UNIKASSEL VERSITÄT | ELEKTROTECHNIK INFORMATIK

Auf dem Weg zum künstlichen Lehrassistenten: Das Lernen von Bewertungsschemata für endliche Automaten mit GNNs

BACHELORARBEIT

zur Erlangung des Grades eines Bachelor of Science
im Fachbereich Theoretische Informatik/Formale Methoden
der Universität Kassel

Eingreicht von: Georg Siebert

Vorgelegt im: Fachbereich Theoretische Infromatik/Formale Methoden

Gutachter: Prof. Dr. Martin Lange
Prof. Dr. rer. nat. Bernhard Sick

Betreuer: M.Sc. Marco Sälzer

eingreicht am: 25. August 2021

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den nach der Prüfungsordnung der Universität Kassel zulässigen Hilfsmitteln angefertigt habe. Die verwendete Literatur ist im Literaturverzeichnis angegeben. Wörtlich oder sinngemäß übernommene Inhalte habe ich als solche kenntlich gemacht. Ebenfalls unterscheidet sich die digitale Abgabe nicht von der schriftlichen Abgabe.

Ort, Datum, Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Ein essentieller Bestandteil jeder Veranstaltung in einem Informatikstudium ist das eigenständige Lernen. Dort ist es vor allem wichtig, die gelernten Techniken mit Übungen zu bestätigen und zu vertiefen. Eine Möglichkeit, ein automatisiertes Feedback zu erhalten, sind Software-Lerntools, welche auch in Lehrveranstaltungen der Theoretischen Informatik genutzt werden. Erfolgreiche Umsetzungen von Lerntools sind z.B. der *Automata Tutor* [6] im Bereich der Automatentheorie oder die Webanwendungen *ILTIS* [9] und *DiMo* [14] für Probleme in der Logik. Andere Tools wie [8] bauen hingegen auf den *Intelligent Tutoring Systems* auf, welche ein sofortiges und personalisiertes Feedback zu den Eingaben des Benutzers geben. Eine Herausforderung bei solchen Tools ist es, die Fehler in den Lösungen der Studierenden zu finden und ein entsprechendes Feedback zu geben.

In dieser Arbeit wird untersucht, ob ein Neuronales Netz das Bewertungsschema eines menschlichen Korrektors für eine Aufgabe aus der Theoretischen Informatik erlernen kann. Die Aufgabenstellung ist dabei, einen nichtdeterministischen endlichen Automaten (NFA) anzugeben, welcher eine gegebene Sprache beschreibt. Erlernt ein Neuronales Netz dieses Bewertungsschema, kann dieses zu einer Lösung dieser Aufgabe ein automatisiertes Feedback gegeben werden. Ein Neuronales Netz lernt auf Basis der korrigierten Lösungen der Studierenden eine Funktion, welche das Bewertungsschema approximiert. Um ein solches Bewertungsschema mit neuronalen Netzen zu lernen, wird zunächst aus diesen eine Klassifikationsaufgabe abgeleitet. Eine Problematik ist dabei, die Eingabe eines NFA mit einem Neuronalen Netz zu verarbeiten. Eine Lösung dafür sind die Graph Neuronalen Netze, da diese auf Graphen arbeiten und ein NFA als Graph dargestellt werden kann. Als Grundlage zum Trainieren eines Graph Neuronalen Netzes, werden 178 Abgaben von Studierenden aus der Veranstaltung *Formale Sprachen und Logik* verwendet. Diese wurden zunächst aufbereitet und anschließend in einer für diese Arbeit entwickelten Weboberfläche gelabelt. Auf Basis dieser gelabelten Daten sind 2 Modelle von Graph Neuronalen Netzen trainiert und optimiert worden.

In Abschnitt 2 werden die notwendigen Grundlagen aus den Bereichen der endlichen Automaten, dem Machine Learning, der Neuronalen Netze sowie der Graph Neuronalen Netze vorgestellt. Wie das Bewertungschema in eine Klassifikationsaufgabe bzw. einzelne Klassen überführt wird und wie sich die Kriterien für diese zusammensetzen, ist in Abschnitt 3 erläutert. Anschließend wird in Abschnitt 4 die Verarbeitung der Hausaufgaben und die Weboberfläche zum Labeln dieser kurz dargestellt. Abschnitt 5 umfasst das Setup der Experimente. Dies umfasst, wie die Daten vorverarbeitet werden, wie die Modelle der Graph Neuronalen Netzwerke umgesetzt sind, wie ein Training abläuft und wie ein Netz abschließend getestet wird. Die Ergebnisse der Experimente sind in Abschnitt 6 aufgelistet und analysiert. Abschließend werden die Ergebnisse in Abschnitt 7 ausgewertet und in Abschnitt 8 wird ein Fazit gezogen sowie ein Ausblick gegeben.

2 Grundlagen

Um ein Bewertungsschema für endliche Automaten mit Techniken des *Machine Learning* zu erlernen, werden Kenntnisse aus der theoretischen Informatik und des maschinellen Lernens benötigt.

In diesem Kapitel werden die notwendigen Grundlagen erläutert. Den Anfang bilden die *endlichen Automaten*. Anschließend werden im notwendigen Umfang Techniken aus dem *Machine Learning* und *Neuronalen Netze* vorgestellt. Abschließend werden die *Graph Neuronalen Netze* eingeführt und aufgezeigt, warum diese anstatt von „klassischen“ tiefen Neuronalen Netzen genutzt werden.

2.1 Endliche Automaten

Alphabet, Wörter und Sprache Ein *Alphabet* Σ ist eine endliche, nicht leere Menge von Symbolen. Ein *Wort* $\mathtt{w}$ über Σ ist eine endliche Folge von Symbolen $\mathtt{a}_1 \dots \mathtt{a}_n$, für die gilt, dass $\mathtt{a}_i \in \Sigma$ mit $\mathtt{i} = 1, \dots, \mathtt{n}$. Dann ist Σ^* die *Menge aller Wörter*, welche sich über dem Alphabet Σ bilden lassen. Eine Menge $\mathtt{L} \subseteq \Sigma^*$ wird als Sprache bezeichnet.

Nichtdeterministischer endlicher Automat Ein *nichtdeterministischer endlicher Automat* sei Σ ein Alphabet. $\mathtt{M}$ (*NFA*) ist ein 5-Tupel $\mathtt{M} = (\mathtt{Z}, \Sigma, \delta, \mathtt{S}, \mathtt{E})$. Dabei ist $\mathtt{Z}$ die Menge der Zustände. Die Transitionsrelation $\delta \subseteq \mathtt{Z} \times \Sigma \times \mathtt{Z}$ beschreibt die möglichen Transitionen des *NFA*. Abschließend ist $\mathtt{S} \subseteq \mathtt{Z}$ die Menge der Startzustände und $\mathtt{E} \subseteq \mathtt{Z}$ die Menge der Endzustände.

Lauf, Akzeptierender Lauf, Sprache eines *NFA* Gegeben sei ein *NFA* $\mathtt{M} = (\mathtt{Z}, \Sigma, \delta, \mathtt{S}, \mathtt{E})$. Ein Lauf $\mathtt{l}$ von M auf einem Wort $\mathtt{w} = \mathtt{a}_1 \dots \mathtt{a}_n \in \Sigma^*$ ist eine Folge von Zuständen $\mathtt{q}_1\mathtt{q}_2 \dots \mathtt{q}_{n+1}$, sodass $\mathtt{q}_1 \in \mathtt{S}$ und $(\mathtt{q}_i, \mathtt{a}_i, \mathtt{q}_{i+1}) \in \delta$ für alle $1 \leq \mathtt{i} \leq \mathtt{n}$ gilt. Ist $\mathtt{q}_n \in \mathtt{E}$, dann ist $\mathtt{l}$ ein *akzeptierender Lauf* und $\mathtt{w}$ wird von $\mathtt{M}$ akzeptiert. Die Sprache von $\mathtt{M}$ ist die Menge $\mathtt{L} = \{\mathtt{w} \in \Sigma^* \mid \mathtt{w}$ wird von $\mathtt{M}$ akzeptiert$\}$.

Darstellung eines NFA als Graph Ein *NFA* kann als *beschrifteter gerichteter Graph* dargestellt werden. Dabei entspricht $\mathtt{Z}$ der Menge der Knoten. Ein Knoten wird als Endzustand beschriftet, wenn der entsprechende Zustand $\mathtt{z} \in \mathtt{E}$ ist. Das gleiche gilt auch für den Startzustand. Ein Knoten wird als Startzustand gekennzeichnet, wenn $\mathtt{z} \in \mathtt{s}$ gilt. Die Kanten des Graphen werden aus der Transitionsrelation δ gebildet. Einer Transition $(\mathtt{z}_1, \mathtt{a}, \mathtt{z}_2)$ entspricht die Kante von $\mathtt{z}_1$ nach $\mathtt{z}_2$, welche mit $\mathtt{a}$ beschriftet ist. Ein Beispiel ist in Abbildung 1 zu sehen.

Für weitere Informationen zu endlichen Automaten kann das Werk [22] verwendet werden.

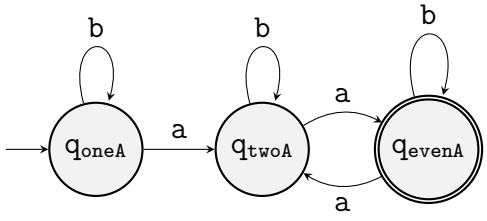


Abbildung 1: Beispiel eines NFA als Graph für die Sprache, dessen Wörter alle eine gerade Anzahl von $\mathtt{a}$ enthalten. Der Startzustand ist durch einen eingehenden Pfeil ohne Startknoten, gekennzeichnet (siehe $\mathtt{q}_{oneA}$). Alle Knoten, welche einen Endzustand darstellen, sind mit einem doppelten Rand dargestellt (siehe $\mathtt{q}_{evenA}$).

2.2 Grundlagen des Machine Learning

Machine Learning (*ML*) befasst sich mit dem Erstellen von statistischen Modellen auf Basis von Daten. In den folgenden Abschnitten werden nur die notwendigsten Techniken und Begriffe für die Arbeit aus dem Bereich des *ML* eingeführt. Als Basis für diesen Abschnitt dienen die Werke [1, 17, 7], welche auch für einen tieferen Einblick verwendet werden können.

Datensatz, Datenpunkt, Merkmal Ein *Datensatz* ist eine Menge von Datenpunkten. Ein *Datenpunkt* ist ein Vektor von Merkmalen. Ein *Merkmal* kann ein numerischer Wert oder ein String sein.

Gelabelter Datensatz Ein *gelabelter Datensatz* ist ein Tupel $\mathtt{D} = (\mathtt{X}, \mathtt{T})$. Dabei ist $\mathtt{X}$ der Datensatz und $\mathtt{T} = \{\mathtt{t}_1, \ldots, \mathtt{t}_n\}$ die Menge der Zielwerte. Für jedes Element $\mathtt{x}_i \in \mathtt{X}$ gibt es einen zugehörigen Zielwert $\mathtt{t}_i \in \mathtt{T}$, welcher auch als Label bezeichnet wird.

Kategorische Merkmale Bei einem *kategorischen Merkmal* nimmt das Merkmal einen Wert aus einer endlichen Menge $\mathtt{K} = \{\mathtt{k}_1, \ldots, \mathtt{k}_n\}$ an. Die Werte der Mengen stehen dabei nicht zueinander in Relation oder einer Ordnung und sind Strings oder numerische Werte. Ein Wert aus $\mathtt{K}$ steht dabei z.B. für eine Kategorie oder einen Typ. Ein Beispiel ist in der linken Tabelle von Abbildung 2 mit dem Merkmal *Art* gegeben.

One-Hot Eine Technik im Bereich der Vorverarbeitung von Datensätzen ist das *One-Hot* Encoding. Diese wird bei kategorischen Merkmalen verwendet. Sei $\mathtt{D}$ ein Datensatz und das $\mathtt{i}$-te Merkmal ist kategorisch mit dem Wertebereich $\{\mathtt{k}_1, \ldots, \mathtt{k}_n\}$, dann füge für jede Kategorie $\mathtt{k}_i$ jedem Datenpunkt aus $\mathtt{D}$ ein Merkmal hinzu mit dem Wert 1, falls $\mathtt{x}_i = \mathtt{k}_i$, ansonsten 0. Ein Beispiel ist in Abbildung 2 zu finden.

Name	Art
Spock	Katze
Nero	Hund
Kirk	Schildkröte
Kahn	Katze

Name	Katze	Hund	Schildkröte
Spock	1	0	0
Nero	0	1	0
Kirk	0	0	1
Kahn	1	0	0

Abbildung 2: Anwendung des *One-Hot* Encoding auf das Merkmal *Art*. Dabei ist links der Datensatz zu sehen, vor der Anwendung des *One-Hot* Encoding und rechts ist der Datensatz nach dessen Anwendung abgebildet.

Klassifikationsaufgabe Die *Klassifikation* arbeitet auf einem gelabelten Datensatz $\mathtt{D} = (\mathtt{X}, \mathtt{T})$, wobei alle $\mathtt{t}_i \in \mathtt{T}$ aus dem Wertebereich $\{1, \ldots, \mathtt{k}\}$ und $\mathtt{X} \subseteq \mathbb{R}^n$ sind. Das Ziel der *Klassifikation* ist es eine Funktion $\mathbb{R}^n \to \{1, \ldots, \mathtt{k}\}$ zu finden, welche einem Datenpunkt $\mathtt{x}_i \in \mathtt{X}$ sein Label $\mathtt{t}_i$ zuordnet. Neben der Klassifikation gibt es noch weitere Problemarten im *ML*. Diese können in [11] nachgelesen werden.

Prädiktion, korrekte Prädiktion, Accuracy, Precision, Recall Gegeben ist ein gelabelter Datensatz $\mathtt{D}$ über den Klassen $1, \ldots, \mathtt{k}$ und eine Klassifikation mit der Funktion $\mathtt{f} : \mathbb{R}^{\mathtt{n}} \to \{1, \ldots, \mathtt{k}\}$. Eine *Prädiktion* y_i ist die Anwendung von $\mathtt{f}$ auf einen Datenpunkt $\mathtt{x_i} \in \mathtt{X}$. Gilt $\mathtt{y_i} = \mathtt{t_i}$, wird die *Prädiktion* als *korrekte Prädiktion* bezeichnet. Die *Accuracy* ist eine Metrik, welche das Verhältnis der korrekten Prädiktionen zu allen Prädiktionen beschreibt. Diese wird durch die Vorschrift $\mathtt{Accuracy} = \frac{\sum_{\mathtt{n}=1}^{\mathtt{k}} \mathtt{n_{positves}}}{|\mathtt{D}|}$ berechnet. Dabei ist $\mathtt{n_{positives}}$ die Anzahl der *korrekten Prädiktionen* für die Klasse $\mathtt{n}$ und $|\mathtt{D}|$ die Anzahl der Datenpunkte von $\mathtt{D}$. Die *Precision* beschreibt das Verhältnis der richtigen Prädiktionen der Klasse $\mathtt{n}$ zu allen Prädiktionen, welche als die Klasse $\mathtt{n}$ eingestuft wurden. Über $\mathtt{precision}^{(\mathtt{l})} = \frac{\mathtt{n_{positives}}}{\mathtt{n_{pred}}}$ wird die *Precision* für die Klasse $\mathtt{l}$ berechnet. $\mathtt{n_{pred}}$ ist die Anzahl der Prädiktionen, welche als Klasse $\mathtt{l}$ eingestuft wurden. Das Verhältnis der korrekten Prädiktionen einer Klasse $\mathtt{l}$ zu der eigentlich Anzahl der Datenpunkte der Klasse $\mathtt{l}$ wird *Recall* genannt. Um den *Recall* für die Klasse $\mathtt{l}$ zu berechnen, wird $\mathtt{Recall}^{(\mathtt{l})} = \frac{\mathtt{n_{positives}}}{\mathtt{n_{actual}}}$ verwendet. Die Anzahl der Datenpunkte, welche der Klasse $\mathtt{l}$ auf dem Datensatz $\mathtt{D}$ angehören, ist $\mathtt{n_{actual}}$.

Gemittelter Recall und Precision Für eine Klassifikation mit $\mathtt{i}$ Klassen ist der *gemittelte Recall* gegeben durch $\mathtt{mRec} = \frac{1}{\mathtt{i}} \sum_{\mathtt{k}=1}^{\mathtt{i}} \mathtt{Recall_k}$. Dabei ist $\mathtt{Recall_k}$ der Recall der Klasse $\mathtt{k}$. Die *gemittelte Precision* $\mathtt{mPrec}$ wird mit $\frac{1}{\mathtt{i}} \sum_{\mathtt{k}=1}^{\mathtt{i}} \mathtt{Precision_k}$, wobei $\mathtt{Precision_k}$ die Precision der Klasse $\mathtt{k}$ ist.

2.3 Neuronale Netzwerke

In diesem Abschnitt werden die notwendigen Inhalte zu *Neuronalen Netzen (NN)* vorgestellt. Zuerst wird der Aufbau und die Funktionsweise von *NN* erläutert. Daraufhin werden die für diese Arbeit relevanten Aktivierungsfunktionen vorgestellt. Abschließend wird beschrieben, wie *NN´s* trainiert werden können, um Klassifikationsaufgaben zu lösen.

Neuron Der kleinste Baustein eines *NN*s sind die *Neuronen*. Ein Neuron berechnet eine Funktion $\mathbb{R}^{\mathtt{n}} \to \mathbb{R}$ mit $\mathtt{o}(\mathtt{x}) = \mathtt{a}(\sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{w_i x_i} + \mathtt{b})$. Dabei ist $\mathtt{w_i}$ das Gewicht der $\mathtt{i}$-ten Eingabe. Als b wird der Bias des Neuron bezeichnet und $\mathtt{a}$ ist eine nicht-lineare Funktion, genannt Aktivierungsfunktion.

(Feed Forward) Neuronal Network Ein *NN* setzt sich aus mehreren Neuronen zusammen, welche in Schichten $1, \ldots, \mathtt{L}$ angeordnet sind. Die Anzahl der Neuronen einer Schicht $\mathtt{l}$ ist gegeben durch $|\mathtt{l}|$. In einer Schicht wird von allen Neuronen die gleiche Aktivierungsfunktion $\mathtt{a}^{(\mathtt{l})}$ genutzt. In den aufeinander folgenden Schichten sind die Neuronen vollständig verbunden. Das heißt, ein Neuron aus der $\mathtt{l}$.-ten Schicht erhält alle Ausgaben der $(\mathtt{l}-1)$-ten Schicht als Eingabe. Dies ist in Abbildung 3 graphisch dargestellt. Für all diese Verbindungen existieren Gewichte. Das Gewicht für die Ausgabe des $\mathtt{j}$-ten Neurons der $(\mathtt{l}-1)$-ten Schicht an das $\mathtt{i}$-te Neuron der $\mathtt{l}$-ten Schicht ist durch $\mathtt{w}_{\mathtt{ji}}^{(\mathtt{l})}$ beschrieben. Der Bias für ein Neuron ist durch das Gewicht $\mathtt{w}_{\mathtt{0i}}^{(\mathtt{l})}$ festgelegt. Damit ist die Ausgabe des $\mathtt{i}$-ten Neurons der $\mathtt{l}$-ten Schicht gegeben durch $\mathtt{o}_{\mathtt{i}}^{(\mathtt{l})} = \mathtt{a}^{(\mathtt{l})}(\sum_{\mathtt{k}=1}^{|\mathtt{l}-1|} \mathtt{w}_{\mathtt{ki}}^{(\mathtt{l})} \mathtt{o}_{\mathtt{k}}^{(\mathtt{l}-1)} + \mathtt{w}_{\mathtt{0i}}^{(\mathtt{l})})$, wobei $\mathtt{o}_{\mathtt{k}}^{(\mathtt{l}-1)}$ die Ausgabe des $\mathtt{k}$-ten Neurons der $\mathtt{l}-1$-ten Schicht ist. Eine Ausnahme bildet die 1-te Schicht, welche auch als Eingabeschicht bezeichnet wird. Diese leitet den an das *NN* anliegenden Eingabevektor $\mathtt{x} = [\mathtt{x}_1, \ldots, \mathtt{x}_{|1|}]$ mittels der Identitätsfunktion an die nachfolgende

Schicht weiter. Damit ist $o_i^{(1)} = x_i$. Die L-te Schicht eines *NN*s ist die Ausgabeschicht und bestimmt den Ausgabevektor $y = [y_1, \ldots, y_{|L|}]$ des *NN*s. Die i-te Ausgabe des Netzes ist also $y_i = o_i^{(L)}$. Ein *NN* berechnet damit die Funktion $f : \mathbb{R}^{|1|} \to \mathbb{R}^{|L|}$. Alle Schichten zwischen der Eingabe- und der Ausgabeschicht werden als versteckte Schichten bezeichnet. Da die Eingabe durch das *NN* von hinten nach vorne propagiert wird, wird diese Art von *NN* auch als *Feed Forward Neuronal Networks* bezeichnet.

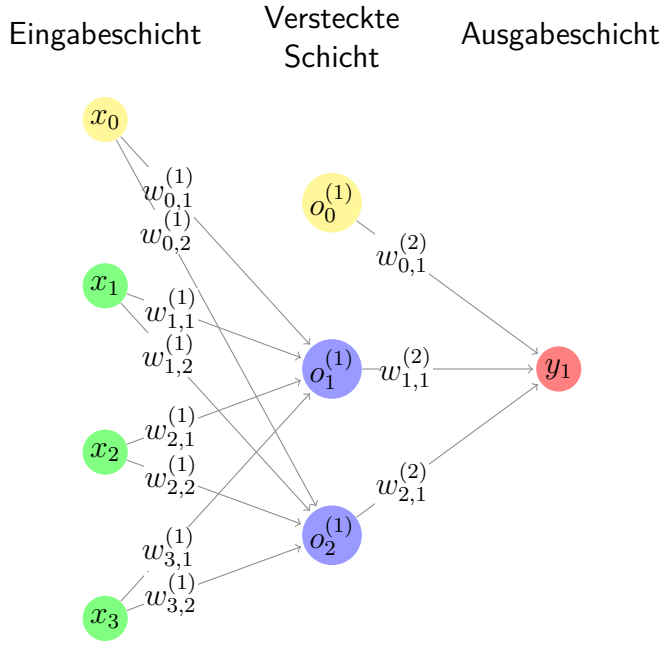


Abbildung 3: Darstellung eines 3-schichtigen *Feed Forward* Netzwerks, welches die Funktion $f : \mathbb{R}^3 \to \mathbb{R}$ berechnet. Die Neuronen in der Eingabeschicht sind grün, in der versteckten Schicht blau und in der Ausgabeschicht rot dargestellt. Der Bias für die darauf folgende Schicht ist mit den gelben Neuronen abgebildet.

Aktivierungsfunktionen Die Ausgabe eines Neurons wird über die Aktivierungsfunktion a bestimmt. Damit eine Funktion als Aktivierungsfunktion verwendet werden kann, sollte sie *nichtlinear* und *differenzierbar* sein. Die Differenzierbarkeit ist für den Lernprozess notwendig. Den Aspekt der *Nichtlinearität* ermöglicht dem Neuronalen Netz jede stetige Funktion zu approximieren. Mehr dazu kann in [18] nachgelesen werden. Für diese Arbeit sind die drei Funktionen *ReLU*, *Sigmoid* und *Softmax* von Interesse.

Sigmoid Die Zielwerte der *Sigmoid* Funktion $\sigma(x) = \frac{1}{1+e^{-x}}$ liegen in dem Intervall von $[0, 1]$. Dabei gehen negative Eingaben gegen 0 und positive Eingaben gegen 1, der Verlauf ist in Abbildung 5 zu finden. Die *Sigmoid* Funktion kann als eine Verteilung zweier Klassen oder als Zugehörigkeit zu einer Klasse interpretiert werden. Die Funktion σ wird für das Lösen binärer Klassifikationsaufgaben verwendet [12]. Typischerweise wird die σ deshalb im Kontext einer binären Klassifikationsaufgabe in der Ausgabeschicht verwendet.

ReLU Die *Rectified Linear Unit* (ReLU) $\texttt{ReLU}(x) = \max(0, x)$ ist eine nichtlinare Funktion. In den meisten Fällen ist die ReLU Funktion eine gute Wahl für die Aktivierungsfunktion [12, 4]. Einen Einstieg zu der ReLU Funktion ist unter [2] zu finden.

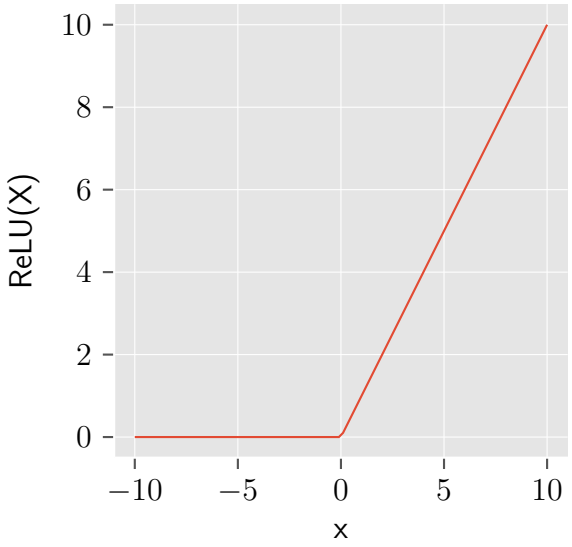


Abbildung 4: Ausschnitt des Funktionsgraphen der ReLU Funktion.

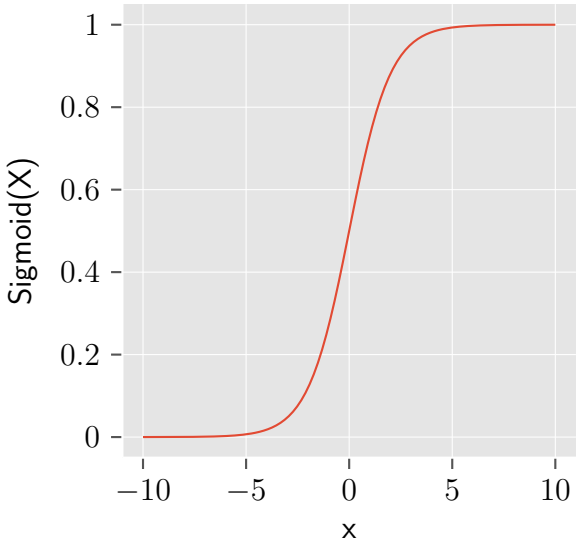


Abbildung 5: Ausschnitt des Funktionsgraphen der Sigmoid Funktion.

Softmax Die Funktion $\texttt{Softmax} : \mathbb{R}^n \to [0, 1]^n$ bildet das i-te Element eines Vektors $x \in \mathbb{R}^n$ auf $\frac{e^{x_i}}{\sum_{k=0}^{n} e^{x_k}}$ ab. Die Softmax Funktion wird bei einem NN als Aktivierungsfunktion ausschließlich in der Ausgabeschicht verwendet. Dann wird der Ausgabevektor $y = [y_1, \ldots, y_k]$ eines NNs nach Anwendung der Softmax Funktion als die Wahrscheinlichkeiten für die möglichen Zielklassen interpretiert. Dabei entspricht y_i der Wahrscheinlichkeit für die Klasse i. Damit eignet sich die Funktion für die Klassifikation mehrer Klassen [11]. Ein praktisches Beispiel ist unter [3] zu finden.

Trainingsprozedur neuronaler Netze Welche Funktion ein *NN* abbildet wird über dessen Gewichte $w_{ji}^{(1)}$ bestimmt. Diese werden bei der Initialisierung des *NN*s zufällig oder konkret festgelegt. Damit die gesuchte Funktion im Kontext einer Klassifikation approximiert werden kann, müssen die Gewichte auf Basis der Prädiktion und dem zugehörigen Zielwert angepasst werden. Das am häufigsten dafür verwendete Verfahren ist die Backpropagation. Dieses arbeitet mit einer Fehlerfunktion E, welche für die Aufgabenstellung entsprechend gewählt wird. Im Folgenden wird das Verfahren der Backpropagation beschrieben und die hier verwendete Fehlerfunktion der *Kreuzentropie* für eine Klassifikationsaufgabe vorgestellt.

Backpropagation Die Backpropagation ist ein iterativer Algorithmus, welcher die Gewichte eines NNs anpasst, sodass eine gegebene Fehlerfunktion E minimiert wird. Dieser kann damit zusammengefasst werden, dass als Erstes die anliegende Eingabe durch das Netz nach vorne propagiert, um die Ausgabe des NNs zu berechnen. Anschließend wird die Änderung des Gewichts

$w_{ji}^{(m)}$ über die Ableitung $\frac{\delta E}{\delta w_{ji}^{(m)}}$ auf Basis der Ausgabe und des erwarteten Zielwerts berechnet, was als *Backpropagation* bezeichnet wird. Dananch werden die Gewichte um ihre Änderung aktualisiert, wobei die Änderung mit der Lernrate η multipliziert wird. Die Lernrate bestimmt, wie stark die Änderung adaptiert wird. Sind die Gewichte angepasst, beginnt der Algorithmus von vorne mit der nächsten Eingabe. Eine ausführliche Beschreibung der *Backpropagation* kann in [1, 19] nachgeschlagen werden.

Kreuzentropie Die *Kreuzentropie* (*CE*) ist gegeben durch $CE(t, y) = -\sum_{i=1}^{k} t_i \cdot \log(y_i)$ für eine Klassifikation mit den Klassen $1, \ldots, k$. Dabei ist $t = [t_1, \ldots, t_n]$ der Vektor der Zielwerte und $y = [y_1, \ldots, y_n]$ der Vektor der Prädiktionen. Durch die *CE* wird der Unterschied der Verteilung der Prädiktionen und der gegebenen Zielverteilung bestimmt. Dabei wird ein t_i und y_i als die Wahrscheinlichkeit für die Klasse i interpretiert. Damit eignet sich die *CE* als Fehlerfunktion für eine Klassifikationsaufgabe. Weitere Informationen zur *CE* sind in [17] zu finden.

2.4 Graph Neuronale Netzwerke

Die *Graph Neuronalen Netzwerke* (*GNN*) sind ein Berechnungsmodell für Funktionen über Graphen. Zu Beginn werden die notwendigen Aspekte von Graphen eingeführt und anschließend ein Konzept für GNNs in der Klassifikation eingeführt. Zuletzt werden die zwei Modelle der *Message Passing Neural Networks* und der *Relational Graph Convolutional Networks* im Kontext von GNNs als Klassifikatoren vorgestellt. Die verwendeten Quellen für diesen Abschnitt sind [16, 32, 20, 21, 10, 30]. In diesen können auch weitere Informationen zu den Themenbereichen gefunden werden.

Graphen Ein *Graph* G ist das Tupel (V, E, f, e). Dabei ist $V = \{v_1, .., v_n\}$ die Menge der Knoten und $E \subseteq V^2$ die Menge der Kanten. Die Funktion $f : V \to \mathbb{R}^m$ ist die *Knotenfunktion*, welche einen Knoten auf seine zugehörigen Merkmale abbildet. Dabei ist m die Dimension eines Knotenmerkmals. Analog dazu ist die *Kantenfunktion* $e : E \to \mathbb{R}^o$ gegeben und bildet eine Kante auf ihre zugehörigen Merkmale ab. o ist die Dimension der Merkmale der Kanten. Die Nachbarschaft N_v eines Knoten v ist die Menge $\{u \in V \mid (u, v) \in E\}$. Die Menge der eingehenden Kanten eines Knotens v ist gegeben durch $C_v = \{(u, v) \in E \mid u \in V\}$.

GNN als Klassifikator Ein *GNN* berechnet eine Funktion, welche einen Graphen G nach $\mathbb{R}^k$ abbildet, wobei k die Anzahl der Klassen der Klassifikationsaufgabe sind. Der Aufbau eines *GNN*s als Klassifikator ist in Abbildung 6 dargestellt. Dieses besitzt $1, \ldots, L$ Schichten. In jeder Schicht l wird für jeden Knoten v ein Zustand $h_v^l \in \mathbb{R}^j$ berechnet und anschließend in eine Aktivierungsfunktion gegeben, wobei j die Dimension des Zustands der Schicht l ist. Nach der Berechnung der Zustände wird der Zustand des Graphen $h_G \in \mathbb{R}^o$ mit $\mathcal{R}(\{h_v^L \mid v \in V\})$ errechnet, wobei $\mathcal{R}$ die *Readout* Funktion ist und o die Dimension des Zustands des Graphen. $\mathcal{R}$ ist eine differenzierbare Funktion, wobei typische Funktionen für diese der Mittelwert der Zustände oder die Summe der Zustände sind. Die Berechnung von h_G wird auch als *Graph Readout* bezeichnet. Abschließend wird die Ausgabe $y \in \mathbb{R}^k$ des *GNN*s über ein Neuronales Netz mit der `Softmax` Funktion als Aktivierungsfunktion berechnet. Die Ausgabe des *GNN*s wird als die Wahrscheinlichkeiten der k Klassen interpretiert. Trainiert wird ein *GNN* mit dem Backpropagation Algorithmus, welcher alle anpassbaren Parameter optimiert. Einen tieferen Einblick in *GNN*s bieten [20, 32, 16, 30].

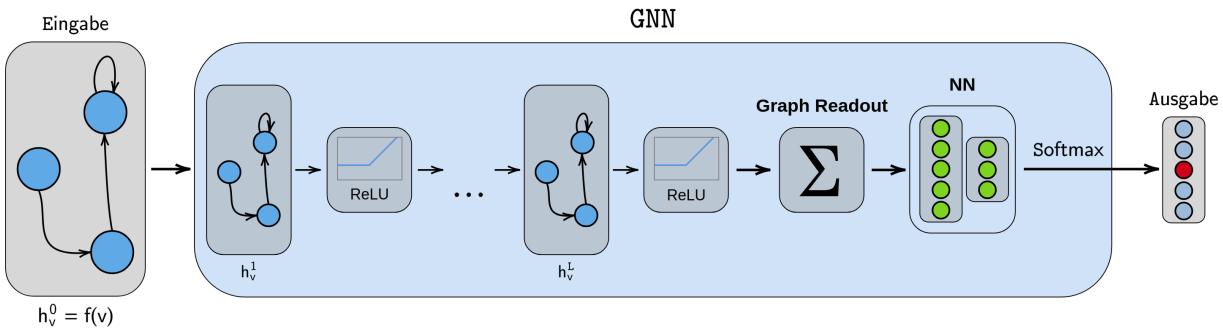


Abbildung 6: Der Aufbau eines *GNN*s als Klassifikator. Die Eingabe in das Netz ist ein Graph und wird über L Schichten verarbeitet. In jeder Schicht l wird für jeden Knoten v als Erstes deren Zustand h_v^l berechnet und als Zweites in eine Aktivierungsfunktion gegeben. Hier ist diese als ReLU Funktion dargestellt. Nachdem alle Schichten durchlaufen sind, wird der Zustand des Graphen h_G mittels der *Readout* Funktion gebildet. Der Zustand des Graphen wird darauf in ein NN mit der $\mathrm{Softmax}$ Funktion als Aktivierungsfunktion gegeben, welches die Ausgabe des *GNN*s bildet. Die Ausgabe des *GNN*s ist ein Vektor, dessen Einträge die Wahrscheinlichkeit der Klasse für die Eingabe beschreiben.

Message Passing Neural Networks Sei $G = (V, E, f, e)$ ein Graph, dann ist der Zustand h_v^0 des Knotens v eines *Message Passing Neural Network* (*MPNN*) die Merkmale des Knotens $f_v(v)$. Um den Zustand h_v^{l+1} des Knotens v für die Schicht $l+1$ zu berechnen, werden zunächst die Nachrichten an den Knoten m_v^{l+1} mit $\sum_{w \in N_v} M_l(h_v^l, h_w^l, e(v,w))$ berechnet. Dabei ist M_l eine differenzierbare Funktion, welche die *message function* der Schicht l ist. Abschließend ist h_v^{l+1} gegeben durch $h_v^{l+1} = U_l(h_v^l, m_v^{l+1})$, wobei U_l die *update function* der Schicht l ist. Die Funktion U_l ist ebenfalls eine differenzierbare Funktion. Der Vorgang zur Berechnung von h_v^L wird als *Message Passing* bezeichnet. Weitere Informationen zu *MPNN* sind unter [10] zu finden.

Relational Graph Convolutional Networks Für ein *Relational Graph Convolutional Network* (*R-GCN*) müssen die Kanten eines Graphen G in Relationstypen eingeteilt werden. Diese Relationstypen werden mit $1, \ldots, R$ bezeichnet. Ebenfalls wird für jeden Knoten eine Selbstkante mit dem Relationstypen 0 eingeführt, damit der Knoten Informationen über sich selbst lernen kann. Ein *R-GCN* berechnet h_v^{l+1} für einen Knoten v für die $l+1$-te Schicht mit $\sigma(\sum_{r \in R} \sum_{w \in N_v^r} \frac{1}{c_{v,r}} W_r^l h_w^l + W_0^l h_v^l)$. Dabei ist N_v^r die Menge der benachbarten Knoten von dem Knoten v, welche eine eingehende Kante mit dem Relationstyp r haben. W_r^l ist das Gewicht der Schicht l für die Relation r und $c_{v,r}$ die Normalisierungskonstante, welche problemspezifisch festgelegt wird. Die Funktion σ ist eine differenzierbare Aktivierungsfunktion, wobei für diese z.B. die ReLU Funktion verwendet wird. Das Modell der *R-GCN* kann in [21] nachgeschlagen werden.

3 Aufgabenkorrektur als Klassifikationsaufgabe

Das Ziel dieser Arbeit ist es, mittels eines GNN das Bewertungsschema von studentischen Abgaben aus dem Bereich der endlichen Automaten anhand eines gelabelten Datensatzes zu erlernen. In den folgenden Abschnitten wird die von den Studierenden bearbeitete Aufgabe, die Bewertung dieser Aufgabe in der händischen Korrektur und die daraus abgeleiteten Klassen des Bewertungsschemas für die Klassifikation vorgestellt.

Aufgabe zu endlichen Automaten Die Aufgabe wurde aus der Veranstaltung *Formale Sprachen und Logik* der Universität Kassel übernommen. Ziel der Aufgabe ist es, einen NFA zu einer gegebenen Sprache zu erstellen. In der Aufgabe soll für eine Sprache L ein NFA N angegeben werden, sodass $\mathtt{L} = \mathtt{L}(\mathtt{M})$ gilt. Dabei ist L die Menge $\{\mathtt{w} \in \{\mathtt{a}, \mathtt{b}\}^* \mid \mathtt{w} \text{ enthält das Wort abba nicht}\}$. Ein Beispiel eines NFAs, welches die Sprache L beschreibt, ist in Abbildung 7 zu finden.

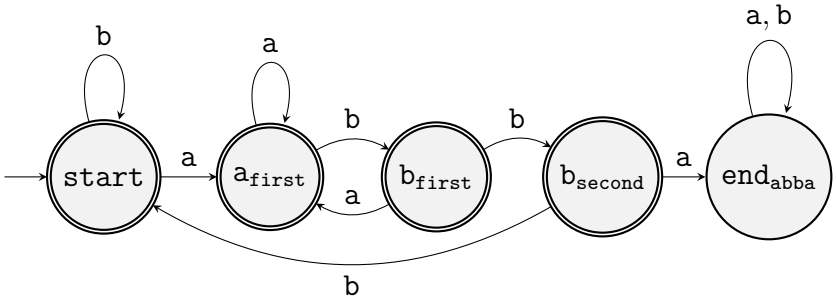


Abbildung 7: Ein NFA, welcher die Sprache $\{\mathtt{w} \in \{\mathtt{a}, \mathtt{b}\}^* \mid \mathtt{w} \text{ enthält das Wort abba nicht}\}$ beschreibt.

Händische Korrektur Wird ein Automat von Hand korrigiert, dann wird dieser auf zwei Aspekte kontrolliert. Es wird überprüft, ob die Sprache des Automaten mit der geforderten Sprache übereinstimmt und ob eine gewisse Strategie beim Erstellen der Automaten verfolgt wurde. Unter einer Strategie wird dabei verstanden, dass die gewählten Zustände des Automaten eine bestimmte Aufgabe haben. Dies wird am Besten an einem Beispiel deutlich, wobei der NFA aus Abbildung 7 verwendet wird. Wenn der NFA in dem Zustand $\mathtt{a}_{\mathtt{first}}$ ist, ist der zuletzt gelesene Buchstabe ein a gewesen, was den Anfang eines abba Teilworts darstellen kann. Ist das nächste Zeichen ein a wird im Zustand $\mathtt{a}_{\mathtt{first}}$ verblieben, da auch dieses a den Start von abba markieren kann. Wird ein b gelesen, dann geht der NFA in den Zustand $\mathtt{b}_{\mathtt{first}}$ über. Ist dieser Zustand erreicht, dann ist das erste b aus abba aufgetreten. Allgemein kann die Strategie so zusammengefasst werden, dass in den Zuständen der aktuell gelesene Teil vom Wort $abba$ gespeichert ist. Im Zustand $\mathtt{b}_{\mathtt{second}}$ enthält das in den NFA eingegebene Wort bereits das Teilwort abb. Ist der Zustand $\mathtt{end}_{\mathtt{abba}}$ erreicht, dann ist das Teilwort abba im Wort enthalten und der NFA akzeptiert das Wort nicht. Deswegen sind alle anderen Zustände auch Endzustände, da dort noch nicht das Teilwort abba enthalten ist.

Ein typischer Fehler, der Studierenden bei dieser Aufgabe unterläuft, ist dass Wörter nicht in der Sprache des NFA enthalten sind, welche in der gegebenen Sprache L enthalten sind. Würde zum Beispiel die Kante von $\mathtt{b}_{\mathtt{second}}$ nach start stattdessen eine Selbstkante zu b sein, dann

würden Wörter wie abbba nicht akzeptiert werden. Diese liegen aber offensichtlich in der Sprache. Diese Fälle werden häufig übersehen, da von Studierenden der Automat so konzipiert wird, dass dieser keine Wörter enthält, welche nicht in der Sprache liegen sollen. Eine weitere Art von Fehlern ist das Übersetzen von Schleifen, welche über mehrere Knoten hinweg entstehen. Würde der NFA z.B. eine weitere Kante von $\mathtt{a}_{\mathtt{first}}$ nach start mit a besitzen, akzeptiert der Automat das Wort $baabba$. Dieses enthält natürlich das Teilwort $abba$ und sollte eigentlich nicht in der Sprache des Automaten liegen. Eine Abgabe ist aber nicht komplett falsch, nur weil solche Fehler gemacht werden. Das heißt, dass es das wichtigste Kriterium für die Korrektur ist, ob eine entsprechende Strategie umgesetzt wurde, sodass der NFA die gegebene Sprache beschreibt. Dabei wird überprüft, ob kleinere oder größere Fehler auftreten, welche zuvor erläutert wurden. Angenommen, die Aufgabe wird mit 4 Punkten bewertet. Dann würden die Abgaben, welche sich aus den zwei beispielhaft beschriebenen Fehlern ergeben, mit 2 bis 3 Punkten bewertet werden. Erst wenn keine Strategie erkennbar ist und mehrere Fehler aufgetreten sind, gilt die Aufgabe als komplett falsch.

Die einzelnen Kriterien und auch die Art und Weise, wie ein Mensch solche Abgaben korrigiert in einem Programm umsetzt, scheinen schwer möglich. Dazu müsste das Programm erkennen, welche Fehler gemacht wurden und auch wie schwerwiegend diese sind. Ebenfalls müsste das Programm erkennen, ob eine Strategie vorhanden ist. Diese Kriterien sind nicht immer nach einem festen Schema auszumachen. Eine Alternative dazu stellen die NN dar. Diese können auf Basis der Abgaben das Bewertungsschema approximieren. Dafür muss die Aufgabenstellung als eine Klassifikationsaufgabe definiert werden. Dies wird im folgendem Abschnitt vorgestellt.

Aufgabenbewertung als Klassifikationsaufgabe Das Bewertungsschema wird mit 5 Klassen umgesetzt, welche in Abbildung 8 aufgelistet sind. Beschreibt ein NFA die gesuchte Sprache korrekt, dann liegt er entweder in Klasse 4 oder 5. Diese unterscheiden sich darin, dass die Automaten, welche in der Klasse 5 liegen, effizienter sind. Zu den Klassen 2 und 3 gehören alle NFA, welche die oben beschriebene Strategie verfolgen, aber kleinere oder größere Fehler enthalten. Alle Lösungen, welche keine Strategie verfolgen, werden der Klasse 1 zugeordnet. Um das Bewertungsschema der Klassen weiter zu verdeutlichen, ist im Folgenden zu allen Klassen ein Beispiel gegeben. Die Beispiele sind aus dem Datensatz entnommen.

- 1 - Die Strategie ist nicht erkennbar, der Automat ist falsch.
- 2 - Die Strategie wurde verfolgt, aber es wurden grobe Fehler gemacht.
- 3 - Die Strategie wurde verfolgt, aber es wurden kleine Fehler gemacht.
- 4 - Die Strategie wurde verfolgt, der Automat ist richtig aber ineffizient. Es wurden mehr Zustände verwendet als notwendig.
- 5 - Die Strategie wurde verfolgt, der Automat ist richtig. Es wurden nur die notwendige Anzahl an Zuständen verwendet.

Abbildung 8: Einteilung des Bewertungsschemas in Klassen.

- **Klasse 1** - Ein Beispiel für die Klasse 1 ist in der Abbildung 9 gegeben. Es fällt sofort auf, dass der Teil des Automaten, welcher mit dem Zustand q_5 beginnt, nicht zu erreichen ist. Dies lässt vermuten, dass der Studierende das Konzept der endlichen Automaten nicht vollständig verstanden hat. Der einzig erreichbare Endzustand ist q_4. Um diesen zu erreichen muss das eingegebene Wort das Teilwort aba oder das Teilwort abba enthalten. Es sind also Wörter in der Sprache des NFA enthalten, welche nicht in der eigentlichen Sprache enthalten sind. Die Zustände q_0 und q_4 ermöglichen es nur noch, dass eine beliebige Folge von a und b vor und nach den beiden Teilwörtern stehen kann. Der Automat bildet also nicht die gesucht Sprache ab. Des Weiteren ist es nicht erkenntlich, dass die Strategie verfolgt wird, da die Zustände nicht die einzelnen Bestandteile von abba erkennen, sondern die Sprache $\{w \in \{a, b\}^* \mid w \text{ enthält das Wort aba oder das Wort abba}\}$. Damit ist der Automat falsch und ist der Klasse 1 zugehörig.

- **Klasse 2** - In Abbildung 10 ist ein Beispielautomat für die Klasse 2 gegeben. Grundsätzlich ist zu erkennen, dass eine Strategie verfolgt wurde. Nachdem aber die Symbole ab gelesen wurden und der Automat im Zustand q_{prevent} steht, kann kein weiteres a akzeptiert werden. Dies sollte aber möglich sein. Des Weiteren würde der Automat auch das Wort aabba akzeptieren, welches nicht in der Sprache L liegen sollte. Das bedeutet, dass zwar die Strategie verfolgt wurde, aber grobe Fehler gemacht wurden.

- **Klasse 3** - Der Beispielautomat ist in Abbildung 11 zu sehen. Genau wie der Automat aus der Klasse 2 akzeptiert dieser das Wort aabba. Bis auf die Transition von q_2 mit a nach q_0 ist der Automat nah an der Musterlösung und verfolgt in der restlichen Struktur die richtige Strategie. Würde die Transition von (q_2, a, q_0) zu der Transition (q_0, a, q_0) und die Transition (q_3, b, q_3) zu (q_3, b, q_0) geändert werden, würde der NFA die Sprache L beschreiben. Der Studierende verfolgt hier also die richtige Strategie und hat nur kleine Fehler gemacht, womit diese Abgabe zur Klasse 3 gehört.

- **Klasse 4** - Alle Automaten der Klasse 4 bilden eine richtige Lösung der Aufgabe und die Sprache L ab. In dem Beispielautomaten (siehe Abbildung 12) kann lediglich der Zustand S mit dem Zustand B_0 verschmolzen werden. Dadurch würde der Automat mit einem Zustand weniger auskommen.

- **Klasse 5** - In der Klasse 5 sind alle Automaten enthalten, welche der Musterlösung entsprechen. Diese NFAs bestehen aus maximal 4 Zuständen und bilden die Sprache L ab. Ein solcher Automat ist in Abbildung 13 zu finden.

4 Von Abgaben endlicher Automaten zu einem gelabelten Datensatz

Der Ausgangspunkt der Arbeit war ein Datensatz von 179 studentischen Abgaben. Diese enthielten NFAs, welche in dem Format aus Abbildung 14 abgespeichert wurden. Um die Daten für eine Klassifikationsaufgabe nutzen zu können, müssen die Daten zunächst aufbereitet und in ein neues Format überführt werden. Des Weiteren müssen die Daten mit Labeln versehen werden, um

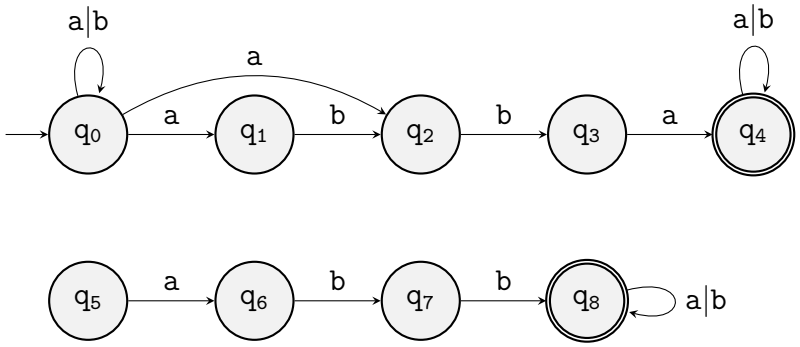


Abbildung 9: Ein NFA der keine Strategie verfolgt, weshalb dieser der Klasse 1 zugeordnet.

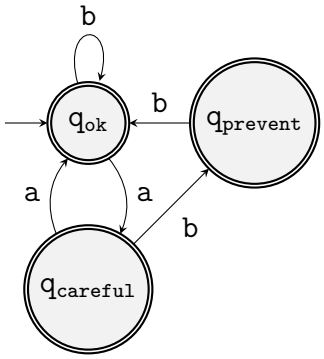


Abbildung 10: Ein Beispielautomat für die Klasse 2, da dieser Worter mit dem Teilwort abba akzeptiert und auch Wört nicht akzeptiert, welche abba nicht enthalten.

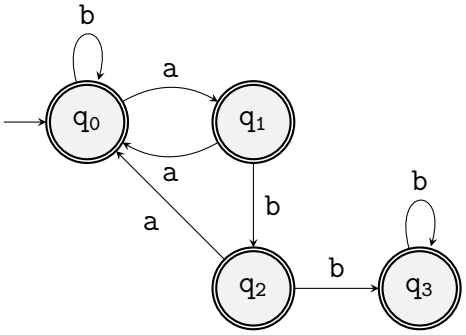


Abbildung 11: Dieser Automat gehört der Klasse 3 an, da dieser Wörter mit dem Teilwort abba akzeptiert. Der NFA verfolgt die Strategie und kann durch die zwei Änderungen $(q_2, a, q_0) \rightarrow (q_0, a, q_0)$ und $(q_3, b, q_3) \rightarrow (q_3, b, q_0)$ zu einer korrekten Lösung umgebaut werden.

einen gelabelten Datensatz zu erhalten. Das Vorgehen wird in diesem Abschnitt der Arbeit näher beleuchtet.

4.1 Aufbereiten der Abgaben

Die Abgaben der Studierenden liegen in dem Format vor, welches in Abbildung 14 zu sehen ist. Eine Abgabe enthält dabei einen NFA, welcher in die Bestandteile *input_alphabet* (Eingabealphabet), *start_states* (Startzustände), *transitions* (Transitionsrelation) und *acc_states* (Endzustände) eingeteilt ist. Diese werden zunächst eingelesen und anschließend in das Json Schema in Abbildung 15 überführt. Dazu wird ein C# Programm mit .NET 5 als Laufzeitumgebung verwendet. [1] Die neue Datenstruktur bietet dabei mehrere Vorteile. Durch das *JSON* Format ist das Laden der Daten in den meisten Programmiersprachen mit wenig Aufwand möglich. Ebenfalls wird das Format in der dokumentorientierten Datenbank MongoDB verwendet, welche für das Labeln der

[1] C# ist eine von Microsoft entwickelte objektorientierte Programmiersprache. .NET 5 ist die Cross-Platform Laufzeitumgebung, welche für die Ausführung gebraucht wird. Mehr Informationen zu C# und .NET sind unter `https://dotnet.microsoft.com/` zu finden.

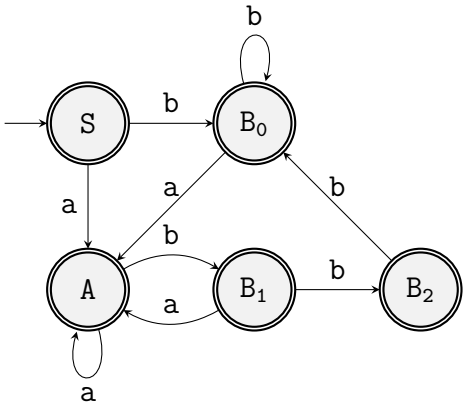


Abbildung 12: Der Automat beschreibt die Sprache L und verfolgt die Strategie. Da ein richtiger Automat auch mit 4 Zuständen umgesetzt werden kann, gehört dieser der Klasse 4 an.

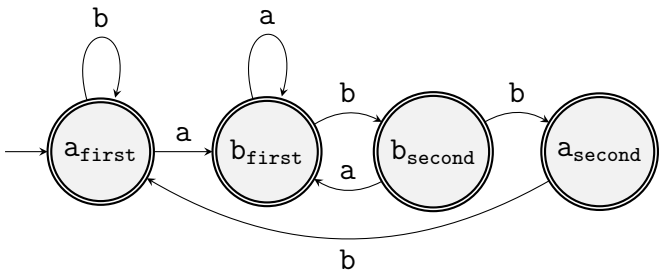


Abbildung 13: Der NFA ist eine korrekte Lösung für die Sprache L. Die Lösung benötigt nur 4 Zustände und löst die Aufgabe damit effizienter als die Automaten aus der Klasse 5.

Daten benötigt wird. Ebenso können die Daten leichter per HTTP versendet werden. Dadurch, dass die Daten bereits in einer Objektstruktur modelliert sind, können diese einfacher in objektorientierten Sprachen weiterverarbeitet werden. Die Daten sind um die Felder *Name* und *Label* erweitert worden. Das Feld *Name* ist ein für die Nutzerinteraktion und das Debugging festgelegter Anzeigename. Mit *Label* wird der Zielwert der Abgabe festgelegt. Eine umgewandelte Abgabe ist in Abbildung 16 dargestellt.

4.2 Labeln der Abgaben

Nachdem die Daten in einem geeigneten Format vorliegen, müssen diese noch von Hand mit Zielwerten versehen werden, um einen gelabelten Datensatz zu erhalten. Dies wird von einer Person durchgeführt, welche die Abgaben auch schon im Rahmen der Veranstaltung *Formale Sprachen und Logik* korrigiert hat. Um diesen Prozess zu erleichtern ist eine Webapplikation entwickelt worden. Diese baut auf *Angular* mit *PrimeNG* als UI-Komponenten-Framework auf und ermöglicht es, die Daten zu verwalten und zu bearbeiten. In einer Übersicht werden die gespeicherten Daten aufgelistet und können durchsucht und gefiltert werden. Die Oberfläche dazu ist in Abbildung 28 zu sehen. Ebenso können in der Ansicht einzelne Datenpunkte gelöscht werden oder zur Bearbeitungsoberfläche navigiert werden. Diese ist in Abbildung 29 dargestellt. Der *Graph-Editor* stellt den Automaten graphisch dar und ermöglicht das Erstellen, Anpassen, Bewerten und Entfernen von Datenpunkten. In der letzten Ansicht, welche in Abbildung 30 abgebildet ist, kann der Datensatz erweitert, heruntergeladen oder gelöscht werden. Um eine Strukturierung der Daten zu ermöglichen, können die Datenpunkte mit Kategorien versehen werden. Anhand dessen können die Datenpunkte bei einer Suchen gefiltert werden und beim Herunterladen der Daten separiert oder strukturiert werden. Für die Verwaltung der Daten wird im Hintergrund ein *Node.js* Webserver mit *Nest.js* als Webframework verwendet. Als Datenbankmanagementsystem wird *MongoDB* genutzt.

```
input_alphabet = a,b
start_states   = q0
transitions    = q0, a -> q1
                 q0, b -> q0
                 q1, a -> q1
                 q1, b -> q2
                 q2, a -> q1
                 q2, b -> q3
                 q3, a -> q4
                 q3, b -> q0
acc_states     = q0,q1,q2,q3
```

Abbildung 14: Definition eines NFA einer studentischen Abgabe.

```
{
    "Name": string,
    "Label": number,
    "Nodes":
    [{
        "Id": 0,
        "Name": string,
        "IsStart": boolean,
        "IsAccept": boolean
    }],
    "Edges":
    [{
        "From": number,
        "To": number,
        "Labels": [ string ]
    }]
}
```

Abbildung 15: JSON Schema für die umgewandelten studentischen Abgaben.

5 Experimentelles Setup

Nachdem der Datensatz um die Zielwerte erweitert wurde, kann die entsprechende Klassifikationsaufgabe aus Abschnitt 3 mit einem GNN gelöst werden. Das dafür erstellte Programm wird in diesem Kapitel vorgestellt und erläutert. Zu Beginn werden die wichtigsten Technologien vorgestellt. Anschließend wird die Aufbereitung der Daten für die GNN erläutert und deren Handhabung über eine Datensatzklasse vorgestellt. Abschließend werden die erstellten GNN Modelle dargestellt und der Ablauf des Trainings in seinen einzelnen Aspekten erklärt. In diesem Abschnitt wird sich auf das Softwareprojekt zu dieser Arbeit bezogen, welches in [23] gefunden werden kann. Insbesondere wird das Teilprojekt *gnn* betrachtet.

5.1 Technologien

Für den Aufbau des experimentellen Setups werden verschiedene Bibliotheken verwendet. Die wichtigsten werden im Folgenden kurz vorgestellt. Daneben werden noch einzelne Funktionen aus *scikit-learn* und *NumPy* verwendet. Diese können unter [28] und [25] nachgeschlagen werden.

Deep Graph Library Die *Deep Graph Library*[2] (*DGL*) ist eine Bibliothek zum Erstellen von GNN. Die Komponenten von *DGL* können in Verbindung mit verschiedenen Deep Learning Frameworks wie PyTorch oder TensorFlow genutzt werden. DGL bringt dabei Funktionen wie Komponenten für das Erstellen von GNN Modellen und Datenstrukturen zum Arbeiten auf Graphen

[2] https://github.com/dmlc/dgl

```
{
  "Name": "5a758b5d-a9cd-4ae7-a8ab-450143c0930a",
  "Nodes": [
    { "Id": 0, "Name": "q0",
      "IsStart": true, "IsAccept": true }],
  "Edges": [
    { "From": 0, "To": 1,
      "Labels": [ "a" ]}],
  "Label": 1
}
```

Abbildung 16: Studentische Abgabe, welche in das JSON Format umgewandelt ist.

mit sich. Ebenfalls wird ein Interface zur Erstellung von eigenen Graph Neural Network Modulen bereitgestellt. Wie DGL genauer funktioniert und aufgebaut ist, kann in [29, 24, 31] nachgelesen werden.

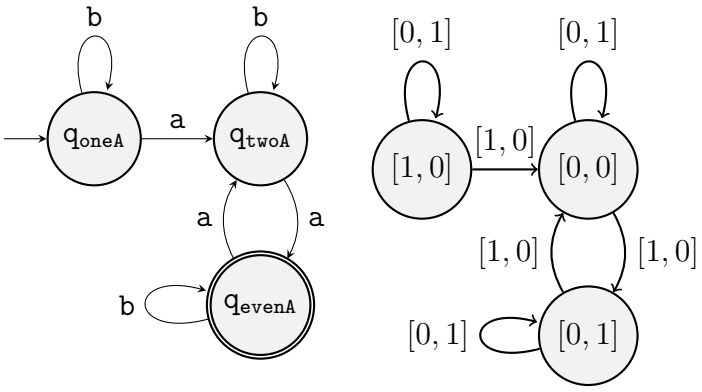
PyTorch *PyTorch* ist ein Machine Learning Framework, welches es ermöglicht NN zu erstellen und zu trainieren. Dabei werden auch Umsetzungen von Fehlerfunktionen und Optimierern mitgeliefert, welche nur noch konfiguriert werden müssen. Des Weiteren ist Pytroch kompatibel mit anderen gängigen Frameworks im Bereich des Machine Learning, wie scikit-learn oder Numpy. Unter [26] ist die aktuelle Dokumentation von Pytorch zu finden.

RayTune *RayTune* ist eine Bibliothek des *Ray* Frameworks und konzentriert sich auf die Ausführung von Experimenten und die Optimierung von Hyperparametern. Die Bibliothek ermöglicht es dem Nutzer einen Suchraum zu definieren, aus welchem Konfigurationen für das Training erstellt werden. Um die Suche und Auswahl der optimalen Parameter weiter zu konfigurieren, stellt *RayTune* Suchalgorithmen und Versuchsplaner bereit. Ebenfalls bietet es eine Schnittstelle für eine parallele Ausführung der Trainings. Die Dokumentation zu *RayTune* ist unter [27] zu finden.

5.2 Vorverarbeitung

Die Basis für das Experiment ist der gelabelte Datensatz, beschrieben in Abschnitt 4. In DGL werden Graphen mittels der *DGLGraph* Struktur repräsentiert. Um einen Graph zu erstellen, werden nur die Kanten benötigt. Dazu werden die Ids der $\mathtt{n}$ Knoten in zwei Listen $\mathtt{u} = [\mathtt{u}_0, \ldots, \mathtt{u}_\mathtt{n} - 1]$ und $\mathtt{v} = [\mathtt{v}_0, \ldots, \mathtt{v}_\mathtt{n} - 1]$ strukturiert, wobei $\mathtt{n}$ die Anzahl der Knoten ist. Eine Kante ergibt sich dabei aus $(\mathtt{u}_\mathtt{i}, \mathtt{v}_\mathtt{i})$. Des Weiteren müssen die Merkmale der Knoten und Kanten umgewandelt werden. Ein Knoten hat die Merkmale *IsStart* und *IsAccept*. Beide Merkmale werden durch das One-Hot Encoding numerisch dargestellt. Bei den Kanten sind die Merkmale die Liste der *Labels*. Bei den Merkmalen handelt es sich um kategorische Daten, weshalb diese mit dem One-Hot Encoding weiterverarbeitet werden. Eine Kante hat dann für jedes Zeichen aus dem Alphabet des NFAs ein Merkmal, welches mit 0 oder 1 beschreibt, ob diese Transition möglich ist oder nicht. Die gesamte Umwandlung ist graphisch in Abbildung 17 dargestellt. Um die NFAs auch mit einem

R-GCN verarbeiten zu können, müssen die Kanten entsprechend in einen Relationstypen eingeteilt werden. Für ein Alphabet Σ sind die möglichen Relationstypen $\mathtt{r_i} \in \mathcal{R}$, wobei $\mathcal{R} = \mathcal{P}(\Sigma) \setminus \{\emptyset\}$. In DGL wird ein Relationstyp durch eine Zahl repräsentiert, wofür eine Abbildung von $\mathtt{f} : \mathcal{R} \to \mathbb{N}$ verwendet wird. Ein Relationstyp wird mit $\mathtt{f(r_i)}$ gespeichert. In diesem Fall wird, mit dem Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ und den daraus resultierenden Relationstypen $\mathcal{R} = \{\mathtt{a}, \mathtt{b}, \mathtt{ab}\}$, die Abbildung $\mathtt{f(r)} = \{\mathtt{a} \to 1, \mathtt{b} \to 2, \mathtt{ab} \to 3\}$ verwendet. In einem DGLGraphen sind die Relationstypen in einer Liste $\mathtt{rel} = [\mathtt{r_1}, ...\mathtt{r_m}]$ abgespeichert, wobei die Länge der Liste gleich der Anzahl der Kanten des Graphen ist. Die Relation an der Position $\mathtt{k}$ gehört zu der Kante, welche sich aus $(\mathtt{u_k}, \mathtt{v_k})$ ergibt. Ein Beispiel für die Umwandlung mit Relationen ist in Abbildung 18 abgebildet.

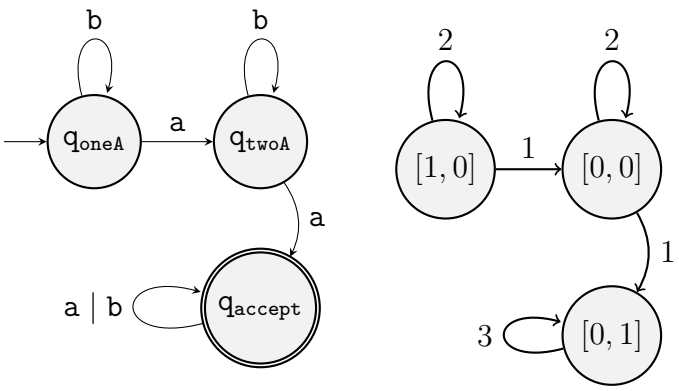



```
Graph(
    nodes = [0, 1, 2],
    edges = (
            [0, 0, 1, 1, 2, 2],
            [0, 1, 1, 2, 2, 1]
        ),
    ndata={ "features" : [
                [1, 0],
                [0, 0],
                [0, 1]
            ]
        },
    edata={ "features" : [
                [0, 1], [1, 0],
                [0, 1], [1, 0],
                [0, 1], [1, 0],
            ]
        }
)
```

Abbildung 17: Umwandlung eines NFA in seine Repräsentation als DGLGraph. Ganz links ist der NFA als Ausgangspunkt dargestellt. Rechts daneben ist dieser als Graph dargestellt. Dabei sind die Merkmale der Kanten und Knoten bereits umgewandelt. In der Abbildung auf der rechten Seite ist der Graph schematisch vereinfacht als DGLGraph dargestellt.

5.3 Datensatz

Der Datensatz wird für das Training in 2 Teile aufgeteilt. Insgesamt 20% des gesamten Datensatzes werden als Testdaten (TestD) verwendet. Die restlichen 80% des Datensatzes werden als Trainingsdatensatz (TrainD) verwendet. Für die Aufteilung des Datensatzes wird von *scikit-learn* die *train_test_split* Methode verwendet, wobei der Datensatz stratifiziert wird. Die Daten werden über die *FiniteAutomataDataset* Klasse vorverarbeitet und geladen. Das Interface der Klasse ist in Abbildung 31 zu finden. Über Ihre Felder stellt die Klasse Metainformationen über den Datensatz zur Verfügung, welche für die Definition der GNN Modelle verwendet werden. Da die Klasse von *DGLDataset* erbt, werden die Daten mit einer vorgegebenen Prozedur geladen. Zunächst wird




```
Graph(
    nodes = [0, 1, 2],
    edges = (
            [0, 0, 1, 1, 2],
            [0, 1, 1, 2, 2]
        ),
    ndata={ "features" : [
                [1, 0],
                [0, 0],
                [0, 1]
            ]
        },
    edata={ "types" :
            [ 2, 1, 2, 1, 3 ]
        }
)
```

Abbildung 18: Umwandlung eines NFA in seine Repräsentation als DGLGraph, wobei die Merkmale der Kanten als Relationstypen umgewandelt werden. Die Transitionen des NFAs werden dabei als Relationstypen $\{\mathtt{a}, \mathtt{b}, \mathtt{ab}\}$ aufgefasst. Anschließend wird ein Relationstyp mittels der Funktion $\mathtt{f(r)} = \{\mathtt{a} \to 1, \mathtt{b} \to 2, \mathtt{ab} \to 3\}$ in eine Zahl umgewandelt. In einem DGLGraph wird der Relationstyp in der List in der *edata* abgespeichert, hier *types*. Ein Eintrag aus der Liste der Relationstypen an der Positions `i` gehört dabei zu der Kante, welche sich aus den Einträgen der beiden Listen aus *edges* an der Position `i` ergibt.

mit der *has_cache* Methode geprüft, ob schon vorverarbeitete Daten gespeichert sind . Sind die Daten bereits vorhanden, dann wird die *load* Methode aufgerufen, welche diese Daten einliest. Ansonsten wird mit der *process* Methode weitergemacht. Dort wird dann der gelabelte Datensatz geladen und anschließend vorverarbeitet, wie in 5.2 bereits erklärt. Anschließend werden die vorverarbeiteten Daten mit der *save* Methode gespeichert, damit diese bei einem späteren Training wieder geladen werden können. Über den boolschen Parameter *train* wird bestimmt, welcher der beiden Datensätze geladen wird.

5.4 Modelle

Für die Experimente werden 2 Modelle von GNN erstellt, das *NNGraphConvClassifier* und das *RelGraphConvClassifier*. In den beiden Modellen ist jeweils definiert, wie die Zustände der Knoten $\mathtt{h}_\mathtt{v}^\mathtt{l}$ berechnet werden. Da die Struktur und der Aufbau für den Graph Readout und für die Berechnung der Ausgabe `g` gleich sind, ist dieser Teil in der Basisklasse *GraphClassifierBase* implementiert. Beide Modelle erben von dieser Klasse, damit der Ablauf vollständig ist.

NNGraphConvClassifier Der *NNGraphConvClassifier* ist eine Umsetzung des MPNN Models. Für die Berechnung der Zustände $\mathtt{h}_\mathtt{v}^\mathtt{L}$ werden insgesamt 2 Schichten des von DGL bereitgestellten Moduls *NNConv* verwendet. Der Zustand $\mathtt{h}_\mathtt{v}^{\mathtt{l}+1}$ eines Knoten `v` einer Schicht $\mathtt{l}+1$ ist dort gegeben mit $\mathtt{h}_\mathtt{v}^\mathtt{l} + \mathtt{aggregate}(\{\mathtt{f}_\Theta(\mathtt{x}_{\mathtt{vu}}) \cdot \mathtt{h}_\mathtt{u}^\mathtt{l} \mid \mathtt{u} \in \mathtt{N}_\mathtt{v}\})$. Dabei ist `aggregate` eine Aggregationsfunktion,

welche die Summe, den Mittelwert oder das Minimum der Menge berechnet. Die Merkmale der Kanten werden über die Funktion f_θ verarbeitet. Für jede Schicht wird f_θ als ein einschichtiges NN umgesetzt. Die Dimension der Ausgabe der einzelnen Schichten ist jeweils frei konfigurierbar. Als Eingabe wird ein vorverarbeiteter Graph als *DGLGraph* erwartet.

RelGraphConvClassifier Ein R-GCN ist mit dem *RelGraphConvClassifier* umgesetzt. Die Berechnung der Zustände h_v^L ist über 2 Schichten realisiert. Eine Schicht l ist eine Instanz des *RelGraphConv* Moduls aus DGL. Dieses berechnet den Zustand h_v^{l+1} analog, wie bereits in 2.4 vorgestellt. Wie beim *NNGraphConvClassifier* sind die Dimensionen der Ausgaben der einzelnen Schichten frei konfigurierbar. Als Eingabe in das Netz wird erwartet, dass die Kanten in entsprechende Relationstypen eingeteilt sind, wie in Abschnitt 5.2 beschrieben.

GraphClassifierBase Nachdem für alle Knoten h_v^L berechnet wurde, wird der Zustand des Graphen h_G über eine Graph Readout Funktion gebildet und anschließend mit einem NN die Ausgabe des GNNs bestimmt. Die verwendete Graph Readout Funktion kann dem GNN per Parameter übergeben werden. Die möglichen Funktionen sind in Abbildung 19 aufgelistet. Das NN setzt sich aus 3 Schichten zusammen. Die Anzahl der Neuronen kann für die ersten beiden Schichten individuell definiert werden. In der letzten Schicht ist die Anzahl der Neuronen gleich der Anzahl der möglichen Klassen. Als Aktivierungsfunktion wird in den versteckten Schichten die ReLU Funktion verwendet. In der Ausgabeschicht wird auf die Softmax Funktion zurückgegriffen, da es sich um ein Multiklassen Klassifikationsproblem handelt.

Folgende Funktionen können als Graph Readout Funktion in einem GNN verwendet werden. Die Eingabe in das GNN ist ein Graph G mit n Knoten.

$$\text{Summe der Zustände} - h_G = \sum_{v \in V} h_v^L$$
$$\text{Mittelwert der Zustände} - h_G = \frac{1}{n} \sum_{v \in V} h_v^L$$
$$\text{Maximum der Zustände} - h_G = \max(\{h_v^L \mid v \in V\})$$
$$\text{Minimum der Zustände} - h_G = \min(\{h_v^L \mid v \in V\})$$

Abbildung 19: Verwendbare Readout Funktionen.

5.5 Training eines Modells

Für das Training eines Modells wird der TrainD in zwei Teile geteilt, den Validierungsteil und den Trainingsteil. Der Validierungsdatensatz (ValD) mit 20% des TrainD wird für die Validierung des Modells verwendet. Der Rest des TrainD wird für die Anpassung der Gewichte genutzt und wird in diesem Abschnitt ebenfalls als TrainD bezeichnet. Um die Parameter des Modells anzupassen wird der Optimizer *Adam* von PyTorch verwendet. Mehr über *Adam* kann in [15] nachgelesen werden. Als Fehlerfunktion wird die Implementierung der Kreuzentropie *CrossEntropyLoss* von

PyTorch verwendet. Der Ablauf des Trainings ist in Abbildung 32 als Pseudocode dargestellt und wird für die folgende Beschreibung als Grundlage genutzt. Ein Training läuft insgesamt n Epochen. In jeder dieser Epochen werden für alle Graphen aus dem TrainD nacheinander folgende Schritte durchgeführt (Zeile 9 bis 12). Als Erstes wird die Ausgabe des GNN berechnet (Zeile 9). Daraufhin wird die Fehlerfunktion auf die Ausgabe und den Zielwert angewendet (Zeile 10) und anschließend die Änderungen für die Parameter berechnet (Zeile 11). Abschließend werden die Änderungen über den Optimizer angewandt. Zum Abschluss einer Epoche wird die Performance des Modells über den ValD mit der *validation* Methode überprüft. In Abbildung 33 ist der Ablauf der Validierung als Pseudocode dargestellt. Für die Validierung werden auf allen Graphen die Ausgabe des GNNs bestimmt und die Fehlerfunktion angewandt. Anschließend wird über die Methode *are_same_class* geprüft, ob die Zuordnung der Klasse korrekt ist. War die Zuordnung richtig wird der Zähler in *correct_predictions* hochgezählt. Ebenfalls wird der berechnete Fehler in *summed_loss* aufsummiert. Für die Auswertung wird dann der Mittelwert des Fehlers und die Accuracy gebildet. Wird RayTune für die Optimierung der Parameter verwendet, werden die Ergebnisse der Validierung mittels *tune.report* weitergeleitet. Wie RayTune für die Hyperparameteroptimierung verwendet wird und wie ein Modell auf den Testdatensatz überprüft wird, ist in den folgenden zwei Abschnitten erläutert.

Hyperparameteroptimierung mit RayTune Die Optimierung der Hyperparameter wird mit RayTune umgesetzt. Dabei wird RayTune mit der Konfiguration für den Suchraum aus Abbildung 34 und dem Versuchsplaner *ASHA* gestartet. Mehr über ASHA kann in [27] nachgelesen werden. Der Versuchsplaner ist dabei so konfiguriert, dass als Überprüfungskriterum der Fehler minimiert wird. Ist RayTune gestartet, zieht dieser eine Konfiguration aus dem gegebenen Suchraum und startet den Trainingprozess mit der gezogenen Konfiguration. Nach jeder Validierungsphase wird das Ergebnis dieser RayTune mitgeteilt, wie es in Abbildung 33 dargestellt ist. Auf Basis dieser Ergebnisse wird entschieden, ob das Training des Modells abgebrochen wird. Zum Start des Experiments wird festgelegt, wie viele Stichproben RayTune aus dem Suchraum ziehen soll. Wurden alle Stichproben überprüft, werden anschließend die Parameter des Modells mit dem niedrigsten Fehler auf den Validierungsdaten ausgelesen und mit den Testdaten ausgewertet.

Auswertung des Modells Für die Auswertung des Modells werden wie bei der Validierung die Ausgaben des GNNs über dem TestD gebildet. Die Ausgaben und Zielwerte werden in zwei entsprechenden Listen gespeichert. Anschließend wird dann auf Basis der zwei Listen die Metriken mit *metrics.confusion_matrix* und *metrics.classification_report* berechnet. Beide Funktionen sind aus *scikit-learn* und können unter [28] nachgeschlagen werden. Die verwendeten Metriken sind hier die *Accuracy*, der *Recall* und die *Precision*. Die *Accuracy* wird verwendet, um eine Allgemeine Aussage über die Performance das Modell zu treffen, da diese angibt, wie viele der Prädiktionen korrekt sind. Der *Recall* ermöglicht eine Aussage darüber, wie viele Datenpunkte einer Klasse gefunden wurden. Mit der *Precision* kann Überprüft werden, wie viele Zuordnungen zu einer Klasse korrekt sind. Ein Beispiel für die berechneten Metriken ist in Abbildung 35 zu finden.

Tabelle 1: Aufteilung der Datenpunkte auf die Klassen in Prozent für alle Szenarien

Szenario	1	2	3	4	5
2 Klassen	51.43%	48.57%	-	-	-
3 Klassen	20%	31.43%	48.57%	-	-
5 Klassen	20%	17.14%	14.29%	42.86%	5.71%

6 Ergebnisse

Auf Basis des in Abschnitt 5 vorgestellten Experimentsetups, sind in diesem Abschnitt die Resultate der Experimente beschrieben. Die Experimente sind in 3 verschiedene Klassenszenarien eingeteilt. Für zwei der folgenden Szenarien werden mehrere Klassen aus Abschnitt 3 zu einer zusammengefasst. Dadurch wird die Klassifikationsaufgabe vereinfacht, da eine Klasse durch mehr Datenpunkte repräsentiert wird und die Kriterien verschiedener Klassen sich stärker unterscheiden.

- *Szenario 1* - Im ersten Szenario findet der Versuch mit den 5 Klassen, welche in Abschnitt 3 vorgestellt wurden, statt.
- *Szenario 2* - Für das zweite Szenario sind die Klassen 2 und 3 zusammengelegt und stellen die Abgaben dar, welche eine Strategie verfolgen, aber Fehler enthalten. Diese wird als Klasse 2 bezeichnet. Die Klasse 4 und die Klasse 5 sind zur der Klasse 3 vereint und enthalten damit alle Abgaben, bei welchen die Aufgabe richtig gelöst wurde. Die Klasse 1 bleibt unverändert.
- *Szenario 3* - Im letzten Szenario wird die Klassifikationsaufgabe auf 2 Klassen reduziert. Dazu werden die Klassen 1, 2 und 3 zur der Klasse 1 zusammengefasst, sowie die Klassen 4 und 5 zu der Klasse 2. Die Klasse 1 repräsentiert damit die falschen Lösungen und die Klasse 2 die richtigen Lösungen. Damit ist dieses Szenario eine binäre Klassifikation.

In Tabelle 1 ist die Aufteilung der Datenpunkte in die einzelnen Szenarien zusammengefasst. Für alle 3 Szenarien werden jeweils 8 Experimente für das R-GCN und das MPNN durchgeführt, wobei sich jeweils die Anzahl der Trainingsepochen unterscheidet. Begonnen wird dabei mit 10 Epochen, welche im nächsten Versuch um 10 erhöht werden. Bei jedem Experiment werden mit RayTune 30 Stichproben an Hyperparametern aus dem Suchraum, welcher in Abbildung 34 zu finden ist, gezogen und anschließend trainiert und validiert. Abschließend wird die Stichprobe mit dem geringsten Fehler erneut trainiert und getestet, um das Trainingsergebnis zu bestätigen und auszuwerten. Eine Übersicht der Ergebnisse ist für die MPNN Modelle in Abbildung 2 und für die R-GCN Modelle in Abbildung 3 dargestellt.

Accuracy In den Graphen in Abbildung 20 und Abbildung 21 sind die Accuracies aller Experimente aufgelistet. Sowohl bei den MPNN Modellen als auch bei den R-GCN Modellen ist die Accuracy bei dem 5-Klassen Szenario am geringsten und ordnet sich im Bereich von 0.5 ein. Die Ausnahme ist das Modell *R-GCN 13*, welche eine Accuracy von 0.657 erzielt hat. Im 3 Klassen Szenario fallen die Werte, mit Werten um die 0.7, etwas höher aus für das R-GCN als auch für das MPNN. Das R-GCN Modell *R-GCN 7* hat mit 0.8 die höchste Accuracy für 3 Klassen. In der binären Klassifikation sind die Ergebnisse am höchsten. Dort sind die besten Modelle *R-GCN*

Tabelle 2: Konfigurationen der Experimente mit dem MPNN. Dabei ist Y die Dimension der Ausgabe des GNNs, LR die Lernrate, h_v^i die Dimension von h_v^i, h_v^i Aggr ist die Aggregationsfunktion, welche für die Berechnung von h_v^i verwendet wird, R die Readout Funktion, NNi die Ausgabedimension der i-ten Schicht des NNs für die Berechnung von g, E die Anzahl der Epochen und Acc die Accurcay des Modells auf den Testdaten.

	Name	Y	LR	h_v^1	h_v^1 Aggr	h_v^2	h_v^2 Aggr	R	NN1	NN2	E	Acc
Szenario 3	MPNN 23	2	0.00512	16	sum	4	mean	sum	8	8	10	0.886
	MPNN 11	2	0.00104	16	sum	16	mean	sum	128	32	20	0.829
	MPNN 8	2	0.00049	128	sum	256	sum	mean	32	32	30	0.886
	MPNN 18	2	0.00043	256	mean	4	mean	sum	64	128	40	0.857
	MPNN 19	2	0.00036	128	mean	4	sum	mean	64	128	50	0.886
	MPNN 10	2	0.00202	16	sum	128	mean	mean	8	8	60	0.829
	MPNN 3	2	0.00074	32	sum	4	sum	sum	4	16	70	0.886
	MPNN 9	2	0.00037	8	mean	16	sum	sum	128	16	80	0.886
Szenario 2	MPNN 20	3	0.00015	16	mean	64	sum	sum	256	128	10	0.743
	MPNN 6	3	0.00049	64	sum	16	mean	sum	128	32	20	0.686
	MPNN 7	3	0.00014	256	sum	256	sum	mean	32	256	30	0.714
	MPNN 21	3	0.00267	4	mean	32	sum	mean	8	16	40	0.629
	MPNN 5	3	0.00153	4	mean	8	sum	sum	16	32	50	0.686
	MPNN 4	3	0.00060	32	sum	64	sum	mean	16	128	60	0.686
	MPNN 15	3	0.00022	128	mean	256	sum	sum	64	32	70	0.714
	MPNN 13	3	0.00015	128	mean	128	sum	sum	64	4	80	0.657
Szenario 1	MPNN 14	5	0.00019	128	sum	256	mean	sum	4	128	10	0.543
	MPNN 0	5	0.00176	64	sum	64	mean	mean	16	64	20	0.486
	MPNN 2	5	0.00047	64	sum	256	mean	sum	256	32	30	0.543
	MPNN 17	5	0.00047	64	sum	32	sum	mean	128	128	40	0.543
	MPNN 22	5	0.00145	16	sum	128	mean	mean	4	256	50	0.514
	MPNN 1	5	0.00022	256	sum	128	mean	mean	64	256	60	0.514
	MPNN 16	5	0.00048	256	mean	128	mean	sum	32	32	70	0.543
	MPNN 12	5	0.00040	256	mean	64	sum	mean	16	64	80	0.543

Tabelle 3: Konfigurationen der Experimente mit dem R-GCN. Dabei ist Y die Dimension der Ausgabe des GNNs, LR die Lernrate, h_v^i die Dimension von h_v^i, R die Readout Funktion, NNi die Ausgabedimension der i-ten Schicht des NN für die Berechnung von g, E die Anzahl der Epochen und Acc die Accuracy des Modells auf den Testdaten.

	Name	Y	LR	h_v^1	h_v^2	R	NN1	NN2	E	Acc
Szenario 3	R-GCN 18	2	0.00193	8	32	mean	16	16	10	0.914
	R-GCN 8	2	0.00028	16	64	sum	128	64	20	0.914
	R-GCN 6	2	0.00102	16	16	sum	16	32	30	0.914
	R-GCN 4	2	0.00039	128	4	mean	8	256	40	0.886
	R-GCN 11	2	0.00149	4	16	sum	32	32	50	0.914
	R-GCN 17	2	0.00017	256	32	sum	128	128	60	0.829
	R-GCN 9	2	0.00098	64	32	mean	128	32	70	0.857
	R-GCN 20	2	0.00020	16	256	sum	64	256	80	0.886
Szenario 2	R-GCN 7	3	0.00190	4	32	sum	64	16	10	0.800
	R-GCN 23	3	0.00030	8	64	sum	256	64	20	0.714
	R-GCN 12	3	0.00118	4	32	mean	16	128	30	0.686
	R-GCN 2	3	0.00018	64	8	mean	8	64	40	0.686
	R-GCN 1	3	0.00320	64	4	mean	8	256	50	0.714
	R-GCN 14	3	0.00011	256	128	sum	16	4	60	0.686
	R-GCN 19	3	0.00342	4	32	sum	16	8	70	0.714
	R-GCN 5	3	0.00211	32	8	mean	64	4	80	0.657
Szenario 1	R-GCN 15	5	0.00334	16	64	mean	4	128	10	0.571
	R-GCN 22	5	0.00467	16	4	mean	8	4	20	0.486
	R-GCN 16	5	0.00754	4	32	mean	4	4	30	0.514
	R-GCN 3	5	0.00030	256	256	sum	128	128	40	0.657
	R-GCN 10	5	0.01867	4	64	mean	8	4	50	0.429
	R-GCN 13	5	0.00176	4	128	sum	32	4	60	0.571
	R-GCN 0	5	0.00320	8	128	mean	8	16	70	0.600
	R-GCN 21	5	0.00103	8	8	sum	64	16	80	0.571

18, *R-GCN 8*, *R-GCN 6* und *R-GCN 11* mit einer Accuracy von 0.914. Insgesamt erziehlen beide Modellarten Ergebnisse mit Werten ≥ 0.82. Die MPNN Modelle erzielen in allen 3 Szenarien geringere Werte als die R-GCN Modelle. Bei allen Verläufen liegen die Werte nah beieinander. Ausreißer treten bis auf die Epochen 40 und 50 im 5-Klassen Szenario der R-GCN Modelle nicht auf.

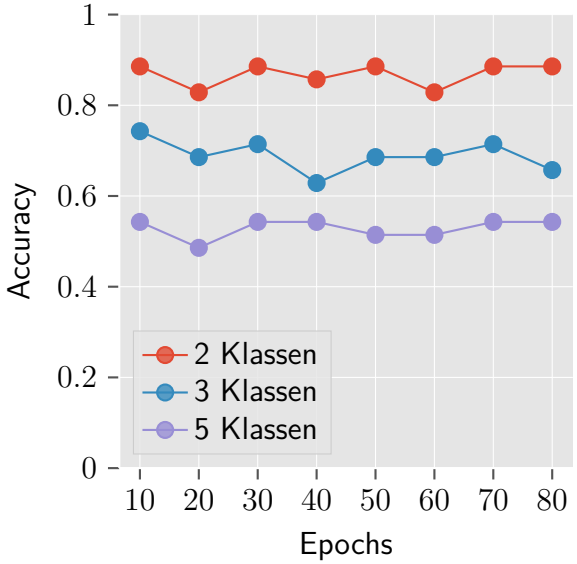


Abbildung 20: Accuracies der besten MPNN Modelle für alle 3 Klassenszenarien.

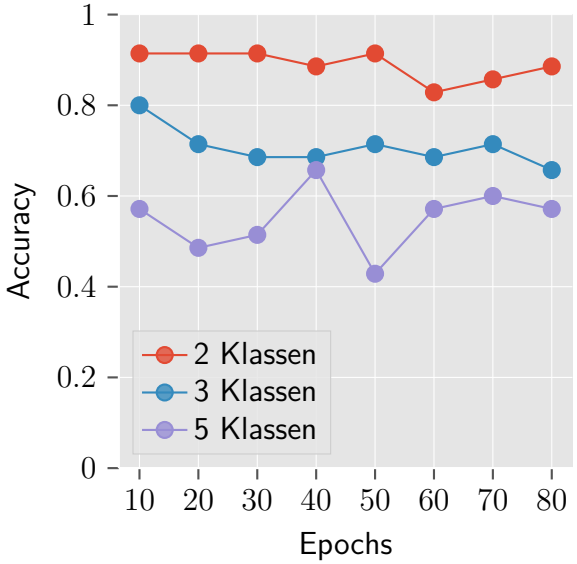


Abbildung 21: Accuracies der besten R-GCN Modelle für alle 3 Klassenszenarien.

Recall Der gemittelte Recall ist in den Abbildungen 22 und 23 dargestellt. Der gemittelte Recall wird in diesem Abschnitt nur als Recall bezeichnet. Für das 2-Klassen Szenario liegen bei beiden Modellarten Ergebnisse über einem Wert von 0.8 und gleichen dem Verlauf der Accuracy. Die Experimente des 2-Klassen Szenarios erreichen in allen Fällen die höchsten Ergebnisse. Im 3-Klassen Szenario erzielen beide Modellarten nur leicht abweichende Ergebnisse im Bereich von 0.6. Das *R-GCN 7* erreicht einen Recall von 0.79 und ist damit der höchste Wert. Bei den R-GCN Modellen fällt der Recall mit einer größer werdenden Epochenzahl weiter ab. Das kleinste Ergebnis wird bei 80 Epochen mit 0.55 erzielt. Bei den MPNN Modellen wird der höchste Wert 0.635 bei einer Anzahl von 70 Epochen erzielt. Insgesamt fallen die Werte der R-GCN Modelle höher aus als die der MPNN Modelle. Die niedrigsten Werte werden im 5-Klassen Szenario erreicht. Diese siedeln sich im Bereich von 0.4 ein. Der Verlauf des Graphen für die MPNN Modelle steigt bis zu 50 Epochen an. Bei 60 Epochen fällt der Wert dann wieder leicht. Der höchste Wert wird bei *MPNN 16* erreicht mit einem Recall von 0.46. Bei den R-GCN Modellen liegt der Recall mit einer Anzahl von 10 bis 30 und 50 Epochen unter einem Wert von 0.4. Für die anderen Epochen sind die Werte > 0.4. Der höchste Wert wird vom R-GCN 3 mit 0.53 erreicht. Insgesamt erreichen die R-GCN Modelle die höchsten Werte für den Recall.

Precision In den Abbildungen 24 und 25 ist die gemittelte Precision zu finden, welche im Folgenden nur als Precision bezeichnet wird. Wie beim gemittelten Recall sind die Resultate für das 2-Klassen Szenario am höchsten und liegen über einem Wert von 0.8. Am höchsten sind die Werte der R-GCN Modelle *R-GCN 18*, *R-GCN 8*, *R-GCN 6* und *R-GCN* mit 0.91. Im 3-Klassen Szenario befinden sich die Resultate der R-GCNs in einem Bereich von $0.4 <$ und < 0.8. Der höchste Wert ist mit 0.78 bei 30 Epochen erreicht. Das kleinste Resultat wird bei 40 Epochen erzielt. Die MPNN Modelle sind bei ihrem höchsten und kleinsten Werte stärker ausgeprägt und bilden einen ähnlichen Verlauf wie die R-GCN Modelle ab. Der höchste Wert liegt bei 0.81 für 70 Epochen im Training. Das ist auch der höchste Wert für das 3 Klassen Szenario. Wie bei den R-GCN Modellen wird der kleinste Wert bei 40 Epochen erzielt mit einem Wert von 0.41. Für das 5 Klassen Szenario steigen die Ergebnisse für die Precision mit wachsender Anzahl an Epochen weiter an. Sie sind dabei in einem Bereich von $0.05 <$ und < 0.6 verteilt. Bei den MPNNs bilden die Epochen 50 und 60 eine Ausnahme, da die Precision kurz fällt. Vom *MPNN 16* ist bei 70 Epochen mit 0.579 der höchste Wert erreicht. Die Epochen 30 bis 50 unterscheiden sich stark von deren benachbarten Werten. Mit den Epochen 30 und 50 werden die niedrigsten Werte für das 5-Klassen Szenario erzielt, bei denen der tiefste Wert bei 0.085 liegt. Der höchste Wert für die R-GCN Modelle ist 0.578. Im Gegensatz zur Accuracy und dem gemittelten Recall erreichen die MPNN Modelle für die Precision die höchsten Werte im 3-Klassen und 5-Klassen Szenario.

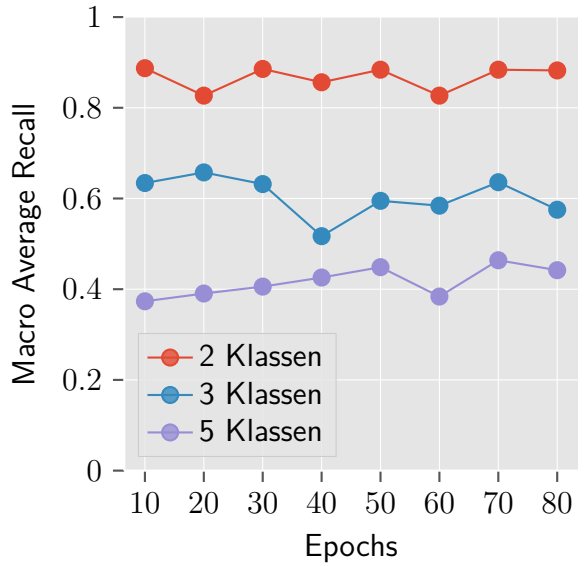


Abbildung 22: Mittlerer Recall der besten MPNN Modelle für alle 3 Klassenszenarien.

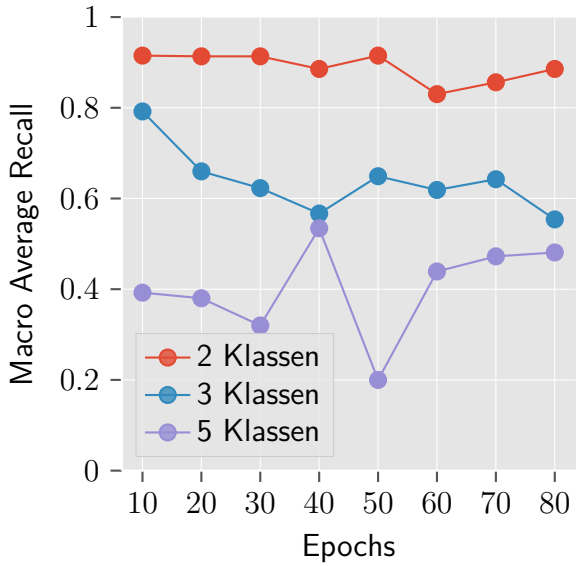


Abbildung 23: Mittlerer Recall der besten R-GCN Modelle für alle 3 Klassenszenarien.

6.1 Precision und Recall der einzelnen Klassen

Im vorhergehenden Abschnitt wurden die gemittelten Werte für den Recall und die Precision betrachtet. Im Folgenden werden für das 3 Klassen Szenario und das 5 Klassen Szenario die Werte für die einzelnen Klasse vorgestellt und untersucht. Dabei wird jeweils das Modell mit der höchsten Accuracy für das R-GCN und MPNN betrachtet.

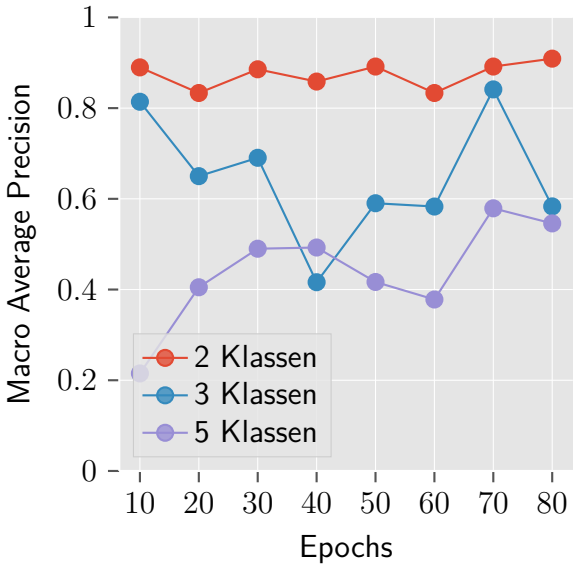


Abbildung 24: Mittlere Precision der besten MPNN Modelle für alle 3 Klassenszenarien.

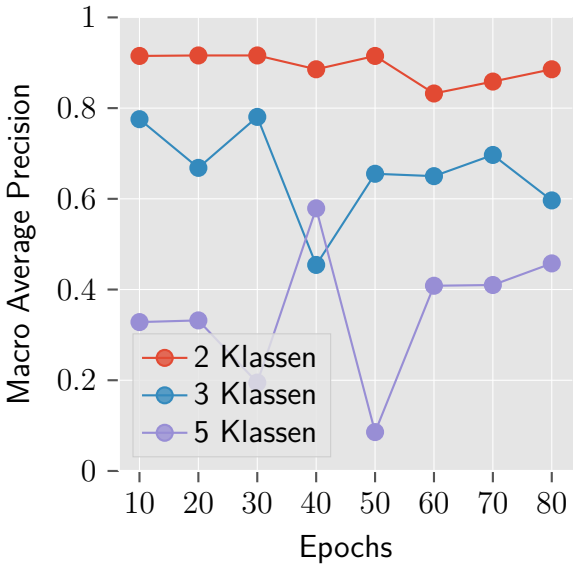


Abbildung 25: Mittlere Precision der besten R-GCN Modelle für alle 3 Klassenszenarien.

3 Klassen Szenario In den Säulendiagrammen in der Abbildung 26 ist der Recall und die Precision der einzelnen Klassen für das 3 Klassen Szenario dargestellt. Auffällig ist, dass das MPNN Modell nur einen geringen Anteil der Datenpunkte der Klasse 1, also die Klasse der falschen Abgaben, erkennt. Jedoch sind die Zuordnungen zur Klasse 1 korrekt. Das R-GCN hat im Vergleich zum MPNN für die Klassen 2 und 3 höhere Werte in der Precision, dafür aber niedrigere Werte im Recall. Lediglich bei der Klasse 1 ist die Precision für das MPNN Modell höher, wobei aber nur ein geringer Anteil der Datenpunkte von Klasse 1 gefunden werden.

5 Klassen Szenario Der Recall und die Precision der einzelnen Klassen für das 5 Klassen Szenario ist in der Abbildung 27 zu finden. Ein markantes Ergebnis ist, dass beide Modelle in der Klasse 3 für den Recall und die Precision einen Wert von 0 haben. Es wurde also von keinem der beiden Modelle ein Datenpunkt in die Klasse 3 eingestuft. Auch die Ergebnisse für die Klasse 4 und 5 ähneln sich sehr. Die richtigen Abgaben werden mit einer hohen Wahrscheinlichkeit ihrer Klasse zugeordnet. Der verhältnismäßig niedrige Wert des Recall der Klasse 5 lässt sich durch die Repräsentation der Klasse von $5,71\%$ Datenpunkte des Testdatensatzes erklären. Vor allem beim MPNN Modell sind die Ergebnisse für die Klassen 1 und 2 bei Werten unter ≤ 0.5 angesiedelt. Die Werte des R-GCN sind bei der Klasse 1 nur leicht höher. Die Klasse 2 wird sicherer erkannt vom R-GCN und auch über die Hälfte der Datenpunkte der Klasse werden gefunden, da der Recall und die Precision hier höher ausfällt.

7 Auswertung

Das Ziel der Arbeit ist es einen Weg zu finden, die Hausaufgaben aus der Veranstaltung *Formale Sprachen und Logik* in Form von endlichen Automaten mit einem NN zu bewerten. Die Bewertung

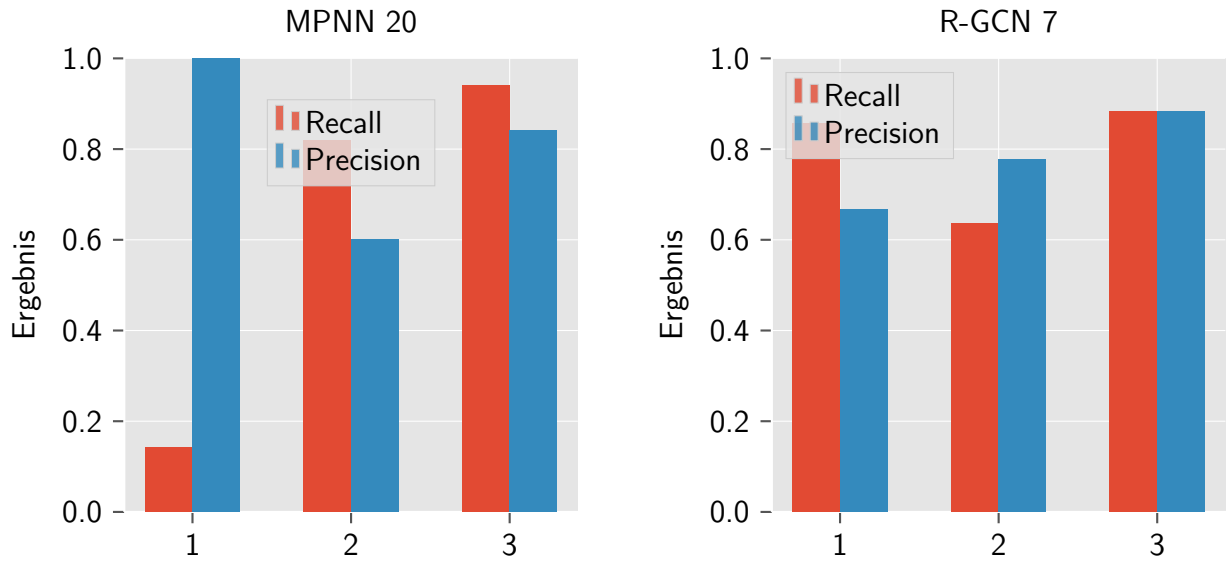


Abbildung 26: Precision und Recall der einzelnen Klassen für das 3-Klassen Szenario für die Modelle mit der höchsten Accuracy. Auf der linken Seite sind die Werte des MPNN Modells dargestellt und auf der rechten Seite die des R-GCNs.

soll sich dabei an den Kriterien eines menschlichen Korrektors orientieren. Mit den GNN ist dafür prinzipiell eine Möglichkeit gefunden. Auch die Resultate der trainierten Modelle sind vielversprechend. Die im Szenario 3 erzielte Accuracy von 0.914 stellt ein sehr gutes Ergebnis dar und ist von 4 R-GCN Modellen reproduziert worden. Auch die restlichen Ergebnisse der Experimente des Szenario 3 befinden sich in einem guten Bereich mit einer minimalen Accuracy von 0.829. Ebenfalls sind die Ergebnisse für das Szenario 2 positiv zu bewerten. Ein Großteil der Modelle erreicht eine Accuracy im Bereich von 0.7. Die entstandenen Netze sind dabei noch nicht für den produktiven Betrieb zu gebrauchen, da hier oft die falschen Automaten nicht korrekt erkannt werden. Vereinzelt werden aber auch Werte von 0.8 und 0.748 erreicht, was zeigt, dass auch hier noch mehr möglich ist. Im 5 Klassen Szenario hingegen wird größtenteils nur eine Accuracy im Bereich von 0.5 erreicht. Das zeigt, dass die Modelle noch weiter angepasst und optimiert werden müssen. Trotzdem gibt es hier ein verhältnismäßig gutes Resultat mit einer Accuracy von 0.657 bei den R-GCN Modellen. Die Performance der Modelle scheint demnach weiter ausbaubar zu sein. Vor allem haben die Modelle Probleme dabei, die Lösungen korrekt einzustufen, welche leichte oder grobe Fehler haben und welche falsch sind. Die richtigen Lösungen hingegen werden zuverlässig gefunden und entsprechend zugeordnet. Ebenfalls auffällig ist, dass die R-GCN Modelle in den meisten Fällen eine bessere Performance aufweisen, als die MPNN Modelle. Die insgesamt besten Resultate wurden auch von den R-GCN Modelle erzielt. Da im Rahmen dieser Arbeit die Modelle nur begrenzt optimiert wurden, bietet sich hier noch eine Möglichkeit, das verwendete MPNN Modell zu verbessern. Ein interessanter Ansatz für die Überarbeitung des MPNNs ist z.B., dass die Funktion für die Verarbeitung der Merkmale der Kanten noch weiter untersucht und ausgebaut wird. Eine weitere interessante Beobachtung ist, dass für das 2 Klassen und das 3 Klassen Szenario bereits gute Resultate mit einer sehr geringen Anzahl von 10 Epochen erreicht wurden. Es scheint, dass die GNNs für diese beiden Szenarien in der Lage sind, mit der entsprechenden

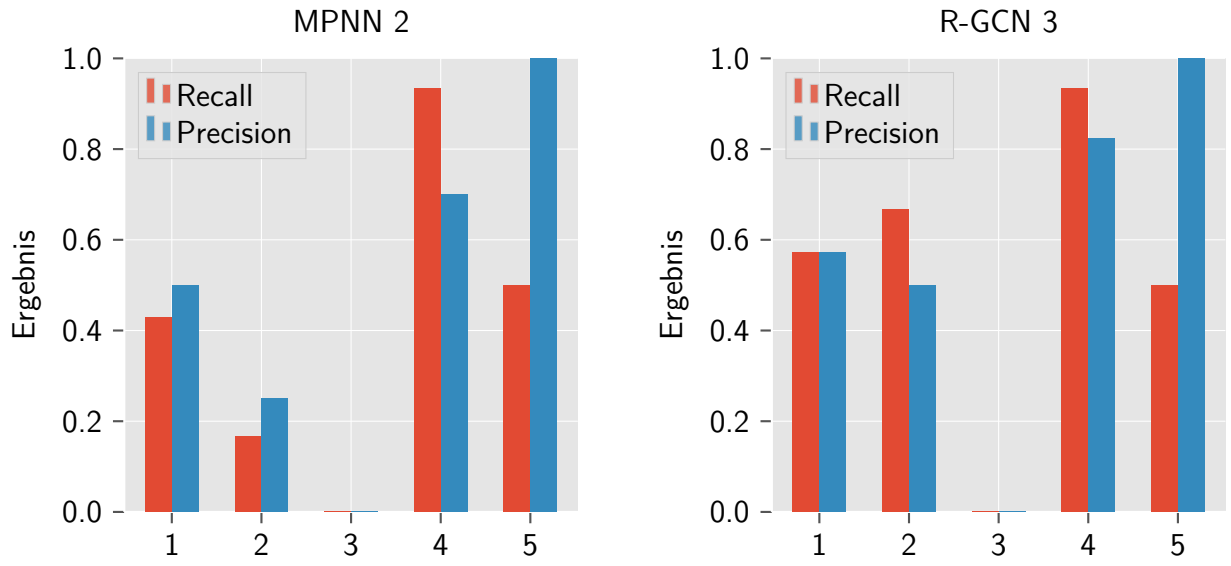


Abbildung 27: Precision und Recall der einzelnen Klassen für das 5 Klassenszenario für die Modelle mit höchsten Accuracy. Auf der linken Seiten sind die Werte des MPNN Modells dargestellt und auf der rechten Seite des R-GCN.

Konfiguration bei einer geringen Anzahl von Epochen das Problem bereits gut zu approximieren. Dabei ist jedoch zu erwähnen, dass im 5 Klassen Szenario die erreichten Ergebnisse für 10 Epochen am schlechtesten sind. Insgesamt kann also das Fazit gezogen werden, dass die GNNs in der Lage sind, eine Repräsentation zu finden, welche die richtigen Eigenschaften erfassen, um einen endlichen Automaten nach dem Bewertungsschema zu klassifizieren. Wie viele Informationen in dieser Repräsentation codiert werden und auch genutzt werden können, muss noch weiter untersucht werden, wie die Ergebnisse für 5 Klassen zeigen. Des Weiteren muss die Performance für die Szenarien 2 und 1 für die Klassen, welche die falschen Automaten oder Automaten mit kleinen oder größeren Fehlern enthalten, verbessert werden. Eine Idee ist es, die GNNs in ein Lerntool einzubauen, in welchem das GNN Feedback für die Eingabe des Studierenden gibt. Das Feedback könnte dabei zunächst die Einstufung der Klasse sein. Bei dem Feedback ist es wichtiger, eine falsche Lösung zuverlässig zu erkennen als eine eine richtige Lösung, da sonst ein Studierender durch das Feedback eine falsche Lösung als richtig auffasst.

7.1 Verbesserungen, Probleme und Einschränkungen

In verschiedenen Aspekten der Arbeit gibt es Möglichkeiten zur Verbesserung der Performance, Probleme welche noch gelöst werden und Einschränkungen die noch überwunden werden müssen. Im Folgenden werden einige dieser Punkte diskutiert.

Kleiner Datensatz Der gesamte Datensatz ist mit 172 Datenpunkten recht klein. Dies fällt beim 2 Klassen Szenario noch nicht sehr ins Gewicht, da beide Klassen ausreichend vertreten sind. Schon bei 3 Klassen zeigt sich dann aber, dass einzelne Klassen unterrepräsentiert sind. Das führt dazu, dass die GNNs für diese Klassen nur wenige Möglichkeiten haben, die notwendigen

Informationen aus den Datenpunkten zu erlernen. Ebenfalls kann die geringe Anzahl an Datenpunkten kombiniert mit einer langen Trainingszeit zum Overfitting führen, da die Netze dann die Ergebnisse auswendig lernen. Die niedrigen Ergebnisse der Accuracy bei 5 Klassen zeigen jedoch, dass dies in den durchgeführten Experimenten nicht der Fall ist. Eine Möglichkeit, dieses Problem zu lösen, wäre es, die vorhanden Daten zu vermehren, indem Kanten und Knoten hinzugefügt oder entfernt werden, ohne dass die Sprache des Automaten oder der Zielwert des Datenpunktes sich verändert.

Verkleinern und intensivere Druchsuchung des Suchraums In dem beschriebenen Setup aus Abschnitt 5 wird der Suchraum mittels dem Ziehen von Stichproben untersucht. Dies ermöglicht es mit einem geringen Zeitaufwand verschiedenste Parameter zu testen und ein Gefühl für den richtigen Bereich im Suchraum zu finden. Um die Parameter weiter zu optimieren, muss der Suchraum weiter eingeschränkt und anschließend mit einer Gridsearch durchsucht werden.

Testen weiterer Architekturen Wie beim Suchraum ist auch die Architektur des GNNs weiter zu untersuchen. Dort sind die Anzahl der Schichten zur Berechnung von $\mathtt{h}_v^L$ und auch die Anzahl der Schichten des NN für die Berechnung von $\mathtt{g}$ zwei interessante Parameter, welche noch weiter untersucht werden müssen. Dazu muss die Implementation aus Abschnitt 5 des R-GCN und des MPNN Modells so erweitert werden, dass die Schichten dynamisch konfiguriert und angepasst werden können. Dadurch ist es dann möglich, einfach verschiedene Architekturen zu testen und auf ihre Performance zu überprüfen. In der Umsetzung des MPNN Modells ist auch der Aufbau der Funktion $\mathtt{f}_\theta$ noch zu untersuchen. In der derzeitigen Umsetzung wird ein einschichtiges NN verwendet. Bei diesem kann auch die Auswirkung von verschiedenen Anzahlen von Schichten auf die Performance untersucht werden. Ein interessanter Ansatz ist die *Principal Neighbourhood Aggregation* aus [5], welcher ausprobiert werden sollte. Alternativ ist eine Umsetzung mit anderen Techniken als GNNs in Erwägung zu ziehen, wie z.B. das Finden von *Graph Embeddings* und deren Weiterverarbeitung mit Modellen des MLs [13].

Generalisierung des Problems Ein Nachteil der entwickelten Modelle ist, dass diese auf eine spezielle Aufgabe trainiert werden. Ein Netz kann also nur Abgaben für diese Aufgaben bewerten. Um Modelle für weitere Aufgaben zu erstellen, muss der in dieser Arbeit beschriebene Prozess wiederholt werden. Das ist ein aufwendiger Prozess, welcher vor allem in der Akquisition und dem Labeln der Daten viel Zeit beansprucht. Ein Modell, welches für mehrere Aufgaben genutzt werden kann, würde diesen Prozess erleichtern. Ebenfalls würde dies die Problematik des kleinen Datensatzes lösen, da dann die Daten aller Aufgaben genutzt werden könnten. Eine Idee für diesen Ansatz wäre es, die Eingabe des Netzes auf zwei Automaten abzuändern. Dort stellt der erste Automat die zu korrigierende Abgabe dar und der zweite Automat die Musterlösung. Das Netz würde dann die Automaten miteinander vergleichen und auf Basis dessen die Abgabe bewerten. Dadurch wäre das Modell von der Aufgabe gelöst. Ein weiteres Problem ist noch die Größe des verwendeten Alphabets der Automaten. Diese bestimmt die Dimension der Merkmale der Knoten und die Anzahl der Relationstypen der Kanten, welche in der Vorverarbeitung im Abschnitt 5 vorgestellt wurden. Die Architektur der in dieser Arbeit vorgestellten Modelle benötigen für diese beiden Parameter einen festgelegten Wert. Damit müssen die Automaten der Abgaben ein Alphabet besitzen, welches maximal so groß ist wie in der Architektur festgelegt.

8 Fazit und Ausblick

Durch diese Arbeit wurde ein Weg gefunden, Abgaben in Form von endlichen Automaten mit Techniken des MLs zu bewerten. Die Resultate sind dabei vielversprechend und motivieren dazu, diese Herangehensweise weiter zu untersuchen. Ebenfalls konnten durch diese Arbeit Teile einer Infrastruktur geschaffen werden, um endliche Automaten aufzubereiten, zu labeln und mit diesen ein GNN zu trainieren. Lediglich die Architektur des GNNs muss für andere Aufgaben neu erstellt werden. Die Arbeit stellt damit eine gute Grundlage für weitere Projekte und Untersuchungen dar und verbindet die Themenbereiche Softwareentwicklung, Theoretische Informatik und Machine Learning.

Um die Erstellung des gelabelten Datensatzes zu erleichtern, ist die Webapplikation aus Abschnitt 4 entstanden. Das Modell für die binäre Bewertung der Abgaben liefert bereits zuverlässige Ergebnisse und kann in Kombination mit dem Graph Editor als ein Lerntool umgesetzt werden. In diesem kann ein Student einen Automaten modellieren und anschließend überprüfen, ob die Lösung korrekt ist. Wird die Performance auf den 3 und 5 Klassen Szenarien noch verbessert, ist auch ein individuelleres Feedback zur Lösung möglich. Des Weiteren können weitere Aufgaben aus dem Bereich der endlichen Automaten adaptiert werden, um die Vorgehensweise zu verbessern und den Pool an erlernten Aufgaben zu vergrößern. Durch die Nutzung von Graphen als Repräsentation für die Eingabedaten sind auch weitere Aufgaben aus dem Bereich der Automatentheorie wie das Erstellen von Turing Maschinen oder auch aus den Formalen Sprachen das Erstellen von Grammatiken denkbar. Insgesamt ist hier viel Potential vorhanden, um die Bewertung der Aufgaben aus den Theoretischen Informatik Veranstaltungen zu automatisieren und in Lerntools zu integrieren.

Anhang

1 Abbildungen

1.1 Von Abgaben endlicher Automaten zu einem gelabelten Datensatz

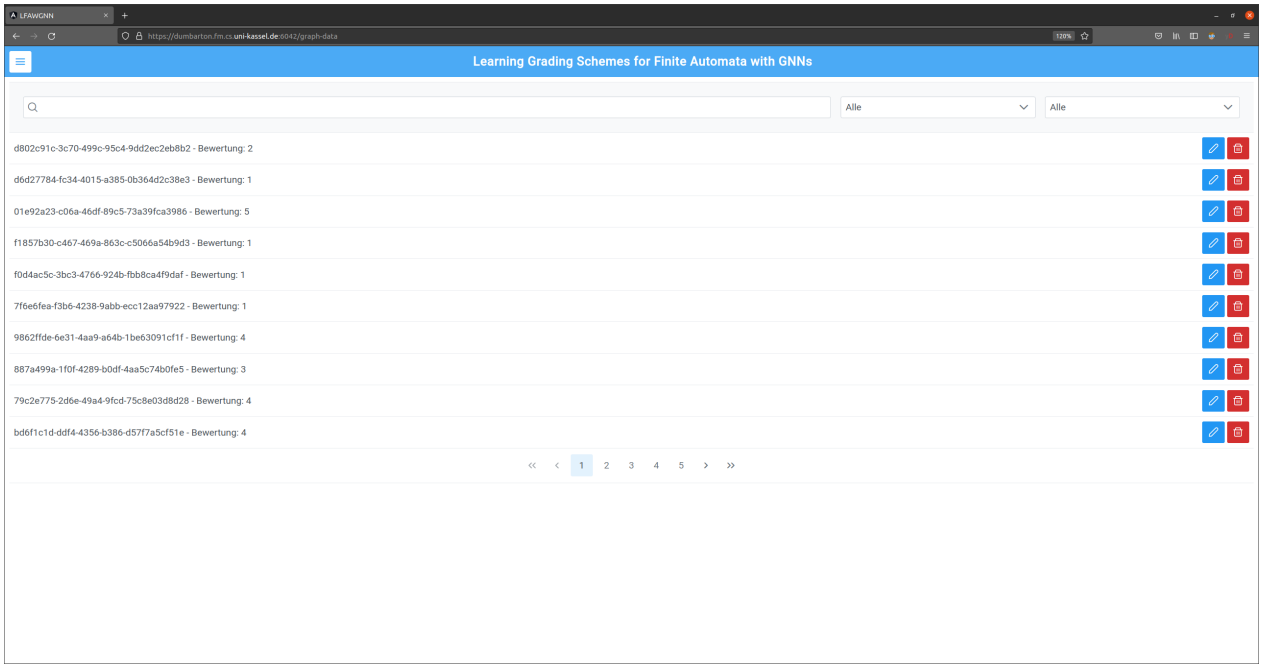


Abbildung 28: In der *Overview* Ansicht werden die Datenpunkte des Datensatzes über ein Seitensystem aufgelistet. In der Leiste über den Datenpunkten ist es möglich, die Datenpunkte über deren Namen zu filtern. Des Weitern können über die Dropdown-Menüs nach der Kategorie und den Labeln der Datenpunkte strukturiert werden. Von einem Datenpunkt wird der Name und seine Bewertung dargestellt. Über die Buttons auf der rechten Seite kann ein Datenpunkt gelöscht (rot) werden oder im Graph-Editor (blau) bearbeitet und gelabelt werden.

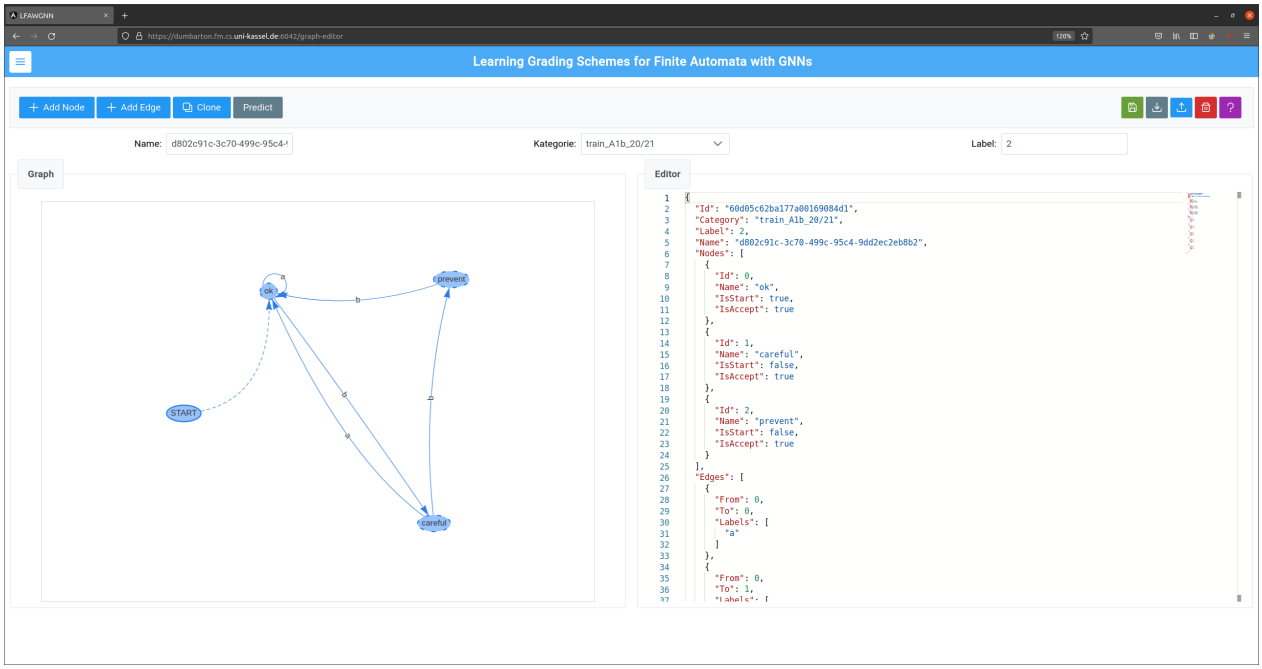


Abbildung 29: Graph Editor welcher das Erstellen, Bearbeiten, Labeln und Löschen von Datenpunken ermöglicht. In der Toolbar unter der Navigationsleiste sind zwei Arten von Operationen vorhanden. Die Buttons auf der Linke Seite ermöglichen Operationen um den Datenpunkt zu verändern, wie das Hinzufügen von Kanten und Knoten und das Duplizieren des Datenpunktes zur weiteren Bearbeitung. Mit den Buttons auf der rechten Seite kann der Datenpunkt im System verwaltet werden. Es sind die Operationen speichern (grün), herunterladen (grau), hochladen (blau), löschen (rot) und die Anzeige eines Hilfedialogs (lila) möglich. Unter der Toolbar sind Inputfelder zum Bearbeiten des Namen, der Kategorie und dem Label. Eine graphische Darstellung des Automaten ist auf der linken Seite der Ansicht zu finden. Diese wird mit *vis.js*[3] umgesetzt. Rechts daneben kann der Datenpunkt über einen Editor frei angepasst werden. Als Editor wird der *Monaco Editor*[4] verwendet. Änderungen werden in der graphischen Ansicht sofort übernommen und dargestellt.

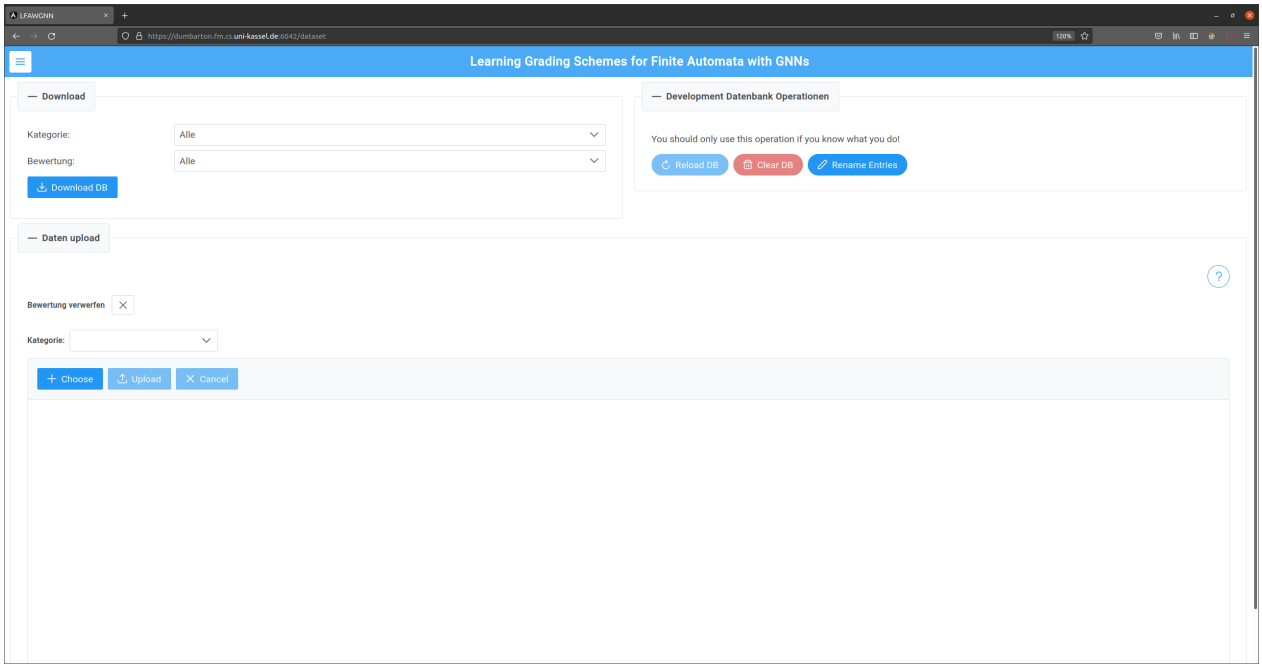


Abbildung 30: Ansicht um Operationen auf der Datenbank auszuführen. Im Bereich Download kann der Datensatz heruntergeladen werden, dabei können die Einträge um eine Kategorie und deren Bewertung gefiltert werden. Bei *Daten upload* können neue Daten hochgeladen werden, wobei beim Upload die Kategorie festgelegt werden kann sowie die Bewertung aus den hochzuladenen Daten verworfen werden. Die *Development Datenbank Operations* ermöglichen es, die Datenbank neu zu laden, die Datenbank zu löschen und die Namen der Einträge nach einem Algorithmus umzubennen.

1.2 Experimentelles Setup

```python
class FiniteAutomataDataset(DGLDataset):

    @property
    def number_classes(self) -> int:

    @property
    def number_labels(self) -> int:

    @property
    def number_node_features(self) -> int:

    @property
    def number_edge_features(self) -> int:

    @property
    def number_edge_types(self) -> int:

    def process(self) -> None:
    def save(self) -> None:
    def load(self) -> None:
    def has_cache(self) -> bool:
```

Abbildung 31: Die Funktionen und Eigenschaften des *FiniteAutomataDataset*, welche das Laden, Speichern und Vorverarbeiten des Datensatzes abstrahiert.

```
    def train(self, model: GraphClassifierBase, epochs: int,
              train_data: List[Tupel[DGLGraph, int]],
              validation_data: List[Tupel[DGLGraph, int]]):
            optimizer: Adam = Adam()
            loss_function: CrossEntropyLoss = CrossEntropyLoss()

            for epoch in epochs:
                for graph, label in train_data:
                    prediction: Tensor = model(graph)
                    loss: Tensor = loss_function(prediction, label)
                    loss.backward()
                    optimizer.step()

                self.validation(model, validation_data, loss_function)
            pass
```

Abbildung 32: Ablauf des Trainings der GNNs als Python-Pseudocode dargestellt.

```
def validation(self, model: Module, validation_data: GraphDataLoader,
       loss_function: CrossEntropyLoss) -> None:
       total_predictions: int = 0
       correct_predictions: int = 0
       summed_loss: int = 0

       for graph, labels in validation_data:
               prediction: Tensor = model(graph)
               loss: Tensor = loss_function(prediction, label)

               summed_loss += loss
               total_predictions += 1
               if self.are_same_class(prediction, label):
                   correct_predictions += 1

       accuracy: float = correct_predictions / total_predictions
       mean_loss: float = summed_loss / total_predictions
       if self.config.is_ray_tune:
           tune.report(loss=mean_loss, accuracy=accuracy)
       pass
```

Abbildung 33: Ablauf der Validierung der GNNs als Python-Pseudocode dargestellt.

```
readout_operations: List[str] =
    [ 'sum', 'max', 'min', 'mean' ]

conv_aggregators: List[str] =
    [ 'mean', 'sum', 'max' ]

config: Dict = {
    'mlp_layer_one' : tune.sample_from(lambda _: 2 ** np.random.randint(2, 9)),
    'mlp_layer_two' : tune.sample_from(lambda _: 2 ** np.random.randint(2, 9)),
    'graph_conv_one' : tune.sample_from(lambda _: 2 ** np.random.randint(2, 9)),
    'graph_conv_one' : tune.sample_from(lambda _: 2 ** np.random.randint(2, 9)),
    'learning_rate' : tune.loguniform(1e-4, 1e-1),
    'readout' : tune.choice(self.readout_operations)
}

if gnn_type is NNGraphConvolution:
    config['conv_one_aggr'] = tune.choice(self.conv_aggregators)
    config['conv_two_aggr'] = tune.choice(self.conv_aggregators)
```

Abbildung 34: Die verwendete RayTune Konfiguration für die Suche nach Hyperparametern. Die Parameter *mlp_layer_one* und *mlp_layer_two* beschreiben die Dimension der Ausgabe der $1.$ und $2.$ Schicht des NN in der Funktion g. Es werden die Zweierpotenzen von 2^2 bis 2^8 getestet. Mit $graph\_conv\_one$ und $graph\_conv\_two$ wird die Dimension der Ausgabe von h_v^1 und h^2 optimiert. Der Suchraum ist der gleiche wie bei den Schichten des NN. Die Lernrate (*learning_rate*) wird aus einer uniformen Verteilung aus dem Bereich von 0.0001 bis 0.1 gezogen. Für die Graph Readout Funktion (*readout*)wird ein Wert aus der List *readout_operations* gezogen. Der gezogene Wert bestimmt die Funktion, welche für die Berechnung des Graph Zustands $\mathtt{h_G}$ verwendet wird. Die Funktionen entsprechen dabei der Auflistung aus Abbildung 19. Die zuvor genannten Parameter werden für die beiden vorgestellten Modelle verwendet. Für das *NNGraphConvClassifier* Modell wird die Aggregationsfunktionen (*conv_one_aggr* und *conv_two_aggr*) für die Berechnung von h_v^{l+1} optimiert. Die Werte werden aus der Liste *conv_aggregators* gezogen, wobei der Wert die Funktion wie beim Graph Readout bestimmt.

```
Confusion Matrix:
pred/true      0  1  2
0           [[ 1  5  1]
1            [ 0 10  1]
2            [ 1  1 15]]

Classification Report:
              precision    recall  f1-score   support
           0       0.50      0.14      0.22         7
           1       0.62      0.91      0.74        11
           2       0.88      0.88      0.88        17
    accuracy                           0.74        35
   macro avg       0.67      0.64      0.62        35
weighted avg       0.72      0.74      0.71        35
```

Abbildung 35: Ausgabe der berechneten Metriken nach der Auswertung eines Models auf dem Testdatensatz. Als Erstes ist die Konfusionsmatrix für die Testergebnisse dargestellt. Eine Zelle C_{ij} enthält die Anzahl der Ausgaben, welche als Klasse j eingestuft wurden, aber zur Klasse i gehören. Dabei sind i die Spalten und j die Zeilen der Tabelle. Unter der Konfusionsmatrix sind die Werte für die Precision und dem Recall der einzelnen Klassen abgebildet. Die Spalte *Support* gibt an, wie viele Datenpunkte für die Berechnung verwendet wurden. Der $F1-Score$ ist eine weitere Metrik, welche aber nicht in dieser Arbeit betrachtet wird. Die Accuracy ist in der gleichnamigen Zeile gegeben. In der Zeile *weighted avg* wird die gewichtete Summe der Metrik für den Datensatz angezeigt. Das Gewicht für die Klasse i ist dabei das Verhältnis $\frac{n_i}{n}$, wobei n_i die Anzahl der Datenpunkte der Klasse i ist und n die Anzahl der Datenpunkte des Datensatzes. Die *macro avg* hingegen ist der Mittelwert der entsprechenden Metrik.

Literatur

[1] Yuichiro Anzai. *Pattern recognition and machine learning*. Elsevier, 2012.

[2] Jason Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. 2019. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (besucht am 23. 07. 2021).

[3] Jason Brownlee. *Softmax Activation Function with Python*. 2020. URL: https://machinelearningmastery.com/softmax-activation-function-with-python (besucht am 23. 07. 2021).

[4] Jason Brownlee. *Softmax Activation Function with Python*. 2021. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ (besucht am 23. 07. 2021).

[5] Gabriele Corso u. a. „Principal Neighbourhood Aggregation for Graph Nets“. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Hrsg. von Hugo Larochelle u. a. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/99cad265a1768cc2dd013f0e740300ae-Abstract.html.

[6] Loris D'Antoni u. a. „Automata Tutor v3“. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Hrsg. von Shuvendu K. Lahiri und Chao Wang. Bd. 12225. Lecture Notes in Computer Science. Springer, 2020, S. 3–14. DOI: 10.1007/978-3-030-53291-8_1. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_1.

[7] Marc Peter Deisenroth, A Aldo Faisal und Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[8] Vladan Devedzic, John K. Debenham und Dusan Popovic. „Teaching Formal Languages by an Intelligent Tutoring System“. In: *J. Educ. Technol. Soc.* 3.2 (2000). URL: http://ifets.ieee.org/periodical/vol%5C_2%5C_2000/devedzic.html.

[9] Gaetano Geck u. a. „Teaching Logic with Iltis: an Interactive, Web-Based System“. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland, UK, July 15-17, 2019*. Hrsg. von Bruce Scharlau u. a. ACM, 2019, S. 307. DOI: 10.1145/3304221.3325571. URL: https://doi.org/10.1145/3304221.3325571.

[10] Justin Gilmer u. a. „Neural Message Passing for Quantum Chemistry“. In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Hrsg. von Doina Precup und Yee Whye Teh. Bd. 70. Proceedings of Machine Learning Research. PMLR, 2017, S. 1263–1272. URL: http://proceedings.mlr.press/v70/gilmer17a.html.

[11] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning. Das umfassende Handbuch*. Grundlagen, aktuelle Verfahren und Algorithmen, neue Forschungsansätze. Frechen: MITP, 2018. ISBN: 978-3-95845-700-3.

[12] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep learning*. MIT press, 2016.

[13] Palash Goyal und Emilio Ferrara. „Graph embedding techniques, applications, and performance: A survey“. In: *Knowl. Based Syst.* 151 (2018), S. 78–94. DOI: 10.1016/j.knosys.2018.03.022. URL: https://doi.org/10.1016/j.knosys.2018.03.022.

[14] Norbert Hundeshagen, Martin Lange und Georg Siebert. „DiMo - Discrete Modelling Using Propositional Logic“. In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Hrsg. von Chu-Min Li und Felip Manyà. Bd. 12831. Lecture Notes in Computer Science. Springer, 2021, S. 242–250. DOI: 10.1007/978-3-030-80223-3_17. URL: https://doi.org/10.1007/978-3-030-80223-3%5C_17.

[15] Diederik P. Kingma und Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Hrsg. von Yoshua Bengio und Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[16] Zhiyuan Liu und Jie Zhou. „Introduction to graph neural networks“. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.2 (2020), S. 1–127.

[17] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[18] Michael A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[19] Nils J Nilsson. „Introduction to machine learning. An early draft of a proposed textbook“. In: (1996).

[20] Franco Scarselli u. a. „The Graph Neural Network Model“. In: *IEEE Trans. Neural Networks* 20.1 (2009), S. 61–80. DOI: 10.1109/TNN.2008.2005605. URL: https://doi.org/10.1109/TNN.2008.2005605.

[21] Michael Sejr Schlichtkrull u. a. „Modeling Relational Data with Graph Convolutional Networks“. In: *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*. Hrsg. von Aldo Gangemi u. a. Bd. 10843. Lecture Notes in Computer Science. Springer, 2018, S. 593–607. DOI: 10.1007/978-3-319-93417-4_38. URL: https://doi.org/10.1007/978-3-319-93417-4%5C_38.

[22] Uwe Schöning. *Theoretische Informatik - kurzgefasst*. 4. A. (korrig. Nachdruck 2003). Spektrum Akademischer Verlag, 2003. ISBN: 3827410991.

[23] Georg Siebert. *LFAWGNN*. 2021. URL: https://syre.fm.cs.uni-kassel.de/Georg/lfawgnn (besucht am 23. 08. 2021).

[24] DGL Team. *Deep Graph Library Documentation*. 2021. URL: https://docs.dgl.ai (besucht am 05. 08. 2021).

[25] NumPy Team. *NumPy Documentation*. 2021. URL: https://numpy.org/ (besucht am 05. 08. 2021).

[26] PyTorch Team. *PyTorch Documentation*. 2021. URL: https://pytorch.org/docs/stable/index.html (besucht am 05. 08. 2021).

[27] Ray Team. *RayTune Documentation*. 2021. URL: https://docs.ray.io/en/master/tune/index.html (besucht am 05. 08. 2021).

[28] scikit-learn Team. *scikit-learn Documentation*. 2021. URL: https://scikit-learn.org/stable/ (besucht am 05. 08. 2021).

[29] Minjie Wang u. a. *Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks*. 2020. arXiv: 1909.01315 [cs.LG].

[30] Zonghan Wu u. a. „A comprehensive survey on graph neural networks“. In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), S. 4–24.

[31] Da Zheng u. a. „Learning Graph Neural Networks with Deep Graph Library“. In: *Companion of The 2020 Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. Hrsg. von Amal El Fallah Seghrouchni u. a. ACM, 2020, S. 305–306. DOI: 10.1145/3366424.3383111. URL: https://doi.org/10.1145/3366424.3383111.

[32] Jie Zhou u. a. „Graph neural networks: A review of methods and applications“. In: *AI Open* 1 (2020), S. 57–81. DOI: 10.1016/j.aiopen.2021.01.001. URL: https://doi.org/10.1016/j.aiopen.2021.01.001.