Technology
Arts Sciences
TH Köln

Bachelorarbeit
an der Technischen Hochschule Köln
im Studiengang Informatik

Identifizierung von Anomalien in Zeitreihen mit Deep Autoencodern

Erstellt von: Julia Oxé
Matrikelnummer: 11099380

Prüfer: Prof. Dr. rer. nat. Wolfgang Konen

TH Köln
Fakultät für Informatik und Ingenieurwissenschaften
Institut für Informatik
Steinmüllerallee 1
D-51643 Gummersbach

Datum der Abgabe: 05.04.2017

Kurzfassung

Künstliche neuronale Netze stehen, vor allem durch zahlreiche Erfolge auf dem Gebiet der Mustererkennung, in den letzten Jahren immer mehr im Fokus des Forschungsbereichs der künstlichen Intelligenz.
In dieser Arbeit soll das Modell eines Deep Autoencoders implementiert werden, um herauszufinden, ob sie zur Lokalisierung von Anomalien in Zeitreihen geeignet sind.
Autoencoder erstellen eine Repräsentation der Eingabedaten, die in den tiefergehenden Schichten immer weiter vereinfacht wird. Aus der codierten Repräsentation wird dann die Eingabe rekonstruiert. Die Modellierung von Autoencodern mit vielen versteckten Schicht gilt als schwierig, da die Fehlerwerte, wenn sie über viele Schichten hinweg propagiert werden, häufig verschwindend klein oder sehr groß werden und so an Aussagekraft verlieren. [1]

.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Seit der Entwicklung von Convolutional Neural Networks im Jahr 2006 hat das Interesse an künstlichen neuronalen Netzen stark zugenommen. Mit verschiedenen Variationen neuronaler Modelle lassen sich Problemstellungen, wie Mustererkennung oder Kategorisierung, lösen, aber auch Vorhersagen auf Grundlage von historischen Daten treffen sowie Funktionen approximieren.
Der Begriff Deep Learning dient als Bezeichnung für eine Gruppe von künstlichen neuronalen Netzmodellen mit vielen tiefergehenden Schichten. Den Gegensatz hierzu bilden traditionelle Modelle mit wenigen, aber dafür breiteren Schichten.
Um die Grundlagen für die Arbeit mit künstlichen neuronalen Netzen zu schaffen, wurden bereits, im Rahmen des Praxisprojekts an der Technischen Hochschule Köln, verschiedene Modelle untersucht. [2] In einem Vergleich der beiden Deep Learning Frameworks H2O und DeepLearning4J (DL4J) wurde festgestellt, dass H2O die Kriterien für die Anwendung von Deep Learning nicht erfüllt. Es wendet kein Pretraining an und ermöglicht es deshalb nicht, das Training neuronaler Netze mit vielen Schichten effizient zu gestalten. DL4J bietet Bibliotheksklassen zur Implementierung verschiedener Deep Learning Modelle an. Eines dieser Modelle ist der Deep Autoencoder.
Autoencoder erzeugen eine codierte komprimierte Version ihrer Eingabewerte, anhand derer die Eingabe approximiert wird. Zumeist wird ein einfaches Modell mit nur einer versteckten Schicht verwendet, da es schwierig ist, den Fehlerwert über viele Schichten aussagekräftig zu halten.
Das Training von Autoencodern mit vielen versteckten Schichten wurde erst durch die Einführung einer Pretraining-Methode von Geoffrey Hinton möglich. [1]

Die vorliegende Arbeit setzt sich mit dem Modell des Deep Autoencoder auseinander, um herauszufinden, ob das Deep Autoencoder Modell von DeepLearing4J in der Lage ist, Zeitreihen so gut zu approximieren, dass eingefügte Anomalien in den Daten während der Testphase erkannt werden. Durch die Anwendung des Modells auf unbekannte Zeitreihen mit ähnlichem Verlauf, soll die Genauigkeit der Schätzung von unbekannten Zeitreihen getestet werden.

Im Grundlagenteil wird sich zunächst mit der Theorie von Autoencodern, Restricted Boltzmann Maschinen und dem Pretraining-Modell von Hinton auseinandergesetzt. Das Deep Learning Booklet von H2O enthält ein Anwendungsbeispiel zur Erkennung von anomalen EEG-Zeitreihen. [3a] Die Implementierung ohne Pretraining legt die Vermutung nahe, dass dieses Modell von H2O kein effizientes Training mit vielen Schichten ermöglicht. Auch das Anwendungsbeispiel zu Autoencodern kann mit drei versteckten Schichten keine tiefe Netzstruktur aufweisen und fällt damit nicht direkt in den Bereich Deep Learning.

DL4J's Deep Autoencoder Modell verwendet, wie Hintons Modell, Restricted Boltzmann Maschinen in den versteckten Schichten und könnte so eine Umsetzung mit vielen Schichten unterstützen.

Bevor das EEG-Anwendungsbeispiel im vierten Kapitel in DL4J umgesetzt wird, werden die wichtigsten Grundlagen des Deep Learning Frameworks für die Implementierung von Autoencoder mit eigenen Daten erläutert.

Für die Umsetzung des EEG-Beispiels müssen einige Änderungen vorgenommen werden, da DL4J's Autoencoder Pretraining verwendet und das Material von H2O nicht ausreichend Einblick über alle Parameter des Modells gibt. So soll nur die Idee des Anwendungsbeispiels aufgegriffen und durch Orientierung an den, von H2O gewählten, Parametern die Modellierung erleichtert werden.

Das EEG-Modell von H2O identifiziert ganze Zeitreihen entweder als anomal oder als normal. In einem weiteren Anwendungsbeispiel soll dies noch einen Schritt weitergeführt werden, indem versucht wird, anormale Datenpunkte in Zeitreihen zu lokalisieren. Hierfür wird der S5 Datensatz der Firma Yahoo verwendet, der Zeitreihen mit verschiedenen Eigenschaften und unterschiedlich stark ausgeprägten Anomalien enthält. [4] Die Daten sind in vier Datensets mit ähnlichen Zeitreihen aufgeteilt. Anhand des Datensets A2, das Zeitreihen mit stark ausgeprägten Anomalien und dem periodischen Verlauf einer Sinuskurve enthält, sollen die Parameter des Modells optimiert und getestet werden, ob und wie gut die Anomalien unter verschiedenen Bedingungen erkannt werden.

Abschließend werden die Ergebnisse zusammengefasst und ein Ausblick auf mögliche Folgeprojekte gegeben.

2. Theoretische Grundlagen

Im Rahmen des Praxisprojekts wurde bereits ausführlich auf die grundlegende Funktionsweise künstlicher neuronaler Netze und deren Umsetzung in Form von verschiedenen Modellen mit dem Backpropagation-Algorithmus eingegangen. [2] In diesem Kapitel werden theoretische Grundlagen zu Autoencodern und Restricted Boltzmann Maschinen erneut ausführlicher behandelt, um anhand dieser das Modell des Deep Autoencoder zu erläutern. Es folgen Abschnitte zur Beschreibung und Analyse von Zeitreihen sowie Hinweise zu weiterführender Literatur.
Im Jahr 1991 veröffentliche Sebastian Hochreiter seine Diplomarbeit, die sich mit tiefen neuronalen Netzen auseinandersetzt und festhält, dass es schwierig ist, Netze mit vielen Schichten und dem Backpropagation-Algorithmus zu trainieren. Als Grund für diesen Effekt gibt er das mittlerweile bekannte Problem des verschwindenden oder explodierenden Gradienten an. Die Fehlersignale können während der Propagierung durch das Netz nicht konstant gehalten werden und „verschwinden“ oder „explodieren“ mit zunehmender Anzahl von Schichten. [5]

2.1. Autoencoder

In diesem Abschnitt wird der traditionelle Autoencoder vorgestellt, der zusammen mit Restricted Boltzmann Maschinen die Grundlage für das Deep Autoencoder Modell von Geoffrey Hinton darstellt. [1]
Autoencoder werden dazu verwendet eine komprimierte Repräsentation des Eingabevektors zu erzeugen und gehören zu den unüberwachten Lernverfahren.
Diese Repräsentation kann als codierte Version der Eingabe betrachtet werden. Der Eingabevektor x wird über mehrere versteckte Schichten mit abnehmender Knotenzahl in eine minimierte Repräsentation y transformiert
Diese Repräsentation der innersten Schicht wird auch als Encoder bezeichnet und schließt den Encoding-Teil des Netzes ab. Die innerste Schicht des Autoencoders stellt den Flaschenhals des Netzes dar. Ab hier beginnt die Decoding-Phase in der die Eingabe rekonstruiert wird. [6]

Abbildung 1 zeigt einen einfachen Autoencoder mit vier Eingabewerten und einer einzigen versteckten Schicht. Der Eingabevektor x wird auf zwei Werte reduziert und aus der versteckten Schicht y die Rekonstruktion der Eingabe z erzeugt.

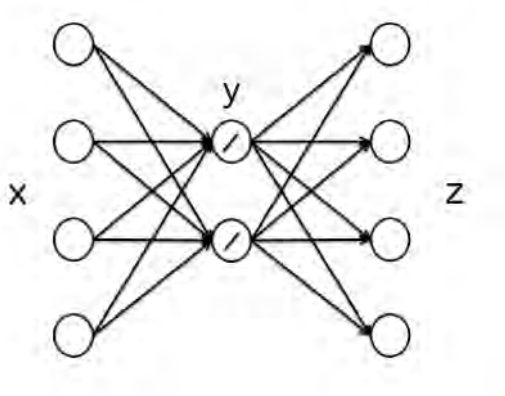


Abbildung 1 Einfacher Autoencoder mit einer versteckten Schicht y (leicht verändert [7])

In der ersten Schicht des Netzes wird lediglich die Eingabe aufbereitet, hierbei werden die Eingabewerte x_i mit den jeweiligen Kantengewichtungen w_i multipliziert und der Bias-Wert hinzuaddiert.
Die Aktivitätslevel der Knoten der versteckten Schicht y werden dann mit Hilfe einer nichtlinearen Aktivitätsfunktion φ aus der aufsummierten Eingabe berechnet und als Eingabe an die nachfolgende Schicht weitergegeben.

$$y = \varphi(W * x + b)$$

Besitzt der Autoencoder mehrere Schichten so wird das Aktivitätslevel der folgenden Schichten auf die gleiche Weise berechnet, bis die innerste Schicht erreicht wurde. Häufig werden unterschiedliche Aktivitätsfunktionen in der Encoding- und der Decoding-Phase verwendet.

Der Encoder y wird dann in der Decoding-Phase als Ausgangssituation verwendet, um die ursprüngliche Eingabe x zu rekonstruieren und auszugeben. Der Ausgabevektor z, auch als Decoder bezeichnet, hat also die gleiche Dimension wie der ursprüngliche Eingabevektor. [6]

$$z = \varphi(W' * y + b')$$

Mit Hilfe des Gradienten Abstiegsverfahrens kann der Backpropagation-Algorithmus anwendet werden, um die Kantengewichtungen nach jedem Trainingsschritt anzupassen. So wird nach jedem Schritt die Differenz zwischen Eingabe- und Ausgabevektor berechnet, um Eingabe- und Ausgabevektor zu vergleichen. Die Verlustfunktion wird dann dazu verwendet, den Fehler des Modells an dieser Stelle zu bestimmen. Da der Autoencoder eine Rekonstruktion seiner Eingabe erzeugt, wird der Fehler der Funktion in der Regel durch die quadratische Abweichung berechnet, damit positive und negative Werte sich nicht gegenseitig aufheben, wenn sie summiert werden.

$$ReconErr\ (x, z) = (x - z)^2$$

Die Anpassung der Gewichtungen wird mit Hilfe dieses Fehlerwertes und der gewählten Lernrate über die gesamte Trainingszeit angepasst und so die Rekonstruktion der Eingabe optimiert. [7]

Der Ausgabevektor stellt jedoch keine perfekte Kopie des Eingabevektors dar, sondern ist vielmehr eine Schätzung auf Grund einer großen Wahrscheinlichkeit. Dies reicht aus, um Ähnlichkeiten in den Eingabewerten zu erkennen und so beispielsweise eine Bildsuche durchzuführen.

Verfügt ein Autoencoder über mehrere versteckten Schichten, so nimmt die Knotenzahl während der Encoding-Phase ab und in der Decoding-Phase umgekehrter Reihenfolge wieder zu. Theoretisch sollte die Repräsentation der Daten also schrittweise reduziert und dann wieder detaillierter gestaltet werden. [6]

In Netzen mit tiefen Strukturen funktioniert die Anpassung der Gewichte mit dem Stochastischen Gradientenverfahren nur, wenn bereits bei der Initialisierung der Gewichte eine annähernd gute Repräsentation geschaffen wurde. Zu kleine Initialisierungs-Werte verursachen, dass der Gradient verschwindet. Bei zu groß gewählten Werten, wird meist ein schlechtes lokales Minimum gefunden. [1]

Neben diesem Verfahren gibt weitere Erweiterungen des traditionellen Autoencoder Modells, auf die im Rahmen dieser Arbeit leider nicht ausführlich eingegangen werden kann.

2.2. Restricted Boltzmann Maschinen

Restricted Boltzmann Maschinen (RBM) sind künstliche neuronale Netze, die eine Wahrscheinlichkeitsverteilung über ihren Input erzeugen.

RBMs unterscheiden sich von einfachen Boltzmann Maschinen (BM) durch die Bedingung, dass jeder Knoten einer Schicht mit allen Knoten der nachfolgenden Schicht verbunden sein muss. Im RBM-Modell darf zudem kein Neuron mit einem anderen Neuron derselben Schicht verbunden sein. Beide Modelle bestehen lediglich aus zwei Schichten, der sichtbaren Inputschicht v und der versteckten Schicht h.

Die Verbindungen sind gerichtet, sodass ein symmetrischer Informationsfluss durch das Netz entsteht. Die Aktivität aller Neuronen ist binär und jede Schicht hat einen Bias-Wert.

Abbildung 2 zeigt RBM und BM im Vergleich, die versteckte Schicht stellen in diesem Fall jeweils die oberen vier Knoten dar, die sichtbare Schicht die unteren vier.

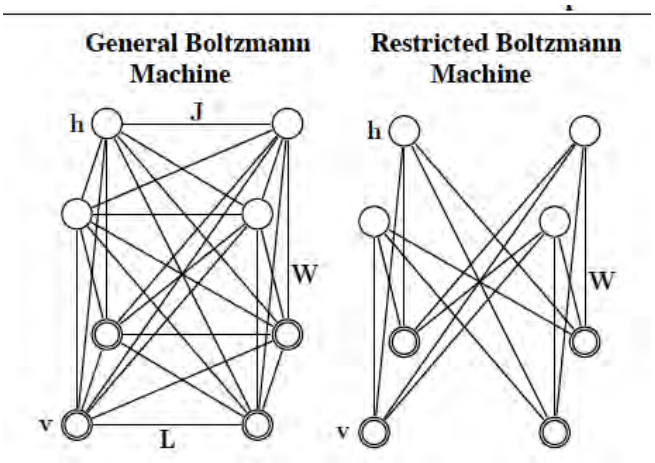


Abbildung 2 Restricted Boltzmann Maschine und Boltzmann Maschinen [9]

Die Initialisierung der Werte der RBM geschieht zufällig. Als Nächstes wird der Gibbs-Sampling-Algorithmus verwendet, um eine gleichgewichtige Verteilung zu erreichen. Dabei wird mit einem beliebigen Neuron der versteckten Schicht h begonnen, um dann abwechselnd die Knoten der aktuellen Schicht, in Abhängigkeit zu der anderen Schicht, zu aktualisieren.

Die Energie des Wertepaares, dass aus einem sichtbaren Knoten v_i und einem versteckten Knoten h_j besteht, ergibt sich aus der folgenden Funktion:

$$E(v,h) = -\sum_{i \in visible} a_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

Die Variable a_i bezeichnet den Bias-Wert der sichtbaren Einheit i und b_j den Bias-Wert des versteckten Knoten j. w_{ij} bezeichnet die Gewichtung der Kante zwischen dem Knotenpaar.
In Abhängigkeit von der Energiefunktion der einzelnen Wertepaare wird jeder Kombination eine Wahrscheinlichkeit zugewiesen.

$$p(v,h) = \frac{1}{Z}\, e^{-E(v,h)}$$

Eine hohe Wahrscheinlichkeit resultiert aus einem niedrigen Energielevel, dass sich aus Eingabe, Gewicht und Bias-Werte berechnet.
Z wird aus der Summe aller (v,h) –Wertepaare gebildet, sodass sichergestellt wird, dass sich die Wahrscheinlichkeiten zu 1 aufsummieren.

$$Z = \sum_{v,h} e^{-E(v,h)}$$

Zur Anpassung der Gewichtungen wird das Gradienten Abstiegsverfahren verwendet. Die Änderung eines Gewichts Δw_{ij} ergibt sich aus der Differenz zwischen Eingabe und Ausgabe des Modells.

$$\frac{d\, log\, p(v)}{dw_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}$$

$$\Delta w_{ij} = \varepsilon\, (\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model})$$

ε bezeichnet die Lernrate, die festlegt wie groß die Anpassung der Gewichtung in jedem Trainingsschritt sein soll.
Die Wertepaar $\langle v_i h_j \rangle_{data}$ lassen sich, da sie keine Verbindungen zwischen Knoten derselben Schicht erlauben, in Abhängigkeit des jeweils anderen Vektors ausrechnen.

Die Wahrscheinlichkeit, dass ein Knoten den Wert 1 zugewiesen bekommt, wird anhand der Aktivitätsfunktion φ, eines zufällig gewählten Testdatensatzes, bestimmt.

$$p(v_i = 1|h) = \varphi(a_i + \sum_i h_j w_{ij})$$

$$p(h_j = 1|v) = \varphi(b_j + \sum_j v_i w_{ij})$$

Das Model-Wertepaar wird dann mit dem Gibbs-Sampling-Algorithmus berechnet, indem bei einer beliebigen sichtbaren Input-Einheit begonnen wird und parallel alle versteckten Knoten mit der ersten Gleichung geändert werden. Danach werden parallel alle versteckten Knoten geändert, solange bis die Abbruchbedingung erfüllt wurde. [8]

Restricted Boltzmann Maschinen eignen sich für mehrdimensionale Reduktionsverfahren, Klassifizierungs- und Regressionsprobleme sowie kollaboratives Filtern.

2.3. Deep Autoencoder mit Pretraining

Bis etwa 2006 war die herrschende Meinung, dass es zu schwierig sei, wirklich tiefe mehrschichtige neuronale Netze effizient zu trainieren. Netzmodell, die mit nur ein oder zwei versteckten Schichten deutlich schneller waren, als Netze mit tieferen Strukturen, erzielten auch deutlich bessere oder zumindest gleichwertige Ergebnisse. Der Grund hierfür könnte sein, dass durch eine zufällige Initialisierung der Algorithmus häufig einer schlechten Lösung stecken bleibt, aus der er nicht mehr hinausfindet. [10]

In der 2006 veröffentlichten Arbeit von Hinton, Osindeto und Teh „A Greedy Algorithm for Deep Belief Networks“ wird ein Verfahren vorgestellt, dass eine Strategie darlegt, um auch tiefe Deep Belief Netze mit vielen versteckten Schichten zu trainieren. [11]

Einfache Autoencoder mit nur einer versteckten Schicht funktionieren gut bei Daten mit geringer Dimension, reichen aber nicht aus, um Daten mit hohen Dimensionen und komplexen Strukturen nachzubilden.

Hintons Algorithmus kombiniert die Autoencoder mit tiefer Struktur mit Restricted Boltzmann Maschinen, um auch in den tiefen versteckten Schichten eine sinnvolle Repräsentation der Daten zu erzeugen. [1]

Um die Initialisierung der Gewichte besser zu gestalten, wird der traditionelle Autoencoder, um eine Pretraining-Methode erweitert. Die Schichten des Netzes werden durch RBMs ersetzt, so stellt die Eingabeschicht die erste sichtbare Schicht dar. Jede versteckte Schicht h wird einzeln trainiert und fungiert für die nachfolgende Schicht als sichtbare Schicht v. Es wird eine Schicht nach der anderen trainiert, wodurch die Gewichte vortrainiert und besser Initialisiert werden.

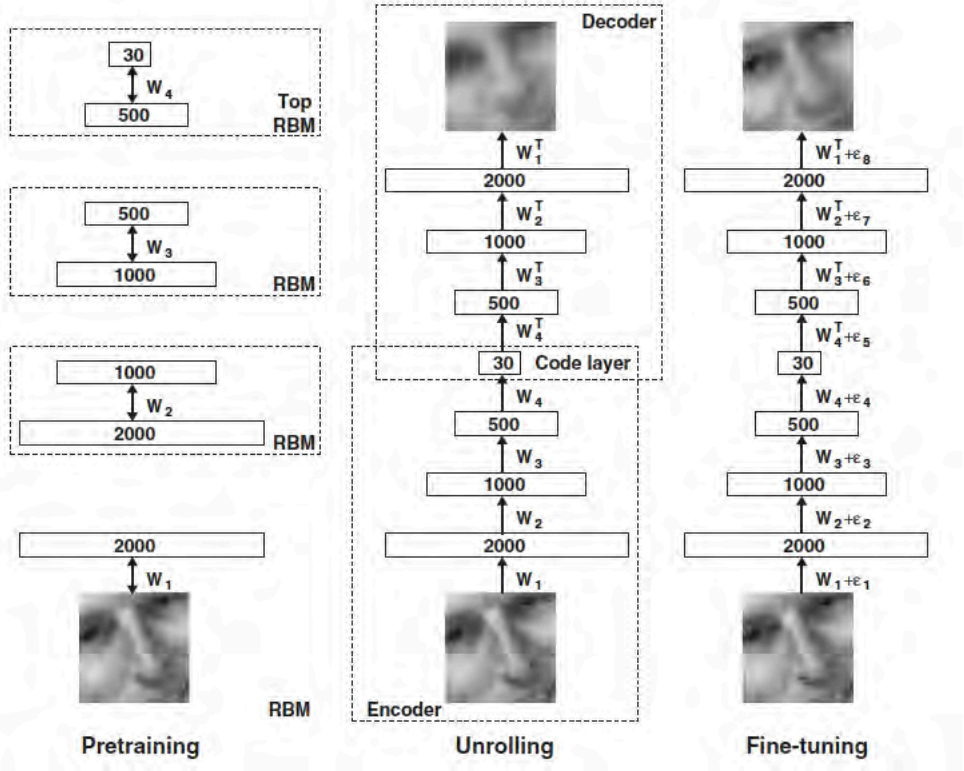


Abbildung 3 Hintons Autoencoder Modell mit Pretraining [1]

Abbildung 3 zeigt die verschiedenen Phasen des Verfahrens. Nachdem die Gewichte vortrainiert wurden, wird das Netz auseinandergefaltet, um die Encoder und Decoder-Phasen mit den im Pretraining initialisierten Gewichten durchzuführen. In der letzten Phase wird der Backpropagation-Algorithmus dazu verwendet, um die Gewichte des gesamten Autoencoders durch Anpassungen zu verfeinern und eine optimale Rekonstruktion zu erhalten.

Die Änderung der Gewichte Δw_{ij} wird anhand von Eingabedaten, Rekonstruktion und Lernrate ε berechnet.

$$\Delta w_{ij} = \varepsilon \left(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{recon} \right)$$

2.4. Umgang mit Zeitreihen in künstlichen neuronalen Netzen

Im Folgenden wird kurz auf die einzelnen Bestandteile von Zeitreihen eingegangen, um die Grundlage für Beschreibung und Analyse der Daten in den Anwendungsbeispielen zu schaffen. Grundsätzlich werden zwei verschiedene Ansätze der Datenanalyse unterschieden.
Zum einen die Querschnittsanalyse bei der die Daten an verschiedenen Untersuchungsobjekten, aber zum gleichen Zeitpunkt erhoben und dann verglichen werden. Zum anderen die Analyse zeitbezogener Daten, auch Längsschnittanalyse genannt, bei denen nur eine Komponente, jedoch zu unterschiedlichen Zeitpunkten, untersucht wird.
Längsschnittanalysen werden zur Beschreibung oder Prognose der zeitlichen Entwicklung einer Variable verwendet und auch als Zeitreihen bezeichnet.

Eine Zeitreihe besteht aus einer Menge von geordneten zeitbezogenen Werten einer Variable Y und kann wie folgt definiert werden:

$$Y = \{y_1, y_2, y_3, \dots, y_T\} \text{ mit } t = 1, 2, 3, \dots, T$$

t bezeichnet den Zeitpunkt an dem die Variable den Wert y_t annimmt. Sofern nicht anders angegeben, wird angenommen, dass die zeitlichen Abstände zwischen den Messerhebungen gleich lang sind.
Die Werte der Messungen sind entweder zeitpunktbezogen und enthalten Informationen über den Zustand des Untersuchungsobjekts zum Zeitpunkt t. (z.B. Anzahl der Studenten, die zum Zeitpunkt t in der Mensaschlange stehen.)
Oder sie bezeichnen Strömungsgrößen, die sich auf Werte innerhalb eines Zeitraums beziehen. (z.B. Anzahl der in der Mensa verkaufte Mahlzeiten im Zeitraum t bis t+1)

Der Verlauf von Zeitreihen unterliegt unterschiedlichen Einflüssen. Mit Hilfe der Zerlegung von Zeitreihen in verschiedene Komponenten, wird es ermöglicht Zeitreihen zu beschreiben und zu gruppieren.

Die Trendkomponente A beschreibt das Wachstum oder die Abnahme der Werte von Y über einen längerfristigen Zeitraum. Dieser kann einen linearen oder nichtlinearen Verlauf annehmen.
Zudem können zyklische Schwankungen auftreten, die über einen bestimmten Zeitraum periodisch wiederkehren. Unterschieden wird zwischen der saisonalen Komponente S, die einen Zyklus über einen festgelegten Zeitraum bildet (Verkauf eines Produkts über eine Saison) und der Konjunktur K, die einen Zyklus über mehrere Zeiträume beschreibt (z.B. Verkauf eines Produkts von Einführungszeitpunkt bis es vom Markt genommen wird). Beide Komponenten verlaufen aufgrund ihrer Eigenschaften nichtlinear.
Weitere Einflüsse werden durch die zufällige Komponente repräsentiert, die eine unabhängige Störgröße u darstellt. [12]

2.5. Verweis auf weiterführende Literatur

Ein Name, der im Zusammenhang mit Autoencodern und Restricted Boltzmann Maschinen immer wieder genannt wird, ist Geoffrey Hinton.
Hinton ist Professor an der Universität von Toronto und veröffentliche zahlreiche Forschungsarbeiten zu diesen Themen. Seine 1986 gemeinsam mit Rumelhart und Williams veröffentliche Arbeit zu Experimenten mit dem Backpropagation-Algorithmus, zeigten eine nützliche Repräsentation in den versteckten Schichten und trugen maßgeblich zur Verbreitung des Backpropagation-Algorithmus bei.
2006 war er beteiligt an den Arbeiten „Reducing the Dimensionality of Data with Neural Networks“ [1] und „A Fast Learning Algorithm für Deep Belief Nets“ [11], die sich mit einem Algorithmus zum Training von tiefen neuronalen Netzen zur Reduktion und Rekonstruktion von Daten beschäftigen.
Neben diesen sehr mathematischen Forschungsarbeiten findet sich weitere Literatur von Hinton, welche dem Leser beispielsweise Restricted Boltzmann Maschinen [8] und Konzepte zum Training von Netzen mit vielen Schichten [14], einfacherer erklären sollen.
Autoencoder werden häufig für Bildrekonstruktionen, zum Beispiel zur Bildsuche verwendet. So zeigen beispielsweise Riedmüller und Lange, wie eine rein visuelle Eingabe mit einem Autoencoder verschlüsselt werden kann. [15]

Eine Erweiterung des Autoencoders, stellt der Stacked Denoising Autoencoder dar. Stacked Denoising Autoencoder werden lokal trainiert, um Rauscheffekte herauszufiltern und können im Gegensatz zu gewöhnlichen Autoencodern, ähnlich dem Gabor-Filter, Umrandungen in natürlichen Bildern zu erkennen. [6]

3. Deep Learning mit DeepLearning4J

Für die Implementierung wird das Framework DeepLearning4J verwendet. Zur Einführung in Deep Learning existiert nur ein Beispiel, das jedoch keine Testphase vornimmt. Die Auswertung muss selbst programmiert und übersichtlich dargestellt werden.
Zur Verarbeitung von eigenen Datensätzen müssen diese zunächst entsprechend aufbereitet werden. Da mit Zeitreihen gearbeitet wird, müssen passende Objekte und Funktionen identifiziert und eventuell Anpassungen vorgenommen werden.

3.1. Überblick über Vorarbeit

Das Praxisprojekt hat, ergänzend zur vorliegenden Bachelorarbeit, bereits vorbereitend Grundlagen aus dem Bereich neuronale Netze behandelt. [2] Die Komplexität des Themas erfordert eine umfangreiche Einarbeitung, um ein Verständnis für die Zusammenhänge und die Funktionsweise neuronaler Modelle zu erlangen.
So wurde sich bereits mit verschiedenen Modellen, wie dem Multilayer-Perceptron, rekurrenten Netzen, Convolutional Netzen, RBMs und auch einfachen Autoencodern auseinandergesetzt. Auch eine Beschreibung der Funktionsweise des Backpropagation-Algorithmus ist in der Dokumentation des Praxisprojekts zu finden. Ziel des Projekts war es, grundlegende Kenntnisse zur praktischen Umsetzung neuronaler Netze mit den Frameworks H2O und DeepLearning4J zu erwerben und ein geeignetes Framework für diese Arbeit auszuwählen. Die beiden Frameworks wurden hinsichtlich der von ihnen bereitgestellten Modelle, Funktionen und Unterlagen verglichen. H2O ist ein Framework für maschinelle Lernverfahren. Deep Learning ist in ihrem Produktportfolio nur eines von vielen Verfahren. Mit H2O ist nur die Erstellung

von einfachen mehrschichtigen Feedforward-Netze und H2O's Version des Deep Autoencoders möglich.
Pretraining-Verfahren, durch die erst das Lernen über vielen Schichten möglich wird, werden nicht angewendet.
DL4J dagegen ermöglicht auch das Training von Netzen mit vielen Schichten, zudem stellt es weit mehr Materialen, Funktionen und Support im Bereich Deep Learning zur Verfügung und wird deshalb als Framework in dieser Arbeit verwendet. Detaillierte Angaben zu den Parametern und Funktionen zum Aufbau neuronaler Netze mit DL4J, können ebenfalls dem Praxisprojekt entnommen werden. [2]

3.2. Deep Autoencoder mit DeepLearning4J

Anders als H2O baut DeepLearning4J seine Deep Autoencoder aus zwei symmetrisch angeordneten Deep Belief Netzen auf. So stellt die erste Hälfte des Netzes den Encoding-Bereich dar und die zweite die Decoding-Phase. Sowohl der Decoding-Teil, als auch der Encoding-Teil bestehen aus vier bis fünf versteckten Schichten. Die Knoten selbst sind Restricted-Boltzmann-Maschinen. So wird wie in Hintons Modell, das Netz erst vortrainiert und zum Abschluss eine Fine-Tuning-Phase mit dem Backpropagation-Algorithmus durchgeführt, um die Anpassungen der Gewichtungen zu optimieren.

Die Erstellung eines Deep Autoencoder wird durch die Erweiterung der *NeuralNetConfiguration*-Klasse möglich, die bereits ausführlich im Praxisprojekt behandelt wurde. Auf der Website von DeepLearning4J wird eine Übersicht über die Klasse dargestellt. [16a]
Die Klasse wird verwendet, um alle vorwärts gerichteten Netzstrukturen in DL4J zu implementieren. Anhand des von DL4J zur Verfügung gestellten Quellcodes zum MNIST-Datensatz, wird nun auf den Aufbau von Deep Autoencodern mit DL4J eingegangen. [16b]

Zunächst werden einige Parameter, wie die Größe der Eingabe (Bild mit 28x28 Pixeln) festgelegt, die das Lernverhalten des Netzes bestimmen. Dann wird der *DataSetIterator* erstellt, der die Eingabedaten dem Modell zur Verfügung stellt.

```
final int numRows = 28;
final int numColumns = 28;
int seed = 123;

int numSamples = MnistDataFetcher.NUM_EXAMPLES;
int batchSize = 1000;
int iterations = 1;
int listenerFreq = iterations/5;

log.info("Load data....");
DataSetIterator iter = new MnistDataSetIterator(batchSize,numSamples,true);
```

Es folgt die Konfiguration des Modells. Die Parameter, welche keine Default-Werte erhalten sollen, müssen angegeben werden. Wichtige Parameter sind beispielsweise die Lernrate, die angibt wie groß die Anpassungen der Gewichtungen in jedem Trainingsschritt sein sollen oder die Initialisierung der Gewichte, die nach unterschiedlichen Kriterien verteilt werden kann und einen großen Einfluss auf das Lernverhalten des Netzes haben kann.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(seed)
            .iterations(iterations)
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DES
CENT)
            .list(10)
```

Im zweiten Schritt der Konfiguration werden die Schichten des Netzes initialisiert, in diesem Fall neun versteckte Schichten und eine Ausgabeschicht. Die versteckten Schichten sind RMBs und als Verlustfunktion wird *RMSE.XENT* verwendet. Dies stellt ein Problem dar, da die Funktion veraltet und nicht mehr nutzbar ist. So muss, während der Optimierung des Modells, eine Alternative gefunden werden.

```
            .layer(0, new RBM.Builder().nIn(numRows * numColumns).nOut(1000
).lossFunction(LossFunctions.LossFunction.RMSE_XENT).build())

            .layer(1, new RBM.Builder().nIn(1000).nOut(500).lossFunction(Lo
ssFunctions.LossFunction.RMSE_XENT).build())

            .layer(2, new RBM.Builder().nIn(500).nOut(250).lossFunction(Los
sFunctions.LossFunction.RMSE_XENT).build())

            .layer(3, new RBM.Builder().nIn(250).nOut(100).lossFunction(Los
sFunctions.LossFunction.RMSE_XENT).build())

            .layer(4, new RBM.Builder().nIn(100).nOut(30).lossFunction(Loss
Functions.LossFunction.RMSE_XENT).build())


            //encoding stops

            .layer(5, new RBM.Builder().nIn(30).nOut(100).lossFunction(Loss
Functions.LossFunction.RMSE_XENT).build())


            //decoding starts

            .layer(6, new RBM.Builder().nIn(100).nOut(250).lossFunction(Los
sFunctions.LossFunction.RMSE_XENT).build())

            .layer(7, new RBM.Builder().nIn(250).nOut(500).lossFunction(Los
sFunctions.LossFunction.RMSE_XENT).build())

            .layer(8, new RBM.Builder().nIn(500).nOut(1000).lossFunction(Lo
ssFunctions.LossFunction.RMSE_XENT).build())

            .layer(9, new OutputLayer.Builder(LossFunctions.LossFunction.RM
SE_XENT).nIn(1000).nOut(numRows*numColumns).build())

.pretrain(true).backprop(true)

            .build();
```

Der Deep Autoencoder verwendet das Stochastische Gradientenabstiegsverfahren für die Anwendung des Backpropagation-Algorithmus und wendet Pretraining an. Die Aktivitätsfunktion ist dem Beispiel nicht zu entnehmen.

Auffällig ist hier die kleine Knotenzahl der innersten Schicht, so wird ein Bild mit 784-Pixeln in einen Vektor mit 30 Werten codiert.
Ebenfalls auffällig ist eine Steigerung der Knotenzahl direkt nach der Eingabeschicht und vor der Ausgabeschicht. Die Eingabedaten werden also in der ersten Schicht durch mehr Knoten repräsentiert, als ihre eigene Anzahl ist. In der Testphase des Yahoo Anomalie Anwendungsbeispiels soll untersucht werden, welchen Einfluss die Anzahl der Schichten sowie die Knoten pro Schicht auf das Netz haben. In der

Decoding Schicht befindet sich die gleiche Schichtenkombination, aber in umgekehrter Reihenfolge.

Zum Abschluss wird das Modell initialisiert und jedes Datenbündel mit der *fit()*-Methode auf das Modell angepasst.

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
     model.init();


     model.setListeners(Arrays.asList((IterationListener) new ScoreIteratio
nListener(listenerFreq)));


     log.info("Train model....");
     while(iter.hasNext()) {
        DataSet next = iter.next();
        model.fit(new DataSet(next.getFeatureMatrix(),next.getFeatureMatrix
()));
```

Das Codebeispiel zeigt nur die Trainingsphase, sodass nicht gezeigt wird, wie das Modell getestet und ausgewertet werden kann. [16b]
Insgesamt sind die, von DL4J zu Autoencodern bereitgestellten Materialien leider nicht sehr umfangreich und oft auch lückenhaft, wie das gerade vorgestellte Beispiel verdeutlicht. Dennoch scheint die Implementierung des Deep Autoencoder an Hintons Modell mit Pretraining orientiert worden zu sein und könnte somit in der Lage sein Autoencoder auch über viele versteckte Schichten hinweg, effizient zu trainieren. [1]

3.3. Weitere verwendete Bibliotheksklassen

Da in den folgenden Beispielen zum ersten Mal Datensätze trainiert werden sollen, die nicht von DL4J stammen, müssen weitere Bibliotheksklassen von DL4J verwendet werden, um die Datensätze einzulesen, aufzubereiten und anschaulich darzustellen.

Die Klasse *RecordReader* ermöglicht das Einlesen von Daten aus CSV-Dateien. Der *RecordReader* splitet durch Komma getrennte oder sich in verschiedenen Zeilen befindliche Daten und gibt sie über die *next()*-Methode einzeln in der eingelesenen

Reihenfolge wieder. Da in unserem Fall Zeitreihen aus mehreren Werten zusammengesetzt werden müssen, werden die einzelnen Werte zunächst in zweidimensionale Float-Arrays einsortiert, die dann in das Format *INDArray* zur weiteren Verarbeitung mit dem *DataSetIterator,* konvertiert werden. Neben der *next()*-Methode wird zusätzlich die Methode *count()* verwendet, um die Anzahl der Zeitreihen und Werte in einer Zeitreihe zu bestimmen. [16c]

Die Klasse *INDArray* gehört zum Framework ND4J, das Klassen und Methoden für Berechnungen zur Verfügung stellt. Sie konvertiert ein- und mehrdimensionale Arrays in eine Matrizendarstellung mit der auch Matrizenberechnungen durchgeführt werden können. Die Eingabe und die Labels der Test- und Trainingsdaten müssen das *INDArray*-Format haben, um vom Modell verarbeitet zu werden. [16d]

Das Interface *DataSetIterator* wird mit diesen *INDArrays* für die Eingabe initialisiert. Sie enthält ein Objekt der Klasse *DataSet*, das mit zwei *INDArrays* für Eingabewerte und Labels initialisiert wird. Da die Labels im Fall des Autoencoders identisch mit den Eingabewerten sind, wird die Variable zweimal bei der Initialisierung verwendet.
Zusätzlich können zwei weitere *INDArrays* angeben werden, sogenannte MaskArrays, die für Angaben über Lücken in den Datenreihen genutzt werden können. Fehlt ein Datenpunkt so hat der MaskArray an dieser Stelle den Wert 0.0, ist der Wert vorhanden den Wert 1.0. In vielen Beispielen von DL4J werden die MaskArrays für Eingabe und Labels mit null initialisiert. Ihre Verwendung bei Zeitreihen mit Anomalien scheint sinnvoll, da Anomalien während des Trainings nicht miteinbezogen werden dürfen. Der Versuch MaskArrays zu verwenden, zeigte aber keinen Erfolg. Deshalb werden die Anomalien in der Implementierung durch ein eigenes *DataSet* in der Klasse *DataSetIterator* repräsentiert. [16e]

Die Dokumentation aller Klassen des Frameworks kann unter dem folgenden Link eingesehen werden: https://deeplearning4j.org/doc/
Um den Nutzern des Programms die Angabe von eigenen Parametern zu ermöglichen und die Ergebnisse des Modells anschaulich darstellen zu können, wurde die Bibliothek JFreeChart zur Implementierung einer Oberfläche für Ein- und Ausgabe verwendet. Mit JFreeChart können zahlreiche Diagrammarten erstellt werden. Die Bibliothek ist open-source und kann mit Swing-Komponenten kombiniert werden. [17]

4. Identifizierung von anormalen Zeitreihen

Im ersten Anwendungsfall wird ein Beispiel für einen mehrschichtigen Autoencoder aus dem Deep Learning Booklet von H2O in DeepLearning4J nachprogrammiert. [3a] Da die Frameworks H2O und DL4J Deep Autoencoder unterschiedlich implementieren und nicht der gesamte Quellcode eingesehen werden kann, ist es nicht möglich das Beispiel vollständig zu übernehmen. Ziel ist also nicht eine Kopie des Beispiels zu implementierten, sondern einen funktionierenden Autoencoder in DL4J zu schaffen, nach dem Vorbild und der Zielsetzung von H2O.
Der Autoencoder soll in der Lage sein ein Ergebnis zu liefern, anhand dessen die unterschiedlichen Zeitreihen eindeutig in die Kategorien „normal“ und „anormal“ eingeordnet werden können.

4.1. EEG-Anomaly-Detection mit H2O

Verwendet wurden 23 EEG-Zeitreihen, 20 mit normalem Verlauf und 3, die einen anormalen Verlauf aufweisen. Mit Hilfe der Berechnung eines Gesamtfehlerwertes soll der Autoencoder die Zeitreihen den beiden Kategorien „anomal“ und „normal“ zuordnen. In der Trainingsphase lernt das Modell ausschließlich mit den „guten“ Datensätzen, in der Testphase werden auch die „schlechten“ Zeitreihen miteinbezogen.

4.1.1. Die EEG-Zeitreihen

EEG-Zeitreihen werden in der Medizin dazu verwendet, den Verlauf von elektrischen Strömen der Gehirntätigkeit graphisch darzustellen. Anhand dieser Daten können Aussagen über den medizinischen Zustand des Patienten getroffen werden. H2O macht keine Angaben über den Ursprung der verwendeten Zeitreihen, sodass im folgenden Abschnitt zunächst der Verlauf der Zeitreihen analysiert wird.

Jede Zeitreihe besteht aus 210 Messwerten. Zusätzlich enthält jeder Datensatz einen Zeitstempel sowie eine boolesche Angabe, ob es sich bei dieser Zeitreihe um eine anomale oder normale Zeitreihe handelt. (1 = Anomalie, 0 = keine Anomalie).

Um die Auswertung besser strukturieren zu können, werden die Zeitreihen in 5 Kategorien abhängig von ihrer Ähnlichkeit einsortiert. Zudem erhalten die Zeitreihen je eine ID, nach deren Reihenfolge die Daten trainiert werden.
Der Wertebereich ist für alle Zeitreihen in etwa gleich und liegt zwischen 2 und 7. Zeitreihen einer Kategorie sind sich in ihrem Verlauf sehr ähnlich und scheinen nur leicht gestreckt oder gestaucht zu sein.

Kategorie A: Zeitreihe 0 – 4

Abbildung 4 zeigt als Beispiel Sequenz 0 aus Kategorie A. Die Funktion zeigt keine saisonalen Komponenten oder einen linearen Trend. Auch eine Störgröße, in Form von Rauschen oder ähnlichem, ist nicht zu erkennen.

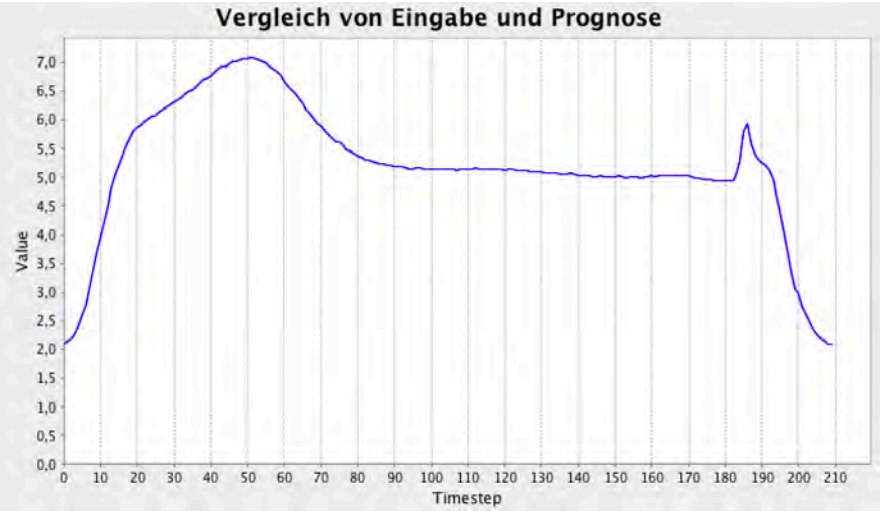


Abbildung 4 Normale Zeitreihe mit ID=0 aus Kategorie A

Kategorie B: Zeitreihe 5 – 10

Abbildung 5 zeigt als Beispiel für Kategorie B die Sequenz 8. Es scheint als wäre der Ausschnitt des gemessenen Zeitraums in der Periode 20 Zeitschritte weiter nach hinten geschoben worden als in Kategorie A. Auch hier lassen sich keine Perioden,

lineare Trends oder Rauschen identifizieren. Im Gegensatz zur ersten Funktion enthält sie ein eindeutiges Minimum.

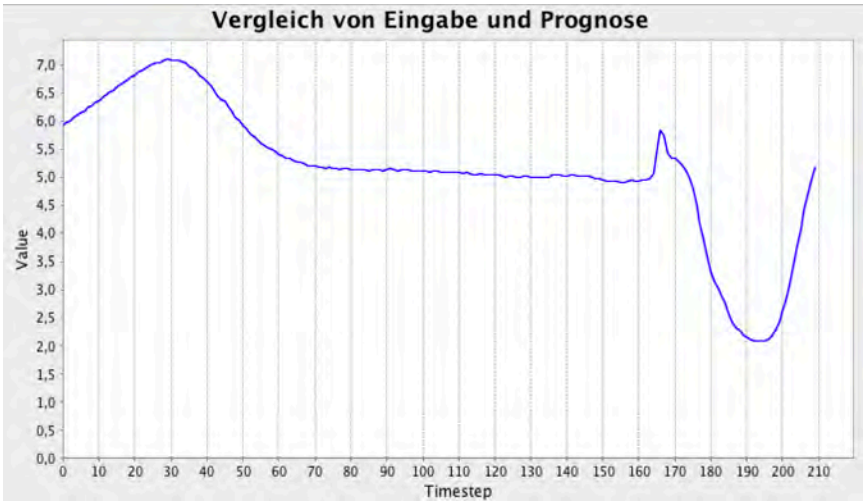


Abbildung 5 Normale Zeitreihe mit ID=8 aus Kategorie B

Kategorie C: Zeitreihe 11 – 19

Abbildung 6 zeigt Sequenz 12 als Beispiel aus Kategorie C. Die Zeitreihen dieser Kategorie ähneln stark Kategorie B, doch der Funktionsverlauf ist gestaucht, sodass etwas mehr von der Kurve zu sehen ist als in Kategorie B.

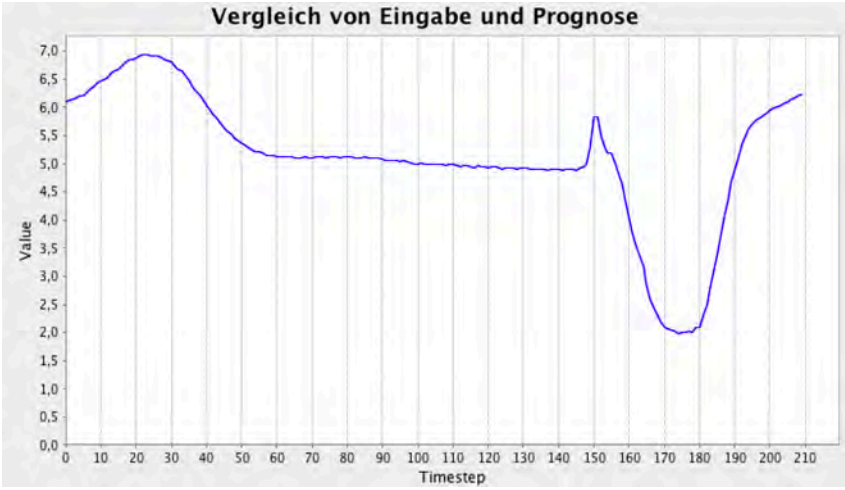


Abbildung 6 Normale Zeitreihe mit ID 12 aus Kategorie C

Kategorie D: Zeitreihe 20

Die Zeitreihe mit ID=20 unterscheidet sich in ihrem Verlauf von allen anderen Zeitreihen und bildet eine eigene Kategorie. Sie ist die einzige Zeitreihe mit einem Wertebereich von 3.5 bis 8.0 und ist in Abbildung 7 zu sehen.

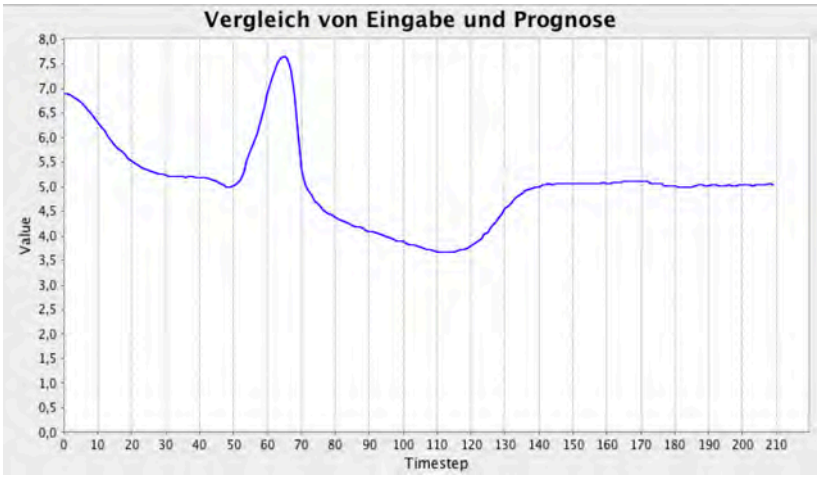


Abbildung 7 Anormale Zeitreihe mit ID = 20 aus Kategorie D

Kategorie E: Zeitreihen 21 und 22

Die anomalen Zeitreihen 21 und 22 ähneln mehr den normalen Beispielen als Zeitreihe 20. Sowohl das Maximum als auch als ein Minimum sind im Verlauf enthalten, allerdings in umgekehrter Reihenfolge als in den normalen Zeitreihen.

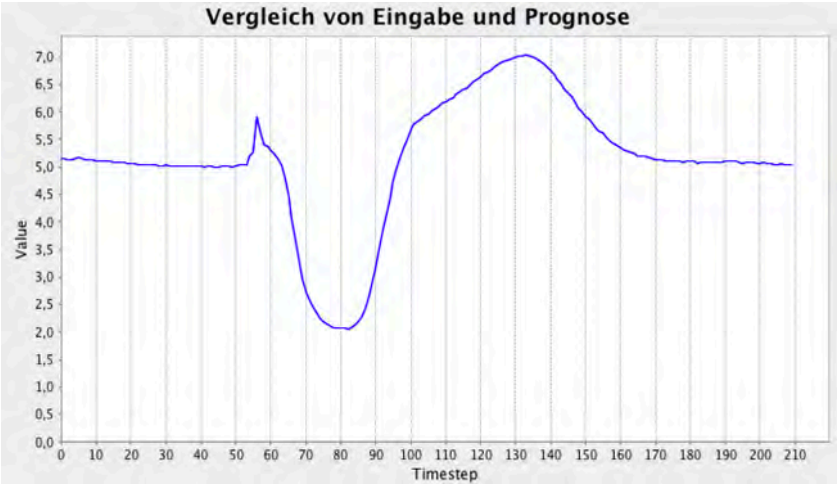


Abbildung 8 Anormale Zeitreihe mit ID = 22 aus Kategorie E

Die einzelnen Zeitreihen umfassen jeweils weniger als eine Periode, enthalten keine linearen Trends und weisen einen nichtlinearen Verlauf auf. Sie enthalten keine sichtbaren Störgrößen und werden deshalb eher als simplere Zeitreihen eingestuft.

4.1.2. Umsetzung in H2O

H2O konfiguriert das Modell in Python mit der Klasse *H2OAutoEncoderEstimator*. Auffällig ist hier, dass im Beispiel der R Version die gleiche Klasse verwendet wird, wie für den Aufbau von Feed-Forward-Netzen. In der Python-Version wurde aber eine eigene Klasse für Autoencoder deklariert, die nicht umfangreich dokumentiert ist und deren Inhalt somit nicht vollständig nachvollzogen werden kann. [3a] [3b]
Zudem bleibt die Implementierung der Methode *anomaly()* des Models, die den Rekonstruktionsfehler für jede Zeitreihe während der Testphase berechnet, unklar.
Die Umsetzung in DL4J kann sich also nur an den geänderten Parametern des Netzes orientieren. Hier fehlen jedoch wichtige Parameter, wie die Lernrate, ohne die das Training nicht funktionieren würde. Diese müssen durch Experimente bestimmt werden, bevor das Modell endgültig getestet werden kann.

Die geänderten Parameter des Beispiels, die in DL4J übernommen werden können, sind:

- Aktivitätsfunktion der versteckten Schichten: TANH
- Die Anzahl der versteckten Schichten [50, 20, 50]
- Und die Epochenzahl von 100

Das Beispiel trainiert ausschließlich mit den 20 „normalen“ Zeitreihen. Die Testphase wird dann mit allen 23 Datensätzen durchgeführt. Anschließend wird ein Fehlerwert für die Rekonstruktion jeder Zeitreihe berechnet. Anhand dieses Rekonstruktionsfehlers sollen die Zeitreihen eindeutig als „anomal“ (großer Gesamtfehler) und „normal“ (kleiner Gesamtfehler) eingeteilt werden können.
Es kann nicht direkt nachvollzogen werden, wie sich der Gesamtfehler für jeden Testdatensatz berechnet, da die Methode nicht einsehbar ist. Es wird vermutet, dass für jeden Zeitpunkt in der Trainingsphase die Varianz berechnet wird und deren Kehrwert dann in der Testphase als Gewichtung dient, um den Fehlerwerten der Testdaten unterschiedlichen Einfluss beimessen zu können. Eine Schätzung der mathematischen Berechnung findet sich im folgenden Kapitel. Eine graphische Auswertung wird bei H2O nicht implementiert, sondern nur eine Liste der Fehlerwerte für jeden Testdatensatz ausgegeben.

4.2. Umsetzung des Modells mit DeepLearning4J

Die Umsetzung befasst sich zunächst mit dem Aufbau der Anwendung und stellt die wichtigsten Funktionen und Parameter vor. Im Anschluss wird anhand des Programmablaufs dessen Funktionsweise dargelegt. Da viele Parameter nicht übernommen werden können, wird in einer kurzen Testphase versucht, das Modell hinsichtlich seiner Parameter so gut wie möglich zu optimieren. Abgeschlossen wird mit einer Analyse der Ergebnisse.

4.2.1. Ablauf des Programms

Im folgenden Abschnitt wird auf den Ablauf des Programms eingegangen, um ein Verständnis für dessen Arbeitsweise zu vermitteln.
Der Ablauf des Programms kann in fünf Schritte eingeteilt werden:

1. Datensätze importieren und DataSetIterator erzeugen.
2. Modell konfigurieren
3. Training und Berechnung der Gewichtungen für die Testphase
4. Testphase und Berechnung des Fehlers für jede Zeitreihe
5. Auswertung der Ergebnisse anhand von Diagrammen

1. Datensätze importieren und DataSetIterator erzeugen.

Die Zeitreihen sind in einer CSV-Datei gespeichert und können so mit Hilfe der Klasse *RecordReader* aus der DataVec-Bibliothek in die Anwendung geladen werden. Zunächst wird der maximale und der minimale Wert aller Zeitreihen bestimmt, um alle Daten auf einen Bereich zwischen 0 und 1 zu normieren. Die Daten müssen normiert werden, da die Ausgabe ebenfalls nur Werte zwischen 0 und 1 produziert. Nach der Normalisierung wird ein Objekt der Klasse *DataSetIterator* erstellt, welches die Trainings- und Testdaten aufbereiten.

2. Modell konfigurieren

Im nächsten Schritt wird das Modell konfiguriert. Es wurde versucht, das Modell möglichst zu übernehmen, um die Anzahl der zu optimierenden Parameter zu

reduzieren. Jedoch sind nicht alle Parameter in H2O einsehbar und können deshalb teilweise nicht übernommen werden. Wie in Kapitel 3 bereits vorgestellt, implementiert DL4J Deep Autoencoder mit RBMs und Pretraining sowie dem Backproagation-Algorithmus.
In der ersten Testphase wurden die Parameter des Modells variiert, um eine sinnvolle Zusammensetzung zu finden. Die Vorgehensweise und die Ergebnisse können dem EEG-Testprotokoll in Anhang A entnommen werden. Eine Zusammenfassung wird im folgenden Kapitel gegeben.

3. Training und Berechnung der Gewichtungen für die Testphase

In jeder Epoche der Trainingsphase wird jede Zeitreihe i des Trainingsdatensatzes genau einmal zum Training des Netzes verwendet.
Zunächst wird mit der *next()*-Methode des *DataSetIterators* der nächste Datensatz geladen und mit der *fit()*-Funktion das Modell auf den Datensatz angepasst. Da wir nur über 20 Trainingsdatensätze verfügen, werden immer alle Daten gleichzeitig ins Modell gegeben.
Anschließend werden die Aktivitätslevel der Ausgabeschicht aus dem Modell geholt, in Form eines mehrdimensionalen *INDArray*s, in dem jede Reihe die Schätzwerte für eine Zeitreihe i des Trainingsdatensatzes enthält.

In der Trainingsschleife, die über 100 Epochen läuft, wird für jeden Wert zum Zeitpunkt j einer Zeitreihe i der Fehler E_{ij}, wie folgt, berechnet:

$$E_{ij} = (in_{ij} - pred_{ij})^2$$

Anschließend wird der *INDArray* einer *ArrayList* angehängt, die ebenfalls als Matrix betrachtet werden kann, aber die Fehlerwerte des Trainings über alle Epochen enthält und im Folgenden mit E bezeichnet wird.

Nach dem Training wird die Fehlermatrix dazu verwendet, um die Gewichtungen für die einzelnen Zeitpunkte in der Testphase zu berechnen. Die Zeitpunkte, die schon während der Trainingsphase einen hohen Fehlerwert aufgewiesen haben, sollen

weniger Einfluss auf das Endergebnis haben als Daten an Zeitpunkten, die schon während des Trainings gut geschätzt wurden.
Da davon ausgegangen wird, dass sich die Trainingsergebnisse immer weiter verbessern, ist es sinnvoll vor allem die Trainingsergebnisse der letzten Epochen in die Berechnung des Fehlers mit einzubeziehen.

Zunächst wird der Fehler für jeden Zeitpunkt j der letzten trainierten Perioden aufsummiert und durch die Anzahl L, der in die Berechnung mit einbezogenen Daten geteilt.

$$ErrTrain_j = \frac{1}{L} \sum_{i \in train} E_{ij}$$

Aus dem Kehrwert des Fehlers für jeden Zeitpunkt ergibt sich dann die Gewichtung w_j für jeden Zeitpunkt j in der Testphase.

$$w_j = \frac{1}{ErrTrain_j}$$

So wird erreicht, dass bei einer kleinen Abweichung eine große Gewichtung erzeugt wird und umgekehrt.

4. Testphase und Berechnung des Fehlers für jede Zeitreihe

In der Testphase werden nun ebenfalls die anormalen Zeitreihen miteinbezogen. Die Testdaten werden ohne Anpassung der Gewichtungen des Netzes mit der Methode *activateSelectedLayers(int from, int to, INDArray input)* geschätzt und anschließend die Fehlermatrix für die Testdaten E_{kj} mit der *reconErr()*-Methode berechnet.
Der Gesamtfehler $GesErr$ für jede Zeitreihe k ergibt aus der Summe der gewichteten Fehlerwerte für jeden Zeitpunkt j.

$$GesErr_k = \sum_j E_{kj} * w_j$$

Der Gesamtfehler dient als Maß zum Vergleich der Testläufe.

5. **Auswertung der Ergebnisse anhand von Diagrammen**

Nach der Berechnung des Gesamtfehlers wird der Gesamtfehler für jede Zeitreihe an die Klasse *Plot_GUI* weitergereicht. Diese stellt den Gesamtfehler für alle 23 Zeitreihen als Balkendiagramm dar.
Das zweite Diagramm, das erzeugt wird, enthält die Eingabe und die Schätzung der Zeitreihe, die zur Analyse ausgewählt wurde sowie die Gewichtungen für die Testphase.
Für das Testprotokoll wurden Kennzahlen zum Vergleich der Diagramme berechnet, um das Testprotokoll übersichtlich und aussagekräftig zu halten.

4.2.2. Optimierung und Training des Modells

In diesem Beispiel geht es zunächst nur darum, einen funktionierenden Deep Autoencoder aufzubauen, mit dem im zweiten Anwendungsbeispiel weitergearbeitet werden kann. So wurde in der Testphase versucht, eine funktionierende Zusammensetzung der Parameter für das Modell zu finden.

In der Testphase werden zunächst einzelne Parameter variiert, um ihre Auswirkungen auf das Modell zu analysieren und eine geeignete Kombination zu finden. DL4J bietet zahlreiche Möglichkeiten, welche unmöglich in sämtlichen Kombinationen getestet werden können. Deshalb wird sich in diesem Projekt darauf beschränkt, die verschiedenen Parameter nacheinander zu variieren und jeweils die Variante mit den besten Ergebnissen als Parameter zu verwenden.

Die folgende Tabelle zeigt die vorgegebenen Parameter von H2O sowie die Parameterkombination der besten Ergebnisse für beide Ansätze aus der Testphase. Für eine genauere Einsicht in die Testfälle, wird auf Anhang A: EEG Testprotokoll verwiesen. Die Aktivitätsfunktion der Ausgabeschicht wird auf die Identität festlegt, da der Autoencoder seine Eingabe rekonstruieren soll.

Parameter	***H2O***	***DL4J mit RBM-Schichten***
Initialisierung der Gewichte	Keine Angabe	XAVIER_LEGACY
Update-Methode	Keine Angabe	ADGRAD
Optimierungs-algorithmus	Keine Angabe	STOCHASTIC GRADIENT DESCENT
Lernrate	Keine Angabe	0.3
Aufbau der Schichten	[210, 50, 20, 50, 210]	[210, 50, 20, 50, 210]
Art der versteckten Schichten	Keine Angaben	RBM
Aktivitätsfunktion der versteckten Schichten	TANH	TANH
Verlustfunktion der versteckten Schichten	-	SQUARE_LOSS
Aktivitätsfunktion der Ausgabeschicht	Keine Angabe	IDENTITY
Verlustfunktion der Ausgabeschicht	-	MSE

4.2.3. Analyse der Ergebnisse

Abbildung 9 zeigt die Gesamtfehlerwerte $GesErr_i$ für jede Zeitreihe i. Die genauen Werte sind der nachfolgenden Tabelle zu entnehmen.
Die „anormalen“ Zeitreihen 20 bis 22 haben sehr viel höhere Fehlerwerte und können somit leicht von dem Programm durch Einführung einer separierenden Fehlergrenze identifiziert werden.

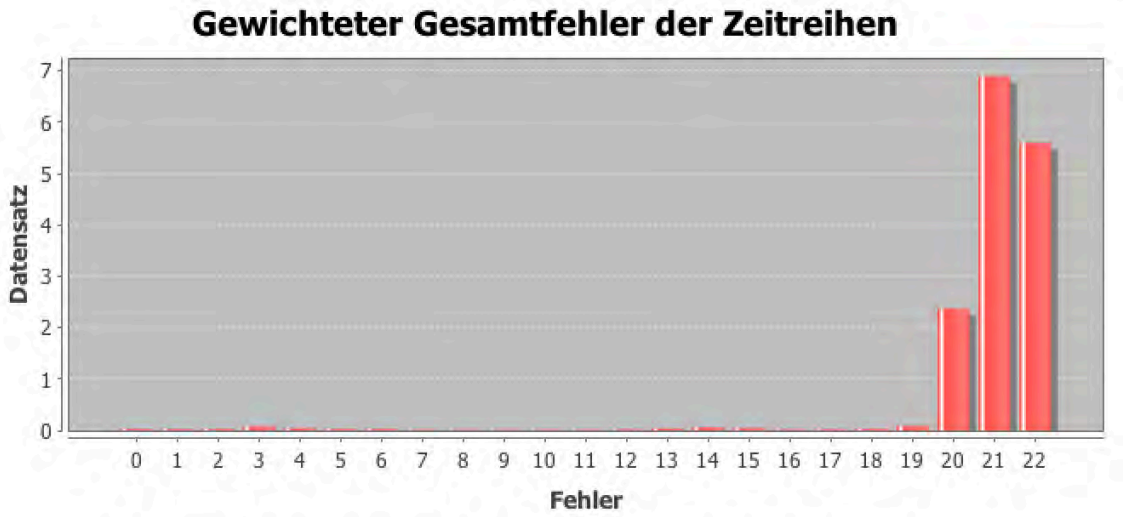


Abbildung 9 Gesamtfehler der Zeitreihen in der Testphase

$i = ID$	$GesErr_i$	$i = ID$	$GesErr_i$	$i = ID$	$GesErr_i$	$i = ID$	$GesErr_i$
0	0.0319	**6**	0.0298	**12**	0.0264	**18**	0.0339
1	0.0297	**7**	0.0149	**13**	0.0408	**19**	0.0940
2	0.0313	**8**	0.0182	**14**	0.0676	**20**	2.3735
3	0.0877	**9**	0.0169	**15**	0.0491	**21**	6.9034
4	0.0463	**10**	0.0155	**16**	0.0201	**22**	5.5995
5	0.0280	**11**	0.0184	**17**	0.0223		

Schaut man sich jedoch die Schätzung für die einzelnen Zeitreihen an, so fällt auf, dass sie alle einen ähnlichen Schätzverlauf haben, wie für jede Kategorie beispielhaft in Abbildung 10 bis 14 gezeigt wird. Die grüne Funktion stellt jeweils die Schätzung des Modells dar. Die blaue Kurve repräsentiert die Eingabe, die das Netz erhält. Die graue Linie, die in allen Abbildungen gleich ist, zeigt die Gewichtungen an den jeweiligen Zeitpunkten. So werden Daten, die sich in der Mitte der Zeitreihe befinden sehr viel stärker miteinbezogen, als Daten am Anfang und am Ende der Zeitreihe.

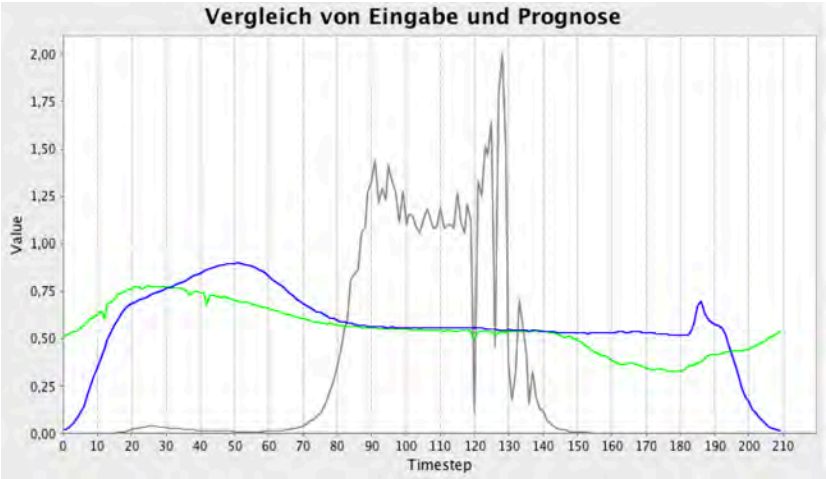


Abbildung 10 Zeitreihe mit ID = 0 aus Kategorie A mit einem Gesamtfehler von 0.0319

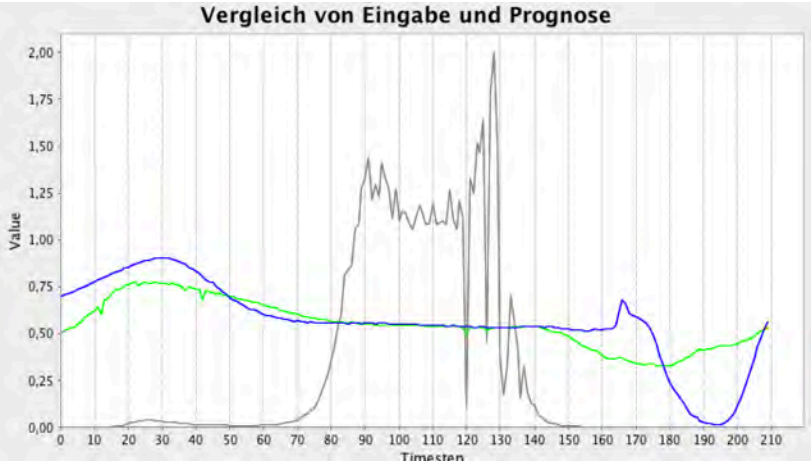


Abbildung 11 Zeitreihe mit ID = 8 aus Kategorie B mit einem Gesamtfehler von 0.0182

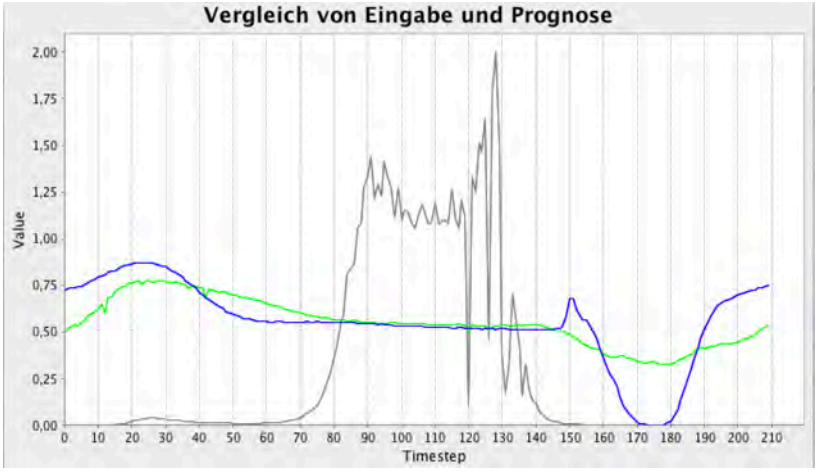


Abbildung 12 Zeitreihe mit ID = 12 aus Kategorie C mit einem Gesamtfehler von 0.0264

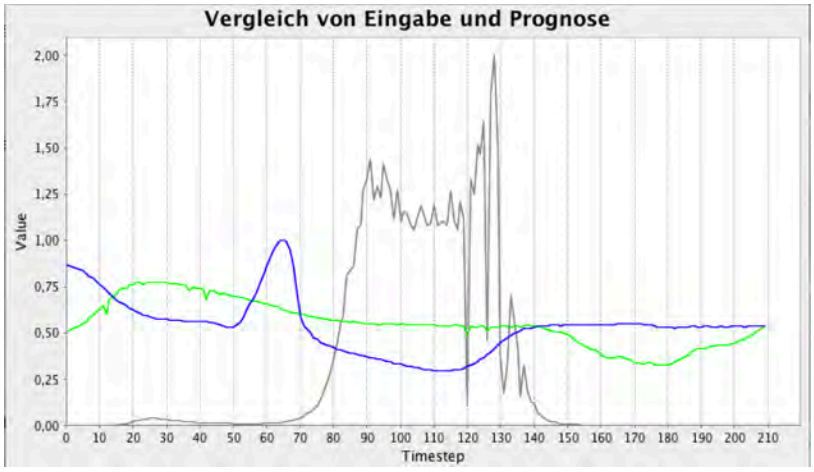


Abbildung 13 Zeitreihe mit ID = 20 aus Kategorie D mit einem Gesamtfehler von 2.3735

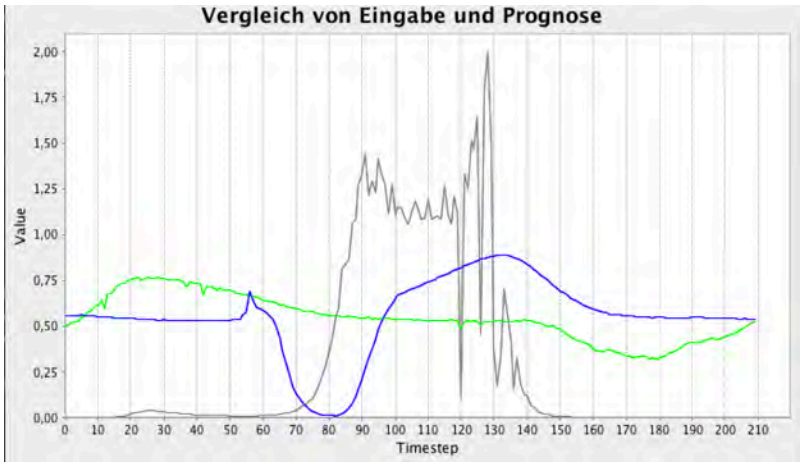


Abbildung 14 Zeitreihe mit ID = 22 aus Kategorie E mit einem Gesamtfehler von 5.5995

Es fällt auf, dass die Schätzungen des Netzes, egal um welche Zeitreihe es sich handelt, alle sehr ähnliche, wenn auch keine identischen, Werte aufweisen. Zudem erinnert der Verlauf der Schätzung am ehesten den Zeitreihen aus Kategorie C, die den größten Teil der Daten ausmachen.

Das Netz lernt einen Durchschnittswert über die Daten zu schätzen. Es ist aber nicht in der Lage, die Eingabe zu rekonstruieren. Die Anomalien werden nur erkannt, weil die guten Zeitreihen in der Mitte alle einen ähnlichen Verlauf haben und so die anormalen Zeitreihen einen hohen Gesamtfehler aufweisen. Das Netz kann mit drei versteckten Schichten, aber auch nicht als wirklich tief bezeichnet werden.

Im folgenden Anwendungsfall werden mehr und längere Zeitreihen verwendet, die auch Perioden enthalten, um zu sehen, ob sich Deep Autoencoder mit diesen Daten umsetzen lassen.

5. Lokalisierung von Anomalien in Zeitreihen

Im folgenden Anwendungsbeispiel soll nun das Modell des Deep Autoencoders mit periodischen Zeitreihen getestet und optimiert werden. Es soll herausgefunden werden, ob der Autoencoder Rekonstruktionen der Eingabe erzeugen kann und dies auch mit mehrschichtigen Varianten des Modells, beibehalten kann.
Die gewichtete Wertung der einzelnen Zeitreihen während der Testphase, die im EEG Beispiel verwendet wurde, hat keine gute Rekonstruktion hervorgebracht, sondern das Ergebnis eher manipuliert. In diesem Anwendungsfall sollen die Datensätze stattdessen vor Eingabe in das Netz mit einem gleitenden Fenster bearbeitet werden, sodass die Größe der Zeitreihen verkleinert wird und mehr Datensätze für die Trainings- und Testphase verwendet werden können. Wie im vorherigen Kapitel werden erst die Datensätze analysiert, bevor der Ablauf der Anwendung vorgestellt und das Modell getestet wird.

5.1. Yahoo Webscope S5 Dataset

Der Webscope-S5 Datensatz der Firma Yahoo besteht aus Sammlungen von Zeitreihen mit gekennzeichneten Anomalien. Enthalten sind sowohl synthetische als auch reale Zeitreihen, die in vier Datensätzen mit ähnlichen Merkmalen zusammengefasst wurden, wodurch die Lokalisierung von Anomalien unter unterschiedlichen Bedingungen möglich wird. [4] Da es sinnvoll ist, das Modell zunächst mit möglichst einfachen Zeitreihen zu testen, soll zunächst nur ein Datensatz zur Optimierung des Modells ausgewählt werden.

Jeder Datensatz besteht aus einem 3-Tupel *{timestamp, value, is_anomaly}*. Der Wert selbst ist eine Fließkommazahl und kann sowohl positiv als auch negativ sein. Zudem ist ein ganzzahliger Zeitstempel, in Form eines Integer-Wertes enthalten, sowie eine boolesche Kennzeichnung, ob es sich bei diesem Messwert um eine Anomalie handelt (1) oder nicht (0).
Datensatz A3 und A4 enthalten zudem Angaben zu Perioden, Rauschen, Trends und Wendepunkten, an denen sich die Komponenten des Verlaufs der Zeitreihe ändern.

Datensatz A1

Datensatz A1 enthält Daten, die in der realen Welt erhoben wurden (Serverdaten etc.) und manuell mit Anomalien versehen wurden.

Abbildung 15 zeigt beispielshaft Zeitreihen aus dem A1 Datensatz nach der Normalisierung. Die Zeitreihen weisen zwar Schwankungen auf, bei denen sich jedoch nicht immer eine eindeutige Regelmäßigkeit erkennen lässt.

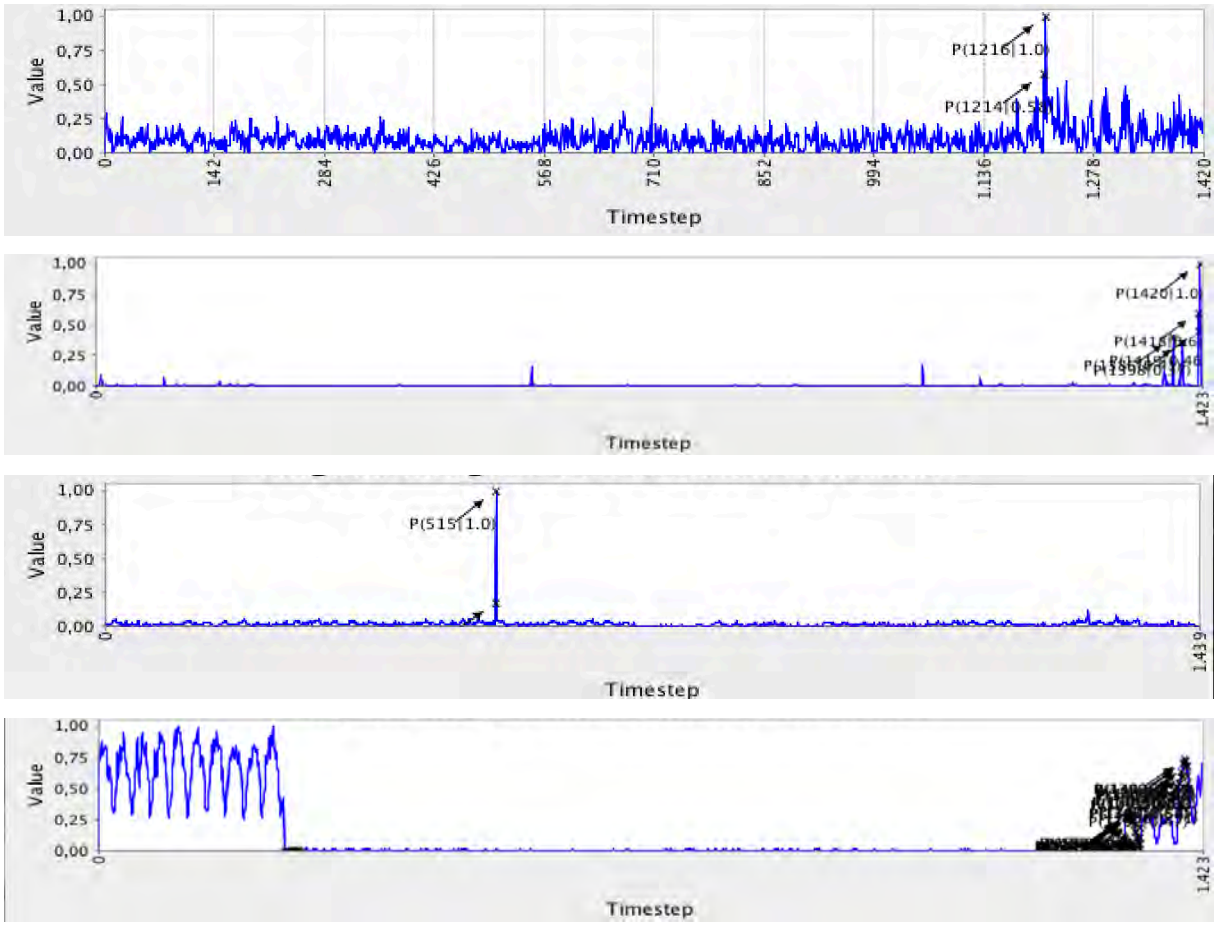


Abbildung 15 Zeitreihen 0, 3, 4, und 6 (von oben nach unten) aus Datensatz A1

Die Anomalien sind teilweise stark ausgeprägt und teilweise visuell kaum von den anderen Datenpunkten zu unterscheiden. Ein linearer Trend lässt sich nicht erkennen und die Anomalien ziehen sich, anders als bei den anderen Datensätzen, häufig über größere Zeiträume hin. Problematisch ist dies, da die Anomalien sich häufig am Ende der Zeitreihe befinden und in manchen Fällen nicht gekennzeichnet sind.

Für die Normalisierung des Trainingsdatensatzes werden die Anomalien nicht miteinbezogen, wodurch nicht gekennzeichnete Anomalien den Normalisierungsbereich beeinflussen können.
Die Zeitreihen weisen teilweise abrupte Wechsel in ihren Verlauf auf. Zudem sind sie unterschiedlich lang, was zu dem Problem führt, dass die Zeitreihen in der Testphase bei Verwendung des Datensatzes A1 nicht immer vollständig in die Auswertung miteinbezogen werden können.

Datensatz A2

Die Zeitreihen dieses Datensatzes bestehen aus synthetisch erzeugten Daten. Sie weisen eine saisonale Komponente, in Form einer Sinus-Funktion, auf. Die eingefügten Anomalien sind stark ausgeprägt und auf einen Bereich von ein bis zwei Zeitschritten begrenzt.

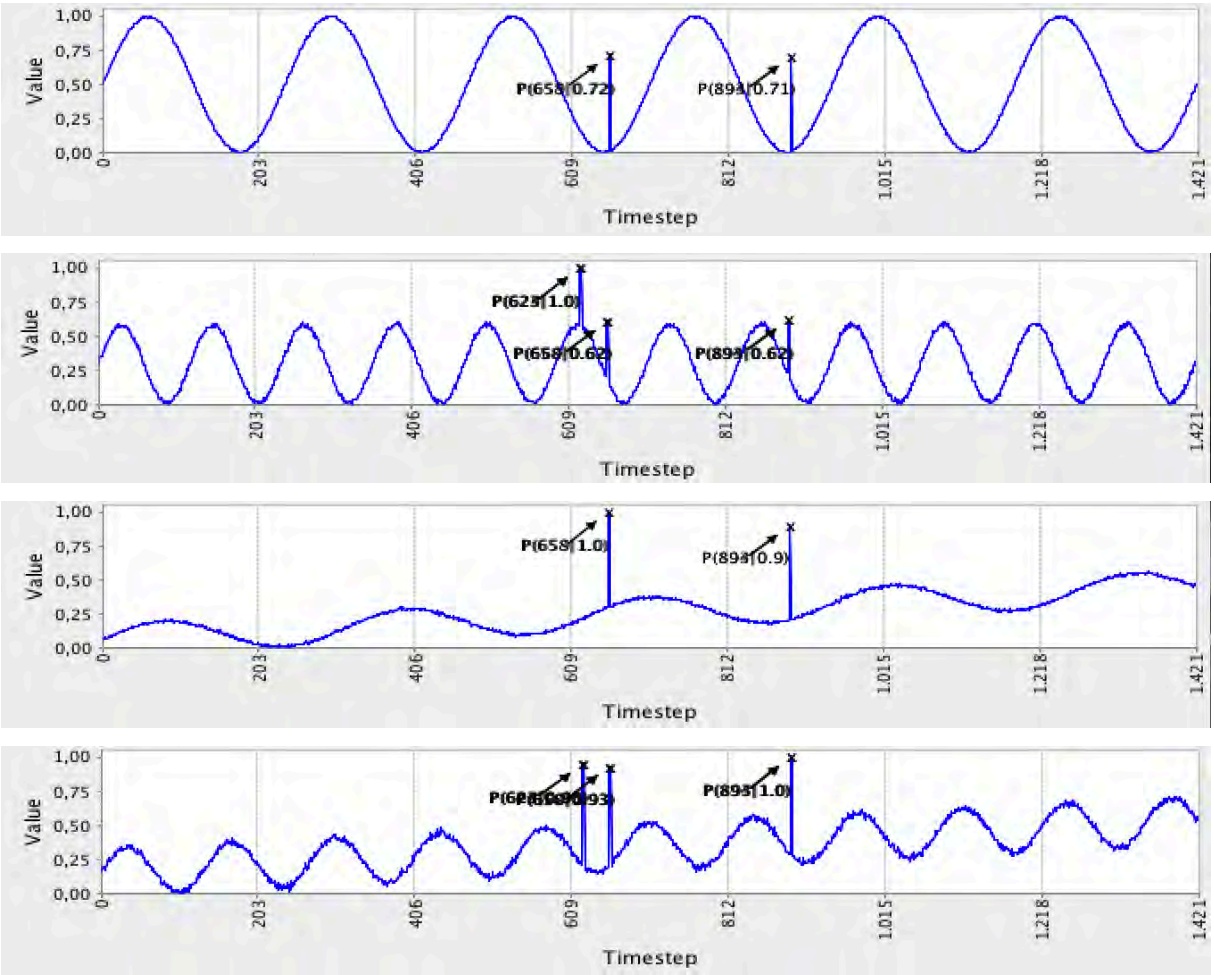


Abbildung 16 Zeitreihen 0, 4, 6 und 10 (von oben nach unten) aus Datensatz A2

Der Beobachtungszeitraum T = 1421 ist für alle 100 Zeitreihen gleich lang. Die Zeitreihen weisen teilweise einen positiven linearen Trend auf.
Zudem haben sie unterschiedliche Amplituden in x- und y-Richtung, wodurch die Länge der Perioden der Zeitreihe variiert und somit auch die Anzahl der Perioden pro Zeitreihe. Bei einigen Zeitreihen wurde ein schwaches Rauschen hinterlegt.
Abbildung 16 zeigt repräsentative Zeitreihen aus dem A2 Datensatz nach der Normalisierung zwischen 0 und 1. Die Zeitreihen werden als deutlich einfacher eingestuft, als die Daten aus A1.

Datensatz A3

Datensatz A3 enthält synthetisch erzeugte Daten. Der Beobachtungszeitraum ist für jede Zeitreihe mit T = 1680 identisch und der Datensatz enthält ebenfalls 100 Beispielsequenzen. Es wurden drei saisonale Komponenten mit unterschiedlicher Periodenlänge eingefügt. Diese sind als Fließkommazahlen im Datensatz enthalten. Auch das Rauschen und der Trend in den Daten sind angegeben. Zudem werden über eine boolesche Angabe, sogenannte „Change Points“ gekennzeichnet, die in diesem Datensatz aber alle den Wert 0 aufweisen und erst für Datensatz A4 relevant werden.
Abbildung 17 zeigt Zeitreihen aus dem A3 Datensatz nach der Normalisierung zwischen 0 und 1. Die Zeitreihen ähneln Datensatz A2, enthalten aber weniger stark ausgeprägte Anomalien, stärkeres Rauschen und auch Beispiele mit negativen linearen Trends. Sie werden deshalb schwieriger als A2 und deutlich leichter als A1 eingestuft.

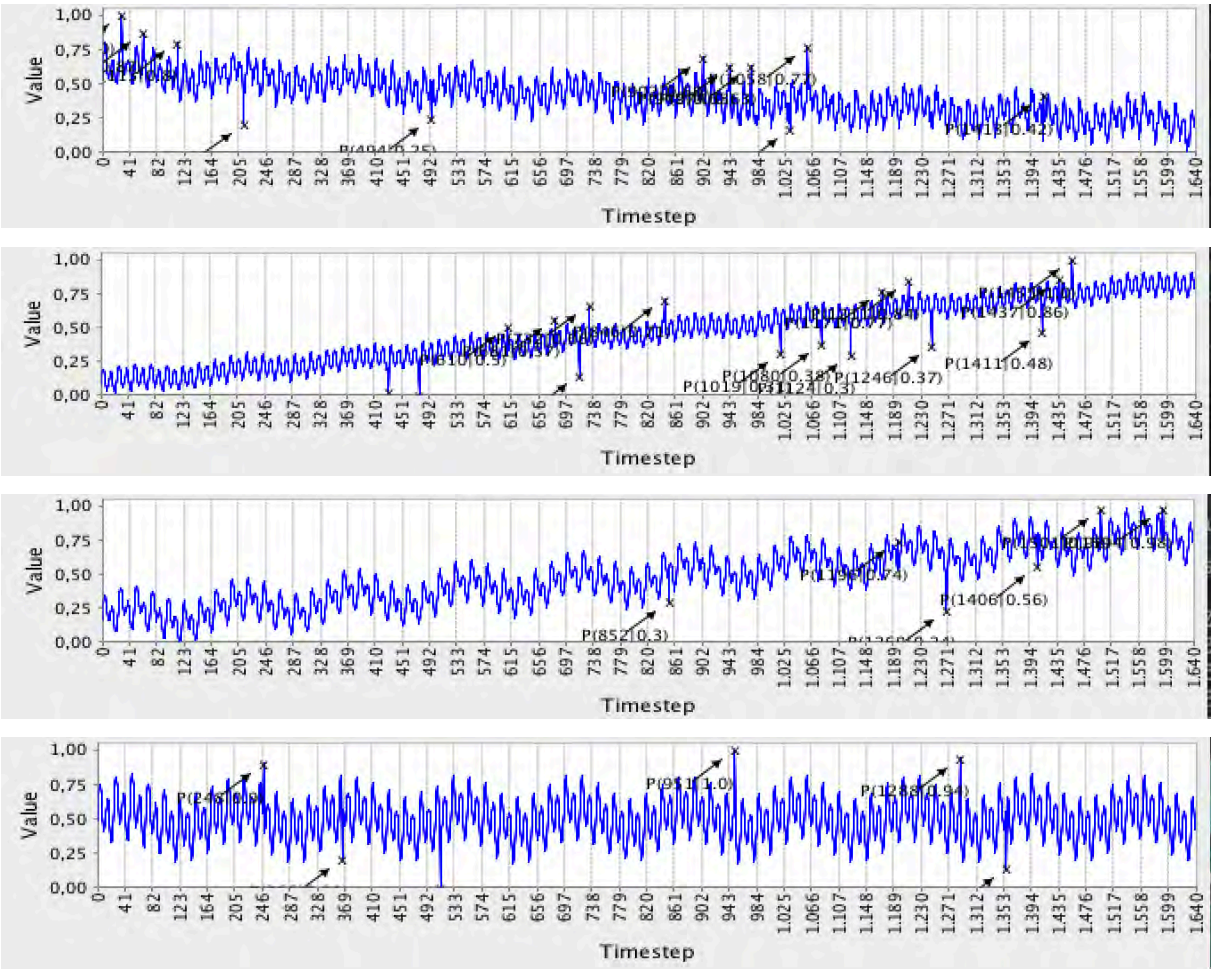


Abbildung 17 Zeitreihen 0, 1, 3 und 5 (von oben nach unten) aus Datensatz A3

Datensatz A4

Datensatz A4 enthält ebenfalls synthetisch erzeugte Daten. Der Beobachtungszeitraum ist für jede Zeitreihe mit T = 1680 gleich lang und besteht wie A2 und A3 aus 100 Zeitreihen. Trend, Rauschen und Angaben zu saisonalen Komponenten sind wie in A3 im Datensatz enthalten. Im Gegensatz zu A3 sind die „Change-Point“-Kennzeichnungen nicht alle 0 und geben einen abrupten Wechsel der Eigenschaften der Komponenten der Zeitreihe an. Datensatz A4 wird als schwieriger als A3 eingestuft, aber als einfacher als A1.

Die vorletzte Zeitreihe in Abbildung 18 zeigt ein Beispiel für eine nicht-gekennzeichnet Anomalie im vierten Abschnitt der Zeitreihe. Solche Fehlkennzeichnungen sind leider in allen Datensätzen enthalten.

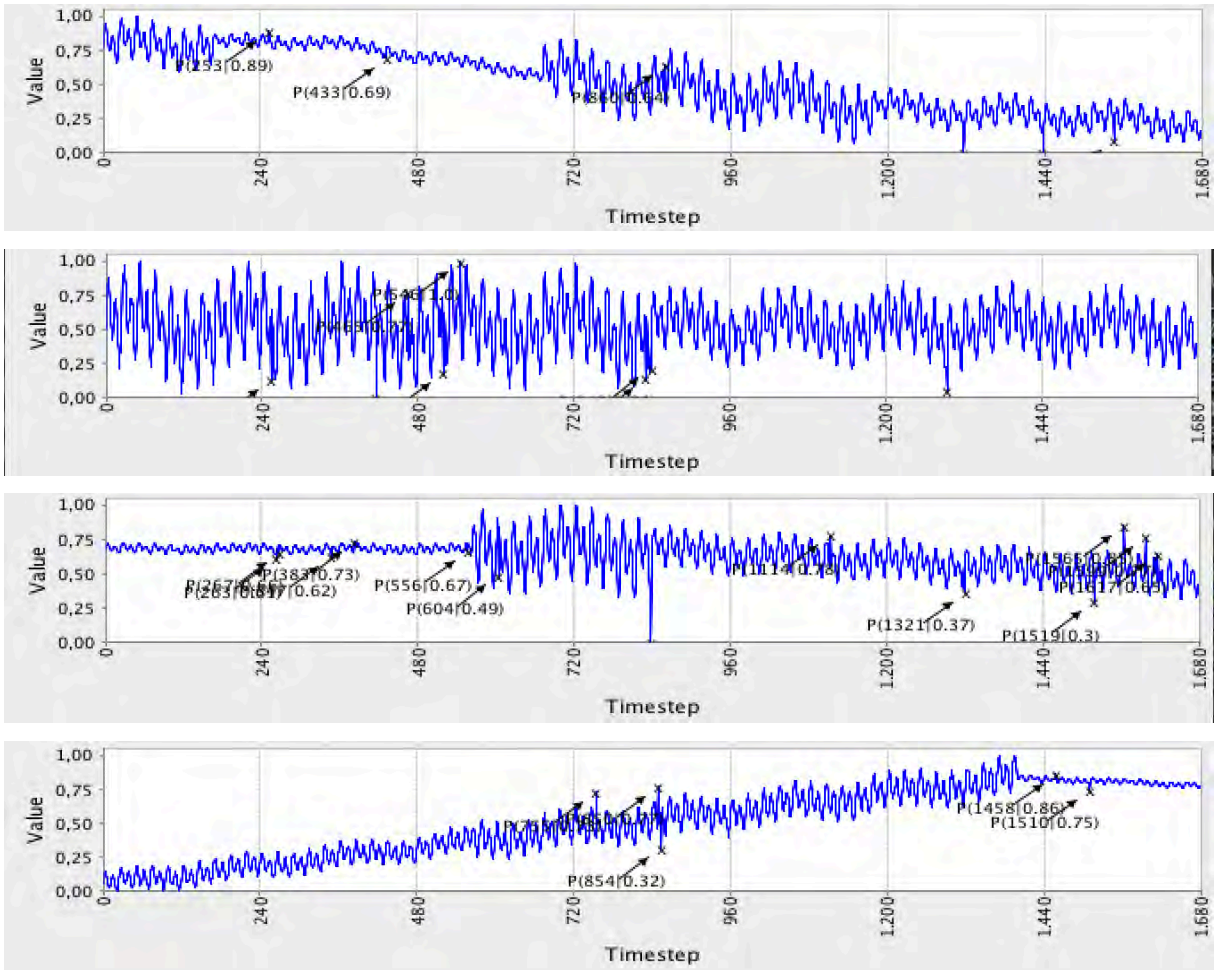


Abbildung 18 Zeitreihen 1, 4, 6 und 9 (von oben nach unten) aus Datensatz A4

Die Datensätze weisen einen unterschiedlichen Schwierigkeitsgrad auf und testen das Modell auf verschiedene Anwendungsfälle.

Datensatz A2 weist die deutlichsten Anomalien auf und wird deshalb in der ersten Testphase verwendet, in der das Modell zunächst optimiert werden soll.

Die Anomalien in Datensatz A3 sind schwieriger zu identifizieren, da sie nicht so deutlich zwischen den anderen Punkten herausstechen. Die Funktion ist unebener, da drei saisonale Schwankungen eingefügt wurde und das Rauschen einen größeren Einfluss hat. A4 enthält zusätzlich das Problem, dass sich die saisonale Komponente oder der Trend ändern können.

A1 enthält viele Zeitreihen mit fehlerhaft gekennzeichneten Anomalien. Des Weiteren lassen sich die Anomalien aufgrund ihres großen Bereichs und weil sie häufig am Ende der Sequenz auftreten, nicht so gut approximieren. Aus diesem Grund wird Datensatz A1 in diesem Projekt nicht weiterverwendet. Das Modell wird zunächst mit Datensatz A2 optimiert. Abschließend soll getestet werden, ob Anomalien auch in Datensatz A3 und A4 erkannt werden.

5.2. Das Konzept des gleitenden Fensters

Die Dimension des Eingabevektors kann während des Durchlaufes des Programms nicht verändert werden. Zudem stehen nur eine begrenzte Anzahl von Datensätzen (100 Zeitreihen im A2 Datensatz) für Optimierung des Netzes zur Verfügung, da die Anwendung zunächst mit möglichst einfachen und einheitlichen Zeitreihen optimiert werden soll. Die Zeitreihen enthalten mehrere Perioden, sodass eine Aufteilung der Zeitreihen in mehrere Teile zu einfacheren und mehr Trainingsdatensätzen führen könnte.
Voraussetzung ist jedoch, dass die einzelnen Teilzeitreihen alle dieselbe Größe haben müssen, da die Zahl der Eingabewerte nicht verändert werden kann. Da alle Zeitreihen der Gruppen A2, A3 und A4 jeweils die gleiche Länge aufweisen, können die Zeitreihen in der Auswertung nur dann vollständig berücksichtigt werden, wenn die Größe der einzelnen Teile der Sequenz ein Teiler der Gesamtlänge der Zeitreihe ist.
Für dieses Experiment, das in Kapitel 5.4 durchgeführt und ausgewertet wird, vereinfacht uns das die Arbeit, da die Anzahl der auswählbaren Parameter beschränkt wird. Zugleich macht es die Ergebnisse leichter auswertbar und einfacher umsetzbar, da immer alle Datenpunkte einer Zeitreihe mitberücksichtigt werden und Werte leichter verglichen werden können. Da die Perioden der Zeitreihen unterschiedlich lang sind, wäre eine einheitliche Teilung der Zeitreihen in die Länge der Perioden nicht möglich und so bringt die Festlegung der Fenstergröße auf einen Teiler der Gesamtsequenz keine großen Nachteile.
Die Aufbereitung der Datensätze mit dem gleitenden Fenster geschieht für die Trainings- und Testphase auf unterschiedliche Weise.

In der Trainingsphase erzeugt das Fenster immer neue Zeitreihen, indem es auf der eingelesenen Zeitreihe immer um eine konstante Schrittzahl weitergeschoben wird. So wird erreicht, dass die Zeitreihen aus unterschiedlichen Perspektiven trainiert werden. Eine kleine Schrittzahl führt damit zu sehr viel mehr Trainingszeitreihen. Um alle Datenpunkte einer Zeitreihe in das Training miteinzubeziehen, darf die Schrittzahl höchstens die Größe des Fensters aufweisen.
In der Testphase soll jeder Datenpunkt nur einmal berücksichtigt werden. Dies wird erreicht, in dem das Fenster immer genau um seine eigne Größe weitergeschoben

wird. Da die Größe des Fensters ein Teiler der Gesamtlänge ist, geht diese Aufteilung genau auf.

5.3. Ablauf des Programms

Im folgenden Abschnitt wird der Ablauf der Anwendung erläutert, um ein Verständnis für die Arbeitsweise des Modells zu erhalten. Anschließend wird das Modell optimiert. Ein verkürztes Testprotokoll, dass die wichtigsten Testläufe dokumentiert und auswertet, findet sich in Anhang B und wird in Kapitel 5.4. zusammengefasst.

Der Ablauf des Programms kann in fünf Schritte eingeteilt werden:

1. Verarbeitung der Benutzereingabe
2. Laden und Aufbereiten der Datensätze
3. Training des Modells
4. Testphase des Modells
5. Auswertung des Modells

1. Verarbeitung der Benutzereingabe

Nach Start des Programms öffnet sich ein Fenster, in dem der Benutzer ausgewählte Parameter festlegen kann, um das neuronale Netz zu initialisieren.
Festgelegt werden müssen die Datensätze, die in die Trainings- und Testphase miteinbezogen werden sollen. Zudem muss der Testdatensatz angegeben werden, der graphisch ausgewertet werden soll.
Ein weiterer Parameter ist die Dauer des Trainings durch Angabe der Epochenanzahl. Das Programm terminiert zusätzlich, wenn die angegebene Epochenzahl noch nicht erreicht wurde, aber das Programm in einem Trainingsschritt den Gesamtfehler nicht verkleinern konnte.
Die Größe des Fensters, welche die Anzahl der Inputwerte bestimmt und die Schrittzahl, die festlegt, wie weit das Fenster während des Trainings weitergeschoben werden soll, müssen ebenfalls festgelegt werden. Startet der Benutzer das Programm, werden die Parameter an die Klasse *YahooAnomalyDetection* weitergegeben, die das neuronale Netz konfiguriert, trainiert und testet.

2. Laden und Aufbereiten der Datensätze

In der Klasse *YahooAnomalyDetection* werden die Datensätze zunächst aus den CSV-Dateien geladen und in den Klassen *TrainDataSetIterator* und *TestDataSetIterator* entsprechend dem Konzept des gleitenden Fensters so aufbereitet, dass ihre Teile alle der Größe des Fensters und so der Knotenanzahl der Eingabeschicht entsprechen. Da Trainings- und Testdatensätze in diesem Modell unterschiedlich behandelt werden, sind zwei Objekte der Klasse DataSetIterator nötig.
Mit der *next()*-Methode werden alle Datensätze des Iterators auf einmal in das Modell geladen.

3. Training des Modells

Die Anzahl der Zeitreihen in der Trainingsphase ist abhängig von der Fenstergröße und der angegebenen Schrittzahl. Zudem wird in jeder Epoche der Gesamtfehler der Rekonstruktion berechnet. Hierfür wird zunächst der Fehlerwert E_{ij} für jeden Wert einer Zeitreihe i bestimmt.

$$E_{ij} = (in_{ij} - pred_{ij})^2$$

Anschließend werden die einzelnen Fehlerwerte aufaddiert, um den Gesamtfehler für jede Zeitreihe zu berechnen.

$$Err_i = \sum_{j \in train} E_{ij}$$

Die Summe der Fehler für jede Zeitreihe der Epoche stellt dann den Gesamtfehler für diese Trainingsepoche dar.

$$Score_n = \sum_{i \in n} Err_i$$

Das Training wird beendet, wenn entweder die angegebene Epochenzahl erreicht wurde oder der Gesamtfehler einer Epoche während des Trainings größer anstatt kleiner wird und so der Algorithmus keine Verbesserung in einem Trainingsschritt erreicht.

4. Testphase des Modells

Der Testdatensatz enthält die ursprünglichen Zeitreihen, entsprechend der Fenstergröße geteilt. Die Datensätze werden mit der Methode *activateSelectedLayers(int from, int to, INDArray)* in das Netz gegeben, ohne die Gewichte des Netzes anzupassen.

Für die Auswertung der Testphase werden zwei unterschiedliche Gesamtfehler berechnet. Zum einen der Gesamtfehler über die als „normal“ gekennzeichneten Datenpunkte und zum anderen der Gesamtfehler für alle Datenpunkte, auch den Anomalien.

Ziel der Anwendung ist es, die Zeitreihen ohne Anomalien möglichst gut darzustellen. Als Maß zum Vergleich der Testläufe wird deshalb der Gesamtfehler ohne Anomalien $ScoreOA$ ins Verhältnis zum Gesamtfehler über alle Daten $ScoreMA$ gesetzt. Zudem kann anhand des Vergleichs des Gesamtfehlers der letzten Epoche des und des Gesamtfehlers der Testdaten, eingeschätzt werden, ob Overfitting auftritt. Overfitting bedeutet, dass das Programm die Trainingsdaten zu genau lernt, sodass es die gelernten Muster nicht auf neue Daten übertragen kann.

$$RelScore_{test} = \frac{ScoreOA}{ScoreMA} = \frac{\sum_{j \in test} ScoreOA_j}{\sum_{k \in test} ScoreMA_k}$$

5. Auswertung des Modells

Die Anwendung gibt Informationen über den Lernfortschritt des Modells über die Konsole aus. Eine graphische Auswertung wird durch Öffnen eines Fensters, nach Durchlauf der Anwendung angezeigt. Enthalten sind Diagramme, welche die Eingabe und Prognose für eine ausgewählte Zeitreihe des ursprünglichen Testdatensatzes anzeigen. Zudem werden für diese Zeitreihen auch die jeweiligen Fehlerwerte E_{ij} dargestellt. Dadurch kann eingeschätzt werden, ob die Anomalien bei dieser Parameterzusammensetzung lokalisiert werden können. Ein weiteres Diagramm zeigt den Lernfortschritt des Netzes während des Trainings.

Diese Diagramme wurden, neben den vorgestellten Fehlerwerten, bei der Optimierung des Modells zur Analyse verwendet und dienen zur Dokumentation der Testreihen.

5.4. Optimierung und Training des Modells

Als Ausgangssituation wird das Modell aus dem EEG-Anwendungsfall übernommen und in mehreren Testphasen optimiert, die in diesem Abschnitt auf die wichtigsten reduziert und kurz zusammengefasst werden.

1. Testphase: Optimierung von Fenstergröße und Schrittzahl

Begonnen wurde in der ersten Testphase mit der Optimierung von Fenstergröße und Schrittzahl, um zu sehen, ob sich das Konzept sinnvoll umsetzen lässt.
In mehreren Testreihen wurden die Testfälle immer weiter eingegrenzt und mit unterschiedlicher Anzahl an Trainingsdaten getestet. Die Ergebnisse waren großenteils sehr schlecht. Ein Ergebnis zeigte jedoch eine deutliche Annäherung der Testdaten bei Verwendung von nur einem Trainingsdatensatz. Dieses Testbeispiel hat eine Fenstergröße von 203 sowie eine Schrittzahl von 203. 203 ist der größte Teiler der Gesamtlänge der Sequenz von 1421. Dies zeigt, dass die Betrachtung der Zeitreihen aus mehreren Perspektiven und in kleineren Abschnitten keine Verbesserung der Performance bringt. Ist die Fenstergröße gleich der Schrittzahl, so wird die Zeitreihe, während des Trainings einfach nur in kleinere Teile zerlegt, aber jeder Punkt gleich häufig trainiert.

2. Testphase: Optimierung von Lernrate und Epochenzahl

In der ersten Testphase wurden die Daten nur über 100 Epochen trainiert. Es zeigte sich deutlich, dass die Sättigung des Lernfortschritts noch nicht erreicht wurden. In der zweiten Testphase wurde deshalb die Trainingszeit optimiert, indem die Lernrate und die Epochenzahl angepasst wurden.
Zunächst wurde die Anzahl der Trainingsepochen so hoch gesetzt, dass das Netz vorher terminiert, weil kein Lernfortschritt mehr erreicht wird. Da nicht nur der Gesamtfehler, sondern auch die Änderungen des Gesamtfehlers immer kleiner werden und sich die Trainingszeit zugunsten winziger Verbesserungen stark verlängert, wird eine zusätzliche Abbruchbedingung eingeführt. Diese ist erfüllt, wenn der Gesamtfehler in einer Epoche kleiner als 50 wird.
Abbildung 19 zeigt das graphische Ergebnis des Autoencoders mit 10 Trainingszeitreihen, die erneut in der Testphase verwendet werden. Die Testdaten

sind dem Autoencoder also schon bekannt. Die blaue Kurve zeigt die ursprüngliche Zeitreihe, die grüne die Rekonstruktion und rot die Fehler für die einzelnen Zeitpunkte der angezeigten Testreihe.

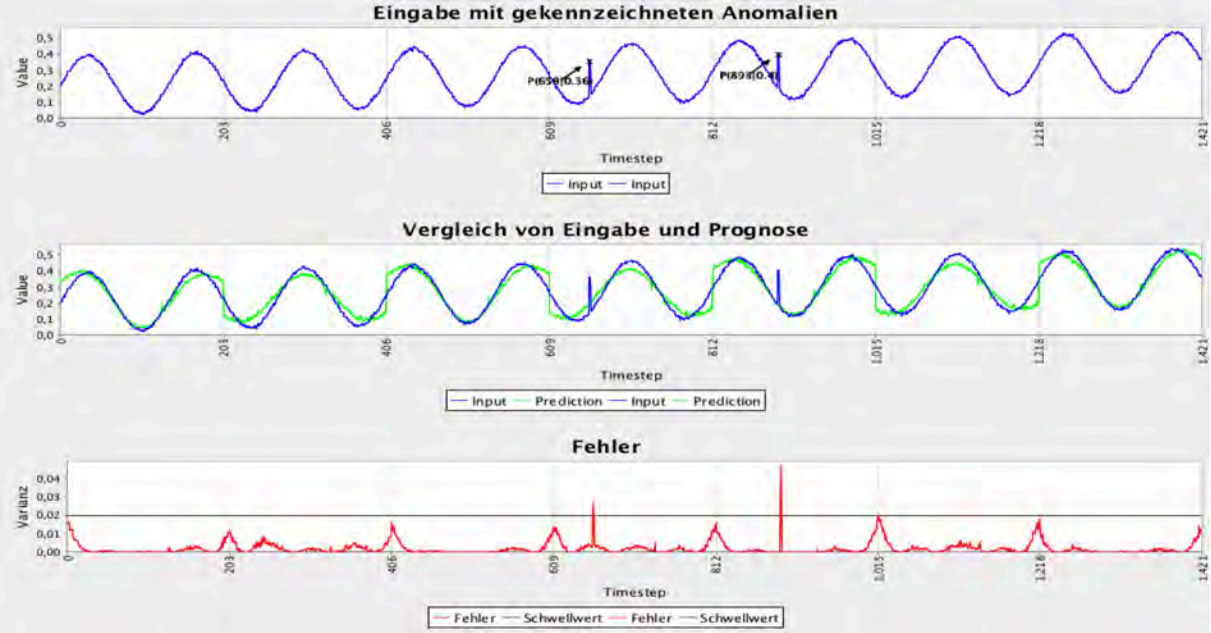


Abbildung 19 Auswertung der Rekonstruktion mit bekannten Daten.

Wie Abbildung 20 zeigt, besitzt das Modell die Fähigkeit zur Generalisierung, da es auch auf unbekannten Zeitreihen nur leicht schlechtere Ergebnisse liefert als bekannte Testdaten. Die Anomalie wird ebenfalls eindeutig erkannt.

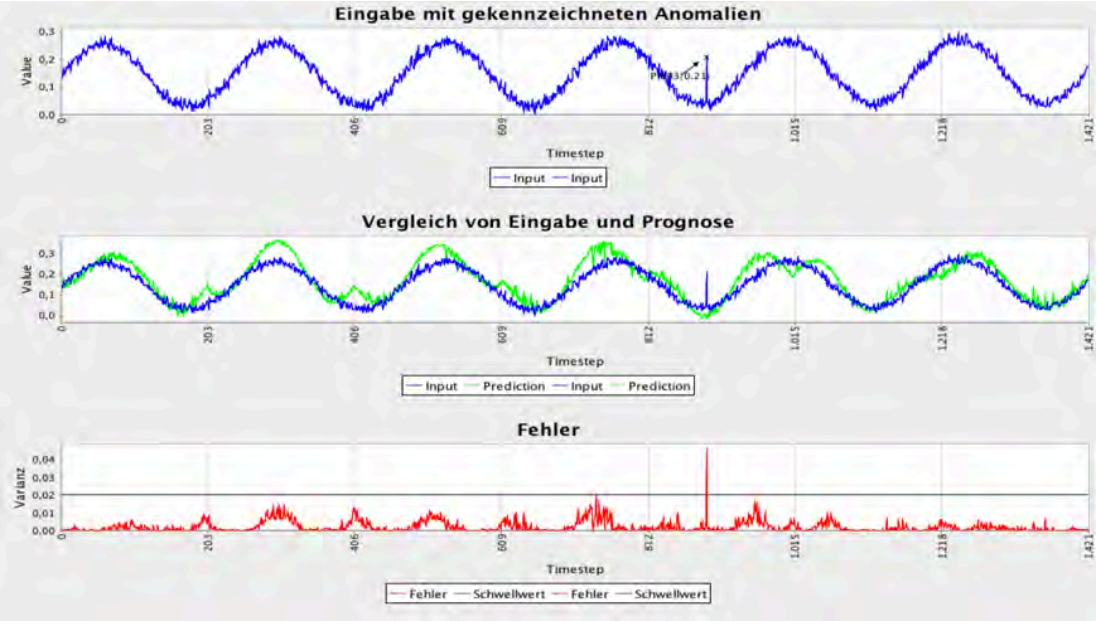


Abbildung 20 Auswertung der Rekonstruktion mit unbekannten Daten.

3. Testphase: Variation der versteckten Schichten

Der Autoencoder mit drei versteckten Schichten lässt sich also trainieren. In der dritten Testphase wurde das Modell auch mit mehr versteckten Schichten getestet, um zu sehen, ob es in der Lage ist, den Fehler auch über viele Schichten konstant zu halten. Zunächst wurde das Modell mit immer kleiner werdenden Schichten getestet. Abbildung 21 zeigt das Ergebnis eines Autoencoders mit der folgenden Knotenverteilung [203, 150, 100, 50, 20, 50, 100, 150, 203] bei einer Lernrate von 0.2.

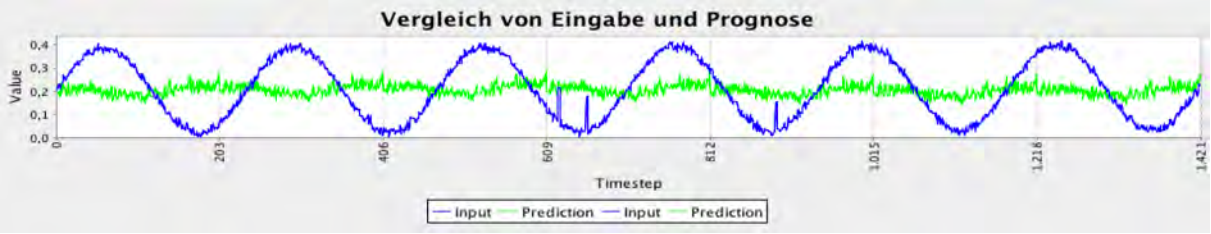


Abbildung 21 1. Beispiel: Autoencoder mit tiefer Struktur [203, 150, 100, 50, 20]

Das Programm terminiert bei einem Gesamtfehler von über 240, weil in einer Epoche keine Verbesserung mehr erzielt wurde. Besitzt die Zeitreihe einen Trend, so wird die Schätzung in den einzelnen Fenstern vertikal verschoben, wie die folgende Abbildung für dieselbe Schichtenverteilung zeigt.

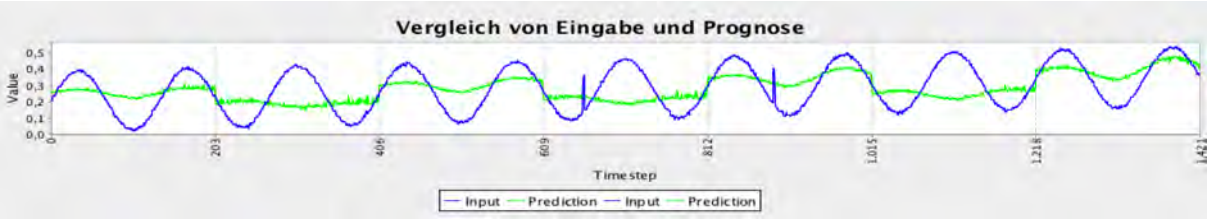


Abbildung 22 2.Beispiel: Autoencoder mit tiefer Struktur [203, 150, 100, 50, 20]

Das Modell ist so nicht in der Lage, den Fehler über viele Schichten konstant zu halten. Es scheint sich Durchschnittswerten anzunähern, da die Prognose in jedem Fenster ähnlich ist. Auffällig ist außerdem, dass die Änderung des Gesamtfehlers während der Trainingsphase zunächst abnimmt und teilweise sehr lange nahe null bleibt, sodass das das Netz in vielen Fällen terminiert, weil kein Fortschritt erreicht wurde. In manchen Fällen fängt sich die Anpassung wieder und wird wieder deutlich größer, bevor sie erneut abnimmt und das Programm terminiert. Doch auch in diesen Fällen

wird keine Rekonstruktion der Eingabe erzeugt, sondern nur ein verbesserter Durchschnittswert.

DL4J gibt an, dass die erste und die letzte versteckte Schicht eine höhere Knotenzahl als die Eingabeschicht haben sollten. Dies wurde ebenfalls getestet, brachte aber ähnliche Ergebnisse wie der gerade vorgestellte Fall. Es wurden zahlreiche Tests mir verschiedener Zusammenstellung von Knoten, Schichten und der Lernrate gemacht. Die aus dem EEG-Modell übernommenen Parameter wurden überdacht und teilweise neu getestet. Ein Modell mit vielen Schichten konnte aber nicht erfolgreich trainiert werden.
Dies bedeutet nicht, dass das Modell mit vielen Schichten generell nicht trainiert werden kann, da es zu viele Parameterkombinationen gibt, um alle Möglichkeiten testen zu können.
Um zu prüfen, ob der Deep Autoencoder ohne Teilung der Zeitreihen besser funktioniert als mit dem Konzept des gleitenden Fensters, wurde auch dieses Modell getestet. (Fenstergröße = 1421 = Länge der Zeitreihe). Es zeigte sich, dass die Teilung der Zeitreihe über viele Epochen bessere Ergebnisse zeigt, als das Training mit der gesamten Zeitreihe in einem Stück. Einschichtige und dreischichtige Autoencoder sind ebenfalls mit diesem Modell trainierbar, zeigen aber etwas schlechtere Ergebnisse. Mehrschichte Modelle zeigten ebenfalls keine erkennbare Rekonstruktion.

4. Testphase: Test der funktionierenden Autoencoder mit mehr Trainingsdaten aus Datensatz A2, A3 und A4

In der vierten Testphase wurde das Modell mit drei versteckten Schichten auf die Datensätze A2, A3 und A4 abschließend anwendet. Es sollte herausgefunden werden, wie die Verwendung von vielen Trainingszeitreihen das Modell beeinflussen. Zudem sollte geprüft werden, ob der Autoencoder mit drei Schichten dazu in der Lage ist die Rekonstruktion so präzise zu erstellen, dass auch die schwachen Anomalien in den komplexeren Zeitreihen aus A3 und A4 lokalisiert werden können.
Zunächst sollten jeweils 50 Zeitreihen trainiert und 50 Zeitreihen als Testdaten verwendet werden. Nach 60000 Epochen und über zwei Stunden Trainingszeit wurde das Training abgebrochen. Die Abbruchbedingung durch den Gesamtfehler während

des Trainings muss für verschiedene Zeitreihen unterschiedlich gewählt werden, um die Trainingszeit effizienter zu gestalten. Da zu diesem Zeitpunkt des Projekts aus Zeitgründen keine weiteren Tests durchgeführt werden können, um das Training zu optimieren, wird sich darauf beschränkt, die Auswertung mit 20 Zeitreihen durchzuführen und den Gesamtfehler bei 50 zu belassen.

Abbildung 23 zeigt die Ergebnisse für Datensatz A2 mit 20 Trainingszeitreihen. Ein Gesamtfehler von weniger als 50 wurde nach 27175 Epochen erreicht. Das erste Diagramm zeigt die Eingabe (blau) mit gekennzeichneten Anomalien. Das zweite stellt Eingabe (blau) und Schätzung (grün) gegenüber und das dritte Diagramm zeigt den Fehler (rot) der ausgewählten Zeitreihe für jeden Zeitpunkt.

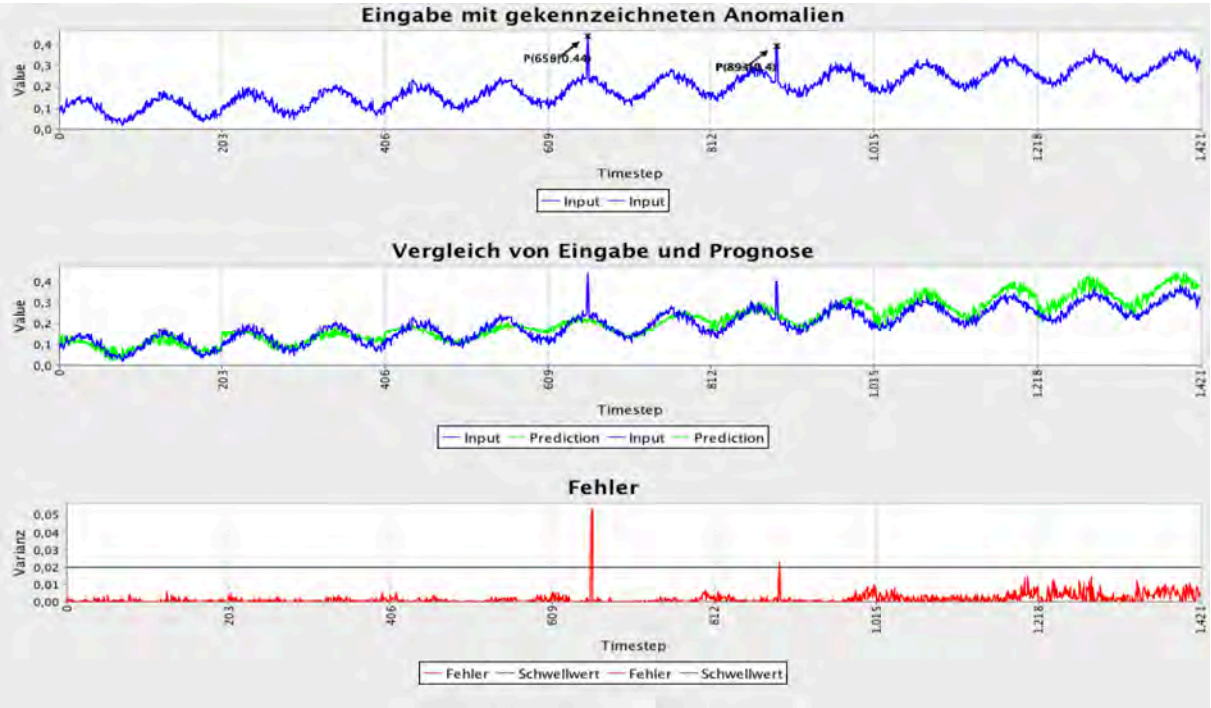


Abbildung 23 Beispiel aus Datensatz A2 nach 27 175 Epochen

Wie Abbildung 24 zeigt, sind auch die komplexeren Zeitreihen aus Datensatz A3 mit dem Autoencoder prognostizierbar. Trotzdem ist die Schätzung nicht so gut, dass alle Anomalien erkannt werden, wie Abbildung 25 zeigt. Dies könnte sich aber durch eine Verlängerung des Trainings noch verbessern. Auffällig ist, dass das Training bereits nach 5363 Epochen beendet wird, obwohl die Zeitreihen aus Datensatz A3 länger sind als die von A2. Der Grund hierfür könnte sein, dass die Zeitreihen in A2 größere

Schwankungen haben und deshalb nicht so schnell angenähert werden können, wie eine Zeitreihe mit flacherem Verlauf.

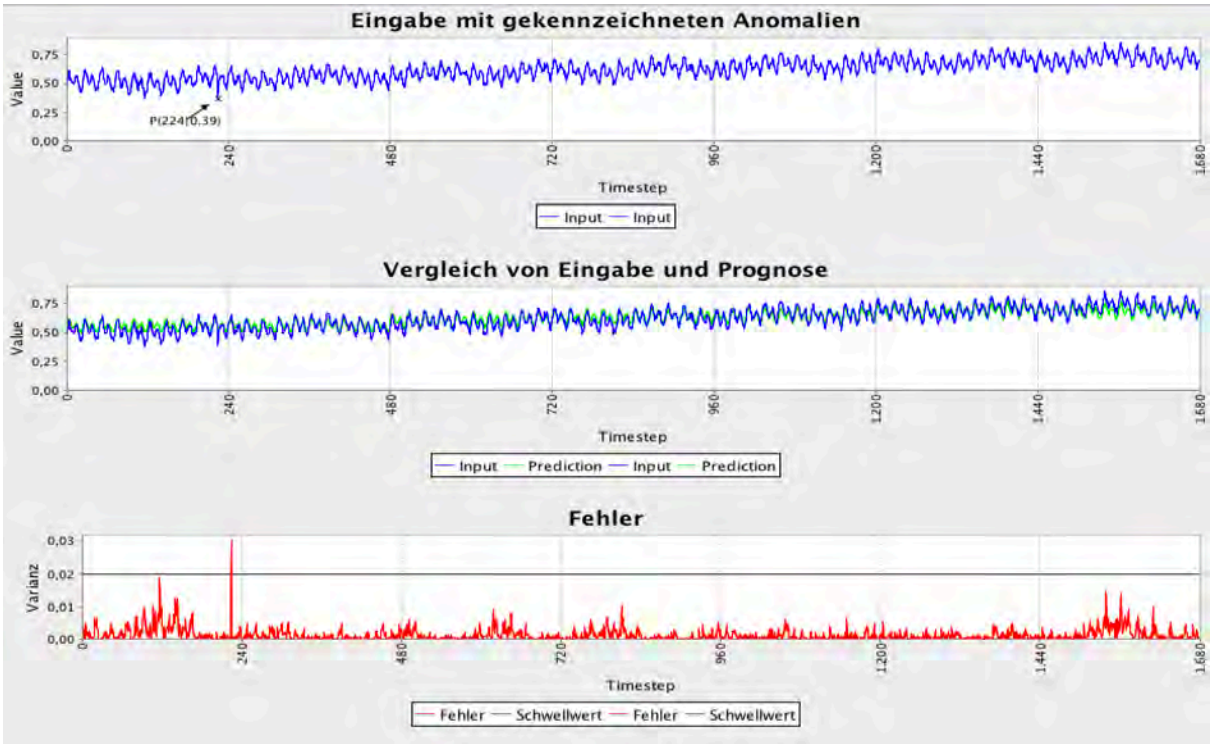


Abbildung 24 Beispiel aus Datensatz A3 nach 5363 Epochen

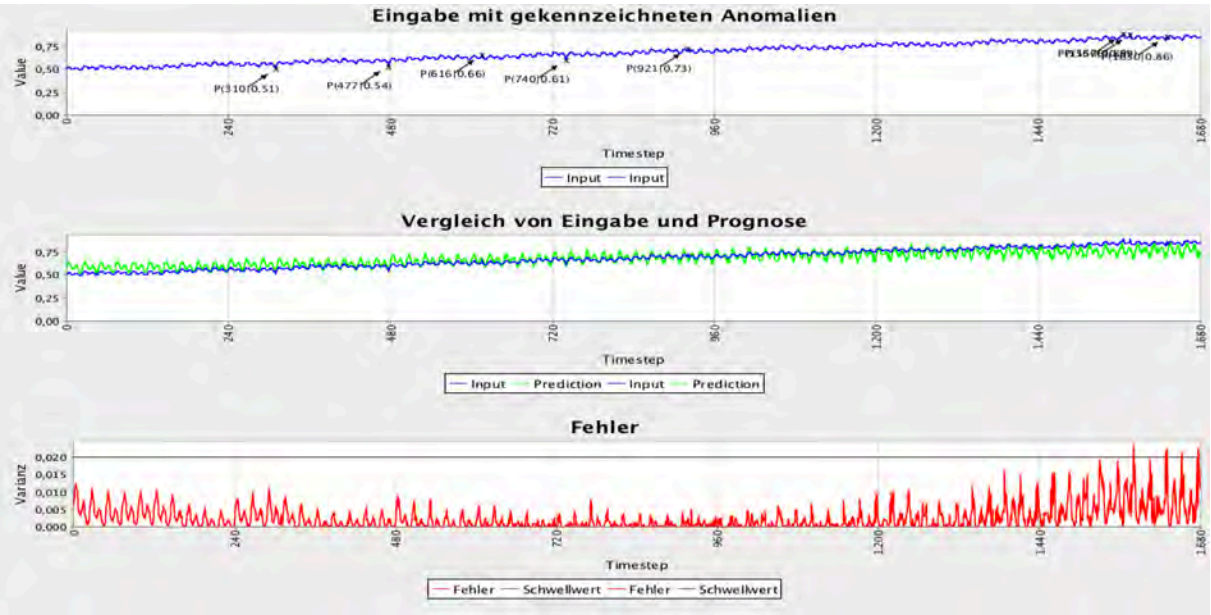


Abbildung 25 Beispiel aus Datensatz A3 nach 5363 Epochen

Die Ergebnisse für Datensatz A4 sehen ähnlich aus, wie die von Datensatz A3, wie Abbildung 26 verdeutlicht. Der Autoencoder schafft es auch die Wendepunkte, die eine Änderung der Eigenschaften der Zeitreihe markieren, mitzulernen. Dennoch sind die Schätzungen leicht schlechter als mit Datensatz A3 und deutlich schlechter als bei Verwendung von Datensatz A2, sodass die Anomalien auch hier nicht eindeutig lokalisiert werden können. Die Trainingszeit hat sich mit 8043 Epochen gegenüber Datensatz A3, wieder etwas erhöht, was für die Theorie bestätigt, dass Zeitreihen mit größeren Schwankungen langsamer angenähert werden können, als flachere.

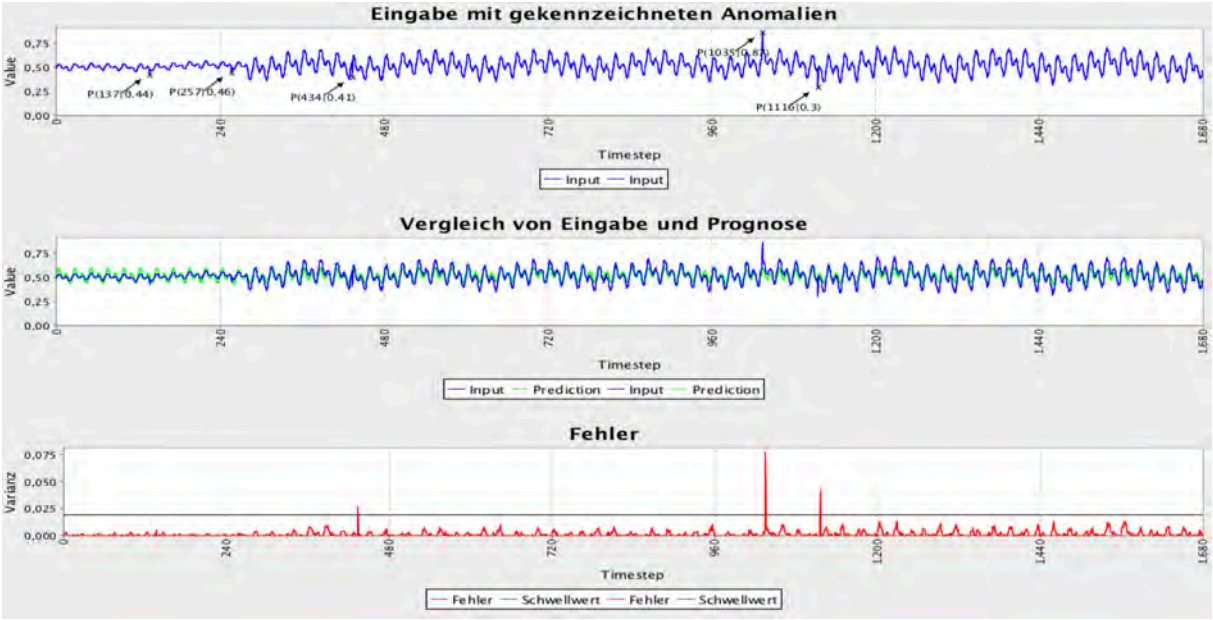


Abbildung 26 Beispiel aus Datensatz A4 nach 8043 Epochen

5.5. Zusammenfassung der Ergebnisse

Das Anwendungsbeispiel hat deutlich gezeigt, wie schwierig das Training von Autoencodern mit tiefer Netzstruktur ist. In dieser Arbeit konnten nur Autoencoder mit ein oder drei versteckten Schichten erfolgreich trainiert werden, da keine Parameterzusammensetzung gefunden werden konnte, die den Fehler des Netzes bei der Propagierung über viele Schichten konstant halten kann. Die Anwendung des gleitenden Fensters zeigt keine guten Resultate, ergab aber die Möglichkeit einer Verbesserung des Trainings durch Teilung der Zeitreihe in kleinere Abschnitte.
Das Modell mit drei versteckten Schichten zeigte die besten Ergebnisse und hat keine Laufzeitnachteile gegenüber dem einschichtigen Modell.

Jedoch ist die Trainingszeit mit vielen Datensätzen sehr lang und die Abbruchbedingung des Trainings müsste auf die Anzahl der trainierten Zeitreihen abgestimmt werden. Die Zeitreihen aus Datensatz A2 konnten so gut rekonstruiert werden, dass die Anomalien mit hoher Zuverlässigkeit erkannt werden. Auch die Rekonstruktion der Zeitreihen aus A3 und A4 ist möglich, aber nicht genau genug, um die meisten Anomalien zu lokalisieren.
Autoencoder mit tiefen Strukturen konnten leider nicht erfolgreich trainiert werden. Dies bedeutet aber nicht, dass Deep Autoencoder nicht mit DeepLearning4J implementiert werden können. Wie die dritte Testphase gezeigt hat, sind schon kleine Veränderungen der Parameter entscheidend für den Zeitraum und Fortschritt des Lernvorgangs. So kann nicht ausgeschlossen werden, dass Autoencoder mit tiefer Netzstruktur und einer anderen Parameterzusammensetzung erfolgreich trainiert werden könnten.

6. Fazit und Ausblick

Die Implementierung des Autoencoders zur Lokalisierung von Anomalien, hat deutlich gezeigt wie schwierig es ist, neuronale Modelle mit tiefer Netzstruktur erfolgreich zu trainieren. Das von DeepLearning4J zur Verfügung gestellte Material reichte nicht aus, um die Anzahl der Parameterkombinationen so einzuschränken, dass Modelle mit mehr als drei versteckten Schichten zufriedenstellend trainiert werden konnten.
Das Modell verfängt sich mit tiefer Struktur fast immer in schlechten Lösungen, welche den Gesamtfehler durch Annäherung an einen Mittelwert zwar reduzieren, aber keine Rekonstruktion der Zeitreihen darstellen.
Anders als für H2O konnte aber gezeigt werden, dass zumindest einfache Autoencoder erfolgreich trainiert werden können, um Zeitreihen zu rekonstruieren. Die erzeugten Rekonstruktionen zeigten zwar schon gute Ergebnisse, können aber durchaus noch verbessert werden.
Einschränkend muss festgehalten werden, dass die Trainingszeit für Autoencoder mit vielen Trainingszeitreihen sehr lang wird und hier eine Balance zwischen Trainingszeit und Verbesserung der Rekonstruktion gefunden werden muss.
Aufgrund ihrer tiefen Struktur könnten Deep Autoencoder dazu in Lage sein, detailliertere Rekonstruktionen in kürzerer Zeit zu erschaffen, wenn die Eingabe

schrittweise immer weiter abstrahiert wird. Dies könnte in aufbauenden Projekten untersucht werden.

Zudem hat sich gezeigt, dass der S5 Datensatz von Yahoo viele nicht gekennzeichnete Anomalien enthält, die das Training des Netzes beeinflussen können. Eine Überarbeitung der Datensätze könnte ebenfalls zu einem besseren Training beitragen.

Da sich herausgestellt hat, dass das Training von Autoencodern hohe Durchlaufzeiten hat, wäre es sinnvoll, das Modell durch eine bessere Abbruchbedingung zu optimieren. Bei neuronalen Netzen mit flacher Hierarchie konnte die Abbruchbedingung, die greift, wenn der Gesamtfehler sich nicht mehr verkleinert oder für eine Epoche kleiner als 50 wird, sinnvoll eingesetzt werden, da das Netz sich in jeder Epoche immer weiter verbessert. Für Netze mit vielen Schichten gilt dies nicht mehr, da der Gesamtfehler nicht stetig kleiner wird, sodass während des Trainings bislang nicht entschieden werden konnte, ob das Netz sich tatsächlich weiter verbessert oder sich nur nicht für eine Richtung der Anpassung entscheiden kann.

Durch die Komplexität der Zusammensetzung neuronaler Netzen werden zahlreiche Ansatzpunkte zur Verbesserung des Modells geboten. Vor allem Parameter, wie die Lernrate, haben schon bei kleinen Anpassungen großen Einfluss auf das Lernverhalten des Netzes.

Die große Anzahl der Möglichkeiten, die DeepLearning4J für die Wahl der Parameter bietet, machen die Modellierung komplex und aufwändig.

Da in dieser Arbeit nur mit synthetischen Zeitreihen gearbeitet wurde, wäre eine Anwendung auf reale komplexere Daten ebenfalls eine interessante Untersuchung, um zu sehen ob auch Muster erkannt werden, die für den Menschen nicht so offensichtlich zu identifizieren sind.

Literaturverzeichnis

[1] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. „Reducing the dimensionality of data with neural networks.“ Science 313.5786 (2006): 504-507.
Link: http://www.cs.toronto.edu/~hinton/ zuletzt abgerufen: 29.03.17 15:40

[2] Oxé, Julia. Anwendung von Deep Learning anhand von MNIST-Beispielen mit H2O und DeepLearning4J. Projektdokumentation im Rahmen des Praxisprojekts, Institut für Informatik, Lehrstuhl Prof. Dr. rer. Nat. Konen, Technische Hochschule Köln, 2017.

[3] Offizielle Website von H2O: http://docs.h2o.ai/h2o/latest-stable/index.html

[3a] Deep Learning Booklet: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/booklets/DeepLearningBooklet.pdf

[3b] Informationen zur Klasse AutoenocoderEstimator:
http://docs.h2o.ai/h2o/latest-stable/h2o-py/docs/modeling.html - h2oautoencoderestimator
http://docs.h2o.ai/h2o/latest-stable/h2o-py/docs/model_categories.html - module-h2o.model.autoencoder

[4] Informationen zum Yahoo S5 Datenset:
https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70

[5] Hochreiter, Sepp. Untersuchungen zu dynamischen neuronalen Netzen. Diss. Diploma Thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[6] Vincent, Pascal, et al. “Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.” Journal of Machine Learning Research 11. Dec (2010): 3371-3408.
Link: http://www.jmlr.org/papers/v11/vincent10a.html zuletzt abgerufen: 29.03.17 16:05

[7] Le, Quoc V. “A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks.” (2015).

Link:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.703.5244&rep=rep1&type=pdf zuletzt abgerufen: 29.03.17 15:55

[8] Hinton, Geoffrey. „A practical guide to training restricted Boltzmann machines.“ Momentum 9.1 (2010): 926.
Link: http://www.cs.toronto.edu/~hinton/ zuletzt abgerufen: 29.03.17 16:09

[9] Salakhutdinov, Ruslan, and Geoffrey E. Hinton. “Deep Boltzmann Machines.” AISTATS. Vol. 1. 2009.
Link:
http://www.jmlr.org/proceedings/papers/v5/salakhutdinov09a/salakhutdinov09a.pdf zuletzt abgerufen: 29.03.17 16:12

[10] Bengio, Yoshua, et al. “Greedy layer-wise training of deep networks.” Advances in neural information processing systems 19 (2007): 153.
Link:
https://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf zuletzt abgerufen: 29.03.17 15:53

[11] Hinton, Geoffrey E., Simon Osindero, and Yee-Whye The. “A fast learning algorithm for deep belief nets.” Neural computation 18.7 (2006): 1527-1554.
Link: http://www.cs.toronto.edu/~hinton/ zuletzt abgerufen: 29.03.17 16:33

[12] Backhaus, Klaus, et al. Multivariate Analysemethoden: Eine anwendungsorientierte Einführung. Springer-Verlag, 2015. S.135 -172

[13] Rumelhart, David E., Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." Cognitive modeling 5.3 (1988): 1.
Link:
http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA164453 zuletzt abgerufen: 29.03.17 16:42

[14] Hinton, Geoffrey E. "Learning multiple layers of representation." Trends in cognitive sciences 11.10 (2007): 428-434.
Link: http://www.cs.toronto.edu/~hinton/absps/tics.pdf zuletzt abgerufen: 29.03.17 16:42

[15] Lange, Sascha, Riedmiller, Martin. "Deep auto-encoder neural networks in reinforcement learning." Neural Networks (IJCNN), The 2010 International Joint Conference on. IEEE, 2010.

[16] Offizielle Website von DeepLearning4J: https://deeplearning4j.org/

[16a] Überblick: ModelConfigurationClass:
https://deeplearning4j.org/neuralnet-configuration.html
[16b] Tutorial: Deep Autoencoder
https://deeplearning4j.org/deepautoencoder
[16c] RecordReader zum Einlesen von csv-Dateien
https://deeplearning4j.org/csv-deep-learning
[16d] Dokumentation der Klasse INDArray
http://nd4j.org/doc/org/nd4j/linalg/api/ndarray/INDArray.html
[16e] Dokumentation der Klasse DataSetIterator
http://nd4j.org/doc/org/nd4j/linalg/dataset/api/iterator/DataSetIterator.html

[17] Offizielle Website von JFreeChart: http://www.jfree.org/jfreechart/

Anhang A: EEG Testprotokoll

Dieses Testprotokoll legt die Voraussetzungen und Ergebnisse für jeden Testlauf in der durchgeführten Reihenfolge dar.
Im Hauptteil der Arbeit werden die Ergebnisse aufgegriffen und aufgearbeitet. Dieses Protokoll dient lediglich der Nachvollziehbarkeit der durchgeführten Tests. Da in diesem Abschnitt ein anderes Beispiel nachprogrammiert wird, werden nur die Parameter getestet, die nicht aus dem H2O Beispiel übernommen werden konnten.

Bewertung der Ergebnisse
Das Ziel des Programms ist es, anomale Zeitreihen zu identifizieren, indem nur mit normalen Zeitreihen trainiert wird. Ein gutes Ergebnis zeichnet sich dadurch aus, dass die bereits trainierten Daten in der Testphase einen deutlich geringeren Fehlerwert aufweisen, als die unbekannten Testdaten. Dies ergibt sich, wenn die trainierten Zeitreihen möglichst gut angepasst werden können und die unbekannten Zeitreihen möglichst schlecht.
Als Maß zur Bewertung eines Testlaufs wird deshalb das Verhältnis des Fehlers der guten Zeitreihen mit dem Fehler aller Zeitreihen der Testphase ins Verhältnis gesetzt.

$$Score = \frac{GesErr_{normal}}{GesErr_{alle}} = \frac{\sum_{k=0}^{20} GesErr_k}{\sum_{k=0}^{23} GesErr_k}$$

Testphase: Optimierung der fehlenden Parameter

Ziel der Testphase: Optimierung des Modells durch Anpassung der fehlenden Parameter, sodass das Programm in der Lage ist, die „normalen“ Zeitreihen von den „anomalen“ Zeitreihen zu unterscheiden.

1.1. Initialisierung der Gewichte

Die Initialisierung bestimmt die Ausgangslage für das Training des Models, da sie festlegt wie die Gewichtungen der Kanten zu Beginn des Trainings verteilt werden.

Ziel: Auswahl der Initialisierungsmethode, welche die trainierten Daten am besten annähert.

Testfälle: XAVIER, UNIFORM, RELU, DISTRIBUTION, SIGMOID-UNIFORM, XAVIER_FAN_IN, XAVIER_LEGACY, XAVIER_UNIFORM, ZERO

Gegebene Parameter:

Anzahl der Knoten pro Schichten [210 | 50 | 20 | 50 | 210]
Lernrate = 0.1
Epochen = 100
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion der versteckten Schichten = SQUARE_LOSS
Verlustfunktion Ausgabeschicht = SQUARE_LOSS

Initialisierung	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
XAVIER_LEGACY	53.9	130.3506651	0.4135
XAVIER	45.19	107.6464983	0.4198
UNIFORM	55.8054	131.6165094	0.4240
XAVIER_FAN_IN	39.93	91.35209334	0.4371
RELU	33.18	65.21226415	0.5088
XAVIER_UNIFORM	52.57	100.573943	0.5257
SIGMOID_UNIFORM	129.97	190.8236676	0.6811
DISTRIBUTION	669.9437	856.7054987	0.7820

Fazit: XAVIER-LEGACY wird als Initialisierungsmethode gewählt, da diese den kleinsten Fehleranteil aufweist.

1.2. Update-Methode zur Gewichtsanpassung

Die Update-Methode legt fest wie die Gewichtungen während des Trainings angepasst werden.

Ziel: Auswahl einer Update-Methode, welche die trainierten Daten am besten annähert.

Testfälle: ADGRAD, NESTEROVS, ADAM, ADADELTA, RMSPROP, SGD

Gegebene Parameter:

Anzahl der Knoten pro Schichten [210 | 50 | 20 | 50 | 210]
Lernrate = 0.1
Epochen = 100
Initialisierungsmethode: XAVIER_LEGACY
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion der versteckten Schichten = SQUARE_LOSS
Verlustfunktion Ausgabeschicht = SQUARE_LOSS

Update-Methode	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
ADAGRAD	53.9	130.3506651	0.4135
NESTEROV	58.11	140.3623188	0.4140
ADAM	79.38	165.09	0.4808
ADADELTA	51.33	97.18	0.5282
RMSPROP	292.28	427.70	0.6834
SGD	875.48	1028.71	0.8510

Fazit: ADAGRAD wird als Update-Methode ausgewählt, da der Fehleranteil bei den normalen Zeitreihen am geringsten ist.

1.3. Optimierungsalgorithmus

Der Optimierungsalgorithmus legt fest wie die Gewichtungen angepasst werden.

Ziel: Auswahl eines Optimierungsalgorithmus, welcher die trainierten Daten am besten annähert.

Testfälle: LINE_GRADIENT_DESCENT,
STOCHASTIC_GRADIENT_DESCENT,
CONJUGATE_GRADIENT_DESCENT

Gegebene Parameter:
Anzahl der Knoten pro Schichten [210 | 50 | 20 | 50 | 210]
Lernrate = 0.1
Epochen = 100
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion der versteckten Schichten = SQUARE_LOSS
Verlustfunktion Ausgabeschicht = SQUARE_LOSS

Optimierungs-algorithmus	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
STOCHASTIC	53.90	130.36	0.4135
CONJUGATE	67.72	103.90	0.6518
LINE	79.38	165.09	0.6518

Fazit: STOCHASTIC_GRADIENT_DESCENT wird als Optimierungsalgorithmus ausgewählt, da der Fehleranteil bei den normalen Zeitreihen am geringsten ist.

1.4. Verlustfunktion der versteckten Schichten

Die Verlustfunktion legt fest wie die das Aktivitätslevel der versteckten Schichten vor der Ausgabe aufbereitet wird.

Ziel: Auswahl einer Verlustfunktion, welche die trainierten Daten am besten annähert.

Testfälle: SQUARE_LOSS, MSE, NEGATIVLOGLIKELIHOOD, KL_DIVERGENCE, RECONSTRUCTION_CROSSENTROPY, DEFAULT

Gegebene Parameter:

Anzahl der Knoten pro Schichten [210 | 50 | 20 | 50 | 210]
Lernrate = 0.1
Epochen = 100
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion Ausgabeschicht = SQUARE_LOSS

Verlustfunktion versteckte Schicht	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
SQUARE_LOSS	53.90	130.36	0.4135
MSE	53.90	130.36	0.4135
NEGATIVLOGLIKELIHOOD	53.90	130.36	0.4135
KL_DIVERGENCE	53.90	130.36	0.4135
RECONSTRUCTION_CROSS	53.90	130.36	0.4135
DEFAULT	53.90	130.36	0.4135

Fazit: Es macht keinen Unterschied, welche Verlustfunktion verwendet wird. Es wird sich für SQUARE_LOSS entschieden.

1.5. Verlustunktion der Ausgabeschicht

Die Verlustfunktion legt fest wie die das Aktivitätslevel der versteckten Schichten vor der Ausgabe aufbereitet wird.

Ziel: Auswahl einer Verlustfunktion, welche die trainierten Daten am besten annähert.

Testfälle: SQUARE_LOSS, MSE, NEGATIVLOGLIKELIHOOD, KL_DIVERGENCE, RECONSTRUCTION_CROSSENTROPY, DEFAULT

Gegebene Parameter:

Anzahl der Knoten pro Schichten [210 | 50 | 20 | 50 | 210]
Lernrate = 0.1
Epochen = 100
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion der versteckten Schicht = SQUARE_LOSS

Initialisierung	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
SQUARE_LOSS	53.90	130.36	0.4135
MSE	53.90	130.36	0.4135
NEGATIVLOGLIKELIHOOD	898.73	1031.43	0.8713
KL_DIVERGENCE	898.73	1031.43	0.8713
RECONSTRUCTION_CROSS	898.73	1031.43	0.8713
DEFAULT	898.73	1031.43	0.8713

Fazit: SQUARE_LOSS und MSE eignen sich beide gleichermaßen. Er wird mit MSE gearbeitet.

1.6. Optimierung der Lernrate

Die Lernrate ist ein sehr wichtiger Parameter, der schon bei kleinen Veränderungen Einfluss auf das Ergebnis haben kann. Da die Epochenzahl nicht verändert werden soll, wird die Lernrate in mehreren Schritten verfeinert. Die Lernrate bestimmt, wie groß die Anpassung der Gewichte in jedem Trainingsschritt ist.

Ziel: Auswahl einer Lernrate, welche die Eingabe am besten annähert.

Testfälle: 0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.5

Gegebene Parameter:

Anzahl der Knoten pro Schichten [210 | 50 | 20 | 50 | 210]
Epochen = 100
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion der versteckten Schichten = SQUARE_LOSS
Verlustfunktion Ausgabeschicht = MSE

Lernrate	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
0.01	72.11	91.09	0.7916
0.02	69.59	101.30	0.6869
0.05	59.99	131.877	0.4549
0.1	53.90	130.36	0.4135
0.2	71.5097	144.15	0.4960
0.3	38.36	76.67	0.5003
0.5	46.56	94.45	0.4930

Zwischenfazit: Eine Lernrate um 0.1 scheint optimal zu sein. Zur Verfeinerung werden noch einige Tests gemacht. Zunächst um den Wert 0.1 herum, um zu sehen in welche Richtung der Fehler kleiner wird.

Weitere Testfälle: 0.0975, 0.0925, 0.125, 0.15

Lernrate	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
0.095	53.48	130.42	0.4101
0.0975	53.67	130.37	0.4117
0.125	73.18	94.88	0.7713
0.15	77.74	104.95	0.7407

Zwischenfazit: Der Fehler wird bei einer kleineren Lernrate auch geringer, aus diesem Grund wird weiter zwischen 0.06 und 0.09 gesucht.

Weitere Testfälle: 0.09, 0.08, 0.07, 0.06

Lernrate	$GesErr_{normal}$	$GesErr_{alle}$	$Score$
0.09	53.32	130.76	0.4077
0.08	53.93	132.56	0.4069
0.07	55.33	135.16	0.4094
0.06	57.02	137.57	0.4145

Fazit: Es wird eine Lernrate von 0.8 festgelegt, da dieser Testlauf den niedrigsten Fehleranteil aufweist. Mit diesem Modell wird auch im nächsten Beispiel weitergearbeitet werden.

Ergebnisse der EEG Testphase

Festgelegte Parameter:

Anzahl der Knoten pro Schichten [210 | 50 | 20 | 50 | 210]

Epochen = 100

Lernrate: 0.8

Initialisierungsmethode: XAVIER_LEGACY

Update-Methode: ADAGRAD

Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT

Aktivitätsfunktion der versteckten Schichten = TANH

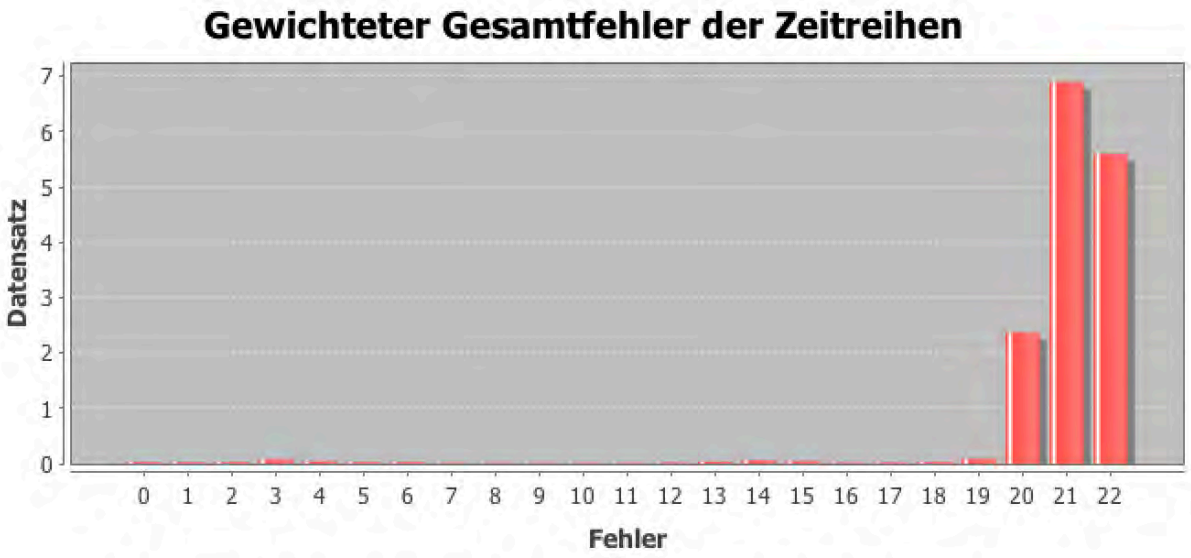
Aktivitätsfunktion der Ausgabeschicht = IDENTITY

Verlustfunktion der versteckten Schicht = SQUARE_LOSS

Verlustfunktion der Ausgabeschicht = MSE

Die folgende Abbildung zeigt, den Gesamtfehler für jede Zeitreihe in der Testphase. Die „normalen“ Zeitreihen weisen einen Gesamtfehler unter 1.0 auf und sind damit deutlich von den „anormalen“ Zeitreihen (20, 21, und 22) zu unterscheiden

Die genauen Werte sind der nachfolgenden Tabelle zu entnehmen.



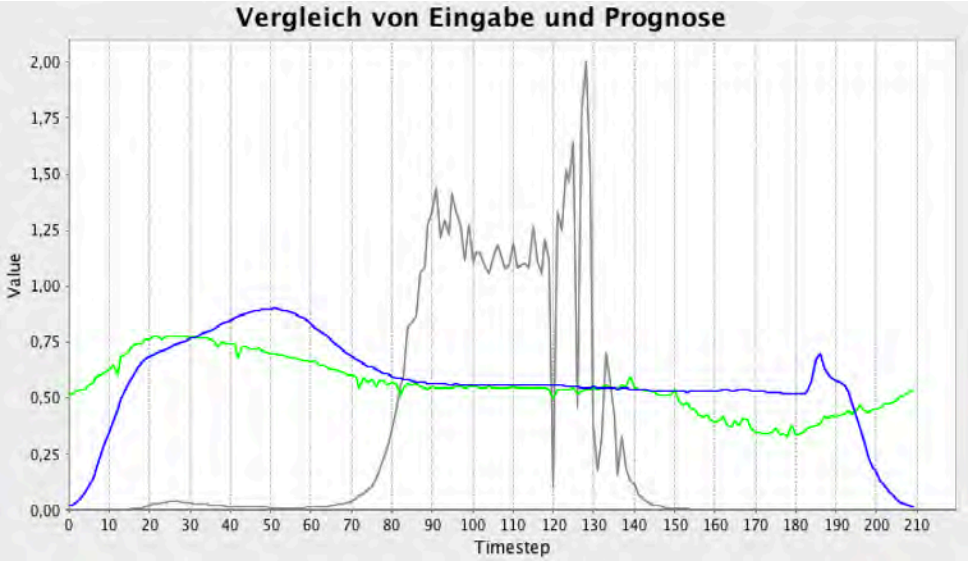
$i = ID$	$GesErr_i$	$i = ID$	$GesErr_i$	$i = ID$	$GesErr_i$	$i = ID$	$GesErr_i$
0	0.0319	**6**	0.0298	**12**	0.0264	**18**	0.0339
1	0.0297	**7**	0.0149	**13**	0.0408	**19**	0.0940
2	0.0313	**8**	0.0182	**14**	0.0676	**20**	2.3735
3	0.0877	**9**	0.0169	**15**	0.0491	**21**	6.9034
4	0.0463	**10**	0.0155	**16**	0.0201	**22**	5.5995
5	0.0280	**11**	0.0184	**17**	0.0223		

Es ist deutlich erkennbar, welche Zeitreihen (20, 21 und 22) nicht ins Training mit einbezogen wurden, da ihre Fehlerwerte sehr viel größer sind, als die der „normalen“ Zeitreihen.

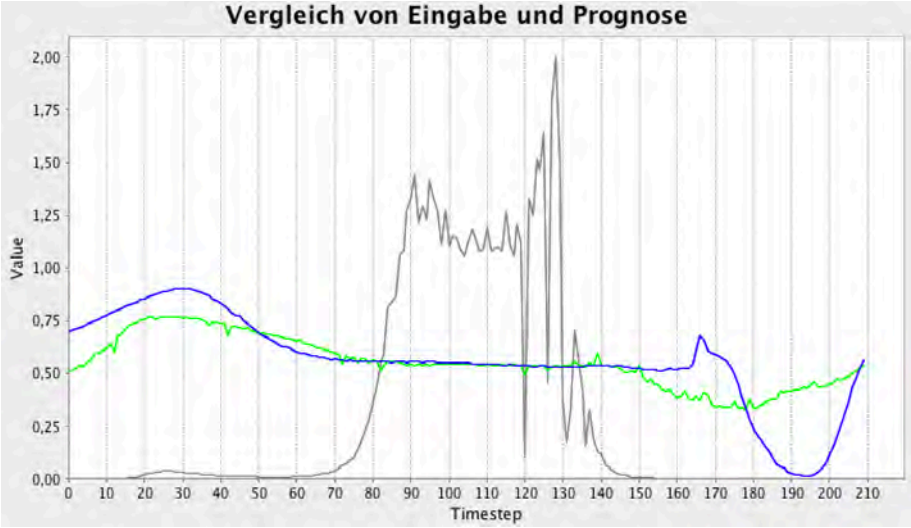
Zum Abschluss betrachten wir das graphische Ergebnis jeweils einer Zeitreihe einer Kategorie, um zu sehen wie gut, die einzelnen Funktion angenähert wurden.

Auffällig ist das Zeitreihe 20, die einen anderen Wertebereich und einen völlig anderen Verlauf hat als alle anderen Zeitreihen, einen geringeren Fehlerwert aufweist, als die beiden anderen „anomalen“ Zeitreihen 21 und 22.

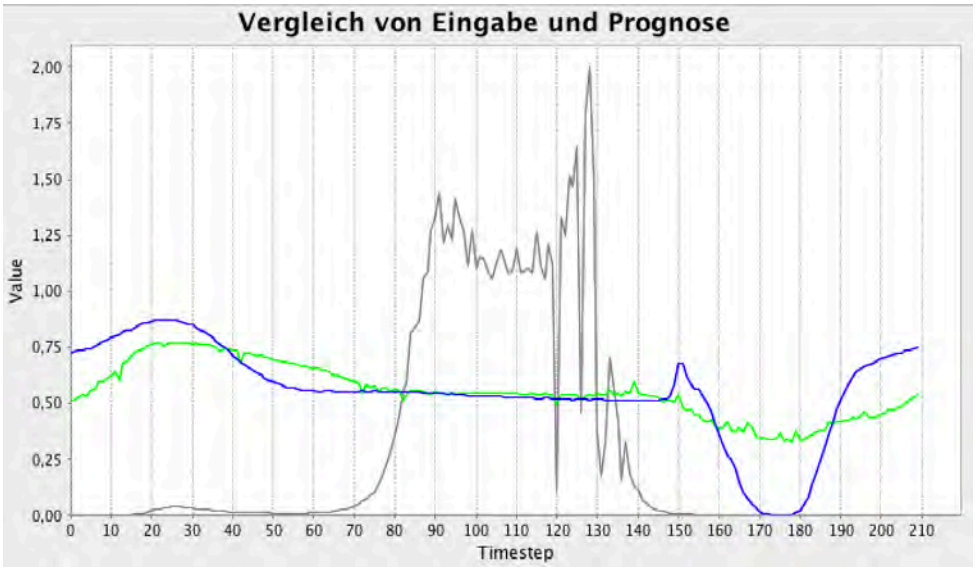
Sequenz 0 aus Kategorie A:
Varianz aus Kategorie A zwischen: 0.0372 – 0.1008



Sequenz 8 aus Kategorie B:
Varianz aus Kategorie B zwischen: 0.0191 - 0.0363

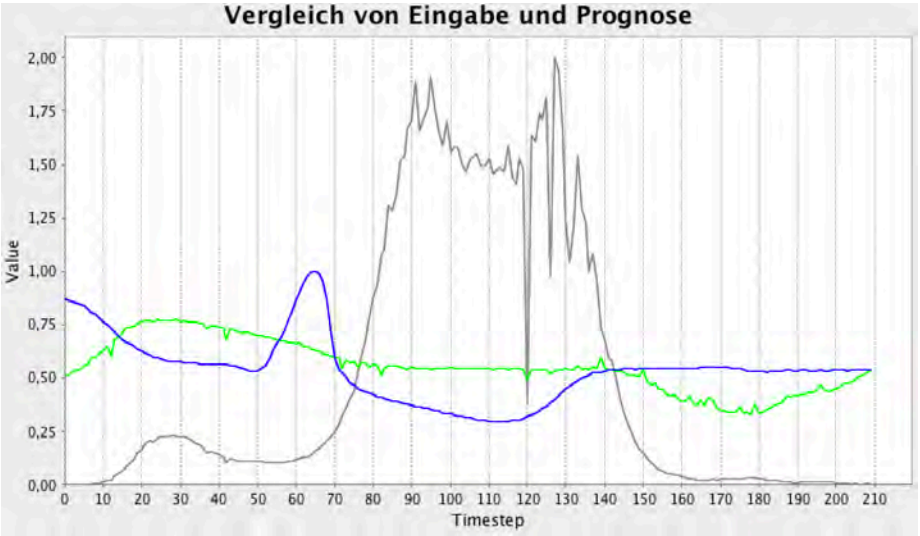


Sequenz 12 aus Kategorie C:
Varianz aus Kategorie C zwischen: 0.0202 - 0.0906

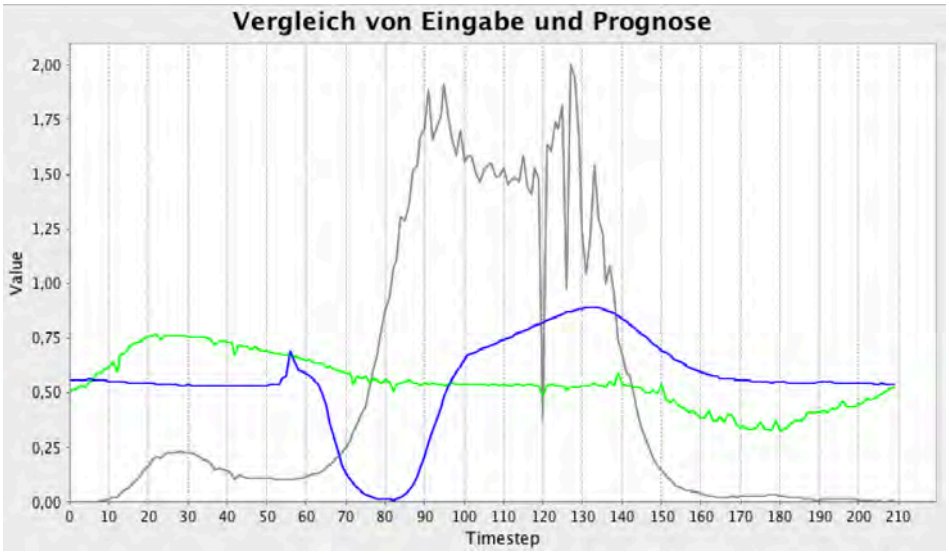


Sequenz 20 aus Kategorie D:

Varianz aus Kategorie D: 2.3492



Sequenz 22 aus Kategorie E:
Varianz aus Kategorie E zwischen: 5.4161 - 6.6941



Anhang B: Testprotokoll zur Lokalisierung von Anomalien

In diesem Testprotokoll soll nun das zweite Anwendungsbeispiel getestet werden. Die Parameter des Modells werden zunächst aus dem EE
G-Beispiel übernommen und sollen während der Testphase auf die neue Situation angepasst werden.
Dieses Protokoll dient der Nachvollziehbarkeit der durchgeführten Tests und daraus resultierenden Schlussfolgerungen. Anders als im EEG-Beispiel wird die Zeitreihe in mehreren Teilfragmenten trainiert. Welche Teile dies sind, wird zum einen durch die festgelegte Fenstergröße bestimmt und zum anderen durch die Schrittzahl, die angibt wie weit das Fenster für jeden neuen Trainingsdatensatz nach vorne geschoben wird.

Bewertung der Ergebnisse

Das Ziel des Programms ist es, Anomalien zu lokalisieren. Dies wird erreicht, indem die Anomalien während des Trainings nicht miteinbezogen werden.
Bereits während der Trainingsphase wird für jeden Datenpunkt der Zeitreihen ein Fehler berechnet, der sich aus der Differenz von Eingabe und Prognose ergibt.

$$E_{ij} = (in_{ij} - pred_{ij})^2$$

Anschließend werden die einzelnen Fehlerwerte aufaddiert, um den Gesamtfehler für jede Zeitreihe i zu berechnen.

$$Err_i = \sum_{j \in train} E_{ij}$$

Die Summe der Fehler für jede Zeitreihe stellt dann den Gesamtfehler für die Trainingsepoche n da.

$$Score_n = \sum_{i \in n} Err_i$$

Das Training wird beendet, wenn entweder die angegebene Epochenzahl erreicht wurde oder der Gesamtfehler einer Epoche während des Trainings größer und nicht kleiner wird. Der Algorithmus also keine Verbesserung in einem Trainingsschritt erreicht.
Ziel der Anwendung ist es die Zeitreihen ohne Anomalien möglichst gut zu rekonstruieren, sodass die Abweichungen an „anomalen“ Stellen möglichst groß sind. Als Maß zum Vergleich der Testläufe wird deshalb der Gesamtfehler ohne Anomalien $\mathrm{ScoreOA}$ für die Testphase ins Verhältnis zum Gesamtfehler über alle Testdaten $\mathrm{ScoreMA}$ gesetzt. Zudem kann anhand des Vergleichs des Gesamtfehlers der letzten Epoche und des Gesamtfehlers der Testdaten, eingeschätzt werden ob Overfitting auftritt, wenn mit unbekannten Zeitreihen getestet wird.

$$RelScore = \frac{ScoreOA}{ScoreMA} \frac{\sum_{j \in test} ScoreOA_j}{\sum_{k \in test} ScoreMA_k}$$

Dieser relative Fehlerwert soll möglichst klein sein.

1. Testphase: Optimierung von Fenstergröße und Schrittzahl

Die Zeitreihen sollen während Trainings- und Testphase in gleich großen Teilen, anhand der festgelegten Fenstergröße, verarbeitet werden. Während der Trainingsphase wird das Fenster für jede Teilsequenz entsprechend der angegebenen Schrittzahl weitergeschoben.
Da die Zahl der Eingabeparameter der Größe des Fensters entspricht und nicht verändert werden kann, muss die Größe des Fensters ein Teiler der Gesamtlänge der Testzeitreihen sein, um die Testzeitreihen vollständig einlesen zu können. Da alle Zeitreihen von Datensatz A2 1421 Datenpunkte enthalten, kann die Fenstergröße den Werten 1, 7, 29, 49, 203 oder 1421 entsprechen. Um keine Datenpunkte während des Trainings zu überspringen, sollte die Schrittzahl kleiner gewählt werden, als die Fenstergröße.

Ziel der Testphase:
In dieser Testphase sollen Erkenntnisse darüber gewonnen werden, wie die Größe des Fensters gewählt werden soll und wie die Schrittzahl bestimmt werden sollte.

Verwendete Datensätze:

In dieser ersten Testreihe wird zunächst nur die Zeitreihe mit ID = 0 für Training und Testphase verwendet. Da aus der Zeitreihe viele kleinere Datensätze generiert werden, erhält das Netz trotzdem mehrere Datensätze abhängig von der gewählten Fenstergröße.

Gegebene Parameter:

Anzahl der Knoten pro Schichten [windowSize | 50 | 20 | 50 | windowSize]
Lernrate = 0.08
Epochen = 100
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion Ausgabeschicht = MSE

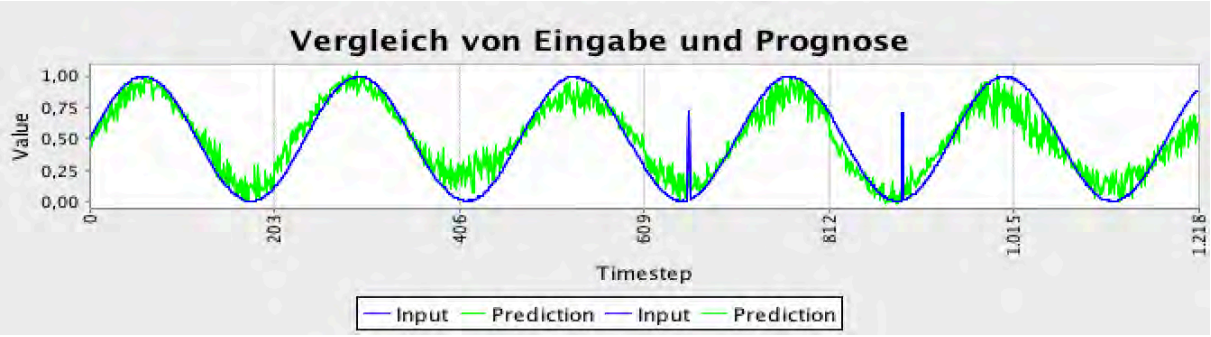
Fenster größe	***Schritt zahl***	***ScoreOA***	***ScoreMA***	***RelScore***	***Verbesserung während Training***
29	1	663.13	664.96	0.9972	Kaum
49	1	551.18	553.67	0.9955	Kaum
203	1	512.55	515.06	0.9951	Kaum
29	10	393.34	394.41	0.9972	Ja
49	10	328.14	329.64	0.9954	Ja
203	10	277.98	279.26	0.9953	Ja
29	20	250.92	251.54	0.9975	Ja
49	20	223.95	224.80	0.9963	Ja
203	20	179.65	180.25	0.9966	Ja
29	29	197.33	197.74	0.9979	Ja
49	49	155.91	156.19	0.9982	Ja
203	**203**	**19.23**	**20.64**	**0.9314**	**Ja**

Fazit:

Je größer das Fenster und je größer die Schrittzahl, desto besser wird das Ergebnis. Die beste Variation ist eindeutig die größte Fenstergröße und eine Schrittzahl von 203.

Dies ist der einzige Fall, wo die Schätzung der Eingabe wirklich ähnlich ist. Dennoch soll im zweiten Teil der Testphase geschaut werden, wie dies aussieht, wenn mit mehreren Zeitreihen trainiert wird.

Die folgende Abbildung zeigt den Fall mit Fenstergröße = 203 und Schrittzahl = 203, einer trainierten Zeitreihe nach 100 Epochen.



Änderungen:

Verwendete Datensätze:

Trainingszeitreihen 0 -24, Testzeitreihen 0-24

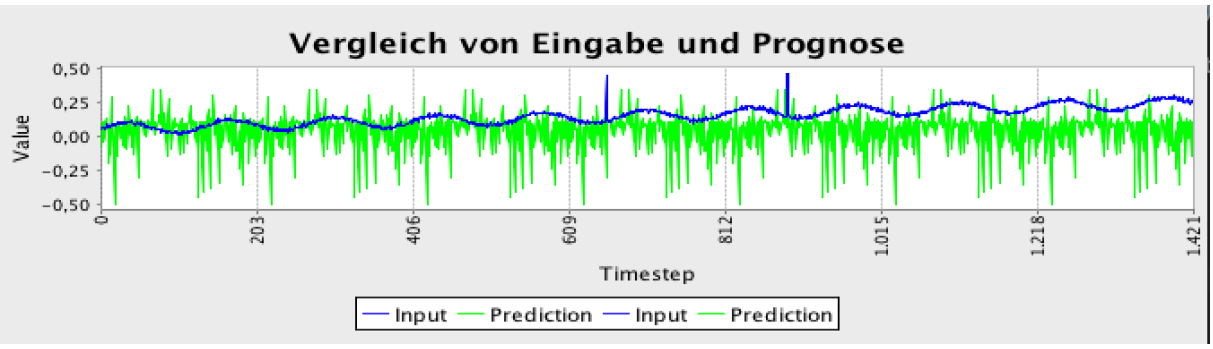
Fenster größe	***Schritt zahl***	***ScoreOA***	***ScoreMA***	***RelScore***	***Verbesserung während Training***
29	1	5625.71	5651.52	0.9954	Nein
49	1	5135.22	5164.923	0.9942	Nein
203	1	2895.6929	2912.84	0.9941	Nein
29	10	5363.21	5388.27	0.9953	Kaum
49	10	4902.34	4930.99	0.9942	Kaum
203	10	2705.27	2721.75	0.9939	Kaum
29	20	5084.88	5109.09	0.9953	Leicht
49	20	4657.88	4685.41	0.9941	Leicht
203	20	2510.50	2526.31	0.9937	Leicht
29	29	4856.92	4880.41	0.9952	Leicht
49	49	4053.26	4077.97	0.9939	Ja
203	**203**	**889.50**	**889.64**	**0.9887**	**Ja**

Fazit:

Auch mit mehr Trainingsdaten liefert das größte Fenster die besten Ergebnisse.

Das folgende Diagramm zeigt die Eingabe und die Schätzung für eine Zeitreihe der letzten Testreihe mit Fenstergröße = Schrittgröße = 203. Die Rekonstruktion verschlechtert sich gegenüber dem Training mit nur einem Datensatz deutlich.

Gegenüberstellung von Schätzung (grün) und Eingabe (blau) für den Fall Fenstergröße = 203 und Schrittgröße = 203.



Änderungen:

Verwendete Datensätze:
Trainingszeitreihen 0 - 99, Testzeitreihen 0 – 99

Ziel:

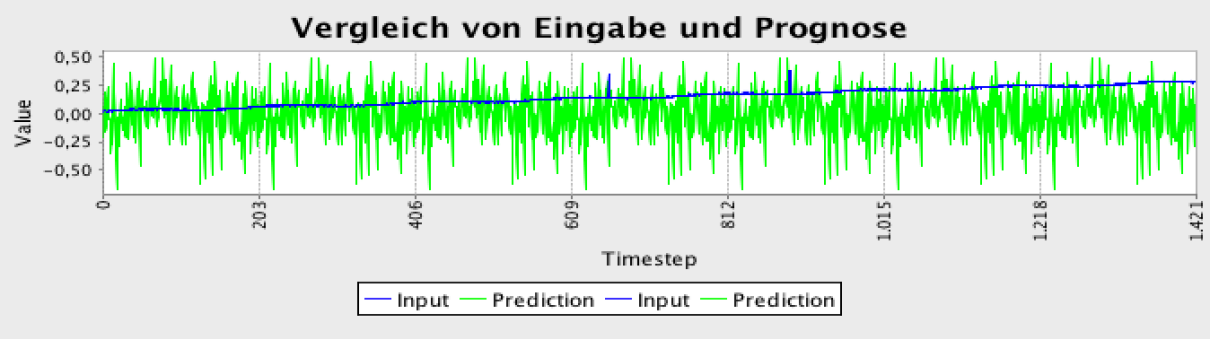
Um zu sehen, ob sich das Ergebnis mit einer kleineren Schrittzahl noch verbessern lässt wird die Fenstergröße auf 203 gesetzt und die Schrittzahl, diesmal mit allen Datensätzen, variiert.

Fenster größe	***Schritt zahl***	***ScoreOA***	***ScoreMA***	***RelScore***	***Verbesserung während Training***
203	25	9808.14	9869.36	0.9937	Kaum
203	50	9345.75	9405.31	0.9937	Kaum
203	75	8924.63	8982.73	0.9935	Leicht
203	100	8539.75	8596.50	0.9934	Leicht
203	125	8075.97	8131.18	0.9932	Ja
203	150	7862.40	7916.78	0.9931	Ja
203	**175**	**7288.78**	**7340.78**	**0.9928**	**Ja**
203	**200**	**7286.88**	**7339.23**	**0.9929**	**Ja**
203	**203**	**7286.79**	**7339.09**	**0.9929**	**Ja**

Fazit:

Die Ergebnisse für 175, 200 und 203 liegen so dicht beieinander, dass sie nicht sinnvoll verglichen werden können. Der Fehleranteil wird in Richtung 175 leicht kleiner, dafür steigt aber auch die Anzahl der Trainingssequenzen in diese Richtung. Die Parameter Fenstergröße und Schrittgröße werden auf 203 festgelegt, da hier weniger Datensätze zum Training verwendet wurden.

Das folgende Diagramm zeigt Schätzung (grün) und Eingabe (blau) für den Fall: Fenstergröße und Schrittgröße gleich 203 nach 100 Epochen mit 100 Datensätzen.



Änderung:

Fenstergröße = 203
Schrittgröße = 203

2. Testphase: Optimierung von Lernrate und Epochenzahl

Die erste Testphase hat gezeigt, dass die Epochenanzahl vergrößert werden kann, weil nach 100 Epochen immer noch ein deutlicher Lernfortschritt erreicht wird. Um die Epochenanzahl möglichst optimal zu gestalten, wird das Programm zusätzlich beendet, wenn das Programm in einem Schritt den Gesamtfehler nicht mehr verkleinert. In dieser Testphase wird die maximale Epochenzahl zunächst auf 2500 festgelegt.

Ziel der Testphase:

In der zweiten Testphase soll das Modell mit drei versteckten Schichten über einen längeren Zeitraum trainiert werden, um zu sehen, ob das Modell überhaupt eine Rekonstruktion der Eingabedaten erzeugen kann.

Verwendete Datensätze:

Trainingsdaten: 1-10
Testdaten: 1-10

Gegebene Parameter:

Anzahl der Knoten pro Schichten [203 | 50 | 20 | 50 | 203]
Epochen = 100
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion Ausgabeschicht = MSE
Fenstergröße = 203
Schrittgröße = 203

$Lernrate$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
0.01	2500	883.34	896.33	885.84	0.9855
0.02	2500	535.73	546.24	537.37	0.9807
0.05	2500	419.58	428.69	421.23	0.9787
0.1	2500	219.79	226.62	220.43	0.9695
0.2	**2500**	**118.24**	**125.43**	**118.07**	**0.9426**
0.5	2500	313.74	322.37	313.05	0.9732

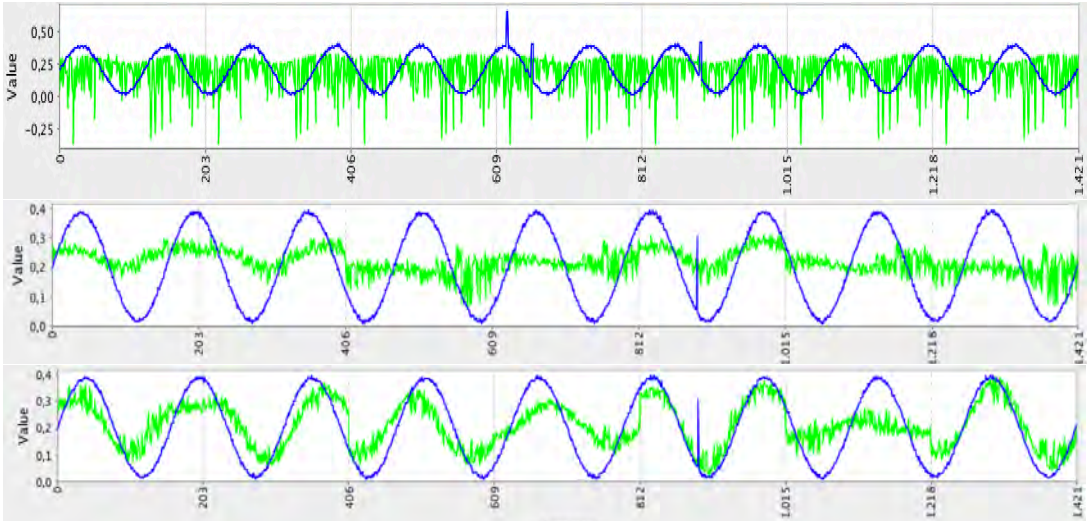
Fazit:

Eine Lernrate von 0.2 ist sehr hoch und führt zu einer deutlichen schnelleren Anpassung, wie die folgende Abbildung zeigt. Auffällig ist, dass bei einer kleinen Lernrate die Fenster alle eine sehr ähnliche Ausgabe zeigen. Bei 0.1 lassen sich dagegen deutliche Unterschiede in den einzelnen Fenstern erkennen, aber keine Rekonstruktion der Daten. Bei einer Lernrate von 0.2 ist dagegen schon deutlich eine Annäherung zu erkennen. Deshalb wird vorerst mit einer Lernrate von 0.2 weitergetestet wird.

Lernrate = 0.01

Lernrate = 0.1

Lernrate = 0.2



Änderungen:

Lernrate = 0.2

Bester Fall mit bekanntem Testdatensatz ohne Beschränkung der Epochen

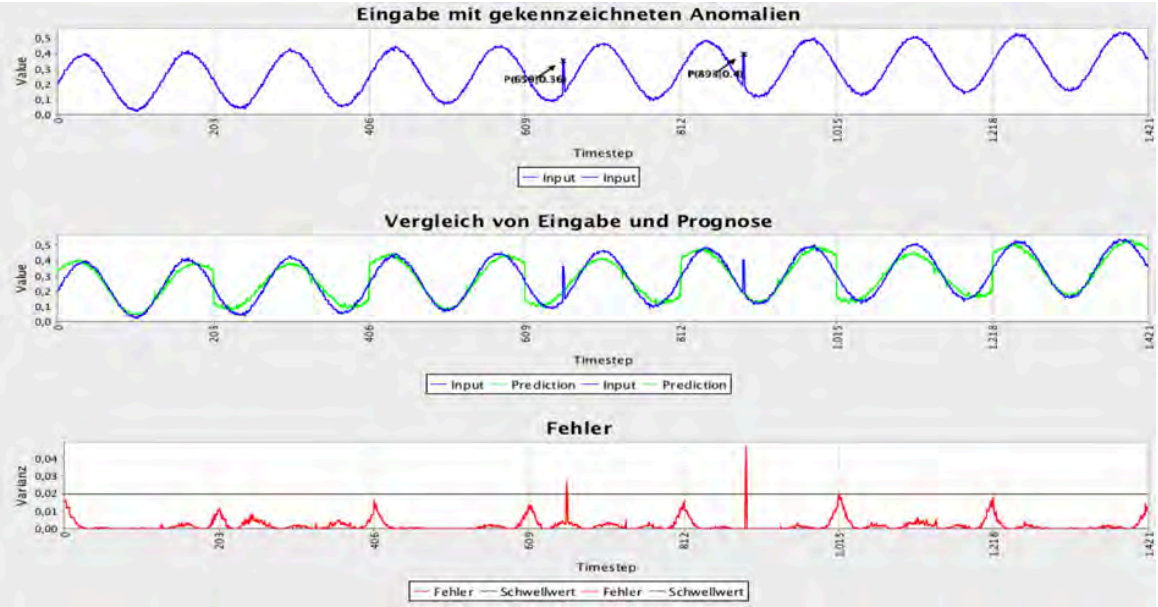
Ziel:

Bei einer Epochenanzahl von 2500 zeigt das Modell immer noch Lernfortschritte. Da diese aber im Laufe der Zeit immer kleiner werden, wird eine zusätzliche Beschränkung eingeführt, die das Programm beendet, wenn der Gesamtfehler während des Trainings kleiner als 50 wird.

$Lernrate$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
0.2	5896	50.13	57.87	49.99	0.8662

Fazit:

Die Rekonstruktion ist bis auf einige Stellen, vor allem am Ende und Anfang jedes Fensters, sehr genau. Beide Anomalien werden eindeutig erkannt.



Änderungen:

Testdaten: 11 – 20

Bester Fall mit unbekanntem Testdatensatz

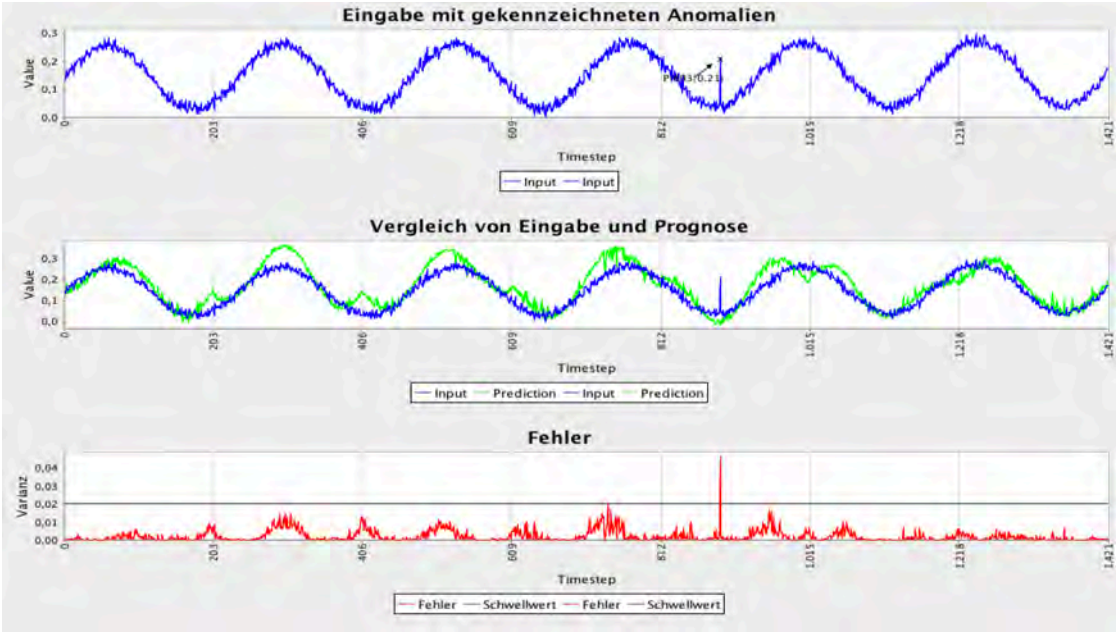
Ziel:

Da bisher nur Zeitreihen getestet wurden, die bereits im Training verwendet wurden, soll das Modell nun an unbekannten Zeitreihen getestet, um zu sehen ob das Netz die Fähigkeit der Generalisierung besitzt.

$Lernrate$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
0.2	5896	78.28	84.57	49.99	0.9256

Fazit:

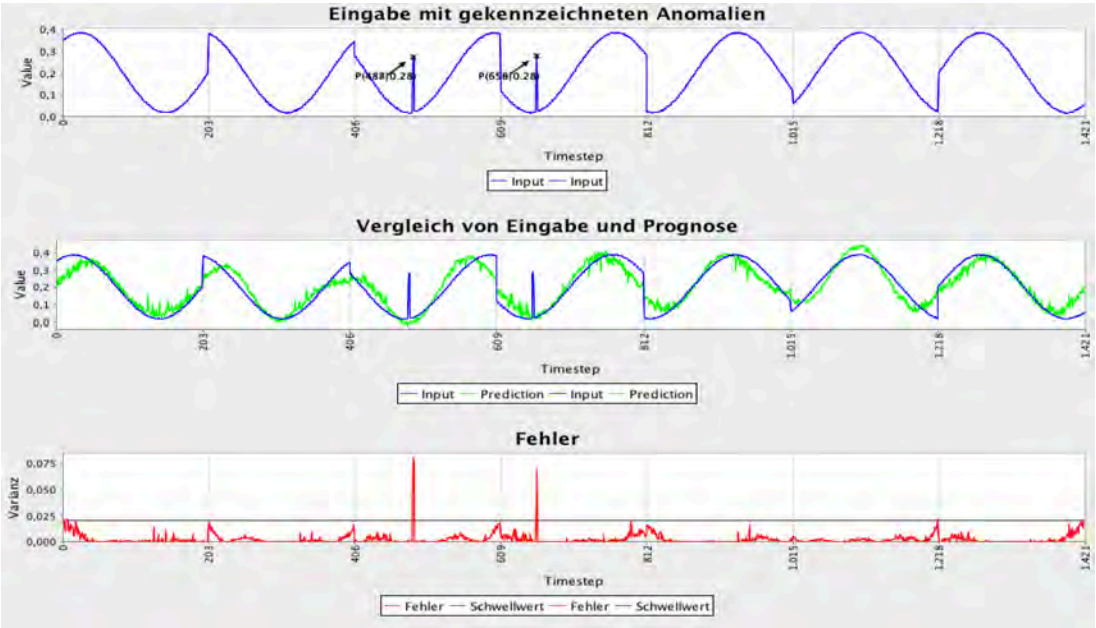
Auch die unbekannten Zeitreihen können rekonstruiert werden, jedoch etwas schlechter als die bekannten Daten. Die Anomalie kann aber eindeutig lokalisiert werden.



Bester Fall mit rückwärts eingelesenem Testdatensatz

Um zu zeigen, dass keine Zusammenhänge zwischen den einzelnen Fenstern bestehen, werden in diesem Testfall die Fenster der Testdaten in umgekehrter Reihenfolge prognostiziert.

$Lernrate$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
0.2	5896	50.13	57.87	49.99	0.8670



Fazit:

Die Rekonstruktion ist genauso gut, wie in richtiger Reihenfolge. Der Autoencoder mit drei versteckten Schichten konnte erfolgreich trainiert werden. Nun soll geschaut werden, ob dies auch mit mehr Schichten möglich ist.

3. Testphase: Variation der versteckten Schichten

In der dritten Testphase soll das Modell mit vielen versteckten Schichten getestet werden, um zu sehen ob sich mit DL4J ein Deep Autoencoder umsetzen lässt. Falls kein Modell mit vielen Schichten trainierbar ist, bedeutet dies jedoch nicht, dass es DL4J nicht möglich ist Deep Autoencoder zu modellieren, da nicht alle Parameterkombinationen getestet werden können.

Ziel der Testphase:

Es soll getestet werden, ob sich auch Autoencoder mit vielen versteckten Schichten auf die gleiche Weise umsetzen lassen, wie das Ergebnis aus Testphase 2.

Verwendete Datensätze:

Trainingsdaten: 1-10
Testdaten: 1-10

Gegebene Parameter:

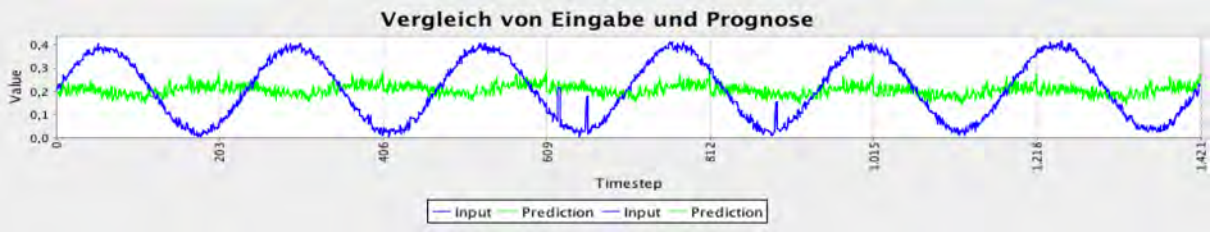
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion Ausgabeschicht = MSE
Fenstergröße = 203
Schrittgröße = 203
Lernrate = 0.2

$Aufbau\ Encoder$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
[203, 20	7693	50.32	57.32	49.99	0.8764
[203, 50, 20	**5896**	**50.13**	**57.87**	**49.99**	**0.8662**
[203, 100, 50, 20	72	485.66	495.12	487.20	0.9809
[203, 150, 100, 50, 20	279	243.90	250.76	242.38	0.9726
[203, 180, 140, 100, 50, 20	219	485.91	494.91	486.99	0.9809

Fazit:

Auch mit einer versteckten Schicht ist der Autoencoder in der Lage eine Rekonstruktion zu erzeugen, die nur leicht schlechter ist als mit drei versteckten Schichten. Auf mehr Schichten lässt sich das Modell aber nicht übertragen.

Wie die folgende Abbildung beispielhaft für das Modell mit den Schichten [203, 150, 100, 50, 20, 50, 100, 150, 203] zeigt, erzeugt der Deep Autoencoder in jedem Fenster eine ähnliche Ausgabe, sodass wahrscheinlich ein Mittelwert der Daten berechnet wurde.



Anhand einer Variation der Lernrate soll geprüft werden, ob sich dieses Ergebnis verbessern lässt.

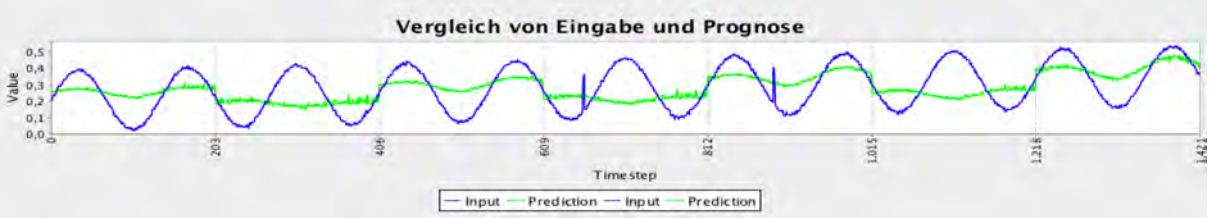
Änderungen

Variation der Lernrate
Schichten: [203, 150, 100, 50, 20]

$Aufbau\ Encoder$	$Lernrate$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
[203, 150, 100, 50, 20	0.2	279	243.90	250.76	242.38	0.9726
[203, 150, 100, 50, 20	0.15	552	230.74	237.45	231.03	0.9718
[203, 150, 100, 50, 20	**0.1**	**1333**	**223.67**	**230.32**	**224.37**	**0.9711**
[203, 150, 100, 50, 20	**0.08**	**2236**	**220.80**	**227.37**	**221.46**	**0.9711**
[203, 150, 100, 50, 20	0.05	431	485.45	494.91	486.98	0.9809

Fazit:

Das Ergebnis lässt sich zwar noch leicht verbessern, aber die Zeitreihen werden nicht rekonstruiert, wie die folgende Abbildung verdeutlicht. Die Fenster zeigen immer noch ein ähnliches Bild, allerdings wird durch vertikale Verschiebung eine Verbesserung des Fehlers erreicht.



Auffällig ist, dass sich die Fehlerabweichung erst stark verringert und sich dann wieder fängt.

Dies geschieht in diesem Fall in dem Zeitraum, wenn der Fehler etwa bei 480 liegt. Einige Zeitreihen terminieren an dieser Stelle, andere vergrößern ihre Anpassungen nach dieser Phase wieder schrittweise, sodass sie den Gesamtfehler unter 250 drücken können. Das Ergebnis ist aber in beiden Fällen ein Durchschnittswert.

Da bei DL4J, die Knoten in der ersten und letzten versteckten Schicht höher als die Anzahl der Eingabewerte setzt, wird dieser Fall ebenfalls getestet.

Änderungen:

Variation der Lernrate
Schichten: [203, 300, 150, 100, 50, 20]

$Aufbau\ Encoder$	$Lernrate$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
[203, 300, 150, 100, 50, 20	0.2	21	485.43	494.89	486.96	0.9809
[203, 300, 150, 100, 50, 20	0.15	28	485.43	494.89	486.97	0.9809
[203, 300, 150, 100, 50, 20	0.1	50	485.44	494.90	486.97	0.9809
[203, 300, 150, 100, 50, 20	0.08	145	485.45	494.91	486.98	0.9808
[203, 300, 150, 100, 50, 20	0.05	145	485.45	494.91	486.98	0.9809

Fazit:

Das Netz lässt sich auch auf diese Weise nicht trainieren. Beim Training mit vielen Schichten kommt es häufig vor, dass die Änderung des Gesamtfehlers nur für eine Epoche negativ wird und das Netz deshalb nur sehr kurz trainiert. Eine Änderung der Abbruchbedingungen, die festlegt, dass der Gesamtfehler zwei Epochen hintereinander größer wird, statt kleiner, sollte dem entgegenwirken.

Es stellte sich jedoch heraus, dass auch häufig Phasen vorkommen, wo der Gesamtfehler im Wechsel größer und wieder kleiner wird, sodass das Netz zwar lange

trainiert und auch leichte Verbesserungen erreicht, aber immer noch einen Durchschnittswert berechnet.
Es wurden zahlreiche weitere Kombinationen der Parameter getestet. Es konnte aber nicht erreicht werden, dass das Modell eine erkennbare Rekonstruktion anfertigt.

Training ohne Aufteilung der Zeitreihen

Zum Abschluss dieser Phase soll das Modell ohne die Aufteilung in kleinere Zeitreihen getestet werden, um zu sehen, ob das Konzept mit geteilten Zeitreihen überhaupt Vorteile bringt.
Zunächst werden die Modelle getestet, die im Modell mit Zerlegung der Zeitreihen gute Rekonstruktionen erzeugt haben.
Der Aufbau der Schichten und die Lernrate wurden aufgrund der Änderung angepasst und an einigen Beispielen vorgetestet, die hier aber nicht weiter dokumentiert werden sollen. Es wurde mit allen 100 Zeitreihen trainiert.

$Aufbau\ Encoder$	$Lernrate$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
[1421, 50	0.1	2461	114.47	176.03	115.34	0.6502
[1421, 500, 50	0.1	2461	114.47	176.03	115.33	0.6503

Fazit:
Die Zerlegung der Zeitreihe führt zu einer Verbesserung des Trainings des Modells, da bessere Rekonstruktionen erzeugt werden, als wenn die Zeitreihe am Stück trainiert wird. Zwar ist das Verhältnis des Gesamtfehlers ohne Anomalien zum Gesamtfehler mit Anomalien wirklich klein. Jedoch bekommt die Schätzung nur den ungefähren Verlauf der Zeitreihe mit, aber keine Schwankungen mit, wie die folgende Abbildung zeigt.

[1421, 50]

[1421, 500, 50]

Mit wenigen versteckten Schichten bringt das Modell ganz gute Ergebnisse hervor, wenn auch keine richtige Annäherung. Für die abschließende Testphase wird deshalb wieder auf das Modell mit Fenstern zurückgegriffen.

4. Testphase: Test der funktionierenden Autoencoder mit mehr Trainingsdaten aus Datensatz A2, A3 und A4

Es wurden zahlreiche weitere Kombinationen ausprobiert, aber keine Verbesserung der Ergebnisse mit vielen versteckten Schichten erreicht. Aus diesem Grund wird zum Abschluss der Autoencoder mit drei versteckten Schichten mit mehr Trainingsdaten getestet, da dieses Modell die besten Resultate erzielt hat.

Ziel:

Zum Abschluss wird ein Test mit den Datensätze A2, A3 und A4 durchgeführt, um zu sehen, ob der Autoencoder auch in der Lage, ist Anomalien in komplexeren Zeitreihen zu lokalisieren und mehr Datensätze zu verarbeiten.

Gegebene Parameter:

Anzahl der Knoten pro Schichten [203 | 50 | 20 | 50 | 203]
Initialisierungsmethode: XAVIER_LEGACY
Update-Methode: ADAGRAD
Optimierungsalgorithmus = STOCHASTIC_GRADIENT_DESCENT
Aktivitätsfunktion der versteckten Schichten = TANH
Aktivitätsfunktion der Ausgabeschicht = IDENTITY
Verlustfunktion Ausgabeschicht = MSE
Lernrate = 0.2

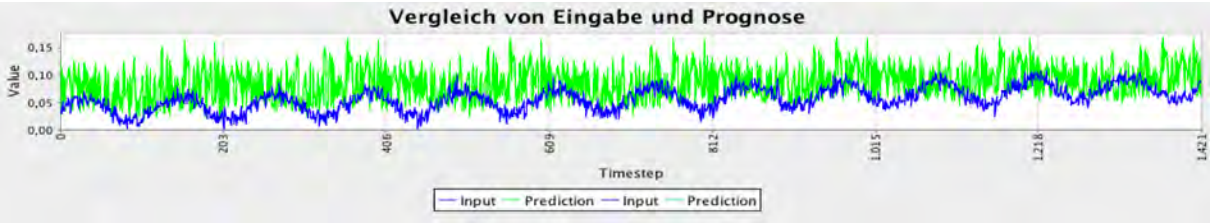
Datensatz A2 – 50 Trainingszeitreihen nach 10 000 Epochen

Fenstergröße =Schrittgröße = 203

$Trainings\ daten$	$Test\ daten$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
0-49	50-99	10000	390.58	442.87	279.60	0.8819

Fazit:

Das Training wird nach 10000 Epochen beendet, da dies als obere Grenze festgelegt wird. Die folgende Abbildung zeigt, dass zu diesem Zeitpunkt noch keine gute Annäherung gefunden werden konnte.



Exkurs Laufzeit

Aus Gründen der Übersichtlichkeit wurde die Laufzeit des Programms bisher vernachlässigt. An dieser Stelle wird sie jedoch relevant. Da die Laufzeit je nach Aufbau und Knotenanzahl des Modells variiert, wird sich hier nur auf den aktuellen Fall mit 3 Schichten bezogen.

Nach diesem Trainingsfall über 10 000 wurde die Epochenzahl erneut erhört, um das Programm erst dann terminieren zu lassen, wenn eine Rekonstruktion zu erkennen ist. Nach mehr als zwei Stunden und über 60 000 Epochen, wurde das Programm mit einem Fehler von etwa 70 gestoppt.

Für das Training wird die Zeitreihe bei einer Fenstergröße von 20 in sieben kleinere Zeitreihen geteilt. In jeder Epoche werden in diesem Fall also 140 Zeitreihen trainiert. Für 100 Epochen mit 20 Datensätzen braucht das Programm 20 Sekunden und für 200 Epochen etwa 40 Sekunden.

Für 50 Datensätze benötigt auf 100 Epochen 22 Sekunden und für 100 Datensätze 25 etwa 25 Sekunden. Der Gesamtfehler ist für mehr Datensätze aber deutlich höher.

Es müssten für das Training mit einer unterschiedlichen Menge an Daten, auch unterschiedliche Abbruchbedingungen angewendet werden, um die Laufzeit auf ein zumutbares Maß zu reduzieren und trotzdem eine gute Rekonstruktion zu erhalten. Da aus Zeitgründen leider auf diese Erweiterung verzichtet werden muss, wird nur mit 20 Datensätze gearbeitet.

Datensatz A2

Änderungen:

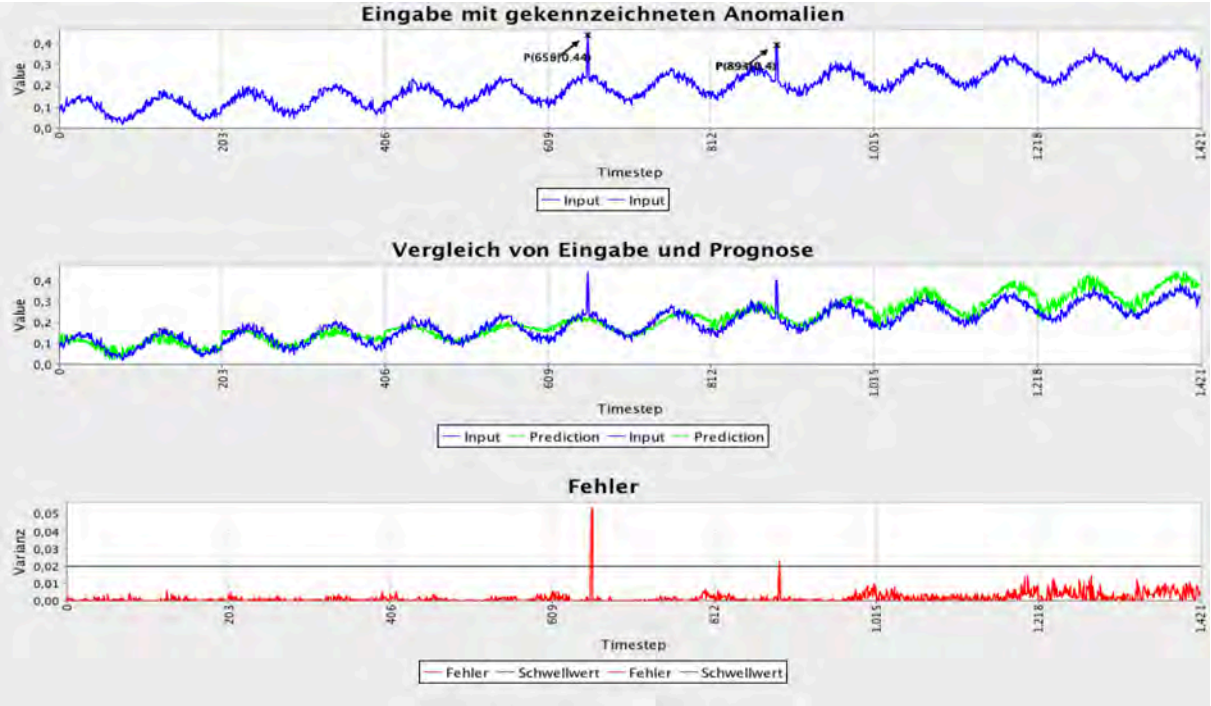
Trainingsdaten 1-20

Abbruch bei einem Gesamtfehler von 50 in einer Epoche

$\boldsymbol{Trainings\ daten}$	$\boldsymbol{Test\ daten}$	$\boldsymbol{Epochen}$	$\boldsymbol{ScoreOA}$	$\boldsymbol{ScoreMA}$	$\boldsymbol{Score_{lastEpoch}}$	$\boldsymbol{RelScore}$
0-19	20-39	27175	69.31	91.81	49.99	0.7544

Fazit:

Die Anomalien werden auch bei einer Begrenzung des Gesamtfehlers auf 50 deutlich erkannt, wie die folgende Abbildung verdeutlicht. Die Epochenanzahl ist aber mit 27175 sehr hoch.



Datensatz A3

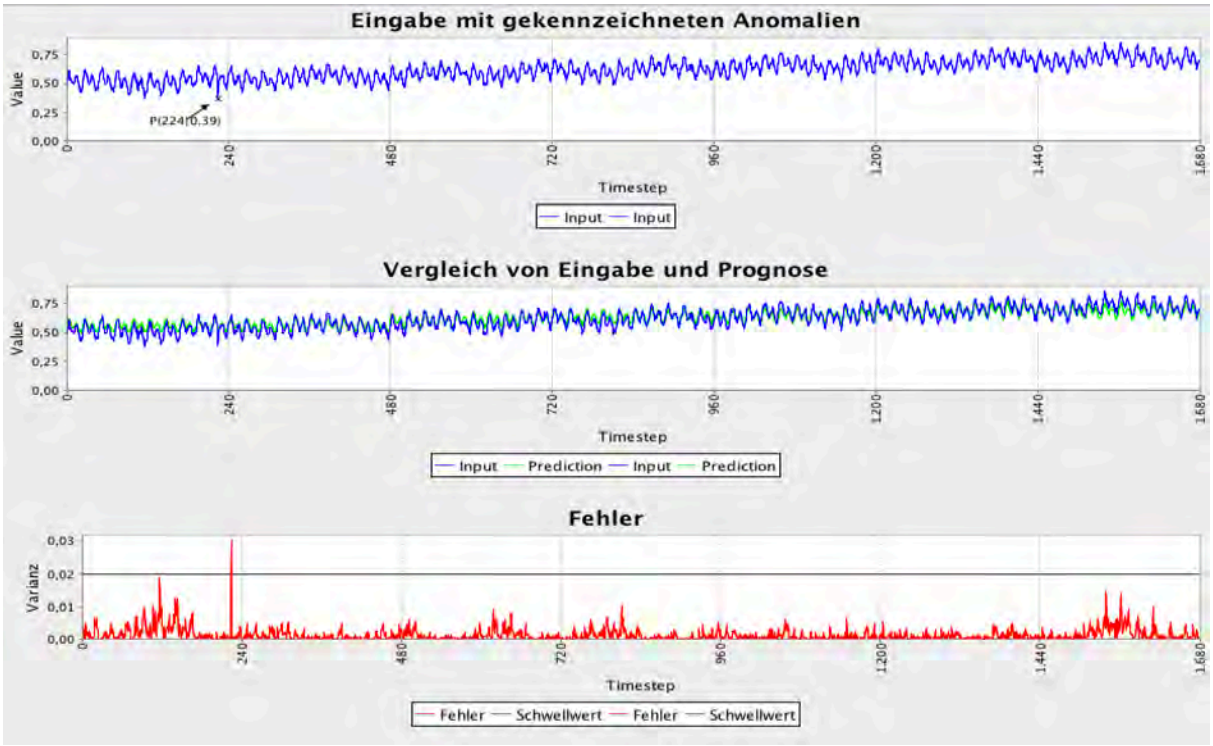
Änderungen:

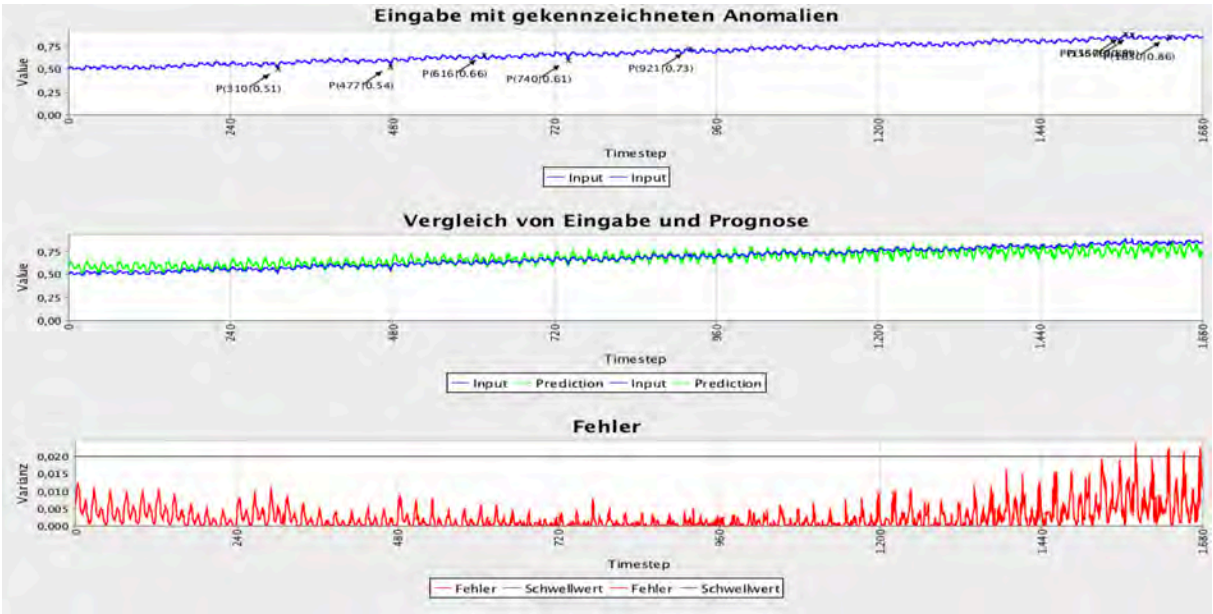
Fenstergröße = Schrittgröße = 240

$Trainings\ daten$	$Test\ daten$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
0-19	20-39	5363	44.02	45.08	49.99	0.9764

Fazit:

Die folgenden Abbildungen zeigen zwei Beispiele für diesen Testfall. Im oberen Fall wird die Anomalie erkannt. Im zweiten Fall sind die Anomalien schwieriger zu identifizieren, da sie nur schwach ausgeprägt sind. Hier wird nur ein Teil der Anomalien lokalisiert. Auffällig ist hier die Epochenanzahl gegenüber Datensatz A2 gesunken ist. Ursache hierfür könnte sein, dass die Zeitreihen in A2 über größeren Schwankungen verfügen und deshalb nicht so schnell angenähert werden können wie Zeitreihen mit flacherem Verlauf.





Datensatz A4

Fenstergröße = Schrittgröße = 240

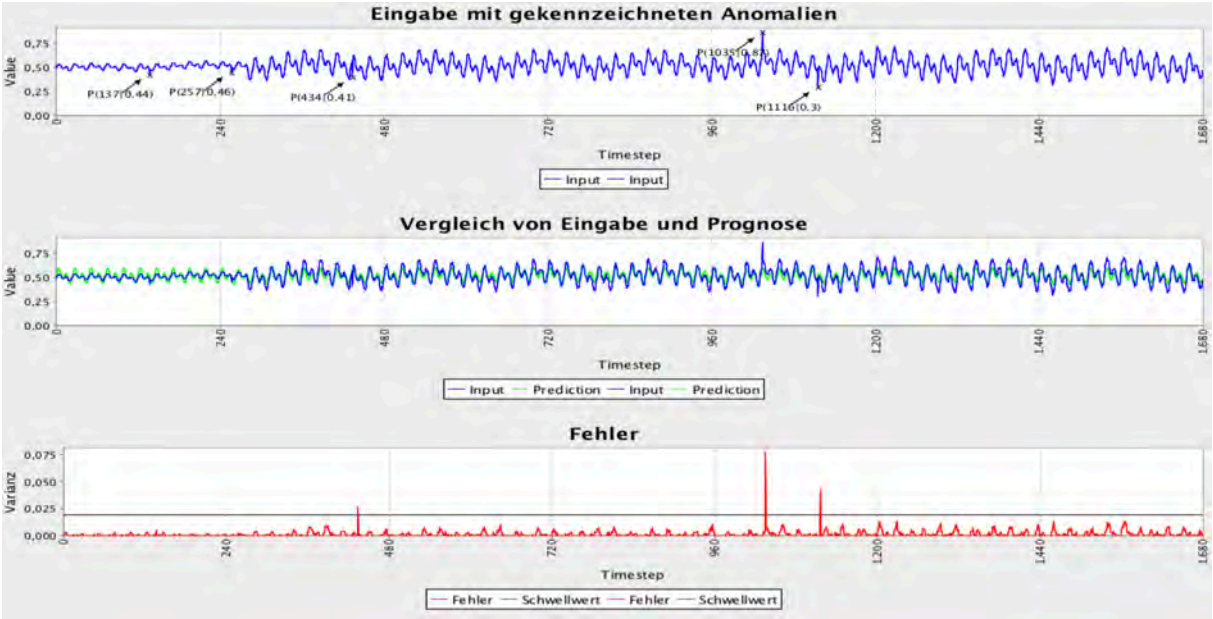
Änderungen:

Datensatz A4

$Trainings\ daten$	$Test\ daten$	$Epochen$	$ScoreOA$	$ScoreMA$	$Score_{lastEpoch}$	$RelScore$
40-59	60-61	8043	51.44	55.87	49.99	0.9206

Fazit:

Die Ergebnisse zu Datensatz A4 sehen ähnlich aus wie zu Datensatz A3. Die Wechsel der Eigenschaften der Funktionen werden mittrainiert, aber die Ausprägung der Anomalien ist zu klein, um alle zu lokalisieren wie die folgende Abbildung zeigt.



Eine Verlängerung der Trainingszeit, durch Herabsenken der Abbruchbedingung auf unter 50, könnte eine noch bessere Annäherung zeigen. Da dies aber die Trainingszeit nochmal deutlich verlängern würde und dies im Rahmen dieses Projekts nicht mehr umsetzbar ist, wird auf eine weitere Optimierung der Ergebnisse verzichtet.

Eidesstattliche Erklärung

„Ich versichere an Eides Statt, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen“.

_____

(Datum, Ort, Unterschrift)