

## KAPITEL 11

---

# Trainieren von Deep-Learning-Netzen

In Kapitel 10 haben wir künstliche neuronale Netze besprochen und unser erstes Deep-Learning-Netz trainiert. Es war aber ein sehr flaches DNN mit nur wenigen verborgenen Schichten. Wie lässt sich eine komplexe Aufgabe wie das Erkennen Hunderter Gegenstände in hochauflösenden Bildern angehen? Dazu müssen Sie ein weitaus tieferes DNN mit zehn oder noch viel mehr Schichten aus jeweils Hunderten Neuronen und Hunderttausenden Verbindungen trainieren. Das wird kein Spaziergang, und die folgenden sind nur ein paar der Probleme, denen Sie sich gegenübersehen können:

- Bei Deep-Learning-Netzen tritt eventuell das Problem der *schwindenden Gradienten* oder das verwandte Problem der *explodierenden Gradienten* auf. Dabei werden die Gradienten immer kleiner oder immer größer, während man sich beim Training rückwärts durch das DNN bewegt. Beide Probleme erschweren das Trainieren der ersten Schichten erheblich.
- Sie haben eventuell nicht ausreichend Trainingsdaten für solch ein großes Netz, oder das Labeln ist zu teuer.
- Das Trainieren bei einem derart großen Netz ist extrem langsam.
- Bei einem Modell mit Millionen Parametern besteht eine ernste Gefahr, die Trainingsdaten zu overfitten, insbesondere wenn es nicht ausreichend Trainingsinstanzen gibt oder diese zu verrauscht sind.

In diesem Kapitel wenden wir uns nacheinander jedem dieser Probleme zu und stellen Techniken zu deren Lösung vor. Wir beginnen mit dem Problem schwindender und explodierender Gradienten und probieren einige der beliebtesten Lösungsstrategien aus. Als Nächstes werden wir uns das Transfer Learning und unüberwachtes Pretraining anschauen, das Ihnen bei komplexen Aufgaben auch dann helfen kann, wenn Sie nur wenige gelabelte Daten haben. Anschließend werden wir unterschiedliche Optimierer betrachten, die bei großen Modellen den Trainingsvorgang erheblich beschleunigen. Schließlich werden wir einige bei großen neuronalen Netzen verbreitete Regularisierungstechniken behandeln.

Mit diesen Werkzeugen werden Sie in der Lage sein, sehr tiefre Netze zu trainieren: Willkommen beim Deep Learning!

## Das Problem schwindender/explodierender Gradienten

Wie in Kapitel 10 besprochen, arbeitet sich der Backpropagation-Algorithmus von der Ausgabeschicht zur Eingabeschicht vor und berechnet unterwegs den Fehlergradienten. Hat der Algorithmus erst einmal den Fehlergradienten der Kostenfunktion nach jedem Parameter im Netz bestimmt, werden diese Gradienten zum Aktualisieren jedes Parameters im Netz verwendet.

Leider werden die Gradienten mit diesem Algorithmus zu den niedrigeren Schichten hin immer kleiner und kleiner. In der Folge ändert das Gradientenverfahren die Gewichte der Verbindungen in den ersten Schichten kaum, und das Training konvergiert nie zu einer annehmbaren Lösung. Dies bezeichnet man als das Problem der *schwindenden Gradienten*. In einigen Fällen kann auch das Gegenteil passieren: Die Gradienten werden größer und größer, sodass die Gewichte vieler Schichten eine extrem große Änderung erfahren und der Algorithmus divergiert. Das bezeichnet man als das Problem der *explodierenden Gradienten*, das vor allem in rekurrenten neuronalen Netzen auftritt (siehe Kapitel 15). Allgemeiner ausgedrückt, sind die Gradienten in Deep-Learning-Netzen instabil; die Lerngeschwindigkeiten unterschiedlicher Schichten weichen stark voneinander ab.

Dieses ungünstige Verhalten wurde schon vor einer Weile beobachtet und war einer der Gründe, aus dem man Deep-Learning-Netze bis in die frühen 2000er-Jahre beiseitegelegt hatte. Es war nicht klar, was die Gradienten dazu bewegte, beim Trainieren eines DNN so instabil zu sein, aber ein Artikel (<https://homl.info/47>) aus dem Jahr 2010 von Xavier Glorot und Yoshua Bengio<sup>1</sup> fand einige Hauptverdächtige, darunter die Kombination der beliebten logistischen Aktivierungsfunktion mit der damals verbreiteten zufälligen Initialisierung der Gewichte, genauer die zufällige Initialisierung mit einer Normalverteilung mit dem Mittelwert 0 und einer Standardabweichung von 1. Kurz, die Autoren wiesen nach, dass mit diesem Muster aus Aktivierungsfunktion und Initialisierung die Varianz der Ausgaben jeder Schicht höher als die Varianz der Eingaben wird. Beim Durchschreiten des Netzes erhöht sich die Varianz von Schicht zu Schicht, bis die Aktivierungsfunktion in den späteren Schichten gesättigt ist. Das Problem, dass die logistische Funktion einen Mittelwert von 0,5 anstatt 0 hat, wird dadurch verschlimmert (der Tangens hyperbolicus besitzt einen Mittelwert von 0 und verhält sich in Deep-Learning-Netzen etwas besser als die logistische Funktion).

Wenn Sie sich die logistische Aktivierungsfunktion ansehen (siehe Abbildung 11-1), erkennen Sie, dass die Funktion bei großen Eingabewerten (negativ oder positiv) eine Sättigung bei 0 oder 1 erreicht und ihre Ableitung sehr nah bei 0 liegt.

---

<sup>1</sup> Xavier Glorot und Yoshua Bengio, »Understanding the Difficulty of Training Deep Feedforward Neural Networks«, *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.

Beim Backpropagation-Verfahren gibt es daher praktisch keinen Gradienten, der sich durch das Netzwerk propagieren ließe, und das bisschen Gradient wird in den späteren Schichten auch noch ausgedünnt. Es bleibt also für die ersten Schichten wirklich nichts übrig.

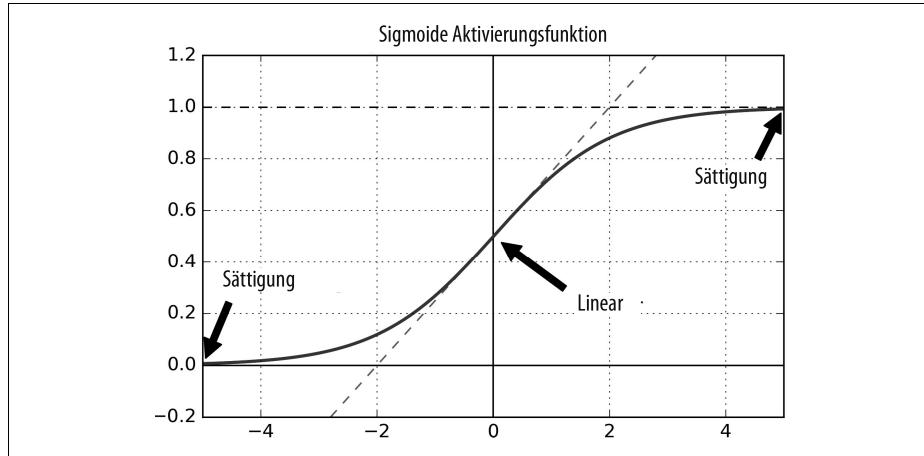


Abbildung 11-1: Sättigung der logistischen Aktivierungsfunktion

## Initialisierung nach Glorot und He

In ihrem Artikel empfehlen Glorot und Bengio einen Weg, das Problem der instabilen Gradienten deutlich zu mildern. Sie weisen darauf hin, dass das Signal in beide Richtungen fließen können muss: bei der Vorhersage vorwärts und beim Propagieren der Gradienten rückwärts. Wir möchten weder, dass das Signal unterwegs verhungert, noch, dass es explodiert und zu einer Sättigung führt. Die Autoren argumentieren, dass für einen guten Signalfluss die Varianz der Ausgaben und die der Eingaben jeder Schicht gleich sein müssen.<sup>2</sup> Außerdem müssen die Gradienten bei der Backpropagation vor und nach einer Schicht die gleiche Varianz besitzen (bitte lesen Sie den Artikel, falls Sie an den mathematischen Details interessiert sind). Beides ist nicht möglich, es sei denn, eine Schicht hat gleich viele eingehende Verbindungen und Neuronen (diese Zahlen werden als *Fan-in* und *Fan-out* der Schicht bezeichnet), aber die Autoren schlugen einen praktisch gut funktionierenden Kompromiss vor: Die Gewichte der Verbindungen müssen zufällig gesetzt werden, wie in Formel 11-1 beschrieben, wobei gilt  $\text{fan}_{\text{avg}} = (\text{fan}_{\text{in}} + \text{fan}_{\text{out}})/2$ . Diese

<sup>2</sup> Hier besteht eine Analogie: Wenn Sie den Verstärkerknopf eines Mikrofons zu nah gen null drehen, hört niemand Ihre Stimme, aber wenn Sie ihn zu nah an das Maximum stellen, ist Ihre Stimme gesättigt, und niemand versteht, was Sie sagen. Stellen Sie sich nun eine Kette mehrerer solcher Verstärker vor: Alle müssen richtig eingestellt sein, damit Ihre Stimme am Ende der Kette laut und deutlich zu verstehen ist. Ihre Stimme muss aus jedem Verstärker mit der gleichen Amplitude herauskommen, mit der sie hineinkommt.

Initialisierungsstrategie wird *Initialisierung nach Xavier* oder *Initialisierung nach Glorot* genannt – nach dem ersten Autor des Artikels.

*Formel 11-1: Initialisierung nach Glorot (beim Verwenden der logistischen Aktivierungsfunktion)*

$$\text{Normalverteilung mit Mittelwert 0 und Standardabweichung } \sigma^2 = \sqrt{\frac{1}{\text{fan}_{\text{avg}}}}$$

$$\text{Oder eine Gleichverteilung zwischen } -r \text{ und } +r, \text{ mit } r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$$

Ersetzen Sie in Formel 11-1  $\text{fan}_{\text{avg}}$  durch  $\text{fan}_{\text{in}}$ , erhalten Sie eine Initialisierungsstrategie, die Yann LeCun in den 1990ern vorgeschlagen hat. Er nannte sie *Initialisierung nach LeCun*. Genevieve Orr und Klaus-Robert Müller empfahlen sie sogar in ihrem Buch *Neural Networks: Tricks of the Trade* (Springer) aus dem Jahr 1998. Die LeCun-Initialisierung ist äquivalent zur Glorot-Initialisierung, wenn  $\text{fan}_{\text{in}} = \text{fan}_{\text{out}}$ . Es dauerte über ein Jahrzehnt, bis die Forscher erkannten, wie wichtig dieser Trick ist. Der Einsatz der Glorot-Initialisierung kann das Training deutlich beschleunigen, und sie ist einer der Tricks, die zum Erfolg von Deep Learning beigetragen haben.

Es wurden ähnliche Strategien für andere Aktivierungsfunktionen vorgeschlagen.<sup>3</sup> Diese unterscheiden sich nur in der Größe der Varianz und dem Einsatz von  $\text{fan}_{\text{out}}$  oder  $\text{fan}_{\text{in}}$ , wie Sie in Tabelle 11-1 sehen (für die gleichförmige Verteilung berechnen Sie einfach  $r = \sqrt{3\sigma^2}$ ). Die Initialisierungsstrategie (<https://homl.info/48>) für die ReLU-Aktivierungsfunktion (und ihre Varianten, einschließlich der kurz beschriebenen ELU-Aktivierung) wird manchmal *Initialisierung nach He* genannt – nach dem ersten Autor des Artikels. Die SELU-Aktivierungsfunktion wird weiter unten noch erläutert. Sie sollte zusammen mit der LeCun-Initialisierung eingesetzt werden (vorzugsweise mit einer Normalverteilung, wie wir sehen werden).

*Tabelle 11-1: Initialisierungsparameter für jede Art von Aktivierungsfunktion*

Initialisierung	Aktivierungsfunktionen	$\sigma^2$ Normal
Glorot	keine, tanh, logistisch, Softmax	$1 / \text{fan}_{\text{avg}}$
He	ReLU und Varianten	$2 / \text{fan}_{\text{in}}$
LeCun	SELU	$1 / \text{fan}_{\text{in}}$

Standardmäßig nutzt Keras die Glorot-Initialisierung mit einer Gleichverteilung. Beim Erstellen einer Schicht können Sie die He-Initialisierung einsetzen, indem Sie

---

<sup>3</sup> Zum Beispiel Kaiming He et al., »Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification«, *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.

`kernel_initializer="he_uniform"` oder `kernel_initializer="he_normal"` verwenden, zum Beispiel so:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Wollen Sie die He-Initialisierung mit einer gleichförmigen Verteilung nutzen, die aber auf  $fan_{avg}$  statt auf  $fan_{in}$  basiert, können Sie den `VarianceScaling`-Initialisierer wie folgt einsetzen:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                    distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

## Nicht sättigende Aktivierungsfunktionen

Eine Erkenntnis des Artikels von Glorot und Bengio aus dem Jahr 2010 war, dass die Auswahl einer ungeeigneten Aktivierungsfunktion eine Teilursache des Problems instabiler Gradienten war. Bis dahin hatten die meisten Menschen angenommen, dass sigmoide Aktivierungsfunktionen eine ausgezeichnete Wahl sein müssten, zumal Mutter Natur in etwa diese in biologischen Neuronen verwendet. Wie sich aber herausstellte, verhalten sich in Deep-Learning-Netzen andere Aktivierungsfunktionen viel günstiger, besonders die ReLU-Aktivierungsfunktion, vor allem weil sie bei positiven Werten keine Sättigung erreicht (und weil sie sich schnell berechnen lässt).

Leider ist auch die ReLU-Aktivierungsfunktion nicht perfekt. Sie krankt an einem Problem, das als *sterbende ReLUs* bekannt ist: Beim Trainieren sterben einige Neuronen praktisch ab, das bedeutet, sie geben nichts anderes als 0 aus. In einigen Fällen kommt es vor, dass die Hälfte der Neuronen im Netzwerk tot sind, besonders wenn Sie eine große Lernrate eingestellt haben. Ein Neuron stirbt, wenn seine Gewichte so angepasst werden, dass die gewichtete Summe seiner Eingaben für alle Instanzen im Trainingsdatensatz negativ ist. Geschieht das, wird immer nur 0 ausgegeben, und die Gradientenmethode hat keine Auswirkungen mehr, weil der Gradient der ReLU-Funktion bei negativen Werten null ist.<sup>4</sup>

Um dieses Problem zu lösen, können Sie eine Variante der ReLU-Funktion verwenden, z.B. *Leaky ReLU*. Diese Funktion ist definiert als  $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$  (siehe Abbildung 11-2). Der Hyperparameter  $\alpha$  definiert, wie stark die Funktion »leckt«: Er entspricht der Steigung der Funktion bei  $z < 0$  und beträgt normalerweise 0,01. Diese geringe Steigung stellt sicher, dass Leaky ReLUs niemals sterben; sie können in ein langes Koma fallen, können aber wieder erwachen. Ein im Jahr 2015 erschienener Artikel (<https://homl.info/49>)<sup>5</sup> verglich mehrere Varianten der

---

<sup>4</sup> Sofern es nicht Teil der ersten verborgenen Schicht ist, kann ein totes Neuron manchmal wieder zum Leben erweckt werden: Die Gradientenmethode kann durchaus Neuronen in tieferen Schichten so verändern, dass die gewichtete Summe der Eingaben des toten Neurons wieder positiv wird.

<sup>5</sup> Bing Xu et al., »Empirical Evaluation of Rectified Activations in Convolutional Network«, arXiv preprint arXiv:1505.00853 (2015).

ReLU-Aktivierungsfunktion und kam zu dem Schluss, dass die Leaky-Varianten der ursprünglichen ReLU-Aktivierungsfunktion stets überlegen sind. Das Setzen von  $\alpha = 0,2$  (ein riesiges Leck) schien zu einer höheren Leistung als  $\alpha = 0,01$  (ein kleines Leck) zu führen. Die Autoren werteten auch die *randomisierte Leaky ReLU* (RReLU) aus, bei der  $\alpha$  beim Trainieren aus einem vorgegebenen Bereich zufällig ausgewählt wird und beim Testen auf einen Durchschnittswert gesetzt wird. Diese Funktion schnitt ebenfalls recht gut ab und schien als Regularisierung zu fungieren (also das Risiko für Overfitting der Trainingsdaten zu senken). Schließlich wurde auch die *parametrisierte Leaky ReLU* (PReLU) betrachtet, bei der  $\alpha$  beim Training erlernt wird (diese PReLU ist kein Hyperparameter, sondern wird ein Parameter, der vom Backpropagation-Verfahren modifiziert werden kann wie jeder andere Parameter). Dieses Verfahren schnitt auf großen Bilddatensätzen sehr viel besser als ReLU ab, neigte bei kleineren Datensätzen aber zum Overfitting.

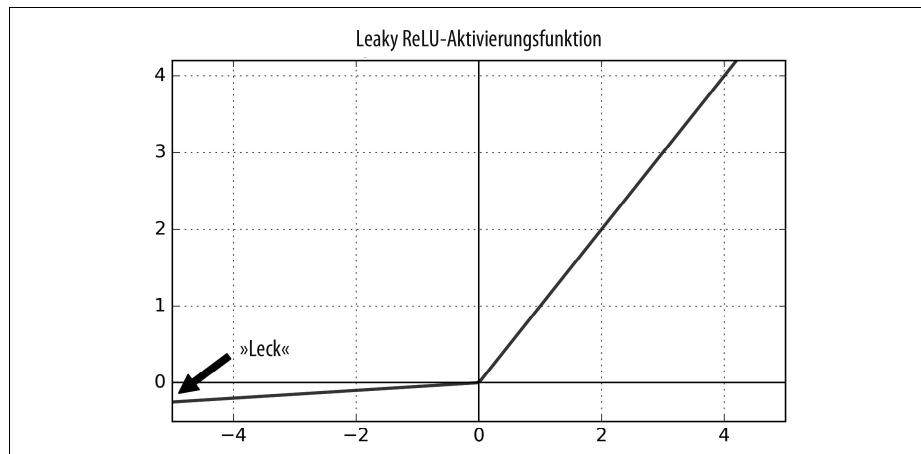


Abbildung 11-2: Leaky ReLU: wie ReLU, aber mit einer kleinen Steigung für negative Werte

Schließlich schlug ein Artikel (<https://hml.info/50>) aus dem Jahr 2015 von Djork-Arné Clevert et al.<sup>6</sup> eine neue Aktivierungsfunktion namens *Exponential Linear Unit* (ELU) vor, die im Experiment sämtliche ReLU-Varianten aus dem Feld schlug: Die Trainingszeit verringerte sich, und das neuronale Netz erzielte auf den Testdaten eine höhere Leistung. Die Funktion ist in Abbildung 11-3 dargestellt und ihre Definition in Formel 11-2 ausgeschrieben.

Formel 11-2: ELU-Aktivierungsfunktion

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{wenn } z < 0 \\ z & \text{wenn } z \geq 0 \end{cases}$$

<sup>6</sup> Djork-Arné Clevert et al., »Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)«, *Proceedings of the International Conference on Learning Representations* (2016).

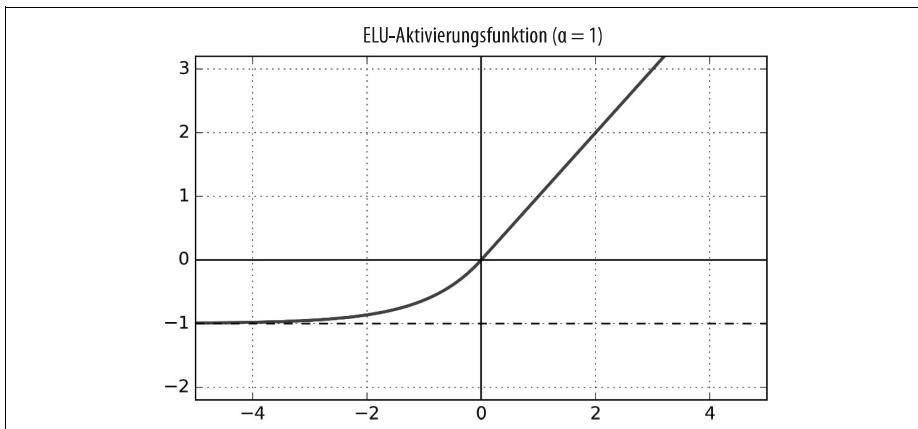


Abbildung 11-3: ELU-Aktivierungsfunktion

Sie sieht der ReLU-Funktion sehr ähnlich, weist aber einige große Unterschiede auf:

- Sie nimmt bei  $z < 0$  negative Werte an, wodurch das Neuron eine durchschnittliche Ausgabe um 0 haben kann und das Problem schwindender Gradienten bekämpft. Der Hyperparameter  $\alpha$  definiert den Wert, dem sich die ELU-Funktion bei stark negativen Werten von  $z$  annähert. Er wird normalerweise auf 1 gesetzt, aber Sie können ihn wie jeden anderen Hyperparameter verändern.
- Die Ableitung für  $z < 0$  ist ungleich 0, was das Problem toter Neuronen vermeidet.
- Ist  $\alpha = 1$ , ist die Funktion an allen Stellen glatt, auch bei  $z = 0$ , was das Gradientenverfahren beschleunigt, da es links und rechts von  $z = 0$  weniger umherspringt.

Der Hauptnachteil der ELU-Aktivierungsfunktion ist, dass sie sich langsamer als ReLU und seine Varianten berechnen lässt (wegen der Exponentialfunktion), aber beim Trainieren wird dies durch die schnellere Konvergenz kompensiert. Beim Testen ist ein ELU-Netz jedoch langsamer als ein ReLU-Netz.

Im Jahr 2017 hat dann ein Artikel (<https://homl.info/selu>) von Günter Klambauer et al.<sup>7</sup> die Scaled-ELU-Aktivierungsfunktion (SELU) vorgestellt: Wie ihr Name schon andeutet, handelt es sich bei ihr um eine skalierte Variante der ELU-Aktivierungsfunktion. Die Autoren haben gezeigt, dass das Netz *selbstnormalisierend* sein wird, wenn Sie ein neuronales Netz nur aus dichten Schichten aufbauen und alle verborgenen Schichten die SELU-Aktivierungsfunktion nutzen: Die Ausgabe jeder Schicht tendiert dann dazu, während des Trainings einen Mittelwert von 0 und eine

<sup>7</sup> Günter Klambauer et al., »Self-Normalizing Neural Networks«, *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 972–981.

Standardabweichung von 1 zu erreichen, was das Problem der verschwindenden oder explodierenden Gradienten löst. Im Ergebnis funktioniert die SELU-Aktivierungsfunktion häufig signifikant besser für solche neuronalen Netze (insbesondere Deep-Learning-Netze). Es gibt allerdings ein paar Bedingungen, damit die Selbstnormalisierung auch geschehen kann (im Artikel finden Sie die mathematischen Begründungen dafür):

- Die Eingabemerkmale müssen standardisiert sein (Mittelwert 0 und Standardabweichung 1).
- Die Gewichte jeder verborgenen Schicht müssen mit der LeCun-Normal-Initialisierung belegt werden. In Keras müssen Sie dafür `kernel_initializer="lecun_normal"` setzen.
- Die Architektur des Netzes muss sequenziell sein. Versuchen Sie, SELU in nicht sequenziellen Architekturen einzusetzen, wie zum Beispiel bei rekurrenten Netzen (siehe Kapitel 15) oder in Netzen mit Skip-Verbindungen (also Verbindungen, die Schichten überspringen, wie zum Beispiel in Wide-&-Deep-Netzen), ist die Selbstnormalisierung leider nicht garantiert, daher wird SELU nicht unbedingt besser funktionieren als andere Aktivierungsfunktionen.
- Der Artikel garantiert eine Selbstnormalisierung nur, wenn alle Schichten dicht sind, aber manche Forscher haben angemerkt, dass die SELU-Aktivierungsfunktion auch die Leistung in Convolutional Neural Networks verbessern kann (siehe Kapitel 14).



Welche Aktivierungsfunktion sollten Sie also in den verborgenen Schichten Ihrer Deep-Learning-Netze verwenden? Auch wenn es im Einzelfall Unterschiede gibt, gilt als Faustregel SELU > ELU > Leaky ReLU (und Varianten) > ReLU > tanh > logistisch. Verhindert die Netzarchitektur eine Selbstnormalisierung, kann ELU besser funktionieren als SELU (da SELU bei  $z = 0$  nicht stetig ist). Wenn es Ihnen sehr auf Geschwindigkeit zur Laufzeit ankommt, sollten Sie Leaky ReLUs den Vorzug vor ELUs geben. Wenn Sie sich nicht mit noch einem Hyperparameter befassen möchten, können Sie die oben empfohlenen Standardwerte für  $\alpha$  verwenden (zum Beispiel 0,3 für Leaky ReLU). Wenn Sie zusätzliche Zeit und Rechenkapazität übrig haben, können Sie weitere Aktivierungsfunktionen über Kreuzvalidierung mit einbeziehen, insbesondere RReLU, falls Ihr Netz zu Overfitting neigt, und PReLU, wenn Ihr Trainingsdatensatz sehr groß ist. Und weil ReLU die (bisher) am häufigsten verwendete Aktivierungsfunktion ist, enthalten viele Bibliotheken und Hardwarebeschleuniger ReLU-spezifische Optimierungen – ist Ihnen also Geschwindigkeit am wichtigsten, kann ReLU weiterhin die beste Wahl sein.

Um die Leaky-ReLU-Aktivierungsfunktion zu verwenden, erzeugen Sie eine Leaky ReLU-Schicht und fügen sie zu Ihrem Modell direkt nach der Schicht hinzu, auf die Sie sie anwenden wollen:

```

model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])

```

Für PReLU ersetzen Sie LeakyReLU(alpha=0.2) durch PReLU(). Es gibt in Keras aktuell keine offizielle Implementierung von RReLU, aber Sie können sie ziemlich einfach selbst implementieren (um zu lernen, wie das geht, schauen Sie sich die Übungen am Ende von Kapitel 12 an).

Zur SELU-Aktivierung setzen Sie beim Erstellen einer Schicht activation="selu" und kernel\_initializer="lecun\_normal":

```

layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")

```

## Batchnormalisierung

Obwohl die Initialisierung nach He zusammen mit der ELU (oder einer Variante der ReLU) das Problem der schwindenden/explodierenden Gradienten zu Beginn des Trainierens deutlich reduzieren kann, sind sie keine Garantie dafür, dass die Probleme nicht später zurückkehren.

In einem Artikel (<https://homl.info/51>) aus dem Jahr 2015<sup>8</sup> schlagen Sergey Ioffe und Christian Szegedy eine Technik namens *Batchnormalisierung* (BN) vor, um diese Probleme anzugehen. Die Technik besteht aus dem Hinzufügen einer Operation kurz vor der Aktivierungsfunktion in jeder verborgenen Schicht des Modells. Dabei werden einfach die Eingaben auf null zentriert und normalisiert, und das Ergebnis wird anschließend mit zwei neuen Parametern pro Schicht skaliert und verschoben (ein Parameter zum Skalieren, einer zum Verschieben). Anders ausgedrückt, kann das Modell durch diesen Vorgang die optimale Skalierung und den Mittelwert für die Eingaben jeder Schicht erlernen. In vielen Fällen brauchen Sie Ihren Trainingsdatensatz nicht zu standardisieren (zum Beispiel mit einem StandardScaler), wenn Sie eine BN-Schicht als allererste Schicht in Ihr neuronales Netz einfügen – die BN-Schicht erledigt das schon für Sie (nun, zumindest näherungsweise, da sie sich immer nur einen Batch gleichzeitig anschaut, zudem kann sie jedes Eingabemerkmal umskalieren und verschieben).

Um die Eingaben auf null zu zentrieren und zu normalisieren, muss der Algorithmus Mittelwert und Standardabweichung der Eingaben schätzen. Dazu werden Mittelwert und Standardabweichung der Eingaben aus dem aktuellen Mini-Batch ausgewertet (daher der Name »Batchnormalisierung«). Die gesamte Prozedur ist in Formel 11-3 zusammengefasst.

---

<sup>8</sup> Sergey Ioffe und Christian Szegedy, »Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift«, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.

Formel 11-3: Algorithmus zur Batchnormalisierung

$$\begin{aligned}
 1. \quad \boldsymbol{\mu}_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \\
 2. \quad \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2 \\
 3. \quad \hat{\mathbf{x}}^{(i)} &= \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
 4. \quad \mathbf{z}^{(i)} &= \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta
 \end{aligned}$$

Dabei gilt:

- $\boldsymbol{\mu}_B$  ist der Vektor der Eingabe-Mittelwerte für den gesamten Mini-Batch  $B$  (er enthält einen Mittelwert pro Eingabe).
- $\sigma_B^2$  ist der Vektor mit den Eingabe-Standardabweichungen, ebenfalls für den gesamten Mini-Batch bestimmt (er enthält eine Standardabweichung pro Eingabe).
- $m_B$  ist die Anzahl Datenpunkte im Mini-Batch.
- $\hat{\mathbf{x}}^{(i)}$  ist der Vektor mit auf null zentrierten und normalisierten Eingaben für Instanz  $i$ .
- $\gamma$  ist der Vektor mit den Ausgabe-Skalierungsparametern für die Schicht (er enthält einen Skalierungsparameter pro Eingabe).
- $\otimes$  steht für eine elementweise Multiplikation (jeder Eingabewert wird mit seinem zugehörigen Ausgabe-Skalierungsfaktor multipliziert).
- $\beta$  ist der Vektor mit den Parametern zum Verschieben der Schicht (Offset). Er enthält einen Offset-Parameter pro Eingabe. Jede Eingabe wird durch ihren zugehörigen Verschiebeparameter verschoben.
- $\epsilon$  ist eine kleine Zahl zum Vermeiden einer Division durch null (normalerweise  $10^{-5}$ ). Diese wird als *Smoothing-Term* bezeichnet.
- $\mathbf{z}^{(i)}$  ist die Ausgabe der BN-Operation: Sie ist eine skalierte und verschobene Version der Eingaben.

Während des Trainings standardisiert BN also seine Eingaben, um sie dann umzu-skalieren und zu verschieben. Gut! Wie sieht es mit dem Testen aus? Nun, das ist nicht so einfach. Tatsächlich müssen wir eventuell Vorhersagen für einzelne Instanzen statt für ganze Batches treffen. Und selbst wenn wir einen Batch haben, kann er zu klein sein, oder die Instanzen sind nicht unabhängig und gleichförmig verteilt, sodass das Berechnen von Statistiken über die Batchinstanzen unzuverlässig wäre. Eine Lösung bestünde darin, bis zum Ende des Trainings zu warten, dann den gesamten Trainingsdatensatz durch das neuronale Netz zu schicken und Mittelwert und Standardabweichung jeder Eingabe der BN-Schicht zu berechnen. Diese »finalen« Eingabe-Mittelwerte und -Standardabweichungen könnten dann statt der

Batch-Eingabe-Mittelwerte und -Standardabweichungen während der Vorhersagen zum Einsatz kommen. Aber die meisten Implementierungen der Batchnormalisierung schätzen diese abschließenden Statistiken während des Trainings mithilfe eines gleitenden Durchschnitts der Mittelwerte und Standardabweichungen der Schichteingaben. Das macht Keras automatisch, wenn Sie die `BatchNormalization`-Schicht einsetzen. Fassen wir zusammen: Vier Parametervektoren werden in jeder batchnormalisierten Schicht gelernt –  $\gamma$  (der Ausgabe-Skalierungs-Vektor) und  $\beta$  (der Ausgabe-Offset-Vektor) werden während der normalen Backpropagation gelernt, während  $\mu$  (der finale Eingabe-Mittelwert-Vektor) und  $\sigma$  (der finale Eingabe-Standardabweichungs-Vektor) mithilfe eines exponentiellen gleitenden Mittelwerts entstehen. Beachten Sie, dass  $\mu$  und  $\sigma$  während des Trainings geschätzt werden, aber nur danach zum Einsatz kommen (um die Batch-Eingabe-Mittelwerte und -Standardabweichungen in Formel 11-3 zu ersetzen).

Ioffe und Szegedy haben gezeigt, dass die Batchnormalisierung alle Deep-Learning-Netze, mit denen sie experimentiert haben, deutlich verbessert hat, was zu einer großen Verbesserung bei der ImageNet-Klassifikationsaufgabe führte (ImageNet ist eine große Datenbank mit Bildern, die in vielen Kategorien klassifiziert sind und oft zum Überprüfen von Bilderkennungssystemen dient). Das Problem der verschwindenden Gradienten wurde so deutlich reduziert, dass sogar gesättigte Aktivierungsfunktionen wie `tanh` oder die logistische Aktivierungsfunktion zum Einsatz kommen konnten. Die Netze reagierten zudem deutlich unempfindlicher auf die Gewichtsinitialisierung. Die Autoren konnten viel höhere Lernraten nutzen und den Lernprozess signifikant beschleunigen. Insbesondere merken sie an:

Angewendet auf ein aktuelles Bildklassifikationsmodell erreicht Batchnormalisierung die gleiche Genauigkeit mit 14 Mal weniger Trainingsschritten und schlägt das Ursprungsmo dell deutlich. [...] Bei einem Ensemble batchnormalisierter Netze verbessern wir das beste veröffentlichte Ergebnis bei der ImageNet-Klassifikation: Wir erreichen 4,9% Top-5-Validierungsfehler (und 4,8% Testfehler) und sind damit besser als menschliche Klassifikatoren.

Und schließlich – wie ein Geschenk, das immer mehr liefert – verhält sich die Batchnormalisierung wie ein Regulierer und verringert die Notwendigkeit für andere Regulierungstechniken (wie das später in diesem Kapitel beschriebene Drop-out).

Aber durch Batchnormalisierung wird das Modell komplexer (auch wenn – wie schon besprochen – dadurch die Notwendigkeit des Normalisierens der Eingabedaten wegfallen kann). Zudem gibt es Zusatzkosten zur Laufzeit: Das neuronale Netz macht langsamere Vorhersagen aufgrund der zusätzlichen Berechnungen, die auf jeder Schicht erforderlich sind. Zum Glück ist es oft möglich, die BN-Schicht nach dem Training mit der vorherigen Schicht zu verschmelzen und damit die Zusatzkosten zu vermeiden. Das geschieht durch das Aktualisieren der Gewichte und Biase der vorherigen Schicht, sodass diese direkt die Ausgaben mit passender Skalierung und Offset ermittelt. Berechnet beispielsweise die vorherige Schicht  $XW + b$ , wird die BN-Schicht  $\gamma \otimes (XW + b - \mu) / \sigma + \beta$  berechnen (wenn wir mal den

Smoothing-Term  $\varepsilon$  im Nenner ignorieren). Definieren wir  $\mathbf{W}' = \gamma \otimes \mathbf{W} / \sigma$  und  $\mathbf{b}' = \gamma \otimes (\mathbf{b} - \mu) / \sigma + \beta$ , vereinfacht sich die Gleichung zu  $\mathbf{X}\mathbf{W}' + \mathbf{b}'$ . Ersetzen wir also in der vorherigen Schicht Gewichte und Biase ( $\mathbf{W}$  und  $\mathbf{b}$ ) durch die aktualisierten Gewichte und Biase ( $\mathbf{W}'$  und  $\mathbf{b}'$ ), können wir die BN-Schicht loswerden (der Optimierer von TFLite tut das automatisch, siehe Kapitel 19).



Sie werden eventuell feststellen, dass das Training eher langsam abläuft, weil jede Epoche beim Einsatz der Batchnormalisierung viel mehr Zeit verbraucht. Das wird im Allgemeinen dadurch ausgeglichen, dass die Konvergenz mit BN viel schneller geschieht und damit weniger Epochen zum Erreichen der gleichen Leistung notwendig sind. Insgesamt wird die *Wall Time* meist kürzer sein (das ist die Zeit, die von Ihrer Uhr an der Wand gemessen wird).

### Implementieren der Batchnormalisierung mit Keras

Wie das meiste bei Keras ist auch das Implementieren der Batchnormalisierung einfach und intuitiv. Fügen Sie einfach eine `BatchNormalization`-Schicht vor oder nach jeder Aktivierungsfunktion verborgener Schichten ein und ergänzen Sie optional eine BN-Schicht als erste in Ihrem Modell. Das folgende Modell wendet beispielsweise BN nach jeder verborgenen Schicht und als erste Schicht an (nach dem Verflachen der Eingabebilder):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Das ist alles! In diesem kleinen Beispiel mit nur zwei verborgenen Schichten ist es unwahrscheinlich, dass die Batchnormalisierung eine sehr positive Auswirkung haben wird – aber für tiefere Netze kann es einen deutlichen Unterschied ausmachen.

Schauen wir uns die Modellzusammenfassung an:

```
>>> model.summary()
Model: "sequential_3"
=====
Layer (type)          Output Shape       Param #
=====
flatten_3 (Flatten)   (None, 784)        0
batch_normalization_v2 (BatchNormalization) (None, 784)        3136
dense_50 (Dense)      (None, 300)        235500
batch_normalization_v2_1 (BatchNormalization) (None, 300)        1200
```

dense_51 (Dense)	(None, 100)	30100
batch_normalization_v2_2 (Batch Normalization)	(None, 100)	400
dense_52 (Dense)	(None, 10)	1010
<hr/>		
Total params: 271,346		
Trainable params: 268,978		
Non-trainable params: 2,368		

Wie Sie sehen, fügt jede BN-Schicht vier Parameter pro Eingabe hinzu:  $\gamma$ ,  $\beta$ ,  $\mu$  und  $\sigma$  (so ergänzt beispielsweise die erste BN-Schicht 3.136 Parameter, also  $4 \times 784$ ). Die letzten beiden Parameter  $\mu$  und  $\sigma$  sind die gleitenden Mittelwerte – sie sind nicht von der Backpropagation betroffen, daher nennt Keras sie »nicht trainierbar«<sup>9</sup> (wenn Sie die Gesamtsumme der BN-Parameter zusammenzählen [3136 + 1200 + 400] und durch 2 teilen, erhalten Sie 2.368, was der Gesamtzahl nicht trainierbarer Parameter in diesem Modell entspricht).

Schauen wir uns die Parameter der ersten BN-Schicht an. Zwei sind trainierbar (durch Backpropagation), zwei nicht:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Erstellen Sie nun eine BN-Schicht in Keras, werden auch zwei Operationen angelegt, die von Keras bei jeder Iteration während des Trainings aufgerufen werden. Diese Operationen aktualisieren die gleitenden Mittelwerte. Da wir das TensorFlow-Backend verwenden, handelt es sich bei diesen Operationen um TensorFlow-Operationen (wir werden diese in Kapitel 12 behandeln):

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

Die Autoren des BN-Artikels plädieren dafür, die BN-Schichten vor den Aktivierungsfunktionen hinzuzufügen und nicht erst dahinter (wie wir es getan haben). Darüber gibt es Diskussionen, aber es hängt auch von der Aufgabe ab – Sie können damit experimentieren, um zu sehen, welche Option mit Ihrem Datensatz am besten funktioniert. Um die BN-Schichten vor die Aktivierungsfunktionen zu setzen, müssen Sie die Funktionen aus den verborgenen Schichten entfernen und als eigene Schichten hinter den BN-Schichten einfügen. Und da eine Batchnormalisierungsschicht einen Offset-Parameter pro Eingabe besitzt, können Sie den Bias-Term aus

---

<sup>9</sup> Sie werden aber während des Trainings basierend auf den Trainingsdaten geschätzt, daher sind sie durchaus trainierbar. In Keras heißt »nicht trainierbar« also eigentlich »nicht von der Backpropagation betroffen«.

der vorherigen Schicht entfernen (übergeben Sie beim Erstellen einfach `use_bias=False`):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Die Klasse `BatchNormalization` besitzt eine Reihe von Hyperparametern, an denen Sie drehen können. Die Standardwerte sind im Allgemeinen schon gut, aber gelegentlich müssen Sie das `momentum` anpassen. Dieser Hyperparameter wird von der `BatchNormalization`-Schicht genutzt, wenn sie die exponentiellen gleitenden Mittelwerte aktualisiert – bei einem neuen Wert  $v$  (also einem neuen Vektor mit Eingabe-Mittelwerten oder -Standardabweichungen, der mit dem aktuellen Batch berechnet wurde) aktualisiert die Schicht den gleitenden Mittelwert  $\hat{v}$  über diese Gleichung:

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

Ein gutes Momentum liegt meist nahe an 1 – zum Beispiel 0,9, 0,99 oder 0,999 (Sie wollen mehr Neunen für größere Datensätze und kleinere Mini-Batches haben).

Ein weiterer wichtiger Hyperparameter ist `axis`: Er bestimmt, welche Achse normalisiert werden soll. Standardwert ist `-1`, womit die letzte Achse normalisiert wird (mit Mittelwerten und Standardabweichungen, die aus den *anderen* Achsen berechnet wurden). Ist der Eingabebatch zweidimensional (dann ist die Batchform [*Batchgröße, Features*]), wird jedes Eingabemerkmals basierend auf dem Mittelwert und der Standardabweichung normalisiert, die über alle Instanzen im Batch berechnet wurden. So wird beispielsweise die erste BN-Schicht im vorherigen Code beispiel jedes der 784 Eingangsmerkmale unabhängig voneinander normalisieren (sowie umskalieren und verschieben). Verschieben wir die erste BN-Schicht vor die `Flatten`-Schicht, wird der Eingabebatch dreidimensional sein, und die Form [*Batchgröße, Höhe, Breite*] haben – die BN-Schicht wird daher 28 Mittelwerte und 28 Standardabweichungen berechnen (eine pro Pixelspalte, berechnet über alle Instanzen im Batch und über alle Zeilen in den Spalten) und alle Pixel in einer gegebenen Spalte mit dem gleichen Mittelwert und der gleichen Standardabweichung normalisieren. Es wird auch nur 28 Skalierungsparameter und 28 Offset-Parameter geben. Wollen Sie stattdessen jedes einzelne der 784 Pixel unabhängig behandeln, sollten Sie `axis=[1, 2]` setzen.

Beachten Sie, dass die BN-Schicht nicht die gleichen Berechnungen während des und nach dem Training durchführt: Sie nutzt Batchstatistik während des Trainings

und die »finale« Statistik nach dem Training (also die abschließenden Werte der gleitenden Mittelwerte). Werfen wir einen Blick in diese Klasse, um zu sehen, wie das geschieht:

```
class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]
```

In der Methode `call()` werden die Berechnungen durchgeführt – wie Sie sehen, besitzt sie ein zusätzliches Argument `training`, das standardmäßig auf `None` gesetzt ist. Die Methode `fit()` setzt sie aber während des Trainings auf `1`. Müssen Sie eine eigene Schicht schreiben und muss sich diese im Training und beim Testen unterschiedlich verhalten, fügen Sie deren Methode `call()` ebenfalls ein Argument `training` hinzu und nutzen dieses in der Methode, um zu entscheiden, was zu berechnen ist<sup>10</sup> (wir werden eigene Schichten in Kapitel 12 behandeln).

`BatchNormalization` gehört in Deep-Learning-Netzen inzwischen zu den am häufigsten eingesetzten Schichten – sie wird in den Diagrammen sogar häufig weggelassen, da davon ausgegangen wird, dass sie hinter jeder Schicht eingefügt ist. Aber ein aktueller Artikel (<https://homl.info/fixup>) von Hongyi Zhang et al.<sup>11</sup> kann diese Annahme eventuell ändern: Durch eine neue *Fixed-Update-* (*Fixup-*) Gewichtsinitialisierungstechnik haben es die Autoren geschafft, ein sehr tiefes neuronales Netz (10. 000 Schichten!) ohne BN zu trainieren und dabei beste Leistungen bei komplexen Bildklassifikationsaufgaben zu erreichen. Da dies aktuellste Forschung ist, sollten Sie aber vielleicht auf zusätzliche Untersuchungen warten, die diese Erkenntnisse untermauern, bevor Sie die Batchnormalisierung wegwerfen.

## Gradient Clipping

Eine weitere beliebte Technik, um das Problem der explodierenden Gradienten zu entschärfen, ist, die Gradienten während der Backpropagation zu kappen, sodass sie niemals einen festgelegten Schwellenwert überschreiten (hilfreich vor allem bei rekurrenten neuronalen Netzen, siehe Kapitel 15). Man nennt diese Technik *Gradient Clipping* (<https://homl.info/52>).<sup>12</sup> Diese wird meist in Recurrent Neural Networks eingesetzt, da sich dort die Batchnormalisierung nur schwer verwenden lässt (wie wir in Kapitel 15 sehen werden). Für andere Arten von Netzen ist die BN meist ausreichend.

---

10 Die Keras-API spezifiziert auch eine Funktion `keras.backend.learning_phase()`, die während des Trainings `1` und ansonsten `0` zurückgeben sollte,

11 Hongyi Zhang et al., »Fixup Initialization: Residual Learning Without Normalization«, arXiv preprint arXiv:1901.09321 (2019).

12 Razvan Pascanu et al., »On the Difficulty of Training Recurrent Neural Networks«, *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.

In Keras müssen Sie zum Implementieren des Gradient Clipping nur beim Erstellen eines Optimierers das Argument `clipvalue` oder `clipnorm` setzen, zum Beispiel:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

Dieser Optimierer beschneidet jede Komponente des Gradientenvektors auf einen Wert zwischen  $-1,0$  und  $1,0$ . Damit werden alle partiellen Ableitungen des Verlusts (bezüglich jedes trainierbaren Parameters) auf den Bereich zwischen  $-1,0$  und  $1,0$  beschränkt. Der Grenzwert ist ein Hyperparameter, den Sie anpassen können. Beachten Sie, dass er die Richtung des Gradientenvektors verändern kann. Hat beispielsweise der ursprüngliche Gradientenvektor den Wert  $[0,9; 100,0]$ , weist er nahezu in die Richtung der zweiten Achse; nach dem Beschneiden erhalten Sie aber  $[0,9; 1,0]$ , was ungefähr diagonal zwischen den beiden Achsen steht. In der Praxis funktioniert dieser Ansatz gut. Wollen Sie sichergehen, dass das Gradient Clipping die Richtung des Gradientenvektors nicht ändert, sollten Sie die Norm begrenzen, indem Sie `clipnorm` statt `clipvalue` setzen. So wird der gesamte Gradient beschnitten, wenn seine  $\ell_2$ -Norm größer als der ausgewählte Grenzwert ist. Setzen Sie beispielsweise `clipnorm=1.0`, wird der Vektor  $[0,9; 100,0]$  auf  $[0,00899964; 0,9999595]$  begrenzt. Damit wird seine Richtung beibehalten, aber die erste Komponente fast ausgelöscht. Beobachten Sie, dass die Gradienten während des Trainings explodieren (sie können die Größe der Gradienten über TensorBoard verfolgen), ist es einen Versuch wert, die Werte oder die Norm mit unterschiedlichen Grenzwerten zu beschränken und zu sehen, welche Option mit dem Validierungsdatensatz am besten funktioniert.

## Wiederverwenden vortrainierter Schichten

Es ist im Allgemeinen keine gute Idee, ein sehr großes DNN von Anfang an zu trainieren: Stattdessen sollten Sie stets versuchen, ein existierendes neuronales Netz zu finden, das eine ähnliche Aufgabe erledigt (wir werden in Kapitel 14 besprechen, wie Sie solche finden). Dann können Sie die ersten Schichten dieses Netzes wiederverwenden: Man bezeichnet dies als *Transfer Learning*, und es beschleunigt nicht nur das Trainieren erheblich, sondern erfordert auch wesentlich weniger Trainingsdaten.

Nehmen Sie beispielsweise an, Ihnen stünde ein DNN zur Verfügung, das zur Klassifizierung von Bildern in 100 unterschiedliche Kategorien wie Tiere, Pflanzen, Fahrzeuge und Alltagsgegenstände trainiert wurde. Sie möchten ein DNN trainieren, mit dem sich bestimmte Arten von Fahrzeugen klassifizieren lassen. Diese Aufgaben sind einander sehr ähnlich, daher sollten Sie Teile des ersten Netzes wieder verwenden (siehe Abbildung 11-4).



Wenn die Bilder bei Ihrer neuen Aufgabe nicht die gleiche Größe haben wie die in der ursprünglichen Aufgabe verwendeten, müssen Sie meist einen Vorverarbeitungsschritt einbauen, der die Bilder auf die vom ursprünglichen Modell erwartete Größe skaliert.

Im Allgemeinen funktioniert Transfer Learning nur, wenn die Eingabedaten auf niedriger Ebene ähnliche Eigenschaften haben.

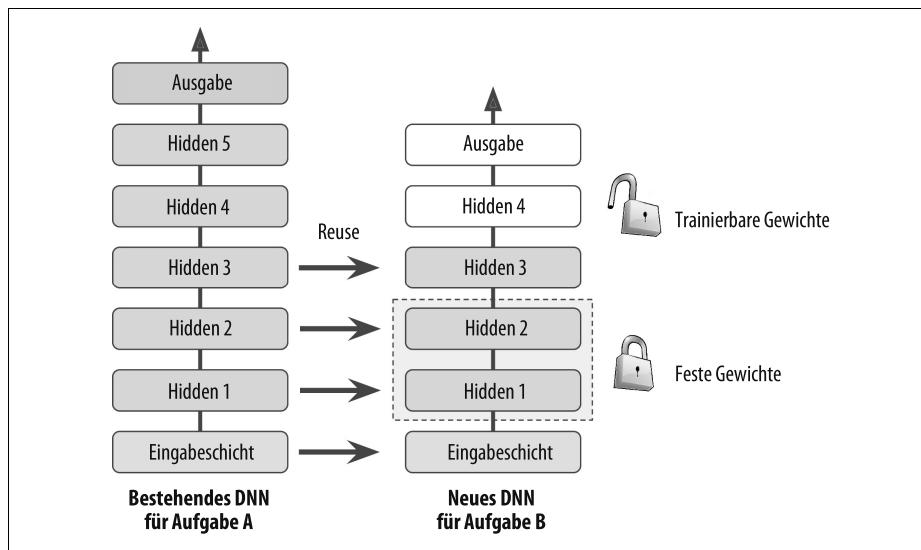


Abbildung 11-4: Wiederverwenden vortrainierter Schichten

Die Ausgabeschicht des ursprünglichen Modells sollte normalerweise ersetzt werden, da sie für die neue Aufgabe meist nicht nützlich ist und es eventuell nicht einmal die richtige Anzahl an Ausgaben gibt.

Genauso sind die oberen verborgenen Schichten des ursprünglichen Modells eher nicht so hilfreich wie die unteren Schichten, da sich die High-Level-Eigenschaften, die für die neue Aufgabe am nützlichsten sind, deutlich von denen unterscheiden können, die für die Ursprungsauflage hilfreich waren. Sie müssen eben die richtige Zahl an Schichten finden, die sich wiederverwenden lassen.



Je ähnlicher sich die Aufgaben sind, desto mehr Schichten können Sie wiederverwenden (beginnend mit den unteren Schichten). Bei sehr ähnlichen Aufgaben können Sie versuchen, alle verborgenen Schichten beizubehalten und die Ausgabeschicht zu ersetzen.

Versuchen Sie, alle wiederverwendeten Schichten zunächst einzufrieren (also ihre Gewichte nicht trainierbar zu machen, sodass die Gradientenmethode sie nicht verändert), um dann Ihr Modell zu trainieren und zu sehen, wie gut es funktioniert. Probieren Sie dann, eine oder zwei der obersten verborgenen Schichten wieder »aufzutauen«, damit die Backpropagation sie anpassen kann – prüfen Sie, ob sich die Leistung dadurch verbessert. Je mehr Trainingsdaten Sie haben, desto mehr Schichten können Sie auftauen. Es ist auch nützlich, dabei die Lernrate zu verringern – so vermeiden Sie, Ihre sorgsam angepassten Gewichte durcheinanderzubringen.

Erreichen Sie immer noch keine gute Performance und haben Sie nur wenige Trainingsdaten, versuchen Sie, die oberste verborgene Schicht (oder mehrere davon) zu verwerfen und die verbleibenden verborgenen Schichten wieder einzufrieren. Sie können das so lange ausprobieren, bis Sie die richtige Zahl von wiederzuverwendenden Schichten gefunden haben. Stehen Ihnen sehr viele Trainingsdaten zur Verfügung, können Sie versuchen, die obersten verborgenen Schichten zu ersetzen, statt sie wegzuerwerfen, oder sogar noch mehr Schichten hinzuzufügen.

## Transfer Learning mit Keras

Schauen wir uns ein Beispiel an. Stellen Sie sich vor, der Fashion-MNIST-Datensatz würde nur acht Kategorien enthalten – beispielsweise alle außer Sandale und Shirt. Jemand hat ein Keras-Modell für diesen Datensatz gebaut und trainiert und eine ausreichend gute Leistung erreicht (>90% Genauigkeit). Nennen wir es Modell A. Sie wollen nun eine andere Aufgabe angehen: Sie haben Bilder von Sandalen und Shirts und wollen einen binären Klassifikator trainieren (positiv = Shirt, negativ = Sandale). Ihr Datensatz ist ziemlich klein, Sie haben nur 200 gelabelte Bilder. Trainieren Sie ein neues Modell für diese Aufgabe (nennen wir es Modell B) mit der gleichen Architektur wie Modell A, wird sich dieses ziemlich gut schlagen (97,2% Genauigkeit). Aber da es sich um eine viel einfachere Aufgabe handelt (es sind nur zwei Kategorien), haben Sie auf mehr gehofft. Während Sie Ihren ersten Kaffee trinken, geht Ihnen auf, dass Ihre Aufgabe der alten Aufgabe ziemlich ähnelt, also kann vielleicht Transfer Learning helfen? Finden wir es heraus.

Zuerst müssen Sie Modell A laden und basierend auf dessen Schichten ein neues Modell erstellen. Wir verwenden dazu alle Schichten außer der Ausgabeschicht:

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Beachten Sie, dass sich `model_A` und `model_B_on_A` nun ein paar gemeinsame Schichten teilen. Trainieren Sie `model_B_on_A`, beeinflusst das auch `model_A`. Wollen Sie das vermeiden, müssen Sie `model_A` klonen, bevor Sie dessen Schichten wiederverwenden. Dazu übertragen Sie die Architektur von Modell A mit `clone_model()` und kopieren dann die Gewichte (da `clone_model()` das nicht tut):

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Jetzt können Sie `model_B_on_A` für Aufgabe B trainieren, aber da die neue Ausgabeschicht mit Zufallswerten initialisiert wurde, wird das zu großen Fehlern führen (zumindest während der ersten paar Epochen), und Sie erhalten große Fehlergradienzen, die die wiederverwendeten Gradienten zerstören können. Um das zu vermeiden, können Sie die wiederverwendeten Schichten während der ersten Epochen einfrieren und der neuen Schicht etwas Zeit zum Erlernen vernünftiger Gewichte geben. Dazu setzen Sie das Attribut `trainable` jeder Schicht auf `False` und kompilieren das Modell:

```

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                      metrics=["accuracy"])

```



Sie müssen Ihr Modell immer kompilieren, nachdem Sie Schichten eingefroren oder aufgetaut haben.

Jetzt können Sie das Modell für ein paar Epochen trainieren, dann die wiederverwendeten Schichten auftauen (wodurch das Modell erneut kompiliert werden muss) und mit dem Training fortfahren, um die wiederverwendeten Schichten für Aufgabe B genauer abzustimmen. Nach dem Auftauen der Schichten ist es meist eine gute Idee, die Lernrate zu verringern, um erneut das Beschädigen der wiederverwendeten Gewichte zu vermeiden:

```

history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # Standard für lr ist 1e-2
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                      metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))

```

Und was ist dabei herausgekommen? Nun, die Testgenauigkeit des Modells liegt bei 99,25%, was heißtt, dass das Transfer Learning die Fehlerrate von 2,8% auf nahezu 0,7% verringert hat! Das ist ein Faktor von vier.

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

Hat Sie das überzeugt? Sollte es aber nicht – ich habe geschummelt! Ich habe viele Konfigurationen ausprobiert, bis ich eine fand, die eine starke Verbesserung brachte. Versuchen Sie, die Kategorien oder die Zufallswerte zu verändern, werden Sie feststellen, dass die Verbesserung im Allgemeinen klein ist, verschwindet oder sich sogar ins Negative verkehrt. Ich habe die Daten so lange gefoltert, bis sie gestanden haben. Sieht ein Artikel zu gut aus, sollten Sie misstrauisch werden: Eventuell hilft diese neue, schicke Technik gar nicht so viel (oder vielleicht macht sie die Ergebnisse sogar schlechter), aber die Autoren haben sehr viele Varianten ausprobiert und nur über die besten Ergebnisse berichtet (die möglicherweise durch pures Glück entstanden sind), ohne zu erwähnen, wie viele Fehlversuche sie vorher hatten. Die meiste Zeit geschieht das gar nicht aus Böswilligkeit, aber es ist mit ein Grund dafür, dass so viele Ergebnisse in der Wissenschaft nie reproduziert werden können.

Warum habe ich geschummelt? Es zeigt sich, dass Transfer Learning mit kleinen dichten Netzen nicht gut funktioniert, vermutlich weil kleine Netze auch nur wenige Muster erlernen und dichte Netze sehr spezifische Muster lernen, die in anderen Aufgaben nicht unbedingt nützlich sind. Transfer Learning funktioniert am besten mit tiefen Convolutional Neural Networks, die eher Feature-Detektoren erlernen, die allgemeiner arbeiten (vor allem in den unteren Schichten). Wir werden uns das Transfer Learning in Kapitel 14 nochmals anschauen und dabei die gerade behandelten Techniken einsetzen (und dann auch nicht schummeln, das verspreche ich!).

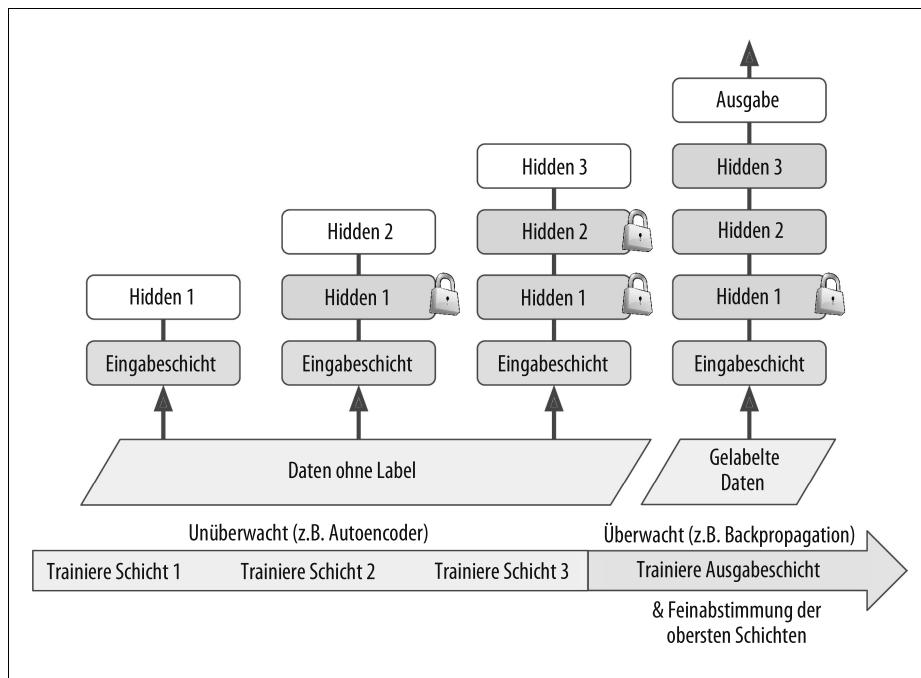
## Unüberwachtes Vortrainieren

Nehmen wir einmal an, Sie möchten eine komplexe Aufgabe bearbeiten, und es stehen Ihnen dafür nicht sehr viele gelabelte Trainingsdaten zur Verfügung. Leider finden Sie kein für eine ähnliche Aufgabe trainiertes Modell. Geben Sie nicht gleich auf! Zuerst sollten Sie natürlich versuchen, weitere gelabelte Trainingsdaten zu finden, aber falls dies nicht möglich ist, können Sie immer noch ein *unüberwachtes Vortraining* durchführen (siehe Abbildung 11-5). Natürlich ist es oft günstig, ungelabelte Trainingsbeispiele zu erhalten, aber es ist teuer, sie mit Labels zu versehen. Wenn Ihnen also reichlich ungelabelte Trainingsdaten zur Verfügung stehen, können Sie versuchen, damit ein unüberwachtes Modell zu trainieren, wie zum Beispiel einen Autoencoder oder ein Generative Adversarial Network (siehe Kapitel 17). Dann können Sie die unteren Schichten des Autoencoder oder des GAN-Diskriminators wiederverwenden, die Ausgabeschicht für Ihre Aufgabe daraufsetzen und das so entstandene Netz mithilfe überwachten Lernens feinjustieren (also mit den gelabelten Trainingsbeispielen).

Diese Technik haben Geoffrey Hinton und sein Team 2006 genutzt, und sie hat zum Revival neuronaler Netze und zum Erfolg von Deep Learning geführt. Bis 2010 war unüberwachtes Vortrainieren – typischerweise mit Restricted Boltzmann Machines (RBMs, siehe Anhang E) – die Norm für tiefe Netze, und erst nachdem das Problem der verschwindenden Gradienten eingedämmt werden konnte, fand das Trainieren von DNNs rein mit überwachtem Lernen weitere Verbreitung. Unüberwachtes Vortrainieren (heutzutage meist nicht mehr mit RBMs, sondern eher mit Autoencodern oder GANs) ist immer noch eine gute Option, wenn Sie eine komplexe Aufgabe lösen müssen, kein ähnliches Modell wiederverwenden können und nur wenige gelabelte, aber viele ungelabelte Trainingsdaten haben.

In den frühen Tagen des Deep Learning war es schwierig, tiefe Modelle zu trainieren, daher wurde eine Technik namens *Greedy Layer-Wise Pretraining* (siehe Abbildung 11-5) genutzt. Erst wurde ein unüberwachtes Modell mit einer einzelnen Schicht trainiert – meist ein RBM –, dann wurde diese Schicht eingefroren und eine weitere daraufgesetzt. Dann wurde das Modell wieder trainiert (eigentlich nur die neue Schicht), diese neue Schicht wieder eingefroren und so weiter. Heutzutage ist es viel einfacher: Die Leute trainieren im Allgemeinen das vollständige unüber-

wachte Modell in einem Durchlauf (in Abbildung 11-5 beginnen Sie einfach bei Schritt 3) und verwenden dann Autoencoder oder GANs statt RBMs.



*Abbildung 11-5: Beim unüberwachten Vortrainieren wird ein Modell mit den ungelabelten Daten (oder allen Daten) mithilfe einer unüberwachten Lerntechnik trainiert und dann für die eigentliche Aufgabe mit den gelabelten Aufgaben mit einer überwachten Lerntechnik im Detail angepasst – der unüberwachte Teil kann wie hier gezeigt eine Schicht nach der anderen oder direkt das ganze Modell trainieren.*

## Vortrainieren anhand einer Hilfsaufgabe

Haben Sie nicht ausreichend gelabelte Daten, stellen wir Ihnen abschließend noch die Möglichkeit vor, zuerst ein neuronales Netz auf einer Hilfsaufgabe zu trainieren, für das wir gelabelte Trainingsdaten leicht erhalten oder generieren können. Anschließend verwenden wir die unteren Schichten dieses Netzes für die eigentliche Aufgabe. Die unteren Schichten des ersten Netzes lernen, Merkmale zu erkennen, die vom zweiten Netz genutzt werden können.

Beispielsweise stehen Ihnen beim Konstruieren eines Systems zur Gesichtserkennung nur wenige Bilder jeder Einzelperson zur Verfügung – sicher nicht genug, um einen guten Klassifikator zu trainieren. Hunderte Bilder jeder Person zu sammeln, wäre sicher nicht praktikabel. Sie könnten allerdings eine Menge Bilder zufällig ausgewählter Personen im Web sammeln und ein erstes neuronales Netz darauf trainieren, ob zwei Bilder die gleiche Person enthalten. Solch ein Netz würde gute

Merkmale von Gesichtern erlernen, sodass sich mit den unteren Schichten auch mit wenigen Trainingsdaten ein guter Klassifikator trainieren ließe.

Für die *natürliche Sprachverarbeitung* (Natural Language Processing, NLP) können Sie einen Korpus mit Millionen Textdokumenten herunterladen und daraus automatisch gelabelte Daten erzeugen. Sie könnten beispielsweise zufällig einzelne Wörter ausblenden und ein Modell trainieren, das vorhersagt, welches Wort fehlt (so sollte es zum Beispiel vorhersagen, dass das fehlende Wort im Satz »What \_\_\_\_\_ you saying?« wahrscheinlich »are« oder »were« ist). Können Sie ein Modell darin trainieren, bei dieser Aufgabe eine gute Leistung zu erreichen, wird es schon ziemlich viel über Sprache wissen, und Sie können es sicherlich für Ihre eigentliche Aufgabe wiederverwenden und es mit Ihren gelabelten Daten optimieren (mehr zu Vortrainingsaufgaben besprechen wir in Kapitel 15).



*Selbstüberwachtes Lernen* heißt, dass Sie automatisch die Labels aus den Daten selbst erzeugen und dann ein Modell mit dem entstandenen »gelabelten« Datensatz mithilfe überwachter Lerntechniken trainieren. Da für diesen Ansatz kein Labeling durch Menschen erforderlich ist, lässt es sich am besten als Form des unüberwachten Lernens klassifizieren.

## Schnellere Optimierer

Das Trainieren eines sehr großen Deep-Learning-Netzes kann nervtötend langsam sein. Wir haben bisher vier Möglichkeiten betrachtet, um das Trainieren zu beschleunigen (und die Lösung zu verbessern): eine geeignete Initialisierungsstrategie für die Gewichte der Verbindungen zu nutzen, die Aktivierungsfunktion sinnvoll zu wählen, Batchnormalisierung zu verwenden und Teile eines vortrainierten Netzes einzusetzen (eventuell gebaut für eine künstliche Aufgabe oder per unüberwachtes Lernen). Eine weitere deutliche Beschleunigung lässt sich durch Verwenden eines schnelleren Optimierers anstelle des gewöhnlichen Gradientenverfahrens erzielen. In diesem Abschnitt stellen wir die verbreitetsten Verfahren vor: Momentum Optimization, Accelerated Gradient nach Nesterov, AdaGrad, RMSProp und schließlich die Adam-und-Nadam-Optimierung.

### Momentum Optimization

Stellen Sie sich eine Bowlingkugel vor, die eine leicht abschüssige, glatte Oberfläche herunterrollt: Sie beginnt langsam, beschleunigt aber bald, bis sie eine Endgeschwindigkeit erreicht (falls es Reibung und Luftwiderstand gibt). Dies ist die einfache Idee der von Boris Polyak im Jahr 1964 vorgeschlagenen *Momentum Optimization* (<https://homl.info/54>).<sup>13</sup> Im Gegensatz dazu führt das Gradientenver-

<sup>13</sup> Boris T. Polyak, »Some Methods of Speeding Up the Convergence of Iteration Methods«, *USSR Computational Mathematics and Mathematical Physics* 4, no. 5 (1964): 1–17.

fahren auf einer schiefen Ebene einfach viele gleich große Schritte nach unten aus, sodass es insgesamt eine längere Zeit bis zum unteren Ende benötigt.

Wie erwähnt, aktualisiert das Gradientenverfahren die Gewichte  $\theta$ , indem es den Gradienten von der nach den Gewichten ( $\nabla_{\theta}J(\theta)$ ) abgeleiteten Kostenfunktion  $J(\theta)$ , multipliziert mit der Lernrate  $\eta$ , direkt abzieht. Die Gleichung hierfür lautet:  $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta)$ . Sie kümmert sich nicht darum, wie groß frühere Gradienten waren. Wenn der lokale Gradient klein ist, arbeitet das Verfahren sehr langsam.

Momentum Optimization berücksichtigt die früheren Gradienten: Bei jedem Schritt fügt es den lokalen Gradienten zum *Momentvektor*  $m$  hinzu (multipliziert mit der Lernrate  $\eta$ ) und aktualisiert die Gewichte, indem es diesen Momentvektor einfach abzieht (siehe Formel 11-4). Anders ausgedrückt, der Gradient wird als Beschleunigung, nicht als Geschwindigkeit interpretiert. Um Reibung nachzubilden und zu verhindern, dass das Moment zu groß wird, gibt es im Verfahren den zusätzlichen Hyperparameter  $\beta$ , das *Moment*, das zwischen 0 (hohe Reibung) und 1 (keine Reibung) liegen muss. Ein typisches Moment beträgt 0,9.

*Formel 11-4: Momentum-Algorithmus*

1.  $m \leftarrow \beta m - \eta \nabla_{\theta}J(\theta)$
2.  $\theta \leftarrow \theta + m$

Sie können leicht nachweisen, dass die Endgeschwindigkeit (der maximale Betrag der Gewichtsveränderung) bei einem konstanten Gradienten gleich dem Produkt aus Gradient, Lernrate  $\eta$  und  $\frac{1}{1-\beta}$  ist (wir ignorieren dabei das Vorzeichen). Wenn

beispielsweise  $\beta = 0,9$  gilt, beträgt die Endgeschwindigkeit 10 mal Gradient mal Lernrate. Die Momentum Optimization bewegt sich also 10 Mal schneller als das Gradientenverfahren! Dadurch kann die Momentum Optimization schneller als das Gradientenverfahren aus Plateaus entkommen. Wir haben in Kapitel 4 gesehen, dass die Kostenfunktion bei unterschiedlich skalierten Eingaben wie eine breite Schüssel aussieht (siehe Abbildung 4-7). Das Gradientenverfahren steigt die steilen Wände schnell herab, benötigt dann aber eine lange Zeit bis ins Tal. Die Momentum Optimization dagegen rollt immer schneller und schneller das Tal entlang, bis sie den tiefsten Punkt (das Optimum) erreicht. Bei Deep-Learning-Netzen ohne Batchnormalisierung haben die oberen Schichten häufig Eingaben mit sehr unterschiedlichen Wertebereichen. In dieser Situation ist die Momentum Optimization sehr wertvoll. Sie hilft auch dabei, an lokalen Optima vorbeizurollen.



Wegen des Moments kann der Optimierer ein wenig über das Ziel hinausschießen, zurückkehren, sich wieder zu weit entfernen und so mehrmals oszillieren, bevor er sich beim Minimum stabilisiert. Aus diesem Grund ist es gut, ein wenig Reibung im System zu haben: Sie eliminiert diese Oszillation und beschleunigt daher die Konvergenz.

Das Implementieren der Momentum Optimization in Keras ist ein Klacks: Nutzen Sie einfach den SGD-Optimierer und setzen Sie dessen Hyperparameter `momentum`, dann lehnen Sie sich anschließend zurück und genießen!

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Der Hauptnachteil der Momentum Optimization ist, dass wir uns um einen weiteren Hyperparameter kümmern müssen. In der Praxis funktioniert der Wert 0,9 für das Moment meist gut und ist fast immer schneller als das Gradientenverfahren.

## Beschleunigter Gradient nach Nesterov

Eine kleine Variante der Momentum Optimization, die von Yurii Nesterov im Jahr 1983 vorgestellt wurde (<https://hml.info/55>),<sup>14</sup> ist so gut wie immer schneller als die reine Momentum Optimization. Die Idee bei der *Momentum Optimization nach Nesterov* oder dem *beschleunigten Gradienten nach Nesterov* (NAG) ist, den Gradienten der Kostenfunktion nicht an der aktuellen Position  $\theta$  zu bestimmen, sondern ein wenig weiter vorwärts in Richtung des Moments  $\theta + \beta m$  (siehe Formel 11-5).

*Formel 11-5: Algorithmus des beschleunigten Gradienten nach Nesterov*

1.  $m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta + \beta m)$
2.  $\theta \leftarrow \theta + m$

Diese kleine Modifikation funktioniert, weil der Momentvektor im Allgemeinen in die richtige Richtung zeigt (also zum Optimum hin). Daher ist das Verwenden des Gradienten ein Stück weiter in dieser Richtung etwas genauer als an der ursprünglichen Position, wie Sie Abbildung 11-6 entnehmen können (wobei  $\nabla_1$  für den Gradienten der am Ausgangspunkt  $\theta$  bestimmten Kostenfunktion steht und  $\nabla_2$  für den Gradienten am Punkt  $\theta + \beta m$ ).

Wie Sie sehen, führt die Änderung nach Nesterov etwas näher ans Optimum heran. Nach einer Weile addieren sich diese kleinen Verbesserungen auf, wodurch NAG deutlich schneller als die gewöhnliche Momentum Optimization ist. Wenn zudem das Moment die Gewichte durch ein Tal zieht, drückt  $\nabla_1$  sie weiter durch das Tal voran, während  $\nabla_2$  sie in Richtung der Talsohle schiebt. Dadurch wird Oszillation unterbunden, und die Konvergenz wird beschleunigt.

NAG beschleunigt das Trainieren im Allgemeinen besser als die gewöhnliche Momentum Optimization. Um sie zu verwenden, setzen Sie einfach beim Erstellen eines SGD-Optimierers den Parameter `nesterov=True`:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

---

<sup>14</sup> Yurii Nesterov, »A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence  $O(1/k^2)$ «, *Doklady AN USSR* 269 (1983): 543–547.

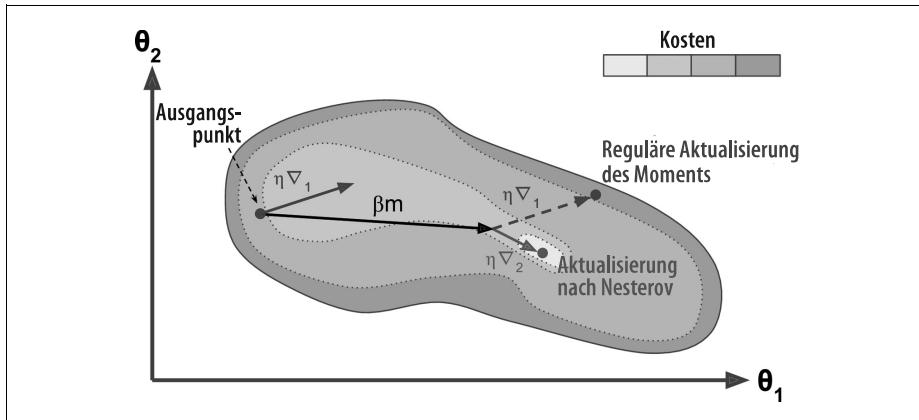


Abbildung 11-6: Gewöhnliche Momentum Optimization im Vergleich zur Variante nach Nesterov: Die normale wendet die berechneten Gradienten vor dem Momentum-Schritt an, die Variante erst danach.

## AdaGrad

Betrachten wir noch einmal das Problem der länglichen Schüssel: Das Gradientenverfahren bewegt sich schnell entlang der steilsten Wand, zeigt dabei aber nicht direkt auf das globale Optimum und geht dann langsam bis zur Talsohle weiter. Es wäre schön, wenn der Algorithmus dies früh erkennen könnte, um sich ein wenig mehr in Richtung des globalen Optimums auszurichten. Der AdaGrad-Algorithmus (<https://homl.info/56>)<sup>15</sup> arbeitet genau so, er skaliert den Gradientenvektor entlang der steilsten Dimensionen herunter (siehe Formel 11-6):

Formel 11-6: AdaGrad-Algorithmus

1.  $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$

Der erste Schritt akkumuliert das Quadrat der Gradienten in den Vektor  $\mathbf{s}$  (das Symbol  $\otimes$  steht für die Multiplikation der einzelnen Elemente). Diese vektorisierte Form entspricht der Berechnung von  $s_i \leftarrow s_i + (\partial J(\theta) / \partial \theta_i)^2$  für jedes Element  $s_i$  des Vektors  $\mathbf{s}$ ; anders ausgedrückt, akkumuliert jedes  $s_i$  die Quadrate der partiellen Ableitung der Kostenfunktion nach dem Parameter  $\theta_i$ . Ist die Kostenfunktion entlang der  $i$ -Dimension steil, wird  $s_i$  von Iteration zu Iteration immer größer.

Der zweite Schritt ist beinahe identisch mit dem Gradientenverfahren, es gibt aber einen großen Unterschied: Der Gradientenvektor wird um den Faktor  $\sqrt{\mathbf{s} + \varepsilon}$  herunterskaliert (das Symbol  $\oslash$  steht für die elementweise Division, und  $\varepsilon$  ist ein Glättungsterm, um Divisionen durch null zu vermeiden, und beträgt normalerweise

<sup>15</sup> John Duchi et al., »Adaptive Subgradient Methods for Online Learning and Stochastic Optimization«, *Journal of Machine Learning Research* 12 (2011): 2121–2159.

$10^{-10}$ ). Diese Vektorschreibweise entspricht der (gleichzeitigen) Berechnung von  $\theta_i \leftarrow \theta_i - \eta \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} / \sqrt{s_i + \epsilon}$  für alle Parameter  $\theta_i$ .

Kurz, dieser Algorithmus baut die Lernrate nach und nach ab, bei steilen Dimensionen erfolgt der Abbau aber schneller als bei Dimensionen mit geringeren Steigungen. Dies bezeichnet man auch als *adaptive Lernrate*. Die daraus entstehenden Aktualisierungen weisen eher in Richtung des globalen Optimums (siehe Abbildung 11-7). Ein zusätzlicher Vorteil ist, dass wesentlich weniger Feinabstimmung der Lernrate als Hyperparameter  $\eta$  nötig ist.

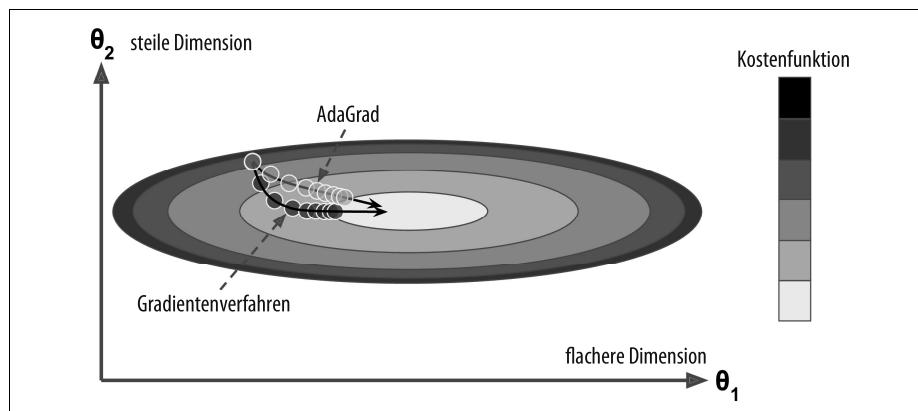


Abbildung 11-7: AdaGrad im Vergleich zum Gradientenverfahren – Ersterer kann frühzeitig auf das Optimum zeigen.

AdaGrad schneidet bei einfachen quadratischen Problemen oft gut ab, hält aber beim Trainieren neuronaler Netze häufig zu früh an. Die Lernrate wird dann so weit herunterskaliert, dass der Algorithmus komplett anhält, bevor er das globale Optimum erreicht. Obwohl es in Keras einen Adagrad-Optimierer gibt, sollten Sie diesen nicht zum Trainieren von Deep-Learning-Netzen verwenden (bei einfachen Aufgaben wie der linearen Regression kann er sich aber als effizient erweisen). Es ist aber trotzdem hilfreich, AdaGrad zu verstehen, um die anderen Optimierer zum Anpassen der Lernrate zu begreifen.

## RMSProp

Wie wir gesehen haben, besteht bei AdaGrad das Risiko, dass er ein wenig zu schnell abbremst und daher in manchen Fällen das globale Optimum nie erreicht. Der *RMSProp*-Algorithmus<sup>16</sup> behebt dieses Problem, indem er nur die Gradienten aus den letzten Iterationen akkumuliert (anstatt sämtliche Gradienten seit Trai-

<sup>16</sup> Dieser Algorithmus wurde von Geoffrey Hinton und Tijmen Tieleman im Jahr 2012 entwickelt und von Geoffrey Hinton in seinem Coursera-Kurs zu neuronalen Netzen präsentiert (Folien: <https://homl.info/57>, Video: <https://homl.info/58>). Da die Autoren keinen Fachartikel über das Verfahren verfasst haben, zitieren Forscher in ihren Artikeln häufig »Folie 29 in Vorlesung 6«.

ningsbeginn zu verwenden). Dazu wird im ersten Schritt ein exponentieller Zerfall verwendet (siehe Formel 11-7).

*Formel 11-7: RMSProp-Algorithmus*

1.  $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$

Die Zerfallsrate  $\beta$  wird normalerweise auf 0,9 gesetzt. Ja, es ist schon wieder ein neuer Hyperparameter, aber dieser Standardwert funktioniert meist so gut, dass Sie ihn nicht weiter verändern müssen.

Wie Sie sich vielleicht schon denken, gibt es in Keras den Optimizer RMSprop:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Außer bei sehr einfachen Aufgaben ist dieser Optimizer AdaGrad so gut wie immer überlegen. Deshalb war dies der von vielen Forschern bevorzugte Algorithmus, bis die Adam-Optimierung auf den Plan kam.

## Adam-Optimierung

Adam (<https://homl.info/59>)<sup>17</sup>, eine Abkürzung für *Adaptive Moment Estimation*, kombiniert die Idee der Momentum Optimization und RMSProp: Wie die Momentum Optimization merkt sich das Verfahren den Durchschnitt der vorherigen Gradienten, und wie RMSProp merkt es sich auch den Durchschnitt der quadrierten Gradienten, beide unter exponentiellem Zerfall (siehe Formel 11-8).<sup>18</sup>

*Formel 11-8: Adam-Algorithmus*

1.  $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2.  $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3.  $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4.  $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5.  $\theta \leftarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$

In dieser Gleichung steht  $t$  für den Index der Iteration (beginnend bei 1).

Wenn Sie sich nur die Schritte 1, 2 und 5 ansehen, bemerken Sie die Nähe der Adam-Optimierung zu Momentum Optimization und RMSProp. Der einzige

---

<sup>17</sup> Diederik P. Kingma und Jimmy Ba, »Adam: A Method for Stochastic Optimization«, arXiv preprint arXiv:1412.6980 (2014).

<sup>18</sup> Dies sind Schätzungen des Mittelwerts und der (nicht zentrierten) Varianz der Gradienten. Der Mittelwert wird oft als *erstes Moment*, die Varianz als *zweites Moment* bezeichnet. Daher röhrt der Name des Algorithmus.

Unterschied ist, dass bei Schritt 1 ein exponentiell abfallender Durchschnitt anstatt einer exponentiell abfallenden Summe berechnet wird. Diese sind aber bis auf einen konstanten Faktor gleich (der abfallende Durchschnitt beträgt  $1 - \beta_1$  mal die abfallende Summe). Die Schritte 3 und 4 enthalten ein technisches Detail: Weil  $\mathbf{m}$  und  $\mathbf{s}$  mit 0 initialisiert wurden, enthalten sie zu Beginn des Trainings ein Bias in Richtung 0. Deshalb werten diese beiden Schritte  $\mathbf{m}$  und  $\mathbf{s}$  zu Beginn des Trainings auf.

Der Hyperparameter für den Abfall des Moments  $\beta_1$  wird normalerweise mit 0,9 initialisiert, der Hyperparameter für den Abfall der Skalierung  $\beta_2$  wird häufig mit 0,999 initialisiert. Wie weiter oben wird der Glättungsterm  $\epsilon$  normalerweise mit einer sehr kleinen Zahl wie  $10^{-7}$  initialisiert. Dies sind die voreingestellten Werte der Klasse Adam (um genau zu sein, steht epsilon standardmäßig auf None, womit Keras keras.backend.epsilon() verwendet, was wiederum standardmäßig  $10^{-7}$  liefert – Sie können es mit keras.backend.set\_epsilon() ändern). So erstellen Sie einen Adam-Optimierer mit Keras:

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Da Adam ein Algorithmus mit adaptiver Lernrate ist (wie auch AdaGrad und RMSProp), muss weniger Feineinstellung der Lernrate  $\eta$  erfolgen. Sie können meist den Standardwert  $\eta = 0,001$  verwenden, wodurch Adam sogar einfacher als das Gradientenverfahren zu nutzen ist.



Sollten Sie sich von all diesen verschiedenen Techniken mittlerweile etwas überfordert fühlen und sich fragen, wie Sie die richtige für Ihre Aufgabe auswählen, machen Sie sich keine Sorgen: Am Ende dieses Kapitels liefere ich Ihnen ein paar praktische Ratschläge.

Es lohnt sich noch, auf zwei Varianten von Adam hinzuweisen:

#### *AdaMax*

In Schritt 2 von Formel 11-8 summiert Adam die Quadrate der Gradienten in  $\mathbf{s}$  auf (mit einer größeren Gewichtung auf die aktuelleren Gradienten). Ignorieren wir in Schritt 5  $\epsilon$  sowie die Schritte 3 und 4 (bei denen es sich nur um technische Details handelt), skaliert Adam die Parameteraktualisierungen mit der Quadratwurzel von  $\mathbf{s}$ . Kurz gesagt, skaliert Adam die Parameteraktualisierungen mithilfe der  $\ell_2$ -Norm der altersberücksichtigten Gradienten (die  $\ell_2$ -Norm ist die Quadratwurzel der Summe der Quadrate). AdaMax – im gleichen Artikel vorgestellt wie Adam – ersetzt die  $\ell_2$ -Norm durch die  $\ell_\infty$ -Norm (eine schwierigere Ausdrucksweise für das Maximum). Genauer gesagt, ersetzt es Schritt 2 in Formel 11-8 durch  $\mathbf{s} \leftarrow \max(\beta_2 \mathbf{s}, \nabla_{\theta} J(\theta))$ , lässt Schritt 4 weg und skaliert in Schritt 5 die Gradientenaktualisierungen um einen Faktor  $\mathbf{s}$  herunter, was gerade das Maximum der altersberücksichtigten Gradienten ist. In der Praxis ist AdaMax dadurch möglicherweise stabiler als Adam, aber das hängt vom Datensatz ab, und im Allgemeinen ist Adam schneller. Somit han-

delt es sich bei AdaMax letztlich um einen weiteren Optimierer, den Sie ausprobieren können, wenn Sie bei einer Aufgabe Probleme mit Adam haben.

### Nadam

Beim Nadam-Optimierer handelt es sich um den Adam-Optimierer plus Nesterov-Trick, wodurch er oft etwas schneller als Adam konvergiert. In dem Artikel (<https://homl.info/nadam>), der diese Technik vorstellt,<sup>19</sup> vergleicht der Forscher Timothy Dozat viele verschiedene Optimierer für diverse Aufgaben und stellt fest, dass Nadam im Allgemeinen schneller als Adam ist, manchmal aber von RMSProp noch übertrumpft wird.



Adaptive Optimierungsmethoden (einschließlich RMSProp, Adam und Nadam) sind oft wunderbar und konvergieren schnell hin zu einer guten Lösung. Aber ein Artikel aus dem Jahr 2017 (<https://homl.info/60>) von Ashia C. Wilson et al.<sup>20</sup> kam zu dem Schluss, dass sie bei manchen Datensätzen zu einer schwachen Verallgemeinerungsleistung führen. Sind Sie also mit der Leistung Ihres Modells unzufrieden, können Sie den klassischen beschleunigten Gradienten nach Nesterov ausprobieren – Ihr Datensatz reagiert eventuell nur allergisch auf adaptive Gradienten. Schauen Sie sich auch die aktuelle Forschung an, denn da passiert ziemlich viel.

Alle bisher besprochenen Optimierungstechniken beruhen lediglich auf den *partiellen Ableitungen erster Ordnung (Jacobians)*. Die Literatur zu Optimierungsverfahren enthält faszinierende Algorithmen, die mit den *partiellen Ableitungen zweiter Ordnung* arbeiten (die *Hessians*, bei denen es sich um die partiellen Ableitungen der Jacobians handelt). Leider lassen sich diese Algorithmen sehr schwer auf neuronale Netze anwenden, weil es pro Ausgabe  $n^2$  Hessians gibt (wobei  $n$  die Anzahl der Parameter ist). Im Gegensatz dazu gibt es nur  $n$  Jacobians pro Ausgabe. Da DNNs normalerweise Zehntausende Parameter enthalten, passen die Algorithmen 2. Grades oft nicht einmal in den Speicher. Selbst wenn sie passen, ist die Berechnung der Hessians einfach zu langsam.

### Trainieren spärlicher Modelle

Alle eben vorgestellten Optimierungsalgorithmen erzeugen dichte Modelle, solche, bei denen die meisten Parameter ungleich null sind. Wenn Sie ein zur Laufzeit blitzschnelles Modell benötigen oder wenn es wenig Speicher verbrauchen soll, sollten Sie ein spärliches Modell in Betracht ziehen.

<sup>19</sup> Timothy Dozat, »Incorporating Nesterov Momentum into Adam« (2016).

<sup>20</sup> Ashia C. Wilson et al., »The Marginal Value of Adaptive Gradient Methods in Machine Learning«, *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.

Eine einfache Möglichkeit hierzu ist, das Modell wie gewohnt zu trainieren und anschließend winzige Gewichte auf null zu setzen. Allerdings wird dies meist nicht zu einem sehr spärlichen Modell führen und dessen Leistung verschlechtern.

Eine andere Möglichkeit besteht darin, während des Trainierens eine starke  $\ell_1$ -Regularisierung anzuwenden (darauf kommen wir später noch zurück), da diese den Optimierer veranlasst, möglichst viele Gewichte auf null zu setzen (wie im Fall der in Kapitel 4 besprochenen Lasso-Regression).

Erweisen sich diese Techniken als unzureichend, werfen Sie einen Blick auf das TensorFlow Model Optimization Toolkit (TF-MOT) (<https://homl.info/tfmot>), das eine aufgeräumte API hat, mit der Sie während des Trainings iterativ Verbindungen abhängig von deren Magnitude entfernen können.

In Tabelle 11-2 werden alle bisher besprochenen Optimierer verglichen (\* ist schlecht, \*\* ist durchschnittlich, und \*\*\* ist gut).

*Tabelle 11-2: Vergleich von Optimierern*

Klasse	Konvergenzgeschwindigkeit	Konvergenzqualität
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	* (stoppt zu früh)
Adagrad	***	
RMSprop	***	** oder ***
Adam	***	** oder ***
Nadam	***	** oder ***
AdaMax	***	** oder ***

## Scheduling der Lernrate

Es ist sehr wichtig, eine gute Lernrate zu finden. Wenn Sie sie zu hoch einstellen, divergiert das Modell beim Trainieren (wie in Kapitel 4 besprochen). Wenn Sie sie zu niedrig einstellen, konvergiert das Modell irgendwann zum Optimum, das Trainieren wird aber sehr lange dauern. Wenn Sie sie ein klein wenig zu hoch einstellen, macht das Modell zunächst sehr gute Fortschritte, tanzt dann aber um das Optimum herum und kommt nie zur Ruhe. Ist Ihre Rechenzeit begrenzt, müssen Sie das Trainieren unterbrechen, bevor es anständig konvergiert, und sich mit einer suboptimalen Lösung zufriedengeben (siehe Abbildung 11-8).

Wie in Kapitel 10 besprochen, finden Sie eventuell eine gute Lernrate, indem Sie das Modell für ein paar Hundert Iterationen trainieren, dabei die Lernrate exponentiell von einem sehr kleinen zu einem sehr großen Wert wachsen lassen, sich dann die Lernkurve anschauen und dann eine Lernrate auswählen, die etwas unter

dem Wert liegt, bei dem die Lernkurve wieder hochschießt. Dann können Sie Ihr Modell neu initialisieren und es mit dieser Lernrate trainieren.

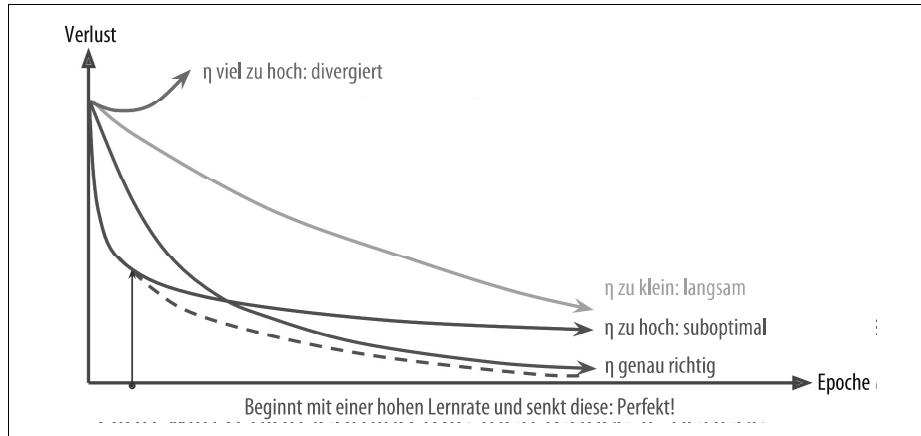


Abbildung 11-8: Lernkurven für unterschiedliche Lernraten  $\eta$

Sie können aber auch etwas Besseres als eine konstante Lernrate erzielen: Wenn Sie mit einer hohen Lernrate beginnen und diese reduzieren, sobald sie keine großen Fortschritte mehr macht, lässt sich eine gute Lösung schneller als mit einer konstanten Lernrate finden. Es gibt viele unterschiedliche Strategien, mit denen sich die Lernrate beim Trainieren verringern lässt. Lohnenswert kann es auch sein, mit einer niedrigen Lernrate zu beginnen, sie zu steigern und dann wieder abfallen zu lassen. Diese Strategien bezeichnet man als *Scheduling der Lernrate* (wir haben den Begriff kurz in Kapitel 4 erwähnt). Die häufigsten Strategien sind:

#### *Power Scheduling*

Setzen Sie die Lernrate auf eine Funktion der Iteration  $t$ :  $\eta(t) = \eta_0 / (1 + t/s)^c$ . Die initiale Lernrate  $\eta_0$ , die Potenz  $c$  (meist mit dem Wert 1) und die Schritte  $s$  sind Hyperparameter. Nach  $s$  Schritten sinkt die Lernrate auf  $\eta_0 / 2$ . Nach  $s$  weiteren Schritten liegt sie bei  $\eta_0 / 3$ , dann sinkt sie auf  $\eta_0 / 4$  und so weiter. Wie Sie sehen, sinkt dieser Schedule erst schnell und dann immer langsamer. Natürlich müssen Sie für das Power Scheduling sowohl  $\eta_0$  wie auch  $s$  (und eventuell  $c$ ) anpassen.

#### *Exponentielles Scheduling*

Legen Sie die Lernrate als Funktion der Iteration  $t$  fest:  $\eta(t) = \eta_0 \cdot 0,1^{t/s}$ . Die Lernrate wird damit nach und nach um einen Faktor von 10 alle  $s$  Schritte fallen. Während das Power Scheduling die Lernrate stärker und langsamer verringert, wird sie beim exponentiellen Scheduling weiterhin alle  $s$  Schritte um den Faktor 10 reduziert.

#### *Vorgefertigte stückweise konstante Lernrate*

Setzen Sie die Lernrate zu Beginn für eine gewisse Anzahl von Epochen auf einen festen Wert (zum Beispiel  $\eta_0 = 0,1$  für 5 Epochen), dann für eine weitere

Anzahl von Epochen auf einen kleineren Wert (zum Beispiel  $\eta_1 = 0,001$  für 50 Epochen) und so weiter. Obwohl diese Lösung sehr gut funktionieren kann, muss man meist ein wenig experimentieren, bis man die richtigen Zeitpunkte und Lernraten gefunden hat.

#### *Performance Scheduling*

Messen Sie alle  $N$  Schritte den Validierungsfehler (wie beim Early Stopping) und verringern Sie die Lernrate um den Faktor  $\lambda$ , sobald der Fehler nicht mehr abfällt.

#### *1cycle Scheduling*

Anders als bei den anderen Ansätzen beginnt das *1cycle Scheduling* (vorgestellt in einem Artikel (<https://homl.info/1cycle>) von Leslie Smith aus dem Jahr 2018<sup>21</sup>) damit, die initiale Lernrate  $\eta_0$  zu erhöhen und bis zur Hälfte des Trainings linear bis zur Lernrate  $\eta_1$  wachsen zu lassen. Dann reduziert es während der zweiten Hälfte des Trainings die Rate wieder linear bis  $\eta_0$ , um sie ganz zum Schluss während der letzten paar Epochen um mehrere Größenordnungen abzusenken (immer noch linear). Die maximale Lernrate  $\eta_1$  wird mit dem gleichen Ansatz ermittelt wie beim Finden der optimalen Lernrate, und die initiale Lernrate  $\eta_0$  wird ungefähr um einen Faktor 10 kleiner ausgewählt. Beim Einsatz eines Momentums beginnen wir zuerst mit einem hohen Momentum (zum Beispiel 0,95), reduzieren es dann in der ersten Hälfte des Trainings (zum Beispiel linear auf 0,85) und bringen es dann wieder in der zweiten Hälfte zurück auf den maximalen Wert, wobei dieser bei den letzten paar Epochen beibehalten wird. Smith hat viele Experimente durchgeführt, die gezeigt haben, dass dieser Ansatz oft dazu in der Lage war, das Training deutlich zu beschleunigen und eine bessere Leistung zu erreichen. So gelangte er beispielsweise mit dem beliebten CIFAR10-Bilddatensatz auf 91,9% Validierungsgenauigkeit in nur 100 Epochen – im Gegensatz zu 90,3% Genauigkeit nach 800 Epochen bei einem Standardvorgehen (mit der gleichen Architektur des neuronalen Netzes).

Ein Artikel (<https://homl.info/63>) aus dem Jahr 2013<sup>22</sup> von Andrew Senior et al. verglich die Leistung der beliebtesten Scheduling-Verfahren zum Trainieren von Deep-Learning-Netzen zur Sprachverarbeitung mit Momentum Optimization. Die Autoren kamen zu dem Schluss, dass in diesem Szenario sowohl Performance Scheduling als auch exponentielles Scheduling gut abschneiden. Sie bevorzugten jedoch das exponentielle Scheduling, weil es sich einfacher abstimmen lässt und ein wenig schneller zur optimalen Lösung gelangt (es wird auch erwähnt, dass es sich leichter

---

21 Leslie N. Smith, »A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay«, arXiv preprint arXiv:1803.09820 (2018).

22 Andrew Senior et al., »An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition«, *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2013): 6724–6728.

als Performance Scheduling implementieren ließ, aber in Keras sind beide Varianten einfach). Allerdings scheint der 1cycle-Ansatz noch schneller zu sein.

Ein Power Scheduling lässt sich in Keras sehr einfach implementieren: Setzen Sie beim Erstellen eines Optimierers den Hyperparameter decay:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

Der decay ist das Inverse von s (die Anzahl an Schritten, nach denen die Lernrate durch eine weitere Einheit dividiert wurde). Keras geht davon aus, dass  $c = 1$  ist.

Exponentielles Scheduling und eine stückweise konstante Lernrate lassen sich ebenfalls einfach umsetzen. Sie müssen zunächst eine Funktion definieren, die die aktuelle Epoche erwartet und die Lernrate zurückgibt. Implementieren wir beispielsweise das exponentielle Scheduling:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

Wollen Sie  $\eta_0$  und s nicht fest angeben, können Sie auch eine Funktion schreiben, die eine konfigurierte Funktion zurückgibt:

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Als Nächstes erstellen Sie einen Callback LearningRateScheduler, übergeben ihm die Schedule-Funktion und reichen diesen Callback an die Methode fit() weiter:

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

Der LearningRateScheduler wird das Attribut learning\_rate des Optimierers zu Beginn jeder Epoche anpassen. Das reicht normalerweise aus, aber wenn die Lernrate häufiger angepasst werden soll – zum Beispiel bei jedem Schritt –, können Sie Ihren eigenen Callback schreiben (siehe den Abschnitt »Exponential Scheduling« des Notebooks). Das Aktualisieren der Lernrate bei jedem Schritt ist sinnvoll, wenn es viele Schritte pro Epoche gibt. Alternativ können Sie den gleich beschriebenen Ansatz keras.optimizers.schedules verfolgen.

Die Schedule-Funktion kann optional die aktuelle Lernrate als zweites Argument übernehmen. So multipliziert beispielsweise die folgende Schedule-Funktion die vorherige Lernrate mit  $0,1^{1/20}$ , was zum gleichen exponentiellen Abfall führt (der allerdings mit Epoche 0 statt mit Epoche 1 beginnt):

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

Diese Implementierung geht von der initialen Lernrate des Optimierers aus (im Gegensatz zur vorherigen Implementierung) – stellen Sie daher sicher, dass Sie sie korrekt setzen.

Sichern Sie ein Modell, werden auch der Optimierer und seine Lernrate mit gesichert. Mit dieser neuen Schedule-Funktion können Sie daher einfach ein trainiertes Modell laden und ohne Probleme mit dem Training dort fortfahren, wo Sie aufgehört haben. Nutzt Ihre Schedule-Funktion allerdings das Argument epoch, wird es komplizierter: Die Epoche wird nicht mitgesichert und bei jedem Aufruf der Methode fit() auf 0 zurückgesetzt. Wollen Sie bei einem Modell dort mit dem Training fortfahren, wo Sie aufgehört haben, kann das zu einer sehr großen Lernrate führen, was vermutlich die Gewichte Ihres Modells beschädigt. Eine Lösung ist, das Argument initial\_epoch der Methode fit() manuell zu setzen, sodass epoch direkt mit dem richtigen Wert loslegt.

Für eine stückweise konstante Lernrate können Sie eine Schedule-Funktion wie die folgende nutzen (wie zuvor können Sie bei Bedarf eine generischere Funktion definieren, siehe den Abschnitt »Piecewise Constant Scheduling« des Notebooks), dann einen Callback LearningRateScheduler mit dieser Funktion erstellen und ihn an die Methode fit() übergeben – so wie wir es beim exponentiellen Scheduling gemacht haben:

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

Für ein Performance Scheduling verwenden Sie den Callback ReduceLROnPlateau. Übergeben Sie beispielsweise den folgenden Callback an die Methode fit(), wird dieser die Lernrate immer dann mit 0,5 multiplizieren, wenn sich der beste Validierungsverlust fünf aufeinanderfolgende Epochen lang nicht verbessert (es stehen noch andere Optionen zur Verfügung – werfen Sie einen Blick in die Dokumentation):

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

Und schließlich bietet tf.keras noch eine alternative Möglichkeit an, ein Lernraten-Scheduling zu implementieren: Definieren Sie die Lernrate über einen der in keras.optimizers.schedules verfügbaren Scheduler und übergeben Sie sie dann an einen Optimierer. Dadurch wird die Lernrate bei jedem Schritt statt nur bei jeder Epoche aktualisiert. So implementieren Sie zum Beispiel den gleichen exponentiellen Schedule wie bei der weiter oben definierten Funktion exponential\_decay\_fn():

```
s = 20 * len(X_train) // 32 # Anzahl Schritte in 20 Epochen (Batchgröße 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

Das ist nett und einfach, und wenn Sie das Modell sichern, werden auch die Lernrate und deren Schedule (einschließlich dessen Status) mit abgespeichert. Dieses Vorgehen ist aber nicht Teil der Keras-API, sondern nur in tf.keras vorhanden.

Bei 1cycle ist die Implementierung auch nicht besonders schwierig: Erstellen Sie einfach einen eigenen Callback, der die Lernrate bei jeder Iteration anpasst (Sie können dafür `self.model.optimizer.lr` ändern). Im Abschnitt »1Cycle Scheduling« des Notebooks finden Sie ein Beispiel dazu.

Zusammengefasst, lässt sich sagen, dass exponentieller Abfall, Performance Scheduling und 1cycle die Konvergenz deutlich beschleunigen können. Probieren Sie es daher einmal aus!

## Vermeiden von Overfitting durch Regularisierung

Mit vier Parametern kann ich einen Elefanten fitten, und mit fünf wackelt er mit dem Rüssel.

—John von Neumann, zitiert von Enrico Fermi in *Nature* 427

Mit Tausenden Parametern können Sie den ganzen Zoo fitten. Deep-Learning-Netze haben meist Zehntausende Parameter, manchmal sogar Millionen. Mit derart vielen Parametern hat das Netz enorm viele Freiheitsgrade und kann eine riesige Bandbreite komplexer Datensätze erfassen. Aber diese Flexibilität bedeutet auch, dass es anfällig für das Overfitten der Trainingsdaten ist.

Wir haben schon eine der besten Regularisierungstechniken in Kapitel 10 implementiert – das Early Stopping. Und auch wenn die Batchnormalisierung dazu gedacht war, das Problem der instabilen Gradienten zu lösen, verhält sie sich zugleich wie ein ziemlich guter Regularisierer. In diesem Abschnitt werden wir einige der beliebtesten Regularisierungstechniken für neuronale Netze vorstellen:  $\ell_1$ - und  $\ell_2$ -Regularisierung, Drop-out und Max-Norm-Regularisierung.

### $\ell_1$ - und $\ell_2$ -Regularisierung

Wie bei den einfachen linearen Modellen in Kapitel 4 können Sie mit der  $\ell_2$ -Regularisierung den Gewichten der Verbindungen eines neuronalen Netzes (aber nicht den Bias-Termen) Beschränkungen auferlegen und/oder die  $\ell_1$ -Regularisierung verwenden, wenn Sie ein Sparse-Modell nutzen wollen (mit vielen Gewichten gleich 0). So wenden Sie die  $\ell_2$ -Regularisierung auf die Verbindungsgewichte einer Keras-Schicht mit einem Regularisierungsfaktor von 0,01 an:

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

Die Funktion `l2()` gibt einen Regularisierer zurück, der während des Trainings bei jedem Schritt aufgerufen wird, um den Regularisierungsverlust zu berechnen. Dieser wird dann dem Gesamtverlust zugeschlagen. Nicht ganz unerwartet können Sie mit `keras.regularizers.l1()` eine  $\ell_1$ -Regularisierung umsetzen – wollen Sie sowohl  $\ell_1$ - wie auch  $\ell_2$ -Regularisierung haben, verwenden Sie `keras.regularizers.l1_l2()` (und geben beide Regularisierungsfaktoren an).

Da Sie normalerweise den gleichen Regularisierer auf alle Schichten Ihres Netzes anwenden wollen – so wie Sie die gleiche Aktivierungsfunktion und Initialisierungsstrategie für alle verborgenen Schichten nutzen –, werden Sie feststellen, dass Sie immer wieder die gleichen Argumente nutzen. Das macht den Code hässlich und fehleranfällig. Um das zu vermeiden, können Sie versuchen, Ihren Code so zu refaktorieren, dass Schleifen zum Einsatz kommen. Eine andere Option ist die Python-Funktion `functools.partial()`, mit der Sie einen dünnen Wrapper für alles Aufrufbare mit Standardargumenten erzeugen können:

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

## Drop-out

Eine der bei Deep-Learning-Netzen beliebtesten Regularisierungstechniken ist *Drop-out*. Sie wurde von Geoffrey Hinton im Jahr 2012 vorgeschlagen (<https://homl.info/64>)<sup>23</sup> und in einem Artikel (<https://homl.info/65>)<sup>24</sup> von Nitish Srivastava et al. genauer ausgeführt. Sie hat sich als höchst erfolgreich erwiesen: Selbst die am weitesten entwickelten neuronalen Netze erfuhren durch das Hinzufügen von Drop-out einen ein- bis zweiprozentigen Zugewinn an Genauigkeit. Dies klingt nicht nach besonders viel, aber wenn ein Modell bereits eine Genauigkeit von 95 % erzielt, bedeuten 2 % Genauigkeit, dass Sie die Fehlerquote um beinahe 40 % senken müssen (von 5 % Fehlern auf etwa 3 %).

Der Algorithmus ist recht einfach: Bei jedem Trainingsschritt wird jedes Neuron (auch die Eingabeneuronen, nicht aber die Ausgabeneuronen) mit einer Wahrscheinlichkeit  $p$  zwischenzeitlich »weggelassen«. Es wird also während dieses Trainingsschritts vollständig ignoriert, kann aber im nächsten Schritt wieder aktiv sein (siehe Abbildung 11-9). Den Hyperparameter  $p$  nennt man die *Drop-out-Rate*, und er wird normalerweise auf 10 % bis 50 % gesetzt: näher an 20 bis 30 % in rekurrenten neuronalen Netzen (siehe Kapitel 15), näher an 40 bis 50 % in Convolutional

---

<sup>23</sup> Geoffrey E. Hinton et al., »Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors«, arXiv preprint arXiv:1207.0580 (2012).

<sup>24</sup> Nitish Srivastava et al., »Dropout: A Simple Way to Prevent Neural Networks from Overfitting«, *Journal of Machine Learning Research* 15 (2014): 1929–1958.

Neural Networks (siehe Kapitel 14). Nach dem Trainieren werden keine Neuronen mehr ausgelassen. Und das ist auch schon alles (bis auf einige technische Details, denen wir uns gleich widmen).

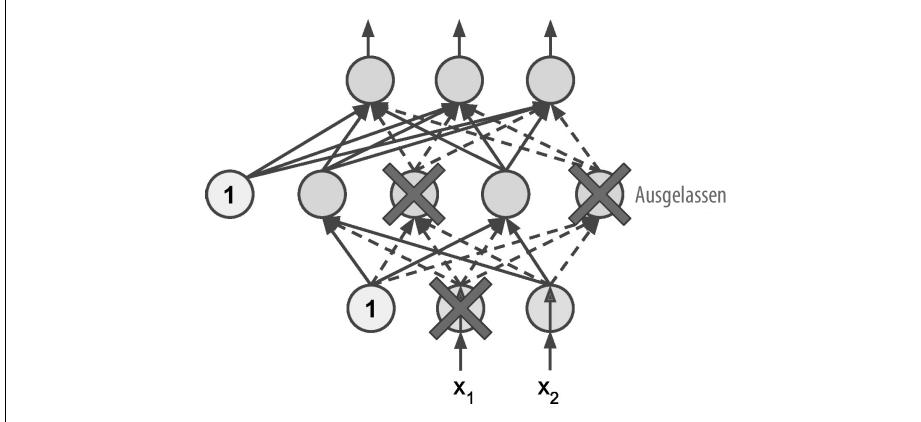


Abbildung 11-9: Bei der Drop-out-Regularisierung wird bei jeder Trainingsiteration eine zufällige Teilmenge aller Neuronen in einer oder mehreren Schichten – außer der Ausgabeschicht – »gedroppt«; diese Neuronen geben in dieser Iteration 0 zurück (dargestellt durch die gestrichelten Pfeile).

Es ist auf den ersten Blick ein wenig überraschend, dass diese destruktive Technik überhaupt funktioniert. Würde ein Unternehmen besser funktionieren, wenn dessen Mitarbeiter jeden Morgen eine Münze werfen, um zu entscheiden, ob sie zur Arbeit gehen? Tja, wer weiß; vielleicht würde es das sogar! Das Unternehmen wäre auf jeden Fall gezwungen, seine Organisation anzupassen; es könnte sich nicht auf eine Einzelperson verlassen, um die Kaffeemaschine zu befüllen oder ähnlich wichtige Aufgaben auszuführen. Die Mitarbeiter müssten lernen, mit vielen ihrer Kollegen zu kooperieren, nicht nur mit einer Handvoll. Das Unternehmen würde dadurch deutlich widerstandsfähiger werden. Falls eine Person kündigt, würde es keinen großen Unterschied machen. Es ist nicht klar, ob diese Idee bei Unternehmen funktionieren würde, aber bei neuronalen Netzen funktioniert sie mit Sicherheit. Mit Drop-out trainierte Neuronen können sich nicht mit ihren Nachbarneuronen co-adaptieren; sie müssen für sich allein so nützlich wie möglich sein. Sie können sich auch nicht exzessiv auf einige Eingabeneuronen verlassen; sie müssen auf jedes ihrer Eingabeneuronen achten. Dadurch sind sie weniger anfällig für kleine Änderungen der Eingabe. Am Ende erhalten Sie ein robusteres Netz, das besser verallgemeinert.

Eine andere Betrachtungsweise, die die Mächtigkeit der Drop-out-Technik illustriert, ist, dass bei jedem Trainingsschritt ein einzigartiges neuronales Netz generiert wird. Da jedes Neuron entweder anwesend oder abwesend sein kann, gibt es insgesamt  $2^N$  mögliche Netze (wobei  $N$  die Gesamtzahl der abschaltbaren Neuronen ist). Aufgrund einer derart großen Zahl ist es praktisch unmöglich, dass das gleiche neuronale Netz doppelt ausgewürfelt wird. Nach 10.000 durchgeföhrten Trai-

ningsschritten haben Sie 10.000 unterschiedliche neuronale Netze trainiert (mit jeweils einem Trainingsdatenpunkt). Natürlich sind diese neuronalen Netze nicht voneinander unabhängig, da sie viele Gewichte untereinander teilen, aber sie sind dennoch alle unterschiedlich. Das am Ende erhaltene neuronale Netz lässt sich als Ensemble aus all diesen kleineren Netzen ansehen.



In der Praxis können Sie Drop-out normalerweise nur in den obersten einen bis drei Schichten anwenden (abgesehen von der Ausgabeschicht).

Es gibt ein kleines, aber wichtiges technisches Detail. Mit  $p = 50\%$  ist ein Neuron beim Testen durchschnittlich mit doppelt so vielen Eingabeneuronen wie beim Trainieren verbunden. Um diesen Umstand zu kompensieren, müssen wir die Gewichte der Eingaben aller Neuronen nach dem Trainieren mit 0,5 multiplizieren. Andernfalls erhält jedes Neuron ein etwa doppelt so großes Eingabesignal und wird vermutlich keine gute Leistung erbringen. Allgemein müssen wir das Gewicht jeder Eingabeverbindung nach dem Trainieren mit der *keep-Wahrscheinlichkeit* ( $1 - p$ ) multiplizieren. Alternativ können wir auch die Ausgabe jedes Neurons beim Trainieren durch die keep-Wahrscheinlichkeit teilen (diese beiden Alternativen sind nicht exakt äquivalent, funktionieren aber gleich gut).

Um Drop-out mit Keras zu implementieren, können Sie die Schicht `keras.layers.Dropout` verwenden. Beim Trainieren lässt sie zufällig einige Neuronen aus (setzt diese auf 0) und teilt die verbliebenen Eingaben durch die keep-Wahrscheinlichkeit. Nach dem Trainieren tut diese Funktion überhaupt nichts, sie gibt die Eingaben einfach an die nächste Schicht weiter. Der folgende Code wendet die Regularisierung mittels Drop-out vor jeder Dense-Schicht an und nutzt dabei eine Drop-out-Rate von 0,2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```



Da der Drop-out nur während des Trainings aktiv ist, kann ein Vergleich von Trainings- und Validierungsverlust in die Irre führen. Insbesondere kann ein Modell den Trainingsdatensatz overfitten und trotzdem gleiche Verluste bei Training und Validierung besitzen. Stellen Sie also sicher, dass Sie den Trainingsverlust ohne Drop-out bestimmen (zum Beispiel nach dem Training).

Wenn Sie Overfitting beobachten, können Sie die Drop-out-Rate erhöhen. Liegt dagegen ein Underfitting der Trainingsdaten vor, sollten Sie die Drop-out-Rate senken. Bei großen Schichten hilft das Erhöhen der Drop-out-Rate ebenfalls und bei kleinen Schichten das Verringern. Außerdem nutzen viele führende Architekturen Drop-out nach der letzten verborgenen Schicht, was Sie auch ausprobieren können, wenn ein vollständiger Drop-out zu stark ist.

Drop-out verlangsamt die Konvergenz erheblich, dafür ist das erhaltene Modell mit den richtigen Einstellungen in der Regel sehr viel besser. Der zusätzliche Zeit- und Vorbereitungsaufwand zahlt sich also grundsätzlich aus.



Wollen Sie ein selbstdnormalisierendes Netz, das auf der SELU-Aktivierungsfunktion basiert (wie weiter oben besprochen), regularisieren, sollten Sie *Alpha-Drop-out* verwenden: Dabei handelt es sich um eine Variante des Drop-out, die den Mittelwert und die Standardabweichung der Eingaben beibehält (sie wurde im gleichen Artikel wie SELU vorgestellt, da das reguläre Drop-out die Selbstdnormalisierung zerstören würden).

## Monte-Carlo-(MC)-Drop-out

Im Jahr 2016 hat ein Artikel (<https://homl.info/mcdropout>) von Yarin Gal und Zoubin Ghahramani<sup>25</sup> ein paar weitere Gründe für Drop-out geliefert:

- Der Artikel zeigte eine stabile Verbindung zwischen Drop-out-Netzwerken (also neuronalen Netzen mit einer Dropout-Schicht vor jeder Gewichtsschicht) und der Approximate Bayesian Inference auf, womit das Drop-out eine solide mathematische Begründung erhält.<sup>26</sup>
- Die Autoren haben eine leistungsfähige Technik namens *MC-Drop-out* vorgestellt, die die Leistung eines jeden trainierten Drop-out-Modells verbessern kann, ohne es neu trainieren oder überhaupt anpassen zu müssen, die Unsicherheit des Modells verbessert und zudem auch noch ausgesprochen einfach zu implementieren ist.

Wenn Ihnen das alles zu schön vorkommt, schauen Sie sich den folgenden Code an. Dabei handelt es sich um die vollständige Implementierung von MC-Drop-out, die das Drop-out-Modell verbessert, ohne es erneut zu trainieren:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                      for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

<sup>25</sup> Yarin Gal und Zoubin Ghahramani, »Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning«, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.

<sup>26</sup> Insbesondere zeigen die Autoren, dass das Trainieren eines Drop-out-Netzes mathematisch äquivalent zur Approximate Bayesian Inference in einem bestimmten Typ von Wahrscheinlichkeitsmodellen namens *Deep Gaussian Process* ist.

Wir machen nur 100 Vorhersagen über den Testdatensatz, setzen `training=True`, um sicherzustellen, dass die Dropout-Schicht aktiv ist, und stacken die Vorhersagen. Da der Drop-out aktiv ist, werden alle Vorhersagen anders sein. Denken Sie daran, dass `predict()` eine Matrix mit einer Zeile pro Instanz und einer Spalte pro Kategorie zurückgibt. Da es 10.000 Instanzen im Testdatensatz und 10 Kategorien gibt, hat diese Matrix die Form [10000, 10]. Wir stacken 100 dieser Matrizen, womit `y_probas` ein Array der Form [100, 10000, 10] wird. Nachdem wir den Mittelwert über die erste Dimension ermittelt haben (`axis=0`), bekommen wir `y_proba` – ein Array der Form [10000, 10] –, wie bei einer einzelnen Vorhersage. Das ist alles! Das Bilden des Mittelwerts über mehrere Vorhersagen mit aktivem Drop-out liefert uns eine Monte-Carlo-Schätzung, die im Allgemeinen zuverlässiger als das Ergebnis einer einzelnen Schätzung mit abgeschaltetem Drop-out ist. Schauen wir uns beispielweise die Vorhersage des Modells für die erste Instanz im Testdatensatz mit abgeschaltetem Drop-out an:

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99]],
```

Das Modell scheint sich sehr sicher zu sein, dass dieses Bild zu Kategorie 9 (Ankle Boot) gehört. Sollten Sie dem vertrauen? Sind Zweifel angebracht? Vergleichen Sie das mit den Vorhersagen bei aktivem Drop-out:

```
>>> np.round(y_probas[:, :1], 2)
array([[[0. , 0. , 0. , 0. , 0. , 0.14, 0. , 0.17, 0. , 0.68]],
       [[0. , 0. , 0. , 0. , 0. , 0.16, 0. , 0.2 , 0. , 0.64]],
       [[0. , 0. , 0. , 0. , 0. , 0.02, 0. , 0.01, 0. , 0.97]]],
```

Das sieht schon anders aus: Anscheinend ist sich das Modell bei aktiviertem Drop-out nicht mehr so sicher. Es scheint immer noch Kategorie 9 zu bevorzugen, kann sich manchmal aber auch die Kategorie 5 (Sandal) oder 7 (Sneaker) vorstellen, was nicht so abwegig ist – es gehört alles zur Fußbekleidung. Bilden wir den Mittelwert über die erste Dimension, erhalten wir die folgenden MC-Drop-out-Vorhersagen:

```
>>> np.round(y_proba[:, :1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.22, 0. , 0.16, 0. , 0.62]],
```

Das Modell geht immer noch davon aus, dass dieses Bild zu Kategorie 9 gehört, aber nur mit einer Sicherheit von 62 %, was deutlich vernünftiger klingt als 99 %. Zudem ist es gut, zu wissen, welche anderen Kategorien noch in Betracht kämen. Sie können auch einen Blick auf die Standardabweichung der Wahrscheinlichkeits-schätzungen (<https://xkcd.com/2110>) werfen:

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:, :1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.28, 0. , 0.21, 0.02, 0.32]],
```

Es scheint eine ziemliche Varianz in den Wahrscheinlichkeitsschätzungen zu geben: Würden Sie ein System in einem Bereich mit echten Risiken bauen (zum Beispiel ein medizinisches oder Finanzsystem), sollten Sie solch einer unsicheren Voraussage besser mit außerordentlicher Vorsicht begegnen – und auf keinen Fall wie eine Voraussage mit 99%. Zudem hat die Modellgenauigkeit einen kleinen Schub von 86,8% auf 86,9% erhalten:

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)
>>> accuracy
0.8694
```



Die Anzahl der genutzten Monte-Carlo-Samples (in diesem Beispiel 100) ist ein Hyperparameter, den Sie anpassen können. Je höher er ist, desto genauer werden die Vorhersagen und deren Unsicherheitsschätzungen. Aber wenn Sie den Wert verdoppeln, wird sich auch die Inferenzzeit verdoppeln. Und über einer gewissen Anzahl von Samples werden Sie nur wenig weitere Verbesserung erreichen. Ihre Aufgabe ist es daher, abhängig von Ihrer Anwendung das richtige Verhältnis zwischen Latenz und Genauigkeit zu ermitteln.

Enthält Ihr Modell andere Schichten, die sich während des Trainings besonders verhalten (wie zum Beispiel BatchNormalization-Schichten), sollten Sie den Trainingsmodus nicht wie von uns durchgeführt erzwingen. Stattdessen sollten Sie die Dropout-Schichten durch die folgende MCDropout-Klasse ersetzen:<sup>27</sup>

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

Dabei erstellen wir einfach eine Unterklasse der Dropout-Schicht und überschreiben die Methode `call()`, um ihr Argument `training` fest auf `True` zu setzen (siehe Kapitel 12). Genauso könnten Sie eine Klasse `MCAphaDropout` erzeugen, indem Sie eine Unterklasse von `AlphaDropout` anlegen. Erstellen Sie ein Modell von Grund auf neu, müssen Sie nur `MCDropout` statt `Dropout` verwenden. Haben Sie aber bereits ein Modell, das mit `Dropout` trainiert wurde, müssen Sie ein neues Modell erstellen, das mit dem bestehenden identisch ist mit Ausnahme der Dropout-Schichten, die nun zu `MCDropout`-Schichten werden. Dann kopieren Sie die Gewichte des bestehenden Modells in Ihr neues Modell.

Kurz gesagt, ist MC-Drop-out eine fantastische Technik, die Drop,out-Modelle besser macht und bessere Unsicherheitsvoraussagen ermöglicht. Und da es während des Trainings ein ganz normales Drop,out ist, verhält es sich auch wie ein Regularisierer.

---

<sup>27</sup> Diese Klasse `MCDropout` arbeitet mit allen Keras-APIs zusammen, einschließlich der Sequential API. Nutzen Sie nur die Functional API oder die Subclassing API, müssen Sie keine Klasse `MCDropout` erzeugen – erstellen Sie einfach eine normale Dropout-Schicht und rufen Sie diese mit `training=True` auf.

## Max-Norm-Regularisierung

Eine weitere Regularisierungstechnik, die bei neuronalen Netzen gerne zur Anwendung kommt, nennt sich *Max-Norm-Regularisierung*: Für jedes Neuron werden die Gewichte  $w$  der eingehenden Verbindungen so beschränkt, dass  $\|w\|_2 \leq r$ , wobei  $r$  der Max-Norm-Hyperparameter und  $\|w\|_2$  die  $\ell_2$ -Norm ist.

Die Max-Norm-Regularisierung fügt der Gesamt-Verlustfunktion keinen Regularisierungsverlust hinzu. Stattdessen wird sie meist implementiert, indem  $\|w\|_2$  nach jedem Trainingsschritt berechnet und  $w$  bei Bedarf neu skaliert wird ( $w \leftarrow w \frac{r}{\|w\|_2}$ ).

Ein Verringern von  $r$  verringert die Regularisierungsstärke und hilft dabei, ein Overfitting zu reduzieren. Die Max-Norm-Regularisierung kann zudem dazu beitragen, die Probleme mit instabilen Gradienten zu dämpfen (wenn Sie keine Batchnormalisierung einsetzen).

Um die Max-Norm-Regularisierung in Keras zu implementieren, setzen Sie das Argument `kernel_constraint` jeder verborgenen Schicht auf einen Constraint `max_norm()` mit dem passenden Max-Wert, zum Beispiel so:

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

Nach jeder Trainingsiteration wird die Methode `fit()` des Modells das von `max_norm()` zurückgegebene Objekt aufrufen, dabei die Gewichte der Schicht übergeben und die umskalierten Gewichte zurückerhalten, die dann die Gewichte der Schicht ersetzen. Wie Sie in Kapitel 12 sehen werden, können Sie bei Bedarf Ihre eigenen Constraint-Funktionen definieren und als `kernel_constraint` einsetzen. Auch können Sie die Bias-Terme beschränken, indem Sie das Argument `bias_constraint` setzen.

Die Funktion `max_norm()` besitzt ein Argument `axis`, das standardmäßig den Wert 0 hat. Eine Dense-Schicht besitzt im Allgemeinen Gewichte der Form [Anzahl Eingaben, Anzahl Neuronen], daher sorgt `axis=0` dafür, dass der Max-Norm-Constraint unabhängig auf den Gewichtsvektor jedes Neurons angewendet wird. Wollen Sie Max-Norm mit Convolutional-Schichten verwenden (siehe Kapitel 14), achten Sie darauf, das Argument `axis` des Constraints `max_norm()` passend zu setzen (normalerweise `axis=[0, 1, 2]`).

## Zusammenfassung und praktische Tipps

In diesem Kapitel haben wir eine Vielzahl an Techniken besprochen. Sie fragen sich vielleicht, welche davon Sie verwenden sollen. Das hängt von der Aufgabe ab, und es gibt daher noch keinen klaren Konsens, aber ich habe festgestellt, dass die in Tabelle 11-3 aufgeführte Konfiguration in den meisten Fällen gut funktioniert, ohne dass allzu viele Hyperparameter angepasst werden müssen. Betrachten Sie diese Standardwerte jedoch nicht als feste Vorgaben!

Tabelle 11-3: Standardkonfiguration eines DNN

Hyperparameter	Standardwert
Kernelinitialisierer	Initialisierung nach He
Aktivierungsfunktion	ELU
Normalisierung	keine, wenn Shallow; Batchnormalisierung, wenn Deep
Regularisierung	Early Stopping (+ $\ell_2$ -Regularisierung bei Bedarf)
Optimierer	Momentum-Optimierer (oder RMSProp oder Nadam)
Scheduler für die Lernrate	1cycle

Handelt es sich beim Netz um eine simple Aufeinanderfolge von dichten Schichten, kann es sich selbst normalisieren, und Sie sollten stattdessen die Konfiguration in Tabelle 11-4 verwenden.

Tabelle 11-4: DNN-Konfiguration für ein selbstdnormalisierendes Netz

Hyperparameter	Standardwert
Kernelinitialisierer	LeCun-Initialisierung
Aktivierungsfunktion	SELU
Normalisierung	keine (Selbstdnormalisierung)
Regularisierung	Alpha-Drop-out bei Bedarf
Optimierer	Momentum-Optimierer (oder RMSProp oder Nadam)
Scheduler für die Lernrate	1cycle

Vergessen Sie nicht, die Eingangsmerkmale zu normalisieren! Sie sollten auch versuchen, Teile eines vortrainierten neuronalen Netzes zu verwenden, wenn Sie eines finden, das ein ähnliches Problem löst, oder unüberwachtes Vortraining zu nutzen, wenn Sie viele ungelabelte Daten haben, oder ein Vortraining für eine künstliche Aufgabe einzusetzen, wenn Sie viele gelabelte Daten für eine ähnliche Aufgabe besitzen.

Während die obigen Richtlinien die meisten Fälle abdecken sollten, gibt es ein paar Ausnahmen:

- Wenn Sie ein spärliches Modell benötigen, können Sie  $\ell_1$ -Regularisierung hinzufügen (und nach dem Trainieren eventuell sehr kleine Gewichte auf null setzen). Wenn Sie ein noch spärlicheres Modell benötigen, können Sie das TensorFlow Model Optimization Toolkit einsetzen. Das wird die Selbstdnormalisierung zerstören, daher sollten Sie in diesem Fall die Standardkonfiguration verwenden.
- Benötigen Sie ein Modell mit geringer Latenz (also eines, das blitzschnell Vorphersagen trifft), müssen Sie eventuell die Anzahl der Schichten reduzieren, die Batchnormalisierungsschichten mit den vorherigen Schichten verbinden und eventuell eine schnellere Aktivierungsfunktion wie Leaky ReLU oder einfach

ReLU einsetzen. Es hilft auch, um ein spärliches Modell zu nutzen. Und schließlich können Sie die Gleitkommagenaugigkeit von 32 Bit auf 16 oder sogar 8 Bit verringern (siehe »Ein Modell auf ein Mobile oder Embedded Device deplo- yen«). Schauen Sie sich außerdem TF-MOT an.

- Bauen Sie eine Anwendung in einem riskanteren Umfeld oder ist die Inferenzlatenz für Sie nicht sehr wichtig, können Sie MC-Drop-out zum Verbessern der Performance einsetzen, um zuverlässigere Wahrscheinlichkeitsschätzungen und Unsicherheitsschätzungen zu erhalten.

Mit diesen Richtlinien sind Sie bereit, sehr tiefe Netze zu trainieren! Ich hoffe, Sie sind jetzt davon überzeugt, dass Sie mit Keras ziemlich weit kommen. Es kann aber trotzdem passieren, dass Sie noch mehr Kontrolle benötigen – zum Beispiel wenn Sie eine eigene Verlustfunktion schreiben oder den Trainingsalgorithmus anpassen wollen. Für solche Fälle werden Sie auf die auf niedrigerer Ebene arbeitende API von TensorFlow zurückgreifen müssen, die Sie im nächsten Kapitel kennenlernen.