

VEDUNET – FALTENDE NEURONALE NETZE ZUM SELBERMACHEN

Rares Sahleanu (15 J.)

Gunzenhausen 91710

Dr.-Ing. Ulrich Kiesmüller

Informatik/Mathematik

Jugend forscht

Bayern

2020/2021

Inhaltsangabe

1. Meine Idee

2. Vorgehensweise

2.1 Neuronen

2.1.1 Aufbau eines neuronalen Netzes

2.2 mathematische Implementierung von Neuronalen Netzen

2.3 Lernweise von Neuronalen Netzen

2.3.1 lineare Backpropagation

2.3.2 Gradientverfahren

2.3.2.1 Ableitung

2.3.2.2 Probleme des Gradientverfahrens

2.3.3 Weitere Überlegungen und Ideen

2.4 Faltende Neuronale Netze

2.4.1 Perzeptron

2.4.1.1 Einlagige Perzeptrons

2.4.1.2 Mehrlagige Perzeptrons

2.4.2 Pooling Layer

2.4.3 Convolutional Layer

2.4.4 Fully-Connected Layer

2.4.5 Wahrscheinlichkeit der biologischen Existenz

3. Experimente und Ergebnisse

3.1 Filter und Pooling

3.2 Funktionsoptimierung mit dem Gradientverfahren

4. Ergebnisdiskussion

5. Resume

6. Quellenangabe

7. Dienstleistungen

1. Meine Idee

Jeder hatte mal Kontakt damit - (faltende) neuronale Netze bzw. KIs – ohne sie würde die Welt, so wie wir sie kennen nicht existieren, da selbst die einfachste Google-Suche alleine schon mehrere Jahrzehnte Forschungsarbeit in diesem Bereich gefordert hat. Doch was machen KIs genau und wie funktionieren sie? Genau dieser Frage bin ich auf dem Grund gegangen, sodass ich mich mit maximaler Motivation an meinen Schreibtisch setzte und begann zu recherchieren. Nach ungefähr 2 Stunden dachte ich „So schwer ist es gar nicht“ – Das ich mich damit irrte wird sich noch später herausstellen... -. Nach 2 Wochen haarsträubender Eigenstudie merkte ich wie komplex-mathematisch die Welt dahinter aufgebaut ist und wieviel Arbeit und Willensstärke es braucht solche Systeme selber zu gestalten und zu entwickeln. Nachdem ich mich mit Experten, Uniprofessoren und Literatur unterhalten/beschäftigt habe begriff ich, dass der Lernprozess (in diesem Beispiel anhand eines Fehlers/Forschungsgegenstands) aus 5 festen Bestandteilen aufgebaut ist:

- I. Reproduktion
- II. Provokation
- III. Komparation
- IV. Observation
- V. Konklusion

In „I“ wird das dargestellte System reproduziert, also man versucht es zu rekonstruieren/simulieren. In „II“ versucht man ein bestimmtes Verhalten hervorzurufen, um aus diesen verwertbare Ergebnisse zu extrahieren, die in „III“ mit Idealwerten (Werte, so wie sie ein sollten – sog. Sollwerte) verglichen werden. In „IV“ wird jeder einzelne Schritt der Reproduktion aus „I“ dokumentiert (Debugging ist eine große Hilfe), sodass aus III und IV in V eine Schlussfolgerung gezogen werden kann, was zu verbessern ist.

Will man nun neuronale Netze richtig begreifen, sodass man sie tiefgründig hinterfragen und somit weiterentwickeln zu können, müssen wir uns an diesem Schema halten. „II“, „III“, „IV“ und „V“ sind durch Algorithmen, Debugging (in Fall IV), und menschlichem Verstand zu bewältigen, jedoch fällt es uns schwer „I“ zu realisieren. Wir müssen unser Konstrukt dynamisch, leicht, flexibel und schnell verändern können, was bei einem solchen Mammutprojekt nur schwer zu verwirklichen ist. Was wir bräuchten wäre etwas, was einem Sandbox-Spiel ähneln würde... etwas, was leicht zu bedienen sei und uns tiefe Einblicke in die Arbeitsweise unserer Systeme gibt, sodass wir sie Schritt für Schritt nachvollziehen können.

Genau hier kommt mein Projekt in Spiel!

Um die weltweite Entwicklung von neuronalen Netzen und KIs zu erleichtern, bzw. verständlicher zu machen, habe ich ein Projekt begonnen, welches folgender Maßen aufgebaut ist:

- Es soll ein graphisches Sandboxspiel (Visual-Educational Neuronal Networks = VeduNet)
- Man soll es durch (so viel wie möglich) Drag & Drop leicht bedienen, sodass man, wie beim Legospielen in der Jugend, neuronale Netze baut und rumexperimentiert kann
- Es soll intuitiv aufgebaut werden, sodass man nicht viel grübeln muss und man sofort weiß wie man es zu bedienen hat
- Es soll genau dokumentieren wie und was es gerade macht und durch ein bestimmtes Debug-System, soll man Tick für Tick (Die Selbsterstellten Zeitabschnitte im Projekt) nach vorne spulen können, sodass man in aller Ruhe sich die Prozesse anschauen und diese begreifen kann
- Man soll Neuronale Netze exportieren und importieren können, sodass man sich Templates und On-Download Projekte importieren kann.

Da ich an der FernUni in Hagen Mathematik und Informatik studiere, hatte ich Kontakt mit vielen Uniprofessoren auf diesem Bereich. Nach zahlreichen Gesprächen mit Kommilitonen und Professoren, kam ich zu dem Entschluss, dass solch ein Programm dringend notwendig wäre. Viele andere Studenten und Professoren meinten, dass die Erklärung im Internet/Uni sehr abstrakt sei, sodass dieses Thema für manche

schwer zu begreifen ist. Die Praxis sei auch erschwert, da es recht viel Aufwand kostet solche Netze zu programmieren, weswegen es nur sehr zäh und sperrig voran geht.

Während meiner Recherche rund ums Thema und das Projekt, fiel mir auf, dass es kein ansatzweise ähnliches Projekt im Internet gibt. Das, was meiner Idee/Vorstellung am nächsten käme, wäre PyTorch, wofür man aber Programmierkenntnisse braucht, was bei meinem Projekt nicht notwendig sein soll.

Da jede Studie/Forschung (ja sogar generell jede Handlung unseres Lebens) mit einem Ziel verknüpft ist, setzte ich mir als Ziel, an Informatik uninteressierten Menschen ein simples Zahlenerkennungsnetz verständlich zu erklären. Diese sollen anschließend selbst ein solches Netz schnell zusammenbauen und daran rumexperimentieren, sodass sie schnell begreifen, wie ein solches Netz funktioniert und welche Wichtigkeit die Mathematik dabei spielt. Es sollen zumindest Schüler ab der 10. Klasse die Möglichkeit haben, ohne enorm viel Aufwand, an Neuronalen Netzen zu forschen und sich eigene Interessen dafür zu bilden.

2 Vorgehensweise

Um ein solches Projekt zu entwickeln, müssen wir erstmal verstehen, wie neuronale Netze funktionieren und versuchen diese abstrakt selbst in der Informatik zu implementieren, sodass dieser Algorithmus für jedes Projekt unseres Nutzers funktioniert.

Als man Neuronale Netze erfunden hat, hat man sich an der Natur orientiert. Durch weitreichende Studien hat man erfahren, dass Gehirne Regionen besitzen, welche aus einzelnen Neuronen bestehen. Aber wie ist ein solches Neuron aufgebaut?

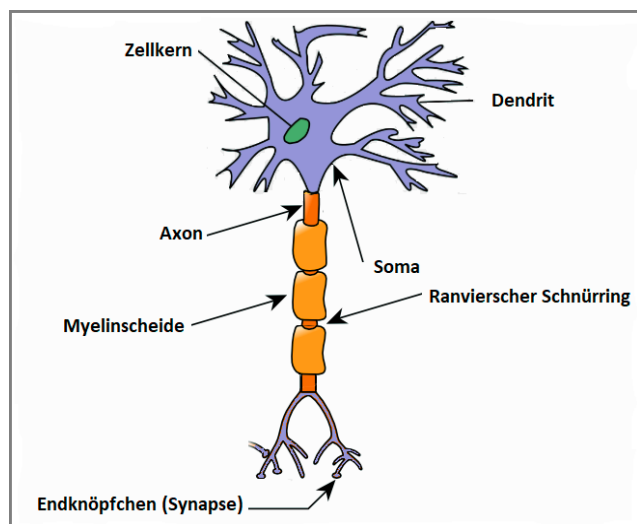
2.1 Neuronen

Ein Neuron besteht aus den Dendriten, welche aufgenommene Signale, die sich nur durch ihre Signalstärke unterscheiden, aufnehmen, und miteinander summieren. Anschließend wird ein

Signalimpuls über das Axon durch die Synapsen weitergegeben. Die Synapsen sind wiederum wieder an die Dendriten anderer Neuronen verbunden und geben so ihr Signal weiter.

Dabei ist wichtig zu erwähnen, dass mehrere Synapsen am gleichen Endneuron verbunden sein können. Dies hat den Effekt, dass sie in größerer Zahl aufsummiert werden und sie somit im Endeffekt „wichtiger“ sind als andere, da eine minimale Veränderung aufsummiert wird.

Also gibt es Dendriten, deren Signale wichtiger sind und daher linear vergrößert werden. Versuchen wir nun solch ein Neuron mit unserem Wissen mathematisch darzustellen:

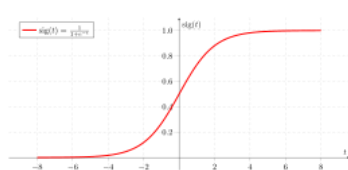


Beschriftetes Neuron

$$\sum_{i=0}^n a_i s_i$$

n ist die Anzahl der Dendriten, die ein Neuron besitzt und s_i ist die Signalstärke/Aktivierungsstärke, die das i -te Dendrit weitergibt. a_i repräsentiert die Wichtung. Eine Wichtung realisiert mathematisch die Wichtigkeit des, in diesem Beispiel i -ten, Dendrits. Umso höher die Wichtung, umso größer ist das Produkt zwischen Wichtung und Signal und umso mehr wird die Summe erhöht, was zu einer höheren Relevanz des Signals (gen. Aktivierung) führt, da die winzigste Änderung der Signalstärke schon bei einer hohen Wichtung einen mit a_i multiplizierten Unterschied des gesamten Ergebnisses verursacht. Unsere Neuronen sind nach einem bestimmten Algorithmus aktiviert. Dieser Algorithmus wird „Activation-function“ genannt und wie es der Name schon sagt, wird nach einer bestimmten Funktion ein Wert, der die Aktivierung darstellt, weitergegeben. Die gängigste Aktivierungsfunktion ist die Sigmoid Funktion. Diese Funktion hat 2 Häufungspunkte, sie konvergiert gegen 0 und 1 und ist stetig und streng monoton steigend. Die Summe der Signale wird in durch diese Abbildung auf \mathbb{R} abgebildet und schon haben wir unser

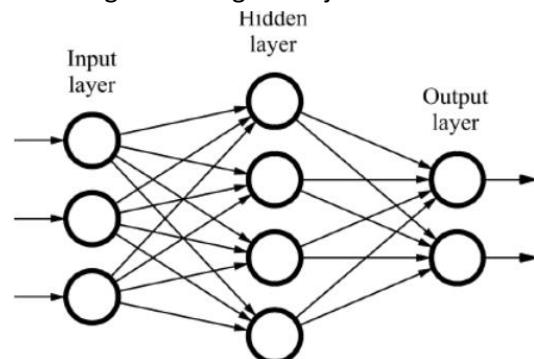
mathematisch beschriebenes Neuron:
$$\frac{1}{1 + e^{-(\sum_{i=0}^n a_i s_i)}}$$



Graph der Sigmoid Funktion

2.1.1 Aufbau eines Neuronalen Netzes

Neuronen werden standardgemäß in Schichten (sog. Layern) geordnet. Dies hat den Grund, dass nie im Gehirn einzelne Neuronen aktiviert werden, sondern Bereiche und Regionen. Jedes Neuron einer Schicht ist mit jedem Neuron der vorigen Schicht verbunden und gibt sein Signal an jedem Neuron der nächsten Schicht weiter. Eine wie gerade beschriebene Schicht wird Hidden-Layer genannt. Ausnahmen bilden der Input-Layer (dieser nimmt nur ein Signal auf und gibt es weiter mit der Wichtung „1.0“) und der Output-Layer (dieser repräsentiert die letzte Schicht und gibt das Signal an keine weitere Schicht weiter).



Neuronales Netz mit jeweils einem Hidden-, Input- und Outputlayer.

2.2 Mathematische Implementierung von Neuronalen Netzen

Ist es möglich ein Neuron durch eine Abbildung zu beschreiben, so muss es auch möglich sein ein ganzes Neuronales Netz durch eine Abbildung zu beschreiben. Wollen wir das in einer Programmiersprache implementieren, so müssen wir uns der Rekursion (Durch die verschiedenen Schichten und deren Neuronen) bedienen. Ich werde jetzt versuchen das mathematisch zu beschreiben:

$$sig : \mathbb{R} \rightarrow \mathbb{R} : x \mapsto \frac{1}{e^{-x}}$$

$$neuron(l, n) = sig\left(\sum_{i=0}^{N_{l-1}} (a(i, l-1, n, l) * neuron(i, l-1) + b(l, n))\right)$$

Okay... Das ist recht viel auf einmal. Lasst mich erstmal $a(x, m, n, l)$ erklären welche Funktionen und $neuron(l, n)$ Konstanten ich benutzt habe. Ist die Funktion welche den Wert annimmt, welcher das n-te $b(l, n)$ Neuron in Layer l ausgibt. N_l ist die Anzahl der Neuronen in der l-ten Zeile. Ist die Wichtung von dem Signal, welches von dem x-ten Neuron im m-ten Layer zum n-ten Neuron in dem l-ten Layer verläuft. Repräsentiert den Bias. Das Bias ist eine Art Schwellenwert, ab welchem das Neuron aktiviert ist. Wie man sieht führt sich die Funktion in sich selbst nochmal aus, Dies wird in der Informatik Rekursion genannt und wird uns häufiger in unserem Neuronalen Netz verfolgen. Zur Vereinfachung werde ich in dieser Arbeit als Ergebnis des $nn(I)$ Neuronalen Netzes verwenden. Diese Funktion nimmt als Wert eine Matrix an (bzw. Vektoren) und verrechnet jedes einzelne Element als Eingabe im Inputlayer. Das Ergebnis ist eine Matrix (bzw. Vektor oder einzelner Wert) welche aus den einzelnen Werten der Neuronen des Outputlayers aufgebaut ist.

Bis hierhin war alles leicht nachvollziehbar und schlussfolgernd, aber der eigentliche Fels kommt jetzt.

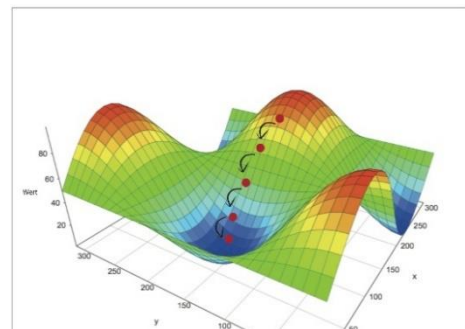
2.3 Lernweise von Neuronalen Netzen

Gegeben ist, dass Neuronale Netze lernen können. Aber wie können sie lernen? Daher, dass alles vordefiniert und festgelegt ist (also Algorithmen etc.) gibt es da nicht viel zu ändern, jedoch bin ich nicht auf die Wichtungen genauer eingegangen. Die Wichtungen sind das einzige variable an einem Neuronalen Netz und sie stellen sozusagen das erlernte Wissen dar, welches unser Neuronales Netz erlangt. Diese werden zu Beginn zufällig ausgewählt, sodass Jede Neuron-Neuron-Verbindung durch eine zufällige Wichtung beeinflusst wird. Folglich liefert $a(x, m, n, l)$ unsere Funktion eine zufällige Zahl, welche wir durch „Lernen“, so verändern, dass sie den Fehler des Neuronalen Netzes verringert. Doch was ist der Fehler?

Als Fehler versteht man bei Neuronalen Netze, die Abweichung des Sollwerts von dem Ist-Wert. Wir definieren $r(I)$ als den Sollwert des Inputs I . $e(I)$ ist die Funktion welche die Abweichung des Sollwerts von dem Ist-Wert zurückgibt. Diese ist definiert durch: $e(I) = nn(I) - r(I)$. Was ist also unser Ziel? Unser Ziel ist es ja ein Neuronales Netz so zu programmieren, dass die Abweichung des Sollwerts von dem Ist-Wert so gering wie möglich ist, also sollen unsere Gewichte so gestaltet werden, dass $e(I) = 0$ ist. Das ist in der Realität schwer möglich, weshalb wir versuchen werden $e(I)$ so gering wie möglich zu halten. Für so etwas hilft uns ein/mehrere Algorithmen, welche unter dem Begriff Backpropagation zu finden sind. Doch versuchen wir unsere Fehlerfunktion zu überarbeiten. Weil wir nicht von Einzelfällen ausgehen wollen, da solche eher messfehleranfällig sind, werden wir hier mehrere Errors aus mehreren Durchläufen in den kompletten-Error ein zu berechnen.

$$E = \frac{1}{2} \sum_{i=0}^n (t_i - o_i)^2$$

E entspricht dem Fehler, welcher bei allen Versuchen in der Summe entstanden ist. n ist die Anzahl der dem Neuronalen Netz vorgestellten Muster. t_i ist der Sollwert (target) des i-ten Musters und analog dazu ist o_i der Ist-Wert (Ausgabe/Output), welcher bei dem i-ten Versuchen erzielt wurde. Die Quadrierung und der Faktor 0.5 existieren zur Vereinfachung der Ableitung, auf welche ich später eingehen werde. Versuchen wir uns nun am einfachsten Backpropagationsalgorithmus, welcher linear ermittelt, welche Wichtungen um wieviel verändert werden müssten, um sich dem Sollwert anzunähern:



Gradientverfahren in einem 3-dimensionalem Raum visualisiert.

2.3.1 Lineare Backpropagation

Hier verfolgt man einer bestimmten Philosophie, welche folgendes besagt:

„Wenn ein Neuron n Verbindungen besitzt, welche insgesamt von n Wichtungen beeinflusst werden einen Fehler e erzeugt, dann bekommt die Wichtung die meiste „Schuld“, welche das Neuron am meisten beeinflusst. Die anderen Wichtungen erhalten demnach proportional viel „Schuld““.

Was heißt das? Das heißt, dass wenn bspw. ein Neuron 2 Wichtungen besteht, dann wird der Fehler der i -ten Wichtung wie folgt berechnet: $w_1 = e \frac{w_1}{(w_1 + w_2)}$. w_n ist die n -te Wichtung und e ist NUR in diesem Beispiel der Fehler des Neurons.

Verallgemeinernd kann man sagen:

$$(w_i)_{neu} = \frac{w_i}{e \sum_{i=0}^n w_i + o_{i-1}} \quad (e_{i-1}) = \frac{e_i}{e \sum_{i=0}^n w_i + o_{i-1}}$$

Wie man sieht, habe ich diesen Error (Schuld) pro Summanden (Wichtung * Neuron von letzter Schicht) nochmal unter sich proportional aufgeteilt. Dies bewirkt, dass ein proportionaler Teil des Errors an den vorigen Layer weitergegeben wird und so alle Wichtungen des gesamten Netzes angepasst werden. Man könnte diesen Error der Wichtung und dem Signal des vorigen Neurons gleichsetzen, doch das hätte den Effekt, dass wir möglicherweise übers Ziel hinausschießen und den Target „verfehlen“. Zudem bekommen die Bias dahingegen den proportionalen vollen Error, da sie keinen 2. Faktor o. ä. besitzen und so e_{i-1} nicht existiert.

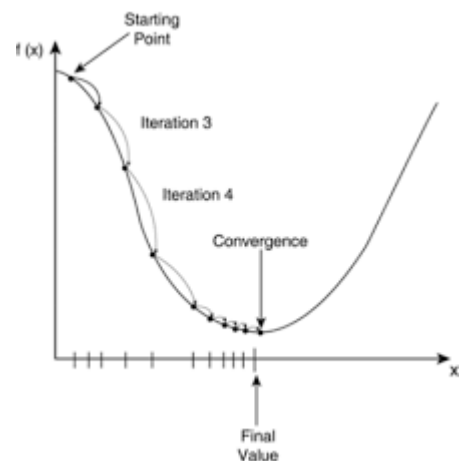
2.3.2 Gradientverfahren

Die Minimierung der Error-Funktion... Das hört sich an wie eine typische Optimierungsaufgabe. Wir können aber schwer die Funktion gleich 0 setzen, da sie so komplex wäre, dass man sie praktisch nicht lösen könnte. Für solche Situationen macht man sich das Gradientenverfahren zu Nutzen und verfolgt folgende Strategie:

Wie suchen uns zufällig irgendwelche Koordinaten auf dem Graphen und „steigen“ langsam die Funktion ab. Diese Methode ist sehr effektiv, jedoch hat sie eine „Macke“, auf die ich später zu sprechen kommen werde, aber erstmal folgende Frage: „Wie wissen wir wo es „Bergab“ geht?“.

Nun ja, dazu helfen uns die Ableitungen oder besser gesagt: Der Gradient. Der Gradient ist ein Differentialoperator, welchen man sich wie folgt vorstellen kann:

$$\text{grad}(f) = \frac{\partial f}{\partial x_1} \hat{e}_1 + \dots + \frac{\partial f}{\partial x_n} \hat{e}_n = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$



Gradientverfahren an einem Graphen visualisiert.

Es ist also nicht mehr als ein Spaltenvektor, welche für jede Reihe, die Ableitung einer Funktion nach einer bestimmten Variablen enthält. Unser imaginärer „Ball“ (Ball, weil dieser auch Bergabrollt), dem wir zufällige Koordinaten zugewiesen haben, wird bspw. Immer 2 Einheiten in die Richtung „rollen“, dessen Steigung in dem Punkt am größten ist. Hier habe ich, um dieses Verfahren natürlicher wirken zu lassen, in meinem Programmcode 2 Verhaltensweisen implementiert. Die erste nimmt wie bereits gesagt die größte Steigung (GradientType.DIRECT), während die zweite (GradientType.SMOOTH) den Ball in alle Richtungen bewegt, jedoch von der Höhe der Steigung abhängig. Hier sieht man auch wieso ich in 2.3 die Funktion quadriert und mit $\frac{1}{2}$ multipliziert habe. Dies hat allein differenzialtechnische Gründe. Die Quadrierung hat die Wirkung, dass die Gesamte Funktion mindestens so groß wie 0 ist, sodass die $f(x_1, x_2, x_3 \dots) \geq 0$ Programmierung des Gradientenverfahrens erleichtert wurde, da ich von dem Fall ausgehen kann. Der Faktor 0.5 dient lediglich zur Vereinfachung der Ableitung. Aber wie kann ein Computer Ableiten?

2.3.2.1 Ableiten

Ich werde nun im Folgenden den Algorithmus der Ableitung beschreiben. Er war recht zäh und kompliziert, da die Ketten-, Quotienten-, Summen- und Produkt-Regel implementiert werden sollte. Die Substitution war ein treuer Begleiter, welcher mir viel Arbeit erspart hat. In *de.rare.visnet.api.math.derivative.derivative.Parser* kann man auch die Funktion `splitAdd()` und `splitMult()` finden, welche mir auch eine Menge geholfen haben. Der Algorithmus ist wie folgt aufgebaut:

1. Aufteilung von f in dessen Summanden.
2. Wenn die Summanden mehr als 1 sind, dann ist es eine Summe (`element.Sum`)
3. Aufteilung der Summanden in Faktoren
4. Wenn die Faktoren mehr als 1 sind, dann ist es ein Produkt (`element.Product`)
5. Substitution der einzelnen Klammern mit „v“
6. Patternmatching der Muster
7. Vordefinierte Ableitung der Verschiedenen „Elementen“ anhand deren Patterns

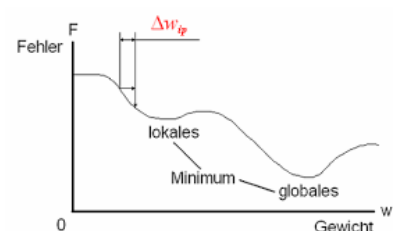
Beispiel:

Wir haben die Abbildung $f : \mathbb{R} \rightarrow \mathbb{R} : n, l \mapsto x^2 * x * x$, welche Wir ableiten wollen. Es gibt 1 Summanden, weswegen es keine Summe gibt. Es gibt 3 Faktoren, weswegen es ein Produkt ist. Nach der Definition der Produktregel: $2x * x * x + x^2 * x + x^2 * x = 4x^3$

Ich hätte mir diese 2 Wochen Arbeit ersparen können, indem ich die Werte nur annähere (Δx und $\Delta f(x)$ vordefinieren, dann minimieren), jedoch wäre das sehr unmathematisch und es können Fehler entstehen.

2.3.2.2 Probleme des Gradient Verfahren

Das Gradientenverfahren weist ein großes Problem auf, welches schwer zu lösen ist. Das Gradientenverfahren findet lediglich lokale Extrempunkte, aber wie habe ich versucht es zu lösen? Nun ja, eine formale Lösung hierfür gibt es nicht. Man kann es nicht lösen, sondern versuchen die Fehler zu minimieren:



visualisiertes Problem des Gradientenverfahrens

- Anstatt einen zufälligen Punkt zu definieren, von dem aus wir unseren Weg $n * n$ nach unten bahnen, erstellen wir ein Netz aus Kugeln, die alle gleichzeitig und unabhängig voneinander runterrollen. Der Ort, an dem sich die tiefste Kugel befindet ist unser lokales Minimum. Dieser Optimierungsversuch ist skalierbar, jedoch nicht sehr effizient. Dieses Netz ist einfach ein Grid, also ein Gitter aus Kugeln, welche in gleichmäßigen Entfernungen aufgereiht werden und somit die Chance ein globales (oder tieferes) Minimum zu erhöhen.
- Wir fügen jeder Kugel eine bestimmte Trägheit hinzu. Diese Trägheit wird berechnet, aus der vorigen Trägheit mal einer gewissen Zerfallskonstante.

2.3.3 Weitere Überlegungen und eigene Ideen

Daraufhin fing ich an zu überlegen welche Optionen es noch gäbe Extrema zu finden. Um Rechenleistung erheblich zu sparen, werden wir uns das Taylorpolynoms genauer anschauen:

(Eine Überlegung die den Rahmen sprengen würde, wäre es neuronale Netze als LGS anzunähern und diese dann einfach und algorithmisch in Abhängigkeit von Parametern zu lösen, sodass die neuronalen Netze damit lernen.)

Taylorpolynom

Wir sollten erstmal versuchen die Funktion vereinfachen um sie schneller und leistungssparender zu analysieren. Ein Polynom wäre natürlich fantastisch und genau hier kommt die Formel von Brook Taylor ins Spiel. Dieser „Algorithmus“ welcher eine Funktion mit Polynomen annähert. Da wir nur einen begrenzten Bereich der Funktion brauchen, oder besser gesagt, analysieren können, wird die Annäherung, welche auf begrenztem Bereich >98% genau ist, ausreichen. Das Taylorpolynom ist wie folgt definiert:

$$T_n f(x; a) = \sum_{k=0}^n \frac{f^{(k)}(a)}{k!} (x - a)^k$$

Das einzige Problem ist es nur, das ganze multidimensional zu verwirklichen. Ich entwarf also ein Algorithmus, aber kurze Zeit später fand ich auf der Website der Uni-München einen Eintrag mit Formel drüber, weswegen ich den zur Hilfe nehmen werde:

$$T(x_1, \dots, x_d) = \sum_{n_1=0}^{\infty} \dots \sum_{n_d=0}^{\infty} \frac{(x_1 - a_1)^{n_1} \dots (x_d - a_d)^{n_d}}{n_1! \dots n_d!} \frac{\partial^{n_1}}{\partial x_1^{n_1}} \dots \frac{\partial^{n_d}}{\partial x_d^{n_d}} f(a_1, \dots, a_d)$$

Hier kann man das Gradientenverfahren anwenden. Die Ableitung ist jetzt sehr einfach und die Speicherung von Neuronalen Netzen in bestimmten Zuständen/Zeiten ist jetzt auch simpel und nicht Rechenleistung brauchend.

2.4 Faltende Neuronale Netze

Um Gesichter u. ä. zu erkennen braucht man allerdings noch mehr als ein simples neuronales Netz. Um Gesichter (abstrakter: Muster) in Bildern zuverlässig zu erkennen benötigen, wir noch einige algorithmische Layer. Diese Layer werden Convolutional Layer genannt, da sie ein Neuronales Netz welches in einer zweidimensionalen Ebene geordnet ist in eine 3-dimensionale überführt. Dies passiert, in dem man ein Bild auf verschiedene Aspekte untersucht und diese dann parallel mit den Originaldaten in den nächsten Layer einspeist. Dies bewirkt, dass Neuronen nicht mehr mit Zahlen rechnen, sondern mit Vektoren (die wiederum aus Zahlen bestehen), welche Informationen über die Umgebung, Farbverlauf deren Konzentration etc. enthalten. Convolutional Layer können auch zur Datenreduktion beitragen, worauf ich später zu sprechen kommen werde. Zuallererst werde ich verschiedene Arten von Convolutional Layern aufzählen und vorstellen.

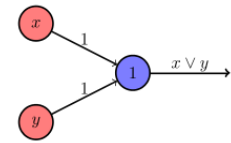
2.4.1 Perzeptron

Ein Perzeptron ist im Prinzip ein vereinfachtes künstliche neuronales Netz welches mit anpassbaren Wichtungen und einem Schwellwert verbunden ist. Unter diesem Begriff werden heute verschiedene Kombinationen des ursprünglichen Modells verstanden. Man unterteilt diese in einlagige und mehrlagige Perzeptrons.

2.4.1.1 Einlagige Perzeptrons

$$o_j = \begin{cases} 1 & \sum_i w_{ij}x_i + b > 0 \\ 0 & \text{ansonsten} \end{cases}$$

Ein Perzeptron ist also recht simpel und leicht zu verstehen. Ich habe noch ein Bild von einem 1 lagigen Perzeptron beigefügt, damit man sieht wie simpel und logisch diese sind. Dieses Perzeptron realisiert das logische ODER und wird recht häufig in der Neuroinformatik benutzt. Um Perzeptrons zu „belehren“ gibt es einen simplen Algorithmus:

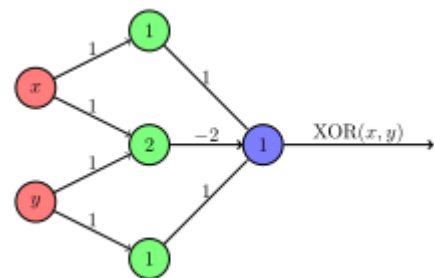


einlagiges Perzeptron

1. Wenn die Ausgabe eines Neurons 1 (bzw. 0) ist und den Wert 1 (bzw. 0) annehmen soll, dann werden die Gewichtungen nicht geändert.
2. Ist die Ausgabe 0, soll aber den Wert 1 annehmen, dann werden die Gewichte inkrementiert/erhöht.
3. Ist die Ausgabe 1, soll aber den Wert 0 annehmen, dann werden die Gewichte dekrementiert/verringert.

2.4.1.2 Mehrlagige Perzeptrons

Mehrlagige Perzeptrons sind ähnlich aufgebaut wie einlagige nur, dass es einen Unterschied gibt: Mehrlagige Perzeptrons haben mehr Layer (wer hätte es gedacht). Dies löst Probleme, die einlagige Perzeptoren einschränkten. Mehrlagige Perzeptrons werden für:



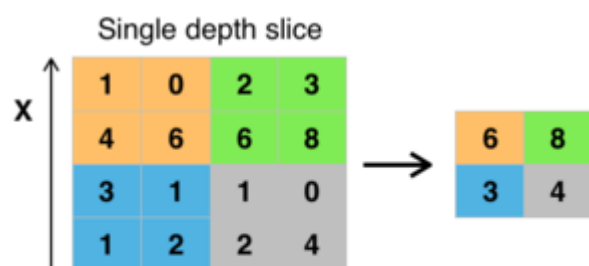
Mehrlagiges Perzeptron

- Short-cuts. Bei dieser Topologie werden einige Neuronen nicht nur mit allen Neuronen der nächsten Schicht verbunden, sondern darüber hinaus mit weiteren Neuronen der übernächsten Schichten.
- Fully-connected-Layer, bei welchen die Neuronen einer Schicht mit allen Neuronen der direkt folgenden Schicht verbunden werden.
- Rekurrente neuronale Netze, wobei die Ausgänge der Neuronen mit derselben oder (einer der) vorigen Schicht(-en) verbunden werden.

2.4.2 Pooling layer

Dieser Layer dient lediglich zur Datenreduktion. Überschüssige Informationen werden verworfen, sodass die benötigte Rechenleistung verringert wird. Ein klassisches Beispiel hierfür ist das Max-Pooling, welches aus $n * n$ einem Quadrat das aktivste

$n * n$ Neuron herauspickt und es repräsentativ für alle Neuronen einsetzt. Dieses Quadrat rutscht dann bei jedem Durchgang um m Neuronen nach rechts und Randelemente (also Elemente die außerhalb unseres Sichtfelds liegen) werden meistens mit 0 gekennzeichnet, aber hierfür gibt es mehrere Variationen.



Beispiel eines Poolinglayers (hier Maxpool) an einer 4*4 großen Eingabematrix

Hier sieht man ein Maxpooling Filter der Größe 2x2 welcher immer 2 Schritte nach rechts geht. Alternativ kann der Filter aber auch (nicht so wie in diesem Beispiel) zum Teil außerhalb des Zielobjekts stehen (je nach dem wo der Filter anfängt und welche Schrittgröße es besitzt).

Während ein klassisches zweilagiges Perzeptron mit jeweils 1000 Neuronen pro Ebene für die Verarbeitung von einem Bild im Format 32 × 32 insgesamt 2 Millionen Gewichte benötigt, verlangt ein CNN mit zwei sich wiederholenden Einheiten, bestehend aus insgesamt 13.000 Neuronen, nur 160.000 (geteilte) zu lernende Gewichte, wovon der Großteil im hinteren Bereich (fully-connected Layer) liegt.

Die zweite Option bildet das Dropout-Verfahren, welches einen bestimmten Anteil der Neuronen einfach deaktiviert (Output =0), weswegen sie im nächsten Layer nicht berücksichtigt werden,

2.4.3 Convolutional Layer

Convolutional Layer vergleichen eine Eingabematrix (meistens Graustufen eines Farbbildes) mit einem bestimmten Muster, welches vorgegeben ist und durch „lernen“ angepasst werden kann.

Convolution Kernel

Convolutional Layer verwenden ein Muster, welches in einer Matrix beschrieben wird. Diese iteriert wie das Pooling Layer durch die gesamte Inputmatrix und vergleicht die somit extrahierte Untermatrix mit einem vorgegeben Muster.

Hat man eine Eingabematrix (bspw. Graustufen eines Teils eines Bildes) und ein convolution kernel welche wie folgt mit einander verrechnet werden.

$$\sum_{y=0}^{hoehe_{Eing.mat.}} \left(\sum_{x=0}^{breite_{Eing.mat.}} (e_{x,y} m_{x,y}) \right)$$

Am Ende kommt ein Wert raus, welcher angibt, wie sehr die Eingabematrix dem convolution kernel ähnelt. Setzt man diesen Wert dann in eine eigene Matrix und wiederholt das mit den anderen Werten, so erhält man eine Matrix, die angibt, wie sehr sie an bestimmten Punkten bestimmten Mustern ähnelt.

Beispiele von convolution kernel:

- Glättungsfilter, Mittelwertfilter

$$\frac{1}{9} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Der Glättungsfilter findet Flächen.

- Schärfungsfilter

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Der Schärfungsfilter findet stellen an denen sich die Farbe sehr stark verändert.

- Kantenfilter, Laplace

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Der Kantenfilter findet Kanten

- Relieffilter

$$\begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

Erzeugt im Bild einen Effekt, bei dem die Kanten wie Erhöhungen und Vertiefungen erscheinen.

Welche Ergebnisse diese (und andere) Filter gegenüber dem Bild im Endeffekt haben werden wir in 3.1 sehen.

2.4.4 Fully-Connected Layer

Nach einigen sich wiederholenden Einheiten bestehend aus Convolutional und Pooling Layer, geht das Netzwerk in einen mehrlagigen Perzeptron über. Das Ende eines CNNs bildet eine Softmax-Funktion (bzw. eine Wahrscheinlichkeitsverteilung)

2.4.5 Wahrscheinlichkeit der biologischen Existenz.

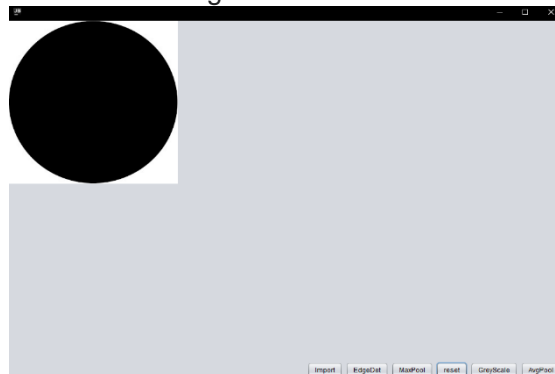
Die biologische Plausibilität der Backpropagation erwies sich in der Biologie als sehr gering, jedoch ist die Wahrscheinlichkeit der biologischen Existenz von der Kantenerkennung als sehr hoch, da durch fMRT (funktionelle Magnetresonanztomographie) gemachte Untersuchungen auf Kantenerkennung und Aktivierungsschichten (= unsere Layer) hinweisen.

Daher, dass das meiste in den vorigen Kapiteln recht logisch war, habe ich mich entschieden die Filter und die Funktionsoptimierung weiter zu untersuchen und diese dann genauer zu klassifizieren.

3 Experimente und Ergebnisse

3.1 Filter und Pooling

Um diese Experimente durchführen zu können, habe ich mir hierfür ein kleines Programm geschrieben, welches meine Filter und Algorithmen auf ein Bild anwendet und es abbildet:



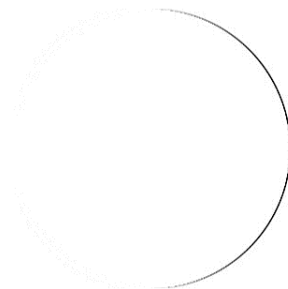
Das graphical user interface meines Programms mit unserem Beispielbild oben links.

Link: <https://github.com/RaresBares/JF-Pooling>

Jeder Filter/Algorithmus lässt sich anpassen, weshalb wir jetzt gleich loslegen:

Zuallererst hat es mich brennend interessiert, wie Edgedetection-Filter angewendet werden. Ich werde nun links die Filter und die Ergebnismatrix (visualisiert durch die Werte dargestellt als Dunkelheit auf weißem Hintergrund) darstellen.

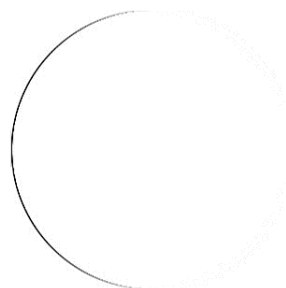
$$\begin{pmatrix} 1 & -1 & 0 \\ 1 & -1 & 0 \\ 1 & -1 & 0 \end{pmatrix}$$



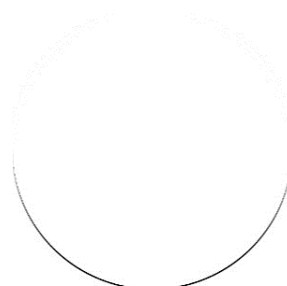
$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & -1 & -1 \\ 0 & 0 & 0 \end{pmatrix}$$



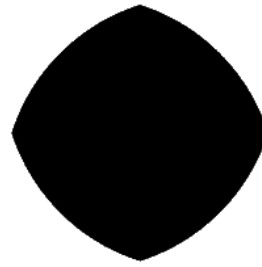
$$\begin{pmatrix} 0 & -1 & 1 \\ 0 & -1 & 1 \\ 0 & -1 & 1 \end{pmatrix}$$



$$\begin{pmatrix} 0 & 0 & 0 \\ -1 & -1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$



(MaxPool)



Durch diese Versuche können wir jetzt sehen, dass diese Matrizen Kanten aufspüren. diese werden benutzt um Formen zu detecten. In Verbindung mit MaxPool wird das ganze erleichtert, denn dieses „rundet“ die halbschrägen Kanten zu schrägen bzw. geraden Kanten ab. Wenn wir diese 4 Matrizen zu einer zusammenführen können wir jede Art von Kanten (vertikal-linksseitig, vertikal-rechtsseitig, horizontal-oben, horizontal-unten) herausfinden. Hat ein Element periodische Kanten (wie zum Beispiel Haare), so kann der Maxpool Algorithmus diese zu einer großen Kante zusammenführen. Als Konkurrent zum Maxpool gibt es den AvgPool, welcher sich dadurch unterscheidet, dass er nicht das Neuron mit der höchsten Aktivität substituiert, sondern einen Durchschnitt der EingabeMatrix bildet und diesen mit ihm selbst assoziiert/substituiert.

$$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$$

AvgPool an einer 3x3 Matrix

3.2 Funktionsoptimierung

Als nächstes wollte ich mit dem Feature rumspielen, mit dem Gradientenverfahren Minima zu finden. Als Funktion nahm ich eine 3-dimensionale Parabel:

$$f(x, y) = x^2 + (y + 2)^2$$

Beispielfunktion

Ich gab dies in meinen selbstprogrammierten Gradientenrechner (inkl. Ableitungsrechner) ein und es kam raus:

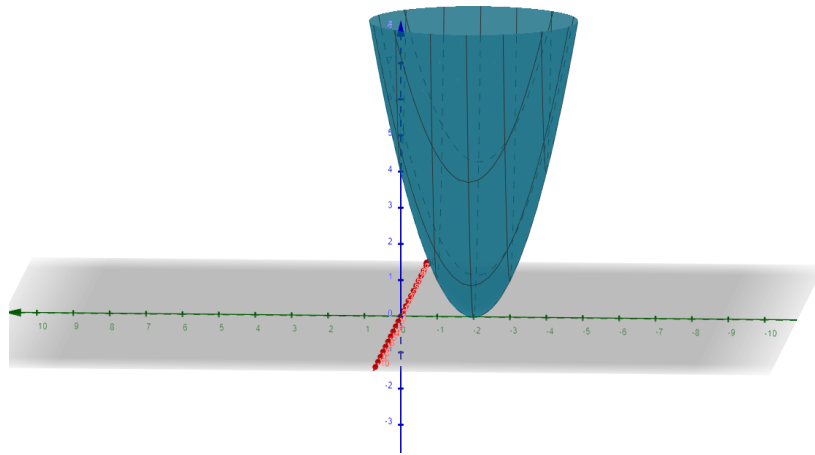
```
Ableitung nach x: (2)*((x)^(2-1))
Ableitung nach y: (2)*((y+2)^(2-1))
```

Ausgabe meines Ableitungsalgorithmus.

$$\text{grad}(f) = \begin{pmatrix} 2 * x \\ 2 * (y + 2) \end{pmatrix}$$

Der daraus resultierende Gradient aus der Funktion

Ich habe zur Visualisierung der Abbildung noch vorigen Graphen hier abgebildet:



Visualisierter Graph unserer Beispielfunktion

Ich habe die „Kugel“ bei $(4.0, 3.0, f(4, 3))$ anfangen lassen und es kam raus:

```
Minimum bei: x = -0.0000000 und y = -2.0000000
```

errechnetes lokales Minimum dieser Funktion

Dies entspricht einer Genauigkeit von über 99,999975%, was heißt, dass mein Algorithmus nicht nur funktioniert, sondern genauer ist, als manch Taschenrechner diesen Wert abbilden kann.

Dieser Algorithmus funktioniert mit einer n-dimensionalen Abbildung, was heißt, dass er sich (wie oben bereits erklärt) perfekt für das Training von Neuronalen Netzen eignet. Die Genauigkeit lässt sich mit den „steps“ einstellen. Diese Steps geben an wie viele Schritte dieser Ball „rollt“, also wie oft die oben beschriebene Näherung durchgeführt wird. Ich habe für die oben vorgeführte Funktion mit 400 Steps 2.6 Sekunden gebraucht, was bedeutet, dass es recht lange dauern würde bis wir eine Funktion mit mehreren Millionen Variablen optimieren.

4. Ergebnisdiskussion

Ich habe mehrere Wochen damit verbracht alles so weit wie möglich in der Informatik zu implementieren und es ist mir geglückt ein kleines Neuronales Netz zu erstellen und dieses dann zu trainieren. Den Sourcecode habe ich auf GitHub hochgeladen unter

<https://github.com/RaresBares/VisualNetwork>

File/Folder	Description	Last Update
.github/ISSUE_TEMPLATE	Update issue templates	last month
.idea	[Derivation update]	13 days ago
API	[Working Gradient method]	12 days ago
Example	+ Examples	25 days ago
Experiments	[Working Gradient method]	12 days ago
VisNet	Add files via upload	25 days ago
assets/img	Implementing vectors and matrices	last month
README.md	Update README.md	12 days ago
VisualNetwork.iml	expanded Math-API and fixed some Bugs.	last month
pom.xml	Implementing vectors and matrices	last month

ToDo: API

- ✓ matrices and vector
- ✓ derivation
- ✓ gradient
- ✓ calculator
- ✓ algorithms for learning and optimization
- ✗ GPU support
- ✗ activation functions
- ✗ convolutional layers
- |-> ✗ algorithms
- ✗ KeypointDetections (FAST, Jarros, Shi)
- ✗ SVM
- ✗ k - nearest neighbours (with weights)
- ✗ implementing multithreading
- ✗ Code cleanup
- ✗ visual interface

[Homepage des Repositories auf Github](#)

Wie man sieht gibt es noch viel zu tun und das mit Abstand am wichtigsten ist höchstwahrscheinlich der GPU support, sodass man die Neuronalen Netze schneller und damit effektiver trainieren kann.

Der Code befindet sich im API-Modul.

5. Resume

Nach stundenlanger Arbeit und 15.000 geschriebenen Zeilen Code kann ich sagen, dass das Programmieren eines Neuronalen Netzes sehr komplex ist. Es ist eine Sache, so etwas anhand eines Beispiels zu programmieren, aber es ist um Dimensionen schwerer, so etwas als Sandboxspiel zu programmieren. Dies hat den Grund, dass man das Ganze abstrahieren muss um diese abstrakten Algorithmen mithilfe von anderen Algorithmen an einem x-beliebigen Beispiel anzupassen. Der mathematische Aufwand dahinter ist enorm und das ist noch nicht alles, da meine Arbeit nur eine Hand voll der Themen war, welche unter dem Kapitel „Deep Learning“ zu finden sind.

Da ich Mathematik/Informatik studiere, werde ich mich weiterhin mit dem Thema so intensiv wie möglich befassen und nicht lockerlassen, bis mein Neuronales Netz das erste Auto erkennt oder die beste Spielertaktik in Schach oder Computerspielen. Als große Hauptherausforderung für die Zukunft nehme ich mir mit meinem Sandboxprogramm gegnerische Spieler in Computerspielen zu

klassifizieren, lokalisieren und autonom diese anzugreifen. So oder so, Neuronale Netze werden immer wichtiger. Sie sind vom Alltag kaum noch wegzudenken und mein Lernprogramm wird jungen Informatikern in dieser Hinsicht enorm helfen.

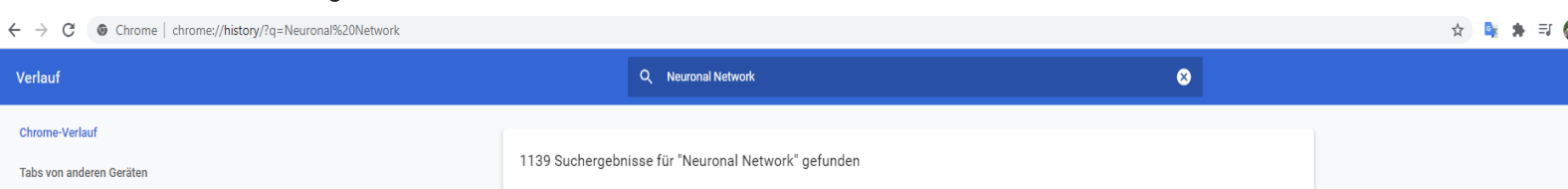
6. Quellen- und Literaturverzeichnis

Art der Quelle / Unterstützungsleistung	Quellenangabe / Angabe von Unterstützungsleistungen	Quelle gefunden am / Unterstützung in Anspruch genommen am
Buch	Florian Modler, Martin Kreh: „Tutorium Analysis 1 und Lineare Algebra 1“, Hannover/Deutschland 2018, S. 283 - S. 303	Empfehlung eines Kommilitonen am 20.11.2020
Buch	Valentina Emilia Balas, Snjiban Sekhar Roy, Dhamendra Sharma, Pijush Samui: „Handbook of Deep Learning Applications“, UNKNOWN 2019, S. 21 – S. 83 und S. 160 – S. 201 und S. 293 – S. 333	Gefunden in Quellenangabe einer Website am 22.11.2020
Buch	Tariq Rashid: „Neuronale Netze selbst programmieren: Ein verständlicher Einstieg mit Python (Animals)“, Heidelberg/Deutschland 2017, S. 1 – S. 92 und S. 179 – S. 199	Gefunden im Internet auf der Suche nach Fachbüchern am 22.11.2020
Internetseite	https://www.minet.uni-jena.de/fakultaet/schukat/ME/Scriptum/ , Prof. Dr. E.G. Schukat-Talamazzini, „VORLESUNG im SOMMERSEMESTER 2020“, über tiefere akademische Einblicke in die Arbeitsweisen von maschinellem Lernen	Gefunden am 23.11.2020 auf Empfehlung von Dr.-Ing. Kiesmüller
Internetseite	http://www1.inf.tu-dresden.de/~ds24/lehre/ml1_ws1415/ , Prof. Dr. Ursula M. Staudinger, „Index of /~ds24/lehre/ml1_ws1415“, über tiefere akademische Einblicke in die mathematischen Tricks hinter maschinellem Lernen	Gefunden am 23.11.2020 auf Empfehlung von Dr.-Ing. Kiesmüller
Internetseite	http://www1.inf.tu-dresden.de/~ds24/segm_2/segm_res.html , Prof. Dr. Ursula M. Staudinger, „Segmentation“, über die KeyPointDetection bzw. Segmentation von Bildobjekten	Gefunden am 23.11.2020 auf Empfehlung von Dr.-Ing. Kiesmüller
Internetseite	https://de.wikipedia.org/wiki/Gradient_(Mathematik) , Wikimedia Foundation Inc., „Gradient (Mathematik)“, über die Errechnung des Gradient	Gefunden am 23.11.2020
Internetseite	https://de.wikipedia.org/wiki/Gradientent , Wikimedia Foundation Inc., „Gradientverfahren“, über die Funktionsweise des Gradientverfahren.	Gefunden am 24.11.2020
Internetseite	https://de.wikipedia.org/wiki/Gradientenverfahren , Wikimedia Foundation Inc., „Gradientverfahren“, über die Funktionsweise des Gradientverfahren.	Gefunden am 24.11.2020
Internetseite	https://www.physik.uni-muenchen.de/lehre/vorlesungen/wise_09_10/t0/uebung/taylor.pdf , Ines Eiban, „Taylorreihe für Funktionen mehrerer Variablen“, über die Errechnung des mehrdimensionalen Taylorpolynoms	Gefunden am 26.11.2020

Art der Quelle / Unterstützungsleistung	Quellenangabe / Angabe von Unterstützungsleistungen	Quelle gefunden am / Unterstützung in Anspruch genommen am
Internetseite	https://www.youtube.com/watch?v=aircAruvnKk , Grant Sanderson , “ But what is a Neural Network? Deep learning, chapter 1 ”, über den Aufbau und Funktionsweise von neuronalen Netzen	Gefunden am 1.12.2020
Internetseite	https://www.youtube.com/watch?v=IHZwWFHWa-w , Grand Sanderson, „ Gradient descent, how neural networks learn Deep learning, chapter 2“, Fortsetzung von Teil 1	Gefunden am 2.12.2020
Internetseite	https://www.youtube.com/watch?v=llq3gGewQ5U , Grand Sanderson, „What is backpropagation really doing? Deep learning, chapter 3“, Fortsetzung von Teil 2	Gefunden am 2.12.2020
Internetseite	https://www.youtube.com/watch?v=tIeHLnjs5U8 , Grand Sanderson, „Backpropagation calculus Deep learning, chapter 4“, Fortsetzung von Teil 3	Gefunden am 2.12.2020
Internetseite	https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz , Wikimedia Foundation Inc., „Künstliches neuronales Netz“, Aufbau neuronaler Netze	Gefunden am 3.12.2020
Internetseite	https://de.wikipedia.org/wiki/Perzeptron , Wikimedia Foundation Inc., „Perzeptron“, über die Typologie und Funktionsweise eines Perzeptrons	Gefunden am 3.12.2020
Internetseite	https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron , Wikimedia Foundation Inc., „Künstliches Neurons“, über die Typologie und Funktionsweise von künstliches Neuronen	Gefunden am 3.12.2020
Internetseite	https://de.wikipedia.org/wiki/Convolutional_Neural_Network , Wikimedia Foundation Inc., „Convolutional Neural Network“, über die Typologie und Funktionsweise von künstlichem neuronalen faltenden Netz	Gefunden am 4.12.2020
Internetseite	https://de.wikipedia.org/wiki/Faltungsmatrix , Wikimedia Foundation Inc. „Faltungsmatrix “, über die Typologie und Funktionsweise von convolutional kernels	Gefunden am 6.12.2020
Internetseite	https://de.wikipedia.org/wiki/%C3%9Cberanpassung , Wikimedia Foundation Inc., „Überanpassung “, über das Phänomen des Overfittings	Gefunden am 7.12.2020

Art der Quelle / Unterstützungsleistung	Quellenangabe / Angabe von Unterstützungsleistungen	Quelle gefunden am / Unterstützung in Anspruch genommen am
Internetseite	http://www.mi.uni-koeln.de/wp-znikolic/wp-content/uploads/2019/06/11-Odenthal.pdf , Dr. Zoran Nikolić, "Convolutional Neural Networks", über den groben Aufbau eines Neuronalen Netzes	Gefunden am 7.12.2020
Internetseite	https://www.youtube.com/watch?v=sxwoQIAkLBw , Brotcrunsher, "Neuronale Netze [028] - Convolutional Layers", über die Funktionsweise von Neuronalen Netzen.	Gefunden am 7.12.2020

Ich habe im Laufe meiner Forschung auf 300+ Quellen zurückgegriffen. Diese aufzulisten würde nur den Rahmen sprengen. Hinzu kommt, dass ich mich nicht die Daten erinnern kann, an denen ich jede Website aufgerufen habe



Screenshot meines Suchverlaufs

Die Quellen die ich oben notiert habe, sind die, die die wichtigsten waren und welche mir am meisten in Erinnerung geblieben sind

7. Dienstleistungen

1. Dr.-Ing. Ulrich Kiesmüller ist Lehrer am Simon-Marius-Gymnasium. Er half mir mit Ratschlägen über den Segmentation-Algorithmus und der Ableitung und Matrizentypologie und empfahl mir Quellen und Univorlesungen die ich mir durchlesen sollte sowie vermittelte er mich mit Uniprofessoren.
2. Prof. Dr. Marc Berges ist Uniprofessor an der Universität Elangen-Nürnberg. Er half mir indem er mir Server zur Verfügung stellte, mit denen meine Experimente wesentlich schneller abliefen und das Training von meinem kleinen neuronalen Netz erleichtert wurde.
3. Fin Liegner ist Schüler an einem Gymnasium in der Nähe von Hamburg (genauere Informationen auf Anfrage). Er empfahl mir Formeln und jegliche mathematische Ausdrücke in Latex zu übersetzen.