# Einleitung

Im ersten Bericht wurde über verschiedene Machine Learning Methoden untersucht, mit welchen es möglich ist, Cyberattacken zu finden. Dieser Bericht soll nun die Grundlagen liefern, wie man eine konkrete Umsetzung realisieren kann und dies wird an hand eines ersten beispiel dargestellt.

# Machine Learning Programmpakete

Da sich speziell im Bereich von Deep Learning in Cyberbereich einiges tut wurde dieser Bereich zur näheren Untersuchung ausgewählt. Die Untersuchung der Programmpakete ist jetzt aber nicht nur auf Deep Learning beschränkt, sondern gilt Allgeimein.

## Rahmenbedingungen

Es wird davon ausgegangen, dass sich der Einsatz der trainierten Modell auf folgenden Betriebssystemen erfolgen soll.

* Windows
* iOS
* Ubuntu
* Android

Des Weiteren hat sich in letzter Zeit der Embedded Markt als mögliche Zielgruppe herauskristallisiert. Deshalb sollen folgende Microcontroller / -platformen berücksichtigt werden:

* ARM Archithektur
* PIC
* MSP430 von TI
* ...

Wesentliches Untscheidungsmerkmal dieser Microcontroller ist, dass sie sich im Sinne der Rechenarchithektur unterscheiden. Das Hauptkritierum ist

* 8bit
* 16bit
* 32bit
* 64bit

Sowie die Unterstützung von

* Fixkommawerten
* Gleitkommawerten

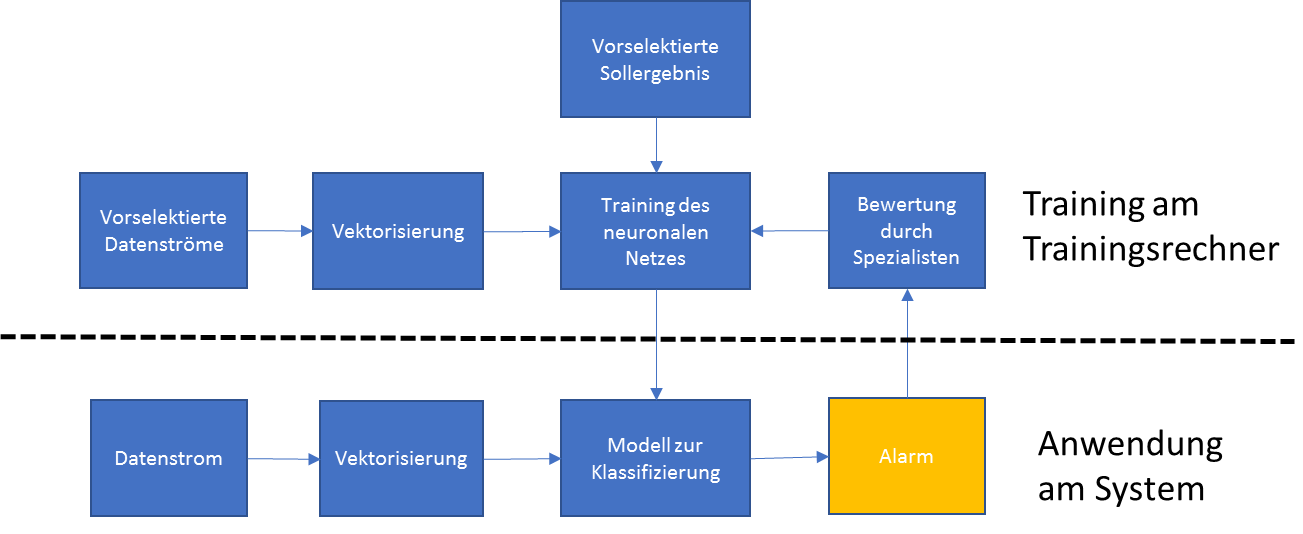
Die Entwicklungsumgebung soll entweder auf Windows oder Linux laufen.

Die Auswertung der Algorithmen kann man auf verschiedenste Art und Weise erfolgen. In der Literatur findet man verschiedenste Ansätze. Die folgende Metrik wird zum Bewerten der Algorithmen oft verwendet:

* False Positive (FP): Die Anzahl der normalen Detektionen, welche als „Anomal“ klassifiziert wurden
* False Negative (FN): die Anzahl der anomalen Detektion, welche als „Normal“ klassifiziert wurden
* Total Number (TN): Totale Anzhal der normalen Detektionen
* Total Attacks (TA): Totale Anzahl der Attacken
* Detektionsrate: DR = (TA-FN)/TA=100 [%]
* Falschalarmrate: FR = FP/TN\*100

## Entwicklungsprozess

Eine weitere Unterscheidung ist, wo das Modell trainiert wird. Dies kann entweder an Board des zu überwachenden Systems erfolgen oder auf einem unabhängigen Rechner. Das Training des Modells ist typischerweise eine sehr rechenaufwendiger Prozess, der eine Grafikkarte benötigt. Deshalb wird hier folgende Vorgangsweise vorgeschlagen:



Die Schnittstelle zwischen der eigentlichen Anwendung am System und dem Trainingscomputer ist das Modell, welches am Trainingsrechner errechnet trainiert wurde. Diese kann z. B. als C/C++ Code an das System übertragen werden und dort z.B. mittels Snort eingebunden werden. Zum Training müssen vorsortierte Daten vorhanden sein. Auf der einen Seite Daten, welche hier als Datensträme vorselektiert sind und auf der anderen Seite, Ergebnisse im Datenstrom, welche das Ergebnis wiederspiegeln. Des Weiteren sollen die vorselektierten Daten in Lerndaten und Evaluierungssdaten aufgeteilt werden. Die Lerndaten werden zum Training des Modell verwendet und die Evaluierungsdaten zur unabhängigen Bewertung des erlernten Modells. Das Modell wird danach im System eingebunden und im Echtzeiteinsatz verwendet. Kommt es zu einem Alarm, dann wird der Alarm mit der Signaltur vom System auf den Trainingsrechner gespielt und ein Spezialitst kann diesen bewerten und so dem Modell zusätzliche Trainingsdaten liefern. Daraus sollte über die Zeit ein Modell entstehen, welches mit einer hohen Detektionrate und einer kleine Falschalarmrate arbeteitet.

Zum initialen Training könnte man z. B. auf Signaturen von Snort zurückgreifen (https://www.snort.org/downloads).

## Programmpakete

Im Internet findet man viele verschiedene Programmpaket zum Entwickeln von Machine Learning Ansätzen. Jedoch kann man nur schwer Aussagen über die definierten Anforderungen im speziellen für die Mikrocontroller finden. Die folgenden Parameter sind Parameter werden hier als maßgeblich zur Auswahl des richtigen Programmpaketes angesehen.

* Programmiersprache
* Interface
* Multiprozessorunterstützung
* Open Computing Language (OpenCL)
* Numerische Ableitung
* CUDA Support
* Rückgekoppelte Netze
* Convolutional Netze
* Parallele Ausführung
* Lizenzen
* Ankaufkosten
* Langfristige Kosten

Auf die Detailausarbeitung wird hier verzichtet, da dies den Umfang sprengen würde. Lediglich die zusammenfassende Tabelle wird hier vorgestellt.

Die nächste Tabelle gibt einen Überblick über die wichtigsten Programmpakete.



Die Auswahl der richtigen Platform ist wichtig, damit eine schnelle Entwicklung erfolgen kann. Geht man davon aus, dass bei GDC hauptsächlich Personen mit IT Background im Einsatz sind, dann ist es vermutlich am Besten ein Programmpaket zu wählen, welches mit einer Schnittstelle über C++ oder Python verfügt. Aus Erfahrung kann man sagen, dass es speziell in Matlab sehr einfach ist, sich schnell in eine Materie einzuarbeiten. Nachteil von Matlab ist, dass es sehr teuer ist, wie man in der Tabelle sehen kann. Aus diesen Gründen kann man folgende Empfehlungen aussprechen:

Schneller Einstieg: MatConvNet und Matlab – sofern eine Matlab Lizenz vorhanden ist. MatConvNet hat einige gute Beispiele.

Langfristige Verwendung: Caffe2 – ist under BSD Lizenz und somit frei verfügbar, hat lt. Dokumentation einen modularen Aufbau und hat auch viele Beispiele verfügbar. Speziell auf Grund des Backgrounds der Mitarbeiter ist diese Programmpaket die richtige Auswahl.

# Deep Learning

Dieses Kapitel soll nun dazu dienen ein Verständnis für das Deep Learning zu verschaffen und den Status der Forschung darzustellen.

## Einleitung

Deep Learning – Methoden wurden auf eine große Menge von Problemen in den letzten Jahren angewandt (Arel, Rose und Karnowski). Die bekanntesten Anwendungen sind in der Bildverarbeitung (Krizhevsky, Sutskever und Hinton), Sprachverarbeitung (Weston) und der Audioanalyse (Lee, Pham und Largman). In vielen dieser Applikationen wurden algorithmenbasierte Ansätze durch Deep-Learning Ansätze in Bezug auf die Performance übertroffen. Der Kern aller Deep-Learning Algorithmen ist die Unabhängigkeit von der Aufgabenstellung, welches so zu einer effizienten Lösung führen kann.

## Traditionelle Neuronale Netze

Artificial neural networks (ANN) sind Techniken des Maschinellen Lernen, welche durch ein biologisches neuronales Netz (BNN) inspiriert sind. Obwohl diese gleich in dem Sinne sind, dass beide eine große Anzahl von verknüpften Recheneinheiten benutzen, um eine hohe Performance für eine komplexe Aufgabenstellungen zu erreichen. Moderne ANN sind stark optimiert für den Einsatz auf Computern und haben deshalb nur wenig Ähnlichkeiten mit BNN. Im Detail sind die zeitabhängigen „integrate-and-fire“-Mechanismen in BNN durch stetige Werte repräsentierende „frequency-and-fire“ Algorithmen ersetzt worden. Desweiteren haben die meisten ANN eine sehr vereinfachte Verbindungsarchithektur, welches einer effizienten Vorrausberechnung erlaubt.

Heutige ANN Archithekturen erlauben keine Schleifen in den Verbindungen (mit der Ausnahme der rückgekoppelten neuronalen Netzwerke (Recurrent Neural Network), welche eine Schleife zur Modellierung von temporalen Korrelationen verwendet (Zipser.), sowie keine Erstellung und das Aufbrauchen von Verbindung während des Trainings erlaubt (mit der erwähnenswerten Ausnahme von der genetischen Algorithmen ((Leung, Lam und Ling), (Richard K Belew)). Falls nicht anders dargestellt, sind alle folgenden Referenzen zu neuronalen Netzen (NN) in Bezug auf ANN zu sehen.

### Netzwerkarchithektur

In einem typsichen neuronalen Netzwrk wrden Knoten in Schichten platziert, wobei die erste Schicht die Eingangsschicht und die letzte Schicht die Ausgangsschicht ist. Die Eingangsknoten sind speziell im sinne, dass deren Ausgänge die korrespondierenden Features des Eingangsvektors sind. Hat z. B. eine Klassifikationsaufgabe einen 3-dimensionalen Eingang (x,y,z( und eine binären Ausgang, dann ist ein möglicher Netzwerkaufbau, dass man drei Eingangsknoten und einen Ausgangsknoten hat. Die Eingangs- und Ausgangsschicht sind normalerweise fest im Netzwerkaufauf fixiert.

Mit nur einer Eingangsschicht und einer Ausgangsschicht, mit allen Eingangsknoten verbunden mit allen Ausgangsknoten, kann das Netzwerk als Matrixmultiplikation oder lineare Transformation dargestellt werden. Diese Netzwerktypen können einfache Aufgabenstellungen, wo der Featurespace linear seperierbar ist, abbilden. Für linear-seperierbare Probleme exisiteren viel einfachere Methoden wie z. B. lineare oder logistische Regression, welche in der Regel gleiche Performance erzielen mit dem einzigen Unterschied, dass diese Trainingsmethoden sind.

Die meisten modernen Anwendungen von neuronalen Netzen verwenden ein oder mehrere versteckte (hidden) Schichten (Layer), also Schichten die zwischen der Eingangs- und der Ausgangsschicht sitzen, damit es möglich wird nicht-linearitäten im Featureraum zu modellieren. Die Anzahl der verstecken Schichten und die Anzahl der versteckten Knoten in jeder Schicht sind Hyper-Parameter, welche nicht immer einfach zu bestimmen sind. Obwohl einige Daumenregeln vorgestellt wurden, müssen diese meistens mittels Trial-and-Error gefunden werden. Das Risiko, wenn man zu kleine Netze benutzt ist, dass es nicht genug das System richtig zu modellieren, während man bei einem zu großen Netz (man spricht auch von Overfitting) Modellkomponenten wie Rauschen modellieren kann. Üblicherweise ist es besser man startet mit großen Netzen, da es sehr effiziente Methoden gibt mit welchen man Overfitting beseitigen kann, wie später noch vorgestellt wird. Jedoch bringt diese Herangehensweise ein paar Probleme mit sich, wie z. B. hohen Zeitaufwand und nicht eine starke Ineffizenz.

Es wruden Methoden zum automatischen Tuning der Hyperparameter, wie z. B. kaskadierte Netze (Schetinin), welche große Netze mittels iterativen Prozess trainieren, in dem zuerst mit einem minimalen Netzwerk gestartet wird und während bei jeder Iteration ein paar Kandidaten des Netzes, welche mehrere Knoten in verschiedenen Schichten haben, trainiert werden und dann die Besten behalten werden. Es gibt ebenfalls eine Tuning Methoden, welche auf genetischen Algorithmen basieren (Leung, Lam und Ling). Diese methoden haben aber nicht ein so große Ausbreitung gesehen, da sich die Trainingszeit stark vergrößert und der daraus entstehende Vorteil nur maginal ist.

Es ist nachgewiesen, dass ein Netz mit einer versteckten Schicht eine kontinurierliche Funktion im Featureraum bis zu einer beliebigen Genauigkeit und ein Netz mit zwei versteckten Schichten eine beliebige Funktion mit einer beliebigen Genauigkeit approximieren kann ((Kurt Hornik), (Moshe Leshno). Unter der Annahme von unendlicher großer Rechenpaower, Speicher und Trainingsdaten ist es theoretisch möglich über zwei versteckte Schichten hinauszugehen. Wie noch dargestellt wird, es es aber um einiges effizienter, dass komplexe Probleme mit mehreren (deeper) Schichten zu lösen.

### Netzwerknoten und Aktivierungsfunktionen

In einem neuronalen Netz hat jeder Knoten, ausgenommen sind die Eingangsknoten, eine oder mehrere skalare Ein- oder Ausgänge. Jede Verbindung zwischen Knoten hat eine skalare Gewichtung und einen Bias um den Aktivierungspunkt zu verschieben.

Der Ausgang von jedem Knoten ist wie in obiger Formel dargestellt berechnet, wobei alle Eingänge, die Gewichte und den Bias für den i-ten Knoten repräsentieren, welche auch als Aktivierungsfunktion bezeichnet wird.

Es sind einige Aktivierungsfuntion die oft verwendet werden. In Ausgangsknoten für Regressionsnetzen wird meistens eine lineare Aktivierungsfunktionen (y=x) benutzt. Für Ausgangsknoten in Klassifizierungsnetzen wird oft eine Softmax Funktion (exponientielle Normalisierung) verwendet, damit man eine Transformation des Ausganges in eine Wahrscheinlichkeitsfunktion zu erreichen.

Für versteckte Schichten ist entweder der Tangens-hypberbolicus () oder die Logistikfunktion ()). Beide Funktionen verfügen über folgende 3 Kondintionen

* Überall differenzierbar
* Monton
* Nicht-linear

Es wurde angenommen, dass diese Eigenschaften essientiell für Aktivierungsfunktionen sind.

Die Eigenschaft der Differenzierbarkeit ist wichtig, weil es möglich sein muss die Ableitung einer Funktion an einem beliebigen Punkt während des Trainings für die gradientenbasierte Methoden zu verwenden. Diese Eigenschaft ist noch notwendig für nicht-gradientenbasierte Methoden, wie zum Beispiel genetische Algorithmen.

Die Eigenschaft der Monotonie ist wichtig, weil im Falle, dass die Aktivierungsfunktion nicht monoton ist, ein zusätzliches lokales Minimum im Parameterraum eingeführt würde und dies das Trainings behindern würde.

Die Nicht-Linearität ist wichtitg, weil das netz sond die Fähigkeit verlieren würde nichtlieare Eigenschaften in den Trainingspattern zu modellieren. Die Nichtlinearität ist erreicht in dem man eine Sättigung und in die Funktionen einbaut. Für die Hyberlfunktion wird das für y=1 und y=-1 erreicht und für die Logistikfunktion wird das für y=0 und y=1.

In der Praxis ist die Logistikfunktion biologisch mehr plausibel, die Tangente der Hyperbelfunktion erlaubt normalerweise ein schnelleres Training, da diese um 0 linear ist, was bedeutet die Knoten beim Start des Trainings nicht in der Sättigung starten, auch wenn die Eingänge 0 oder negative sind (Xavier Glorot).

### Training von neurnalen Netzen

Trainingsalgorithmen für neuronale Netze fallen zwei Hauptkategorien, welche sind

* gradientenbasierte
* nicht-gradientenbasierte Methoden.

Hier wird auf gradientenbasierte Methoden fokussiert, da diese öfters benutzt werden und normalerweise schneller konvergieren.

Wie bereits weiter oben beschrieben hat jeder Knoten einen im angeschlossene Gewichtung für jeden Eingang und einen skalaren Bias. Die Gewichtung und der Bias eines Knoten sind somit Parameter des Knoten. Wenn man nun alle Gewichte und Biases aller Knoten in einen Vektor zusammenfasst, kann man damit das gesamte Verhalten eines Netzes beschreiben (zumindest für eine gegebene Mende von Hyperparametern).

Falls die Menge der Hyperparameter in die Funktion f() codiert ist, dann kann man den Ausgang des Netzes wie in der obigen Gleichung darstellen, wo y und x der Ausgangs- und Eingangsvektor sind.

Das Ziel des Trainingsprozesses ist es nun den Parametervektor so zu bestimmen, dass die Funktion bestmöglich approximiert, die das Modell beschreibt. In anderen Worten, hat man Eingangsparameter und gewünschte Ausgangsparameter, versucht man ein zu finden, welches die Differenz zwischen gewünschten Ausgang und Netzausgang für alle Eingangstrainigsparameter minimiert. Dazu kann man den Messfehler definieren

Ein solcher Fehler ist der mittlere Fehler der kleinsten Fehlerquadrate (mean-squared-error - MSE), welches der am weitesten verbreitete Fehler ist, wie man in der obigen Gleichung sehen kann. Ts ist die Menge der Eingangstrainingsdaten und der zugehörige Ausgang der Eingangstrainingsdaten. N ist die Anzahl der Einträge der Trainingsmenge und ist der Netzausgang.

Das Ziel ist es nun für ein gegebenes zu minimieren. Für sehr kleine Netzwerke kann es sinnvoll sein, dass man nach allen möglichen Lösungen sucht, damit man die minimierenden Parameter findet, aber für Netze ab einer gewissen Größe ist dies nicht mehr möglich. Die Gradient-Descent Methode ist übleicherweise verwendete Optimierungsmethode für neuronale Netze.

Für die Gradient-Descent Methode wird zuerst initialierst mit zufälligen Parameterraumwerten. Die Gewichte und Basises werden typischerweise so gewählt, dass die meisten Knoten am Beginn in der linearen Region sind. Eine gängige methode ist Zufallszahlen von einer gleichförmigen Verteilung () gewählt, wobei a in Abhängigkeit der Sättigungsfunktion gewählt wird (dort wo die Sättigung startet) und ist die Anzahl der ein Eingänge der Knoten (Yam und Chow).

Nach der Initialisierung optimiert die Descent-Gradienten Methode den Parameterraum basierend auf dem Gradient der Fehlerfläche. In der einfachsten Form wird in jeder Interation in die entgegengesetzte Richtung des Gradienten mit einer Schrittweite, welche proportional Größe des Gradienten ist, und einer fixen Lernrate durchgeführt. Die ist in obiger Gleichung dargestellt.

Die partielle Ableitung der Fehlerfunktion für jeden Parameter ist in obiger Gleichung gezeigt. Wendet man die Kettenregel an, wobei das Resultat für die Summation (Ausgangs zur Aktivierungsfunktion), der Ausgang der Aktivierungsfunktion und sind die Gewichte gefunden wird.

Der einzige Unterschied zwischen der Ausgangsschicht und den versteckten Schichten ist, dass die versteckten Schichten keinen wirklichen Fehler haben. Wenn man trotzdem Fehlerterme für diese berechnet, kann man deren Einfluss auf die Aktivierungsfunktion des nächsten Knoten darstellen. Dies ist äquivalent zu für die Anwendung der Kettenregel, da die Eingänge der Aktivierungsfunktion einfach die Summe der beitragenden Knoten der vorhergehenden Schicht sind. Es ist vorteilhaft diese Rückwärtsberechnung mit dem finalen Fehler als Ausgangsbasis vorzunehmen, indem man die Ableitung der Aktivierungsfunktion mit dem Fehler multipliziert, bevor man eine Strafe, proportional zu den Gewichten, auf den vorherigen Knoten. Das ist die grundsätzliche Vorgangsweise zur Fehlerrückführung (Backpropagation) und dies ist auch der Grund warum die Aktivierungsfunktion differenzierbar sein muss. Auf eine genaue Herleitung wird hier verzeichtet.

Viele Varianten des Gradient Descent wurden vorgestellt, wobei jede über unterschiedliche Performanceeigenschaften verfügt. Diejenige mit der größten Anwendung ist die „Gradient-Descent with Momentum“, wo anstelle der Berechnung des Gradienten zu jedem Zeitpunkt als Schrittweite verwendet wird, ein Teil des Gewichtes aus dem vorherigen Schritt verwendet wird. Das erlaubt schnellere Konvergenz in Situationen wo der Gradient in einigen Dimensionen viel größer ist, als in anderen (Pearlmutter).

Eine weitere gängige Variation ist das zeitliche Anpassen der Lernrate in Abhängigkeit der Lernfortschrittes. Das Ziel ist, dass man schnell zum lokalen Minimum kommt und dann verlangsamt, damit ein Überschießen (Overshoot) vermieden wird. Diese Idee wurde mittels „Resilent Back-Propagation (RPROP)“ weiterentwickelt, wo nur das Vorzeichen des Gradienten verwendet wird. In RPROP hat jedes Gewicht eine unabhängige Lernrate, welche erhöht wird, falls sich das Vorzeichen des Gradienten im Bezug auf die vorherige Iteration nicht ändert und reduziert, falls sich das Vorzeichen ändert (Braun). Dies erlaubt, dass alle Gewichte nahe an ihrer optimalen Lernrate trainieren und es eliminiert Lernratenparameter, welche bei anderen Methoden manuell angepasst werden müssen. Eine initale Lernrate muss noch manuell eingestellt werden, diese hat aber keinen signifikanten Effekt auf die Trainigszeit oder das Resultat.

### Regularisierung

Falls die Trainingsmenge von der Größer her limitiert ist, was normalerweise der Fall ist, dann ist es gefährlich das Netz ohne Beschränkungen zu trainieren, da das Netz sonst Rauschen modellieren.

Ein möglicher Weg Overfitting zu vermeiden ist Regularisierung. Die Idee dahinter ist, dass Gewichte kleinere Werte bekommen. Die üblichste Variante ist die L2 Regularisierung, wobei die L2 Norm von (dem Parametervektor) zur Fehlerfunktion hinzugefügt wird, wie in der vorherigen Funktion gezeigt wurde. Der Parameter ist ein Parameter mit dem die Stärke der Regularisierung eingestellt werden kann, dieser muss manuell eingestellt werden. Fall zu klein ist, wird im Netz ein Overfitting erreicht, falls zu groß ist, dass wird man ein Underfitting erreichen.

Andere Normen, wie z. B. L1 oder L0.5 können ebenfalls benutzt werden, jedoch mit anderen Effekten. Dies wird weiter unten noch näher dargestellt.

## Deep Learning

Wie bereits weiter oben erwähnt wurde, ist ein Netz mit zwei versteckten Schichten theoretisch bereits ein unversieller Funktionsapproximator, welcher im Stande ist eine beliebige Funktion zu approximieren, kontinuierlich oder nicht, mit einer beliebigen Genauigkeit. In diesem Licht macht es offensichtlich keinen Sinn mehere verdeckte Schichten hinzuzufügen.

Der Hauptvorteil eine tiefes Netz (deep net) zu verwenden ist Knoteneffizenz. Es ist oft möglich komplexe Funktionen mit der gleichen Genauigkeit durch die Anwendung von tieferen Netzen mit einer beträchtlich geringeren Knotenanzahl im Vergleich mit einem Netz mit nur zwei verdeckten Schichten zu approximieren. Neben der Vorteil, dass man weniger Rechenkapazitäten benötigt, benötigt ein Modell mit weniger Freiheitsgraden (Anzahl der Parameter) kleinere Datenmengen zum trainieren (Schwarz).

Beispielhaft sei angenommen, dass ein Bitmapbild, welches Dreiecke, Rechtecke, Pentagone und Hexagone enthält, existiert und es die Aufgabe die Anzahl der jeder Form zu zählen. In dem Fall eines Deep Net kann die erste Schicht die Lowlevel Aufgabe zur Berechnung der Bildgradienten und der Identifikation der Linien im Bild übernehmen und dies dann in eine notwendige Form transformieren. Höhere Schichten können dann die Klassifikation und das Zählen der einfachen Darstellungen übernehmen.

Auf der anderen Seite, falls ein flaches Netz vorhanden ist, dann müssen die Lowlevel Aufgaben oftmals ausgeführt werden, da fasst keine Kreuzverbindungen zwischen Zwischenresultaten vorhanden sind. Dies würde zu viel redundanter Arbeit führen.

Mit sehr tiefen Netzen ist es möglich Funktionen mit viele Abstraktionsschichten zu modellieren, wie z. B. das Klassifizieren von Personen in Bildern oder die Type von Objekten. Es ist nicht praktisch die Aufgaben mit flachen Netzen zu realisieren, weil der Rechenaufwand exponentiell mit der Anzahl der Schichten wächst und weil ein flaches Netz eine exponentiell größere Trainingsmenge benötigt.

### Verschwindende Gradienten

Der Fakt, dass tiefere Netze eine höhere Effizenz in bezug auf den Rechenaufwand haben, ist lange bekannt, deshalb wurden erste Versuche mit solchen Netzen bereits in den 80er durchgeführt (Fukushima). 1989 zeigte (Boureau, Ponce und LeCun) das erste mal erfolgreich ein Netz mit drei verdeckten Schichten zum die Postleitzahl zu erkennen. Es war aber nicht möglich dies zu einer höheren Anzahl von Schichten oder komplexeren Aufgaben zu erweitern. Der Grund dafür war das sehr langsame Training, welches aber nicht verstanden wurde.

1991 zeigte (Hochreiter) das Problem der verschwindenden Gradienten. Dies ist im Wesentlichen was mit dem Gradienten passiert, wenn man den Fehler im Netz rückwärts (backpropagation) rechnet. Sobald ein Knoten in die Sättigung (Gradient geht gegen 0) kommt, wird der Gradient reduziert. Multiplizert man den Gradienten mit dem Fehler, erhält man nur mehr Änderungen, welche im wesentlichen Rauschen und somit werden folgenden Knoten immer langsamer trainiert. Für viele Jahre wurde dafür keine Lösung gefunden.

#### Lösungsansatz 1 – Schichtenweises Vortraining

(Hinton) zeigte 2007 eine Lösung für das beschriebene Problem und startet so das momentane große Interesse für Deep Learning. Die ist es zuerst jede Schicht in einer unüberwachten und gierigen (greedy) Art zu trainieren, um applikationsunabhängige Features für jede Schicht zu identifizieren, bevor das eigentliche Trainings mit gekennzeichneten (labeled) Daten erfolgt.

Der Weg, wie dies implementiert wurde ist ein generatives Modell in der Form einer Boltzmann Maschine, wo jede Schicht eine Menge von binären Variablen mit assozierten Wahrscheinlichkeitsverteilungen und jede Schicht ist trainiert die vorherige Schicht vorherzusagen. Dies geschieht mit einem Algorithmus genannt Contrastive Divergence.

Eine andere Idee mit dem gleichen Hintergrund ist es den originalen Eingang zu reproduzieren (Vincent, Larochelle und Bengio).

Die Idee ist es zuerst mit der Eingangsschicht und einer versteckten Schicht zu starten und das Netz (unter Verwendung der Backpropagation Gradient Descent – Methode) den originallen Input zu rekonstruieren. Falls die verdeckte Schicht weniger Knoten als Eingangsschicht hat, dann repräsentiert der Ausgangsschicht eine komporimierte und mehr abstrakte Form des Einganges. Dieser Prozess wird für jede verdeckte Schicht wiederholt. Das finale Netz kann dann mit der Ausgangsschicht trainiert werden. Dies verhindert das Problem mit den verschiedenen Gradienten, weil wenn die finale Back-Propagation ausgeführt wird, sind die meisten führeren Schichten bereits mit einer Anwendungsunabhängigen Abstraktion trainiert.

In modernen Implementierungen wird künstliches Rauschen hinzugefügt, damit man die die Robustheit erhöht.

#### Lösungsanstz 2 – Rektifizierte lineare Aktivierungseinheiten

(A. B. Xavier Glorot) zeigte 2011 einen einfacheren Ansatz um das Problems der verschwindenen Gradienten zu lösen. Deren Ansatz ist es einfach eine Aktivierungsfunktion zu benutzen, welche den Fehler nicht reduziert wenn er zurückgerechnet (backpropagated) wird (A. B. Xavier Glorot).

Die vorgeschlagene Funktion ist eine rektifizierte lineare Aktivierungsfunktion (ReLU), y=max(0,x). Die golende Abbildung zeigt einen Vergleich zwischen verschiedenen Aktivierungsfunktionen, wobei ReLU rot ist.



In der Aktivierungsregion () ist die Ableitung 1 und in der deaktivierten Region () ist die Ableitung 0. Es wird somit der Fehler nur durchgereicht und nicht verändert.

Zusammengefasst scheint diese Vorgangsweise für 2 Gründe unüblich:

* An der Stelle 0 ist die Funktion nicht differenzierbar
* Die Funktion hat keine Begrenzung auf der positiven Seite

Dass die Funktion an der Stelle nicht differenzierbar ist hat keine Konsequenzen, da die neuronalen Netze im realen Zahlen arbeiten. Es ist sehr unwahrscheinlich, dass zu irgendeinem Zeitpunkt ist. In der Praxis definiert man an der Stelle 0 die Ableitung entweder 0 oder 1. Dies wird keinen elementaren Effekt in der realen Welt haben.

Das die Funktion keine Begrenzung auf der rechten Seite hat, hat eine höhere Problematik, weil dies zu sehr großen Aktivierungeren in späteren Schichten führen kann. Dies kann zu numerischen Problemen führen. Dies kannmit einer L1 Regularisieurng gelöst werden, welches nicht nur den Betrag des Gewichtes limitiert, sondern auch eine dünne Besetzung (sparsity) nach sich zieht.

Ein dünn besetztes Netz ist eines, wo man für einen gegebenen Eingang nur eine kleine Submenge von Knoten aktiviert wird. Es ist ein Fakt, dass eine L1 und L0.5 Regularisierung eine dünne Besetzung fördert, während die L2 Regularisierung genau da Gegenteil bewirkt. L2 Regularisierung bewirkt, dass der Ausgang von vielen kleinen Gewichten (welche kleinere L2 Normen haben) abhängt, anstelle dass es von einem großen Gewicht mit einer größeren L2 Norm abhängt, welches gleich der L1 Norm, aber kleiner als die L0.5 Norm ist, wie man in der folgenden Abbildung sehen kann.



Zusammengefasst, die L1 und L0.5 Norm fördert Knoten voneinander unabhängig zu sein und keine gemeinsame Beziehung aufzubauen. Dies kann zu einer mehr effizienten Nutzung der verfügbaren Knoten führen, welches Vorteile bei der notwendigen Rechenkapazität bei Netzen mit Aktivierungsfunktionen mit harten Sättigungen (wie z. B. die rektifizierte lineare Aktivierungen), da im Falle, dass die Ausgangsknoten 0 sind, keine Kommunikation stattfinden muss.

Ein leicht besseres Ergebnis kann mit einer rektifizierten Aktivierungsfunktion mittels L1 Regularisierung und ohne Vortraining im Vergleich mit vortrainierten Netzen erreicht werden.

Ein möglicher Grund für die Erhöhung der Performance ist die effizientere Nutzung der Knoten. Mit einem vortrainierten Netz, werden Knoten trainiert diskriminierende Muster aus ein Eingangdaten herauszufiltern, auch dann wen einige dieser Muster irrelevant sind für die eigentlich Aufgabenstellung. Ohne Vortraining würden diese Muster nicht encodiert, welches somit diese Knoten für relevantere Muster freigibt.

Es wurde ebenfalls gezeigt, dass in einer Situation das Vortraining von Vorteil ist – nämlich für semi-supervised Problemstellungen. In einem semi-supervized Problem werden große Trainingsdatenmengen , aber nur ein kleine Menge davon ist markiert. In diesem Falle, beginnend mit einem Vortraining basierend auf der gesamten Datenmenge das gesamte Netz mit der markierten Datenuntermenge kann zu besseren generalisierten Ergebnissen führen.

Für reine supervized Problemstellungen, haben die meisten Forscher das Vortraining nicht mehr in Verwendung, stattdessen haben sie ReLU + L1 Regularisierung verwendet.

Es gibt nur unzureichende Untersuchungen um zu Entscheiden ob ReLU besser als traditionelle Aktivierungsfunktionen in flacheren Netzen ist. Die meisten flachen Netze benutzen nach wie vor hyperbolische Tangenten oder Logistikfunktionen, weil diese nicht von den verschiedenen Gradientenproblem berührt sind.

### Dropout

(Hinton, Srivastava und Krizhevsky) zeigte eine Methode welche Netzperformance für den Fall von limitierten Trainingsdaten stark erhöht. Die Idee stammt von Beobachtungen, wenn die Menge der Trainingsdaten klein sind, dann wird es viele mögliche Modelle geben, welche gute Eigenschaften für die Trainingsmenge haben, aber nur wenige werden auf der Testmenge funktionieren.

Die traditionelle Lösung ist es ein Ensemble der Netze von unterschiedlichen Initialisierungen und/oder Untermengen von den Trainingsmengen zu trainieren und diese dann mit deren Ausgängen einen finalen Ausgang zu realisieren (oft durch Mittelwertbildung). Dieser Ansatz ist bekannt das er die Modelperformance erhöht, aber es ist oft wegen dem Rechenaufwand für Deep Learning nicht brauchbar, wo ein Training eines einzelnen Netzes viele Stunden oder Tage dauern kann. Viele trainierte neuronale Netze werden in Echtzeitanwendungen verwendet und ein Ensemble würde das Modell verlangsamen.

Es wurde deshalb vorgeschlagen, dass 50% der Noten, zufällig ausgewählt, während jeder Trainingsiteration deaktiviert werden, auch Drop Out genannt. Ein deaktiverter Knoten würde nicht in der Vorwärtsberechnung (da er 0 ist) teilnehmen und würde so ein Fehlersignal für die Vorwärtsberechnung blockieren. Wenn das Trainings abgeschlossen ist, werden alle Knoten aktiviert, aber alle Gewichte würden halbiert um den gleichen Ausgangsbereich zu erreichen.

Diese würden die gleichen Resultate wie Ensembles für große Anzahl von Netzen mit lediglich dem 2-fachen Rechenaufwand eines einzelnen Netzes erreichen. Es wird vermutet, dass der Grund dafür die zufällige Deaktivierung von 50% der Knoten für jede Iteration ist, was dazu führt, dass Koten unabhängig von einander lernen, anstatt gleichzeit zu lernen. Da gleichzeitige Lernen ist nich optimal, weil z. B. einige Knoten zu Knoten wandern können die Fehler erlernt haben. Beim DropOut kann dies nicht passieren, da die Knoten gezwungen werden unabhängig von einander zu lernen.

### Modellkompression und Augumentierug der Trainigsmenge

Wie bereits im vorherigen Kapitel vorgestellt können Ensembles von Netzen eine höhere Performance verglichen mit einzelnen Netzwerken, aber sind unbrauchbar für reale Anwendungen in Bezug auf den notwendigen Rechenaufwand. Das ist insbesondere für kleinen markiertne Datenmengen der Fall, wo einzelne Netzwerke overfitten.

(Cristian Bucilua) stellte für semi-supervized Problemstellungen, wo eine große Datenmenge, aber nureine kleine Untermenge markiert ist, vorhanden ist, kann es von Vorteil sein ein Ensemble mit den markierten Daten zu trainieren und das so trainierte Netz zu benutzen, dass man alle Daten damit markiert. Dann kann man alle markierten Daten zum Training eines einzelnen Netzes verwenden.

In Anwendungen wo es nicht genüngend markierte und unmarkierte Daten gibt, kann die Trainingsmenge künstlich augumentiert werden. Diese künstliche Augmentierung ist Anwendungs-spezifisch. Zum Beispiel wird in der Bildverarbeitung sollte das Netz gegen Translation und Rotation invariant (unabhängig) sein. Man kann also Bilder aus der originalen Trainingsmenge rotieren, skalieren und verschieben, sodass man diese Eigenschaften abdeckt.

### Tiefe Netze flach machen

Entlang des Pfades der Modelreduktion (Ba und Caruana) zeigte, dass mit der Hilfe von tiefen Hochperformancenetzen mit einem flachen Netz, das ein flaches Netz mit einer besseren Performance trainiert werden kann, verglichen mit einem flachen Netz, welches direkt mit den Trainingsdaten trainiert wurde.

Diese Algorithmen trainieren zuerst das tiefe Netz (oder ein Ensemble von tiefen Netzen) mit der originalen Trainingsmenge und benutzt es dann um „erweiterte Markierungen“ für alle Eingänge der Trainingsmenge. In Falle des Klassifikationsproblems , wo die Ausgangsschicht oft Softmax ist, sind die „erweiterten Markierungen“ der Eingänge der Eingang zur Softmaxschicht. Die Eingänge der Softmaxschicht are logarithmische Wahrscheinlichkeiten jeder Klasse.

Ein flaches Netz kann nun trainiert werden um die logarithmische Wahrscheinlichkeit anstelle des originalen Einklassenlabel hervorzusagen. Das macht das Training viel einfacher für flache Netze, weil logarithmische Multiklassenwahrscheinlichkeitslabels mehr Informationen als Einklassenlabels bitten. Mit der Modellierung der logarithmischen Wahrscheinlichkeit imitieren flache Netze die Generalisierung von ungesehen Daten von tiefen Netzen nach, was hauptsächlich auf den relativen Werten der logarithmischen Wahrscheinlichkeiten der Klassen, welche nicht die höchsten Werte haben, abhängt. Die Performance von diesen flachen Netzen ist viel höher als bei Netzen, welche mit Singleklassenlabels trainiert wurden.

Das Resultat ist signifikant, da es beweisst, dass die Gründe warum flache Netze viel schlechter als tiefe Netze performen nicht wesentlich durch die Reduktion der reräsentativen Stärke und der Flexibilität der tiefen Netze ist. Der eigentliche Grund ist, dass die aktuellen Trainingsdaten suboptimal für flache Netze sind, und falls man bessere Trainingsalgorithmen finden kann, dann ist es potentiell möglich signifikant bessere Performance bei flachen Netzen zu erreichen.

Wie auch immer, die Performance dieser imitierenden flachen Netze ist noch immer nicht so gut wie die derer von tiefen Netzen oder deren Esembles, die sie nachbilden. Deshalb ist die Option imitierende falche Netze zu verwenden ein Tradeoff zwischen Genauigkeit und Geschwindigkeit.

### Archithekturen von neuronalen Netzen

### Convolutional Neural Networks

…

Konkrete Anwendungen auf Cybersecurity

Ev. Anhand von einfachen beispielen.

Deep Learning Research

<https://medium.com/@jason_trost/collection-of-deep-learning-cyber-security-research-papers-e1f856f71042>

Deep learning Software:

<https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software>

<https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software/Resources>

(<http://disalw3.epfl.ch/Teaching/signals_instruments_systems/ay_2008-09/lecture/ERS_08-09_W11_lecture.pdf>).