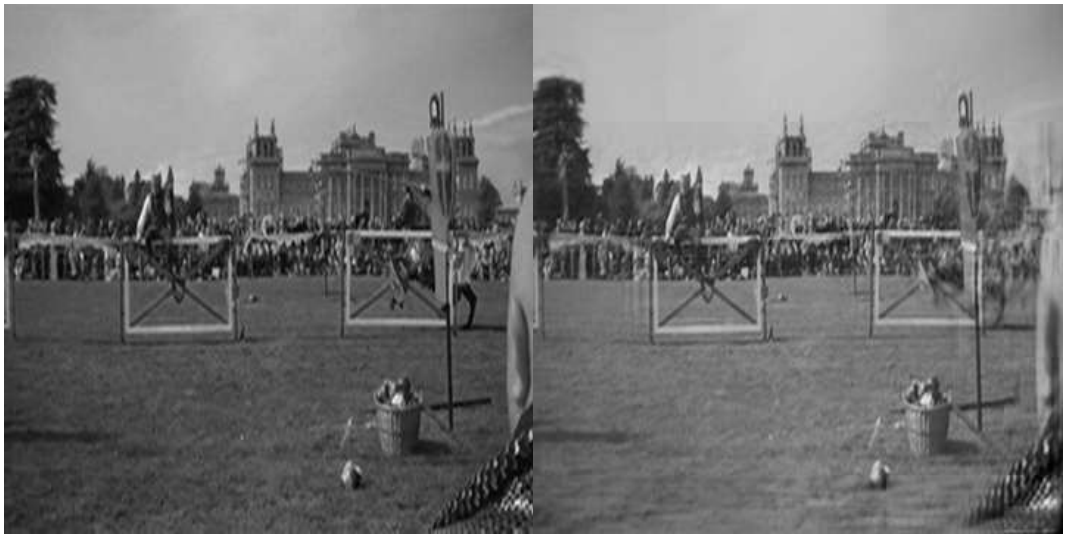


ABBILDUNG 34 LINKS ORIGENAL RECHTS VORHERGESAGTE

Erst nach der Differenzierung sieht man alle Unterschiede zwischen dem Original und dem vorhergesagten Bild.

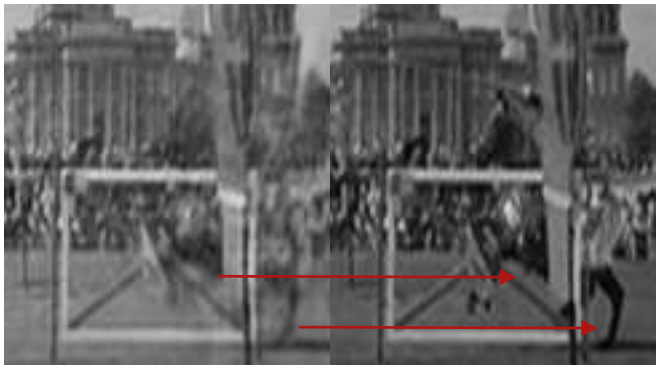


Abbildung 35 Vergleich von dem Beweglichem Objekt

Abbildung 36 Differenzbild zwischen dem Original und Vorhergesagten Bild

Bei der Vergrößerung erkennt man an dem Ausgangsbild auch das Bein von dem Pferd und auch den Bauchteil, der sich im vorherigen Bild bewegt hat.

Auch der Kopf hat sich bewegt und steht an derselben Stelle wie der Kopf in dem Originalbild. In diesem Fall kann man ihn besser erkennen als das Pferd im Hintergrund.

Abbildung 37 Vergleich das Kopf im Vordergrund

Die 10-Frames-Vorhersage sah genauso aus wie die 1-Frame-Vorhersage. Für die größere Vorhersage fehlt es an Inputinformationen. Man muss dafür mehr Inputneuronen verwenden und gleichzeitig auch mehr Bilder zur Vorhersage benutzen. Dafür braucht man aber auch mehr RAM.

4) Experiment mit einem kleinen Bild

In diesem Versuch wurde die Auswirkung von dem Netz auf ein sehr kleines Bild untersucht. So könnte man die Vorhörsage auf mehrere Stufen unterteilen, wo in der ersten Stufe wird erst ein kleines Bild vorhergesagt, dann mit Hilfe von dem Kleinem Bild soll ein Größeres Bild vorhergesagt werden und so weiter.

Für dieses Experiment wurde der Schritt mit Abschneiden umgegangen in dem die Ausschnittgröße genauso gestellt wurde wie das Eingangsbild.

Wegen RAM Overflow es wurde mit Bilder mit 40x40 Größe experimentiert. Ans Eingang wurde es 4 Bilder eingeführt und ans Ausgang nur 1.

Inputlayer: 80x80 (6400 Bildpunkte)
Hiddenlayer 1: (Input + Output)/2 (4000 Bildpunkte)

Outputlayer: 1 (1600 Bildpunkt)

PSNR: 35.86 dB

Da in so einem kleinen Bild sehr wenige Änderungen waren ist die Vorhersage in diesem Fall viel besser als es in vorherigen Versuchen war. Das Vorhergesagte Bild hat aber starkes Rauschen.

ABBILDUNG 38 ORIGENALBILD LINKS VORHERGESAGTE RECHTS

3. Quellkode Beschreibung

In diesem Abschnitt werden die wichtigen Teile aus dem Quellcode genauer beschrieben. Der Code wurde speziell so geschrieben, dass man nur anhand von Parameteranpassung jeden von oben genannten Verfahren einstellen kann.

1) Quellcode

```
#include <opencv2/videoio.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/ml.hpp>
#include <iostream>
#include <vector>
#include <QList>
#include <QFileInfo>
#include <QString>
#include <QDir>
#include <QDateTime>
#include <QDebug>
#include <QApplication>

using namespace cv;
using namespace std;

int mInputWidth, mInputHeight,mOutputWidth, mOutputHeight, mInputSize,
mOutputSize, tXBegin, tYBegin, tImageHeight, tImageWidth;

void SaveToImages3(QString xPath,int xTeiler, int xLayerSize, int s, int w,
int h, Mat xOutPutImages, bool xCutOff=false, int xFrame=1,double xError=0,
int xX=0, int xY=0)
{
    Mat image2;
    if(xLayerSize==1)
        image2=Mat::zeros(h,w,CV_32FC1);
    if(xLayerSize==2)
        image2=Mat::zeros(h,w,CV_32FC2);
    if(xLayerSize==3)
        image2=Mat::ones(h,w,CV_32FC3);
    if(xLayerSize==4)
        image2=Mat::zeros(h,w,CV_32FC4);
    int x=0;
    int y=0;
    int nameID=0;
    auto last=image2.ptr<float>(0);
    for( int r=0; r < xOutPutImages.rows;r++)
    {
        auto tList=xOutPutImages.ptr<float>(r);
        Mat tTempOut=Mat::zeros(s,s,image2.type());
```

```
        if(xCutOff)
        {
            tTempOut=Mat::zeros(s/xTeiler,s/xTeiler,image2.type());
        }
        last=tTempOut.ptr<float>(0);
        for (int col = 0;col<xOutPutImages.cols;tList++,last++,col++)
        {
            *last=(*tList)*255;
        }

        tTempOut.copyTo(image2(Rect( x*s/xTeiler, y*s/xTeiler, s/xTeiler,
s/xTeiler)));
        if(xCutOff)
        {
            if((y++)*s/xTeiler+s>=(h))
            {
                x++;
                y=0;
            }
            if(x*s/xTeiler+s>w)
            {
                QFile
outputFile(QDir().currentPath()+xPath+QString::number(xFrame)+".txt");
                outputFile.open(QIODevice::WriteOnly | QIODevice::Append);

                QTextStream out(&outputFile);

                out << QString::number(xError) << endl;

                outputFile.close();

                QString
Name=QDir().currentPath()+xPath+QString::number(xFrame)+".jpg";
                qDebug() << Name;
                Mat tImage;

                if(QFileInfo(Name).exists())
                    tImage = imread(Name.toStdString(),
CV_LOAD_IMAGE_GRAYSCALE);
                else
                    tImage =
Mat(tImageHeight*1.5,tImageWidth*1.5,image2.type(),255);
                image2.copyTo(tImage(Rect(xX,xY,image2.cols,image2.rows)));
                imwrite( Name.toStdString(), tImage );
                x=0;
                y=0;
            }
        }
        else
        {
            if((y++)*s/xTeiler+s>=(h))
            {
                x++;
                y=0;
            }
            if(x*s/xTeiler+s>w)
```

```
            {
                QString
Name=QDir().currentPath()+xPath+QString::number(++nameID)+".jpg";
                qDebug() << Name;
                imwrite( Name.toStdString(), image2 );
                x=0;
                y=0;
            }
        }
    }
}

QList<Mat> LoadImage2(QString Path, int xLayerSize,int file=1, int
filestoload=4)
{
    QList<Mat> tInput;

    int filenr=0;
    QString Name=QDir().currentPath()+Path+QString::number(file++)+".jpg";
    do
    {
        if(QFileInfo(Name).exists())
        {Mat temp;
            if(xLayerSize==1)
            {
                temp=imread(Name.toStdString(),CV_LOAD_IMAGE_GRAYSCALE);
                imwrite((Name+"BW.png").toStdString(),temp);// Speichere
Schwarzweises Bils für PSNR
            }
            else
                temp=imread(Name.toStdString());
            tInput.append(temp);
            Name=QDir().currentPath()+Path+QString::number(file++)+".jpg";
        }
    }while(QFileInfo(Name).exists()&&++filenr<filestoload);

    return tInput;
}

Mat normolizeImage(int m, int w, int h, QList<Mat> &tInput)
{
    int tH=(tInput[m].rows>h)?(h):(tInput[m].rows);
    int tW=(tInput[m].cols>w)?(w):(tInput[m].cols);
    Mat tInputImage=tInput[m](Rect( 0, 0, tW, tH));
    Mat tTempImage;
    tTempImage=Mat::zeros(h,w,tInputImage.type());

    tInputImage.copyTo(tTempImage(Rect( 0, 0, tW, tH)));

    return tTempImage;
}

Mat normolizeImage(int x, int y, int m, int w, int h, QList<Mat> &tInput)
{
    Mat tInputImage;
    int tH=(tInput[m].rows-y>h)?(h):(tInput[m].rows-y);
    int tW=(tInput[m].cols-x>w)?(w):(tInput[m].cols-x);
```

```
    if(tInput[m].rows-y>0&&tInput[m].cols-x)
        tInputImage=tInput[m](Rect( x, y, tW, tH));
    Mat tTempImage;
    tTempImage=Mat::zeros(h,w,tInputImage.type());
    if(tInput[m].rows-y>0&&tInput[m].cols-x)
        tInputImage.copyTo(tTempImage(Rect( 0, 0, tW, tH)));

    return tTempImage;
}

void MatToArray(QList<QList<float> > &tBWList, QList<Mat> tSplitedImage, int
xLayetSize)
{
    for(auto
tIterator=tSplitedImage.begin();tIterator<tSplitedImage.end();tIterator++)
    {
        Mat tQuad = (*tIterator).clone();
        auto tPtr=tQuad.ptr<uchar>(0);
        QList<float> tBWRow;
        for (int tY = 0; tY < tQuad.rows; tY++)
        {
            for (int tX = 0; tX < tQuad.cols*xLayetSize; tX++)
            {
                tBWRow.append((float)(*tPtr++)/255);
            }
        }
        tQuad.release();
        tBWList.append(tBWRow);
    }
    tSplitedImage.clear();
}

void SplitImages(int startPosX, int startPosY,int xTeiler, int xFrameCount,
int xLayetSize, int xSize, int h, int w, QList<Mat> &xInput,
QList<QList<float> > &xBWList, bool xCutOff=false, bool xFullImage=false)
{
    //Zusammenfüge x Bilder
    for(int m=0;m < xInput.length() - (xFrameCount-1);m++)
    {
        QList<Mat> tTempImage;
        for (int t = 0; t < xFrameCount; ++t) {
            tTempImage.append(normolizeImage(startPosX,startPosY, m+t,
mOutputWidth, mOutputHeight, xInput));
        }

        QList<Mat> tSplitedImage;
        int ColMax=(w/xSize)*xTeiler*sqrt(xFrameCount)-xTeiler-
(xFullImage)?(1):(0);
        int RowMax=(h/xSize)*xTeiler*sqrt(xFrameCount)-xTeiler-
(xFullImage)?(1):(0);

        for (int iCol = 0; iCol <= ColMax; iCol++)
        {
            for (int iRow = 0; iRow <= RowMax; iRow++)
            {
```

```
                Rect tRect = Rect( iCol*xSize/(sqrt(xFrameCount)*xTeiler),
iRow*xSize/(sqrt(xFrameCount)*xTeiler), xSize/sqrt(xFrameCount),
xSize/sqrt(xFrameCount));
                Mat tTemp(xSize,xSize,tTempImage.last().type(),255.0);s

                if(xCutOff)
                {

tTemp=Mat(xSize/xTeiler,xSize/xTeiler,tTempImage.last().type(),255.0);
                }

                for (int im = 0; im < tTempImage.count(); ++im)
                {
                    int x=tRect.width*(im%((int)sqrt(xFrameCount)));
                    int y=tRect.height*((int)im/((int)sqrt(xFrameCount)));
                    Mat dst_roi;
                    if(xCutOff)
                    {
                        dst_roi =
tTemp(Rect(x,y,tRect.width/xTeiler,tRect.height/xTeiler));
                        tRect.height=tRect.height/xTeiler;
                        tRect.width=tRect.width/xTeiler;
                    }
                    else
                        dst_roi = tTemp(Rect(x,y,tRect.width,tRect.height));

                    Mat tCopy=tTempImage[im](tRect);
                    tCopy.copyTo(dst_roi);
                    //if(xCutOff)
//                    {
//                        namedWindow( "Display window", WINDOW_AUTOSIZE );//
Create a window for display.
//                        imshow( "Display window", tTemp );
//                        waitKey(1000);
//                    }
                }
                tSplitedImage.append(tTemp);
            }
        }

        MatToArray(xBWList, tSplitedImage, xLayetSize);
    }
}

void ConvertToMat(int maxOutput, Mat responses, QList<QList<float>> tOutList,
Mat samples, int maxInput, QList<QList<float>> tInList, QList<Mat>
tResponses)
{
    int index=0;
    while(index < maxInput)
    {
        for(int s = 0; s < tInList.length(); s++)
        {
            auto t= tInList[s][index];
            samples.at<float>( Point( index, s ) ) = t;
        }
```

```
        index++;
    }
    index=0;
    while(index < maxOutput)
    {
        for(int s = 0; s < tOutList.length(); s++)
        {
            responses.at<float>( Point( index, s ) ) = tOutList[s][index];
        }
        index++;
    }

    tInList.clear();
    tResponses.clear();
}

int main(int argc, char** argv)
{
    QApplication a(argc,argv);
    QDateTime end ;
    QDateTime start = QDateTime::currentDateTime();
    qDebug() << "start time: " << start << "\n";

    // Parameter setzen
    int tChanalls=1;
    int tTeiler=1;//20;
    int tFrameanzahlgesammt = 4;// wie viel Bilder werden für Training geholt
    int tFrameanzahl=4;// wie viel Bilder werden für ein Input benutzt

    //Die erste Aufteilung
    mInputHeight=40*2;//120;//360;//576;//1008;
    mInputWidth=40*2;//120;//360;//704;
    mOutputHeight=40;//120;//360;//576;//1008;
    mOutputWidth=40;//120;//360;//704;

    //die Zweite Aufteilung
    mInputSize=40*2;//120;//40;
    mOutputSize=40;//120


    for (int var = 0; var < tFrameanzahlgesammt-tFrameanzahl+1; ++var)
    {
        tXBegin=tYBegin=0;
        //Bilder Laden
        qDebug() << "Load Image";
        QList<Mat> tInput=LoadImage2("/Input/", tChanalls, var+1);
        QList<Mat> tResponses=LoadImage2("/Responses/", tChanalls, var+1, 1);

        QList<QList<float>> tInList;
        QList<QList<float>> tOutList;
        tImageHeight=tInput[0].rows;
        tImageWidth=tInput[0].cols;

        int maxInput=mInputSize*mInputSize*tChanalls;
        int maxOutput=mOutputSize*mOutputSize*tChanalls/(tTeiler*tTeiler);
        int inputLayerSize = maxInput;
        int outputLayerSize = maxOutput;
```

```
        qDebug() << "Split Image";
        //Bilder auf kleinere Teile aufteilen
        SplitImages(tXBegin,tXBegin,tTeiler,tFrameanzahl, tChanalls,
mInputSize, mInputHeight, mInputWidth, tInput, tInLis, false, true);
        SplitImages(tXBegin,tXBegin,tTeiler,1, tChanalls, mOutputSize,
mOutputHeight, mOutputWidth, tResponses, tOutList, true);

        int numSamples = tOutList.length();

        //Bilder ins dataset konvertieren
        Mat samples( Size( inputLayerSize, numSamples ), CV_32F );
        Mat responses( Size( outputLayerSize, numSamples ), CV_32F );

        ConvertToMat(maxOutput, responses, tOutList, samples, maxInput,
tInList, tResponses);

        //Ebenen aufteilen
        vector<int> layerSizes = { maxInput, (maxInput+maxOutput)/2,
maxOutput };
        Ptr<ml::ANN_MLP> nnPtr;

        //Wenn die Parameterdatei vorhanden ist laden, sonst neue Netz
erstellen
        if(QFileInfo("Color.yml").exists())
        {
            cout << "Loading\n" << endl;
            nnPtr=Algorithm::load<ml::ANN_MLP>("Color.yml");
        }
        else
        {
            nnPtr = ml::ANN_MLP::create();

            nnPtr->setLayerSizes( layerSizes );
            nnPtr->setActivationFunction(cv::ml::ANN_MLP::SIGMOID_SYM, 1, 1
);
            nnPtr->setTrainMethod(ml::ANN_MLP::BACKPROP,0.000001,0.000001);

            TermCriteria tc;
            tc.epsilon=0.001;
            tc.maxCount=10000;
            tc.type = TermCriteria::MAX_ITER + TermCriteria::EPS;

            //nnPtr->setTermCriteria(tc);

            cout << "begin training:\n" << endl;
            if ( !nnPtr->train( samples, ml::ROW_SAMPLE, responses ) )
                return 1;
            cout << "end training:\n" << endl;
            nnPtr->save("Color.yml");
            QDateTime end = QDateTime::currentDateTime();
            qDebug() << "end training: " << end << "\n";
            qDebug() << "diff time training: " << start.msecsTo(end) << "\n";
        }

        Mat output;
```

```
        double tErrorDiff=0;
        double tError=1;
        int tDiff=1;

        double oldErrorDiff=0;
        do
        {
            // Das gelernte anwenden
            cout << "Predicting\n" << endl;
            nnPtr->predict( samples, output );
            cout << "Calculate Error:\n" << endl;

            tErrorDiff=0;
            tError=1;
            tDiff=1;
            for( int r=0; r < samples.rows;r++)
            {
                auto Out1=output.ptr<float>(r);
                auto Out2=responses.ptr<float>(r);
                for( int c=0; c < output.cols;c++)
                {
                    tErrorDiff+=abs(Out1[c]-Out2[c]);
                    if(Out1[c]!=0&&Out2[c]!=0)
                    {
                        tError+=Out2[c]/Out1[c];
                    }
                    tDiff++;
                }
            }

            tErrorDiff=tErrorDiff/tDiff;
            tError=tError/tDiff;

            cout << "Error Diff: " << tErrorDiff << endl;
            cout << "Error Factor: " << tError << endl;

            end = QDateTime::currentDateTime();
            qDebug() << "loop end time: " << end << "\n";
            qDebug() << "loop diff time: " << start.msecsTo(end) << "\n";

            //Bild und gelernte Speichern
            nnPtr->save("Color.yml");
            cout << "Saving\n" << endl;
            SaveToImages3("/Output/", tTeiler, tChanalls, mOutputSize,
mOutputWidth, mOutputHeight, output, true, var+1,tErrorDiff,tXBegin,tYBegin);
            cout << "Finished\n" << endl;
            oldErrorDiff=tErrorDiff;

            // Zum nechstem Bildteil rübergehen und wieder Trenieren
            tInList.clear();
            tOutList.clear();

            tXBegin+=mOutputWidth-mInputSize;
            if(tXBegin>tImageWidth)
            {
                tYBegin+=mOutputHeight-mInputSize;
                tXBegin=0;
```

```
                //wenn Bildteile durgearbeitet wurden aus der Schleife
rausgehen
                if(tYBegin>tImageWidth)
                    break;
            }

            SplitImages(tXBegin,tYBegin,tTeiler,4, tChanalls, mInputSize,
mOutputHeight, mOutputWidth, tInput, tInList, maxInput);
            SplitImages(tXBegin,tYBegin,tTeiler,1, tChanalls, mOutputSize,
mOutputHeight, mOutputWidth, tResponses, tOutList, maxOutput,true);

            ConvertToMat(maxOutput, responses, tOutList, samples, maxInput,
tInList, tResponses);

            cout << "begin training:\n" << endl;
            //if ( !nnPtr->train( samples, ml::ROW_SAMPLE, responses ) )
            //    return 1;
            cout << "end training:\n" << endl;
        }while(true);

        // Speicher freigeben
        for(auto i :  tInput  )
        {
           i.release();
        }

        for(auto i :  tResponses  )
        {
           i.release();
        }
        tInput.clear();
        tResponses.clear();
    }

    end = QDateTime::currentDateTime();
    qDebug() << "end time: " << end << "\n";
    qDebug() << "diff time: " << start.msecsTo(end) << "\n";
    return a.exec();
}
```

2) Parameter

Die meisten Parametervariablen stehen am Anfang von der Mainfunktion. Nur die Parameter von dem Neuronalem Netz stehen in der Nähe von dem Aufruf von dem Neuronalem Netz, das es die Werte von den vorherigen Funktionen für die Eingabe braucht.

tFrameanzahlgesammt:
Hier kann man Anzahl von Frames festlegen, die für das Training benutzt werden soll.

tFrameanzahl:
Die tFrameanzahl Variable sagt dem Programm wie viele Bilder gleichzeitig an die Inputschicht eingeführt werden.

tChanalls:
Sagt dem Programm ob es RGB Bild holen soll oder den schwarz-weißen Kanal.

mInputHeight, mInputWidth, mOutputHeight, OutputWidth:
Diese Variablen sagen dem Programm welchen Bereich von dem Bild wird in diesem Durchgang für das Training benutzt.

mInputSize, mOutputSize:
Mit Hilfe von dieser Variablen wird die Schneidegröße geregelt.

tTeiler:
Durch diese Variable kann man die Schrittweite einstellen. Der Schritt wird dann aus der abgeschnittenen Bildgröße errechnet.

Schritt = ***mInputSize/ tTeiler***

3) Bilder aufschneiden (SplitImages)

In diesem Abschnitt wird die Funktion SplitImages Erklärt.

Parameter:

startPosX, startPosY:

Mit Hilfe von dieser Variablen kann man die Starposition im Bild setzen. So kann man einzelne Bildabschnitte unabhängig voneinander behandeln.

h, w:

Ist Höhe und Breite von dem Bildabschnitt, das man bearbeiten will. So kann man auswählen welches Bereich wird jetzt für das Training benutzt.

xLayetSize:

Sagt ob es ein Schwarzweises Bild mit 1 Layer oder ein RGB Bild mit 3 oder ein RGBA Bild mit 4 Layer).

xSize:

Liegt fest groß sollen abgeschnittene Teile sein.

xTeiler:

Schrittweite Schritt = ***(h|w)/ xTeiler***

xFrameCount:

Wie viel Bilder sollen zusammengefasst werden.

xInput:

Liste mit Bildern die diese Funktion Bearbeiten soll.

xBWList:

Liefert eine QList<QList> mit Bildpunkte, die man später für den Netzinput benutzen kann.

xCutOff:

Wird für den Output von 2 Verfahren benutzt. Als Default Wert wird immer falsch zugewiesen.

xFullImage:

Steht für den dritten Verfahren. Als Default Wert wird immer falsch zugewiesen.

Am Anfang wird das geladene Bild mit Hilfe von der Funktion normolizeImage an die eingegebenen Parameter angepasst. Wenn das Bild kleiner ist, als die Eingegebene Bildgröße, dann wird das Bild geschnitten, wenn das Bild kleiner ist als die Eingegebene Bildgröße, dann wird der fehlende Teil zu dem Bild Eingefügt.

```
QList<Mat> tTempImage;
for (int t = 0; t < xFrameCount; ++t)
{
        tTempImage.append(normolizeImage(startPosX,startPosY, m+t,
                          OutputWidth, OutputHeight, tInput));
}
```

Dann werden die Bilder, die zusammengefasst werden sollen in einem Rechteck verteilt. Dafür wird es ausgerechnet, wie viel Bilder passen in die Horizontale und wie viel Bilder sollen passen in die Vertikale.

Mit Hilfe von der Variable „fulllmage“, kann man dem Programm sagen ob das Bild abgeschnitten werden soll oder nicht. Diese Variable wurde wegen dem Drittem Verfahren eingefügt. Sonst kommt 1 als Ergebnis und die Schleife läuft noch eine Extrarunde mit fehlender Information. Für den Ersten und den Zweiten Fall wird diese Runder für den Rand des Bildes benutzt.

```
int ColMax=(w/xSize)*xTeiler*sqrt(xFrameCount)-xTeiler-(fullImage)?(1):(0);
int RowMax=(h/xSize)*xTeiler*sqrt(xFrameCount)-xTeiler-(fullImage)?(1):(0);
```

Der Algorithmus läuft durch jedes Bild, das zum Input zusammengefasst werden muss. Mit Hilfe von 2 For-Schleifen werden die Positionen von den Bildern gesetzt.

Dann wird die Ausschneideregion ermittelt und in die Rect Variable gespeichert.

```
for (int iCol = 0; iCol <= ColMax; iCol++)
{
 for (int iRow = 0; iRow <= RowMax; iRow++)
 {
Rect tRect = Rect( iCol*xSize/(sqrt(xFrameCount)*xTeiler),
                   iRow*xSize/(sqrt(xFrameCount)*xTeiler),
                   xSize/sqrt(xFrameCount), xSize/sqrt(xFrameCount));
```

In diesem Teil wird die Matrix vorbereitet für den Bildteil vorbereitet. Um die Fehler besser nachvollziehen zu können, wird die Matrix mit Farbe gefühlt. So kann man merken ob am Ende die ganze Fläche mit dem Bildteil gefüllt wurde oder nicht.

Da im zweiten Verfahren für den Output ein kleineres Bild benutzt werden soll, es wird hier eine kleinere Matrix hergestellt.

Die Größe hängt von dem xTeiler Parameter. Die xCutOff Variable schaltet dann zwischen den 2 Matrizen um.

```
Mat tTemp(xSize,xSize,tTempImage.last().type(),255.0);

if(xCutOff)
{
 tTemp=Mat(xSize/xTeiler,xSize/xTeiler,tTempImage.last().type(),255.0);
}
```

In diesem Codeabschnitt werden Bilder abgeschnitten und zusammengefasst. Nach dem alle Bilder in einer Bildmatrix stehen wird diese Bildmatrix in die tSplitedImage liste eingefügt.

```
for (int im = 0; im < tTempImage.count(); ++im)
{
 int x=tRect.width*(im%((int)sqrt(xFrameCount)));
 int y=tRect.height*((int)im/((int)sqrt(xFrameCount)));

 Mat dst_roi;
 if(xCutOff)
 {
  dst_roi = tTemp(Rect(x,y,tRect.width/xTeiler,tRect.height/xTeiler));
  tRect.height=tRect.height/xTeiler;
  tRect.width=tRect.width/xTeiler;
 }
 else
 dst_roi = tTemp(Rect(x,y,tRect.width,tRect.height));

 Mat tCopy=tTempImage[im](tRect);
 tCopy.copyTo(dst_roi);
}
tSplitedImage.append(tTemp);
```

Am Ende von der Funktion werden einzelne Pixel mit Hilfe von der Funktion MatToArray in die Arrayform umgewandelt und in die QList<QList<float> > eingefügt.

V. Zusammenfassung

In dieser Arbeit wurde eine Vorhersage von Videosequenzen mit Hilfe von neuronalen Netz untersucht. Es soll anhand von mehreren Bilder aus einer Videosequenz, die in ein neuronales Netz eingeführt werden, ein Zukunftsbild gefunden werden.

In erstem Schritt wurden Information über den allgemeinen Aufbau von neuronalen Netzen untersucht und zusammengefasst. Um den Ablauf besser zu verstehen, wurde in diesem Schritt ein neuronales Netz programmiert und an einfachen Aufgaben wie „AND“, „OR“ und „XOR“ getestet.

Im nachfolgenden Schritt wurden die fertigen Programme und Bibliotheken für die Analyse des neuronalen Netzes ausgesucht. Aus dieser Sammlung von Programmen wurde eine für den Fall gut passende Bibliothek namens OpenCV ausgesucht, da OpenCV auch gut mit Bilder und Videos funktioniert, ist es ein sehr guter Kandidat für die Arbeit mit Videosequenzen.

Ein weiterer Kandidat für die Untersuchung von Videosequenzen mit Hilfe von neuronalen Netzen ist das Programm namens Matlab. Da es ein Programm und keine Bibliothek ist, kann man leider kein eigenständiges Programm schreiben.

Im nächsten Schritt wurden passende Daten ausgesucht. Hier wurden die allgemeinen Modellierungsfälle zusammengefasste und für die Aufgabe passende Datasets geladen.

Nach der Zusammenfassung der gesamten Informationen über neuronale Netz und das Deep Learning, wurden Artikel über fertige Projekte im Bereich Vorhersage mit Hilfe von neuronalen Netze ausgesucht.

Als nächstes wurde versucht, das vorhandene Wissen auf das Thema anzuwenden. Zuerst wurde versucht, ein Bild in das Dataset Format umzuwandeln.

Wegen der Vorhersage wurden an die Inputschicht 4 Bilder eingegeben. Da es zu einem Problem mit dem RAM Speicher führte, musste man den Fehler finden um die Datamenge zu reduzieren. In diesem Schritt wurde das Bild auf mehrere kleinere Teile aufgeteilt und als mehrere kleinere Bilder betrachtet.

Durch die Tests wurde gezeigt, dass die Ausgabe mit einer größeren Bildpunkteanzahl zu einem falschen Bildausgang führt. Mit einer kleineren Bildpunkteanzahl am Eingang bleibt sehr wenig Spielraum für die Vorhersage. Deswegen wurde das Programm so umgeschrieben, dass es am Eingang mehrere Bildpunkte haben kann, und am Ausgang so wenig wie möglich. Somit wurden die Bildqualität und die Prädiktion verbessert.

Als Dritte Variante wurde es wurde auch mit einem kleineren, nicht geschnittenen Bild experimentiert. Dieses Experiment sollte Zeigen ob man mit Unterteilung auf mehreren Stufen zu einem besseren Ergebnis kommen kann.

In der erster Stufe soll das kleine Bild Trainiert, dann das Ergebnis soll für die zweite Stufe benutzt werde. In diesem Schritt soll Ein Größeres Bild anhand von dem Kleinerem Bild und vorherigen Bilder Trainiert.

VI. Ausblick

Besseres Qualität

Um die Qualität zu verbessern und schnelle QHD Vorhersagen zu ermöglichen, kann man die ganze Vorgehensweise auf mehrere Stufen aufteilen.

In der ersten Stufe wird eine Vorhersage mit Bilder mit kleineren Auflösung durchgeführt.

Die nächste Stufe verbessert die Qualität der vorherigen Stufe indem man im zweiten neuronalen Netz die Ausgangswerte aus dem ersten neuronalen Netz zusammen mit dem Bild aus dem Bild von dem Input mit besseren Qualität einfügt.

In die Dritte Stufe kann man dann die Schwarzweise mit dem Neuronalem Netz erkannt Bilder mit der Hilfe von dem Farbigen Bild den Vorhergesagten Bild färben.

Beschleunigung und mehr RAM

Das Programm wurde mit QT auf Windows geschrieben. Leider kann das QT Compiler (MinGW), das ich benutzt hatte nur in den 32 Bits Format kompilieren. Das bringt Probleme mit dem Speicher. Man kann OpenCV für ein 64 Bit fähiges c++ Compiler kompilieren, dann hat man Möglichkeit mehr Neuronen zu benutzen und auch größere Bilder zu trainieren.

Um den Code zu beschleunigen kann man es an ein Paar stellen in Threads aufteilen und für das Netz die Grafikkarte zu benutzen.

VII. Quellverzeichnis

[1] „Einführung in Neuronale Netze," [Online]. Available: https://wwwmath.uni-muenster.de:16030/Professoren/Lippe/lehre/skripte/wwwnnscript/ge.html. [Zugriff am 08 07 2016].

[2] „Künstliches neuronales Netz," [Online]. Available: https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz. [Zugriff am 08 07 2016].

[3] „Neuronales Netz," [Online]. Available: https://de.wikipedia.org/wiki/Neuronales_Netz. [Zugriff am 08 07 2016].

[4] „Wie viele Neurone stecken im Gehirn?," [Online]. Available: http://www.abendblatt.de/ratgeber/wissen/article107575383/Wie-viele-Neurone-stecken-im-Gehirn.html. [Zugriff am 2016 07 08].

[5] G. D. Rey und K. F. Wender, Neuronale Netze, Huber, 2011.

[6] J. Heaton, Artificial Intelligence for Humans, Volume 1: Fundamental Algorithmus, Kindle.

[7] „An AI Watched 600 Hours of TV and Started to Accurately Predict What Happens Next," [Online]. Available: http://futurism.com/an-artificial-intelligence-program-watched-600-hours-of-youtube-videos-to-study-human-interaction/. [Zugriff am 31 07 2016].

[8] „Netflix bringt Maschinen das Videogucken bei," [Online]. Available: http://www.golem.de/news/sehen-wie-menschen-netflix-bringt-maschinen-das-videogucken-bei-1606-121345.html. [Zugriff am 30 07 2016].

VIII. Anhang

1. Ordnerstruktur

00_Software:

Dieser Ordner enthält die Passende Bibliotheken und Programmbeispiele die Sich mit Neuronalen Netzen und OpenCV befassen.

- **EmugoDLLs:** Enthält die Sammlung von OpenCV DLL'S
- **opencv-examples:** Enthält Beispiele Für Neuronale Netze
- **Ubuntu_Install:** Installation für Ubuntu
- **Windows_Install_VS:** Installation für Windows (Visual Studio C++)
- **How to install for QT**: Textdatei mit der Beschreibung der Installation für QT

01_XNeuron:

Enthält Project, das kopiert das Verhalten von dem Neuronalem Netz (ohne Bibliotheken).

02_OpenCV:

Tests mit der Benutzung von der OpenCV Bibliothek. Enthält auch das Hauptproject.

- **Emugo_C_Sharp**: Versuch den Neuronalen Netz von OpenCV mit C# zu starten.
- **OpenCV_Multiplatform_Test:** Enthält das Experimentale Project mit OpenCV und Neuronalen Netzen.
    - **build-ImageColor-Desktop_Qt_5_6_0_MinGW_32bit-Debug**: Das Kompilierte Project.
    - **image_Color**: Das Endproject mit der Vorhersage.
    - **Neuronal_Test_AND_OR_XOR:** Das erste Test das Zeigt ob NN Funktioniert oder nicht.
    - **Objecterkennung**: Versuch die Objekte im Bild zu erkennen.
    - **poseevaluator_dataset:** Test mit einem vertilgen Dataset.
    - **Test1, test2**: Waren die ersten Test um OpenCV zu Starten.
- **OpenCVData:** Die Vichtige Daten und Dll's für den den OpenCV (damit es in den Nähe von den Projekten liegt, um einfacher den Include zu machen)

03_OpenCV: Kompiliertes Programm

CD

GitHub Link:

https://github.com/XNeuron/Xneuron.git