1 Was ist Reinforcement Learning?

Dieses Kapitel beinhaltet:

- einen kurzen Überblick über Machine Learning (maschinelles Lernen),
- die Einführung von Reinforcement Learning (bestärkendes Lernen – RL) als Teilbereich,
- das grundlegende Framework (Programmiergerüst) des Reinforcement Learning.

Computersprachen der Zukunft werden sich mehr mit Zielen und weniger mit vom Programmierer vorgegebenen Prozeduren beschäftigen.

Marvin Minksy, 1970 ACM-Turing-Vorlesung

Wenn Sie dieses Buch lesen, sind Sie wahrscheinlich damit vertraut, wie tiefe neuronale Netze für Dinge wie Bildklassifikation oder Vorhersagen verwendet werden (und wenn nicht, lesen Sie einfach weiter; wir haben auch einen Crash-Kurs zu Deep Learning (tiefem Lernen) im Anhang). *Deep Reinforcement Learning* (DRL) ist ein Teilgebiet des Machine Learning, das Deep-Learning-Modelle (d. h. neuronale Netze) für Reinforcement-Learning-(RL)-Aufgaben (siehe Abschnitt 1.2 verwendet. Bei der Bildklassifizierung haben wir eine Menge von Bildern, die einer Reihe von diskreten Kategorien entsprechen, wie z. B. Bilder von verschiedenen Tierarten, und wir möchten, dass ein Machine-Learning-Modell ein Bild interpretiert und die Tierart auf dem Bild klassifiziert (siehe Bild 1.1).

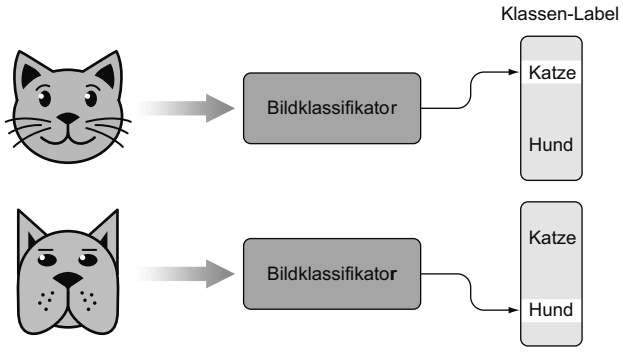

Bild 1.1 Ein Bildklassifikator ist eine Funktion oder ein Lernalgorithmus, der ein Bild aufnimmt und ein Klassenlabel zurückgibt, das das Bild einer Kategorie oder Klasse zuordnet, die aus einer endlichen Anzahl möglicher Kategorien ausgewählt wurde.

1.1 Das „Tiefe“ beim Deep Reinforcement Learning

Deep-Learning-Modelle sind nur eine von vielen Arten von Machine-Learning-Modellen, die wir zur Klassifikation von Bildern verwenden können. Im Allgemeinen brauchen wir nur eine Art Funktion, die ein Bild aufnimmt und ein Klassenlabel zurückgibt (in diesem Fall das Label, das angibt, welche Art von Tier im Bild dargestellt ist), und normalerweise hat diese Funktion einen festen Satz einstellbarer *Parameter* – wir nennen diese Art von Modellen

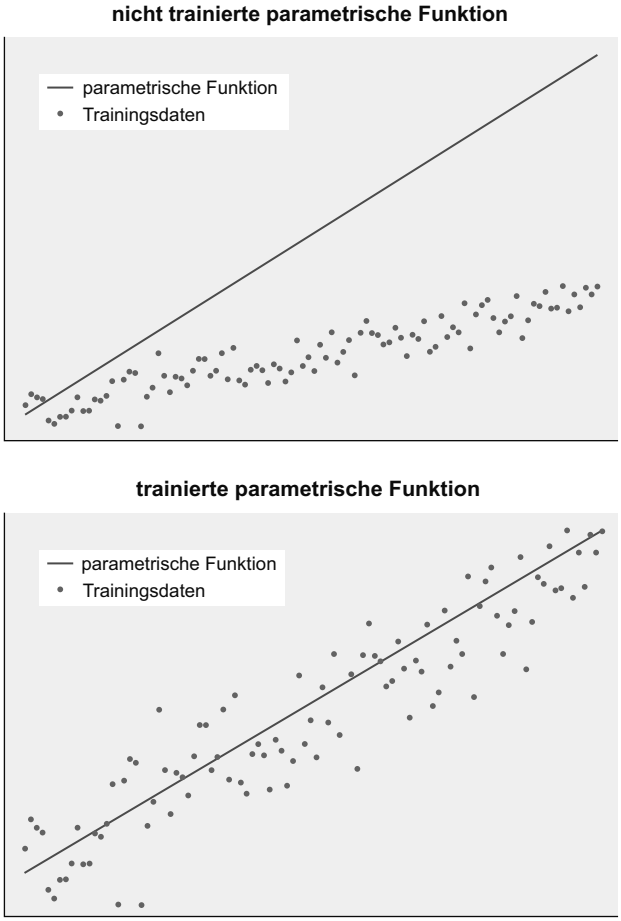

Bild 1.2 Das vielleicht einfachste Machine-Learning-Modell ist eine lineare Funktion der Form $f(x) = mx + b$ mit den Parametern m (die Steigung) und b (der Achsenabschnitt). Da es einstellbare Parameter hat, nennen wir es eine *parametrische* Funktion oder ein *parametrisches* Modell. Wenn wir einige zweidimensionale Daten haben, können wir mit einem zufällig initialisierten Satz von Parametern beginnen, wie z. B. [m = 3,4, b = 0,3], und dann einen Trainingsalgorithmus verwenden, um die Parameter so zu optimieren, dass sie zu den Trainingsdaten passen, wobei der optimale Satz von Parametern nahe bei [m = 2, b = 1] liegt.

parametrische Modelle. Wir beginnen mit einem parametrischen Modell, dessen Parameter mit Zufallswerten initialisiert werden – dies erzeugt zufällige Klassenlabels für die Eingabebilder. Dann verwenden wir eine *Trainingsprozedur*, um die Parameter so anzupassen, dass die Funktion iterativ immer besser darin wird, die Bilder korrekt zu klassifizieren. Irgendwann werden die Parameter bei einem optimalen Satz von Werten liegen, was bedeutet, dass das Modell bei der Klassifizierungsaufgabe nicht mehr besser werden kann. Parametrische Modelle können auch für die *Regression* verwendet werden, bei der wir versuchen, ein Modell an einen Datensatz anzupassen, damit wir Vorhersagen für ungesehene Daten machen können (Bild 1.2). Ein ausgefeilterer Ansatz könnte sogar noch besser funktionieren, wenn er mehr Parameter oder eine bessere interne Architektur hätte.

Tiefe neuronale Netze sind beliebt, weil sie in vielen Fällen die genauesten parametrischen Machine-Learning-Modelle für eine bestimmte Aufgabe, wie z. B. die Bildklassifikation, sind. Dies ist weitgehend auf die Art und Weise zurückzuführen, wie sie Daten repräsentieren. Tiefe neuronale Netze haben viele Schichten (daher die „Tiefe“), was das Modell dazu veranlasst, geschichtete Darstellungen von Eingabedaten zu lernen. Diese Schichtdarstellung ist eine Form der Kompositionalität, was bedeutet, dass ein komplexes Datenstück als Kombination von elementareren Komponenten dargestellt wird, und diese Komponenten können weiter in noch einfachere Komponenten zerlegt werden, und so weiter, bis man zu atomaren Einheiten gelangt.

Die menschliche Sprache ist kompositorisch (Bild 1.3). Ein Buch besteht zum Beispiel aus Kapiteln, Kapitel aus Absätzen, Absätze aus Sätzen usw., bis man zu einzelnen Wörtern kommt, die die kleinsten Bedeutungseinheiten darstellen. Doch jede einzelne Ebene vermittelt Bedeutung – ein ganzes Buch soll Bedeutung vermitteln, und seine einzelnen Absätze sollen kleinere Punkte vermitteln.

Tiefe neuronale Netze können ebenfalls eine kompositorische Darstellung von Daten erlernen – sie können beispielsweise ein Bild als die Zusammensetzung primitiver Konturen und Texturen darstellen, die in elementare Formen zusammengesetzt werden, und so weiter, bis man ein vollständiges, komplexes Bild erhält. Diese Fähigkeit, Komplexität mit kompositorischen Darstellungen zu handhaben, macht Deep Learning so wirkungsvoll.

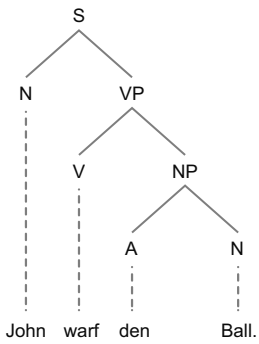

Bild 1.3 Ein Satz wie „John warf den Ball „ kann in immer einfachere Teile zerlegt werden, bis wir die einzelnen Wörter erhalten. In diesem Fall können wir den Satz (mit S bezeichnet) in ein Nomen (N) und eine Verbalphrase (VP) zerlegen. Die VP kann weiter in ein Verb „warf“ und eine Nominalphrase (NP) zerlegt werden. Die NP kann dann in die einzelnen Wörter „den“ und „Ball“ zerlegt werden.

1.2 Reinforcement Learning

Es ist wichtig, zwischen Problemen und ihren Lösungen zu unterscheiden oder, mit anderen Worten, zwischen den Aufgaben, die wir lösen wollen, und den Algorithmen, die wir zu ihrer Lösung entwerfen. Deep-Learning-Algorithmen können auf viele Problemtypen und Aufgaben angewendet werden. Bildklassifikations- und Vorhersageaufgaben sind häufige Anwendungen des Deep Learning, da die automatisierte Bildverarbeitung vor dem Deep Learning angesichts der Komplexität der Bilder sehr begrenzt war. Aber es gibt noch viele andere Aufgaben, die wir vielleicht automatisieren möchten, wie zum Beispiel Autofahren oder das Ausbalancieren eines Portfolios von Aktien und anderen Vermögenswerten. Zum Autofahren gehört ein gewisses Maß an Bildverarbeitung, aber noch wichtiger ist, dass der Algorithmus lernen muss, wie er *sich verhalten* muss, und nicht nur klassifizieren oder vorhersagen kann. Diese Probleme, bei denen Entscheidungen getroffen werden müssen oder bestimmte Verhaltensweisen umgesetzt werden müssen, werden kollektiv als *Steuerungsaufgaben* bezeichnet.

Reinforcement Learning ist ein generisches Framework für die Darstellung und Lösung von Steuerungsaufgaben, aber innerhalb dieses Rahmens können wir frei wählen, welche Algorithmen wir auf eine bestimmte Steuerungsaufgabe anwenden wollen(Bild 1.4). Deep-Learning-Algorithmen sind dafür prädestiniert, da sie in der Lage sind, komplexe Daten effizient zu verarbeiten. Aus diesem Grund konzentrieren wir uns auf *Deep* Reinforcement Learning, aber vieles von dem, was Sie in diesem Buch lernen werden, ist das allgemeine Reinforcement-Framework für Steuerungsaufgaben (siehe Bild 1.5). Dann werden wir uns ansehen, wie Sie ein geeignetes Deep-Learning-Modell modellieren können, das zum Framework passt und eine Aufgabe löst. Das bedeutet, dass Sie eine Menge über Reinforcement Learning lernen werden, und Sie werden wahrscheinlich auch einiges über Deep Learning lernen, was Sie bisher noch nicht wussten.

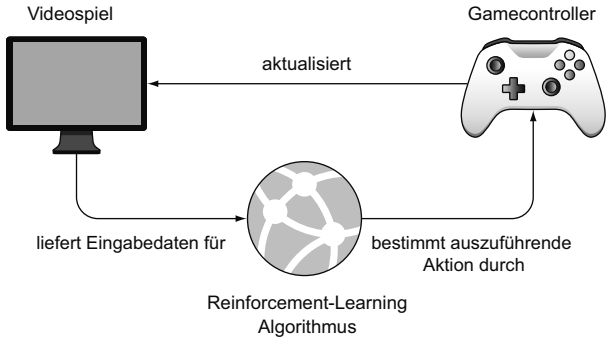

Bild 1.4 Im Gegensatz zu einem Bildklassifikator interagiert ein Reinforcement-Learning-Algorithmus dynamisch mit Daten. Er konsumiert kontinuierlich Daten und entscheidet, welche Aktionen durchgeführt werden sollen – Aktionen, die die ihm nachfolgend präsentierten Daten verändern. Ein Videospielbildschirm könnte die Eingabedaten für einen RL-Algorithmus darstellen, der dann unter Verwendung des Gamecontrollers entscheidet, welche Aktion zu ergreifen ist, und dies bewirkt eine Aktualisierung des Spiels (z. B. der Spieler bewegt sich oder feuert eine Waffe ab).

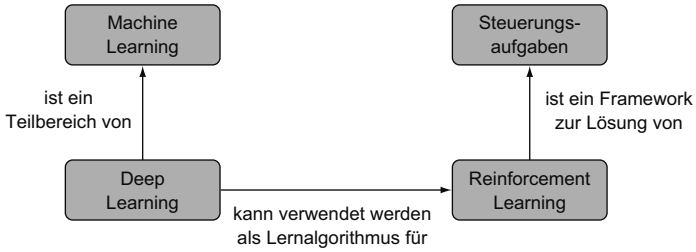

Bild 1.5 Deep Learning ist ein Teilbereich des Machine Learning. Deep-Learning-Algorithmen können zur Unterstützung von RL-Ansätzen zur Lösung von Steuerungsaufgaben verwendet werden.

Eine zusätzliche Komplexität beim Übergang von der Bildverarbeitung zum Bereich der Steuerungsaufgaben bringt das Element der Zeit mit sich. Bei der Bildverarbeitung trainieren wir normalerweise einen Deep-Learning-Algorithmus an einem festen Datensatz von Bildern. Nach einer ausreichenden Menge an Training erhalten wir in der Regel einen Hochleistungsalgorithmus, den wir auf einige neue, ungesehene Bilder anwenden können. Wir können uns den Datensatz als einen „Datenraum" vorstellen, in dem ähnliche Bilder in diesem abstrakten Raum näher beieinander liegen und unterschiedliche Bilder weiter voneinander entfernt sind (Bild 1.6).

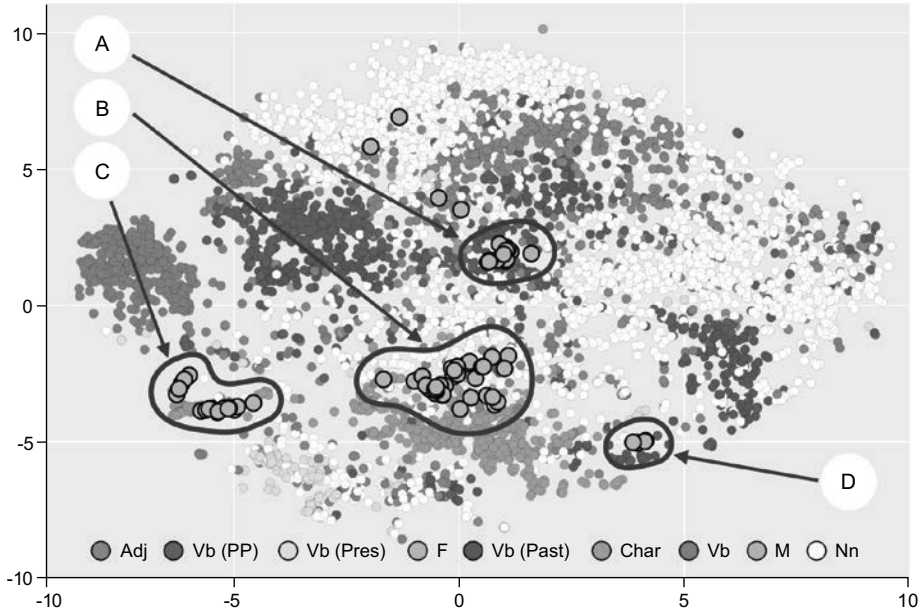

Bild 1.6 Diese grafische Darstellung von Wörtern in einem 2-D-Raum zeigt jedes Wort als einen farbigen Punkt. Ähnliche Wörter gruppieren sich und unähnliche Wörter liegen weiter auseinander. Daten leben natürlich in einer Art „Raum", wobei ähnliche Daten näher beieinander leben. Die Labels A, B, C und D weisen auf bestimmte Cluster von Wörtern hin, die eine gemeinsame Semantik haben.

Bei Kontrollaufgaben haben wir in ähnlicher Weise einen Raum von zu verarbeitenden Daten, aber jede Dateneinheit hat auch eine zeitliche Dimension – Daten existieren in Zeit und Raum. Das bedeutet, dass das, was der Algorithmus zu einem bestimmten Zeitpunkt entscheidet, von dem beeinflusst wird, was zu einem früheren Zeitpunkt geschah. Dies ist bei gewöhnlicher Bildklassifizierung und ähnlichen Problemen nicht der Fall. Die Zeit macht die Trainingsaufgabe dynamisch – der Datensatz, auf dem der Algorithmus trainiert, ist nicht unbedingt fixiert, sondern ändert sich aufgrund der Entscheidungen, die der Algorithmus trifft.

Gewöhnliche bildklassifikationsähnliche Aufgaben fallen in die Kategorie des *Supervised Learning* (überwachtes Lernen), da der Algorithmus darauf trainiert wird, Bilder richtig zu klassifizieren, indem er die richtigen Antworten gibt. Der Algorithmus stellt zunächst nach dem Zufallsprinzip Vermutungen an, und er wird iterativ korrigiert, bis er die Features (Merkmale) im Bild lernt, die dem passenden Label entsprechen. Dies setzt voraus, dass wir bereits wissen, was die richtigen Antworten sind, was umständlich sein kann. Wenn Sie einen Deep-Learning-Algorithmus trainieren wollen, um Bilder von verschiedenen Pflanzenarten korrekt zu klassifizieren, müssten Sie mühsam Tausende solcher Bilder erfassen, jedem Bild manuell Klassenlabels zuordnen und die Daten in einem Format aufbereiten, mit dem ein Algorithmus für Machine Learning arbeiten kann, in der Regel in einer Art Matrix.

Im Gegensatz dazu wissen wir bei RL nicht bei jedem Schritt genau, was das Richtige ist. Wir müssen nur wissen, was das letztendliche Ziel ist und welche Dinge man vermeiden sollte. Wie bringt man einem Hund einen Trick bei? Man muss ihm Leckerbissen geben. In ähnlicher Weise trainieren wir einen RL-Algorithmus, indem wir ihm einen Anreiz geben, ein hochgestecktes Ziel zu erreichen, und ihn möglicherweise davon abhalten, Dinge zu tun, die wir nicht wollen, dass er sie tut. Im Falle eines selbstfahrenden Autos könnte das hochrangige Ziel sein, „unfallfrei von Ausgangspunkt A aus zu Punkt B zu gelangen“. Wenn er die Aufgabe erfüllt, belohnen wir ihn, und wenn er einen Unfall verursacht, bestrafen wir ihn. Wir würden das alles in einem Simulator machen, statt auf den realen Straßen, sodass wir ihn immer wieder versuchen lassen könnten an der Aufgabe zu scheitern, bis er lernt und belohnt wird.

TIPP

In der natürlichen Sprache bedeutet „Belohnung“ immer etwas Positives, während es im Jargon des Reinforcement Learning eine zu optimierende numerische Größe ist. Eine Belohnung kann also positiv oder negativ sein. Wenn sie positiv ist, entspricht sie dem natürlichsprachlichen Gebrauch des Begriffs, wenn sie negativ ist, entspricht sie dem natürlichsprachlichen Wort „Strafe“.

Der Algorithmus hat ein einziges Ziel – die Maximierung seiner Belohnung – und um dieses zu erreichen, muss er mehr elementare Fähigkeiten erlernen, um das Hauptziel zu erreichen. Wir können auch negative Belohnungen liefern, wenn der Algorithmus Dinge tut, die uns nicht gefallen. Da er versucht, seine Belohnung zu maximieren, wird er lernen, Aktionen zu vermeiden, die zu negativen Belohnungen führen. Aus diesem Grund wird es *Reinforcement Learning* genannt: Wir verstärken (*to reinforce*) bestimmte Verhaltensweisen entweder positiv oder negativ durch Belohnungssignale (siehe Bild 1.7). Das ist ganz ähnlich, wie Tiere lernen: Sie lernen, Dinge zu tun, mit denen sie sich gut oder zufrieden fühlen, und Dinge zu vermeiden, die Schmerzen verursachen.

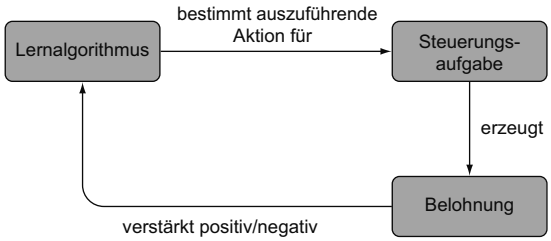

Bild 1.7 Im RL-Framework entscheidet eine Art Lernalgorithmus, welche Aktionen für eine Steuerungsaufgabe (z. B. das Steuern eines Staubsaugerroboters) durchzuführen sind, und die Aktion führt zu einer positiven oder negativen Belohnung, die diese Aktion positiv oder negativ verstärkt und somit den Lernalgorithmus trainiert.

1.3 Dynamische Programmierung versus Monte Carlo

Sie wissen jetzt, dass Sie einen Algorithmus trainieren können, um eine schwierige Aufgabe zu erledigen, indem Sie der Erledigung der Aufgabe eine hohe Belohnung (d. h. positive Verstärkung) zuweisen und Dinge, die wir nicht wollen, negativ bestärken. Lassen Sie uns das konkretisieren. Nehmen wir an, das vorrangige Ziel ist es, einen Staubsaugerroboter so zu trainieren, dass er sich von einem Raum in einem Haus zu seinem Dock, das sich in der Küche befindet, bewegt. Er hat vier Aktionen: nach links, nach rechts, vorwärts und rückwärts gehen. Zu jedem Zeitpunkt muss der Roboter entscheiden, welche dieser vier Aktionen er ausführen soll. Wenn er das Dock erreicht, erhält er eine Belohnung von +100, und wenn er auf seinem Weg an etwas stoßt, erhält er eine negative Belohnung von –10. Angenommen, der Roboter verfügt über eine vollständige 3-D-Karte des Hauses und hat die genaue Lage des Docks, aber er weiß immer noch nicht genau, welche Abfolge einfacher Aktionen er ausführen muss, um zum Dock zu gelangen.

Ein Ansatz zur Lösung dieses Problems heißt *dynamische Programmierung* (DP). Er wurde erstmals 1957 von Richard Bellman formuliert. Die dynamische Programmierung sollte besser als *Zielzerlegung* bezeichnet werden, da sie komplexe Probleme löst, indem sie diese in immer kleinere Teilprobleme zerlegt, bis sie zu einem einfachen Teilproblem gelangt, das ohne weitere Informationen gelöst werden kann.

Anstatt zu versuchen, mit einer langen Sequenz primitiver Aktionen zum Dock zu gelangen, kann der Roboter das Problem zunächst in „in diesem Raum bleiben“ versus „diesen Raum verlassen“ zerlegen. Da er über einen vollständigen Plan des Hauses verfügt, weiß er, dass er den Raum verlassen muss, da sich das Dock in der Küche befindet. Er weiß jedoch immer noch nicht, welche Abfolge von Aktionen es ihm erlaubt, den Raum zu verlassen, also gliedert er das Problem weiter auf in „sich auf die Tür zubewegen“ oder „sich von der Tür wegbewegen“. Da sich die Tür näher am Dock befindet und es einen Weg von der Tür zum Dock gibt, weiß der Roboter, dass er sich auf die Tür zubewegen muss, aber auch hier weiß er nicht, mit welcher Abfolge primitiver Aktionen er sich auf die Tür zubewegt.

Schließlich muss er entscheiden, ob er sich nach links, rechts, vorwärts oder rückwärts bewegen soll. Er sieht, dass sich die Tür vor ihm befindet, also bewegt er sich vorwärts. Er hält diesen Prozess aufrecht, bis er den Raum verlässt. Dann muss er noch einige Zielzerlegungen vornehmen, bis er zum Dock gelangt.

Das ist das Wesen der dynamischen Programmierung. Es handelt sich um einen generischen Ansatz zur Lösung bestimmter Arten von Problemen, die sich in Unter- und Teilprobleme zerlegen lassen, und es gibt Anwendungen in vielen Bereichen, darunter Bioinformatik, Wirtschaft und Informatik.

Um die dynamische Programmierung nach Bellman anwenden zu können, müssen wir in der Lage sein, unser Problem in Teilprobleme zu zerlegen, von denen wir wissen, wie sie zu lösen sind. Aber selbst diese scheinbar harmlose Annahme ist in der realen Welt schwer zu verwirklichen. Wie zerlegt man das hochgesteckte Ziel für ein selbstfahrendes Auto, „unfallfrei von Punkt A aus zu Punkt B zu gelangen", in kleine, unfallfreie Teilprobleme? Lernt ein Kind das Laufen, indem es zuerst die leichteren Teilprobleme löst? In RL, wo wir oft nuancierte Situationen haben, die ein gewisses Zufallsprinzip enthalten, können wir die dynamische Programmierung nicht genau so anwenden, wie Bellman es beschrieben hat. Tatsächlich kann DP als ein Extrem eines Kontinuums von Problemlösungstechniken betrachtet werden, bei dem das andere Ende zufälliges Ausprobieren wäre.

Eine andere Möglichkeit, dieses Lernkontinuum zu betrachten, besteht darin, dass wir in einigen Situationen ein maximales Wissen über die Umgebung haben und in anderen ein minimales Wissen über die Umgebung, und wir müssen in jedem Fall unterschiedliche Strategien anwenden. Wenn Sie die Toilette in Ihrem eigenen Haus benutzen müssen, wissen Sie genau (na ja, zumindest unbewusst), welche Abfolge von Muskelbewegungen Sie von jeder Ausgangsposition aus (d. h. von der dynamischen Programmierung her) zur Toilette bringen wird. Das liegt daran, dass Sie Ihr Haus sehr gut kennen – Sie haben ein mehr oder weniger perfektes *Modell* Ihres Hauses im Kopf. Wenn Sie auf einer Party in einem Haus sind, in dem Sie noch nie zuvor gewesen sind, müssen Sie sich vielleicht lange umschauen, bis Sie die Toilette alleine finden (durch Trial-and-Error), weil Sie kein gutes Modell des Hauses dieser Person haben.

Die Trial-and-Error-Strategie wird im Allgemeinen zu den *Monte-Carlo-Methoden* gezählt. Eine Monte-Carlo-Methode ist im Wesentlichen eine Stichprobe nach dem Zufallsprinzip aus der Umgebung. Bei vielen Problemen der realen Welt haben wir zumindest ein gewisses Wissen über die Umgebungssituation, sodass wir am Ende eine gemischte Strategie aus Trial-and-Error und einer Ausnutzung dessen, was wir bereits über die Umgebung wissen, anwenden, um die einfachen Teilziele direkt zu lösen.

Ein etwas albernes Beispiel für eine gemischte Strategie wäre, wenn man Ihnen die Augen verbindet, Sie an einen unbekannten Ort in Ihrem Haus bringt und Ihnen sagt, Sie sollen das Badezimmer finden, indem Sie Kieselsteine werfen und auf das Geräusch achten. Sie könnten damit beginnen, das übergeordnete Ziel (das Badezimmer zu finden) in ein leichter zugängliches Unterziel zu zerlegen: Finden Sie heraus, in welchem Raum Sie sich gerade befinden. Um dieses Unterziel zu lösen, könnten Sie nach dem Zufallsprinzip ein paar Kieselsteine in zufällige Richtungen werfen und die Größe des Raumes abschätzen, was Ihnen genügend Informationen liefert, um daraus abzuleiten, in welchem Raum Sie sich befinden – sagen wir im Schlafzimmer. Dann müssten Sie zu einem weiteren Unterziel schwenken: zur Tür navigieren, damit Sie den Flur betreten können. Sie würden wieder anfangen, Kieselsteine

zu werfen, aber da Sie sich an die Ergebnisse Ihres letzten nach dem Zufallsprinzip ausgeführten Kieselsteinwurfs erinnern, könnten Sie Ihren Wurf auf weniger sichere Bereiche ausrichten. Durch Iteration dieses Vorgangs könnten Sie schließlich Ihr Badezimmer finden. In diesem Fall würden Sie sowohl die Zielzerlegung der dynamischen Programmierung als auch die Zufallsprinzipien der Monte-Carlo-Methode anwenden.

1.4 Das Reinforcement-Learning-Framework

Richard Bellman führte die dynamische Programmierung als eine allgemeine Methode zur Lösung bestimmter Arten von Kontroll- oder Entscheidungsproblemen ein, aber sie nimmt einen der äußersten Punkte des RL-Kontinuums ein. Bellmans wichtigerer Beitrag bestand wohl darin, zur Entwicklung des Standard-Frameworks für RL-Probleme beizutragen. Das RL-Framework ist im Wesentlichen der Kernsatz von Begriffen und Konzepten, in dem jedes RL-Problem formuliert werden kann. Damit steht nicht nur eine standardisierte Sprache für die Kommunikation mit anderen Ingenieuren und Forschern zur Verfügung, sondern wir sind auch gezwungen, unsere Probleme so zu formulieren, dass sie sich einer dynamischen Programmierung – ähnlich der Zerlegung von Problemen – unterwerfen lassen, sodass wir über lokale Unterprobleme hinweg iterativ optimieren und Fortschritte bei der Erreichung des globalen Ziels auf hoher Ebene erzielen können. Glücklicherweise ist das auch ziemlich einfach.

Um das Framework konkret zu veranschaulichen, betrachten wir die Aufgabe des Aufbaus eines RL-Algorithmus, der lernen kann, den Energieverbrauch in einem großen Rechenzentrum zu minimieren. Computer müssen kühl gehalten werden, damit sie gut funktionieren, sodass in großen Rechenzentren erhebliche Kosten durch Kühlsysteme entstehen können. Der schlichte Ansatz, ein Rechenzentrum kühl zu halten, bestünde darin, die Klimaanlage ständig auf einem Niveau zu halten, das dazu führt, dass kein Server jemals zu heiß läuft; dies würde kein ausgefallenes Machine Learning erfordern. Aber das ist ineffizient, und Sie könnten es besser machen, da es unwahrscheinlich ist, dass alle Server im Rechenzentrum zur gleichen Zeit heiß laufen und dass die Auslastung des Rechenzentrums immer auf dem gleichen Niveau ist. Wenn Sie die Kühlung dorthin lenken würden, wo und wann es am wichtigsten ist, könnten Sie dasselbe Ergebnis für weniger Geld erzielen.

Der erste Schritt im Framework ist, Ihr Gesamtziel zu definieren. In diesem Fall besteht unser Gesamtziel darin, die für die Kühlung aufgewendeten Mittel zu minimieren, mit der Einschränkung, dass kein Server in unserem Zentrum eine bestimmte Schwellentemperatur überschreiten darf. Obwohl dies zwei Ziele zu sein scheinen, können wir sie zu einer neuen zusammengesetzten *Zielfunktion* bündeln. Diese Funktion gibt einen Ertrag zurück, der angibt, wie weit wir bei der Erreichung der beiden Ziele angesichts der aktuellen Kosten und der Temperaturdaten für die Server vom Ziel abweichen. Die tatsächliche Zahl, die unsere Zielfunktion zurückgibt, ist nicht wichtig; wir wollen sie nur so niedrig wie möglich halten. Daher brauchen wir unseren RL-Algorithmus, um den Ertrag dieser Zielfunktion (Fehler) in Bezug auf einige Eingabedaten zu minimieren, die definitiv die laufenden Kosten und die

Temperaturdaten umfassen, aber auch andere nützliche Kontextinformationen enthalten können, die dem Algorithmus helfen können, die Nutzung des Rechenzentrums vorherzusagen.

Die Eingabedaten werden von der *Umgebung* erzeugt. Im Allgemeinen ist die Umgebung einer RL- (oder Steuerungs-) Aufgabe jeder dynamische Prozess, der Daten erzeugt, die für das Erreichen unseres Ziels relevant sind. Obwohl wir „Umgebung" als technischen Begriff verwenden, ist er nicht allzu weit von seiner alltäglichen Verwendung abstrahiert. Da Sie selbst ein Beispiel für einen sehr fortschrittlichen RL-Algorithmus sind, befinden Sie sich immer in irgendeiner Umgebung, und Ihre Augen und Ohren nehmen ständig Informationen auf, die von Ihrer Umgebung produziert werden, damit Sie Ihre täglichen Ziele erreichen können. Da die Umgebung ein *dynamischer Prozess* (eine Funktion der Zeit) ist, kann sie einen kontinuierlichen Strom von Daten unterschiedlicher Größe und Art produzieren. Um die Dinge algorithmusfreundlich zu gestalten, müssen wir diese Umgebungsdaten in diskrete Pakete bündeln, die wir den *Zustand* (der Umgebung) nennen, und sie dann in jedem ihrer diskreten Zeitschritte an unseren Algorithmus liefern. Der Zustand spiegelt unser Wissen über die Umgebung zu einem bestimmten Zeitpunkt wider, so wie eine Digitalkamera zu einem bestimmten Zeitpunkt einen einzelnen Schnappschuss einer Szene aufnimmt (und ein konsistent formatiertes Bild erzeugt).

Zusammenfassend lässt sich sagen, dass wir bisher eine Zielfunktion (Minimierung der Kosten durch Optimierung der Temperatur) definiert haben, die eine Funktion des Zustands (aktuelle Kosten, aktuelle Temperaturdaten) der Umgebung (des Rechenzentrums und aller damit verbundenen Prozesse) ist. Der letzte Teil unseres Modells ist der RL-Algorithmus selbst. Dies könnte *jeder* parametrische Algorithmus sein, der aus Daten lernen kann, um eine Zielfunktion zu minimieren oder zu maximieren, indem seine Parameter modifiziert werden. Es muss *kein* Deep-Learning-Algorithmus sein; RL ist ein eigenständiges Gebiet, das von den Belangen eines bestimmten Lernalgorithmus getrennt ist.

Wie wir bereits bemerkt haben, besteht einer der Hauptunterschiede zwischen RL (oder Supervised-Learning-Aufgaben im Allgemeinen) und gewöhnlichem Supervised Learning darin, dass der Algorithmus bei einer Kontrollaufgabe Entscheidungen treffen und Aktionen ausführen muss. Diese Aktionen haben eine kausale Auswirkung auf das, was in der Zukunft geschieht. Eine Aktion zu ergreifen ist ein Schlüsselwort im Framework, und es bedeutet mehr oder weniger das, was man erwarten würde. Jede Aktion ist jedoch das Ergebnis der Analyse des aktuellen Zustands der Umgebung und des Versuchs, auf der Grundlage dieser Informationen die beste Entscheidung zu treffen.

Das letzte Konzept im RL-Framework besteht darin, dass der Algorithmus nach jeder Aktion eine *Belohnung erhält*. Die Belohnung ist ein (lokales) Signal dafür, wie gut der Lernalgorithmus bei der Erreichung des globalen Ziels abschneidet. Die Belohnung kann ein positives Signal sein (d. h. es läuft gut, weitermachen) oder ein negatives Signal (d. h. das nicht tun), auch wenn wir beide Situationen als „Belohnung" bezeichnen.

Das Belohnungssignal ist das einzige Signal, an dem sich der Lernalgorithmus orientieren muss, während er sich in der Hoffnung auf eine bessere Leistung im nächsten Zustand der Umgebung aktualisiert. In unserem Rechenzentrumsbeispiel könnten wir dem Algorithmus eine Belohnung von +10 (ein willkürlicher Wert) gewähren, wenn seine Aktion den Fehlerwert reduziert. Oder, was vernünftiger ist, wir könnten ihm eine Belohnung gewähren, die proportional dazu ist, wie sehr er den Fehler verringert. Wenn er den Fehler erhöht, würden wir ihm eine negative Belohnung gewähren.

Schließlich sollten wir unserem Lernalgorithmus einen schickeren Namen geben, indem wir ihn *Agent* nennen. Der Agent ist der handlungs- oder entscheidungsorientierte Lernalgorithmus bei jedem RL-Problem. Wir können das alles zusammensetzen, wie in Bild 1.8 gezeigt.

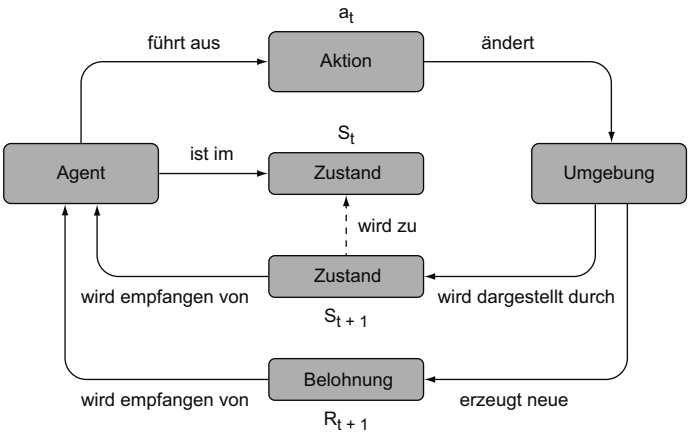

Bild 1.8 Das Standard-Framework für RL-Algorithmen. Der Agent führt eine Aktion in der Umgebung aus, wie z. B. das Ziehen einer Schachfigur, die dann den Zustand der Umgebung aktualisiert. Für jede Aktion, die er ausführt, erhält er eine Belohnung (z. B. +1 für das Gewinnen der Partie, –1 für das Verlieren der Partie, sonst 0). Der RL-Algorithmus wiederholt diesen Prozess mit dem Ziel, die Belohnungen langfristig zu maximieren, und er lernt schließlich, wie die Umgebung funktioniert.

In unserem Beispiel für ein Rechenzentrum hoffen wir, dass unser Agent lernt, wie wir unsere Kühlungskosten senken können. Wenn wir ihn nicht mit vollständigem Wissen über die Umgebung versorgen können, wird er ein gewisses Maß an Trial-and-Error anwenden müssen. Wenn wir Glück haben, lernt der Agent vielleicht so gut, dass er in einer anderen Umgebung als der, in der er ursprünglich trainiert wurde, eingesetzt werden kann. Da der Agent der Lernende ist, wird er als eine Art Lernalgorithmus implementiert. Und da es sich um ein Buch über *Deep* Reinforcement Learning handelt, werden unsere Agenten unter Verwendung von *Deep-Learning-Algorithmen* (auch bekannt als tiefe *neuronale Netze*, siehe Bild 1.9) implementiert. Aber denken Sie daran, dass es bei RL mehr um die Art des Problems und der Lösung geht als um einen bestimmten Lernalgorithmus, und Sie könnten sicherlich Alternativen zu tiefen neuronalen Netzen verwenden. Tatsächlich werden wir in Kapitel 3 damit beginnen, einen sehr einfachen nicht-neuronalen Netz-Algorithmus zu verwenden, und am Ende des Kapitels werden wir ihn durch ein neuronales Netz ersetzen.

Das einzige Ziel des Agenten besteht darin, seine erwartete Belohnung auf lange Sicht zu maximieren. Er wiederholt einfach diesen Zyklus: die Zustandsinformationen verarbeiten, entscheiden, welche Aktion zu ergreifen ist, sehen, ob er eine Belohnung erhält, den neuen Zustand beobachten, eine weitere Aktion durchführen und so weiter. Wenn wir all dies richtig einrichten, wird der Agent schließlich lernen, seine Umgebung zu verstehen und bei jedem Schritt verlässlich gute Entscheidungen zu treffen. Dieser allgemeine Mechanismus lässt sich auf autonome Fahrzeuge, Chatbots, Robotik, automatisierten Aktienhandel, Gesundheitswesen und vieles mehr anwenden. Einige dieser Anwendungen werden wir im nächsten Abschnitt und an weiteren Stellen im Buch erkunden.

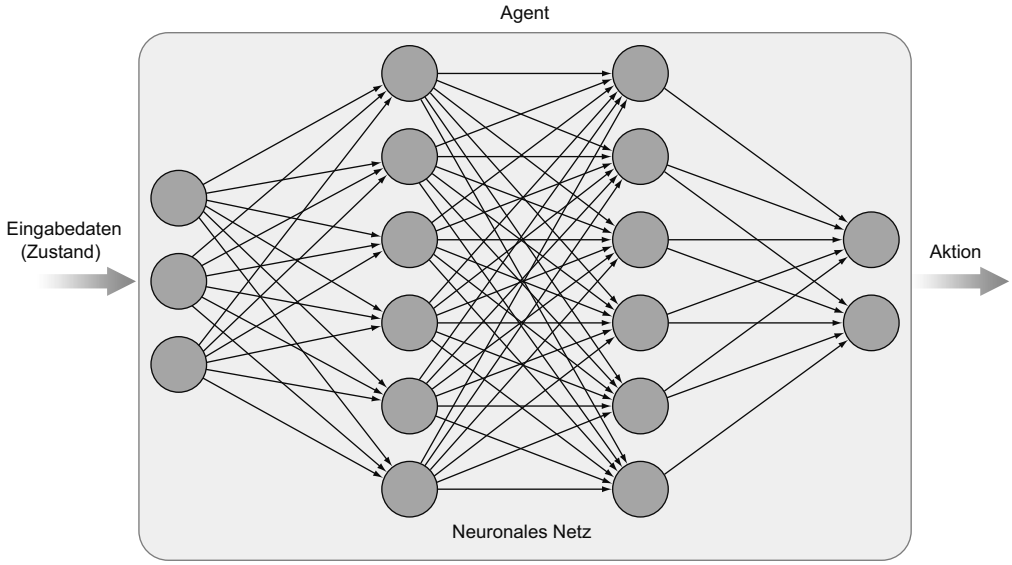

Bild 1.9 Die Eingabedaten (d. h. der Zustand der Umgebung zu einem bestimmten Zeitpunkt) werden in den Agenten (in diesem Buch als tiefes neuronales Netz implementiert) eingegeben, der diese Daten dann auswertet, um eine Aktion durchzuführen. Der Prozess ist etwas aufwendiger als hier gezeigt, aber die Illustration trifft den Kern.

Die meiste Zeit in diesem Buch werden Sie damit verbringen, zu lernen, wie man Probleme in unserem Standardmodell strukturiert und wie man ausreichend leistungsfähige Lernalgorithmen (Agenten) implementiert, um schwierige Probleme zu lösen. Für diese Beispiele müssen Sie keine Umgebungen konstruieren – Sie müssen sich in bestehende Umgebungen (wie Spiele-Engines oder andere APIs) einklinken. OpenAI hat zum Beispiel eine Python Gym-Bibliothek veröffentlicht, die uns eine Reihe von Umgebungen und eine unkomplizierte Schnittstelle für unseren Lernalgorithmus zur Verfügung stellt, mit dem wir interagieren können. Der Code auf der linken Seite von Bild 1.10 zeigt, wie einfach es ist, eine dieser Umgebungen einzurichten und zu verwenden – ein Autorennspiel benötigt nur fünf Zeilen Code.

```
import gym
env = gym.make('CarRacing-v0')
env.reset()
env.step(action)
env.render()
```

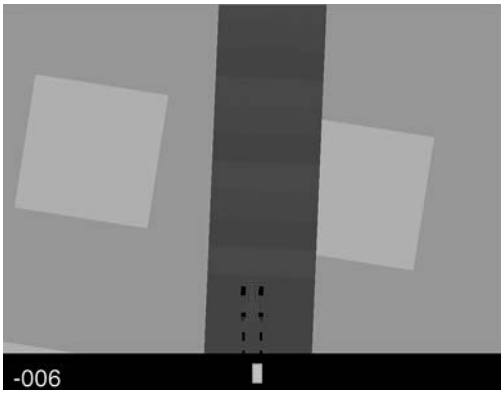

Bild 1.10 Die OpenAI-Python-Bibliothek wird mit vielen Umgebungen und einer einfach zu bedienenden Schnittstelle für einen Lernalgorithmus geliefert, mit dem man interagieren kann. Mit nur wenigen Zeilen Code haben wir ein Autorennspiel hochgeladen.

1.5 Was kann ich mit Reinforcement Learning anfangen?

Wir begannen dieses Kapitel mit einem Überblick über die Grundlagen gewöhnlicher Algorithmen des Supervised Machine Learning, wie z. B. Bildklassifikatoren, und obwohl die jüngsten Erfolge beim Supervised Machine Learning wichtig und nützlich sind, wird uns Supervised Learning nicht zur Artificial General Intelligence (generelle künstliche Intelligenz – AGI) führen. Letztendlich suchen wir nach Allzweck-Lernmaschinen, die mit minimaler bis gar keiner Beaufsichtigung auf mehrere Probleme angewendet werden können und deren Repertoire an Fähigkeiten domänenübergreifend übertragen werden kann. Große Firmen mit vielen Daten können von Ansätzen des Supervised Learning gewinnbringend profitieren, aber kleinere Firmen und Organisationen haben möglicherweise nicht die Ressourcen, um die Möglichkeiten des Machine Learning auszunutzen. Allzweck-Lernalgorithmen würden das Spielfeld für alle ebnen, und Reinforcement Learning ist derzeit der vielversprechendste Ansatz für solche Algorithmen.

Die RL-Forschung und -Anwendungen sind noch nicht ausgereift, aber es gab in den letzten Jahren viele aufregende Entwicklungen. Die DeepMind-Forschungsgruppe von Google hat einige beeindruckende Ergebnisse vorgelegt und internationale Aufmerksamkeit erregt. Das erste Mal gelang dies 2013 mit einem Algorithmus, der ein Spektrum von Atari-Spielen auf übermenschlichen Leveln spielen konnte. Frühere Versuche, Agenten zur Lösung dieser Spiele zu schaffen, beinhalteten die Feinabstimmung der zugrunde liegenden Algorithmen, um die spezifischen Regeln des Spiels zu verstehen, was oft als *Feature Engineering (Merkmal-Engineering)* bezeichnet wird. Diese Feature-Engineering-Ansätze können für ein bestimmtes Spiel gut funktionieren, aber sie sind nicht dazu geeignet, Wissen oder Fähigkeiten auf ein

neues Spiel oder eine neue Domäne zu übertragen. Der Deep-Q-Network-Algorithmus (DQN) von DeepMind war robust genug, um bei sieben Spielen ohne spielspezifische Anpassungen zu funktionieren (siehe Bild 1.11). Er hatte nicht mehr als die Rohpixel vom Bildschirm als Eingabe und wurde lediglich angewiesen, die Punktzahl zu maximieren, dennoch lernte der Algorithmus über ein menschliches Expertenniveau hinaus zu spielen.

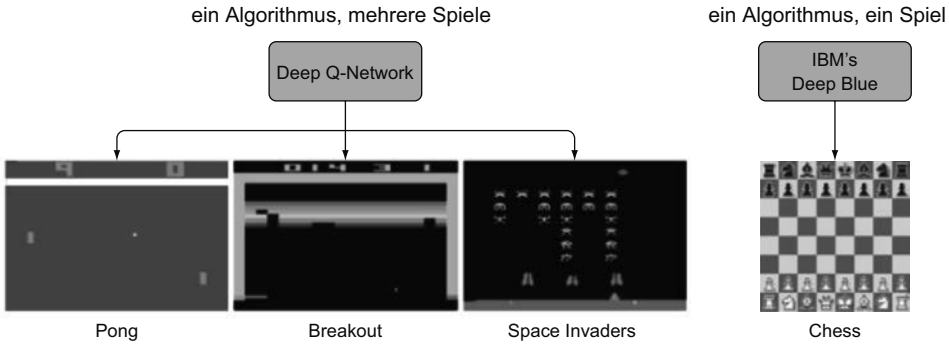

Bild 1.11 Der DQN-Algorithmus von DeepMind lernte erfolgreich, sieben Atari-Spiele zu spielen, wobei er nur die Rohpixel als Eingabe und die Maximierung der Punktzahl als Ziel hatte. Frühere Algorithmen, wie z. B. IBMs Deep-Blue, mussten für ein bestimmtes Spiel feinabgestimmt werden.

In jüngerer Zeit schlugen die DeepMind-Algorithmen AlphaGo und AlphaZero die weltbesten Spieler beim alten chinesischen Spiel Go. Experten glaubten, dass künstliche Intelligenz mindestens ein weiteres Jahrzehnt lang nicht in der Lage sein würde, Go konkurrenzfähig zu spielen, weil das Spiel Eigenschaften aufweist, die Algorithmen normalerweise nicht gut handhaben können. Die Spieler wissen nicht, welcher Zug am besten ist, und erhalten erst am Ende der Partie Feedback auf ihre Aktionen. Viele professionelle Spieler sahen sich eher als Künstler denn als berechnende Strategen und beschrieben Gewinnzüge als schön oder elegant. Bei über 10^{170} legalen Brettstellungen waren Brute-Force-Algorithmen (mit denen IBMs Deep-Blue-Algorithmus beim Schach gewann) nicht durchführbar. AlphaGo gelang dieses Kunststück vor allem dadurch, dass es Millionen von simulierten Go-Spielen spielte und lernte, welche Aktionen die Belohnungen für ein gutes Spiel maximierten. Ähnlich wie im Atari-Fall hatte AlphaGo nur Zugang zu den Informationen, die auch ein menschlicher Spieler hat: wo sich die Figuren auf dem Brett befanden.

Während Algorithmen, die Spiele besser als Menschen spielen können, bemerkenswert sind, gehen das Versprechen und das Potenzial von RL weit über die Herstellung besserer Spiel-Bots hinaus. DeepMind konnte ein Modell zur Senkung der Kühlungskosten von Googles Rechenzentrum um 40 % erstellen, eine Anwendung, die wir bereits zu einem früheren Zeitpunkt in diesem Kapitel als Beispiel untersucht haben. Autonome Fahrzeuge nutzen RL, um zu lernen, welche Reihe von Aktionen (Beschleunigen, Abbiegen, Bremsen, Signalgebung) dazu führt, dass die Fahrgäste ihr Ziel rechtzeitig erreichen, und um zu lernen, wie Unfälle vermieden werden können. Und Forscher trainieren Roboter so, dass sie Aufgaben wie das Laufenlernen erledigen, ohne explizit komplexe motorische Fähigkeiten zu programmieren.

Viele dieser Beispiele sind große Herausforderungen, wie z. B. das Fahren eines Autos. Man kann eine Lernmaschine nicht einfach durch Trial-and-Error lernen lassen, wie man

ein Auto fährt. Glücklicherweise gibt es immer mehr erfolgreiche Beispiele dafür, dass man Lernmaschinen in harmlosen Simulatoren laufen lässt und sie, sobald sie den Simulator beherrschen, reale Hardware in der realen Welt ausprobieren lässt. Ein Beispiel, das wir in diesem Buch erkunden werden, ist der automatisierte Handel (Algorithmic Trading). Ein beträchtlicher Teil des gesamten Aktienhandels wird von Computern mit wenig bis gar keinem Input durch menschliche Operatoren ausgeführt. Die meisten dieser algorithmischen Händler werden von riesigen Hedgefonds verwaltet, die Milliarden von Dollar verwalten. In den letzten Jahren haben wir jedoch ein zunehmendes Interesse einzelner Händler an der Entwicklung von Handelsalgorithmen beobachten können. So bietet etwa Quantopian eine Plattform, auf der einzelne Benutzer Handelsalgorithmen in Python schreiben und sie in einer sicheren, simulierten Umgebung testen können. Wenn die Algorithmen gut funktionieren, können sie für den Handel mit echtem Geld verwendet werden. Viele Händler haben mit einfacher Heuristik und regelbasierten Algorithmen relativen Erfolg erzielt. Da Aktienmärkte jedoch dynamisch und unvorhersehbar sind, hat ein kontinuierlich lernender RL-Algorithmus den Vorteil, dass er sich in Echtzeit an sich ändernde Marktbedingungen anpassen kann.

Ein praktisches Problem, das wir in diesem Buch behandeln werden, ist die Platzierung von Werbung. Aus Anzeigen erzielen viele Web-Unternehmen beträchtliche Einnahmen, die oft an die Anzahl der Klicks gebunden sind, die diese Anzeigen erzielen können. Es besteht ein großer Anreiz, Anzeigen dort zu platzieren, wo sie die Klicks maximieren können. Die einzige Möglichkeit, dies zu erreichen, besteht jedoch darin, das Wissen über User zu nutzen, um die am besten geeigneten Anzeigen zu schalten. Im Allgemeinen wissen wir nicht, welche Eigenschaften des Users mit der richtigen Anzeigenauswahl zusammenhängen, aber wir können RL-Techniken einsetzen, um einige Fortschritte zu erzielen. Wenn wir einem RL-Algorithmus einige potenziell nützliche Informationen über den User geben (was wir die Umgebung oder den Zustand der Umgebung nennen würden) und ihm sagen, er solle die Anzeigenklicks maximieren, lernt er, wie er seine Eingabedaten mit seinem Ziel in Verbindung bringen kann, und er lernt schließlich, welche Anzeigen die meisten Klicks von einem bestimmten User erzeugen.

1.6 Warum Deep Reinforcement Learning?

Wir haben uns für Reinforcement Learning ausgesprochen, aber warum *Deep* Reinforcement Learning? RL gab es schon lange vor dem populären Aufstieg des Deep Learning. Tatsächlich beinhalteten einige der frühesten Methoden (die wir zu Lernzwecken betrachten werden) nichts anderes als die Speicherung von Erfahrungen in einer Umsetzungstabelle (z. B. einem Python-Wörterbuch) und die Aktualisierung dieser Tabelle bei jeder Iteration des Lernalgorithmus. Die Idee war, den Agenten in der Umgebung herumspielen zu lassen, um zu sehen, was passiert ist, und seine Erfahrungen mit dem Geschehen in einer Art Datenbank zu speichern. Nach einer Weile konnte man auf diese Wissensdatenbank zurückblicken und beobachten, was funktionierte und was nicht – ganz ohne neuronalen Netze oder andere ausgefallene Algorithmen.

Für sehr einfache Umgebungen funktioniert das eigentlich recht gut. Zum Beispiel gibt es in Tic-Tac-Toe 255.168 gültige Brettpositionen. Die *Umsetzungstabelle* (auch *Speichertabelle* genannt) würde so viele Einträge haben, die von jedem Zustand auf eine bestimmte Aktion (wie in Bild 1.12 gezeigt) und die beobachtete Belohnung (nicht abgebildet) übertragen werden. Während des Trainings könnte der Algorithmus lernen, welche Bewegung zu günstigeren Positionen führte, und diesen Eintrag in der Speichertabelle aktualisieren.

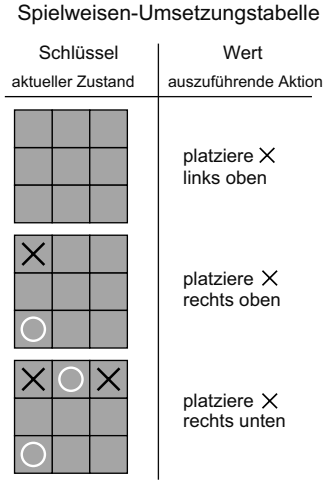

Bild 1.12 Eine Umsetzungstabelle für Tic-Tac-Toe mit nur drei Einträgen, wobei der „Spieler" (ein Algorithmus) X spielt. Wenn der Spieler eine Brettstellung mitgeteilt bekommt, diktiert die Umsetzungstabelle den Zug, den er als Nächstes machen soll. Es gibt einen Eintrag für jeden möglichen Zustand in der Partie.

Sobald die Umgebung komplizierter wird, wird die Verwendung einer Speichertabelle schwierig. Zum Beispiel könnte jede Bildschirmkonfiguration eines Videospiels als ein anderer Zustand betrachtet werden (Bild 1.13). Stellen Sie sich vor, Sie versuchen, jede mögliche Kombination von gültigen Pixelwerten, die in einem Videospiel auf dem Bildschirm angezeigt werden, zu speichern! Der DQN-Algorithmus von DeepMind, der Atari spielte, wurde bei jedem

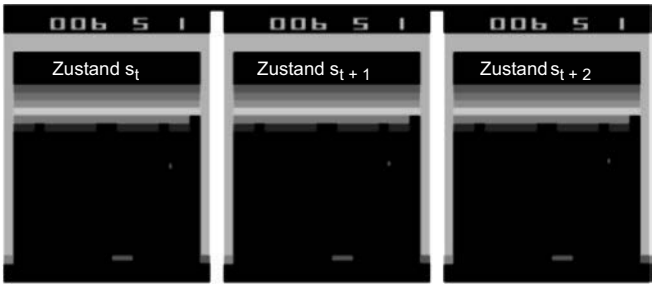

Bild 1.13 Eine Folge von drei Bildern von Breakout. Die Platzierung des Balls ist in jedem Rahmen leicht unterschiedlich. Wenn Sie eine Umsetzungstabelle verwenden würden, würde dies der Speicherung von drei eindeutigen Einträgen in der Tabelle entsprechen. Eine Umsetzungstabelle wäre unbrauchbar, da es viel zu viele Spielzustände zu speichern gibt.

Schritt mit vier 84 × 84 Graustufenbildern gefüttert, was zu 256^{28228} einzigartigen Spielzuständen führen würde (256 verschiedene Graustufen pro Pixel und 4 ∗ 84 ∗ 84 = 28228 Pixel). Diese Zahl ist viel größer als die Anzahl der Atome im beobachtbaren Universum und würde definitiv nicht in den Computerspeicher passen. Und das, nachdem die Bilder verkleinert wurden, um die Größe der ursprünglich 210 × 160 Pixel großen Farbbildern zu reduzieren.

Es ist nicht möglich, jeden möglichen Zustand zu speichern, aber wir könnten versuchen, die Möglichkeiten einzuschränken. Im Spiel Breakout steuern Sie ein Paddel am unteren Bildschirmrand, das sich nach rechts oder links bewegen kann; Ziel des Spiels ist es, den Ball abzulenken und so viele Blöcke am oberen Bildschirmrand zu zerbrechen wie möglich. In diesem Fall könnten wir Einschränkungen definieren, wobei wir uns nur die Zustände ansehen, in denen der Ball zum Schläger zurückkehrt, da unsere Aktionen nicht wichtig sind, während wir auf den Ball am oberen Bildschirmrand warten. Oder wir könnten unsere eigenen Features zur Verfügung stellen – anstatt das Rohbild zu liefern, geben wir einfach die Position des Balles, des Schlägers und der restlichen Blöcke an. Diese Methoden setzen jedoch voraus, dass der Programmierer die dem Spiel zugrunde liegenden Strategien versteht, und sie ließen sich nicht auf andere Umgebungen verallgemeinern.

An dieser Stelle kommt Deep Learning ins Spiel. Ein Deep-Learning-Algorithmus kann lernen, die Details spezifischer Pixelanordnungen zu abstrahieren, und er kann die wichtigen Features eines Zustands lernen. Da ein Deep-Learning-Algorithmus eine endliche Anzahl von Parametern hat, können wir ihn dazu verwenden, jeden möglichen Zustand in etwas zu komprimieren, das wir effizient verarbeiten können, und dann diese neue Darstellung verwenden, um unsere Entscheidungen zu treffen. Durch die Verwendung neuronaler Netze hatte der Atari DQN nur 1792 Parameter (Convolutional Neural Network (CNN, faltendes neuronales Netz) mit 16 8 × 8 Filtern, 32 4 × 4 Filtern und einer 256 Knoten umfassenden, vollständig verbundenen verborgenen Schicht) im Gegensatz zu den 256^{28228} Schlüssel-Wert-Paaren, die zur Speicherung des gesamten Zustandsraums erforderlich wären.

Im Fall des Breakout-Spiels könnte ein tiefes neuronales Netz von sich aus lernen, dieselben High-Level-Features zu erkennen, die ein Programmierer in einer Umsetzungstabelle von Hand entwickeln müsste. Das heißt, es könnte lernen, den Ball, das Paddel, die Blöcke zu „sehen“ und die Richtung des Balles zu erkennen. Das ist ziemlich erstaunlich, wenn man bedenkt, dass er nur Rohdaten in Form von Pixeln erhält. Und noch interessanter ist, dass die gelernten High-Level-Features möglicherweise auf andere Spiele oder Umgebungen übertragbar sind.

Deep Learning ist die geheime Zutat, die all die jüngsten Erfolge in RL möglich macht. Keine andere Klasse von Algorithmen hat eine vergleichbare Darstellungskraft, Effizienz und Flexibilität von tiefen neuronalen Netzen gezeigt. Außerdem sind neuronale Netze eigentlich ziemlich einfach!

1.7 Unser didaktisches Werkzeug: String-Diagramme

Die grundlegenden Konzepte des RL sind seit Jahrzehnten gut etabliert, aber der Bereich entwickelt sich sehr schnell, so dass jedes einzelne neue Ergebnis bald veraltet sein könnte. Deshalb konzentriert sich dieses Buch auf die Vermittlung von Fähigkeiten, nicht auf Details mit kurzen Halbwertszeiten. Wir behandeln zwar einige neuere Fortschritte auf diesem Gebiet, die sicherlich in nicht allzu ferner Zukunft verdrängt werden, aber wir tun dies nur, um neue Fertigkeiten zu vermitteln, und nicht, weil das spezielle Thema, das wir behandeln, notwendigerweise eine bewährte Technik ist. Wir sind zuversichtlich, dass die Fähigkeiten, die Sie lernen, nicht veraltet sein werden, auch wenn einige unserer Beispiele veraltet sein sollten, und dass Sie darauf vorbereitet sein werden, RL-Probleme noch lange Zeit in Angriff zu nehmen.

Außerdem ist RL ein weites Feld, auf dem es viel zu lernen gibt. Wir können unmöglich hoffen, dass wir in diesem Buch alles davon abdecken können. Unser Ziel ist es nicht, eine erschöpfende RL-Referenz oder ein umfassender Kurs zu sein, sondern Ihnen die Grundlagen von RL zu vermitteln und einige der aufregendsten jüngsten Entwicklungen auf diesem Gebiet zu präsentieren. Wir erwarten, dass Sie in der Lage sein werden, das, was Sie hier gelernt haben, zu übernehmen, um sich in den vielen anderen Bereichen von RL leicht zurechtzufinden. Darüber hinaus haben wir einen Abschnitt in Kapitel 11, der Ihnen einen Überblick über Bereiche gibt, die Sie sich nach der Lektüre dieses Buches vielleicht ansehen möchten.

Dieses Buch ist darauf ausgerichtet, gut, aber auch rigoros zu unterrichten. Reinforcement Learning und Deep Learning sind grundlegend mathematisch. Wenn Sie irgendwelche Forschungsartikel in diesen Bereichen lesen, werden Sie möglicherweise auf ungewohnte mathematische Notationen und Gleichungen stoßen. Die Mathematik ermöglicht es uns, präzise Aussagen darüber zu machen, was wahr ist und wie die Dinge zusammenhängen, und sie bietet rigorose Erklärungen dafür, wie und warum die Dinge funktionieren. Wir könnten RL ohne Mathematik unterrichten und einfach Python verwenden, aber dieser Ansatz würde dazu führen, dass Sie zukünftige Entwicklungen nicht verstehen können.

Wir denken also, dass die Mathematik wichtig ist, aber wie unser Lektor bemerkte, gibt es in der Verlagswelt ein gängiges Sprichwort, das wahrscheinlich einen wahren Kern hat: „Für jede Gleichung im Buch wird die Leserschaft halbiert“. Es gibt einen unvermeidlichen kognitiven Overhead beim Entziffern komplexer mathematischer Gleichungen, es sei denn, man ist ein professioneller Mathematiker, der den ganzen Tag lang Gleichungen liest und schreibt. Angesichts des Wunsches, eine rigorose Darstellung von DRL zu präsentieren, um den Lesern ein erstklassiges Verständnis zu vermitteln, und dennoch so viele Menschen wie möglich zu erreichen, haben wir uns ein unserer Meinung nach sehr charakteristisches Merkmal für dieses Buch ausgedacht. Wie sich herausstellt, sind selbst professionelle Mathematiker der traditionellen mathematischen Notation mit ihrem riesigen Spektrum von Symbolen überdrüssig geworden, und in einem bestimmten Zweig der fortgeschrittenen Mathematik, der *Kategorientheorie*, haben Mathematiker eine rein grafische Sprache entwickelt, die *String-Diagramme* genannt wird. String-Diagramme sehen Flussdiagrammen und Schaltplänen sehr ähnlich, und ihre Bedeutung ist weitgehend intuitiv zu erfassen, wobei sie genauso streng und präzise sind wie traditionelle mathematische Notationen, die weitgehend auf griechischen und lateinischen Symbolen basieren.

Bild 1.14 zeigt ein einfaches Beispiel für ein String-Diagramm, das auf komplexe Art ein neuronales Netz mit zwei Schichten darstellt. Machine Learning (insbesondere Deep Learning) umfasst viele Matrix- und Vektoroperationen, und String-Diagramme eignen sich besonders gut, um diese Art von Operationen grafisch zu beschreiben. String-Diagramme eignen sich auch hervorragend zur Beschreibung komplexer Prozesse, da wir den Prozess auf verschiedenen Abstraktionsebenen beschreiben können. Die obere Tafel von Bild 1.14 zeigt zwei Rechtecke, die die beiden Schichten des neuronalen Netzes darstellen, aber dann können wir in die Schicht 1 „hineinzoomen“ (in den Kasten hineinschauen), um zu sehen, was sie im Detail tut; dies wird in der unteren Tafel von Bild 1.14 gezeigt.

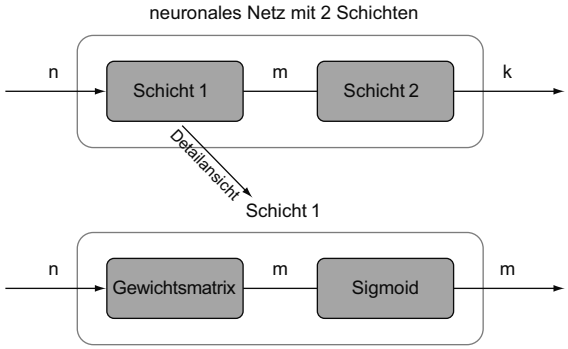

Bild 1.14 Ein String-Diagramm für ein zweischichtiges neuronales Netz. Von links nach rechts gelesen, stellt das obere String-Diagramm ein neuronales Netz dar, das einen Eingabevektor der Dimension n annimmt, ihn mit einer Matrix der Dimensionen $n \times m$ multipliziert und einen Ertrag der Dimension m zurückgibt. Dann wird die nichtlineare Sigmoid-Aktivierungsfunktion auf jedes Element im *m-dimensionalen* Vektor angewendet. Dieser neue Vektor wird dann durch die gleiche Schrittfolge in Schicht 2 geführt, was die endgültige Ausgabe des neuronalen Netzes erzeugt, einen *k-dimensionalen* Vektor.

Wir werden im gesamten Buch häufig String-Diagramme zur Vermittlung von komplexen mathematischen Gleichungen bis hin zu den Architekturen tiefer neuronaler Netze verwenden. Wir werden diese grafische Syntax im nächsten Kapitel beschreiben, und wir werden sie im weiteren Verlauf des Buches weiter verfeinern und aufbauen. In einigen Fällen ist diese grafische Notation für das, was wir zu erklären versuchen, übertrieben, sodass wir eine Kombination aus klarer Prosa und Python oder Pseudocode verwenden werden. In den meisten Fällen werden wir auch die traditionelle mathematische Notation verwenden, sodass Sie die zugrunde liegenden mathematischen Konzepte auf die eine oder andere Weise lernen können, je nachdem, was Ihnen am besten zusagt: Diagramme, Code oder normale mathematische Notation.

1.8 Wie geht es weiter?

Im nächsten Kapitel tauchen wir direkt in das eigentliche Wesen von RL ein und behandeln viele der Kernbegriffe, wie z. B. den Kompromiss zwischen Erkundung und Ausnutzung, Markov Decision Processes, Wert-Funktionen und Policies (diese Begriffe werden Sie bald verstehen). Doch zunächst werden wir zu Beginn des nächsten Kapitels einige der Lehrmethoden vorstellen, die wir in diesem Buch anwenden werden.

Der Rest des Buches befasst sich mit zentralen DRL-Algorithmen, auf denen ein Großteil der neuesten Forschung aufbaut, beginnend mit Deep Q-Networks, gefolgt von Policy-Gradient-Ansätzen und modellbasierten Algorithmen. Wir werden in erster Linie (das bereits erwähnte) OpenAI's Gym nutzen, um unsere Algorithmen zu trainieren, um nichtlineare Dynamiken zu verstehen, Roboter zu steuern und Spiele zu spielen (Bild 1.15).

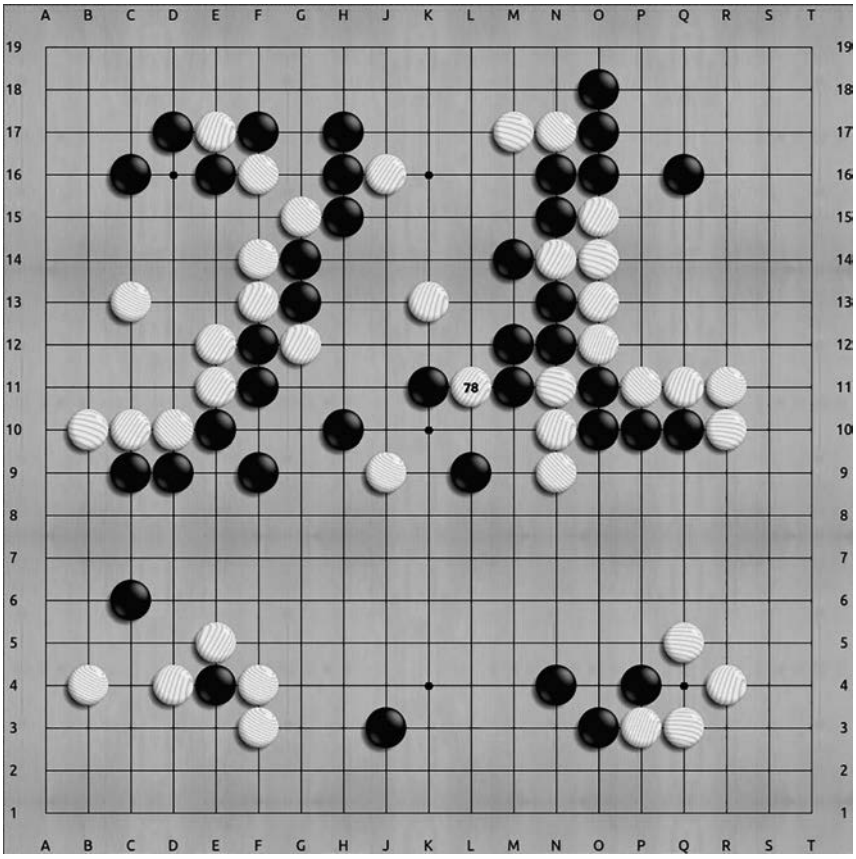

Bild 1.15 Darstellung eines Go-Bretts, das Google DeepMind als Testumfeld für seinen AlphaGo-Algorithmus für Reinforcement Learning verwendete. Der professionelle Go-Spieler Lee Sedol gewann nur eine von fünf Partien. Damit wurde ein Wendepunkt für Reinforcement Learning markiert, da Go lange Zeit als unempfindlich gegenüber der Art von algorithmischer Argumentation galt, der das Schachspiel unterliegt. Quelle: *http://mng.bz/DNX0*

In jedem Kapitel beginnen wir mit einem Hauptproblem oder -projekt, das wir verwenden werden, um die wichtigen Konzepte und Skills für dieses Kapitel zu veranschaulichen. Im weiteren Verlauf jedes Kapitels können wir dem Ausgangsproblem Komplexität oder Nuancen hinzufügen, um einige der Prinzipien zu vertiefen. Zum Beispiel beginnen wir in Kapitel 2 mit dem Problem der Maximierung der Belohnungen an einer Casino-Spielautomatik, und indem wir dieses Problem lösen, werden wir die meisten Grundlagen von RL behandeln. Später fügen wir diesem Problem eine gewisse Komplexität hinzu und ändern die Einstellung von einem Kasino zu einem Unternehmen, das Werbeklicks maximieren muss, was es uns ermöglicht, einige weitere Kernkonzepte darzustellen.

Obwohl dieses Buch für Leser gedacht ist, die bereits Erfahrung mit den Grundlagen des Deep Learning haben, gehen wir davon aus, dass es Ihnen nicht nur unterhaltsame und nützliche RL-Techniken vermittelt, sondern auch Ihre Fähigkeiten zum Deep Learning verfeinert. Um einige der anspruchsvolleren Projekte zu lösen, werden wir die neuesten Entwicklungen im Bereich des Deep Learning einsetzen müssen, wie z. B. Generative Adversarial Networks (GANs, generative gegnerische Netze), evolutionäre Methoden, Meta-Lernen und Transferlernen. Auch dies alles steht im Einklang mit unserer auf Fähigkeiten fokussierten Lehrmethode, sodass die Einzelheiten dieser Fortschritte nicht das Wichtigste sind.

1.9 Zusammenfassung

- Reinforcement Learning ist eine Unterklasse des Machine Learning. RL-Algorithmen lernen durch die Maximierung der Belohnung in einer bestimmten Umgebung, und sie sind nützlich, wenn es bei einem Problem darum geht, Entscheidungen zu treffen oder Aktionen durchzuführen. RL-Algorithmen können im Prinzip jedes statistische Lernmodell verwenden, aber es ist effektiver und zunehmend populärer, tiefe neuronale Netze zu verwenden.
- Der Agent steht im Mittelpunkt eines jeden RL-Problems. Er ist der Teil des RL-Algorithmus, der Eingabe verarbeitet, um zu bestimmen, welche Aktion zu ergreifen ist. In diesem Buch konzentrieren wir uns in erster Linie auf Agenten, die als tiefe neuronale Netze implementiert sind.
- Die Umgebung sind die potenziell dynamischen Bedingungen, unter denen der Agent arbeitet. Allgemeiner gesagt, ist die Umgebung der Prozess, der die Eingabedaten für den Agenten erzeugt. Zum Beispiel könnten wir einen Agenten haben, der ein Flugzeug in einem Flugsimulator fliegt, somit wäre der Simulator die Umgebung.
- Der Zustand ist eine Momentaufnahme der Umgebung, zu der der Agent Zugang hat und die er nutzt, um Entscheidungen zu treffen. Die Umgebung besteht oft aus einer Reihe von sich ständig ändernden Bedingungen, aber wir können Proben aus der Umgebung nehmen, und diese Proben zu bestimmten Zeiten sind die Zustandsinformationen der Umgebung, die wir dem Agenten geben.
- Eine Aktion ist eine von einem Agenten getroffene Entscheidung, die eine Veränderung in seiner Umgebung bewirkt. Das Bewegen einer bestimmten Schachfigur ist eine Aktion, ebenso wie das Drücken des Gaspedals in einem Auto.

- Eine Belohnung ist ein positives oder negatives Signal, das einem Agenten von der Umgebung gegeben wird, nachdem er eine Aktion ausgeführt hat. Die Belohnungen sind die einzigen Lernsignale, die dem Agenten gegeben werden. Das Ziel eines RL-Algorithmus (d. h. des Agenten) ist es, die Belohnungen zu maximieren.
- Die allgemeine Pipeline für einen RL-Algorithmus ist eine Schleife, in der der Agent Eingabedaten (den Zustand der Umgebung) empfängt, der Agent diese Daten auswertet und eine Aktion aus einem Satz möglicher Aktionen unter Berücksichtigung seines aktuellen Zustands ausführt, die Aktion die Umgebung verändert und die Umgebung dann ein Belohnungssignal und neue Zustandsinformationen an den Agenten sendet. Dann wiederholt sich der Zyklus. Wenn der Agent als tiefes neuronales Netz implementiert ist, bewertet jede Iteration eine Verlustfunktion auf der Grundlage des Belohnungssignals und führt eine Backpropagation (Rückführung) durch, um die Leistung des Agenten zu verbessern.