INSTITUT FÜR INFORMATIK
Lehrstuhl für algorithmische
Bioinformatik

Universitätsstr. 1 D–40225 Düsseldorf

Vorhersage von Methicillin-resistenten Staphylococcus-aureus-Typen mittels Deep Learning

Andre Walter

Bachelorarbeit

Beginn der Arbeit:	01. Mai 2019
Abgabe der Arbeit:	01. August 2019
Gutachter:	Prof. Dr. Gunnar Klau
	Prof. Dr. Stefan Harmeling

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 01. August 2019

A. Walter

Andre Walter

Zusammenfassung

Durch immer leistungsfähigere Prozessoren erlebt die Disziplin der Neuroinformatik, insbesondere der Teilbereich des maschinellen Lernens eine Renaissance. Da es nun möglich ist, die stetig ansteigende Masse an Informationen mit künstlicher Intelligenz zu analysieren, entwickelt die Wissenschaft immer effektivere Konzepte, die in beinahe allen Bereichen der Informatik angewandt werden können.
Diese Bachelorarbeit nutzt den Fortschritt der Forschung, um eine Klassifizierung von Staphylococcus-aureus-Genomen mit einem künstlichen neuronalen Netz vorzunehmen. Das Ergebnis ist vor allem für medizinische Einrichtungen interessant, da ihnen ermöglicht wird, basierend auf den sequenzierten Reads des Staphylococcus-aureus-Genoms der Patienten, Verbreitungswege des Keims nachzuvollziehen. Als Datengrundlage dient der Ridom SpaServer, der alle Typen des Staphylococcus aureus' mit der entsprechenden Genomsequenz und ihrer Frequenz auflistet. Die typspezifischen Sequenzen werden als Referenzgenom für den Sequenzierungssimulator wgsim genutzt und so Trainingsdaten generiert. Es wurden diverse künstliche neuronale Netze mit der Python Bibliothek pytorch implementiert und auf die 200 häufigsten Typen trainiert. Das effektivste Netz erreicht eine Präzision von 94.6% und eine Top-5-Präzision von 99.9%. Dabei wird eine durchschnittliche Konfidenz von 92.9% in den korrekten Typen angegeben. Der Quellcode ist online unter dem DOI `10.5281/zenodo.3354558` archiviert.
Sämtliche Berechnungen für diese Arbeit wurden auf dem High-Performance Server des Zentrums für Informations- und Medientechnologie (ZIM) der Universität Düsseldorf (Deutschland) durchgeführt.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Hintergrund 1
1.2 Zielsetzung 1

2 Maschinelles Lernen **2**
2.1 Künstliche neuronale Netze 2
2.2 Funktionsweise 3
2.2.1 Ausgabe berechnen 3
2.2.2 Bewertung eines Durchlaufs 5
2.2.3 Anpassung der Gewichte 6

3 Komponenten **9**
3.1 Workflow-Management-System 9
3.2 Simulierung der Sequenzierung 9
3.3 Bibliothek für maschinelles Lernen 10

4 Implementierung **10**
4.1 Arbeitsschritte 10
4.2 Architektur des künstlichen neuronalen Netzes 11
4.3 Training 12

5 Evaluation **15**
5.1 Wichtige Hyperparameter 15
5.2 Effektive Architekturen 19

6 Diskussion **21**
6.1 Ergebnisse 21
6.2 Alternative Ansätze 22

Literatur **23**

1 Einleitung

1.1 Hintergrund

In der letzten Zeit erregen Krankenhauskeime, besonders aus klinischer Sicht, viel Aufmerksamkeit. Durch die schwierig zu behandelnde Krankheit und seine Resistenzen gegenüber Antibiotika, ist der Methicillin-resistente Staphylococcus aureus (MRSA) eine außerordentliche Bedrohung für immungeschwächte Patienten. Damit der Patient fachgerecht behandelt werden kann und die Verbreitungswege des Keims nachvollziehbar sind, ist es essentiell, den MRSA-Typen klassifizieren zu können [1].
Ein Bereich der DNA, anhand der der vorliegende Typ bestimmt werden kann, ist die hypervariante Region X auf dem spa-Gen, welches für das Protein A codiert. Mit der Sequenzierung dieses Abschnitts werden Informationen gesammelt, die zur Klassifizierung des MRSA-Typen führen sollen. Problematisch sind Punktmutationen im Genom des Bakteriums, sowie die Fehler bei der Sequenzierung und des Alignments, die eine klare Zuordnung zu dem vorliegenden Typen erschweren [2]. Der Ridom SpaServer [3] ist die Datengrundlage dieser Arbeit, da dieser rund 18.000 Typen mit entsprechenden Sequenzen zur Verfügung stellt, deren jeweiliger Anteil an allen bisher klassifizierten Typen angegeben ist. Die Ridom GmbH vertreibt außerdem eine Software, mit der MRSA-Keime typisiert werden können [4].

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, ein künstliches neuronales Netz (im Folgenden kurz: KNN) zu konstruieren und zu trainieren, das anhand von sequenzierten Reads des Staphylococcus-aureus-Genoms eines Patienten auf den jeweiligen MRSA-Typen schließt. Die Vermutung ist, dass ein KNN hierfür besonders geeignet ist, da sich die Polymorphismen und Sequenzierungsfehler durch die Generalisierung weniger stark auf das Ergebnis der Typisierung auswirken als bei alternativen Ansätzen .
Die Idee dabei ist, eine komplexe, nicht lineare Funktion mittels maschinellen Lernens zu ermitteln, die als Eingabe eine zuvor prozessierte Form einer Sequenz entgegen nimmt und darauf basierend die Wahrscheinlichkeiten für alle Typen ausgibt.
Ist das Netz zu einem ausreichenden Grad der Genauigkeit trainiert worden, können echte sequenzierte Reads, die dem Netz bisher fremd sind, eingegeben werden. Anhand der enthaltenen Informationen trifft das KNN eine Vorhersage über die vermuteten Typen und gibt entsprechende Wahrscheinlichkeiten an. Das Uniklinikum Düsseldorf sequenziert MRSA-Genome mit dem HiSeq System des Illumina Unternehmens und stellt acht von ihnen zu Testzwecken zur Verfügung.
Diese Arbeit soll einen Einblick in die Chancen und Limitierungen künstlicher Intelligenz in diesem Anwendungsbereich geben. Zu diesem Zweck werden zuerst die Grundlagen eines klassifizierenden Netzes dargestellt, einige benötigte Anwendungen beschrieben und zum Schluss eine mögliche Implementierung vorgestellt und ausgewertet.
Das präsentierte Netz ist auf die 200 häufigsten Typen trainiert, hat eine Trefferquote von 94.6%, eine durchschnittliche Konfidenz von 92.9% und gibt den korrekten Typen zu 99.9% unter den fünf wahrscheinlichsten Klassen an. Der Quellcode ist online unter dem DOI `10.5281/zenodo.3354558` zu finden.

2 Maschinelles Lernen

2.1 Künstliche neuronale Netze

Der Aufbau und die Funktionsweise von künstlichen neuronalen Netzen sind dem menschlichen Gehirn nachempfunden. Ein Netz besteht aus Knoten, die Neuronen repräsentieren und gewichteten Kanten, wie sie im Gehirn in Form von Synapsen existieren. Neuronen leiten Reize an benachbarte Neuronen. Wie stark das ausgehende Signal dabei ist, wird durch eine Aktivierungsfunktion kontrolliert. Bei einem KNN kann ein Schwellwert definiert sein, den der eingehende Impuls überschreiten muss, bevor dieser weitergeleitet wird. Einfachheitshalber sind Knoten zu einer Schicht zusammengefasst, dessen Knoten in der Regel nicht durch interne Kanten verbunden sind, sondern nur zu den Knoten der vorangehenden und folgenden Schicht [5].
Der Informationsfluss beginnt bei der Eingabeschicht, die so viele Knoten enthält, wie nötig sind, um die Merkmale der ursprünglichen Eingabe darstellen zu können. Typischerweise werden Bilder vor der Eingabe komprimiert, um die Anzahl der Pixel zu reduzieren. Darauf folgen v verdeckte Schichten, wobei man bei $v \geq 2$ von Deep Learning spricht [6]. Wird keine verdeckte Schicht benutzt, kann das Netz nur lineare Abbildungen repräsentieren, die für viele Problemstellungen nicht ausreichen. Erst mit einer verdeckten Schicht kann ein Netz jede stetige Funktion erlernen.
Neben v gehört die Anzahl der Knoten in den verdeckten Schichten m zu den Hyperparametern, die bei der Konstruktion eines KNN eingestellt werden müssen. Es existieren diverse Faustregeln, die als Orientierung bei der Wahl von m dienen können, wie beispielsweise $m = \sqrt{in \cdot out}$ oder $m = \frac{2}{3} \cdot in + out$, wobei in und out den Anzahlen der Knoten in der Ein- und Ausgabeschicht entsprechen [7]. Letztendlich gibt es jedoch keine Formel, die die optimalen Werte für v oder m angibt, sodass diese Parameter durch Experimente bestimmt werden müssen.
Den Abschluss bildet die Ausgabeschicht, die im Fall eines klassifizierenden Netzes aus c Knoten besteht, wobei c die Anzahl der Klassen ist. Gilt $c = 2$, handelt es sich um eine binäre Klassifikation, bei der, wie in Abbildung 1 dargestellt, ein Ausgabe-Neuron ausreicht.

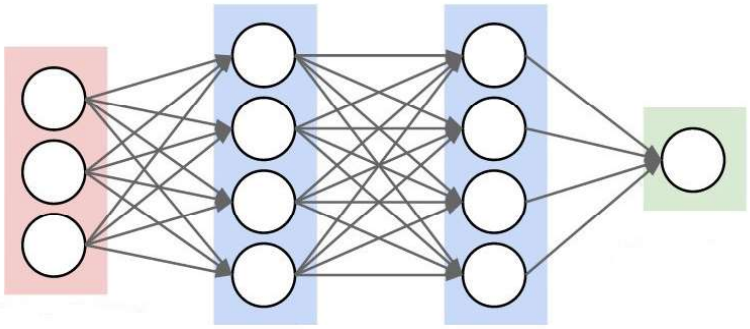

Abbildung 1: Ein klassisches neuronales Netzwerk mit vier Schichten.
Die rote Schicht nimmt die Eingabe entgegen, die zwei blauen Schichten verarbeiten die Informationen und die grüne Schicht dient als Ausgabe.

2.2 Funktionsweise

Dieses Kapitel soll das nötige Vorwissen über das verwendete klassifizierende vorwärts gerichtete Netz vermitteln, und vernachlässigt daher andere Modelle wie rekurrente, rekursive oder faltende Netze. Es wird dabei folgende Notation genutzt:
Sei $c(N)$ die Anzahl der Schichten des Netzes N bestehend aus den Schichten S_i für $i = \{1, 2, ..., c(N)\}$. Es wird für jeden Knoten j zwischen Nettowert x_j und Aktivierungslevel a_j differenziert. Liegt j in der i-ten Schicht, gilt $j \in S_i$. Das Gewicht der Kante von j zu k sei als $w_{k,j}$ bezeichnet.
Es wird zwischen dem Training und der Anwendung eines KNN unterschieden. Das Training erfolgt in drei Phasen, die im Folgenden erklärt werden. Bei der Anwendung trifft das Netz basierend auf dem Gelernten eine Vorhersage für die wahrscheinlichsten Klassen.

2.2.1 Ausgabe berechnen

Der *feed forward*-Prozess bezeichnet den ersten Schritt des Lernvorgangs. Dabei werden die Eingabewerte in die Knoten der Eingabeschicht geschrieben und darauf basierend die Werte der folgenden Neuronen berechnet. Der Nettowert eines Neurons $x_k \in S_{i+1}$ ist gleich der Summe der Aktivierungslevel der Knoten der vorangehenden Schicht S_i, je multipliziert mit dem Gewicht der Kante, die zu dem Knoten k führt. Sämtliche Kanten werden bei der Initialisierung des Netzes typischerweise auf zufällige Werte aus dem Intervall $[-1, 1]$ gesetzt [8]. Formal ausgedrückt gilt:

$$x_k = \sum_{j \in S_i} w_{k,j} \cdot a_j. \tag{1}$$

An dieser Stelle wird in der Regel ein initial zufällig gewählter Schwellenwert addiert. Die Gleichung (1) wird dadurch wie folgt erweitert:

$$x_k = b_k + \sum_{j \in S_i} w_{k,j} \cdot a_j.$$

Um die Komplexität zu reduzieren, wird ein Knoten b_i in jede Schicht S_i außer der Ausgabeschicht hinzugefügt, der dauerhaft den Wert Eins enthält, daher auch on-Neuron genannt wird. Werden die Gewichte der Kanten auf $w_{k,b_i} := b_k$ gesetzt, gilt wieder (1).
Werden die Aktivierunglevel aller Neuronen einer Schicht S_i als ein vertikaler Vektor A_i betrachtet und der horizontale Vektor w_k mit den Gewichten der Kanten zum Knoten k aufgestellt, lässt sich die Formel weiter als Vektormultiplikation vereinfachen:

$$x_k = \langle w_k, A_i \rangle. \tag{2}$$

Auf diesen Wert wird eine Aktivierungsfunktion $\sigma(x)$ wie beispielsweise die $\mathrm{Sigmoid}$-Funktion oder der Tangens hyperbolicus ($\tanh$) angewandt, die den Nettowert x_k auf das Aktivierungslevel a_k abbildet. Diese Funktion ist notwendig, damit die Werte der Neuronen nicht exponentiell groß werden, sondern auf ein von σ abhängiges Intervall gestaucht werden. Im Bereich des Deep Learning hat sich die ReLU-Funktion (von engl. *rectified linear unit*) weitgehend etabliert [9].

Der oben erwähnte Schwellenwert wird an dieser Stelle benötigt, da ohne ihn im Fall einer Eingabe gleich Null durch die Multiplikationen mit den Gewichten kein anderer Nettowert als Null entstehen kann, und dadurch die Ausgaben von σ fixiert wären. Grafisch gesehen, verschiebt der Schwellenwert die Aktivierungsfunktion nach links oder rechts um die gewünschten Aktivierungslevel generieren zu können [10]. Erweitert man die Gleichung (2) um die Aktivierungsfunktion, ergibt sich die vollständige Formel für jeden beliebigen Knoten:

$$a_k = \sigma(x_k) = \sigma(\langle w_k, A_i \rangle).$$

An dieser Stelle kann eine Ausfall-Funktion eingebaut werden, die vor Kurzem von Google patentiert wurde [11]. Das sogenannte *dropout gate* blockiert während des Trainings mit einer anzugebenen Wahrscheinlichkeit die Ausgabe von Neuronen, wodurch das Netz robuster werden kann und eine Überanpassung vermieden wird. Als robust wird ein Netz bezeichnet, dessen Ausgabe durch die Vielzahl der Knoten determiniert wird.
Um die Aktivierungslevel der Neuronen einer ganzen Schicht A_{i+1} zu bestimmen, ist es notwendig, eine Gewichtsmatrix W_i aufzustellen, die in ihrer p-ten Reihe den Gewichtsvektor vom p-ten Neuron der Schicht S_i enthält. Damit lassen sich durch Matrix-Vektor-Multiplikation die Nettowerte der gesamten nächsten Schicht in Form eines vertikalen Vektors berechnen:

$$X_{i+1} = W_i \times A_i.$$

Wird die Aktivierungsfunktion so definiert, dass $\sigma \begin{pmatrix} a \\ \vdots \\ b \end{pmatrix} = \begin{pmatrix} \sigma(a) \\ \vdots \\ \sigma(b) \end{pmatrix}$ gilt, ergeben sich die Aktivierungslevel einer Schicht:

$$A_{i+1} = \sigma(X_{i+1}) = \sigma(W_i \times A_i)$$

mit der Startbedingung, dass A_1 der Eingabe des Netzes entspricht. Weiter verallgemeinert, lässt sich die Ausgabe y iterativ formulieren als

$$y = A_{c(S)} = \sigma(W_{c(S)-1} \times A_{c(S)-1}).$$

Ein klassifizierendes Netz wird so trainiert, dass sich die Ausgabe y der Zielausgabe y^* annähert. Typischerweise wird während der Trainingsphase und bei dem Anwendungsfall eine Funktion wie der Softmax auf die Ausgabe angewandt, sodass diese als Liste von Konfidenzen interpretiert werden kann. Dann enthält y an der i-ten Stelle die Wahrscheinlichkeit, mit der das Netz die Eingabe der i-ten Klasse zuordnen würde. Handelt es sich um eine binäre Klassifikation, wird das Netz in der Regel mit den Zielen -1 und $+1$ trainiert, sodass an dem Vorzeichen des Ausgabeneurons zu erkennen ist, welcher Typ vorhergesagt wird.
Ist ein neuronales Netz bereits trainiert, finden die zwei folgenden Abschnitte keine Anwendung, da für die darin vorgestellten Konzepte die Zielausgabe y^* benötigt wird.

2.2.2 Bewertung eines Durchlaufs

Nach jeder Berechnung der Ausgabe $y = (y_1, ..., y_c)^T$ wird mittels einer Verlustfunktion die Abweichung von der Zielausgabe berechnet und als Loss oder $L(y^*, y)$ bezeichnet. Es existieren diverse Verlustfunktionen, die verschiedene Charakteristika der Abweichung mit unterschiedlicher Stärke bestrafen. Die native Form ist die Formel $L_1 = \frac{1}{c} \cdot \sum_{i=1}^{c} |y_i^* - y_i|$, die die Abweichung für alle c Klassen gleich gewichtet. Eine bei Regressionsproblemen übliche Verlustfunktion ist die mittlere quadratische Abweichung (MSE), die durch Quadrierung große Differenzen stark bestraft. Die davon berechnete Abweichung lautet $L_{\mathrm{MSE}} = \frac{1}{c} \cdot \sum_{i=1}^{c} (y_i^* - y_i)^2$ [12].
Im Gegensatz dazu fokussiert die bei Klassifikationen häufig in Kombination mit der Softmax-Funktion verwendete Cross-Entropy-Loss-Funktion falsche, hoch konfidente und korrekte, niedrig konfidente Klassifikationen. Das Loss der Kreuzentropie ist definiert als $L_{\mathrm{CE}} = -\sum_{i=1}^{c} y^* \cdot \log y_i$, was bei Klassifikationsproblemen deutlich vereinfacht wird. Für eine Eingabe der k-ten Klasse ist die Zielausgabe y^* ein sogenannter *one-hot-vector*, der abgesehen von einer Eins an der k-ten Stelle nur Nullen enthält. Das Loss hierfür ist daher $L_{\mathrm{CE}} = -\log y_k$ und bildet das Interval $[0, 1]$ auf $[0, \infty]$ ab, wobei $0 \mapsto \infty$ und $1 \mapsto 0$ gilt [13].
Die Verlustfunktion betrachtet die Eingabe x als fixiert und bewertet nur die Gewichte des Netzes. Da die Abweichung der Ausgabe zur Zielausgabe minimiert werden soll, müssen die Gewichte im Rückwärtsschritt so angepasst werden, dass das Loss möglichst gering ist, wie es in Abbildung 2 für ein Gewicht zu sehen ist. Hierbei gibt es im wesentlichen zwei Ansätze: Bei dem *batch processing* werden im Gegensatz zum *stream processing* die Gewichte nicht nach jedem Durchlauf verändert, sondern die berechneten Anpassungen über mehrere Durchläufe in einem Batch gesammelt und dann kumuliert angewandt.

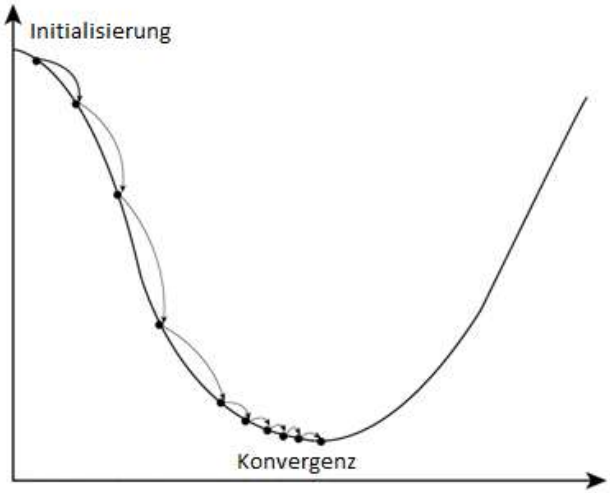

Abbildung 2: Anpassungen der Gewichte zur Verbesserung des Loss.
Jeder Schritt entlang des Graphen beschreibt eine Anpassung der Gewichte (X-Achse), die eine Veränderung des Loss (Y-Achse) bewirkt.

2.2.3 Anpassung der Gewichte

Dieser Abschnitt orientiert sich an der Vorlesung CS231n der Stanford University [14].
Folgt man den Schritten der letzten zwei Abschnitte, kann bei gegebenen Daten für einen Satz W von Gewichten durch das Loss auf die Qualität dieser Gewichte geschlossen werden. Ein trivialer Ansatz, um die besten Gewichte zu finden, ist es, für alle Gewichte zufällige Werte zu wählen, bis ein zufriedenstellender Satz erreicht ist. Die Netzarchitektur, die sich unter Kapitel 4.2 als die Beste für das Problem dieser Arbeit erweist, enthält über zwei Milliarden Gewichte, für die der triviale Ansatz eine inakzeptable Laufzeit hat.
Betrachtet man ein KNN hingegen nicht als starre Datenstruktur, sondern als komplexe Funktion $f(x, W) = y$, lässt sich das Loss als Funktion der Gewichte formulieren $L_f(W) = L(y^*, f(x, W))$, dessen globales Minimum für fixierte x und y^* gesucht wird. So wie das Minimum einer einfachen Funktion bestimmt wird, kann die Ableitung von $L_f(W)$ an der Stelle W berechnet werden, welche die Steigung in dem hochdimensionalem Raum beschreibt. Daraufhin können die Gewichte entgegen dieses Gradienten ∇W angepasst werden. Hier findet die oben definierte Lernrate μ Verwendung, die die Stärke dieser Anpassung reguliert. In der Regel gilt $L_f(W) > L_f(W - \nabla W \cdot \mu)$. Wird diese Berechnungen bis zur Konvergenz iteriert, ist ein lokales Minimum von L_f gefunden. Durch Wiederholung des Verfahrens mit zufälligen initialisierten Gewichten, wird das globale Minimum und somit die optimalen Gewichte des Netzes gefunden [15].
Die Ableitung der Funktion $L_f(W)$ nach W, also $\frac{\delta}{\delta W} L_f(W)$, wird durch die rekursive Anwendung der Ketten-Regel bestimmt.
Als Beispiel sei die Funktion $f(a, b, c) = (a + b) \cdot c = y$ gegeben, für deren drei Parameter die Gradienten im Punkt $p = (-2, 5, -4)$ zu bestimmen sind. Zuerst wird die Hilfsfunktion $h(a, b) = a + b = h$ definiert, sodass sich f zu $y = h \cdot c$ vereinfachen lässt. Dann wird im *forward pass* $h(-2, 5) = -2+5 = 3$ und $f(h, c) = 3 \cdot -4 = -12$ berechnet. Im Anschluss lassen sich die lokalen Gradienten von h und den Parametern a, b und c bestimmen:

$$\begin{aligned}
\frac{\delta h}{\delta a} &= \frac{\delta(a+b)}{\delta a} &&= \frac{\delta h}{\delta b} = 1 \\
\frac{\delta f}{\delta c} &= \frac{\delta(h \cdot c)}{\delta c} &&= h \\
\frac{\delta f}{\delta h} &= \frac{\delta(h \cdot c)}{\delta h} &&= c.
\end{aligned} \tag{3}$$

Sei die Aussage einer Bewertung, dass die Ausgabe mit einer Stärke von 1 erhöht werden soll. Dieses Signal wird rückwärts entlang der Gatter getragen, wobei jedes Gatter vom dem Gradienten seines Outputs auf die Gradienten seines Inputs schließt, und diese Gradienten bis zu den Eingangs-Parametern propagiert werden. Daher stammt die Bezeichnung *backpropagation*. Mit welcher Stärke und in welche Richtung die Ausgabe angepasst werden soll, wird von der Ableitung der Verlustfunktion festgelegt.
An dieser Stelle werden die Gradienten aus (3) benötigt, denn nach der Ketten-Regel gilt:

$$\begin{aligned}
\frac{\delta f}{\delta a} &= \frac{\delta f}{\delta h} \cdot \frac{\delta h}{\delta a} = c \cdot 1 &&= -4 \\
\frac{\delta f}{\delta b} &= \frac{\delta f}{\delta h} \cdot \frac{\delta h}{\delta b} = c \cdot 1 &&= -4 \\
\frac{\delta f}{\delta c} &= h &&= 3.
\end{aligned}$$

Der Gradient $\nabla f = (\frac{\delta f}{\delta a}, \frac{\delta f}{\delta b}, \frac{\delta f}{\delta c})$ an dem Punkt $p = (-2, 5, -4)$ ist also $\nabla f(p) = (-4, -4, 3)$. Durch Subtraktion dieses Gradienten von den ursprünglichen Parametern wird ein höheres Ergebnis als zuvor erwartet, was die folgende Rechnung bestätigt:

$$f(p - \nabla f) = f(2, 9, 7) = (2 + 9) \cdot 7 = 77 > -12 = f(-2, 5, -4) = f(p).$$

Um die Auswirkung der Subtraktion zu regulieren, werden der Gradienten zuvor mit der oben genannten Lernrate μ multipliziert. Meist liegt μ zwischen 0.00001 und 0.01, damit die Parameter nicht zu sprunghaft angepasst werden und so verhindert wird, dass optimale Einstellungen verpasst werden. Gilt $\mu = 1$, bewirkte die Subtraktion des Gradienten jedes Gewichts, dass für diese Eingabe x die Ausgabe $y = y^*$ ist. Aufsummiert über p Gewichte wäre die Veränderung p-fach zu stark. Zusätzlich überschreibt eine zu hohe Lernrate die zuvor gelernten Tendenzen, sodass das Netz nicht generalisieren kann. Die Abbildung 3 visualisiert die Auswirkung von falsch gewählten Lernraten.

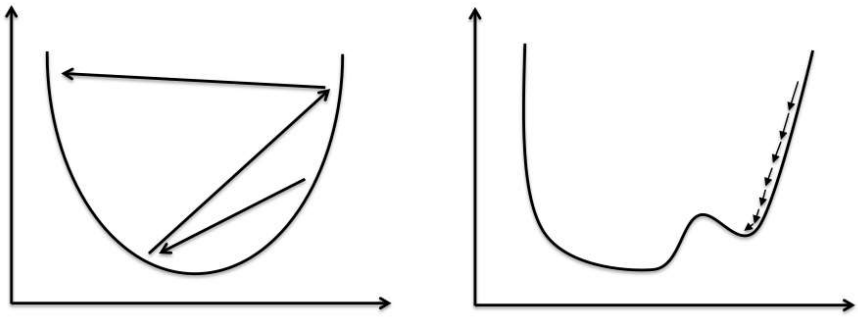

Abbildung 3: Auswirkung der Lernrate.
Links ist das *overshooting* einer zu großen Lernrate zu sehen und rechts die Falle eines lokalen Minimums bei zu kleiner Lernrate. Ein exemplarisches Gewicht ist auf der X-Achse und das entsprechende Loss auf der Y-Achse aufgetragen.

Die i-te Anpassung der Parameter p_i ist wie folgt definiert:

$$p_{i+1} = p_i - \mu \cdot \nabla f(p_i). \tag{4}$$

Wird p durch die Gewichte W des Netzes ersetzt und alle Ableitungen der verwendeten Gatter berechnet, resultiert daraus die Berechnungsvorschrift für ∇W. Damit können die Gewichte eines KNN optimiert werden.
Die Ableitung einer Funktion f, wie in Gleichung 3, kann immer gebildet werden, wenn diese Funktion aus differenzierbaren Teilfunktionen besteht. Ist eine Funktion zur Berechnung des Aktivierungslevels des Knoten k mit $a_k = f(a_i, a_j, w_{k,i}, w_{k,j}) = \sigma(w_{k,i} \cdot a_i + w_{k,j} \cdot a_j)$ gegeben (in diesem Fall sei σ die Sigmoid-Funktion), wird diese Funktion in mehrere Teilfunktionen gespalten. Von diesen müssen die Gradienten der Inputs berechnet werden, die später mit dem Gradienten des Outputs multipliziert werden.

$$\begin{aligned} f(a_i, a_j, w_{k,i}, w_{k,j}) &= \sigma(w_{k,i} \cdot a_i + w_{k,j} \cdot a_j) \\ &= \sigma(\mathrm{add}(\mathrm{mul}(w_{k,i}, a_i), \mathrm{mul}(w_{k,j}, a_j))). \end{aligned}$$

Die sigmoide Funktion ist $\sigma(x) = \frac{1}{1+e^{-x}}$ und die Hilfsfunktionen add und mul sind definiert als $\mathrm{add}(x, y) = x + y$ und $\mathrm{mul}(x, y) = x \cdot y$.
Die Gradienten der Hilfsfunktionen sind bereits aus der Gleichung 3 bekannt und die $\mathrm{Sigmoid}$-Funktion ist so gewählt, dass ihre Ableitung effizient berechenbar ist. In ihre Komponenten zerlegt, lässt sich die Ableitung einfach berechnen:

$$\begin{aligned}\sigma(x) &= \frac{1}{1+e^{-x}} \\ &= \mathrm{inv}(\mathrm{inc}(\exp(\mathrm{neg}(x)))),\end{aligned}$$

mit den Hilfsfunktionen

$$\begin{aligned}
\mathrm{inv}(x) &= \frac{1}{x} & \longrightarrow \quad \frac{\delta\,\mathrm{inv}}{\delta x} &= -\frac{1}{x^2} \\
\mathrm{inc}(x) &= 1 + x & \longrightarrow \quad \frac{\delta\,\mathrm{inc}}{\delta x} &= 1 \\
\exp(x) &= e^x & \longrightarrow \quad \frac{\delta\,\exp}{\delta x} &= e^x \\
\mathrm{neg}(x) &= -x & \longrightarrow \quad \frac{\delta\,\mathrm{neg}}{\delta x} &= -1.
\end{aligned}$$

Durch Anwendung der Ketten-Regel, ergibt sich die Ableitung der $\mathrm{Sigmoid}$-Funktion

$$\frac{\delta\sigma(x)}{\delta x} = \sigma(x) \cdot (1 - \sigma(x))$$

und der Vorteil dieser Funktion als Gatter wird offensichtlich: Bildet $\sigma : x_0 \mapsto a_0$ ab und kommt im *backward pass* der Output-Gradient ν zurück, ist die Berechnung des weiter zu propagierenden Gradienten an der Stelle x_0 mit $a_0 \cdot (1 - a_0) \cdot \nu$ extrem effizient. Zusätzlich staucht die $\mathrm{Sigmoid}$-Funktion das Intervall $[-\infty, \infty]$ auf das bezüglich Wahrscheinlichkeiten leichter interpretierbare Intervall $[0, 1]$.
Der Hauptgrund, warum die $\mathrm{Sigmoid}$-Funktion kaum mehr Anwendung findet, ist die annähernde Stagnation von $\sigma(x)$ für $x < -10$ oder $x > 10$, durch die der Gradient an Punkten dieser Bereiche gegen Null konvergiert. Da der Output des Gatters der Aktivierungsfunktion mit dem lokalen Gradienten des Inputs multipliziert wird, erreicht bei einem so kleinen lokalen Gradienten nur ein Bruchteil des Signals die vorherigen Gatter.
Für andere Aktivierungsfunktionen wie die Identität, die ReLU-Funktion oder der Tangens hyperbolicus sind die Ableitungen wohldefiniert und ähnlich einfach wie die $\mathrm{Sigmoid}$-Funktion:

$$\begin{aligned}
\mathrm{id}(x) &= x & \longrightarrow \quad \frac{\delta\,\mathrm{id(x)}}{\delta x} &= 1 \\
\mathrm{ReLU}(x) &= \max(x, 0) & \longrightarrow \quad \frac{\delta\,\mathrm{ReLU(x)}}{\delta x} &= \left\{ \begin{array}{ll} 1 & \text{, falls } x > 0 \\ 0 & \text{, falls } x \leq 0 \end{array} \right\} \\
\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} & \longrightarrow \quad \frac{\delta\,\tanh(x)}{\delta x} &= 1 - \tanh^2(x).
\end{aligned}$$

Der Tangens hyperbolicus erweitert die $\mathrm{Sigmoid}$-Funktion um einen negativen Wertebereich, sodass die Ausgabe um Null zentriert ist und so für ein neuronales Netz bessere generelle Eigenschaften aufweist. Es kann immer noch zur Sättigung des Gatters

kommen, da $\tanh(x) = 2 \cdot \sigma(2x) - 1$ gilt und daher die Gradienten für stark positive sowie stark negative x ebenfalls gegen Null konvergieren. Bei der ReLU-Funktion tritt die oben genannte Saturation nicht auf, da der Gradient für $x > 0$ stets 1 ist, jedoch können die Gatter bildlich gesehen absterben. Mit einer tendenziell kleineren Lernrate kann diesem Problem vorgebeugt werden. Alternativ wird mit einer parametrisierten Version der ReLU-Funktion wie $\mathrm{PReLU}_\alpha(x) = \left\{ \begin{array}{ll} x & \text{, falls } x > 0 \\ \alpha \cdot x & \text{, falls } x \leq 0 \end{array} \right\}$ das Absterben verhindert. Das Alpha ist dabei ein Parameter der wie ein Gewicht durch das Training optimiert wird [16].

3 Komponenten

3.1 Workflow-Management-System

Für diese Arbeit bietet snakemake die Möglichkeit, unabhängig vom Betriebssystem und den installierten Bibliotheken, komplexe Abläufe mit einer Python ähnlichen Syntax zu vereinfachen. Das Konzept von einem Snakefile entspricht dem eines Makefiles (GNU) : Es lassen sich Regeln deklarieren, die Anweisungen enthalten, die abhängig von dem Input einen Output generieren. Durch dieses Konzept können von der Zieldatei rückwirkend Abhängigkeiten in einer Baumstruktur erfasst und so Parallelitäten ausgenutzt werden [17].
Jede Regel bietet die Möglichkeit ein eigenes conda-Umfeld zu definieren, das Bibliotheken einer wählbaren Version enthält, die nur innerhalb von snakemakes Umgebung angelegt werden. Für dieses Arbeit ermöglicht snakemake die zeitgleiche Simulierung von Sequenzierungsdaten mehrerer MRSA-Typen, die Verwaltung diverser Bibliotheken mit conda und die Zentralisierung der Konfiguration [18].

3.2 Simulierung der Sequenzierung

Bei wgsim handelt es sich um ein Programm, das als Eingabe ein Referenzgenom erwartet und als Ausgabe Reads generiert, wie sie bei einer Sequenzierung dieses Genoms entstehen würden [19]. Neben der Anzahl und Länge der Reads lässt sich die Fehlerrate einstellen. Da das neuronale Netz für Reads von Illuminas HiSeq System trainiert werden soll, würde die dafür typische Errorrate von einem Prozent gewählt [20].
Als Referenzgenom dienen die Sequenzen des Ridom Spa Servers, die an beiden Enden mit 100 Basenpaaren aufgefüllt werden, sodass die Reads bei der Sequenzierung gleichverteilt über die typspezifische Sequenz auftreten. Die Füllung wird beispielhaft aus der Region X vom Typen t037 ausgeschnitten. Die genauen Koordinaten des Ausschnitts können der Konfigurationsdatei entnommen werden.

3.3 Bibliothek für maschinelles Lernen

Neben Tensorflow ist pytorch eine der anerkanntesten Bibliotheken für Python im Bereich des maschinellen Lernens. Es beinhaltet unter anderem Implementierungen von diversen Netzwerkschichten, Optimierern, Verlustfunktionen und Gattern [21]. Das Training kann parallelisiert auf CPUs oder per CUDA [22] auf GPUs ausgeführt werden. Alternativ besteht die Möglichkeit, bereits trainierte Modelle wie das ResNet-18 oder VGG-16 zu laden. Zusätzlich zu der nahtlosen Verbindung zu verbreiteten Bibliotheken wie numpy und cython, bietet pytorch die Möglichkeit, eigene Daten in Datensätze zu konvertieren. Diese enthalten die Eingaben und zugehörigen Klassen, die per Dataloader effizient und automatisiert geladen und sofort in das Netz eingesetzt werden können.
Die Klassen und Funktionen der pytorch Bibliothek implementieren alle in Kapitel 2 aufgeführten notwendigen Schritte, die in dem Trainingszyklus eines künstlichen neuronalen Netzwerk ausgeführt werden. Das heißt, die Berechnung der Ausgabe durch vielfache Multiplikationen, Summierungen und Anwendungen der Aktivierungfunktionen im *forward-pass*, die Bestimmung des Losses sowie die komplexen Kalkulationen von Gradienten und die Anpassungen der Parameter im *backward-pass* werden mit wenigen Zeilen Code aufgerufen und effizient ausgeführt [21]. Hierfür können Optimierer genutzt werden, die verschiedene Strategien verwenden, um das Netz zu trainieren. Der aktuell favorisierte Optimierer ist Adam [23], dessen adaptives Momentum die besten Eigenschaften der beiden Vorgänger AdaGrad und RMSProp kombiniert. Adam stellt beispielsweise für jedes Gewicht eine eigene Lernrate auf und nimmt vorherige Gradienten zum Teil in die Berechnung von ∇W auf.
Sämtliche Berechnungen für diese Arbeit wurden auf dem High-Performance Server des Zentrums für Informations- und Medientechnologie (ZIM) der Universität Düsseldorf (Deutschland) mit 16 Intel® Xeon® Prozessoren E5-2667 v4 (3.20 GHz) und 503 GB RAM durchgeführt.

4 Implementierung

4.1 Arbeitsschritte

Der Workflow von snakemake wird im `Snakefile` definiert, das aus verschiedenen Regeln besteht. Die Ziel-Regel,die die Wurzel im Baum der Abhängigkeiten (siehe Abbildung 4) bildet, ist nach der Konvention die `all` Regel, die eine `score.txt` Datei erwartet, wie sie von der `train_net` Regel temporär erzeugt wird. Diese Regel ruft das Python Skript `trainTestNN.py` auf, welches das in der Konfigurationsdatei beschriebene Netz zu laden versucht, oder es alternativ generiert, dann trainiert und anschließend validiert. Hierfür werden Reads benötigt, die die Regeln `simulate_train` und `simulate_val` mit wgsim erzeugen, wobei die Reads für die Validation nur temporär angelegt werden. Beide Regeln können dank snakemake jeweils parallel laufen, sodass innerhalb von einer Sekunde die Sequenzierung von etwa 50 Typen simuliert werden kann. Die read-generierenden Regeln sind abhängig von der `all_types_to_fa` Regel, die die gefragten Typen aus der Textdatei `types_pad.txt` extrahiert und in einer

separaten Datei speichert. In `types_pad.txt` sind alle 18.702 (Stand 1. Mai 2019) typspezifischen Sequenzen mit Füllung enthalten, wie in Abschnitt 3.2 beschrieben.
Neben dem eigentlichen Workflow erzeugt die Regel `view_scores` mit dem Skript `scoreViewer.py` Grafiken aus den Dateien, die während des Trainings generiert werden, um verschiedene Messwerte über den Trainingsablauf visualisieren zu können.
Über snakemakes Parameter-Direktive werden in sämtliche Regeln Hyperparameter eingepflegt, die alle in der `config.yaml` Datei zentral definiert und gespeichert sind. Aktuell sind nur die laut Ridom Spa Server 200 häufigsten Typen enthalten, von denen der Seltenste mit einer Frequenz von 0.05% auftritt. Diese Liste kann jederzeit erweitert oder verändert werden, sodass neue Netze generiert und trainiert werden können. Die Wahl der Hyperparameter basiert auf experimentellen Trainingsergebnissen, wie sie im Kapitel 5 dargestellt und Kapitel 6 interpretiert werden.

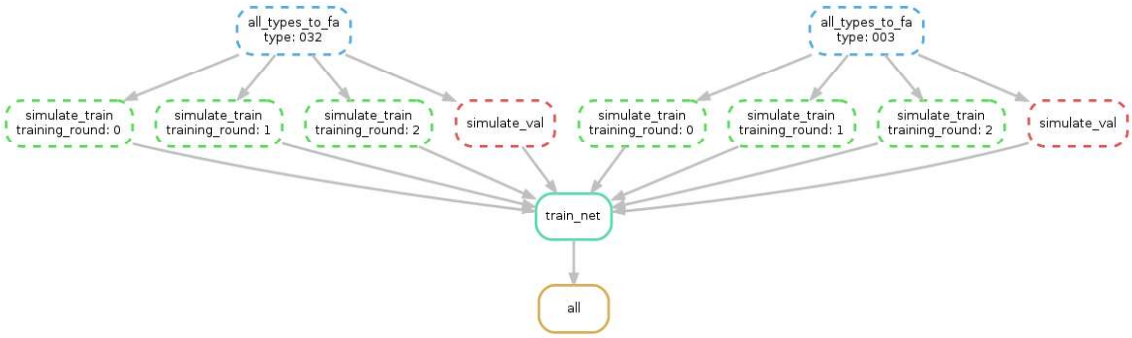

Abbildung 4: Workflow von snakemake.
Der gerichtete azyklische Graph zeigt die Abhängigkeiten der fünf Regeln eines verschlankten Trainingsdurchlaufs mit nur zwei Typen und drei Epochen.

4.2 Architektur des künstlichen neuronalen Netzes

Die gewählte Modellierung des KNN entspricht dem Standardmodell aus dem Bereich des Deep Learning. Das Netz wird von vier Schichten gebildet:

- Eingabeschicht
- Zwei verdeckte Schichten
- Ausgabeschicht

Eine verdeckte Schicht reicht für die Lösung dieses Problems nicht aus, weil die Profile der Reads nicht mit einer stetigen Abbildung den Typen zugeordnet werden können. Da ein Netz mit zwei verdeckten Schichten bereits jede glatte Abbildung zu einer unbeschränkten Genauigkeit erlernen kann, erzielt eine dritte verdeckte Schicht nicht zwingend bessere Ergebnisse [24].

Die Aufbereitung der Daten beinhaltet einen Ansatz mit gewichteten k-Meren. Das heißt, die Anzahlen der Teilsequenzen der Länge k werden ermittelt und anschließend die Quoten berechnet. Die Sequenz $ATGAT$ enthält die 2-Mere $\{AT, TG, GA, AT\}$, wobei AT am häufigsten auftritt, daher mit einer Eins repräsentiert wird. Die beiden anderen 2-Mere TG und GA sind halb so oft wie AT enthalten, daher wird der Wert 0.5 zugewiesen. An den Stellen aller anderen 2-Mere steht eine Null. Ein k-Mer-Ansatz bietet sich an, da sich die Reads durch ihre variable Länge nicht als direkte Eingabe eignen.
Die erste Schicht besteht aus 4^k Knoten, entsprechend der Anzahl der möglichen k-Mere bei vier Nukleotidbasen (Adenin, Cytosin, Guanin und Thymin). Die Eingabe ist alphabetisch nach den korrespondierenden k-Meren sortiert, wie es beispielhaft in der Funktion `count_k_meres` des Skripts `Dataset.py` implementiert ist.
Zur zweiten Schicht findet eine lineare Matrixmultiplikation mit den Gewichten statt, worauf eine ReLU-Aktivierungsfunktion folgt. Der gleiche Ablauf gilt für die dritte Schicht, wobei die beiden verdeckten Schichten aus der Hälfte der Knoten der jeweiligen vorangehenden Schicht bestehen. Dieses Verhältnis hat die besten Ergebnisse erzielt und ist zurückzuführen auf eine verborgene Struktur der Informationen, die in der Eingabe enthalten sind. Die letzte Schicht wird von 200 Knoten gebildet, was der Anzahl der betrachteten Typen entspricht.
Es wurde das Cross-Entropy-Loss [21] gewählt, das eine Kombination aus dem logarithmischen Softmax und der negativen logarithmischen Likelihood bildet. Diese beiden Komponenten bieten sich hier an, da die Minimierung der negativen logarithmischen Likelihood genau der Maximierung der Konfidenz der korrekten Klasse entspricht. Wird vor der Berechnung des Losses ein logarithmischer Softmax auf die Ausgabe angewandt, bewirkt die Maximierung der korrekten Klasse gleichzeitig die Minimierung der Konfidenz der inkorrekten Klassen. Durch diese Kombination trainiert das Netz in Richtung der für den Benutzer interpretierbaren Ausgabe.
Die durchschnittliche quadratische Abweichung (MSE) ist für eine Klassifikation von einer hohen Anzahl Klassen wie 200 ungeeignet, da keine Priorisierung stattfindet. Zusätzlich wird die Ausgabe nicht normalisiert, sodass sie vom Nutzer nicht gedeutet werden kann [21].

4.3 Training

Eine effektive Architektur, wie sie in Abschnitt 5.2 vorgestellt wird, ist das Ergebnis von vielfachen Vergleichen verschiedener Instanzen eines neuronalen Netzes. Zuerst müssen die Parameter der Datenaufbereitung (Länge der Füllung, sowie Länge, Anzahl und Errorate der Reads bei wgsim) festgelegt werden. Dann werden die Hyperparameter des Modells eingestellt und ihre Ausprägungen kombiniert getestet:

Hyperparameter	Ausprägungen	Kommentar
k	{1, ..., 8, ... }	$k \geq 10$ verlangsamen das Netz stark.
Knoten Schicht 1	{0, ..., 1024, ... }	frei wählbar.
Knoten Schicht 2	{0, ..., 1024, ... }	0 wenn erste Schichte keine Knoten hat.
Aktivierungsfunktion	$\{\text{Sigmoid}, \tanh, \text{ReLU}, ...\}$	siehe Kapitel 2
Ausfall	[0, 1)	siehe Abschnitt 2.2.1
Lernrate	[0.00001, 0.01]	siehe Abschnitt 2.2.3.
Verlustfunktion	{Cr. Entropy, MSE, NLL, ...}	siehe Abschnitt 2.2.2.
Optimierer	{Adam, SDG, RMSProp ,...}	jeder Optimierer hat eigene Parameter.
Größe des Bündels	{..., 256, 512, ...}	aus numerischen Gründen 2^n

Es ist nicht möglich, jeden Hyperparameter gesondert zu betrachten, weil diese abhängig voneinander sind, wie beispielsweise die Anzahl der Schichten, die der jeweiligen Knoten und die Größe von k. Letzteres determiniert die Anzahl der Knoten in der Eingabeschicht. Um die Anzahl der zu testenden Hyperparameter zu verringern, wird im Vorhinein der Adam Optimierer und die Kreuz-Entropie als Verlustfunktion festgelegt, da diese für das vorliegende Problem, wie in den Abschnitten 3.3 und 2.2.2 beschrieben, am besten geeignet sind. Weiterhin sorgt eine Bündelgröße von 256 Proben für die besten Ergebnisse, wenn also 256 ∇W berechnet und dann gemittelt auf die Gewichte angewandt werden. Demnach fallen drei der neun Variablen aus der Gleichung. Im nächsten Schritt wird eine Reihe möglicher Ausprägungen jedes Parameters aufgestellt. Basierend auf einzelnen Tests, ergeben sich folgende Werte:

- k = {6, 8, 10}
- Knoten Schicht 1 = $\{0, \frac{1}{2}, \frac{1}{4}\} \cdot 4^k$.
- Knoten Schicht 2 = $\{0, \frac{1}{4}, \frac{1}{16}, \frac{1}{64}\} \cdot 4^k$.
- Aktivierungsfunktionen = $\{\text{Sigmoid}, \tanh, \text{ReLU}, \text{PReLU}\}$
- Lernrate = {0.0001, 0.001, 0.01}
- Ausfall = {0, 0.1, 0.25, 0.5}

Die Anzahl der dann zu testenden Kombinationen ergibt sich durch das Produkt der Kardinalitäten der Hyperparameter, also $3 \cdot (1 + 2 \cdot 4) \cdot 4 \cdot 3 \cdot 4 = 1.296$ Netze.
Unabhängig von den anderen Parametern, führt eine höhere Ausfallquote zu schlechteren Ergebnissen (vgl. Abbildungen 15 und 16). Weiterhin sorgt die Lernrate von 0.0001 für zu langsame Verbesserungen der Gewichte, wie in den Abbildung 11 und 12 zu sehen ist. Die Netze mit $k = 10$ können aufgrund von mangelnden Ressourcen nicht mit

verdeckten Schichten getestet werden, die aus annähernd $4^{10} \cdot \frac{1}{4} = 262.144$ Knoten bestehen. Dementsprechend besteht nur die Möglichkeit, vereinzelte Netze mit $k = 10$, also $1.048.576$ Eingabe-Knoten zu testen. Insgesamt reduziert sich damit die Anzahl der zu testenden Netze auf $12 + 2 \cdot (1 + 2 \cdot 4) \cdot 4 \cdot 2 \cdot 1 = 156$.
Um eine derartige Anzahl von Tests durchzuführen, ist es notwendig, eine effiziente Datenversorgung aufzubauen. Es werden einmalig für die Sequenzen aller Typen, auf die das Netz trainiert werden soll, eigene Sequenzierungsdaten angelegt, die dann wiederholt als Referenzgenom von wgsim genutzt werden können. Um ein Gleichgewicht zwischen Speichernutzung und Effektivität des Trainings zu erhalten, werden vor dem ersten Trainingsdurchlauf für jeden der 200 Typen je 128 Proben für das Training und vier Proben für die Validierung generiert. Nachdem die Proben von dem Dataloader geladen wurden, werden sie mit einer Wahrscheinlichkeit von 20% gelöscht und vor dem nächsten Trainingsdurchlauf neu generiert. Dadurch ist sichergestellt, dass ein Netz die meisten Proben mehrfach sieht und gleichzeitig immer wieder neue Reads erhält, was zu Generalisierung beiträgt. Jedes Netz, das in dem Trainingsdurchlauf inbegriffen ist, trainiert mit den gleichen Daten und wird mit den gleichen Proben validiert, sodass ein Vergleich möglich ist. Sobald ein KNN eine Probe von jeder Klasse gesehen hat, wird dies als Epoche bezeichnet. Ein Trainingsdurchlauf entspricht daher 128 Epochen.
Wie ein Netz bewertet wird, hängt von den gewählten Kennzahlen ab. Dabei werden in erster Linie die Quote der richtig klassifizierten Proben und die durchschnittliche Konfidenz in den tatsächlichen Typen betrachtet. Letzteres deckt durch die angewandte Softmax-Funktion auch die Konfidenz in die falschen Typen ab, die es zu minimieren gilt. Als richtig klassifizierte Proben gelten die, für die der korrekte Typ als der Wahrscheinlichste vorhergesagt wurde, also die eigentliche Präzision und die, für die der korrekte Typ als einer der fünf Wahrscheinlichsten ausgegeben wurde. Diese Kennzahl wird als Top-5-Präzision bezeichnet. Die Werte werden über die vier Testdurchläufe jedes Typen gemittelt, sodass Ausreißer weniger starke Auswirkungen haben. Andere, als weniger wichtig eingestufte Zahlen sind die maximale und minimale Konfidenz in die korrekte Klasse. Weiterhin werden Speichernutzung und Geschwindigkeit nicht beachtet, da diese Kennzahlen zur Anwendungszeit skalieren, und dementsprechend performant sind, wenn nur wenige Proben klassifiziert werden sollen.
Das Training erfolgt in drei Phasen:

1. Alle Kombinationen absolvieren zwei Trainingsdurchläufe. Anschließend werden die Netze gelöscht, die eine der folgenden Kriterien nicht erfüllen:
   - Top-5-Präzision ≥ 99
   - Präzision ≥ 85
   - Konfidenz ≥ 80.
2. Die übrigen 50 Kandidaten trainieren für 128 weitere Epochen. Nur die 10 stärksten Netze kommen in die nächste Phase.
3. Aus den Favoriten wird in zwei weiteren Trainingsdurchläufen das optimale beziehungsweise stärkste Netz festgestellt.

Die Stärke eines Netzes ergibt sich aus dem gewichteten Mittel der drei oben genannten Kennzahlen: 0.5 · Top-5-Präzision + 0.3 · Konfidenz + 0.2 · Präzision. Das Ziel dieser Arbeit ist es, ein Netz zu konstruieren, dass zuverlässig den korrekten Typen als einen der Wahrscheinlichsten ausgibt, daher ist die Top-5-Präzision die wichtigste Kennzahl. Die 50 Netze, die die zweite Phase erreichen, übertrafen bereits eine Top-5-Präzision von 99%. An zweiter Stelle steht die Konfidenz, wofür es mehrere Gründe gibt. Zum einen impliziert eine hohe Konfidenz direkt eine hohe Präzision, da für jede Probe (Konfidenz > 50%) ⇒ (Präzision = 1) gilt. Zum anderen sagt eine hohe Konfidenz aus, dass das Netz erfolgreich trainiert wurde, denn die Verlustfunktion bezieht sich explizit auf die Konfidenz und nur implizit auf die Präzision.

5 Evaluation

In diesem Kapitel werden Charakteristika der getesteten neuronalen Netze miteinander verglichen, wesentliche Unterschiede herausgearbeitet und die besten Netze im Detail betrachtet. Dabei folgt die Benennung der Netze dem einheitlichen Stil *L3*,*L2*,*L1*,*act*,drop:*dp*,lr:*lr* mit folgenden Bedeutungen:

- *L3* - Anzahl der Knoten in der zweiten verdeckten Schicht.
- *L2* - Anzahl der Knoten in der ersten verdeckten Schicht.
- *L1* - Anzahl der Knoten in der Eingabeschicht, wobei $\log_4 L1 = k$ gilt.
- *act* - Name der Aktivierungsfunktion (PReLU = parametrisierte ReLU- Funktion).
- *dp* - Quote der ausfallenden Knoten, dp= $0.3 \rightarrow 70\%$ der Knoten senden Signal.
- *lr* - Lernrate μ.

5.1 Wichtige Hyperparameter

Die folgenden Ergebnisse stellen je einen Hyperparameter dar, wobei die anderen Parameter meist fixiert sind, denn so lässt sich die Auswirkung der jeweiligen Ausprägung am besten beobachten. Um die optimale Einstellung zu finden ist es allerdings notwendig alle Kombinationen von Hyperparametern zu testen. Alle abgebildeten Netze wurden mit der ReLU-Funktion, $dq = 0$ und $\mu = 0.001$ trainiert, sofern keine anderen Angaben gemacht werden.
Wird nur die Größe der verdeckten Schichten betrachtet, ergeben sich die Abbildungen 5 und 6. Diese verdeutlichen, dass die Netzgröße bei einem kleinen k große Auswirkungen hat, bei einem größeren k jedoch keine wesentlichen Veränderungen hervorruft.

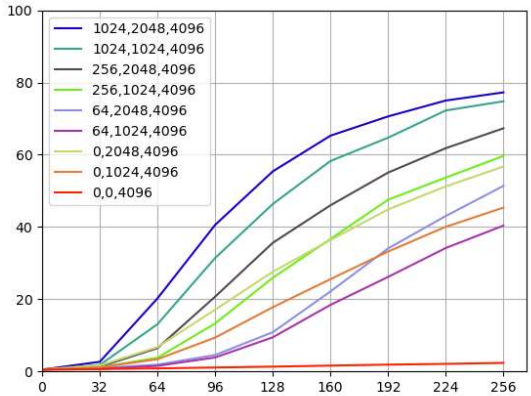

Abbildung 5: Netzgröße mit $k = 6$.
Konfidenz (Y-Achse) über 256 Epochen (X-Achse).

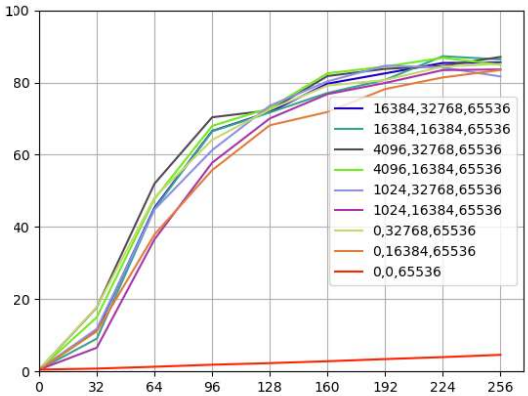

Abbildung 6: Netzgröße mit $k = 8$.
Konfidenz (Y-Achse) über 256 Epochen (X-Achse).

Nach der Erkenntnis, dass ein großes Netz tendenziell bessere Ergebnisse erzielt als ein kleines, besteht die Frage nach den Unterschieden bei den Größen von k. Abbildung 7 zeigt die Konfidenz in den korrekten Typen für alle $k \in \{6, 8, 10\}$ und den vier Aktivierungsfunktionen. Wie bereits in Abbildung 5 und 6 zu sehen ist, ist diese bei Netzen ohne verdeckte Schichten sehr niedrig. Dem gegenüber ist die Präzision deutlich höher (siehe Abbildung 8). Die Höhe von k korreliert positiv mit dem Lernerfolg. Das ist selbstverständlich, da im optimalen Fall ein k-Mer die Länge einer ganzen Sequenz hat und so die Eingabe zu einem *one-hot-vector* wird, der sogar ohne verdeckte Schicht per linearer Abbildung dem entsprechenden Typen zugewiesen werden kann.

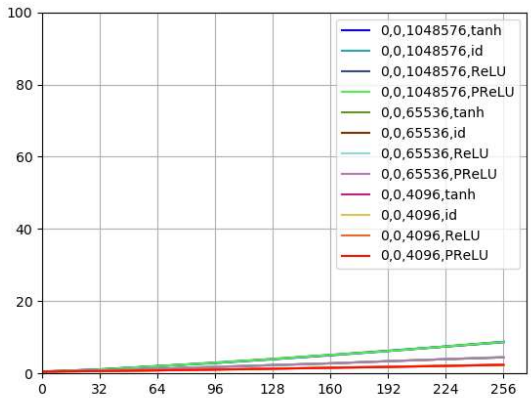

Abbildung 7: $k = \{6, 8, 10\}$.
Konfidenz (Y-Achse) über 256 Epochen (X-Achse).

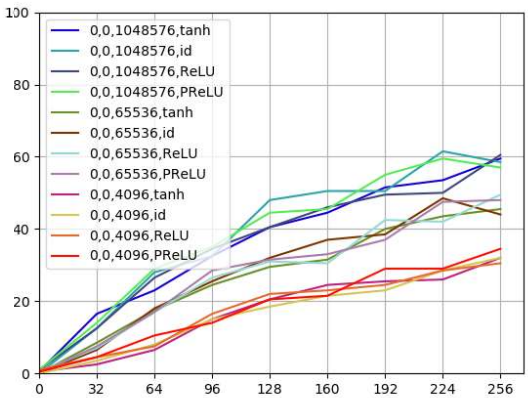

Abbildung 8: $k = \{6, 8, 10\}$.
Präzision (Y-Achse) über 256 Epochen (X-Achse).

Die Hinzunahme einer verdeckten Schicht bewirkt eine direkte Verbesserung. Den Abbildungen 9 und 10 kann entnommen werden, um welches Ausmaß ein Netz mit einer verdeckten Schicht besser klassifiziert. Die Konfidenz steigt besonders bei einem großen k.

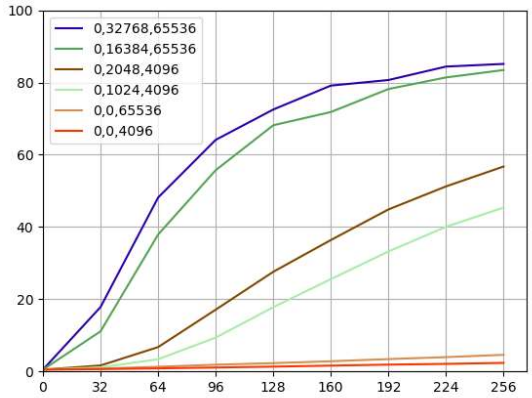

Abbildung 9: Eine verdeckte Schicht. Konfidenz (Y-Achse) über 256 Epochen (X-Achse).

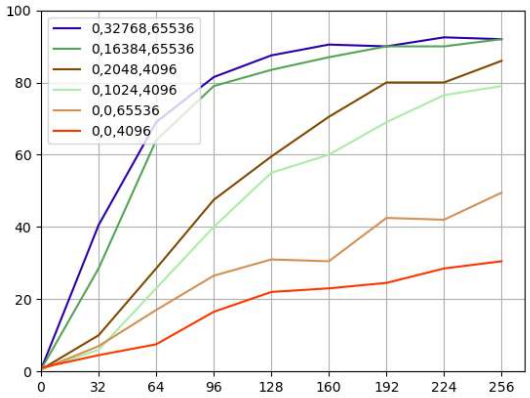

Abbildung 10: Eine verdeckte Schicht. Präzision (Y-Achse) über 256 Epochen (X-Achse).

Ist die optimale Größe gefunden, wird das Verhältnis der Lernrate μ zur Netzgröße betrachtet. Bei einem kleinen Netz bewirkt eine niedrige Lernrate kaum Veränderungen, ein zu großes μ sorgt dagegen für schlechtere Lernerfolge als eine mittlere Lernrate. Für größere Netze ist eine kleine Lernrate deutlich besser geeignet, was auf die höhere Anzahl der Gewichte zurückzuführen ist. Insgesamt stellte sich eine Lernrate von $\mu = 0.001$ als die Beste heraus (vgl. Abbildungen 11 und 12).

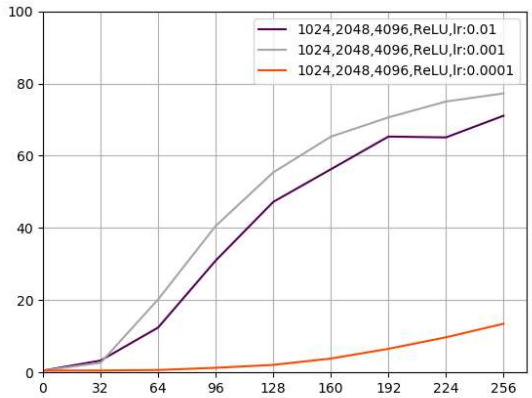

Abbildung 11: Lernrate $\sim$ kleines Netz. Konfidenz (Y-Achse) über Epochen (X-Achse).

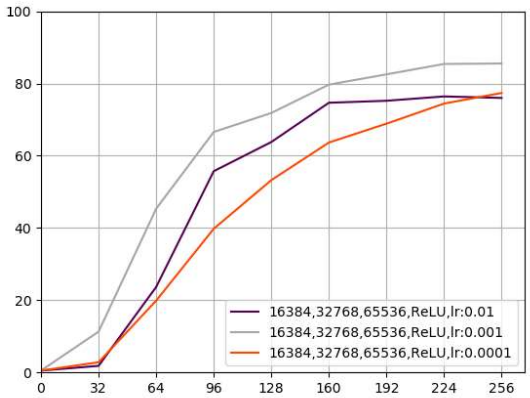

Abbildung 12: Lernrate $\sim$ großes Netz. Konfidenz (Y-Achse) über Epochen (X-Achse).

Beim Vergleich von verschiedenen Aktivierungsfunktionen σ wird deren Abhängigkeit von der Netzgröße klar (vgl. Abbildungen 13 und 14). Die ReLU-Funktionen sind deutlich geeigneter für große Netze als der Tangens hyperbolicus, welcher bei kleinen Netzen bessere Ergebnisse erzielt. Diese Beobachtung ist auf die Stagnation des $\tanh$ für große Eingabewerte zurückzuführen, wie in Abschnitt 2.2.3 beschrieben. Diese Auswirkung ist besonders in Abbildung 14 zu sehen: Das Netz mit dem Tangens hyperbolicus als Aktivierungsfunktion verzeichnet keine Verbesserungen über 256 Epochen. Die Identitäts-Funktion sorgt bei kleinen Netzen für die größten Lernerfolge, unterliegt allerdings bei großen Netzen. Entgegen den Erwartungen, ist die parametrisierte ReLU-Funktion, außer bei kleinen Netzen, keine Verbesserung zum Original.

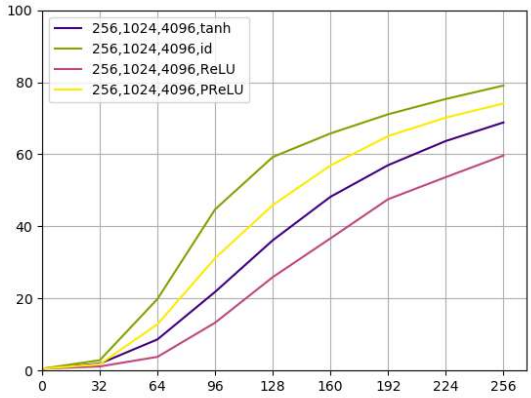

Abbildung 13: $\sigma \sim$ kleines Netz.
Konfidenz (Y-Achse) über 256 Epochen (X-Achse).

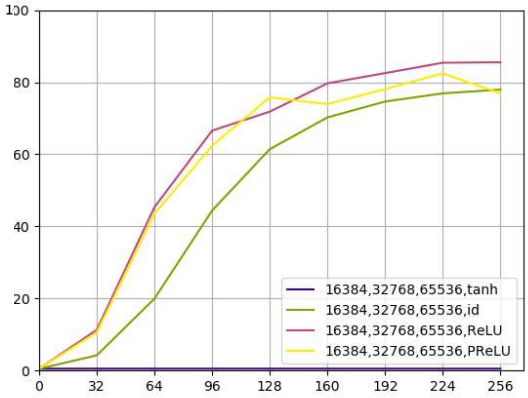

Abbildung 14: $\sigma \sim$ großes Netz.
Konfidenz (Y-Achse) über 256 Epochen (X-Achse).

Als letzten Hyperparameter gilt es, die Ausfallwahrscheinlichkeit dp zu betrachten, die unabhängig von der Aktivierungsfunktion bei einem kleinen Netz (vgl. Abbildung 15) für die Werte 0, 0.1, 0.25 und 0.5 immer schlechtere Lernergebnisse hervorbringt. Bei einem größeren Netz (vgl. Abbildung 16) wirkt sich die Ausfallquote nicht mehr so stark auf die Lernerfolge aus, es ist dennoch erkennbar, dass $dp = 0$ für dieses Problem die besten Ergebnisse liefert.

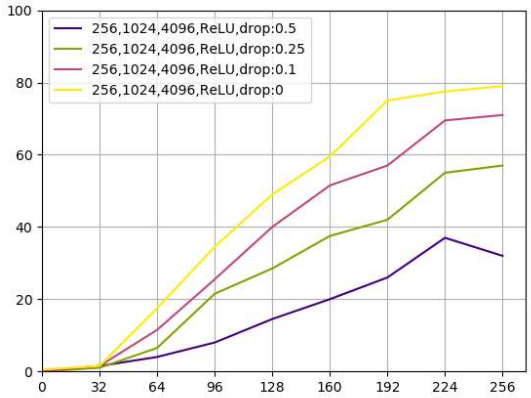

Abbildung 15: $dp \sim$ kleines Netz.
Präzision (Y-Achse) über 256 Epochen (X-Achse).

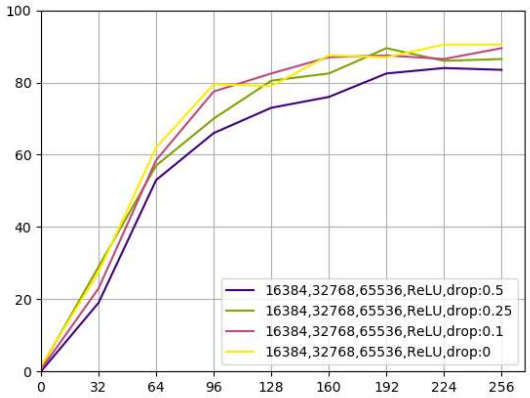

Abbildung 16: $dp \sim$ großes Netz.
Präzision (Y-Achse) über 256 Epochen (X-Achse).

5.2 Effektive Architekturen

Kombiniert mal alle oben vorgestellten Erkenntnisse, liegt nahe, dass ein Netz mit $k \geq 8$, zwei verdeckten Schichten mit einer großen Anzahl Knoten, der ReLU-Aktivierungsfunktion, einer Ausfallwahrscheinlichkeit von 0% und einer Lernrate von 0.001 das optimale Netz für die Lösung dieses Problems ist. Unter den stärksten zehn Netze, treten jedoch verschiedenste Variationen auf:

- Zwei Netze mit nur einer verdeckten Schicht.
- Ein kleines Netz mit $k = 6$.
- Drei Netze mit $\tanh$ und ein Netz mit id.
- Ein Netz mit einer Ausfallwahrscheinlichkeit von 10%.
- Drei Netze mit $\mu = 0.01$, ein Netz mit $\mu = 0.0001$.

Nach der dritten Phase werden alle zehn Netze mit 20 statt vier Validierungsproben jedes Typen getestet. Dabei ergeben sich die Intervalle [99.78, 99.95] für die Top-5-Präzisionen, [88.16, 92.85] für die Konfidenzen, [91.43, 94.63] für die Präzisionen und das daraus resultierende Intervall [94.69, 96.72] der Stärken.
Das Netz α = `16384_32768_65536_tanh_drop_0_lr_0.0001` ist nach 640 Epochen mit einer Stärke von 0.5 · 99.88 + 0.3 · 92.85 + 0.2 · 94.63 = 96.72 das effektivste KNN. An zweiter Stelle steht das Netz `1024_32768_65536_tanh_drop_0_lr_0.001`, das die geringere Größe mit einer höheren Lernrate kompensiert, und so eine Stärke von 96.43 erreicht. Das durch die einzelnen Hyperparametern favorisierte Netz `16384_32768_65536_ReLU_drop_0_lr_0.001` unterliegt deutlich (Stärke = 95.3).

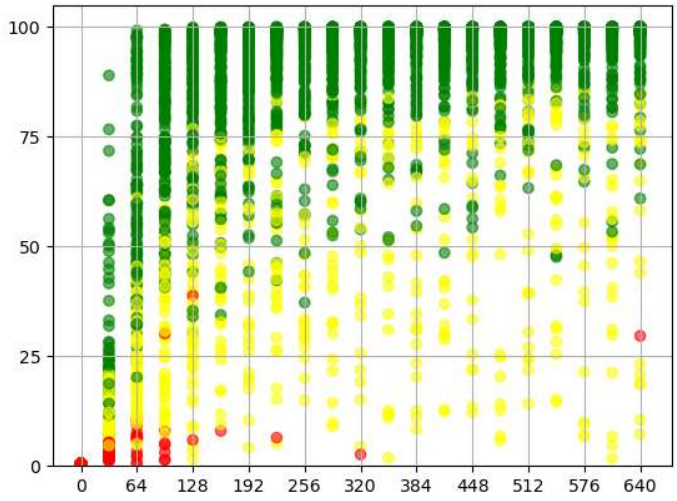

Abbildung 17: Die Entwicklung des Netzes α.
Durchschnittliche Konfidenz (Y-Achse) über 640 Epochen (X-Achse).

Die Abbildungen 17 zeigt die Entwicklung des Netzes α über 640 Epochen. Jeder Punkt in einem vertikalen Streifen repräsentiert die durchschnittliche Konfidenz des Netzes in dieser Epoche für je einen der 200 Typen.
Ein grüner Punkt gibt an, dass alle 20 Validierungsproben richtig klassifiziert wurden. Ein gelber Punkt bedeutet, dass bei mindestens einer der Proben der korrekte Typ nur unter den 5 wahrscheinlichsten Typen angegeben wurde. War bei mindestens einer der 20 Proben der korrekte Typ nicht unter den 5 wahrscheinlichsten Typen, ist der Punkt rot, was bedeutet, dass das Netz ein nicht zufriedenstellendes Ergebnis geliefert hat.
Die größten Verbesserungen erfolgen in den ersten 160 Epochen, nach denen nur vereinzelt falsch klassifizierte Proben auftreten. Das Netz erzielt bis Epoche 640 stetig Lernerfolge, sodass es sich zum stärksten KNN entwickelt.

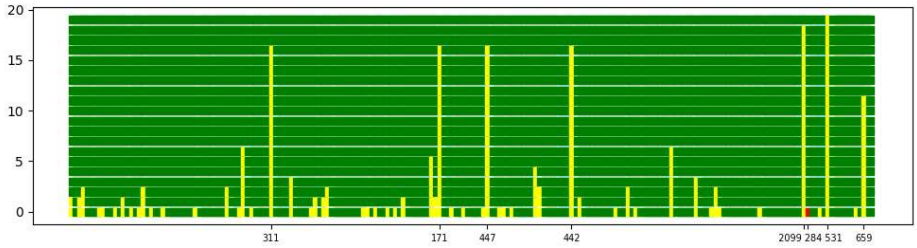

Abbildung 18: Übersicht Netz α nach 640 Epochen.
Y: Farben für die Bewertungen aller 20 Validierungsproben (vgl. Abb. 17).
X: Typen nach Häufigkeit sortiert.

Die Abbildung 18 stellt dar, wie oft jeder der 200 Typen in dem Validierungsdurchgang mit 20 Proben pro Typ richtig klassifiziert wurde. Es wird deutlich, dass das Netz α vorallem die häufigsten Typen sehr zuverlässig klassifiziert und nur den seltenen Typen t284 nicht immer unter den fünf Wahrscheinlichsten angibt. Dementsprechend sind 94.63% der Fläche grün, 99.88% - 94.63% = 5.25% gelb und 0.12% rot.
An der X-Achse sind die von Netz α nicht gut klassifizierten Typen t311, t171, t447, t442, t2099, t284, t531, t659 eingetragen.

Für die acht Proben der Uniklinik Düsseldorf wurde der Typ t032 diagnostiziert, das Netz α sagt hingegen folgende Typen mit zugehörigen Konfidenzen voraus:

Probe	1. Typ	2. Typ	3. Typ	4. Typ	5. Typ
I	t022 (37.5%)	t294 (14.9%)	t136 (14.7%)	t025 (6.8%)	t009 (6.4%)
II	t022 (26.7%)	t294 (22.2%)	t025 (11.3%)	t041 (11.0%)	t214 (7.9%)
III	t022 (34.6%)	t294 (18.0%)	t136 (13.0%)	t025 (7.5%)	t041 (6.7%)
IV	t022 (23.3%)	t294 (20.0%)	t136 (15.4%)	t025 (11.9%)	t041 (9.5%)
V	t022 (39.4%)	t136 (13.7%)	t294 (13.4%)	t009 (8.3%)	t025 (6.0%)
VI	t022 (43.5%)	t136 (12.7%)	t294 (11.8%)	t009 (7.8%)	t041 (7.0%)
VII	t022 (32.8%)	t136 (19.2%)	t294 (14.8%)	t025 (6.9%)	t041 (6.2%)
VIII	t022 (32.3%)	t136 (16.5%)	t294 (11.2%)	t041 (10.3%)	t009 (9.6%)

Das Netz klassifiziert alle acht Proben als Typ t022, jedoch stets mit einer Konfidenz, die deutlich niedriger als der Durchschnitt bei den von wgsim generierten Daten ist. Durch die Betrachtung der Repeats aus denen die Genomsequenzen von t032, t022 und t294 bestehen, wird deren Ähnlichkeit offensichtlich. Der Typ t032 konsistiert aus den 15 Repeats von t022 mit einem zusätzlichen Repeat eingeschoben. Analog besteht t294 aus den 16 Repeats von t032 mit einem weiteren Repeat an der vierten Stelle.

6 Diskussion

6.1 Ergebnisse

Ein künstliches neuronales Netz kann also so trainiert werden kann, dass es 200 Methicillin-resistente Staphylococcus-aureus-Typen zuverlässig klassifiziert. Das Netz muss mindestens eine verdeckte Schicht enthalten, eine zweite Schicht bewirkt im Regelfall eine Verbesserung. Weiterhin erweisen sich der Adam Optimierer und die Verlustfunktion der Kreuz-Entropie als wirksame Komponenten.

Für die Lösung des vorliegenden Problems ist es jedoch wichtig, verschiedene Hyperparameter bei der Konstruierung der Netzwerkarchitektur in Betracht zu ziehen. Besonders die Lernrate μ muss variiert werden, denn die drei stärksten Netze wurden mit jeweils unterschiedlichen Lernraten erzeugt. Von einer Ausfallwahrscheinlichkeit > 10% ist abzuraten, da diese Netze nicht für die Verarbeitung der vorliegenden Daten geeignet sind. Trotz der Erkenntnis, dass ein größeres k, und damit eine umfangreichere Eingabeschicht die Effektivität des Netzes steigert, sollten Varianten mit verdeckten Schichten verschiedener Größen getestet werden. Insbesondere die letzte verdeckte Schicht erzielt auch mit einer geringen Anzahl Knoten bemerkenswerte Lernerfolge.
Das stärkste Netz klassifiziert die acht Proben des Uniklinikums Düsseldorf nicht entsprechend den diagnostizierten Typen. Abgesehen davon, lassen die hohe Präzision von 94.9% und die Top-5-Präzision von 99.9% darauf schließen, dass es möglich ist, ein Netz auf mehr als 200 MRSA-Typen zu trainieren.

6.2 Alternative Ansätze

Mehr Ressourcen gewährleisten, dass größere Netz mit $k \geq 10$ konstruiert werden können, von denen noch höhere Lernerfolge zu erwarten sind. Um auch bei limitierten Ressourcen den Effekt von einem höheren k zu erreichen, besteht die Möglichkeit, die Liste der k-Mere zu trimmen. Dafür müssen alle möglichen k-Mere aus den Trainingsdaten erfasst werden, damit bestimmt werden kann, welche Eingabeknoten notwendig sind und welche entfernt werden können. Wurde die Anzahl der Eingabeknoten reduziert, können die k-Mere auf die kleinere Eingabe abgebildet und so der Speicherbedarf und der Rechenaufwand reduziert werden.
Durch einen k-Mer-Ansatz gehen Informationen über die Reihenfolge der Basen in einer Genomsequenz verloren, diese ist jedoch determinierend für die Expression der Gene. Kann eine Konsensus-Sequenz durch ein Alignment generiert werden, besteht die Möglichkeit, diese sequentiell in ein rekurrentes Netz einzugeben. Da jedoch nicht immer ein eindeutiges, wahrheitsgetreues Alignment gefunden werden kann, ist die Eingabe möglicherweise verzerrt. Die Varianz der Eingabe kann dadurch den Toleranzbereich eines Netzes übertreffen, sodass dieses nicht lernen kann.
Ein faltendes Netz eignet sich durch die Erkennung von Mustern ausgezeichnet für die Klassifikation von Bildern. Wird eine effektive Methode gefunden, um die sequenzierten Reads als Bild darzustellen, wäre ein *convolutional network* eine weitere geeignete Netzarchitektur für dieses Problem.
Statt die Architektur der Netzes zu variieren, kann die Datengrundlage angepasst werden. Da die Ursprungsländer der MRSA-Typen auf dem Ridom SpaServer zum Großteil erfasst werden, können für verschiedene Regionen jeweils eigene Netze trainiert werden. Dies erhöht die Relevanz der aufgenommen Typen und verringert gleichzeitig die Anzahl der benötigten Klassen. Zu kritisieren ist die Verlässlichkeit der Daten auf dem SpaServer, da diese von anonymen Beiträgen stammen.

Literatur

[1] *MRSA Allgemein*. URL: https://mrsa-net.nl/de/personal/mrsa-allgemein-personal (besucht am 28.06.2019).

[2] Prof. Dr. Wolfgang Witte. *spa-Typisierung als Sequenz-basiertes Typisierverfahren*. URL: https://www.rki.de/DE/Content/Infekt/NRZ/Staphylokokken/hinweise/spa-Typisierung.pdf (besucht am 28.06.2019).

[3] *Ridom SpaServer*. URL: URL:https://www.spaserver.ridom.de (besucht am 29.06.2019).

[4] *Ridom bioinformatics*. URL: https://www.ridom.de/staphtype (besucht am 29.06.2019).

[5] Stefan Luber. *Was ist ein Neuronales Netz?*. URL: https://www.bigdata-insider.de/was-ist-ein-neuronales-netz-a-686185 (besucht am 30.06.2019).

[6] Tomaso Poggio et al. "Why and When Can Deep-but Not Shallow-networks Avoid the Curse of Dimensionality: A Review". In *International Journal of Automation and Computing* (2017). DOI: 10.1007/s11633-017-1054-2 (besucht am 30.06.2019).

[7] Jeff Heaton. "Introduction to Neural Networks for Java, Second Edition". Heaton Research Inc., 2008, S. 158.

[8] Julian Moeser. *Künstliche neuronale Netze - Aufbau & Funktion*. URL: https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-29 (besucht am 30.06.2019).

[9] *ReLU*. URL: https://www.tinymind.com/learn/terms/relu (besucht am 05.07.2019).

[10] *Neural Network Bias: Bias Neuron, Overfitting and Underfitting*. URL: https://missinglink.ai/guides/neural-network-concepts/neural-network-bias-bias-neuron-overfitting-underfitting (besucht am 05.07.2019).

[11] Anirudh VK. "Google Activates 'Dropout' Patent, Neural Network Engineers To Be On Alert?". In *Analytics India Magazine* (2019). URL: https://www.analyticsindiamag.com/google-activates-dropout-patent-neural-network-engineers-to-be-on-alert (besucht am 14.07.2019).

[12] Jason Brownlee. *How to Choose Loss Functions When Training Deep Learning Neural Networks*. URL: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks (besucht am 06.07.2019).

[13] *Loss Functions*. URL: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html (besucht am 06.07.2019).

[14] Andrej Karpathy et al. "CS231n Convolutional Neural Networks for Visual Recognition". Stanford University. 2019. Lecture. URL: http://cs231n.github.io (besucht am 06.07.2019).

[15] Ian Goodfellow, Yoshua Bengio and Aaron Courville. "Deep Learning". *MIT Press* (2016). URL: http://www.deeplearningbook.org (besucht am 06.07.2019).

[16] Kaiming He et. al "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In *CoRR* (2015). URL: arXiv:1502.01852 (besucht am 12.07.2019).

[17] *Snakemake - a pythonic workflow system*. URL: https://bitbucket.org/snakemake/snakemake/src/master (besucht am 05.05.2019).

[18] *Snakemake*. URL: https://snakemake.readthedocs.io/en/stable (besucht am 05.05.2019).

[19] *wgsim*. URL: https://github.com/lh3/wgsim (besucht am 06.05.2019).

[20] *2014 NGS Field Guide – Table 3c – Error rates*. URL: http://www.molecularecologist.com/next-gen-table-3c-2014 (besucht am 06.05.2019).

[21] *pytorch.org*. URL: https://pytorch.org (besucht am 06.05.2019).

[22] *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone (besucht am 20.05.2019).

[23] Jason Brownlee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning (besucht am 15.06.2019).

[24] *The Number of Hidden Layers*. URL: https://www.heatonresearch.com/2017/06/01/hidden-layers.html (besucht am 10.07.2019).