

Duale Hochschule Baden-Württemberg Mosbach
Integrationsseminar

Analyse der Einflüsse von GCN-Layers auf Hidden Representations von numerischen Tabellenspalten

Bearbeitungszeitraum	18. April 2023 – 09. Juli 2023
Gruppenteilnehmer	Damien Arriens (Matrikel-Nr.: 4364633) Daniel Boger (Matrikel-Nr.: 6015969) Simon Di Latte (Matrikel-Nr.: 4089707) Fabian Qarqur (Matrikel-Nr.: 3655976) Julian Stipovic (Matrikel-Nr.: 8994343)
Kurs	WI20B
Betreuender Dozent	Sven Langenecker

Inhaltsverzeichnis

Abbildungsverzeichnis	III
1 Zielsetzung	1
2 Graph Neural Network (GNN)	2
3 GNN-Modell.....	4
4 Vorgehen zur Lösungsbearbeitung	7
4.1 Erste Schritte.....	7
4.2 Extrahierung und Speicherung der Vektorrepräsentationen	7
4.3 Extrahierte Vektoren mit Tabellen- und Spaltenname verknüpfen	8
4.4 Analyse der Vektorenrepräsentationen	9
5 Ergebnis der Analyse	14
5.1 Allgemeine Daten	14
5.2 PCA-Analyse	14
5.3 t-SNE-Analyse	17
5.4 Cosinus Ähnlichkeit.....	17
5.5 Vorhersagegenauigkeit semantischen Typen.....	18
6 Schlussbemerkung.....	20
6.1 Fazit	20
6.2 Ausblick und Weiterentwicklungsmöglichkeiten	20

Quellenverzeichnis

I

Ehrenwörtliche Erklärung

Abbildungsverzeichnis

Abbildung 1	Aufbau des CA-GNN.....	4
Abbildung 2	Speicherung der Tabelleninformationen für die Vektoren	8
Abbildung 3	Value Error t-SNE.....	9
Abbildung 4	Abbild vom Code einer PCA-Analyse.....	10
Abbildung 5	Abbild vom Code einer Cosinus-Ähnlichkeit.....	11
Abbildung 6	Abbild ValueError Cosinus-Ähnlichkeit	11
Abbildung 7	Abbild vom Code einer t-SNE-Analyse	12
Abbildung 8	Abbild vom Code: Bestimmung Vorhersagegenauigkeit	12
Abbildung 9	Abbild vom Code: Speicherung der letzten Epoche	13
Abbildung 10	Zusammengefasste Beschreibung CSV-Datei	14
Abbildung 11	PCA-Analyse einer Dimension.....	15
Abbildung 12	PCA-Analyse jeweiligen Layers.....	15
Abbildung 13	PCA-Analyse mit numerischem Datentyp.....	16
Abbildung 14	t-SNE-Analyse	17
Abbildung 15	Cosinus-Ähnlichkeit Input und Output Layer.....	18
Abbildung 16	Heatmap Cosinus-Ähnlichkeit	18
Abbildung 17	Vorhersagegenauigkeit semantischen Typen.....	19

1 Zielsetzung

Das Ziel dieser Aufgabenstellung ist es, den Einfluss von GCN-Layern (Graph Convolutional Network) auf die Hidden Representations numerischer Spalten in der Tabelle zu analysieren. Diese Arbeit untersucht verschiedene Aufgaben und Probleme.

GCN-Modelle sind für ihre Komplexität bekannt, insbesondere wenn sie auf numerische Tabellenspalten angewendet werden. Die Verarbeitung von GCNs erfordert ein tiefes Verständnis des Modells und seiner Funktionsweise. Um dies zu erreichen, soll zunächst ein Verständnis für GNN entwickelt werden, ebenso wie für den bereitgestellten Code.

Eine weitere Herausforderung besteht darin, die Hidden Representations einzelner GCN-Layers zu untersuchen. Hier werden nämlich die Vektoren der numerischen Spalten in den entsprechenden Tensoren angezeigt. Diese müssen extrahiert und gespeichert werden.

Um die Vektoranalyse und -auswertung zu erleichtern, werden Diagramme verwendet, um Vektordarstellungen visuell zu vergleichen. Solche Darstellungen geben Einblick in die Unterschiede und Gemeinsamkeiten zwischen verschiedenen Schichten.

Das Hauptziel besteht darin, die Vorhersagegenauigkeit des semantischen Typs anhand der analysierten Vektoren zu bewerten. Dabei dienen die Vektoren als Grundlage für die Vorhersage des semantischen Typs einer Spalte. Die Herausforderung besteht darin, die Genauigkeit dieser Vorhersagen zu analysieren und zu bewerten.

Durch die Analyse der Einflüsse eines GCN-Layers auf Hidden Representations von numerischen Tabellenspalten sowie die Bewertung der Vorhersagegenauigkeit des semantischen Typen können wertvolle Erkenntnisse gewonnen werden.

2 Graph Neural Network (GNN)

Üblicherweise werden Datensätze von Deep-Learning-Anwendungen im euklidischen Raum repräsentiert. Allerdings gibt es eine wachsende Anzahl von Daten, die nicht-euklidisch sind und als Graphen dargestellt werden.

GNN ist eine Klasse von verschiedenen Algorithmen des maschinellen Lernens, die für den Umgang mit graphenartigen Datenstrukturen entwickelt wurden. Ein Graph besteht aus einer Menge von Objekten (Knoten) und ihren Beziehungen zueinander (Kanten). Sowohl die Knoten als auch die Kanten verfügen über Features. Es existieren verschiedene Graphentypen, wie gerichtete oder ungerichtete Graphen. Je nach GNN-Modell können sie entweder auf einen spezifischen Graphentyp oder auf alle Arten von Graphen angewendet werden.

GNNs ermöglichen es, Informationen über diese Beziehungen zu erfassen und zu verarbeiten, um Muster und komplexe Beziehungen in den Daten zu erkennen, wie die Klassifizierung von Knoten (vgl. [Karagiannakos (2021)]). Ein neuronales Graphennetz besteht aus mehreren Schichten von Neuronen, die auf den Knoten und Kanten eines Graphen arbeiten. Die Eingaben für ein GNN sind eine Adjazenzmatrix, die die Verbindungen zwischen den Knoten darstellt sowie einer Knotenmerkmalmatrix, die Informationen über die Eigenschaften der Knoten enthält. In jeder Schicht des GNN werden die Merkmale der Knoten und Kanten durch Aggregation, Transformation und Umverteilung aktualisiert.

Die Kernidee hinter GNN ist die Nachrichtenübertragung zwischen den Knoten im Graphen. Jeder Knoten sammelt Informationen von seinen Nachbarknoten und kombiniert sie mit seinen eigenen Merkmalen, um eine aktualisierte Darstellung zu erzeugen. Diese aktualisierten Merkmale werden dann an die nächste Schicht weitergegeben. Durch wiederholte Anwendung dieses Prozesses können GNNs Informationen aus dem gesamten Graphen sammeln und komplexe Muster erkennen (vgl. [Pohl (2023)]).

GNNs wurden bereits erfolgreich in einer Vielzahl von Anwendungen eingesetzt. Sie wurden beispielsweise in der Computer Vision zur Verbesserung der Objekterkennung in Bildern und in der Molekularchemie zur Analyse von Verbindungen eingesetzt. Sie können auch in Empfehlungssystemen eingesetzt werden, um das Nutzerverhalten in sozialen Netzwerken zu modellieren und personalisierte Empfehlungen zu erstellen (vgl. [IONOS SE (2020)]).

Context-aware Graph Neural Networks (CAGNN) stellen eine Weiterentwicklung von GNNs dar, die zusätzlich zu den Grafiken auch den Kontext berücksichtigen. Der Kontext kann externe Informationen wie Zeit, Ort oder Benutzerkontext umfassen. CAGNN berücksichtigt diesen Kontext, um die Leistung des Modells in dynamischen oder personalisierten Umgebungen zu verbessern. Die Integration von Kontext in GNNs erfordert häufig eine Erweiterung der grafischen Darstellung, um den Kontext zu berücksichtigen (vgl. [Zhu et al. (2020)]).

3 GNN-Modell

In Abbildung eins ist ein CA-GNN-Modell ersichtlich, welches für diese Arbeit genutzt wird.

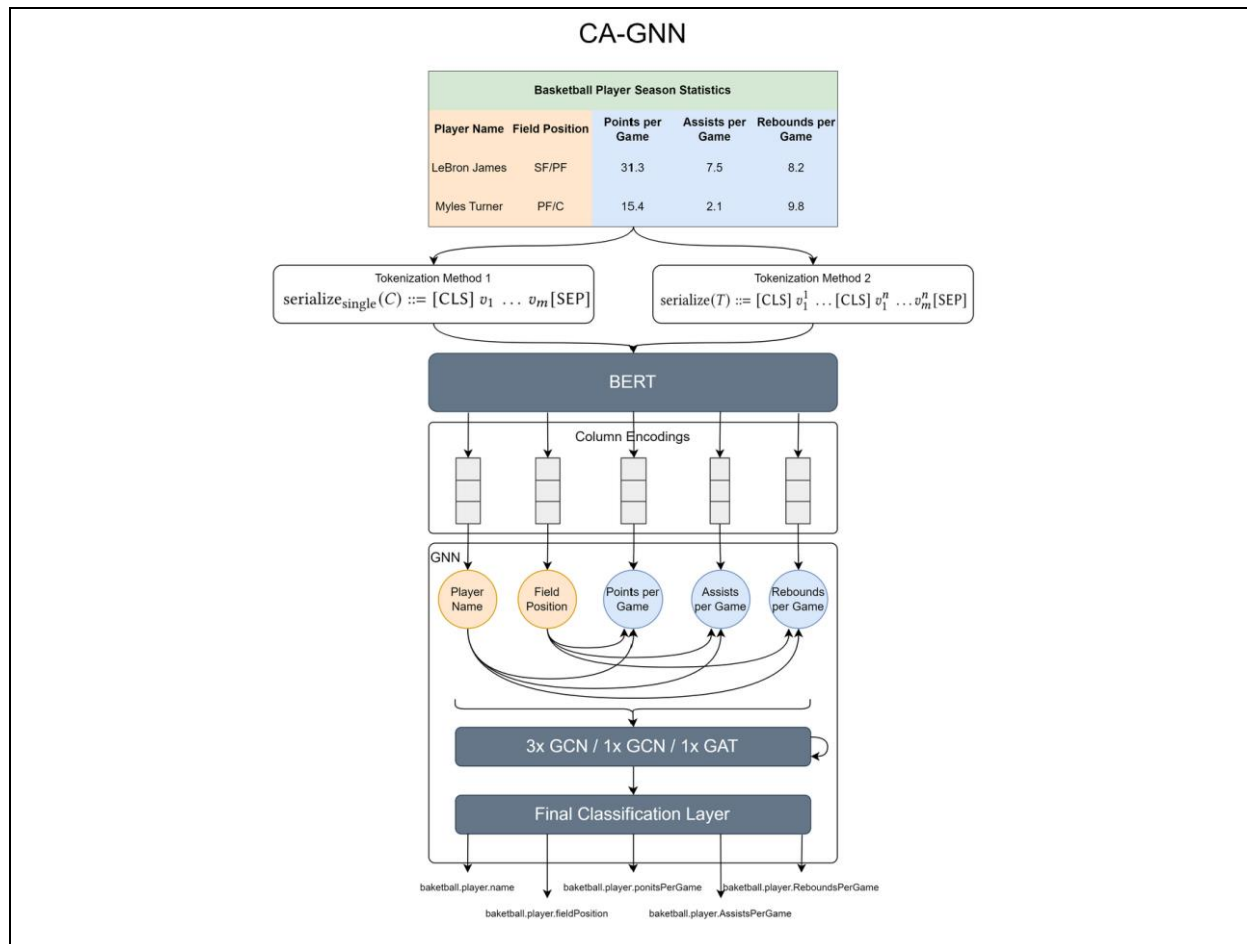


Abbildung 1 Aufbau des CA-GNN
(Quelle: Moodle)

Das Modell ist eine Anwendung für ein GNN, genauer gesagt ein GCN oder ein Graph Attention Network (GAT). Das Modell nutzt fortschrittliche Techniken wie die BERT-Tokenisierung und Spaltencodierung, um Basketballspielerstatistiken zu analysieren. Im folgenden Abschnitt wird dieses Model genauer erklärt und deren Funktionen erläutert.

Zunächst werden die Eingaben, wie der Name des Spielers, seine Position auf dem Feld und seine Spielstatistiken, tokenisiert. Das bedeutet, dass der Text in kleinere Einheiten (Tokens)

aufgeteilt wird, um eine bessere Verarbeitung zu ermöglichen. Hier kommt BERT ins Spiel, ein Verfahren zur Texttokenisierung.

Das Modell verwendet entweder Methode eins oder Methode zwei, um die Texte zu zerlegen. Nach der Tokenisierung folgt eine Spaltencodierung. Dabei werden die unterschiedlichen Datenarten in den Spalten, wie Spielernamen, Position auf dem Feld und verschiedene Spielstatistiken, so repräsentiert, dass sie von den nachfolgenden Modellschichten verarbeitet werden können. Diese Schicht ermöglicht es dem Modell, die relevanten Informationen aus den Spalten zu extrahieren und zu verstehen.

Nun werden die codierten Daten in ein GNN eingespeist. In unserem Fall werden die Basketballspieler und ihre Beziehungen zueinander als Graph betrachtet. Die Spieler werden als Knoten repräsentiert, während die Kanten die Beziehungen zwischen den Spielern darstellen. Diese Beziehungen können auf ihren Statistiken oder ihrer Position auf dem Feld basieren.

Es gibt verschiedene Optionen für die Art des verwendeten GNN, nämlich 3 x GCN, 1 x GCN oder 1 x GAT. GCNs ähneln CNNs, sind jedoch speziell für die Verarbeitung von Graphdaten entwickelt worden. Sie ermöglichen es dem Modell, wichtige Muster und Informationen in den Spielerdaten zu erkennen. GATs hingegen nutzen Aufmerksamkeitsmechanismen, um zu bestimmen, wie die Knoten im Graphen miteinander interagieren. Sie können die Bedeutung und Relevanz bestimmter Spieler in Bezug auf das Gesamtsystem hervorheben.

Schließlich gibt es eine Klassifikationsschicht, die auf Grundlage der Ausgabe des GNN eine endgültige Vorhersage oder Klassifikation erstellt. Diese Schicht ermöglicht es dem Modell, eine Entscheidung zu treffen oder eine Kategorie zuzuweisen, basierend auf den analysierten Basketballspielerstatistiken.

Unterschied GCNs und GATs

GCNs und GATs sind zwei leistungsstarke Algorithmen im Bereich des maschinellen Lernens, die speziell für die Verarbeitung von Daten in Graphenstrukturen entwickelt wurden. In dieser Erläuterung werden die beiden Modelle näher betrachtet und ihre Unterschiede sowie ihre Anwendungsbereiche beleuchtet.

GCNs ist einer der häufig verwendeten Architekturen der GNN-Modelle, die darauf abzielen, Informationen aus komplexen Datenstrukturen wie Graphen zu extrahieren und zu verarbeiten. Ein GCN besteht in der Regel aus einer Vielzahl von Schichten, die sogenannte "Graph

Convolutions" durchführen. Die Kernidee ist, dass jeder Knoten im Graphen seine Informationen aggregiert und mit seinen Nachbarn teilt, um eine verbesserte Darstellung zu erhalten. Diese Aggregation basiert häufig auf einer Funktion, die die Merkmale der Nachbarn gewichtet (vgl. [Pohl (2023)]).

GATs sind eine weitere Variante von GNNs. Der Hauptunterschied zwischen GATs und GCNs liegt in der Art und Weise, wie Informationen zwischen den Knoten des Graphen aggregiert werden. Im Gegensatz zu GCNs, die eine gewichtete Summation der Nachbarn verwenden, um Informationen zu aggregieren, führen GATs eine Attention-Mechanismus-basierte Aggregation durch. Das bedeutet, dass jeder Knoten in einem GAT einen Gewichtungsfaktor für seine Nachbarn berechnet. Diese Gewichtungsfaktoren werden mithilfe einer Attention-Funktion erzeugt, die die Bedeutung der Nachbarn für den zentralen Knoten bestimmt. Die gewichteten Merkmale der Nachbarn werden dann gewichtet summiert, um die neue Darstellung des Knotens zu erstellen.

Die wesentlichen Unterschiede zwischen GCN und GAT lassen sich wie folgt zusammenfassen:

- Aggregationsmethode: GCN verwendet gewichtete Summation, während GAT einen Attention-Mechanismus verwendet, um die Informationen der Nachbarn zu gewichten.
- Flexibilität: Aufgrund des Attention-Mechanismus ermöglichen GATs eine flexiblere und feinere Gewichtung der Nachbarn im Vergleich zu GCNs.
- Skalierbarkeit: GATs sind in der Regel aufgrund des höheren Rechenaufwands durch die Attention-Mechanismus-basierte Aggregation rechenintensiver als GCNs. Dies kann zu Problemen bei großen Graphen führen (vgl. [Karagiannakos (2021)]).

GCNs und GATs sind leistungsstarke Modelle zur Verarbeitung von Graphendaten. Während GCNs Informationen von allen Nachbarknoten gleich behandeln, ermöglichen GATs eine feinere Gewichtung und Kontrolle der Beiträge einzelner Nachbarn durch den Einsatz eines Aufmerksamkeitsmechanismus. Die Wahl zwischen GCNs und GATs hängt von der spezifischen Aufgabe und den Anforderungen ab, wobei GATs in bestimmten Szenarien eine bessere Leistung bieten können. Die fortlaufende Forschung in diesem Bereich trägt zur Entwicklung immer fortschrittlicherer Modelle bei, die uns helfen, komplexe Strukturen und Beziehungen in Graphen besser zu verstehen und zu analysieren.

4 Vorgehen zur Lösungsbearbeitung

4.1 Erste Schritte

Bei unserem ersten Treffen haben wir die Rahmenbedingungen festgelegt, wie wir in diesem Projekt vorgehen. Es wurde vereinbart, uns mindestens einmal pro Woche zu treffen, vorwiegend über Discord. Diese Treffen dienten dazu, den aktuellen Stand jedes Einzelnen zu besprechen und gemeinsam Probleme zu lösen. Die Aufgaben wurden nach Möglichkeit aufgeteilt, wobei es jedoch gelegentlich erforderlich war, sie gemeinsam zu erledigen.

Wir haben das Projekt damit gestartet, dass sich jeder ein theoretisches Verständnis über das GNN aneignet und sich mit dem uns zur Verfügung gestellten Code vertraut macht.

4.2 Extrahierung und Speicherung der Vektorrepräsentationen

Im nächsten Schritt benötigten wir die Vektorrepräsentationen, um diese zu analysieren. Dafür haben wir uns zunächst im Code für das GCN-Modell entschieden. Das Modell besteht aus zwei Hidden Layers sowie einem Input und Output Layer. Durch die Ausgabe der Tensoren von `pooled_output`, `h_one`, `h_two` und `gnn_output` wurde deutlich, dass diese die Vektoren enthalten. Die Anzahl der Vektoren, die dem Modell zugeführt werden, hängt von der Anzahl der Spalten der trainierten Tabelle ab. Das bedeutet, wenn eine Tabelle mit fünf Spalten in das Modell eingespeist wird, werden fünf Vektoren in einem Tensor erzeugt. Hat die Tabelle mehr Spalten, sind mehr Vektoren im Tensor enthalten.

Das Problem bei der Extrahierung war, die Vektoren nicht innerhalb des Tensors in eine Liste zu extrahieren, sondern den Tensor aufzuteilen, um die einzelnen Vektoren in die Liste zu bekommen. Zunächst haben wir die Vektoren der jeweiligen Schichten in Listen abgelegt. Mit einer `for`-Schleife haben wir dann die Vektoren aus den jeweiligen Listen in eine Liste eingefügt. Dieser Schritt war notwendig, damit die Vektoren in der richtigen Reihenfolge in der CSV-Datei abgespeichert werden.

Nachdem die Vektoren in eine Liste extrahiert wurden, müssen sie in eine CSV-Datei abgespeichert werden. Dazu haben wir zunächst dem Code mitgegeben, dass eine CSV-Datei erstellt werden soll, falls noch keine vorhanden ist und die Überschriften miteingefügt werden sollen. Ist die CSV-Datei erstellt und die Vektoren extrahiert, werden diese in die CSV-Datei geschrieben. Bei diesem Vorgang hatten wir oft das Problem, dass die Vektoren nicht richtig in die Datei gespeichert wurden.

4.3 Extrahierte Vektoren mit Tabellen- und Spaltenname verknüpfen

Mit den gespeicherten Vektoren konnten wir beginnen, diese zu analysieren. Indem wir die Gruppe in zwei Bereiche aufgeteilt haben, konnte eine Gruppe bereits mit der Analyse beginnen, während sich die andere zunächst darum kümmerte, weitere Informationen in die CSV-Datei einzufügen. Um eine Bedeutung zu erhalten, von welcher Tabelle und Spalte der jeweilige Vektor stammt, mussten wir die Tabellen- und Spaltennamen, den Datentyp sowie die Labels der jeweiligen Spalte in die CSV-Datei einfügen. Dabei sind wir auf das Problem gestoßen, dass eine Fehlermeldung erschien, die besagte, dass der `batch_idx` nicht definiert ist, wenn wir dem Modell zusätzliche Informationen geben wollten. Nach verschiedenen Versuchen konnten wir den Fehler nicht beheben.

Um uns stärker auf die Analyse konzentrieren zu können, entschieden wir uns für eine eher unelegante Lösung. Wir haben eine weitere CSV-Datei nach der Datenaufbereitung erstellt, in der nur die benötigten Informationen enthalten sind. Wie in Abbildung zwei ersichtlich ist, wird im Code eine neue CSV-Datei erstellt und die Daten werden darin gespeichert.

```
extracted_list = self.df[["table_name", "column_name", "columns_data_type", "column_label"]].values.tolist()
with open("/ext/daten-wi/wi20b/column_annotation_gnn/extracted_vector/output_tabledata_test.csv", "a", newline="") as csvfile:
    for i in range(10):
        writer = csv.writer(csvfile)
        writer.writerow(extracted_list)
```

Abbildung 2 Speicherung der Tabelleninformationen für die Vektoren
(Quelle: SportsDB_data_loader.py)

Da es zehn Epochen gab, haben wir diese zunächst anhand einer for-Schleife zehnmal durchlaufen lassen, damit später alle Vektoren ihre jeweilige Information erhalten. Daraufhin konnten wir nach dem Trainingsprozess die Vektorentabelle mit der anderen Tabelle matchen und weitere Analysen beginnen. Dabei ist wichtig zu beachten, dass die Vektoren und die Informationen in der gleichen Reihenfolge vorliegen, da diese Methode sonst nicht funktioniert hätte.

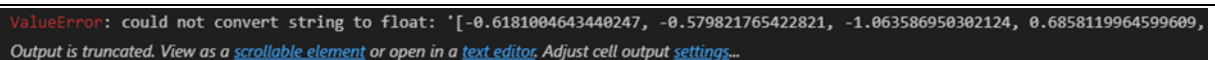
Nachdem wir eine vollständige CSV-Datei haben, mit den extrahierten Vektoren sowie den wichtigen Bezeichnungen, wie beispielsweise den Tabellennamen und Spaltennamen, konnten wir uns komplett auf die Analyse konzentrieren.

Durch die Nutzung der CPU statt von CUDA haben die Schritte 4.2 und 4.3 bei der Durchführung mehrere Stunden gedauert. Mit der Nutzung von CUDA hätte das Modell schneller ausgeführt werden können. Aufgrund von Fehlern konnten jedoch CUDA nicht aktiviert werden.

4.4 Analyse der Vektorenrepräsentationen

Bei der Analyse hatten wir zunächst Schwierigkeiten herauszufinden, welche Analysemöglichkeiten wir nutzen können, um die Vektorenrepräsentation zu untersuchen. Im Internet haben wir versucht, einen Überblick darüber zu bekommen, welche Visualisierungen sinnvoll wären, insbesondere bei einer großen Anzahl von Werten. Dabei wurden häufig t-SNE und Cosinus-Ähnlichkeit genannt.

Unser erster Ansatz, um einen Einblick in die Daten zu erhalten, war die Verwendung des t-SNE (t-distributed stochastic neighbor embedding). T-SNE ist eine verbreitete Methode zur Visualisierung hochdimensionaler Daten und liefert oft effektive Ergebnisse. In unserem Fall stießen wir jedoch zunächst auf wiederkehrende Probleme, die eine erfolgreiche Analyse behinderten.



```
ValueError: could not convert string to float: '[-0.6181004643440247, -0.579821765422821, -1.063586950302124, 0.6858119964599609, ...]'
```

Abbildung 3 Value Error t-SNE
(Quelle: analysis_vector.ipynb)

Der in Abbildung drei gezeigte Fehler ist nur einer von vielen, die dabei aufgetreten sind. Aufgrund des zeitlichen Drucks zur Abgabefrist haben wir uns entschieden, eine andere Analyse-methode zu verwenden, nämlich die Hauptkomponentenanalyse Principal Component Analysis (PCA).

```
# PCA analysis all layers
fig, ax = plt.subplots(figsize=(8, 6))

for i, col in enumerate(col_names):
    data = df[col].apply(parse_list).values
    vectors = np.array([np.array(x) for x in data])
    pca = PCA(n_components=2)
    transformed_data = pca.fit_transform(vectors)
    ax.scatter(transformed_data[:, 0], transformed_data[:, 1], color=colors[i], label=col)

ax.legend()
ax.set_title("PCA Scatterplot für die vier Layers")
plt.show()
```

Abbildung 4 Abbild vom Code einer PCA-Analyse
(Quelle: analysis_vector.ipynb)

Die PCA ist ein statistisches Verfahren, das dazu dient, die wesentlichen Merkmale in unseren Daten zu identifizieren. Durch das Verschieben der Perspektive auf die Daten kann die PCA wichtige Muster oder Merkmale hervorheben, die als "Hauptkomponenten" bezeichnet werden.

Unsere Anwendung der PCA ging über die üblichen Verfahren hinaus, indem wir die Methode auf einzelne Vektoren innerhalb einer Zeile eines bestimmten Layers angewendet haben. Jede Zeile dieses Layers umfasst mehrere Vektoren, die alle in unserer Analyse berücksichtigt wurden. In Abbildung vier ist ein Ausschnitt einer PCA-Analyse ersichtlich, bei der für jeden Layer eine Farbe definiert wird und die Vektoren in einen anderen Datentyp umgewandelt werden, da diese als Strings betrachtet werden.

Um die Zusammenhänge der Vektoren zwischen den Layers zu betrachten, haben wir die Cosinus-Ähnlichkeit angewendet. Die Cosinus-Ähnlichkeit wird verwendet, um die Ähnlichkeit zwischen zwei Vektoren darzustellen. Hierbei liefert sie einen Wert zwischen -1 und 1 zurück, wobei 1 eine perfekte Übereinstimmung darstellt.

```
input_vectors = []
hidden_layer1_vectors = []
hidden_layer2_vectors = []
output_vectors = []

for input_row, hidden1_row, hidden2_row, output_row in zip(df['input_layer'], df['hidden_layer1'], df['hidden_layer2'], df['output_layer']):
    input_vector_str = parse_list(input_row)
    input_vector = [float(x) for x in input_vector_str]
    input_vectors.append(input_vector[:300])

    hidden1_vector_str = parse_list(hidden1_row)
    hidden1_vector = [float(x) for x in hidden1_vector_str]
    hidden_layer1_vectors.append(hidden1_vector[:300])

    hidden2_vector_str = parse_list(hidden2_row)
    hidden2_vector = [float(x) for x in hidden2_vector_str]
    hidden_layer2_vectors.append(hidden2_vector[:300])

    output_vector_str = parse_list(output_row)
    output_vector = [float(x) for x in output_vector_str]
    output_vectors.append(output_vector[:300])

input_vectors = np.array(input_vectors)
hidden_layer1_vectors = np.array(hidden_layer1_vectors)
hidden_layer2_vectors = np.array(hidden_layer2_vectors)
output_vectors = np.array(output_vectors)

similarities_input_output = cosine_similarity(input_vectors, output_vectors)
similarities_hidden1_output = cosine_similarity(hidden_layer1_vectors, output_vectors)
similarities_hidden2_output = cosine_similarity(hidden_layer2_vectors, output_vectors)

print("Similarities Input-Output:")
print(similarities_input_output)
print()

print("Similarities HiddenLayer1-Output:")
print(similarities_hidden1_output)
print()

print("Similarities HiddenLayer2-Output:")
print(similarities_hidden2_output)
print()
```

Abbildung 5 Abbild vom Code einer Cosinus-Ähnlichkeit
(Quelle: analysis_vector.ipynb)

In Abbildung fünf ist ersichtlich, dass wir die Betrachtung der Vektoren auf 300 Zeilen beschränken mussten, um die Ähnlichkeit zu bestimmen. Diese Entscheidung wurde getroffen, da ein ValueError als Fehlermeldung angezeigt wurde, der darauf hinwies, dass die Matrizen nicht korrekt dargestellt sind. Während unserer Arbeit konnten wir diesen Fehler nicht beheben.

```
ValueError: Incompatible dimension for X and Y matrices: X.shape[1] == 768 while Y.shape[1] == 462
```

Abbildung 6 Abbild ValueError Cosinus-Ähnlichkeit
(Quelle: analysis_vector.ipynb)

```
fig, ax = plt.subplots(figsize=(8, 6))

for i, col in enumerate(col_names):
    data = df[col].apply(parse_list).values
    vectors = np.array([np.array(x) for x in data])
    tsne = TSNE(n_components=2, random_state=42)
    transformed_data = tsne.fit_transform(vectors)
    ax.scatter(transformed_data[:, 0], transformed_data[:, 1], color=colors[i], label=col)

ax.legend()
ax.set_title("t-SNE Scatterplot für die vier Layer")
plt.show()
```

Abbildung 7 Abbild vom Code einer t-SNE-Analyse
(Quelle: analysis_vector.ipynb)

Als letzte Visualisierungsmöglichkeit haben wir es noch einmal mit t-SNE versucht und konnten nach vielen Versuchen die Grafik darstellen. Wie in Abbildung sieben zu sehen ist, ist der Code für t-SNE ähnlich aufgebaut wie die PCA-Analyse.

Nach der Analyse der Vektoren aus den einzelnen Schichten wollten wir die Analyse der Vektorrepräsentationen von numerischen Spalten in den verschiedenen GCN-Layern im Hinblick auf die Vorhersagegenauigkeit des semantischen Typs durchführen.

```
X_train, X_test, y_train, y_test = train_test_split(numerical_vectors_input, labels_input, test_size=0.2, random_state=20)

classifier_input = RandomForestClassifier()
classifier_input.fit(X_train, y_train)
predictions_input = classifier_input.predict(X_test)
accuracy_input = accuracy_score(y_test, predictions_input)

print("Metric für input_layer:")
print(f"Accuracy: {accuracy_input}")

classifier_layer1 = RandomForestClassifier()
classifier_layer1.fit(X_train, y_train)
predictions_layer1 = classifier_layer1.predict(X_test)
accuracy_layer1 = accuracy_score(y_test, predictions_layer1)

print("Metric für hidden_layer1:")
print(f"Accuracy: {accuracy_layer1}")

classifier_layer2 = RandomForestClassifier()
classifier_layer2.fit(X_train, y_train)
predictions_layer2 = classifier_layer2.predict(X_test)
accuracy_layer2 = accuracy_score(y_test, predictions_layer2)

print("Metric für hidden_layer2:")
print(f"Accuracy: {accuracy_layer2}")

classifier_output = RandomForestClassifier()
classifier_output.fit(X_train, y_train)
predictions_output = classifier_output.predict(X_test)
accuracy_output = accuracy_score(y_test, predictions_output)

print("Metric für output_layer:")
print(f"Accuracy: {accuracy_output}")
```

Abbildung 8 Abbild vom Code: Bestimmung Vorhersagegenauigkeit
(Quelle: analysis_vector.ipynb)

In diesem Bereich haben wir zunächst die Daten in Trainingsdaten und Testdaten aufgeteilt. In nächstem Schritt haben wir jeweils einen Layer mit Vektoren verwendet und diese durch das Modell laufen lassen, um die Vorhersagegenauigkeit zu ermitteln und auszugeben.

Während der Analyse haben wir festgestellt, dass die Verarbeitung sehr lange dauert. Aus diesem Grund haben wir die Vektoren erneut extrahiert und nur die Vektoren aus der letzten Epoche in eine CSV-Datei gespeichert.

Indem wir dem Modell die Anzahl der Epochen und die aktuelle laufende Epoche mitgeben, konnten wir mit einer if-Schleife bestimmen, wann die Vektoren gespeichert werden sollen, wie in Abbildung neun zu sehen ist. Auch musste die Klasse SportsTablesDGLDataset angepasst werden, da die for Schleife nicht mehr benötigt wurde.

```
table_size = len(pooled_output)
table_counter = 0
if (epoch_size-1) == epoch:
    #save vectors in a CSV file
    with open(self.csv_file, 'a', newline='') as csvfile:
        writer = csv.writer(csvfile)

        for element in range(table_size):
            extracted_vectors = []
            extracted_vectors.append(input_layer[table_counter])
            extracted_vectors.append(hidden_layer1[table_counter])
            extracted_vectors.append(hidden_layer2[table_counter])
            extracted_vectors.append(output_layer[table_counter])
            writer.writerow(extracted_vectors)
            table_counter += 1
```

Abbildung 9 Abbild vom Code: Speicherung der letzten Epoche
(Quelle: gcn.py)

Durch die Reduzierung konnten wir effizienter die Analyse bearbeiten. Außerdem hatten wir eine Momentaufnahme des endgültigen Zustands des Modells, wodurch die Leistung des Modells besser zu bewerten ist.

Dieses gesamte Vorgehen sollte auch auf die anderen Sportstables angewendet werden können oder es könnte gleichzeitig auf alle Sportstables angewendet werden. Da wir nur auf die CPU zugreifen konnten, konnten wir dies nicht testen, da der Durchlauf mindestens einen ganzen Tag gedauert hätte.

Während der gesamten Lösungsbearbeitung wurden lediglich die Python-Dateien gcn.py, train_gcn.py und SportsDB_data_loader.py bearbeitet. Die Analyse sowie die CSV-Dateien sind im Ordner extracted_vectors enthalten.

5 Ergebnis der Analyse

5.1 Allgemeine Daten

Im ersten Abschnitt der Analyse wurde der Inhalt der CSV-Datei beschrieben. Wie in Abbildung zehn zu erkennen ist, enthält jede Spalte 2.997 Werte und es gibt keine leeren Zeilen. Außerdem ist hervorzuheben, dass es mindestens 2/3 der Daten einen numerischen Datentyp haben.

	table_name	column_name	columns_data_type	column_label	input_layer	hidden_layer1	hidden_layer2	output_layer
count	2997	2997	2997	2997	2997	2997	2997	2997
unique	139	99	2	102	2961	2962	2962	2962
top	mlb_season_team_pitching_stats_2005	Tm	numerical	baseball.team.name	[-0.9955307841300964, -0.4307440519332886, -0...	[0.7017772197723389, 1.0636335611343384, 1.022...	[0.16311699151992798, 1.1638725996017456, 0.84...	[-0.6628270745277405, -2.145338535308838, -2.9...
freq	36	137	2449	138	6	6	6	6


```
df.shape
✓ 0.0s
(2997, 8)
Python

df.dtypes
✓ 0.0s
Python
table_name      object
column_name     object
columns_data_type object
column_label    object
input_layer     object
hidden_layer1   object
hidden_layer2   object
output_layer    object
dtype: object
```

Abbildung 10 Zusammengefasste Beschreibung CSV-Datei
(Quelle: analysis_vector.ipynb)

Auch sind sehr wenige Vektoren in den einzelnen Layers ähnlich. Weiterhin enthält die CSV-Datei 2.997 Zeilen und acht Spalten. Hierbei sind alle Spalten als Objekte deklariert.

5.2 PCA-Analyse

Unsere PCA-Analyse auf die Vektoren einer Zeile lieferte eine visuelle Darstellung der Verteilung der Vektoren innerhalb jeder Schicht. Diese Analyse ist jedoch nur auf eine einzige Zeile beschränkt und gibt daher keine umfassenden Einblicke in die allgemeine Struktur oder Beziehungen innerhalb unserer Daten. Man kann lediglich in Abbildung elf erkennen, dass die Werte im input_layer stärker verteilt sind, während sie sich in den nachfolgenden Schichten auf einen bestimmten Bereich konzentrieren.

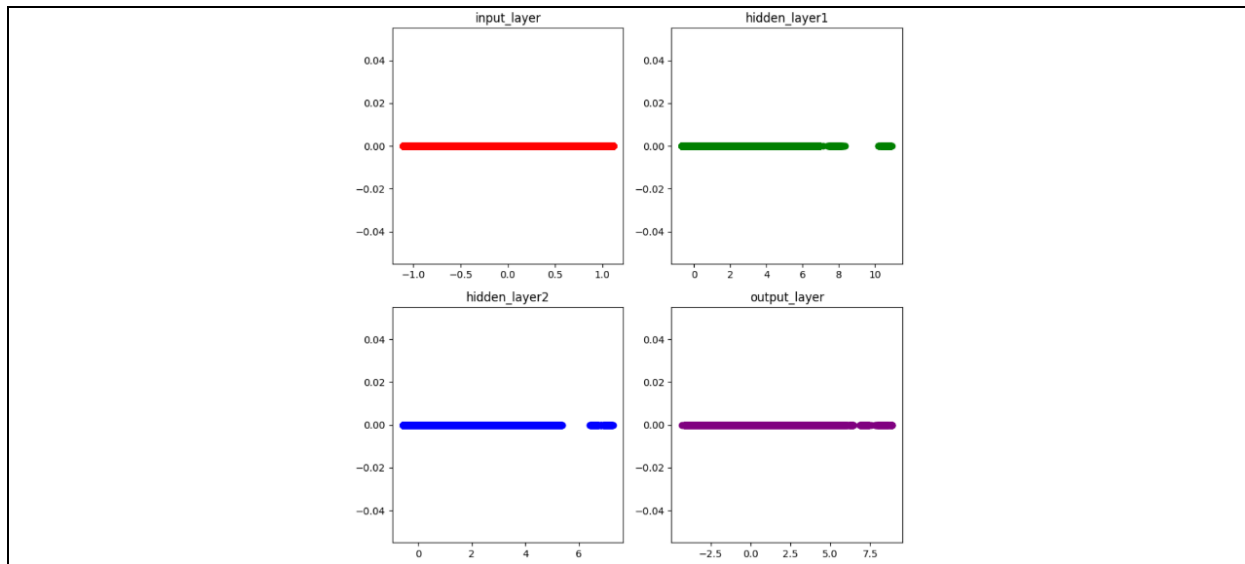


Abbildung 11 PCA-Analyse einer Dimension
(Quelle: analysis_vector.ipynb)

Darüber hinaus haben wir separate Plots für jeden Layer unserer Daten erstellt. Diese Visualisierung hebt die Unterschiede zwischen den einzelnen Layers hervor und ermöglicht es uns, die spezifischen Merkmale jedes Layers besser zu verstehen. Wie in Abbildung zwölf zu sehen ist, sind die Werte im Input Layer viel stärker verteilt als in den anderen Layers.

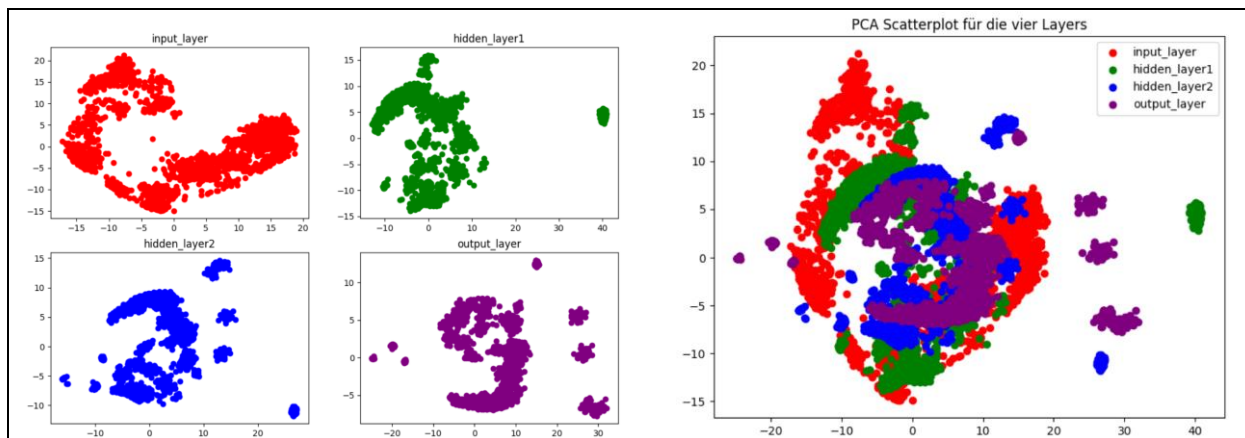


Abbildung 12 PCA-Analyse jeweiligen Layers
(Quelle: analysis_vector.ipynb)

Die Darstellung aller Layer in einem einzigen Plot war besonders aufschlussreich. Hier zeigte sich, dass die Daten im Laufe der Layers beginnen, sich zu clustern. Während die Datenpunkte im Input-Layer mehr verteilt waren, konnten wir im Output-Layer deutlich abgegrenzte Cluster erkennen. Dies lässt auf eine zunehmende Spezialisierung der Daten im Verlauf der Layer schließen.

Schließlich führten wir eine PCA-Analyse durch, die nur auf den numerischen Zeilen unserer Daten basierte. Der Vergleich dieser Analyse mit der vorherigen Analyse, die alle Datenpunkte umfasste, zeigte nur geringfügige Unterschiede auf, was darauf hindeutet, dass unsere Daten hauptsächlich aus numerischen Werten bestehen. Die geringfügigen Unterschiede könnten auf einige wenige nicht-numerische Werte zurückzuführen sein. Trotz dieser Unterschiede blieben die grundlegende Struktur und Verteilung der Datenpunkte in den Plots ähnlich.

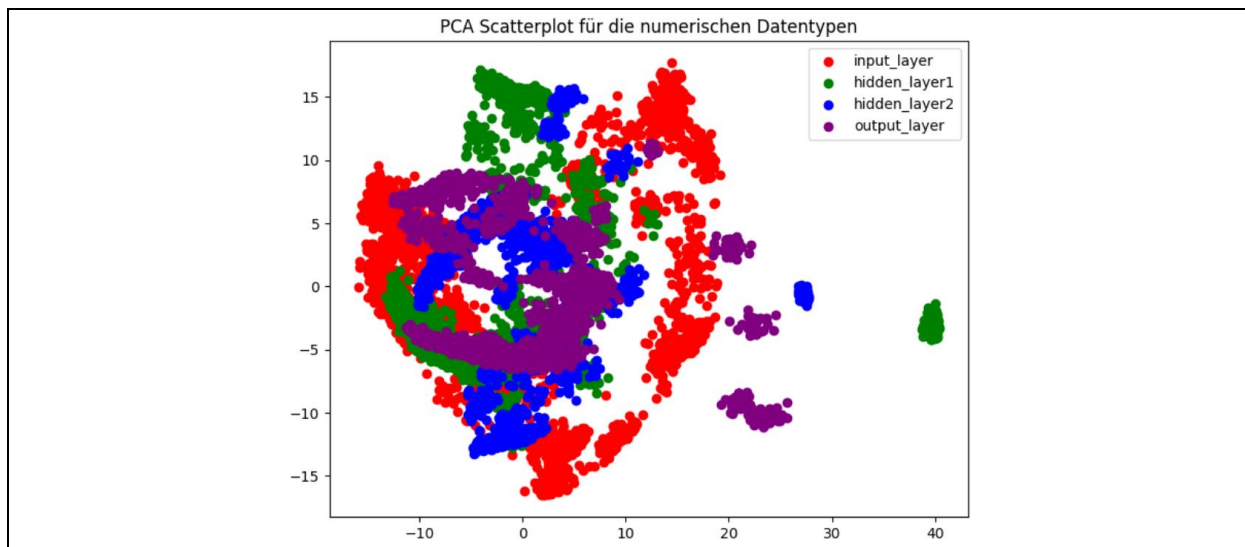


Abbildung 13 PCA-Analyse mit numerischem Datentyp
(Quelle: analysis_vector.ipynb)

Zusammenfassend lässt sich sagen, dass unsere Analyse mithilfe der PCA wesentliche Einblicke in die Struktur unserer Daten lieferte. Durch die Anwendung der PCA auf einen Vektor, separate Layer und alle Layer in einem einzigen Plot konnten wir eine umfassende Visualisierung unserer Daten erreichen. Diese Analysen haben uns wertvolle Einblicke in die Verteilung und Struktur unserer Daten geliefert und dazu beigetragen, die Spezialisierung der Daten über den Layer hinweg zu verstehen.

5.3 t-SNE-Analyse

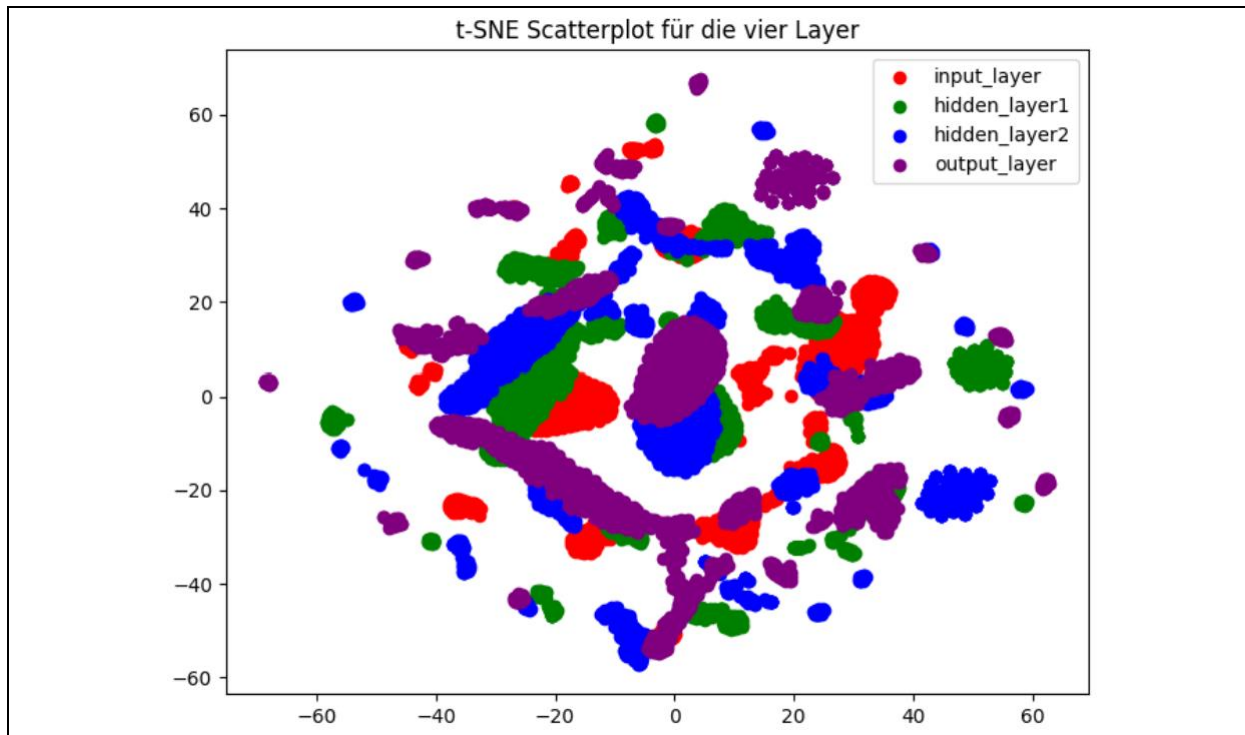


Abbildung 14 t-SNE-Analyse
(Quelle: analysis_vector.ipynb)

In Abbildung 14 ist ersichtlich, dass sich jeder Layer in einzelnen Clustern befindet. Es ist zu erkennen, dass die Cluster des Input-Layers einen größeren Umfang haben als beim Output-Layer. Die Werte des Output-Layers sind stärker zu einem Cluster zusammengefügt. Ähnlich wie bei der PCA-Analyse ist zu erkennen, dass sich die Werte mit jeder weiteren Schicht immer stärker verdichten.

Dadurch dass wir die Vektoren aus der letzten Epoche nutzen, sind bereits die Vektoren aus dem input layer in einer gewissenweise geclustert. Auch bei der PCA-Analyse sieht man bereits eine leichte Clusterung im input layer.

5.4 Cosinus Ähnlichkeit

Mit der Ausgabe der Cosinus-Ähnlichkeit ist erkennbar, wie sich zwei Layer voneinander unterscheiden oder ähneln. Da es sehr viele Vektoren in den jeweiligen Schichten gibt, ist es schwer festzustellen, ob sich die Vektoren innerhalb der Schichten stark ähneln. Wie in Abbildung 15 ersichtlich, gibt es keine Werte nahe eins, die eine genaue Ähnlichkeit anzeigen. Auch bei den anderen Berechnungen zwischen den Schichten hidden-layer1 und output-layer sowie

hidden-layer2 und output-layer konnte keine Ähnlichkeit festgestellt werden. Es könnte auch der Grund sein, dass die Darstellung die hohen Ähnlichkeiten nicht anzeigt.

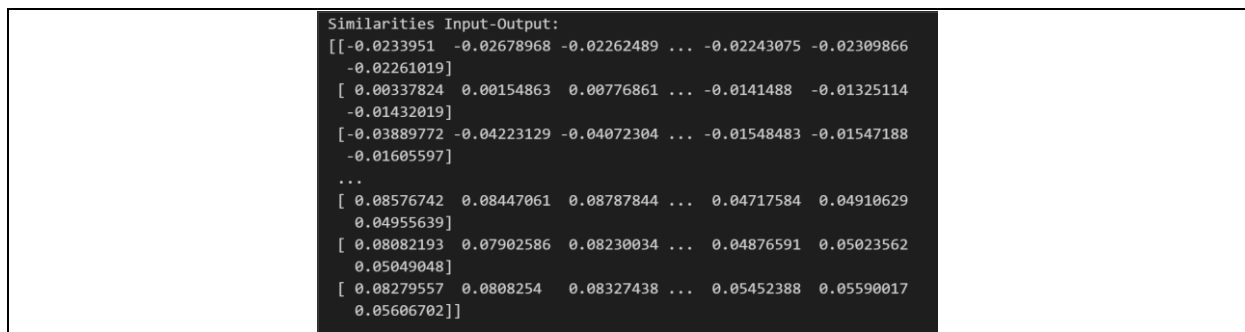


Abbildung 15 Cosinus-Ähnlichkeit Input und Output Layer
(Quelle: analysis_vector.ipynb)

Da es schwierig ist, anhand der reinen Ausgabe der Cosinus-Ähnlichkeit klare Schlussfolgerungen zu ziehen, haben wir die Ähnlichkeiten in einer Heatmap dargestellt. In Abbildung 16 ist zu sehen, dass in Bereichen mit höheren Werten mehr positive Werte enthalten sind. Interessanterweise enthält die Heatmap für HiddenLayer1-Output viele positive Werte, während wir für die Heatmap von Hidden-Layer2-Output eher auf mehr positive Werte gehofft hatten. Es war überraschend, dass in dieser Heatmap viele Werte im negativen Bereich liegen.

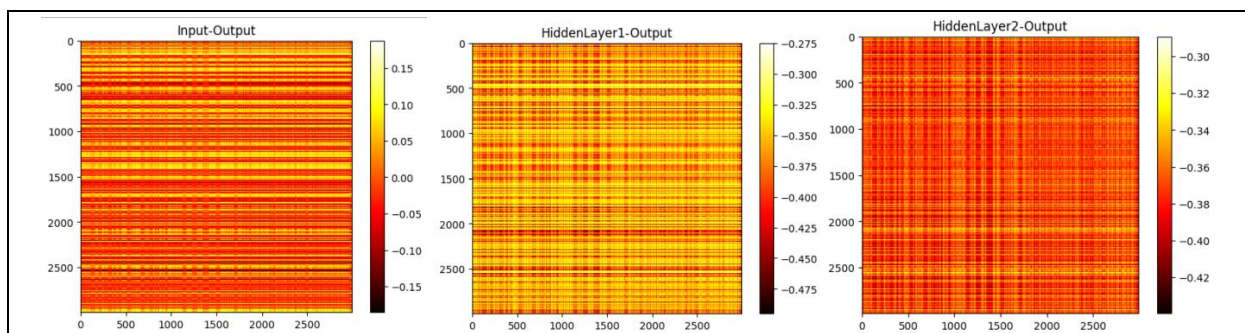


Abbildung 16 Heatmap Cosinus-Ähnlichkeit
(Quelle: analysis_vector.ipynb)

5.5 Vorhersagegenauigkeit semantischen Typen

Eine der größten Herausforderungen, auf die wir in unserer Arbeit gestoßen sind, war die Zuordnung der semantischen Typen unserer Daten. Dies wäre ein wichtiger Schritt gewesen, um einen tieferen Einblick in die Beziehungen und Muster in unseren Daten zu bekommen, da es uns erlaubt hätte, den semantischen Kontext der Daten in die Analyse einzubeziehen.

Theoretisch könnte dies zu weiteren Erkenntnissen führen, insbesondere wenn man versucht, Muster zu verstehen oder Vorhersagen auf der Basis der Daten zu treffen.

Im Output wurden uns die Vorhersagegenauigkeiten der einzelnen Schichten für den semantischen Typ dargestellt. Hier ist ersichtlich, dass die Genauigkeit bei allen Schichten sehr ähnlich ist, wobei sie bei den letzten beiden Schichten marginale Unterschiede aufweist.

	<pre>Metric für input_layer: Accuracy: 0.673469387755102 Metric für hidden_layer1: Accuracy: 0.6775510204081633 Metric für hidden_layer2: Accuracy: 0.689795918367347 Metric für output_layer: Accuracy: 0.6816326530612244</pre>	
--	---	--

Abbildung 17 Vorhersagegenauigkeit semantischen Typen
(Quelle: analysis_vector.ipynb)

Diese Werte erschienen uns jedoch fragwürdig, und wir konnten auch durch andere Methoden kein zufriedenstellendes Ergebnis erzielen. Wir hatten gehofft, dass zwischen dem Hidden Layer 2 und dem Output Layer eine höhere Vorhersagegenauigkeit erzielt werden würde als in den vorherigen Schichten.

In der Praxis waren wir nicht in der Lage, eine wirksame Methode zu entwickeln, um den semantischen Typ der Daten zu bestimmen und den Output entsprechend besser zuzuordnen.

6 Schlussbemerkung

6.1 Fazit

Die Analyse der Hidden Representation von GCN-Layers veranschaulichte ein genaues Verständnis davon, wie sich die Vektorrepräsentationen im Modell verändern. Anhand der Visualisierung der einzelnen Layer konnte beobachtet werden, dass sich im Input Layer die Werte eher verteilt, während sich im Output Layer die Werte stark geclustert haben. Es konnte auch festgestellt werden, dass die Vorhersagegenauigkeit mit den jeweiligen Schichten leicht zunahm. Allerdings sind weitere Methoden notwendig, um diese Ergebnisse zu bestätigen. Diese Analyse ermöglichte es, die Einflüsse von GCN-Layers auf die Hidden Representation darzustellen.

6.2 Ausblick und Weiterentwicklungsmöglichkeiten

Die korrekte Vorhersage des semantischen Typs kann eine Vielzahl von Vorteilen bieten, die das Verständnis der Daten verbessern und die Entscheidungsfindung beeinflussen können.

Durch die genaue Bestimmung des semantischen Typs der Daten mittels GCN-Layers können Beziehungen zwischen den verschiedenen statistischen Informationen hergestellt werden. Dies ermöglicht es, komplexe Zusammenhänge zwischen den Datenpunkten zu erkennen und tiefergehende Analysen durchzuführen. Durch diese Verbindung können beispielsweise unterschiedliche Aussagen über die Leistung der Spieler abgeleitet werden. Indem Daten wie Punkte, Rebounds und Assists miteinander in Beziehung gesetzt werden, kann ein umfassenderes Bild davon erhalten werden, wie ein Spieler zur Mannschaft beiträgt und welche Auswirkungen er auf das Spiel hat. Eine genauere Profilerstellung ist möglich, die nicht nur auf bloßen Zahlen basiert, sondern die Beziehung zwischen den verschiedenen statistischen Kategorien berücksichtigt. Dies ermöglicht eine genauere Bewertung der Spielerleistung und kann dazu beitragen, eine bessere Rangliste oder Bewertung der Spieler zu erstellen. Darüber hinaus können die gewonnenen Erkenntnisse aus der Analyse der Hidden Representations und der Beziehungen zwischen den Datenpunkten die Entscheidungsfindung beeinflussen. Wenn ein tieferes Verständnis

für die Zusammenhänge der Daten gewonnen wird, können möglicherweise bessere Entscheidungen getroffen werden.

Es ist wichtig anzumerken, dass die korrekte Vorhersage des semantischen Typs eine anspruchsvolle Aufgabe ist, die weiterer Forschung bedarf. Es erfordert die Entwicklung und Optimierung von geeigneten Modellen und Algorithmen, um eine hohe Vorhersagegenauigkeit zu erzielen. Zusätzlich zu den numerischen Spalten könnten auch andere Datenarten wie Kategorien oder Texte einbezogen werden, um ein umfassenderes Verständnis der Basketballstatistiken zu ermöglichen. Die Berücksichtigung von Zeitabhängigkeiten könnte ebenfalls ein vielversprechender Schritt sein, um das Verhalten von Spielern und Teams besser zu verstehen und präzisere Vorhersagen zu treffen, da sich die Daten schnell ändern.

Ein weiterer vielversprechender Bereich zukünftiger Forschung ist die Anwendung der gewonnenen Erkenntnisse in Echtzeitsystemen. Dies könnte zu verbesserten Spielstrategien, Leistungsvorhersagen und taktischen Entscheidungen führen. Dadurch können Trainer, Spieler und Teams von den Fortschritten im Bereich der Datenanalyse und des maschinellen Lernens profitieren, um ihre Leistung und ihren Erfolg zu maximieren. Neben der praktischen Anwendung in der Welt des Sports könnte die Forschung zur Analyse von GCN-Layers und Hidden Representations in numerischen Tabellenspalten auch auf andere Bereiche ausgeweitet werden. Zum Beispiel könnten ähnliche Ansätze in anderen Sportarten, aber auch in verschiedenen Bereichen wie Finanzen, Gesundheitswesen oder Verkehr angewendet werden. Dies würde zu einem breiteren Einsatz von GCN-Layers führen und das Potenzial dieser Techniken für die Analyse und Vorhersage komplexer Datenstrukturen aufzeigen.

Insgesamt bietet das Thema "Analyse der Einflüsse von GCN-Layers auf Hidden Representations von numerischen Tabellenspalten" vielversprechende Möglichkeiten für zukünftige Forschung und Anwendungen. Die Kombination von GCN-Layers, semantischer Typvorhersage und Hidden Representations kann zu einem tieferen Verständnis der Basketballstatistiken führen, innovative Systeme für Entscheidungsunterstützung ermöglichen und die Leistung von Spielern und Teams verbessern. Indem diese Forschungsrichtungen weiterverfolgt werden, kann ein bedeutender Beitrag zur Weiterentwicklung des Sports und der Datenanalyse geleistet werden.

Quellenverzeichnis

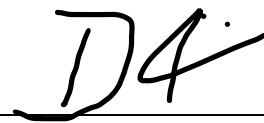
- IONOS SE (2020): Graph Neural Network: Der nächste Schritt für Deep Learning, <https://www.ionos.de/digitalguide/online-marketing/suchmaschinenmarketing/graph-neural-network/>, Abruf am 2023-06-03.
- Karagiannakos, Sergios (2021): Best Graph Neural Network architectures: GCN, GAT, MPNN and more, AI Summer. <https://theaisummer.com/gnn-architectures/>, Abruf am 2023-05-07.
- Pohl, Golo (2023): Graph Neural Networks: Eine Einführung in die Welt der Graphen-basierten KI-Modelle, Lamarr-Institut für Maschinelles Lernen und Künstliche Intelligenz. <https://lamarr-institute.org/de/graphen-basierten-ki-modelle/>, Abruf am 2023-06-15.
- Zhu, Yanqiao; Xu, Yichen; Yu, Feng; Wu, Shu; Wang, Liang (2020): CAGNN: Cluster-Aware Graph Neural Networks for Unsupervised Graph Representation Learning, arXiv. <https://arxiv.org/abs/2009.01674>, Abruf am 2023-06-15.

Ehrenwörtliche Erklärung

Wir versichern hiermit, dass wir die Dokumentation selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mosbach, 09.07.2023

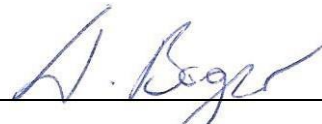
Ort, Datum



Unterschrift Damien Arriens

Brackenheim, 09.07.2023

Ort, Datum



Unterschrift Daniel Boger

Siegelbach, 09.07.2023

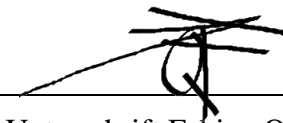
Ort, Datum



Unterschrift Simon Di Latte

Neckarsulm, 09.07.2023

Ort, Datum



Unterschrift Fabian Qarqur

Heilbronn, 09.07.2023

Ort, Datum



Unterschrift Julian Stipovic