KAPITEL 2

Erste Modelle

In Kapitel 1 habe ich die Grundbausteine, um Deep Learning überhaupt zu verstehen, beschrieben: verschachtelte, stetige, differenzierbare Funktionen. Ich habe gezeigt, wie diese Funktionen als Rechengraphen dargestellt werden, wobei jeder Knoten des Graphen für eine einfache Funktion steht. Dabei habe ich demonstriert, wie die Ableitung der Ausgabe der verschachtelten Funktion bezogen auf ihre Eingaben berechnet werden kann: Wir werten einfach die Teilableitungen dieser Einzelfunktionen bei der Eingabe aus und multiplizieren die Ergebnisse. Aufgrund der Kettenregel führt das zu einer korrekten Ableitung für die *gesamte* verschachtelte Funktion. Anhand einfacher Beispiele konnten wir sehen, dass dies tatsächlich funktioniert. Als Eingaben wurden NumPy-`ndarrays` verwendet und `ndarrays` als Ausgaben erzeugt.[1]

Ich habe gezeigt, dass dieses Berechnungsverfahren auch mit mehreren `ndarrays` als Eingabe funktioniert, die mithilfe einer *Matrizenmultiplikation* kombiniert werden. Im Gegensatz zu anderen Operationen wurde dabei die Form (siehe Kapitel 1) der Eingaben verändert. Ist die erste Eingabe *(X)* für diese Operation ein `ndarray` der Form B × N und die zweite Eingabe (*M*) ein `ndarray` der Form N × M, dann ist die Ausgabe *P* der Operation ein `ndarray` der Form B × M.

Obwohl wir die Ableitung dieser Operation nicht kennen, konnte ich zeigen, dass die Ableitung auch dann über einen einfachen Ausdruck berechnet werden kann, wenn eine verschachtelte Funktion eine Matrizenmultiplikation $v(X, W)$ als Einzeloperation enthält. Hierbei kann insbesondere die Rolle von $\frac{\partial v}{\partial u}(W)$ durch X^T und die Rolle von $\frac{\partial v}{\partial u}(X)$ durch W^T ersetzt werden.

In diesem Kapitel beginnen wir, diese Konzepte in echte Applikationen zu übersetzen. Im Einzelnen werden wir:

1. lineare Regression mithilfe unserer Grundbausteine ausdrücken,

1 `ndarray` steht für ein n-dimensionales Array, also ein Array mit einer beliebigen Anzahl von Dimensionen. Ein Datentyp aus der Python-Programmierbibliothek NumPy, den wir im in diesem Buch immer wieder verwenden werden.

2. zeigen, dass unsere Überlegungen zu den Ableitungen aus Kapitel 1 es uns ermöglichen, ein lineares Regressionsmodell zu trainieren, sowie
3. dieses Modell (ebenfalls anhand der Grundbausteine) zu einem neuronalen Netz mit einer Schicht erweitern.

Diese Schritte werden es uns in Kapitel 3 erleichtern, die gleichen Bausteine für die Erstellung von Deep-Learning-Modellen einzusetzen.

Zuvor wollen wir uns aber einen Überblick über das *überwachte Lernen (Supervised Learning*, einen Teilbereich des Machine Learning*)* verschaffen. Das soll uns dabei helfen, zu verstehen, wie neuronale Netze zur Problemlösung eingesetzt werden können.

Überblick über das überwachte Lernen

Allgemein kann man das Machine Learning als die Erstellung von Algorithmen beschreiben, die *Beziehungen* in Daten entdecken oder »lernen« können. Das überwachte Lernen ist ein Teilbereich des Machine Learning, der sich mit dem Aufspüren von Beziehungen zwischen *Eigenschaften der Daten* beschäftigt, *die bereits gemessen wurden*.[2]

In diesem Kapitel geht es um ein typisches Problem des überwachten Lernens, das auch in der wahren Welt auftreten kann: das Auffinden der Beziehung zwischen den Eigenschaften eines Hauses und dessen Wert. Sicher gibt es irgendeine Beziehung zwischen der Anzahl der Räume, der Wohnfläche oder der Nähe zu Schulen und dem Wunsch, in diesem Haus zu wohnen oder es zu besitzen. Auf abstrakter Ebene liegt das Ziel des überwachten Lernens darin, eben diese Beziehungen aufzudecken. Hierfür müssen diese Eigenschaften *bereits ermittelt bzw. gemessen* worden sein.

Mit »messen« meine ich hier eine präzise Definition dieser Eigenschaften und die Möglichkeit, sie durch eine Zahl auszudrücken. Viele Eigenschaften eines Hauses, wie die Anzahl der Schlafzimmer, die Wohnfläche etc., lassen sich leicht als Zahlen formulieren. Bei anderen Formen der Information, beispielsweise einer Beschreibung des Hauses in natürlicher Sprache von TripAdvisor, wäre dieses Problem deutlich schwerer zu lösen. Ohne eine nachvollziehbare Übersetzung dieser kaum strukturierten Daten in Zahlen wäre es fast unmöglich, diese Beziehungen zu finden. Außerdem müssen wir für die Beschreibung der einzelnen Konzepte, wie den Wert des Hauses, eine passende Zahl zuordnen. Um den Wert des Hauses auszudrücken, bietet sich natürlich der Preis als Zahl an.[3]

2 Die andere Art des Machine Learning, das *unüberwachte Lernen*, kann man sich vorstellen als das Finden von Beziehungen zwischen Dingen, die Sie gemessen haben, und Dingen, die noch nicht gemessen wurden. Zusätzliche Informationen hierzu finden Sie auch in Kapitel 7 im Abschnitt »Nachtrag: Unüberwachtes Lernen mit Autoencodern« auf Seite 212.

3 In der realen Welt wäre selbst die Wahl des Preises nicht offensichtlich: Wäre es der Preis, für den das Haus das letzte Mal verkauft wurde? Wie sieht es bei einem Haus aus, das lange Zeit nicht zum Verkauf stand? In diesem Buch verwenden wir Beispiele, in denen die Daten offensichtlich sind oder bereits für Sie entschieden wurden. In der wahren Welt kann eine korrekte Ermittlung dagegen ziemlich schwierig sein.

Nachdem wir die Eigenschaften in Zahlen übersetzt haben, müssen wir entscheiden, mit welcher Struktur wir diese darstellen wollen. Im Machine Learning wird hierfür üblicherweise eine Form verwendet, die zudem die Berechnungen vereinfacht. Hierbei wird jeder Zahlensatz für eine einzelne *Beobachtung*, z. B. ein einzelnes Haus, als *Datenzeile* dargestellt. Diese Zeilen werden dann »übereinandergestapelt«, wobei die *Datenstapel* (Batches) schließlich in Form zweidimensionaler `ndarrays` an unsere Modelle übergeben werden. Die Modelle geben anschließend ihre Vorhersagen als Ausgabe-`ndarrays` zurück. Dabei steht jede Vorhersage für eine Beobachtung wiederum in einer Zeile. Die Zeilen werden – wie bei der Eingabe – zu *Batches* zusammengefasst.

Jetzt benötigen wir ein paar Definitionen: Wir sagen, dass die Länge jeder Zeile in diesem `ndarray` der Anzahl der *Merkmale* (Features) unserer Daten entspricht. Allgemein gesagt, kann jede Eigenschaft aus mehreren Merkmalen bestehen. Ein klassisches Beispiel wäre eine Eigenschaft, die unsere Daten als Teil bestimmter *Kategorien* beschreibt, zum Beispiel: ein rotes Backsteinhaus, ein braunes Backsteinhaus, ein mit Schiefer verkleidetes Haus.[4] Diese Eigenschaft könnten wir mit drei Merkmalen beschreiben. Die Abbildung von Eigenschaften unserer Beobachtungen auf bestimmte Merkmale bezeichnet man als *Feature Engineering (*Merkmalskonstruktion)[5]. Ich werde in diesem Buch nur am Rande auf diesen Prozess eingehen. Das Beispiel in diesem Kapitel verwendet für jede Beobachtung 13 numerisch ausgedrückte Merkmale.

Wie gesagt, es geht beim überwachten Lernen letztendlich darum, die Beziehungen zwischen bestimmten Dateneigenschaften herauszufinden. In der Praxis wählen wir hierfür die Eigenschaft aus, für die wir Vorhersagen treffen wollen. Diese nennen wir unser *Ziel* (Target). Die Auswahl des Ziels ist vollkommen beliebig und hängt vom zu lösenden Problem ab. Wollen Sie beispielsweise die Beziehung zwischen Hauspreisen und der Anzahl der Räume beschreiben, können Sie dafür ein Modell mit den Hauspreisen als Ziel trainieren und die Anzahl der Räume als Merkmal definieren – oder umgekehrt. In beiden Fällen beschreibt das resultierende Modell die Beziehung dieser zwei Eigenschaften. Dadurch können Sie beispielsweise sagen, dass eine höhere Anzahl von Räumen den Hauspreis erhöht. Wollen Sie jedoch die Hauspreise vorhersagen, für die keine Preisinformationen zur Verfügung stehen, müssen Sie den Preis als Ziel wählen. Dadurch können Sie das Modell nach dem Training auch mit anderen Informationen füttern.

Abbildung 2-1 zeigt die Hierarchie der Beschreibungen des überwachten Lernens. Auf oberster Ebene geht es um das Finden der Beziehungen in Daten; ganz unten werden die Daten durch das Training von Modellen ausgewertet, um numerische Beziehungen zwischen den Merkmalen und dem Ziel zu entdecken.

4 Die meisten von Ihnen wissen wahrscheinlich, dass diese Merkmale auch als »kategorische« Merkmale bezeichnet werden.

5 Mehr zur Merkmalskonstruktion finden Sie beispielsweise in dem Buch *Merkmalskonstruktion für Machine Learning* von Alice Zheng und Amanda Casari (O'Reilly). *https://www.oreilly.de/buecher/13279/9783960090939-merkmalskonstruktion-für-machine-learning.html*.

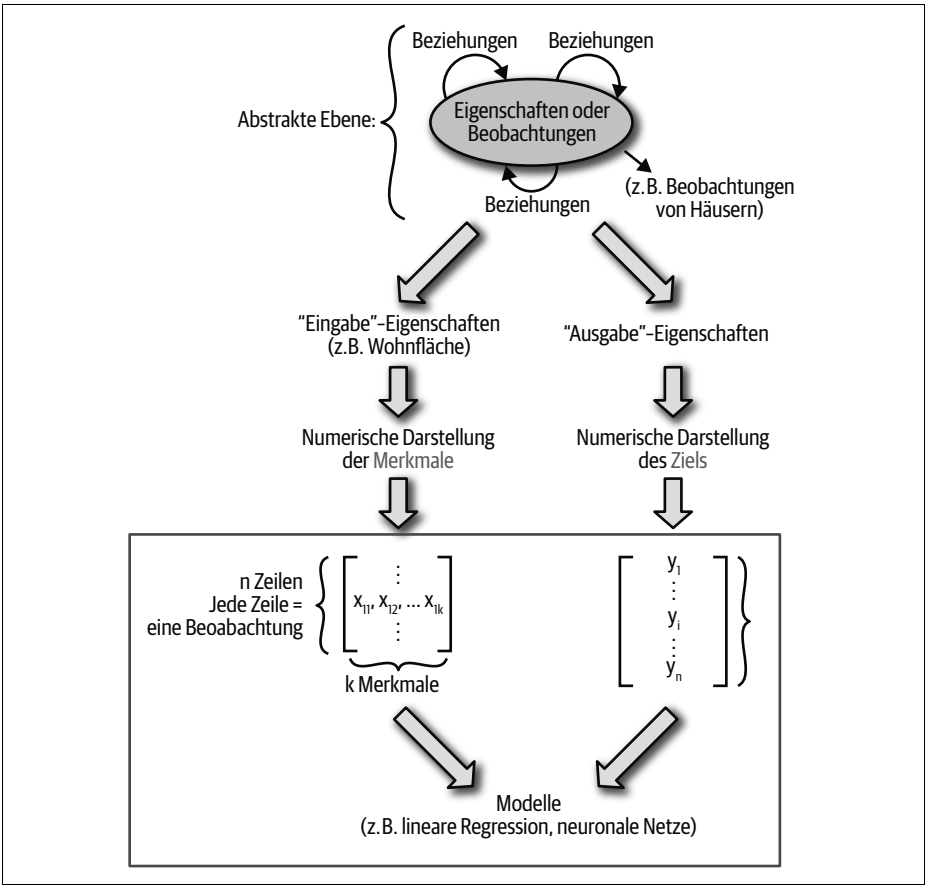

Abbildung 2-1: Überblick über das überwachte Lernen

Wie gesagt, wir werden fast die gesamte Zeit auf der unten in der Grafik (aus Abbildung 2-1) hervorgehobenen Ebene verbringen. Dennoch ist es bei vielen Problem deutlich schwieriger, die Elemente im oberen Teil, also das Sammeln der richtigen Daten, die Definition des zu lösenden Problems und die Merkmalskonstruktion, richtig einzurichten als die eigentliche Modellierung. Da es in diesem Buch jedoch um das Modelling geht – genauer gesagt, um das Verständnis der Funktionsweise von Deep-Learning-Modellen –, wollen wir uns jetzt wieder diesem Thema zuwenden.

Modelle für das überwachte Lernen

Damit haben Sie jetzt einen allgemeinen Überblick über die Verwendung der Modelle des überwachten Lernens. Wie gesagt, diese Modelle sind im Prinzip nichts anderes als verschachtelte mathematische Funktionen. Anhand der Überlegungen aus dem vorherigen Kapitel kann ich das Ziel des überwachten Lernens

jetzt mathematisch und als Code (die Diagramme kommen etwas später) präziser formulieren: Das Ziel des überwachten Lernens besteht darin, eine bestimmte Funktion zu finden. Als *Eingabe* der Funktion verwenden wir ein `ndarray`, das die gelernten Merkmale enthält. Die *Ausgabe* soll ein weiteres `ndarray` sein, dessen Werte eine »gewisse Nähe« zu dem `ndarray` besitzen, das unser Ziel enthält. Die Funktion muss also in der Lage sein, *die Eigenschaften unserer Beobachtungen auf das Ziel abzubilden*.

Hierfür stellen wir unsere Daten als Matrix X mit n Zeilen dar. Jede Zeile steht für eine Beobachtung mit k numerischen Merkmalen. Jede Zeile mit Beobachtungen ist ein Vektor, z. B. $x_i = [x_{i1}\ x_{i2}\ x_{i3}\ \ldots\ x_{ik}]$. Diese Beobachtungen werden als Batch übereinandergeschichtet. Ein Batch der Größe 3 sähe beispielsweise so aus:

$$X_{batch} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \ldots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \ldots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \ldots & x_{3k} \end{bmatrix}$$

Für jeden Batch mit *Beobachtungen* gibt es einen entsprechenden Batch mit *Zielen*. Dabei entspricht jedes Element dem Zielwert für die jeweilige Beobachtung. Das können wir als eindimensionalen Vektor ausdrücken:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Anhand der Werkzeuge aus dem vorherigen Kapitel wollen wir für dieses Array eine Funktion erstellen, die als Eingabe einen Batch (oder mehrere) mit Beobachtungen der Struktur X_{batch} übernimmt. Sie erzeugt Vektoren aus Werten p_i, die wir als »Vorhersagen« interpretieren (bezogen auf unseren Datensatz X), die den Zielwerten y_i möglichst »nah« sein sollen.

Und damit können unser erstes Modell für einen realen Datensatz erstellen. Wir beginnen mit einer einfachen Form, der *linearen Regression,* und zeigen, wie diese mithilfe unserer Grundbausteine ausgedrückt werden kann.

Lineare Regression

Lineare Regression wird oft darstellt als:

$$y_i = \beta_0 + \beta_1 \times x_1 + \ldots + \beta_n \times x_k + \epsilon$$

Diese Darstellung beschreibt mathematisch unsere Annahme, dass der numerische Wert jedes Ziels eine lineare Kombination der k Merkmale von X ist. Zu diesem wird der Term β_0 addiert, um den Wert der »Grundlinie« der Vorhersage anzupas-

sen (d.h. die Vorhersage, die getroffen wird, wenn alle Merkmale den Wert 0 haben). Diese »Verschiebung« wird üblicherweise als *Bias*-Term bezeichnet. (Wir verwenden in diesem Buch die Begriffe »Verschiebung« und »Bias« gleichbedeutend.) Details zur Verwendung des Bias finden Sie im Abschnitt »Den Bias-Term einbauen« auf Seite 49.

Um das programmieren zu können und ein solches Modell zu »trainieren«, müssen wir es in die Sprache der Funktionen aus Kapitel 1 übertragen. Und das geht am besten anhand eines Diagramms.

Lineare Regression: ein Diagramm

Wie lässt sich die lineare Regression als Rechengraph darstellen? Wir könnten ihn in seine Einzelteile zerlegen, wobei jedes x_i mit einem anderen Element w_i multipliziert wird, und dann die Ergebnisse addieren, wie in Abbildung 2-2 gezeigt.

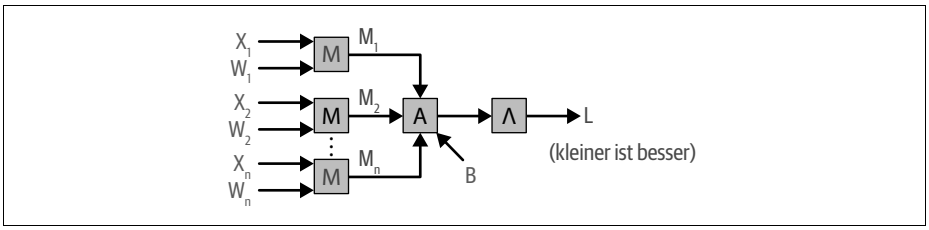

Abbildung 2-2: Die Operation einer linearen Regression, gezeigt auf der Schicht einzelner Multiplikationen und Additionen

Aus Kapitel 1 wissen wir, dass wir die Funktion vereinfachen können, indem wir diese Operationen als Matrizenmultiplikation ausdrücken. Dabei wird die Ableitung der Ausgabe bezogen auf die Eingaben immer noch korrekt berechnet, und wir können unser Modell trainieren.

Sehen wir uns zunächst ein einfaches Beispiel an. Hierbei verwenden wir keinen Term für den Bias (β_0). Wir können die Ausgabe eines linearen Regressionsmodells als Skalarprodukt jedes Beobachtungsvektors darstellen: $x_i = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_k \end{bmatrix}$, wobei wir einen weiteren Vektor mit Parametern verwenden, den wir W nennen:

$$W = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix}$$

Unsere Vorhersage wäre dann einfach:

$$p_i = x_i \times W = w_1 \times x_{i1} + w_2 \times x_{i2} + \dots + w_k \times x_{ik}$$

Das heißt, wir können die »Erzeugung der Vorhersagen« für eine lineare Regression einfach als *Skalarprodukt* darstellen.

Um Vorhersagen für einen Batch an Beobachtungen zu treffen, können wir die lineare Regression mit einer weiteren Einzeloperation verwenden: der Matrizenmultiplikation. Hier ein Beispiel für einen Batch der Größe 3:

$$X_{batch} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3k} \end{bmatrix}$$

Führen wir die Matrizenmultiplikation von X_{batch} mit W durch, erhalten wir wie gewünscht einen Vektor mit Vorhersagen für diesen Batch:

$$p_{batch} = X_{batch} \times W = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3k} \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix} =$$

$$\begin{bmatrix} x_{11} \times w_1 + x_{12} \times w_2 + x_{13} \times w_3 + \dots + x_{1k} \times w_k \\ x_{21} \times w_1 + x_{22} \times w_2 + x_{23} \times w_3 + \dots + x_{2k} \times w_k \\ x_{31} \times w_1 + x_{32} \times w_2 + x_{33} \times w_3 + \dots + x_{3k} \times w_k \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Mithilfe der Matrizenmultiplikation können wir also Vorhersagen für einen Batch mit Beobachtungen in einer linearen Regression treffen. Im folgenden Abschnitt zeige ich Ihnen, wie Sie das zusammen mit unseren Grundbausteinen verwenden können, um unser Modell zu trainieren.

Das Modell trainieren

Was bedeutet es, ein Modell zu »trainieren«?

Allgemein formuliert, kombinieren die Modelle die übergebenen Daten[6] mit bestimmten Parametern und erzeugen daraus Vorhersagen. Unser lineares Regressionsmodell übernimmt beispielsweise die Daten X und die Parameter W und erzeugt daraus anhand einer Matrizenmultiplikation die Vorhersagen p_{batch}:

$$p_{batch} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

6 Jedenfalls die in diesem Buch verwendeten.

Um unser Modell zu trainieren, müssen wir außerdem wissen, ob unsere Vorhersagen zutreffen oder nicht. Um das herauszufinden, verwenden wir den Vektor mit den Zielen y_{batch}, bezogen auf den Batch unserer Beobachtungen X_{batch}. Wir berechnen eine einzelne Zahl, die eine Funktion aus y_{batch} und p_{batch} ist. Sie steht für die »Strafe« des Modells für die angestellten Vorhersagen (mehr dazu im folgenden Abschnitt). Eine gute Wahl ist hier die *mittlere quadratische Abweichung* (Mean Squared Error, MSE) – das ist einfach das Quadrat des Durchschnittswerts, um den unsere Vorhersagen »danebenlagen«.

$$MSE(p_{batch}, y_{batch}) = MSE\left(\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}\right) = \frac{(y_1 - p_1)^2 + (y_2 - p_2)^2 + (y_3 - p_3)^2}{3}$$

Das Ermitteln dieser *Abweichung* (engl. Loss, daher hier als L bezeichnet), ist äußerst wichtig. Sobald wir diese Zahl haben, können wir alle Techniken aus Kapitel 1 verwenden, um den *Gradienten* dieser Zahl, bezogen auf jedes Element von W, zu berechnen. Danach können wir diese Ableitungen benutzen, um jedes Element von W in der Richtung zu aktualisieren, die eine Verringerung von L zur Folge hat. Durch vielfaches Wiederholen dieser Prozedur hoffen wir, dass sich unser Modell immer weiter dem formulierten Ziel annähert. *Genau das bedeutet es, unser Modell zu trainieren*. Wie wir sehen werden, funktioniert das tatsächlich! Um zu zeigen, wie die Gradienten berechnet werden, stellen wir die lineare Regression als Rechengraph dar.

Lineare Regression: ein hilfreicheres Diagramm (und die dazugehörige Mathematik)

Abbildung 2-3 zeigt, wie man die lineare Regression als Diagramm ausdrücken kann:

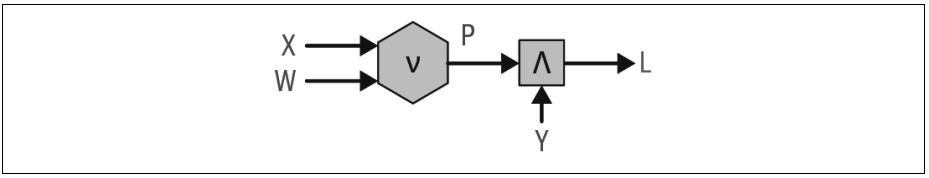

Abbildung 2-3: Die Gleichungen der linearen Regression, ausgedrückt als Rechengraph. X, W und Y sind die Dateneingaben an die Funktion, W steht für die Gewichtungen (weights).

Auch diese Darstellung steht für eine verschachtelte mathematische Funktion. Um sie zu verfeinern, können wir die Abweichung L wie folgt berechnen:

$$L = \Lambda((v(X, W), Y)$$

Den Bias-Term einbauen

Das Diagramm verdeutlicht, wie wir dem Modell einen Bias hinzufügen können. Hierfür wird es um einen weiteren Schritt ergänzt, der einen sogenannten *Bias-Term* (Verschiebung, im Diagramm als *B* bezeichnet) addiert (siehe Abbildung 2-4).

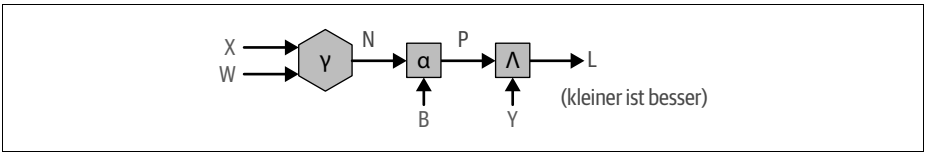

Abbildung 2-4: Der Rechengraph einer linearen Regression, bei dem ein Bias-Term hinzugefügt wurde

Bevor wir das in Code umsetzen, sollten wir mathematisch darüber nachdenken, was hier passiert. Mit dem hinzugefügten Bias-Term besteht jedes Element der Vorhersage unseres Modells p_i aus dem bereits beschriebenen Skalarprodukt, zu dem der Wert von b addiert wurde:

$$p_{batch\_with\_bias} = x_i \text{dot} W + b = \begin{bmatrix} x_{11} \times w_1 + x_{12} \times w_2 + x_{13} \times w_3 + \ldots + x_{1k} \times w_k + b \\ x_{21} \times w_1 + x_{22} \times w_2 + x_{23} \times w_3 + \ldots + x_{2k} \times w_k + b \\ x_{31} \times w_1 + x_{32} \times w_2 + x_{33} \times w_3 + \ldots + x_{3k} \times w_k + b \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

In der linearen Regression wird jeder Beobachtung in der Bias-Operation *die gleiche Zahl* hinzuaddiert. Wir werden weiter unten in diesem Kapitel genauer darauf eingehen, was das bedeutet.

Lineare Regression: der Code

Jetzt können wir die Einzelteile zusammenfügen und die Funktion programmieren, die anhand von Batches mit Beobachtungen X_{batch} und den dazugehörigen Zielen y_{batch} Vorhersagen trifft und Abweichungen berechnet. Vergessen Sie dabei nicht, dass die Berechnung von Ableitungen für verschachtelte Funktionen anhand der Kettenregel zwei Schrittfolgen benötigt: Zuerst führen wir die Vorwärtspropagation durch, bei der die Eingaben schrittweise durch eine Reihe von Operationen geleitet wird und die berechneten Werte gespeichert werden. Danach verwenden wir diese Werte in der Rückwärtspropagation, um die entsprechenden Ableitungen zu berechnen.

Der folgende Code speichert die berechneten Werte aus der Vorwärtspropagation in einem Python-Dictionary. Um sie von den eigentlichen Parametern unterscheiden zu können (die wir ebenfalls für die Rückwärtspropagation brauchen), erwartet unsere Funktion die Übergabe eines Dictionarys, das die Parameter enthält:

```
def forward_linear_regression(X_batch: ndarray,
                              y_batch: ndarray,
                              weights: Dict[str, ndarray])
                              -> Tuple[float, Dict[str, ndarray]]:
    '''
    Vorwärtspropagation für die schrittweise durchgeführte lineare Regression.
    '''
    # Zusichern, dass die Batch-Größen von x und y gleich sind.
    assert X_batch.shape[0] == y_batch.shape[0]

    # Zusichern, dass die Matrizenmultiplikation funktioniert.
    assert X_batch.shape[1] == weights['W'].shape[0]

    # Zusichern, dass B einfach ein ndarray der Größe 1x1 ist.
    assert weights['B'].shape[0] == weights['B'].shape[1] == 1

    # Die Operationen der Vorwärtspropagation ausführen.
    N = np.dot(X_batch, weights['W'])

    P = N + weights['B']

    loss = np.mean(np.power(y_batch - P, 2))

    # Die in der Vorwärtspropagation berechneten Informationen speichern.
    forward_info: Dict[str, ndarray] = {}
    forward_info['X'] = X_batch
    forward_info['N'] = N
    forward_info['P'] = P
    forward_info['y'] = y_batch

    return loss, forward_info
```

Damit haben wir alle Teile beisammen, um dieses Modell zu trainieren. Im nächsten Abschnitt zeigen wir im Detail, was das bedeutet und wie es funktioniert.

Das Modell trainieren

Jetzt werden wir die Grundbausteine aus dem vorherigen Kapitel benutzen, um $\frac{\partial L}{\partial w_i}$ für jedes w_i in W sowie $\frac{\partial L}{\partial b}$ zu berechnen. Nachdem unsere Eingaben in der Vorwärtspropagation durch eine Reihe verschachtelter Funktionen geleitet wurden, müssen wir in der Rückwärtspropagation die Teilableitungen der einzelnen Funktionen berechnen. Hierfür werten wir die Ableitungen an ihren Eingaben aus und multiplizieren sie miteinander. Trotz der Matrizenmultiplikation können wir anhand der Überlegungen aus dem vorherigen Kapitel damit umgehen.

Die Gradienten berechnen: ein Diagramm

Vom Konzept her brauchen wir in etwa das in Abbildung 2-5 Gezeigte.

Hier bewegen wir uns einfach rückwärts durch den Regengraphen. Dabei berechnen wir die Ableitungen der Einzelfunktionen und werten diese an den Eingaben

aus, die die Funktionen in der Vorwärtspropagation erhalten haben. Zum Schluss multiplizieren wir diese Ableitungen miteinander. Das ist nicht weiter schwer, also wenden wir uns nun den Details zu.

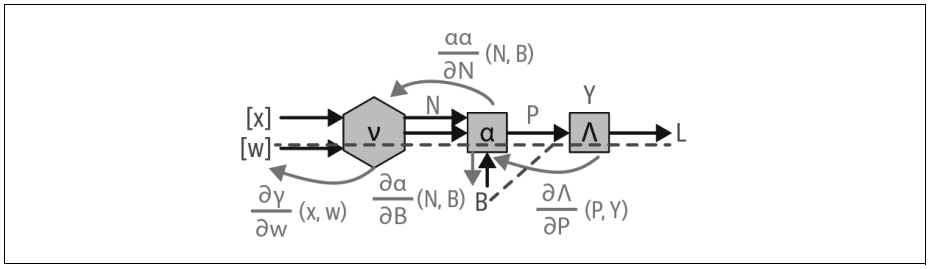

Abbildung 2-5: Die Rückwärtspropagation durch den Rechengraphen der linearen Regression

Die Gradienten berechnen: die Mathematik (und etwas Code)

In Abbildung 2-5 sehen Sie, dass das Ableitungsprodukt, das wir eigentlich berechnen wollen, so aussieht:

$$\frac{\partial \Lambda}{\partial P}(P, Y) \times \frac{\partial \alpha}{\partial N}(N, B) \times \frac{\partial v}{\partial W}(X, W)$$

Es gibt drei Bestandteile, die wir nacheinander berechnen wollen:

Zuerst haben wir $\frac{\partial \Lambda}{\partial P}(P, Y)$. Weil $\Lambda(P, Y) = (Y - P)^2$ für jedes Element in Y und P, erhalten wir:

$$\frac{\partial \Lambda}{\partial P}(P, Y) = -1 \times (2 \times (Y - P))$$

Wir greifen hier etwas vor. Der Programmiercode ist leichter verständlich:

```
dLdP = -2 * (Y - P)
```

Als Nächstes haben wir einen Ausdruck mit Matrizen: $\frac{\partial \alpha}{\partial N}(N, B)$. Aber weil α eine einfache Addition ist, gilt auch hier die Logik, die wir anhand von Zahlen im vorherigen Kapitel hergeleitet haben: Die Erhöhung eines Elements von N um eine Einheit erhöht $P = \alpha(N, B) = N + B$ ebenfalls um eine Einheit. Daher ist $\frac{\partial \alpha}{\partial N}(N, B)$ eine Matrix aus +1+s mit der gleichen Form wie N.

Dieser Ausdruck sähe im Programmiercode so aus:

```
dPdN = np.ones_like(N)
```

Und dann haben wir noch $\frac{\partial v}{\partial W}(X, W)$. Aufgrund unserer Erkenntnisse über die Matrizenmultiplikation aus dem vorherigen Kapitel könnten wir sagen, dass:

$$\frac{\partial v}{\partial W}(X, W) = X^T$$

was im Code schlicht so aussähe:

```
dNdW = np.transpose(X, (1, 0))
```

Das Gleiche machen wir mit dem Bias-Term. Da wir ihn einfach nur addieren, lautet die Teilableitung des Bias-Terms, bezogen auf die Ausgabe, einfach 1:

```
dPdB = np.ones_like(weights['B'])
```

Im letzten Schritt müssen wir die Bestandteile miteinander multiplizieren. Dabei müssen wir sicherstellen, dass wir die korrekte Reihenfolge für die Matrizenmultiplikationen mit `dNdW` und `dNdX` einhalten.

Die Gradienten berechnen: der (vollständige) Code

Unser Ziel besteht darin, alle Ergebnisse der Berechnungen und Eingaben aus der Vorwärtspropagation – zu denen nach dem Diagramm in Abbildung 2-5 X, W, N, B, P und y gehören – zu verwenden, um $\frac{\partial \Lambda}{\partial W}$ und $\frac{\partial \Lambda}{\partial B}$ zu berechnen. Der folgende Code übernimmt W und B als Eingaben in einem Dictionary namens `weights`, während die übrigen Werte in einem Dictionary namens `forward_info` gespeichert werden:

```
def loss_gradients(forward_info: Dict[str, ndarray],
                   weights: Dict[str, ndarray]) -> Dict[str, ndarray]:
    '''
    dLdW und dLdB für das schrittweise lineare
    Regressionsmodell berechnen.
    '''
    batch_size = forward_info['X'].shape[0]

    dLdP = -2 * (forward_info['y'] - forward_info['P'])

    dPdN = np.ones_like(forward_info['N'])

    dPdB = np.ones_like(weights['B'])

    dLdN = dLdP * dPdN

    dNdW = np.transpose(forward_info['X'], (1, 0))

    # Hier muss eine Matrizenmultiplikation verwendet werden, wobei
    # dNdW auf der linken Seite steht
    # (siehe Hinweis am Ende des vorigen Kapitels).
    dLdW = np.dot(dNdW, dLdN)

    # Hier muss die Summe entlang der Dimension gebildet werden,
    # die für die Batchgröße steht
    # (siehe Hinweis am Ende des vorigen Kapitels).
    dLdB = (dLdP * dPdB).sum(axis=0)

    loss_gradients: Dict[str, ndarray] = {}
    loss_gradients['W'] = dLdW
    loss_gradients['B'] = dLdB

    return loss_gradients
```

Wir können die Ableitungen also einfach für jede Operation berechnen und dann miteinander multiplizieren. Dabei müssen wir darauf achten, dass wir die Matrizenmultiplikation in der richtigen Reihenfolge durchführen.[7] Wie wir gleich sehen, funktioniert das tatsächlich, was mit dem Wissen um die Kettenregel aus dem vorherigen Kapitel keine allzu große Überraschung sein sollte.

Noch ein Implementierungsdetail zu den Abweichungsgradienten: Wir speichern die Gradienten in einem Dictionary. Dabei dienen die Namen der Gewichtungen als Schlüssel. Die Werte, um die eine Erhöhung dieser Gewichtungen sich auf die Abweichungen auswirken, verwenden wir als Werte des Dictionarys. Das `weights`-Dictionary ist auf die gleiche Weise strukturiert. Daher iterieren wir über die Gewichtungen in unserem Modell wie folgt:

```
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

Verwendeten wir hier eine andere Datenstruktur als das Dictionary, würden wir ebenfalls darüber iterieren, die Werte aber anders auslesen.

Die Gradienten verwenden, um das Modell zu trainieren

Und jetzt wiederholen wir die folgende Prozedur einfach immer wieder:

1. Wähle einen Batch aus.
2. Führe die Vorwärtspropagation des Modells aus.
3. Führe die Rückwärtspropagation des Modells aus und verwende dabei die in der Vorwärtspropagation berechneten Informationen.
4. Verwende die in der Rückwärtspropagation berechneten Gradienten, um die Gewichtungen zu aktualisieren.

Auf der Website zu diesem Buch finden Sie ein Jupyter Notebook (*https://oreil.ly/2TDV5q9*). Es enthält eine `train`-Funktion, die den Code für die obigen Schritte implementiert. Sie besitzt zudem ein paar sinnvolle Extras, wie das Mischen der Daten, um sicherzustellen, dass sie in zufälliger Reihenfolge übergeben werden. Hier die wichtigsten Zeilen, die in einer `for`-Schleife wiederholt werden:

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)

# 'weights' und 'loss_grads' verwenden die gleichen Schlüssel:
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

7 Zusätzlich dazu müssen wir `dLdB` entlang der Achse 0 summieren. Diesen Schritt erklären wir weiter unten in diesem Kapitel.

Dann führen wir `train` für eine bestimmte Zahl von Zyklen (auch als Epochen [engl. Epoch] bezeichnet, mehr dazu im folgenden Kapitel) über den gesamte Trainingsdatensatz durch:

```
train_info = train(X_train, y_train,
                   learning_rate = 0.001,
                   batch_size=23,
                   return_weights=True,
                   seed=80718)
```

Die `train`-Funktion gibt ein Python-Tupel (`Tuple`) namens `train_info` zurück. Es enthält die Parameter oder *Gewichtungen*, die wiedergeben, was das Modell gelernt hat.

Die Begriffe *Parameter* und *Gewichtungen* (auch Gewicht, engl. Weight) werden im Deep Learning gleichbedeutend verwendet. Daher tun wir das in diesem Buch ebenfalls.

Das Modell bewerten: Trainingsdaten oder Testdaten?

Um zu verstehen, ob unser Modell Beziehungen in unseren Daten gefunden hat, brauchen wir einige Fachbegriffe und Konzepte aus der Statistik. Wir stellen uns jeden Datensatz als *Stichprobe* einer *Population* vor. Unser Ziel besteht grundsätzlich darin, ein Modell zu finden, das Beziehungen in der Population entdeckt, obwohl wir nur eine Stichprobe sehen.

Dabei besteht immer die Gefahr, dass wir ein Modell erstellen, das Beziehungen entdeckt, die zwar in der Stichprobe existieren, nicht aber in der Population. In unserer Stichprobe kann es beispielsweise sein, dass gelbe Klinkerhäuser mit drei Badezimmern relativ preiswert sind. Ein kompliziertes neuronales Netzmodell könnte diese vermeintliche Beziehung entdecken, obwohl sie in der Population überhaupt nicht existiert. Dieses Phänomen wird als *Überanpassung (*Overfitting*)* bezeichnet. Wie können wir herausfinden, ob unsere Modellstruktur möglicherweise auch dieses Problem hat?

Die Lösung besteht darin, unsere Stichprobe in einen *Trainingsdatensatz* und einen *Testdatensatz* aufzuteilen. Anhand der Trainingsdaten wird das Modell trainiert (d.h., die Gewichtungen werden iterativ aktualisiert). Danach überprüfen wir es anhand der Testdaten, um seine tatsächliche Leistungsfähigkeit einschätzen zu können.

War unser Modell in der Lage, Beziehungen zu finden, die eine Verallgemeinerung des *Trainingsdatensatzes* bezogen auf den Rest der *Stichprobe* darstellt (unser gesamter Datensatz), ist es wahrscheinlich, dass die gleiche »Modellstruktur« als Verallgemeinerung unserer *Stichprobe*, also unseres gesamten Datensatzes bezogen auf die *Population,* angesehen werden kann. Genau das wollen wir.

Das Modell bewerten: der Code

Mit diesem Verständnis können wir unser Modell anhand der Testdaten überprüfen. Zuerst schreiben wir eine Funktion, um Vorhersagen zu treffen. Hierfür verkürzen wir die bereits bekannte `forward_pass`-Funktion:

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]):
    '''
    Aus dem schrittweisen linearen Regressionsmodell Vorhersagen erzeugen.
    '''
    N = np.dot(X, weights['W'])

    return N + weights['B']
```

Danach verwenden wir die zuvor aus der `train`-Funktion zurückgegebenen Gewichtungen und schreiben:

```
preds = predict(X_test, weights)  # weights = train_info[0]
```

Wie aussagekräftig sind diese Vorhersagen? Wir sollten nicht vergessen, dass unser scheinbar seltsamer Ansatz, unsere Modelle als eine Reihe von Operationen zu definieren, noch nicht bestätigt ist. Diese werden durch wiederholtes Anpassen der Gewichtungen (bzw. Parameter) »trainiert«. Für die Anpassung verwenden wir die Teilableitungen der anhand der Kettenregel berechneten Abweichungen. Dabei ist aber überhaupt noch nicht klar, ob das tatsächlich funktioniert.

Um das zu überprüfen, können wir die Vorhersage des Modells zunächst einmal grafisch darstellen. Die x-Achse steht für die Vorhersagen, die y-Achse für die tatsächlichen Werte. Lägen alle Punkte exakt auf der 45-Grad-Linie, wäre das Modell perfekt. Abbildung 2-6 zeigt eine grafische Darstellung der von unserem Modell vorhergesagten und der tatsächlichen Werte.

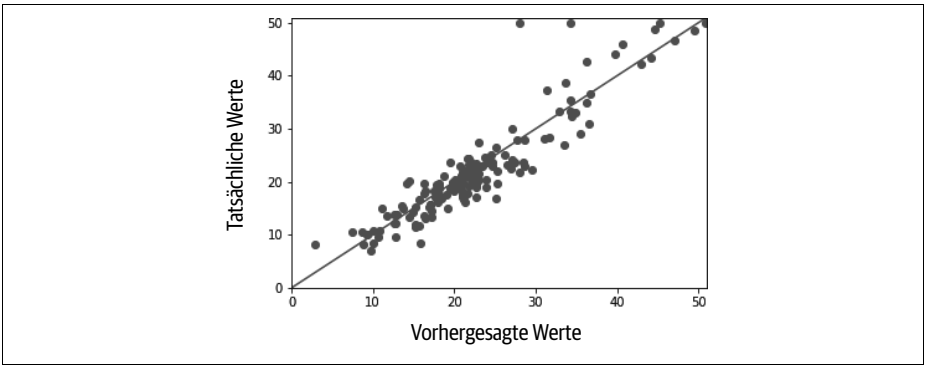

Abbildung 2-6: Vorhergesagte und tatsächliche Werte für unser selbst entwickeltes lineares Regressionsmodell

Das sieht doch schon recht gut aus. Trotzdem wollen wir möglichst genau ermitteln, wie gut unser Modell wirklich ist. Dafür gibt es verschiedene Wege:

- Den durchschnittlichen Abstand in absoluten Werten zwischen den Vorhersagen unseres Modells und den tatsächlichen Werten berechnen. Diese Kennzahl wird auch als *mittlere absolute Abweichung* (Mean Absolute Error, MAE) bezeichnet.

```
def mae(preds: ndarray, actuals: ndarray):
    '''
    Die mittlere absolute Abweichung berechnen.
    '''
    return np.mean(np.abs(preds - actuals))
```

- Die mittlere quadratische Verschiebung zwischen den Vorhersagen unseres Modells und den tatsächlichen Werten berechnen. Diese Kennzahl nennt man auch *Wurzel der mittleren Fehlerquadratsumme* (Root Mean Squared Error, RMSE).

```
def rmse(preds: ndarray, actuals: ndarray):
    '''
    Die Wurzel der mittleren Fehlerquadratsumme berechnen.
    '''
    return np.sqrt(np.mean(np.power(preds - actuals, 2)))
```

Die Werte für dieses Modell lauten:

```
Mean absolute error: 3.5643
Root mean squared error: 5.0508
```

Der RMSE ist ein besonders häufig verwendeter Kennwert, da er den gleichen Maßstab wie das Ziel hat. Dividieren wir diese Zahl durch den Mittelwert des Ziels, können wir messen, wie weit unsere Vorhersage durchschnittlich von ihrem tatsächlichen Wert entfernt ist. Der Wert von `y_test` ist `22,0776`. Das bedeutet, die Vorhersagen liegen durchschnittlich um `5,0508 : 22,0776 ≅ 22,9%` daneben.

Haben diese Zahlen also irgendeine Aussagekraft? Im Jupyter Notebook (*https://oreil.ly/2TDV5q9*) für dieses Kapitel zeige ich, dass eine lineare Regression an diesem Datensatz unter Verwendung von *Scikit-learn*, einer sehr beliebten Python-Bibliothek für das Machine Learning, einen mittleren absoluten Fehler von `3,5666` ergibt, während die Wurzel der mittleren Fehlerquadratsumme `5,0482` beträgt.

Das ist quasi identisch mit den Werten, die wir zuvor mit unserer grundbausteinbasierten linearen Regression berechnet haben. Das sollte Ihnen Zuversicht geben. Unser in diesem Buch verfolgter Denkansatz für das Training der Modelle scheint gültig zu sein! Etwas später werden wir diesen Ansatz auf neuronale Netze und Deep-Learning-Modelle erweitern.

Das wichtigste Merkmal analysieren

Vor der Modellierung haben wir alle Merkmale unserer Daten auf den Mittelwert 0 und die Standardabweichung 1 skaliert. Das hat gewisse Vorteile bei der Berechnung, die wir in Kapitel 4 detailliert behandeln werden. Ein Vorteil für die lineare

Regression liegt darin, dass wir die absoluten Werte der Koeffizienten so interpretieren können, dass sie der Wichtigkeit verschiedener Merkmale des Modells entsprechen. Ein größerer Koeffizient bedeutet, dass dieses Merkmal wichtiger ist. Hier die Koeffizienten:

```
np.round(weights['W'].reshape(-1), 4)
```

```
array([-1.0084,  0.7097,  0.2731,  0.7161, -2.2163,  2.3737,  0.7156,
       -2.6609,  2.629 , -1.8113, -2.3347,  0.8541, -4.2003])
```

Der letzte Koeffizient ist der größte. Das bedeutet, dass das letzte Merkmal im Datensatz das wichtigste ist.

In Abbildung 2-7 stellen wir dieses Merkmal unserem Ziel gegenüber:

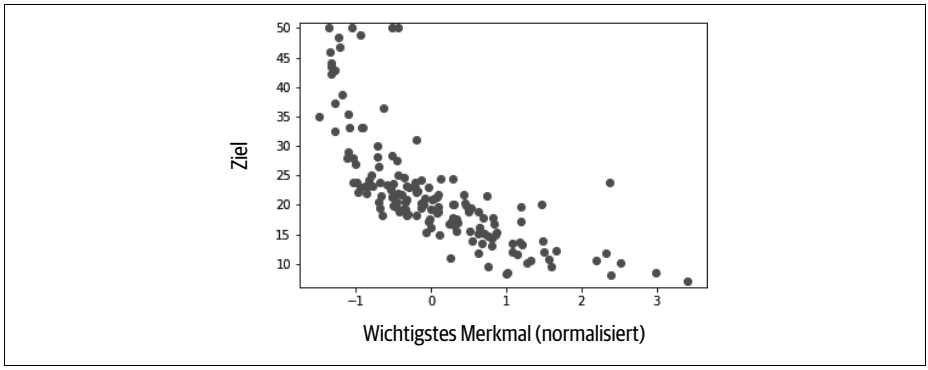

Abbildung 2-7: Das wichtigste Merkmal im Vergleich mit dem Ziel in unserer linearen Regression

Wir sehen, dass dieses Merkmal tatsächlich eine starke Korrelation zu unserem Ziel hat: Wird das Merkmal größer, verringert sich das Ziel und umgekehrt. Dabei ist die Beziehung jedoch *nicht* linear. Bei einer Änderung des Merkmals von –2 zu –1 erwarten wir den gleichen Zielwert wie bei einer Änderung von 1 zu 2. Das ist aber nicht der Fall. Später mehr dazu.

In Abbildung 2-8 überlagern wir unsere Darstellung mit der Beziehung zwischen diesem Merkmal und den *Vorhersagen des Modells*. Diese erzeugen wir, indem wir dem trainierten Modell die folgenden Daten übergeben:

- Die Werte aller Merkmale in Form ihrer Mittelwerte.
- Die Werte des wichtigsten Merkmals, linear über 40 Schritte interpoliert von –1,5 bis 3,5. Dies entspricht ungefähr dem Bereich dieses skalierten Merkmals in unseren Daten.

Diese Abbildung zeigt (wörtlich) eine Grenze der linearen Regression: Obwohl es eine deutlich sichtbare und »modellierbare« *nicht lineare* Beziehung zwischen diesem Merkmal und dem Ziel gibt, kann unser Modell aufgrund seiner Struktur nur eine *lineare* Beziehung »lernen«.

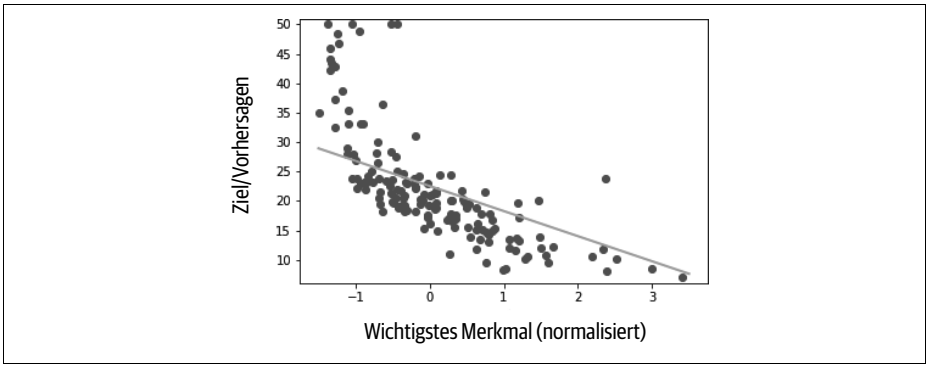

Abbildung 2-8: Das wichtigste Merkmal im Vergleich mit Ziel und Vorhersage in unserer linearen Regression

Um auch komplexere nicht lineare Beziehungen zwischen Merkmalen und Ziel zu lernen, reicht die lineare Regression offenbar nicht aus. Wie es aussieht, brauchen wir ein komplexeres Modell. Aber wie soll das gehen? Die Antwort bringt uns, basierend auf bestimmten Prinzipien, zur Erstellung eines neuronalen Netzes.

Neuronale Netze von Grund auf

Gerade haben wir gesehen, wie wir anhand unserer Grundbausteine ein lineares Regressionsmodell erstellen und trainieren können. Wie können wir diese Idee zu einem komplexeren Modell erweitern, dass auch nicht lineare Beziehungen lernen kann? Die Grundidee besteht darin, zunächst eine große Zahl an *linearen* Regressionen auszuführen, die Ergebnisse an eine *nicht lineare* Funktion zu übergeben und schließlich eine letzte *lineare* Regression durchzuführen, die am Ende die Vorhersagen trifft. Tatsächlich können wir die Berechnung der Gradienten dieses komplizierteren Modells auf die gleiche Weise herleiten wie für das lineare Regressionsmodell.

Schritt 1: Eine Reihe linearer Regressionen

Was meinen wir mit einer »großen Zahl linearer Regressionen«? Für eine lineare Regression musste eine Matrizenmultiplikation mit einer Reihe von Gewichtungen durchgeführt werden: Angenommen, unsere Daten X hätten die Dimensionen `[batch_size, num_features]` (Batchgröße, Anzahl der Merkmale). Diese werden mit einer Gewichtungsmatrix W mit den Dimensionen `[num_features, 1]` (Anzahl der Feature, 1) multipliziert, um eine Ausgabe der Dimensionen `[batch_size, 1]` zu erhalten. Dann ist diese Ausgabe für jede Beobachtung im Batch einfach eine *gewichtete Summe* der Ausgangsmerkmale. Um mehrere lineare Regressionen durchzuführen, multiplizieren wir unsere Eingabe einfach mit einer Gewichtungsmatrix der Dimensionen `[num_features, num_outputs]` (Anzahl der Merkmale, Anzahl der Ausgaben). Das Ergebnis ist eine Ausgabe der Dimensionen `[batch_`

`size, num_outputs]` (Batchgröße, Anzahl der Ausgaben). Damit haben wir für jede Beobachtung `num_outputs` gewichtete Summen der Ausgangsmerkmale.

Was sind diese gewichteten Summen? Wir können sie uns als »gelernte Merkmale« vorstellen. Hierfür versucht das Netz, Kombinationen der Ausgangsmerkmale zu finden, die bei der genauen Vorhersage der Hauspreise helfen. Da wir 13 Ausgangsmerkmale erstellt haben, verwenden wir auch 13 gelernte Merkmale.

Schritt 2: Eine nicht lineare Funktion

Jetzt übergeben wir die gewichteten Summen (unsere »gelernten Merkmale«) an eine nicht lineare Funktion. Für einen ersten Versuch verwenden wir die `sigmoid`-Funktion, die wir in Kapitel 1 bereits kurz kennengelernt haben. Abbildung 2-9 zeigt eine grafische Darstellung der `sigmoid`-Funktion.

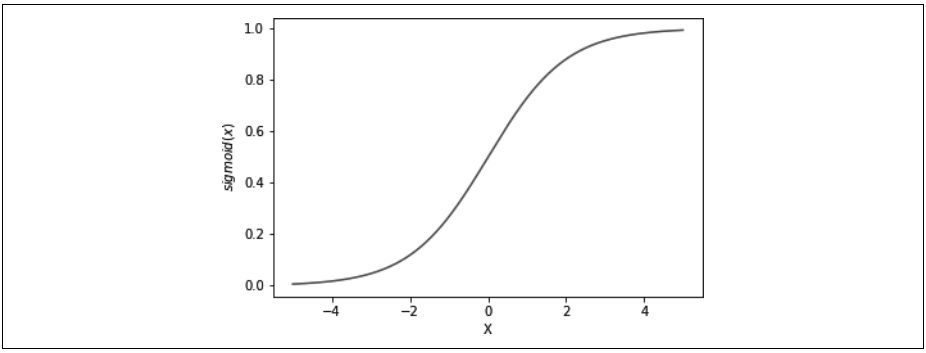

Abbildung 2-9: Grafische Darstellung der sigmoid-Funktion im Bereich zwischen $x = -5$ und $x = 5$

Warum ist die Verwendung einer nicht linearen Funktion hier sinnvoll? Warum benutzen wir nicht die `square`-Funktion $f(x) = x^2$? Dafür gibt es eine Reihe von Gründen. Einerseits soll die verwendete Funktion *monoton* sein, damit sie die Informationen zu den übergebenen Zahlen »behält«. Angenommen, zwei unserer linearen Regressionen hätten für die übergebenen Daten die Werte –3 bzw. 3 erzeugt. Übergeben wir diese an die `square`-Funktion, erhalten wir für beide den Wert 9. Nach der Bearbeitung mit `square` würde also jede Funktion, die diese Zahlen als Eingabe übernimmt, die Information über ihr Vorzeichen »verlieren«.

Der zweite Grund ist natürlich, dass die Funktion nicht linear ist. Durch diese Nichtlinearität ist unser neuronales Netz in der Lage, die grundsätzlich nicht lineare Beziehung zwischen Merkmalen und Ziel abzubilden.

Außerdem hat die `sigmoid`-Funktion noch die nette Eigenschaft, dass ihre Ableitungen in Form der Funktion selbst ausgedrückt werden können:

$$\frac{\partial \sigma}{\partial u}(x) = \sigma(x) \times (1 - \sigma(x))$$

Das werden wir uns gleich zunutze machen, wenn wir die `sigmoid`-Funktion in der Rückwärtspropagation unseres neuronalen Netzes verwenden.

Schritt 3: Noch eine lineare Regression

Zum Schluss nehmen wir die 13 Ergebniselemente – jedes eine Kombination der Ausgangsmerkmale, die an die `sigmoid`-Funktion übergeben wurden, damit ihre Werte zwischen 0 und 1 liegen – und übergeben sie an eine reguläre lineare Regression. Dabei benutzen wir sie auf die gleiche Weise wie zuvor unsere Ausgangsmerkmale.

Danach versuchen wir, die *gesamte* resultierende Funktion auf die gleiche Weise zu trainieren wie zuvor die lineare Regression. Wir übergeben unserem Modell die Daten und verwenden die Kettenregel, um herauszufinden, wie stark eine Erhöhung der Gewichtungen die Abweichung erhöhen (oder verringern) würde. Danach verändern wir bei jeder Iteration die Gewichtungen in der Richtung, die den Verlust verringert. Mit der Zeit sollten die Vorhersagen des Modells immer genauer werden, da es die inhärente Nichtlinearität der Beziehung zwischen unseren Merkmalen und dem Ziel »gelernt« hat. (So hoffen wir jedenfalls).

Es folgen ein paar Abbildungen, die das Beschriebene anschaulicher machen sollen.

Diagramme

Das Diagramm in Abbildung 2-10 zeigt, wie unser komplexeres Modell jetzt aussieht.

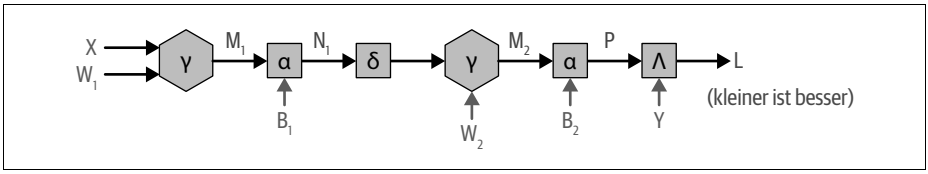

Abbildung 2-10: Die Schritte 1 bis 3, ausgedrückt als Rechengraph

Wie Sie sehen, beginnen wir auch hier mit einer Matrizenmultiplikation und -addition. Hierzu wollen wir zunächst ein paar Fachbegriffe klären: Wenn wir diese Operationen im Verlauf einer verschachtelten Funktion anwenden, bezeichnen wir die erste Matrix, die wir für die Umwandlung der Eingaben verwenden, als *Gewichtungsmatrix*. Die zweite Matrix, die jedem resultierenden Satz an Merkmalen hinzuaddiert wird, nennen wir *Bias*. Aufgrund der englischen Bezeichnungen Weight (Gewichtung) und Bias nutzen wir hier die Buchstaben W_1 und B_1.

Nach der Anwendung dieser Operationen übergeben wir die Ergebnisse an die `sigmoid`-Funktion. Dann wiederholen wir den Prozess mit einem weiteren Satz an Gewichtungen und Biases, die entsprechend W_2 und B_2 heißen, um schließlich unsere finale Vorhersage (*Prediction*) P zu erhalten.

Noch ein Diagramm?

Eine Frage beschäftigt uns in diesem Buch immer wieder: Hilft Ihnen diese Aufteilung in einzelne Schritte beim Verständnis der Vorgänge? Um neuronale Netze umfassend verstehen zu können, verwenden wir verschiedene Darstellungsweisen für einzelne Teilaspekte. Die Darstellung in Abbildung 2-10 gibt wenig Aufschluss über die Struktur des Netzes. Dafür zeigt sie deutlich, wie ein solches Modell trainiert wird: In der Rückwärtspropagation berechnen wir die Teilableitungen für jede Einzelfunktion, ausgewertet am Eingang zur jeweiligen Funktion. Danach berechnen wir die Gradienten für die Abweichung, bezogen auf die Gewichtungen. Hierfür multiplizieren wir diese Ableitungen einfach miteinander – wie wir es bereits von der einfachen Kettenregel aus Kapitel 1 kennen.

Für neuronale Netze gibt es aber auch noch eine andere Darstellungsweise, die deutlich weiter verbreitet ist. Hierfür wird jedes Ausgangsmerkmal als Kreis dargestellt. Für 13 Merkmale brauchen wir also 13 Kreise. 13 weitere Kreise stehen für die Ausgaben unserer »Lineare-Regression-Sigmoid«-Operation. Zusätzlich ist jeder Kreis eine Funktion (d.h. eine Kombination oder Verknüpfung) aller 13 Ausgangsmerkmale. Wir müssen also die 13 Kreise am Anfang mit allen Kreisen am Ende verbinden. Daher sprechen wir hier von *vollständig verbundenen neuronalen Netzen* (gelegentlich trifft man auch auf den Begriff »vermascht«, angelehnt an die Maschen eines Netzes).[8]

Zum Schluss werden alle 13 Ausgaben verwendet, um eine einzelne Vorhersage zu treffen. Also zeichnen wir ganz rechts einen weiteren Kreis, der für die abschließende Vorhersage steht, und 13 Linien, die verdeutlichen, dass die 13 »Zwischenausgaben« mit der abschließenden Vorhersage verbunden sind.

Abbildung 2-11 zeigt das fertige Diagramm.[9]

Wenn Sie bereits einmal etwas über neuronale Netze gelesen haben, kennen Sie Diagramme wie das in Abbildung 2-11 vermutlich schon: Kreise, die durch Linien verbunden sind. Diese Art der Darstellung hat zugegeben gewisse Vorteile. So können Sie auf den ersten Blick erkennen, um welche Art neuronales Netz es sich handelt, wie viele Schichten es gibt und so weiter. Aus dem Diagramm wird aber nicht ersichtlich, welche Berechnungen angestellt werden können oder ob ein solches Netz trainiert werden kann.

Wir zeigen Ihnen dieses Diagramm hauptsächlich, damit Sie verstehen, welche Verbindung zwischen der oben gezeigten Darstellungsweise und der in diesem Buch hauptsächlich verwendeten Form besteht. Zur Erinnerung: Wir nutzen durch Linien verbundene Kästen. Jeder Kasten steht für eine Funktion, die definiert, was

8 Das bringt uns auf eine interessante Idee: Wir könnten die Ausgaben also auch nur mit einigen unserer ursprünglichen Merkmale verbinden. Und genau das passiert in faltungsbasierten neuronalen Netzen, auf die wir in Kapitel 5 detailliert zu sprechen kommen.

9 Na ja, nicht ganz: Wir haben nicht *alle* 169 Linien gezeichnet, die nötig wären, um die Verbindungen zwischen den ersten beiden Schichten der Merkmale herzustellen. Es sind aber genug, um Ihnen zu zeigen, worum es hier geht.

während der Vorwärtspropagation des Modells zu tun ist. Damit versuchen wir, vorherzusagen, was in der Rückwärtspropagation passieren muss, damit das Modell lernen kann.

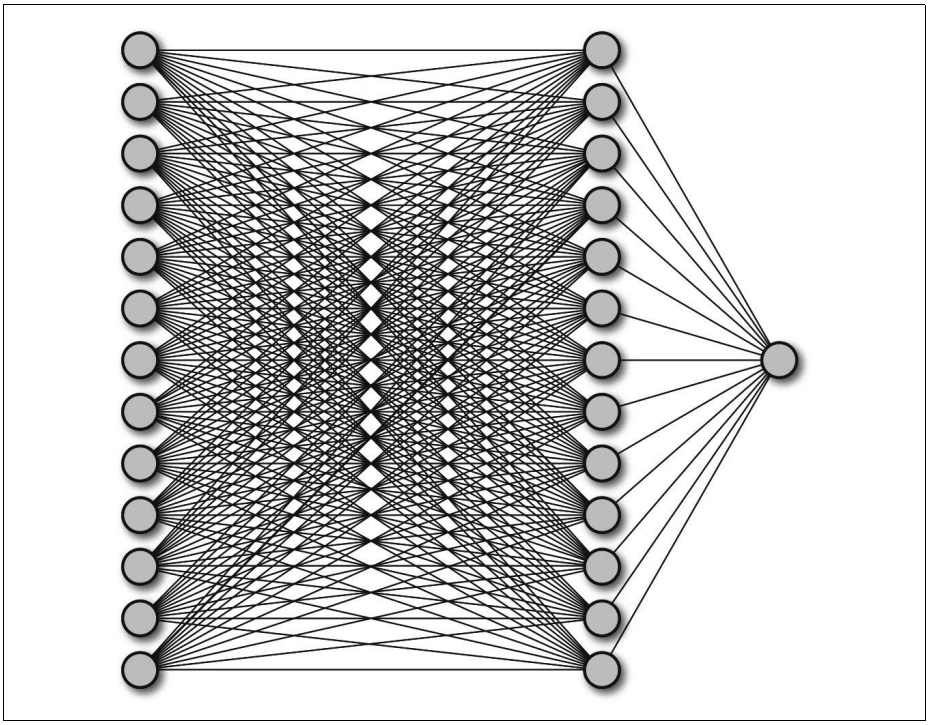

Abbildung 2-11: Eine häufiger verwendete (aber nicht unbedingt hilfreichere) visuelle Darstellung eines neuronalen Netzes

Im folgenden Kapitel werden wir sehen, wie wir diese Diagramme direkter ineinander »übersetzen« können. Hierfür programmieren wir jede Funktion als Python-Klasse, die von einer Basisklasse namens `Operation` erbt. Und rein zufällig geht es auch im folgenden Abschnitt um die Programmierung.

Code

Die Programmierung folgt der gleichen Funktionsstruktur wie bei der einfachen linearen Regressionsfunktion. Wir übernehmen `weights` als Dictionary und geben die Abweichung (`loss`) sowie das Dictionary `forward_info` zurück. Dabei ersetzen wir die Interna der linearen Regression durch die in Abbildung 2-10 gezeigten Operationen:

```
def forward_loss(X: ndarray,
                 y: ndarray,
                 weights: Dict[str, ndarray]
                 ) -> Tuple[Dict[str, ndarray], float]:
```

```
'''
Die Vorwärtspropagation und die Abweichung für das
schrittbasierte neuronale Netzmodell berechnen.
'''
M1 = np.dot(X, weights['W1'])

N1 = M1 + weights['B1']

O1 = sigmoid(N1)

M2 = np.dot(O1, weights['W2'])

P = M2 + weights['B2']

loss = np.mean(np.power(y - P, 2))

forward_info: Dict[str, ndarray] = {}
forward_info['X'] = X
forward_info['M1'] = M1
forward_info['N1'] = N1
forward_info['O1'] = O1
forward_info['M2'] = M2
forward_info['P'] = P
forward_info['y'] = y

return forward_info, loss
```

Trotz der höheren Komplexität bewegen wir uns auch hier Schritt für Schritt durch die einzelnen Operationen, führen die nötigen Berechnungen durch und speichern die Ergebnisse in `forward_info`.

Neuronale Netze: die Rückwärtspropagation

Die Rückwärtspropagation funktioniert wie bei unserem einfachen linearen Regressionsmodell. Es gibt nur ein paar zusätzliche Schritte.

Diagramm

Zur Erinnerung hier noch einmal die nötigen Schritte:

1. Die Ableitung jeder Operation berechnen und sie an der jeweiligen Eingabe auswerten.
2. Die Ergebnisse multiplizieren.

Auch hier sehen wir, dass dies aufgrund der Kettenregel funktioniert. Abbildung 2-12 zeigt alle Teilableitungen, die wir berechnen müssen.

Vom Konzept her wollen wir uns rückwärts durch unsere Funktion bewegen und dabei sämtliche Teilableitungen berechnen. Diese werden wie im linearen Regressionsmodell multipliziert, um die Gradienten der Abweichung bezogen auf die jeweiligen Gewichtungen zu erhalten.

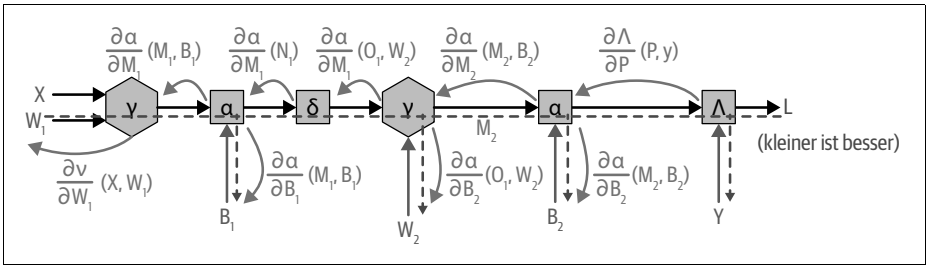

Abbildung 2-12: Die zu jeder Operation im neuronalen Netz gehörigen Teilableitungen, die in der Rückwärtspropagation multipliziert werden

Mathematik (und Code)

Tabelle 2-1 enthält eine Liste der Teilableitungen mit den entsprechenden Codezeilen.

Tabelle 2-1: Ableitungstabelle für das neuronale Netz

Ableitung	Code
$\frac{\partial \Lambda}{\partial P}(P, y)$	`dLdP = -2 * (forward_info[y] - forward_info[p])`
$\frac{\partial \alpha}{\partial M_2}(M_2, B_2)$	`np.ones_like(forward_info[M2])`
$\frac{\partial \alpha}{\partial B_2}(M_2, B_2)$	`np.ones_like(weights[B2])`
$\frac{\partial \nu}{\partial W_2}(O_1, W_2)$	`dM2dW2 = np.transpose(forward_info[O1], (1, 0))`
$\frac{\partial \nu}{\partial O_1}(O_1, W_2)$	`dM2dO1 = np.transpose(weights[W2], (1, 0))`
$\frac{\partial \sigma}{\partial u}(N_1)$	`dO1dN1 = sigmoid(forward_info[N1] × (1 - sigmoid(forward_info[N1])`
$\frac{\partial \alpha}{\partial M_1}(M_1, B_1)$	`dN1dM1 = np.ones_like(forward_info[M1])`
$\frac{\partial \alpha}{\partial B_1}(M_1, B_1)$	`dN1dB1 = np.ones_like(weights[B1])`
$\frac{\partial \nu}{\partial W_1}(X, W_1)$	`dM1dW1 = np.transpose(forward_info[X], (1, 0))`

Weil jeder Zeile im durchlaufenen Batch dasselbe Bias-Element hinzugefügt wird, müssen die für den Abweichungsgradienten (bezogen auf die Bias-Terme `dLdB1` und `dLdB2`) berechneten Ausdrücke entlang jeder Zeile summiert werden. Die Details finden Sie im Abschnitt »Gradient der Abweichung, bezogen auf die Bias-Terme« auf Seite 225.

Der Gradient für die Gesamtabweichung

Die vollständige `loss_gradients`-Funktion (Abweichungsgradientenfunktion) finden Sie im Jupyter Notebook (*https://oreil.ly/2TDV5q9*) für dieses Kapitel im GitHub-Repository zu diesem Buch. Die Funktion berechnet sämtliche in Tabelle 2-1 gezeigten Teilableitungen und multipliziert sie, um die Abweichungsgradienten bezogen auf alle `ndararrays` mit Gewichtungen zu ermitteln:

- `dLdW2`
- `dLdB2`
- `dLdW1`
- `dLdB1`

Hierbei müssen wir nur darauf achten, dass wir die für `dLdB1` und `dLdB2` berechneten Ausdrücke entlang von `axis = 0` summieren, wie im Abschnitt »Gradient der Abweichung, bezogen auf die Bias-Terme« auf Seite 225 beschrieben.

Und damit haben wir unser erstes neuronales Netz von Grund auf erstellt! Jetzt wollen wir überprüfen, ob es tatsächlich besser ist als unser lineares Regressionsmodell.

Unser erstes neuronales Netz trainieren und bewerten

Wie wir gesehen haben, funktionieren die Vorwärts- und die Rückwärtspropagation für unser neuronales Netz auf die gleiche Weise wie für das lineare Regressionsmodell. Das Gleiche gilt für das Training und die Auswertung: Für jede Iteration der Daten reichen wir die Eingaben während der Vorwärtspropagation in der Funktion weiter, berechnen in der Rückwärtspropagation die Abweichungsgradienten bezogen auf die Gewichtungen und verwenden diese Gradienten schließlich, um die Gewichtungen zu aktualisieren. Tatsächlich können wir folgenden identischen Code in der Trainingsschleife benutzen:

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)

for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

Der Unterschied liegt einfach nur im Inhalt der Funktionen `forward_loss` und `loss_gradients` sowie im `weights`-Dictionary. Letzteres hat nun nicht mehr zwei Schlüssel, sondern vier (`W1`, `B1`, `W2` und `B2`). Hier sehen wir eine wichtige Erkenntnis aus diesem Buch: Selbst für komplexe Architekturen sind die mathematischen Prinzipien und die allgemeinen Trainingsverfahren die gleichen wie für einfache Modelle.

Und auch die Vorhersagen (`preds`) für dieses Modell erhalten wir auf die gleiche Weise:

```
preds = predict(X_test, weights)
```

Der Unterschied liegt hier ebenfalls im Inhalt der `predict`-Funktion:

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]) -> ndarray:
    '''
    Erzeuge Vorhersagen aus dem schrittweise arbeitenden
    neuronalen Netzmodell.
    '''
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']

    O1 = sigmoid(N1)

    M2 = np.dot(O1, weights['W2'])

    P = M2 + weights['B2']

    return P
```

Anhand dieser Vorhersagen können wir auch diesmal an den Testdaten die mittlere absolute Abweichung und die mittlere quadratische Abweichung berechnen:

```
Mean absolute error: 2.5289
Root mean squared error: 3.6775
```

Beide Werte liegen deutlich unter denen des vorherigen Modells! Die Betrachtung des Graphen der Vorhersagen im Vergleich mit den tatsächlichen Werten (siehe Abbildung 2-13) zeigt ähnliche Verbesserungen.

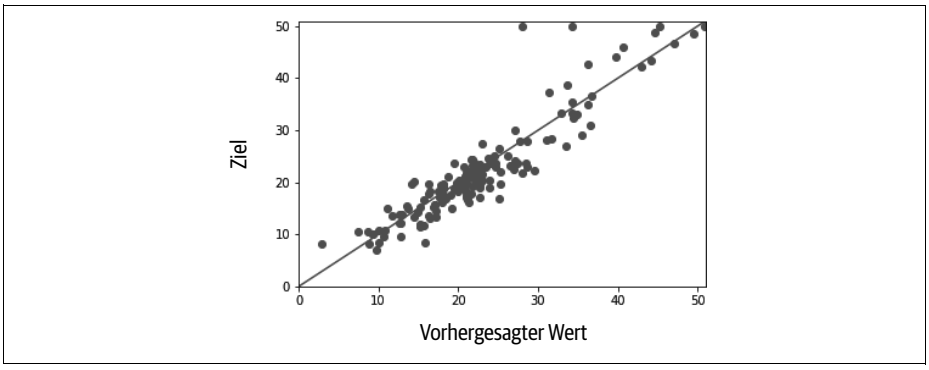

Abbildung 2-13: Vorhergesagte Werte im Vergleich mit Zielwerten bei der Regression in einem neuronalen Netz

Visuell liegen die Punkte deutlich näher an der 45-Grad-Linie als in Abbildung 2-6. Ich empfehle Ihnen, sich hierzu das Jupyter Notebook (*https://oreil.ly/2TDV5q9*) auf der GitHub-Seite zu diesem Buch anzusehen und den Code selbst auszuführen!

Zwei Gründe, warum das passiert

Warum scheint dieses Modell besser zu funktionieren als das vorherige Modell? Vergessen Sie nicht, dass zwischen dem wichtigsten Merkmal unseres früheren Modells und unserem Ziel eine *nicht lineare* Beziehung bestand. Trotzdem war unser vorheriges Modell darauf beschränkt, *lineare* Beziehungen zwischen einzelnen Merkmalen und dem Ziel zu lernen. Ich behaupte, dass wir durch die Verwendung einer nicht linearen Funktion unserem Modell ermöglicht haben, die korrekte, d.h. nicht lineare, Beziehung zwischen unseren Merkmalen und unserem Ziel zu lernen.

Sehen wir uns das noch einmal in grafischer Form an. Abbildung 2-14 zeigt den gleichen Graphen, den wir bereits im Abschnitt zur linearen Regression gesehen haben. Er zeigt die normalisierten Werte des wichtigsten Merkmals unseres Modell zusammen mit den Werten unseres Ziels und der Vorhersage, die sich ergeben würde, wenn wir die Durchschnittswerte der anderen Merkmale eingeben. Dabei variieren wir die Werte des wichtigsten Merkmals wie zuvor von –3,5 bis 1,5.

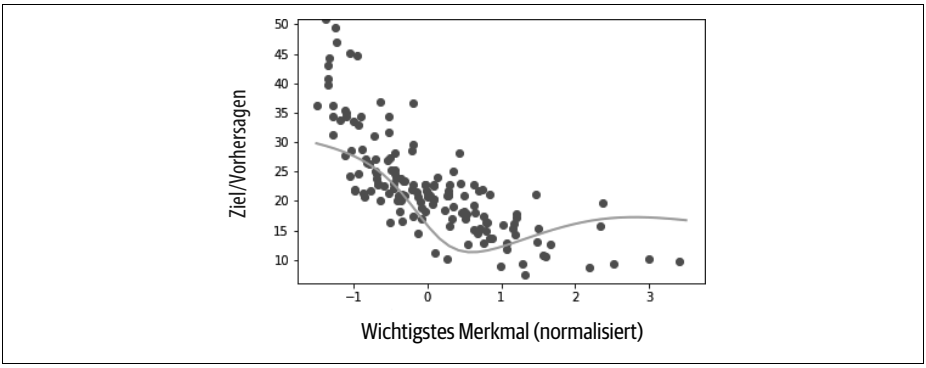

Abbildung 2-14: Das wichtigste Merkmal verglichen mit dem Ziel und den Vorhersagen bei der Regression im neuronalen Netz

Wir können sehen, dass die untersuchte Beziehung wie gewünscht zum einen nicht linear ist und zum anderen besser auf die Beziehung zwischen diesem Merkmal und dem Ziel (dargestellt durch die Punkte) passt. Durch die Erweiterung unseres Modells um die nicht lineare Funktion war es also in der Lage, durch wiederholtes Anpassen der Gewichtungen beim Training die nicht lineare Beziehung zwischen den Ein- und Ausgaben zu lernen.

Das ist der erste Grund dafür, dass unser neuronales Netz besser funktioniert hat als eine lineare Regression: Anstatt nur einzelne Merkmale zu lernen, war unser neuronales Netz in der Lage, Beziehungen zwischen *Kombinationen* unserer Ausgangsmerkmale und dem Ziel zu finden. Das funktioniert, weil das neuronale Netz eine Matrizenmultiplikation verwendet, um 13 »gelernte Merkmale« zu erzeugen, die jeweils eine Kombination aller Ausgangsmerkmale sind. Auf der GitHub-Seite zu diesem Buch (*https://oreil.ly/deep-learning-github*) finden Sie eine explorative

Datenanalyse, die zeigt, welche der 13 Ausgangsmerkmale die wichtigsten sind, und zwar:

$$-4.44 \times \text{feature}_6 - 2.77 \times \text{feature}_1 - 2.07 \times \text{feature}_7 + \ldots$$

und:

$$4.43 \times \text{feature}_2 - 3.39 \times \text{feature}_4 - 2.39 \times \text{feature}_1 + \ldots$$

Diese werden zusammen mit elf weiteren gelernten Merkmalen in einer linearen Regression der letzten beiden Schichten des neuronalen Netzes benutzt.

Aus diesen Gründen sind neuronale Netze für die Lösung realer Probleme oft besser geeignet als eine einfache Regression: Neuronale Netze lernen *nicht lineare* Beziehungen zwischen einzelnen Merkmalen und unserem Ziel sowie *Kombinationen* aus Merkmalen und Ziel.

Schlussbemerkung

In diesem Kapitel haben Sie gelernt, wie Sie die Grundbausteine und Gedankenmodelle aus Kapitel 1 anwenden können, um zwei Standardmodelle des Machine Learning zu verstehen, zu erstellen und zu trainieren, um damit Probleme aus der realen Welt zu lösen. Zu Beginn habe ich gezeigt, wie man ein einfaches Modell des Machine Learning aus der klassischen Statistik – die lineare Regression – unter Verwendung eines Rechengraphen darstellt. Diese Art der Darstellung ermöglichte es uns, die Gradienten der Abweichung aus diesem Modell bezogen auf die Parameter des Modells zu berechnen. So konnten wir das Modell trainieren, indem wir es immer wieder mit den Trainingsdaten gefüttert und dabei die Parameter (Gewichtungen) des Modells so angepasst haben, dass die Abweichung immer geringer wurde.

Dann haben wir die Beschränkungen dieses Modells erkannt: Es kann nur *lineare* Beziehungen zwischen den Merkmalen und dem Ziel lernen. Das brachte uns darauf, ein Modell zu entwickeln, das *nicht lineare* Beziehungen zwischen den Merkmalen und dem Ziel erkennen kann. Dadurch kamen wir wiederum zur Entwicklung unseres ersten neuronalen Netzes. Sie haben gelernt, wie neuronale Netze funktionieren, indem Sie von Grund auf selbst eines erstellt haben. Sie haben ebenfalls gesehen, wie ein solches Netz anhand des gleichen allgemeinen Verfahrens trainiert werden kann, das auch bei linearen Regressionsmodellen zum Einsatz kommt. Daraufhin haben Sie empirisch festgestellt, dass das neuronale Netz eine bessere Leistung zeigt als das einfache lineare Regressionsmodell, und hierfür zwei Gründe erfahren: Das neuronale Netz konnte *nicht lineare* Beziehungen zwischen den Merkmalen und dem Ziel lernen und war zudem in der Lage, Beziehungen zwischen *Kombinationen* der Merkmale und dem Ziel zu lernen.

Selbstverständlich gibt es einen Grund dafür, dass wir das Kapitel mit einem verhältnismäßig einfachen Modell beendet haben: Bei der Definition neuronaler Netze auf diese Weise ist enorm viel Handarbeit nötig. Die Definition der Vorwärtspropagation benötigte sechs einzeln programmierte Schritte, bei der Rückwärtspropagation waren es sogar 17. Aufmerksamen Lesern wird aufgefallen sein, dass diese Schritte eine Menge Wiederholungen enthalten. Durch sinnvolle Abstraktionen sollten wir in der Lage sein, Modelle nicht mehr in Form einzelner Operationen (wie in diesem Kapitel), sondern auf Basis ebendieser Abstraktionen zu definieren. Dadurch werden wir in der Lage sein, komplexere Modelle – dazu gehören auch Deep-Learning-Modelle – zu erstellen. Gleichzeitig werden wir Ihr Verständnis von der Funktionsweise dieser Modelle vertiefen. Damit beginnen wir im folgenden Kapitel.

Und weiter!