How to Observer

Inhaltsverzeichnis

1. Einführung

Der Observer hat die Aufgabe, Low-Level-Informationen in High-Level-Informationen umzuwandeln, auf denen basierend später Spielentscheidungen getroffen werden können. Bei den Low-Level-Informationen handelt es sich um Daten, die direkt aus dem Kamerabildern entnommen werden können, dazu gehören beispielsweise die Positionen der Roboter oder die Position des Balls. Daraus lassen sich komplexere Informationen ableiten, wie beispielsweise das Gefahrenlevel der Gegner oder die aktuelle Torwahrscheinlichkeit.

Diese Informationen sind für die Robotersteuerung essenziell. So sollen Roboter etwa nur dann auf das Tor schießen, wenn eine realistische Chance besteht, mit dem Schuss auch tatsächlich ein Tor zu erzielen. Gleichermaßen sollten gefährliche Gegner auch besser verteidigt werden, um so Gegentore zu vermeiden. Daher müssen die Informationen des Observers akkurat sein und das tatsächliche Spielgeschehen gut beschreiben. Dabei sind viele verschiedene Faktoren entscheidend, die bei der Entwicklung ermittelt und miteinander gewichtet werden müssen. Dieser Schritt wird auch als **Feature Engineering** bezeichnet. Aus den gefundenen Faktoren (= *Features*) lassen sich dann Formeln und Modelle ableiten, die einen Input (z.B. Roboterpositionen) in einen Output (z.B. Torwahrscheinlichkeit) umwandeln. Die Formeln lassen sich in Form von **Heuristiken** selbst designen, diese unterliegen oftmals jedoch einem menschlichen Bias. Stattdessen lassen sich auch Methoden des **maschinellen Lernens** verwenden, um einerseits das Erstellen solcher Modelle zu automatisieren und andererseits den menschlichen Bias zu reduzieren.

Um maschinelle Lernmethoden sinnvoll einsetzen zu können, müssen entsprechende Trainingsdaten erzeugt werden. Hierzu können bestimmte Szenarien generiert werden, in denen eine bestimmte Aktion ausgeführt und vorab definierte Features aufgenommen werden. Dies wird später noch anhand eines Beispiels ausführlicher betrachtet. Da das Designen und Durchführen eines solchen Trainings jedoch recht aufwendig sind, sollen zunächst weiter Heuristiken verwendet werden. Diese sollen dann nach und nach durch maschinelle Lernverfahren ersetzt werden. Die Güte der erstellten Modelle ist anhand von Simulationen und idealerweise auch anhand des realen Systems zu validieren.

Einschub: Erklärung der Grundbegriffe und weiterführende Informationen

Feature Engineering (FE):
Bezeichnet den Prozess, durch den Features aus Rohdaten und Domänenwissen extrahiert und aufbereitet werden. Features sind dabei die relevanten Faktoren, die eine aussagekräftige und korrekte Beschreibung des aktuellen Zustandes liefern.

- Kurzübersicht über FE:
  https://www.researchgate.net/publication/287743399_A_survey_of_feature_selection_and_feature_extraction_techniques_in_machine_learning

Heuristik:
Bezeichnet ein analytisches Vorgehen, bei dem mit begrenztem Wissen Schätzungen und Mutmaßungen über ein System getroffen werden können.

- Mehr zu Heuristiken und Feature Engineering:
  https://ayc-data.com/data_science/2020/10/18/machine-learning-feature-engineering-and-heuristics.html
- Mehr zur Bias Problematik (aus soziologischer Sicht):
  https://strukturierte-analyse.de/mentale-abkuerzungen-biases-und-heuristiken/

Maschinelles Lernen (ML):
Oberbegriff für die künstliche Generierung von Wissen aus Erfahrung. In einem Trainingsprozess wird aus Beispielen gelernt und das Wissen später soweit verallgemeinert, dass es auch auf unbekannte Daten anwendbar ist. Ziel ist meist eine *Klassifikation* (Zuordnung von Daten zu Gruppen) oder eine *Regression* („Interpolation" zwischen Werten). Liegt in den Trainingsdaten das Ergebnis bereits vor, handelt es sich um *supervised Learning* (= *überwachter Lernvorgang*). Das Modell soll den Zusammenhang von Input und Output lernen. Ist der Zusammenhang hingegen noch unbekannt, liegt *unsupervised Learning* (= *unüberwachtes Lernen*) vor, wobei das Modell selbstständig Strukturen in den Daten finden soll. Beim *Reinforcement Learning* erhält ein Agent je nach Zustand und Aktion eine Belohnung bzw. Bestrafung und versucht die Aktionen so zu wählen, dass die Belohnung maximiert wird (Ist für den Observer jedoch weniger relevant).

- Allgemeine Kurzübersicht ML:
  https://arxiv.org/pdf/1808.02342.pdf
- Vollständiger Online-Kurs zu Grundlagen ML der Stanford University:
  https://www.coursera.org/specializations/machine-learning-introduction

2. Übersicht über die vorhandenen Low-Level-Informationen

Im Folgenden sind Daten aufgelistet, die als gegeben vorausgesetzt werden können. Diese Daten können als Input für die Observer-Funktionen genutzt werden. Der Großteil der Informationen findet sich auch in der Dokumentation des *Game Data Providers*. Das ist das Softwaremodul, welches sich um die Vorverarbeitung und die Sammlung der meisten der hier aufgeführten Informationen kümmert. Dabei werden die Daten auch schon aufbereitet, um z.B. das Kamerarauschen und das Ausschließen von Fehlerhaften und unvollständigen Daten zu gewährleisten.

Roboterposition	Besteht aus den x- und y-Koordinatenwerten sowie der Drehung um die z-Achse. Ist für alle eigenen sowie gegnerischen Roboter bekannt.
Ballposition	Besteht aus den x- und y-Koordinatenwerten des Balls.
Robotergeschwindigkeit	Beinhaltet die translatorischen Geschwindigkeiten in x- und y-Richtung sowie die Rotationsgeschwindigkeit um die z-Richtung aller Roboter.
Ballgeschwindigkeit	Beinhaltet die translatorische Geschwindigkeit des Balls in x- und y-Richtung.
Lichtschrankenzustand	Ist nur für unsere eigenen Roboter verfügbar; Gibt an, ob der Roboter grade den Ball hat.
Spielzustand	Information vom Game-Controller, ob das Spiel grade normal läuft oder z.B. aufgrund eines Freistoßes etc. pausiert ist.
Weitere Auto-Ref.-/ Gamecontroller-Infos	Name der Teams, abgelaufene Zeit, Anzahl der Timeouts, Spielstand, gelbe Karten… o Mehr dazu in der Doku zur Software des offiziellen Game-Controllers: https://github.com/RoboCup-SSL/ssl-game-controller

3. Übersicht über benötigte High-Level Informationen für Bod-23

Im Folgenden werden die wichtigsten High-Level-Informationen aufgelistet, die der Observer zwingend bereitstellen muss. Im Laufe der Entwicklung können jedoch noch weitere Faktoren hinzukommen, daher ist eine enge Zusammenarbeit mit den Entwicklern des Taskmanagers erforderlich.

Torwahrscheinlichkeit	Gibt die Chance an, bei einem direkten Schuss auf das Tor auch tatsächlich ein Tor zu erzielen. Es ist dabei zu berücksichtigen, dass Gegner versuchen werden den Ball abzufangen. Ebenso ist die Schussposition (Abstand und Winkel zum Tor) entscheidend.
Bestes Torschussziel	Gibt den Punkt an, auf den der Schütze schießen sollte, um die Torwahrscheinlichkeit zu maximieren. Hierbei sind Faktoren wie die Ungenauigkeit des Kickers, die Position des Schützen, die Zeit zum Ausrichten und etwaige Verteidiger zu berücksichtigen.
Passwahrscheinlichkeit	Gibt die Chance an, mit der ein zum Mitspieler gepasster Ball auch tatsächlich bei diesem ankommt. Ähnlich zur Torwahrscheinlichkeit.
Bester Passempfänger	Gibt an, welcher Mitspieler im Moment die beste Anspielstation darstellt. Dieser Mitspieler sollte nicht gedeckt sein (=> hohe Passwahrscheinlichkeit) und sich in einer Position befinden, von der aus das Spiel einen guten Fortschritt Richtung Gegnertor machen kann.
Bester Passpunkt	Ähnlich zum besten Passempfänger, allerdings soll dem Empfänger in den Lauf gepasst werden. Der Punkt sollte Richtung Gegnertor liegen, damit der Ball schon während des Passens einen weiteren Fortschritt Richtung Tor macht, jedoch muss der Empfänger diesen noch sicher erhalten können. Daher ist auf Gegner etc. zu achten.
Gefahrenlevel	Gibt an, wie gefährlich ein Gegner/Mitspieler ist. Das Gefahrenlevel ergibt sich aus der Wahrscheinlichkeit an den Ball zu gelangen (z.B. durch einen Pass) und mit dem Ball einen Fortschritt zu machen bzw. ein Tor zu erzielen (siehe Torwahrscheinlichkeit).
Dribbeldistanz	Gibt an, wie weit ein Roboter sich noch mit dem Ball bewegen darf, bevor er abgespielt werden muss.
Ballprädiktion	Gibt an, wo ein sich bewegender Ball ankommen wird. Kann genutzt werden, um Bälle abzufangen, Torwahrscheinlichkeiten zu ermitteln etc.
Schussprädiktion	Ermittelt, ob und wohin ein Roboter schießen/passen könnte. Relevant sind hierfür Position des Balls und die Drehung des Schützen, ggf. können seine Absichten geschätzt werden. Wichtig für das Abfangen von Schüssen und Verteidigen von Sichtlinien etc.
Spielmodus	Gibt an, ob sich unser Team grade im Angriff oder der Verteidigung befindet. Hierfür können Faktoren wie die Richtung der Ballbewegung oder das Team mit dem letzten Ballbesitz verwendet werden.

4. Konkretes Beispiel: Torwahrscheinlichkeit

Im Folgenden soll beispielhaft gezeigt werden, wie eine Torwahrscheinlichkeit berechnet werden kann. Dabei wird zunächst ein heuristischer Ansatz gezeigt und danach ein Ansatz mittels maschineller Lernmethoden erläutert.

4.1. Lösung durch Heuristik

Eine Lösung über eine Heuristik ist meist einfacher und schneller umzusetzen als eine mittels maschineller Lernmethoden. Hierbei müssen während der Entwicklung zunächst die maßgebenden Faktoren (*Features*) ermittelt werden. Dafür werden Domänenwissen und Intuition verwendet. Dadurch ist dieses Vorgehen aber auch anfällig für sogenannte *Bias*, also einer Verzerrung z.B. durch fehlerhafte menschliche Eingebung. Es empfiehlt sich daher, das Vorgehen mit anderen Leuten zu diskutieren und die Ergebnisse anhand von Simulationen (und idealerweise auch anhand des echten Systems) zu validieren.

Das Vorgehen ist dabei wie folgt:

1. Ermittlung geeigneter Features
2. Gewichtung der gefundenen Features
3. Zusammenführung der Features zu einem Modell
4. Validierung und ggf. Überarbeitung

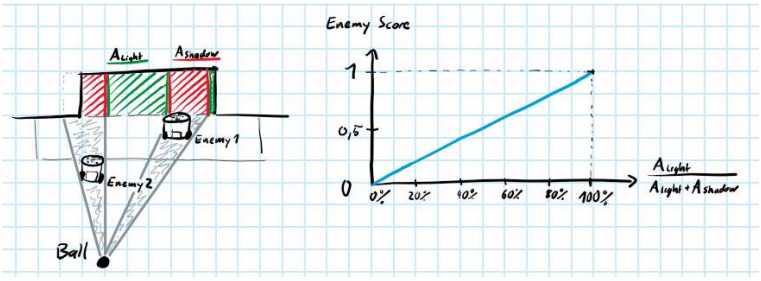
Aus der Beobachtung diverser Spiele ist klar, dass die Torwahrscheinlichkeit im Wesentlichen von zwei Faktoren abhängt: Zum einen von der Position des Schützen und zum anderen von der Position der Verteidiger. Die Position sollte dabei möglichst nahe am gegnerischen Tor sein und die Schusslinie möglichst im Lot zur Torlinie stehen. Dadurch wird die Wahrscheinlichkeit das Tor überhaupt zu treffen maximiert und die Zeit, die der Torwart zum Abfangen hat, minimiert. Die verteidigenden Gegner sollten möglichst wenig von der Torfläche verdecken.

Somit ergeben sich die drei relevanten Features, ausgedrückt als Scores:

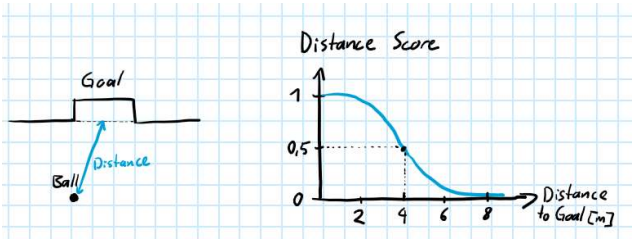
- *Enemy Score:* „Wie viel Torfläche wird aus Sicht des Balls von Gegnern verdeckt?“
- *Distance Score:* „Wie weit ist der Schütze vom Tor entfernt?“
- *Angle Score:* „In welchem Winkel steht die Schusslinie zum Lot auf die Torlinie?“

Da diese Faktoren unterschiedliche Einheiten besitzen und später miteinander verrechnet werden müssen, empfiehlt sich eine Normierung der Daten. Somit wird die z.B. die Entfernung nicht mehr in Metern angegeben, stattdessen wird ein Score zwischen 0 und 1 ausgegeben, wobei 0 der maximalen Entfernung (damit schlechtem Score) und 1 der minimalen Entfernung (somit gutem Score) entspricht. Es können auch nicht-linearitäten eingebaut werden, indem beispielsweise *atan*-, *sigmoid* oder *parabel*-Funktionen verwendet werden.

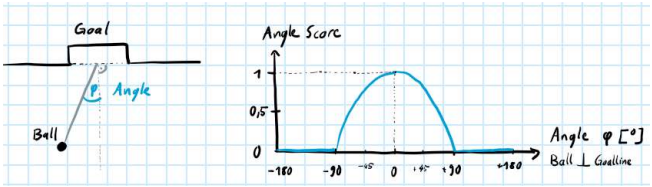
Für den Enemy Score wird der Ball als Lichtquelle angenommen und ausgerechnet, wie viel Prozent der Torfläche vom Ball „beleuchtet“ wird, und wie viel durch Gegner „beschattet“ werden. Der Score lässt sich dann durch $A_{beleuchte} \ / A_{gesamt}$ berechnen.



Aus Beobachtungen wurde festgestellt, dass es keinen großen Unterschied macht, ob der Ball aus einer Entfernung von 0.5 m, 1 m oder 2 m geschossen wird, aber ab etwa 4 m die Trefferwahrscheinlichkeit schon deutlich gesunken ist. Diese Beobachtungen lassen sich z.B. durch eine *atan*-Funktion darstellen:



Der Angle Score soll maximal sein, wenn sich der Ball frontal vor dem Gegnertor befindet, und kleiner werden, wenn z.B. aus einer der Ecken aufs Tor geschossen wird. Wie auch bei der Distanz lassen Beobachtungen vermuten, dass z.B. zwischen 0° und 5° kein signifikanter Unterschied hinsichtlich der Torwahrscheinlichkeit besteht. Ab einem Winkel von unter -90° bzw. über 90° ist ein Treffen des Tors nicht mehr möglich. Aus Domänenwissen ist bekannt, dass die treffbare Fläche mit dem Cosinus des Winkels abnimmt, das lässt sich alternativ aber auch als Parabel beschreiben.



Jetzt, da die Features und deren Berechnung bekannt sind, müssen diese gewichtet und zu einer Formel zusammengefasst werden. Da alle Features normiert wurden, lassen sie sich direkt miteinander addieren oder multiplizieren. Ggf. werden noch Gewichtungen benötigt, die angeben, ob ein Feature bedeutender als ein anderes ist. Die Ermittlung und Optimierung dieser Gewichte kann viel Zeit in Anspruch nehmen und da es sich um einen menschlichen Prozess handelt ist auch hier ist wieder auf Bias zu achten.

Darstellung als gewichtete Linearkombination mit den Scores S_i und Gewichten a, b, c:

$$P_{Total} = \frac{1}{a+b+c} \cdot (a \cdot S_1 + b \cdot S_2 + c \cdot S_3)$$

Darstellung als ungewichtete Multiplikation mit den Scores S_i:

$$P_{Total} = S_1 \cdot S_2 \cdot S_3$$

Natürlich gibt es auch weitere Methoden, um die Scores zu einer Gesamtwahrscheinlichkeit zusammenzufassen. Das beste Vorgehen ist dabei immer vom konkreten Problem abhängig. Jedoch haben alle Vorgehen miteinander gemeinsam, dass es sich bei dem Ergebnis um keine objektiv korrekte Wahrscheinlichkeit handelt. Viel mehr gibt der Wert ein Indiz dafür, ob z.B. grade ein Torschuss gewagt werden sollte oder nicht, bei einem Score von 0,8 könnten aber auch in weitaus weniger als 80% der Fälle tatsächlich ein Tor geschossen werden. Damit der Score einer Wahrscheinlichkeit entspricht müssen alle Faktoren sinnvoll gewichtet und alle relevanten Features gefunden und korrekt beschrieben werden. Das ist aufgrund des menschlichen Bias, des begrenzten Wissens über das System sowie dem notwendigen Optimierungsaufwand jedoch weder praktikabel noch möglich, grade wenn die Anzahl an Features und Gewichten groß ist. Trotzdem kann eine Heuristik bereits ausreichend gute Ergebnisse liefern. Allerdings hängt die Qualität auch stets stark von der Optimierung sowie der Eignung der verwendeten Features ab. Die Optimierung lässt sich durch maschinelles Lernen vereinfachen, darauf geht der nächste Abschnitt ein. Für eine manuelle Optimierung sollte stets eine gute Visualisierung (z.B. mit *Heatmaps*) verwendet werden (Dafür sollte *luhviz* entsprechende Funktionen zur Verfügung stellen).

4.2. Lösung durch maschinelles Lernen

Die Aufgaben des Observers fallen in eine der beiden Kategorien: **Klassifikation** oder **Regression**. Beides sind klassische Anwendungsfälle für maschinelle Lernverfahren. Diese Begriffe sowie Links zu weiterführenden Informationen sowie eine Auswahl geeigneter Algorithmen sind in den nachfolgenden Einschüben zu finden. Es ist zu beachten, dass es beim maschniellen Lernen keine Universallösungen gibt und der beste Algorithmus stets von der vorliegenden Problemstellung abhängt. Eine Hilfe kann das automatisierte Maschine Learning sein, mehr dazu im nachfolgenden Kapitel.

Wie zuvor bei der Lösung mithilfe einer Heuristik muss auch für die Lösung mittels maschineller Lernverfahren zunächst ein Feature Engineering durchgeführt werden. Ein großer Vorteil ist jedoch, dass die Gewichtung der Features automatisiert wird, dadurch können auch deutlich mehr Features verwendet werden. Features, die nicht aussagekräftig sind, werden automatisch weniger berücksichtigt, als Features, die essenziell für die Klassifikation oder Regression sind. Dafür werden allerdings Trainingsdaten benötigt, mit denen das Modell dann die ideale Gewichtung anlernen kann. Die Qualität des Modells hängt maßgeblich von der Qualität der Trainingszenarien und der Menge an Trainingsdaten ab. Je nach Komplexität des Problems werden einige tausend bis millionen Trainingsdaten benötigt. Das Trainingszenario sollte realistisch sein, also Szenarien entsprechen, die auch im echten Spiel erwartet werden können.

Zur Entscheidung, ob ein Schuss auf das Tor gewagt werden soll, kann ein Klassifikationsmodell verwendet werden. Hierbei gibt es die Klassen „schießen“ und „nicht schießen“. Es sind weiterhin die Position des Balls sowie die Positionen der verteidigenden Gegner relevant, um eine sinnvolle Entscheidung treffen zu können. In der Heuristik wurde die von Gegnern „beschattete“ Fläche berücksichtigt. Das ist jedoch problematisch, da die Gegner keinesfalls statisch sind, sondern aktiv versuchen werden, den Ball zu blocken. Die Heuristik berücksichtigt das nicht und kann daher zu subidealen Entscheidungen führen. Im folgenden wird ein Verfahren vorgestellt, bei dem auch das Verhalten der Gegner berücksichtigt wird.

Einschub: Regressionsanalyse

Bei Regressionsaufgaben wird der Zusammenhang von einer abhängigen Output-Variable und einer oder mehreren unabhängigen Input-Variablen modelliert. Somit lassen sich mithilfe des antrainierten Modells Output-Daten vorhersagen.

Beispiele Regression:

- Schätzung des Alters in Abhängigkeit zur Körpergröße
- Vorhersage der Mietkosten in Abhängigkeit von Wohnungsgröße, Baujahr und Stadtnähe
- Berechnung der Wahrscheinlichkeit eines Torschusserfolgs

Geeignete Algorithmen:

- **Lineare Regression**: Punktewolke wird durch Gerade angenähert
  - Wikipedia
    https://en.wikipedia.org/wiki/Linear_regression
- **Support Vector Regression**
  - Blogpost
    https://towardsdatascience.com/an-introduction-to-support-vector-regression-svr-a3ebc1672c2
- **Lasso Regression**
  - Blogpost
    https://datascience.eu/de/maschinelles-lernen/lasso-regression-einfache-definition/
- **Random Regression Forests**
  - Blogpost mit Beispiel
    https://towardsdatascience.com/random-forest-regression-5f605132d19d
- **Neuronale Netze** siehe Einschub: neuronale Netze

Einschub: Klassifikation

Bei der Klassifikation werden zunächst Gruppen definiert, denen später dann Datenpunkte zugeordnet werden können. Die Gruppen können manuell erstellt werden, oder durch Cluster-Verfahren (siehe z.B. *k-means*) automatisch anhand unbekannter Daten ermittelt werden.

Beispiele Klassifikation:

- Spamfilter, der E-Mails den Gruppen „Spam“ oder als „kein Spam“ zuordnet.
- Bildklassifikator, der erkennt, welche Person auf einem Bild zu sehen ist.
- Prädiktion, ob ein Torschuss erfolgreich sein wird („Tor“ / „kein Tor“)

Geeignete Algorithmen:

- **Clusteralgorithmen** wie *k-means* oder *mean-shift*; Finden Strukturen (*Cluster*) in Punktwolken, die sich zu einer Gruppe zusammenfassen lassen. Die Strukturen ergeben sich i.d.R. durch die Nähe der Punkte zueinander. Klassifikation erfolgt über die Lage eines unbekannt Punktes relativ zu den Cluster(-zentren).
  - Übersicht über Algorithmen: https://link.springer.com/article/10.1007/s40745-015-0040-1
  - Beispiel für k-means https://neptune.ai/blog/k-means-clustering
- **Support-Vector-Machines** (SVM); Finden Trennlinien, um Punktewolken voneinander zu separieren. Die Trennlinie wird später verwendet, um Punkte zu klassifizieren („liegt Punkt ober- oder unterhalb der Trennlinie?“)
  - Erklärung mit Beispiel: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47
- **Decision Trees** und **Random Forests**; Bestehen aus einem oder mehreren Bäumen, bei dem ausgehend von der Wurzel nacheinander Entscheidungskriterien geprüft werden, die bestimmen, mit welchem Zweig fortgefahren wird. Am Ende des letzten Zweiges steht die Klasse, der der Datenpunkt zugeordnet werden soll. Die Bäume und Wälder lassen sich auch durch Auto-ML automatisch aus Datenpunkten erstellen und optimieren. Ein *Random Forest* setzt sich aus mehreren Bäumen zusammen, die Gesamtentscheidung ergibt sich aus einem Voting der einzelnen Bäume (Mehr dazu in einem späteren Einschub).
  - Wikipedia Decision Tree: https://de.wikipedia.org/wiki/Entscheidungsbaum
  - Blogpost zu Random Forests: https://towardsdatascience.com/understanding-random-forest-58381e0602d2
  - Einführung Random Forests: https://www.tandfonline.com/doi/full/10.1080/21642583.2014.956265
  - Paper zu Decision Trees und Random Forests https://www.researchgate.net/publication/259235118_Random_Forests_and_Decision_Trees
  - Hyperparameter Tuning (eher speziell, nicht so relevant): https://www.researchgate.net/publication/324438530_Hyperparameters_and_Tuning_Strategies_for_Random_Forest
- **Boosting** Verfahren, wie z.B. *AdaBoost*. Es werden Entscheidungen anhand einzelner Features getroffen. Im Trainingsprozess werden die Gewichte so optimiert, dass die Anzahl an Fehlentscheidungen minimiert werden. Aus den Einzelentscheidungen ergibt sich dann eine Gesamtentscheidung.
  - Blogpost zum Boosting: https://blog.paperspace.com/adaboost-optimizer/
- **Neuronale Netze** siehe Einschub: neuronale Netze

Einschub: Neuronale Netze

Neuronale Netze sind ein wichtiger Bestandteil des maschinellen Lernens, mit denen beliebige Funktionen angenähert werden können. Es lassen sich damit sowohl Aufgaben der **Regression** als auch **Klassifikation** für beliebig komplexe Probleme lösen, allerdings sollte nach Möglichkeit eher auf eines der anderen hier beschriebenen Verfahren zurückgegriffen werden.

Ein neuronales Netz besteht aus einer Input-Schicht, ein oder mehreren versteckten Schichten und einer Output-Schicht. Jede Schicht besteht wiederum aus einer definierten Anzahl an Neuronen. Ist das Netz *fully connected*, dann sind alle Neuronen mit allen Neuronen der vorherigen Schicht über Kanten verbunden (Eine solche Schicht wird auch als *dense Layer* bezeichnet). Zu jeder Kante gehört ein Gewicht, das im Trainingsprozess festgelegt wird. In einem *feedforward* Prozess werden die Input-Werte dann jeweils mit den Gewichten der Kanten multipliziert, die Ergebnisse für jedes Neuron aufsummiert und die Ergebnisse anschließend in eine Aktivierungsfunktion eingesetzt. Hier werden standardmäßig *ReLu*-Funktionen verwendet, Alternativen sind etwa *atan*- oder *sigmoid*-Funktionen. Durch diese Aktivierungsfunktionen lassen sich auch nicht-lineare Zusammenhänge zwischen Input und Output darstellen. Die Ergebnisse werden dann wiederum an die nächste Schicht weitergereicht und dabei entsprechend wieder mit den Kantengewichten der nächsten Schicht multipliziert. Generell kann so bei einer unendlichen Anzahl an Neuronen jede beliebige Funktion approximiert werden.

Das Training von Neuronalen Netzen geschieht zumeist durch *Backpropagation*, wobei es sich um ein *überwachtes Lernverfahren* handelt. Der erwartete Output des Netzes wird dabei mit dem tatsächlichen Output verglichen und die Abweichung auf die einzelnen Gewichte des Netzes rückprojeziert, die dann entsprechend angepasst werden.

- Der Backpropagation Algorithmus Schritt für Schritt erklärt: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Neben den oben beschriebenen *dense Layers* gibt es weitere Schichten. Die wichtigsten sind:

- *Convolutional Layers:* Faltungsschichten, mit denen räumliche Zusammenhänge stärker berücksichtigt werden können, gut für gerasterte Daten wie Bilder
- *Pooling Layers:* Reduzieren die Daten und helfen somit auch gegen **Overfitting**.
- *Recurrent Layers:* Können Daten aus vorherigen Berechnungen „speichern“
- *Dropout:* Wie Dense, aber deaktiviert beim Training Neuronen => gegen **Overfitting**

- Blogpost mit Übersicht über Layer-Typen: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

Wann welche Schicht und mit welchen Hyperparametern verwendet werden soll, ist stets vom zu lösenden Problem abhängig. Es empfiehlt sich, zunächst online nach ähnlichen Problemen zu suchen und von den Lösungen inspirieren zu lassen. Ansonsten gilt: So wenig Neuronen und Schichten wie möglich, so viele wie nötig.

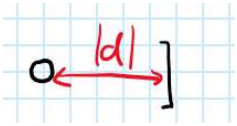
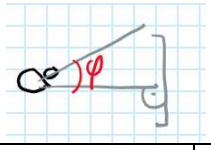
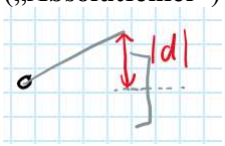
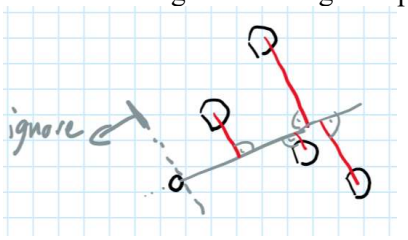
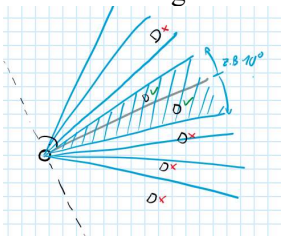
Generell ist es eine gute Praxis die verwendeten Daten zu normalisieren. Grade beim Input ist das wichtig, da alle Daten etwa in derselben Größenordnung liegen müssen, damit sie später sinnvoll miteinander verrechnet werden können. Hierbei wird i.d.R der Wertebereich [0…1] verwendet. Für die Normierung des Outputs empfiehlt sich die Verwendung einer *Softmax*-Funktion in der letzten Schicht.

Weitere Ressourcen:

- Kleinschrittige Erklärung zu NN mit Code-Beispiel für ein *Perceptron* (= primitives Neuron): https://towardsdatascience.com/first-neural-network-for-beginners-explained-with-code-4cfd37e06eaf
- Tutorial zu Neuronalen Netzen in Keras mit Google Colab: https://neptune.ai/blog/how-to-use-google-colab-for-deep-learning-complete-tutorial
- Survey Paper zu (Faltungs-)Netzen (*Convolutional Neural Networks*): https://arxiv.org/ftp/arxiv/papers/2004/2004.02806.pdf

4.2.1 Feature Engineering

Zunächst werden wieder Features definiert. Features wie der Abstand zum Tor oder der Winkel zwischen Schussrichtung und Lot der Torlinie werden beibehalten, der Enemy Score wird jedoch durch die relative Lage der Gegner zur Schusslinie ersetzt. Dabei werden zum einen die Winkel zwischen Gegner-Ball und der Schusslinie betrachtet, zum anderen die Abstände der Gegner zur Schusslinie. Neben den kleinsten Abständen / Winkeln werden auch die Anzahl der Gegner berücksichtigt, die näher als 10 cm sind, näher als 20 cm, etc. Gegner, die sich nicht zwischen Ball und Tor befinden, können ignoriert werden, da diese den Schuss nicht beeinflussen werden. Eine Auflistung und Skizzierung möglicher Features ist in der nachfolgenden Tabelle zu sehen.

<table>
<tr><th colspan="3">Mögliche Features zur Bestimmung, ob ein Tor fallen wird</th></tr>
<tr>
<td>Abstand von Ball zu Tor
  </td>
<td>Winkel zwischen Schussrichtung und Lot durch das Tor
  </td>
<td>Distanz von Tormitte zum geschätzten Auftreffpunkt („Absolutfehler“)
  </td>
</tr>
<tr>
<td colspan="2">Nähe der Gegner zur Schusslinie
➔ Ignoriere Gegner hinter dem Ball
➔ Verschiedene Größen:
• Mittlerer Abstand
• Abstand des nächsten Gegners
• Anzahl Gegner näher 10cm/20cm/30cm…
➔ Analog auch für eigene Spieler
  </td>
<td>Winkel zwischen Ball-Gegner und Schusslinie
➔ Ignoriere Gegner hinter dem Ball
➔ Verschiedene Größen:
• Mittlerer Winkel
• Kleinster Winkel
• Anzahl Gegner mit Winkel kleiner 5°/10°/20°…
➔ Analog auch für eigene Spieler
  </td>
</tr>
</table>

4.2.2 Design eines Trainingsszenarios und Datenaufnahme

Nachdem die Features definiert wurden, können Trainingsszenarien generiert werden. Hierbei ist eine zufällige Generierung wünschenswert, um möglichst viele verschiedene Situationen beim Training zu berücksichtigen. Dabei ist jedoch auch darauf zu achten, dass die Szenarien valide sind und sich der Schütze z.B. nicht außerhalb des Spielfeldes oder innerhalb des gegnerischen Strafraums befindet. Durch die zufällige Generierung und die Abstraktion des Spielgeschehens auf die elementaren Features wird auch das sogenannte **Overfitting** reduziert.

Einschub: Overfitting

Beim Overfitting handelt es sich um eine Überanpassung des Modells auf die Trainingsdaten. Bildlich gesprochen kann man sich Overfitting als einen Studenten vorstellen, der Fragen einer Altklausur auswendig lernt, aber das Gelernte nicht versteht. Das führt dazu, dass die Altklausur zwar zu 100% richtig beantwortet werden kann, der Student aber versagt, wenn er in der tatsächlichen Prüfung mit unbekannten Fragen konfrontiert wird. Dies ist unter allen Umständen zu vermeiden. Dazu kann eine bessere Abstraktion durch eine Dimensionsreduktion erzwungen werden, außerdem können Inputdaten abstrahiert werden oder die Anzahl und Variation der Trainingsdaten erhöht werden.

- Visualisierung von Overfitting anhand einer Regression: https://www.statsoft.de/glossary/O/Overfitting.htm
- Blogpost zu Overfitting und Tipps wie man Overfitting vermeidet: https://www.v7labs.com/blog/overfitting

Für das Beispiel der Torwahrscheinlichkeit bietet es sich an, einen Schützen mit Ball zufällig auf dem Spielfeld und außerhalb der Strafräume zu platzieren. Zudem sollte ein Torwart in Tornähe gespawnt werden, sowie mehrere weitere Gegner an zufälligen Stellen des Spielfeldes. Um das Szenario realisitischer zu gestalten, kann der Torwart auch direkt auf der Torlinie und mit Ausrichtung zum Ball platziert werden. Es bietet sich auch an, 1-2 Gegner in Ballnähe zu platzieren, da diese im echten Spiel auch versuchen werden, an den Ballbesitz zu gelangen. Das sollte nicht nur durch die Position der Gegner, sondern auch durch ihr Verhalten während der Simulation berücksichtigt werden. Es sind verschiedene Verhalten denkbar und es sollte auch ein Mix aus zufälligen Verhalten (auch mit zufälligen Geschwindigkeiten) verwendet werden, um Overfitting zu reduzieren. In der nachfolgenden Tabelle sind Beispiele für erwartbares Gegnerverhalten dargestellt. Während das *Intercept*-Manöver eine ideale Schussverteidigung approximiert, sollten auch *Stand Still* oder *Gaussian* verwendet werden, um die etwas willkürliche Natur der SSL Division B zu modellieren. Verhalten wie *Pong* und *Drive To Target* eigenen sich besonders gut für den Torwart, Feldspieler können auch Verhaltensweisen wie *Closest Enemy* zugewiesen bekommen.

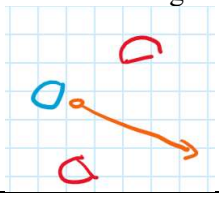
Nachdem das Szenario generiert wurde, wird vom Schützen der Ball geschossen (hierbei sollte in der Simulation auch die Streuung des Kickers und das Verhalten der Gegner berücksichtigt werden). Nach etwa 2 Sekunden Simulationszeit wird überprüft, ob ein Tor erzielt wurde und diese Information zusammen mit den Features zum Startzeitpunkt der Simulation in einer csv-Tabelle gespeichert. Mit jedem Simulationsdurchlauf wird eine weitere Zeile in die csv-Tabelle eingefügt (siehe Bild). Diese Tabelle ist die Grundlage für den nachfolgenden Trainingsprozess, bei dem der Zusammenhang der Features (Abstand zum Tor, Nähe der Gegner…) und dem Ergebnis (Tor / kein Tor) hergestellt werden soll. Es empfielt sich die Simulation beschleunigt und mit mehreren parallelen Instanzen durchzuführen.

WICHTIG: Es kann vorkommen, dass bei zu starken Gegnern nur äußerst selten Tore fallen. Es ist allerdings wichtig, dass die Trainingsdaten ausgeglichen sind, also jede Klasse etwa gleich oft vorkommt. Dies kann entweder durch eine geschickte Generation geschehen (Schütze oft nah am Tor, schaut sehr wahrscheinlich Richtung Tor), oder indem Treffer später stärker gewichtet werden (Gewicht Tor = Fehlschüsse / Gesamtzahl Schüsse; Gewicht kein Tor = Treffer / Gesamtzahl Schüsse).

Gegnerverhalten während der Simulation

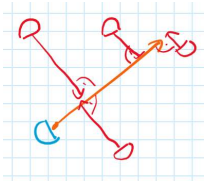
Stand Still:

➔ Roboter bewegen sich nicht



Intercept:

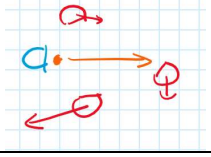
➔ Fahre auf kürzestem Weg zu Schusslinie



Gaussian:

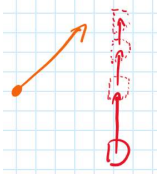
➔ Fahrt in zufällige Richtung

➔ Fahrt mit zufälliger Geschwindigkeit



Pong:

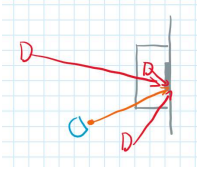
➔ Kopiere y-Position des Balls



Drive To Target:

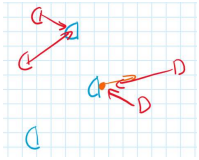
➔ Schätze Schnittpunkt von Torlinie und Schussbahn

➔ Fahre zum Schnittpunkt (ggf. Strafraum beachten)



Closest Enemy:

➔ Fahre zum nächsten Gegner



Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
goal	dist_ball_goal	goal_angle	est_shooting_error	lowest_dist_enemy_arc	avrg_dist_enemy_arc	num_enemies_closer_30cm	num_enemies_clo
0	2.318807	0.3479372	1.309945	0.6796782	2.117545	0	1
0	3.210909	0.4584826	0.9807542	0.2483601	1.234285	1	2
1	2.395676	0.4697456	0.2241197	0.5198879	0.5198879	0	1
1	2.80907	0.05909776	0.1717244	0.994158	0.994158	0	1
0	5.673707	0.2590371	0.800692	0.04925305	0.7677177	2	5
0	1.515362	1.499682	1.304521	0.5806326	2.961927	0	1
1	2.589319	0.232539	0.08530591	0.03495786	1.679483	1	1
1	1.689004	0.5933359	0.244614	0.3862527	0.3862527	0	1
0	5.593705	0.6582029	1.884868	1.206623	2.658165	0	0
1	2.366253	1.156117	0.5501188	0.2494188	3.460799	1	1
0	6.882394	0.3260787	0.7696968	0.1523667	0.4152985	1	4
0	3.43838	0.213109	0.6411991	0.04774465	1.765756	1	1
0	5.708242	0.04156538	0.1543233	1.023245	2.034089	0	0
0	7.385169	0.1958009	1.41181	0.2682469	1.598842	1	2
1	3.899253	0.4363827	0.5607135	0.2180958	1.806468	2	2
0	4.587988	0.7242958	2.027781	0.4259114	1.75101	0	1
0	5.615498	0.5504435	2.767974	0.5440404	1.956262	0	1
0	5.171366	0.9637549	3.586319	1.170151	1.954526	0	0

Abbildung 1: Beispielhafter Aufbau einer csv-Datei mit Trainingsdaten

4.2.3 Training des Modells in Tensorflow / Keras

Zum Trainieren wird ein Python-Skript sowie die im vorherigen Abschnitt erstellte csv-Tabelle verwendet. Da bereits bekannt ist, ob in einem bestimmten Szenario ein Tor gefallen ist oder nicht, liegt ein überwachtes Lernverfahren (*supervised learning*) vor. Zudem handelt es sich um eine Klassifikation, da jedes Szenario der Kategorie „Tor“ oder „kein Tor“ zugeordnet werden soll. Dies kann mittels eines **Random Forests** gelöst werden. Tensorflow bzw. das darauf aufbauende Keras stellt alle dafür benötigten Funktionen bereits zur Verfügung.

- Tensorflows offizielle Dokumentation zu Random Forests, inklusive Beispiele und Tutorials: https://www.tensorflow.org/decision_forests
- Ausführliche Dokumentation aller Parameter in Tensorflow: https://www.tensorflow.org/decision_forests/api_docs/python/tfdf/keras/RandomForestModel
- Weiteres Beispiel: https://keras.io/examples/structured_data/classification_with_tfdf/

Einschub: Random Forest im Detail

Bei Random Forest handelt es sich um Klassifikatoren aus dem Bereich des überwachten Lernens. Ein Forest besteht aus mehreren Bäumen (siehe Einschub: Klassifikation). Beim Training wird eine Vielzahl an Bäumen erstellt, mit denen sich die Klassifikation durchführen lässt. Jeder Baum verwendet dabei jedoch nur einen zufälligen Teil der Trainingsdaten (wird als *Bootstrapping* bezeichnet) und nur eine zufällige Auswahl der vorhandenen Features. Dadurch wird garantiert, dass die Bäume untereinander möglichst wenig redundante Informationen enthalten. Die Gesamtentscheidung erfolgt dann durch einen Mehrheitsentscheid (*Majority Voting*), bei der zunächst jeder Baum die für ihn relevanten Features eines unbekannten Szenarios anschaut und darauf basierend eine Entscheidung trifft (z.B. „Tor“). Die Gesamtentscheidung ergibt sich dann durch die Mehrheit (z.B. 75% voten für „Tor“, 25% voten für „kein Tor“ => Entscheidung „Tor“).

Die Wahl welche Features in welchem Baum und auch in welcher Reihenfolge sie verwendet werden sollen, wird vom Trainingsalgorithmus automatisch übernommen. Features, die keinen großen Einfluss haben (z.B. Gegner die weiter als 2m von der Schusslinie entfernt sind) werden dabei seltener verwendet und können ggf. sogar aus der Feature-Liste entfernt werden.

- Youtube Empfehlung zu Random Forests (8 Minuten): https://www.youtube.com/watch?v=v6VJ2RO66Ag
- Blogpost mit Praxisbeispiel und Vergleich zu Decision Trees: https://towardsdatascience.com/random-forests-algorithm-explained-with-a-real-life-example-and-some-python-code-affbfa5a942c
- Weiterer Blogpost: https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

Aufbau eines Trainingsskripts

1. Importieren der Bibliotheken (vor allem Tensorflow, pandas und numpy)
2. Einlesen der csv-Datei mittels
```
dataset = pandas.read_csv("<path>")
```
3. Aufteilen der Daten in Trainings- und Testdaten (meist etwa 80% Training, 20% Test). Dafür gibt's von Pandas die Funktionen:
```
train_data = dataset.sample(frac=0.8)
test_data = dataset.drop(train_data)
```
4. Definition der bei der Klassifikation verwendeten Kategorien (in diesem Fall „0" und „1"), sowie Name der Spalte in der die Ergebnisse der Simulation stehen (hier „goal"). Das geht mittels:
```
TARGET_COLUMN_NAME = "goal"
TARGET_LABELS = ["0", "1"]
```
5. Namen der numerischen und kategorischen Features definieren (= Namen der Spalten)
```
NUMERIC_FEATURE_NAMES = ["dist_ball_goal", "goal_angle", …]
CATEGORICAL_FEATURE_NAMES = [] #für dieses Beispiel irrelevant
```
6. Konvertierung der Labels von Strings in Integer
```
def prepare_dataframe(dataframe):
    for feature_name in CATEGORICAL_FEATURE_NAMES:
        dataframe[feature_name] = dataframe[feature_name].astype(str)
prepare_dataframe(train_data)
prepare_dataframe(test_data)
```
7. Definition der Hyperparameter (können auch automatisch ermittelt werden, mehr dazu später)
```
NUM_TREES = 1000
MIN_EXAMPLES = 6
MAX_DEPTH = 5
SUBSAMPLE = 0.65
SAMPLING_METHOD = "RANDOM"
VALIDATION_RATIO = 0.2
```
8. Funktionen zum Anlernen des Modells, ggf. Gewichte ergänzen (hier auskommentiert). Diese können aus Beispielen übernommen werden, davon sind online einige verfügbar.
```
def run_experiment(model, train_data, test_data, num_epochs=1,
batch_size=None):
    train_dataset = tfdf.keras.pd_dataframe_to_tf_dataset(
        train_data, label=TARGET_COLUMN_NAME#, weight=WEIGHT_COLUMN_NAME
    )
    test_dataset = tfdf.keras.pd_dataframe_to_tf_dataset(
        test_data, label=TARGET_COLUMN_NAME#, weight=WEIGHT_COLUMN_NAME
    )
    model.fit(train_dataset, epochs=num_epochs, batch_size=batch_size)
    _, accuracy = model.evaluate(test_dataset, verbose=0)
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")

def specify_feature_usages():
    feature_usages = []
    for feature_name in NUMERIC_FEATURE_NAMES:
        feature_usage = tfdf.keras.FeatureUsage(
            name=feature_name, semantic=tfdf.keras.FeatureSemantic.NUMERICAL
        )
        feature_usages.append(feature_usage)
    for feature_name in CATEGORICAL_FEATURE_NAMES:
        feature_usage = tfdf.keras.FeatureUsage(
            name=feature_name, semantic=tfdf.keras.FeatureSemantic.CATEGORICAL
        )
        feature_usages.append(feature_usage)
    return feature_usages
```

```
def create_gbt_model():
    gbt_model = tfdf.keras.GradientBoostedTreesModel(
        features=specify_feature_usages(),
        exclude_non_specified_features=True,
        num_trees=NUM_TREES,
        max_depth=MAX_DEPTH,
        min_examples=MIN_EXAMPLES,
        subsample=SUBSAMPLE,
        validation_ratio=VALIDATION_RATIO,
        task=tfdf.keras.Task.CLASSIFICATION,
    )
    gbt_model.compile(metrics=[keras.metrics.BinaryAccuracy(name="accuracy")])
    return gbt_model
    gbt_model.compile(metrics=[keras.metrics.BinaryAccuracy(name="accuracy")])
    return gbt_model
```

9. Ausführen und Anzeigen der Ergebnisse:
```
gbt_model = create_gbt_model()
run_experiment(gbt_model, train_data, test_data)
print(gbt_model.summary())
```
10. Interpretation der Ergebnisse und ggf. Anpassung des Trainingsszenarios
11. Speichern des Modells, damit dieses später im C++ Projekt importiert werden kann

Interpretation

Der abschließende Report gibt Aufschluss über die Güte des Modells und der verwendeten Features. Das nachfolgende Bild zeigt die Wichtigkeit der Features an. Es ist deutlich zu sehen, dass der *shooting Error* und *Distance to Goal* die beiden einzig wirklich aussagekräftigen Features sind. Alle anderen Features sind unwichtig und werden für die Klassifikation „Tor“ / „kein Tor“ im Grunde nicht benötigt.

Auch die Position der Gegner ist kaum relevant. Das liegt daran, dass bei Aufnahme der Trainingsdaten die Gegner nur das Verhalten *Gaussian* ausgeführt und somit nicht aktiv versucht haben, den Torschuss zu verhindern. Somit war das hier verwendete Trainingszenario unbrauchbar und sollte durch eines ersetzt werden, bei dem die Gegner auch mit *Intercept* oder ähnlichen Verhalten versuchen, das Tor aktiv zu verhindern. Es ist zu erwarten, dass vor allem Features wie *Anzahl Gegner näher als 10cm an Schusslinie* dadurch stark an Bedeutung gewinnen. Generell empfiehlt es sich, einen zufälligen Mix aus verschiedenen Verhalten zu verwenden, um möglichst viele Spielweisen des Gegners abzudecken.

```
Variable Importance: SUM_SCORE:
    1.      "est_shooting_error" 2026227.102533 ################
    2.          "dist_ball_goal" 1852675.131683 ##############
    3.   "lowest_dist_enemy_arc" 120876.926504
    4.    "lowest_dist_ally_arc" 101391.840334
    5.              "goal_angle" 32354.932081
    6.      "num_enemy_angle_95" 13516.942915
    7.     "avrg_dist_enemy_arc" 8471.666391
    8.      "num_enemy_angle_90" 2330.931543
```

Die Gesamtgüte des Modells kann durch die erzielte Accuracy beurteilt werden. Diese gibt an, wie viel Prozent der beim Training nicht verwendeten Testdaten nach Abschluss des Trainings richtig klassifiziert werden konnten. Damit diese jedoch aussagekräftig ist, sollten alle Klassen etwa gleich oft vorkommen (siehe Hinweis im nachfolgenden Kapitel zu SciKit-Learn). Generell ist eine Accuracy von >90% wünschenswert. Es ist zu beachten, dass eine hohe Accuracy auf Overfitting hindeuten kann (wenn Test Accuracy >> Training Accuracy), bei einer niedrigen Accuracy liegt eventuell Underfitting vor. In diesem Fall ist das Problem für das verwendete Modell zu komplex bzw. die Daten zu zufällig, um einen Zusammenhang herstellen zu können. Eine Lösung kann eine Anpassung der Hyperparameter und/oder der Features darstellen, zudem sollte ein größerer Datensatz für das Training verwendet werden.

4.2.4 Automatisierung und Hyperparamteroptimierung mittels SciKit-Learn

Statt der wie im vorherigen Abschnitt beschriebenen manuellen Festlegung und Parametrisierung eines Trainingsalgorithmus, können auch Auto-ML Methoden verwendet werden. Diese automatisieren den Prozess und legen auch wichtige Hyperparameter (wie die Menge an Bäumen, die maximale Größe eines Baumes...) fest. Tutorials und weitere Informationen finden sich am Ende des Abschnitts.

Vorgehen zur Ermittlung der Torwahrscheinlichkeit

1. Importieren wichtiger Bibliotheken (autosklearn, sklearn, pandas, pprint)
```
import autosklearn.classification
from sklearn import (model_selection, datasets, metrics)
import pandas as pd
from pprint import pprint
from sklearn.model_selection import train_test_split, StratifiedKFold
import random
from autosklearn.metrics import (accuracy, f1, roc_auc, precision, average_precision, recall, log_loss)
```
2. Einlesen der csv-Datei mit den Trainingsdaten durch pandas
```
raw_data = pd.read_csv("<path>")
```
3. Zuordnung der Spalten zu Kategorisch/Numerisch
```
num_cols=["dist_ball_goal", "goal_angle",…]
cat_cols=["goal"]
```
4. Konvertierung der Daten in Kategorien/numerische Werte
```
df[num_cols] = df[num_cols].apply(pd.to_numeric)
df[cat_cols] = df[cat_cols].apply(pd.Categorical)
```
5. Ausgangs- und Eingangsvariablen festlegen (Ausgang y und Eingang X)
```
y = df.pop('goal')
X = df.copy()
```
6. Trainings-/Testsplit. Generell sollte man etwa 70-80% der Daten alsTrainingsdaten und entsprechen 20-30% als Testdaten verwenden.
```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.2, random_state=1, stratify=y)
```
7. Definition der Trainingsfunktion (kann aus Internetbeispielen übernommen werden)
   Die Trainingszeiten können erhöht werden, dann dauert die Ausführung länger, aber es werden mehr Modelle untersucht
```
automl = autosklearn.classification.AutoSklearnClassifier(
    time_left_for_this_task=120,       #Gesamtzeit des Trainings in s
    per_run_time_limit=30,             #Zeit pro Model in s
    tmp_folder="<Speicherpfad>" + str(random.randrange(99999)),
    n_jobs=8,
    resampling_strategy='holdout',
    resampling_strategy_arguments={'folds':5},
    metric = average_precision,
    scoring_functions=[roc_auc, average_precision, accuracy, f1, precision,
recall, log_loss]
)
```
8. Aufruf der Trainingsfunktion (= „fitten“ des Modells):
```
if __name__ == '__main__':
automl.fit(X_train, y_train)
```
   ACHTUNG: Fehlt die Abfrage, ob es sich um den Main-Thread handelt, kann das gesamte Programm evtl. einfrieren!

9. Anzeigen der Trainingsergebnisse
```
print("\n\n ====== leaderboard ======")
print(automl.leaderboard())
print("\n\n ====== models ======")
pprint(automl.show_models(), indent=4)
print("\n\n ====== prediction ======")
predictions = automl.predict(X_test)
print("Accuracy score:", sklearn.metrics.accuracy_score(y_test, predictions))
```

Erklärung und Interpretation:

Beim Auto-ML werden verschiedene Modelle mit verschiedenen Hyperparametern ausprobiert und optimiert. Am Ende wird ein Leaderboard ausgegeben, das angibt, welche Modelle die besten Ergebnisse erzielen konnten:

```
 ====== leaderboard ======
          rank  ...  duration
model_id        ...
281          1  ...  1.169862
248          2  ...  1.247445
237          3  ...  1.127138
271          4  ...  1.290632
234          5  ...  1.180188
181          6  ...  1.314683
13           7  ...  0.913168
191          8  ...  1.349392
235          9  ...  1.245892

[9 rows x 5 columns]
```

In diesem Fall hat Modell Nr. 281 die besten Ergebnisse erzielt. Die Details zu dem Modell finden sich weiter unten in der gedruckten Ausgabe:

```
281: {   'balancing': Balancing(random_state=1),
         'classifier': <autosklearn.pipeline.components.classification.ClassifierChoice object at 0x7f40728b8bb0>,
         'cost': 0.03827073864136343,
         'data_preprocessor': <autosklearn.pipeline.components.data_preprocessing.DataPreprocessorChoice object at 0x7f406de42b20>,
         'ensemble_weight': 0.1,
         'feature_preprocessor': <autosklearn.pipeline.components.feature_preprocessing.FeaturePreprocessorChoice object at 0x7f40728b8c40>,
         'model_id': 281,
         'rank': 1,
         'sklearn_classifier': RandomForestClassifier(bootstrap=False, criterion='entropy', max_features=20,
                               min_samples_leaf=10, min_samples_split=10,
                               n_estimators=512, n_jobs=1, random_state=1,
                               warm_start=True)}}
```

Wie hier zu sehen ist, handelt es sich bei Modell 281 um einen Random Forest Classifier. Außerdem werden geeignete Hyperparameter definiert. So soll der Forest aus 512 einzelnen Bäumen aufgebaut werden, die jeweils maximal 20 Features betrachten dürfen etc. Diese Informationen können im tensorflow aus dem vorherigen Abschnitt bei Schritt 7 eingetragen werden.

Der Accuracy Score gibt an, wie viel Prozent der Testdaten richtig klassifiziert wurden. Je höher der Accuracy Score, desto besser. Ein sehr hoher Score (>95%) kann auf Overfitting hindeuten, ein kleiner Score (<70%) auf Underfitting. In diesen Fällen muss das Trainingszenario angepasst werden.

```
 ====== prediction ======
Accuracy score: 0.98
```

ACHTUNG: sind in den Test- und Trainingsdaten nur etwa 2% der Fälle überhaupt tatsächlich auch Tore, dann wird die Accuracy auch dann 0.98 sein, wenn immer vorhergesagt wird, dass kein Tor fallen wird. In dem Fall ist das Modell absolut unbrauchbar, da die Fälle, in denen ein Tor fällt, einfach komplett ignoriert werden. Ein möglicher Fix ist die Anpassung des Trainingszenarios, sodass mehr Tore fallen (z.B. durch eine wahrscheinlichere Ausrichtung des Schützens Richtung Tor, um weniger Schüsse „ins Leere“ zu produzieren), oder die Einführung eines Gewichtungsfaktors, damit erzielte Tore stärker berücksichtigt werden. Alternativ können auch die Daten ohne Tor ausgedünnt werden.

Weitere Informationen zu auto-ML mit SciKit-Learn (sklearn):

- Einführung in sklearn mit Decision Tree Classifier: https://machinelearningmastery.com/a-gentle-introduction-to-scikit-learn-a-python-machine-learning-library/
- Weitere Einführung mit mehreren Praxisbeispielen: https://docs.neptune.ai/integrations/sklearn/
- Einführung für Beginner mit kleinschrittiger Erklärung: https://towardsdatascience.com/data-science-101-start-with-pandas-scikit-learn-and-google-colab-ecbb6a247cc9