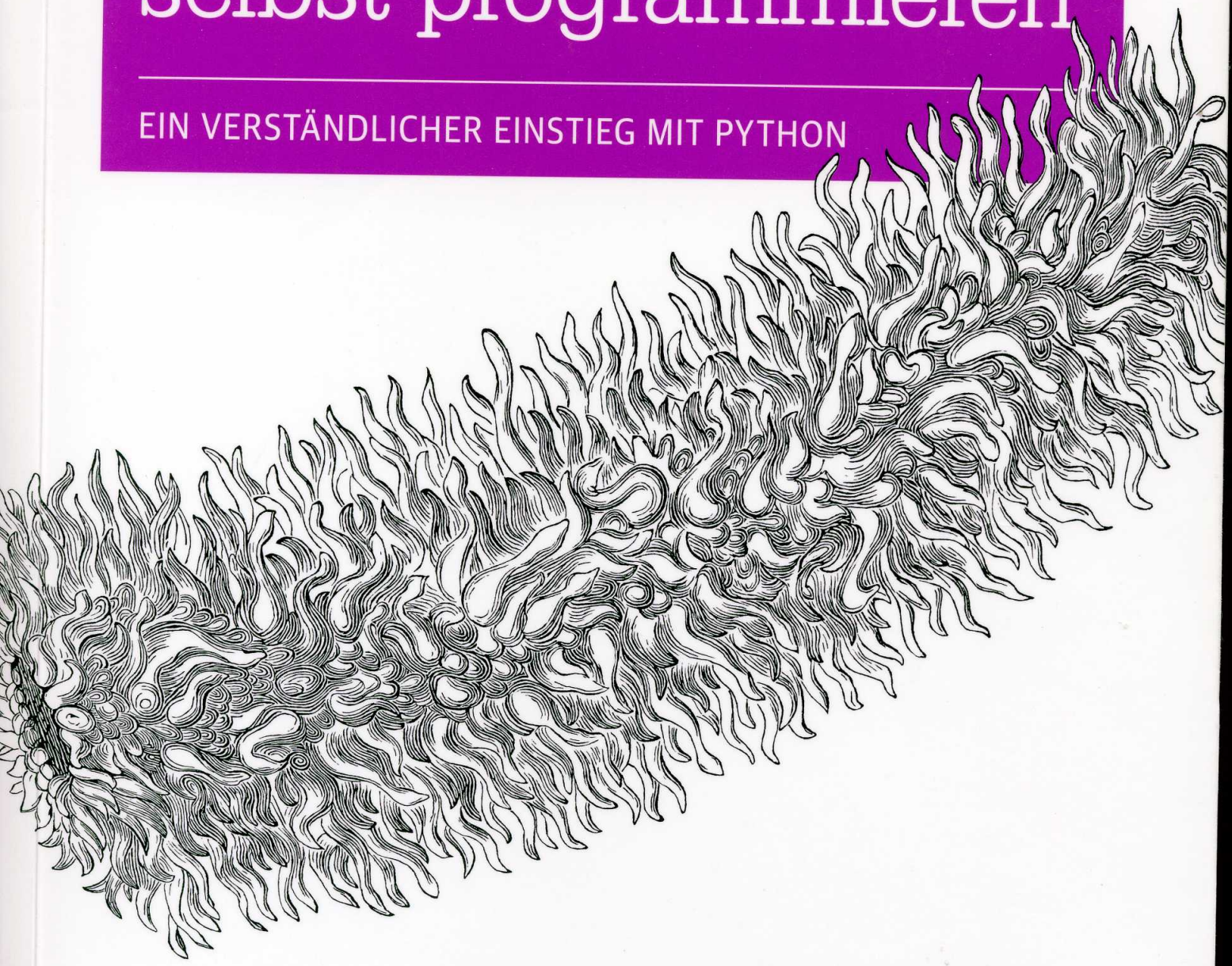
O'REILLY®

Neuronale Netze selbst programmieren

EIN VERSTÄNDLICHER EINSTIEG MIT PYTHON



Tariq Rashid

Übersetzung von Frank Langenau

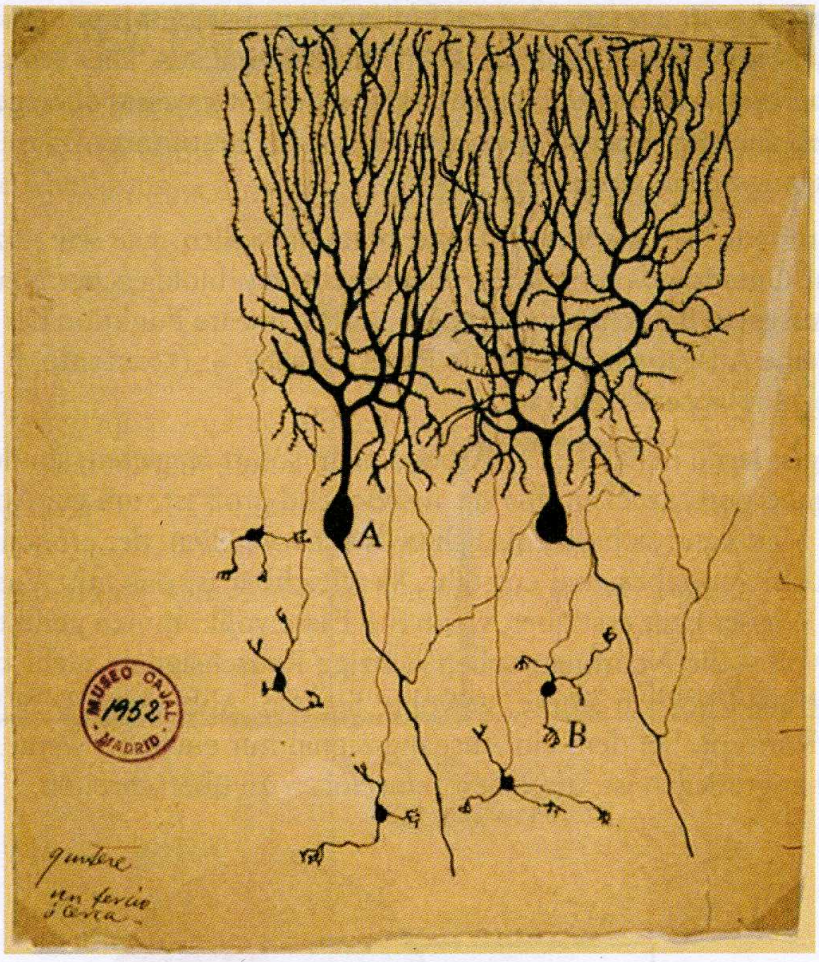


Abbildung 1-28: Skizze von Neuronen in einem Taubengehirn

Wie viele Neuronen brauchen wir, um interessantere, komplexere Aufgaben zu realisieren?

Das sehr leistungsfähige menschliche Gehirn enthält ungefähr 100 Milliarden Neuronen! Eine Fruchtfliege besitzt lediglich 100.000 Neuronen und ist damit schon in der Lage, zu fliegen, zu fressen, Gefahren auszuweichen, Nahrung zu suchen und viele andere ziemlich komplexe Aufgaben zu erledigen. Da die Anzahl von 100.000 Neuronen im Kapazitätsbereich moderner Computer liegt, könnte man doch versuchen, ein solches Gehirn nachzubilden. Ein Fadenwurm hat nur 302 Neuronen, was geradezu verschwindend gering ist, verglichen mit den Ressourcen heutiger Digitalrechner! Doch dieser Wurm kann einige recht nützliche Aufgaben bewältigen, mit denen herkömmliche Computerprogramme von viel größerem Umfang nicht zurechtkämen.

Worin liegt also das Geheimnis? Warum sind biologische Gehirne so leistungsfähig, wenn man bedenkt, dass sie – verglichen mit modernen Computern – wesentlich langsamer sind und aus relativ wenigen Verarbeitungselementen bestehen? Die vollständige Funktionsweise von Gehirnen, wie zum Beispiel des Bewusstseins, ist immer noch ein Geheimnis. Doch weiß man inzwischen genügend über Neuronen, um auf verschiedene Arten der Verarbeitung schließen zu können.

Neuronen – die Rechenmaschinen der Natur

Wir haben weiter oben bereits angesprochen, dass die Gehirne von Tieren die Wissenschaftler vor Rätsel stellen, weil selbst kleine Exemplare wie das Gehirn einer Taube weitaus leistungsfähiger sind als Digitalcomputer mit einer riesigen Anzahl von elektronischen Verarbeitungselementen, einem riesigen Speicherplatz und Ausführungsgeschwindigkeiten, die wesentlich höher liegen als die fleischigen, schwammigen natürlichen Gehirne.

Man hat also seine Aufmerksamkeit auf die architektonischen Unterschiede gerichtet. Herkömmliche Computer verarbeiten die Daten vor allem sequenziell und nach ziemlich konkreten Vorschriften. Bei ihren kalten, harten Berechnungen gibt es weder Unschärfe (engl. Fuzziness) noch Mehrdeutigkeit. Dagegen verarbeiten tierische Gehirne, obwohl sie offensichtlich mit wesentlich langsameren Taktgeschwindigkeiten laufen, die Signale parallel, und Fuzziness ist ein entscheidendes Merkmal ihrer Verarbeitung.

Sehen wir uns in Abbildung 1-27 die Grundeinheit eines biologischen Gehirns an – das *Neuron*.

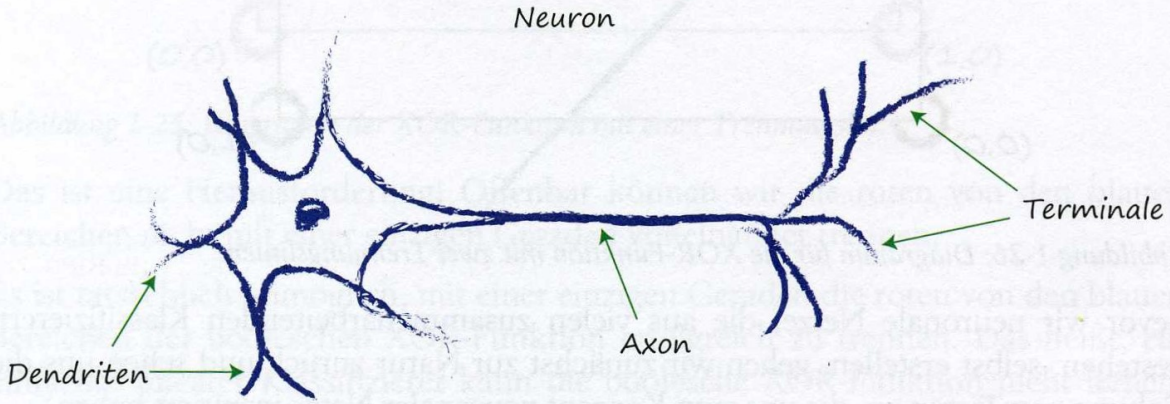


Abbildung 1-27: Neuron als Grundeinheit eines biologischen Gehirns

Es gibt zwar verschiedene Formen von Neuronen, doch übertragen sie alle ein elektrisches Signal von einem Ende zum anderen, von den Dendriten entlang der Axone zu den Terminalen. Diese Signale werden dann von einem Neuron an ein anderes übergeben. Auf diese Weise nehmen wir Licht, Töne, Druck, Wärme usw. wahr. Signale von spezialisierten Sensorneuronen werden entlang des Nervensystems zum Gehirn übertragen, das selbst wieder aus Neuronen besteht.

Abbildung 1-28 zeigt eine Skizze von Neuronen in einem Taubengehirn, das ein spanischer Neurowissenschaftler 1899 gezeichnet hat. Die wesentlichen Bestandteile – die Dendriten und die Terminale – sind deutlich zu erkennen.

Sehen wir uns also an, wie ein Neuron funktioniert. Es übernimmt ein elektrisches Eingangssignal und gibt ein anderes elektrisches Signal aus. Dies erinnert stark an die Klassifizierungs- oder Vorhersagemaschinen, die wir weiter oben gezeigt haben und die einen Eingang übernehmen, das Signal auf bestimmte Weise verarbeiten und ein Ausgangssignal ausgeben.

Könnten wir Neuronen als lineare Funktionen darstellen, wie wir es zuvor schon getan haben? Gute Idee – funktioniert aber nicht. Ein biologisches Neuron produziert keine Ausgabe, die schlichtweg eine einfache lineare Funktion der Eingabe ist. Das heißt, seine Ausgabe hat nicht die Form `Ausgabe = (Konstante * Eingabe) + (vielleicht_eine_andere_Konstante)`.

Beobachtungen legen nahe, dass Neuronen nicht sofort reagieren, sondern stattdessen die Eingabe unterdrücken, bis sie ausreichend groß ist, um ein Ausgabesignal auszulösen. Man kann sich dies als Schwellwert vorstellen, der erreicht sein muss, bevor irgendein Ausgabesignal entsteht. Vergleichbar ist das mit Wasser in einer Tasse – das Wasser läuft erst über, wenn die Tasse vollkommen gefüllt ist. Intuitiv ist das sinnvoll – die Neuronen sollen winzige Rauschsignale nicht durchlassen, sondern nur ausdrücklich starke, gewollte Signale. Abbildung 1-29 veranschaulicht dieses Konzept, bei dem ein Ausgangssignal nur entsteht, wenn die Eingabe ausreichend angestiegen ist, um einen *Schwellwert* zu überschreiten.

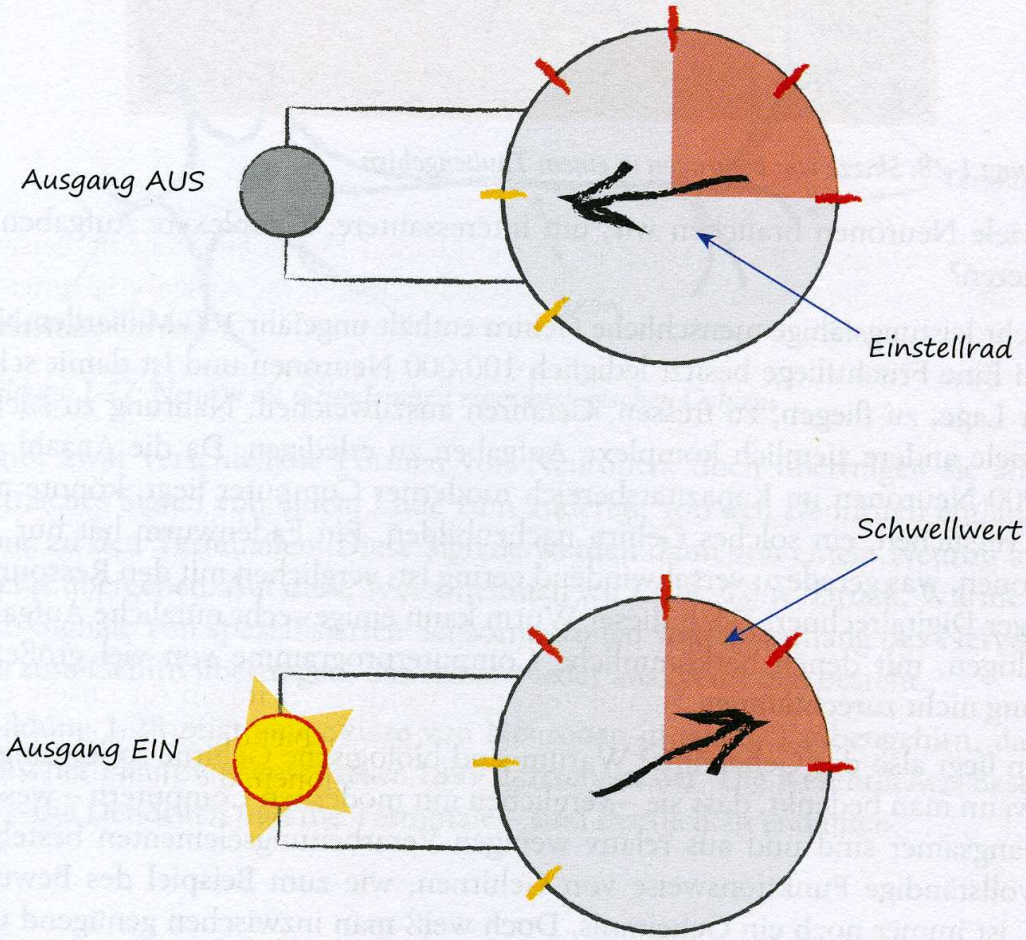


Abbildung 1-29: Ausgangssignal in Abhängigkeit von einem Schwellwert

Eine Funktion, die das Eingangssignal übernimmt und ein Ausgangssignal generiert, dabei aber eine Art Schwellwert berücksichtigt, wird *Aktivierungsfunktion* genannt. Im mathematischen Sinne gibt es viele derartige Aktivierungsfunktionen, die diesen Effekt erzielen. So ist das mit einer einfachen *Stufenfunktion* zu erreichen, wie sie Abbildung 1-30 zeigt.

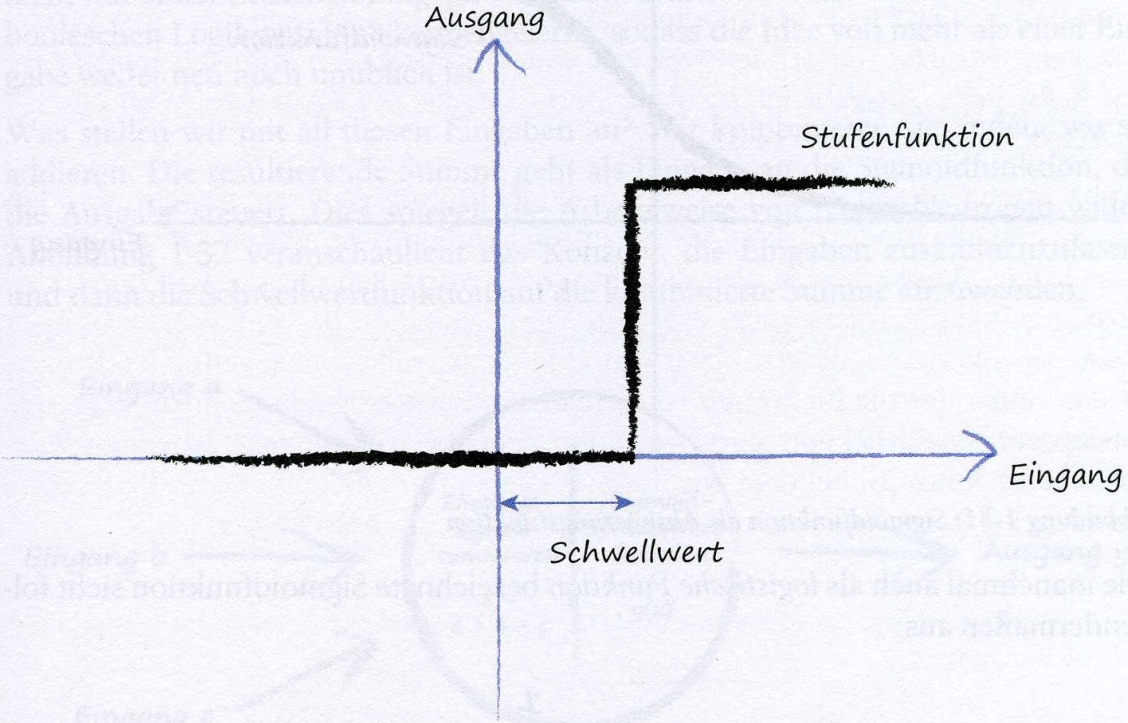


Abbildung 1-30: Eine einfache Stufenfunktion

Wie aus dem Diagramm hervorgeht, ist bei kleinen Eingabewerten die Ausgabe null. Nachdem aber die Eingabeschwelle erreicht ist, geht die Ausgabe sprungartig nach oben. Ein künstliches Neuron, das sich so verhält, wirkt fast wie ein reales biologisches Neuron. Der von Wissenschaftlern verwendete Begriff beschreibt dieses Verhalten treffend: Sie sagen, dass Neuronen *feuern*, wenn die Eingabe den Schwellwert erreicht.

Die Stufenfunktion lässt sich noch verbessern. Die in Abbildung 1-31 dargestellte s-förmige Funktion wird als *Sigmoidfunktion* bezeichnet. Sie verläuft sanfter als die abrupte Stufenfunktion, was sie natürlicher und realistischer macht – »natura non facit saltus« (die Natur macht keine Sprünge).

Eine solche sanfte s-förmige Sigmoidfunktion werden wir fortan für unsere eigenen neuronalen Netze verwenden. Forscher auf dem Gebiet der künstlichen Intelligenz nehmen auch andere, ähnlich aussehende Funktionen, die Sigmoide ist aber einfach und tatsächlich sehr gebräuchlich, sodass wir uns in guter Gesellschaft befinden.

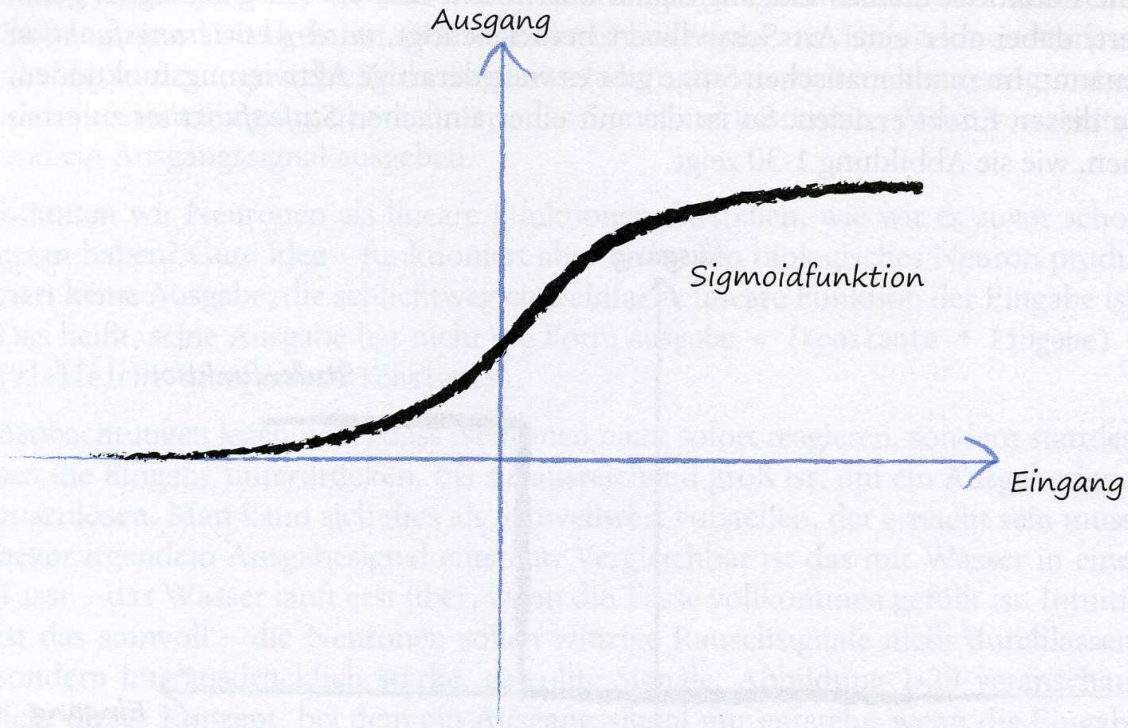


Abbildung 1-31: Sigmoidfunktion als Aktivierungsfunktion

Die manchmal auch als *logistische Funktion* bezeichnete Sigmoidfunktion sieht folgendermaßen aus:

$$y = \frac{1}{1 + e^{-x}}$$

In dieser Formel bezeichnet e die *Eulersche Zahl* 2,71828..., die nach dem Schweizer Mathematiker Leonhard Euler benannt ist. Diese Konstante spielt in vielen Bereichen der Mathematik und Physik eine zentrale Rolle. Die Punkte nach der Zahl weisen darauf hin, dass noch unendlich viele Dezimalstellen folgen, weil es sich um eine irrationale und transzendente Zahl handelt. Für unsere Zwecke kommen wir mit dem Näherungswert 2,71828 meistens ganz gut aus. Die Eingabe x wird negiert und e zur Potenz dieses -x erhoben. Zum Ergebnis addieren wir 1 und erhalten $1+e^{-x}$. Schließlich bilden wir den Kehrwert dieses Gesamtausdrucks, d. h., wir dividieren 1 durch $1+e^{-x}$. Genau dies stellt die Funktion oben mit dem Eingabewert x an, um den Ausgabewert y zu liefern. Am Ende ist der Ausdruck also gar nicht so schlimm!

Beachten Sie, dass bei x gleich null der Ausdruck e^{-x} gleich 1 ist, weil jede Zahl hoch 0 gleich 1 ist. In diesem Fall wird y zu 1 / (1 + 1) oder 1/2. Die standardmäßige Sigmoidfunktion schneidet also die y-Achse bei y = 1/2.

Es gibt noch einen weiteren sehr wichtigen Grund für die Wahl der Sigmoidfunktion gegenüber den unzähligen anderen s-förmigen Funktionen, die wir für die Ausgabe eines Neurons verwenden könnten: Mit der Sigmoidfunktion sind Be-

rechnungen wesentlich leichter durchzuführen als mit anderen s-förmigen Funktionen, wie wir später noch in der Praxis sehen werden.

Kommen wir zu den Neuronen zurück und überlegen wir, wie sich ein künstliches Neuron modellieren lässt.

Zunächst ist festzustellen, dass reale biologische Neuronen viele Eingaben und nicht nur einen einzelnen Eingabewert übernehmen. Wir haben das bereits bei der booleschen Logikmaschine kennengelernt, sodass die Idee von mehr als einer Eingabe weder neu noch unüblich ist.

Was stellen wir mit all diesen Eingaben an? Wir kombinieren sie, indem wir sie addieren. Die resultierende Summe geht als Eingabe an die Sigmoidfunktion, die die Ausgabe steuert. Dies spiegelt die Arbeitsweise von realen Neuronen wider. Abbildung 1-32 veranschaulicht das Konzept, die Eingaben zusammenzufassen und dann die Schwellwertfunktion auf die kombinierte Summe anzuwenden.

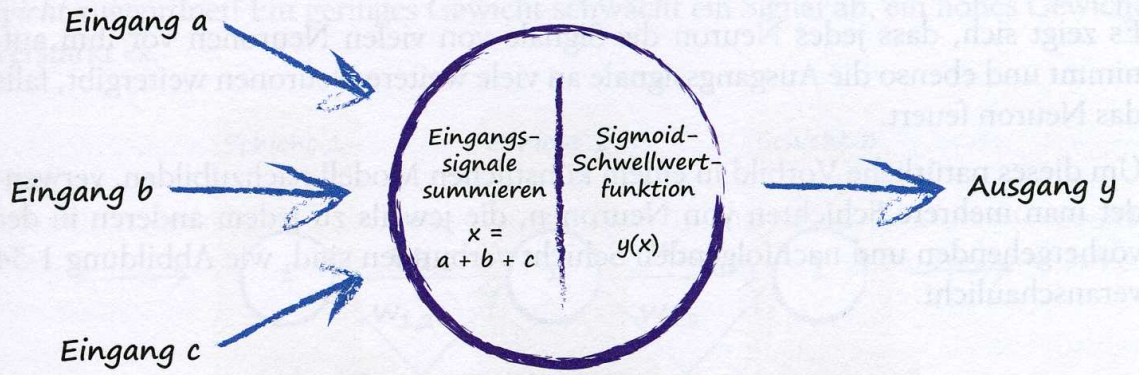


Abbildung 1-32: Modell eines künstlichen Neurons

Wenn das kombinierte Signal nicht groß genug ist, unterdrückt die sigmoidale Schwellwertfunktion das Ausgangssignal. Ist die Summe x ausreichend groß, bewirkt die Sigmoidfunktion, dass das Neuron feuert. Interessant ist Folgendes: Wenn nur einer von mehreren Eingängen groß ist und die übrigen Eingänge lediglich einen geringen Beitrag leisten, kann das bereits genügen, damit das Neuron feuert. Darüber hinaus kann das Neuron auch feuern, wenn einige der Eingänge für sich genommen ziemlich, aber nicht ausreichend groß sind, in der Summe aber ein Signal ergeben, das die Schwelle überwinden kann. Dies liefert einen ersten Eindruck von den komplexeren, in gewissem Sinne unscharfen Berechnungen, die derartige Neuronen realisieren können.

Die elektrischen Signale werden von den Dendriten gesammelt, und diese wirken zusammen, um ein stärkeres elektrisches Signal zu bilden. Wenn das Signal stark genug ist, um die Schwelle zu überwinden, feuert das Neuron ein Signal entlang des Axons zu den Terminalen, um es an die Dendriten der nächsten Neuronen weiterzugeben. Abbildung 1-33 zeigt, wie mehrere Neuronen auf diese Weise miteinander verbunden sind.

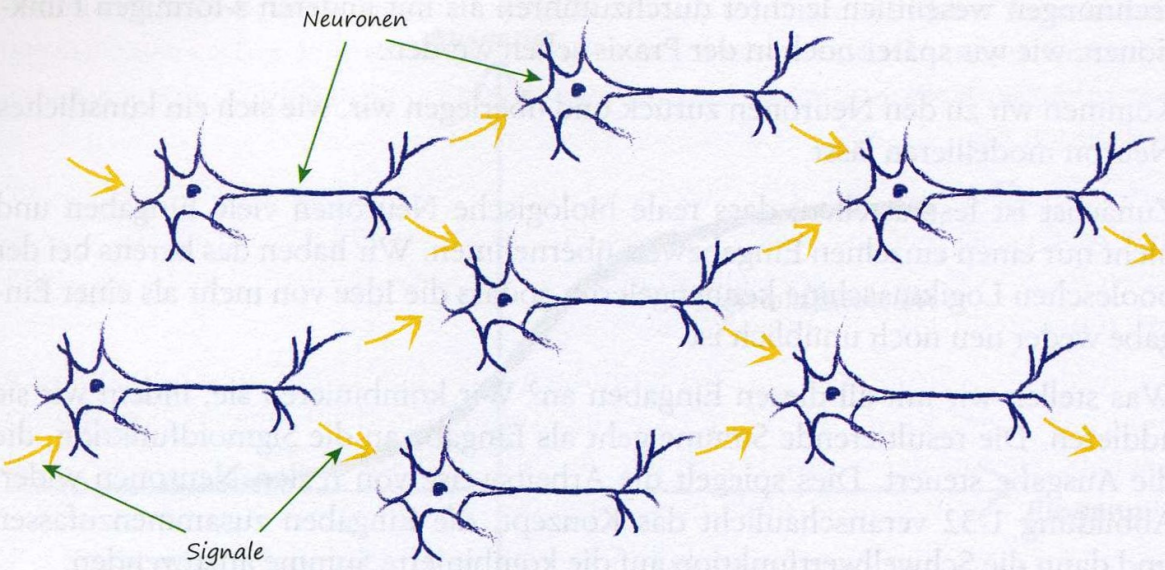


Abbildung 1-33: Mehrere Neuronen, die miteinander verbunden sind

Es zeigt sich, dass jedes Neuron die Signale von vielen Neuronen vor ihm aufnimmt und ebenso die Ausgangssignale an viele weitere Neuronen weitergibt, falls das Neuron feuert.

Um dieses natürliche Vorbild in einem künstlichen Modell nachzubilden, verwendet man mehrere Schichten von Neuronen, die jeweils zu jedem anderen in der vorhergehenden und nachfolgenden Schicht verbunden sind, wie Abbildung 1-34 veranschaulicht.

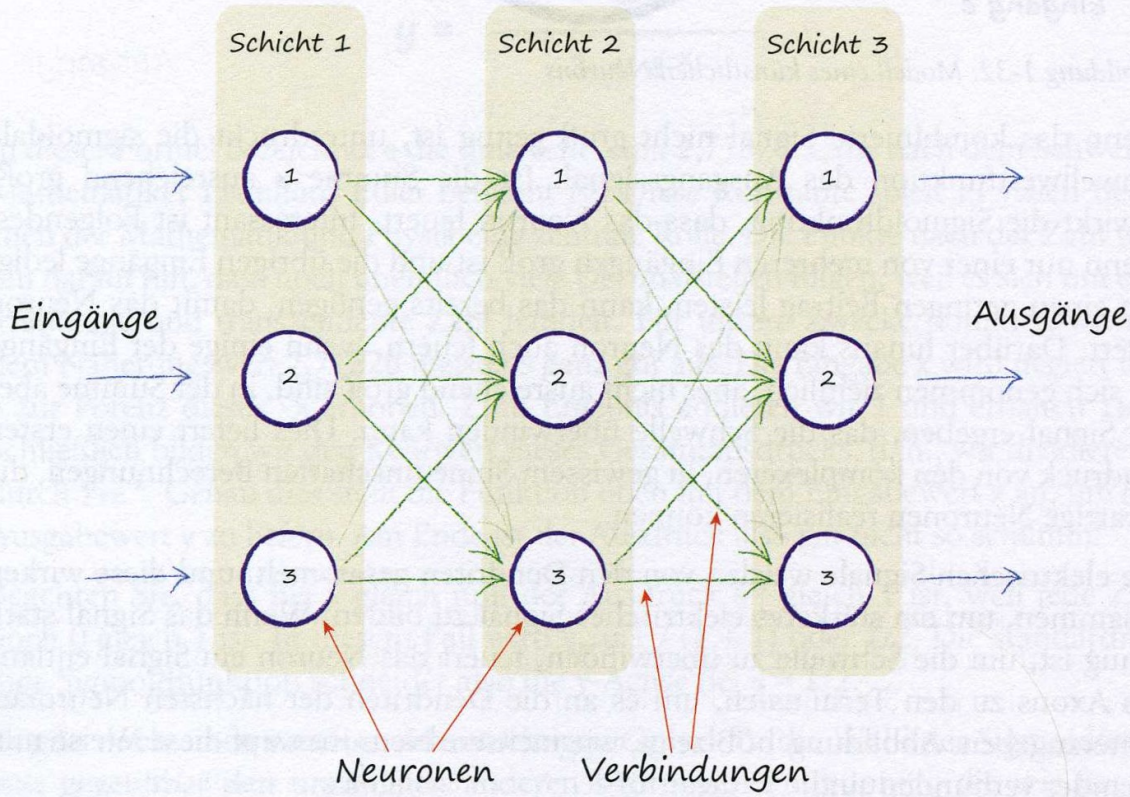


Abbildung 1-34: Modell mit mehreren Schichten von Neuronen, die jeweils mit den Neuronen der vorhergehenden und nachfolgenden Schichten verbunden sind

Das Modell in Abbildung 1-34 besteht aus drei Schichten, die jeweils drei künstliche Neuronen oder *Knoten* enthalten. Außerdem ist zu sehen, dass jeder Knoten mit jedem anderen Knoten in den vorhergehenden und nachfolgenden Schichten verbunden ist.

Großartig! Doch welcher Teil dieser cool aussehenden Architektur ist für das Lernen zuständig? Was passen wir als Reaktion auf Trainingsbeispiele an? Gibt es einen Parameter, den wir verfeinern können, wie es beim Anstieg der weiter oben behandelten linearen Klassifizierer geschehen ist?

Es liegt nahe, die Stärke der Verbindungen zwischen den Knoten anzupassen. Innerhalb eines Knotens könnte man die Summation der Eingänge beeinflussen oder die Gestalt der Schwellwertfunktion ändern, doch ist das komplizierter, als einfach die Stärke der Verbindungen zwischen den Knoten einzustellen.

Wenn das einfachere Konzept funktioniert, bleiben wir doch dabei! Abbildung 1-35 zeigt die verbundenen Knoten, doch dieses Mal ist jeder Verbindung auch ein *Gewicht* zugeordnet! Ein geringes Gewicht schwächt ein Signal ab, ein hohes Gewicht verstärkt es.

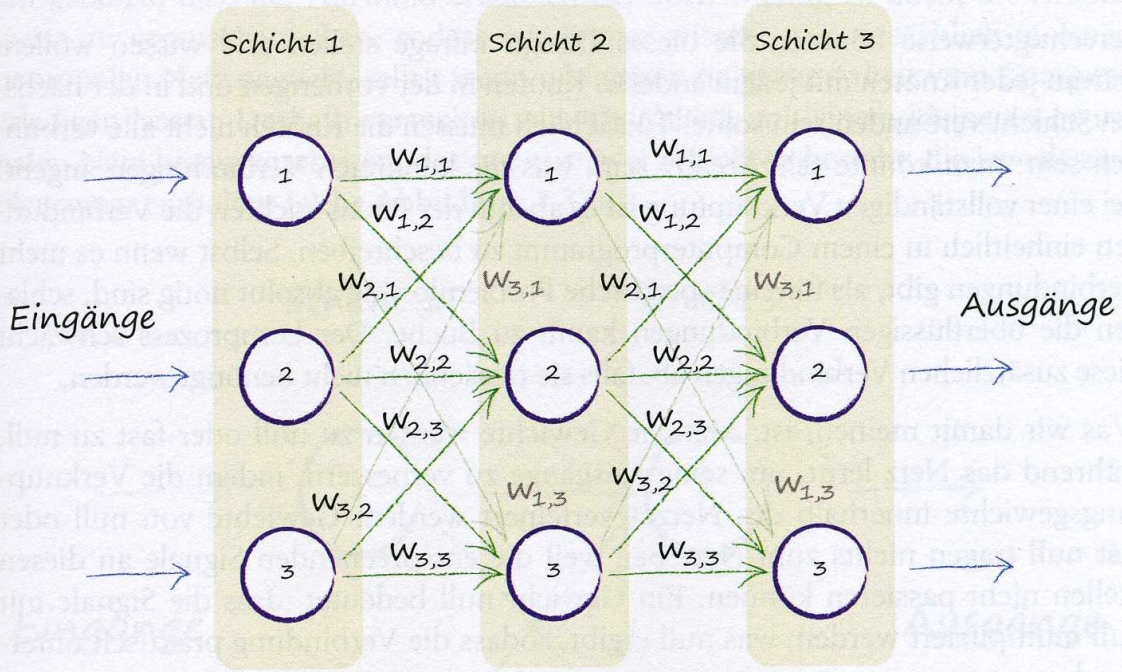


Abbildung 1-35: Schichtenmodell, in dem den Verbindungen zwischen den Knoten Gewichte zugeordnet sind

Die kleinen Zahlen neben den Gewichtssymbolen verdienen eine nähere Erläuterung. Das Gewicht $w_{2,3}$ ist dem Signal zugeordnet, das von Knoten 2 in der einen Schicht zu Knoten 3 in der nächsten Schicht übergeben wird. Somit ist $w_{1,2}$ das Gewicht, das das Signal zwischen Knoten 1 in der einen und Knoten 2 in der nächsten Schicht abschwächt bzw. verstärkt. Um dieses Konzept zu veranschaulichen, sind in Abbildung 1-36 diese beiden Verbindungen zwischen der ersten und zweiten Schicht rot dargestellt.

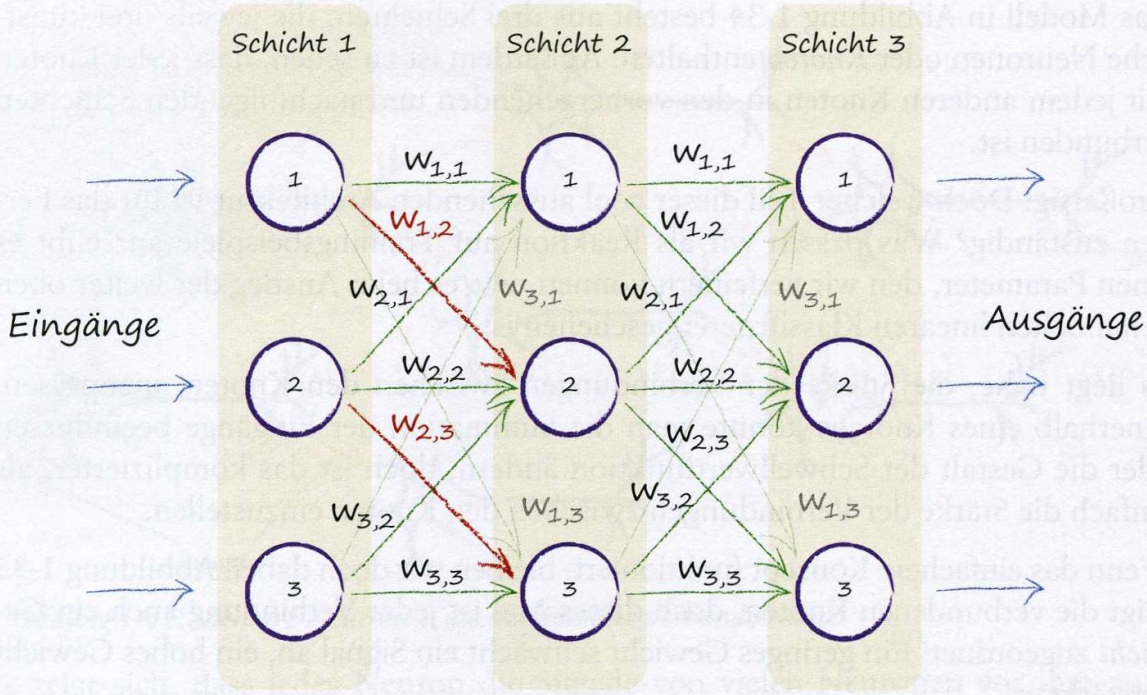


Abbildung 1-36: Die im Text beschriebenen Verbindungen sind hier rot gekennzeichnet.

Berechtigterweise könnten Sie dieses Design infrage stellen und wissen wollen, warum jeder Knoten mit jedem anderen Knoten in der vorherigen und in der nächsten Schicht verbunden sein sollte. Tatsächlich müssen die Knoten nicht alle verbunden sein; man könnte sehr kreativ sein, was die konkreten Verbindungen angeht. Bei einer vollständigen Verknüpfung ist es aber in der Praxis leichter, die Verbindungen einheitlich in einem Computerprogramm zu beschreiben. Selbst wenn es mehr Verbindungen gibt, als für eine spezifische Problemlösung absolut nötig sind, schlagen die überflüssigen Verbindungen kaum zu Buche. Der Lernprozess schwächt diese zusätzlichen Verbindungen ab, falls sie tatsächlich nicht benötigt werden.

Was wir damit meinen, ist: Manche Gewichte werden zu null oder fast zu null, während das Netz lernt, um seine Ausgänge zu verbessern, indem die Verknüpfungsgewichte innerhalb des Netzes verfeinert werden. Gewichte von null oder fast null tragen nichts zum Netz bei, weil die entsprechenden Signale an diesen Stellen nicht passieren können. Ein Gewicht null bedeutet, dass die Signale mit null multipliziert werden, was null ergibt, sodass die Verbindung praktisch unterbrochen ist.

Kernideen

- Biologische Gehirne führen anspruchsvolle Aufgaben wie fliegen, Nahrung finden, Sprachen lernen und vor Feinden fliehen scheinbar mühelos aus, obwohl sie offensichtlich weniger Speicherkapazität besitzen und wesentlich langsamer laufen als moderne Computer.

→

- Biologische Gehirne sind zudem unglaublich robust gegenüber beschädigten und unvollkommenen Signalen, verglichen mit herkömmlichen Computersystemen.
- Biologische Gehirne, die aus verknüpften Neuronen bestehen, sind die Inspiration für künstliche neuronale Netze.

Signalen durch ein neuronales Netz folgen

Dieses Bild mit drei Schichten aus Neuronen, wobei jedes Neuron mit jedem anderen in der nächsten und der vorherigen Schicht verbunden ist, sieht schon erstaunlich aus.

Doch das Verarbeitungskonzept, wie Signale von den Eingängen über die Schichten weiterlaufen, um zu Ausgängen zu werden, scheint ein wenig abschreckend zu sein und auch jede Menge Arbeit zu bedeuten!

Ich gebe zu, dass der Aufwand erheblich ist, doch wichtig ist auch, die Arbeitsweise zu veranschaulichen, sodass wir immer wissen, was tatsächlich in einem neuronalen Netz passiert, selbst wenn wir später die ganze Arbeit vom Computer erledigen lassen. Deshalb versuchen wir, die Abläufe mit einem kleineren neuronalen Netz herauszuarbeiten, das nur aus zwei Schichten besteht, die jeweils zwei Neuronen enthalten (siehe Abbildung 1-37).

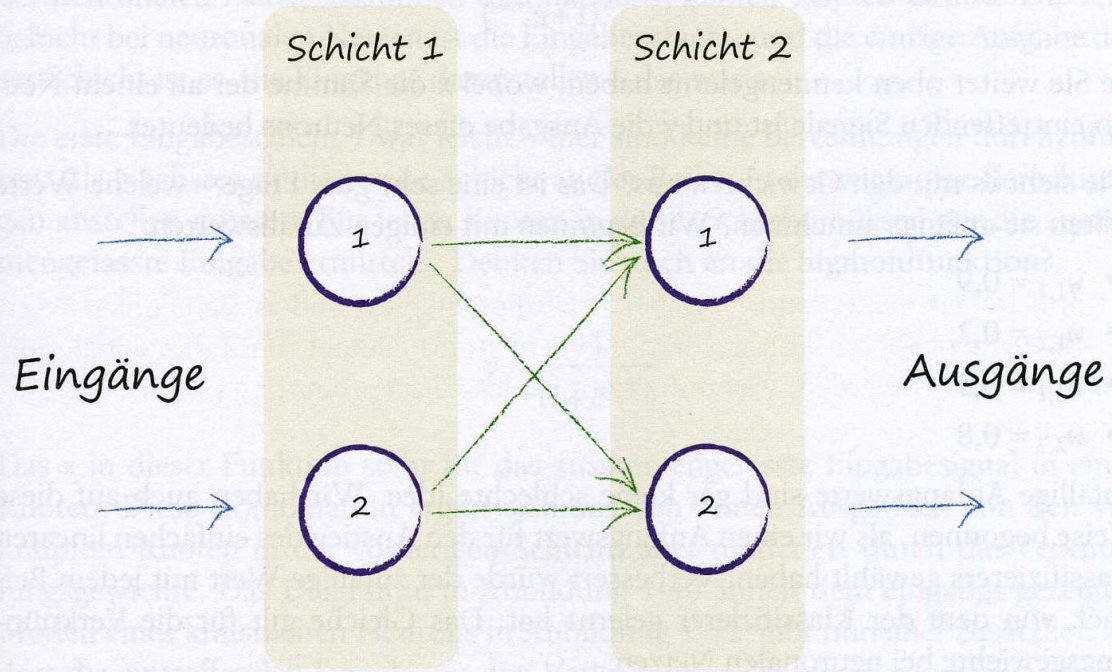


Abbildung 1-37: Einfacheres neuronales Netz

Nehmen wir an, dass an den beiden Eingängen die Werte 1,0 und 0,5 anliegen. Abbildung 1-38 zeigt das kleinere neuronale Netz mit diesen Eingaben.

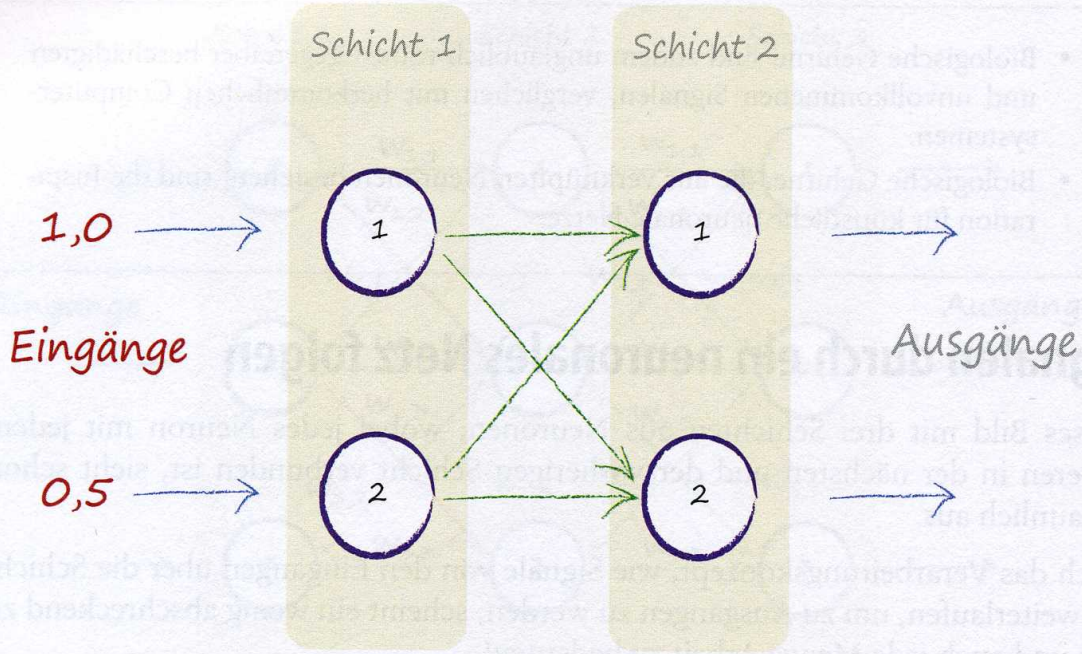


Abbildung 1-38: Neuronales Netz mit Eingabewerten

Genau wie zuvor wandelt jeder Knoten die Summe der Eingaben mithilfe einer Aktivierungsfunktion in eine Ausgabe um.

Wir verwenden auch hier die Sigmoidfunktion

$$y = \frac{1}{(1+e^{-x})}$$

die Sie weiter oben kennengelernt haben, wobei x die Summe der an einem Neuron eintreffenden Signale ist und y die Ausgabe dieses Neurons bedeutet.

Wie sieht es mit den Gewichten aus? Das ist eine sehr gute Frage – welche Werte sollten sie anfangs annehmen? Wir beginnen mit einigen Zufallswerten:

- $w_{1,1} = 0{,}9$
- $w_{1,2} = 0{,}2$
- $w_{2,1} = 0{,}3$
- $w_{2,2} = 0{,}8$

Zufällige Anfangswerte sind gar keine schlechte Idee. Wir haben auch auf diese Weise begonnen, als wir einen Anfangswert für den Anstieg des einfachen linearen Klassifizierers gewählt haben. Verbessert wurde der zufällige Wert mit jedem Beispiel, von dem der Klassifizierer gelernt hat. Das Gleiche gilt für die Verknüpfungsgewichte bei neuronalen Netzen.

In diesem kleinen neuronalen Netz gibt es nur vier Gewichte, und das sind sämtliche Kombinationen für die Verknüpfung der zwei Knoten in jeder Schicht. Im Diagramm, das Abbildung 1-39 zeigt, sind alle diese Zahlen jetzt markiert.

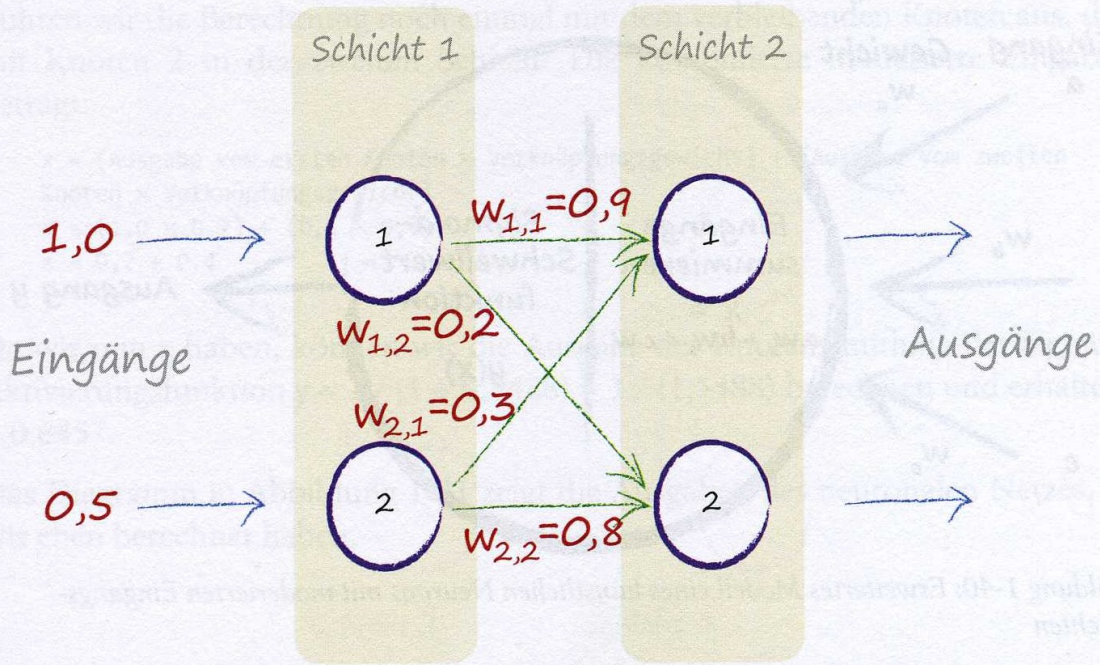


Abbildung 1-39: Das kleine neuronale Netz, in dem die zufällig gewählten Anfangsgewichte markiert sind

Fangen wir an zu rechnen!

Die erste Schicht der Knoten ist die Eingabeschicht. Ihr fällt lediglich die Aufgabe zu, die Eingabesignale darzustellen. Das heißt, die Eingabeknoten wenden keine Aktivierungsfunktion auf die Eingabe an. Das hängt mit der Entwicklungsgeschichte der neuronalen Netze zusammen und hat sonst keinen tieferen Grund. Die erste Schicht bei neuronalen Netzen ist die Eingabeschicht, und die einzige Aufgabe dieser Schicht ist es, die Eingänge darzustellen – das war's.

Die erste Eingabeschicht 1 war leicht – hier sind keine Berechnungen durchzuführen. Als Nächstes ist die zweite Schicht an der Reihe, in der wir einige Berechnungen anstellen müssen. Für jeden Knoten in dieser Schicht müssen wir die zusammengefasste Eingabe ermitteln. Denken Sie noch an die Sigmoidfunktion?

$$y = \frac{1}{(1+e^{-x})}$$

Das x in dieser Funktion steht für das zusammengefasste Eingabesignal in einen Knoten. Diese Kombination ergibt sich aus den reinen Ausgängen von den verknüpften Knoten in der vorherigen Schicht, aber moderiert durch die Verknüpfungsgewichte. Das Diagramm in Abbildung 1-40 ähnelt dem eingangs gezeigten Modell eines künstlichen Neurons in Abbildung 1-32, gibt nun aber zusätzlich an, dass die eintreffenden Signale mit den Verknüpfungsgewichten moderiert werden müssen.

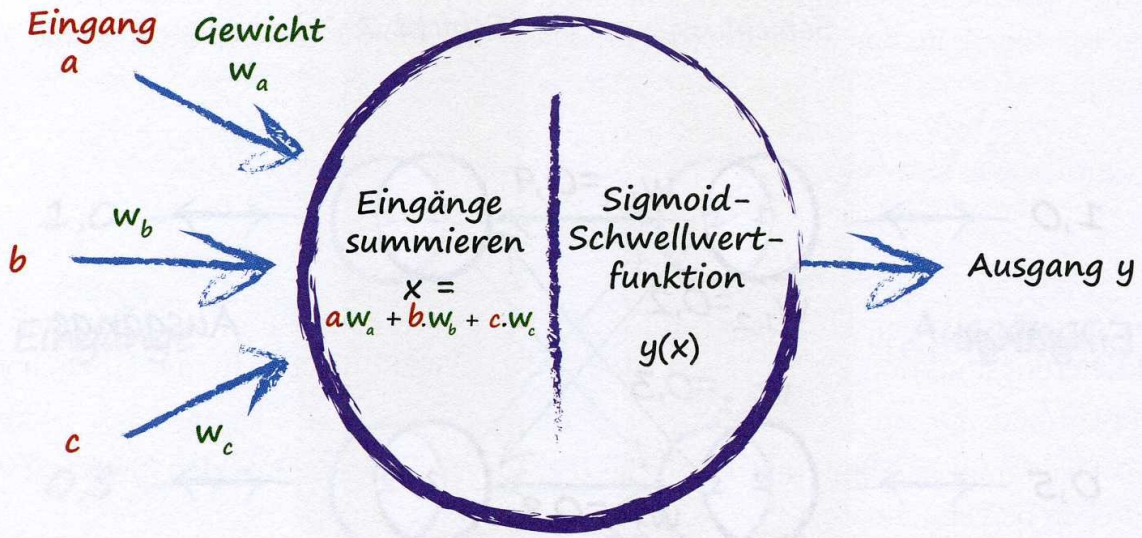


Abbildung 1-40: Erweitertes Modell eines künstlichen Neurons mit moderierten Eingangsgewichten

Konzentrieren wir uns zunächst auf Knoten 1 in Schicht 2. Beide Knoten in der ersten Eingabeschicht sind mit ihm verbunden. Diese Eingabeknoten haben die Rohwerte 1,0 und 0,5. Der Verknüpfung vom ersten Knoten ist ein Gewicht von 0,9 zugeordnet, der Verknüpfung vom zweiten Knoten ein Gewicht von 0,3. Der kombinierte moderierte Eingang berechnet sich also wie folgt:

```
x = (Ausgabe vom ersten Knoten × Verknüpfungsgewicht) +
(Ausgabe vom zweiten Knoten × Verknüpfungsgewicht)
x = (1,0 × 0,9) + (0,5 × 0,3)
x = 0,9 + 0,15
x = 1,05
```

Wenn wir das Signal nicht moderiert hätten, wäre es eine sehr einfache Addition der Signale 1,0 + 0,5, doch das wollen wir gar nicht. Es sind die Gewichte, die für das Lernen in neuronalen Netzen zuständig sind, indem sie iterativ verfeinert werden, um immer bessere Ergebnisse zu liefern.

Momentan haben wir also x = 1,05 für die kombinierte moderierte Eingabe in den ersten Knoten der zweiten Schicht bekommen. Schließlich können wir die Ausgabe dieses Knotens mithilfe der Aktivierungsfunktion

$$y = \frac{1}{(1+e^{-x})}$$

berechnen – wenn Sie wollen, mit einem Taschenrechner. Das Ergebnis lautet y = 1 / (1 + 0,3499) = 1 / 1,3499. Somit ist y = 0,7408.

Gute Arbeit! Wir haben nun eine tatsächliche Ausgabe von einem der beiden Ausgabeknoten des Netzes.

Führen wir die Berechnung noch einmal mit dem verbleibenden Knoten aus, d. h. mit Knoten 2 in der zweiten Schicht. Die kombinierte moderierte Eingabe x beträgt:

```
x = (Ausgabe vom ersten Knoten × Verknüpfungsgewicht) + (Ausgabe vom zweiten
Knoten × Verknüpfungsgewicht)
x = (1,0 × 0,2) + (0,5 × 0,8)
x = 0,2 + 0,4
x = 0,6
```

Da wir nun x haben, können wir die Ausgabe des Knotens mithilfe der Sigmoid-Aktivierungsfunktion y = 1 / (1 + 0,5488) = 1 / (1,5488) berechnen und erhalten y = 0,6457.

Das Diagramm in Abbildung 1-41 zeigt die Ausgaben des neuronalen Netzes, die wir eben berechnet haben.

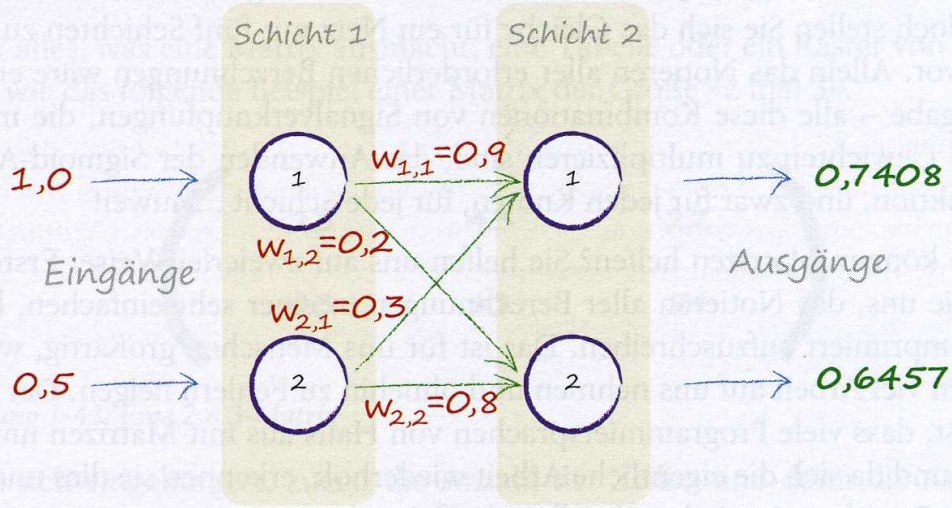


Abbildung 1-41: Das neuronale Netz mit den berechneten Ausgabewerten

Das war recht viel Arbeit, nur um zwei Ausgaben von einem sehr vereinfachten Netz zu bekommen. Für ein größeres Netz möchte ich die gesamten Berechnungen keinesfalls manuell erledigen! Erfreulicherweise sind Computer prädestiniert dafür, Unmengen von Berechnungen schnell auszuführen, ohne müde zu werden.

Dennoch bin ich nicht erpicht darauf, die Computeranweisungen für die Berechnung eines Netzes mit mehr als zwei Schichten und vielleicht mit vier, acht oder sogar Hunderten von Knoten in jeder Schicht zu schreiben. Denn selbst das Formulieren der Anweisungen ist mühselig, und ich werde höchstwahrscheinlich Fehler machen, wenn ich alle diese Anweisungen für alle diese Knoten und alle diese Schichten schreiben müsste, geschweige denn die eigentlichen Berechnungen ausführen müsste.

Zum Glück hilft uns hier die Mathematik, die erforderlichen Berechnungen für die Ermittlung der Ausgabe von einem neuronalen Netz äußerst knapp zu formulie-

ren, selbst bei einem Netz mit wesentlich mehr Schichten und Knoten. Diese Prägnanz ist nicht nur gut für uns menschliche Leser, sondern auch hervorragend für Computer, weil die Anweisungen viel kürzer sind und die Ausführung zudem wesentlich effizienter ist.

Dieser prägnante Ansatz verwendet *Matrizen*, die wir uns als Nächstes ansehen.

Matrizenmultiplikation ist nützlich – ehrlich!

Matrizen haben einen schlechten Ruf. Sie rufen Erinnerungen wach an zähneknirschende, langweilige, mühselige und scheinbar nutzlose Stunden, die wir in der Schule mit Matrizenmultiplikation zugebracht haben.

Weiter oben haben wir bereits die Berechnungen für ein zweischichtiges Netz mit lediglich zwei Knoten in jeder Schicht manuell durchgeführt. Das war genug Arbeit, doch stellen Sie sich das Gleiche für ein Netz mit fünf Schichten zu je 100 Knoten vor. Allein das Notieren aller erforderlichen Berechnungen wäre eine riesige Aufgabe – alle diese Kombinationen von Signalverknüpfungen, die mit den richtigen Gewichten zu multiplizieren sind, das Anwenden der Sigmoid-Aktivierungsfunktion, und zwar für jeden Knoten, für jede Schicht ... auwei!

Wie also können Matrizen helfen? Sie helfen uns auf zweierlei Weise. Erstens erlauben sie uns, das Notieren aller Berechnungen in einer sehr einfachen, kurzen Form komprimiert aufzuschreiben. Das ist für uns Menschen großartig, weil wir nicht gern viel Arbeit auf uns nehmen und ohnehin zu Fehlern neigen. Der zweite Vorteil ist, dass viele Programmiersprachen von Haus aus mit Matrizen umgehen können, und da sich die eigentliche Arbeit wiederholt, erkennen sie dies und realisieren die Berechnungen sehr schnell und effizient.

Kurz gesagt, wir sind mit Matrizen in der Lage, die erforderlichen Arbeiten prägnant und einfach auszudrücken, und Computer können die Berechnungen schnell und effizient ausführen.

Nachdem klar ist, warum wir uns mit Matrizen befassen, werden wir sie – wegen der schmerzlichen Erfahrungen aus der Schulzeit – zunächst einmal demystifizieren.

Eine *Matrix* ist lediglich eine Tabelle, ein rechteckiges Raster von Zahlen. Mehr nicht. Hinter einer Matrix versteckt sich nichts Kompliziertes.

Wenn Sie bereits mit einer Tabellenkalkulation gearbeitet haben, sind Sie auch mit der Anordnung von Zahlen in einem Raster vertraut. Man kann dies als Tabelle bezeichnen, aber auch von einer Matrix sprechen. Abbildung 1-42 zeigt eine Tabellenkalkulation mit einer Tabelle aus Zahlen.

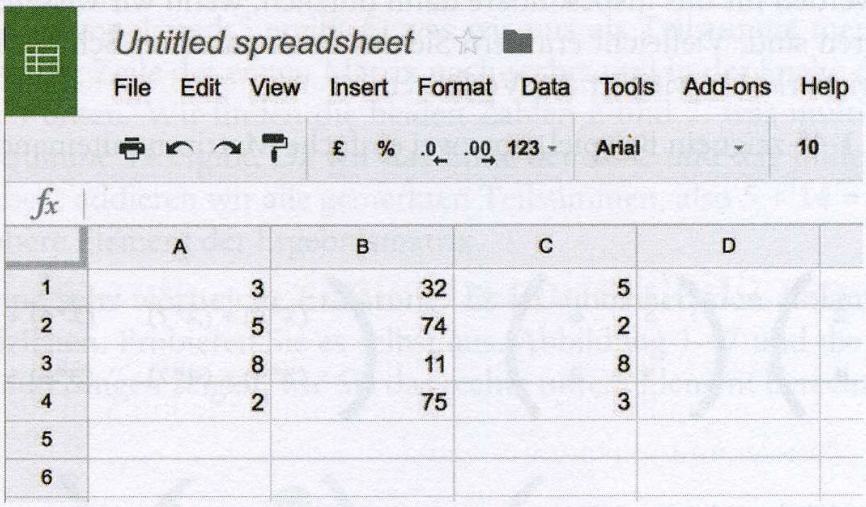


Abbildung 1-42: Eine Matrix in Form einer Tabellenkalkulation

Das ist alles, was eine Matrix ausmacht, eine Tabelle oder ein Raster von Zahlen – genau wie das folgende Beispiel einer Matrix der Größe »2 mal 3«.

$$\begin{pmatrix} 23 & 43 & 22 \\ 43 & 12 & 54 \end{pmatrix}$$

Abbildung 1-43: Eine 2 × 3-Matrix

Es hat sich eingebürgert, zuerst die Anzahl der Zeilen und dann die Anzahl der Spalten zu nennen. Die Matrix in Abbildung 1-43 ist also keine 3 × 2-, sondern eine 2 × 3-Matrix.

Außerdem ziehen es manche Menschen vor, Matrizen in eckige Klammern zu schreiben, während andere die runden Klammern wie hier im Buch bevorzugen.

Die Matrixelemente müssen nicht unbedingt Zahlen sein, sie können auch Größen sein, denen wir einen Namen geben, denen aber möglicherweise kein numerischer Wert zugeordnet ist. So zeigt Abbildung 1-44 eine Matrix, bei der jedes Element eine *Variable* ist, die eine Bedeutung und eventuell einen zugeordneten numerischen Wert besitzt, doch wir haben einfach noch nicht gesagt, was das für ein Wert ist.

$$\begin{pmatrix} \text{Längengrad des Schiffes} & \text{Längengrad des Flugzeugs} \\ \text{Breitengrad des Schiffes} & \text{Breitengrad des Flugzeugs} \end{pmatrix}$$

Abbildung 1-44: Eine Matrix, deren Elemente Variablen sind

Matrizen werden für uns insbesondere dann nützlich, wenn wir wissen, wie sie zu multiplizieren sind. Vielleicht erinnern Sie das noch aus Ihrer Schulzeit, doch für alle Fälle wiederholen wir hier das Verfahren.

Abbildung 1-45 zeigt ein Beispiel, um zwei einfache Matrizen miteinander zu multiplizieren.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ (3*5) + (4*7) & (3*6) + (4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Abbildung 1-45: Beispiel für eine Matrizenmultiplikation

Wie aus Abbildung 1-45 hervorgeht, multiplizieren wir nicht einfach die korrespondierenden Elemente miteinander. Das linke obere Element des Ergebnisses ist nicht 1×5, und das Element rechts unten ist nicht 4×8.

Stattdessen sind bei der Matrizenmultiplikation verschiedene Regeln anzuwenden. Wenn Sie wollen, können Sie sich die Regeln anhand des Beispiels in Abbildung 1-45 selbst herleiten. Für alle anderen erläutern die folgenden Ausführungen, wie das linke obere Element in der Ergebnismatrix berechnet wird.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ (3*5) + (4*7) & (3*6) + (4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Abbildung 1-46: Matrizenmultiplikation – das linke obere Element der Ergebnismatrix berechnen

Um das linke obere Element der Ergebnismatrix zu berechnen, folgen Sie der obersten Zeile der ersten Matrix und der linken Spalte der zweiten Matrix. Dabei multiplizieren Sie die jeweiligen Zeilen- und Spaltenelemente und bilden eine laufende Gesamtsumme. Wir beginnen mit dem ersten Element in der obersten Zeile des ersten Arrays, wo wir die Zahl 1 finden, und multiplizieren es mit dem ersten

Element in der linken Spalte der zweiten Matrix, wo wir die Zahl 5 finden. Die Multiplikation von 1 und 5 ergibt 5, was wir uns als Teilsumme merken. Dann gehen wir in der Zeile der ersten Matrix nach rechts und in der Spalte der zweiten Matrix nach unten. Wir finden die beiden Zahlen 2 und 7, was multipliziert die zweite Teilsumme 14 ergibt. Da wir das Ende der Zeile und das Ende der Spalte erreicht haben, addieren wir alle gemerkten Teilsummen, also 5 + 14 = 19. Das ist das linke obere Element der Ergebnismatrix.

Das war eine sehr wortreiche Erklärung. Es ist einfacher, den Ablauf praktisch nachzuvollziehen. Probieren Sie es selbst aus. Abbildung 1-47 und die dazugehörenden Erläuterungen zeigen, wie Sie das rechte untere Element berechnen.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ (3*5) + (4*7) & (3*6) + (4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Abbildung 1-47: Matrizenmultiplikation für das Matrixelement rechts unten

Wieder folgen wir der Zeile und Spalte des korrespondierenden Elements, das wir berechnen wollen (der zweiten Zeile und der zweiten Spalte in diesem Beispiel), und erhalten (3 × 6) und (4 × 8), was 18 + 32 = 50 liefert.

Für das linke untere Element ergibt sich (3 × 5) + (4 × 7) = 15 + 28 = 43. Und schließlich berechnen wir für das rechte obere Element: (1 × 6) + (2 × 8) = 6 + 16 = 22.

Abbildung 1-48 veranschaulicht die Regel mithilfe von Variablen anstelle von Zahlen.

$$\begin{pmatrix} a & b & .. \\ c & d & .. \end{pmatrix} \begin{pmatrix} e & f \\ g & h \\ .. & .. \end{pmatrix} = \begin{pmatrix} (a*e) + (b*g) + ... (a*f) + (b*h) + ... \\ (c*e) + (d*g) + ... (c*f) + (d*h) + ... \end{pmatrix}$$

$$= \begin{pmatrix} ae+bg+... \; af+bh+... \\ ce+dg+... \; cf+dh+... \end{pmatrix}$$

Abbildung 1-48: Matrizenmultiplikation – Berechnungsregel, dargestellt durch Variablen

Dies ist lediglich eine andere Art, den Ablauf bei einer Matrizenmultiplikation zu erklären. Mithilfe von Buchstaben, die jeden beliebigen Zahlenwert annehmen können, haben wir den generischen Ansatz für die Multiplikation von Matrizen noch deutlicher gemacht. Der Ansatz ist generisch, weil er sich auch auf Matrizen unterschiedlicher Größen anwenden lässt.

Wir haben zwar gesagt, dass das Verfahren für Matrizen verschiedener Größen funktioniert, doch es gibt eine wichtige Beschränkung. Man kann nicht einfach zwei beliebige Matrizen multiplizieren, sie müssen auch kompatibel sein. Das haben Sie möglicherweise schon geahnt, als Sie den Zeilen der ersten Matrix und den Spalten der zweiten Matrix gefolgt sind. Denn wenn die Anzahl der Elemente in den Zeilen nicht mit der Anzahl der Elemente in den Spalten übereinstimmt, funktioniert die Methode nicht. Es ist also nicht möglich, eine 2×2-Matrix mit einer 5×5-Matrix zu multiplizieren. Probieren Sie es aus – Sie werden feststellen, warum es nicht klappt. Um Matrizen zu multiplizieren, muss die Anzahl der Spalten in der ersten Matrix gleich der Anzahl der Zeilen in der zweiten Matrix sein.

Manche Lehrbücher bezeichnen eine derartige Matrizenmultiplikation als *Punktprodukt* oder *inneres Produkt*. Für Matrizen sind prinzipiell verschiedene Arten der Multiplikation möglich, wie zum Beispiel das Kreuzprodukt, doch was wir hier brauchen, ist tatsächlich das Punktprodukt.

Warum sind wir hinabgestiegen in die Niederungen der gefürchteten Matrizenmultiplikation und widerwärtigen Algebra? Dafür gibt es einen sehr guten Grund ... bleiben Sie neugierig!

Sehen Sie sich an, was passiert, wenn wir die Buchstaben durch Wörter ersetzen, die für unsere neuronalen Netze aussagekräftiger sind. Die zweite Matrix ist eine 2×1-Matrix, doch das Multiplikationsverfahren ist das gleiche.

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input_1 \\ input_2 \end{pmatrix} = \begin{pmatrix} (input_1 * w_{1,1}) + (input_2 * w_{2,1}) \\ (input_1 * w_{1,2}) + (input_2 * w_{2,2}) \end{pmatrix}$$

Abbildung 1-49: Matrizenmultiplikation – mit der Terminologie von neuronalen Netzen

Wunderbar!

Die erste Matrix enthält die Gewichte zwischen den Knoten der beiden Schichten. In der zweiten Matrix stehen die Signale der ersten Eingabeschicht. Wenn wir diese beiden Matrizen multiplizieren, erhalten wir als Antwort die kombinierten moderierten Signale, die zu den Knoten der zweiten Schicht gehen. Sehen Sie sich das genau an, um es sich klarzumachen. Für den ersten Knoten der zweiten Schicht wird der Eingang `input_1` mit dem moderierten Gewicht $w_{1,1}$ zum zweiten Eingang `input_2` mit dem moderierten Gewicht $w_{2,1}$ addiert. Das sind die Werte von `x`, bevor die Sigmoid-Aktivierungsfunktion angewendet wird.

Abbildung 1-50 verdeutlicht diesen Schritt.

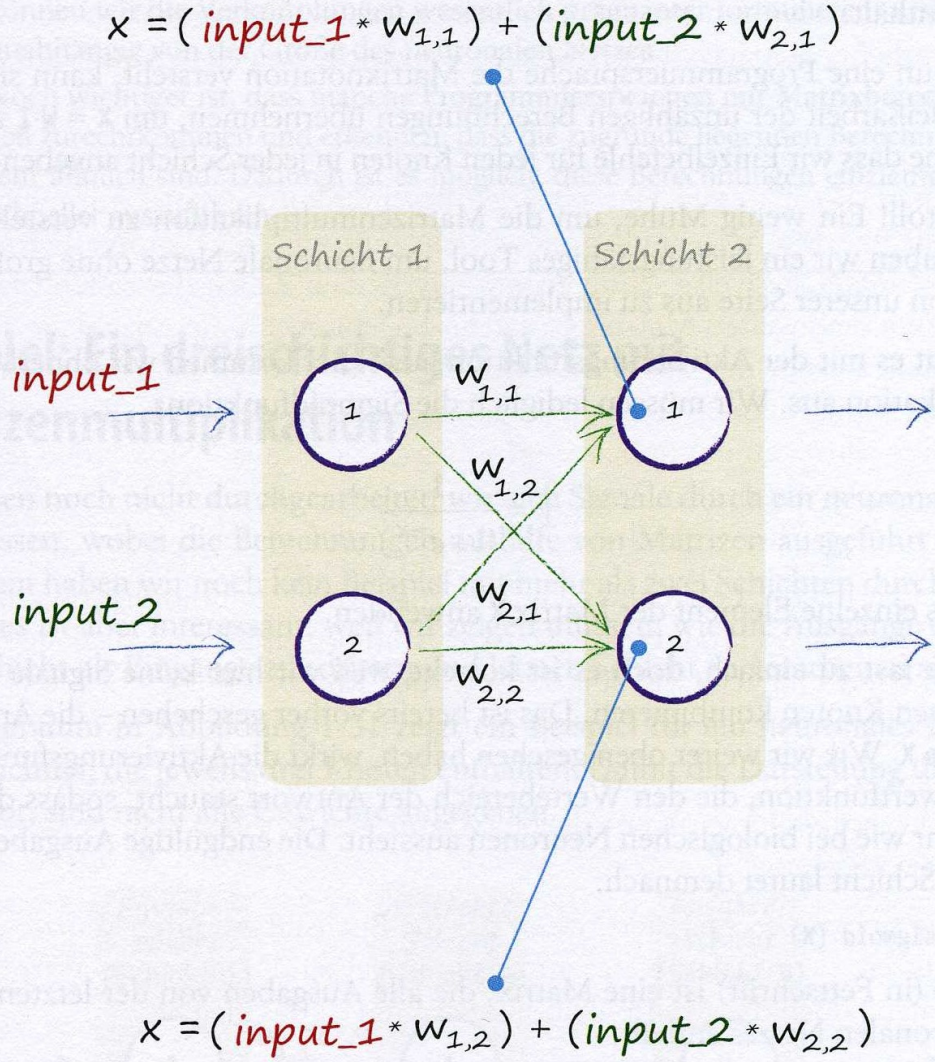


Abbildung 1-50: Bildung der Eingangswerte für die zweite Schicht

Dies ist wirklich sehr nützlich!

Warum? Weil wir sämtliche Berechnungen, die mit den Ausgangssignalen der ersten Schicht anzustellen sind, um für jeden Knoten der zweiten Schicht das moderierte Signal x zu bilden, durch Matrizenmultiplikation ausdrücken können. Das lässt sich kurz und bündig wie folgt formulieren:

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

Das heißt, **W** ist die Gewichtsmatrix, **I** ist die Eingabematrix, und **X** ist die resultierende Matrix der kombinierten moderierten Signale für Schicht 2. Es hat sich eingebürgert, Matrizen in Fettschrift zu schreiben, um deutlich zu machen, dass es sich tatsächlich um Matrizen und nicht nur um einzelne Zahlen bzw. Elemente handelt.

Wir brauchen uns jetzt nicht mehr so sehr darum zu kümmern, wie viele Knoten in jeder Schicht vorhanden sind. Wenn wir mehr Knoten haben, werden die Mat-

rizen lediglich größer. Doch wir müssen nichts länger oder größer notieren. Wir können einfach **W·I** schreiben, selbst wenn **I** nur 2 Elemente oder sogar 200 Elemente enthält.

Wenn nun eine Programmiersprache die Matrixnotation versteht, kann sie die gesamte Fleißarbeit der unzähligen Berechnungen übernehmen, um **X** = **W·I** zu ermitteln, ohne dass wir Einzelbefehle für jeden Knoten in jeder Schicht angeben müssen.

Das ist toll! Ein wenig Mühe, um die Matrizenmultiplikation zu verstehen, und schon haben wir ein leistungsfähiges Tool, um neuronale Netze ohne großen Aufwand von unserer Seite aus zu implementieren.

Wie sieht es mit der Aktivierungsfunktion aus? Hier kommen wir ohne Matrizenmultiplikation aus. Wir müssen lediglich die Sigmoidfunktion

$$y = \frac{1}{(1+e^{-x})}$$

auf jedes einzelne Element der Matrix **X** anwenden.

Es klingt fast zu einfach, doch es ist korrekt, weil wir hier keine Signale von verschiedenen Knoten kombinieren. Das ist bereits vorher geschehen – die Antworten stehen in **X**. Wie wir weiter oben gesehen haben, wirkt die Aktivierungsfunktion als Schwellwertfunktion, die den Wertebereich der Antwort staucht, sodass die Reaktion mehr wie bei biologischen Neuronen aussieht. Die endgültige Ausgabe von der zweiten Schicht lautet demnach:

```
O = sigmoid (X)
```

Dieses **O** (in Fettschrift) ist eine Matrix, die alle Ausgaben von der letzten Schicht des neuronalen Netzes enthält.

Der Ausdruck **X** = **W·I** gilt für die Berechnungen zwischen einer Schicht und der nächsten. Wenn das neuronale Netz zum Beispiel drei Schichten umfasst, führen wir die Matrizenmultiplikation einfach erneut aus, wobei die Ausgänge der zweiten Schicht als Eingänge für die dritte Schicht dienen und dort natürlich wieder kombiniert und mit weiteren Gewichten moderiert werden.

Genug der Theorie! Wir wollen uns an einem realen Beispiel ansehen, wie das Ganze funktioniert. Dieses Mal aber verwenden wir ein etwas größeres Netz mit drei Schichten, die jeweils drei Knoten enthalten.

Kernideen

- Die vielen erforderlichen Berechnungen, um ein Signal durch ein neuronales Netz zu leiten, lassen sich als Matrizenmultiplikation ausdrücken.

→

- Da sich die Berechnungen als Matrizenmultiplikation ausdrücken lassen, können wir die Verknüpfungen wesentlich prägnanter formulieren, und zwar unabhängig von der Größe des neuronalen Netzes.
- Noch wichtiger ist, dass manche Programmiersprachen mit Matrixberechnungen zurechtkommen und erkennen, dass die zugrunde liegenden Berechnungen sehr ähnlich sind. Dadurch ist es möglich, diese Berechnungen effizienter und schneller auszuführen.

Beispiel: Ein dreischichtiges Netz mit Matrizenmultiplikation

Wir haben noch nicht durchgearbeitet, wie sich Signale durch ein neuronales Netz leiten lassen, wobei die Berechnungen mithilfe von Matrizen ausgeführt werden. Außerdem haben wir noch kein Beispiel mit mehr als zwei Schichten durchgenommen. Dies ist aber interessant, weil wir zeigen müssen, wie die Ausgänge der mittleren Schicht als Eingänge zur dritten und letzten Schicht zu verarbeiten sind.

Das Diagramm in Abbildung 1-51 zeigt ein Beispiel für ein neuronales Netz mit drei Schichten, die jeweils drei Knoten enthalten. Damit die Darstellung übersichtlich bleibt, sind nicht alle Gewichte angegeben.

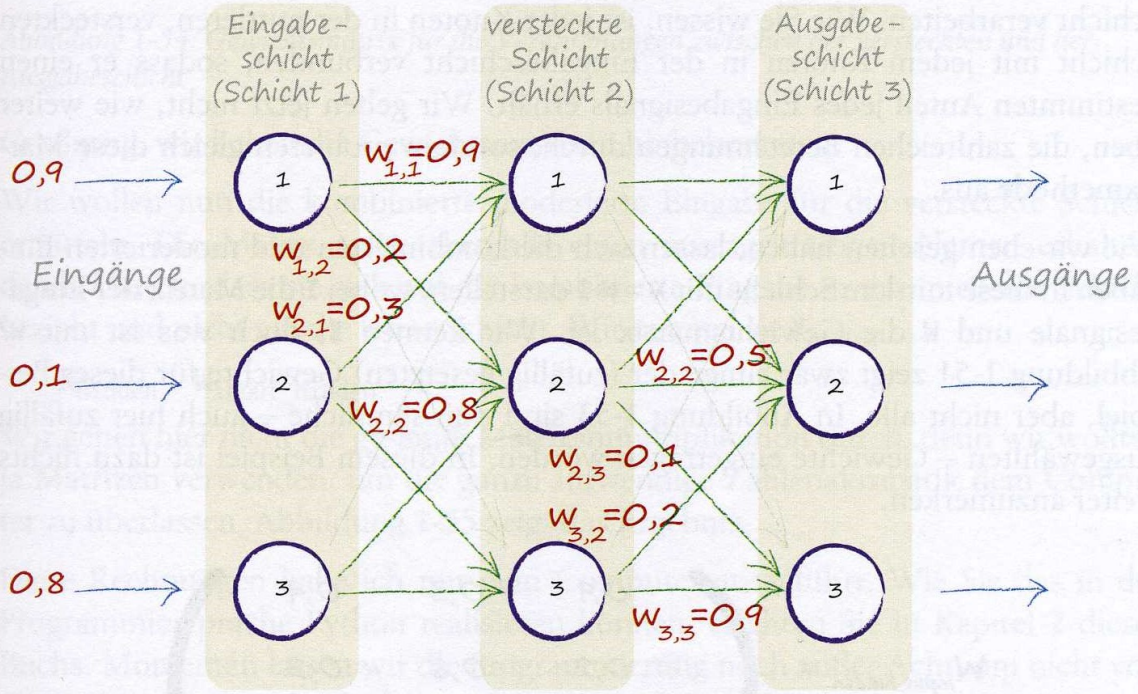


Abbildung 1-51: Beispiel für ein neuronales Netz mit drei Schichten zu je drei Knoten

Erst mal führen wir hier einige der häufig verwendeten Begriffe ein. Wie wir bereits wissen, ist die erste Schicht die *Eingabeschicht* (engl. Input Layer). Ebenfalls

bekannt ist die letzte Schicht, die sogenannte *Ausgabeschicht* (engl. Output Layer). Bei der mittleren Schicht spricht man von der *versteckten Schicht* (engl. Hidden Layer). Das mag geheimnisvoll und dunkel klingen, doch leider gibt es dafür keinen geheimnisvollen dunklen Grund. Der Name hat sich einfach gehalten, weil die Ausgänge der mittleren Schicht nicht unbedingt als Ausgänge in Erscheinung treten, also »verborgen« oder »versteckt« sind. Das ist zwar wenig überzeugend, doch es gibt tatsächlich keine bessere Begründung für den Namen.

Gehen wir nun das Beispielnetz durch, das in Abbildung 1-51 dargestellt ist. Es sind drei Eingaben mit 0,9, 0,1 und 0,8 zu sehen. Die Eingabematrix `I` sieht also wie in Abbildung 1-52 aus.

$$I = \begin{pmatrix} 0{,}9 \\ 0{,}1 \\ 0{,}8 \end{pmatrix}$$

Abbildung 1-52: Eingabematrix des Beispielnetzes

Das war leicht. Damit ist die erste Eingabeschicht fertig, denn mehr hat eine Eingabeschicht nicht zu tun – sie repräsentiert lediglich die Eingabe.

Als Nächstes kommt die mittlere oder versteckte Schicht. Hier müssen wir die kombinierten (und moderierten) Signale zu jedem Knoten in dieser mittleren Schicht verarbeiten. Wie Sie wissen, ist jeder Knoten in der mittleren, versteckten Schicht mit jedem Knoten in der Eingabeschicht verbunden, sodass er einen bestimmten Anteil jedes Eingabesignals erhält. Wir gehen jetzt nicht, wie weiter oben, die zahlreichen Berechnungen durch, sondern probieren gleich diese Matrixmethode aus.

Wie wir eben gesehen haben, lassen sich die kombinierten und moderierten Eingaben in diese mittlere Schicht mit $X = W \cdot I$ darstellen, wobei `I` die Matrix der Eingabesignale und `W` die Gewichtsmatrix ist. Wir kennen `I`, doch was ist mit `W`? Abbildung 1-51 zeigt zwar einige der (zufällig gesetzten) Gewichte für dieses Beispiel, aber nicht alle. In Abbildung 1-53 sind nun sämtliche – auch hier zufällig ausgewählten – Gewichte eingetragen worden. In diesem Beispiel ist dazu nichts weiter anzumerken.

$$W_{input_hidden} = \begin{pmatrix} 0{,}9 & 0{,}3 & 0{,}4 \\ 0{,}2 & 0{,}8 & 0{,}2 \\ 0{,}1 & 0{,}5 & 0{,}6 \end{pmatrix}$$

Abbildung 1-53: Gewichtsmatrix für die Verknüpfungen zwischen Eingabeschicht und versteckter Schicht

Wie Sie in der Matrix ablesen können, hat das Gewicht $w_{1,1}$ zwischen dem ersten Eingabeknoten und dem ersten Knoten der mittleren, versteckten Schicht den Wert 0,9, genau wie es die Netzdarstellung in Abbildung 1-51 zeigt. Ebenso können Sie das Gewicht für die Verknüpfung zwischen dem zweiten Eingabeknoten und dem zweiten Knoten der versteckten Schicht mit $w_{2,2} = 0{,}8$ ablesen, wie ebenfalls im Netzdiagramm zu sehen ist. Das Netzdiagramm in Abbildung 1-51 zeigt aber nicht das Gewicht der Verknüpfung zwischen dem dritten Eingabeknoten und dem ersten versteckten Knoten, das wir hier mit $w_{3,1} = 0{,}4$ gewählt haben.

Moment mal! Warum haben wir diese Matrix **W** mit »input_hidden« gekennzeichnet? Weil $\mathbf{W}_{input_hidden}$ die Gewichte zwischen der Eingabeschicht und der versteckten Schicht enthält. Für die Verknüpfungen zwischen der versteckten und der Ausgabeschicht brauchen wir eine andere Matrix, die wir $\mathbf{W}_{hidden_output}$ nennen.

Abbildung 1-54 zeigt diese zweite Matrix $\mathbf{W}_{hidden_output}$ mit den eingetragenen Gewichten. Auch hier sollten Sie in der Lage sein, zum Beispiel die Verknüpfung zwischen dem dritten versteckten Knoten und dem dritten Ausgabeknoten als $w_{3,3} = 0{,}9$ zu erkennen.

$$\mathbf{W}_{hidden_output} = \begin{pmatrix} 0{,}3 & 0{,}7 & 0{,}5 \\ 0{,}6 & 0{,}5 & 0{,}2 \\ 0{,}8 & 0{,}1 & 0{,}9 \end{pmatrix}$$

Abbildung 1-54: Gewichtsmatrix für die Verknüpfungen zwischen der versteckten und der Ausgabeschicht

Großartig, wir haben die Gewichtsmatrizen hinbekommen.

Wir wollen nun die kombinierte moderierte Eingabe für die versteckte Schicht ermitteln. Die Matrix sollte ebenfalls einen aussagekräftigen Namen erhalten, damit klar ist, dass sie sich auf die kombinierte Eingabe in der mittleren Schicht bezieht und nicht auf die letzte Schicht. Wir nennen sie $\mathbf{X}_{hidden}$.

$$\mathbf{X}_{hidden} = \mathbf{W}_{input_hidden} \cdot I$$

Wir gehen hier nicht die gesamte Matrizenmultiplikation durch, denn wir wollten ja Matrizen verwenden, um die ganze aufwendige Zahlenakrobatik dem Computer zu überlassen. Abbildung 1-55 zeigt das Ergebnis.

Diese Rechnungen habe ich mit dem Computer ausgeführt. Wie Sie das in der Programmiersprache Python realisieren können, erfahren Sie in Kapitel 2 dieses Buchs. Momentan lassen wir die Programmierung noch außer Acht, um nicht von Computersoftware abgelenkt zu werden.

$$X_{hidden} = \begin{pmatrix} 0{,}9 & 0{,}3 & 0{,}4 \\ 0{,}2 & 0{,}8 & 0{,}2 \\ 0{,}1 & 0{,}5 & 0{,}6 \end{pmatrix} \cdot \begin{pmatrix} 0{,}9 \\ 0{,}1 \\ 0{,}8 \end{pmatrix}$$

$$X_{hidden} = \begin{pmatrix} 1{,}16 \\ 0{,}42 \\ 0{,}62 \end{pmatrix}$$

Abbildung 1-55: Ergebnis der Matrizenmultiplikation für die Eingänge der versteckten Schicht

Wir haben also die kombinierten moderierten Eingaben in die mittlere (versteckte) Schicht mit den Werten 1,16, 0,42 und 0,62. Für die aufwendigen Berechnungen haben wir uns auf Matrizen gestützt. Darauf können wir stolz sein!

Wir visualisieren nun diese kombinierten moderierten Eingaben in die zweite versteckte Schicht (siehe Abbildung 1-56).

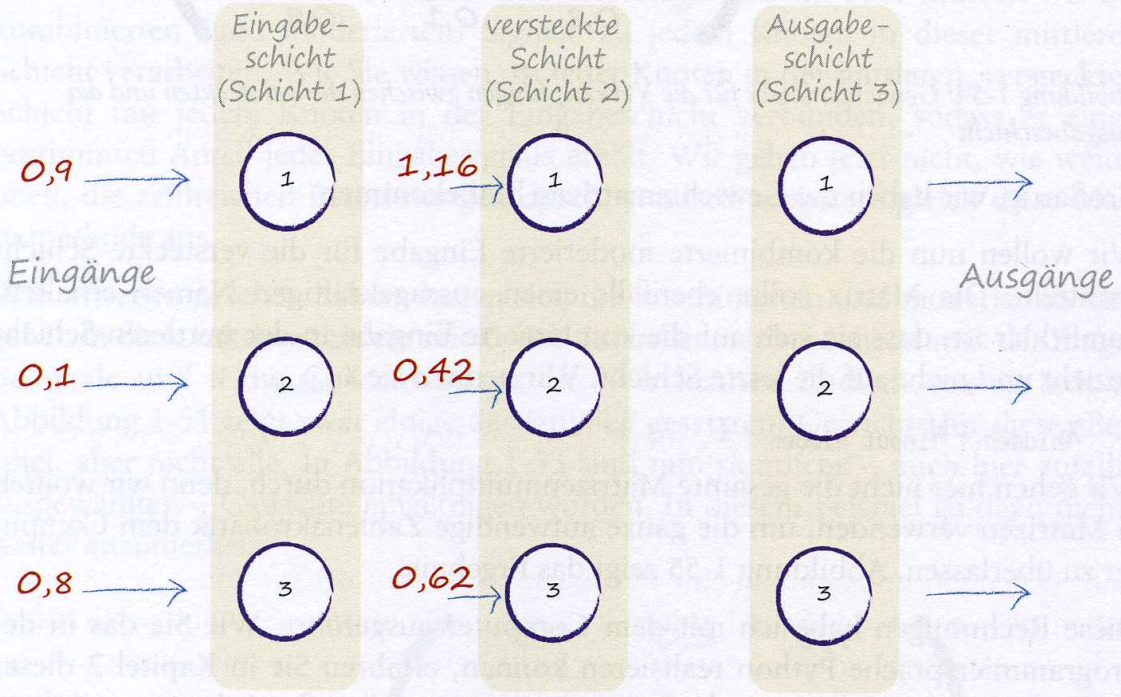


Abbildung 1-56: Die berechneten Eingaben für die versteckte (mittlere) Schicht des Beispielnetzes

So weit, so gut, doch es gibt noch mehr zu tun. Wie Sie bereits wissen, wenden diese Knoten eine Sigmoid-Aktivierungsfunktion an, damit die Antwort auf das Eingangssignal dem natürlichen Vorbild näher kommt. Also tun wir das:

$$\mathbf{O}_{hidden} = \text{sigmoid}(\ \mathbf{X}_{hidden}\ )$$

Die Sigmoidfunktion wird auf jedes Element in X_{hidden} angewendet, um die Matrix mit den Ausgängen der mittleren (versteckten) Schicht zu erzeugen (siehe Abbildung 1-57).

$$O_{hidden} = \text{sigmoid}\begin{pmatrix} 1{,}16 \\ 0{,}42 \\ 0{,}62 \end{pmatrix}$$

$$O_{hidden} = \begin{pmatrix} 0{,}761 \\ 0{,}603 \\ 0{,}650 \end{pmatrix}$$

Abbildung 1-57: Anwendung der Sigmoidfunktion auf die Elemente der versteckten Schicht

Zur Kontrolle überprüfen wir das erste Element. Die Sigmoidfunktion lautet:

$$y = \frac{1}{(1+e^{-x})}$$

Wenn also x = 1,16 ist, wird $e^{-1,16}$ zu 0,3135. Damit ergibt sich y = 1 / (1+ 0,3135) = 0,761.

Des Weiteren ist zu sehen, dass alle Werte zwischen 0 und 1 liegen, weil diese Sigmoidfunktion keine Werte außerhalb dieses Bereichs erzeugt. Sehen Sie sich dazu noch einmal den Graphen der logistischen Funktion aus Abbildung 1-31 an.

Puh! Legen wir noch einmal eine Pause ein und sehen wir uns an, was wir getan haben. Wir haben das Signal ermittelt, wenn es die mittlere Schicht passiert hat, d. h. die Ausgänge von der mittleren Schicht berechnet. Um vollkommene Klarheit zu schaffen, stellen wir die Frage: Welche sind die kombinierten Eingänge in die mittlere Schicht, auf die dann die Aktivierungsfunktion angewendet wird? In Abbildung 1-58 ist das Beispielnetz mit diesen neuen Informationen dargestellt.

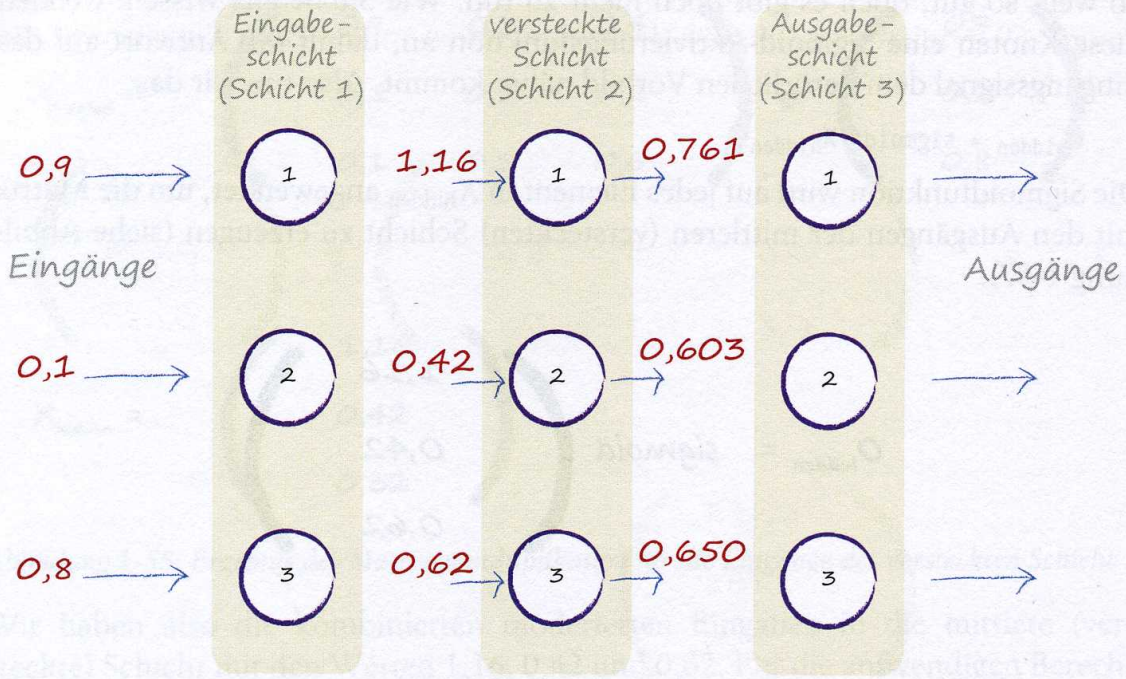


Abbildung 1-58: Das Beispielnetz mit den Werten an den Ein- und Ausgängen der mittleren Schicht

Wenn dies ein neuronales Netz mit zwei Schichten gewesen wäre, würden wir jetzt aufhören, weil die erhaltenen Werte die Ausgänge von der zweiten Schicht sind. Wir sind aber noch nicht fertig, weil es eine dritte Schicht gibt.

Wie ermitteln wir das Signal durch die dritte Schicht? Es ist der gleiche Ansatz wie bei der zweiten Schicht, ein wirklicher Unterschied besteht nicht. In der dritten Schicht sind Eingangssignale zu verarbeiten, genau wie es in der zweiten Schicht geschehen ist. Des Weiteren gibt es Verknüpfungen mit Gewichten, um diese Signale zu moderieren. Und wir haben wieder eine Aktivierungsfunktion, die das Antwortverhalten der Knoten dem natürlichen Vorbild näherbringt. Wir merken uns also, dass sich jede Schicht unabhängig von der Anzahl der Schichten wie jede andere behandeln lässt – mit Eingabesignalen, die wir kombinieren, Verknüpfungsgewichten, um die Eingabesignale zu moderieren, und einer Aktivierungsfunktion, um die Ausgabe von dieser Schicht zu erzeugen. Dabei spielt es überhaupt keine Rolle, ob wir in der 3. oder 53. oder sogar der 103. Schicht arbeiten – das Konzept ist überall gleich.

Machen wir also weiter und berechnen wir die kombinierte moderierte Eingabe in diese letzte Schicht $\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$, genau wie wir es schon getan haben.

Die Eingaben in diese Schicht sind die Ausgaben der zweiten Schicht, $\mathbf{O}_{hidden}$, die wir eben berechnet haben. Die Gewichte sind den Verknüpfungen zwischen der zweiten und dritten Schicht $\mathbf{W}_{hidden_output}$ zugeordnet und nicht denjenigen, die wir eben zwischen der ersten und zweiten Schicht verwendet haben.

$$\mathbf{X}_{output} = \mathbf{W}_{hidden_output} \cdot \mathbf{O}_{hidden}$$

Wenn wir dies also in der gleichen Weise berechnen, erhalten wir das in Abbildung 1-59 gezeigte Ergebnis für die kombinierten moderierten Eingaben in die letzte Ausgabeschicht.

$$X_{output} = \begin{pmatrix} 0{,}3 & 0{,}7 & 0{,}5 \\ 0{,}6 & 0{,}5 & 0{,}2 \\ 0{,}8 & 0{,}1 & 0{,}9 \end{pmatrix} \cdot \begin{pmatrix} 0{,}761 \\ 0{,}603 \\ 0{,}650 \end{pmatrix}$$

$$X_{output} = \begin{pmatrix} 0{,}975 \\ 0{,}888 \\ 1{,}254 \end{pmatrix}$$

Abbildung 1-59: Ergebnisse für die Ausgabeschicht

Abbildung 1-60 zeigt das aktualisierte neuronale Netz, das das Signal von der anfänglichen Eingabe direkt durch die kombinierten Eingaben bis zur letzten Schicht leitet.

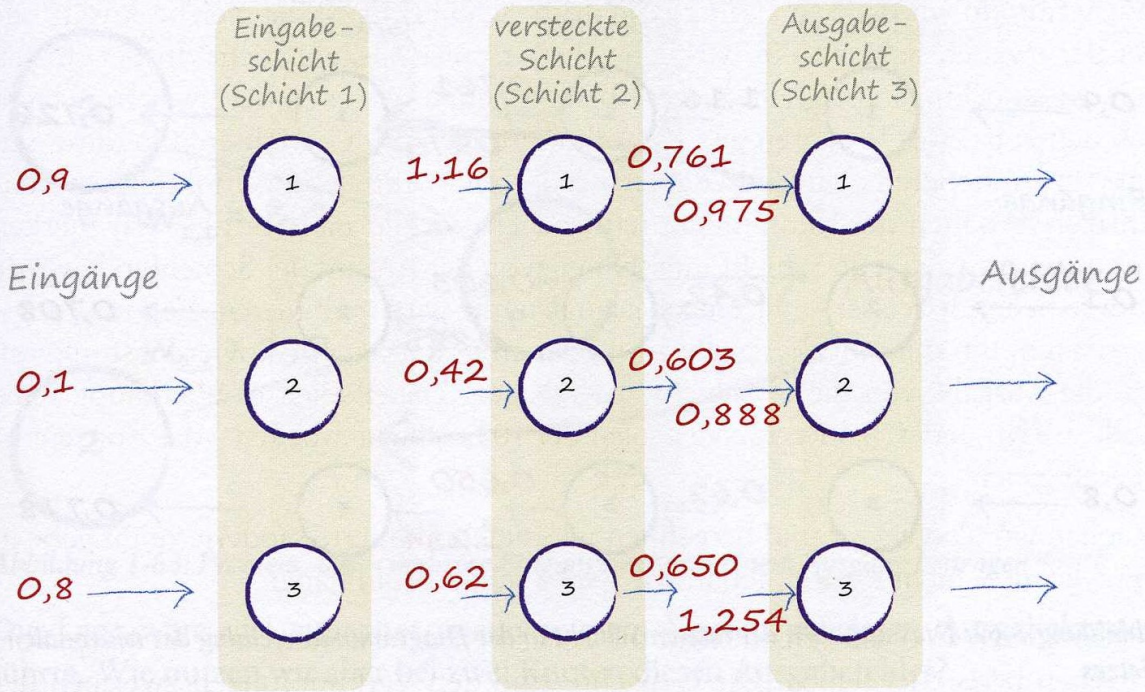


Abbildung 1-60: Das Beispiel für das neuronale Netz mit den aktualisierten Gewichten

Jetzt müssen wir nur noch die Sigmoid-Aktivierungsfunktion anwenden, was ganz einfach ist (siehe Abbildung 1-61).

$$O_{output} = \text{sigmoid}\begin{pmatrix} 0{,}975 \\ 0{,}888 \\ 1{,}254 \end{pmatrix}$$

$$O_{output} = \begin{pmatrix} 0{,}726 \\ 0{,}708 \\ 0{,}778 \end{pmatrix}$$

Abbildung 1-61: Sigmoid-Aktivierungsfunktion, angewendet auf die Knoten der Ausgabeschicht

Das war's! Wir haben die letzten Ausgaben vom neuronalen Netz. Abbildung 1-62 zeigt auch diese in der Diagrammdarstellung.

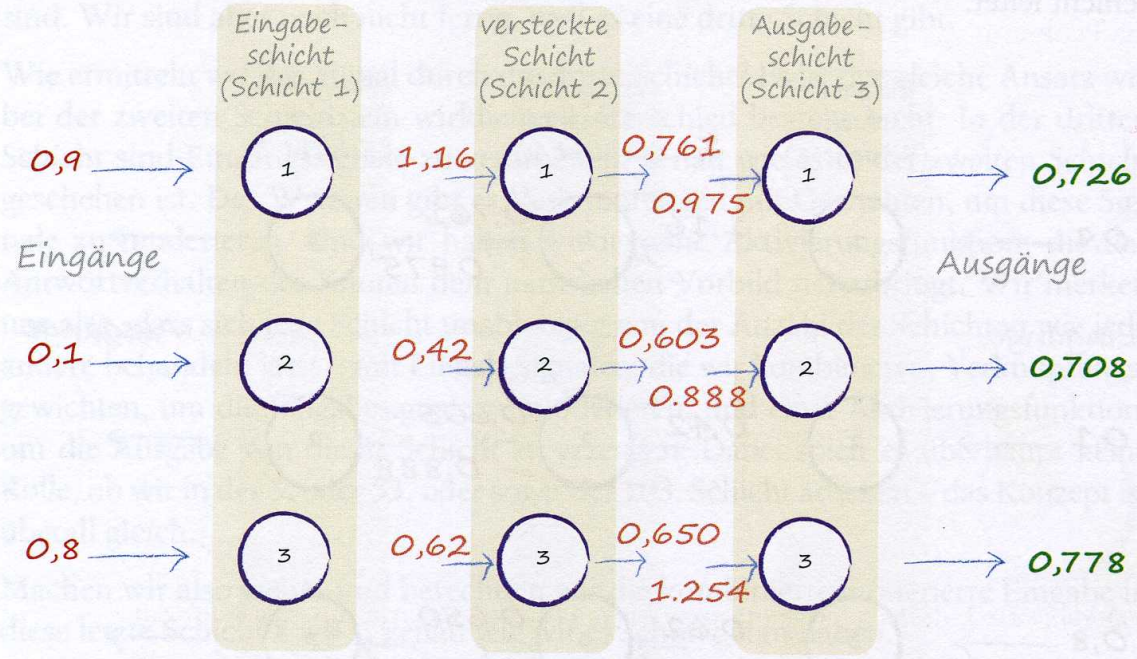


Abbildung 1-62: Die Ausgaben der letzten Schicht in der Diagrammdarstellung des neuronalen Netzes

Die letzten Ausgaben im Beispiel des neuronalen Netzes mit drei Schichten sind also 0,726, 0,708 und 0,778.

Wir haben das Signal von seinem anfänglichen Eintritt in das neuronale Netz über die einzelnen Schichten bis zum Austritt aus der letzten Schicht erfolgreich verfolgt.

Was jetzt?

Im nächsten Schritt vergleichen wir den Ausgang vom neuronalen Netz mit dem Trainingsbeispiel, um einen Fehler zu ermitteln. Anhand dieses Fehlers müssen wir das neuronale Netz selbst verfeinern, sodass es seine Ausgänge verbessert.

Dieser Schritt ist wahrscheinlich am schwierigsten zu verstehen, sodass wir ihn behutsam angehen und die Konzepte parallel dazu veranschaulichen.

Gewichte von mehr als einem Knoten lernen

Weiter oben haben wir einen linearen Klassifizierer verfeinert, indem wir als Parameter den Anstieg der linearen Funktion des Knotens angepasst haben. Gesteuert haben wir diese Verfeinerung anhand des Fehlers, d. h. der Differenz zwischen der vom Knoten erzeugten Antwort und dem, was wir als wahre Antwort kennen. Das hat sich als einfach herausgestellt, weil die Beziehung zwischen dem Fehler und der erforderlichen Anstiegsanpassung sehr einfach zu ermitteln war.

Wie aktualisieren wir nun die Verknüpfungsgewichte, wenn mehr als ein Knoten zu einem Ausgang und seinem Fehler beiträgt? Abbildung 1-63 veranschaulicht das Problem.

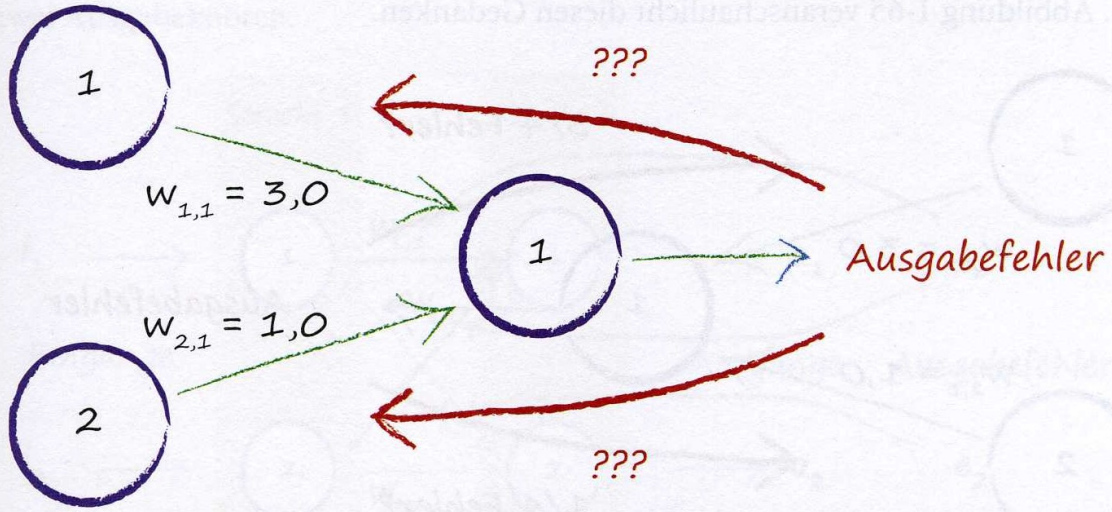


Abbildung 1-63: Problem, wenn mehrere Eingangswerte zu einem Ausgang beitragen

Die Lage wäre viel einfacher, wenn nur ein Knoten zu einem Ausgabeknoten führte. Wie nutzen wir aber bei zwei Knoten diesen Ausgabefehler?

Es ist nicht sinnvoll, mit dem gesamten Fehler nur ein Gewicht zu aktualisieren, weil dann die andere Verknüpfung und ihr Gewicht ignoriert werden. Immerhin sind für diesen Fehler mehrere Verknüpfungen verantwortlich.

Es gibt eine winzige Chance, dass der Fehler nur auf eine einzige Verknüpfung zurückzuführen ist, doch ist diese Chance verschwindend gering. Würden wir ein bereits »korrektes« Gewicht ändern, was zu einem größeren Fehler führte, würde es während der nächsten Iterationen verbessert, sodass nicht alles verloren ist.

Man könnte beispielsweise den Fehler unter allen beitragenden Knoten gleichmäßig aufteilen, wie Abbildung 1-64 zeigt.

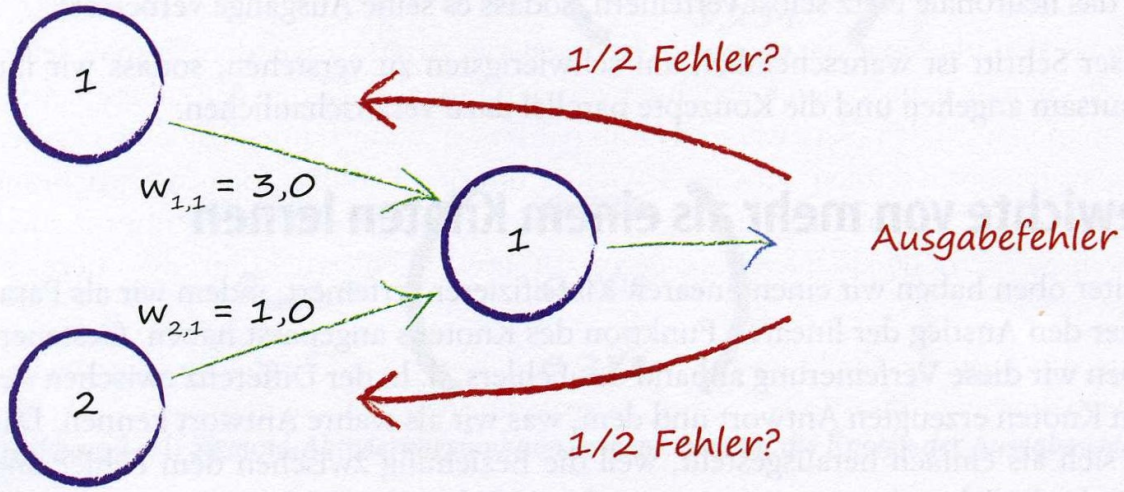


Abbildung 1-64: Den Fehler unter allen beitragenden Knoten gleichmäßig aufteilen

Eine andere Idee ist, den Fehler zwar aufzuteilen, aber nicht gleichmäßig. Stattdessen ordnen wir den Verbindungen, die größere Verknüpfungsgewichte haben, einen größeren Anteil des Fehlers zu. Warum? Weil sie mehr zum Fehler beitragen. Abbildung 1-65 veranschaulicht diesen Gedanken.

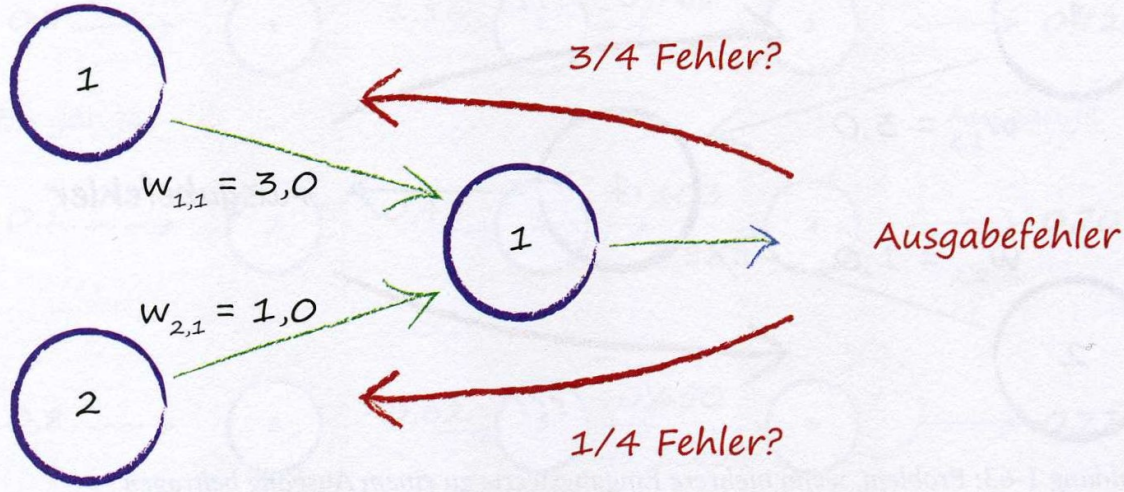


Abbildung 1-65: Den Fehler unter allen beitragenden Knoten je nach Verknüpfungsgewicht aufteilen

Im Beispiel von Abbildung 1-65 tragen zwei Knoten ein Signal zum Ausgabeknoten bei. Die Verknüpfungsgewichte sind 3,0 und 1,0. Wenn wir den Fehler proportio-

nal zu diesen Gewichten aufteilen, gehen 3/4 des Ausgabefehlers zum ersten, größeren Gewicht und 1/4 des Fehlers zum zweiten, kleineren Gewicht.

Die gleiche Idee können wir auf beliebig viele Knoten erweitern. Hätten wir 100 Knoten mit einem Ausgabeknoten verbunden, würden wir den Fehler auf die 100 Verbindungen zu diesem Ausgabeknoten aufteilen, und zwar anteilig zum Beitrag, den jede Verknüpfung zum Fehler beisteuert, was sich aus der Größe der Verknüpfungsgewichte ergibt.

Es zeigt sich, dass wir die Gewichte auf zweierlei Arten verwenden. Erstens leiten wir die Signale je nach den Gewichten von den Eingabe- zu den Ausgabeschichten in einem neuronalen Netz weiter. Daran haben wir zuvor schon ausgiebig gearbeitet. Zweitens verwenden wir die Gewichte, um den Fehler in der Gegenrichtung von der Ausgabe zurück in das Netz zu leiten. Es dürfte Sie nicht überraschen, dass diese Methode als *Backpropagation* (dt. etwa »Fehlerrückführung«) bezeichnet wird.

Hätte die Ausgabeschicht zwei Knoten, würden wir das Gleiche für den zweiten Ausgabeknoten tun. Dieser zweite Ausgabeknoten wird seinen eigenen Fehler haben, der analog auf die Verbindungsgewichte aufgeteilt wird. Das sehen wir uns als Nächstes an.

Fehler von mehreren Ausgabeknoten zurückführen

Abbildung 1-66 zeigt ein einfaches Netz mit zwei Eingabeknoten, aber auch mit zwei Ausgabeknoten.

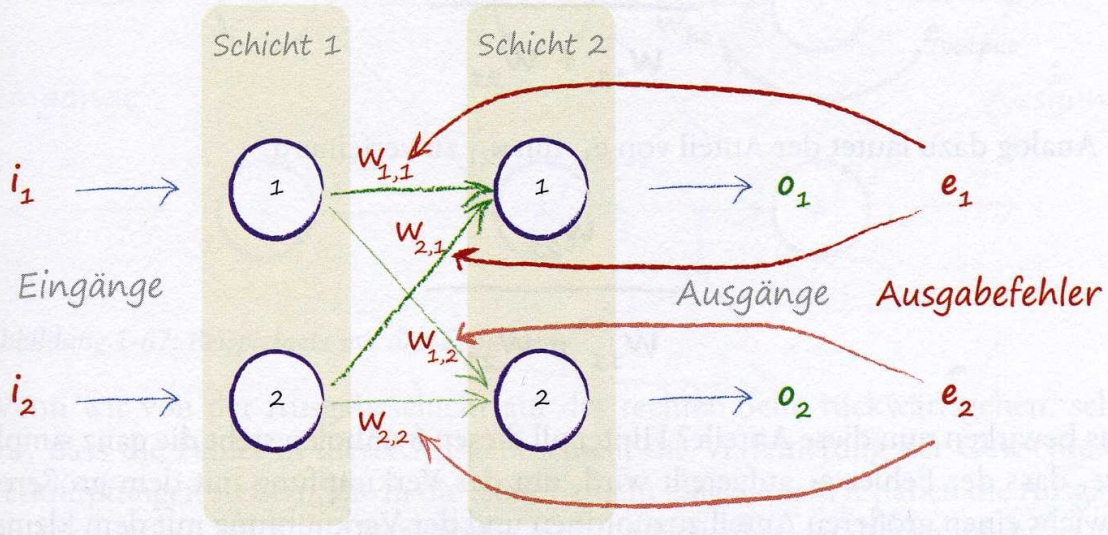


Abbildung 1-66: Beispiel für ein einfaches neuronales Netz mit zwei Eingabeknoten und zwei Ausgabeknoten

Beide Ausgabeknoten können einen Fehler haben – das ist in der Tat sogar äußerst wahrscheinlich, wenn das Netz noch nicht trainiert wurde. Man kann sehen, dass

Es gibt eine winzige Chance, dass der Fehler nur auf eine einzige Verknüpfung zurückzuführen ist, doch ist diese Chance verschwindend gering. Würden wir ein bereits »korrektes« Gewicht ändern, was zu einem größeren Fehler führte, würde es während der nächsten Iterationen verbessert, sodass nicht alles verloren ist.

Man könnte beispielsweise den Fehler unter allen beitragenden Knoten gleichmäßig aufteilen, wie Abbildung 1-64 zeigt.

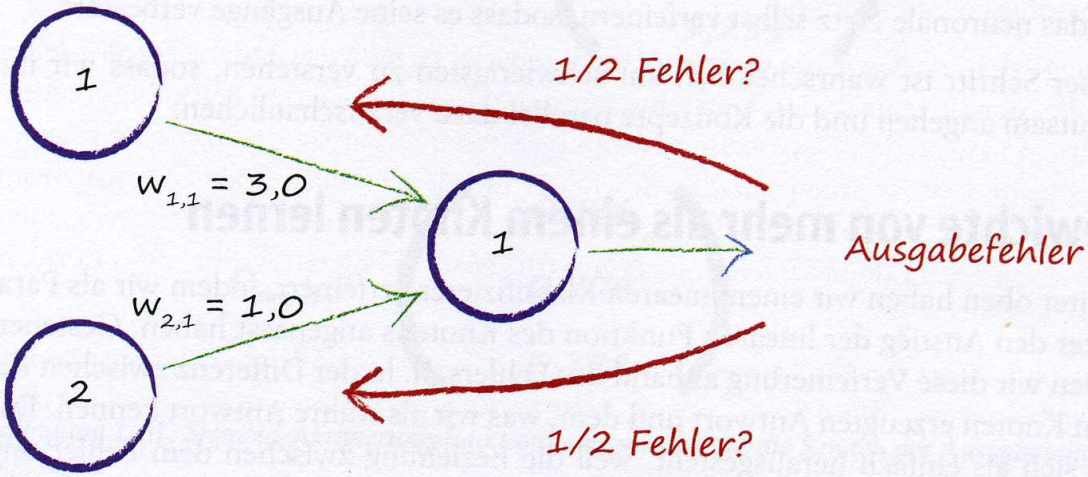


Abbildung 1-64: Den Fehler unter allen beitragenden Knoten gleichmäßig aufteilen

Eine andere Idee ist, den Fehler zwar aufzuteilen, aber nicht gleichmäßig. Stattdessen ordnen wir den Verbindungen, die größere Verknüpfungsgewichte haben, einen größeren Anteil des Fehlers zu. Warum? Weil sie mehr zum Fehler beitragen. Abbildung 1-65 veranschaulicht diesen Gedanken.

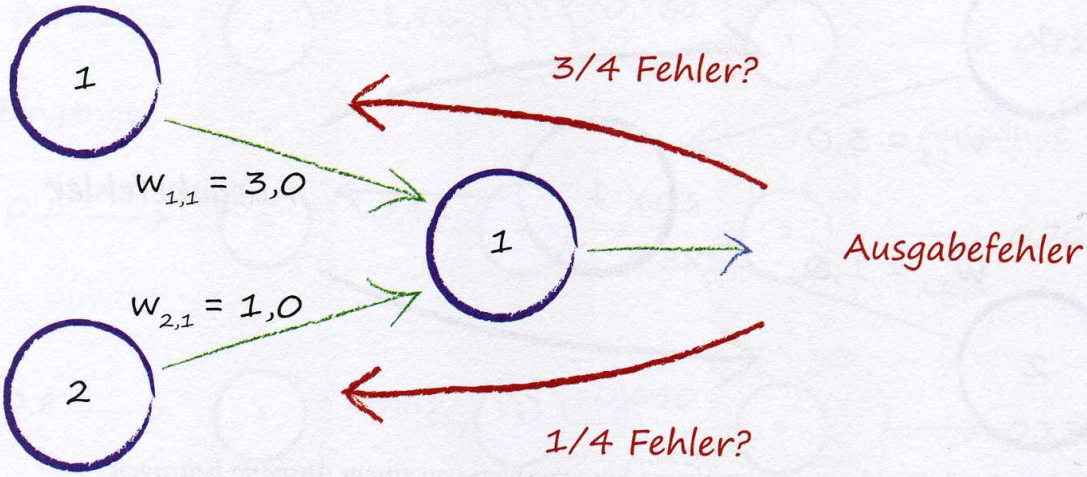


Abbildung 1-65: Den Fehler unter allen beitragenden Knoten je nach Verknüpfungsgewicht aufteilen

Im Beispiel von Abbildung 1-65 tragen zwei Knoten ein Signal zum Ausgabeknoten bei. Die Verknüpfungsgewichte sind 3,0 und 1,0. Wenn wir den Fehler proportio-

nal zu diesen Gewichten aufteilen, gehen 3/4 des Ausgabefehlers zum ersten, größeren Gewicht und 1/4 des Fehlers zum zweiten, kleineren Gewicht.

Die gleiche Idee können wir auf beliebig viele Knoten erweitern. Hätten wir 100 Knoten mit einem Ausgabeknoten verbunden, würden wir den Fehler auf die 100 Verbindungen zu diesem Ausgabeknoten aufteilen, und zwar anteilig zum Beitrag, den jede Verknüpfung zum Fehler beisteuert, was sich aus der Größe der Verknüpfungsgewichte ergibt.

Es zeigt sich, dass wir die Gewichte auf zweierlei Arten verwenden. Erstens leiten wir die Signale je nach den Gewichten von den Eingabe- zu den Ausgabeschichten in einem neuronalen Netz weiter. Daran haben wir zuvor schon ausgiebig gearbeitet. Zweitens verwenden wir die Gewichte, um den Fehler in der Gegenrichtung von der Ausgabe zurück in das Netz zu leiten. Es dürfte Sie nicht überraschen, dass diese Methode als *Backpropagation* (dt. etwa »Fehlerrückführung«) bezeichnet wird.

Hätte die Ausgabeschicht zwei Knoten, würden wir das Gleiche für den zweiten Ausgabeknoten tun. Dieser zweite Ausgabeknoten wird seinen eigenen Fehler haben, der analog auf die Verbindungsgewichte aufgeteilt wird. Das sehen wir uns als Nächstes an.

Fehler von mehreren Ausgabeknoten zurückführen

Abbildung 1-66 zeigt ein einfaches Netz mit zwei Eingabeknoten, aber auch mit zwei Ausgabeknoten.

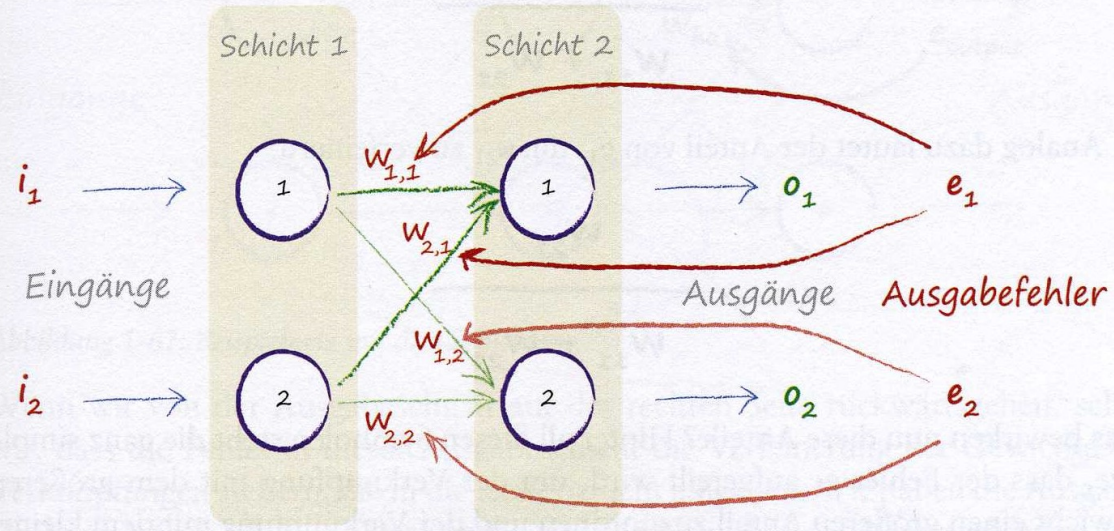


Abbildung 1-66: Beispiel für ein einfaches neuronales Netz mit zwei Eingabeknoten und zwei Ausgabeknoten

Beide Ausgabeknoten können einen Fehler haben – das ist in der Tat sogar äußerst wahrscheinlich, wenn das Netz noch nicht trainiert wurde. Man kann sehen, dass

diese beiden Fehler die Verfeinerung der internen Verknüpfungsgewichte im Netz anregen müssen. Wir können den gleichen Ansatz wie zuvor verwenden, wobei wir den Fehler eines Ausgabeknotens unter den beisteuernden Verknüpfungen proportional zu ihren Gewichten aufteilen.

Durch den Umstand, dass wir mehr als einen Ausgabeknoten haben, hat sich praktisch nichts geändert. Wir wiederholen einfach für den zweiten Ausgabeknoten, was wir bereits für den ersten getan haben. Warum ist das so einfach? Weil die Verknüpfungen zu einem Ausgabeknoten nicht von den Verknüpfungen zu einem anderen Ausgabeknoten abhängen. Beide Gruppen von Verknüpfungen sind unabhängig voneinander.

Sehen Sie sich noch einmal Abbildung 1-66 an. Der Fehler am ersten Ausgabeknoten ist mit e_1 bezeichnet. Wie bereits erwähnt, ist das die Differenz zwischen der Sollausgabe, die von den Trainingsdaten t_1 vorgegeben wird, und der tatsächlichen Ausgabe o_1. Das heißt, $e_1 = (t_1 - o_1)$. Der Fehler beim zweiten Ausgabeknoten ist mit e_2 bezeichnet.

Wie aus dem Diagramm in Abbildung 1-66 hervorgeht, wird der Fehler e_1 anteilig auf die Verknüpfungen mit den Gewichten w_{11} und w_{21} aufgeteilt. Analog dazu wird e_2 proportional zu den Gewichten w_{12} und w_{22} aufgeteilt.

Wir schreiben nun auf, wie diese Teilbeträge aussehen, damit keinerlei Zweifel aufkommen. Der Fehler e_1 steuert die Verfeinerung der beiden Gewichte w_{11} und w_{21}. Er wird so geteilt, dass der Anteil von e_1 für die Aktualisierung von w_{11} gleich

$$\frac{w_{11}}{w_{11} + w_{21}}$$

ist. Analog dazu lautet der Anteil von e_1, um w_{21} zu verfeinern:

$$\frac{w_{21}}{w_{11} + w_{21}}$$

Was bewirken nun diese Anteile? Hinter all diesen Symbolen steht die ganz simple Idee, dass der Fehler e_1 aufgeteilt wird, um der Verknüpfung mit dem größeren Gewicht einen größeren Anteil zuzuordnen und der Verknüpfung mit dem kleineren Gewicht einen kleineren Anteil.

Wenn w_{11} doppelt so groß ist wie w_{21} – beispielsweise $w_{11} = 6$ und $w_{21} = 3$ –, ist der Anteil von e_1 für die Aktualisierung von w_{11} gleich $6/(6 + 3) = 6/9 = 2/3$. Damit bleibt 1/3 von e_1 für das andere kleinere Gewicht w_{21}, was sich mit dem Ausdruck $3/(6 + 3) = 3/9$, tatsächlich also 1/3, bestätigen lässt.

Wären die Gewichte gleich groß, machen beide Anteile erwartungsgemäß jeweils die Hälfte aus. Auch das wollen wir kontrollieren. Wenn wir $w_{11} = 4$ und $w_{21} = 4$ annehmen, ist der Anteil 4/(4 + 4) = 4/8 = 1/2 für beide Fälle.

An dieser Stelle wollen wir einen Schritt zurückgehen und uns mit etwas Abstand ansehen, was wir getan haben. Bekannt war, dass wir den Fehler verwenden mussten, um die Verfeinerung bestimmter Parameter innerhalb des Netzes zu steuern, in diesem Fall die Verknüpfungsgewichte. Wie dabei vorzugehen ist, haben wir eben bei den Verknüpfungsgewichten gesehen, die die Signale zur letzten, zur Ausgabeschicht eines neuronalen Netzes moderieren. Außerdem haben wir erfahren, dass es kaum komplizierter wird, wenn es mehrere Ausgabeknoten gibt. Wir tun einfach das Gleiche für jeden Ausgabeknoten. Großartig!

Als Nächstes ist die Frage zu beantworten, was bei mehr als zwei Schichten passiert. Wie aktualisieren wir die Verknüpfungsgewichte in den Schichten, die von der Ausgabeschicht weiter entfernt sind?

Fehler auf mehrere Schichten zurückführen

Abbildung 1-67 zeigt ein einfaches neuronales Netz mit drei Schichten – einer Eingabeschicht, einer versteckten Schicht und der Ausgabeschicht.

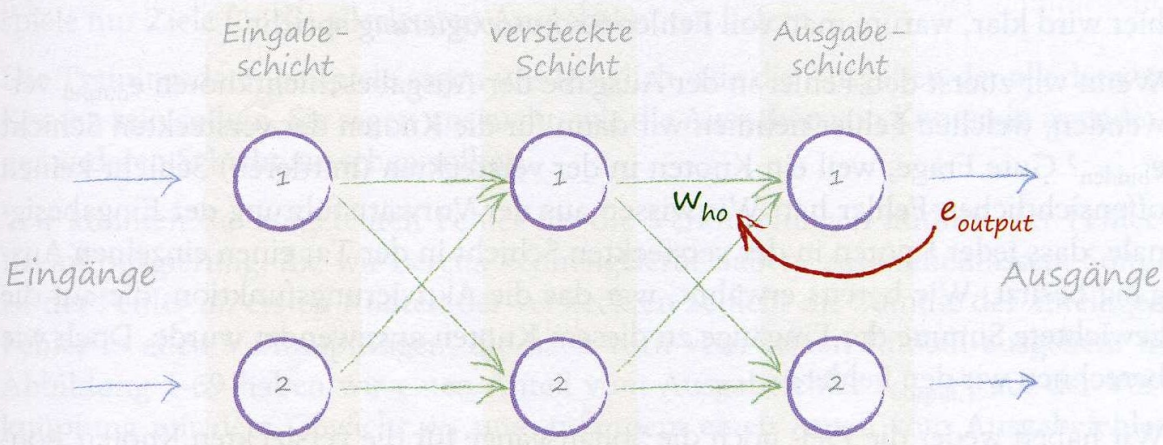


Abbildung 1-67: Beispielnetz mit drei Schichten

Wenn wir von der Ausgabeschicht auf der rechten Seite rückwärtsgehen, sehen wir, dass die Fehler in dieser Ausgabeschicht die Verfeinerung der Gewichte der Verknüpfungen steuern, die in die letzte Schicht eingehen. Wir haben die Ausgabefehler generisch als e_{output} bezeichnet und die Gewichte der Verknüpfungen zwischen der versteckten und der Ausgabeschicht als w_{ho}. Außerdem haben wir die konkreten Fehleranteile ermittelt, die jeder Verknüpfung zugeordnet werden, indem wir die Fehler proportional zur Größe der Gewichte selbst aufgeteilt haben. Abbildung 1-68 veranschaulicht, was wir für die neu hinzugekommene Schicht

unternehmen müssen. Wir nehmen einfach diejenigen Fehler, die den Ausgabeknoten der versteckten Schicht e_{hidden} zugeordnet sind, und teilen diese erneut proportional über die vorhergehenden Verknüpfungen zwischen der Eingabeschicht und der versteckten Schicht w_{ih} auf.

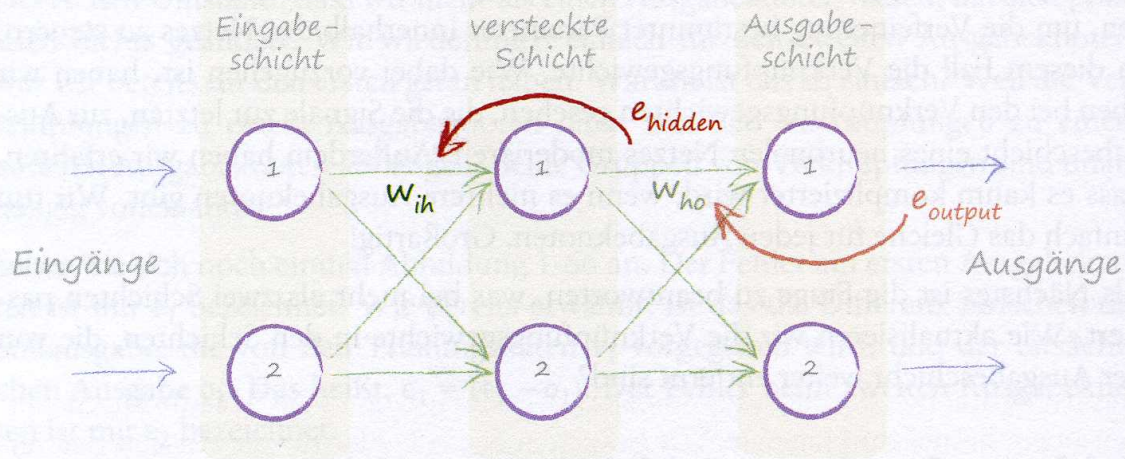


Abbildung 1-68: Aufteilung des Fehlers bis zurück zur versteckten Schicht

Besteht das neuronale Netz aus noch mehr Schichten, wenden wir das gleiche Konzept wiederholt auf jede Schicht an, wobei wir von der Ausgabeschicht aus rückwärtsgehen. Die Rückführung der Fehlerinformationen ist einleuchtend. Auch hier wird klar, warum man von Fehler-*Backpropagierung* spricht.

Wenn wir zuerst den Fehler in der Ausgabe der Ausgabeschichtknoten e_{output} verwenden, welchen Fehler nehmen wir dann für die Knoten der versteckten Schicht e_{hidden}? Gute Frage, weil ein Knoten in der versteckten (mittleren) Schicht keinen offensichtlichen Fehler hat. Wir wissen aus der Vorwärtsführung der Eingabesignale, dass jeder Knoten in der versteckten Schicht in der Tat einen einzelnen Ausgang besitzt. Wie bereits erwähnt, war das die Aktivierungsfunktion, die auf die gewichtete Summe der Eingänge zu diesem Knoten angewendet wurde. Doch wie berechnen wir den Fehler?

Wir haben weder die Ziel- noch die Sollausgänge für die versteckten Knoten, sondern nur die Zielwerte für die Knoten der (letzten) Ausgabeschicht, und diese stammen von den Trainingsbeispielen. Sehen Sie sich noch einmal Abbildung 1-68 an, um sich inspirieren zu lassen! Vom ersten Knoten in der versteckten Schicht gehen zwei Verknüpfungen aus, die den Knoten mit den beiden Knoten der Ausgabeschicht verbinden. Wir wissen, dass wir den Ausgabefehler entlang jeder dieser Verknüpfungen aufteilen können, wie wir es bereits getan haben. Das heißt, wir haben eine Art Fehler für jede der beiden Verknüpfungen, die aus diesem Knoten der mittleren Schicht entspringen. Wir könnten diese beiden Verknüpfungsfehler kombinieren, um den Fehler für diesen Knoten als eine zweitbeste Annäherung zu bilden, weil wir praktisch keinen Zielwert für den Knoten der mittleren Schicht haben. Abbildung 1-69 veranschaulicht diese Idee.

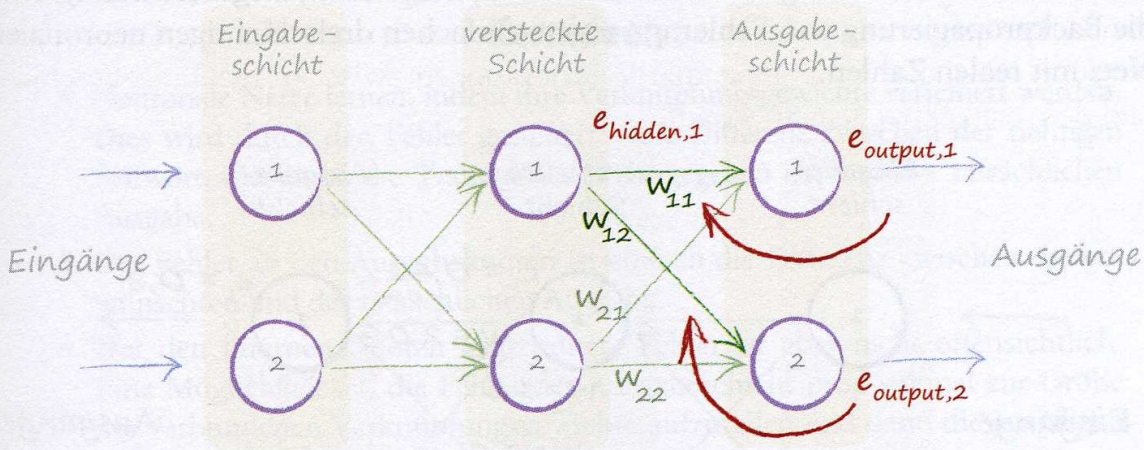


Abbildung 1-69: Aus den Fehlern der Ausgabeknoten die Fehler der Knoten in der versteckten Schicht bilden

Hier ist deutlicher zu sehen, was passiert. Doch wir wollen die Abläufe zur Sicherheit noch einmal durchgehen. Wir brauchen einen Fehler für die Knoten der versteckten Schicht, um mit ihm die Gewichte in der vorhergehenden Schicht aktualisieren zu können. Diese Fehler nennen wir e_{hidden}. Doch wir haben keine offensichtliche Antwort darauf, was die Fehler tatsächlich sind. Wir können nicht sagen, dass der Fehler die Differenz zwischen dem gewünschten Zielwert von diesen Knoten und den tatsächlichen Ausgaben ist, weil unsere Trainingsdatenbeispiele nur Ziele für die allerletzten Ausgabeknoten liefern.

Die Trainingsdatenbeispiele sagen uns lediglich, wie die Ausgaben der allerletzten Knoten sein sollten. Sie sagen uns nicht, wie die Ausgaben von Knoten in irgendeiner anderen Schicht aussehen sollten.

Wir könnten die aufgeteilten Fehler für die Verknüpfungen mithilfe der Fehler-Backpropagierung, die wir bereits kennengelernt haben, zusammenfassen. Somit ist der Fehler im ersten Knoten der versteckten Schicht die Summe der anteiligen Fehler in allen Verknüpfungen, die nach vorn vom selben Knoten ausgehen. In Abbildung 1-69 haben wir einen Anteil vom Ausgabefehler $e_{output,1}$ auf der Verknüpfung mit dem Gewicht w_{11} und außerdem einen Anteil vom Ausgabefehler $e_{output,2}$ vom zweiten Ausgabeknoten auf der Verknüpfung mit dem Gewicht w_{12}.

Wir notieren also:

$$e_{hidden,1} = \text{Summe der aufgeteilten Fehler auf den Verknüpfungen } w_{11} \text{ und } w_{12}$$

$$= e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

Da es hilfreich ist, diese ganze Theorie in Aktion zu sehen, zeigt Abbildung 1-70 die Backpropagierung von Fehlern in einem einfachen dreischichtigen neuronalen Netz mit realen Zahlen.

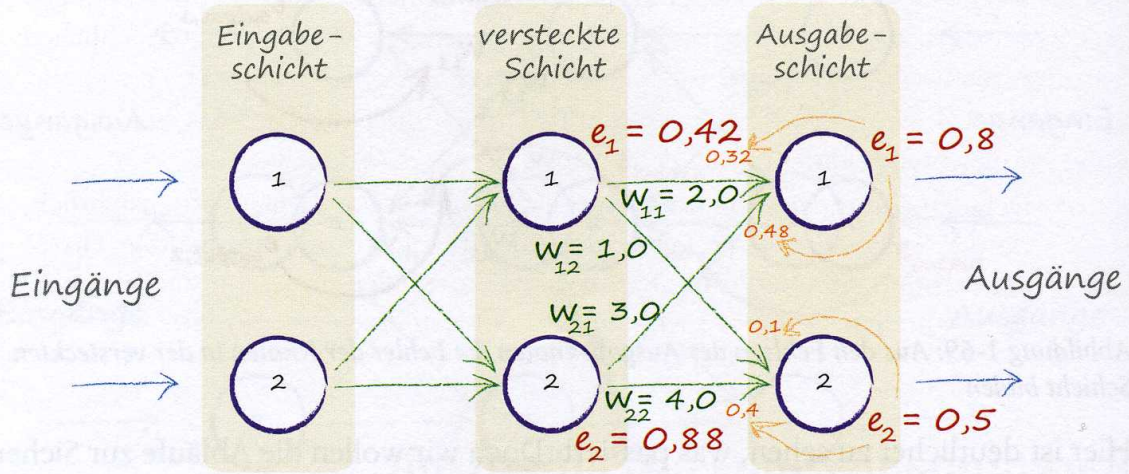


Abbildung 1-70: Das neuronale Beispielnetz mit Fehler-Backpropagierung bis zur verdeckten Schicht

Verfolgen wir einen Fehler zurück. Der Fehler 0,5 beim zweiten Knoten der Ausgabeschicht wird mit 0,1 und 0,4 proportional auf die beiden Verknüpfungen mit den Gewichten 1,0 und 4,0 aufgeteilt. Außerdem zeigt Abbildung 1-70, dass der zusammengefasste Fehler am zweiten Knoten der versteckten Schicht die Summe der verbundenen aufgeteilten Fehler – hier 0,48 und 0,4 – ist, was 0,88 ergibt.

Abbildung 1-71 stellt die gleiche Idee dar, angewandt auf die vorhergehende Schicht, um den Fehler noch weiter zurückzupropagieren.

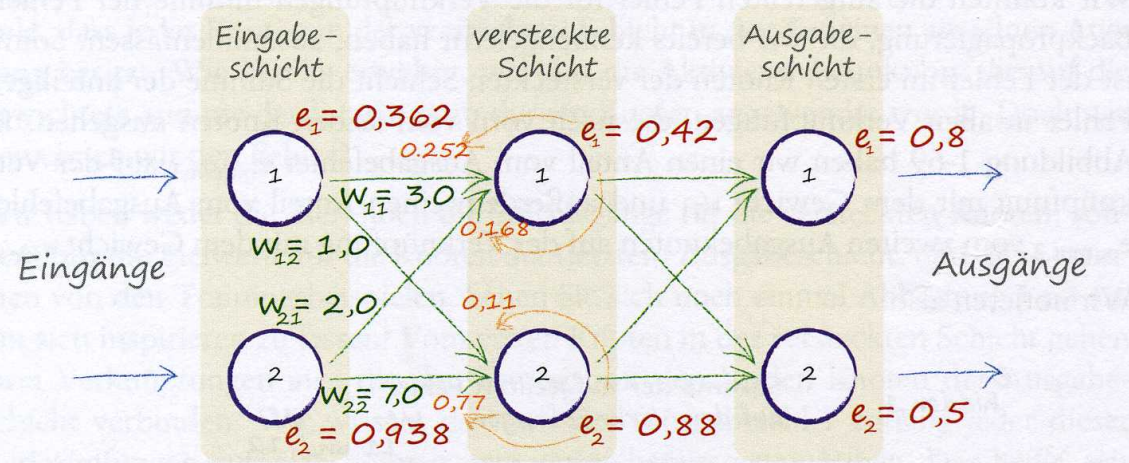


Abbildung 1-71: Das neuronale Beispielnetz mit Fehler-Backpropagierung bis zur Eingabeschicht

Kernideen

- Neuronale Netze lernen, indem ihre Verknüpfungsgewichte verfeinert werden. Dies wird durch den Fehler gesteuert – die Differenz zwischen der richtigen Antwort, die durch die Trainingsdaten vorgegeben ist, und der tatsächlichen Ausgabe.
- Der Fehler an den Ausgabeknoten ist einfach die Differenz zwischen der gewünschten und der tatsächlichen Ausgabe.
- Der den internen Knoten zugeordnete Fehler ist aber nicht offensichtlich. Eine Möglichkeit ist, die Fehler der Ausgabeschicht proportional zur Größe der verbundenen Verknüpfungsgewichte aufzuteilen und dann die jeweiligen Anteile an jedem internen Knoten zusammenzufassen.

Backpropagierung von Fehlern mit Matrizenmultiplikation

Können wir alle diese aufwendigen Berechnungen per Matrizenmultiplikation vereinfachen? Dieses Verfahren hat bereits weiter oben geholfen, als wir Unmengen von Berechnungen bei der Weiterleitung der Eingabesignale ausgeführt haben.

Um festzustellen, ob sich die Fehler-Backpropagierung mithilfe der Matrizenmultiplikation prägnanter ausdrücken lässt, notieren wir die einzelnen Schritte mit Symbolen. Man sagt auch: Es wird versucht, den Prozess zu *vektorisieren*. Wenn wir in der Lage sind, die zahlreichen Berechnungen in Matrixform auszudrücken, können wir sie viel prägnanter niederschreiben, und Computer können die gesamte Arbeit wesentlich effizienter ausführen, weil sie von den sich wiederholenden Ähnlichkeiten in den erforderlichen Berechnungen profitieren.

Den Ausgangspunkt bilden die Fehler, die sich in der letzten Schicht, der Ausgabeschicht des neuronalen Netzes, ergeben. Im Beispiel haben wir nur zwei Knoten in der Ausgabeschicht, und die Fehler sind e_1 und e_2:

$$error_{output} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Als Nächstes konstruieren wir die Matrix für Fehler der versteckten Schicht. Da das schwierig zu sein scheint, gehen wir jeden Schritt einzeln durch. Los geht es mit dem ersten Knoten in der versteckten Schicht. Sehen Sie sich noch einmal Abbildung 1-70 an. Hier ist zu erkennen, dass es für den Fehler des ersten versteckten Knotens zwei Verbindungen von der Ausgabeschicht gibt, die Fehleran-

Da es hilfreich ist, diese ganze Theorie in Aktion zu sehen, zeigt Abbildung 1-70 die Backpropagierung von Fehlern in einem einfachen dreischichtigen neuronalen Netz mit realen Zahlen.

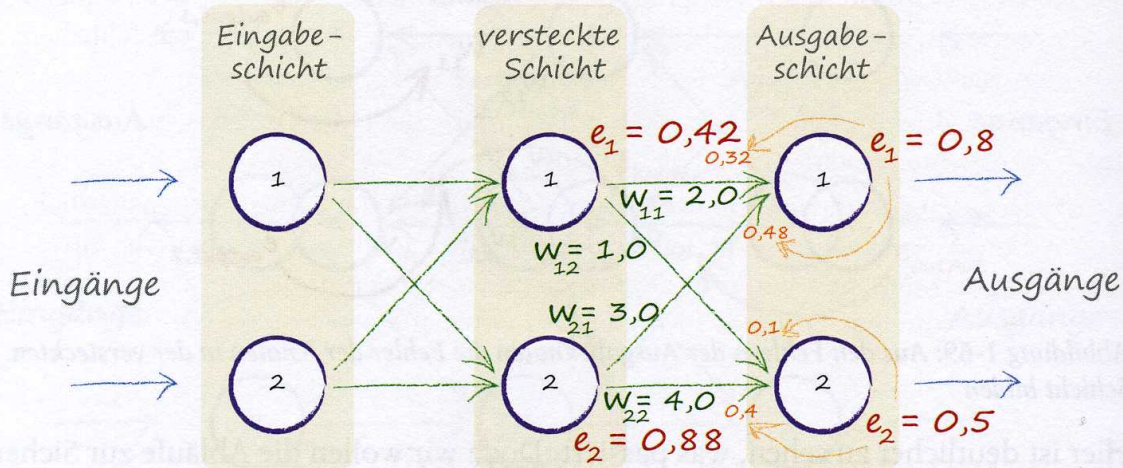


Abbildung 1-70: Das neuronale Beispielnetz mit Fehler-Backpropagierung bis zur verdeckten Schicht

Verfolgen wir einen Fehler zurück. Der Fehler 0,5 beim zweiten Knoten der Ausgabeschicht wird mit 0,1 und 0,4 proportional auf die beiden Verknüpfungen mit den Gewichten 1,0 und 4,0 aufgeteilt. Außerdem zeigt Abbildung 1-70, dass der zusammengefasste Fehler am zweiten Knoten der versteckten Schicht die Summe der verbundenen aufgeteilten Fehler – hier 0,48 und 0,4 – ist, was 0,88 ergibt.

Abbildung 1-71 stellt die gleiche Idee dar, angewandt auf die vorhergehende Schicht, um den Fehler noch weiter zurückzupropagieren.

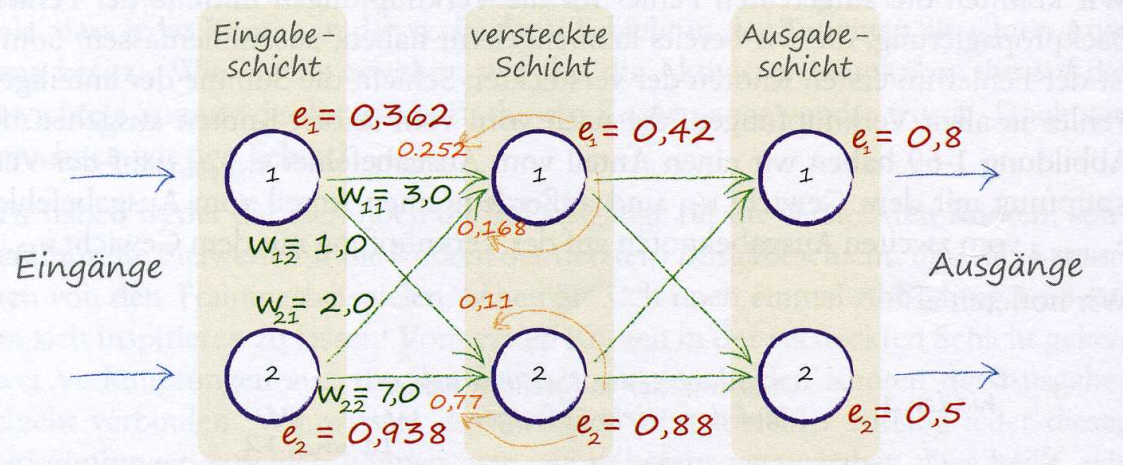


Abbildung 1-71: Das neuronale Beispielnetz mit Fehler-Backpropagierung bis zur Eingabeschicht

Kernideen

- Neuronale Netze lernen, indem ihre Verknüpfungsgewichte verfeinert werden. Dies wird durch den Fehler gesteuert – die Differenz zwischen der richtigen Antwort, die durch die Trainingsdaten vorgegeben ist, und der tatsächlichen Ausgabe.
- Der Fehler an den Ausgabeknoten ist einfach die Differenz zwischen der gewünschten und der tatsächlichen Ausgabe.
- Der den internen Knoten zugeordnete Fehler ist aber nicht offensichtlich. Eine Möglichkeit ist, die Fehler der Ausgabeschicht proportional zur Größe der verbundenen Verknüpfungsgewichte aufzuteilen und dann die jeweiligen Anteile an jedem internen Knoten zusammenzufassen.

Backpropagierung von Fehlern mit Matrizenmultiplikation

Können wir alle diese aufwendigen Berechnungen per Matrizenmultiplikation vereinfachen? Dieses Verfahren hat bereits weiter oben geholfen, als wir Unmengen von Berechnungen bei der Weiterleitung der Eingabesignale ausgeführt haben.

Um festzustellen, ob sich die Fehler-Backpropagierung mithilfe der Matrizenmultiplikation prägnanter ausdrücken lässt, notieren wir die einzelnen Schritte mit Symbolen. Man sagt auch: Es wird versucht, den Prozess zu *vektorisieren*. Wenn wir in der Lage sind, die zahlreichen Berechnungen in Matrixform auszudrücken, können wir sie viel prägnanter niederschreiben, und Computer können die gesamte Arbeit wesentlich effizienter ausführen, weil sie von den sich wiederholenden Ähnlichkeiten in den erforderlichen Berechnungen profitieren.

Den Ausgangspunkt bilden die Fehler, die sich in der letzten Schicht, der Ausgabeschicht des neuronalen Netzes, ergeben. Im Beispiel haben wir nur zwei Knoten in der Ausgabeschicht, und die Fehler sind e_1 und e_2:

$$error_{output} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Als Nächstes konstruieren wir die Matrix für Fehler der versteckten Schicht. Da das schwierig zu sein scheint, gehen wir jeden Schritt einzeln durch. Los geht es mit dem ersten Knoten in der versteckten Schicht. Sehen Sie sich noch einmal Abbildung 1-70 an. Hier ist zu erkennen, dass es für den Fehler des ersten versteckten Knotens zwei Verbindungen von der Ausgabeschicht gibt, die Fehleran-

teile liefern. Über diese Verbindungen kommen die Fehlersignale $e_1 \times w_{11} / (w_{11} + w_{21})$ und $e_2 \times w_{12} / (w_{12} + w_{22})$. Auch beim zweiten Knoten der versteckten Schicht sehen wir zwei Verbindungen, die zum Fehler des Knotens beitragen, nämlich $e_1 \times w_{21} / (w_{21} + w_{11})$ und $e_2 \times w_{22} / (w_{22} + w_{12})$. Wir haben bereits weiter oben gesehen, wie diese Ausdrücke ermittelt werden.

Für die versteckte Schicht bekommen wir also die in Abbildung 1-72 gezeigte Matrix. Sie ist ein wenig komplizierter, als ich es mag.

$$error_{hidden} = \begin{pmatrix} \frac{w_{11}}{w_{11} + w_{21}} & \frac{w_{12}}{w_{12} + w_{22}} \\ \frac{w_{21}}{w_{21} + w_{11}} & \frac{w_{22}}{w_{22} + w_{12}} \end{pmatrix} \bullet \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Abbildung 1-72: Fehlermatrix für die versteckte Schicht

Es wäre großartig, wenn sich diese Form als ganz einfache Multiplikation von Matrizen umschreiben ließe, über die wir bereits verfügen. Das sind die Gewichts-, Vorwärtssignal- und Ausgabefehlermatrizen. Wie bereits erwähnt, ergeben sich damit riesige Vorteile.

Leider können wir dies nicht in eine super einfache Matrizenmultiplikation umwandeln, wie es noch bei der Vorwärtsweiterleitung der Signale möglich war (siehe weiter oben). Die Fehleranteile in dieser großen, voll besetzten Matrix (siehe Abbildung 1-72) sind schwer zu entwirren! Es wäre hilfreich gewesen, wenn wir diese voll besetzte Matrix in eine einfache Kombination der verfügbaren Matrizen hätten aufteilen können.

Was können wir tun? Wir möchten unbedingt diese Vorteile der Matrizenmultiplikation nutzen, um die Berechnungen effizient ausführen zu können.

Zeit also, etwas frecher zu agieren.

Sehen Sie sich noch einmal den Ausdruck in Abbildung 1-72 an. Das Wichtigste ist die Multiplikation der Ausgabefehler e_n mit den verknüpften Gewichten w_{ij}. Je größer das Gewicht, desto stärker wirkt der Ausgabefehler zurück auf die versteckte Schicht. Das ist der entscheidende Teil. Der Nenner dieser Brüche ist eine Art Normalisierungsfaktor. Wenn wir diesen Faktor ignorieren, verlieren wir lediglich die Skalierung der zurückgeführten Fehler. Das heißt, $e_1 \times w_{11} / (w_{11} + w_{21})$ würde zum viel einfacheren Ausdruck $e_1 \times w_{11}$ werden.

Mit dieser Vereinfachung nimmt die Matrizenmultiplikation die in Abbildung 1-73 gezeigte Form an.

$$error_{hidden} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \bullet \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Abbildung 1-73: Vereinfachte Gewichtsmatrix

Diese Gewichtsmatrix sieht wie diejenige aus, die wir zuvor konstruiert haben, ist aber entlang einer Diagonalen gespiegelt, sodass das Element rechts oben jetzt links unten liegt und das Element links unten nach rechts oben gewandert ist. Diese Operation heißt *Transponieren* einer Matrix und wird geschrieben als $\mathbf{w}^T$.

Abbildung 1-74 gibt zwei Beispiele für die Transponierung von Matrizen an. Anhand der gewählten Zahlen können Sie deutlich sehen, was passiert. Wie das untere Beispiel zeigt, funktioniert die Transponierung auch für Matrizen, bei denen die Anzahl der Zeilen von der Anzahl der Spalten abweicht.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Abbildung 1-74: Beispiele für die Transponierung von Matrizen

Wir haben also das, was wir angestrebt haben – eine Matrixlösung, um die Fehler zurückzuführen (siehe Abbildung 1-75).

$$error_{hidden} = w^T_{hidden_output} \bullet error_{output}$$

Abbildung 1-75: Ausdruck für die Fehler-Backpropagierung in Matrixform

Dies ist zwar großartig, doch war es richtig, diesen Normalisierungsfaktor wegzulassen? Es zeigt sich, dass diese einfachere Rückführung der Fehlersignale genauso gut arbeitet wie die kompliziertere, die wir zuvor ermittelt hatten. Im Blog zu diesem Buch findet sich ein Post-Eintrag, der die Ergebnisse verschiedener Arten der

Fehler-Backpropagierung angibt. Wenn der einfachere Weg wirklich gut funktioniert, behalten wir ihn bei!

Denkt man genauer darüber nach, wird klar, dass sich das Netz selbst bei übermäßig großen oder kleinen zurückgeführten Fehlern während der nächsten Lerniterationen selbst korrigieren wird. Wichtig ist vor allem, dass die zurückgeführten Fehler die Stärke der Verknüpfungsgewichte beachten, weil sich hieraus am besten ableiten lässt, wie der Fehler geteilt werden soll.

Wir haben viel Arbeit bewältigt, wirklich eine riesige Menge!

Kernideen

- Die Fehler-Backpropagierung lässt sich als Matrizenmultiplikation ausdrücken.
- Dadurch können wir ungeachtet der Netzgröße prägnanter formulieren, und Computersprachen, die Matrixberechnungen verstehen, können effizienteren und schnelleren Code erzeugen.
- Das bedeutet, dass sowohl die Signalweiterleitung als auch die Fehlerrückführung mithilfe von Matrixberechnungen effizient ausgedrückt werden können.

Wir legen jetzt eine wohlverdiente Pause ein, weil der nächste und letzte Abschnitt zur Theorie zwar ziemlich cool ist, aber unsere volle Aufmerksamkeit fordert.

Wie aktualisieren wir eigentlich die Gewichte?

Bislang haben wir die sehr zentrale Frage, wie die Verknüpfungsgewichte in einem neuronalen Netz aktualisiert werden, noch nicht in Angriff genommen. Wir haben daran gearbeitet, bis zu diesem Punkt zu kommen, und wir sind fast da. Allerdings müssen wir noch einen Kerngedanken verstehen, bevor wir dieses Geheimnis lüften können.

Die Fehler haben wir bis jetzt zu jeder Schicht des Netzes zurückgeführt. Warum? Weil wir uns am Fehler orientieren, wenn es darum geht, die Verknüpfungsgewichte anzupassen, um die vom Netz gegebene Gesamtantwort zu verbessern. Grundsätzlich tun wir damit das Gleiche wie beim linearen Klassifizierer zu Beginn dieses Leitfadens.

Doch diese Knoten sind keine einfachen linearen Klassifizierer. Diese etwas komplexeren Knoten summieren die gewichteten Signale, die in den Knoten eingehen, und wenden die sigmoidale Schwellwertfunktion an. Wie also aktualisieren wir die Gewichte für Verknüpfungen, die diese komplexeren Knoten verbinden?

Warum können wir nicht mit irgendeiner raffinierten Algebra direkt herausfinden, wie groß die Gewichte sein sollten?

Wir können mit einer raffinierten Algebra die Gewichte nicht direkt ermitteln, weil die mathematischen Operationen zu schwierig sind. Es sind einfach zu viele Kombinationen von Gewichten und zu viele Funktionen von Funktionen von Funktionen ... zu kombinieren, wenn wir das Signal in Vorwärtsrichtung durch das Netz leiten. Stellen Sie sich nur einmal ein kleines neuronales Netz mit drei Schichten und drei Neuronen in jeder Schicht vor, wie wir es weiter oben verwendet haben. Wie würden Sie ein Gewicht für eine Verknüpfung zwischen dem ersten Eingabeknoten und dem zweiten versteckten Knoten anpassen, sodass der dritte Ausgabeknoten seine Ausgabe um beispielsweise 0,5 erhöht? Selbst wenn wir erfolgreich wären, könnte der Effekt zunichtegemacht werden, indem ein anderes Gewicht angepasst wird, um einen anderen Ausgabeknoten zu verbessern. Offenbar ist das Ganze überhaupt nicht trivial.

Um ein Bild davon zu bekommen, wie untrivial das ist, sollten Sie sich den furchtbaren Ausdruck in Abbildung 1-76 ansehen, der die Ausgabe eines Ausgabeknotens als Funktion der Eingänge und der Verknüpfungsgewichte für ein einfaches dreischichtiges neuronales Netz mit drei Knoten in jeder Schicht zeigt. Der Eingang bei Knoten i ist x_i, die Gewichte für die Verknüpfungen, die den Eingabeknoten i mit dem versteckten Knoten j verbinden, sind $w_{i,j}$. Analog wird die Ausgabe des versteckten Knotens j mit x_j bezeichnet, und die Gewichte für die Verknüpfungen, die den versteckten Knoten j mit dem Ausgabeknoten k verbinden, heißen $w_{j,k}$. Das Symbol Σ_a^b bedeutet, dass der danach stehende Ausdruck für alle Werte zwischen a und b summiert wird.

$$o_k = \frac{1}{1 + e^{-\sum_{j=1}^{3}\left(w_{j,k} \cdot \frac{1}{1+e^{-\sum_{i=1}^{3}(w_{i,j} \cdot x_i)}}\right)}}$$

Abbildung 1-76: Ausdruck, der die Ausgabe eines Ausgabeknotens als Funktion der Eingänge und Verknüpfungsgewichte für ein dreischichtiges neuronales Netz mit drei Knoten in jeder Schicht beschreibt

Entsetzlich! Das wollen wir lieber nicht entwirren.

Anstatt nun überschlau sein zu wollen, könnten wir nicht einfach zufällige Kombinationen von Gewichten ausprobieren, bis wir eine gute Kombination gefunden haben?

Diese Idee ist gar nicht so verrückt, vor allem wenn man in einem schwierigen Problem feststeckt. Das Konzept, mit dessen Hilfe man zum Beispiel sogar Passwörter zu knacken versucht, ist auch als *Brute-Force-Methode* bekannt. Das kann durchaus funktionieren, wenn das Passwort ein kurzes Wort oder ein Name ist, weil dann ein schneller Heimcomputer in kurzer Zeit alle Möglichkeiten durchprobieren kann. Stellen Sie sich nun vor, dass jedes Gewicht 1.000 Möglichkeiten zwischen –1 und +1 haben könnte, wie zum Beispiel 0,501, –0,203 und 0,999. Dann gibt es bei einem dreischichtigen neuronalen Netz mit drei Knoten in jeder Schicht 18 Gewichte, sodass 18.000 Möglichkeiten zu testen wären. Wenn wir ein eher typisches neuronales Netz mit 500 Knoten in jeder Schicht betrachten, müssten 500 Millionen Möglichkeiten für die Gewichte getestet werden. Nimmt man für jeden Satz von Kombinationen eine Rechendauer von einer Sekunde an, würde es 16 Jahre dauern, die Gewichte nach nur einem Trainingsbeispiel zu aktualisieren! Und bei 1.000 Trainingsbeispielen wären wir bei 16.000 Jahren!

Die Brute-Force-Methode ist offenbar in der Praxis überhaupt nicht anwendbar. Die Lage wird schnell noch schlimmer, wenn weitere Schichten, Knoten oder mögliche Gewichtswerte hinzukommen.

An diesem Rätsel bissen sich Mathematiker jahrelang die Zähne aus, es wurde erst in den 1960er- bis 1970er-Jahren praktisch gelöst. Es gibt verschiedene Ansichten darüber, wem dies zuerst gelang oder wer den entscheidenden Durchbruch erzielte, doch wichtig ist vor allem, dass diese späte Entdeckung zu einer Explosion moderner neuronaler Netzen führte, die manche sehr eindrucksvolle Aufgaben durchführen können.

Wie lösen wir nun ein derartiges anscheinend schweres Problem? Ob Sie es glauben oder nicht, Sie besitzen bereits die Werkzeuge, um es in eigener Regie zu bewerkstelligen. Wir haben alles dazu Notwendige bereits weiter oben behandelt. Machen wir also weiter!

Zunächst müssen wir aber *Pessimismus* akzeptieren.

Die mathematischen Ausdrücke, die zeigen, wie alle Gewichte zur Ausgabe eines neuronalen Netzes führen, sind zu komplex, um sie leicht entwirren zu können. Die Gewichtskombinationen sind zu zahlreich, um eine nach der anderen testen und die beste finden zu können.

Es gibt sogar noch mehr Gründe, um pessimistisch zu sein. Die Trainingsdaten reichen eventuell nicht aus, um ein Netz ordnungsgemäß anzulernen. Sie enthalten vielleicht sogar Fehler, sodass wir nicht mehr davon ausgehen können, dass sie perfekt wahr sind, um von ihnen lernen zu können. Das Netz selbst enthält möglicherweise nicht genügend Schichten oder Knoten, um die richtige Lösung für das Problem zu modellieren.

Das bedeutet: Wir müssen einen realistischen Ansatz wählen, der diese Beschränkungen erkennt. Wenn wir das tun, könnten wir ein Konzept finden, das zwar

mathematisch nicht perfekt ist, tatsächlich aber bessere Ergebnisse liefert, weil es keine falschen, idealisierten Annahmen trifft.

Wir wollen veranschaulichen, was damit gemeint ist. Stellen Sie sich eine sehr komplizierte Landschaft vor mit Bergspitzen und Tälern sowie Bergen mit tückischen Unebenheiten und Spalten. Es ist finster, und Sie können nichts sehen. Sie wissen, dass Sie sich auf einer Anhöhe befinden und ganz nach unten gelangen müssen. Von der gesamten Landschaft besitzen Sie keine genaue Karte. Allerdings haben Sie eine Taschenlampe. Was tun Sie jetzt? Wahrscheinlich werden Sie im Schein der Taschenlampe den Boden in der nahen Umgebung inspizieren. Weiter entferntes Gelände ist überhaupt nicht zu sehen und gleich gar nicht die gesamte Landschaft. Sie können erkennen, welcher Teil des Bodens anscheinend nach unten führt, und kleine Schritte in dieser Richtung gehen. Auf diese Weise tasten Sie sich langsam den Berg hinunter, immer Schritt für Schritt, ohne eine vollständige Karte zu besitzen und ohne im Voraus eine Route geplant zu haben.

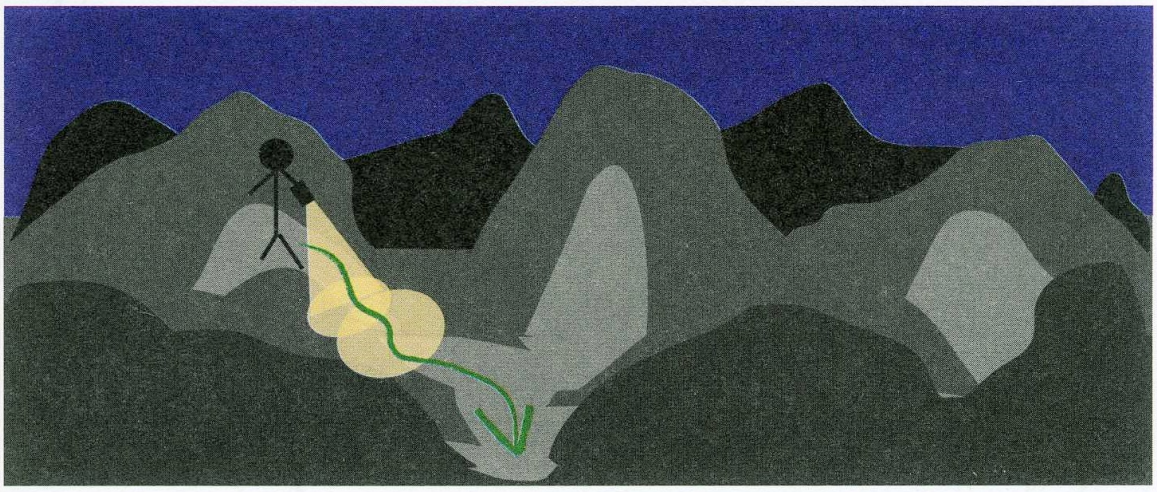


Abbildung 1-77: Beispiel für ein bergiges Gelände, in dem Sie sich fast blind bewegen müssen

Die mathematische Version dieses Konzepts heißt *Gradientenverfahren* oder auch *Verfahren des steilsten Abstiegs* – es dürfte klar sein, warum. Nachdem Sie einen Schritt gegangen sind, untersuchen Sie wieder die nahe Umgebung, um festzustellen, in welcher Richtung Sie Ihrem Ziel näher kommen. Dann wagen Sie erneut einen Schritt in diese Richtung. Das setzen Sie so lange fort, bis Sie glücklich am Fuß der Berge angekommen sind. Der Gradient entspricht der Bodenneigung. Sie gehen in die Richtung, wo die Neigung am steilsten nach unten führt.

Stellen Sie sich nun diese komplexe Landschaft als mathematische Funktion vor. Das Gradientenverfahren erlaubt es uns, das Minimum zu finden, ohne diese komplexe Funktion so weit zu verstehen, um sie mathematisch beschreiben zu können. Wenn eine Funktion so schwierig ist, dass sich das Minimum nicht ohne Weiteres mithilfe der Algebra finden lässt, können wir stattdessen diese Methode verwenden. Zweifellos dürfen wir keine genaue Antwort erwarten, weil wir uns der Antwort schrittweise nähern und unsere Position dabei Stück für Stück verbes-

sern. Doch das ist besser, als überhaupt keine Antwort zu bekommen. Immerhin können wir die Antwort mit immer kleineren Schritten in Richtung tatsächliches Minimum immer weiter verfeinern, bis wir mit der erreichten Genauigkeit zufrieden sind.

Wie hängt nun dieses wirklich coole Gradientenverfahren mit neuronalen Netzen zusammen? Wenn die komplexe schwierige Funktion den Fehler des Netzes darstellt, bedeutet ein Hinabsteigen, um das Minimum zu finden, dass wir den Fehler minimieren. Wir verbessern die Ausgabe des Netzes. Das ist genau das, was wir wollen!

Die Idee des Gradientenabstiegs wollen wir an einem äußerst einfachen Beispiel veranschaulichen, um sie richtig verstehen zu können.

Der Graph in Abbildung 1-78 zeigt eine einfache Funktion $y = (x - 1)^2 + 1$. Wäre es eine Funktion, bei der y den Fehler bedeutet, würden wir das x suchen, das den Fehler minimiert. Für einen Moment wollen wir so tun, als wäre dies keine einfache Funktion, sondern eine komplizierte, schwierige Funktion.

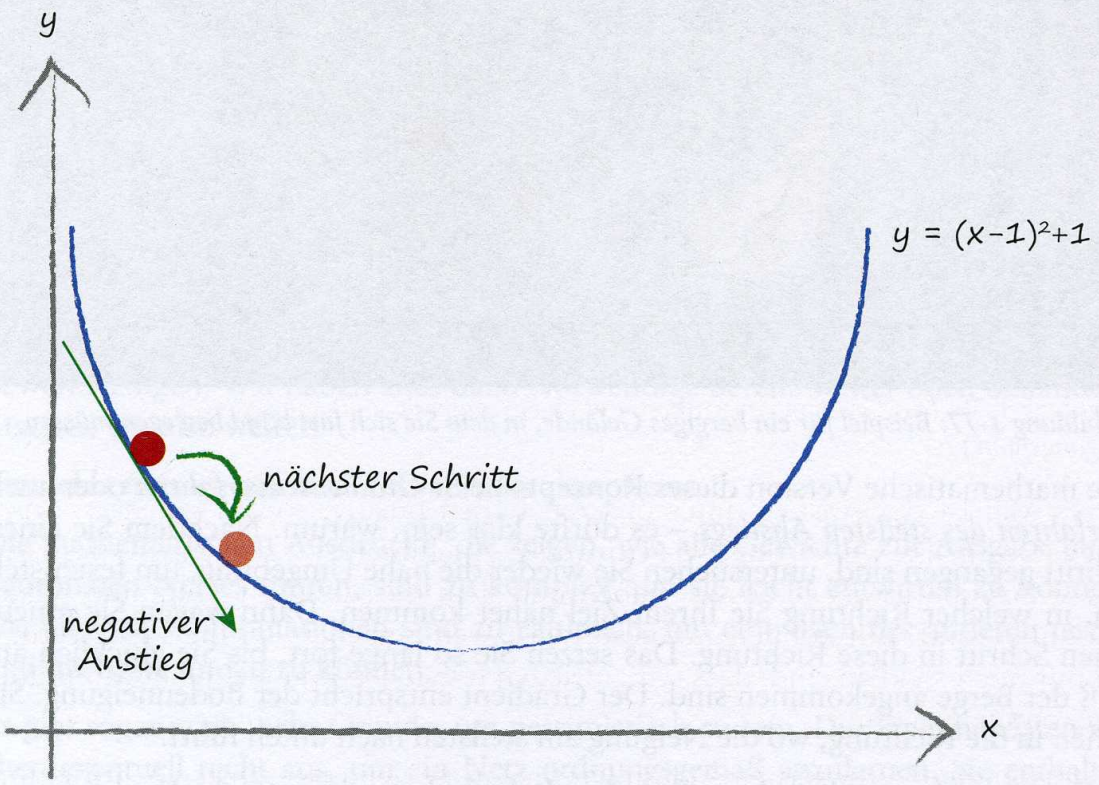


Abbildung 1-78: Beispiel einer Funktion, um das Gradientenverfahren zu veranschaulichen

Um das Gradientenverfahren durchzuführen, müssen wir irgendwo beginnen. Der Graph zeigt unseren zufällig gewählten Ausgangspunkt. Wie der Bergsteiger sehen wir uns die Umgebung unserer aktuellen Position an und stellen fest, in welche Richtung es nach unten geht. Der Anstieg ist auf dem Graphen markiert und in diesem Fall ein negativer Gradient. Wir folgen der Abwärtsrichtung, sodass wir

uns entlang der x-Achse nach rechts bewegen. Das heißt, wir erhöhen x ein wenig. Das ist der erste Schritt unseres Bergsteigers. Es ist zu erkennen, dass wir unsere Position verbessert haben und näher an das tatsächliche Minimum herangekommen sind.

Nehmen wir jetzt an, dass wir an einer anderen Stelle gestartet sind, wie es der Graph in Abbildung 1-79 zeigt.

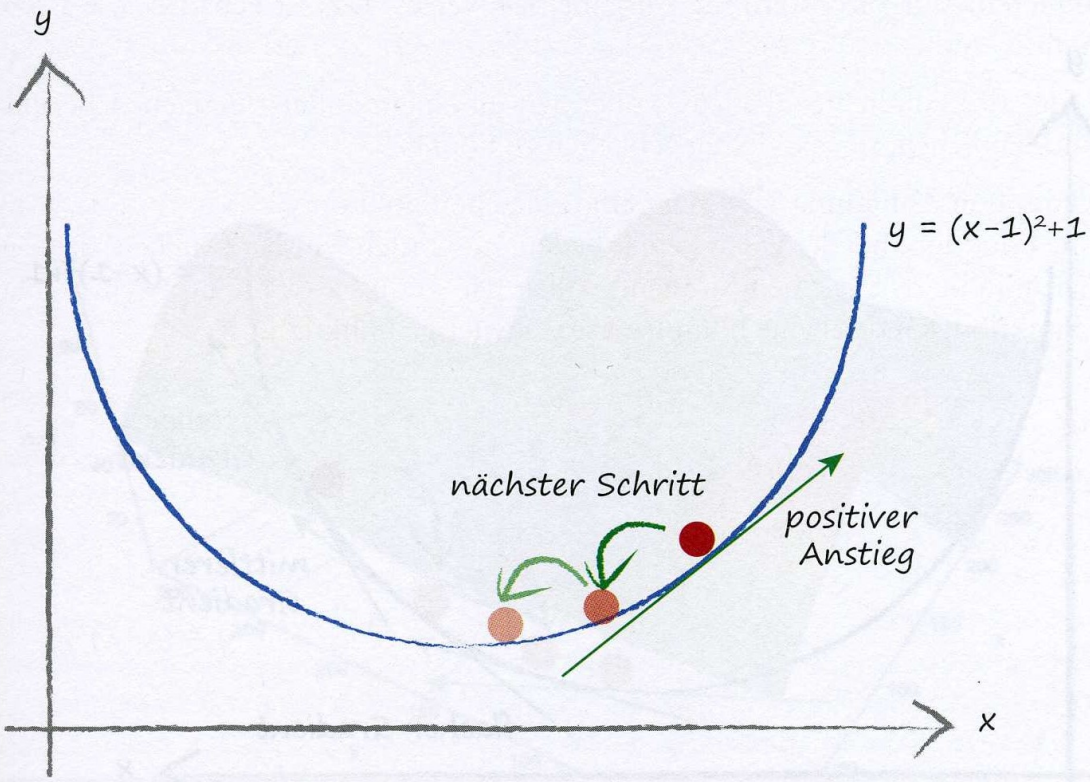


Abbildung 1-79: Das Gradientenverfahren mit einem anderen zufällig gewählten Ausgangspunkt

Dieses Mal ist der Anstieg unter unseren Füßen positiv, sodass wir nach links gehen. Das heißt, wir verringern x ein wenig. Auch hier ist zu erkennen, dass wir unsere Position etwas verbessert haben und dem tatsächlichen Minimum näher gekommen sind. Wir können auf diese Weise weitermachen, bis die Verbesserungen nur noch so klein sind, dass wir zufrieden feststellen können, beim Minimum angekommen zu sein.

Als notwendige Verfeinerung ändern wir die Größe der ausgeführten Schritte, um zu vermeiden, über das Minimum hinauszuschießen und ewig um das Minimum herumzupendeln. Wenn man nur Zweimeterschritte machen kann, aber bereits 0,5 Meter entfernt vom wahren Minimum sind, ist leicht einzusehen, dass uns jeder Schritt in Richtung Minimum über das Minimum hinausbringt. Passen wir nun die Schrittgröße so an, dass sie proportional zur Größe des Gradienten ist, führen wir kleinere Schritte aus, wenn wir uns dem Minimum nähern. Das setzt aber voraus, dass der Anstieg bei der Annäherung an das Minimum tatsächlich fla-

cher wird. Für die meisten sanften stetigen Funktionen ist diese Annahme durchaus zutreffend. Für wilde Zickzackfunktionen mit Sprüngen und Lücken, die der Mathematiker als *Unstetigkeiten* bezeichnet, ist die Annahme allerdings nicht brauchbar.

Abbildung 1-80 veranschaulicht dieses Konzept der Schrittgrößenanpassung bei kleiner werdendem Funktionsgradienten, der ein guter Indikator dafür ist, wie nahe wir einem Minimum sind.

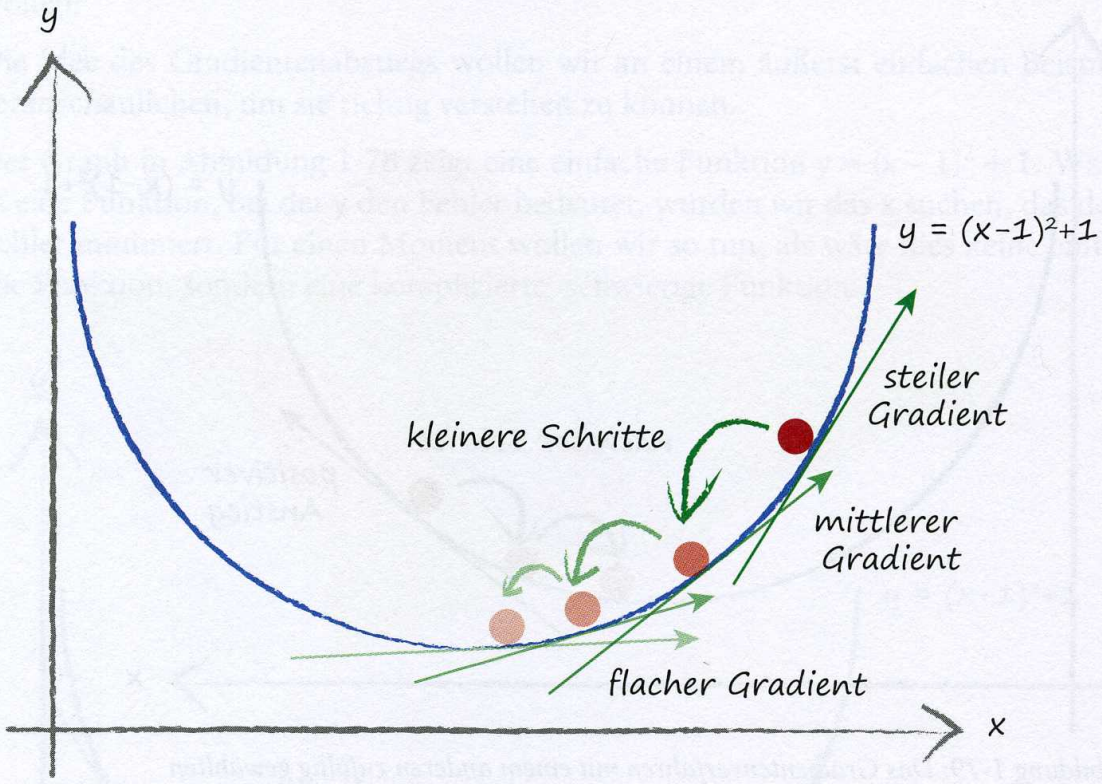


Abbildung 1-80: Gradientenverfahren mit kleiner werdenden Schritten

Haben Sie übrigens bemerkt, dass wir x in der Gegenrichtung zum Gradienten erhöht haben? Ein positiver Gradient bedeutet, wir verringern x. Ein negativer Gradient bedeutet, wir erhöhen x. Die Graphen zeigen das zwar deutlich, doch man vergisst es leicht und schlägt die falsche Richtung ein.

Wir haben bei diesem Gradientenabstieg das wahre Minimum nicht mittels Algebra ermittelt, weil wir so getan haben, als wäre die Funktion $y = (x - 1)^2 + 1$ zu komplex und zu schwierig. Selbst wenn wir nicht in der Lage sind, den Anstieg mit mathematischer Genauigkeit zu bestimmen, können wir ihn schätzen. Und wie Sie sehen, funktioniert das trotzdem ziemlich gut, um uns in die generell richtige Richtung zu bringen.

Diese Methode überzeugt vor allem bei Funktionen mit vielen Parametern. Das heißt, y hängt nicht nur von x ab, sondern vielleicht auch von a, b, c, d, e und f.

Wie bereits erwähnt, hängen die Ausgabefunktion und demzufolge die Fehlerfunktion eines neuronalen Netzes von vielen, vielen weiteren Gewichtsparametern ab. Oftmals von Hunderten davon!

Abbildung 1-81 veranschaulicht ebenfalls das Gradientenverfahren, doch mit einer etwas komplexeren Funktion, die von zwei Parametern abhängt. Dies lässt sich in drei Dimensionen darstellen, wobei die Höhe den Wert der Funktion angibt.

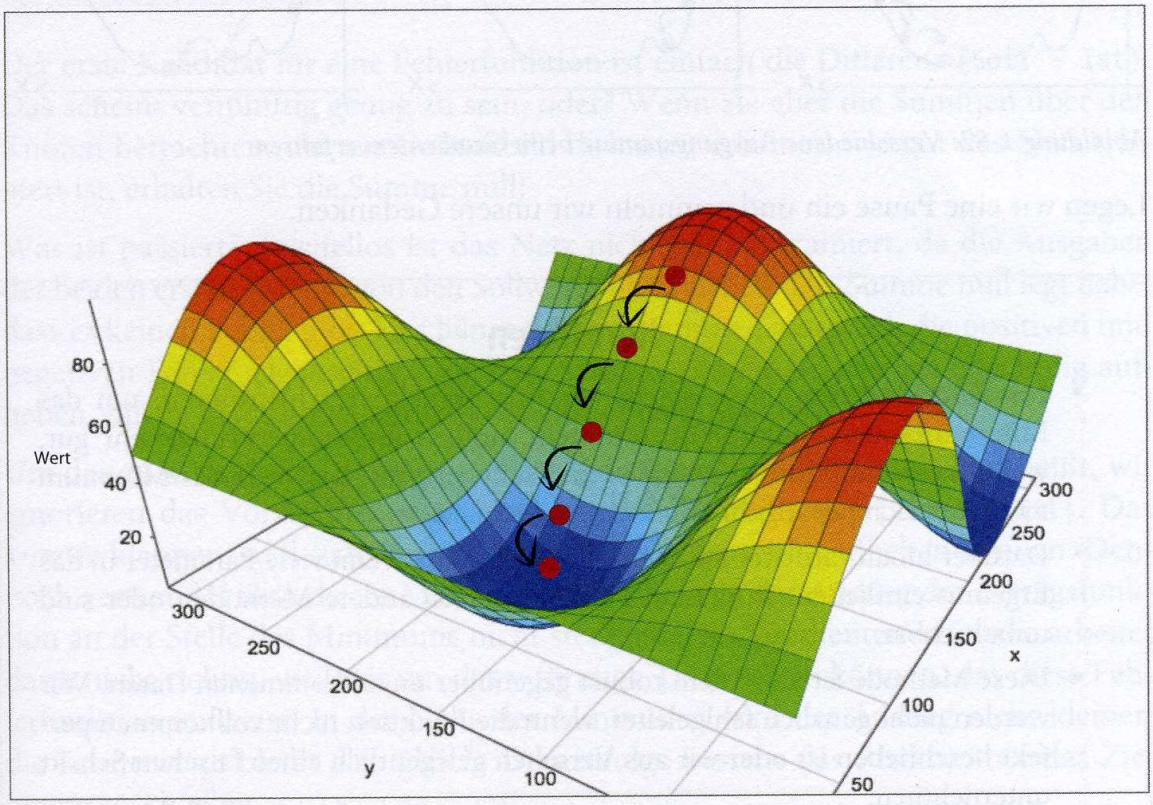


Abbildung 1-81: Gradientenverfahren bei einer komplexeren Funktion

Angesichts dieser dreidimensionalen Oberfläche fragen Sie sich möglicherweise, ob der Gradientenabstieg auch in dem anderen Tal auf der rechten Seite enden kann. Allgemeiner gedacht, stellt sich die Frage: Bleibt das Gradientenverfahren manchmal im falschen Tal stecken, weil bestimmte komplexe Funktionen viele Täler haben? Welches ist das falsche Tal? Es ist ein Tal, das nicht am tiefsten liegt. Die Antwort lautet: Ja, das kann passieren.

Um zu vermeiden, im falschen Tal – oder Funktionsminimum – zu landen, trainieren wir neuronale Netze mehrere Male, wobei jeweils verschiedene Ausgangspunkte auf dem Berg sicherstellen, dass wir nicht immer wieder im falschen Tal landen. Verschiedene Ausgangspunkte heißt, verschiedene Startparameter auszuwählen. Und bei neuronalen Netzen bedeutet das, mit verschiedenen Verknüpfungsgewichten zu beginnen.

Abbildung 1-82 veranschaulicht das Gradientenverfahren mit drei verschiedenen Ausgangspunkten, wobei ein Abstieg im falschen Tal gefangen wird.

Abbildung 1-82: Verschiedene Ausgangspunkte beim Gradientenverfahren

Legen wir eine Pause ein und sammeln wir unsere Gedanken.

Kernideen

- Das *Gradientenverfahren* ist ein wirklich brauchbares Instrument, um das Minimum einer Funktion zu ermitteln, und es funktioniert auch sehr gut, wenn diese Funktion so komplex und schwierig ist, dass sich das Minimum mathematisch per Algebra nur schwer finden lässt.
- Darüber hinaus arbeitet die Methode auch dann, wenn viele Parameter in das Ergebnis einfließen. In diesem Punkt scheitern andere Methoden oder sind unbrauchbar.
- Diese Methode ist außerdem robust gegenüber unvollkommenen Daten. Wir werden nicht gänzlich fehlgeleitet, wenn die Funktion nicht vollkommen perfekt beschrieben ist oder wir aus Versehen gelegentlich einen falschen Schritt unternehmen.

Die Ausgabe eines neuronalen Netzes ist eine komplexe und schwierige Funktion, deren Rückgabewert von vielen Parametern, den Verknüpfungsgewichten, beeinflusst wird. Können wir überhaupt mit dem Gradientenverfahren die richtigen Gewichte ermitteln? Ja, sofern wir uns für die richtige Fehlerfunktion entscheiden.

Die Ausgabefunktion eines neuronalen Netzes ist selbst keine Fehlerfunktion. Doch wir wissen, dass wir sie leicht in eine Fehlerfunktion umwandeln können, weil sich der Fehler aus der Differenz zwischen den Solltrainingswerten und den tatsächlichen Ausgabewerten berechnet.

Hierbei ist etwas zu beachten. Sehen Sie sich Tabelle 1-6 mit Trainingswerten (Sollwerten) und tatsächlichen Werten (Istwerten) für drei Ausgabeknoten und den Kandidaten für eine Fehlerfunktion an.

Tabelle 1-6: Kandidaten für eine Fehlerfunktion

Netzausgabe	Sollausgabe	Fehler (Soll – Ist)	Fehler |Soll – Ist|	Fehler $(\text{Soll} - \text{Ist})^2$
0,4	0,5	0,1	0,1	0,01
0,8	0,7	–0,1	0,1	0,01
1,0	1,0	0	0	0
Summe		0	0,2	0,04

Der erste Kandidat für eine Fehlerfunktion ist einfach die Differenz (`Soll - Ist`). Das scheint vernünftig genug zu sein, oder? Wenn Sie aber die Summen über den Knoten betrachten, um ein Gesamtbild davon zu erhalten, wie gut das Netz trainiert ist, erhalten Sie die Summe null!

Was ist passiert? Zweifellos ist das Netz nicht perfekt trainiert, da die Ausgaben der beiden ersten Knoten von den Sollwerten abweichen. Die Summe null legt nahe, dass es keinen Fehler gibt. Das hängt damit zusammen, dass sich die positiven und negativen Fehler gegenseitig aufheben. Selbst wenn sie sich nicht vollständig aufheben, dürfte klar sein, dass dies kein gutes Fehlermaß ist.

Um das zu korrigieren, nehmen wir den *Absolutwert* der Differenz. Das heißt, wir ignorieren das Vorzeichen, in mathematischer Schreibweise `|Ziel - Ist|`. Das könnte klappen, weil sich hierbei keine Werte gegenseitig aufheben können. Dennoch verwendet man dieses Maß so gut wie nie, weil der Anstieg der Betragsfunktion an der Stelle des Minimums nicht stetig ist. Das Gradientenverfahren arbeitet dann nicht richtig, weil wir um das v-förmige Tal pendeln können, das diese Fehlerfunktion aufweist. In der Nähe des Minimums wird der Anstieg nicht kleiner, die Schrittweite bleibt also gleich. Demzufolge besteht das Risiko, über das Ziel hinauszuschießen.

Bei der dritten Option wird das Quadrat der Differenz $(\text{Soll} - \text{Ist})^2$ genommen. Es gibt unter anderem folgende Gründe, warum wir diese dritte Option gegenüber der zweiten bevorzugen:

- Die erforderliche Algebra, um den Anstieg für das Gradientenverfahren zu bestimmen, ist mit diesem quadratischen Fehler leicht genug.
- Die Fehlerfunktion verläuft sanft und stetig, wodurch die Gradientenmethode gut funktioniert – es gibt weder Lücken noch abrupte Sprungstellen.
- Der Gradient wird in der Nähe des Minimums kleiner, sodass sich auch das Risiko verringert, über das Ziel hinauszuschießen, wenn wir uns bei den Schrittweiten am Gradienten orientieren.

Gibt es eine vierte Option? Ja, man kann alle Arten von komplizierten und interessanten Kostenfunktionen konstruieren. Manche funktionieren überhaupt nicht gut, manche nur für bestimmte Arten von Problemen, und andere wieder funktionieren zwar, sind aber die zusätzliche Komplexität nicht wert.

Richtig, wir befinden uns jetzt in der letzten Runde!

Um den Gradientenabstieg durchzuführen, müssen wir den Anstieg der Fehlerfunktion in Bezug auf die Gewichte ermitteln. Hierfür ist die *Differenzialrechnung* erforderlich. Möglicherweise sind Sie bereits mit der Differenzialrechnung vertraut, doch sollte das nicht der Fall sein oder Sie brauchen lediglich eine Auffrischung, finden Sie in Anhang A eine leicht verständliche Einführung. Differenzialrechnung ist einfach eine mathematisch exakte Methode, um herauszufinden, wie sich etwas ändert, wenn sich etwas anderes ändert. Wie ändert sich zum Beispiel die Länge einer Feder, wenn sich die Kraft ändert, die die Feder dehnt? Hier sind wir daran interessiert, wie die Fehlerfunktion von den Verknüpfungsgewichten in einem neuronalen Netz abhängt. Man kann die Frage auch anders stellen: »Wie empfindlich ist der Fehler gegenüber Änderungen in den Verknüpfungsgewichten?«

Wir beginnen mit einem Bild, das uns dabei hilft, uns daran zu orientieren, was wir zu erreichen versuchen.

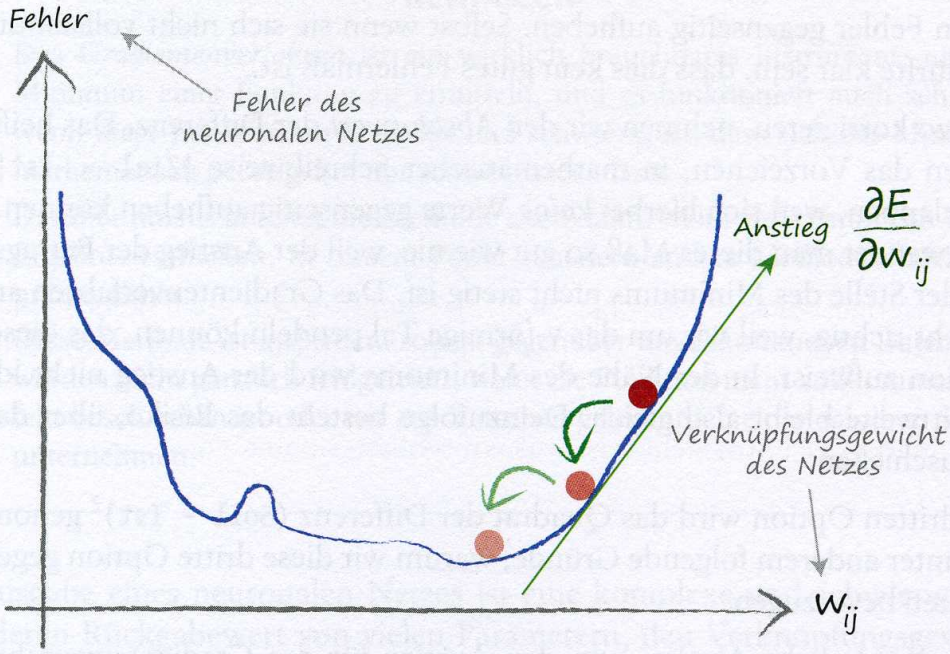


Abbildung 1-83: Funktion, die die Abhängigkeit des Fehlers von den Verknüpfungsgewichten beschreibt

Der Graph in Abbildung 1-83 entspricht der weiter oben in Abbildung 1-82 gezeigten Darstellung, um zu betonen, dass wir nichts grundsätzlich anderes machen. Dieses Mal ist aber die Funktion, die wir zu minimieren versuchen, der Fehler des neuronalen Netzes. Und der Parameter, der verfeinert werden soll, ist ein Verknüpfungsgewicht im Netz. In diesem einfachen Beispiel haben wir nur ein Gewicht gezeigt, doch wir wissen, dass neuronale Netze sehr viele Gewichte enthalten.

Das nächste Diagramm in Abbildung 1-84 stellt zwei Verknüpfungsgewichte dar. Dieses Mal ist die Fehlerfunktion eine dreidimensionale Oberfläche, die sich verändert, wenn die beiden Gewichte variieren. Wir versuchen wieder, den Fehler zu minimieren, der jetzt eher einer bergigen Landschaft mit einem Tal ähnelt.

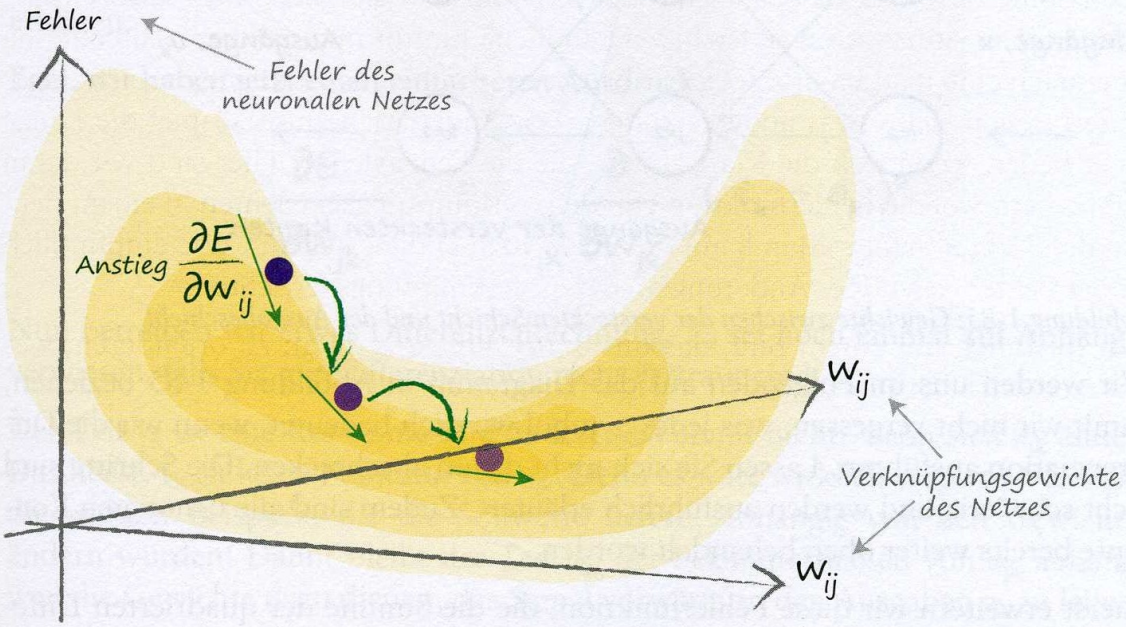


Abbildung 1-84: Zwei Verknüpfungsgewichte mit einer dreidimensionalen Oberfläche für die Fehlerfunktion

Es ist schwieriger, diese Fehleroberfläche als Funktion von noch mehr Parametern zu visualisieren, doch die Idee, das Minimum per Gradientenverfahren zu finden, ist immer noch die gleiche.

In mathematischer Form sieht das, was wir wollen, so aus:

$$\frac{\partial E}{\partial w_{jk}}$$

Dieser Ausdruck gibt an, wie sich der Fehler E ändert, wenn das Gewicht w_{jk} geändert wird. Das ist der Anstieg der Fehlerfunktion, die wir zum Minimum hinabsteigen wollen.

Bevor wir diesen Ausdruck analysieren, konzentrieren wir uns für den Moment auf die Verknüpfungsgewichte zwischen der versteckten Schicht und der Ausgabeschicht. In Abbildung 1-85 ist der interessierende Bereich hervorgehoben. Auf die Verknüpfungsgewichte zwischen der Eingabeschicht und der versteckten Schicht kommen wir später zurück.

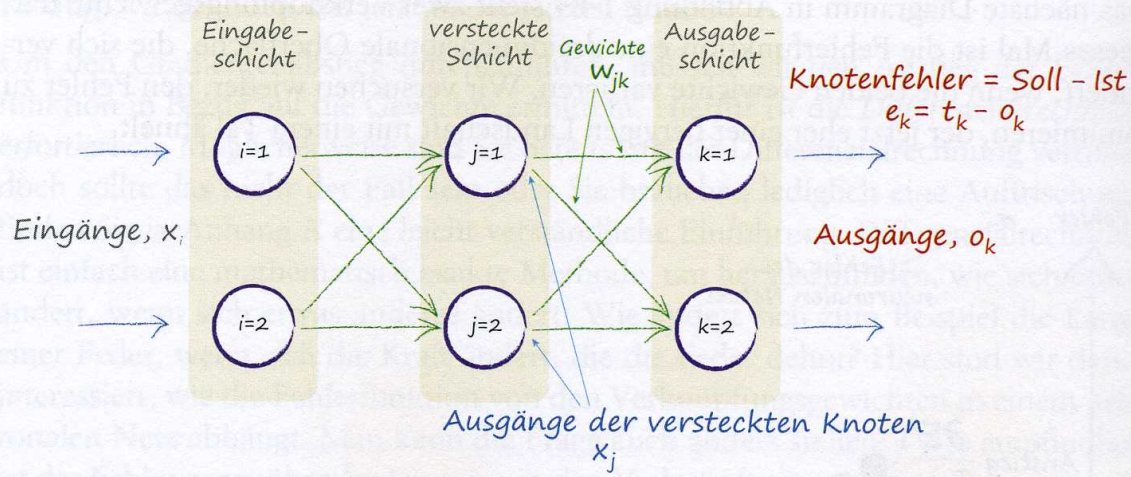


Abbildung 1-85: Gewichte zwischen der versteckten Schicht und der Ausgabeschicht

Wir werden uns im Folgenden auf das Diagramm in Abbildung 1-85 beziehen, damit wir nicht vergessen, was jedes Symbol wirklich bedeutet, wenn wir die Differenziation ausführen. Lassen Sie sich nicht davon abschrecken. Die Schritte sind nicht schwierig und werden ausführlich erläutert. Zudem sind alle benötigten Konzepte bereits weiter oben behandelt worden.

Zuerst erweitern wir diese Fehlerfunktion, die die Summe der quadrierten Differenzen zwischen Soll- und Istwerten ist und wo diese Summe über alle n Ausgabeknoten gebildet wird.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \Sigma_n (t_n - o_n)^2$$

Wir haben hier lediglich das ausgeschrieben, was die Fehlerfunktion E tatsächlich verkörpert.

Wir können das sogleich vereinfachen, indem wir feststellen, dass die Ausgabe bei einem Knoten n, die o_n ist, nur von den Verknüpfungen abhängt, die mit ihm verbunden sind. Das heißt, dass für einen Knoten k die Ausgabe o_k nur von den Gewichten w_{jk} abhängt, weil diese Gewichte den Verknüpfungen zum Knoten k zugeordnet sind.

Man kann dies auch so betrachten, dass die Ausgabe eines Knotens k nicht von den Gewichten w_{jb} abhängt, wobei b ungleich k ist, weil es keine Verknüpfung gibt, die sie verbindet. Das Gewicht w_{jb} ist einer Verknüpfung zugeordnet, die zum Ausgabeknoten b und nicht zu k führt.

Das bedeutet, wir können alle o_n aus dieser Summe entfernen mit Ausnahme des Knotens, mit dem das Gewicht w_{jk} verknüpft ist, d. h. o_k. Dadurch wird diese lästige Summe komplett entfernt! Ein guter Trick, den Sie sich merken sollten.

Vermutlich haben Sie erkannt, dass die Fehlerfunktion die Summe über sämtliche Ausgabeknoten eigentlich nicht gebraucht hat. Wie wir gesehen haben, hängt das damit zusammen, dass die Ausgabe eines Knotens nur von den verbundenen Verknüpfungen und folglich ihren Gewichten abhängt. Dies wird oftmals außer Acht gelassen in vielen Texten, die die Fehlerfunktion lediglich nennen, ohne sie zu erklären.

Egal, wir haben jetzt einen einfacheren Ausdruck.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

Nun betreiben wir etwas Differenzialrechnung. Es sei noch einmal auf Anhang A verwiesen, falls Sie mit Differenziation nicht vertraut sind.

Der Teil t_k ist eine Konstante und ändert sich somit nicht, wenn sich w_{jk} ändert. Das heißt, t_k ist keine Funktion von w_{jk}. Denn es wäre wirklich seltsam, wenn sich die wahren Beispiele, die die Sollwerte liefern, abhängig von den Gewichten ändern würden! Damit bleibt der Teil o_k, der bekanntermaßen von w_{jk} abhängt, weil die Gewichte dazu dienen, das Signal vorwärts zu den Ausgaben o_k zu leiten.

Mithilfe der Kettenregel zerlegen wir diese Differenziationsaufgabe in besser handhabbare Terme. Auch hier sei auf Anhang A verwiesen, wo Sie eine Einführung in die Kettenregel finden.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}}$$

Jetzt können wir nacheinander die einfacheren Teilaufgaben angehen. Der erste Teil ist leicht, da wir lediglich eine quadratische Funktion ableiten. Wir erhalten:

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

Der zweite Teil verlangt etwas mehr Überlegung, aber auch nicht zu viel. Dieses o_k ist die Ausgabe des Knotens k, der – wie Sie wissen – die Sigmoidfunktion auf die gewichtete Summe der eingehenden Signale anwendet. Wir schreiben das auf, um es deutlich zu machen:

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} sigmoid(\Sigma_j w_{jk} \cdot o_j)$$

Dieses o_j ist die Ausgabe vom Knoten der vorhergehenden versteckten Schicht und nicht die Ausgabe von der letzten Schicht o_k.

Wie differenzieren wir die Sigmoidfunktion? Wir könnten den langen und schwierigen Weg einschlagen und dabei auf die grundlegenden Konzepte zurückgreifen, die Anhang A erläutert, doch andere haben uns diese Arbeit bereits abgenommen. Wir können einfach die wohlbekannte Antwort übernehmen, genau wie es Mathematiker jeden Tag überall in der Welt tun.

$$\frac{\partial}{\partial x} sigmoid(x) = sigmoid(x)\left(1 - sigmoid(x)\right)$$

Manche Funktionen wandeln sich zu entsetzlichen Ausdrücken, wenn man sie differenziert. Diese Sigmoidfunktion hat aber ein hübsch einfaches und leicht zu verwendendes Ergebnis. Das ist einer der Gründe, warum die Sigmoidfunktion als Aktivierungsfunktion in neuronalen Netzen so beliebt ist.

Wenden wir also dieses coole Ergebnis an, um den Ausdruck zu erhalten, den Abbildung 1-86 zeigt.

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot \frac{\partial}{\partial w_{jk}}(\Sigma_j w_{jk} \cdot o_j)$$

$$= -2(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

Abbildung 1-86: Ausdruck für den Fehleranstieg nach Differenziation

Woher stammt dieses zusätzliche letzte Element? Es ergibt sich aus der Kettenregel, die erneut auf die Ableitung der Sigmoidfunktion angewendet wurde, weil der Ausdruck innerhalb der Funktion `sigmoid()` ebenfalls bezüglich w_{jk} differenziert werden muss. Auch das ist leicht- und die Antwort ist einfach o_j.

Bevor wir die endgültige Antwort notieren, wollen wir diese 2 vor dem ganzen Ausdruck verwerfen. Wir können das tun, weil wir nur an der Richtung des Anstiegs der Fehlerfunktion interessiert sind, sodass wir hinabsteigen können. Es spielt keine Rolle, ob vor diesem Ausdruck ein konstanter Faktor von 2, 3 oder sogar 100 steht, solange wir konsequent darin sind, woran wir uns halten. Werfen wir die Konstante also über Bord, um den Ausdruck weiter zu vereinfachen.

Abbildung 1-87 zeigt nun die abschließende Antwort, auf die wir hingearbeitet haben, diejenige, die den Anstieg der Fehlerfunktion beschreibt, sodass wir die Gewichte w_{jk} anpassen können.

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

Abbildung 1-87: Anstieg der Fehlerfunktion

Yes! Geschafft!

Das ist der magische Ausdruck, nach dem wir gesucht haben – der Schlüssel, um neuronale Netze zu trainieren.

Ein zweiter Blick auf den Ausdruck ist angebracht, und durch die farbige Codierung sind die einzelnen Elemente deutlich zu unterscheiden. Der erste Teil ist einfach der wohlbekannte Fehler (`Ziel` – `Ist`). Der Summenausdruck innerhalb der `sigmoid()`-Funktionen ist das Signal in den Knoten der letzten Schicht, den wir i_k hätten nennen können, um den Ausdruck nochmals zu vereinfachen. Es handelt sich lediglich um das Signal in einen Knoten, bevor die stauchende Aktivierungsfunktion angewendet wird. Der letzte Teil ist die Ausgabe vom Knoten `j` der vorherigen versteckten Schicht. Es lohnt sich, diese Ausdrücke in diesen Termen zu betrachten, weil Sie damit ein Gefühl dafür bekommen, was physisch in diesen Anstieg – und letztlich in die Verfeinerung der Gewichte – einfließt.

Das ist ein tolles Ergebnis, und wir sollten wirklich mit uns zufrieden sein. Viele Menschen haben echte Schwierigkeiten, bis zu diesem Punkt vorzudringen.

Jetzt bleibt nur noch wenig zu tun. Dieser Ausdruck, an dem wir uns abgemüht haben, dient der Verfeinerung der Gewichte zwischen der versteckten Schicht und der Ausgabeschicht. Wir müssen nun den Job zu Ende bringen und einen ähnlichen Fehleranstieg für die Gewichte zwischen der Eingabeschicht und der versteckten Schicht finden.

Wir könnten das wieder mit viel Algebra bewerkstelligen, doch das müssen wir gar nicht. Wir nutzen einfach die physische Interpretation, die wir eben durchgeführt haben, und bauen erneut einen Ausdruck für den neuen Satz von Gewichten auf, an dem wir interessiert sind.

Dabei ist jetzt zu beachten:

- Der erste Teil, der den Fehler (`Ziel` – `Ist`) angegeben hat, wird jetzt zum rekombinierten backpropagierten Fehler aus den versteckten Knoten, wie wir es oben gesehen haben. Diesen Fehler nennen wir e_j.
- Die Sigmoidteile können gleich bleiben, doch die Summenausdrücke in den `sigmoid()`-Funktionen beziehen sich auf die vorhergehenden Schichten, sodass die Summe über allen Eingängen durch die Gewichte zu einem versteckten Knoten `j` moderiert wird. Wir könnten dies i_j nennen.
- Der letzte Teil ist jetzt die Ausgabe der Knoten der ersten Schicht o_i, was hier die Eingabesignale sind.

Dieser elegante Weg, Unmengen von Arbeit zu vermeiden, profitiert lediglich von der Symmetrie im Problem, um einen neuen Ausdruck zu erzeugen. Wir nennen diese Technik zwar einfach, doch sie ist extrem leistungsfähig und wurde von einigen der großartigsten Mathematiker und Wissenschaftler entwickelt. Sicherlich können Sie Ihre Kollegen damit beeindrucken!

Der zweite Teil der gesuchten endgültigen Antwort – der Anstieg der Fehlerfunktion für die Gewichte zwischen der Eingabeschicht und der versteckten Schicht – sieht also wie in Abbildung 1-88 aus.

$$\frac{\partial E}{\partial w_{ij}} = -(e_j) \cdot sigmoid(\Sigma_i w_{ij} \cdot o_i)(1 - sigmoid(\Sigma_i w_{ij} \cdot o_i)) \cdot o_i$$

Abbildung 1-88: Anstieg der Fehlerfunktion für die Gewichte zwischen Eingabeschicht und versteckter Schicht

Wir verfügen nun über alle diese entscheidenden magischen Ausdrücke für den Anstieg und können sie jetzt einsetzen, um die Gewichte nach jedem Trainingsbeispiel zu aktualisieren.

Wie bereits erläutert und in den Diagrammen weiter oben illustriert, werden die Gewichte in der entgegengesetzten Richtung zum Gradienten geändert. Außerdem moderieren wir die Änderung mithilfe einer Lernrate, die wir auf ein bestimmtes Problem abstimmen können. Wir haben das bereits bei den linearen Klassifizierern kennengelernt, um zu vermeiden, durch schlechte Trainingsbeispiele zu weit in die falsche Richtung gestoßen zu werden. Gleichzeitig haben wir damit sichergestellt, dass wir nicht dauernd um ein Minimum herumpendeln, indem wir es ständig überschreiten. Abbildung 1-89 zeigt, wie das mathematisch ausgedrückt wird.

$$neu\ w_{jk} = alt\ w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

Abbildung 1-89: Mathematischer Ausdruck für die Gewichtsanpassung

Das aktualisierte Gewicht w_{jk} ist das alte Gewicht, das mit dem eben ermittelten negativen Fehleranstieg angepasst wird. Der Fehlerterm wird subtrahiert, weil wir das Gewicht bei einem positiven Anstieg verringern und bei einem negativen Anstieg vergrößern wollen, wie Sie es bereits weiter oben gesehen haben. Der Faktor α moderiert die Stärke dieser Änderungen, um ein Überschwingen zu vermeiden. Oftmals bezeichnet man diesen Faktor als *Lernrate*.

Dieser Ausdruck gilt auch für die Gewichte zwischen der Eingabeschicht und der versteckten Schicht und nur für die Gewichte zwischen versteckter Schicht und Ausgabeschicht. Der Unterschied liegt im Fehlergradienten, für den wir die beiden obigen Ausdrücke haben.

Bevor wir diesen Punkt abhaken können, müssen wir uns erst noch ansehen, wie sich diese Berechnungen als Matrizenmultiplikationen ausführen lassen. Letztlich tun wir das, was wir schon weiter oben getan haben: Wir schreiben im Detail auf, wie jedes Element der Gewichtsänderungsmatrix aussehen soll (siehe Abbildung 1-90).

$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1(1-S_1) \\ E_2 * S_2(1-S_3) \\ E_k * S_k(1-S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} O_1 & O_2 & O_j & \dots \end{pmatrix}$$

Werte der vorherigen Schicht

Werte der nächsten Schicht

Abbildung 1-90: Anpassung der Gewichte per Matrizenmultiplikation

Die Lernrate α habe ich weggelassen, da sie lediglich eine Konstante ist und praktisch nichts daran ändert, wie wir unsere Matrizenmultiplikation organisieren.

Die Matrix der Gewichtsänderungen enthält Werte, die das Gewicht $w_{j,k}$ anpassen, das den Knoten j in der einen Schicht mit dem Knoten k in der nächsten verknüpft. Wie Abbildung 1-90 zeigt, verwendet der erste Teil des Ausdrucks Werte aus der nächsten Schicht (Knoten k) und der letzte Teil des Ausdrucks Werte von der vorherigen Schicht (Knoten j).

Gegebenenfalls müssen Sie sich Abbildung 1-90 erst eine Zeit lang ansehen, um zu erkennen, dass der letzte Teil, die horizontale Matrix mit nur einer einzigen Zeile, die transponierte Matrix der Ausgänge von der vorherigen Schicht $\mathbf{0}_j$ ist. Die Farbcodierung zeigt, dass das Punktprodukt richtig herum geschrieben ist. Wenn Sie zweifeln, schreiben Sie das Punktprodukt doch einfach mit diesen Werten andersherum – Sie werden sehen, es funktioniert nicht.

Abbildung 1-91 zeigt die Matrixform dieser Gewichtsaktualisierung. In einer Programmiersprache, die effizient mit Matrizen umgehen kann, lässt sich diese Form direkt in einem Computerprogramm implementieren.

$$\Delta W_{jk} = \alpha \cdot E_k \cdot O_k(1 - O_k) \cdot O_j^T$$

Abbildung 1-91: Matrixform der Gewichtsaktualisierung

Eigentlich ist dieser Ausdruck überhaupt nicht kompliziert. Die Sigmoidfunktionen sind verschwunden, weil sie einfach die Ausgaben der Knoten 0_k sind.

Das war's! Erledigt.

Kernideen

- Der Fehler eines neuronalen Netzes ist eine Funktion der internen Verknüpfungsgewichte.
- Ein neuronales Netz verbessern bedeutet, diesen Fehler zu verringern – indem diese Gewichte geändert werden.
- Es ist zu schwierig, die richtigen Gewichte direkt auszuwählen. Ein alternativer Ansatz ist, die Gewichte iterativ zu verbessern, indem man in kleinen Schritten die Fehlerfunktion hinabsteigt. Jeder Schritt geht von der aktuellen Position aus in Richtung des größten nach unten weisenden Anstiegs. Diese Methode heißt Verfahren des steilsten Abstiegs oder Gradientenverfahren.
- Der jeweilige Anstieg der Fehlerfunktion lässt sich durch Differenziation berechnen, was nicht zu schwierig ist.

Gewichtsaktualisierung am konkreten Beispiel

Wir wollen eine Reihe von Beispielen anhand von konkreten Zahlen durcharbeiten, allein um die Arbeitsweise der Gewichtsaktualisierung zu veranschaulichen.

Das Netz in Abbildung 1-92 kennen Sie bereits, doch dieses Mal haben wir Beispielwerte für die Ausgaben des ersten versteckten Knotens $o_{j=1}$ und des zweiten versteckten Knotens $o_{j=2}$ angefügt. Die Zahlen wurden zufällig gewählt, um die Methode zu veranschaulichen, und wurden nicht ermittelt, indem Signale von der Eingabeschicht aus durchgeleitet wurden.

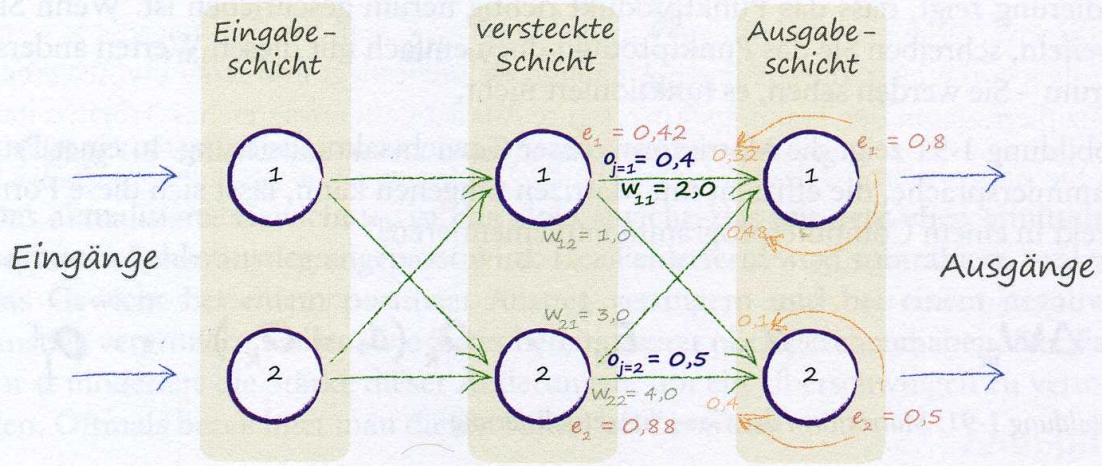


Abbildung 1-92: Beispiel für ein neuronales Netz mit Werten für die Ausgabeknoten der versteckten Schicht

Wir möchten das Gewicht w_{11} zwischen der versteckten Schicht und der Ausgabeschicht aktualisieren. Momentan hat es den Wert 2,0. Abbildung 1-93 gibt noch einmal den ausgeschriebenen Fehleranstieg an.

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

Abbildung 1-93: Der Fehleranstieg in ausgeschriebener Form

Führen wir das nun Stück für Stück durch:

- Der erste Term $(t_k - o_k)$ ist der Fehler $e_1 = 0{,}8$, wie wir ihn zuvor gesehen haben.
- Die Summe in den Sigmoidfunktionen $\Sigma_j w_{jk} o_j$ ist $(2{,}0 \times 0{,}4) + (3{,}0 \times 0{,}5) = 2{,}3$.
- Die Sigmoidfunktion $1/(1 + e^{-2{,}3})$ liefert dann 0,909. Dieser mittlere Ausdruck ergibt $0{,}909 \times (1 - 0{,}909) = 0{,}083$.
- Das letzte Element ist einfach o_j, was $o_{j=1}$ entspricht, weil wir am Gewicht w_{11} interessiert sind, bei dem $j = 1$ ist. Hier hat es einfach den Wert 0,4.

Multipliziert man alle diese drei Terme (das Minuszeichen am Anfang nicht vergessen), ergibt sich –0,0265.

Wenn wir eine Lernrate von 0,1 haben, ergibt das eine Änderung von $-(0{,}1 \times -0{,}0265) = +0{,}00265$. Das neue Gewicht w_{11} ist somit das ursprüngliche Gewicht $2{,}0 + 0{,}00265 = 2{,}00265$.

Dies ist eine recht kleine Änderung, doch über viele Hunderte oder Tausende von Iterationen werden sich die Gewichte schließlich zu einer Konfiguration einpegeln, sodass das gut trainierte neuronale Netz Ausgaben produziert, die den Trainingsbeispielen entsprechen.

Die Daten vorbereiten

In diesem Abschnitt beschäftigen wir uns damit, wie man die Trainingsdaten bestmöglich vorbereitet, die zufälligen Anfangsgewichte festlegt und sogar die Ausgaben konzipiert, damit der Trainingsprozess gute Aussicht auf erfolgreiches Arbeiten hat.

Ja, Sie haben richtig gelesen! Nicht alle Bestrebungen, neuronale Netze einzusetzen, sind von Erfolg gekrönt. Das hat viele Gründe. Einige Ursachen für ein Scheitern lassen sich ausräumen, wenn man genau über die Trainingsdaten nachdenkt, die anfänglichen Gewichte geeignet festlegt und ein zweckmäßiges Schema für die Ausgabe konzipiert. Diese Komponenten sehen wir uns jetzt nacheinander an.

Eingaben

Sehen Sie sich in Abbildung 1-94 den Graphen der Sigmoid-Aktivierungsfunktion an. Hier ist deutlich zu sehen, dass die Aktivierungsfunktion bei großen Eingabewerten sehr flach verläuft.