KAPITEL 4

Konvolutionsnetze

In diesem Kapitel stellen wir Konvolutionsnetze (engl. Convolutional Neural Networks, CNNs) und deren Bestandteile und Arbeitsweise vor. Wir beginnen mit einem einfachen Klassifikationsmodell für den MNIST-Datensatz, stellen dann den CIFAR10-Datensatz zur Objekterkennung vor und wenden unterschiedliche CNN-Modelle darauf an. Die in diesem Kapitel gezeigten CNNs sind zwar klein und schnell, aber dennoch repräsentativ für die in der Praxis zur Objekterkennung eingesetzten modernen Modelle.

Einführung in Konvolutionsnetze

Konvolutionsnetze haben in den letzten Jahren eine Sonderstellung als besonders vielversprechende Art des Deep Learnings erlangt. Aus der Bildverarbeitung kommend, sind Konvolutionsschichten in praktisch sämtliche Teilgebiete des Deep Learnings vorgedrungen, meist mit großem Erfolg.

Der grundlegende Unterschied zwischen *vollständig verbundenen Netzen* und *Konvolutionsnetzen* ist das Verbindungsmuster aufeinanderfolgender Schichten. In einer vollständig verbundenen Schicht ist, wie der Name vermuten lässt, jede Einheit mit allen Einheiten der vorigen Schicht verbunden. Ein Beispiel dafür haben wir in Kapitel 2 gesehen, wo die 10 Ausgabezellen mit sämtlichen Eingabepixeln verbunden waren.

In der Konvolutionsschicht eines CNN dagegen ist jede Einheit mit (normalerweise wenigen) räumlich nahen Einheiten der vorigen Schicht verknüpft. Außerdem sind sämtliche Einheiten mit der vorigen Schicht auf dieselbe Weise, mit genau denselben Gewichten und derselben Struktur verbunden. Dadurch wird ein als *Konvolution* bezeichneter Vorgang ermöglicht, der für diese Architektur namensgebend ist. (Dieses Konzept wird in Abbildung 4-1 veranschaulicht.) Im nächsten Abschnitt werden wir uns mit der Konvolution etwas ausführlicher beschäftigen. Es wird für uns aber im Wesentlichen darauf hinauslaufen, dass wir kleine »Fenster« von Gewichten (sogenannte *Filter*) wie in Abbildung 4-2 über ein Bild hinweg anwenden.

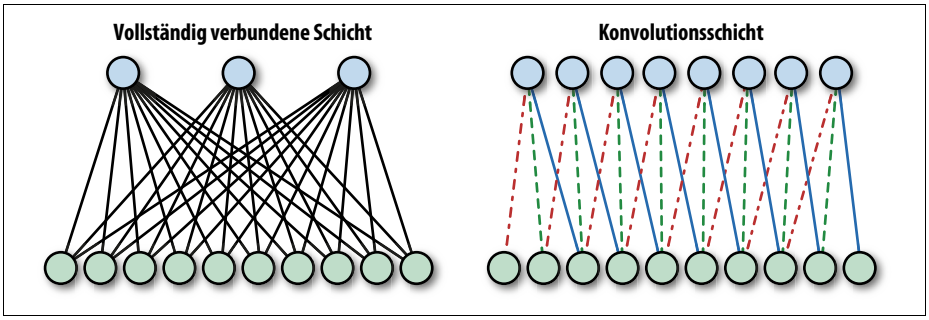

Abbildung 4-1: In einer vollständig verbundenen Schicht (links) ist jede Einheit mit allen Einheiten der vorigen Schicht verbunden. In einer Konvolutionsschicht (rechts) ist jede Einheit mit einer festgelegten Anzahl Einheiten in einem benachbarten Teilbereich der vorigen Schicht verbunden. Außerdem verwenden alle Einheiten einer Konvolutionsschicht die gleichen Gewichte für diese Verbindungen, wie durch die Linienarten angedeutet wird.

Es werden einige unterschiedliche Inspirationsquellen für die in CNNs umgesetzte Grundidee genannt. Die erste ist die Neurowissenschaft, die zweite sind Erkenntnisse über Eigenschaften von Bildern, und die dritte bezieht sich auf Lerntheorien. Wir werden diese in Kürze besprechen, bevor wir uns mit der Arbeitsweise von CNNs im Einzelnen beschäftigen.

In der Vergangenheit wurden neuronale Netze, insbesondere Konvolutionsnetze, häufig als von der Biologie inspirierte Rechenmodelle beschrieben. Bisweilen wurde sogar postuliert, dass sie *die Arbeitsweise des Gehirns nachahmen*. Auch wenn diese Analogie irreführend ist, wenn man sie zu wörtlich nimmt, ist sie dennoch ein interessanter Blickwinkel.

Die Nobelpreisträger und Neurophysiologen Hubel und Wiesel hatten bereits in den 1960er-Jahren entdeckt, dass die ersten Stufen des Sehzentrums im Gehirn den gleichen lokalen Filter (beispielsweise das Erkennen von Kanten) auf alle Teile des Sichtfelds anwenden. Nach dem gegenwärtigen Stand der Neurowissenschaften werden während der visuellen Verarbeitung Informationen hierarchisch aus immer größeren Teilbereichen des Gesichtsfelds zusammengefügt.

Konvolutionsnetze folgen dem gleichen Muster. Jede Konvolutionsschicht betrachtet einen immer größeren Teil des Bilds, je weiter wir im Netz voranschreiten. Normalerweise folgen darauf vollständig verbundene Schichten, die in der biologischen Analogie für die höheren Ebenen der Bildverarbeitung stehen, die Gesichtsfeld-übergreifende Informationen auswerten.

Ein zweiter, ingenieursmäßigerer Blickwinkel, der sich mehr auf Tatsachen stützt, bezieht sich auf die Struktur von Bildern und ihrer Inhalte. Wenn wir in einem Bild ein bestimmtes Objekt suchen, sagen wir, den Kopf einer Katze, möchten wir es normalerweise unabhängig von seiner Lage im Bild erkennen. In echten Bildern kann also der gleiche Inhalt an unterschiedlichen Stellen vorhanden sein. Diese

Eigenschaft bezeichnet man als *Invarianz* – solche Invarianzen können auch in Bezug auf (kleine) Rotationen, unterschiedliche Lichtverhältnisse usw. vorliegen.

Dementsprechend sollte sich ein System zur Objekterkennung ebenfalls invariant gegenüber Verschiebungen verhalten (und je nach Anwendungsfall vermutlich auch in Bezug auf Rotationen und vielerlei andere Verformungen, aber das steht auf einem anderen Blatt). Einfach gesagt, ist es also sinnvoll, ein und dieselbe Berechnungsvorschrift auf unterschiedliche Teile eines Bilds anzuwenden. Zu diesem Zweck berechnet eine Konvolutionsschicht die gleichen Merkmale für sämtliche Bereiche eines Bilds.

Schließlich lässt sich die Struktur eines Konvolutionsnetzes auch als Regularisierungsmechanismus auffassen. Aus diesem Blickwinkel verhalten sich die Konvolutionsschichten wie vollständig verbundene Schichten, aber anstatt die Gewichte im Raum aller möglichen Matrizen (einer bestimmten Größe) zu suchen, beschränken wir die Suche auf Matrizen, die Konvolutionen fester Größe beschreiben. So reduzieren wir die Anzahl der Freiheitsgrade auf die Größe der Konvolution, die für gewöhnlich sehr klein ist.

Regularisierung

Der Begriff der *Regularisierung* wird im gesamten Buch verwendet. Beim maschinellen Lernen und in der Statistik ist mit Regularisierung meistens die Einschränkung einer Optimierungsaufgabe gemeint, indem die Komplexität der Lösung mit einem Strafterm belegt wird, um Overfitting an die gegebenen Lerndaten zu vermeiden.

Overfitting tritt auf, wenn eine Regel (beispielsweise ein Klassifikator) so berechnet wird, dass er die Anlerndaten gut erklärt, aber bei der Verallgemeinerung auf noch nicht gesehene Daten versagt.

Regularisierung wird meist umgesetzt, indem implizite Informationen über das gewünschte Ergebnis angegeben werden. (Wir könnten so beispielsweise ausdrücken, dass wir beim Durchsuchen eines Funktionenraums eine glattere Funktion bevorzugen.) Im Fall eines Konvolutionsnetzes legen wir explizit fest, dass wir nach Gewichten in einem relativ niedrigdimensionalen Teilraum suchen, der Konvolutionen mit einer festgelegten Größe entspricht.

In diesem Kapitel werden wir die Arten von Schichten und Operationen behandeln, die bei Konvolutionsnetzen auftreten. Zu Beginn schauen wir uns noch einmal den MNIST-Datensatz an und erzeugen diesmal ein Modell mit einer Genauigkeit von etwa 99 %. Anschließend werden wir uns dem interessanteren Datensatz CIFAR10 und damit der Objekterkennung zuwenden.

MNIST: Zweite Runde

In diesem Abschnitt betrachten wir den MNIST-Datensatz ein zweites Mal. Diesmal verwenden wir als Klassifikator ein kleines Konvolutionsnetz. Zuvor müssen wir uns mit einigen Elementen und Operationen vertraut machen.

Konvolution

Wie Sie sich anhand des Namens sicher schon denken, ist die Konvolution die grundlegende Operation, mit der die Schichten in Konvolutionsnetzen miteinander verbunden sind. Wir verwenden die in TensorFlow eingebaute Funktion `conv2d()`:

```
tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

Dabei enthält `x` die Daten – das Eingabebild oder eine abgeleitete Merkmalskarte, die später im Netz nach Anwendung einer vorgeschalteten Konvolutionsschicht generiert wird. Wie oben erwähnt, stapeln wir die Konvolutionsschichten in einem typischen CNN hierarchisch aufeinander, und *Merkmalskarte* ist nichts weiter als ein gebräuchlicher Begriff für die Ausgabe jeder solchen Schicht. Die Ausgaben dieser Schichten können auch als *verarbeitete Bilder*, das Ergebnis der Anwendung eines Filters und eventuell weiterer Operationen, betrachtet werden. Hierbei steht der Parameter `W` des Filters für die angelernten Gewichte, die den Konvolutionsfilter repräsentieren. Es sind also gerade die Gewichte innerhalb des kleinen »gleitenden Fensters« in Abbildung 4-2.

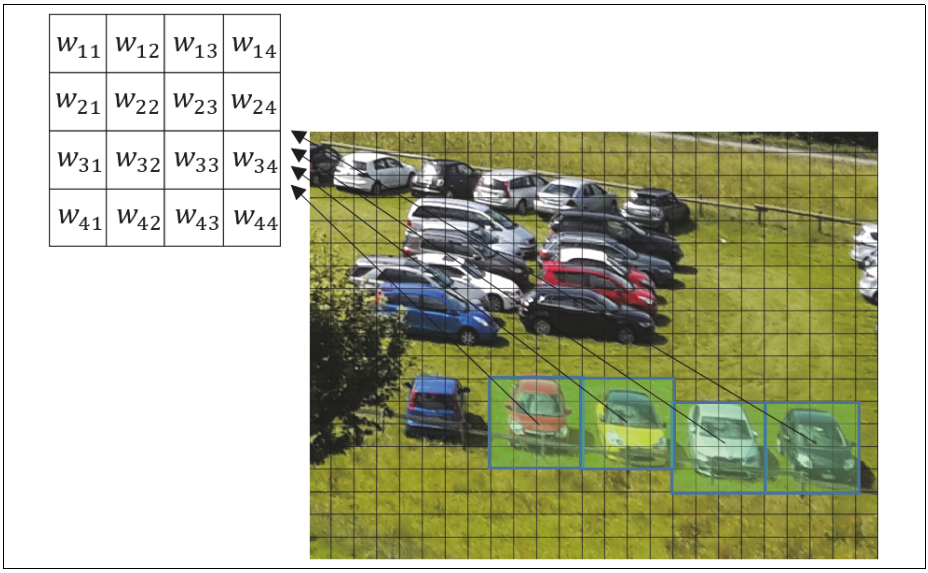

Abbildung 4-2: Der gleiche Konvolutionsfilter wird – als »gleitendes Fenster« – auf das gesamte Bild angewendet.

Die Ausgabe dieser Operation hängt von der Gestalt von `x` und `W` ab und ist in unserem Fall vierdimensional. Die Bilddaten `x` haben die Gestalt:

```
[None, 28, 28, 1]
```

Das bedeutet, dass wir eine unbekannte Anzahl Bilder haben, von denen jedes 28 × 28 Pixel groß ist und genau einen Farbkanal besitzt (weil es Graustufenbilder sind). Die Gewichte `W` haben die Gestalt:

```
[5, 5, 1, 32]
```

Dabei steht 5 × 5 × 1 für die Größe des kleinen »Konvolutions-Fensters« im Bild, in unserem Fall ein 5 × 5 Pixel großer Ausschnitt. In Bildern mit mehreren Farbkanälen (also RGB, wie in Kapitel 1 besprochen) behandeln wir jedes Bild als dreidimensionalen Tensor mit RGB-Werten, aber bei nur einem Farbkanal sind die Daten lediglich zweidimensional, und die Konvolutionsfilter werden auf zweidimensionale Regionen angewandt. Sobald wir uns dem CIFAR10-Datensatz zuwenden, werden wir auch Beispiele für Bilder mit mehreren Farbkanälen sehen und erfahren, wie sich die Größe der Gewichte `W` entsprechend anpassen lässt.

Die 32 am Ende ist die Anzahl der Merkmalskarten. Anders ausgedrückt, haben wir für die Konvolutionsschicht mehrere Versionen der Gewichte, in unserem Fall 32. Der Grundgedanke einer Konvolutionsschicht ist, ein und dasselbe Merkmal über das gesamte Bild hinweg zu berechnen; wir möchten möglichst viele Merkmale ausrechnen und verwenden daher mehrere Konvolutionsfilter.

Der Parameter `strides` steuert die Bewegung des Filters `W` über das Pixelgitter des Bilds (oder der Merkmalskarte) `x`.

Der Wert `[1, 1, 1, 1]` bedeutet, dass dieser Filter in jeder Dimension in Ein-Pixel-Intervallen über die Eingabedaten gelegt wird, was einer »vollständigen« Konvolution entspricht. Mit anderen Werten für diesen Parameter überspringt der Filter bestimmte Positionen, wodurch die entstehende Merkmalskarte kleiner wird – eine übliche Vorgehensweise, die wir uns später zunutze machen werden.

Schließlich setzen wir `padding` auf `'SAME'`, wodurch die Randbereiche von `x` rechnerisch so erweitert werden, dass das Ergebnis die gleiche Gestalt wie `x` aufweist.

Aktivierungsfunktionen

Es ist gängige Praxis, im Anschluss an lineare Schichten (Konvolutionsschichten oder vollständig verbundenen) eine nichtlineare *Aktivierungsfunktion* anzuwenden. (Beispiele finden Sie in Abbildung 4-3.) Ein praktischer Aspekt der Aktivierungsfunktionen ist, dass sich aufeinanderfolgende lineare Operationen durch eine einzelne ersetzen lassen. Deshalb erhöht die Tiefe des Netzes dessen Ausdruckskraft nicht, solange wir keine nichtlineare Aktivierungsfunktion zwischen den linearen Schichten anwenden.

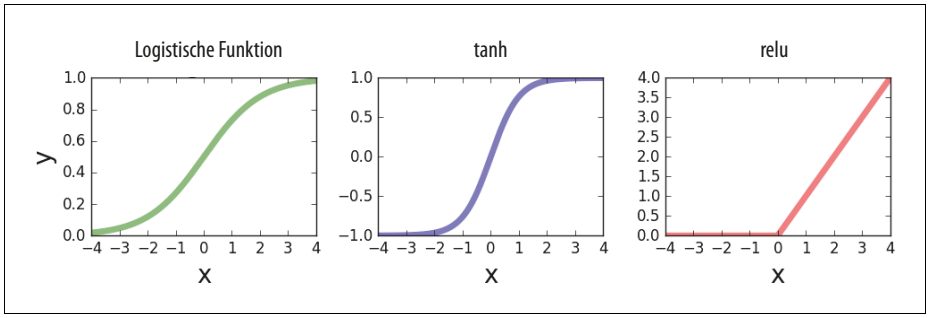

Abbildung 4-3: Häufig verwendete Aktivierungsfunktionen: logistische Funktion (links), Tangens hyperbolicus (Mitte) und linearer Gleichrichter (rechts)

Pooling

Üblicherweise wendet man auf die Ausgabedaten von Konvolutionsschichten ein Pooling an. Technisch wird beim *Pooling* die Datenmenge durch eine lokale Aggregationsfunktion verringert, meist innerhalb einer Merkmalskarte.

Die Gründe hierfür sind sowohl technischer als auch konzeptioneller Natur. Der technische Aspekt ist, dass Pooling die Menge der weitergereichten Daten reduziert. Dadurch lässt sich die Gesamtzahl der Modellparameter drastisch verringern, besonders wenn nach den Konvolutionsschichten vollständig verbundene Schichten folgen.

Ein eher theoretisches Argument für den Einsatz von Pooling ist, dass die berechneten Merkmale unempfindlich gegenüber kleinen Verschiebungen des Bilds sein sollten. Beispielsweise sollte ein Merkmal, das rechts oben in einem Bild nach Augen sucht, sich nicht allzu sehr ändern, wenn wir die Kamera bei der Aufnahme ein kleines Stück nach rechts bewegen, wodurch die Augen ein wenig in die Mitte rutschen. Die räumliche Aggregation des »Augenerkennungs-Merkmals« erlaubt es dem Modell, über solche räumlichen Variationen im Bild hinwegzusehen und so die zu Beginn des Kapitels erwähnten Invarianzen zu erfassen.

In unserem Beispiel wenden wir die Max-Pooling-Operation auf Bereiche der Größe 2×2 jeder Merkmalskarte an:

```
tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

Beim Max-Pooling wird das Maximum der Eingabe in Bereichen vorgegebener Größe ermittelt (hier 2 × 2). Der Parameter `ksize` steuert die Größe des Poolings (2 × 2), der Parameter `strides` stellt wie bei der Konvolutionsschicht die Schrittweite beim »Verschieben« des Pooling-Bereichs über `x` ein. Ein Gitter der Größe 2 × 2 bedeutet, dass die Ausgabe des Pooling genau die halbe Höhe und Breite des ursprünglichen Bilds hat und damit insgesamt ein Viertel seiner Größe.

Dropout

Als letzten Baustein für unser Modell benötigen wir noch *Dropout*. Dropout ist eine Regularisierungstechnik, mit der wir das Netz dazu zwingen, die angelernte Darstellung über sämtliche Neuronen zu verteilen. Beim Dropout schalten wir einen zufälligen, vorher bestimmten Anteil der Einheiten einer Schicht ab, indem wir ihre Werte beim Anlernen auf null setzen. Die so auszulassenden Neuronen werden – bei jeder Berechnung erneut – zufällig ausgewählt, was das Netz dazu zwingt, eine Darstellung zu erlernen, die auch nach dem Dropout funktioniert. Dieser Vorgang wird oft als Anlernen eines »Ensembles« mehrerer Netze beschrieben, was eine bessere Verallgemeinerbarkeit bedeutet. Verwenden wir das Netz hingegen später als Klassifikator für eine Vorhersage (bei der »Inferenz«), gibt es kein Dropout – dann wird das vollständige Netz eingesetzt.

Neben der Schicht, auf die wir Dropout anwenden möchten, ist in unserem Beispiel der einzige Parameter `keep_prob`, der Anteil der bei jedem Schritt funktionsfähigen Neuronen:

```
tf.nn.dropout(layer, keep_prob=keep_prob)
```

Um diesen Wert zu verändern (beim Testen müssen wir für diesen Wert `1.0` einstellen, da dort überhaupt kein Dropout erwünscht ist), verwenden wir ein Objekt vom Typ `tf.Variable` und übergeben einen Wert zum Anlernen (`.5`) und einen zweiten zum Testen (`1.0`).

Das Modell

Zunächst definieren wir einige Hilfsfunktionen zum Erstellen der Schichten, die wir in diesem Kapitel ausgiebig nutzen werden. Dadurch wird das eigentliche Modell kurz und übersichtlich. (Später im Buch werden wir mehrere Frameworks kennenlernen, die Bausteine für Deep Learning noch weiter abstrahieren. Mit ihnen können wir uns darauf konzentrieren, mit wenig Aufwand unsere Netze zu entwerfen, anstatt alle nötigen Elemente in mühsamer Kleinarbeit einzeln zu definieren.) Unsere Hilfsfunktionen sind:

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)


def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```



```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')


def conv_layer(input, shape):
    W = weight_variable(shape)
    b = bias_variable([shape[3]])
    return tf.nn.relu(conv2d(input, W) + b)


def full_layer(input, size):
    in_size = int(input.get_shape()[1])
    W = weight_variable([in_size, size])
    b = bias_variable([size])
    return tf.matmul(input, W) + b
```

Betrachten wir sie etwas genauer:

`weight_variable()`
: Diese Funktion legt die Gewichte für entweder eine vollständig verbundene oder eine Konvolutionsschicht an. Die Gewichte werden mit Zufallszahlen aus einer beidseitig abgeschnittenen Normalverteilung mit einer Standardabweichung von 0,1 initialisiert. Diese Art der Initialisierung ist recht gebräuchlich und führt in der Regel zu guten Ergebnissen (siehe auch den Hinweis auf zufällige Initialisierung weiter unten).

`bias_variable()`
: Diese Funktion definiert die Bias-Terme in einer vollständig verbundenen oder einer Konvolutionsschicht. Sie werden in beiden Fällen mit der Konstanten 0,1 initialisiert.

`conv2d()`
: Diese Funktion definiert die normalerweise eingesetzte Konvolution. Eine vollständige Konvolution (ohne Überspringen von Bildpunkten), bei der die Ausgabe genauso groß wie die Eingabe ist.

`max_pool_2x2`
: Diese Funktion stellt Max-Pooling auf die halbe Höhe und Breite der Eingabedaten ein, also auf insgesamt ein Viertel der Größe der Merkmalskarte.

`conv_layer()`
: Diese Funktion erstellt die eigentliche Konvolutionsschicht. Die mit `conv2d` definierte lineare Konvolution mit einem Bias-Term, gefolgt von der nichtlinearen ReLU-Aktivierungsfunktion.

`full_layer()`
: Erzeugt eine gewöhnliche vollständig verbundene Schicht mit einem Bias-Term. Wir haben hier keine ReLU-Aktivierungsfunktion hinzugefügt. Damit können wir diese Funktion auch für die Ausgabeschicht verwenden, bei der wir den nichtlinearen Teil nicht benötigen.

Nachdem wir diese Schichten definiert haben, können wir nun unser Modell erstellen (eine bildliche Darstellung dazu finden Sie in Abbildung 4-4):

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

x_image = tf.reshape(x, [-1, 28, 28, 1])
conv1 = conv_layer(x_image, shape=[5, 5, 1, 32])
conv1_pool = max_pool_2x2(conv1)

conv2 = conv_layer(conv1_pool, shape=[5, 5, 32, 64])
conv2_pool = max_pool_2x2(conv2)

conv2_flat = tf.reshape(conv2_pool, [-1, 7*7*64])
full_1 = tf.nn.relu(full_layer(conv2_flat, 1024))

keep_prob = tf.placeholder(tf.float32)
```

```
full1_drop = tf.nn.dropout(full_1, keep_prob=keep_prob)

y_conv = full_layer(full1_drop, 10)
```

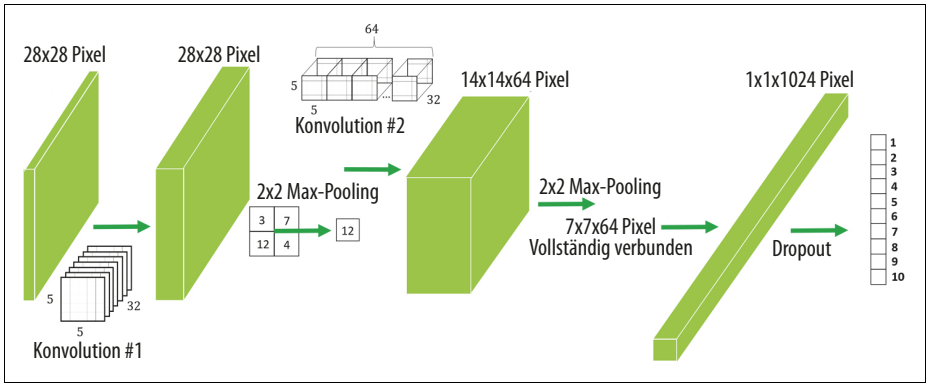

Abbildung 4-4: Eine Darstellung der verwendeten CNN-Architektur

Zufällige Initialisierung

Im vorigen Kapitel haben wir verschiedene Initialisierungsverfahren besprochen, darunter die zufällige Initialisierung, die wir für die Gewichte unserer Konvolutionsschicht einsetzen:

```
initial = tf.truncated_normal(shape, stddev=0.1)
```

Es ist viel darüber gesagt worden, wie wichtig die Initialisierung eines Deep-Learning-Netzes ist. Einfach ausgedrückt, kann ein schlecht ausgewähltes Initialisierungsverfahren dazu führen, dass der Anlernprozess »stecken bleibt« oder aus numerischen Gründen völlig scheitert. Mit zufälliger anstatt konstanter Initialisierung wird die Symmetrie zwischen den erlernten Merkmalen gebrochen, wodurch das Modell eine vielfältige und reichhaltige Darstellung der Daten erlernt. Ein begrenzter Wertebereich hilft unter anderem dabei, den Betrag der Gradienten unter Kontrolle zu halten, sodass das Netz effizient konvergieren kann.

Zu Beginn definieren wir die Platzhalter für Bilder `x` und korrekte Markierungen `y_`. Anschließend formen wir die Bilddaten in ein 2-D-Format mit der Größe 28 × 28 × 1 um. Bei unserem bisherigen MNIST-Modell mussten wir den räumlichen Aspekt der Daten nicht berücksichtigen, da alle Pixel unabhängig voneinander behandelt wurden. Dagegen verdanken Konvolutionsnetze einen wesentlichen Teil ihrer Vorhersagekraft der Ausnutzung räumlicher Bezüge in den Bilddaten.

Als Nächstes haben wir zwei aufeinanderfolgende Schichten für Konvolution und Pooling, jede davon mit 5 × 5 Konvolutionen und 64 Merkmalskarten, gefolgt von einer einzelnen vollständig verbundenen Schicht mit 1024 Einheiten. Vor dem Anwenden der vollständig verbundenen Schicht reihen wir alle Bildpunkte wieder zu einem einzelnen Vektor auf, da ihre räumliche Bedeutung von der vollständig verbundenen Schicht nicht mehr benötigt wird.

Beachten Sie, dass die Größe des Bilds nach den zwei Konvolutions- und Pooling-Schichten 7 × 7 × 64 beträgt. Das ursprüngliche Bild mit 28 × 28 Pixeln wird zuerst auf die Größe 14 × 14 reduziert und anschließend von den Pooling-Operationen auf 7 × 7 verdichtet. Die 64 ist die Anzahl der in der zweiten Konvolutionsschicht erzeugten Merkmalskarten. Die Gesamtzahl der im Modell angelernten Parameter, von der sich ein Großteil in der vollständig verbundenen Schicht befindet (von 7 × 7 × 64 auf 1024), beträgt 3,2 Millionen. Ohne Max-Pooling wäre diese Zahl 16 Mal so groß (28 × 28 × 64 × 1024 oder etwa 51 Millionen).

Die Ausgabe schließlich ist eine vollständig verbundene Schicht mit 10 Einheiten, was der Anzahl der Kategorien entspricht. (Der MNIST-Datensatz enthält handschriftliche Ziffern, daher ist die Anzahl möglicher Kategorien 10.)

Der Rest entspricht unserem ersten MNIST-Modell in Kapitel 2 mit geringfügigen Änderungen:

`train_accuracy`
: Wir geben alle 100 Schritte die Genauigkeit des Modells bzgl. der zum Anlernen verwendeten Daten aus. Das geschieht jeweils *vor* dem Anlernschritt und ist daher ein guter Schätzwert für die aktuelle Vorhersageleistung des Modells auf dem Anlerndatensatz.

`test_accuracy`
: Wir teilen den Testlauf in 10 Abschnitte zu je 1000 Bildern auf. Das ist vor allem bei wesentlich größeren Datensätzen von Bedeutung.

So sieht der vollständige Code aus:

```
mnist = input_data.read_data_sets(DATA_DIR, one_hot=True)

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_conv,
                                                                       labels=y_))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for i in range(STEPS):
        batch = mnist.train.next_batch(50)

        if i % 100 == 0:
            train_accuracy = sess.run(accuracy, feed_dict={x: batch[0],
                                                           y_: batch[1],
                                                           keep_prob: 1.0})
            print("Schritt {}, Genauigkeit Anlernen {}".format(i, train_accuracy))

        sess.run(train_step, feed_dict={x: batch[0], y_: batch[1],
                                        keep_prob: 0.5})

    X = mnist.test.images.reshape(10, 1000, 784)
```

```
    Y = mnist.test.labels.reshape(10, 1000, 10)
    test_accuracy = np.mean([sess.run(accuracy,
                            feed_dict={x:X[i], y_:Y[i],keep_prob:1.0})
                            for i in range(10)])

print("Genauigkeit Test: {}".format(test_accuracy))
```

Die Vorhersageleistung dieses Modells ist bereits relativ gut: Nach nur 5 Epochen, also 5000 Schritten mit Teildatensätzen der Größe 50, liegt es bereits zu mehr als 99 % richtig.[1]

Eine Liste von Modellen, die im Lauf der Jahre an diesem Datensatz ausprobiert wurden, sowie einige Ideen zur weiteren Verbesserung des Ergebnisses finden Sie unter *http://yann.lecun.com/exdb/mnist/*.

CIFAR10

*CIFAR10*ist ein weiterer Datensatz mit einer langen Tradition in Bilderkennung und maschinellem Lernen. Wie auch der MNIST-Datensatz, ist er ein gebräuchlicher Tauglichkeitstest für unterschiedliche Verfahren. CIFAR10 besteht aus 60000 Farbbildern mit je 32 × 32 Pixeln, von denen jedes zu einer von zehn Kategorien gehört: Flugzeuge, Autos, Vögel, Katzen, Hirsche, Hunde, Frösche, Pferde, Schiffe und Lastwagen.

Die führenden Deep-Learning-Modelle beherrschen die Klassifikation dieser Bilder ebenso gut wie der Mensch. In diesem Abschnitt beginnen wir mit wesentlich einfacheren Methoden, die sich relativ schnell ausführen lassen. Anschließend werden wir besprechen, worin die Unterschiede zwischen unseren und den modernen Modellen bestehen.

Laden des CIFAR10-Datensatzes

In diesem Abschnitt erstellen wir eine Datenverwaltung für den CIFAR10-Datensatz ähnlich der eingebauten Funktion `input_data.read_data_sets()`, die wir für die MNIST-Daten verwendet haben.[2]

Laden Sie sich zunächst die Python-Version des Datensatzesherunter und extrahieren Sie die Dateien in ein lokales Verzeichnis. Sie sollten nun die folgenden Dateien sehen:

1 Beim maschinellen Lernen und besonders beim Deep Learning ist mit einer *Epoche* ein einzelner Durchlauf aller Anlerndaten gemeint; das lernende Modell hat also jeden Lerndatenpunkt genau ein Mal gesehen.

2 Sie soll vor allem der Veranschaulichung dienen. Es gibt Open-Source-Bibliotheken, die solche Hilfsfunktionen für viele beliebte Datensätze bereitstellen. Sehen Sie sich dazu beispielsweise das Modul datasets in Keras (`keras.datasets`) an, und davon insbesondere `keras.datasets.cifar10`.

- *data_batch_1*, *data_batch_2*, *data_batch_3*, *data_batch_4*, *data_batch_5*
- *test_batch*
- *batches_meta*
- *readme.html*

Die Dateien mit Namen nach dem Muster *data_batch_X* enthalten serialisierte Anlerndaten, und *test_batch* ist eine ähnliche serialisierte Datei mit den Testdaten. Die Datei *batches_meta* enthält die Zuordnung von numerischen zu semantischen Markierungen. Die *.html*-Datei ist eine Kopie der Webseite des CIFAR-10-Datensatzes.

Da dieser Datensatz relativ klein ist, laden wir ihn komplett in den Arbeitsspeicher:

```
class CifarLoader(object):
    def __init__(self, source_files):
        self._source = source_files
        self._i = 0
        self.images = None
        self.labels = None

    def load(self):
        data = [unpickle(f) for f in self._source]
        images = np.vstack([d[b"data"] for d in data])
        n = len(images)
        self.images = images.reshape(n, 3, 32, 32).transpose(0, 2, 3, 1)\
                          .astype(float) / 255
        self.labels = one_hot(np.hstack([d[b"labels"] for d in data]), 10)
        return self

    def next_batch(self, batch_size):
        x, y = self.images[self._i:self._i+batch_size],
               self.labels[self._i:self._i+batch_size]
        self._i = (self._i + batch_size) % len(self.images)
        return x, y
```

Dazu verwenden wir folgende Hilfsfunktionen:

```
DATA_PATH = "/pfad/zu/CIFAR10"

def unpickle(file):
    with open(os.path.join(DATA_PATH, file), 'rb') as fo:
        dict = pickle.load(fo, encoding="bytes")
    return dict

def one_hot(vec, vals=10):
    n = len(vec)
    out = np.zeros((n, vals))
    out[range(n), vec] = 1
    return out
```

Die Funktion `unpickle()` liefert ein Dictionary mit den Feldern `data` und `labels`, die Bilddaten bzw. Markierungen enthalten. Die Funktion `one_hot()` wandelt die Mar-

kierungen von ganzen Zahlen (zwischen 0 und 9) in Vektoren der Länge 10 um, die eine 1 an der Position der jeweiligen Zahl und sonst lauter Nullen enthalten.

Schließlich erstellen wir einen Klasse zur Verwaltung der Daten, die sowohl die Anlerndaten als auch die Testdaten enthält:

```
class CifarDataManager(object):
    def __init__(self):
        self.train = CifarLoader(["data_batch_{}".format(i)
        for i in range(1, 6)])
            .load()
        self.test = CifarLoader(["test_batch"]).load()
```

Nun können wir diese Verwaltungsklasse verwenden, um einige der CIFAR10-Bilder mit Matplotlib darzustellen und einen besseren Eindruck des Datensatzes zu bekommen:

```
def display_cifar(images, size):
    n = len(images)
    plt.figure()
    plt.gca().set_axis_off()
    im = np.vstack([np.hstack([images[np.random.choice(n)] for i in range(size)])
                    for i in range(size)])
    plt.imshow(im)
    plt.show()

d = CifarDataManager()
print("Anzahl der Bilder zum Anlernen: {}".format(len(d.train.images)))
print("Anzahl der Markierungen zum Anlernen: {}".format(len(d.train.labels)))
print("Anzahl der Bilder zum Testen: {}".format(len(d.test.images)))
print("Anzahl der Markierungen zum Testen: {}".format(len(d.test.labels)))
images = d.train.images
display_cifar(images, 10)
```


Matplotlib

Matplotlib ist eine nützliche Python-Bibliothek für Diagramme, deren Aussehen und Verhalten an die Diagramme in Matlab angelehnt sind. Oft ist das die einfachste Möglichkeit, einen Datensatz schnell grafisch darzustellen und zu veranschaulichen.

Die Funktion `display_cifar()` akzeptiert die Parameter `images` (ein Iterable mit den Bildern) und `size` (die Anzahl anzuzeigender Bilder) und erzeugt ein `size×size`-Gitter mit Bildern. Dazu reiht sie die Bilder vertikal und horizontal zu einem großen Bild aneinander.

Bevor wir das Gitter mit den Bildern anzeigen, geben wir die Größe der Anlern- und Testdatensätze aus. Der CIFAR10-Datensatz enthält 50000 Bilder zum Anlernen und 10000 Bilder zum Testen:

```
Anzahl der Bilder zum Anlernen: 50000
Anzahl der Markierungen zum Anlernen: 50000
Anzahl der Bilder zum Testen: 10000
Anzahl der Markierungen zum Testen: 10000
```

Das so erstellte und in Abbildung 4-5 gezeigte Bild soll Ihnen einen Eindruck davon vermitteln, wie die CIFAR10-Bilder aussehen. Jedes dieser kleinen Bilder mit 32 × 32 Pixeln enthält einen einzigen vollständigen Gegenstand, der sowohl zentriert als auch selbst bei dieser Auflösung halbwegs gut erkennbar ist.

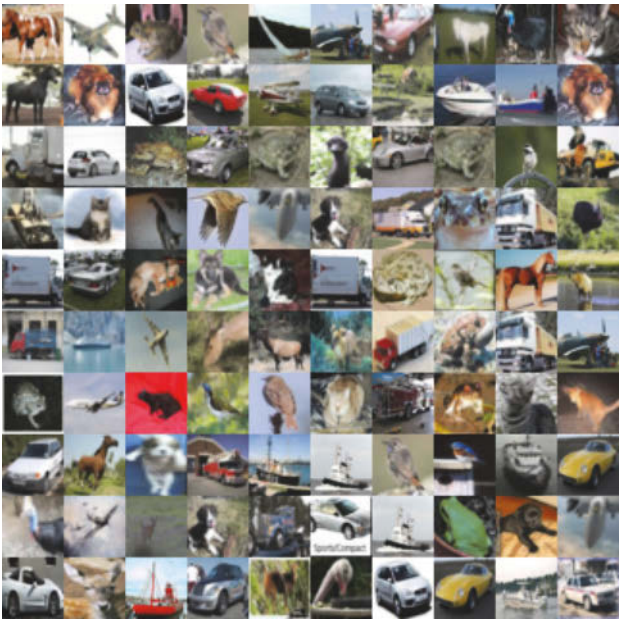

Abbildung 4-5: 100 zufällig ausgewählte CIFAR10-Bilder

Einfache CIFAR10-Modelle

Wir verwenden zunächst das Modell, das wir bereits erfolgreich für den MNIST-Datensatz eingesetzt hatten. Wie Sie sich erinnern werden, besteht der MNIST-Datensatz aus Graustufenbildern mit 28 × 28 Pixeln, die Bilder im CIFAR10-Datensatz haben dagegen 32 × 32 Pixel. Deshalb sind einige kleine Anpassungen bei der Erstellung des Berechnungsgraphen nötig:

```
cifar = CifarDataManager()

x = tf.placeholder(tf.float32, shape=[None, 32, 32, 3])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
keep_prob = tf.placeholder(tf.float32)

conv1 = conv_layer(x, shape=[5, 5, 3, 32])
conv1_pool = max_pool_2x2(conv1)

conv2 = conv_layer(conv1_pool, shape=[5, 5, 32, 64])
conv2_pool = max_pool_2x2(conv2)
conv2_flat = tf.reshape(conv2_pool, [-1, 8 * 8 * 64])
```

```
full_1 = tf.nn.relu(full_layer(conv2_flat, 1024))
full1_drop = tf.nn.dropout(full_1, keep_prob=keep_prob)

y_conv = full_layer(full1_drop, 10)

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_
conv,
                                                               labels=y_))
train_step = tf.train.AdamOptimizer(1e-3).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

def test(sess):
    X = cifar.test.images.reshape(10, 1000, 32, 32, 3)
    Y = cifar.test.labels.reshape(10, 1000, 10)
    acc = np.mean([sess.run(accuracy, feed_dict={x: X[i], y_: Y[i],
                                                 keep_prob: 1.0})
                   for i in range(10)])
    print("Genauigkeit: {:.4}%".format(acc * 100))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for i in range(STEPS):
        batch = cifar.train.next_batch(BATCH_SIZE)
        sess.run(train_step, feed_dict={x: batch[0], y_: batch[1],
                                        keep_prob: 0.5})

    test(sess)
```

Dieser erste Versuch erreicht innerhalb weniger Minuten eine Genauigkeit von etwa 70 % (bei einer Teildatensatz-Größe von 100; natürlich hängt das auch von der Hardware und Konfiguration ab). Ist das ein guter Wert? Im Moment erreichen die führenden Deep-Learning-Methoden auf diesem Datensatz eine Genauigkeit von über 95 %,[3] benötigen hierzu aber deutlich größere Modelle und normalerweise viele, viele Stunden Anlernzeit.

Es gibt einige Unterschiede zwischen diesem und dem bereits vorgestellten ähnlichen MNIST-Modell. Erstens bestehen die Eingabedaten aus Bildern der Größe 32 × 32 × 3, wobei die dritte Dimension die drei Farbkanäle sind:

```
x = tf.placeholder(tf.float32, shape=[None, 32, 32, 3])
```

Zweitens bleiben dementsprechend nach den zwei Pooling-Operationen 64 Merkmalskarten der Größe 8 × 8 übrig:

```
conv2_flat = tf.reshape(conv2_pool, [-1, 8 * 8 * 64])
```

3 Eine Liste von Methoden und den dazugehörigen Fachartikeln finden Sie unter Who Is the Best in CIFAR-10?

Schließlich legen wir der Bequemlichkeit halber den Testvorgang als eigene Funktion `test()` an und geben die Genauigkeitswerte beim Anlernen nicht aus (wozu wir aber nur denselben Code wie im MNIST-Modell wieder einfügen müssten).

Sobald wir ein Modell mit einer halbwegs akzeptablen Genauigkeit als Vergleichswert haben (egal ob diese von einem einfachen MNIST-Modell oder einem modernen Modell für einen anderen Datensatz abgeleitet ist), ist es gängige Praxis, es durch kleine Änderungen zu verbessern, bis es unseren Zweck erfüllt.

In diesem Fall lassen wir alles unverändert und fügen lediglich eine dritte Konvolutionsschicht mit 128 Merkmalskarten und Dropout hinzu. Wir verringern außerdem die Anzahl von Einheiten in der vollständig verbundenen Schicht von 1024 auf 512:

```
x = tf.placeholder(tf.float32, shape=[None, 32, 32, 3])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
keep_prob = tf.placeholder(tf.float32)

conv1 = conv_layer(x, shape=[5, 5, 3, 32])
conv1_pool = max_pool_2x2(conv1)

conv2 = conv_layer(conv1_pool, shape=[5, 5, 32, 64])
conv2_pool = max_pool_2x2(conv2)

conv3 = conv_layer(conv2_pool, shape=[5, 5, 64, 128])
conv3_pool = max_pool_2x2(conv3)
conv3_flat = tf.reshape(conv3_pool, [-1, 4 * 4 * 128])
conv3_drop = tf.nn.dropout(conv3_flat, keep_prob=keep_prob)

full_1 = tf.nn.relu(full_layer(conv3_drop, 512))
full1_drop = tf.nn.dropout(full_1, keep_prob=keep_prob)

y_conv = full_layer(full1_drop, 10)
```

Dieses Modell benötigt etwas mehr Zeit (aber auch ohne besondere Hardware noch deutlich unter einer Stunde) und erreicht eine Genauigkeit von etwa 75 %.

Das ist immer noch weit entfernt von den besten bekannten Methoden. Es gibt einige Ansatzpunkte, die voneinander unabhängig helfen können, diese Lücke zu schließen:

Modellgröße
: Die meisten bei solchen Datensätzen erfolgreichen Methoden verwenden weitaus tiefere Netze mit wesentlich mehr anpassbaren Parametern.

Zusätzliche Arten von Schichten und Methoden
: Es gibt weitere verbreitete Arten von Schichten, die häufig mit den hier vorgestellten zusammen eingesetzt werden, beispielsweise Local Response Normalization.

Tricks zur Optimierung
: Mehr dazu später!

Fachspezifisches Wissen
: Eine Vorverarbeitung der Daten mithilfe fachspezifischen Wissens macht oft eine Menge aus. In diesem Fall wäre das die herkömmliche Bildverarbeitung.

Datenvermehrung
: Es kann helfen, neue Anlerndaten auf Grundlage der vorhandenen hinzuzufügen. Wenn beispielsweise das Bild eines Hunds horizontal gespiegelt wird, ist es mit Sicherheit noch immer das Bild eines Hunds (wie sieht es aber bei einer vertikalen Spiegelung aus?). Kleine Verschiebungen und Rotationen werden ebenfalls oft angewandt.

Wiederverwendung erfolgreicher Methoden und Architekturen
: Wie bei den meisten Ingenieurstätigkeiten lohnt es sich häufig, eine bewährte Methode an die eigenen Bedürfnisse anzupassen. Beim Deep Learning werden oft bereits angelernte Modelle weiter optimiert.

Das letzte Modell, das wir in diesem Kapitel vorstellen möchten, ist eine kleinere Version eines Modelltyps, der auf diesem Datensatz sehr gute Resultate erzielt. Das Modell ist immer noch kompakt und schnell und erreicht nach etwa 150 Epochen eine Genauigkeit von ungefähr 83 %:

```
C1, C2, C3 = 30, 50, 80
F1 = 500

conv1_1 = conv_layer(x, shape=[3, 3, 3, C1])
conv1_2 = conv_layer(conv1_1, shape=[3, 3, C1, C1])
conv1_3 = conv_layer(conv1_2, shape=[3, 3, C1, C1])
conv1_pool = max_pool_2x2(conv1_3)
conv1_drop = tf.nn.dropout(conv1_pool, keep_prob=keep_prob)

conv2_1 = conv_layer(conv1_drop, shape=[3, 3, C1, C2])
conv2_2 = conv_layer(conv2_1, shape=[3, 3, C2, C2])
conv2_3 = conv_layer(conv2_2, shape=[3, 3, C2, C2])
conv2_pool = max_pool_2x2(conv2_3)
conv2_drop = tf.nn.dropout(conv2_pool, keep_prob=keep_prob)

conv3_1 = conv_layer(conv2_drop, shape=[3, 3, C2, C3])
conv3_2 = conv_layer(conv3_1, shape=[3, 3, C3, C3])
conv3_3 = conv_layer(conv3_2, shape=[3, 3, C3, C3])
conv3_pool = tf.nn.max_pool(conv3_3, ksize=[1, 8, 8, 1], strides=[1, 8, 8, 1],
                            padding='SAME')
conv3_flat = tf.reshape(conv3_pool, [-1, C3])
conv3_drop = tf.nn.dropout(conv3_flat, keep_prob=keep_prob)

full1 = tf.nn.relu(full_layer(conv3_flat, F1))
full1_drop = tf.nn.dropout(full1, keep_prob=keep_prob)

y_conv = full_layer(full1_drop, 10)
```


Dieses Modell enthält drei Gruppen von Konvolutionsschichten und anschließend die vollständig verbundene und die Ausgabeschicht, die wir bereits einige Male gesehen haben. Jede Gruppe enthält drei aufeinanderfolgende Konvolutionsschichten, gefolgt von einer einzelnen Pooling-Schicht und Dropout.

Die Konstanten `C1`, `C2` und `C3`bestimmen die Anzahl der Merkmalskarten in jeder Schicht der drei Gruppen. Die Konstante `F1` legt die Anzahl der Einheiten in der vollständig vebundenen Schicht fest.

Nach der dritten Gruppe mit Konvolutionsschichten verwenden wir eine Max-Pooling-Schicht der Größe 8 × 8:

```
conv3_pool = tf.nn.max_pool(conv3_3, ksize=[1, 8, 8, 1], strides=[1, 8, 8, 1],
                            padding='SAME')
```

Da an diesem Punkt alle Merkmalskarten die Größe 8 × 8 besitzen (von den ersten zwei Poolings verkleinert jedes die 32-×-32-Pixel-Bilder in jeder Achsenrichtung um die Hälfte), wird jede der Merkmalskarten über das gesamte Bild aggregiert und nur der maximale Wert beibehalten. Die Anzahl der Merkmalskarten im dritten Block wurde auf 80 gesetzt, sodass nach dem Max-Pooling die Darstellung der Daten auf nur 80 Zahlen reduziert ist. Damit bleibt die Gesamtgröße des Modells überschaubar, weil die Anzahl der Parameter beim Übergang zur vollständig verbundenen Schicht 80 × 500 nicht übersteigt.

Zusammenfassung

In diesem Kapitel haben wir Konvolutionsnetze und ihre typischen Bestandteile vorgestellt. Wenn Sie erst einmal in der Lage sind, kleine Modelle korrekt zum Laufen zu bekommen, versuchen Sie es mit größeren und tieferen nach den gleichen Bauprinzipien. Auch wenn Sie stets in der neuesten Fachliteratur nachschlagen können, was funktioniert und was nicht, können Sie auch durch Versuch und Irrtum selbst eine Menge herausbekommen. In den nächsten Kapiteln werden wir sehen, wie man mit Text und Sequenzdaten arbeiten und Abstraktionen von TensorFlow verwenden kann, um CNN-Modelle einfacher zu konstruieren.